Univerza na Primorskem
FAMNIT

Programsko inženirstvo

Skripta

povzetek predavanj

2013
Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Kazalo

1 Uvod **6**
1.1 Vsebina predmeta 6
1.2 Programski produkt 6
1.3 Delitev programske opreme 6
1.4 Posebnosti programskega inženirstva 6
1.5 Stopnja okvar 7
1.6 Uspešnost programskim produktov 7
1.7 Razlogi za neuspeh 7
1.8 Cena programskih produktov 8
1.9 Lastnosti dobrih programskih produktov 8
1.10 Inženirski pristop 8

2 Programski proces **9**
2.1 Faze programskega procesa 9
2.1.1 Definicija problema 9
2.1.2 Študija izvedljivosti 10
2.1.3 Analiza in definiranje zahtev 11
2.1.4 Načrtovanje sistema 13
2.1.5 Načrtovanje programa 13
2.1.6 Izvedba 14
2.1.7 Testiranje 14
2.1.8 Predaja produkta 14
2.1.9 Obratovanje in vzdrževanje 15
2.2 Razvojni modeli programskega procesa 15
2.2.1 Prednosti formalnih procesov 15
2.2.2 Prednosti neformalnih procesov 15
2.3 Razvojni modeli programskih procesov 16
2.3.1 Build and fix 16
2.3.2 Linearni model 16
2.3.3 Linearni model s povratki 16
2.3.4 "V" model 17
2.3.5 Spiralni model 17
2.3.6 Razvoj z uporabo prototipov 18
2.3.7 Evolucijski razvoj 18
2.3.8 Razvoj z opustitvijo prototipa 19
2.3.9 Postopni (fazni) razvoj 19
2.3.10 Unified Process (UP) 20
2.4 Ogrodja programskih procesov 21
2.5 Zrelostni nivoji programskih procesov 21

2.6 Cena programskega inženirstva 21

3 Diagramske tehnike 22
3.1 Zakaj diagrami? 22
3.2 Diagrami in modeli 22
3.2.1 Vrste 22
3.2.2 Nivoji modelov 22
3.3 Diagrami pri načrtovanju programskih produktov 23
3.3.1 Diagram poteka (ang. flowchart) 23
3.3.2 Diagram podatkovnega toka (ang. Data Flow Diagram) 23
3.3.3 Entitetno relacijski diagram - ER 23
3.3.4 UML diagrami 23

4 Analiza zahtev 26
4.1 Problemi analize 26
4.2 Specifikacija zahtev 26
4.2.1 Postopek 26
4.3 Seznanjanje zahtev 26
4.3.1 Problem obsega sistema 26
4.3.2 Problemi razumevanja zahtev 27
4.3.3 Problemi spremenljivosti zahtev 27
4.4 Zbiranje zahtev 27
4.4.1 Študij literature 27
4.4.2 Nasveti strokovnjakov 28
4.4.3 Spraševanje (ankete) 28
4.4.4 Izpeljave iz obstoječega sistema 28
4.4.5 Sinteze iz lastnosti okolice 28
4.4.6 Izdelava prototipov 28
4.5 Atomske zahteve 29
4.6 Analiza, modeliranje in usklajevanje 29
4.7 Recenzija 29
4.7.1 Recenzija specifikacije zahtev 29
4.7.2 Tipi recenzij 30
4.8 Upravljanje z zahtevami 30

5 Načrtovanje sistema 31
5.1 Stabilna arhitektura 31
5.2 Načrtovalski cilji 31
5.2.1 učinkovitost (ang. performance) 31
5.2.2 zanesljivost (ang. dependability) 31
5.2.3 cena (ang. cost) 32

5.2.4 zmožnost vzdrževanja (ang. maintainance) 32
5.2.5 uporabniške zahteve (ang. user criteria) 33
5.3 Dekompozicija programskega sistema 33
5.3.1 “Divide and conquer” 33
5.3.2 Koncept podsistemov 33
5.4 Sloji in particije 33
5.4.1 Sloji (ang. layers) 33
5.4.2 Particije (ang. partitions) 33
5.5 Kohezija podsistemov 34
5.5.1 Funkcijska kohezija 34
5.5.2 Slojna kohezija 34
5.5.3 Komunikacijska kohezija 34
5.5.4 Sekvenčna kohezija 34
5.5.5 Proceduralna kohezija 34
5.5.6 Časovna kohezija 34
5.5.7 Kohezija pripomočkov 34
5.6 Modeli odvisnosti 34
5.6.1 Matrika odvisnosti 34
5.6.2 Graf odvisnosti 34
5.7 Relacije med programsko in strojno opremo 34
5.8 Upravljanje s perzistentnimi podatki 35
5.9 Politika nadzora dostopa 35
5.10 Mehanizem globalnega kontrolnega toka 35
5.11 Obravnavanje mejnih primerov 35
5.12 Arhitekturni vzorci 35
5.12.1 “Layers” 35
5.12.2 “Model/View/Controller” 35
5.12.3 “Shared repository” 35
5.12.4 “Microkernel” 35
5.12.5 “Client/Server” 35
5.12.6 “Peer-to-peer” 35
5.12.7 “Pipes and filters” 35

6 Načrtovanje programa 36
6.1 Objektni in strukturni pristop 36
6.2 Usmeritve pri načrtovanju 36
6.2.1 Dedovanje (ang. inheritance) 36
6.2.2 Delegiranje in dedovanje 36
6.2.3 Liskov princip zamenjave - LSP 37
6.2.4 Abstrakcija 37
6.2.5 Odprto/zaprti princip - (ang. the open/closed principle) 37

6.2.6 Ovijanje (ang. encapsulation) 37
6.2.7 Kohezija razreda (ang. class cohesion) 38
6.2.8 Modularnost 38
6.3 Načrtovalski vzorci 39
6.3.1 Delitev načrtovalskih vzorcev 39
6.4 Dokumentiranje 39
6.4.1 Specifikacija vmesnikov 40

7 Testiranje 41
7.1 Verifikacija in validacija 41
7.2 Recenzija 41
7.3 Testiranje 41
7.3.1 Principi testiranja 41
7.3.2 Kdo testira? 42
7.3.3 Testabilnost (ang. testability) 42
7.4 Načini testiranja 43
7.4.1 Strukturno testiranje (white-box testing) 43
7.4.2 Vedenjsko testiranje (black-box testing) 43
7.5 Razhroščevanje 43

8 Vzdrževanje 44
8.1 Cena vzdrževanja 44
8.2 Evolucija 44
8.2.1 Evolucijski tipi programskih produktov 45
8.3 Lehmanovi zakoni evolucije 45
8.3.1 1. Law of continuing change 45
8.3.2 2. Law of increasing complexity 45
8.3.3 6. Law of continuous growth 46
8.3.4 7. Law of declining quality 46
8.4 Veljavnost Lehmanovih zakonov 46
8.5 Proces vzdrževanja 46
8.6 Tehnike in orodja za vzdrževanje 47
8.6.1 Inženiring (ang. forward engineering) 47
8.6.2 Obratni inženiring (ang. reverse engineering) 47
8.6.3 Ponovni inženiring (ang. Reengineering) 47
8.7 Racionalnost 47

9 Uporabniški vmesniki 48
9.1 CLI in GUI 48
9.1.1 Prednosti GUI 48
9.2 Principi pri načrtovanju GUI 48

9.2.1 Miselni modeli (ang. mental models) 48
9.2.2 Metafora (ang. metaphor) 48
9.2.3 Evidentna funkcionalnost (ang. affordance) 48
9.2.4 Princip povezanosti (ang. mapping) 49
9.2.5 Princip povratnih informacij (ang. feedback) 49
9.2.6 Omejitve 49
9.3 Prikaz informacij 50
9.3.1 Analogno in digitalno 50
9.3.2 Uporaba barv 50
9.3.3 Uporaba pogovornih oken 50
9.3.4 Obvestila o napakah 50
9.4 Načrtovanje uporabniških vmesnikov 51
9.4.1 Zmote 51
9.4.2 Značaji programov 51
9.4.3 Način dela 52
9.4.4 The Seven Stages of Action 53
9.4.5 Dokumentiranje 53
9.4.6 Vrednotenje uporabniških vmesnikov 53

1 Uvod

1.1 Vsebina predmeta

Življenjski cikel programskih produktov

- Vse faze življenja programskih produktov
- Načini upravljanja programskih procesov
- Napotki za uspešen razvoj programskih produktov.

1.2 Programski produkt

- **Računalniški programi** (programska oprema), ki omogočajo, da z njihovim izvajanjem dosežemo željen učinek.
- **Podatkovne strukture**, ki programom omogočajo manipulacijo (obdelavo) podatkov (informacij).
- **Dokumenti**, ki opisujejo delovanje in uporabo programov

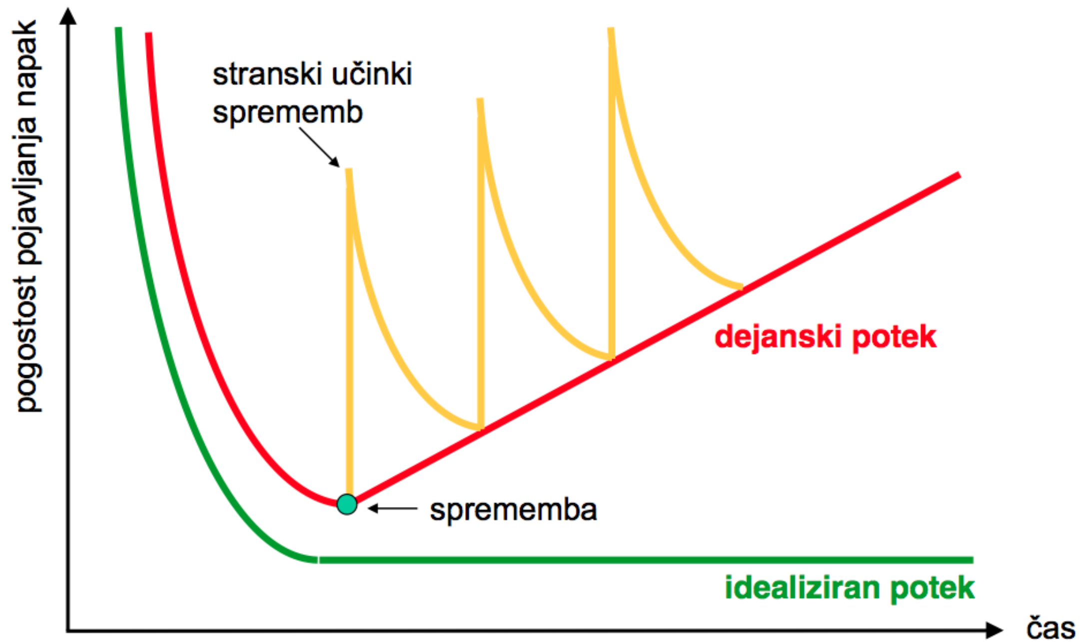
1.3 Delitev programske opreme

- Sistemska
- Za delo v realnem času (real-time)
- Poslovna
- Znanstvena
- Vgrajena (embedded)
- Osebna (za osebno uporabo na osebnih računalnikih)
- Mrežna
- Umetna inteligenca

1.4 Posebnosti programskega inženirstva

Programska oprema vsebuje skrita stanja kot je na primer veliko število nevidnih spremenljivk. Programske opreme ne "proizvajamo"v klasičnem pomenu besede, pač pa jo razvijamo. Programska oprema je v večini primerov izdelana po naročilu. Programska oprema je pogosto najdražji del celotnega sistema. Težavno merjenje napredka pri razvoju. Vzdrževanje pomeni popravke, izboljšave in dodatne funkcionalnosti.Programska oprema se z uporabo ne iztroši. V splošnem pa njena kvaliteta z vzdrževanjem in s spreminjanjem upada.

1.5 Stopnja okvar



Programska oprema tipično vsebuje napake. Programsko inženirstvo nudi tehnike za omejitev oziroma zmanjšanje števila in usodnosti napak. Pa vendar se napakam težko (nikoli?) popolnoma izognemo...

