4 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, kateri 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). Kot tudi ostali intelektualni in kreativni procesi tudi ti slonijo na odločitvah posameznikov. 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). Obstajajo standardni modeli procesov, ki ne predstavljajo končnega procesa razvoja ampak služijo kot abstraktne forme s katerimi lahko opišemo različne pristope razvoja programske opreme. (Sommerville 2010, 29).

4.1 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).

4.1.1 SLAM DUNK MODEL

Slam dunk[[1]](#footnote-1) 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).

4.1.2 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 4.1, 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 4.1 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).

4.2 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[[2]](#footnote-2) , ki odraža medsebojno povezanost procesnih elementov (Pressman in Maxim 2014, 41).

4.2.1 KASKADNI MODEL

Pomenljiv napredek priložnostnemu modelu predstavlja kaskadni model (waterfall model) (Lethbridge in Laganiere 2005, 428). Kaskadni 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 (gl. Sliko 4.2) (Pressman in Maxim 2014, 42).

Slika 4.2. kaskadni model po Pressman in Maxim

  
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). Sommerville (2010) bolj podrobno opiše posamezne korake modela. Osnovni koraki povsem opišejo temelje aktivnosti modela (gl. Sliko 4.3). 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 in kasnejšo implementacijo ob ustreznosti delovanja na podlagi sistemske specifikacije (Sommerville 2010, 31).

Slika 4.3 kaskadni model po Sommerville

  
vir: (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 zaobite. 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).

Zaradi dokumentacije, ki je proizvedena na koncu vsakega koraka je ta model transparenten (Sommerville 2010, 32). Stroga kontrola nad ustreznostjo dokumentacije ima pozitiven vpliv na kvaliteto, zanesljivost in vzdržljivost programske opreme (CMS 2005, 2). Napredek razvoja je merljiv (CMS 2005, 2) in lažje primerljiv s časovnim načrtom (Sommerville 2010, 32). Model je idealen za manj izkušene projektne skupine in manj izkušeno vodstvo (CMS 2005, 2). V primeru dobro definiranih zahtev naročnika, nominalni verjetnosti sprememb (Sommerville 2010, 32), ob pričakovanju večje fluktuacije zaposlenih, večjem, časovno fleksibilnem projektu z daljšim razvojnim ciklom in potrebi po ohranitvi sredstev je ta model primeren za uporabo (CMS 2005, 2).

Slika 4.4 kaskadni model po Lethbride in Laganiere  
  
vir: (Lethbridge in Laganiere 2005, 429)

V literaturi se pojavljajo različne slike, ki ta model opisujejo z različnimi ali več koraki. Za primer so slike 4.2, 4.3 (gl. str. X) in sliko 4.4 (gl. str. X). Kot je razvidno vse slike ustrezajo klasičnemu pogledu inženiringa programske opreme. Kot tak se mora začeti s potrebami, nadaljevati v načrtovanju in končati z zagotavljanjem kakovosti. Takšen pogled ustreza kaskadnemu modelu. Proces specifikacije in pridobivanja informacij je vedno pred načrtovanjem, implementacijo in zagotavljanjem kakovosti. Načrtovanje in implementacija se vedno odvijata po pridobljenih informacijah in zagotavljanje kakovosti je v vseh primerih kaskadnega modela na zadnjem mestu.

4.2.2 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 (gl. Sliko 4.5). 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).

Slika 4.5 v-model po Pressman in Maxim  
  
vir: (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 stane veliko. (Palmquist in dr. 2013, 9).

4.2.3 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).

Slika 4.6 fazni model po Lethbridge in Laganiere



Vir: (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 4.6) je načrtovanje odvisno od originalne specifikacije (Lethbridge in Laganiere 2005, 430).

4.2.5 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 4.6 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 4.6 inkrementalni model po Pressman in Maxim  
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 različne delovne skupine posamično razvijajo dele programske opreme. (Sommerville 2010, 34). Pred začetkom novega inkrementa so tehnične zahteve splošnega, končnega sistema pogosto spregledane (CMS 2005, 5).

4.3 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).

4.3.1 PROTOTIPIRANJE

Paradigma se začne s komunikacijo in se nadaljuje v slogu, ki ga prikazuje slika 4.7 spodaj. 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 4.7 prototipiranje po Pressman in Maxim



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. Lahko je bil uporabljen neprimeren programski jezik, operacijski sistem ali neefektiven algoritem. 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).

4.3.2 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 popolnejše različice. Spiralni model definira razvojna ekipa z umestitvijo aktivnosti v model. Evolucijski proces se začne v središču (sl. 4.8). 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 4.8 spiralni model po Pressman in Maxim

  
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).

4.3.3 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 4.9 vzporedni razvoj po Pressman in Maxim



Vir: (Pressman in Maxim 2014, 51)

Slika 4.9 prikazuje primer pristopa k vzporednemu modeliranju. Aktivnosti so lahko tudi druga opravila, naloge ali aktivnosti (npr.: komuniciranje ali izgradnja). Aktivnost[[3]](#footnote-3) je lahko na različnih stopnjah razvoja kot prikazuje slika 4.9 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 4.10 vzporedni model po Lethbridge in Laganiere

  
vir: (Lethbridge in Laganiere 2005, 432)

Po Lethbrige in Laganiere (2005) gre pri tem modelu gre za princip fragmentiranja dela (sl. 4.10). Vsaka razvojna ekipa deluje na svojem fragmentu navadno po spiralnem ali evolucijskem modelu. Slika 4.10 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 (gl. pog. 4.4.4. Rapidni razvoj).

4.4 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).

