# Om JavaScript og designmønstre

Johannes Akse

15 juni 2014

Sagt om klassiske design mønstre:

> " And the reason why design patterns look smart is that they push crappy languages like C++ or Java to their limits". — Sergey Mozgovoy

Sagt om JavaScript:

> JavaScript is "[...] clearly a grossly substandard language by modern standards and thus can be legitimately described as a toy language." — David Arno

# Innhold

# Innledning

Jeg vil i dette essayet se nærmere på noen utfordringer forbundet med implementeringen av klassiske designmønstre i JavaScript. Som empirisk grunnlag skal jeg, i min masteroppgave, redesigne et av verktøyene til "The Genomic HyperBrowser", en samling verktøy for statistisk analyse av gendata på nett. Dette essayet vil ikke direkte berøre redesignet av hyperbrowseren, men heller være en konseptuel diskusjon rundt designmønstre og de forskjellige implementeringsteknikkene en har tilgjenglig i programmeringsspråket JavaScript.

Selv om det fortsatt finnes enkelte som mener JavaScript er et lekeprogrammeringsspråk har JavaScript fått en kraftig renessanse de siste 5-6 årene. Store og svært populære rammeverk som Backbone.js og Angular.js på klientsiden og spesielt Node.js på serversiden viser hvordan JavaScript av mange nå anerkjennes som fullgodt programmeringsspråk på linje med de store språkene som Java, C++ og Python.

JavaScript er også standardprogrammeringsspråket for verdensveven(www) (Flanagan 2011, s. 1) og er dermed også et av verdens mest benyttede programmeringsspråk. Med HTML5 har vi fått programmeringsgrensesnitt (api) som "DOM Storage" og "WebSockets" som begge er bygget på JavaScript.

Douglas Crockford skriver i sin bok "JavaScript – The good parts" at under haugen av gode intensjoner og feilgrep ligger et vakkert, elegant og svært uttrykksfullt programmeringsspråk (Crockford 2008, s. 2)Dette essayet er ment å skulle være en oppdagelsesferd i et landskap hvor JavaScript møter klassiske designmønstre.

# JavaScript-programmer trenger struktur

JavaScript er så lett at det er vanskelig. Alt du trenger for å komme i gang er en nettleser. Koden blir tolket der og da og resultatet vises på skjermen. En trenger ikke flere års universitetsutdannelse for å forstå det grunnleggende. Gode opplæringstekster og videoer florerer på internett. Språket er også relativt lett å sette seg inn i. En trenger ikke inngående kunnskap om intrikate forskjeller på heltall(integer), på flyttall(double/floting point) eller på tekststrenger. Både heltall, flyttall, tekststrenger, lister, funksjoner og objekter kan legges i samme peker eller i samme liste av pekere(array). Det finnes ikke private variable, grensesnitt(interface) eller andre avanserte konsepter som gjør læringskurven bratt. Det gir stor frihet og det er fint; terskelen er lav for å begynne å programmere i JavaScript.

Dette har også vært det store ankepunktet for at JavaScript har hatt så vanskelig for å bli anerkjent:

> "Most of the people writing in JavaScript are not programmers. They lack the training and discipline to write good programs. JavaScript has so much expressive power that they are able to do useful things in it, anyway. This has given JavaScript a reputation of being

> strictly for the amateurs, that it is not suitable for professional programming." (Crockford 2001b)

Stor uttrykkskraft og frihet til å kode som en hver finner det for godt kan lett skape anarki og dermed vanskeliggjøre samarbeid både for programmer og for programmerere. Programmer som ikke følger regler eller konvensjoner vil ha problemer med å finne møtepunkter for samhandling med andre programmer. Det vil være vanskelig å bygge ut programmene fordi fremtidige problemer ikke lett lar seg integrere i den nåværende løsningen. JavaScripts mangel på private variable og det globale objektet skaper lett interferens med andre programmer som tilfeldigvis benytter samme navn på en variabel. Det samme kan også skape problemer når en ønsker å benytte tredjeparts programvare.

Problemet ved å benytte eksisterende programmer er ofte ikke like fremtredende fordi promotert programvare, altså programmer som blir anbefalt på nett og i et kodemiljø, ofte vil følge anerkjente programmeringsprinsipper og standarder for god kode. De vil sørge for god innkapsling av variable og logikk og dermed skape modulære komponenter som ikke søler til det globale objektet slik ustrukturert JavaScript-kode ofte gjør.

