TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Informaatika aluste õppetool

Javascripti praktilisus disainimustrite kasutamisel

Magistritöö

Üliõpilane: Peeter Kõbu

Üliõpilaskood: 132676IAPM

Juhendaja: Tarmo Veskioja

Tallinn
2015

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

---------- ----------

(*kuupäev*) (*allkiri*)

Annotatsioon

Antud töö eesmärgiks on uurida erinevate GOF disainimustrite kasutatavust javascriptiga ning uurida, kas kõiki disainimustreid on võimalik realiseerida kasutades javascripti ja kui mitte, siis leida uusi alternatiive. Lisaks on eesmärgiks uurida pakutud lahendusi ja hinnata neid tuginedes GRASP printsiipidele.

Paljudes õppekeskkondades ning raamatutes tutvustatakse erinevaid disainimustreid kasutades klassikalisi OO keeli nagu Java või C#. Javascripti loetakse samuti OO keeleks, kuid sellel puuduvad OOP-le omased tunnused nagu klass või liides, mille tõttu võib tunduda, et disainimustrite kasutamine javascriptiga on keeruline või teatud situatsioonides isegi võimatu. Antud töös on javascript programmeerimiskeele võrdluseks valitud C# programmeerimiskeel, kus erinevate disainimustrite realiseerimisega ei kaasne probleeme.

Töö käigus on võrreldud erinevaid lahendusi GOF disainimustrite realiseerimiseks ning esitatud ka alternatiive, püüdes samal ajal säilitada erinevaid GRASP printsiipe. Lõpptulemusena on välja selgitatud, et javascriptiga on võimalik realiseerida kõiki GOF disainimustreid ning iga mustri realiseerimiseks on mitmeid erinevaid võimalusi. Erinevates mustrites leidub aga sarnast kirjapilti, mis on põhjustatud asjaolust, et javascriptis kirjeldatakse kõike funktsioonide, prototüüpide ja objektide abil.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 78 leheküljel, 5 peatükki, 36 joonist, 3 tabelit.

Abstract

The purpose of this thesis is to investigate the usage of different GOF design patterns with javascript. To explore whether all design patterns can be implemented using javascript, and if not, then find alternatives. Also to explore offered solutions and evaluate them based on the GRASP principles.

Many learning environments and books present different design patterns using classical OO language like Java or C#. Javascript is also considered as an OO language, but it is missing the OOP features such as class or interface, that is why it seems that the use of JavaScript design patterns are difficult or even impossible for some situations. In this thesis C# language, where implementation of different design patterns is not a problem, has been chosen to comparasion with topical javascript language.

During the work has been compared different solutions of implementation of GOF design patterns, and also provided a variety of alternatives in attempt to maintain the GRASP principles. As a result, it has been found that it is possible to realize all JavaScript GOF design patterns, and each pattern has more than one way of implementation. However, implementations of different patterns contain similar structures, but it is because everything in JavaScript is described with functions, prototypes and objects.

The thesis is in estonian and contains 78 pages of text, 5 chapters, 36 figures, 3 tables.

Lühendite ja mõistete sõnastik

GOF	***Gang of Four*** GOF disainimustrid
GRASP	***General Responsibility Assignment Software Patterns*** GRASP printsiibid
OO	***Object oriented*** Objektorienteeritud
OOP	***Object oriented programming*** Objektorienteeritud programmeerimine
UML	***Unified modeling language*** Unifitseeritud modelleerimiskeel
GPS	***Global Positioning System*** Globaalne positsioneerimise süsteem
AJAX	***Asynchronous JavaScript and XML*** Asünkroonne javascript ja xml
XML	***EXtensible Markup Language*** Laiendatav märgistuskeel

Jooniste nimekiri

Tabelite nimekiri

Sisukord

1. Sissejuhatus

Infotehnoloogia kiire arenguga kasvab järjest suurem nõudlus tarkvara järele. Tarkvara on riistvara lahutamatu osa ning suur osa tarkvarast esineb infosüsteemide kujul. Infosüsteemile on antud erinevaid definitsioone, kuid eesmärk on üks ja seesama – info kogumine, töötlemine, salvestamine ja säilitamine. Kiire andmemahtude kasvuga kaasneb ka infosüsteemide kasv, mis nõuab üha suuremat rõhuasetust infosüsteemi töökindlusele ja skaleeruvusele. Infosüsteemidele arendatakse pidevalt juurde uut funktsionaalsust, neid optimeeritakse ning töötatakse välja uusi arendusprotsesse, et luua üha paremaid ja töökindlamaid infosüsteeme.

Tarkvara loomine ei ole ainult infosüsteemide loomine, vaid luuakse tarkvara ka meid ümbritsevadele seadmetele – näiteks telefonid, pesumasinad, autod jms. Samuti luuakse tarkvara ka meelelahutuseks mängude näol. Erinevate süsteemide loomine eeldab hästi struktureeritud ja läbipaistvat koodi. Siinkohal on abiks programmeerimiskeelest sõltumatud erinevad arendusmetoodikad, objekt-orienteeritud lähenemine, arhitektuuri ja disaini mustrid ja paljud muud head tavad.

Any fool can write code that a computer can understand. Good programmers write code that humans can understand. – Martin Fowler. [1]

Käesoleva töö eesmärgiks on lühidalt tutvustada erinevaid disainimustreid ja hinnata nende praktilist kasutust javascriptiga. Teadagi javascript erineb teistest programmerimiskeeltest objektorienteeritud põhimõtete poolest. Samuti on eesmärk uurida, kas kõiki disainimustreid on võimalik implementeerida javascriptiga ja vajadusel leida alternatiive. Lisaks uuritakse kuidas käituvad pakutud implementatsioonid põhinedes GRASP printsiipidele ja hinnatakse pakutud kasutusvaldkondade headust. Selle tulemusena saame kogemuse, mis aitab meil paremini mõista disainimustreid ja nende kasutust javascriptiga. Eesmärgi saavutamiseks on antud töö üles ehitatud järgmiselt.

Sissejuhatuseks töö teises peatükis kirjeldatakse esmalt erinevaid objektorienteeritud programmeerimise põhimõtteid ja disainimustreid mõistmaks töö praktilist osa, milleks on disainimustrite võrdlus ja hindamine.

Kolmandas peatükis tutvustatakse javascripti ja OO põhimõtteid javascriptiga ning põhjendatakse javascripti valikut antud töös.

Neljas peatükk keskendub Gang-Of-Four (GOF) disainimustritele ja nende implementeerimisele javascriptiga. Vaadeldakse milliseid lahendusi on pakutud, hinnatakse pakutud lahendusi võrreldes GRASP printsiipidega ning vajadusel pakutakse alternatiivseid lahendusi.

Käesoleva töö skoobist jäävad välja arhitektuurimustrid, arendusmetoodikad ja muud tarkvara arendusega seotud teemad, mida ei ole eelnevalt mainitud.

2. Objektorienteeritud programmeerimine ja disain

Objektorienteeritud (OO) lähenemine programmeerimisel lihtsustab mõista reaalmaailma probleeme ja käitumist modelleerides ja kapseldades neid objektidesse. OO disaini lähenemisel on põhieesmärgiks objektide, nende vaheliste seoste ja käitumise kirjeldamine. Seejuures objekt ei ole tühjalt füüsilise objekti projektsioon vaid ta võib kirjeldada ka mingit allüksust, näiteks taaskasutatavat osa või disainimustrit.

2.1 OOP ajalugu

Paljud usuvad, et OOP on 1980ndate produkt ja kogu OOP seisnes C keele muutmises objekt orienteerituks tehes tavalisest C keelest C++ keele. Tegelikult SIMULA 1 (1962) ja SIMULA 67 (1967) on kaks varasemat OOP keelt. Lisaks, enamik OOP eelised olid saadaval varasemates Simula keeltes. Alles pärast C++ juurdumist 1990ndates hakkas OOP õitsema. Paljud arendajad võivad tunnistada, et C++ oli võimas keel ja on seda siiani ning on laialt kasutusel. Seejuures suure võimsusega kaasneb ka keerulisus. Keele arendajad tahtsid lihtsamat ja vähem keerulist keelt OO programmeerimiseks. [2]

Järgmine samm OOP arendamises sai algust 1991. aasta jaanuaris. Taheti välja töötada intelligentsed elektroonikaseadmed, mida oli võimalik tsentraalselt kontrollida ja programmeerida taskuarvutiga. Leiti, et OOP on õige suund, kuid C++ keel ei sobinud selle töö jaoks. Tulemuseks oli Oak programmeerimiskeel, millest hiljem sai Java programmeerimiskeel. [2]

2.2 OOP põhimõtted

Tabel 1 – OOP põhimõtted [3]

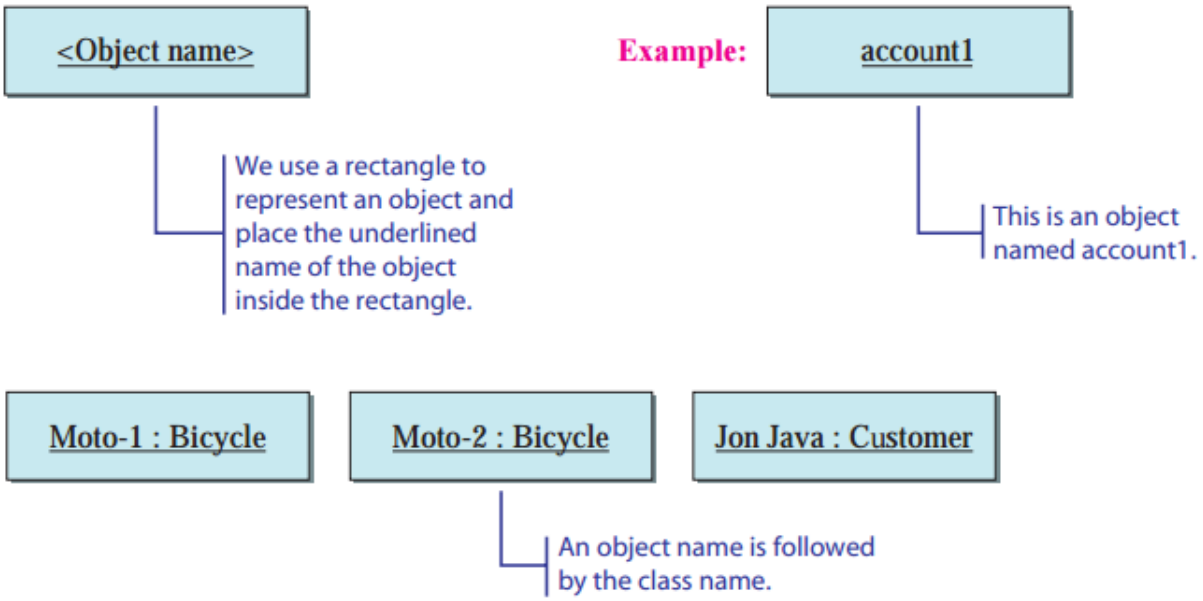
Nimetus	**Kirjeldus**
Struktuuraalsed	
Abstraktsioon	Klassi loomine lihtsustamaks reaalsuse aspekte, kasutades esiletõstetud probleeme.
Klass	Ühe või mitme objekti käitumiste ja omaduste kirjeldused.
Kapseldamine	Klasside kirjeldamine nii, et piirata ligipääs objekti teatud omadustele või sõnumite vastuvõtmisele.
Pärimine	Ühe klassi omadused ja käitumine sisaldavad või pärinevad teise klassi omadustel ja käitumisel.
Objekt	Üksik tuvastatav ühik kas abstraktne või reaalne, mis sisaldab infot enda kohta ja enda andmete manipuleerimise kirjeldusi.
Käitumuslikud	
Sõnum	Võimalus ligipääseda, muuta või manipuleerida objekti infot.
Sõnumi edastamine	Protsess, kus objekt saadab infot teisele objektile või sunnib teist objekti käivitama meetodit
Polümorfism	Lubab erinevatel objektidel vastata samale sõnumile erineva käitumisega.

OO väljavaateid võib liigitada strukturaalseteks või käitumuslikukeks lähenemisteks. Struktuuraalsed lähenemised on suunatud klasside ja objektide kirjeldamisele. Klass on objekti abstraktsioon. Klass/objekt kapseldab omadused ja käitumise ning pärimine lubab kapseldatud omadused ja käitumise pärida olemasolevalt klassilt. Teisalt käitumuslikud lähenemised on fokuseeritud objekti tegevustele. Sõnumi saatmine on protsess, kus objekt saadab infot teisele objektile või sunnib teist objekti käivitama meetodit. Viimaks, polümorfism lubab erinevatel objektidel vastata samale sõnumile erineva käitumisega. [3]

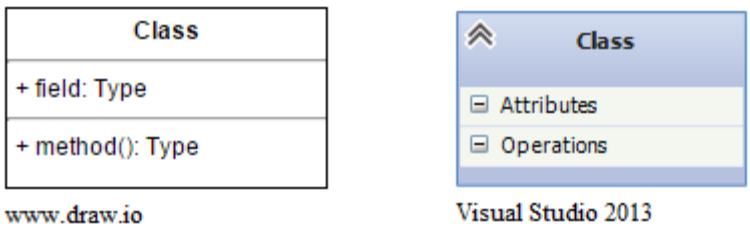
Käitumine ja struktuur on omavahel seotud selles mõttes, et käitumine manipuleerib struktuuriga, kuid käitumine lisaks toetab süsteemi ka tegevustega. Selleks, et hinnata GOF disainimustreid on vaja esmalt kirjeldada põhilised mõisted ja teemad.

2.2.1 Klassid ja objektid

Kaks kõige tähtsamat OOP mõistet on klass ja objekt. Klass sisaldab endas omaduste ja meetodite kirjeldusi. Klass võib kirjeldada füüsilise objekti omadusi ja käitumist süsteemis. Objekt esindab ühte klassi instantsi ehk objekt omab klassis kirjeldatud omadusi ja meetodeid. Objekt võib olla füüsilise objekti projektsioon kujutletavas süsteemis. Joonisel 1. olev UML (Unified modeling language) diagramm illustreerib klassi ja objekti. Objekti esitamiseks UML skeemil kasutatakse ristkülikut. [4]. Visual Studio või www.draw.io pakuvad aga klassi esitamiseks ristkülikut, mis sisaldab mitte ainult klassi nime vaid ka tema omadusi ja meetodeid. Nagu näha joonisel 2, erinevad modelleerimis vahendid pakuvad natuke erinevaid lahendusi disaini kujundamiseks, kuid idee jääb samaks.



Joonis 1. Klassi esitamine UML keeles [4]

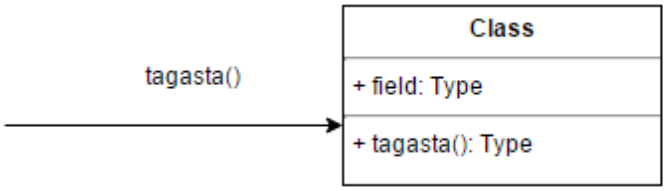


Joonis 2. Klassi kujundamine erinevates modelleerimisvahendites

Edaspidi on kasutusel www.draw.io veebikeskkonna pakutud mudelite kujundid, sest see on internetis pakutav tasuta teenus modelleerimiseks.

2.2.2 Sõnumid ja meetodid

Nagu ka päris elus saame grupeerida sarnaseid objekte, grupeeritakse programmeerimises ühesuguste omadustega objektid klassideks. Näiteks ruut ja ristkülik on mõlemad kujundid – klass kujund. Igal kujundil on olemas ümbermõõt ja pindala – klassi omadused. Kirjutates objektorienteeritud rakendusi peame me kõigepealt defineerima klasse ja kui rakendus on töös, siis kasutame klasse ja loodud objekte, et täita ülesandeid [4]. Ülesande all on mõeldud programmile antud ülesannet või osa sellest, mis tuleb ära lahendada. Ülesandeks võib pidada näiteks nii dokumendi loomist kui ka kahe arvu liitmist. Selleks, et klass või töös oleva rakenduse puhul, objekt saaks täita ülesandeid ehk käske, tuleb klassi käitumine eelnevalt vastavalt kirjeldada. Selleks, et mingit klassi objekt saaks mingit ülesannet täita, tuleb talle sõnum saata. Sõnumi saatmine on tegelikult klassi mingisuguse meetodi välja kutsumine. Sõnumi saatmisel saab kaasa anda ka parameetreid. Meetod võib täita ülesandeid ja/või tagastada mingit tüüpi andmeid. Oleneb kuidas meetod kirjeldatud on. Joonisel 3. kutsutakse välja meetod nimega „tagasta“ ning see tagastab Type tüüpi objekti.

