B3 Programmeerimine

B 3.1 Tarkvara arendusmeetodid ja tehnikad

Tarkvara arendamine on tänapäeval enamasti meeskonnatöö ja seetõttu on siin valdkonnas mõeldud välja erinevaid reeglistike ja meetodeid. Need reeglid ja head tavad on väljatöötatud eelkõige selleks, et tarkvaraarendusega seotud inimesed mõistaksid üksteist ja nende tööd oleks võimalik kõigile arusaadavalt standardiseerida. Standardiseerimine abil on võimalik tagada tarkvara kvaliteeti ja vähendada tarkvara arendamiseks kuluvat aega ja raha.

Tarkvara arenduse võib laias laastus jagada järgmisteks alamülesanneteks
1. Vajaduste kirjeldamine ja nende analüüs

2. Tarkvaratoote disain

3. Teostamine

4. Testimine

5. Toote väljalase (juurutamine)

6. Toote hooldus

## Vajaduste kirjeldamine ja nende analüüs

Tarkvara loomine algab tavaliselt vajaduste kirjeldamisest ja nende analüüsist. Mida täpsem ja korrektsem on tarkvara vajaduste kirjeldus ja nende analüüs, seda lihtsam on hiljem kõiki muid arenduse etappe teostada. Peamine probleem selles faasis seisneb tellija ja tarkvara arendaja erinevas nägemuses. Tihti on nii, et tarkvaratellija ei tea midagi programmeerimisest ega tarkvara disainist, aga tal on väga selge nägemus sellest, mida ta vajab. Tarkvaraarendaja aga ei kujuta väga täpselt ette tellija tööprotsessi, vaid kipub liiga vara mõtlema tarkavara teostamise võimalustele ja vahenditele.

## Tarkvaratoote disain

Tarkvara disaini etapis peab arendaja mõtlema, kuidas tarkvara teostada. Peamiseks ülesandeks siin etapis on mõelda välja kuidas võimalikult väikeste raha ja ajakuluga teostada tellijale vajalik tarkvaratoode. Tuleb jälgida kuidas loodav tarkvaratoote disain vastaks järgmistele nõudmistele:

* Usaldusväärsus
* Muutmise võimalikkus
* Arusaadavus
* Uuestikasutatavus

Usaldusväärsuse all mõeldakse siinkohal tarkvara vastavust tellija vajadustele ja milliselt käitub tarkvara tellija poolt mittekirjeldatud olukordades. Tarkvara loetakse veatuks, kui see rahuldab tellija poolt kirjeldatud vajadusi. Kui tarkvara töötab ootuspäraselt ka olukordades, mida tellija ei ole kirjeldanud, siis loetakse tarkvaratoode viimistletuks (ik *robust*). Muutmise

Muutmise võimalikkus on tänapäeva tarkvaralahenduste juures samuti väga tähtis, sest iga olemasolev tarkvaratoode võib olla aluseks mõnele uuele tootele. Seega on väga oluline mõelda, mida on vaja teha loodava tarkvara uuesti kasutamiseks sarnastes toodetes. Samuti on muutmise võimalikkus oluline, kui tarkvara tootes leitakse vigu või mittevastavusi vajadustele.

Arusaadavuse vajalikkus on tulenev eelkõige sellest, et tarkvara teostatakse tänapäeval meeskonnatööna ja lahendus peab olema arusaadav kõigile tarkvaratoote teostamise, testimisega ja hooldamisega seotud inimestele ning seda ka hiljem, pärast tarkvara valmimist.

http://www.cs.tlu.ee/~inga/SE\_materjal/Tarkvaratehnika\_sissejuhatus\_2008.pdf

http://www.cs.tlu.ee/~inga/SE\_materjal/Tarkvaratehnika\_etapid\_2008.pdf

 3.1.1 Tunneb üldjoontes põhilisi programmide loomise meetodeid nagu objektorienteeritud disain, ülalt alla projekteerimine, struktuurne programmeerimine

Tarkvara disaini etapis tuleb otsustada ka see, kui palju ja millisteks alamosadeks loodav tarkvaralahendus jagatakse. Alamosadeks jaotamisel tuleb eriti arvestada uuestikasutatavuse nõudega- liig suurte alamosade korral võib nende uuestikasutamine osutuda keeruliseks, liig väikeste osade korral võivad esmased arenduskulud muutuda ebaotstarbekalt suureks.  Täna on väga vähe lahendusi, kus on võimalik monoliitne lahendus. Tarkvara lahenduse jagamisel alamosadeks on kaks lähenemist: ülalt-alla (ik *top-down*) ja alt-üles (ik *bottom-up*).

Ülalt-alla lähenemisel jagatakse lähteülesanne alamülesanneteks, mis võivad igaüks koosneda omakorda alamülesannetest, seejärel jagatakse saadud alamülesanded omakorda alamülesanneteks kuni lähteülesanne koosneb lihtsatest ja üheselt mõistetavatest alamülesannetest. Sellise lähenemise puuduseks on see, et suurtel lahendustel on seda ebaotstarbekas kasutada, sest lähteülesanne jagatakse sellisel juhul väga paljudeks alamülesanneteks, kusjuures ilmselt esineb olukordi, kus ühte ja sama probleemi lahendatakse korduvalt (sest mitmed alamülesanded nõuavad sarnaste probleemide lahendamist). Lisaks eelnevale peab tarkvaradisainer hakkama üsna varakult mõtlema konkreetsete algoritmide peale, millega  etteantud probleemid lahendatavad oleks.