4.4.1 KOMPONENTNI MODEL

Model nastavlja proces razvoja programske opreme z že prevedeno programsko opremo. Komercialne komponente (COTS[[4]](#footnote-4)) 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 4.11). Te so lahko v obliki konvencionalnih programskih modulov, objektno-orientiranih razredih ali njihovih paketov. Model vodi k ponovni rabi[[5]](#footnote-5) že uporabljene programske opreme (Pressman in Maxim 2014, 53).

Slika 4.11 komponenti model povzet po Sommerville

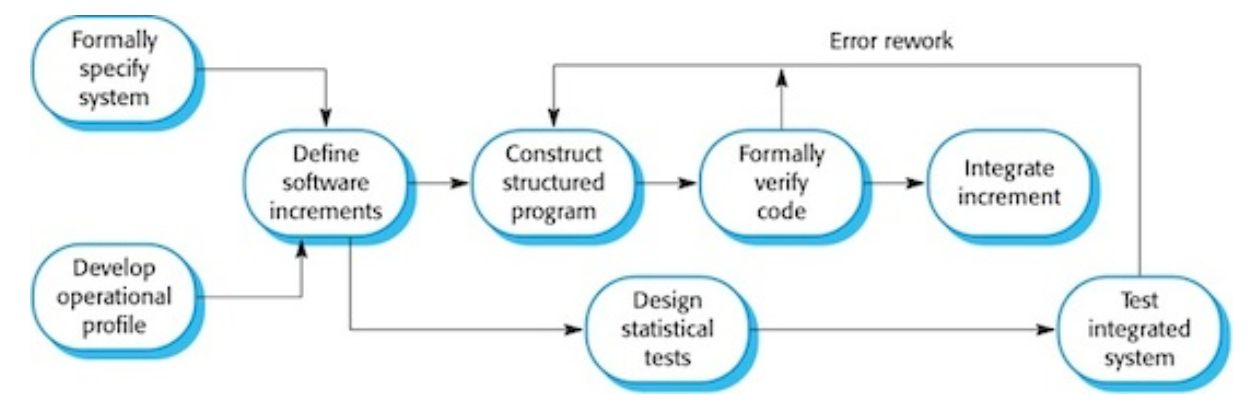
  
vir: (Sommerville 2010, 35)

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).

4.4.2 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[[6]](#footnote-6). 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).

Slika 4.12 formalni model čiste sobe po Pressman in Maxim



Vir: („Cleanroom development“ 2017)

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 jamstvo 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).

4.4.3 ASPEKTNO USMERJEN RAZVOJ

Inženirji kompleksnih rešitev vedno implementirajo določene lokalizirane funkcionalnosti ne glede na to kateri model izberejo. Lokalizirane karakteristike programske opreme se modelirajo kot komponente[[7]](#footnote-7) 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, pravil delovanja, sinhronizacije opravil ali upravljanje pomnilnika. Ko se te pojavljajo v več sistemskih funkcijah ali informacijah za njih uporablja termin presečne zadeve[[8]](#footnote-8). Ta paradigma je relativno nova. Ponuja proces in metodološki pristop k definiranju, načrtovanju in izdelavi aspektov, mehanizmov za lokalizacijo presečnih zadev (Pressman in Maxim 2014, 55).

Ta model še ni dozorel. Ob uporabi največkrat povzame karakteristike evolucijskih ali vzporednih procesnih modelov. Pri evolucijskih so aspekti identificirani pred razvojem. Lastnost sočasnosti vzporednega razvoja pa pripomorejo k razvoju neodvisnih lokaliziranih programskih komponent čeprav imajo aspekti na njih neposreden vpliv. Zato je potrebo inicializirati asinhrono komunikacijo med procesnimi aktivnostmi in že vgrajenimi komponentami (Pressman in Maxim 2014, 55).

4.4.4 RAPIDNI RAZVOJ

Dobra lastnost rapidnega razvoja[[9]](#footnote-9) 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[[10]](#footnote-10), računalniško podprto programsko inženirstvo[[11]](#footnote-11), sisteme za upravljanje podatkovnih baz[[12]](#footnote-12), programske jezike četrte generacije[[13]](#footnote-13), 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[[14]](#footnote-14) (Sabharwal 2009, 19). Modularna fragmentacija omogoča uporabo komponent na podlagi katerih se skrajša časovni okvir razvojnega cikla in stroške razvoja (Sabharwal 2009, 20).

Slika 4.13 rapidni razvoj po Sabharwal



Vir: (Sabharwal 2009, 19)

Model sestoji iz štirih faz (sl. 4.13); definiranja potreb in zahtev s pomočjo evokacijskih tehnik viharjenja možganov[[15]](#footnote-15), 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. Slednje morajo imeti zmožnost fragmentacije v module in uporabne razvojne komponente (Sabharwal 2009, 21).

4.4.5 UNIFICIRAN MODEL

V knjigi *Unified Process*[[17]](#footnote-17)avtorji[[18]](#footnote-18) 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 razvoja. 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[[19]](#footnote-19) in povezanega unificiranega razvojnega procesa programske opreme. V kontrastu s tem modelom drugi procesni modeli predstavljajo enojni pogled na proces. Ta proces navadno opisuje tri perspektive; dinamično, statično in praktično. Pri dinamični prikazuje faze modela skozi čas. Statična predstavlja sprejete procesne aktivnosti, praktična pa predlaga uporabo dobrih praks v času razvoja (Sommerville 2010, 50).

