CHALMERS

Databasintegration för småföretag

– en generell lösning för databasintegration

RICKARD PERSSON
JESPER SJÖVALL

Examensarbete
Högskoleingenjörsprogrammet för datateknik
Institutionen för data - och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige, 2012

Abstract

For the majority of all small businesses, IT has become a natural part of the business, from handling of basic functions, to more advanced uses. Despite this dependence on IT, is it rarely used to its full potential. This report describes the process of developing a software that provides a general solution for information management for small businesses through the integration of databases. The report describes the steps of planning, implementation and testing the software. The general solution will be applied in a software that manages a searchable customer and order system. The general solution contains tools for searching, indexing, foreign key, updating, deleting, etc: There are also modules to simplify the construction of GUI and the connection to the underlying database model. The software dosn't require the end user to have knowledge in database management. An application has been developed for a company for managing a searchable customer and order system.

Keywords: Enterprise, IT-solution, Java, MVC+Template, Relational Database.

Sammanfattning

För majoriteten av Sveriges småföretagare har IT blivit ett naturligt inslag i verksamheten från hantering av grundläggande funktioner till mer avancerad användning. Trots detta beroende av IT utnyttjas sällan den fulla potentialen. Denna rapport beskriver processen att utveckla en programvara som tillhandahåller en generell lösning för informationshantering för småföretag genom integrering av databashantering. Rapporten beskriver följande steg: planering, implementering och testning av programvaran. Den generella lösningen kommer att appliceras mot en programvara för att hantera ett sökbart kund- och orderregister. Den generella lösningen innehåller verktyg för sökning, indexering, foreign key, uppdatering, radering, etc. Det finns även grafiska komponenter för att förenkla uppbyggandet av gränssnittet och sammankopplingen till den underliggande databasmodellen. Lösningen medför också att slutanvändaren ej behöver ha kunskaper i databashantering. Som en fallstudie har en applikation för hantering av ett sökbart kund- och orderregister utvecklats för ett valt företag.

Nyckelord: Företag, IT-lösning, Java, MVC+Template, Relationsdatabas.

Förord

Utvecklingen av denna programvara har varit lärorik och då framförallt i skapandet av den generella lösningen vilket ingår i detta examensarbete.

Vi önskar tacka de personer som bistått oss med detta, då främst Alex Löfvgren på Lundh Sails AB som har varit vår företagskontakt och lagt mycket tid och möda på att ge omfattande respons på programvaran. Vi önskar också tacka Uno Holmer på Chalmers Tekniska Högskola som varit vår handledare under examensarbetet, han har coachat oss genom arbetet och åtskilliga gånger läst igenom rapporten och kommit med förbättringar. Vill speciellt tacka Ingmar Jonson som har tagit sig tid för att hjälpa till med rättstavning och grammatik i denna rapport.

Slutligen önskar vi tacka Lundh Sails AB för att vi gavs möjlighet att utveckla en programvara för hantering av kund och order-register.

Innehållsförteckning

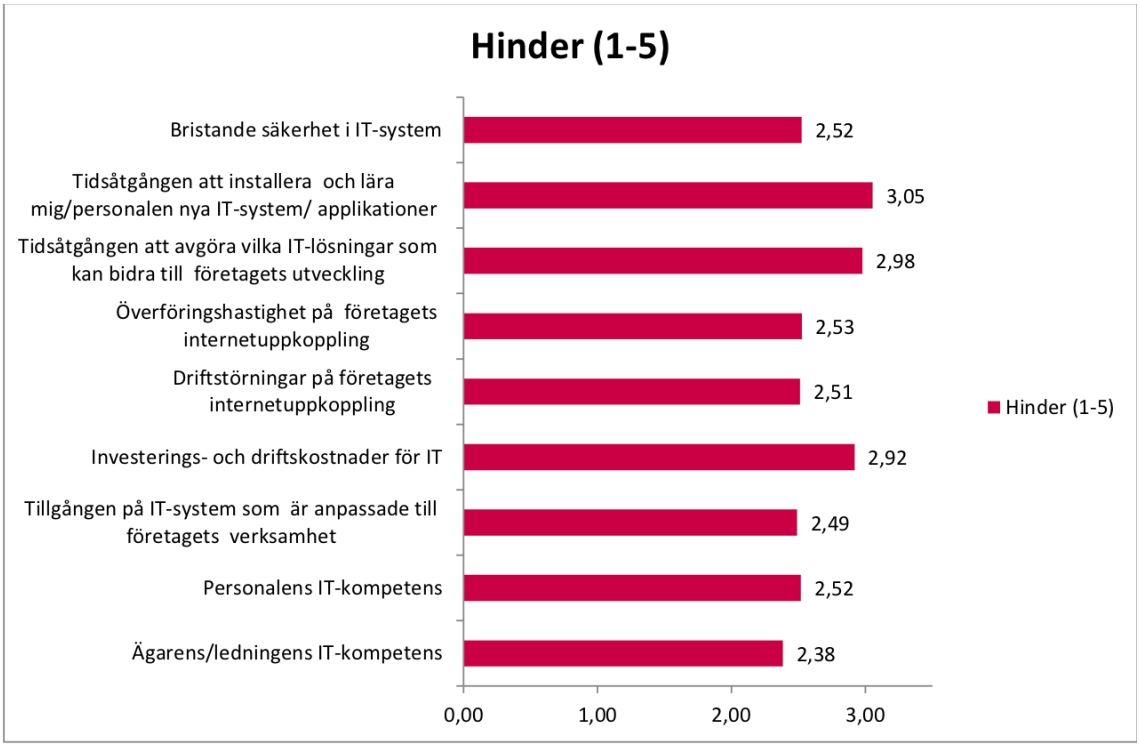
1 Inledning

Detta examensarbete ”*Business Invoice System*” på Chalmers Tekniska Högskola fokuserar på problematiken för småföretag att finna fungerande IT-lösningar som är anpassade efter företagens behov. Examensarbetat har inneburit att utveckla en generell IT-lösning för småföretag. Modellen har även anpassats till Lundh Sails ABs IT-behov.

1.1 Problembakgrund

För majoriteten av Sveriges småföretagare har IT-lösningar blivit ett naturligt inslag i verksamheten för hantering av grundläggande funktioner som inköp, försäljning och kundregister, till mer avancerad användning beroende på företagets inriktning. Trots denna betydande användning av IT, utnyttjas möjligheterna sannolikt inte optimalt för företaget, vilket beror på en mångfasetterad bild av svårigheter, exempelvis bristande kunskaper.

Detta medför att företag kan komma att använda en IT-lösning som blivit föråldrad, men även att verksamheten förändrats vilket medför att de lösningar som var anpassade för företaget tidigare, ej längre uppfyller de krav som ställs på programvaran.



Figur 1: Diagram över vanliga svårigheter för att tillämpa IT hos småföretag. Statens Offentliga Utredning 2012:21

1.2 Problem

Det saknas generella IT-lösningar för hantering av relationsdatabaser som kan användas utan avancerade datakunskaper.

1.3 Syfte

Att skapa en generell lösning som stödjer grundläggande operationer och typer i relationsdatabaser med tillhörande grafiska komponenter för att underlätta uppbyggandet av ett grafiskt gränssnitt.
Att tillämpa denna generella lösning för att skapa ett kund- och ordersystem för Lundh Sails AB.

1.4 Avgränsningar

För att möjliggöra en generell lösning för databaserna har vissa begränsningar gjorts i hur programvaran stöder databasoperationer och -typer. De typer som stöds är heltal, decimaltal, textsträngar, binär data, datum & tid och *foreign keys*. Applikationer kommer endast att hantera transaktioner av typen *dirty reads* vilket medför att pågående överföring kan ångras utan hänsyn till parallella händelser.

Programvaran kommer därför endast att erbjuda stöd för databashanterarna (DBMS) *SQLite* och *MySQL*. Initialt var också *Oracle SQL* planerat att stödjas men beroende på ett flertal faktorer, däribland tidsaspekter under projektet togs beslutet att ej stödja databaser av denna typ.

Applikationen för den generella databaslösningen fokuseras ej på optimering av databashanteringen då programvaran riktar sig mot småföretag där belastningen förmodas vara begränsad.

Lundh Sails – applikationen ska ej hantera fakturering eller liknande monetära transaktioner, pga juridiska aspekter samt att dessa hanteras av annan programvara.

1.5 Disposition

Eftersom problematiken med att integrera IT i en verksamhet är ett återkommande problem för många småföretag, är det viktigt att det finns hjälpmedel för att underlätta detta. Grundtanken med utvecklingen av programvaran som beskrivs i rapporten, är att underlätta för småföretag att utnyttja databaser för intern informationshantering. Behovet av avancerade kunskaper för att administrera databaser, minskas genom att administration kan ske via ett grafiskt gränssnitt.

Programvaran som beskrivs utgörs av två huvudkomponenter, den första (benämnd *Coreside)* utgör en generell lösning på ovan nämnda problem. Den andra delen (benämnd *Frontend*) är en anpassning av *Coreside* till ett kund- och ordersystem för företaget Lundh Sails AB.

Applikationen är i huvudsak utvecklad enligt *Model-View-Controller*-modellen, men genom att databasdelen av *Coreside* utgör modell i programvaran och att källkoden för denna generella lösning ej ska behöva redigeras vid anpassning till företag. Problemet med *MVC*-modellen är att den inte tillhandahåller de specifikationer som efterfrågades. Detta löstes genom att *MVC* utökades till *MVC+T* där *T* står för *Template*. Detta möjliggör att en generell modell kan anpassas för det specifika företaget genom att tillhandahålla en mall för modellen, utan att källkoden för modellen behöver förändras.

För att underlätta skapandet av ett grafiskt gränssnitt och eftersom modellen är känd har ett antal grafiska komponenter utvecklats under projektet. Dessa kan delas upp i två huvudkategorier, de som skapar ett färdigt gränssnitt samt de som används för skapandet av ett gränssnitt. Den första delen utgörs av komponenter som bildar en traditionell tabellvy av data från databasen. Data i denna vy är sökbara. Den andra delen är komponenter som kan användas vid inmatning av data och som ger en återkoppling till modellen, för att underlätta användandet av denna.

Som omnämnts ovan innefattar projektet också utvecklingen av ett kund- och ordersystem för företaget Lundh Sails AB. Detta är utvecklat med en tidigare programvara som utgångspunkt som dock blivit föråldrad. Målet med denna anpassade programvara är att göra det lättare att överblicka samtliga kunder och order men även order hos en specifik kund. Viktigt för denna applikation är även att den skulle uppfattas som lätt att använda, denna aspekt utvecklats genom täta kontakter med företaget.

2 Terminologi

Application Programming Interface *(API)*: Regeluppsättning för hur en viss programvara kan kommunicera med annan programvara.

Array: En lista av fast längd som innehåller objekt av samma typ.

Bibliotek: Avser programdelar som skrivits generellt för att kunna användas i flera olika projekt. Paketeras ofta i en egen fil avskilt från andra delar av projektet för att underlätta hantering.

Boyce–Codd Normal Form *(BCNF)*: Optimeringsmetodik för databasdesign.

Business Invoice System *(BIS)*: Projektnamnet för detta examensarbetat och programvara.

Database Managment System (*DBMS)*: Programvara för modererande av databaser.

Datatyper: Möjliggör lagring av olika typer av data på ett digitalt medium, t.ex. heltal och decimaltal är olika datatyper.

Designmönster: Är typlösningar för vanligt förekommande programmeringsproblem.

Dummy: Dummy kod är ett begrepp som beskriver kod i ett program vars existens ej avses finnas med i slutgiltig version och användas ofta för att ersätta riktiga indata i programmet under utveckling.

Foreign Key: En referens i en specifik relationsdatabastabell till ett värde i en annan tabell i samma relationsdatabas, vilket möjliggör sammanlänkning av tabeller i en databas.

Grafiska gränssnitt *(GUI, Graphical User Interface)*: Möjliggör för en användare att interagera med programmet.

Hårdkodning: Avser att man lagrar centrala inställningar och parametrar i källkoden för ett program vilket förhindrar ändring av dessa i efterhand.

Interface: En given mall i vilken egenskaper hos ett objekt i programmet specificeras.

Integrated Development Environment *(IDE):* Ett program eller programsvit för att utveckla datorprogram i. Innehåller oftast textredigerare, kompilator och debugger.

Java DataBase Connectivity *(JDBC)*: Standard-API för informationshantering mot SQL-databaser via Java.

Komponent: Är en avgränsad del av programmet som oftast utför logiskt relaterade uppgifter.

Model-View-Controller *(MVC)*: Programstruktur för större projekt för att avdela kod i tre logiska element, *model* som utgör modellen för programmet, *view* som visar det som återfinns i modellen och slutligen *controller* som binder samman *view* med *model*.

Model-View-Controller+Template *(MVC+T)*: Utökning av Model-View-Controller modellen där *template* utgör mallen för modellen.

Programstruktur: Övergripande logisk uppbyggnad av ett program, som bestäms både av programmeraren och valt programmeringsspråk.

Query: Term inom SQL och avser en fråga mot en databas, frågan kan t.ex. innehålla vad som önskas hämtas ut och under vilka villkor.

3 Teknisk Bakgrund

Teknisk bakgrund utgör en kort genomgång av de olika teknikerna, verktygen och programmen som har använts under detta examensarbete.

3.1 Designmönster

Större program indelas oftast i programmeringstypiska problem, som kan lösas genom att använda ett designmönster. Ett designmönster är en given idé eller mall för hur man löser ett problem. Att använda ett existerande designmönster är ett bra sätt att både snabba på implementeringsfasen och att underlätta för andra programmerare att sätta sig in i källkoden.

3.1.1 Model-View-Controller

Model-View-Controller (MVC) är en programstruktur som bygger på en uppdelning av ett program i en *Model*-del som ansvarar för underliggande data, beräkningar och i många fall är en teoretisk och matematisk modell av det som programmet avser att utföra. *View*-delen ansvarar för att visa den information som modellen känner till för användaren. *Controller* i MVC hjälper till att samordna informationsflödet mellan *Model* och *View*. [4]

3.1.2 Event-driven programming

Event-driven programming är en programstruktur som bygger på idén att det finns en eller flera källor som skapar händelser som programmet sedan hanterar. En händelse kan vara t.ex. knapptryckningar från en användare, meddelanden från en annan dator eller programvara. Det som sker när en händelse skapas är att en given del av programmet tar emot en händelse och sedan bearbetar eller förmedlar denna händelse till rätt del av programmet. [3]

3.1.3 Factory Pattern

Factory Pattern är ett skapande mönster vars syfte är att skapa ett objekt från argument.

Figur 2: Factory Pattern. *Bearbetning av bilder från wikimedia.org*

En fabrik fungerar på liknande sätt som i figur 2, där ett argument skickas in, som i bilden symboliseras av ett dokument. Sedan processas objektet till önskad slutprodukt.

