Tillämpning av Unified Process och Design Patterns vid integrering av system

Andreas Jönsson

Examensarbete för 20 p, Institutionen för datavetenskap,
Naturvetenskapliga fakulteten, Lunds universitet

Thesis for a diploma in computer science, 20 credit points,
Department of Computer Science, Faculty of Science, Lund University

Tillämpning av Unified Process och Design Patterns vid integrering av system

Sammanfattning

Datakonsultföretaget Create in Lund AB vill underlätta sina rutiner för fakturering, då det i nuläget krävs att administrationen tar fram och jämför information från två skilda system.

Syftet med examensarbetet är att utveckla ett faktureringssystem som förenklar faktureringsprocessen för administrationen på Create. Under utvecklandet av systemet har jag valt att följa programvaruutvecklingsprocessen Unified Process, samt försökt att finna designmönster som kan användas vid integreringen.

Resultatet av mitt arbete blev en fristående applikation utvecklad i programmeringsspråket Java, som kommunicerar med de båda andra systemens databaser och presenterar relevant information för användaren som ett antal fakturor. Av de 23 designmönster jag studerade, hade jag användning av fyra av dem i faktureringssystemet. Jag kunde dock inte hitta någon direkt koppling mellan något designmönster och integrering av system. Jag konstaterade också att Unified Process är lämplig att använda vid utveckling av stora eller mycket stora programvarusystem.

System integration using Unified Process and Design Patterns

Abstract

The computer consultant company Create in Lund AB wishes to facilitate their invoice routines, as it at present is required that the administration finds and compares information from two different systems.

The purpose of this thesis is to develop an invoice management system that simplifies the invoice management process for the administration at Create. During the development of the system, I have chosen to follow the software development process Unified Process and also tried to find design patterns that can be applied to the integration.

The result of my project is a self-contained application developed using the Java programming language that communicates with the databases of the other two systems and presents relevant information to the user as a set of invoices. Of the 23 design patterns I studied, I used four of them in the invoice management system. However I couldn't find any connection between any design pattern and the integration of systems. I also stated that Unified Process is suitable when developing large or very large software systems.

Innehållsförteckning

1 INLEDNING 5

1.1 BAKGRUND 5
1.2 PROBLEM 5
1.3 SYFTE OCH MÅLGRUPP 6
1.4 AVGRÄNSNING 6

2 PROGRAMVARUUTVECKLINGSPROCESSER 7

2.1 VATTENFALLSMODELLEN 7
2.2 EXTREME PROGRAMMING 8
2.3 UNIFIED PROCESS 9

3 STUDIE AV UNIFIED PROCESS 11

3.1 KRAVHANTERING 11
3.2 ANALYS 14
3.3 DESIGN 17
3.4 IMPLEMENTATION 21
3.5 TEST 25

4 STUDIE AV DESIGN PATTERNS 28

4.1 DESIGN PATTERNS 28
4.2 COMMAND PATTERN 28
4.3 MEDIATOR PATTERN 29
4.4 FAÇADE PATTERN 29
4.5 ITERATOR PATTERN 29

5 TILLÄMPNING AV UNIFIED PROCESS OCH DESIGN PATTERNS VID INTEGRERING AV SYSTEM 30

5.1 KRAVHANTERING 30
5.2 ANALYS 31
5.3 DESIGN 33
5.4 IMPLEMENTATION 38
5.5 TEST 39

6 RESULTAT 42

6.1 SÖK FAKTUROR 42
6.2 VISA FAKTURA 43
6.3 INSTÄLLNINGAR 44

7 DISKUSSION 45

7.1 INTEGRERING AV SYSTEM 45
7.2 SLUTSATSER 45
7.3 FORTSATT UTVECKLING 46

REFERENSER 47

ÖVRIG LITTERATUR 47

BILAGA A: TESTFALL 49

Förord

Först och främst vill jag tacka min handledare Ferenc Belik för hans värdefulla hjälp och vägledning under arbetets gång. Jag vill också ge ett stort tack till Petros Tsakiris på Create in Lund AB för att han gav mig möjligheten att genomföra det här examensarbetet. Slutligen vill jag tacka Toni Salov på Create in Lund AB som ställt upp och svarat på frågor kring databaserna.