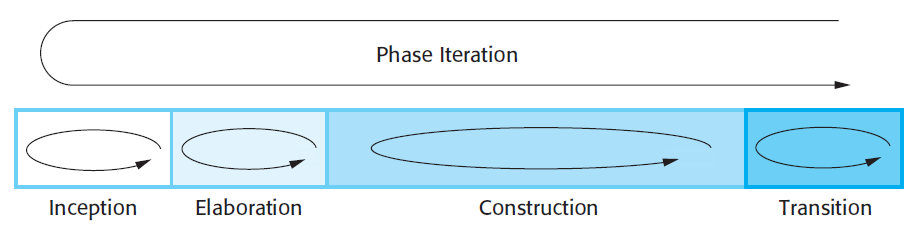
Slika 4.14 Unificiran procesni model po Pressman in Maxim



Vir: (Pressman in Maxim 2014, 57)

Unificiran model ima navadno pet faz (sl. 4.14). 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).

Slika 4.15 iterativni prikaz unificiranega procesa po Sommerville



Vir: (Sommerville 2010, 51)

Iteracija znotraj modela je podprta v obe smeri. Vsaka faza se lahko izvede iterativno z inkrementiranimi rezultati (sl. 4.15). Lahko pa se vse faze izvedejo kot en inkrement (sl. 4.14). 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 4.16 (Sommerville 2010, 51).

Slika 4.16 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).

4.4.6 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[[20]](#footnote-20) 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[[21]](#footnote-21) poročilo (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, prenapihnjenost in neusklajenost zahtev. Pomaga nam, da se učimo iz preteklih uspehov in napak. Cilj tega poročila je vpogled v metode in prakse, ki ovirajo proces in uporabiti analizo za njegovo izboljšavo (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).

4.4.7 EKIPNI MODEL

Ker je razvoj v organizacijah velikokrat situiran okoli ekip je Watts Humphrey razširil spoznanja iz vpeljave osebnega modela in predstavil ekipni[[22]](#footnote-22) 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).

4.4.8 SINHRONIZACIJSKI IN 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[[23]](#footnote-23) 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. 4.17) (Peters 2008, 120).

Slika 4.17 odstotek dovoljenih sprememb glede na izdajo po Peters



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).

4.5 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).

Zaradi težnje po konstantnih spremembah je stabilnost sistema pri tem pristopu ena od težav. Čeprav se sistem stalno testira se prav tako dogajajo spremembe, kar je v nasprotju s priporočili Brooksa in drugih[[24]](#footnote-24). 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). Slika 4.18 spodaj jih bolje opisuje.

Slika 4.18 podrobno opisana načela agilnega pristopa po Sommerville

  
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. Posamezni razvijalci tudi nimajo osebnih sposobnosti za obvladovanje intenzivne vpletenosti in slabo komunicirajo z drugimi ekipami. V projektih z več vlagatelji je nasploh težavno postavljati prioritete spremembam. Navadno vlagatelji nimajo skupnih prioritet (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).

4.5.1 EKSTREMNO PROGRAMIRANJE

Ekstremno programiranje (XP[[25]](#footnote-25)), uporablja paradigmo objektno-orientiranega razvoja in zajema pravila in prakse štirih okvirnih aktivnosti (sl. 4.19): 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).

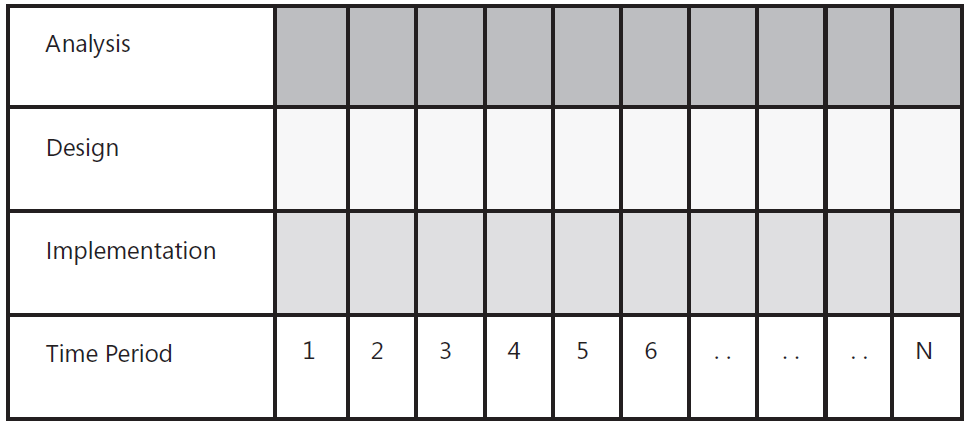
Zgodbe se nahajajo v posameznih časovnih periodah, ki jih predstavljajo stolpci na sliki 4.20. En tak stolpec predstavlja 2-tedenski interval. Sicer lahko uporabljamo daljše vendar v praksi niso priporočljivi. Vsak ta interval predstavlja obseg dela, ki je sestavljen iz zbirke zgodb[[26]](#footnote-26). 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[[27]](#footnote-27) indeksa posameznim zgodbam, ki je relativna glede na njeno težavnost (Peters 2008, 116).

Slika 4.19 prikaz procesa ekstremnega programiranja po Pressman in Maxim



Vir: (Pressman in Maxim 2014, 72)

Slika 4.20 generični prikaz agilnega življenjskega cikla[[28]](#footnote-28) po Peters

  
Vir: (Peters 2008, 116)

Slika 4.21 detajlni prikaz zgradbe pristopa po teoriji Petersa in Pressman in Maxim

  
vir: lasten

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). Na primer;

**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 (sl 4.22). Ta tehnika je podobna inženirskemu principu načrtovanja[[29]](#footnote-29) 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).

