UNIVERZA V LJUBLJANI
EKONOMSKA FAKULTETA

MAGISTRSKO DELO

JANEZ STRMEC

UNIVERZA V LJUBLJANI
EKONOMSKA FAKULTETA

MAGISTRSKO DELO

PRENOVA METODE RAZVOJA PROGRAMSKIH REŠITEV V PODJETJU HERMES SOFTLAB

LJUBLJANA, NOVEMBER 2009 **JANEZ STRMEC**

IZJAVA

Študent Janez Strmec izjavljam, da sem avtor tega magistrskega dela, ki sem ga napisal pod mentorstvom dr. Andreja Kovačiča, in da v skladu s 1. odstavkom 21. člena Zakona o avtorskih in sorodnih pravicah dovolim njegovo objavo na fakultetnih spletnih straneh.

V Ljubljani, dne 12.11.2009 Podpis: _____

KAZALO

KAZALO SLIK

KAZALO TABEL

UVOD

Kot v vseh ostalih ekonomskih dejavnostih trg tudi pri razvoju programskih rešitev sili v optimizacijo procesov in virov. Razvoj programskih rešitev poteka v dinamičnem, hitro spreminjajočem se okolju, brez dobro definiranih industrijski standardov in brez zanesljivih kvalitativnih podatkov iz sorodnih projektov. Tehnologija se praktično spreminja iz dneva v dan, tako da panoga zahteva nenehno sledenje spremembam in izobraževanje zaposlenih. Sodobne smernice so izpostavile dve načeli, s katerima lahko pripomoremo k učinkovitejši izrabi virov. Tudi podjetja, ki razvijajo programske rešitve, niso imuna na nižjo ceno delovne sile v ekonomsko manj razvitih deželah. Uporaba učinkovitih in »lahkih« oz. agilnih metodologij nam omogoča, da skrajšamo čas razvoja programskih rešitev in ustrezno obvladujemo potek projektov (Agile Manifesto, 2001). Na ta način je izraba virov tudi bolj smotrna in dodana vrednost programske rešitev višja, kar pripomore k neposredni ekonomski koristi za podjetje.

Čeprav je danes na voljo precej metodologij za razvoj programskih rešitev in projektnega vodenja, se podjetja težko odločajo za njihovo uporabo. Razlog za to je po eni strani v metodologijah samih, ker so mnoge metodologije preveč rigorozne in obširne in zato neprimerne za uporabo v manjših podjetjih ali pa preveč osiromašene, da bi lahko učinkovito podpirale velike projekte. Po drugi strani pa tudi v samih podjetjih ni pravega interesa za vpeljavo enotnega procesa, saj omogoča neformalni »ad-hoc« ali namenski pristop razvoja programskih rešitev (model kodiraj in popravljaj) zelo hiter začetek izdelave programske rešitve. Vendar pa uspeh s takšnim pristopom temelji izključno na razpoložljivosti in uspešnosti posameznikov in ne zagotavlja podlage za dolgoročno produktivnost in kakovostno izboljšanje razvoja programskih rešitev v celotnem podjetju. Do neprestanega izboljšanja razvoja programskih rešitev lahko podjetje pride le z osredotočenim in trajnim prizadevanjem za krepitev procesa infrastrukture za učinkovit razvoj programskih rešitev in praks upravljanja (Paulk, Curtis, Chrissis & Weber, 1993).

V podjetjih je še vedno v precejšnji meri prisotno razvijanje programskih rešitev po zaporednemu modelu (angl. *»Waterfall model«*) (Laplante & Neill, 2004). Vendar pa se zaporedni modela razvoja programskih rešitev v podjetjih, zaradi svojih pomanjkljivosti, uporablja vedno manj (Avison & Fitzgerald, 2002). Zaporedni model je model razvoja programske rešitve, v katerem razvoj poteka linearno skozi faze analize, načrta, izvedbe, testiranja, integracije in vzdrževanja (Royce, 1970). Pri tem si faze razvoja programske rešitve sledijo popolnoma zaporedno in se ne prekrivajo. Naslednja faza se začne, ko se predhodna konča, kar omogoči natančno definiranje aktivnosti posameznih faz razvoja programskih rešitev.

Sodobne metodologije uvajajo iterativni in agilni pristop (agilni model). Pri agilnem modelu razdelimo programsko rešitev na različne funkcionalnosti tako, da jih lahko

razvijamo sočasno in/ali izmenično. Ob končanemu razvoju vsake funkcionalnosti le-to testiramo in integriramo z ostalimi deli programske rešitve. Pristop omogoča precejšnje povečanje prilagodljivosti razvojnega procesa. Agilne metodologije razvoja prinašajo dodano vrednost zaradi svoje prožnosti, prilagodljivosti, konstruktivnega sodelovanja in popolne usmerjenosti v poslovno vrednost, kar izhaja že iz »Agile Manifesto« ali agilnega manifesta (Agile Manifesto, 2001):

- posamezniki in njihova interakcija pred procesi in orodji,
- delujoča programska rešitev pred podrobno dokumentacijo,
- sodelovanje s stranko pred pogodbenimi pogajanji in
- odziv na spremembe pred sledenjem načrta.

Namen magistrskega dela

Namen magistrskega dela je s pomočjo domače in tuje strokovne literature preučiti razloge za prenovo modela razvoja programskih rešitev v podjetju Hermes SoftLab d.o.o. Opredelil sem razloge za prenovo modela razvoja programskih rešitev, kaj jih sili v prenovo ter kakšne so prednosti in slabosti le teh. Ugotovil sem, kakšne pogoje postavlja trg pri razvoju programskih rešitev ter kakšne so njihove smernice. Glede na to, da je razvoj programskih rešitev hitro spreminjajoča se dejavnost, je potreba po stalnih spremembah in prilagajanju podjetij ključnega pomena za njihov obstoj ter uspešno delovanje.

Cilj magistrskega dela

Cilj magistrskega dela je s pomočjo analize preučiti smotrnost prenove modela razvoja programskih rešitev v podjetju Hermes SoftLab d.o.o. in predstaviti možnosti nadaljnje prenove modela razvoja programskih rešitev na podlagi preučitve obstoječih metodologij razvoja programskih rešitev. Cilj je analizirati dva vidika razvoja programskih rešitev, in sicer zaporedni model in agilni model na podlagi praktičnega primera. Podjetje Hermes SoftLab d.o.o. posluje v velikih timih nad 20 ljudi, v razmeroma dobro načrtovanih organizacijah, vendar pa bi danes brez agilnih metodologij težko dosegali učinkovitost. Skušal sem ugotoviti, ali obstajajo skupne točke med tema dvema modeloma ter ali novejši, bolj sodoben in timsko orientiran model, pripomore k hitrejšemu, racionalnejšemu in kvalitetnejšemu razvoju programskih rešitev v podjetju. Skušal sem tudi ugotoviti možnosti nadaljnje izboljšave sedanjega modela razvoja programskih rešitev.

Metode dela

Magistrsko delo v prvem delu vsebuje poglobljeno teoretično-analitični pogled strokovne literature, znanstvenih razprav in raziskav ter člankov tujih strokovnjakov s področja obravnavane teme. Poleg kritične analize strokovnoteoretičnih dogajanj so zaradi lažjega razumevanja za ilustracijo navedeni tudi nekateri praktični primeri iz strokovne literature. Ta del magistrskega dela sem analiziral s pomočjo opisne metode, s katero so združena spoznanja mnogih avtorjev s področja prenove razvoja programskih rešitev. Drugi, empirični del magistrskega dela, temelji na analizi prenove razvoja programskih rešitev v okviru izboljšanja poslovanja podjetja.

Pri izdelavi magistrskega dela sem uporabil tudi teoretična znanja, pridobljena v okviru podiplomskega študija, in znanje, ki sem ga pridobil kot zaposlen v podjetju Hermes SoftLab d.o.o.

Struktura magistrskega dela

Magistrsko delo je sestavljeno iz treh poglavij, tematika pa je dodatno razdeljena v podpoglavjih. V uvodnem poglavju sem predstavil obravnavano problematiko in opredelil načrt raziskave. Metodologije razvoja programskih rešitev so predstavljene v prvem poglavju, v katerem je opredeljena uporaba metodologije pri razvoju programskih rešitev ter njihov pomen. Metodologije razvoja programskih rešitev so se s časom močno spremenile zaradi izjemno hitrega tehnološkega razvoja in sprememb na področju organizacije. Uspešnost in učinkovitost organizacije temeljita predvsem v novi obliki organizacije in timskega dela, kar sem podrobneje opredelil v drugem poglavju. V tretjem poglavju sem na podlagi podjetja Hermes SoftLab d.o.o. s pomočjo analize poslovanja podjetja preučil smotrnost prenove modela razvoja programskih rešitev. Opisal sem razvojne metodologije v podjetju, njihovo vodenje in kakovost. Predstavljen je razvoj programskih rešitev v podjetju, ki je v veliki meri prilagojen zaporednemu modelu. Analiziral sem vpliv uvedbe agilnega modela na poslovanje v podjetju. Preučil sem tudi nadaljnje možnosti prenove modela razvoja programskih rešitev v podjetju. Magistrsko delo sem zaključil s sklepnimi ugotovitvami.

1 METODOLOGIJE RAZVOJA PROGRAMSKIH REŠITEV

Metodologija razvoja programskih rešitev je okvir, ki se uporablja za konstrukcijo, načrtovanje, nadzor in postopek razvoja programskih rešitev. Metodologijo za razvoj programskih rešitev lahko na splošno opredelimo kot skupek postopkov, tehnik, orodij in pripomočkov za dokumentiranje, ki koristijo razvijalcem sistema pri njihovem prizadevanju razvoja programske rešitve. Metodologijo razvoja programskih rešitev sestavljajo faze, ki so same sestavljene iz podfaz in ki vodijo razvijalce programskih rešitev pri izbiri tehnik, ki so primerne v vsaki fazi projekta ter jim pomagajo načrtovati, upravljati, kontrolirati in ocenjevati projekte razvoja programskih rešitev (Avison & Fitzgerald, 2002). Vendar pa metodologije razvoja programskih rešitev ne uporabljajo samo naključnih sklopov pravil, ampak vsebujejo koncepte in prepričanja, ki opredeljujejo vsebino in obnašanje objektov programskih rešitev ter tudi vrednote, ki določajo, kakšne so lastnosti v programskih rešitvah, ki so dobre in zaželene (Papatsoutsos, 2001).

1.1 Nastanek metodologij razvoja programskih rešitev

Metodologije razvoja programskih rešitev so se prvič pojavile leta 1960 (Elliott, 2004). V tistem času ni bilo jasnih smernic za pisanje programskih rešitev, ki opredeljujejo zahteve in uspešno upravljanje zapletenih projektov. Takratni razvoj programskih rešitev brez kakršnih koli omejitev je privedel do propada večjega števila projektov razvoja programskih rešitev. Zato so na tiskovni konferenci NATO (The NATO Software Engineering Conferences) leta 1968 v Bruslju poskušali najti rešitev za težave, ki nastajajo, in nato prvič uporabili izraz programsko inženirstvo (Naur & Randell, 1968). Rešitev za kaos, ki se je pojavljal pri projektih, povezanih z razvojem programskih rešitev, je uporaba preverjenih metod in načel, ki se že dlje časa uspešno uporabljajo na področju strojništva in gradbeništva. Pri tem se je pojavilo vprašanje, ali se programska rešitev lahko primerja z drugimi vejami inženiringa. Zaključek je bil, da je težko najti načela za razvoj programske rešitve. Potrebno je bilo pridobiti praktična priporočila pri razvoju programskih rešitev (najboljše prakse). Praktična priporočila so niz metod, praks, orodij in proces, ki se na splošno šteje, da so v določenem razvoju dokazano optimalni.

Obstajajo različne metodologije razvoja programskih rešitev, ki so se postopoma razvijale, vendar pa ima vsaka metodologija razvoja programskih rešitev svoje prednosti in slabosti. Določena razvojna metodologija ni nujno primerna za uporabo v vseh projektih. Vsaka od razpoložljivih metodologij je primerna za različne vrste projektov, ki temeljijo na različnih tehničnih, organizacijskih, projektnih in timskih značilnostih (Green & DiCaterino, 1998).

Nekatere metodologije razvoja programskih rešitev poudarjajo človeške vidike razvoja programskih rešitev, druge so osredotočene bolj na znanstveni pristop, nekatere so pragmatične, nekatere skušajo avtomatizirati čim več dela pri razvoju programskih rešitev. Različne metodologije razvoja programskih rešitev lahko sledijo različnim ciljem. Ti cilji so lahko (Avison & Fitzgerald, 2002):

- natančno zabeležiti zahteve programske rešitve. Metodologija razvoja programske rešitve pomaga uporabnikom določiti njihove zahteve in funkcionalne potrebe, nato pa razvijalci programske rešitve preučujejo in analizirajo uporabniške zahteve.
- določiti sistematično metodo razvoja programske rešitve, kjer se lahko napredek učinkovito meri in opazuje. Težave pri velikih projektih, ki niso izvedeni v rokih, lahko povzročijo velike stroške ali druge negativne posledice za podjetje. Dobro načrtovane faze razvoja programske rešitve in vmesni cilji v metodologiji razvoja programske rešitve omogočijo učinkovitost tehnik načrtovanja projekta.
- razviti programske rešitve v okviru časovnega roka pri sprejemljivih stroških. Čas, namenjen posameznim tehnikam, naj bo omejen, sicer je mogoče iskanju popolnosti posvetiti ogromno časa (Kovačič, 1998).
- dokumentirati razvoj programske rešitve. Kvalitetna in natančna dokumentacija omogoča izvedbo modifikacij programske rešitve, ki so lahko posledica sprememb v organizaciji in njenem okolju z manj vpliva na druge dele programske rešitve.
- ugotoviti spremembe, ki jim bo potrebno slediti čim bolj zgodaj v procesu razvoja programske rešitve. S tem, ko razvoj programske rešitve napreduje skozi faze analize in načrtovanja do faze implementacije, se povečujejo stroški, povezani s spremembami programske rešitve. Prej ko realiziramo spremembe, manjši so stroški projekta.
- izdelati programske rešitve, ki ustrezajo ljudem, na katere programska rešitev vpliva. Če programska rešitev ustreza naročnikom programske rešitve (stranke, vodstvo, uporabniki), je bolj verjetno, da se bo programska rešitev uporabljala in bo bolj uspešna.

Našteti cilji predstavljajo le nekaj primerov, ki jim lahko metodologija razvoja programskih rešitev sledi. Metodologija razvoja programskih rešitev se lahko osredotoča na več ciljev ali pa sledi zgolj enemu. Bistveno pri vsem tem pa je razumevanje, da v ozadju vsake metodologije razvoja programskih rešitev stoji neka ideja, ki nas usmerja pri delu tako, da določi kvalitete programskih rešitev, ki jih skušamo pri razvoju doseči.

1.2 Formalna in neformalna metodologija razvoja programskih rešitev

Metodologija razvoja programskih rešitev zajema poleg formalno opredeljenih elementov, ki so običajno zapisani v obliki postopkov, pravil, napotkov, smernic, standardov itd. v navadnih ali elektronskih priročnikih, tudi številne druge nedokumentirane, neformalne

elemente. Posebno mesto med njimi nosi znanje, ki ga člani organizacije uporabljajo pri svojem delu. To je za organizacijo ključnega pomena, saj predstavlja tiste vrednote, ki so organizaciji lastne in so njena konkurenčna vrednost. Posledično velja to tudi za metodologijo samo. Temeljni elementi metodologije se skrivajo ravno v znanju in izkušnjah posameznikov (Reade & Froome, 1990).

Meje med formalnim in neformalnim delom metodologije seveda ni enostavno postaviti. Če je formalni del metodologije obsežen, ga težko vzdržujemo, kar ima za posledico zastaranost metodologije in njeno neustreznost dejanskemu stanju. Metodologija je namreč dinamična in se stalno spreminja. Po drugi strani pa je obsežna neformalna metodologija tvegana za organizacijo, saj je popolnoma odvisna od njenih uporabnikov. Potreba po formalizaciji metodologije izhaja že iz dejstva, da je razvoj programskih rešitev sistematičen proces, ki mora biti ustrezno zasnovan in dokumentiran, da lahko učinkovito usmerja posameznike in skupine do dobrih rezultatov. Če je metodologija opredeljena zgolj na neformalni ravni, je razvoj težko standardizirati, sam postopek pa postane hitro nepregleden in posledično neobvladljiv. S pomočjo iskanja ustreznega ravnovesja med formalno in neformalno metodologijo skušajo doseči ustrezno prilagodljivost, ki omogoča »ad-hoc« nastavitev metodologije, tako da ta čim bolj ustreza konkretnemu primeru (Bjorner, 2005).

1.3 Organizacijsko okolje metodologij razvoja programskih rešitev

V mnogih primerih razvoja programskih rešitev so metodologije prilagojene določenemu razvoju programskih rešitev v podjetju (Land, 1998). To se zgodi zato, ker je bila večina metodologij, katerih namen je zagotoviti okvir za doseganje uspeha, zgrajenih in razvitih samo pod nekaterimi določenimi okoliščinami. Če se okoliščine spremenijo izven meja modela, so rezultati razvoja programskih rešitev z določeno metodologijo nepredvidljivi. Zato je včasih potrebno spremeniti ali zamenjati obstoječo metodologijo razvoja programskih rešitev za prilagoditev novim okoliščinam. Land (1998) domneva, da je primerna metodologija razvoja programskih rešitev odvisna predvsem od organizacijskega okolja.

Nespremenljivo okolje – zahteve po spremembi programske rešitve so nespremenljive za čas trajanja razvoja programske rešitve. Te zahteve je mogoče tudi natančno in celovito opredeliti in določiti. V takem okolju bi formalne metode razvoja programskih rešitev (npr. zaporedni model ali spiralni model) zagotovile popolnost in natančnost, ki jih zahteva programska rešitev.

Nemirno okolje – organizacija doživlja nenehne spremembe, kar povzroči, da se tudi zahteve programskih rešitev nenehno spreminjajo. Programske rešitve, razvite na osnovi

običajnih modelov (npr. zaporedni model), bi bile zastarele še pred zaključkom razvoja programskih rešitev. Uspešne metodologije razvoja programskih rešitev bi bile tiste, ki omogočijo hiter in modularen razvoj (npr. razvoj na podlagi prototipov).

Negotovo okolje – zahteve programskih rešitev niso znane ali so negotove, zato ni mogoče natančno opredeliti zahteve v naprej. V takem okolju je potrebno, da je razvoj programskih rešitev zelo inovativen. Tu je potrebno upoštevati in uporabljati metodologije razvoja programskih rešitev na podlagi učenja, ki izkoriščajo hiter in agilen razvoj.

Prilagodljivo okolje – okolje se lahko spremeni na podlagi razvoja programske rešitve. Metodologije razvoja programskih rešitev morajo v tem okolju omogočiti enostavno uvajanje novih pravil in prilagodljivost organizacije.

1.4 Prednosti uporabe metodologij za razvoj programskih rešitev

Z uporabo metodologij za razvoj programskih rešitev se pričakujejo nekatere prednosti, ki omogočajo lažji razvoj programskih rešitev (Jayaratna, 1994):

Struktura nalog in funkcij – metodologija razvoja programskih rešitev ni samo zbirka nalog in aktivnosti, vendar vključuje in razloži tako proces za razvoj nove programske rešitve kot tudi notranjo logiko same programske rešitve.

Nadzor stroškov in nadzor izvajanja projekta – večji projekti razvoja programskih rešitev običajno vključujejo precejšnje naložbe, ki vključujejo strojno in programsko rešitev, potrebno za delovanje tima in za razvoj in delovanje programske rešitve. Z metodologijo razvoja programskih rešitev se trdno opredelijo koraki pri razvoju programskih rešitev, določi se napreden nadzor nad operativnimi stroški in končni obliki programske rešitve.

Zmanjševanje negotovosti – vsak korak pri razvoju programskih rešitev je jasen vnaprej, tako so lahko rezultati bolj predvidljivi, kar pomeni, da je razvoj programskih rešitev bolj merljiv in obvladljiv.

Nadzor timskega dela – metodologija razvoja programskih rešitev opredeljuje tako sposobnosti, ki se pričakujejo od članov razvojnega tima, kot tudi vloge, ki so jim dodeljene. Ta oblika strukture in upravljanja tima pojasnjuje razmerja med člani in zmanjšuje težave, ki nastajajo zaradi nepredvidljivih vedenj članov tima.

Orodje učenja – s ponavljajočimi koraki metodologije razvoja programskih rešitev se pridobivajo znanja in izkušnje o značilnostih programske rešitve in odnosov v organizaciji, kar omogoča lažji in bolj kakovosten nadaljnji razvoj programskih rešitev.
Upoštevanje zunanjih dejavnikov – druge organizacije silijo, da organizacije pri razvoju programskih rešitev upoštevajo certifikate kakovosti (npr. ISO 9000). Z uporabo metodologije razvoja programskih rešitev se razvoj programske rešitve z upoštevanjem certifikatov kakovosti močno olajša.

1.5 Slabosti uporabe metodologij razvoja programskih rešitev

Kljub dobrim lastnostim uporabe metodologij razvoja programskih rešitev se zaradi njihove narave in strukture (Paul, 1993) ter neprestanega spreminjanja organizacijskega okolja (Fitzgerald, 1999) pojavljajo pri njihovi uporabi naslednje težave:

Izrek o negibni točki – vse metodologije razvoja programskih rešitev temeljijo na predpostavki znani kot »izrek o negibni točki« (angl. *»Fixed Point Theorem«*) (Paul, 1993). Ta izrek ima za samoumevno, da obstaja trenutek, ko vse zainteresirane strani v razvoju programske rešitve vedo, kaj hočejo in da bo to ostal nespremenjeno. V metodologijah razvoja programskih rešitev se ta trditev ni nikoli izkazala za resnično. Celo tako imenovani agilni pristopi ne upoštevajo izreka v celoti, saj je potrebno upoštevati več parametrov, ki so med seboj prepleteni.

Hitro spreminjajoče se okolje – edina stalnica v današnjem poslovnem okolju je sprememba. V nestanovitnem okolju je potrebno hitro prilagoditi informacijski sistem na nenehne spreminjajoče se razmere (Salmela, Lederer & Reponen, 2000). Glede na zgoraj navedeno metodologija razvoja programskih rešitev, ki temelji na izreku o negibni točki, ne more izpolniti teh zahtev.

Stroški – uporaba metodologije razvoja programskih rešitev podjetju prispeva dodatne stroške. To pomeni, da če podjetja in organizacije pred začetkom razvoja programskih rešitev nimajo dovolj sredstev, potem pri razvoju programskih rešitev preprosto sledijo modelu kodiraj in popravljaj.

Čas – uporaba metodologije razvoja programskih rešitev je dolgotrajen postopek. To pomeni, da formalizirane metodologije razvoja programskih rešitev niso videti primerne za vse hitrejše spreminjajoče se okolje. To, v povezavi z izrekom o negibni točki, naleti na dejstvo, da bodo med dolgoročnim projektom okolje in pogoji na koncu projekta drugačni od tistih na začetku.