1.6 Uspešnost programskim produktov

Uspešnost programskih produktov je v splošnem manjša od uspešnosti projektov na drugih inženirskih področjih !
Programska rešitev je lahko uspešna če

- je zaključen razvoj
- je uporabna
- je uporabljena

1.7 Razlogi za neuspeh

- Slaba kvaliteta - razviti sistem vsebuje napake
- Nedoseganje ključnih rokov (čas dobave)
- Preseganje omejenih stroškov
- Nezadovoljevanje uporabnikovih potreb
- Nezmožnost ali težavnost vzdrževanja

- Cena razvoja in vzdrževanja je bistveno višja od pričakovane (ocenjene)
- Neustrezno planiranje razvoja
- Nedefinirani cilji projekta
- Slabo razumevanje uporabnikovih potreb
- Nezadostno preverjanje in kontrola kvalitete
- Nezadostno tehnično znanje članov razvojnega tima
- Slabo razumevanje potrebnih razvojnih vložkov in cene, na strani razvijalca in uporabnika.

1.8 Cena programskih produktov

Največji del cene razvoja programskih produktov predstavlja cena dela ljudi. Spremembe v času izvedbe projekta le tega močno podražijo.

1.9 Lastnosti dobrih programskih produktov

- **Funkcionalnost** - ki zadosti potrebe uporabnika
- **Zmožnost vzdrževanja** – možnost razvoja glede na spreminjajoče potrebe.
- **Zanesljivost** – programska oprema mora biti vredna zaupanja.
- **Učinkovitost** – uporabe sistemskih zmogljivosti.
- **Sprejemljivost** – sprejeta mora biti s strani uporabnikov za katere je bila načrtovana – biti mora razumljiva, uporabna in združljiva z drugimi sistemi.

1.10 Inženirski pristop

- Ocenjevanje stroškov/vložkov
- Časovno načrtovanje dela
- Vključitev uporabnikov ob definiranju zahtev
- Določitev faz v razvoju
- Določitev končnih produktov
- Spremljanje napredovanja projekta
- Načrtovanje pregledov in kontrole kvalitete
- Obsežno testiranje

2 Programski proces

Programski proces je proces gradnje programskega produkta od zaznane potrebe po programskem produktu, do konca vzdrževanja. V programskem procesu se soočamo s tehničnimi in organizacijskimi izzivi. Rezultat programskega procesa je programski produkt. Različni programski produkti zahtevajo različne programske procese.

2.1 Faze programskega procesa

Definicija problema	Definicija
Študija izvedljivosti Analiza in definiranje zahtev	Analiza
Načrtovanje sistema Načrtovanje programa	Načrtovanje
Izvedba Testiranje Predaja produkta	Izvedba
Obratovanje in vzdrževanje	Obratovanje

2.1.1 Definicija problema

- Odgovor na vprašanja:
  - **Kaj je problem?**
  - Kje se problem pojavi in kdo ga občuti?
  - Zakaj se problem sploh pojavi?
  - Kako se problem lahko natančneje razišče?
  - Uporabniki in management se morajo strinjati o problemu.
  - Izogibati se je potrebno dvoumnim definicijam (možnosti različnih interpretacij).
- Ta korak je kratek in tipično traja dan ali dva.
- Če problem obstaja, če ga je potrebno rešiti in če so za njegovo reševanje na voljo sredstva, potem to lahko postane projekt.
  - Potrebna je ocena stroškov za naslednji korak (študijo izvedljivosti).
  - Potrebna je groba ocena celotnih stroškov projekta, s čimer uporabnik dobi občutek o obsežnosti projekta. Boljša ocena se pripravi v kasnejših korakih.
- Možna so različna izhodišča, na primer:
- Nezadostna prepustnost obstoječega sistema, slaba odzivnost.
  - Obstoječa rešitev ni ekonomična (stroški obratovanja)
  - Nezadostna točnost obstoječe rešitve.

- Nezanesljivost obstoječe rešitve.
  - Nezadostna varnost obstoječe rešitve.
  - Obstoječa rešitev ne omogoča pridobivanja ključnih informacij.
  - Potreba po novi funkcionalnosti.
- Kaj je problem in kaj rešitev!?
  - "Računalniški sistem za..." je rešitev in ne problem!
- Rezultat definicije problema je dokument - opis problema (1-2 strani):
  - naziv projekta,
  - jedrnat opis problema,
  - cilji projekta,
  - morebitne začetne ideje o možnih rešitvah,
  - potreben čas in stroški za naslednji korak (študijo izvedljivosti),
  - groba ocena celotnih stroškov projekta.

2.1.2 Študija izvedljivosti

Odgovori na vprašanja:

- **Ali je projekt tehnično izvedljiv?**
- **Ali je projekt ekonomsko upravičen?**
- **Ali je projekt operativno izvedljiv (zakoni, kultura, dogovori)?**
- Kakšne prednosti prinaša projekt?
- Kakšen je obseg predlaganega projekta?

Predlaganih je lahko več alternativ izvedbe. Med njimi se odločamo glede na definirane (in dokumentirane) kriterije. Študija izvedljivosti vodi v odločitev za začetek projekta.

Tehnična izvedljivost:

- Za vsako alternativo izvedbe je potrebno okvirno proučiti tehnične načine izvedbe vseh nadaljnjih faz projekta (analiza zahtev, načrtovanje, izvedba, obratovanje) in podati tehnično oceno tveganja.
- Vključiti je potrebno tehnično osebje (ki kasneje lahko postane del projektnega tima).

Ekonomska upravičenost

- Cena (costs):
  - Enkratni stroški opreme (HW+SW), izobraževanj, programske opreme, konzultacije...
  - Variabilni stroški plač, sredstev, vzdrževanja, najemnin...
- Prednosti (benefits):
  - Prihranki (prihranki plač, materiala, povečanje proizvodnje, zmanjšanje stroškov obratovanja...)
  - Doprinosi (nudenje izboljšanih storitev strankam, izboljšan nadzor poslovanja (proizvodnja, inventar, finance...) zmanjšanje okvar,

prisotnosti napak...

Kako oceniti stroške projekta?

- Z razdružitvijo sistema v več komponent. Posamezne komponente je lažje ocenjevati kot celoten sistem.
- Z uporabo zgodovinskih podatkov (npr. predhodni projekti)
- Z ocenitvijo stroškov osebja (za razvoj in obratovanje) stroški so časovno odvisni, zato je potreben predhoden časovni načrt.
- Z uporabo standardov na nivoju organizacije, za ocenitev dodatnih stroškov (vodenje, tajništvo, prostori, elektrika...)

Rezultat študije izvedljivosti je poročilo, ki se ga predstavi vodstvu in uporabnikom. Ti ocenijo:

- Ali so alternative sprejemljive?
- Ali rešujejo pravi problem?
- Ali katera od rešitev ponuja bistvene prednosti in zagotavlja uspešnost projekta? Upoštevati je potrebno življenjsko dobo sistema, ki je običajno 5-7 let.

Uporabniki in vodstvo izberejo eno od alternativ. To je osnova za začetek projekta. Veliko število (nesojenih) projektov tu "umre".

2.1.3 Analiza in definiranje zahtev

Namen analize in definiranja zahtev je v celoti opisati obnašanje sistema:

- Funkcijske zahteve: primeri uporabe (ang. use cases), za vse interakcije uporabnika s sistemom.
- Ostale (nefunkcijske) zahteve, ki omejujejo načrtovanje ali izvedbo sistema
  - Obratovalne lastnosti (varnost, uporabljivost, zahtevana prepustnost sistema...)
  - Evolucijske lastnosti (možnost testiranja, možnost vzdrževanja, razširljivost, nadgradljivost...)

Analizo in definiranje zahtev izvaja sistemski analitik. Napačna, nepopolna nekonsistentna ali dvoumna specifikacija zahtev so pogosto vzrok za neuspeh projektov in spore.

Izzivi:

- Uporabniki pogosto ne poznajo točno dejanskih potreb in ne vejo kako lahko računalniški sistem izboljša njihove procese.
- Uporabniki s časom spreminjajo svoje želje.
- Uporabniki imajo lahko neskladne zahteve.

- Uporabniki težko ločujejo med tem kaj je (cenovno) izvedljivo in kaj ne.
- Uporabniki pogosto ne ločujejo med željami in zahtevami
- Sistemski analitik ima običajno omejeno znanje iz področja uporabe sistema.
- Pogosto uporabnik in stranka (naročnik) nista ista oseba.

Proces analize in definiranja zahtev:
- Pridobivanje znanja iz področja uporabe.
- Seznanjanje s problemom (intervjuji, vprašalniki, iskanje soodvisnosti, identifikacija vrzeli).
- Študija obstoječih sistemov.
- Pogosto analiza poteka od zunaj navznoter – od potrebnih izhodnih podatkov do vhodnih podatkov, potrebnih procesov, potreb po hranjenju podatkov.
- Izdelava specifikacije (opisi sistema z naravnim jezikom, diagrami...)
- Recenzija rezultatov analize skupaj z uporabnikom.
- Po potrebi nova iteracija analize in definiranja zahtev.

Rezultat analize in definiranja zahtev je dokument **Specifikacija zahtev**, ang. Software requirements specifications (SRS), (tudi funkcijske specifikacije -FS) Uporabniku nudi informacije na osnovi katerih lahko presodi o pravilnosti razumevanja problema s strani organizacije, ki bo za njih razvijala sistem. Problem razgradi na manjše komponente. Uporabi se za načrtovanje sistema. Uporabi se pri validaciji sistema.
Vsebina specifikacije zahtev se mora prilagajati potrebam projekta, tipično pa vsebuje:
- Uvod (namen, obseg, definicije, reference na druge dokumente)
- Splošni opis (relacije z drugimi sistemi, pregled funkcij/komponent, značilnosti uporabnikov, splošne omejitve)
- Opis zahtev vsake posamezne komponente (opis, vhodi, obdelava, izhodi)
- Zahteve zunanjih vmesnikov (formati, strojna oprema, povezave z drugimi programi ali sistemi)
- Zahteve po prepustnosti sistema (performance)
- Omejitve pri načrtovanju (standardi, omejitve strojne opreme...)
- Ostale zahteve

Faza analize in definiranja zahtev se konča z recenzijo specifikacije zahtev
- Pregled vseh zahtev:
  - Veljavnost. Ali sistem nudi funkcije, ki najbolje zadostijo potrebam uporabnika?
  - Konsistentnost. Ali so kakšne zahteve med seboj v konfliktu?

- ∘ Popolnost. Ali so upoštevane vse funkcije, ki jih zahteva uporabnik?
  - ∘ Realnost. Ali so zahteve lahko realizirane s predvideno tehnologijo in v okviru omejitev stroškov?
- Preverjanje posameznih zahtev:
  - ∘ Preverljivost. Ali je zahtevo realno mogoče preveriti?
  - ∘ Razumljivost. Ali je zahteva pravilno razumljena?
  - ∘ Sledljivost. Ali je izvor zahteve jasno naveden?
  - ∘ Prilagodljivost. Ali je zahtevo mogoče spremeniti brez velikega vpliva na druge zahteve?

2.1.4 Načrtovanje sistema

- V fazi načrtovanja sistema določimo arhitekturo programskega sistema:
  - ∘ strukture programskih komponent,
  - ∘ zunanjih lastnosti komponent in
  - ∘ relacij med komponentami.
- Razdelitev zahtev celotnega sistema med posamezne komponente (določitev funkcij).
- Delitev sistema na komponente (dekompozicija)
  - ∘ Razdeli sistem tako, da se doseže čim večja neodvisnost komponent (bistveno za vzdrževanje).
  - ∘ Dekompozicija poteka vertikalno in horizontalno:
    - Particije vertikalno delijo sistem v več neodvisnih (ali šibko povezanih) podsistemov, ki nudijo storitve na istem nivoju abstrakcije.
    - Nivoji (layer) so podsistemi, ki uporabljajo storitve nižjega nivoja in nudijo storitve višjemu nivoju.
  - ∘ Komponente sistema so procesne in podatkovne.

