Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Miha Klun

Standardi in metode za specifikacijo zahtev programske opreme

diplomsko delo

na univerzitetnem študiju računalništva in informatike

Mentor: izr. prof. dr. Viljan Mahnič

Ljubljana, 2014

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Opišite različne pristope, ki se uporabljajo za specifikacijo zahtev pri razvoju programske opreme: standard IEEE 830-1998, primere uporabe in uporabniške zgodbe. Uporabo vsakega od naštetih pristopov prikažite na primeru iz realnega sveta, npr. pri razvoju študijskega informacijskega sistema. Omenjene pristope primerjajte med seboj in analizirajte njihove dobre in slabe lastnosti.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Miha Klun, z vpisno številko **63060122**, sem avtor diplomskega dela z naslovom:

Standardi in metode za specifikacijo zahtev programske opreme

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Viljana Mahniča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 12. junija 2014 Podpis avtorja:

Iskreno se zahvaljujem mojemu mentorju, izr. prof. dr. Viljanu Mahniču, za vso pomoč in napotke, ki sem jih med pisanjem velikokrat potreboval, ter vestno pregledovanje mojega dela.

Zahvaljujem se tudi staršem, ki so tako potrpežljivo čakali na ta izdelek in Tjaši, ki je vedno verjela vame.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Predstavitev in uporaba metod **3**

2.1 Teoretični del 3

2.1.1 Opis funkcionalnih zahtev po metodi, ki jo priporoča standard IEEE 830 3

2.1.2 Metoda primerov uporabe 10

2.1.3 Metoda uporabniških zgodb 20

2.2 Praktični del 41

2.2.1 Standard IEEE 830 41

2.2.2 Metoda primerov uporabe 44

2.2.3 Metoda uporabniških zgodb 50

3 Analiza in primerjava metod **53**

3.1 Popolnost specifikacij 53

3.2 Razumljivost specifikacij za končnega uporabnika sistema . . . 54

3.3 Vidik razvijalcev 55

3.4 Dopolnjevanje specifikacije 56

3.5 Čas izdelave, kompromisi in omejitve uporabe 57

3.6 Splošna primerjava 58

4 Zaključek **61**

Literatura **63**

Dodatek: Specifikacija zahtev programske opreme po standardu IEEE 830-1998 **65**

Povzetek

V tem diplomskem delu primerjam tri izbrane standarde in metode za pisanje specifikacij zahtev programske opreme. Metoda specifikacij IEEE 830 in metoda primerov uporabe predstavljata starejšo, medtem ko uporabniške zgodbe predstavljajo novejšo generacijo metod za pisanje specifikacij.

Vsaka od metod je najprej predstavljena teoretično, nato pa še na praktičnem primeru. Za praktični primer smo vzeli kar študentom na Fakulteti za računalništvo in informatiko dobro poznani sistem E-študent. Seveda pa ni opisan cel, ampak samo delček sistema, saj nam to za potrebe primerjave popolnoma zadošča. Teoretični in praktični predstavitvi metod pa sledi še primerjava, kjer tematsko primerjamo posamezne metode med seboj in tako prikažemo največje razlike med njimi.

Ključne besede: specifikacija zahtev programske opreme, IEEE 830, primeri uporabe, uporabniške zgodbe, E-študent.

Abstract

This thesis presents the comparison between three selected standards and methods for software requirements specifications. IEEE 830 specification and use cases represent older, while user stories represent newer generation of methods for specification writing.

Each method is first explained in theory and then on a practical example. E-študent, a well-known application to the students of the Faculty of computer and information science, serves as our practical example. The application is only partially described, since it suffices for our comparison. The theoretical and practical presentation of the methods is followed by the comparison, in which the three methods are being compared theoretically to emphasize the biggest differences between them.

Keywords: software requirements specification, IEEE 830, use cases, user stories, E-študent.

Poglavje 1

Uvod

V tem diplomskem delu bom teoretično in praktično predstavil ter primerjal tri različne standarde in metode za specifikacijo zahtev programske opreme in sicer standard IEEE 830, primere uporabe in uporabniške zgodbe.

Kaj pa sploh so specifikacije zahtev programske opreme? Specifikacije zahtev programske opreme so opis obnašanja sistema v razvoju. Te specifikacije so nekakšna navodila za razvijalce o tem, kako naj implementirajo zahtevano programsko opremo, zato je pomembno, da pri vsakem projektu izberemo najbolj ustrezno metodo. Vsaka izmed metod ima svoje prednosti in slabosti, sami pa se moramo odločiti, katere lastnosti so za naš projekt najpomembnejše.

Čisto za začetek pa nekaj zgodovinskih dejstev o izbranih metodah.

Zgodovina primerov uporabe sega v leto 1986, ko je Ivar Jacobson uporabil besedno zvezo "primer uporabe" za strukturiran opis obnašanja celotnega sistema v različnih pogledih [3]. Leta 1992 objavljena knjiga *Object-Oriented Software Engineering - A Use Case Driven Approach* [4], katere soavtor je bil, je veliko pripomogla k razmahu uporabe primerov uporabe. Od takrat naprej pa je k razvoju te tehnike prispevalo veliko avtorjev, eden izmed njih je tudi Alistair Cockburn [1]. Po njegovih predlogah so opisani tudi primeri uporabe v tej diplomski nalogi.

Osnutki uporabniških zgodb so se prvič pojavili leta 1998 pri ekstremnem

programiranju, bili so del procesa načrtovanja iteracije in poimenovani "tako kratki primeri uporabe, da jih lahko napišemo na kartice" [6]. Kasneje so se v literaturi čedalje bolj oddaljevali od primerov uporabe, dokler ni leta 2001 Ron Jeffries [5] predlagal sedaj dobro poznani koncept kartice, pogovora in potrditev (angl. Card, Conversation, Confirmation), ki zaobjema vse dele uporabniške zgodbe.

O nastanku IEEE 830 ni znanega veliko. Dokument je prvič kot vodič za pisanje specifikacij programske opreme izšel že leta 1984 [7]. Dopolnjen je bil leta 1994 [8], zadnja različica pa je izšla leta 1998 [9].

Osrednji del diplomske naloge bo sestavljen iz dveh poglavij (2. in 3. poglavje).

Poglavje 2 bo sestavljeno iz teoretičnega (2.1) in praktičnega (2.2) dela. Teoretični del bo razdeljen na tri podpoglavja, v vsakem od njih pa bom predstavil eno metodo za specifikacijo zahtev programske opreme. V prvem podpoglavju (2.1.1) bom predstavil metodo IEEE 830, v drugem (2.1.2) metodo primerov uporabe, v zadnjem (2.1.3) pa metodo uporabniških zgodb. Tudi praktični del (2.2) je razdeljen na več podpoglavij (2.2.1, 2.2.2 in 2.2.3), v katerih bom prikazal praktično uporabo vseh treh metod na primeru spletne aplikacije E-študent.

V zadnjem poglavju osrednjega dela, poglavju 3 pa bom vse tri na konkretnem primeru uporabljene metode analiziral in primerjal med sabo.

Poglavje 2

Predstavitev in uporaba metod

2.1 Teoretični del

2.1.1 Opis funkcionalnih zahtev po metodi, ki jo priporoča standard IEEE 830

Predstavitev IEEE 830

Tukaj opisana navodila za izdelavo IEEE 830 so povzeta po [9].

IEEE 830 opisuje pristope za specifikacijo zahtev programske opreme. Rezultat tega procesa je nedvoumen in celosten dokument specifikacij. Naročnikom programske opreme pomaga izraziti želje, razvijalcem razumeti želje kupcev, posameznikom pa doseči določene cilje (razvoj specifikacij programske opreme za lastno organizacijo, določitev formata in vsebine specifikacij programske opreme ter preverjanje kvalitete dokumenta).

Dokument IEEE 830 nosi naslov Specifikacija zahtev programske opreme, vendar ga bomo v tem podpoglavju imenovali kar IEEE 830, da ne bi prihajalo do nejasnosti, saj so pravzaprav vse tri opisane metode specifikacije zahtev programske opreme.

Obseg IEEE 830

IEEE 830 opisuje vsebino in lastnosti dobro napisane specifikacije programske opreme. Uporablja se pri določanju zahtev programske opreme, pomaga pa lahko tudi pri nekaterih programskih produktih za interno ali komercialno uporabo. V tem dokumentu sta opisana proces razvoja in vsebina produkta.

Napotki za pisanje

Specifikacija zahtev programske opreme IEEE 830 je specifikacija za določen programski izdelek, program ali skupino programov, ki v nekem okolju opravljajo določene funkcije. IEEE 830 lahko pišejo predstavniki razvijalcev, predstavniki naročnikov ali oboji. Pisci naj se osredotočijo na funkcionalnost, zunanje vmesnike, zmogljivosti, značilnosti (prenosljivost, vzdrževanje ...) in na implementacijo vezane omejitve.

Ker ima IEEE 830 v procesu razvijanja programske opreme določeno vlogo, morajo biti pisci tega dokumenta previdni, da ne presežejo svojih zadolžitev. Dokument mora definirati vse zahteve programske opreme, ne sme opisovati podrobnosti načrtovanja in implementacije ter ne sme vsiljevati dodatnih omejitev programski opremi.

IEEE 830 naj bo:

- **Pravilen:** IEEE 830 je pravilen, če in samo če programska oprema zadosti vsaki opisani zahtevi. Pravilnost lahko potrdimo z drugo dokumentacijo projekta, drugimi standardi ali pa jo potrdi kar kupec oziroma uporabnik.

- **Nedvoumen:** IEEE 830 je nedvoumen, če in samo če ima vsaka zahteva, ki je zapisana, samo eno interpretacijo. To dosežemo tako, da vsako karakteristiko opišemo vsaj z istim izrazom. Dvoumnosti se lahko izognemo tudi tako, da dokument pregleda še nekdo drug ali pa ga pišemo v posebnem, specifikaciji namenjenem jeziku.

- **Popoln:** IEEE 830 je dovršen, če in samo če obravnavamo vse zunanje

zahteve, ki jih zahteva sistemska specifikacija, definiramo vse odzive tako na pravilne kot na nepravilne vnose in se v besedilu sklicujemo na vse slike, razpredelnice in diagrame.

- **Konsistenten:** IEEE 830 je dosleden, če in samo če si posamezne enake zahteve v različnih delih besedila ne nasprotujejo.

- **Razvrščen glede na pomembnost:** IEEE 830 je razvrščen glede na pomembnost, če in samo če ima vsaka opisana zahteva oznako, s katero označujemo pomembnost te zahteve. Označujemo jih lahko z oznakami *nujno, pogojno* in *neobvezno.*

- **Preverljiv:** IEEE 830 je preverljiv, če in samo če za vsako zapisano zahtevo človek ali računalnik v končnem stroškovno učinkovitem procesu lahko preveri, ali je izpolnjena.

- **Prilagodljiv:** IEEE 830 je prilagodljiv, če in samo če je struktura zahtev taka, da lahko njihove spremembe pišemo enostavno, popolno in dosledno, pri tem pa ohranimo strukturo in stil.Pri tem je pomembno predvsem izogibanje podvajanju, ki ob spremembah lahko privede do nekonsistentnosti.

- **Sledljiv:** IEEE 830 je izsledljiv, če je izvor vsake zahteve jasen in nam sklicevanje na te zahteve lajša nadaljnji razvoj in dopolnjevanje dokumenta.

Priporočljivo je, da razvijalci, naročniki in uporabniki sodelujejo pri pisanju dokumenta, saj ga bodo le tako razumeli vsi. Pomanjkanje izkušenj uporabnikov in naročnikov pri problematiki razvoja lahko omilimo z izdelovanjem prototipov, saj tako pridemo do novih vprašanj in odgovorov nanje.

Pri pisanju IEEE 830 se poskušamo čim bolj izogibati načrtovanju sistema, saj si s tem omejujemo možnosti za alternativno oblikovanje pri implementaciji. Pri odločitvah, kot so na primer, postavljanje določenih funkcij v ločene module, omejevanje komunikacije med deli programa in preverjanje

integritete kritičnih spremenljivk pa brez določanja oblike sistema vseeno ne gre.

Deli IEEE 830

V tem poglavju bomo govorili o zgradbi IEEE 830. Ni nujno, da uporabljamo ravno v priporočilu za pisanje IEEE 830 navedeno kazalo poglavij, vendar pa moramo vedno navesti vse ključne informacije.

a) Uvod (prvi razdelek dokumenta): Uvod IEEE 830 naj nam da splošno sliko dokumenta in naj vsebuje naslednje informacije:

- **Namen:** podpoglavje naj določi namen in ciljno publiko IEEE 830.
- **Obseg:** podpoglavje naj poimenuje programski izdelek, pojasni, kaj bo oziroma česa ne bo opravljal, navede koristi in cilje uporabe aplikacije in naj bo dosleden pri podobnih trditvah.
- **Definicije, kratice in okrajšave:** podpoglavje naj definira vse termine, kratice in okrajšave, potrebne za razumevanje IEEE 830.
- **Viri:** podpoglavje naj našteje vse vire, na katere se sklicujemo v IEEE 830, in naj navede naslov dokumenta, datum in založbo ter pove, kje je vir dosegljiv.
- **Pregled:** podpoglavje naj pove, kako je preostanek IEEE 830 sestavljen in kaj vsebuje.

b) Splošni opis (drugi razdelek dokumenta): v tem razdelku navedemo splošne dejavnike, ki vplivajo na programski izdelek in njegove zahteve. Naštevamo samo okvirne zahteve, konkretne pa naštejemo v tretjem razdelku dokumenta. Razdelek vsebuje naslednja podpoglavja:

- **Kontekst programskega izdelka:** podpoglavje programski izdelek predstavi v povezavi s podobnimi izdelki in naj opiše, kako se sistem

obnaša pod določenimi omejitvami. Vse možne omejitve so naštete in opisane v nadaljevanju. Pri *sistemskih vmesnikih* naštejemo vse vmesnike in določimo delovanje programskega izdelka, pri tem pa moramo paziti na to, da se sistemske zahteve in opis vmesnika ujemajo. *Uporabniški vmesniki* opisujejo logične značilnosti vsakega vmesnika med programskim izdelkom in njegovimi uporabniki, kot so na primer oblika okna, določanje bližnjic ... Določajo tudi vse možnosti za optimizacijo vmesnika v sodelovanju z uporabnikom. To poglavje je preprost seznam opravil, ki jih vmesnik omogoča oziroma ne omogoča. *Strojni vmesniki* določajo logične značilnosti vsakega vmesnika med programskim izdelkom in strojno opremo. Tukaj naštejemo na primer podprte naprave, razne protokole, število vrat in podobno. *Programski vmesniki* definirajo vso ostalo potrebno programsko opremo in vmesnike z drugimi aplikacijami. Ne smemo pozabiti napisati tudi vseh zahtevanih podrobnosti. Pri *komunikacijskih vmesnikih* določimo vse vmesnike za komunikacijo. Pri *spominskih omejitvah* navedemo vse omejitve glavnega in pomožnih pomnilnikov. Naštejemo vse *operacije*, ki jih zahteva uporabnik (načini operacij, dolžina operacij,...) *Prilagoditve ob posameznih namestitvah* določajo vse zahteve, ki so značilne za sistem in funkcije, ki jih bomo izboljšali.

- **Funkcije izdelka:** podpoglavje zaobjame glavne funkcije, ki jih bo sistem opravljal. Funkcije naj bodo opisane tako, da bodo razumljive naročniku oziroma uporabniku, pri tem lahko uporabimo diagrame.

- **Značilnosti uporabnikov:** podpoglavje opiše splošne značilnosti uporabnikov, katerim je programski izdelek namenjen (stopnja izobrazbe, izkušnje, ...) in pove, zakaj so te značilnosti pomembne.

- **Omejitve:** podpoglavje na splošno našteje dejavnike, ki lahko omejujejo programski izdelek, kot so na primer: omejitve strojne opreme, vmesnike za druge aplikacije, vzporedne operacije, nadzorne funkcije, stopnja zanesljivosti,...

- **Predpostavke in odvisnosti:** podpoglavje našteje vse dejavnike, ki vplivajo na zahteve, navedene v IEEE 830, kot je na primer prilagoditev operacijskemu sistemu, na katerem bo tekel program.
- **Porazdelitev zahtev:** podpoglavje vsebuje zahteve, ki bodo izpolnjene v kasnejših verzijah programskega izdelka.

c) Specifične zahteve (tretji razdelek dokumenta): v tretjem razdelku naštejemo vse zahteve programskega izdelka tako natančno, da lahko izdelek, ki tem zahtevam zadosti, programerji napišejo, testerji pa testirajo. Vsaka zahteva vsebuje vsaj opis vsakega vhoda in odziva sistema ter vseh funkcij, ki jih sistem opravlja. Ta razdelek je najobsežnejši in najpomembnejši del IEEE 830, zato morajo posamezne zahteve: biti v skladu z napotki za pisanje (gl. podpoglavje *Napotki za pisanje*), se ujemati s starejšimi dokumenti, biti enolično določljive in urejene tako, da je dokument čim bolj čitljiv.