Ponovna uporaba – najbolj klasično oblikovane metodologije razvoja programskih rešitev pogosto ponovno izumljajo »kolo«, medtem ko se agilne metodologije razvoja programskih rešitev nagibajo k ponovni uporabi že razvitih programskih rešitev. Razvoj programskih rešitve je potrebno prilagoditi tako, da se ne troši sredstev, kar pripomore uporaba sodobnih agilnih metodologij razvoja programskih rešitev (DeMarco, 1996).

Alternativni vidiki – klasično oblikovane metodologije razvoja programskih rešitev ne upoštevajo »umetniške« strani razvoja programskih rešitev. Ta vidik je še posebej pomemben pri razvoju programskih rešitev končnemu uporabniku, pri katerem je potrebno upoštevati, da uporabnik uporablja programsko rešitev brez usposabljanja.

1.6 Klasične metodologije

Težke ali klasične metodologije se obravnavajo kot tradicionalni način razvoja programske rešitve, ki se uporabljajo v praksi že zelo dolgo časa. Te metode temeljijo na zaporednem nizu korakov, kot so opredelitev zahtev, razvoj rešitve, testiranje in uvajanje. Nalagajo discipliniran proces razvoja programske rešitve z namenom, da bi bil razvoj programske rešitve bolj predvidljiv in bolj učinkovit. Fowler (2006) kritizira, da so te metodologije birokratske, kar celotno hitrost razvoja upočasni. Klasične metodologije so nagnjene k prvotnemu načrtu, ki podrobno opisuje projekt za dolgo časovno obdobje ter obravnava dokumentacijo kot ključni del projekta. Ta pristop omogoči razvijalcem disciplino, kjer je razvoj predvidljiv in ponovljiv. Velik poudarek je na vizualni predstavitvi potreb programske rešitve in učinkoviti rešitvi te potrebe. Vizualizacija omogoča opredelitev programske rešitve, ki deluje kot temelj za gradnjo, napoveduje naloge in delegacijo, razumno napoveduje program in proračun za gradnjo. Obstaja veliko različnih klasičnih metodologij, vendar sem v nadaljevanju predstavil le najpomembnejše.

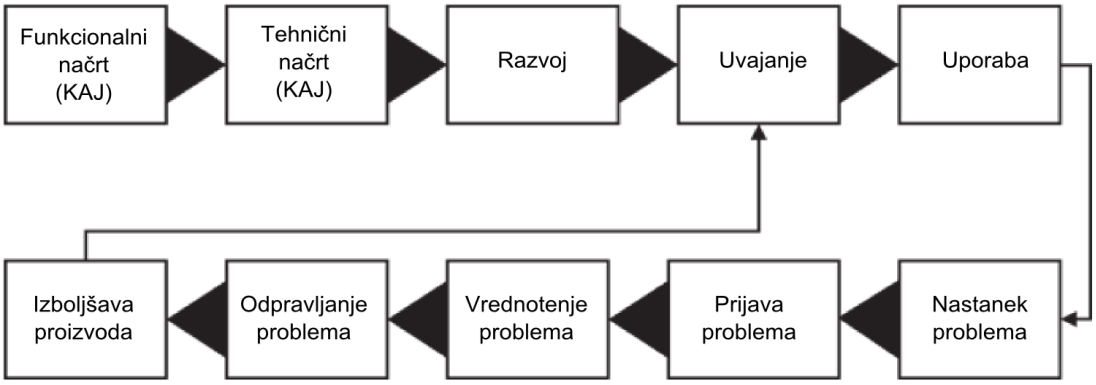
1.6.1 Kodiraj in popravljaj

Kodiraj in popravljaj ni pristop ali metodologija v pravem pomenu besede (Slika 1). Razvoj programske rešitve je sestavljen iz dveh dejavnostih v zaporednem vrstnem redu: kodiranje in popravljanje. Uporabnik se opredeli, kaj je najbolj pomembno pri razvoju. Uporabnikove želje so lahko v obliki skice, elektronskega sporočila ali druge vrste šibke specifikacije. Razvoj programske rešitve se nato v celoti izvede. Končano programsko rešitev se pošlje uporabniku, ki preizkusi produkt ter priskrbi razvijalcem potrebne povratne informacije. Razvijalci tako večino svojega časa porabijo za razvoj in popravljanje napak. Pri takem načinu razvoja programskih rešitev se pojavi veliko negativnih stvari:

- kakovost programske rešitve je nizka.

- pogosto se projekt konča neusklajeno s strankinimi cilji.
- programsko rešitev je težko izboljšati in vzdrževati.
- težje je razviti kompleksnejše in časovno izdatnejše programske rešitve.

Slika 1: Razvojni model kodiraj in popravljaj



Vir: B. K. Jayaswal & P. C. Patton, Software Development Methodology Today, 2006, str. 11.

Model razvoja programskih rešitev na podlagi kodiranja in popravljanja se je izkazal za nepraktično iz dveh razlogov (Schach, 1999). Če stranka tekom razvoja programske rešitve spremeni zahtevo, je potrebno ponovno pričeti z razvojem ali pa popraviti trenutno programsko rešitev, preden se lahko razvoj nadaljuje. Drugi razlog je neuporaba že obstoječih programskih rešitev. V modelu kodiraj in popravljaj se pri razvoju programskih rešitev ne uporablja že obstoječih programskih rešitev, vendar podjetje v celoti razvije svojo programsko rešitev, kar je lahko časovno in stroškovno potratno.

1.6.2 Zaporedni model

V začetku leta 1970 je Royce predstavil rešitev za prihajajočo krizo razvoja programskih rešitev v obliki modela, ki opisuje razvoj programske rešitve kot življenjski cikel, imenovan zaporedni model (angl. *»Waterfall model«*) (Dorfman & Thayer, 2000). Po zaporednem modelu si faze razvoja programskih rešitev sledijo zaporedno in se ne prekrivajo. Naslednja faza se začne, ko se predhodna konča. Prednost zaporednega modela razvoja programskih rešitev je natančna definicija aktivnosti posameznih faz razvoja programskih rešitev, ki so potrebne za prehod v naslednjo fazo. Guimarães & Vilela (2005) opisujeta zaporedni model kot enega od najbolj pomembnih modelov razvoja programskih rešitev.

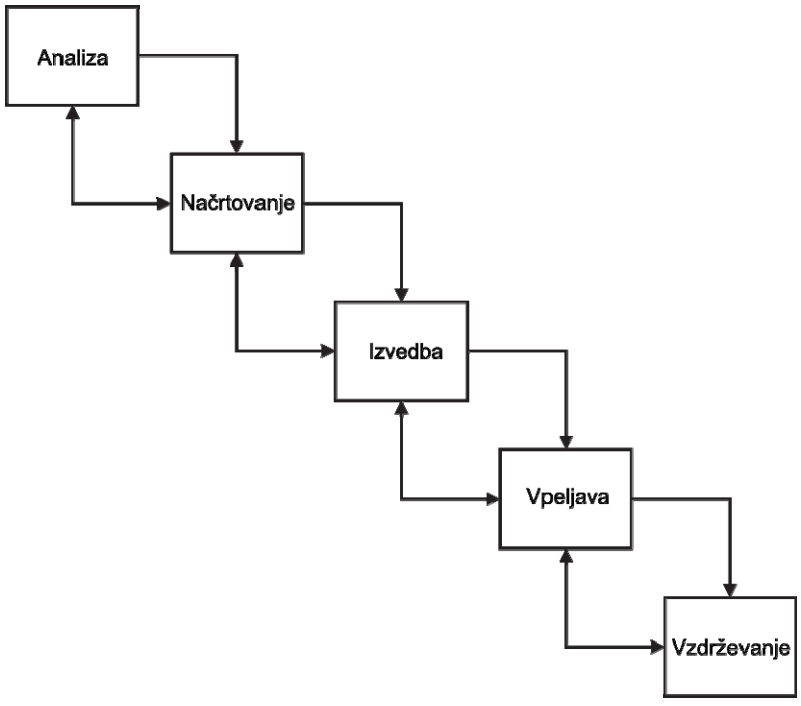
Ker je zaporedni model razvoja programskih rešitev organiziran v seriji linearnih faz, to spodbuja k natančno določenim in dokumentiranim postopkom razvoja programskih rešitev (Guimarães & Vilela, 2005). Tako dokumentacija odločno opredeljuje ugotovitve

prejšnjih faz in omogoči bolj formaliziran nadaljnji razvoj programskih rešitev v naslednjih fazah. Vendar pa zaporedni model ne deluje dobro pri interaktivnih programskih rešitvah za končne uporabnike. Določanje zahtev za take programske rešitve je težko, saj je oblikovanje vmesnika zelo subjektivno ter tudi uporabniki zelo redko razumejo resnične potrebe, ki jih programska rešitev mora vsebovati (Boehm, 1988).

1.6.2.1 Razčlenitev zaporednega modela razvoja programskih rešitev na posamezne faze

Ena od poglavitnih značilnosti zaporednega modela razvoja programskih rešitev je, da ima razvoj programske rešitve svoj začetek in konec. Vmes poteka vrsta aktivnosti, katerih skupni cilj je razvoj zahtevane programske rešitve. Glede na vrsto aktivnosti in njihovo intenziteto razdelimo razvoj programske rešitve s pomočjo zaporednega modela na več faz. Kot je razvidno iz slike zaporednega modela (Slika 2), je preskok na naslednjo fazo mogoč le, če je predhodna faza v celoti izpolnjena. V literaturi zasledimo nekaj različnih modelov, ki se predvsem razlikujejo po podrobnosti delitve, vendar pa klasični razvojni cikel sestavlja pet faz.

Slika 2: Zaporedni model



Vir: B. K. Jayaswal & P. C. Patton, Software Development Methodology Today, 2006, str. 12.

Faza analize je prva faza zaporednega modela. V tej fazi nas predvsem zanima odgovor na vprašanje, kaj je vsebina projekta. Zanima nas, katere funkcije in delovne procese mora

podpirati programska rešitev, kaj se izvaja v elementarnih aktivnostih, katera poslovna pravila morajo funkcije upoštevati in katere podatke potrebujejo elementarne aktivnosti, poslovna pravila in delovni procesi za svoje delovanje. Naloga faze analize je izdelati razumljiv opis področja, na katerega se nanaša razvoj programske rešitve, organizacijskega sistema ter procesov, ki se v njem izvajajo, in pravil, ki v njem veljajo.

Naloga **faze načrtovanja** je izdelati načrt programske rešitve na podlagi modelov in ostalih specifikacij, ki so bili izdelani v fazi analize. V fazi načrtovanja v ospredje stopi vprašanje, kako izpolniti v predhodni fazi dogovorjene zahteve. Poglavitne aktivnosti te faze so povezane z razvojem arhitekture programske rešitve. Zasnova arhitekture programske rešitve je bila sestavljena že v fazi analize, v fazi načrtovanja pa se arhitekturo še podrobneje preuči in dopolni. Analizirajo se morebitna tveganja ter se sestavi dokument morebitnih tveganj.

Naloga **faze izvedbe** je razviti programsko rešitev glede na načrt, izdelan v fazi načrtovanja. V sklopu faze izvedbe kreiramo fizično podatkovno bazo, generiramo programske module in izvedemo vse potrebne programerske posege na modulih. V sklopu obravnavane faze je potrebno še do konca izdelati dokumentacijo, izpeljati opravila testiranja do izvedbe potrditvenega testa, v primeru prenovitve izpeljati testno prevedbo podatkov, nadaljevati dela za področje prehoda na novo programsko rešitev oz. uporabe programske rešitve in pripraviti vse potrebno za začetek uvajanja uporabnikov v uporabo programske rešitve. Najprej se izdelajo detajlne izvedbene specifikacije. Pri tem sodelujejo vsi člani tima. Samo izvedbo pa prevzamejo člani, ki posredujejo ustrezno strokovno znanje in hkrati niso obremenjeni z drugimi aktivnostmi. O poteku izvedbe vodje razvojnih timov izdajajo sprotna periodična poročila. Timi, predvideni za testiranja, vzporedno razvijajo specifikacije testnih postopkov.

Naloga **faze vpeljave** je vpeljati programsko rešitev v uporabo v okolje, za potrebe katerega je bila razvita. V okviru obravnavane faze se v primeru prenovitve izvede prevedba podatkov in s tem zaključi aktivnost prevedbe podatkov. Izvede se potrditveni test, opravi se uvajanje uporabnikov in skrbnikov programske rešitve ter se v okviru aktivnosti uporabe programske rešitve pripravi vse potrebno za začetek uporabe nove programske rešitve.

Naloga **faze vzdrževanja** je spremljati delovanje programske rešitve, se odzivati na odkrite probleme v njegovem delovanju, izvajati nadgradnje in odpravljati napake ter planirati dodelave. Vzdrževanje je faza, ki se po vpeljavi programske rešitve v uporabo stalno odvija in je sestavni del razvoja programske rešitve.

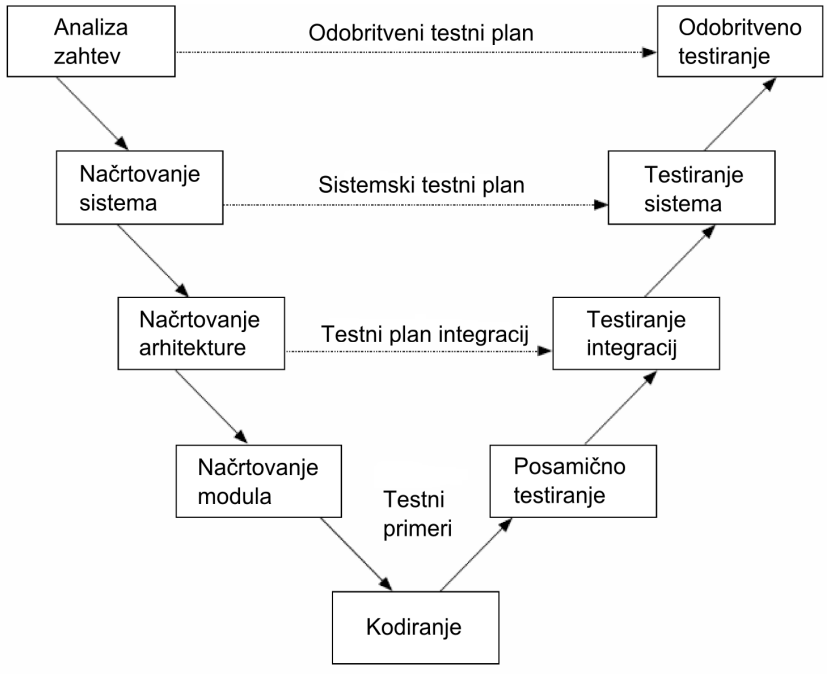
Morda ena največjih okoliščin, v kateri lahko zaporedni model razvoja programskih rešitev uspe, je zgodnja podrobna opredelitev zahtev. V prvi fazi je ključnega pomena, da so

zahteve dobro opredeljene in dokumentirane, saj se je tako lahko izogniti dragim napakam razvoja programskih rešitev zaradi napačne razlage. Stroški popravila odkrite napake v zgodnji fazi razvoja programskih rešitev so veliko nižja, kot pa če se odkrije in popravi napako v kasnejših fazah razvoja programskih rešitev (Dorfman & Thayer, 2000).

1.6.3 Model V

Model V je razširitev zaporednega modela. Za model V je mogoče reči, da je nastal kot rezultat razvoja programske rešitve in testiranja le te. Jasno so opredeljene različne tehnike testiranja, ki so ločene ene od druge ter skupaj z zaporednim modelom vodijo do modela V (Slika 3).

Slika 3: Model V



Vir: Student Accountant magazine, 2006.

Prva faza modela V je **analiza zahtev**. Na tej stopnji se mora opredeliti, kakšna je idealna programska rešitev, vendar v tej fazi ni mogoče točno določiti, kakšno programsko rešitev je potrebno razviti. Potrebno je narediti razgovore z uporabnikom ter priskrbeti ustrezne dokumente za razvoj programske rešitve. Ti dokumenti opisujejo sistem z vidika uporabnika. Uporabniki morajo opraviti skrbno študijo dokumentov, saj bodo služili kot vodič za razvoj programske rešitve. V tej fazi se preizkusi tudi sprejem programske rešitve s strani uporabnikov s pomočjo analize in opredelitve razvoja programske rešitve.

Druga faza se imenuje **načrtovanja sistema**. Na tej stopnji načrtovalci izvršijo analizo funkcij predlagane programske rešitve, ki temelji na zahtevah iz prejšnje faze. Opredelijo se možnosti in tehnike, preko katerih se bodo implementirale zahteve uporabnikov. Če zahteva ni izvedljiva, je potrebno obvestiti uporabnika, najti ustrezno rešitev v skladu z dokumentacijo ter posodobiti zahtevo. V tej fazi je potrebno tudi pripraviti dokumentacijo za preizkus programske rešitve.

Načrtovanje arhitekture programske rešitve je naslednja faza. Potrebno je izbrati arhitekturo programske rešitve, ki je sposobna, da izpolni zahteve v določenem času glede na stroške in vire. V arhitekturi programske rešitve je opisan model, ki je običajno sestavljen iz baz podatkov, uporabniških vmesnikov in poslovne logike. Podrobno so opisani tudi vsi moduli in komponente, ki sestavljajo model. V tej fazi je napisan dokument, ki vsebuje seznam modulov, njihovo povezavo s tabelo v podatkovni bazi, podrobnosti o tehnologiji itd. Opredeljen je tudi načina testiranja in integracije modula.

Naslednja faza je **načrtovanje modula**. Zdaj je programska rešitev zasnovan na manjše enote in vsaka od njih je podrobno opisana tako, da se lahko kodiranje programske rešitve začne.

Ob vseh razvojnih fazah programske rešitve tečejo paralelno tudi faze **testiranja** programske rešitve. Napake, ki se odkrijejo tukaj, so veliko cenejše za popravilo, kot če se odkrijejo pri uporabi sistema. Faze testiranja programske rešitve vključujejo analizo programske kode in odpravo napak, ki je v skladu s sprejetimi standardi razvoja programske rešitve. Preizkušanje programske rešitve se opravi na podlagi testov, ki so bili pripravljeni v posameznih fazah načrtovanja in analize programske rešitve. Pri preizkušanju programske rešitve se opravi najprej posamično testiranje določenega modula, nato se opravi testiranje celotne programske rešitve, na koncu pa se izvede še uporabniško testiranje končane programske rešitve. Ta preizkus se izvaja z dejanskimi podatki in delom ljudi, ki bodo pozneje uporabljali programsko rešitev.

1.6.4 Spiralni model

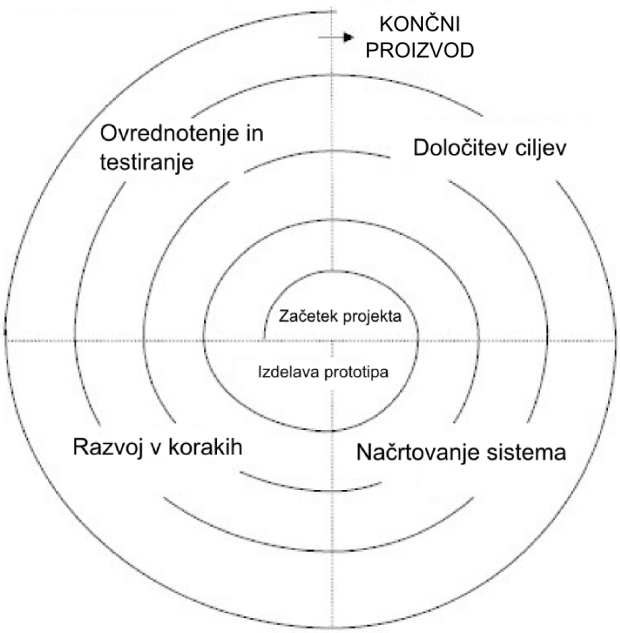
Čeprav zaporedni model ponuja urejeno strukturo za razvoj programskih rešitev, zahteve trga za skrajšanje časa razvoja naredijo model neprimeren. Naslednji klasični model razvoja programskih rešitev je spiralni model, ki združuje elemente zasnove programske rešitve in izdelavo prototipov v fazah, v prizadevanju, da združi prednosti konceptov zaporednega modela in V modela. Spiralni model je razvil Boehm (1988) na podlagi izkušenj z različnimi izboljšavami zaporednega modela za uporabo razvoja programske rešitve pri velikih projektih. Ta model se uporablja pri razvoju velikih programskih rešitev in je še posebej uporaben, ko se pri razvoju uporablja že obstoječa programska rešitev ter

ko se uporabljajo specifični cilji kakovosti programske rešitve, kar pripomore k natančni oceni tveganja med razvojem programske rešitve. To je odvisno od nadzora vseh faktorjev ter odpravo ali vsaj zmanjšanje zunanjih vplivov. Kot druga razširitev in izboljšava zaporednega modela je podajanje povratnih informacij iz prejšnjih stopenj. Spiralni model razvija programsko rešitev s postopno gradnjo končne različice programske rešitve, s pričetkom ob sredini spirale navzven.

Obstajajo štiri faze spiralnega modela (Slika 4):

- Določitev ciljev – opredeljeni so razumljivi cilji projekta;
- Načrtovanje sistema – projekt se pregleda in napravi se načrt razvoja;
- Razvoj v korakih – postopoma se razvija celotni sistem;
- Ovrednotenje in testiranje – ustrezen model je izbran in testiran.

Slika 4: Spiralni model.



Vir: B. W. Boehm, A Spiral Model of Software Development and Enhancement, 1988, str. 64.

Spiralni model razvoja programske rešitve odraža razmerja nalog s hitro izdelavo prototipov, povečanje vzporednosti pri oblikovanju in izgradnji dejavnosti. Vendar pa je pri spiralni metodi razvoja programskih rešitev še vedno potrebno metodično načrtovanje z opredeljenimi cilji in nalogami za vsak korak v spirali (Boehm, 1988). Z vsako zanko spirale uporabnik ocenjuje delo in poda predloge za naslednjo spremembo. Poleg tega se z vsako zanko spirale analizira tveganja pri razvoju programske rešitve. Če so določene nevarnosti prevelike, se projekt razvoja programske rešitve lahko tudi zaključi (Frankovich, 1998). Tako spiralni model razvoja programske rešitve obravnava problem

tehnološke zahteve z razvojem prototipov ter obravnava potrebo po obvladovanju tveganja z analizo tveganj v vsaki fazi razvoja programske rešitve.

1.6.5 Model RAD

Model hitrega razvoja programskih rešitev ali Rapid application development (RAD) je uvedel Martin (1991), s katerim je opisal metodologijo razvoja programskih rešitev, ki uporablja minimalno načrtovanje in iterativni razvoj v korist hitre izdelave prototipov. Razvoj prototipa pomaga analitikom in uporabnikom preverjati poslovne zahteve in formalno izboljšati postopke razvoja programskih rešitev. Model RAD je združitev različnih strukturiranih tehnik, še posebej na osnovi informacijskih podatkov in prototipnih tehnik za pospešen razvoj programskih rešitev (Whitten, Bentley & Dittman, 2004). Tako lahko podjetje z modelom RAD uspešna razvija programsko rešitev v nestabilnim okoljem in nejasnimi poslovnimi zahtevami.

