UNIVERZA V LJUBLJANI   
FAKULTETA ZA DRUŽBENE VEDE

Andraž Gruden

**Proces razvoja video iger**

Diplomsko delo

Ljubljana, 2018

UNIVERZA V LJUBLJANI   
FAKULTETA ZA DRUŽBENE VEDE

Andraž Gruden

Mentor: izr. prof. dr. Jaroslav Berce

**Proces izdelave video iger**

Diplomsko delo

Ljubljana, 2018

Zahvala

SUMMARY

*Kardeljeva ploščad 5*

*1000 Ljubljana, Slovenija*

*telefon 01 58 05 128*



**I Z J A V A O A V T O R S T V U**

diplomskega dela

Spodaj podpisani/-a      , z vpisno številko      , sem avtor/-ica diplomskega dela z naslovom:      .

S svojim podpisom zagotavljam, da:

* je predloženo diplomsko delo izključno rezultat mojega lastnega raziskovalnega dela;
* sem poskrbel/-a, da so dela in mnenja drugih avtorjev oz. avtoric, ki jih uporabljam v predloženem delu, navedena oz. citirana v skladu s fakultetnimi navodili;
* sem poskrbel/-a, da so vsa dela in mnenja drugih avtorjev oz. avtoric navedena v seznamu virov, ki je sestavni element predloženega dela in je zapisan v skladu s fakultetnimi navodili;
* sem pridobil/-a vsa dovoljenja za uporabo avtorskih del, ki so v celoti prenesena v predloženo delo in sem to tudi jasno zapisal/-a v predloženem delu;
* se zavedam, da je plagiatorstvo – predstavljanje tujih del, bodisi v obliki citata bodisi v obliki skoraj dobesednega parafraziranja bodisi v grafični obliki, s katerim so tuje misli oz. ideje predstavljene kot moje lastne – kaznivo po zakonu (Zakon o avtorskiin sorodnih pravicah (UL RS, št. 16/07-UPB3, 68/08, 85/10 Skl.US: U-I-191/09-7, Up-916/09-16)), prekršek pa podleže tudi ukrepom Fakultete za družbene vede v skladu z njenimi pravili;
* se zavedam posledic, ki jih dokazano plagiatorstvo lahko predstavlja za predloženo delo in za moj status na Fakulteti za družbene vede;
* je elektronska oblika identična s tiskano obliko diplomskega dela ter soglašam z objavo diplomskega dela v zbirki »Dela FDV«;
* je diplomsko delo lektorirano in urejeno skladno s fakultetnim Pravilnikom o diplomskem delu.

V Ljubljani, dne       Podpis avtorja/-ice:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1 UVOD

Računalniška tehnologija je danes prisotna v domala vseh sferah družbenega življenja. Eskalacija tehnološkega razvoja je v zadnjih letih privedla do tega, da se je računalniška tehnologija iz laboratorijev preselila v naše domove in znašla v sleherni sredini našega socialnega življenja. Hedonično dimenzijo uporabnosti računalniške tehnologije v prvi vrsti predstavljajo video igre. Igrajo se z namenom doživljanja novih izkušenj, pridobivanja občutka dosežka, za interakcijo s prijatelji, družino in za preživljanje prostega časa (Fullerton 2014, 1). So vsestranski informacijski sistemi (Aleem, Capretz, in Ahmed 2016), ki poleg zabave služijo tudi kot orodja za informiranje, oskrbo zdravja in medicine, usposabljanja, in učenja.

Igre kot učni pripomočki[[1]](#footnote-1) pozitivno vplivajo na analitične, prostorske, strateške, psihomotorične sposobnosti, kratkoročni in dolgoročni spomin, vizualno pozornost, identifikacijo, reševanje problemov in socialne spretnosti (Madani, Pierce, in Mirchi 2017, 4). Igre imajo velik potencial pri podpori učenja konceptualnega razumevanja, procesov, praks, epistemologije in ne nazadnje obnašanje (Clark, Tanner-Smith, in Killingsworth 2016, 79).

Igre so se izkazale kot avtentična orodja za ohranjanje in spodbujanje zdravja. Spodbujajo lahko kognitivne in praktične spretnosti profesionalcev. Igra *911 Paramedic* zajema 35 medicinskih primerov in 40 realističnih medicinskih orodij. Igralcu omogoča realistično izvajanje protokolov zdravljenja (Gareth Schott in Darrin Hodgetts 2006, 311). Spodbujajo se simulacije z VR[[2]](#footnote-2) pripomočki z namenom praktičnega izobraževanja neizkušenih kirurgov pri katerih je ogrožanje zdravja pacientov med procesom izključeno (Ma in Zheng 2011, 171). Obstajajo tehtni dokazi, da lahko igre posredujejo vplive, ki uporabnika za določena zdravstvena stanja naredijo bolj učinkovitega (Ma in Zheng 2011, 311). Igralni pripomočki kot so *Dance Dance Revolution, Sony Eyetoy, Nintendo Wii* spodbujajo zdravje s ponujanjem alternativnih pristopov fizičnih aktivnosti. Igra *Rex Ronan* spodbuja zdrav življenjski slog s spodbujanjem proti kadilskih nagnjenj (Gareth Schott in Darrin Hodgetts 2006, 311). Ulbin (2017) predstavi nevsiljiv način spremembe življenjskega sloga z izdelavo igre, ki bi pomagala otrokom s prekomerno telesno težo. Številne raziskave nakazujejo tudi na terapevtske potenciale. Igre pozitivno vplivajo na kognitivne dejavnosti, mobilnost, rehabilitacijo bolnikov, ki trpijo za izgubo ali oslabitev motoričnih funkcij (kap, parkinsonizem) (Salmon in dr. 2017, 45). Dokaz sta rešitvi BREATHING+, ki je nastal na pobudo podjetja Zdrav dih in Bimeo, slovenskega podjetja Kinestica. Obe rešitvi temeljita na rehabilitaciji zdravja s pomočjo igranja iger („Kinestica“ 2018; „Breathing Labs“ 2018).

Prihodki industrije iger vztrajno rastejo. Samo v ZDA so leta 2012 dosegli 14,8 milijard dolarjev, kar je zasenčilo filmsko industrijo (Fullerton 2014, XXV). Zadnje poročilo ocenjuje prihodek industrije iger na 30,4 milijarde dolarjev (ESA, 2017). Na porast vpliva tudi vzpon mobilne tehnologije. V letu 2010 je bilo prodanih več mobilnih naprav kot osebnih računalnikov. V zadnjem četrtletju je bilo po svetu prodanih 100,9 milijonov telefonov (L. Rakestraw, V. Eunni, in Kasuganti 2013, 2). S porastom prodaje teh naprav je sunkovito poskočilo povpraševanje po igrah. Leta 2011 je bilo vsak teden izdanih 15.000 novih aplikacij. (L. Rakestraw, V. Eunni, in Kasuganti 2013, 2). Na podlagi popularnosti mobilnih iger se vlagajo velike vsote v razvoj in promocijo, s čimer se je drastično dvignila tudi konkurenčnost na trgu (Soomro, Ahmad, in Sulaiman 2013, 247).

Industrija iger je nedvomno velika industrija, ki ustvarja visokokvalitetna delovna mesta. Gamasutra navaja, da so razvijalci iger za leto 2013 v Evropi v povprečju zaslužili več kot 45,000 dolarjev (Gamasutra 2017). Povprečje za EU28, ki ga navaja Eurostat pa je 38,000 dolarjev (Eurostat, 2017). Na podlagi poročila EY, je v Evropi zaposlenih 108,000 ljudi v industriji iger, ki ustvarijo preko 16 milijard evrov dobička. (EY 2017). Tako postane toliko bolj mikavno za vlagatelje kot tudi za razvijalce, da usmerijo svoje moči v razvoj iger. Rast je privedla do tega, da je tehnologija postala priročnejša in dostopnejša. Vedno več posameznikov je motiviranih za izdelavo lastnih iger (Aleem, Capretz, in Ahmed 2016, 55). Zaradi želje po hitrem vstopu na trg, vlagatelji pogosto pritiskajo na razvijalce. Posledično se zato podaljšujejo roki in slabo definirajo ocene zaključka projektov. (O’Hagan, Coleman, in O’Connor 2014, 182). Na roke vplivajo tudi slaba organizacija, upravljanje („What went wrong? A survey of problems in game development“ 2017, 4) in izbira metodologije. Igre s slabo razvojno metodologijo bolj verjetno presežejo finančna sredstva in časovne termine (Kanode in Haddad 2009, 556). Standish group je ocenil, da je bilo 30% projektov nedokončanih in 53% projektov, ki so presegli časovni termin s 189% presežkom sredstev (Petrillo in dr. 2008, 707). Postalo je izredno pomembno, da se proces izdelave iger izboljša zavoljo konkurenčnosti in finančnih ciljev organizacij (Aleem, Capretz, in Ahmed 2016).

Kompleksnost iger je eskalirala (Blow 2004, 29) in z njo razvojni procesi (O’Hagan, Coleman, in O’Connor 2014, 182). Organizacije zaradi neprilagojenih procesov pogosto razvijejo lastne procese, ki pa lahko vsebujejo zastarele tehnike in ne uporabljajo najboljših praks industrije razvoja (Sommerville 2010, 28, 30). Moramo poudariti, da ne obstaja en način razvoja iger (Aktaş in Orçun 2016, 249). Razlog je, da proces video iger ni podrobno definiran. (McAllister in White 2015, 14). Po naravi je proces razvoja iger nestrukturiran, intuitiven in organski proces, ki temelji na razvojnih praksah. To je bilo zadovoljivo v času, ko so bile razvojne ekipe in količine potrebnih virov za izgradnjo igre manjše (Aktaş in Orçun 2016, 240). Danes je napoved razpona projekta skoraj nemogoča (Kanode in Haddad 2009, 555). V nekaterih primerih lahko projekt vključuje tudi do tisoč ljudi in lahko traja več let (Kanode in Haddad 2009). Težavnost se lahko le še stopnjuje, če so ekipe geografsko razdeljene (O’Hagan, Coleman, in O’Connor 2014, 182). Igre so oblikovane s strani ekip izkušenih posameznikov, ki lahko vključujejo več visoko usposobljenih strokovnjakov z različnih področij, računalništva, umetnosti, medijskega oblikovanja, poslovanja (O’Hagan, Coleman, in O’Connor 2014, 182) in izobraževanja (Aslan in Balci 2015, 307). Za načrtovanje in upravljanje tako kompleksnih multidisciplinarnih projektov je potrebna metodologija, kjer ad hoc načini upravljanja ne pridejo v poštev (Aslan in Balci 2015, 307). Čeprav so bile določene dobre prakse prevzete od tradicionalnega razvoja programske opreme se te razlikujejo od razvoja iger (O’Hagan, Coleman, in O’Connor 2014, 182). V razvoju iger je močno vključeno ustvarjanje kreativnih vsebin, ki definirajo izgled igre (Aktaş in Orçun 2016, 249). Vizualna podoba igre je emocionalna pogodba[[3]](#footnote-3) formulirana med oblikovalcem, razvijalcem in končnim uporabnikom. Trenutno procesi nimajo orodja za zajem zahtev končnih uporabnikov, kar bi pripomoglo k izboljšanju kreativnega procesa ustvarjanja sredstev (O’Hagan in O’Connor 2015, 15). Vendar igre niso povsem le umetnost kot tudi ne produkt popolnega inženiringa. Razvoj igre je bolj podoben ustvarjanju izdelka s prepletanjem aspektov umetnosti, glasbe, programiranja, igranja in poslovnega upravljanja (Ramadan in Widyani 2013, 95). Igre bolj stremijo k proizvajanju uporabniške izkušnje kot uporabnosti. (O’Hagan, Coleman, in O’Connor 2014, 182). Poudarek je na evalvaciji uporabniške izkušnje in uporabo povratnih informacij, ki nato vodijo tok iteracij razvoja (O’Hagan, Coleman, in O’Connor 2014, 183). Pressman zagovarja, da so igre programska oprema, ki ponujajo zabavo (Ramadan in Widyani 2013, 95), vendar ni metode za določanje subjektivnega elementa zabave, na katero se osredotočajo oblikovalci iger. Zato je razvoj iger toliko bolj kompleksen v nasprotju s tradicionalnim razvojem programske opreme. Potrebno je razširiti podedovane tradicionalne tehnike razvoja programske opreme, da bi lahko podprli kreativni proces razvoja video iger („What went wrong? A survey of problems in game development“ 2017, 19). Razvoj iger potrebuje specifične usmeritve, ki temeljijo na najboljših praksah in ocenjevalni model za reševanje izzivov, s katerimi se soočajo razvijalci pri izvajanju trenutnih procesov (Ramadan in Widyani 2013, 95; Aleem, Capretz, in Ahmed 2016, 55). V kratki zgodovini discipline razvoja iger, usmeritve in prakse še niso bile povsem raziskane (Aleem, Capretz, in Ahmed 2016, 27). Posledično ni bilo izdelanega procesnega modela, ki bi temeljil na najboljših praksah razvoja iger. Takšen model, bi lahko zmanjšal čas razvoja, čas vstopa na trg ali celo izboljšal kvaliteto iger (O’Hagan in O’Connor 2015, 15).

Prav tako do nedavnega še ni bilo zrelostnega modela, ki bi direktno naslovil težave ocenjevanja in izboljšanja procesov (Aleem, Capretz, in Ahmed 2016, 58).

To je privedlo v motivacijo identifikacije usmeritev in praks ter predlaganje procesnega modela za igre, ki ga bomo v delu optimizirali s prvim[[4]](#footnote-4) primernim zrelostnim modelom za igre[[5]](#footnote-5). Delo je prikaz sistematičnega pristopa k izgradnji domenskega procesa, kateri kljub sistematizaciji in optimizaciji ohranja prostor za kreativnost.

2 METODOLOGIJA

Metodološki pregled pretekle literature je ključnega pomena vsakega akademskega raziskovanja. Področja kot so inženiring, so zaradi podcenjevanja pomembnosti metode pregleda literature kronično utrpela pomanjkanje raziskovalnih del, kar je otežilo teoretski in konceptualni napredek. Metoda nam omogoča pregled preteklih dognanj, ki služijo kot dobri temelji prihodnjim raziskavam (Levy in J. Ellis 2006, 181). Industrija video iger ponuja veliko literature z usmeritvami razvoja video iger vendar je na to temo malo akademske. Razvoj iger je inherentno agilna aktivnost in veliko organizacij je podedovalo agilne prakse (Ruonala in YLIOPISTO 2016, 1) za katere tudi obstaja veliko literature, ki pa ni usmerjena v razvoj iger (Barbosa 2017, 292). S pregledom literature smo disciplini razvoja video iger zagotovili temelje nadaljnjim raziskavam na področju raziskovanja razvojnih procesov.

2.1 NAMEN RAZISKOVANJA

Raziskava se nanaša na raziskovanje procesov, ki se uporabljajo pri razvoju video iger. Namen je odkriti vse procesne modele in njihove značilnosti, kar vodi v identifikacijo aktivnosti, ki so najbolj pogoste v procesu razvoja. Cilj naloge je sestaviti procesni model, ki bi bil primeren za uporabo in optimizacijo z zrelostnim modelom za igre.

Na podlagi tega smo formulirali naslednja raziskovalna vprašanja:

**RV1:** Katere procesne modele, se uporablja pri razvoju video iger?

**RV2:** Katere so osnovne aktivnosti v procesu, ki bi lahko sestavljale korake procesa?

**RV3:** Ali lahko sestavimo ustrezen procesni model, ki bi bil primeren za optimizacijo z zrelostnim modelom za igre?

2.2 ZBIRANJE PODATKOV

Kvaliteta literature vpliva na celostno napredovanje znanja tematike (Levy in J. Ellis 2006, 183). Za zagotavljanje kvalitete iskanja literature smo uporabljali sistematične tehnike iskanja. Proces zbiranja podatkov smo razdelili na digitalne in analogne tehnike.

**Digitalne tehnike** vključujejo zbiranje literature iz najbolj stabilnih in priznanih podatkovnih skladišč, knjižnic in izjemoma spletnih strani, katerih moderatorji so večje organizacije. Za iskanje literature smo izvajali poizvedbe na spletnih straneh in iskalnikih:

* google.si,
* books.google.com,
* amazon.com,
* link.springer.com,
* tandfonline.com,
* sciencedirect.com,
* onlinelibrary.wiley.com,
* uk.sagepub.com,
* Web of Knowledge,
* ACM Digital Library,
* IEEE Xplore.

Večina podatkovnih skladišč ponuja lastne grafične vmesnike s funkcijami omejevanja (Booth, Papaioannou, in Sutton 2012, 77), ki omogočajo modifikacijo poizvedb. V kolikor je bilo to mogoče smo se posluževali teh. V veliko primerih smo z namenom strnitve zadetkov bili primorani uporabiti napredne tehnike.

Elektronske baze tipično dovoljujejo iskanje s prosim tekstom. Pristop, ki se velikokrat izkaže za pomanjkljivega smo podprli z uporabo krajšav ('\*','$', '%') in nadomestnih ('?', '!') simbolov. Poleg iskanja s prostim tekstom smo uporabili iskanje s tezavri[[6]](#footnote-6), ki služijo podatkovni bazam za naslavljanje predmetov ali opisov (Booth, Papaioannou, in Sutton 2012, 74). Naprednejše tehnike je vključevala tudi uporaba logičnih operatorjev (AND, OR, NOT):

* OR združuje termine z istim konceptom in s tem razširi iskanje,
* AND združuje termine z različnimi koncepti in s tem strni iskanje,
* NOT izključuje nepomembne termine in strni iskanje (Booth, Papaioannou, in Sutton 2012, 76).

**Analogne tehnike** so vključevale iskanje avtorjev in metodo snežene kepe. Z iskanjem po bibliografijah smo spoznavali avtorje in konstrukte, ki so najbolj povezani s tematiko raziskave. Sledili smo verigam citatov in odkrivali zgradbo literatur. To nam je pomagalo pri spoznavanju vplivnih literatur, avtorjev in spoznanju terminologije tematike.

Kot vplivne literature smo identificirali dela:

1. Ruonala, Henna-Riikka. 2016. „Agile Game Development: A Systematic Literature Review“.
2. Aleem, Saiqa, Luiz Fernando Capretz, in Faheem Ahmed. 2016. „A Digital Game Maturity Model (DGMM)“. *Entertainment Computing* 17 (Supplement C): 55–73. doi:10.1016/j.entcom.2016.08.004.

Prvo delo je sistematični pregled literature, ki vsebuje analize člankov in njihove povezave. Slednje delo je prvi primer zrelostnega pristopa za igre katerega smo uporabili tudi v našem, delu. Iz terminologije smo razbrali gesla, ki so nam pomagala pri kreiranju poizvedb. Gesla sledijo po pomembnosti:

* Game,
* Process,
* life-cycle,
* method,
* engineering,
* development.

Iz teh smo sestavljali kombinacije, ki veljajo za indekse. Sledita primera:

* Game Development Process OR Life-Cycle.
* Game Process OR Method OR Life-Cycle.

2.3 POTEK DELA

Uvodoma smo predstavili potencial, ki ga predstavljajo video igre. Kot je razvidno iz literature je bilo malo pozornosti posvečene razvoju procesov za video igre. V tretjem poglavju bomo poskusili predstaviti vse procese, ki jih lahko najdemo v literaturi. To znanje bo omogočalo lažje razumevanje delovanja procesov in spoznanje terminologije. Nato bomo v četrtem poglavju zbrali vse pretekle poskuse izgradnje procesov in jih pokomentirali na podlagi problematike, ki se pojavlja v razvojih video iger. V tem poglavju bomo spoznali vse aktivnosti in naravnanost discipline razvoja video iger, ki bo pripomogla k usmeritvam pri izgradnji procesnega modela.

Izgradnja modela bo potekala sistematično in si sledi po poglavjih:

* Pog. 5: spoznanje discipline, ki gradi metode za izgradnjo procesnih modelov,
* Pog. 6: izgradnja metode za definiranje procesnega modela,
* Pog 7: apliciranje metode in izgradnja procesnega modela,
* Pog. 8: spoznanje pristopov optimizacije in ilustracija optimiziranje.

Raziskava je bila pisana v urejevalniku besedil Word. Zbrano literaturo smo upravljali z odprtokodno programsko opremo Zootero. Vsi procesni modeli so bili narisani z odprtokodno rešitvijo ArgoUML. Za znižanje tveganj smo uporabljali spletno rešitev spremljanja izvorne kode GitHub (https://github.com/andrazg/game\_development\_process), ki smo o uporabili za shranjevanje dokumentacije raziskave.

2.4 OMEJITVE

Upoštevati moramo, da smo pri iskanju literature v večini uporabljali digitalne tehnike. Ker je raziskovalno področje razvojnih procesov slabo raziskano je velika možnost nekonsistentne terminologije ali nepravilne rabe. V literaturah se prepletajo trije termini, ki definirajo procese. Čeprav vsak termin posamezno definira drugačen obseg lastnosti in namenov (pog. 5) so jih avtorji enačili. Ti termini si sledijo po pojavnostih: proces (process), življenjski cikel (life-cycle) in metoda (method). Poleg tega smo opazili nekonsistentnost pri poimenovanju procesnih modelov. Navedeno je nekaj ekvivalenčnih izrazov procesnih modelov (življenjskih ciklov, metod):

1. Code and fix (Tsui, Karam, in Bernal 2016, 58) **~** Build and fix (Sabharwal 2009, 8).
2. Stage gate (Peters 2008, 113) **~** Phased-release (Lethbridge in Laganiere 2005, 430).
3. Incremental Software Development Life Cycle (Sabharwal 2009, 16) **~** Incremental delivery (Sommerville 2010, 47) **~** Incremental Process (Pressman in Maxim 2014, 43)

Nevarnost rezultatov poizvedb predstavljajo tudi ključne besede, katerih pomen ne opisuje vsebine literature. Tukaj smo si pomagali s sopomenkami (inženiring, razvoj). Za prevajanje strokovnih izrazov v slovenščino smo uporabljali Računalniški slovarček[[7]](#footnote-7). Besed, ki jih nismo našli smo prevedli sami in podprli z opombo v izvorni obliki.

3 PROCESNI MODELI

Programski inženiring je uporaba discipliniranega, merljivega in sistematičnega pristopa v razvoj, zagon in vzdrževanje programske opreme. Temelj programskega inženiringa predstavlja proces (Pressman in Maxim 2014, 15). V informacijskih sistemih je definiran kot pot, ki jo moramo upoštevati, da pridemo do produkta (Rolland 1998, 3). Proces je zbirka aktivnosti, del in nalog, ki se izvajajo ob ustvarjanju produkta. (Pressman in Maxim 2014, 16). Procesi so kompleksne aktivnosti, ki lahko vsebujejo tudi sebi podrejene procese. (Sommerville 2010, 28). Vse te aktivnosti bivajo v okvirjih ali modelih, ki definirajo njihove povezave s procesom in drugimi aktivnostmi (Pressman in Maxim 2014, 31). Procesi so se razvili, da bi izkoristili sposobnosti ljudi v organizaciji in karakteristike sistema katerega razvijajo. (Sommerville 2010, 28). Procesi iste narave so razdeljeni v procesne modele (Rolland 1998, 8). Procesni model je poenostavljena predstavitev procesa razvoja. Vsak model predstavlja proces z določene perspektive in tako podaja le delček informacije o dejanskem procesu (Sommerville 2010, 29). Procesni model je bolj ali manj grobo pričakovanje poteka procesa, ki ga dejansko vidimo šele ob uporabi v praksi (Rolland 1998, 8). Pri uporabi se modeli med seboj ne izključujejo. V velikih sistemih se jih pogosto uporablja več skupaj (Sommerville 2010, 30). Procesni model je primarni pristop, ki projekt organizira v aktivnosti (Lethbridge in Laganiere 2005, 428).

3.1 AD HOC PROCESNI MODELI

Ti pristopi so povsem odvisni do znanj in izkušenj razvijalca ali razvojne ekipe. Inštitut programskega inženiringa je tak način razvoja označil za nepredvidljivega. Vsakršna izguba člana razvoja ima negativen učinek na proces razvoja (Muffatto 2006, 73).

3.1.1 SLAM DUNK MODEL

Slam dunk[[8]](#footnote-8) je model, ki je povsem enostaven predhodnik vsem procesnim modelom. Pri razvoju se vsi vpleteni zavedajo kaj je potrebno narediti. Posledično ne razvijejo nobenega procesa in ne izdelajo načrta. Tak pristop pogosto ne privede do izdaje programske opreme. Čeprav v določenih primerih na začetku kažejo veliko potenciala, so brez načrtov in definiranih ciljev, ki bi lahko spremljali napredek razvoja (Peters 2008, 109). Podoben pristop je Big Bang, ki celoten produkt dostavi v enem dahu (Jalote in dr. 2004, 2).