V nadaljevanju bomo opisali sestavne dele teh zahtev:

- **Zunanji vmesniki:** tukaj natančno opišemo vse vhode in izhode, pri tem pa pazimo na to, da ne ponavljamo opisov iz drugega razdelka. Vključimo sledeče podatke: ime vmesnika, namen, izvor vhoda in cilj izhoda, dopustne vrednosti, merske enote, natančnost, oblike podatkov...
- **Funkcije:** tukaj definiramo osnovne operacije, ki jih opravlja programska oprema. Stavki se ponavadi začnejo s "Sistem mora..." in naj vsebujejo: preverjanje vhodnih podatkov, točno zaporedje operacij, napake in njihovo obravnavanje, vpliv parametrov, razmerja med vhodi in izhodi.
- **Zahteve delovanja:** to podpoglavje definira statične in dinamične številčne zahteve, ki se nanašajo na sistem oziroma uporabnika in vključujejo: število podprtih terminalov, število hkratnih uporabnikov ter količino in tip obravnavanih podatkov.

- **Logične zahteve podatkovne baze:** tukaj določimo vse logične zahteve za informacije, ki jih vpisujemo v bazo in vsebujejo: tipe informacij, ki jih uporabljajo različne funkcije, pogostost uporabe, dostop do zmogljivosti, entitete in razmerja med njimi,...

- **Načrtovalske omejitve:** tukaj opišemo vse omejitve, ki nam jih postavljajo drugi standardi, strojna oprema,... V podpoglavju *Združljivosti s standardi* opišemo omejitve obstoječih standardov, ki jih moramo upoštevati: oblika poročil, naštevanje podatkov, računovodski postopki, sledenje za revizijo.

- **Lastnosti programskega sistema:** tu opišemo vse tiste značilnosti sistema, ki služijo kot zahteve in jih lahko objektivno preverimo. To so: *zanesljivost* (dejavniki, ki vplivajo na zanesljivost sistema), *dosegljivost* (dejavniki, ki vplivajo na dosegljivost sistema – kontrolne točke, obnovitev in ponovni zagon), *varnost* (dejavniki, ki preprečujejo nenamerni in zlonamerni dostop do sistema – uporaba kriptografije, dnevnikov, preverjanje podatkovne integritete), *vzdrževanje* (dejavniki, ki pripomorejo k lažjemu vzdrževanju) in *prenosljivost* (dejavniki, ki omogočajo enostavnejšo prenosljivost – odstotek od gostitelja odvisne kode, uporaba prenosljivega programskega jezika).

- **Organizacija posameznih zahtev:** zahteve so lahko organizirane na različne načine. Pri vsakem projektu posebej moramo dobro premisliti, kateri način bomo izbrali, saj vsak projekt zahteva svojo rešitev (za različne predloge za organizacijo zahtev gl. [9])

- **Dodatni komentarji:** Če uporabljamo več v IEEE 830 navedenih metod, tukaj definiramo, kako so te metode med sabo povezane.

č) Dodatki dokumentu: k dodatkom štejemo kazalo vsebine, indeks in priloge. Kazalo vsebine in indeks imata standardno obliko, k prilogam pa štejemo primere vhodov in izhodov programskega izdelka, izsledke študij in

anket, dodatne informacije, ki so v pomoč bralcu. Kadar vključujemo priloge, moramo izrecno povedati, ali so tudi te del IEEE 830.

2.1.2 Metoda primerov uporabe

V tej nalogi je opis metode primerov uporabe povzet po Cockburnu [1].

Primer uporabe zajema dogovor med deležniki (angl. stakeholders) o delovanju sistema. Primer uporabe opiše obnašanje sistema pod različnimi pogoji, ko se sistem odzove na zahtevo interesenta, imenovanega glavni akter (angl. primary actor). Glavni akter začne interakcijo s sistemom zato, da doseže določen cilj. Sistem zaščiti interese vseh interesentov. Različna sosledja dogodkov se lahko odvijajo glede na zahtevo, ki je bila narejena in glede na njene pogoje, primer uporabe pa zajame vse te različne poteke. Primeri uporabe so večinoma v tekstovni obliki, lahko pa jih zapišemo tudi v obliki diagramov, Petrijevih mrež ali s programskimi jeziki. Ker služijo za komunikacijo med ljudmi, je ponavadi izbrana tekstovna oblika. Primere uporabe lahko pišemo z različnimi nivoji strogosti in podrobnosti, a osnovna pravila pisanja so za vse različice enaka.

Kadar primeri uporabe dokumentirajo poslovni proces organizacije, je sistem v obravnavi organizacija sama. Interesenti so delničarji, stranke, prodajalci ... Glavni akterji so lahko tudi kupci in morda dobavitelji. Pri primerih uporabe, ki opisujejo obnašanje programske opreme, je sistem v obravnavi računalniški program. Deležniki so ljudje, ki uporabljajo program, podjetje, ki si ga lasti, vladne agencije in drugi računalniški programi. Glavni akter je uporabnik, ki sedi pred zaslonom.

Dober primer uporabe je lahko berljiv. Sestavljen je iz stavkov, ki opisujejo posamezne korake, v katerih akter doseže rezultat ali preda informacijo drugemu akterju. Pri pisanju primerov uporabe moramo upoštevati tri koncepte:

- **Doseg**: kaj je sistem v obravnavi?
- **Glavni akter**: kdo ima cilj?

- **Stopnja**: kako visoko oziroma nizko raven ima cilj?

Primeri uporabe so oblika pisanja, ki jo lahko uporabimo v različnih okoliščinah, in sicer pri opisovanju poslovnega procesa, funkcionalnih zahtev sistema, dokumentiranju zasnove sistema, spodbujanju razprave o zahtevah novega sistema, vendar ne pri opisovanju zahtev.

V vsaki situaciji uporabimo drugačen stil pisanja. Tukaj je naštetih nekaj oblik primerov uporabe:

- **Neformalne** (angl. casual) primere uporabe uporabljajo manjše skupine, ki zbirajo zahteve, in večje skupine, ki o njih razpravljajo; **formalne** (angl. fully dressed) primere uporabe pa uporabljajo večje oziroma uradno naravnane skupine.

- **Poslovne** primere uporabe uporabljajo poslovneži za opis poslovnih procesov; **sistemske** primere uporabe pa razvijalci programske in strojne opreme.

- Glede na količino podrobnosti, ki jih želimo, opišemo **sumarni cilj** (angl. summary goal), **uporabniški cilj** (ena seja) ali **podfunkcijo** (del uporabniškega cilja).

- Pri pisanju zahtev pišemo primere uporabe po načelu **črne škatle** (angl. black box), ki se ne spuščajo v zgradbo sistema ali primere uporabe po načelu **bele škatle** (angl. white box), ki pokažejo, kako tečejo notranji procesi.

Izbira oblike ni vedno enostavna in pričakovati je, da bomo pri tem imeli težave. Največje razlike vidimo pri stopnji formalizacije, ki jo lahko razložimo na primerih:

- Skupina dela na velikem, poslovno kritičnem projektu. Člani skupine se odločijo, da se splača porabiti več časa za pisanje zahtev, zato mora biti predloga za primere uporabe podrobnejša, napisana v enakem stilu, da ne bi bilo dvoumnosti in nesporazumov, pregledi pa naj bodo strogi, da se česa ne izpusti.

- Skupina petih ljudi piše sistem, v katerem napaka v razvitem sistemu nima velikega vpliva. Ni vredno zapravljati preveč časa za pisanje zahtev, zato se odločijo za preprostejšo predlogo, večjo variacijo pisalnega stila in manj pregledov z večjo toleranco napak na primerih uporabe.

Nobena odločitev ni napačna, vsak projekt ubere svojo pot. Naredimo lahko le to napako, da se odločimo za preveč rigorozen način, kadar ta ni potreben, in si tako ustvarimo prevelike stroške. V splošnem sta dovolj dve predlogi z različno stopnjo formalizacije, ki ju po potrebi prilagodimo.

Primeri uporabe so specifikacije zahtev programske opreme, določenih specifikacij pa se ne da napisati v obliki primerov uporabe, kot so na primer zunanji vmesniki, podatkovni tipi, poslovna pravila in zapletene formule.

Primer uporabe kot dogovor o obnašanju sistema

V tem poglavju bomo govorili o primerih uporabe kot dogovorih o obnašanju sistema.

Najprej bomo na primerih razložili, kaj so **cilji** akterja. Ko receptor v hotelu prejme klic, je njegov cilj, da ga računalnik sprejme in začne zahtevo, odgovornost sistema pa je enaka. Za izvrševanje te odgovornosti sistem oblikuje podcilje. Nekatere podcilje lahko izpolni sam, pri izpolnjevanju drugih pa potrebuje pomoč podpornih akterjev (angl. supporting actor), kot so tiskalnik, druga organizacija itd.

Podcilje lahko delimo na dodatne podcilje v neskončnost, pri tem pa moramo paziti, da jih ne razvijemo preveč. Uslužbenec mora imeti rezervni načrt v primeru, ko sistem ne more izpolniti obveznosti, npr. list papirja in pisalo. Če sistem naleti na napako v enem izmed svojih podciljev, jo lahko popravi ali pa cilj opusti (npr. dvig prevelike količine gotovine iz bankomata).

Doseganje cilja poteka po zaporedju sporočil, ki ga imenujemo **potek**, opisane korake pa lahko ločujemo ali združujemo po potrebi. Vsak korak ali interakcija zajema cilj. Obnašanje sistema na visokem nivoju opišemo z zgoščenimi cilji in interakcijami, z ločevanjem pa opišemo delovanje sistema

z željeno natančnostjo. Prihodnje interakcije opisujemo z množicami zaporedij, ki jim določimo pogoj, pod katerim se zgodijo. Tudi te množice lahko ločujemo in združujemo po potrebi.

Povedati je treba še, da se vse interakcije primera uporabe nanašajo na cilj istega akterja in da se primer uporabe začne, ko ga sproži dogodek, in se nadaljuje, dokler ni cilj dosežen ali opuščen in dokler sistem ne izpolni obveznosti do interakcije.

Vsak potek opiše zaporedje korakov, ki nam povedo kako se bodo posamezne interakcije odvile, primeri uporabe pa vse te različne poteke zberejo. Segment, v katerem se nahajajo poteki, je razdeljen na dve veji. V prvi so uspešni, v drugi pa neuspešni poteki. Primeri uporabe vsebujejo poteke, ti pa so sestavljeni iz podprimerov uporabe (korakov).

Da bi lažje razložili notranje obnašanje sistema, vpeljemo še model deležnikov in interesov, ki nam pove, kaj bomo v primer uporabe vključili oziroma iz njega izpustili. Ko sistem teče, je prisoten samo glavni akter, ne pa vsi deležniki oziroma obstranski akterji. Primer uporabe kot dogovor o obnašanju sistema v celoti zajame le obnašanje, ki se tiče deležnikov. Da končamo primer uporabe, naštejemo vse člane deležnikov, njihove interese glede na primer uporabe in določimo, kaj zanje pomeni uspešen zaključek primera uporabe ter kakšna zagotovila pričakujejo od sistema. Za zadovoljitev vseh interesov deležnikov opišemo tri vrste akcij:

- interakcijo med dvema akterjema (za dosego cilja),
- potrjevanje (za zaščito akterja),
- notranjo spremembo stanja (v imenu akterja).

Obseg

Z besedo **obseg** (angl. scope) označujemo vse, kar načrtujemo sami. Orodje in/out seznam uporabljamo, ko razpravljamo o obsegu. Sestavljen je iz treh stolpcev, v prvem je napisana tema, za katero nismo prepričani, če sodi v obseg, v drugem in tretjem stolpcu pa sta kategoriji notri (angl. in) ter zunaj

(angl. out). Kadar nismo prepričani, če tema spada v obseg, jo napišemo v tabelo in za mnenje o tem vprašamo sodelujoče. Tabela nam je v pomoč pri nadaljnjem načrtovanju sistema, kadar se nam zazdi, da je razprava zašla s poti.

Funkcionalni obseg se nanaša na usluge, ki jih nudi sistem in ki bodo sčasoma opisane s primeri uporabe. Funkcionalni obseg določamo hkrati z razpoznavo primerov uporabe, saj sta ti dve nalogi neločljivo povezani. Pri tem nam poleg in/out seznama pomagata še dve drugi orodji, seznam akter-cilj in kratek pregled vseh primerov uporabe.

Seznam akter-cilj vsebuje cilje uporabnika, ki jih podpira sistem. Vsebuje samo usluge, ki jih bo sistem podpiral. Sestavljen je iz treh stolpcev, v prvega napišemo imena glavnih akterjev oziroma akterjev s cilji, v drugega cilje vsakega akterja z ozirom na sistem, v zadnjega pa prioriteto oziroma predviden časovni okvir izida, v katerem bo sistem podpiral cilj. Seznam posodabljamo sproti, zato da odraža stanje našega dela. Ta seznam je začetna točka pogajanj med predstavniki uporabnikov, finančnih sponzorjev in med razvijalci. V njem se osredotočimo na oris in vsebino projekta.

Seznam akter-cilj je najnižja stopnja natančnosti opisa sistema. Naslednja, še natančnejša stopnja opisa sistema je **tipični potek** oziroma kratka razlaga primera uporabe. To je dva- do šeststavčni opis obnašanja primera uporabe, ki kratko opisuje njegovo najpomembnejšo dejavnost in neuspehe. Uporablja se za ocenjevanje kompleksnosti dela. Kratka razlaga je lahko napisana v obliki tabele, dodatnega stolpca v seznamu akter-cilj ali pa je neposredno del jedra primera uporabe v prvem osnutku.

Načrtovalski obseg je skupek strojne in programske opreme, ki jo razvijamo ali načrtujemo. Funkcionalni obseg je določen s seznamom akter-cilj in kratko razlago primera uporabe, medtem ko je načrtovalski obseg tema vsakega primera uporabe. Vsi sodelujoči se morajo strinjati o načrtovalskem obsegu, napačna odločitev ima lahko katastrofalne posledice za pogodbo. Načrtovalski obseg je pomemben, ker vsebina ni vedno razvidna le iz imena primera uporabe ali glavnega akterja. Vsak primer uporabe označimo z la-

stnim načrtovalskim obsegom. Če na primer Telekom razvija sistem Nova aplikacija, ki vključuje podsistem Iskalec, so načrtovalski obsegi: Telekom (podjetje), Nova aplikacija (sistem) in Iskalec (podsistem).

V primeru večjih sistemov je dobro napisati sumarne primere uporabe (angl. summary-level use case), saj nam dajo dober pregled nad določeno množico primerov uporabe. Pokažejo nam, kako sistem koristi najbolj oddaljenim uporabnikom. Služijo lahko tudi kot kazalo vsebine obnašanja sistema.

Med načrtovanjem sistema lahko pride do določenih sprememb v načrtih in takrat se spremenijo skica načrtovalskega obsega, in/out seznam in seznam akter-cilj. Za pregled nad celotnim razvojem in načrtovanjem uvedemo še četrti element, vizijsko izjavo, ki nam pomaga tudi pri odločanju o tem, kaj spada in kaj ne spada v obseg. Tako se med načrtovanjem spreminja tudi ta element.

Deležniki in akterji

V tem poglavju se bomo posvetili **akterjem**. Akter je lahko neka oseba, podjetje, računalniški program ali računalniški sistem.

Deležnik je nekdo oziroma nekaj, ki ga zanima obnašanje primera uporabe. Vsak glavni akter je deležnik, nekateri deležniki pa ne sodelujejo neposredno s sistemom in torej niso akterji, čeprav jih zanima, kako se sistem obnaša.

Glavni akter je deležnik, ki potrebuje sistem, da mu naredi uslugo. Primer uporabe se začne, ko ga glavni akter kakorkoli sproži. Obstajata dva primera, ko glavni akter ni sprožilec primera uporabe in sicer, ko ga namesto njega sproži nekdo drug in ko je za to odgovoren časovni sprožilec.

Iskanje glavnih akterjev nam pomaga našteti vse cilje in videti splošno sliko sistema. Pri naštevanju glavnih akterjev se lahko cilji podvajajo, zato pri ponovnem pregledu dvojnike odstranimo. Naštevanje glavnih akterjev nam pomaga, ker nas osredotoči na uporabnike, postavi strukturo za seznam akter-cilj in nam pomaga pri deljenju primerov uporabe na manjše dele, ki jih lahko razdelimo med različne razvijalce. Glavne akterje lahko tudi natančneje

opišemo, zato da razvijalci sistem bolje prilagodijo uporabnikom.

Med razvijanjem sistema glavni akterji postanejo nepomembni, saj nas bolj zanima vsebina primera uporabe, kot pa njen uporabnik. Za zmanjšanje števila teh na začetek primera uporabe napišemo hierarhično lestvico akterjev. Glavni akterji zopet postanejo pomembni tik pred razpečevanjem sistema.

V primerih uporabe sta lahko prisotna tudi naslednja dva akterja:

- podporni akter, ki je zunanji akter in sistemu nudi uslugo (tiskalnik, spletna storitev);
- sistem v obravnavi, ki ga nazivamo z imenom in je opisan v načrtovalskem obsegu.