För att det skall vara möjligt att ha en fabrik som kan skapa fler än ett objekt så krävs det att varje föremål som kommer ut har vissa kända egenskaper. Dessa fördefinierade egenskaper beskrivs inom programmering med *Interface*. I fallet med stolen kan sådana egenskaper vara, färg, vikt och storlek vilket även hade kunnat beskriva ett bord.

```
// Exempel kod på Factory pattern skrivet i Java.
// Det abstrakta interfacet som varje objekt måste stödja.
abstract class Furniture{
    abstract String name();
    // Detta är fabriken som används för att skapa ett objekt.
    static Furniture newFurnitur(String type) {
        if(type.equals("chair")) {
            return new Chair();
        } else if(type.equals("table")) {
            return new Table();
        }
        return null;
    }
}

// Objekt: Stol som har färgen svart.
class Chair extends Furniture {
    String name() { return "Chair"; }
    Color color() { return Color.BLACK; }
}

// Objekt: Bord som har plats för 4 stolar.
class Table extends Furniture {
    String name() { return "Table"; }
    int nrOfSeat() { return 4; }
}
```

3.1.4 Observer Pattern

Observer Pattern är ett lyssnarmönster vars avsikt är att möjliggöra för olika delar av ett program att lyssna på en eller flera händelser. Detta sker genom att en komponent *(Observer)* som är intresserad av en händelse registrerar sig till en komponent som är O*bservable.* När en händelse sker skickar komponenten som är O*bservable* ut information om detta till alla de *Observer* som har registrerats för den specifika händelsen. Mönstret utnyttjas för att komponenten som är O*bservable* ska kunna köras oberoende av om det finns *Observers* registrerade. [15]

3.1.5 Proxy Pattern

Proxy Pattern är ett buffertmönster vars avsikt är att spara data i minnet, istället för behöva läsa/återskapa det igen. Man använder sig av en *Proxy Pattern* när en komponent ofta kommer att anropas. Varje gång som komponenten anropas returneras alltid samma svar. Detta för att inläsningen eller återskapandet av data är långsamt i jämförelse med andra delar av programmet.

Ett vanligt exempel är om programmet läser in en bild som skall visas och genom att buffra bilden i datorns minne kan man undvika att läsa in samma bild från hårddisk mer än en gång. Ett annat exempel är webbläsare som temporärt sparar hemsidor som besökts för att snabbare kunna visa dessa. [7]

3.2 Tester

Tester utförs antingen manuellt eller automatiserat för att verifiera att aktuell programvara fungerar enligt specifikation. Tester kan ske utifrån två perspektiv, *black box* varvid exekverande för en viss metod eller del av applikationen är okänd utifrån testet, det är endast en verifiering av att indata och utdata är korrekta. Det andra perspektivet benämns *white box*, där den interna implementationen är känd och därmed kan verifieras.

3.2.1 Testfall

Ett testfall är en samling punkter som ska genomgås för att pröva en specifik del av programmet, denna typ av test sker vanligtvis manuellt och från ett *black box* perspektiv. Ett specifikt testfall innehåller information om aktuellt test och vad som ska ske, samt vilka felkällor som återfinns.

När testet genomförs, exekveras programmet genom att utföra de åtgärder som återfinns i testfallsdokumenten, därefter dokumenteras utfallet, varvid det framkommer om programmet exekverar som tilltänkt eller ej.

3.2.2 JUnit

Automatiserade tester sker inom programmering för att testa delar av en applikation utan att manuellt behöva utföra var och en av dessa tester. Detta underlättar upprepning av test efter förändringar av koden.

JUnit är ett ramverk för automatiserade tester i Java. Testerna definieras i en specifik klass genom att metoder som ska köras annoteras med `@Test`. Antal metoder med testannoteringen kan vara fler än en och köas i slumpmässig ordning. Därtill möjliggör JUnit att metoder definieras som ska köras före och efter alla tester, detta gör att data kan initieras och på ett acceptabelt sätt avinitieras. Exempelvis kan en anslutning initieras till en databas innan testerna, och sedan kopplas ner när testerna är klara.

Testerna kan genomföras på ett flertal sätt, det första och måhända enklaste är att endast låta testmetoderna anropa de funktioner som ska testas för att se om ev fel uppstår. En annan testmetod som är möjlig genom ramverket är att använda *Assert* vilket innebär att metoder anropas och därefter testa om aktuell utdata är korrekt.

När testerna genomförs kommer testaren att få ut statistik över testerna, utifrån hur många av de som kördes lyckades eller om några misslyckades. De tester som misslyckades åtföljs vanligtvis av en beskrivning om varför ett visst test gav felaktigt resultat, ofta i form av textrepresentation vid den tidpunkt när felet uppstod.[10]

```
public class TestKlass{

  @Test
  public void test1(){
     DB db = new DB();
     // Testa om databas länk är öppen och fungera.
     assertTrue('Kan ej ansluta till databasen', db.isOpen());
  }
}
```

3.3 Databaser

En databas sparar en mängd data under en längre tidsperiod. Relationsdatabaser började bli vanliga efter Ted Codds rapport från 1970. Dessa databaser har data organiserad i tabeller där varje kolumn innehåller en specifik typ av data och där varje rad i tabellen representerar en post. Därtill kan relationer skapas mellan olika tabeller genom att importera värden från en tabell till en annan. Dessa värden benämns som *foreign key.* MySQL och SQLite är exempel på relationsdatabaser. [6]

För att hantera relationsdatabaser används *SQL* (Structured Query Language) vilket är ett standardiserat språk som tillhandahåller ett flertal funktioner för att ändra, lägga till och radera data i databasen. Därtill finns möjligheter att göra sökningar efter specifika data, samt koppla logik till dessa sökningar t.ex. att endast modifiera data som överensstämmer med vissa sökkriterier eller endast acceptera tillägg till databasen om värdet inte redan existerar.

Speciellt för relationsdatabaser är att endast vissa väldefinierade typer får lagras. Dessa typer kan t.ex. definieras som numeriska värden, textsträngar, datum och tid, samt binär data. Den binära datan kan representera komplexa objekt som bilder och filmer men i databasen sparas den endast i form av ettor och nollor. [9]

Exempel på SQL syntax.
Sökning

```
SELECT * FROM tabell WHERE villkor ORDER BY sortera;
```

Insättning

```
INSERT INTO tabell (column1, column2, ...)
   VALUES (värde1, värde2, ...);
```

Uppdateringar

```
UPDATE tabell SET column1=värde, column2=värde2 WHERE villkor;
```

Borttagning

```
DELETE FROM tabell WHERE villkor;
```

Sammanslagning av tabeller

```
SELECT * FROM tabell1 INNER JOIN tabell2 WHERE villkor;
```

3.3.1 SQLite

SQLite är en relationsdatabas som används tillsammans med SQL. Denna är i relation till andra databaser av samma typ, fast enklare byggd. Följden av detta är att den blir enklare att använda och genom den minskade komplexiteten förbättras pålitligheten. Nackdelen blir att SQLite inte är optimerad för större datamängder, största teoretiska storleken är två terabyte, men i många fall begränsas av andra faktorer så som begränsningar i filstorlek hos filsystemet, diskaccess, etc.

Hur databasen läser och skriver till databasen påverkar också lämpligheten vid fleranvändarsystem. När data skrivs till databasen kommer detta att medföra att hela databasen låses vilket medför att skrivningar till andra delar av databasen ej kan genomföras.

Av ovan nämnda anledning lämpar sig SQLite ej heller för klient-server-baserade system, däremot är SQLite lämplig när ett program kräver en inbäddad databas. Det kan nämnas att SQLite används i Android-plattformen. SQLite är öppen källkod vilket medför att koden för programmet är tillgänglig för andra än tillverkarna. [1][14]

3.3.2 MySQL

MySQL är en relationsdatabas som använder SQL för att moderera data och MySQL är öppen källkod. Till skillnad från SQLite tillhandahåller MySQL fler funktioner vilket breddar användningsområdet, vilket avspeglar sig i att MySQL var den mest populära databasen med öppen källkod 2008. [11]

Följden av att tillhandahålla fler funktioner är att det blir mer komplext att konfigurera en MySQL databas. Till skillnad från SQLite utnyttjar MySQL auktorisering för att användare ska ha möjlighet att läsa och modifiera databasen. Denna egenskap tillsammans med databasens stöd för hanteringen av flera användare gör att MySQL lämpar sig för klient-server-modeller. De bättre möjligheterna till konfiguration gör också att databasen kan hantera en större mängd data där aktörer som Wikipedia och Google använder sig av MySQL databaser. [16][17]

3.4 Versionshantering

Versionshanteringssystem är ett sätt att hantera förändringar i dokument. Systemet bygger på att en användare förmedlar förändringar i en viss fil till ett centraliserat eller decentraliserat system. Förändringarna i filen medför att filen sparas som en ny version, medan tidigare versioner innehåller filen innan förändringarna.

Versionshantering används vanligen vid programutveckling för att synkronisera kod mellan flera olika utvecklare. Vanligtvis tillhandahåller dessa program även hantering av konflikter. Dessa uppkommer när två användare oberoende av varandra redigerar samma fil där den ene laddar upp ändringarna till versionshanteraren före den andre. Då kommer den senare användaren att få en konflikt vid uppladdning av sina ändringar, då den version som redigerades är föråldrad i relation till den senaste versionen i systemet.

Versionshantering möjliggör också att gå tillbaka i historiken för en fil. Vid utvecklingen av ett program upptäcks det att den senaste versionen av en fil orsakar problem vid körning, då är det möjligt att återgå till en tidigare version.

Versionshanteringssystem brukar delas upp i två kategorier nämligen distribuerade- och centraliserade system, skillnaden mellan dessa är kortfattat att distribuerade-system har en struktur liknande *peer-2-peer nätverk* medan centraliserade-system har *klient-server-modell*.

3.4.1 Subversion

Subversion, Apache Subversion eller SVN är ett centraliserat versionshanteringssystem med öppen källkod, det är en vidareutveckling av *Concurrent Versions System (CVS)*. Själva biblioteket för filer och versioner installeras på en server, varefter klienter ansluter till servern för att hämta och ladda upp uppdateringar. Klienten kan vara ett enskilt program men även en del i ett annat, t.ex. i utvecklingsmiljöerna Eclipse och Netbeans. [2]

3.5 Java

Java inom datavetenskap är både ett programmeringsspråk och en plattform. Denna plattform kan köras på flera olika system, exempelvis Windows, Unix (Linux), Mac, Android, etc. Detta möjliggör att program skrivna i programmeringsspråket Java kan köras på flera system, programspråket är alltså plattformsoberoende.

Javaplattformen kan delas upp i två delar, *Java Runtime Enviorment (JRE)* och *Java Development Kit (JDK),* det förstnämnda används för att köra kompilerade Java program och är troligen den delen som fått störst spridning. JDK är den delen som används för att utveckla program, JDK innehåller fler olika delar t.ex. kompilator och debugg-verktyg.

Den senaste versionen av Java vid skrivandet av rapporten är 1.7 vilken kom ut under sommaren 2011. Den stora skillnaden mellan 1.7 jämfört med föregående version för en programmerare är att den underlättar att deklarera listor, vilket medför att det nu är möjligt att deklarera innehållet vid instansiering. Det är också möjligt att använda strängar i *switch*-satser, samt ett flertal andra förbättringar. En annan stor förändring är att prestandan för denna nya version har förbättrats i jämförelse med tidigare versioner. [13]

3.6 *Utvecklingsmiljöer* (Java)

Utvecklingsmiljöer (*Integrated Development Environment)* bruka inkludera följande tjänster : redigering av källkod med syntaxfärgkodning, kompilering av programmet samt möjligheter att debugga respektive programvaran.

Utvecklingsmiljöer brukar även tillhandahålla stöd för att sätta ut *brytpunkter* för specifika rader. När programmet sedan körs kommer exekveringen av programmet att pausas vid brytpunkterna.

Två vanligsvis förekommande utvecklingsmiljöer för Java är Eclipse och Netbeans, vilka i stort erbjuder liknande tjänster men skiljer sig designmässigt åt. Val av IDE är ofta baserat på vilka funktioner och utseende som utvecklaren önskar. I många fall fungerar det utan problem att utveckla i samma projekt i flera olika IDE samtidigt. [8] [12]

4 Metod

Under projekts gång har ett antal nya metoder och lösningar på specifika problem utvecklats. I detta kapitel belyses ett antal av de nya metoder och lösningar som har använts.

4.1 Designmönster – MVC+Template

En del av projektet har varit att undersöka och utvärdera vilka designmönster som är lämpliga att använda i projektet med avseende på målet och valt programmeringspåk. De övergripande designmönster som har undersökts är *Model-View-Controller* och *Event-driven programming*. De båda programstrukturerna har sina för- och nackdelar avseende struktur, enkelhet, spårbarhet/felsökningsmöjligheter, samt i hur väl de möter kravet på delbarhet av kod, där det skall vara möjligt att dela upp kod i *Coreside* och *Frontend*.

Vid beslut om vilket av ovan nämnda designmönster som skall användas togs en lista fram med några för- och nackdelar med varje designmönster.

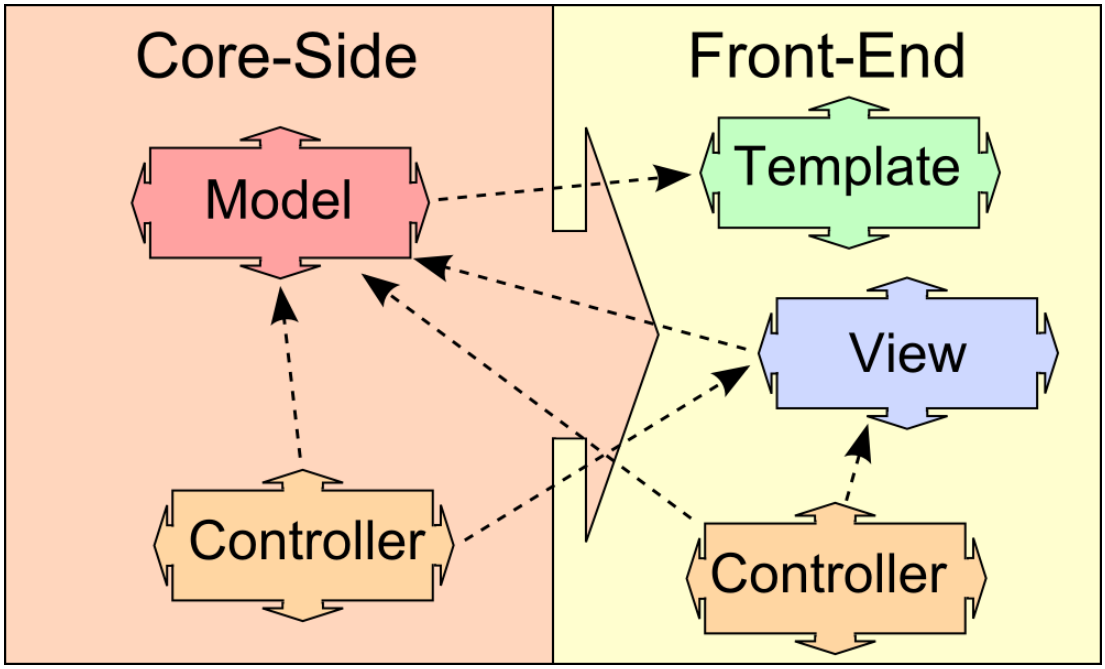
Model-View-Controller	
Fördelar	***Nackdelar***
Möjligt att testa och debugga olika komponenterna separat, i en isolerad miljö.	Ökad mängd kod och dokumentation behövs för samma uppgift.
Möjliggör sammarbete mellan flera olika deltagare på ett praktisk sätt.	Kräver mer förarbete innan faktisk kod kan skrivas.