Slika 4.22 primer preračuna povprečnega števila točk na 2-tedenski interval po Peters



Vir: (Peters 2008, 117)

Oblikovanje sledi principu KIS[[30]](#footnote-30). Priporoča se enostavnejše oblikovanje pred kompleksnimi. Pristop spodbuja uporabo kartic CRC[[31]](#footnote-31) 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[[32]](#footnote-32). 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). Sprejemna testiranja imenovana tudi testiranja uporabnikov so izvedena na podlagi specifikacije naročnika in se fokusirajo na splošno delovanje funkcionalnosti (Pressman in Maxim 2014, 75).

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).

4.5.2 SCRUM

Scrum[[33]](#footnote-33) 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. 4.23). 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 dokumenti pridobljenih izkušenj (postmortem) na projektu. 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 (sl. 4.23).

Slika 4.23 osnovne faze scrum procesa po Sommerville



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 4.24 podroben prikaz poteka Scrum modela po Pressman in Maxim



Vir: (Pressman in Maxim 2014, 78)

Sprinti 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[[34]](#footnote-34) 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 identifikacija 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 zaščititi razvojno ekipo pred zunanjimi motnjami, (Sommerville 2010, 73) skrbi za posodobitev dnevnika, organizira sestanke, evidentira odločitve in vzdržuje komunikacijo z vodstvom (Sommerville 2010, 74). Cikel se zaključi s predstavitvijo različice naročnikom (Sommerville 2010, 73).

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).

4.5.3 RAZVOJ DINAMIČNIH SISTEMOV

Metoda za razvoj dinamičnih sistemov ali DSDM[[35]](#footnote-35) 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).

4.5.4 CRYSTAL

Kristalno[[36]](#footnote-36) 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. Kristalno čista metodologija je primerna za manjše nekritične projekte medtem, ko lahko kristalno oranžno metodologijo uporabimo pri projetih, 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). 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).

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). 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[[37]](#footnote-37) 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).

4.5.5 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[[38]](#footnote-38). 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 tudi posebno nadarjenim razvijalcem. Veliko jih dela brezplačno in pridobivajo motivacijo z zadovoljstvom uporabnikov (Tsui, Karam, in Bernal 2013, 98).

4.5.6 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[[39]](#footnote-39), 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). Njihov 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 pa 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).

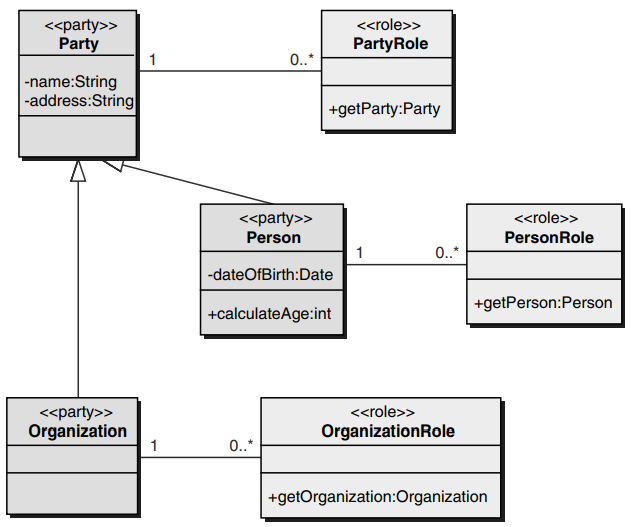
Opaziti je prepričanje, da je pisanje programske opreme podobno izgradnji avtomobila, kar ne upošteva, da je programska oprema nevidna. Potrebne so izčrpne meritve, ki so skladne z organizacijskimi cilji ustanoviteljev Lean metode. Ostale metode zajemajo drugačne vrednote kot Lean, vendar zanemarjajo potrebo po namestitvi preverjene kulture upravljanja za premagovanje nenadzorovane programske opreme in konstantno izboljšanje njenega razvoja. Lean metodologija, ki bi jo lahko implementirali še ne obstaja (Janes in Succi 2014, 130). Medtem, ko se ostale agilne metode nagibajo k agilnosti se ta metoda k efektivnosti (Janes in Succi 2014, 144).

4.5.7 FUNKCIONALNO USMERJEN RAZVOJ

Funkcionalno[[40]](#footnote-40) 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[[41]](#footnote-41) 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). Diagram (sl. 4.25) zagotavlja celosten okvir na katerega dodajamo funkcionalnosti. Pomaga pri razumevanju konceptualne integritete sistema (Palmer in Felsing 2002, 37). Pri modeliranju se uporablja tudi tehnika barv. Modeliranje v barvah uporablja štiri-barvno lestvico za prikazovanje razrednih interakcij (Palmer in Felsing 2002, 38).

Slika 4.25 domenski diagram funkcionalnosti po Palmer in Felsing



Vir: (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[[42]](#footnote-42). 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).

*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).

4.5.8 KANBAN

Kanban[[43]](#footnote-43) 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. 4.26) prikazuje potek sistema skozi različne stopnje procesa od leve proti desni (Anderson in Carmichael 2016, 13).

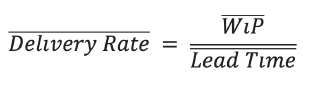
Slika 4.26 tabla poteka dela za Kanban po Anderson in Carmichael



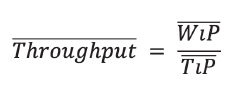
Vir: (Anderson in Carmichael 2016, 13)

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[[44]](#footnote-44). 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 tudi kot delo v teku[[45]](#footnote-45).  
Hitrost s katero se izdajajo postavke imenujemo obseg[[46]](#footnote-46) 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. 4.27). Imenujemo ga tudi mali[[47]](#footnote-47) zakon.