Alt-üles lähenemise korral jagatakse lähteülesanne alammooduliteks, milliseid käsitletakse „mustade kastidena”. Seejärel kirjeldatakse mida iga alammoodul tegema peab ja milliste teiste moodulitega peab see infot vahetama. Nii on võimalik suur lähteülesanne jagada iseseisvateks alamülesanneteks, mida on võimalik eraldi arendama hakata. Alt-üles lähenemisel on lihtne leida olemasolevatest lahendustest korduvkasutatavaid mooduleid, mida saab uue lahenduse koostamisel kasutada, mis omakorda kiirendab ja lihtsustab oluliselt uue lahenduse väljatöötamist. Sellise lahenduse suurimaks probleemiks on tihti see, et jäetakse täpselt kirjeldamata moodulitevaheline infovahetus ja selle korraldus.

Tihti on kasulik lähteülesanne jagada „alt-üles” lähenemise abil alammooduliteks ja need omakorda „ülalt-alla” lähenemise abil lõplikult lahendada.

## Tarkvaraarenduse meetodid

Tarkvaraarenduse jaoks on kasutusel mitmeid projekteerimismeetodeid, peamised kasutatavad meetodid on:  struktureeritud ja objekt-orienteeritud projekteerimine.

Struktuurprojekteerimise korral valmib terve programm korraga, projekteerimise etapid (analüüs, disain jne) on selgelt piiritletud ja järgmine etapp ei käivitu enne, kui eelmine on lõppenud. Sellisel projekteerimisel on mitmeid tõsiseid puuduseid: kui disainimisel tehti vigu, siis ilmneb see tihti alles arenduse lõppfaasis, lahenduse korduvkasutus on komplitseeritud, lahendus koosneb harva moodulitest ja funktsionaalsuse lisamine lahendusele on keeruline. Nende puuduste kõrvaldamiseks kasutatakse tihti prototüüpimist:  lahendusele tehakse kiiranalüüs, valmistatakse piiratud funktsionaalsusega prototüüp, korraldatakse testimine tellija juures ja edasisel lahenduse väljatöötamisel lähtutakse ka prototüübi tagasidest.

Lisalugemist: [wikipedia.org](http://en.wikipedia.org/wiki/Unified_Modeling_Language)

Objekt-orienteeritud (OO) projekteerimise korral lähtutakse sellest, et arendus hakkab toimuma objekt-orienteeritud arendusvahenditega. Seega teostatakse analüüsi ja disainimise etapid vahenditega, mis toetavad hilisemat lahenduse teostamist objekt-orienteeritud vahenditega. Suured lahendused jagatakse väiksemateks eraldiseisvateks lahendusteks, mida on võimalik arendada iseseisvalt. Erinevalt struktuurprojekteerimisest on tellija kaasatud tihedalt arendusetappidesse. Lahendus ei valmi tervikuna korraga, vaid arendus toimub tsüklitena, kus iga tsükliga lisatakse lahendusele täiendavaid funktsionaalsus, sellist arendust nimetatakse iteratiivseks arendusprotsessiks.

Lisalugemist: [wikipedia.org](http://en.wikipedia.org/wiki/Object_oriented)da

OO projekteerimise korral kasutatakse analüüsi ja disaini etapis lahenduse kirjeldamisel UML (Unified Modeling Language) vahendeid.

Lisalugemist: [wikipedia.org](http://en.wikipedia.org/wiki/Unified_Modeling_Language)

 3.1.2 Kirjeldab abstrahheerimist probleemide lahendamise ja tarkvara loomise meetodina.

## Abstraktsioon

**Tarkvara loomisel on oluline osa abstraktsioonil. Selle all mõeldakse lähenemisviisi, kus mingi nähtuse analüüsimisel jäetakse mängust välja selle ebaolulised küljed, mis ei mängi rolli antud probleemi lahendamise kontekstis.**

**Selles mõttes on abstraktsioon infotehnoloogias sarnane abstraktsiooniga matemaatikas, kus matemaatiliste mõistete (sirge, arv, funktsioon jne.) defineerimisel “jäetakse alles” üksnes need reaalmaailma objektide-nähtuste omadused, millel on tähtsust vastava matemaatilise teooria ülesehitamisel.**

**Näiteks võib tuua abstraktse mõiste arv, millel on tähendus nii matemaatikas kui programmeerimiskeeltes. Arvu kujutamine arvutis sõltub nii tark- kui riistvarast, kui see ei muuda selle mõiste sisu.**

**Üldistades võib abstraktsioonid programmeerimiskeeltes jagada kahte gruppi:**

1. **juhtimisabstraktsioon (ingl. k. *control abstraction*);**
2. **andmeabstraktsioon (ingl. l. *data abstraction*)**

**Struktuurprogrammeerimise korral mõistetakse juhtimisabstraktsiooni all süstemaatilist alammoodulite ning juhtimisvoo käskude (iteratsioon, valik, jne.) kasutamist. Andmeabstraktsiooni all mõeldakse reaalmaailma andmete adekvaatset peegeldamist programmeerimiskeele andmestruktuurides (vektorid, kirjed jne).**

**Objektorienteeritud programmeerimiskeeltes on juhtimis- ja andmeabstraktsioonid ühildatud.**

 3.1.3 Tunneb spetsiifilisi eripärasid vanade rakenduste juures nagu keerukas struktuur, vilets dokumentatsioon, vananenud riist- või tarkvara, ärikriitilisus.

## Pärandsüsteemid

Pärandsüsteemi (ingl k *legacy system*) all mõeldakse vananenud, varem kasutusel olnud rakendusprogrammi või riistvaraseadet, mida ka tänapäeval kasutatakse.

Pärandsüsteemidega seonduvad järgnevad võimalikud probleemid:

1. sageli töötavad nad vananenud riistvara peal, mille tööshoidmine on kulukas;
2. organisatsioonis puudub kompetents süsteemi arendamiseks ja haldamiseks (näiteks võivad selle loonud programmeerijad olla organisatsioonist lahkunud);
3. uute süsteemide kasutuselevõtul tuleb hoolitseda „tagasiühilduvuse” (ingl. k. *backward compatibility*) eest.