3.1.2 PRILOŽNOSTNI MODEL

Organizacije se velikokrat ne poslužujejo dobrim praksam razvoja programske opreme. Zaradi česar končajo pri uporabi modela, ki ga imenujemo priložnostni model (Lethbridge in Laganiere 2005, 428). V teoriji se uporablja tudi izraz ad hoc razvoj (Muffatto 2006, 73). Pri tem pristopu, ki je viden na sliki 3.1, razvijalci konstantno modificirajo svoj produkt dokler ta ne ustreza njim ali naročniku (Lethbridge in Laganiere 2005, 428). Ker se kodiranje smatra za osrednjo aktivnost tega procesa ga v praksi imenujejo tudi (code-and-fix) kodiraj in popravi (Tsui, Karam, in Bernal 2016, 58) ali (build and fix) gradi in popravi (Sabharwal 2009, 8).

Slika 3.1 priložnostni model  
vir: (Lethbridge in Laganiere 2005, 428)

Slabost tega pristopa je, da pred začetkom implementacije ne predvideva pomembnosti zahtev in načrtovanja. Načrtovanje v tem modelu je ad hoc aktivnost. Ker ne vsebuje podrobnih načrtov ni jasnih ciljev, posledično ni jasne slike ali gre razvoj v pravo smer ali ne. Pristop eksplicitno ne predvideva sistematičnega testiranja ali drugih načinov zagotavljanja kakovosti kar pomeni, da lahko končen produkt vsebuje pomanjkljivosti. (Lethbridge in Laganiere 2005, 428).

3.2 SEKVENČNI PROCESNI MODELI

Procesni modeli iz te družine stremijo k vzpostavitvi strukture in ureditve razvoja programske opreme. Aktivnosti in naloge se izvajajo sekvenčno z definiranimi časovnimi mejniki. Spadajo tudi med predpisujoče, ker predpisujejo niz procesnih elementov; okvirnih aktivnosti, dejanj programskega inženiringa, nalog, zagotavljanja kvalitete in mehanizme sprememb. Vsak procesni model predpisuje določen delovni tok[[9]](#footnote-9) , ki odraža medsebojno povezanost procesnih elementov (Pressman in Maxim 2014, 41).

3.2.1 KASKADNI MODEL

Model imenovan tudi klasični življenjski cikel, narekuje sekvenčni, sistematski pristop k razvoju programske opreme. Začne se s specifikacijo potreb naročnika, nato se nadaljuje v načrtovanje, modeliranje, konstrukcijo in zagon, ki kulminira v nadaljnji podpori končanega projekta (sl. 3.2) (Pressman in Maxim 2014, 42).

Slika 3.2. kaskadni model

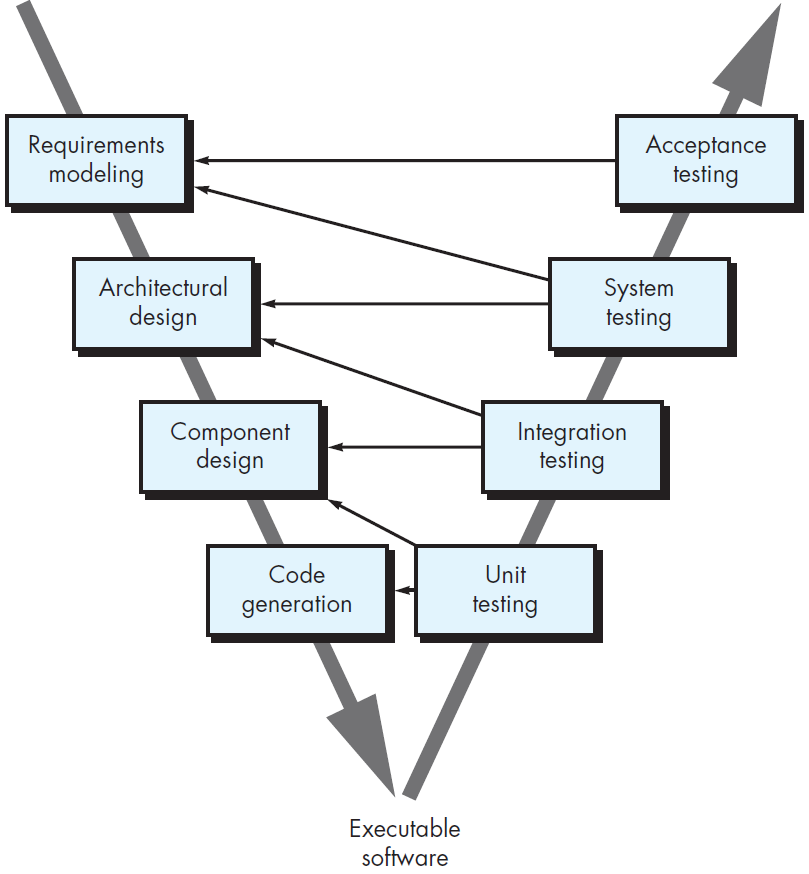
  
vir: (Pressman in Maxim 2014, 42)

Model predstavlja klasični pogled na inženiring programske opreme z upoštevanjem pomembnosti potreb, načrtovanja in zagotavljanja kakovosti. (Lethbridge in Laganiere 2005, 428). Model je primer načrtno-usmerjenih procesov, ker je v praksi vsako aktivnost potrebno načrtovati. (Sommerville 2010, 31). Sistemske zahteve in cilji so vzpostavljeni preko komunikacije z uporabniki ali naročniki. Ti oblikujejo sistemsko specifikacijo. Proces načrtovanja vzpostavi sistemsko arhitekturo na podlagi zahtev po programski ali strojni opremi. Implementacija in testiranje enot služi za verifikacijo delovanja posameznih fragmentov sistema na podlagi sistemske specifikacije. Naslednji korak prestavlja testiranje in verifikacijo celotnega sistema ter kasnejšo implementacijo ob ustreznosti delovanja (Sommerville 2010, 31).

Načeloma je rezultat vsakega koraka potrditev enega ali več dokumentov. (Sommerville 2010, 32). Validacija in zagotavljanje kakovosti pri vsakem koraku omogoča, da se naslednji korak prične na trdnih temeljih (Lethbridge in Laganiere 2005, 429). Čeprav se praviloma naslednji korak ne more pričeti dokler prejšnji ni zaključen se koraki v praksi deloma prekrivajo in prenašajo informacije eden do drugega. Dokumenti, ki so že bili potrjeni se morajo spremeniti, da ustrezajo spremembam v korakih. Zaradi ponovnega pregledovanja dokumentacije lahko te iteracije predstavljajo dodatne stroške in občutno več dela. Težave so velikokrat zamrznjene, prestavljene za kasnejšo obravnavo, prezrte ali spregledane. Za kakršenkoli poseg v sistem, ki je že v delovanju, bo razvoj najverjetneje potreboval ponoviti pretekle korake procesa (Sommerville 2010, 32). Naročnik je navadno prisoten pri podajanju zahtev, potem večinoma ponikne v fazi analiz, načrtovanja ali kodiranja. Pojavi se zopet pri testiranju in predaji produkta (Palmquist in dr. 2013, 5). Kasnejšim zahtevam naročnika je zato težko ugoditi (Sommerville 2010, 32). Ta okorelost je še posebej moteča ob dejstvu, da se zahteve naročnikov konstantno spreminjajo. Ob zaključku procesa model predvideva le vzdrževanje. Nadaljnji razvoj se razume kot manjši poseg, kjer ni potreb po ponovnem načrtovanju. Na žalost se velikokrat izkaže da temu ni tako, saj se sistem konstantno spreminja (Lethbridge in Laganiere 2005, 429).

3.2.2 V-MODEL

Variacija kaskadnega modela se imenuje V-model (Pressman in Maxim 2014, 43). Imenovan tudi kot variacijski in validacijski model (Tutorialspoint 2017). Verifikacija in validacija (V&V) sta namenjeni za potrditev ustreznosti programske opreme na podlagi specifikacije in pričakovanj naročnika. (Sommerville 2010, 41). Ta model prikazuje odnose med aktivnostmi zagotavljanja kvalitete in komunikacije (sl. 3.3). S pomikanjem razvijalcev po levi strani V-modela se osnovni problem preoblikuje v vedno bolj podrobno in tehnično reprezentativno rešitev. Ko je del kode generiran se ekipa pomika po drugi strani V-modela navzgor in opravi serijo preizkusov. V osnovi ni razlike med kaskadnim in v-modelom. Slednji le pripomore k vizualizaciji poteka verifikacije in validacije posameznih korakov v kaskadnem modelu (Pressman in Maxim 2014, 43). Model predvideva, da se zahteve, njihove prioritete in red ne bodo spreminjale, kar je moteče če upoštevamo, da vsak korak skriva potencialne napake. Prvi testi so narejeni daleč v razvoju, kar je pozno in lahko predstavlja stroške. (Palmquist in dr. 2013, 9).

Slika 3.3 v-model  
  
vir: (Pressman in Maxim 2014, 43)

3.2.3 FAZNI MODEL

Fazni model (phase-release model) izvira iz proizvodnega sektorja. Ima veliko poimenovanj eden izmed njih je tudi stopenjski (stage gate) (Peters 2008, 115). V Cambridgskem slovarju najdemo razlago na stage-gate; opisuje točko v razvoju na kateri je mogoče preučiti napredek in narediti katerekoli spremembe, ki vplivajo na stroške, vire ali zaslužek („Cambridge English Dictionary: Meanings & Definitions“ 2017). Model izboljša nekaj težav kaskadnega modela. Najpomembnejša sprememba je, da predstavlja koncept inkrementalnega razvoja. Model predlaga delitev projekta na posamezne podprojekte ali faze takoj po definiranju zahtev in načrtovanju. Vsako fazo se ob končanju sprosti stranki. Tako so lahko nekateri deli sistema vidni prej kot bi bili z uporabo kaskadnega modela (Lethbridge in Laganiere 2005, 430). Model še vedno vsebuje glavne težave kaskadnega modela. Ob začetku razvoja morajo biti definirane vse zahteve. Ko razvoj pride do druge faze (Slika 3.4) je načrtovanje odvisno od originalne specifikacije (Lethbridge in Laganiere 2005, 430).

Slika 3.4 fazni model



Vir: (Lethbridge in Laganiere 2005, 430)

3.2.5 INKREMENTALNI MODEL

V modelu se združujejo elementi linearnega in paralelnega procesnega toka. V časovnem toku se odvijajo linearne sekvence na različnih stopnjah. Vsaka sekvenca predstavlja del sistema programa v razvoju. Posamezno sekvenco imenuje inkrement (Pressman in Maxim 2014, 44).

Sekvenca korakov predstavlja faze procesa. Ta pristop je v osnovi sestavljen iz prepleta specifikacije, razvoja in validacije, ki sestavljajo sosledje različic projekta. Tak pristop je inkrementalen pri čemer vsaka naslednja različica doda funkcionalnost prejšnji. (Sommerville 2010, 30). Iz slike 3.5 je razvidno, da inkrementalni model vsebuje sledi klasične paradigme inženiringa programske opreme, saj posamezni inkrement upošteva sosledje potreb, načrtovanja in zagotavljanja kakovosti. Primarni namen tega modela je zmanjšati tveganje razvoja s fragmentacijo projekta v manjše celote. (CMS 2005, 5).

Slika 3.5 inkrementalni model  
vir: (Pressman in Maxim 2014, 44)

Prvi inkrement je navadno jedro produkta (Pressman in Maxim 2014, 44). Vsebuje najpomembnejše funkcionalnosti (Sommerville 2010, 33) Ta je uporabljen za naročnikovo evaluacijo na podlagi katere se oblikuje načrt za nov inkrement. Proces se ponavlja z izdajanjem vedno novejšega in popolnejšega inkrementa do končnega produkta. Ker se produkt razvija po stopnjah je ceneje in lažje uveljavljati spremembe (Pressman in Maxim 2014, 44). Investitor lahko ob vsaki stopnji pridobi stvarno predstavo statusa projekta skozi razvoj. (CMS 2005, 5).

Tak pristop razvoja je danes najpogostejši. Model velja za temelj agilnim pristopom, kljub temu pa je lahko načrtno-usmerjen, agilen ali agregat obeh. V načrtno-usmerjenih so inkrementi definirani vnaprej. Ob asimilaciji agilne umeritve pa so zgodnji inkrementi izdelani, prihodnji pa so odvisni od prioritet naročnika (Sommerville 2010, 34). Postopne implementacije omogočajo spremljanje vpliva vgrajenih sprememb, zajezitev težav in prilagoditev preden bi lahko te ogrožale delovanje organizacije. Model potrebuje le zmeren poseg vodenja, ki se izvaja na podlagi dokumentacije in formalnih kritik po končanih večjih razvojnih mejnikih (CMS 2005, 5).

Uporaba tega modela je težavna pri velikih, kompleksnih in dolgoročnih projektih kjer delovne skupine posamično razvijajo dele programske opreme. (Sommerville 2010, 34).

3.3 EVOLUCIJSKI PROCESNI MODELI

Med razvojem se zahteve pogosto spreminjajo kar onemogoča linearnost procesov. Tesni razvojni cikli onemogočajo dokončanje zapletenih programskih paketov. Zavoljo poslovnih in konkurenčnih pritiskov pa mora razvoj vseeno izdati omejeno različico paketa. Kadar imamo produkt, ki se konstantno razvija in spreminja, ko so temeljne zahteve programske opreme dobro poznane, razširitve slednjih pa še v načrtovanju so najprimernejši evolucijski procesni modeli. Ti modeli so ponavljajoči se, iterativni. Sem spadajo prototipiranje, spiralni in sočasni procesni model. (Pressman in Maxim 2014, 45).

3.3.1 PROTOTIPIRANJE

Paradigma se začne s komunikacijo in se nadaljuje v slogu, ki ga prikazuje slika 3.6. Načrt in izvedba prve iteracije sta hitra. Investitor po iteraciji prototip pregleda in poda povratno informacijo. Takšen potek omogoča natančnejše definiranje zahtev. Iteracija se zaključi ob rektifikaciji protitipa in izpolnitvi želja investitorjev, sočasno pa pripomore k boljšemu razumevanju prihodnjih. Navadno ta model služi kot idealen mehanizem za prepoznavo programskih zahtev (Pressman in Maxim 2014, 45). Prototip lahko služi kor prva verzija sistema. Interimni so narejeni za čas razvoja in so kasneje zavrženi, medtem, ko se evolucijski iterativno razvijajo v dejanski sistem (Pressman in Maxim 2014, 46). Služi lahko tudi kot orodje za eksperimentiranje predlaganega načrta (Sommerville 2010, 45). Diagrami in slikovni opisi slabo izražajo potrebe uporabniškega vmesnika. Hitro prototipiranje v sodelovanju s končnim uporabnikom je edini razumen pristop k razvoju uporabniškega vmesnika. (Sommerville 2010, 45).

Slika 3.6 prototipiranje



vir: (Pressman in Maxim 2014, 46)

Prikaz delovanja programske opreme je lahko zavajajoče. Kar investitorju predstavlja delujoč program je lahko arbitrarno, v naglici sestavljen sistem, ki ne upošteva celovito kvaliteto in kasnejšega vzdrževanja. Programski inženirji pri implementacijah pogosto sprejemajo kompromise v želji po hitro delujočem prototipu (Pressman in Maxim 2014, 46). Čez čas ti kompromisi, čeprav slabši od idealnih postanejo del sistema. (Pressman in Maxim 2014, 47). Slabost sistema globalno prestavlja dokumentacija. Hitre spremembe med razvojem onemogočajo kreiranje dokumentacije. Specifikacija načrta je izražena samo z izvozno kodo kar ni dobro za dolgoročno vzdrževanje (Sommerville 2010, 46).

Čeprav lahko ta model uporabljamo samostojno je v praksi največkrat uporabljen znotraj konteksta drugih procesnih modelov (Pressman in Maxim 2014, 46). Uporaba tega modela kot samostojnega je lahko kritična in terja velik finančni zalogaj. Za brezhibno delovanje tega modela je potrebna nizka fluktuacija visoko usposobljenega razvojnega kadra pod vodstvom izkušenega projektnega vodje (CMS 2005, 4).

3.3.2 SPIRALNI MODEL

Model združuje iterativno naravo prototipiranja in kontroliran sistematični pogled kaskadnega modela. Omogoča hiter razvoj z izdajanjem vse bolj popolne različice programa. Zgodnje izdaje predstavljajo modele ali prototipe kasnejše iteracije pa vedno volj popolne različice. Spiralni model definira razvojna ekipa z umestitvijo aktivnosti v model. Evolucijski proces se začne v središču (sl. 3.7). Nadaljuje se z aktivnostmi v smeri, ki jih implicira spiralni potek. Sidrne točke so doseženi mejniki in označujejo nov evolucijski prehod (Pressman in Maxim 2014, 48).

Slika 3.7 spiralni model

  
vir: (Pressman in Maxim 2014, 48)

Vsak prehod skozi aktivnost načrtovanja modificira projektni načrt. Na podlagi povratne informacije so posodobljeni časovni načrti, preračunani stroški in število potrebnih iteracij do izida končne različice programa.(Pressman in Maxim 2014, 49). Ta model pojmuje analizo tveganj procesnim modelom. Pred začetkom vsake iteracije je potrebno identificirati pomanjkljivosti, ki potrebujejo obravnavo. V skladu z njimi se oblikuje nadaljnji projektni načrt. (Lethbridge in Laganiere 2005, 431). Medtem ko je mogoče spiralni model aplicirati v celotno življenjsko dobo programa se drugi modeli končajo z njegovo končno izdajo. Model lahko miruje in se po potrebi izboljšav programske opreme inicializira. Spiralni model lahko izpopolnjuje programsko opremo vse do njenega umika. Zato je model primeren za večje sisteme. Z evolucijo procesa se tako razvijalec kot investitor bolje zavedata tveganj, ki jih prinaša vsak evolucijski krog. Spiralni model zmanjšuje tveganje z uporabo prototipiranja, ki ga lahko aplicira v vsakem trenutku (Pressman in Maxim 2014, 49).

3.3.3 VZPOREDNI MODEL

Model omogoča uporabo iterativnih in vzporednih elementov v kombinaciji z vsemi spoznanimi modeli v tem poglavju. Vse razvojne aktivnosti se odvijajo vzporedno, vendar je lahko vsaka aktivnost na svoji stopnji razvoja. (Pressman in Maxim 2014, 50). Model prestavlja abstrakcijo dejanskega procesnega cikla. V njem se definira serijo sekvenčnih dogodkov vseh aktivnosti, dejanj in opravil. (Pressman in Maxim 2014, 50). Namesto, da bi te dogodke omejili v sekvence jih ta model organizira v procesno omrežje. Vsaka aktivnost, dejanje ali opravilo se odvijajo vzporedno z drugo aktivnostjo, dejanjem in opravilom (Pressman in Maxim 2014, 51).

Slika 3.8 vzporedni razvoj

  
Vir: (Pressman in Maxim 2014, 51)

Slika 3.8 prikazuje primer pristopa k vzporednemu modeliranju. Aktivnosti so lahko tudi druga opravila, naloge ali aktivnosti (npr.: komuniciranje ali izgradnja). Aktivnost[[10]](#footnote-10) je lahko na različnih stopnjah razvoja kot prikazuje slika 3.3 zgoraj. Primer; aktivnost komunikacije se je končala in naročnik je predal vse zahteve za izgradnjo programske opreme, sedaj se aktivnost komunikacije prestavi v stanje čakanja sprememb. V tem trenutku se začne nova aktivnost (npr.: kodiranja), ki je predhodno bivala na stopnji neaktivnost. Ta se sedaj prestavi v stopnjo izdelave. V kolikor se pokaže potreba po spremembah zahtev se aktivnost kodiranja prestavi v stanje čakanja sprememb, aktivnost komunikacije pa zopet postane aktivna (Pressman in Maxim 2014, 50).

Slika 3.9 vzporedni model  
  
vir: (Lethbridge in Laganiere 2005, 432)

Po Lethbrige in Laganiere (2005) gre pri tem modelu za princip fragmentacije dela (sl. 3.9). Vsaka razvojna ekipa deluje na svojem fragmentu navadno po spiralnem ali evolucijskem modelu. Slika 3.9 prikazuje eno iteracijo, ki se začne z nujnim načrtovanjem in konča v periodičnih integracijah. (Lethbridge in Laganiere 2005, 432). Njuna razlaga vzporednega modela je povsem identična modelu rapidnega razvoja, ki sledi (pog. 3.4.4. Rapidni razvoj).

3.4 SPECIALIZIRANI PROCESNI MODELI

Ti modeli povzemajo večina karakteristik enega ali več tradicionalnih modelov. Lahko jih definiramo tudi kot skupek posamičnih tehnik ali metodologij za doseganje specifičnih ciljev razvoja, ki kot sami tudi implicirajo proces (Pressman in Maxim 2014, 52).

3.4.1 KOMPONENTNI MODEL

Model nastavlja proces razvoja programske opreme z že prevedeno programsko opremo. Komercialne komponente (COTS[[11]](#footnote-11)) zagotavljajo namenske funkcionalnosti z dobro definiranimi vmesniki, ki omogočajo integracijo v programsko opremo v razvoju. Model vključuje veliko karakteristik spiralnega pristopa. Model je evolucijski, ki zahteva iterativni pristop k razvoju. Modeliranje in konstrukcija se prične z identifikacijo ključnih komponent (sl 3.10). Te so lahko v obliki konvencionalnih programskih modulov, objektno-orientiranih razredih ali njihovih paketov. Model vodi k ponovni rabi[[12]](#footnote-12) že uporabljene programske opreme (Pressman in Maxim 2014, 53).

Slika 3.10 komponenti model

  
vir: (Sommerville 2010, 35)

Tako orientiran pristop zmanjšuje obseg potrebnega razvoja in pripomore k vzdrževanju tveganj in stroškov. Neizogibno je sprejemanje kompromisov kateri lahko vodijo v razvoj sistema, ki ne izpolnjuje pričakovanj naročnika (Sommerville 2010, 36).

3.4.2 FORMALNI MODEL

Formalni model zajema zbirko aktivnosti, ki vodijo v formalno, matematično specifikacijo programske opreme. Ta model omogoča specifikacijo, razvoj in verifikacijo sistemov z apliciranjem strogih matematičnih notacij (Pressman in Maxim 2014, 53). Primer formalnega razvojnega procesa je model čiste sobe[[13]](#footnote-13). Vsak inkrement v razvoju ima formalno specifikacijo na podlagi katere se izvede implementacija. Popravke programske opreme argumentirajo s formalnim pristopom. Model ne predvideva testiranja med procesom. Testiranje je fokusirano v preverjanje zanesljivosti sistema. Cilj modela čiste sobe je izdaja zanesljivejše programske opreme brez pomanjkljivosti. Ta model je navadno uporabljen na projektih kjer je veliko vprašanje tveganja in varnosti. Zaradi svoje kompleksnosti potrebuje specializiran in izkušen kader. Z vidika upravljanja z viri ta proces nima bistvenih prednosti pred drugimi procesnimi modeli (Sommerville 2010, 32).

Formalne metode vsebujejo posebne mehanizme, ki omogočajo eliminacijo problemov kateri so težko rešljivi z uporabo drugih razvojnih paradigem. Dvoumnosti, pomanjkljivosti in nekonsistentnosti je moč lažje odkriti in odpraviti vendar ne z ad-hoc pregledom temveč z aplikacijo matematičnih analiz. Čeprav model ni pogosto v uporabi ponuja razvoj programske opreme brez pomanjkljivosti. (Pressman in Maxim 2014, 54).

Uporaba tega modela je časovno in stroškovno potratna. Malo razvijalcev poseduje ustrezno strokovno znanje za upravljanje tega modela kateri zahteva obsežnejša izobraževanja. Za neizkušene naročnike predstavlja komunikacija skozi ta model težavna. Navkljub težavam pa uporaba modela prestavlja veliko prednost pri razvoju varnostno kritičnih sistemih in projektih z visoko stopnjo tveganja pri razvoju (Pressman in Maxim 2014, 54).

3.4.3 ASPEKTNO USMERJEN RAZVOJ

Inženirji kompleksnih rešitev vedno implementirajo določene lokalizirane funkcionalnosti ne glede na to kateri model izberejo. Lokalizirane karakteristike programske opreme se modelirajo z objektno usmerjeno paradigmo kot komponente, katere so implementirane v kontekst sistemske arhitekture. Z rastjo kompleksnosti in sofisticiranosti sistema določene uporabniške nastavitve ali področja zanimanja razširjajo celotno arhitekturo. Lahko gre za varnostne nastavitve, pravila delovanja, sinhronizacije opravil ali upravljanje pomnilnika. Ko se te pojavljajo v več sistemskih funkcijah ali informacijah se za njih uporablja termin presečne zadeve. Ta model še ni dozorel. Ob uporabi največkrat povzame karakteristike evolucijskih ali vzporednih procesnih modelov (Pressman in Maxim 2014, 55).