Slika 4.27 povprečni obseg izdaje

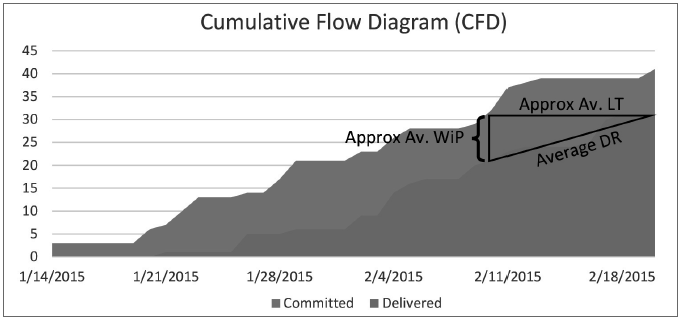
  
vir: (Anderson in Carmichael 2016, 15)

Če bi želeli pridobiti informacijo za druge poteke dela Kanban sistema lahko namesto dobavnega časa uporabimo čas v procesu (TiP – Time in Process) (sl. 4.28). Tedaj dobimo kvocient, ki predstavlja pretočnost[[48]](#footnote-48). 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 4.28 povprečni obseg izdaje  
  
vir: (Anderson in Carmichael 2016, 15)

Mali zakon lahko predstavimo tudi kot kumulativni graf pretočnosti (tab. 4.1), ki prikazuje kumulativno število prispelih in izdanih postavk sistema. Hipotenuza trikotnika na v tabeli predstavlja povprečni obseg izdaje (Anderson in Carmichael 2016, 16).

Tabela 4.1 kumulativni graf pretočnosti po Anderson in Carmichael



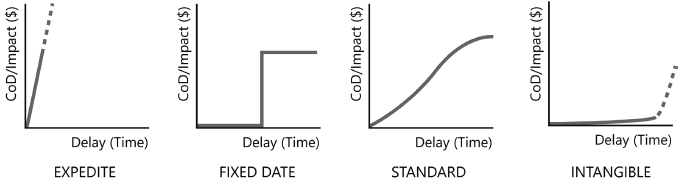
Vir: (Anderson in Carmichael 2016, 16)

Osnovna 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. 4.26), 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. 4.28) (Anderson in Carmichael 2016, 21).

Slika 4.28 stroški zamude za Kanban po Anderson in Carmichael

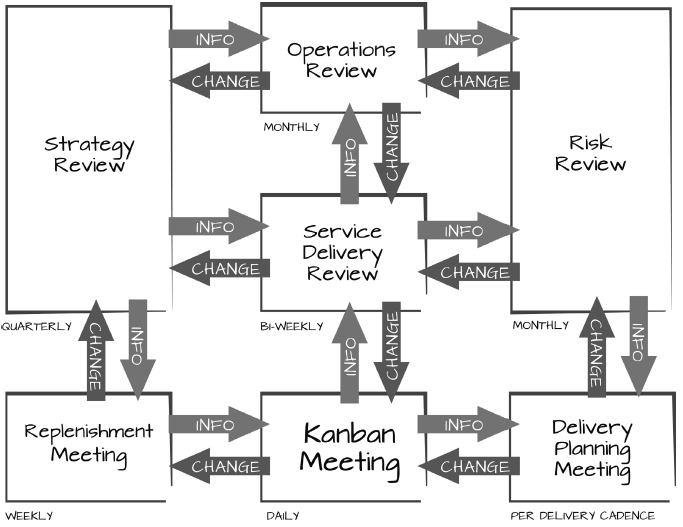


Vir: (Anderson in Carmichael 2016, 21)

*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. 4.29). 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. Kanban sestanek in sestanek dopolnjevanja veljata za osnovi. Namen dodanih kadenc je razširitev obstoječih sestankov z namenom pridobitve informacij njihovega konteksta (Anderson in Carmichael 2016, 25). V osnovi lahko en sestanek zajema več kadenc (Anderson in Carmichael 2016, 26).

Slika 4.29 kadence po Anderson in Charmichael



Vir: (Anderson in Carmichael 2016, 24)

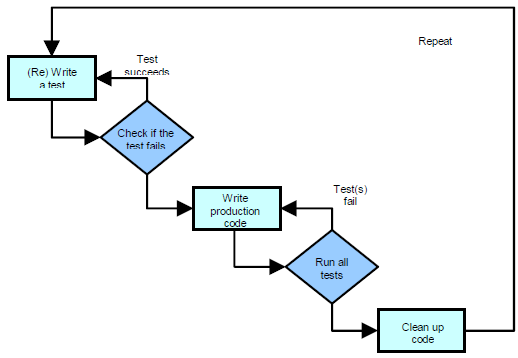
*Optimizacija* je temeljni cilj metode Kanban. Metoda uporablja paradigmo vitke metodologije za doseganje evolucijskih in inkrementalnih izboljšav. Metoda omogoča sprejemanje in oblikovanje uporabnih sprememb in hkrati zavračanje neuporabnih (Anderson in Carmichael 2016, 26).

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).

