Datavetenskap

Folker Daniel, Jonsson Henrik

Utveckling av konfigurerbart användargränssnitt för databaser

---

Examensarbete, C-nivå

2005:01

Utveckling av konfigurerbart användargränssnitt för databaser

Folker Daniel, Jonsson Henrik

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Folker Daniel, Jonsson Henrik

Godkänd, Date, Arkiv | Egenskaper | Eget

Handledare: Zuccato Albin, Arkiv | Egenskaper | Eget

Examinator: Examiner, Arkiv | Egenskaper | Eget

Sammanfattning

Uppsatsen beskriver utvecklingen av ett grafiskt användargränssnitt för en databas med regler skrivna i XML-filer åt Industrisystem i Karlskoga AB. Utvecklingen utfördes med hjälp av ”Unified Process” och är influerat av tidigare program skrivna vid Industrisystem. Programmet är skrivet i c# med .NET och är uppdelat i två delar. Användargränssnitt och en logikdel som är skriven som en dll. Logikdelen hanterar reglerna från XML-filerna och kommunicerar med databasen. Användargränssnittet genereras från reglerna i logikdelen vilket gör det enkelt att ändra den information som visas samt de regler som ska användas. Fördelen med programmet vi utvecklade är alltså att det enkelt går att anpassa om databasen som används skulle förändras. Detta gör det också möjligt för flera kunder att använda programmet trots att de använder olika databaser.

Abstract

This essay describes the development of a graphical user interface (GUI) for a database with rules written in XML for Industrisystem in Karlskoga. The project has been developed using the Unified Process and is influenced by previous programs written at Industrisystem. The solution is written in C# with .NET and created as two parts. The GUI and a subsystem written as a dll that holds all the rules from the XML and interacts with the database. The GUI is generated from the rules read by the subsystem. This makes it easy to change witch tables and what rules affect the program and its tables as this is specified in the XML.The benefits of using this program is that if the database changes the XML can easily be rewritten for the new database without rewriting the program. This also makes it possible for several customers with different databases to use the program.

Innehållsförteckning

Figurförteckning

Tabellförteckning

1 Inledning

Vi har använt oss av romarnas gamla statsmannavisdom som säger söndra och härska (Divide et empera) för att dela upp uppsatsen. Detta gäller även för arbetet som utförts då vi valt att använda en metod som även den använder sig av detta. Denna metod har hjälp oss att lösa uppgiften i mindre delar och gjort utvecklingsarbetet mindre komplext.

1.1 Bakgrund

IndustriSystem i Karlskoga AB (ISAB) är ett företag med säte i Karlskoga. ISAB har regionkontor i Luleå, Stockholm, Göteborg, Malmö, Odense och Oslo samt en utvecklingsenhet i Lidköping.

ISAB levererar främst system till tillverkande eller distribuerande företag inom både mekanisk- och livsmedelssektorn. Systemen omfattar hårdvara, både egenutvecklad och inköpt, samt system bestående av egenutvecklad programvara som i regel säljs nyckelfärdigt inklusive utbildning med mera.

Ett typiskt projekt tillför användarvänlighet, maskinkopplingar ute i produktion samt märkning. Data till systemen kommer vanligtvis från affärssystem till vilka kopplingar görs för datautbyte. Detta innebär att systemen, i princip undantagslöst, används av datorovana användare i en tung eller stressig miljö. Systemen används även i produktion under alla dygnets 24 timmar i många fall även mobilt i truckar och lastfordon. Detta ställer stora krav på felhantering, stabilitet och att användargränssnitt är enkla.

1.2 Mål

Som ett led i att ta fram produkter så har ISAB gjort en idéskiss kring en produkt för att administrera grunddata i en datakälla.

De jobbar idag mot en egen framtagen standarddatabas som i dom flesta fall körs i Microsoft SQL-server. Det behövs i de flesta system ett verktyg för att administrera grunddata i

databasen. Målet är att ta fram ett program som kan konfigureras för att administrera en databas.

Programmet skall helt gå att konfigurera dynamiskt. Om det tillkommer en tabell så skall man enkelt kunna lägga till administration av den tabellen.

Krav från ISAB

- Programmet skall utvecklas i Microsoft Visual Studio .NET och i språket C#.
- Användargränssnittet skall vara enkelt och lättanvänt.

Konfigurering av systemet skall ske via XML-filer. Detta för att det är enkelt och beskrivande i sig samt väldigt dynamiskt. Man skall om det tillkommer en tabell kunna lägga till administration av denna genom att skapa en ny XML-fil som beskriver administrationen av den tabellen.

För fullständig information över programmets funktionalitet se bilaga A kravspecifikation

1.3 Tillvägagångssätt

Denna uppsats är uppdelad i fem kapitel. Det andra kapitlet, efter denna introduktion, förklarar vårat val av UP som utvecklingsprocess. För att motivera valet av UP har vi jämfört det med XP som är en annan utvecklingsprocess.

På grund av kravet att kunna kontrollera om de XML-filer som används är korrekt skrivna har vi jämfört två tekniker för detta. Det tredje kapitlet beskriver de två valmöjligheterna XML-schema och DTD tillsammans med en motivering av varför vi valt att använda XML-schema.

Efter dessa kapitel följer kapitel fyra, genomförande, där arbetet med att utveckla programmet tas upp. Detta kapitel är influerat av den utvecklingsprocess som valts för arbetet. Eftersom vi valde att använda Unified Process som utvecklingsprocess är detta kapitel uppdelat efter utvecklingsprocessens olika faser.

Den kod som skrivits under arbetets gång finns inte med i arbetet. Detta beror på att programmet skrivits åt ett företag och att det alltså inte är vi som äger koden. Det beror också på att koden inte är väsentlig för att arbetet ska gå att beskriva då det finns diagram som är tillräckliga för att beskriva detta. Detta trots att vi är medvetna om att Beck [1] hävdar att koden är den bästa dokumentationen.

Vi har valt att översätta de termer till svenska. Då det gäller termer angående UP så har vi använt oss av översättningar från *”Översättning engelska - svenska för uttryck inom RUP”* [13].

2 Process

Då detta examensarbete till stor del innefattar utveckling av mjukvara behövs en metod för utvecklingsarbetet. Det finns flera olika metoder för att utveckla mjukvara, men vi har valt att undersöka UP och XP för att se om någon av dessa metoder skulle vara användbar för vår uppgift.

Många utvecklingsprocesser har olika metoder för att angripa de problem som ofta uppstår under utvecklingsarbetet och vi har därför valt att beskriva dessa två närmare för att sedan kunna motivera varför vi valt UP istället för XP.

2.1 Unified Process

Unified Process är en beskrivning för hur man skall gå tillväga för att utveckla program. Processen följer romarnas gamla statsmannavisdom som säger söndra och härska (Divide et empera). Processen delar upp arbetet i mindre delar som är lättare att angripa än om allt skulle göras på en gång. Arbetet delas även upp mellan flera personer som får olika roller i utvecklingen. UP beskriver alltså ett tillvägagångssätt för utveckling av programvara som ett arbete av flera personer som jobbar med olika delar, men mot ett gemensamt mål.

Utvecklingsarbetet är i UP mera inriktat på arkitekturen av programmet än på implementationen. Till hjälp för arbetet använder UP flera olika diagram och figurer vars uppgift är att beskriva programmet ur en arkitekts synvinkel. Genom att noggrant beskriva programmets arkitektur är förhoppningen att många problem skall upptäckas och åtgärdas innan någon kod är skriven. Detta medför också att det färdiga programmet har en genomtänkt struktur som gör det enkelt att lägga till ytterligare funktionalitet om så skulle behövas i ett senare skede.

2.1.1 Hörnstenar

Det finns tre hörnstenar i UP: användningsfall, arkitektur (architechture-centric) och iterativ utveckling (incremental/iterative). UP använder sig av dessa hörnstenar för att driva arbetet mot en färdig produkt. Vi beskriver dessa mer detaljerat i 2.1.2, 2.1.3 och 2.1.5

2.1.2 Användningsfall

UP använder sig av så kallade användningsfall för att definiera vad slutanvändaren skall kunna göra i programmet. Dessa användningsfall representerar funktionella och ickefunktionella krav, ett bra exempel på en sådan är ”avsluta” i ”arkiv-menyn” som många program har. Användningsfallet för denna knapp har till uppgift att tydligt visa att användaren kan avsluta programmet. För att förtydliga detta används ett så kallat användningsfallsdiagram som visar vad en aktör kan göra i programmet. Aktören behöver dock inte vara personen som använder programmet utan kan till exempel vara ett annat program som utför en operation.

Det finns även ickefunktionella krav i form av prestanda och liknande. Kanske måste programmet gå att köra på en dator av en viss typ eller i en viss miljö. Det kan finnas krav så som att en uppdatering av en databas inte får ta orimligt lång tid utan måste vara klar inom en viss tid så att det går att fortsätta att jobba med programmet. Programmet kan även ha krav ställda på sig angående utseende och enkelhet vilka också räknas som ickefunktionella krav. Detta kan vara speciellt viktigt att tänka på om programmet skall användas i en stressig miljö eller av datorovana personer.

Användningsfall är inte bara ett sätt för kunden att se vad dess användare kan göra i programmet utan är även en stor hjälp när programmet skall utvecklas. Varje användningsfall kan då utvecklas som en egen del där målet är att programmet skall klara att hantera alla användningsfall. Enligt UP är användningsfall det som driver arbetet framåt eftersom de visar vad som måste ske i programmet för att det skall göra någon nytta. Allt eftersom dessa användningsfall blir planerade, implementerade och testade så går arbetet mot en färdig produkt. När alla användningsfall är implementerade och testade så är programmet färdigt för installation hos slutanvändarna.

2.1.3 Arkitekturen i centrum

Att UP har placerat arkitekturen i centrum betyder att allt arbete sker med arkitekturen som modell. Det är inte bara för programmerarna att sätta igång och programmera efter att kunden ställt sina krav utan dessa krav måste först förstås innan något annat arbete kan påbörjas. När detta arbete är färdigt kan programmerarna fortfarande inte sätta igång och skriva kod utan flera modeller måste skapas och utvärderas innan detta arbete kan sätta igång.

Detta leder till att mera arbete läggs på att planera hur ett program skall fungera och se ut än på att implementera programmet. Det kan vara ovant för de involverade till en början, men har väldigt många fördelar då den färdiga produkten kommer att vara strukturerad och genomtänkt.

Att arkitekturen får så stor del i arbetet leder också till att det kommer att produceras en stor mängd dokumentation och modeller. Detta gör att det enkelt går att sätta sig in i hur programmet fungerar utan att läsa hundra- kanske tusentals rader kod. Det blir med andra ord enkelt för en nyanställd person att sätta sig in i arbetet i ett senare skede om detta skulle behövas.

2.1.4 Faser

UP definierar arbetet i en process i fyra olika faser: förberedelse-, etablerings-, konstruktions- och överlämningsfasen. Dessa fyra faser definierar vilket sorts arbete som skall utföras och hur mycket.

- **Förberedelsefasen**

  I förberedelsefasen specificeras de krav som ställs på programmet. Hur arbetet skall genomföras planeras till viss del i denna fas. En viss prioritering av de användningsfall som tagits fram sker i denna fas. Viktiga användningsfall för programmet kan tack vare detta utvecklas i ett tidigare skede då dessa är nödvändiga för att programmet skall bli användbart.

- **Etableringsfasen**

  I etableringsfasen ska programmets kravspecifikation bli så gott som färdig och de värsta riskerna reduceras. När denna fas är färdig skall utvecklarna kunna uppskatta

kostnaden för att utveckla programmet samt vara tillräckligt införstådda i hur programmet skall fungera för att kunna planera konstruktionsfasen.

- **Konstruktionsfasen**
  Konstruktionsfasen är den fas där det mesta av implementationen görs och när denna är genomförd skall programmet vara i sådant skick att det skall kunna börja installeras hos kunden.

- **Överlämningsfasen**
  Den sista fasen under projektets gång är överlämningsfasen i vilken produkten levereras till användarna. De sista fixarna i programmet görs i denna fas då problem kan komma att uppstå när programmet körs i kundens system som inte kunde upptäckas tidigare. Innan denna fas är slutförd tränas användarna för att använda programmet.
  [5]

2.1.5 Iterationer

För att dela upp arbetet i mindre och mera hanterbara delar använder sig UP av iterationer. En iteration är en tidsrymd då ett visst specificerat arbete skall slutföras. Genom att dela upp hela projektet i flera iterationer skapas programmet i steg och dessa kan sedan utvärderas och kontrolleras. Hur många iterationer som ingår i ett projekt är inte bestämt från början utan beror på hur stort projektet är och hur lång tid det uppskattningsvis kommer att ta. Det kommer dock alltid att finnas minst fyra iterationer, en för varje fas. De olika faserna beskrivs i 2.1.4

Iterationerna är dock inte helt avgränsade från varandra eftersom nästa iteration i ordningen planeras i slutet av den pågående. Detta är viktigt eftersom saknade krav som upptäckts under utvecklingen av iterationen kan läggas in i nästa iteration.

Iterationerna är i sig uppdelade i flera arbetsflöden. Dessa behandlar krav, analys, design, implementation och test. En iteration är alltså även den uppdelad i flera delar där olika arbetssätt används för att iterationen skall gå mot ett slutförande.

- **Krav**

  Huvuduppgiften i detta moment är att utveckla en modell av systemet som skall utvecklas med dess användningsfall enligt Jacobson et al [5]. Denna modell tas fram tillsammans med kunden och kommer därför att ge en bra bild av vad kunden begär av produkten efter att iterationen avslutats.

- **Analys**

  Här skrivs kraven om så att de speglar de krav som ställs på programmet från programmerarnas synvinkel, den så kallade ”inre vyn” av systemet.[5] Den arkitektur som tidigare specificerats kan också komma att ändras till viss grad då fler av de användningsfall som tagits fram planeras.

- **Design**

  I designmomentet skall en design av programmet som följer de krav som ställts i kravmomentet samt under analysen skapas. Denna design kommer alltså att stämma överens med kundens önskningar eftersom den tar hänsyn till det som gjorts i kravmomentet, men den måste också ta hänsyn till resultatet av analysmomentet och kommer därför att ge en bild av programmet som fungerar efter kundens önskemål och går att implementera.

- **Implementation**

  Under implementationsmomentet skrivs den kod som bygger upp det slutliga programmet. Eftersom designen av iterationen är färdig är detta en av de enklare momenten då allt som har planerats bara ska skrivas ned så att det sedan går att kompilera.

- **Test**

  När implementationsmomentet är avslutat tar testmomentet vid och i detta moment är målet att testa det som utvecklats i iterationen så att det fungerar som specificerat. Detta kan göras genom att skriva automatiserade test som kontrollerar den färdiga

delen automatiskt. Det kan dock vara så att alla delar av ett program inte kan testas på detta sätt och programmet måste testas manuellt. Åtminstone 75% av testen bör dock vara automatiserade enligt Jacobsen et al [5]. När alla test är gjorda och programmet fungerar korrekt avslutas iterationen.

2.1.6 Roller

UP delar upp de anställda i ett projekt i många roller och definierar vad varje roll har för åtaganden under utvecklingsarbetet. Det finns dock inget som säger att en person bara kan ha en roll utan i små projekt som använder UP så kan det finnas personer som har flera roller. Den i särklass viktigaste rollen i UP är arkitekten eftersom denne ansvarar för arkitekturen av programmet. Utan denne skulle programmerarna vara tvungna att prata direkt med kunden och detta kan ställa till problem om det är flera programmerare och de får olika bilder av hur slutprodukten är tänkt att fungera.

Genom att ha en tydlig roll som arkitekt kan denne ta kundens krav och skapa en bild av hur programmet skall se ut som alla andra involverade i projektet sedan kan ta del av för att förstå vad de behöver göra. Detta medför att programmerarna får en mera tillbakaskjuten roll i UP än i många andra utvecklingsprocesser.

2.2 Extremprogrammering

Extremprogrammering (XP) är en lättviktsprocess som fokuserar på enkelhet, kommunikation, återkoppling och mod enligt Beck [1]. Extremprogrammering är baserad på tolv vanor som presenteras i 2.2.1 XP skiljer sig mycket mot UP då den är anpassad för små grupper med snabbt förändrade krav. Det är en ganska ny och kontroversiell teknik. Arbetssättet framtvingar kundtillfredsställelse då processen är designad för att leverera vad kunden behöver, när kunden behöver det. [3]

XP har skapats av Kent Beck och baseras på hans erfarenheter med objektorienterad programmering. Det är designat för mindre projekt med grupper bestående av mellan två och tolv programmerare. [3] Det finns ett antal aktiviteter som man måste genomföra för att säga att man utövar XP. Viktiga inslag i XP är incrementell planering, utveckling i små cykler med

klar och kontinuerlig återkoppling. Kommunikation är viktigt, XP föredrar muntlig kommunikation före skriftlig.