3.4.4 RAD

Dobra lastnost rapidnega razvoja je povečana komunikacija med razvojem in naročnikom v vseh korakih razvoja s pomočjo zmogljivih razvojnih orodij (Sabharwal 2009, 19). Orodja lahko vključujejo gradnike grafičnih vmesnikov, računalniško podprto programsko inženirstvo[[14]](#footnote-14), sisteme za upravljanje podatkovnih baz, programske jezike četrte generacije[[15]](#footnote-15), generatorje kode ali objektno umerjeno paradigmo razvoja (CMS 2005, 8). Model predvideva fragmentacijo programskih zahtev v posamezne module. Ti se nato razvijajo in neodvisno integrirajo v produkt. Najpomembnejši atribut modela je hitrost poteka razvoja od analize zahtev do končnega sistema. Čas izdaje enega modula navadno traja 60 do 90 dni in ga imenujemo časovni okvir (Sabharwal 2009, 19). Modularna fragmentacija omogoča uporabo komponent na podlagi katerih se skrajša časovni okvir razvojnega cikla in stroške razvoja (Sabharwal 2009, 20).

Slika 3.11 rapidni razvoj



Vir: (Sabharwal 2009, 19)

Model sestoji iz štirih faz (sl. 3.11): definiranja potreb in zahtev s pomočjo evokacijskih tehnik viharjenja možganov, podrobno definiranje potreb in zahtev na podlagi povratne informacije eksploatacije prototipa, rektifikacije prototipa do končnega produkta in preseka testiranja in izobraževanja naročnika (Sabharwal 2009, 19). Pristop upošteva razvoj na način skupnega načrtovanja[[16]](#footnote-16) aplikacije (CMS 2005, 8). Predvideva prisotnost in vpliv povratne informacije na vseh stopnjah razvoja kar pripomore k zadovoljstvu naročnika. (Sabharwal 2009, 20). Model je primeren za uporabo pri modularnih posodobitvah že obstoječih sistemov, krajših razvojnih ciklih in dobro definiranih zahtevah (Sabharwal 2009, 21).

3.4.5 UNIFICIRAN MODEL

V avtorji[[17]](#footnote-17) modela navajajo potrebo po iterativnem in inkrementalnem procesnem ciklu, ki temelji na uporabnosti in arhitekturi. Unificiran proces je poskus združitve najboljših značilnosti procesnih modelov z umestitvijo njihovih najboljših praks v agilne procese. Proces upošteva pomembnost komunikacije z naročnikom in poenostavlja naročnikov pogled na sistem na podlagi spisanega uporabniškega priročnika. Tega spiše naročnik in služi kot podlaga za oblikovanje celovitega modela analize. (Pressman in Maxim 2014, 56).

Procesni model je primer modernega procesnega modela, ki je bil izpeljan iz unificiranega jezika modeliranja[[18]](#footnote-18) in povezanega unificiranega razvojnega procesa programske opreme (Sommerville 2010, 50).

Slika 3.12 unificiran procesni model  


Vir: (Pressman in Maxim 2014, 57)

Unificiran model ima navadno pet faz (sl. 3.12). Začetna faza zajema komunikacijo in načrtovanje skupaj z naročnikom. V tej fazi se definirajo zahteve, postavi se osnova arhitekture sistema in izdela načrt iteracij. Temeljne zahteve so razložene v uporabniškem pripročniku, ki opisuje lastnosti in funkcije uporabe sistema. Faza elaboracije vključuje komunikacijo in modeliranje aktivnosti generičnih procesnih modelov (Pressman in Maxim 2014, 57). Cilj te faze je razviti razumevanje problema, definirati arhitekturo sistema, izdelati načrt projekta in identificirati ključne faktorje tveganja. Faza konstrukcije zajema načrtovanje sistema, programiranje in testiranje. Nekateri deli sistema se razvijajo in implementirajo vzporedno. Ob koncu te faze model pričakuje delujoč sistem s priloženo specifikacijo, ki je pripravljen za izdajo uporabniku. Faza prehoda je namenjena za prenos sistema iz razvojnega okolja v uporabniško okolje (Sommerville 2010, 51). Produkcijska faza prestavlja zadnjo fazo procesa. V tej fazi se spremlja delovanje sistema, omogoča se sistemsko podporo za samo delovanje, identifikacijo napak in njihovo reševanje. (Pressman in Maxim 2014, 58).

Iteracija znotraj modela je podprta v obe smeri. Vsaka faza se lahko izvede iterativno z inkrementiranimi rezultati. Lahko pa se vse faze izvedejo kot en inkrement. Statičen vidik modela se osredotoča na aktivnosti procesa razvoja. Te imenuje potek dela. V kontekstu prepoznamo šest osnovnih delovnih tokov in tri podporne delovne tokove, ki jih razberemo iz slike 3.13 (Sommerville 2010, 51).

Slika 3.13 temeljni delovni tokovi in faze unificiranega modela po IBM

  
vir: („IBM developerWorks : rational“ 2017, 10)

Temeljni delovni tokovi so: tok poslovnega modeliranje, potrebe, analiza in oblikovanje, implementacija, testiranje, namestitev. Podporne delovne tokove pa sestavljajo; projektno upravljanje, konfiguracija in spremembe upravljanja in razvojna okolja („IBM developerWorks : rational“ 2017, 10).

Model ni primeren za vse tipe razvoja (Sommerville 2010, 53). Naloge, ki jih ta procesni model navaja ne veljajo za vsak projekt. Razvojna ekipa prilagodi proces, da ustreza njihovim potrebam (Pressman in Maxim 2014, 58). Pomemben napredek tega modela je ločitev projektnih faz in delovnih tokov. Faze so dinamične in definirane s cilji. Delovni tokovi so statični in predstavljajo tehnične aktivnosti, ki niso povezane s fazami vendar jih uporabljamo za doseganje ciljev posamezne faze (Sommerville 2010, 53).

3.4.6 OSEBNI MODEL

Vsak razvijalec uporablja nek proces za izgradnjo programske opreme. Proces je lahko nenavaden, ad hoc, lahko se spreminja dnevno, lahko je efektiven ali celo neefektiven. Vendar proces obstaja. Osebni programski proces[[19]](#footnote-19) poudarja tako produkt v delu kot njegovo končno kvaliteto. Razvijalec je odgovoren za načrtovanje projekta in nadzor kvalitete produkta. Osebni model združuje pet okvirnih aktivnosti; načrtovanje, abstraktno oblikovanje, pregled oblikovanja, razvoj in obdukcijsko poročilo[[20]](#footnote-20) (Pressman in Maxim 2014, 59) Ravno slednji je posebnost osebnega modela. Povratna informacija o poteku procesa se lahko pridobi iz pregleda poteka projekta, ki je zabeležen v obdukcijskem poročilu(Pressman in Maxim 2014, 267). Industrija programske opreme zagotavlja, da je izdelava teh poročil dobra praksa po vsakem končanem projektu (Collier, DeMarco, in Fearey 1996, 65). S temi poročili lahko statistično napovemo kvaliteto nadzora nad človeškimi napakami kot so podcenjevanje, napihnjenost in neusklajenost zahtev (Collier, DeMarco, in Fearey 1996, 66).

Ta model ni pogosto v uporabi. Model je intelektualno zahteven in težko sprejemljiv s strani vodstva (Pressman in Maxim 2014, 60). Pomanjkanje analitičnega kadra, nestrinjanje kriterijev evaluacije in pritiski na delu so največkrat razlogi, da se obdukcijska analiza ne izvaja (Collier, DeMarco, in Fearey 1996, 66). Posledično pa se tudi ne poslužujejo tega modela. Usposabljanje za uporabo tega modela je dolgotrajno in drago (Pressman in Maxim 2014, 60).

3.4.7 EKIPNI MODEL

Ker je razvoj v organizacijah velikokrat situiran okoli ekip je Watts Humphrey razširil spoznanja iz vpeljave osebnega modela in predstavil ekipni[[21]](#footnote-21) procesni model. Cilj modela je sestava avtonomnih razvojnih ekip za proizvajanje visokokvalitetnih programskih rešitev. Naloge modela so; izgradnja ekipe, ki je zmožna samostojno vzpostaviti cilje projekta njegove procese in načrt, prikazati način kako motivirati ekipo za vzdrževanje učinkovitosti, pospešiti izboljšave procesov razvoja, zagotoviti usmeritve za implementacijo izboljšav in omogočiti izobraževanje. (Pressman in Maxim 2014, 61).

3.4.8 SINHRONIZACIJSKO STABILIZACIJSKI MODEL

Pristop je podoben inkrementalnemu. Začne se z zahtevami. Nato razvojna ekipa izdela specifikacijo, določi prioritete in razdeli razvoj na štiri večje izdaje. Izdaje[[22]](#footnote-22) so kandidat za javnost (RC) 1 do 3 in izdaja v proizvodnjo. Kandidat za javnost je prvi večji izid in predstavlja mejnik v razvojnem ciklu z možnostjo širše modifikacije kode in vsebine (sl. 3.14) (Peters 2008, 120).

Slika 3.14 odstotek dovoljenih sprememb glede na izdajo

  
Vir: (Peters 2008, 120)

S časom vsak naslednji izid postane izhodišče. Odstotek dovoljenih sprememb v kodi, število novih funkcionalnosti in napake se zmanjšujejo. Največji ali celo najmanjši defekti so z izidom RC3 odpravljeni. Med izdajami se lahko aplicira tudi agilni ali katerikoli drugi pristop. Cilj tega pristopa je doseganje stabilnosti in zrelosti izdanega produkta (Peters 2008, 121).

3.5 AGILNI PROCESNI MODELI

Agilni procesi so družina razvojnih metodologij, ki proizvajajo programsko opremo s kratkimi iteracijami in dovoljujejo večje spremembe v načrtovanju (Tsui, Karam, in Bernal 2016, 84). Agilno programiranje je pravzaprav nabor najboljših praks zbranih iz drugih življenjskih ciklov in uspešnih praks kodiranja (Peters 2008, 116). Na začetku opomnimo, da niso vse karakteristike agilnega procesa nove ali revolucionarne. Veliko jih izhaja iz leta izkušenj, ki sta jih prinesla uporaba iterativnih in inkrementalnih procesov. Ne obstaja končna definicija kaj sestavlja Agilno metodo, vendar obstaja kar nekaj karakteristik, ki so metodam sorodne (Tsui, Karam, in Bernal 2013, 84). Leta 2001 so Kent Beck in 16 ostalih priznanih razvijalcev, piscev in svetovalcev (t.i. agilno zavezništvo) podpisali manifest agilnega razvoja, ki povzema;

• Individuals and interactions over processes and tools,

• working software over comprehensing documentation,

• customer collaboration over contract negotiation,

• responding to change over following a plan (Pressman in Maxim 2014, 66).

Agilni pristop zajema tri osnove: načine poslovanja z naročnikom, prakse razvoja in prakse zagotavljanja kvalitete (Peters 2008, 118). Konstantna interakcija s klientom je vzpostavljena z integracijo njegovega predstavnika v razvojno ekipo. (Peters 2008, 116). Agilni pristop vpelje koncept konstantnega testiranja, ki je izvedljivo zavoljo izdaje delujočih različic sistema ob končanih intervalih. Posledično lahko naročnik poda oceno funkcionalnosti zelo zgodaj v razvoju.(Peters 2008, 118). Naročnik med razvojnim procesom nenehno podaja povratne informacije o spremembah potreb ali zahtev preko neformalnega komunikacijskega kanala, kar pripomore k zmanjšanju nepotrebne dokumentacije (Sommerville 2010, 58). Agilni pristop eksplicitno zavrača procese katerih vodenje zahteva veliko dokumentacije (Lethbridge in Laganiere 2005, 433). Pristop je primeren za majhne projekte, ki vključujejo negotove, spreminjajoče zahteve in druge dejavnike visokega tveganja. Tveganje je zmanjšano, ker razvoj ne izdaja ničesar velikega. V kolikor iteracija ni zadovoljila zahtev se le to ponovi kar pa ne predstavlja večjih stroškov (Lethbridge in Laganiere 2005, 433).

Čeprav se sistem stalno testira se prav tako dogajajo spremembe, kar je v nasprotju s priporočili Brooksa in drugih[[23]](#footnote-23). Po njihovih priporočilih naj bi se testiranje in spremembe sistema izvajalo ločeno. (Peters 2008, 117). Skrb za stabilnost se rešuje z nenehno izdajo delujočih različic, njihovim testiranjem in apliciranjem spremenjenih standardov kodiranja (Peters 2008, 117).

Čeprav agilne metode slonijo na inkrementalnem razvoju in izdaji, predlagajo drugačne procese za doseganje tega. Ne glede na raznolikost v procesih si delijo načela predstavljena v agilnem manifestu. (Sommerville 2010, 59). Tabela 3.1 spodaj jih podrobno opisuje.

Tabela 3.1 podrobno opisana načela agilnega pristopa  
vir: (Sommerville 2010, 60)

V praksi se tudi pri agilnih metodah srečamo s težavami. Ker se ti pristopi fokusirajo na manjše integrirane ekipe obstajajo težave umestitve pristopov v večje sisteme. Sicer obstajajo primeri integracije agilnih pristopov v sisteme kritičnega inženirstva vendar so bile potrebne modifikacije metod za zagotavljanje jamstva, varnosti in zanesljivosti. Uspeh integracije predstavnika naročnika v razvoj je odvisen od naročnikovih razpoložljivih sredstev ali sposobnosti zastopanja vlagateljev. V projektih z več vlagatelji je nasploh težavno postavljati prioritete, ker navadno vlagatelji nimajo skupnih ciljev (Sommerville 2010, 60).

Najbolj utemeljena kritika pristopov, DeMarco in Boehm navajata tako prednosti kot slabosti agilnih metod. Priporočata uporabo hibridnega pristopa kjer v agilne pristope integriramo tehnike načrtno-usmerjenega razvoja (Sommerville 2010, 62). Veliko podjetij je v praksi trdilo, da so uporabili agilne pristope razvoja medtem, ko so dejansko integrirali agilne metode v svoj načrtno-usmerjen proces razvoja. (Sommerville 2010, 64).

3.5.1 EKSTREMNO PROGRAMIRANJE

Ekstremno programiranje (XP), uporablja paradigmo objektno-orientiranega razvoja in zajema pravila in prakse štirih okvirnih aktivnosti (sl. 3.15): načrtovanja, oblikovanja, kodiranja in testiranja (Pressman in Maxim 2014, 72). Načrtovanje se začne s poslušanjem in zbiranjem aktivnosti katere omogočajo razvijalcem razumevanje poslovnega konteksta in širšo sliko potrebnih funkcionalnosti. Poslušanje vodi v izdelavo zgodb(Pressman in Maxim 2014, 73) ali scenarijev (Sommerville 2010, 65), ki opisujejo potreben output, funkcionalnosti in lastnosti programske opreme (Pressman in Maxim 2014, 73).

Zgodbe se nahajajo v posameznih časovnih periodah, ki jih predstavljajo stolpci intervalov na sliki 3.16. En tak stolpec predstavlja 2-tedenski interval. Sicer lahko uporabljamo daljše vendar v praksi niso priporočljivi. Vsak ta interval predstavlja obseg dela, ki je sestavljen iz zbirke zgodb[[24]](#footnote-24). Vsaka zgodba je napisana na kartico indeksa ali drug medij, ki je dostopen celotni razvojni ekipi. Naročnik vsaki zgodbi pripiše prioriteto na podlagi splošnega števila točk celotnega projekta ali funkcionalnosti (Pressman in Maxim 2014, 73). Ko so zgodbe (scenariji) definirani jih razvojna ekipa razdeli na naloge (Sommerville 2010, 65). Člani ekipe pripiše oceno[[25]](#footnote-25) indeksa posameznim zgodbam, ki je relativna glede na njeno težavnost (Peters 2008, 116).

Slika 3.15 prikaz procesa ekstremnega programiranja



Vir: (Pressman in Maxim 2014, 72)

Slika 3.16 detajlni prikaz zgradbe XP  


Razvojna ekipa določi število zgodb, ki jih lahko izdela v času prvega 2-tedenskega intervala. Po posvetovanju z naročnikom izberejo določeno število zgodb in seštejejo njihove ocene težavnosti. Seštevek ocen težavnosti služi kot referenca za obseg dela enega 2-tedenskega intervala (Peters 2008, 117).

**Seštevek vseh točk projekta = 4000,**

**pričakovano povprečno število točk na 2-tedenski interval = 50,**

**pričakovan čas razvoja (4000/50) = 80 intervalov.**

Na koncu prvega intervala ekipa sešteje število doseženih točk in na podlagi rezultata prilagodi pričakovan čas izdaje programske opreme. Prvi intervali služijo za določitev predpostavljene hitrosti dela. Ta proces se ponavlja po vsakem končanem intervalu. Ostanek točk je preračunan in popravi pričakovano povprečno število točk na 2-tedenski interval (tab. 3.17). Ta tehnika je podobna inženirskemu principu načrtovanja[[26]](#footnote-26) glede na stroške. To pomeni, da lahko proizvedemo toliko, kolikor imamo na voljo časa in denarja. Poleg tega ima naročnik nadzor nad funkcionalnostmi katerim lahko dodeli prioritete glede na tržišče in strategijo organizacije (Peters 2008, 118).

Tabela 3.2 primer preračuna povprečnega števila točk na 2-tedenski interval



Vir: (Peters 2008, 117)

Oblikovanje sledi principu KIS[[27]](#footnote-27). Priporoča se enostavnejše oblikovanje pred kompleksnimi. Pristop spodbuja uporabo kartic CRC[[28]](#footnote-28) kot uspešen mehanizem za identifikacijo in organizacijo objektno-orientiranih razredov, ki so potrebni sledeči inkrement. Pri zapletenem oblikovanju na podlagi težavnega dela načrta razvojna ekipa ustvari prototip, ki ga imenujejo ostra rešitev[[29]](#footnote-29). Namen te rešitve je zmanjšanje tveganja ob začetku pravega razvoja. Osrednja esenca tega pristopa je oblikovanje, ki se pojavi pred in po začetku kodiranja (Pressman in Maxim 2014, 75).

Osrednji koncept aktivnosti kodiranja zajema skupinsko kodiranje (Pressman in Maxim 2014, 75). Pri agilnem tipu programiranja razvojni inženirji delajo v parih. En par dela na eni delovni postaji in si deli tipkovnico. Par navadno sestavljata izkušenejši in mladi razvijalec (Peters 2008, 116). Obstajajo določena mnenja, da mlajši zmanjšuje produktivnost z oviranjem izkušenega razvijalca. Čeprav se to zdi verjetno, se produktivnost celotne razvojne ekipe v razumno daljših projektih (6 mesecev) poveča (Peters 2008, 117). V praksi imata različne naloge. Medtem ko en skrbi za standard kodiranja se drugi osredotoča na detajle kodiranja določenega dela zgodbe. To zagotavlja reševanje problemov in zagotavljanje kvalitete v realnem času (Pressman in Maxim 2014, 75).

Enote za testiranje so ustvarjene s pomočjo posebnih orodij, ki omogočajo, da se jih avtomatizira. To omogoča strategijo regresivnega testiranja. Enote testiranj so organizirane v univerzalne testne pakete, ki dnevno integrirajo in validirajo delovanje sistema (Pressman in Maxim 2014, 75). Med izdajami različic je kratek časovni razkorak (Sommerville 2010, 65). Nove različice so lahko izdane večkrat dnevno. Za vsako novo različico je potrebno pognati definirane testne pakete. Nova verzija je sprejeta samo če se na njej uspešno izvedejo vsi testni scenariji (Sommerville 2010, 67).

Industrijsko ekstremno programiranje (IXP) je organska evolucija XP. Vključuje minimalistično, naročniku in testiranju usmerjeno prepričanje. IXP se od XP razlikuje po večjem vključevanju managementa, razširjeni vlogi naročnika in posodobljenim tehničnim praksam (Pressman in Maxim 2014, 75). IXP vključuje šest novih praks, ki omogočajo delovanje XP modela v večjih organizacijah: ocena pripravljenosti, usposobljenost skupnosti, uporabnost projekta, usmerjenost k testiranju, retrospektiva in konstantno učenje. Ocena pripravljenosti ugotavlja ali vsi vpleteni razumejo problem projekta. Usposobljenost skupnosti preverja ali so na naloge postavljeni ustrezno usposobljeni posamezniki. Ekipa IXP določi zbirko časovnih mejnikov, ki ocenjujejo potek dela. Na podlagi mejnikov pa ustvarijo mehanizme, ki bodo sporočali ali so bili doseženi. Med prakticiranjem retrospektive ekipa izvede specializiran tehnični pregled različice programa. Pregled preuči aktivnosti, napake in pridobljeno znanje. Konstantno učenje je zadnja izmed razširjenih praks, ki ga uvede IXP. Ekipa spodbuja učenje novih metod in tehnik, ki vodijo v boljšo kvaliteto produkta (Pressman in Maxim 2014, 76).

3.5.2 SCRUM

Scrum[[30]](#footnote-30) model je agilna metoda, ki se fokusira na upravljanje inkrementalnega razvoja. Čeprav model ne definira tehničnih aspektov agilnih pristopov je zato bolj primeren za uporabo ob bolj tehničnih pristopih kot je model XP (Sommerville 2010, 72). Scrum zajema tri faze (sl. 3.17). Prva je faza načrtovanja. Tukaj se definirajo osnovni cilji in arhitektura projekta (Sommerville 2010, 72). Načrtovanju sledijo cikli sprintov, kjer vsak cikel predstavlja en inkrement programa. Proces se konča v zaključku projekta, kjer se dokončajo dokumentacije kot so dokumenti za vzdrževanje programske opreme, navodila za uporabo in dokument obdukcije razvoja. Inovativna lastnost modela so sprint cikli (Sommerville 2010, 73). Sprint je enota načrtovanja, kjer se oceni potrebno delo, izberejo funkcionalnosti za razvoj in opravijo implementacije programske opreme (Sommerville 2010, 73). Sprint je sestavljen iz ocene, izbire, razvoja in pregleda.

Slika 3.17 osnovne faze scrum procesa



Vir: (Sommerville 2010, 73)

Principi modela se skladajo z agilnim manifestom in zajemajo okvirne aktivnosti; potreb, analiz, oblikovanja, evolucije in izdaje (Pressman in Maxim 2014, 79). Znotraj aktivnosti v sprintu se izvajajo opravila. Opravila definirajo in modificirajo člani Scrum ekipe na podlagi problematike (Pressman in Maxim 2014, 79).

Slika 3.18 podroben prikaz Sprinta



Vir: (Pressman in Maxim 2014, 78)

Slika 3.18 prikazuje en potek Sprinta. Ti so fiksnih dolžin, navadno 2-4 tedne. Sovpadajo z izdajanjem različice po XP modelu (Sommerville 2010, 73). Sprinte sestavljajo delovne enote, ki so definirane z zahtevami napisanimi v dnevniku zaostankov (Pressman in Maxim 2014, 79). Dnevnik[[31]](#footnote-31) zaostankov je seznam zahtev ali funkcionalnosti, ki za naročnika pomenijo poslovni kapital (Pressman in Maxim 2014, 79). To je seznam del, ki jih je na projektu potrebno narediti. Faza ocenjevanja v sprintu služi za pregled seznama opravil, posodobitev prioritet in identifikacijo tveganj. Naročnik je v tej fazi tesno povezan z razvojem in lahko predstavi nove zahteve ali naloge na začetku novega sprinta. Med fazo izbire, ekipa razvijalcev skupaj z naročnikom izbere funkcionalnosti, ki se bodo razvijala med naslednji ciklel. Nato se ekipa osredotoči na razvoj (Sommerville 2010, 73). Vsakodnevni, kratki (tipično do 15 min) sestanki (Pressman in Maxim 2014, 79) služijo za pregled napredka dela in možnost spremembe prioritet. Med to fazo je razvojna ekipa izolirana. Komunikacija do naročnika poteka preko Scrum vodje. Njegova naloga je zaščitita razvojne ekipe pred zunanjimi motnjami, (Sommerville 2010, 73) skrbi za posodobitev dnevnika, organizira sestanke, evidentira odločitve in vzdržuje komunikacijo z vodstvom (Sommerville 2010, 74). Model želi spodbuditi člane ekipe k sprejemanju odločitev. Na sestankih je prisotna celotna ekipa. Vsi člani delijo informacije o napredku od prejšnjega sestanka, problemih, ki so nastali in kaj imajo načrtovano za prihajajoči dan. Vsi člani tako vedo kaj se dogaja in lahko načrte prilagodijo glede na problematiko. Nadrejeni ne usmerjajo razvoja, kratkoročno načrtovanje je v domeni vsakega posameznika (Sommerville 2010, 74).