Sistem ponavadi obravnavamo kot črno škatlo, kjer interni akterji namenoma niso omenjeni. V nekaterih primerih pa sestavni deli sistema prevzamejo vloge akterjev, zato taki obravnavi sistema rečemo bela skrinjica.

Nivoji ciljev

V tem poglavju bomo opisali različne **nivoje ciljev in njihove oznake**. Cockburn [1] uporablja več različnih nivojev ciljev, mi pa se bomo omejili na tri najbolj uporabljane:

- Sumarni cilj: označimo ga tako, da imenu primera uporabe dodamo znak '+'.
- Uporabniški cilj: označimo ga tako, da imenu primera uporabe dodamo znak '!'.
- Podfunkcija: označimo jo tako, da imenu primera uporabe dodamo znak '-'.

Vsak v sumarnem cilju opisan korak je hkrati tudi uporabniški cilj. Ta nivo ciljev uporabljamo zato, da pokažemo kontekst, v katerem delujejo, zaporedje življenjskih ciklov sorodnih ciljev in kazalo nižjih nivojev primerov

uporabe. To, kar opisujejo, lahko traja od nekaj ur do nekaj let. Kot smo povedali že v podpoglavju *Obseg* tega poglavja, je na začetku pisanja primerov uporabe priporočljivo napisati nekaj sumarnih ciljev. V splošnem to naredimo tako, da določimo uporabniški cilj, glavnega akterja in obseg, poiščemo vse uporabniške cilje, ki imajo isti obseg in glavnega akterja, ter napišemo cilj primera uporabe.

Uporabniški cilj je najpomembnejši nivo primera uporabe, zato je priporočljivo, da je čim bolj razumljivo napisan. To je cilj, ki ga ima glavni akter, ko opravlja določeno delo. Taki primeri uporabe ponavadi trajajo do dvajset minut in so sestavljeni iz podfunkcij oziroma redkeje iz drugih uporabniških ciljev.

Podfunkcije so potrebne za izvrševanje uporabniških ciljev, napišemo jih samo takrat, kadar jih nujno potrebujemo zaradi bralnega razumevanja ali zato, ker jih uporabljajo mnogi drugi cilji. Pri pisanju takih primerov uporabe pazimo, da pri korakih po pomoti ne zaidemo v prevelike podrobnosti, kot je na primer "Pritisni enter".

Cockburn [1] priporoča, naj ima primer uporabe zaradi jasnosti in preglednosti dva do največ deset korakov. Priporoča tudi, naj uporabniški cilj iščemo tako, da se sprašujemo: "Kaj glavni akter hoče?" in "Zakaj glavni akter to sploh počne?"

Predpogoji, zagotovila in sprožilci

To poglavje bomo posvetili predpogojem, sprožilcem in zagotovilom.

Predpogoj je stanje, v katerem se mora sistem nahajati, preden se primer uporabe začne, in se med izvajanjem primera uporabe ne preverja ponovno. Predpogoj ponavadi označuje, da se mora pred izvedbo tega primera uporabe izvesti še nek drug primer uporabe.

Mininalno zagotovilo je najmanjše zagotovilo deležnikom predvsem takrat, ko ciljev glavnega akterja ne moremo izpolniti. V minimalnih zagotovilih ne naštevamo vseh načinov, na katere lahko primer uporabe ne uspe. Minimalna zagotovila morajo biti resnična po vsakem izvajanju primera upo-

rabe.

Zagotovilo uspeha našteva vse interese deležnikov, ko se primer uporabe uspešno zaključi s tipičnim potekom ali neko drugo alternativo. Zagotovila uspeha morajo biti resnična po uspešnem izvajanju primera uporabe.

Sprožilec je dogodek, ki sproži izvajanje primera uporabe. Sprožilec lahko izvajanje sproži že pred prvim korakom primera uporabe ali pa je kar sam prvi korak.

Poteki in koraki

V tem poglavju se bomo posvetili potekom in korakom. Vsak primer uporabe ima zgodbo, ki nam pokaže, kako sistem izpolnjuje oziroma opušča cilje uporabnika. Zgodba je sestavljena iz **tipičnega poteka** in njegovih **razširitev**.

Najprej napišemo tipični potek, ki je lahko razumljiv, v katerem je izpolnjen cilj glavnega akterja in ki zadosti zahtevam deležnikov. Tipični potek in vse njegove razširitve so del strukture, ki je sestavljena iz:

- pogoja, pod katerim se potek odvija,
- cilja, ki ga poskušamo doseči,
- akcijskih korakov,
- končnega pogoja,
- opcijskih razširitev.

Vsak korak v telesu poteka je interakcija med dvema akterjema, potrditev zahtev deležnika ali notranja sprememba, ki zadovolji interes deležnika. Akcijski koraki so enostavni dogodki, v katerih akter konča nalogo ali posreduje informacijo drugemu akterju. Časa začetka novega koraka ne pišemo, saj si koraki tesno sledijo.

Cockburn [1] priporoča, da se držimo naslednjih smernic:

- Struktura stavka koraka naj bo: osebek, povedek, predmet in predložna zveza.

- V koraku mora biti vedno jasno določeno, kdo je nosilec dejanja.
- Primer uporabe naj bo napisan v tretji osebi.
- V korakih ne opisujemo uporabniškega vmesnika.
- Koraki naj bodo enostavni za branje, zato jih raje razdelimo na ustrezno število odstavkov.
- V korakih ne pišemo pogojnih stavkov, ampak raje trdilne.
- Čas v korakih pišemo samo takrat, kadar je to potrebno.
- Uporabimo idiom "Uporabnik pove sistemu A, naj pridobi podatke od sistema B." To pomeni, da uporabnik določi začetni čas teh zaporednih dogodkov, idiom pa nam pokaže zadolžitve vseh treh sistemov.
- Ponavljanje in poljubno zaporedje korakov označimo po zadnjem koraku, za katerega to velja.

Razširitve

Poznamo več načinov pisanja razširitev, mi pa bomo v tem poglavju opisali samo način, ki ga priporoča Cockburn [1]. Pri tem načinu razvejimo korake za vsak izjemni dogodek, ki se lahko zgodi na tem koraku. Pravimo jim razširitve in ne neuspehi oziroma izjeme, saj zraven spadajo tudi alternativna izpolnjevanja cilja. Razširitve pišemo med zbiranjem zahtev, tako da razmislimo o vseh možnih izidih poteka.

Prav tako kot primer uporabe se tudi razširitev začne s pogojem, ki opisuje vzrok razširitve. Vsebuje zaporedje akcijskih korakov, ki opisujejo, kaj se pod tem pogojem zgodi, in se konča z doseženim oziroma opuščenim razširitvenim ciljem. Pogoj razširitve je pogoj, pod katerim sistem izbere drugo pot. Pri pisanju razširitev moramo paziti na to, da sistem zmore zaznati pogoj in obravnavati njegovo zaznavo.

Če je korak primera uporabe še en primer uporabe, ki ima napisane razširitve, v prvem primeru uporabe napišemo samo eno razširitev, ki povzame vse razširitve drugega primera uporabe.

Včasih se zgodi, da pri razširitvah potrebujemo tudi razširitve razširitev. Cockburn [1] priporoča, da te razširitve v besedilu zamaknemo en nivo nižje. Nižanje nivojev seveda ne more potekati v nedogled, zato v primeru velikega števila razširitev napišemo kar samostojen primer uporabe. Novemu primeru uporabe iz razširitve določimo primernega glavnega akterja in ustrezen nivo. Pri glavnem primeru uporabe se moramo zavedati dejstva, da lahko nov primer uporabe ne uspe, zato lahko napišemo pogoj uspeha in/ali neuspeha.

Cockburnovi [1] smernici za pisanje razširitev:

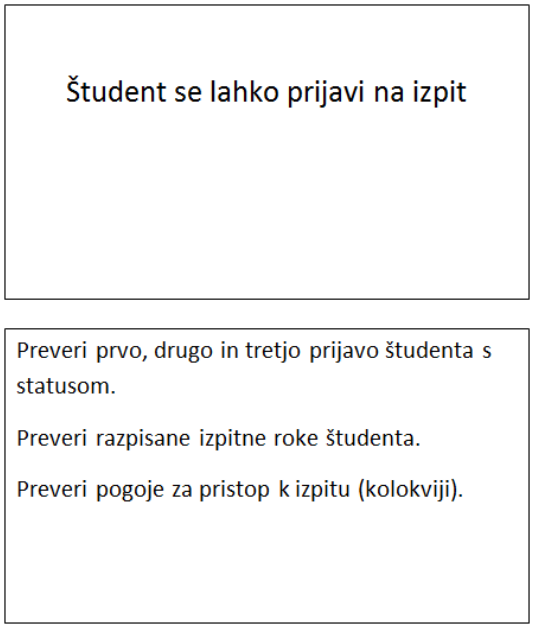
- zapišemo, kaj sistem zazna in ne samo tega, kaj se je zgodilo;
- pogoj razširitve napišemo pred dvopičje v svoji vrstici, korake razširitve pa zamaknemo en nivo nižje in vsakega zapišemo v svojo vrstico.

2.1.3 Metoda uporabniških zgodb

Metodo uporabniških zgodb smo opisali po Cohnovem [2] zgledu.

V tem poglavju bodo na splošno opisane **uporabniške zgodbe**. Uporabniška zgodba opisuje funkcijo sistema oziroma programske opreme, ki bo koristna uporabniku ali naročniku. Sestavljajo jo trije deli: kartica, pogovori in testi. Na kartici je opisana uporabniška zgodba, ki služi načrtovanju sistema oziroma programske opreme in je neke vrste opomnik, ni pa to dokumentacija. Med pogovori podrobneje opišemo zgodbe, testi pa služijo kot kriteriji za dokončanost zgodbe in hkrati dokumentirajo podrobnosti. Na sliki 2.1 vidimo primer kartice: “Študent se lahko prijavi na izpit.”

Pomembno je, da uporabniške zgodbe uporabljajo jezik, ki ga razumejo uporabniki oziroma naročniki, saj je le na tak način mogoča komunikacija z njimi. Tako npr. naročnika, ki je naročil izdelavo spletnega portala za iskanje zaposlitve, ne zanima, če bo aplikacija napisana v programskem jeziku python.



Slika 2.1: Primer uporabniške zgodbe

Prav tako je za zgodbe bolje, da so krajše kot pa dolge. Velike zgodbe imenujemo **preobsežne zgodbe** (angl. epic). Take zgodbe moramo razdeliti na manjše dele.

Cohn [2] predlaga, da pri razvoju sodelujemo z naročniško skupino, ki jo sestavljajo produktni vodja (angl. product owner), končni uporabniki, oblikovalci interakcije in testerji. Uporabniki so prisotni pri celotnem razvoju zgodbe. Uporabnike razdelimo na več uporabniških vlog, npr. študent, profesor, asistent...

Ko so uporabniške zgodbe določene, ocenimo njihovo zahtevnost, razvijalci pa povejo, kakšna bo njihova hitrost, to je število točk, ki jih lahko realizirajo v eni iteraciji. Glede na oceno trajanja in prioriteto zgodb se potem odločimo, katere zgodbe bodo prišle v prvi izid.

Zgodbam določimo prioriteto glede na to, ali bomo najprej ugodili največji skupini končnih uporabnikov ali manjši skupini pomembnejših uporabnikov ali pa bomo dali prednost kohezivnosti zgodb v relaciji z drugimi zgodbami.

Pri razvrščanju zgodb moramo paziti, da ne presežemo predvidene hitrosti.

Sprejemne teste zgodb uporabniki začnejo pisati takoj na začetku. Zaželjeno je, da programerji teste avtomatizirajo, saj jih tako lahko v primeru neuspeha hitro ponovijo.

Pri uporabniških zgodbah je poudarek na verbalni in ne pisni komunikaciji.

Lastnosti dobrih zgodb

Ni vsaka zgodba dobra, dobre so le tiste, ki imajo določene lastnosti. Dobra zgodba je torej neodvisna, ocenljiva, koristna uporabnikom in naročnikom, majhna in taka, da jo je mogoče testirati ter se o njej pogajati.

Neodvisnost pomeni, da zgodba ni odvisna od drugih zgodb. Tako odvisnost lahko odpravimo na dva načina. Prvi način je, da manjše zgodbe združimo v večjo, a ne preveliko zgodbo, drugi pa, da zgodbo razdelimo drugače. Imamo zgodbe: "Podjetje lahko plača s kartico Visa", "Podjetje lahko plača s kartico MasterCard", "Podjetje lahko plača s kartico Diners". Tu pa naletimo na problem pri določanju zahtevnosti vsake zgodbe. Zgodba, ki se je bomo lotili najprej, bo imela najdaljši čas realizacije, ostali dve pa krajšega. Ta problem lahko rešimo z drugačno razdelitvijo večjih zgodb na manjše. Zgodbe bi lahko združili v eno večjo, a če bi bila ta predolga, bi lahko uporabili sledečo razdelitev: "Podjetje lahko plača z enim tipom kartice" in "Podjetje lahko plača še z dvema tipoma kartice". Če zgodb nočemo združevati ali pa drugačnega načina razdeljevanja ne najdemo, lahko zgodbam določimo dve oceni. Eno, če se zgodbe lotimo prej in drugo, če se je lotimo kasneje, pri tem pa je pomembno, da se druge zgodbe lotimo šele, ko je končana prva.

Zgodbe naj bi bile **ocenljive**. To pomeni, da programerji lahko ocenijo, koliko časa bodo porabili za realizacijo zgodbe. Pri tem se lahko pojavijo težave, če jim primanjkuje domenskega ali tehničnega znanja. V takem primeru se zgodba razdeli na dva dela. Prvi del obsega učenje nepoznane snovi,

drugi del pa je realizacija zgodbe. Zgodba ni ocenljiva, če je prevelika, lahko pa preobsežne zgodbe služijo tudi kot opomnik, kaj je potrebno v prihodnosti še realizirati, kar pomeni, da zgodbe ne razdelimo takoj na začetku.

Primer zgodbe, ki je **koristna uporabnikom ali naročnikom** je: "Vse nastavitve programske opreme se preberejo iz centralne lokacije". Uporabnikom je vseeno, od kje se preberejo nastavitve, naročnike pa bi to verjetno zanimalo.

Zgodbe naj bodo **majhne**. Poznamo dve vrsti velikih zgodb, sestavljene in kompleksne, in dva načina, kako jih lahko razdelimo. Sestavljena zgodba je zgrajena iz več manjših. Zgodba se lahko glasi: "Uporabnik spletne strani lahko objavi svoj življenjepis". Ta zgodba pa dejansko pomeni: "Uporabniki lahko označijo življenjepis kot neaktiven", "Uporabniki imajo lahko več življenjepisov", "Uporabniki lahko urejajo svoje življenjepise" itd. Kompleksne zgodbe pa so zgodbe, ki so velike in se jih ne da enostavno razdeliti v množico sestavljenih zgodb. Zgodbo imamo za kompleksno, če programerji nimajo dovolj znanja oziroma izkušenj, da bi jo realizirali. Primer take zgodbe je lahko: "Podjetje lahko plača za storitev s kreditno kartico". Če se nihče od programerjev ni še nikoli ukvarjal s plačevanjem s kreditnimi karticami preko spleta, zgodbo razdelimo na dva dela, in sicer "Razišči uporabo kreditnih kartic preko spleta" in "Podjetje lahko plača storitev s kreditno kartico". Včasih pa so zgodbe premajhne. To se ponavadi pojavi pri odpravljanju hroščev, ko razvijalci zgodbe niti nočejo zapisati, saj bi jim to vzelo več časa kot pa sama odprava teh hroščev. V takem primeru se nekaj manjših zgodb združi v eno, ki razvijalcem predstavlja od pol do nekaj dni dela.

Možnost testiranja pomeni, da je zgodbe možno testirati. Primer zgodbe, ki je ne moremo testirati, je: "Uporabnik nikoli ne čaka dolgo na prikaz okna." Problem je v besedi *dolgo*, ki nima konkretnega pomena. Bolj pravilno bi bilo, če bi napisali: Uporabnik v 95% primerov ne sme čakati več kot 5 sekund za prikaz okna. Tako zgodbo pa lahko preverimo in kar je še bolje, test se da avtomatizirati, kar pomeni, da ob rasti nastajajočega sistema tega pogoja ni težko večkrat preveriti.

Da se je o zgodbi **možno pogajati**, pomeni, da na kartici nimamo točno določenih podrobnosti, lahko pa v opombo napišemo kakšne že vnaprej znane podrobnosti. Opomba ne sme biti predolga, ker zavira komunikacijo in daje občutek, da so vse podrobnosti že zajete. Prostor za podrobnosti je zato v sprejemnih testih.