Event-driven programming	
Fördelar	***Nackdelar***
Rakt programflöde, löser problem som uppstår.	Svårt och debugga mer komplexa fel.
Kräver mindre förarbete och går ofta fortare att implementera kod.	Riskerar blir svåröverskådligt när programmet växer

Med hänsyn till följande fakta så valdes MVC modellen som grund för programvaran.

Eftersom MVC modellen inte var helt passad till kravet på uppdelad *Coreside* och *Frontend* har *MVC+Template* skapats vilket är en vidareutveckling av MVC modellen för att anpassa denna till de designkrav som finns med avseende på skilda delar för *Coreside* och *Frontend,* vilket innebär att den del av programmet som är gemensam för

Business Invoice System skall vara strikt skilt från den del av programmet som är företagsspecifikt för att möjliggöra anpassning av Business Invoice System mot andra företag utan behov att skriva om stora delar av programmet.



Figur 3: MVC+Template Schema

MVC+Template består av fyra moduler (figur 3) i stället för endast de tre som i MVC, de olika modulerna är *Model*, *Controller*, *View* och *Template*. *Model* modulen tillhör *Coreside* medan *Template* och *View* tillhör *Frontend*, samt *Controller* som finns i båda delarna av programmet.

4.1.1 Model

I MVC+T (*MVC+Template*) så utgör *Model* kärnan för datalagring i systemet vilket i normalfallet är applikationsspecifikt. Beroende på tillämpning så önskar man kunna lagra olika typer av data. Ett företag kanske vill spara en kunddatabas medan ett annat vill lagra vad som är beställt. Men eftersom det finns krav på skild kod så fungerar ej MVC utan denna behöver utökas till MVC+T.

Förändringen i MVC+T till skillnad mot MVC är att modellen har fått ett väldefinierat ansvarsområde, nämligen datalagring. Med hjälp av *Template* delen som mall kan modellen skapas för att möjliggör lagring av data enligt specifikationerna i *Template*.

4.1.2 Controller

Controller kan ses som programmets styrenhet vars ansvar är att samordna och styra dataflödet mellan *Model* och *View* i programmet. Som figur 3 visar återfinns *Controller* både i C*oreside* och *Frontend*, eftersom vid noggrant övervägande slutsatsen nåddes att skapa *Controller* på båda sidorna, vilket möjliggör att både kravet på uppdelad kod men

även minimerad mängd kod på *Frontend* då vissa standardfunktioner ingår i *Coreside*. Det hade varit möjligt att enbart ha *Controller* på *Frontend* men detta hade medfört mycket extra kod i *Frontend*, då varje anrop mellan GUI:t och modellen skulle behövts deklareras på nytt för varje projekt .

4.1.3 View

View delen motsvarar det grafiska gränssnittet som möjliggör för en användare att interagera med programvaran. Genom att använda strikt MVC eller MVC+T möjliggörs det teoretiskt att skapa ett nytt GUI till en annan plattform eller efter nya behov utan att påverka andra delar av programvaran.

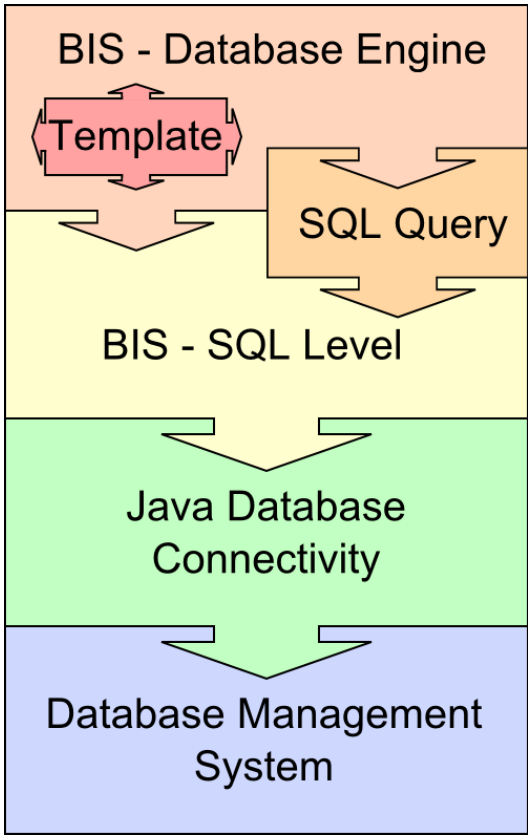
4.1.4 Template

Template är den nya delen som har tillkommit i MVC+T och som inte återfinns i MVC. Syftet med *Template* är att agera som mall när *Model* skapas dynamiskt för att kunna lagra önskad data. Detta sker genom att *Model* automatisk läser av *Template* för att sedan autokonfigurera den underliggande databasmodellen.

4.2 Databasmodellen

Ett mål med projektet har varit att göra en generell lösning, vilket även inkluderar stöd för flera olika databashanterare (*DBMS*) för att göra systemet plattformsoberoende.

För att nå målet med en generell databaslösning krävs en lösning som överbryggar de skillnader mellan de olika SQL-dialekter som kan finnas mellan olika *DBMS*. Dessa skillnader är bland annat hur tabeller skapas, ändras och raderas. Detta löses genom att tillämpa ett eller flera överliggande lager i databasmodellen som översätter en databasbegäran till en SQL Query (figur 4).



Figur 4: Databasstack för BIS.

4.2.1 BIS – Database Engine

BIS – Database Engine är det översta lagret i databas-modellen och motsvarar den virtuella databasmodellen som laddas in i systemet. I detta lager finns tabelldefinitioner och styrlogik för vad som kan göras med de skapade tabellerna, samt upprätthålla förändringar i tabelldefinitioner.

4.2.2 BIS – SQL Query

BIS – SQL Query är ett byggverktyg för att skapa SQL Query från logik och villkor som är skrivet i programkod. Detta möjliggör på ett enkelt sätt att skapa SQL Querys från programlogik.

4.2.3 BIS – SQL Level

BIS – SQL Level är den nivån i databasmodellen som ansvarar för att överbrygga de skillnader som finns mellan olika DBMS i SQL Querys när det gäller syntax och nyckelord som används när en query skapas. Det finns även rutiner för att skicka SQL Query till DBMS genom JDBC samt buffra data som fås tillbaka från databasen, vilket är en möjlighet som JDBC inte tillhandahåller.

4.2.4 DBMS & JDBC

DBMS och JDBC är de två lägsta lagerna i databasmodellen (figur 4). *Database Management System* motsvarar den verkliga databasen i systemet och endast denna del sparar data långvarigt. *Java Database Connectivity* är Javas API mot DBMS och har i uppgift att skicka SQL Query till databasen och ta emot svaren på sådan sätt att det är möjligt att läsa varje rad och kolumn på ett hanterbart sätt. JDBC ger även viss möjlighet att få ut metadata om vald databas så som vilka tabeller som finns, vilka egenskaper dessa har, etc.

5 Upplägg och Genomförande

I följande kapitel beskrivs hur projektet genomfördes, där bland annat vattenfallsmodellen har använts som projektmodell, MVC+T som programmeringsmodell, samt hur databasen har designats i enlighet med Lundh Sails krav och önskemål.

5.1 Projektmodell

För att genomföra projektet har vattenfallsmodellen använts som grund. Vattenfallsmodellen är en projektmodell som bygger på idén om att man gör ett steg i taget enligt en logisk ordning i ett projekt. Dessa steg skulle i ett programmeringsprojekt kunna vara: planering, kravspecifikation, designspecifikation, implementering, testning samt leverans och underhåll. Orsaken till varför vattenfallsmodellen har användas är att det är en av de få modeller som fungera praktiskt i en projektgrupp med endast två deltagare medan många andra projektmodeller kräver att en deltagare är projektledare vilket det inte finns möjlighet till med hänsyn till projektgruppens storlek samt den snäva tidsramen för projektet. [5]

5.2 Kravspecifikation

I samband med projektets uppstart skapades ett antal olika kravspecifikationer. Ett steg var att analysera vilka behov och krav som fanns för projektet Business Invoice System samt underprojektet Lundh Sails. Målet med kravspecifikationen var att tydliggöra vad den färdiga produkten skulle kunna göra och inte göra för att få tydliga avgränsningar i projektet.

För att kunna klargöra alla kraven skapades två olika kravspecifikationer för de olika delarna av projektet och en teknisk specifikation för vilka egenskaper och funktioner som de underliggande databashanterarna måste stödja.

- Kravspecifikation för Business Invoice System, som tar upp grundläggande krav och målsättning med den generella lösningen. Bland annat tydliggörs vad som skall finnas med i den generella lösningen. Det tas även upp systemkrav som är bestämt till att vara Java Runtime Environment för 1.6 (Java SE) men inga krav ställs gällande operativsystem.
  Se även Appendix A.

- Kravspecifikation för Lundh Sails AB, som tar upp de specifika kraven för den lösning som avser att hjälpa Lundh Sails med deras IT-lösning. Kraven tar bland annat upp grundläggande utseende av GUI, vilka funktioner och data som programvaran skall kunna spara.
  Se även Appendix B.

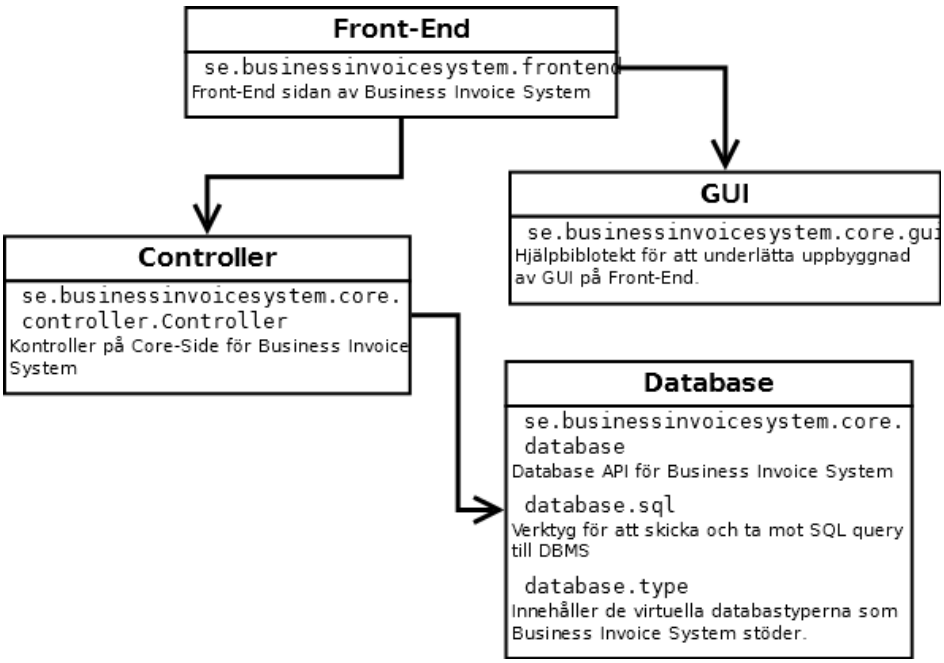
- Teknisk specifikation för databasen är vilka funktioner och egenskaper som måste stödjas av vald *DataBase Management System*. Initialt skall SQLite och MySQL stödjas. Specifikationen tar bland annat upp datatyper, transaktioner, etc.
  Se även Appendix C.

5.3 MVC+T modellen

För att realisera projektet i en verklig programvara utvecklas MVC+T modellen som är anpassad för krav på en både generell och företagsspecifik lösning. MVC+T modellen har tre huvudkomponenter samt två *Controllers* som hjälper till att samordna dataflödet i modellen.

5.3.1 Coreside

Coreside är själva kärnan i programmet och den del av programmet som är generellt byggd för att kunna fungera för ett annat företag eller efter nya behov i framtiden. *Coreside* består av tre olika delar som tillsammans utgör ramverket för Business Invoice System.



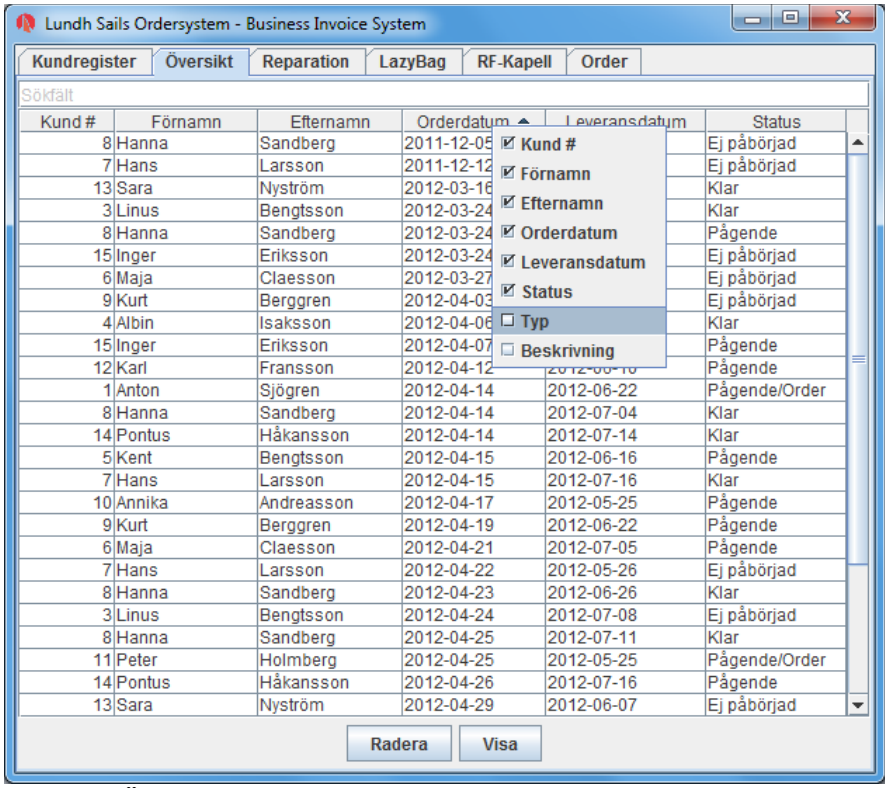
Figur 5: Översikt över Coreside, Frontend är med för tydlighetens skull.

5.3.2 Lundh Sails GUI

Eftersom tidsåtgången för att skapa ett fungerande GUI ofta överskrider givna ramar,var GUI:t bland de saker som påbörjades först och parallellt med att kravspecifikation togs fram för projektet. Detta var möjligt eftersom företaget Lundh Sails, sedan tidigare har haft en föråldrad programvara vars GUI de överlag är nöjda med. Vid utveckling av det nya GUI:t har bland annat den gamla programvarans GUI kunnat utgöra en grundmall, vilket underlättar i utvecklingsfasen, men framförallt underlättar för de anställda att använda den nya programvaran då de kommer att känna igen betydelsebärande element från den gamla programvaran.

