1. Uvod u softversko inženjerstvo

U ovom uvodnom poglavlju najprije ćemo objasniti osnovne pojmove koji se susreću u softverskom inženjerstvu. Zatim ćemo detaljnije govoriti o modelima koji ugrubo opisuju kako bi se trebao odvijati proces razvoja softvera. Dalje ćemo govoriti o danas relevantnim metodama razvoja softvera, koje profinjavaju i konkretiziraju pojedine modele. Na kraju ćemo istaći da softversko inženjerstvo ima i svoj menadžerski aspekt, dakle da se u okviru razvoja softvera pojavljuje i problematika upravljanja složenim projektima.

1.1. Osnovni pojmovi vezani uz softversko inženjerstvo

Osnovni pojmovi u ovom udžbeniku odnose se na razvoj softvera i već smo ih susretali u disciplinama kao što su programski jezici, strukture podataka i algoritmi, baze podataka, mreže računala i slično. Ipak, sad tim istim pojmovima nastojimo dati preciznije značenje.

1.1.1. Softverski produkt i softversko inženjerstvo

Softverski produkt je skup računalnih programa i pripadne dokumentacije, stvoren zato da bi se isporučio nekom korisniku. Može biti razvijen za sasvim određenog korisnika (*bespoke product*, *customized product*) ili općenito za tržište (*generic product*). Softverski produkt često ćemo kraće nazivati *softver* ili (softverski) *sustav*.

Za današnji softver podrazumijeva se da on mora biti kvalitetan. Preciznije, od softverskog produkta očekuje se da se on odlikuje sljedećim atributima kvalitete:

- *Mogućnost održavanja*. Softver se mora moći mijenjati u skladu s promijenjenim potrebama korisnika.
- *Pouzdanost i sigurnost*. Softver se mora ponašati na predvidiv način te ne smije izazivati fizičke ili ekonomske štete.
- *Efikasnost*. Softver mora imati zadovoljavajuće performanse te on treba upravljati računalnim resursima na štedljiv način.
- *Upotrebljivost*. Softver treba raditi ono što korisnici od njega očekuju, sučelje mu treba biti zadovoljavajuće te za njega mora postojati dokumentacija.

Softversko inženjerstvo (*software engineering*) je disciplina koja se bavi svim aspektima proizvodnje softvera. Dakle, softversko inženjerstvo bavi se modelima, meto-

www.element.hr

dama i alatima koji su nam potrebni da bi na što učinkovitiji način mogli proizvoditi što kvalitetnije softverske produkte.

Softversko inženjerstvo može se smatrati znanstvenom disciplinom, no također i tehničkom strukom. U oba slučaja, ono je u bliskoj vezi s još dva područja znanja:

- *Računarstvo* (*computer science*). Poznavanje teorijskog računarstva potrebno je softverskom inženjeru na sličan način kao što je poznavanje mehanike potrebno strojarskom inženjeru.
- *Sistemsko inženjerstvo* (*system engineering*). Bavi se razvojem složenih sustava koji se sastoje od hardvera, softvera i ljudskih aktivnosti. Softverski inženjer mora svoje softversko rješenje uklopiti u takav složeniji sustav.

Softversko inženjerstvo također je i profesija. U mnogim zemljama softverski inženjeri organizirani su u strukovne udruge koje štite njihova prava, ali donekle i ograničavaju slobodu njihovog djelovanja. Primjeri takvih udruga su ACM, IEEE-CS i British Computer Society. Udruge donose pravila ponašanja za svoje članove. Na *web*-stranici [2] može se naći etički kodeks softverskih inženjera koji su zajednički usvojili ACM i IEEE-CS.

Softversko inženjerstvo počelo se razvijati krajem 60-ih godina 20. stoljeća. Sam naziv izgleda da je bio skovan na jednoj NATO-ovoj konferenciji u Njemačkoj 1968. godine. Disciplina je nastala kao odgovor na takozvanu softversku krizu. Naime, pojavom računala treće generacije (npr. IBM serija 360) stvorila se potreba za složenijim softverom (primjerice *multitasking* operacijski sustav). Pokrenuti razvojni projekti višestruko su premašili planirane troškove i rokove. Vidjelo se da se dotadašnje neformalne tehnike individualnog programiranja ne mogu uspješno skalirati na velike programe gdje sudjeluje velik broj programera. Osjećala se potreba za metodama razvoja softvera koje bi bile u stanju kontrolirati kompleksnost velikog softverskog projekta. Stanje svijesti iz tih vremena plastično je opisano u čuvenoj knjizi [12].

Od 60-ih godina 20. stoljeća do danas softversko inženjerstvo prešlo je dug put. Ispočetka su se predlagale metode razvoja softvera oblikovane po analogiji s metodama iz tradicionalnih tehničkih struka (kao što je npr. gradnja mostova). Kasnije se uvidjelo da je softver po mnogočemu specifičan te zahtijeva drukčije pristupe. Razvijali su se novi programski jezici, bolji načini utvrđivanja zahtjeva, grafički jezici za modeliranje, formalne metode za specifikaciju i verifikaciju, pouzdaniji načini vođenja projekata i procjene troškova, alati koji nastoje automatizirati proces razvoja softvera. Zahvaljujući takvom razvoju, softversko inženjerstvo uspjelo se etablirati kao važan dio tehnike i računarstva. Softver se u današnje vrijeme proizvodi daleko predvidljivije i efikasnije nego prije. Ipak, još uvijek postoji širok prostor za poboljšanje.

Važna osobina softverskog inženjerstva je da u njemu nema "jednoumlja" ni jednoobraznosti. Za razliku od matematike gdje se proučavaju nepobitne istine, u softver-

www.element.hr

skom se inženjerstvu razvijaju različite ideje i pristupi koji su često u međusobnom nesuglasju. Glavni razlog za takvu raznolikost pristupa je raznolikost samih softverskih produkata – zaista, teško je očekivati da se sve vrste softvera mogu razvijati na isti način. Drugi razlog za raznolikost je činjenica da softversko inženjerstvo još uvijek nije doseglo svoju zrelu fazu – ta disciplina se i ovog trenutka intenzivno razvija, a nove spoznaje stalno revidiraju stare.

