GÖTEBORGS UNIVERSITET

CHALMERS

Ett mjukvarusystem för automatisk och optimal schemaläggning

Kandidatarbete inom Datateknik, Datavetenskap, Informationsteknik

Linnea Andersson
Philip Bogdanffy
Erik Jungmark
Mikael Ragnhult
Yukie Takahashi Boman
Behrouz Talebi

Chalmers tekniska högskola
Göteborgs universitet
Institutionen för Data- och Informationsteknik
Göteborg, Sverige, juni 2015

Hazel
Ett mjukvarusystem för automatisk och optimal schemaläggning

LINNEA ANDERSSON
PHILIP BOGDANFFY
ERIK JUNGMARK
MIKAEL RAGNHULT
YUKIE TAKAHASHI BOMAN
BEHROUZ TALEBI

Supervisor: Emil Axelsson
Examiner: Arne Linde, Jan Skansholm

[Cover: The font used in the logo is called LSLeaves, copyright Lady Sara 2003.
Commercial use allowed.]

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2015

Förord

Denna rapport beskriver ett kandidatarbete som genomfördes av studenter från Chalmers tekniska högskola och Göteborgs universitet under våren 2015.

Vi skulle vilja tacka vår handledare Emil Axelsson för hans råd och stöd under detta arbetet. Vi vill även tacka Dag Wedelin för rådgivning och inspiration till detta projekt, Koen Claessen för hjälpen med SAT-lösare i Haskell, Anton Ekblad för all hjälp med datastrukturer i Haskell, Johan Angervall för värdefulla kommentarer och respons om schemaläggning. Till sist skulle vi även vilja tacka de som hjälpt oss med korrekturläsning av projektet.

Sammandrag

Denna rapport dokumenterar utvecklingen av ett schemaläggande mjukvarusystem vid namn *Hazel*. Detta system består av en server där en automatisk schemaläggare har implementerats samt en webb- och mobilapplikation som låter användare interagera med systemet. Schemaläggningsmjukvaran har utvecklats med det funktionella programmeringsspråket Haskell och använder sig av en SAT-lösare.

Manuell schemaläggning har sedan länge varit ett tidskrävande arbete för personalansvariga, då det ofta är en komplicerad uppgift att matcha ihop organisationens och personalens behov. Det existerar automatiska schemaläggare sedan tidigare, dessa är dock antingen dyra eller skräddarsydda för en specifik organisation. Således har en generell schemaläggare utvecklats som fungerar för olika typer av mindre organisationer.

Abstract

This report documents the development of a software system for automatic scheduling called *Hazel*. This system is comprised of a server where an automatic scheduler has been implemented, as well as a web and mobile application that allows users to interact with the system. The scheduler has been developed with the functional programming language Haskell and uses a SAT-solver.

Manual scheduling has long been the bane of managers everywhere, as matching the needs of the employer with those of the employee is often a difficult and time-consuming task. There already exists several automatic schedulers that attempt to solve this problem. The issue with said schedulers is that thay tend to be expensive, or developed to only fulfill the needs of a particular organisation. As such, this project has developed a generic scheduler that may be used within several types of smaller organisations.

Innehåll

1 Inledning **1**
- 1.1 Syftet med denna rapport . . . 1
- 1.2 Avgränsningar för projektet . . . 1

2 Problembeskrivning **2**
- 2.1 Problemanalys . . . 2
  - 2.1.1 Nurse Scheduling Problem . . . 2
  - 2.1.2 NSP tillhör NP-Hard . . . 3
  - 2.1.3 Problemets utgångspunkt . . . 4
  - 2.1.4 Projektets mål . . . 4
- 2.2 Problemspecifikation . . . 5
  - 2.2.1 Arbetare . . . 5
  - 2.2.2 Tidsblock . . . 5
  - 2.2.3 Arbetspass . . . 6
  - 2.2.4 Regler . . . 6
  - 2.2.5 Önskemål . . . 6
- 2.3 Server och databas . . . 7
- 2.4 Klienter . . . 7

3 Teknisk bakgrund **10**
- 3.1 SAT-lösare . . . 10
- 3.2 Implementerade SAT-lösare i Haskell . . . 10
- 3.3 NSP med SAT-lösare . . . 11
- 3.4 Konjunktiv normalform . . . 11
- 3.5 Android . . . 12
- 3.6 REST . . . 13
- 3.7 SQL . . . 13
- 3.8 Haskell . . . 13

4 Förstudie av schemaläggning i praktik och teori **15**
- 4.1 Val av programmeringsspråk till schemaläggaren . . . 15
- 4.2 Förstudie . . . 15
  - 4.2.1 Relaterad forskning . . . 15
  - 4.2.2 Befintliga system . . . 16
- 4.3 Formell specifikation och modell av problemet . . . 16
- 4.4 Planering och uppföljning . . . 18

5 Utvecklingen av Hazel **19**
- 5.1 Schemaläggare . . . 19
  - 5.1.1 Val av SAT-lösare . . . 19
  - 5.1.2 Implementering av regler . . . 20
  - 5.1.3 Implementering av önskemål . . . 21
  - 5.1.4 Testning av Schemaläggare . . . 22
- 5.2 Server och Databas . . . 22
- 5.3 Mobilklient . . . 24

5.4 Webbklient 26

6 Resultat **27**
6.1 Schemaläggare 27
6.1.1 Regler och önskemål 27
6.1.2 Exempel på testad beräkningstid 28
6.2 Server och databas 28
6.3 Mobilklient 29
6.4 Webbklient 33

7 Metoddiskussion **38**
7.1 Schemaläggaren 38
7.2 Server och databas 38
7.3 Mobilklient 38
7.4 Webbklient 39

8 Resultatdiskussion **40**
8.1 Schemaläggaren 40
8.2 Server och databas 40
8.3 Mobilklient 41
8.4 Webbklient 41
8.5 Framtida utveckling 42
8.5.1 Schemaläggaren 42
8.5.2 Server och databas 42
8.5.3 Mobilklient 42
8.6 Webbklient 42
8.7 Hazel i samhället 43

9 Slutsats **44**

Referenser **45**

Bilaga A Intervju med en potentiell framtida kund **48**

Bilaga B Databasdiagram **49**

Ordlista

API Application Programming Interface, specifikation över hur olika program ska kunna kommunicera med varandra

HTTP HyperText Transfer Protocol, ett vanligt protokoll som används för kommunikation över internet.

JSON JavaScript Object Notation, ett öppet och standardiserat format för transport av data. Används ofta i webbaserad kommunikation.

1 Inledning

Manuell schemaläggning kan vara en mycket tidskrävande uppgift [1]. Organisationens och personalens behov måste tillfredställas samtidigt som arbetslagar måste respekteras. Till exempel får personal inte jobba för länge utan rast, de måste ha en viss vilotid mellan varje arbetspass och endast personal med specialkompetens får jobba på ett visst pass. Att manuellt lägga detta pussel regelbundet kan vara en mycket svår och tidsslukande uppgift.

1.1 Syftet med denna rapport

Syftet med denna rapport är att beskriva utvecklingen av *Hazel* – en webb- och Android-applikation som organisationer kan använda sig av för automatisk schemaläggning. Denna rapport ämnar även ge läsaren insikt i den teoretiska svårigheten med att generera scheman, samt hur detta problem har lösts i Hazel.

1.2 Avgränsningar för projektet

Syftet med projektet är inte att skapa mjukvara som kan ge en lösning för alla scheman inom rimlig tid. Det kommer klargöras i avsnitt 2.1.2 att detta problem saknar en tidseffektiv lösning. Fokus ligger istället på att skapa en lösning som kan ta fram ett schema för en mindre mängd personal inom en rimlig tidsram. Denna lösning ska även vara tillräckligt generell för att kunna användas inom många olika sorters organisationer med varierande krav.

För att produkten ska kunna vara användbar kan dock lösningens tidseffektivitet inte fullkomligt ignoreras. Således definieras det som ett minimikrav för schemaläggaren att den ska kunna schemalägga åtminstone tio arbetare under en månads tid på mindre än en timme.

2 Problembeskrivning

I detta kapitel ska problemet analyseras och omvandlas från naturligt språk till en specifikation. Målet med att lösa problemet med automatisk schemaläggning är att spara tid åt organisationer som i dagsläget schemalägger manuellt. Detta kommer att ske i följande steg:

- Analys av problemet.
- Specifikation av problemet.
- Specifikation av delproblemen.

Således är målet att gå från det presenterade problemet, automatisk schemaläggning, till en tolkning och specifikation. Slutligen ska specifikationen användas som grund för en mjukvarulösning.

2.1 Problemanalys

Schemaläggning inom arbetsplatser tilldelar arbetare arbetspass under en given tidsperiod. Organisationer har olika storlek och krav, vilket betyder att schemaläggning tar olika lång tid beroende på en organisations storlek. Detta problem skalar inte heller linjärt, det vill säga att en fördubbling av problemets storlek mer än fördubblar lösningstiden.

Som ett konkret exempel kan vi betrakta tilldelningen av sjuksköterskor till sjukhusets arbetspass under en månad. Det kan även röra sig om andra scenarion, till exempel där personal, arbete och tidsperiod representerar byggnadsarbetare för ett byggnadsprojekt under två år. För att underlätta utgår detta projekt från exemplet med sjuksköterskor och ser det som ett konkret fall av det generella problemet.

2.1.1 Nurse Scheduling Problem

Ett känt schemaläggningsproblem kallat “Nurse Scheduling Problem” (NSP) [2] är en typ av *beslutsproblem* [3]. Det består av ett antal sjuksköterskor, dag-, kvälls- och nattskift, regler samt önskemål från sjuksköterskorna. Regler, kallade “hard constraints” (hårda restriktioner), måste följas vid schemaläggning och definierar vad som är ett giltigt schema. Önskemål, kallade “soft constraints” (mjuka restriktioner), används för att utvärdera om ett schema är optimalt. Ett schema S är optimalt om inget annat schema S' existerar sådant att S' har fler uppfyllda önskemål än S. Problemets lösning grundar sig i att schemalägga alla skift, följa alla regler och optimera över önskemålen. NSP skiljer sig dock en aning ifrån det problem som presenterats då det endast behandlar fasta skift och inte godtyckliga arbetspass.

2.1.2 NSP tillhör NP-Hard

NSP, samt det allmänna problemet med automatisk schemaläggning, ingår i en klass av problem vilka beskrivs som "NP-Hard" (Non-deterministic Polynomial-time hard) [4]. Utan att gå in alltför djupt i komplexitetsteori kan det påstås att problem inom denna klass saknar en tidseffektiv lösning och är erkänt svåra problem inom datavetenskap. Kortfattat kan NP-hard beskrivas som ett beslutsproblem där en delmängd av en mängd N ska väljas som motsvarar ett bestämt värde. Antal kombinationer som utgör delmängder är då ekvivalent med $2^{|N|}$.

Ett exempel på ett NP-Hard-problem är *delmängd-summa-problemet* som ställer frågan: givet en mängd X av heltal, finns det en delmängd $x \subseteq X$ som kan summeras till ett godtyckligt tal y? Detta kräver att summan av respektive delmängd $x \subseteq X$ räknas ut tills en summa y hittas. Detta saknar en tidseffektiv lösning eftersom varenda delmängd måste kontrolleras vilket handlar om $2^{|X|}$ kombinationer som redan vid $|X| = 20$ överstiger en miljon kombinationer. Följande exempel illustrerar varför NSP klassificeras som ett NP-Hard-problem och varför sådana problem saknar en tidseffektiv lösning:

- Ett sjukhus har 10 sjuksköterskor och ett antal skift under en vecka som omfattar totalt 50 arbetspass.
- Ett schema består utav par *(sjuksköterska, arbetspass)*, för att beskriva vilken sjuksköterska som jobbar på respektive arbetspass.
- Varje potentiellt schema kommer nu innehålla en *kombination* av de $(10 \cdot 50)$ möjliga par som finns. Det vill säga totalt 2^{500} olika scheman.
- Ett giltigt schema kan vara definierat sådant att:
  - Alla arbetspass måste ha en och endast en sjuksköterska tilldelad.
  - Alla arbetspass måste vara med i schemat.
  - En sjuksköterska får inte jobba på pass som överlappar varandra i tid.
- Anta att en dator kan kontrollera giltigheten för cirka en miljon scheman per sekund. Det skulle fortfarande ta datorn längre tid än vad universum existerat för att kontrollera de 2^{500} olika kombinationerna efter giltiga scheman.

Detta exempel visar tydligt varför dessa problem är ansedda som svåra, även för ett sådant här litet exempel. I verkligheten testar inte ett datorprogram eller någon som lägger ett schema manuellt alla möjliga kombinationer. Snarare används *heuristik* för att komma fram till en lösning. Med heuristik menas förkunskap, till exempel genom att ignorera absurda fall eller att nöja sig med den första upptäckta lösningen som uppfyller alla minimikrav.

Ovan exempel går möjligtvis att schemalägga manuellt utan svårigheter, men när antalet anställda och arbetspass ökar blir det dock inte längre möjligt att göra detta inom rimlig tid. Således är svårigheten med denna typ av problem inte att finna lösningar utan att göra detta på ett tidseffektivt sätt.

2.1.3 Problemets utgångspunkt

Utgångspunkten för exempelproblemet som detta avsnitt behandlar liknar NSP i och med att det finns ett visst antal sjuksköterskor, ett sjukhus med arbetspass samt regler och önskemål som reglerar tilldelningarna. Sjuksköterskorna kan vara heltids- eller timanställda samt ha olika kvalifikationer och kompetenser. Arbetspassen kan i sin tur vara olika långa, kräva specialistutbildning eller vara återkommande. Exempel på regler kan vara att alla arbetspass måste fyllas, en viss mängd specialistsjuksköterskor krävs på en viss avdelning, undre och övre begränsningar av sjuksköterskors arbetstimmar eller ett minimum av vilotimmar mellan arbetspass. I nuläget görs detta manuellt av schemaansvarig eller med ett skräddarsytt schemaläggningssystem.

Som tidigare nämnts kan manuell schemaläggning vara tidskrävande. Detta är en nackdel jämfört med automatiskt schemaläggning då en dator kan utföra beräkningar snabbare i jämförelse med en människa. Undantag gäller dock för arbetsplatser med fasta scheman eller arbetsplatser med få arbetare och skift, då de kräver ett fåtal beräkningar som sedan bara kan dupliceras.

För att få en tydligare bild av vad för krav en potentiell användare av Hazel har av systemet utfördes en intervju av en lämplig kandidat (se bilaga A). Den potentiella användaren som intervjuades berättade att de kan lägga så mycket som 3 dagar på att schemalägga 10 arbetsdagar på en arbetsplats med ett 20-tal anställda och ungefär lika många arbetspass per vecka. Detta trots att kunden tar hjälp av digitalt lagrade scheman och kalendrar samt att det finns få restriktioner och önskemål. Vidare kan det vara svårt för en människa att garantera att ett schema är optimalt eller ens giltigt givet olika regler. En större probleminstans (fler sjuksköterskor, regler, önskemål och arbetspass) blir näst intill omöjligt att lösa under rimlig tid genom att "testa sig fram". I senare kapitel beskrivs en del av de verktyg och metoder som används i dagsläget.