2.1.5 Načrtovanje programa

- Glede na izbran načrt izvedbe sistema se načrtuje posamezne programske komponente.
  - ∘ Celotna struktura programa
  - ∘ Moduli, ki jih je potrebno izvesti (kodirati)
  - ∘ Načrt podatkovne baze in datotečnih formatov.
- V tem koraku se vsaka komponenta definira tako podrobno, da na osnovi tega lahko sledi izvedba programa.
- Izdelki načrtovanja programa so:
  - ∘ Specifikacije programa (npr. diagrami, psevdokoda, opisi)
  - ∘ Načrt datotek (organizacija, način dostopa...)
  - ∘ Specifikacija strojne opreme

- Načrt testiranja
- Časovni načrt izvedbe
- Konča se s tehničnim pregledom.

2.1.6 Izvedba

- V fazi izvedbe na osnovi načrta programa zgradimo programski produkt:
  - Množico programov ali programskih enot, ki so lahko
    - napisane v ta namen
    - pridobljene od drugod
    - spremenjene obstoječe programske enote
  - Podatkovne strukture, ki programom omogočajo manipulacijo (obdelavo) podatkov (informacij).
  - Dokumentacijo, ki opisuje delovanje (in uporabo) programov.
- Izvedba se konča z verifikacijo:
  - Posamezne komponente se testira glede na skladnost s specifikacijami programa.
  - Posamezne komponente se združi v programski sistem (faza integracije) in testira glede na specifikacije sistema.

2.1.7 Testiranje

- V fazi testiranja se izvaja validacija, s katero se preveri ali programski produkt ustreza potrebam uporabnikov.
  - Poteka po vnaprej pripravljenem načrtu, ki izhaja iz specifikacije zahtev.
  - Po odpravi (morebitnih) ugotovljenih nepravilnosti se testiranje ponovi.
- V času testiranja se izdela/dodela navodila za uporabo sistema.

2.1.8 Predaja produkta

- Prevzem s strani uporabnika
  - Skupaj z uporabnikom se preveri delovanje celotnega sistema. Uporabnik ob tem potrdi ali zavrne ustreznost sistema.
  - Testiranje se lahko izvrši pri razvijalcu (FAT) ali pri uporabniku (SAT).
- Predaja produkta naročniku
  - Celotni sistem je dostavljen naročniku in predan v obratovanje.
  - Uporabnikom so na voljo navodila, organizirana je pomoč uporabnikom...

2.1.9 Obratovanje in vzdrževanje

- Sistem mora izpolnjevati uporabnikove potrebe – pravilno in neprestano.
- Vzdrževanje pomeni spreminjanje programskega produkta:
  - Korektivne spremembe (odpravljanje napak).
  - Adaptivne spremembe (prilagajanje spremenjenim pogojem, npr. razširitev funkcionalnosti, spremembe vmesnikov...)
  - Perfektivne spremembe (izboljšanje programskega produkta, npr. prenos na nove platforme, dodatne funkcionalnosti...)
  - Preventivne spremembe (odpravljanje poslabševanja programskega produkta)
- Življenjski cikel programskih produktov se koncča s prenehanjem uporabe.

2.2 Razvojni modeli programskega procesa

Razvojni model programskega procesa je načrt izvajanja faz programskega procesa.
Vsak programski proces vsebuje vse faze programskega procesa, ki pa so lahko izvedene **formalno** ali **neformalno** in v različnem zaporedju.

2.2.1 Prednosti formalnih procesov

Formalno izvajanje faz programskega procesa prinaša vrsto prednosti:
- Boljši nadzor nad sredstvi (resoursi).
- Boljši odnos s stranko.
- Krajši čas razvoja.
- Nižji stroški.
- Višja kvaliteta in zanesljivost.
- Večja produktivnost.
- Boljša organizacija in koordinacija dela.
- Manj stresno delo.

Prednosti so bolj izražene pri večjih projektih.

2.2.2 Prednosti neformalnih procesov

Pri manjših projektih so prednosti manj izražene in pogosto pride ugibanj o potrebnosti formalnega izvajanja faz:
- Manj potreb po dokumentiranju?
- Manj potrebnih sredstev za organizacijo in koordinacijo?
- Večja svoboda in motiviranost razvijalcev?

Neformalno izvajanje določenih faz je lahko vzrok za težave v kasnejših fazah!

2.3 Razvojni modeli programskih procesov

2.3.1 Build and fix

Težave
- Ni specifikacij
- Ni načrta izvedbe
- Ni določenih faz
- Ni mejnikov

Neprimeren pristop
- Nemogoč nadzor nad tekom projekta
- Težavno zagotavljanje kvalitete
- Visoka obremenjenost stranke, padec zaupanja vprodukt...

2.3.2 Linearni model

Linearni (klasični, kaskadni) model, ang. waterfall model:
- Faze si sledijo ena za drugo in se ne prekrivajo. Naslednja faza se začne po zaključeni predhodni fazi.
- Vsaka faza se zaključi s pregledom oziroma testiranjem.
- Možen le v primeru dobro določenih zahtev/ciljev in jasnega načina izvedbe, saj je upoštevanje kasnejših sprememb težavno.
- Zahteva potrpljenje stranke – delujoča verzija ji je na voljo šele ob predaji končnega produkta

Prednosti:
- preglednost,
- ločenost faz,
- stalen nadzor nad kvaliteto,
- dober nadzor nad stroški.

V praksi šele vsak naslednji korak omogoča širše/boljše razumevanje prejšnjih faz, zato so pogosto nujne revizije predhodnih korakov – Linearni model tega ne omogoča.

2.3.3 Linearni model s povratki

Omogoča revizije predhodnih korakov. Možna je izvedba z delnim prekrivanjem faz.

2.3.4 "V" model

Razširitev linearnega modela.
Vsaka faza razvoja je povezana s pripadajočo fazo testiranja.

2.3.5 Spiralni model

Vsak cikel vsebuje faze linearnega modela.
V vsakem ciklu se izdela prototip.
Vsak cikel sestoji iz štirih kvadrantov:

- Določitev ciljev, alternativ in omejitev.
  - **Cilji**: funkcionalnost, prepustnost/zmogljivost, določitev vmesnikov, faktorji za uspeh projekta...
  - **Alternative**: razvoj, uporaba že razvitih komponent, nakup komponent/produkta, podizvajalci...
  - **Omejitve**: stroški, časovne omejitve, vmesniki...
- Ocenjevanje alternativ, identifikacija tveganj.
  - Presoja alternativ glede na cilje in omejitve
  - Identifikacija tveganj (pomanjkanje izkušenj, nove tehnologije, kratki časovni roki...
  - Ocenitev tveganj (možnost neuspeha, smiselnost nadaljnjega razvoja...)
- Razvoj produkta trenutnega cikla.
  - Cikli linearnega modela: načrtovanje, izvedba,testiranje.
  - Kontrola kvalitete (pregledi, verifikacije, validacija)
- Načrtovanje naslednjega cikla.
  - Projektno planiranje naslednjega cikla
  - Upravljanje konfiguracije (identifikacija, organiziranje in kontrola sprememb razvoja programske opreme v projektni skupini)
  - Načrtovanje testiranj
  - Načrtovanje namestitev sistema

Prednosti:

- Omogoča zgodnjo identifikacijo nepremostljivih ovir – z majhnimi stroški.
- Uporabniki zgodaj dobijo vpogled v sistem (prototipi)
- Najprej se razvijejo najbolj kritične funkcije sistema
- Nepopolno načrtovanje je sprejemljivo
- Uporabniki imajo vpogled v razvoj sistema
- Zgodnja povratna informacija od uporabnikov

Slabosti:

- Izguba časa zaradi večkratnega ponovnega načrtovanja, predefiniranja zahtev, analize tveganj in izgradnje prototipov.
- Model je kompleksen in ne določa konca projekta (število ciklov).
- Razvijalci v nekaterih kvadrantih niso izkoriščeni.

Kdaj ga uporabiti?
- Ko je upravičena izgradnja prototipov
- Ko je pomembno ocenjevanje tveganj – za bolj tvegane projekte.
- Ko je dopustno daljše časovno obdobje razvoja.
- V primeru kompleksnih zahtev.
- Ko so pričakovane večje spremembe zahtev.

2.3.6 Razvoj z uporabo prototipov

Razvoj z uporabo prototipov je način hitrega razvoja sistema (ang. rapid development).
Prvotni namen prototipa je pomoč stranki in razvijalcem pri razumevanju zahtev sistema.
- Uporabniki lahko preverijo kako se odražajo njihove zahteve.
- Prototip razkrije napake in manjkajoče zahteve.

Razvoj prototipov lahko razumemo kot aktivnost zmanjševanja rizika pri specificiranju zahtev.
Prototip je na voljo zgodaj v programskem procesu in lahko služi tudi za usposabljanje uporabnikov in testiranje sistemov.

V preteklosti je prototip veljal za manjvrednega zato je izgradnji prototipa vedno sledil nadaljnji razvoj. Danes poznamo v programskih procesih dve možnosti uporabe prototipov:
- Evolucijski proces
  - Začetni prototip je postopno izboljšan in tako preide v končni sistem.
  - Namen prototipa je zgraditi končni sistem.
  - Razvoj se začne pri zahtevah, ki so najbolj jasne.
- Razvoj z opustitvijo prototipa
  - Prototip je namenjen izključno odkrivanju zahtev in se po specificiranju zahtev opusti. Končni sistem se razvije neodvisno.
  - Razvoj se začne pri zahtevah, ki so najbolj nejasne.

2.3.7 Evolucijski razvoj

Prednosti:

- Hitra dobava. Včasih je bolj pomembna od funkcionalnosti ali od možnosti dolgoročnega vzdrževanja.
- Vključenost uporabnika povečuje verjetnost, da sistem ustreza uporabnikovim potrebam, hkrati pa povečuje verjetnost uporabe sistema.

Slabosti:
- Obstoječi procesi managementa tipično predpostavljajo linearni model.
- Razvoj prototipa (brez specifikacij) zahteva dodatne sposobnosti razvijalcev.
- Težavnost vzdrževanja. Neprestano spreminjanje negativno vpliva na strukturo sistema, zato je dolgoročno vzdrževanje težavno.
- Pogodbene težave. Uporabniki morajo prevzeti dodatne obveznosti, večkratna izdaja produkta.

2.3.8 Razvoj z opustitvijo prototipa

Namen prototipa je zmanjšati tveganje pri določitvi zahtev.
Prototip je razvit na osnovi začetnih specifikacij zahtev.
Namenjen je eksperimentiranju (ugotavljanju lastnosti različnih izvedb, tehnologij...) in se ga za tem zavrže.
Prototip NI namenjen kakršni koli uporabi v končnem sistemu:
- Prototip ni narejen z upoštevanjem vseh potrebnih lastnosti sistema.
- Prototip ni dokumentiran in bi ga bilo zato težko dolgoročno vzdrževati.
- Arhitektura prototipa je napravljena "ad-hoc" zato je praviloma neprimerna za dolgotrajno vzdrževanje.

Razvijalci so pogosto pod pritiskom, da prototip predajo kot končni sistem.
Zakaj to ni primerno?
- Prototipi običajno ne ustrezajo nefunkcijskim zahtevam in je le te kasneje težko ali celo nemogoče zagotoviti.
- Prototip je nedokumentiran.
- Arhitektura/struktura prototipa ni namenjena kasnejšemu dograjevanju in z nadaljnjim razvojem/vzdrževanjem postane neustrezna.
- Pri razvoju prototipa se pogosto/večinoma ne upošteva običajnih standardov kakovosti!

2.3.9 Postopni (fazni) razvoj

- Sistem je razvit in dostavljen uporabniku v več fazah.
- Osnova za razvoj (prve faze) je arhitekturni načrt celotnega sistema!

- Faze, ki so že predane uporabniku, že zagotavljajo neko zaokroženo (ćeprav nepopolno) funkcionalnost.
- Uporabniki lahko preverjajo ali celo uporabljajo rešitve prvih faz, med tem ko so naslednje faze še v razvoju.
- Na ta način fazni pristop nudi prednosti razvoja s prototipi in hkrati zagotavlja boljšo preglednost projekta, boljše vodenje projekta in v končni fazi boljšo strukturo produkta.