Ustrukturert kode er også vanskelig å teste. Når programmene ikke er bygget opp av små, uavhengige komponenter som kan testes hver for seg, vil det bli vanskelig å avgjøre hvilke komponenter som skaper uønsket oppførsel og resultater.

Java, C++ og andre strengt typede kodespråk presser, på den andre siden, programmereren til å følge visse standarder for god kode. Det tilbys ofte gjennomtenkte mønstre for oppdeling og samarbeid mellom objekter og det tilbys mulighet for å kategorisere funksjonalitet. Etablerte mønstre for arv og polymorfi manifesteres i klassehierarkier. I tillegg finnes det en rekke klassiske designmønstre.

## Klassiske designmønstre

Klassiske designmønstre har en dobbel betydning. På den ene siden referer "klassiske" designmønstre til etablerte og vel utprøvde maler for god programdesign(Gamma mfl. 1996, s. 12). På den andre siden refererer "klassiske" designmønstre til objektorientert programmering hvor klasser fungerer som en mal for objektproduksjon (Stefanov 2010, s. 115). Den etter hvert "klassiske" og svært innflytelsesrike boka "Design patterns: Elements of Reusable Object-Oriented Software" foreslår som en grunnstamme for videre utvidelse, 23 navngitte objektorienterte designmønstre. Hovedformålet med boka var å fange og nedtegne design ekspertise i en slik form at den ble lett tilgjengelig for effektiv bruk (Gamma mfl. 1996, s. 12). Gode designmønstre med dertil gjennomtenkte og beskrivende navn kan både lette kommunikasjonen mellom erfarne programmerere, være til god hjelp for raskere og bedre forståelse av objektorienterte programmeringsprinsipper og fungere som kognitive hjelpemidler for ferske programmerere og programmeringsstudenter. Boka har i følge Wikipedia solgt over 500.000 eksemplarer og er utgitt på mer 13 forskjellige språk (*Design Patterns*

2014).

### JavaScript som objektorientert språk

JavaScript er ikke i utgangspunktet et "klassisk" programmeringsspråk i forståelsen hierarkier av klasser som mal for objektproduksjon, men tilbyr likevel muligheten til å emulere klassiske prinsipper som arv, polymorfisme og innkapsling. JavaScript tilbyr konstruktørfunksjoner og operatoren "new" men i motsetning til klassiske språk vil malen alltid være et faktisk, prototypisk objekt og ikke en klasse. JavaScript er i så måte objektorientert. Alt er objekter og alle objekter, med unntak av et grunnstammeobjekt, er utstyrt med en usynlig peker til objektet som fungerte som mal for reproduksjonen eller sagt på en annen måte: som objektet ble klonet av. Det er dette systemet av usynlige prototypelenker som utgjør den såkalte prototypekjeden. Dette skaper muligheten for gjenbruk av funksjonalitet. Når man kaller en metode i et objekt som ikke har denne metoden, vil prototypekjeden bli fulgt til neste objekt i kjeden. Finnes ikke metoden her heller følges prototypekjeden videre til den enten finner metoden eller den når bunnen, grunnstammeobjektet; Objekt.prototype er et objekt som kommer ferdig laget med JavaScript. Ved å legge funksjonalitet et sted i prototypekjeden, vil den kunne gjenbrukes av objektene som ligger over. Forskjellige designmønstre kan utnytte og manipulere prototypekjeden til i stor grad å likne klassehierarkier.

Det er med andre ord mulig å skrive JavaScriptprogrammer med klasseliknende struktur. Likevel frarådes bruken av "new" fordi det bare gir inntrykk av å gjenskape klassisk arv (Crockford 2006) Det finnes fire forskjellige måter å kalle en funksjon på. Bonuspekeren "this", som automatisk blir tilgjenglig ved opprettelsen av en funksjon eller metode, vil få forskjellig verdi ettersom hvilket funksjonskallmønster som benyttets. "This" i en metode peker for eksempel tilbake til objektet som kalte metoden og dermed bindes "this" til variable i den respektive kallmetoden og ikke til variable i objektet hvor metoden ble definert. Om en ikke er bevisst disse raritetene i JavaScripts implementasjon av "this" og antar en standard klassisk tilnærming vil det lett oppstå uforutsette situasjoner som kan være vanskelige å feilsøke (Crockford 2008, s. 27-30).