Model RAD je bil odgovor na zaporedno metodologijo razvoja programskih rešitev, pri katerih so se lahko zaradi dolgotrajnega razvoja programskih rešitev zahteve spremenile, preden je bila programska rešitev v celoti razvita, kar pogosto povzroči neuporabno programsko rešitev (McConnell, 1996). Zato se model RAD pogosto uporablja v primerih časovne omejitve, v katerih ima hitrejši razvoj programskih rešitev prednost pred popolno funkcionalnostjo in učinkovitostjo. Model RAD lahko omogoči kompromis pri funkcionalnosti in zmogljivosti programske rešitve v zameno za hitrejši razvoj in lažje vzdrževanje programske rešitve (Martin, 1991). Model RAD razdeli razvoj programske rešitve na različne zahteve razvoja programske rešitve z različnimi prioritetami. To omogoči, da se v primeru pomanjkanja časa za razvoj programske rešitve zahteve z manjšo prioriteto premaknejo v naslednji časovni okvir razvoja programske rešitve (Keyes, 2002).

Struktura življenjskega cikla metode RAD je zasnovana tako, da zagotovi, da razvijalci gradijo programsko rešitev, ki jo uporabniki resnično potrebujejo. Tako model RAD vsebuje štiri faze (Keyes, 2002), ki vključujejo vse dejavnosti in naloge, potrebne za opredelitev obsega in poslovnih zahtev programske rešitve ter razvoj, izvajanje in uporabo programske rešitve (Slika 5).

Načrtovanje zahtev – ta faza opredeli poslovne funkcije in zahteve, ki jih bo programska rešitev podpirala.

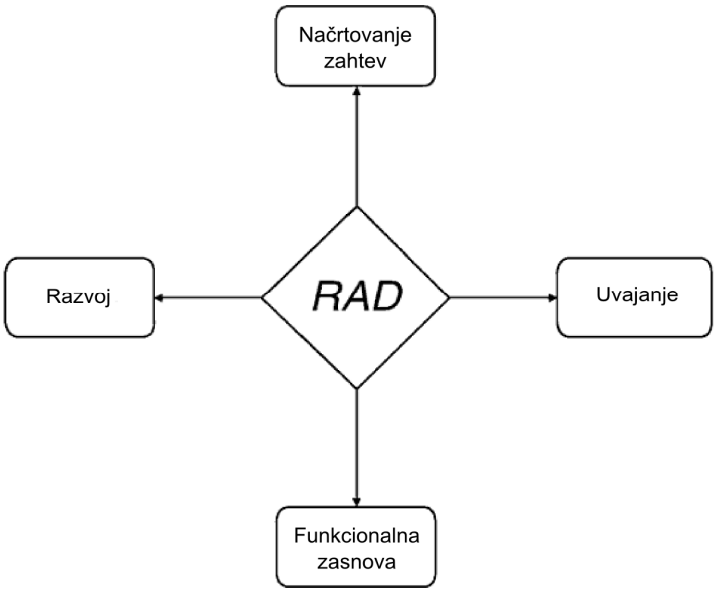
Funkcionalna zasnova – v tej fazi se razvije delujoči prototip programske rešitve, ki vsebuje modele procesov in zahtev končne programske rešitve.

Razvoj – ta faza zaključi razvoj programske rešitve s pomočjo razvoja in izboljšav prototipov. Prototip programske rešitve uporabniki preizkušajo in vračajo razvojnemu timu

za popravilo ali predelavo programske rešitve toliko časa, dokler se prototip ne razvije v končno programsko rešitev. Razvoj prototipa se tako uporablja za vizualno pomoč uporabnikom in zahtev programskih rešitev ter omogoča ponavljajoči razvoj programskih rešitev.

Uvajanje – ta faza vključuje preizkušanje in usposabljanje končnega uporabnika za pravilno uporabo končne programske rešitve.

Slika 5: Model RAD



Vir: J. Keyes, Software Engineering Handbook, 2002, str. 132.

1.7 Agilne metodologije

Agilni pristop se je na področju metodologij razvoja informacijskih sistemov pojavil kot posledica kritičnega pogleda na celovite, večinoma težko obvladljive metodologije, ki proces razvoja predpišejo do vsake najmanjše potankosti. Agilni pristop se je razvil kot odziv na pomanjkljivosti zaporednega razvoja programskih sistemov. Kljub kakovosti, ki jo takšne metodologije navadno dosegajo, je njihova uporaba v praksi omejena, saj jim z željo po čimprejšnjih rezultatih težko sledimo. Novi trendi, ki jih povezujemo s pojmom agilnost, priporočajo zasnovo ter uporabo prilagodljivih metodologij, ki so »ravno dovolj« predpisane ter hitro in enostavno prilagodljive. Osnovna načela agilnih metodologij razvoja programskih rešitev obsegajo kvalitetno kodo, učinkovitosti ljudi skupaj z »dobro voljo« in osredotočenje na timsko delo (Ambler, 2002).

Spremenljivo okolje vpliva na poslovne procese razvoja programskih rešitev. Agilni razvoj programskih rešitev prilagodi razvoj tako, da zadovolji uporabnikove zahteve v trenutku začetka projekta ter se pripravi na neizogibne spremembe v razvoju programskih rešitev tekom trajanja projekta (Highsmith & Cockburn, 2001). Pri agilnih metodologijah razvoja programskih rešitev niso pomembne nove prakse, ki se uporabljajo, ampak priznanje, da je tim primarni dejavnik uspeha projekta razvoja programskih rešitev, skupaj s poudarkom na intenzivno učinkovitostjo in manevrski prostor pri razvoju programskih rešitev (Highsmith & Cockburn, 2001).

Fowler (2006) meni, da obstajata dve ključni značilnosti agilnih metodologij pri razvoju programskih rešitev, iz katerih izvirajo vse bistvene razlike:

- agilne metodologije razvoja programskih rešitev se raje prilagajajo kot predvidevajo. V klasičnem pristopu razvoja programskih rešitev je večji del razvoja načrtovan do podrobnosti za daljše časovno obdobje razvoja programskih rešitev. To deluje, dokler se zahteve in okolje ne spremenijo. Njihova narava je torej taka, da se upirajo spremembam. Agilne metodologije razvoja programskih rešitev pa sprejemajo spremembe, jim prilagajajo proces razvoja programske rešitve in se tudi same lahko spreminjajo.
- agilne metodologije razvoja programskih rešitev so bolj usmerjene k ljudem kot k procesom razvoja programskih rešitev. Ideja metodologij, ki so procesno usmerjene, je, da se definira proces, ki deluje ne glede na to, kdo ga izvaja. Agilne metodologije razlagajo, da proces ne more nadomestiti veščin razvojne skupine. Proces naj le podpira razvijalce pri njihovem delu.

1.7.1 Nastanek agilnih metodologij razvoja programskih rešitev

Leta 2001 so se v mestu Snowbird (Utah, USA) zbrali izkušeni inženirji razvoja programskih rešitev in prvič uporabili ime agilni model. Nekateri od njih so kasneje ustanovili neprofitno organizacijo »The Agile Alliance«, katere cilj je promoviranje agilnega razvoja programskih rešitev. Avtorji agilne metodologije razvoja programskih rešitev so ustvarili manifest agilnih metodologij, ki predpisuje principe in postopke, ki bi jih morale upoštevati vse agilne metode razvoja programskih rešitev. Agilne metode razvoja programskih rešitev odkrivajo boljše načine razvoja programskih rešitev. V tej dejavnosti agilne metodologije razvoja programskih rešitev vrednotijo (Manifesto for Agile Software Development, 2009):

- bolj posameznike ter interakcije med njimi kot procese in orodja,
- bolj delujoč izdelek kot temeljito dokumentacijo,
- bolj sodelovanje z uporabnikom kot pogajanje na osnovi pogodbe,
- bolj odzivnost na spremembe kot sledenje planu.

Pristop na osnovi zaporednega modela razvoja programskih rešitev predpisuje bolj ali manj strogo določen proces, v katerem posamezniki nastopajo v točno določenih vlogah. Posamezniki v tem pogledu niso pomembni, pomembne so le vloge, ki jih opravljajo. Posameznik je nadomestljiv. Vloge so določene in v primeru, da nekdo zapusti svoje mesto, lahko na mesto njega v vsakem trenutku vstopi nekdo drug. Agilni razvoj programskih rešitev temu nasprotuje in poudarja pomembnost posameznikov. Dober proces projekta ne reši pred neuspehom, kadar v moštvu ni dobrih igralcev, slab proces pa lahko onemogoči tudi moštvo najmočnejših igralcev. Skupina močnih igralcev lahko pričakuje neuspeh tudi v primeru, da ne zna delovati kot moštvo (Martin, 2003). Agilni razvoj tako daje prednost posameznikom, ki komunicirajo bolje, ti pa niso nujno tudi tisti, ki so tehnično najbolj sposobni. V tem pogledu agilni razvoj, namesto da bi predpisoval okolje (procese in orodja), v katerem naj delujejo posamezniki, poudarja samoorganiziranost razvojnega tima. To pomeni, da je timu prepuščeno, da si samo izbere okolje, ki mu najbolj ustreza glede na značilnosti članov tima in značilnosti naloge, ki se mora opraviti.

Agilni razvoj programskih rešitev priznava pomembnost dokumentacije in obenem opozarja na slabosti in neučinkovitosti, ki nastanejo ob pretiranem poudarku na dokumentiranju sistema. Ključno vprašanje je tako, koliko dokumentacije je pravzaprav smiselno imeti. Obsežni dokumenti vzamejo ogromno časa za izdelavo in še več časa za redno osveževanje. Agilni razvoj predlaga dokumentacijo, ki je kratka in jedrnata. Pomembna pa je tudi komunikacija med člani razvojnega tima, ki nadomesti odvečno dokumentacijo.

Agilni razvoj programskih rešitev v ospredje postavlja sodelovanje z uporabnikom, kjer je pogodbeno razmerje šele drugotnega pomena in pri izjemno tesni povezanosti kupca in izvajalca morda sploh odveč (Cockburn, 2002). Sodelovanje zagotavlja izvajalcu reden tok povratnih informacij. Izvajalec se tako ne zanaša le na pogodbo in začetne zahteve, pač pa razvoj sistema sproti prilagaja dejanskim potrebam in zahtevam naročnika.

Več dejavnikov lahko povzroči, da plan, ki ni prilagodljiv, postane zastarel. Poslovno okolje se nenehno spreminja, kar posledično spremeni funkcionalne zahteve naročnika. Funkcionalne zahteve se bodo največkrat spremenile, ko naročniki vidijo sistem v delovanju. Tudi kadar so zahteve dane in smo prepričani, da se ne bodo spremenile, le s težavo ocenimo, koliko časa bo trajal razvoj. Strategija planiranja naj bo takšna, da se planira podrobno za kratko obdobje in zgolj površinsko za daljša obdobja. Različne agilne metodologije določajo različno dolga časovna obdobja, razvojne cikle, znotraj katerih poteka fiksno planiranje.

1.7.2 Značilnosti agilnih metodologij razvoja programskih rešitev

Agilne metodologije razvoja programskih rešitev so skupina ponavljajočih in evolucijskih metod in temeljijo na iterativnem in oportunistične razvoju programskih rešitev. Poglavitna razlika med agilnimi metodologijami in klasičnimi metodologijami razvoja programskih rešitev je dolžina vsake ponovitve razvoja programske rešitve. Pri klasičnih metodologijah razvoja programskih rešitev traja lahko iteracija med razvojem programske rešitve tri ali šest mesecev. Z uporabo agilnim metodologij razvoja programskih rešitev pa ponovitve trajajo od enega do štiri tedne in načeloma ne presegajo 30 dni. Raziskave so pokazale, da se pri krajših iteracijah razvoja programskih rešitev pojavijo manjša tveganja, manjša je kompleksnost programske rešitve, boljše so potratne informacije, s tem pa tudi večja produktivnost in uspešnost projekta (Larman, 2004).

Agilne metodologije tako spodbujajo hitro vidne rezultate, odzivnost na spremembe, osredotočenost na prioritetne naloge, izvajanje rednih kontrol ter zavedanje znanja v organizaciji. Treba je poudariti, da pojem agilnost ne pomeni hitrosti razvoja na račun kakovosti ali minimalnega opisa procesa. Je le zmožnost organizacije, da se ustrezno prilagaja spremembam v okolju ter zahtevam tega okolja. Cilj uporabe agilnih metodologij je skrajšanje časa, znižanje stroškov in povečanje fleksibilnosti razvoja in uvedbe programske rešitve ob pogoju, da razvita programska rešitev uspešno in kakovostno deluje. Dejavniki uspeha so pri vsem tem verjetno najbolj odvisni od velikosti projekta. Ker projekt raste, postane ustna komunikacija težja. To pomeni, da je projekt bolj uspešen z uporabo agilne metode, v majhnih timih pod 20 ljudi (Cockburn, 2002).

Najbolj priljubljeni modeli razvoja programskih rešitev na področju agilnih metodologij so model XP, model SCRUM, model FDD, model DSDM in prilagodljiv razvoj programske opreme.

1.7.3 Model XP

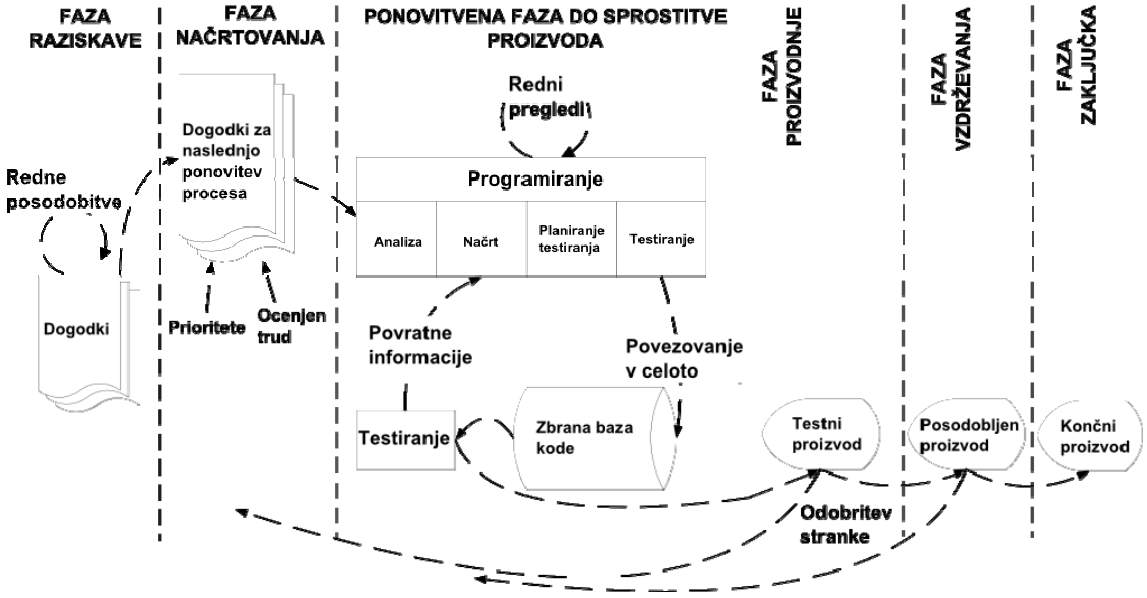
Najbolj znani agilni model razvoja programskih rešitev je ekstremno programiranje ali model XP (angl. *»Extreme Programming«*). Model XP se je razvil iz težav daljšega cikla razvoja programskih rešitev, ki so jih povzročale klasične metodologije razvoja programskih rešitev (Beck, 1999a). Model XP je mogoče označiti s kratkimi razvojni cikli, primarnim načrtovanjem, stalne povratne informacije in komunikacijo. Z vsemi zgoraj navedenimi lastnostmi se XP programerji odzivajo na spreminjajoče se okolje z veliko več poguma. Člani skupine XP porabijo skupaj dnevno nekaj minut za programiranje, nekaj minut za upravljanje projekta, nekaj minut pri oblikovanju, nekaj minut za povratne informacije in nekaj minut za oblikovanje in grajenje timskih odnosov (angl. *»team*

building«) (Williams, 2003). Ena od temeljnih idej modela XP je, da ne obstaja ena metodologija razvoja programskih rešitev, ki bi bila uporabna v vsakem projektu, ampak obstajajo prakse, ki bi morale biti prilagojene potrebam posameznih projektov (Beck, 1999b).

Beck (1999b) navaja šest razvojnih faz projekta po modelu XP (Slika 6):

- raziskovanje,
- načrtovanje,
- ponavljanje do sprostitve,
- proizvodnja,
- vzdrževanje in
- fazo zaključka.

Slika 6: Model XP



Vir:K. Beck, Extreme Programming Explained, 1999, str. 132.

V fazi raziskovanja stranka napiše svoje želje, ki se bodo upoštevale in vključile v razvoj programske rešitve. To vodi do faze načrtovanja, kjer se določi prednostni vrstni red želja do vsakega uporabnika ter razpored prve sprostitve produkta. Nato v fazi ponavljanja do sprostitve razvojni timi ustvarijo arhitekturo celotne programske rešitve, nato nenehno dodajajo, kodirajo in preizkušajo svoj razvoj. Dodatno testiranje in preverjanje uspešnosti programske rešitve, preden se programska rešitev lahko dostavi stranki, se opravi v fazi proizvodnje. Preložene ideje in predloge v fazi proizvodnje so dokumentirane za poznejšo izvedbo posodobitve programske rešitve za uporabnike v fazi vzdrževanja. Zadnja faza je faza zaključka programske rešitve, ko stranka nima več želja, ki se izvajajo, vsa potrebna

dokumentacija programske rešitve je zapisana, sprememb v arhitekturi, oblikovanju ali kodi ni več.

Eden od ključnih ciljev modela XP je zmanjšati stroške sprememb in popravkov v procesu razvoja programske rešitve. Spremembe v modelu XP so predvidene kot neizogibne, normalne in so celo zaželen del procesa. Model XP prepozna niz petih kritičnih vrednosti razvojne uspešnosti:

- komunikacija,
- preprostost,
- povratna informacija,
- pogum in
- spoštovanje.

Komunikacija se nanaša na sporočanje zahtev uporabnika, širjenje projektnih znanj med člani skupine in delitev rezultatov in idej z uporabniki v številnih interakcijah. Model XP priporoča začetek razvoja programskih rešitev na podlagi najenostavnejšega načina ter preoblikovanje v bolj kompleksno rešitev le, če je to zahtevano. Ta poudarek razvoja programske rešitve in oblikovanja samo za današnje potrebe je eden od ključnih razlik med modelom XP in drugimi klasičnimi modeli. Povratna informacija se nanaša na vse udeležence pri razvoju programskih rešitev. Pogum in spoštovanje izvirata iz timskega dela in nimata neposredne vrednosti v razvojni metodi, vendar pa je potrebno, da se člani tima spoštujejo in zavedajo, kako pomembne so spremembe ter spremembe vpeljejo v projekt hitro in kakovostno.

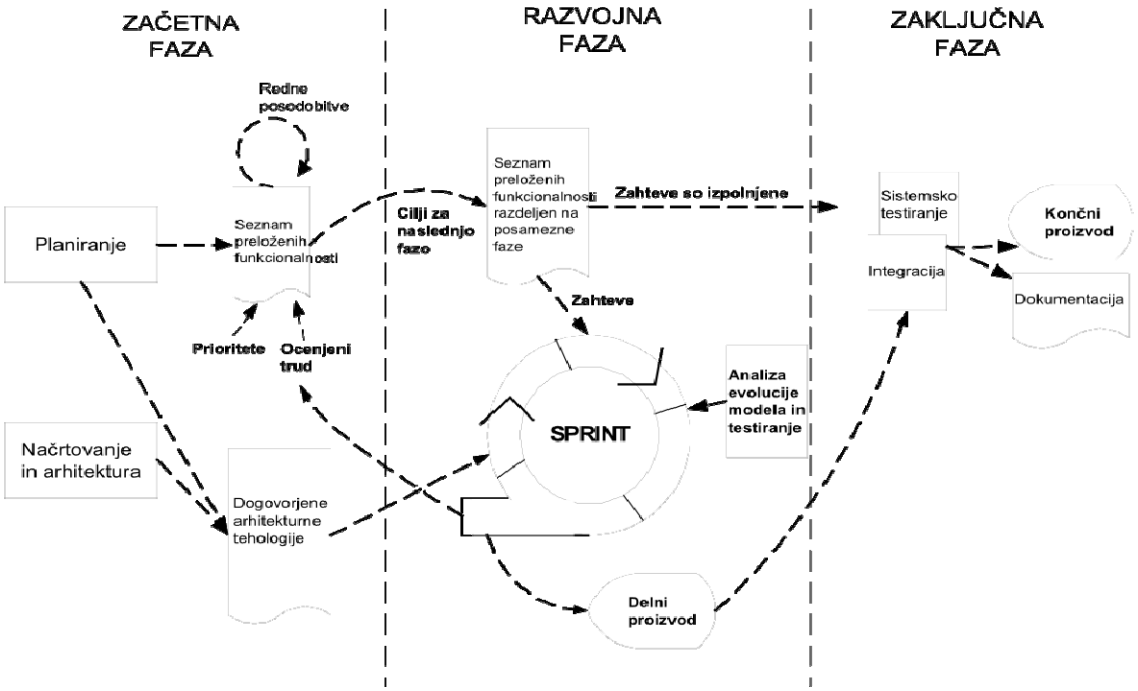
Model XP nikakor ni primeren povsod, pa tudi še vse meje niso bile ugotovljene in raziskane v praksi (Beck, 1999b). Model XP zahteva več empiričnih in eksperimentalnih raziskav na tem področju iz različnih perspektiv. Ugotovljene pa so bile nekatere omejitve na področji razvoja programskih rešitev in uporabe modela XP v praksi. Model XP je namenjen za majhne in srednje velike time. Beck (1999b) predlaga tim, ki je omejen med tri in največ dvajset članov v projektu. Komuniciranje in usklajevanje med člani projekta morata biti omogočena ves čas. Beck (1999b) navaja, da je razpršenost programerjev v dveh nadstropjih ali celo razpršenost v enem nadstropju nesprejemljiva za model XP. Vendar pa geografska porazdelitev timov ni nujno zunaj obsega modela XP v primeru, da so timi, ki sodelujejo na projektu, povezani z omenjeno interakcijo (Beck, 1999b). Poslovna kultura tudi vpliva na uspešnost modela XP projekta. Vsak odpor proti praksam in načelom metode v imenu projekta, članov, uprave ali odjemalcev lahko pripelje do tega, da projekt propade. Prav tako bi lahko tehnologija onemogočila uspeh projekta po modelu XP. Na primer, tehnologija, ki ne podpira zahtevanega časa povratne informacije ali hitrih sprememb, ni primerna za model XP.