2.3.10 Unified Process (UP)

Unified Process (UP) (enoten process) je ogrodje, ki ga je mogoče prilagoditi specifičnim organizacijam oziroma projektom. IBM je privzel UP in ga trži pod imenom Rational Unified Process. Ogrodje določa faze razvoja produkta in inženirske discipline potrebne za realizacijo teh faz. Vsaka faza je zgrajena iz korakov, ki so osnovni gradniki procesa.

Projekt sestoji iz štirih faz:
- **začetek (ang. inception)**: definicija problema, študija izvedljivosti, zaključi se z mejnikom ciljev (ang. Lifecycle Objective Milestone)
- **dopolnitev (ang. elaboration)**: analiza in definiranje zahtev, načrtovanje sistema, načrtovanje programa, zaključi se z arhitekturnim mejnikom (ang. Lifecycle Architecture Milestone)
- **gradnja (ang. construction)**: načrtovanje programa, izvedba, testiranje, konča se z mejnikom uporabljivosti (ang. Initial Operational Capabilit Milestone)
- **prehod (ang. transition)**: validacija, prevzem in izdaja, šolanje uporabnikov, konča se z mejnikom izdaje produkta (ang. Product Release Milestone)

Unified Process definira šest inženirskih disciplin:
- poslovno modeliranje (ang. bussiness modeling)
- definiranje zahtev (ang. requirements discipline)
- analiza in načrtovanje (ang. analysis and design)
- izvedba (ang. implementation)
- testiranje (ang. test discipline)
- izdaja (ang. deployment)

Proces je zgrajen iz osnovnih elementov, ki določajo kaj naj bo izdelano, kakšna so potrebna znanja in kako doseči cilj:
- **Vloge (kdo)**: znanja, kompetence, odgovornosti.

- **Produkt (kaj)**: rezultat naloge - dokumenti, modeli, podatki, porogrami.
- **Naloge (kako)**: opis postopka za dosego cilja.

Dobra praksa, ki jo določa UP:
- Razvijaj iterativno
- Upravljaj zahteve
- Uporabljaj komponente
- Modeliraj vizualno
- Preverjaj kvaliteto
- Obvladuj spremembe

2.4 Ogrodja programskih procesov

Organizacija lahko zgradi svoje ogrodje programskih procesov ter z njim definira svoje programske procese.

2.5 Zrelostni nivoji programskih procesov

Mera efektivnosti programskih procesov v podjetju:
1. **Začetna** - uspešnost odvisna od posameznikov.
2. **Ponovljiva** - uvedeno osnovno vodenje projektov za sledenje stroškov, rokov in funkcionalnosti. Proces je mogoče ponoviti na podobnih projektih.
3. **Določena** – programski procesi so dokumentirani, standardizirani in potrjeni znotraj podjetja. Na teh procesih temeljijo vsi projekti podjetja.
4. **Vodena** – pri vseh projektih se meri vrsto indikatorjev, ki govorijo o uspešnosti projekta in s tem programskega procesa.
5. **Optimirana** – stalno izboljševanje programskih procesov na podlagi kazalcev uspešnosti, novih idej in tehnologij.

2.6 Cena programskega inženirstva

Okvirno 60% cene predstavlja razvoj in 40% testiranje. Za programsko opremo po meri je značilno, da stroški prilagajanja po prvi predaji stranki pogosto presegajo stroške razvoja. Stroški so odvisni od tipa razvijalnega sistema in zahtev kot sta prepustnost sistema in zanesljivost. Porazdelitev stroškov je odvisna od uporabljenega modela programskega procesa.

3 Diagramske tehnike

3.1 Zakaj diagrami?

- Omogočajo pregleden opis modelov.
- So glavno orodje pri modeliranju:
  - Omogočajo opis kompleksnih sistemov.
  - Nedvoumen opis prispeva k lažjemu in enotnemu razumevanju problema in rešitve.
  - Olajšajo načrtovanje željene strukture in obnašanja sistemov.

3.2 Diagrami in modeli

Diagram je **delna** grafična predstavitev modela sistema.
Model lahko vključuje več diagramov ter dokumentacijo. Primer:

- model primera uporabe (ang. use-case) sestoji iz diagrama (use-case diagram) ter dokumenta, ki opisuje primer uporabe.

V tej povzetku se omejujemo na diagramske tehnike in ne predstavljamo celotnih modelov.

3.2.1 Vrste

- **Kontekstualni (ang. context)**
  - Pogled sistema od zunaj. Opredelitev sistema v okolju.
- **Vedenjski (ang. behavioural)**
  - Obnašanje sistema: primeri uporabe, delovanje procesov, tok podatkov, prehodi med stanji...
- **Strukturni (ang. structural)**
  - Struktura sistema, komponent/objektov, podatkov...

3.2.2 Nivoji modelov

- Konceptualni model
  - Predstavitev osnovnih lastnosti sistema, visokonivojski, osnovni/grobi pogled na sistem.
- Logični model
  - Konceptualni model dopolnjuje s podrobnostmi, omejuje se na delovanje oziroma uporabo (procesi, podatki...)
- Fizični model
  - Logičnemu modelu doda podrobnosti o dejanski fizični izvedbi (tipi, spremenljivke, funkcije...)

3.3 Diagrami pri načrtovanju programskih produktov

3.3.1 Diagram poteka (ang. flowchart)

Diagram sestoji iz tipičnih simbolov za začetek in konec, kontrolni tok, procesne korake, vhodne in izhodne operacije, odločitvene funkcije...

3.3.2 Diagram podatkovnega toka (ang. Data Flow Diagram)

Modeliranje toka podatkov (NE kontrolnega toka).
Logični pogled na sistem (NE fizični).
Primerno za komunikacijo z uporabniki, vodstvom in drugimi poznavalci IS.
Primeren za analizo obstoječih in predlaganih sistemov. Relativno enostavna tehnika.

3.3.3 Entitetno relacijski diagram - ER

Namenjen je ilustraciji strukture podatkov.
Definicije:

- **Entiteta** (ang. entity) – objekt iz realnega sveta, ki ga lahko ločimo od ostalih objektov. Entiteta sestoji iz množice atributov.
- **Entitetna množica** (ang entity set) – zbirka podobnih entitet, z istimi atributi.
- **Atribut** (ang attribute) – lastnost entitete.
- **Relacija**, tudi razmerje (ang. relationship) - zveza med dvema ali večimi entitetami
- **Ključ** (ang. key) – atribut, ki omogoča razločevanje ene entitete od druge.

3.3.4 UML diagrami

UML (The Unified Modeling Language) se uporablja za specifikacijo, vizualizacijo, modifikacijo, konstruiranje in dokumentiranje procesov in objektov razvijajočega sistema. Je diagramska tehnika za zapis različnih vrst diagramov v programskem inženirstvu. Poudarek je na objektno orientiranem pristopu. Gre za standardiziran način predstavitev arhitekture sistema. 20% poznavanje UML zadošča za 80% primerov modeliranja sistemov.

Strukturni UML diagrami (structure)

- **Class diagram (razredni diagram)** opisuje strukturo sistema s prikazom razredov (class), atributov in relacij med razredi.

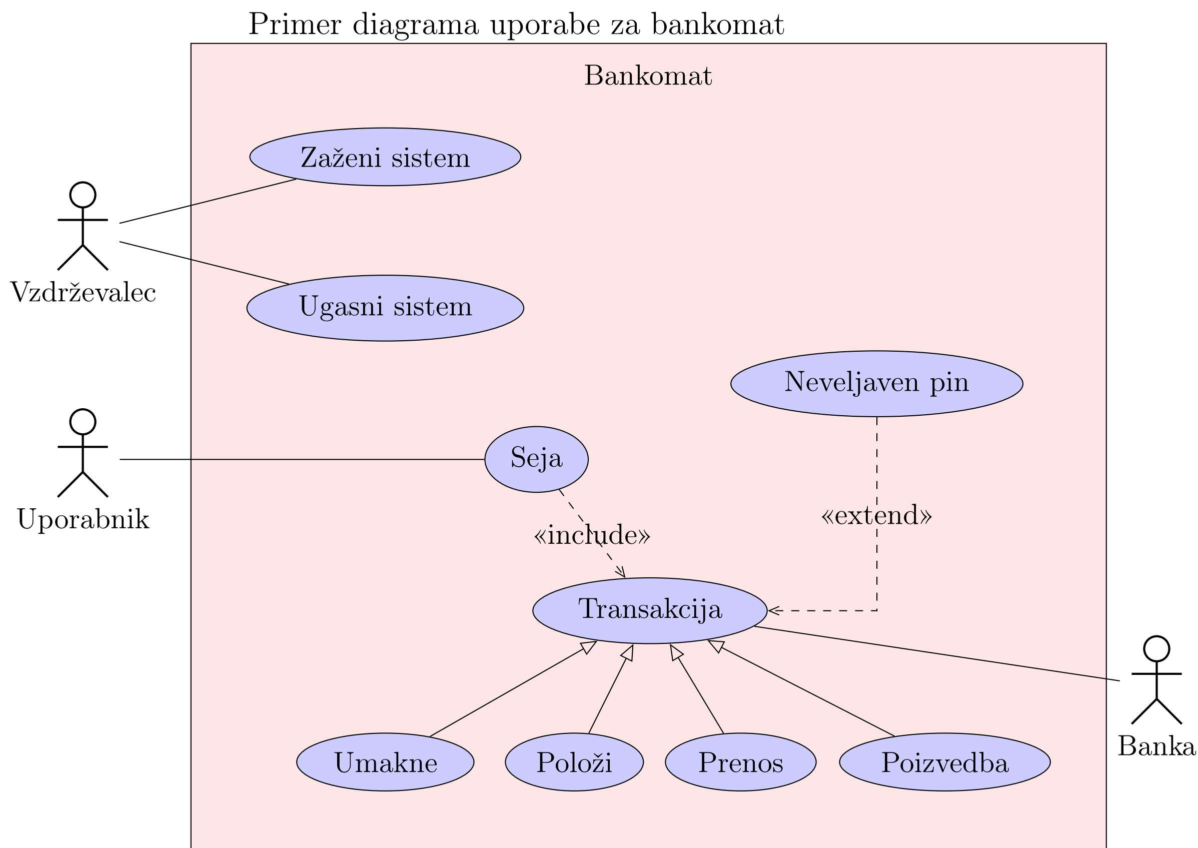
- Component diagram (komponentni diagram) prikazuje kako je programski sistem razdeljen na komponente in kakšne so njihove medsebojne odvisnosti.
- Composite structure diagram prikazuje notranjo strukturo razreda in sodelovanja, ki jih ta struktura omogoča.
- Deployment diagram (diagram postavitve) prikazuje strojno opremo in okolje, v katerem deluje programska oprema, vključno s povezavami med komponentami.
- Object diagram (objektni diagram) prikazuje celotno ali delno strukturo objektov sistema v specifičnem trenutku.
- Package diagram (paketni diagram) opisuje razdelitev sistema na logične enote in relacije med njimi.

Vedenjski UML diagrami (behavior)

- **Activity diagram (diagram aktivnosti)** je diagram poteka izvajanja korakov sistema. Prikazuje kontrolni tok pri delovanju sistema.
- **State machine diagram (diagram stanj)** prikazuje stanja in možno prehanje med stanji kot posledico dogodkov v sistemu.
- **Use case diagram (diagram primera uporabe)** prikazuje funkcionalnost sistema v smislu akterjev in njihovih ciljev pri uporabi sistema, hkrati pa tudi povezanost primerov uporabe.
- Communication diagram (komunikacijski diagram) prikazuje interakcije med objekti ali deli sistema, ki so prikazani v obliki strukture sistema.
- Interaction overview diagram je tip diagrama aktivnosti, kjer elementi predstavljajo diagrame interakcij.
- **Sequence diagram (sekvenčni diagram)** prikazuje komunikacijo med objekti v smislu sekvence sporočil.
- Timing diagrams (časovni diagram) je poseben tip diagrama interakcij, kjer so prikazane časovne omejitve pri delovanju sistema.

