Datavisualisering

AXEL ÖSTERMAN
BACHELOR´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Datavisualisering

Av

Axel Österman

Instutionen för Elektro- och informationsteknik Lunds Universitet

Sammanfattning

Detta arbete har utförts för företaget Verifyter AB i Lund. Deras huvudkunder är mikrochiptillverkare. Verifyter tillhandahåller ett vertyg vid namn PinDown som gör automatisk debugging för sina kunders testkörningar. Verktyget skapar en stor mängd data med information kring dessa testkörningar. Företaget vill kunna visualisera delar av denna information för sina kunder så att kunderna kan få bättre insyn i hur deras projekt utvecklas.

Rapporten går igenom allmänt vad visualisering är och varför det behövs. Baserat på detta har vi sedan valt de bästa visualiseringsteknikerna för de olika typer av data som Verifyter vill visualisera, vilket är ett linjediagram för sekventiell data samt en filtrerbar lista över annan data.

För att kunna implementera dessa diagram undersöktes vilket eller vilka bibliotek som lämpar sig bäst för just de typer av visualisering som skall utföras. För detta arbete lämpade sig D3.js i kombination med NVD3.js för att visualisera diagram medan List.js användes för att skapa den filtrerbara listan.

Arbetet resulterade i två olika visualiseringar, ett linjediagram och en lista. Linjediagrammet visar ett antal olika dataserier och hur de ändras över tid. Dessa dataserier är t.ex. projektkvalitet, projektkostnad och projektstatus. Listan som implementerades visar hur ofta filer förekommer i misslyckade tester. Denna lista blev väldigt lång och för att underlätta för användaren skapades funktioner för filtrering, sortering samt sökning. Den implementerade listan såväl som diagrammet uppfyller de krav och önskemål som Verifyter hade på produkten.

Nyckelord

Informationsvisualisering

Diagram

Lista/or

JavaScript

JSON

HTML

Abstract

This work was carried out for the company Verifyter AB in Lund. Their main customers are microchip manufacturers. Verifyter provides a tool called PinDown that does automatic debugging on their customers' test runs. The tool creates a large amount of data with information about these test runs. The company wants to be able to visualize this data for their customers in order for their customers to be able to get better insight in how their projects are developing.

This work broadly brings up what visualization is and why it is needed. We have then slected the best visualization techniques for the different kind of data that Verifyter wants to visualize, which is a line chart for sequential data and a filterable list for other types of data.

To implement these charts different libraries were investigated to find the most suitable library for the visualizations that was to be impelemented. For this work D3.js in conjunction with NVD3.js was the best library to visualize the line chart while List.js was used to create the filterable list.

This resulted in two different visualizations, a line chart and a list. The line chart visualizes a range of different data series and how they change over time. Some of these data series are for example project cost, project quality and project status. The list that was implemented shows how often files are occurring in failed tests. This list became very long and in order to make it easier for the user functions for filtering, sorting and seach were implemented. The implemented list as well as the chart meet the requirements and desires that Verifyter had on the product.

Keywords

Information visualization

Chart(s)

List(s)

JavaScript

JSON

HTML

Innehållsförteckning

1. Inledning

1.1. Bakgrund

Verifyter AB är ett litet startupföretag i Lund med bara två anställda utvecklare. Deras kunder är mikrochiptillverkare som främst finns i USA. Verifyter utvecklar ett automatiskt debuggingprogram vid namn PinDown som kör kundernas testsviter. Dessa körningar genererar en stor mängd testresultat och information som lagras i en databas. Företaget vill kunna visualisera denna data för kunden så att kunden tydligt kan se projektstatus, projektkostnad samt produktkvalitet.

Den data som lagras är information om var i koden tester fallerar. Den visar exakt vilka filer som misslyckas i tester samt vem som skrev koden som får testet att misslyckas. Datan kan också visa hur många misslyckade exekveringar en viss fil ger upphov till och på så sätt kan man se vilka filer som innehåller flest felkällor och vilka som misslyckas i flest test.

I dagsläget får företagets kunder delar av denna information automatiskt skickad till sig via mail när PinDown hittar fel. Dessa mail skickas till den individ som verktyget ser som ansvarig för felet, oftast den som har skrivit de felaktiga kodraderna.

Problemet med detta är att det kan vara svårt att få en överblick över kundernas projekt. Implementerade fel blir visserligen korrigerade fort med hjälp av verktyget men man kan ju tänka sig att användaren vill se projektens status över en tidsperiod eller se statistik över vilka filer eller vilken utvecklare som står för flest misstag.

För att tillhandahålla en bättre helhetsbild över kundernas projekt vill Verifyter kunna visualisera data i olika typer av diagram. Detta kommer att medföra att kunderna enklare kan se i vilka filer fel uppkommer och få en bättre överblick över projektens status. PinDown kommer fortfarande att fungera på samma sätt som tidigare och visualiseringen av datan har egentligen inget med verktyget att göra mer än att det är PinDown som genererar datan.

För tillfället har Verifyter ett program/webbapplikation som heter TestHub. Detta program fungerar som en utvecklingsplattform för Verifyter där de testar funktionalitet som de vill lägga till den hemsida som deras kunder har tillgång till. Det är i TestHub som Verifyter vill utveckla funktionalitet för att visualisera data från PinDowns testkörningar. På så sätt kan Verifyter få en känsla av hur visualiseringen kan fungera innan den läggs till i den applikation som kunderna har tillgång till.

1.2. Syfte

Genom att presentera datan från testkörningarna kan Verifyters kunder bättre se var i deras kod som fel uppkommer samt hur projektstatus, projektkostnad och produktkvalitet varierar över tiden. På så sätt kan kunderna se hur PinDown effektiviserar testkörningarna vilket ger ökat förtroende för verktyget.

1.3. Målformulering

Målen var att implementera åtminstone ett diagram och en filtrerbar lista för att visualisera data från testkörningar med PinDown. Det skulle vidare ges förslag till förbättringar hur Verifyters databas bör vara utformad så att det snabbare och effektivare går att visualisera datan.

För att kunna implementera visualisering av relevant data behöver man först ta redan på vilka grafer, diagram och listor som bör användas. Verifyter gav förslag på att utveckla ett linjediagram för att visualisera projektstatus etc. över tid samt att utveckla en filtrerbar lista för andra typer av data.

Det bästa hade varit att ta fram vilka grafer och diagram som skall utvecklas i samarbete med Verifyters kunder då det är de som kommer att vara slutanvändarna av produkten. Tyvärr är Verifyters kunder större företag i USA, så att få någon kontakt med dem är svårt, och de är i nuläget inte involverade i utvecklingen av produkten.

Anledningen till att det inte kändes angeläget att ha kontakt med Verifyters kunder är att det inte är en specifik kund som vill ha

arbetet utfört. Att visualisera datan är en idé från Verifyter, vilket innebär att företagets kunder inte riktigt har några krav på produkten då de inte vet om att den utvecklas. Trots att det är Verifyters kunder som är slutanvändarna har de inget att säga till om gällande utvecklingen och designen av produkten.

För att säkerställa att de diagram som Verifyter föreslagit faktiskt var de bästa diagrammen för deras data gjordes det en litteraturstudie över olika typer av diagram. Denna studie gick igenom vad för olika typer av diagram man kan använda sig av samt vilka diagram som passar till olika typer av data. Studien tog också upp generellt vad visualisering av data innebär, varför det är viktigt och hur man kan validera att man har valt rätt visualiseringsteknik. Den tog också upp hur man bör tänka och gå till väga när man skapar en visualisering av data.

Vid val av bibliotek togs en lista över krav på biblioteken fram. Dessa krav kommer ifrån den tidigare kravställningen på vilka diagram som skall tas fram. På så sätt kan biblioteken utvärderas utifrån de diagram som skall tas fram. Utifrån olika artiklar som listar olika bibliotek samt de krav som fanns på diagrammen kunde de potentiella biblioteken minska i antal fram tills bara ett fåtal potentiella bibliotek kvarstod. För de kvarvarande biblioteken gjordes en mer detaljerad jämförelse av för- och nackdelar mellan dem för att sedan få fram det bäst lämpade biblioteket för visualisering av data i detta projektet.

Efter att ha valt visualiseringstekniker samt vilka bibliotek som skulle användas kunde diagram implementeras med den data som Verifyter ville visualisera. Med de studier som genomförts finns det underlag för framtida utvecklingsmöjligheter av andra diagram som visualiserar annan data.

För att ge kommentarer på databasstrukturen hölls en dialog med Verifyter om hur datan kan sparas på ett bättre sätt för att minska användandet av Java vid databehandlingen. Dessa kommentarer presenteras i kapitel 6, Slutsats, då förändringarna inte implementerades i detta projekt.

1.4. Problemformulering

Utifrån syftet och målformuleringen kan följande huvudproblem fastställas:

- Vilken typ av diagram passar bäst till de olika typerna av data och visualisering av dessa?
- Vilket/vilka bibliotek lämpar sig bäst för att implementera visualiseringarna?
- Hur bör ett grafiskt gränssnitt vara utformat för att visa projektstatus, projektkostnad samt produktkvalitet på ett optimalt vis?
- Hur visualiserar man *projektstatus*, *projektkostnad* samt *produktkvalitet* i samma diagram?
- Hur ska en filtrerbar lista implementeras för att visa data som bättre visas i en lista än i ett diagram?
- Hur bör databasen vara utformad för att prestandan skall bli så bra som möjligt?

1.5. Motivering till examensarbetet

När PinDown automatiskt analyserar testfel samlar verktyget på sig en mängd data. I det här projektet vill Verifyter undersöka om det går att presentera denna data på ett sätt som tillför kunderna värdefull information om projektstatus samt kunskap om vilka delar av kundernas system som har kvalitetsproblem. Om detta faller väl ut kommer det vara intressant för alla mjuk- och hårdvaruprojekt eftersom man då bättre kan planera både nuvarande och framtida projekt.

1.6. Avgränsningar

Rapporten omfattar en analys över vilka visualiseringsteknik som är bäst för Verifyters data samt vilka bibliotek som lämpar sig bäst för att implementera dessa diagram. Dessa diagram har sedan implementerats.

I examensarbetet ingick det inte att utveckla en hemsida/webbapplikation från grunden. Istället skulle examensarbetet bygga på ett redan existerande system, TestHub, och bara lägga till ytterligare funktionalitet till webbapplikationen.

Det ingick inte att ändra i den existerande databasens struktur. I examensarbetet ingick det bara att föreslå förbättringar, de behövde inte implementeras. Dessa förbättringar handlar till största del om prestandan för databasen.

2. Teknisk bakgrund

TestHub är utvecklat med ett ramverk vid namn Play! Framework. TestHub har också ett antal olika bibliotek som används för diverse funktionaliteter så som diagramverktyg, tabellverktyg mm. Systemet fungerar ungefär som en singlepage hemsida. Detta innebär att alla element och all funktionalitet som finns på hemsidan laddas in direkt när programmet körs. De delar av programmet som inte ska visas på en specifik flik av hemsidan döljs tills användaren trycker på den knapp som är bunden till en annan flik.

Detta gör att webbapplikationen kan ta lite tid att ladda första gången den körs då den måste ladda in alla JavaScript. När man väl har laddat in allt är hemsidan mycket snabb då den inte behöver gå till olika URL:er för att visa olika delar av hemsidan.

Kärnan i programmet är byggt med MVC-arkitektur. Vyn innehåller en HTML-fil "main.scala.html" som kör alla de JavaScript- och CSSfiler man har i programmet. Ifall man lägger till ett nytt JavaScript eller en ny CSS-fil måste dess sökväg läggas till i denna HTML-fil så att filen körs när programmet startas.

En annan viktig HTML-fil i vyn är "index.scala.html". Denna fil används i princip för all html i programmet. Vill man lägga till en knapp, en bild etc. så är det i denna fil man lägger till den.

Modellen innehåller Javaklasser som till största del används för att representera olika data. T.ex. finns det en Javaklass för att representera ett projekt. Ifall man vill visa någon data är det lämpligt att skapa en passande klass i modellen för att representera datan.

Det är dock inte bara klasser för representation av objekt som finns i modellen. I modellen kan man lägga i princip vilka Javaklasser man vill exempelvis hjälpklasser. I modellen finns också Databaseklassen. Det är Database som hanterar all databaskommunikation och ifall man vill lägga till en ny SQL-fråga för att få ut ny data lägger man till en ny metod i Database som behandlar SQL-frågan.

Kontrolldelen innehåller bara en Javaklass, Application. Application innehåller de metoder som tar data från Database och sedan formaterar den till så kallade JSON-strängar så att

JavaScripten sedan kan läsa och tolka datan. JSON är ett format som är enkelt för människor att läsa medan det samtidigt är enkelt för maskiner att gå igenom.

För att datan skall kunna propagera från Application till något JavaScript finns det en fil, "routes", i programmet. Denna fil definierar de "vägar" eller routes som programmet kan använda för att komma åt data eller redigera data. Varje route definierar först en HTTP-metod, dessa kan vara t.ex. GET (komma år data), PUT (lägga till data), POST (uppdatera data). Efter HTTP-metoden definieras den path som metoden skall ha samt vilken metod från Application som sökvägen skall använda sig av. Detta är väldigt likt hur sökvägar fungerar på en vanlig hemsida och om man skulle gå till t.ex. `localhost:9000/getAllProjects` skulle man få ut den JSON-sträng som Applications returnerar i metoden för `getAllProjects()`.

Sammanfattning av hur man får fram ny data för visualisering i TestHub:

- Om det behövs, skapa en klass i modellen för att representera data
- Skapa en metod i Database för att få ut data
- Skapa en metod i Application för att formatera data
- Lägg till en sökväg i routes så att JavaScript kan nå data

En annan mycket viktigt del av programmet är JavaScript och CSS. CSS-filen används som för alla hemsidor till att ändra utseendet på element på hemsidan. Här kan man t.ex. välja vilka färger element skall ha, hur stora de skall vara etc. JavaScript används för att hantera och visualisera data på hemsidan. JavaScript används också för interaktivitet på hemsidan som t.ex. mouseover, drag-and-drop mm. JavaScripten i TestHub används för att göra hemsidan till singlepage. Genom att binda olika JavaScript till olika knappar kan man visa och dölja olika element beroende på vad man specificerar i de olika JavaScripten. T.ex. om man klickar på knappen för att visa alla projekt har JavaScriptet för projekt specificerat att om knappen trycks skall htmltaggen för projekt visas och de andra elementen skall döljas. På så sätt är det genom JavaScript som man ger funktionalitet till hemsidan. För detta projekt är JavaScriptdelen av TestHub mycket viktig då det är i JavaScript som diagram kommer att implementeras.