Uporabniške vloge

V tem poglavju bomo raziskovali različne **uporabniške vloge** programske opreme, ki jo izdelujemo. Uporabniška vloga je pri razvijanju strani za objavljanje in iskanje oglasov s službami lahko iskalec zaposlitve, odpuščeni delavec, iskalec zaposlitve na specifičnem območju, delodajalec, nekdo, ki oglase spremlja le občasno itd. Vedno pa obstajajo tudi prekrivanja med uporabniki. Vse uporabniške vloge, to so na primer iskalec zaposlitve, odpuščeni delavec in iskalec zaposlitve na specifičnem območju, bodo uporabljali iskalnik zaposlitev na spletni strani.

Uporabniške vloge določimo v štirih korakih:

- določitev začetne množice uporabniških vlog,
- organizacija začetne množice,
- združevanje vlog,
- izpopolnjevanje vlog.

Določitev začetne množice uporabniških vlog poteka v obliki zbiranja predlogov (angl. brainstorming). Uporabniki in razvijalci se usedejo za isto mizo ali se postavijo pred tablo. Vsak dobi kup praznih kartic in ko se spomni določene uporabniške vloge, jo napiše na kartico, prebere naglas in postavi na mizo oziroma pripne na tablo. To vsi člani skupine ponavljajo toliko časa, dokler jim ne zmanjka idej.

Zatem je čas za organiziranje teh popisanih kartic. Uporabniki in razvijalci kartice položijo na mizo in jih prerazporedijo tako, da njihova lega

Slika 2.2: Primer organiziranja uporabniških vlog

odraža razmerje med njimi. Če se vloge popolnoma prekrivajo, se popolnoma prekrivajo tudi kartice, če pa se vloge prekrivajo deloma, se delno prekrivajo tudi kartice. Primer takega organiziranja vidimo na sliki 2.2

Združevanje uporabniških vlog je proces zgoščevanja uporabniških vlog. Avtorji prekrivajočih se kartic povedo, kaj so si zamislili pod imenom vloge in po kratki razpravi se skupina odloči, ali so vloge ekvivalentne. Če so, potem lahko razvijalci vloge združijo v eno. Če pa se odločijo, da določena vloga predstavlja zelo majhen delež uporabnikov ali pa je nepomembna, lahko nekatere izmed vlog tudi zavržejo. Ko se razvijalci odločijo, katere vloge je vredno razlikovati in katerih ne, preostale kartice prerazporedijo tako, da spet odražajo medsebojna razmerja. Iskalec zaposlitve je generična vloga, ki jo postavimo nad odpuščenega delavca in iskalca prve zaposlitve zato, ker sta to specializirani vlogi iskalca zaposlitve.

Pri fazi izpopolnjevanja vlog v prejšnjem koraku izbranim vlogam določimo atribute. Atributi vloge so dejstva ali delci koristnih informacij o uporabnikih, ki spadajo v to vlogo. Kakršnakoli informacija, ki eno vlogo razlikuje od druge, se lahko zapiše sem.

Primeri atributov, za katere se lahko odločimo:

- pogostost uporabe programske opreme uporabnika,
- uporabnikovo poznavanje problemske domene,
- uporabnikova splošna izurjenost pri rokovanju z računalniki.

Na splošno moramo pri določanju atributov upoštevati programsko opremo, ki jo razvijamo, in pomembne lastnosti uporabnikov v povezavi z njo. Ugotovljene atribute zapišemo na kartico, kamor smo zapisali vlogo uporabnika.

Ponavadi se tukaj določanje uporabniških vlog konča. Obstajata še dve tehniki, in sicer tehnika, ki uporablja persone, in tehnika, ki uporablja ekstremne karakterje, ki pa se ju poslužujemo samo v primeru, ko pričakujemo njun znaten prispevek k projektu.

Zajemanje zgodb

To poglavje bomo posvetili opisu različnih **tehnik za pridobivanje uporabniških zgodb**. Pri agilnih metodah je sprejeto dejstvo, da vseh zgodb ne moremo zajeti v prvem prehodu. Pri teh metodah se pomembnost zgodbe spreminja glede na čas in nabor zgodb, ki so bile produktu dodane v prejšnjih iteracijah. Čeprav se zavedamo, da je vse zgodbe projekta nemogoče napisati vnaprej, še vedno poizkusimo napisati vse tiste, ki jih lahko, pa čeprav le na bolj abstraktnem nivoju. Napišemo lahko zgodbo "Uporabnik lahko išče službe," ki nam služi kot opomnik ali pa, ker je v tistem trenutku to vse, kar vemo. Kasneje lahko to zgodbo razdrobimo na manjše zgodbe. Uporabniških zgodb naj se na začetku ne bi pisalo dolgo časa. Bolje je, da uporabimo že omenjene opomnike in jih natančneje zapišemo šele potem, ko bolje spoznamo projekt ali ko uporabnik bolj točno ve, kaj hoče. Opomniki nam služijo za predstavo o velikosti aplikacije, saj tako projektu že na začetku lažje določimo ceno.

Ker se zgodbe razvijajo s časom, potrebujemo tehnike za pridobivanje zgodb, ki jih uporabljamo iterativno. Tehnike ne smejo biti preveč ovirajoče,

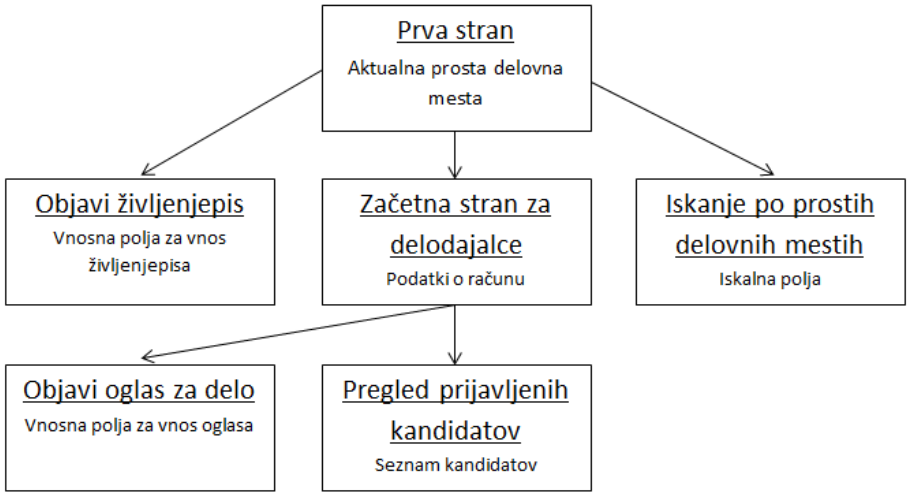
zato da jih lahko uporabljamo bolj ali manj ves čas. Nekaj najbolj cenjenih tehnik za pridobivanje zgodb:

- intervjuji z uporabniki,
- ankete,
- opazovanja,
- delavnice pisanja zgodb.

Intervjuji z uporabniki so najpogostejša metoda za pridobivanje zgodb. Najbolje je, da intervjuje opravimo z dejanskimi uporabniki, ne z njihovimi zastopniki (angl. proxies) (gl. pogl. *Zastopniki uporabnikov*) ter da intervjuvamo različne tipe uporabnikov.

Vprašanja v intervjuju naj bodo odprtega tipa, saj tako dobimo najbolj uporabne odgovore. Primer vprašanja zaprtega tipa je: "Ali želite, da bo naša nova aplikacija tekla v internetnem brskalniku?" Na to vprašanje bo skoraj vsak odgovoril z "Da". Le redkokateri uporabnik ve, kaj to dejansko pomeni in čemu se mora odpovedati pri implementaciji aplikacije, ki teče v internetnem brskalniku, v primerjavi z implementacijo aplikacije, ki teče kot program na operacijskem sistemu. Veliko bolje je, če vprašamo: "Čemu bi se bili pripravljeni odpovedati, da bi aplikacija tekla v internetnem brskalniku?" S takim vprašanjem uporabnika prisilimo k razmišljanju o tem, kaj od aplikacije želi. Seveda je treba sčasoma preiti tudi na bolj specifična vprašanja. Vprašanja odprtega tipa so pomembna, ker nam omogočajo odkrivanje zgodb, ki jih z uporabo izključno specifičnih vprašanj najbrž ne bi odkrili.

Ankete so učinkovit način za zbiranje informacij o zgodbah, ki jih že imamo in jim zato lažje določimo prioriteto. Ankete pa niso primerne za začetno zbiranje zgodb, ker podvprašanj ne moremo postavljati naknadno, tako kot to lahko počnemo v intervjuju, prav tako pa ne moremo slediti poti, če skupaj z uporabnikom odkrijemo kakšno zanimivo zgodbo. V anketi lahko uporabnike sprašujemo, katere funkcije v obstoječih programih uporabljajo



Slika 2.3: Primer organiziranja uporabniških vlog

najpogosteje in o razlogih, zakaj druge funkcije uporabljajo manj pogosto. S temi ugotovitvami lahko popravimo oziroma določimo prioriteto zgodb.

Z opazovanjem uporabnikov lahko dobimo veliko novih idej o tem, kako bi lahko izboljšali njihovo izkušnjo ob uporabi in/ali produktivnost narejene aplikacije. Opazovanje uporabnikov sicer ni vedno mogoče, je pa priporočljivo, da se te tehnike poslužujemo, kadarkoli se le da.

Delavnica pisanja zgodb je sestanek, ki se ga udeležijo razvijalci, uporabniki in vsi ostali, ki lahko prispevajo k pisanju zgodb. Tukaj prioritet še ne dodeljujemo, saj bo naročnik imel priložnost to storiti kasneje. Delavnice lahko skličemo tudi kasneje, a ponavadi to niti ni potrebno. V pravilno vodeni delavnici lahko napišemo veliko zgodb v kratkem času.

V delavnici pisanja zgodb najprej zarišemo najbolj osnovne interakcije med elementi načrtovane aplikacije, nato pa iterativno poglabljamo podrobnosti. Tukaj prepoznavamo predvsem konceptualne poteke dela in ne dejanskega uporabniškega vmesnika.

Na sliki 2.3 vsak okvir predstavlja novo komponento spletne strani. Naslov komponente je podčrtan. Pod naslovom je kratek seznam vsega, kar komponenta dela ali vsebuje. Puščice med okvirji predstavljajo povezave

med komponentami. Na primeru spletne strani je povezava lahko nova stran ali prostor na trenutni strani. Za katero od teh dveh gre, pa trenutno ni pomembno, saj se razvijalci in uporabniki o tem dogovorijo kasneje.

Pri prototipu se najprej odločimo, s katero uporabniško vlogo bomo začeli. Proces ponovimo z vsako izmed vlog, zato vrstni red ni pomemben. Potem narišemo prazen okvir in udeležencem povemo, da je to glavni zaslon programa, ter jih vprašamo, kaj lahko obravnavana uporabniška vloga naredi od tu naprej, in to ponavljamo, dokler ne uporabimo vseh akcij uporabniške vloge. Iz slike lahko naredimo sledeče zgodbe:

- iskalec zaposlitve lahko objavi svoj življenjepis,
- delodajalec lahko objavi oglas za delo,
- delodajalec lahko pregleda prijavljene kandidate,
- iskalec zaposlitve lahko išče po prostih delovnih mestih itd.

Nobena od naštetih zgodb ne zahteva, da vemo, kako bo videti zaslon, ko bo oblikovan, vendar pa bi to vseeno olajšalo pisanje zgodb vseh udeleženih.

Narejeni prototip moramo čez nekaj dni zavreči, saj se stvari spreminjajo, zastareli prototip pa lahko ustvari zmedo, če ga imamo na vidnem mestu.

Ko se sprehajamo skozi prototip, si postavljamo vprašanja, ki nam pomagajo najti manjkajoče zgodbe, kot so:

- Kaj bi uporabnik najverjetneje želel storiti v prihodnje?
- Kakšne napake lahko uporabnik stori?
- Kaj lahko zmede uporabnika?
- Kakšne dodatne informacije bi lahko uporabnik potreboval?

Med potekom delavnice se raje kot na kvaliteto osredotočimo na kvantiteto. Četudi imamo zgodbe v elektronski obliki, jih raje zapisujemo na papir. Če kasneje kakšno zgodbo zamenjamo z boljšo, lahko staro preprosto

zavržemo. Če se nam med procesom kje zatakne, se lahko zgledujemo po konkurenčnih produktih.

Med delavnico naj prevladuje pogovor, saj je naš cilj napisati čim več zgodb v čim krajšem času. To ni čas za oblikovanje zaslonov ali reševanje problemov.

Zastopniki uporabnikov

Včasih do dejanskih uporabnikov ni mogoče dostopati, bodisi zato, ker so preveč oddaljeni ali pa zato, ker so časovno nedostopni. V takem primeru uporabimo zastopnike uporabnikov, ki niso končni uporabniki, ampak ljudje, ki končne uporabnike le predstavljajo. V tem poglavju bomo govorili o **različnih tipih zastopnikov uporabnikov**, to so: vodja uporabnikov, vodja razvoja, prodajalec, domenski eksperti, bivši uporabniki, naročniki, inštruktorji in tehnična podpora ter poslovni in sistemski analitiki.

Eden od tipov zastopnikov uporabnikov je vodja uporabnikov. To bo lahko dejanski uporabnik aplikacije, ki jo razvijamo, ali pa ne. Tudi če bo ta uporabnik aplikacijo dejansko uporabljal, ima lahko različne vzorce uporabe od dejanskih uporabnikov. Vodja uporabnikov ponavadi uporablja druge funkcije aplikacije kot ostali uporabniki in zna se zgoditi, da pri razvoju konkretne aplikacije da večji poudarek manj kot pa bolj uporabljanim funkcijam.

Vodja razvoja je eden najslabših zastopnikov uporabnikov, ki jih lahko uporabljamo, razen če razvijamo aplikacijo za vodenje razvoja. Vodja razvoja na primer lahko daje poudarek drugačnim zgodbam kot dejanski uporabniki, ker hoče predstaviti novo tehnologijo. Lahko ga zavede tudi želja po letni nagradi, ki je vezana na dokončanje projekta, in zato aplikacijo sprejme za končano prej, kot pa bi to storil končni uporabnik.

Nevarnost pri uporabi tipa prodajalca je ta, da ponavadi ta daje prednost tistim funkcijam aplikacije, katerih odsotnost mu je nazadnje preprečila, da bi aplikacijo prodal. Če je prodajalec izgubil stranko, ker aplikacija na primer nima možnosti razveljavitve, bo tej funkciji seveda dal največjo prioriteto. Pri tem se rado zgodi, da zaradi implementacije teh stvari izgubimo pogled

na dolgoročni cilj oziroma na to, kaj naj bi aplikacija sploh počela.

Domenski eksperti so pomembni zato, ker razumejo domeno, ki se bo uporabljala pri aplikaciji. So nepogrešljiv del pri razvijanju aplikacije, ki se ukvarja s področjem, ki ga razvijalci zelo malo oziroma sploh ne poznajo. Domenski eksperti niso nujno dejanski uporabniki, potrebujemo pa jih vsaj pri postavitvi domenskega modela.

Uporabimo lahko tudi bivše uporabnike, vendar moramo pri tem paziti, da so njihovi cilji enaki ciljem končnih uporabnikov.

Naročniki so tisti, ki se odločijo za nakup aplikacije, niso pa to nujno tudi njeni uporabniki. Primer razlike med naročniki in uporabniki je na primer oddelek informatikov v podjetju, ki se odloči za nakup urejevalnika besedila. Čeprav ga ne bodo uporabljali informatiki, jih vseeno zanimajo nekatere funkcije programa, na primer varnost. Nevarnost, ki se pojavi pri uporabi tega zastopnika uporabnika je, da ne more z gotovostjo povedati določenih stvari, recimo v kateri datotečni format bi uporabniki radi izvažali podatke, saj ne vedo, kateri program za obdelavo podatkov jim je najbolj domač.

Inštruktorji in tehnična podpora dajejo lažen vtis, da so dobri zastopniki uporabnikov, ker imajo veliko stikov z njimi, a če jih uporabimo kot zastopnike uporabnikov, bomo na koncu dobili aplikacijo, ki se je je lahko naučiti oziroma za katero je lahko dajati tehnično podporo.

Poslovni in sistemski analitiki so dobri zastopniki uporabnikov, ker imajo znanje iz tehnološkega sveta in domene aplikacije. Se pa lahko pri njih pojavi problem, da preveč razmišljajo o tem, kaj bi uporabniki hoteli, namesto da bi jih o tem povprašali. Zna se zgoditi tudi, da se bodo hoteli predolgo sestajati v delavnicah za pisanje zgodb.

Kadar imamo dostop do zastopnikov uporabnikov, poznamo dve tehniki, s katerima lahko napišemo dobro aplikacijo.

Kadar imamo omejen dostop do končnih uporabnikov in imamo dodeljenega zastopnika uporabnika, ki bo odločal o vsem, moramo vseeno vzpostaviti stik s končnimi uporabniki. V tem primeru projektno skupino sestavimo iz dejanskih uporabnikov. Skupina zastopniku svetuje in ga vodi, ideje pa si

izmenjujejo na sestankih.

