Tallinna Ülikool
Matemaatika-Loodusteaduskond
Informaatika osakond

Priit Valdmees

Ekstreemprogrammeerimine: ülevaade, praktikad ja võrdlus teiste väledate arendusmeetoditega

Bakalaureusetöö

Juhendaja: Jaagup Kippar

Autor: ..... „.....“ 2007
Juhendaja: ..... „.....“ 2007
Õppetooli juhataja: ..... „.....“ 2007

Tallinn 2007

Sisukord

SISSEJUHATUS **5**

1 Sissejuhatus ekstreemprogrammeerimisse **6**

1.1 Ajalugu 6

1.2 Päritolu 6

1.3 Hetkeseis 7

1.4 Eesmärgid 7

1.5 Väärtused 8

1.5.1 Suhtlus *8*

1.5.2 Lihtsus *9*

1.5.3 Tagasiside *9*

1.5.4 Julgus *10*

1.5.5 Tunnustus *10*

1.6 Põhimõtted 11

2 Arendusprotsess **12**

2.1 Rollid 12

3 Tegevus **14**

3.1 Kodeerimine 14

3.2 Testimine 14

3.3 Kuulamine 15

3.4 Disainimine 15

4 Praktikad **17**

4.1 Paarisprogrammeerimine 18

4.1.1 Kasud *18*

4.1.2 Kriitika *18*

4.2 Plaanimismäng 19

4.2.1 Redaktsioonide planeerimine *20*

4.2.2 Iteratsioonide planeerimine *21*

4.3 Väikesed redaktsioonid 23

4.4 Metafoor 23

4.5 Lihtne disain 24

4.6 Kollektiivne omand 24

4.7 Pidev integreerimine 24

4.8 Ühtne kodeerimisstandard 25

4.9 Meeskonnatöö 25
4.10 Sobiv tempo 25
4.11 Disaini täiustamine 25
4.12 Test-juhitud arendus 26
5 Testimisprotsess 27
5.1 Automaattestid 27
5.1.1 Lihtsustab muudatuste tegemist 27
5.1.2 Lihtsustab integreerimist 27
5.1.3 Dokumentatsioon 28
5.1.4 Kasutajaliidese eristamine rakendusest 28
5.1.5 Automaattestide piirangud 29
5.1.6 Automaattestimise rakendused 29
5.1.7 Tehnika 29
5.2 Integratsioonitestid 30
5.2.1 Integratsioonitestide eesmärk 30
5.3 Süsteemitestid 31
5.3.1 Terve süsteemi testimine 31
5.4 Sobivustestid 32
5.4.1 Protsess 32
6 Vastuolulised aspektid 33
6.1 Ebastabiilsed nõuded 33
6.2 Kasutajate konfliktid 33
6.3 Teised aspektid 33
6.4 Mõõdetavus 34
6.5 Vaidlus raamatus 34
6.6 Ekstreemprogrammeerimise evolutsioon 35
6.7 Ühendatud metoodikad 35
7 Erinevate väledate arendusmeetodite võrdlus 36
7.1 Rationali unifitseeritud arendusprotsess 36
7.2 Erisus-juhitud arendus 37
7.3 Adaptiivne tarkvaraarendus 37
7.4 Dünaamiline süsteemiarendusmeetod 37
7.5 Crystal Clear 38
7.6 Scrum 39

7.7 Kokkuvõte võrdlusest 39
8 Ekstreemprogrammeerimise teooria ja praktilise projekti võrdlus 42
8.1 Diagrammid 42
8.2 Kasutajalood 42
8.3 Redaktsioonid 42
8.4 Iteratsioonid 43
8.5 Disainimine 43
8.6 Funktsionaalsus 43
8.7 Rekodeerimine 43
8.8 Kliendi kohalolek 44
8.9 Kokkulepitud standardid 44
8.10 Paarisprogrammeerimine 44
8.11 Pidev integreerimine 44
8.12 Kollektiivne omand 44
8.13 Testimine 45
8.14 Kokkuvõte võrdlusest 45
Kokkuvõte 46
Viited 47
Summary 49

SISSEJUHATUS

Tarkvara roll on muutunud aastatega inimeste seas oluliselt. Nii väga kui me seda ka ei tahaks uskuda – me oleme täielikult sõltuvuses tarkvarast. Tarkvara puudumine muudaks kasutuks kõik arvutid ja riistvara meie ümber. Tarkvara on midagi, millega oleme juba harjunud ning milleta jääks meie elu seisma. Kuna aga ühiskond on pidevalt arenemas, siis peab ka tarkvaratööstus sellega kaasas käima – arendades järjest uusi meetmeid ning lahendusi.

Erinevaid metoodikaid on tekkinud palju ja nende hulgas orienteerumine ning selle õige valimine muutub järjest keerulisemaks. Lisaks valimisele võib osutuda metoodika teoreetiline pool eksitav. Tihtipeale on teoorias töötav metoodika osutunud reaalses lähenemises suhteliselt kasutuks, kuna teooria ning praktika on teineteisest lahku läinud.

Tarkvaratehnika valdkonnas on hakanud viimastel aastatel populaarsust koguma erinevad väledate (agile) arendusmeetodite esindajad.

Kuna väledaid arendusmeetodeid on mitmeid, siis sai teemaks valitud ekstreemprogrammeerimine, tema pidevalt kasvavale populaarsusele arendajate seas, põneva nime ning ka autori enda huvi pärast.

Eesmärgiks on luua põgus ülevaade ekstreemprogrammeerimisest, selle praktikatest, testimisest ning võrdlus teiste väledate metoodikatega. Jõuda järeldusele, kas ekstreemprogrammeerimine väärib oma kõrget populaarsust tarkvaraarenduse maastikel.

Töö esimeses osas on antud üldine ülevaade ekstreemprogrammeerimisest, selle põhimõttest, praktikatest ning testimisest. Töö teises osas on toodud esile ka teised väledate arendusmeetodite tehnikad, antud ülevaade ning võrreldud neid nii omavahel kui ka ekstreemprogrammeerimisega, et selgitada head ja halvad küljed. Lisaks veel võrdlus teooria ning praktilise projekti vahel, selgitamaks, kas kirjas olev vastab reaalsele projektis kasutatavale metoodikale.

1 Sissejuhatus ekstreemprogrammeerimisse

1.1 Ajalugu

Ekstreemprogrammeerimine (Extreme programming – XP) sai alguse Kent Becki, Ward Cunninghami ja Ron Jeffriesi poolt ajal, mil nad töötasid Chrysler Comprehensive Compensation System (C3) palgaarvestussüsteemi projekti kallal. Kent Beck määrati C3-e projekti juhiks 1996. aasta märtsis ning ta hakkas selles kasutatud arendusmetoodikat viimistlema. Beck kirjutas kasutatud metoodikast raamatu ning 1999. aastal ilmus „*Extreme Programming Explaned*" [Beck 1999]. Chrysler lõpetas C3 projekti 2000. aasta veebruaris, kuid metoodika jäi silma sel ajal tarkvara-arenduse maastikel. 2006. aasta seisuga on ekstreemprogrammeerimise metoodika laialt kasutatav kogu maailmas [C2].

1.2 Päritolu

90. aastate tarkvaraarendust kujundasid põhiliselt kaks suuremat mõju: sisemiselt, objekt-orienteeritud programmeerimine vahetas välja protseduurse programmeerimise; välimiselt, interneti areng ja veebilehekülgede järjest kasvav populaarsus hakkasid mängima firmade arengus tähtsat rolli. Kiirelt muutuvad nõuded hakkasid tahtma lühemaid elutsükleid ning tihti jäädi väga kaugele traditsioonilistest tarkvaraarenduse metoodikatest.

Chrysler Comprehensive Compensation projekt algatati, kasutades palgaarvestussüsteemi kui uurimusalust objekti ja Smalltalki programmeerimiskeelt, et selgitada parim viis, kuidas kasutada objekt-tehnoloogiat. Kohale kutsuti Kent Beck, lootustandev Smalltalki praktikant, et ta optimiseeriks süsteemi koodi. Aga Becki rolli suurendati, kui ta leidis mõningaid küsitlevaid kohti arenduse protsessis. Ta kasutas võimalust ning lisas muudatusi, mida ta oli eelnevalt kasutanud oma varasemas töös koos Ward Cunninghamiga. Beck kutsus projekti ka Ron Jeffriese, aitamaks arendada ja rafineerida uusi meetodeid.

Ekstreemprogrammeerimise põhimõtteid ja praktikaid levitati laiale maailmale tol ajal läbi arutelu Cunninghami WikiWikiWebis [C2].

Ekstreemprogrammeerimise mõistet on seletatud lahti mitmeid aastaid, kasutades XP koduleheküIge [XP 2] ning seal olevaid diagramme. Erinevad kaasaaitajad arutasid ja laiendasid ideid ja tulemuseks olid mõningad metoodika kõrvalsaadused.

Beck toimetas ka mitme ekstreemprogrammeerimise teemat käsitleva raamatu kallal, alustades enda „Extreme Programming Explaned *(1999),* millega levitas oma ideid palju suuremale ning ka vastuvõtlikumale publikumile. Raamatus oli käsitletud erinevaid ekstreemprogrammeerimise aspekte ning praktikaid.

1.3 Hetkeseis

Ekstreemprogrammeerimine tekitas päris palju ärevust 90. aastate lõpus ja 2000. aasta algusaegadel ning selle praegune kasutatavus erineb radikaalselt sellest, mis see alguses oli. Väledate arendusmeetodite praktikad ei ole muutumatud olnud – ekstreemprogrammeerimine areneb jätkuvalt, omandades aina enam rakendusi erinevate kogemuste teel. *„Extreme Programming Explaned”* raamatu teises väljalaskes lisas Beck uusi väärtusi ning praktikaid, eristamaks peamiseid ning järeldatud praktikaid [XP 2].

1.4 Eesmärgid

„Extreme Programming Explained” kirjeldab ekstreemprogrammeerimist kui [XP 2]:

- Püüd ühendada inimlikkust ja tootlikkust
- Sotsiaalse muutuse mehhanism
- Teekond arenemise suunas
- Arenduse stiil
- Tarkvara arendamise viis

Ekstreemprogrammeerimise üks eesmärke, lisaks kvaliteetse ning nõuetekohase tarkvara loomisele, on vähendada muudatuste maksumust. Traditsioonilistes süsteemiarendusmeetodites fikseeritakse arendusprojekti nõuded projekti alguses ning hiljem neid muuta enam ei saa. Seega, hilisemad muudatuste tegemised projektis muutuvad kulukaks. Ekstreemprogrammeerimine üritab vähendada muudatuste kulutusi, tutvustades elementaarseid väärtusi, põhimõtteid ning praktikaid. Ekstreemprogrammeerimise puhul on süsteemi arendus paindlikum muudatuste suhtes [Beck 1999].