XP är ett antal idéer för att producera bra program snabbt. Reglerna måste användas i sammanhang och användas klokt tillsammans med XP:s fyra värden. XP lovar att minska projektets risker, klara av konstant ändrade affärskrav/behov och ökar produktiviteten genom systemets livslängd. Allt detta på en gång [7].

2.2.1 Extremprogrammeringens 12 vanor

XP har tolv vanor. Det är en anpassningsbar process och gruppen kan anpassa vanorna så att de passar deras förutsättningar. På grund av att XP är resultatet av flera olika personer och att vanorna förfinas så är det omöjligt att ge en definitiv förklaring av vanorna. Vi har tagit vanorna från Beck [1]. Vanorna är:

- **Kund på plats**

  I ett XP-projekt måste kunden vara på plats under hela utvecklingsarbetet. Kunden svarar på frågor från programmerarna, gör mindre prioriteringar samt skriver de funktionella testerna. Kunden har en viktig roll i ett XP-projekt.

- **Kort Releasecykel**

  Det är viktigt att releasa tidigt och ofta. Det ska inte vara en demoversion utan en riktig version som ger kunden affärsvärde. Man startar med minsta möjliga funktioner som skapar värde för kunden och lägger sedan till ett fåtal funktioner i varje release.

- **Planering**

  I planeringen bestäms nästa uppgift som skall utföras och vilka datum de olika delarna skall vara klara. Planeringen är uppdelad i två delar. Releaseplanering där kunden presenterar önskade funktioner som programmerarna uppskattar tiden på. Iterationsplanering där gruppens arbete planeras. De hålls vanligtvis varannan vecka.

- **Systemmetafor**

Varje projekt har en övergripande metafor som är en enkel historia som beskriver hur systemet skall fungera. Metaforen ersätter vad andra utvecklingsprocesser kallar för arkitektur.

- **Enkel design**

  Ett systems design ska vara så enkel som möjligt. Den får inte ha dubblerad logik, så få klasser och metoder som möjligt samt gå igenom alla tester.

- **Parprogrammering**

  All produktionskod skrivs av två programmerare som sitter vid en dator. De har olika roller. Den som skriver tänker på implementationsdataljer i aktuell del. Den andra programmeraren tänker mer strategiskt.

- **Kollektivt ägande av koden**

  Alla medlemmar i projektet kan ändra vilken kod som helst i projektet när som helst.

- **Kontinuerlig integration**

  Koden integreras och testas omedelbart, minst en gång per dag. Om testerna misslyckas så måste koden rättas innan den får integreras. Går det inte att korrigera koden så att den går igenom testerna ska den slängas.

- **Omfaktorisering**

  Går ut på att omforma systemet utan att förändra funktionaliteten. Meningen är att förbättra kommunikation, enkelhet och flexibilitet. Det kan också öka chanserna att komponenterna blir återanvändbara.

- **Kontinuerlig testning**

  Testerna består av enhets- och funktionalitetstester.

- **40 timmars arbetsvecka**

Behovet av en andra övertidsvecka i rad signalerar att någonting är fel i projektet. XP kräver att programmerarna är fokuserade och detta blir de om de inte jobbar för mycket övertid.

- **Kodningsstandard**
  Alla involverade i projektet ska använda samma stil när de skriver koden. Det ska alltså inte gå att se vem som skrivit koden genom att undersöka stilen.

2.2.2 XP:s fyra värden

Enlig Beck [1] har XP fyra värden:

- **Enkelhet**
  Betyder att man ska göra systemet så enkelt som det går. Systemet skall vara så simpelt som möjligt vid alla tidpunkter. Enkelhet kan vara svårt att åstadkomma. Inga onödiga funktioner som inte används får förekomma.

- **Kommunikation**
  För att ett projekt ska bli framgångsrikt så måste alla i gruppen kunna kommunicera med varandra så att alla vet vad som händer. Dålig kommunikation kan leda till att ett helt projekt misslyckas. I XP så föredras muntlig kommunikation före skriftlig kommunikation. Tanken med att kunden ska vara närvarande under hela projektet är för att förbättra kommunikationen mellan kunden och programmerarna.

- **Återkoppling**
  Återkoppling är viktigt för att lyckas med ett projekt. Programmerarna skriver enhetstester som ger direkt återkoppling då de kan kontrollera resultatet av testerna. Kunden ansvarar för acceptanstest i varje iteration. Återkoppling fungerar tillsammans med kommunikation och enkelhet. Med återkoppling blir det lättare att kommunicera.

- **Mod**
  Programmerarna uppmanas att ha mod att skriva om kod och omfaktorisera. Mod att kasta koden och börja om ifall det inte går att lösa problemen.

Hur värderingarna hänger ihop

De fyra värderingarna enkelhet, kommunikation, återkoppling och mod är beroende av varandra. Mer pengar kan hjälpa till att förbättra utvecklingen men för mycket pengar kan skapa mer problem än fördelar. Mer tid kan utöka omfattningen och förbättra kvalitén. Får projektet för lite tid så blir kvaliteten lidande. Mindre omfattning kan visserligen öka kvalitén men det måste vara tillräckligt så att det löser kundens problem.

2.2.3 Roller

XP fokuserar på de två rollerna programmeraren och kunden. XP specificerar vissa rättigheter och skyldigheter för de olika rollerna i projektet. Reglerna är viktiga för projektets medlemmar och för att projektet skall lyckas.

Kunden

Kunden har en viktig roll i ett XP projekt på grund av att det är den person som känner till de funktionella kraven på programmet. Det är också viktigt att han kan kommunicera med programmerarna så att de förstår vad kunden vill. Kunden ska vara på plats under hela tiden och vara med och driva projektet, svara på frågor från programmerarna, göra prioriteringar utföra acceptanstester samt att representera slutanvändaren.

Programmeraren

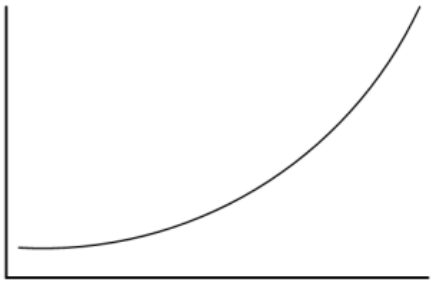
Programmerarens uppgift är att skapa fungerande kod utifrån kundens berättelser. I det ingår analys, test, programmering och integration av systemet. Programmerarrollen i XP skiljer sig dock en del mot den samma i mera traditionella utvecklingsprocesser. En viktig egenskap för programmeraren är att kommunicera med andra människor.

Andra roller

Det finns andra roller i ett XP-projekt som spårare, testare, handledare och konsult. De är inte lika viktiga som programmeraren och kunden och måste inte finnas som en formell roll i projektet. [1]

2.2.4 Kostnad för ändringar

Vid traditionell systemutveckling så antas det att kostnaden för att göra ändringar i ett program stiger exponentiellt över tiden. Se kostnad för ändringar Figur 2.1. Kent Beck menar att detta inte är giltigt längre om man använder sig av XP:s tekniker och vanor så stiger inte kostnaden för ändringar dramatiskt över tiden utan ligger ganska konstant. Se kostnad för ändringar Figur 2.2



Figur 2.1: Traditionell kurva över kostnad för ändringar

Figur 2.2: Kent Beck's kurva över kostnad för ändringar

Att kostnaden inte stiger dramatiskt över tiden är en av sakerna som gör XP möjligt på grund av att XP går ut på att göra stora val så sent som möjligt i processen. Det skulle inte vara möjligt med den traditionella kostnadskurvan. Tekniskt sett så är det användandet av objekt som möjliggör detta då objekt är lätta att återanvända.

2.3 Val och motivering

Vi har valt att använda UP som utvecklingsprocess för att utveckla vårt program på grund av det omfattande kravet på dokumentation. Vi tror därför att UP passar bättre för detta arbete då det verkar mer strukturerat än Extremprogrammering. Detta är en fördel när vi ska skriva rapporten.

En annan fördel med UP är att UML används av UP. UML är ett standardiserat språk för modulering av mjukvara. Modulering gör det enklare att förstå komplicerade system, kommunicera mellan projektmedlemmar samt att ta fram en bra arkitektur. Detta är till fördel då det kan vara svårt att förstå hur ett system fungerar genom att bara studera kod eller genom att någon förklarar hur det fungerar. UML medför med andra ord att risken för missförstånd minskar.

UP använder sig också av iterationer för att dela upp arbetet med väl definierade milstolpar och har ett system för att minimera risker. UP använder sig av användningsfall för att dela upp arbetet i mindre delar. Dessa delar kan sedan prioriteras så att användningsfall som är viktiga för programmet implementeras tidigt i projektet. Detta gör att riskerna med att projektet till exempel blir försenat minskar.

Valet var dock inte självklart då XP har vissa väldigt lockande egenskaper som parprogrammering och friare rörlighet vad gäller kravspecifikationen. En orsak till att vi inte valde XP är att den utvecklingsprocessen kräver väldigt erfarna programmerare enligt [2].

Vi var även nyfikna på hur UP fungerade och hur det skulle vara att utveckla ett projekt med hjälp av UP. Det kändes också som att det skulle vara en bra erfarenhet att ha använt UP som utvecklingsprocess under arbetet med ett program. Detta var en stor anledning till varför vi valde UP som utvecklingsprocess.

3 XML

XML står för ”Extensible Markup Language” och är ett standardiserat regelverk för att skapa märkspråk ett så kallat metaspråk [8]. XML är en rekommendation framtagen av ”World wide web consortium” (W3C). Det är oberoende av hårdvara och mjukvara, skriftspråk och själva informationen. XML är skapat så att det kan utvidgas av användaren genom att man kan skapa sina egna märkord. XML skiljer på struktur, presentation och data. XML passar både som lagrings- och överföringsformat för att knyta ihop olika system. [8]

3.1 DTD

DTD står för ”Document Type Definition” och används för att kontrollera strukturen av till exempel SGML- och XML-dokument. En DTD är ett regelverk för hur dessa dokument får vara strukturerade. DTD:n beskriver varje element som får vara med i SGML eller XML-dokumentet och vilka attribut de tillåts att ha. Även ordningen av elementen kan begränsas i DTD:n.

DTD har dock en del problem

- DTD följer inte reglerna för hur ett XML-dokument ska se ut och eftersom en DTD kan vara en del av ett XML-dokument så gör detta också att XML-dokumentet inte kan tolkas som korrekt XML. Detta kan ställa till problem då program skrivna för att tolka XML-dokument inte kan tolka den del av filen som utgör DTD definitionen.

- Att skriva en DTD är en bra metod för att kontrollera så att strukturen av det dokument som skall kontrolleras är korrekt, men är sämre på att kontrollera innehållet i elementen. Detta kan leda till att element som endast får innehålla numeriska värden för att fungera korrekt även får innehålla alfanumeriska värden enligt DTD:n. Detta kan i sin tur ställa till med stora problem om dokumentet som

kontrolleras till exempel används för att lagra och läsa in inställningar i ett program. Detta beror på att DTD:n endast kan ange vilka värden som är tillåtna i elementet explicit och inte om det till exempel endast är tillåtet med positiva heltal.

- DTD har inget stöd för namnrymder vilket leder till att det är svårt att återanvända kod då man ofta stöter på problem med att olika ”kontrolldefinitioner” har samma namn men kontrollerar olika saker.

Även om dessa nackdelar väger tungt har DTD fördelen att det är relativt enkelt att komma igång med då syntaxen är enkel att lära sig och inte alltför komplicerad. [11]

3.2 XML-schema

XML-schema är ett språk för att beskriva giltig struktur och innehåll i ett XML-dokument. Schemat beskriver giltig struktur, datatyper och begränsningar för element eller attribut i ett XML-dokument.

Standardiseringen av XML-schema styrs av W3C. I mars 2001 så publicerades XML-schema som kandidatversion vilket innebär att tekniken anses vara stabil. Det är ett nyare sätt att beskriva struktur och innehåll för XML-dokument. (Den äldre varianten är DTD). Fördelen med XML-schema är att det är mycket mer kraftfullt än DTD. [6]

3.2.1 Datatyper

XML-schema skiljer på enkla och komplexa datatyper. Enkla datatyper är de som inte innehåller några element eller attribut. Komplexa datatyper kan innehålla flera element och attribut.

XML-schema innehåller en stor mängd definierade datatyper liknande ett vanligt programmeringsspråk. En primitiv datatyp är en grundläggande datatyp som inte kan delas upp i enklare datatyper. Det finns 19 primitiva datatyper. Exempel på primitiva datatyper är string, boolean och double. Det finns också härledda datatyper som kan härledas ur de primitiva datatyperna. Exempel på sådana datatyper är short och nonNegativeInteger.

Trots att X-ml-schema har så många definierade typer så kan det vara nödvändigt att definiera egna datatyper. Om man till exempel har något slags personregister där man vill lagra en persons ålder så är inte värden som 255 rimliga. Då kan man skapa en egen datatyp som begränsar värdet mellan 0 och 120. Man utgår då från en existerande datatyp som exempelvis ”positiveInteger” och anger en begränsning av det högsta värdet till 120.

Vill man begränsa innehållet ytterligare i ett element kan man använda reguljära uttryck. Det är en sorts mönsterpassning och används för att matcha ett visst mönster av tecken. Detta är användbart om man till exempel vill kontrollera om en e-postadress är giltig.

3.2.2 Objektorienterat

XML-schema har många mekanismer som underlättar återanvändning. Det är lätt att återanvända delar på grund av att det går att dela upp schemat i komponenter. Nya scheman kan även ärva ifrån existerande scheman. Det finns stöd för arvsmekanismer som abstrakta element som inte kan förekomma i instanser utan som andra element kan ärva av.

XML-schema stödjer ”XML Namespaces” vilket innebär att ett schema kan ha olika namnrymder. Fördelen med detta är att man kan skilja på objekt med samma namn vilket är viktigt om man arbetar med data från flera olika organisationer. Varje namnrymd har ett prefix. Namnrymder identifieras ofta med hjälp av en ”unified resource locator” (URL) som representerar företaget. Det är inte en sökväg till schemat utan ett sätt att få unika namn. [6]

Fördelar

XML-scheman är giltiga XML-dokument vilket innebär att det är möjligt att parsa dem med en XML-parser, manipulera dem med exempelvis DOM (Document Object Model), eller editera dem med en XML-editor. De kan alltså behandlas som vanliga XML-dokument. [6]

Nackdelar

Nackdelen med XML-schema är att det är så komplext vilket gör det svårt att lära sig.

3.3 Motivering

Då innehållet i XML-elementen är av lika stor betydelse som strukturen i vårt fall har vi valt XML-schema. DTD har nämligen en väldigt begränsad möjlighet att kontrollera innehållet i de element som finns i XML-filerna medan XML-schema ger möjligheten att definiera egna datatyper som sedan kan jämföras med innehållet. Det finns även en mängd inbyggda datatyper som går att använda i XML-schema. Detta gör att XML-schema är överlägset DTD då innehållet i elementen är av betydelse.

Då extra funktionalitet ska kunna läggas till i efterhand är XML-schema ett bra val då det har stöd för till exempel reguljära uttryck, polymorfism och namnrymder. Detta gör att mera komplicerade kontroller som inte används idag kan läggas till i efterhand utan att för den skull behöva byta metod för att kontrollera XML-filerna i den kod vi utvecklade.

XML-schema är dock mera komplicerat att skriva än DTD, men eftersom DTD inte klarar av att kontrollera det vi behöver kontrollera så kan inte DTD användas. Att XML-schema är svårare att skriva är helt enkelt något vi måste hantera för att kontrollen skall fungera som tänkt.

XML-schema skrivs enligt XML-standarden medan DTD skrivs på ett sätt som bryter mot denna standard. Detta gör att en XML-fil med en DTD inte följer XML-standarden medan en XML-fil med tillhörande schema gör det.

XML-schema är även en nyare teknik för att kontrollera innehållet i XML-filer än DTD och har tagits fram på grund av de begränsningar som finns i DTD. Detta gör att DTD får en mera tillbakaskjuten roll då XML-schema tar över mer och mer. Detta gör att XML-schema är ett bättre val än DTD med tanke på framtiden.

DTD är visserligen bra på att kontrollera strukturen på ett enkelt XML-dokument, men för vår specifika uppgift är det för simpelt och begränsat. Vi var därför tvungna att välja XML-schema för att lösa vår uppgift.

4 Genomförande

Eftersom vi valde att utföra vårt arbete efter den plan som UP ger så har vi delat upp projektet i iterationer. Varje iteration är väl avgränsad och eftersom UP beskriver fyra faser samt att vårat projekt är så pass litet har vi valt att genomföra arbetet i fyra iterationer där varje iteration är begränsad till en fas.