1.1.2. Softverski proces, metode i alati

Softverski proces je skup aktivnosti i pripadnih rezultata čiji je cilj razvoj ili evolucija softvera. Osnovne aktivnosti unutar softverskog procesa su: utvrđivanje zahtjeva, oblikovanje, implementacija, verifikacija i validacija te održavanje odnosno evolucija.

Model za softverski proces je idealizirani prikaz softverskog procesa, kojim se određuje poželjan način odvijanja i međusobnog povezivanja osnovnih aktivnosti. Primjerice, model može zahtijevati slijedno, odnosno simultano odvijanje aktivnosti. Najvažniji modeli bit će opisani u potpoglavlju 1.2.

Metoda razvoja softvera je profinjenje i konkretizacija odabranog modela za softverski proces. Metoda uvodi specifičnu terminologiju. Također, ona dijeli osnovne aktivnosti u podaktivnosti te propisuje što se sve mora raditi unutar pojedine podaktivnosti. Dalje, metoda uvodi konkretan način dokumentiranja rezultata podaktivnosti (dijagrami, tabele, pseudojezik...) te daje naputke vezane uz organizaciju rada, stil oblikovanja, stil programiranja itd. Više o metodama razvoja softvera bit će rečeno u potpoglavlju 1.3.

CASE alati (*Computer Aided Software Engineering*) su softverski paketi koji daju automatiziranu podršku za pojedine aktivnosti unutar softverskog procesa. Obično su napravljeni u skladu s određenom metodom razvoja softvera, implementiraju pravila iz te metode, sadrže editore za odgovarajuće dijagrame i služe za izradu odgovarajuće dokumentacije. Dijele se u dvije vrste:

- *Upper-CASE* alati daju podršku za početne aktivnosti unutar softverskog procesa, kao što su utvrđivanje zahtjeva i oblikovanje.
- *Lower-CASE* alati podržavaju samu realizaciju softvera, dakle programiranje, verifikaciju i validaciju te eventualno održavanje.

Na vježbama iz ovog kolegija obradit ćemo jednu konkretnu metodu razvoja softvera, koja se zove UP i koja je potpomognuta grafičkim jezikom UML. Služit ćemo se *upper*-CASE alatom Visual Paradigm [64] za modeliranje sustava i crtanje UML dijagrama. Na vježbama iz drugih kolegija studenti su imali prilike raditi s *lower*-CASE alatom Microsoft Visual Studio [12] koji daje podršku za implementaciju i verifikaciju softvera pisanog u jezicima poput C++ ili C#.

www.element.hr

1.1.3. Dokumentacija softvera

Kad smo govorili o softverskom produktu, rekli smo da se on sastoji od računalnih programa i pripadne *dokumentacije*. Dakle, razvoj softvera ne svodi se samo na razvoj programa, nego također i na sastavljanje dokumenata koji prate te programe. Dokumentacija je sastavni i nezaobilazni dio onoga što softverski inženjeri proizvode.

Dokumentacija softvera može se podijeliti u dvije kategorije:

- *Sistemska dokumentacija* namijenjena je softverskim inženjerima ili članovima razvojnog tima, opisuje zahtjeve ili građu ili funkcioniranje sustava te omogućuje sam razvoj softvera i njegovo kasnije održavanje.
- *Korisnička dokumentacija* namijenjena je korisnicima, omogućuje im da se koriste sustavom te opisuje funkcije sustava na njima razumljiv način.

U daljnjim poglavljima opisat ćemo nekoliko primjera dokumenata koji spadaju u sistemsku dokumentaciju. Primjerice, to su sljedeći dokumenti:

- *Dokument o zahtjevima*. Opisuje zahtjeve na sustav, dakle što sustav treba raditi i uz koja ograničenja.
- *Dizajn sustava* (projektna dokumentacija). Nastaje kao rezultat oblikovanja sustava. Opisuje kako sustav treba biti građen te kako treba raditi da bi obavio svoje funkcije.
- *Plan testiranja*. Sadrži primjere test-podataka i odgovarajućih očekivanih rezultata. Služi nakon bilo kakve promjene programa za provjeru je li sačuvana dotadašnja funkcionalnost programa.
- *Izvorni tekst programa.* Pisan je u programskom jeziku i nadopunjen odgovarajućim komentarima. Nastaje kao rezultat implementacije sustava i na najizravniji način dokumentira sustav.

Kad god se koristimo nekim generičkim softverskim produktom, npr. operacijskim sustavom ili paketom za uredsko poslovanje ili sustavom za upravljanje bazom podataka, susrećemo se s raznim dokumentima koji spadaju u korisničku dokumentaciju. Evo nekoliko primjera takvih dokumenata:

- *Funkcionalni opis*. Namijenjen je procjeniteljima sustava, dakle osobama koje će odlučiti hoće li se koristiti baš taj sustav ili neki drugi. Daje se grubi opis što sustav može, a što ne može raditi. Priloženi su primjeri, tablice i dijagrami. Ne objašnjavaju se operativni detalji.
- *Vodič za instalaciju*. Namijenjen je sistemskom administratoru ili korisniku. Opisuje instalaciju sustava na zadanoj vrsti računala. Sadrži opis distribucijskog medija, definiciju minimalne hardverske i softverske konfiguracije potrebne za pokretanje sustava, proceduru instaliranja i podešavanja.

www.element.hr

- *Uvodni priručnik*. Namijenjen je novim korisnicima. Daje neformalni uvod u sustav te opisuje njegovu "normalnu" upotrebu. Pisan je u formi "tečaja", sadrži mnogo primjera te uputa kako izbjeći uobičajene pogreške.
- *Referentni priručnik*. Namijenjen je iskusnim korisnicima. U potpunosti i na vrlo precizan način dokumentira sve funkcije sustava, sve oblike njegove upotrebe te sve pogreške koje mogu nastupiti. Nije u formi tečaja. Stil je formalan, struktura je stroga. Snalaženje u tekstu osigurano je preko indeksa pojmova.
- *Priručnik za administriranje*. Namijenjen je sistemskim administratorima ili računalnim operaterima koji su odgovorni za rad instaliranog sustava. Opisuje aktivnosti poput izrade sigurnosne kopije (*backupa*), upravljanja resursima, podešavanja performansi, evidentiranja korisnika, postavljanja zaštite.