1.7.4 Model SCRUM

Model SCRUM za razvoj programskih rešitev izvira iz hitre izdelave prototipov, ki naj bi podpirala okolje, v katerem so bile zahteve ne samo nepopolne na začetku, vendar so se lahko tudi zelo hitro spreminjale. Z razliko med modelom XP model SCRUM na prvo mesto vključuje vodstveni in ne razvojni proces, ki naj bi za razliko od težkih metodologij dramatično povečal produktivnost v timu. Model SCRUM je empirična tehnika vodenja, za katero je realnost pomembnejša od načrtov, potrebno je nenehno sodelovanje, zaupanje v ljudi in napredek (Schwaber & Beedle, 2002). SCRUM temelji na samoorganizaciji, v kateri se tim odloči, kaj storiti, medtem ko vodstvo vodi in odpravlja težave.

Model SCRUM združuje vse predhodno naštete skupne lastnosti agilnih metodologij razvoja programskih rešitev, njena posebnost pa je organizacija ter izvajanje vodenja in hkrati obvladovanje projektnega tima. Uporaba modela SCRUM temelji na realizaciji realno mogočega, dnevnega preverjanja napredka in pogostih korektur. Bistvo so majhni projektni timi (do deset ljudi), serija »sprintov« (eden do štirje tedni), vidne in uporabne nadgradnje ter časovno omejevanje. Razvoj programske rešitve se zaključi, ko je uporabnik zadovoljen z napredkom in ustreznostjo programske rešitve. Po končani programski rešitvi nato tim izvede testiranje, usposabljanje in napiše potrebno dokumentacijo, ki je potrebna za sprostitev programske rešitve. Ključne faze modela SCRUM so predstavljene in prikazane na Sliki 7 (Abrahamsson, Salo & Ronkainen, 2002).

Slika 7: Model SCRUM



Vir: P. Abrahamsson, O. Salo & J. Ronkainen, Agile software development methods, 2002, str. 28.

Začetna faza razvoja programske rešitve vsebuje dve razvojni stopnji, in sicer načrtovanje arhitekture in modeliranje programske rešitve. Načrtovanje programske rešitve vključuje opredelitev programske rešitve, ki se razvija, uvrstitev zahtev in seznam preloženih produktov. To je seznam vseh funkcij in spremembe, ki jih je potrebno razviti. Ta seznam so želje in cilji več akterjev, kot so uporabniki, marketing in projektne skupine. Lastnik programske rešitve je odgovoren za vzdrževanje preloženih produktov in nedokončanega dela. Seznam preloženih produktov se stalno posodablja z novimi in bolj podrobnimi predmeti ter tudi z bolj natančnimi ocenami in nalogami. Načrtovanje vključuje tudi opredelitev projektne skupine, orodij in drugih virov, oceno tveganja in prioritet ter potreb po usposabljanju. Modeliranje programske rešitve se napravi s pomočjo seznama preloženih produktov. Tukaj se predvidijo spremembe obstoječe programske rešitve skupaj s težavami, ki jih sprememba lahko povzroči.

Razvojna faza je tako imenovani agilni del modela SCRUM. Ta faza se obravnava kot nepredvidljivo. Pričakujejo se razne okoljske in tehnične spremenljivke, kot so časovni okvir, kakovost, zahteve, viri, tehnologije in orodja za izvajanje ter celo razvojni model. Namesto da bi se te spremenljivke upoštevale na začetku razvoja programske rešitve, je model SCRUM namenjen nadzoru nad njimi ves čas, da se lahko prožno prilagodijo spremembam. V razvojni fazi se sistem razvija s pomočjo serij kratko ponavljajočih se faz, razdeljenih na razvojne procese (sprint). Pred vsako fazo člani tima ugotovijo in postavijo delo za to fazo ter v poteku faze preverjajo delovanje in razvoj faze. Na koncu faze se temeljito pregleda in preveri skupni razvojni napredek.

Zaključna faza se je začela, ko je bil sklenjen dogovor, da so okoljske in tehnične spremenljivke izpolnjene. Izvedeno je bilo testiranje, napisana je bila potrebna dokumentacija programske rešitve. Programska rešitev je sedaj pripravljena za sprostitev.

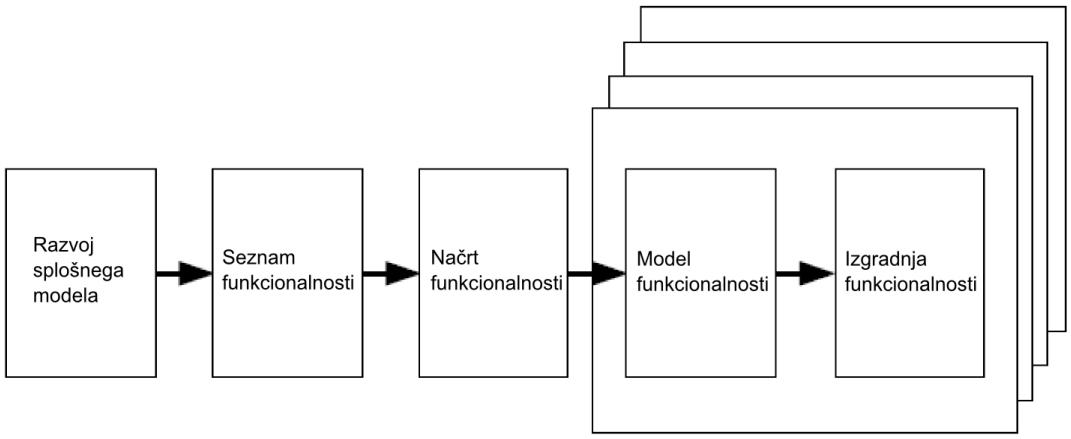
1.7.5 Model FDD

Funkcionalno voden razvoj programskih rešitev ali model FDD (angl. *»Feature Driven Development«*) za razliko od drugih modelov ne zajema celotne programske rešitve razvojnega procesa, temveč se osredotoča na načrtovanje in gradbene faze (Palmer & Felsing 2002). Model FDD pooseblja iterativni razvoj z najboljšimi praksami, ki se je izkazal za učinkovito metodo predvsem v industrijski dejavnosti. Poudari vidike kakovosti v celotnem procesu in vključuje pogoste in opredmetene dobave programskih rešitev, skupaj z natančnim spremljanjem napredka projekta.

Model FDD je sestavljen iz petih zaporednih postopkov in določa metode, tehnike in smernice, ki jih potrebujejo sodelujoči v projektu za razvoj programske rešitve (Slika 8). Za razliko od nekaterih drugih agilnih modelov model FDD trdi, da je primeren za razvoj

kritičnih sistemov (Palmer & Felsing 2002). Prve tri faze se opravi na začetku projekta. Zadnji dve fazi sta ponavljajoči del procesa, ki podpira agilni razvoj programske rešitve skupaj s hitro prilagodljivostjo programske rešitve spremembam in poslovnim zahtevam.

Slika 8: Model FDD



Vir: S. R. Palmer & J. M. Felsing, A Practical Guide to Feature-Driven Development, 2002, str. 57.

Razvoj splošnega modela – razišče in določi se obseg programske rešitve in kontekst delovanja članov razvojnega tima ter napiše se vsa potrebna dokumentacija s področja uporabe in funkcij programske rešitve.

Seznam funkcionalnosti – izdelan je seznam funkcionalnosti programske rešitve za podporo uporabniških zahtev.

Načrt funkcionalnosti – razvojna skupina določi funkcionalnost glede na njeno prioriteto in odvisnost ter jo dodeli določenim razvijalcem. Določi se tudi razpored in mejniki razvoja funkcionalnosti programske rešitve.

Model funkcionalnosti in izgradnja funkcionalnosti – to je iterativni postopek, v katerem tim ustvarja diagrame zaporedja za dodeljene funkcije. Te sheme se prenesejo na razvijalce, ki izvajajo elemente, potrebne za podporo oblikovanja določene funkcije programske rešitve. Lahko je več različnih timov, ki istočasno načrtujejo in gradijo lasten nabor funkcionalnosti. Razvito kodo se nato pregleda in preizkusi. Po uspešnem testiranju se funkcionalnosti zaključijo in se prestavijo v končno programsko rešitev.

1.7.6 Model DSDM

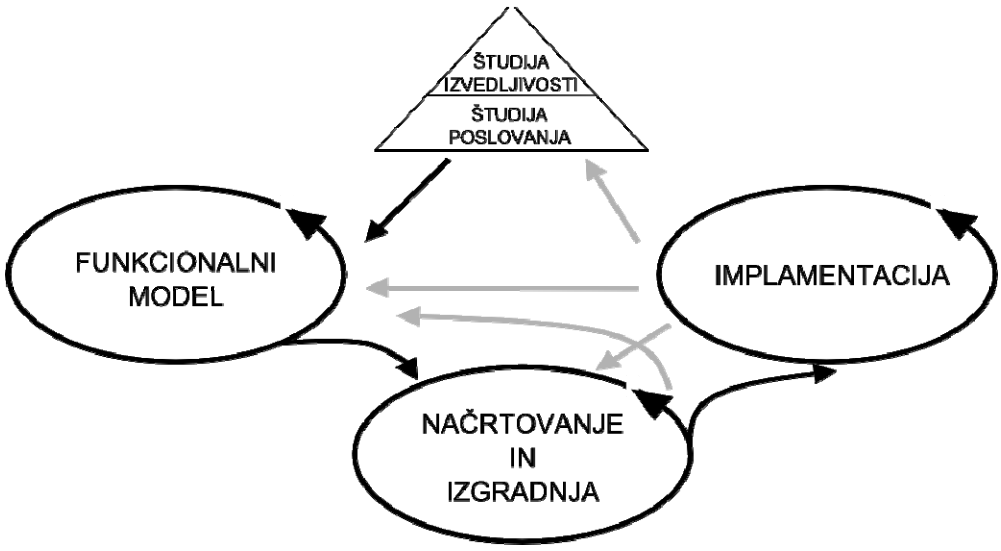
Dinamični sistemski model razvoja programskih rešitev ali model DSDM (angl. *»Dynamic System Development Method«*) je bil razvit v Veliki Britaniji v sredini leta 1990. Temeljna zamisel modela DSDM je določiti čas in sredstva in nato prilagodi funkcionalnost programske rešitve.

Model DSDM je sestavljen iz petih faz (Slika 9):

- študija izvedljivosti,
- študija poslovanja,
- funkcionalni model,
- načrtovanje in izgradnja ter
- implementacija.

Prvi dve fazi sta zaporedni in se opravita samo enkrat. Zadnje tri faze, med katerimi je dejansko razvojno delo, se ponavljajo do zaključitve projekta. Čas trajanja projekta je vnaprej določen. V modelu DSDM je tipično obdobje trajanje projektnega cikla nekaj dni do nekaj tednov.

Slika 9: Model DSDM



Vir: J. Stapleton, Dynamic systems development method – The method in practice, 1997, str. 3.

Študija izvedljivosti – v tej fazi je sprejeta odločitev, ali uporabiti model DSDM za razvoj programske rešitve ali ne. Ta je določena s pomočjo presojanja tipa projekta, organizacije v podjetju in med ljudmi. Poleg tega sta narejena poročilo o izvedljivosti in načrt razvoja programske rešitve.

Študija poslovanja – priporočeni pristop te faze je organizirati delavnice za razumevanje poslovnega področja projekta, opredelitev programske rešitve in načrt prototipa.

Funkcionalni model – ta faza vključuje analizo, kodiranja in razvoj prototipov. Ključen izid te faze je funkcionalen model, ki je sestavljen iz prototipa kode in analize modelov.

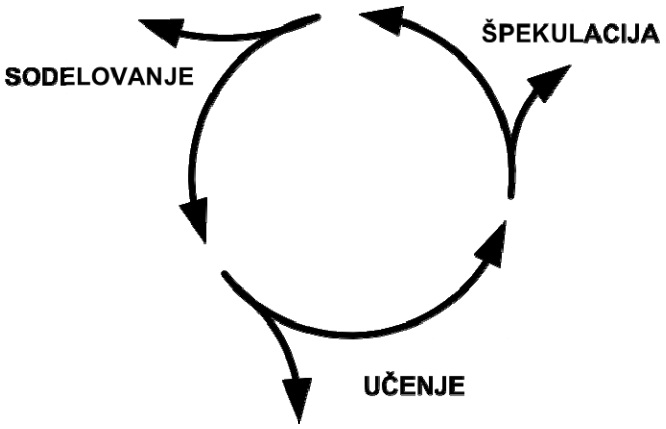
Načrtovanje in izgradnja – v tej fazi je programska rešitev zgrajena s pomočjo načrtovanja in funkcionalnih prototipov ter na pripombah in predlogov uporabnikov.

Implementacija – v tej končni fazi se končana programska rešitev izroči uporabnikom. Po končani fazi lahko vzdrževanje programske rešitve razumemo kot nadaljnji razvoj. Namesto zagona novega projekta se lahko programska rešitev vrne na eno izmed faz ter s pomočjo modela ponovitev razvijemo novo programsko rešitev.

1.7.7 Model ASD

Prilagodljivi razvoj programskih rešitev ali model ASD (angl. *»Adaptive Software Development«*) je razvil Highsmith (2000). Model ADS ponuja prilagojen pristop k razvoju programskih rešitev v projektih z velikimi in hitrimi spremembami. V hitro napredujočem in nepredvidljivem poslovnem okolju ni mogoče uspešno načrtovati projekta, tako da je v modelu ASD statični plan zamenjan z dinamičnim, špekulativnim in učečim se življenjskim ciklom razvoja programske rešitve. Model ASD je sestavljena iz treh nelinearnih in prekrivajočih se faz (Slika 10):

Slika 10: Model ASD



Vir: J. A. Highsmith, Adaptive Software Development, 2000, str. 18.

Špekulacija – za določitev poslanstva projekta se je potrebno zavedati, kaj je v projektu nejasno in neznano.

Sodelovanje – poudarja pomen timskega dela v projektih z velikimi in hitrimi spremembami.

Učenje – ta faza poudarja, da je potrebno priznati in se odzvati na napake ter da se lahko zahteve tudi spremenijo med razvojem programske rešitve.

V nepredvidljivem okolju morajo sodelujoči med sabo tesno sodelovati, v kar jih vodstvo tudi spodbuja. Highsmith (2000) poudarja, da je v prilagodljivem okolju učenje izziv za vse zainteresirane strani (razvijalce in njihove uporabnike). V klasičnem načrtovanju, ko stvari ne gredo po načrtu, se to obravnava kot napaka, ki jo je treba popraviti. Vendar pa nas v prilagodljivem okolju odstopanja vodijo do pravilne rešitve. Model ASD se bolj osredotoča na rezultate in njihovo kakovost kot pa na naloge ali procese, ki se uporabljajo za razvoj programskih rešitev. Tako je učenje stalen in pomemben element pri uspešnosti projekta. Model ASD nima podrobnih načel, kot jih ima model XP, ampak zagotavlja okvir o tem, kako spodbuditi sodelovanje in učenje v določenemu projektu. Model ASD ni predstavljen kot metodologija za razvoj programskih rešitev, ampak kot pristop ali odnos, ki ga sprejmejo organizacije pri uporabi agilnih procesov.

2 ORGANIZIRANJE RAZVOJNEGA PODJETJA PROGRAMSKIH REŠITEV

Pri razvoju programskih rešitev je način organizacije podjetja ključnega pomena za uspešnost podjetja. S korenitimi spremembami na področju organizacije podjetja je možno, vsaj v nekaterih dejavnostih, bistveno izboljšati uspešnost podjetja. Virtualne organizacije s pomočjo sodobne tehnologije in virtualnih timov, ki se oblikujejo znotraj organizacije, poslujejo z večjo učinkovitostjo in z manjšimi stroški. Predvsem pa bi lahko rekli, da so virtualni timi ustanovljeni zato, da premagajo geografske in časovne prepreke (Cascio & Shurygailo, 2002). Toda nekatere tehnologije, izdelki, storitve in celo organizacijske oblike se postopoma ekonomsko izčrpajo in ne omogočajo več rasti dodane vrednosti. Zaradi konkurence na trgu pa taka podjetja ne morejo ostati na vrhuncu, ampak začnejo nazadovati in na koncu izginejo s trga, če se kljub sodobni organizacijski ureditvi neprestano ne razvijajo in učijo.

Vodenje v virtualnem timu je izraženo skozi tehnologijo, zato morajo vodje timov razumeti tehnologijo, da jo lahko koristno uporabijo. Različne ideje o uporabi tehnologije omogočajo različno razumevanje tehnologije, predvsem v odnosu do dela, ki ga opravlja tim. Vsaka ideja lahko ponudi različne rešitve pri usklajevanju specifičnih značilnosti tehnologije in naloge, ki pomagajo pri razumevanju tega, kako priti do boljših rezultatov ter razvoju razmišljanja v timu in izrazu komunikacije.

2.1 Timsko delo

Ljudje se že od samega začetka povezujejo v skupine, ki so temelj človekovega družinskega življenja. Obnašanje skupin je lahko popolnoma kaotično, lahko pa ga z organiziranjem povzdignemo do popolne urejenosti, ki zagotavlja uspeh. Ljudje, ki se udeležujejo projektov v virtualnih organizacijah, prispevajo svoja znanja in izkušnje, ki pripomorejo k uresničitvi cilja, kot so:

- tehnična oz. funkcionalna znanja,
- sposobnosti reševanja problemov in sposobnosti odločanja in
- medčloveške sposobnosti in sposobnosti interakcije.

Vse bolj postaja jasno, da so skupine najuspešnejše, če jih uspemo preoblikovati v učinkovitejše enote, ki jih imenujemo timi (Maddux, 1992). Daft (1991) pojmuje tim kot enoto dveh ali več oseb, ki medsebojno sodelujejo z namenom doseči določen cilj. Možina (1994) opredeli tim kot skupino, za katero je značilno, da njeni člani sodelujejo pri odločanju in v medsebojni pomoči pri opredeljevanju in doseganju ciljev. Lipovec (1987) pravi, da je tim skupina delavcev iz različnih strokovnih področij, ki zelo enakopravno sodelujejo v izvajanju raziskovalne naloge. Enakopravno sodelovanje je pri takih nalogah nujno, ker je le tako možno doseči, da vsi sodelujoči delajo ustvarjalno, kar pa je pogoj za izvedbo naloge in uresničitve cilja.

Tim je skupina ljudi. Večina avtorjev se strinja, da je to skupina od 5 do 10 ljudi, ne več kot 15 metrov narazen, ki so zbrani skupaj z določenim razlogom, da rešijo določen projekt oz. dosežejo določen cilj (Lipičnik, 2002). Pri gradnji učinkovitih timov se je treba zavedati, kakšne lastnosti (sposobnosti, osebnost, motivacijo) posameznik prispeva k timu. Pogosto se pozablja, da ravno raznolikost teh lastnosti prispeva k moči in fleksibilnosti tima. V uspešnem timu igra vsak predpisano vlogo, tako da v največji možni meri uveljavi svoj talent in znanje. Ko člani združijo vse svoje sposobnosti, da bi s tem pokazali svojo premoč in čim bolj zmanjšali vpliv svojih šibkih lastnosti, so cilji tima v večini doseženi. Pri delu v timu so torej ljudje skupaj, imajo veliko sestankov, se skupaj odločajo in poskušajo doseči zastavljeni cilj. Vendar pa podjetja poslujejo z ostalimi podjetji z različnih geografskih območij, drugih jezikov, kultur, kar lahko prinaša s seboj veliko težav, ki jih je treba premagati. Za vse udeležence tima tudi ni mogoče, da imajo redne sestanke zaradi razdalj, časa in s tem povezanih stroškov. Zaradi tega se v sodobni organizaciji tradicionalni timi preoblikujejo v virtualne time.

2.1.1 Nastanek virtualnih timov in njihovo vodenje

Virtualne time lahko definiramo kot skupino posameznikov, ki so geografsko, organizacijsko ali kako drugače razkropljeni in sodelujejo s pomočjo komunikacijskih in

informacijskih tehnologij, da bi dosegli določen cilj. Ta definicija pomeni, da imajo virtualni timi skupen cilj, ki se zanaša na tehnologijo (Mowshowitz, 1994).

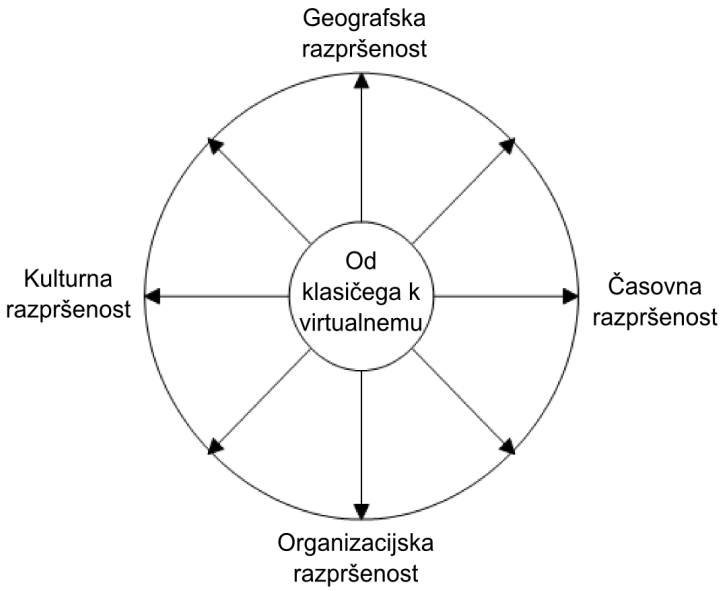
Najpomembnejše med virtualnimi timi je komunikacija, v kateri pa se pojavljajo tudi ključni problemi pri uspešnosti virtualnega tima (Gould, 2005):

- kakšno tehnologijo uporabljati, da so komunikacijski kanali uspešni,
- ali so vloge in odgovornosti definirane dovolj nazorno,
- kako obvladati vse vrste različnosti, na primer kulturo, jezik, čas,
- kako zagotoviti aktualne informacije vsakemu, ki jih potrebuje,
- kako zagotoviti sinergijo, (samo) motivacijo, učenje, zaupanje, prilagajanje,
- kako zagotoviti dovolj učinkovito planiranje in kontrolo.

Na virtualni tim lahko gledamo kot na okolje, v katerem lahko vsakdo sodeluje in ima enakovredno vlogo. Komunikacijska tehnologija lahko igra vlogo izenačevalca in daje vsakemu članu tima možnost sodelovanja. Tehnologija lahko izravna razlike v položaju in tudi to, kar je pomembno pri osebnem stiku, kot so videz, nastop, ton glasu.

Virtualni timi se pojavljajo v različnih oblikah in virtualnost se kot karakteristika lahko definira v različnih dimenzijah. Posebej določena točka, na kateri tim postane virtualni tim, ne obstaja. Vodstvo mora oceniti vsebino dela tima in stopnjo virtualnosti glede na različne dimenzije (Slika 11). S čim več vidikov in dimenzij je tim razdeljen, tem bolj virtualen je. Krog, ki se širi, nakazuje, da bolj virtualen kot postaja tim, kompleksnejša postajajo vprašanja, s katerimi se mora soočati, da lahko učinkovito deluje.