2.1.4 Projektets mål

Målet med detta projekt är att utveckla ett mjukvarusystem som på ett generellt sätt ska kunna schemalägga automatiskt och optimalt. Den ska tilldela arbetare till alla arbetspass under en given tidsperiod på ett sätt sådant att inga regler bryts och antalet tillfredsställda önskemål maximeras. Likaså ska den även kunna fungera på flera olika typer av arbetsplatser. Mjukvarusystemet ska kunna erbjuda en tidsvinst gentemot manuell schemaläggning och garantera att scheman är korrekta. Som tidigare nämnts är grundkravet i detta projekt att den slutgiltiga produkten på under en timme ska kunna schemalägga tio arbetare under en månad. Där ska en mängd regler och önskemål ingå som kan reflektera ett realistiskt scenario på en godtycklig arbetsplats.

För att kunna uppnå målet krävs en specifikation av problemet och dess lösning för att sedan implementera detta som mjukvara. Det krävs en tydlig struktur för datan som representerar arbetarna, arbetspassen, reglerna, önskemålen och schemat. Denna data ska sedan mjukvarusystemet kunna lagra och tolka för att senare ge en lösning. Problemanalysen ger en möjlighet att försöka specificera problemet tydligare för att i avsnitt 4.3 kunna göra en formell specifikation. Huvudmålet som tidigare

nämnts kan nu delas in i följande delmål:

- Göra en specifikation av problemet och dess lösning.
- Implementera mjukvara som följer specifikationen.

2.2 Problemspecifikation

Efter att i föregående avsnitt givit en grundläggande analys av problemet specificeras detta nu mer noggrant:

- Givet arbetare, arbetspass, regler, önskemål och ett givet tidsintervall ska ett schema läggas.
- Schemat består av tilldelning av arbetare till arbetspass under det givna tidsintervallet.
- Ett schema är endast giltigt om inga regler bryts.
- Ett schema är optimalt om antalet tillfredsställda önskemål är maximalt.
- Arbetare kan ha kompetenser och önskemål.
- Arbetspass har start- och sluttider samt kan kräva kompetenser.

Lösningen består av ett schema för ett tidsintervall som är giltigt och optimalt. En probleminstans kan resultera i att ingen lösning hittas, endast en lösning hittas eller att det finns flera lösningar.

2.2.1 Arbetare

För att systemet ska vara allmänt och därmed kunna tillämpas på olika arbetsområden krävs det även en generell definition av vad en arbetare är:

- En lista av önskemål för arbete *(vill/vill inte arbeta på datum/arbetspass, . . .)*.
- En lista av *kvalifikationer (körkort, yrkesbehörigheter, licenser, . . .)*.

Denna definition täcker inte nödvändigtvis alla yrken och arbetsområdens krav men målet är att den ska täcka det generella fallet.

2.2.2 Tidsblock

Ett tidsblock representerar en start- och sluttid inklusive datum. Nedan följer en generell definition av ett tidsblock:

- Start- och sluttid *(2015-05-01-08:00:00, 2015-05-01-17:00:00)*.

2.2.3 Arbetspass

Likt föregående sektion bör också definition av ett arbetspass vara så generellt som möjligt. Nedan följer en generell definition av ett arbetspass:

- Start- och slutdatum *(2015-05-01, 2015-08-30)*.
- Information om huruvida passet är återkommande *(ett tillfälle, varje vardag, varannan vecka, ...)*.
- Ett eller flera tidsblock beroende på om det är ett återkommande arbetspass.
- Det minimala och maximala antalet arbetare som arbetspasset kräver.
- En lista av kvalifikationer som krävs för att få jobba på detta arbetspass *(körkort, yrkesbehörigheter, licenser, ...)*.

2.2.4 Regler

Regler för ett schema är till för att begränsa tilldelningar av arbetare till arbetspass. En del av dessa är självklara och det ses därför som ett krav att implementera dem i Hazel. Dessa regler är följande:

- En arbetare kan endast jobba på ett tidsblock samtidigt.
- Ett arbetspass måste minst ha det antalet arbetare som angetts som minimum.
- Ett arbetspass måste som mest ha det antalet arbetare som angetts som maximum.
- Ingen arbetare får jobba mer än ett visst antal timmar i sträck.
- Arbetare som schemaläggs till ett arbetspass som kräver kvalifikationer måste ha dessa kvalifikationer.

2.2.5 Önskemål

Önskemålen för ett schema används för att optimera schemat så att så många arbetare samt schemaansvariga som möjligt blir nöjda. Önskemålen kan ha en prioritetsnivå som definierar hur viktiga dessa är. Ett krav för projektets slutresultat är att man ska kunna lägga till de mest självklara önskemålen, vilka är följande:

- En arbetare vill respektive vill inte jobba på ett visst tidsblock, arbetspass, eller en viss tidsperiod.
- En schemaansvarig (vidare kallad *administratör*) vill prioritera arbetare till ett visst tidsblock, arbetspass eller en viss tidsperiod.

Ett schema är optimalt om det tillfredsställer det högsta antalet möjliga önskemål.

2.3 Server och databas

Slutmålet med Hazel är en onlinetjänst som ska vara tillgänglig från olika klientapplikationer. Detta är önskvärt då det tillåter administratörer att hantera och inspektera scheman oavsett var de är samt tillåter arbetare att enkelt få tillgång till sitt schema.

För att kunna erbjuda en onlinetjänst krävs en server som ger klienterna tillgång till den schemaläggande mjukvaran. Detta ska uppnås genom att servern erbjuder ett *API* som agerar mellanhand för klienterna och schemaläggningsmjukvaran. Detta API ska tillåta de båda klienterna att skicka HTTP-anrop till servern för att begära, manipulera, ta bort eller lägga till information på servern samt begära att en period ska schemaläggas. All information ska lagras i en databas sådan att datan lätt kan hanteras.

Alla användare ska dock inte ha tillgång till all funktionalitet inom API:t. Till exempel ska arbetare inte kunna lägga in nya arbetspass utan en administratörs tillåtelse. Servern ska även kunna skilja på anrop från användare på olika arbetsplatser och endast visa samt tillåta manipulering av den information som är relevant för den användare som skickar anropet. Således behöver servern utföra någon form av autentisering vid varje anrop samt avgöra vad varje användare är tillåten att se och manipulera. Att tilldela rättigheter kan uppnås genom att associera en *åtkomstnivå* till varje användare. Denna åtkomstnivå beskriver huruvida en användare är en administratör, arbetare eller har en annan passande roll.

Data som ska lagras i serverns databas består främst av information som är nödvändig för schemaläggning: arbetare, arbetspass, tidsblock, regler och önskemål. Utöver dessa krävs dock två kategorier till: *kvalifikationer* och *användarkonton*.

Kvalifikationer används för att definiera vad arbetare har för träning, erfarenheter, et cetera samt vad som krävs för att jobba på ett arbetspass. I serverns databas ska dessa lagras som diskret information för sig själva, för att enkelt kunna associera arbetspass och arbetare med existerande kvalifikationer.

För att servern ska kunna autentisera användare som skickar HTTP-anrop är det viktigt att lagra deras inloggningsinformation (användarnamn och lösenord) i ett användarkonto. Varje användarkonto ska även ha en viss åtkomstnivå lagrad, vilken avgör vilka funktioner som en användare har tillgång till inom serverns API. Utöver detta kan varje användarkonto även vara kopplat till en arbetare, så att en användare kan hämta information om sitt personliga schema.

2.4 Klienter

De båda klientapplikationerna, webbapplikationen och mobilapplikationen, ger arbetare och administratörer möjligheten att komma åt det schemaläggande systemet på servern. Målet är att dessa ska kunna hantera scheman och personal enligt problemet som specificerats i tidigare sektioner.

För att en arbetare eller schemaläggare ska komma åt information från servern krävs det att respektive användare har ett konto bestående av användarnamn och lösenord. Detta måste sedan tidigare finnas inlagt i systemet. När användaren med hjälp av klienten vill komma in i systemet och har skrivit in dessa uppgifter i klienten skickas dessa till systemet för verifikation. Om uppgifterna godkänns kommer ett

gränssnitt till schemaläggaren att presenteras för användaren. Beroende på vilken åtkomstnivå den inloggade användaren har, antingen arbetare, administratör eller systemadministratör, kommer användargränssnittet erbjuda olika funktioner. För den som är inloggad som arbetare ska en meny visas som länkar till följande delar:

- Möjligheten att se sitt eget schema.

  I denna del ska arbetaren kunna se ett uppritat schema som visar de arbetspass som denne har tilldelats under en viss period (se avsnitt 2.2). Den ska på ett översiktligt vis presentera schemat för användaren så att start- och sluttider lätt kan överblickas.

- Möjligheten att se arbetsplatsens schema.

  Denna del ska likna den tidigare med skillnaden att denna visar alla arbetspass som är satta för alla arbetare på arbetsplatsen under en viss period. Även de arbetspass som ännu inte har tilldelats någon arbetare ska visas i schemat och skiljas ut visuellt gentemot de som redan har fastlagts. För de arbetspass som ännu inte har fastlagts ska arbetaren kunna ange önskemål (se avsnitt 2.2.5).

- Möjligheten att se alla arbetare på arbetsplatsen.

  Arbetare ska här kunna se kollegors namn, position och grundläggande kontaktuppgifter (se avsnitt 2.2.1).

Om användaren är inloggad med administratörsrättigheter kommer menyn att se annorlunda ut:

- Arbetsplatsens schema är identiskt med det som visas för arbetare.

- Möjligheten att se all personal på arbetsplatsen kommer även att finnas här.

  Skillnaden är att det ska finnas möjligheter för administratören att ta bort eller redigera olika arbetares uppgifter. En administratör ska även kunna skapa nya arbetare och lägga till information om denne såsom inloggningsuppgifter, kontaktinformation och position på arbetsplatsen.

- Möjligheten att hantera arbetsplatsens olika arbetspass.

  Här ska administratören kunna se en lista över arbetspass och grundläggande information (se avsnitt 2.2.3) om dessa. Arbetspass ska kunna gå att klicka på och en ny del visas där mer utförlig information om denna presenteras. Administratören ska även kunna ta bort arbetspass, skapa nya och redigera deras information.

- Möjligheten att se en lista över alla de kvalifikationer (se avsnitt 2.2.1) som finns på arbetsplatsen och möjligheten att skapa nya. Kvalifikationer ska även kunna döpas om och tas bort.

- Administratören ska kunna se en lista med regler (se avsnitt 2.2.4) som gäller för arbetsplatsen. Även här ska administratören kunna skapa, redigera och ta bort dessa.

En användare med den högsta åtkomstnivån, en systemadministratör som arbetar med utveckling eller administration av *Hazel*, ska kunna se en lista över de arbetsplatser som finns sparade, lägga till nya sådana samt lägga till administratörer till dessa.

3 Teknisk bakgrund

I detta kapitel beskrivs den teknik som används i mjukvaran samt en granskning av olika lösningsmodeller som kan användas vid schemaläggning.

3.1 SAT-lösare

SAT-lösarens mål är att kontrollera huruvida ett givet boolesk uttryck kan ge en lösning. Detta görs genom att sätta värden på de variabler det givna uttrycket innehåller. Exempel 1 nedan, kan lösas genom att sätta variabeln a =“sant” och b =“sant”. Däremot är exempel 2 omöjlig att lösa då a inte kan anta värdena “sant” och “falskt” samtidigt.

Exempel 1 : a $\wedge$ b

Exempel 2 : a $\wedge$ $\neg$a

3.2 Implementerade SAT-lösare i Haskell

Det finns olika implementationer av SAT-lösare i det valda programmeringsspråket Haskell, bland annat Satsolver [5], Minisat [6] och Satplus [7].

- **Satsolver**

  Denna SAT-lösare är baserad på en algoritm av Davis-Putnam-Logemann-Loveland [5]. I detta Haskell-bibliotek finns en datatyp som representerar booleska uttryck, vilket gör det enkelt att skapa regler. Det framgår att biblioteket inte är fullt testat då det beskrivs som experimentellt.

- **Minisat**

  I tävlingen *SAT 2005 competition* utsågs Minisat av Pseudo Boolean Evaluation 2005 till den SAT-lösare som löste flest problem, av de tävlande. Det visade sig även att Minisat var en av de SAT-lösare som inte hade många programfel[8]. Minisat är skrivet i programmeringsspråket C, men finns även tillgänglig som ett bibliotek till Haskell. Detta uppdaterades senast januari 2015, vilket var precis innan detta projektet startade. Biblioteket saknar dock utförlig dokumentation.

- **Satplus** Satplus är ett Haskell-bibliotek som har vidareutvecklats från Minisat. Det har utökats med nya funktioner, till exempel möjligheten att maximera antalet sanna variabler i en lista. Detta bibliotek tillåter även användaren att ställa krav på lösningar som SAT-lösaren genererar, till exempel att minst tre av elva variabler måste vara sanna.

3.3 NSP med SAT-lösare

Nedan beskrivs en tidigare utforskad modell för att lösa NSP med SAT (se avsnitt 2.1.1) [9]. Denna går ut på att det finns arbetare, dagar och bestämda skift. Med booleska formler går det att bilda de olika begränsningarna som sedan kan lösas med hjälp av SAT-lösaren. Först följer en förklaring av variablerna som uttrycket kommer att innehålla, därefter visas uppbyggnaden av en *regel*.

Låt $X_{i,j,k}$ vara en variabel sådan att:
i representerar arbetares id, där $0 < i \leq$ antal arbetare
j representerar en dag, där $0 < j \leq$ antalet dagar
k representerar 1 av 4 skift, där $k \in \{1,2,3,4\}$, där 1 står för morgon, 2 för eftermiddag, 3 för natt och 4 för ledig.

Detta innebär att det behövs så många som (antal skift) $\times$ (antal arbetare) $\times$ (antal dagar) olika variabler, det vill säga, alla kombinationer av skift, arbetare och dagar. Givet att vi har exempelvis 4 skift, 10 arbetare och 7 dagar, blir det 280 variabler.

Som nämnts i början av kapitel 3.1 kan de olika variablerna anta värdena sant eller falskt. Om $X_{i,j,k}$ är sann innebär det, i detta fallet, att arbetare i jobbar dag j, skift k. I fallet då $X_{i,j,k}$ är falsk innebär det att arbetare i inte jobbar dag j skift k.

En regel kan representeras av en logisk formel, även kallat ett booleskt uttryck. Ett exempel kan vara en regel som innebär att personalen måste jobba exakt ett av de fyra skiften varje dag. Denna kan delas upp i två delar. En del som innebär att personalen måste jobba minst ett skift om dagen och den andra delen som innebär att personalen får jobba maximalt ett skift om dagen. Nedan följer den första delen följt av den andra delen:

Del 1: $X_{i,j,1} \vee X_{i,j,2} \vee X_{i,j,3} \vee X_{i,j,4}$

Del 2: $\neg X_{i,j,k_1} \vee \neg\ X_{i,j,k_2}$, $\forall\ k_1, k_2$ sådana att $k_1 \neq k_2$

Sedan kombineras del 1 och 2, genom att binda ihop dessa med $\wedge$, för att båda delarna måste gälla. Resultatet blir en boolesk formel som ser ut som följande:

$(X_{i,j,1} \vee X_{i,j,2} \vee X_{i,j,3} \vee X_{i,j,4}) \wedge ((\neg X_{i,j,1} \vee \neg X_{i,j,2}) \wedge (\neg X_{i,j,1} \vee \neg X_{i,j,3})$
$\wedge ... \wedge (\neg X_{i,j,3} \vee \neg X_{i,j,4}))$

3.4 Konjunktiv normalform

Vissa SAT-lösare, till exempel Minisat, tar enbart in ett booleskt uttryck i konjunktiv normalform (KNF). KNF är ett sätt att strukturera en logisk formel. En formel är i KNF om den är av formen:

$$X_1 \wedge X_2 \wedge ... \wedge X_n$$

där varje X_k är av formen $Y_1 \vee Y_2 \vee ... \vee Y_m$, där varje Y_k antingen är en variabel eller en negation av en variabel. Till exempel, följande formel är i KNF:

$$(p \vee q \vee \neg s) \wedge (t \vee \neg s) \wedge (t \vee \neg t)$$

3.5 Android

Mobilklienten till Hazel är utvecklad för det mobila operativsystemet Android. Detta är ett operativsystem som utvecklas av Google och används på moderna mobiltelefoner och surfplattor. Utveckling av applikationer till detta system görs vanligast i programmeringsspråket Java och märkspråket XML som används för resurser [10].