De olika faserna har olika mål. I förberedelsefasen är det viktigt att komma fram till vad programmet ska göra samt att göra en avgränsning. I etableringsfasen ligger fokus på att ta fram riktlinjer för arkitekturen och att reducera risker. I konstruktionsfasen utvecklas systemet till den punkt där den planerade funktionaliteten är färdigimplementerad. I överlämningsfasen som är den sista ser man till att programmet överlämnas till användarna och att det fungerar korrekt i slutmiljön.

Iterationerna under arbetet är dock inte helt avgränsade från varandra utan kommer att ”lappa” över varandra. Det vill säga när till exempel iteration två påbörjas håller arbetet med att slutföra iteration ett fortfarande på. Detta gör att saker som upptäckts i den föregående iterationen men inte genomförts enkelt kan planeras i början av nästa iteration medan den planeras. Se bilaga B

Figur 4.1: Översikt av iterationerna

4.1 Förberedelsefas

I denna iteration var målet att ta fram en kravspecifikation samt att reducera riskerna med projektet genom att identifiera de viktigaste delarna av den färdiga produkten så att de kunde genomföras så tidigt som möjligt. Eftersom denna fas hade som mål att ta fram en kravspecifikation och planera resten av arbetet var det svårt att planera själva iterationen och

vi började därför med att titta igenom den specifikation vi fått från företaget för att hitta eventuella problem.

Tillsammans med uppdragsgivaren så tog vi fram en kravspecifikation samt begränsade projektets omfattning så att vi sedan kunde använda detta i vår planering. Bilaga A Kravspecifikation innehåller information om vilken funktionalitet programmet ska tillhandahålla samt en kort beskrivning av denna. Eftersom det ställdes krav på att programmet skulle vara lätt att använda då målgruppen bestod av datorovana personer fick vi tänka efter så att programmet skulle bli lätt att förstå. För att lättare kunna bedöma om programmet var användarvänligt tog vi fram en prototyp som sedan bedömdes av uppdragsgivaren.

4.1.1 Fånga kraven som användningsfall

Utifrån kravspecifikationen kunde vi ta fram de användningsfall som behövdes för att programmet skulle kunna klara av de krav som ställts. För att ge en bra överblick över de användningsfall och aktörer som identifierats togs användningsfallsmodell fram som presenterade dessa på ett enkelt sätt. Se bilaga C UML Diagram.

De aktörer som identifierades var

- **Användare**

  Användaren är den person som kommer att använda den färdiga versionen av programmet för att göra ändringar i databasen. Detta kan vara en datorovan användare.

- **Administratör**

  Administratören är den person som konfigurerar programmet åt användarna genom att lägga till och redigera de XML-filer som programmet använder. Administratören är en datorvan användare som har goda kunskaper om XML och databaser.

De användningsfall som identifierades

- **Lägg till data**

  Användaren har möjlighet att lägga till ny data i databasen. Den inmatade informationen kontrolleras i logikdelen så att den följer reglerna som är specificerade i XML-filerna.

- **Ta bort data**

  Användaren har möjlighet att ta bort data i databasen.

- **Redigera data**

  Användaren har möjlighet att redigera data i databasen. Den uppdaterade informationen kontrolleras på samma sätt som i ”lägg till data”. Ändringen sparas i en loggfil.

- **Uppdatera data**

  Data från vald tabell hämtas från databasen.

- **Filtrera data**

  ”Filtrera data” är en del av användningsfallet ”uppdatera data” och ger användaren möjlighet att filtrera den data som visas.

- **Läs in XML-filer**

  Information om reglerna för tabellerna läses in från XML-filer.

- **Kontrollera XML-filerna**

  Administratören ska ha möjlighet att kontrollera att XML-filerna är giltiga.

Användningsfallen kommer att bli mer detaljerade i etableringsfasen.

4.1.2 Identifiera de ickefunktionella kraven

När användningsfallen var framtagna det vill säga programmets funktionalitet var framtagen så identifierades de ickefunktionella kraven. De ickefunktionella kraven brukar delas upp i användbarhet, tillförlitlighet, prestanda, underhållbarhet och designkrav. [9]

De ickefunktionella kraven specificerades tillsammans med uppdragsgivaren. För en komplett lista över dessa se bilaga A Kravspecifikation.

- **Dokumentation**

  En användardokumentation och en dokumentation för administratören krävdes. Dessa finns i bilaga D Användardokumentation respektive bilaga E Användardokumentation Administratör.

- **Användbarhet**

  Eftersom programmet ska användas i stressiga miljöer har vi lagt ned mycket arbete på att göra det användarvänligt och lätt att förstå. Vi har därför tittat på liknande produkter som används och försökt att efterlikna dem till stor del så att användarna lätt ska känna igen sig. Bland dessa produkter finns program som företaget tidigare använt för samma målgrupp.

- **Prestanda**

  Då programmet ska användas för att mata in och underhålla information i databasen så är det viktigt att det inte tar för lång tid att hämta posterna från databasen. Då de tabeller som visas upp i programmet kan innehålla stora mängder data så fanns det även krav på att användaren ska kunna begränsa hur mycket data som hämtas från databasen.

  Hur lång tid det tar att hämta de poster som begärts beror då förstås på mängden data i varje post samt hur många som begärts. Programmet ska dock klara att hämta 500 poster per sekund.

- **Säkerhet**

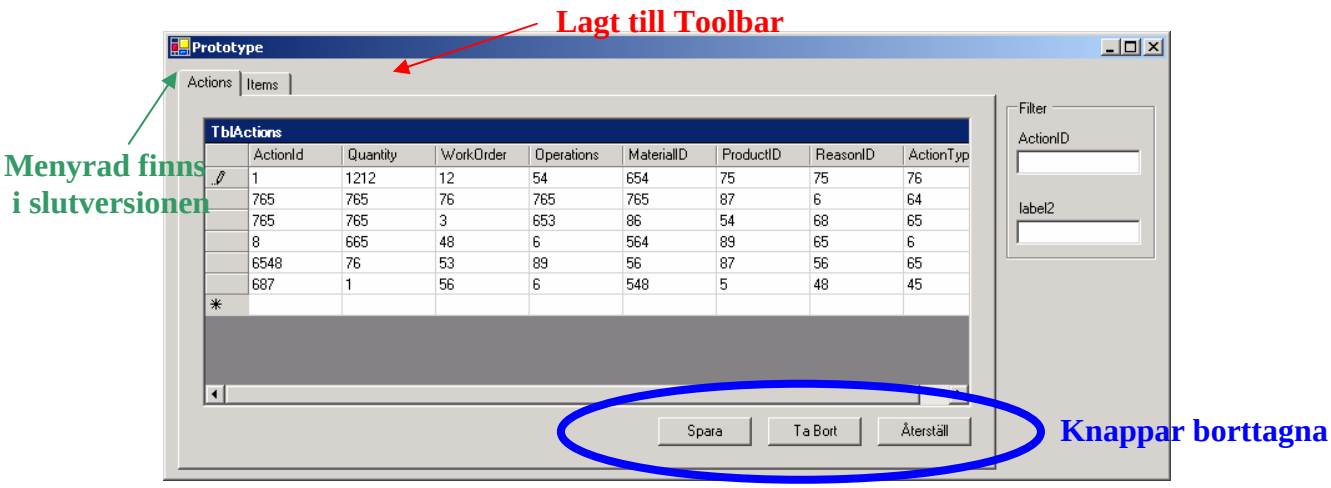
  Programmet skall logga de ändringar som användaren gör i programmet samt erbjuda en högre grad av loggning om programmet skulle behöva vidareutvecklas. Det som ska loggas är när användaren lägger till, uppdaterar eller tar bort data samt tiden för detta. Integritetshanteringen sköts av databasmotorn.

4.1.3 Prototyp

En prototyp är en första version av programmet som används för att demonstrera funktionalitet och visa på olika designval och framförallt för att bättre förstå problemet och olika lösningar. En prototyp hjälper till att minska risken för missförstånd i kravspecifikationen. [10]

Vi valde att göra en prototyp på grund av att det är en bra hjälp för att lösa problem med användargränssnittet som kan vara svåra att beskriva i text och diagram. Detta ger även användarna möjlighet att komma med synpunkter på designen.

För att kunna visa upp en bild av hur vi tänkt att programmet skulle se ut gjorde vi en användargränssnittprototyp i den tänkta utvecklingsmiljön. Denna prototyp användes för att förklara hur vi tänkt lösa uppgiften. Uppdragsgivaren fick en chans att komma med synpunkter. Prototypen gav även en bra bild över de användningsfall som det färdiga programmet skulle klara av. Se Figur 4.2



Figur 4.2: Användargränssnitt prototyp

Uppdragsgivaren hade vissa synpunkter på hur prototypen såg ut och detta gjorde att vi enkelt kunde anpassa designen. Uppdragsgivaren ville till exempel att knapparna spara, ta bort och återställ skulle flyttas till en så kallad ”toolbar” längst upp i programmet. Uppdragsgivaren ville även att det skulle finnas en meny i programmet där dessa knappar skulle finnas som val tillsammans med en avsluta knapp.

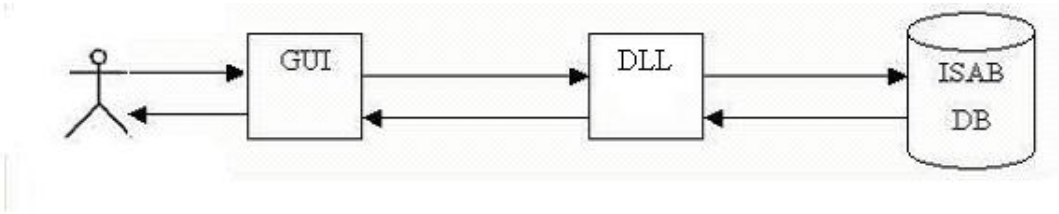
4.1.4 Hantering av risker

Det var viktigt att identifiera riskerna i projektet och försöka minska dem så mycket som möjligt. En stor del av riskhanteringen går ut på att identifiera användningsfallen som har störst risker och utföra dem först. Se kapitel 4.1.6 Prioritera användningsfall.

Vi fann också risker som berodde på användandet av ny teknik. Då varken företaget eller vi använt .NET i någon större omfattning kan man se det som en risk. Ett av kraven var att vi skulle använda oss av Microsofts .NET ”DataGrid” för användargränssnittet. Då vi inte arbetat med denna tidigare satte vi upp den som en risk.

4.1.5 Design

Den färdiga produkten skulle enligt kravspecifikationen vara uppdelad i två delar, användargränssnitt och dll, där en del är ett gränssnitt mot den andra. Vi var därför tvungna att lägga tid på att på ett bra sätt separera de två delarna.



Figur 4.3: Design treskiktslösning

4.1.6 Prioritera användningsfall

För att reducera riskerna prioriterade vi de användningsfall vi tagit fram. Genom att planera att genomföra de viktiga och svåraste användningsfallen så tidigt som möjligt i projektet kunde vi reducera risken att projektet skulle ta längre tid än beräknat. Eftersom vi gjorde detta så tidigt i projektet var det möjligt att hantera användningsfallen på ett bra sätt.

Vi kom fram till följande prioritering:

Användningsfall	**Prioritering**
Läs in XML	1
Uppdatera data	2

Lägg till data	2
Redigera data	2
Ta bort data	2
Kontrollera XML	3

Tabell 4.1: Prioritering av användningsfall

4.1.7 Planera etableringsfasen

Mot slutet av förberedelsefasen var det dags att planera nästa fas i projektet. I etableringsfasen ska de användningsfall som tagits fram beskrivas mera grundligt och därför såg vi till att ta fram en lista över de användningsfall som fanns i programmet och kontrollerade med uppdragsgivaren om dessa var korrekta.

Vi gjorde även bedömningen att projektet var så pass begränsat att etableringsfasen skulle genomföras i en iteration. Att projektet var så begränsat ledde även till att vi bedömde att den analys som ska göras i etableringsfasen enligt UP var något överdriven och vi bestämde oss därför för att begränsa denna något.

4.2 Etableringsfas

Efter förberedelsefasen så kunde etableringsfasen startas. Målet var att ta fram riktlinjer för en bra arkitektur, att reducera de kvarvarande riskerna med projektet samt att utveckla projektplanen med mer detaljer.

4.2.1 Detaljera användningsfall

Enligt Jacobson et al [5] så ska man i etableringsfasen detaljera ett användningsfall som har speciellt betydelse för arkitekturen. Vi valde då att detaljera användningsfallet ”Uppdatera data” då det är viktigt för arkitekturen. De andra användningsfallen är beroende av detta. När detta gjordes fick vi fram riktlinjer för arkitekturen som vi sedan kunde arbeta efter.

Vi upptäckte att vi missat att lägga till användningsfallet ”Filtrera”. Detta användningsfall hade vi i föregående fas tolkat som en del av ”Uppdatera data” användningsfallet, men

eftersom detta är något som slutanvändaren utför separat ansåg vi att detta är ett separat användningsfall. Användningsfallet är dock beroende av ”Uppdatera data” då filtren begränsar vilken data som skall hämtas. Vi kom även fram till att de användningsfall som administratören kan utföra måste implementeras först eftersom alla andra användningsfall är beroende av dessa.

Efter att filtrera lades till blev prioriteringen av användningsfallen följande:

Användningsfall	**Prioritet**
Läs in XML	1
Uppdatera data	2
Lägg till data	2
Redigera data	2
Ta bort data	2
Filtrera	3
Kontrollera XML	3

Tabell 4.2: Ny prioritering av användningsfall

4.2.2 Analys

Vi har valt att inte uppdatera analysen senare i projektet utan har använt analys som ett hjälpmedel för att bättre förstå uppgiften och ej som en del av dokumentationen. Här avviker vi från Jacobson et al [5] på grund av att vi tycker att designen av användningsfallen är möjlig utan analys. Vi tror även att designen är tillräcklig i vårt projekt.

Då de flesta av de användningsfall som vi tog fram i förberedelsefasen var komplexa och beroende av funktionalitet i logikdelen för att läsa in och hantera XML-filer så bedömde vi att vi behövde analysera dessa. Se bilaga C.2 samarbetsdiagram

Under analysen kom vi fram till att så gott som alla användningsfall använder sig av samma bakomliggande kod även om varje användningsfall kräver ytterligare funktionalitet från de redan existerande klasserna. Tack vare detta ansåg vi oss ha genomfört den nödvändiga analysen genom att bara analysera ett användningsfall.

Vi valde då att analysera användningsfallet ”uppdatera” då det var det första användningsfallet enligt vår prioritering som involverade konstruktion av nya klasser.

4.2.3 Design

Enligt Jacobson et al [5] så ska man endast designa ett fåtal av användningsfallen i etableringsfasen. Bara de användningsfall, klasser och delsystem som har en stor betydelse för arkitekturen av programmet analyseras och designas i denna fas.

Därför har vi designat användningsfallet ”Läs in XML-filer”. Vi diskuterade flera olika designlösningar och kom till slut fram till en lösning som vi båda kunde enas om. För att kunna följa denna design i projektet skapade vi ett antal diagram se bilaga C UML Diagram. Dessa diagram beskriver hur de olika delarna av programmet fungerar tillsammans.

Vi valde att inte göra diagram för alla delar av programmet utan endast för de som var av relevans för arkitekturen.

Identifiera arkitekturen

I designen ingår det att ta fram en skiktad arkitektur. Det var till stor del redan klart då det fanns krav från uppdragsgivaren på en treskiktslösning med användargränssnitt, logik och databas. Logikdelen skulle vara skriven som en dll-fil. Se bilaga A.2.3 Arkitektur. Vi fick dock lägga ned tid på att separera de olika delarna från varandra.

Resultatet av denna aktivitet var sekvensdiagram, klassdiagram och krav på implementationen. Se bilaga C UML Diagram.

Avändningsfall-Läs in XML

Vi valde att implementera ”Läs in XML” först eftersom vi bedömt att detta användningsfall var av stor betydelse för att de andra användningsfallen skulle gå att implementera. Användningsfallet ”Läs in XML” läser nämligen in de XML-filer som finns och skapar en ”struktur” av objekt som de andra användningsfallen är beroende av.

Detta beror på att XML-filerna specificerar till exempel vilket fält i varje tabell som är primärnyckel. Detta används sedan av till exempel användningsfallet ”redigera data” då programmet måste specificera vilken rad som skall ändras i databasen. Detta genomförs genom att ange värdet på primärnyckeln på den raden som skall ändras och därför måste denna vara känd. Även användningsfallet ”ta bort” fungerar på detta sätt.

