HÖGSKOLAN DALARNA

Examensarbete

Kandidatnivå

Ensidesapplikationers kodkvalitet och förvaltningsbarhet

En jämförelse av förvaltningsbarhet hos single-page applications utvecklade med AngularJS och React

Code Quality and Maintainability of Single-page Applications – A Comparison of AngularJS and React

Författare: Tom Niskanen, Johan Nyström
Handledare: Pär Eriksson, Hans Jernberg
Examinator:
Ämne/huvudområde: Informatik
Kurskod: IK2017
Poäng: 15hp
Examinationsdatum: 2016-05-26

Abstract: Single-page applications have historically been subject to strong market forces driving fast development and deployment in lieu of quality control and changeable code, which are important factors for maintainability. In this report we develop two functionally equivalent applications using AngularJS and React and compare their maintainability as defined by ISO/IEC 9126. AngularJS and React represent two distinct approaches to web development, with AngularJS being a general framework providing rich base functionality and React a small specialized library for efficient view rendering. The quality comparison was accomplished by calculating Maintainability Index for each application. Version control analysis was used to determine quality indicators during development and subsequent maintenance where new functionality was added in two steps.

The results show no major differences in maintainability in the initial applications. As more functionality is added the Maintainability Index decreases faster in the AngularJS application, indicating a steeper increase in complexity compared to the React application. Source code analysis reveals that changes in data flow requires significantly larger modifications of the AngularJS application due to its inherent architecture for data flow. We conclude that frameworks are useful when they facilitate development of known requirements but less so when applications and systems grow in size.

Sammanfattning: Ensidesapplikationer har historiskt sett påverkats av starka marknadskrafter som pådriver snabba utvecklingscykler och leveranser. Detta medför att kvalitetskontroll och förändringsbar kod, som är viktiga faktorer för förvaltningsbarhet, blir lidande. I denna rapport utvecklar vi två funktionellt ekvivalenta ensidesapplikationer med AngularJS och React samt jämför dessa applikationers förvaltningsbarhet enligt ISO/IEC 9126. AngularJS och React representerar två distinkta angreppsätt på webbutveckling, där AngularJS är ett ramverk med mycket färdig funktionalitet och React ett mindre bibliotek specialiserat på vyrendering. Kvalitetsjämförelsen utfördes genom att beräkna förvaltningsbarhetsindex för respektive applikation. Versionshanteringsanalys användes för att bestämma andra kvalitetsindikatorer efter den initiala utvecklingen samt två efterföljande underhållsarbeten.

Resultaten visar inga markanta skillnader i förvaltningsbarhet för de initiala applikationerna. I takt med att mer funktionalitet lades till sjönk förvaltningsbarhetsindex snabbare för AngularJS-applikationen, vilket motsvarar en kraftigare ökning i komplexitet jämfört med React-applikationen. Versionshanteringsanalys visar att ändringar i dataflödet kräver större modifikationer för AngularJS-applikationen på grund av dess förbestämda arkitektur. Utifrån detta drar vi slutsatsen att ramverk är användbara när de understödjer utvecklingen mot kända krav men att deras nytta blir begränsad ju mer en applikation växer i storlek.

Nyckelord: Single-page applications, software quality, Maintainability Index

Innehållsförteckning

Figurförteckning

Begreppslista

AngularJS: Ett ramverk framtaget av Google för att skapa ensidesapplikationer.

DOM, dokumentobjektmodellen *(eng. document object model):* Ett gränssnitt för att läsa och manipulera innehåll i dokument skrivna med exempelvis HTML och XML.

Ensidesapplikation *(eng. single-page application):* En typ av rik webbapplikation. Någon bestämd definition saknas, men gemensamt för ensidesapplikationer är att de laddas genom ett initialt anrop mot en webbserver, varefter ny HTML genereras direkt i webbläsaren.

Flux: Ett designmönster för att strukturera applikationer med ett enkelriktat data-flöde tillsammans med React.

Förvaltningsbarhet: En del av kvalitetsbegreppet i ISO/IEC 9126 som avser ett systems förmåga till analyserbarhet, förändringsbarhet, stabilitet och testbarhet.

Insticksprogram: En programvara som installeras i ett annat program för att tillföra ny funktionalitet.

Kodkvalitet: Avser ett systems inre kvaliteter och förmåga att kodmässigt uppfylla de krav en användare eller verksamhet ställer.

React: Ett bibliotek framtaget av Facebook och Instagram för att effektivt uppdatera dokumentobjektmodellen. Används ofta för att skapa ensidesapplikationer.

SOLID: En samling principer för gott mjukvaruhantverk.

1 Inledning

I detta kapitel redogör vi för arbetets forskningsfråga och syfte. En bakgrund till kvalitetsbegreppet inom systemutveckling ges tillsammans med en kort redogörelse för hur ensidesapplikationer utvecklas.

1.1 Bakgrund

Mjukvarukvalitet är ett begrepp med skilda innebörder för olika aktörer. För att kunna mäta och följa upp mjukvarukvalitet krävs en gemensam definition av begreppet, varför många modeller och standarder har tagits fram. En av dessa är ISO/IEC 9126, som är en internationellt vedertagen standard för mjukvarukvalitet som ofta används inom forskningssammanhang (Kanellopoulos, o.a., 2010, s. 18). Den angriper kvalitetsbegreppet från tre perspektiv: extern kvalitet, intern kvalitet samt kvalitet vid användning. Extern kvalitet avser den färdiga produktens förmåga att erbjuda värde i sin avsedda miljö och kvalitet vid användning avser slutanvändarens nöjdhet med produkten. Den interna kvaliteten avser källkodens egenskaper, vilket även går att mäta under produktens utveckling. Intern kvalitet delas in i sex huvudkategorier: funktionalitet, pålitlighet, användbarhet, effektivitet, förvaltningsbarhet och portabilitet (ISO/IEC, 2004).

Förvaltningsbarhet som en intern kvalitetsegenskap avser i sammanhanget källkodens förmåga till analyserbarhet, stabilitet, testbarhet och förändringsbarhet (ISO/IEC, 2004), vilka länge varit föremål för forskning inom systemutveckling. Denna forskning saknas dock för webbapplikationer. Förvaltningsbarhet är ett av de viktigaste kvalitetsmåtten för mjukvara då förvaltning utgör den absolut största delen av ett systems livscykel (Glass, 2002). Av förvaltningskostnaden utgörs den största delen av att förbättra och utöka funktionalitet, medan knappt en femtedel av kostnaden består av att åtgärda defekter i koden (Glass, 2002).

En systematisk genomgång av forskningslitteraturen om webbapplikationer från 2014 visar att kvalitet och mått för kvalitet är ett av de minst beforskade områdena (Casteleyn, Garrig'os, & Maz'on, 2014, s. 15). Webbapplikationer utsätts för samma kvalitetskrav och risker som vilken annan typ av applikation som helst men unikt för webben är dess snabba utvecklingstakt, vilket i kombination med en marknad där snabba leveranser prioriteras medför att applikationernas kodmässiga kvalitet hamnar i skymundan (Maurya & Shankar, 2012, s. 33).

Detta, tillsammans med ökade krav på bättre funktionalitet och användbarhet, medför att nya tekniker för att utveckla webapplikationer snabbt tas i bruk. Den senaste trenden är att överge insticksprogram för att köra avancerad programkod i webbläsaren (t.ex. Adobe Flash eller Microsoft Silverlight) till förmån för applikationer skrivna i ren JavaScript, s.k. ”ensidesapplikationer” (eng. *single-page applications*) (Mikowski & Powell, 2014). För att effektivisera utvecklingen av sådana applikationer används olika ramverk och bibliotek som förser en utvecklare med färdig funktionalitet att utgå från.

Skillnaden mellan ramverk och bibliotek är att ramverk tenderar att tillhandahålla mycket färdig funktionalitet. Bibliotek löser oftast endast ett visst problem men är lättare att kombinera med andra bibliotek och egen programkod (Mardan, 2016). De

tekniker som finns för att skapa ensidesapplikationer använder något av dessa angreppssätt. Oavsett om man använder ett stort ramverk eller flera mindre bibliotek är det kostsamt och tidskrävande att byta teknik i efterhand. I förlängningen har valet av ramverk eller bibliotek stor inverkan på en applikation under hela dess livstid (Martin, 2014), men det är oklart hur det påverkar inre kvaliteter så som förvaltningsbarhet och förändringsbarhet.

I dagsläget är det mest använda ramverket AngularJS, som har skapats och underhålls av Google. Det bibliotek som på senare tid har rönt stora framgångar är React, som är ett resultat av ett samarbete mellan Facebook och Instagram.

1.2 Problemformulering

Det förekommer ofta stora skillnader i hur befintliga ramverk och bibliotek löser olika problem, vilket leder till att den kod som skrivs tenderar att bli starkt bunden till det valda ramverket eller biblioteket (Martin, 2014). Det är också svårt att på förhand avgöra vilka konsekvenser valet av teknik kommer få för en webbapplikations förvaltningsbarhet.

Detta mynnar ut i följande frågeställning:

- Hur påverkas en ensidesapplikations förvaltningsbarhet om den utvecklas med ett ramverk jämfört med bibliotek?

1.3 Syfte

Att jämföra kodkvalitet och förvaltningsbarhet hos ensidesapplikationer utvecklade med AngularJS och React.

1.4 Mål

Målet är att ta fram en jämförelse som klargör hur applikationer utvecklade med AngularJS och React förhåller sig till förvaltningsbarhet, vilket även ger en indikation om generella kvalitetsskillnader för webbutveckling med ramverk eller bibliotek. Detta är till nytta för alla som planerar inför större webbprojekt, då kodkvalitet i allmänhet och förvaltningsbarhet i synnerhet är faktorer som behöver tas i beaktande inför beslut om teknikval.

1.5 Avgränsningar

Vi avgränsar oss till att undersöka förvaltningsbarhet som en del av intern kvalitet enligt ISO/IEC 9126, med särskilt fokus på förändringsbarhet. Detta eftersom arbetet med att förbättra och utöka funktionalitet upptar en stor majoritet av den tid och de resurser som läggs på en applikation under dess livscykel (Glass, 2002). En nyare standard togs fram 2011 (ISO/IEC 25010). Den är dock inte lika väl beforskad och medför inga skillnader för vår forskning då förvaltningsbarhet fortfarande ingår.

1.6 Samarbetspartner

Vår samarbetspartner är Triona AB med huvudkontor i Borlänge. Triona har ca 130 medarbetare utspridda på olika platser i Sverige och Norge. De har stor teknisk expertis inom logistik och GIS samt lång erfarenhet av systemförvaltning i många

olika projekt och har bidragit med handledning och inspiration för arbetets utformning.

2 Kodkvalitet, mätvärden och ensidesapplikationer

I det här kapitlet definieras kvalitet (2.1) och beskrivs den standard för kodkvalitet (ISO/IEC 9126) vi valt att använda som teoretisk referensram. Därefter beskrivs förvaltningsbarhetsindex som en modell för att mäta förvaltningsbarhet och dess enskilda komponenter (2.3.1). Slutligen definierar vi begreppet ”ensidesapplikation” (2.4) och beskriver AngularJS och React.

2.1 Kvalitet

David Garvin (1984, ss. 25-28) beskriver kvalitet som ett begrepp med olika innebörder för olika aktörer. Användare, tillverkare och leverantörer har alla olika perspektiv på kvalitet. Kitchenham & Pleeger (1996, s. 13) argumenterar för att samtliga av Garvins perspektiv på produktkvalitet är relevanta inom systemutveckling. Användarperspektivet beskriver de som den grad i vilken mjukvaran uppfyller slutanvändarens behov. För att kontinuerligt kunna uppfylla dessa behov förutsätts dock att ett system har vissa inre kvaliteter, som användaren inte kommer i direkt kontakt med. Mjukvarukvalitet kan därför kategoriseras som extern (användarorienterad) eller intern (kodmässig) kvalitet (McConnel, 2004, ss. 263, 264). Den främsta inre kvaliteten är systemets förvaltningsbarhet, d.v.s. att det ska kunna anpassas och förändras i takt med att behoven och kraven förändras (Granja-Alvarez & Barranco-Garcia, 1997, s. 162). För att det skall vara möjligt måste programkoden vara strukturerad på ett sätt som gör den lätt att testa, enkel att förstå samt möjlig att modifiera.

2.2 Kvalitetsmodeller

2.2.1 Behov av modeller

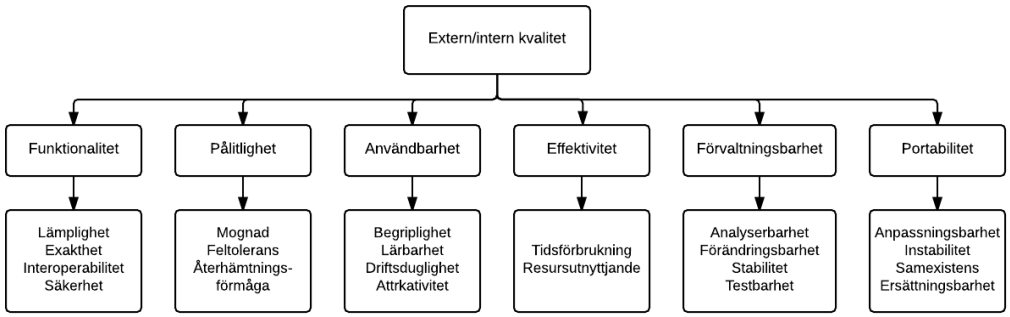
Genom åren har flera standarder för kvalitetsmodeller tagits fram. En kvalitetsmodell ger en teoretisk referensram och en standardiserad terminologi, vilket underlättar kommunikation och målstyrning mot kvalitet (Heitlager, Kupiers, & Visser, 2007).

Deissenboeck, Juergens, Lochmann, & Stefan (2009) kategoriserar kvalitetsmodeller efter vilka syften de fyller. Som grund för en definition av kvalitet behövs en *beskrivande* modell som klassificerar kvalitetsbegreppets olika delar och standardiserar innebörden av kvalitet inom ett projekt eller en organisation. För att sedan mäta och följa upp kvalitet kan en *utvärderande* modell användas. När en beskrivande och utvärderande modell är på plats möjliggörs även implementeringen av en *prediktiv* modell som kan användas för att förutsäga kvalitet.

En beskrivande modell är dels användbar för att standardisera kommunikation kring kvalitet, men även för att lägga grunden för kodstandarder och kvalitetskriterier i ett projekt. En utvärderande modell ligger till grund för hur själva mätningen av kvalitet kan utföras objektivt. En prediktiv modell kan underlätta projektstyrning genom att visa hur och när projekt kan anses bli klara och driftsättas (Deissenboeck, Juergens, Lochmann, & Stefan, 2009). Ett exempel på en beskrivande modell är ISO/IEC 9126 som kan användas tillsammans med förvaltningsbarhetsindex (kap. 2.3.1) som en utvärderande modell.