Slika 11: Dimenzije virtualnih timov



Vir: I. Zigurs, Leadership In Virtual Teams, 2002, str. 342.

Če tesno povezani tradicionalni tim razširimo z dodajanjem zunanjih pogodbenih sodelavcev iz svetovalne organizacije, že dobimo začetke virtualnega tima. Če pa skupino razširimo še na čezmorske pogodbene sodelavce, geografski in organizacijski razpršenosti dodamo še kulturno in časovno, se virtualnost še poveča. Vsaka oblika tima je drugačna glede na mešanico značilnosti, primerne načine komunikacijskih procesov in najboljše možne izbire tehnologije in vloge članov.

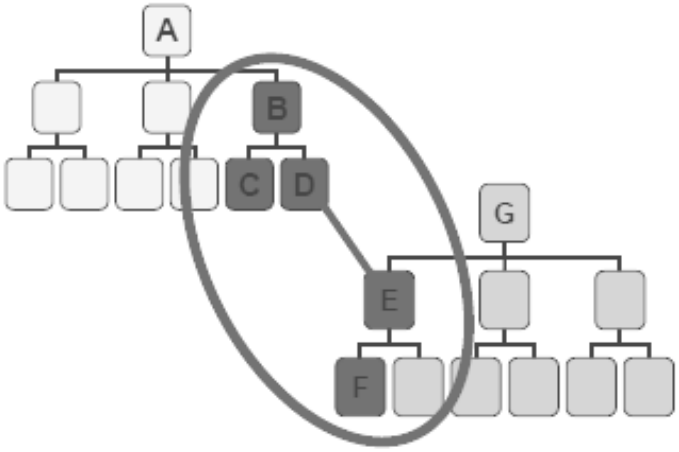
Kot pri tradicionalnih timih je v virtualnih timih zelo pomemben dober vodja, ki zna tim s pozitivnim vedenjem premakniti bližje zastavljenemu cilju. Vodje virtualnih timov igrajo pomembno vlogo pri ustvarjanju timov in določanju pravil za člane timov z namenom, da bi proces timskega dela stekel uspešno in predvsem učinkovito (Zigurs, 2003). Zaradi geografske in časovne razdalje so izzivi vodenja pri virtualnih timih povečani, vendar pa vloga vodje v virtualnemu timu ni povsem jasno definirana, saj virtualne time pogosto zaznamuje visok nivo samostojnosti, ne pa direktnega nadzora (Reilly & Ryan, 2007). Vloge pri vodenju se po večini porazdelijo po članih virtualnega tima in so le redko povsem odvisne od samo enega člana. Timi imajo lahko vodjo ali pa tudi ne, vseeno pa se mora pojaviti vedenje vodje, če hoče tim napredovati. Različni ljudje lahko privzamejo tako vodenje ob različnih časih. Glede na razpršenost virtualnih timov bomo najbrž pogosteje naleteli na vodilne vloge, ki se selijo od člana do člana v različnih časovnih obdobjih v življenju njihovega skupnega dela.

2.2 Virtualna organizacija

Virtualna organizacija je geografsko porazdeljena mehko povezana mreža neodvisnih institucij, podjetij ali specializiranih posameznikov (od 2 do 25 posameznikov, če jih je več, se utegnejo razdeliti na podskupine), ki se ob intenzivni uporabi informacijske in komunikacijske tehnologije spontano združujejo na določenem poslu, da bi na trgu dosegli neko primerjalno prednost pred svojimi konkurenti. Virtualne organizacije so vedno človeške združbe in jih ne moremo nadomestiti z nečim, česar ni, kar naj bi pomenila virtualnost (Drucker, 1999). To pomeni, da moramo natančno vedeti, kaj je v organizaciji lahko virtualno in kaj ne more biti. Podjetje tu ni sinonim za podjetje kot samostojno gospodarsko družbo ali celotno organizacijo. Pogosto se nanaša na organizacijsko enoto ali samo na določene funkcije v okviru posameznega podjetja, ki so organizirane virtualno, ostali deli podjetja pa lahko poslujejo na tradicionalen način.

Virtualna organizacija je predvsem začasna tvorba, ki je ustanovljena zato, da lahko partnerji dosežejo točno določene poslovne cilje in deluje samo toliko časa, dokler taki cilji niso doseženi (Slika 12). Oddelki iz različnih podjetji se združijo in skupaj tvorijo novo virtualno podjetje z namenom hitrejše in boljše opraviti določeni cilj (Bavec, 2002).

Slika 12: Virtualna organizacija



Vir: C. Bavec, An assessment of the organization virtuality with three different reference models, 2002, str. 26.

2.2.1 Na poti do virtualne organizacije

Pritisk globalizacije sili podjetje k racionalizaciji svojega poslovanja in vzpostavitve takega organizacijskega modela, da podjetje deluje konkurenčno. Vedno več podjetij vidi rešitev v uspešnosti poslovanja v virtualnih organizacijah. Temelji za ustanovitev virtualne organizacije so različni in se med seboj spreminjajo, vendar je ključni razlog v tvorjenju virtualne organizacije v namenu zasledovanja in uresničitve poslovne priložnosti na globalnem trgu. Speier (1998) je s primeri dokazal, da virtualna organizacija ustvari nove produkte veliko hitreje, zmanjša navidezno velikost organizacije in zmanjša tveganje iskanja novih poslovnih priložnosti. Virtualna organizacija omogoča skupno zbiranje različnih delov, ki jih potrebuje, hkrati pa omogoča razpustitev celotnega sistema, ko ta ni več potreben. Te organizacije odsevajo veliko fleksibilnost na globalnih trgih in ustvarjajo in razpuščajo subjekte, kakor določa trg.

S poslovnega zornega kota sta osnovna cilja virtualne organiziranosti optimalna izraba finančnih, tehnoloških in drugih virov, s katerimi razpolaga podjetje, ter največja možna odzivnost na zunanje okoliščine. Virtualne organizacije praviloma izrabljajo več virov, kot jih realno imajo, s tem da se navidezno združujejo z drugimi organizacijami in izrabljajo še njihove vire. Oba pogoja, optimizacija virov in prilagodljivost, sta nujna za poslovanje v globalni ekonomiji, zato delujejo najuspešnejše virtualne organizacije globalno in so najpogostejše v okoljih, ki slonijo na globalni ekonomiji, kot so ZDA in razviti del Evrope (Papows, 1999).

2.2.2 Značilnosti virtualne organizacije

Pogoj, da se podjetje odloči za virtualno poslovanje, je predvsem v učinkovitosti virtualne organizacije. Zaradi tega ima virtualna organizacija kar precej specifičnih značilnosti, ki se razlikujejo od tradicionalnih (Tabela 1).

Tabela 1: Značilnosti virtualne organizacije

Značilnosti	Virtualna organizacija
Geografska razpršenost	Virtualna organizacija posluje globalno in ima svoje dejavnosti na različnih krajih po vsem svetu.
Začasna organizacija	Večina strokovnjakov se strinja, da je virtualna organizacija po naravi začasna organizacijska struktura.
Mreža neodvisnih organizacij	Virtualno organizacijo sestavljajo neodvisne organizacije, povezane v mrežo.
Osredotočenje na porabnika in prilagajanje porabniku	Ta značilnost se nanaša na sposobnost prilagajanja produkta ali storitve določenemu porabniku.
Zanašanje na inovacije	Virtualna organizacija se mora odzivati na poslovne priložnosti na trgu, zato mora razvijati inovacijske rešitve, da ustvari konkurenčne prednosti.
Osredotočenje na timsko delo	Virtualni timi uspejo veliko hitreje in učinkoviteje odkriti in realizirati poslovne priložnosti na trgu.
Delno prekrivanje nalog	Organizacije znotraj povezave morajo izvajati še druge naloge.
Osredotočenje na sposobnosti virtualne organizacije	Vsaka organizacija v povezavi je kvalitetna na določenem področju, zato mora prinesti svojo kvaliteto k skupni povezavi.

Vir: R. T. K. Larsen & R. C. Mcinerney, Preparing To Work in The Virtual Organization, 2002, str. 449.

3 PRENOVA METODE RAZVOJA PROGRAMSKIH REŠITEV V PODJETJU HERMES SOFTLAB d.o.o.

Podjetje Hermes SoftLab d.o.o. je bilo ustanovljeno leta 1990. Podjetje je eden največjih evropskih izdelovalcev programskih rešitev in ponudnikov strokovnih storitev z obsežnimi izkušnjami na področju upravljanja sistemov za shranjevanje podatkov (angl. *»Storage Data Management«*) ter upravljanje omrežij in sistemov (angl. *»Network & Systems Management«*). Hermes SoftLab d.o.o. je do danes opravil že preko 6000 »inženirskih let« dela in ob tem razvil število dolgoročnih partnerstev z vodilnimi svetovnimi podjetji, ki

podjetju Hermes SoftLab d.o.o. priznavajo kakovost, inovativnost, visoko usposobljenost zaposlenih ter veliko predanost potrebam kupcev. Potrdilo ISO9001/TickIT in delovanje po principu celovitega obvladovanja kakovosti (angl. *»Total Quality Management«*) dokazujeta predanost kvaliteti za doseganje večje učinkovitosti, kakovosti produktov in zadovoljstva strank. Storitve podjetja so namenjene predvsem podjetjem in državnim ustanovam, ki potrebujejo razvojnega partnerja na evropskih trgih. Leta 2008 je podjetje Hermes SoftLab d.o.o. postalo član skupine ComTrade Group d.o.o., ki skupno zaposluje 1.600 ljudi in deluje v 14 državah s skupnim prihodkom 300 milijonov evrov.

Podjetje Hermes SoftLab d.o.o. je razdeljeno na več podskupin, ki sodelujejo na več projektih. V grobem je podjetje razdeljeno na tri delovne skupine:

- razvoj programskih rešitev – procesi razvoja in usposobljenost zaposlenih v podjetju Hermes SoftLab d.o.o. dosegajo raven, ki jo zagotavljajo mednarodne IT korporacije. Svojim strankam nudijo dostop do ključnih tehnoloških virov, načrtovalnih in razvojnih skupin strokovnjakov ter do preizkušenih procesov razvoja programskih rešitev. S tem jim omogočajo, da se osredotočijo na svojo osnovno dejavnost ter izkoriščajo prednosti tehnologij z namenom povečevanja svoje konkurenčne prednosti in operativne fleksibilnosti.
- strokovne rešitve – strankam svetujejo pri načrtovanju, implementaciji in integraciji programskih rešitev. Pomagajo jim ovrednotiti in izbrati najprimernejše tehnologije ter zagotoviti nemoteno integracijo programskih rešitev v obstoječi informacijski sistem. Le s pravilno integracijo lahko stranka doseže maksimalno donosnost investicije.
- produkti – rešitve SMART Plug-in in Management Packs zagotavljajo učinkovito povezavo med programi, ki tečejo na strežniku, in drugimi strežniškimi komponentami. Najbolj iskana izdelka na svetovnih trgih sta trenutno SPI za Siebel in SPI za Citrix MetaFrame. Modula omogočata nadzor in upravljanje aplikacij Siebel oziroma Citrix MetraFrame strežnikov v HP OpenView okolju.

Podjetje nastopa kot pomemben partner družbam: Hewlett-Packard d.d., Agilent Technologies d.d., Ericsson d.d., Mobitel d.d., Sixt AG d.o.o., SKB Banka d.d., Raiffeisen Banka d.d., Hypo Alpe-Adria-Banka d.d., Nova Ljubljanska Banka d.d., Atronic d.o.o., Peiker d.o.o., Vodafone d.d..

3.1 Razvoj programskih rešitev z zaporednim modelom v podjetju Hermes SoftLab d.o.o.

Podjetje Hermes SoftLab d.o.o. je za razvoj programskih rešitev uporabljalo zaporedni model. Trajanje projekta je definirano s strani stranke s sodelovanjem projektnega vodje. Projekt v večini primerov traja nekje od enega leta do največ dveh. Glede na to, da so

razvijali programsko rešitev, kateri se dodaja nova funkcionalnost z vsakim projektom, so si bili projekti zelo podobni, kar pomeni, da obstaja nek ustaljen in preverjen sistem poročanja in spremljanja projekta (Molan, 2008).

Za zagotovitev uspešnega projekta je potrebno kvalitetno projektno planiranje. Prvotno je potrebno razumeti postavljene zahteve stranke, jih skupaj s stranko analizirati in ovrednotiti ter postaviti prioritete projekta. Eden pomembnejših delov projektnega planiranja je življenjski cikel projekta, v katerem se definirajo mejniki projekta. Dobra komunikacija z uporabnikom programske rešitve je pomembna za uspešno dokončan projekt. Zaporedni model razvoja programskih rešitev je skupek aktivnosti, ki omogočajo prehod od naročnikovih zahtev do delujoče programske rešitve. Primarno želi zaporedni model razvoja programskih rešitev odgovoriti na vprašanji, kaj naj naredimo v nadaljevanju projekta in kako dolgo naj to počnemo.

Zaporedni model razvoja programskih rešitev se dobro obnese pri projektih, kjer so programske zahteve zelo stabilne in dobro znane že na začetku projekta. Koncept zmanjšuje količino odvečnega planiranja, saj je kompleten projektni plan narejen že ob začetku življenjskega cikla. Ker so faze med seboj ostro ločene, lahko v različnih fazah projekta sodelujejo različni posamezniki. Pogoj za prehod ene faze v drugo je dobro dokumentirana in zaključena prejšnja faza.

3.1.1 Faze zaporednega modela razvoja programskih rešitev v podjetju Hermes SoftLab d.o.o.

Podjetje Hermes SoftLab d.o.o. uporablja zaporedni model vodenja projektov, ki se deli na naslednje faze (Tabela 2):

- zagon projekta,
- faza analize projekta,
- faza načrtovanja programskega produkta,
- faza specifikacije,
- faza implementacije in
- faza prevzema.

Tabela 2: Faze zaporednega modela v podjetju Hermes SoftLab d.o.o.

Faza	Ime	Cilji	Izvedba faze
-1	Zagon projekta	Ocenitev poslovnih priložnosti.	Preučitev predlogov, analiza trga.
0	Analiza	Naročniku se predloži opis	Analiza specifikacij zahtev

		programske rešitve.	programske rešitve.
1	Načrtovanje	Potrebno je definirati programsko rešitev in projekt.	Izdelava projektnega plana, plana kvalitete in plana testiranja programske rešitve.
2	Specifikacija	Specifikacija programske rešitve in projekta.	Specifikacija zunanjih zahtev.
3	Implementacija	Implementacija programske rešitve.	Podrobni dizajn in testna specifikacija programske rešitve.
4	Prevzemanje	Uspešno prevzetje programske rešitve s strani uporabnika.	Finalno testno poročilo programske rešitve.

Vir: Hermes SoftLab d.o.o., interno gradivo, 2006.

3.1.1.1 Zagon projekta

Zagon projekta ni ena izmed faz zaporednega modela razvoja programske rešitve, ampak je raziskava in odločitev podjetja, ali se odloči za izvedbo projekta. Odločitev o zagonu projekta je poslovna priložnost, ki je ekonomsko upravičena in podjetju nudi tržno prednost. Podjetje preveri izkušnje na področju, za katerega zavzema projekt, če ima zadosti virov in drugi resursov za izvedbo projekta.

3.1.1.2 Faza analize projekta

Ta faza vključuje analiziranje programskega problema s kompletnimi specifikacijami naročnikovih zahtev in potreb obnašanja programske rešitve. Nato se razjasni zadnje potencialne nejasnosti specifikacij funkcionalnosti programske rešitve. Po začetni izdelavi posnetka stanja in analize zahtev se naročnik in izvajalec uskladita v detajlnih funkcionalnih in tehnoloških zahtevah programske rešitve, na podlagi katerega se izdelajo prvi visokonivojski dokumenti koncepta celotne programske rešitve v obliki arhitekturne zasnove programske rešitve.

Faza analize projekta vključuje tudi funkcionalen opis, funkcionalne zahteve in specifikacije programske rešitve. Kot rezultat dobimo specifikacije zahtev programske rešitve ali dokument FURPS+. Specifikacije naj bi opisovale funkcionalne zahteve, značilnosti strojnega okolja, obliko uporabniških vmesnikov in odzivni čas ali zahtevane zmogljivosti programske rešitve. Če uporabnik ne more jasno izraziti svojih zahtev programske rešitve, je dolžnost projektne skupine, da uporabniku pomaga pri njihovem oblikovanju. Prav tako poskrbimo za vse aktivnosti, potrebne za nemoten potek implementacijske faze.

Dokument FURPS+ vsebuje naslednje specifikacije analize projekta:

- functionality – navedba zahtevanih funkcij programske rešitve ter razdelitev le teh po prioriteti;
- usability – uporabnost programske rešitve;
- reliability –opisana zahtevana zanesljivost ob predaji programske rešitve;
- performance – opisane zahtevane izboljšave, vsebujoč hitrost, razpoložljivost in odzivni čas programske rešitve;
- supportability – opisane zahteve po vzdrževanju programske rešitve;
- portability – opisane zahteve po prenosljivosti programske rešitve na druge sisteme in/ali platforme;
- internationalisation & localisation – opisane potrebe po lokalizaciji programske rešitve;
- opredelitev in opis dodatnih ciljev, ki so odvisni od obravnavane programske rešitve.

V tej fazi je uporabniško sodelovanje ključnega pomena, saj prispeva k jasni predstavi končne programske rešitve. Potem ko so zahteve stranke jasno določene, dokumentirane in potrjene s strani stranke, definirajo aktivnosti, ki so potrebne za izvedbo projekta, in izberejo ustrezni razvojni proces.

3.1.1.3 Faza načrtovanja programske rešitve

V tej fazi se uporabniku predloži opis programske rešitve, kot jo vidi podjetje. Dokument HLA (angl. *»High Level Architecture«*) opisuje visokonivojsko arhitekturo programske rešitve (tehnični dokument, ki je bolj uporaben za programerja kot naročnika). Začnejo se pogajanja in usklajevanja predlagane programske rešitve, česar posledica je sprejetje predlagane oz. modificirane programske rešitve ali zavrnitev programske rešitve. V tej fazi se posodobi tudi dokument FURPS+.

Dokument HLA vsebuje:

- podrobnejši pregled zahtev FURPS+,
- logično razčlenitev dokumenta FURPS+, ki se nanaša na funkcionalnosti programske rešitve,
- fizično razčlenitev logičnih nivojev programske rešitve,
- določitev vpliva določene platforme na prenosljivost, lokalizacijo, varnost in hitrost.

Potrebno vedeti, da je ta faza zelo pomembna. Če naročnik v tej fazi spremeni ali drugače formulira svoje zahteve, ne povzroči dosti dodatnega dela, če pa to naredi v eni od prihodnjih faz, pa se stroški projekta zelo povečajo.

3.1.1.4 Faza specifikacije

V tej fazi se naročniku predloži popolna dokumentacija projekta ali dokument ERS (angl. *»External Reference Specification«*). Dokument ERS opisuje, kako se bo določena programska rešitev namestila (kaj rabi za namestitev in normalno delovanje, kako bo zagotovljena kakovost programske rešitve, bo podpiral internacionalizacijo…). Ta dokument je na nivoju povprečnega naročnika, ki ga prebere in vidi, če mu programska rešitev ustreza za njegovo specifično uporabo. Nato sledijo pogajanja in usklajevanja dokumentacije. Ko so pogajanja končana, ima podjetje dovolj informacij za podpis pogodbe z naročnikom in opredelitev potrebnega časa za izvedbo projekta.

Dokument ERS vsebuje:

- podroben opis programske rešitve; zamisel celega projekta,
- operacije med objekti, ki so vidni uporabniku,
- uporabniški vmesnik,
- podporo drugih programskih rešitev in/ali podporo strojni opremi.

3.1.1.5 Faza implementacije

To je najdaljša faza celotnega razvoja programske rešitve. Tu se programska rešitev razvija, testira, dokumentira. Če v tej fazi pride do večjih sprememb projekta (zahtev), se je potrebno vrniti v eno od prejšnjih faz. Začne se izdelava posameznih modulov programske rešitve. Moduli se najprej samostojno testirajo, nato testirajo komunikacijo z drugimi moduli, nato se testira še programsko rešitev kot celoto. Preveri se pravilnost izračunavanja. Opravi se analiza kvalitete rezultatov, na koncu pa se programsko rešitev preizkusi pri uporabniku samem.

Implementacijo vsakega sklopa pričnejo z natančno analizo potencialne tehnične rešitve, ki ji sledi predlagana implementacija – to velja za vse komponente in module programske rešitve (uporabniški del, centralne aplikacije, administrativni moduli, instalacija in podatkovne baze). Po sklopu implementacije sledi sklop z natančnim testiranjem po vnaprej pripravljenih načrtih. V tem sklopu morajo biti vse osnovne funkcionalnosti že izvedene. Najpomembnejše opravilo tega sklopa je preizkušanje programske rešitve, tako s stališča verifikacije funkcionalnosti kot s stališča pravilnosti delovanja. Vzporedno s preizkušanjem, ki poteka vse do zadnjega dne prevzema programske rešitve, se sistematsko odpravljajo odkrite napake. Najprej se odpravljajo najresnejše napake, ki najbolj zavirajo ostale funkcionalnosti programske rešitve.

3.1.1.6 Faza prevzema

To je zadnja faza projekta. Cilj te faze je predstaviti in dokazati, da programska rešitev ustreza podpisani pogodbi (torej določenim zahtevam) z namenom, da nam uporabnik potrdi sprejetje programske rešitve. Izdelana programska rešitev se predstavi uporabniku, pokaže se mu, kako se dela s programsko rešitvijo ter se pregleda, če izpolnjuje vse pogodbene obveznosti. Na podlagi rezultatov uporabnik v tej fazi sprejme programsko rešitev.

3.1.2 Sistem poročanja in spremljanja projekta

Na začetku projekta se definirajo mejniki razvoja programske rešitve. Ti mejniki se zgodijo ob koncu vsake faze, in sicer jih vodi tako uporabnik kot izvajalec. Ob vsakem takem mejniku je spisan formalno določen dokument razvoja programske rešitve, ki vsebuje:

- projektne cilje,
- status končane faze,
- pregled plana oz. razpored,
- zaključne kriterije,
- nedokončane naloge,
- statistiko,
- pregled dokončanih nalog in
- možnosti za izboljšanje.

Poleg mejnikov se uporabniku poroča razvoj programske rešitve še s pomočjo četrtletnih sestankov in tedenskih telefonskih konferenc, v katerih se poroča trenutni status faze. Ob koncu vsakega projekta se s stranko naredi sestanek, kjer se razčleni in analizira projekt ter se poda slabosti in prednosti končane programske rešitve in definira naloge za naprej.