Vi hade även tidigare kommit fram till att om vi läste in varje del av XML-filerna endast när vi behövde dem så skulle det behövas väldigt många metoder för att hämta informationen. Dessa metoder skulle också bli relativt stora eftersom XML-filerna är dynamiska och därför gjorde vi så att vi skapade egna klasser som enkelt kan lagra de olika inställningarna. I dessa klasser skapade vi sedan metoder för att enkelt kontrollera de olika inställningarna.

4.2.4 Implementation

Implementationsmomentet i denna iteration blev omfattande trots att UP rekommenderar att endast en liten del av användningsfallen som identifierats ska implementeras i etableringsfasen. Anledningen till detta är att användningsfallet som implementerades var omfattande och att flera av de resterande användningsfallen var beroende av detta användningsfall.

Användningsfall - Läs in XML

Implementationen av ”Läs in XML” använder sig av färdiga komponenter från Microsofts .NET paket för att läsa in XML-filerna. Dessa komponenter har stöd för att kontrollera innehåll i XML-taggar och attribut. Detta gör att endast kod för att tolka det XML-format vi använt behövt skrivas.

Inläsningen av XML-filerna sker i flera steg då det finns en fil ”main.xml” som innehåller information om vart de andra XML-filerna är placerade. Denna fil läses in först och när de andra filerna är kända läses även de in. Varje tabell som skall kunna användas i programmet har en egen XML-fil som beskriver tabellen och dess fält.

Den kod som behövs för att tolka XML-filerna är dock inte helt enkel då de träd som skapas utifrån XML-filerna i vissa fall är komplexa. Detta beror på att de XML-filer som

programmet använder inte alltid innehåller alla element och attribut då dessa i vissa fall inte är nödvändiga.

För att lagra de olika inställningarna skapades en klass för tabeller (Table) och en klass för fälten (Field). Därefter skapades även en abstrakt klass (Validator) som superklass för de olika valideringsklasser som behövdes i programmet.

Objekten som skapas ifrån de olika ”validerare” som gjorts lagras i de fält som är bundna till dem enligt XML-filerna. På samma sätt lagras fälten i de tabeller som de är bundna till enligt XML-filerna. Dessa tabeller lagras i en lista. Genom att iterera listan kan man sedan få information om den tabell man är intresserad av.

XML-inläsningen läser inte bara in de tabeller, fält och ”validerare” som finns i programmet utan även en fil som anger vilken databas som skall användas och hur man ansluter mot denna. (adress och inloggning) Då det inte fanns några krav på att denna information skulle skyddas är den lagrad som vanlig text i denna XML-fil.

Eftersom programmet konfigureras dynamiskt utifrån XML-filer skrevs även gränssnittet så att det kunde anpassas efter de tabeller som används. Gränssnittet skapades så att det enkelt går att lägga till tabeller genom att skapa nya ”TabPages”. En ”TabPage” kan beskrivas som en flik i en mapp med en liten text på. När användaren klickar på fliken öppnas en sida med tabellen på.

När programmet startas kontrollerar gränssnittet genom logiken vilka tabeller som skall visas, hur dessa ser ut och skapar sedan flikarna.

4.2.5 Test

Eftersom programmet är uppdelat i de tre delarna, användargrässnitt, logik och databas så valde vi att använda oss utav automatiserade tester på logikdelen och manuella tester på användargränssnittet. Detta till stor del på grund av att det är svårt att testa användargränssnittet med automatiserade tester.

Användningsfall - Läs in XML

För att testa ”läs in XML” skrevs några enkla XML-filer samt ett program som ”frågade” logiken vad som lästs in. De XML-filer som skrevs innehöll alla olika element och attribut som programmet förväntas klara av och genom att kontrollera värdet på dessa kunde användningsfallet enkelt testas.

4.2.6 Planera konstruktionsfasen

Mot slutet av etableringsfasen var det dags att planera nästa fas nämligen konstruktionsfasen. I konstruktionsfasen ska de användningsfall vi planerat implementeras och testas. Vi gjorde även inför denna fas bedömningen att vårt projekt var så pass begränsat att konstruktionsfasen kunde genomföras i en iteration. Mycket av planeringen inför konstruktionsfasen blev i vårt fall överflödig då vi bara är två personer i projektet och stor del av denna vanligtvis går till att administrera hur de olika grupperna av människor ska samarbeta. Vi kontrollerade de användningsfall vi tagit fram och uppskattade hur lång tid varje användningsfall skulle ta att implementera. Till sist kontrollerade vi vår uppskattning gentemot vår planering för att se om denna behövde korrigeras.

4.3 Konstruktionsfas

Målet i konstruktionsfasen är att analysera, designa, implementera och testa de resterande användningsfallen. Konstruktionsfasen leder med andra ord fram till en första körbar version av programmet där all funktionalitet som krävts är implementerad och testad.

4.3.1 Krav

Eftersom vi detaljerade alla våra användningsfall i etableringsfasen då de var så få genomfördes detta arbetsflöde egentligen i den fasen. Vi har därför helt enkelt bara kontrollerat de diagram och beskrivningar vi då gjorde för att försöka upptäcka eventuella felaktigheter.

4.3.2 Analys & Design

Då vi valde att endast analysera de komplexa delarna eller de som är relevanta för arkitekturen av programmet så behövde vi ingen detaljerad analys. Därför valde vi att slå ihop arbetsflödena analys och design till ett.

Användningsfall - Uppdatera data

Användningsfallet ”Uppdatera data” läser in data från databasen i given tabell och presenterar denna för användaren. För att hämta data från databasen tillåts gränssnittet fråga logikdelen efter en tabell. Logikdelen genererar sedan en SQL-fråga som hämtar data från databasen och skickar tillbaka denna till gränssnittet som presenterar den i en så kallad ”DataGrid”.

Användningsfall - Lägg till data

Användningsfallet ”Lägg till data” låter användaren lägga till rader i en tabell i gränssnittet och skickar sedan dessa ändringar till logikdelen. Varje gång en ändring görs i ett fält på den nya raden kontrolleras ändringen så att den stämmer överens med de regler som definierats för fältet i XML-filerna. Om det inmatade värdet bryter mot någon av de regler som angetts för fältet får användaren ett felmeddelande.

När användaren matat in de nya värdena kan de sparas genom att trycka på spara. Logikdelen anropas sedan med de nya data som ska sparas och denna genererar en SQL-fråga som lägger till de data som skickats in i given tabell.

Om databasen av någon anledning inte kan spara den data som skickats in returneras en felkod tillbaka till gränssnittet. Beroende på vilken felkod som returneras visas ett fördefinierat felmeddelande upp för användaren. Exempel på fel som kan uppstå är att databasen inte går att nå och liknande.

Användningsfall - Redigera data

Detta användningsfall låter användaren göra ändringar på befintliga rader i databasen och sedan spara dem. Detta fungerar på samma sätt som ”Lägg till data” förutom att SQL-frågan ser annorlunda ut.

Användingsfall - Ta bort data

Användningsfallet ”Ta bort data” fungerar även det som ”Lägg till data” med den skillnaden att SQL-frågan skiljer sig från de föregående två.

Användningsfall - Filtrera data

Användningsfallet ”Filtrera data” är en del av användningsfallet ”Uppdatera data” och ger användaren möjligheten att filtrera den data som skall hämtas från databasen. Användaren gör detta genom att ange vilka filter han/hon vill använda i ett antal textrutor.

Vilka fält som användaren kan använda filter på är angett i XML-filerna. De filter som användaren anger klarar av att hantera så kallade jokertecken.

Användningsfall - Kontrollera XML

Kontrollen av XML-filerna med XML-schema sker med ett program som administratören har tillgång till. Kontroll av XML-filerna är nämligen inte nödvändig för att programmet ska fungera så länge de är korrekt skrivna. Eftersom användaren inte kommer att göra några ändringar i XML-filerna behöver de heller inte ha tillgång till detta program.

Kontrollen använder sig av existerande XML-schemafiler som kan kontrollera så att de olika XML-filerna är korrekta. Programmet är skrivet med hjälp av .NET paketets komponenter för XML-inläsning och validering.

För att programmet ska vara till nytta för administratören ger det även information om vad som är fel i XML-filerna om de inte är korrekt skrivna.

Loggning

Detta användningsfall skiljer sig från de andra användningsfallen då loggen måste gå att använda i alla delar av programmet. Loggen måste nämligen klara av att skriva ned information om saker som att en knapp har tryckts ned eller att ett värde i databasen har ändrats.

Enligt kravspecifikationen måste det också gå att ändra vilka meddelanden som ska loggas beroende på vem som använder programmet. Standardnivån loggar endast de ändringar som användaren gör i databasen samt vilken tid de utförts. De högre loggnivåerna är till för att underlätta felsökning av programmet då de även loggar vilka metoder som anropas i programmet.

4.3.3 Implementation

Implementationen av följande användningsfall använder sig till stor del av den implementation som är gjord i användningsfallet ”Läs in XML”. Detta beror på att implementationen av användningsfallen är beroende av information som hämtats från XML-filerna.

Programmet visar tabellerna som hämtas i en ”DataGrid” som också låter användaren göra ändringar som sedan sparas när användaren trycker på en spara knapp.

Då programmet är skrivet med en så kallad treskiktslösning är alla användningsfall implementerade i det mellersta lagret med passande gränssnitt i det översta. Gränssnittet sköter all inmatning från användaren och visar även upp de tabeller som finns i databasen, men klarar inte av att göra några ändringar i databasen.

Användningsfall - Uppdatera data

Användningsfallet ”Uppdatera data” har till uppgift att hämta data från vald tabell i databasen och visa upp den för användaren. För att göra detta skrevs en klass som kan hantera inläsningar från databasen. Eftersom .NET paketet användes var detta inte speciellt svårt då det finns en mängd olika sätt att göra detta med befintliga klasser.

Det skrevs två metoder i klassen ”DBConnection” för att hantera inläsning av data från databasen. En av dessa räknar antalet poster i tabellen och den andra läser in ett givet antal poster i tabellen. Metoden för att räkna antalet poster är nödvändig för att programmet inte ska hämta poster som inte finns i tabellen genom att ange ett för högt antal poster.

En av de stora uppgifterna i detta användningsfall var att generera de SQL-frågor som behövs för att hämta information från databasen. Detta berodde på att frågorna måste vara dynamiska då användaren kan välja att hämta till exempel post 100-200 i en given tabell. Det finns visserligen stöd för att generera SQL-frågor i färdiga komponenter i .NET paketet, men de frågor de var kapabla att generera var tyvärr inte tillräckliga för uppgiften.

Användningsfall - Lägg till data

Användningsfallet ”Lägg till data” har vissa likheter med ”uppdatera data” då båda användningsfallen ställer SQL-frågor mot databasen. Frågorna skiljer sig dock avsevärt ifrån

varandra. ”Lägg till data” ställer nämligen en INSERT fråga till skillnad från ”uppdatera data”:s SELECT fråga.

För att lägga till data i databasen via en SQL-fråga måste innehållet kontrolleras så att det inte innehåller tecken som gör att SQL-frågan blir ogiltig. En SQL-fråga som lägger till data sätter nämligen den data som skall läggas till innanför citationstecken och om datan innehåller ett citationstecken kommer SQL-frågan att bli ogiltig. Detta problem löstes enkelt genom att den data som ska läggas till genomsöks efter dessa tecken. Om ett citationstecken hittas läggs det till ytterligare ett direkt efter det första vilket gör att de förlorar sin speciella mening i SQL-frågan.

För detta användningsfall skrevs två metoder i ”DBConnection”, en för att generera SQL-frågan som behövs samt skicka den data som lagts till och en för att korrigera datan så att eventuella specialtecken inte orsakar problem i SQL-frågan.

Metoden för att lägga till data tar emot ett ”DataTable” som skickas ifrån gränssnittet. Denna innehåller den tabell som tidigare hämtats från databasen samt de ändringar som gjorts i den. Metoden kontrollerar om nya poster har lagts till i denna och genererar sedan en SQL-fråga för insättningen.

Användningsfall - Redigera data

Användningsfallet ”redigera data” påminner till stor del om ”lägg till data”, men behöver en annan SQL-fråga. Denna fråga skiljer sig från INSERT-frågan och för att ändringar ska kunna göras i en specifik post måste även värdet på primärnyckeln vara känt.

Detta gör att koden för att generera denna fråga skiljer sig från den som gjordes i ”Lägg till” användningsfallet. Ett ”DataTable” används dock också för att skicka ändringarna även i detta användningsfall.

Koden för att redigera data i databasen placerades därför i samma metod som den för att spara poster som lagts till.

Användningsfall - Ta bort data

Användningsfallet ”ta bort data” är väldigt likt ”Redigera data” eftersom också det använder ett ”DataTable” för att skicka de ändringar som gjorts. En SQL-fråga genereras som tar bort posten ur vald tabell i databasen och denna ställs.

Användningsfall - Filtrera data

Användningsfallet ”filtrera data” skrevs som en del av ”uppdatera data” då de filter som användaren skriver in påverkar den SQL-fråga som används för att hämta data från databasen.

Filtren som sätts används genom att sätta in nyckelordet LIKE i SQL-frågan. När denna fråga ställs hämtas endast de poster som matchar uttrycket som står i LIKE. Detta gör att användaren även kan fylla i SQL specifika jokertecken. För information om dessa jokertecken se bilaga E Användardokumentation Administratör.

Användningsfall - Kontrollera XML

Kontrollen av XML-filerna med hjälp av XML-schema sker med hjälp av .NET bibliotekets komponenter för XML-inläsning och validering. Detta har implementerats i ett program skilt från användargränssnittet och logikdelen.

Valideringen sker med hjälp av en klass som heter XmlValidatingReader. XML-filerna jämförs med de scheman som skrivits för att kontrollera dem och om något är felaktigt i dem får administratören ett meddelande med information om vad som är fel och på vilken rad.

De XML-scheman som används är skrivna så att de klarar av att kontrollera innehållet i XML-filerna oavsett till exempel vilken tabell de beskriver.

Loggning

Loggningen har implementerats som en singeltonklass där en nivå på loggningen sätt vid uppstart av programmet. Beroende på vilken nivå som anges skriver klassen sedan meddelanden till en fil med dagens datum som namn. Alla meddelanden som ska loggas skickar sedan med en nivå till loggningsobjektet som kontrollerar om denna nivå är mindre än eller lika med den nivå som angetts vid programstart. Loggen lägger även till tidpunkten då varje meddelande lagts till i filen så att det enkelt går att kontrollera när specifika ändringar gjorts.

En singelton var det val som verkade bäst för detta ändamål då alla delar av projektet ska kunna skriva till loggen när de behöver det. Singelton är ett designmönster av kategorin ”skapande mönster”. Det ser till att endast en instans av en klass skapas och att den är tillgänglig globalt. [4] Detta medför i sin tur att koden som skrevs för loggningen blev enkel.

De meddelanden som skickas till objektet loggas till en fil med dagens datum som filnamn och ”.log” som filändelse. Detta gör att det skrivs en ny logg varje dag och raderna i loggen har även de en tidstämpel som anger när meddelandet skrevs.

4.3.4 Test

För att testa de olika användningsfallen som är implementerade i logikdelen skrevs ett program som gör anrop mot den och kontrollerar resultatet mot det som förväntas. Programmet anropar de metoder som erbjuds av logikdelen och kontrollerar om korrekta parametrar behandlas som de ska samt om felaktiga parametrar resulterar i väntat resultat.

Eftersom logikdelen skickar tillbaka felkoder om något skulle gå snett är det enkelt att kontrollera om felaktig data resulterar i att korrekt felkod returneras.

Uppdatera data (med filter)

För att testa ”uppdatera data” skrevs en metod i testprogrammet som ”frågar” logikdelen efter en viss tabell. Metoden testar även att hämta vissa specifika rader som sedan undersöks för att se om förväntad data returnerats. Testet kontrollerar även att korrekt felkod returneras när en icke-existerande tabell begärs.

För att testa så att filtren fungerar skrevs ytterligare en metod som testar vad som returneras om ett filter skickas med i begäran. För att testa att felaktiga filter inte gör att den SQL-fråga som genereras blir felaktig testades även vad som returneras om filtret innehåller tecken som har speciell mening i SQL-frågor. Exempel på sådana tecken är ”.

Användningsfall - Lägg till data

Test skrev även för att testa så att ”lägg till data” fungerar för korrekt data såväl som för data som bryter mot regler som finns i databasen. För att kontrollera att användningsfallet

fungerade korrekt kontrollerades sedan om den korrekta data som skickats in verkligen fanns i databasen och att försöket att lägga till felaktig data i databasen resulterat i korrekt felkod .

Användningsfall - Redigera data

”Redigera data” testades på liknande sätt som ”lägg till data” med skillnaden att testet genomfördes på redan befintliga poster i databasen. Testet gjordes genom att försök gjordes att lägga till både korrekt och felaktig data i enlighet med databasens regler. Felkoderna kontrollerades sedan samt att databasen tagit emot och ändrat den data som var korrekt.