## Designprinsipper – Gjenbruk av kode

### Arv

JavaScript er i så måte et vanskelig programmeringsspråk å beherske. Det er også et svært uttrykksfullt programmeringsspråk i den forstand at det tillater flere forskjellige tilnærminger og kodemønstre for å uttykke det samme(Harmes og Diaz 2008, s. xxii). Crockford identifiserer for eksempel fem forskjellige måter å oppnå arv eller hierarkier av gjenbrukbar kode (Crockford 2001a). Å kunne gjenbruke kode, skriver Crockford videre, er en viktig grunn for at klassehierarkier har en så sentral plass i objektorientert programmering.

Samtidig, ved slik gjenbruk av funksjonalitet, også kalt "White-box"-gjenbruk, vil de indre implementasjonene av den aktuelle funksjonaliteten ofte være synlig for underklasser. Det skaper en nær tilknytning til foreldreklassene noe som lett kan medføre tungvindt vedlikeholdsarbeide. Hvis en foreldreklasse må forandre implementeringen vil det få direkte konsekvenser for de klassene som arver funksjonaliteten, noe som kan medføre at de også må skrives om (Gamma mfl. 1996, s. 31).

## Type

En annen viktig aktør i klassisk objektorientering er konseptet funksjonstype. Klassiske programmeringsspråk tilbyr grensesnitt hvor funksjonalitet beskrives men den implementeres ikke. Dermed kan forskjellige implementeringer av det samme grensesnittet tilby liknende men samtidig forskjellig funksjonalitet. Pekere i et "brukerobjekt" kan slik referere kun til ønsket funksjonstype – en "har"-relasjon og ikke til en faktisk implementering – en "er"-relasjon (Freeman mfl. 2004, s. 23). Dette muliggjør at objekter kan bytte funksjonalitet dynamisk under kjøring av programmet. Dette er blitt kalt komposisjon og danner grunnlaget for et viktig objektorientert designprinsipp: Favoriser objektkomposisjon fremfor klassisk arv (Gamma mfl. 1996, s. 32).

Når en komponerer på denne måten, det kalles ofte "Black-box"-gjenbruk, har brukerobjekter ingen innsikt i hvordan funksjonaliteten er implementert og programmet får således en løsere kobling mellom objekter. Slik oppnås avgrenset funksjonalitet og en mer modulær programstruktur hvor hver enhet opptrer mest mulig uavhengig. Da blir enhetene mindre utsatte for implementasjonsforandringer enn de blir ved funksjonsarv.

Det vil også være mye lettere å teste enheter hver for seg. Et objekt som ikke har en nær tilknytning til et annet objekt vil i en testsituasjon kunne utføre sin funksjon uten behov for innblanding av annen kode og kan derfor testes som en uavhengig enhet. Feil som er forårsaket av en implementering i avhengige objekter vil dermed ikke forflytte seg og slik skjule hvor feilen oppsto i utgangspunktet.

Grensesnitt og funksjonalitetsspesifisering ved hjelp av type er en meget viktig forutsetning for gode objektorienterte programmer. Det første designprinsippet som nevnes i den originale boka til Gamme et al. om klassiske objektorienterte designmønstre sier at en skal programmere mot grensesnitt og ikke implementasjoner. JavaScript tilbyr ikke grensesnitt men er likevel så fleksibelt at det lett kan utvides med tilleggsfunksjonalitet som emulerer klassisk bruk av grensesnitt mellom objekter. Fleksibiliteten skyldes i stor grad at det ikke er krav om å spesifisere type i JavaScript. Det er altså et løst typet språk (loosly typed) i motsetning til stengt typede språk (strongly typed). Dette gjør også at mye av grunnlaget for å ønske å benytte grensesnitt i utgangspunktet faller bort:

> "The thing about design patterns in relation to JavaScript is that, although language-independent, the design patterns were mostly stu-

> died from the perspective of strongly typed languages, such as C++ and Java. Sometimes it doesn't necessarily make sense to apply them verbatim in a loosely typed dynamic language such as JavaScript. Sometimes these patterns are workarounds that deal with the strongly typed nature of the languages and the class-based inheritance. In JavaScript there might be simpler alternatives" (Stefanov 2010, s. 2).

