Kandidatarbete
COM
Thesis no: MCS-20xx-yy
Juni 2011

Jämförelse av funktionsbibliotek för JavaScript

- En prestandajämförelse av funktionsbibliotek för JavaScript vid de vanligaste DOM-interaktionerna i de vanligaste webbläsarna

Eric Ericsson & Petter Andersson

School of Computing
Blekinge Institute of Technology
SE – 371 79 Karlskrona
Sweden

This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 20 weeks of half time studies.

Contact Information:
Author(s):
Petter Andersson
E-mail: petter89@hotmail.com

Eric Ericsson
E-mail: er.ericsson@gmail.com

University advisor:
Stefan Johansson
COM/ Blekinge Institute of Technology

School of Computing
Blekinge Institute of Technology
SE – 371 79 Karlskrona
Sweden

Internet : www.bth.se/com
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

SAMMANFATTNING

När de traditionella skrivbordsapplikationerna görs om till webbapplikationer, ställer det krav på JavaScript som används mer och mer för att få fram ett responsivt gränssnitt på webben. För att underlätta utvecklingen av JavaScriptapplikationer har ett antal funktionsbibliotek skapats.

I vår studie undersöker vi därför vilket av de två populäraste JavaScriptbiblioteken idag, jQuery och Prototype, som presterar bäst i dagens mest använda webbläsare. Dessa tester har utförts i ett testramverk som vi själva utvecklat för att vara webbläsaroberoende och inte kräva något av de bibliotek vi testar. Testerna är uppdelade i fyra testfall som körs 20 gånger för att ge ett mer tillförlitligt resultat. Vi har testat hur varje bibliotek hanterar traversering och manipulation av DOM-trädet, sätter och hämtar stilar och attribut på element i DOM-trädet och hanterar event på element i DOM-trädet.

Testerna visade att biblioteket Prototype presterade bättre på alla utom ett testfall i majoriteten av våra utvalda webbläsare; det enda testfallet där jQuery presterade bättre än Prototype var där DOM-trädet skulle manipuleras. Trots att Prototype inte alls är lika omtalat som jQuery, verkar det vara ett bättre bibliotek att använda till webbapplikationer som ska ha ett interaktivt gränssnitt då det i flertalet av våra tester presterar bättre.

INNEHÅLLSFÖRTECKNING

1 INLEDNING 3

1.1 FORSKNINGSFRÅGA 3
1.2 HYPOTES 3
1.3 FRÅGESTÄLLNINGAR 3
1.4 MÅL OCH SYFTE 3
1.5 MÅLGRUPP 3
1.6 METOD 4
1.7 AVGRÄNSNINGAR 4

2 BAKGRUND 5

2.1 HUR FUNGERAR DET 6
2.2 DOCUMENT OBJECT MODEL – DOM 6
2.2.1 DOM-Scriptande 8
2.2.2 Prestanda och DOM 9
2.3 FUNKTIONSBIBLIOTEK 9
2.3.1 jQuery 10
2.3.2 Prototype 10

3 UTFORMNING AV EXPERIMENT 11

3.1 TESTMILJÖ 12
3.1.1 Hårdvaruspecifikation 12
3.1.2 Mjukvara och funktionsbiblioteksversioner 12
3.2 TESTFALL LADDNINGSTIDER 13
3.3 TESTFALL TRAVERSERING OCH MANIPULATION 13
3.4 TESTFALL ATTRIBUT OCH STILAR 14
3.5 TESTFALL EVENTRESPONS 14

4 RESULTAT 15

4.1 RESULTAT TESTFALL LADDNINGSTIDER 15
4.2 RESULTAT TESTFALL TRAVERSERING OCH MANIPULATION 16
4.3 RESULTAT TESTFALL ATTRIBUT OCH STILAR 20
4.4 RESULTAT TESTFALL EVENTRESPONS 22

5 DISKUSSION 24

5.1 VAD KAN PÅVERKAT MÄTNINGARNA 26

6 SLUTSATS 28

6.1 FÖRSLAG PÅ FORTSATT FORSKNING 28

7 ORDLISTA 30

8 LITTERATURFÖRTECKNING 31

8.1 BÖCKER 31
8.2 WEBBSIDOR 31

9 BILAGOR 32

9.1 RESULTATTABELLER 32
9.1.1 Firefox 4.0.1 32
9.1.2 Chrome 11.0.696.68 34
9.1.3 Safari 5.0.5 36
9.1.4 Opera 11.10 38
9.1.5 Internet Explorer 9.0.8112.1421 40
9.2 BERÄKNINGAR 43
9.3 TESTFALLSKOD 44
9.3.1 jQuery Index 44
9.3.2 jQuery Traversal 45

1 INLEDNING

JavaScript har länge varit en del av var webbutvecklares verktygslåda och dess popularitet har gått upp och ner med åren. I och med satsningarna på modernare JavaScriptmotorer i webbläsarna och att dagens applikationer och dynamiska gränssnitt flyttar ut på webben[1] ställs det högre krav på webbläsarnas sätt att hantera JavaScript. Samtidigt som JavaScript i sig utvecklas genom nya funktionsbibliotek som underlättar arbetet för utvecklarna.

Då vi ska ta fram ett dynamiskt webbgränssnitt ställs vi för en hel del frågeställningar angående JavaScript idag. Vi väljer därför att undersöka hur de två största funktionsbiblioteken idag hanterar de mest prestandakrävande operationerna i ett webbläsaroberoende webbgränssnitt. Det vi kommer testa är så kallade DOM-interaktioner, DOM står för Document Object Model och används för att på ett objektorienterat sätt beskriva strukturen i ett HTML- eller XML-dokument.

1.1 Forskningsfråga

Vilket av funktionsbiblioteken jQuery och Prototype presterar bäst i olika webbläsare vid olika typer av DOM-interaktioner?

1.2 Hypotes

Vi tror att webbläsaroberoende webbgränssnitt med DOM-interaktioner skrivna i jQuery presterar bättre än samma gränssnitt skrivet i Prototype.

1.3 Frågeställningar

1. Hur påverkar sättet vi laddar in funktionsbiblioteken tiden det tar innan vi kan börja använda gränssnittet?

2. Vilket av biblioteken hanterar DOM- manipulation och traversering snabbast?

3. Vilket av biblioteken hanterar förändring av DOM-attribut och stilar snabbast?

4. Vilket av biblioteken skapar, triggar och tar bort DOM-events snabbast?

1.4 Mål och syfte

Syftet med arbetet är att visa vilka prestandakostnader som finns i ett dynamiskt webbgränssnitt skrivet i JavaScript och som använder sig av DOM-interaktioner. Arbetet visar också hur prestandan skiljer sig mellan olika funktionsbibliotek och webbläsare.

1.5 Målgrupp

Målgruppen för jämförelsen är svenska webbutvecklare. Både de som skriver egna JavaScript hemma och de som utvecklar webbtjänster och framförallt gränssnitt för webben i kommersiellt syfte, kan ha intresse av resultaten.

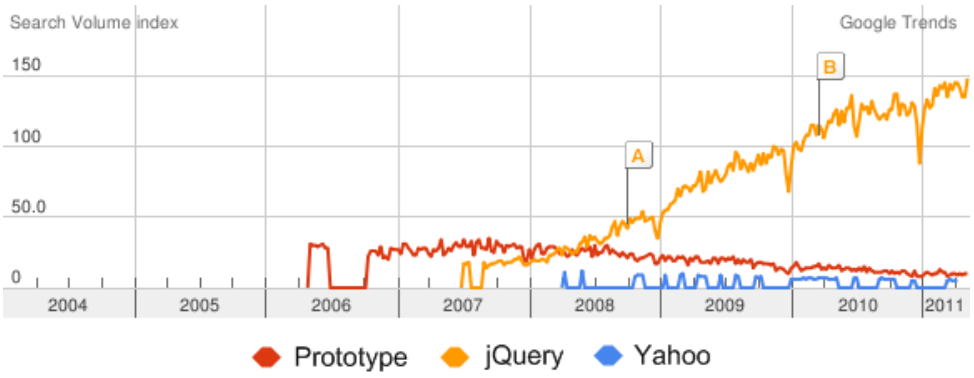
[1] DOM Scripting, kap. 12

1.6 Metod

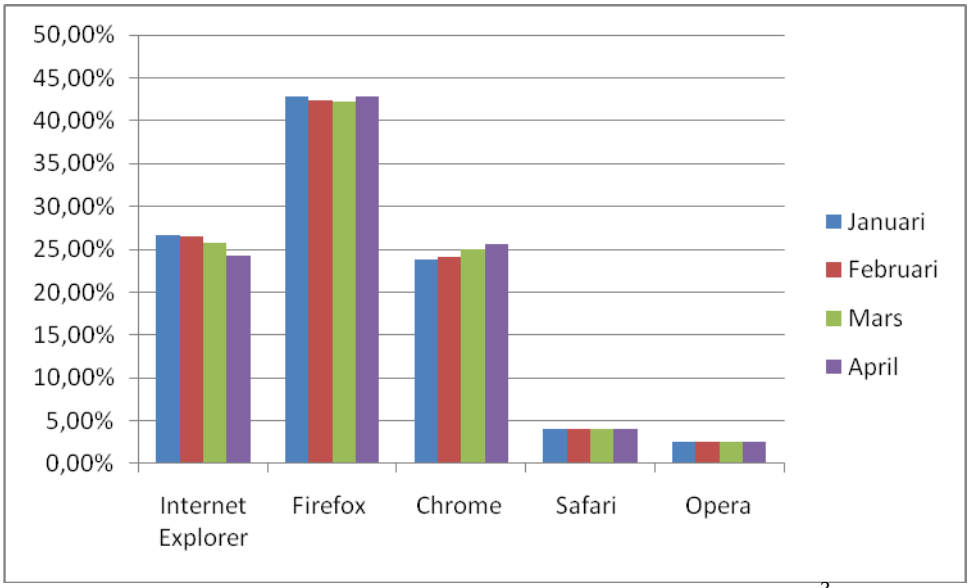
Undersökningen granskar funktionsbiblioteken jQuery och Prototype i flertalet tester som berör prestanda och responsivitet i webbgränssnitt. Motsvarande funktionalitet i de båda funktionsbiblioteken ställs mot varandra och testerna genomförs i ett flertal olika webbläsare för objektivitet och en mer heltäckande bedömning.

1.7 Avgränsningar

Arbetet kommer fokusera på de funktioner i de två funktionsbiblioteken som har störst påverkan på DOM. Vi har valt att undersöka funktioner i jQuery och Prototype då de är de två största funktionsbiblioteken för JavaScript i skrivande stund.(Figur 1) Mätningarna sker i de senaste versionerna av de enligt w3c fem mest använda webbläsarna för tillfället, Internet Explorer, Firefox, Safari, Chrome och Opera. (Figur 2)



Figur 1 - Vanligaste biblioteken baserat på Google Trends[2]



Figur 2 - Procentuell fördelning av vanligaste webbläsare i dag[3]

[2] http://www.google.com/trends?q=Yahoo+javascript%2C+Prototype+javascript%2C+Jquery+javascript&ctab=0&geo=all&date=all&sort=0

[3] http://www.w3schools.com/browsers/browsers_stats.asp

2 BAKGRUND

JavaScript är ett klientscriptspråk för webben. Som många andra språk liknar det C, C++ och Java syntaxmässigt. JavaScript utvecklades av Netscape i samarbete med SUN-microsystems. Det introducerades första gången i Netscape Navigator 2 1995. Innan JavaScript inkluderades i webbläsarna var det endast enkla applikationer som kunde utvecklas, de klarade bara av att tolka och visa hypertextdokument. JavaScript har mindre att göra med Java än vad namnet antyder, SUN-microsystems var visserligen med vid utvecklingen och syntaxen har vissa likheter. Närmare Java än så kommer inte JavaScript. Från första början var det tänkt att det skulle heta LiveScript.[4]

Det ursprungliga målet med JavaScript var att förbättra användbarheten på webbsidor genom att underlätta saker som formulärvalidering för användaren. [5] När JavaScript släpptes var uppkopplingarna och prestandan på datorerna inte alls lika bra som idag och det tog lång tid för en sida att laddas. Hade man då ett formulär på sidan kunde användaren sitta och vänta på att formuläret som skickades in kom tillbaka med meddelandet att formuläret var fel ifyllt. I värsta fall kom formuläret tillbaka tomt och användaren blev tvungen att börja om på nytt. Då var JavaScript till stor hjälp när formulären skulle valideras eftersom valideringen skedde på klientsidan först.

När JavaScript standardiserades i ECMA-262 var det baserat på flera liknande tekniker, framförallt Netscapes JavaScript och Microsofts JScript.[6] När språket var standardiserat och webbläsartillverkarna varit delaktiga i utvecklingen av en standardiserad DOM passade tillslut utvecklingsplattformen alla webbläsare. Detta ledde till att webbapplikationerna blev mer dynamiska och gränssnitten kunde förändras utan omladdning av hela sidan. JavaScript har haft ett dåligt rykte under en tid på grund av att scripten blev mer och mer resurskrävande men de exekverades på samma gamla JavaScriptmotorer som inte hängde med i utvecklingen.[7]

Trots att JavaScriptmotorerna i webbläsarna har tagit ett stort kliv i utvecklingen de senaste åren finns det fortfarande problem med prestandan inom JavaScript. Det finns funktioner i språket som inte prestandan i JavaScriptmotorn kan påverka. All DOM-integration i JavaScript och även hämtningen av scriptfilerna från servern till klienten är utanför JavaScriptmotorernas scope.