Lund, mars 2006
Andreas Jönsson

1 Inledning

I detta kapitel beskriver jag bakgrunden till problemet, problemformuleringen, syftet med arbetet och dess målgrupp, samt de avgränsningar jag har gjort.

1.1 Bakgrund

Create in Lund AB är ett datakonsultföretag som verkar inom systemutveckling och kvalitetssäkring/test. Create in Lund AB ingår i koncernen Create Group AB och är ett renodlat tjänsteföretag.

Create har ett webbaserat tidrapporteringssystem som är implementerat i ASP, Visual Basic och SQL Server. I systemet rapporteras bland annat arbetad tid i olika kundprojekt, semester och sjukfrånvaro. Create har också ett CRM-system (Customer Relationship Management) som är implementerat i Java och MySQL. I den lösningen administrerar Create information om sina kunder, försäljning, samt samtlig kommersiell information om sina uppdrag.

Create fakturerar månadsvis i efterskott. Processen är väldigt omständlig då det krävs av administrationen att dels ta fram alla tidrapporter från tidrapporteringssystemet och dels att jämföra tidrapporterna med den uppdragsinformation som finns i CRM-systemet. Istället önskar Create att relevant information för fakturering samlas från de båda systemen och presenteras under ett och samma gränssnitt. Det finns idag nyckelfält i databaserna som gör att information från respektive system kan kopplas samman.

1.2 Problem

Efter diskussion med min handledare och vice VD för Create in Lund AB, kom vi fram till att jag skulle utveckla en ny, fristående applikation. Applikationen ska hämta relevant information från tidrapporterings- och CRM-systemet och presentera den för användaren som ett antal fakturor. Administration slipper då den omständliga rutinen och sparar därmed både tid och möda.

För att höja abstraktionsnivån på arbetet har jag även valt att studera olika programvaru-utvecklingsprocesser och designmönster. Framförallt har jag valt att fördjupa mig i och följa Unified Process under utvecklandet av applikationen. Problemformuleringen kan därmed sammanfattas i följande frågeställning:

- Hur kan Unified Process och design patterns användas vid integrering av system?

Anledningen till att jag i första hand valde att fördjupa mig i Unified Process beror på flera faktorer:

- UP har stark anknytning till det standardiserade notationsspråket UML.
- UP är användningsfallsdrivet.
- UP är arkitekturcentrerat.
- UP är iterativt och inkrementellt.
- UP ligger i tiden, eftersom det är anpassat för att utveckla dagens omfattande och komplexa programvarusystem.
- Eget intresse.

1.3 Syfte och målgrupp

Syftet med det här examensarbetet är att utveckla ett faktureringssystem som ska underlätta faktureringsprocessen för administrationen på Create in Lund AB. Under tiden undersöker jag hur utvecklingsprocessen Unified Process förhåller sig till integrering av system. Jag försöker också finna designmönster som kan användas vid integreringen.

Målgruppen bör kunna innefatta programvaruutvecklare, studenter och andra individer som har intresse av eller vill veta mer om integrering av system, programvaruutvecklingsprocesser och/eller designmönster. Särskilt borde utvecklare av programvara med anknytning till integrering av system kunna få nytta av de resultat och slutsatser jag eventuellt kommer att komma fram till.

1.4 Avgränsning

På grund av Unified Process omfattning har jag inte haft möjlighet att ta med varenda aspekt av processen i uppsatsen. Jag har dock försökt att få med de mest väsentliga koncepten. Då Unified Process är ett generellt processramverk som kan anpassas för olika tillämpnings-områden och projektstorlekar, har jag valt att endast använda de artefakter som jag ansett vara nödvändiga för mitt specifika tillämpningsområde. Jag har inte heller lagt någon särskild fokus på applikationens användargränssnitt. Ovanstående begränsningar har framförallt berott på tidsbrist.