 3.1.4 Eristab avatud ning suletud koodiga tarkvaraarendust

 3.1.5 Tunneb üldjoontes erinevaid litsensivajadusi kinnise tarkvara, avatud koodiga tarkvara, vaba tarkvara ja vabavara juures

<http://www.slideshare.net/hanspoldoja/2010-1206autorioigus>

http://akadeemia.kakupesa.net/ITSPEA/loengud/loeng6

3.2 Algoritmid ja andmestruktuurid

 3.2.1 Kirjeldab liht- ja struktuuriandmetüüpe. Tunneb mitmesuguseid andmestruktuure näiteks kirjeid, massiive, ahelaid.

## Muutujad

Programmeerimise käigus on reeglina vaja hoida meeles mingit hulka andmeid (vahetulemused, toimunud sündmused, sisendväärtused, väljundväärtused jne). Neid väärtused tuleb hoida mälus. Selleks defineeritakse koht mälus, mida kasutatakse andmete hoidmiseks ning seda defineeritud kohta nimetatakse muutujaks. Kuna andmed, mida hoitakse, võivad olla väga erinevad, siis öeldakse ka muutuja defineerimisel, mis tüüpi andmeid selles muutujas hoidma hakatakse (muutujatüüp).

Lihttüübid

Lihttüüpi muutujas on märksõna all peidus üks väärtus (enamasti arvuna loetav) ning ta on sealt otse kättesaadav. Tuntumad lihttüübid on: märgiga täisarv, märgita täisarv, reaalarv (komakohaga), tähemärk, tõeväärtus. Keeleti võivad nad mõnevõrra erineda.

Struktuurtüübid

Struktuurtüüpides koondatakse ühe märksõna alla mitu kokkukuuluvat väärtust nagu näiteks punkti koordinaadid või inimese ees- ja perekonnanimi. Selliselt on andmekogumit kergem korraga edasi anda. Samas andmeid kasutada või muuta tuleb enamasti ikka struktuuri seest ükshaaval.

Massiivid

Massiiv on ühetüübiliste muutujate kogum, millistel on üks nimi ja mis on üksteisest eristatavad indeksi poolest. Massiivid lihtsustavad oluliselt ühetüübiliste andmete töötlemist. Lihtsustamine tuleneb sellest, et programmi täitmise käigus on võimalik indeksit lihtsalt muuta ja seega on lihtsam pöörduda vajaliku muutuja poole.

Massiivid võivad olla ühemõõtmelised (jada, rida), kahemõõtmelised (tabel, maatriks), kolmemõõtmelised (kuup), jne.

Näide (C#)

int**[] mass =** newint**[10];**

**mass[0]=1;**

**mass[1]=1;**

for **(**int **i = 2; i <= 9; i++)**

**mass[i] = mass[i - 1] + mass[i - 2];**

for **(**int **i = 0; i <= 9; i++)**

Console.Write(mass[i]);

|  |
| --- |
|   |

123456789

|  |
| --- |
| 1 |

|  |
| --- |
| 2 |

|  |
| --- |
| 3 |

|  |
| --- |
| 5 |

|  |
| --- |
| 8 |

|  |
| --- |
| 13 |

|  |
| --- |
| 21 |

|  |
| --- |
| 34  |

|  |
| --- |
| 55 |

Selles näites luuakse ühemõõtmeline massiiv mass, millel on kümme liiget, antakse kahele esimesele liikmele väärtus „1” ja täidetakse ülejäänud liikmed kahe eelneva liikme summaga.



Seejärel trükitakse kõik väärtused järjest (alustades nullindast liikmest ja lõpetades üheksandaga) konsooliaknasse.

http://enos.itcollege.ee/~jpoial/java/i200loeng4.html

## Kirjed

Hoidmaks erinevate tüüpi andmeid, mis moodustavad koos mingi seostatud kogumi kasutatakse kirjeid. Näiteks moodustub kirje inimene järgmistest andmetest: eesnimi (tekst), perenimi (tekst), sugu (tõeväärtus, 0- naine, 1 -mees), kaal (reaalarv). Sellised andmed moodustavad ühtse terviku ühe inimese kirjeldamiseks, kuid eraldiseisvana on väga erinevat tüüpi.

Näide (C#)

structinimene **{**

publicstring **eesnimi;**

publicstring **perenimi;**

publicbool **sex;**

publicfloat **weight;**

**}**

Sellise kirjega saame luua uue muutuja kasutaja ja anda kasutajale, eesnime, perenime, soo ja kaalu:

inimene **kasutaja;**

**kasutaja.eesnimi =** "Jaan"**;**

**kasutaja.perenimi =** "Mets"**;**

**kasutaja.sex = 1;**

**kasutaja.weight = 80.0;**

## Ahelloendid ja puud

Kaasajal kasutatakse tihti andmete hoidmiseks loendeid (*list*). Kui iga listi liige viitab järgmisele, siis on tegemist ahelloendiga, ahelloendi lõppu märgib tühiliige (*null)*. Ahelloend, kus iga liige viitab järgmisele nimetatakse ühesuunaliseks loendiks, ahelloend, kus iga liige viitab eelmisele ja järgmisele nimetatakse kahesuunaliseks loendiks. Ahelloend, kus puudub esimene ja viimane liige ning iga liige viitab järgmisele nimetatakse ringloendiks. Ahelloendi pikkus on loendi liikmete arv. Loendi esimene liige on pea (*head*) ja ühejäänud liikmed saba (*tail*).

Pinu (*stack*) on ahelloend, kus viimasena lisatud liige loetakse välja esimesena ( LIFO – *Last In First Out*).