4.5.9 TESTNO USMERJEN RAZVOJ

Pristop za izboljšanje procesa razvoja in zmanjšanje ponavljajočega dela je testno[[49]](#footnote-49) usmerjen razvoj. Sloni na ponovitvah 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 4.30 prikaz cikla TDD po Duka in Hribar

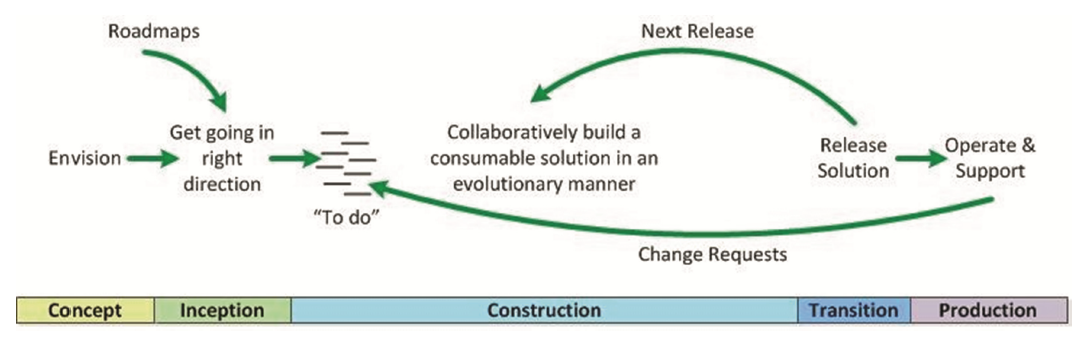
  
vir: (Duka in Hribar 2010, 2)

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. 4.5.1 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).

4.5.10 DISCIPLINIRAN AGILNI PROCES

Cilj metode je 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 DA (Disciplinirana Agilna Metoda) okvirja pa je težko vedeti katere spadajo skupaj (Ambler in Lines 2016, 5). DA omogoča usmeritve kdaj in kako jih umestiti v proces. DA podpira celoten cikel izdaje (Ambler in Lines 2016, 6). Slika 4.31 prikazuje abstrakcijo procesnega cikla.

Slika 4.31 življenjski cikel discipliniranega agilnega procesa po Amber in Lines



Vir: (Ambler in Lines 2016, 6)

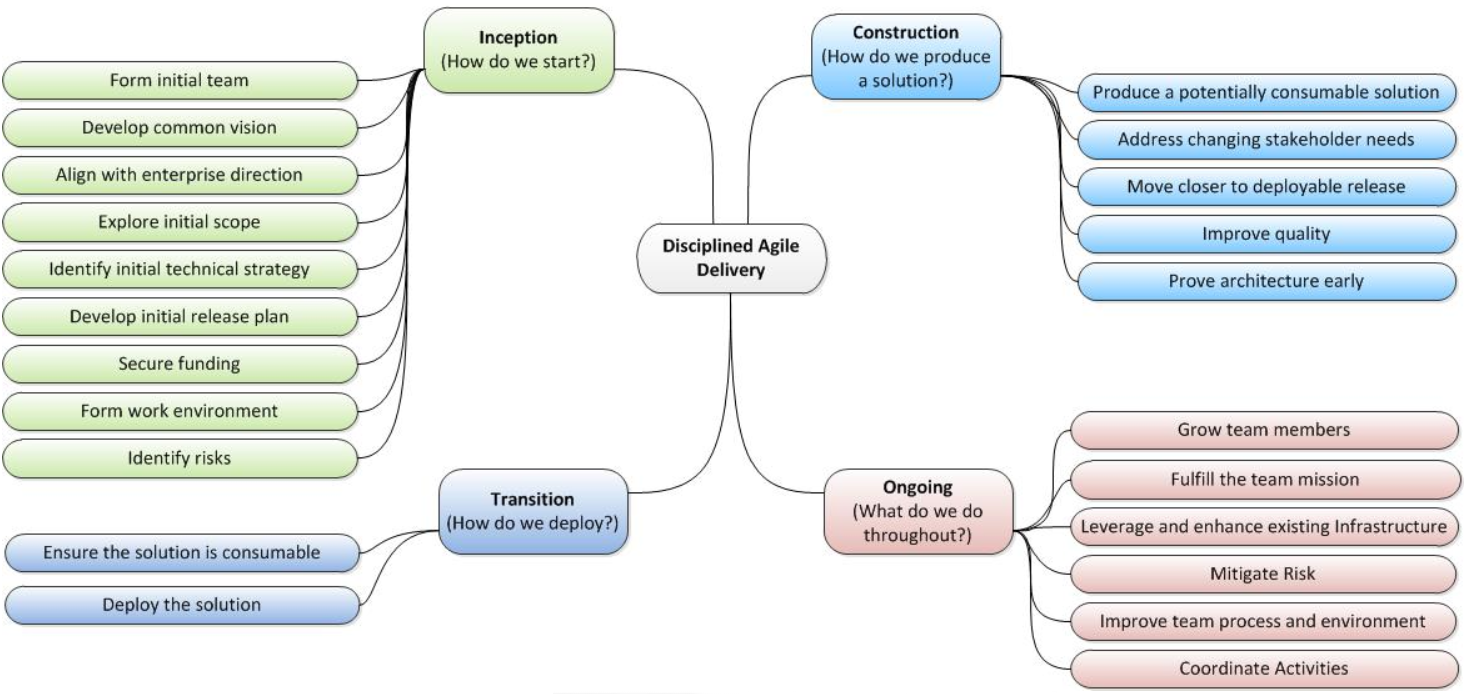
Notranje tri faze začetka, izgradnje, in prehoda, formirajo razvojni del. V tem delu se izvaja inkrementacija produkta. Navadno gredo sistemi večkrat skozi ta del procesa (Ambler in Lines 2016, 6). DA pozna štiri različice razvojnega dela procesa. Prva je agilna/bazična različica, ki razširja cikel izgradnje 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[[50]](#footnote-50) pristop. Razvojna ekipa na podlagi problematike izbere ustrezno različico (Ambler in Lines 2016, 7).