Stefanovs refleksjoner gjelder i stor grad det første designprinsippet om å kode mot grensesnitt og ikke mot implementasjoner. Selv om en skulle emulere grensesnitt vil det ikke være en kompilator som kan garanterer at grensesnittkonvensjonene opprettholdes. Fordi det heller ikke gjøres forskjell på pekere og at de uansett bare kan typebetegnes med "var" kan det se ut til at det gir lite eller ingen nytte å emulere grensesnitt i JavaScript.

## JavaScript som funksjonelt språk

Om en ser bort fra syntaksen har JavaScript mer til felles med funksjonelle programmeringsspråk som Lisp og Scheme (Crockford 2001b) enn objektorienterte språk som Java og C. JavaScripts funksjoner er førsteklasses objekter, det vil si at de kan sendes med som argument til andre funksjoner. Når funksjoner kan sendes med som argument til "brukerobjektet" kan også den aktuelle funksjonaliteten som representeres ved funksjonen kalles direkte når den trengs. Funksjonaliteten kan like lett byttes ut under kjøring. Et slikt språkspesifikt paradigme opprettholder prinsippet om uavhengige enheter og å skulle kode mot grensesnitt og ikke implementeringer. Brukerobjektet trenger ikke vite noe om hvordan argumentfunksjonen er implementert, den trenger bare å kalle den når den trengs.

JavaScriptfunksjoner er også lambdaer (Crockford 2001d). En lambda i funksjonelle språk er en operator som betegner anonyme funksjoner (*Lambda* 2014). En anonym funksjon kan brukes til å skape det som kalles " closure". En "closure" er et funksjonelt designmønster som avgrenser en del av et program. Variable som er definert inne i den anonyme funksjonen ikke kan nås direkte fra andre steder i programmet. I JavaScript er det kun funksjoner som avgrenser programmet på en slik måte. Variable definert inne i vanlige objekter kan nåes også fra utsiden, de er altså ikke private. Ved å deklarere variable i en anonym funksjon og deretter returnere et objekt med metoder, for eksempel "setter" og "getter" metoder, som tar i bruk variablene definert i den anonyme funksjonen, vil objektet som returneres kunne sies å ha fått private variable. Nå er det kun mulig å sette eller hente variablenes verdi via setter- og getter-metodene i objektet. I tillegg til å spare det globale objektet fra å ha direkte tilgang til alle variable har en nå tilgjengelig et mønster for å separere et programs funksjonalitet i ulike ansvarsområder. Det globale objektet kan deles opp i moduler med egne navnerom (namespases) for privat lagring av tilstandsvariable og funksjonalitet.

# Rammeverk – implementering av designmønstre

## Model View Controller

Oppdeling av ansvar er et viktig designprinsipp. Det fremmer fleksibilitet og gjenbruk (Gamma mfl. 1996, s. 14). Allerede i 1979 formulerte Trygve Reenskaug Model-View-Controller (MVC) (*Trygve Reenskaug* 2014). MVC er et høynivå designmønster som skisserer tre avgrensete komponenter med sine respektive ansvarsområder. View er grensesnittet mot brukeren. Model er en komponent som modellerer domenespesifikk data. Controller tar imot brukerinteraksjon fra et view og meddeler modellen om de ønskete forandringene. MVC adskiller brukergrensesnitt og modeller ved å etablere et abonnent/varsler forhold mellom dem. På denne måten kan en la en og samme modell bli presentert på forskjellig vis. En modell kan for eksempel fremvises som et venn-diagram, en graf og et søylediagram (Gamma mfl. 1996, s. 14-15). Siden ansvarsområdene er avgrenset kan en lett bytte ut et view med et annet.