Poleg poročanja uporabniku pa podjetje izvaja tudi interno poročanje in spremljanje projekta. Enkrat mesečno projektni vodja poroča v obliki nestandardiziranega dokumenta vodji programa in stranki o stanju razvoja programske rešitve. Posamezni vodilni inženirji imajo redne sestanke s svojimi timi, večino spremljanja in poročanja pa je opravljenega preko elektronske pošte.

3.1.3 Testiranje in zagotavljanje kvalitete

Testiranje programskih rešitev je namenjeno preventivi in odkrivanju napak. Preventiva pomeni spremljanje in izboljševanje procesa razvoja programskih rešitev. Odkrivanje napak pa pomeni, da se programske rešitve razvijajo v kontroliranem okolju in da so rezultati razvoja programskih rešitev ovrednoteni.

Pri procesu testiranja programskih rešitev nastopajo posamezniki z različnimi vlogami. Vodja testnega tima koordinira testne aktivnosti, obvešča vodjo projekta o trenutnem stanju testiranja in vodi testni tim. Zagotovi potrebno strojno opremo in jo pripravi za testiranje. Pomembno vlogo pri testiranju programske rešitve predstavlja vsebinski vodja testiranja, ki določa, kateri podatki se testirajo in v kakšnih okoliščinah. Ne nazadnje nastopajo pri testiranju še sami preizkuševalci, ki dejansko opravijo testiranje programske rešitve.

Proces testiranja se začne z določanjem strategije testiranja programske rešitve (Tabela 3). S strategijo je potrebno opisati zahtevano programsko in strojno opremo za izvedo testiranja, testno metodologijo, vloge in odgovornosti članov testne skupine, funkcionalne in tehnične zahteve programske rešitve. V tej fazi je določen tudi časovni plan testiranja, ki opredeljuje roke za izvedbo testnih aktivnosti in posameznike, ki bodo testirali programsko rešitev. Faza se konča s pregledom in odobritvijo strategije testiranja, ki predstavlja vhodni dokument za fazo načrtovanja testiranja.

Tabela 3: Izvajanje testiranja programske rešitve na različnih nivojih

Način testiranja	Opis testiranja
Testiranje modula	Testiranje modula je prvi nivo testiranja in je večinoma v domeni razvijalcev, ne pa testne skupine.
Testiranje funkcionalnosti	Testiranje funkcionalnosti je testiranje, kjer se preverjajo funkcionalnosti posameznih modulov. Test mora opraviti testna skupina, ne pa razvijalci programske rešitve.
Testiranje uporabnosti	Poudarek je na testiranju prijaznosti do končnega uporabnika. Seveda je to zelo subjektivna tema, ki je odvisna od končnega uporabnika oziroma naročnika.
Integracijsko testiranje	Integracijsko testiranje programske rešitve se lahko začne, ko so končana testiranja vseh posameznih modulov. Pri integracijskem testiranju gre za preverjanje, kako se posamezni moduli integrirajo v celoto. Opravi ga testna skupina. Integracijsko testiranje se konča s primerjavo pričakovanih in dejanskih rezultatov.

Sistemsko testiranje	Sistemsko testiranje sledi končanemu integracijskemu testu. Med sistemskim testiranjem je celotna programska rešitev nastavljena tako, da simulira dejansko produkcijsko okolje. Testirajo se vse funkcionalnosti programske rešitve, ki bodo zahtevane v dejanskem uporabniškem okolju. Testiranje vključuje povezovanje rešitev do relacijskih baz, uporabo mrežnih komunikacij ter povezovanje z ostalimi viri podatkov in strojno opremo.
Obremenitveno testiranje	Obremenitveno testiranje preizkuša programsko rešitev v primeru velike obremenitve. Testiranje meri, kako velika obremenitev vpliva na odzivne čase končnih uporabnikov in ugotavlja, ali lahko prevelika obremenitev pomeni tudi prenehanje delovanja programske rešitve.
Namestitveno testiranje	Potrebno je preveriti celotno namestitev programske rešitve in namestitev posameznih modulov oziroma nadgradnjo programske rešitve. Podatki, zbrani med testiranjem, so pomembni za izdelavo navodil za namestitev programske rešitve.
Varnostno testiranje	Varnostno testiranje meri odziv programske rešitve na različne neavtorizirane poskuse pristopa. Zaradi vedno novih metod vdora v informacijske sisteme je test zelo težaven in se mora izvajati periodično.
Testiranje sprejemljivosti	Testiranje sprejemljivosti je podoben sistemskemu testiranju s to razliko, da se izvaja s strani končnega uporabnika programske rešitve. S tem testiranjem uporabnik še enkrat preveri funkcionalnosti programske rešitve, preden gre ta v produkcijo.
Alfa testiranje	Alfa testiranje je testiranje programske rešitve, ko je razvoj praktično zaključen. Testiranje izvajajo končni uporabniki, glede na njihove ugotovitve pa so možne še manjše spremembe programske rešitve.
Beta testiranje	Beta testiranje je testiranje programske rešitve, ko je razvoj zaključen. Namen testiranja je odkriti še zadnje napake pred končno verzijo rešitve, preden gre programska rešitev v produkcijsko uporabo. Tako kot alfa testiranje tudi beta testiranje izvajajo končni uporabniki.

Vir: Hermes SoftLab d.o.o., interno gradivo, 2006.

V fazi načrtovanja testiranja je poleg potrjene testne strategije potrebno tudi razumevanje kompleksnosti programske rešitve in povezanosti posameznih programskih modulov.

Bistvo faze je izgradnja testnih scenarijev. Scenarij je zaporedje korakov, ki sledi funkcionalnosti rešitve in določa testne pogoje, podatke, uporabljene za test, ter določa želene rezultate. Scenariji so načrtovani tako, da zaobjamejo tipične in netipične dogodke, ki se lahko dogodijo v rešitvi. Za lažje sledenje v času izvedbe testiranja programske rešitve se scenariji zapišejo v obliki testnih skript. Testna skripta je opis zaporednih korakov, potrebnih za izvedbo enega ali več scenarijev. Tako zagotovijo, da testiranje ostane v okviru predpisanih zahtev za programsko rešitev. Potrjeni scenariji, določeni testni pogoji, testni podatki in strategija testa so osnova za izvedbo testiranja.

Zadnja faza v procesu testiranja je izvedba testa. Testne aktivnosti se izvajajo po terminskem planu testiranja. Med izvajanjem testiranja programske rešitve se testna skupina sestaja na rednih sestankih, kjer pregleduje in obravnava potek testiranja, status posameznih aktivnosti in usklajuje nadaljnje aktivnosti. Testiranje modulov mora natančno slediti testnim scenarijem. Vsak preizkušeni postopek se zabeležiti v dnevnik testiranja, skupaj z morebitnimi odkritimi odstopanji od pričakovanih rezultatov.

Rezultati testiranja programske rešitve so ovrednoteni s strani članov projektnega tima, ki ugotovijo, ali so rezultati testiranja v sladu s pričakovanji. Vsa ugotovljena odstopanja in anomalije so dokumentirani za nadaljnjo obravnavo in reševanje. Glede na vrsto ugotovljene napake je možno napako odpraviti že med izvajanjem testiranja ali pa jo zabeležiti kot nerešen problem, ki je sporočen razvijalcem programske rešitve. Ko je testiranje tudi formalno zaključeno, je pomembno poudariti, da bo preizkušena programska rešitev prestavljena v produkcijsko okolje šele po uspešno opravljenem testiranju končnih uporabnikov oziroma naročnika.

3.1.4 Prednosti in slabosti zaporednega modela razvoja programskih rešitev v podjetju Hermes SoftLab d.o.o.

Zaporedni model razvoja programskih rešitev je primeren za relativno kompleksne projekte, če zahteve dobro razumemo in se med projektom ne bodo bistveno spreminjale. Uporaben je predvsem, kadar imajo kakovostno zajete zahteve prednost pred stroški in časom, porabljenim za zajem zahtev. Taki projekti so bili v podjetju v preteklosti stalna praksa. Zaporedni razvoj pomaga zmanjševati količino režijskega dela, ki ni v neposredni povezavi z izdelavo programskih rešitev (npr. vodenje projekta), saj je mogoče načrtovanje v celoti izvesti vnaprej. Po drugi strani se ob striktnem upoštevanju pravil čistega zaporednega razvoja količina takega dela lahko tudi poveča. Čas, ki je porabljen v začetnih fazah razvoja programskih rešitev, lahko v kasnejših fazah ustvari velike prihranke. Dokazano je, da je napaka veliko cenejša, če je bila odkrita v začetnih fazah opredelitve zahtev in načrtovanja programskih rešitev (Dorfman & Thayer, 2000).

Z vedno večjim napredkom v tehnologiji in konkurenci so se projekti v podjetju postopoma časovno zmanjševali in v razvoju stalno spreminjali. Glavna kritika zaporednega razvoja je, da ni fleksibilen (Boehm, 1988). Vsaka naknadna sprememba namreč zahteva veliko dodatnega napora. Posledica je, da ni primeren za projekte, pri katerih pričakujemo pogosto spreminjanje in dopolnjevanje zahtev. Čeprav zaporedni razvoj omogoča tudi vračanje, to vračanje ni namenjeno iterativnemu izvajanju postopkov, ampak odpravi morebitnih napak oziroma pomanjkljivosti programskih rešitev. Osnovna različica zaporednega razvoja ne omogoča paralelnega izvajanja delov postopkov, saj zahteva, da je nek postopek v celoti zaključen, preden lahko nadaljujemo z naslednjim.

3.2 Razlogi za prenovo razvoja programskih rešitev v podjetju Hermes SoftLab d.o.o.

Kljub omenjenim lastnostim in dejstvu, da zaporedni model razvoja programskih rešitev uvaja disciplinirano izvajanje aktivnosti posameznih faz, ki so dobro dokumentirane, pregledane in testirane, se zaradi nekaterih slabosti danes v podjetju v čisti obliki ne uporablja več (Avison & Fitzgerald, 2002). Njegova glavna pomanjkljivost je, da ne odraža resničnega razvojnega procesa. Skoraj nikoli namreč ne velja, da najprej popolnoma zaključijo eno fazo, nato drugo, tretjo itd., ampak po potrebi med fazami prehajajo (npr. iz načrtovanja in programiranja se večkrat vrnejo v analizo ipd.). Po zaporednem pristopu pa prehod v že izvedene faze ni mogoč. Zaradi pogostih sprememb, ki so značilne za današnja poslovna okolja, to predstavlja še posebej veliko oviro, saj se morajo pogosto vračati v fazo analize.

Kljub omenjenim slabostim pa velja, da se je zaporedni model v preteklosti izkazal kot učinkovit pristop h gradnji velikih in kritičnih sistemov, še posebej tistih, kjer so bile zahteve poznane ali vsaj dobro definirane. Vendar je bila prenova ključnega pomena za podjetje, saj so bili razvojni cikli predolgi. To je pripeljalo do velikega števila popravkov, predelav, kar pa je občutno dražilo projekt ter slabšalo timsko moralo. Podjetje je potrebovalo bolj modularni pristop k razvoju programskih rešitev, ki omogoča razvoj po funkcionalnosti. To omogoča več odziva na trgu ter večjo razpoložljivost programskih rešitev. Kritika metodologije življenjskega cikla izpostavlja nekatere potencialne slabosti razvoja programskih rešitev (Parnas & Clements, 1986), ki so predstavljene v nadaljevanju (Avison & Fitzgerald, 2002).

3.2.1 Spreminjanje zahtev uporabnika

Uporabnik svoje zahteve programske rešitve pogosto spreminja:

- spreminjajo se poslovni cilji in prioritete uporabnika, programska rešitev v razvoju pa je tehnološko tesno povezana s cilji in prioritetami le-tega.
- uporabnik nima prave predstave, do katere mere mu bo programska rešitev pomagala pri njegovih aktivnostih in pogosto spreminja zahteve programske rešitve, ki spreminjajo že dogovorjene funkcionalnosti.
- uporabnik je obremenjen z dotedanjim procesom dela in zahteva funkcionalnosti, za katere se naknadno izkaže, da so nepotrebne in povsem nepraktične.
- prvotne zahteve uporabnika so preveč ohlapne in nedorečene. Za implementirane funkcionalnosti uporabnik ugotovi, da ne ustrezajo povsem in da so potrebni popravki programske rešitve.
- pojavljajo se tehnološki nesporazumi med uporabnikom in analitikom, ker uporabnik ni podal nekaterih temeljnih zahtev, za katere meni, da so same po sebi razumljive, analitik pa ni dovolj natančno prikazal rezultatov analize programske rešitve uporabniku.

V kasnejši fazi lahko pride do spreminjanja zahtev programske rešitve, kar privede do ponovne analize, kodiranja in testiranja, kar posledično pomeni višje stroške implementacije. Lahko pride tudi do neskladja funkcionalnosti, zmanjšane odpornosti na napake in celo do nedokončane implementacije dane funkcionalnosti. Praviloma pa uporabnik ne odstopa od prvotno zastavljenih rokov in to dodatno poveča možnost, da programska rešitev ne bo dokončana.

3.2.2 Spreminjanje razvojnega tima

Razvojni tim se lahko zaradi različnih dejavnikov spremeni ali zamenja (bolniške odsotnosti, prerazporeditev na druge projekte, reševanje nepričakovanega tehničnega problema na sorodnem področju in podobno). Spremenjeni razvojni tim lahko potrebuje več časa za razvoj programske rešitve, poslabša se lahko tudi kvaliteta izdelka in implementirane funkcionalnosti. Vendar pa je v primeru, da imamo na razpolago dodatni kader, pričakovati, da bo razvoj programske rešitve končan prej, izdelan kvalitetnejše ali pa bo vseboval več funkcionalnosti, kot je bilo prvotno načrtovano.

3.2.3 Vpliv tehnoloških in arhitekturnih rizikov

Med razvojem programske rešitve lahko naletimo na napake in pomanjkljivosti uporabljenih orodij in arhitekture, ki jih moramo potem zaobiti ali kako drugače rešiti. Še

bolj problematično je, če arhitekturne ali težje funkcionalne napake ugotovimo med končno integracijo modulov ali celo v produkciji programske rešitve kljub sprotnemu intenzivnemu testiranju.

3.2.4 Krajšanje cikla razvoja in uporabe programske rešitve

Celotni cikel razvoja programske rešitve, od načrtovanja, začetka uporabe in prehoda v fazo, v kateri se programska rešitev ne podpira in vzdržuje, je čedalje krajši. Pri tem uporabnik pričakuje vedno kvalitetnejšo programsko rešitev, ki je prijazna do njega in z bogatim naborom funkcionalnosti, medtem ko je cena programske rešitve nizka. Zato je potrebno določene faze razvoja programske rešitve izvajati vzporedno, če lahko pri razvoju sodeluje večje število razvijalcev, da je razvojni cikel čim krajši. V kolikor je programska rešitev namenjen končni prodaji, mora biti čim prej na tržišču, da si pridobi ustrezno visok tržni delež. Četudi je programska rešitev namenjena interni uporabi v podjetju, je smiselno, da se jo začne uporabljati in vpeljevati čim prej, saj se s tem povečajo poslovne prednosti podjetja.

3.2.5 Vpliv konkurenčne programske rešitve

V kolikor je programska rešitev naprodaj bodisi na prostem tržišču bodisi na majhnem tržišču specializiranih programskih rešitev, na njegovo prodajo in tržni delež vplivajo tudi konkurenčne programske rešitve. To pomeni, da je po potrebi potrebno programsko rešitev funkcionalno dopolniti ali spremeniti, da lahko konkurira na trgu.

3.2.6 Spreminjanje datuma produkcijske izdaje programske rešitve

Datum končne izdaje produkcijske verzije programske rešitve se lahko spreminja iz različnih razlogov:

- izdaja programske rešitve je povezana s predstavitvenimi aktivnostmi na konferencah, sejmih in predstavitvah.
- uporaba programske rešitve je vezana na druge aktivnosti v podjetju, šolanje, ureditev podatkov, vpeljavo kakšnega drugega programskega izdelka.
- programska rešitev mora biti predčasno na tržišču, da si pridobi ustrezno velik tržni delež, ki bi ga ob prepozni predstavitvi ne imel.

Če se rok za produkcijsko izdajo programske rešitve podaljša in imamo na razpolago vire, je smiselno programsko rešitev še dodatno dopolniti. V primeru, da se rok skrajša, se seveda kateri od parametrov kvalitete programske rešitve poslabšajo. Povečajo se stroški

razvoja programske rešitve, vendar pa se pričakuje, da je ob novo postavljenem roku programska rešitev vseeno pripravljena za produkcijsko izdajo.

3.3 Uvedba modela FDD v podjetju Hermes SoftLab d.o.o.

Podjetje Hermes SoftLab d.o.o. že nekaj časa ne uporablja čistega zaporednega modela pri razvoju programskih rešitev. Zaradi številnih kritik so se v podjetju začele pojavljati modificirane različice zaporednega razvoja programskih rešitev, ki so tako uvajale bolj enostavno prehajanje in vračanje med postopki, paralelno izvajanje delov različnih postopkov itd. Vendar pa se podjetje samo z modifikacijo zaporednega modela ni uspešno kosalo s konkurenco in novo tehnologijo.

Ker agilne metodologije razvoja programskih rešitev prinašajo dodano vrednost zaradi svoje prožnosti, fleksibilnosti, konstruktivnega sodelovanja in popolne usmerjenosti v poslovno vrednost, se je podjetje Hermes SoftLab d.o.o. odločilo, da spremeni način razvoja programskih rešitev in nadgradi obstoječi zaporedni model razvoja programskih rešitev v funkcionalno voden razvoj programskih rešitev ali model FDD. Podjetje Hermes SoftLab d.o.o. je z modelom FDD velike projekte razvoja programskih rešitev razdelilo na manjše dele razvoja ali funkcionalnosti. S tem si je podjetje omogočilo vzporedni in ločeni razvoj programske rešitve. Glede na to, da se podjetje v zadnjih leti močno poslužuje zunanjega izvajanja, je potrebno tudi spremeniti pogled na sestavo delovnih procesov. Timi postajajo virtualni in taka je tudi sama organizacija podjetja. Funkcionalno voden razvoj programskih rešitev z modelom FDD podjetju omogoča lažjo vzpostavitev virtualnega razvoja programskih rešitev.

3.3.1 Funkcionalno voden razvoj programskih rešitev v podjetju Hermes SoftLab d.o.o.

Z uvedbo modela FDD je podjetje Hermes SoftLab d.o.o. razvoj celotne programske rešitve razdelilo na določene dele ali funkcionalnosti programske rešitve. Pri vsakem razvoju funkcionalnosti programske rešitve je podjetje omejilo število ljudi v timu in čas trajanja. Največje število razvijalcev naj ne bi presegalo 6 ljudi. Predvideni čas trajanja za izvajanje je v razponu od 3 do 5 mesecev, vendar pa je to odvisno od velikosti in zahtevnosti funkcionalnosti programske rešitve. Projektni vodja je odgovoren za upravljanje razvojnega cikla in vseh sprememb, ki se pojavijo med razvojem programske rešitve, vendar mora upoštevati določen čas in vire na projektu.

Razvoj funkcionalnosti se deli na več faz (Slika 13). Projekt se **začne** s podrobnim načrtovanjem med produktnim vodstvom in razvojnim timom. Pregleda se možnost

izpolnitve novih zahtev in zaostankov programske rešitve. Zahteve se razdeli na manjše enote (kjer je to potrebno) ter postavijo se prednostne naloge. Določi se mejnike v projektu, projektno strukturo, vloge in odgovornosti.

Pri fazi **raziskovanje funkcionalnosti** programske rešitve je potrebno potrditi predlagane možnosti izvajanja in oblikovalskih zasnov programske rešitve. Ugotoviti je potrebno ustrezno arhitekturo in osnovno idejo o tem, kako izvajati razvoj programske rešitve z izbrano arhitekturo.

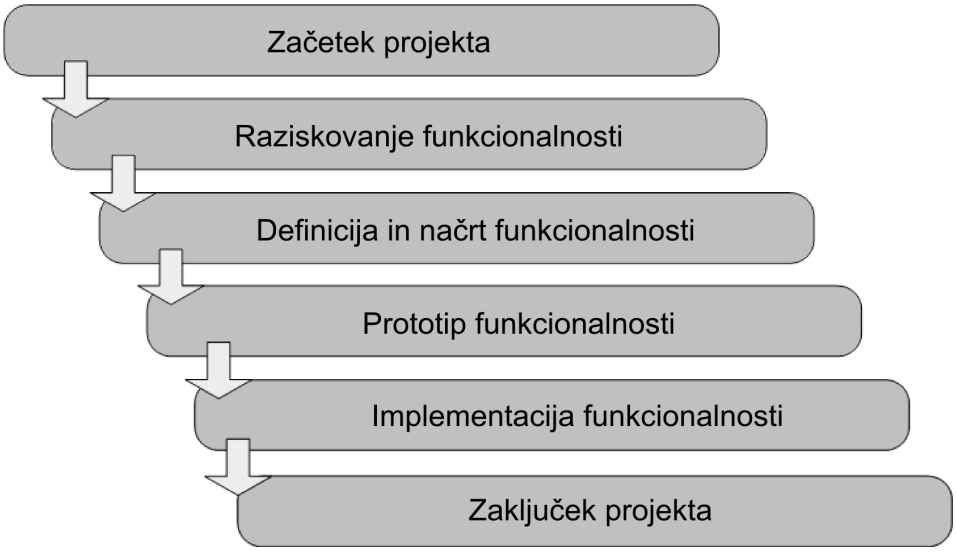
Faza **definicija in načrt funkcionalnosti** programske rešitve dokončno potrdi načrt, podrobno dokumentacijo in oblikovalsko zasnovo projekta, ki je sedaj pripravljena za razvoj programske rešitve.

V fazi **prototipa funkcionalnosti** se razvije prototip programske rešitve, ki je sinhroniziran z dejanskim načrtovanjem in zahtevam uporabnika.

V fazi **implementacije funkcionalnosti** se programska rešitev dejansko razvije. V tej fazi je potrebno paziti na zahteve uporabnika in cilje kakovosti programske rešitve. Opravi se temeljito testiranje in napiše se vsa potrebna dokumentacija.

Projekt se **zaključi**, ko se testiranje programske rešitve uspešno konča, dokumentacija je napisana, vse zahteve in cilji so izpolnjeni. Funkcionalnost je pripravljena za sprostitev programske rešitve.

Slika 13: Razvoj funkcionalnosti po modelu FDD v podjetju Hermes SoftLab d.o.o.



Vir: Hermes SoftLab d.o.o., interno gradivo, 2008.

Novi model razvoja programskih rešitev na podlagi funkcionalnosti je v podjetje Hermes SoftLab d.o.o. uvedel veliko sprememb v primerjavi s starim zaporednim modelom razvoja programskih rešitev (Tabela 4).