Izradu dokumentacije jednim je dijelom moguće automatizirati ili barem olakšati korištenjem CASE-alata. No, za konačno oblikovanje dokumenata potrebni su nam standardni uredski alati poput tekstnog procesora ili alata za crtanje dijagrama. Osim u obliku dokumenata (bilo papirnatih bilo elektroničkih), dokumentacija može postojati i u interaktivnom (*online*) obliku, dakle ona može biti izravno dostupna iz samog softvera koji opisuje.

1.2. Modeli za softverski proces

Zajednička osobina svih modela za softverski proces je da se oni u većoj ili manjoj mjeri zasnivaju na otprilike istim osnovnim aktivnostima. Zaista, to su sljedeće aktivnosti:

- *Utvrđivanje zahtjeva* (*specifikacija*). Analiziraju se zahtjevi korisnika. Utvrđuje se *što* softver treba raditi.
- *Oblikovanje* (*design*). Oblikuje se građa sustava, način rada njegovih dijelova te sučelje između dijelova. Dakle projektira se rješenje koje određuje *kako* će softver raditi.
- *Implementacija* (*programiranje*). Oblikovano rješenje realizira se uz pomoć raspoloživih programskih jezika i alata.
- *Verifikacija* i *validacija*. Provjerava se radi li softver prema specifikaciji, odnosno radi li ono što korisnik želi. Obično se svodi na *testiranje*, makar postoje i druge tehnike.
- *Održavanje* odnosno *evolucija*. Nakon uvođenja u upotrebu, softver se dalje popravlja, mijenja i nadograđuje, u skladu s promijenjenim potrebama korisnika.

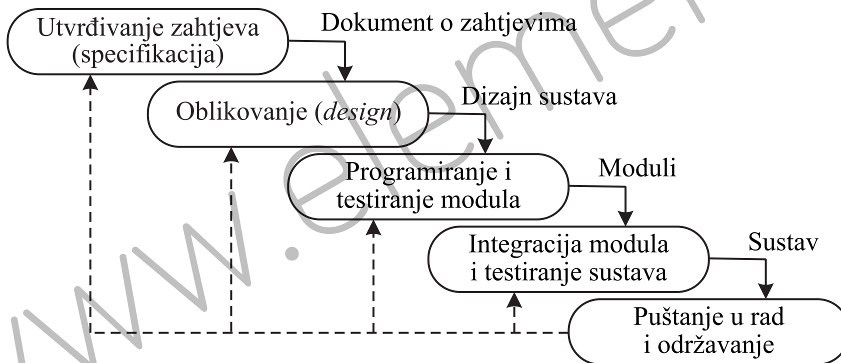
Modeli se razlikuju po načinu odvijanja i međusobnog povezivanja osnovnih aktivnosti. Također, svaki od njih može staviti veći ili manji naglasak na pojedinu aktiv-

www.element.hr

nost. U sljedećim odjeljcima opisat ćemo nekoliko modela te ćemo istaknuti njihove prednosti i mane.

1.2.1. Model vodopada

Model vodopada (*waterfall model*) nastao je u ranim 70-im godinama 20. stoljeća, kao neposredna analogija s procesima iz drugih inženjerskih struka (npr. gradnja mostova). Softverski proces građen je kao niz vremenski odvojenih aktivnosti, u skladu sa slikom 1.1.



Slika 1.1. Softverski proces prema modelu vodopada

Model je dobio ime zbog oblika dijagrama. Vidimo da se aktivnosti odvijaju kao faze slijedno jedna iza druge. Svaka faza daje neki rezultat koji "teče" po vodopadu i predstavlja polazište za iduću fazu. Makar je na slici naznačena mogućnost povratka u raniju fazu (u slučaju naknadnog otkrivanja pogreške), ta se mogućnost samo iznimno koristi. Povratak je nepoželjan jer remeti normalni tijek procesa te izaziva kašnjenje.

Prednosti modela vodopada su sljedeće:

- Model omogućuje detaljno planiranje cijelog softverskog procesa i podjelu poslova na velik broj suradnika.
- Lagano se može utvrditi stanje u kojem se proces trenutno nalazi.

Zato menadžeri dobro prihvaćaju model.

Mane modela vodopada su sljedeće:

- Navedene faze u praksi je teško razdvojiti, pa dolazi do naknadnog otkrivanja pogrešaka i nepoželjnog vraćanja u prethodne faze.
- Zbog tendencije "zamrzavanja" pojedine faze u određenom trenutku zbog poštovanja rokova, dešava se da se sustav nastavlja razvijati u nezadovoljavajućem obliku.