3. Metod

Arbetet har utförts i två faser. I första fasen undersöktes vad informationsvisualisering är samt vad som faktiskt ska implementeras. I den andra fasen användes informationen som tagits fram i fas ett för att implementera prototypen. På så sätt är arbetet uppdelat i en större teoretisk del samt en lite mindre implementationsdel som tas upp i kapitel 4, Analys.

Arbetet har följt följande arbetsgång:

- Vad är visuell design och hur validerar man det?
- Ta fram de mest lämpliga diagrammen.
- Ta fram vilka bibliotek som skall användas.
- Sätta sig in i hur TestHub fungerar.
- Lägg till knapp för sida samt JavaScriptfil till knappen.
- Importera bibliotek.
- Lägg till exempel på diagram/tabeller.
- Lägg till knappar på sidan för att gå mellan olika diagram.
- Lägg till databasen i systemet.
- Hur gör man med gammal/ny databas?
- Vad för data ska visas? (Sker iterativt för varje dataserie).
    - Lägg till metoder i Database(Two) för att få fram data.
    - Lägg till klasser för att representera data (cost, quality, etc.).
    - Lägg till metoder i Application för att göra data till JSON.
    - Lägg till routes för metoderna för att få ut data.
- Åtgärda diagram så att de ser bra ut.
    - Knappar, axeletiketter, färger, zoomknappar.
- Implementera lista.
    - Knappar, sökfunktion, filterfunktion.

3.1. Design av informationsvisualisering

En stor bit av detta examensarbete handlade om visualisering av data. Arbetet med vilket typ av visualisering som skulle användas gjordes till stor del med hjälp av litteraturstudier samt diskussioner med Verifyter. Detta kapitel kommer att ta upp vad informationsvisualisering är samt de val som har tagits gällande visualisering. Dessa val och slutsatser användes sedan vid utvecklingen av produkten.

3.1.1. Vad är visualisering

Visualisering handlar om hur och varför man ska visa sin data för någon användare. I Visualization Analysis & Design dedikerar Munzner hela första kapitlet till just denna fråga, "What's visualization, and why do it?" [1]. Författaren sammanfattar det som:

> *"Computer-based visualization systems provide visual representations of dataset designed to help people carry out tasks more effectively", Munzner [1].*

Allt fler jobb blir automatiserade och utförda av datorer. Även om man har en väldigt avancerad visualisering av data kan en människa inte utföra vissa arbetsuppgifter tillräckligt snabbt. Det finns förstås områden där arbetsuppgiften inte är specificerad på ett sådant sätt att en maskin kan utföra arbetet. I sådana fall kan visualisering av data underlätta förståelse.

Trots detta behövs datorer när det gäller visualisering. Det är inte realistiskt att en människa ska rita ut stora datamängder med penna och papper eller för hand på en datorskärm. När man har ett litet antal mätpunkter fungerar det att en människa bearbetar datan, men när datamängderna blir stora blir det betydligt snabbare att använda datorer för att visualisera data.

Visualisering hjälper en att se hur data hänger ihop. Att bara se värden i en tabell ger inte samma intryck som att se värdena utritade i en graf. Detta problem tar Munzner upp genom s.k. Anscombe's Quartet och kan ses i Tabell I samt Fig. 1 [1]. Här har fyra mindre dataset tagits fram. Dessa fyra dataset har olika värden i sig men

deras medelvärde, samt varians är densamma. När man sedan har gjort en linjär regression över dessa värden får de fyra dataseten samma R-värde. Det visualiseringen vill visa är att trots att data kan se väldigt lika ut, betyder det inte att de är lika.

TABELL I. DATA FRÅN ANSCOMBE'S QUARTET. ALLA DATASET HAR SAMMA MEDELVÄRDE OCH VARIANS. TABELL TAGEN FRÅN MUNZNER [1].

	1		2		3		4	
	X	Y	X	Y	X	Y	X	Y
	10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
	8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
	13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
	9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
	11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
	14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
	6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
	4.0	4.26	4.0	3.10	4.0	5.39	19	12.5
	12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
	7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
	5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89
Mean	9	7.5	9	7.5	9	7.5	9	7.5
Variance	10	3.75	10	3.75	10	3.75	10	3.75
Correlation	0.816		0.816		0.816		0.816	

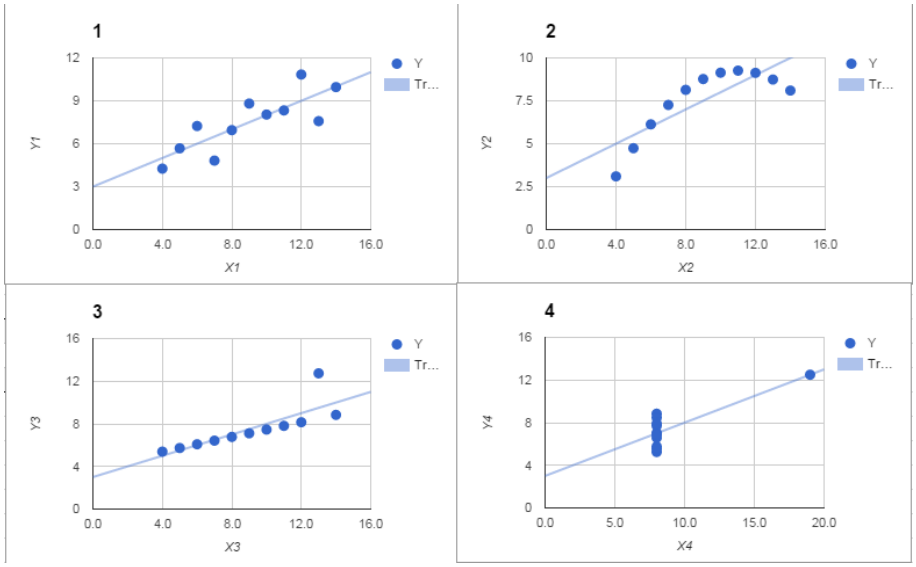

Fig. 1. Anscombe's Quartet utritad i grafer. De fyra dataset:en har samma R-värde med en linjär regression trots att punkterna är placerade så olika. Data för graferna kommer från Tabell 1.

Alla visualiseringstekniker passar inte till alla typer av data. Det är därför viktigt att ta reda på vad man har för data och vad man vill få ut av den innan man bestämmer sig för vilken teknik eller metod som man tänker använda.

Det svåra med visualisering är inte alltid att hitta rätt visualiseringsteknik, utan det svåra är att validera att man har valt rätt. Det finns en massa frågor som kommer upp när man försöker validera valet av visualisering. Hur vet man att man har valt rätt? Hur vet man att det fungerar som det är tänkt? Är det bättre än att göra samma sak manuellt? Munzner tar upp ytterligare frågor men dessa har mindre relevans för det här arbetet [1].

3.1.2. Validering av visualisering

Validering handlar i princip om att kolla så att man har utvecklat rätt produkt utifrån de krav som ställs på produkten.

I kapitel 4 i Visualization Analysis & Design tar Munzner upp varför man bör validera och hur man skall gå tillväga [1]. När det gäller visualisering finns det många olika tekniker och metoder och

många av dem är mindre bra än andra beroende på vilken data man har och vad man vill göra med datan.

Munzner tar upp fyra nästlade nivåer av validering när det gäller visualisering. De nästlade nivåerna för validering går att se i Fig. 2.

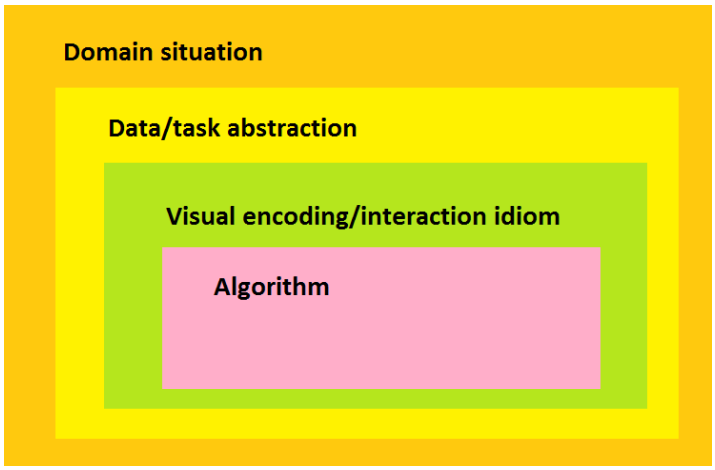

Fig. 2. De fyra nästlade nivåerna för validering av visualisering specificerat av Munzner [1].

Den yttersta nivån *Domain Situation* handlar om vad användaren skall använda produkten till. Det är viktigt att ta reda på vad användaren vill få ut av systemet. Det kan dock vara svårt för användaren att förklara vad den vill få ut av produkten. Användarna vet att de vill kunna se sin data men de vet inte alltid hur eller varför. En annan svårighet är att man har olika vokabulär inom olika domäner och på så sätt kan det bli svårt att förmedla vad man är ute efter.

I *Data/task abstraction* kan man abstrahera så att de olika orden från domänerna får en mer lik betydelse. Två helt olika målbilder kan abstraheras till att vara ganska lika. Dessa kan vara t.ex. att jämföra data eller lägga samman data.

Visual encoding/interaction idiom handlar om vad för visualiseringsteknik man använder. Utifrån den abstraktion man gjort i den tidigare nivån kan man lättare se vilken typ av diagram eller liknande som man bör använda sig av.

Den sista nivån *Algorithm* handlar just om de algoritmval man har gör. Denna nivå handlar mycket om hur snabb och minneskrävande algoritm(er) man har valt.

För dessa fyra nivåer har författaren specificerat ett hot mot validiteten för de olika nivåerna.

- *Domain situation*: Man misstolkade användarnas behov.
- *Data/task abstraction*: Man visar användarna fel sak.
- *Visual encoding/interaction idiom*: Visualiseringen fungerar inte.
- *Algorithm*: Implementationen är långsam.

Med dessa nivåer och hot i åtanke har författaren satt upp olika strategier för hur man kan gå tillväga för att validera de olika nivåerna. Författaren menar också att många av dessa hot är nästlade och man kan inte validera en nivå utan att ha validerat eller implementerat nivåerna under. Bara för att det finns fyra nivåer för validering innebär det inte att alla nivåer är applicerbara för alla projekt och alla nivåer kan och bör inte valideras samtidigt.

För den översta nivån kan man som validering intervjua slutanvändarna, eller så gott som slutanvändarna, för att ta reda på vad det är man ska ta fram eller om man har tagit fram rätt. Man kan också utföra fältstudier där man observerar hur användarna använder det nya systemet. Man kan på så sätt se ifall det nya systemet löser de problem som användarna hade samt om man har utvecklat rätt produkt.

I abstraktionsnivån uppstår problem ifall man har abstraherat fel och på sätt utvecklat något som inte löser den kravställning som användaren har. De viktiga här är att man testar produkten mot användarna med riktiga uppgifter istället för att testa med abstraherade uppgifter.

Hotet mot *Visual encoding/interaction idiom* är att den teknik man har valt för visualisering inte är tillräckligt bra. För att komma runt detta hot bör man noga gå igenom de olika val man har och inte ta för snabba beslut. Man kan också göra prototyper, på olika nivåer, på några av de bättre valen man har. På så sätt kan man testa sina

designval och få fram information som kan peka på vilken visualiseringsteknik som är bäst lämpad för ändamålet.

För att lösa hoten mot *Algorithm* måste man testa och analysera sin kod för att se hur den presterar. Användarna har förväntningar på hur ens produkt skall prestera. Ifall ens kod inte presterar på den nivå som användarna förväntar sig får man optimera sin kod eller hitta andra lösningar.

Applicerar man validering på det här arbetet kan man se att validering kring de två lägsta nivåerna har till viss del gjorts under arbetes gång. Undersökningen om vilka bibliotek som är bäst liknar den lägsta nivån *Algorithm*. Det är inte mycket man kan göra för att förbättra biblioteken i sig men genom att se till att man har valt rätt bibliotek har man samtidigt validerat *Algorithm*.

Undersökningen om olika diagramtyper kan ses som validering för nivån *Visual encoding/interaction idiom*. Efter att har undersökt vilka diagram som är bäst för olika typer av data kan man komma fram till det diagram som passar bäst för en specifik typ av data.

Det som kvarstår är då validering av de två översta nivåerna Domain situation och Data/task abstraction. När det gäller abstraktion finns det inte mycket att validera. De uppgifter som visualiseringen skall utföra är redan till viss del abstraherade av Verifyter.

Validering har på så sätt skett i dialog med Verifyter där ge har haft möjlighet att ge feedback på implementationen. Denna feedback har säkerställt att det som implementerades är rätt produkt och att visualiseringen kommer att vara användbar.

3.1.3. Olika typer av diagram för stora datamängder

När man behandlar och ska visualisera stora datamängder även kallat Big Data är det viktigt att tänka på vilka typer av diagram och grafer man använder sig av. Eftersom de stora datamängderna innehåller så många datapunkter kan vissa diagram bli väldigt osammanhängande. Andra diagram kan bli mindre osammanhängande men kan behövas renderas om varje gång en ny datapunkt införs. Man vill gärna undvika att rendera om grafer och diagram som innehåller stora mängder datapunkter då det kommer att ta mycket resurser från datorn.

Gorodov & Gubarev tar upp olika diagramtyper för att visualisera Big Data [2]. Författarna utgår från tre kriterier när de utvärderar diagrammen. Dessa kriterier handlar om vad diagrammen klarar av angående Big Data. Kriterierna är följande:

> *"large data volume, data variety and data dynamics", Gorodov & Gubarev [2].*

Det första kriteriet är ganska självförklarande, om ett diagram skall hantera Big Data måste det såklart kunna hantera stora datamängder. Data variety innebär att man skall kunna visa flera typer av data i diagrammet. Data dynamics innebär att man skall kunna ändra hur datan visas genom att t.ex. sätta fokus på olika punkter eller värden.