1.5 Väärtused

Ekstreemprogrammeerimises tunti algselt nelja väärtust. Uus, viies väärtus, lisati „*Extreme Programming Explained"* raamatu teises väljalaskes. Need viis väärtust oleksid järgmised:

- Suhtlemine
- Lihtsus
- Tagasiside
- Julgus
- Tunnustus (viimasena lisatud)

1.5.1 Suhtlus

Arendusprotsessi efektiivsuse aluseks on suhtlus kõigi osapoolte vahel.

Tarkvarasüsteemide loomine nõuab suhtlemist, et süsteemi loojad teaksid, mida tulevane süsteem tegema peab. Tavapärasel tarkvaraarendusel kasutati selleks dokumentatsiooni. Ekstreemprogrammeerimises seevastu aga dokumentatsiooni üldjuhul ei looda. Seda muidugi ei tohiks sõna-sõnalt võtta – ekstreemprogrammeerimises on dokumentatsiooniks erinevad testid, normide kohaselt vormistatud lähtekood ning erinevad kommentaarid. Ekstreemprogrammeerimise tehnikat saab vaadelda kui tarkvara kiiret loomist ning arendusmeeskonnavahelist teadmiste ühtlast jagamise metoodikat. Eesmärk on anda kõikidele arendajatele ühine vaade süsteemist nii, nagu seda omavad tulevase süsteemi kasutajad.

Seega, ekstreemprogrammeerimine pooldab lihtsat disaini, tavalisi metafoore, koostööd kasutajate ja programmeerijatega, tihedat verbaalset kommunikatsiooni ning tagasisidet [Beck 1999].

1.5.2 Lihtsus

Ekstreemprogrammeerimine julgustab kasutama lihtsamaid lahendusi, et hiljem oleks neid kergem täiustada. Erinevus sellise lähenemise ja tavakohase süsteemiarendusmeetodiga on see, et keskendutakse disainile ja koodile, et see rahuldaks hetke nõudeid, mitte tulevasi. Ekstreemprogrammeerimise pooldajad leiavad, et see on puudus, mis toob kaasa suuremaid jõupingutusi hiljem, kui vaja teha uusi muudatusi süsteemis. Koodi kirjutamine ja disainimine hilisematele muutustele mõeldes, tekitab riski, et raisatakse ressursse millegi peale, mida ei lähe võib-olla kunagi tarvis.

Viidates suhtlemise mõistele, siis lihtsustamine disainis ning koodis endas peaks tõstma suhtlemise kvaliteeti. Lihtne disain ning lihtne kood peaks olema arusaadav enamikele programmeerijatele mingis kindlas meeskonnas [Beck 1999].

1.5.3 Tagasiside

Adekvaatne tagasiside aitab tarkvara täiustada.

Ekstreemprogrammeerimise tagasiside viitab süsteemiarenduse erinevatele dimensioonidele:

- Tagasiside süsteemilt: Kasutades automaatteste või perioodilisi integreerimisteste, saavad programmeerijad otsest tagasisidet süsteemi olekust kohe pärast muutuste sisse viimist.
- Tagasiside kliendilt: Funktsionaalsuse testid (ehk sobivuse testid) on valmistatud kliendi ja testijaga kooskõlas. Need annavad kindlat tagasisidet süsteemi kindlast hetkeseisust. Ülevaade on planeeritud iga kahe või kolme nädala tagant, seega saab klient kergelt suunata arenduse käiku.

- Tagasiside tiimilt: Kui klient pakub välja uusi nõudeid plaanimismängu, siis arendusmeeskond annab talle umbkaudse aja palju sellise nõude realiseerimine aega võiks võtta [Beck 1999].

Tagasiside on tihedalt seotud suhtluse ja lihtsustamisega. Vead süsteemis on kerged välja tulema, kui kirjutada automaatteste, mis tõestavad, et kindel osa koodist ei toimi. Otsene tagasiside süsteemilt annab programmeerijale teada, millist osa koodist tuleks ümber kirjutada või täiustada. Klient saab süsteemi testida perioodiliselt, vastavalt programmi funktsionaalsuse nõuetele.

1.5.4 Julgus

Mõningased praktikad väljendavad julgust. Üks sellistest on käsk, et alati kirjutada koodi vastavalt tänastele nõuetele, mitte homsetele. Selline tegevus hoiab ära koodi liiga suureks kasvamise, mistõttu on vaja palju suuremaid jõupingutusi uute lisade lisamisel. Julgus lubab arendajal tunda end vabamalt koodi ümberloomisel. Teine näide julgusest on teadmine, et on vaja kood nn „minema visata" – vana koodi asendamine täiesti uuega: julgus eemaldada ebavajalikku algkoodi, ükskõik kui palju vaeva selle valmistamiseks ka ei läinud. Lisaks, julgus tähendab püsivust : programmeerija võib ühe probleemi juurde jääda terveks päevaks, ning lahendab selle alles järgmine päeval [Beck 1999].

1.5.5 Tunnustus

Tunnustuse väärtus avaldub mitmel viisil: meeskonna liikmed tunnustavad teineteist, kuna programmeerijaid ei tohiks kunagi teha muudatusi, mis takistaks kompilatsiooni, mille tõttu automaattestid nurjuksid, või muul moel venitaksid oma kaastöötajate tööga. Liikmed tunnustavad tööd, püüdes alati leida parimaid disainimislahendusi ning kõrget töö-kvaliteeti [Beck 1999].

1.6 Põhimõtted

Ekstreemprogrammeerimise peamised põhimõtted baseeruvad eelnevalt kirjeldatud väärtustel ning on mõeldud selleks, et süsteemiarendusel kerkiksid üles erinevad valikud. Põhimõtted on mõeldud olema palju konkreetsemad kui väärtused ning samas ka lihtsamad selgitama praktilisi olukordi.

Tagasiside on efektiivne, kui seda tehakse tihti. Tegevuse ja tema tagasiside vaheline aeg on uurimiseks ning muudatuste tegemiseks oluline. Erinevalt traditsioonilistest süsteemiarendusmeetoditest toimub ekstreemprogrammeerimises side kliendi vahel väikeste iteratsioonide kaudu. Klienil on siis selge ülevaade arendatavast süsteemist. Klient saab anda ka tagasisidet ning arendusprotsessi mõjutada, kui vaja.

Automaattestid annavad suure panuse tagasiside põhimõttele. Kui kirjutada koodi, siis automaattest annab otsest tagasisidet sellest, kuidas süsteem reageerib muutustele. Näiteks, kui muutused mõjuvad süsteemi selles osas, mida too kindel programmeerija ei ole teinud, siis ei märkaks ta viga. Võimalik, et sellised vead tuleksid välja alles siis, kui süsteem on juba tootmises.

„Eeldades lihtsust" tähendab, et igat probleemi käsitletakse kui äärmiselt lihtsa lahendusega asja. Traditsioonilises süsteemiarenduse meetodis planeeritakse hilisematele muudatustele mõeldes. Ekstreemprogrammeerimine sellist ideed ei kasuta.

Ekstreemprogrammeerimise pooldajad ütlevad, et korraga kõiki suuri muudatusi teha pole hea mõte.

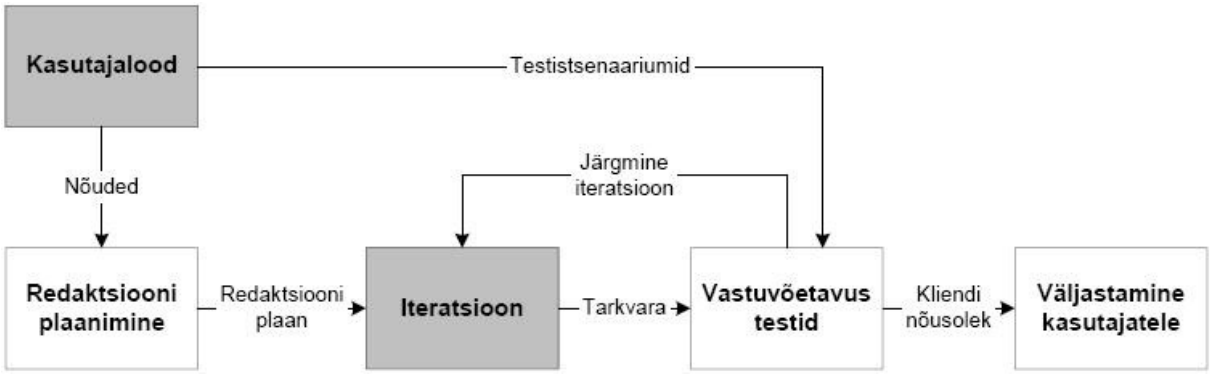
Ekstreemprogrammeerimine kasutab lisanduvaid muutusi: süsteemil võib olla iga kolme nädala tagant uus väljalase. Tehes palju väikeseid samme, saab klient rohkem kontrollida arenduse protsessi ning arendatavat süsteemi.

Muutuste omaksvõtu põhimõte seisneb selles, et ei tule olla muutuste vastu, vaid vastupidiselt hoopis võtta neid omaks. Näiteks, kui tuleb iteratsioonimiitingul välja, et kliendi nõuded on rohkesti muutunud, siis programmeerija võtab selle omaks ja planeerib uued nõuded järgmiseks iteratsiooniks [Informit].

2 Arendusprotsess

Ekstreemprogrammeerimise arendusprotsess (Joonis 1) koosneb üksteisele järgnevatest redaktsioonidest.

Iga redaktsiooni järel viiakse tarkvara üle üldisesse töökeskkonda. Esimene redaktsioon kestab teistest kauem, kuna protsess nullist kuni esimese töökõlbuliku versioonini võtab rohkem aega. Üldjuhul lõpeb projekt plaanitud ajal, olenemata, kas tellijal on veel soove. Arendajate jaoks on ootel järgmised projektid ning seega ei saa projektiga tegeleda seni, kuni klient soovib. Klient võib esitada uued soovid tarkvara muutmiseks või edasiarenduseks, kuid seda võib lugeda juba uue projektina ning sellega tegeletakse siis, kui selle kord saabub. Iga redaktsioon koosneb hulgast iteratsioonidest. Redaktsiooni pikkus on tavaliselt 1..3 kuud, üks iteratsioon kestab 1..3 nädalat. Esimene, teistest pikem redaktsioon kestab enamasti 2..6 kuud [Beck 1999].