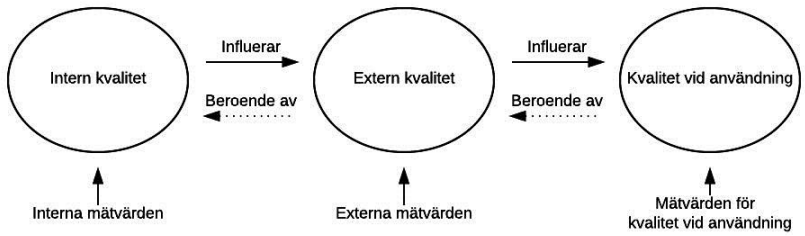
2.2.2 ISO/IEC 9126

Den mest använda kvalitetsmodellen idag är ISO/IEC 9126 (Kanellopoulos, o.a., 2010, s. 18) som är en beskrivande kvalitetsmodell vilken, likt Garvin (1984), delar in kvalitet i olika perspektiv. De huvudsakliga perspektiven på kvalitet enligt ISO/IEC 9126 är intern kvalitet, extern kvalitet samt kvalitet vid användning (ISO/IEC, 2004). Intern kvalitet avser kvaliteten på själva källkoden medan extern kvalitet mäts genom att testa den färdiga produkten i dess avsedda miljö (ISO/IEC, 2004). Kvalitet vid användning avser användarens nöjdhet med systemet. Dessa tre kvalitetsperspektiv delar samma sex huvudkategorier: *Funktionalitet*, *pålitlighet*, *användbarhet*, *effektivitet*, *förvaltningsbarhet* och *portabilitet.* vilka har ett ytterligare antal attribut som karaktäriserar dem (se figur 1) (ISO/IEC, 2004).



Figur 1. Intern/extern kvalitet i ISO/IEC 9126

Utmärkande för intern kvalitet är att det går att mäta på ett ofullständigt system under utveckling. Med hjälp av mätvärden kan en utvecklare vidta åtgärder för att förbättra den interna kvaliteten tidigt i utvecklingsprocessen vilket får positiva följder för det färdiga systemet. Detta då den interna kvaliteten influerar den externa kvaliteten, samt kvaliteten vid användning (ISO/IEC, 2004). Sambanden mellan de olika perspektiven illustreras i figur 2.



Figur 2. Intern kvalitet relaterat till extern kvalitet (ISO/IEC, 2004)

2.3 Förvaltningsbarhet

Som nämnt i kapitel 2.1 delar ISO/IEC 9126 in intern kvalitet i sex huvudkategorier. Av dessa är den enskilt viktigaste kategorin förvaltningsbarhet. Detta eftersom förmågan att kunna förändra mjukvara är direkt kopplad till hur väl systemet uppfyller slutanvändarens behov, vilket ökar systemets affärsvärde (Bakota, Hegedus, Körtvélyesi, Ferenc, & Gyimóthy, 2011). Flera försök har gjorts att försöka kvantifiera förändringsbarhet för att möjliggöra mätning och uppföljning. De två tidigaste måtten är cyklomatisk komplexitet (McCabe, 1976) och Halsteads mått. För att ge en mer balanserad bild av förvaltningsbarhet är det vanligt att dessa aggregerats i en utvärderande kvalitetsmodell kallad förvaltningsbarhetsindex.

2.3.1 Förvaltningsbarhetsindex

Förvaltningsbarhetsindex (eng. *Maintainability Index*) är ett sammansatt mått först föreslaget av Oman och Hagemeister i början av 1990-talet som beräknas på ovan nämnda mätvärden för genomsnittlig cyklomatisk komplexitet (CC) och medelvärdet av Halsteads ansträngning (HE) tillsammans med antal rader kod (LOC). Beroende på vilken variant som används kan även förekomsten av kommentarer (CM) inkluderas enligt följande funktioner (Welker, 2001, s. 18):

Utan kommentarer: $171 - 3{,}42\,ln(HE) - 0{,}23CC - 16{,}2\,ln(LOC)$

Med kommentarer: $171 - 3{,}42\ln(HE) - 0{,}23CC - 16{,}2\ln(LOC) + 0{,}99CM$

Anpassningsfunktionerna är framtagna genom statistisk korrelationsanalys av källkod från ett stort antal olika system och är kalibrerad i samråd med de tekniska experter som arbetat med underhåll av systemen i fråga (Heitlager, Kupiers, & Visser, 2007). I allmänhet gäller att ju högre tal för förvaltningsbarhetsindex, desto lättare är systemet att underhålla. Som går att utläsa av funktionerna är 171 det högsta möjliga erhållbara värdet, åtminstone för versionen utan kommentarer. Vedertagna gränsvärden för icke-objektorienterade system är att värden över 85 indikerar god förvaltningsbarhet, 65-85 indikerar måttlig förvaltningsbarhet och allt under 65 indikerar undermålig förvaltningsbarhet (Sjøberg, Anda, & Mockus, 2012, s. 108). Motsvarande intervall för objektorienterade system saknas men forskare inom området argumenterar för att gränsvärdena för dessa bör vara högre, då klasser inom objektorienterade språk är mindre än moduler i andra typer av språk (Sjøberg, Anda, & Mockus, 2012, s. 108).

Sedan framtagandet av den ursprungliga formeln har vidare forskning lagt större vikt vid Halsteads volym snarare än ansträngning vid beräkning av förvaltningsbarhetsindex. Hur stor inverkan förekomsten av kommentarer bör ha är också föremål för diskussion, vilket har lett fram till att andra varianter av anpassningsfunktioner för förvaltningsbarhetsindex har tagits fram. (Welker, 2001, s. 18)

En nackdel med ett sammansatt mått som förvaltningsbarhetsindex är att det inte möjliggör någon grundorsaksanalys, d.v.s. visar vad i koden som orsakar ett lågt värde, eller kan indikera lämpliga förbättringsåtgärder. Dessutom belyser inte anpassningsfunktionen några kausala samband mellan brister i källkoden och dess förvaltningsbarhet, eftersom den är framtagen genom korrelationsanalys. Detta är något som har visat sig negativt påverka acceptansen hos utvecklare och företagsledare (Heitlager, Kupiers, & Visser, 2007). Vidare är förvaltningsbarhetsindex inte något intuitivt mått, vilket ytterligare inverkar på dess upplevda nytta. Faktorerna och parametrarna som ingår i anpassningsfunktionen kan upplevas som godtyckliga – varför multipliceras den cyklomatiska komplexiteten med just 0,23 och hur kommer det sig att Halsteads ansträngning och antal rader kod ingår som logaritmiska parametrar? Även fast anpassningsfunktionen är framtagen med gedigna statistiska metoder möts den av dessa anledningar med visst motstånd inom systemutvecklingsbranschen. (Heitlager, Kupiers, & Visser, 2007)

Trots sina brister har förvaltningsbarhetsindex brett vetenskapligt stöd för sin användbarhet inom utvecklingsprojekt och förvaltningsorganisationer (Welker,

2001, ss. 18-19). Som med alla mätningar behöver det dock användas med medvetenhet och försiktighet. Om kommentarsfrekvensen ska inkluderas i beräkningen av förvaltningsbarhetsindex behöver en separat kvalitetskontroll av kommentarerna genomföras för att resultatet ska bli rättvisande, eftersom utdaterade eller missvisande kommentarer snarare stjälper än hjälper vid underhållsarbete.

2.3.2 Cyklomatisk komplexitet

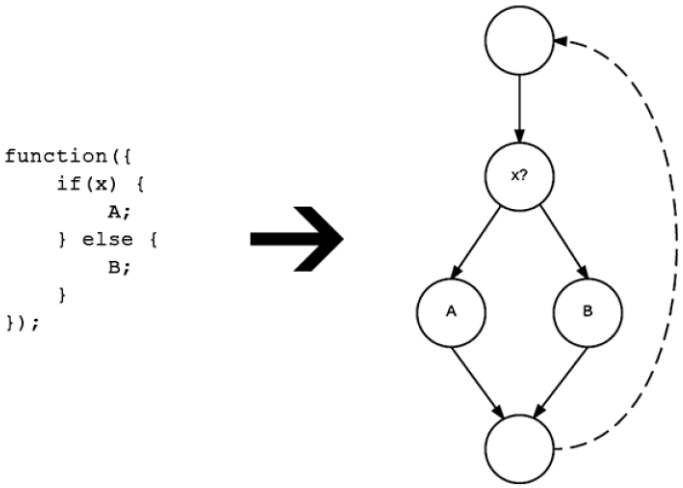
Thomas McCabe (1976) föreslog ett sätt att utvärdera ett programs komplexitet utifrån dess kontrollflödesgraf, vilket är ett sätt att visualisera ett programs exekveringsflöde. Genom att applicera teorem från grafteori på kontrollflödesgrafen åstadkom han en metod för att matematiskt beräkna antalet unika linjära exekveringsvägar genom ett program (McCabe, 1976, s. 308).

McCabe (1976 s. 315) utgick från Eulers formel, som säger att det cyklomatiska numret *V(G)* för en graf *G* med *n* hörn, *e* kanter är $v(G) = e - n + 2$. Eulers formel är applicerbar under förutsättning att grafen är av sådan art att den

1. är riktad
2. har en start- och en slutnod
3. varje nod är sammankopplad på ett sådant vis att den är nåbar från startnoden samt själv kan nå slutnoden

(McCabe, 1976)

Följande enkla kodexempel (figur 3) ger upphov till nedanstående kontrollflödesgraf:



Figur 3. Kontrollflödesgraf

När den cyklomatiska komplexiteten beräknas antas att varje nod i grafen motsvarar ett kodblock med ett sekventiellt flöde och att varje förgrening i programmets exekveringsflöde ger upphov till en ny kant i grafen.

Den cyklomatiska komplexiteten för exemplet ovan blir $v(G) = 5-5+2$, d.v.s. 2. McCabe (1976 s. 314) föreslog själv ett värde på 10 som en lämplig övre gräns för hanterbar komplexitet. Senare forskning har dock ifrågasatt nyttan med att mäta en genomsnittlig cyklomatisk komplexitet, främst då objektorienterade språk tenderar att ha en komplexitetsnivå per modul som följer en potenslagsdistribution

(Heitlager, Kupiers, & Visser, 2007), vilket innebär att den genomsnittliga komplexiteten i systemet alltid blir låg. Enkla metoder för att skriva eller läsa värden (getters och setters) i exempelvis Java förekommer ofta och har låg komplexitet, vilket drar ner genomsnittsmåttet i en applikation. Istället förespråkas lokala mätningar som klassificerar komplexiteten på enhetsnivå snarare än modul- eller systemnivå (Heitlager, Kupiers, & Visser, 2007).

Källkodens komplexitet är kopplad till hur förändringsbar och testbar den är. Ju mer komplex kod, desto svårare är den för en utvecklare att förstå och förändra (Heitlager, Kupiers, & Visser, 2007). Likaså är det besvärligare att skriva enhetstester för komplex kod, då den cyklomatiska komplexiteten direkt motsvarar antalet testvägar som krävs (McCabe & Watson, 1996, ss. 3-4).

2.3.3 Halsteads mått

Halsteads mått mäter en applikations egenskaper utifrån antalet operatorer och operander i källkoden. De fundamentala mätvärden som definieras är (Shen, Conte, & Dunsmore, 1983, s. 155):

η_1 = antal unika operatorer
η_2 = antal unika operander
N_1 = total förekomst av operatorer
N_2 = total förekomst av operander

Utifrån detta definieras ett programs vokabulär $\eta = \eta_1 + \eta_2$ och dess längd $N = N_1 + N_2$, som sedan används för att beräkna dess volym $V = N * log_2 \eta$ (Shen, Conte, & Dunsmore, 1983, s. 156).

Vidare definieras ett programs svårighet $D = (\eta_1 * N_2) / 2\eta_2$ och ansträngningen E för att implementera ett sådant program som $E = D * V$ (Kearney, Sedlmeyer, Thompson, Gray, & Adler, 1986, s. 1045).

Halsteads volym är ett alternativt storleksmått som indikerar hur många variabler som finns i källkoden och hur dessa används. Volymen och ansträngningen har visat sig väl kunna förutsäga ett systems förändringsbarhet vid olika experiment (Coleman & Ash, 1994, s. 46). Av de olika Halsteadmåtten är dessa de mest använda och brukar inkluderas vid beräkningen av sammansatta kvalitetsmått, som t.ex. förvaltningsbarhetsindex (Welker, 2001).

2.4 Från webbsidor till ensidesapplikationer

2.4.1 Webbsidor och webbapplikationer

”Webbsidor” är idag ett vedertaget begrepp, men vad skiljer en webbsida från en webbapplikation? Generellt kan man säga att webbsidor är *informativa* medan webbapplikationer är *interaktiva* (Shapiro, 2013). En helt statisk sida som enbart tillhandahåller information kan anses vara en webbsida, medan en webbaserad epostklient där användare både kan läsa och skicka e-post är en interaktiv applikation. Det är dock sällan så enkelt att göra en tydlig gränsdragning mellan vad som

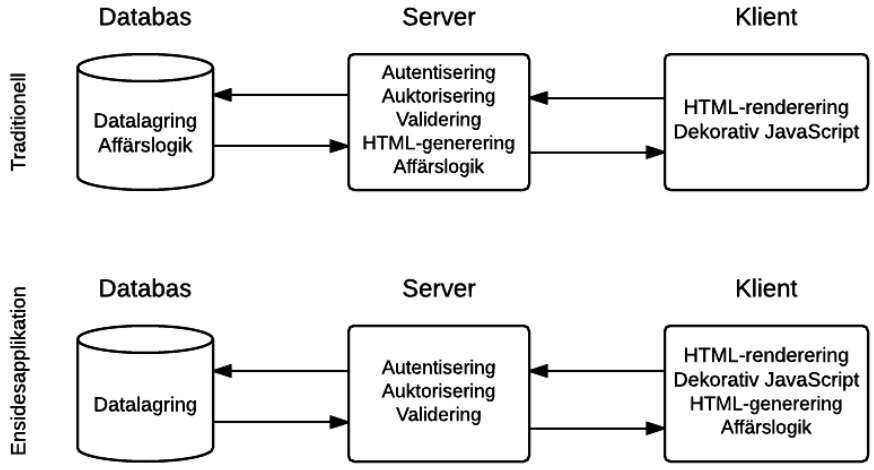
är en webbsida och vad som är en webbapplikation då en stor del av alla webbsidor idag innehåller någon form av interaktiva element.

2.4.2 Rika webbapplikationer

I början av 2000-talet myntades begreppet *rika webbapplikationer* – applikationer som körs i en webbläsare med funktionalitet och prestanda som är snarlik den hos traditionella datorprogram (Casteleyn, Garrig'os, & Maz'on, 2014; Mikowski & Powell, 2014). Gemensamt för alla tidiga rika webbapplikationer var att de krävde att användaren installerade insticksprogram i webbläsaren för att fungera eftersom de var skrivna i ett programspråk som webbläsaren inte kunde tolka (Mikowski & Powell, 2014, s. 5). Några vanligt förekommande exempel på dessa tidiga tekniker är Adobe Flash/Flex, Java Applet/Java FX och Microsoft Silverlight. De senaste åren har ett flertal tekniska innovationer gjort det möjligt att utveckla rika webb-applikationer enbart med hjälp av standardiserade webbprogramspråk, som HTML, JavaScript och CSS. Eftersom dessa språk har en standardiserad implementering i alla moderna webbläsare behövs inga insticksprogram för att köra dem. Trenden de senaste åren har därför varit att överge insticksprogrammen och istället utveckla rika webbapplikationer med hjälp av öppna ramverk skrivna i ren JavaScript (Mikowski & Powell, 2014, ss. 5-6).