[4] DOM Scripting, kap. 1
[5] High Performance JavaScript, Preface
[6] DOM Scripting kap. 1,
http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf
[7] High Performance JavaScript, Preface

2.1 Hur fungerar det

Till skillnad från till exempel Java och C++ är inte JavaScript ett språk som kompileras utan det tolkas istället av webbläsaren direkt när det körs. Detta leder till att finns det fel i koden visas dessa när sidan visas i webbläsaren.[8] När JavaScripten körs i webbläsaren hindrar det resten av sidan att genereras. Finns det då ett script som körs en längre tid tar det en stund innan något syns i webbläsaren och sidan blir redo att användas av användaren.[9] Det webbläsaren använder för att tolka JavaScripten kallas JavaScriptmotor, som Tabell 1 visar har alla webbläsare sin egen JavaScriptmotor.

Webbläsare	**JavaScriptmotor**
Mozilla Firefox 4	Jägermonkey [10]
Google Chrome	V8 [11]
Opera	Carakan [12]
Safari	Nitro [13]
Internet Explorer 9	Chakra [14]

Tabell 1

2.2 Document object model – DOM

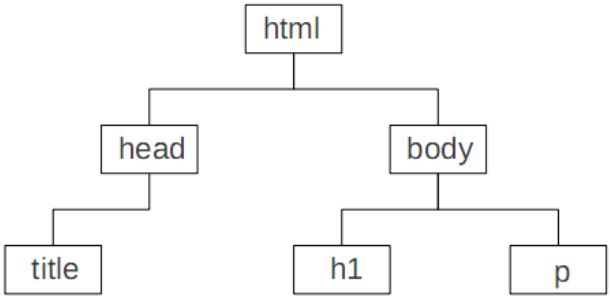
DOM står för Document Object Model och är ett objektorienterat sätt att beskriva strukturen i ett HTML- eller XML-formaterat dokument. När ett HTML- eller XML-dokument läses in av en webbläsare skapar den en modell som representerar dokumentet som sedan kan läsas och modifieras av diverse olika scriptspråk. Men för att kunna använda sig av modellen behöver man veta vilka konventioner som gäller för DOM.

Det vanligaste och tydligaste sättet att representera DOM på, är att visa det som ett träd. Med träd menas en enkel sammanhängande graf utan cykler.

```
<html>
    <head>
        <title></title>
    </head>
    <body>
        <h1></h1>
        <p></p>
    </body>
</html>
```

Exempel 1kan även representeras som trädet i Figur 3

[8] DOM Scripting, kap. 2
[9] High Performance JavaScript, kap. 1
[10] https://wiki.mozilla.org/JaegerMonkey
[11] http://code.google.com/p/v8/
[12] http://my.opera.com/core/blog/2009/02/04/carakan
[13] http://www.apple.com/safari/features.html
[14] http://blogs.msdn.com/b/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx



Figur 3

Figur 3 är en representation av ett HTML-dokument. I dokumentet beskrivs endast strukturen på sidan men allt innehåll saknas. Då HTML- och XML-dokument innehåller taggar som i sin tur kan innehålla taggar är det enkelt att på ett överskådligt vis representera dokumentet som ett träd. Roten i trädet är i exemplet HTML-taggen. Under den finner vi först två grenar, head och body. Eftersom de ligger på samma nivå i grafen blir de syskon. Med syskon menas att de båda har samma nod över dem i grafen vilket i detta fall är HTML-taggen. Vilket också gör att de ses som barn till den. Taggen head har i sin tur egna barn vilket gör att head både är barn och förälder. Denna familjehierarkistruktur är viktig för att man ska kunna på ett enhetligt sätt traversera över noderna i trädet för att hämta den information som eftersöks.

I exemplet ovan användes endast en typ av nod. I DOM finns ett antal olika noder för att beskriva olika saker.

Elementnod: En elementnod är de noder som syns i Exempel 1. De har en motsvarande tagg i HTML- och XML-dokument. En elementnod kan innehålla andra noder även andra elementnoder. Vilket gör att den karaktäristiska trädstrukturen kan byggas upp.[15]

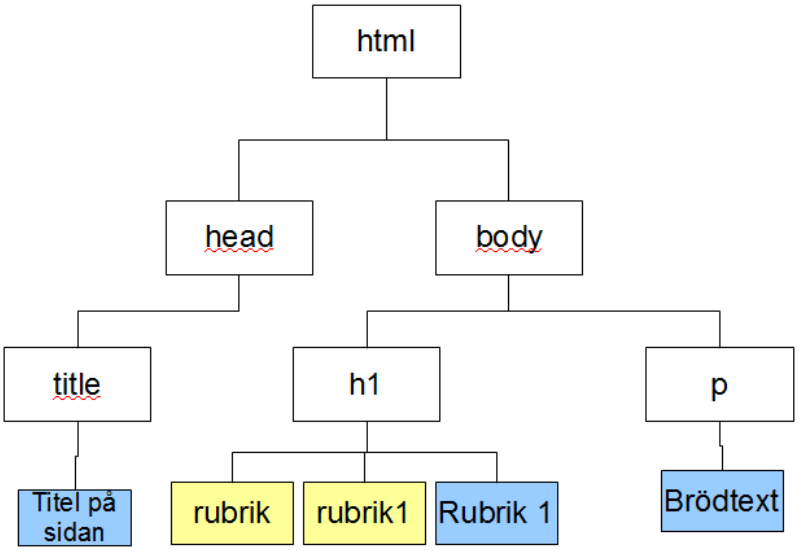
Textnod: En textnod är en annan typ av nod som är vanlig i HTML-dokument. Den används för att hålla ren text. Det är vanligt att den är ett barn till, till exempel elementnoden <p> för att hålla dess text. [16]

Attributnod: Attributnoder används för att beskriva specifik information om varje elementnod. Det kan både vara ett standardattribut från HTML som href eller title. Men kan även hålla egendefinierade attribut. Attributnoden håller både namnet på attributet och dess värde om det är definierat.[17]

[15] DOM Scripting, kap. 3
[16] DOM Scripting, kap. 3
[17] DOM Scripting, kap. 3

```
<html>
    <head>
        <title>Titel på sidan</title>
    </head>
    <body>
        <h1 class=”rubrik” id=”rubrik1”>Rubrik 1</h1>
        <p>Brödtext</p>
    </body>
</html>
```

Exempel 2 kan även beskrivas som trädet i Figur 4



Figur 4

2.2.1 DOM-Scriptande

För att kunna dra nytta av DOM-trädets egenskaper behöver man kunna välja ut noder samt kunna förflytta sig i trädet. Detta kan man göra på flera olika sätt. För att välja ut en bestämd elementnod kan man använda sig av HTML-attributet ID. Alltså samma ID som används av cascading style sheets(CSS) när sidan designas. DOM-trädet söks då igenom efter noden som har en attributnod med det unika ID:t.

Man kan även välja ut en mängd av alla noder genom att söka efter en viss typ av elementnoder som till exempel <p> vilket returnerar alla sådana noder i trädet. Som standard söker man igenom hela trädet men man kan även själv definiera vilken av noderna man vill börja sökningen ifrån.

DOMs kanske starkaste egenskap är att man kan lägga till och ta bort noder från trädet dynamiskt efter att dokumentet har laddats i en webbläsare. Vilket gör att man kan ändra innehållet på en sida utan att behöva gå till servern och ladda om allting. Man kan till exempel i JavaScript skapa egna noder och sedan applicera dem i dokumentet direkt efter inladdning eller beroende på olika användarinteraktioner. Vilket leder till en dynamisk sida som ändras efter användarens behov utan att behöva bero på sidomladdningar.

2.2.2 Prestanda och DOM

Att DOM används i alla webbläsare har öppnat många möjligheter för dynamiska webbapplikationer och gränssnitt. Det finns dock ett problem, att hantera och manipulera DOM-träd är kostsamt för applikationens prestanda. Då trädet manipuleras på klientsidan via klientscriptspråk ligger prestandakravet hos klienten. Det gör att det är extra viktigt att hålla ett öga på prestandan då det kommer att leda till att olika användare kan få helt olika upplevelser av gränssnittet.

”DOM scripting är kostsamt för prestanda och en vanlig flaskhals i rika webbapplikationer” – Stoyan Stefanov Kapitel 3 High Performance JavaScript

De tre största prestandabovarna i DOM-scripting är:

- Access och modifiering av DOM-träd
- Modifiering av attribut och stilar via DOM-trädet.
- Hanteringen av användarinteraktion genom DOM-events

Det är vanligt hos webbläsare att separera klientscript(ex JavaScript) och de genererade DOM-träden. I Internet Explorer till exempel lever JavaScriptet i filen jscript.dll medan DOM-implementationen lever i en annan fil mshtml.dll. Separationen gör att andra programmeringsspråk kan dra nytta av det genererade DOM-trädet. Men detta skapar en prestandabrygga då man vill nå DOM-trädet från JavaScriptet. Kostnaden för varje anrop till DOM filen är lika hög varje gång så prestandakostanden beror på antalet anrop. Det leder till att mycket dynamisk interaktion på sidan ger en hög prestandakostnad. Ett anrop sker varje gång en nod hämtas ur trädet och varje gång innehållet förändras i trädet. Vid en förändring kan webbläsaren behöva måla om sidan då förändringen kan påverkat vad användaren ser för tillfället. Detta leder då till en ännu högre prestandakostnad.[18]

2.3 Funktionsbibliotek

Funktionsbibliotek används för att underlätta och snabba upp utvecklingen av JavaScriptapplikationer. De brukar även innehålla ett antal visuella effekter som ger möjligheten att få element att glida in från sidan eller försvinna ut ur sidan på ett lite mer livfullt sätt.

jQuery har som slogan “write less do more” vilket stämmer bra in på de olika biblioteken. I de båda funktionsbiblioteken vi använder finns det till exempel möjlighet att anropa en enda funktion för att få element att glida in från sidan. Att skriva detta själv i JavaScript kräver en betydligt större arbetsinsats av utvecklaren mot att göra samma sak i funktionsbiblioteken, där behövs det en enda rad för att få det att fungera. De olika funktionsbiblioteken är helt enkelt utvecklade för att samla den vanligaste funktionaliteten på ett ställe och för att det ska räcka med att anropa en enda funktion för att köra en annars komplex funktionalitet.

Det finns ett antal olika funktionsbibliotek och i vår studie har vi koncentrerat oss på de två populäraste jQuery och Prototype.

[18] High Performance JavaScript, kap. 3

2.3.1 jQuery

jQuery använder sig av funktionen $() för att med hjälp av samma teknik som CSS hittar element i DOM-trädet, #-tecknet för att välja ett element med ett visst ID och en punkt(.) för att välja element med en viss klass. jQuerys funktionalitet omnämndes första gången augusti 2005 i skaparens blogg där han nämner ett JavaScriptbibliotek som använder samma teknik som CSS. Version 1.0 av jQuery släpptes ett år senare i augusti 2006.[19]

2.3.2 Prototype

Prototype kan välja element ur DOM-trädet på ett antal olika sätt. Om man ska välja ut ett element med ID:t element-id används funktionen $() med ID:t inom citationstecken på detta vis; $(”element-id”).[20] I motsats till jQuery går det bara att hämta element utifrån ett viss ID med $()-funktionen, för att hämta element med samma teknik som jQuery används istället $$()-funktionen.[21] Prototype har även en $F()-funktion som returnerar värdet av ett formulärelement utifrån ID:t på det elementet.[22]

[19] http://jquery.org/history/
[20] http://api.prototypejs.org/dom/dollar/
[21] http://api.prototypejs.org/dom/dollar-dollar/
[22] http://api.prototypejs.org/dom/dollar-F/

3 UTFORMNING AV EXPERIMENT

Det första testfallet genomförs genom att varje test laddar in biblioteket tio gånger som sedan körs tio gånger för att få fram ett relevant resultat på hur lång tid det tar för varje bibliotek att laddas in.

För att genomföra testfallen två till fyra har ett testramverk byggts i JavaScript utan att använda sig av något funktionsbibliotek. På detta vis ger vi inte något bibliotek fördel och för att försäkra sig om att vara webbläsaroberoende. Testramverket kör ett antal funktioner 20 gånger var och mäter hur lång tid det tar för funktionen att exekvera. Sedan beräknas ett medelvärde över de 20 gånger funktionen körs. Funktionen innehåller i sin tur en av de funktionsbiblioteksspecfika funktioner vi vill testa som körs i en for-loop med ett antal iterationer. Antalet iterationer varierar mellan de funktionsbiblioteksspecifika funktionerna då tiden det tar för dem att exekvera varierar kraftigt. Antalet iterationer används för att se till att funktionen exekverar i över 750 ms så att vi kan få ett bra snittvärde för funktionen.

Ramverket håller de funktioner som mäts rena från allt annat än metoden som ska testas, for-loopen och accesser till fördefinierade globala variabler. Ramverket hanterar även uppbyggnad samt städning när testfallet körts färdigt. Den kan även gå in mellan två körningar av en metod och antingen skapa noder eller variabler som behövs eller städa efter föregående test. Det håller även övriga DOM-interaktioner till ett minimum för att se till att testfallet kan köras så snabbt som möjligt.

Från medelvärdena som ramverket räknar fram behöver vi hitta en gemensam faktor för alla funktioner så vi kan jämföra dem med varandra. Eftersom funktionerna itererar olika antal gånger så kan vi inte bara jämföra hur lång tid de tar i millisekunder med varandra. För att kunna jämföra räknas hur många gånger en funktion hinner köra per sekund fram.

Körningar per sekund, KPS
T1 = Medelvärdet av mätvärdena
I = Antal iterationer, specifikt för funktionen

$$KPS = 1/((T1/I)/1000)$$

3.1 Testmiljö

Den dator som testerna körs på är hårdvarumässigt en typisk standarddator som kan finnas i var hem. Vi kör testerna under Windows för att ha möjlighet att köra testfallen i Internet Explorer. Det är just den här typen av datorer som troligtvis kör JavaScript applikationer idag.