Järjekord (*queue*) on ahelloend, kus esimesena lisatud liige loetakse välja esimesena ( FIFO – *First In First Out*)

Lisalugemist: [wikipedia.org](http://en.wikipedia.org/wiki/Linked_list)

Puu on andmestruktuur, kus andmed on paigutatud puukujuliselt, koosneb tippudest (*nodes*) ja kaartest (*edges*), mis ühendab tippe (viited). Tipud, mis on ühendatud kaarega üleval asuva tipu külge on lapsed (*childs*) ja üleval asuv tipp on sellisel juhul vanem (*parent*). Kõige ülemine tipp on juur (*root*). Tippu, millel ei ole lapsi, nimetatakse leheks (*leaf*).



Liikudes tipust vanemasse, sealt vanemasse jne jõuame Juurde. Esivanemad on kõik tipud mis jäävad vaadeldava tipu ja juure vahepeale. Puu kõrgus (*tree height*) on pikim tee lehest juureni.

Järjestatud puu korral on defineeritud juur ja otse juurega ühendatud tipud on esimese taseme tipud (*first level nodes*, juure lapsed), esimese taseme tippudega otse ühendatud tipud on teise taseme tipud (esimese taseme tippude lapsed) jne ning laste järjekord vasakut paremale on oluline.

Lisalugemist: [wikipedia.org](http://en.wikipedia.org/wiki/Tree_%28data_structure%29)

Kahendpuu on selline puu, kus iga vanema võib olla mitteühtegi, üks või kaks last ja laste järjekord on oluline.

Kahend-otsingupuu (*binary search tree*) on kahendpuu, mis on järjestatud. Tipust vasakul on alati väiksem suurus ja tipust paremal suurem suurus.



Sellisest puust otsides võrreldakse otsitavat väärtust juurega, kui otsitav väärtus võrdub juure väärtusega, siis väärtus eksisteerib, kui aga juure väärtus ei võrdu otsitavaga, siis võrreldakse väärtust edasi, vastavalt kas vasaku või parema tippude hulgast kuni jõutakse leheni. Kui otsitav väärtus on võrdne mõne tipu väärtusega, siis on otsitav element olemas, kui aga ei leidu võrdset väärtust, siis otsitavat elementi ei ole. Selline otsimise viis on kordi kiirem kui oleks näiteks ahelloendi või massiivi läbivaatamine.

Lisalugemist: [wikipeida.org](http://en.wikipedia.org/wiki/Binary_search_tree)

B-puu (B tree )on otsingupuu, milles iga tipu tütarde arv asub vahemikus (t-1) kuni (2t-1) , kus t on suvaline konstant

Lisalugemist: [wikipedia.org](http://en.wikipedia.org/wiki/B-tree)

B\*-puu  on B-puu, kus tippude täituvus hoitakse 2/3 juures, täites kaks tütartippu võtmete ümberjaotamise teel ja tükeldades nad seejärel kolmeks tipuks.

 3.2.2 Suudab hinnata levinumate sortimis- ja otsimisalgoritmide ning andmestruktuuride sobivust olukordades.

http://www.cs.tlu.ee/~inga/alg\_andm/complexity\_2010.pdf

3.3 Programmeerimiskeeled

 3.3.1 Kirjeldab programmeerimiskeelte tüüpe - funktsionaalsed, protseduursed, objektorienteeritud keeled. Tunneb nende põhilisi väärtusi ning suudab tüüpe eristada.

## Programmeerimiskeelte põlvkonnad

Programmeerimise vajadus tekkis juba enne programmeeritavaid arvuteid. Nii on teada näiteks 18. sajandist kangasteljed, mis olid programmeeritavad puuliistude, kuhu oli tehtud õigetesse kohtadesse augud, abil.

Programmeerimise arengule andis hoo John von Neumanni poolt 1945. aastal avaldatud idee, kus ta kirjeldas arvutit, kus mälus hoitakse lisaks andmetele ka käske.

Esimese põlvkonna programmeerimiskeelteks nimetatakse masinkoodi keeli. Masinkood koosneb käskudest, mida arvuti (protsessor) täita oskab ( ning nende käskude juurde kuuluvatest andmetest). Masinkoodis programmeerimisel pidi programmeerija kirjutama oma programmi kahendkoodis, nii et see oleks protsessorile arvusaadav ja täidetav. Sisuliselt nõuab selline programmeerimine häid teadmisi ja arusaamist riistvarast, sest programmeerimise käigus peab teadma mida protsessor teha oskab, kus asuvad sisend-väljundseadmed (I/O- Input-Output) ning kuidas nendega suhelda tuleb ja kui kaua aega mingi tegevuse tarvis kulub. Niisiis on masinkood väga tugevasti seotud riistavaraga, mille peal vastav programm tööle hakkab. Masinkood ei ole tänapäeva arvutitest kuhugi kadunud, kogu tegevus madaltasemel (riistvara tasemel) toimub endiselt masinkoodis ehk siis ükskõik millises programmeerimiskeeles kirjutatud programm teisendatakse lõpuks ikkagi riistavarale arusaadavasse masinkoodi.

Teise põlvkonna programmeerimiskeeled on *assembler-*keeled. Kui masinkoodis programmeerimisel toimus kogu programmeerimine kahendkoodis ja selle lugemine ning silumine oli tänu sellele väga keeruline, siis *assembler*-keeles programmeerimisel on käsud inimesele sobivamal kujul. Kogu programmeerimine on üsna sarnane masinkoodis programmeerimisele, sest käsud on samad, mis masinkoodis (ainult teisel kujul). *Assembler-*keeles kirjutatud programm näeb välja tavaliselt midagi sarnast:

MOV AL,19

ADD AL, 4

OUT 2

Antud koodijupp paneb AL registrisse väärtuse 19 (tavaliselt on väärtused antud 16nd arvusüsteemi arvudena),  liidab AL registris olevale väärtusele 4 ning saadab selle seejärel väljundliidesele 2. Selliselt kirjutatud programm tõlgitakse *assembleri* masinkoodi ning seejärel on võimalik protsessor seda masinkoodi täitma panna.

Masinkoodi- ja *assembler*-keeled on madaltaseme programmeerimiskeeled.