Det finnes en rekke konseptuelt forskjellige implementeringer av det orginale MVC mønstret til Reenskaug. Både Model-View-Adapter(MVA) og Model-View-VeiwModel(MVVM) er varianter av MVC. Slik Gamma et al. beskriver MVC i sin designmønsterbok ser en at det er et høynivå designmønster som kombinerer flere andre designmønstre. I det en bruker av systemet trykker en knapp i brukergrensesnittet kontaktes kontrolleren som er assosiert med dette viewet. Kontrolleren ber modellen oppdatere seg i forhold til brukerens ønsker. Modellen meddeler så viewet direktet om at dets tilstand er forandret. Dermed kan viewst kalle modellen og få de konkrete forandringene og oppdatere seg selv i henhold. Dette mønsteret kaller Gamma et al. for "the Observer pattern" (Gamma mfl. 1996, s. 15).

Views kan også bli lagt inni hverandre i en treliknende struktur hvor viewet både kan opptre som rotnode i sin egen trestruktur men kan også opptre som løvnode uten flere underliggende noder. For å finne de forskjellige nodene kan en implementere "the Composite pattern" (Freeman mfl. 2004, s. 530-532). Da kan en lettere iterere over hele samlingen uten å ta hensyn til om det nåværende objektet er en løvnode eller om det har flere noder under seg. Dette kan for eksempel være en måte å finne elementer i en nettlesers DOM struktur på.

MVC mønsteret muliggjør også at et view kan forandre oppførsel ved brukerinteraksjon uten å forandre utseende (Gamma mfl. 1996, s. 16). Fordi et view kan benytte en instans av en kontrollunderklasse – eller like gjerne en klasse som implementerer et kontrollgrensesnitt – kan denne lett byttes ut med en instans av en annen kontrollunderklasse hvis disse har de samme metodene tilgjengelig. Å kunne bytte oppførsel dynamisk på denne måten kaller Gamma et al. for "The stratey pattern" (Freeman mfl. 2004, s. 532).

MVC var i utgangspunktet tiltenkt konvensjonell GUI implementeringen for desktopapplikasjoner. Ettersom verdensveven fikk fotfeste utover 1990-tallet ble MVC raskt en selvfølgelighet også i webapplikasjoner. De første 10-15 årene, før netthastighet ble kapabel til å overføre større programmer innen rimelig tid og før nettlesernes JavaScript-tolkere var blitt så raske som de er idag, ble

MVC-mønsteret implementert på serversiden i det som har blitt kalt "Modell 2" (Freeman mfl. 2004, s. 557). Nå er det mer og mer vanlig å implementere MVC-mønsteret i det som ofte kalles tykke klienter (thick clients) eller enkeltsideapplikasjoner (SinglePageApplications). Her blir det meste av funksjonalitet implementert i kode som blir sent til nettleseren ved oppstart nettsiden. Kommunikasjonen med serveren foregår nå oftest mot et REST-api som kun utfører enkle operasjoner som å hente, lagre, oppdatere og slette data. XML-formatet er i stor grad byttet med det noe lettere JSON-formatet, som er et forenklet JavaScript objekt med nøkkel/verdi par (key-value pair) uten metoder. Nettleserens programmeringsspråk er JavaScript og det voksende antallet SPA-applikasjoner og rammeverk som implementeres som SPA-applikasjoner er med på å befeste språkets viktighet og anerkjennelse.

## Angular.js

Angular.js er en SPA-applikasjon. Kodebasen lastes inn fra serveren i en operasjon i det en navigerer til en adresse knyttet til det aktuelle domene. Deretter er det Angular.js som dirigerer underadresser (URL'er) av domenet til riktig view. Deretter blir alle andre oppdateringer av innhold gjort ved asynkrone kall til serveren. Dette gjør at en side kan fremstå som rask å laste mellom undersider.

Angular implementerer også en variant av designmønsteret MVC. Igor Minar, en av sjefsutviklerene av Angular.js, ønsket ikke å gå dypt inn i diskusjonen om hvorvidt det er mer hensiktsmessig å implementere et MVC, MVP, MVA eller MVVM. Han deklarerte derfor at Angular.js implementerer Model-View-Whatever (MVW), hvor whatever står for "hva som måtte fungere for deg" (*Model View Whatever* 2012).

I Angularvokabularet finner man både view og controllere men modellen kaller de for scope. scope er et ferdig produsert objekt hvor en kan legge til forskjellige egenskaper (properties) fra databaseoppføringer. Disse egenskapene til scopeobjektet er direkte tilgjengelig for viewet og har en såkalt toveis databinding. Det vil si at når det skjer en forandring, enten i viewet eller i modellen, vil begge bli oppdatert automatisk. scopeobjektet blir som regel injisert i en controller og det kan slik sett se ut som om det er controlleren som medierer all kontakt mellom viewet og modellen.