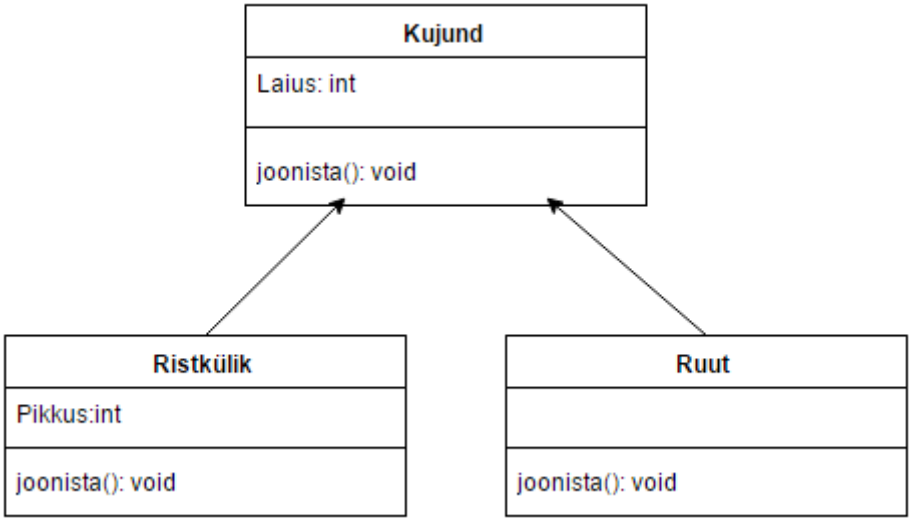


Joonis 3. Sõnumi saatmine ehk meetodivälja kutsumine

2.2.3 Pärimine

Eelnevalt toodud kujundite näite põhjal võime liikuda edasi pärimise juurde. Oletame, et meil on meetod joonista(), siis millist kujundit klass Kujund joonistama peaks? Lahenduseks võime defineerida kaks erinevat meetodit Kujundi klassis, kuid see ei ole hea lahendus. Parem on luua eraldi alamklassid Ruut ja Ristkülik, mis omakorda pärinevad ülemklassilt Kujund. Pärimise idee seisneb selles, et alamklassid jagavad baasklassis/ülemklassis ühiseid tunnuseid – antud näite puhul on ühiseks tunnuseks näiteks Laius. Joonisel 4. kujutatud pärimisel on näha, et ristkülikul on lisaks laiusele ka pikkus, mis on integer (täisarv) tüüpi, ruudul seda

vaja ei ole. Samuti on näha ka ühist meetodit joonista(), kuid see on defineeritud mitte ainult baasklassis vaid ka alamklassis. Sellest lähemalt peatükis 2.2.7.

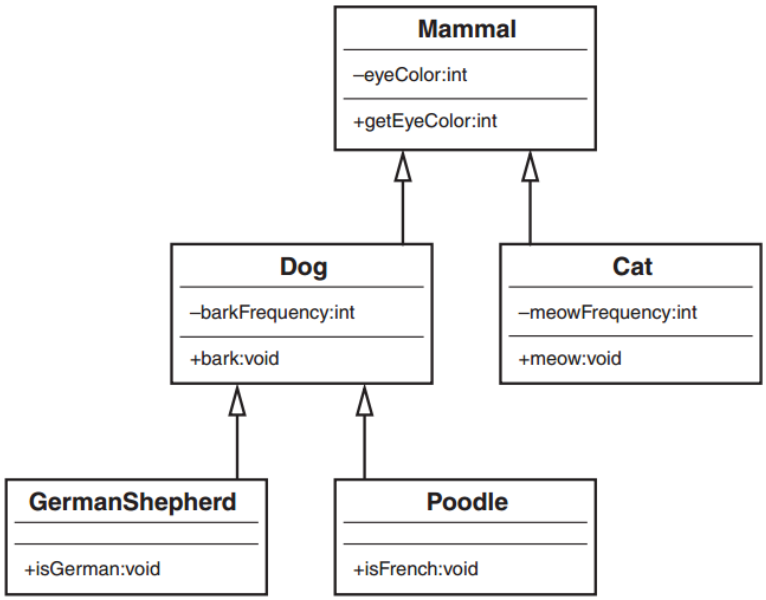


Joonis 4. Pärimine

Pärimise võlu seisneb lisaks ka selles, et pärimise abil on võimalik sama koodi dubleerimata taaskasutada ning pärimise astmed ei ole piiratud. See tähendab, et kui leiame alamklassidel ühiseid omadusi, siis oleks mõistlik viia need omadused ülemklassi. Näiteks kui meil on olemas klass „Inimene“, kus on defineeritud põhiomadused – pikkus, vanus, elukoht. Nüüd kui tekib vajadus luua klassid „ehitaja“ ja „teetööline“, siis saame taaskasutada olemasolevat klassi, sest „teetööline“ ja „ehitaja“ mõlemad omavad klassile „inimene“ omaseid tunnuseid. Pärimine on väga võimas ning kui seda õigesti kasutada on võimalik luua väga keerulisi ja tõhusaid süsteeme [4].

2.2.4 Abstraktsioonid

Pärimise puu võib kasvada üsna suureks. Pärimise jõud peitub selle abstraktsioonides ja organiseerimise tehnikas [5]. Eelnevalt kirjeldatud pärimist võib tõlgendada ka abstraktsioonina. Klass „Kujund“ on alamklasside „Ruut“ ja „Ristkülik“ abstraktsioon. Siinkohal kasutame Matt Weisfeld toodud näitet loomadega joonisel 5. Näeme, et baasklass on „Imetaja“ ning alamklassid on „Koer“ ja „Kass“. „Koer“ aga jaguneb omakorda saksa lambakoeraks ja puudliks. Uusi loomi on lihtne lisada, kuid kui mängu tulevad tõud, siis tuleb kasutada abstraktsioone – koer on koeratõugude abstraktsioon. Samas koeratõugudel on kõik imetaja omadused, kuna koer pärineb imetajalt.



Joonis 5. Abstraktsioon loomade näitel [5]

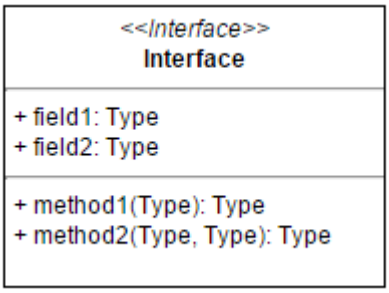
2.2.5 Kapseldamine

Mõnikord on vajadus kirjeldada klassile lisaks peidetud andmed/omadused, mida teised klassid ei kasuta ja ka ei „näe“. Teiste klasside all on silmas peetud ükskõik mis klasse – alam, ülem jm klassid. Andmete peitmisega me seame klassi andmetele/omadustele ligipääsu piirangu väljaspoolt klassi ennast. Näiteks peatükis 2.2.3 me võtsime aluseks kujundi ja kujundil oli meetod „joonista()“. Antud meetod võib pöörduda ka teiste meetodite poole, kuid kuhu täpselt ta pöördub seda me ei tea ja seda teada ei olegi vaja, sellepärast, et me kapseldame joonistamise abimeetodid klassi sisse. Meetodi „joonista()“ välja kutsumisel huvitab meid vaid see, et kujund saaks joonistatud, mitte see kuidas kujund joonistatakse. Antud näite puhul oleks mõistlik ka kasutada polümorfismi ja liideseid. Nii saame joonista meetodi deklareerida ühes failis ja implementeerida teistes vastavalt kujundile.

2.2.6 Liidesed

Inglise keelsest sõnast „Interface“ tõlgituna on eesti keeles nimetatud liides abiks polümorfismi kasutusel. Liidest võib nimetada ka lepinguks, sest liideses me kirjeldame klasside käitumismudeli ehk loome klassidega lepingu. Iga klass, mis implementeerib liidese, peab sisaldama liideses kirjeldatud omadusi ja meetodeid. Siinkohal võib tuua uuesti kujundite näite. Kui liides deklareerib meetodi pindala arvutamise jaoks, siis iga kujund, mis implementeerib liidese, peab implementeerima ka pindala arvutamise. Kuna iga kujundi pindala arvutatakse erineva valemi järgi, siis igal kujundil tulebki kasutada oma realisatsiooni. Kui tekib vajadus arvutada paljude erinevate teadaolevate kujundite pindalad, siis on lihtne

läbi liidese välja kutsuda pindala arvutamise meetod, mis on kõigil ühine. Sellist realisatsiooni võib vaja minna näiteks siseruumide pindala arvutamisel. Polümorfismi implementeerimiseks võib kasutada ka abstraktseid klasse, kuid kui rakendus jõuab testimise faasi, siis on liideseid lihtsam „mokkida“ ehk võltsida. Joonisel 6, liidese UML mudel sarnaneb klassi mudelile, erinevus seisneb ainult päises olevas märkes: „<<*Interface*>>“.



Joonis 6. Liidese UML kirjeldus

2.2.7 Polümorfism

Polümorfism tähendab kreeka keeles mitmekujulisust [6]. Objektorienteeritud programmeerimises tähendab polümorfism seda, et ühel meetodil võib olla mitu erinevat implementatsiooni. See lubab saata sama sõnumit erinevatele klassidele ning nende klasside käitumine on erinev. Polümorfism lubab kirjutada koodi nii, et seda on lihtne muuta või võimaldab valutult lisada uut funktsionaalsust. Siin võib tuua näiteid koduloomade näol. Oletame, et on olemas koer ja kass ning mõlemal loomal on omadus teha häält. Kirjeldame iga looma erineva klassiga, kuid hääle tegemise meetod on ühine, seejuures koer ja kass tagastavad erinevat häält. Tegelikult on erinevaid koduloomi väga palju ja me ei saagi eeldada, et üks klass kirjeldab kõiki koduloomi, seepärast tulebki realisatsioon tükeldada väiksemateks osadeks ja grupeerida erinevatesse klassidesse. Osa kirjeldusest võib olla realiseeritud ka abstraktsiooni abil, mis annab loomadele ühisosa – näiteks kõikidel loomadel on neli jäset ja kaks silma. Abstraktsed meetodid aga realiseeritakse igas alamklassis eraldi. Samuti nagu abstraktsiooni, kasutatakse polümorfismi järgimiseks ka liideseid. Liideses kirjeldatakse kõigi loomade ühised meetodid ning iga looma kirjeldav klass, realiseerib liidese omamoodi.

3. Javascript

Programmeerimine tähendab käskude kirjutamist arvutile. Arvuti ei ole ainult lauaarvuti vaid võib olla ka näiteks kontroller. Meid ümbritsevad elektroonilised seadmed sisaldavad kõik teatud kujul arvuteid. Ka pesumasin, elektripliit või mobiiltelefon on programmeeritud täitma teatud ülesandeid.

Programmeerimine on üks osa tarkvara loomise protsessist. Tarkvara loomine hõlmab endas erinevaid elutsükleid ja etappe – alates analüüsist ja lõpetades testimise, juurutamise ja hooldusega. Analüüsi ja testimise vahele jääbki programmeerimine ehk tarkvara realiseerimine, mis põhineb analüüsil.

Programmeerimiskeeled hakkasid arenema koos arvutite arenguga. Algselt kirjutati programme „assembleris“, mis tähendab tõlki. See tõlgib lähtekoodi ehk lähteülesanded masinkoodi üks-ühele. See tähendab, et iga allika juhis tõlgitakse täpselt üheks masina juhiseks. [7]

Edaspidi hakati looma juba kompilaatoreid ja kõrgema taseme keeli, millest on välja arenenud tänapäeval kasutusel olevad keeled. Üks populaarsemaid keeli on C, mis on kasutusel ka täna. C ei ole küll objektorienteeritud, kuid ka tänapäeval täidab ta väga olulist rolli programmeerimises. Näiteks osasid programmeeritavaid kontrollereid programmeeritakse just nimelt C keele abil. C oli ka aluseks paljudele teistele keeltele – näiteks C++, Java, C#, mis on juba ka objektorienteeritud keeled. Objektorienteeritud programmeerimiskeeli on väga palju ja kõiki antud peatükis ei mainita. Järgnevalt võtame võrdluseks C# keele ja valime kõrvale teemakohase javascripti.

Javascript, mis ei ole Java, oli loodud kümne päevaga 1995. aasta mais Eich Brendan´i poolt. Javascripti algne nimetus oli Mocha, mille valis Marc Andreessen, Netscape´i looja. Sama aasta septembris oli nimetus juba Livescript ning 1995. aasta detsembris nimetati see Javascriptiks. Aastatel 1996 – 1997 viidi Javascript ECMA-sse (European Computer Manufacturers Association), et luua standardne spetsifikatsioon. Selle alusel said teised brauserite müüjad luua Javascripti toe oma brauseris. Ametlikuks standardi nimetuseks on saanud ECMAScript, mis seab brauseritele teatud nõudmised. [8]

Pärast ECMAScripti välja töötamist ja Javascripti populaarseks muutumist pakkus Jesse James Garrett välja mõiste Ajax (asynchronous JavaScript and XML), mis oli aluseks uuele tehnoloogiale, mis võimaldas luua veebilehti nii, et lehe laadimine toimub taustal ning ei nõua terve lehe värskendamist, mis tõi veebirakendustesse rohkem dünaamilisust. See tõi kaasa Javascripti kasutamisega eesotsas, erinevad avatud lähtekoodiga teegikogumikud nagu Prototype, JQuery, Dojo ja muud. [8]

Eelnevalt oli kirjeldatud erinevaid OOP põhimõtteid ning üldiselt seostub OOP mingisuguse programmeerimiskeelega, mida on koolis õpetatud või kuskil kirjanduses kasutatud. Palju kirjandust on OOP kohta kirjutatud kasutades justnimelt C++ või Java programmeerimiskeelt. Kuna Javascript on prototüübipõhine keel, kus OO mõisted nagu klass ja pärimine ei oma tähendust või omavad teistsugust tähendust, siis antud töö eesmärk on uurida kui palju on sarnasust Javascriptil teiste pakutud OOP keeltega ja kas Javascriptiga on võimalik kirjutada funktsionaalsust objektorienteeritult järgides erinevaid disainimustreid.

3.1 OOP Javasriptiga

Javascript omab kõiki samu omadusi, nagu OOP keeled, vaatamata sellele, et näiteks klasside mõistet Javascriptis ei ole [10]. Peatükis 2.2 kirjeldatud OOP põhimõtted võib uuesti välja tuua võrdlemaks neid Javascriptiga.

- Abstraktsioon
- Klass
- Kapseldamine
- Pärimine
- Objekt
- Sõnum
- Sõnumi edastamine
- Polümorfism

3.1.1 Klass, kapseldamine, sõnumid

Javascriptis ei ole sellist mõistet nagu klass. Siinkohal on mitu erineva võimalust. Üks võimalus on defineerida funktsioon ja anda talle parameetrid, teine võimalus on luua objekt nagu on joonisel 7.

```
//Variant 1
function Loom(tyyp) {
    this.Tyyp = tyyp;
    this.Nimi = "Vasikas";
    this.meetod = function() {
        console.log("Vasikas on loodud");
    }
}
//Variant 2
var Loom = {
    Tyyp : "Tüüp",
    Nimi: "Vasiaks",
    meetod : function() {
        console.log("Vasikas on loodud");
    }
}
```

Joonis 7. Klassi kirjeldamine javascriptiga

Nagu näha on javascript omapärane selle poolest, et javascriptiga on võimalik defineerida funktsiooni sisse omakorda funktsioon, mida teiste keeltega võimalik teha ei ole.

Funktsiooni defineerimisel on võimalik kasutada ka kapseldamist. Kui näiteks C# keeles me defineerime klassi, siis ma klassi omadustele saame ette anda eesliitena kas ta on avalik või mitte ehk siis saame kapseldada teatud omadused, mida me tahame peita teiste klasside eest. Javascriptis on see samuti võimalik, kuid siin ei ole meil eesliiteid. Selle asemel me saame defineerida funktsiooni kahel erineval kujul nagu on näidatud joonisel 8.

```
//Variant 1
function Loom1() { this.Nimi = "Vasikas";}
//Variant 2
function Loom2() {
    var Nimi = "Vasiaks";
    this.AnnaNimi = function () { return Nimi; }
}
var loom1 = new Loom1();
var loom2 = new Loom2();
console.log(loom1.Nimi);
console.log(loom2.Nimi);
console.log(loom2.AnnaNimi());
//Väljund:
//Vasikas
//undefined
//Vasikas
```

Joonis 8. Kapseldamine

Joonisel 8 on näha, et funktsiooni sees defineeritud muutujad on kapseldatud ning funktsiooni omadused ei ole. Seega, peame teadma täpselt kuidas kirjeldada javascriptis „klasse“, vastasel juhul võib näiteks omaduse ja muutuja nimi kattuda ning skript ei tööta oodatud kujul.

Sõnumite saatmine toimub analoogselt teiste keeltega – selleks, et välja kutsuda mingisuguse objekti meetod, tuleb tema poole pöörduda.