Kadar nimamo na voljo nobenega pravega uporabnika, ampak samo zastopnike, je dobro, da poizkusimo pridobiti še več zastopnikov. S to tehniko zmanjšamo možnost, da bi naredili aplikacijo, ki bi zadostovala potrebam le ene določene osebe. Namesto dveh domenskih ekspertov lahko vzamemo na primer enega in nekoga iz trženja. Pri odločitvah sta lahko enakovredna ali pa ne, kljub temu pa se mora glavni ozirati tudi na mnenje podrejenega. Pri izdelavi aplikacije je dober vir dodatnih zgodb tudi konkurenčna aplikacija, za katero si lahko pogledamo ocene kritikov ter njene dobre in slabe lastnosti. Alternativen pristop do problema, ko končnih uporabnikov nimamo, pa je, da prvo izdajo predamo v uporabo kar se da hitro, saj s tem lahko hitreje dobimo mnenja uporabnikov in jih primerjamo z mnenji zastopnikov uporabnikov.

Pri sestavljanju uporabniške ekipe se moramo zavedati, da so končni uporabniki vredni več kot zastopniki. Zaradi tega postavimo v ekipo čim več končnih uporabnikov, zastopnike pa dodamo šele takrat, ko končnih uporabnikov nimamo dovolj.

Pri sestavljanju uporabniške ekipe poznamo štiri korake.

Najprej dodamo prave uporabnike. Če aplikacijo uporablja več tipov uporabnikov, poskušamo vanjo vključiti vse tipe le-teh.

Potem določimo "prvega med enakimi". V komercialnih aplikacijah je to ponavadi vodja projekta, lahko pa je to tudi kakšen drug član ekipe. Ta oseba je odgovorna za sodelovanje uporabniške ekipe, pri predstavitvi ugotovitev pa morajo biti člani ekipe enotni.

Tretji korak je določanje dejavnikov, ki so ključni za uspeh. Če delamo naslednjo generacijo že obstoječe aplikacije, je eden od ključnih faktorjev za uspeh enostavnost prehoda na nov sistem. Uporabniško ekipo dopolnimo z zastopniki z znanjem, izkušnjami in spretnostmi, ki bodo odločali o kritičnih faktorjih aplikacije.

Sprejemni testi

V tem poglavju se bomo posvetili sprejemnim testom, ki nam v prvi vrsti povedo, kdaj je zgodba zaključena. Sprejemne teste se med drugim piše tudi zato, ker se na tak način izrazi mnogo podrobnosti, ki se pojavijo v pogovorih med uporabniki in razvijalci. Teste pišemo v dveh korakih. Prvi korak je, da vsak test, ki se ga kdo spomni, najprej zapišemo na hrbtno stran kartice, v drugem koraku pa se ti testi razvijejo v prave teste, ki dokažejo, ali je zgodba pravilno in do konca izpeljana. Za zgodbo "Podjetje lahko plača za objavo s kreditno kartico" lahko na hrbtno stran kartice zapišemo take teste:

- Preveri s karticami Visa, MasterCard in American Express (sprejme);
- Preveri s kartico Diners Club (zavrne);
- Preveri s pravilno, nepravilno in pomanjkljivo številko kartice;
- Preveri s pretečeno kartico;
- Preveri nakup z različnimi znesko (vključno z zneskom, ki presega limit).

Nekatere stvari lahko programer predvidi tudi sam; v primeru, da uporabnik ne izpolni polja "Telefonska številka", se v pregledu postavka "Telefonska številka" sploh ne pojavi.

Ker se teste piše vnaprej, jih lahko programerji uporabljajo kot opomnike in tako implementirajo tudi stvari, na katere bi drugače pozabili. Teste ponavadi napišemo:

- kadar se uporabniki in razvijalci pogovarjajo o zgodbi in želijo zajeti specifične podrobnosti;
- že na začetku iteracije še preden se začne programiranje;
- kadar se novi testi odkrijejo med ali po programiranju zgodbe.

Uporabnik naj zgodbe pregleda na začetku iteracije in dopiše še dodatne teste, ki se jih spomni. Pri tem so mu lahko v pomoč vprašanja: Kaj morajo programerji še vedeti o tej zgodbi? Kaj bo v to zgodbo implementirano? Ali obstajajo okoliščine, ki bodo povzročile, da se bo aplikacija obnašala drugače? Kaj gre pri tej zgodbi lahko narobe?

Aplikacija je zamisel uporabnika in on lahko pove, kdaj je zgodba končana, zato naj on določi tudi teste, programer pa naj mu pomaga, da bodo ti testi točni. Programerji ali testerji ponavadi dodajo še kakšen test, ki se ga spomnijo med programiranjem.

Testi za trivialne napake, ko je na primer 30. februar zaznan kot napačen vnos, naj bodo narejeni kot že vnaprej avtomatizirani testi, saj uporabnik ni odgovoren za odkrivanje vseh mogočih napak.

Poleg testov funkcionalnosti poznamo še testiranje uporabniškega vmesnika, testiranje enostavnosti uporabe, testiranje hitrosti delovanja pod različnimi obremenitvami in stresni test, kjer aplikacijo testiramo pod ekstremnim številom uporabnikov, transakcij in česarkoli drugega, kar aplikacijo obremenjuje.

Nasveti za pisanje uporabniških zgodb

V tem poglavju bomo zapisali nekaj splošnih nasvetov za celoten postopek pisanja uporabniških zgodb. Na velikih projektih, posebej takih, ki imajo veliko uporabniških vlog, se je najbolje osredotočiti na vsako vlogo posebej in razbrati, kaj bi določen uporabnik z našo aplikacijo rad dosegel. To pomeni, da v bistvu napišemo velike zgodbe in jih razdelimo na manjše dele, ki so lahko zgodba sama zase ali pa podlaga za izpeljavo novih zgodb.

Deljenje velikih zgodb naj poteka na tak način, da bodo tudi novonastale manjše zgodbe uporabne. Zgodbo “Iskalec zaposlitve lahko objavi življenjepis” lahko na primer razdelimo na ti dve zgodbi “Iskalec zaposlitve lahko izpolni obrazec za življenjepis” in “Informacija iz obrazca se zapiše v podatkovno bazo”. Taka razdelitev je slaba. Če v iteracijo pride samo prva zgodba, dejansko ne naredi ničesar, saj se podatki nikamor ne shranijo.

Boljša je razdelitev, kjer sta zgodbi, ko sta realizirani, že uporabni. Taki zgodbi sta na primer "Iskalec zaposlitve lahko objavi življenjepis z osnovnimi informacijami" in "Iskalec zaposlitve lahko objavi življenjepis, ki vsebuje vse ostale informacije, ki bi jih delodajalec želel videti".

Zgodbe naj bodo take, da bo uporabnik imel občutek, da je nekaj dosegel. Takim zgodbam pravimo zaključene (angl. closed) zgodbe. To pomeni, da ne pišemo zgodb v smislu "Uporabnik lahko ureja sliko", ampak zgodbe, kot je na primer "Uporabnik lahko izreže del slike". Pri tem moramo paziti, da ne napišemo prevelikih ali premajhnih zgodb.

Uporabne so tudi kartice z omejitvami. Te kartice ne spadajo v klasičen razvojni cikel iteracije, ampak stojijo na vidnem mestu, saj nam služijo kot opomniki, kakšna naj bo aplikacija kot celota. Primera takih kartic sta kartici "Aplikacija mora teči na vseh verzijah Windows" in "Sistem mora 99,99% časa delovati neprekinjeno." Omejitvena kartica naj ima nekje na robu napisano opombo "omejitev".

Dobro je, da zgodbe z uporabniškim vmesnikom implementiramo čim kasneje, saj razvijalci velikorat radi zamešajo zahteve in specifikacije rešitve.

Uporabniške zgodbe so sicer zelo prilagodljive, a vseeno niso primerne za opis čisto vsega. Dober primer tega je uporabniški vmesnik, saj za njegov opis namesto uporabniških zgodb raje uporabimo dokument, ki vsebuje veliko slik.

Pri pisanju uporabniških zgodb v opisu uporabimo kar uporabniško vlogo in ne generičnega uporabnika. Zgodbo "Uporabnik lahko objavi svoj življenjepis", bi lahko z uporabo uporabniške vloge veliko bolje zapisali kot "Iskalec zaposlitve lahko objavi svoj življenjepis".

Pri razumevanju zgodb je pomembno vedeti, da si lažje predstavljamo zgodbe, ki so pisane v ednini in tvorniku, če je to le mogoče. Zgodba "Iskalci zaposlitve lahko odstranijo svoj življenjepis s spletne strani" je zavajajoča, saj bi jo lahko kdo interpretiral kot da lahko iskalec zaposlitve odstrani ne le svoj življenjepis, ampak tudi kakšnega, ki ga ni ustvaril on.

Zaželjeno je, da uporabniki sami pišejo zgodbe, razvijalci pa naj jim po-

magajo pri dejanskem pisanju ali s predlogi. Uporabniki so odgovorni za določanje prioritete zgodb, ki bodo šle v iteracijo, zato je pomembno, da razumejo vse zgodbe, kar najlažje dosežemo tako, da jih napišejo sami.

Uporabniške zgodbe nas le opominjajo na diskusijo o funkciji aplikacije, ki smo jo imeli med pisanjem določene zgodbe, zato naj bodo kratke. Podrobnosti, ki jih potrebujemo, dodajmo na kartico, nikar pa komunikacije ne zamenjajmo z dodajanjem preveč podrobnosti na kartico.

Ocenjevanje trajanja uporabniških zgodb

V tem poglavju bomo raziskali metodo za ocenjevanje trajanja posamezne zgodbe. Najboljši način ocenjevanja trajanja uporabniške zgodbe mora:

- dovoljevati spremembo ocene vsakič, ko o zgodbi dobimo nove informacije,
- delovati pri velikih in manjših zgodbah,
- nuditi uporabne informacije o opravljenem in preostalem delu,
- tolerirati netočnosti v ocenjevanju,
- omogočati uporabo pri načrtovanju izdaj.

Pristop, ki zadosti vsem tem zahtevam, je ocenjevanje zgodb s številom točk (angl. story points), ki jih vsaka skupina lahko definira po želji. Ena točka tako lahko ustreza idealnemu delovnemu dnevu (tj. dnevu brez ostalih motenj), idealnemu delovnemu tednu ali pa je merilo kompleksnosti zgodbe.

Določanje točk poteka tako, da zberemo uporabnike in naročnike ter vsakemu izmed njih damo kupček praznih kartic. Iz kupčka zgodb naključno izberemo eno in jo vsem navzočim preberemo naglas, razvijalci potem sprašujejo o vsem, kar jih o zgodbi zanima, uporabnik pa jim skuša na vsa vprašanja odgovoriti. Če česa ne ve, ugiba ali pa prosi, da obravnavanje zgodbe prestavijo na kasnejši čas. Ko zmanjka vprašanj, vsak razvijalec zapiše svojo oceno trajanja na kartico, vendar je ne pokaže ostalim. Po

končanem ocenjevanju razvijalci pokažejo kartice drug drugemu in ker so ocene najverjetneje zelo različne, razvijalca, ki sta dala najvišjo in najnižjo oceno trajanja, svojo odločitev argumentirata. Vsak v skupini si zapomni vse komentarje, nato pa še naprej diskutirajo o trajanju zgodbe. Če med tem najdejo še kakšno izpuščeno zgodbo, jo napišejo naknadno. Po tej diskusiji razvijalci še enkrat napišejo vsak svojo oceno in če ocene zopet niso enake, ponovijo proces komentiranja diskutiranja in argumentiranja. Vse to ponavljajo toliko časa, dokler niso ocene približno enake. Najbolje je, da po vsaki določitvi točk zgodbi to zgodbo primerjajo s tistimi, ki imajo točke že določene, saj tako ne izgubijo občutka za to, koliko časa naj bi predstavljala ena točka.

Na koncu iteracije preštejejo, koliko točk so realizirali, in privzamejo, da bodo toliko točk realizirali tudi v naslednji iteraciji. S terminom "hitrost" označujemo število točk, ki jih skupina realizira v eni iteraciji. Za določitev hitrosti je dovolj, da vzamemo iteracijo, v kateri se ni zgodilo nič nenavadnega, na primer delanje nadur ali zaposlitev dodatnega programerja, za dobro določitev hitrosti iteracije pa moramo vzeti iteracijo, ki vsebuje neodvisne zgodbe. Za realno določitev hitrosti ne smemo vzeti iteracije, ki je vsebovala samo zgodbe s precenjenimi ali podcenjenimi zgodbovnimi točkami.

Med določanjem zgodbovnih točk se moramo zavedati, da točke, ki jih določi ena skupina, niso enakovredne točkam, ki jih določi druga in pa tudi, da kadar se večja zgodba razdrobi na manjše sestavne dele, ni nujno, da je vsota točk v teh sestavnih delih enaka številu točk večje zgodbe.

Načrtovanje izdaje

V tem poglavju bomo nekoliko bolj natančno razložili planiranje izdaje. V idealnem primeru se razvijalci in uporabniki namesto o točno določenem datumu dogovorijo za nek časovni okvir, ki naj bi bil potreben za izid. Če namesto točnega datuma določimo samo nek časovni okvir, ima skupina večjo fleksibilnost pri določanju za izid potrebnega časa. Primer je izjava: "Po šestih ali sedmih iteracijah naj bi bile minimalne zahteve izpolnjene, po de-

setih ali dvanajstih pa naj bi bilo narejeno vse, kar spada v prvi izid." Kadar imamo določen točen datum, je načrtovanje izida lažje, ker imamo manj spremenljivk, je pa težja odločitev o tem, katere zgodbe bomo vanj vključili.

Prioritete zgodb označujemo v skladu z metodologijo DSDM, ki za določanje prioritete predlaga tehniko, imenovano MoSCoW (gl. [2], str. 98) in je akronim za "Must have, Should have, Could have in Won't have this time", oziroma po slovensko "mora imeti, naj bi imelo, lahko bi imelo in zdaj ne bo imelo". "Must have" zahteve so tiste, ki so bistvene za sistem, "Should have" so funkcije, ki so pomembne, a jih lahko za kratek čas zaobidemo, "Could have" zahteve so tiste zahteve, ki jih lahko izpustimo, če nam zanje zmanjka časa, "Won't have this time" pa zahteve, ki so tam samo zato, da vemo, da obstajajo, in jih bomo realizirali kasneje.

Obstajata dva načina, po katerih lahko razvrščamo zgodbe:

- tveganje, da zgodba ne bo končana do željenega roka;
- vpliv, ki ga ima lahko zgodba na druge zgodbe, če jo odložimo.

Uporabnik ima lahko tudi svoj način, po katerem zgodbam določa prioriteto:

- realizacija funkcije aplikacije, ki je namenjena širši skupini uporabnikov;
- realizacija funkcije aplikacije, ki je namenjena manjšemu številu uporabnikov, ki so pomembni;
- kohezivnost zgodbe v relaciji z ostalimi zgodbami.

Ker imajo razvijalci svoje razloge za določanje prioritete, naročniki pa svoje, v primeru nesoglasja glede določanja prioritet vedno zmaga naročnik. Uporabniki od razvijalcev dobijo določene informacije, kot je na primer čas implementacije določene zgodbe, nato pa prioritete določajo glede na informacije, ki so jih dobili od razvijalcev in glede na svoje ocene uporabnosti zgodbe.

Uporabniki in razvijalci lahko skupaj določijo dolžino iteracije, ki jim najbolj ustreza. Ponavadi te iteracije trajajo od dva do štiri tedne. Krajše iteracije nam omogočajo sprotno prilagajanje in vpogled v potek projekta, vendar imamo s tem več dodatnega dela, povezanega z začetkom iteracije. Iteracije naj imajo enako dolžino, saj s tem pademo v nek konstanten ritem, ki ustreza celi skupini.

Hitrost iteracije lahko določamo glede na zgodovinske vrednosti, glede na dejansko hitrost v začetni iteraciji ali pa jo kar ugibamo. Uporaba zgodovinskih vrednosti je najboljša, ampak uporabna samo, če poznamo ekipo, ki se ukvarja s približno enakim problemom kot mi in pri kateri se sestava skupine ne spreminja. Določanje hitrosti glede na že končano iteracijo je tudi dober način, seveda pa obstajajo primeri, ko tak način ni uporaben. Ko po pisanju zgodb določimo zgodbovne točke, nam naročnik pove, da bo aplikacija pokrila stroške razvoja, če bo realizirana dovolj hitro. V tem primeru ne moremo kar reči, da mu bomo sporočili po dveh tednih, ko bomo končali prvo iteracijo, saj bi bilo to za naročnika že prepozno. Drugačen način za določanje hitrosti od določanja zgodbovnih točk pa je ugibanje. Če ena zgodbovna točka pomeni en idealen dan, lahko začetno hitrost predvidimo tako, da predvidimo, koliko dni ta dejansko predstavlja. Ponavadi je hitrost zmanjšana na polovico ali celo tretjino razpoložljivih človek-dni. Če imamo na primer dva tedna dolgo iteracijo, kar znese deset delovnih dni in šest programerjev, nam to predstavlja hitrost 60 realiziranih zgodbovnih točk na iteracijo. Če pa to prenesemo na dejanske dni, znese hitrost nekje med 20 in 30 realiziranimi zgodbovnimi točkami na iteracijo.