3.5.3 RAZVOJ DINAMIČNIH SISTEMOV

Metoda za razvoj dinamičnih sistemov ali DSDM[[32]](#footnote-32) je metoda za razvoj sistemov z zahtevnimi časovnimi omejitvami s pomočjo inkrementalnega prototipiranja v kontroliranem okolju. Filozofija modela sloni a modificiranem Paretovem principu, da je mogoče narediti 80% aplikacije v 20% časa, ki bi bil potreben za izdajo celotne aplikacije. Cikel se prične v študiji izvedljivosti katera omogoča definiranje zahtev in omejitev. Nato se nadaljuje v poslovno študijo kjer se identificirajo informacijske in funkcionalne zahteve (Pressman in Maxim 2014, 80). Metoda definira tri različne iterativne cikle; funkcionalni cikel, načrtovalno-razvojni cikel in cikel implementacije. Funkcionalni cikel proizvede zbirko inkrementalnih prototipov, ki demonstrirajo delovanje naročniku. Namen je pridobiti dodatne informacije za izboljšanje specifikacije. Načrtovalno-razvojni cikel zagotavlja, da prototipi ustrezajo poslovnim načrtom naročnika. V določenih primerih se ta cikel odvija vzporedno z funkcionalnim. Cikel implementacije postavi zadnji inkrement v delovno okolje. Nato se razvoj nadaljuje z aktivnostmi v funkcionalnem ciklu (Pressman in Maxim 2014, 80).

Z uporabo DSDM z XP pridobimo hibridni pristop, ki definira stabilni procesni model za razvoj programske opreme (Pressman in Maxim 2014, 80).

3.5.4 CRYSTAL

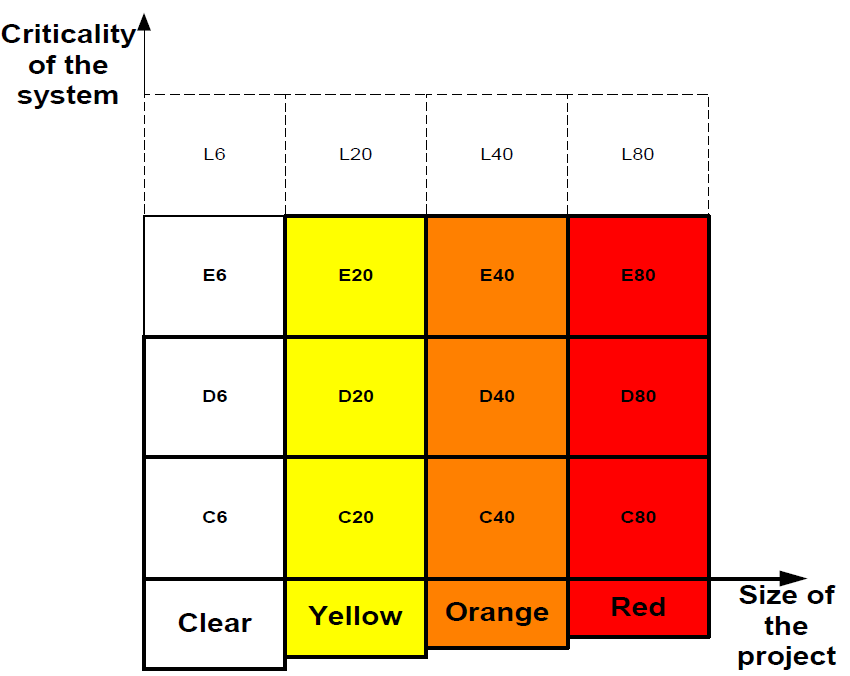
Kristalno družino metod je razvil Alistair Cockburn z namenon prilagoditve metodologij projektom. Cockburn razdeli projekte na faktorje velikosti, tveganj in prioritet. Velikost je sorazmerna glede na število razvijalcev na projektu. Tveganja predstavljajo izgube, ki bi se pojavile ob izpadu delovanja. Prioriteta pa izraža časovni pritisk na projekt (Tsui, Karam, in Bernal 2013, 91).

Faktor prioritete je za potrebe določitve kompleksnosti in kvalitete projekta razdeljen na štiri stopnje kritičnosti;

* Življenja; so težave, ki lahko fizično ali celo fatalno ogrozijo posameznike,
* osnovnega kapitala; so težave izgube kapitala, ki grozijo preživetju organizacije,
* diskrecijskega kapitala; so težave izgube kapitala, ki ne ogrožajo organizacije,
* udobja; so težave manjše izgube kapitala, ki zmanjšajo zadovoljstvo uporabnikov (Tsui, Karam, in Bernal 2013, 91).

Cockburn definira osnovne principe metodologij in njihov način prilagoditve projektom. Primeri metodologij so: Crystal Clear, Crystal Orange, Crystal Orange Web. Metodologije so definirane z barvami. Temnejša kot je barve težavnejša je metodologija (sl. 3.19). Kristalno čista metodologija je primerna za manjše nekritične projekte medtem, ko lahko kristalno oranžno metodologijo uporabimo pri projektih, ki štejejo do 40 ljudi (Tsui, Karam, in Bernal 2013, 91). Kristalna družina metodologij ima generično osnovo. Ta osnova pomaga, da lahko ustvarimo novega člana družine če ga potrebujemo. Pristop predvideva, da se novega člana ustvari pred vsakim projektom (Cockburn 2006, 94).

Sl. 3.19 dimenzije kristalnih metod

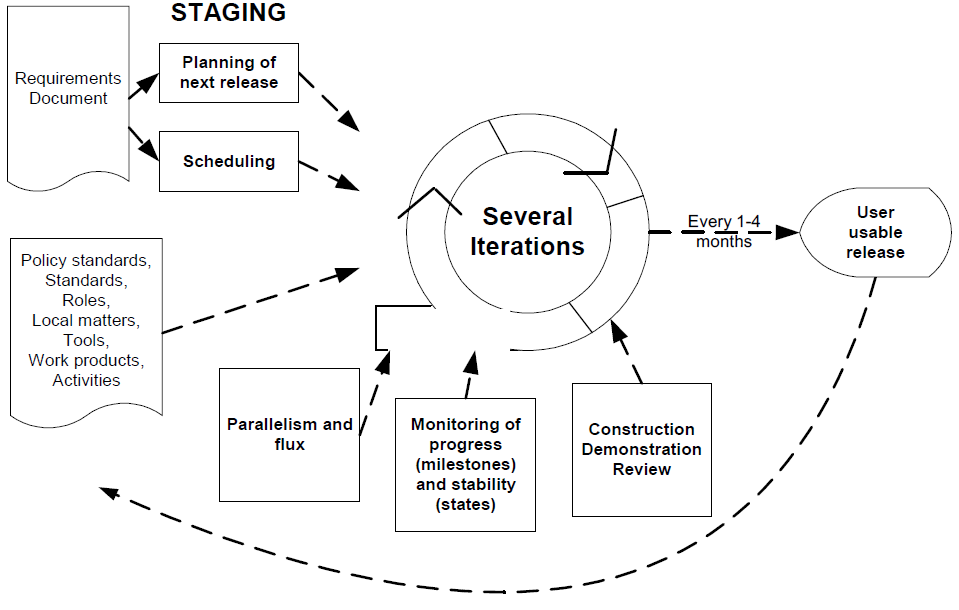


Vir: (Abrahamsson in dr. 2017, 39)

Pri kreiranju nove metodologije moramo upoštevati sledeče principe:

* Uporabiti je potrebno večje metodologije za večje ekipe,
* uporabiti je potrebno težavnejše metodologije za bolj kritične projekte,
* dati je potrebno preferenco lažjim metodologijam, ker težje stanejo več,
* dati je potrebno preferenco osebni komunikaciji pred formalno dokumentacijo,
* razumeti moramo, da zanesljivost ljudi v ekipi skozi čas niha,
* predvidevati je potrebno, da želijo biti posamezniki dobri državljani. Lahko prevzamejo iniciativo (Tsui, Karam, in Bernal 2013, 91).

Slika 3.20 en inkrement kristalno oranžne metode



Vir: (Abrahamsson in dr. 2017, 43)

Poleg tega obstaja še sedem lastnosti, ki razširjajo te principe in podajajo dodatne usmeritve k modeliranju. Te so; pogosto izdajanje različic, refleksivne izboljšave, tesna komunikacija, osebna varnost, fokusiranje, lahek dostop uporabnikov in dobro tehnično okolje (Tsui, Karam, in Bernal 2013, 92). Slika 3.20 prikazuje potek procesa in predstavlja en inkrement kristalno oranžne metode. Priporočeno je večkrat mesečno izdajanje različic testiranih, stabilnih in uporabnih različic. Ne glede na to na kateri stopnji je projekt, je mogoče izvajati izboljšave. Refleksivne izboljšave pomagajo spreminjati in izboljšati proces med samim projektom. Osebna komunikacija pripomore k reševanju težav brez nepotrebnih izgub časa. Priporoča se osmozna komunikacija, ki predstavlja nezavedno sprejemanje informacij. To pomeni, da komunikacija poteka tako, da ostali člani, ki niso neposredno v komunikaciji vsaj slišijo pogovor. Osebna varnost posameznika pomeni, da so vsi posamezniki pripravljeni spregovoriti brez strahu posledic. To prinaša psihološko varnost posameznikov ekipe in pripomore k podajanju iskrenih povratnih informacij. Fokus opozarja na težavo motenj pri koncentraciji. Velikokrat se dogaja, da so izkušenejši člani ekipe bolj obremenjeni zaradi svojega znanja. Drugi želijo izkoristiti njihovo znanje pri čemer se slednji ne morejo osredotočiti na svoje delo. Za potrebe reševanja takšnih težav se priporoča vzpostavitve konice[[33]](#footnote-33) tišine. Lažji dostop uporabnikom pripomore k hitrim povratnim informacijam. Za vzpostavitev dobrega tehničnega okolja je potrebno implementirati avtomatizirano testiranje, frekvenčne integracije in sistema upravljanje konfiguracij (Tsui, Karam, in Bernal 2013, 93).

Ljudje smo boljši pri modificiranju kot ustvarjanju. V ta namen obstajajo ti principi in lastnosti kristalne metodologije. Niso namenjeni, da se jih prevzame in uporablja nedotaknjene. Obstajajo zato, da se jih prevzame, preučuje, odvzema podrobnosti dokler ne ustrezajo potrebam. Modifikacija metodologije je nazadnje le temeljni element kristalnih pristopov (Cockburn 2006, 97).

3.5.5 ODPRTOKODNI MODEL

Model brezplačno distribuira programsko opremo z izvorno kodo. Posamezniki nato prispevajo izboljšavam brez plačila uporabnikov. Ti posamezniki so navadno entuziasti ali uporabniki opreme. Model pričakuje, da bodo člani skupnosti zato imeli razlog za izboljšanje programa. Odprtokodno gibanje je prineslo širok spekter pomembnih sistemov kot so Linux operacijski sistemi in orodja GNU[[34]](#footnote-34). Zagotavljanje kakovosti izvaja skupnost sama (Lethbridge in Laganiere 2005, 434). Odprtokodni model se agilnim metodam zelo približuje. Frekventno se izdajajo manjše različice, na podlagi specifikacij pridobljenih preko e-pošte, oglasnih desk in drugih neformalnih medijev. Integracije se dogajajo pogosto, preko spleta. Razvijalci imajo skupne vizije, saj opremo potrebujejo in jo razvijajo za lastne potrebe. Uspeh te metodologije lahko pripišemo posebno nadarjenim razvijalcem. Veliko jih dela brezplačno in pridobivajo motivacijo z zadovoljstvom uporabnikov (Tsui, Karam, in Bernal 2013, 98).

3.5.6 LEAN

Lean ali vitka metoda je bila ustvarjena za potrebe avtomobilske industrije. Povzema sedem principov; znižanje odpadkov, kvaliteto produkta, ustvarjanje znanja, proizvodnjo v pravem trenutku[[35]](#footnote-35), spoštovanje ljudi in konstantno optimizacijo. Curt Hibbs in kolegi predlagajo drugačni pristop za integracijo vitkega upravljanja v razvoj programske opreme (Janes in Succi 2014, 131). Pristop je usmerjen v kodo in zajema naslednje prakse:

* Upravljanje izvorne kode in avtomatizacija izdajanja,
* avtomatizirano testiranje,
* konstantna integracija,
* manj kode,
* kratke iteracije,
* participacija naročnika (Janes in Succi 2014, 131).

Uporaba upravljanja izvorne kode in avtomatizacija izdajanja skupaj z avtomatskim testiranjem je način avtonomizacije razvoja. Proces se ustavi če izvorna koda ne ustreza pričakovanjem. Poleg tega se preprečuje nalaganje defektne kode v produkcijo. Konstantna integracija je prav tako en del avtonomnosti razvoja. Problemi, ki jih lahko posamezna integracija prinese v sistem so takoj evidentirani in rešeni. Kratke iteracija in prisotnost naročnika pripomoreta k pridobivanju hitrih povratnih informacij in razumevanja dodane vrednosti naročnika (Janes in Succi 2014, 133).

Opaziti je prepričanje, da je pisanje programske opreme podobno izgradnji avtomobila, kar ne upošteva, da je programska oprema nevidna. Potrebne so izčrpne meritve, ki so skladne z organizacijskimi cilji ustanoviteljev Lean metode. Medtem, ko se ostale agilne metode nagibajo k agilnosti se ta metoda k efektivnosti (Janes in Succi 2014, 144).

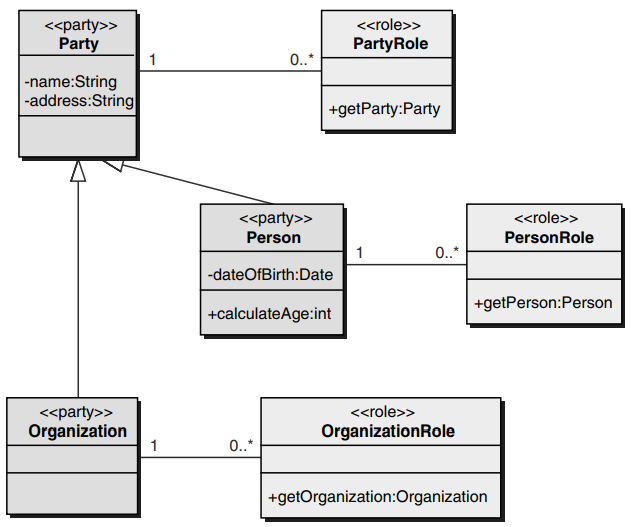
3.5.7 FUNKCIONALNO USMERJEN RAZVOJ

Funkcionalno[[36]](#footnote-36) usmerjen razvoj je zgrajen okoli jedra najboljših praks (Palmer in Felsing 2002, 36). Jedro sestavljajo; domensko modeliranje objektov, razvoj po funkcionalnostih, odgovornosti, razvojne ekipe, kontrolo, redne izdaje, upravljanje konfiguracij[[37]](#footnote-37) in poročanje.

Domensko modeliranje objektov je postopek izgradnje razrednih diagramov, ki predstavljajo posamezne tipe objektov in njihove medsebojne odvisnosti znotraj nekega problema (Palmer in Felsing 2002, 36). Slika 3.21 predstavlja diagram, kateri zagotavlja celosten okvir na katerega dodajamo funkcionalnosti. Pomaga pri razumevanju konceptualne integritete sistema (Palmer in Felsing 2002, 37).

Razvoj po funkcionalnostih se nadaljuje po identifikaciji razredov. Zbrane zahteve naročnika predstavljajo razredi s svojimi metodami (funkcijami), ki skupaj tvorijo funkcionalnosti. Funkcionalnosti nato ovrednotimo glede na uporabno vrednost naročnika. Ovrednotene služijo kot vodilo in sledenje napredku razvoja (Palmer in Felsing 2002, 39). Velikost in kompleksnost funkcionalnosti mora biti takšna, da je njena implementacija mogoča v času največ dveh tednov. Kompleksnejše funkcionalnosti se razgradi na manjše funkcije dokler vsak pod-problem ne predstavlja samostojne funkcionalnosti (Palmer in Felsing 2002, 41).

Slika 3.21 domenski diagram funkcionalnosti



Vir: (Palmer in Felsing 2002, 37)

Odgovornost določa področja odgovornosti razvoja za posamezni razred (del kode) (Palmer in Felsing 2002, 42). Vsakemu razvijalcu se pripiše odgovornost za določen del razredov, ki so predstavljeni v domenskem diagramu (Palmer in Felsing 2002, 43). To omogoča, nadzor nad kvaliteto in delovanjem za vsak posamezni del kode, ki je v domeni enega razvijalca in onemogoča preobremenjenost posameznikov z delitvijo odgovornosti, posledično dela (Palmer in Felsing 2002, 44).

Razvojne ekipe so sestavljene okoli funkcionalnosti. Te so navadno skupek razredov, zato ekipo sestavlja več posameznikov. Kot razredom se tudi funkcionalnostim pripišejo skrbniki, ki predstavljajo vodje razvojnih ekip (Palmer in Felsing 2002, 46). Ko je funkcionalnost končana se ekipa razpusti. Skrbnik funkcionalnosti nato znova sestavi ekipo, ki je odvisna od lastnikov razredov, kateri predstavljajo funkcionalnost (Palmer in Felsing 2002, 47).

Kontrola služi za zagotavljanje kvalitete načrtovanja in kodiranja (Palmer in Felsing 2002, 49). Kontrole se uporablja tudi z namenom procesnih izboljšav in tehnik (Palmer in Felsing 2002, 50). Izvajati se morajo objektivno, saj lahko vodijo v ponižanje razvijalcev. Slednji se morajo zavedati, da je kontrola odlično orodje za razhroščevanje in učenje ter, da ni orodje za identifikacijo učinkovitosti posameznika (Palmer in Felsing 2002, 51).

Redne izdaje sestavljajo končane funkcionalnosti integrirane v sistem. Sistem se lahko predstavi naročniku, odkrijejo se napake in težave, ki bi lahko pomenile tveganje v prihodnosti (Palmer in Felsing 2002, 51). Redne izdaje lahko odkrijejo napake zelo zgodaj v razvoju, generirajo ali posodobijo dokumentacijo (Palmer in Felsing 2002, 52).

Upravljanje konfiguracije je lahko enostavno ali izredno kompleksno. V teoriji FDD potrebuje sistem za spremljanje izvorne kode, s časovnim žigom in zgodovino sprememb. Napaka je v razmišljanju, da nadzor nad različico potrebuje le izvorna koda. Analize, načrti, komercialne pogodbe in ne nazadnje najpomembnejše, zahteve za razvoj predstavljajo artefakte, ki se uporabljajo in spreminjajo med razvojem (Palmer in Felsing 2002, 53).

Poročanje pomaga pri razumevanju trenutnega napredka projekta, hitrosti implementacije funkcionalnosti in splošni želeni rezultat. S tem znanjem projektni vodja usmerja tok poteka razvoja (Palmer in Felsing 2002, 53).

3.5.8 KANBAN

Kanban[[38]](#footnote-38) je metoda za definiranje, upravljanje in izboljšanje storitev, ki proizvajajo znanje. Je usmerjevalec hitrih in fokusiranih sprememb z namenom njihovega izkoriščanja za dosego ciljev. Metoda se izvaja z uporabo kanban sistema, ki omejuje delo v teku z uporabo vizualnih signalov (Anderson in Carmichael 2016, 1). Da lahko sistem označimo za Kanban mora ta ustrezati več pogojem. Sistem mora imeti vizualno ločiti delo v teku, določeno točko zaveze in izdaje (Anderson in Carmichael 2016, 13).

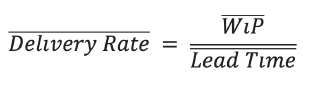
Proces se lahko definira kot serija korakov. Kanban tabla (sl. 3.22) prikazuje potek sistema skozi različne stopnje procesa od leve proti desni (Anderson in Carmichael 2016, 13).

Slika 3.22 kanban tabla poteka dela

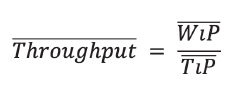
Vir: (Anderson in Carmichael 2016, 13)

Zaveza je eksplicitni dogovor med naročnikom in razvojem, ki narekuje razvoju delo na tistih postavkah katere želi naročnik. Zahteve ali postavke se nahajajo v bazenu idej, ki so lahko izbrane ali pa tudi ne. Čas od izbrane postavke do izdaje definira naročnikov dobavni rok[[39]](#footnote-39). Dobavni rok razvoja pa je čas od izbranih postavk do njihove dostave naročniku. Ta čas definira vse postavke, ki so v danem trenutku obravnavane in ga označuje kot delo v teku[[40]](#footnote-40).  
Hitrost s katero se izdajajo postavke imenujemo obseg[[41]](#footnote-41) izdaje, ki je definiran z številom postavk v določenem dobavnem roku. V poteku dela, ki nima trenda se ga lahko izračuna kot kvocient povprečja postavk v obravnavi in povprečje dobavnih rokov razvoja (sl. 3.23). Imenujemo ga tudi mali[[42]](#footnote-42) zakon.

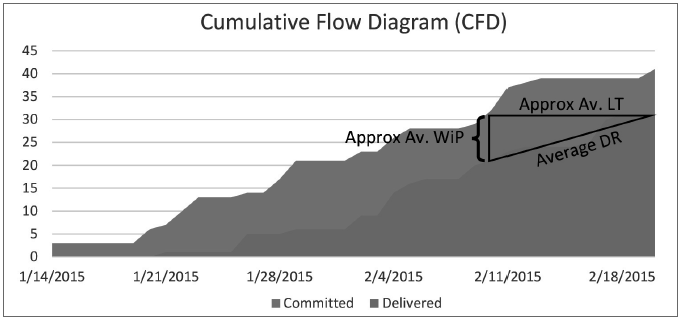
Slika 3.23 povprečni obseg izdaje

  
vir: (Anderson in Carmichael 2016, 15)

Če bi želeli pridobiti informacijo za druge poteke dela Kanban sistema lahko namesto dobavnega časa uporabimo čas v procesu (TiP – Time in Process) (sl. 3.24). Tedaj dobimo kvocient, ki predstavlja pretočnost[[43]](#footnote-43). Namesto TiP lahko uporabimo tudi druge specifične časovnice; čas v razvoju (TiD), čas v testu (TiT), čas v sistemu (TiS) ali čas v čakalni vrsti (TiQ) (Anderson in Carmichael 2016, 15).

Slika 3.24 povprečni obseg izdaje  
  
vir: (Anderson in Carmichael 2016, 15)

Mali zakon lahko predstavimo tudi kot kumulativni graf pretočnosti (tab. 3.3), ki prikazuje kumulativno število prispelih in izdanih postavk sistema. Hipotenuza trikotnika na v tabeli predstavlja povprečni obseg izdaje (Anderson in Carmichael 2016, 16).

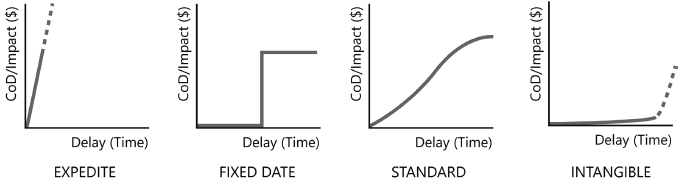
Tabela 3.3 kumulativni graf pretočnosti

Vir: (Anderson in Carmichael 2016, 16)

Načela Kanbana definirajo primarne aktivnosti za uspešno prakticiranje Kanban sistema. Sestavljajo jih vizualizacija, omejevanje dela v teku, upravljanje poteka, eksplicitno določanje pravil, zanke povratnih informacij in optimizacija (Anderson in Carmichael 2016, 17).   
Vizualizacija se prične s Kanban tablo (sl. 3.22), na kateri morajo biti vizualno označene in definirane točke zaveze in jasno opredeljene meje dela v teku. Vizualizacije je lahko digitalna vendar je znano, da ekipe, ki uporabljajo fizične table pogosto najdejo druge kreativne načine za prikaz informacij (Anderson in Carmichael 2016, 18).

Omejevanje dela v teku pomaga, da se nove postavne ne začnejo dokler drugo delo ni končano ali preklicano. Preveč dela naenkrat podaljšuje pretočnost, onemogoča odzivnost naročniku in spreminjajočim razmeram ter priložnostim. Opazovanje, omejevanje in optimizacija količine dela vodi v izboljšano pretočnost, kvaliteto in skrajšan čas razvoja (Anderson in Carmichael 2016, 19).

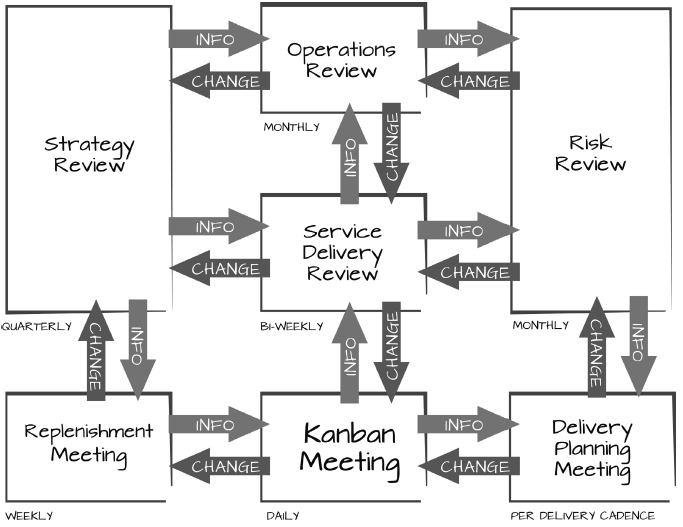
Upravljanje pretoka omogoča maksimiranje dobavnih vrednosti, zmanjšanje dobavnih rokov in omogoča predvidljivost. Ključno za razumevanje maksimiranja pretoka so stroški zamude. To so količine izgubljenega kapitala ob daljši časovni zamudi implementacije. Navadno so stroški zamude odvisni od časa vendar niso vedno konstantni. Kanban zato uporablja štiri tipe za določitev nastalih stroškov v odvisnosti od časovne zamude (sl. 3.25) (Anderson in Carmichael 2016, 21).