Författarna har också specificerat tre olika typer av Big Data och olika diagram är olika bra på att hantera de olika typerna av data. De tre typerna av data som specificeras är:

> *"large volume of data (Volume), multiformat data presentation (Variety), and high data processing speed (Velocity)", Gorodov & Gubarev [2].*

Utifrån dessa kriterier och typer av Big Data har författarna tagit fram två tabeller, Tabell II och III, över några olika diagram. Tabellerna visar vilka av de specificerade kriterierna som de olika diagrammen uppfyller.

TABELL II. OLIKA DIAGRAM UTIFRÅN DE KRITERIERNA LARGE DATA VOLUME, DATA VARIETY OCH DATA DYNAMICS. TABELL TAGEN FRÅN GORODOV & GUBAREV [2].

	Large data volume	Data variety	Data dynamics
Treemap	+	-	-
Circle packing	+	-	-
Sunburst	+	-	+

Cicrular network diagram	+	+	-
Parallel coordinates	+	+	+
Streamgraph	+	-	+

TABELL III. VILKEN TYP AV DATA SOM DE OLIKA GRAFERNA KAN HANTERA UTIFRÅN DE SPECIFICERADE BIG DATATYPERNA VOLUME, VARIETY OCH VELOCITY. TABELL TAGEN FRÅN GORODOV & GUBAREV [2].

Method name	Big data class
Treemap	Can be applied only to hierarchical data
Circle packing	Can be applied only to hierarchical data
Sunburst	Volume + Velocity
Circular network diagram	Volume + Variety
Parallel coordinates	Volume + Velocity + Variety
Streamgraph	Volume + Velocity

En annan artikel, skriven av Krstajic & Keim, tar upp hur olika diagram ändras när data ändras [3]. Artikeln tar också upp hur uppfattningen av datan förändras när diagrammen ändras. Ett exempel som tas upp är Treemap. Ifall man lägger till mycket data och rutorna ändras mycket i storlek kan uppfattningen av datan förändras. I artikeln tar författarna upp vad som kan ändras i ett antal vanliga diagram samt hur diagrammen kan ändras för att man ska tappa sammanhanget för diagrammet.

Utifrån dessa två artiklar är de bara två diagram som båda artiklarna tar upp, treemap och streamgraph. Några andra grafer som tas upp individuellt i de två artiklarna kan dock vara intressanta att fokusera mer på än andra.

Treemap

Båda artiklarna får treemap att se relativt dåligt ut som val för att visualisera Big Data. Treemap kan hantera stora mängder data men är dålig på att hantera flera typer av data. Man kan i princip bara visualisera data på två sätt med treemap, storlek på rutorna samt färgen på rutorna. Krstajic & Keim menar att om man ändrar för mycket med storlek och färg på rutorna kan uppfattningen av datan ändras vilket man vill undvika [3]. Det är också viktigt att påpeka att treemap nästan bara kan visualisera hierarkisk data vilket är bra om den data man har är hierarkisk. Treemap är dock nästan oanvändbar om man inte har hierarkisk data. Exempel på treemap finns i Fig. 3.

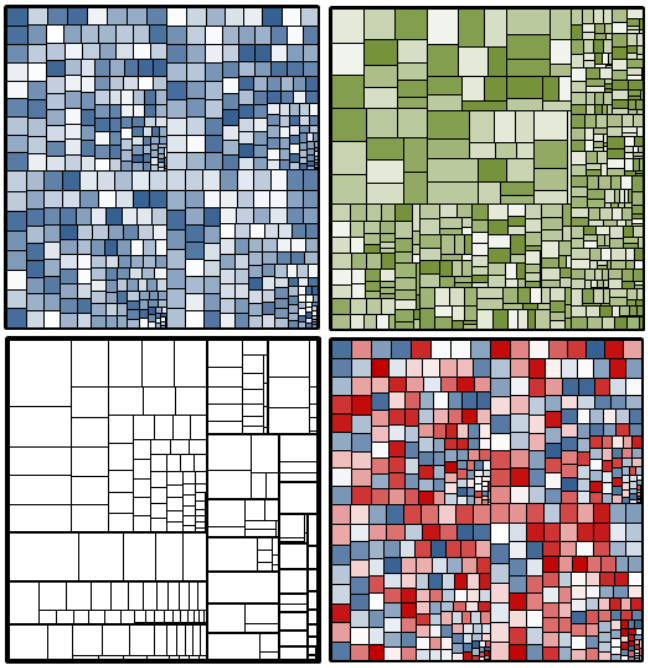

Fig. 3. Exempel på hur en treemap kan se ut. Bild tagen från FPPT [4].

Streamgraph

Det andra diagrammet som båda artiklarna tar upp är streamgraph. Denna typ av diagram ger intryck av att vara bättre än treemap enligt

båda artiklarna. Streamgraph är till för att visa data över tid för olika dataserier men samtidigt också visa summan av värdena

Det streamgraph fallerar på enligt Gorodov & Gubarev är data variety [2]. Detta då streamgraph bara kan visualisera ett antal olika dataserier innan det blir för plottrigt i grafen. Det man kan ändra med streamgraph enligt Krstajic & Keim är intervallen på axlarna, man kan ändra ordningen på "strömmarna" samt lägga till och ta bort strömmar [3]. Problem uppstår dock när man ändrar ordningen på strömmarna då man får en helt annan uppfattning av diagrammet. Ett exempel på en streamgraph finns i Fig. 4.

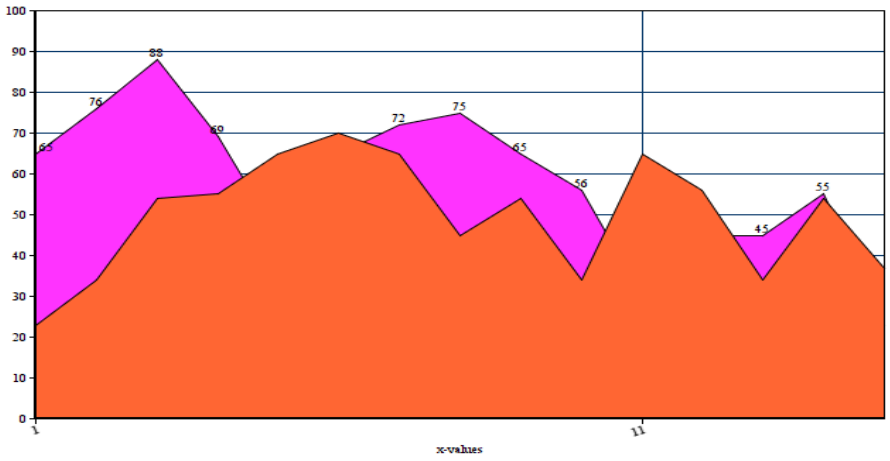

Fig. 4. Mycket enkel streamgraph med bara 2 strömmar. Grafen har också icke-negativa värden vilket kan förekomma vid denna typ av diagram.

Linjediagram

Linjediagram är en av de diagramtyper som tas upp av Krstajic & Keim [3]. Enligt författarna kan linjediagram ändras i två aspekter. Man kan ändra intervallen för axlarna samt lägga till och ta bort linjer. Problem med linjediagram uppkommer ifall man lägger till data utanför x- och y-axlarna. Om detta sker kan man hantera det på två sätt. Man kan ändra maxvärdet för axlarna för att få plats med den nya datan. Problemet med detta är att den redan existerande datan blir flyttad och man kan tappa sammanhanget för grafen. Det andra sättet man kan lösa detta problem på är att flytta datan på axlarna. T.ex. om man har en graf som visar hur något förändras över tid och man

lägger till nytt värde vid x-max, istället för att öka x-max tar man bort det äldsta värdet och på så sätt har lika många mätvärden i grafen.

Ett annat problem som författarna tar upp gällande linjediagram är att det kan bli rörigt om har för många mätserier i ett diagram. Detta är lätt att lösa genom att se till att inte lägga till för många linjer i grafen.

Det som är bra med linjediagram är att de är lätta för användaren att förstå. Man kan lätt se hur de olika datapunkterna skiljer sig från varandra i en dataserie men också mellan olika dataserier. Ifall man har två värden där man vill se hur det ena värdet förändras beroende på det andra värdet är linjediagram mycket bra och enkla att implementera. Exempel på ett linjediagram finns i Fig. 5.

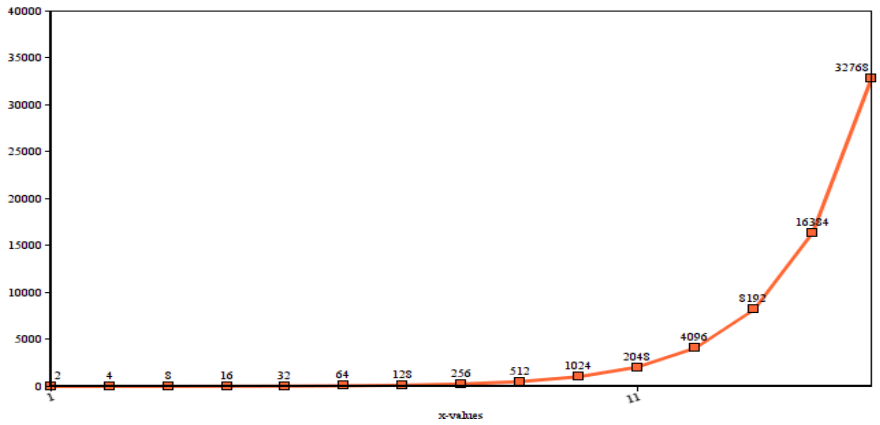

Fig. 5. Exempel på ett linjediagram med bara en mätserie.

Sunburstdiagram

Detta diagram tas bara upp av Gorodov & Gubarev [2]. Författarna anser att sunburstdiagrammet fungerar som ett alternativ till treemap. I ett sunburstdiagram använder man inte höjd och bredd, istället använder man sig av radie och båglängd för att visualisera de olika värdena. Enligt författarna innebär detta att man inte behöver rita om diagrammet när ny data tillkommer, man behöver bara uppdatera de sektioner som har ny data. Exempel på ett sunburstdiagram går att se i Fig. 6.

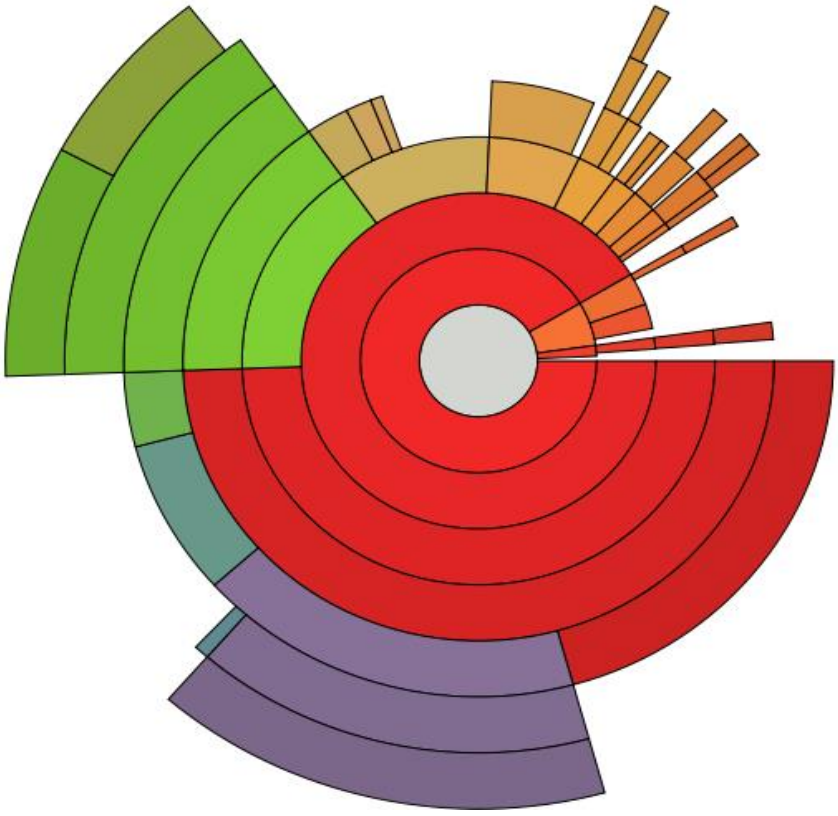

Fig. 6. Exempel på hur ett sunburstdiagram kan se ut. Radie samt båglängd används för att visualisera storlek på data, färg kan användas för att visualisera olika typer av data. Bild tagen från Upload.wikimedia.org [5].

3.1.4. Välja visualiseringsteknik

Munzner tar upp många viktiga saker att tänka på vid utveckling av visuell design [1]. Författaren delar upp visuell design i tre steg: Vad, Varför, Hur.

Vad tar upp generellt vad man kan visualisera i termer av olika typer av data, dataset, attributtyper etc. Frågan om vad som ska visualiseras är inte speciellt relevant för detta arbete då arbetet inte går ut på att ta fram något helt från grunden. Verifyter vet redan vad de vill visualisera vilket innebär att den här punkten redan är besvarad.

Verifyter har två primära typer av data de vill visualisera. Den första typen av data är av sekventiell. De vill visa hur deras kunders projekt förändras angående vissa frågor över tid. Den andra typen av data som Verifyter vill visualisera är större mängder data som enligt de beskrivna typerna av Munzner bäst beskrivs som datasettypen "tables" [1]. Här vill Verifyter kunna visa hur ofta olika filer finns med i misslyckade test.

Den andra punkten *Varför* är mer relevant för arbetet. Denna punkt handlar om varför man ska visualisera datan. Författaren delar upp punkten i flera underrubriker och kategorier. De två huvudkategorierna som författaren beskriver är *Actions* och *Targets*.

Actions handlar om vad man gör med datan. Även den här punkten är uppdelad i underrubriker. De tre underrubrikerna till *Actions* är *Analyze*, *Search* och *Query*.

Första punkten som beskrivs är *Analyze*. *Analyze* handlar om hur man ska ta in och presentera datan. Munzner tar upp att man t.ex. kan visualisera data bara för att presentera data som redan är känd men att man tydligare kan se datan. Man kan också visualisera data för något som inte tidigare är känt och på så sätt få ny information som kan leda till forskning, fler visualiseringar eller liknande.

Nästa punkt i *Actions* är *Search*. *Search* handlar om hur man hittar data när den är visualiserad. Beroende på vad som tidigare är känt om datan kan olika typer av *Search* appliceras på visualiseringen.