Joonis 1. Ekstreemprogrammeerimise lihtsustatud arendusprotsess [XP 2]

Iga redaktsioon algab selle planeerimisega, mille käigus klient määrab teostatavad kasutajalood. Iga iteratsioon algab planeerimisega, kus valitakse hulk lugusid välja selles teostamiseks [XP 2].

2.1 Rollid

XP-metoodikas on olulised ka rollid, mida arendajad peavad täitma. Kokku on määratud kaheksa rolli [Beck 2000].

1. Programmeerija (*programmer*), kes kirjutab testid ja realiseerib rakenduse, on põhiline roll ekstreemprogrammeerimises.

2. Kasutaja (*customer*) kirjutab soovilugusid. Soovilugude vahendusel annab kasutaja programmeerijale teada, mida on vaja teha.

3. Testija (*tester*) aitab kasutajal teha sobivusteste. Lisaks hoolitseb automaatsete testide käivitamise eest.

4. Jäljekütt (*tracker*) jälgib arendustegevusega seotud numbrilisi väärtusi: mitu ideaalpäeva on kulunud mingi ülesande realiseerimiseks; mitu ülesannet käesolevas nädalas on realiseeritud; mitu ülesannet on vaja veel realiseerida, et püsida graafikus jne.

5. Treener (*coach*) on vastutav kogu arendusprotsessi eest. Treener otsustab, kas arendusmeeskond on graafikus (ning teeb õiget asja) või mitte ja viib ellu muudatused selleks, et meeskond järjele saada.

6. Konsultant (*consultant*) tegeleb tehniliste eriküsimustega ja annab meeskonna liikmetele eelkõige tehnilist konsultatsiooni.

7. Suur juht (*Big Boss*) on meeskonna juht ja varustab meeskonda vajalike ressurssidega. Peab omama projektist üldist pilti, olema kursis projekti seisuga. Üldjuhul arendustegevusega ei tegele.

8. Vahemees (*customer proxy*) on kasutaja asemik, kui tegu on väga suuri inimestehulki hõlmava tarkvaraga (tihtilugu tootejuht).

3 Tegevus

Ekstreemprogrammeerimine kirjeldab nelja peamist tegevust, mida tehakse tarkvaraarenduse protsessi juures: kodeerimine, testimine, kuulamine, disainimine [Beck 1999].

3.1 Kodeerimine

Ekstreemprogrammeerimise pooldajad arvavad, et ainuke tõeline süsteemiarenduse protsessi produkt on kood. Ilma koodita ei ole midagi.

Koodiks võivad olla joonistatud diagrammid, mis genereerivad koodi, veebipõhiste süsteemide skriptid või kompileerimist vajav kood.

Koodi saab kasutada ka selleks, et leida sobivamaid lahendusi. Näiteks, ekstreemprogrammeerimise puhul võib tulla ette, et ühe probleemi lahendamiseks on mitu erinevat võimalust. Et välja selgitada, milline lahendus oleks parim, selleks saab lihtsalt kõik variandid valmis kirjutada ning automaattestidega välja selgitada parim. Siiski pole mõistlik suuremate moodulite puhul kõiki variatsioone valmistada, kuna seda võib lugeda liigseks ajakuluks.

Koodiga saab ka anda edasi mõtteid seoses programmeerimisega. Kui programmeerija tegeleb keerulise programmeerimise probleemiga ja tal on raske selle lahendust seletada kaasprogrammeerijale, siis ta võib selle koodina välja kirjutada ning näidata demonstratsiooni teel, mida ta mõtleb. Kood ise on alati selge ning seda ei saa mõista rohkem kui ühel viisil. Teised programmeerijad saavad anda tagasisidet sellest koodist, kasutades oma mõtete edastajana sammuti koodi [Beck 1999].

3.2 Testimine

Ei saa olla milleski kindel enne, kui pole seda testitud. Kuigi testimine pole kohustuslik, on seda siiski vaja kliendile, kinnitamaks, et kõik toimib nii, nagu vaja. Palju tarkvara on lastud käiku ilma korraliku eelneva testimiseta, kuid tarkvara on ikkagi toiminud.
Ekstreemprogrammeerimine väidab, et ei saa olla kindel, et funktsioon töötab enne kui seda pole testitud. See tõstatab küsimuse, et milles me siis kindlad olla ei saa:

- Võib olla ebakindlust selles, et kood, mida kirjutati, ei tee seda mida see peaks. Et seda ebakindlust testida, kasutatakse automaatteste, mis testivad koodi ennast. Programmeerija kirjutab võimalikult palju teste, mis kõik proovivad koodi „murda". Kui kõik testid läbitakse edukalt, siis on kood valmis.

- Võib olla ebakindlust selles, et mõtlesid vale asja. Selle ebakindluse testimiseks kasutatakse ekstreemprogrammeerimises nn. sobivuse teste, mis baseeruvad kliendi nõuetel ja mis on saadud väljalaske planeerimise avastamise faasist. [XP 1]

3.3 Kuulamine

Programmeerijad ei pea teadma midagi arendatava süsteemi ettevõtte töövaldkonnast. Süsteemi funktsioon tuleb ettevõtte poolt, kellele süsteemi luuakse. Selleks, et programmeerija saaks teada süsteemi funktsionaalsusest, peab ta kuulama ettevõtte esindajat.

Programmeerijad peavad kuulama klienti ja tema soove. Lisaks, nad peavad üritama mõista ärilisi probleeme ja lahendusi ning andma kliendile tagasisidet probleemist, seletades, milles viga seisneb.

Programmeerija ja kliendi vahelist suhtlust kirjeldatakse kui plaanimismängu [Beck 1999].

3.4 Disainimine

Lihtsustamise nurga alt vaadatuna võiks öelda, et süsteemiarendus ei vaja muud kui koodi, testimist ning osapoolte ärakuulamist. Kui need tingimused on täidetud, siis peaks tulemuseks alati olema töötav süsteem. Tegelikult aga see nii päris ei toimi. Alati saab kaugele jõuda ilma disainimiseta, kuid mingi hetk muutub see ikkagi vajalikuks. Süsteem muutub liiga keeruliseks ning süsteemiosade omavaheline sõltuvus ebaselgeks.

Seda saab vältida, luues teatud disainistruktuur, mis organiseerib süsteemi loogikat. Hea disain hoiab ära paljud sõltuvused süsteemis; see tähendab, et muutes üht osa süsteemist, ei mõjuta see teisi osasid [Beck 1999].

4 Praktikad

Ekstreemprogrammeerimises on 12 erinevat praktikat, mis on grupeeritud nelja kategooriasse[Beck 1999]:

Tagasiside

- Paarisprogrammeerimine
- Plaanimismäng
- Test-juhitud arendus
- Meeskonnatöö

Katkematu protsess

- Pidev integreerimine
- Disaini täiustamine
- Väikesed redaktsioonid

Jagatud arusaam

- Ühtne kodeerimisstandard
- Kollektiivne omand
- Lihtne disain
- Metafoor

Programmeerija heaolu

- Sobiv tempo

4.1 Paarisprogrammeerimine

Paarisprogrammeerimine vajab kahte arendajat, kes kombineeritult teevad tööd ühe arvuti taga. Kumbki neist teeb mingit tegevust, mida teine ei tee: kui üks kirjutab automaattesti, siis teine mõtleb, millise klassi jaoks seda kasutada. Seega, üks on „sõitja" ja teine „kaardilugeja". Iga natukese aja tagant vahetatakse rollid [PairProgramming].

4.1.1 Kasud

Paarisprogrammeerimine peaks tooma järgmised kasud:

- Tõusev distsipliin : Paarilised teevad rohkem õigeid asju ning kasutavad vähem pause.
- Parem kood : Paarilised teevad vähem „halba" koodi.
- Mitme arendaja panus disaini : Kui paare vahetatakse tihti, siis on rohkem arendajaid kaasatud arendamaks mingit kindlat osa.
- Kõrgendatud moraal : Paarisprogrammeerimine võib olla palju nauditavam kui seda üksi tehes.
- Kollektiivne omand : Kui terves projektis kasutatakse paarisprogrammeerimist ning paarid vahetuvad pidevalt, siis saavad kõik aru kogu koodibaasist.
- Õpetlik : Paarisprogrammeerimises saab alati üks pool teisele midagi uut õpetada.
- Meeskonna sidusus : Paarisprogrammeerimise puhul saab meeskond omavahel kiiremini tuttavaks, kui üksi töötades.
- Vähem vahelesegamisi : Inimesed segavad harvemini paaris töötavaid inimesi kui üksi töötavat inimest [Methods&Tools].

Uurimused väidavad, et kaks programmeerijat on ühest rohkem kui kaks korda produktiivsemad [The Economist].

4.1.2 Kriitika

- Kogenud arendajad leiavad, et paarisprogrammeerimise puhul on tülikas hakata õpetama vähem kogenenumat programmeerijat .
- Paljud eelistavad üksi töötamist ning arvavad, et paaris töötamine on kohmakas.
- Erinevused koodi stiilis võivad viia konfliktini.
- On raske võrrelda, kumb on tootlikum moodus.

4.2 Plaanimismäng

Plaanimismäng on tagasiside praktika.

Ekstreemprogrammeerimise peamist plaanimisprotsessi nimetatakse plaanimismänguks.

Plaanimise protsess on jagatud kaheks :

Redaktsioonide planeerimine: See on vajalik välja selgitamaks, milliseid nõuded on lisatud millistele reaktsioonidele ja millal need käiku lastakse. Klient ja arendaja planeerivad seda koos. Redaktsioonide planeerimine koosneb kolmest faasist :

- Avastamise etapp : Selles etapis annab klient kõik oma süsteeminõuded. Nendest kirjutatakse kasutajaloo kaardid.
- Pühendamise etapp : Pühendamise etapis klient ning arendaja pühenduvad funktsionaalsuse leidmisele ning kuupäeva selgitamine järgmiseks väljalaskeks.
- Juhtimise etapp : Juhtimise etapis saab esialgset plaani muuta, uusi nõudeid lisada ning vanu eemaldada või muuta.

Iteratsioonide planeerimine : See planeerib arendajate tegemisi ning ülesandeid. Selles osas üldjuhul ei kaasata klienti ennast, kuigi väidetavalt on kliendi kaasamine andnud häid tulemusi. Iteratsioonide planeerimine ise koosneb kolmest osast :

- Uurimise etapp : Selles etapis on kõiksugused nõuded tõlgendatud ümber erinevateks ülesanneteks.
- Pühendamise etapp : Eelmises etapis olnud ülesanded antakse programmeerijatele ning arvutatakse nende lahendamiseks kuluv umbkaudne aeg.