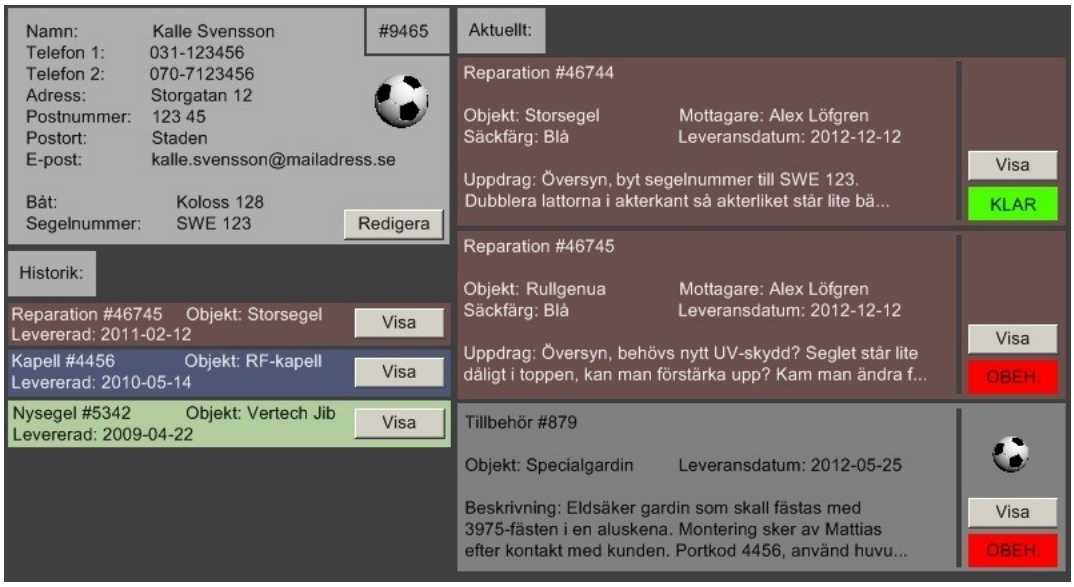
Utveckling av GUI har skett i tre faser. Den första fasen var att utveckla ett dummy GUI med de övergripande GUI komponenterna som skulle finnas med. Därefter påbörjades ihopkoppling mot den utvecklade databasen i systemet. Slutligen gjordes finputsningen, som innebar att åtgärda mindre detaljer och buggar i GUI:t.

GUI är uppdelat i två huvudkomponenter. Den första komponenten är en översiktsvy (figur 6) som bland annat visar kundregister, reparation samt en sammanställning. Det är möjligt att från översiktsvyn välja både vilka kolumner som skall visas samt söka efter specifika data som namn, orderdatum, etc.

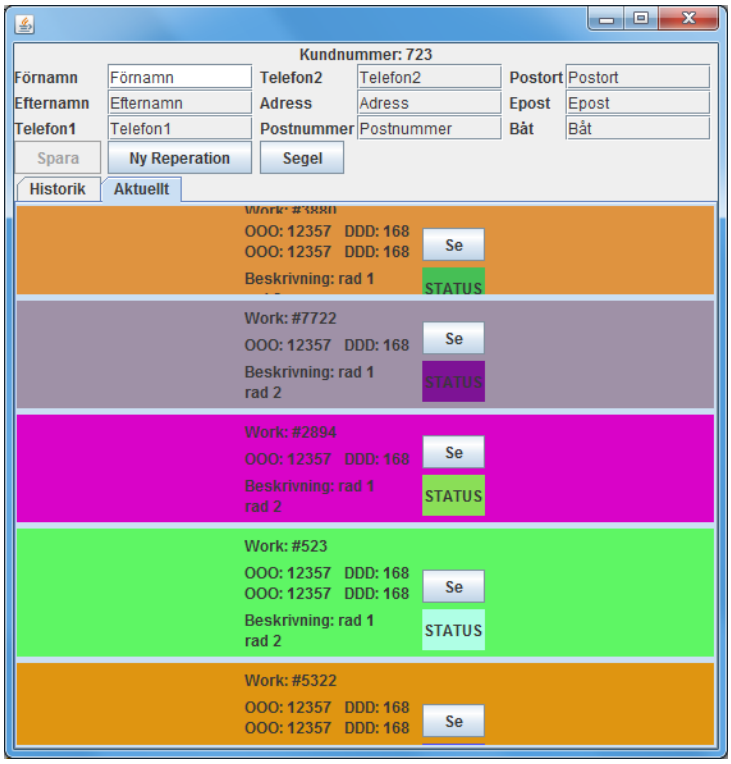


Figur 6: Översiktsvy av det färdiga programmet, Alla kunduppgifter är fingerade.

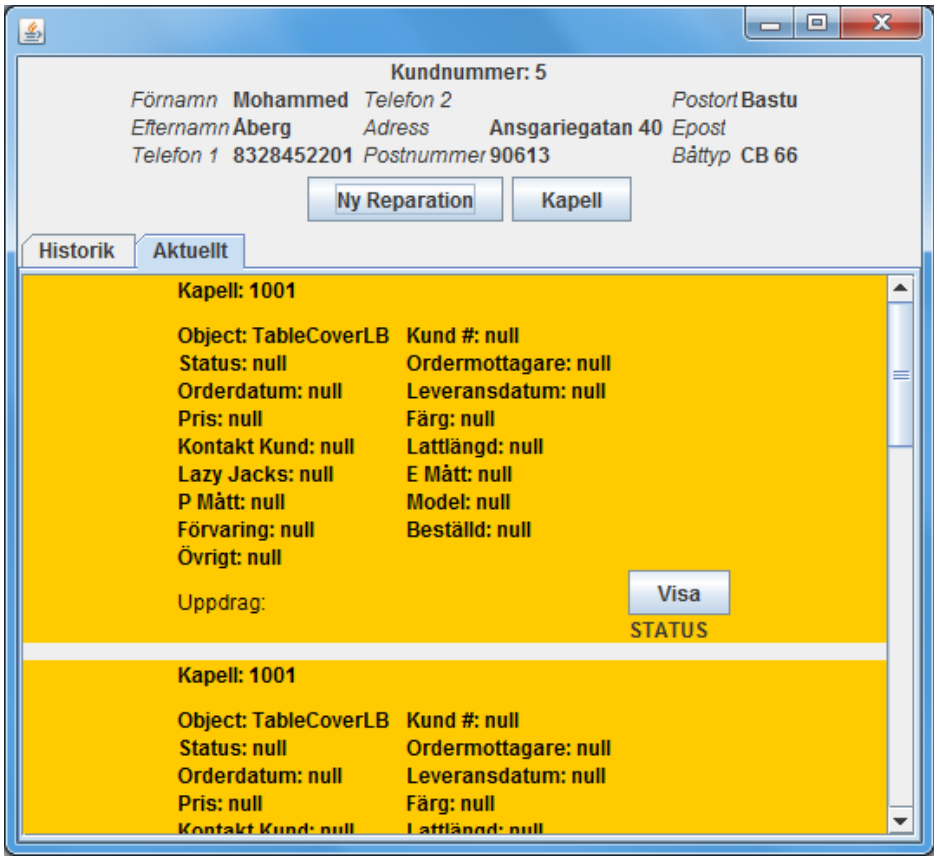
Den andra huvudkomponenten är kundkortet vilket ger detaljerad översikt över en specifik kund, pågående reparationer, order som finns, samt även historik. Om man t.ex. önskar visa en pågående reparation visas detta som en del av kundkortet för den berörda kunden för att underlätta översikt och hålla antal fönster i programmet litet.



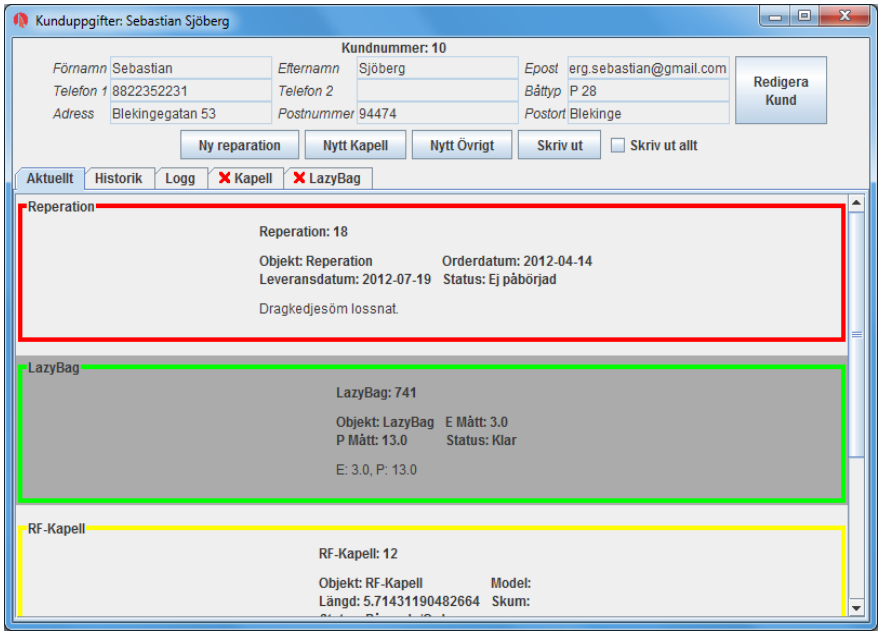
Figur 7: Föreslaget på Kundkort GUI från Lundh Sails.



Figur 8: Kundkort, dummy GUI som visar grundläggande funktioner och knappar.



Figur 9: Kundkort, ihopkoppling mot riktig databas har påbörjats.



Figur 10: Kundkort, det färdiga GUI:t för att visa en kund. Alla kunduppgifter är fingerade.

5.3.3 Template

I enlighet med MVC+T modellen skall det finnas en *Template* vars syfte är att förmedla databasschemat till databasdelen av MVC+T. För att spara databasschemat i programmet går det att göra på i princip två sätt. Det första är att hårdkoda SQL Query för att skapa valda tabeller, dock var detta ej något alternativ eftersom ett av målen med projektet var att gömma SQL-logik från slutanvändaren och programmeraren av *Frontend*. Det andra alternativet var att skapa ett objekt i Java som tillhandahåller databasschemat på lämpligt sätt.

Att skapa ett sådan objekt i Java visade sig inte vara helt lätt med hänsyn till existerande kodstruktur i Java, läsbarhet och kodmängd, då ett mål var att minimera mängden kod på *Frontend*. För att kunna finna en lämplig lösning testades flera olika sätt att ange databasschemat.

Det första förslaget på hur *Template* skulle se ut var enligt*:

```
public class TestTable extends DBTablePrototype {
  public TestTable() {
     super("testtable");
     this.addColumn(DBFactory.TEXT, "fname", DBTablePrototype.map("size", 32));
     this.addColumn(DBFactory.TEXT, "lname", DBTablePrototype.map("size", 32));
     this.addColumn(DBFactory.DATE, "date");
  }
}
```

Denna lösning skulle ha fungerat, men avböjdes eftersom detta gav alltför undermålig felhantering i IDE, på grund av att nycklarna definieras som strängar kan IDE ej kontrollera om syntaxen är korrekt. Nästa förslag på lösning se ut enligt:

```
public class TestTable extends DBTableResult<TestTable.Labels> {
  public static enum Labels {
     FIRST_NAME,LAST_NAME,DATE
  };
  public String getLabel() {
     return "testtable";
  }
  public void loadMetaTable(DBTableMeta<Labels> tableMeta) {
     tableMeta.addColumn(Labels.FIRST_NAME, DBFactory.TEXT, "fname",
     DBTableMeta.map(DBFactory.Arg.MAX_LENGTH, 32));
     tableMeta.addColumn(Labels.LAST_NAME, DBFactory.TEXT, "lname",
     DBTableMeta.map(DBFactory.Arg.MAX_LENGTH, 32));
     tableMeta.addColumn(Labels.DATE, DBFactory.DATE, "date");
  }
}
```

Denna lösningen blev betydligt bättre i avseende på felhantering i IDE, men behövde bearbetas något mer för att göra den lättläst för programmerare.

* Av praktiska skäll så kommer delar av koden att vara mycket förenklad eller utelämnad.

Så här ser den slutgiltiga versionen ut hur man definierar databasschemat:

```
public class TestTable extends DBTableResult<TestTable.Labels> {
   public static enum Labels implements DBLabelInterface {
      FIRST_NAME("fname"),
      LAST_NAME("lname"),
      DATE("date");
   };
   public String getLabel() {
      return "testtable";
   }
   public void loadMetaTable(DBTableMeta<Labels> tableMeta) {
         tableMeta.addColumn(Labels.FNAME, DBTText.class, DBArg.MAX_LENGTH, 32);
         tableMeta.addColumn(Labels.LNAME, DBTText.class, DBArg.MAX_LENGTH, 32);
         tableMeta.addColumn(Labels.DATE, DBTDate.class);
   }
}
```

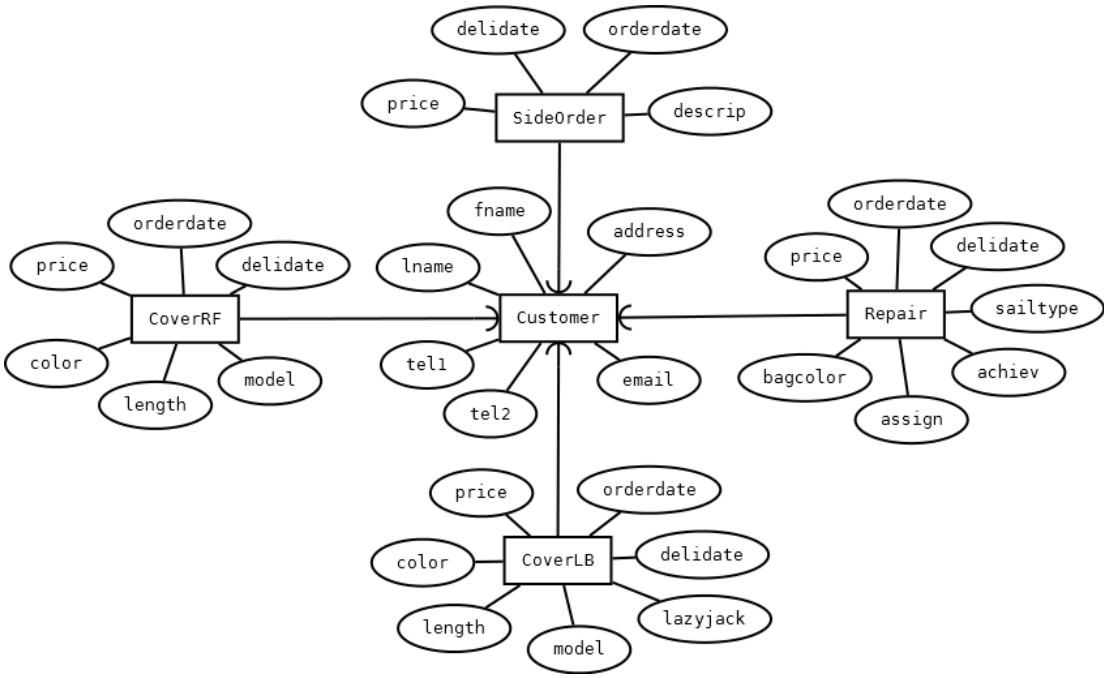
Alla tre kodavsnitten ovan avser att skapa en enkel tabell där man kan spara förnamn, efternamn samt datum.