2 Programvaruutvecklingsprocesser

I det här kapitlet presenterar jag tre olika programvaruutvecklingsprocesser. Jag har valt att använda skalan tungviktsmodeller – lättviktsmodeller, där vattenfallsmodellen är ett exempel på en tungviktsmodell, Extreme Programming är ett exempel på en lättviktsmodell och Unified Process är ett mellanting.

> ”A software development process is the set of activities needed to transform a user’s requirements into a software system.” (Booch, Jacobson och Rumbaugh 1999, s. 4)

2.1 Vattenfallsmodellen

Vattenfallsmodellen är den första publicerade modellen av programvaruutvecklingsprocessen och har sitt ursprung i mer generella systemingenjörsprocesser [1]. Modellen delas in i följande grundläggande utvecklingsaktiviteter:

1. Kravdefinition – Systemets tjänster, restriktioner och mål upprättas i samråd med systemanvändare. De definieras sedan i detalj och fungerar som en system-specifikation.

2. System- och programvarudesign – Systemdesignprocessen delar in kraven i antingen hårdvaru- eller programvarusystem. Den etablerar en samlad systemarkitektur. Programvarudesign innefattar att identifiera och beskriva de grundläggande programvarusystemabstraktionerna och deras relationer.

3. Implementation och enhetstestning – Programvarudesignen realiseras som en uppsättning av program eller programenheter. Enhetstestning innebär att man verifierar att varje enhet uppfyller sin specifikation.

4. Integration och systemtestning – De enskilda programenheterna eller programmen integreras och testas som ett fullständigt system för att se till att programvarukraven har uppnåtts. Efter testning överlämnas programvarusystemet till kunden.

5. Drift och underhåll – Systemet installeras och tas i bruk. Underhåll innefattar att korrigera fel som inte upptäcktes i tidigare stadier av livscykeln, förbättra implementationen av systemenheter och utöka systemets tjänster när nya krav hittas.

Resultatet av varje fas är en eller flera godkända dokument. Den påföljande fasen ska inte starta förrän den tidigare fasen har avslutats. I praktiken överlappar dessa steg varandra och förser varandra med information.

Fördelarna med vattenfallsmodellen är att dokumentation produceras vid varje fas och att den passar med andra ingenjörsprocessmodeller [1]. Dess stora problem är dess oflexibla indelning av projektet i skilda faser. Beslut måste tas i ett tidigt skede i processen, vilket gör det svårt att agera vid förändringar av kundkraven. Vattenfallsmodellen ska därför endast användas när kraven är välförstådda och osannolikt kommer att förändras radikalt under systemutvecklingen [1].

2.2 Extreme programming

Inom extreme programming uttrycks alla krav som scenarion, som direkt implementeras som en följd av arbetsuppgifter. Programmerare arbetar i par och utvecklar test för varje arbetsuppgift innan koden skrivs. Alla test måste exekveras med lyckat resultat när ny kod integreras i systemet. Det är korta tidsluckor mellan olika utgåvor av systemet [1].

I tabell 1 presenteras de principer som tillämpas vid användning av utvecklingsprocessen extreme programming. Namnen på ett par principer anges endast på engelska, då det var svårt att hitta lämpliga översättningar.