3.1.2 Abstraktsioon, pärimine, polümorfism

Vastavalt kirjeldusele abstraktne klass on klass, mis omab puudulikku või ei oma üldse implementatsiooni. Javascriptis objektid pärivad oma omadusi prototüüpidelt, mitte klassidelt. Prototüübid on ise standardsed objektid, nad saavad pärineda teistelt prototüüpidelt, mis on omakorda ise ka objektid. Kui pöördutakse mingisuguse omaduse poole, siis Javascript otsib objekti omadust samast objektist. Kui omadust ei leita, siis otsitakse prototüüpidest. [9]

Kui kaks objekti jagavad sama funktsionaalsust, kuid erinevad omaduste poolest, siis üldjuhul nende prototüüp peaks olema sama objekt ja omama jagatud funktsionaalsust. Siinkohal tuleb ära märkida, et „sama“ ei tähenda identset objekti vaid sama tüüpi objekti. Tavalises OOP-is on erinevus klassi ja objekti vahel selge ja me kipume nägema Javascriptis sama. Klassi funktsioone ei kopeerita iga uue objektiga, sest nad ei muutu. Javascriptis aga selliseid definitsioone ei eksisteeri. Kui on vajadus pärida ühiselt klassilt, siis me kasutame sama klassi objekti prototüübina. [9]

Näitena loome joonisel 9. prototüübi, klassid ja objektid, kus vaatleme implementatsiooni struktuuri. Ruumi säästmiseks ei ole jäetud reavahesid.

```
var Baas = function () {
            this.kirjuta = function () { console.log('Tere ' + this.nimi); };
};
var AlamklassA = function (nimi) { this.nimi = nimi + ' A'; };
AlamklassA.prototype = new Baas();
var AlamklassB = function (nimi) { this.nimi = nimi + ' B'; };
AlamklassB.prototype = new Baas();
var AlamklassC = function (nimi) { this.nimi = nimi + ' C'; };
AlamklassC.prototype.kirjuta = function () {
    console.log('Mina erinen ' + this.nimi); };
var a = new AlamklassA('Mina olen '); a.kirjuta(); //Tere Mina olen  A
var b = new AlamklassB('Mina olen '); b.kirjuta(); //Tere Mina olen  A
var c = new AlamklassC('Mina olen '); c.kirjuta(); //Mina erinen Mina olen  C
//Väljund:
//Tere Mina olen  A
//Tere Mina olen  B
//Mina erinen Mina olen  C
```

Joonis 9. Abstraktsioon Javascriptiga

Näeme, et Javascriptiga on võimalik defineerida abstraktne objekt, mida hiljem prototüübina kasutada. Näha on ka seda, et tegelikult me loome A ja B puhul uue prototüübi objekti. Klassikalises OOP-is on abstraktne klass üks ja sama, kuid Javascriptis on ta uus objekt. Objekt C näitab, et seal kasutatud prototüüp ei ole sama. Siin jääb mulje, nagu abstraktsioon ei ole muud, kui samanimeliste funktsioonide defineerimine. Kui me tahame, et A ja B omaksid ühist prototüüpi, siis defineerime prototüübi teisiti nagu on näidatud joonisel 10.

```
//loome prototüübi
var Baas = {
    kirjuta : function () { console.log('Tere ' + this.nimi); }
};
//Alamklass A
var AlamklassA = function (nimi) { this.nimi = nimi + ' A'; };
AlamklassA.prototype = Baas;
//Alamklass B
var AlamklassB = function (nimi) { this.nimi = nimi + ' B'; };
AlamklassB.prototype = Baas;
//Muudame ühe objekti parameetrit
AlamklassA.prototype.kirjuta = function () {
    console.log('Mina erinen ' + this.nimi);
};
//Loome objektid
var a = new AlamklassA('Mina olen '); a.kirjuta(); //Tere Mina olen  A
var b = new AlamklassB('Mina olen '); b.kirjuta(); //Tere Mina olen  A
//Väljund:
//Mina erinen Mina olen  A
//Mina erinen Mina olen  B
```

Joonis 10. Abstraktsioon Javascriptiga ühise prototüübiga

Defineerisime prototüübi nii, et „kirjuta“ funktsioon on üks prototüübi parameetreid. Näeme joonisel 10, et ühe objekti prototüübi parameetri muutmine muudab kõigi teiste objektide implementatsiooni. Näha on, et Javascript on paindlik, kuid seda on ebamugav kasutada. Kui seda on ebamugav kasutada, siis ta ei ole praktiline.

Abstraktsiooni oleks vaja kasutada erinevate objektide loomiseks, millel on mingi ühisosa ning implementatsioon erinev. Siin on sarnasus ka liidestega – erinev implementatsioon. Teadagi liideseid Javascriptis ei eksisteeri, sest liidest implementeeriv klass peab realiseerima kõik liideses kirjeldatud parameetrid ja meetodid. Liideste osas on olemas muidugi ka teeke, mis lubavad seda teha teatud mööndustega, kuid seda lähemalt uurima ei hakka [10]. Samuti ei eksisteeri sellist mõistet nagu klass. Selle asemel on javascriptis võimalik objektid panna käituma nagu klassid andes neile omadused ja kirjeldades nende sisse funktsioone.

Kuna javascripti kasutatakse ka veebimängude loomiseks, siis toome ka ühe praktilisema näite. Oletame, et on vaja kirjeldada loomad ja nende käitumine. Selleks defineerime prototüübi Loom ja kaks klassi: Kass ja Koer nagu on joonisel 11.

```
//loome prototüübi
var Loom = function (nimi, haalefn) {
    this.Nimi = nimi;
    this.TeeHaalt = haalefn;
};
var Kass = new Loom('Miisu', function ()
{ console.log(this.Nimi + ':  Miau'); });
var Koer = new Loom('Laika', function ()
{ console.log(this.Nimi + ':  Auh'); });
Kass.TeeHaalt();
Koer.TeeHaalt();
//Väljund:
//Miisu:  Miau
//Laika:  Auh
```

Joonis 11. Abstraktsioon Javascriptiga loomade näitel

Nagu näha, me ei kirjelda enam prototüübis funktsiooni vaid anname selle prototüübile sisendina. See ei erine tegelikult tavalisest funktsiooni kirjeldamisest ja sellisel kujul ei ole mõtet prototüüpi kasutada. Eesmärk on luua kass ja koer nii, et me ei defineeri loomisel nende käitumist, vaid käitumine oleks juba eelnevalt kirjeldatud. Joonisel 12 saame lõpuks oodatud tulemuse, kus näeme, et koer ja kass ei ole kirjeldatud funktsioonidena, vaid kasutame Object.create() meetodit, et luua vastavalt prototüübile uus objekt.

```
//loome prototüübi
var Loom = {
    loo: function(nimi) {
        this.Nimi = nimi;
        return this;
    }
};
var Kass = Object.create(Loom);
Kass.TeeHaalt = function() {
    console.log(this.Nimi + ':  Miau');
}
var Koer = Object.create(Loom);
Koer.TeeHaalt = function() {
     console.log(this.Nimi + ':  Auh');
}
var miisu = Object.create(Kass).loo('Miisu').TeeHaalt();
var laika = Object.create(Koer).loo('Laika').TeeHaalt();

//Väljund:
//Miisu:  Miau
//Laika:  Auh
```

Joonis 12. Abstraktsioon Javascriptiga kassi ja koera näitel

Javascripti loetakse ka OOP keeleks ning veendusime, et tõepoolest on võimalik realiseerida javascriptiga teatud OOP põhimõtteid, kuid nägime ka seda, et võimalusi on rohkem kui üks. Näiteks C# või Java programmeerimiskeeles on klassi kirjeldamiseks ja pärimiseks ainult üks võimalus. Javascripti paindlikkus on hea omadus, kuid algajale võib see olla takistuskiviks, sest halvasti kirjutatud koodi on kohati raskem ümber teha, kui nullist uuesti kirjutada. Samuti tuleb ära märkida ka see, et javascripti kirjutatakse üldjuhul ühte faili ja suurema rakenduse puhul võib see fail venida väga pikaks. Võrdluseks võib tuua näiteks Java, mis ei lubagi kirjeldada kahte klassi ühes failis, seetõttu on kood ka loetavam ja arusaadavam.

4. Mustrid

4.1 GOF ehk Gang of Four disainimustrid

OOP disain kirjeldab 23 GOF (Gang of Four) disainimustrit. Disainimuster on tarkvara lahenduseks tarkvara disaini probleemidele ja koodi taaskasutatavusele. Disainimustrid on jaotatud kolme põhigruppi: Loovad, Struktuursed ja Käitumuslikud. Tabel 2. annab lühikirjelduse grupeeritud GOF mustritele. [11]

Tabel 2. Gang of Four disainimustrid

Loovad	
Abstraktne tehas (Abstract Factory)	Loob objekti teatud klassist.
Ehitaja (Builder)	Eraldab objekti konstruktori objektist endast
Tehase meetod (Factory method)	Loob objekti tuletatud klassidest
Prototüüp (Prototype)	Loodud objekti kopeerimiseks või kloonimiseks
Singel (Singletone)	Klass, mida võib eksisteerida ainult üks eksemplar
Struktuursed	
Adapter (Adapter)	Sobitab erinevaid liideseid
Sild (Bridge)	Eraldab objekti liidese selle implementatsioonist
Ühend (Composite)	Puu struktuur üksik- ja liitobjektidest
Dekoraator (Decorator)	Lisab objektile kohustusi dünaamiliselt
Fassaad (Facade)	Üksik klass, mis esindab allsüsteemi
Karbeskaal (Flyweight)	Peen objekt efektiivseks jagamiseks
Esindaja (Proxy)	Objekt, mis esindab teist objekti
Käitumuslikud	
Käsuliin (Chain of responsibility)	Päringu saatmise viis objektide jadas
Käsk (Command)	Kapseldab käsu objektina
Interpretaator (Interpreter)	Keele elementide lisamine rakendusse
Iteraator (Iterator)	Järjestik juurdepääs objektide massiivile
Vahendaja (Mediator)	Defineerib lihtsustatud suhtluse objektide vahel
Memento (Memento)	Säilitab ja taastab objekti seisu

Vaatleja (Observer)	Objektide muudatustest teatamise viis
Olek (State)	Muuda objekti käitumist kui olek selle muutub
Strateegia (Strategy)	Kapseldab algoritmi klassi sees
Šabloonmeetod (Template method)	Eraldab üldise algoritmi varieeruvast alamosast
Külastaja (Visitor)	Defineerib klassile uue operatsiooni seda muutmata

4.2 GRASP General Responsibility Assignment Software Patterns

OOP juures ei piisa ainult erinevate disainimustrite tundmisest. Selleks, et mõista kuidas erinevaid mustreid kasutada oleks vaja mõista ka kuidas neid hinnata. Üheks hindamise mõõdupuuks võiks võtta GRASP printsiibid [12], mis on kirjeldatud tabelis 3. GRASP printsiipe on kokku üheksa ja nad kirjeldavad baasprintsiipe, mida tuleks järgida OO süsteemide disainimise puhul.

Tabel 3. GRASP printsiibid [12]

Ekspert (Expert)	Anda vastustus infoeksperdile - klassile
Looja (Creator)	Kes tekitab mingi klassi objekte
Madal sõltuvus (Low Coupling)	Jaota vastutus nõnda, et sõltuvus klasside vahel jääks madalaks
Kõrge kokkukuuluvus (High cohesion)	Klass omab ühte kindlat ülesannet hästi, mitte ei tee palju erinevaid ülesandeid.
Kontroller (Controller)	Kes haldab süsteemi taseme sündmust
Polümorfism (Polymorphism)	Kuidas hallata klassist sõltuvaid alternatiive
Puhas väljamõeldis (Pure fabrication)	Anna kõrge kokkukuuluvusega vastutused kunstlikule klassile, mis ei esine domeenimudelis.
Kaudsus (Indirection)	Anna vaheobjektile vastutus teiste komponentide ja teenuste suhtluse vahendamiseks, et hoiduda otsesest sõltuvusest.
Kaitstud variatsioonid (Protected variantions)	Kuidas disainida klasse, süsteeme ja allsüsteeme nii, et teatud elementide ebastabiilsusest tulenevad variatsioonid ei omaks mõju teistele elementidele.

4.3 Implementeerimine ja hindamine

4.3.1 Abstraktne tehas (Abstract Factory) ja Tehase meetod (Factory method)

See muster toetab toodete loomist, teadmata toote tüüpi. Abstraktne tehas võib olla konkreetsete tehaste kogum, mis igaüks loovad erinevat tüüpi objekte ja erinevaid kombinatsioone nendest. See muster isoleerib toote definitsioone ja nende klasside nimesid kliendist nii, et ainus võimalus on neile ligi pääseda tehase kaudu. Sel põhjusel võivad tooted lihtsasti olla muudetavad ilma, et peaks muutuma kliendi struktuur. [12]

See võib tunduda esmapilgul tühi töö, kui me kasutame tehaseid rakenduse siseselt objektide loomiseks. Miks me ei võiks lihtsalt kasutada klassi konstruktorit või lihtsalt luua uus instants? Põhjus seisneb selles, et toote loomine ei ole objekti loomine. Kui üks konkreetne tehas loob kindlat tüüpi toodete objekte, siis abstraktne tehas jätabki selle töö konkreetsele tehasele ja ise ainult suunab. Lõpptulemuseks võib olla loodud objekt koos alamobjektidega. Lisas 1 on illustreeritud abstraktse tehase muster UML skeemina, kus on näha, et kliendil on ligipääs vaid tehasele ning tehas muretseb alamklasside loomise eest.

Eesmärk on implementeerida tehase muster Javascriptiga ning kõige paremini sobib sel juhul veebimängu näide. Oletame, et me peame rakenduse käigus looma kolme erinevat tüüpi vaenlasi. Vaenlased olgu kõik abstraktsed klassid ning igal vaenlasel on ka eraldi relvad. Kes selle struktuuri meile loob? Siin tulebki abiks abstraktne tehase muster. Me anname sisendiks vaenlase nimetuse ning saame vastuseks vastavalt vaenlase erinevate relvadega.

Joonisel 13 ei ole realiseeritud relvade loomist, kuid idee on ilusti välja toodud. Kuna teistes keeltes on kasutusel liides, siis joonisel 13, viimane väljundi rida näitabki seda, et javascriptiga peame me ise tagama selle, et igas konkreetses objektis on olemas kõik samasugused funktsioonid ja meetodid. Siinkohal võib märkida ära, et antud näites kehtib ka kõrge kokkukuuluvus ning madal sõltuvus. Abstraktse tehase töö ei sõltu vaenlase muudatustest ning igal objektil on oma kindel ülesanne.

```
function VaenlaseProto() {this.rynda = function() {
     console.log(this.Nimi + " Ründas.");
};}
function Vaenlane1(nimi) {
    this.Nimi = nimi;
    this.Tase = 0;
    this.vahetaRelva = function() {
         console.log(this.Nimi + " vahetas relva.");
    }
}
Vaenlane1.prototype = new VaenlaseProto();
function Vaenlane2(nimi) {
    this.Nimi = nimi;
    this.Tase = 10;
}
Vaenlane2.prototype = new VaenlaseProto();
function Venlane1Tehas() {
    this.loo = function (nimi) {return new Vaenlane1(nimi);};
}
function Venlane2Tehas() {
    this.loo = function (nimi) {return new Vaenlane2(nimi);};
}
function AbstraktneTehas(tase) {
    var vaenlase1Tehas = new Venlane1Tehas();
    var vaenlase2Tehas = new Venlane2Tehas();
    if (tase == 0) {return vaenlase1Tehas.loo("Vaenlane 1");}
    if (tase == 10) {return vaenlase2Tehas.loo("Vaenlane 2");}
    return null;
}
function run() {
    var vaenlased = [];
    vaenlased.push(new AbstraktneTehas(0));
    vaenlased.push(new AbstraktneTehas(10));
    for (var i = 0, len = vaenlased.length; i < len; i++) {
        vaenlased[i].rynda();
        vaenlased[i].vahetaRelva();
    }
}
$(document).ready(function () { run(); });
//Väljund
//Vaenlane 1 Ründas.
//Vaenlane 1 vahetas relva.
//Vaenlane 2 Ründas.
//Uncaught TypeError: vaenlased[i].vahetaRelva is not a function
```

Joonis 13. Abstraktne tehas

4.3.2 Ehitaja (Builder)

Mõnikord on vajadus luua keerukas objekt ning objekti komponentide tüübid võivad varieeruda olenevalt kontekstist. Eesmärk on kaitsta süsteemi nende variatsioonide eest ning peita keerukas komponendi lisamise protsess rakenduse eest. Selleks, et lihtsustada keerukate objektide loomist tuleb kasuks ehitaja muster. [13] Lisas 2. on illustreeritud ehitaja UML skeem ning joonisel 14 selle implementatsioon.

```
function Tootmine() {
    this.LooToode = function(ehitaja) {
        ehitaja.looMoobel();
        ehitaja.lisaKomponente();
        return ehitaja.TagastaToode();
    }
}
function Kapp() {
    this.Uksi = 0;
    this.Seinu = 4;
    this.lisaUksi = function() {this.Uksi = 2;}
    this.Naita = function() { console.log("Kapp: "+this.Uksi + " Ust " +
    this.Seinu + " Seina"); }
}
function Riiul() {
    this.Uksi = 0;
    this.Seinu = 3;
    this.lisaUksi = function () {this.Uksi = 1;}
    this.Naita = function () { console.log("Riiul: "+this.Uksi + " Ust " +
    this.Seinu + " Seina"); }
}
function KapiEhitaja() {
    this.kapp = null;
    this.looMoobel = function() {this.kapp = new Kapp();}
    this.lisaKomponente = function() {this.kapp.lisaUksi();}
    this.TagastaToode = function() {return this.kapp;}
}
function RiiuliEhitaja() {
    this.riiul = null;
    this.looMoobel = function () {this.riiul = new Riiul();}
    this.lisaKomponente = function () {this.riiul.lisaUksi();}
    this.TagastaToode = function () {return this.riiul;}
}
var tootja = new Tootmine();
var kapiehitaja = new KapiEhitaja();
var riiuliEhitaja = new RiiuliEhitaja();
var kapp = tootja.LooToode(kapiehitaja);
var riiul = tootja.LooToode(riiuliEhitaja);
kapp.Naita(); riiul.Naita();
//Väljund:
//Kapp: 2 Ust 4 Seina
//Riiul: 1 Ust 3 Seina
```