Slika 3.25 stroški zamude za Kanban

Vir: (Anderson in Carmichael 2016, 21)

Eksplicitno določanje pravil je artikulacija in definicija procesa izven poteka dela. Pravila omejijo aktivnosti v procesu in izzovejo nastajanje posebnih značilnosti omejitev, ki se jih lahko optimizira. Obnašanje kompleksnih sistemov je težko napovedati. Pravila, ki se zdijo intuitivna pogosto ustvarijo obratne rezultate (Anderson in Carmichael 2016, 22). Zato je pomembna aplikacija pravil in mehanizmov za njihovo optimizacijo v kolikor se spoznajo za neproduktivne (Anderson in Carmichael 2016, 23).

Zanke povratnih informacij so pomemben del vsakega procesa, še posebej, če gre za procese, ki vključujejo evolucijske spremembe. Kanban definira sedem sklopov povratnih informacij, ki jih imenuje kadence (sl. 3.26). Kadenca se nanaša na čas med pregledi. Ti so lahko dnevni ali mesečni. Izbira prave kadence je odvisna od konteksta in vpliva na kvaliteto rezultatov (Anderson in Carmichael 2016, 24). Implementacija sedmih kadenc ne implicira dodajanje sedmih različnih sestankov (Anderson in Carmichael 2016, 25). V osnovi lahko en sestanek zajema več kadenc (Anderson in Carmichael 2016, 26).

Slika 3.26 kadence po Anderson in Charmichael  
  
Vir: (Anderson in Carmichael 2016, 24)

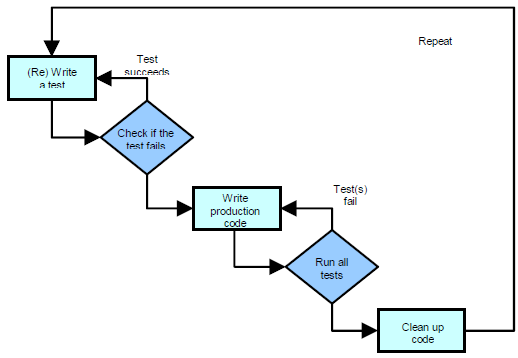
Optimizacija je temeljni cilj metode Kanban. Metoda uporablja paradigmo vitke metodologije za doseganje evolucijskih in inkrementalnih izboljšav. Metoda omogoča sprejemanje in oblikovanje uporabnih sprememb in hkrati zavračanje neuporabnih (Anderson in Carmichael 2016, 26).

V nasprotju z metodo Scrum, Kanban dovoljuje spreminjanje dostavnih rokov postavk, med katerokoli stopnjo procesa. Prav tako lahko Kanban spreminja prioritete postavkam. V kolikor se pojavi urgentna funkcionalnost ta nastopi kot naslednja za v razvoj. Sistem ohranja veliko fleksibilnosti Agilnih procesov in je lahko apliciran na razvojne ekipe vseh velikosti in specializiranosti posameznikov (Unger in Novak 2011, 180).

3.5.9 TESTNO USMERJEN RAZVOJ

Pristop za izboljšanje procesa razvoja in zmanjšanje ponavljajočega dela je testno[[44]](#footnote-44) usmerjen razvoj. Sloni na ponovitvah (sl 3.27) kratkih razvojnih ciklov. Je evolucijski proces pri katerem prvo izdela test pred produkcijsko kodo, ki ustreza temu scenariju (Duka in Hribar 2010, 1). Z razliko od standardnega testiranja je pri TDD test napisan pred kodiranjem. Razlog tega je, da testni scenariji vodijo načrtovanje, saj ti določajo potrebe (Duka in Hribar 2010, 2).

Slika 3.27 prikaz cikla TDD

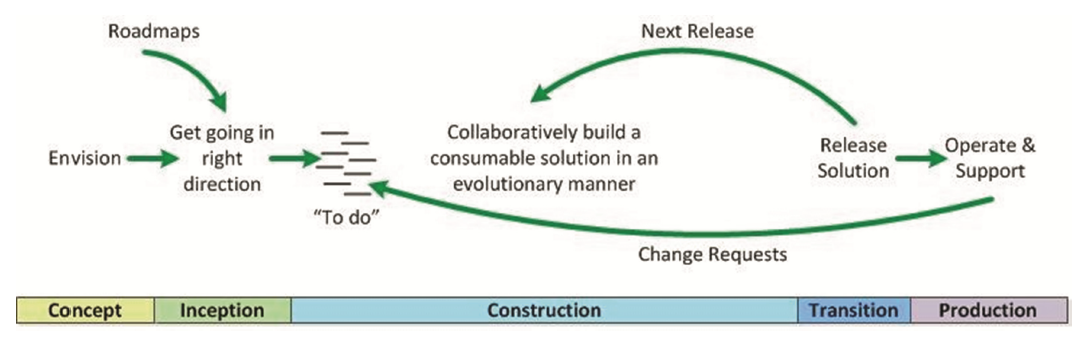
  
vir: (Duka in Hribar 2010, 2)

Pri TDD je prvi korak izdelava avtomatiziranih testnih scenarijev, ki predstavljajo zahteve. Testi vsebujejo trditve, ki imajo trdilni ali nikalni izid (true, false). Vsaka izgradnja funkcionalnosti se začne s testom. Za izdelavo testa mora razvijalec zahteve in specifikacije poznati do popolnosti. Pri tem mu lahko pomagajo primeri uporabe ali zgodbe (gl. pog. 3.5.1 XP). Po pisanju testa sledi razvoj funkcionalnosti. Pomembno je, da je ta napisana le z namenom uspešnega prestajanja testa. Uspešni izid testiranja funkcionalnosti potrdi delovanje kode in njeno ustreznost specifikaciji. Po uspešnem testiranju sledi refaktoriranje. Proces predstavlja optimizacijo kode z brisanjem podvojenih instanc ali drugih namenov brez nevarnosti ogrožanja obstoječega delovanja (Duka in Hribar 2010, 2). S ponovitvijo novega testa potisnemo funkcionalnost v delovanje (Duka in Hribar 2010, 3). TDD prinaša hitre rezultate, fleksibilnost, avtomatsko beleženje regresijskih testov in enostavno delujočo kodo (Duka in Hribar 2010, 4).

3.5.10 DISCIPLINIRANA AGILNA DOSTAVA

DAD[[45]](#footnote-45) je agilna metoda, katere osrednji del je izbira dostavnih (izdajnih) pristopov na podlagi problematike. Metoda omogoča taktična umestitev agilnih strategij razvoja skozi življenjski cikel glede na potrebe razvoja. Zagovarja efektivnost ekip z istočasno uporabo agilnih in Lean usmeritev razvoja (Ambler in Lines 2016, 4). To predstavlja veliko prednost, saj oba pristopa posedujeta veliko tehnik, strategij in praks. Brez DAD okvirja pa je težko vedeti katere spadajo skupaj (Ambler in Lines 2016, 5). DAD omogoča usmeritve kdaj in kako jih umestiti v proces. DAD podpira celoten cikel izdaje (Ambler in Lines 2016, 6). Slika 3.28 prikazuje prikaz procesnega cikla.

Slika 3.28 življenjski cikel discipliniranega agilnega procesa



Vir: (Ambler in Lines 2016, 6)

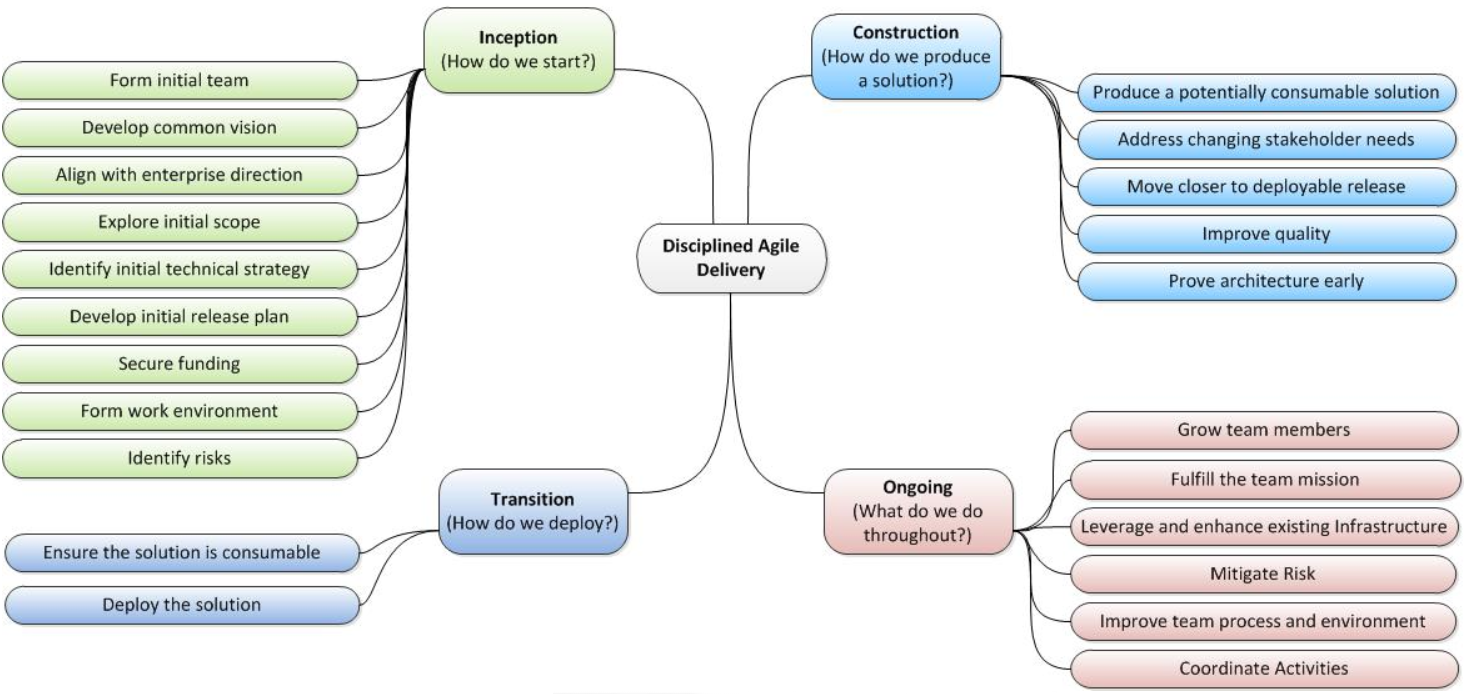
Notranje tri faze začetka, izgradnje, in prehoda, formirajo razvojni del. V tem delu se izvaja inkrementacija produkta. Navadno gredo sistemi večkrat skozi ta del procesa (Ambler in Lines 2016, 6). DAD pozna štiri različice razvojnega dela procesa. Prva je agilna/bazična različica, ki razširja cikel izgradnje po Scrum metodi. Druga je naprednejša/lean, kateri osnova je Kanban. Tretja je neskončni cikel izdajanje in zadnji četrta je raziskovalna, kateri osnova predstavlja Lean Start-up[[46]](#footnote-46) pristop. Razvojna ekipa na podlagi problematike izbere ustrezni pristop razvojenga dela (Ambler in Lines 2016, 7).

Za potrebe izboljšav procesa DAD posvoji uporabo ciljno usmerjenih tehnik. DAD razdeli fazne odločitve na naloge, ki so povzete in predstavljene z miselnimi vzorci (sl. 3.29). Vsaka ekipa lahko definira svoje naloge, ki jih odkrivajo z izkušnjami (Ambler in Lines 2016, 7). Naloge predstavljajo možnosti, ki jih imajo ekipe v določeni fazi procesa. Zaradi različnih praks, ki se odkrivajo in spreminjajo je nemogoče naloge definirati (Ambler in Lines 2016, 8).

DAD je okvir, ki zagotavlja delovanje geografsko razpršenih številčnih ekip. Omogoča razvoj tehnično enostavnih ali kompleksnejših projektov z življenjsko dobo podpore (Ambler in Lines 2016, 9).

DAD proces upošteva načela; izbire tipa razvoja, optimizacije celotnega cikla, konstantnih izboljšav, sprejemanje procesnih sprememb, lastništvo procesov razvojnih ekip in učenje iz izkušenj (Ambler in Lines 2016, 13).

Slika 3.29 miselni vzorec ciljev razvoja

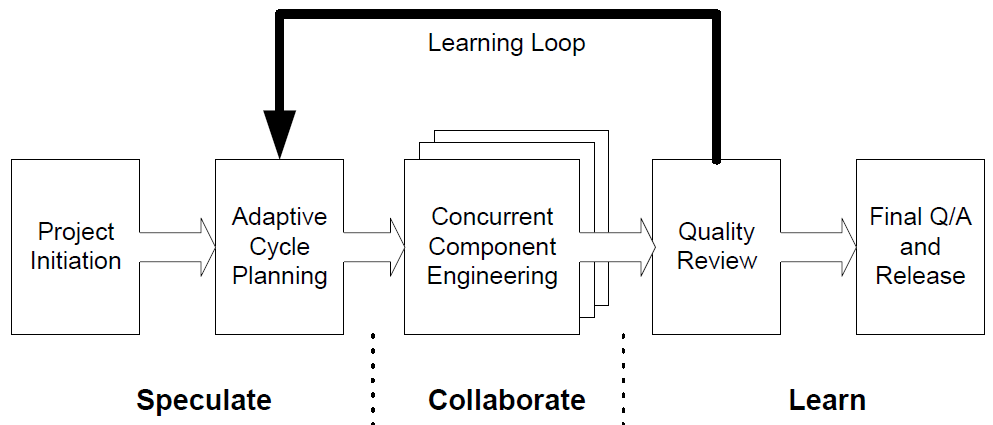


Vir: (Ambler in Lines 2016, 6)

3.5.11 PRILAGODLJIV RAZVOJNI MODEL

Prilagodljiv razvojni model (ASD[[47]](#footnote-47)) se je razvil iz praks uporabe RAD pristopa. Pristop se fokusira na razvoj velikih in kompleksnih sistemov. Metoda spodbuja uporabo inkrementalnih in iterativnih tehnik s konstantnim prototipiranjem. ASD spremljajo tri faze razvoja; predvidevanje, sodelovanje in učenje (Abrahamsson in dr. 2017, 72). Slika 3.30 prikazuje ASD bolj podrobno. Inicializacijo projekta predstavlja definiranje projektne naloge. Z njo predstavimo grob okvir končnega produkta na podlagi katere se usmerja razvoj. Pomemben del pri definiranju projektne naloge je pridobitev informacij, ki so potrebne za uspešen razvoj. Deli se na tri dokumente; projektno vizijo, podatke projekta in opis projekta. Inicializacijska faza določi splošni razpored projekta kot tudi razpored dela in nalog znotraj cikla. Cikel navadno traja 4 do 8 tednov (Abrahamsson in dr. 2017, 73). ASD je bolj usmerjen na delovanje komponent kot izvrševanje nalog. To povzema faza sodelovanja, kjer je lahko več komponent v vzporednem razvoju. Načrtovanje faz je iterativen proces, katerih podlaga so ponavljajoče kontrole kvalitete, ki se izvajajo skupaj z naročnikom ali skupino strokovnjakov ob koncu vsakega cikla. Redkost kontrol ASD rešuje z izvajanjem projektnih JAD delavnic z namenom preciziranja želenega delovanja produkta in negovanja komunikacije (Abrahamsson in dr. 2017, 74).

Slika 3.30 podrobnejši prikaz procesnih korakov ASD



vir: (Abrahamsson in dr. 2017, 74)

Zadnja faza modela sestoji iz kontrole kvalitete in izdaje produkta. ASD ne predpisuje delovanje te faze vendar opozarja na pomembnosti zavedanja pridobljenih izkušenj. Dokumenti obdukcije projektov so izredno pomembni dokumenti, ki te izkušnje povzemajo (Abrahamsson in dr. 2017, 75).

ASD ne postavlja omejitev njegove uporabe. V nasprotju z večino razvojnih metod ASD ne spodbuja razvoja na kolokaciji. Izgradnja kompleksnih sistemov včasih potrebuje obsežno znanje, ki združuje strokovnjake različnih področij. Razvoj pogosto sestavljajo ekipe, ki so prostorsko, časovno ali organizacijsko deljene. ASD spodbuja uporabo komunikacijskih orodij in strategij s postopnim stopnjevanjem delovnih omejitev za zagotavljanje podpore distribuiranemu razvoju (Abrahamsson in dr. 2017, 76).

3.5.12 SCRUMBAN

Scrumban je hibrid Scrum in Kanban agilnih modelov namenjen obvladovanju dinamičnih sprememb naročniških zahtev in frekvenčnih težav z izvorno kodo. Od metode Scrum je povzel tehnike dnevnih sestankov, definiranja zgodb in samo organizacijski pogled na razpored dela. Zaradi boljše organizacije dela, preglednosti napredka in sprememb se za koordinacijski mehanizem uporablja Kanban tehniko WiP. Ta omogoča koordinacijo vlečenja dela v nasprotju z Scrum, ki prakticira potiskanje dela. Zaradi potiskanja dela v praksi velikokrat prihaja do zastojev zaradi večopravilnosti razvoja. Tehnika vlečenja pomeni, da je toliko enot dela v teku kolikor jih lahko razvoj zaključi. Optimizacija dela v teku je odvisna od prave izbire nalog za dosego optimalne pretočnosti. WiP pomaga razvijalcem omejiti večopravilnost z namenom povečanja produktivnosti (Yilmaz in O’Connor 2016, 238).

4 RAZVOJ VIDEO IGER

Veliko je interesa na temo raziskovanja discipline metodologij razvoja video iger (Ampatzoglou in Stamelos 2010, 889). Opaziti je bilo povečano število raziskav na tem področju. Poleg tega aktivnost raziskovanja inženiringa video iger v primerjavi z raziskovanjem razvoja programske opreme raste hitreje (Ampatzoglou in Stamelos 2010, 893). Od leta 2008 je bilo opaziti več študij na temo upravljanja razvoja video iger (Ampatzoglou in Stamelos 2010, 894), ki zajema tehnike upravljanja s procesi (Novak 2012, 351). To potrdi tudi sistematičen pregled literature, ki so ga izvedli Aleem in kolegi (2016).

Razvidno je da je število raziskav doseglo vrhunec med leti 2008-2015 (PRILOGA A). Kljub porastu raziskav smo opazili, da se organizacije poslužujejo različnim pristopom in njihovim hibridom predstavljenih v prejšnjem poglavju. Našli smo le nekaj raziskav, ki so definirale razvojni proces, metodologijo ali življenjski cikel izključno za razvoj video iger. Morda je krivo to, da v literaturi še ni bilo podrobneje predstavljenih najboljših praks ali strategij za razvoj iger. Poleg tega je bila disciplina okrnjena orodja za optimizacijo procesa, ki je ključno za izboljšanje procesa (Aleem, Capretz, in Ahmed 2016, 26). Menimo, da celovito podedovanje procesnih modelov in metod iz standardnega razvoja programske opreme za igre ni primerno. Zaradi svoje agilnosti, tendence po zmanjšanju dokumentacije, multidisciplinarnosti, kreativnosti, abstraktno definiranih zahtev (Ruonala 2016, 5), krajšega življenjskega cikla (O’Hagan in O’Connor 2015, 3) posedujejo lastnosti, ki jih evidentno ločuje od standardnega razvoja programske opreme.

4.1 ZNAČILNOSTI IGER

Za razvoj iger je jasno, da poseduje značilnosti, ki niso prisotne v ostalih disciplinah inženiringa programske opreme (Abrahamsson in dr. 2017, 1).

Igre se osredotočajo na igranje (Esposito 2005, 3). Igranje predstavlja element, ki emocionalno pritegne igralce. Je nekakšno stanje, ki je podobno zabavi (Fullerton 2008, 91). Posledično se igre smatrajo za najbolj ekstremne hedonične informacijske sisteme, pa vendar se pri utilitarnih namenih uporabe uporabna motivacija premakne iz zabave v uporabnost (Hamari in Keronen 2017, 136).

Igre lahko vodijo tako intrinzične kot ekstrinzične motivacije (Novak 2012, 198). Številne študije kažejo pozitivne učinke učnih iger na področju motivacije do učenja in učnega izkoristka. Motivacija in pozitivne emocije sta pomembna faktorja pri učenju (Imlig-Iten in Petko 2014, 151), katerih uravnoteženost igre predstavlja efektivnost, ki je pridobljena s konsistenco zahtevnosti in zabave pri igranju[[48]](#footnote-48) (Novak 2012, 202). To predstavlja zabavno izkušnjo, ki je najpomembnejše merilo uspeha iger (Cooper in Scacchi 2015, 12). V razvoju iger se uporablja termin uporabniška izkušnja[[49]](#footnote-49), ki jo z iterativnim procesom upoštevanja uporabniških povratnih informacij oblikovalci izboljšujejo tekom procesa razvoja (Fullerton 2008, 2). Potrebno je razumevanje, da igra ni izkušnja vendar je izkušnja nekaj kar igra omogoča (Schell 2008, 10). S povratnimi informacijami je povezana tudi pozornost na defekte. Zavoljo tega osrednja aktivnost razvoja video iger navadno postane konstantno testiranje igranja (Cooper in Scacchi 2015, 10). To aktivnost predstavlja testni protokol in velja za iterativni proces med načrtovanjem, razvojem in testiranjem (Sylvester 2013, 295). Zato se razvoji iger nagibajo od tradicionalnih življenjskih ciklov razvoja k bolj inkrementalnim, kateri na podlagi uporabniških povratnih informacij izpopolnjujejo ali izboljšujejo verzije programa (Cooper in Scacchi 2015, 10).

Zanimivost igre je tudi v njenem življenjskem ciklu razvoja. Njen življenjski cikel se nikoli ne konča. Navadno manj izkušena ekipa ostane in skrbi za posodabljanje virtualnega sveta (Bartle 2003, 93). Izdajajo se popravki, posodobitve in razširitve v glavnem z namenom daljšanja življenjske dobe igre (Novak 2012, 365). Posebnost je tudi vključenost uporabnikov v življenjski cikel razvoja, ki se bistveno razlikuje od standardnega razvoja programske opreme. Uporabniki so lahko vključeni v beta testiranja (odprto beta testiranje) programske opreme in pripomorejo k zagotavljanju kvalitete v zameno za brezplačno igranje (Levy in Novak 2009, 54) ali pa modificirajo[[50]](#footnote-50) igre s čimer se vključijo v primitiven razvojni proces igre (Cooper in Scacchi 2015, 5).

4.2 PROCESI V PRAKSI

Po podatkih spletne ankete v Avstrijski industriji iger 23% podjetij razvija igre z ad-hoc pristopi, 77% pa jih uporablja Scrum ali XP. Verjetno je k rezultatu botrovala velikost samih podjetij, saj ima 85% podjetij vsaj 4 zaposlene medtem, ko ima le 15% podjetij 15 ali več zaposlenih. Ne glede na rezultate so vsa podjetja nakazala uporabo nekakšnih fleksibilnih, sekvenčnih ali agilnih pristopov (Musil in dr. 2010, 5).

Raziskava, ki so jo naredili O'Hagan in kolegi je pokazala, da se pri razvoju iger uporablja 47% agilnih in 53% hibridnih procesov. Izmed 404 študij so identificirali 23 procesnih modelov izmed katerih so bili vidnejši: XP, Scrum, Kanban, rapidni, inkrementalni in komponentni razvojni modeli. Medtem, ko so prvi trije našteti strogo agili, vsi bazirajo na iteracijah. Vsi modeli so se razlikovali le po številu iteracij, katerih število je bilo večje pri agilnih in manjše pri hibridnih procesih (O’Hagan, Coleman, in O’Connor 2014, 187).

Koutonen in Leppänen (2013) sta izvedla raziskavo o uporabi agilnih procesov v finski industriji igre. Raziskava je zajemala najmanjša podjetja kot tudi enega največjih (npr. Rovio Entertainment). Vsa podjetja razen enega so v vsaj enem koraku procesa uporabili agilne metode (Koutonen in Leppänen 2013, 12).

Na podlagi dokumentov obdukcij razvoja sta Petrillo in Pimenta (2010) naredila raziskavo, ki je ugotavljala uporabo agilnih praks v razvoju. Ugotovila sta, da se v razvoju uporabljajo agilne prakse. Uporaba se lahko s prakticiranjem agilnih praks izvaja povsem nezavedno in celo neformalno (Petrillo in Pimenta 2010, 14).