Tabell 1 Principer som tillämpas inom extreme programming

Princip	**Beskrivning**
Inkrementell planering (Incremental planning)	Krav registreras på storykort, eller handlingskorts, och de handlingar som ska inkluderas i en utgåva bestäms av deras inbördes prioritet och tillgänglig tid.
Små utgåvor (Small releases)	Den funktionalitet som är minst och som är av värde för verksamheten utvecklas först. Utgåvor av systemet förekommer ofta och ökar funktionaliteten inkrementellt.
Enkel design (Simple design)	Endast tillräcklig design utförs för att uppfylla de aktuella kraven, inte mer.
Test-first development	Ett ramverk för automatiserade enhetstest används för att skriva test för ny funktionalitet innan själva funktionaliteten implementeras.
Refactoring	Alla utvecklare förväntas att förbättra koden fortlöpande så fort eventuella förbättringar av koden upptäcks. Detta håller koden enkel och i gott skick.
Parprogrammering (Pair programming)	Utvecklare arbetar i par. De kontrollerar varandras arbete och ger varandra stöd.
Kollektivt ägandeskap (Collective ownership)	Paren av utvecklare arbetar inom alla områden av systemet och alla utvecklare äger all kod. Vem som helst kan ändra vad som helst.
Kontinuerlig integration (Continuous integration)	Så fort en arbetsuppgift är färdig integreras den i systemet. Efter en integrering måste alla enhetstesterna bli godkända.
Konsekvent hastighet (Sustainable pace)	Stora mängder av övertid accepteras inte, då det ofta innebär att kvaliteten på koden och produktiviteten reduceras.
Kunden på plats (On-site customer)	En representant av systemets slutanvändare ska finnas tillgänglig på heltid för användning utav utvecklingsteamet. Inom extreme programming är kunden en medlem av utvecklingsteamet och ansvarar för att komma med systemkrav som ska implementeras.

2.3 Unified Process

Unified Process (UP) är inte bara en programvaruutvecklingsprocess utan ett generellt processramverk som kan specialiseras för olika tillämpningsområden, olika typer av organisationer, olika kompetensnivåer och olika projektstorlekar. UP är komponentbaserat, vilket innebär att programvarusystemen byggs upp av komponenter.

UP använder sig av UML för att visualisera programvaran i ritningar och diagram. Faktum är att UML är en integrerande del av Unified Process, de utvecklades hand i hand [2].

De mest karakteristiska aspekterna av UP är, enligt författarna, att processen är:

- Användningsfallsdriven
- Arkitekturcentrerad
- Iterativ och inkrementell

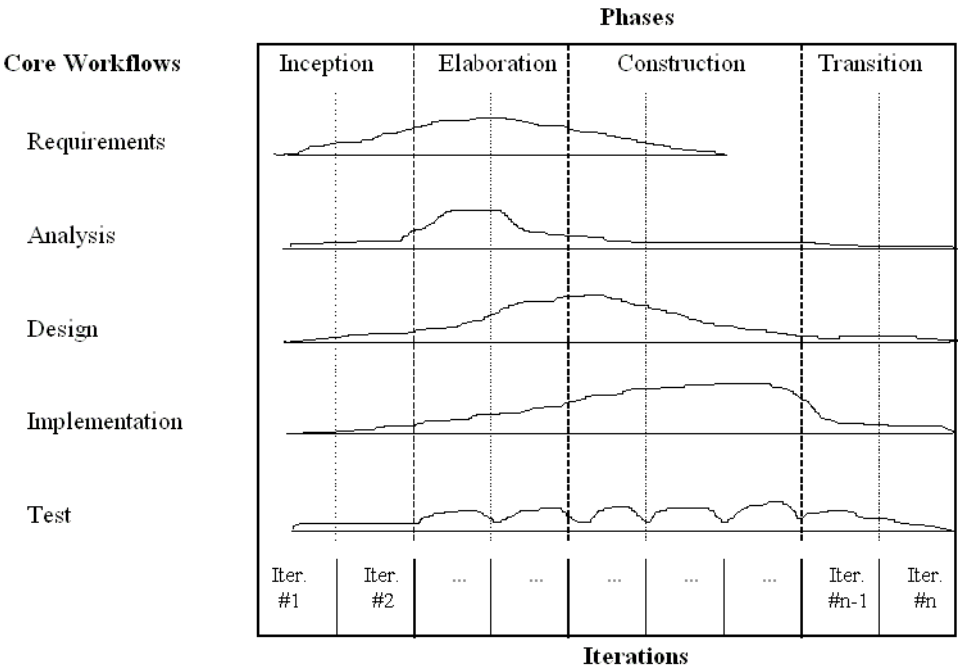
Ett användningsfall är ett stycke funktionalitet i systemet som ger en användare ett resultat av värde [2]. Användningsfall fastställer funktionella krav. Alla användningsfall tillsammans bildar användningsfallsmodellen, som beskriver hela systemets funktionalitet. Användningsfallen utgör också grunden för systemets design, implementation och test. De driver alltså utvecklingsprocessen. Användningsfallsdriven betyder att utvecklingsprocessen följer ett flöde, processen fortskrider genom en rad arbetsflöden som har sitt ursprung i användningsfallen [2].