3.1.1 Hårdvaruspecifikation

Processor

Specifikation:	Intel(R) Core(TM)2 Duo CPU E8500 @ 3,16GHz
Antal kärnor:	2 (max 2)
Antal trådar:	2 (max 2)
Kärnhastighet:	1997,9 MHz
L1 Data cache:	2 x 32 Kbytes 8-way set associative, 64-byte line size
L1 Instruction cache:	2x32 Kbytes 8-way set associative, 64-byte line size
L2 cache	6144 Kbytes, 24-way set associative, 64-byte line size

Moderkort

Modell:	P5Q SE
Tillverkare:	ASUSTeK Computer INC.

Minne

Minnestyp:	DDR2
Minnesstorlek:	4096 MBytes
Kanaler:	Dual, (Symetric)
Minnesfrekvens:	399,6 MHz
Databredd:	64 bits

3.1.2 Mjukvara och funktionsbiblioteksversioner

Mjukvara
Operativsystem: Windows Vista 64 SP2

Webbläsarversioner
Opera: 11.10
IE: 9.0.8112.1421
Firefox: 4.0.1
Chrome: 11.0.696.68
Safari: 5.0.5 (7533.21.1)

Scriptbibliotek
jQuery: 1.5.2
Prototype: 1.7

3.2 Testfall laddningstider

Test ett går ut på att låta testmiljön hämta hem de två olika funktionsbibliotek på ett antal olika sätt. I testet vill vi ta reda på hur platsen som scripten hämtas från påverkar laddningstiden samt hur storleken på filen påverkar laddningstiden. Eftersom laddningstiden påverkar hur snabbt en användare kan börja arbeta med webbgränssnittet ska testet utvisa vilket av de valda funktionsbiblioteken som laddar snabbast totalt över de tre deltesterna.

Två deltester:
- Hämta mot servern/lokalt mot att hämta från repository
- Minimerade mot vanliga scriptfiler

I deltest ett ska laddningstiden på ominimerade varianter av funktionsbiblioteken mätas. Först mäts tiden det tar att lokalt på testmiljön ladda in scriptfilen i webbläsaren samt köra den kod som behövs för att lösa de beroenden som kan uppstå vid eventuella flera filer. Sedan kommer testet hämta hem samma opaketerade bibliotek från en extern källa på internet och sedan jämföra tiderna. Testet är skapat då det har blivit vanligare att ladda alla beroenden av funktionalitetsbibliotek i en webbapplikation från en extern källa. Att ha funktionsbiblioteken på en extern källa gör det enklare att flytta eller strukturera om applikationen, då sökvägarna till funktionsbiblioteken inte behövs skrivas om.

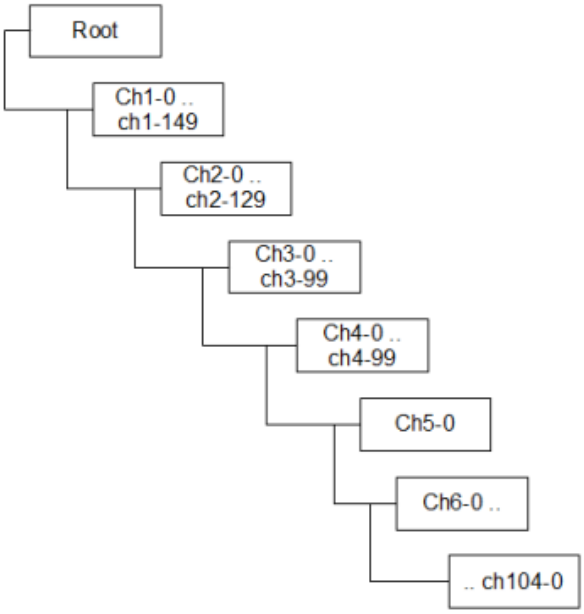
Deltest två undersöker hur funktionsbibliotekens storlek påverkar deras laddningstider. Det finns verktyg för att komprimera JavaScriptfilerna från vanlig källkod till en minimerad variant som sedan laddas. Den minimerade varianten blir mindre i storlek men om man öppnar den är den mycket svårläst då den vanligtvis är på en rad och saknar mellanslag, tabbar och radbrytningar beroende på minimeringsverktyg. Det verktyg vi använder för detta är YUI Compressor.[23] De minimerade filerna läses bara in lokalt från testmiljön då Prototype inte finns officiellt någon minimerad version på nätet.

3.3 Testfall traversering och manipulation

I testfall två ställs de två funktionsbiblioteken mot två uppgifter. I den första testas funktioner för att traversera DOM-trädet. Hur snabbt kan de olika biblioteken nå en föräldernod, syskonnod, barnnod, hämta en mängd av barn etc. Den andra uppgiften består av att testa bibliotekens funktioner för att modifiera DOM-trädet vi testar på. Genom att placera nya noder i början och i slutet av nivåer samt lägga till nya barn.

Figur 5 visar trädet som används i detta testfall, det består av en rot som har 150 barn som sedan i sig har 150 barn och dessa barn har ytterligare 130 barn. Ett av dessa barn har ytterligare 100 barn och ett av dessa har 100 barn. Ett av barnen på den här nivån har ett barn och sedan fortsätter trädet med att varje barn har ett barn fram till dess att det finns totalt 104 nivåer i trädet. Alla element har en klass som visar vilken nivå elementet ligger på och vilket element i ordningen det är på just den nivån. Till exempel ch2-3 visar att elementet ligger på nivå 2 och är det fjärde elementet i ordningen.

[23] http://refresh-sf.com/yui/



Figur 5

3.4 Testfall attribut och stilar

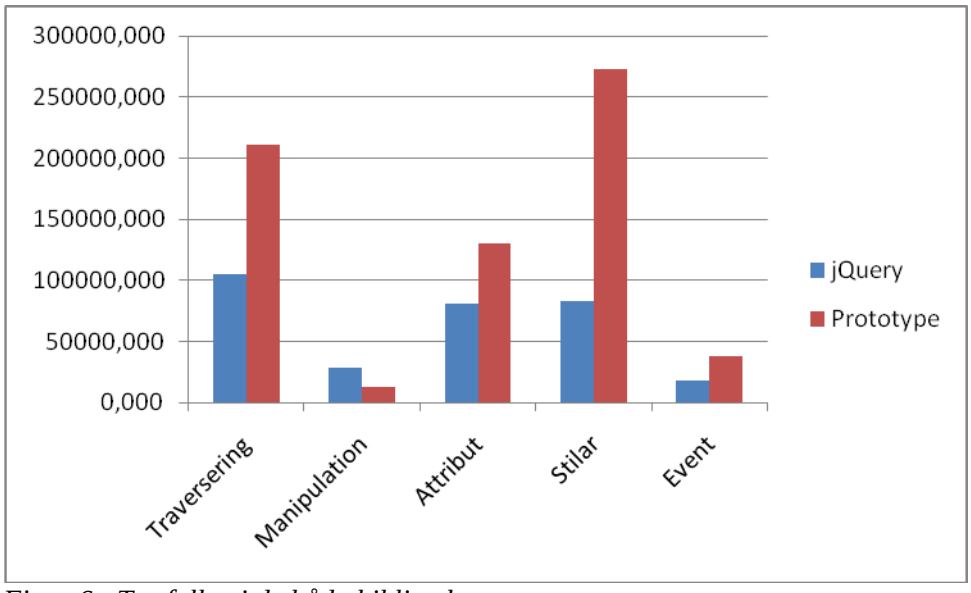
I testfall tre testas bibliotekens funktioner för att ändra och läsa attributnoder samt hur de dynamiskt kan förändra dokumentets stilmallar efter laddning. Som i test två testas varje funktion enskilt i de båda biblioteken för att sedan läggas ihop till en totalsumma för hela testet.

3.5 Testfall eventrespons

I testfall fyra kommer ett antal eventlyssnare sättas, triggas och tas bort med hjälp av bibliotekens funktioner. Eventen som sätts och triggas är tomma JavaScriptfunktionskroppar.

4 RESULTAT

Alla mätningar är utförda med motsvarande funktioner i de båda biblioteken för att utföra en så rättvis bedömning som möjligt. I figurerna representerar y-axeln antal körningar per sekund, en hög stapel är ett snabbare resultat.



Figur 6 - Testfallen i de båda biblioteken

Biblioteksjämförelse över alla webbläsare	**Firefox**	**Chrome**	**Safari**	**Opera**	**IE**
jQuery	59 174,50	86 407,4	82 053,56	56 494,83	33 132,99
Prototype	84 340,54	356 374,5	110 970,9	95 284,9	17 750,29

Tabell 2 – Jämförelse över alla webbläsare, värdet representerar antal körningar per sekund

4.1 Resultat testfall laddningstider

Tiden i Tabell 3 är ett medelvärde över 100 hämtningar av den specifika filen. Testet visar att om biblioteket hämtas från en extern källa tar det dubbelt så lång tid för biblioteket att laddas in än om det legat lokalt på datorn.

Fil	**Storlek (KB)**	**Tid (ms)**
jquery-min-yui.js	92,8	20,5
jquery-1.5.2.js	214,1	19,8
http://code.jquery.com/jquery-1.5.2.js	214,1	44,1
prototype-min.js	90,0	20,1
prototype.js	159,5	24,0
https://ajax.googleapis.com/ajax/libs/prototype/1.7.0.0/prototype.js	159,5	57,3

Tabell 3 - laddningstider

4.2 Resultat testfall traversering och manipulation

Testfallet visar att jQuery hade stora problem med att hämta ut en nods alla föräldrar med funktionen parents. Tabellerna nedan visar hur webbläsaren Firefox presterade i testfallet för traversering i de båda biblioteken.

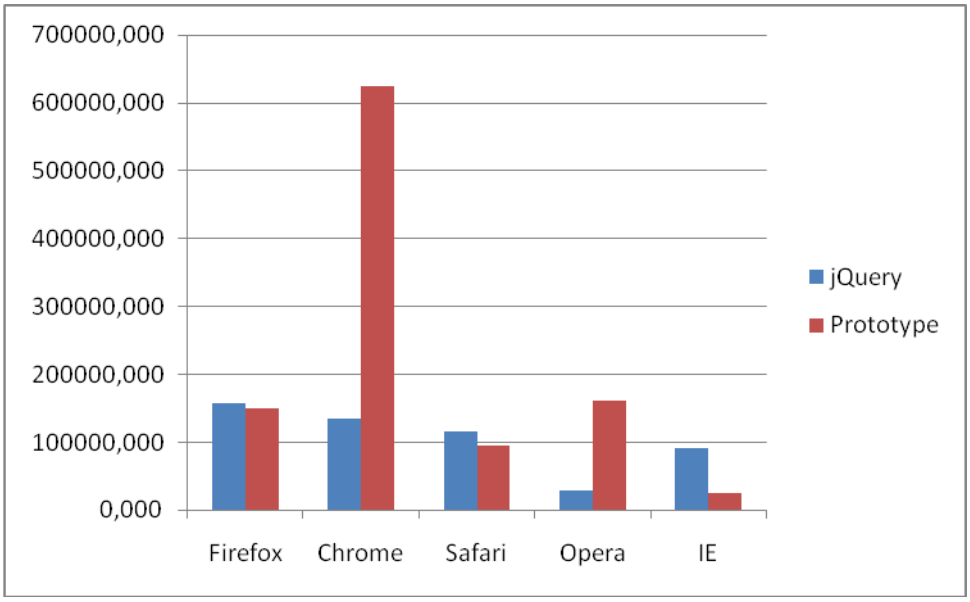
Traversering jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Parent	873	400 000	458 190,149
Parents	1 315,85	5	3,800
Prev	1 091,95	298 000	272 906,269
PrevAll	6 177,35	30 000	4 856,451
Next	1 096,10	298 000	271 873,004
NextAll	6 751,65	30 000	4 443,358
Siblings	7 620,70	50 000	6 561,077
Children	1 209,80	300 000	247 974,872

Tabell 4 – Resultat av traversering i jQuery

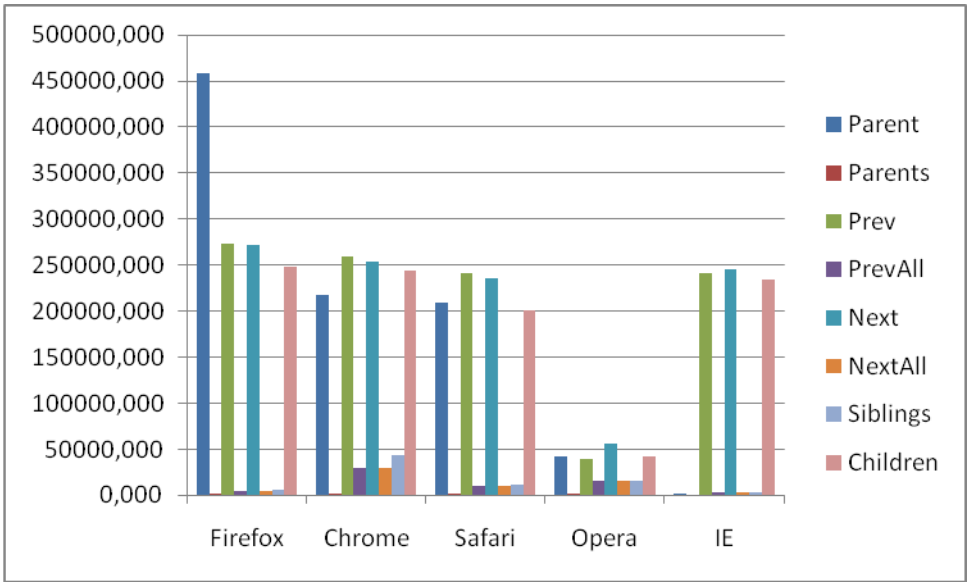
Traversering Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Up	6 035,95	3 000 000	497 022,010
Ancestors	1 888,85	30 000	15 882,680
ChildElements	6 505,40	1 490 000	229 040,489
Next	7 076,75	2 980 000	421 097,255
Siblings	2 062,50	20 000	9 696,970
PreviousSiblings	1 891,50	20 000	10 573,619
Previous	10 854,70	30 000	2 763,780
NextSiblings	1 899,65	20 000	10 528,255