Joonis 14. Ehitaja muster

Nagu näha ka allikas [11], on selle mustri kasutamine vähe populaarne ja üldjuhul luuakse klassidele konstruktorid, mis teevad sama töö ära, mida teeb ehitaja. Joonisel 15 on toodud kummuti näide, kus ei ole kasutatud ehitaja mustrit ning väljund on sama. Siin läheb kaduma küll GRASP-i Looja printsiip, kuid koodi on kordades vähem.

```
function Kummut() {
    this.Uksi = 0;
    this.Seinu = 4;
    this.lisaUksi = function () { this.Uksi = 4; }
    this.Naita = function () { console.log("Kummut: " + this.Uksi + " Ust " +
    this.Seinu + " Seina"); }
    this.lisaUksi();
}
var kummut = new Kummut();
kummut.Naita();
//Väljund:
//Kummut: 4 Ust 4 Seina
```

Joonis 15. Ehitaja mustri alternatiiv

4.3.3 Prototüüp (Prototype)

Prototüüp loob uue objekti, kuid mitte tühja, vaid juba initsialiseeirutd objekti. Seda võib nimetada ka objekti kloonimiseks. Klassikalised keeled nagu Java või C# ei kasutavad prototüüpe harva, kuid javascript on prototüübi põhine keel ning siin kasutatakse seda tihti. [13]

Joonisel 16 on kujutatud prototüübi implementatsioon. Kuna Javascript on väga paindlik keel, siis on samuti siin erinevaid viise kuidas saab seda implementeerida, kuid Joonisl 17 tuleb välja Javascripti huvitav omadus.

```
function KasutajaPrototyyp(proto) {
    this.proto = proto;
    this.klooni = function () {
        var uuskasutaja = new Kasutaja();
        uuskasutaja.Nimi = proto.Nimi;
        uuskasutaja.Vanus = proto.Vanus;
        return uuskasutaja; };
}
function Kasutaja(nimi, vanus) {
    this.Nimi = nimi; this.Vanus = vanus;
    this.Naita = function () {console.log(this.Nimi + " " + this.Vanus);};
}
var olemasolevKasutaja = new Kasutaja("Nimi", 15);
console.log("Olemasolev kasutaja: ");
olemasolevKasutaja.Naita();
var prototyyp = new KasutajaPrototyyp(olemasolevKasutaja);
var uuskasutaja = prototyyp.klooni();
console.log("Uus kasutaja: ");
uuskasutaja.Naita();
```

Joonis 16. Prototüüp muster

Prototüübi kasutamisel tuleb olla ettevaatlik selle struktuuri ülesehitamisel ja arvestada javascripti omadusi ning teada kas javascriptis toimub info edastamine viitamise või kopeerimise kaudu. Joonisel 18, nägime, et kui oleks edastatud puhtalt väärtus, siis „objekt1.vaartus“ muudatus ei oleks mõjutanud objekt1-te väljaspool funktsiooni. Kui see oleks edastatud viitena, siis arv oleks olnud 100 ja objekt2.vaartus oleks „muudetud“. Selles situatsioonis väärtus oli edastatud puhtalt väärtusena, kuid see, väärtus ise oli viit.

```
function Muuda(arv, objekt1, objekt2) {
    arv = arv * 10;
    objekt1.vaartus = "muudetud";
    objekt2 = { vaartus: "muudetud" };
}
var arv = 10;
var objekt1 = { vaartus: "muutmata" };
var objekt2 = { vaartus: "muutmata" };
Muuda(arv, objekt1, objekt2);
console.log(arv);
console.log(objekt1.vaartus);
console.log(objekt2.vaartus);
//Väljund
//10
//muudetud
//muutmata
```

Joonis 17. Javascripti omadus Call by Sharing

Kui üritame kloonida objekti parameetreid, siis see ei õnnestu nagu on näha joonisel 18.

```
function Klooni(objekt1, objekt2) {
    objekt2 = { vaartus: objekt1.vaartus };
}
var objekt1 = { vaartus: "1" };
var objekt2 = { vaartus: "null" };
Klooni(objekt1, objekt2);
console.log(arv);
console.log(objekt1.vaartus);
console.log(objekt2.vaartus);
//Väljund
//1
//null
```

Joonis 18. Parameetrite kloonimine

4.3.4 Singel (Singletone)

Singel ei vaja täpsemat kirjeldus ega UML skeemi. Seda võib nimetada ka kui globaalne muutuja. Javascriptiga ongi globaalse muutuja defineerimine kõige lihtsam osa. Allikas [13] pakutud Singel mustri lahendus on arusaadav, kuid pisut pikavõitu (Joonis 19).

```
var Westeros;
(function (Westeros) {
    var Wall = (function () {
        function Wall() {
            this.height = 0;
            if (Wall._instance)
                return Wall._instance;
            Wall._instance = this;
        }
        Wall.prototype.setHeight = function (height) {
            this.height = height;
        };
        Wall.prototype.getStatus = function () {
            console.log("Wall is " + this.height + " meters tall");
        };
        Wall.getInstance = function () {
            if (!Wall._instance) {
                Wall._instance = new Wall();
            }
            return Wall._instance;
        };
        Wall._instance = null;
        return Wall;
    })();
    Westeros.Wall = Wall;
})(Westeros || (Westeros = {}));
```

Joonis 19. Singel muster

Alternatiiviks sellele võib öelda, et kõik muutujad, mis on defineeritud väljaspool funktsiooni peegeldavad Singel mustrit. C# keeles võib selle tunnuseks olla „static“ eesliide klassi ees.

4.3.5 Adapter (Adapter)

Adapter aitab konverteerida ühe klassi liides selliseks mida teine klass ootab [11]. Üks lahendus võib olla ka see, et kohandame klassi ja liidest nii, et see sobiks meie teise klassiga, kuid tihtipeale tuleb tarkvara arendada koos kolmanda osapoolega ning teiste lähtekoodile meil ligipääsu ei ole. Teine oht on ka see, et meie olemasolev liidese realisatsioon võib olla kuskil mujal kasutusel. Selleks, et välistada suurt sõltuvust objektide vahel võib luua vahekihi ehk adapteri.

Adapterit võib kasutada erinevates situatsioonides – näiteks kolmanda osapoolega suhtlemisel või kasvõi funktsioonide grupeerimisel. Järgnevalt joonisel 20 on toodud näide grupeerimisel.

```
function Auto() {
    this.SidurAlla = function() {console.log("1");}
    this.SidurYles = function() {console.log("4");}
    this.JargmineKaik = function() {console.log("3");}
    this.Tyhikaik= function() {console.log("2");}
}
var AutoAdapter = function () {
    this._auto = new Auto();
}
AutoAdapter.prototype.VahetaKaiku = function() {
    this._auto.SidurAlla();
    this._auto.Tyhikaik();
    this._auto.JargmineKaik();
    this._auto.SidurYles();
}
var Uusauto = new AutoAdapter();
Uusauto.VahetaKaiku();
//Väljund:
//1
//2
//3
//4
```

Joonis 20. Adapter

Selgub, et antud näide on igati täiuslik - Uusauto on kaitstud variatsioonide vastu, objektid omavahel madala sõltuvusega ning kehtib ka GRASP kaudsuse printsiip. Teisalt tekib küsimus, et kui palju kasutatakse javascripti-s kolmandate osapooltega suhtlust, sest Joonis 20 illustreeris ainult rakenduse sisest suhtlust objektide vahel? Sellele küsimusele võib vastuse anda ka Ajax (asynchronous JavaScript and XML). Kui me teeme päringu kasutades Ajax-it näiteks veebiteenuse poole, siis me ei näe teenuse lähtekoodi ning siinkohal võib olla adapter praktilisem, otsustades ise millise veebiteenuse poole me pöördume vastavalt vajadusele. See peegeldab osaliselt ka polümorfismi põhimõtteid – erinevad teenused käituvad erinevalt.

4.3.6 Sild (Bridge)

Silla mustrit võib iseloomustada kui adapteri täiendust. Andes erinevad liidesed me saame ehitada mitmed adapterid, mis igaüks käitub vahendajana erinevatele implementatsioonidele. [13]

Eelnevalt mainitud veebiteenuste näide tuleb nüüd realiseerida. Oletame, et meil on klient ja kaks erinevat teenust ning rakendus pöördub adapteri poole, mis omakorda suunab päringu vastavalt vajadusele erinevatesse teenustesse. Selliselt käituv adapter ongi sild.

```
function SwedAdapter() {return "Swed intress: 10";}
function SEBAdapter() {return "SEB intress: 12";}
function ArvutaintressAdapter(pank) {
    if (pank == 1) { return SwedAdapter(); }
    else if (pank == 2) {return SEBAdapter();}
    return "Intress 0";
}
var makseandmed = 1;
var makseandmed2 = 2;
var intress1 = ArvutaintressAdapter(makseandmed);
var intress2 = ArvutaintressAdapter(makseandmed2);
console.log(intress1);
console.log(intress2);
//Väljund
//Swed intress: 10
//SEB intress: 12
```

Joonis 21. Sild

Joonisel 21 näeme, et pöördudes „Arvutaintressadapter“ poole me ei tea kuhu meil on vaja pöörduda ja makseandmed on abstraktne objekt, seepärast ongi meil vajadus sellise struktuuri järele. Selliselt on säilitatud GRASP kaudsuse printsiip ning makseandmed kaitstud variatsioonide vastu, kui peaks tekkima uus pank. Selgub, et silla realiseerimine javascriptiga ei erine teistest keeltest.

4.3.7 Ühend (Composite)

Eelnevalt on mainitud, et eelistuseks on alati madal sõltuvus. Kuna pärimine on üks kõrge sõltuvuse vorm, siis selle asemel on pakutud ühendi muster [13]. Ühendi muster on erijuht, mida koheldakse kui vahetatavate osadega objekti. Seda võib nimetada ka objektipuuks, mis kirjeldab mingit hierarhiat. Näiteks pildigaleriis on erinevad albumid ja albumites omakorda pildid. Siin vaataks lihtsustatud näited, mis on Lisas 5. Näeme, et see käitubki täpselt nagu kahendpuu, mille alamosad võivad olla pildid või albumid.

Vaatame ka allikas [13] pakutud lahendust, mis on lisas 6. Näeme, et antud lahendus ei ole muud kui massiivi kirjutamine ja sealt lugemine. Kui võtame testvektorid viimastelt ridadelt, mis on kujutatud joonisel 22 ja üritame lahendada lühemalt oodates sama väljundit.

```
var egg = new SimpleIngredient("Egg", 155, 6, 0);
var milk = new SimpleIngredient("Milk", 42, 0, 0);
var sugar = new SimpleIngredient("Sugar", 387, 0, 0);
var rice = new SimpleIngredient("Rice", 370, 8, 0);
var ricePudding = new CompoundIngredient("Rice Pudding");
ricePudding.AddIngredient(egg);
ricePudding.AddIngredient(rice);
ricePudding.AddIngredient(milk);
ricePudding.AddIngredient(sugar);
```

Joonis 22. Ühendi testvektorid

Pakutud lahendus ei olnud kaitstud variatsioonide vastu ning seal oli kõrge sõltuvus. Kui oleks vaja lisada uus koostisosa, millel on rohkem omadusi kui 4, siis me peaksime muutma kogu koodi, seepärast ei erine see järgnevalt joonisel 23 pakutud alternatiivsest näitest.

```
Komponent = function(args) {
    var list = new [args[0], args[1], args[2], args[3]];
    return list;
}
Kook =  {
    komponendid : new Array(),
    AddIngredient: function (komponent) {
        komponendid.push(komponent);
    },
    GetCalories : function() {
        var total = 0;
        for (var i = 0; i < this.komponendid.length; i++) {
            total += this.komponendid[2];
        }
        return total;
    }
}
var egg = new SimpleIngredient("Egg", 155, 6, 0);
var milk = new SimpleIngredient("Milk", 42, 0, 0);
var sugar = new SimpleIngredient("Sugar", 387, 0, 0);
var rice = new SimpleIngredient("Rice", 370, 8, 0);
var ricePudding = new CompoundIngredient("Rice Pudding");
ricePudding.AddIngredient(egg);
ricePudding.AddIngredient(rice);
ricePudding.AddIngredient(milk);
ricePudding.AddIngredient(sugar);
console.log(ricePudding.GetCalories() + " calories");
//Väljund
//954 calories
```

Joonis 23. Pakutud Ühendi mustri lihtsam versioon

Pakutud lühendatud näide on lühem ka sellepärast, et seal ei ole implementeeritud kõiki meetodeid, mis algses näites, kuid idee seisneb selles, et tegelikult me ei vaja sellist abstraktsiooni ja prototüüpide loomist, mis on javascriptiga ebamugav. Me saime sama tulemuse ka lihtsamalt ning lõhkumata GRASP printsiipe.

4.3.8 Dekoraator (Decorator)

Eesmärk on lisada objektile vastutusi ilma alamklasse defineerimata ehk dünaamiliselt. Selleks võib kasutada dekoraator objekte, mis lisavad uusi meetodeid ja delegeerivad olemasolevad operatsioonid dekoreeritavale objektile. Näiteks C# keeles on dekoraatoril meetodite eesliiteks „override“ ning dekoreeritav klass on abstraktne klass koos abstraktse meetodiga. Kui toimub objekti meetodi poole pöördumine, siis täidetakse nii algselt kirjeldatud operatsioon kui ka dekoraatori poolt lisatud operatsioonid. Vaatame näidet javascriptiga Lisas 7, mis on pärit allikast [13]. Näeme, et väljundiks on 1 ja jääb mulje, et midagi ei ole muutnud, kuid kui vahetame dekoraatoris olevas „CalculateDamageFromHit“ meetodis ära kohakuti kaks rida, siis tagastatakse hoopis uus väärtus. See tähendab, et dekoraator muutis algset objekti tööpõhimõtet dekoreerides seda uue funktsionaalsusega.

Selgub, et dekoraatori puhul javascriptis me tegelikult kopeerime algse objekti koos tema käitumismudelitega, muudame seda ja tagastame uue muudetud objekti. Seepärast võib sama loogika kirjutada ümber jällegi teisiti nagu on illustreeritud joonisel 24.

```
var Kasutaja = function (nimi) {
    this.Nimi = nimi;
    this.ytle = function () {console.log("Kasutajanimi: " + this.Nimi);};
}
var DekoraatorKasutaja = function (kasutaja, vanus) {
    this.DKasutaja = kasutaja;
    this.Vanus = vanus;
    this.ytle = function () {
        console.log("Dekoreeritud kasutaja: " + DKasutaja.Nimi + " " +
        this.Vanus);
    };
}
var kasutaja = new Kasutaja("Kasutaja");
kasutaja.ytle();
var dekoreeritudKasutaja = new DekoraatorKasutaja(kasutaja, 25);
dekoreeritudKasutaja.ytle();
//Väljund
//Kasutajanimi: Kasutaja
//Dekoreeritud kasutaja: Kasutaja 25
```

Joonis 24. Dekoraator

4.3.9 Fassaad (Facade)

Fassaadi muster on jällegi üks adapter mustri erijuht, kus fassaad võib esindada tervet allsüsteemi. Siin võib näha sarnasust peatükis 4.3.6 pakutud realisatsiooniga, kus on näitena võetud pangad joonisel 25.

```
var Intress = function (name) {
    this.name = name;
}
Intress.prototype = {
    tagasta: function (amount) {
        var vastus = "";
        var swed = new Swed();
        var seb = new Seb();
        if (!swed.arvuta(this.name, amount)) {
            console.log("Swed vastus: negatiivne");
            vastus = "negatiivne";
        }
        if (seb.anna(this.name)) {
            console.log("Seb vastus: positiivne");
            vastus = "positiivne";
        }
        return vastus;
    }
}
var Swed = function () {
    this.arvuta = function (nimi, summa) {/* siin võib olla keeruline
                                             loogika*/return false;}
}
var Seb = function () {
    this.anna = function (nimi) {/* siin võib olla keeruline
                                    loogika*/ return true;}
}
var intr = new Intress("Klient");
var vastus = intr.tagasta("50");
console.log("Lõplik vastus: "+vastus);
//Väljund
//Swed vastus: negatiivne
//Seb vastus: positiivne
//Lõplik vastus: positiivne
```

Joonis 25. Fassaad

Javascriptiga fassaadi ja adapterite implementeerimine ei erine teistest keeltest oluliselt. Erinevus seisneb selles, et me ei kasuta siin liideseid vaid abstraktsiooni, mida tegelikult saaks kasutada ka teiste keelte puhul. Teisalt on see muster javascriptis väga kõrge kasutatavusega [14] ning tuues veel ühe näite allikast [13] joonisel 26, võib öelda, et see on igati praktiline.