En arkitektur är en vy över hela systemets design med den viktigaste karakteristiken mer synliggjord genom att utelämna detaljer [2]. Processen hjälper arkitekten att fokusera på de rätta målen, som förståelse, anpassningsbarhet och återanvändbarhet.

Vid utveckling av stora programvarusystem är det praktiskt att dela in arbetet i mindre delar eller miniprojekt. Varje miniprojekt utgör en iteration som leder till en tillväxt i systemet [2]. Iterationerna ska vara kontrollerade, dvs. de ska väljas ut och genomföras på ett planerat sätt.

Dessa tre koncept, användningsfallsdriven, arkitekturcentrerad och iterativ och inkrementell, utgör tillsammans grunden i Unified Process. Arkitekturen tillhandahåller strukturen som leder arbetet i iterationerna, medan användningsfallen definierar målen och driver arbetet i varje iteration [2].

Unified Process upprepas över en serie av cykler under systemets livstid, där varje cykel avslutas med en produktutgåva. Varje cykel består av fyra faser: förberedelse, etablering, konstruktion och överlämning. Varje fas indelas ytterligare i iterationer och inom varje iteration utförs arbete i olika utsträckning i någon av UP:s fem arbetsflöden: krav, analys, design, implementation och test; se figur 1. Varje fas avslutas i en milstolpe, vilket innebär att vissa modeller eller dokument ska finnas tillgängliga.



Figur 1 UP:s faser och arbetsflöden. Kurvorna uppskattar i vilken utsträckning arbete sker i de olika arbetsflödena inom varje fas (och iteration).

3 Studie av Unified Process

Som jag tidigare nämnt delas Unified Process in i faser och iterationer över tiden. Trots detta har jag valt att dela upp det här kapitlet i UP:s fem arbetsflöden: krav, analys, design, implementation och test. En av anledningarna till detta är att jag sammanhängande vill kunna presentera de artefakter som ingår eller kan ingå i respektive modell. Jag tycker dessutom att man får en mer överskådlig bild över UP:s aktiviteter än om jag skulle delat upp kapitlet i faser och iterationer.

I Unified Process används ordet *artefakt* som en generell term för all slags information som skapas, produceras, ändras eller används av arbetare i utvecklandet av ett system. En artefakt kan vara en modell, ett modellelement eller ett dokument [2].

3.1 Kravhantering

Det primära syftet med ett arbetsflöde för kravhantering är att styra utvecklingen mot det rätta systemet. Detta uppnås genom att beskriva systemkraven tillräckligt noga, så att kunden och systemutvecklarna kan nå en överenskommelse om vad systemet ska och inte ska göra.

Användningsfall erbjuder ett systematiskt och intuitivt sätt att fastställa de funktionella kraven [2]. Genom att använda användningsfall tvingas man tänka i termer av användare och vilka verksamhetsbehov som kan uppfyllas genom dem [2]. Eftersom de flesta icke-funktionella krav är specifika för ett enda användningsfall, anges även de i samband med användningsfallet. De återstående icke-funktionella kraven, de som är gemensamma för många eller alla användningsfall, samlas i ett separat dokument och kallas för kompletterande krav.