Sista punkten i *Actions* är *Query* vilket handlar om frågor och jämförelser som användaren kan vilja göra på datan som har tagits fram. Denna punkt tar upp saker som sammanfattning av datan, identifiering av extremfall samt jämförelse av datapunkter.

Författaren delar upp den andra huvudkategorin, *Targets*, i ett par olika typer men går inte in lika mycket i detalj över de olika såsom det gjordes för *Actions*. *Targets* handlar dock om vad man vill visa med sin visualisering. Detta kan vara saker som att visa trender, extremfall, likheter, samband, former mm.

Tredje punkten som tas upp av Munzner, *Hur*, är väldigt bred [1]. Denna punkt handlar om hur man faktiskt kan visualisera sin data på olika sätt och vad man kan göra med den. Munzner tar inte upp några specifika diagram för olika datatyper i denna punkt. Istället tar

författaren upp ett par underrubriker till hur man kan visualisera data. Dessa underrubriker tar upp olika saker man kan göra med datan. Detta kan t.ex. vara navigering i visualiseringen, hur visualiseringen kan ändra vyer för att visa olika versioner av samma data, hur man kan filtrera data för att få fram det viktiga mm.

Applicerar man *Hur* på de två datatyperna som Verifyter har får man dela upp tankegången utifrån de två datatyperna. Det är dock ganska svårt att sätta sig in i hur man kan manipulera datan innan man har utvecklat sin visuella design. Vet man vad man har för typ av diagram kan man enkelt säga vilka funktioner som en användare skulle vilja ha samt vilka funktioner som går att implementera.

3.1.5. Val av visualiseringsteknik

Utifrån *Vad*, *Varför*, *Hur*, som togs upp i 3.1.4 är det bara *Varför* som inte uppfylls från Verifyters beskrivning av uppgiften. För att applicera Varför på Verifyters data och önskemål får man dela upp det i två delar. Detta pga. att det blir en visualisering (diagramtyp) per dataset, dvs en visualisering för den sekventiella datan samt en visualisering för den större datamängden i “tables”.

För den sekventiella datan är det tänkt att Verifyters kunder skall kunna se hur deras projekts kvalitet mm. förändras över tiden. När det gäller *Search* vet användaren var datan finns, ofta kopplat till en tid eller ett datum. På så sätt kan användaren leta efter data utifrån tidsvärdet. I sista punkten, *Query*, kommer jämförelser att vara mest användbart. Det är tänkt att ett par olika tidsserier skall visas i samma diagram. Användaren kan då jämföra de olika serierna med varandra för att dra slutsatser.

Eftersom datan förändras över tid kommer användaren på så sätt att kunna se trender i datan. Man skulle också kunna se extremfall i sekventiell data i form av “spikar” i linjerna.

Tanken för ”tables”-datan är att denna data skall visualiseras med hjälp av någon form av tabell eller lista. Från en tabell eller lista kan användaren dra egna slutsatser utifrån den data som visas När det gäller *Search* kan många olika typer av sökning fungera. Har man en tabell eller lista vet man ofta var datan finns i fall verktyget är bra strukturerat. Men man kan också tänka sig att man letar efter någon

specifik data och kan då i strukturen hitta datan man letar efter. För *Query* kan man tänka sig att användaren vill kunna identifiera olika data. Användaren skulle också kunna använda tabellen eller listan för att sammanfatta datan på olika sätt och då få ut annan information från samma data.

Tanken med denna data är att användaren skall kunna se var fel uppstår. På så sätt kan man säga att visualiseringen riktar sig mot samband och fördelningar i datan då det är detta som kommer att vara intressant för användaren.

Den sekventiella datan skall användas för att visualisera projektstatus mm. över tid innebär att dess diagramtyp måste kunna hantera sekventiell data. Pga. detta uteblir alla de hierarkiska visualiseringsmetoderna så som treemap och sunburstdiagram.

Kvar för den sekventiella datan är då linjediagram och streamgraph som är de bättre valen för sekventiell data. Utav dessa kvarvarande diagram är linjediagrammet det bästa för denna typ av data och ändamål. I princip alla kan avläsa ett linjediagram och det krävs inte några speciella hänvisningar till diagrammet förutom en liten legend som säger vilken linje som är vilken samt axeletiketter.

Den andra typen av data är svårare att delegera ett specifikt diagram till. Denna data är data som är svårt att visualisera i ett diagram. Istället kommer denna data att visualiseras i en filtrerbar lista eller tabell.

3.1.6. Välja bibliotek

Det finns en mängd olika bibliotek för visualisering och alla har olika funktioner och specialiteter. FusionCharts har tagit fram en lista över vad som är bra att tänka på när man ska välja vilka bibliotek man ska använda sig av [6]. Tabell IV är en sammanställning av vad som tas upp av FusionCharts.

TABELL IV. SAMMANFATTNING AV DE RÅD SOM GES AV FUSIONCHARTS NÄR MAN SKALL VÄLJA ETT VISUALISERINGSBIBLIOTEK [6].

	Kort förklaring/sammanfattning
1. Cross Browser Compatability	Vissa bibliotek fungerar inte i alla browsers. Detta måste man ha i åtanke när man väljer bibliotek. Vilka browsers använder sig slutanvändarna av?
2. Cross Device Compatibility	Vilka plattformar skall biblioteket användas på? Har det stöd för olika skärmupplösningar?
3. Input Data Format	I vilket format är datan? JSON blir mer och mer standard men det finns andra datatyper som man kanske vill använda sig av.
4. Customizability	Hur mycket och vad kan man ändra på? Fler möjligheter för anpassning ger ett mer komplext program.
5. Range of Available Charts	Vilka diagram kan man skapa med biblioteket?
6. Learning Curve	Hur svårt är biblioteket att lära sig? Om denna punkt spelar roll i ett givet projekt beror helt på hur mycket tid man har etc.
7. Compatibility With Other Parts of Code OR Compatibility with Web-Frameworks	Är biblioteket kompatibelt med existerande kod?
8. Performance	Prestanda för biblioteket.
9. Exporting	Går det att exportera diagrammen och i så fall vilka filformat kan man exportera till?

10. Design and Interactivity	Hur ser diagram ut? Hur fungerar interaktivitet för de olika diagrammen?
11. Community and Support	Finns det någon dedikerad support eller sker all support över forum eller användarfrågor?
12. Accessibility	Hur tillgänglig är datan för utomstående? Detta är olika från företag till företag gällande säkerhet.
13. Pricing and Licensing Terms	Vad för licens har biblioteket? Vad får du göra med källkoden?
14. Open Source or Commercial	Man måste vara försiktig om man väljer att gå för open source då det kan vara dålig eller ingen support. Kommersiella bibliotek har ofta bättre support men kan kosta.

Denna tabell har sedan använts när man evaluerar olika bibliotek. Vissa av punkterna får då större och mindre vikt. Till exempel gällande open source och betald licens. Verifyter vill gärna att något open source bibliotek används. Detta pga. att TestHub är till för utveckling och testning. När något i TestHub går till liveversion kan det tänkas att man byter till något kommersiellt bibliotek. Dock för utveckling bör ett open source bibliotek användas.

FusionCharts har också tagit fram ett verktyg för att jämföra olika JavaScriptbibliotek som är till för att göra diagram [7]. Detta verktyg tar inte upp alla möjliga bibliotek, istället har man fokuserat på ett par av de vanligaste och mest välkända biblioteken.

En annan källa som listar flera bibliotek för visualisering är Wang et al [8]. I denna artikel har författarna listat ett antal bibliotek för visualisering. Artikeln specificerar att undersökningen är till för bibliotek för “biologisk” data. Detta betyder däremot inte att vissa bibliotek inte går att använda till annan data.

Sammanfattning av listan går att finna i Tabell VI i Appendix. Kategorier såsom “description” och URL uteblir.

Wang et al. har valt att dela upp biblioteken utifrån vad de kan visualisera. De har valt de tre kategorierna: Charts, Networks samt Hierarchies. De specificerar de olika kategorierna som:

> *"(i) Charts: Multidimensional charts and plots are data in which items with n attributes become points in an n-dimensional space. (ii) Networks: Structures where items are linked to an arbitrary number of other items. (iii) Hierarchies: Collections of items in which each item has a link to one parent or child item.", Wang et al [8].*

Från de tidigare slutsatserna om vilka diagram som ska tas fram blir det uppenbart att diagram för den sekventiella datan faller in under "charts". På så sätt försvinner alla bibliotek som inte klarar av charts. För tabellen eller listan som ska implementeras är det lite svårare att specificera. Eftersom man potentiellt kan vilja utvidga tabelldatan med någon form av ett hierarkiskt diagram kan det vara bra att planera att ha med "hierarchy" som en av de former av visualisering biblioteket klarar av.

TestHub är utformat på ett sådant sätt att de visualiseringsbibliotek som används måste vara JavaScriptbaserade. Så trots att Wang et al. tar upp väldigt många olika bibliotek [8], är det ett litet antal som blir kvar som potentiella kandidater när man tar med kraven på vad biblioteken skall klara av samt att de skall vara JavaScriptbaserade. De bibliotek som blir kvar går att se i Tabell V.

TABELL V. FÖRENKLAD VERSION AV TABLE VI DÄR BARA DE POTENTIELLA KANDIDATERNA TAS MED.

Name	Language	Platform	Type	Supported Categories
D3.js	JavaScript	Web	Open source	Charts/Hierarchies
Google Charts	JavaScript	Web	Free	Charts/Hierarchies
Flot	JavaScript	Web	Open source	Charts

Wang et al. går igenom några bibliotek per kategori charts, hierarchies, networks [8]. För de tre kandidaterna, D3.js, Google Charts och Flot, skriver författarna om både D3.js samt Google Charts för både Charts och Hierarchies. Wang et al. tar inte upp någon extra information om Flot i artikeln. Detta ger intryck av att Flot inte är lika stort eller välkänt som de två andra biblioteken.

Enligt Wang et al. är D3.js ett väldigt kraftfullt verktyg. Problemen med D3 är dels att det är ganska svårt att lära sig och dels att det inte finns några mallar för diagram. Eftersom examensarbetet har en begränsad tid kan detta komma att bli ett problem med D3. Detta blev dock i slutändan inte ett problem. Det positiva med D3 är dock att det är ett väldigt välkänt JavaScriptbibliotek och det finns mycket dokumentation samt forum där man kan få hjälp.

En lösning till att D3 saknar mallar för diagram tas upp av Wang et al.. Det finns andra bibliotek som bygger på D3 som har kod för återanvändbara diagram [8]. En lista över sådana bibliotek ges av McDearmon där många bibliotek som bygger på D3 tas upp [9]. De extra bibliotek som Wang et al. listar är: Dimple, NVD3 och Crossfilter [8]. Alla dessa har sina egna för och nackdelar och är specialiserade inom olika områden.

Google Charts fungerar enligt Wang et al. för både nybörjare och mer välrutinerade utvecklare [8]. Google Charts har många mallar för olika typer av diagram, allt från enkla linjediagram till mer komplexa treemaps.

Sungchul et al. tar upp och jämför fyra olika bibliotek för visualisering gällande renderingstid och hur de hanterar stora datamängder [10]. De bibliotek som tas upp är Google Charts, D3.js, Flex och OFC. OFC och Flex är inte så relevanta för detta arbete då de är Flashbaserade. Flash används mindre och mindre medan HTML5 blir mer och mer vanligt.

I artikeln jämför Sungchul et al. tiden det tar för “Layout Time”, “Data Transformation Time” samt “Rendering Time”. Vid små datamängder, ca. 100 till 1000 punkter är prestandan väldigt lika för både Google Charts och D3 i alla aspekter. När man sedan ökar till 10000 till 100000 är D3 snabbare i alla mätningar. T.ex. “Rendering Time” tar D3 nästan en tredjedel av tiden som det tar för Google Charts.

Sungchul et al. nämner dock att när man har så pass många datapunkter kan det vara bra att slå ihop punkter som är nära varandra. Dels för att göra diagrammet mindre plottrigt och dels för att öka prestandan. En nackdel med att slå ihop datapunkter är att man tappar granulariteten, detta är dock inte ett problem vid 100000 datapunkter då man kraftigt överstiger pixelbredden på de vanligaste skärmarna. En datapunkt måste använda minst en pixel vid visualisering. Om antalet datapunkter överstiger antalet pixlar kommer punkter att ritas över varandra.

D3.js är också open source vilket passar bra med de krav som Verifyter har på licensen på biblioteken.

Google Charts är free to use men inte open source. Detta innebär att man får använda Google Chart gratis, men man har inte tillgång till källkoden och kan därför inte ändra i den. Google Charts har dock ett problem som bara märks för vissa applikationer. Enligt Google får Google Chart inte användas för "Client-side software" [11]. Detta innebär att om man vill använda Google Charts måste man alltid ha en internetuppkoppling utåt då själva skapandet av diagrammet sker hos Google. Detta gör att Google Charts blir oanvändbart för Verifyter då de vill ge sina kunder möjligheten att använda verktygen inom sitt slutna system. Detta då kunderna kan ha känslig information som man inte vill läcka ut till konkurrerande företag.

Då Flot inte är lika stort som t.ex. D3 kan man använda sig av FusionCharts JavaScriptbibliotek-jämförare [8] för att jämföra Flot med andra JavaScriptbibliotek. Den mest intressanta jämförelsen är mellan D3 och Flot eftersom de är de bästa kandidaterna efter att Google Chart har valts bort pga. licensen. Man kan då se att man med D3 kan skapa flera diagramtyper än vad man kan med Flot. Detta innebär att det antagligen är bättre att välja D3 som bibliotek för visualisering då man i framtiden kan tänka sig att man vill lägga till fler diagram.

Problemet med D3 kvarstår, det finns inga mallar för diagram. För att lösa detta kan något av de bibliotek som bygger på D3 som listas av McDearmon användas [9]. Målet med att använda sig av ett av dessa bibliotek är att få tillgång till mallar och återanvändbara diagram. Detta underlättar arbetet då diagrammen inte behöver tas fram från grunden.

NPMCompare jämför fyra olika bibliotek som bygger på D3 [12]. Dessa bibliotek är: D3.chart, NVD3.js, Rickshaw och Vega. I denna jämförelse anses NVD3 vara markant bättre än de andra biblioteken. Denna jämförelse säger inte så mycket om vad de olika biblioteken är kapabla till. Istället jämför NPMCompare hur väl de olika biblioteken underhålls och hur populära de är. Denna jämförelse samt att NVD3 var ett av de bibliotek som Wang et al. [8] tar upp får NVD3 att se ut som den klara vinnaren.