Instead of writing the following code:
$.ajax({method: "PUT",
url: "https://settings.blob.core.windows.net/container/set1",data: "setting data 1"});
$.ajax({method: "PUT",
url: "https://settings.blob.core.windows.net/container/set2",data: "setting data 2"});
$.ajax({method: "PUT",
url: "https://settings.blob.core.windows.net/container/set3",data: "setting data 3"});
A façade could be written which encapsulates all of these calls and provides an interface like:
public interface SettingSaver{
Save(settings: Settings); //previous code in this method
Retrieve():Settings;
}

Joonis 26. Väljavõte raamatust – Fassaadi näide [14]

4.3.10 Kärbeskaal (Flyweight)

Kärbeskaal on laialt kasutusel ning idee seisneb mälukasutuse optimeerimises [14]. Kui tekib vajadus suure hulga objektide loomise järele, siis tuleb arvestada ka seda, et mida rohkem uusi instantse, seda suurem mälukasutus. Objektid võivad olla ka keerukad ja nii suured, et mõnes olukorras ongi piiratud arv objekte lubatud kasutada enne kui süsteem hangub [13]. Vaatleme joonist 27, kus on objektide loomiseks kasutatud tehast.

```
var Pruss = function (tootja, pikkus, laius, korgus) {
    this.puitmaterjal = KarbeskaalTehas.AnnaYks(pikkus, laius, korgus);
    this.Tootja = tootja;
}
function Puitmaterjal(pikkus, laius, korgus) {
    this.Pikkus = pikkus;
    this.Laius = laius;
    this.Korgus = korgus;
}
KarbeskaalTehas = (function() {
    var puitmaterjalid = {}
    var kogus = 0;
    return {
        AnnaYks: function (pikkus, laius, korgus) {
            if (puitmaterjalid[pikkus + laius] == null) {
                puitmaterjalid[pikkus + laius] = new Puitmaterjal(pikkus,
                                                                  laius, korgus);
                kogus++;
            }
            return puitmaterjalid[pikkus + laius];},
        AnnaKogus: function() {return kogus;}
    }
})();
function Puitmaterjalid() {
    var prussid = {};
    var kogus = 0;
    return {
        lisa: function (tootja, pikkus, laius, korgus) {
            prussid[tootja] = new Pruss(tootja, pikkus, laius, korgus);
            kogus++; },
        annaKogus: function () {return kogus;}
    };
}
var puitmaterjalid = new Puitmaterjalid();
puitmaterjalid.lisa("tootja1", "2", "3", "4");
puitmaterjalid.lisa("tootja1", "2", "3", "5");
puitmaterjalid.lisa("tootja1", "2", "3", "5");
puitmaterjalid.lisa("tootja1", "2", "3", "6");
puitmaterjalid.lisa("tootja2", "3", "3", "4");
puitmaterjalid.lisa("tootja2", "3", "3", "4");
console.log("Prussid: " + puitmaterjalid.annaKogus());
console.log("Kärbeskaale: " + KarbeskaalTehas.AnnaKogus());
//Väljund
//Prussid: 6
//Kärbeskaale: 2
```

Joonis 27. Kärbeskaal

Tulemustest võib välja lugeda, et prusse tuli 6, kuid nad olid taandatud 2-le kärbeskaalule. Seda võib võrrelda Huffmani [15] pakkimise algoritmiga, kus seatakse ühisosale indeksid. Selgub, et siingi ei ole javascriptiga implementeerimisel ainult üks võimalus vaid mitmeid nagu on toodud allikas [13] näites, mis on lisas 8. Veendusime ka selles, et implementatsioon javascriptiga lisab paindlikkust ja veelgi rohkem kasutusmugavust antud mustri puhul.

4.3.11 Esindaja (Proxy)

Esindaja käitub täpselt samuti nagu algne objekt esindades seda – see on justkui algse objekti koopia. Võtame aluseks allikas [14] toodud näite google asukoha määramise süsteemist, mis on lisas 9. Kui tehakse palju päringuid ühe asukoha saamiseks, siis ei ole mõtet alati teha uut päringut serveri poole, sest see võib olla liiga kulukas. Esindaja salvestab tulemused vahemällu ning tagastab nad kiiremini kui teenus. Näitest näeme ka seda, et tegelikult esindaja muster sisaldab kärbeskaalu mustrit ning kombineerides kahte erinevat disainimustrit salvestab juba olemasolevad tulemused vahemällu nagu on kujutatud joonisel 27. Selgub, et javascripti implementatsioon kärbeskaalu ja esindaja mustri puhul on enamjaolt sama. Samuti võib näha sarnasusi adapteri ja silla mustriga, kus toimub suunamine. Lisaks on võimalik siia pookida ka sisendite konverteerimine ning see viitaks juba dekoraatori mustrile. Javascript lubab selliselt kombineerida erinevaid mustreid, säilitades enamjaolt sarnase kirjapildi, mis võib olla takistuseks näiteks olemasoleva süsteemi mustrite kaardistamisel, sest klassikalises OO keeles on kasutusel liidesed ja erinevad eesliited, kuid javascriptis kirjeldatakse kõik läbi prototüüpide ja objektide.

4.3.12 Käsuliin (Chain of responsibility)

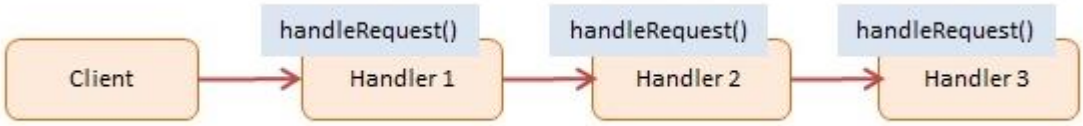
Käsuliin on laialt kasutatav muster javascriptiga. Veebipõhistes süsteemides võib näiteks tuua lingile vajutuse – me saame enne algse koodi käivitumist lisada samale nupule veel mitu funktsiooni. Nii tekib jada erinevaid funktsioone, mida käivitatakse kuni algse funktsionaalsuseni. [13] See tähendab, et peame leidma objektistruktuurist üleasande täitmiseks sobiva objekti.

Vaatleme kõigepealt antud lahendust C# keeles, mis on toodud allikast [11] lisas 10. Näeme, et iga objekt käitub loojana, sest ta on tihendalt seotud järgmise objektiga, mis viitab jällegi kõrgele sõltuvusele. Samas ülesande saatjaga on sõltuvus madal. Lisaks näeme ka GRASP kontrolleri printsiibile omaseid tunnuseid – esindab ülemobjekti. See tähendab, et kui päring saadetakse kõige alumisele otsustamiseks ning lõpliku otsuse teeb kõige ülemine, siis vastuse

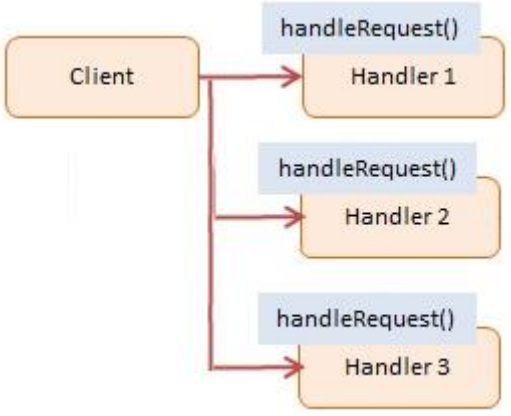
tagastab lõpuks kõige alumine ehk see kelle poole pöörduti. Siin näeme, et antud muster ei ole kuigi praktiline arvestades GRASP printsiipe.

Nüüd vaatleme allikas [13] toodud kohtusüsteemi, kus probleemi lahendust vaatlevad mitu inimest ning loogika on selline, et kui kõige esimene lahendust ei leia, siis suunatakse probleem järgmisele inimesele. Kood on lisas 11.

Nägime, et javascriptiga oli sarnane probleem lahendatud „for“ tsükliga. Loodud oli massiiv, kus olid erinevate rollidega objektid ning kui objekt sobis antud ülesannet täitma, siis ta seda ka tegi. Selle näite puhul ei ole näha sõnumite saatmise ahelat objektide vahel, mis tegelikult peaks olema antud mustri omadus. Võtame veel ühe näite, mis on toodud lisas 12 allikast [14], kus on püütud lahendada panga automaadi tööpõhimõtet arvestades erinevate kupüüridega. Lähemal vaatlusel selgub jällegi, et ei ole sarnasust jadaga nagu on toodud samas allikas UML diagrammil, mis on joonisel 28. Lahendust vaadates võib UML diagrammi ümber teha selliseks nagu on joonisel 29.



Joonis 28. Käsuliin



Joonis 29. Kohandatud Käsuliin

Kasutades sama lahendust on võimalik seda kohandada käituma rekursiivselt nii, et lahendus vastaks UML diagrammile, kus „Handler 2“ välja kutsuja oleks „Handler 1“. Javascriptis on selleks jällegi mitmeid võimalusi, kuid üritame teha seda võimalikult lühidalt (Joonis 30).

```
var Request = function (amount) {
    this.amount = amount;
    this.Bills = [100, 50, 20, 10, 5, 1];
    this.CheckedBills = [];
    console.log("Requested: $" + amount + "\n");
};
Request.prototype = {
    getWithChain: function () {
        for (var i = 0; i < this.Bills.length; i++) {
            if (this.CheckedBills[i] != this.Bills[i]) {
                var count = Math.floor(this.amount / this.Bills[i]);
                this.amount -= count * this.Bills[i];
                console.log("Dispense " + count + " $" + this.Bills[i] + "
                bills");
                this.CheckedBills[i] = this.Bills[i];
                if (this.amount > 0) {
                    this.getWithChain();
                }
            }
        }
        return this;
    }
};

var request = new Request(378);
request.getWithChain();
```

Joonis 30. Käsuliin kohandatud lahendus

4.3.13 Käsk (Command)

Käsu muster hõlmab endas nii parameetrite kui ka objekti seisu ning välja kutsutava meetodi kapseldamist. See muster lubab koguda kogu info selleks, et hiljem oleks võimalik meetod uuesti välja kutsuda. See on kasutusel näiteks „undo“ ja „redo“ funktsioonide realiseerimisel. [13]

Võtame aluseks kalkulaatori näite, kus on neli põhifunktsiooni – liitmine, lahutamine, korrutamine ja jagamine. Kui meil on vajadus mõni samm tagasi astuda, siis on see lihtsasti realiseeritav lisas 13. Näeme, kuidas on võimalik kapseldada funktsioone andes need erinevatele käskudele sisenditena, seejärel välja kutsuda käske, mitte funktsioone. See paistab nagu oleks silla realisatsioon, kus me suuname tegevusi edasi ning realisatsioon on madala sõltuvusega, sest kui me tahame lisada uut funktsionaalsust, siis tuleb lisada vaid uus kask, mis ei mõjuta teiste käskude tööd. Samas on märgata ka seda, et me peame kohe defineerima käsule ka „undo“ funktsionaalsuse. Sama loogikat on tegelikult võimalik realiseerida ka teisiti

kogudes muudatuste ajalugu massiivi ja hiljem neid muudatusi vastupidiselt käivitada, kuid see kaotaks käsu mustri struktuuri. Selgub ka tõsiasi, et javascriptiga on seda mustrit realiseerida lihtsam, kuna javascript lubab meetodi sisendina edastada funktsioone ja nii ei vaja objektid täiendavat abstraktsiooni, mis oleks nõutud klassikalise OO keele puhul. Samas alati ei ole võimalik teha kõikvõimalikke „undo“ funktsioone ning sellest tuleb juttu täpsemalt peatükis 4.3.17.

4.3.14 Interpretaator (Interpreter)

Interpretaatori muster erineb teistest mustritest oma olemuse poolest. See lubab luua oma keele ehk tõlgendada olemasoleva funktsionaalsuse kasutamist teisiti. Osades situatsioonides on vajadus kirjeldada lihtne oma keel, mis kirjeldab nõudmisi. Justnimelt lihtne, sest kui keel muutub keerulisemaks, siis selle eelised kaovad keerulise struktuuri taha. [13]

Vaatleme allikas [13] pakutud lahendust, mis on lisas 14. Lahendatud on mänguvälja ehitamine nii, et anname sisendina ainult teksti ning tagastatakse struktuur. Selline realisatsioon käitub loojana ja lihtsustab meie objektide loomise kirjapilti. Nagu öeldud ka allikas [13], on selle mustri realiseerimisel mitmeid viise ja ühte kindlat ei ole. Võrreldes ka ühe ja sama lähteülesande lahendust, allikas [11] pakutud lahendust C# keeles ja allika [14] javasciptiga lahendust, siis näeme kohe, et javascriptiga on realisatsioon tunduvalt lühem ja ülevaatlikum. Selle põhjal võib väita, et antud mustri kasutamine on javascriptiga mugavam, kuid selgub ka tõsiasi, et see ei ole populaarne muster ning selle reaalne kasutus on minimaalne.

4.3.15 Iteraator (Iterator)

Objektide massiivide läbivus on väga tavaline probleem, kus näiteks C# keel pakub „foreach“ funktsionaalsust. Iteraator kirjeldab lihtsa liidese, mida realiseerivad konkreetsed iteraatorid ning peavad järge milleni iteratsioon on hetkel jõudnud. [13]

Lihtsamalt võib seda sõnastada seda kui eeldefineeritud „for“ tsüklit, tegelikult ta sisaldabki endas „for“ tsüklit nagu on toodud joonisel 31.

```
var Iteraator = function (kirjed) {
    this.indeks = 0;
    this.Kirjed = kirjed;
}
Iteraator.prototype = {
    foreach: function (callback) {
        for (var kirje = this.Kirjed[0]; this.indeks <= this.Kirjed.length ;
             kirje = this.Kirjed[this.indeks++]) {
            callback(kirje);
        }
    }
}
var massiiv = ["kirje 1", 2, "kolm", true, "viies kirje"];
var iteraator = new Iteraator(massiiv);
//Meie loodud iteraator
iteraator.foreach(function (kirje) {
    console.log(kirje);
});
//ECMAScript 6 iteraator
for (var kirje of massiiv) {
    console.log(kirje);
};
//Väljund
// kirje 1
// 2
// kolm
// true
// viies kirje
```

Joonis 31. Iteraator muster

Üritasime luua C# keeles oleva „foreach“ sarnase funktsiooni javascriptiga, mis tegelikult ongi iteraator. Selle kasutus on nii suur, et ECMAScript 6 standardis on kirjeldatud juba iteraatori funktsionaalsus nagu on näha joonisel 31. Selgub, et ülimalt kasulik, igapäevaselt kasutatav „for“ tsükkel ongi üks disainimuster, mida esmapilgul tähele ei panda ja javascriptiga ei ole selle implementeerimine probleem, seepärast seda pikemalt uurima ei hakka.

4.3.16 Vahendaja (Mediator)

Probleem seisneb selles, et on vaja panna hulk objekte omavahel suhtlema teatud reeglite järgi, vältides samal ajal sõltuvust objektide vahel. Selleks tuleb luua objekt, mis kapseldab objektide omavahelise suhtluse reeglid. Vahendaja mustri puhul säilib madala sõltuvuse printsiip ja ta laseb varieeruda suhtlusreeglitel sõltumata objektide tüüpidest. Vahendaja mustrit võib käsitleda ka kui äriloogika kirjeldamise kihti. Näiteks allikas [14] on kirjeldatud selle mustri kasutusvaldkonda kui lennupiletite reserveerimise veebilehte, kus on vaja sisestada korrektsed kuupäevad ja muu info ning vahendaja kontrollib seda. Vaatleme pakutud realisatsiooni allikast [14], mis on lisas 15.

Näeme, et vahendaja käitub kui adapter või sild, mis sisaldab endas teatud käitumisreegleid ja ühendab erinevate klasside suhtlemise. Selle realisatsioon javascriptiga on lihtsam kui C# realisatsioon allikas [11]. Selgub ka tõsiasi, et igapäevaselt kasutuses olev jQuery teekide kogumik käitub ka nagu vahendaja. Nagu on mainitud allikas [13] saab jQuery abil leida kõik samasse klassi kuuluvad elemendid ning neid muuta korraga ilma, et peaks iga elemendi poole eraldi pöörduma. Tundub, et see peaks sisaldama endas ka iteraatorit, millest oli juttu peatükis 4.3.15, sest kõigi elementide muutmine eeldab iteratsiooni nende hulgas. Selline muster peegeldab ka GRASP kontrolleri printsiipi, kus ta haldab sündmusi – antud näite puhul sõnumi saatmist osaliste vahel. Näeme, et nagu ka teised keeled, kasutab javascript teekide kogumikke, mis on igapäevatöö üks osa ning seega ei olegi javascript nii kuiv keel, vaid on samuti piiramatult laiendatav.

4.3.17 Memento (Memento)

Eelnevalt nägime peatükis 4.3.13 kuidas on võimalik luua „undo“ funktsionaalsust. Mainitud sai ka tõsiasi, et alati ei ole võimalik luua seda funktsionaalsus sellisel kujul nagu käsu muster seda pakub. Vaja on säilitada objekti olek ilma vana objekti kopeerimata, sest mõnikord hoiab meid huvitavat objekti olekut objekti atribuutide alamhulk ja seepärast ongi raske koopiale kõiki viiteid suunata. [13]

Sellele probleemile pakub Memento muster lahenduse – objekt peab väljastama oma olekule vastavaid objekte ja nende objektide abil peab saama hiljem ka oma olekut muuta. Vaatame lahendust C# keeles, mis on lisas 16. Näeme selgelt, et antud näites on võimalik salvestada ainult üht olekut, kuid see ei ole probleemiks. Proovime samalaadse mustri implementeerida javascriptiga täiendades seda nii, et oleks võimalik rohkem mementosid salvestada. Lahendus on kujutatud joonisel 32.