Načrtovanje iteracije

Zadnje poglavje pri tej metodi pa bo posvečeno malo daljši razlagi načrtovanja iteracije. Po končani iteraciji se celotna skupina programerjev in uporabnikov zbere na sestanku, na katerem načrtujejo nadaljnje iteracije. Na sestanku diskutirajo o zgodbi, razčlenjujejo zgodbo na sestavne dele, posamezni razvijalci si izberejo naloge in zanje prevzamejo odgovornost. Po pogovoru o

zgodbah in potem, ko razdelijo vse naloge, vsak razvijalec posebej pregleda naloge, ki so mu pripadle, in se prepriča, da mu te ne bodo vzele preveč časa.

Potem sledijo pogovori o zgodbah, ki so še ostale. Lahko se zgodi, da si uporabnik premisli glede prioritete določene zgodbe in takrat je pravi čas, da to izrazi. Na začetku sestanka uporabnik prebere zgodbo z najvišjo prioriteto. Razvijalci potem postavljajo vprašanja tako dolgo, dokler ne znajo zgodbe razdeliti na sestavne dele z ustreznim trajanjem. Paziti morajo, da se ne spuščajo preveč v podrobnosti, ker bi se sestanek tako lahko preveč zavlekel, vsem njegovim udeležencem pa tudi ni treba slišati vseh podrobnosti.

Tudi če so zgodbe že dovolj majhne, jih lahko razstavimo na še manjše dele, ker bo na njih delalo več v različnih tehnologijah specializiranih razvijalcev. Razstavljanje zgodbe na sestavne dele je uporabno zato, ker tako izpostavimo dele, ki bi jih lahko drugače pozabili. Zgodbo "Uporabnik lahko išče hotel po različnih parametrih" lahko razdelimo na sledeče dele: sprogramiraj osnovni zaslon iskanja, sprogramiraj napredni zaslon iskanja, sprogramiraj zaslon rezultatov, spiši in optimiziraj poizvedbo za podatkovno bazo za osnovna iskanja, dopolni dokumentacijo navodil za uporabo... V to zgodbo vključimo tudi dokumentacijo, čeprav je zgodba ne omenja. Skupina že iz prejšnjih iteracij ve, da dokumentacija obstaja in da mora biti na koncu iteracije točna. Zgodbe razčlenjujemo po teh pravilih:

- Če del zgodbe zahteva posebno pozornost, jo napišemo kot nalogo.
- Če različne dele zgodbe lahko naredijo različni razvijalci, te dele napišemo kot naloge. Tako razčlenjevanje je priročno, saj tako lahko več razvijalcev dela na isti zgodbi, kar se lahko izkaže za posebej pomembno proti koncu iteracije, ko se nam izteka čas.
- Če nam koristi, da je del zgodbe realiziran, potem jo razdelimo na naloge. Razvijalec, ki se na primer ukvarja s povezovanjem s podatkovno bazo, lahko zgodbo z osnovnim zaslonom poveže z bazo takoj, ko je zgodba končana, in mu ni treba čakati na to, da bo končana tudi zgodba za napredno iskanje.

Prevzemanje odgovornosti pomeni, da se nekdo iz skupine javi, da bo realiziral določene naloge. Njegova odgovornost je, da se naloga konča, če pa mu na koncu zmanjkuje časa, mu lahko pomagajo tudi drugi razvijalci, saj se trajanja nalog ponavadi ne da oceniti čisto natančno.

Te korake ponavljamo, dokler ne zapolnimo določenega števila zgodbovnih točk v iteraciji, nato pa vsak razvijalec posebej oceni, koliko časa bo potreboval za svoje delo v idealnem času. Ko razvijalec oceni potreben čas, sešteje svoje ocene in presodi, ali bo lahko te naloge realiziral med iteracijo. Če ocena približno sovpada ali zavzema več časa, kot ga bo imel v iteraciji, potem lahko: obdrži vse naloge in upa, da jih bo pravočasno končal, vpraša koga drugega, če bi prevzel kakšno izmed nalog ali pa vpraša uporabnika, če lahko kakšno zgodbo izpusti.

2.2 Praktični del

V tem poglavju bomo prej opisane metode predstavili tudi na konkretnih primerih. Vse metode bodo opisale spletni sistem E-študent, kakršnega smo na Fakulteti za računalništvo in informatiko uporabljali med leti 2003 in 2013. Izbrana je manjša podmnožica specifikacij, saj za potrebe primerjave ne potrebujemo vseh, vsaka od metod pa opisuje iste specifikacije. Pri vseh metodah privzamemo, da je to, kar opisujejo, celoten sistem.

2.2.1 Standard IEEE 830

IEEE 830 je pisan po uradni predlogi, vendar so poglavja, ki bi ostala prazna, izpuščena. Dokument uporablja lastno oštevilčevanje poglavij, ki zato niso navedena v glavnem kazalu diplomske naloge.

V tem podpoglavju najdemo samo tisti del specifikacij, ki vsebuje podrobne funkcionalne zahteve. Preostali dokument najdemo v dodatku. Poudariti je potrebno tudi to, da specifikacija v dodatku ni napisana v celoti, saj to ni bil namen naše naloge. Že to, kar je napisano, nam da dovolj dobro predstavo o tem, kako naj bi specifikacija izgledala.

3.1.1 Sistem mora študentu omogočati prijavo na izpitni rok

3.1.1.1 Sistem mora izpisati letnike, predmete, ki jih ima študent v predmetniku in pripadajoče izpitne roke, ki jim še ni potekel rok za prijavo.

3.1.1.2 Sistem mora omogočati študentu prijavo na izpitni rok in označbo, ali je pisni izpit že opravil s kolokviji.

3.1.1.3 Sistem mora preveriti, ali je študent pisni izpit res opravil s kolokviji in ali mora izpitni rok plačati.

3.1.1.3.1 Sistem mora preveriti, ali študent plača izpitni rok.

3.1.1.3.1.1 Študent brez statusa plača vsa polaganja.

3.1.1.3.1.2 Študent s statusom plača četrto in nadaljnja polaganja.

3.1.1.3.2 Sistem mora pri sedmi prijavi na izpitni rok preklicati prijavo in opozoriti uporabnika o napaki.

3.1.1.3.3 Sistem mora preklicati prijavo, če je od prijave na izpitni rok iz istega predmeta minilo manj kot 14 dni in je študent na njem dobil negativno oceno.

3.1.1.3.4 Sistem mora preprečiti prijavo študenta na izpitni rok če je označil, da je opravil kolokvije, vendar jih ni.

3.1.1.4 Sistem mora pravilno določiti zaporedno številko polaganja.

3.1.1.4.1 Določi se zaporedna številka polaganja, če študent ponavlja letnik, se polaganja v času prvega vpisa ne upoštevajo.

3.1.1.5 Sistem mora v primeru uspešne preverbe zapisati podatke v bazo.

3.1.2 Sistem mora omogočati pregled izpitnih rokov

3.1.2.1 Sistem mora izpisati vse letnike, ki jih je študent do sedaj že obiskoval.

3.1.2.2 Sistem mora izpisati vse izpitne roke za izbrani letnik v tekočem študijskem letu.

3.1.3 Sistem mora omogočati razpis izpitnega roka

3.1.3.1 Sistem mora ponuditi na izbiro vse predmete, ki jih izvaja profesor.

3.1.3.2 Sistem mora po izboru predmeta vprašati za določitev prostora izpitnega roka, možne točke pisnega izpitnega roka, datum in uro roka.

3.1.3.3 Sistem mora preveriti vnesene podatke.

3.1.3.3.1 Sistem mora preveriti prisotnost in resničnost podatkov (prostor, datum, točke).

3.1.3.3.2 Sistem mora preveriti, ali je na isti datum že razpisan rok za izbran predmet in če je, zahtevati ponovni vnos datuma.

3.1.3.4 Sistem mora v primeru uspešne preverbe zapisati podatke v bazo.

3.1.4 Sistem mora omogočati vnos končne ocene

3.1.4.1 Sistem mora omogočati izbiro izpitnega roka.

3.1.4.2 Sistem mora po izbiri izpisati seznam tistih prijavljenih na izpitni rok, ki ocene ali oznake za vrnjeno prijavnico še nimajo.

3.1.4.3 Sistem mora preveriti vnesene podatke.

3.1.4.3.1 Vnesene ocene morajo biti cela števila med ena in deset.

3.1.4.3.2 Vnašajo se lahko še oznake za vrnjeno prijavnico ali kasnejši vnos ocene.

3.1.4.3.3 Na koncu vnosa mora imeti vsak študent vneseno oceno ali oznako.

3.1.4.4 Sistem mora v primeru uspešne preverbe zapisati podatke v bazo.

3.1.4.5 Sistem mora po uspešnem vnosu ocen vsem prijavljenim študentom poslati sporočilo o oceni.

3.1.4.5.1 V primeru nedostavljenega sporočila mora sistem o tem opozoriti uporabnika.

3.1.5 Sistem mora omogočati brisanje izpitnega roka

3.1.5.1 Sistem mora izpisati vse razpisane izpitne roke profesorja.

3.1.5.2 Po izbiri predmeta mora sistem, če prijave na izpitni rok obstajajo, uporabnika vprašati, če vseeno želi izbrisati izpitni rok.

3.1.5.2.1 Če se uporabnik s tem strinja, mora sistem izbrisati vse prijave na izpitni rok in o tem obvestiti študente.

3.1.5.2.2 Če se uporabnik s tem ne strinja, se brisanje prekliče.

3.1.5.3 Sistem mora v primeru uspešne preverbe zapisati podatke v bazo.

3.1.6 Sistem mora omogočati kreiranje predmeta

3.1.6.1 Sistem mora na strani za kreiranje predmeta omogočati vnos imen profesorjev, ki izvajajo predmet, imena predmeta in števila kreditnih točk.

3.1.6.2 Sistem mora po potrditvi vnosa s strani uporabnika preveriti podatke.

3.1.6.2.1 Sistem mora preprečil dodajanje predmeta, ki ima enako zaporedje imen profesorjev in enako ime predmeta.

3.1.6.2.2 Sistem mora preprečiti vnos kreditnih točk, če to ni število na intervalu od 0 do 10 s korakom 0,5.

3.1.6.3 Sistem mora v primeru uspešne preverbe zapisati podatke v bazo.

2.2.2 Metoda primerov uporabe

Tudi primeri uporabe so napisani v formalni obliki. Napisan je en sumarni cilj, označen je z znakom '+', sledijo mu uporabniški cilji, ki so označeni z znakom '! '.

Primer uporabe 1: Izpit +
Primarni akter: Profesor, študent
Obseg: E-študent

Stopnja: Sumarni cilj
Tipični potek:

1. Profesor razpiše nov izpitni rok.
2. Študent se prijavi na izpitni rok.
3. Profesor vnese končno oceno.

Razširitve:

1a. Ni mrežne povezave ali je napaka na strani strežnika (opomba: razširitev velja za vse vpise v podatkovno bazo pri vseh primerih uporabe, zato je napisana samo enkrat):

1a1. Sistem opozori uporabnika, primer uporabe se konča.

Primer uporabe 2: Prijava na izpitni rok !
Primarni akter: Študent
Obseg: E-študent
Stopnja: Uporabniški cilj
Sprožilec: Študent izbere "Prijava na izpit"
Tipični potek:

1. Študent izbere letnik in predmet, ki ga ima v predmetniku.
2. Sistem izpiše vse izpitne roke, ki jim še ni potekel rok za prijavo.
3. Študent izbere izpitni rok in označi, če je pisni izpit opravil že s kolokviji.
4. Sistem preveri vnesene podatke.
5. Sistem vpiše podatke v podatkovno bazo.

Razširitve:

4a. Četrta in vsaka naslednja prijava na izpitni rok iz istega predmeta (pri tem ne štejemo polaganj izpitov v času prvega vpisa, če je predmet v predmetniku letnika, ki ga ponavlja) ali študent nima statusa študenta:

4a1. Sistem opozori uporabnika, da je izpitni rok plačljiv.

4b. Sedma prijava na izpitni rok:

4b1. Sistem javi napako o sedmem opravljanju izpitnega roka, primer uporabe se zaključi.

4c. Od prejšnje prijave na izpitni rok iz istega predmeta je minilo manj kot 14 dni in študent je na njem dobil negativno oceno:

4c1. Sistem javi napako o minimalnem obveznem preteku dni od zadnjega opravljanja izpita, primer uporabe se zaključi.

4d. Študent je označil, da je pisni izpit opravil že s kolokviji, vendar kolokvijev ni opravil:

4d1. Sistem javi napako, vrnemo se na točko 3.

Primer uporabe 3: Pregled izpitnih rokov !
Primarni akter: Študent
Obseg: E-študent
Stopnja: Uporabniški cilj
Sprožilec: Študent izbere “Pregled izpitnih rokov”
Tipični potek:

1. Študent izbere letnik, za katerega želi pregledovati izpitne roke.
2. Sistem izpiše vse izpitne roke za v tekočem študijskem letu izbran letnik.

Razširitve:

2a. Razpisan ni noben izpitni rok:

2a1. Sistem nas opozori, da izpitni rok ni razpisan.

Primer uporabe 4: Razpis izpitnega roka !
Primarni akter: Profesor
Obseg: E-študent
Stopnja: Uporabniški cilj
Predpogoj: Predmet je ustvarjen
Sprožilec: Profesor izbere "Razpis izpitnega roka"
Tipični potek:

1. Sistem izpiše vse predmete, ki jih izvaja profesor.
2. Profesor izbere predmet, vnese prostor, možne točke pisnega izpita, pogoje za pristop, datum in uro izpitnega roka.
3. Sistem preveri pravilnost podatkov.
4. Sistem vpiše podatke v podatkovno bazo.

Razširitve:

3a. Za predmet je že razpisan izpitni rok na isti datum:

3a1. Sistem opozori na napako, vrnemo se na točko 1.

3b. Zahtevan podatek ni vpisan (prostor, predmet, datum in ura):

3b1. Sistem opozori na napako, vrnemo se na točko 1.

Primer uporabe 5: Vnos končne ocene !
Primarni akter: Profesor
Obseg: E-študent
Stopnja: Uporabniški cilj
Predpogoj: Izpitni rok je razpisan in trenutni datum je kasnejši ali enak datumu razpisanega roka
Sprožilec: Profesor izbere "Vnos končne ocene"
Tipični potek:

1. Profesor izbere izpitni rok.
2. Sistem izpiše seznam prijavljenih na izpitni rok.
3. Profesor vnese končne ocene ali oznake za vrnjeno prijavnico oziroma kasnejši vnos ocene samo študentom, ki ocene ali oznake za vrnjeno prijavnico še nimajo.
4. Sistem vpiše podatke v podatkovno bazo.
5. Sistem vse prijavljene obvesti o vneseni oceni.

Razširitve:

3a. Vnesena ocena ni celo število med 1 in 10 ali ni oznaka za vrnjeno prijavnico oziroma kasnejši vnos ocene:

3a1. Sistem opozori uporabnika, zahteva ponovni vnos.

3b. Niso vnesene vse ocene ali oznake za vrnjeno prijavnico oziroma kasnejši vnos ocene:

3b1. Sistem opozori uporabnika, zahteva vnos manjkajočih.

5a. Sporočilo ni bilo dostavljeno:

5a1. Sistem opozori uporabnika, katerim študentom sporočilo ni bilo dostavljeno.

Primer uporabe 6: Brisanje izpitnega roka !
Primarni akter: Profesor
Obseg: E-študent
Stopnja: Uporabniški cilj
Sprožilec: Profesor izbere “Brisanje izpitnega roka”
Tipični potek:

1. Profesor izbere izpitni rok.

2. Sistem preveri ali obstajajo prijave na rok.

3. Sistem izbriše izpitni rok.

Razširitve:

2a. Obstajajo prijave na izpitni rok:

2a1. Sistem vpraša uporabnika ali želi vseeno izbrisati rok:

2a1a. Uporabnik se strinja, izbrišejo se vse prijave na rok, študente se obvesti, da je bil izpitni rok izbrisan in primer uporabe se nadaljuje.

2a1b. Uporabnik se ne strinja:

2a1b1. Primer uporabe se zaključi.

Primer uporabe 7: Ustvarjanje predmeta !
Primarni akter: Referent
Obseg: E-študent
Stopnja: Uporabniški cilj
Sprožilec: Referent izbere "Ustvarjanje predmeta"
Tipični potek:

1. Referent določi profesorje, ki izvajajo predmet (do tri), ime predmeta in število kreditnih točk.

2. Sistem preveri vpisane podatke.

3. Sistem vpiše podatke v podatkovno bazo.

Razširitve:

2a. Zaporedje profesorjev in ime predmeta že obstaja:

2a1. Sistem javi napako in se vrne na točko 1.

2b. Vnos kreditnih točk ni število na intervalu od 0 do 10 s korakom 0,5:

2b1. Sistem javi napako in se vrne na točko 1.