3.1.7. Val av bibliotek

Av alla de visualiseringsbibliotek som finns är Google Charts och D3.js de bästa för just detta arbetet. På grund av Google Charts licens går Google Charts inte att använda för arbetet. Problemet med D3 är att det inte finns några mallar för diagram. För att lösa detta problem användes ett extra bibliotek som bygger på D3.js. Utav dessa extra bibliotek användes NVD3 då det är ett av de mer populära biblioteken som bygger på D3. NVD3 har också många mallar och exempeldiagram som går att återanvända.

Det positiva med dessa val är att de diagram som redan finns i TestHub redan är gjorda med hjälp av just D3.js och NVD3. På så sätt kommer de nya diagrammen att vara enhetliga med resten av TestHub.

För att visualisera tabeller och listor kan vanlig JavaScript och html användas. Dock kommer List.js, ett litet JavaScriptbibliotek för listhantering, att användas för detta ändamål för att förenkla arbetet. List.js är under MIT licens, så det går bra att använda för detta arbete.

3.2. Källkritik

Vissa källor är mindre pålitliga än andra. De mindre pålitliga källorna är de som kan vara partiska och de källor som är skrivna som bloggar. T.ex. McDearmon [9] och NPMCompare [12] tar upp olika bibliotek som bygger på D3. Man kan inte veta om de har tagit med just de bibliotek som de föredrar eller om de har utelämnat något annat viktigt bibliotek. De är dock bra för att få en förståelse över vilka bibliotek som finns.

En annan typ av källa som man måste vara kritisk med är FusionCharts [6][7]. Här har verktyg för jämförelse tagits fram när FusionCharts själva säljer ett bibliotek som tas upp i deras jämförare. De kan då välja hur de vill visa och vinkla informationen. Ifall något bibliotek är bättre i någon aspekt kan de välja att inte ta upp den aspekten i sin jämförelse då de själva har skapat verktyget. Trots detta kan dessa källor vara bra att använda så länge man har i kommer ihåg att informationen antagligen är gjord på ett sådant sätt att FusionCharts ser bättre ut än deras konkurrens.

De andra källorna gällande visuell design är mycket mer pålitliga. Dessa är vetenskapliga artiklar som har publicerats och granskats vilket gör att man kan lita på innehållet i dem. Några exempel är Gorodov & Gubarev, Krstajic & Keim och Wang et al. [2][3][8].

Samma sak gäller för Munzner [1]. Detta är en bok som själv har ett stort antal referenser till vetenskapliga artiklar och andra böcker. Detta gör att referensen går att lita på. Från denna källa har datan till tabell 1 tagits. Denna data är inget som Munzner kan ha copyright på då hon själv refererar till den från Anscombe's Quartet. Denna data i sin tur är allmänt känd och går att finna på t.ex. Wikipedia. Fig. 1 som kommer från denna tabell har gjorts i Excel utifrån datan i tabell 1 och bör på så sätt inte strida mot någon copyright.

Källor för bilder så som t.ex. FPPT [4] går ofta att lita på. Problemet med dessa källor är upphovsrätt. Därför är det viktigt att källor gällande bilder och tabeller är källor som det är tillåtet att kopiera ifrån. Just FPPT [4] bör gå att använda då förkortningen står för "free power point templates".

Andra källor som används är information från olika företag eller utvecklare. Dessa källor är Google Developers FAQ [11], Novus Partners hemsida för NVD3 samt deras Github repository för NVD3 [13][14], Strömberg som är utvecklaren för List.js [15], dokumentationen för ArrayNode samt ObjectNode [17][18] och slutligen JSON [16] som beskriver vad JSON är. Dessa källor kan anses vara korrekta då det är i företagen och utvecklarnas intresse att förmedla korrekt information. Ifall någon fel information skulle förmedlas i någon av dessa källorna skulle deras tänka användare tappa förtroende för produkten.

4. Analys

Detta kapitel behandlar den programmering som behövdes göras innan några diagram med riktig data kunde implementeras. Resultaten från diagramimplementationen går att se i kapitel 5, Resultat.

4.1. Skapa en nya sida samt lägga till bibliotek

Efter att man har satt sig in i hur TestHub fungerar gäller det att skapa en ny sida där diagrammen kan visas. För att göra detta behöver en knapp läggas till i HTML-filen index. Denna knapp kommer sedan att bindas till en JavaScriptfil, dataGraphs.js, så att när man trycker på knappen kommer diagrammen att visas och allt annat kommer att döljas.

Tidigt i utvecklingen blev det tydligt att det skulle bli väldigt rörigt om man ska visa flera diagram och tabeller/listor på samma sida. Ifall man skulle populera en webbsida med många element skulle användaren behöva scrolla för att hitta det diagram som hen är ute efter. För att lösa detta problem togs ett par knappar fram. Dessa knappar döljer och tar fram olika diagram samt en knapp för att dölja alla diagram. Dessa knappar ger en möjlighet till att enkelt implementera nya diagram då nya idéer kommer upp. Dessa knappar samt knapp för att komma till sidan med grafer går att se i Fig. 7.

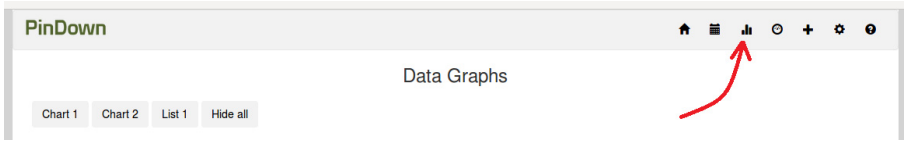

Fig. 7. Bilden visar sidan för diagrammen. Bilden visar hur hemsidan ser ut när knappen “Hide all” har tryckts. Pilen visar den knapp som har implementerats in menyn som tar användaren till sidan för diagram.

För att sedan lägga till de JavaScriptbibliotek som man skall använda sig av är det bara att lägga filerna till biblioteket i “public”-mappen i TestHub. Det är dock viktigt att komma ihåg att också lägga till sökvägen till JavaScripten i htmlfilen main då det är där som JavaScripten initieras när programmet startas. Eftersom både D3.js och NVD3.js reda används av TestHub behöver dessa inte importeras.

Däremot behöver List.js importeras från utvecklarens hemsida, http://listjs.com/, och sedan läggas till i TestHub.

4.2. Lägga till exempeldiagram

Eftersom NVD3 valdes till det bibliotek som skulle användas för arbetet var det bara att implementera diagram utifrån NVD3s redan färdiga diagram. De exempel på diagram som finns på NVD3s hemsida [13] är inte helt uppdaterade och har inte speciellt många diagramtyper. Istället har de ett GitHub repository med många olika diagram med exempeldata [14]. Eftersom det första diagrammet som skulle implementeras var någon typ av linjediagram var det ganska självklart att skapa ett diagram av typen "lineChart". Exemplet för "stackedAreaFocusChart" såg ut att fungera ganska likt ett linjediagram. Dock när detta diagram implementerades blev datan förvrängd vilket gjorde att denna typ av diagram inte kunde användas.

En annan intressant typ av diagram var "lineChartWithFocusChart". Detta diagram ser ut och fungerar precis som ett vanligt linjediagram. Skillnaden är att det finns ett extra diagram under linjediagrammet. I detta extra diagram kan användaren välja att zooma in på specifika delar av datan som ser intressant ut. Denna extra funktion kan inte anses försvåra för användaren då man kan välja att bara använda diagrammet som ett vanligt linjediagram ifall man inte vill använda på zoomfunktionen. Eftersom detta diagram har samma funktioner som ett vanligt linjediagram fast med en extra funktion valdes detta diagram till att visualisera den sekventiella datan.

Ett litet problem som uppstod var att ibland när diagrammet skulle visas så blev hela diagrammet hoptryckt. Detta problem går att se i Fig. 8.

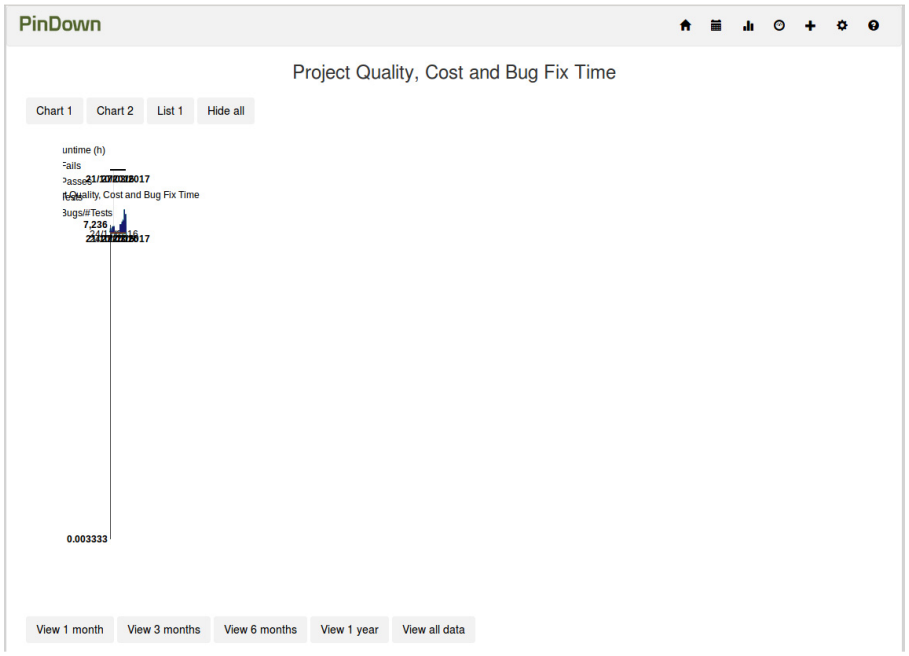

Fig. 8. Bilden visar hur diagrammet är hoptryckt när det skall visas.

Diagrammet återgick till rätt storlek så fort man ändrade något med visualiseringen som t.ex. ändra storleken på browserfönstret eller ta fram/bort en dataserie. Att diagrammet rättades till beror på att diagrammet uppdateras när man ändrar fönsterstorlek eller vilken data man vill visa. För att göra så att diagrammet alltid laddades in med rätt storlek sattes en "window resize" till knappen för att visa diagrammet. På så sätt "ändras" fönsterstorleken på webbläsaren när man vill ta fram diagrammet. Denna ändring i fönsterstorlek är inget som användaren ser och medför inga märkbara försämringar i prestanda. Programmet säger bara till JavaScriptet att fönstret har ändrats, fast det egentligen inte har ändrats, och kod som då är beroende på detta kan på så sätt köras.

För den andra typen av data skulle någon form av tabell eller lista användas. Eftersom NVD3 inte har något diagram av denna typ används List.js för detta ändamål. List.js har ett par små exempel på hur biblioteket fungerar. Ett av dessa exempel var hur man skapar en ny lista utifrån data i en JSON-sträng [15]. Detta exempel passar utmärkt för hur data hanteras i TestHub och implementerades därför.

Diagrammen fungerade utmärkt med den testdata som fanns för både linjediagrammet och listan. Dock blev det problem när den riktiga datan skulle användas.

4.3. Problem med JSON

För att få ut data på hemsidan och därigenom diagram på hemsidan, används JSON. Detta är ett format som gör det lätt för programmet att läsa datan. En JSON-sträng är uppbyggd av objekt, inneslutna i krullparenteser "{ }", samt vektorer, inneslutna i hakparenteser "[ ]" [16]. För att få ut data till ett diagram känns det naturligt att ha en vektor med de punkter man vill rita ut. Datan till de diagram som redan fanns i TestHub var utformad som en vektor innehållande ett objekt som håller många småvektorer med värden. Exempel på en sådan JSON-sträng är följande:

```
{"data":[{"key":"fail","values":[[1463504400000,6], [1463504460000,2
],…
```

Detta format är relativt enkelt att förstå. Den yttre vektorn används för att kapsla in datan och det första objektet innehåller olika attribut för diagrammet. I de mindre vektorerna som följer är det första värdet X-värdet och det andra värdet är Y-värdet. Dessa JSON-strängar är gjorda i Java där klassen ArrayNode representerar vektorerna och klassen ObjectNode representerar objekten.

Problemet som uppstod med datan i linjediagrammet har att göra med hur JSON-strängarna var utformade. De diagram som redan fanns i TestHub behövde ha datan utformad på sättet med småvektorer som representation av punkterna. Linjediagrammet behöver däremot objekt med faktiska X- och Y-värden. Listorna gjorda med List.js behöver objekt med de värden som man vill visa i listan, t.ex. namn, status, körtid etc. Detta innebär att hantera punkter som småvektorer inte fungerar och att någon annan lösning som faktiskt hanterar objekt måste implementeras.

Det första försöket att skapa en JSON-array med objekt använde Javaklassen Point för att representera objekt. Detta kändes naturligt då Point har definierade variabler X och Y. Dock kan ArrayNode inte innehålla klassen Point vilket innebar att denna lösning förkastades.

En annan lösning som testades och som fungerade var att formatera datan så som den tidigare hade gjorts med vektorer som representation av punkter. Sedan gjordes en funktion i JavaScript för att gå igenom JSON-strängen och byta tecken i strängen så att JSON-strängen fick det format man var ute efter. På så sätt såg JSON-strängen ut att innehålla objekt med X- och Y-värden.

Denna lösning fungerade bra dock tog det ibland lång tid att läsa in JavaScriptet då varje gång man startade programmet behövde gå igenom en lång sträng och ändra den. Ju större datamängd man vill visa desto längre tid skulle det ta att gå igenom strängen. På grund av dessa problem förkastades denna lösning också.

Den slutgiltiga lösningen till problemet var att försöka göra om JSON-strängen så att den faktiskt innehöll objekt istället för vektorer från första början.