Ett Android-projekt byggs upp i fyra olika delar:

- **Manifest**

  Här definieras metadata till projektet såsom namnet på applikationen, vilka behörighetskrav den behöver från enheten för att kunna köras samt vilken aktivitet som är den första som ska visas när applikationen startas.

- **Gradle**

  Här lagras den information som krävs för att kompilatorn ska kunna bygga projektet, till exempel vilken version av Android-systemet som applikationen riktas till. Applikationer kan välja vilken version av systemet som är den lägsta som krävs och då kommer kompilatorn att kontrollera så att ingen kod finns som enbart stöds i högre versioner av systemet.

- **Java**

  Android-applikationer är som tidigare nämnts vanligtvis skrivna i programmeringsspråket Java. I denna del lagras alla de klasser, gränssnitt och andra filer som rör själva koden.

- **Resurser**

  I denna del lagras applikationens alla olika resurser. De textsträngar som visas för användaren i olika delar i applikationen hämtas från XML-filer i denna del. Dessa har olika namn beroende på vilket språk de är skrivna i. Till exempel mappen *values/* innehåller en fil med strängar på standardspråket engelska och *values-sv/* innehåller en fil med motsvarande textsträngar på svenska. Android-systemet hämtar vid körning rätt textsträngar beroende på vilket språk enheten är inställd på.

  De menyer som visas på olika delar i applikationen lagras även de som XML-filer som innehåller en lista över menyns element och i vilken ordning de ska komma.

  Även de bilder som används i programmet lagras i denna sektion. Samma bild kan lagras i olika upplösning och systemet väljer ut rätt bild som ska användas beroende på vilken upplösning och storlek skärmen på enheten har som applikationen körs på.

Det användargränssnitt som visas på skärmen är designat i olika layouter. Här definieras var olika knappar och textfält och andra element ska befinna sig, vilken storlek de ska ha och hur de ska se ut.

Aktiviteter är det som användaren ser på skärmen på en Android-enhet. En applikation kan ha flera aktiviteter med olika innehåll. När användaren utför en händelse kan till exempel en ny aktivitet startas. Aktiviteter kan visa upp en layout vilket antingen är en XML-fil som hämtas från resurserna eller dynamiskt skapad i Java-kod. [11]

Aktiviteter kan även innehålla fragment. Dessa kan ses som mindre aktiviteter som kan läggas till och tas bort i dess värd-aktivitet. Likt denna kan även fragment visa upp en egen layout [12].

3.6 REST

"Representational State Transfer" (REST) är en modell för utveckling av mjukvara avsedd för onlinetjänster [13]. En av REST-egenskaperna är *klient-server*, vilket innebär att servern inte har någon kunskap om vad som exekveras hos klienten och vice versa. Servern har således ingen kunskap angående hur det grafiska gränssnittet representeras hos användaren, utan svarar endast med data vid HTTP-anrop. Det är sedan upp till klienten att hantera den givna datan från servern på ett valfritt sätt. Dessa anrop sker till så kallade "routes" som är definierade på servern. En route kan liknas vid en stig som ett HTTP-anrop följer, vilket leder till ett visst kodstycke på servern. Om till exempel en användare skickar ett anrop till http://hazel.se/worker och http://hazel.se/task kommer två olika kodstycken köras och således får användaren två olika svar.

3.7 SQL

"Structured Query Language" (SQL) är ett standardspråk för att interagera med databashanteringssystem där alla värden i databasen delas upp i ett antal *tabeller*. System som följer standarden för SQL är baserade på *relationer*, där rader i en tabell kan innehålla värden från andra tabeller. Detta möjliggör att man lagrar diskret information som är kopplad eller beroende på annan information i databasen.

Ett exempel på ett databassystem som följer SQL kallas för Sqlite [14]. En av de stora fördelarna med Sqlite är att det är mycket lättanvänt. Detta till skillnad från andra databas-system, som till exempel Mysql, behöver inte Sqlite köras på en server. En Sqlite-databas består endast av en fil som SQL-förfrågningar kan skickas till.

3.8 Haskell

Haskell är ett funktionellt och typsäkert programmeringsspråk. Funktionella programmeringsspråk skiljer sig från imperativa språk, som till exempel Java, på så sätt att istället för att skriva en serie av instruktioner som ska följas så deklarerar programmeraren en samling *matematiska funktioner*. Dessa funktioner evalueras

endast baserat på de argument som ges till dem, utan något beroende på ett *tillstånd* (state). Dessa egenskaper gör Haskell praktiskt för att modellera teoretiskt svåra problem, då funktioner ofta är väldigt kompakta, samt att det är lättare att resonera kring matematiska funktioner än en serie instruktioner i ett imperativt språk.

4 Förstudie av schemaläggning i praktik och teori

Projektet inleddes med en omfattande planeringsfas där grunden för Hazel utformades. I denna fas fastställdes det allmänt hur schemaläggaren skulle fungera, samt vilken typ av klienter som skulle kopplas till denna. Därefter gjordes efterforskningar kring vilken lösningsmetod som skulle vara lämpligast för att implementera denna schemaläggare. Här modellerades även den data och de funktioner som krävdes.

4.1 Val av programmeringsspråk till schemaläggaren

Från ett tidigt skede var det bestämt att Hazel skulle utvecklas i det funktionella programmeringsspråket *Haskell*, vilket beskrivs i avsnitt 3.8. En av de främsta anledningarna till detta var det starka intresset och att det fanns goda kunskaper inom språket. Dessutom är få applikationer skrivna i funktionella språk [15], vilket sågs som en utmaning.

Dock finns det självklart även annan motivation bakom valet av programmeringsspråk. Som det beskrivs i avsnitt 3.8 är det relativt enkelt att resonera kring kod skriven i ett funktionellt språk, tack vare dess kompakta struktur. Detta var högst önskvärt då det var känt från projektets start att schemaläggning är ett teoretiskt svårt problem.

4.2 Förstudie

Det första steget var att studera olika matematiska lösningsmetoder för denna typ av problem. De huvudsakliga kandidaterna som studerades var *Tabu search*, *SAT*, *Greedy algorithms* samt *Interval Scheduling*. Det fanns vetenskapliga artiklar som påpekade att Tabu search är den optimala lösningen för denna typ av problem [16][17] men den visade sig vara för avancerad för att kunna implementeras i detta projekt med så begränsad tidsram.

Slutligen valdes det att schemaläggaren skulle implementeras med hjälp av en SAT-lösare. De främsta anledningarna till detta var att projektgruppen besatt goda förkunskaper om SAT, samt att det fanns flera implementerade bibliotek för SAT-lösare i Haskell.

4.2.1 Relaterad forskning

I kapitel 1 nämndes syftet och behovet med en automatisk schemaläggare. I detta avsnitt diskuteras forskning som är relaterade till detta problem. “Operations Research” är ett forskningsfält som handlar om att använda avancerade analytiska metoder för att ta bra beslut. Forskningsområdet innefattar bland annat schemaläggning och planering inom organisationer. Ämnet har utforskats i decennier [18] men är fortfarande väldiskuterat och relevant än idag och nya metoder prövas kontinuerligt inom olika industrier. I bageribranschen har en lösning presenterats [19] som inte bara involverar schemaläggningen av personal, utan även planering av produktion och transport.

Forskningsområdet är relaterat till många fält inom bland annat matematisk modellering, algoritmer och beslutsproblem. Tidigare forskning inom schemaläggning

har producerat goda resultat och gett lösningar som är välfungerande [9] och som liknar resultatet av detta projekt. I de efterforskningar som gjorts i projektet kunde det dock konstateras att de flesta lösningar var väldigt specifika för ett yrkesområde och inte generella. De flesta lösningar var dessutom inte implementerade i praktiken utan påvisade en lösning i teorin. En del lösningar kombinerade flera algoritmer för att lösa problemet [20] vilket resulterar i en väldigt omfattande och tidskrävande implementation. För detta projekt valdes att nyttja en enklare modell som endast använde sig av en algoritm på grund av projektets korta tidsram.

4.2.2 Befintliga system

Befintliga automatiserade schemaläggare undersöktes för att se vilka funktioner dessa erbjuder. I dessa identifierades fördelar och brister som gav inspiration för hur projektets mål skulle uppnås och vad som skulle undvikas. De befintliga systemen hade brister i bland annat generalitet och automation. I systemet *Time care* [21] som Göteborgs stad använder inom äldrevården visade det sig att scheman skapas automatiskt men efteråt krävs det fortfarande manuellt arbete av personalansvarig och personal för att göra justeringar och lösa schemakonflikter. En annan brist som uppmärksammades var mjukvara, till exempel *Optaplanner* [22], som enbart erbjuder stöd för fasta skift. Detta är i kontrast till Hazel, där varje administratör ska kunna anpassa ett schemas skift helt valfritt.

Som nämndes i avsnitt 2.1.3 intervjuades en potentiell användare som var insatt inom schemaläggning (se bilaga A). Detta för att få ytterligare information om hur schemaläggning fungerar i verkligheten. Han fick frågor om vilken funktionalitet de var i behov av från en automatiskt schemaläggare. De flesta funktioner som önskades var redan ett krav för Hazel eller för skräddarsydda till kundens behov för ett generellt system som Hazel. Två mer intressanta regler som önskades var att kunna fästa en arbetare vid ett specifikt arbetspass respektive neka en arbetare att jobba ett visst arbetspass.

4.3 Formell specifikation och modell av problemet

En formell specifikation av problemet behövdes för att vidare kunna implementera detta i mjukvara. Utgångspunkten för modellen vilar på den problemspecifikation som tidigare definierats (se avsnitt 2.2). En artikel om modellering av detta problem studerades för att ge inspiration till hur modellen skulle se ut [23]. Här följer den modell som blev resultatet av förstudien:

- Låt K vara mängden av alla definierade kvalifikationer k sådant att:
  $K = \{\, uniktId, namn \,\}$
- Låt A vara mängden av alla arbetare a sådant att:
  $A = \{\, (uniktId, personligInformation, \{w_1 \dots w_n\}, \{k_1 \dots k_n\}\,)$
  $: w \in W, k \in K \,\}$
- Låt T vara mängden av alla tidsblock t sådant att:
  $T = \{(uniktId, starttid, sluttid) \; : sluttid > starttid \,\}$

- Låt P vara mängden av alla arbetspass p sådant att:
  $P = \{ (minArb, maxArb, datumIntervall, g, \{k_1 \dots k_n\}) :$
  $minArb, maxArb \in \mathbb{N}^+, maxArb \geq minArb,$
  $datumIntervall \in (startDatum, slutDatum), slutDatum \in \{datum, \infty\}$
  $g \in G, k \in K \}$

- Låt G vara mängden av information för upprepningar g sådant att:
  $G = \{ (intervall, avstånd) :$
  $intervall \in \{ett\,till fälle, dag, [veckodagar], månad\},$
  $avstånd \in \mathbb{N}^+\}$

- Låt R vara mängden av alla definierade regler sådant att:
  $R = \{ (a, b, d) : a \in A, b \in \{ t \in T, datum, p \in P\}, d \in \{sant, falskt\} \}$
  Där d = $sant$ innebär att a måste jobba på b och d = $falskt$ att a inte får jobba på b

- Låt W vara mängden av alla definierade önskemål sådant att:
  $W = \{ (a, b, d, prio) : a \in A, b \in \{ t \in T, datum, p \in P\},$
  $d \in \{sant, falskt\}, prio \in \mathbb{N}^+\}$
  Där $sant$ innebär att a vill jobba på b och $falskt$ att a inte vill jobba på b

- Låt S vara mängden av alla scheman s där:
  $s = \{ (a,t) : a \in A, t \in T \}$

- Låt gen vara en sådan funktion att:
  $gen(startDatum, slutDatum, p) = \{(t_1, \dots, t_n) : t \in T, p \in P \}$
  Där alla tidsblock t inom intervallet $startDatum$-$slutDatum$ som existerar genereras.

- Låt $Hazel$ vara en schemaläggande funktion sådant att:
  $Hazel(startDatum, slutDatum, a, p, r) = s :$
  $s \in S, a \in A, p \in P, r \in R$
  Notera att $Hazel$ kan generera ingen eller en av de möjliga lösningarna.

- Ett schema är giltigt om och endast om:
  - Schemat tilldelar inom ett intervall mellan det minsta och maximala antalet arbetare som krävs till alla arbetspass:
    $|s| \geq \sum_{p \in P} minArb(p) \cdot gen(startDatum, slutDatum, p),$
    $|s| \leq \sum_{p \in P} maxArb(p) \cdot gen(startDatum, slutDatum, p) :$
    $s = Hazel(startDatum, slutDatum, a, p, r)$
  - Schemat ej tilldelar arbetare tidsblock som bryter reglerna i R:
    $\forall r \in R, r = (a,t, falskt) : \neg \exists s \in S, s = (a,t)$
    $\forall r \in R, r = (a,t, sant) : \exists s \in S, s = (a,t)$
  - En arbetare kan inte jobba på två eller flera tidsblock som överlappar:
    $\forall a \in s, \forall b \in s, s \in S,\ a = (x,t_1),\ b = (x,t_2),\ t_1 \neq t_2,\ t_1 \cap t_2 = \emptyset : \forall t_1\, t_2 \in T$

- Låt en funktion *opto* vara sådan att:
  $opto(W, s) = s' : s \in S, s' \in S$
  Där s' är ett giltigt och optimerat schema givet ett schema och önskemål.

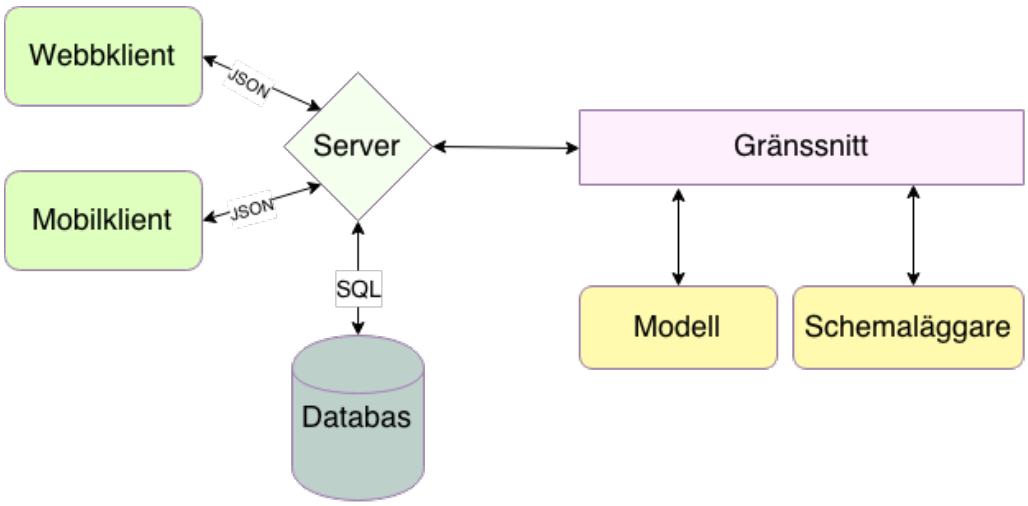
- Ett schema är optimalt om och endast om:
  $\forall s \in Hazel(startDatum, slutDatum, a,p,r), s' = opto(W,s)$

4.4 Planering och uppföljning

När en modell för Hazel hade upprättats skapades en tidsplanering. Projektet delades in i olika ansvarsområden såsom två olika klienter samt en server. Dessa områden delades i sin tur in i olika uppgifter och varje vecka valdes ett visst antal uppgifter ut som skulle ha utförts till nästa vecka – så kallade "sprints". Varje vecka avstämdes utförandet av dessa uppgifter varefter nya delades ut inom gruppen.