Za potrebe izboljšav procesa DA posvoji uporabo ciljno usmerjenih nalog. Naloge so povzete in predstavljene z miselnimi vzorci (sl. 4.32). 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).

DA 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).

DA 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 4.32 miselno vzorec ciljev razvoja po Amber in Lines



Vir: (Ambler in Lines 2016, 6)

Procesni model ni enostaven linearen model ampak vključuje potrebo po povratnih informacijah od koraka do koraka (Sommerville 2010, 31).

Parts of the system which are difficult to

specify in advance, such as the user interface, should always be developed using an

incremental approach.(Sommerville 2010, 30)

The waterfall model forms the foundation of many software development

methodologies in use today. However, it has some limitations and, if followed

too strictly, can lead to the following types of problems: (Lethbridge in Laganiere 2005, 429)

V uvod

Ne obstaja idealen proces in večina organizacij je razvila svoje procese razvoja (Sommerville 2010, 28). Čeprav ne obstaja idealen proces, velja, da v veliko organizacijah obstaja prostor za izboljšave. Procesi lahko uporabljajo zastarele tehnike ali pa ne izkoriščajo najnovejših in najboljših praks. Veliko organizacij še vedno ne uporablja najnovejših pristopov pri razvoju lastne programske opreme. Proces razvoja je lahko izboljšan s standardizacijo procesov. To vodi v izboljšano komunikacijo, manjšo porabo časa pri uvajanju in zmanjšanje stroškov avtomatiziranega vodenja. (Sommerville 2010, 29).

Pressman (2014) deli modele glede na njihovo strukturo in namembnost na predpisujoče, specializirane, enotne, osebne in ekipne.

Navadno je najdlje trajajoča aktivnost vzdrževanje programske opreme (Sommerville 2010, 31).

Življenjski cikel razvoja programske opreme predstavlja organizacijsko shemo procesa razvoja (Glass 2002).

Proces razvoja programske opreme je skupek aktivnosti, ki vodijo v produkcijo programske opreme. (Pressman in Maxim 2014, 15).

Inženiring procesa razvoja programske opreme omogoča racionalen in časovno sprejemljiv razvoj programske opreme (Pressman in Maxim 2014, 15).

Also, the earlier admonitions regarding changes were based

on a Big Bang style of development. That is, work went on for months with no running code,

and then, at some point, a build happened. (Peters 2008, 117).

This software was developed by large teams working for different companies. Teams

were often geographically dispersed and worked on the software for long periods of

time. An example of this type of software is the control systems for a modern aircraft,

which might take up to 10 years from initial specification to deployment. These plandriven

approaches involve a significant overhead in planning, designing, and documenting

the system. This overhead is justified when the work of multiple development teams

has to be coordinated, when the system is a critical system, and when many different

people will be involved in maintaining the software over its lifetime.

However, when this heavyweight, plan-driven development approach is applied

to small and medium-sized business systems, the overhead involved is so large that it

dominates the software development process. More time is spent on how the system

should be developed than on program development and testing. As the system

requirements change, rework is essential and, in principle at least, the specification

and design has to change with the program.

Dissatisfaction with these heavyweight approaches to software engineering led a

number of software developers in the 1990s to propose new ‘agile methods’. (Sommerville 2010, 58).

These

allowed the development team to focus on the software itself rather than on its design

3.1 \_ Agile methods **59**

and documentation. Agile methods universally rely on an incremental approach to software

specification, development, and delivery. They are best suited to application development

where the system requirements usually change rapidly during the development

process. They are intended to deliver working software quickly to customers, who can

then propose new and changed requirements to be included in later iterations of the system.

They aim to cut down on process bureaucracy by avoiding work that has dubious

long-term value and eliminating documentation that will probably never be used. (Sommerville 2010, 59)

Tudi uvoddddddddddd:::::::LKJLČČKMČLKČPJHJIG

Čeprav je industrija razvoja iger sprva prevzela najboljše prakse tradicionalnega razvoja programske opreme je osnovna razlike v tem, da se razvoj iger osredotoča bolj na uporabniške izkušnje kot na produkt sam (O’Hagan, Coleman, in O’Connor 2014, 182). Razvoj igre je bolj podobno izdelku, ustvarjenem s prepletanjem aspektov, umetnosti, glasbe, programiranja, igranja, poslovnega upravljanja, integriranih v eno celoto. (Ramadan in Widyani 2013, 95). Potrebno je razširiti tradicionalne tehnike razvoja, da bi lahko podprli kreativni proces razvoja elektronskih iger („What went wrong? A survey of problems in game development“ 2017, 19). Za načrtovanje in upravljanje takšnih kompleksnih multidisciplinarnih projektov je potrebna metodologija, kjer ad hoc načini upravljanja ne pridejo v poštev (Aslan in Balci 2015, 307).

Proces video iger ni podrobno definiran. (McAllister in White 2015, 14). Dejstvo je, da je več študij narejenih na akademski strani, poročila s strani industrije iger pa se nahajajo v sivih literaturah, revijah, spletnih straneh. (O’Hagan, Coleman, in O’Connor 2014, 190). Večina raziskav pa se osredotoča na konec življenjskega cikla razvoja iger, kar je dobro za manjše spremembe ni pa to efektivno za spremembo ključne mehanike v igri (McAllister in White 2015, 14).

Ni procesnega modela za razvoj video iger, ki bi vključeval najboljše prakse. Takšen model bi bil zelo koristen za industrijo iger, saj bi lahko skrajšal čas razvoja in vstopa na trg in izboljšal kvaliteto video iger (O’Hagan in O’Connor 2015, 15). Potrebno je boljše razumevanje življenjskega cikla razvoja iger (McAllister in White 2015, 14). Raziskava na temo razvojnih procesov video iger bi bila koristna (O’Hagan in O’Connor 2015, 14).