- Juhtimise etapp : Ülesanded lahendatakse ning lõpp-resultaati võrreldakse kasutajalugudega [Joseph Bergin].

4.2.1 Redaktsioonide planeerimine

Avastamise etapp

See on iteratsiooniline nõuete kogumise ning nõuete ja töö omavahelise mõju arvestamise protsess.

- Kliendi nõuete saamine : Ettevõte esitab oma probleemi; koosolekul üritatakse määratleda probleem ning selle nõuded.
- Kasutajaloo kirjutamine : Kasutajalugu kirjutatakse kliendi poolt vastavalt ettevõtte vajadusele. Selles mainitakse, mida peab mingi kindel süsteemiosa tegema. On oluline, et arendusmeeskond ei mõjutaks kliendil selle kirjutamisel.
- Kasutajaloo poolitamine : Kui arendusmeeskond ei suuda kasutajalugu hinnata, siis tuleb see poolitada ning uuesti kirjutada. Jällegi ei tohi mõjutada kliendi nõudeid.
- Kasutajaloo hindamine : Arendusmeeskond hindab kaudselt, kui kaua võib aega võtta kasutajaloo realiseerimine tööks [Adaption].

Kui klient ei suuda enam uusi nõudeid välja mõelda, siis minnakse edasi järgmisesse etappi - pühendamine.

Pühendamise etapp

Selles etapis selgitatakse välja hind, kasud ning ajakava. See sisaldab endas nelja komponenti:

- Sorteerimine hindamise järgi : Klient sorteerib kasutajalugusid hindamise järgi.
- Sorteerimine riski järgi : Arendajad sorteerivad kasutajalugusid riski järgi.
- Kiiruse määramine : Arendajad selgitavad, kui kiiresti annaks seda projekti teha.
- Käsitlusala valimine : Valitakse kasutajalood järgmiseks redaktsiooniks. Vastavalt kasutajaloole määratakse ka redaktsiooni valmimisaeg.

Sorteerimine hindamise järgi

Klient sorteerib kasutajalood vastavalt ettevõtte hindamisele. Nad jagavad kasutajalood kolme kuhja :

- Kriitilised : Kasutajalood, milleta süsteem ei saa toimida või kaotab oma mõtte.
- Oluline väärtus : Mitte-kriitilised kasutajalood, millel on oluline väärtus.
- Head, et olemas on : Kasutajalood, mis ei oma olulist väärtust – näiteks täiustus kasutatavuses või esitamises.

Sorteerimine riski järgi

Arendajad sorteerivad kasutajalugusid riski järgi. Nad jagavad sammuti kõik kasutajalood kolme kuhja : madal, keskmine ja kõrge risk. Seejärel hakatakse kasutajalugusid hindama erinevates kategooriates[Adaption].

Juhtimise etapp

Juhtimise etapis saavad nii arendajad kui ettevõtte esindajad juhtida protsessi käiku. Erinevad kasutajalood ning suhtelised eesmärgid võivad muutuda hindamise käigus. Nüüd on võimalus muuta plaani vastavalt nendele [Joseph Bergin].

4.2.2 Iteratsioonide planeerimine

Iteratsioonide planeerimine jaguneb samamoodi kolmeks etapiks : Avastamise, pühendamise ning juhtimise etapp.

Avastamise etapp

Avastamise etapp on enamjaolt erinevate ülesannete loomine ning nende realiseerimiseks kuluva aja hindamine.

- Kasutajalugude kogumine : Järgmiseks väljalaskeks mõeldud kasutajalugude kirjutamine ja kokkukogumine.
- Ülesannete ühendamine/poolitamine : Kui programmeerija ei suuda hinnata ülesannet, kuna see on liiga suur või väike, siis peab ta ülesandeid ühendama või poolitama.
- Ülesannete hindamine : Ülesande realiseerimiseks kuluva aja hindamine.

Pühendamise etapp

Iteratsioonide planeerimise pühendamises etapis antakse programmeerijatele ülesandeid vastavalt kasutajalugudele.

- Programmeerija nõustub ülesandega : Iga programmeerija valib ühe ülesande mille eest ta suudab vastutada.
- Programmeerija hindab ülesannet : Kuna programmeerija on nüüd vastutav ülesande eest, siis annab ta omapoolset edaspidist hinnangut selle konkreetse ülesande kohta.
- Koormusfaktori määramine : Koormusfaktor näitab ideaalset aega, kui kaua peaks minema ühel programmeerijal ühe iteratsiooni peale.
- Tasakaalustamine : Kui meeskonnas on kõigile programmeerijatele jagatud ülesanded, hakatakse võrdlema ülesannete sooritamise ennustatavat aega ning koormusfaktorit. Sellest lähtuvalt tasakaalustatakse ülesanded kõikide programmeerijate vahel ära [Joseph Bergin].

Juhtimise etapp

Juhtimise etapis toimub ülesannete lisamine süsteemi.

- Ülesande kaart : Programmeerija võtab ühe ülesande millele ta eelnevalt on pühendunud.

- Otsi partner : Programmeerija hakkab seda ülesannet realiseerima koos teise programmeerijaga.
- Disaini ülesanne : Vajadusel võib programmeerija disainida ülesande funktsionaalse poole.
- Kirjuta automaatteste : Enne, kui programmeerijad hakkavad funktsionaalselt poolt kirjutama, teevad nad automaatteste.
- Koodi kirjutamine : Programmeerijad alustavad koodi kirjutamisega.
- Testimine : Automaattestid testivad koodi

4.3 Väikesed redaktsioonid

Iga redaktsiooni lõpus antakse tarkvara üle lõppkasutajatele ning see läheb reaalsetes töötingimustes kasutusse. Redaktsioonid peavad olema võimalikult lühikesed, et arendajad saaksid vahetumat tagasisidet ning kasutajad saaksid kiiresti kõige olulisemaid osasid süsteemist kasutama hakata. Kuigi redaktsioonid on lühikesed, peab iga redaktsiooni tulemus olema ka äriliselt mõttekas ning looma kasutajatele reaalset väärtust. Seetõttu võtab tavaliselt esimese redaktsiooni loomine rohkem aega kui järgnevad. Lühikesed redaktsioonid teevad ka plaanimise lihtsamaks, kuna muutuvate nõuete tõttu on raske pikalt ette näha tegelikke vajadusi [XP 1].

4.4 Metafoor

Süsteemi ehitust kirjeldatakse lihtsa metafooriga, millest kõik aru saavad – nii arendajad kui ka klient. Metafoor täidab arhitektuuri-kirjelduse rolli, millega kannab edasi tarkvara üldist tööpõhimõtet. Metafoor annab tiimile ka ühtse sõnavara, mida kasutada omavaheliseks suhtlemiseks. Parim metafoor on lihtne selgitus [Seeba 2002].

4.5 Lihtne disain

Tarkvara disain peab olema nii lihtne kui võimalik. Parim disain on minimaalne, mis läbib kõik testid ning milles pole dubleerivat loogikat. Kunagi ei üritata ette näha tulevasi nõudmisi ning ennustada, mida võiks vaja minna homme. Kuna muutuste tegemine on sama kallis igas projekti faasis, jõuab tulevikus vajaminevaid asju ka hiljem lisada ning nendele ei pea tarkvara disainides mõtlema. Nii lähenedes ei tehta liigset tööd ning tarkvara saab valmis kiiremini [XP 1].

4.6 Kollektiivne omand

Koodi omamise all mõistetakse seda, et ainult omanik võib tema omanduses olevat koodi muuta. Näiteks võib igal failil, klassil või arhitektuurilisel kihil olla individuaalne omanik. See tähendab, et kui keegi vajab oma töö tegemiseks muutust koodis, mis ei kuulu talle, peab ta paluma omanikul muutuse sisse viia. Sellise omandivormi eelis seisneb selles, et kõik programmeerijad ei pea tundma kogu koodi. Ekstreemprogrammeerimine eelistab programmikoodi kollektiivset omandit. Iga programmeerija võib muuta kogu koodi, kui tal parasjagu selleks vajadus tekib. See tagab, et kogu tiimil on olemas ülevaade kogu süsteemi toimimisest ning iga programmeerijate paar saab tegutseda teiste järel ootamata [XP 1].

4.7 Pidev integreerimine

Tarkvara integreeritakse ja testitakse peale iga muutuse sisseviimist ning see peab toimuma vähemalt üks kord päevas. Sellega on garanteeritud integreerimis-probleemide varajane avastamine ja seeläbi integreerimisega seotud riskide kiirem maandamine. Peale iga muudatuse integreerimist käivitatakse uuesti kõik seni loodud automaattestid. Integreeritud tarkvara peab läbima kõik testid. Kui see ei õnnestu, siis tuleb muudatuse tegijail vead kõrvaldada [MF].

4.8 Ühtne kodeerimisstandard

Kõik programmeerijad järgivad ühtset koodi kirjutamise standardit. Standard hõlmab muuseas viisi, kuidas kirjutatakse kommentaare, tähistatakse muutujaid, kirjutatakse meetodeid jne. Kuna ekstreemprogrammeerimine kasutab koodi kollektiivset omandit, siis pole mõeldav, et igaüks kirjutab omas stiilis. Ühtne kodeerimisstandard muudab koodi kõigile arusaadavaks ja muutuste tegemise lihtsamaks [XP Exchange].

4.9 Meeskonnatöö

Loodava süsteemi tulevane kasutaja peab olema tiimiga samas ruumis ja pidevalt kättesaadav kõigile arendajatele. Tema ülesandeks on vastata küsimustele, lahendada vaidlusi ning määrata prioriteete. Reaalne kasutaja annab arendajatele väärtuslikku infot, mis aitab luua täpsemini tegelikele vajadustele vastava tarkvara. Selle praktika rakendamise teeb raskeks see, et kliendil on süsteemi tulevast kasutajat vaja ka põhitöö tegemiseks. Kasutaja saatmine pikaks ajaks arendustiimi juurde on seetõttu problemaatiline ja kulukas [XP 2].

4.10 Sobiv tempo

Programmeerijad peavad olema värsked ja puhanud igal hommikul ning võimelised lahendama probleeme loominguliselt. Kestev ületöötamine seda ei soodusta. Pidevad ületunnid on tavaliselt märgiks muudest, tõsisematest probleemidest, mida ei saa lahendada pelgalt ületundidega. Kaks nädalat järjest pole ekstreemprogrammeerimise projektis lubatud ületunde teha, kuna piisav puhkus on oluline eeldus arendajate töövõime säilimiseks [Mayford].