5 Utvecklingen av Hazel

Hazel utvecklades i fem olika delar: En modell av schemaläggningsproblemet, en algoritm som beräknar lösningarna för modellen, servermjukvara, samt webb- och mobilklient (se figur 1). De olika delarna utvecklades först självständigt för att senare kopplas samman.



Figur 1: Figuren visar en övergripande modell över Hazels olika delar och hur de kommunicerar.

Eftersom arbetet utfördes av olika parter som arbetade med projektet på olika håll krävdes det en gemensam plats där projektets filer kunde synkroniseras. För detta ändamål användes den decentraliserade lagringstjänsten *Github* som bygger på versionshanteringssystemet *Git*. På så sätt kunde alla förändringar i koden spåras över tid.

5.1 Schemaläggare

Gränssnittet och modellen i schemaläggaren utgår ifrån den formella specifikationen i avsnitt 4.3. Gränssnittet utvecklades på så vis att den anropar funktioner i modellen och SAT-lösaren i en viss ordning, som till slut returnerar ett möjligt schema till servern, om ett schema som följer alla regler existerar. Denna process involverar allt ifrån att förbehandla data och eliminera tilldelningar som inte kan ske enligt reglerna, till att beräkna vilka tidsblock som krockar eller är relevanta för det begärda intervallet.

I denna sektion kommer det beskrivas hur valet av SAT-lösare samt implementering av regler och önskemål gick till.

5.1.1 Val av SAT-lösare

Parallellt med modellens uppbyggnad påbörjades en undersökning av vilken SAT-lösare i Haskell som skulle användas. Det första som gjordes var att försöka använda biblioteket Satsolver i Haskell [5]. Detta var lätthanterligt och implementeringen

kunde påbörjas omgående. Reglerna byggdes på liknande sätt som avsnitt 3.3 beskriver. Eftersom det redan fanns forskning om hur dessa regler kunde skapas blev det enkelt att implementera dem.

Efter att ha rådfrågat Koen Claessen som är forskare inom området [24] beslutades dock att använda en annan SAT-lösare vid namn Minisat [6]. Detta bibliotek (se avsnitt 3.2) visade sig vara mycket mer användarbart i detta projekt då Satsolver saknar en del av den funktionalitet som Minisat har. Dock var dokumentationen om Minisat bristfällig och det krävdes mycket arbete för att kunna använda det.

De båda SAT-lösarna har funktionalitet för att skapa variabler, mata in regler och om möjligt få ut en lösning. Skillnaden är att Satsolver inte kan återgå till ett tidigare stadie om en regel eller ett önskemål läggs in som inte går att uppfylla, vilket i sin tur innebär att ett schema inte kan skapas. Följaktligen behöver allt göras om från början, för att sedan testa med andra önskemål. Detta var ohållbart eftersom det skulle ta för lång tid att generera ett optimalt schema, då alla önskemål måste testas. I Minisat kunde däremot önskemål testas utan att påverka lösningen. Om det visade sig att det gick att lösa problemet med de givna önskemålen kunde dessa sedan läggas in i SAT-lösaren.

En annan skillnad mellan de olika SAT-lösarna var att i Minisat krävs det att de regler som läggs till ska vara i konjunktiv normalform (se 3.4). Därför skapades en funktion som omvandlade reglerna till konjunktiv normalform så att Minisat kunde hantera dem.

5.1.2 Implementering av regler

SAT-lösare använder sig av literaler, vilka är booleska variabler som kan anta värdena 0 eller 1. I Hazels fall symboliserar ett par av en arbetare och ett tidsblock.

$$literal(a,t) = \begin{cases} 1 & \text{om arbetaren } a \text{ arbetar tidsblocket } t \\ 0 & \text{annars} \end{cases}$$

I Minisat finns det en funktion som lägger till en regel genom att binda ihop en given lista av literaler med operatorn $\vee$. Literalerna kan även negeras, till exempel om en person inte fick jobba. I de fallen då literaler behöver sättas samman med $\wedge$ matas listorna in i omgångar till SAT-lösaren. Till exempel, för att mata in $(a \wedge \neg b) \vee c$, där a, b och c är olika literaler krävdes det att dessa omvandlades till konjunktiv normalform, det vill säga $(a \vee c) \wedge (\neg b \vee c)$. Dessa bröts sedan ner till listor i form av $[a, c]$ och $[\neg b, c]$. Dessa listor behandlades sedan i turordning av SAT-lösaren.

Minisat har även möjligheten att enkelt summera antalet sanna variabler. Det är däremot svårare att räkna ut andra begränsningar, till exempel summan av arbetstimmar för en arbetare under en viss period. Detta krävdes när administratör skulle sätta begränsningar med minsta eller maximala arbetstimmar. Den tidigare nämnda forskaren kom med förslaget att använda Satplus, ett bibliotek som forskaren själv håller på att utveckla (se avsnitt 3.2). Detta bibliotek har följande

funktioner som var till stor nytta för att skapa reglerna:

- *fromList*: Givet en lista av par $[(a_1,x_1), ..., (a_n,x_n)]$, där a_k är tal och x_k är literaler, skapas en så kallad *Term* som representerar summan:

$$a_1 \cdot x_1 + a_2 \cdot x_2 + a_3 \cdot x_3 + ... + a_n \cdot x_n$$

- *greaterThan* : Givet x, y , där x är en Term och y är ett tal, lägger in ett villkor som kräver att $x > y$.
- *lessThan* : Givet x, y , där x är en Term och y är ett tal, lägger in ett villkor som kräver att $x < y$.
- *greaterThanEqual* : Givet x, y , där x är en Term och y är ett tal, lägger in ett villkor som kräver att $x >= y$.
- *lessThanEqual* : Givet x, y , där x är en Term och y är ett tal, lägger in ett villkor som kräver att $x <= y$.

Dessa funktioner kunde effektivisera implementationen av regler. Reglerna kunde göras om från att vara stora booleska uttryck till beräkningsuttryck, vilket Satplus kunde lösa mycket snabbare. Ett exempel som kan nämnas är "Ett tidsblock måste ha minst x antal arbetare". Om man skapar denna regel på ett naivt sätt blir det booleska uttrycket både komplicerat och stort. Istället anropades funktionen fromList med en lista som indata, där listan innehöll alla variabler som innehöll det givna tidsblocket, för att sedan få ut en Term. Sedan kunde funktionen greaterThanEqual anropas som lade till regeln: Term måste vara större eller lika med x.

5.1.3 Implementering av önskemål

Efter att ha anpassat reglerna utifrån de nya funktionerna påbörjades arbetet med önskemål (se avsnitt 2.2.5). En funktion för att hantera ett önskemål implementerades, vilket gjorde att arbetarna nu kunde ange vilka dagar, arbetspass eller tidsblock som de ville respektive inte ville, arbeta på. Detta kunde göras enkelt då det enbart handlade om en variabel som motsvarade den specifika personen med det valda tidsblocket. Det uppstod dock frågor om vad ett optimalt schema var. Det diskuterades hur detta kunde göras för att få ett rättvist schema där alla arbetare skulle bli så nöjda som möjligt. Ett flertal utomstående personer rådfrågades och de förslag som kom upp var följande:

- **Förslag 1:** Uppfyll så många önskemål som möjligt.
- **Förslag 2:** Enligt följande algoritm:
  1. Låt $v(i)$ representera antal uppfyllda önskemål för arbetare i.
  2. Låt l vara en lista med alla arbetare som kan få ytterligare ett önskemål uppfyllt.
  3. Om l är tom, avsluta algoritmen.

4. Låt a vara en arbetare, där $a \in l$ och $v(a) = minimum(\{v(i) : i \in l\})$.
5. Uppfyll ett önskemål för arbetare a.
6. Återgå till steg 2.

- **Förslag 3:** Enligt följande algoritm:

1. Låt $a = 1$.
2. Låt x vara antalet arbetare som kan få minst a önskemål uppfyllda.
3. Låt r vara regeln som beskriver att x antal personer måste ha minst a önskemål uppfyllt.
4. Lägg till r.
5. Om $x \neq 0$ återgå till steg 2 och sätt $a = a + 1$, annars avsluta algoritmen.

Det beslutades att implementera förslag 3 eftersom det ansågs vara mest rättvist och enkelt att implementera. Algoritmen krävde ett flertal anrop till en funktion, vid namnet *maximize*, vilken maximerar antalet sanna literaler i en given lista. Denna användes för att maximera antalet arbetare som kunde få x antal önskemål uppfyllda.

Enligt problembeskrivningen (se 2.2.5) skulle önskemål ha olika prioritet. Därför uppfylldes önskemål enligt prioritetsordning. Även administratören kunde lägga in önskemål. De kunde sätta högre prioritet än arbetarna och på så sätt styra schemaupplägget.

5.1.4 Testning av Schemaläggare

Ett testbibliotek till Haskell vid namn *Quickcheck* kan användas för att testa egenskaper av programfunktioner. Biblioteket används för att utföra tester baserat på slumpmässigt genererad data inom definierade intervall. Det var dock svårt och tidskrävande att definiera slumpgeneratorer för arbetare och tidsblock som skulle leda till bra testfall. Testning fick således göras manuellt med testfiler som lästes in och skapade olika tidsblock och arbetare med deras önskemål. Efter att en schemaläggning genomförts verifierades det lagda schemat till en början manuellt då dessa tester inte var stora. Senare skapades ett testprogram som automatiskt verifierade att det lagda schemat var korrekt, det vill säga att alla regler följdes.

5.2 Server och Databas

För att enkelt integrera API:t med schemaläggaren togs beslutet att det skulle utvecklas via ett webb-ramverk i Haskell. Det första som undersöktes var *Yesod* [25]. Dock visade det sig efter nogrannare forskning att Yesod är mer inriktat mot att utveckla en fullkomlig webbapplikation, snarare än det enklare REST-API som skulle utvecklas i detta projekt. Efter vidare undersökningar beslutades det att istället använda webbramverket *Scotty* [26].

Scotty är ett webbramverk som följer mjukvaruarkitekturen REST (se avsnitt 3.6). Det innehåller funktioner för att definiera routes och koppla dem till Haskell-funktioner samt hantera HTTP-anrop. Den främsta anledningen till att Scotty valdes är att det är ett väldokumenterat och lättanvänt ramverk.

Som det diskuterades i 2.3 så utvecklades serverns API för att svara på HTTP-anrop med data. Den överförda datan formaterades i JSON-format för att enkelt kunna tolkas på båda sidor i överföringen. Detta uppnåddes via funktioner i Scotty som utnyttjade Haskell-paketet *Aeson* [27], vilket innehåller funktioner och typklasser för att tolka datatyper i Haskell till JSON-format.

Serverns databas utvecklades med hjälp av *Persistent* [28]. Detta är ett Haskell-bibliotek som tillåter programmeraren att generera en SQL-databas genom att ange ett *databas-schema* (beskrivning av alla tabeller i databasen) baserat på Haskell-datatyper. Persistent kan generera flera olika sorters databaser som följer SQL-standarden, såsom *Postgresql* och *Mysql*. I detta projekt användes dock Sqlite (se avsnitt 3.7). Valet att använda en SQL-databas var naturligt, då Hazels databas skulle innehålla flera diskreta informationskategorier som är relaterade till varandra.

För att kunna autentisera att ett HTTP-anrop kommer från en godkänd användare lagras information om användares inloggningsuppgifter i serverns databas. Det hade dock varit en säkerhetsrisk om lösenord lagrades direkt i databasen, utifall att någon skulle få tillgång till databasen. Det finns även vissa etiska problem med att systemadministratörer för servern skulle ha möjlighet att söka igenom databasen och ta reda på alla användares lösenord. Således lagras istället ett *hash* av användarens användarnamn och lösenord. Ett hash för en text-sträng utvinns från en *hash-funktion*. Hash-funktioner tar en text-sträng av arbiträr längd som argument och returnerar en text-sträng av fix längd, vilken har blivit krypterad på ett sådant sätt att det är i praktiken omöjligt att gå från ett hash till den text-sträng som matades in i hash-funktionen. Hash-funktioner är dessutom *deterministiska*, så att de alltid ger samma resultat för samma text-sträng. För detta projekt användes hash-funktionen *Bcrypt* [29].

I avsnitt 2.3 nämndes det att varje användare endast ska kunna se och manipulera den information i databasen som är "relevant" för användaren. Detta problem löstes genom att associera all information i serverns databas med en *arbetsplats*, inklusive användare. En arbetsplats representerar en organisation som använder sig av Hazel. Således har varje användare tillgång till den information som är associerad med dennes arbetsplats.

Utvecklingen av servern skedde till större delen iterativt. Fokus låg på att få alla funktioner associerade med en viss informationskategori (arbetare, arbetspass et cetera) som diskuterades i avsnitt 2.3 att fungera, innan arbetet fortskred med nästa kategori.

En stor del av arbetet med databasen skedde dock i början av utvecklingen, då det redan från start fanns en grov bild av all data som behövde lagras. Denna första databas planerades via ett *Entity Relationship-diagram* (en grafisk representation av en databas), som sedan översattes till ett databas-schema i Persistent. Dock behövde databasen uppdateras något över projektets gång, allt eftersom att problem upptäcktes och den teoretiska modellen förfinades.

För att försäkra att API:t fungerade korrekt skedde kontinuerliga tester genom att manuellt skicka HTTP-anrop till servern och sedan undersöka den JSON-data som skickades som svar, samt jämföra svarsdatan med den data som fanns på servern.

5.3 Mobilklient

Mobilapplikationen Hazel byggdes först upp med två aktiviteter (se avsnitt 3.5) – en för att logga in och en huvudaktivitet där större delen av interaktionen med användaren skedde. Först skapades en klass som kunde hantera HTTP-anrop till en testserver. Detta för att kunna simulera en inloggning eftersom den riktiga servern ännu inte var funktionell. Manuella tester gjordes för att få fungerande HTTP-GET och HTTP-POST med data i JSON-format.