Användningsfall - Ta bort data

”Ta bort data” testades på samma sätt som de två tidigare användningsfallen. Försök att ta bort icke-existerande rader gjordes såväl som existerande. Felkoder kontrollerades och databasen undersöktes för att se att rätt rader plockats bort.

Manuella test

För att testa användargränssnittet, programmet för kontroll av XML-filerna samt loggningen användes manuella test då ett automatiserat test för detta är svårt att åstadkomma.

För att testa gränssnittet provade vi att lägga till, redigera och ta bort data ur de olika tabeller som användes. I de tabeller som gav möjlighet att filtrera informationen som visades testades även detta.

Programmet för kontroll av XML-filerna testades genom att det skapades XML-filer med avsiktliga felaktigheter. Dessa filer kontrollerades sedan med hjälp av programmet för att kontrollera att rätt felmeddelanden visades.

4.3.5 Planering av överlämmningsfasen

Det var inte möjligt att planera överlämningsfasen i detalj då arbetet i fasen till stor del påverkas av återkopplingen man får från testerna. Vi planerade dock hur testerna skulle utföras. Vi kom överens med vår handledare på företaget att han skulle genomföra testerna.

4.4 Överlämningsfas

Målet i överlämningsfasen är att kunden ska få den färdiga produkten installerad och att denne ska ha godkänt produkten för användning. Kunden får alltså i denna fas testköra programmet och komma med synpunkter och klagomål på saker som eventuellt inte fungerar som kunden tänkt sig. De sista fixarna görs alltså i denna fas och avslutas när kunden är nöjd med produkten. Det vill säga när produkten fungerar som tänkt i kundens miljö.

4.4.1 Krav

Programmet installerades på en testmiljö hos företaget. Detta gjordes så att vi kunde kontrollera att alla nödvändiga filer inkluderades. Detta gjorde det också möjligt för uppdragsgivaren att utföra ett acceptanstest.

Ett acceptanstest utfördes av uppdragsgivaren i vilket det visade sig att programmet inte klarade de krav som ställts angående prestanda. Programmet klarade nämligen inte att hämta tillräckligt många poster från databasen i de tabeller som användes under en begränsad tid.

4.4.2 Design

Flera tester gjordes för att utreda vad den långsamma uppdateringen berodde på och de loggfiler som skrivits undersöktes noggrant för att hitta problemet. I dessa loggfiler visade det sig att det tog längre tid att hämta enstaka poster ur databasen än beräknat. Det visade sig att eftersom programmet uppdaterade en ”DataGrid” samtidigt som den hämtade poster ur databasen ritades denna om varje gång en post hämtats.

4.4.3 Implementation

Detta problem kunde sedan enkelt åtgärdas genom att koppla från den ”DataGrid” som användes under tiden uppdateringar gjordes. Kopplingen mot tabellen återställdes sedan så att den hämtade datan kunde visas.

Detta gjorde att uppdateringarna gick betydligt snabbare än tidigare och de prestandakrav som ställt på programmet var inte längre något problem.

Användardokumentation

Eftersom programmet ska användas av datorovana personer skrevs en användarmanual där programmets delar och funktionalitet beskrivs. Denna manual skrevs så att den ska vara enkel att förstå även för personer som aldrig använt programmet tidigare. Manualen finns i bilaga D Användardokumentation.

Det skrevs ytterligare en manual vars innehåll beskriver hur administratörer av programmet skall göra för att konfigurera programmet. Denna manual är skriven för personer som redan har god förståelse för hur datorer och databassystem fungerar. Denna manual finns i bilaga E Användardokumentation Administratör.

4.4.4 Test

Programmet testades en sista gång av uppdragsgivaren i testmiljön för att kontrollera om de korrigeringar som gjorts fungerade som tänkt samt att inget annat påverkats. Detta test klarade programmet och var nu klart för användning.

5 Slutsats

> "In theory, there is no difference between theory and practice. But, in practice, there is." (I teorin finns det ingen skillnad mellan teori och praktik. Men i praktiken finns det.)

Jan L.A. van de Snepscheut

Innebörden av ovanstående citat har bekräftats mer och mer under arbetet med detta projekt då de kunskaper vi fått under vår utbildning har testats praktiskt. Detta kapitel beskriver meningen av projektet, de problem som uppstått, de resultat som uppnåtts och tar även upp förslag för vidare arbete.

5.1 Mening

Det huvudsakliga målet med uppsatsen var att utveckla ett program med konfigurerbart användargränssnitt för Industrisystems databas (IsabDB). I programmet skall användarna kunna lägga till och uppdatera information i databasen. Informationen i databasen kommer

ifrån företagets ekonomisystem. Det finns ett behov att kunna lägga till och ändra denna då all nödvändig information kanske inte finns i affärssystemet.

Fördelen med programmet vi har skrivit är att det är konfigurerbart. Det går att anpassa till alla kunder även fast deras databaser kan se annorlunda ut. I nuläget har de varit tvungna att skriva separata program för de olika kunderna. Med det nya programmet är det bara att ändra i XML-filerna så att de passar det aktuella företaget.

5.2 Problem

Vi hade vissa problem med användningsfall. Vi tyckte det var svårt att dela upp programmet i användningsfall då stora delar av programmet var sådant som används av flera användningsfall. Vi konsulterade vår handledare angående detta och kunde sedan fortsätta med projektet. Detta gjorde dock att förberedelserna tog något längre tid än beräknat.

Vi underskattade risken med ny teknik då den komponent som användes för att visa upp tabellerna (DataGrid) var svår att använda. Vi var visserligen medvetna om att detta var en risk men den visade sig vara större än vad vi hade räknat med. Detta gjorde att mycket tid fick läggas ner för att lösa problemen som uppstod på grund av detta.

5.3 Lärdomar och erfarenheter

Under arbetet med detta projekt har vi använt UP och därmed också lärt oss att bättre förstå hur denna utvecklingsprocess fungerar. Under projektets gång har vi också fått bra insikt i varför det är så viktigt att ha en bra planering för hur arbetet ska utföras.

Vi har även fått lära oss att använda UML-diagram på ett bra sätt för att visa hur ett program fungerar. Detta är bra då många projekt använder dessa eller liknande diagram för att visa hur ett program fungerar eller är tänkt att fungera.

Eftersom programmet utvecklats i C# som vi inte har några tidigare erfarenheter i har vi även fått lära oss att använda detta. Det kan vara av stor betydelse då många företag som tidigare utvecklat program i till exempel Visual Basic har eller funderar på att byta till C#.

Vi har även fått erfarenhet av att utveckla program som använder sig av Microsofts .NET bibliotek. Detta är en värdefull erfarenhet då .NET biblioteket finns för många olika programmeringsspråk. Sannolikheten att vi stöter på .NET biblioteket i andra projekt är alltså relativt stor.

Användandet av XML-dokument för att lagra inställningar har också varit till stor nytta då vi fått lära oss hur dessa fungerar. XML används mer och mer i väldigt varierande applikationer så denna kunskap kan visa sig vara väldigt värdefull. Vi har även fått lära oss att använda XML-schema för att kontrollera innehållet i XML-filer och vilka fördelar det finns med att använda XML-schema jämfört med DTD.

5.4 Resultat

Eftersom vi valde att använda UP som utvecklingsprocess för arbetet har detta styrt hur vi arbetat under utvecklingen av programmet. Framtagande av UML-diagram är till exempel en del av arbetet med UP. Det tog tid att få fram alla diagram, men vi tycker att det var värt jobbet då diagrammen var ett bra verktyg för att utvärdera och vidareutveckla designen av systemet. Diagrammen var också bra att ha när vi kommunicerade med våra handledare på företaget och universitetet då de gjorde det lättare för dem att förstå hur systemet är uppbyggt.

Det har dock varit svårt att följa UP:s alla steg då vi inte använt oss av denna utvecklingsprocess tidigare. Vi har därför fått lägga ned tid på att bättre förstå hur ett utvecklingsarbete som följer denna utvecklingsprocess ska genomföras. UP definierar många olika roller i ett projekt. Då vi bara var två personer som genomförde arbetet så hade vi problem med att dela upp dessa roller.

Vi tycker dock att det har varit en bra erfarenhet att få använda UP under utvecklingsarbetet då det ger en bra syn på hur ett arbete kan utföras med hjälp av rigorös planering och dokumentering. Om vi fick valet att göra ett liknande arbete igen är det dock tveksamt om vi skulle välja att använda UP igen.

5.5 Förslag till vidare arbete

Det framkom ytterligare funktionalitet under processen som inte ansågs kritisk för programmet och därför inte var med bland kraven som ställdes från uppdragsgivaren. Denna funktionalitet hade varit trevlig att ha med, men då programmet utvecklats under en begränsad tid implementerades de inte. Den funktionalitet som inte implementerades var stöd för språkhantering och inloggning.

Med språkhantering menas att användaren ska kunna välja vilket språk som ska användas i programmet. Detta hade medfört att programmet varit användbart för en större kundkrets. En inloggning hade varit bra då till exempel loggningen som görs hade kunnat ha med information om vem som utfört specifika ändringar.

Referenser

[1] Beck, Kent, *Extreme programming Explained, embrace change*, Addison-Wesley, 1999

[2] Demarco, Tom & Boem, Barry, *The agile methods fray, Computer*, IEEE, 2002

[3] Don Wells, What is Extrem programming?, http://www.extremeprogramming.org/what.html (2004-10-11)

[4] Gamma, Erich & Graham, Ian, *Design patterns: elements of reusable objected-oriented design*, Addison-Wesley, 1998

[5] Jacobson, Ivar & Booch, Grady & Rumbaugh, James, *The Unified Software Development Process*, Addison-Wesley, 1999

[6] Jadeep, Roy & Anupama, Ramanujan, *XML Schema language: Taking XML to the next level*, Computer, IEEE, 2001

[7] Jeffries, Ron & Anderson, Ann & Hendrickson, Chet, *Extreme programming Installed*, Addison-Wesley, 2001

[8] Liljegren, Gustav, *Vad är XML?*, (2004-11-19), http://www.xml.se/xml.vad.html, 2001

[9] Robillard, Pierre m fl, *Software Engineering Process with the UPEDU*, Addison-Wesley, 2002

[10] Sommerville Ian, 2001: *Software Engineering*, Addison-Wesley

[11] Statskontoret, 2000: *XML-familjen Vad är XML-Schema och XML-DTD?*, (2004-10-20) http://www.statskontoret.se/upload/Publikationer/2000/200032.pdf

[12] The basics of using XML Schema to define elements, http://www-128.ibm.com/developerworks/xml/library/xml-schema/index.html, 2004-11-01

[13] *Översättning engelska - svenska för uttryck inom RUP*, KTH (2004-12-04), http://www.isk.kth.se/kursinfo/6b2027/rup-eng-sve.html

A Kravspecifikation

A.1 Bakgrund

Industrisystem arbetar idag mot en egen framtagen standarddatabas som i de flesta fall körs i en MS SQL-server. Informationen i databasen importeras från affärssystem Det behövs i de flesta system ett verktyg för att administrera grunddata i databasen och målet är att få fram ett program som dynamiskt kan konfigureras för att lösa detta. Programmet skall vara enkelt att använda för slutanvändaren då de kan vara datorovana personer som jobbar i en tung och stressig miljö. Det skall på grund av detta vara uppbyggt så att användaren får begripliga felmeddelanden om denne försöker mata in felaktiga värden i programmet.

Programmet skall dynamiskt kunna konfigureras via inställningsfiler skrivna i XML.

Användarkategorier

Det finns två grupper av användare av systemet. Användare och administratörer. Användarna är de personer som slutligen kommer att använda applikationen och administratörerna är de personer som konfigurerar XML-filerna.

Användarmiljö

Programmet skall användas på windowsplattformen och databasen som körs är Microsoft SQL server.

A.2 Funktionella krav

Nedan följer en lista av funktionalitet som skall erbjudas i applikationen.

Lägg till post:

Användaren får först välja vilken tabell han eller hon vill lägga till en post genom att välja en av ”flikarna” i programmet. När tabellen väl syns i programmet kan användaren mata in nya värden längst ned i tabellen genom att markera den cell han eller hon vill lägga till information i. När all information är inmatad kan användaren spara den nya posten genom att

trycka på sparaknappen. En ny post har nu lagts till i databasen givet att den angivna informationen är korrekt inmatad.

Ta bort post:
För att ta bort en post får användaren först välja vilken tabell som skall redigeras. Användaren får sedan markera en post i tabellen och trycka på knappen ta bort. För att användaren inte skall ta bort poster av misstag måste denne sedan trycka på spara knappen för att ändringen ska genomföras.

Redigera fält:
För att redigera fält måste användaren först välja vilken tabell som ändringarna skall göras i och sedan kan användaren fritt ändra i de fält som finns i tabellen. Vissa fält kan dock vara skrivskyddade. För att spara ändringarna i databasen klickar sedan användaren på spara knappen.

Uppdatera data:
Användaren kan välja att uppdatera den data som visas genom att trycka på knappen ”uppdatera”. Den visade tabellen kommer då att uppdateras med de värden som finns i databasen. Användaren kan även välja att ”filtrera” den data som skall hämtas genom att skriva in egna filter, ange hur många poster som ska visas samt vilken post som skall visas först av de poster som hämtats. Vilka fält som är möjliga att filtrera styrs via tabellens XML-fil.

Kontrollera XML:
Administratören kan kontrollerar att de skrivna XML-filerna är korrekta och att de följer reglerna som är definierade i XML-schemat. Detta ska inte utföras av användaren utan endast av administratörer.

A.2.1 Konfigurering via XML
Konfigurering av systemet skall ske via XML-filer. Detta för att det är enkelt och beskrivande i sig samt väldigt dynamiskt. Man skall om det tillkommer en tabell kunna lägga till administration av denna genom att skapa en ny XML-fil som beskriver administrationen av

den tabellen. Varje tabell som skall administreras ska alltså konfigureras via en egen XML-fil. Filnamnet på XML-filen skall vara namnet på tabellen.

Grundfunktioner som skall finnas är:

- Ny post
- Ta bort post
- Redigera post

A.2.2 Inmatningskontroll av data

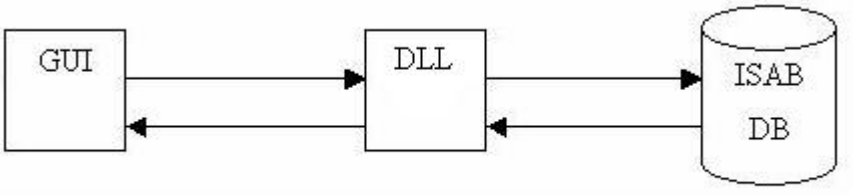
Vid uppdatering eller ny post skall man för varje fält kunna ställa in inmatningskontroll. Följande kontroller/inställningar skall finnas:

- Typ av data
  - Numeriskt
  - Alfanumeriskt
- Tomma fält
  - Tillåtet eller inte
- Nullhantering
  - Om ett fält är tomt byts det ut mot null om tillåtet.
- Längd på fält
- Hantering av decimaltal (komma eller punkt)
  - Konvertera till ”korrekt” typ
- Max/Minvärde
- Stränghantering SQL-server, inmatning citationstecken (’).
  - Tecknen ”görs om” för att kunna läggas in i databasen.

Inmatningkontrollen skall ske varje gång användaren matat in ett nytt värde i tabellen. Detta betyder dock inte att ändringen skall göras i databasen utan bara att en kontroll skall göras. Användaren skall även kunna ändra i flera celler utan att dessa ändringar sparas i databasen. För att spara ändringarna måste användaren trycka på spara knappen.

A.2.3 Arkitektur

Programmet använder en treskiktslösning. Det är uppdelat i: användargränssnitt, logik och databas. Den mesta funktionaliteten skall implementeras i logikdelen.



Figur A.0.1: Treskiktslösning

A.3 Ickefunktionella krav

Förutom de funktionella kraven på programmet finns det även ickefunktionella krav. Dessa krav är inte krav på viss funktionalitet i programmet utan avser prestanda och liknande.

- **Användbarhet**

  Eftersom produkten ska användas i stressiga miljöer måste det vara enkelt att använda och förstå.

- **Prestanda**

  Då programmet ska användas för att mata in och underhålla information i databasen så är det viktigt att det inte tar för lång tid att hämta posterna från databasen. Då de tabeller som visas upp i programmet kan innehålla stora mängder data så fanns det även krav på att användaren ska kunna begränsa hur mycket data som hämtas från databasen.

  Hur lång tid det tar att hämta de poster som begärts beror då förstås på mängden data i varje post samt hur många som begärts. Programmet ska dock klara att hämta 500 poster per sekund.