4.11 Disaini täiustamine

Disaini täiustamine on programmi ümberstruktureerimine eemaldades kordusi, lihtsustades ja lisades paindlikkust nii, et süsteemi funktsionaalsus säilib. See on ekstreemprogrammeerimise

viis tarkvara projekteerida ja see toimub pidevalt kogu projekti jooksul. Rekodeerimine võib küll tähendada rohkemat tööd, kuid tagab disaini arusaadavuse ning teeb edasiarendamise lihtsamaks [XP 2].

4.12 Test-juhitud arendus

Iga programmi omaduse jaoks on olemas automaattestid. Automaattest on programm, mis testib teise programmi tööd. Selle eelis käsitsi testimise ees seisneb selles, et automaatteste saab samamoodi käivitada korduvalt ning testimine toimub oluliselt kiiremini. Iga kasutajaloo realiseerimine algab automaattestide kirjutamisest. Seejärel alles kirjutatakse programm, mis neid teste rahuldab. Ka kliendid kirjutavad teste kasutajalugude verifitseerimiseks (*acceptance tests*) – need ei ole automaattestid, vaid pigem testistsenaariumid, mida järgides saab tuvastada, kas loodud programm rahuldab kasutajaloo nõudmisi [C2].

5 Testimisprotsess

Testimine on vägagi tähtis osa ekstreemprogrammeerimisest. Pidev rekodeerimine, suur hulk iteratsioone ning redaktsioone pidevalt nõuavad testimist, et süsteem kokku oleks terviklik.

5.1 Automaattestid

Automaattestide (*Unit testing*) eesmärk on isoleerida osa programmist ning näidata, et individuaalsed tükid programmist toimivad. Automaattestidega kirjutatakse ette karmid nõuded, mida komponent peab täitma. Selle tulemusel aga saab ülejäänud programm omale lisaväärtusi [XP 2].

5.1.1 Lihtsustab muudatuste tegemist

Automaattestide kasutamine lubab programmeerijal hiljem koodi uuesti luua ning kannab hoolt selle eest, et moodulid töötaksid. Kasu seisneb selles, et testimine julgustab programmeerijat muudatuste tegemisel ning on lihtne kontrollida, kas mingi osa koodist toimib korralikult. Head automaattestid suudavad käsitleda kogu kontrollitavat moodulit või komponenti ning suunavad põhirõhu kordamistele [CodeP].

Pidevas automaattestimise keskkonnas ning läbi jätkuvate muudatuste praktikas näitavad automaattestid koodi plaanitud kasutust muudatuste suhtes.

5.1.2 Lihtsustab integreerimist

Automaattestimine aitab kaotada ebakindlust komponentides ning saab kasutada ka „alt-üles" (bottom-up) testimise stiilis lähenemist, mida saab võtta kui individuaalsete väikeste programmiosade (moodulite) ühendamist omavahel suurema programmiosa saavutamiseks.

Testides programmiosasid kõigepealt ja seejärel programmiosade ühendamisest tekkinud tervikut, muutub integratsioontestimine lihtsamaks.

Palju on vaieldud ka manuaalse integratsioonitestimise teemal. Kuna väljatöödeldud automaattestide hierarhia on saavutanud integratsioonitestide taseme, võib tekitada see valestimõistmise, sest integratsioontestimine hindab ka teisi eesmärke, mida saab tõestada ainult läbi inimfaktori. Mõned vaidlevad jällegi selle üle, et andes automaattestide süsteemile piisavalt valikuid, muutub integratsioonitestimises inimfaktor mittevajalikuks. Realistlikult vaadeldes oleneb tegelik nõue lõpuks siiski projekti tunnustest ning selle mõeldud kasutusvaldkonnast [Brett G.Palmer].

5.1.3 Dokumentatsioon

Automaattestid on nagu „elav dokumentatsioon". Klient ja arendajad saavad mooduli kasutamisvõimalustest ja rakenduse arendusliidesest õppida vaadates automaatteste.

Automaattestide juhtumid väljendavad tunnuseid, mis on kriitilise tähtsusega komponendi valmissaamise edukuses. Need tunnused võivad näidata komponendi õiget või valet kasutust. Kuigi paljud tarkvaraarenduse keskkonnad ei toetu arendatava produkti automaatsele dokumenteerimisele, siis automaattestid ise dokumenteerivad kriitilistest tunnustest.

5.1.4 Kasutajaliidese eristamine rakendusest

Kuna mõned klassid viitavad teistele, siis testides mingit klassi juhtub teinekord nii, et testitakse valet asja. Näide sellest, kui testitakse klasse, mis sõltuvad andmebaasidest: testimiseks kirjutab testija tihtipeale koodi, mis suhtleb andmebaasi endaga. See on viga, kuna klass ei tohiks kunagi väljuda iseenda piiridest. Arendaja peab hoopis tekitama ühtse liidese andmebaasi ümber ning lisama sellele oma „mock-objekti", mis on simulatsioon testitavast objektist. Selline teguviis vähendab süsteemiosade omavahelist sõltuvust [IBM].

5.1.5 Automaattestide piirangud

Automaattestid ei suuda leida kõiki programmis olevaid vigu. See testib ainult kindlate komponentide funktsionaalsust. Seega ei suuda leida integratsioonivigu, jõudluse probleeme või muid ülesüsteemilisi küsimusi.

Automaattest näitab ainult vigade olemasolu; see ei suuda näidata vigade puudumist. Need kaks piirangut kehtivad kõigil tänapäeva tarkvaratestimistel [CF].

5.1.6 Automaattestimise rakendused

Ekstreemprogrammeerimine toetub automatiseeritud automaattestide võrgustikul. See võrgustik võib olla loodud projekti arendusmeeskonna poolt või siis kolmandate isikute, näiteks xUnit automaatestimise raamistiku, poolt.

Ekstreemprogrammeerimine kasutab tekitatud automaatteste testitud arenduseks. Arendaja kirjutab automaattesti, mis paljastab kas mõne programmi nõude või defekti. Test kukub läbi, kui mõnda vajalikku osa ei ole veel lisatud programmi või leiab koodist defektse koha. Seejärel hakkab arendaja koodi muutma seni, kuni testid saadakse sooritatud.

Testitakse kõiki klasse süsteemis. Arendaja väljastab automaattesti koodi samaaegselt selle koodiga, mida ta testib. Ekstreemprogrammeerimise automaattestid lubavad kõiki eelnimetatud eeliseid: lihtsam ja kindlam koodiarendus, lihtsustatud integreerimine ja täpne automaatselt genereeritud dokumentatsioon.

5.1.7 Tehnika

Tavapärase ja üldiselt aktsepteeritud tööstuspraktikana viiakse automaattestimist läbi automatiseeritud keskkonnas kasutades kolmanda osapoole poolt pakutud komponenti või võrgustikku. Kuigi on palju erinevaid standardeid, siis IEEE (Institute of Electrical and Electronics Engineers, Inc.) ei kirjuta ette kumbagi lähenemist – automaatset ega manuaalset.

Manuaalne lähenemine automaattestimisele võib sisaldada endas samm-sammult instruktsioone dokumendina. Sellegipoolest on automaattestimise eesmärk isoleerida komponent ja valideerida selle õigsus. Automaatsus on väga efektiivne sellist seisu saavutama ning annab juurdepääsu kõikidele nendele kasudele, millest juttu on olnud. Kui aga ei plaanita hoolikalt, siis ettevaatamatu manuaalne test võib toimida kui integratsioonitest [Andy Glover].

Et tõeliselt mõista isolatsiooni tähtsust, käivitatakse automaatsel lähenemisel komponent või koodi osa testimisvõrgustikus väljaspool loomulikku keskkonda, ehk teisisõnu, väljaspool rakendust, mille jaoks ta loodi. Testimine isolatsioonis näitab mittevajalikku koodi ja muude komponentide vahelist sõltuvust.

Kasutades automaattestide raamistikku, kodeerib arendaja automaattestide kriteeriumid, mille alusel saab komponentide õigsust kontrollida. Testimisel võrgustik märgib üles kõik juhtumid, kui midagi kukkus läbi mõne kriteeriumi. Paljud võrgustikud ka märgistavad ära ning tekitavad kokkuvõtte testi mitteläbinud juhtumitest. Paljude läbikukkumiste korral peatab võrgustik edasise testimise [Andy Glover].

5.2 Integratsioonitestid

Integratsioonitestimine (mõnikord nimetatud ka Integratsioon ja Testimine, I&T) on see tarkvara testimise faas, kus individuaalsed tarkvaramoodulid kombineeritakse ja testitakse kui ühist suurt gruppi. Integratsioonitestimine järgneb peale automaattestimist ning enne süsteemi testimist.

Integratsioonitestimine võtab oma sisendiks moodulid, mida on eelnevalt automaattestitud, grupeerib nad suuremaks kogumiks, realiseerib nendele testid vastavalt integratsioonitestimise kavale ning väljastab väljundi integreeritud süsteemina, mis on valmis süsteemtestimiseks [Matt Albrecht].

5.2.1 Integratsioonitestide eesmärk

Integratsioonitestide eesmärk on veenduda süsteemi funktsionaalsuses, toimimises ning usaldusväärsuse vajadustest üldistes disainiosades. Neid disainiosasid või komponentide gruppe käivitatakse läbi nende endi kasutajaliidese Black box testidega. Black box on testimine, mille puhul ei teata, mis toimub programmi sees - testid kirjutatakse puhtalt nõuete järgi: millise sisendi korral peab olema väljund. Õigete parameetrite ning andmesisenditega simuleeritakse töötavaid vigaseid juhtumeid. Kõik need testimised on vajalikud, et veenduda kõikide komponentide omavahelises grupisiseses toimimises [Methods&Tools 2].

5.3 Süsteemitestid

Süsteemiteste tehakse valmis ehk integreeritud süsteemidel, et hinnata süsteemi võimalusi tema algsetest ettekirjutustest. Testid enamjaolt jäävad Black box testimise käsitlusalasse ning ei vaja arusaama programmi sisemisest disainist, koodist ega loogikast.

Reeglina võtab süsteemitest sisendiks kõik integreeritud tarkvarakomponendid, mis on läbinud integratsioonitestid. Süsteemitestimine üritab leida defekte moodulites ja süsteemis kui tervikus [Rex Black].

5.3.1 Terve süsteemi testimine

Süsteemitestimine tehakse kogu süsteemile vastavalt funktsionaalsete nõuete spetsifikatsiooni (*Functional Requierment Specificatio, FRS*) ja süsteeminõuete spetsifikatsiooni *(System Requierment Specificatio, SRS*) ettekirjutustele. Lisaks on süsteemitestimine uurimuslik testimise faas, kus keskendutakse mitte ainult disaini, vaid ka süsteemi käitumise ning kokkusobivusele kliendi kirjeldustega [Rex Black].