Prav tako agilne pristope podpira Fullerton, saj meni, da je Scrum primeren za reševanje zapletenih problemov oblikovanja iger (Fullerton 2014, 369). Svoje mnenje podaja John Comes[[51]](#footnote-51), ki meni, da naj opustimo kaskadne procese in se sprejmemo agilne (John Comes v Novak 2012, 366).

4.3 PROCESNI MODELI ZA IGRE

Babu in Maruthi (2013) podrobneje definirata faze življenjskega cikla za igre. Faze si sledijo linearno: izdelava zgodbe, razvoj skripte, študija izvedljivosti, promocijski demo, oblikovanje, oblikovanje postavitev, modeliranje, teksturiranje, animiranje, oblikovanje stopenj, kodiranje, testiranje, razhroščevanje, integracija, testiranje igranja (Babu in Maruthi 2013, 1491). Z definicijo življenjskega cikla podata razširjen vpogled v faze razvoja iger (Babu in Maruthi 2013, 1502).

Ramadan in Widyani (2013) na podlagi ključnih aktivnosti različnih organizacij in raziskovalcev predlagata GDLC[[52]](#footnote-52). Pristop predlaga faze: iniciacijo, pred produkcijo, produkcijo, testiranje, beta in izdajo. Iniciacija služi za stvaritev koncepta igre in enostaven opis specifikacije. Pred produkcija zajema revidiranje oblikovanja in izdelava prototipa. Po tej fazi je končan GDD[[53]](#footnote-53) na podlagi katerega je izdelan prototip. V prvi iteraciji produkcije služi prototip za temelje in strukturo v nadaljnjih pa se ga izboljšuje. Produkcija se konča z izdelanim prototipom, ki predstavlja celoto (Ramadan in Widyani 2013, 98). Sledi testiranje, ki izda poročilo defektov in seznam izboljšav. Rezultat testiranja odloča ali se razvoj nadaljuje v beta fazo. Slednja predstavlja identifikacijo defektov in povratnih informacij uporabnikov. Iz beta lahko ponovno sledi faza produkcije z namenom izboljšanja igre ali faza izdaje v kolikor je rezultat beta testiranja zadovoljiv (Ramadan in Widyani 2013, 99).

Aslan in Balci (2015) predstavita metodologijo za kompleksni razvoj iger GAMED[[54]](#footnote-54). PRILOGA B prikazuje življenjski cikel DEG[[55]](#footnote-55), ki je osnova metodologije. DEG cikel je sestavljen iz štirih faz: oblikovanja igre, oblikovanja programske opreme, implementacije (izdaja) in učenja na podlagi iger (povratne informacije) (Aslan in Balci 2015, 309). Metodo označujeta za iterativno, saj pričakuje povratne tranzicije. V kolikor se pojavi potreba se postavimo en korak nazaj in ponovimo delo. Tipično se pomikamo naprej in nazaj med procesi dokler ne dosežemo zadovoljive kvalitete delavnih produktov (Aslan in Balci 2015, 310). Kot podporni proces metodologija uporablja spiralni model, ki ga aplicira v fazo oblikovanja igre (PRILOGA C). Oblikovanje poteka po spiralnem vzorcu pri čemer vsaka iteracija pomeni izpopolnjen dokument specifikacije ideje. Iteracija vsebuje aktivnosti: prototipiranja, testiranja igranja, evalvacija in analize tveganj. Vzorec se nadaljuje dokler kvaliteta oblikovanja ni potrjena v aktivnosti evalvacije. Končni dokument te aktivnosti predstavlja specifikacija oblikovanja igre (Aslan in Balci 2015, 313). Specifikacijo oblikovanja se v aktivnosti zbiranja zahtev nadgradi v dokument specifikacije potreb. Specifikacija potreb se po potrebi zahtevne arhitekture nadgradi v dokument specifikacije arhitekture. V zadnji aktivnosti faze oblikovanja se predhodni dokument (specifikacija arhitekture ali specifikacija potreb) posodobi v dokument specifikacije oblikovanja programske opreme iger (Aslan in Balci 2015, 315–316). Na podlagi tega dokumenta se v fazi implementacije izvede aktivnost programiranja v kateri se zgradi DEG ali njene komponente. Metoda GAMED predvideva vzdrževanje, ki se izvaja na podlagi povratnih informacijah. Po potrebi posodobitve se ponovno izvede celoten življenjski cikel (Aslan in Balci 2015, 317).

Barbosa (2017) predstavi Game-Scrum, ki je hibrid Scrum in XP procesnih modelov (PRILOGA D). Scrum skrbi za upravljanje projekta medtem, ko XP zagotavlja inženiring. Procesni hibrid je primeren za razvojne ekipe z malo ali nič izkušnjami (Barbosa 2017, 293). Proces se deli na faze: pred produkcija, faza izdelave GDD, produkcija in post produkcija. V pred produkciji poteka iskanje dejavnika zabave, izdelava idealnega koncepta in oblikovanja. Delo poteka po principu metode poskušanja in popravljanja napak. V tej fazi se predvideva izgradnja enostavnega prototipa, ki je zaradi narave svoje hitre konstrukcije navadno zavržen. Sledi izgradnja GDD, ki je v produkcijski fazi preveden v dnevnik zaostankov. Produkcija poteka iterativno, priporoča Kanban metodo za kreacijo umetnin in podajanje časovnih okvirjev, mejnikov njihove izdelave (Barbosa 2017, 293). V kolikor je v ekipi več programerjev se priporoča uporaba Scrum in XP tehnik. Po končani igri se uporabi testiranje igranja za zagotavljanje kvalitete in dejavnika zabave. Zadnje dejanje v razvoju predstavlja izdelava dokumenta obdukcije razvoja, ki služi za identifikacijo pomanjkljivosti v preteklem razvojem procesu (Barbosa 2017, 294).

4.4 POMANJKLJIVOSTI PROCESNIH MODELOV ZA IGRE

Različice modelov se pojavijo tudi v sivi literaturi. GUP[[56]](#footnote-56) predstavi rang procesnih pristopov od RUP do XP. Vendar kljub vsemu GUP ne definira poteka aktivnosti niti ne poda osnovnih usmeritev kako združiti estetske discipline razvoja z inženirskimi (Wilson Brotto Furtado, 22). Flood (2003) pri definiranju GUP predstavi in komentira GWP[[57]](#footnote-57). Kljub številnim pomanjkljivostim ta proces prvič predstavi artefakte, ki so specifični za igre (Wilson Brotto Furtado, 21) (specifikacija igre, umetniška biblija, tehnična specifikacija) („Game Unified Process - EcuRed“ 2018). GWP omenjata tudi Flynt in Salem (2004) kot pristop, ki so ga uporabili razvijalci igre Ankh. GWP ima težavo, da predstavlja linearen proces. Ocenjevalci, testerji, ravnatelji ne vidijo produkta dokler je ta v razvoju. V realnosti je tako, da se po evalvaciji navadno zahtevajo spremembe. To zahteva, da razvoj ponovi faze, ki so bile predvidoma že končane („Game Unified Process - EcuRed“ 2018). Razvoj iger ni linearen proces (Flynt in Salem 2004, 151). V praksi je tendenca po ogibanju uporabe kaskadnih pristopov, ker je nemogoče definirati in načrtovati program vnaprej (Rucker 2002, 38). V praksi je GWP (razvoj igre Ankh) predstavljal napor. Razvoj igre potrebuje fleksibilnost zato, da lahko ekipa vključi spoznanja, ki so se identificirala tekom razvoja z namenom konstantne izboljšave kvalitete (Flynt in Salem 2004, 18). GWP in prav tako GUP ne podajata priporočila uporabe ali referenčne modele za aplikacijo teh procesov, k temu najverjetneje vpliva tajna narava industrije iger (Wilson Brotto Furtado, 22).

Med neuspešne spada tudi model, ki ga objavi Demachy (2003) na Gamasutra. Interpretira XP metodologijo in predlaga XGD[[58]](#footnote-58) za video igre. Vendar pristop ne zagotavlja strukturiranih usmeritev, deloma zato, ker gre za agilno metodologijo (Wilson Brotto Furtado, 22).

Poleg naštetih se v akademski literaturi pojavita tudi dve pobudi procesov AGP[[59]](#footnote-59) in Predpisujoča metodologija za razvoj video iger, ki sta nastala v okviru zaključnih nalog. Obema predlogoma primanjkuje evalvacijski proces za določanje njune ustreznosti (Wilson Brotto Furtado, 22). Poleg tega igre niso predpisujoče (Koster 2013, 156). Tipične lastnosti razvoja iger kot so oblikovanje ali definiranje igranja je težko ali celo nemogoče specificirati brez različice sistema, ki jo lahko preizkusimo (Petrillo in Pimenta 2010, 10). Zato kakršnikoli predpisujoči (sekvenčni) pristopi niso priporočljivi.

Kljub temu, da predlogi procesov v prejšnjem poglavju predstavljajo bolj strukturirane jih spremljajo številne pomanjkljivosti. V tem razdelku smo že večkrat komentirali linearnost procesov, ki se izražajo s predpisovanjem poteka razvoja. Kljub temu, da Babu in Maruthi (2013) podrobneje definirata življenjski cikel in opišeta posamezne faze je model linearno orientiran. GDLC je v nasprotju s tem visoko iterativen, ki se lahko iz faze beta testiranja vrne v pred produkcijo, medtem, ko med produkcijo poteka iterativna izboljšava prototipa. Čeprav pristop podpira post produkcijsko fazo, ki jo označuje za izdajo, zelo slabo opisuje aktivnosti, ki se izvajajo v tej fazi. V razvoju je postala nekakšna praksa, da so popravki takoj po izdaji, saj zaradi hitenja ni mogoče testirati vseh funkcionalnosti (Bates 2004, 216). Glavni razlog je ujemanje dobavnih rokov in najbolj priljubljenih časov izdaj (božič) (Ruonala 2016, 8). Zato je bistvenega pomena, da je proces nadaljnje podpore dobro definiran. Poleg tega GDLC-ju primanjkuje definiranja artefaktov predvsem dokumentacije. Poleg prototipa, GDD obstajajo še umetniška sredstva, ki lahko že med produkcijo vplivajo na testiranje igranja. (Sylvester 2013, 286). Posledično GDLC tako ne definira estetskih disciplin produkcije. V nasprotju z GDLC je metoda GAMED podprta z dokumentacijo. Težava je le v tem, da je dokumentacija zgrajena na dopolnjevanju. V PRILOGA E je prikaz artefaktov, ki jih avtorji identificirajo med razvojem igre. Čeprav ni določenih standardov za kreiranje dokumentacije veljajo določeni dokumenti za osnovne komponente trdnih temeljev. In te predstavljajo koncept, predlog igre, GDD, TDD[[60]](#footnote-60), vodnik umetniškega sloga projektni načrt (Novak 2012, 380-394) Vsi so predstavljeni kot samostojni dokumenti, nekatere ekipe lahko celo razdelijo GDD tudi več dokumentov, ki posebej predstavljajo napredovanje po stopnjah, igranje in zgodbo (III 2004, 319). GAMED predstavlja odlično podlago za definiranje učnih iger vendar je pristop terminološko oslabljen. Pri drugih avtorjih se določena terminologija ponavlja tako pri definiranju artefaktov (PRILOGA E) kot pri definiranju faz razvoja (PRILOGA F), katere pa pristop ni podedoval. Največjo težavo pristopa predstavlja obnašanje cikla. Faza oblikovanja je iterativen proces (Schell 2008, 79), kar predvideva tudi GAMED, vendar se razvoj prototipa konča s fazo oblikovanja in definiranja dokumenta, ki po opisu predstavlja GDD. Prototip je ob koncu zavržen, razvoj programske opreme pa se ponovno izvede tokrat linearno na podlagi podrobno definirane dokumentacije (GDD), ki služi kot funkcionalna specifikacija. V nasprotju s funkcionalnimi specifikacijami je GDD zaradi svoje narave bolj organski in dinamičen. (III 2004, 310) in se v toku produkcije dnevno spreminja (Novak 2012, 391). Ker je prototip zavržen je opravljen presežek dela, saj je potrebno produkt ponovno razvijati. Prav tako se za iterativni razvoj smatra izvajanje iteracij med oblikovanjem igre, prototipiranjem in testiranjem (Novak 2012, 367), kar je v nasprotju z linearno realizacijo zahtev iz dokumentacije. Ta lahko privede do pozlačevanja[[61]](#footnote-61), ki pomeni sprejemanje nemogočih zahtev (Rucker 2002, 27) in je nasprotje iterativnemu razvoju (Novak 2012, 367).

5 INŽENIRING METOD

Inženiring metod (IM) je disciplina, ki se ukvarja z razvojem metod, tehnik in orodij za razvoj informacijskih sistemov (Engels in Sauer 2010, 412). Slika 5.1 prikazuje model splošnega življenjskega cikla IM. Po identifikaciji domene diskurza se prične analiza potreb, ki predstavlja prvi korak. Sledi več-stopenjski razvojni proces. Po razvoju se metodo namesti v uporabo in preveri. Preverjanje privede do novih ugotovitev ali potreb, katere vodijo v zagon ponovnega evolucijskega cikla (Engels in Sauer 2010, 422).

Slik 5.1 življenjski cikel IM



Vir: (Engels in Sauer 2010, 422)

Cilj inženiringa ni le definiranje procesa in vseh njegovih aktivnosti, temveč se osredotoča tudi na produkte, naloge, organizacijske vloge, orodja, tehnike in medsebojne relacije teh konceptov, zato takšne procese označuje za metode inženiringa programske opreme (MIPO) (Engels in Sauer 2010, 411-12). MIPO so sistematični postopki ali tehnike izvajanja opravil z namenom doseganja določenih ciljev (Engels in Sauer 2010, 414). MIPO koordinirajo in dokumentirajo razvojne procese, aktivnosti in artefakte. Strokovnjaki jih pogosto imenujejo procesni modeli (Engels in Sauer 2010, 415). Procesni model ustreza načinu dela, ki ga predpisuje metodologija v uporabi. Model lahko razumemo tudi kot koncept načrta (Rolland in Ufr 1994, 3). MIPO so v večini primerov sestavljene iz več delov kateri delijo proces razvoja na obvladujoče entitete (Sunyaev, Hansen, in Krcmar 2009, 646). Entitete v stroki predstavljajo kose, fragmente ali komponente. Poimenovanja so odvisna od pristopov k strukturiranju metod in granulacije posameznih entitet (Sunyaev, Hansen, in Krcmar 2009, 648). Harmsen, Brinkkemper in Oei označijo fragmente za osnovne gradnike metod. Na podlagi tega se razvijeta produktna in procesna dimenzija fragmentov metod (Sunyaev, Hansen, in Krcmar 2009, 647). Produktna dimenzija definira artefakte, ki se morajo izdelati in njihovo medsebojno povezanost (Engels in Sauer 2010, 413). Artefakti (produktni fragmenti) so ciljno orientirani in jih predstavljajo produkti, dokumenti, modeli, diagrami in drugo. Procesna dimenzija pa mora definirati postopke, ki so potrebni za izdelavo artefaktov in njihove izvajalce (Engels in Sauer 2010, 413). Ti postopki (procesni fragmenti) so procesno orientirani in jih predstavljajo faze, naloge in aktivnosti (Sunyaev, Hansen, in Krcmar 2009, 647). Ralyté in Rolland nadgradita pristop fragmentacije metod z uporabo kosov. Obstoječe procese in produktne fragmente združita v kose metod, ki zagotavljajo tesno povezavo procesnega in produktnega vidika (Sunyaev, Hansen, in Krcmar 2009, 648). Karlson in Wistrand pa delita metode na komponente. Komponento smatrata za najmanjšo smiselno entiteto metod katera sestoji iz procesa, notacije in koncepta (Sunyaev, Hansen, in Krcmar 2009, 648).

5.1 PROCESI

Procesi razvoja programske opreme so definirani z IEEE[[62]](#footnote-62) standardom 610.12, ki navaja: ''Postopek, s katerim se potrebe uporabnikov prevedejo v programsko opremo'' („IEEE 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology“ 2017). Definirajo pravila in priporočila za razvoj informacijskih sistemov in podajajo informacijo v kakšnem zaporedju izvajati ukrepe (Sunyaev, Hansen, in Krcmar 2009, 648). Podrobneje jih lahko opišemo kot zbirko deloma decidiranih korakov s podskupino medsebojno povezanih artefaktov, fizičnih in virtualnih virov, organizacijskih struktur in omejitev (García-Borgoñón in dr. 2014, 104). Proces je predstavljen s procesnim modelom. Razumemo ga kot sestav delovnih elementov (aktivnosti, naloge, faze) in toka dela, ki opredeljuje njihovo časovno sosledje (Engels in Sauer 2010, 425). Aktivnosti se lahko prekrivajo ali si iterativno sledijo. Te predstavljajo hierarhične kompozite katere skupaj tvorijo potek dela MIPO (Engels in Sauer 2010, 415). S procesi sta povezana tudi dva termina, ki tvorita časovno perspektivo procesa razvoja programske opreme: cikel razvoja programske opreme in življenjski cikel razvoja programske opreme. Življenjski cikel razvoja identificira vse aktivnosti, ki so potrebne za izgradnjo, zagon in vzdrževanje informacijskih sistemov. Čeprav je mnogo variacij procesov znotraj življenjskih ciklov veljajo naslednje za osnovne;

1. Identifikacija problema ali potreb in odobritev za nadaljevanje,
2. načrtovanje in spremljanje projekta,
3. odkrivanje in razumevanje podrobnosti problema ali potreb,
4. oblikovanje sistemskih komponent, ki rešijo problem ali zadovoljijo potrebam,
5. izgradnja, testiranje in integracija sistemskih komponent,
6. testiranje sistema in prenos v delovno okolje (Satzinger, Jackson, in Burd 2011, 6).

5.2 NOTACIJE

Notacije predstavljajo semantična, sintaktična in simbolna pravila za dokumentiranje. (Sunyaev, Hansen, in Krcmar 2009, 648). Za definiranje procesa je potrebna globoka metodologija skupaj z močnim in ekspresivnim jezikom procesnega modeliranja (Jablonski 2010, 393). Jeziki modeliranja so bili izdelani na podlagi programerskih jezikov, Petrijevih mrež in jezikov, ki temeljijo na pravilih (García-Borgoñón in dr. 2014, 104). Uporabnost jezikov modeliranja se razlikuje glede na njihovo paradigmo. Nekateri so bolj primerni za nadzorovanje procesa, objektno-orientirani za nadzor funkcionalnosti in kode, podatkovni pa za nadzor in definiranje podatkovnih baz (Sutton 2011, 295). Procese lahko predstavimo na podoben način kot programsko opremo. Pri izgradnji se pogosto uporablja termin programiranje procesov, pri čemer beseda programiranje implicira obsežnost programskega inženiringa, ne le kodiranja. Čeprav strokovnjaki zagovarjajo algoritemski zapis procesov je potrebno upoštevati dejstvo, da razvoj programske opreme zajema ustvarjalne vidike, ki so inherentno ne algoritmični. Procesi so deležni posebnih, nepričakovanih iregularnosti, ki onemogočajo slednje algoritemskim predpisom (Sutton 2011, 293).

5.3 KONCEPTI

Vsem MIPO so skupni aspekti, kateri nakazujejo njihovo zgradbo:

* Delovni potek razvojnega procesa,
* aktivnosti, ki se morajo izvesti,
* definicije delovnih produktov ali njihovih fragmentov,
* kriterij zaključka delovnih produktov,
* sposobnosti in odgovornosti akterjev in
* standarde, usmeritve, tehnike in orodja (Engels in Sauer 2010, 417).

Avtorji znotraj teh aspektov uporabljajo različne konstrukte, ki jih inženirji metod imenujejo koncepti (PRILOGA G) (Engels in Sauer 2010, 416).

Koncepti so kategorije, ki so vključene v procese in notacije. S koncepti opisujemo področja problema ali samo opravilo (Sunyaev, Hansen, in Krcmar 2009, 648). V praksi se srečamo z različnimi definicijami konceptov, pogosteje se uporabljajo: vloge (ljudje), procesi in aktivnosti (naloge), delavni produkti (artefakti), orodja (programska orodja in drugi pripomočki). Strokovnjaki se strinjajo, da mora programski inženiring zajemati vsaj tri koncepte: produkte, procese in akterje (Engels in Sauer 2010, 415).

6 METODE INŽENIRINGA

Inženiring metod nam podaja različne pristope k inženiringu procesov (Engels in Sauer 2010, 412). Kljub veliki izbiri pristopov je razbrati (Pog. 4), da je v domeni razvoja video iger primerna uporaba agilne filozofije. Verjetnost za to obstaja, ker se produkcijski proces iger konstantno razvija. Tradicionalni procesi (derivati manufaktur) enostavno ne morejo dohajati hitrosti sprememb razvoja iger (Unger in Novak 2011, 180). Primerno metodo uporabe nam nakazuje praksa uporabe procesnih modelov (Pog. 4.2), ki evidentno sporoča uporabo agilnih pristopov in njim podobnih hibridov, ki slonijo na iteracijah. Dejstvo je, da razvoj iger sloni na iteracijah (Novak 2012, 366) in kot smo spoznali (Pog. 1) tudi inkrementih. Iterativen in inkrementalen razvoj pa predstavljata temeljne principe agilnega modeliranja (Ambler 2002, 44). Spoznanja vodijo v prepričanje, da bi bila izbira metode agilnega modeliranja za izgradnjo procesnega modela upravičena. Vendar uporaba agilne metodologije ne bi zadostovala za izgradnjo sistematičnega procesa, saj potrebujemo proces v uporabi formalizirati. To pomeni, da mora biti proces dokumentiran in frekvenčno revidiran (Rucker 2002, 32). Agilna metodologija je bolj produkcijska filozofija kot trdna zbirka pravil (Unger in Novak 2011, 181). Predstavlja skupek najboljših praks, ki slonjo na principih in vrednotah predstavljenih v Agilnem Manifestu. Agilno modeliranje ne predpisuje postopkov za izgradnjo določenega procesa ampak spodbuja konstantno in efektivno modeliranje skozi nasvete in dobre prakse (Ambler 2002, 8). Zato bomo uporabili dodaten pristop k modeliranju z namenom podpore dokumentaciji in sistematizaciji procesa. Obetaven pristop naproti sistematičnemu in strukturiranemu razvoju programske opreme je uporaba tehnik meta-modeliranja za izgradnjo procesnih modelov (Engels in Sauer 2010, 419).

6.1 AGILNO MODELIRANJE

Bistvo agilnega modeliranja (AM) leži v njegovih praksah. Prakse AM so organizirane v štiri kategorije: iterativno in inkrementalno modeliranje, skupinsko delo, enostavnost in validacija (Ambler 2002, 44–45).

Iterativno in inkrementalno kategorijo definirajo prakse:

* apliciranja pravih artefaktov,
* izdelave več vzporednih modelov,
* iteracije do naslednjega artefakta,
* izvajanja manjših inkrementov (Ambler 2002, 45).

Artefakte predstavljajo UML grafikon stanja, izvorna koda, diagram poteka podatkov, primeri uporabe in drugi (PRILOGA E). Vsak artefakt ima svoje prednosti pri uporabi v določenih situacijah (Ambler 2002, 45). V nekaterih primerih je bolj efektivno uporabiti diagram kot pa napisati 1024 vrstic kode. Pri modeliranju je pomembno razumevanje kdaj je smotrno uporabiti določen artefakt in kdaj ne (Ambler 2002, 46). Vzporedno modeliranje omogoča simultani zajem informacij več različnih artefaktov (Ambler 2002, 48). V primeru, da informacije postanejo neprimerne za določen artefakt se izvede iteracija do naslednjega artefakta. S tem se informacije prenesejo na drug artefakt in posledično omogočijo napredovanje v procesu. Proces se izvaja inkrementalno, kar predstavlja temelj agilnega modeliranja. Agilno modeliranje si prizadeva k fragmentaciji kompleksnejših nalog v manjše obvladljive entitete, ki se izvajajo v krajših intervalih. Daljši intervali se lahko zgodijo vendar so zgolj izjeme. Daljše kot je obdobje brez povratne informacije večja je nevarnost, da je bilo opravljeno delo zaman (Ambler 2002, 51). Daljša iteracija ne pomeni, da je bilo narejeno več dela. Agilni procesi se bolj fokusirajo na efektivne ure razvoja in ne na njihovo količino (Unger in Novak 2011, 180).

Skupinsko delo definirajo prakse:

* skupinskega modeliranja,
* aktivna participacija vlagateljev,
* kolektivno lastništvo,
* javni prikaz modelov (Ambler 2002, 44–45).