2.2.3 Metoda uporabniških zgodb

Uporabniške zgodbe so napisane v formalni obliki. Zgodba je napisana v odebeljeni pisavi, sledijo ji sprejemni testi. Vsak odstavek je svoja zgodba.

Kot študent se želim prijaviti na izpitni rok preko interneta, da lahko to storim od doma

Preveri prvo, drugo in tretjo prijavo redno vpisanega študenta.
Preveri ali je polaganje izpita plačljivo (četrto in vsako naslednje polaganje, študent brez statusa študenta).
Preveri prijavo študenta, ki ponavlja (ne upoštevajo se polaganja izpitov v času prvega vpisa).
Preveri prijavo na izpitni rok in če je od izpitnega roka iz istega predmeta minilo manj kot 14 dni in je študent dobil negativno oceno, se prijava ne izvede.
Preveri sedmo prijavo na izpitni rok (prijava se ne izvede).
Preveri, da se študent lahko prijavi samo na izpitne roke iz predmetov, ki so v njegovem predmetniku in ki jih še ni opravil.
Preveri prijavo na izpitni rok s kolokviji, študent je kolokvije res opravil.
Preveri prijavo na izpitni rok s kolokviji, študent kolokvijev ni opravil (prijava se ne izvede).

Kot študent želim pregledovati izpitne roke preko interneta, da si lažje organiziram potek učenja

Preveri da se izpišejo samo roki izbranega letnika in samo roki za predmete iz študentovega predmetnika.
Preveri, da je uporabnik takrat, ko ni razpisan noben izpitni rok, o tem obveščen.

Kot profesor želim razpisati izpitni rok preko interneta, da lahko razpišem izpitni rok od doma ali iz svoje pisarne

Preveri razpis še enega izpitnega roka za isti predmet na isti dan (se ne sme zgoditi).

Preveri obveznost vnosa datuma, ure, predmeta, prostora in možnih točk izpita .

Kot profesor želim vnesti končne ocene izpitnega roka preko interneta, da lahko ta vnos opravim od doma ali iz svoje pisarne

Preveri vnose ocen med ena in deset.
Preveri, da ima vsak študent po končanih vnosih vneseno oceno ali oznako za vrnjeno prijavnico oziroma kasnejši vnos ocene.
Preveri, da se pri ponovnem ocenjevanju istega roka že vnesenih ocen ne da več spreminjati in da se ne pokaže študentov, ki so pri prejšnjem ocenjevanju dobili vrnjeno prijavnico.
Preveri pošiljanje elektronskega sporočila vsem študentom, ki so dobili oceno.

Kot profesor želim izbrisati izpitni rok preko interneta, da lahko to dejanje opravim od doma ali iz svoje pisarne

Preveri brisanje brez obstoječih prijav na rok.
Preveri brisanje z obstoječimi prijavami na rok in potem preklic brisanja.
Preveri brisanje z obstoječimi prijavami na rok in potem potrjevanje brisanja (študente se obvesti o brisanju roka).

Kot referent želim vnesti predmet preko interneta, da lahko to dejanje opravim v svoji pisarni

Preveri vnos kreditnih točk (samo števila na intervalu od 0 do 10 s korakom 0,5).
Preveri ustvarjanje dodatnega predmeta z istimi izvajalci v enakem vrstnem redu (se ne sme zgoditi).

Poglavje 3

Analiza in primerjava metod

V tem poglavju bomo med seboj primerjali posamezne metode za pisanje specifikacij, omejili pa se bomo na naslednje kriterije: popolnost specifikacij, razumljivost specifikacij za končnega uporabnika sistema, vidik razvijalcev, dopolnjevanje specifikacije, čas izdelave, kompromisi in omejitve uporabe ter splošna primerjava. Vsaka primerjava vsebuje temo in razloge, zakaj je obravnava te sploh smiselna, ter dodatne informacije o obravnavani temi z vidika vsake od metod.

3.1 Popolnost specifikacij

Zaželeno je, da specifikacije na čim boljši način razložijo delovanje sistema. Razumljive naj bi bile tudi, če bi morali ponovno prebrati in razumeti že nekaj let zaključen projekt. Po kateri metodi bi torej najhitreje razumeli delovanje sistema?

Pri metodi uporabniških zgodb dobimo kup kartic ali v kakšnem programu navedenih zgodb, iz katerih potem razberemo delovanje sistema. Ena možnost je, da te zgodbe organiziramo glede na vloge uporabnikov in skušamo ugotoviti, katera, če sploh, sledi kateri. Proces je naporen, pa tudi ni zagotovila, da bomo na koncu prišli do pravilne rešitve. Druga možnost je, da jih razdelimo glede na dodeljene prioritete in skušamo glede na njihovo po-

membnost dojeti, kaj sistem počne. Zaradi formalne zgradbe pa lahko hitro razumemo, kaj skuša uporabnik doseči s posamezno zgodbo.

Pri metodi primerov uporabe je situacija nekoliko bolj preprosta. Zberemo vse sumarne cilje, če smo jih pri projektu kaj zapisali, in iz njih razberemo, kako približno poteka uporabniški proces. Če sumarnih ciljev nismo napisali, nas čaka napornejše opravilo: branje posameznih primerov uporabe in razporejanje na podlagi predpogojev. Tisti primeri, ki predpogoja nimajo, so preprosto začetni ali pa samostojni. Opravilo je hitrejše kot pri metodi uporabniških zgodb, zagotovo pa pridemo tudi do pravilnega ali vsaj približno pravilnega poteka.

IEEE 830 pa nam ponudi celostno podobo delovanja sistema, saj vključuje tako funkcionalne in nefunkcionalne zahteve kot tudi splošen opis sistema. Celoten potek lahko razberemo že pred tretjim poglavjem, ko pridejo na vrsto poglavja *Namen, Obseg* in *Kontekst programskega izdelka.*

3.2 Razumljivost specifikacij za končnega uporabnika sistema

Ker specifikacije potrdi končni uporabnik, mora biti prepričan, da specifikacije pravilno dokumentirajo njegove zahteve. V naslednjih odstavkih bomo primerjali, koliko truda je potrebnega, da uporabnik razume po posameznih metodah napisane specifikacije.

Ker pri pisanju zgodb uporabnik sodeluje z nekom, ki metodo dobro pozna, in ker zgodbe ne vsebujejo strokovnih besed, je ta metoda izmed vseh treh uporabniku najbolj prijazna. Uporabnik bi na podlagi prebrane zgodbe in sprejemnih testov moral hitro doumeti, kaj poskuša zgodba oziroma specifikacija razložiti. Razložiti mu je potrebno samo to, kako naj bere sprejemne teste. Uporabniške zgodbe so tudi kratke in jedrnate, kar je ugodno za branje.

Primeri uporabe imajo predpisano obliko in sestavo. Ko se uporabnik nanju navadi, branje primerov uporabe poteka hitreje. Dejstvi, da so napisani

v bolj tehničnem jeziku in vsebujejo pojme, kot so *stopnja, obseg, predpogoj, sprožilec,* mu lahko sprva otežujeta razumevanje primerov uporabe. Zaradi predpisane strukture primeri uporabe za zapis specifikacije uporabljajo več besed v primerjavi z uporabniškimi zgodbami.

IEEE 830 obsega ves dokument z veliko več informacijami, kot jih uporabnik potrebuje za razumevanje sistema. Dokument na primer vsebuje tudi podatke o tem, kakšna programska oprema je potrebna na strežniški strani, katero podatkovno bazo bomo uporabili itn. Specifikacije vsebujejo veliko ponavljajočih se besed, ki ne pripomorejo k razumevanju specifikacij in so tudi za končnega uporabnika težje berljive kot ostali dve metodi, saj je njihovo oštevilčevanje preveč razčlenjeno. Ta standard je v primerjavi z drugima dvema metodama najobsežnejši, vendar ta lastnost ne pripomore veliko k razumevanju.

3.3 Vidik razvijalcev

Specifikacije so namenjene v prvi vrsti razvijalcem, zato bomo sedaj predstavili prednosti in slabosti posameznih metod z njihovega vidika.

Prednost uporabniških zgodb je preverjanje pravilnosti implementacije zgodbe tako, da s pomočjo sprejemnih testov preverjamo pravilnost. Sprejemni testi zaobjemajo vse za popolno implementacijo potrebne kriterije. Napisani so tako, da jih lahko enostavno odkljukamo, ko je test potrjen. Če imamo program, ki samodejno testira pravilnost implementacije, mu lahko kot kriterije za pregledovanje določimo ravno te teste.

Zgradba primerov uporabe je taka, da napeljuje k implementaciji specifikacij po v razdelku *Tipični potek* ali *Razširitve* napisanem poteku. Opisan imamo torej postopek, zato lažje razumemo, kako bomo prišli do želenega rezultata. Pisec primerov uporabe mora biti izkušen, da zna izbrati pravilen in najbolj optimalen potek. Podobno velja tudi za razdelek Razširitve, le da gre tam za alternativni potek oziroma zaporedje javljanja napak. Primeri uporabe so vedno napisani po točno določeni predlogi, le postavke, ki bi

ostale prazne, lahko izpustimo. Ko se navadimo efektivnega branja primerov uporabe, je delo še lažje. Ostali dve metodi teh prednosti nimata.

IEEE 830 vsebuje veliko podatkov, ki jih razvijalec pri programiranju ne potrebuje. Dokument je zato po nepotrebnem preobsežen, nekoristno pa je tudi preveč razvejano oštevilčevanje razdelkov pri specifikacijah, kar razvijalcu otežuje branje. K težji berljivosti pa svoje dodajo tudi ponavljajoče se besedne zveze, kot so "Sistem mora."

3.4 Dopolnjevanje specifikacije

Pri pisanju specifikacij se lahko zgodi, da pisec pozabi upoštevati kakšno pomembno lastnost specifikacije. Ta pomanjkljivost se praviloma pokaže med postopkom implementacije specifikacije. S to primerjavo bomo ugotavljali, kako enostavno je dodajanje teh manjkajočih lastnosti pri vsaki od metod.

Pri metodi uporabniških zgodb lahko to manjkajočo lastnost enostavno dopišemo k sprejemnim testom s povedjo, ki se začne s "Preveri..." Na kartico dopišemo želene sprejemne teste, če pa nam na njej slučajno zmanjka prostora, jo podaljšamo z dodatnim kosom papirja. Pri uporabi programske opreme za pisanje uporabniških zgodb pa dodatne vrstice kar dopišemo, saj nas fizične dimenzije ne omejujejo.

Nekoliko težje pa je dodajanje teh lastnosti pri primerih uporabe. Kadar lastnosti dodajamo k tipičnemu poteku, s tem spremenimo tudi zaporedne številke korakov. Paziti moramo na to, da ustrezno spremenimo tudi oznake pri razširitvah, saj so vse razširitve vezane na točno določen korak tipičnega poteka. Pri dodajanju lastnosti tipičnemu poteku in razširitvam pa moramo paziti tudi na smiselno dodajanje k že obstoječim korakom. Če dobimo že natisnjene primere uporabe, moramo spremembe najprej vnesti v programsko opremo. To je lahko težavna naloga, če besedilo ni napisano ravno v programski opremi, ki bi bila namenjena pisanju primerov uporabe.

Najtežje pa je dodajanje lastnosti standardu IEEE 830, razlog pa je zopet v preveč razčlenjenem oštevilčevanju. Pri vrivanju moramo biti pozorni na

to, da se oštevilčevanje spremeni tudi vsem sledečim korakom. Opozoriti je potrebno še na to, da si to delo lahko nekoliko olajšamo, če pri pisanju IEEE 830 v urejevalniku besedila uporabljamo vgrajeno samodejno oštevilčevanje. Namenske programske opreme za pisanje standarda nismo našli.

3.5 Čas izdelave, kompromisi in omejitve uporabe

Pri pisanju specifikacij zahtev je pomemben čas izdelave specifikacij. Zavedati se moramo, da čas pogojuje tudi to, čemu se moramo odpovedati.

Najmanj časa nam vzame pisanje uporabniških zgodb, saj se nam ni treba ukvarjati s potekom in nefunkcionalnimi zahtevami ter raznimi dodatki (slike, diagrami itd.). Ni nujno, da so sprejemni testi že med pisanjem popolni. Sprejemne teste dopolnjujemo med samim razvojem, metoda namreč daje poudarek komunikaciji med razvijalci in produktnim vodjo oziroma končnimi uporabniki. Ta metoda je primerna za projekte, pri katerih je pomembna hitra izdelava, pri katerih nimamo veliko funkcionalnih zahtev in pri katerih sodelujemo z izkušenimi ljudmi. Zanjo pa se odločimo tudi, če smo pri podobnih projektih že sodelovali.

Pisanje primerov uporabe traja nekoliko dlje, ker moramo razmisliti še o poteku, o nefunkcionalnih zahtevah pa ne, saj nam dodatke do neke mere nadomešča potek. Metoda primerov uporabe glede na čas izdelave in kompromise predstavlja srednjo pot med opisanimi tremi metodami. Metodo uporabljamo pri projektih, pri katerih opis poteka veliko pripomore k lažji implementaciji. To metodo bi uporabili, če bi na primer prvič izdelovali spletno trgovino ali program za blagajno v trgovini.

Za pisanje IEEE 830 pa potrebujemo nekoliko več časa, saj moramo razmisliti o ciljih, nefunkcionalnih zahtevah in morebitnih dodatkih. Pri tej metodi je najbolj pomembna čimbolj podrobna razlaga sistema v izdelavi. Metoda je primerna za projekte, pri katerih je velik poudarek na nefunkcionalnih zahtevah (podroben opis vmesnikov, uporabljen programski jezik

itd.) in pri katerih je potrebno slikovno gradivo. Uporaba te metode bi bila primerna pri razvoju programske opreme, ki lahko naredi veliko škode (medicinski aparati, sistemi v vesoljskih plovilih).

3.6 Splošna primerjava

Na koncu bomo na splošno primerjali vse obravnavane metode.

Uporabniške zgodbe imajo kratek obseg, kljub temu pa nam nudijo vse potrebne informacije za razvoj sistema. Zgradba je preprosta in lahko razumljiva tako uporabnikom kot razvijalcem. Kot primer nam lahko služi zgodba o pregledovanju izpitnih rokov. Zgodbo lahko razumemo tudi brez sprejemnih testov oziroma dodatnih pojasnil. Temu primerno kratka je zato tudi zgodba. V prvi vrsti so bile uporabniške zgodbe razvite za agilni razvoj programske opreme, načeloma pa jih lahko uporabljamo tudi pri klasičnem slapovnem (angl. waterfall) modelu razvoja.

Primeri uporabe so obsežnejši kot uporabniške zgodbe, k temu pripomoreta predvsem njihova sestava in uporaba predpisanih fraz. Pri branju primera uporabe "Pregled izpitnih rokov" hitro razumemo, kaj se od implementacije pričakuje, kljub temu pa je primer uporabe relativno dolg. Zaradi kompleksnejše zgradbe je za njihovo razumevanje potrebno določeno predznanje. Primere uporabe lahko uporabljamo tako pri agilnem kot tudi pri slapovnem modelu razvoja. Pri uporabniških zgodbah velike zgodbe razdrobimo na več manjših, pri primerih uporabe pa namesto tega napišemo sumarne cilje, potem pa iz njih izpeljemo uporabniške cilje. Primeri uporabe pri dopolnjevanju specifikacij niso tako prilagodljivi kot uporabniške zgodbe.

IEEE 830 ima med obravnavanimi metodami največji obseg. Nanj v veliki meri vpliva večja raznolikost podatkov, kot jo zahtevata drugi dve metodi. Ta standard zahteva tudi pisanje poglavij, ki jih drugi dve metodi ne poznata in ne zahtevata. Ta poglavja natančneje predstavijo kontekst in nefunkcionalne zahteve sistema. Različnim poglavjem tega standarda lahko dodamo dopolnilno gradivo, kot so na primer slike, diagrami in podobno. Drugi dve

metodi pa v osnovi nista namenjeni uporabi dopolnilnega gradiva. Metoda se najbolje obnese pri slapovnem modelu razvoja, za agilni razvoj pa ni preveč primerna, saj ne vsebuje nobene za njegov razvoj uporabne lastnosti, saj ta standard zaradi že prej omenjenih razlogov težko dinamično razširjamo, pa tudi poglavje *Kontekst programskega izdelka* je napisano preveč na splošno, da bi lahko nadomestilo na primer sumarne cilje pri primerih uporabe. Druga poglavja, ki bi nekoliko podrobneje, ampak vseeno ne tako natančno kot specifikacije, opisovala posamezne izseke, pa ne obstajajo.