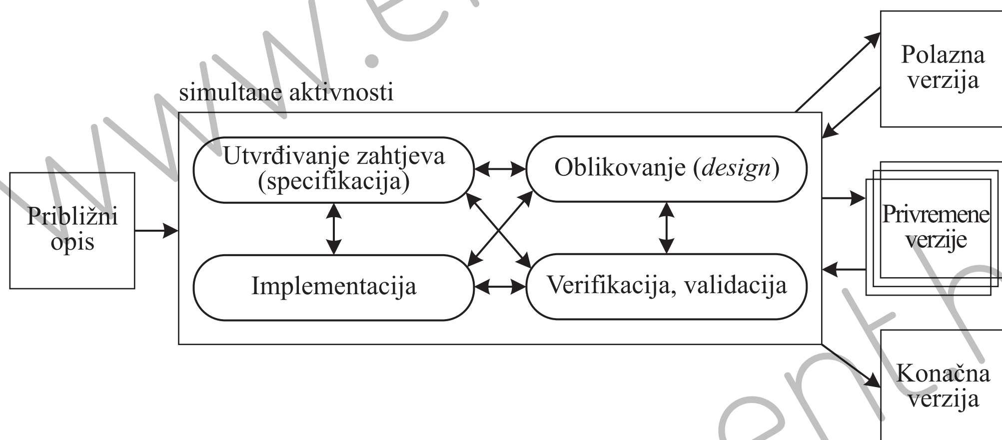
www.element.hr

- Proces je spor, pa se može dogoditi da u trenutku puštanja u rad sustav već bude neažuran i zastario.

Zbog navedenih prednosti te unatoč navedenim manama, model se često koristi u praksi. Model je pogodan kad treba razviti veliki sustav s relativno jednostavnim i jasnim zahtjevima uz pomoć velikog broja programera.

1.2.2. Modeli evolucijskog i inkrementalnog razvoja

Model evolucijskog razvoja (*evolutionary development*) nastao je kao protuteža modelu vodopada. U tom se modelu na osnovi približnog opisa problema razvija polazna verzija sustava koja se pokazuje korisniku. Temeljem korisnikovih primjedbi, ta se polazna verzija poboljšava, opet pokazuje itd. Nakon dovoljnog broja iteracija dobiva se konačna verzija sustava. Unutar svake iteracije osnovne se aktivnosti obavljaju simultano i ne daju se razdvojiti. Postupak je ilustriran slikom 1.2.



Slika 1.2. Evolucijski razvoj softvera

Evolucijski razvoj čiji je cilj da se s korisnikom istraže zahtjevi te zaista proizvede konačni sustav zove se *istraživačko programiranje*. No, ako je jedini cilj istraživanje zahtjeva (nakon čega slijedi oblikovanje i implementacija modelom vodopada), tada je riječ o prototipiranju – vidjeti potpoglavlje 2.2.

Prednosti modela evolucijskog razvoja su sljedeće:

- Model je u stanju proizvesti brzi odgovor na zahtjeve korisnika.
- Razvoj je moguć i onda kad su zahtjevi ispočetka nejasni.
- Specifikacija se može postupno razvijati u skladu sa sve boljim korisnikovim razumijevanjem problema.

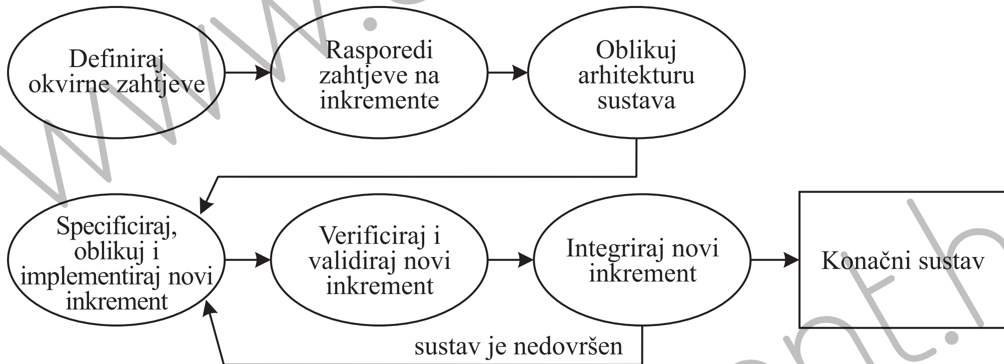
www.element.hr

Mane modela evolucijskog razvoja su sljedeće:

- Proces nije transparentan za menadžere, naime oni ne mogu ocijeniti koliki je dio posla napravljen i kad će sustav biti gotov.
- Konačni sustav obično je loše strukturiran zbog stalnih promjena te je neprikladan za kasnije održavanje.
- Zahtijevaju se posebni alati i natprosječni softverski inženjeri.

Model se uspješno koristi za razvoj *web*-sjedišta te za male sustave s kratkim životnim ciklusom, pogotovo za sustave s nejasnim zahtjevima.

Model inkrementalnog razvoja (*incremental development*) sličan je modelu evolucijskog razvoja i može se shvatiti kao hibrid vodopada i evolucije. Sustav se opet razvija u nizu iteracija. No za razliku od evolucijskog modela, pojedina iteracija ne dotjeruje već realizirani dio sustava, nego mu dodaje sasvim novi dio – inkrement. Razvoj jednog inkrementa unutar jedne iteracije odvija se po bilo kojem modelu – npr. kao vodopad. Ideja je prikazana na slici 1.3.



Slika 1.3. Inkrementalni razvoj softvera

Prednosti modela inkrementalnog razvoja su sljedeće:

- Proces je još uvijek prilično transparentan za menadžere, jer je vidljivo do kojeg smo inkrementa došli.
- Korisnici ne moraju dugo čekati da bi dobili prvi inkrement koji zadovoljava njihove najvažnije potrebe. Razvoj slijedi prioritete.