5.4 Databasdesign

Till projektet skall även en databas designas i enlighet med kravspecifikation och önskemål från Lundh Sails. I korthet önskade de kunna lagra ett kundregister innehållande namn, adress, telefonnummer och e-post. Till kundregister skall något av följande ”order” objekt kunna kopplas: Reparation, order på nytt kapell och övriga order. Alla dessa objekt skall bland annat innehålla kunduppgifter, mottagningsdatum, leveransdatum och pris. ”Reparation” ger även möjlighet att spara uppgifter om vad som skall lagas, vad som är gjort, mm. Medan ”nya kapell” innehåller information om längd, storlek och modell. ”Övriga order” är det sista objektet de önskar kunna spara och innehåller de saker som ej faller under någon av de andra grupperna, t.ex. om en kund önskar få sytt ett solskydd till utedäcket.

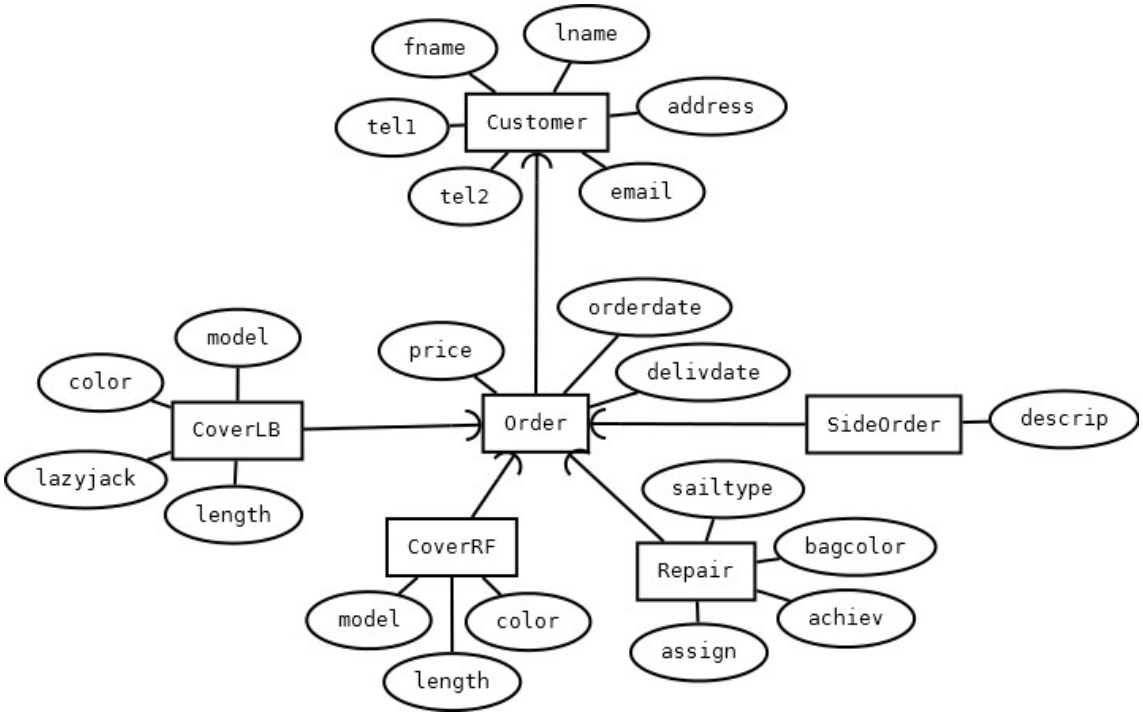
När man väljer databasdesign bör man både överväga design enligt t.ex. BCNF, men även den kunskap som finns om hur, var och när databasen skall användas. Detta kan i många fall också vara en avgörande faktor ur prestandasynpunkt.

När databasdesignen togs fram skapades först en enkel databas som strider mot principen om att minimera redundans i databasen enligt BCNF (figur 11).



Figur 11: Databas Design innan BCNF tillämpats, ger en enkel databas att överblicka då data som hör ihop alltid finns i samma tabell.

Efter att tagit fram den enkla databasdesignen optimerades denna enligt BCNF vilket i korthet innebar att samma data endast får existera en gång i databasen. När man optimerar en databas skall man aldrig helt lita på optimering då det finns fall där en optimering endast leder till sämre design.



Figur 12: Databasdesign efter BCNF som har tillämpats, ger en bättre optimerad databas men svårare att ändra och överblicka för en ovan person.

Från de två ovanstående (figur 11 och 12) designlösningarna valdes den första (figur 11) som ej tillämpar någon optimering enligt BCNF. Detta beslutades utifrån den kunskap som fanns om hur databasen skall användas i programmet. Att använda designlösningen enligt figur 12 skulle innebära att vid varje tillfälle som systemet önskar läsa något från databasen skulle åtminstone en *JOIN* behövas men sannolikt dubbla *JOIN*'s då resultatet från den första *JOIN* behöver slås ihop med rätt kund från kundregistret. En annan faktor som gjorde att den första lösningen (figur 11) används är att man från Lundh Sails sida önskade en databas som är lätt att vid behov administrera i annan programvara.

5.5 Portera mot gamla programvaran

Innan den nya programvaran kan tas i drift så krävs det att den gamla databasen överförs till det nya systemet. Detta för att minimera behovet av att köra både det nya och gamla programmet samtidigt under övergångsfasen. Eftersom portningen är något som endast behöver ske en gång, så har ett program i *PHP: Hypertext Preprocessor* utvecklas för detta syfte. Orsaken till varför programmet har utvecklats i PHP istället för i Java är att PHP har bättre och enklare stöd för att hantera MySQL databaser, vilken är

den underliggande databas som användes av Lundh Sails. En annan faktor för valet är att Lundh Sails har möjlighet att köra PHP skript på deras system (Det är även möjligt att kompilera PHP skript vid behov).

```
<?php
$sql_old_link = new SQLLib(...); // Open a MySQL link to the old database
$sql_new_link = new SQLLib(...); // Open a MySQL link to the new database
// List all customer in the old system.
$customers = $sql_old_link->query_array('SELECT * FROM `customers`;');
foreach ($customers as $customer) {
    $insert_customer = $sql_new_link->make_query(...);
    $sql_new_link->query_boolean($insert_customer,null, $autogen_id);
        // Copy data from the repair table for this customer.
        $repairs = $sql_old_link->query_array('SELECT * FROM `repair` WHERE id='.
        $customer['id']);
        foreach ($repairs as $repair) {
            $query = $sql_new_link->make_query('INSERT INTO ...', $autogen_id,
...);
            $sql_new_link->query_boolean($query);
        }
        // Do same for the other tables from the old database.
}
?>
```

I korthet fungerar programmet genom att först lista alla kunder i det gamla systemet, sedan kopierar programmet alla tillhörande objekt för aktuell kund till samma kund i det nya systemet.

5.6 Problem

Under projektets gång har ett antal intressanta problem dykt upp. Orsaken till problemen har varit val av programmeringspåk till programmeringsmodell och projektstruktur. Den gemensamma nämnaren för problemen är att det berör de mest fundamentala byggstenarna i programmering och datavetenskap. I följande avsnitt beskrivs problemen kortfattat och hur de har påverkat projektet, för vidare instudering se appendix.

5.6.1 Listor och Arrayer

Java stöder ett antal olika rutiner för att skicka in ett okänt antal objekt till en metod. För detta finns det i princip tre olika sätt:

```
public static void metod1(String ... inarg) {}        // varargs
public static void metod2(String[] inarg) {}          // fält
public static void metod3(List<String> inarg) {}      // lista
```

För att skicka argument mellan olika funktioner i databasen användes `metod1` vilken i många fall är den enklaste av ovan tre metoder för att skicka in ett okänt antal argument. Men eftersom `metod1` inte ger ett entydigt och framförallt intuitivt utseende vid mer komplexa metoder, där kanske inargumentet är av typen `Object[]` eller `Object[]`

`[]` valdes `metod3` som ofta innebär mer kod men tydligare utseende vid komplexa metoder. Denna lösning möjliggör också att det är enklare att slå ihop flera listor med varandra för att skicka vidare till nästa metod. Se vidare Appendix D.

5.6.2 Datum & Tid

Att hantera datum och tid är något som visar sig vara långt ifrån enkelt inom datorsystem. Detta visar sig bero på de många olika sätt att ange datum, samt hur dessa lagras digitalt. Det första problemet när det gäller datum och tidsangivelse är kulturellt då man över världen anger datum och tid på olika sätt. Ta följande exempel ”12/04/10” vilket kan tolkas som den 4 december 2010, den 10 april 2012 eller den 12 april 2010.

Ett annat problem ur samma aspekt är hur man digitalt lagrar en tidsangivelse, då ett datorsystem både har begränsningar i mellan vilka intervall och med vilken noggrannhet en tidsangivelse kan lagras. Valet av tidsangivelsesystem i ett datorsystem är beroende på flera faktorer, som tillgängligt minne och behov av noggrannhet.

I Java användas en variant av Unix Epoch vilket i korthet är antal millisekunder sedan ett fast datum. Medan SQL ofta använder ISO 8601 eller likande vilket t.ex. kan vara `2012-04-17T15:43:11+02:00`. SQL stöder även ofta något som kallas för zero-time (exempel `0000-00-00 00:00:00`) vilket inte i strikt mening är ett giltigt datum men ofta användbart som standardvärde då de uppfyller villkor för olika datumoperationer i en databas. Det visar sig att den automatiska omvandlingen mellan Javas och SQL tidssystem var mer problematisk än förväntat och det krävdes flertal olika specialfall för att hantera omvandlingen någorlunda korrekt. Se vidare Appendix E.

6 Resultat

I detta kapitel kommer det slutgiltiga resultatet att presenteras, som bland annat omfattar en kort översikt av programvaran och dess funktioner.

6.1 Programvaran

Programvaran består av de två delarna C*oreside* och *Frontend*. Båda dessa delar är sedan uppdelade i en databas och en GUI-del. Till dessa två delar tillkommer en kontrollenhet som möjliggör sammankoppling mellan databasen och GUI:t. Programvaran är utvecklad enligt *MVC+T* modellen.

6.1.1 Coreside

Coreside utgör den del av programmet som tillhandahåller den generella lösningen. Detta medför att denna del kan användas i olika projekt utan att koden behöver förändras. Genom att hela programvaran är uppdelad i två huvuddelar finns det också en specifik kontrollenhet för C*oreside.* Denna komponent möjliggör anrop från andra delar av programmet till databasmodulen utan att dessa delar har kännedom om hur databasen fungerar eller vilka komponenter som ingår.

6.1.1.1 Databas

Eftersom denna del av programmet implementerar den generella lösningen så innehåller inte databasmodulen några specifika definitioner om hur själva databasen ska vara organiserad utifrån kolumnnamn och liknande. Det som denna del gör är att tillhandahålla metoder för att skapa, modifiera och ändra databastabeller.

För att göra databasmodulen på C*oreside* generell och lösningen överskådlig valdes att begränsa antalet databaser som stöds till två, nämligen SQLite och MySQL. Anledningen till detta var att även om olika SQL-syntaxer är snarlika finns det bland annat skillnader mellan vilka databastyperna, indexering, auto-nycklar, etc som de olika databashanterarna stödjer.

Databasmodulen fungerar genom att användaren definierar ett antal klasser som används som mall för de tabeller som skapas. När sedan användaren önskar få åtkomst till den specifika tabellen och kolumnen används de specifika namnen i Java. Detta medför att användaren inte använder de specifika kolumn- och tabellnamnen som återfinns i underliggande databas.

```
public class TestTable extends DBTableResult<TestTable.Labels> {
  public static enum Labels implements DBLabelInterface {
     FIRST_NAME("fname"),
     LAST_NAME("lname"),
     DATE("date");
  };
  public String getLabel() {
     return "testtable";
  }
  public void loadMetaTable(DBTableMeta<Labels> tableMeta) {
        tableMeta.addColumn(Labels.FNAME, DBTText.class, DBArg.MAX_LENGTH, 32);
        tableMeta.addColumn(Labels.LNAME, DBTText.class, DBArg.MAX_LENGTH, 32);
        tableMeta.addColumn(Labels.DATE, DBTDate.class);
  }
}
```

Eftersom programvara även riktar sig till personer som inte nödvändigtvis har djupgående kunskaper i databaser och SQL, kommer programmet att förändra namnen på tabeller och kolumner genom att lägga till ett prefix. Anledningen till detta beslut, var att det finns en risk att användaren vid skapandet av tabeller och kolumner skulle använda nyckelord för SQL.

Att inte låta användaren få tillgång till de specifika kolumn- och tabellnamnen gör att programmet förhindrar användandet av nyckelord men även säkerhetsvaliderar anropen till databasen, vilket gör att databasen är injektionssäker.

Databasmodulen är designad på ett sådant sätt att användaren ej heller kommer att använda sig av den specifika SQL-syntaxen för sökning, insättning, ersättning och borttagning, utan dessa anrop sker genom specifika metoder som återfinns i databasmodulen. Om användaren utnyttjar den generella kontrollenheten kommer ej heller metoderna i databasen att vara synliga, utan endast de metoder som utför motsvarande funktioner i kontrollenheten genom anrop till databasen.

6.1.1.2 GUI

Även om GUI:t ofta måste anpassas för det specifika företaget, erbjuds ett antal standardkomponenter i den generella lösningen. Dessa standardkomponenter är specialanpassade för att fungera tillsammans med en skapad databasmodell.

Coreside för GUI:t kan delas upp i två typer av komponenter, den ena utgörs av underkomponenter som kan användas i komponenter för att underlätta kopplingen mot databasmodellen, medan den andra delen utgör en komplett komponent vilken producerar en grafisk representation av data i databasen.

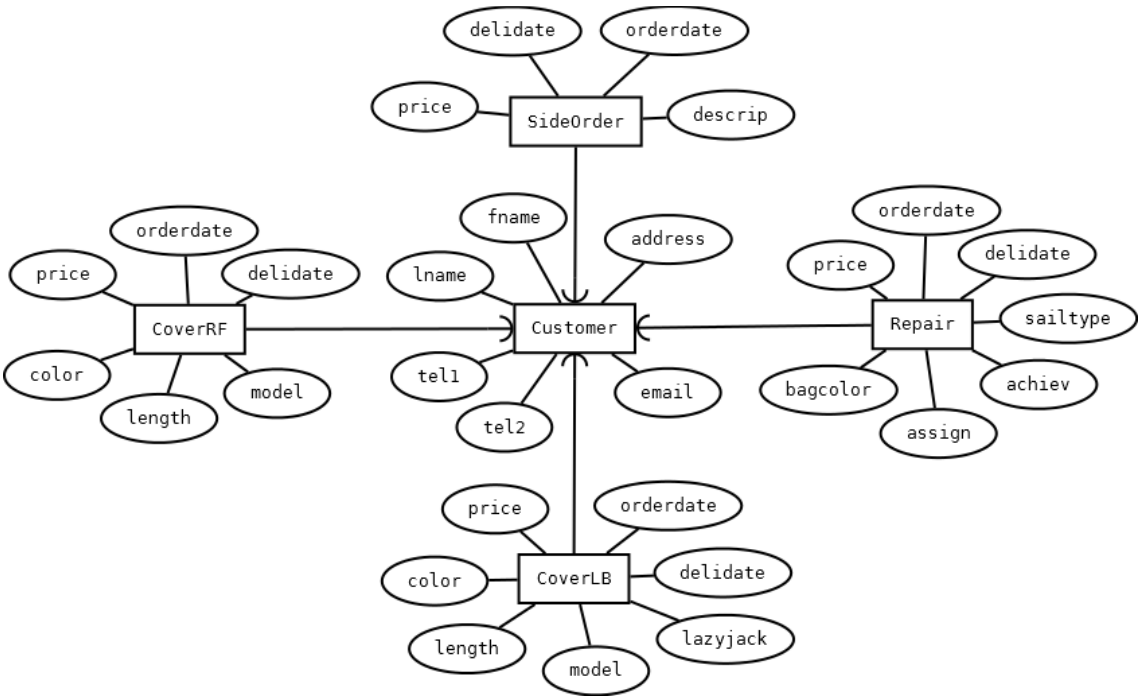
Det finns bland annat komponenter som möjliggör visuell representation av data som tabellvyn där kolumnerna kan visas och gömmas samt placeras om efter behov. De andra komponenterna är designade utifrån att programmeraren ska använda dessa när det krävs inmatning och utmatning av data till GUI:t.