Tabell 5 – Resultat av traversering i Prototype

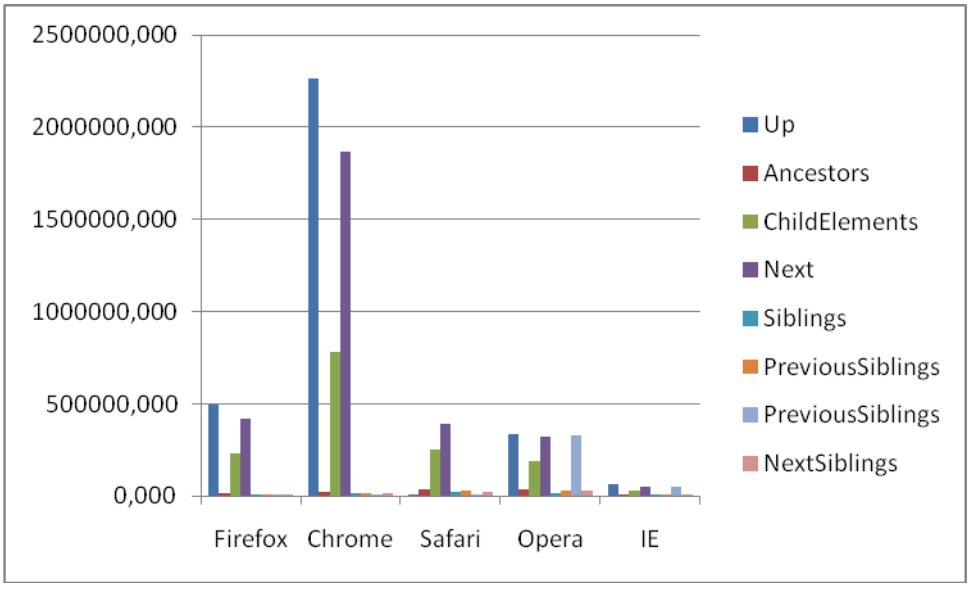
Traverseringen i de båda biblioteken hade ett spritt resultat mellan de olika webbläsarna. Prototype presterade mycket bättre i Chrome och Opera medan jQuery presterade bättre i de resterande webbläsarna. Som Figur 9 visar har Up och Next en bidragande orsak till hur resultatet ser ut i Chrome.



Figur 7 - Testfall traversering

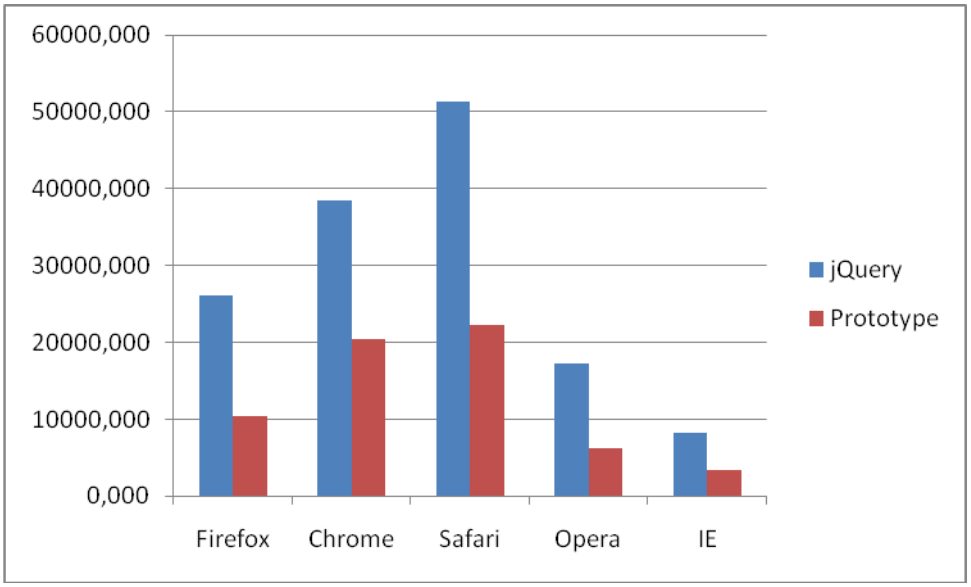


Figur 8 - Traversering jQuery

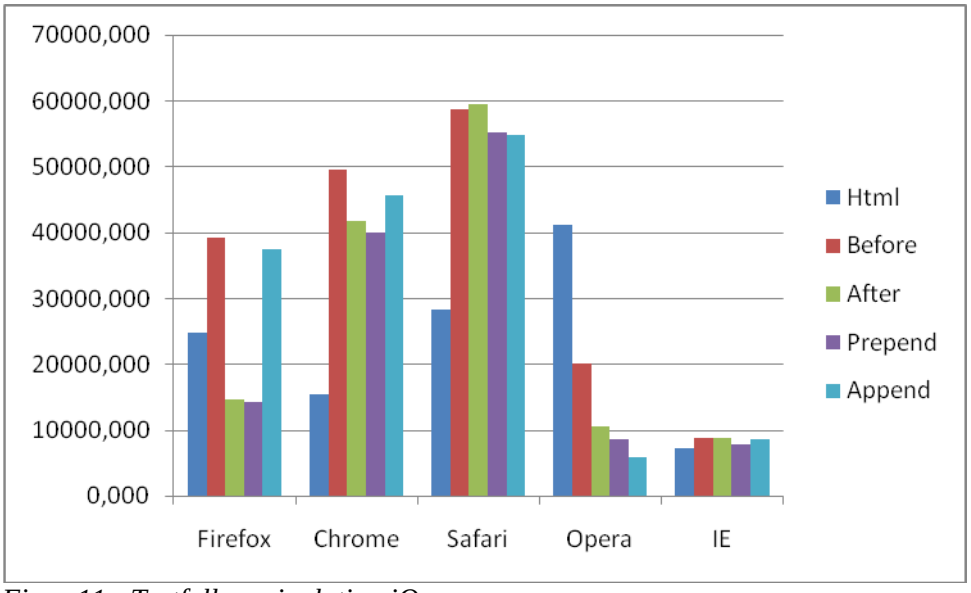


Figur 9 - Traversering Prototype

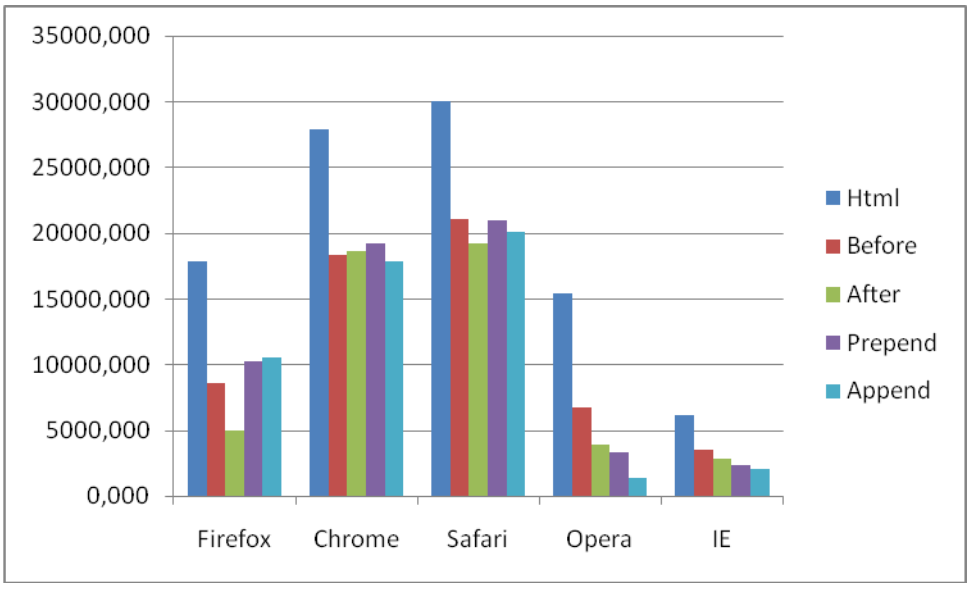
Testet där biblioteken manipulerar DOM-trädet visar att jQuery har nästan dubbelt så många körningar per sekund som Prototype i samtliga webbläsare.



Figur 10 – Testfall manipulation

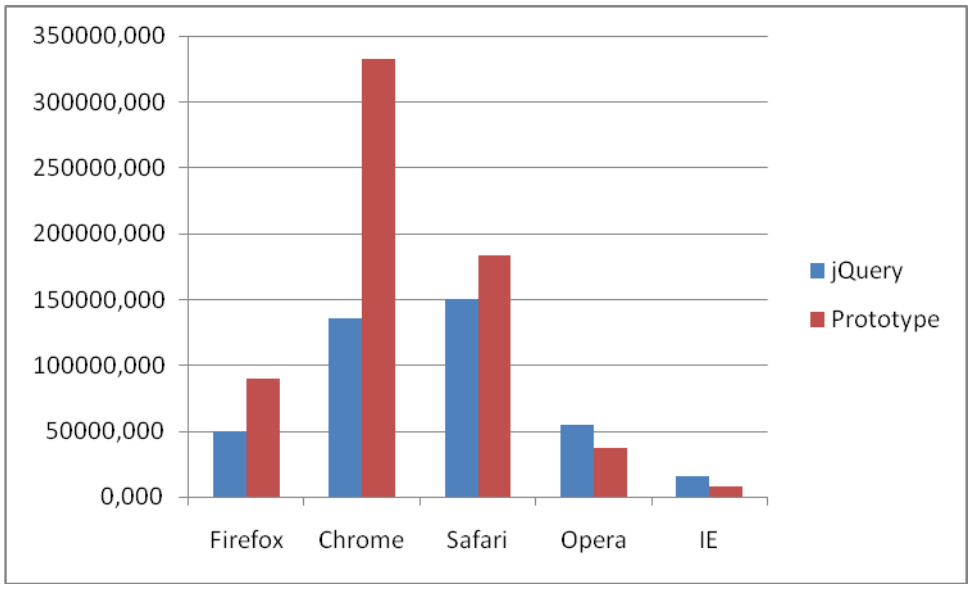


Figur 11 - Testfall manipulation jQuery

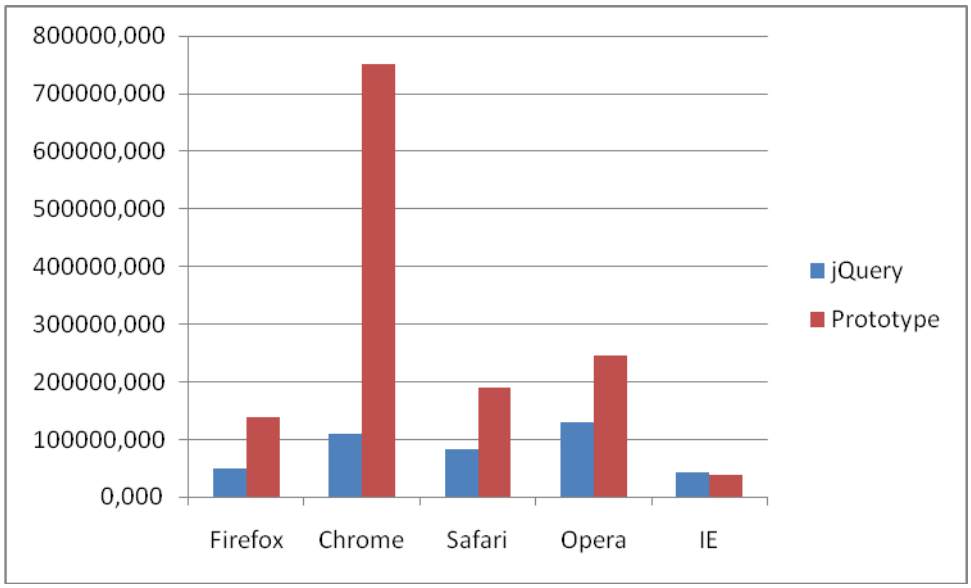


Figur 12 - Testfall manipulation Prototype

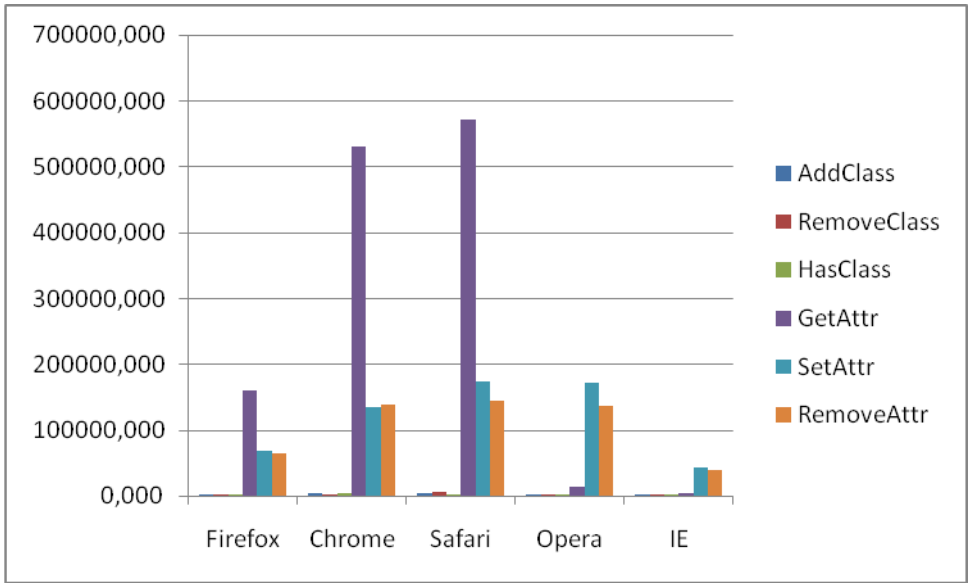
4.3 Resultat testfall attribut och stilar