Skupinsko modeliranje omogoča boljše razumevanje idej in ustvarjanje skupne vizije projekta. Prav tako pripomore k izboljšani komunikaciji, izgradnji skupnega besednjaka in povečanju možnosti izvajanja kvalitetnega dela. Z izboljšano komunikacijo poda tudi priložnost prenosa znanja med zaposlenimi (Ambler 2002, 52). Navadno je za uspeh projekta potrebna večja stopnja vključenosti vlagateljev. Aktivna participacija tako spodbuja prisotnost uporabnikov ali vlagateljev na lokaciji z namenom izmenjave informacij glede zahtev in pričakovanj projekta (Ambler 2002, 53). Za kolektivno lastništvo velja, da lahko vsakdo sodeluje pri izgradnji določenega modela, kar predstavlja priložnost večje identifikacije napak (Ambler 2002, 54). Prav v ta namen se uporabljajo table za javni prikaz modelov, kar spodbuja odprto komunikacijo med zaposlenimi (Ambler 2002, 55).

Enostavnost definirajo prakse:

* ustvarjanja preproste vsebine,
* enostavnega prikaza modela,
* uporaba enostavnih orodij (Ambler 2002, 45).

Vsebino je priporočljivo poenostaviti do mere, da še vedno zadovoljuje projektu. To pomeni, da model še vedno zadovoljuje potrebe in ima svoj namen. Poleg tega model ne vsebuje podvojenih informacij in je sestavljen iz karseda nizkega števila elementov, kar vodi v enostaven prikaz (Ambler 2002, 56–57). Priporoča se izogibanje: križanim tranzicijam, zavitim tranzicijam, diagonalnim tranzicijam, različnim velikostim balonov, velikemu številu balonov (ne več kot 7 +/−2) in nepotrebnim podrobnostim (Ambler 2002, 57). Večina modelov lahko narišemo na tablo, kar je potrditev, da je fizična tabla najboljše orodje modeliranja. Uporaba enostavnih orodij deluje, saj je večina diagramov za enkratno uporabo (Ambler 2002, 58).

Validacijo definirajo prakse:

* upoštevanja preverljivosti,
* dokazovanja s kodo (Ambler 2002, 45).

Moderni procesi vključujejo aktivnosti testiranja in zagotavljanje kvalitete čez celoten življenjski cikel. Nekateri modeli celo priporočajo definiranje testnih konceptov pred izgradnjo sistema. Če ne moremo testirati sistema je bolje, da ga ne gradimo (Ambler 2002, 58). K temu stremi praksa upoštevanja preverljivosti. Ob zgrajenem modelu pa je potrebno preveriti ali je mogoča implementacija poslovnega pravila v model, torej ali ga bo mogoče izvajati. To lahko preverimo tudi z uporabo kode. V praksi se priporoča izvajati cikel modeliranja, kodiranja in testiranja (Ambler 2002, 59).

6.2 META MODELIRANJE

V literaturi je definiranih več pristopov meta-modeliranja različnih avtorjev. Med njimi sta dva standardizirana, ki uporabljata meta-modele za definiranje MIPO: ISO 24744:2007[[63]](#footnote-63) in SPEM[[64]](#footnote-64). Meta-model se osredotoča na prikaz vsebine metode in ima številne prednosti:

* Zagotavlja formalne temelje za specifikacijo MIPO,
* zgrajeni procesi so primerljivi na podlagi uporabe okvirja meta modela,
* formalizacija zagotavlja natančne temelje razvojnim orodjem,
* zagotavlja analiziranje konsistence in skladnosti,
* zagotavlja formalno podlago za modifikacije (Engels in Sauer 2010, 412–13).

Konzorcij OMG[[65]](#footnote-65) je v standardu MOF[[66]](#footnote-66) definiral štiri-plastno referenčno arhitekturo meta-modela na podlagi pogosto uporabljenih konceptov (Engels in Sauer 2010, 419). Kot je razvidno iz slike 6.2 se plasti delijo na plast: M0, M1, M2 in M3 (Engels in Sauer 2010, 419–20).

Slika 6.2 temeljna 4-plastna hierarhija meta-modela po standardu MOF  
  
vir: (Engels in Sauer 2010, 420)

**M0** predstavlja plast izvajanja in predstavlja področje uporabe (razvoj programske opreme). V domeni IM plast M0 predstavlja temeljne objekte, ki se proizvajajo tekom življenjskega cikla razvoja programske opreme (Engels in Sauer 2010, 419–20).

**M1** je plast modeliranja, ki predstavlja model MIPO (Engels in Sauer 2010, 420). V tej plasti so definirani konstrukti in procesni modeli (Jablonski 2010, 400). M1 se izražajo v realnosti kot lastne instance v M0 (Engels in Sauer 2010, 420).

**M2** je plast kjer se izvaja meta-modeliranje (Engels in Sauer 2010, 420). Konstrukti so glede na domeno uporabe definirani tukaj. Definirajo jih procesni jeziki na nivoju domenskega[[67]](#footnote-67) in abstraktnega[[68]](#footnote-68) modela. Abstraktni služi za podajanje osnovnih funkcionalnosti jezikov modeliranja kot sta potek in kontrola podatkov, domenski model pa predstavlja dejanski jezik modeliranja. Elemente nivoja M2 prestavljajo primitivne procesne entitete. Definiramo jih s pomočjo generičnih elementov modeliranja (škatle, loki), ki jih podaja plast M3 (Jablonski 2010, 400).

**M3** predstavlja najvišjo plast. Tukaj so definirani meta-meta-modeli (modeli meta-modelov). Namenjeni so za opisovanje meta-modelov v plasti M2 (Engels in Sauer 2010, 420).

Inženiring se dogaja v treh domenah: IMM[[69]](#footnote-69), IM in razvoj programske opreme. Vsaka od domen odgovarja različni stopnji abstrakcije, katere so izražene s hierarhijo plasti prikazani na sliki 6.3. Domene zahtevajo opravljanje različnih nalog za proizvajanje želenih produktov na posameznih nivojih meta-modela. Te naloge se izvajajo z dodelitvijo vlog kot kaže slika 5.5. Inženir meta-metod je odgovoren za definiranje meta-metode za IM na nivoju M2. To predstavlja domeno IMM. Definirano meta-metodo nato inženir metod aplicira v domeno IM z namenom izdelave stabilne MIPO za nivo M1. Izdelano MIPO lahko razvijalci uporabijo v domeni razvoja, ki ga predstavlja nivo M0 (Engels in Sauer 2010, 422).

Slika 6.3 dodeljene vloge za izdelavo delovnih produktov na različnih nivojih hierarhije meta-modela

Vir: (Engels in Sauer 2010, 423)

6.2.1 METODA METAME

Velika pomanjkljivost dosedanjih meta-metod je odsotnost definicij opravil in procesov, ki specificirajo izgradnjo MIPO z uporabo meta-modeliranja. Čeprav večina pristopov podpira nekakšno integracijo produktnih in procesnih fragmentov se le ta dogaja na visokem nivoju abstrakcije. Zaradi tega niso sposobni tvoriti kompleksne vzorce medsebojno povezanih strukturnih in vedenjskih modelov (Engels in Sauer 2010, 413).

Na podlagi temeljev IM, navedenih pomanjkljivosti in standardov meta-modeliranja so strokovnjaki razvili metodo MetaME[[70]](#footnote-70), ki je meta-metoda za razvoj MIPO (Engels in Sauer 2010, 424). Metoda sledi ideji izgradnje novih MIPO pred modifikacijo starih velikih procesov (Steenweg, Kuhrmann, in Méndez Fernández 2014, 14). Za potrebe izgradnje definira jezik za opis metode (produktna perspektiva) in proces, ki se uporablja za razvoj programske opreme (procesna perspektiva) (Engels in Sauer 2010, 424–25). MetaME metoda je nastala z združitvijo produktnega in procesnega modela, ki združujeta njima sorodne perspektive (Engels in Sauer 2010, 425). Inženir mora v produktnem modelu definirati katere artefakte je potrebno ustvariti in kako so ti artefakti med seboj povezani. Medtem mora v procesnem modelu definirati postopek izdelave artefaktov, kaj je potrebno narediti in kdo je za to odgovoren. Produktni model tako razumemo kot model artefaktov, procesni model pa kot model poteka dela in opravil (Engels in Sauer 2010, 413–14). Nato se ta dva modela združita v enoten proces. Ta združitev tvori MIPO, katera služi razvijalcem v praksi.

Slika 6.4 temeljni proces meta-metode za izgradnjo metode inženiringa programske opreme



Vir: (Engels in Sauer 2010, 430)

Slika 6.4 prikazuje temeljni proces meta-metode za izdelavo MIPO. V prvem koraku moramo definirati domeno uporabe metode in njene discipline (Engels in Sauer 2010, 429). Lahko predstavljajo različne nivoje abstrakcije, poglede ali parcialne modele disciplin. Nato je zgrajen model iz konceptov ali temeljnih aktivnosti domene, katere so organizirane na podlagi disciplin (Engels in Sauer 2010, 429-30). Ta model predstavlja produktni model meta-metode. Sledi izbira notacije, ki skrbi za primerno predstavitev konceptov. V tem koraku je potrebno identificirati jezike, podjezike in elemente jezika. Navadno izbiramo med jeziki modeliranja (Engels in Sauer 2010, 430). Naslednji korak predstavlja definiranje tipa artefaktov. V tem koraku se jeziki in njegovi elementi pripišejo konceptom z namenom izražanja njihovih lastnosti. Medtem, ko domenski model predstavlja semantično domeno (pomen) konceptov, jeziki predstavljajo sintaktično domeno (oblika). Hierarhiji domenskega modela konceptov in artefaktov morata biti kompatibilna. Temu sledi definiranje procesnega modela pri katerem je potrebno definirati aktivnosti, ki so potrebne za izpolnitev nalog. Procesni model je strukturiran iz aktivnosti, mejnikov in elementov nadzora toka. Zadnji korak predstavlja izbira orodij, tehnik in pripomočkov skupaj s koncepti uporabe, ki so potrebni za usmerjanje in poenostavitev izgradnje artefaktov (Engels in Sauer 2010, 431).

6.3 SESTAVA METODE

Ta poglavje zaključujemo z izbiro in inženiringom metode, ki jo bomo uporabili pri izgradnji procesa oz. širše MIPO. Sedaj se podajamo v vlogo inženirja metode, ki ga predstavlja prva tretjina v sliki 6.3. Metodo bomo sestavili iz AM, MetaME metod, ki so bile predstavljene. Ko kaže slika 6.6 bo metoda MetaME predstavljala hrbtenico osnovnega procesnega toka razvoja video iger, AM pa bo s svojimi praksami služila za podporo iterativnim, inkrementalnim in ostalim konceptom agilnih praks.

Slika 6.6 izboljšan temeljni proces meta-metode z uporabo Agilnega modeliranja



Vir: (Ambler 2002, 10)

7 IZGRADNJA PROCESA

Izdelava procesa pomeni definiranje procesnega modela, ki predstavlja zgradbo sistema (Jablonski 2010, 394). Potrebno je definirati zbirko elementarnih primitivnih procesnih entitet iz katerih je mogoče zgraditi prilagojen procesni model, ki omogoča apliciranje v različne domene (Jablonski 2010, 396). Našo domeno predstavlja razvoj video iger, ki je hkrati prvi korak v procesu izgradnje MIPO po MetaME.

7.1 DEFINIRANJE DOMENE IN DISCIPLIN

Domeno predstavlja razvoj video iger katere discipline so povezane z vlogami in odgovornostmi. Discipline so lahko predstavljene tudi z različnimi nivoji abstrakcije, ki lahko predstavljajo korake ali faze procesa (zahteve, analiza, razvoj, itd.) (Engels in Sauer 2010, 429–30). Osnovne faze pri razvoju programske opreme predstavljajo analiza, načrtovanje, kodiranje in testiranje (Ramadan in Widyani 2013, 95). Faze, ki jih posamezni avtorji navajajo so prikazane v PRILOGA B. Na podlagi literature smo konsolidirali faze razvoja iger in jih kronološko razvrstili: formalizacija problema, razvoj koncepta, pred produkcija, produkcija, testiranje, post produkcija.

7.2 IZGRADNJA DOMENSKEGA MODELA

Resne igre potrebujejo posebno fazo v razvoju. Aslan in Balci (2015) definirata formulacijo problema kot prvo fazo pri razvoju resnih iger. Ta faza definira probleme različnih domen in načine reševanja, ki privedejo do efektivnega učenja na podlagi iger.

Za drugo fazo definirata generacijo idej (Aslan in Balci 2015, 311), ki pa predstavlja prvo fazo po Bates (2004) in Novak (2012). Widyani (2013) jo imenuje iniciacija. V tej fazi je predstavljena ideja igre v pisni obliki (Novak 2012, 352). Čeprav Unger in Novak (2011) označujeta pred produkcijo za prvo fazo pa navajata, da se začne po uspešno sprejetem konceptu. Zato bomo pred produkcijo kronološko postavili za fazo razvoja koncepta.

Pred produkcija je faza v kateri se sprejme odločitev kakšna bo igra in kako bo izdelana (Unger in Novak 2011, 176). Predstavlja fazo načrtovanja (Novak 2012, 353). Vrhunec pred produkcije predstavlja dokaz koncepta, ki ga predstavlja prototip. Namen je dokazati zmožnost premagovanja tehničnih zahtevnosti (Bates 2004, 88). Po potrditvi prototipa se lahko začne produkcija, ki predstavlja fazo razvoja igre (Novak 2012, 358).

Produkcija je kompleksna in časovno potratna. Za uspešno izvajanje faze je priporočena fragmentacija opravil v manjše naloge, katerim se strogo sledi. Priporočen je tedenski pregled napredka (Bates 2004, 212). Čeprav se testiranje v smislu produkcijskega testiranja dogaja tudi v fazi produkcije se v industriji iger ločijo posebne faze testiranja. Ko je razvoj igre tako daleč, da je mogoče igro igrati od začetka do zaključka se začnejo faze testiranja (Novak 2012, 359).

Testiranje se deli na Alfa in Beta faze. V razvoju mobilnih iger sta fazi navadno združeni (Unger in Novak 2011, 176). Alfa faza predstavlja točko na kateri je mogoče testirati celotno igranje. Testna ekipa v tej fazi testira vse module igre, zapiše defekte v bazo in ustvari testni načrt. Ob končani alfa fazi morajo biti narejeni naslednji elementi:

* celotno igranje,
* primarni jezikovni paket,
* osnovni uporabniški vmesnik z dokumentacijo,
* kompatibilnost z definiranimi strojnimi in programskimi konfiguracijami,
* test minimalnih zahtev,
* test kompatibilnosti vmesnikov (ali vsaj večino),
* nadomestni primeri zvoka in umetnin,
* test več igralske funkcionalnosti (če je implementirana),
* osnutek igralnega priročnika (Novak 2012, 359).

Beta faza je namenjena popravilu defektov in implementaciji vseh sredstev (zvok, umetnine) v igro. S tem se popolnoma konča produkcijski proces. Namen faze je stabilizacija projekta in eliminacija večine ali vseh defektov pred izdajo. Po končani beta fazi morajo biti končani naslednji elementi:

* koda,
* vsebina,
* jezikovni paket,
* igralna navigacija,
* uporabniški vmesnik,
* programska in strojna kompatibilnost,
* zvok in umetnine,
* vodnik igranja. (Novak 2012, 360).

Beta faza se deli tudi na zaprto in odprto beta fazo. Za zaprto beta fazo velja privatno testiranje pri kateri se razvoj fokusira na poliranje igre. Odprta faza pa je navadno uporabljena pri igrah ki vsebujejo spletne komponente. V tej fazi se preveri delovanje strežnikov, uravnoteži igranje in identificira defekte (Levy in Novak 2009, 54–55).

Ko igra prestane beta fazo sledi njena izdaja. V tej fazi ravnateljstvo opravi še en pregled produkta in pridobi seznam prisotnih defektov. Po pregledu sledi izdaja produkta na trg. V industriji iger je za to fazo pogost izraz ''zlata'' (Novak 2012, 362).

Po fazi izdaje sledi post produkcijska faza. Faza predstavlja nadaljevalno produkcijo, ki jo Fullerton (2008) imenuje tudi vzdrževanje. Slika 7.1 tako predstavlja domenski model, ki je zgrajen iz razvojnih faz in hkrati predstavlja podlago za procesni model.

Slika 7.1 domenski model na podlagi razvojnih faz



7.3 IZBIRA NOTACIJ

V tem koraku bomo definirali jezik uporabe, ki nam bo podal notacije. Jezik, ki ga bomo uporabili predstavlja UML z notacijami v tabeli 7.2. Z notacijami bomo povezali stanja in artefakte v procesni model.

Tabela 7.2 prikaz izbranih notacij UML jezika za definiranje procesa

|  |  |  |
| --- | --- | --- |
| NAZIV NOTACIJE | NAMEN | NOTACIJA |
| ZAČETNO STANJE | STANJE, KI NAKAZUJE ZAČETEK AKTIVNOSTI |  |
| IZBIRA | STANJE, KI OMOGOČA IZBIRO |  |
| TRANZICIJA | PRIKAZUJE SMER TRANZICIJE |  |
| STANJEs | PREDSTAVLJA STANJE |  |
| VILICE ALI ZDRUŽITEV | DELI DEJAVNOSTI NA VEČ TRANZICIJ ALI ZDRUŽUJE V ENO |  |

Vir: (Rumbaugh, Jacobson, in Booch 2004, 40–90)

7.4 DEFINIRANJE ARTEFAKTOV

Kot smo spoznali v petem poglavju artefakte sestavljajo produkti, dokumenti in diagrami. Artefakte smo razvrstili na podlagi literature in so zbrani v PRILOGI C.

Če se razvoj nameni izdelati učno igro mora skozi fazo, formalizacije problema. V tej fazi se kreira dokument specifikacije učnega problema (Aslan in Balci 2015, 310).

Nato sledi faza razvoja koncepta katere namen je definiranje osrednje funkcionalnosti igre, predstavitev grafičnega izgleda in zgodbe. Dokumente, ki jih razvoj koncepta proizvede so: višji koncept, predlog igre (''pitch doc'') in koncept (Bates 2004, 203). Vsak tip dokumenta, ki predstavlja koncept ima svoj namen, zato ni potrebno, da uporabimo vseh. Višji koncept služi za hitro predstavitev igre, ki bi denimo trajala do 30 sekund. Predlog igre je daljši dokument, ki služi za iskanje investitorjev. Tega Novak (2012) imenuje koncept. Pomaga ravnateljstvu oceniti izvedljivost ideje (Novak 2012, 382). Naslednji dokument Bates (2004) imenuje koncept in je obsežnejši, tudi do 20 strani in zajema opis žanra, igranja, funkcionalnosti, zgodbe, ciljne publike, izbiro platforme, pričakovane vire, analizo konkurenčnosti, analizo tveganja in povzetek (Bates 2004, 204). Novak (2012) ga imenuje predlog igre. Namen predloga igre je predstavitev igre podjetju ali partnerju. Ta dokument se bolje posveti zgodbi in opisu likov (Novak 2012, 387). Zaradi zagotavljanja konsistence bomo razvoj dokumenta koncepta razdelili v tri faze. V fazo izdelave inicialne ideje (višjega koncepta), kateremu sledi dokument koncepta igre (pitch) in izdelava predloga igre.

Predlog igre podpira prvi prototip[[71]](#footnote-71) igre. IGDA[[72]](#footnote-72) ga imenuje demo in predstavlja najpomembnejši material, ki ga lahko proizvedemo v tej fazi (Fullerton 2014, 486). Velja za dokaz delovanja koncepta in zmožnost premagovanja tehničnih zahtevnosti (Bartle 2003, 88). Če je dokaz koncepta uspešen se začne faza načrtovanja ali pred produkcija (Novak 2012, 353). Cilj je izdelava načrta igre, umetniške biblije, definiranje produkcijske poti in stvaritev projektnega načrta. Načrt igre je najobsežnejši dokument v razvoju igre. Novak (2012) navaja, da se ta dokument konstantno posodablja. Schell (2008) pa dodaja, da se občasno posodablja in je po navadi neurejen. Na polovici projekta se ga navadno opusti, saj vsebuje igra sama vse pomembne podrobnosti (Schell 2008, 382). Na podlagi načrta igre tehnična ekipa izdela tehnični dokument. Ta dokument predstavlja produkcijsko pot, ki vzpostavi načrt kako se bo razvoj premaknil od koncepta do programske opreme (Novak 2012, 393). Nato sledi izdelava načrta projekta, ki se začne izdelovati po definiranju grobih nalog, ki jih podaja tehnični dokument. Ta dokument vsebuje načrt porabe virov, proračun, časovni načrt, in mejnike za pomoč pri spremljanju napredovanja projekta (Novak 2012, 394).

Izdelava umetnikove biblije se začne v pred produkciji. Služi za definiranje umetniškega sloga igre (Bates 2004, 208). Ker je odvisna od žanra se začne po načrtu igre (Novak 2012, 393). Umetnikova biblija se navadno gradi iterativno tudi v času produkcije, ker se je v fazi pred produkcije še ne potrebuje (Bartle 2003, 87-88).

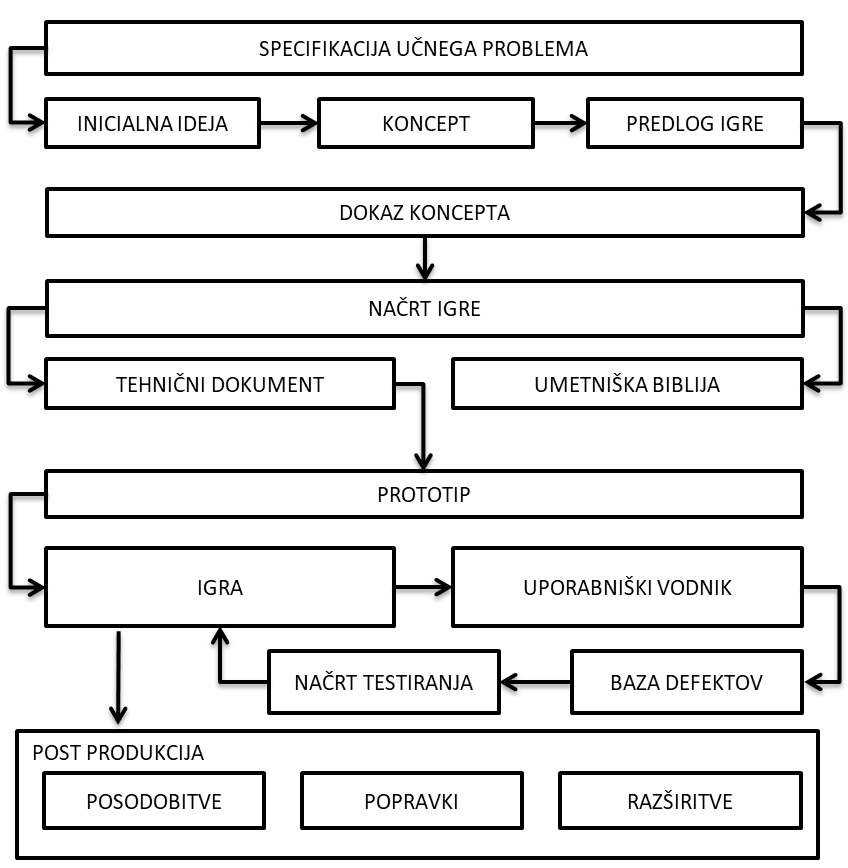
Zadnje dejanje pred produkcijske faze predstavlja izdelava prototipa (Bates 2004, 207–8). Ta artefakt je lahko analogne ali digitalne oblike. Pred izdelavo digitalnega je dobra praksa izdelava enostavnega analognega prototipa katerega namen je zagotoviti, da je igralna mehanika zabavna in prepričljiva. Končan prototip je lahko dokaz delovanja produkcijske poti od ideje do realizacije (Novak 2012, 354). Kot navajata Levy in Novak (2009) se lahko produkcija začne z izdelavo prototipa, ki je groba prestava igre in vsebuje primer igranja.

Nato sledi faza produkcije. V času produkcije se proizvajata dva artefakta. Sama igra ali njen fragment (prototip) in uporabniški vodnik. V praksi je priporočljivo sprotno dopolnjevanje priročnika, vzporedno z razvojem. Vodnik vsebuje razlago igre, vodnika po namestitvi, hiter začetek in detajlno razlago funkcionalnosti uporabniškega vmesnika (Rucker 2002, 6).

Produkciji sledi faza testiranja. Ta faza predstavlja zagotavljanje kakovosti[[73]](#footnote-73) in skrbi, da igra zadovoljuje vse potrebe preden se izda na trg (Levy in Novak 2009, 57). V tem koraku se izdelata podatkovna baza defektov in načrt testiranja. Po testiranju sledi faza post produkcije. V tej fazi se kot artefakti pojavijo popravki, posodobitve in razširitve. V tej fazi so lahko izdane različne verzije programa, ki služijo kot izboljšave. Te verzije so brezplačne in so ustvarjene z apliciranjem popravkov na originalno različico izdaje. Popravki, so lahko aplicirani tudi z namenom reševanja ostalih produkcijskih defektov. Posodobitve izboljšujejo originalno izdajo igre. Te so večinoma ustvarjene z namenom podaljšanja življenjske dobe igre. Zadnji artefakt v razvoju predstavljajo razširitve. Včasih delujejo kot samostojne igre ali pa potrebujejo originalno igro za delovanje (Novak 2012, 365).