Use case diagram

Primer diagrama uporabe za bankomat



Vrste relacij med primeri uporabe:

- Asociacija
  Komunikacijska povezava med akterjem in primerom uporabe, v kateri sodeluje.
- Razširja (extend)
  Primer uporabe na začetku puščice (lahko) razširi obnašanje primera uporabe na koncu puščice.
- Vsebuje (include)
  Primer uporabe na začetku puščice vsebuje obnašanje primera uporabe na koncu puščice.
- Generalizacija
  Relacija med splošnim in bolj specifičnim primerom uporabe, ki podeduje značilnosti splošnega primera uporabe in mu doda nove.

4 Analiza zahtev

4.1 Problemi analize

- Implicitni modeli niso ubesedeni.
- Implicitni modeli se s časom spreminjajo.
- Naročnik in sistemski analitik ne govorita istega "jezika".
- Vsega implicitnega znanja se ne da fomralno zapisati.

4.2 Specifikacija zahtev

Namen dokumeta "specifikacije zahtev" (ang. Software Requirements Rpecification - SRS):

- seznaniti se s potrebami in omejitvami uporabnika,
- uskladiti razumevanje problema in rešitve z uporabnikom,
- določiti izhodišča za načrtovanje sistema,
- določiti izhodišča za validacijo sistema.

Vsebina dokumenta specifikacij zahtev (SRS):

1. Uvod (namen, obseg, definicije, reference na druge dokumente)
2. Splošni opis (relacije z drugimi sistemi, pregled funkcij/komponent, značilnosti uporabnikov, splošne omejitve)
3. Opis zahtev vsake posamezne komponente (opis, vhodi, obdelava, izhodi)
4. Zahteve zunanjih vmesnikov (formati, strojna oprema, povezave z drugimi programi ali sistemi)
5. Zahteve po prepustnosti sistema (performance)
6. Omejitve pri načrtovanju (standardi, omejitve strojne opreme...)
7. Ostale zahteve

4.2.1 Postopek

1. Seznanjanje s problemom in zbiranje zahtev
2. Analiza, modeliranje in usklajevanje
3. Recenzija (validacija zahtev)
4. Upravljanje z zahtevami

4.3 Seznanjanje zahtev

4.3.1 Problem obsega sistema

- Meje sisteme morda niso jasne?

- Kdo bo neposredno uporabljal sistem?
  - Kateri drugi sistemi so povezani s sistemom?
  - Katera funkcionalnost naj bo del sistema?
- Stranka lahko navaja nepotrebne tehnične detajle, ki lahko zmedejo razumevanje obsega sistema (npr. brez razloga navajan operacijski sistem, jezik, strojna oprema)

4.3.2 Problemi razumevanja zahtev

- Stranke/uporabniki morda niso prepričani kaj točno potrebujejo:
  - "Naredite tako, da nam bo v pomoč"
  - "Poskusite izboljšati sistem"
- Morda slabo poznajo zmogljivost in omejitve svoje računalniške opreme ali procesa.
  - Nimajo popolnega razumevanja problemskega področja.
  - Ne znajo opredeliti zahtev sistemskemu analitiku.
  - Izpustijo informacije, ki naj bi bile "očitne".
  - Navajajo zahteve, ki nasprotujejo zahtevam drugih.
  - Navajajo nejasne zahteve in zahteve, ki jih ni mogoče preveriti. Sistemski analitik pogosto ni poznavalec problemskega področja

4.3.3 Problemi spremenljivosti zahtev

- Zahteve se s časom spreminjajo
- Spremembe so neizogibne zaradi naslednjih dejavnikov:
  - Spremembe v strankini organizaciji (npr. novi produkti, oddelki...)
  - Spremembe obsega delovanja (npr. dnevno število transakcij, število uporabnikov, povečana propustnost povezav...).
  - Zunanje spremembe
    - Spremembe zakonov (npr. obdavčevanje)
    - Spremembe mednarodnih standardov (npr. MPEG)
  - Stranke dobijo nove ideje ko spoznajo zmožnosti novega sistema

4.4 Zbiranje zahtev

4.4.1 Študij literature

- Knjige s problemskega področja
- Opisi procesov
- Študije izboljšav procesov
- Razvojna dokumentacija
- Evidenca težav in prijavljenih napak

- Predlogi

4.4.2 Nasveti strokovnjakov

- Prisluhni stokovnjakom z danega področja.
- Strokovnjaki naj ne bodo nujno le tisti, ki so povezani z naročnikom.
- Strokovnjaki
    - Poznajo zakonitosti problemskega področja.
    - Pomagajo določiti obseg sistema.
    - Lahko usmerijo sistemskega analitika k študiji alternativnih rešitev.

4.4.3 Spraševanje (ankete)

- Sistemski analitik sprašuje stranke in/ali uporabnike
    - Kaj pričakujejo od novega sistema.
    - Upamo, da znajo obiti svoje omejitve in predsodke.
    - Spraševanje lahko poteka v obliki intervjuja, brainstorminga ali vprašalnika.
- Lahko gre za neposredno komunikacijo s stranko ali za anketiranje uporabnikov.

4.4.4 Izpeljave iz obstoječega sistema

Analizo začnemo na osnovi obstoječega sistema.

- Analiza lahko poteka na osnovi sistema, ki ga želimo nadomestiti.
- Analiza lahko poteka na osnovi sistema v neki drugi organizaciji.
- Analiza lahko poteka na osnovi opisa sistema v literaturi.

4.4.5 Sinteze iz lastnosti okolice

- Za uspešno delovanje programske opreme je potrebno upoštevati zakonitosti okolja.
- Gre za t.i. procesno analizo, normativno analizo ali decizijsko analizo.

4.4.6 Izdelava prototipov

- V uporabi predvsem takrat, ko brez prototipa ne znamo jasno določiti zahtev.
- Prototip je delujoč model dela sistema ali celotnega sistema.
- Tipično se model od končnega sistema razlikuje zaradi neizpolnjevanja nefunkcijskih zahtev.

4.5 Atomske zahteve

Zahteve so atomske, če:

- natančno določajo potrebo ne da bi jih bilo treba deliti na več zahtev
- so merljive,
- so preverljive (testabilne),
- so sledljive.

4.6 Analiza, modeliranje in usklajevanje

Koraki:

- Analiza
  - Analiza podanih zahtev v smislu konsistentnosti, popolnosti in realnosti.
- Modeliranje
  - Izgradnja eksplicitnega modela zahtev, v obliki naravnega jezika, diagramov in formalnih notacij (npr. matematično).
- Usklajevanje
  - Dogovarjanje o obsegu rešitve, konfliktnih zahtevah, prioritetah, rizikih...

Običajno se koraki izmenjujejo iterativno.

4.7 Recenzija

Recenzija je pregled in vrednotenje izdelka ali procesa s strani kvalificiranega posameznika ali skupine. Recenzijo specifikacij zahtev opravijo predstavniki nosilcev interesov (strank, managementa...).

4.7.1 Recenzija specifikacije zahtev

Pregled vseh zahtev:

- Veljavnost. Ali sistem nudi funkcije, ki najbolje zadostijo potrebam uporabnika?
- Konsistentnost. Ali so kakšne zahteve med seboj v konfliktu?
- Popolnost. Ali so upoštevane vse funkcije, ki jih zahteva uporabnik?
- Realnost. Ali so zahteve lahko realizirane s predvideno tehnologijo in v okviru omejitve stroškov?

Preverjanje posameznih zahtev:

- Preverljivost. Ali je zahtevo realno mogoče preveriti?

- Razumljivost. Ali je zahteva pravilno razumljena?
- Sledljivost. Ali je izvor zahteve jasno naveden?
- Prilagodljivost. Ali je zahtevo mogoče spremeniti brez velikega vpliva na druge zahteve?

4.7.2 Tipi recenzij

- Kontrola s strani posameznika
  - pogosto s strani avtorja
  - uporaba kontrolnih seznamov
- Skupinski pregled
  - S strani skupine pregledovalcev
  - v skupini ni določenih vlog, postopek ni pred-definiran
  - običajno vsi pregledovalci predhodno pregledajo sami (kontrola posameznika)
- Nadzor
  - formalna recenzija skupine izšolanih pregledovalcev (nadzornikov) z določenimi vlogami
  - uporaba kontrolnih seznamov
  - obvezna predpriprava pregledovalcev

4.8 Upravljanje z zahtevami

Spremembe zahtev:
- so neizogibne
- se dogajajo v celem življenjskem ciklu programskih produktov.

Upravljanje z zahtevami pomeni
- identificiranje,
- sledenje in
- nadzor nad zahtevami in spremembami

5 Načrtovanje sistema

Načrtovanje sistema vključuje:

- definicijo načrtovalskih ciljev,
- dekompozicijo sistema na podsisteme,
- izbiro že izdelanih komponent,
- relacije med programsko in strojno opremo,
- izbira infrastrukture za upravljanje s perzistentnimi podatki,
- določitev politike nadzora dostopa,
- izbira mehanizma globalnega kontrolnega toka,
- obravnavanje mejnih primerov.

5.1 Stabilna arhitektura

Stabilna arhitektura zagotavlja zanesljivost sistema in enostavno vzdrževanje. Stabilna arhitektura je tista, ki omogoča enostavno dodajanje funkcionalnosti z le majhnimi spremembami arhitekture.

5.2 Načrtovalski cilji

Definiranje načrtovalskih ciljev je prvi korak načrtovanja sistema. Mnogi načrtovalski cilji izhajajo iz nefunkcijskih zahtev ali domene aplikacije. Preostale načrtovalske cilje je potrebno uskladiti s stranko.
V splošnem lahko načrtovalske cilje izberemo iz seznama splošno željenih kvalitet programskih produktov, ki jih uredimo v pet skupin:

5.2.1 učinkovitost (ang. performance)

- Odzivni čas (ang. response time)
  Koliko časa preteče od zahteve do odziva?
- Prepustnost (ang. throughput)
  - Koliko nalog lahko sistem izvrši v časovni enoti?
- Pomnilnik (ang. memory)
  - Koliko pomnilnika (na splošno prostora) je potrebnega za delovanje sistema?

5.2.2 zanesljivost (ang. dependability)

- Robustnost (ang. robustness)
  - Zmožnost preživetja ob neveljavnem uporabniškem zahtevku (vhodni tok).
- Zanesljivost (ang. reliability)

  - Razlika med specificiranim on opaženim obnašanjem.
- Razpoložljivost (ang. availability)
  - Delež časa, ko je sistem lahko uporabljen za opravljanje običajnih nalog.
- Odpornost (ang. fault tolerance)
  - Zmožnost delovanja pod neustreznimi pogoji (ang. erroneous conditions).
- Varnost (ang. security)
  - Zmožnost zoperstavljanja zlonamernim napadom.
- Varnost (ang. safety)
  - Zmožnost ne ogrožanja človeških življenj, tudi v primeru prisotnosti napak in odpovedi.

5.2.3 cena (ang. cost)

- Cena razvoja (ang. development cost)
  - Cena razvoja (začetnega) sistema.
- Cena namestitve (ang. deployment cost)
  - Cena namestitve in izobraževanja uporabnikov.
- Cena nadgradnje (ang. upgrade cost)
  - Cena prevedbe podatkov predhodnega sistema.
- Cena vzdrževanja (ang. maintenance cost)
  - Cena odpravljanja napak in razširjanja sistema.
- Cena administriranja (ang. administration cost)
  - Stroški administriranja sistema.

5.2.4 zmožnost vzdrževanja (ang. maintainance)

- Razširljivost (ang. extensibility)
  - Kako enostavno je dodajanje novih funkcionalnosti?
- Spremenljivost (ang. modifiability)
  - Kako enostavno je spremeniti funkcionalnost sistema?
- Prilagodljivost (ang. adaptability)
  - Kako enostavno je sistem prevesti za uporabo na drugih področjih uporabe?
- Prenosljivost (ang. portability)
  - Kako enostavno je prenesti sistem na druge platforme?
- Čitljivost (ang. readability)
  - Kako enostavno je razumeti sistem na osnovi branja izvorne kode?
- Sledljivost zahtev (ang. tracebility of requirements)
  - Kako enostavno je kodo povezati s specifično zahtevo?