Viewet er vanlig HTML men med det Angular.js kaller "Directives". Dette er markører som legges til et DOM element og som forteller Angular.js kompilatoren at det skal legges til ny funksjonalitet til elementet. Slik kan man dele opp og avgrense deler av HTML dokumentet og tildele ansvarsområder for kontrollere eller oppførsel av annen type. Slik kan en godt si at Angular.js lærer opp nettleseren til å forstå ny syntaks og utvikler DOM'en til å bli mer enn et statisk dokument for strukturering av innhold. Dermed tilbyr det mer enn et rammeverk:

> "The impedance mismatch between dynamic applications and static documents is often solved with:

- a library - a collection of functions which are useful when writing web apps. Your code is in charge and it calls into the library when it sees fit. E.g., jQuery.
- frameworks - a particular implementation of a web application, where your code fills in the details. The framework is in charge and it calls into your code when it needs something app specific. E.g., knockout, ember, etc.

Angular takes another approach. It attempts to minimize the impedance mismatch between document centric HTML and what an application needs by creating new HTML constructs. Angular teaches the browser new syntax through a construct we call directives." (*What is Angular* 2014)

Angular.js tilbyr webapplikasjonsutviklere et solid utgangspunkt for å følge viktige designprinsipper som "innkapsling" av egenskaper som varierer og "favoriser komposisjon fremfor arv". Dette siste poenget blir ivaretatt ved avhengighetsinjeksjon (dependency injection), hvor en injiserer en peker til et objekt som argument i stedet for å opprette nye objekter fra objektet som trenger funksjonaliteten. Objektene blir mer uavhengige og dermed lettere å enhetsteste.

Samtidig som Angular.js gjør webapplikasjonsutvikling lettere ved å sørge for god struktur og objekter som automatisk samarbeider på hensiktsmessige måter tvinges også utvikleren til å følge den utviklingsfilosofien som Angular.js legger opp til:

"Angular simplifies application development by presenting a higher level of abstraction to the developer. Like any abstraction, it comes at a cost of flexibility. In other words not every app is a good fit for Angular. Angular was built with the CRUD application in mind. Luckily CRUD applications represent the majority of web applications" (*What is Angular* 2014).

Når det meste av struktur som trengs allerede er på plass er det lett å falle inn i klassisk objektorienterte tankebaner og glemme JavaScripts funksjonelle natur, spesielt for utviklere som er bedre trenet i klassiske programmeringsspråk som Java og C++.

## Backbone.js

Backbone er, i motsetning til Angular.js, et bibliotek og ikke et rammeverk (*Backbone.js* 2013). Her er det opp til utvikleren å skape en kodestruktur som er hensiktsmessig for applikasjonens bedrif tsmodell. Backbone tilbyr å utvide allerede eksisterende objekter med tilleggsfunksjonalitet. Dette muliggjør å fortsette et JavaScriptkodeparadigme fremfor å skulle blande inn klasser og andre klassiske konstrukter.

Selv om Backbone kun er et bibliotek som tilbyr tilleggsfunksjonalitet opererer de med konsise moduler hvor funksjonaliteten er organisert etter et MVW.

Der finnes både View og Model i tillegg til andre avgrensede funksjonsområder som "routing", events og collections. Deres Backbone.Model.extend funksjon sørger for at en korrekt prototypekjede blir satt opp slik at objekter som er laget etter dette mønsteret også vil kunne bli bygget videre (*Backbone* 2013).