Mane modela inkrementalnog razvoja su sljedeće:

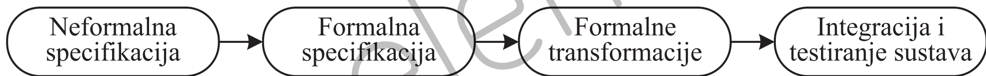
- Koji put je teško podijeliti korisničke zahtjeve u smislene inkremente.
- Budući da cjelokupni zahtjevi nisu dovoljno razrađeni na početku, teško je odrediti zajedničke module koji su potrebni raznim inkrementima i koji bi morali biti implementirani u prvom inkrementu.

www.element.hr

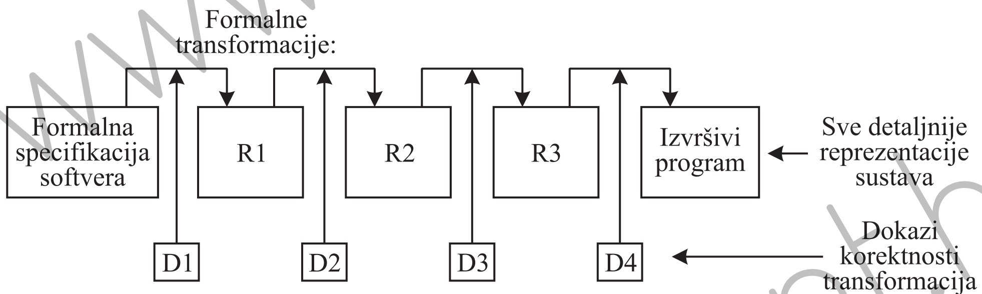
Ipak, ovo je vrlo upotrebljiv model kojim se intenzivno koristimo u praksi. Svi generički softverski paketi poput Microsoft Officea [42] i sličnih zapravo su se razvijali u inkrementima. Svaka njihova nova verzija donosila je nove mogućnosti.

1.2.3. Modeli formalnog i grafičkog razvoja

Model formalnog razvoja (*formal systems development*) zasniva se na korištenju takozvanih formalnih metoda za razvoj softvera. Zahtjevi se precizno definiraju na formalni način, korištenjem nedvosmislene matematičke notacije. Zatim se ta formalna specifikacija transformira do programa, nizom koraka koji čuvaju korektnost. Cijeli proces prikazan je slikom 1.4, dok je ideja formalnih transformacija detaljnije ilustrirana slikom 1.5.



Slika 1.4. Proces formalnog razvoja softvera



Slika 1.5. Transformacije u sklopu formalnog razvoja softvera

Prednosti modela formalnog razvoja su sljedeće:

- Nakon izrade formalne specifikacije, sve daljnje faze razvoja softvera mogle bi se automatizirati.
- Postoji "matematički" dokaz da je program točna implementacija polazne specifikacije; nema velike potrebe za testiranjem.
- Rješenje je neovisno o platformi, dakle prenosivo (portabilno).

Mane modela formalnog razvoja su sljedeće:

- Izrada formalne specifikacije zahtijeva velik trud i znanje.
- Dokazi korektnosti transformacija postaju preglomazni za iole veće sustave.
- Korisnici ne mogu pratiti razvoj.
- Postojeći specifikacijski jezici nisu pogodni za interaktivne sustave.

www.element.hr

Model formalnog razvoja za sada se relativno malo koristi u praksi te se preporučuje jedino za sustave gdje se zahtijeva izuzetno velika pouzdanost i sigurnost. Moguće je da će se model znatno više koristiti u budućnosti, ako se razviju bolji specifikacijski jezici te bolji alati za automatske transformacije.

Model grafičkog razvoja (*model driven engineering*) u osnovi je sličan modelu formalnog razvoja. Umjesto formalne specifikacije razvija se grafički model (skup dijagrama) koji prikazuje građu i ponašanje sustava. Koristi se grafički jezik za modeliranje poput UML-a – vidjeti potpoglavlje 2.3. Primijetimo da se ovdje riječ "model" pojavljuje u dva značenja: model sustava nije isto što i model softverskog procesa. Pretpostavlja se da postoji alat koji dijagrame automatski pretvara u program. Cijeli proces prikazan je slikom 1.6.

Slika 1.6. Proces grafičkog razvoja

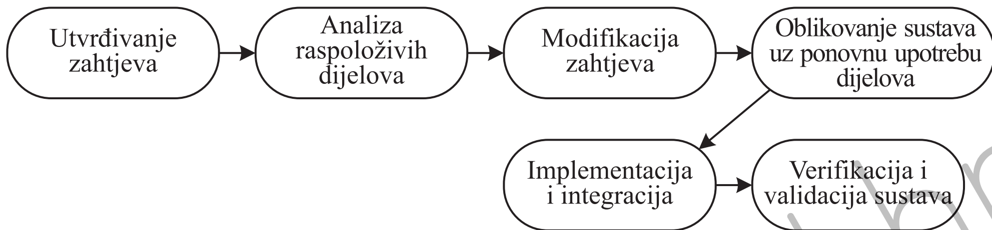
Model grafičkog razvoja ima svoje pobornike i trenutno je predmet intenzivnog istraživanja. Prednosti su mu slične kao kod formalnog razvoja, s time da bi trebao biti jednostavniji za korištenje. Glavna mana mu je da za sada nije upotrebljiv osim u vrlo jednostavnim primjerima. Današnji CASE alati u stanju su generirati dio programskog koda iz UML dijagrama, no daleko su od toga da bi mogli generirati cijeli program.

Kod složenijih sustava UML dijagrami zapravo i ne mogu jednoznačno opisati sve detalje. To se u UML verziji 2.0 nastoji nadoknaditi uvođenjem posebnog jezika zvanog *Object Constraint Language* – OCL; on služi za specificiranje ograničenja koja nisu vidljiva iz dijagrama. Očekuje se da će OCL i slični jezici pomoći da se ostvari automatsko generiranje programa iz modela sustava. No uvođenjem takvih jezika grafički pristup postaje još sličniji formalnom pristupu.

1.2.4. Model usmjeren na ponovnu upotrebu

Model usmjeren na ponovnu upotrebu (*reuse-oriented development*) polazi od pretpostavke da već postoje gotove i upotrebljive softverske cjeline, kao što su javno dostupni sustavi ili komponente opće namjene ili dijelovi prije razvijenih sustava. Novi sustav nastoji se u što većoj mjeri realizirati spajanjem postojećih dijelova, u skladu sa slikom 1.7. Ista ideja prisutna je i u drugim tehničkim disciplinama: novi mehanički ili elektronički proizvod nastoji se sklopiti od postojećih standardnih dijelova (vijaka, čipova...).

www.element.hr



Slika 1.7. Proces razvoja softvera uz ponovnu upotrebu

Prednosti modela usmjerenog na ponovnu upotrebu su sljedeće:

- Smanjuje se količina softvera koju stvarno treba razviti te se tako smanjuje vrijeme, trošak i rizik.
- Stavlja se oslonac na provjerene i dobro testirane dijelove softvera.