Joonisel 32 näeme, et toimub tavaline parameetrite säilitamine ehk kopeerimine eraldi massiivi. Kuna sama funktsionaalsus leiab aset peatükis 4.3.3, siis ei ole siin midagi innovatiivset ja võiks väita, et Memento on prototüübi mustri erivorm, kus ei kopeerita mitte objekti, vaid tema parameetreid.

```
var Originator = function() {this._state = null;}
Originator.prototype = {
    CreateMemento : function() {
        var memento = this._state;
        return memento;
    },
    SetMemento: function(memento) {
        this._state = memento;
    }
}
var CareTaker = function () {
    this.mementos = {};
    this.add = function (key, memento) {this.mementos[key] = memento;},
    this.get = function (key) {return this.mementos[key];}
}
var o = new Originator();
o._state = "On";
var caretaker = new CareTaker();
caretaker.add(1, o.CreateMemento());
o._state = "Off";
console.log("Enne oleku taastamist: " +o._state);
o.SetMemento(caretaker.get(1));
console.log("Pärast oleku taastamist: "+o._state);
//Väljund
//Enne oleku taastamist: Off
//Pärast oleku taastamist: On
```

Joonis 32. Memento

Kui seda lahendust kohandada ja kombineerida kärbeskaalu mustriga, siis oleks võimalik javascriptiga vägagi optimeeritult kirjutada „undo“ funktsionaalsust, sellepärast, et täiendus oleks vaja teha ainult „CareTaker“ funktsioonis (Justnimelt funktsioonis, sest me ei saa rääkida javascriptis klassidest). Arvestades asjaolu, et kliendipoolsetes süsteemides on „undo“ funktsionaalsus igati kasulik, siis see muster on täiesti praktiline ja vajalik selliste süsteemide loomisel.

4.3.18 Vaatleja (Observer)

Vaatleja muster on enimkasutatud muster javascripti maailmas. Seda kasutatakse üksiklehe veebirakenduste puhul [13]. Tihtipeale on objektid huvitaud mõne teise objekti muudatustest ja peavad reageerima muudatustele. Seejuures peab jääma sõltuvus objektide vahel madalaks – me ei taha, et nad oleksid tihedalt seotud. Vaatleme allikas [14] pakutud näidet (lisas 17), kus ühele nupu vajutusele peavad reageerima teatud funktsioonid. Näeme, et käivituvad ainult need funktsioonid, kes on registreeritud vaatlejateks.

Leiame ka seda, et sama funktsionaalsus on peidus ka eelnevalt mainitud jQuery teekide kogumikus. Kui me pöördume samasse klassi kuuluvate objektide poole muudatustega, siis käituvad need kõik objektid kui vaatlejad, kuid sel juhul nad reageerivad ühte moodi. Vaatleja eesmärk on pigem see, et iga vaatleja reageerib muudatustele omamoodi.

Oletame, et meil on veebipõhine mäng, kus on erinevad osalised ja igaüks ründab erineva relvaga. Sel juhul on vaatleja muster igati praktiline, kui me teatame, et meil on vaja minna ründele, siis iga vaatleja läheb ründama rünnates erinevalt. Siit tuleb nähtavale ka GRASP polümorfismi printsiip, mis eeldabki samale sõnumile erinevat reaktsiooni. Võrreldes selle mustri realiseerimist C# keelega, siis võib öelda, et javascriptiga on see jällegi lihtsam, sest jäävad ära abstraktsioonide ja liideste defineerimine.

4.3.19 Olek (State)

Olekud on igati kasulikud programmeerimises. Kahjuks nende kasutus on väike, sest paljud programmeerijad implementeerivad mõningaid probleeme kasutades hulk „if“ lauseid. Selle asemel on võimalik kasutada oleku mustrit. [13]

Siin tuleks defineerida jällegi liides, mida erinevad olekuklassid realiseerivad. Seejärel tuleks delegeerida olekust sõltuvad operatsioonid oleku objektile. Vaatame valgusfoori näidet joonisel 33.

Näeme joonisel 33, et fooritulede muutmisega tegeleb objekt „Foor“, mis käitub põhinedes GRASP printsiibile kontrollerina. Võrreldes antud lahendust jällegi C# keelega, on see tunduvalt lihtsamini realiseeritav, kuna me saame anda edasi ühte ja sama foori objekti erinevate tulede vahel. Märkame ka seda, et järgmise oleku annab ette olekuobjekt ise, mitte foor. C# keeles nõuaks see jällegi abstraktsiooni kasutust. Lisaks on võimalik seda implementeerida ka kasutades prototüüpe tänu javascripti paindlikkusele. Siin on näha ka mitmeid potentsiaalseid täienduste võimalusi – näiteks olnud olekute salvestamine, mille pookimine antud koodi oleks võrdlemisi lihtne, säilitades objektide vahel madala sõltuvuse.

```
var Foor = function() {
    var i = 0;
    this.muuda = function(olek) {
        if (i++ >= 4) return; //vältime lõpmatut rekursiooni
        olek.muuda();
    }
    this.start = function () {new Punane(this).muuda();};
}
var Punane = function (foor) {
    this.muuda = function () {
        console.log("Punane 100 sek");
        foor.muuda(new Roheline(foor));
    }
};
var Roheline = function (foor) {
    this.muuda = function () {
        console.log("Roheline 100 sek");
        foor.muuda(new Kollane(foor));
    }
};
var Kollane = function (foor) {
    this.muuda = function () {
        console.log("Kollane 10 sek");
        foor.muuda(new Punane(foor));
    }
};
new Foor().start();
//Väljund
// Punane 100 sek
// Roheline 100 sek
// Kollane 10 sek
// Punane 100 sek
// Roheline 100 sek
```

Joonis 33. Olek

4.3.20 Strateegia (Strategy)

Alati on võimalik kirjeldada ühte objekti mitmel erinval viisil ja sama kehtib programmeerimises. Oletame, et meil on vaja leida oma asukoht. Nutitelefonil on selleks kolm erinevat võimalust – GPS (Global Positioning System), telefoni triangulatsioon või wifi. Erinevad meetodid kasutavad erinevalt telefoni ressursse. Strateegia muster aitab valida nende vahel. C# keeles on võimalik defineerida liides või abstraktne klass. Javascriptiga peame looma prototüübi. Vaatleme järgnevalt antud näite realisatsiooni joonisel 34.

```
var Asukoht = function () {};
Asukoht.prototype = {
    valiStrateegia: function (variant) {
        this.Variant = variant;
    },
    leiaAsukohaMaksumus: function () {
        return this.Variant.arvuta();
    }
};
var WIFI = function () {
    this.arvuta = function () {return 15;}
};
var GPS = function () {
    this.arvuta = function () {return 20;}
};
var TRING = function () {
    this.arvuta = function () {return 10;}
};
var asukoht = new Asukoht();
asukoht.valiStrateegia(new WIFI());
console.log("wifi strateegiat kasutades maksumus: " +
            asukoht.leiaAsukohaMaksumus());
asukoht.valiStrateegia(new GPS());
console.log("gps strateegiat kasutades maksumus: " +
             asukoht.leiaAsukohaMaksumus());
asukoht.valiStrateegia(new TRING());
console.log("tring strateegiat kasutades maksumus: " +
            asukoht.leiaAsukohaMaksumus());
//Väljund
// wifi strateegiat kasutades maksumus: 15
// gps strateegiat kasutades maksumus: 20
// tring strateegiat kasutades maksumus: 10
```

Joonis 34. Strateegia

Joonis 34 on ülimalt sarnane abstraktsiooni kirjeldusega. Anname sisendiks asukoha arvutamise abstraktsiooni ja arvutamisel tagastatakse vastav väärtus. Näib, et strateegia muster ja abstraktsiooni on võimalik implementeerida täpselt samuti javascriptiga. Lisaks on näha ka sarnasust ehitaja mustriga, kus erinevad ehitajad tagastasid erinevaid objekte. Antud näite puhul tagastati samuti erinevaid väärtusi, mis võiksid tegelikult olla ka objektid. Tundub, et strateegia muster ei olegi piisavalt põhjendatud muster javascriptiga realiseerimisel vaid pigem peegeldab endas juba olemasolevaid lahendusi, sest näiteks C# keeles oleks tehase ja strateegia mustri vahe märgatav.

4.3.21 Šabloonmeetod (Template method)

Strateegia muster lubab vahetada välja tervet algoritmi mingisuguse lahenduse leidmisel, nagu oli näha peatükis 4.3.20. Tihtipeale on terve algoritmi vahetus liig, sest osa algoritmist võib kattuda teiste algoritmidega. Šabloonmeetod lubab algoritmidel jagada mingit ühisosa ning erinevused implementeerida eraldi. [13]

Vaatleme näidet C# keeles allikast [11], kus on vajalik suhtlemine andmebaasiga, lisas 18, kus näeme, et ühenduse loomine on jagatud funktsionaalsus. Proovime sama implementeerida kasutades javascripti joonisel 35.

```
var DataAccessObject = {
    Run: function () {
        this.connect();this.select();this.process();
        this.disconnect();
        return true;
    },
    connect: function () { console.log("Ühendamine"); },
    disconnect:function () { console.log("Ühenduse lõpp"); }
};
function Categories(prototyyp) {
    var fn = function() {};
    fn.prototype = prototyyp; return new fn();
}
var kategooriad = new Categories(DataAccessObject);
kategooriad.select = function () {console.log("Select lause");}
kategooriad.process = function() {console.log("Töötlemine");}
kategooriad.Run();
```

Joonis 35. Šabloonmeetod

Joonisel 35 kujutatud lahenduses näha jällegi tuttavat abstraktsiooni kasutamist. Seejuures „Categories“ funktsioon käitub nagu pärimine. Kuna javascriptis pärimise mõistet klassikalisel kujul ei eksisteeri, siis on võimalik imiteerida seda, seega antud näites tuli välja väga hea alternatiiv pärimise lahendusele. Kui aga vaadata selle lahenduse negatiivset poolt, siis on siinjuures samuti nagu ka abstraktsiooni puhul vaja teada, millised meetodeid implementeeridaja ja siin on pärimise tõttu kõrge sõltuvus objektide vahel. Teisalt, on võimalik ka üle kirjutada olemasolevaid ühiskasutuses olevad meetodeid – ühendamine ja ühenduse sulgemine, mis lisab paindlikkust.

4.3.22 Külastaja (Visitor)

Külastaja muster lubab teatud klassi hierarhiale lisada operatsioone nii, et me operatsioone klassidesse ei defineeri kõrge kokkukuuluvuse säilitamiseks. Samuti nagu ka eelnevates mustrites on võimalus siin kirjutada pikk „if“ lausete kogu, kuid sellel ei ole mõtet. Idee seisneb ühise liidese loomises, mis võimaldab objektidel võtta vastu külalisi ja neid teavitada, mis objektiga on tegu. [13]

Element mida külastatakse kutsub välja oma defineeritud külastamise meetodi ja annab end argumendina kaasa, et külastaja teaks kellega on tegu. Nii ei pea me eristama erinevaid tüüpi objekte vaid teame kohe objekti tüüpi.

```
var myyja =   (function() {
    function myyja() {this.Tyyp = "myyja";}
    myyja.prototype.AnnaNimetus = function () {console.log("myyja");}
    myyja.prototype.kylasta = function (kylastaja) {kylastaja.kylasta(this);}
    return myyja;
})();
var koristaja = (function() {
    function koristaja() {this.Tyyp = "koristaja";}
    koristaja.prototype.AnnaNimetus = function() {console.log("koristaja");}
    koristaja.prototype.kylasta = function(kylastaja)
    {kylastaja.kylasta(this);}
    return koristaja;
})();
var Kylastaja = (function () {
    function Kylastaja() {}
    Kylastaja.prototype.kylasta = function(tootaja) {
        if (tootaja.Tyyp == "koristaja") { tootaja.AnnaNimetus(); }
        else { console.log("ei ole koristaja"); }
    };
    return Kylastaja;
})();
var tootajad = [];
tootajad.push(new koristaja());
tootajad.push(new myyja());
var visitor = new Kylastaja();
for (var i = 0; i < tootajad.length; i++) {
    tootajad[i].kylasta(visitor);
}
//Väljund
// koristaja
// ei ole koristaja
```

Joonis 36. Külastaja [13]

Joonisel 36 kujutatud lahenduse puhul on näeme, et tüübi tuvastamise funktsionaalsus on „Kylastaja“ objektis, mis teeb sõltuvuse objektide vahel madalamaks. Näha on ka seda, et defineerisime külastatavatele tüübi parameetri, sest kui me peaksime kasutama „typeof“ või „instanceof“ meetodeid, siis pruugi antud lahendus töötada olenevalt objektide loomisest [13]. Selles lahenduses on näha ka sarnasust C# keelega, kus antakse külastajale täpselt samuti kaasa parameetrina külastatav ise, seega peale abstraktsiooni erinevuse siin muid erinevusi ei ole. Allika [14] põhjal võib öelda, et see on vähekasutatav muster ning allikas [13] isegi soovitab mitte kasutada seda, sest selle tööle saamise nõudmised on keerulised ja ebaselged.

5. Tulemused ja järeldused

Eesmärgiks oli katta kõik GOF disainimustrid javascripti lahendustega ning hinnata nende headust põhinedes GRASP printsiipidele. Juba peatükis 3.1, kus olid kirjeldatud OOP põhimõtted kasutades javascripti, nägime, et javascript erineb klassikalistest keeltest. Samas selgus ka see, et javascriptiga on võimalik järgida kõiki OOP põhimõtteid, kuid teatud mööndustega. Lisaks oli näha ka javascripti suurt paindlikkust, mis võimaldab kirjeldada OOP põhimõtteid mitut erinevat moodi, mille tõttu tundus javascript ebapraktiline lähtudes OOP põhimõtetest. Osaliselt ongi javascript ebapraktiline, kuna implementeerimisel võib kergesti vigu tekkida ja näiteks liideste kirjeldamiseks tuleb kasutada abstraktsiooni, kus tuleb teada täpselt kuidas seda kirjeldada javascriptiga. Järgnevalt peatükis 4 oli tutvustatud GOF disainimustreid ja GRASP printsiipe.

GOF disainimustreid on kokku 23 ja kõik disainimustrid said kaetud javascripti implementatsiooniga. Lihtsamad lahendused on autori poolt koostatud ning hinnatud ka enda pakutud lahendusi vaadates, et säiliksid erinevad GRASP printsiibid. Üldiselt olid implementeerimisel aluseks võetud erinevad allikad ning seal leiduvad olemasolevad lahendused. Olemasolevate lahenduste puhul oli hinnatud mitte ainult lahendust, vaid üritatud leida ka alternatiive. Näiteks „singel“ muster oli lahendatud umbes 30 realise koodiga, kuid tegelikult see täitis globaalse muutuja rolli, mis ongi „singel“ mustri tunnus.

Lisaks hindamisele oli vaadatud ka pakutud lahenduste erinevusi ja realisatsiooni struktuuri, kus osad lahendused ei vastanud samas allikas pakutud joonistele. Näiteks „käsuliini“ mustrit oli vaja kohandada vastavalt joonisele, et seal säiliks rekursiivne käitumine. Erinevates allikates oli näha ka erinevaid lahenduse struktuure, mis viitas jällegi javascripti paindlikkusele.

Tänu javascripti paindlikkusele said kõik GOF disainimustrid kaetud realisatsioonidega. Selgus, et võrreldes klassikalise OOP keelega, antud töös võrdluses oleva C# keelega, on javascriptiga võimalik implementeerida igat disainimustrit mitmel erineval moel, mida C# keel ei luba. Viimaste disainimustrite vaatlemisel hakkas välja paistma ka ülimalt sarnane kirjapilt eelnevate mustritega.

Kuna javascript on prototüübipõhine keel, siis klasse ei eksisteeri javascripti kontekstis, seepärast oli vaja kirjeldada kõiki lahendusi kasutades objekte, prototüüpe ja muutujaid. See oli ka üks põhjus, miks erinevad lahendused olid sarnased. Viimaste lahenduste puhul oli tegelikult näha ka seda, et kombineerides erinevaid mustreid, on mõnel juhul võimalik javascriptiga lahendada mõnda probleemi isegi optimaalsemalt kui klassikalise OOP keelega.

Esmapilgul ebamugavana tundunud keel on tegelikult väga paindlik oma olemuselt ning kui seda õigesti kasutada, siis on võimalik kirjeldada vägagi keerulisi probleeme. Vaadates aga veebiarendajate vaatepunktist, kes arendavad tavalisi infosüsteeme, tundub aga vajadus mustrite järele üsna väike, sest üldjuhul kasutatakse javascripti mingisuguse kastikese värvimiseks või kinnitussõnumi küsimiseks. Veebiarenduses on javascript kirjutatud üldjuhul ka ühte faili, mille tõttu javascripti kood ei ole nii ülevaatlik ja see raskendab ka rakenduse kirjutamist. Seepärast võiks edasiseks uurimiseks kaardistada kohad, kus on disainimustrite kasutamine vajalik ning uurida ka seda, et kui palju Javascripti kasutamisel järgitakse disainimustreid..

Kokkuvõte

Antud töö eesmärgiks oli lühidalt tutvustada erinevaid disainimustreid ja hinnata nende praktilist kasutust javascriptiga ning uurida kas kõiki disainimustreid on võimalik implementeerida kasutades javascripti. Kui mitte, siis pakkuda alternatiive ning uurida alternatiive ka juba pakutud lahendustele.