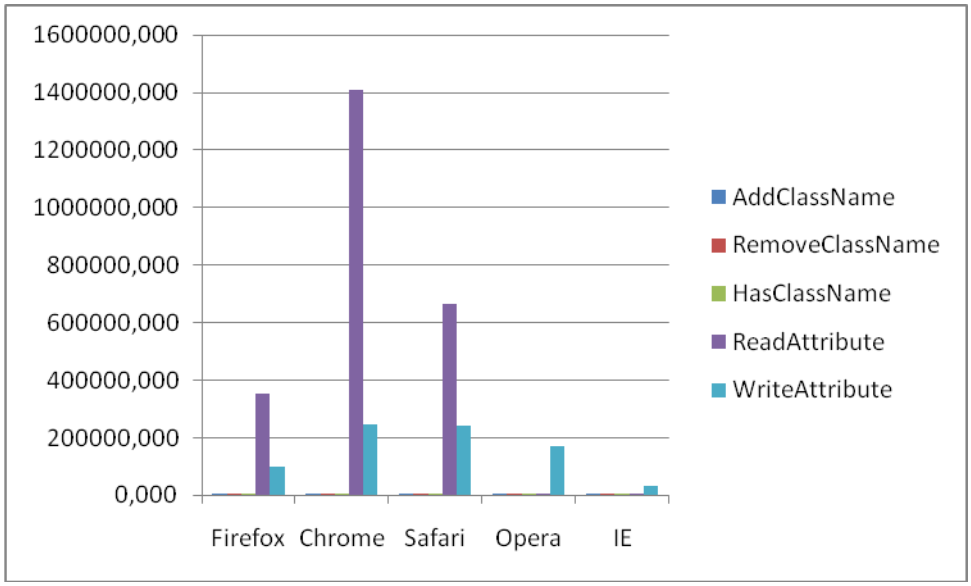
Figur 13 - Testfall attribut



Figur 14 - Testfall stilar



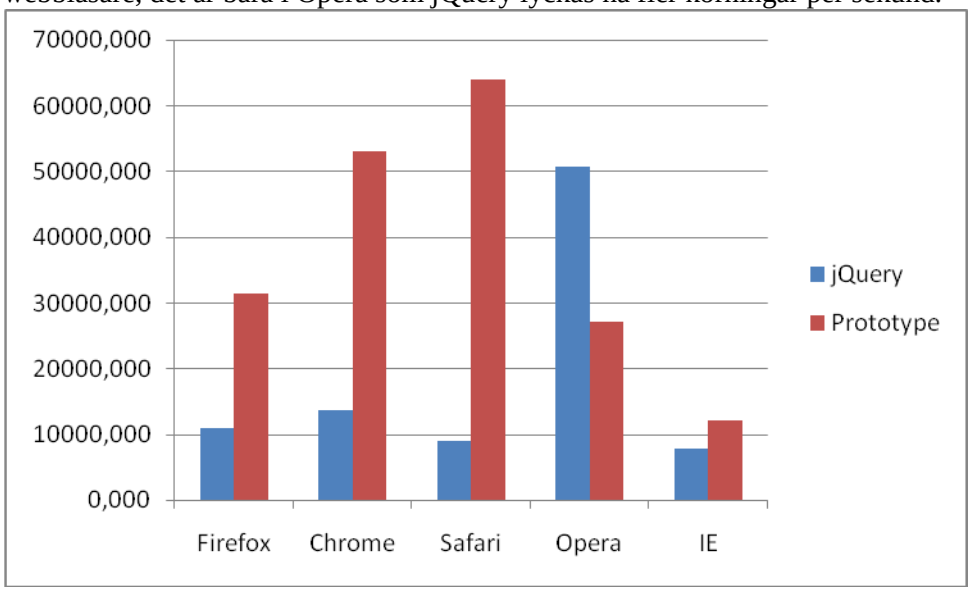
Figur 15 - Testfall attribut jQuery



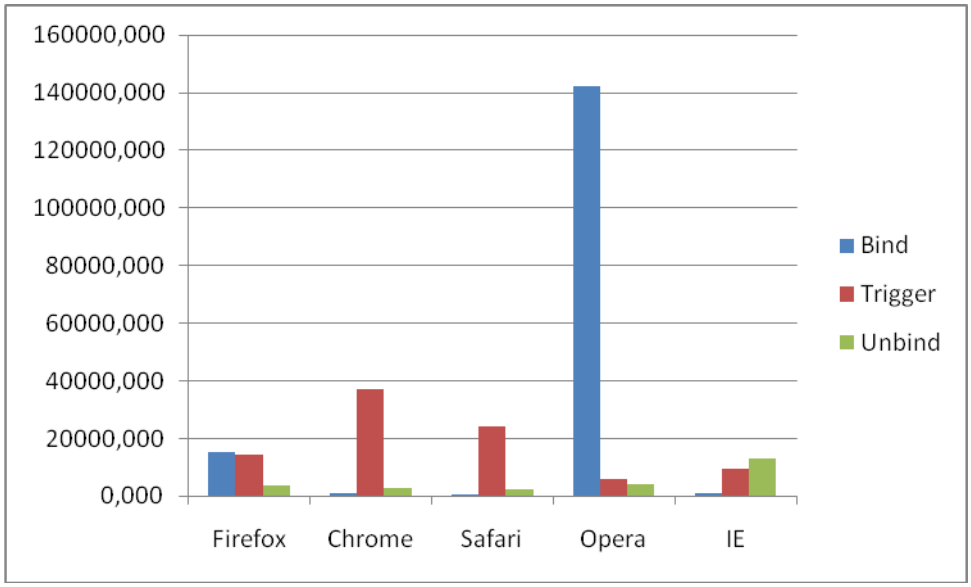
Figur 16 - Testfall attribut Prototype

4.4 Resultat testfall eventrespons

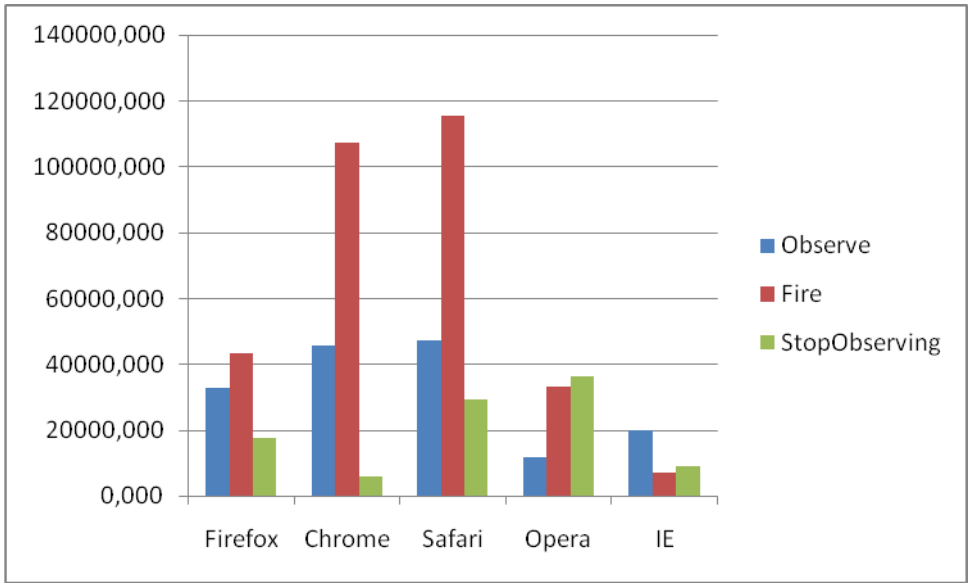
Återigen visar resultaten att Prototype är klart snabbare jQuery i flertalet webbläsare, det är bara i Opera som jQuery lyckas ha fler körningar per sekund.



Figur 17 - Testfall eventrespons



Figur 18 - Testfall eventrespons jQuery

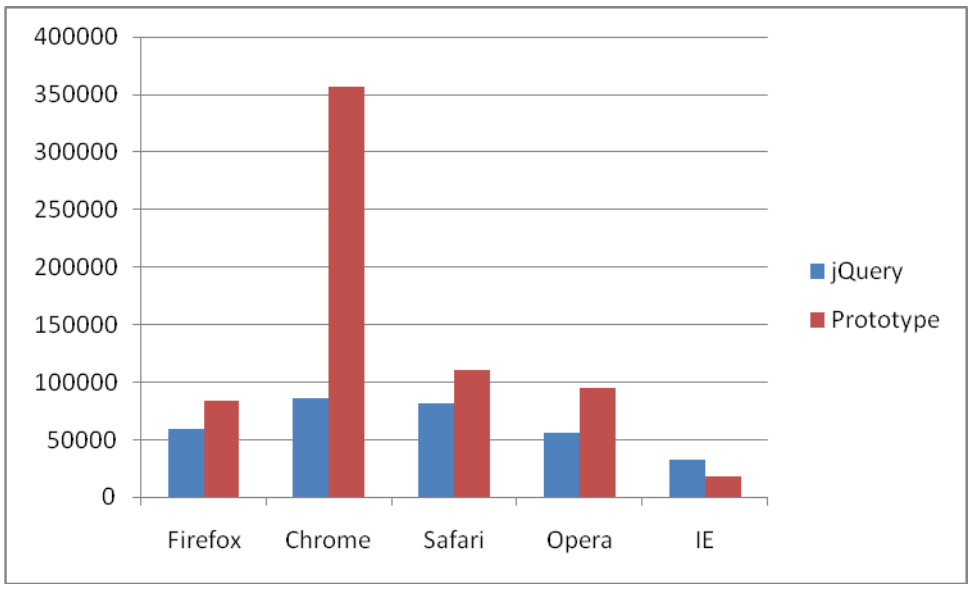


Figur 19 - Testfall eventrespons Prototype

5 DISKUSSION

Resultatet i Tabell 3 visar en tydlig skillnad på laddningstider om filen ligger lokalt eller som en extern fil som måste laddas ner från en annan server. När man sedan tittar på resultatet när filerna ligger lokalt och det endast är storleken som skiljer de åt, ser man att filstorleken inte påverkar laddningstiden nämnvärt.

Det vi kan utläsa ur resultaten i Figur 6 har Prototype kunnat köra flest antal iterationer av varje funktion per sekund. Det enda testfallet där jQuery hade ett överlägset bättre resultat var där DOM-trädet skulle manipuleras. Som Figur 10 visar kan jQuery köra i princip dubbelt så många iterationer i alla webbläsare mot vad Prototype kan prestera. Om man tittar närmare på hur funktionerna körs i Figur 11 och Figur 12, kan man se att det är Safari som drar upp resultatet för jQuery med nästan tre gånger så många körningar per sekund för nästan alla funktioner. Det är bara Html som körs i princip lika många gånger per sekund i både Prototype och jQuery i Safari.



Figur 20 - Snabbaste webbläsaren

I Figur 20 ser vi att Chrome gick snabbast med både Prototype och jQuery sett över ett medelvärde av alla funktioners körningar per sekund i alla våra testfall. Detta var vad vi hade förväntat oss då Chrome har haft ett bra rykte vad det gäller JavaScriptprestanda och var tidiga ute med sin förbättrade JavaScriptmotor. Att IE var långsammast i våra tester är inte speciellt förvånande då de under att antal versioner haft markant sämre hantering av JavaScript än övriga webbläsare. Vi förväntade oss dock att IE skulle prestera så pass bra som den gjorde med den senaste versionen Internet Explorer 9 ser Microsoft ut att ha gjort goda framsteg. [24]

I Figur 20 ser vi också att Safari som presterade bra i våra tester, så pass bra att den blir näst snabbaste webbläsaren efter Chrome. Vilket är lite imponerande då Safari först och främst är byggd för Mac och nu istället kördes under Windows med de övriga läsarna. Tabell 2 visar även att Opera går snabbare än Firefox med totalt 143 514 mot 151 742 körningar per sekund.

[24] http://ie.microsoft.com/testdrive/benchmarks/sunspider/default.html

jQuery testerna i Firefox körde som man kan se i siffrorna nedan marginellt snabbare än Operas men även Opera slår Firefox och får en bättre totaltid på grund av snabbbare Prototype exekvering.

Firefox jQuery: 59174 KPS
Opera jQuery: 56494 KPS
Firefox Prototype: 84340 KPS
Opera Prototype: 95284 KPS

Att hämta och sätta attribut som visas i Figur 14 går åter igen snabbast i Prototype i alla webbläsare utom Internet Explorer. Det som gör att resultatet i Chrome är så högt ser vi i Figur 15 och 16, funktionen ReadAttribute kör närmare 1 400 000 körningar per sekund och motsvarande funktion GetAttr i jQuery bara kör runt 530 000 körningar per sekund.