Lisalugemist: [wikipedia.org](http://en.wikipedia.org/wiki/Assembly_language)

Kolmanda põlvkonna programmeerimiskeelteks nimetatakse juba kõrgtaseme programmeerimiskeeli. Sellised programmeerimiskeeled ei ole enam nii väga seotud riistvaraga. See tähendab, et programmeerija ei pea enam teadma väga täpselt riistvara ehitust ja omapära, vaid saab kirjutada programmi riistvarast võrdlemisi sõltumatult ning hiljem tõlgitakse kirjutatud programm mitmesuguste erinevate vahendite abil riistvarale arusaadavale kujule. Kuidas seda täpselt tehakse selgitame hiljem.

Kolmanda põlvkonna keelte hulka kuulub enamik tuntumaid ja kasutatavaid keeli, näiteks:

FORTRAN (*The IBM Mathematical FORmula TRANslating System*) – 1950ndatel aastatel loodud arvutuste ja teaduslikeks eesmärkideks loodud programmeerimiskeel

COBOL (*COmmon Business Oriented Language*) – 1959. aastal loodud objekt-orienteeritud programmeerimiskeel, peamiselt ärivajadusele suunatud rakenduste kirjutamiseks.

BASIC (*Beginner's All-purpose Symbolic Instruction Code*) – 1963. aastal väljatöötatud programmeerimiskeel, mis algselt loodi selleks, et inseneride oleks võimalik teostada arvutitel erinevaid simulatsioone.

Pascal – 1970ndatel loodud programmeerimiskeel, mis loodi programmeerimise õpetamiseks.

C – nimi tuleneb sellest, et keel baseerus paljuski B nimelisel programmeerimiskeelel, loodi operatsioonisüsteemide kirjutamiseks (paljud Unixi –laadsed operatsioonisüsteemid on enamikus kirjutatud C keeles), pikka aega üks populaarsemaid programmeerimiskeeli.

C++ - objektorienteeritud C.

Java – C++ põhjal arendatud programmeerimiskeel

Visual Basic, Delphi, Python, C# jne on kõik kolmanda põlvkonna programmeerimiskeeled. Paljud kolmanda põlvkonna programmeerimiskeeled on nooremad (uuemad) kui mitmed neljanda ja viienda põlvkonna programmeerimiskeeled.

Neljanda põlvkonna programmeerimiskeeled on loodud eesmärgiga lihtsustada nende õppimist ja kasutamist. Neljanda põlvkonna programmeerimiskeeled on tavaliselt mitte-protseduurilised ja ühe rakenduse kesksed. Üheks neljanda põlvkonna programmeerimiskeeleks on näiteks SQL (*Structured Query Language*).

Viienda põlvkonna programmeerimiskeeled on loodud tehisintelligentsete süsteemide loomiseks ja tehisintelligentsusega seotud probleemide lahendamiseks.

Lisalugemist: [wikipedia.org](http://en.wikipedia.org/wiki/Alphabetical_list_of_programming_languages)

## Programmeerimiskeelte põhitüübid

Erinevalt programmeerimiskeelte põlvkondadest kirjeldavad põhitüübid kuidas on võimalik mingite keelte abil programmeerida. Peamised põhitüübid on: protseduurilised, funktsionaalsed ja objektorienteeritud programmeerimiskeeled.

Protseduurilistes programmeerimiskeeltes kirjeldatakse programmeerimiskeeltes tegevused ja nende täitmise kord ja jagatakse need tegevused gruppidesse (alamprogrammideks). Protseduuridest kujunevad omamoodi koodi struktuurid, mida on võimalik korduvalt kasutada.

Funktsionaalsetes programmeerimiskeeltes kirjeldatakse kogu lahendus funktsioonide abil.

Objektorienteeritud programmeerimiskeeltes teostatakse lahendus klassides (*class*) kirjeldatud funktsioonide ja andmestruktuuride abil. Igast klassist on võimalik moodustada objekte, millel on mingi hulk omadusi ja/või meetodeid.

Omadused on väärtused, mida objekt suudab hoida ja mis võivad mõjutada objekti käitumist. Näiteks klassi „konsooliaken” põhjal saab moodustada objekti „konsool1” mis kasutajale paistab lihtsalt ühe konsooliaknana. Sellel objektil on mitmeid omadusi (nähtav, mittenähtav, laius, kõrgus, teksti värv konsooliaknas, taustavärv jne), neid omadusi muutes muutub konkreetsel juhul objekti välimus.

Sama näite puhul on sellel objektil ka mitmeid meetodeid, nii saab vastava meetodi poole pöördudes kirjutada konsooliaknasse mingi teksti, lugeda muutujasse kasutaja poolt sisestatud teksti jne.

Teiseks näiteks võib tuua tekstimuutuja, luues tekstimuutuja klassi baasil objekti „MingiTekst”, loome ühe tekstimuutuja, mille peamine omadus on hoitav tekstiline väärtus, kuid tegelikult on omadusi sellisel objektil rohkem (näiteks hoitava testilise väärtuse pikkus). Ning samuti on tekstimuutjas ilmselt kirjeldatud mingi hulk meetodeid (muuda hoitav väärtus väiketähtedeks, suurtähtedeks, eemalda teatud sümbolid jne).

Selliselt klasse kirjeldades ning objektidega manipuleerides on võimalik koostada väga keerulisi programme ja täita mitmesuguseid tegevusi.

## Interpreteeritavad ja kompileeritavad keeled

Enne seda kui arvuti saab kõrgkeeles kirjutatud programmi täita, tuleb see „tõlkida” talle arusaadavasse keelde e. masinkoodi. Seda tõlkimisprotsessi nimetatakse transleerimiseks ja tõlkeprogrammi translaatoriks. Translaatorid jagunevad kahte klassi: kompilaatorid ja interpretaatorid.