En klass med enbart en instans, så kallad *singleton-klass*, skapades för att kunna upprätthålla tillståndet i applikationen oavsett vilken aktivitet som var aktiv. Denna kunde då åkallas från både logga in-aktiviteten och huvudaktiviteten. Eftersom huvudaktiviteten skulle kunna visa olika delar i programmet såsom scheman och arbetare krävdes det att en meny implementerades för att användaren skulle kunna navigera till applikationens olika delar. Detta gjordes med hjälp av ett fragment, (se avsnitt 3.5) en så kallad *Navigation Drawer*, vilket är ett vanligt sätt att skapa en meny på i Android-applikationer. För att kunna dela in applikationen i olika delar skapades en mängd fragment för dessa. Det krävdes bland annat ett fragment för att visa arbetare, ett annat för att visa enbart en person och så vidare. I menyn skapades olika alternativ för att navigera mellan de olika delarna. När användaren valde ett alternativ ersattes det aktiva fragmentet i huvudaktiviteten med det som valts i menyn. De delar som skapades utgick från problembeskrivningen (se kapitel 2.4):

- **Schemavisning av eget schema**

  I schemavisningen skulle scheman kunna ritas upp så att användaren lätt skulle kunna överblicka sitt egna schema (se avsnitt 2.4). Dock fanns det inget standardbibliotek för detta i Android-systemet och att skapa ett eget skulle kräva för mycket tid. Dessbättre fanns det ett bibliotek med öppen källkod som var fritt att använda vid namn *Weekview* [30] som erbjöd denna funktionalitet. När detta hade importerats kunde det läggas in i layouten och hanteras av fragmenten. Detta gjorde att användarens schemalagda tider nu kunde ritas upp som olika block i schemat. För att kunna visa ytterligare information om blocket skapades en dialog som visades vid klick på ett schemablock. Här visas exakt start- och sluttid för blocket samt vilka fler arbetare som är tilldelade detta.

- **Schemavisning av arbetsplatsens schema**

  Detta fragment skapades för att visa hela arbetsplatsens schema med hjälp av samma bibliotek som nämndes ovan. För att kunna skilja på vilka block som redan schemalagts och vilka som ännu inte hade tilldelats så ritades de upp i olika färger. De schemalagda blocken färgades gröna och de icke-schemalagda färgades orange. Vid klick på ett tidsblock visades även här en dialog med de exakta start- och sluttiderna och om några arbetare schemalagts till detta visades en lista över detta.

- **Arbetare**

  Här skapades ett fragment vars layout enbart innehåller en lista av arbetare. I denna lista visas arbetarnas namn samt vilken position de har på företaget.

För att kunna visa ytterligare information om arbetare valdes det att skapa ett nytt fragment för detta ändamål. I det nya fragmentet visas arbetarens person- och kontaktuppgifter, kvalifikationer samt deras minsta och högsta antal arbetstimmar per vecka. I detta fragmentets meny skapades alternativ för att redigera och radera nuvarande vald arbetare. Dessa menyalternativ gjordes osynliga om den inloggade inte var administratör.

För att kunna skapa nya och redigera existerande arbetare krävdes det två nya fragment med textfält där information kunde anges om arbetaren. I fragmentet för att skapa en användare lades även möjligheten till att skapa ett användarnamn och lösenord åt den nya arbetaren samt ange vilken åtkomstnivå denna ska få tillgång till.

- **Arbetspass**

  Likt fragmentet för arbetarna skapades i detta fragment en lista över arbetsplatsens olika arbetspass. Detta menyval är enbart synlig för administratörer. Vid val av arbetspass i listan visas information om namn, beskrivning, tider, upprepning, kvalifikation samt minsta och maximalt antalet arbetare. Här lades det även till menyalternativ som möjliggör redigering och borttagning av valt arbetspass. I listan av arbetspass lades det till ett menyalternativ för att skapa nya. Vid val av detta menyalternativ visas ett fragment där administratören kan mata in information om arbetspass. För att välja datum då arbetspasset ska starta och sluta används systemets inbyggda dialog för att välja datum. Även en liknande dialog används för att välja start- och sluttid för arbetspasset. För att välja hur arbetspasset ska upprepas skapades en dialog där olika val fanns huruvida arbetspasset ska repeteras enbart en gång eller dagligen, veckovis eller månadsvis.

- **Kvalifikationer**

  Detta är ett fragment som enbart visar en lista över kvalifikationer och information om hur många i arbetare som har varje kvalifikation. Dessa kvalifikationer kan redigeras och raderas i en meny. Det finns även ett menyalternativ för att skapa nya kvalifikationer. Kvalifikationer visas enbart i menyn för administratörer och är dolt för inloggade arbetare.

Det sista som gjordes var att översätta den engelska XML-filen (se avsnitt 3.5) med applikationens alla textsträngar till en ny fil för svenska motsvarigheter. Med hjälp av detta visas applikationen automatiskt på svenska när enheten är inställd på detta språk annars visas den som standard på engelska.

Applikationen testades regelbundet på en mobiltelefon i och med att nya funktioner skapades. Detta gjordes genom att välja “kör” i utvecklingsmiljön som sedan kompilerar applikationen, för över den till mobiltelefonen och sedan exekverar den där automatiskt.

Applikation testades på version 5 av Android-systemet men den skapades med stöd från version 4.0.3 och högre. Detta gjordes för att applikationen skulle kunna köras på så många möjligheter som möjligt – 96 % av de Android-enheter som finns i världen skulle därmed kunna klara av att köra denna applikation [31].

5.4 Webbklient

En webbapplikation utvecklades för att möjliggöra interaktion mellan användare och Hazel via en hemsida. Webbapplikationen utvecklades i språken HTML, CSS och Javascript, som är en standard för utveckling av webbapplikationer [32]. För modularitet och skalbarhet i det grafiska gränssnittet användes CSS-ramverket *Less*. Detta ramverk har stöd för bland annat variabler och funktioner vilket gör det lättare att uppnå en enhetlig design på gränssnittet [33].

Vidare användes *Angularjs*, ett MVC-ramverk skrivet i Javascript [34]. Angularjs har egenskaper som ger stöd för bland annat förbättrad dynamik och struktur i Javascript-koden genom de egenskaper som finns i MVC-struktur. MVC står för "Model View Controller" och är ett arkitekturmönster för utveckling av mjukvara [35]. Med hjälp av Angularjs och dess MVC-struktur delades gränssnittet upp i olika vyer (views) och det skapades en kontroller (controller) för varje vy. Via respektive kontroller kunde data hämtas och skickas till servern via HTTP-anrop. Den hämtade datan kunde i sin tur skickas vidare till tillhörande vy med hjälp av Angularjs egenskap för tvåvägsbindning [36] mellan det grafiska gränssnittet och Javascript-koden.

För varje vy i gränssnittet skapades det en Less-fil innehållande egenskaper för bland annat utseende och positionering för respektive vys element. *Gulp.js* användes för att kompilera dessa less-filer till CSS-kod samt för att samla all kompilerad CSS-kod i samma CSS-fil för att undvika onödiga importer. Gulp.js användes även för att komprimera och kryptera Javascript-koden som exekveras hos användaren för att förhindra potentiella hackingförsök. Javascript-koden komprimerades för att förminska laddningstid hos respektive användares webbläsare, då Javascript exekveras lokalt på varje användares dator.

För schemauppritning användes ett Javascript-bibliotek vid namn *Fullcalendar* [37]. Med hjälp av Fullcalendar skapades det ett schema i det grafiska gränssnittet. Själva tidsblocken som ritas ut på schemat hämtas från backenden via ett HTTP-GET-anrop från kontrollern som tillhör schemavyn och efter att svaret från servern har anlänt så matas dessa tidsblock in till Fullcalendar-biblioteket som i sin tur lägger till tidsblocken i gränssnittet. Ett schemalagt tidsblock visas med grön färg medan ett icke-schemalagt visas med orange färg.

Javascript-logiken testades med hjälp av manuella testfunktioner med olika typer av indata.

6 Resultat

Målet var att bygga upp ett mjukvarusystem för automatisk schemaläggning (se avsnitt 2.1.4) som skulle kunna användas på en mindre arbetsplats. I dagsläget klarar produkten av detta, även om det finns utrymme för förbättring. Ett mål var även att göra produkten så generell som möjligt, vilket har hållits i åtanke under utvecklingen. Hazel kan nu användas på olika typer av arbetsplatser oavsett om det är ett pappersbruk eller bokning av DJ:s på en nattklubb. Dock är en del regler utformade efter svensk arbetstidslag och passar därmed inte för användning utomlands.

6.1 Schemaläggare

Nedan beskrivs de regler och önskemål som finns tillgängliga för schemaläggaren samt resultat över tester som gjordes för att undersöka hur lång tid schemaläggningen skulle ta.

6.1.1 Regler och önskemål

De regler som finns kan delas in i två olika delar, regler som alltid gäller och valbara regler. De regler som var ett krav för Hazel (se avsnitt 2.2.4) är skapade och finns tillgängliga. Utöver dem finns även valbara regler som den intervjuade önskade (se avsnitt 4.2.2), nämligen:

Valbara regler:

- Fästa arbetare vid ett visst tidsblock.
- Fästa arbetare vid en viss uppgift.
- Fästa arbetare vid en specifik dag.
- Förbjuda arbetare att arbeta på ett visst tidsblock.
- Förbjuda arbetare vid en viss uppgift.
- Förbjuda arbetare att arbeta på en specifik dag.

Utöver reglerna finns även alla de önskemål Hazel hade som krav (se avsnitt 2.2.5). Önskemål prioriteras så att en arbetare kan välja huruvida ett önskemål är viktigt eller inte. Prioriteringar av högre rank väljs före de av lägre rank. Önskemål kan även läggas till av administratörer – till exempel om en arbetare önskas jobba på ett visst pass men vill inte ange det som ett krav.

Då alla önskemål kanske inte kan uppfyllas krävdes det ett sätt att välja ut dessa på ett sätt sådant att prioritet beaktas samt för att försöka uppfylla så många önskemål som möjligt på ett rättvist sätt. Önskemål grupperas därför beroende på prioritet och därefter behandlas önskemålen i rangordning. Först uppfylls ett önskemål för så många arbetare som möjligt. Därefter uppfylls två önskemål för så många som möjligt. På så sätt fortsätter algoritmen tills det inte går att uppfylla

fler önskemål. Detta garanterar inte att maximalt antal önskemål uppfylls utan att så många arbetare som möjligt på ett rättvist sätt ska få sina önskemål uppfyllda.

6.1.2 Exempel på testad beräkningstid

Det är svårt att mäta hur lång tid schemaläggningen tar, då detta beror på vilken typ av schema som ska läggas och hur många giltiga scheman som finns. Till exempel kan ett längre schema med en stor mängd arbetare där många tidsblock, regler och önskemål har specificerats ta mycket längre tid än ett mindre avancerat schema.

Enligt problembeskrivningen (se kapitel 2) var minimikravet att kunna schemalägga tio personer under en månads tid på mindre än en timme. Därför är det dessa mängder som har testats. Schemat som testats är baserat på ett schema som används i verkligheten på ett pappersbruk där det måste finnas arbetare dygnet runt.

I tabell 1 sammanställs den tid det tog att att lägga schemat beroende på olika antal regler och önskemål. Alla fasta regler som specificerades i avsnitt 6.1 är med i samtliga tester. Alla arbetare måste också jobba mellan 38 och 42 timmar i veckan och varje tidsblock har ett specificerat antal minimum och maximum arbetare. Vidare kräver arbetspassen inga kvalifikationer. Önskemålen som använts i två av tre tester består av önskemål från arbetare som vill respektive inte vill arbeta på vissa tidsblock. De valbara reglerna som använts, i ett av de tre testerna, är att en arbetare måste respektive inte får jobba på ett tidsblock.

Tabell 1: Tabell över testade mätresultat. Dessa tester gjordes på en MacBook pro med en Intel Core i5 processor och 8 GB RAM.

Antal personer	Antal tidsblock	Valbara regler	Önskemål	Tid
10	75	inga	inga	24.92 s
10	75	inga	50	37.51 s
10	75	50	50	35.21 s

6.2 Server och databas

I avsnitt 2.3 definierades att användare skulle ha olika åtkomstnivåer. I det slutgiltiga API:t finns det tre åtkomstnivåer: arbetare, administratör och systemadministratör. En arbetare har endast tillgång till att se information på servern, utan att lägga in eller ta bort någon information. Arbetare har även möjlighet att ändra på sina inloggningsuppgifter, samt personlig information. Administratörer har tillgång till att se, lägga in, manipulera och ta bort all information som är kopplad till deras arbetsplats. Systemadministratörer har tillgång till all information som lagras på servern.

Det största problemet med API:t var att finna ett rimligt sett att lagra information om ett schema. Som definierats i avsnitt 4.3 har varje arbetspass information om hur ofta det ska upprepas. Detta kan leda till att ett arbetspass som teoretiskt sett har ett oändligt antal tidsblock om upprepnings-information inte har ett slutdatum. Således sågs det som opraktiskt att lagra alla tidsblock associerade med ett

arbetspass. Istället lagras upprepningsinformationen, samt alla tidsblock för varje arbetspass inom ett visst tidsintervall. Varje arbetsplats lagrar de tidsblock som förekommer inom ett visst datumintervall. Varje gång som en administratör lägger in ett nytt arbetspass genereras alla tidsblock för passet inom intervallet och lagras i databasen. Intervallet uppdateras när en användare skickar ett anrop till servern för att se information om schemat inom en tidsperiod som ligger utanför det nuvarande schemaintervallet. När schemaintervallet uppdateras genereras alla tidsblock för det nya intervallet.

Ett annat problem var att finna ett sätt att lagra regler och önskemål. I schemaläggningsmjukvaran ses både regler och önskemål som booleska formler vilka kan vara väldigt långa. Det sågs som opraktiskt att lagra dessa, så istället utvecklades en underlättande datatyp som schemaläggaren kan tolka om till booleska formler. Denna datatyp kan uttrycka alla önskemål och valbara regler som nämns i avsnitt 6.1. I databasen separeras alla regler och önskemål i tre tabeller, en för varje sorts regel/önskemål. För mer information om databasen, se bilaga B för ett fullständigt Entity Relationship-diagram.

6.3 Mobilklient

Mobilapplikationen utvecklades enligt den specifikation som definierades i avsnitt 2.4. Där står bland annat att en användare måste ange användarnamn och lösenord för att komma åt det schemaläggande systemet. Därför är det första som visas vid applikationens uppstart en inloggningsruta där dessa uppgifter kan skrivas in (se figur 2(a)). Nedanför textfälten visas en kryssruta som möjliggör att inloggningsuppgifterna kan sparas vid en lyckad inloggning. Vid nästa uppstart av applikationen kommer då inloggningen att ske automatiskt. Efter att inloggningen har lyckats visas det schemaläggande gränssnittet för användaren. Navigering sker med hjälp av en meny som länkar till applikationens olika delar. Denna meny ser olika ut beroende på om användaren är inloggad som arbetare eller som administratör. För en användare visas endast de alternativ som är relevanta för denna (se figur 2(c):

- **Möjligheten att se sitt eget schema** (se figur 2(d))

  Här kan användaren se det fastlagda schema som hen har tilldelats av Hazel. Schemat visas som olika block med start- och sluttid indelat i kolumner om veckor. Om dessa block klickas på visas en dialogruta med information om vilka fler som arbetar på detta arbetspass.

- **Möjligheten att se arbetsplatsens schema** (se figur 2(e))

  Här visas hela arbetsplatsens schema. De block som är schemalagda av servern visas i grön färg, de som ännu inte har tilldelats personal visas i en orange färg. De fastlagda blocken går likt ovan att klicka på för att visa ytterligare information.

- **Möjligheten att se all personal på arbetsplatsen**

  Här visas en lista över alla de arbetare som finns på arbetplatsen samt vilken position de har. Om en arbetare klickas på visas ytterligare information såsom kontaktuppgifter för denna.