6.1.2 Frontend

Frontend är den delen av programmet som är specifikt utformat för det aktuella företaget, i detta fall Lundh Sails AB. Även i *Frontend* tillkommer en kontrollenhet för att hantera anrop till databasen som är unika för det aktuella företaget. Det finns en komponent i programmet för att hantera texter som ska visas i det grafiska gränssnittet. Dessa texter hämtas från en språkfil. Detta möjliggör ändring av språkfilen i efterhand, utan kunskaper om hur programkoden är uppbyggd eller att behöva kompilera om programmet.

6.1.2.1 Databas

Databasen som används av Lundh Sails är en MySQL databas. Designen som användas visas i figur 13. Denna design för databasen är ej normaliserad enligt några av de metoder för databasoptimering som finns. Anledningen till detta är att under utvecklingens gång framkom att anropen till databasen skulle bli på ett sådant sätt att vyer skulle behöva skapas som liknar relationerna i figur 13. En ytterligare anledning till denna förenklade design, var att IT-ansvarig på Lundh Sails önskade ha möjlighet att manuellt redigera databasen, vilket skulle försvåras med en normaliserad databas.

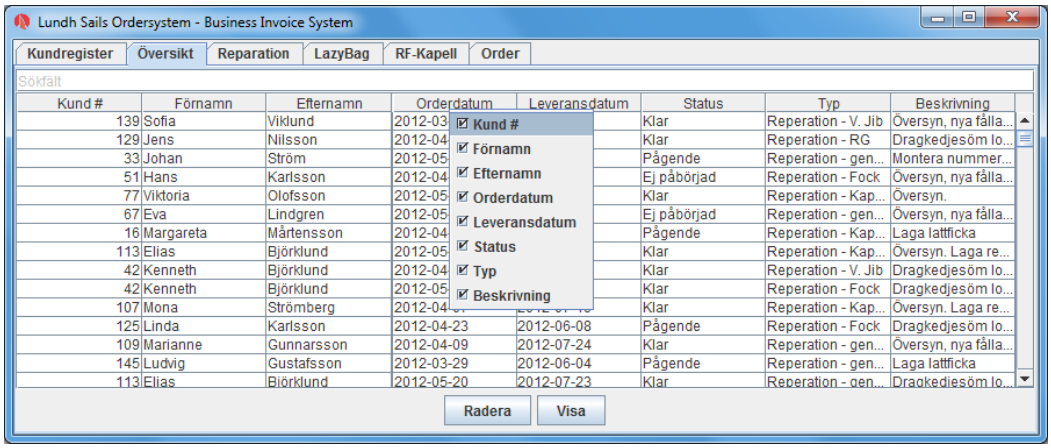


Figur 13: Databas Design innan BCNF tillämpats, ger en enkel överblickbar databas då data som hör ihop alltid finns i samma tabell.

Programkoden för databasen för *Frontend* är minimal och utgörs endast av de klasser som definierar tabellerna i databasen med tillhörande kolumner, samt en klass som definierar vyer för programmet.

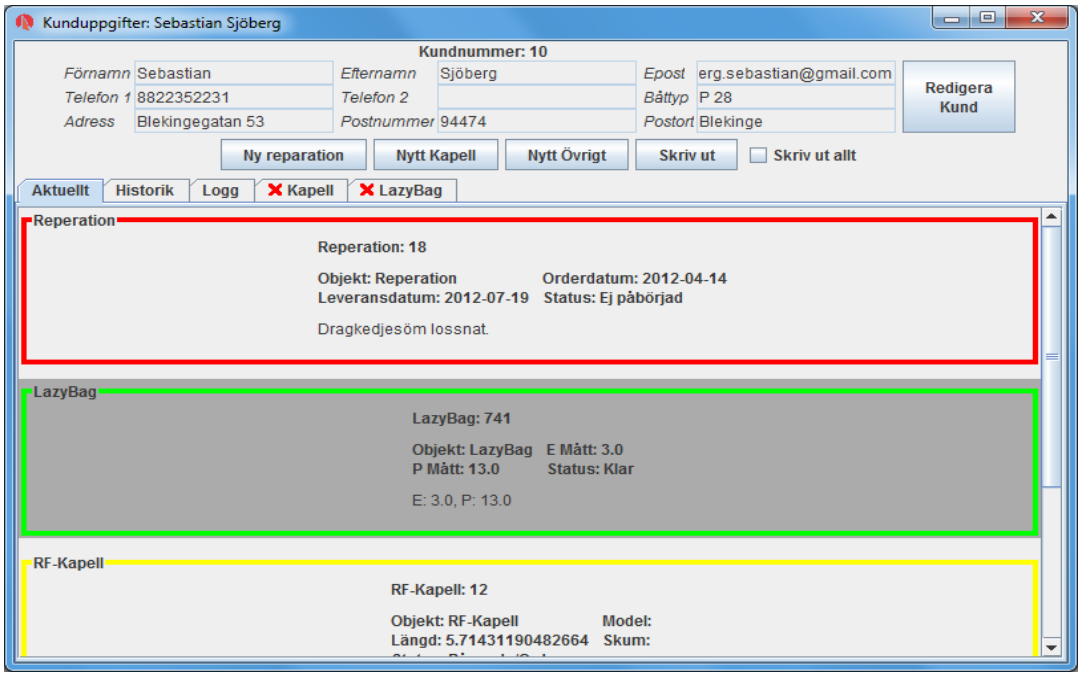
6.1.2.2 GUI

När användaren startar programmet (figur 14) visas en översikt över de data som återfinns i databasen. Detta inkluderar kund- och orderregister av varierande slag. Förutom detta är det möjligt att söka i databasen, visa, gömma samt byta ordning på kolumnerna i tabellvyn. Dessa ändringar bibehålls också efter omstart av programmet.



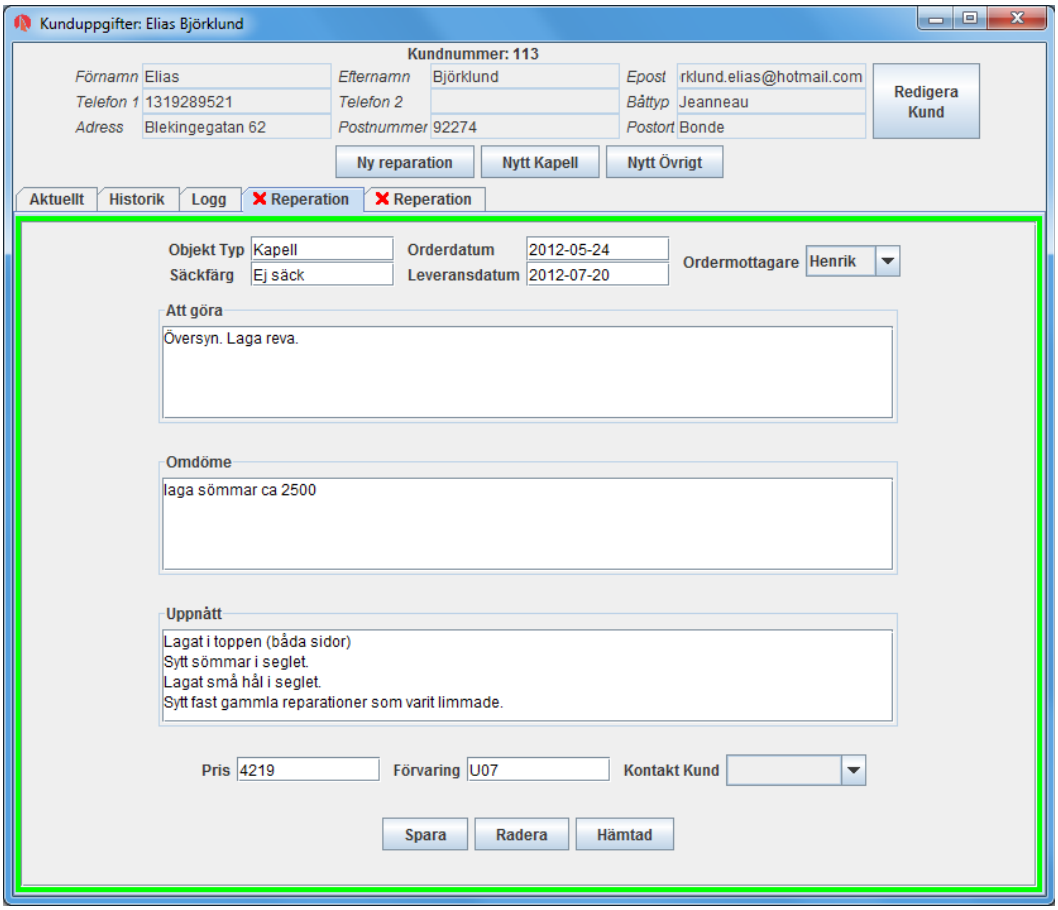
Figur 14: Översikt över pågående händelser och kunduppgifter. Alla kunduppgifter är fingerade.

Om användaren sedan väljer att visa en specifik kund eller post kommer denna data att visas i ett fönster (figur 15), om bara en kund önskas visas kommer ett fönster som endast innehåller flikarna *Aktuellt, Historik* och *Logg* att visas. Från detta kundfönster kan användaren överblicka nuvarande och tidigare order, men även lägga till nya samt överblicka tidigare händelser via loggen.



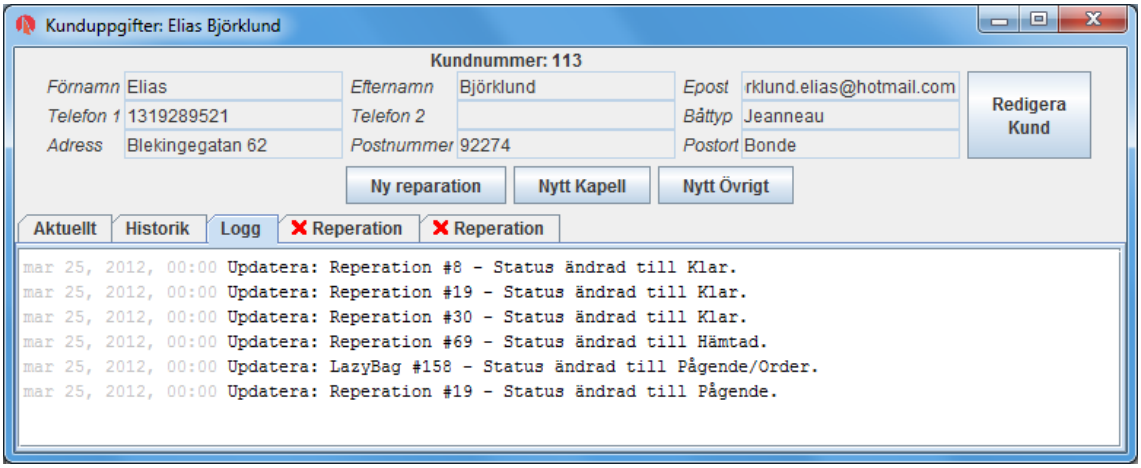
Figur 15: Kundkort. Alla kunduppgifter är fingerade.

Om man däremot väljer att visa en specifik order-post kommer ett liknande fönster att öppnas för aktuell kund. Därtill kommer en flik att visas med uppgifter om den aktuella ordern, från vilken användaren kan ändra uppgifterna (figur 16). Det går att stänga en flik via krysset vilket medför att om uppgifterna ej är sparad kommer användaren att tillfrågas om användaren önskar stänga aktuell flik innan den stängs. Om uppgifterna däremot är sparade kommer fliken att stängas direkt vid klick på krysset.



Figur 16: Pågående Reparation. Alla kunduppgifter är fingerade.

För varje ändring som användaren gör i programmet, vilka blir sparade, kommer information om den aktuella händelsen att sparas i loggen (figur 17). Detta sker för att användaren enkelt ska kunna överblicka aktiviteterna för en viss kund.



Figur 17: Logginformation om förändringar för aktuell kund. Alla kunduppgifter är fingerade.

7 Resultatanalys

Detta kapitel analyserar resultaten som uppnåddes med programvaran.

7.1 Coreside

Den generella databaslösningen uppfyller kraven i kravspecifikationen. Dessutom infördes validering av definitionerna i Template vilket medför att felaktiga definitioner leder till kompileringsfel. Dessa valideringar medför dock att det blir mer komplicerat att definiera tabellklasserna, men detta är en acceptabel kompromiss. Om inte valideringar fanns skulle felsökning försvåras.

Initialt i projektet var det ej säkert om det skulle finnas generella komponenter för GUI:t men efterhand som projektet fortgick kom ett fåtal sådana komponenter att utkristalliseras. Dessa kom dock endast att inkludera de mest grundläggande funktionerna då ett GUI ofta behöver anpassas för det specifika företaget.

Kombinationen av den generella lösningen med både databas- och GUI-komponenter, medför att det är relativt enkelt att skapa en applikation som kan hantera enklare databasstrukturer, utan behovet att skriva till stora mängder kod för det specifika fallet.

Även om nämnda komponenter är relativt enkla att implementera finns det andra delar som är mer komplicerade. Detta inkluderar vyer som är kombinationer av två eller fler tabeller i en databas. Sökmetoder finns i två upplagor från den enklare vilken endast söker efter matchande objekt till den mer komplicerade där sökning kan ske i enlighet med de logiska operatorer som återfinns i kravspecifikationen för databasen (se Appendix C).

Att lagra data till specifika typer ansågs ej heller vara helt enkelt, speciellt med grafiska komponenter som innehåller ett flertal inmatningsfält där vart och ett av dessa är knutna till en specifik kolumn i en databas. Av denna anledning utvecklades komponenter som kan utgöra substitut istället för Javas standardkomponenter, vilket underlättar arbetet att skapa ett GUI.

Även om vissa komponenter från den generella lösningen utifrån kan upplevas enkla, finns det bakomliggande logik av varierande komplexitet för att dessa ska kunna hantera data dynamiskt mot underliggande databas. Detta medför att det kan vara problematiskt för utomstående att förstå kodens funktion, trots kommentering. Följden blir att dessa delar kan bli problematiska att underhålla eller förändra.