2.4.3 Ensidesapplikationer

Rika webbapplikationer skrivna med JavaScript kallas vardagligt för *single-page applications* (SPA). För att underlätta språkbruket i rapporten har vi valt att använda en egen svensk översättning och benämner dem istället *ensidesapplikationer*. Rent tekniskt fungerar ensidesapplikationer genom att en webbserver laddar hela applikationen vid det första serveranropet, varefter utseende och funktion ändras dynamiskt med hjälp av JavaScript i användarens webbläsare (Mikowski & Powell, 2014). Det som avser ”en sida” i namnet ensidesapplikationer är den initiala sidladdningen. Figur 4 jämför ensidesapplikationer med den traditionella klient/server-arkitektur som används för de flesta webbsidor.



Figur 4: Traditionella webbapplikationer och ensidesapplikationer (Mikowski & Powell, 2014, s. 8)

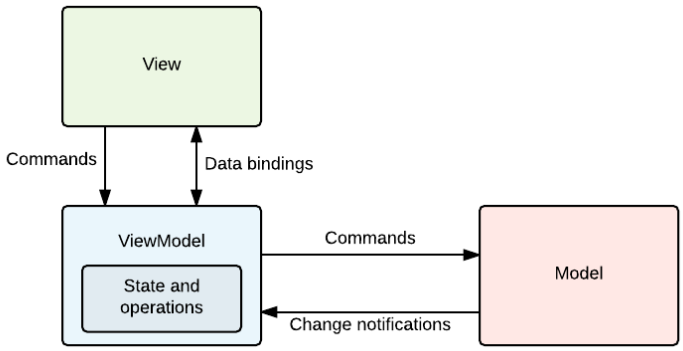
2.5 Utveckling av ensidesapplikationer

2.5.1 Varför ramverk och bibliotek?

Som figur 4 visar sker inte bara HTML-rendering utan även HTML-generering i webbläsaren i en ensidesapplikation. Att skapa HTML-sidor med interpolerad data från datalagret eller affärsprocesser kräver ofta stora mängder kod och är något som brukar ske på en server. En av de främsta uppgifterna för ett ramverk eller bibliotek är därför att underlätta skapandet av HTML-sidor och komponenter samt tillhandahålla effektiva algoritmer för att uppdatera (rendera om) vyn i webbläsaren. Att utgå från färdig funktionalitet kan ofta bespara ett projekt tusentals rader kod och leda till avsevärt kortare utvecklingscykler (Ruebbelke, 2015, s. 5).

2.5.2 AngularJS

Ett av de populäraste ramverken idag är AngularJS, utvecklat och underhållet av Google. AngularJS är ett omfattande ramverk som kommer med standardfunktioner med syfte att öka utvecklares produktivitet. Utmärkande för AngularJS är att det använder s.k. *tvåvägsdatabindning* för att interpolera data i HTML. Tvåvägsdatabindning fungerar i praktiken som en direkt koppling mellan en vy och datamodell. Om en användare manipulerar vyn uppdateras datamodellen, och om datamodellen ändras uppdateras vyn. AngularJS använder en variant av det klassiska MVC-designmönstret som kallas för MVVM (Ruebbelke, 2015). MVVM står för model, view, view-model och illustreras i figur 5.

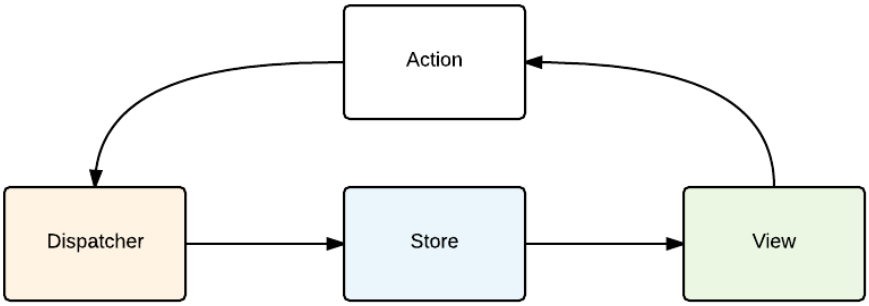


Figur 5: MVVM i AngularJS

2.5.3 React och Flux

React, som är skapat av Facebook, är ett relativt nytt bibliotek som snabbt blivit mycket populärt. Till skillnad från AngularJS, som erbjuder ett komplett paket för att skapa ensidesapplikationer, är React till för att generera HTML på ett så effektivt sätt som möjligt (Mardan, 2016). Fokus på snabb vyrendering har lett till att React har hög prestanda och lämpar sig väl för att skapa interaktiva webbapplikationer. React löser så få problem att de flesta är överens om att det inte är ett ramverk, utan snarare ett bibliotek med specialiserad funktionalitet (Mardan, 2016). Reacts tillämpningsbarhet som verktyg för att skapa ensidesapplikationer är dock vedertaget.

En annan viktig skillnad mot AngularJS är att React inte kommer med ett föreskrivet designmönster för att styra dataflöden i applikationer. Eftersom det är svårt att rent praktiskt bygga en applikation utan en genomtänkt arkitektur har Facebook lanserat *Flux* (se figur 6) som ett rekommenderat mönster att utgå ifrån när man skapar webbapplikationer med React (Mardan, 2016).



Figur 6: Flux (Mardan, 2016)

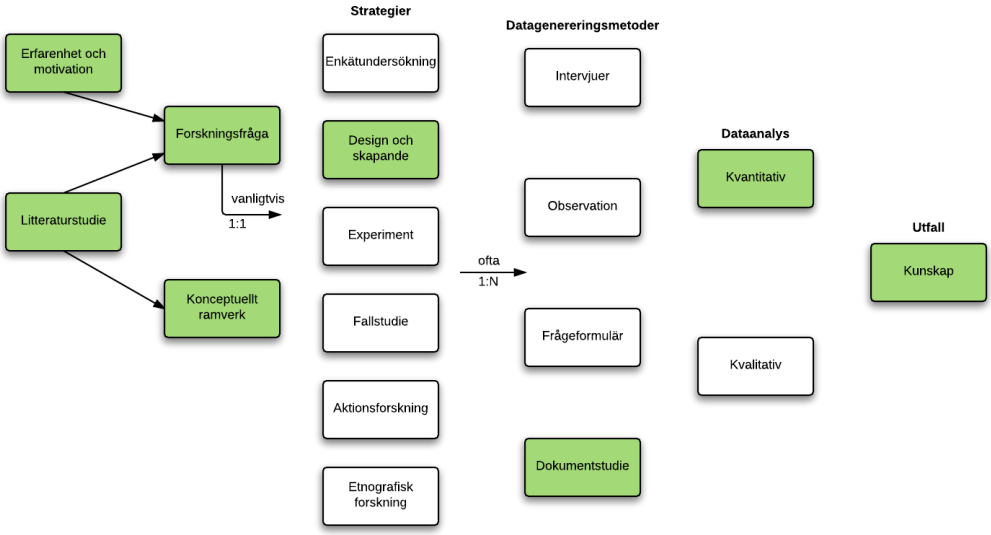
Flux, till skillnad från MVVM, undviker tvåvägsdatabindningar till förmån för ett enkelriktat dataflöde. När en användare interagerar med applikationens vy skapas händelser (actions) som styrs av en central enhet (dispatcher) och förs därefter ned genom applikationen i en enkel riktning (Mardan, 2016).

3 Metod

I det här kapitlet ger vi en översikt över vald forskningsmetodik (kap. 3.1) och redogör för val av forskningsstrategi (kap. 3.2) samt datainsamlings- (kap. 3.3) och analysmetod (kap. 3.4). Slutligen beskrivs hur arbetet har genomförts (3.5).

3.1 Forskningsmetodik

3.1.1 Forskningsprocessen



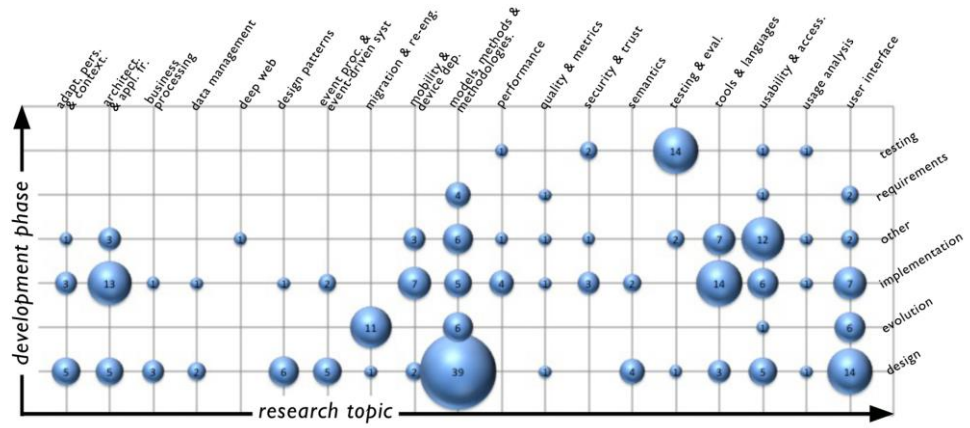
Figur 7. Överblick över forskningsprocessen, tagen från Oates (2006)

Figur 7 ger en överblick över den forskningsprocess som lett fram till detta arbete. Inledningsvis genomfördes en litteraturstudie (kap. 3.1.2) som gav underlag till ett konceptuellt ramverk och hjälpte oss att hitta en relevant forskningsfråga. Design och skapande användes som forskningsstrategi (kap. 3.2) genom att vi tog fram artefakter som medel för forskning (Oates, 2006, s. 109). Dokumentstudier genomfördes i form av statisk kodanalys samt jämförelse av källkodens förändring efter underhåll (kap. 3.3.2). Genom en kvantitativ analys (kap. 3.4) fick vi fram data för att kunna besvara frågeställningen och uppnå syftet, och därigenom ge ett beskrivande kunskapsbidrag.

3.1.2 Litteraturstudier

Enligt Oates (2006, s. 71) är litteraturstudien en tvådelad process som initialt syftar till att utforska ämnet och hitta uppslag för forskning, varefter en djupare litteraturstudie görs för underbygga kunskapsbidraget. Inför arbetet sökte vi information kring forskning om kvalitet för ensidesapplikationer för att kartlägga forskningsläget. Främst sökte vi information genom *IEEE*, *Summon* och *Google Scholar*, med nyckelord som*: rich internet applications*, *single-page applications* och *software quality*. Oates (2006, s. 83) menar även att en viktig del av litteraturstudien är att bedöma källornas trovärdighet, varför varje källa granskades kritiskt och bedömdes utifrån huruvida den var vetenskapligt granskad, hur ofta den refererats till samt vilken tidskrift som publicerat den. Det visade sig tidigt att det

finns en brist på vetenskapligt granskad litteratur kring ensidesapplikationers kvalitet. Detta bekräftades av en omfattande systematisk sammanställning av forskningsläget kring rika webbapplikationer (Casteleyn, Garrig'os, & Maz'on, 2014). Figur 8 visar en sammanställning av forskning bedriven kring rika webbapplikationer mellan 2004 och 2011.



Figur 8: Forskningsläget kring rika webbapplikationer 2014 (Casteleyn, Garrig'os, & Maz'on, 2014, s. 17)

Vidare fann vi en artikel som påpekade att webbapplikationers kvalitet ofta är åsidosatt till fördel för marknadskrafter och krav på snabba leveranstider (Maurya & Shankar, 2012, s. 33).

Den efterföljande, fördjupande litteraturstudien hjälpte oss att identifiera vår teoretiska referensram för arbetet, vilken utgörs av en beskrivande och en utvärderande modell för kvalitet. Dessa redogörs för i kapitel *2.2 Kvalitetsmodeller*. Litteraturstudien låg även till grund för den efterföljande resultatanalysen.

3.2 Forskningsstrategi

Design och skapande

Design och skapande är en forskningsmetod som syftar till att bedriva forskning genom skapande av en eller flera artefakter. Dessa artefakter kan ha olika roller i forskningen beroende på om syftet är att utvärdera en särskild utvecklingsmetod, undersöka hur en viss teknik fungerar i ett nytt sammanhang eller, som i vårt fall, som ett medel för vidare forskning (Oates, 2006, s. 109). För att uppnå syftet, *Att jämföra kodkvalitet och förvaltningsbarhet hos ensidesapplikationer utvecklade med AngularJS och React*, behövde vi tillgång till applikationer att jämföra. Dessa applikationer behövde vara funktionellt ekvivalenta för att de kvalitetsmässiga skillnaderna skulle kunna kopplas till AngularJS eller React snarare än andra faktorer. Om lämpliga applikationer funnits att tillgå hade vi kunnat utföra en fallstudie på dessa snarare än att utveckla egna artefakter.

Under förarbetet identifierade vi TodoMVC som en befintlig källa till funktionellt ekvivalenta applikationer (TodoMVC, 2016). Vi valde ändå att utveckla egna applikationer, då de som fanns att tillgå var framtagna för att visa hur applikationer kan struktureras med olika ramverk och bibliotek snarare än som funktionella demon-

strationer (TodoMVC, 2016). Att utveckla egna artefakter gav oss också möjligheten att enkelt planera för utökning av funktionalitet för att mäta förvaltningsbarhet efter förändring.

I valet av vilken sorts artefakter som skulle utvecklas var två faktorer viktiga att ta i beaktande. För det första var vi i behov av artefakter av lämplig storlek som utnyttjade så mycket som möjligt av den funktionalitet som AngularJS och React erbjöd, samtidigt som vi behövde förhålla oss till givna tidsramar. För det andra behövde artefakterna vara tillräckligt verklighetsnära för att kunna möjliggöra generaliserbara resultat. Givet detta valde vi att utveckla webbshoppar, då sådana dels är en vanligt förekommenade applikationstyp och dels kräver flera vyer och en lämplig mängd affärslogik.