5.2.5 uporabniške zahteve (ang. user criteria)

- Koristnost (ang. utility)
  - Kako dobro sistem podpira delo uporabnika?
- Uporabnost (ang. usability)
  - Kako enostavno je uporabniku uporabljati sistem?

5.3 Dekompozicija programskega sistema

5.3.1 "Divide and conquer"

Obravnavanje velikih sistemov je veliko težje od obravnavanja serije majhnih komponent - podsistemov.
Deli in zmagaj!

- Manjše komponente je lažje razumeti kot velike.
- Za vsak podsistem lahko skrbijo drugi ljudje.
- Možnost specializacije posameznih programskih inženirjev.
- Komponente je mogoče zamenjati brez večjih popravkov ostalega sistema.

5.3.2 Koncept podsistemov

Podsistemi nudijo storitve drugim podsistemom.
Storitev je skupina sorodnih operacij.
Ob načrtovanju sistema (arhitektura) je potrebno definirati podsisteme v smislu storitev, ki jih podsistemi nudijo.
Operacije podsistema, ki so na voljo drugim sistemom, tvorijo vmesnik podsistema (ang. subsystem interface), imenovan tudi programski vmesnik (ang. application programmer interface) (API).

5.4 Sloji in particije

Vsak podsistem (sloj ali particija) je odgovoren za drugačne storitve. Podsistemi so medsebojno šibko povezani in pogosto delujejo samostojno.

5.4.1 Sloji (ang. layers)

Sloji (ang. layers) ločujejo podsisteme hierarhično, vertikalno.

5.4.2 Particije (ang. partitions)

Particije (ang. partitions) ločujejo podsisteme istega sloja, horizontalno.

5.5 Kohezija podsistemov

Kohezija (ang. cohesion) govori o notranji povezanosti znotraj podsistema. Kohezija je velika,če podsistem združuje medseboj odvisne oz. povezane komponente ter ne vsebuje ostalih, nepovezanih komponent).
Zaželjena je velika kohezija podsistema, kajti takšni podsistemi so lažji za razumevanje in spreminjanje.

5.5.1 Funkcijska kohezija

5.5.2 Slojna kohezija

5.5.3 Komunikacijska kohezija

5.5.4 Sekvenčna kohezija

5.5.5 Proceduralna kohezija

5.5.6 Časovna kohezija

5.5.7 Kohezija pripomočkov

5.6 Modeli odvisnosti

Modeli odvisnosti pomagajo pri analizi programskih arhitektur v smislu odkrivanja težavnih odvisnosti med moduli.

5.6.1 Matrika odvisnosti

5.6.2 Graf odvisnosti

5.7 Relacije med programsko in strojno opremo

Mnogi sistemi delujejo na več računalnikih, ki med seboj komunicirajo preko intraneta ali interneta.
Potrebna je porazdelitev komponent sistema po računalnikih (vozliščih) in načrt komunikacijske infrastrukture.
Pogosto so pri delitvi podsistemov po vozliščih potrebni dodatni podsistemi:

- Komunikacijski podsistemi, namenjeni prenosu podatkov in zagotavljanju nadzora nad sočasnim izvajanjem transakcij (ang. concurrency),
- Kontrolni sistemi, namenjeni zagotavljanju zanesljivosti delovanja (ang. reliability).

Z razporeditvijo podsistemov po vozliščih je mogoče funkcionalnost in procesorsko moč porazdeliti tako, da je na voljo tam, kjer jo potrebujemo. Izzivi hranjenja, prenašanja podvajanja in sinhronizacije podatkov.

5.8 Upravljanje s perzistentnimi podatki

5.9 Politika nadzora dostopa

5.10 Mehanizem globalnega kontrolnega toka

5.11 Obravnavanje mejnih primerov

5.12 Arhitekturni vzorci

5.12.1 "Layers"

5.12.2 "Model/View/Controller"

5.12.3 "Shared repository"

5.12.4 "Microkernel"

5.12.5 "Client/Server"

5.12.6 "Peer-to-peer"

5.12.7 "Pipes and filters"

6 Načrtovanje programa

6.1 Objektni in strukturni pristop

Objektno orientiran pristop (OOAD):

- Načrtovanje sistema
  - Dekompozicija na podsisteme
- Načrtovanje objektov
  - Izbira implementacijskega jezika
  - Izbira algoritmov in določitev podatkovnih struktur

Strukturna analiza in načrtovanje (SAD):

- Predhodno načrtovanje (Preliminary Design)
  - Dekompozicija na podsisteme
  - Določitev podatkovnih struktur
- Podrobno načrtovanje (Detailed Design)
  - Izbira algoritmov
  - Dopolnitev podatkovnih struktur
  - Izbira implementacijskega jezika
  - Običajno enotna aktivnost s predhodnim načrtovanjem

6.2 Usmeritve pri načrtovanju

6.2.1 Dedovanje (ang. inheritance)

Dedovanje se uporablja za doseganje dveh različnih ciljev:

- Razvrščanje razredov v skupine (taksonomija)
  - Uporablja se v fazi analize zahtev
  - Za identifikacijo objektov, ki so medsebojno hierarhično odvisni.
  - Pripomore k razumljivosti modelov.
- Ponovna uporaba
  - Uporablja se v fazi načrtovanja objektov.
  - Pripomore k možnosti uporabe obstoječih objektov ter možnosti njihovega spreminjanja in razširjanja
  - Razširitev funkcionalnosti (implementacijsko dedovanje), Specifikacija vmesnikov (specifikacijsko dedovanje)

6.2.2 Delegiranje in dedovanje

Delegiranje

+ Fleksibilnost
+ Omogočeno za vse programske jezike

- Slabša učinkovitost, zaradi ovijanja (encapsulation)

Dedovanje

+ Enostavnost
+ Podprto v mnogih programskih jezikih
+ Enostavno dodajanje novih funkcionalnosti
- Izpostavljenost detajlov osnovnega razreda
- Vsaka sprememba osnovnega razreda zahteva spremembo dedovanega razreda

6.2.3 Liskov princip zamenjave - LSP

Liskov princip zamenjave (ang. Liskov substitution principle) po Barbari Liskov in njenemu predavanju "Data abstraction and hierarchy" 1987.
Razred S "ustreza" razredu T, če je mogoče vsak objekt razreda T nadomestiti z objektom razreda S, ne da bi pri tem ogrozili pravilnost delovanja programa. Če osnovni razred zadosti zahtevam aplikacije, potem mora tudi LSP nadomestni razred zadostiti istim zahtevam, tako da s stališča vmesnika deluje povsem enako in da za enake argumente enake rezultate. Če želimo da je nek razred S podtip (ang. subclass) razreda T, mora S "ustrezati" T.

6.2.4 Abstrakcija

Označuje bistvene lastnosti objekta, ki ta objekt ločijo od ostalih objektov. Določa konceptualne meje objektov, hkrati pa dopušča različno interpretacijo, odvisno od pogleda opazovalca.

6.2.5 Odprto/zaprti princip - (ang. the open/closed principle)

Razred naj bo odprt (open) za razširitve –dodajanje operacij ali polj podatkovne strukture. Razred naj bo zaprt (closed) za spremembe, če je na voljo za uporabo drugim razredom – določati mora stabilnen in dobro definiran vmesnik. V objektno orientiranih programskih jezikih je to realizirano z dedovanjem (ang. inheritance) in polimorfnostjo (ang. polymorphism).

6.2.6 Ovijanje (ang. encapsulation)

Ovijanje je proces zaokrožanja elementov z določanjem abstrakcije, ki določa strukturo in obnašanje elementov. Ovijanje služi za ločevanje "pogodbenih" vmesnikov (določenih z abstrakcijo) od implementacije.

6.2.7 Kohezija razreda (ang. class cohesion)

Kohezija razreda meri medsebojno povezanost metod zunanjega vmesnika razreda.
Kako zaključeno deluje razred v smislu implementacije abstrakcije.
Želimo visoko kohezijo!
Kvantitativna mera kohezije razreda ne obstaja.
Simptomi nizke kohezije:

- Razred vsebuje nekaj komponent (metod), ki niso definirane za vse objekte razreda. Eventuelno izboljšanje z ločitvijo v dva razreda.
- Razred vsebuje komponente (metode), ki glede na namen razreda (domeno) niso relevantne.

6.2.8 Modularnost

Modularen program sestoji iz dobro definiranih, konceptualno enostavnih in neodvisnih enot - modulov, ki komunicirajo preko dobro definiranih vmesnikov.
Prednosti modularnosti

- enostavnejše razumevanje in razlaganje,
- enostavnejše dokumentiranje,
- enostavnejše spreminjanje,
- enostavnejše testiranje in razhroščevanje,
- boljša večkratna uporabljivost,
- enostavnejše nastavljanje.
- Modul je vsaka enota programa, ki sestoji iz več delov (program, podprogram, paket, razred, operacija, vrstica kode...)

Principi modularnosti

Majhni moduli

- Načrtovanje z majhnimi moduli je boljše

Skrivanje informacij (ang. information hiding)

- Vsak modul naj drugim modulom prikrije detajle o njegovi interni strukturi in procesiranju.

Najmanjši možni privilegiji

- Moduli naj nimajo dostopa do nepotrebnih virov.

Povezanost, odvisnost

- Povezanost med moduli mora biti čim manjša

Kohezija

- Kohezija (odvisnost znotraj modula) mora biti čim večja.

6.3 Načrtovalski vzorci

Načrtovalski vzorci so:
- Elegantna rešitev, ki začetnikom ni samoumevna.
- Neodvisni od sistema, programskega jezika ali domene aplikacije.
- Dodatano uspešni v realnih OO sistemih.
- Enostavni, vsebujejo le nekaj razredov.
- Večkrat uporabni in na tak splošen način tudi dokumentirani.
- Objektno orientirani, uporabljajo OO mehanizme kot so razredi, objekti, generalizacija in polimorfizem.

Določa jih:
- Ime vzorca
- Problem – načrtovalski vzorci vsebujejo opis problema, za katerega je primerno uporabiti vzorec.
- Rešitev – načrtovalski vzorci podajajo elemente in njihove medsebojne relacije, ki tvorijo rešitev problema.
- Posledice – rezultati in posledice uporabe načrtovalskega vzorca.

6.3.1 Delitev načrtovalskih vzorcev

Glede na namen:
  - Kreacijski (ang. creational)
  - Določajo proces kreiranja objektov
- Strukturni (ang. structural)
  - Določajo kompozicijo razredov in objektov
- Vedenjski (ang. behavioural)
  - Določajo način interakcije med razredi in porazdelitev odgovornosti.

Glede na področje delovanja:
- Razredni (class)
  - Vzorec se ukvarja z razredi in njihovimi medsebojnimi relacijami, ki so določene z dedovanjem in so statične.
- Objektni (object)
  - Vzorec se ukvarja z relacijami med objekti, ki so dinamične in se lahko spreminjajo v času izvajanja.

6.4 Dokumentiranje

- Rezultat načrtovanja programa (načrtovanja objektov) je načrt programa, tudi detajlni načrt.

- Načrt programa (detajlni načrt) mora določati
  - notranjo zgradbo vseh delov programa,
  - njihovo povezanost (vmesniki)
  - njihovo strukturo (operacije, atributi, dedovanja, kompozicije)
  - način procesiranja,
  - algoritme,
  - podatkovne strukture,
  - lastnosti (prepustnost, zanesljivost...)
- Izdelava detajlnega načrta se pogosto deloma prekriva s kodiranjem.

6.4.1 Specifikacija vmesnikov

- Sintaksa (ang. Syntax)
  - Določa elemente komunikacijskega medija in način njihovega združevanja v sporočila.
- Semantika (ang. Semantics)
  - Določa pomen sporočil
- Pragmatika (ang. Pragmatics)
  - Določa kako so sporočila uporabljena za opravljanje nalog.

7 Testiranje

7.1 Verifikacija in validacija

Verifikacija:

- Posamezne komponente so testirane glede na skladnost s specifikacijami programa – načrta programa.
- Posamezne komponente so združene v programski sistem in testirane glede na specifikacije sistema – načrta sistema.