Efter att ha granskat dokumentationen för både ArrayNode, ObjectNode [17][18], samt strukturen för hur diagrammet ville att JSON-strängen skulle vara utformad framkom följande lösning. Som objekt med X- och Y-värden kan man ha en ObjectNode för varje punkt. I en ObjektNode lägger man sedan till värden för punkten med en sträng “x” eller “y” före värdet för punkten. På så sätt tolkar programmet strängen "x" och "y" som variabelnamn och kopplar sedan ihop värdet efter strängen med variabelnamnet. Exempel på hur man gör JSONobjekt med ObjectNode:

```
for (Cost c : cost) {
          ObjectNode costPair = new ObjectNode(factory);
          costPair.put("x", c.commitTime.getTime());
          costPair.put("y", c.runnTime);
          jsonResponse.add(costPair);
}
```

Denna ObjectNode läggs sedan till i ArrayNode:n som representerar JSONvektorn som innehåller alla punkter. Exempel på hur JSONsträngen ser ut med ObjectNodes som objekt:

```
{”data”:[{”key”:”RunTime (h)","color":"GoldenRod","values":[{"x"
:1477025549000,"y":0.24111},...
```

4.4. Databasen

I TestHub finns det för den data som redan visas och används en "H2"-databas. Denna databas har ett initialiseringsskript som måste köras varje gång systemet startas om. Denna databas fungerar bra för den data som redan finns i TestHub då det inte är några stora mängder data, SQL-skriptet är på ca 300 rader för att skapa tabeller samt lägga in data i tabellerna.

Den data som Verifyter vill visualisera i detta arbete är dock mycket större än den tidigare datan, bara en av tabellerna innehåller ca 6500 rader. Att ha denna data som ett skript som skall köras varje gång systemet startas om är inte hållbart då det kommer att ta väldigt lång tid att köra. Istället har Verifyter skapat en PostgreSQL-databas för denna data. Play! Framework som TestHub är uppbyggt av skall enligt dokumentationen kunna hantera mer än en databas.

För att lägga till en databas behöver man lägga till information om databasen i en fil som heter application.conf. Information här är t.ex. databas URL, lösenord och användarnamn till databasen. Det är också här som det skall gå att lägga till flera databaser.

När denna funktionalitet skulle implementeras gick det mindre bra. Det gick att få TestHub att bara köra på den nya databasen eller bara den gamla men inte båda samtidigt. Detta var dock inte ett större problem. Verifyters mål är att när TestHub sedan är färdigt skall man använda sig av en riktig databas istället för den lokala databasen som Play använder sig av. Eftersom den redan existerande databasen i princip bara var ett kort SQL-skript kunde detta skript köras på den nya databasen och på sätt kan både den nya och den gamla datan nås från den nya databasen. Det positiva med detta är att PostgreSQL-databasen inte behöver köra ett initialiseringskript varje gång programmet startas om.

När försök gjordes att använda två databaser skapade vi en ny javaklass “DatabaseTwo” i tron att man skulle ha en Databasklass per databas. Detta innebar att ett par metoder implementerades i DatabaseTwo, dock när all data slogs ihop till en databas är det inte nödvändigt att ha två databasklasser. Metoderna från DatabaseTwo går dock ganska lätt att flytta till den ursprungliga databasklassen. Det kan dock vara en bra idé att ha kvar klassen DatabaseTwo för att

separera de olika databasmetoderna då det inte bör medföra några problem med att ha två databasklasser.

5. Resultat

5.1. Implementation av diagram

Verifyter specificerade ursprungligen tre olika dataserier som de vill visualisera. Dessa dataserier är “Quality, Cost och Bug Fix Time” vilket från inledningen kan ses som ”projektkvalitet, projektkostnad och projektstatus”. Eftersom datan i databasen är utspridd i olika tabeller måste tabeller slås ihop genom SQL-frågor för att få fram rätt data. Dessa SQL-frågor varierar i komplexitet och i vissa fall måste två SQL-frågor användas och sedan behandlas i Java. Ett exempel på detta är att tid för testkörningar finns inte för alla tabeller. Istället finns det en tabell som har tider för alla testkörningar och denna tabell måste man sedan slå ihop med sina andra SQL-frågor för att få fram starttider för testkörningar.

Quality är den mest grundläggande dataserien av de tre och implementerades först. Enligt Verifyter kan Quality ses som antal fallerande test, “fails” per körning. Dessa sparas i databasen som både “build fails” och “test fails” men båda räknas som fails. Resultatet för Quality finns i Fig. 9.

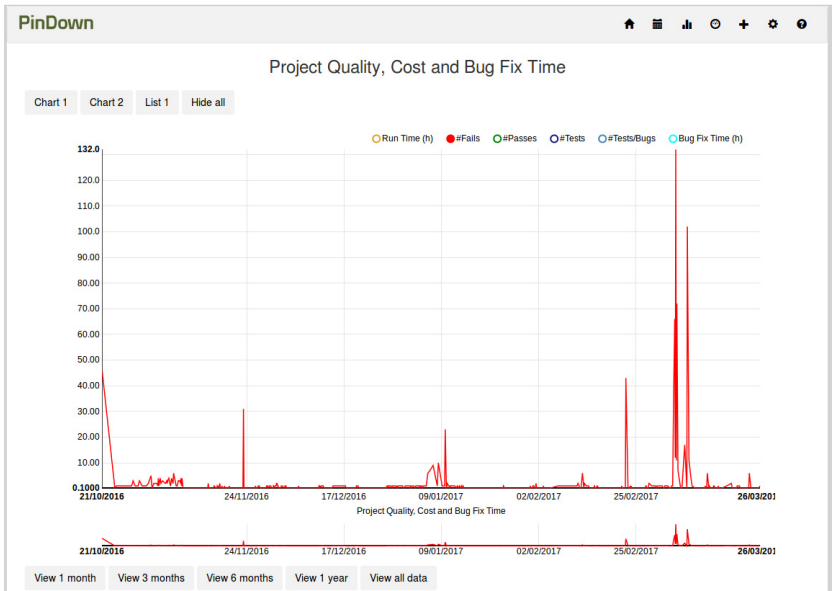

Fig. 9. Diagram som visar Quality, antal fails.

Nästa dataserie som implementerades var Cost. Denna dataserie specificeras av Verifyter som “antal test delat med antalet buggar”. Antalet buggar finns inte sparat som något värde i databasen istället får man räkna unika commitmeddelanden för en körning. Ifall ett commitmeddelande förekommer mer än en gång i tabellen över buggar anser man att buggen först förekom i den körning som körs först och bör då räknas in i antalet buggar för den tidigare körningen.

Antalet test fås från en annan tabell som innehåller alla testkörningar. Här räknar man alla körningar med samma nummer, “runno”, för att på så vis få fram hur många gånger t.ex. runno 678 har körts.

Denna dataserie blev väldigt spikig och svårläst pga. två problem. Det första problemet var att ifall det inte fanns några buggar för en körning fick man väldigt stora värden i förhållande till de punkter där det faktiskt fanns buggar. Det andra problemet var att körningarna inte körs dygnet runt. Detta medförde att tider då inga test kördes gick kurvan ner till noll för att sedan bara några timmar senare stiga. Detta medförde att dataserien blev väldigt spikig och det var svårt att utläsa något från grafen.

För att lösa dessa två problem infördes två lösningar. Först sattes de körningar som var utan buggar till 0 i diagrammet. Eftersom y-värdet räknas ut som antalet test delat med antalet buggar skulle man ha delat med 0 ifall man tog med dessa testkörningar i diagrammet. På så sätt kan man få större perioder i grafen där y-värdet är noll. Detta innebär då att man under den tidsperioden inte har haft några buggar. Den andra lösningen var att ha en veckas medelvärde som utdata till grafen. Detta medför att grafen inte blir lika spikig och att användaren lättare kan utläsa trender i grafen.

Ett mindre problem som uppstod var att Verifyter först specificerade Cost som “antal buggar delat med antal test”. Det de egentligen ville har ut var inversen. Detta problem var dock lätt att åtgärda genom att bara ändra täljare och nämnare när datan skulle beräknas. Bild för hur Cost ser ut går att se i Fig. 10.

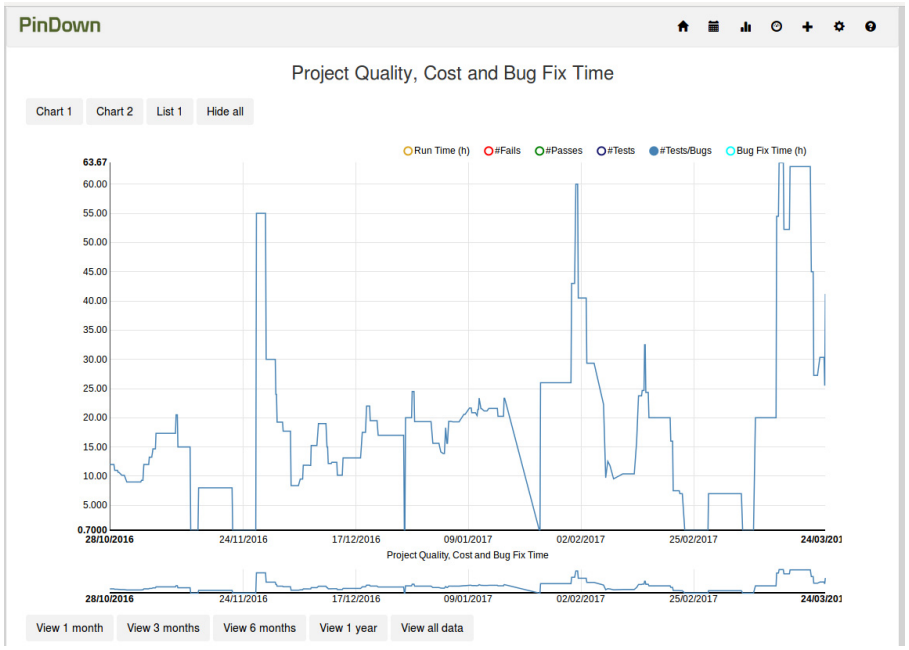

Fig. 10. Diagram som visar Cost, antal test delat med antalet buggar. Dataserien är mer utjämnad pga. veckomedelvärdet som beräknas för varje punkt.

Den sista dataserien som Verifyter specificerade var Bug Fix Time. Denna dataserie specificerades genom Formel (1).

$$\frac{\sum(Verified\ Fix\ Time - Commit\ Bugs) + \sum(Verified\ Fix\ Time - Detect)}{\#Bugs}$$

(1) Formel för att beräkna Bug Fix Time specificerad av Verifyter.

Då denna formel var svår att tolka samt att inte all data fanns i den databas som var tillhandahållen förenklades formeln istället till Formel (2).

$$\frac{Verified\ Fix\ Time}{\#Bugs}$$

(2) Förenklad formel för att ta fram Bug Fix Time.

Med denna formel får man ut tiden det tar att fixa de buggar som finns i en testkörning.

För att få fram “verified fix time” blir SQL-frågan ganska lik Cost. Man behöver först få fram starttiden för körningen där buggen först uppkom. För att sedan anse att buggen är garanterat fixad tar man fram starttiden för körningen efter den körning där buggen sist förekommer. Till exempel om körning 345 är sista gången bug 5 uppkommer är det vid starttiden för körning 346 som buggen kan anses vara garanterat löst. På så sätt har man två stycken tider, tar man skillnaden mellan dessa tider får man ut verified fix time. Antalet buggar går att få ut på samma sätt som för Cost. Bug Fix Time går att se i Fig. 11.

Fig. 11. Diagram som visar Bug Fix Time.

När dessa tre primära dataserier var implementerade ville Verifyter ha tre nya dataserier för att ge användaren en bättre överblick av projekten. Dessa nya dataserier var “Run time, antal passes och antal test”. Alla dessa dataserier var relativt lätta att implementera utifrån den data som fanns.

För att ta fram Run Time vill man ha den längsta tiden sparad i databasen för ett specifikt körnummer. Denna tid är i databasen sparad som sekunder vilket inte säger användaren speciellt mycket.

För grafen är denna tid omräknad till timmar. Run Time går att se i Fig. 12.

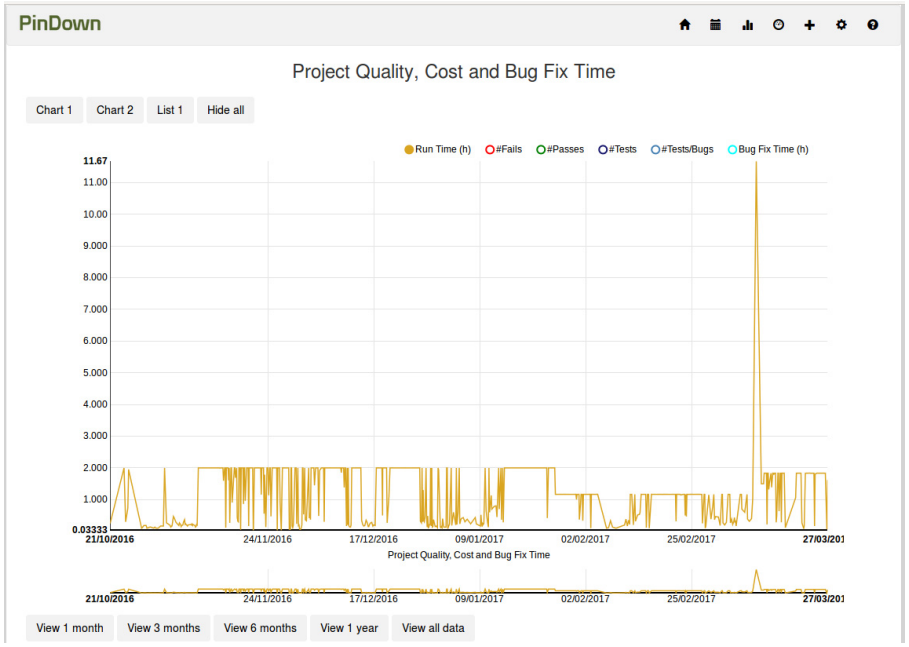

Fig. 12. Diagram som visar Run Time.

Antal passes och antal test är väldigt lika. Den enda skillnaden är att antal test är antal passes plus antal fails. Antal fails har redan tagits fram för Quality vilket innebär att det är bara passes som behövs tas fram. Passes tas fram på liknande sätt som för fails. Passes sparas som "build pass" och "test pass", dessa värden kan då tas fram per körning och läggas ihop. För att sedan få ut antal test lägger man ihop antal passes med antal fails. Antal passes och antal test kan går att se i Fig 13 och Fig 14.