Tabela 4: Spremembe in izboljšave novega modela razvoja programskih rešitev

	Sprememba	**Izboljšanje**
A	Razdelitev na manjše projekte in razvojne enote programskih rešitev.	Boljše vodenje in napovedi pri razvoju programskih rešitev.
B	Funkcionalno vodenje in funkcionalni timi (namesto projektnega vodenja).	Jasni fokus razvojnega modela programskih rešitev.
C	Tehnični strokovnjaki za določeno področje znanja.	Pravilna usposobljenost, odgovornost in lastništvo.
D	Testiranje se pojavi že v fazi razvoja programskih rešitev.	Zgodnejše testiranje programskih rešitev pomeni manjše stroške popravkov in sprememb.
E	Razvoj funkcionalnosti programskih rešitev se konča najkasneje vsakih šest mesecev.	Uporabnik dobi programske rešitve bolj pogosto.
F	Boljša komunikacija in boljša povezanost v timu med razvijalci in vodstvom.	Bolj učinkovit prenos informacij.
G	Manjše začetno planiranje in večje sodelovanje z uporabnikom.	Osredotočenje na funkcionalnost programskih rešitev.
H	Kreiranje dokumentacije med razvojem programskih rešitev.	Dokumentacija se vedno ujema z razvojem in implementacijo programskih rešitev.
I	Neprestan razvoj ostale funkcionalnosti programskih rešitev.	Omogoči stalni razvoj funkcionalnosti programskih rešitev na projektu.

Vir:Hermes SoftLab d.o.o., interno gradivo, 2008.

3.3.2 Zunanje izvajanje in virtualni timi v podjetju Hermes SoftLab d.o.o.

Zunanje izvajanje del (angl. *»outsourcing«*) je v razvoju programskih rešitev že dolgo prisotno. Glavna motivacija pri odločitvi za zunanje izvajanje je običajno cena ali pa tudi

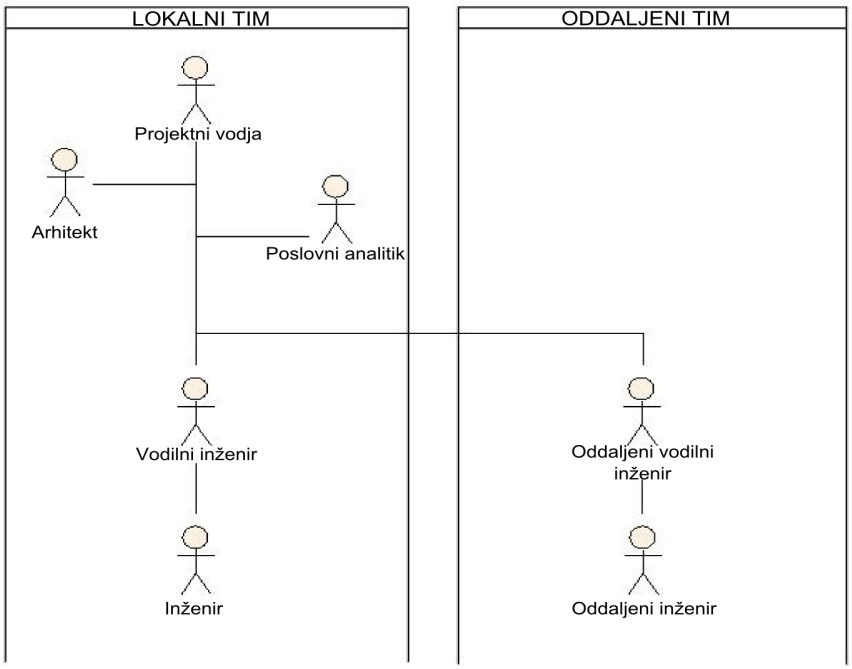
specialno znanje, ki ga poseduje zunanji izvajalec. Obstaja pa tudi možnost, da v določenem obdobju povpraševanje po storitvah razvoja programskih rešitev presega ponudbo in na lokalnem trgu enostavno ni možno najti dovolj kvalificiranih delavcev v kratkem času. Porast zunanjega izvajanja je dodatno podprta tudi z razvojem komunikacijskih tehnologij, ki smo jim priča v zadnjih nekaj letih (širokopasovne povezave, internetna telefonija, spletne aplikacije, orodja za kolaboracijo...).

Podjetje Hermes SoftLab d.o.o. ima dolgoletne izkušnje z zunanjim izvajanjem. Uporaba modela FDD pri razvoju programskih rešitev omogoča podjetju Hermes SoftLab d.o.o., da s pomočjo zunanjega izvajanja in virtualnih timov skrajša čas razvoja programskih rešitev ter ustrezno obvladuje prioritete razvoja programskih rešitev. Na ta način je izraba virov tudi bolj smotrna in dodana vrednost programske rešitev višja, saj prioritete narekujejo neposredno ekonomsko korist. Od začetkov, ko so bili predvsem ponudnik razvoja programskih rešitev, do uveljavljenega ponudnika programskih rešitev na ključ so se spoznavali z obema vidikoma zunanjega izvajanja. Nastopajo kot ponudnik storitve zunanjega izvajanja, hkrati pa zunanje izvajanje izkoriščajo tudi sami. Danes imajo tako razvojne centre v Bosni in Hercegovini, Srbiji in Črni gori.

3.3.2.1 Zunanje izvajanje projektov po modelu FDD v podjetju Hermes SoftLab d.o.o.

Slika 14 prikazuje tipično postavitev projektov po modelu FDD v podjetju Hermes SoftLab d.o.o. Oddaljen tim mora prevzeti lastništvo in odgovornost kompletnih funkcionalnosti programske rešitve. Vendar pa razvoj programskih rešitev ne more potekati uspešno brez kompetentnega vodje oddaljenega tima. V nasprotnem primeru je podjetje obsojeno na mikro vodenje oddaljenih ljudi, kar vzame ogromno časa in energije. Mikro vodenje je način upravljanja, kjer vodstvo natančno izvaja nadzor nad delom svojih podrejenih ali zaposlenih. Mikro vodenje se običajno uporablja kot negativni izraz (Chambers, 2004). Pojem mikro vodenja je mogoče razširiti na vse družbene kontekste, kjer ena oseba, ki ima neprimerno raven nadzora, vpliva na ostale člane tima. Če na oddaljeni lokaciji ni izkušenega vodje, je na dolgi rok edina smiselna rešitev izučitev takšne osebe.

Slika 14: Postavitev projektov po modelu FDD v podjetju Hermes SoftLab d.o.o.



Vir: Hermes SoftLab d.o.o., interno gradivo, 2008.

V idealnem primeru bi bila organizacija projekta razvoja programskih rešitev takšna, kot je prikazana na Sliki 14. Jedro projektne skupine sestavljajo projektni vodja, arhitekt in poslovni analitik, ki je v tesnem odnosu z uporabnikom. Razvijalci so organizirani v skupine od 3 do 5 inženirjev, od katerih eden prevzame vlogo vodilnega inženirja. Vsak takšni tim lahko samostojno prevzame neko funkcionalnost programske rešitve in jo v celoti implementira.

Kljub temu da je ureditev videti dokaj strogo in hierarhično, v praksi temu ni tako. Komunikacija seveda poteka na vseh nivojih. Vsak razvijalec seveda lahko komunicira tudi z arhitektom, projektnim vodjem ali analitikom (ker je analitik zastopnik uporabnika, bi bilo celo potrebno, da z njim komunicira). Dejansko se na manjših projektih nekatere od teh vlog tudi združijo (projektni vodja je hkrati analitik, arhitekt je eden od vodilnih inženirjev). Vendar pa na velikih projektih razvoja programskih rešitev kljub vsemu podjetje običajno skuša ustvariti nekoliko bolj formalizirano obliko komunikacije. Na ta način naj bi se vsakodnevne, rutinske odločitve sprejemale na nivoju delovne skupine inženirjev, kadar potrebujejo tehnično pomoč ali pa ima odločitev vpliv na potek projekta, pa se v reševanje vključita tudi arhitekt ter projektni vodja. Stranski učinek prevzemanja odgovornosti s strani oddaljenega tima je tudi manjši nadzor nad njihovim delom oziroma izgube nadzora nad vso implementacijo. Zanimivo je, da nekateri sodelujoči pri razvoju programskih rešitev nikoli ne morejo priti preko te točke ter enostavno želijo obdržati popoln nadzor, čeprav jih to vodi v neučinkovitost in mikro vodenje. To je odločilna točka,

na kateri se podjetje v vlogi razvijalca programskih rešitev odloči, ali sploh še želi koristiti zunanji razvoj programskih rešitev.

Kljub vsemu pa je pomembno, da se razvoj programskih rešitev oddaljenega tima ažurno spremlja, preizkuša in opazuje. Dokaj enostaven in preprost pristop je uporaba skupne izvorne kode, skupne razvojne podatkovne baze in rednih, pogostih kratkih sestankov vseh udeleženih pri razvoju programskih rešitev. Na ta način se vse kritične deviacije od skupnih načel dovolj hitro opazijo in korigirajo. Pri zunanjem razvoju programskih rešitev lahko iz majhnih nesporazumov kmalu zraste večji problem, zato je zelo pomembna stalna komunikacija med razvojnimi timi (Filev, 2006).

Vidiki zunanjega razvoja programskih rešitev so očitno dosti bolj zapleteni kot sama cena dela. Kljub navidezni ugodni ceni je lahko strošek oddaljenega razvoja programskih rešitev precej višji. Uspešnost zunanjega razvoja programskih rešitev je odvisna od izkušenosti projektnega tima s strani uporabnika in oddaljenega izvajalca. Ključnega pomena za uspeh takih aktivnosti je predvsem v zadostni komunikaciji skozi cel projekt, od specifikacij, preko razvoja do testiranja programskih rešitev.

3.3.3 Težave modela FDD pri razvoju programskih rešitev v podjetju Hermes SoftLab d.o.o.

V praksi razvoja programskih rešitev v podjetju Hermes SoftLab d.o.o. se je izkazalo, da je najtežje slediti načelu »sodelovanje z uporabnikom«, saj uporabnik programske rešitve običajno ni navzoč med razvojem le-te, pa tudi pogodbe se večinoma sklepajo za fiksen znesek in čas. To lahko privede do resnega problema pri razvoju programske rešitve, saj podjetje in uporabnik drugače vidita projekt kot celoto. Podjetje zato skozi celoten projekt razvoja programske rešitve določi poslovnega analitika, ki je zelo dober približek kooperativnega uporabnika. Naloga poslovnega analitika je čim bolj približati problematiko uporabnika programske rešitve razvojnemu timu. Na ta način podjetje smatra analitika kot podaljšek uporabnika, tako da ima lahko podjetje vsaj enega »predstavnika« uporabnika, ki je vedno na voljo za komunikacijo pri razvoju programskih rešitev.

Eden od poglavitnih težav vseh agilnih metodologij razvoja programskih rešitev je učinkovita komunikacija znotraj razvojnega tima. Kadar je tim ločen na več lokacij, je komunikacija še posebej pomembna. Praksa rednih jutranjih kratkih sestankov se je izkazala kot učinkovit način tako povezovanja kot koordinacije tima. Po eni strani na ta način podjetje gradi pripadnost timu in zavzemanju za isti cilj, po drugi strani pa doseže oster fokus celotnega tima na aktualne naloge razvoja programske rešitve. Ker na takšnih dnevnih sestankih sodeluje tudi vodstvo projekta (projektni vodja, arhitekt in poslovni analitik), sestanki preprečujejo »zaprtost« vodstva v svoj svet, ampak ga dnevno soočajo z

morebitnimi težavami in tveganji. Na ta način je tveganja možno identificirati in omiliti dosti prej kot sicer. Seveda pa se mora podjetje Hermes SoftLab d.o.o. dosledno držati načela, da so ti sestanki namenjeni dnevnemu pregledu dela in ne konkretnemu reševanju problemov pri razvoju programskih rešitev. Rešitve konkretnih problemov se vedno iščejo na ločenih sestankih.

Osebni odnosi med člani tima potrebujejo določen čas, da dozorijo. Fizična oddaljenost in razlika v kulturi pri tem seveda ne pomagata, zato je podjetje prešlo na interno uporabo mehanizma, ki ga sicer izkoriščamo tudi pri prodaji strankam. Gre za koncept »ambasadorjev«. To so posamezniki, ki se ob pričetku projekta razvoja programskih rešitev zberejo skupaj (bodisi lokalno ali na oddaljeni lokaciji) in postavijo temelje projekta. To vključuje tako spoznavanje nove domene, določanje obsega in ciljev projekta, organizacijo tima, dogovor o uporabljenih orodjih, metodologiji, tehnologijah... Na ta način se pospeši prenos znanja in idej ter se krepi medosebne stike. Nato se ambasador vrne na svojo lokacijo in z lokalnim timom nadaljuje pri razvoju programskih rešitev. Ambasadorji tudi pomagajo pri vzpostavitvi medsebojnega odnosa in razumevanja, pomagajo pa tudi premagovati morebitne predsodke ali občutke več/manjvrednosti. Oddaljen tim namreč zelo dobro čuti, kdaj ga podjetje smatra kot enakovreden dodatek lokalnemu timu in kdaj ga smatra zgolj v vlogi cenejše delovne sile.

3.4 Možnosti in izboljšave razvoja programskih rešitev v podjetju Hermes SoftLab d.o.o.

Podjetje Hermes SoftLab d.o.o. je s prenovo razvoja programskih rešitev veliko pridobilo. Današnji razvoj programskih rešitev se lahko kosa z vedno večjo konkurenco in nepredvidljivimi trgi. S pravilnimi odločitvami je mogoče današnji model FDD še izboljšati. Potrebno bi bilo stalno upravljanje in pregledovanje funkcionalnosti programskih rešitev, ki ob času sprostitve programskih rešitev še niso bile v celoti razvite in zaključene. Podjetje bi moralo prilagoditi razvoj in sprostitev programskih rešitev razvoju funkcionalnosti.

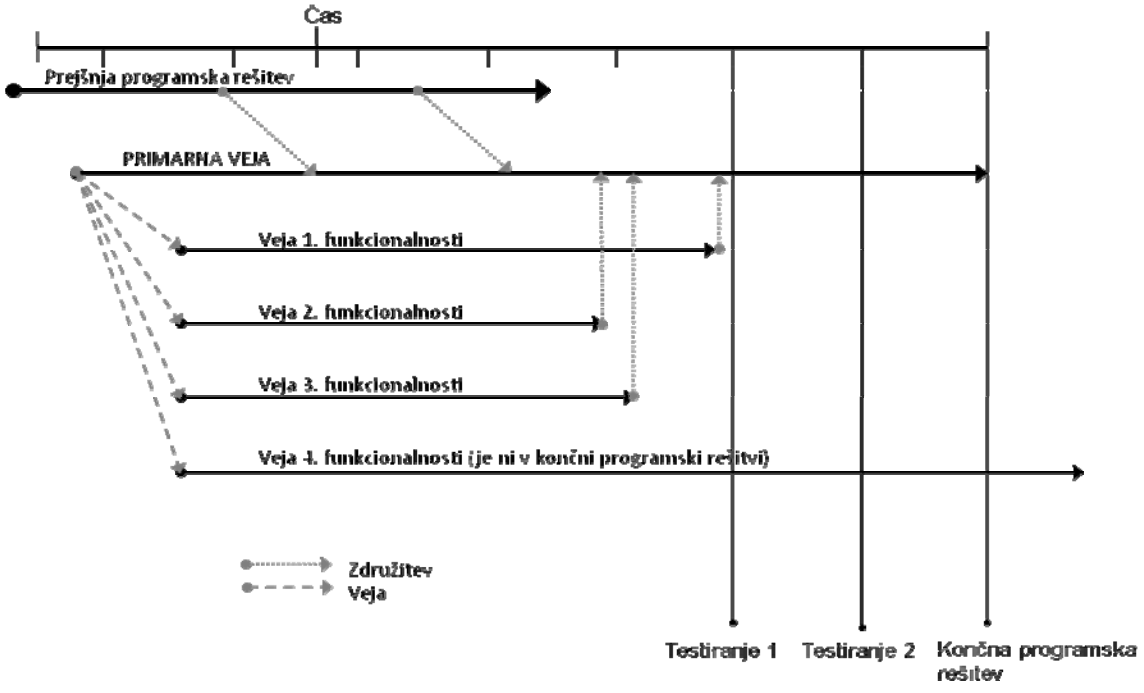
S postavitvijo kvalitetnega razvojnega tima je možno boljše oceniti napor in tveganje razvoja programskih rešitev ter stalno izboljševati proces razvoja programskih rešitev, ki temeljijo na izkušnjah, kar pripomore k dolgoročni učinkovitosti razvoja programskih rešitev. To omogoči hitro celotno sliko o kompleksnosti in razvojnem tveganju programskih rešitev. S prepoznavanjem in primerjanjem najboljših praks in procesov bi lahko izboljšali celoten razvoj programskih rešitev. To bi omogočili s pomočjo uvedbe mesečnih sestankov vseh razvojnih timov določene funkcionalnosti, v katerih bi primerjali delovanje v timih in iskali najboljše prakse in možne izboljšave procesa razvoja programskih rešitev.

3.4.1 Izboljšanje sedanjega modela FDD z uvedbo »modela vej«

Sedanji model razvoja programskih rešitev v podjetju Hermes SoftLab d.o.o. je razmeroma enostaven. Ko se razvoj določenih funkcionalnosti programske rešitve konča, se funkcionalnosti združijo v končno programsko rešitev. Vendar pa se zaradi ločenega razvoja funkcionalnosti programske rešitve lahko zgodi, da celotna programska rešitev ne deluje pravilno, kar lahko podaljša razvoj in popravilo končne programske rešitve tudi za več mesecev. To pomeni, da bo moral uporabnik počakati, da se po združitvi določenih nekompatibilnih funkcionalnosti programska rešitev popravi, kar pa povzroči, da se lahko stroški razvoja programske rešitve močno povečajo.

Podjetje bi težavo lahko rešilo z uvedbo modela vej (angl. *»Branch model«*) (Slika 15). Poglavitna lastnost modela je, da dovoli vzporedni razvoj ter omogoča lažji pregled in usklajevanje sprememb pri razvoju programskih rešitev (Berczuk & Appleton, 2003). V modelu vej neprestano obstaja primarna veja, ki je pripravljena za izdajo programske rešitve. Ko se zaključi določena funkcionalnost na razvojni veji, se le ta spoji s primarno vejo. Dejansko končno testiranje in spojitev se tako zgodi na primarni veji, iz katere tudi izhaja končna programska rešitev. Na primarno vejo se lahko spojijo celotne funkcionalnosti ali le določeni popravki programske rešitve. Razvoj programske rešitve na primarni veji omogoča hiter in učinkovit sistem za izdajo končne programske rešitve v razmeroma kratkem času.

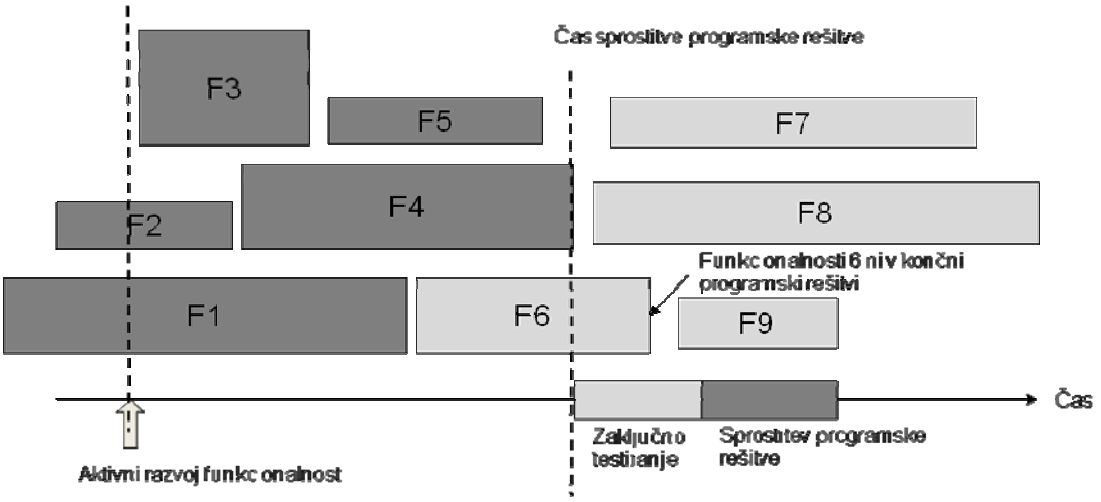
Slika 15: Model vej



Vir: C. Walrad & D. Strom, The Importance of Branching Models in SCM, 2002, str. 33.

Model vej tako omogoči, da razvoj različnih funkcionalnosti programske rešitve poteka nemoteno in bolj pregledno (Slika 16). Vsaka funkcionalnost (F1, F2,…) se razvija neodvisno od druge. Zaradi neodvisnega razvoja od glavnega razvojnega projekta omogoča večjo prožnost pri izbiri modelov razvoja programske rešitve. Funkcionalnosti programske rešitve so v celoti razvite in preizkušene na ločenih podružnicah, nato pa se spojijo na primarno vejo. Ko so določene funkcionalnosti programske rešitve zaključene ali ko uporabnik zahteva sprostitev programske rešitve, je končna programska rešitev že na voljo na primarni veji. Sledi zaključno testiranje programske rešitve ter sprostitev programske rešitve končnemu uporabniku. Funkcionalnosti programske rešitve, ki ob času sprostitve programske rešitve še niso v celoti razvite in zaključene, bodo izdane v naslednji verziji ali popravku programske rešitve. Podjetje Hermes SoftLab d.o.o. lahko tako poljubno določa čas sprostitve programske rešitve, kar pripomore k večji konkurenčnosti in bolj prilagodljivemu razvoju programske rešitve.

Slika 16: Razvoj funkcionalnosti po modelu vej v podjetju Hermes SoftLab d.o.o.



Z uvedbo modela vej bi se podjetje Hermes SoftLab d.o.o. tako lahko odločilo katere funkcionalnosti si želi na primarni veji in s tem tudi pri končni programski rešitvi. Podjetje Hermes SoftLab d.o.o. bi lahko model vej uvedlo razmeroma hitro, saj ni potrebno korenito spremeniti sedanji model FDD. Potrebno bi bilo replicirati dosedanje razvojno okolje kar omogoči, da razvoj določene funkcionalnosti programske rešitve poteka popolnoma ločeno. Razvijalci lahko tako spreminjajo in popravljajo programsko kodo nemoteno od ostalega razvoja programske rešitve.