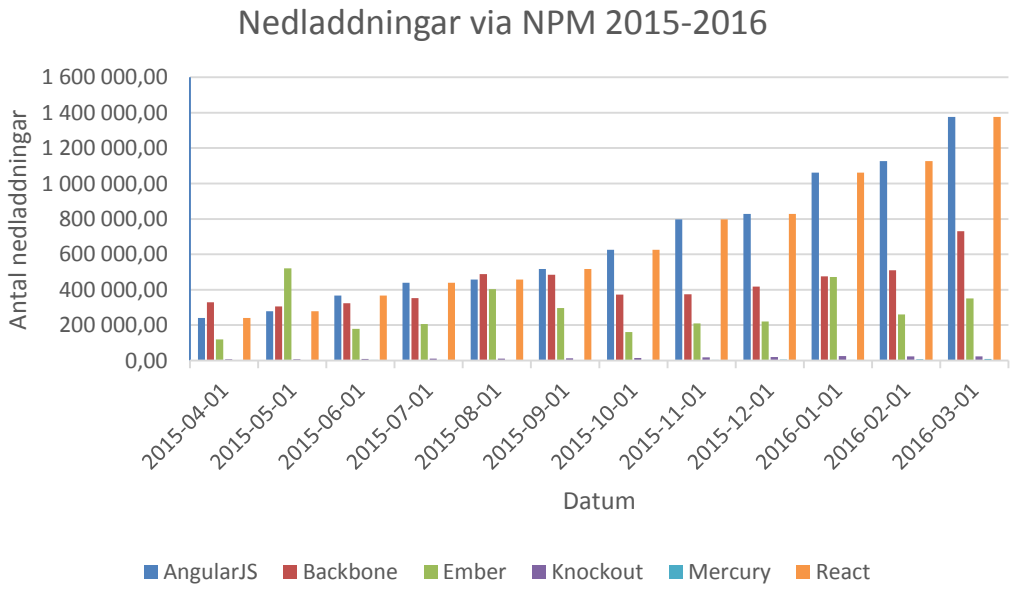
3.3 Datainsamling

3.3.1 Urval av ramverk och bibliotek

Under litteraturstudien identifierades ett antal olika ramverk och bibliotek som kan användas vid utveckling av ensidesapplikationer. Dessa inventerades och de som inte var färdigutvecklade eller befann sig i betastadium undantogs. Det slutgiltiga valet av ramverk och bibliotek baserades av bekvämlighet på popularitet. Populariteten beräknades utifrån två parametrar:

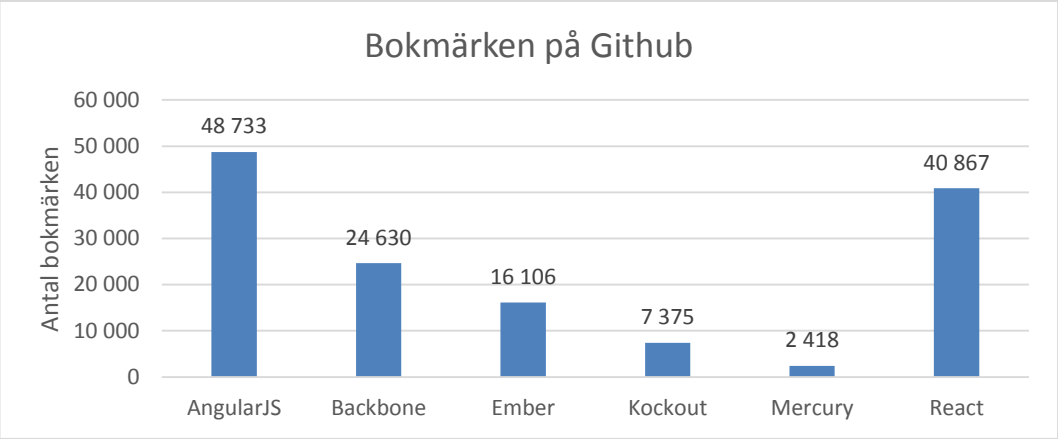
- Antal nedladdningar via NPM, som är en distributionskanal för JavaScript-projekt
- Antal bokmärken på GitHub, där projektens källkod hanteras

NPM är inte den enda distributionskanalen, och ger därför inte en komplett bild av användningen av ramverk och bibliotek, men det ger en god uppskattning. Figur 9 visar antalet nedladdningar bland kandidater vi valt mellan. Det framgår tydligt att AngularJS och React är de överlägset mest nedladdade produkterna via NPM.



Figur 9. Antal nedladdningar via NPM under 2015-2016

Gemensamt för alla ramverk och bibliotek är att deras källkod lagras på GitHub. GitHub har en funktion som låter utvecklare spara bokmärken till kodbaser som de tycker är intressanta. Figur 10 visar totala antalet bokmärken för respektive kodbas, vilket ger en indikation på popularitet.



Figur 10: Bokmärken på GitHub

Antalet bokmärken på GitHub visar en samstämmig popularitetsordning jämfört med antalet nedladdningar från NPM. Sammantaget medförde detta att vi valde AngularJS som ramverk och React som bibliotek.

3.3.2 Dokumentstudier

Dokumentstudier som datagenereringsmetod

För att kunna mäta och jämföra artefakternas kodkvalitet och förvaltningsbarhet behövde vi studera deras källkod. Huruvida källkod bäst konceptualiseras som en sorts dokument, likt en byggnadsritning, eller som intrinsiska aspekter av en entitet – likt de rör, ledningar och övriga komponenter som ingår vid uppförandet av en byggnad – är avhängigt forskarens syfte och vilket teoretiskt perspektiv som anläggs.

För att undvika utsvävningar om verklighetens beskaffenhet och källkodens ontologiska natur nöjer vi oss med att konstatera att båda ansatserna är förenliga med vårt syfte såväl som övergripande metodik och forskningsstrategi. Konceptet *källkod-som-dokument* kräver dock färre metafysiska ställningstaganden då det låter oss referera till Oates (2006), som definierar dokument som ”*alla symboliska representationer som kan lagras och återhämtas för analys*” (Oates, 2006, s. 235). Vidare låter detta oss ytterligare definiera artefakternas källkod som, av forskaren, skapade dokument innehållande *data* som kan analyseras (Oates, 2006, s. 239), vilket sammanfaller väl med vårt syfte.

En alternativ datagenereringsmetod som övervägdes var att låta erfarna utvecklare hos vår samarbetspartner studera och kvalitetsbedöma all källkod för att sedan genomföra intervjuer med dem. I slutändan valde vi ändå att genomföra statisk kodanalys, då detta direkt genererade objektiva värden att analysera och bättre sammanföll med vår kvantitativa ansats. Att erhålla expertutlåtanden via intervjuer hade också involverat fler aktörer, vilket hade tagit mer tid och resurser i anspråk samt

ökat komplexiteten i analysen då fler subjektiva faktorer att ta i beaktande uppstår, som t.ex. forskarens inverkan på datainsamlingen eller respondenternas förkunskap, motivation och uppsåt (Oates, 2006, ss. 198-199).

Statisk kodanalys

Vi använde oss av Plato, som är ett verktyg för statisk kodanalys, för att analysera artefakternas källkod och ta fram mätvärden för kodkvalitet. Detta innefattar förvaltningsbarhetsindex och de komplexitetsmått som krävs för beräkning därav. Endast den källkod vi själva producerat analyserades. För att ge en rättvisande bild vid jämförelsen av artefakterna inkluderades all HTML-kod vid beräkningen av antal rader kod. Underliggande biblioteks- ramverkskod undantogs då den i normala fall inte är föremål för förändring annat än av leverantören själv.

De första mätningarna gjordes efter att artefakterna var färdigutvecklade enligt kravspecifikation (se bilaga 1). Då förvaltningsbarhetsindex endast ger en övergripande indikation för applikationernas förändringsbarhet (Welker, 2001, s. 20) gjordes nya mätningar på respektive artefakt efter planerade underhållsarbeten där ytterligare funktionalitet lades till i två steg (se bilaga 1). Detta gav oss mätvärden för hur mycket komplexiteten för respektive artefakt ökat relativt till en baslinje. Utöver detta undersöktes även hur många rader kod som modifierades och lades till i respektive applikation vid förändringsarbetet. Data för detta hämtades från källkodhanteringssystemet och användes för att ge ytterligare ett perspektiv på artefakternas förändringsbarhet.

Verktyg

Då de verktyg vi använde både analyserar källkoden och utför beräkningarna av komplexitet och förvaltningsbarhetsindex är det av stor vikt att de håller hög kvalitet. De värden som beräknas måste vara relevanta och tillförlitliga för att ha något värde i vetenskapliga sammanhang. Plato, som användes för den statiska kodanalysen, är ett verktyg som skapar rapporter av data framtagna med andra verktyg, däribland *escomplex* (se bilaga 3 för en redogörelse över använda verktyg och deras programversioner).

Escomplex är ett verktyg för kodkomplexitetsanalys som nyttjar den ursprungliga formeln för förvaltningsbarhetsindex utan kommentarer (Stilwell, 2016) som redogjordes för i *2.3.1 Förvaltningsbarhetsindex*. Det hade varit önskvärt med ett verktyg som använder någon av de modernare formlerna för förvaltningsbarhetsindex men då escomplex är det enda verktyg som i skrivande stund finns tillgängligt för kodanalys av JavaScript var detta inte möjligt. Detta har dock ingen inverkan på våra resultat eftersom vi inte enbart gör en mätning för respektive artefakt utan flera i takt med att funktionalitet läggs till, vilket möjliggör relativa jämförelser artefakterna emellan.

3.4 Dataanalys

Kvantitativ analys

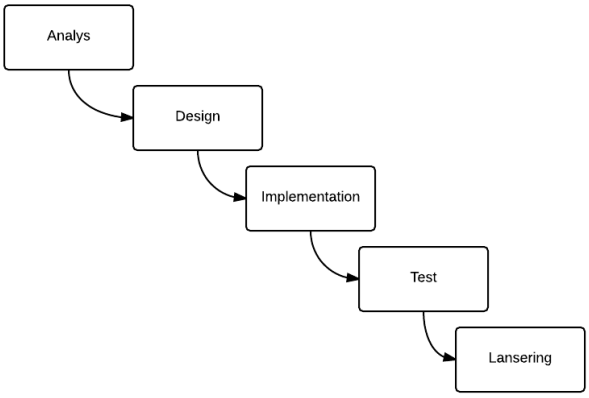
Syftet med en kvantitativ analys är att identifiera mönster i insamlad data för att sedan kunna dra slutsatser från dessa (Oates, 2006, s. 245). De verktyg vi använde

oss av för statisk kodanalys genererar kvantitativa data, närmare bestämt diskreta intervalldata. Kvantitativa data kan i sin enklaste form presenteras i tabeller, diagram och grafer för att visualisera mönster och trender. Vidare kan grundläggande statistiska metoder, t.ex. beräkning av medelvärde, användas för att identifiera ytterligare mönster. (Oates, 2006, s. 246)

3.5 Genomförande

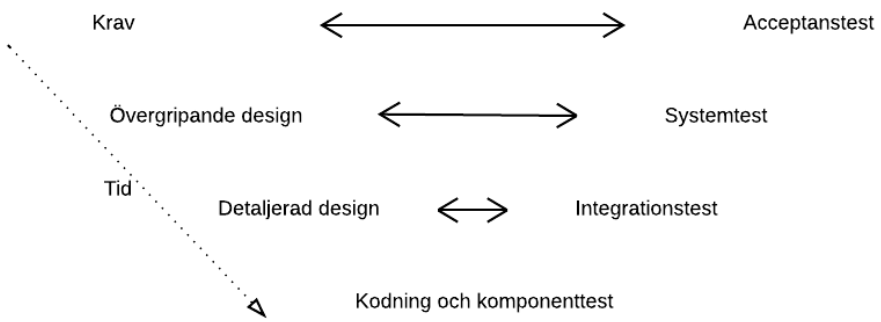
3.5.1 Val av utvecklingsmetod

Innan utvecklingen av artefakterna utvärderade vi ett antal lämpliga utvecklingsmetoder. Utvecklingsmetoder kan generellt delas in i *sekventiella* eller *iterativa* metoder. Sekventiella metoder delar in systemutvecklingen i olika faser, där avslutandet av en fas initierar en annan. Den mest kända av dessa är den så kallade *vattenfallsmodellen* där utveckling börjar med analys/kravfångst, för att gå vidare med design, implementation, test och slutligen lansering/drift. Vattenfallsmodellen illustreras i figur 11.



Figur 11: Vattenfallsmodellen

En annan sekventiell metod är *v-modellen.* V-modellen lägger ett större fokus på kvalitetssäkring genom test. Detta genom att integrera olika testnivåer med de olika delarna av livscykeln. Figur 12 visar hur olika faser på vänster sida i modellen har en motsvarande testfas, där testfall tas fram parallellt. Exempelvis tas acceptanstest fram redan under analys/kravarbetet (Eriksson, 2008).

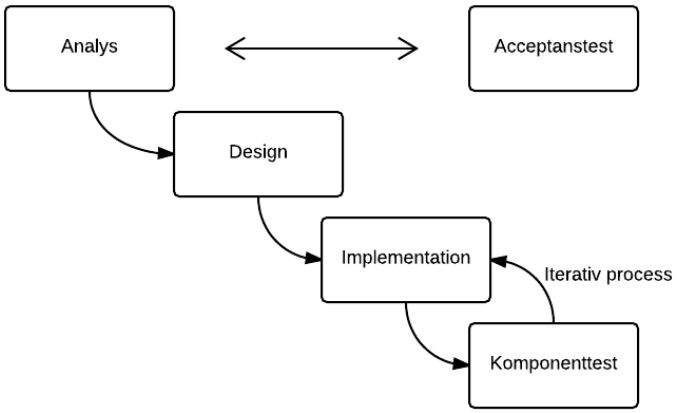


Figur 12: V-modellen (Eriksson, 2008)

Iterativa modeller kallas vanligtvis för *agila* modeller. Dessa skiljer sig från de sekventiella i att utvecklingen går i cykler, där planering, utveckling, test och utvärdering sker i avgränsade perioder, vilket ger en större flexibilitet vid förändrade behov och krav.

Vilken metod som är mest lämpad beror till stor del på hur pass troligt det är att kraven ändras, vilken den tänka målgruppen är samt utvecklingens omfattning (Balaji & Murugaiyan, 2012). Eftersom de artefakter vi utvecklade var tänkta att vara funktionellt ekvivalenta fanns ingen vinning i att använda en iterativ process. V-modellen lämpar sig bäst i stora projekt där kraven på validering är stora (Balaji & Murugaiyan, 2012, s. 29). Att använda vattenfallsmodellen vore därmed rimligt för utvecklingen av våra artefakter, i och med att vi hade bestämda krav, saknade slutanvändare och enbart behövde validera att kraven uppnåddes. I slutändan valde vi att använda en anpassad variant av vattenfallsmodellen som lånar både från iterativa modeller och v-modellen, detta för att arbeta testdrivet för att säkerställa artefakternas kodmässiga kvalitet.

Figur 13 illustrerar vår utvecklingsmodell. Till en början togs de funktionella kraven fram och parallellt bestämdes acceptanstest för att validera kraven. Analysen ledde fram till ett dokument i form av en kravspecifikation (se bilaga 1). Designfasen utfördes genom att ta fram designskisser (se bilaga 2) samt HTML-mallar som kunde ligga till grund för samtliga artefakter. Implementationen skedde sedan iterativt. För att garantera kodens testbarhet och minska dess komplexitet (Martin, 2008, s. 172) skrev vi test innan vi skrev programkod.



Figur 13: Vår utvecklingsmodell

Samma utvecklingsmodell användes vid bägge underhållsarbetena för respektive applikation. Utökade kravspecifikationer (se bilaga 1) och designskisser (se bilaga 2) togs fram och dessa implementerades sedan med iterativ, testdriven utveckling.