Slika 7.3 spodaj prikazuje vse dokumente, diagrame ali produkte, ki predstavljajo model artefaktov, ki je ustrezno kronološko razvrščen. Podjetja imajo lahko različne standarde dokumentacije. Nekatera ustvarijo več dokumentov, druga manj. Navadno je dokumentacija obratno proporcionalna stopnji zaupanja založnika in izkušenosti razvojne ekipe. Ne glede na izkušnje ekipe, dobra dokumentacija pripomore k boljšem razvoju (III 2004, 319).

Slika 7.3 artefakti pri razvoju video iger



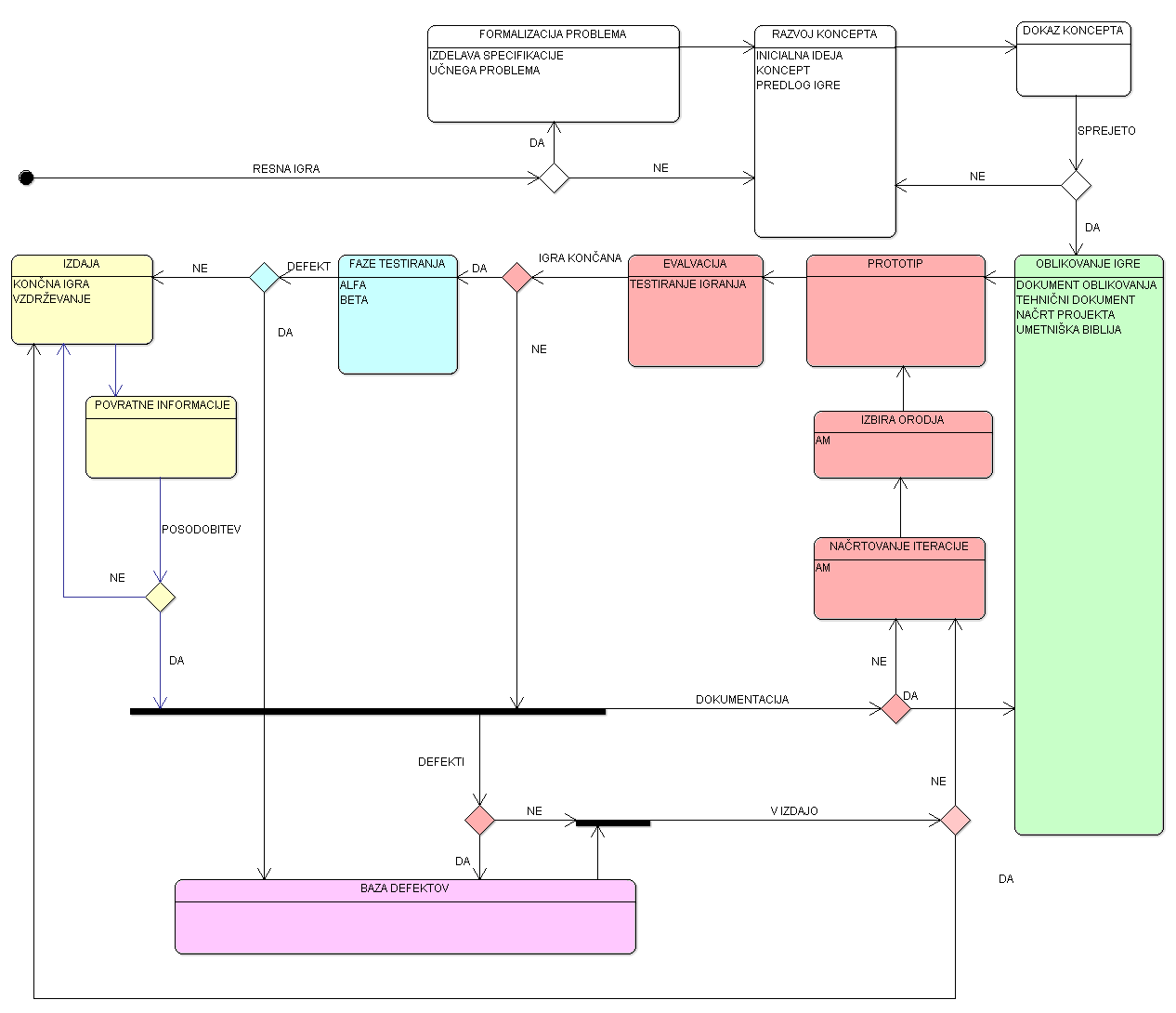
7.5 DEFINIRANJE PROCESNEGA MODELA

Slika 7.4 definira procesni model za video igre. Bela stanja predstavljajo fazo razvoja koncepta, ki se konča z dokazovanjem koncepta. Slednjega predstavlja prototip in velja za poenostavljeno različico igre. Oblikovalci iger jih uporabljajo za testiranje funkcionalnosti (Adams 2013, 49). Uporaben je tudi za dokazovanje izvedljivosti idej, saj predstavlja delujoči model ideje. Z njim si omogočimo formalizacijo ideje in izolacijo problemov (Fullerton 2014, 197). Prvi prototip, ki služi za dokazovanje izvedljivosti ideje je prvi mejnik s katerim se konča prva faza razvoja. V tem procesnem modelu se prototip pojavi že zgodaj v konceptualni fazi, ki dokazuje delovanje ideje preden se začne dolga faza načrtovanja. Ta lahko traja več mesecev (Adams 2013, 49).

Proces se nadaljuje v zeleno stanje, ki predstavlja pred produkcijo. Je proces, ki si ga je industrija iger sposodila od filmske industrije. Ta se navadno konča, ko je investitor že videl igralno različico igre in je zadovoljen z ekipo in delom. Če projektu poda zeleno luč se začne produkcijska faza (Adams 2013, 46).

Produkcijska faza se odvija med fazami načrtovanja, prototipa in evaluacije (Novak 2012, 367). Fullerton (2014) jo opisuje kot iterativni proces med testiranjem igranja, evalvacije in revidiranja. V našem modelu se testiranje igranja izvaja v stanju evalvacije. Po načrtovanju se izvede izgradnja prototipa. V fazi evalvacije ekipa igro testira in se odloči ali se vrne v fazo načrtovanja in nadgradi prototip. Ta proces se izvaja dokler igra ne predstavlja več prototipa ampak končno igro. Vsaka iteracija mora imeti svoj cikel razvoja skupaj z načrtovanjem potreb, artefaktov in urnika (Novak 2012, 367), zato ima model na sliki 7.4 v produkcijski fazi dva stanja (načrtovanje iteracij, izbire orodij), ki vsebujeta prakse in orodja, ki podpirajo te potrebe. Stanja produkcije so označena z rdečo barvo in se odvijajo iterativno dokler igra ni končna. Vzporedno se v produkciji odvijata tudi procesa posodabljanja baze defektov in dokumentacije. Baza defektov je označena z vijolično barvo in predstavlja aktivnost, ki se izvaja v več fazah (produkcija, testiranje, post produkcija).

Fazo testiranja predstavljajo modra stanja. V fazo testiranja preidemo, ko igra predstavlja končni izdelek ali če nas na to obvezujejo pogodbene obveznosti s strani investitorja. Ta faza se deli na alfa in beta fazo testiranja. Alfa stremi k zaklepanju funkcionalnosti medtem, ko se beta faza osredotoča na poliranje igre in reševanje defektov (Levy in Novak 2009, 52–53). V obeh faza se proces nadaljuje iterativno dokler obstajajo defekti ali dokler ti niso označeni za nepomembne (WNF[[74]](#footnote-74), NAB[[75]](#footnote-75)) (Levy in Novak 2009, 103), zato lahko v procesu pridemo do izdaje tudi preko baze defektov. Izdaja predstavlja zadnjo fazo v procesu in je v modelu označena z rumeno barvo. Izdelava popravkov po končani igri je skoraj neizogibno. Vzrok ne leži nujno v rani izdaji temveč v tisoče različnih strojnih konfiguracijah katere je nemogoče v celoti predvideti in testirati. Prav tako kot popravki predstavljajo posodobitve majhne projekte, ki zahtevajo načrtovanje mejnikov, testiranja in druge elemente dobrih praks (Bates 2004, 216). Procesni model to zagotavlja, saj se ob potrebi po posodobitvah (nadgradnje, popravki, razširitve) ponovno začne iterativni proces razvoja. Proces razvoja igre se tako nikoli ne konča. Razvijalci kljub izdanem produktu na podlagi povratnih informacij objavljajo popravke in včasih celo dodajo funkcionalnosti (Fullerton 2014, 421).

7.4 Procesni model za video igre  


7.5 IZBIRA ORODIJ

Izbira se odvija v času produkcije ali post produkcije (posodobitve) pred vsakim začetkom iteracije razvoja. Ta korak omogočata agilno modeliranje in modelno usmerjen inženiring v stanjih načrtovanja iteracije in izbire orodij. Orodja niso determinirana vendar se izbirajo po potrebi projekta ali same naloge. Tukaj se tudi izrazi v praksi, da se z aplikacijo AM v proces modelira več kot prej. V vsakem koraku aplikacije AM je cilj pridobiti globlje razumevanje enega ali več perspektiv sistema in s tem prakso, ki nam ustreza (Ambler 2002, 9). AM nam ponuja različne prakse, ki so se razvile na podlagi osnovnih kategorij (pog. 6.1) znotraj posameznih agilnih procesnih modelov. Nekaj vidnejših praks posameznih procesnih modelov je predstavljenih v PRILOGA F. Te prakse nato uporabimo za podporo našemu temeljnemu procesu v kombinaciji s popolnim agilnim procesom (sl. 6.6) (Ambler 2002, 10).

V razvoju video iger obstajajo tudi neke osrednje usmeritve, ki omogočajo uspeh projektov. Potrebno je zagotoviti: načrt projekta, razumljiv tehnični in časovni načrt, razumevanje tveganj in znanje ponastavitve projekta v težavah (Bates 2004, 219), ki predstavljajo bolj tehnične usmeritve. Ne tehnične predstavljajo: vzdrževanje komunikacije (z ekipo in vodstvom), sledenje stroškom, vzdrževanje ekipnega duha in identitete (Bates 2004, 213). Vse te usmeritve lahko zadovoljimo z izbiro pravega orodja ali več njih. Če pa se orodje ne izkaže za pravo v naslednji iteraciji izberemo drugega.

Pri načrtovanju razvoja je priporočljiva fragmentacija večjih nalog v manjše obvladljive naloge (Bates 2004, 212). Za takšno opravilo je priporočljiva uporaba Scrum prakse definiranja zgodb. Kot smo spoznali pa je Scrum le orodje za upravljanje in ne vsebuje tehničnih aspektov vodenja projekta. Zato so najverjetneje podjetja Scrum metodologijo posvojila v različnih oblikah. Dr. Lennart E. Nacke za Novak (2012) navaja:

*»V naši ekipi smo dodali samolepilne liste na tablo za spremljanje nalog, kar omogoča izogibanje presežku poročanja in vizualno predstavo napredka posameznikov ali njihovih težav vsem v razvoju.« Dr. Lennart E. Nacke.*

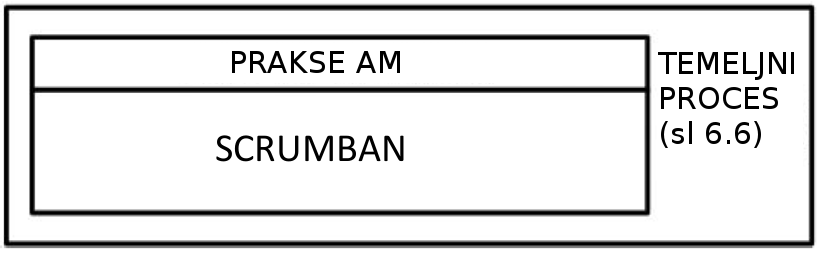
To nakazuje na uporabo hibrida Scrum in Kanban metodologije, ki jo predstavita Yilmaz in O'Connor. Imenovana Scrumban, omogoča, da so deležniki razvojnega procesa socialno povezani kar omogoča motivacijo v efektivno uporabo njihovih spretnosti (Yilmaz in O’Connor 2016, 237). Kanban dimenzija metodologije omogoča, da imamo v vsakem trenutku znanje koliko dela je v delovnem toku in na podlagi obremenitve lahko izračunamo stroške in napredek projekta, medtem, ko Scrum dimenzija omogoča temeljito vodenje projekta in fragmentacijo celotnega projekta na obvladujoče naloge z uporabo zgodb.

Načrtovanje lahko podpremo tudi z uporabo digitalnih programskih paketov. Ravnateljstvo že dolgo uporablja takšna orodja za administracijo individualnih ali skupine projektov. Orodja kot so: Scrumwise, Kanbanery, Yodiz, ZenHub, Leankit, Jira, Trello, so idealna za spremljanje nalog, saj omogočajo kreiranje Kanban tabel. Poleg tega omogočajo tudi komunikacijo med ekipami in integracijo z rešitvami za spremljanje izvorne kode kot je GitHub[[76]](#footnote-76). Poleg teh poznamo še Fat Panda, Pivotal Tracker, Active Collab, HacknPlan, Asana in Slack. Orodja lahko omogočajo tudi definiranje t.i. Gantt[[77]](#footnote-77) ali PERT[[78]](#footnote-78) grafov, ki zadostijo definiranjem časovnih načrtov.

Novi paketi takšnih orodij prinašajo prednosti, saj omogočajo upravljanje s tveganji, upravljanje z najboljšimi praksami, e-poštne notifikacije in kolaboracije. Potencialni kupci takšnih rešitev potrebujejo način s katerim lahko izberejo ustrezen programski paket. Za lažjo odločitev lahko uporabimo analitični hierarhični proces[[79]](#footnote-79), ki omogoča izbiro paketa na podlagi teorije izbire po Saatyu. Primarna naloga postopka je kvantifikacija relativnih prioritet za podano zbirko alternativ (PRILOGA E) na podlagi sodb, ki jih poda odločevalec (Ahmad in Laplante 2006, 76). Kriteriji se lahko spreminjajo in predlagani predstavljajo le ilustracijo procesa izbora. Kriterije je potrebno definirati na podlagi hierarhije, ki predstavlja za nas najbolj primerno orodje (Ahmad in Laplante 2006, 81).

V času razvoja je nevarnost, da projekt zaradi količine defektov postane neobvladljiv. Zavoljo tega obstajajo rešitve katere omogočajo spremljanje različic izvorne kode in takojšnjo ponastavitev na delujočo različico. Programska oprema kot so igre izstopajo predvsem zaradi količine delov iz katerih so sestavljene. Celo enostavna igra lahko vsebuje tisoče izvornih datotek kode, zvoka ali slik. Zato se priporoča izbira rešitve, ki omogoča nadzor izvorne kode (McShaffry in Graham 2012, 111). Komercialne rešitve predstavljajo SourceSafe, Perforce in AlienBrain, odprtokodne pa Subversion, TortoiseSVN in Git (McShaffry in Graham 2012, 15–114).

Slika 7.5 najvišja abstrakcija procesnega modela



Vir: lasten

V našem primeru predlagamo uporabo Scrumban procesa kot pomožnega orodja osnovnemu procesu v kombinaciji z agilnimi praksami. Scrumban izboljša Kanbanov WiP z vpeljavo sistema vlečenja, ki omeji količino dela v teku (Yilmaz in O’Connor 2016, 242). Obremenitev razvoja je tako minimalna in dnevnik zaostankov fiksen. Ocena hitrosti dela tako ni odvisna od števila zgodb, ki jih razvoj proizvede ampak od dnevnika zaostankov. Prav tako model posodobi dnevne sestanke, katere nadgradi v pol strukturirane intervjuje. Posledično ti sestanki omogočajo ustvarjanje sistematičnih in primerljivih informacij (Yilmaz in O’Connor 2016, 242). V realnosti veliko razvojnih ekip zgreši pri napovedi trajanja projekta. Zavoljo tega se proti koncu produkcijskega cikla velikokrat pojavi čas krize[[80]](#footnote-80) (Novak 2012, 358). Z uporabo tega procesnega modela pridobimo na reševanju težav slabih časovnih napovedi razvoja. Poleg tega predlagamo uporabo XP prakse konstantno prisotnega vlagatelja in fiksnega 40 urnega tedenskega delavnika, ki še podpre reševanje kriznega časa in onemogoča preobremenitev zaposlenih.

1. Obstaja veliko nomenklatur: e-učenje, avtentični učni proces, igre z alternativnim namenom, sintetična učna okolja in zabavno izobraževanje (edutainment). [↑](#footnote-ref-1)
2. Virtualna resničnost. Danes najkvalitetnejši produkti: Oculus Rift, Gear VR, HTC Vive, PlayStation VR. [↑](#footnote-ref-2)
3. Konsenz. Igralci lahko pri testiranju igranja navadno v beta fazi podajo svoja mnenja in vplivajo na igro. [↑](#footnote-ref-3)
4. Zrelostni model za učne igre poda tudi. Aslan, Serdar. 2016. „Digital Educational Games: Methodologies for Development and Software Quality“. Imenuje ga IDEALLY - dIgital eDucational gamE softAre quaLity evaLuation methodologY. [↑](#footnote-ref-4)
5. DGMM - Digital Game Maturity Model. [↑](#footnote-ref-5)
6. (angl). Thesaurus. Zbirka sopomenk (sinonimov). [↑](#footnote-ref-6)
7. Gams, Matjaž, in Boštjan Kaluža. 2013. *Računalniški slovarček*. Kamnik: Amebis. [↑](#footnote-ref-7)
8. Ime modela izvira iz košarkarskega pojma, ki pomeni zabijanje žoge skozi obroč. [↑](#footnote-ref-8)
9. Workflow. [↑](#footnote-ref-9)
10. Uporablja se tudi izraz modeliranje. [↑](#footnote-ref-10)
11. Commercial off-the-shelf. [↑](#footnote-ref-11)
12. Software reuse; je uporaba obstoječe programske opreme ali spoznanj v namen izgradnje nove programske opreme (Frakes in Kang 2005, 529). [↑](#footnote-ref-12)
13. Procesni model je vpeljalo in ga uporablja podjetje IBM. [↑](#footnote-ref-13)
14. Computer Aided Software Engineering – CASE. [↑](#footnote-ref-14)
15. Forth-generation programing language – 4GL. Danes (2017) v uporabi že 5GL. [↑](#footnote-ref-15)
16. Joint Application Development – JAD. [↑](#footnote-ref-16)
17. Jacobson, Ivar, Grady Booch, in James Rumbaugh. 1999. *The unified software development process*. Reading, Mass.: Addison-Wesley. [↑](#footnote-ref-17)
18. UML – Unified modeling language. V delu ga uporabljamo za grafično definiranje procesnih modelov. [↑](#footnote-ref-18)
19. Personal Software Process – PSP. [↑](#footnote-ref-19)
20. Postmortem. [↑](#footnote-ref-20)
21. (angl.) Team Software Process – TSP. [↑](#footnote-ref-21)
22. Release Candidate-RC, Release to Manufacturing (RTM). [↑](#footnote-ref-22)
23. Frederick P. Brooks, Jr. *The Mythical Man-Month*. 1995 [1975]. [↑](#footnote-ref-23)
24. Uporablja se tehnika stimulacije dražljaja (angl.) Stimulus-Response scenario technique – SRS. [↑](#footnote-ref-24)
25. Index score (story point) po Peters (2008) in cost po Pressman in Maxim (2014) . [↑](#footnote-ref-25)
26. (angl.) Design to cost. [↑](#footnote-ref-26)
27. Keep it simple. [↑](#footnote-ref-27)
28. Class responsibility collaborator. Orodje za viharjenje možganov. [↑](#footnote-ref-28)
29. Spike solution. [↑](#footnote-ref-29)
30. Metoda vzpostavitve igre v rugbyu pri kateri se soigralci držijo skupaj in poskušajo pridobiti posest nad žogo s potiskanjem nasprotnikov z rameni v enaki formaciji. [↑](#footnote-ref-30)
31. Backlog. [↑](#footnote-ref-31)
32. Dynamic System Development Method. DSDM konzorcij je skrbnik metode. [↑](#footnote-ref-32)
33. Cone of silence. Določen čas, da se izkušeni lahko posvetijo svojim nalogam. [↑](#footnote-ref-33)
34. Družina operacijskih sistemov podobni Unixu. [↑](#footnote-ref-34)
35. Just in time – JIT. [↑](#footnote-ref-35)
36. Feature-driven development – FDD. [↑](#footnote-ref-36)
37. Configuration Management – CM. Sistem (orodje), ki skrbi za konsistenco produkta v razvoju na podlagi zahtev in načrtov. Najbolj znana orodja; CFEngine, Puppet, SaltStack, Docker, PowerShell DSC, Git. [↑](#footnote-ref-37)
38. Japonska beseda. Pomeni znak, signalno karto ali veliko vizualno tablo. [↑](#footnote-ref-38)
39. Lead Time. [↑](#footnote-ref-39)
40. Work in Progress – WiP. [↑](#footnote-ref-40)
41. Delivery Rate. [↑](#footnote-ref-41)
42. Little's Law. [↑](#footnote-ref-42)
43. Throughtput. [↑](#footnote-ref-43)
44. Test-driven development . [↑](#footnote-ref-44)
45. Disciplined Agile Delivery. [↑](#footnote-ref-45)
46. Je metodologija za proizvajanje produktov in ustvarjanje organizacij, ki se usmerja na krajše življenjske cikle v kombinaciji z eksperimentiranjem poslovnih hipotez, ponavljajočim izdajanjem produktov in empiričnim učenjem. [↑](#footnote-ref-46)
47. Adaptive software development. Knjiga: Adaptive Software Development: A Collaborative Approach To manging Complex Systems, James A. Highsmith, 2000. [↑](#footnote-ref-47)
48. To predstavlja game flow. [↑](#footnote-ref-48)
49. (angl.) User Experiece-UX. [↑](#footnote-ref-49)
50. (angl.) Modding. Razvijalci izdajo orodja za modificiranje igre katere uporabniki manipulirajo za ustvarjanje novih vsebin v igri. Za začetnika moddinga se smatrajo razvijalci Doom. [↑](#footnote-ref-50)
51. Avtor igre Wolfshade MUD, senior oblikovalec na projektu Earth & Beyond, pri Electronic Arts Los Angeles izda: Command & Conquer: Generals – Zero Hour in Lord of the Rings: The Battle for Middle-Earth, pri Powered Games izda: Superme Commander in Demigod. [↑](#footnote-ref-51)
52. Game development lifecycle. [↑](#footnote-ref-52)
53. Game design document. [↑](#footnote-ref-53)
54. diGital educAtional gaMe dEvelopment. [↑](#footnote-ref-54)
55. Digital education game. [↑](#footnote-ref-55)
56. Game Unified Process. [↑](#footnote-ref-56)
57. Game Waterfall Process. [↑](#footnote-ref-57)
58. eXtreme Game Development Process. [↑](#footnote-ref-58)
59. Agile Game Process. [↑](#footnote-ref-59)
60. Tehnical Design Document. [↑](#footnote-ref-60)
61. Gold-plating. [↑](#footnote-ref-61)
62. Institute of Electrical and Electronic Engineers – IEEE. [↑](#footnote-ref-62)
63. SoftwareEngineering – Metamodel for Development Methodologies. [↑](#footnote-ref-63)
64. Software & Systems Process Engineering Meta-Model – SPEM. [↑](#footnote-ref-64)
65. Object Management Group – OMG. [↑](#footnote-ref-65)
66. Meta Object Facility – MOF. [↑](#footnote-ref-66)
67. (angl.) Domain Specific Process Meta Model – DSPMM. [↑](#footnote-ref-67)
68. (angl.) Abstract Process Meta Model – AMM. [↑](#footnote-ref-68)
69. Inženiring meta-metod. [↑](#footnote-ref-69)
70. Meta-Method for Software Engineering Methods. [↑](#footnote-ref-70)
71. (angl.) Proof of concept. [↑](#footnote-ref-71)
72. International Game Development Association. [↑](#footnote-ref-72)
73. QA- Quality assurance. [↑](#footnote-ref-73)
74. Will Not Fix, pomeni, da defekt ni dovolj pomemben. [↑](#footnote-ref-74)
75. Not A Bug, pomeni da ni defekt. [↑](#footnote-ref-75)
76. Git distribuiran sistem za nadzor različice kode. [↑](#footnote-ref-76)
77. Je grafikon, ki predstavlja časovni razpored projekta. [↑](#footnote-ref-77)
78. Program evaluation and review technique, je statistično orodje za analizo nalog, ki so vključene v razvoj. [↑](#footnote-ref-78)
79. (angl.) AHP – Analytical Hierarchy Process. [↑](#footnote-ref-79)
80. (angl.) Crunch time. [↑](#footnote-ref-80)