Süsteemitestimist võib lugeda viimaseks lõhkuvaks testimise faasiks enne sobivusetesti.

5.4 Sobivustestid

Sobivusteste viiakse läbi koos kasutajate või sponsoritega, kasutades black box testimist. Tulemusest selgub antud süsteemi sobivus.

Sobivustestid on üldiselt komplekt erinevaid teste, mida hakatakse kasutama valminud süsteemis. Iga test testib mingit kindlat süsteemi osa ning väljastab tulemuse. Tulemus võib olla kas edukas või läbikukkunud. Testimiskeskkond on tehtud võimalikult sarnane sellega, mis võiks olla kasutajal. Need testid peavad kasutama sisendiks test-andmeid või siis tegevuse kirjeldust, mille abil teste läbi viia [XP 2].

5.4.1 Protsess

Süsteemi testitakse kliendilt saadud andmetega ning selle vastuseid võrreldakse oodatud tulemustega. Test on sooritatud, kui igal alal on vastuseks see, mida oodati. Kui vastus pole see, mida oodati, siis süsteem võetakse vastu või lükatakse tagasi vastavalt kokkulepitud tingimustele arendaja ja kliendi vahel.

Eesmärk on kindlustada, et süsteem toimiks vastavalt kliendi nõuetele. Sobivustestide otstarve on kontrollida, et süsteem läbiks testi edukalt - siis on süsteem lõplikult valmis [XP 2].

6 Vastuolulised aspektid

Kõige suurem vastuolu tekitav aspekt ekstreemprogrammeerimise juures on protsessi muutuste juhtimise aspekt. Traditsioonilise tarkvaaraarenduse protsessid vajavad muudatuste taotluse analüüsi ning muudatuste tegemise lubamist vastava juhatuse poolt. Ekstreemprogrammeerimises seevastu aga küsib kohapeal olev klient mitteametlikult muudatuste tegemist – tihti vaid suuliselt arendusmeeskonda teavitades.

6.1 Ebastabiilsed nõuded

Ekstreemprogrammeerimise pooldajad väidavad, et kui klient nõuab mitteametlikult muudatuste tegemist, siis kogu protsess muutub paindlikumaks ning hoiab kokku üldkulutusi. Need muudatused kõik hinnatakse mingis kindlas punktisüsteemis, ning klient meeskonna liikmena peab ise nimetama ka vastavas mahus funktsionaalsusi, millest ta selles konkreetses projekti skoobis loobub. Ekstreemprogrammeerimise vastased aga leiavad, et see tähendab ainult pidevat ümbertegemist ning projekti käsitlusala väljub nendest piiridest, mis algselt oli kokku lepitud ning rahastatud [Matt Stephens].

6.2 Kasutajate konfliktid

Kuna kõik saavad koodis muudatusi teha, siis võib esineda probleeme eesmärkide arusaamises. Ekstreemprogrammeerimises oodatud metoodika on mõnevõrra sõltuv programmeerija võimest eeldada kliendi ühtset seisukohta, et saaks keskenduda rohkem koodi kirjutamisele kui dokumentatsioonile. See kehtib ka siis, kui kaasatud on mitmed programmeerimisega tegelevad organisatsioonid [Matt Stephens].

6.3 Teised aspektid

Teised ekstreemprogrammeerimise vastuolulised aspektid oleksid [Matt Stephens] :

- Nõuded on pigem väljendatud automatiseeritud vastuvõtutestidena kui spetsiifiliste dokumentidena.
- Nõuded on kirjeldatud lisanduvalt, mitte kõik korraga.
- Tarkvaraarendajad peavad töötama paaridena.
- Disainimist ei tehta suuresti ette ära. Enamus disainist tehakse jooksvalt lisadena, alustades kõige lihtsamast töötavast osast ning lisades keerulisemaid, kui testid seda nõuavad. Kriitikud kardavad, et selline tegevus nõuab hoopis rohkem muudatusi kui disainimine vastavalt nõuete muutumisele.
- Kliendi esindaja on seotud projektiga. Sellest rollist võib saada projekti läbikukkumise koht ning paljud on väitnud, et see on väga stressitekitav positsioon.

6.4 Mõõdetavus

Ekstreemprogrammeerimise vastased väidavad, et ekstreemprogrammeerimine toimib vaid 12 või vähemarvulistes tiimides, kuigi on väidetud, et ekstreemprogrammeerimine on olnud ka edukas üle 100 liikmelistes tiimides [Matt Stephens].

6.5 Vaidlus raamatus

2003. aastal avaldasid Matt Stephens ja Doug Rosenberg raamatu „*Programming Refactored: The Case Against XP",* mis pani kahtluse alla ekstreemprogrammeerimise põhimõtted, ning pakkusid välja viise, kuidas seda paremaks muuta. See käivitas aga ulatusliku vaidluse erinevates artiklites, interneti uudisegruppides ning jututubades. Raamatu peamine argument oli see, et ekstreemprogrammeerimise praktikad on iseseisvad, kuid osad organisatsioonid ei ole võimelised omaks võtma kõiki praktikaid; seega kogu protsess kukub läbi. Raamat sarnastab ekstreemprogrammeerimise „kollektiivset omandit" ja sotsialismi. Kõike seda siis negatiivsel viisil [Steve Yegge].

6.6 Ekstreemprogrammeerimise evolutsioon

Peale „*Extreme Programming Refactored”* ilmumist 2003. aastal on kindlad ekstreemprogrammeerimise aspektid muutunud: Näiteks, ekstreemprogrammeerimises kohandatakse muudatusi praktikate järgi täpselt nii kaua, kuni vajalikud eesmärgid on saavutatud. Ekstreemprogrammeerimises kasutatakse ka protsesside jaoks üldiseid tingimusi. Mõned väidavad, et need muudatused lükkavad ümber eelnevat kriitikat, teised jällegi arvavad, et need mõned täiustused ei ole veel piisavad [Steve Yegge].

6.7 Ühendatud metoodikad

Paljud on üritanud ühendada ekstreemprogrammeerimist mõne vanema metoodikaga, et luua ühte üldist metoodikat. Mõned neist oleksid võimelised isegi asendama ekstreemprogrammeerimist, näiteks: kaskaad- e. koskmudel [Matt Stephens].

JPMorgan Chase ja Co nimeline ettevõte üritas ühendada ekstreemprogrammeerimist teiste programmeerimise metoodikate, nagu CMMI (Capability Maturity Model Integration) ja Six Sigma. Nad leidsid, et need kolm süsteemi sobisid omavahel hästi ning muutsid süsteemiarendust natuke paremaks.

Ekstreemprogrammeerimine ei ole ainult vastuolusid tekitav metoodika, kuna tänu temale tekib pidevalt ka vaidlusi ning kriitikat teistele tarkvaraarenduse meetoditele [Steve Yegge].

7 Erinevate väledate arendusmeetodite võrdlus

Lisaks ekstreemprogrammeerimisele on veel ka teisigi väledaid arendusmeetodeid. Käesolevas peatükis loob autor kiire ülevaate ning võrdleb neid omavahel, kui ka ekstreemprogrammeerimise endaga.

Katsealusteks meetoditeks said autori arvates, lisaks ekstreemprogrammeerimisele, kuus üldlevinumat väledat arendusmetoodikat: rationali unifitseeritud arendusprotsess, erisus-juhitud arendusprotsess, adaptiivne tarkvaraarendus, dünaamiline süsteemiarendusmeetod, crystal clear ning scrum.

7.1 Rationali unifitseeritud arendusprotsess

Rationali unifitseeritud arendusprotsess (*Rational Unified Process*, RUP) on iteratiivne tarkvara arendusprotsessi raamistik, mis loodi Rational Corporationi poolt, kes alates 2002 aastast kuulub IBM alla. Iteratiivne arendus sellepärast, et siis on parem tegeleda erinevate probleemidega sammhaaval ja liikuda edasi väikeste iteratsioonide kaupa. Eesmärgiks on vähendada varakult projekti riske. Mudelite loomisel kasutatakse UML modelleerimiskeelt. Rationali unifitseeritud arendusprotsessis kasutatakse, sarnaselt ekstreemprogrammeerimisega, erinevaid praktikaid, mis arendusprotsessis on üsna suure osakaaluga ning väga detailselt ära määratud [RUP].

Viimasel ajal on rationali unifitseeritud arendusprotsess just rõhku pannud väikestele projektidele ning väledatele protsessidele. Kaasatud on erinevate väledate metoodikate praktikad ning juhtnöörid nende kasutamiseks.

7.2 Erisus-juhitud arendus

Erisus-juhitud arendus (*feature driven development*, FDD) on sarnaselt rationali unifitseeritud arendusprotsessile iteratiivne ja eelkõige suunatud tulemustele. Erisuse all mõeldakse loodava süsteemi omadust – mida süsteem tegema peab. See peaks olema kliendile teada ning selle kasutegur ka eelnevalt väljaselgitatud. See võimaldab arendajatel jõuda kiiresti tulemustele kvaliteeti oluliselt kaotamata. Keskendutakse pigem inimestele ning dokumentatsioon on jäetud tahaplaanile. Mõeldud rohkem väikestele tiimidele aga sobib ka suurematele [FDD].

Ühe sellise erisuse arendus peaks maksimaalselt võtma aega kaks nädalat, keskmiselt aga paarist tunnist mõne päevani.

Erisus-juhitud arendus on aga vaid analüüsi kirjeldamiseks, ülejäänud osa jaoks tuleb kasutada lisaks mõnda muud metoodikat.

7.3 Adaptiivne tarkvaraarendus

Adaptiivne tarkvaraarendus (*adaptive software development*, ASD) võtab tarkvara loomist kui midagi, mida ei saa ennustada, kus planeerimine on võimatu või siis väga keeruline. Hiljem on aga tehtud vigadest hea õppida ning seejärel proovida neid välistada.

Adaptiivne tarkvaraarendus seab ka tähtsale kohale pideva õppimise. Seda iseloomustab pidev muutumine ning ümberhindamine. Tähtsaks loetakse ka väga tihedat koostööd kõigi osapoolte vahel, kuna kohanduva lähenemise puhul on efektiivne tagasiside eriti oluline [ASD].

7.4 Dünaamiline süsteemiarendusmeetod

Dünaamiline süsteemiarendusmeetod (*Dynamic Systems Development Method***,** DSDM) on inkrementaalne ja väledatest metoodikatest üks põhjalikem.

Dünaamiline süsteemiarendusmeetod ütleb [DSDM], et igal projektil on kolm peamist näitajat: rahakulu, ajakulu ning funktsionaalsus. Põhimõte seisneb selles, et projektile fikseeritakse sellele vajaminev aeg ning ressursid ning lastakse funktsionaalsusel kohanduda.