Validacija:

- Poteka po vnaprej pripravljenem načrtu, ki izhaja iz specifikacij zahtev. Preverja se skladnost sistema z zahtevami (funkcijskimi in nefunkcijskimi, upoštevaje tudi implicitne oziroma "samoumevne" zahteve).

7.2 Recenzija

Gre za aktivnost pri kateri eden ali več ljudi sistematično pregleda izvorno kodo ali dokumentacijo z namenom iskanja morebitnih napak. Običajno recenzija vključuje sestanek, pri katerem sodelujejo tudi avtorji, čeprav recenzenti lahko pregled opravijo tudi samostojno oziroma ločeno.

7.3 Testiranje

Testiranje je proces izvajanja programa z namenom iskanja napak (error) pred predajo končnim uporabnikom.
Preizkuševalec (tester) izvede serijo testnih primerov z namenom porušiti pravilno delovanje programa.
Dober testni primer je tisti, za katerega obstaja velika verjetnost da bo razkril še ne odkrito napako.
Test je uspešen, če je z njim najdena prej nepoznana napaka.

7.3.1 Principi testiranja

- Vsi testni primeri naj bodo povezani z zahtevami.
  - Najhujše napake so tiste, ki povzročijo, da program ne izpolnjuje zahtev.
- Testiranje naj bo načrtovano pred začetkom testiranja.
  - Načrtovanje testiranja se lahko začne takoj ko so izdelane specifikacije zahtev.
- Pri testiranju velja zakon 80-20 (princip Pareto).

    - 80% vseh napak odkritih med testiranjem izvira iz 20% programskih komponent. Te komponente je potrebno testirati toliko bolj temeljito.
- Izčrpno testiranje ni mogoče.
    - Število možnih poti izvajanja programa je že pri manjših programih lahko preveliko, da bi lahko preizkusili vse možnosti.
- Najučinkovitejše testiranje opravijo neodvisni preizkuševalci
    - Razvijalec, ki je program naredil, ni najprimernejši preizkuševalec zaradi navzkrižja interesov.

7.3.2 Kdo testira?

Razvojni inženir

- Dobro razume sistem.
- Glavni interes je pravočasna predaja produkta.
- Je konstruktiven in pri testiranju do sistema "nežen".

Neodvisni preizkuševalec:

- Ne pozna sistema in ga mora še spoznati.
- Glavni interes je zagotavljanje kakovosti.
- Je destruktiven in bo iskal načine za zlom sistema.

7.3.3 Testabilnost (ang. testability)

Testabilnost je lastnost, ki pove kako enostavno je testirati program. Dobra testabilnost programa naj bi bila eden od ciljev pri načrtovanju.
Testabilnost je odvisna od več drugih lastnosti programa:

- **Izvedljivost** (ang. operability) – program je mogoče neovirano izvajati.
- **Možnost opazovanja** (ang. observalibiliy) – program nudi jasen vpogled v rezultate izvajanja.
- **Možnost nadzora** (ang. controlability) – stopnja do katere je testiranje mogoče avtomatizirati in optimirati.
- **Razstavljivost** (ang. decomposability) – testiranje je lahko osredotočeno na posamezen modul.
- **Enostavnost** (ang. simplicity) – enostavnost programa omogoča temeljitejše testiranje.
- **Stabilnost** (ang. stability) – stopnja spreminjanja med testiranjem.
- **Razumljivost** (ang. understandability) – razumevanje delovanja programa, tudi na osnovi dokumentacije.

7.4 Načini testiranja

7.4.1 Strukturno testiranje (white-box testing)

Testiranje temelji na poznavanju notranje strukture programa.
Cilj je zagotoviti, da je bil vsak pogoj upoštevan vsaj enkrat in vsak del izvorne kode izveden vsaj enkrat.

Upoštevanje celotne strukture Pri testiranju preveri celotno strukturo programa – vse poti izvajanja programa.

- Logične napake in napačne predpostavke so inverzno sorazmerne verjetnosti izvajanja poti.
- Pogosto verjamemo, da ni verjetno, da bo določena pot izvedena. Realnost je pogosto v nasprotju z intuicijo.
- Tipkarske napake se pojavljajo naključno. Verjetno je, da bodo prisotne tudi na netestiranih poteh.

Neodvisna pot je pot skozi program, ki vsebuje vsaj en nov del programske kode ali nov pogoj. Število neodvisnih poti je odvisno od **ciklomatične kompleksnosti** - mere logične kompleksnosti programa. Izračun ciklomatične kompleksnosti V(G) na osnovi grafa izvajanja G:
$V(G)$ = število enostavnih odločitev +1

7.4.2 Vedenjsko testiranje (black-box testing)

Testiranje temelji na (deklariranih) zunanjih lastnostih sistema.

7.5 Razhroščevanje

Razhroščevanje je diagnostični proces kot posledica uspešnega testiranja. Namen je odprava odkritih napak. Manifestacija napake (simptom) in vzrok zanjo nimata nujno očitne medsebojne povezave. Razhroščevanje je miselni proces, ki simptom poveže z vzrokom napake.

Tehnike:

- Obsežno testiranje
- Sledenje
- Indukcija
- Dedukcija
- Uporabi orodja, (npr. debugger) za boljši vpogled v dogajanje v programu.
- Po "odpravi" napake ne pozabi izvesti regresijskega testiranja!

8 Vzdrževanje

Splošno:

- Popravila in vse druge aktivnosti, ki ohranjajo produkt v stanju obratovanja.

Programski produkti:

- Vzdrževanje? Programski produkti se ne kvarijo in ne potrebujejo periodičnih vzdrževalnih servisiranj...
- Vzdrževanje pomeni spremembe programskega produkta po izdaji z namenom:
- odpravljanja napak,
- izboljšanja zmogljivosti (prepustnost sistema...)
- prilagoditve produkta spremenjenemu okolju.
- prilagajanje novim zahtevam

Vzdrževanje je ključno za ohranjanje uporabe (programskega) produkta!

8.1 Cena vzdrževanja

Vzdrževanje je drago!
Običajno je cena vzdrževanja 2 do 100 krat višja od cene razvoja!
Na ceno (težavnost) vzdrževanja vplivajo tako tehnični kot tudi netehnični dejavniki kot na primer: tip aplikacije, edinstvenost sistema, fluktuacija in izkušenost kadra, življenjska doba sistema, odvisnost od spremenljivega okolja, karakteristike strojne opreme, kakovost načrtovanja (strukturiranost), kakovost izvorne kode, kakovost dokumentacije, kakovost testiranja, število uporabnikov, uporaba orodij...

8.2 Evolucija

Vzdrževanje programskih produktov pomeni njihovo evolucijo!

Kaj se razvija?

- Specifikacija zahtev
  - Evolucija zahtev
- Načrtovanje
  - Evolucija arhitekture
- Izvedba
  - Evolucija podatkov
  - Evolucija izvorne kode
  - Evolucija dokumentacije

    - Evolucija uporabljene tehnologije
- Testiranje
    - Evolucija testiranja (testni primeri)

Evolucija mora biti usklajena (ang. co-evolution problem)

8.2.1 Evolucijski tipi programskih produktov

Meir Manny Lehman je glede na potrebo po vzdrževanju opredelil tri tipe programskih produktov:
- S-system: formalno definirani sistemi, ki v celotni življenjski dobi ne zahtevajo sprememb. V praksi pojavijo zelo redko!
- P-system: sistemi katerih zahteve temeljijo na aplikativni rešitvi problema. Potrebujejo inkrementalno prilagajanje dejanskemu svetu.
- E-system: sistemi vgrajeni v dejanski svet, zahtevajo stalno prilagajanje dejanskemu svetu. V to skupino sodi večina programskih produktov!

8.3 Lehmanovi zakoni evolucije

Meir Manny Lehman je zbral 8 zakonov evolucije programskih produktov.
1. Continuing Change
2. Increasing Complexity
3. Large Program Evolution
4. Invariant Work-Rate
5. Conservation of Familiarity
6. Continuing Growth
7. Declining Quality
8. Feedback System

Posvetili se bomo štirim izmed omenjenih zakonov.

8.3.1 1. Law of continuing change

Zakon o stalnosti spreminjanja:
Sisteme je potrebno stalno prilagajati, sicer postopno vse slabše zadovoljujejo potrebe uporabnikov. Odstopanje med sistemom in njegovim področjem delovanja ustvarja povratno zanko, ki spodbuja spreminjanje sistema.

8.3.2 2. Law of increasing complexity

Zakon o povečevanju kompleksnosti:
Z razvojem (spreminjanjem) sistema se kompleksnost sistema povečuje, razen če se kompleksnosti z dodatnim vloženim delom ne zmanjšuje. Če se

spreminjanje sistema dogaja brez premisleka o njegovi strukturi, se kompleksnost sistema povečuje in otežuje nadaljnje spremembe. Nasprotno, če se sredstva porabijo tudi za boj proti kompleksnosti, jih je manj na voljo za ostalo spreminjanje sistema. Ne glede na izbrano razmerje se hitrost rasti sistema neizogibno manjša.

8.3.3 6. Law of continuous growth

Zakon o stalni rasti:
Če želimo ohranjati zadovoljstvo uporabnikov, se morajo funkcijske zmožnosti programskega produkta ves čas življenjske dobe povečevati. Vsaki izvedbi sistema je zahteve potrebno omejiti. Izpuščene ali prezrte lastnosti bodo sprožile nadaljnje zahteve po spremembah. To je povratna zanka od uporabnikov.

8.3.4 7. Law of declining quality

Zakon o upadanju kakovosti:
Če se sistem striktno ne prilagaja spremembam v okolju, v katerem deluje, bo videti, da kakovost sistema upada. Sistem je zgrajen na množici predpostavk in ne glede na njihovo veljavnost v določenem trenutku, bo spreminjajoči svet na njih vplival v smeri njihovega zavračanja. Če se teh predpostavk ne identificira in popravi, bo videti, da kakovost sistema upada, še posebej v primerjavi z alternativnimi produkti, ki so na trg prišli z bolj sveže formuliranimi predpostavkami.

8.4 Veljavnost Lehmanovih zakonov

Lehmanovi zakoni se dobro prilegajo komercialnim programskim produktom.

V primeru "nestandardne" programske opreme so prisoti dvomi o veljavnosti Lehmanovih zakonov.

- Odprtokodna programska oprema...
  - Linux primer...
- Drugačnost:
  - razvijalci rešujejo lastne probleme,
  - manjši časovni pritisk...

8.5 Proces vzdrževanja

Glavni koraki so:

- Pridobivanje vlog za vzdrževanje / ideje.
- Specificiranje zahteve po spremembah.
- Načrtovanje sprememb
- Izvedba sprememb
- Testiranje
- Izobraževanje
- Izdaja
- Dokumentiranje

8.6 Tehnike in orodja za vzdrževanje

Upravljanje konfiguracij
Analiza vplivov (impact analysis)

8.6.1 Inženiring (ang. forward engineering)

Inženiring je tradicionalni proces ki poteka od visokega nivoja abstrakcije in logičnega, izvedbeno neodvisnega načrtovanja, do fizične izvedbe sistema.

8.6.2 Obratni inženiring (ang. reverse engineering)

Obratni inženiring je proces analize sistema za identifikacijo sistemskih komponent, njihovih medsebojnih relacij in predstavitve sistema v drugačni obliki z višjo stopnjo abstrakcije.

8.6.3 Ponovni inženiring (ang. Reengineering)

Ponovni inženiring je preiskava in spreminjanje sistema z namenom rekonstrukcije in implementacije v novo obliko z enako zunanjo funkcionalnostjo.

8.7 Racionalnost

Vključuje:

- **Problematiko**, na katero se nanaša
- **Alternative**, ki so bile upoštevane
- **Odločitve**, ki so bile sprejete za razrešitev problematike
- **Kriterije**, ki so vodili do odločitve in
- **Diskusije**, ki so bile v razvojni skupini potrebne za dosego odločitve.