Mane modela usmjerenog na ponovnu upotrebu su sljedeće:

- Zbog kompromisa u specifikaciji moguće je da sustav neće u potpunosti odgovoriti stvarnim potrebama korisnika.
- Djelomično je izgubljena kontrola nad evolucijom sustava, jer ne upravljamo razvojem novih verzija korištenih dijelova.

Očekuje se da će ovaj model ipak postati prevladavajući u 21. stoljeću, jer je broj gotovih rješenja sve veći, a korisnici imaju sve manje vremena za čekanje rješenja. Ideja ponovne upotrebe te razni načini njene realizacije detaljno će biti obrađeni u poglavlju 6.

1.3. Klasične i agilne metode razvoja softvera

Rekli smo da se rad softverskog tima obično organizira u skladu s nekom metodom razvoja softvera. Svaka od metoda slijedi jedan od modela za softverski proces ili kombinira više modela. Osnovne aktivnosti iz modela podijeljene su u manje aktivnosti. Propisuje se način odvijanja svake aktivnosti te način njenog dokumentiranja. U sljedećim odjeljcima navodimo osnovna svojstva pojedinih vrsta metoda te detaljnije opisujemo dvije danas aktualne metode: *United Process* – UP odnosno *Extreme Programming* – XP.

1.3.1. Vrste metoda i njihova svojstva

Od početka softverskog inženjerstva do danas pojavilo se nekoliko stotina metoda za razvoj softvera. Njih ugrubo možemo podijeliti u klasične i agilne metode.

www.element.hr

Klasične metode pojavile su se još u 70-im godinama 20. stoljeća, a razvijaju se i danas. One se odlikuju sljedećim svojstvima:

- Razvoj softvera promatra se kao pažljivo planirani proces koji ima svoj početak i kraj. U tom procesu u potpunosti se utvrđuju zahtjevi te se softver realizira u skladu s njima.
- Provodi se upravljanje projektom. Određuje se precizni plan aktivnosti, po potrebi i za dulji vremenski period unaprijed. Određuje se raspored ljudi i resursa po aktivnostima.
- Inzistira se na urednom dokumentiranju svih aktivnosti i svih proizvedenih dijelova softvera.

Klasične metode dalje se dijele na funkcionalno-orijentirane te na objektno orijentirane. Starije *funkcionalno-orijentirane* metode poput Yourdonove ili Jacksonove JSD (80-e godine 20. stoljeća) slijede logiku starijih funkcionalno-orijentiranih programskih jezika (Cobol, C, Fortran). Novije *objektno orijentirane* metode predstavljaju nadgradnju objektno orijentiranih programskih jezika (Java, C++, C#) i danas su se integrirale u zajednički standard UP (*Unified Process* – Booch, Rumbaugh, Jacobson – 90-e godine) koji se oslanja na grafički jezik UML (*Unified Modelling Language*) [3].

Kao protuteža klasičnim metodama, početkom 21. stoljeća pojavile su se *agilne* metode. Njihova glavna namjera bila je ubrzati razvoj softvera i izbjeći nepotrebnu administraciju. Agilne metode imaju sljedeća svojstva:

- Razvoj softvera promatra se kao kontinuirani niz iteracija čiji broj nije moguće predvidjeti. Svaka iteracija usklađuje sustav s trenutnim zahtjevima. Zahtjevi se stalno mijenjaju. Ne postoji cjelovita ili konačna specifikacija.
- Ne postoji upravljanje projektom u pravom smislu riječi. Aktivnosti se dogovaraju s korisnicima te se odvijaju uz njihovo aktivno sudjelovanje. Planiranje je kratkoročno – jedan ili dva tjedna unaprijed.
- Izbjegavaju se svi oblici dokumentacije osim onih koji se mogu automatski generirati iz programa. Smatra se da sam program u izvornom obliku predstavlja svoju najpouzdaniju dokumentaciju.

Najpopularnija agilna metoda je *Extreme Programming* – XP [7]. Također je poznata metoda *Scrum* [36] koja se uglavnom bavi agilnim vođenjem projekta i može se kombinirati sa XP. Ima i pokušaja da se definira agilna varijanta inače klasične metode UP.

U literaturi su zabilježena brojna istraživanja koja mjere i uspoređuju učinkovitost pojedinih metoda za razvoj softvera. Prema tim istraživanjima, agilne metode imaju svojih prednosti kad je riječ o malim i kratkoročnim projektima u brzo promjenjivom poslovnom okruženju. No nije se još dokazalo da se agilne metode mogu uspješno skalirati na velike projekte. Zato se danas veliki softverski sustavi i dalje razvijaju

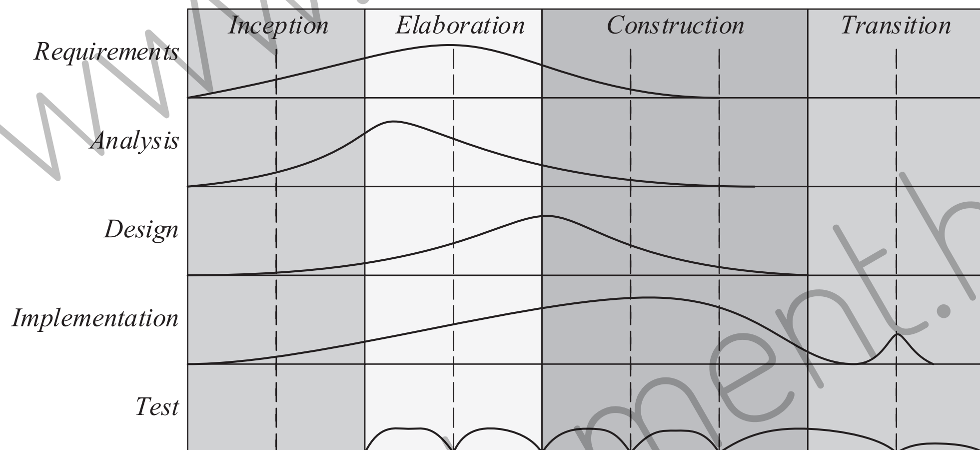
www.element.hr
www.element.hr

klasičnim, i to objektno orijentiranim metodama. Agilne metode našle su svoje mjesto npr. u razvoju *web*-aplikacija ili u razvoju mobilnih aplikacija.