För en användare som är inloggad med administratörsrättigheter ser menyalternativen annorlunda ut (se figur 2(b)):

- **Möjligheten att se sitt egna och arbetsplatsens schema**

  Dessa alternativ ser identiska ut som de för arbetare. Den enda skillnaden är att det finns ett menyalternativ i arbetsplatsens schema som ger administratören möjligheten att lägga schemat under en valfri tidsperiod.

- **Möjligheten att hantera arbetare**

  Här visas alla arbetare och möjligheten finns att lägga till nya (se 2(g)).

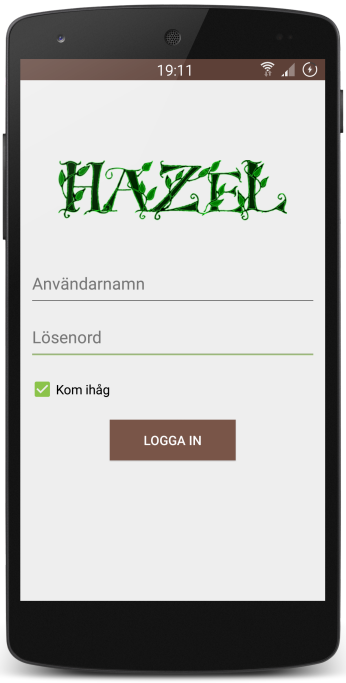
- **Möjligheten att se och hantera kvalifikationer**

  Här visas en lista över arbetsplatsens alla kvalifikationer. Dessa kan redigeras, tas bort och möjligheten finns att skapa nya.

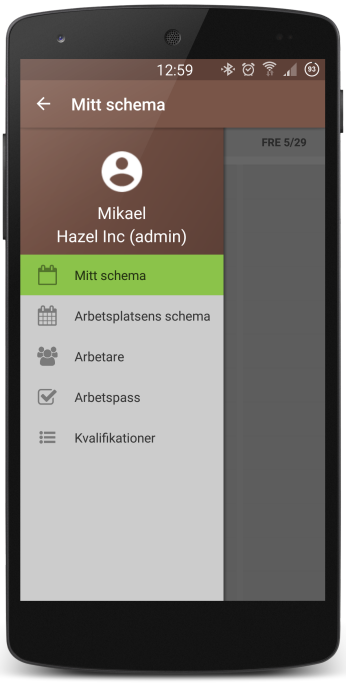
- **Möjligheten att se och hantera uppgifter**

  Denna skärm visar arbetplatsens alla uppgifter, dessa kan redigeras (se 2(h)), tas bort och nya kan skapas.

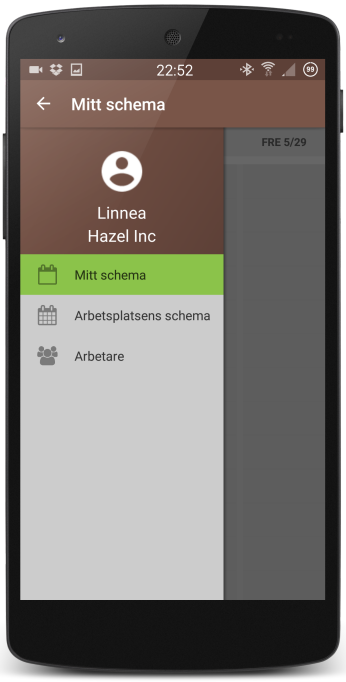
Figur 2: Bilder från utvalda delar i mobilapplikationen



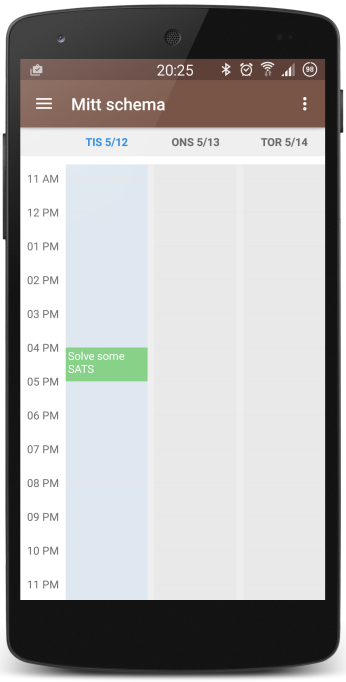
(a) Inloggningen



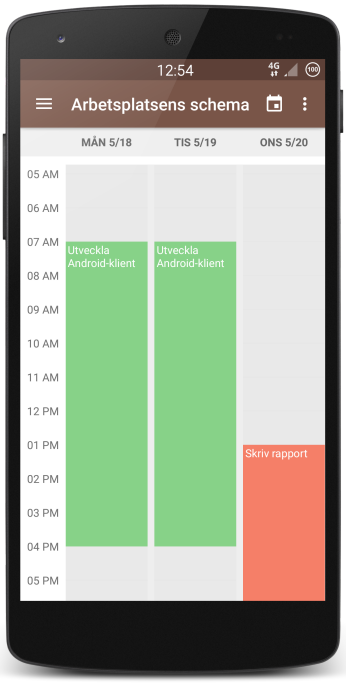
(b) Meny för administratörer



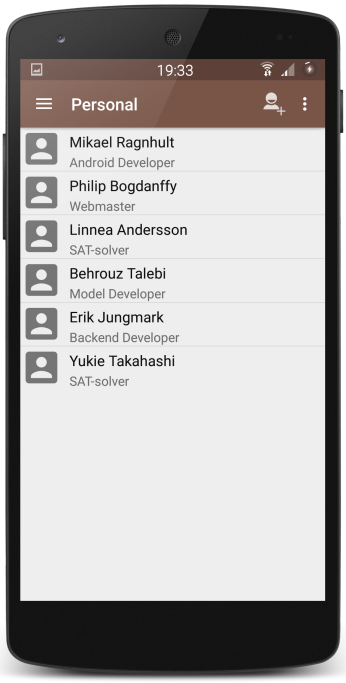
(c) Meny för arbetare



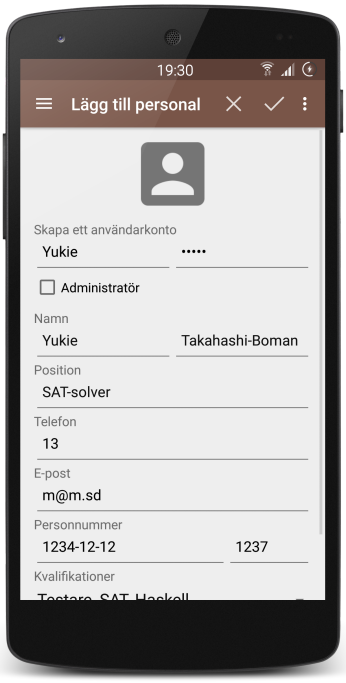
(d) En enskild arbetares schema



(e) Hela arbetsplatsens schema



(f) Alla arbetare



(g) Lägga till en ny arbetare

(h) Redigera en uppgift

6.4 Webbklient

Det grafiska gränssnittet för webbapplikationen består av följande vyer:

- **Login**
  Vid uppstart av webbapplikationen presenteras användaren inför en inloggningsskärm (se figur 3(a)). Gränssnittet för inloggning ser likadan ut för alla typer av användare.

- **Schema**
  När en användare har validerat sig med korrekt användarnamn och lösenord presenteras arbetsplatsens schema. Schemat har fyra olika lägen som illustreras i figur 3(b), 3(c) och 3(d). Längst upp på schemat visas ett verktygsfält med knappar för att växla läge. För användare med administratörrättigheter finns det även en knapp för att schemalägga. När schemaläggningsknappen klickas i visas en ruta där användaren får välja start- och slutdatum för schemaläggningen.

  I figur 3(e) visas schemat med ett markerat tidsblock. Detta läge visar bland annat vilka arbetare som har tilldelats respektive uppgift. Schemalagda tidsblock visualiseras med grön färg medan icke-schemalagda visas med orange färg. Varje tidsblock i schemat är klickbart oavsett vilket läge schemat är inställt på.

- **Sidmeny**
  På skärmens vänstra sida befinner det sig en sidmeny med olika valmöjligheter till att ändra vy. Sidmenyn innehåller olika valmöjligheter beroende på om användaren är en administratör eller arbetare (se figur 4(a) och 4(b)). Bredvid sidmenyn finns en knapp för att visa respektive dölja sidmenyn. När sidmenyn är framhävd får resten av vyn en genomskinlighet (se figur 4(c)).

- **Önskemål**
  Vidare kan en användare med arbetarrättigheter välja de dagar de vill och inte vill jobba. Detta görs med hjälp av en kalender som visas i 5(a).

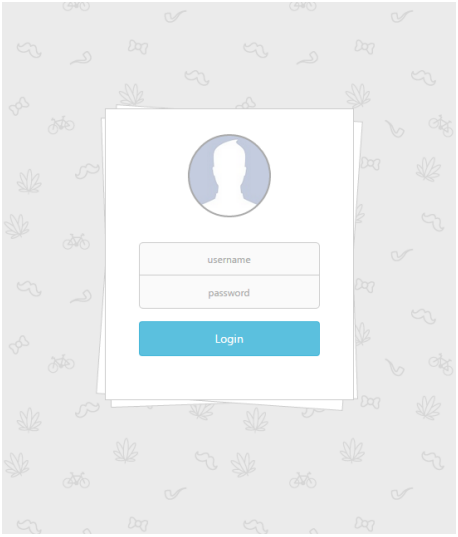
- **Arbetspass**
  För användare med administratörsrättigheter finns det en vy för att skapa, uppdatera och ta bort arbetspass. I denna vy visas all nödvändig information för respektive arbetspass (se figur 5(b)).

- **Kvalifikationer**
  Denna vy implementerades så att en användare med administratörsrättigheter har möjlighet att få reda på vilka arbetare som bemästrar respektive kvalifikationer på ett intuitivt sätt. Här ges även möjligheten att lägga till och ta bort kvalifikationer som krävs för arbetsplatsens uppgifter (se figur 5(c)).

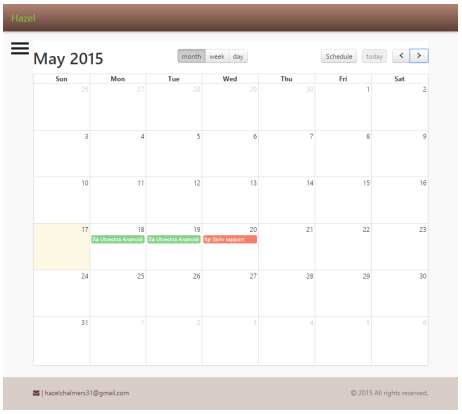
- **Arbetare**
  Arbetar-vyn implementerades så att en användare med arbetarrättigheter endast kan se vilka arbetare som jobbar på samma arbetsplats medan en användare med administratörprivilegier har ett tillägg i gränssnittet som möjliggör uppdatering av befintliga arbetare, borttagning samt tillökning av nya

arbetare. I denna vy presenteras varje arbetares personinfo under respektive arbetares profilbild (se figur 5(d)).