Kompileerimine seisneb selles, et masinkoodis programm (nimetatakse kompilaatoriks) teisendab mingis programmeerimiskeeles kirjutatud programmi masinkoodi. Seejärel täidetakse saadud masinkoodis programm. Näidetena kompileeritavatest keeltest võib nimetada C, Fortran, Pascal..

Interpreteerimine seisneb selles, et masinkoodis programm (interpretaator) loeb sisemällu programmifaili ja asub seda rida-realt täitma. Näitena võib tuua vana BASIC-keele.

Programmi interpreteerimine on ca 10-200 korda aeglasem, kui kompileeritud koodi täitmine. Vastukaaluks on interpreteeritava programmi silumine (vigade eemaldamine programmist) reeglina lihtsam kui transleeritava programmi korral.

Põhimõtteliselt saab igas keeles kirjutatud programme nii interpreteeritult täita kui kompileerida.

 3.3.2 Kirjeldab funktsioonide ja protseduuride kasutust ning eristab väärtuse ja viite kaudu parameetrite edastamist.

## Protseduurid ja funktsioonid

Programmi struktuuri lihtsustamiseks on otstarbekas jagada suurem programmeerimisülesanne väiksemateks alamülesanneteks. Alamülesannete lahendamine jäetakse väiksemate terviklike alammoodulite – protseduuride ja funktsioonide ülesandeks. Selline lähenemisviis lihtsustab programmi silumist ja haldamist. Alammoodulite korduvkasutuse võimaluse tõttu vähendab selline meetod ka programmeerimistööd.

Protseduurid ja funktsioonid erinevad nende kasutusviisi poolest. Funktsiooni kasutatakse funktsiooniviite abil, protseduuri aga protseduurilause abil. Erinevalt protseduurist tagastab funktsioon alati väärtuse.

Seos alammoodulite ja neid väljakutsuva mooduli vahel organiseeritakse mooduli parameetrite abil. Parameetrite edastamisviisi poolest jagatakse nad kahte gruppi:

a) aadresskutse (ingl. k. *call by reference*) puhul saab kutsutav moodul muuta kutsuva mooduli salvestatud parameetrite väärtusi. Tehniliselt on selline kutse realiseeritud nii, et kutsuv moodul annab kutsutavale moodulile talle üleantavate parameetrite aadressid.

**b)** kutse väärtusega e. väärtuskutse (ingl. k. *call by value*)- kutsuv moodul annab kutsutavale moodulile talle üleantavate parameetrite tegelikud väärtused. Väärtuskutses ei saa kutsutav moodul muuta kutsuva mooduli salvestatud või tema jaoks salvestatud parameetrite väärtusi.

**Näide**

Kui meil on defineeritud kahe parameetriga protseduur *myproc* ja ühe parameetriga funktsioon *myfunc*, siis on võimalik neid käivitada nii:

myproc (a, b);

d = myfunc( e/f );

Esimeses neist lausetest täidetakse protseduur *myproc*, teises ag arvutatakse funktsiooni *myfunc* väärtus ja salvestatakse see muutujasse d.

Protseduure ja funktsioone nimetatakse sageli ka alamprogrammideks (ingl. k. *routine*). Programmi, millest pöördutakse alamprogrammide poole, mis aga ise alamprogramm ei ole, nimetatakse põhiprogrammiks (ingl. k. *main program*).

http://enos.itcollege.ee/~jpoial/java/i200loeng2.html

 3.3.3 Teab, mida tähendab programmeerimiskeele süntaks ning oskab seda tähtsustada.

3.4 OOP

<http://enos.itcollege.ee/~jpoial/java/i200loeng3.html>

<http://minitorn.tlu.ee/~jaagup/kool/java/loeng/pohikursus03/objektid.htm>

<http://www.tud.ttu.ee/material/vladimir/PROGRAMMEERIMINE/Program_II_11/lisamat/Mikli_AB_konspekt/OODisain.pdf>

<http://nrg.tartu.ee/algkursus/Teema12.htm>

 3.4.1 Kirjeldab objektorienteeritud disaini põhimõtteid

 3.4.2 Kirjeldab objektorienteeritud programmeermise põhimõtteid

 3.4.2 Kirjeldab, mis on klass, objekt, meetod ning nende omavahelisi suhteid.

 3.4.4 Kirjeldab päriluse ülesehitust ning selle tarvilikkust programmeerijale.

 3.4.5 Kirjeldab abstrahheerimist ja kapseldamist (teabe peitmine).

 3.4.6 Kirjeldab, kuidas polümorfism toetab korduvkasutatavate komponentide abil tarkvara efektiivsemat loomist.

3.5 Põhikonstruktsioonid

Iga programm koosneb tavaliselt järgmisest osadest:

* Deklaratsioonid
* Aritmeetika- ja loogikatehted
* Juhtkäsud
* Sisend-väljund (I/O) käsud

Deklaratsioonidega seotakse mingi sümbolitest koosnev nimi mõne andmeobjekti või programmi osaga. Aritmeetika ja loogikatehete abil muudetakse programmi käiku ja andmeobjektide väärtusi. Juhtkäsud määravad programmi täitmise korra (erinevate programmiosade täitmise, vahelejätmise, korduvtäitmise jne). Sisend-väljund käsud lubavad programmil suhelda sisend-väljund seadmete või teiste programmidega.

## Deklaratsioonid

Deklaratsioonidega seotakse mingi sümbolitest koosnev nimi mõne andmeobjekti või programmi osaga.

Deklareerimise käigus seotakse tihti andmeobjektiga mingi koht mälus, kus siis seda konkreetset andmeobjekti iseloomustavaid väärtuseid hoitakse.

Kõige levinumad deklaratsioonid on muutujate deklaratsioonid, näiteks kasutades EPL (*EUCIP Programming Language, baseerub C-keelel*) pseudokeelt võime deklareerida muutujaid järgmiselt:

*char alfa;*

*int beeta;*

*float gamma;*

Nii oleme me deklareerinud kolm muutujat tähemärgi tüüpi muutuja alfa, täisarv tüüpi muutuja beeta ja reaalarv tüüpi muutuja gamma.