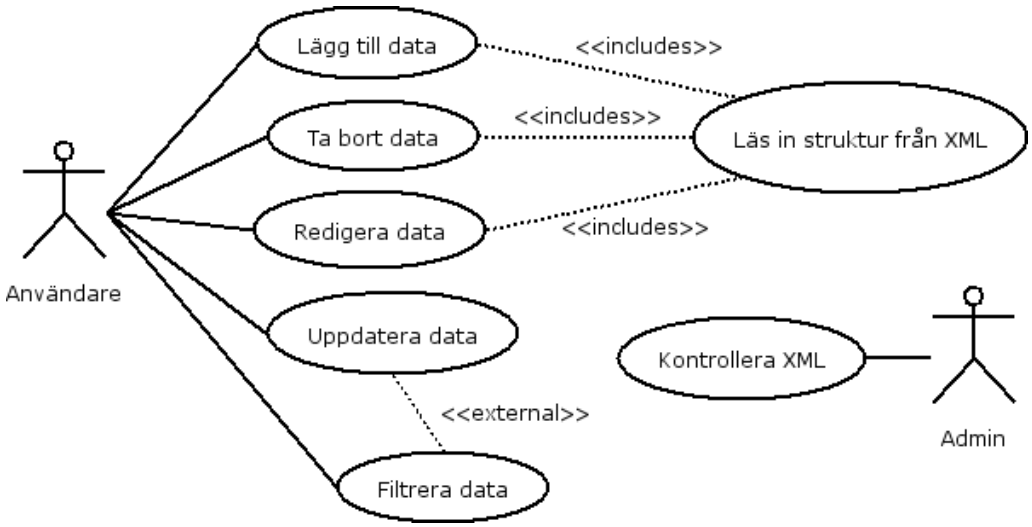
- **Säkerhet**

  Programmet skall logga de ändringar som användaren gör i programmet samt erbjuda en högre grad av loggning om programmet skulle behöva vidareutvecklas. Det som ska loggas är när användaren lägger till, uppdaterar eller tar bort data.