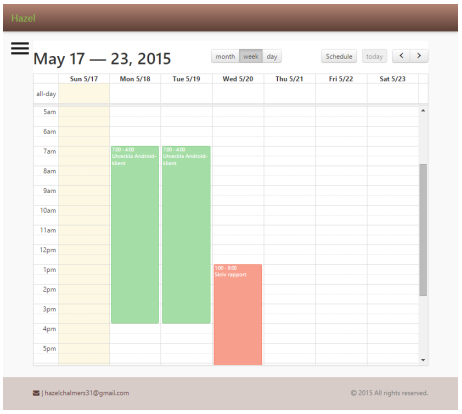
Figur 3: Bilder i webbgränssnittet



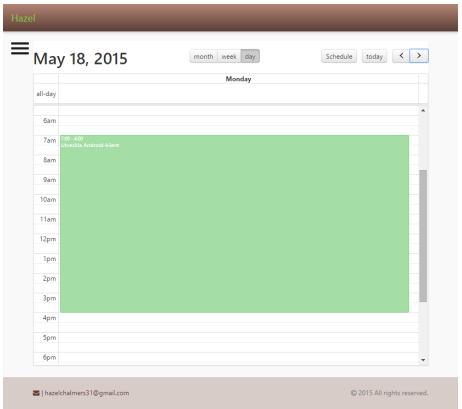
(a) Inloggningsskärm



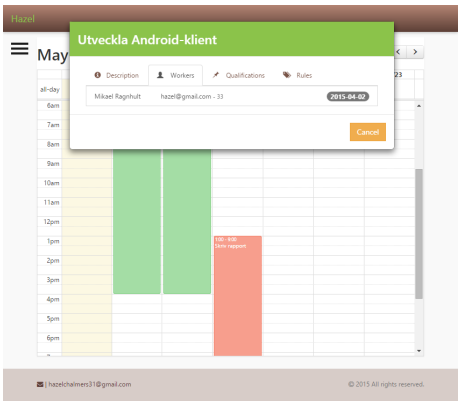
(b) Schema månadsläge



(c) Schema veckoläge

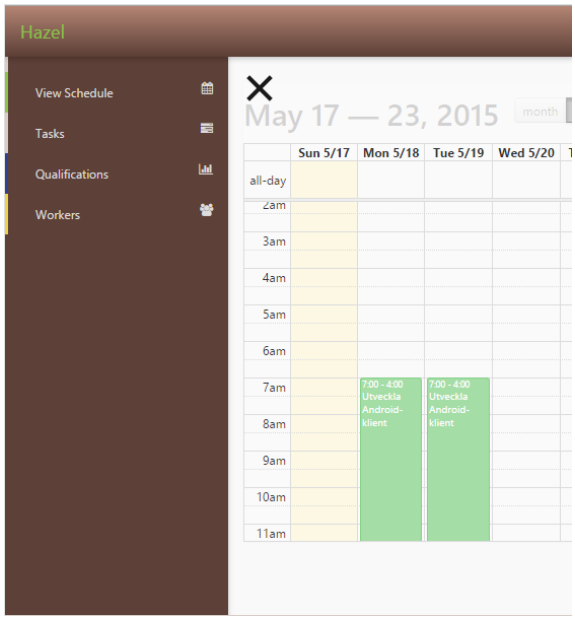


(d) Schema dagsläge

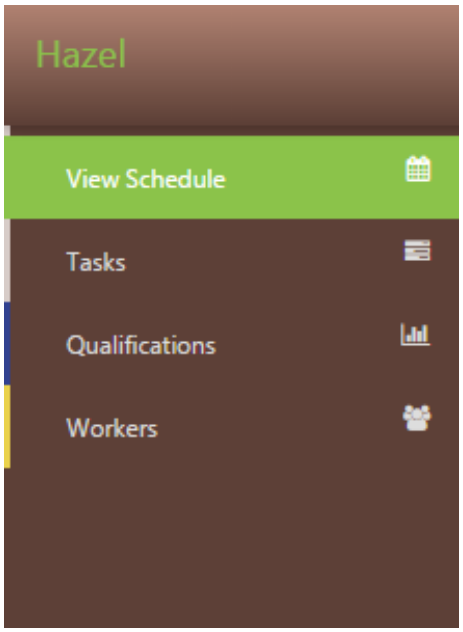


(e) Schema månadsläge med markerat tidsblock

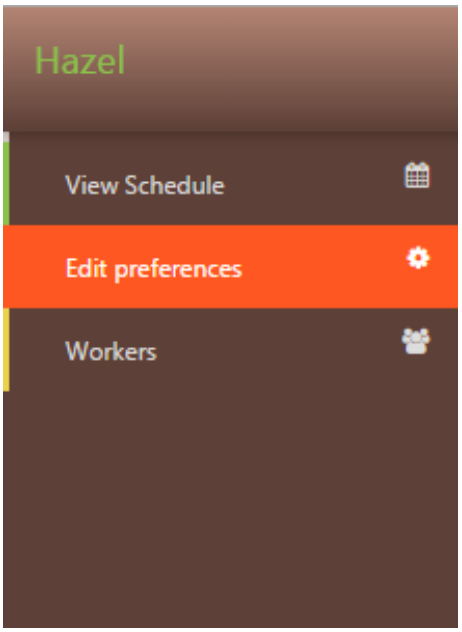
Figur 4: Bilder i webbgränssnittet



(a) Sidmeny framhävd

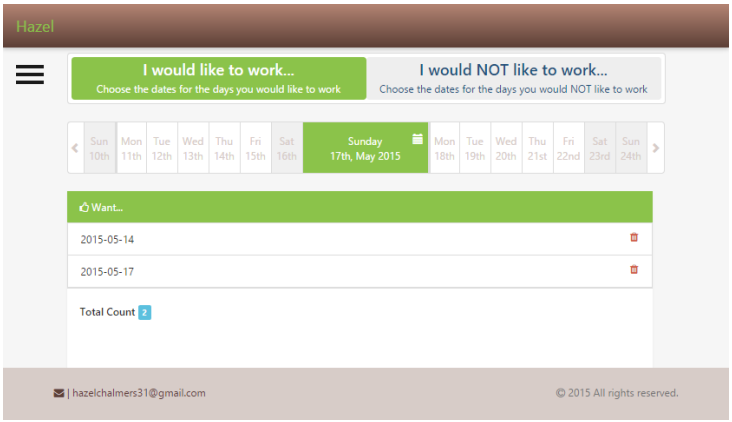


(b) Sidmeny för administratör

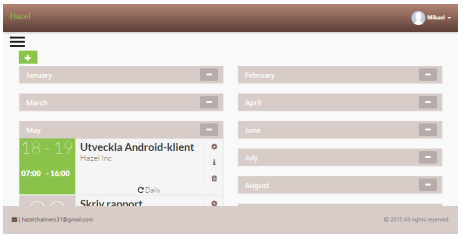


(c) Sidmeny för arbetare

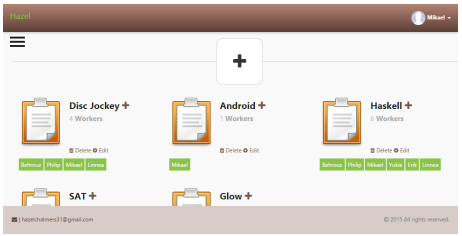
Figur 5: Bilder i webbgränssnittet



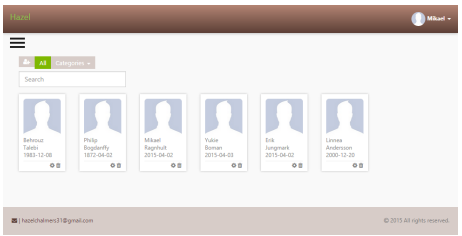
(a) Arbetarpreferenser



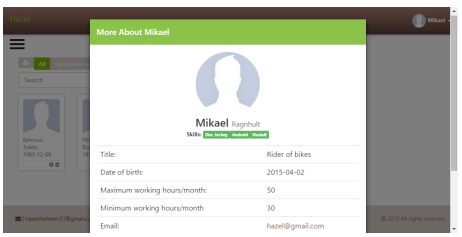
(b) Arbetsplatsuppgifter



(c) Arbetarkvalifikationer



(d) Arbetare



(e) Arbetarinfo

7 Metoddiskussion

Hazel utvecklades i en stegvis och samtida process. Specifikationen av modellen ändrades mycket i början, vilket resulterade i redigering av programkoden i de olika delarna av systemet.

Här diskuteras de val och beslut som togs under projektets gång.

7.1 Schemaläggaren

Valet av det funktionella programmeringsspråket Haskell underlättade utvecklingen avsevärt. Koden blev kompakt i jämförelse med objektorienterad språk som gruppen använt sig av i andra projekt, den funktionella aspekten lämpade sig väl för att implementera funktioner i modellen och det fanns mycket kunniga inom ämnet på institutionen.

Ett av de svåraste designbeslut inom projektet var definitionen av vad som gör ett schema optimalt. I problembeskrivningen (se avsnitt 2.2.5) beskrevs att ett optimalt schema var när så många önskemål som möjligt uppfylldes. Det beslutades dock att välja önskemål med högre prioriteringar före de med lägre, oavsett hur många önskemål med lägre prioriteringar som bröts. Detta innebär att vi nu inte maximerar antalet uppfyllda önskemål. Beslutet gjordes då administratörernas önskemål, som kan ha högre prioritet än arbetarnas, ska gå före alla andra. Det ansågs även rimligt att prioritet i många fall ska gå före antalet önskemål.

7.2 Server och databas

Som nämnt i inledningen för detta kapitel uppdaterades den underliggande modellen för schemaläggaren flera gånger under projektets gång. Detta orsakade svårigheter för utvecklingen av databasen och serverns API, då de var baserade på modellen. Databasen fick i princip göras om från början ett antal gånger för att passa till den uppdaterade modellen. Tyvärr var detta svårt att undvika, men det orsakade signifikanta tidsförluster.

Utöver detta borde även alla användningsfall för API:t ha studerats noggrannare från början. Ett problem som märktes sent i projektets gång hade att göra med hur information om scheman lagras i databasen. Från början var tanken att endast tidsblock som har blivit schemalagda skulle lagras i databasen. När utvecklingen på API:ts funktioner för schemavisning påbörjades insågs det dock snabbt att det finns ett vanligt användningsfall för att kunna se icke-schemalagda tidsblock, både för administratörer och anställda. Denna insikt orsakade en större omstrukturering av databasen, vilket kunde ha undvikits om alla användningsfall hade analyserats mer noggrant i projektets planeringsfas.

7.3 Mobilklient

Den största svårigheten med att utveckla mobilklienten var likt för servern att modellen inte var fullständig innan arbetet påbörjades. Detta innebar att det från en början inte fanns en server att kommunicera med. All kommunikation simulerades

då till en början med testdata, till exempel skapades listor av arbetare och kvalifikationer manuellt. Under utvecklingens gång ändrades även utformningen av olika objekt vilket gjorde att koden till slut blev ostädad. På grund av detta gjordes det i mitten av utvecklingsfasen en omstart och mycket av koden gjordes om från början för att bättre passa ihop med de nya funktionerna på servern. En annan svårighet var att kunna hantera tillståndet och dess sparade data då det på mobila enheter kan förstöras. Detta måste beaktas till exempel vid rotation mellan porträtt och landskapsläge. Då förstör systemet det gamla tillståndet för att skapa ett nytt för att passa den nya rotationen. Det valdes att enbart låsa aktiviteterna till porträttläge eftersom applikationen inte kunde hantera återskapandet av tillståndet på ett felsäkert sätt.

7.4 Webbklient

Det grafiska gränssnittet till hemsidan delades upp i olika vyer för att underlätta andvändning av specifika områden utan att behöva fokusera på flera saker samtidigt.

Schemabiblioteket Fullcalendar valdes främst för att likartade bibliotek med samma funktionalitet kräver licenser för cirka 300$ [38]. Fullcalendar var även lättanvänt och väldokummenterat. Vidare användes ramverket Angularjs för att tidigare erfarenhet fanns inom Javascript-utveckling med hjälp av Angularjs.

En stor del av arbetet gick åt att behandla och formatera datumen som skickades och hämtades från servern på grund av att servern krävde ett visst format på datumen och Fullcalendar ett annat. Detta kunde ha undvikits om funktionaliteten för schemavisning och serverimplementation rörande datum hade skett simultant.

Testningen av webbapplikationens logik och funktioner hade kunnat testas med hjälp av Angularjs men detta ansågs inte ha högsta prioritet då manuella tester kunde utföras utan svårigheter.

8 Resultatdiskussion

Här diskuteras projektets slutgiltiga produkt samt hur väl den lever upp till projektets målsättning.

8.1 Schemaläggaren

Minimimålet, som var att schemalägga tio personer under en månads tid med både önskemål och regler, uppnåddes med god marginal. Detta var dock bara ett minimimål som kan anses vara något pessimistiskt. Eftersom tidigare forskningar hade visat på att SAT-lösaren inte var den snabbaste algoritmen och klarade av alla de utmaningar som problemspecifikationen angav (se kapitel 2.2), anses resultatet ändå vara riktigt bra. Det kan tänkas dock att det finns scheman som tar mycket längre tid än de exempel som står beskrivna i kapitel 6.1.2. Om det exempelvis finns exakt en lösning innebär detta att man måste hitta exakt den lösningen, vilket kan ta tid. De flesta schemaproblem har dock mer än en lösning.

Som nämnts i avsnitt 6.1.2 är det svårt att mäta hur långt tid schemaläggning tar, då detta beror på en mängd faktorer. Genom att studera datan i tabell 1 kan slutsatsen dras att flera regler inte alltid innebär att schemaläggningen tar längre tid. Det hade behövts mer tid för att utforska hur tidskomplexiteten är beroende på regler och önskemål.

Då det kan finnas oändligt många regler valdes det att implementera de som ansågs viktigast. Om fler regler hade implementerats hade SAT-lösaren haft en större mängd data att gå igenom och därmed hade problemet tagit längre tid att lösa. Då en av Sveriges arbetstidslagar implementerades och problemet gick mycket snabbt att lösa, går det att se stor potential för schemaläggare lösta med hjälp av en SAT-lösare i framtiden.

Överlag kan det sägas att schemaläggningen lyckades då den uppfyllde krav för både tid, regler, önskemål och resultat. Det finns självfallet förbättringar som kan göras, men tiden var begränsad i detta projekt.

8.2 Server och databas

Databasen och det tillhörande API:t uppfyllde större delen av de mål som sattes i 2.3, då alla funktioner som hanns med utvecklades. Det största problemet med hur API:t implementerades är att det har väldigt dålig *feltolerans*. Till exempel, om en användare skickar ett anrop till servern om att visa information om ett specifikt arbetspass som ej existerar kommer detta resultera i ett HTTP-felmeddelande, då API:t ej hanterar denna situation väl. Det finns flera andra liknande felmed API:t, såsom att om en period ej går att schemaläggas så meddelar API:t ändå att schemaläggningen är klar, utan att några tidsblock faktiskt schemaläggs.

Det andra stora felet med API:t är att dess säkerhet är något begränsad. Säkerheten på lösenord är acceptabel tack vare bcrypt, dock krypteras ej de meddelanden som skickas mellan server och klient. Detta skulle kunna åtgärdas genom att använda den krypterade varianten av HTTP men det har ej implementerats i API:ts nuvarande form.

Till sist är ännu ett problem att i det slutgiltiga API:t tillåts ej arbetare att lägga in något i databasen, inklusive önskemål. Anledningen till att detta ej implementerades var att det ej finns ett universellt svar på *vilka* önskemål som arbetare ska få lägga in. Då Hazel är ämnat att vara så generiskt som möjligt betyder det att arbetare på olika arbetsplatser skulle ha rättigheter att lägga in väldigt olika önskemål. Till exempel, i den intervju vars sammanfattning finns i bilaga A framkom det att den potentiella kunden ville att hans arbetare skulle kunna specificera vilka arbetspass som de kan och inte kan arbeta på. Uttryckt i Hazels modell så ville han att de skulle kunna lägga in regler om vilka arbetspass som de får arbeta på. Detta är dock inte något som bör gälla på alla arbetsplatser. Då det ej fanns tid till att bygga någon sorts struktur för att avgöra vilka regler och önskemål som är tillgängliga för arbetare på varje arbetsplats implementerades det ingen möjlighet för arbetare att lägga in önskemål.

8.3 Mobilklient

Mobilklienten uppfyllde många av de mål som sattes upp i problembeskrivningen i kapitel 2.4. Det skapades ett användargränssnitt som anpassas beroende på om den inloggade är antingen arbetare eller administratör. Det som dock inte lyckades bli färdigt var följande:

- Möjligheten att hantera regler.
- Möjligheten för arbetare att ange önskemål om arbetspass och dagar.
- Möjligheten att logga in med högsta åtkomstnivå – som Hazel-ansvarig. I detta fall skulle en systemadministratör kunna logga in och hantera klienter, till exempel skapa nya och lägga till administratörer till dessa. Detta måste i dagsläget göras manuellt på serversidan.

Likaså finns det ännu många brister av valideringen av indata. Till exempel kan administratören sätta att fem stycken med en kvalifikation ska jobba på ett arbetspass trots att det kanske inte finns fem stycken på hela arbetsplatsen som har just denna kvalifikationen. Detta på grund av att det inte fanns tid och att det var själva funktionen som var viktigt och inte felhanteringen.

8.4 Webbklient

På grund av tidsbrist skapades det inget grafiskt gränssnitt för användare med högsta åtkomstnivån. Det skapades heller inga vyer för att hantera regler och för arbetare att ange önskemål och arbetspass. Ytterligare saknas validering av data för bland annat att slutdatumet för schemaläggning måste vara senare i tiden än startdatumet.

8.5 Framtida utveckling

8.5.1 Schemaläggaren

Det finns många saker kvar att utveckla hos schemaläggaren, även om den idag fungerar så finns det möjlighet till förbättringar. Fler regler skulle kunna läggas till som tar hänsyn till olika länder och företag. En möjlighet att själv kunna definiera sina regler hade varit optimalt. Idag väljer schemaläggaren bara ett schema även om det finns flera att välja mellan. En förbättring som skulle kunna göras här är att administratören får ett flertal scheman att välja mellan, så att denna blir så nöjd som möjligt. Den kanske viktigaste utvecklingen som hade behövts för att få en riktigt bra schemaläggare är felmeddelanden. I dagsläget skriver den bara ut att det är något fel, i framtiden vore det önskvärt att veta vad som orsakar felet och kanske ett förslag till hur det skulle kunna lösas.

Det finns inga hinder för fortsatt utveckling, bara tiden finns så kommer det kunna bli en mycket användbar schemaläggare.

8.5.2 Server och databas

I framtida iterationer bör feltolerans och säkerhet ökas, då det som nämnt finns vissa besvär inom dessa två kategorier. De routes som saknas för regler och önskemål bör självklart även implementeras. Framtida utveckling skulle även kunna inkludera funktioner som önskades i våra utförda intervjuer, som till exempel att administratörer kan få statistik över hur mycket varje arbetare har arbetat den senaste månaden och dylikt.

Utöver detta finns det lite utrymme för framtida utveckling av API:t, förutom om något skulle läggas till eller förändras i den underliggande modellen.

8.5.3 Mobilklient

Idag är mobilklienten ännu inte komplett, framförallt saknas funktioner för att hantera regler och önskemål. Detta skulle vara önskvärt framförallt eftersom det redan finns implementerat i den schemaläggande algoritmen. En annan funktion som skulle göra applikationen mer användbar vore validering av indata eller möjligheter att se, om användaren skapat ett felaktigt schema, vad som gör att schemat inte kan läggas. Det går till exempel att skapa en användare som bara får jobba en timma i veckan med vissa kvalifikationer. Om det sedan skapas en uppgift på två timmar som enbart tidigare nämnd person har kvalifikationer för så läggs detta till utan varningssignaler. När det senare görs ett försök att lägga schemat kommer servern säga att det är omöjligt utan att ange orsaken.