 3.5.1 Tunneb sisend-väljundlauseid.

## Sisend-väljund

Sisend-väljund käsud on mõeldud suhtlemaks välismaailmaga läbi sisend-väljund seadmete. Sisend-väljund käsud võivad erinevates keeltes suurtesti varieeruda: mitmetes keeltes ei ole eraldi sisend-väljund käske (näiteks Motorolla 88k assembler suhtleb sisend-väljund seadmetega nagu mäluga), C’l baseeruvatel keeltel on reeglina eraldi välised funktsioonid, mida sisendi ja väljundiga suhtlemiseks kasutatakse (tihti kasutatakse selleks operatsioonisüsteemi teenuseid ja vahendeid).

EPL keeles on sisend-väljund käskudeks ainult kaks funktsiooni: *printf*() ja *readf*().

Funktsioon *printf*() on kasutatav järgmiselt:

*printf*(„Mingi tekst”, muutuja1[,muutuja2...])

Antud näites on tekstilõik „Mingi tekst” väljundisse kirjutatav tekst ja muutuja1 ning muutuja2 väljundisse kirjutatavad muutujate väärtused. Väljundit nimetatakse siinkohal standardseks väljundiks (stdout – standart output) ja tavaliselt suunatakse enamikus programmeerimiskeeltes see infovoog ekraanile, kuid samas on enamikus programmeerimiskeeltes ka võimalus selle voo ümbersuunamiseks faili, printerisse, teise programmi jne.

Funktsioon *readf*() on kasutatav järgmiselt:

*readf*(muutuja1[,muutuja2...])

Antud näites loetakse sisendvoost väärtused muutujatesse muutuja1, muutuja2 jne. Sisendit nimetatakse siinkohal standardseks siseniks (stdin – standart input) ja tavaliselt moodustub see infovoog klaviatuurilt sisestatud väärtustest, kuid samas on see on võimalik tihti see voog ühendada mõne failiga, teise programmi väljundiga jne.

 3.5.2 Tunneb juhtlauseid.

## Juhtkäsud

Juhtkäsud määravad programmi täitmise korra (erinevate programmiosade täitmise, vahelejätmise, korduvtäitmise jne).

Üldiselt on kolme liiki juhtlaused:

* Tingimusteta hüpped
* Tingimuslaused
* Kordused

Tingimusteta hüpete abil on võimalik „hüpata” koodis ringi ja muuta nii kirjutatud koodi täitmise järjekorda. Selliste juhtkäskude kasutamine raskendab oluliselt koodist arusaamist ja seetõttu soovitatakse sellisest konstruktsioonist hoiduda. Mitmed tänapäevased keeled ei luba tingimusteta hüpete kasutamist (sh EPL).

Tingimustega juhtlausete korral kasutatakse tingimuse realiseerimiseks kasutatakse võrdlusoperaatoreid:

==          võrdus, tulemus on tõene, kui võrreldavad väärtused on võrdsed

!=           mittevõrdsus, tulemus on tõene, kui võrreldavad väärtused ei ole võrdsed

>           suurem kui, tulemus on tõene kui operaatorist vasakul olev väärtus on suurem kui paremal olev väärtus

<          väiksem kui, tulemus on tõene kui operaatorist vasakul olev väärtus on väiksem kui paremal olev väärtus

>=        suurem või võrdne, tulemus on tõene kui operaatorist vasakul olev väärtus on suurem kui paremal olev väärtus ning ka siis, kui vastavad väärused on võrdsed

<=        väiksem või võrdne, tulemus on tõene kui operaatorist vasakul olev väärtus on väiksem kui paremal olev väärtus ning ka siis, kui vastavad väärused on võrdsed

Tingimusi on võimalik omavahel kombineerida kasutades järgmisi loogikatehteid

&&         ja, tingimus on tõene kui mõlemad tingimused on tõesed

||           või, tingimus on tõene, kui üks tingimustest on tõene

Nii võiks näiteks tingimuse

                a>=b

kirja panna ka kui

(a>b) || (a=b)

Tuntum juhtkäsk on kindlasti *if*:  if lause võimaldab täita erinevat koodi sõltuvalt  tingimuse tulemusest:

if(tingimus)

                kood mis täidetakse, kui tingimuse tulemus on tõene

else

                kood mis täidetakse, kui tingimus ei ole tõene

*else*-osa võib ka puududa.

*if*-käsu näide:

*if(a>3)*

*b=5;*

 Kui a on suurem kolmest, siis b võrdustakse viiega.

*if*-käsu näide:

*if(a>3)*

*b=5;*

*else*

*c=6;*

Kui a on suurem kolmest, siis b võrdustakse viiega ja kui a ei ole suurem kolmest, siis c omistatakse väärtus kuus.

*If*-käsu keerulisemad versioonid on *case* ja *switc*h, need juhtkäsud võimaldavad rohkemaid jagunemisi vastavalt tingimustele, kui kuna EPL keel neid ei sisalda, siis siinkohal me neid ei käsitle.

Kordused võimaldavad mingisugust koodi osa vastavalt etteantud tingimusele korduvalt täita. Kordused jagunevad: eelkontrolliga ja järelkontrolliga ning määratud pikkusega ja määramata pikkusega kordusteks.

*while*-kordus on eelkontrolliga kordus, see tähendab, et korduse jätkamiseks vajalikku tingimust kontrollitakse enne korduse täitmist.

*while(tingimus)*

*kood, mida täidetakse kuni tingimus on tõene*

Näiteks:

*int a=0;*

*int b=0;*

*while(a<10)*

*{*

*a++;*

*b+=a;*

*}*

Kordust täidetakse kuni a on väiksem kümnest, kusjuures iga korduse täitmise jooksul liidetakse muutuja a väärtus muutuja b väärtusele juurde, järelikult b=1+2+3+4+5+6+7+8+9+10=55.