Töö käigus võrreldi disinimustrite kasutamist klassikalise keele C# ja javascripti vahel tuginedes GRASP printsiipidele. Tulemustena toodi välja erinevad lahendused ning mõnel juhul ka alternatiivid. Lisaks leiti ka mõningad javascripti huvitavad omadused nagu näiteks „call-by-sharing“. Lõpptulemusena sai kaetud kõik GOF disainimustrid javascripti realisatsiooniga.

OOP kirjeldamine javascriptiga tundus esmapilgul ebamugav ja ebapraktiline, sest seal ei ole tuttavaid klasse ja liideseid. Töö käigus aga selgus, et on olemas erinevad teekide kogumikud, mis isegi võimaldavad luua liideseid. Nägime ka seda, et javascriptiga on võimalik imiteerida klassile omast käitumist ja mitmel erineval viisil.

Probleemi püstitamisel eeldati skeptiliselt, et javascriptiga ei ole võimalik kõiki disainimustreid realiseerida, seega tulemus oli oodatust natuke erinev, kuid tunduvalt parem. Tänu javascripti paindlikkusele selgus tõsiasi, et antud keelega on isegi rohkem kui üks võimalust iga disainimustri implementeerimiseks. Negatiivse poole pealt vaadates oli näha seda, et osade mustrite realisatsioon kattus, kuid arvestades asjaolu, et kõik oli kirjeldatud funktsioonide, prototüüpide ja muutujate abil, siis javascript ei ole sugugi halvem klassikalistest OO keeltest.

Käesoleva töö käigus on uuritud ainult disainimustrite realiseerimise võimalusi, kuid ei ole täpsemalt uuritud erinevaid kasutusvaldkondi. Edasiseks uurimiseks võiks püstitada järgmised küsimused:

Millises valdkonnas on vajadus disainimustrite kasutamise järele javascriptiga kõige suurem?

Kui palju praegusel ajal javascriptiga disainimustreid järgitakse?

Summary

The purpose of the thesis is to introduce different design patterns and evaluate their practical usage with javascript and research the possibility to implement all design patterns with javascript. If not so, the author will propose new alternatives and evaluate previously proposed alternatives.

The usage of design patterns between classical C# and javascript programming languages based on GRASP principles are compared in the thesis. As a result the author has formulated different solutions and in some cases new alternatives. In addition, some interesting javascript features were found, for example „call-by-sharing“. Ultimately all design patterns with javascript implementation were covered.

Describing OOP with javascript may initially be impractical and unfamiliar, because there are no familiar classes and interfaces. However, it became clear in the thesis that there are different library collections which allow creating interfaces. It is possible to imitate the class behavior in different ways using javascript.

When the main question of the thesis was composed, it was anticipated that with javascript it is impossible to implement all design patterns. But the result was more positive than expected. It turned out that javascript has more than one ways for each design pattern to implement thanks to the flexibility. On the negative side, the implementation of some patterns was practically the same, but considering the fact that all the patterns were described by using functions, prototypes and variables, javascript is not inferior to the other classic OO languages.

In the thesis only the implementation possibilities of design patterns were researched. Different areas of usage were not explored. Therefore, for further research in the field the following questions are composed:

Where lays the highest need for usage of design patterns with javascript?

How high is the actual usage of design patterns with javascript at the present time?

Kasutatud kirjandus

[1] M. Fowler, Refactoring: Improving the Design of Existing Code.

[2] J. Pudrum, Beginning Object Oriented Programming with C#.

[3] D. J. Armstorng, The Quarks of Object—Oriented Development.

[4] C. Thomas, An Introduction to Object-Oriented Programming with Java.

[5] M. Weisfeld, Object-Oriented Thought Process, 3rd Edition.

[6] „Polymorphism (C# Programming Guide),“ [Võrgumaterjal]. Available: https://msdn.microsoft.com/en-us/library/ms173152.aspx.

[7] D. Salomon, Assemblers and Loaders (1993).

[8] „A Short History of JavaScript,“ [Võrgumaterjal]. Available: https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript.

[9] M. Vézina, „Javascript prototypes as abstract objects,“ 10 12 2012. [Võrgumaterjal]. Available: http://lab.la-grange.ca/en/javascript-prototypes-as-abstract-objects.

[10] „How to Implement Interfaces in JavaScript,“ 19 09 2010. [Võrgumaterjal]. Available: http://www.javascriptbank.com/how-implement-interfaces-in-javascript.html.

[11] „.NET Design Patterns,“ [Võrgumaterjal]. Available: http://www.dofactory.com/net/design-patterns.

[12] D. Rao, „GRASP Design Principles,“ [Võrgumaterjal]. Available: http://www.cs.colorado.edu/~kena/classes/5448/f12/presentation-materials/rao.pdf.

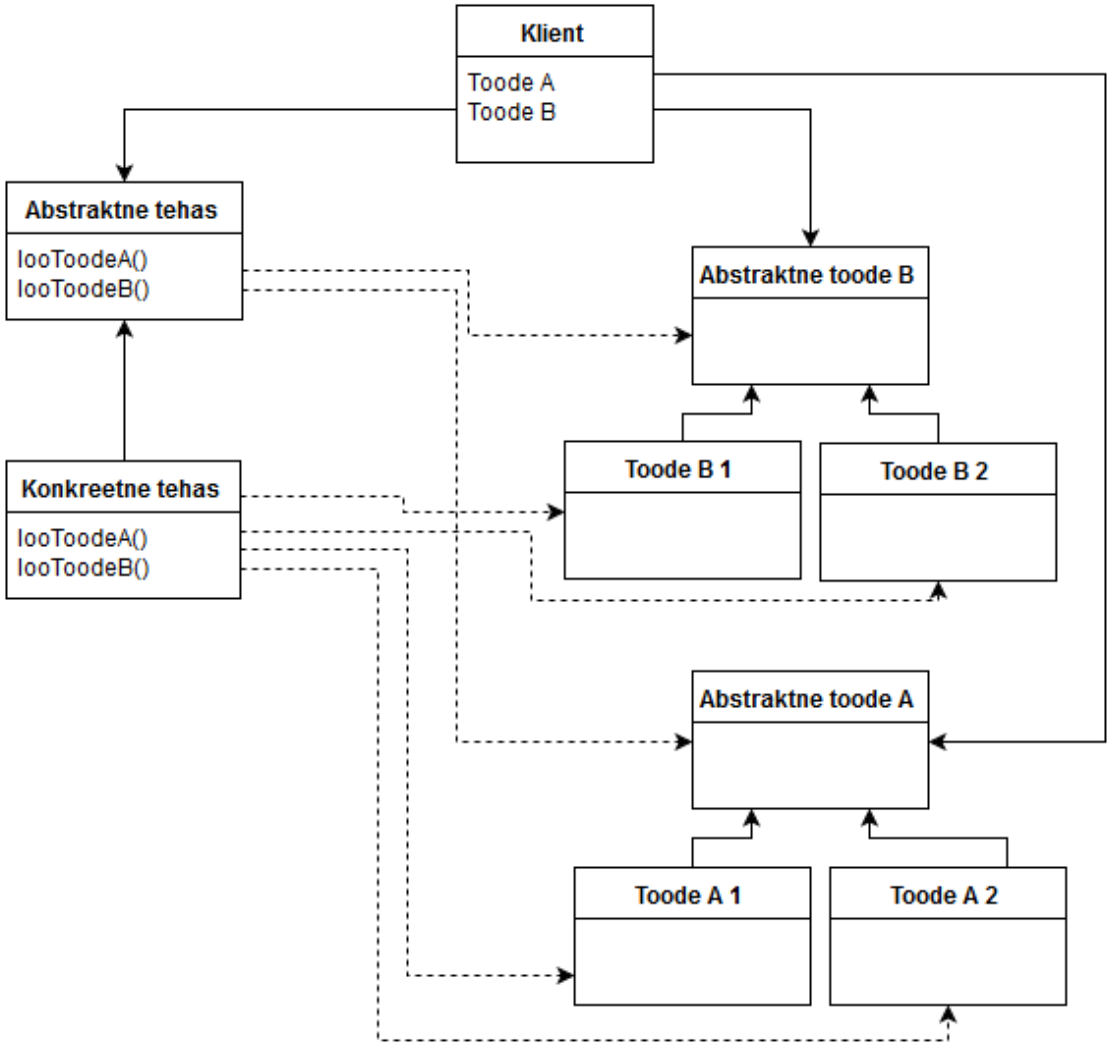
[13] J. Bishop, „C# 3.0 Design Patterns,“ [Võrgumaterjal]. Available: http://it-ebooks.info/book/202/.

[14] S. Timms, „Mastering JavaScript Design Patterns“.

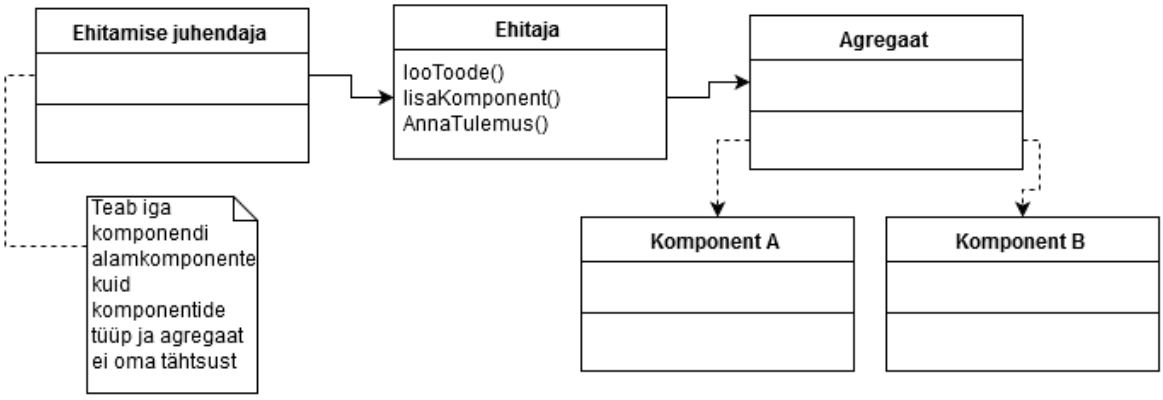
[15] „Javascript design patterns,“ [Võrgumaterjal]. Available: http://www.dofactory.com/javascript/composite-design-pattern.

[16] A. Allain, „Huffman Encoding Compression Algorithm,“ [Võrgumaterjal]. Available: http://www.cprogramming.com/tutorial/computersciencetheory/huffman.html.

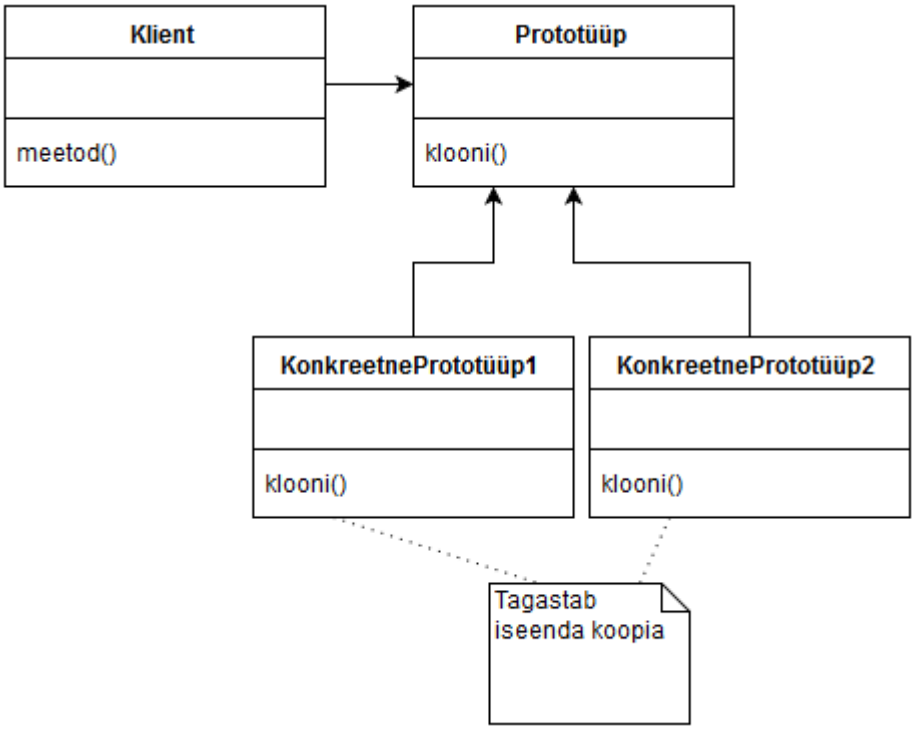
Lisa 1. Abstraktne tehas



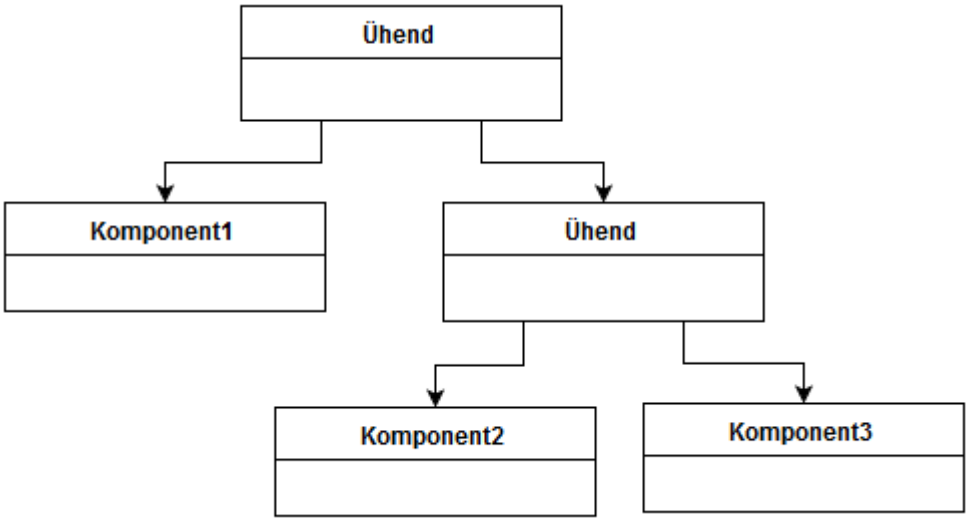
Lisa 2. Ehitaja



Lisa 3. Prototüüp



Lisa 4. Ühend



Lisa 5. Ühendi näide [14]

```
    var Node = function (name) {
        this.children = [];
        this.name = name;
    }
    Node.prototype = {
        add: function (child) {
            this.children.push(child);
        },
        remove: function (child) {
            var length = this.children.length;
            for (var i = 0; i < length; i++) {
                if (this.children[i] === child) {
                    this.children.splice(i, 1);
                    return;
                }
            }
        },
        getChild: function (i) { return this.children[i];},
        hasChildren: function () { return this.children.length > 0; }
    }
    // recursively traverse a (sub)tree
    function traverse(indent, node) {
        console.log(Array(indent++).join("--") + node.name);

        for (var i = 0, len = node.children.length; i < len; i++) {
            traverse(indent, node.getChild(i));
        }
tree }
    var      = new Node("root");
    var left = new Node("left")
    var right = new Node("right");
    var leftleft = new Node("leftleft");
    var leftright = new Node("leftright");
    var rightleft = new Node("rightleft");
    var rightright = new Node("rightright");
    tree.add(left);
    tree.add(right);
    tree.remove(right);  // note: remove
    tree.add(right);
    left.add(leftleft);
    left.add(leftright);
    right.add(rightleft);
    right.add(rightright);
    traverse(1, tree);
    //Väljund
    //root
    // --left
    // ----leftleft
    // ----leftright
    // --right
    // ----rightleft
    // ----rightright
```

Lisa 6. Ühendi näide allikast [13]

```
 var SimpleIngredient = (function () {
    function SimpleIngredient(name, calories, ironContent,
    vitaminCContent) {
        this.name = name;
        this.calories = calories;
        this.ironContent = ironContent;
        this.vitaminCContent = vitaminCContent;
    }
    SimpleIngredient.prototype.GetName = function () {
        return this.name;
    };
    SimpleIngredient.prototype.GetCalories = function () {
        return this.calories;
    };
    SimpleIngredient.prototype.GetIronContent = function () {
        return this.ironContent;
    };
    SimpleIngredient.prototype.GetVitaminCContent = function () {
        return this.vitaminCContent;
    };
    return SimpleIngredient;
})();

var CompoundIngredient = (function () {
    function CompoundIngredient(name) {
        this.name = name;
        this.ingredients = new Array();
    }
    CompoundIngredient.prototype.AddIngredient =
    function (ingredient) {
        this.ingredients.push(ingredient);
    };
    CompoundIngredient.prototype.GetName = function () {
        return this.name;
    };

    CompoundIngredient.prototype.GetCalories = function () {
        var total = 0;
        for (var i = 0; i<this.ingredients.length; i++) {
            total += this.ingredients[i].GetCalories();
        }
        return total;
    };
    CompoundIngredient.prototype.GetIronContent = function () {
        var total = 0;
        for (var i = 0; i<this.ingredients.length; i++) {
            total += this.ingredients[i].GetIronContent();
        }
        return total;
    };
    CompoundIngredient.prototype.GetVitaminCContent = function () {
        var total = 0;
        for (var i = 0; i<this.ingredients.length; i++) {
            total += this.ingredients[i].GetVitaminCContent();
        }
```

```
        return total;
    };
    return CompoundIngredient;
})();

var egg = new SimpleIngredient("Egg", 155, 6, 0);
var milk = new SimpleIngredient("Milk", 42, 0, 0);
var sugar = new SimpleIngredient("Sugar", 387, 0, 0);
var rice = new SimpleIngredient("Rice", 370, 8, 0);
var ricePudding = new CompoundIngredient("Rice Pudding");
ricePudding.AddIngredient(egg);
ricePudding.AddIngredient(rice);
ricePudding.AddIngredient(milk);
ricePudding.AddIngredient(sugar);
console.log("A serving of rice pudding contains:");
console.log(ricePudding.GetCalories() + " calories");
//Väljund
//A serving of rice pudding contains:
//954 calories
```