3.5.2 Planerade underhållsarbeten

De underhållsarbeten som planerades var sådana som var av lämplig art och omfattning för vår typ av artefakter samtidigt som de uppfyllde de krav vi hade från vår samarbetspartner. Underhållsarbete 1 kom därför att utgöras av ett rekommendationssystem för att föreslå ytterligare produkter för användaren när denne lagt en

produkt i varukorgen samt på produktsidorna för att visa produkter andra kunder inhandlat. Sådan funktionalitet krävde utökning och modifikation av befintlig kod samt ytterligare affärslogik för att implementeras, vilket är vanligt förekommande vid underhållsarbete (McConnel, 2004).

Underhållsarbete 2 var något större och involverade geografiska data, vilket var det enda uttryckta kravet från vår samarbetspartner. En kartkomponent med markörer för fiktiva fysiska butiker lades till tillsammans med öppettider för dessa. Öppettiderna skulle även visas i respektive applikations navigeringsmeny efter ett klick på motsvarande kartmarkör. Utöver nyskriven kod att infoga i de befintliga kodbaserna ställde detta även krav på utökning och förändring av applikationernas interna dataflöden, vilket är något som sätter en god och genomtänkt applikationsarkitektur på prov.

3.5.3 Hantverksmässigt kodande

För att säkerställa att de utvecklade artefakterna höll en god grundkvalitet strävade vi efter att hålla oss till bästa praxis avseende kodens stilregler, projektens struktur samt projektens arkitektur för dataflöden och datahantering. Inför utvecklingen togs även prototyper fram. Detta för att säkerställa att skillnaderna i de slutgiltiga artefakterna inte skulle bero på godtyckliga skillnader i kodmässig standard.

Prototyper

Hunt och Thomas (2000) menar att lärandet är den främsta fördelen med att utveckla prototyper. Vi utvecklade prototyper som uppfyllde de ursprungliga kraven (se bilaga 1) för att ta med eventuella lärdomar om AngularJS och React i de slutliga artefakterna. Således minskade vi även risken att vi skulle ha bättre kännedom om AngularJS och React under underhållsarbetena än under den initiala utvecklingen.

Stilregler

Stilregler avser hur JavaScript-kod bör skrivas för att öka dess läsbarhet och minska risken för defekter. Detta inkluderar exempelvis gränser för hur många parametrar en metod får ta emot eller hur många rader kod en metod som mest bör bestå av. För att upprätthålla reglerna användes Eslint (jQuery Foundation, 2016) som kodanalysverktyg under hela utvecklingsprocessen. Eslint konfigurerades för att följa bästa praxis med stilregler från Airbnb, som har regler för både ren JavaScript-kod samt specifikt för React (Airbnb, 2016).

Arkitektur

Eftersom de utvecklade artefakterna i sig hade förhållandevis lite funktionalitet hade det gått bra att utveckla dem ad-hoc, utan en genomtänkt strategi för hantering av data. För att ge ett mer realistiskt komplexitetsmått följdes dock bästa praxis för datadesign för både AngularJS och React. AngularJS är byggt för användas med ett MVVM-mönster, varför det föll sig naturligt att förhålla sig till det. Eftersom React enbart är ett bibliotek för att hantera vyer har det inget föreskrivet sätt att hantera applikationers data, men det av tillverkaren rekommenderade sättet är att använda

Flux. Flux är ett designmönster som är implementerat i ett bibliotek för att underlätta användningen tillsammans med React.

Utvecklingsprinciper

För att förhindra att artefakternas grundkvalitet skiljdes åt utgick vi från vedertaget goda principer under utvecklingen. SOLID är samling principer, ursprungligen framtagna för objektorienterade program, för att främja kod som är förvaltningsbar och håller hög inre kvalitet (Martin, 2003). En viktig del av SOLID är *Single Responsibility Principle (SRP),* som syftar till att källkod bör struktureras i små delar som endast har en orsak att förändras (Martin, 2008, s. 138). En annan viktig SOLID-princip är *Open Closed Principle (OPC)*. OPC handlar om att kod ska vara stängd för modifikation men öppen för utökning (Martin, 2008, s. 149). Ett annat sätt att uttrycka det är att ny funktionalitet ska kunna införas genom att ny kod skrivs, snarare än att befintlig kod ändras. Vi strävade efter att uppfylla dessa principer genom att utveckla artefakterna genom löst kopplade delar med hög inre samhörighet.

AngularJS-artefakten delades in i olika moduler efter funktion (se figur 14), vilket gav en god separation mellan artefakternas olika delar. I varje modul gjordes separata filer för vyer, vymodeller och tjänster. Inga tjänster eller vymodeller hade några direkta beroenden mellan varandra.

```
.
├── about
│   ├── about.controller.js
│   ├── about.html
│   └── about.module.js
├── app.constants.js
├── app.module.js
├── assets
│   └── styles
│       └── styles.css
├── cart
│   ├── cart.controller.js
│   ├── cart.html
│   ├── cart.module.js
│   └── cart.service.js
├── common
│   ├── common.module.js
│   └── common.store.service.js
├── contact
│   ├── contact.html
│   └── contact.module.js
├── index.html
├── productList
│   ├── product.html
│   ├── productList.controller.js
│   ├── productList.directive.js
│   ├── productList.html
│   ├── productList.module.js
│   ├── productList.products.service.js
│   └── products.json
└── shell
    ├── inline-cart.directive.js
    ├── nav.controller.js
    ├── shell.module.js
    └── shell.routes.js
```

```
.
├── App.jsx
├── actions
│   ├── cartActions.js
│   └── storeActions.js
├── components
│   ├── about
│   │   ├── about.page.jsx
│   │   └── map..jsx
│   ├── cart
│   │   ├── cart.page.jsx
│   │   └── cart.row.jsx
│   ├── common
│   │   ├── addToCart.jsx
│   │   ├── footer.component.jsx
│   │   └── navbar.component.jsx
│   ├── contact
│   │   ├── contact.field.jsx
│   │   └── contact.page.jsx
│   └── products
│       ├── product.card.jsx
│       ├── product.list.jsx
│       ├── product.page.jsx
│       ├── product.searchFilter.jsx
│       ├── product.sortOrder.jsx
│       └── products.page.jsx
├── constants
│   └── actionTypes.js
├── dispatcher
│   └── appDispatcher.js
├── index.html
├── index.js
├── products.json
├── stores
│   ├── cartStore.js
│   ├── physicalStoreStore.js
│   └── productStore.js
└── styles.css
```

Figur 14: Struktur av artefakter. AngularJS (vänster) och React (höger)

React har ingen naturlig separation mellan vy och vymodell likt den AngularJS har. Den enda byggstenen är en *komponent*. För att separera vykod och logik skapade vi främst komponenter vars huvudsakliga syfte var att presentera data, medan ett fåtal andra komponenter försåg dessa med data som hämtades från modeller via Flux. Med hjälp av denna separation kunde standardkomponenter lätt återanvändas inom olika delar av artefakten. Projektet strukturerades efter funktion på ett liknande sätt som AngularJS-artefakten.

3.6 Mätning av förvaltningsbarhetsindex och källkodsförändringar

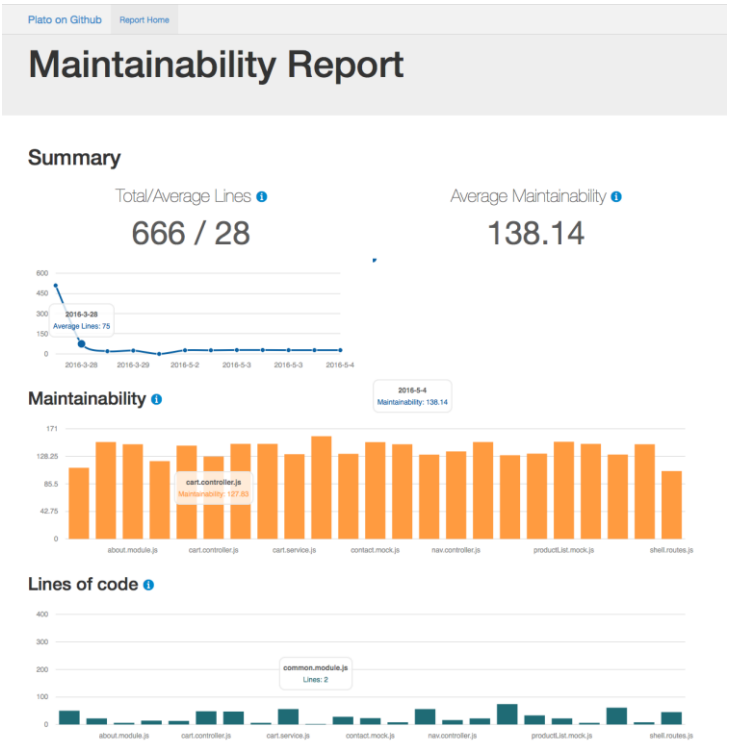
3.6.1 Förvaltningsbarhetsindex med Plato

Mätningarna av förvaltningsbarhetsindex utfördes med Plato (kap. 3.3.2) genom att först placera alla källkodsfiler i en mapp och sedan analysera dem. Denna process automatiserades med Gulp, vilket är ett automatiseringsverktyg som används via kommandotolken. Exempel på en körning i kommandotolken visas i figur 15.

```
[13:57:07] Using gulpfile ~/examensarbete/artefakter/artifact-react/gulpfile.js
[13:57:07] Starting 'plato'...
[13:57:07] Finished 'plato' after 22 ms
```

Figur 15: Kodanalys med Gulp och Plato

Plato mäter endast förvaltningsbarhetsindex på JavaScript-filer, vilket inte är något problem för React som endast använder JavaScript. Däremot har AngularJS ett antal HTML-filer, vilket försvårar analysen. HTML är dock ett märkspråk utan någon logisk funktionalitet eller cyklomatisk komplexitet. AngularJS placerar vissa egna attribut i HTML-element som används för att generera ytterligare HTML från dynamisk data. Denna kod är deklarativ medan den imperativa, komplexa, koden ligger i den relaterade vymodellen (Ruebbelke, 2015, s. 30). Vi fann inget vedertaget sätt att ta med HTML-filer i en förvaltningsbarhetsuträkning, varför vi valde att räkna HTML enbart som rader kod. Vi räknade heller inte in stilmallar (CSS) till komplexiteten för någon av artefakterna, vilket ändå inte hade någon inverkan då samma stilmallar användes för båda artefakterna.



Figur 16: Platorapport

Plato genererade både ett genomsnittligt förvaltningsbarhetsindex (kap. 4.1) och förvaltningsbarhetsindex per fil (kap. 4.2). Ett exempel på en färdig rapport kan ses i

figur 16. En rapport genererades efter att artefakterna var färdiga i sitt grundutförande, varefter ytterligare rapporter togs fram efter varje slutfört underhåll (kap. 4.1). Data från Plato matades in i kalkylark i Excel som användes för att kategorisera data i tabeller och visualisera data i olika typer av diagram.

3.6.2 Mätning av källkodsförändringar

Git användes som versionshanteringssystem för bägge artefakterna. Med hjälp av Git kunde vi spara all kod efter grundutförandet samt efter varje utfört underhåll. Koden lagrades på GitHub, som även har verktyg för att se skillnaden mellan källkoden i olika faser. Med detta verktyg kunde vi lätt jämföra hur mycket kod och hur många filer som lagts till eller tagits bort efter varje underhåll. Figur 17 visar hur en källkodsfil förändrats genom att två rader kod lagts till mellan underhåll 1 och underhåll 2 i AngularJS-artefakten.

Figur 17: Exempel på jämförelse av källkod

Kodförändring mättes i samtliga filer över samtliga etapper och samanställdes i diagram (kap. 4.3).

4 Resultat

I det här kapitlet redogör vi för de resultat som framkom utifrån utförd källkodsanalys. Kapitel 4.1 visar skillnader i uppmätt förvaltningsbarhetsindex. Kapitel 4.2 ger en visuell representation över hur den genomsnittliga förvaltningsbarheten är distribuerad per fil. I kapitel 4.3 redogörs för hur själva källkoden förändras i storlek efter varje genomfört underhåll.

4.1 Förvaltningsbarhetsindex

De värden för förvaltningsbarhetsindex som framkom under analysen visar små skillnader mellan AngularJS och React. Tabell 1 visar respektive grundvärden samt förändringarna vid bägge underhållsarbetena och tabell 2 visar de gränsvärden för förvaltningsbarhet som redogjordes för i kapitel *2.3.1 Förvaltningsbarhetsindex*.

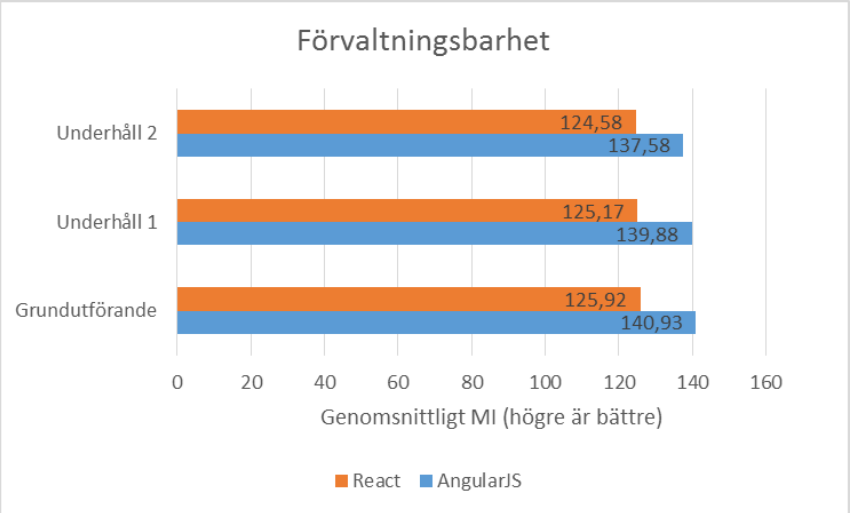
Genomsnittligt förvaltningsbarhetsindex (MI)			
Ramverk	**Grundutförande**	**Underhåll 1**	**Underhåll 2**
AngularJS	140,93	139,88	137,58
React	125,92	125,17	124,58

Tabell 1. Genomsnittligt förvaltningsbarhetsindex

Gränsvärden för förvaltningsbarhetsindex (MI)	
>85	God förvaltningsbarhet
65-85	Måttlig förvaltningsbarhet
<65	Undermålig förvaltningsbarhet

Tabell 2. Gränsvärden för förvaltningsbarhetsindex (Sjøberg, Anda, & Mockus, 2012, s. 108)

Den generella trenden är att förvaltningsbarhetsindex blir sämre för bägge applikationerna när de utökas med mer funktionalitet, vilket i sig inte är underligt då blott en ökning av antal rader kod sänker förvaltningsbarhetsindex. Figur 18 visar att även om skillnaderna före och efter underhållsarbetena är små för respektive applikation så finns det en skillnad mellan AngularJS och React, där AngularJS generellt har bättre värden för förvaltningsbarhetsindex. Bägge artefakterna har dock ett genomsnittligt förvaltningsbarhetsindex långt över 85, vilket visar på mycket god förvaltningsbarhet.