• začni delati s tem kar imaš  
• prizadevaj si za izboljšanje na podlagi evolucijskih sprememb  
•spodbujaj

UPRAVLJANJE SPREMEMB

ZAGOTAVLJANJE STORITEV

1. Ime modela izvira iz košarkarskega pojma, ki pomeni zabijanje žoge skozi obroč (Peters 2008, 109). [↑](#footnote-ref-1)
2. (angl.) Workflow. [↑](#footnote-ref-2)
3. Uporablja se tudi izraz modeliranje. [↑](#footnote-ref-3)
4. Commercial off-the-shelf. [↑](#footnote-ref-4)
5. Software reuse; je uporaba obstoječe programske opreme ali spoznanj v namen izgradnje nove programske opreme (Frakes in Kang 2005, 529). [↑](#footnote-ref-5)
6. Procesni model je vpeljalo in ga uporablja podjetje IBM. [↑](#footnote-ref-6)
7. (angl.) Object oriented paradigm – OOP. [↑](#footnote-ref-7)
8. Crosscutting concerns. [↑](#footnote-ref-8)
9. Uveljavi ga IBM leta 1980. Kasneje je bil predstavljen v knjigi Jamesa Martina Rapid Application Development. Skrajšano RAD. [↑](#footnote-ref-9)
10. (angl.) Graphical User Interface builders – GUI builders. [↑](#footnote-ref-10)
11. (angl.) Computer Aided Software Engineering – CASE. [↑](#footnote-ref-11)
12. (angl.) Database Management System – DBMS. [↑](#footnote-ref-12)
13. (angl.) forth-generation programing language – 4GL. Danes (2017) v uporabi že 5GL. [↑](#footnote-ref-13)
14. (angl.) Time box. [↑](#footnote-ref-14)
15. (angl.) Brainstorming. [↑](#footnote-ref-15)
16. (angl.) Joint Application Design – JAD. [↑](#footnote-ref-16)
17. Proces imenovan tudi Rational Unified Process (RUP) po podjetju Rational Corporation v lasti IBM. [↑](#footnote-ref-17)
18. Ivar Jacobson, Grady Booch in James Rumbaugh. [↑](#footnote-ref-18)
19. (angl.) Unified Modelling Language – UML; je rezultat dela Unified Process, ki vsebuje robustne usmeritve modeliranju in razvoju objektno-orientiranih sistemov. Do leta 1997 postane UML stanard za objektno-orientiran razvoj programske opreme. [↑](#footnote-ref-19)
20. (angl.) Personal Software Process – PSP. [↑](#footnote-ref-20)
21. (angl.) Postmortem. [↑](#footnote-ref-21)
22. (angl.) Team Software Process – TSP. [↑](#footnote-ref-22)
23. Release Candidate-RC, Release to Manufacturing (RTM). [↑](#footnote-ref-23)
24. Frederick P. Brooks, Jr. [*The Mythical Man-Month*](https://en.wikipedia.org/wiki/The_Mythical_Man-Month). 1995 [1975] [↑](#footnote-ref-24)
25. Extreme programming. [↑](#footnote-ref-25)
26. Za zbiranje zgodbe se uporablja tehnika stimulacije dražljaja (angl.) Stimulus-Response scenario technique – SRS. [↑](#footnote-ref-26)
27. (angl.) Index score (story point) po Peters (2008) in cost po Pressman in Maxim (2014) . [↑](#footnote-ref-27)
28. IBM Rational Group, 2003. [↑](#footnote-ref-28)
29. (angl.) Design to cost. [↑](#footnote-ref-29)
30. (angl.) Keep it simple. [↑](#footnote-ref-30)
31. (angl.) Class responsibility collaborator. Orodje za viharjenje možganov. [↑](#footnote-ref-31)
32. (angl.) Spike solution. [↑](#footnote-ref-32)
33. 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-33)
34. (angl.) Backlog. [↑](#footnote-ref-34)
35. (angl.) Dynamic System Development Method. DSDM konzorcij je skrbnik metode. [↑](#footnote-ref-35)
36. (angl.) Crystal. [↑](#footnote-ref-36)
37. (angl.) Cone of silence. Določen čas, da se izkušeni lahko posvetijo svojim nalogam. [↑](#footnote-ref-37)
38. Družina operacijskih sistemov podobni Unixu. [↑](#footnote-ref-38)
39. (angl.) Just in time – JIT. [↑](#footnote-ref-39)
40. (angl.) Feature-driven development – FDD. [↑](#footnote-ref-40)
41. (angl.) Configuration Management – CM. Sistem (orodje), ki skrbi za konsistenco produkta v razvoju na podlagi zahtev in načrtov. Najbolj znana orodja; CFEngine, Puppet, Chef, Ansible, SaltStack, Docker, PowerShell DSC, Git. [↑](#footnote-ref-41)
42. (angl.) Client-valued functions or features. [↑](#footnote-ref-42)
43. Japonska beseda. Pomeni znak, signalno karto ali veliko vizualno tablo. [↑](#footnote-ref-43)
44. (angl.) Lead Time. [↑](#footnote-ref-44)
45. (angl.) Work in Progress – WiP. [↑](#footnote-ref-45)
46. (angl.) Delivery Rate. [↑](#footnote-ref-46)
47. (angl.) Little's Law. [↑](#footnote-ref-47)
48. (angl.) Throughtput. [↑](#footnote-ref-48)
49. (angl.) Test-driven development – TDD. [↑](#footnote-ref-49)
50. 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-50)