Lisa 7. Dekoraatori näide allikast [13]

```
var BasicArmor = (function () {
    function BasicArmor() {
    }
    BasicArmor.prototype.CalculateDamageFromHit = function (hit) {
        return 1;
    };
    BasicArmor.prototype.GetArmorIntegrity = function () {
        return 1;
    };
    return BasicArmor;
})();
var ChainMail = (function () {
    function ChainMail(decoratedArmor) {
        this.decoratedArmor = decoratedArmor;
    }
    ChainMail.prototype.CalculateDamageFromHit = function (hit) {
        hit.Strength = hit.Strength * .8;
        return this.decoratedArmor.CalculateDamageFromHit(hit);
    };
    ChainMail.prototype.GetArmorIntegrity = function () {
        return .9 * this.decoratedArmor.GetArmorIntegrity();
    };
    return ChainMail;
})();

//To make use of this armor, you simply use the following code:
var armor = new ChainMail(new BasicArmor());
console.log(armor.CalculateDamageFromHit({Location: "head",
    Weapon: "Sock filled with pennies", Strength: 12
}));
//Väljund
//1
```

Lisa 8. Kärbeskaalu näide allikast [13]

We can see the simple set of fields in the following code:

```
var Soldier = (function () {
        function Soldier() {
        this.Health = 10;
        this.FightingAbility = 5;
        this.Hunger = 0;
        }
        return Soldier;
})();
```

Of course, with an army of 10,000 soldiers, keeping track of all of this requires quite some memory. Let's take a different approach and use prototypes:

```
var Soldier = (function () {
        function Soldier() { }
        Soldier.prototype.Health = 10;
        Soldier.prototype.FightingAbility = 5;
        Soldier.prototype.Hunger = 0;
        return Soldier;
})();
```

Using this approach, we are able to defer all requests for the soldier's health to the prototype. Setting the value is easy too, as shown in the following code:

```
var soldier1 = new Soldier();
var soldier2 = new Soldier();
console.log(soldier1.Health); //10
soldier1.Health = 7;
console.log(soldier1.Health); //7
console.log(soldier2.Health); //10
delete soldier1.Health;
console.log(soldier1.Health); //10
```

You'll note that we make a call to remove the property override and return the value back to the parent value.

Lisa 9. Esindaja mustri näide allikast [14]

```
function GeoCoder() {
this.getLatLng = function(address) {
if (address === "Amsterdam") {return "52.3700° N, 4.8900° E";} else if (address ===
"London") {return "51.5171° N, 0.1062° W";} else if (address === "Paris") {
return "48.8742° N, 2.3470° E";} else if (address === "Berlin") {return "52.5233° N,
13.4127° E";} else {return "";}};
}
function GeoProxy() {
    var geocoder = new GeoCoder();
    var geocache = {};
    return {
           getLatLng: function(address) {
                     if (!geocache[address]) {
                            geocache[address] = geocoder.getLatLng(address);
                     }
               log.add(address + ": " + geocache[address]);
               return geocache[address];
           },
           getCount: function() {
                  var count = 0;
                  for (var code in geocache) { count++; }
                  return count;
           }
    };
};
var log = (function() {
                  var log = "";
                  return {
                  add: function(msg) { log += msg + "\n"; },
                  show: function() { alert(log); log = ""; }
           }
})();
function run() {
var geo = new GeoProxy();
geo.getLatLng("Paris");
geo.getLatLng("London");
geo.getLatLng("Amsterdam");
geo.getLatLng("Amsterdam");
geo.getLatLng("Amsterdam");
geo.getLatLng("Amsterdam");
geo.getLatLng("London");
geo.getLatLng("London");

log.add("\nCache size: " + geo.getCount());
log.show();
}
```

Lisa 10 Käsuliini näide allikast [11]

```
class MainApp
{
  /// <summary>
  /// Entry point into console application.
  /// </summary>
  static void Main()
  {
    // Setup Chain of Responsibility
    Handler h1 = new ConcreteHandler1();
    Handler h2 = new ConcreteHandler2();
    Handler h3 = new ConcreteHandler3();
    h1.SetSuccessor(h2);
    h2.SetSuccessor(h3);
    // Generate and process request
    int[] requests = { 2, 5, 14, 22, 18, 3, 27, 20 };
    foreach (int request in requests)
    {
      h1.HandleRequest(request);
    }
    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Handler' abstract class
/// </summary>
abstract class Handler
{
  protected Handler successor;
  public void SetSuccessor(Handler successor)
  {
    this.successor = successor;
  }
  public abstract void HandleRequest(int request);
}
```

```
  class ConcreteHandler1 : Handler
  {
    public override void HandleRequest(int request)
    {
      if (request >= 0 && request < 10)
      {
        Console.WriteLine("{0} handled request {1}",
          this.GetType().Name, request);
      }
      else if (successor != null)
      {
        successor.HandleRequest(request);
      }
    }
  }
  class ConcreteHandler2 : Handler
  {
    public override void HandleRequest(int request)
    {
      if (request >= 10 && request < 20)
      {
        Console.WriteLine("{0} handled request {1}",
          this.GetType().Name, request);
      }
      else if (successor != null)
      {
        successor.HandleRequest(request);
      }
    }
  }

  class ConcreteHandler3 : Handler
  {
    public override void HandleRequest(int request)
    {
      if (request >= 20 && request < 30)
      {
        Console.WriteLine("{0} handled request {1}",
          this.GetType().Name, request);
      }
      else if (successor != null)
      {
        successor.HandleRequest(request);
      }
    }
  }
}
```

Lisa 11 näide allikast [13]

```
 var Complaint = (function () { function Complaint() {
            this.ComplainingParty = "";
            this.ComplaintAbout = "";
            this.Complaint = "";
}
return Complaint; })();
var ClerkOfTheCourt = (function () {
            function ClerkOfTheCourt() {
            }
ClerkOfTheCourt.prototype.IsAbleToResolveComplaint = function(complaint) {
            //decide if this is a complaint that can be solved by the clerk return false;
            };
ClerkOfTheCourt.prototype.ListenToComplaint = function(complaint) {
            //perform some operation
            //return solution to the complaint return "";
            };
return ClerkOfTheCourt;})();
JudicialSystem.ClerkOfTheCourt = ClerkOfTheCourt;
var King = (function () {
            function King() {}
            King.prototype.IsAbleToResolveComplaint = function (complaint) { return true; };
            King.prototype.ListenToComplaint = function (complaint) {
            //perform some operation
            //return solution to the complaint return "";
}; return King;})();
JudicialSystem.King = King;
var ComplaintResolver = (function () {
            function ComplaintResolver() {
                        this.complaintListeners = new Array();
                        this.complaintListeners.push(new ClerkOfTheCourt());
                        this.complaintListeners.push(new King());
            }
ComplaintResolver.prototype.ResolveComplaint = function(complaint) {
            for (var i = 0; i < this.complaintListeners.length; i++) {
                        if (this.complaintListeners[i]. IsAbleToResolveComplaint(complaint)) {
                                    return this.complaintListeners[i].ListenToComplaint(complaint);
                        }
            }
};
return ComplaintResolver;})();
```

Lisa 12 Käsuliini näide allikast [11]

```
var Request = function(amount) {
	this.amount = amount;
	log.add("Requested: $" + amount + "\n");
}

Request.prototype = {
	get: function(bill) {
		var count = Math.floor(this.amount / bill);
		this.amount -= count * bill;
		log.add("Dispense " + count + " $" + bill + " bills");
		return this;
	}
}
// log helper
var log = (function() {
	var log = "";
	return {
		add: function(msg) { log += msg + "\n"; },
		show: function() { alert(log); log = ""; }
	}
})();
function run() {
	var request = new Request(378);
	request.get(100).get(50).get(20).get(10).get(5).get(1);
	log.show();
}
```

Lisa 13 Käsu kohandatud näide allikast [13]

```
function liida(x, y) { return x + y; }
function lahuta(x, y) { return x - y; }
function korruta(x, y) { return x * y; }
function jaga(x, y) { return x / y; }
var Kask = function (nimi, kaivita, undo, vaartus) {
    this.Nimi = nimi;
    this.Kaivita = kaivita;
    this.Undo = undo;
    this.SisendVaartus = vaartus;
}
var LiidaKask = function (vaartus) {return new Kask("Liida",liida, lahuta,
                vaartus);};
var LahutaKask = function (vaartus) {return new Kask("Lahuta",lahuta, liida,
                 vaartus);};
var KorrutaKask = function (vaartus) {return new Kask("Korruta",korruta, jaga,
                  vaartus);};
var JagaKask = function (vaartus) {return new Kask("Jaga",jaga, korruta,
               vaartus);};
var Kalkulaator = function () {
    var hetkevaartus = 0;
    var kasud = [];
    return {
        kaivita: function (kask) {
            hetkevaartus = kask.Kaivita(hetkevaartus, kask.SisendVaartus);
            kasud.push(kask);
            console.log(kask.Nimi + ": " + kask.SisendVaartus);
        },
        tagasi: function () {
            var kask = kasud.pop();
            hetkevaartus = kask.Undo(hetkevaartus, kask.SisendVaartus);
            console.log("Undo " + kask.Nimi + ": " + kask.SisendVaartus);
        },
        AnnaVaartus: function () {return hetkevaartus;}
    }
}
var kalkulaator = new Kalkulaator();
kalkulaator.kaivita(new LiidaKask(100));
kalkulaator.kaivita(new LahutaKask(24));
kalkulaator.kaivita(new KorrutaKask(6));
kalkulaator.kaivita(new JagaKask(2));
kalkulaator.tagasi();
kalkulaator.tagasi();
console.log("Lõplik väärtus: " + kalkulaator.AnnaVaartus());
//Väljund
//Liida: 100
//Lahuta: 24
//Korruta: 6
//Jaga: 2
//Undo Jaga: 2
//Undo Korruta: 6
//Lõplik väärtus: 76
```

Lisa 14 Interpretaatori näide allikast [13]

```
var Battle = (function () {
    function Battle(battleGround, agressor, defender, victor) {
        this.battleGround = battleGround;
        this.agressor = agressor;
        this.defender = defender;
        this.victor = victor;
    }
    return Battle;
})();

var Parser = (function () {
    function Parser(battleText) {
        this.battleText = battleText;
        this.currentIndex = 0;
        this.battleList = battleText.split("\n");
    }
    Parser.prototype.nextBattle = function () {
        if (!this.battleList[0])
            return null;
        var segments = this.battleList[0].
        match(/\((.+?)\s?->\s?(.+?)\s?<-\s?(.+?)\s?->\s?(.+)/);
        return new Battle(segments[2], segments[1], segments[3], segments[4]);
    };
    return Parser;
})();

var text = "(Robert Baratheon -> River Trident <- RhaegarTargaryen) -> Robert
            Baratheon";
var p = new Parser(text);
console.log(p.nextBattle());
//Väljund
//Battle {battleGround: "River Trident", agressor: "Robert Baratheon",
  defender: "RhaegarTargaryen)", victor: "Robert Baratheon"}
```

Lisa 15 Vahendaja näide allikast [14]

```
var Participant = function (name) {this.name = name; this.chatroom = null;
};
Participant.prototype = {
    send: function (message, to) {this.chatroom.send(message, this, to);},
    receive: function (message, from) {
        console.log(from.name + " to " + this.name + ": " + message);
    }
};
var Chatroom = function () {
    var participants = {};
    return {register: function (participant) {
            participants[participant.name] = participant;
            participant.chatroom = this;
        },
        send: function (message, from, to) {
            if (to) { to.receive(message, from);// single message
            } else {  // broadcast message
                for (key in participants) {
                    if (participants[key] !== from) {
                        participants[key].receive(message, from);
                    }
                }
            }
        }
    };
};
var yoko = new Participant("Yoko");
var john = new Participant("John");
var paul = new Participant("Paul");
var ringo = new Participant("Ringo");
var chatroom = new Chatroom();
chatroom.register(yoko);
chatroom.register(john);
chatroom.register(paul);
chatroom.register(ringo);
yoko.send("All you need is love.");
yoko.send("I love you John.");
john.send("Hey, no need to broadcast", yoko);
paul.send("Ha, I heard that!");
ringo.send("Paul, what do you think?", paul);

//Väljund
//Yoko to John: All you need is love.
// Yoko to Paul: All you need is love.
// Yoko to Ringo: All you need is love.
// Yoko to John: I love you John.
// Yoko to Paul: I love you John.
// Yoko to Ringo: I love you John.
// John to Yoko: Hey, no need to broadcast
// Paul to Yoko: Ha, I heard that!
// Paul to John: Ha, I heard that!
// Paul to Ringo: Ha, I heard that!
// Ringo to Paul: Paul, what do you think?
```

Lisa 16 Memento kohandatud näide allikast [11]

```
class Program
{
    static void Main(string[] args)
    {
        Originator o = new Originator();
        o.State = "On";
        // Store internal state
        Caretaker c = new Caretaker();
        c.Memento = o.CreateMemento();
        // Continue changing originator
        o.State = "Off";
        // Restore saved state
        o.SetMemento(c.Memento);
        // Wait for user
        Console.ReadKey();
    }
}
class Originator
{
    private string _state;
    // Property
    public string State
    {
        get { return _state; }
        set
        {
            _state = value;
            Console.WriteLine("State = " + _state);
        }
    }
    public Memento CreateMemento()
    { return (new Memento(_state)); }
    public void SetMemento(Memento memento)
    {   Console.WriteLine("Restoring state...");
        State = memento.State;
    }
}
class Memento
{
    private string _state;
    public Memento(string state)
    { this._state = state; }
    public string State
    { get { return _state; }
}
class Caretaker
{
    private Memento _memento;
    public Memento Memento
    {   set { _memento = value; } get { return _memento; } }
}
```

Lisa 17 Vaatleja näide allikast [14]

```
function Click() {
    this.handlers = [];  // observers
}
Click.prototype = {
    subscribe: function (fn) {
        this.handlers.push(fn);
    },
    unsubscribe: function (fn) {
        this.handlers = this.handlers.filter(
            function (item) {
                if (item !== fn) {
                    return item;
                }
            }
        );
    },
    fire: function (o, thisObj) {
        var scope = thisObj || window;
        this.handlers.forEach(function (item) {
            item.call(scope, o);
        });
    }
}
var clickHandler = function (item) {
    console.log("fired: " + item);
};
var click = new Click();
click.subscribe(clickHandler);
click.fire('event #1');
click.unsubscribe(clickHandler);
click.fire('event #2');
click.subscribe(clickHandler);
click.fire('event #3');
//Väljund
//fired: event #1
//fired: event #3
```

Lisa 18 Šabloonmeetodi näide allikast [11]

```
class Program
{
    static void Main(string[] args)
    {
        DataAccessObject daoCategories = new Categories();
        daoCategories.Run();
        DataAccessObject daoProducts = new Products();
        daoProducts.Run();
        Console.ReadKey();
    }
}
abstract class DataAccessObject
{
    protected string connectionString;
    protected DataSet dataSet;
    public virtual void Connect()
    {
        connectionString = "provider=Microsoft.JET.OLEDB.4.0; " +
          "data source=..\\..\\..\\db1.mdb";
    }
    public abstract void Select();
    public abstract void Process();
    public virtual void Disconnect()
    {connectionString = "";}
    // The 'Template Method'
    public void Run()
    {
        Connect();
        Select();
        Process();
        Disconnect();
    }
}
class Categories : DataAccessObject
{
    public override void Select()
    {
        string sql = "select CategoryName from Categories";
        OleDbDataAdapter dataAdapter = new OleDbDataAdapter(sql,
                                           connectionString);
        dataSet = new DataSet();
        dataAdapter.Fill(dataSet, "Categories");
    }
    public override void Process()
    {
        Console.WriteLine("Categories ---- ");
        DataTable dataTable = dataSet.Tables["Categories"];
        foreach (DataRow row in dataTable.Rows)
        {Console.WriteLine(row["CategoryName"]);}
        Console.WriteLine();
    }
}
```

```
class Products : DataAccessObject
{
    public override void Select()
    {
        string sql = "select ProductName from Products";
        OleDbDataAdapter dataAdapter = new OleDbDataAdapter(sql,
                                                connectionString);
        dataSet = new DataSet();
        dataAdapter.Fill(dataSet, "Products");
    }
    public override void Process()
    {
        Console.WriteLine("Products ---- ");
        DataTable dataTable = dataSet.Tables["Products"];
        foreach (DataRow row in dataTable.Rows)
        {Console.WriteLine(row["ProductName"]);}
        Console.WriteLine();
    }
}
```