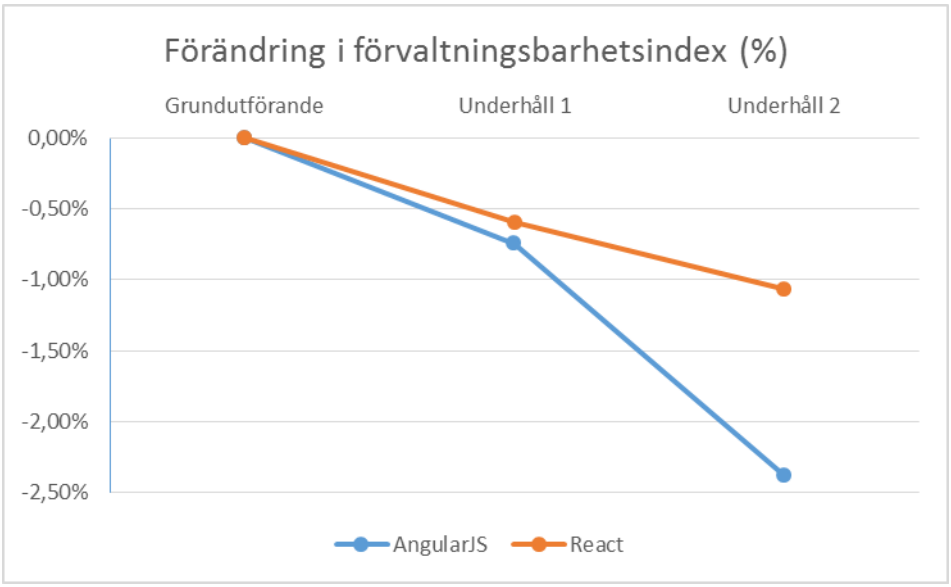
Figur 18. Förvaltningsbarhet

Om förändringen i förvaltningsbarhetsindex beräknas som en procentuell förändring i förhållande till värdet för respektive applikations grundutförande (tabell 3) framkommer att utbyggnad av AngularJS-applikationen snabbare leder till minskningar i förvaltningsbarhetsindex jämfört med React-applikationen.

Förändring av förvaltningsbarhetsindex i %		
Ramverk	**Underhåll 1**	**Underhåll 2**
AngularJS	– 0,7 %	– 2,4 %
React	– 0,6 %	– 1,1 %

Tabell 3. Förändring av förvaltningsbarhetsindex i %

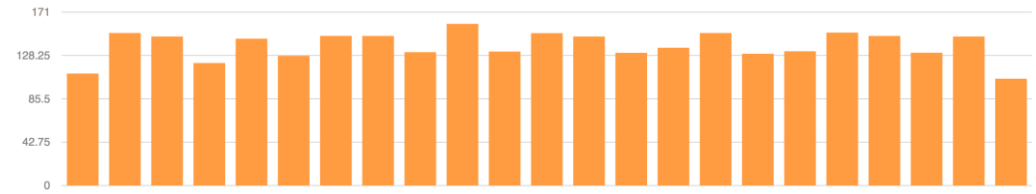
Denna trend syns tydligt i figur 19. Även då bägge applikationerna utökades med samma funktionalitet drabbas AngularJS-applikationen av försämrade värden i snabbare takt.



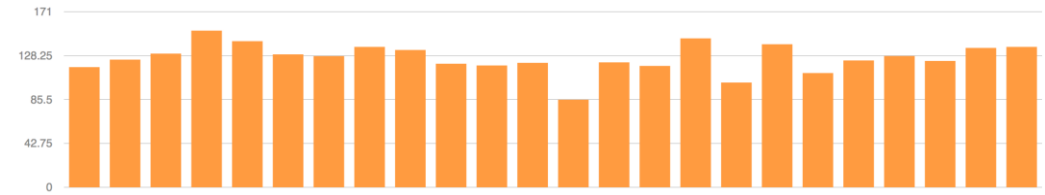
Figur 19. Procentuell förändring av förvaltningsbarhetsindex

4.2 Förvaltningsbarhetsindex per fil

Om förvaltningsbarhetsindex beräknas per fil i respektive kodbas framträder en jämn distribution av värden där ingen enskild fil understiger godkända tröskelvärden. Detta visar att komplexiteten i artefakterna är väl utspridd, vilket indikerar en god separation av funktionalitet. Figur 20 visar fördelningen för AngularJS-applikationen och figur 21 den för React-applikationen.



Figur 20: Förvaltningsbarhetsindex per fil i AngularJS efter underhåll 2

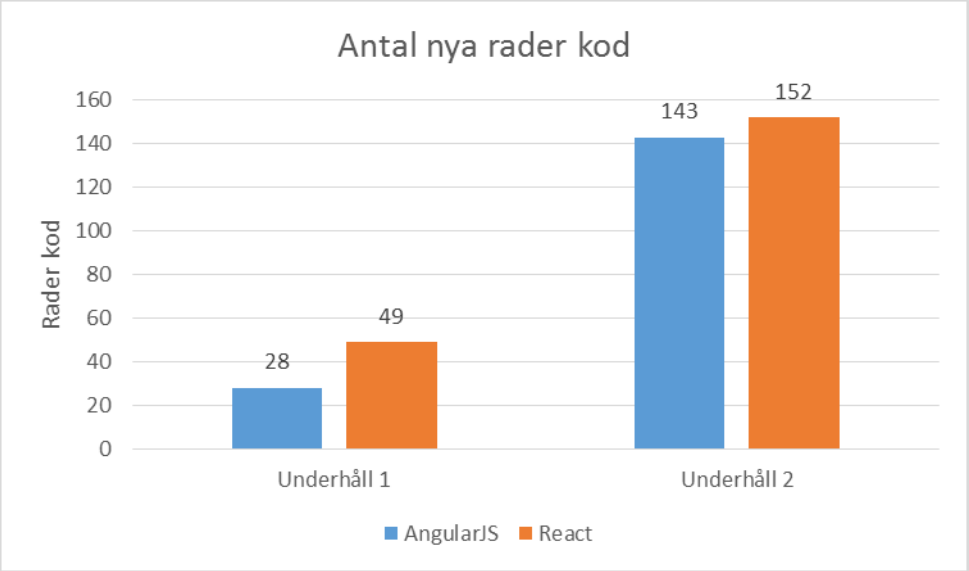


Figur 21: Förvaltningsbarhetsindex per fil i React efter underhåll 2

De filer som har lägst förvaltningsbarhetsindex i respektive applikation är de som innehåller logik för routing, detta då både AngularJS och React kräver många variabeldeklarationer och funktioner med åtskilliga parametrar för detta.

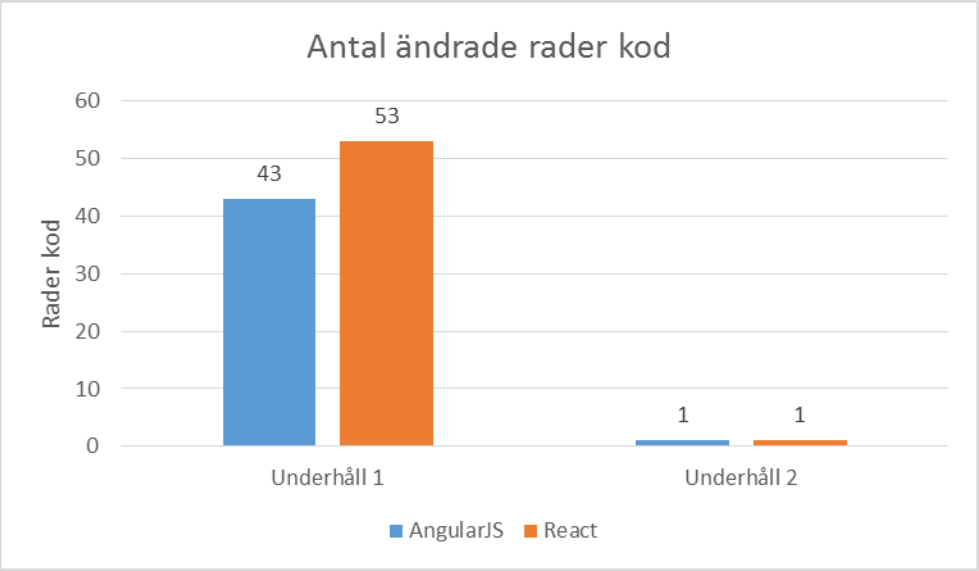
4.3 Förändringar i kodbaserna

Figur 22 visar antalet nya rader kod som krävdes för respektive underhållsarbete (vilka redogörs för i kap. *3.5.2 Planerade underhållsarbeten*) i applikationerna. Föga förvånande krävdes det mindre kod för att bygga ut AngularJS-applikationen, detta då ramverket tillhandahåller mer grundfunktionalitet än vad React gör.



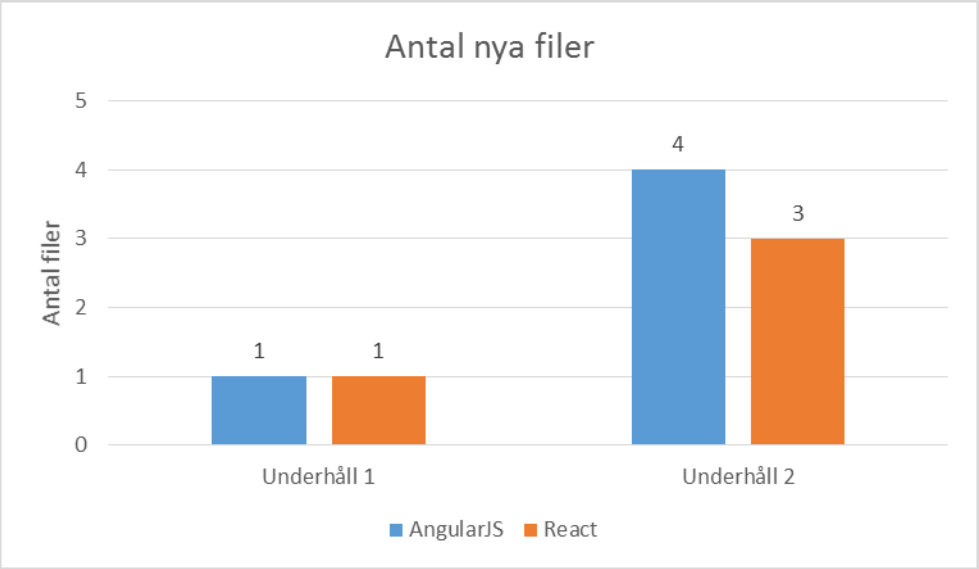
Figur 22. Antal rader ny kod vid utbyggnad av funktionalitet

Då underhåll 1 avsåg produktrekommendationer både i varukorg och på produktsidor medförde implementeringen förhållandevis stora modifikationer av den befintliga koden, som figur 23 visar. Dessa modifikationer utgjordes av förändring eller flytt av kod. Underhåll 2, däremot, bestod av helt ny funktionalitet och genererade betydligt fler rader ny kod men krävde endast att en rad kod modifierades i respektive applikation. Inga kodrader togs bort vid något av underhållsarbetena.



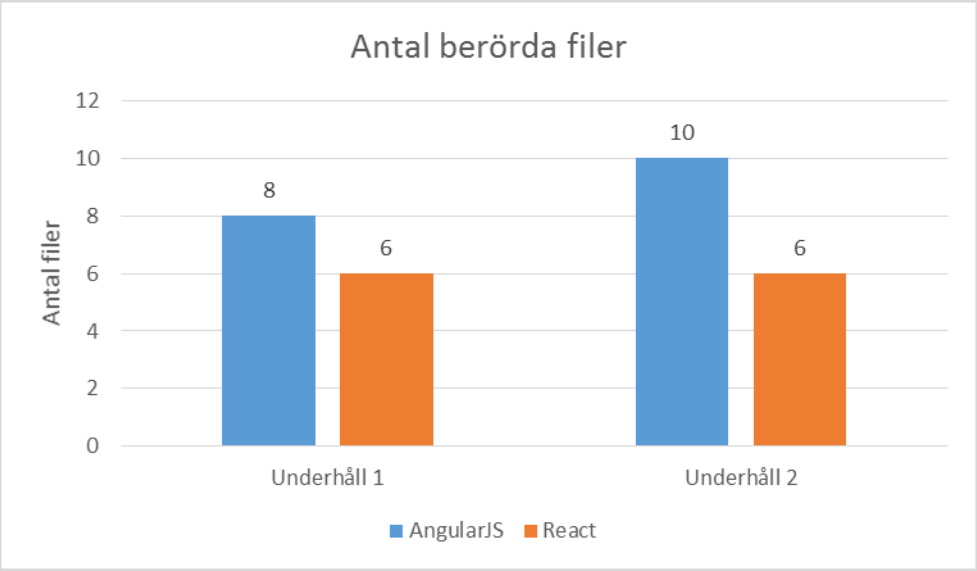
Figur 23. Antal borttagna rader kod vid utbyggnad av funktionalitet

Sett till hur många nya filer som tillkom är det ingen markant skillnad mellan applikationerna (se figur 24). Det krävdes få filer för att lägga till den nya funktionaliteten i underhåll 1 men desto fler för underhåll 2, som var mer omfattande.



Figur 24. Antal nya filer vid utbyggnad av funktionalitet

Större skillnader framkom när antal berörda filer undersöktes (se figur 25). Överlag berördes fler filer av förändringar när underhållsarbete utfördes i AngularJS-applikationen. Vid underhåll 1 var skillnaden förhållandevis liten men underhåll 2 visade en skillnad på nära dubbelt så många berörda filer i AngularJS-applikationen jämfört med React-applikationen.



Figur 25. Antal berörda filer vid utbyggnad av funktionalitet

5 Analys och diskussion

I det här kapitlet diskuterar vi de resultat som uppmätts och återkopplar dem till problemformulering (kap. 1.2) och syfte (kap. 1.3).

En sammanställning av den vetenskapliga litteraturen kring rika webbapplikationer visar att det bedrivs ytterst lite forskning om kvalitet och mått för kvalitet beträffande dessa, vilket vi redogör för i kapitel *3.1.2 Litteraturstudier*. De studier som har gjorts kring kvantitativa mått för kvalitet fokuserar på förvaltningsbarhetsindex och dess användning för företag och förvaltningsorganisationer inom traditionell mjukvaruutveckling snarare än webbutveckling, vilket framkommer i kapitel *2.3.1 Förvaltningsbarhetsindex*. Eftersom webbutveckling karaktäriseras av korta leveranstider är det viktigt att kunna mäta och utvärdera förvaltningsbarhet kvantitativt och automatiserat tidigt under utvecklingen för att möta de kvalitetskrav dagens webbanvändare ställer (se kap. *1.1 Bakgrund* och kap. *3.1.2 Litteraturstudier*) Forskning om kvalitet och förvaltningsbarhet för ensidesapplikationer saknas helt (se kap. *3.1.2 Litteraturstudier).*

Att jämföra förvaltningsbarhetsindex mellan två olika artefakter kan vara vanskligt då kvalitativa skillnader dessa emellan riskerar att göra jämförelsen orättvis (Welker, 2001, s. 19). Exempelvis bäddar React in kod för att generera HTML i sina JavaScript-filer, vilka inkluderas vid mätningen av förvaltningsbarhetsindex. Detta gör att mycket av den kod som skrivs för React har låg komplexitet och är utspridd på många rader. AngularJS däremot har HTML-kod i mallar som inte kan hanteras av Plato och får därmed färre rader kod att analysera. Vi valde därför att artificiellt inkludera alla HTML-mallar för AngularJS-artefakten vid våra mätningar för att få så rättvisande resultat som möjligt.