Protsessijuhtimise, reaalajasüsteemide ja ülikriitiliste tarkvarade puhul on dünaamilise süsteemiarendusmeetodi rakendamine raskem, kuna dünaamilise süsteemiarendusmeetodi puhul on oluline iteratiivne arendus ning kasutajate kaasamine projekti. Samas on ka raskendatud põhjaliku spetsifikatsiooni loomine ning selle põhjal tarkvara valideerimine.

Tiimid on üldjuhul väikesed (2..6 inimest), et vältida juhtimis- ja kommunikatsiooni probleeme.

Dünaamilist süsteemiarendusmeetodit haldab DSDM konsortsium mille liikmetel on litsents selle metoodika kasutamiseks ning selle eest maksavad nad aastamaksu. Konsortsium samas ka koolitab, pakub kasutajatuge ning arendab metoodikat. Tänu sellele on dünaamiline arendusmeetod arenenud ning laialt levinud [DSDM].

7.5 Crystal Clear

Crystal clear tähtsustab eelkõige inimesi ja nendevahelist suhtlust. Sarnaselt erisus-juhitud arendusega, ei ole metoodikad mõeldud lõplikena. Arendusmeeskonnad peaksid võtma seda kui vundamendina ning sellest lähtuvalt looma endale sobivaim lähenemine[CC].

Eelistatakse inimestega suhtlemist näost-näkku ning üritatakse vältida paberdokumentide kasutamist. Samas crystal clear rõhutab ka tiimiliikmete vahelist lähedast suhtlust.

Projektid liigitatakse meeskonna suuruse ja tarkvara kriitilisuse järgi. Selle järgi saab igale projektile vaadata sobiva lähenemise. Meeskonna suurus on oluline metoodika valimisel. Üldjuhul on meeskonnad väga väikesed (4..6 inimest) ning üritatakse tegeleda vähekriitiliste projektidega, kus on olulisel kohal kiirus ja efektiivsus [CC].

7.6 Scrum

Scrum on üks väledatest arendusmetoodikatest, mis kõige enam üritab protsessi juhtida ja kontrollida. Tarkvaraarenduseks mingit kindlat tehnikat ei määrata, pigem jäetakse see arendajate teha. Tihti kasutatakse selles ekstreemprogrammeerimise meetodeid [Scrum].

Scrumi iseloomustavad tihedad koosolekud, kus meeskonna liikmed selgitavad, et mida nad eelmine päev teha jõudsid ning mis neil edasi päevakorras on. Koosolekutel räägitakse ka kõikidest probleemidest ja takistustest, mis siiani on tekkinud.

Arendusprotsess jaguneb 2..6 nädala pikkusteks iteratsioonideks. Iga iteratsiooni käigus disainitakse, kodeeritakse ja testitakse tarkvara. Iga iteratsiooni eel koostatakse nimekiri nõuete kohta. Neid hakkavad teostama 5..8-liikmelised meeskonnad [Scrum].

7.7 Kokkuvõte võrdlusest

Võrdlusesse kaasatud metoodikad erinevad üksteisest mitmeti Arenduses hõlmavad suure osa nii rationali unifitseeritud arendusprotsess ja dünaamiline süsteemiarendusmeetod. Seda siis alates lihtsamatest analüüsidest kuni hoolduseni välja.

Scrum ja erisus-juhitud arendus aga, vastupidiselt eelnevatele metoodikatele, ei haara nii suurt ala arendusprotsessist. Scrum kirjeldab iteratsioonide ja protsesside suunamist ning erisus-juhitud arendus tegeleb pigem nõuete kirjapaneku ja programmeerimisega.

Metoodikate omavahelist erinevust saab võrrelda ka nende detailsuse poolest. Ekstreemprogrammeerimine ja rationali unifitseeritud arendusprotsess on üsna kindlalt piiritlenud, et milliseid praktikaid peaks nendes meetodites kasutama. Vastanditeks on adaptiivne tarkvaraarendus ja crystal clear, millel antakse suhteliselt vabad käed metoodika valimiseks.

Erinevates olukordades on aga metoodikad üldjuhul suhteliselt piiritletud. Enamus käesolevatest väledatest arendusmeetoditest on eelkõige mõeldud suhteliselt väikeste tiimide jaoks, eriti ekstreemprogrammeerimine ja crystal clear, kes selle üsna karmilt paika pannud.

Rationali unifitseeritud arendusprotsess aga tiimi suurusele piiranguid ei sea ning laseb arendusmeeskonna juhil seda ise valida.

Oluline on ka loodava tarkvara kriitilisus, seda eelkõige dünaamilise süsteemiarendusmeetodi ja crystal cleari puhul, sest mõlemad meetodid annavad mõista, et kriitiliste projektide korral on nende metoodika kasutamine raskendatud.

Väledad metoodikad eelistavad iteratiivset arendusprotsessi kus iteratsiooni pikkustele pole kindlat piiri seatud. Erisus-juhitud arenduse ja ekstreemprogrammeerimise puhul on iteratsioonid väga lühikesed (päevadest kuni nädalateni) ning crystal clear ja rationali unifitseeritud arendusprotsess kasutavad pikemaid iteratsioone (kuudes). Lühemate iteratsioonidega suudetakse erinevaid riske kiiremini eristada ning kiirendatakse kasutajate vahelist tagasisidet. Parem tagasiside tuleb ka kliendi kaasamisel projekti. Seda nõutakse eriti ekstreemprogrammeerimise ja dünaamilise süsteemiarendusmeetodi puhul, kus kliendid on kaasatud tiimi liikmeteks.

Iga metoodika puhul on küllaltki sarnaseks osutunud nõuete kogumine ja nende rahuldamine. Need jaotatakse tükkideks mis hakkavad kirjeldavad süsteemi tulevasi omadusi.

Väledaid metoodikaid eristatakse ka tiimide ülesehituse järgi. Üldjuhul on tiimid kindlalt paika määratud ning ei muutu. Erandiks võib lugeda erisus-juhitud arendust, kus tiime pidevalt muudetakse. Erisus-juhitud arenduses muudetakse tiime iga uue erisuse arendamisel ning tihti kuuluvad arendajad mitmesse tiimi korraga. Üheks erandiks on ka ekstreemprogrammeerimise juures kasutatav paarisprogrammeerimise praktika, kus pidevalt vahetatakse partnereid. Selline tiimide muutmine soodustab erineva informatsiooni levimist liikmete vahel.

Erisus-juhitud arendus nõuab igale koodiosale eraldi omanikku, seevastu aga ekstreemprogrammeerimine eelistab kollektiivset omandit. Kollektiivse omandi puhul saavad konkreetset koodi muuta mitmed inimesed, mis kiirendab arendusprotsessi sest keegi ei pea teiste järel ootama. See aga paneb teatud piirid koodi suurusele, sest suuremaid koodihulki ei jõua kõik omastada.

Visuaalset modelleerimist tarkvara projekteerimisel eelistavad rationali unifitseeritud arendusprotsess ja erisus-juhitud arendus. Sama ei arva aga ekstreemprogrammeerimine ja crystal clear, kuna nende arvates kulub liigseid ressursse mudelite loomiseks, ning see pole eesmärk.

Ekstreemprogrammeerimine erineb teistest metoodikatest oma praktikate poolest, eelkõige paarisprogrammeerimise, test-juhitud arenduse ning koodi korduvkasutamisega.

Võrreldes teiste väledate meetoditega on dünaamilisel arendusmeetodil oma keskne mittetulunduslik konsortsium, kes tegeleb uuendustega ning pakub kasutajatuge. Seega saab metoodikat kasutada vaid litsentsi alusel.

Kokkuvõtteks, meetodi valimisel tuleks arvestada sellega, et igale projektile ei ole otstarbekas kasutusele võtta ühte ja sama metoodikat. Samas tuleks arvestada ka projekti suurusest ning keerukusest. Ekstreemprogrammeerimine tundub olevat üsnagi mõistlik valik, kui arendajad soovivad luua uut tarkvara detailsete praktikate süsteemis. Ekstreemprogrammeerimine jätab kasutamata erinevad visuaalsed modelleerimised ning saab hakkama ka kriitiliste projektidega.

8 Ekstreemprogrammeerimise teooria ja praktilise projekti võrdlus

Tihtilugu ei vasta teooria praktikale. Autor üritab siinkohal võrrelda ekstreemprogrammeerimise teooriat ning selle reaalset kasutamist. Võrdlus põhineb erinevate kirjatükkide, aruteludele ning lähedaste-tuttavate poolt saadud andmete analüüsist.

8.1 Diagrammid

Kuna reeglina ekstreemprogrammeerimises diagramme ei kasutata, siis on leitud, et on palju efektiivsem, kui klient oma kasutajalugusid ette joonistab. Eriti hea on see kasutajaliidese küsimustes. Diagrammile lisatakse kommentaaridena selgitused. Tähtis on ka kasutajalood korrektselt nimetada, et ei tekiks hiljem õige kasutajaloo leidmisega segadusi. Parim viis oleks neid nummerdada kasvavalt.

8.2 Kasutajalood

Tänu kasutajalugudele ning kliendi ja arendaja tihedale koostööle tekib arendajal parem arusaam tarkvarale esitatavatest nõuetest ning suureneb kliendi rahulolu eelkõige just rakenduse väljanägemise suhtes. Lisaks paraneb kliendi üldine suhtumine arendajatesse. Klient hakkab rohkem mõistma arendatava rakenduse mahtu ning sunnib end rohkem ja täpsemalt rakenduse nõudeid sõnastama.

8.3 Redaktsioonid

Lühikesed redaktsioonid on efektiivsed ning neid tuleks kasutada. Samas on neid ka suhteliselt lihtne projekti sisse viia. See, et nad efektiivsed ja head on, ei tähenda, et neid tuleks väga tihti teha. Redaktsioonid planeeritakse vastavalt vajadustele.

Redaktsioonide funktsionaalsus planeeritakse väga üldiselt. Selleks tehakse vastav koosolek, kuhu on kutsutud mõlema poole esindajaid. Koosolek ise kestab kaua ning selle jooksul lepitakse kokku redaktsiooni väljalaske kuupäevad ning selles sisalduv funktsionaalsus.