1.3.2. Primjer klasične metode

Kao primjer klasične metode razvoja softvera opisat ćemo *Unified Process* – UP. Riječ je o javno dostupnoj verziji komercijalne metode *Rational Unified Process* – RUP. I UP i RUP koriste se za razvoj objektno orijentiranih sustava. Više detalja o UP-u može se naći u knjizi [3].

UP je nastala na samom kraju 20. stoljeća u tvrtki Rational Corporation koja je danas dio IBM-a. Stvorio ju je Ivar Jacobson u suradnji s Gradyjem Boochom i Jamesom Rumbaughom. Ime je odabrano zato što je UP objedinio nekoliko prijašnjih metoda istih autora. UP je u bliskoj vezi s grafičkim jezikom UML – vidjeti potpoglavlje 2.3 – obje tvorevine razvijene su u okviru istog projekta s namjerom da budu komplementi jedna drugoj. Zaista, UP određuje kako će se odvijati proces razvoja softvera, a UML omogućuje da se rezultati tog razvoja dokumentiraju.



Slika 1.8. Faze i temeljne aktivnosti unutar UP-a

Proces razvoja softvera u skladu s UP-om sastoji se od 4 *faze* (*phases*) koje se realiziraju u vremenskom slijedu jedna iza druge. Pojedina faza završava onda kad se dosegne njena kontrolna točka, takozvani *međaš* (*milestone*), dakle kad se postignu njeni ciljevi. Svaka faza realizira se kroz simultano i iterativno odvijanje 5 *temeljnih aktivnosti* (*core workflows*). Sve je to ilustrirano slikom 1.8 koja je preuzeta iz knjige [3].

Zbog postojanja vremenski odvojenih faza mogli bismo reći da UP uspostavlja softverski proces nalik modelu vodopada. S druge strane, iteracija temeljnih aktivnosti liči na model evolucijskog razvoja. Dakle UP ustvari kombinira ta dva modela nastojeći iskoristiti prednosti jednog i drugog.

www.element.hr

Slika 1.8 također pokazuje da u svakoj fazi postoji jedna ili dvije osnovne aktivnosti na koje je stavljen poseban naglasak i koje se obavljaju s povećanim intenzitetom. Kod manjih projekata faze možemo poistovjetiti s odgovarajućim dominantnim aktivnostima, pa dobivamo proces sličan vodopadu.

U nastavku detaljnije opisujemo faze u UP-u, njihove ciljeve, naglaske i međaše:

- *Inception*. Ciljevi su: dokazati izvedivost i isplativost projekta, utvrditi ključne zahtjeve da bi se vidio kontekst (doseg) sustava, prepoznati rizike. Naglasak je na temeljnim aktivnostima: zahtjevi i analiza. Naziv međaša je: *Life Cycle Objectives*.
- *Elaboration*. Ciljevi su: stvoriti izvršivu jezgru arhitekture sustava. Profiniti procjenu rizika, definirati atribute kvalitete, evidentirati slučajeve uporabe (*use case*) koji pokrivaju 80 % funkcionalnih zahtjeva, stvoriti detaljni plan za fazu konstrukcije. Naglasak je na temeljnim aktivnostima: zahtjevi, analiza te pogotovo oblikovanje. Naziv međaša je: *Life Cycle Architecture*.
- *Construction*. Ciljevi su: dovršiti zahtjeve, analizu i oblikovanje te dograditi jezgru arhitekture do konačnog sustava. Pritom treba očuvati integritet arhitekture i oduprijeti se pritiscima da se sustav dovrši na brzinu. Obaviti beta-test i pokrenuti sustav. Naglasak je na temeljnim aktivnostima: oblikovanje te pogotovo implementacija. Naziv međaša je: *Initial Operational Capability*.
- *Transition*. Ciljevi su: popraviti uočene pogreške, priredititi sustav za rad u korisničkoj okolini, distribuirati ga na korisnička radna mjesta, stvoriti korisničku dokumentaciju, organizirati podršku korisnicima, napraviti *post-project review*. Naglasak je na temeljnim aktivnostima: implementacija te pogotovo testiranje. Naziv međaša je: *Product Release.*

Dalje opisujemo temeljne aktivnosti u UP-u:

- *Zahtjevi* (*requirements*). Utvrđuje se što sustav treba raditi.
- *Analiza* (*analysis*). Zahtjevi se analiziraju, profinjuju i strukturiraju.
- *Oblikovanje* (*design*). Predlaže se građa sustava koja će omogućiti da se zahtjevi realiziraju.
- *Implementacija* (*implementation*). Stvara se softver koji realizira predloženu građu.
- *Testiranje* (*test*). Provjerava se radi li stvoreni softver zaista onako kako bi trebao.

Iako je nastala relativno nedavno, UP je po svojim osobinama ipak klasična metoda. Naime, ona razvoj softvera promatra kao kompleksan projekt koji ima svoj početak i kraj i kojim treba upravljati. Također, ona predviđa da će se svi rezultati dokumentirati, npr. s pomoću odgovarajućih UML dijagrama.

www.element.hr

1.3.3. Primjer agilne metode

Kao primjer agilne metode razvoja softvera opisat ćemo *Extreme Programming* – XP. Metoda je nastala oko 2000-te godine i autorstvo se pripisuje Kentu Becku. Detaljnije je opisana u knjizi [7]. Naziv je odabran zato što XP ideju iterativnog razvoja softvera s naglaskom na programiranje tjera do ekstremnih razmjera.