B Projektplan

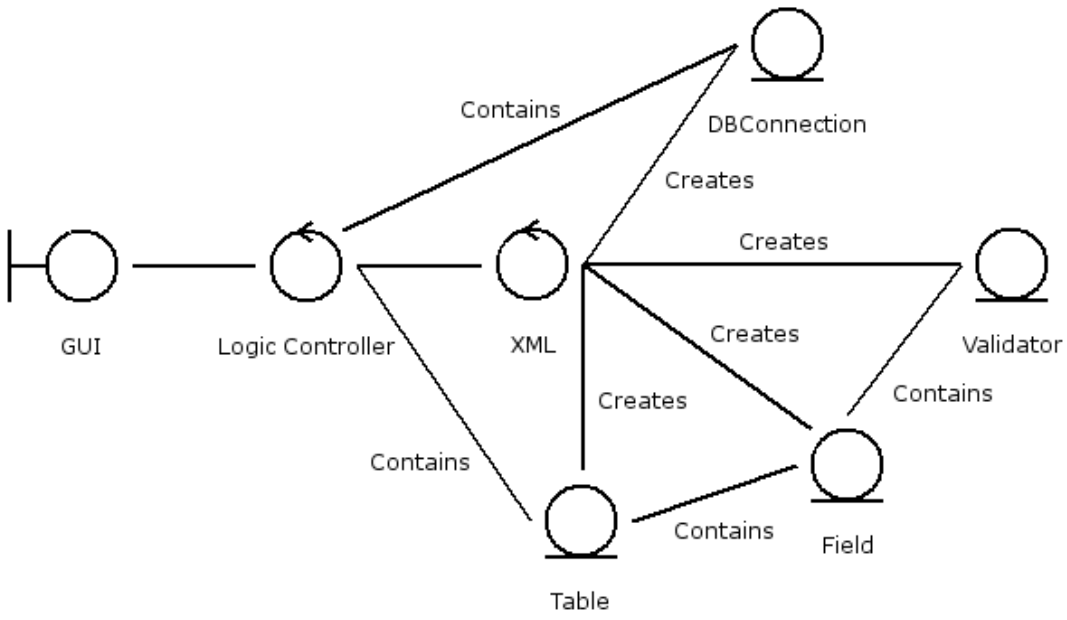
C UML Diagram

C.1 Krav

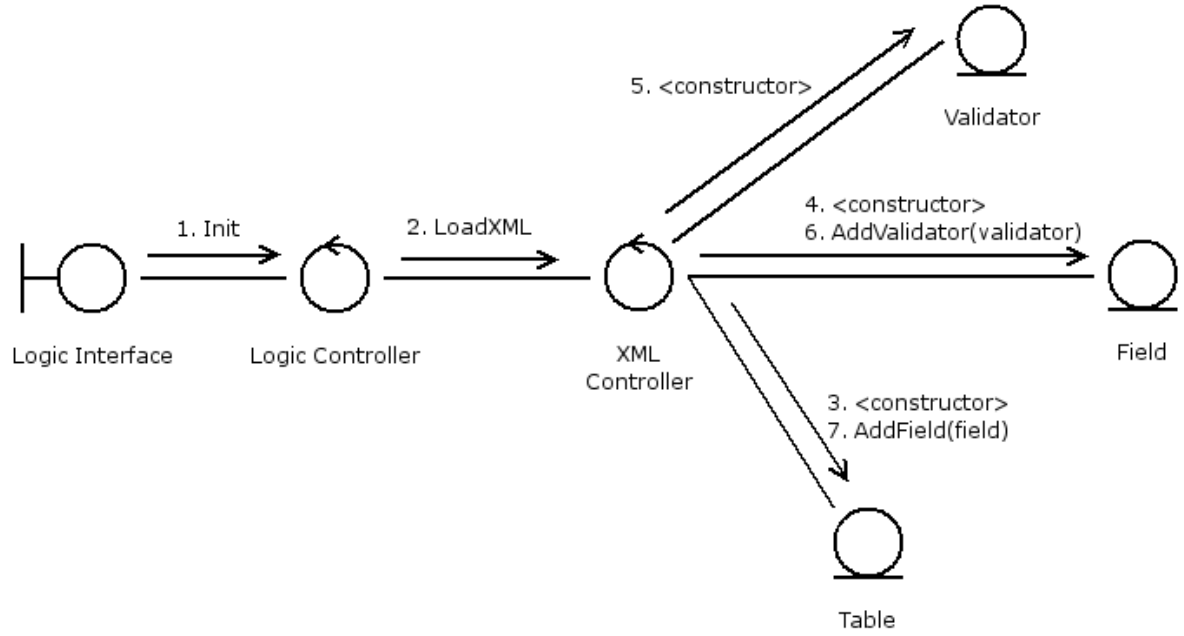


Figur C.0.2: Användingsfallsmodellen

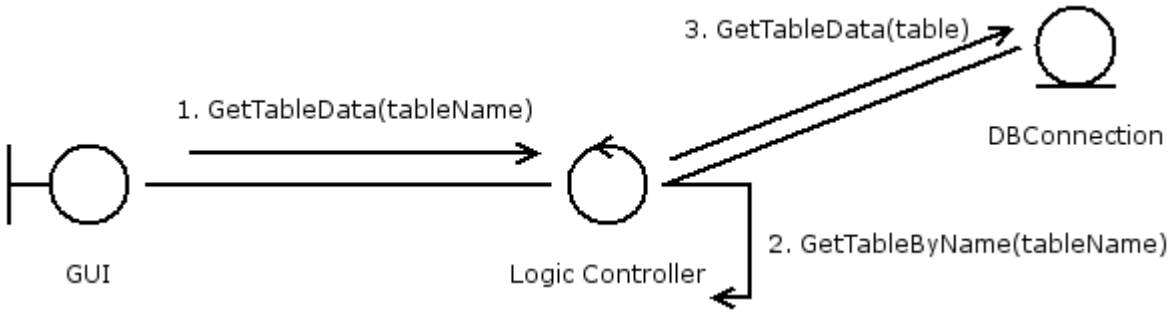
C.2 Analys



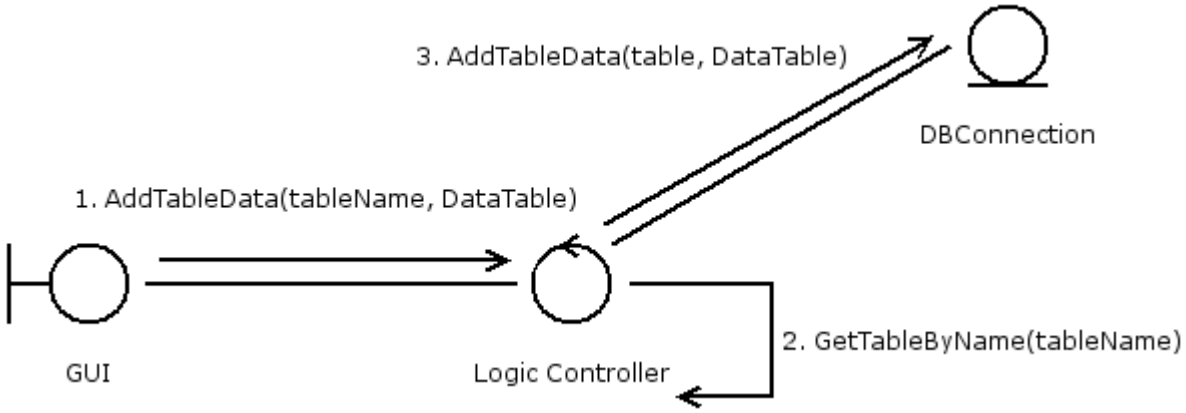
Figur C.0.3: Analys klassdiagram



Figur C.0.4: analys ”Läs in XML” användningsrealisering

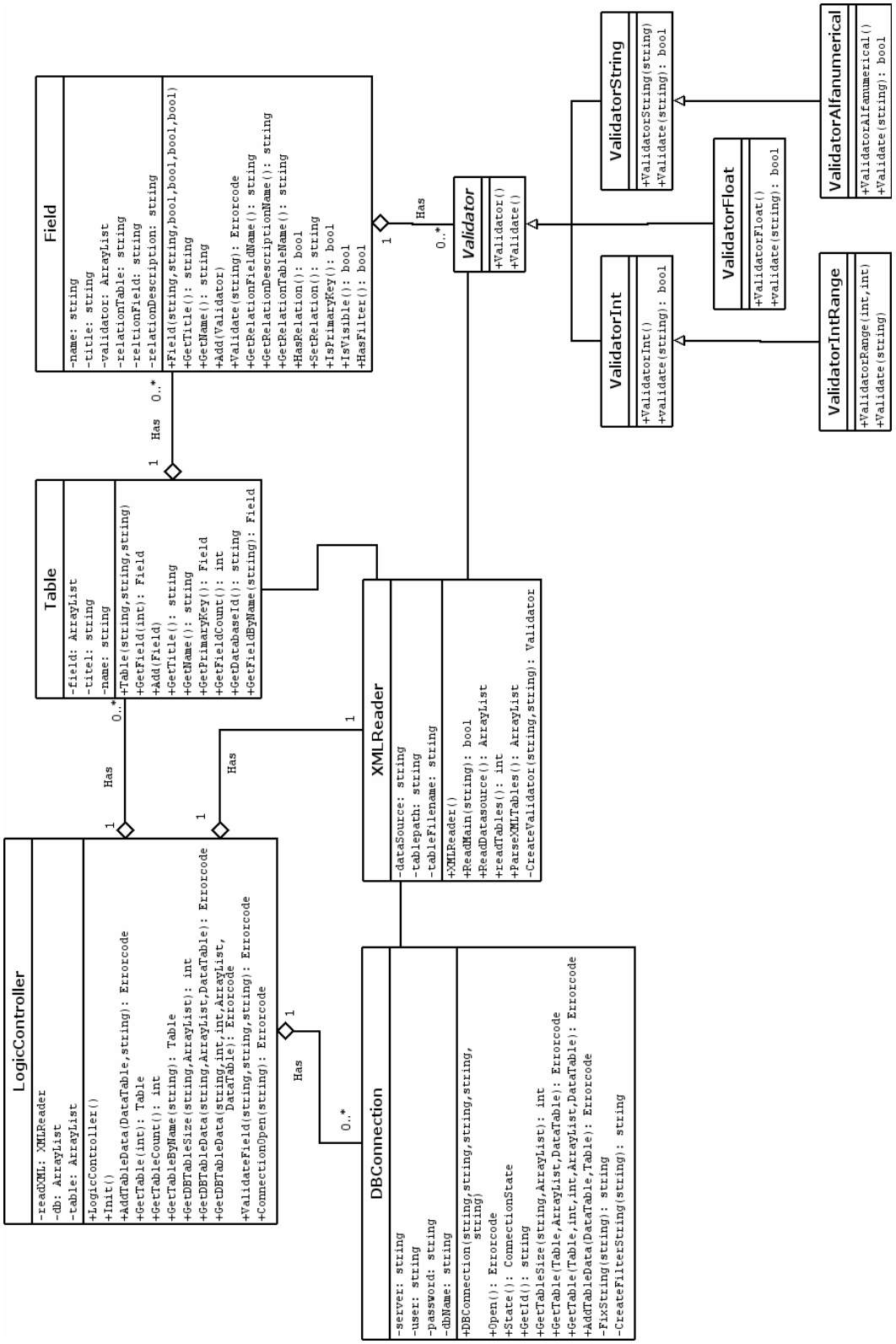


Figur C.0.5: Analys ”Uppdatera data” användningsrealisering

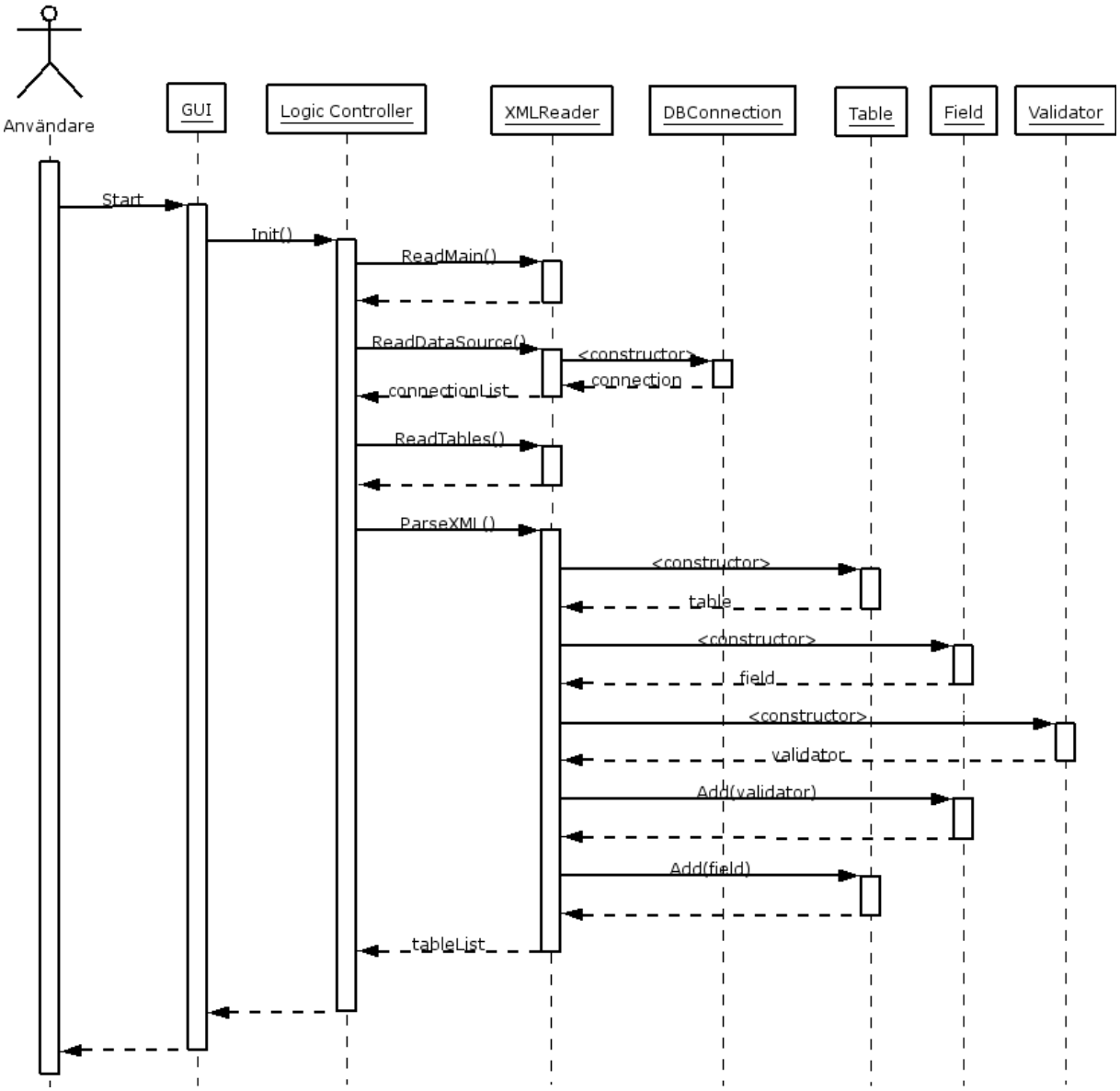


Figur C.0.6: Analys ”Lägg till data” användningsrealisering

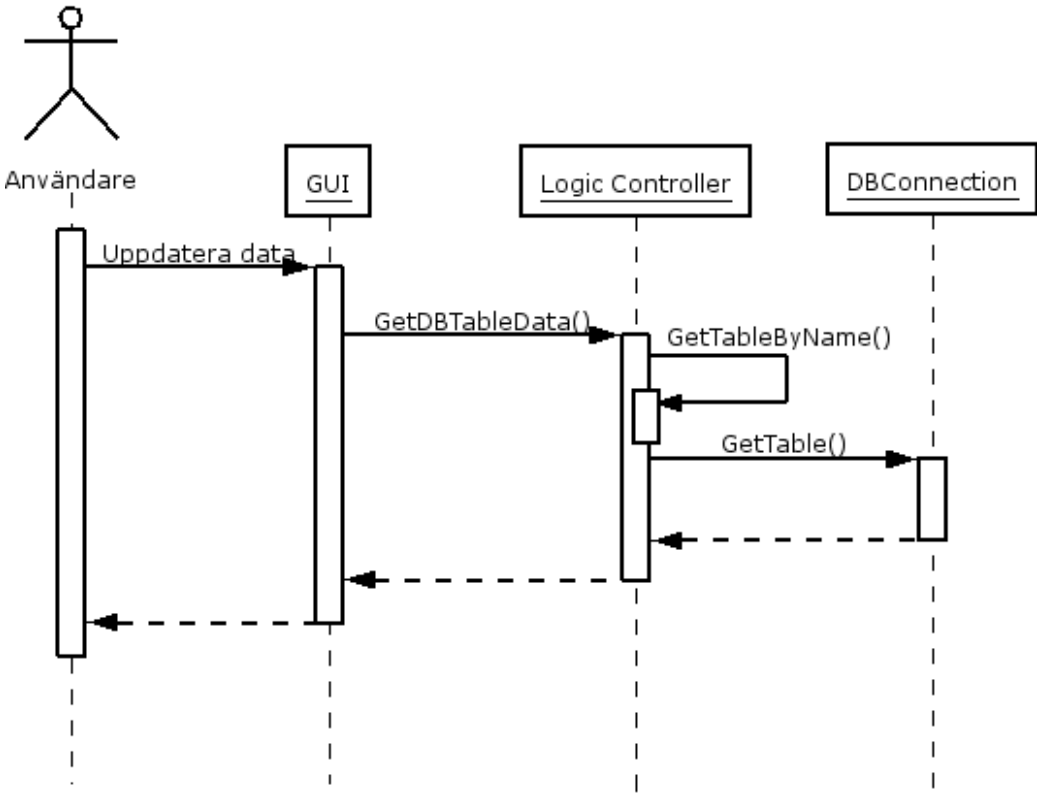
C.3 Design



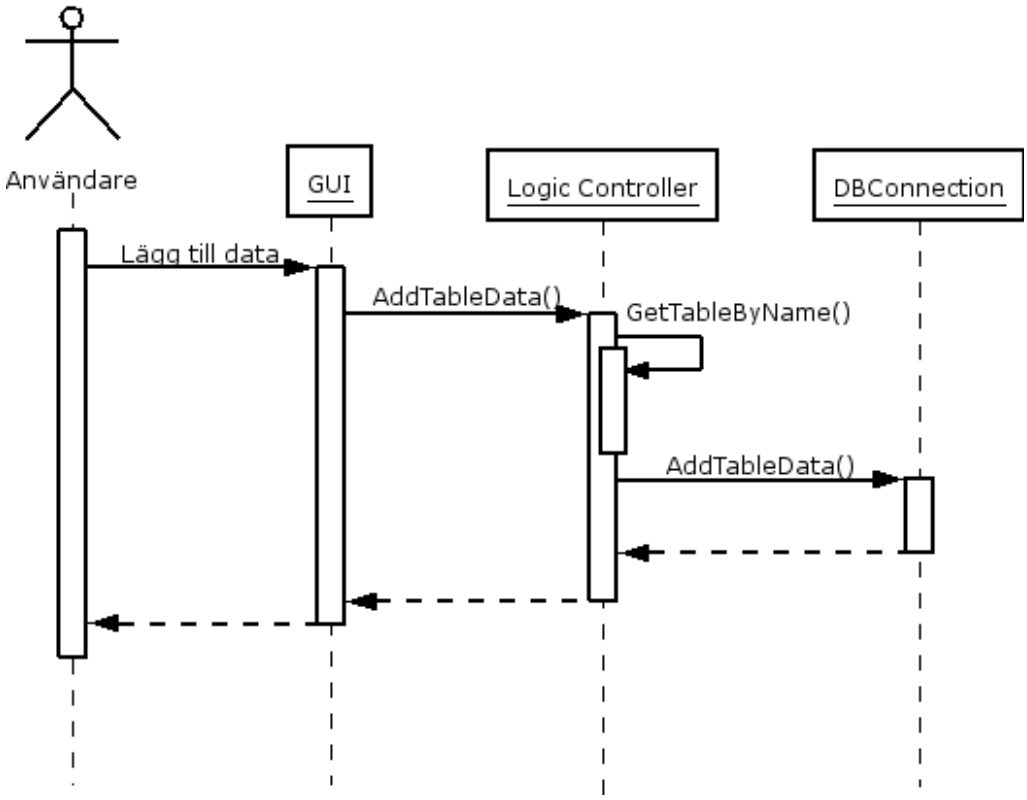
Figur C.0.7: Klassdiagram



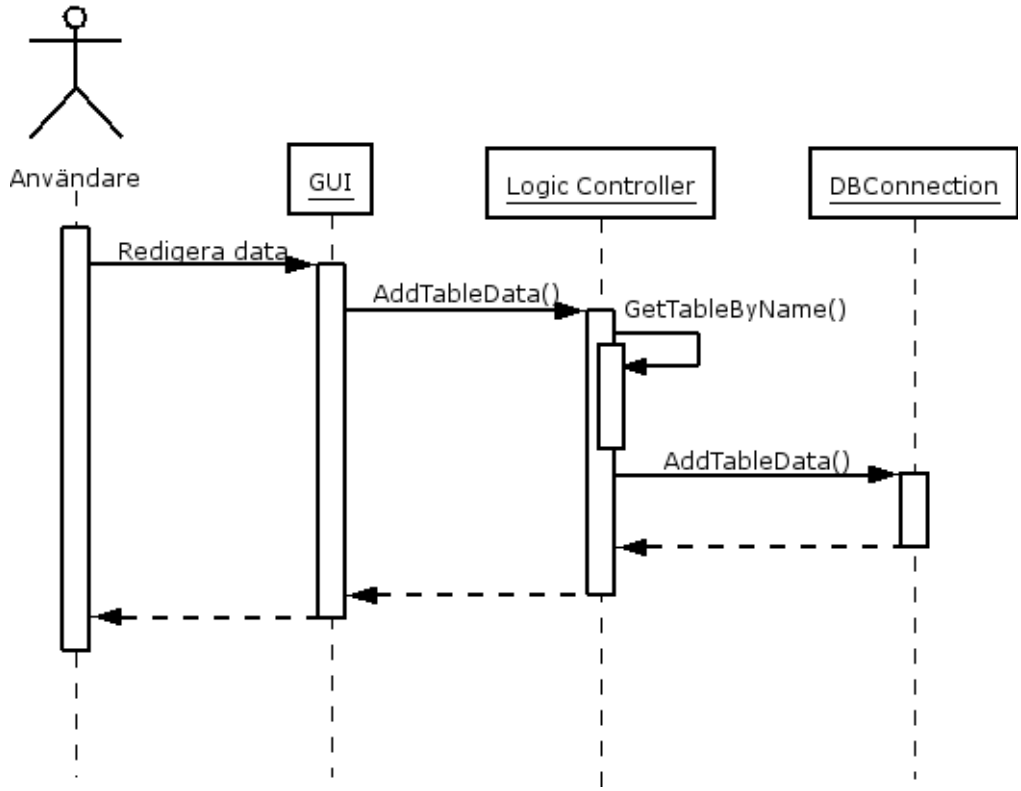
Figur C.0.8: ”Läs in XML-filer” sekvensdiagram



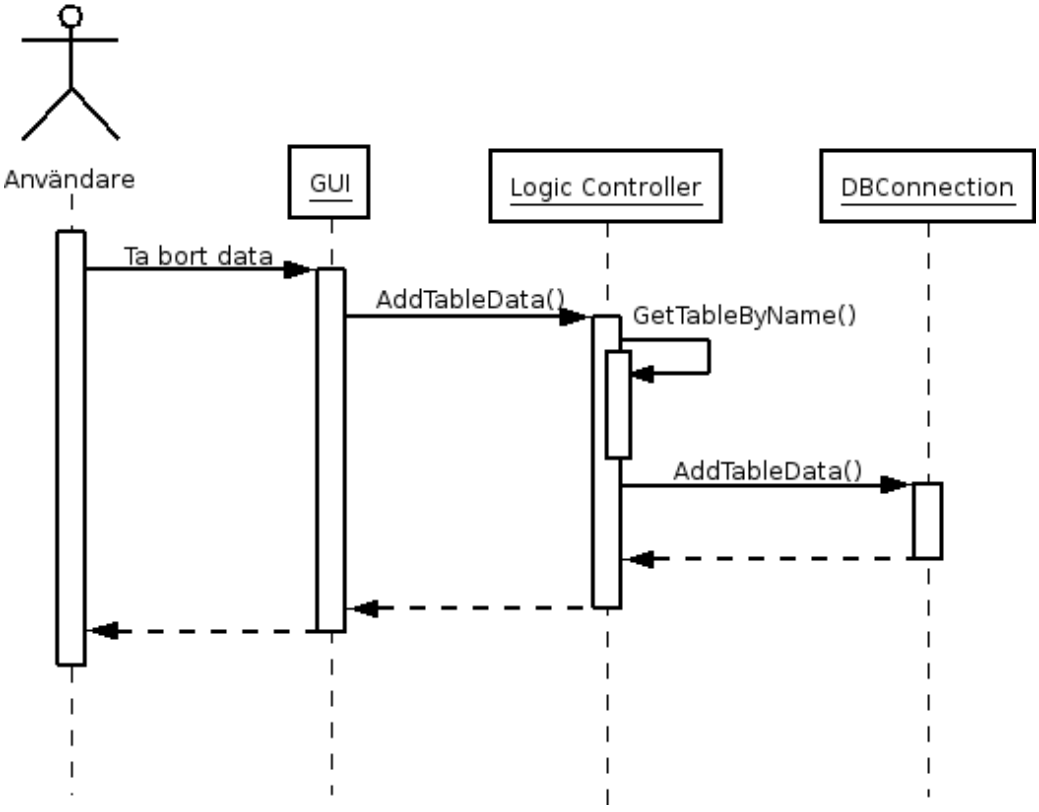
Figur C.0.9: ”Uppdatera data” sekvensdiagram



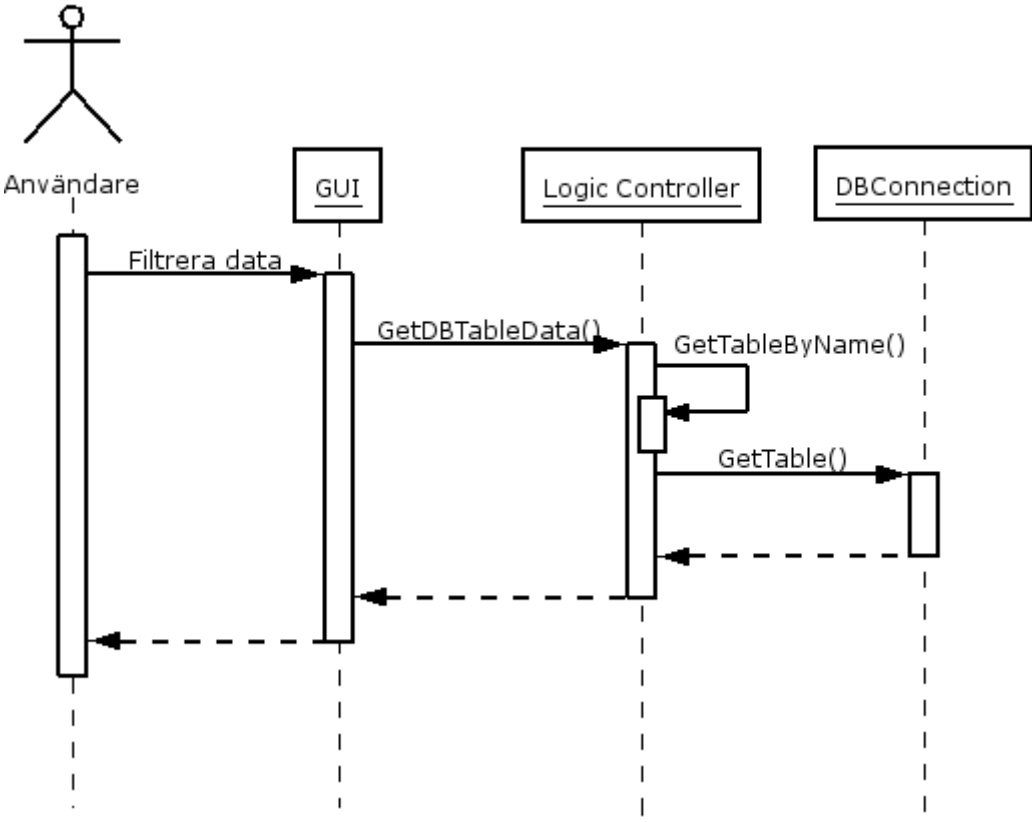
Figur C.0.10: ”Lägg till data” sekvensdiagram



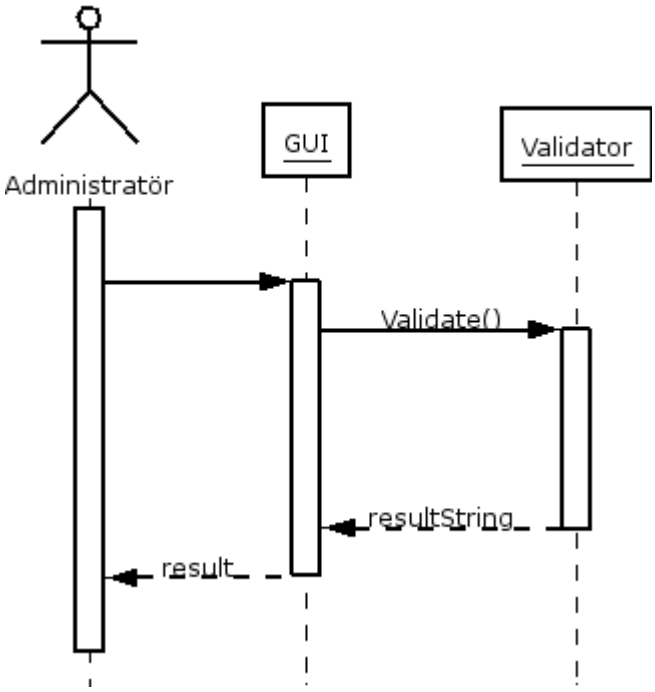
Figur 0.11: ”Redigera data” sekvensdiagram



Figur 0.12: ”Ta bort data” sekvensdiagram



Figur 0.13: ”Filtrera data” sekvensdiagram



Figur 0.14: ”Kontrollera XML” sekvensdiagram

D Användardokumentation

D.1 Installation

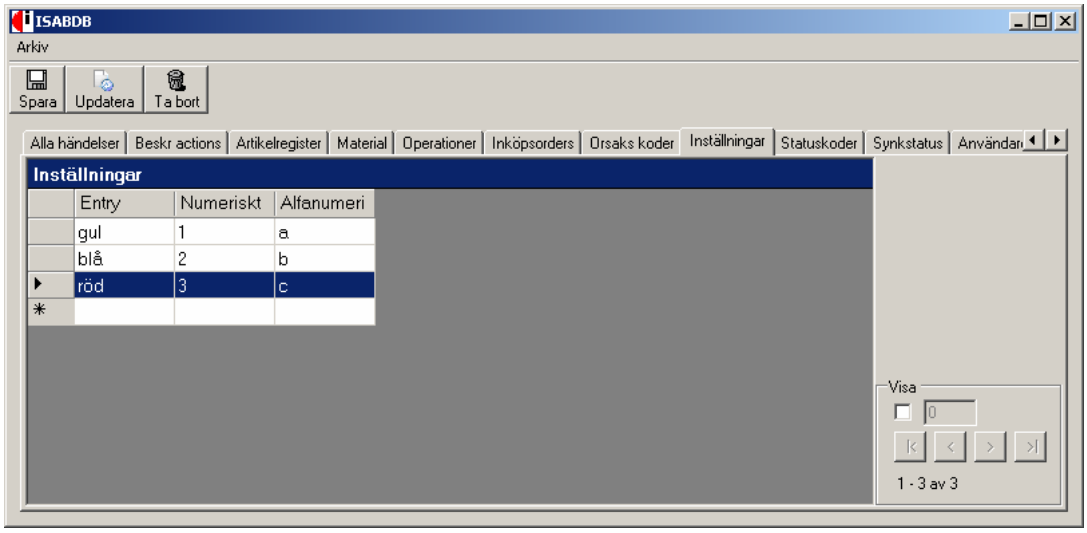
1. Installera .NET Framework 1.1
2. Kopiera katalogen ISABDB till lokal dator
3. Om databasen inte ligger på lokaldator så måste programmet konfigureras. Databas inställningarna ändras i filen data/datasource.xml

D.2 Körning

När programmet startas visas den första tabellen.