Testfallet Event visar i Figur 17 ett intressant resultat i webbläsaren Opera där jQuery visar sig prestera bättre än vad Prototype gör. Men om man sedan tittar på hur varje funktion kördes i webbläsarna ser man i Figur 18 att funktionen Bind i jQuery får ett väldigt avvikande resultat mot alla andra webbläsare. För oss tyder detta på att något inte stöds eller att funktionen Bind inte riktigt har körts på ett korrekt sätt i just denna webbläsare. Då vi inte alls ser samma pik i Prototypes eventhantering i Figur 19 så tyder detta på att något inte stöds eller att funktionen bind inte riktigt har körts på ett korrekt sätt i just denna webbläsare. Annat interessant att notera angående events i Figur 18 och 19 är att det varierar vad som tar längst tid att sätta ett event med Bind och Observe eller att ta bort det med Unbind och StopObserving. Överlag ser det ut som att ta bort ett event tar längre tid än att knyta event i jQuery medan i Prototype så är det tvärt om.

I figur 20 ser vi också att över alla testfall är Prototype överlag snabbare i alla webbläsare utom Internet Explorer. Jämförs testfall för testfall ser allt ungefär ut som i övriga webbläsare. jQuery går snabbare i manipulationen medan Prototype vinner resten, förutom traverseringen i Figur 7. Där jQuery går snabbare i tre av webbläsarna men Prototype klarar av fler körningar per sekund på grund av ett bra resultat i Chrome. Som man kan se i Figur 8 och 9 så går funktionerna Prev, Next och Children i jQuery betydligt snabbare än i Prototype**.** Prev och Next går ca 4,5 gånger snabbare i jQuery än motsvarande funktion i Prototype och Children går strax över 8,5 gånger snabbare enligt tabellerna jQuery Traversering och Prototype Traversering i Bilaga 9.1.5 Internet Explorer. Dessa resultat ser ut att vara den största enskilt bidragande faktorn till att jQuery totalt sett går snabbare än Prototype i Internet Explorer.

Testet av traversering med jQuery i Firefox i Tabell 4 visar ett väldigt intressant värde där funktionen Parents endast behöver köras fem gånger för att nå upp i ett medelvärde på 1 315 ms vilket är väldigt lite om man jämför med motsvarande funktion i Prototype i Tabell 5 som kör 30 000 iterationer för att komma upp i 1 888,85 ms.

Funktionerna Is som kontrollerar om ett element är dolt eller synligt, Show som sätter ett element till synligt och Hide som döljer ett element ställer till vissa problem i våra mätningar. Det är just dessa funktioner som gör att Prototype får så höga värden i Figur 14. Beroende på implementationen i funktionsbiblioteken och utförandet i de olika webbläsarna kan de ta väldigt olika tid.

Problemet ligger i att vi itererar över funktionen och försöker i till exempel Hide dölja ett element. På varv två och framåt i loopen kommer vi då köra Hide på ett

redan dolt element vilket i princip inte tar någon tid alls. Det bästa hade varit att kunna köra Hide på ett då synligt element. Men då hade vi antigen fått sätta tillbaka elementet till synligt efter varje iteration i loopen vilket hade kostat tid och påverkat mätningen eller så hade vi behövt dölja ett annat element. Men även där hade vi behövt traversera testträdet eller hämtat en ny nod, vilket hade varit fler DOM-interaktioner och därför kostat en hel del tid och påverkat mätningsvärdet. Det hade kanske blivigt mindre påverkan på testfallet i sin helhet om vi hade exkluderat dessa funktioner. Toggle som är en liknande funktion fungerar däremot utmärkt att iterera över, då den döljer elementet om det är synligt eller gör elementet synligt om det är dolt.

Då Prototype inte har någon removeAttributefunktion saknas en funktion i testfallet för attribut, detta kan påverkat Prototypes resultat åt något håll. Funktionerna för att ta bort ett element helt från DOM-trädet kunde också varit med i manipulationstestet då de bör vara ganska tunga tidsmässig.

Det hade kanske varit enklare att räkna och jämföra mätvärdena på de olika funktionerna om de hade itererat samma antal gånger i alla olika webbläsare. Samt att alla funktioner i sig hade itererat samma antal gånger. Tanken var att göra så från början men det ställde till problem då det var för stora skillnader mellan de olika funktionerna och hur snabbt webbläsarna arbetade. Vilket gjorde att vi istället valde att variera antalet iterationer beroende på webbläsare, men fortfarande såg till att funktionen kördes i de 750ms som var satt som gräns för minsta tid ett test fick köras.

5.1 Vad kan påverkat mätningarna

För att mäta tiden i våra testfall använder vi oss av följande kodstycke. Det tar dagens datum och klockslag när funktionen startar och mätningen slutar. Sedan drar vi av starttiden från sluttiden för att få fram tiden vår funktion exekverat.

```
var timeDiff = {
   setStartTime:function(){
     d = new Date();
     time = d.getTime();
   },
   getDiff:function (){
     d = new Date();
     return (d.getTime()-time);
   }
}
```

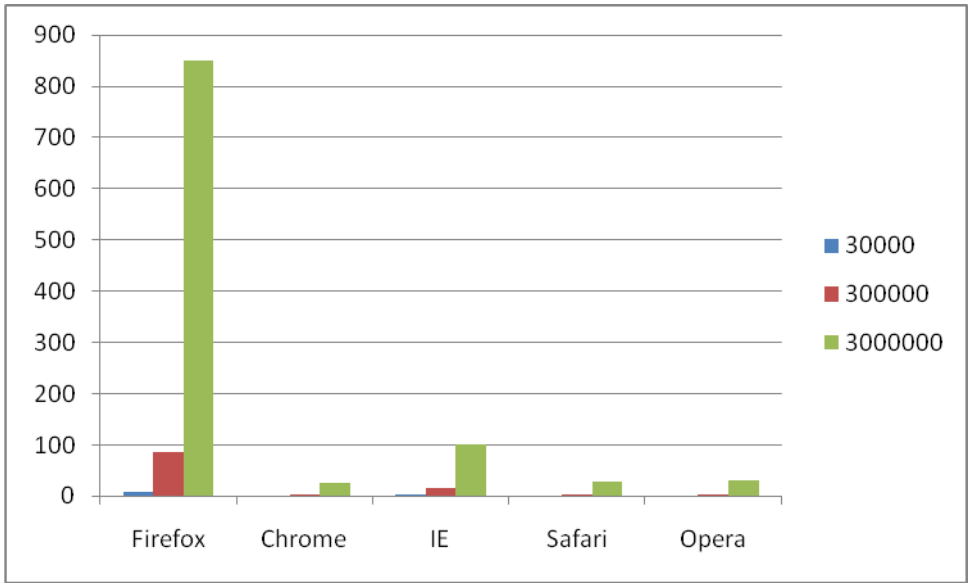
Samma princip används i alla testverktyg som finns för JavaScript idag. Det har dock ett problem. På windowsbaserade system finns en fördröjning som gör att webbläsarna ligger skiftar den mellan 0 och 15 millisekunder efter den riktiga tiden beroende på när systemtiden senast blev uppdaterad. Detta ställer till problem då våra funktioner inte tar lång tid alls att exekvera och påverkas därför av en felmarginal. [25]

För att motverka denna felmarginal har vi i våra test försökt att se till att varje funktion exekverar så många gånger att det tar minst 750 ms. Felmarginalen ligger då runt 1 % av resultatet eftersom medelvärdet av noll och 15 är 7,5 och 7,5 genom 750 är 0,01.

[25] High Performance JavaScript, kap. 6

Mätvärdena vid inladdning av de olika biblioteken i testfall ett kan påverkas av andra processer på datorn som skriver eller läser på hårddisken där filen ligger. Den del av testet där filen hämtas från en extern källa kan den tillfälliga belastningen på nätet och klientens bandbredd påverka hur snabbt det går att hämta hem de olika filerna. I övriga testfall har internetåtkomsten stängts av för att minska den externa påverkan på mätvärdena. Övriga processer har också hållits till ett minimum.

I testfall två till fyra bortser vi från tiden det tar att iterera över de funktioner som testas då iterationerna inte består av så stor del av exekveringstiden. Nedan följer våra mätningar på iterationstiderna och dessa mätningar påverkas också av 0-15 millisekunders felmarginal som beskrevs tidigare.



Figur 21 - Iterationstider

Figur 21 visar lite intressanta resultat. Vi antog att iterationskostnaden i de olika webbläsarna är så pass låg att den kan räknas bort eftersom vi inte räknade med att behöva iterera mer än 30 000 gånger över någon funktion. Men för att nå upp till timern på 750 ms visade sig att några av funktionerna behövde fler iterationer på sig i de snabbare webbläsarna. På de flesta funktioner fungerade det bra på 30 000 iterationer och kostnaden för detta ser vi som försumbar och inom den redan befintliga 1% felmarginalen på 0-15 millisekunder. Undantagen från detta var främst Prototypes funktioner Next och Up som båda kör tre miljoner gånger i både Chrome och Firefox. Även i Safari behövde funktionen Next köras ungefär 3 miljoner gånger. Iterationskostnaden i Firefox för tre miljoner iterationer kan bidragit till att Firefox inte presterat så pass bra som vi trodde den skulle göra.

6 SLUTSATS

Testresultaten har både förvånat och förvirrat oss lite. I början trodde vi att jQuery var populärast på grund av att det presterade bäst, men som vi nu har sett i resultaten har Prototype presterat bättre än jQuery i nästan alla enskilda testfall överlag. Det finns vissa webbläsare som varit bättre eller sämre på att hantera de båda biblioteken men överlag har Prototype tagit hem de flesta testerna.

Att Firefox ser ut att ha underpresterat något i våra tester kan ha med utformningen av testfallen att göra. I övrigt gick alla webbläsare riktigt snabbt och det ser ut som mycket har hänt på JavaScriptfronten de senaste åren. Snabbast av alla webbläsare var Chrome med ganska god marginal. Internet Explorer var den enda webbläsaren som stack ut ur mängden och jQuery överlag presterade bättre än Prototype i testerna.

Vår hypotes att jQuery skulle prestera bättre än Prototype i ett webbgränsnitt med DOM-interaktioner ser enligt våra testresultat inte ut att stämma. Prototype presterade bäst i våra tester sett över alla testfall och webbläsare. Undantaget på testfallen var manipulation där jQuery var snabbast och det kan tänkas att det finns specialfall där en webbapplikations prestandaflaskhals är just manipulationen. Då kan jQuery vara ett alternativ ur prestandaperspektiv, i alla andra fall presterar Prototype bäst.

Vårt resultat gör att alla som använder jQuery inte ska byta till Prototype bara för att det presterar mycket bättre i flertalet tester. Att manipulation görs så mycket snabbare i jQuery kan vara en anledning till att fortsätta använda jQuery och hoppas på att prestandan utvecklas även i de andra fallen. Får Prototype bättre prestanda vid manipulation i kommande versioner utan att påverka de andra funktionernas prestanda kan Prototypes popularitet öka mycket i framtiden.

6.1 Förslag på fortsatt forskning

Under arbetets gång har nya funderingar och frågor dykt upp som vi inte har haft möjlighet att studera. I vårt arbete har vi tittat på hur olika javascriptbibliotek har presterat i DOM-interaktioner. Vad som inte testas mer än i vårt lilla iterationstest är hur prestandan för JavaScript har förändrats i de nya JavaScriptmotorerna till webbläsarna idag. Redan i det testet ser vi att Firefox hamnar långt efter internet Explorer som i sin tur ligger en bit bakom de andra. Det vore intressant att undersöka om det endast gäller för for-loopen eller om det även stämmer för andra loopkonstruktioner och övrig grundläggande JavaScriptprestanda.

Man skulle även kunna titta på andra funktioner i de två JavaScriptbiblioteken, då vi endast testat en del av de funktioner som används vid DOM-interaktioner och inget annat. Till exempel skulle man kunna undersöka hur de båda biblioteken hanterar Ajax-requests och om det är någon skillnad på prestandan där. Samt om det är någon skillnad på vilken webbläsare som används. Då webbaserade applikationer blir mer och mer interaktiva och behöver klara av en högre belastning av Ajax-lösningar än tidigare. Det är förmodligen lite svårare att mäta Ajax-requesten då allt som går över Internet har så många externa faktorer som spelar in. Det finns idag stöd i de flesta webbläsare för någon form av utvecklarverktyg där man enkelt kan se och profilera Ajax-request.

Nya versioner av biblioteken släpps kontinuerligt och den 12 maj släpptes en ny version av biblioteket jQuery. Då detta var mitt i arbetet hade vi inte möjlighet att byta ut vår kod för att stödja det nya biblioteket. Det nya innehåller bland annat förändringar på hur jQuery hanterar DOM-attribut vilket hade varit intressant för vår undersökning. Så att testa detta nya bibliotek och eventuella flera uppdateringar kommer vara intressanta.