De relativa skillnader i förvaltningsbarhetsindex som uppstår när respektive kodbas byggs ut är dock inte beroende av någon jämförelse artefakterna emellan. Istället visar dessa värden hur den interna förvaltningsbarheten förändras vid underhållsarbete. Eftersom samma funktionalitet lades till för bägge artefakterna ger detta en indikation för hur snabbt komplexiteten växer för respektive artefakt. Då vi enbart utförde tre mätningar per artefakt är det dock svårt att extrapolera trenden som framkommer i våra resultat. Vidare forskning behöver göras kring kvalitet och förvaltningsbarhet av ensidesapplikationer innan några definitiva slutsatser kan dras.

Mätningen av förvaltningsbarhetsindex på de initiala artefakterna visar små skillnader med en viss fördel för AngularJS. En av de främsta orsakerna till det bättre förvaltningsbarhetsindexet är att AngularJS-artefakten krävde färre rader kod för att implementera alla funktionella krav i kravspecifikationen (se bilaga 1). Funktioner som att filtrera och sortera listor finns färdigt att använda i AngularJS medan de fick lov att implementeras på egen hand i React. Så långt är det alltså fördelaktigt att använda ett ramverk som kan utföra viss grundläggande funktionalitet.

Att utföra det första underhållet, som främst krävde en utökning av affärs- och presentationslogiken skiljde sig inte avsevärt mellan artefakterna. Förvaltningsbarhetsindex sjönk snarlikt (0,7 % respektive 0,6 %) i båda artefakterna. Med både AngularJS och React kunde befintlig kod på ett enkelt sätt struktureras om för att

tillåta hög grad av återanvändbarhet av befintliga komponenter för att tillgodose förändrade och utökade behov.

Underhåll 2, som krävde nya dataflöden mellan olika delar av artefakterna, visade större skillnader. React-artefakten minskade i förvaltningsbarhetsindex i snarlik mängd som vid underhåll 1 medan AngularJS-artefakten visade en kraftigare minskning. Detta trots att mängden ny kod var snarlik. En orsak är att den kod som behövde skrivas var mer komplex i AngularJS-artefakten. En ytterligare orsak till den kraftigare minskningen är att AngularJS-artefakten hade färre rader kod i sitt grundutförande jämfört med React-artefakten. Därför påverkades dess förvaltningsbarhetsindex i större utsträckning av ett likvärdigt antal rader ny kod. En annan skillnad mellan artefakterna är att det krävdes fler nya filer för AngularJS-artefakten samtidigt som ett större antal befintliga filer behövde modifieras. En anledning till att färre filer påverkades för React-artefakten är att React enbart har en typ av komponent, medan AngularJS har flera. I React kan en befintlig komponent enklare modifieras för att tillgodose nya behov.

Underhåll 2 är också där skillnader mellan MVVM och Flux träder fram. Med Angulars dataarkitektur finns det flera sätt att skicka data mellan olika delar av applikationen. Exempelvis kan en hierarki av vymodeller (controllers i AngularJS) skapas där man kan välja att skicka data antingen upp eller ner i hierarkin. I Flux kan data enbart färdas i en riktning, från en toppnivåkomponent ner till alla underkomponenter i applikationen. Att använda MVVM med AngularJS är enkelt eftersom det är så ramverket är konstruerat. Att implementera Flux med React kräver dock en större investering. Detta manifesterades dels i att React fick en högre initial komplexitet, och därmed ett lägre förvaltningsbarhetsindex, men även i att React-artefakten kunde hålla ett jämnare förvaltningsbarhetsindex mellan underhållen.

Bägge artefakterna erhöll dock mycket goda värden för förvaltningsbarhetsindex, även efter de båda underhållsarbetena, och höll därmed hög intern kvalitet. Våra genomsnittliga värden låg högt över 85, som är tröskelvärdet för god förvaltningsbarhet (Sjøberg, Anda, & Mockus, 2012, s. 108). Vårt kunskapsbidrag är dock något begränsat då vi utvecklade förhållandevis små applikationer. Mer forskning behövs kring utveckling och förvaltning av större ensidesapplikationer, över större tidsperioder.

I den sammansställning som Casteleyn, Garrig'os, & Maz'on (2014, s. 24) utförde anmärkte de på att en betydande andel artiklar bidrog med nya lösningsförslag medan få utförde någon empirisk validering eller utvärdering av befintliga lösningar. Detta, menar de, tyder på en omognad inom forskningsområdet. Ett annat problem de belyser är att verktyg och tekniker utvecklas i snabbare takt än vad forskningen bedrivs, vilket leder till att forskningens verksamhetsnytta blir begränsad. Bristen på analysverktyg fick vi praktiskt erfara under arbetets gång. För att underlätta vidare forskning behövs analysverktyg som är lättare att anpassa efter valda mätvärden och programspråk samt en vedertagen mätmetod för kvalitet för webbapplikationer. De verktyg som finns tillgängliga för analys av ensidesapplikationer idag är utvecklade av entusiaster snarare än forskare, vilket kan anses vara beklämmande för forskarsamfundet.

6 Slutsatser

Det avslutande kapitlet, i vilket vi ämnar besvara vår frågeställning: Hur påverkas en ensidesapplikations förvaltningsbarhet om den utvecklas med ett ramverk jämfört med bibliotek?

AngularJS uppmätte bättre värden för förvaltningsbarhetsindex än React i applikationens grundutförande såväl som efter två underhållsarbeten. Värdena för AngularJS sjönk dock snabbare när applikationerna byggdes ut. Vidare behövde fler filer modifieras för att utföra underhåll av AngularJS-applikationen, även när helt ny funktionalitet lades till.

Utifrån resultaten drar vi slutsatsen att användning av ett stort ramverk för att utveckla ensidesapplikationer kan leda till hög kodkvalitet och god förvaltningsbarhet så länge som den standardfunktionalitet ramverket tillhandahåller överensstämmer med applikationens funktionella krav. Om applikationen kräver funktionalitet som saknas i ramverket eller som skiljer sig från ramverkets förordnade implementering kan ramverket bli mer av ett hinder än en tillgång. Små bibliotek ger mindre färdig funktionalitet vilket resulterar i applikationer som har högre initial komplexitet och därmed lägre förvaltningsbarhet. Små bibliotek ger dock utvecklarna mer kontroll över funktionaliteten under förvaltningsarbeten, utan att förlita sig på de av ramverket föreskrivna sätten att lösa problem. När applikationer växer blir vinsten med att använda ramverk mindre medan den högre initiala komplexiteten av en applikation utvecklad med små bibliotek lönar sig mer.

I slutändan är de uppmätta skillnaderna små. Vi utvecklade applikationerna med vedertagna designmönster för dataflöden, efter sunda utvecklingsprinciper och med en genomtänkt utvecklingsmetod, vilket återspeglas av de mycket goda värdena för förvaltningsbarhetsindex bägge applikationerna uppmätte. Detta hantverksmässiga kodande har sannolikt en mer betydande effekt på förvaltningsbarheten än valet av ramverk eller bibliotek har, och det kan vara så att valet överhuvudtaget inte är särskilt betydande för en applikations förvaltningsbarhet. Mer forskning behöver göras inom ämnet, framförallt när det kommer till utveckling av analysverktyg och förvaltning av stora ensidesapplikationer över längre tidsperioder.

Litteraturförteckning

Airbnb. (den 24 April 2016). *JavaScript Style Guide*. Hämtat från Github: https://github.com/airbnb/javascript den 28 April 2016

Bakota, T., Hegedus, P., Körtvélyesi, P., Ferenc, R., & Gyimóthy, T. (2011). A Probabilistic Software Quality Model. *Software Maintenance (ICSM), 2011 27th IEEE International Conference* (ss. 243 - 252). Williamsburg, VI : IEEE.

Balaji, S., & Murugaiyan, S. M. (den 29 Juni 2012). Waterfall vs V-model vs Agile: A comparative study on SDLC. *International Journal of Information Technology and Business Management*, ss. 26 - 30.

Casteleyn, S., Garrig'os, I., & Maz'on, J.-N. (Juni 2014). Ten Years of Rich Internet Applications: A Systematic Mapping Study, and Beyond. *ACM Transactions on the Web (TWEB)*, ss. 1 - 46.

Coleman, D., & Ash, D. (1994). Using Metrics to Evaluate Software System Maintainability. *Computer, 27*(8).

David, G. A. (1984). What does "product quality" really mean? *MIT Sloan Management Review*, 25-45.

Deissenboeck, F., Juergens, E., Lochmann, K., & Stefan, W. (2009). Software Quality Models: Purposes, Usage Scenarios and Requirements. *Software Quality, 2009. WOSQ '09. ICSE Workshop on* (ss. 9 - 14). Vancouver, BC : IEEE.

Eriksson, U. (2008). *Test och Kvalitetssäkring av IT-system.* Lund: Studentlitteratur AB.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* Boston: Addison Wesly.

Glass, L. R. (2002). *Facts and Fallacies of Software Engineering.* Boston: Addison Wesley.

Granja-Alvarez, J. C., & Barranco-Garcia, M. J. (1997). A Method for Estimating Maintenance Cost in a Software Project: A Case Study. *Software Maintenace Research and Practice, 9*, 161-175.

Heitlager, I., Kupiers, T., & Visser, J. (2007). A Practical Model for Measuring Maintainability. *Quality of Information and Communications Technology, 2007*.

Hunt, A., & Thomas, D. (2000). *The Pragmatic Programmer: From Journeyman to Master.* Reading, Mass: Addison-Wesley Professional.

ISO/IEC. (den 01 04 2004). ISO/IEC 9126-1:2001. *Software engineering - Product quality - Part 1: Quality model.* ISO/IEC.

jQuery Foundation. (2016). *Getting Started with ESLint*. Hämtat från ESLint: http://eslint.org/docs/user-guide/getting-started den 28 April 2016

Kanellopoulos, Y., Panos, A., Antoniou, D., Makris, C., Theodoridis, E., Tjortjis, C., & Nikos, T. (2010). Code Quality Evaluation Using the ISO/IEC 9126 Standard. *International Journal of Software Engineering & App lications* , 17 - 36.

Kearney, J. K., Sedlmeyer, R. L., Thompson, W. B., Gray, M. A., & Adler, M. A. (November 1986). Software complexity measurement. *Communications of the ACM, vol 29 no 11*, ss. 1044-1050.

Kitchenham, B., & Pfleeger, S. L. (1 1996). Software Quality: The Elusive Target . *IEEE Software*, ss. 12 - 21.

Mardan, A. (2016). *React Quickly.* New York: Manning Publications Co.

Martin, R. C. (2003). *Agile Software Development: Principes, Patterns and Practices.* Upper Saddle River, NJ: Prentice Hall.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship.* Upper Saddle River, NJ: Prentice Hall.

Martin, R. C. (den 11 05 2014). *Framework Bound*. Hämtat från 8th Light: https://blog.8thlight.com/uncle-bob/2014/05/11/FrameworkBound.html

Maurya, S. L., & Shankar, G. (den 7 Juli 2012). Maintainability Assessment of Web Based Application. *Journal of Global Research in Computer Science*, ss. 30 - 33.

McCabe, T. J. (1976). A Compexity Measure. *IEEE Transactions on Software Engineering, VOL. SE-2, NO.4*.

McCabe, T. J., & Watson, A. H. (1996). *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.* Gaithersburg: National Institute of Standards and Technology.

McConnel, S. (2004). *Code Complete.* Redmond: Microsoft Press.

Mikowski, M. S., & Powell, J. C. (2014). *Single Page Web Applications.* New York: Manning Publications Co.

Oates, J. B. (2006). *Researching Information Systems and Computing.* London: SAGE Pulications Inc.

Ruebbelke, L. (2015). *AngularJS in Action.* Shelter Island: Manning Publications Co.

Shapiro, B. (den 16 April 2013). *Website vs. Web Application: What's the Difference?* Hämtat från Segue Technologies: http://www.seguetech.com/blog/graphic/2013/04/16/website-vs-web-application-difference den 14 April 2016

Shen, V. Y., Conte, S. D., & Dunsmore, H. E. (1983). Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support. *IEEE Transactions on Software Engineering.*

Sjøberg, D. I., Anda, B., & Mockus, A. (2012). Questioning Software Maintenance Metrics: a Comparative Case Study. *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '12* (ss. 107-110). New York: ACM.

Stilwell, J. (den 19 April 2016). *jared-stilwell/escomplex: Software complexity analysis of JavaScript-family abstract syntax trees.* Hämtat från https://github.com/jared-stilwell/escomplex

TodoMVC. (2016). Hämtat från TodoMVC: http://todomvc.com/ den 21 04 2016

Welker, K. D. (2001). The Software Maintainability Index Revisited. *Journal of Defense Software Engineering*.