Poglavje 4

Zaključek

Vsaka od metod ni primerna izbira pri vsakem projektu. Glede na potrebe projekta, razvijalcev in/ali uporabnikov se odločimo, katera izmed njih bi bila v določenem primeru najbolj uporabna. Pri izdelavi aplikacije E-študent bi metodo za pisanje specifikacij izbral glede na izkušenost razvijalcev in dostopnost končnih uporabnikov. Če bi vedel, da bom od končnih uporabnikov med razvojem dobival koristne informacije, bi v vsakem primeru izbral metodo uporabniških zgodb. Iz lastnih izkušenj vem, da se med razvojem sistema velikokrat pojavijo nova vprašanja, ravno pri uporabniških zgodbah pa se da specifikacije najlažje dopolnjevati. Za uporabniške zgodbe bi se odločil tudi takrat, kadar bi sodeloval z bolj izkušeno skupino, vendar pa na voljo ne bi imel dovolj končnih uporabnikov oziroma koristnih informacij z njihove strani. V takem primeru bi nam pisanje zgodb vzelo manj časa, pa tudi opis poteka zaradi izkušenosti razvijalcev ne bi bil potreben. Kadar sistem razvija manj izkušena skupina, pa so po mojem mnenju primernejši primeri uporabe, saj zahtevo opisujejo s potekom, kar pripomore k boljšemu razumevanju zahteve. Standard IEEE 830 bi uporabil pri razvoju E-študenta, ki ga ne bi uporabljala le ena izmed fakultet, ampak cela univerza. Pri takem razvoju bi bil v primerjavi s zgornjimi primeri večji poudarek na nefunkcionalnih zahtevah, kot so na primer dosegljivost, zanesljivost, različne nastavitve glede na potrebe različnih fakultet... Za povprečno komercialno

podjetje s povprečno izkušenimi zaposlenimi so, po mojem mnenju, v večini primerov najbolj uporabne uporabniške zgodbe.

Imam osebne izkušnje pri dolgoletnem razvoju programske opreme s področja financ. Ker že obstoječi programski paket le dopolnjujemo s funkcionalnostmi, za pisanje specifikacij uporabljamo prilagojeno metodo uporabniških zgodb. Zapišemo samo zgodbo, ne pa tudi sprejemnih testov, saj jih veliko predvidimo že sami. Manjkajoče dele specifikacije dobimo v obliki ustnih navodil, med samim razvojem pa dodatne informacije dobimo še od vodje projekta, končnih uporabnikov ali tehnične podpore. V primeru, da gre za obsežnejšo zahtevo pa specifikacije dobimo v nekoliko prilagojeni obliki neformalnih primerov uporabe, ki se od običajnih razlikujejo v tem, da so daljši in se mestoma dotikajo implementacije. Če je zahtevana tudi implementacija vmesnika, pa poleg dobimo še podrobnejšo dokumentacijo, na primer predpisano zgradbo XML dokumenta, protokol spletnega servisa (angl. web service) itd.

Do zgornjih zaključkov sem prišel med pisanjem tega diplomskega dela ter na podlagi dosedanjih delovnih izkušenj. Če bi na delovnem mestu imel manj stika s končnimi uporabniki oziroma če bi imel veliko več delovnih izkušenj, bi mogoče na metode znal pogledati tudi iz drugih zornih kotov, zato bi bili po vsej verjetnosti tudi moji zaključki drugačni.

Literatura

[1] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2000

[2] M. Cohn, *User Stories Applied: For Agile Software Development*, Addison-Wesley, 2011

[3] I. Jacobson, "Object-oriented Development in an Industrial Environment", v zborniku: OOPSLA '87 Conference proceedings on Object-oriented programming systems, languages and application, št. 12, zv. 22, str. 183–191, 1987.

[4] I. Jacobson [et al.], *Object Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992

[5] R. Jeffries, Essential XP: Card, Conversation, Confirmation, dostopno na:
http://xprogramming.com/articles/expcardconversationconfirmation/

[6] E. R. Keith, *Agile Software Development Processes: A Different Approach to Software Design*, dostopno na:
http://www.cs.nyu.edu/courses/spring03/V22.0474-001/lectures/agile/AgileDevelopmentDifferentApproach.pdf

[7] 830-1984 - IEEE Guide to Software Requirements Specifications, dostopno na:
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=278253

[8] 830-1993 - IEEE Recommended Practice for Software Requirements Specifications, dostopno na:
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=392555

[9] 830-1998 - IEEE Recommended Practice for Software Requirements Specifications, dostopno na:
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=720574

Dodatek: Specifikacija zahtev programske opreme po standardu IEEE 830-1998

E-študent

Specifikacija zahtev programske opreme

1.0

23.5.2014

Miha Klun
Vodja razvijalcev

Pripravljeno za
Fakulteto za računalništvo in informatiko

Zgodovina popravkov

Datum	**Opis**	**Avtor**	**Komentarji**
23.5.2014	Dokument ustvarjen	Miha Klun	Prvi vnos

Kazalo

1 Uvod 5
1.1 Namen 5
1.2 Obseg 5
1.3 Definicije in okrajšave 5
1.4 Viri 5
1.5 Pregled 5
2 Splošni opis 6
2.1 Kontekst programskega izdelka 6
2.1.1 Vmesniki sistema 6
2.1.2 Uporabniški vmesniki 6
2.1.3 Strojni vmesniki 6
2.1.4 Programski vmesniki 6
2.2 Komunikacijski vmesniki 7
2.2.1 Spominske omejitve 7
2.2.2 Operacije 7
2.2.3 Prilagoditve ob namestitvi 7
2.3 Funkcije izdelka 7
2.4 Značilnosti uporabnikov 8
2.5 Omejitve 8
2.6 Predpostavke in odvisnosti 8
3 Specifične zahteve 8
3.1 Funkcionalne zahteve 8
3.1.1 Sistem mora študentu omogočati prijavo na izpitni rok 8
3.1.2 Sistem mora omogočati pregled izpitnih rokov 9
3.1.3 Sistem mora omogočati razpis izpitnega roka . 9
3.1.4 Sistem mora omogočati vnos končne ocene . . 10
3.1.5 Sistem mora omogočati brisanje izpitnega roka 10
3.1.6 Sistem mora omogočati kreiranje predmeta . . 11
3.2 Lastnosti sistema 11

3.2.1 Zanesljivost 11
3.2.2 Dosegljivost 11
3.2.3 Varnost 11

1 Uvod

1.1 Namen

Namen tega dokumenta je razvijalcem in preizkuševalcem sistema na jasen način opisati specifikacije zahtev za sistem E-študent.

1.2 Obseg

Naredili bomo spletno stran E-študent, s katero se bodo študenti lahko od doma prijavljali na izpitne roke in pregledovali svoje ocene. Profesorji bodo razpisovali izpitne roke in jih zatem tudi ocenili. Referentom bo omogočeno enostavno kreiranje predmetov.

1.3 Definicije in okrajšave

Predmet – učno področje, določeno z izvajalci in številom kreditnih točk. **Predmetnik** – predpis, ki za šolo določene stopnje določa učne predmete po razredih in število tedenskih ur zanje. **Izpit** – postopek, pri katerem se ugotavlja in vrednoti znanje, sposobnost kandidata. **Izpitni rok** – rok za opravljanje izpita, ki je določen s predmetom, datumom, uro, prostorom in številom možnih točk. **Študijsko leto** – obdobje od začetka oktobra do konca septembra.

1.4 Viri

IEEE Std. 830-1998: IEEE Recommended Practice for Software Requirements Specifications

1.5 Pregled

Poglavje 2 vsebuje splošen opis E-študenta, specifikacije zahtev pa so v poglavju 3.

2 Splošni opis

2.1 Kontekst programskega izdelka

Študenti morajo, če se želijo prijaviti na izpitni rok, iti na fakulteto in se prijavljati od tam. E-študent jim bo omogočil opravljanje tega od doma ter ob urah, ob katerih je fakulteta zaprta. Z možnostjo izpisa svojih izpitnih rokov bodo lažje načrtovali potek učenja. Profesorji bodo razpisovali roke enostavno, sistem pa bo poskrbel, da se jim izpitni rok iz istega predmeta slučajno nebi prekrival. Prav tako jim bo olajšal obveščanje študentov o uspehu na izpitu s samodejnim pošiljanjem elektronske pošte po vnosu ocen. Referenti bodo s pomočjo sistema lažje kreirali predmete.

2.1.1 Vmesniki sistema

Sistem bo sestavljen iz uporabniškega modula, strežniškega modula in podatkovne baze. Uporabniški modul teče v uporabnikovem spletnem brskalniku in preko interneta komunicira s strežniškim modulom, strežniški modul pa komunicira z uporabniškim modulom in podatkovno bazo.

2.1.2 Uporabniški vmesniki

Uporabniški vmesnik aplikacije bo napisan tako, da bo funkcionalen in pravilno prikazan na spletnih brskalnikih Mozilla Firefox 26, Internet Explorer 9, Google Chrome 30 in na njihovih kasnejših verzijah.

2.1.3 Strojni vmesniki

Vse komponente se morajo izvajati na osebnih računalnikih.

2.1.4 Programski vmesniki

Uporabniški modul mora podpirati JavaScript 1.8.0, na strežniku mora biti nameščen Internet Information Services (krajše IIS) verzije 7.0 ali novejše, podatkovna baza mora biti SQL Server 2005 ali novejši.

2.2 Komunikacijski vmesniki

Uporabniški modul komunicira s strežniškim modulom preko protokola TCP/IP, IIS in SQL Server pa naj bosta nameščena na istem računalniku.

2.2.1 Spominske omejitve

Uporabniški modul zahteva vsaj 1 GB spomina (vključno s spletnim brskalnikom), prav tako vsaj 1 GB pa potrebujeta tudi IIS in SQL Server.

2.2.2 Operacije

Uporabniški modul mora biti enostaven za uporabo ter ne sme zahtevati posebnega predhodnega znanja. IIS naj bo nameščen tako, da ne moti nič drugega, kar je že nameščeno na strežniku. Pri podatkovni bazi naj bo nastavljeno samodejno shranjevanje varnostnih kopij in njihova vzpostavitev ob kvarjenju podatkovne baze.

2.2.3 Prilagoditve ob namestitvi

Noben modul ne zahteva prilagajanja ob namestitvi.

2.3 Funkcije izdelka

E-študent bo referentom na fakulteti omogočil kreiranje predmetov in predmetnikov ter popravljanje osebnih podatkov, vpisnih listov in predmetnikov študentov. Profesorjem bo omogočal razpis izpitnih rokov in njihovo brisanje, po zaključenem pisnem delu pa bo lahko vnesel točke pisnega dela ter končne ocene. Omogočen bo tudi vnos ocen s prijavnic. Sistem bo profesorjem izpisal prijave na izpitni rok ter omogočal odjavo študentov od izpitnega roka.

Študentje se bodo lahko preko sistema vpisali v prvi oziroma višji letnik. Tekom študija bodo lahko pregledovali razpisane izpitne roke ter se nanje prijavili oziroma se od njih odjavili. Po opravljenih izpitih pa bodo imeli tudi možnost pregledovanja svojih ocen.

2.4 Značilnosti uporabnikov

Uporabniki so vešči osnovnega brskanja po spletu ter poznajo osnovne pojme, ki jih srečujemo pri študiju na fakulteti, kot so izpit, predmet, predmetnik, izpitni rok, prijavnica na izpit,...

2.5 Omejitve

Sistem mora nuditi osnovno zaščito pri prijavi, hramba podatkov pa mora biti zanesljiva.

2.6 Predpostavke in odvisnosti

Predpostavljeno je, da je na strežniku nameščen operacijski sistem Windows Server verzije 2003 ali novejši.

3 Specifične zahteve

3.1 Funkcionalne zahteve

3.1.1 Sistem mora študentu omogočati prijavo na izpitni rok

3.1.1.1 Sistem mora izpisati letnike, predmete, ki jih ima študent v predmetniku in pripadajoče izpitne roke, ki jim še ni potekel rok za prijavo.

3.1.1.2 Sistem mora omogočati študentu prijavo na izpitni rok in označbo, ali je pisni izpit že opravil s kolokviji.

3.1.1.3 Sistem mora preveriti, ali je študent pisni izpit res opravil s kolokviji in ali mora izpitni rok plačati.

3.1.1.3.1 Sistem mora preveriti, ali študent plača izpitni rok.

3.1.1.3.1.1 Študent brez statusa plača vsa polaganja.

3.1.1.3.1.2 Študent s statusom plača četrto in nadaljnja polaganja.

3.1.1.3.2 Sistem mora pri sedmi prijavi na izpitni rok preklicati prijavo in opozoriti uporabnika o napaki.

3.1.1.3.3 Sistem mora preklicati prijavo, če je od prijave na izpitni rok iz istega predmeta minilo manj kot 14 dni in je študent na njem dobil negativno oceno.

3.1.1.3.4 Sistem mora preprečiti prijavo študenta na izpitni rok če je označil, da je opravil kolokvije, vendar jih ni.

3.1.1.4 Sistem mora pravilno določiti zaporedno številko polaganja.

3.1.1.4.1 Določi se zaporedna številka polaganja, če študent ponavlja letnik, se polaganja v času prvega vpisa ne upoštevajo.

3.1.1.5 Sistem mora v primeru uspešne preverbe zapisati podatke v bazo.

3.1.2 Sistem mora omogočati pregled izpitnih rokov

3.1.2.1 Sistem mora izpisati vse letnike, ki jih je študent do sedaj že obiskoval.

3.1.2.2 Sistem mora izpisati vse izpitne roke za izbrani letnik v tekočem študijskem letu.

3.1.3 Sistem mora omogočati razpis izpitnega roka

3.1.3.1 Sistem mora ponuditi na izbiro vse predmete, ki jih izvaja profesor.

3.1.3.2 Sistem mora po izboru predmeta vprašati za določitev prostora izpitnega roka, možne točke pisnega izpitnega roka, datum in uro roka.

3.1.3.3 Sistem mora preveriti vnesene podatke.

3.1.3.3.1 Sistem mora preveriti prisotnost in resničnost podatkov (prostor, datum, točke).

3.1.3.3.2 Sistem mora preveriti, ali je na isti datum že razpisan rok za izbran predmet in če je, zahtevati ponovni vnos datuma.

3.1.3.4 Sistem mora v primeru uspešne preverbe zapisati podatke v bazo.

3.1.4 Sistem mora omogočati vnos končne ocene

3.1.4.1 Sistem mora omogočati izbiro izpitnega roka.

3.1.4.2 Sistem mora po izbiri izpisati seznam tistih prijavljenih na izpitni rok, ki ocene ali oznake za vrnjeno prijavnico še nimajo.

3.1.4.3 Sistem mora preveriti vnesene podatke.

3.1.4.3.1 Vnesene ocene morajo biti cela števila med ena in deset.

3.1.4.3.2 Vnašajo se lahko še oznake za vrnjeno prijavnico ali kasnejši vnos ocene.

3.1.4.3.3 Na koncu vnosa mora imeti vsak študent vneseno oceno ali oznako.

3.1.4.4 Sistem mora v primeru uspešne preverbe zapisati podatke v bazo.

3.1.4.5 Sistem mora po uspešnem vnosu ocen vsem prijavljenim študentom poslati sporočilo o oceni.

3.1.4.5.1 V primeru nedostavljenega sporočila mora sistem o tem opozoriti uporabnika.

3.1.5 Sistem mora omogočati brisanje izpitnega roka

3.1.5.1 Sistem mora izpisati vse razpisane izpitne roke profesorja.

3.1.5.2 Po izbiri predmeta mora sistem, če prijave na izpitni rok obstajajo, uporabnika vprašati, če vseeno želi izbrisati izpitni rok.

3.1.5.2.1 Če se uporabnik s tem strinja, mora sistem izbrisati vse prijave na izpitni rok in o tem obvestiti študente.

3.1.5.2.2 Če se uporabnik s tem ne strinja, se brisanje prekliče.

3.1.5.3 Sistem mora v primeru uspešne preverbe zapisati podatke v bazo.

3.1.6 Sistem mora omogočati kreiranje predmeta

3.1.6.1 Sistem mora na strani za kreiranje predmeta omogočati vnos imen profesorjev, ki izvajajo predmet, imena predmeta in števila kreditnih točk.

3.1.6.2 Sistem mora po potrditvi vnosa s strani uporabnika preveriti podatke.

3.1.6.2.1 Sistem mora preprečil dodajanje predmeta, ki ima enako zaporedje imen profesorjev in enako ime predmeta.

3.1.6.2.2 Sistem mora preprečiti vnos kreditnih točk, če to ni število na intervalu od 0 do 10 s korakom 0,5.

3.1.6.3 Sistem mora v primeru uspešne preverbe zapisati podatke v bazo.

3.2 Lastnosti sistema

3.2.1 Zanesljivost

Sistem mora imeti odzivni čas dve sekundi, kar pomeni, da se morajo vse kalkulacije in transakcije izvesti v manj kot dveh sekundah.

3.2.2 Dosegljivost

Podatkovna baza naj na vsako uro ustvari varnostno kopijo, zato da se izgubi čim manj podatkov. Sistem naj bo razpoložljiv 99.9% časa.

3.2.3 Varnost

Uporabniška gesla morajo biti v podatkovni bazi shranjena z algoritmom SHA-256. Zahtevajo naj se gesla vsaj dolžine 8 znakov, sestavljena iz številk in črk.