7 ORDLISTA

DOM - Document Object Model

JS - JavaScript

CSS - Cascading Style Sheets

HTML - Hyper Text Markup Language

XHTML - eXtensible HyperText Markup Language

XML - eXtensible Markup Language

Ajax - Asynchronous JavaScript and XML

IE - Internet Explorer

ECMA - European Computer Manufacturers Association

KPS – Körningar Per Sekund

8 LITTERATURFÖRTECKNING

8.1 Böcker

DOM Scripting, Jeremy Keith, Apress 2005, ISBN: 978-1-59059-533-6

High Performance JavaScript, Nicholas C. Zakas, Yahoo Inc. 2010,
ISBN: 978-0-596-80279

8.2 Webbsidor

Google Trends, (Besökt 2011-05-20)
http://www.google.com/trends?q=Yahoo+javascript%2C+Prototype+javascript%2C+Jquery+javascript&ctab=0&geo=all&date=all&sort=0

W3Schools Browser Satistics, (Besökt 2011-05-20)
http://www.w3schools.com/browsers/browsers_stats.asp

ECMA International 262 Standard, (Besökt 2011-05-20)
http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

The Mozilla Foundation wiki, (Besökt 2011-05-20)
https://wiki.mozilla.org/JaegerMonkey

About v8 JavaScript Engine, (Besökt 2011-05-20)
http://code.google.com/p/v8/

Opera Core Concerns - Charakan, (Besökt 2011-05-20)
http://my.opera.com/core/blog/2009/02/04/carakan

Safari Browser Information, (Besökt 2011-05-20)
http://www.apple.com/safari/features.html

MSDN Blog IEBlog, (Besökt 2011-05-20)
http://blogs.msdn.com/b/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx

jQuery Project History, (Besökt 2011-05-20)
http://jquery.org/history/

Prototype API Documentation, (Besökt 2011-05-20)
http://api.prototypejs.org/dom/dollar/

Prototype API Documentation, (Besökt 2011-05-20)
http://api.prototypejs.org/dom/dollar-dollar/

Prototype API Documentation, (Besökt 2011-05-20)
http://api.prototypejs.org/dom/dollar-F/

Online YUI Compressor, (Besökt 2011-05-20)
http://refresh-sf.com/yui/

WebKit SunSpider JavaScript Benchmark Results, (Besökt 2011-05-20)
http://ie.microsoft.com/testdrive/benchmarks/sunspider/default.html

9 BILAGOR

9.1 Resultattabeller

9.1.1 Firefox 4.0.1

Traversering jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Parent	873	400 000	458 190,149
Parents	1 315,85	5	3,800
Prev	1 091,95	298 000	272 906,269
PrevAll	6 177,35	30 000	4 856,451
Next	1 096,10	298 000	271 873,004
NextAll	6 751,65	30 000	4 443,358
Siblings	7 620,70	50 000	6 561,077
Children	1 209,80	300 000	247 974,872

Traversering Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Up	6 035,95	3 000 000	497 022,010
Ancestors	1 888,85	30 000	15 882,680
ChildElements	6 505,40	1 490 000	229 040,489
Next	7 076,75	2 980 000	421 097,255
Siblings	2 062,50	20 000	9 696,970
PreviousSiblings	1 891,50	20 000	10 573,619
Previous	10 854,70	30 000	2 763,780
NextSiblings	1 899,65	20 000	10 528,255

Manipulation jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Html	1 209,85	30 000	24 796,462
Before	1 017,20	40 000	39 323,634
After	2 033,70	30 000	14 751,438
Prepend	2 107,80	30 000	14 232,849
Append	800,30	30 000	37 485,943

Manipulation Prototype			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Html	1 676,15	30 000	17 898,159
Before	3 476,10	30 000	8 630,362
After	5 987,95	30 000	5 010,062
Prepend	2 934,40	30 000	10 223,555
Append	2 844,75	30 000	10 545,742

Attribut jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
AddClass	1 321,30	1 500	1 135,246
RemoveClass	1 352	1 500	1 109,467
HasClass	1 083,90	3 000	2 767,783
GetAttr	1 878,40	300 000	159 710,392
SetAttr	1 455,20	100 000	68 719,076
RemoveAttr	1 562,30	100 000	64 008,193

Attribut Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
AddClassName	1 340,90	1 500	1 118,652
RemoveClassName	1 354,55	1 500	1 107,379
HasClassName	3 035,50	3 000	988,305
ReadAttribute	854,25	300 000	351 185,250
WriteAttribute	1 025,60	100 000	97 503,900

Stilar jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
SetCss	996,50	50 000	50 175,615
GetCss	1 012,15	50 000	49 399,793
Height	1 161,85	15 000	12 910,445
Width	1 134,60	15 000	13 220,518
Is	1 852,40	30 000	16 195,206
Show	1 389,20	300 000	215 951,627
Hide	1190,45	50 000	42 000,924
Toggle	765,50	5 000	6 531,679

Stilar Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
SetStyle	1 067,55	50 000	46 836,214
GetStyle	877,55	50 000	56 976,810
GetHeight	1 022,45	50 000	48 902,147
GetWidth	1 048,20	50 000	47 700,820
Is	730,05	350 000	479 419,218
Show	1 066,75	150 000	140 614,015
Hide	748,10	150 000	200 507,953
Toggle	1 016,65	100 000	98 362,268

Event jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Bind	1 313,15	20 000	15 230,552
Trigger	1 046,35	15 000	14 335,547
Unbind	1 418,8	5 000	3 524,105

Event Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Observe	911,9	30 000	32 898,344
Fire	805,45	35 000	43 453,970
StopObserving	1 126,3	20 000	17 757,258

9.1.2 Chrome 11.0.696.68

Traversering jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Parent	1 838,10	400 000	217 616,017
Parents	2 702,95	5	1,850
Prev	1 150,75	298 000	258 961,547
PrevAll	1 009,25	30 000	29 725,043
Next	1 171,70	298 000	254 331,313
NextAll	1 009,85	30 000	29 707,382
Siblings	1 145,20	50 000	43 660,496
Children	1 227,25	300 000	244 448,971

Traversering Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Up	1 325,45	3 000 000	2 263 382,248
Ancestors	1 321,60	30 000	22 699,758
ChildElements	1 903,80	1 490 000	782 645,236
Next	1 593,10	2 980 000	1 870 566,819
Siblings	1 396,80	20 000	14 318,442
PreviousSiblings	1 338,85	20 000	14 938,193
Previous	2 456,85	30 000	12 210,758
NextSiblings	1 373,10	20 000	14 565,582

Manipulation jQuery			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Html	1 940,05	30 000	15 463,519
Before	845,80	42 000	49 657,129
After	1 003,70	42 000	41 845,173
Prepend	1 050,25	42 000	39 990,478
Append	919,25	42 000	45 689,421

Manipulation Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Html	1 074,40	30 000	27 922,561
Before	1 636,50	30 000	18 331,806
After	1 611,25	30 000	18 619,085
Prepend	1 561,75	30 000	19 209,220
Append	1 678,40	30 000	17 874,166

Attribut jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
AddClass	1 357,45	6 000	4 420,052
RemoveClass	1 715,35	6 000	3 497,828
HasClass	1 355,75	5 000	3 687,996
GetAttr	942,35	500 000	530 588,423
SetAttr	814,35	110 000	135 077,055
RemoveAttr	868,05	120 000	138 240,885

Attribut Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
AddClassName	1 189,90	5 000	4 202,034
RemoveClassName	1 125,75	5 000	4 441,483
HasClassName	2 131,20	3 000	1 407,658
ReadAttribute	2 130,70	3 000 000	1 407 987,985
WriteAttribute	1 219,80	300 000	245 941,958

Stilar jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
SetCss	810,75	80 000	98 674,067
GetCss	918,60	80 000	87 089,049
Height	1 348,50	30 000	22 246,941
Width	1 334,90	30 000	22 473,594
Is	1 102,55	50 000	45 349,417
Show	924,95	500 000	540 569,761
Hide	1 143,85	50 000	43 712,025
Toggle	796,20	10 000	12 559,658

Stilar Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
SetCss	1 156,85	150 000	129 662,445
GetCss	1 092,75	100 000	91 512,240
Height	1 211,5	150 000	123 813,454
Width	1 229,85	150000	121 966,093
Is	1 088,55	2 500 000	2 296 633,136
Show	826,9	2 000 000	2 418 672,149
Hide	760,8	300 000	394 321,767
Toggle	2 308,85	1 000 000	433 116,053

Event jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Bind	1 001,85	1 000	998,153
Trigger	807,25	30 000	3 7163,208
Unbind	784,05	2 300	2 933,486

Event Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Observe	871,15	40 000	45 916,318
Fire	932,25	100 000	10 7267,364
StopObserving	835,2	5 000	5 986,590

9.1.3 Safari 5.0.5

Traversering jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Parent	1 905,30	400 000	209 940,692
Parents	3 767,10	5	1,327
Prev	1 233,10	298 000	241 667,342
PrevAll	2 837,05	30 000	10 574,364
Next	1 260,75	298 000	236 367,242
NextAll	2 990,95	30 000	10 030,258
Siblings	4 236,65	50 000	11 801,777
Children	1 498,25	300 000	200 233,606

Traversering Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Up	6 462,25	30 000	4 642,346
Ancestors	764,55	30 000	39 238,768
ChildElements	5 937,25	1 490 000	250 957,935
Next	7 605,95	2 980 000	391 798,526
Siblings	995,70	20 000	20 086,371
PreviousSiblings	749,80	20 000	26 673,780
Previous	1 1197,20	30 000	2 679,241
NextSiblings	755,25	20 000	26 481,298

Manipulation jQuery			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Html	1 058,05	30 000	28 354,048
Before	850,25	50 000	58 806,233
After	839,20	50 000	59 580,553
Prepend	905	50 000	55 248,619
Append	911,85	50 000	54 833,580

Manipulation Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Html	999,65	30 000	30 010,504
Before	854,15	18 000	21 073,582
After	937,20	18 000	19 206,146
Prepend	858,85	18 000	20 958,258
Append	894,55	18 000	20 121,849

Attribut jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
AddClass	1 329,5	6 000	4 512,975
RemoveClass	940,35	7 000	7 444,037
HasClass	1 485,45	5 000	3 365,983
GetAttr	874,35	500 000	571 853,377
SetAttr	1 150,5	200 000	173 837,462
RemoveAttr	902,1	130 000	144 108,192

Attribut Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
AddClassName	833,35	5 000	5 999,880
RemoveClassName	2 937,7	15 000	5 106,035
HasClassName	2 731,25	3 000	1 098,398
ReadAttribute	1 504,7	1 000 000	664 584,303
WriteAttribute	1 250,95	300 000	239 817,739

Stilar jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
SetCss	772,5	80 000	103 559,871
GetCss	841,25	120 000	142 644,874
Height	948,75	30 000	31 620,553
Width	958,4	30000	31 302,170
Is	1 468,05	50 000	34 058,785
Show	1 912,45	500 000	261 444,744
Hide	913,9	50 000	54 710,581
Toggle	861,45	10 000	11 608,335

Stilar Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
SetCss	1 010,55	150 000	148 434,021
GetCss	1 056,1	100 000	94 688,003
Height	1 502,6	150 000	99 826,967
Width	1 589,65	150 000	94 360,394
Is	7 091,05	2 500 000	352 557,097
Show	4 728,6	2 000 000	422 958,169
Hide	1 645,05	300 000	182 365,278
Toggle	8 060,65	1 000 000	124 059,474

Event jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Bind	1477	1 000	677,048
Trigger	1 230	30 000	24 390,244
Unbind	1 025	2 300	2 243,902

Event Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Observe	1 056,10	50 000	47 344,002
Fire	866,05	100 000	115 466,774
StopObserving	853,55	25 000	29 289,438

9.1.4 Opera 11.10

Traversering jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Parent	935,05	40 000	42 778,46
Parents	1 726,55	5	2,90
Prev	1 505,45	59 600	39 589,49
PrevAll	1 237,65	20 000	16 159,66
Next	789,45	44 700	56 621,70
NextAll	1 288,40	20 000	15 523,13
Siblings	965,90	15 000	15 529,56
Children	944,50	40 000	42 350,45

Traversering Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Up	896,05	300 000	334 802,75
Ancestors	770,80	30 000	38 920,60
ChildElements	796,10	149 000	187 162,42
Next	919,50	298 000	324 089,18
Siblings	1 031,20	20 000	19 394,88
PreviousSiblings	774,60	21 000	27 110,77
Previous	904,55	298 000	329 445,58
NextSiblings	766,40	21 000	27 400,84

Manipulation jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Html	850,25	35 000	41 164,36
Before	1 245,85	25 000	20 066,62
After	1 125,35	12 000	10 663,35
Prepend	1 154,55	10 000	8 661,38
Append	1 169	7 000	5 988,02

Manipulation Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Html	974,10	15 000	15 398,83
Before	883,15	6 000	6 793,86
After	1 382,45	5 500	3 978,44
Prepend	1 507,40	5 000	3 316,97
Append	2 193,50	3 000	1 367,68

Attribut jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
AddClass	971	2 500	2 574,67
RemoveClass	2 535,65	5 000	1 971,88
HasClass	1 962,40	5 000	2 547,90
GetAttr	1 359,80	20 000	14 708,05
SetAttr	1 161,35	200 000	172 213,37
RemoveAttr	946,25	130 000	137 384,41

Attribut Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
AddClassName	762,20	2 500	3 279,98
RemoveClassName	1 380,45	4 000	2 897,61
HasClassName	1 227,20	5 000	4 074,32
ReadAttribute	1 104	7 000	6 340,58
WriteAttribute	1 006,50	170 000	168 902,14

Stilar jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
SetCss	1 030,75	150 000	145 525,10
GetCss	1 097,80	120 000	109 309,53
Height	1 445,15	30 000	20 759,09
Width	1 440,90	30 000	20 820,32
Is	919,95	55 000	59 785,86
Show	878,65	500 000	569 054,80
Hide	960,40	100 000	104 123,28
Toggle	931,50	15 000	16 103,06

Stilar Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
SetCss	5 557,95	2 000	359,84
GetCss	1 395,20	100 000	71 674,31
Height	883,60	45 000	50 928,02
Width	890,60	45 000	50 527,73
Is	903,10	550 000	609 013,40
Show	934,70	600 000	641 917,19
Hide	953,70	300 000	314 564,33
Toggle	907,80	200 000	220 312,84

Event jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Bind	1 054,45	150 000	142 254,26
Trigger	863,75	5 000	5 788,71
Unbind	877,3	3 500	3 989,51

Event Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Observe	999,15	12 000	12 010,21
Fire	1 496,95	50 000	33 401,25
StopObserving	828,9	30 000	36 192,54