*do-while*-kordus on järelkontrolliga kordus, kordus, see tähendab, et korduse jätkamiseks vajalikku tingimust kontrollitakse pärast korduse täitmist.

*do*

*kood, mida täidetakse kuni tingimus on tõene*

*while(tingimus)*

Näiteks:

*int a=0;*

*int b=0;*

*do*

*{*

*a++;*

*b+=a;*

*}*

*while(a<10)*

Kordust täidetakse kuni a on väiksem kümnest, kusjuures iga korduse täitmise jooksul liidetakse muutuja a väärtus muutuja b väärtusele juurde, järelikult b=1+2+3+4+5+6+7+8+9+10=55.

Nüüd võis jääda mulje, et ei ole vahet, kas kasutada eel- või järelkontrolliga kordust, tegelikult on aga nende vahe selles, et järelkontrolliga korduse sees olevat koodi täidetakse, vähemalt ühe korra, kuid võib juhtuda, et eelkontrolliga kordust ei täideta ühtegi korda.

Näiteks:

*int a=10;*

*int b=0;*

*while(a<10)*

*{*

*a++;*

*b+=a;*

*}*

Tulemuseks on, et a=10 ja b=0, sest korduse sees olevat tegevust ei täidetud ühtegi korda, tingimus oli kohe alguses väär ja tingimust kontrolliti enne korduse sees oleva tegevuse tegemist.

*int a=10;*

*int b=0;*

*do*

*{*

*a++;*

*b+=a;*

*}*

*while(a<10)*

Tulemuseks on, et a=11 ja b=11, sest korduse sees olevat tegevust tehti üks kord, tingimust kontrolliti alles pärast seda.

*for*-kordus on aga eelkontrolliga ning määratud pikkusega kordus.

*for(muutuja, tingimus, muutuja muutmise kord)*

*kood, mida täidetakse, kuni tingimus on tõene*

*for*-kordus erineb eelpool käsitletud kordustest eelkõige selle poolest, et korduse jätkamise tingimusega seotud muutuja väärtuse muutmine kirjeldatakse korduse sees.

Näiteks:

*int b;*

*for(int a=0; i<10; i++)*

*b+=a;*

Tulemus on sarnane do kordusele, muutuja a väärtus pannakse alguses võrduma nulliga, seejärel kontrollitakse korduse täitmise tingimust ja kui see on tõene, siis liidetakse muutujale b muutuja a väärtus, seejärel muudetakse muutuja a väärtust vastavalt etteantud reeglile (antud juhul a++, ehk suurendatakse a väärtust ühe võrra), kontrollitakse jätkamise tingimust, ning kui see on tõene, siis tehakse uuesti korduse sees olev tegevus jne. Tulemuseks on, et a=10 ja b=0+1+2+3+4+5+6+7+8+9=45.

 3.5.3 Tunneb aritmeetilisi ja tõeväärtusavaldisi.

## Aritmeetika- ja loogikatehted

Peamiselt läheb meil vaja viit aritmeetikatehet:

Liitmine: kasutatakse märki „+” ja liidab kokku kaks väärtust, näiteks c=a+b liidab muutujate a ja b väärtused ja paneb muutuja c väärtuse võrduma tehte tulemusega (c=3+2, järelikult c=5).

Lahutamine: kasutatakse märki „-” ja lahutab esimesest väärtusest teise , näiteks c=a-b lahutab muutuja a  väärtusest muutuja b väärtuse ja paneb muutuja c väärtuse võrduma tehte tulemusega(c=3-2, järelikult c=1).

Korrutamine: kasutatakse märki „\*” ja korrutab kaks väärtust, näiteks c=a\*b korrutab muutujate a ja b väärtused ja paneb muutuja c väärtuse võrduma tehte tulemusega(c=3\*2, järelikult c=6).

Jagamine: kasutatakse märki „/” ja jagab ühe väärtuse teisega, näiteks c=a/b jagab muutuja a väärtuse muutuja b väärtusega ja paneb muutuja c väärtuse võrduma tehte tulemusega (c=3/2, järelikult c=1,5 või c=1, oleneb keelest).

Jagamise jääk: kasutatakse märki „%” ja jagab ühe väärtuse teisega, näiteks c=a%b jagab muutuja a väärtuse muutuja b väärtusega ja paneb muutuja c väärtuse võrduma jagamise jäägiga. (c=5%3 järelikult c=2).

EPL keeles ei ole tõeväärtus tüüpi muutujat, seetõttu on loogikatehteid ainult juhtlausetes. Keeltes, kus on tõeväärus tüüpi muutujad on võimalik kasutada ja loogikatehteid, näiteks c= a && b, c=true kui a ja b on võrdsed ja c=false kui a ja b ei ole võrdsed. Loogikatehetest tuleb pikemalt juttu juhtlausete käsitlemisel.

Tehete järjekord on osades programmeerimiskeeltes erinev ja osades samataoline: osad keeled teevad kõigepealt korrutamis- ja jagamistehted ning seejärel liitmis- ja lahutamistehted. Tehteid täidetakse vasakult paremale. Tehete järjekorda on võimalik määrata sulgudega (sulgude sees olevad tehted tehakse kõigepealt). Näiteks c=a+a\*b on erinev tehtest c=(a+a)\*b.

Erinevad programmeerimiskeeled lubavad erinevate aritmeetikaoperaatorite kasutamist:

Näiteks c += a liidab muutuja c väätusele juurde muutuja a väärtuse (ehk c = a + c), a++ on muutuja iteratsioon ehk väärtuse suurendamine ühe võrra (ehk a=a+1) jne.

3.6 Testimine

<http://enos.itcollege.ee/~jpoial/java/>

http://uuslepo.it.da.ut.ee/~at/prog/