Razvojni proces u skladu sa XP-om prikazan je na slici 1.9. Proces se sastoji od kontinuiranog niza vrlo brzih iteracija. Svaka iteracija stvara novo izdanje sustava koje se isporučuje korisniku. Vremenski razmak između dvije isporuke nije veći od dva tjedna.

Slika 1.9. Jedna iteracija metode XP

Novo izdanje sustava može sadržavati novu funkcionalnost – tada XP podsjeća na model inkrementalnog razvoja za softverski proces. No moguće je da novo izdanje samo popravlja ili mijenja već postojeću funkcionalnost – tada XP liči na model evolucijskog razvoja. Dakle, XP kombinira inkrementalni i evolucijski model. No naglasak je na izuzetno kratkim i brzim iteracijama.

Zahtjevi u XP prikazuju se kao skup kratkih kartica teksta, takozvanih *korisničkih priča* (*user stories*). Na početku iteracije biraju se one priče koje će se implementirati u dotičnom izdanju – uzima se onoliko priča koliko se može implementirati u dva tjedna i to one s najvišim prioritetom. Ostale se priče ostavljaju za buduća izdanja. Odabrane se priče razgrađuju u manje zadatke koji se raspoređuju programerima u skladu s procijenjenom složenošću. Opisani neformalni postupak planiranja novog izdanja naziva se *planning game*.

U XP-u ne može doći do kašnjenja isporuke – ono što se do predviđenog roka isporuke uspjelo implementirati, to se isporučuje, ostalo se ostavlja za buduća izdanja. Naručitelj programerima plaća utrošeno radno vrijeme. Nema ugovora koji bi vezao plaćanje uz traženu funkcionalnost.

U metodi XP iznimno važnu ulogu igraju korisnici. Postoji predstavnik korisnika koji je član razvojnog tima i stalno je dostupan ostalim članovima. On donosi korisničke priče, određuje prioritete, dogovara se o sadržaju idućeg izdanja, sudjeluje u testiranju sustava. Također, predstavnik korisnika odlučuje što će se napraviti s

www.element.hr

nerealiziranim pričama te treba li neku od već implementiranih priča promijeniti i ponovno implementirati.

Zanimljiva inovacija koju je XP donio u programersku praksu naziva se *razvoj gdje test ide prvi* (*test-first development*). Pri realizaciji bilo kojeg programerskog zadatka najprije se sastavljaju testovi za dotični novi komad funkcionalnosti, a tek onda se ta funkcionalnost ide implementirati. Čim je novi programski kod napisan, on se odmah testira. Da bi ovakav način rada bio efikasan, nužan je odgovarajući CASE-alat koji omogućuje lagano zadavanje, pohranjivanje te automatsko izvršavanje testova.

Testovi dobiveni primjenom *test-first developmenta* trajno se pohranjuju. Kad god se stvori nova verzija softvera, nad njom se osim najnovijih testova također moraju izvršiti i svi prijašnji testovi. Time se osigurava da najnovije promjene nisu narušile staru funkcionalnost.

Druga inovacija koju je XP donio u programersku praksu je *programiranje u paru* (*pair programming*). Dakle, jedan zadatak ne obavlja jedan programer nego dvojica – oni sjede za istom radnom stanicom i zajedno pišu isti programski kod. Prednost takvog načina rada je da dvojica suradnika u zajedničkoj diskusiji lakše dolaze do dobrog rješenja te jedan drugom ispravljaju pogreške. Parovi nisu fiksirani, već se mijenjaju od zadatka do zadatka. Time se postiže *kolektivno vlasništvo* nad programskim kodom, gdje su svi upućeni u sve dijelove programa i sve mogu promijeniti.

Još jedna osobina XP-a je da se softver oblikuje tako da podrži trenutačne zahtjeve i ništa više od toga. Takvo oblikovanje je u suprotnosti s principima tradicionalnog softverskog inženjerstva, gdje se softver nastoji izgraditi dovoljno općenito da bi se u budućnosti lako mogao mijenjati. Pobornici XP-a smatraju da se promjene ionako ne mogu predvidjeti, pa se ne isplati o njima brinuti unaprijed.

Opasnost kod bilo koje vrste iterativnog razvoja softvera je da se struktura softvera postupno degradira uslijed stalnih promjena. XP se bori protiv te opasnosti povremenim reorganiziranjem programskog koda, takozvanom *refaktorizacijom* (*refactoring*). Dakle, kad programer primijeti da se struktura programa previše narušila, on je popravlja tako da pojednostavi ili premjesti neke dijelove koda, eliminira ponavljanje sličnih dijelova, itd.

Metoda XP ne propisuje nikakve oblike dokumentacije. Naime, prema zagovornicima XP-a, jednostavan i pregledno napisani programski kod sam sebe najbolje dokumentira. Štoviše, iz tog koda moguće je automatski generirati razne izvještaje i dijagrame kakvima se inače koristimo u dokumentima. Ovakav je pristup realističan u situaciji kad se nove verzije softvera proizvode gotovo svakodnevno – ažuriranje zasebne dokumentacije predstavljalo bi preveliki teret i bilo bi podložno pogreškama.

Iz svega izloženog vidimo da XP ima sve osobine agilne metode za razvoj softvera. Brojni softverski timovi u cijelosti su prihvatili XP i pokazali da agilni način rada može biti efikasan barem kod manjih projekata s vrlo promjenjivim zahtjevima. No, još je veći broj onih timova koji su izabrali samo neke od XP-ovih značajki, npr. razvoj gdje test ide prvi ili programiranje u paru te ih uklopili u neku klasičnu metodu.

www.element.hr
www.element.hr