8.6 Webbklient

I framtiden skulle det vara önskvärt att erbjuda fler funktioner i det grafiska gränssnittet än de som är tillgängliga nu. Bland annat skulle integration med Google Calendar tillfredsställa Hazel-användare då människor skulle vilja samla alla uppgifter i en och samma kalender. Vidare skulle de funktioner som inte uppnåddes i projektets mål implementeras med avseendet att göra applikationen fullständig.

Dessa funktioner är de som nämnts tidigare och innefattar bland annat möjligheten att hantera regler och önskemål.

8.7 Hazel i samhället

Det finns möjligheter att slutprodukten av Hazel skulle kunna användas inom olika organisationer i samhället för att bespara stora mängder tid och resurser. Till exempel inom vården eller andra organisationer och företag som inte har fasta scheman, eller där personal ofta flyttas runt. Om Hazel skulle börja användas av organisationer skulle anpassningar kunna göras för att tillgodose deras specifika behov. Säkerheten skulle även behöva ses över så att all dataöverföring och lagring krypteras, till exempel för att säkra användarnas personliga uppgifter. Tillgängligheten i klienternas användargränssnitt skulle kunna anpassas för att passa användare med speciella behov och Då möjligheten ska finnas för användare att lägga in önskemål i schemat skulle detta kunna leda till att arbetsmiljön förbättrades med hjälp av Hazel. Till exempel skulle de som vill arbeta tidiga pass kunna få den möjligheten och för de som föredrar sovmorgon skulle även dessa önskemål försöka tillgodoses.

9 Slutsats

Projektets syfte var att utveckla ett system för automatisk schemaläggning som är generell och tidseffektiv. I systemet ingick olika delar, såsom två sorters klienter som användare kan komma åt systemet med, samt själva hjärnan i projektet – en schemaläggande server. Att överhuvudtaget automatiskt generera scheman är en ytterst svår uppgift men tack vare hårt arbete samt assistans från forskare på Chalmers kunde det problem som sattes upp lösas. Eftersom svårigheten redan från början var förutsedd sattes målet med detta projekt att kunna schemalägga tio arbetare över en månad på mindre än en timme. Slutresultat blev över förväntan och produkten klarar nu av denna uppgift på ungefär en halv minut. Likaså är produkten uppbyggd så att den lätt kan vidareutvecklas och ny funktionalitet kan läggas till.

För att organisationer ska kunna använda sig av systemet skapades två klienter – en webb- och en mobilapplikation. Dessa behövde även en server att kommunicera med. Att sätta ihop dessa pusselbitar och få dem att fungera ihop på ett bra sätt var även det en utmaning.

Klienterna kan i dagsläget hantera personal och lägga scheman på ett lätthanterligt sätt. En del funktioner kunde inte implementeras såsom hanteringen av vissa önskemål på grund av tidsbegränsningar, men de grundläggande funktionerna lyckades genomföras med ett lyckat resultat.

Detta projekt har varit mycket givande för alla i projektgruppen, då det har gett en stor insikt i vikten av planering och kommunikation inom ett projekt. Det har också visat vilken komplex process mjukvaruutveckling kan vara. Trots detta har gruppen uppnått ett mycket lyckat resultat, där alla varit produktiva med sin insats. Vi har även fått insikt om hur det är att bygga upp ett större system i olika delar såsom: Hur man modellerar problem, hur SAT-lösare fungerar, hur man sätter upp en server och en databas samt hur webb- och mobilapplikationer kan utvecklas till ett sådant system. I framtiden hoppas vi kunna vidare utveckla detta projekt för att få fram en fullt fungerande produkt.

I denna rapport har vi beskrivit utvecklingen av ett schemaläggande systems olika delar; vad som har lyckats och vad som kunde ha gjorts annorlunda. För den som är fortsatt intresserad av schemaläggning finns det mycket vetenskapliga artiklar att studera inom ämnet.

Referenser

[1] S. Hasselqvist. Affärsliv om kostnaden av dåliga scheman [online]. http://www.affarsliv.com/start/fler-nyheter/daliga-scheman-kostar-miljoner, 1 2014. (Visited on 10/05/2015).

[2] J. Rath E. Miller, P. Pierskalla. Nurse scheduling using mathematical programming, June 1975.

[3] M. Sipser. Introduction to the theory of computation, 1996.

[4] D. R. Zanwar M. S. Gondane. staff scheduling in health care systems [online]. http://www.iosrjournals.org/iosr-jmce/papers/vol1-issue6/E0162840.pdf, 2012. (Visited on 06/01/2015).

[5] Haskell-biblioteket data.boolean.satsolver [online]. https://hackage.haskell.org/package/incremental-sat-solver-0.1.7/docs/Data-Boolean-SatSolver.html. (Visited on 04/22/2015).

[6] Haskell-biblioteket minisat [online]. https://hackage.haskell.org/package/minisat-0.1/docs/MiniSat.html. (Visited on 04/22/2015).

[7] K. Claessen. Haskell-biblioteket minisat+ [online]. https://github.com/koengit/satplus. (Visited on 04/30/2015).

[8] N. Sörensson N. Eén. Minisat+ dokumentation [online]. http://minisat.se/MiniSat+.html. (Visited on 04/30/2015).

[9] S. Acharyya S. Kundu. A sat approach for solving the nurse scheduling problem, 2008.

[10] Android developers [online]. http://developer.android.com/guide/components/fundamentals.html. (Visited on 04/05/2015).

[11] Android developers – aktiviteter i android [online]. http://developer.android.com/guide/components/activities.html. (Visited on 05/08/2015).

[12] Android developers – fragment i android [online]. http://developer.android.com/guide/components/fragments.html. (Visited on 05/08/2015).

[13] R. T. Fielding. Vad rest är [online]. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, 2000. (Visited on 05/18/2015).

[14] Om sqlite [online]. http://www.sqlite.org/about.html. (Visited on 04/29/2015).

[15] TIOBE Software BV. Tiobe index for may 2015.

[16] KA. Dowsland och JM. Thompson. Solving a nurse scheduling problem with knapsacks,networks and tabu search. 2000.

[17] U. Aickenlin och KA. Dowsland. An indirect genetic algorithm for a nurse scheduling problem. 2004.

[18] G.J. Rath H.E. Miller, W.P. Pierskalla. Nurse scheduling using mathematical programming, 1976.

[19] AT. de Azevedo FAM. da Silva, AC. Moretti. A scheduling problem in the baking industry, 2014.

[20] J.M. Thompson K.A. Dowsland. Solving a nurse scheduling problem with knapsacks, networks and tabu search, 2000.

[21] Göteborgs stad – timecare [online]. `https://intranat.goteborg.se/wps/portal/int/helastaden/personalingangen/medarbetare/`. (Visited on 04/06/2015).

[22] Optaplanner – mjukvara för att hitta lösningar till nsp [online]. `docs.jboss.org/optaplanner/release/6.2.0.Final/optaplanner-docs/html_single/index.html`. (Visited on 03/01/2015).

[23] I. Blöchliger. Modeling staff scheduling problems, a tutorial, 2004.

[24] K. Claessen. Intervju, April 2015.

[25] Haskell-biblioteket yesod [online]. `https://hackage.haskell.org/package/yesod`. (Visited on 04/22/2015).

[26] Haskell-biblioteket scotty [online]. `https://hackage.haskell.org/package/scotty`. (Visited on 04/22/2015).

[27] Haskell-biblioteket aeson [online]. `https://hackage.haskell.org/package/aeson`. (Visited on 04/22/2015).

[28] Haskell-biblioteket persistent [online]. `https://hackage.haskell.org/package/persistent`. (Visited on 04/22/2015).

[29] Haskell-biblioteket bcrypt [online]. `https://hackage.haskell.org/package/bcrypt`. (Visited on 04/22/2015).

[30] Weekview – bibliotek för att visa scheman i android [online]. `https://github.com/alamkanak/Android-Week-View`. (Visited on 05/08/2015).

[31] Statistik över användandet av olika versioner av android-systemet [online]. `https://developer.android.com/about/dashboards/index.html`. (Visited on 05/08/2015).

[32] w3.org om webb-standarderna html, css och javascript [online]. `http://www.w3.org/community/webed/wiki/The_web_standards_model_-_HTML_CSS_and_JavaScript`. (Visited on 10/05/2015).

[33] Css-biblioteket less [online]. `http://lesscss.org/`. (Visited on 10/05/2015).

[34] Javascript-biblioteket angularjs [online]. `https://angularjs.org/`. (Visited on 10/05/2015).

[35] W3schools – handledning om mvc [online]. `http://www.w3schools.com/aspnet/mvc_intro.asp`. (Visited on 10/05/2015).

[36] Tutorial / 4 - two-way data binding [online]. `https://docs.angularjs.org/tutorial/step_04`. (Visited on 10/05/2015).

[37] Javascript-biblioteket fullcalendar [online]. `http://fullcalendar.io/`. (Visited on 04/29/2015).

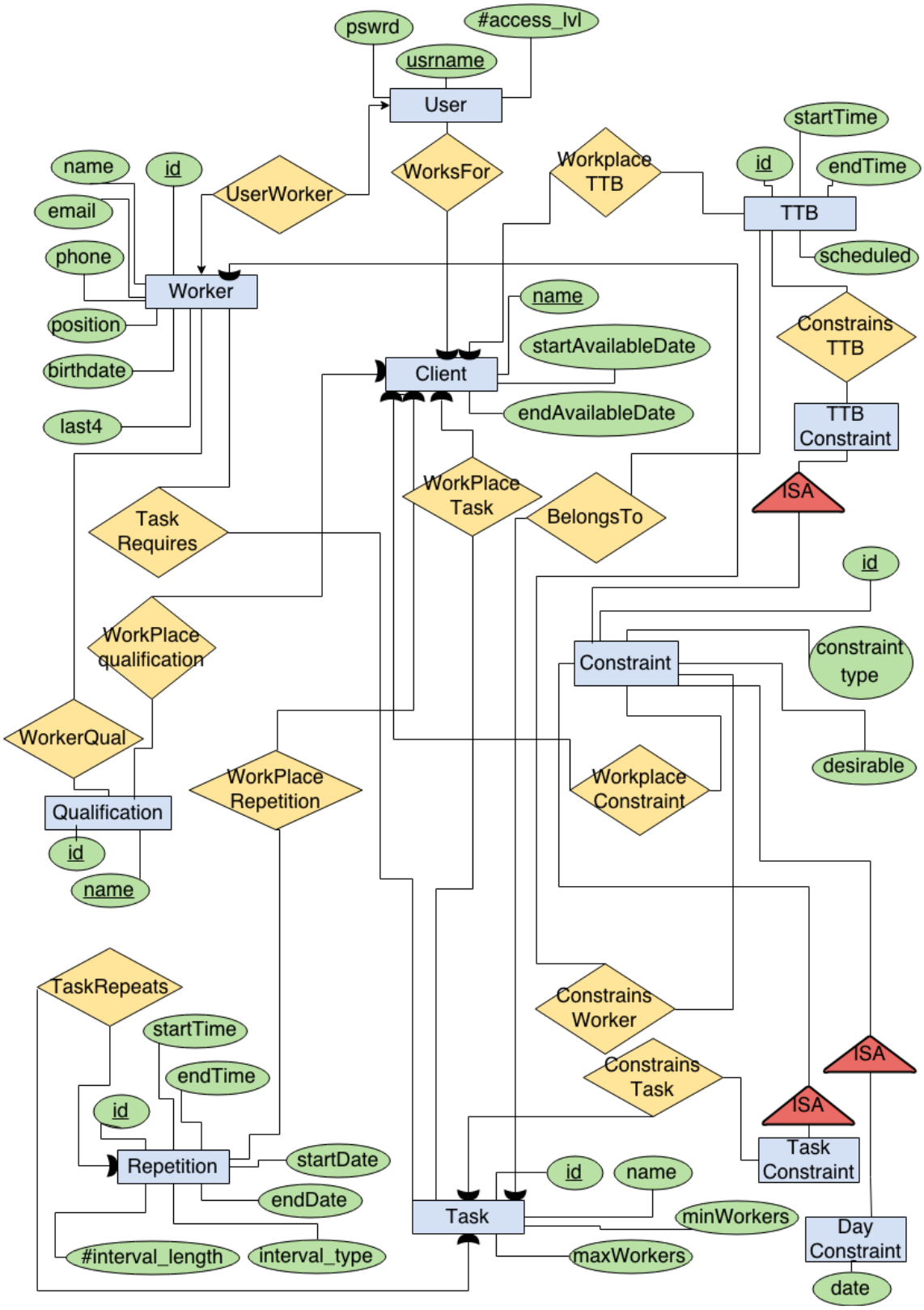
[38] Dhtmlxsuite – bibliotek för schemavisning i webb-applikationer [online]. `http://dhtmlx.com/docs/products/licenses.shtml`. (Visited on 10/05/2015).

Bilaga A Intervju med en potentiell framtida kund

För att få insikt i problematiken med schemaläggning samt vilka funktioner som skulle vara önskvärda i Hazel intervjuade vi Johan Angervall. Johan administrerar ett bemanningsföretag för DJ's gentemot nattklubbar. Detta innebär en mängd unika svårigheter, då det ej är acceptabelt att till exempel hyra ut en DJ med en viss musikstil till en nattklubb med annan musikstil. Utöver digital loggning görs detta manuellt idag genom att koordinera via telefon/sms/sociala medier. En månads schemaläggning kan i nuläget ta allt ifrån 1-3 dagar att färdigställa. Nedan följer en lista på funktioner som Johan önskade sig utav Hazel:

- Begärda regler och önskemål
  - Arbetare ska kunna ha en prioritetslista med tidsblock/arbetsplats.
  - Arbetare ska kunna specifiera vilka arbetspass de överhuvudtaget har möjlighet att arbeta på
  - Tidsblock/arbetsplatser ska kunna ha en prioritetslista med arbetare.
  - Administratör ska kunna bestämma vart och när en arbetare får arbeta.
  - Arbetare ska kunna ange när de kan arbeta.
  - Administratör ska kunna bestämma att en viss arbetare ska arbeta på ett specifikt tidsblock.
- Statistik
  Johan önskade kunna se statistik över diverse saker, såsom hur många timmar varje arbetare har arbetat, samt hur mycket lön varje arbetare har förtjänat. Statistik över hur mycket en nattklubb ska faktureras varje månad är också önskvärt.
- Integration med Google calendar
  För att kunna få en överblick över sin fullständiga kalender önskade Johan att scheman från Hazel skulle kunna exporteras till Google Calendar, eller att scheman från Google Calendar skulle kunna importeras till Hazel.
- Reservpersonal
  Om ett schema ej går att läggas p.g.a. avsaknad av personal skulle det intervjuade partiet kunna kontakta samt använda sig utav en pool av reservarbetare.
- Push-notiser
  Johan skulle vilja kunna skicka push-notiser till arbetares mobiltelefoner via en mobilapp.
- Lön som en faktor i evaluering av optimalitet
  När det finns flera möjliga lösningar på schemaläggningsproblemet ska lönen vara möjlig att vägas in som en faktor.

Bilaga B Databasdiagram



Figur 6: Entity Relationship-diagram som visar databasens struktur