Bilaga 1 – Kravspecifikation

Kravspecifikation webbshop

- Header
    - Logotyp
    - Navigation
    - Sökfält
    - Förhandsvisning av varukorg
- Produktsida
    - Lista alla produkter
    - Filtrera på namn
    - Sortera på pris – fallande och stigande
    - Sortera på varunamn – bokstavsordning och inverterad bokstavsordning
    - Köpknapp som lägger till vara i varukorg
    - Knapp för mer produktinformation
- Omsida
    - Kort företagsinformation
- Kontaktsida
    - Kontaktuppgifter
    - Kontaktformulär
        - Namn
        - Epost
        - Fritext
        - Validering
- Footer
    - Företagsnamn
    - E-post
- Varukorg
    - Visa produkt(er)
    - Ta bort produkt
    - Inkrementera kvantitet
    - Dekrementera kvantitet
    - Se varupris
    - Se radpris
    - Se totalsumma

Underhållsarbete 1

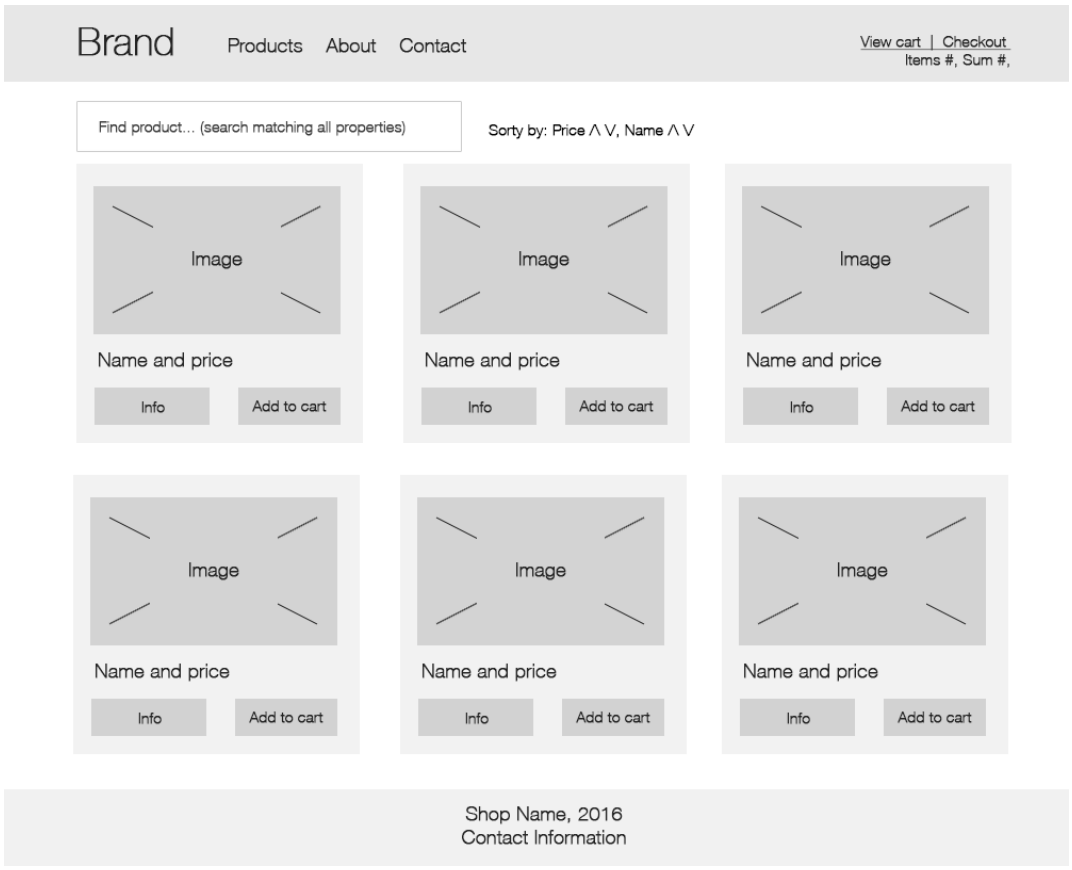
- Rekommendationsfunktionalitet
    - Föreslå och presentera produkter andra användare köpt baserat på vald produkt

Underhållsarbete 2

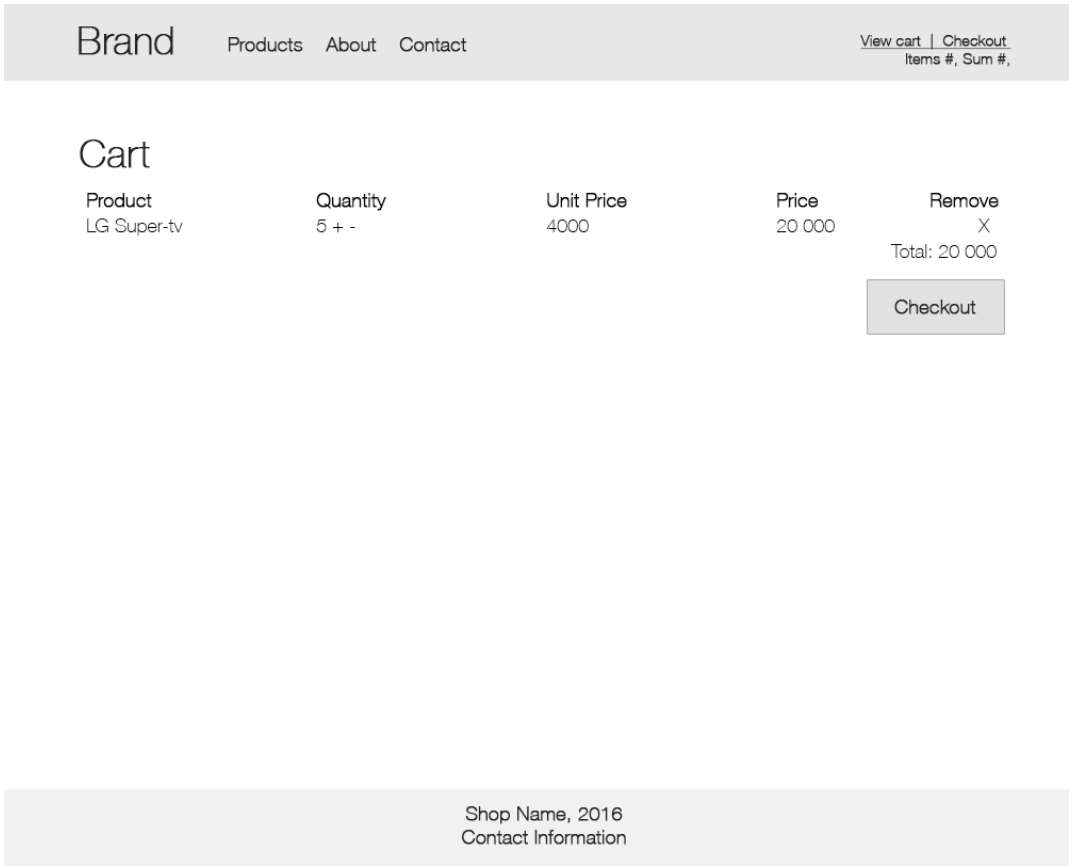
- Omsida
    - Karta
        - Alla butiker utmärkta
        - Närmsta butik centrerad
        - Vid klick visa butiksinfo utanför kartan
        - Vid klick visa butiksinfo i navigationsmeny

Bilaga 2 – Designmallar

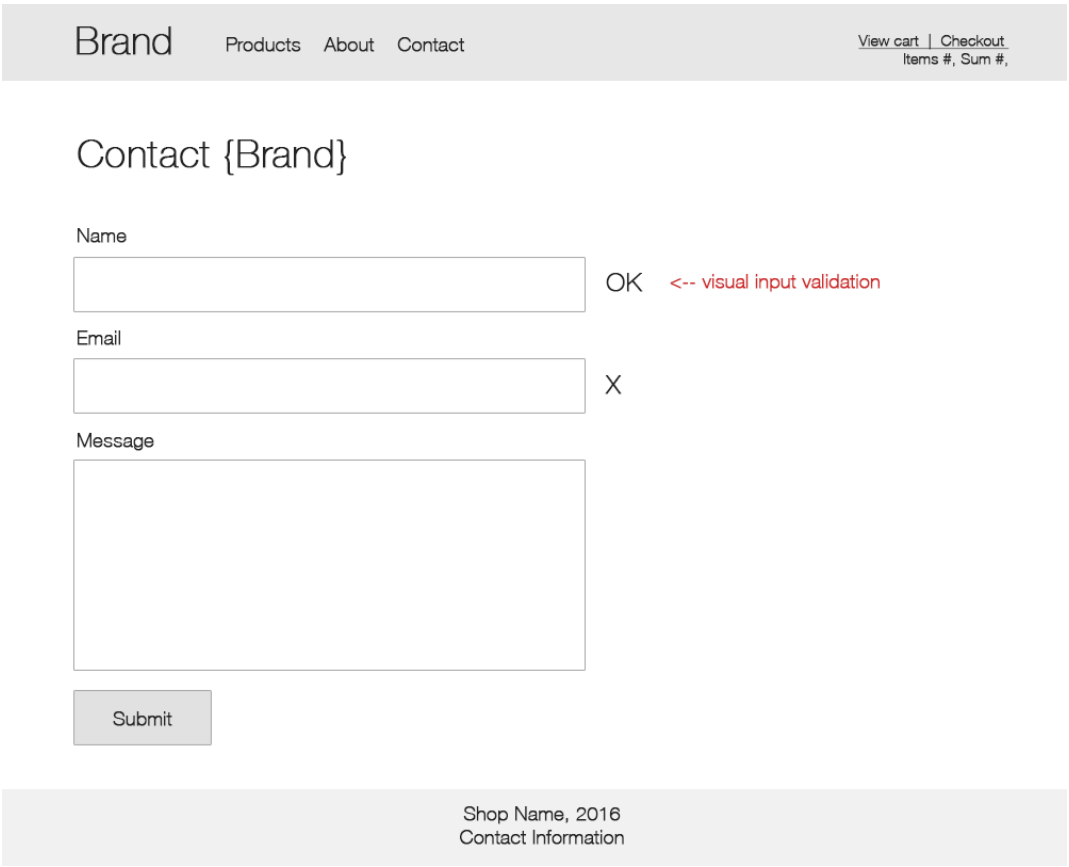
Produktlista



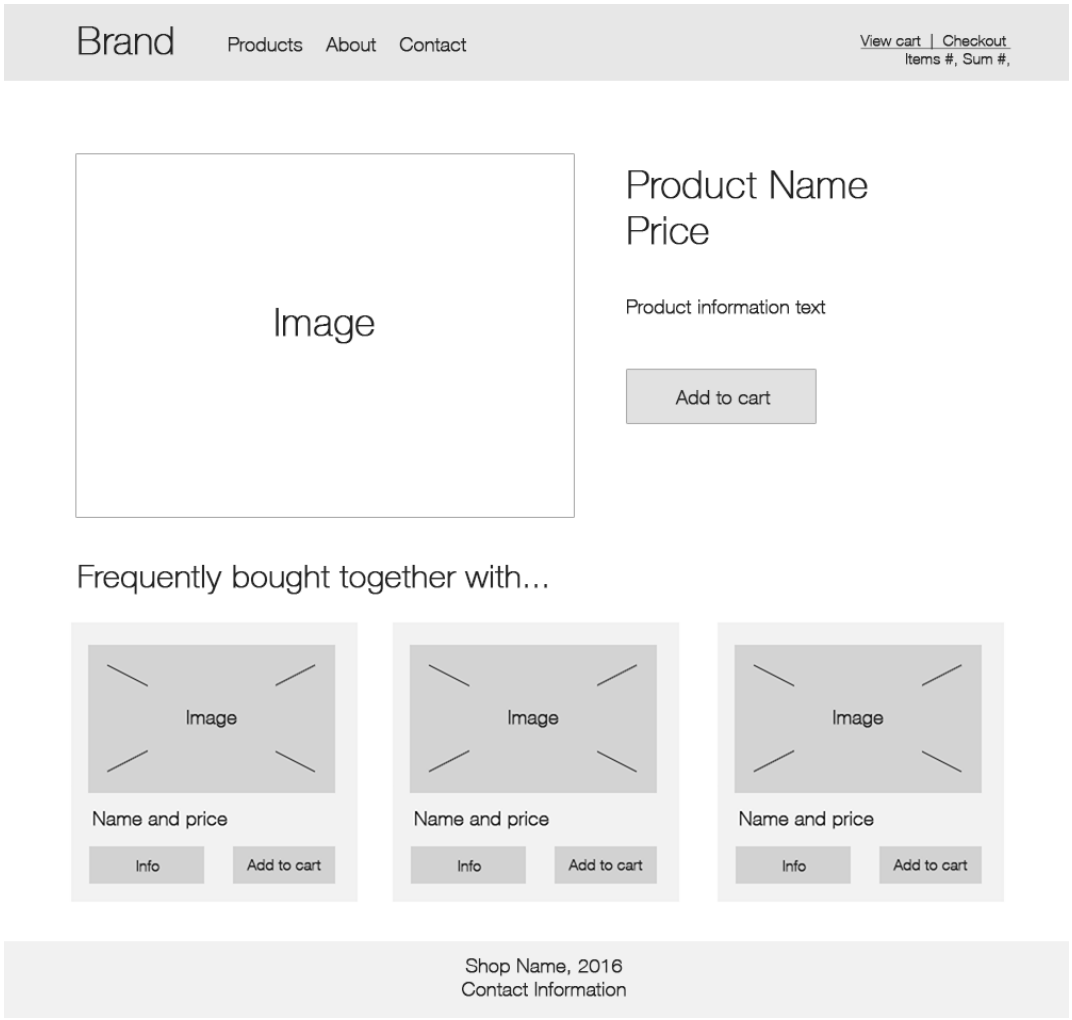
Varukorg


Brand
Products
About
Contact
View cart | Checkout
Items #, Sum #,
Cart
Product
LG Super-tv
Quantity
5 + -
Unit Price
4000
Price
20 000
Remove
X
Total: 20 000
Checkout
Shop Name, 2016
Contact Information

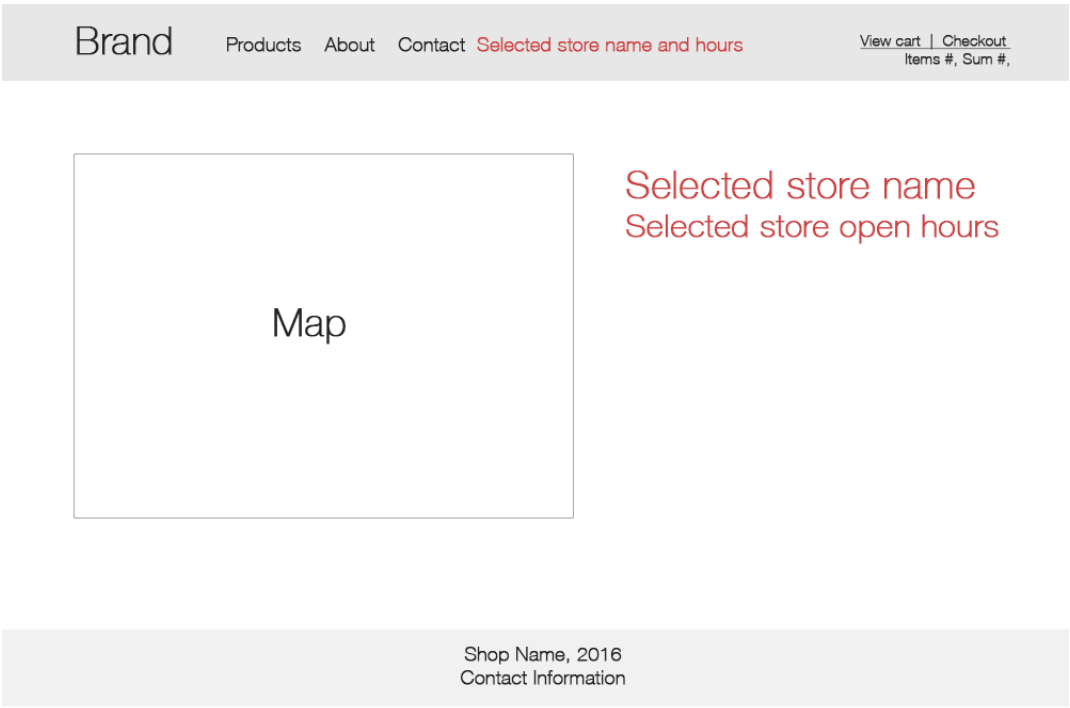
Kontaktinformation



Produktinformation



Om


Brand
Products
About
Contact
Selected store name and hours
View cart | Checkout
Items #, Sum #,
Map
Selected store name
Selected store open hours
Shop Name, 2016
Contact Information

Bilaga 3 – Programversioner

Namn	Version
JSCS	2.11.0
JSHint	2.9.1
Plato	1.5.0
AngularJS	1.5.5
Angular-ui-router	0.2.18
Jasmine	2.4.1
Karma	0.13.22
Babel-core	6.7.6
Babel-loader	6.2.4
Babel-preset-react	6.5.0
Complexity-report	2.0.0
Escomplex	2.0.0
Eslint-plugin-react	5.0.1
Flux	2.1.1
React	15.0.0
React-dom	15.0.0
React-router	2.0.1