7.2 Frontend

Även om *Coreside* till stora delar utvecklades i början av projektet, kom denna del ändå att utvecklas parallellt med *Frontend.* Efterhand som projektet utvecklades upptäcktes att vissa funktioner efterfrågades och att andra behövde förbättras. Det skedde alltså en kontinuerlig analys av utvecklad programvara i relation till nytillkommen kod. Detta är fördelen med att arbeta med en företagsanpassad version i kombination med den generella lösningen, jämfört med att endast utveckla en generell lösning som då hade innehållit brister som inte hade upptäcktes förrän vid praktisk tillämpning.

Att definiera tabellerna för Lundh Sails blev enkelt pga av hur *Model* i *Coreside* är implementerad, det som medförde viss tidsåtgång var att skapa vyer, då ett antal alternativ testades innan en lämplig lösning hittades. Det som förenklade definitionen av vyer var att de enskilda kolumnnycklarna i olika tabeller kan identifieras utan behov att på förhand veta vilken tabell en nyckel tillhör.

Valet av databasdesign berodde framförallt på vetskapen om hur anropen till databasen sker. Att genomföra en normaliserad av databas skulle innebära att det krävdes *JOIN* för nästan samtliga frågor. Detta skulle troligen få följden att den ev prestandavinsten för en normaliserad databas skulle gå förlorad. Därav valet av den icke-normaliserade databasdesignen.

Utvecklingen av GUI:t skedde stegvis där olika layouter provades och visades för företaget och IT-ansvarig. Den slutgiltiga layouten medförde att antalet fönster begränsades, vilket gör informationen mer överblickbar. Tillägget med sökfunktion underlättar också detta, vilket är en stor förbättring i jämförelse med tidigare program där detta saknades.

Svårigheterna att få överblick över kundregistret i tidigare programvara medförde att det fanns dubbletter i systemet. Genom introduktionen av en sökfunktion minimeras detta problem. Förutom sökfunktionen har det också tillkommit en funktion som kontrollerar om en ny kund redan existerar i databasen, vilket också minimera problemet med dubbletter.

En faktor som påverkat utvecklingen av GUI:t i hög grad är att programmet ska upplevas som intuitivt även för personer med begränsad erfarenhet av datorer. Eftersom utvecklare har god datorvana samt att de utformar designen och därför är medvetna om hur ett program ska agera, är det ej helt enkelt att avgöra vad som anses intuitivt för en ovan person. Detta har medfört att olika funktioner som agerar som tillägg till layouten, t.ex. knappar, klickytor, etc, har förändrats vartefter respons inkommit från företaget beträffande layouten.

7.3 I en verklig tillämpning

Som tidigare påpekats har även en programvara för kund- och orderregister utvecklas för Lundh Sails AB. Under rådande omständigheter har detta projekt fortlöpt snabbare än planerat. Detta medförde att den utvecklare programvaran har tagits i drift hos företaget Lundh Sails innan detta examensarbete slutfördes vilket inte ingick i den ursprungliga planeringen. Som förväntat upptäckters en del småfel och buggar i

programvaran som har förblivit oupptäckta under utveckling pga bristfällig manuell testning med avseende på tidsramen för projektet. Dessa kommer löpande att åtgärdas tillsammans med mindre justering av detaljer efter önskemål från Lundh Sails.

Lundh Sails anser att den nyutvecklade programvaran uppfyller de önskemål och krav som fanns vid projektets början. Sökfunktionen och dubblettkontrollen vid ny kund är de funktioner som tillför mest nytta för företaget i avseende på att snabba upp hanteringen av ”kunder” och ”order” i sitt system.

IT-ansvaring på Lundh Sails, ser fram emot vidare arbete för att göra programvaran fullständig med alla de önskemål som finns, detta för att skapa en heltäckande IT-lösning för Lundh Sails. Utökningen av programvaran kommer att ske stegvis så att alla har möjlighet att lära sig den nya programvaran innan fler funktioner tillförs.

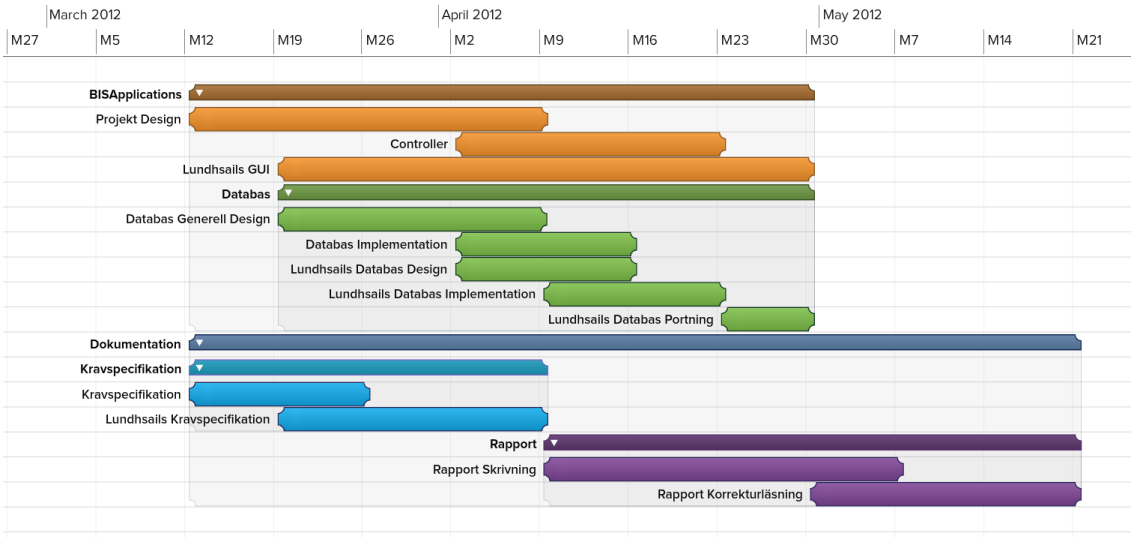
8 Slutsats

I detta kapitel diskuteras företeelser som utvecklarna har upplevt som både bra och mindre bra. dels i avseende på programmet, programmeringsspråk och metodik. Även med avseende på hur kontakten mellan utvecklarna och företaget har varit och vad som kan förbättras för framtida projekt. Det kommer även kortfattat att tas upp möjligheter till framtida projekt.

Resultatet av examensarbetet är att det har utvecklas en ny programvara för hantering av relationsdatabaserna MySQL och SQLite, genom den generella lösningen. Den utvecklade programvaran användas som ett kund- och ordersystem för Lundh Sails. Programvaran är uppdelad i två huvuddelar nämligen *Coreside* och *Frontend* där båda dessa är uppdelade i GUI, databas och *controller*. *Coreside* är den del av programmet som tillhör den generella lösningen och kan därmed också användas i andra projekt. Det som innefattas i denna del är framförallt databashanteringen men även visa färdiga GUI-komponenter samt några för att underlätta anslutningen till databasmodellen. *Frontend* utgör den anpassade delen av programvaran för det aktuella företaget, detta innefattar GUI:t samt mallen av databastabellerna.

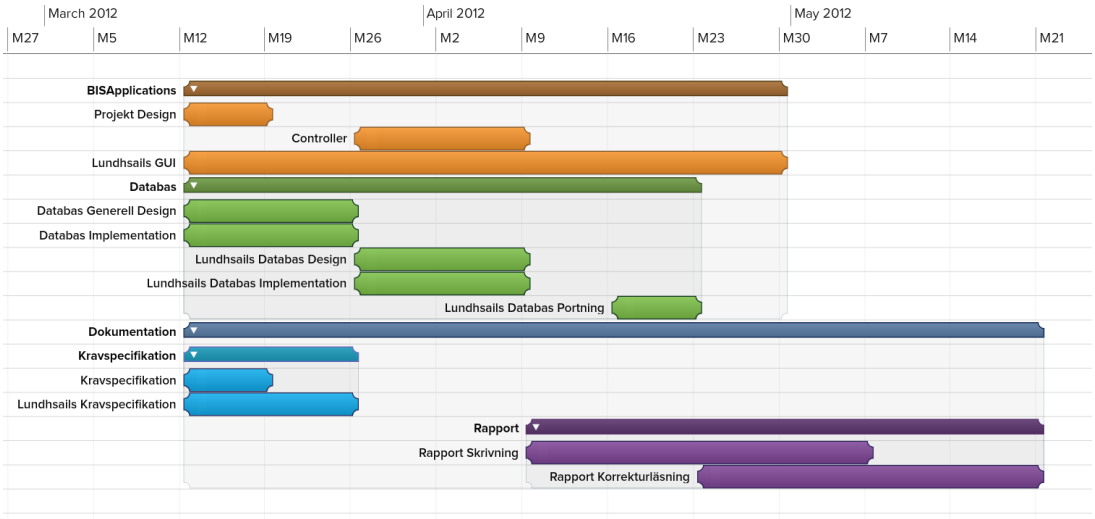
8.1 Tidsschema och planering

Som en del av förarbetet för projektet Business Invoice System togs ett tidsschema fram med syfte att säkerställa att det fanns tidsplan för alla momenten i projektet. För att grafisk visa tidschemat skapades följande Ganttschema.



Figur 18: Ganttschema, ursprunglig tidsschema för Business Invoice System.

Figur 18 visar det ursprungliga Ganttschemat som skapades vid projektets start, men som i många projekt blir den faktiska tidsåtgången en annan än den uppskattade.



Figur 19: Ganttschema, faktisk tidsschema för Business Invoice System.

Figur 19 visar den faktiska tidsåtgången för varje moment i projektet vilket i många fall krävde mindre tid än först planerat. En avgörande faktor till varför projektet fortskred fortare än planerat var att mycket av förarbetet och projektdesignen blev gjorda innan projektet officiellt påbörjades.

8.2 Kundkontakt i en aktiv process

Under projektets gång har en viktigt del varit att företaget som önskar programvaran aktivt är delaktigt med att utforma programmet. Det som upptäcktes var att kravspecifikationerna som skapades innan programmeringen hade påbörjats med tiden kom att förändras varför vissa begränsningar behövde göras, efter önskemål från företaget. Detta mönster följer vad som bör förväntas under mjukvaruutveckling, nämligen att kraven förändras efterhand som utvecklingen fortgår.

När programvara utvecklas med ej fastställda kravspecifikationer är det viktigt att kontaktpersonen hos företaget är insatt i hur programutveckling sker. Detta för att denna ska kunna förstå om något är möjligt eller inte att införa i programmet. Om inte så är fallet finns risken att personen sent lägger fram förslag och ändringar som är ogenomförbara.

8.3 Utveckling av GUI

Utvecklingen av det grafiska gränssnittet tog längre tid än vad som initialt förväntades, detta berodde på två faktorer. Företagets krav på GUI:t förändrades med tiden som programmet blev mer fullständigt. Detta var delvis förväntat då den initiala designen var något vag inom vissa områden, samt svårigheten att koppla ihop GUI:t med resten av programvaran, skriva styrlogik och kontrollogik för att få GUI:t att bete sig på önskvärt sätt.

Inför ett framtida projekt är det viktigt att planera in tillräckligt med tid för att skapa ett GUI och att felsöka detta i samband med att fel uppstår. Det är även viktigt att påbörja GUI:t tidigt i projektet, för att kunna leverera demo till företaget, så att detta lätt kan komma in med förändringar och förslag medan det finns tid och möjlighet att åtgärda dessa.

8.4 Java DataBase Connectivity

En stor del av projektet har varit att skriva ett gränssnitt mot en SQL databas genom att använda Java DataBase Connectivity. Detta har visat sig i många fall vara svårare än förväntat eftersom alla funktionerna som JDBC specificerar inte stöds av underliggande databas. Detta har lett till att man i många fall har fått göra specialfall av vissa lösningar för att kunna överbrygga problemen med JDBC och vald databas. Följande lista är några av de större problem som har upptäckts i JDBC under projektet:

- Felaktigt eller avsaknad av stöd för metadata om kolumner.
- Bristande mappning mellan SQL datatyper och Java datatyper.
- Ofullständig implementering av JDBC, inom vissa områden.

Under rådande omständigheter har ovanstående problem varit större än förväntat och något som behöver belysas tydligare i förarbete inför kommande projekt.

8.5 Framtida Projekt

Under projektets gång har ett antal olika möjligheter till framtida projekt som har sin grund i detta projektet uppkommit.

Ett huvudprojektet kan vara att göra denna programvara kommersiell. Då detta var en grundidé från början med separering mellan *Frontend* och *Coreside* i programmet. Genom att använda det utvecklat ramverk och *Coreside* skulle tidsåtgången för att utvecklar en ny programvara för ett annat företag avsevärt minska.

En annan möjlighet för framtida projekt är att utveckla en fullskalig relationsdatabas i Java som bygger på en valfri underliggande relationsdatabas (*DBMS*) genom *JDBC*. Detta för att överbrygga skillnaderna som existerar mellan olika *DBMS*. I så fall behöver nuvarande modell för databasen som används i detta projektet ses över samt att associationsnivån mellan olika databaslager behöver minskas för att förbättra prestandan.

9 Referenser – Litteraturförteckning

[1] Android, SQLiteDatabase, 2012-04-18, http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html

[2] Apache, Subversion, 2012-06-08, http://subversion.apache.org/

[3] Bravaco R., Simonson S., Java Programming, 2010, ISBN: 9780070181397

[4] Bravaco, R., Simonson S., Java Programming, 2010, ISBN: 9780070181397

[5] CMS, Vattenfallsmodellen, 2012-04-17, http://sms.unikom.ac.id/taryana/download/jad/SelectingDevelopmentApproach.pdf

[6] Codd, Edgar F, A Relational Model of Data for Large Shared Data Banks, 1970, doi: 10.1145/362384.362685

[7] David Geary, Take control with the Proxy design pattern, 2012-06-08, http://www.javaworld.com/javaworld/jw-02-2002/jw-0222-designpatterns.html

[8] Eclipse, , 2012-06-08, http://www.eclipse.org/

[9] Garcia-Molina, H. , Ullman, J.D., Widom, J., Database Systems, 2009, ISBN: 9780131354289

[10] JUnit, , 2012-06-08, http://junit.sourceforge.net/

[11] MySQL, Market Share, 2012-04-19, http://www.mysql.com/why-mysql/marketshare/

[12] Netbeans, , 2012-06-08, http://netbeans.org/

[13] Oracle, Java, 2012-06-08, http://www.oracle.com/us/technologies/java/overview/index.html

[14] SQLite, Appropriate Uses For SQLite, 2012-04-18, http://www.sqlite.org/whentouse.html

[15] Tony Sintes, Speaking on the Observer pattern, 2012-06-08, http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html

[16] Wikipedia, Wikimedia servers, 2012-06-08, http://meta.wikimedia.org/wiki/Wikimedia_servers#System_architecture

[17] Zack Urlocker, Google Runs MySQL, 2012-06-08, http://zurlocker.typepad.com/theopenforce/2005/12/googles_use_of_.html

10 Referenser – Illustrationsförteckning

Appendix

Appendix för examensarbetet Business Invoice System.