9.1.5 Internet Explorer 9.0.8112.1421

Traversering jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Parent	993,55	20	20,13
Parents	51 383,75	1	0,02
Prev	924,7	223 500	241 700,01
PrevAll	938,1	2 500	2 664,96
Next	910,05	223 500	245 590,90
NextAll	933,2	2 500	2 678,95
Siblings	761,9	2 500	3 281,27
Children	1 067,3	250 000	234 235,92

Traversering Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Up	1 144,15	75000	65550,85
Ancestors	1 103	2000	1813,24
ChildElements	1 354,90	37250	27492,80
Next	1 385,40	74500	53775,08
Siblings	1 304,05	1500	1150,26
PreviousSiblings	1 273,50	1500	1177,86
Previous	1386	74500	53751,80
NextSiblings	1 263,75	1500	1186,94

Manipulation jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Html	833,35	6 000	7 199,86
Before	848,40	7 500	8 840,17
After	848,40	7 500	8 840,17
Prepend	956,45	7 500	7 841,50
Append	863,90	7 500	8 681,56

Manipulation Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Html	968,65	6 000	6 194,19
Before	1 428,75	5 000	3 499,56
After	1 562,70	4 500	2 879,63
Prepend	1 681,40	4 000	2 378,97
Append	1 687,50	3 500	2 074,07

Attribut jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
AddClass	810,25	2 000	2 468,37
RemoveClass	966,30	2 000	2 069,75
HasClass	1 030,80	3 000	2 910,36
GetAttr	1 382,30	7 000	5 064,02
SetAttr	925,65	40 000	43 212,88
RemoveAttr	888,55	35 000	39 390,02

Attribut Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
AddClassName	1 146,75	1 800	1 569,65
RemoveClassName	1 235,80	800	647,35
HasClassName	1 075,80	600	557,72
ReadAttribute	988,90	5 000	5 056,12
WriteAttribute	1 018,30	35 000	34 371,01

Stilar jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
SetCss	796	15 000	18 844,22
GetCss	1 003,10	70 000	69 783,67
Height	935,70	15 000	16 030,78
Width	933,70	15 000	16 065,12
Is	1 484,30	50 000	33 685,91
Show	963,40	100 000	103 799,05
Hide	1 306,10	100 000	76 563,82
Toggle	1 442	7 000	4 854,37

Stilar Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
SetCss	889	30 000	33 745,78
GetCss	934,80	40 000	42 789,90
Height	5 476,90	10 000	1 825,85
Width	14 129,40	7 000	495,42
Is	1 041,70	75 000	71 997,70
Show	930,50	65 000	69 854,92
Hide	947,70	65 000	68 587,11
Toggle	1 076,30	25 000	23 227,72

Event jQuery			
Funktion	**Medelvärde(ms)**	**Antal iterationer**	**Körningar per sekund**
Bind	1 053,15	1 000	949,53
Trigger	742,95	7 000	9 421,90
Unbind	1 149,50	15 000	13 049,15

Event Prototype			
Funktion	**Medelvärde(ms)**	**Antal Iterationer**	**Körningar per sekund**
Observe	1 001,50	20 000	19 970,04
Fire	977,20	7 000	7 163,32
StopObserving	1 307,85	12 000	9 175,36

9.2 Beräkningar

Tabell för att jämföra biblioteken på körningar per sekund totalt sett över alla webbläsare.

Totalt medel över alla test	**jQuery**	**Prototype**
Traversering	105 615,143	211217,905
Manipulation	28 320,243	12540,691
Attribut	81 486,670	130407,498
Stilar	83 377,955	273002,206
Event	18 463,287	37552,853

Tabell för att jämföra de båda biblioteken i alla webbläsare baserat på ett medelvärde av alla testfalls medelvärde.

Biblioteksjämförelse över alla webbläsare	**Firefox**	**Chrome**	**Safari**	**Opera**	**IE**
jQuery	59 174,50	86 407,4	82 053,56	56 494,83	33 132,99
Prototype	84 340,54	356 374,5	110 970,9	95 284,9	17 750,29

Tabell för att jämföra de båda biblioteken i testfallet traversering.

Medelvärde för alla funktioner, traversering					
	Firefox	**Chrome**	**Safari**	**Opera**	**IE**
jQuery	158 351,123	134 806,577	115 077,076	28 569,418	91 271,521
Prototype	149 575,632	624 415,879	95 319,783	161 040,876	25 737,354

Tabell för att jämföra de båda biblioteken i testfallet manipulation.

Medelvärde för alla funktioner, manipulation					
	Firefox	**Chrome**	**Safari**	**Opera**	**IE**
jQuery	26 118,065	38 529,144	51 364,607	17 308,748	8 280,651
Prototype	10 461,576	20 391,368	22 274,068	6 171,157	3 405,285

Tabell för att jämföra de båda biblioteken i testfallet attribut.

Medelvärde för alla funktioner, attribut					
	Firefox	**Chrome**	**Safari**	**Opera**	**IE**
jQuery	49 575,026	135 918,706	150 853,671	55 233,379	15 852,567
Prototype	90 380,697	332 796,224	183 321,271	37 098,923	8 440,373

Tabell för att jämföra de båda biblioteken i testfallet stilar.

Medelvärde för alla funktioner, stilar					
	Firefox	**Chrome**	**Safari**	**Opera**	**IE**
jQuery	50 798,226	109 084,314	83 868,739	130 685,130	42 453,366
Prototype	139 914,931	751 212,167	189 906,175	244 912,210	39 065,550

Tabell för att jämföra de båda biblioteken i testfallet eventrespons.

Medelvärde för alla funktioner, eventrespons					
	Firefox	**Chrome**	**Safari**	**Opera**	**IE**
jQuery	11 030,068	13 698,283	9 103,731	50 677,494	7 806,861
Prototype	31 369,857	53 056,757	64 033,405	27 201,334	12 102,911

9.3 Testfallskod

Här bifogas ett utdrag av testfallskoden för att möjliggöra en eventuell återskapning av testfallen. I 9.3.1 ser vi hur vår index.html fil ser ut. Den är liknande för de båda funktionsbiblioteken och skiljer sig endast på vilket bibliotek och vilka testfall som inkluderas. I 9.3.2 visas hur testfallet för traversering i jQuery ser ut. Övriga testfall följer samma struktur.

9.3.1 jQuery Index

```
<html>
    <head>
                    <meta http-equiv="content
typecontent="text/html;charset=utf-8" />
                    <title>Test Cases</title>

       <!--link to library -->
       <script type="text/javascript" src="jquery.js"></script>

       <!--links to testcases -->
       <script type="text/javascript"
src="jQueryTraversal.js"></script>
       <script type="text/javascript"
src="jQueryManipulation.js"></script>
       <script type="text/javascript" src="jQueryEvent.js"></script>
       <script type="text/javascript"
src="jQueryAttribute.js"></script>
       <script type="text/javascript" src="jQueryStyle.js"></script>

       <script type="text/javascript"
src="emptyIterators.js"></script>

       </head>
     <body>
       <!--Root tag, all tests will run within this div -->
       <div id="root"></div>

       <script type="text/javascript">

         //Defining start and stop functions for the timers
         var timeDiff = {
             setStartTime:function(){
               d = new Date();
               time = d.getTime();
             },
             getDiff:function (){
               d = new Date();
               return (d.getTime()-time);
             }
         }

         //Creating a new testcase
         var testcase = new emptyIterators();

         //Run the testcase
         var result = testcase.runTestCase(20);
```

```
          //Print results
          for(l=0;l<result[0].length;l++) {
            document.write(result[0][l][0]+" : "+result[0][l][1]+" ");
            document.write("<br/>");

            for(o=0;o<result[2][l].length;o++) {
              document.write(result[2][l][o]+" ");
            }

            document.write("<br/><br/>");
          }
        </script>

      </body>
    </html>
```

9.3.2 jQuery Traversal

```
    /*
    * Testcase jQueryTraversal
    * ------------------------------
    * -library jQuery
    * -libraryversion 1.5.2
    * -testcaseversion 1.0
    * -authors Petter Andersson, Eric Ericsson
    * -date 2011-05-10
    *
    * -description
    * This testcase tests jQuerys abillity to traverse DOM trees.
    *
    * Using functions: Parent, Parents, Prev, PrevAll, Next, NextAll,
  Siblings, Children
    *
    */
    function jQueryTraversal() {
    /*
    * ---------------------------------------------------------------------
    * - Test framework parameters
    * ---------------------------------------------------------------------
    */
      this.tests;
      this.timers;
      this.result;

    /*
    * ---------------------------------------------------------------------
    * - Define local test parameters here
    * ---------------------------------------------------------------------
    */
      var totalnrOfElem;
      var nrOfElem;
      var root;
      var testParentRoot;
      var testPrevRoot;
      var testNextRoot;
      var testSiblingsRoot;
      var testChildrenRoot;
    /*
    * ---------------------------------------------------------------------
    * - Test case main
    * ---------------------------------------------------------------------
    */
      this.runTestCase = function (numberOfRuns) {

        this.setup();
        for(i=0;i<this.tests.length;i++) {
          for(j=0;j<numberOfRuns;j++) {

               this.build(i);
```

```
                timeDiff.setStartTime();
                eval("this.test"+this.tests[i]+"()");
                this.timers[i][j] = timeDiff.getDiff();

                this.clean(i);
            }
          }

          this.tearDown();

          return this.calculateResult();
        }
      /*
      * -------------------------------------------------------------------
      * - Setup and tearDown
      * -------------------------------------------------------------------
      */
        this.setup = function() {

            this.tests = new
Array("Parent","Parents","Prev","PrevAll","Next","NextAll","Siblings","
Children");
            this.timers = new Array();
            this.result = new Array();

            for(i=0;i<this.tests.length;i++) {
                this.timers[i] = new Array();
                this.result[i] = new Array();
            }

            totalnrOfElem = 0;
            nrOfElem = 150;
            root = $('div');
            root.attr('id',"root");

            var node1;
            var node2;
            var node3;
            var node4;

            for(i=0; i<nrOfElem; i++) {
                      node1 = $('<div>');
                      node1.attr('class','ch1-'+i);
                      for(j=0; j<nrOfElem-20; j++) {
                                  node2 = $('<div>');
                                  node2.attr('class','ch2-'+j);
                                  node1.append(node2);
                                  totalnrOfElem++;
                      }
                      root.append(node1);
                      totalnrOfElem++;
            }

            node2 = $('.ch2-8');
            for(k=0; k<100; k++) {
                      node3 = $("<div>");
                      node3.attr('class','ch3-'+k);
                      node2.append(node3);
                      totalnrOfElem++;
            }

            node3 = $('.ch3-6');
            for(k=0; k<100; k++) {
                      node4 = $("<div>");
                      node4.attr('class','ch4-'+k);
                      node3.append(node4);
                      totalnrOfElem++;
```

```
        }

        var nodes;
        var tmp;

        node4 = $('.ch4-3');
        tmp = $(node4);

        for(i=0; i<100; i++){
                  nodes = $("<div>");
                  nodes.attr('class','ch'+(i+5)+'-'+0);
                  tmp.append($(nodes));
                  tmp = $('.ch'+(i+5)+'-'+0);
                  totalnrOfElem++;
        }

        testParentRoot = $('.ch104-0');
        testPrevRoot = $('.ch1-149');
        testNextRoot = $('.ch1-0');
        testSiblingsRoot = $('.ch1-50');
        testChildrenRoot = $('.ch4-3');
    }

    this.tearDown = function() {
        var node = document.body;
        if (node.hasChildNodes()) {
            while (node.childNodes.length >= 1) {
                node.removeChild(node.firstChild);
            }
        }

    }

    this.build = function(testId) {

    }

    this.clean = function(testId) {

    }
  /*
  * ---------------------------------------------------------------------
  * - Result calculation
  * ---------------------------------------------------------------------
  */
    this.calculateResult = function() {

         var testcaseResult = new Array();
         var totalTimer = 0;

         for(i=0;i<this.tests.length;i++) {

          totalTimer = 0;
          for(j=0;j<this.timers[i].length;j++) {
            totalTimer = totalTimer + this.timers[i][j];
          }

          this.result[i][0] = this.tests[i];
          this.result[i][1] = totalTimer / this.timers[i].length;
         }

         testcaseResult[0] = this.result;
         testcaseResult[1] = this.tests;
         testcaseResult[2] = this.timers;

         return testcaseResult;
    }
```

```
/*
* ---------------------------------------------------------------------
* - Testfunctions
* ---------------------------------------------------------------------
*/
  this.testParent = function() {
    var node1 = testParentRoot;
      for(z=0;z<20;z++) {
                              node1 = node1.parent();
                }
  }

  this.testParents = function() {
    var leaf = testParentRoot;
     var node2 = leaf.parents();
  }

  this.testPrev = function() {
    var node3 = testPrevRoot;
    for(yy=0;yy<1500;yy++) {
                for(zz=0;zz<149;zz++){
                              node3 = node3.prev();
                }
    }
  }

  this.testPrevAll = function() {
    var root1 = testPrevRoot;

    for(xxx=0;xxx<2500;xxx++) {
                var node4 = root1.prevAll();
    }
  }

  this.testNext = function() {
    var node5 = testNextRoot;
    for(yyy=0; yyy<1500; yyy++) {
      for(zzz=0; zzz<149; zzz++){
                              node5 = node5.next();
                }
    }
  }

  this.testNextAll = function() {
    var root2 = testNextRoot;

    for(xx=0;xx<2500;xx++) {
                var node6 = root2.nextAll();
     }
  }

  this.testSiblings= function() {
    var root3 = testSiblingsRoot;

    for(x=0;x<2500;x++) {
     var node7 = root3.siblings();
    }
  }

  this.testChildren = function() {
    var node8 = testChildrenRoot;

    for(yyyy=0;yyyy<2500;yyyy++){
                for(zzzz=0;zzzz<100;zzzz++){
                              node8 = node8.children();
                }
    }
  }}
```