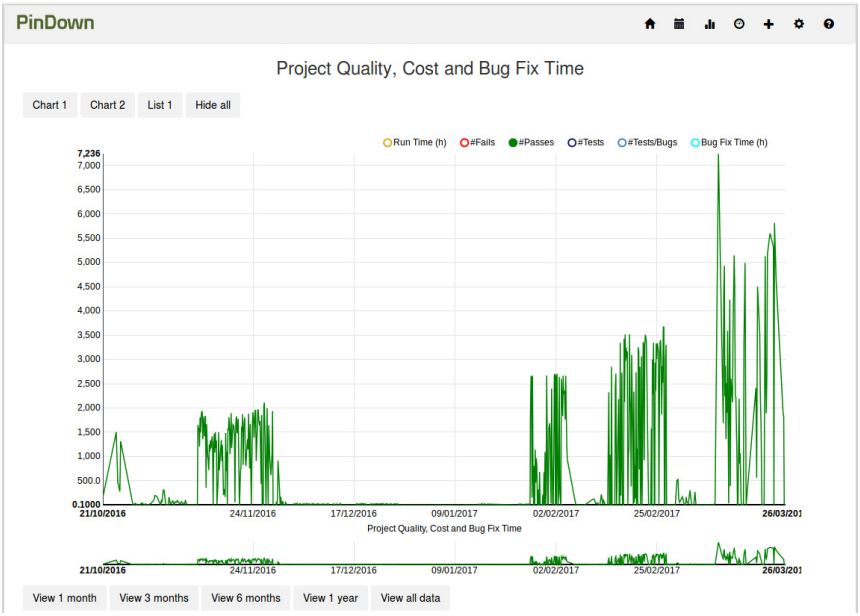

Fig. 13. Bilden visar dataserien för antal passes.

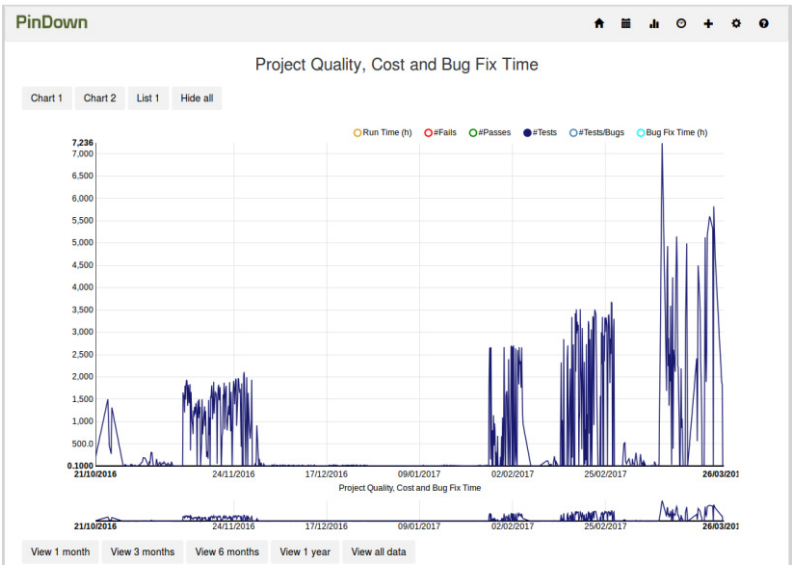

Fig. 14. Bilden visar dataserien för antal tests. För just den här datan är antal test väldigt lik antal passes. Ifall det skulle finns fler fallerande tester skulle denna graf bli mer olik grafen för antal passes.

Ett mindre problem med att ha så pass många olika dataserier i samma graf är att y-värdena för de olika dataserierna är väldigt olika. T.ex. så är det högsta värdet för Run Time ungefär 12 timmar medan det högsta värdet för antal test är nästa 7000. Detta gör att det är väldigt svårt att se alla linjer samtidigt. För att minska detta problem finns det en teckenförklaring överst i diagrammet. Här kan användaren välja vilka linjer som skall visas i diagrammet utan att behöva flera diagram. Exempel på hur detta fungerar finns i Fig. 15.

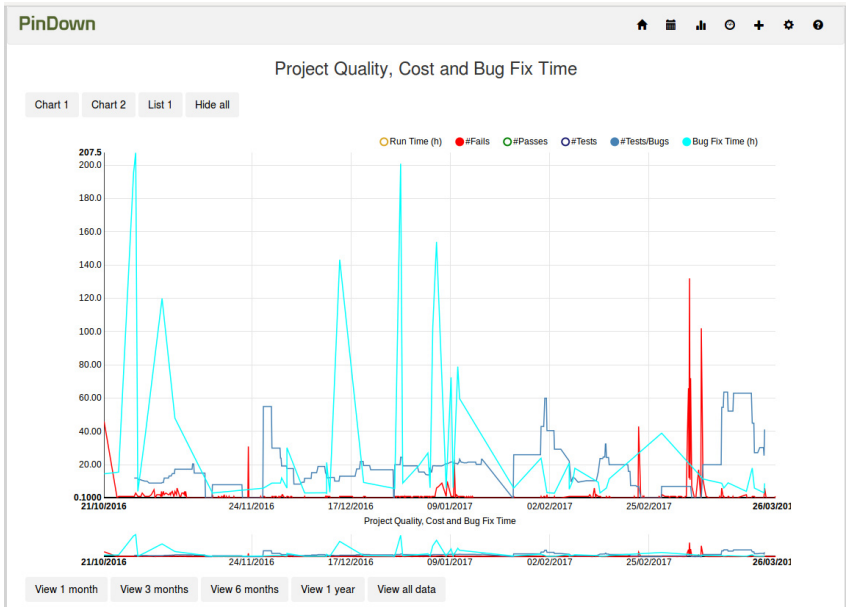

Fig. 15. Bilden visar hur man i teckenförklaringen (uppe till höger i diagrammet) kan välja vilka dataserier som skall visas. Exemplet visar Fails (Quality), Tests/Bugs (Cost) samt Bug Fix Time.

En kommentar på diagrammet som Verifyter kom med var att ibland vill man se specifika delar av datan t.ex. senaste månaden, senaste tre månaderna etc. Verifyter ville att det ska finnas knappar under grafen för att ställa in så att fokusgrafen ställs in för ett par olika intervall istället för att man manuellt ska behöva ställa in det.

De intervall som valdes var: en månad, tre månader, sex månader, ett år samt en knapp för att visa all data (“reset zoom”). Dessa knappar är bundna till olika zoom på fokusgrafen och använder datumet för sista mätpunkten för att sedan beräkna var zoomen skall lägga sig. Exempel på zoom finns i Fig. 16.

Fig. 16. Bilden visar en månads zoom för dataserierna Quality, Cost och Bug Fix Time. Den mindre undre grafen visar zoomfältet medan det större diagrammet visar den data som ligger i zoomfältet.

Andra småsaker som behövdes fixas till med diagrammet var att sätta rätt färg på de olika dataserierna samt ha rätt axeletiketter. För färg gav Verifyter två direktiv, fails ska vara röd och passes ska vara grön. De andra dataserierna kan i princip ha vilken färg som helst. Det viktiga är att hitta färger som är så pass olika att man kan urskilja vilken som är vilken ifall man visar båda linjerna i samma graf. Man kan inte välja färger som är för ljusa. Dessa färger kan, beroende på vilken skärm man använder, nästan försvinna eller bli mycket svåra att se då bakgrunden är vit. Slutligen valdes sex stycken olika linjer alla med färger som man lätt kan skilja mellan varandra. Dock, ifall man vill implementera flera dataserier bör man nog fundera på att implementera ett nytt diagram.

När det gäller axeletiketter är det självklart att x-axeln bör ha tid som enhet. I databasen är tiderna sparade som “Timestamps” med faktiska tider, för att få ut datan måste den sparas som millisekunder. Dessa millisekunder är räknade i Unixtid, dvs tid i millisekunder

sedan midnatt 1 januari 1970. D3 kan sedan lägga på ett filter för att formatera millisekunderna till datum.

För y-axeln är det desto svårare att sätta en enhetlig axeletikett. Eftersom de olika dataserierna har olika enheter (tid för Run Time, antal för Fails) beslutades det att y-axeln bara ska visa värdet utan någon speciell formatering. Istället så visas det i teckenförklaringen vad de olika dataserierna har för enhet, t.ex. så visas Run Time som "Run Time (h) och Fail visas som #Fails. Legenden visas i Fig. 17.

Fig. 17. Exempel på hur datan visas i teckenförklaringen för att visa enheten för dataserierna.

5.2. Implementation av lista

Verifyter har data över vilka filer som finns med i olika buggar. Det de vill visa är hur ofta de olika filerna förekommer i buggar. Denna data går att få fram ungefär på samma sätt som att ta fram antalet buggar. Man kan få fram datan genom att slå samman tabellen för bugfiler med tabellen för commitmeddelanden där körnummer och bugnummer är samma. Den data man får ut är en tabell där varje fil som förekommer i en bug är sammankopplad till ett commitmeddelande. Som för att ta ut antalet buggar gäller det att varje unikt commitmeddelande kan anses som en ny bug. Detta innebär att alla filer som har commitmeddelande "XXX" förekommer i samma bug.

Implementationen av listan utgick från ett exempel från Strömberg, författaren av List.js [15].

För att skapa en lista med List.js behöver man först lägga till ett element i HTML-filen index där man vill att listan skall skapas. I detta element kan man sedan lägga till inputfält för sökning samt knappar för sortering. För denna data lades ett sökfält till då användaren kan tänkas vilja söka på olika filter eller antal buggar. En knapp för sortering av antal buggar lades också till. Då man inte vet hur många buggar det kan finnas beroende på hur man filtrerar datan kan vara bra att kunna sortera datan i både stigande och fallande ordning.

I JavaScriptfilen lades sedan kod för att skapa en lista bunden till det element som tidigare skapas i indexfilen.

För att skapa en lista i JavaScript med List.js behövs tre parametrar. Först behöver man specificera vilket element i HTML-koden som listan skall bindas till. Nästa parameter är "options", här kan man specificera vilka attribut varje element i listan skall ha. Man säger också vilka HTML-tagger som datan skall bindas till och hur de skall vara formaterade. Man kan också lägga till andra alternativ till listan som t.ex. vad sökfältet skall söka efter etc. Den sista parametern som behövs när man skapar en lista med List.js är själva datan.

Datan till listan fås ut på samma sätt som datan för diagrammet. En funktion i databasklassen måste skapas som tar fram och behandlar den data man vill visa. Denna funktion används sedan av Application för att skicka datan till JavaScriptet.

För att listan inte skulle bli allt för rörig att hitta i lades "pagination" till i options för listan. Pagination innebär att datan blir uppdelad på olika sidor och att man genom knappar kan stega sig igenom listan. Ett problem med detta var att om man är mitt i listan finns det inget enkelt sätt att ta sig till början eller slutet då List.js inbyggt bara visar knappat för fem sidor åt gången med pagination. För att lösa detta implementerades fyra nya knappar, fram ett steg, bak ett steg, gå till början och gå till slutet. Dessa knappar lades till i indexfilen och deras funktioner skapades i JavaScriptet.

Det finns dock fördelar med att se all data på samma sida trots att användaren behöver scrolla för att hitta data. Man kan tänka sig att man letar efter mönster i datan och detta kan vara enklare att se ifall all data visas på samma sida. För att ge användaren möjligheten att se datan på en sida eller använda pagination skapades en knapp på sidan för att ta användaren mellan de olika versionerna av listan.

En sista funktion som Verifyter ville ha gällande listan var att användaren skulle kunna applicera något filter så att istället för att se hela filens sökväg kan man välja att bara se själva filen. Detta filter skulle implementeras så att användaren kan skriva in en sträng i form av regex. Regex används för att specificera ett mönster för sökning på strängar. Det första som behövdes implementeras för detta var ett inputfält på sidan för listan där användaren kan mata in ett regex för filtrering. För att kunna använda datan som användaren matar in i

fältet behövdes en ny route skapas. Denna route tar emot datan som skrivs in och sparar strängen. Denna sträng används sedan som inparameter till Application metoden som tar fram listan. Denna metod kallar i sin tur på databasmetoden som tar fram datan för listan, fast med regexfilter som inparameter. Det är på så sätt databasemetoden som filtrerar datan och skickar tillbaka den till Application som sedan kan tas emot av JavaScriptet.

Detta innebär att varje gång ett filter läggs på måste listan skapas på nytt. Detta medför att det kan ta ett par sekunder att ladda listan då all data först måste hämtas och sedan filtreras innan den kan renderas av programmet.

Den framtagna listan med dess funktioner går att se I Fig. 18 samt Fig. 19. Filnamnen har tagits bort pga. sekretesskäl.

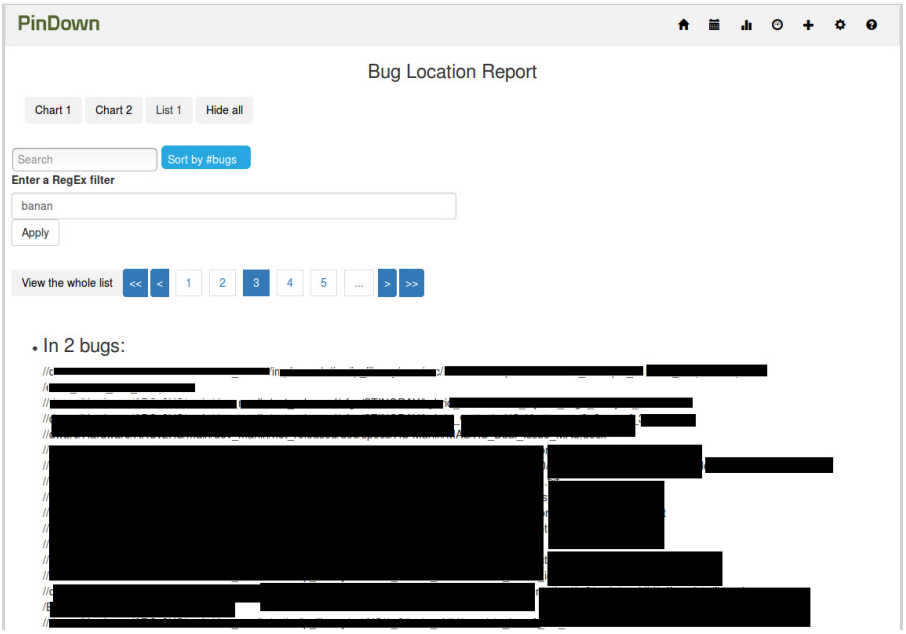

Fig. 18. Bilden visar den implementerade listan. Överst finns sökfältet samt knapp för sortering av listan. Det undre fältet används för att skriva in det filter man vill applicera på listan. Precis över listan finns det s.k. pagination som gör att användaren kan stega igenom listan utan att få all data på samma sida samt knappen för att se all data.