Appendix A

Kravspecifikation för projekt Business Invoice System.
Själva kravspecifikationen kan ha förändrats under projektets gång och motsvarar ej nödvändigtvis den faktiska slutprodukten.

Vad är Business Invoice System?

Business Invoice System är ett system som syftar till att tillhandahålla logistikhantering eller enklare databasfunktioner för mindre företag. Målsättningen är att systemet ska bestå av ett antal grundkomponenter från vilken personer med programmeringskunskap inom Java kan utveckla en applikation som tillgodoser företagets unika behov.

Vad skall Business Invoice System kunna göra?

Tillhandahålla möjlighet att skapa enklare databaser utan avancerade kunskaper i SQL eller databashanteringspolicy.

Ger möjlighet att avskilja modell och grafiskt gränssnitt mot underliggande styrlogik i programvarulösningen för att enkelt kunna ändra modell eller grafiskt gränssnitt utan ingående kunskap om underliggande logik.

Tillhandahålla färdiga grafiska delkomponenter som har stöd från underliggande logik för att underlätta byggandet av det grafiska gränssnittet

Möjlighet till att lokalt kunna ha eventhantering. Om en ändring i en specifik tabell sker så skall intresserad moduler kunna notifieras. Denna funktion skall endast hantera förändringar som sker via det lokala systemet, inga händelser kommer att genereras vid ändringar från annat system.

Vad skall Business Invoice System INTE kunna göra?

Tillhandahålla stöd i databasen för triggers samt views. (Undantag då triggers/views behövs för att uppfylla systemkrav för databasen)

Hantera användarbehörigheter för vem som har åtkomst till programmet eller får ändra uppgifter i databasen.

Systemkrav

Java Runtime Environment för 1.6 (Java SE).
Databas Management för valt databasmedium:

- SQLite: 3.6.14.2
- MySQL: 5.5.20

Se vidare Appendix C.
Stöd för grafisk miljö, minimal skärmupplösning 1024x768
Anpassat för liten skala, mellan 1 till 10 användare. Ger ej stöd för multiredigering av samma objekt i databasen. Liten databas, max 65000 rader i databasen.

Vad önskar vi att Business Invoice System skall kunna göra?

Anpassningsbart till andra Java Runtime Environment än 1.6 Java SE.

Ordlista

Business Invoice System, BIS
Avser programvarulösningen som skall utvecklas.
Databashanteraren, DB
Den mjuka databas som är skriven i Java och översättaren mellan SQL och *Frontend*.
Databasen, SQL
Den underliggande och faktiska databashanterare som ansvara för lagring på disk av data.
Coreside, Core
Programkärnan, som ansvarar för ramverket för *Frontend*.
Frontend
Klientspecifikt lager som utgör anpassningen för ett specifikt företag/part.

Appendix B

Kravspecifikation för underprojektet Lundh Sails (Business Invoice System).
Själva kravspecifikationen kan ha förändrats under projektets gång och motsvarar ej nödvändigtvis den faktiska slutprodukten.

Vad är Lundh Sails Ordersystem?

Programvaran syftar till att möjliggöra hanteringar av offerter och kunddatabasen samt data relaterad till dessa.

Vad skall Lundh Sails Ordersystem kunna göra?

Programvaran ska kunna hantera reparationer vilket inkluderar offerter, status, kundregister och anteckningar, det ska också vara möjligt att registrera order för nyproduktion av segel. Detta sker genom att nämnd data sparas i databasen, samt att ett GUI möjliggör visualisering och ändring av databasen.

Kundregister

Detta register ska vara sökbart, vara sammankopplat med de andra tabellerna som motsvarar olika orderobjekt.

- Kundnummer – Numeriskt (autogenerat)
- Förnamn – Text
- Efternamn – Text
- Telefon1 – Text-Numeriskt
- Telefon2 – Text-Numeriskt
- Adress – Text
- Postnummer – Numeriskt
- Postort – Text
- Epostadress – Text
- BåtTyp – Text
- Annat – Text

LazyBag

Order objekt för att hantera en order på en LazyBag.

- Kundnummer - Numeriskt
- Färg – Text
- Lagerplats – Text
- Längd – Numeriskt
- Modell – Text
- Lazy Jacks – Text
- Lattlängd – Numeriskt
- E-Mått – Numeriskt
- P-Mått – Numeriskt
- Ordermottagare – Enum

- Orderdatum – Datum
- Leveransdatum – Datum
- Beställd – Datum
- Kontaktakund – Enum
- Status – Text
- Pris – Text
- Annat – Text

RF-Kapell

Order Objekt för hantera en order på ett RF-Kapell.

- Kundnummer - Numeriskt
- Färg – Text
- Skum – Enum
- Lagerplats – Text
- Längd – Numeriskt
- Modell – Text
- Ordermottagare – Enum
- Orderdatum – Datum
- Leveransdatum – Datum
- Beställd – Datum
- Status – Text
- Pris – Text
- Annat – Text
- Kontaktakund – Enum

Reparationer

Order Objekt för hantera en reparation.

- Objekttyp – Text/Enum
- Kundnummer -- Numeriskt
- Säckfärg – Text
- Orderdatum – Datum
- Leveransdatum – Datum
- Ordermottagare – Text
- Pris – Numeriskt
- Lagerplats – Text
- Status – Enum
- Åtgärdats – Text
- Återstår – Text
- Omdöme – Text
- Kontaktakund – Enum

Övrigt

Övrigt innefattar order som ej omfattas av de överstående.

- Produktbeskrivning – Text
- Kundnummer – Numeriskt
- Ordermottagare – Text
- Orderdatum – Datum

- Leveransdatum – Datum
- Produkttyp – Text
- Kontaktakund – Enum
- Lagerplats – Text
- Status – Text
- Pris – Text

Logg

Loggen ska journalföra händelser och ändringar för den enskilda kunden.

- (Datum , Tid) – Datum, Tid
- Kundnummer – Numeriskt
- Objekt – Text
- Objekt-id – Numeriskt
- Loggtyp – Sträng
- Loggmeddelande – Text

(Ovan tabelldefinitioner tillhör ursprunglig kravspecifikation och kan ha ändrats under projektets gång.)

Vad skall Lundh Sails Ordersystem INTE kunna göra?

Programvaran ska ej hantera fakturering eller monetära transaktioner.

Vad önskar vi att Lundh Sails Ordersystem skall kunna göra?

Förmedla orderstatus via företagets hemsida till kunden, göra utskrift av uppgifter från databas, skriva inlämningskvitto (endast förmedlar orderstatus via hemsida infördes aldrig i programvaran).

Appendix C

Följande dokument är en teknisk specifikation för vilka krav som den underliggande databasen måste uppfylla för att vara möjlig att integrera tillsammans med *Business Invoice System*.
System krav
Databaser som i dag stöds av *Business Invoice System*.

- SQLite: 3.6.14.2
- MySQL: 5.5.20

Eventuellt i framtiden även:

- Oracle RDBMS

Stöd för Java DataBase Connectivity

Varje DBMS som önskas kunna använda i *Business Invoice System*. Måste har ett grundläggande stöd för Javas *”Java DataBase Connectivity”* samt stöd för följande typer och operationer som detta dokument anger.

Primär nyckel ”PRIMARY KEY”

Vald DBMS måste stödja att den primära nyckeln är av typen heltal och kunna hålla talen mellan 0 till 2 147 483 647, detta innebär att en tabell aldrig kan innehålla fler rader än 2 147 483 648.

Transaktioner

Vald DBMS måste stödja grundläggande transaktioner, ”Dirty reads”, vilket innebär att en pågående transaktion skall kunna ångras, men utan hänsyn till parallella händelser.

Auto-Increment

Vald DBMS måste stöda auto-increment för index av typ int. Detta innebär att om ett index för en INSERT utelämnas, kommer systemet fylla i den kolumnen med ett ledigt värde. Som en del av auto-incremenet så måste det vara möjligt att säkert läsa ut vilket värde som tilldelas som index för den nya posten i databasen.

Datatyper

Följande databastyper måste DBMS stödjas för att kunna vara en del av ”Business Invoice System”.

Text/Sträng

DBMS måste stödja text i variabel längd mellan 1 till 65535 tecken och om texten är kortare än 255 tecken så skall texten vara indexerbar.
DBMS måste stödja text i fast längd mellan 1 till 255 tecken, och skall vara indexerbar.

Alla textsträngar skall vara sökbara även om det kanske inte rekommenderas att i *Frontend* tillåta sökning i en text som man vet är 65535 tecken lång.
Alla textsträngar i databasen kodas enligt UTF-8 och skall ej kunna vara ”NULL”.

Blob/Binära filer

DBMS skall kunna lagra så kallade Blob (en fil utan hänsyn till teckenkodning), varje blob måste åtminstone kunna lagarna 16777216 byte (2^{24} eller 16MB).
En blob skall inte kunna vara sökbar eller vara en del av index, den skall ej kunna vara ”NULL”.

Tal och nummer

DBMS skall kunna lagra ett flertal olika typer av tal och nummer, varje tal skall vara indexerbart, sökbart och kunna anta värdet ”NULL”. (Undantag är om ett tal är en del av DBMS primära nyckel, som ej kan vara ”NULL”).
De tal och nummertyper som skall stödjas är:

- Heltal mellan -2^{63} till $2^{63}-1$ (-9223372036854775808, 9223372036854775807)
- Booleska värden mellan 0 till 1, eller mer vanligt TRUE/FALSE.
- 8-byte IEEE floating point number.

Datum tid

DBMS skall kunna hantera flera olika typer av datum och tid beroende på tillämpning.
Alla datum och skall vara sökbara och kunna vara en del av ett index.
Följande datumtyper skall det finnas stöd för, stödet skall även gälla skottår vid behov, men ej nödvändigt skottsekunder:

- Datum och Tid mellan 1000-01-01 00:00:00 till 9999-12-31 23:59:59.
- Datum mellan 1000-01-01 till 9999-12-31.
- Tid mellan 00:00:00 till 23:59:59.

Inga av ovan datum och tidformat behöver ta hänsyn till olika tidszoner eller eventuellt sommartid/vintertid. Alla datum och tidstyper måste kunna lagra värdet NULL vilket i så fall skall motsvara ett icke giltigt datum eller tidsangivelse.

Ref/Foreign key

DBMS skall kunna hantera referenser, så kallade foreign key till andra tabeller inom samma databas. Dock så skall DBMS enbart behöva ta hand om enkla referenser, det vill säga där enbart en databas-kolumn används som referens. Typer som används i referenser är heltal eller sträng under 255 tecken (av variabel/fast längd). DMBS skall beroende på om vald referens kan anta värde ”NULL” eller inte hantera brott mot en foreign key. Om DMBS hanterar ”NULL” så skall det sättas till ”NULL” när ett brott mot en foreign key uppstår. Om inte vald referens hanterar ”NULL” så skall ej borttagning av post få ske om det skapar ett brott mot en foreign key.

Villkor och operationer

Alla DMBS måste stöda villkorsoperationer (WHERE statement), de operationer som skall stödas är följande:
EQUAL, NOT_EQUAL, LESS_THEN, GREATER_THEN, LESS_THEN_OR_EQUAL, GREATER_THEN_OR_EQUAL, LIKE, REGEXP.
För att kunna skapa komplexa villkor så måste DMBS stöda följande villkor
AND, OR, NOT, samt inkapsling i parenteser för att styra ordning för validering.

Appendix D

Java stöder ett antal olika sätt att hantera funktions argument som är listor eller en uppräkning av objekt, för att skicka in ett okänt antal objekt till en funktion. Det finns i princip tre sätt:

```
public static void metod1(String ... inarg) {}   // varargs
public static void metod2(String[] inarg) {}     // fält
public static void metod3(List<String> inarg) {} // lista
```

För att anropa någon av metoderna gör man så här enligt java:

```
metod1(”Hello”, ”World”);
metod2(new String[]{”Hello”, ”World”});
metod3(Arrays.asList(”Hello”, ”World”))
```

Metod1 är det sätt som kanske ser ut att vara enklast att använda, vilket det även är i många fall. Men problemet med metod1 är att det inte alltid ger en intuitiv lösning för mer komplexa fall. Ett exempel är om man skickar in en array av strängar enligt följande sätt:

```
metod1(new String[]{”Hello”, ”World”});
```

I detta fall är Java så smart att den tolkar detta anrop som:

```
metod1(”Hello”, ”World”);
```

Eftersom metoden deklarerar att argumentet skall vara typen sträng, när Java kompilatorn upptäcker att inargumenet är en array av strängar omvandlar Java det med automatik till detta.

Vad händer då med följande metoden:

```
public static void metodObj(Object ... inarg) {};
```

Java definierar att varje objekt i programmet alltid kan omvandlas till typen Object då skulle följande fungera:

```
metodObj((Object[])new String[]{"Hello", "World"});

Object[] obj = (new String[]{"Hello", "World"});
metodObj(obj);
```

Då kan det antagas att även följande fungera:

```
Object obj = (new String[]{"Hello", "World"});
metodObj(obj);
```

Vilket inte är fallet eftersom Java inte längre vet att det är en array av typen Object som skickas in som argument.

Vad blir resultatet av följande metod?

```
public static void metodObj(Object[] ... inarg) {};
```

Slutsats; att använda variabla argument enligt första metoden är i många fall en enkel lösning men inte praktisk tillämpbart på mer avancerad strukturer såsom array av array. Särskilt inte då objektet utgörs av array av array med objekt där objektet också skulle kunna vara en array. I så fall är kanske den sistnämnda metoden som använder sig av Java List bättre eftersom den ger ett entydigt resultat.

Appendix E

Att hantera datum och tid på ett naturligt sätt har visa sig vara mycket svårare och mer komplext än vid första anblick:

- Vad är ett giltigt datum och tidsangivelse?
- Hur anger man ett giltigt datum och tidsangivelse?

Svaret är beroende på kultur, nationalitet, tillämpning därför är det väldigt olika vad som anses som ett giltigt datum. Ta bara följande exempel som alla är datum och tidsangivelse för exakt samma tidpunkt.

Tidssystem	**Tidsangivelse**
ISO 8601	2012-04-17T15:43:11+02:00
Unix Epoch/POSIX time	1334670191
International Atomic Time	2012-04-17 13:43:45
GPS-Time	2012-04-17 13:44:04
Julian day	2456035.5716551
Svensk tidsangivelse	Tisdagen den 17 april 2012, klockan 15:43:11

Ovanstående exempel är några sätt bland många för att ange datum och tid, vilket skapar problem när tid hanteras i ett datorsystem. För att kunna hantera tid och datum i ett system så måste det tas ställning till följande frågor :

- Första och sista giltiga datum, räcker från datum från 1970-01-01 till 2038-01-19 eller behövs större datumomfång, räcker då 0000-01-01 till 9999-12-31?
- Finns behov av att spara tidszoner i systemet?
- Behöver systemet kunna hantera skottsekunder?
- Vad är egentligen ett giltigt datum och tid, är 0000-00-00 00:00:00 ett giltigt datum och tid?

När ett system designas som skall hantera datum och tid krävs således noggrann eftertanke vid valet av vilket datum- och tidshanteringssystem som skall användas. Ett alltför omfattande datum- och tidshanteringssystem riskerar att göra datasystemet långsamt eller onödigt komplext medan ett allt för snävt system gör det oanvändbart för tilltänkt ändamål.