Pri sedanjemu modelu razvoja programske rešitve podjetje ne uporablja primarne veje, vendar se ob določenem času vse funkcionalnosti programske rešitve združijo v končno programsko rešitev. Z uporabo primarne razvojne veje si lahko Hermes SoftLab d.o.o. skrajša čas, ki je potreben za združitev vseh funkcionalnosti programske rešitve. Ko se funkcionalnost programske rešitve zaključi, se le ta združi s primarno vejo. Tako ima

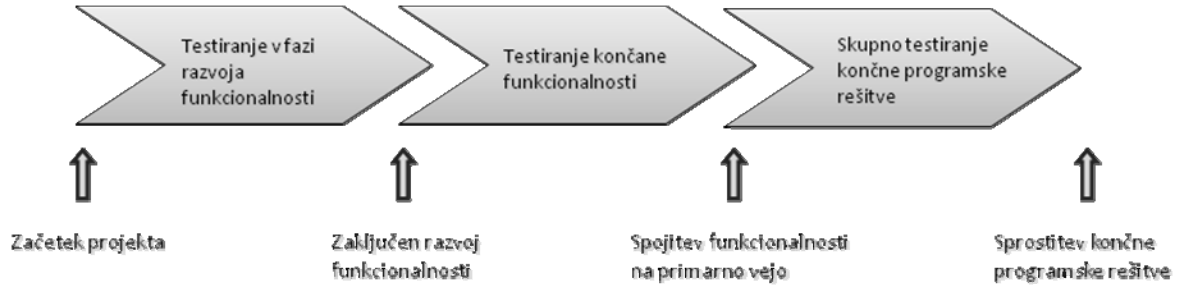
podjetje na voljo končno programsko rešitev veliko hitrejše kot pri obstoječemu modelu FDD.

3.4.2 Testiranje programske rešitve v fazi razvoja funkcionalnosti

Vendar pa razvoj programske rešitve po modelu vej predstavlja edinstven niz težav, ki ne nastajajo pri zaporednem razvoju programske rešitve. Poglavitna težava je v določitvi ciljne arhitekture programske rešitve. Po spojitvi vseh določenih funkcionalnosti programske rešitve se lahko pojavijo napake, ki jih je težko ločiti in pripisati določeni funkcionalnosti. Potrebno bi bilo napako raziskati in popraviti, vendar je v nekaterih primerih težko določiti pravi tim, zato je potrebno, da raziskavo in po možnosti tudi popravilo opravi tim, ki je problem odkril. Razvoj se paralelno odvija v različnih timih, ki razvijajo določeno funkcionalnost. Zaradi hkratnega razvoja lahko privede do razhajanja in nekompatibilnosti pri razvoju programske rešitve, kar privede do neučinkovitosti razvoja ter posledično povečanja stroškov projekta razvoja programske rešitve.

Podjetje Hermes SoftLab d.o.o. bi potrebovalo uvesti ločeno testiranje v fazi razvoja funkcionalnosti na svoji veji (Slika 17). Prvotno testiranje se naredi na razvojni veji, kjer se določena funkcionalnost programske rešitve ob razvoju v celoti testira in popravi. Prvo pravo testiranje se opravi, ko se spajajo vse zahtevane funkcionalnosti programske rešitve na primarno vejo (Slika 15). Ko teče drugo testiranje, se spojitve ustavijo. Primarna veja se uporablja samo za popravljanje napak in ne za razvoj določenih funkcionalnosti, kar pripomore k stabilnosti programske rešitve. Ko je funkcionalnost razvita in temeljito testirana na razvojni veji funkcionalnosti, jo je potrebno pregledati in odobriti spojitev na primarno vejo. S tem se onemogoči avtomatična spojitev in izdaja določene funkcionalnosti programske rešitve.

Slika 17: Testiranje programske rešitve v fazi razvoja funkcionalnosti



Podjetje Hermes SoftLab d.o.o. bi tako lahko poleg izboljšave razvoja programske rešitve izboljšalo tudi testiranje in zagotavljanje kvalitete programske rešitve. To lahko zagotovi s testiranjem v fazi razvoja funkcionalnosti programske rešitve, kjer se funkcionalnost

neprestano in sproti testira v fazi razvoja programske rešitve, kar omogoči sprotno odkrivanje in popravljanje napak. Testiranje v fazi razvoja funkcionalnosti programske rešitve lahko podaljša razvoj funkcionalnosti, vendar pa večja naložba v testiranje funkcionalnosti programske rešitve v fazi razvoja skrajša fazo testiranja, s tem pa tudi stroške odkrivanja in popravljanja napak. Kvaliteta funkcionalnosti programske rešitve se ob spojitvi na primarno vejo izboljša, kar privede do hitrejše izdaje programske rešitve.

Z uvedbo modela vej se vse funkcionalnost programske rešitve neprestano testirajo tudi na primarni veji, kar lahko omogoči podjetju Hermes SoftLab d.o.o., da ima na voljo končno programsko rešitev, ko jo potrebuje. Pri obstoječemu modelu FDD je za končno programsko rešitev najprej potrebno vse zahtevane funkcionalnosti programske rešitve združiti, jih testirati ter popraviti morebitne napake, kar pa lahko podaljša razvoj programske rešitve.

SKLEP

Projekti s področja informacijske tehnologije in razvoja programskih rešitev postavljajo projektnemu vodenju velik izziv. Razvoj programskih rešitev poteka v dinamičnem, hitro spreminjajočem se okolju, brez dobro definiranih industrijskih standardov in brez zanesljivih kvalitativnih podatkov iz sorodnih projektov. Virtualne organizacije so gotovo prihodnost in želja podjetij, katerih cilj je vedno naravnan k večjemu zaslužku in čim manjšim stroškom poslovanja. Tehnologija se praktično spreminja iz dneva v dan, tako da panoga zahteva nenehno sledenje spremembam in izobraževanje zaposlenih. Vodenje v virtualnem timu je izraženo skozi tehnologijo, zato morajo vodje timov razumeti tehnologijo, da jo lahko koristno uporabijo. Različne ideje o uporabi tehnologije omogočajo različno razumevanje tehnologije, predvsem v odnosu do dela, ki ga opravlja tim. Vsaka ideja lahko ponudi različne rešitve pri usklajevanju specifičnih značilnosti tehnologije, naloge pri razumevanju tega, kako uskladitev pomaga pri boljših rezultatih, razvoju razmišljanja v timu in izrazu komunikacije. Projektni pristop razvoja programskih rešitev pomeni delo v dinamičnem okolju, izraža potrebe po visoki učinkovitosti, zahteva optimalno izrabo kadrov in hiter pogled v tekoče stanje del na projektu.

Glede na teoretične osnove s področja projektnega vodenja pri razvoju programskih rešitev in praktičnih izkušenj je v delu izpostavljenih 5 glavnih dejavnikov neuspeha projektov s področja razvoja programskih rešitev:

- nerealen plan,
- nedefiniran proces razvoja,
- slabo opredeljene uporabniške zahteve,
- neustrezno testiranje in
- slaba komunikacija.

V prihodnje lahko računajo na uspeh pri razvoju programskih rešitev predvsem podjetja, ki imajo dobro definiran proces razvoja programskih rešitev in zelo dobro poznajo svoj način dela in trženja rešitev. Dobro definiran razvojni proces omogoča podjetju ponovljivost postopkov, ponovno uporabo programske kode in precejšnjo neodvisnost pred begom zaposlenih. S ponovno uporabo kode se drastično zmanjša število programskih napak, zmanjšajo se stroški testiranja programskih rešitev, poveča pa se tudi zadovoljstvo zaposlenih, ki jim ni potrebno vedno znova razvijati »istih« programskih rešitev. Seveda pa je pogoj za takšno delo precej široko razmišljanje v fazi načrtovanja programskih rešitev in usmerjenost podjetja za uspeh na daljši časovni rok.

Drugi ključni segment pri razvoju programskih rešitev ostaja model ocenjevanja rokov in stroškov projekta. Oceno je potrebno podati v zgodnji fazi projekta, ko uporabniške zahteve še niso dobro definirane. Podjetja, ki bodo razvila ustrezne modele za oceno rokov in stroškov projekta bodo, vsekakor imela konkurenčno prednost na trgu, saj bodo lažje in uspešneje izpolnila pričakovanja in želje naročnikov. Na trgu se bo pojavljalo vedno več ponudnikov in metod razvoja programskih rešitev, tako da bo izbira ustrezne rešitve vedno težja. V tem pogledu je za podjetje Hermes SoftLab d.o.o. zelo pomembno definirati ocenjevalno metodo, ki bo z določeno verjetnostjo omogočala podjetju, da se odloči za najbolj primerno programsko rešitev. Kadar ocenjujemo primernost posamezne metodologije razvoja informacijskega sistema, je vedno potrebno v razmislek vzeti tudi posamezne parametre projekta. Kakšen je obseg projekta, kako je določeno pogodbeno razmerje med kupcem in izvajalcem, koliko se bodo zahteve spreminjale, so le nekatera vprašanja, ki vplivajo na izbiro pravšnje metodologije razvoja programskih rešitev.

Študije so pokazale, da so klasične metodologije vsebujejo preveč podrobno predstavljene projekte programskih rešitev. Pri nadaljnjih raziskavah je potrebno preučiti predvsem agilne metodologije razvoja programskih rešitev in jih prilagoditi poslovanju določenega podjetja. Pri razvoju programskih rešitev je za uspešnost projekta ključnega pomena prava izbira metode razvoja programskih rešitev, ki je prilagojena razvoju programskih rešitev v določenemu podjetju ter pravilna sestava razvojnega tima. Pri sestavi razvojnega tima je pomemben tudi vidik zunanjega razvoja programskih rešitev. Zunanji razvoj programskih rešitev je načeloma dosti bolj zapleten kot sama cena dela. Kljub navidezni ugodni ceni je lahko strošek zunanjega razvoja precej višji, kar je odvisno od zrelosti projektne skupine tako na strani uporabnika kot na strani oddaljenega izvajalca.

V magistrski nalogi sem opisal in predstavil različne pristope k razvoju programskih rešitev s pomočjo klasičnih in agilnih metodologij razvoja programskih rešitev. Ugotovil sem, da ne obstaja enotna in najboljša metodologija razvoja programskih rešitev, vendar pa je v magistrski nalogi prikazano, da agilne metodologije razvoja programskih rešitev lahko zagotavljajo nov način reševanja težav razvoja programskih rešitev. Metodologije razvoja programskih rešitev, ki se uporabljajo danes, morajo biti kar najbolj prilagodljive, a hkrati

še vedno toliko formalne in dosledne, da zagotavljajo zaključek projekta z razpoložljivimi viri in vpogled v stanje projekta v vsakem trenutku. Ustrezni model razvoja programskih rešitev je osnova za merljivost projektnih rezultatov, analizo napak in učenje o projektnem načinu dela v podjetju ter omogoča nenehno izboljševanje razvojnega procesa. Analiza in uporabnost novega modela vej razvoja programskih rešitev v podjetju Hermes SoftLab d.o.o. je pokazala, da bi podjetje lahko povečalo kakovost programskih rešitev, predvsem pa bi podjetje zaradi vzporednega razvoja programskih rešitev in uporabe primarne veje omogočilo hitrejši razvoj programskih rešitev. Podjetju Hermes SoftLab d.o.o. uvedba modela vej omogoči, da ima na voljo končno programsko rešitev, ko jo potrebuje.

Z uvedbo modela vej bi se podjetje Hermes SoftLab d.o.o. tako lahko odločilo katere funkcionalnosti si želi na primarni veji in s tem tudi pri končni programski rešitvi. Pri sedanjemu modelu razvoja programske rešitve podjetje ne uporablja primarne veje, vendar se ob določenem času vse funkcionalnosti programske rešitve združijo v končno programsko rešitev. Z uporabo primarne razvojne veje si lahko Hermes SoftLab d.o.o. skrajša čas, ki je potreben za združitev vseh funkcionalnosti programske rešitve. Ko se funkcionalnost programske rešitve zaključi, se le ta združi s primarno vejo. Tako ima podjetje na voljo končno programsko rešitev veliko hitrejše kot pri obstoječemu modelu FDD. Testiranje v fazi razvoja funkcionalnosti programske rešitve lahko podaljša razvoj funkcionalnosti, vendar pa večja naložba v testiranje funkcionalnosti programske rešitve v fazi razvoja skrajša fazo testiranja, s tem pa tudi stroške odkrivanja in popravljanja napak. Z uvedbo modela vej se vse funkcionalnost programske rešitve neprestano testirajo tudi na primarni veji, kar lahko omogoči podjetju Hermes SoftLab d.o.o., da ima na voljo končno programsko rešitev, ko jo potrebuje. Pri obstoječemu modelu FDD je za končno programsko rešitev najprej potrebno vse zahtevane funkcionalnosti programske rešitve združiti, jih testirati ter popraviti morebitne napake.

Na primeru podjetju Hermes SoftLab d.o.o. sem prikazal, da je ključnega pomena za uspeh razvoja programskih rešitev, predvsem v zadostni komunikaciji skozi cel projekt od specifikacij, preko razvoja do testiranja. Z uvedbo modela FDD je podjetje Hermes SoftLab d.o.o. razvoj celotne programske rešitve razdelilo na določene dele ali funkcionalnosti programske rešitve. Pri tem je podjetje razdelilo delo na manjše razvojne time, kar jim omogoči boljšo komunikacijo in s tem povezan kvalitetnejši razvoj programske rešitve. Nenazadnje, kljub napredni tehnologiji, so medčloveški stiki in skupno risanje po tabli še vedno ključ do uspeha. Verjetno je tako največji izziv podjetij pri razvoju programskih rešitev pravilna in uspešna predstavitev procesa razvoja programskih rešitev, ki temelji na agilnih metodologijah.

LITERATURA IN VIRI

1. Abrahamsson, P., Salo, O. & Ronkainen, J. (2002). Agile software development methods. *VTT Electronics,* University of Oulu, Finland.
2. Ambler, S. (2002). Lessons in Agility from Internet-Based Development. *IEEE Software, 19* (2), str. 66-73.
3. Avison, D. E., Fitzgerald, G. (2002). *Information Systems Development: Methodologies, Techniques and Tools.* London: McGraw-Hill.
4. Bavec, C. (2002). *An assessment of the organization virtuality with three different reference models.* Ljubljana: Informatica 4, str. 26.
5. Beck, K. (1999a). Embracing Change With Extreme Programming. *IEEE Computer,* str. 70-77.
6. Beck, K. (1999b). *Extreme Programming Explained.* Boston: Addison-Wesley Professional.
7. Beck, K. & Fowler, M. (2000). *Planning Extreme Programming.* Boston: Addison-Wesley Professional.
8. Berczuk, S. & Appleton, B. (2003). *Software Configuration Management Patterns: Effective Teamwork, Practical Integration.* Boston: Addison-Wesley Professional.
9. Bjorner, D. (2005). *Software Engineering 3: Domains, Requirements, and Software Design.* New York: Springer.
10. Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer,* str. 61-72.
11. Cascio, W. & Shurygailo, S. (2002). *E-Leadership and Virtual Teams.* London: Elsevier Science.
12. Chambers, H. (2004). *My Way or the Highway: The Micromanagement Survival Guide.* San Francisco: Berrett Koehler Publishers.
13. Cockburn, A. (2002). *Agile Software Development.* Boston: Addison-Wesley Professional.
14. Curtis, G. & Cobham, D. (2008). *Business information systems: analysis, design, and practice.* New York: Pearson Education.
15. Daft, L. R. (1991). *Menagement (2nd ed.)*. Orlando: Dryden Press.
16. DeMarco, T. (1996). The role of software development methodologies: past, present and future. *Proceedings of the 18th International Conference on Software Engineering*. Berlin: IEEE Computer Society, str. 2-4.
17. Dorfman, M., & Thayer, R. H. (2000). *Software engineering.* Indianapolis: Wiley Publishing, Inc.
18. Elliott, G. (2004). *Global Business Information Technology: an integrated systems approach.* Boston: Addison-Wesley Professional.

19. Filev, A. (2006). Using an Agile Software Process with Offshore Development. Najdeno 08.10.2009 na spletnem naslovu http://msdn2.microsoft.com/en-us/library/bb245671.aspx
20. Fitzgerald, B. (1999). *Systems Development Methodologies: the problem of tenses.* Bradford: MCB UP Ltd; Information Technology & People.
21. Fowler, M. (2006). Adopting and Benefiting from Agile Processes in Offshore Software Development. Najdeno 08.10.2009 na spletnem naslovu http://martinfowler.com/articles/agileOffshore.html
22. Frankovich, J. (1998). *Synopsis of the Carnegie Mellon University course on Requirements Engineering.* Pittsburgh: Carnegie Mellon University.
23. Gould, D. (2005). Virtual Organization. Najdeno 08.10.2009 na spletnem naslovu http://www.seanet.com/~daveg/vrteams.htm
24. Green, D. & DiCaterino, A. (1998). *A Survey of System Development Process Models.* New York: University at Albany.
25. Guimarães, L. R., & Vilela, P. R. S. (2005). *Comparing software development models using CDM.* New Jersey: Newark, Sigite, str. 339-347.
26. Hermes SoftLab d.o.o., interno gradivo (2009).
27. *Hermes SoftLab d.o.o.* Najdeno 08.10.2009 na spletnem naslovu http://www.hermes-softlab.si
28. Highsmith, J. A. (2000). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems.* New York: Dorset House Publishing.
29. Highsmith, J. & Cockburn, A. (2001). Agile Software Development: The Business of Innovation. *Computer 34*, str. 120-122.
30. Jayaswal, B. K. & Patton, P. C. (2006). *Software Development Methodology Today.* New Jersey: Prentice Hall.
31. Keyes, J. (2002). *Software Engineering Handbook (1st ed.).* New York: Auerbach Publications.
32. Kovačič, A. (1998). *Informatizacija poslovanja.* Ljubljana: Ekonomska fakulteta.
33. Kovačič, A. (2004). *Prenova in informatizacija poslovanja.* Ljubljana: Ekonomska fakulteta.
34. Land, F. (1998). *A Contingency Based Approach to Requirements Elicitation and Systems Development.* London: School of Economics, Systems Software, str. 3-6.
35. Laplante, P. A., Neill, C. J. (2004). *The Demise of the Waterfall Model Is Imminent.* New York: Queue.
36. Larman C. (2004). *Agile and Iterative Development: A Manager's Guide.* Boston: Addison Wesley Professional.
37. Larsen, R. T. K. & Mcinerney, R. C. (2002). *Preparing To Work in The Virtual Organization.* Amsterdam: Information and Management 39, str. 445-456.
38. Lipičnik, B. (2002). *Organizacija podjetja.* Ljubljana: Ekonomska fakulteta.
39. Maddux, B. R. (1992). *Oblikovanje tima.* Ljubljana: Mladinska Knjiga.

40. Manifesto for Agile Software Development. *Agile Alliance.* Najdeno 15.10.2009 na spletnem naslovu http://www.agilemanifesto.org
41. Martin, J. (1991). *Rapid Application Development.* Middlesbrough: Macmillan Coll Div.
42. Martin, C. R. (2003). *Agile Software Development: Principles, Patterns and Practices.* New Jersey: Prentice Hall.
43. Martin, C. R., & Martin, M. (2006). *Agile Principles, Patterns, and Practices in C# (1st ed.).* New Jersey: Prentice Hall.
44. McConnell, S. (1996). *Rapid Development: Taming Wild Software Schedules.* ZDA: Washington: Microsoft Press.
45. Molan, G. (2008). *Improvements of Software Testing for LSP; The example in the case of internationalization and localization.* Ljubljana: Hermes Softlab d.o.o. Research Group, Hermes Softlab d.o.o.
46. Mowshowitz, A. (1994). *Virtual organization: a vision of management in the information age.* New York: The Information Society 10, str. 267-288.
47. Naur, P. & Randell, B. (1968). *Software Engineering: Report of a conference sponsored by the NATO Science Committee.* Brussels: Scientific Affairs Division.
48. Nelson, G. (2007). *Reexamining the Waterfall Model.* Maryland: University of Maryland.
49. Palmer, S. R. & Felsing, J. M. (2002). *A Practical Guide to Feature-Driven Development.* Upper Saddle River, New Jersey: Prentice Hall.
50. Papatsoutsos, D. (2001). *Information Systems Development Methodologies in the Age of Digital Economy.* Athens: University of Athens: Department of Informatics.
51. Papows, J. (1999). *Enterpise.com: Market Leadership in The Information Age.* London: Nicolas Brealey.
52. Parnas, D. L. & Clements, P. C. (1986). A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering archive Vol. 12,* 251-257.
53. Paul, R. J. (1993). Why Users Cannot »Get What They Want«. *ACM SIGIOS Bulletin 14,* 8-12.
54. Paulk, M. C., Curtis, B., Chrissis, M. B., Weber, C. V. (1993). *Capability Maturity Model for Software.* Software Engineering Institute, str. 18.
55. Reade, C. & Froome, P. (1990). Formal methods for reliability. London: Elsevier Applied Science. Software reliability handbook (51-82).
56. Reilly, R. R. & Ryan, M. (2007). *Leadership in Virtual Teams.* Massachusetts: Wesley J. Howe school of Technology management.
57. Royce, W. W. (1970). *Managing the Development of Large Software Systems.* California: Proceedings of IEEE WESCON.
58. Salmela, H., Lederer A. L., Reponen, T. (2000). Information systems planning in a turbulent environment. *European Journal of Information Systems 9,* 3-15.

59. Schach, S. R. (1999). *Classical and Object-Oriented Software Engineering with UML and C++*. London: McGraw-Hill.
60. Schwaber, K. & Beedle, M. (2002). *Agile Software Development With Scrum*. Upper Saddle River, New Jersey: Prentice Hall.
61. Stapleton, J. (1997). *Dynamic systems development method – The method in practice.* Boston: Addison Wesley.
62. Strmec, J. (2005). Virtualne organizacije in pomen virtualnih timov. Ljubljana: Ekonomska fakulteta (zaključna strokovna naloga visoke poslovne šole).
63. Walrad, C. & Strom, D. (2002). The Importance of Branching Models in SCM. *IEEE Computer, str. 31-38.*
64. Williams, L. & Cockburn, A. (2003). Agile Software Development: It's about Feedback and Change. *IEEE Computer,* str. 39-43.
65. Whitten, J. L., Bentley, L. D., Dittman, K. C. (2004*). Systems Analysis and Design Methods (6th ed.).* London: McGraw-Hill.
66. Zigurs, I. (2003). *Leadership In Virtual Teams: Oxymoron or Opportunity?* London: Elsevier Science, str. 339-351.

PRILOGA 1

Seznam kratic

ASD – Adaptive Software Development (Prilagodljivi razvoj programskih rešitev)
DSDM – Dynamic System Development Method (Dinamični sistemski model razvoja programskih rešitev)
ERS – External Reference Specification (Zunanja specifikacija)
FDD – Feature Driven Development (Funkcionalno voden razvoj)
FURPS+ – Functionality, Usability, Reliability, Performance, Supportability, Portability, Internationalisation & Localisation (Funkcionalnost, uporabnost, zanesljivost, zmogljivost, podpora, prenosljivost, internacionalizacija in lokalizacija)
HLA – High Level Architecture (Visokonivojska arhitektura)
ISO – International Organization for Standardization (Mednarodna organizacija za standardizacijo)
NATO – North Atlantic Treaty Organisation (Organizacija severnoatlantskega sporazuma)
RAD – Rapid application development (Hitri razvoja programskih rešitev)
SPI – Smart plug-in (Programski sistem za mrežno spremljanje in upravljanje mrežnih aplikacij)
XP – Extreme Programming (Ekstremno programiranje)