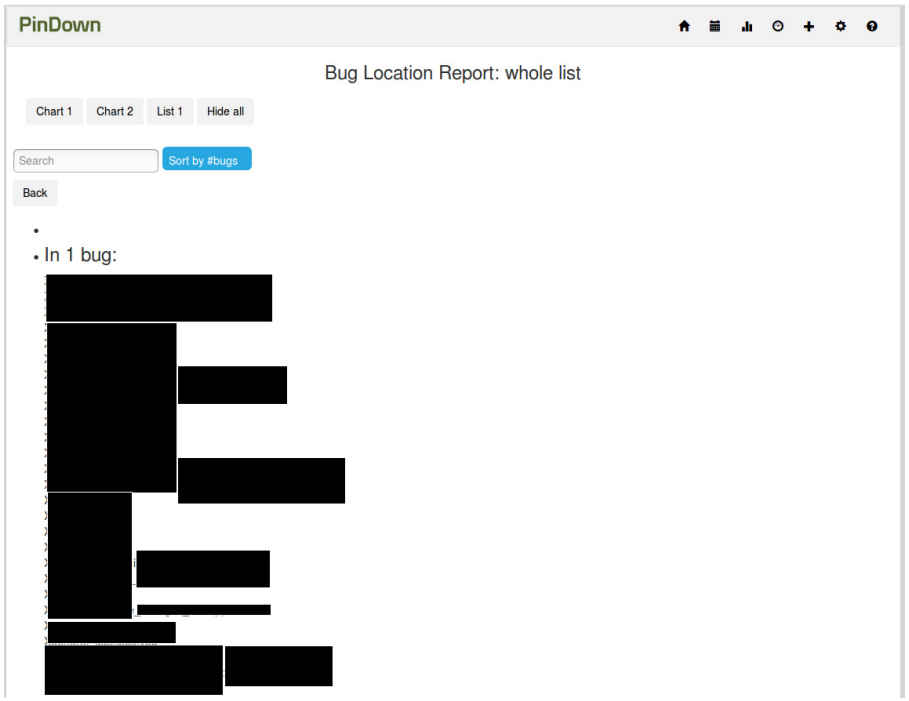

Fig. 19. Bilden visar listan efter att “View whole list”-knappen har tryckts. Hela listan visas inte då listan är på flera hundra rader. Även här kan man använda sökfältet och sorteringsknappen.

6. Slutsats

Detta kapitel tar upp och reflekterar över resultatet utifrån den problemställning som gjordes i början av rapporten. Reflektioner över etiska aspekter samt framtida utvecklingsmöjligtheter tas också upp.

6.1. Reflektion

För att förmedla komplex information till användare kan visualisering vara ett mycket kraftfullt (och viktigt) verktyg, och beroende på vad man vill förmedla och visualisera kan olika tekniker användas. Utifrån den data som Verifyter ville visualisera samt de olika för- och nackdelar som finns bland olika diagramtyper blev ett linjediagram den bästa diagramtypen för att visualisera projektkostnad, projektstatus och projektkvalitet.

Linjediagrammet fungerade bra för att visualisera flera dataserier i ett diagram och samtidigt inte göra diagrammet för rörigt. Detta hade varit svårt att göra lika bra med en annan diagramtyp. För att kunna visualisera projektstatus, projektkostnad samt projektkvalitet i samma diagram krävs det att diagrammet kan visa hur data förändras över tid. På så sätt blir många hierarkiska diagram oanvändbara för denna data.

Den andra typen av data som Verifyter ville visualisera blev, utifrån de olika kriterier och egenskaper hos diagram, bäst visualiserad som en lista. En lista eller tabell är bättre än ett diagram på att visa stora datamängder där själva informationen i datan är intressant för användaren. På så sätt kan användaren söka igenom en strukturerad lista för att hitta den intressanta datan. Denna data kan sedan visualiseras i något hierarkiskt diagram dock kan viss information om datapunkterna gå förlorad.

Det finns en mängd olika bibliotek för diagram och visualisering i allmänhet. Beroende på vad man vill visualisera, hur erfaren man är samt vilken typ av licens man kan använda kan man välja de bibliotek som lämpar sig bäst för ens arbete. För detta examensarbete var D3.js med NVD3.js det bäst lämpade biblioteket för att skapa grafer samt List.js för att skapa listor.

Databasen som användes för detta arbete fungerade bra för de flesta dataserier när datan väl blev förklarad. Dock för vissa dataserier kan databasen optimeras för att göra så mycket arbete som möjligt i SQL-frågan och minska arbetet i Java. Det hade t.ex. varit bra om starttiden för alla körningar fanns för alla tabeller i databasen. På så sätt behöver man inte slå ihop tabeller varje gång man vill få ut starttiden. Eftersom körnummret ofta är det man kopplar samman tabellerna på så borde denna förändring inte vara allt för svår. En annan möjlighet för databasen är att ha en extra tabell där man sparar de medelvärden man vill använda. Just nu räknar man bara medelvärden för en dataserie. Om man i framtiden vill ha fler dataserier som använder sig av någon form av medelvärde kan det vara bra att spara de medelvärden man vill använda direkt i databasen. Detta medför att färre beräkningar behöver göras i Java vilket bör göra systemet snabbare.

Det implementerade diagrammet och listan medför att Verifyter bättre kan se hur datan förändras över tid. De kan också bättre se vilka filer som är med i flest buggar. Eftersom detta arbete fungerar som en prototyp och utvecklingsplatform kan resultatet direkt användas. Diagrammet och listan kommer att fungera som en mall för framtida utveckling. Resultatet från detta arbete kan Verifyter sedan använda för att koppla varje kunds databas till systemet. På så sätt kan Verifyter visa för varje specifik kund hur deras projekt förändras över tid samt var buggar uppkommer.

6.2. Reflektion över etiska aspekter

Då mycket av programmerandet i detta arbete har involverat JavaScript finns det många forum så som Stack Overflow, Githubtrådar mm. som man vända sig till för inspiration. Det är dock viktigt alltid viktigt att tänka på hederskodex, man kan inte bara gå in på forum, hitta lösningar på ens problem och kopiera dem.

Självklart beror detta på vad det är man kopierar. Ifall man har något litet problem t.ex. en algoritm för någon sortering eller liknande känns det inte speciellt viktigt att tänka på hederskodex. Algoritmer och andra små programmeringsmässiga saker är ofta mer allmänt kända och det är näst intill omöjligt att ge erkännande till rätt person. Det kan finnas flera trådar på samma eller olika forum där samma

problem ställs och olika personer ger samma svar. Det är då omöjligt att veta vem som faktiskt kom på ursprungslösningen.

För större problem eller större implementeringar bör man dock ha hederskodex i åtanke. Man kan inte kopiera någons kod utan vidare. Även om kod för en större lösning finns tillgänglig på en allmän sida bör man ge erkännande till författaren av källan ifall man använder sig av den.

För just detta arbete har inga större programmeringsmässiga lösningar tagits från internet. De största delarna som används är de diagram som kommer från NVD3. Dessa diagram har då också refererats till och är gjorda för återanvändning. Annan kod som har tagits från olika källor är t.ex. en två raders kodlösning för att skapa "window-resize". Dessa små kodbitar är kod som ingen specifik individ har kommit på och bör då vara helt okej att använda.

6.3. Framtida utvecklingsmöjligheter

TestHub har i sig många framtida utvecklingsmöjligheter. Bara för detta examensarbete finns det en del utvecklingsmöjligheter.

Som det nämnts tidigare kan datan som visualiseras i listan läggas in i någon form av hierarkiskt diagram. Detta kan tänkas vara Treemap, Sunburst men också ett cirkeldiagram med "drill down" funktion. På så sätt kan man visa för användaren hur de olika buggarna fördelar sig. T.ex. kan man visa fördelningen av buggar överst i filstrukturen. Går man sedan djupare i filstrukturen kommer fördelningen av buggar att ändras beroende på hur djupt man går.

Man kan också lägga till fler diagram av valfri typ. Det enda som krävs är att man lägger till en knapp för det diagram man vill lägga till samt att implementera diagrammet. På så sätt kan man ha flera linjediagram som kan visa olika typer av data som man kan vilja jämföra mellan. Man kan också lägga till fler listor som visar annan data än den listan som redan är implementerad. T.ex. kan man tänka sig vilja visa vem som är ansvarig för vilka buggar istället för att bara visa hur ofta en bug förekommer.

7. Källförteckning

[1] T. Munzner, "*Visualization analysis and design*," Boca Raton, CRC Press, 978-1-4665-0891-0, 2014.

[2] E. Gorodov and V. Gubarev, "Analytical Review of Data Visualization Methods in Application to Big Data," *Journal of Electrical and Computer Engineering*, vol. 2013, , pp. 1-7.

[3] M. Krstajic and A. Keim, "Visualization of Streaming Data: Observing Change and Context in Information Visualization Techniques," in the *IEEE International Conference on Big Data*, pp. 41-47, 2013.

[4] FPPT. 2017. *Treemap Visualization* [ONLINE] Available at: http://www.free-power-point-templates.com/articles/treemap-visualization/. [Accessed 27 March 2017].

[5] Upload.wikimedia.org. 2017. *Sunburst*. [ONLINE] Available at: https://en.wikipedia.org/wiki/Sunburst#/media/File:Disk_usage_(Boabab).png. [Accessed 27 March 2017].

[6] FusionCharts. 2017. *JavaScript Charts: The Definitive Guide*. [ONLINE] Available at: http://www.fusioncharts.com/JavaScript-charts-guide/. [Accessed 27 March 2017].

[7] FusionCharts. 2017. *JavaScript (HTML5) Charting Library Comparisons*. [ONLINE] Available at: http://www.fusioncharts.com/JavaScript-charting-comparison/. [Accessed 27 March 2017].

[8] R. Wang, Y. Perez-Riverol, H. Hermjakob and J. Antonio Vizcaíno, "Open source libraries and frameworks for biological data visualization: A guide for developers," *Proteomics*, vol. 2015, no. 5, pp. 1356-1374. 2015.

[9] M. McDearmon. 2017. *Data Visualization Libraries Based on D3.JS – Mike McDearmon*. [ONLINE] Available at: http://mikemcdearmon.com/portfolio/techposts/charting-libraries-using-d3. [Accessed on 27 March 2017].

[10] L. Sungchul, J. Ju-Yeon, K. Yoohwan, "Performance Testing of Web-Based Data Visualization," in the *IEEE International Conference on Systems, Man and Cybernetics*," pp. 1648-1653, San Diego CA, USA, October 2014.

[11] Google Developers. 2017. *Frequently Asked Questions | Charts | Google Developers*. [ONLINE] Available at: https://developers.google.com/chart/interactive/faq#policy. [Accessed 27 March 2017].

[12] NPMCompare. 2017. *Comparing d3.chart vs. NVD3.js vs. rickshaw vs. vega*. [ONLINE] Available at: https://npmcompare.com/compare/d3.chart,NVD3.js,rickshaw,vega. [Accessed 27 March 2017].

[13] Novus Partners. 2017. *NVD3.JS*. [ONLINE] Available at: http://NVD3.js.org/. [Accessed 02 May 2017].

[14] GitHub. 2017. *novus/NVD3.js*. [ONLINE] Available at: https://github.com/novus/NVD3.js/tree/master/examples. [Accessed 02 May 2017].

[15] J. Strömberg 2017 *List.js*. [ONLINE] Available at: http://listjs.com/examples/new-list/. [Accessed 02 May 2017].

[16] JSON 2017 Json.org. [ONLINE] Available at: http://www.json.org/. [Accessed 05 May 2017].

[17] FasterXML 2017 *ArrayNode (jackson-databind 2.0.0 API)*. [ONLINE] Available at: https://fasterxml.github.io/jackson-databind/javadoc/2.0.0/com/fasterxml/jackson/databind/node/ArrayNode.html. [Accessed 05 May 2017].

[18] FasterXML 2017 *ObjectNode (jackson-databind 2.0.0 API)*. [ONLINE] Available at: http://fasterxml.github.io/jackson-databind/javadoc/2.1.0/com/fasterxml/jackson/databind/node/ObjectNode.html. [Accessed 05 May 2017].

8. Appendix

TABELL VI. SAMMANFATTNING AV DEN TABELL SOM WANG ET AL. TAGIT FRAM ÖVER OLIKA BIBLIOTEK OCH RAMVERK FÖR VISUALISERING. [9] DENNA TABELL ÄR UPPDELAD FÖR TRE KATEGORIER: CHARTS, NETWORKS OCH HIERARCHIES. TABELL TAGEN FRÅN WANG ET AL. [9]

Name	Language	Platform	Type	Supported Categories
D3.js	JavaScript	Web	Open source	Charts/Hierarchies
JFreeChart Orson Charts	Java	Desktop Web	Free	Charts
Google Charts	JavaScript	Web	Free	Charts/Hierarchies
matplotlib	Python	Desktop	Open source	Charts/Hierarchies
MPLD3	Python	Web	Open source	Charts
GRAL	Java	Desktop	Open source	Charts
Jzy3d	Java	Desktop	Open source	Charts
XChart	Java	Desktop	Open source	Charts
Flot	JavaScript	Web	Open source	Charts
Bokeh	Python	Web	Open source	Charts
Cytoscape	Java	Desktop	Open source	Networks/Hierarchies
Cytoscape.js	JavaScript	Web	Open source	Networks/Hierarchies

Gephi	Java	Desktop	Open source	Networks/Hierarchies
Graphviz	DOT, Java, Python, C, C++	Desktop/Web	Open source	Networks/Hierarchies
Sigma.js	JavaScript	Web	Open source	Networks
mxGraph	JavaScript	Web	Commercial	Networks/Hierarchies
JGraphX	Java	Desktop	Open source	Networks/Hierarchies
JUNG	Java	Desktop	Open source	Networks/Hierarchies

Printed by Tryckeriet i E-huset, Lund 2017

LUND
UNIVERSITY

Series of Bachelor's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2017-576

http://www.eit.lth.se