D.3 Val av Tabell

För att välja en tabell i programmet klicka på fliken med tabellens namn. Den nya tabellen kommer att laddas och presenteras i programmet.



Figur D.0.15: Val av tabell

Om ändringar gjorts i den tabell som visas kommer programmet fråga om den ska spara ändringarna. Om ändringarna ska sparas svara ja, om ändringarna inte ska sparas nej och om du vill göra flera ändringar i tabellen tryck på avbryt.

D.4 Lägg till data

För att lägga till data i aktuell tabell välj den sista raden i tabellen. Skriv sedan in de värden som skall stå på den nya raden genom att använda ”TAB” tangenten för att byta fält. Vid varje byte kontrolleras om det inmatade värdet är korrekt. Om det skulle vara felaktigt dyker ett felmeddelande upp och du tillåts inte lämna fältet. När raden är färdig kan du välja att lägga till fler rader på samma sätt eller spara dina ändringar genom att trycka på spara knappen i verktygsmenyn högst upp i programmet.

Figur D.0.16: Exempel på felmeddelande.

Om du skulle välja att byta tabell innan du sparat dina ändringar får du en fråga om ändringarna skall sparas innan bytet. Om du väljer att inte spara ändringarna kommer de inte att finnas kvar nästa gång tabellen visas.

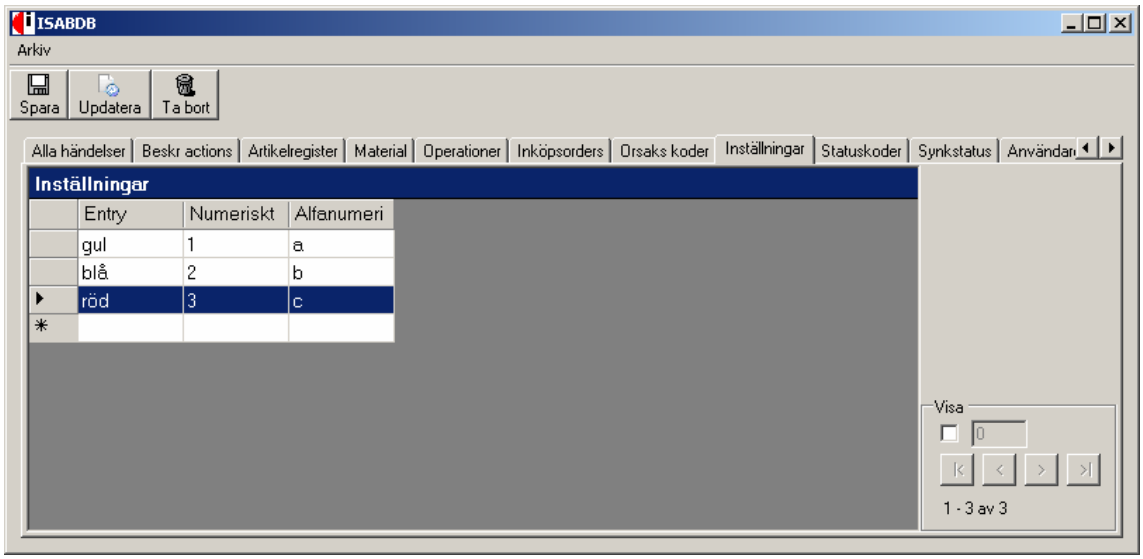
Figur D.0.17: Exempel på förfrågan om att spara.

D.5 Redigera data

För att redigera redan existerande data i tabellen välj den cell som skall ändras och mata in det nya värdet. Detta värde kommer att kontrolleras så att det är korrekt på samma sätt som i 1.2 ”Lägg till data”. När alla ändringar är gjorda kan tabellen sparas genom att trycka på knappen spara i verktygsfältet längst upp i programmet.

D.6 Ta bort rad

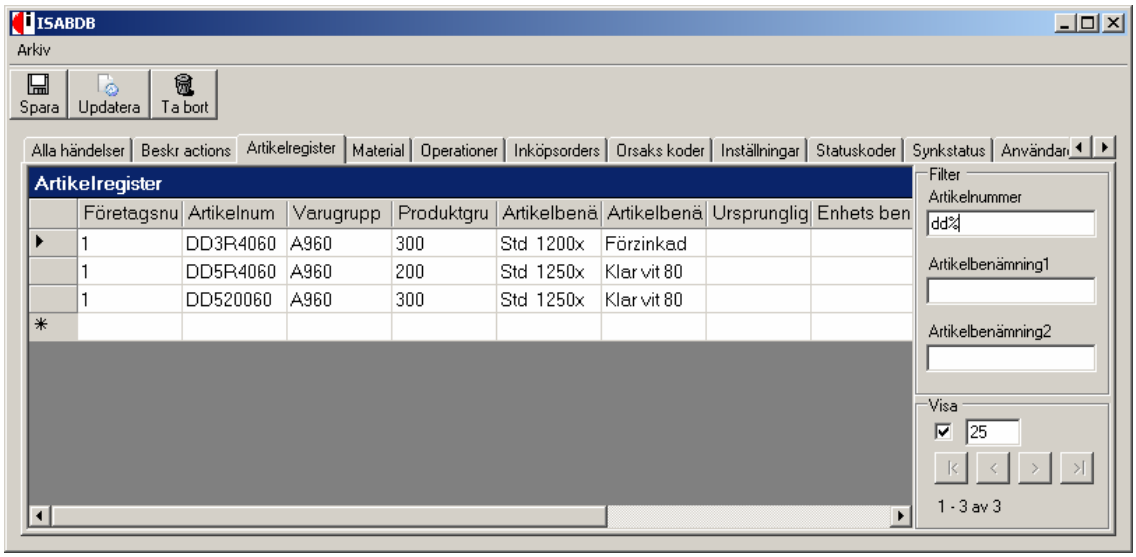
För att ta bort en rad i aktuell tabell markera raden och tryck på ta bort knappen i vertygsmenyn. För att markera en rad klicka på vänsterkanten. Det går att markera flera rader på en gång med hjälp av knapparna ”shift” och ”ctrl”. Trycker man ner ”shift” så markeras flera efterföljande rader. ”ctrl” använd för att markera flera icke efterföljande rader.



Figur D.0.18: Ta bort rad

D.7 Filter

Det finns möjlighet att filtrera den information som hämtas från databasen. Detta görs genom att fylla i textrutorna i ”filterrutan” till höger i programmet.



Figur D.0.19: Filter

Genom att skriva in ett filter kommer endast de poster som matchar filtret att hämtas från databasen, se exempel ovan.

D.7.1 Jokertecken (Wildcards)

Om man vill ange ett filter som endast visar t ex. de poster som i ett visst fält har ”rör”, men även visa de poster som börjar med ”rör” i samma fält, exempelvis ”rörtång” går detta att åstadkomma med så kallade jokertecken.

Ett jokertecken är ett tecken som ersätter ett eller flera andra tecken och på så sätt kan motsvara flera olika filter.

I programmet går det att använda jokertecknen ”%” och ”_”, utan ” tecknen.

Genom att skriva ”rör%” i textboxen kommer programmet även att hämta poster som endast börjar med ”rör”.

% tecknet matchar alltså ett eller flera tecken och det går även att använda mitt i texten. Exempelvis matchar ”r%ng” alla poster som börjar med ”r” och slutar med ”ng” i det fält som filtreras.

Jokertecknet ”_” används på liknande sätt som ”%”, men representerar endast enstaka tecken. Detta är användbart om man t ex. vill filtrera ett fält med personers efternamn eller liknande, men inte vet om personen man söker använder ”é” istället för ”e” i namnet.

Om t ex. filtret ”Fransen” används för att filtrera efternamnen kommer posten med ”Fransén” (notera é) inte att visas, men om filtret ”Frans_n” används istället kommer även den posten att matcha.

D.8 Uppdatera

Det går att uppdatera den information som visas i tabellen genom att trycka på knappen uppdatera. Om ändringar har gjorts i tabellen visas en fråga om de ska sparas.

D.9 Sidor

Programmet kan visa en tabell som en följd av sidor där val av sida sker i ”visaboxen”. Det går att byta sida genom att använda pilknapparna i ”visaboxen”.

Det går även välja att visa ett annat antal poster per sida genom att skriva in ett nytt värde i ”visaboxen” och trycka ”enter” eller uppdatera tabellen. Detta är bra att använda om den aktuella tabellen innehåller många poster. Detta används även med framgång tillsammans med filter (se 2.5 filter).

Notera att ändringar som inte sparats vid byte av sida inte kommer att finnas kvar om du inte väljer att spara dem när tillfrågad. Antalet sidor avgörs av hur många poster du valt att visa per sida.

E Användardokumentation Administratör

E.1 Konfigurering

Konfigurering av systemet sker via XML-filer. Detta för att det är enkelt och beskrivande i sig samt väldigt dynamiskt. Om det tillkommer en tabell går det att lägga till administration av denna genom att skapa en ny XML-fil som beskriver administrationen av den tabellen.

E.2 Filstruktur

Varje tabell som skall administreras skall konfigureras via en egen XML-fil. Filnamnet på XML-filen skall vara namnet på tabellen.

E.2.1 Main

Filen main.xml innehåller sökvägar till de andra XML-filerna programmet behöver. Elementet datasource inehåller sökvägen till filen med information om datakällan. Elementet ”tables” innehåller sökvägen till filerna med reglerna för tabellerna. Se figur 1. Elementet ”loglevel” beskriver vilken nivå av loggning programmet utför. Giltiga nivåer är 1-5 där 1 är den normala nivån. De högre nivåerna ger mer information.

```
<main>
   <datasource>datasource.xml</datasource>
   <tables>C:\Exjobb\xml\tables</tables>
   <loglevel>1</loglevel>
</main>
```

Figur E.0.20: main.xml

E.2.2 Databaskopplingar

Filen datasource.xml innehåller information om datakällorna i programmet. Programmet kan ha många kopplingar mot databaser. Elementet ”source” representerar en koppling. Kopplingarna måste ha unika namn. Elementet ”server” innehåller namnet på databasen, ”user” användarnamnet, ”password” lösenordet och ”database” databasnamnet.

```
<datasource>
   <source name="db1">
      <server>local</server>
      <user>isabdb</user>
      <password></password>
      <database>isabdb_plannja</database>
   </source>
</datasource>
```

Figur E.0.21: datasource.xml

E.2.3 Felmeddelanden

Filen errorcodes.xml innehåller alla felmeddelanden som visas i programmet. Alla felmeddelanden har unika nummer. Detta innebär att det enkelt går att ändra felmeddelanden som visas för användarna. ”Errortype” är olika feltyper. Det finns tre typer av fel 1 valideringsfel, 2 databasfel och 3 XML-fel.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<errorcodes xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
   <errortype number="1">
     <errorcode number="1">Texten är för lång</errorcode>
     <errorcode number="2">Värdet är inte numeriskt</errorcode>
   </errortype>
   <errortype number="2">
     <errorcode number="1">Databasen kan inte nås</errorcode>
     <errorcode number="547">Raden kan ej tas bort eftersom det
finns en koppling mot den.</errorcode>
   </errortype>
</errorcodes>
```

Figur E.0.22: errorcodes.xml

E.2.4 Tabeller

Varje tabell som ska kunna användas i programmet har en egen XML-fil som beskriver tabellen och dess fält. XML-filen innehåller ett element ”table” som beskriver hur tabellen ser ut, vad den heter och vilken databaskoppling som den använder. Detta kan enkelt ändras genom att ändra i attributen ”name”, ”datasource” och ”title”.

Varje tabell innehåller även en mängd fält som definieras i XML-filerna. Dessa fält har också de en mängd attribut som beskriver vad fältet heter och hur det fungerar.

Attributen är:

filter

Anger om fältet kan användas som filter i programmet.
Möjliga värden: true, false

name

Anger namnet på det aktuella fältet (samma som i databasen).

nillable

Anger om fältet tillåts ha värdet <null> i programmet.
Möjliga värden: true, false

primarykey

Anger om fältet är primärnyckel I tabellen.
Möjliga värden: true, false
Vanligtvis: false

readonly

Anger om fältet är skrivskyddat.
Möjliga värden: true, false
Vanligtvis: false

rel

Anger att fältet är har en koppling mot ett annat fält.

Möjliga värden: Måste följa ett visst format. ”tabell.fält1.fält2” där tabell är namnet på den tabell mot vilken fältet mappar, fält1 den främmandenyckel som används och fält2 den beskrivning som till sist ska visas.

title

Anger titeln på det fält som visas i programmet.

visible

Anger om fältet ska visas eller inte.

Möjliga värden: true, false

För att kontrollera om användarens inmatade värden är korrekta används ett element kallat ”Validator” i fälten. Vad en validator kontrollerar avgörs av dess ”name” attribut. En validator med ”int” i ”name” attributet kontrollerar t ex. så att det värde användaren matat in i fältet är ett numeriskt värde.

De validators som finns är: string, int, float, alfa

validator string:

Kontrollerar om det inmatade värdet är en sträng. Innehållet i denna validator anger den maximalt tillåtna längden på strängen.

validator int:

Kontrollerar om det inmatade värdet är ett numeriskt värde. Om innehållet i detta element inte är tomt förväntas det innehålla ett intervall i formatet ”0,100”. (0 till och med 100). Intervallet klarar även av negativa värden som t ex. ”-200,-10”.

validator float:

Kontrollerar om det inmatade värdet är ett decimaltal som t ex. ”10,4”. Även tal som ”.9” och ”6.” anses giltiga av denna.

validator alfa:

Kontrollerar att om det inmatade värdet är alfanumeriskt dvs. endast innehåller a-z, A-Z.

Dessa ”validators” kan kombineras för att ytterligare begränsa vad användaren får mata in. En string validator och en int validator kan t ex. kombineras för att begränsa fältet till ett heltal som max kan vara x tecken långt.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
  <table name="TblStatusCodes" datasource="db1" title="Statuskoder">
     <field name="StatusCodeID" visible="false" nillable="false"
     primarykey="true">
        <Validator name="int"></Validator>
     </field>
     <field name="Description" title="Beskrivning" visible="true"
     nillable="false">
        <Validator name="string">50</Validator>
     </field>
  </table>
```

Figur E.0.23: table.xml

E.3 Kontroll av XML-filerna

För att kontrollera att XML-filerna har en korrekt struktur samt innehåll kan man använda programmet Xml Schema Validator. Det använder XML-scheman för att kontrollera filerna.

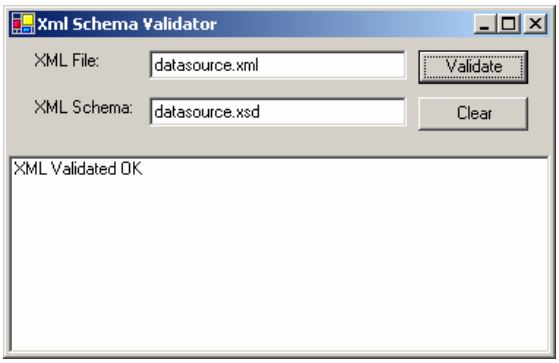
Man får ange XML-fil och schema.

Scheman:

datasource.xsd	kontrollerar datasource.xml
main.xsd	kontrollerar main.xml
errorcodes.xsd	kontrollerar errorcodes.xml

table.xsd kontrollerar tabellfilerna

Är filen korrekt så anges ”XML Validated OK” och är den felaktig så anges ”XML Validated fail”. Är filen felaktig ger programmet information vad som är fel och på vilken rad felet finns.



Figur E.0.24: Kontroll av XML-filer