9 Uporabniški vmesniki

Interakcija človeka z računalnikom (ang. Human-Computer Interaction (HCI)):
- tipkovnica
- miška
- igralna palica
- wii kontroler
- rokavice

9.1 CLI in GUI

CLI = Command-line interface
GUI = Graphical user interface

9.1.1 Prednosti GUI

- So enostavni za uporabo
  - Uporabniki brez izkušenj se lahko hitro naučijo dela s sistemom
- Uporabnik lahko hitro prehaja iz ene naloge na drugo, med različnimi aplikacijami, ki tečejo hkrati
  - Informacije posamezne aplikacije se z delom z drugo aplikacijo ne izgubijo (izgled)
- Možnost hitre interakcije s sistemom na področju celotnega zaslona

9.2 Principi pri načrtovanju GUI

9.2.1 Miselni modeli (ang. mental models)

Miselni modeli so človekov lastni in osebni način razumevanja sveta. Ljudje izoblikujemo miselne modele preko izkušenj, vzgoje, šolanja, navodil...
Miselni modeli so v principu pogosto napačni, a kljub temu efektivni (pravilni v smislu uporabe).

9.2.2 Metafora (ang. metaphor)

Metafore so primerjave, ki pokažejo kako dve stvari, ki sta v večini pogledov različni, po drugi strani v enem pomembnem pogledu enaki. Metafore so analogije med dvema objektoma ali idejama.

9.2.3 Evidentna funkcionalnost (ang. affordance)

Evidentna funkcionalnost je lastnost, da nek objekt kaže možnosti, ki jih nudi uporabniku.

Evidentna funkcionalnost je bistveno pomembnejša kot uporabniška navodila – uporabniku postane način dela evidenten že iz predmeta samega (npr. iz izgleda uporabniškega vmesnika).

9.2.4 Princip povezanosti (ang. mapping)

Princip povezanosti odraža povezavo med dvema stvarema.

- Volan – zasuk levo pomeni zavijanje levo...
- Upravljanje računalnika z miško...
- Glasnejši glas pomeni večjo pomembnost...
- Temperaturno kodiranje: barva odraža temperaturo – rdeča = vroče!

Z uporabo naravnih povezanosti lahko načrtovalec uporabniškega vmesnika doseže, da si uporabnik možne funkcije enostavno nauči in zapomni!

9.2.5 Princip povratnih informacij (ang. feedback)

Posredovanje povratnih informacij nazaj uporabniku:

- Katera aktivnost se je dejansko izvedla / se izvaja
- Kakšni so rezultati predhodne aktivnosti

Povratne informacije morajo biti pravočasne!
Primeri:

- Indikatorji napredovanja operacije pri časovno daljših operacijah (pridobivanje podatkov iz omrežja, kopiranje, namestitve...)
- Tipkovnice bi lahko bile povsem tihe, pa vendar uporabniki ob pritisku tipke pričakujemo "klik".
- Premikanje miške po zaslonu. Kadar je sistem prezaposlen in miška zaostaja, je uporabnikom neudobno.

9.2.6 Omejitve

- Omejitve predstavljajo način za olajšanje interakcije uporabnika s sistemom.
- Omejitve so lahko fizične, semantične, kulturne, logične.
- Omejitve lahko omejujejo uporabo določenih aktivnosti:
  - Določene aktivnosti ne smejo biti izvedene.
  - Določene aktivnosti morajo biti izvedene.
- Z omejitvami lahko podrobneje določimo uporabnikove možnosti, npr. pri vnosu podatkov.
  - Omejitev možnosti izbire (radio box, on/off gumbi, seznami...)
- Uporabnika je pri interakciji potrebno omejiti na funkcionalnost, ki jo nudi aplikacija.

9.3 Prikaz informacij

Informacije so lahko posredovane neposredno ali pa so pretvorjene v nek način njihove predstavitve. Model-View-Controller arhitekturni vzorec podaja način, kako omogočiti več različnih predstavitev istih podatkov (glej načrtovanje sistema).

9.3.1 Analogno in digitalno

Analogna predstavitev

- Hitro podajanje približnih vrednosti.
- Možnost podajanja relativnih vrednosti
- Boljša indikacija izjemnih vrednosti

Digitalna predstavitev

- Kompakten prikaz zavzame malo prostora
- Podajanje natančnih vrednosti

9.3.2 Uporaba barv

Barve predstavljajo dodatno dimenzijo uporabniškega vmesnika. Barve je mogoče uporabiti tako, da uporabnik lažje razume kompleksne strukture informacij. Barve so lahko uporabljene za poudarjanje izjemnih dogodkov.
Pogoste napake pri uporabi barv?

- Uporaba barv za označevanje pomena
- Prekomerna uporaba barv.

9.3.3 Uporaba pogovornih oken

Zahtevana pogovorna okna so lahko velika in na zaslonu dominantna. Nezahtevana pogovorna okna naj bodo manjša in nevsiljiva. Pogovorna okna imajo prehoden značaj. Nezahtevanim pogovornim oknom se je potrebno izogibati.

9.3.4 Obvestila o napakah

Obvestila o napakah so pogosto zlorabljena. Obvestila o napakah poskušajmo odpraviti. Obvestila o napakah nikoli niso priljubljena.

9.4 Načrtovanje uporabniških vmesnikov

9.4.1 Zmote

- Uporabniški vmesnik se mi zdi zadovoljiv – torej bo zadovoljil tudi druge.
- Uporabniški vmesnik zadovolji povprečno osebo – torej bo zadovoljil kogar koli.
- Raznolikost ljudi je tako velika, da je ne moremo zadovoljiti z nobenim vmesnikom. Vendar, ker smo ljudje čudovito prilagodljivi, to niti ni pomembno.
- Dobri uporabniški vmesniki so dragi, vendar pa produkte običajno kupujemo na osnovi izgleda, torej uporabniški vmesnik ni pomemben.
- Dobri uporabniški vmesniki so čudovita ideja – kadar načrtujem se tega zavedam – delam intuitivno in po zdravi pameti, zato mi jih ni potrebno testirati.

9.4.2 Značaji programov

Suveren značaj

- Programi, ki se polastijo celotnega zaslona za daljše časovno obdobje.
- Npr. urejevalniki besedil, preglednice, programska okolja...
- Načrtovana so za optimalno delo izkušenih uporabnikov.
- Čas učenja je relativno kratek glede na celoten čas porabljen za delo z aplikacijo.
- Pričakuje se, da bo deloval v maksimiranem oknu.
- Uporablja naj neizstopajoče barve in teksture.
- Uporabniški vmesnik je lahko zelo bogat, npr. z več orodnimi vrsticami (ang. toolbar).

Prehoden značaj

- Programi, ki se pokažejo in končajo, ki izvedejo eno samo operacijo.
- Delujejo kot podpora suverenim aplikacijam.
- Npr. Windows Explorer, WinZip
- Uporabniki z njimi porabijo bistveno manj časa kot s suverenimi programi.
- Zaželeno je, da porabijo čim manjši delež prostora na zaslonu.
- Uporabniški vmesnik mora biti enostavnejši in bolj poudarjen, saj ima uporabnik manj časa za spoznavanje z aplikacijo.
- Vse relevantne informacije morajo biti vidne takoj.
- Navodila naj bodo vgrajena v izgled aplikacije.

- Ni nujno, da obstaja potreba po povečevanju oken, npr. Windows Calculator

Demonski značaj

- Programi, ki običajno ne zahtevajo interakcije z uporabnikom in delujejo v ozadju.
- Npr. Print Manager, Screen Savers
- Uporabniški vmesnik mora biti enostaven in poudarjen.
- Uporabniki bodo redko videli uporabniški vmesnik.
- Do uporabniškega vmesnika se dostopa na vnaprej določen način, npr. preko kontrolne plošče (ang. Control Panel). Ne uporablja se ikon na namizju.

Parazitski značaj

- Aplikacija, ki je vedno prisotna, vendar je majhna in opravlja podporne vloge.
- Npr. ura
- Uporabniški vmesnik mora biti enostaven, zavzemati mora zelo majhen del zaslona.

9.4.3 Način dela

Ko uporabnik želi opraviti določeno nalogo, se srečamo s tremi aktivnostmi.

- Določanje ciljev (ang. goals) – kaj uporabnik želi, kaj pričakuje od naprave in kakšen je njegov namen uporabe?
- Izvajanje (ang. execution) – kako uporabnik doseže cilje z uporabo dane naprave?
- Vrednotenje (ang. evaluation) – kako uporabnik ve ali je dosegel zastavljene cilje?

Srečamo se z dvema razkorakoma med želenim in dejanskim obnašanjem naprave/programa.

- Prepad izvajanja
  - razlike med uporabnikovimi nameni in aktivnostmi, ki so na voljo.
  - na kakšen način naj uporabnik sistemu poda informacije?
- Prepad vrednotenja
  - trud, ki je potreben, da uporabnik interpretira stanje naprave.
  - kako uporabniku predstaviti informacije?
  - stopnja zadovoljstva uporabnika glede na namene in pričakovanja.

Način dela uporabnika opisuje model sedmih korakov aktivnosti. (ang. The Seven Stages of Action)

9.4.4 The Seven Stages of Action

1. Tvorjenje namena iz cilja – iskanje projekta.
2. Določitev aktivnosti – potreben pregled strani, da ugotovimo kako opraviti aktivnost.
3. Izvajanje aktivnosti – vnos podatkov v polje Search.
4. Zaznavanje stanja – osveži se vsebina strani.
5. Razumevanje stanja – najdemo rezultate iskanja.
6. Vrednotenje rezultata – ali smo našli ustrezno povezavo?
7. Celoten proces lahko poteka v več iteracijah.

Za načrtovanje uporabniških vmesnikov. Uporabniku želimo kar se da olajšati prehod med koraki aktivnosti in kar se da zmanjšati število ciklov. Za dosego teh ciljev uporabimo že omenjene principe načrtovanja.

9.4.5 Dokumentiranje

Splošno priznane metode dokumentiranja uporabniških vmesnikov NI. Tudi nekega formalnega načina specificiranja uporabniških vmesnikov NI.
Možnosti, ki so na voljo:

- Slike izgleda
  - Dejanski izgled aplikacije, različna okna (zaslon) v različnih situacijah.
- Primeri uporabe
  - Funkcionalnost lahko opišemo s primeri uporabe, ki se sklicujejo na slike izgleda vmesnika.
- UML diagrami interakcije
  - Z njimi je mogoče prikazati interakcije med okni.
  - To ni standardni način uporabe diagramov interakcij.

9.4.6 Vrednotenje uporabniških vmesnikov

Vrednotenje uporabniških vmesnikov je lahko zelo obsežno, težavno in zato tudi drago.
Nekaj metod vrednotenja:

- Eksperimentiranje – preizkus različnih načinov uporabe elementov uporabniškega vmesnika ločeno in kot celota.
- Vprašalniki – uporabniki odgovorijo na vprašanja, ki zadevajo njihovo interakcijo s sistemom.
- Neposredno opazovanje – opazovanje interakcije uporabnikov s sistemom.
- Posredno opazovanje – analiza video posnetkov interakcije uporabnikov s sistemom.

- Avtomatsko pridobivanje podatkov – analiza na podlagi vgrajenih postopkov beleženja dela z uporabniškim vmesnikom.

Kaj vrednotiti?

- **Čas učenja** – koliko časa potrebuje uporabnik, da je zmožen učinkovito uporabljati sistem.
- **Hitrost dela** – kako se hitrost odzivanja sistema sklada z uporabnikovimi potrebami oziroma navadami.
- **Robustnost** – kako toleranten je sistem do uporabniških napak.
- **Regenerativnost** – kakšna je sposobnost sistema po vzpostavitvi normalnega delovanja po uporabniški napaki.
- **Prilagodljivost** – kako močno je sistem vezan na en sam način uporabe.
- **Zanesljivost** – kakšna je pogostost napak v delovanju sistema.
- **Uporabniško zadovoljstvo** – subjektivno mnenje uporabnikov glede uporabe sistema.
- **Ponavljajoče operacije** – pogostost uporabe posameznih ukazov uporabniškega vmesnika (pogosto povezano z dostopnostjo teh operacij).