Slik leder også Backbone utviklere til å tenke designmønstre uten å binde dem opp til en gitt struktur. Det er opp til utvikleren å ta de riktige designavgjørelsene:

> "Backbone supplys helpful methods to manipulate and query your data, not on HTML widgets that are reinventing the javascript object model" (*Backbone* 2013)

# Avslutning

Det er etter hvert blitt utviklet et uttall JavaScript- biblioteker og -rammeverk som tilbyr utviklere verktøy for å strukturere programmer etter gode designprinsipper. Noen med mer og noen med mindre rigide oppsett av designmønstre. Hovedpoenget med dette essayet har vært å gi et innblikk i noen av de mulighetene som finnes for å gi JavaScript som programmeringsspråk den strukturen og verktøyene det i utgangspunktet har manglet. Dette kan være med å gi JavaScripteutviklere et løft i retning av å produsere programmer som følger mønstre for god kodeskikk og øke mengden av applikasjoner som er verdig et i utgangspunktet uttrykksfullt men vanskelig strukturelt språk. Kritikken språket har vært utsatt for er dermed vanskeligere å opprettholde og JavaScript kan forsvare sin posisjon som det mest brukte programmeringsspråket.

# Litteraturliste


Arno, David (2013). *Local Guide to BibLaTeX*. http://www.davidarno.org/2010/05/18/why-javascript-is-a-toy-language/ (Sett: 13.06.2014).

*Backbone* (2013). Webpage. http://backbonejs.org/ (Sett: 12.06.2014) (se s. 12).

*Backbone.js* (2013). Webpage. http://documentcloud.github.io/backbone/ (Sett: 07.06.2014) (se s. 11).

Crockford, Douglas (2001a). *Classical Inheritance in JavaScript*. Webpage. http://javascript.crockford.com/inheritance.html (Sett: 08.06.2014) (se s. 6).

— (2001b). *JavaScript:The World's Most Misunderstood Programming Language*. Webpage. http://javascript.crockford.com/javascript.html (Sett: 09.06.2014) (se s. 5, 8).

— (2001c). *The Little JavaScripter*. Webpage. http://javascript.crockford.com/little.html (Sett: 10.06.2014).

— (2001d). *The Little JavaScripter*. Webpage. http://javascript.crockford.com/little.html (Sett: 08.06.2014) (se s. 8).

— (2006). *JavaScript, We Hardly new Ya*. Webpage. http://yuiblog.com/blog/2006/11/13/javascript-we-hardly-new-ya/ (Sett: 10.06.2014) (se s. 6).

— (2008). *JavaScript:The good parts*. First Edition. Sebastopol, CA 95472: O'Reilly Media, Inc. (se s. 4, 6).

*Design Patterns* (2014). Webpage. http://en.wikipedia.org/wiki/Design_Patterns (Sett: 11.05.2014) (se s. 5).

Flanagan, David (2011). *JavaScript:The Definitive Guide*. Sixth Edition. Sebastopol, CA 95472: O'Reilly Media, Inc. (se s. 4).

Freeman, Eric mfl. (2004). *Head First Design Patterns*. First Edition. Sebastopol, CA 95472: O'Reilly Media, Inc. (se s. 7, 9, 10).

Gamma, Erich mfl. (1996). *Design Patterns : Elements of Reusable Object-Oriented Software*. Eight Printing. Addison Wesley Publishing Company, Inc. (se s. 5, 7, 9).

Harmes, Ross og Dustin Diaz (2008). *Pro JavaScript:Design Patterns*. First Edition. Berkeley, CA 94705: Apress (se s. 6).

*Lambda* (2014). Webpage. http://en.wikipedia.org/wiki/Lambda (Sett: 10.06.2014) (se s. 8).

*Model View Whatever* (2012). Webpage. https://plus.google.com/+AngularJS/posts/aZNVhj355G2 (Sett: 15.06.2014) (se s. 10).

*Model–view–controller* (2014). Webpage. http://en.wikipedia.org/wiki/Model-view-controller (Sett: 13.06.2014).

Mozgovoy, Sergey (2013). *The biblatex package*. http://mirrors.ctan.org/macros/latex/contrib/biblatex/doc/biblatex.pdf (Sett: 13.06.2014).

*ObjectPlayground* (2013). Webpage. http://www.objectplayground.com/ (Sett: 08.06.2014).

Stefanov, Stoyan (2010). *JavaScript patterns*. First Edition. Sebastopol, CA 95472: O'Reilly Media, Inc. (se s. 5, 8).

*Trygve Reenskaug* (2014). Webpage. `http://en.wikipedia.org/wiki/Design_Patterns` (Sett: 05.06.2014) (se s. 9).

*What is Angular* (2014). Webpage. `https://docs.angularjs.org/guide/introduction` (Sett: 15.06.2014) (se s. 11).