8.4 Iteratsioonid

Iga redaktsiooni juures üritatakse iteratsioone vähem teha kui ekstreemprogrammeerimine seda ette näeb. Pigem üritatakse igas iteratsioonis võimalikult palju ära teha. Tavaliselt on iteratsioonid üleplaneeritud ehk teisisõnu, kõiki iteratsiooni planeeritud kasutajalugusid ei suudeta realiseerida. Klient kirjutab soovilood ning reastab need tähtsuse järjekorras ning arendaja realiseerib neid vastavalt sellele. Kui kõiki ei suudeta ühte iteratsiooni panna siis lähevad esimesest välja jäänud funktsionaalsused edasi järgmisesse iteratsiooni. Iteratsioonide kasutamine on aidanud vältida ajahätta jäämisest tingitud negatiivseid kõrvalefekte – suhtlemise vähenemist juhtide ja klientidega

8.5 Disainimine

Rakenduste disainimist üritatakse võimalikult lihtsalt läbi viia, kuna kallimate disainimisvahendite ost ning vajalike inimeste koolitamine võib olla kulukas ning aeganõudev - seda vähemalt väikeste firmade puhul

8.6 Funktsionaalsus

Funktsionaalsuse poolest püütakse olla võimalikult minimaalne – tehakse vaid seda mida klient on oma kasutajalugudes maininud.

8.7 Rekodeerimine

Rekodeerimist ei tehta niipalju kui ekstreemprogrammeerimine ette näeb. Pigem tehakse seda siis, kui on aega või kui seda on ilmtingimata tarvis koodi puhastamiseks.

8.8 Kliendi kohalolek

Kliendi kohalolek on hea, et arendajad saaksid parema ülevaate loodavast tarkvarast. Samas oleks hea kui klient ise teaks võimalikult vähe tarkvara arendamisest või selles kastutatavas tehnilisest keelest, kuna sellest võib tekkida erinevaid lahkhelisid.

8.9 Kokkulepitud standardid

Kasutatakse kokkulepitud standardit, et iga programmeerija tiimis kirjutaks koodi sarnaselt teistele. Kuigi ekstreemprogrammeerimine tahab, et kõik kirjutaksid koodi kui üks, siis jääb ikkagi teatav individuaalsus koodikirjutamises igale programmeerijale külge, mis ei pruugi halb olla, seni kuni kood on lihtsalt loetav ja arusaadav.

8.10 Paarisprogrammeerimine

Paarisprogrammeerimist üldiselt ei taheta kasutada, sest programmeerijad on siiski inimesed, kes ei taha oodata teiste järgi.

8.11 Pidev integreerimine

Pidev integreerimine on tihti kasutatav ning lihtne juurutada. Selle kasutamine annab sama efekti mida ekstreemprogrammeerimine lubab. See muidugi eeldab, et käsil on testidel tuginev arendustegevus. Pidev integreerimine on kasulik, sest et selline tegevus hilisemas arengujärgus võib osutuda keerulisemaks ning nõuab rohkem ressursse. Hea oleks integreerimiseks kasutada eraldi arvutit, mille taga muud arendustööd ei tehta.

8.12 Kollektiivne omand

Kollektiivse omandina kasutatav kood on küll hea, kuid keeruline juurutada. Tore oleks, kui programmeerijad oleks võimelised igat kohta parandama ning täiustama. Tegelikult aga nii lihtne see pole – asjad lähevad kaduma, vanad ja valed koodiosad ilmuvad uude koodi jne. Kuid aja jooksul tekib see vilumus, mida oleks vaja.

8.13 Testimine

Testimist sisse viia on ilmselt raskem kui seda arvatakse. Kuid ka efekt on kõige suurem. Keeruline on kirjutada testi enne tarkvara. Testidel tuginev arendamine suurendab rakendusele kuluvat aega umbes 40 %, samas annab see teadud kindlustunde, et tarkvara toimib.

8.14 Kokkuvõte võrdlusest

Sellest võrdlusest saame järeldada, et ekstreemprogrammeerimisele omaseid praktikaid üldjuhul kasutatakse, aga mitte täielikult. Arendajad kasutavad keerulisemate ja aeganõudvatena praktikate puhul selliseid lahendusi, mis on eelnevates projektides kasutusel olnud ning hästi teada. Minnakse lihtsama vastupanu teed ning jäetakse vahele aeganõudvad protseduurid.

Kokkuvõte

Käesoleva bakalaureusetöö eesmärkideks oli anda ülevaade ekstreemprogrammeerimisest, selles kasutatavatest praktikatest, testimisest ning võrrelda seda teiste väledate arendusmeetodite esindajatega. Ülevaates sai välja toodud ekstreemprogrammeerimise tekke- ja arengulugu ning arendusprotsess. Lisaks oli juttu ka ekstreemprogrammeerimise vastuolulistest aspektidest ning võrdlus teooria ning selle kasutamisest reaalsetes tingimustes.

Antud tööst selgub, et sobivad ekstreemprogrammeerimise projektid on sellised :

- Mis kaasavad uut tehnoloogiat, kus nõuded muutuvad pidevalt või arendustöös tuleb enne seninägematuid rakendusprobleeme.
- Uurimusprojektid, kus tulemus ei ole mitte tarkvara produkt ise, vaid mingi üldisem teadus.
- Väikesed projektid, mida on lihtsam hallata.

Võrdlustest saime teada, et kuigi ekstreemprogrammeerimine on täiesti arvestatav väledate arendusmeetodite seas, ning selle kasutamisel on isegi teatav eelis teiste ees, siis arendajad väldivad ikkagi kasutamast kõiki meetodile omaseid praktikaid. Iseasi, kas me saame meetodit õigeks lugeda, kui selles ei kasutata kõiki olulisi praktikaid.

Käesolev töö on hea algus antud teemat käsitleva eestikeelse materjali loomisel, kuna ekstreemprogrammeerimisest eestikeelset materjali on leida väga vähe.

Viited

[Adaption] *Adaption Software – The Planning Game,* http://www.adaptionsoft.com/xp_practices_planning_game.html (21.10.2006)

[Andy Glover] *Test Categorization Tehcniques With Testing*, http://dev2dev.bea.com/pub/a/2006/09/testng-categorization.html (14.12.2006)

[ASD] Adaptive SD, http://www.adaptivesd.com/ (04.05.2007)

[Beck 1999] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[Beck 2000] Kent Beck, Martin Fowler, *Planning Extreme Programming*, Addison- Wesley, 2000

[Brett G.Palmer] Brett G. Palmer, *Testing Strategies*, http://www.ujug.org/stuff/TestingStrategies.pdf (02.12.2006)

[C2] *Cunningham & Cunningham, Inc*, http://c2.com/xp/HistoryOfExtremeProgramming.html (03.12.2006)

[CC] *Crystal Clear : A Human-Powered Methodology for Small Teams*, Alistair Cockburn, 2004

[CF] *ColdFusion Unit Testing Framework*, http://coldfusion.sys-con.com/read/46361.htm (01.11.2006)

[CodeP] *Code Project*, http://www.codeproject.com/gen/design/onunittesting.asp (13.12.2006)

[DSDM] *DSDM Consortium*, http://www.dsdm.org/ (05.05.2007)

[FDD] Feature Drivern Development, http://www.featuredrivendevelopment.com/ (12.02.2007)

[IBM] *IBM*, http://www-128.ibm.com/developerworks/library/j-mocktest.html (11.11.2006)

[Informit] Informit*: The Principles of Extreme Programming,* http://www.informit.com/articles/article.asp?p=26261&rl=1 (12.11.2006)

[Joseph Bergin] *Learning the Planning Game An Extreme Exercise,* http://csis.pace.edu/~bergin/xp/planninggame.html (10.11.2006)

[The Economist] *The Economist*, http://www.economist.com (10.11.2006)

[PairProgramming] *Pair Programming, an Extreme Programming practice*, http://www.pairprogramming.com/ (05.12.2006)

[Rex Black] Rex Black, *Managing the Testing Process*, Wiley Publishing, 2002

[RUP] IBM Rational Software, http://www-306.ibm.com/software/rational/ (02.05.2007)

[Mayford] *Mayford Technologies*, http://www.mayford.ca/xp/40hourweek.shtml (11.11.2006)

[Matt Albrecht] *Integration Unit Test*, http://groboutils.sourceforge.net/testing-junit/art_iut.html (09.12.2006)

[Matt Stephens] Matt Stephens, *The Case Against Extreme Programming*, http://www.softwarereality.com/lifecycle/xp/case_against_xp.jsp (03.11.2006)

[MF] Martin Fowler, *Continuous Integration*, http://www.martinfowler.com/articles/continuousIntegration.html (07.12.2006)

[Methods&Tools] *Methods and Tools,* http://www.methodsandtools.com/archive/archive.php?id=10 (17.11.2006)

[Methods&Tools 2] *Methods and Tools,* http://www.methodsandtools.com/archive/archive.php?id=13 (09.12.2006)

[Scrum] Agile Thoughts – *What is scrum,* http://agilethinking.net/blog/what-is-scrum/ (15.03.2007)

[Seeba 2002] Asko Seeba, *Väledad protsessid ja XP (Extreme Programming),* http://www.cs.ut.ee/~asko/tarkvaratehnika/valedad_protsessid_ja_xp/ (20.11.2006)

[Steve Yegge] Steve Yegge, *Good Agile, Bad Agile,* *http://steve-yegge.blogspot.com/2006/09/good-agile-bad-agile_27.html* *(12.12.2006)*

[XP 1] *Extreme Programming Resource*, http://www.xprogramming.com/ (01.12.2006)

[XP 2] *Extreme Programming: A gentle introduction*, http://www.extremeprogramming.org/ (25.11.2006)

[XP Exchange] *XP Exchange*, http://www.xpexchange.net/english/intro/collectiveCodeOwnership.html (12.10.2006)

Summary

Extreme programming: overview, practices and comparison to other agile development methods

Priit Valdmees

Extreme Programming has been advocated recently as an appropriate programming method of the high-speed, volatile world of Internet and Web software development. It has evolved since 1999 when Kent Beck wrote a book about his recently used methods in Chrysler. It consists of many specific values, practices, principles and activities.

The goal of this thesis was to introduce extreme programming and to find out if it is over valued or not.
This paper gives a brief overview of the method, its practices, testing, comparison with other agile development methods and differences between the theoretical and practical side of extreme programming.

The author found that even though there are many other agile development methods around, extreme programming is still one of the best because it has a detailed practice system and using it doesn't cost anything. With extreme programming developers don't produce any documentation and visual modeling. It can handle critical projects and it reduces costs of projects and helps develop them faster.

Also, when comparing practical projects with extreme programming theory the author found that developers don't use all the practices. They mostly only use those that are easy to include in the project and which distinguish them from other methods. The others practices are replaced with those that are more common or familiar to programmers.

Extreme programming is one of the best methods in agile development and it's a good method to use when the project requirements are changing rapidly.