Användningsfallsmodellen kan sammanfattas med följande punkter:

- Beskrivs med kundens språk.
- Är en yttre vy av systemet.
- Struktureras med användningsfall; ger struktur åt den yttre vyn.
- Används i första hand som ett kontrakt mellan kunden och utvecklarna om vad systemet ska och inte ska göra.
- Kan innehålla redundans och inkonsekvenser bland kraven.
- Fastställer systemets funktionalitet.
- Definierar användningsfall.

3.1.1 Artefakter

Användningsfallsmodell

Användningsfallsmodellen fungerar som ett avtal mellan kunden och utvecklarna och tillhandahåller viktig data för analys, design och test. Användningsfallsmodellen innehåller aktörer, användningsfall och deras relationer och beskriver vad systemet gör för varje typ av användare.

Aktör

Varje typ av användare och varje externt system som interagerar med systemet representeras som en eller flera aktörer. Aktörer representerar alltså deltagare utanför systemet som samarbetar med systemet. När man har identifierat systemets alla aktörer, har man även identifierat systemets omgivning.

Användningsfall

Varje sätt aktörerna använder systemet representeras som ett användningsfall [2]. Användningsfall är fragment av funktionalitet som är av värde för systemets aktörer och specificerar en sekvens av händelser som systemet kan utföra. En användningsfalls-beskrivning kan innehålla tillståndsdiagram, aktivitetsdiagram, samarbetsdiagram och sekvensdiagram.

Tillståndsdiagram specificerar användningsfallsinstansers livscykel i termer av tillstånd och övergångar mellan tillstånd. En användningsfallsinstans är exekveringen av ett användningsfall. Aktivitetsdiagram beskriver livscykeln i mer detalj genom att också beskriva sekvensen av händelser som inträffar inom varje övergång. Samarbets- och sekvensdiagram används för att beskriva interaktionen mellan en aktörinstans och en användningsfallsinstans.

Händelseförlopp

Ett händelseförlopp specificerar vad systemet gör och hur det interagerar med aktörerna när ett användningsfall utförs. Händelseförloppet för ett användningsfall kan beskrivas i ett separat textdokument.

Specialkrav

Specialkraven är en textbeskrivning som huvudsakligen samlar icke-funktionella krav som är relaterade till användningsfallet och som behöver hanteras i efterföljande arbetsflöden.

Arkitekturbeskrivning

Arkitekturbeskrivningen ska innehålla en arkitekturell vy av användningsfallsmodellen, som beskriver de arkitekturellt betydelsefulla användningsfallen. Den arkitekturella vyn av användningsfallsmodellen ska omfatta användningsfall som beskriver viktig och kritisk funktionalitet eller som innefattar viktiga krav som måste utvecklas tidigt i programvarans livscykel. Vyn används som en utgångspunkt när användningsfall prioriteras inom en iteration.

Ordlista

En ordlista kan användas för att definiera viktiga och vanliga termer som används av utvecklare när de beskriver systemet och minskar risken för missförståelse.

Användargränssnittsprototyp

Användargränssnittsprototyper hjälper utvecklarna att förstå interaktionerna mellan mänskliga aktörer och systemet, samt bidrar till en ökad förståelse av användningsfallen.

3.1.2 Arbetsflöde

Arbetsflödet för kravhantering innehåller följande aktiviteter:

- Hitta aktörer och användningsfall
- Prioritera användningsfall
- Specificera användningsfall
- Bygga användargränssnittsprototyper
- Strukturera användningsfallsmodellen

Resultatet av den första iterationen genom detta arbetsflöde består av en första version av användningsfallsmodellen, användningsfallen och associerade användargränssnittsprototyper [2]. Resultatet av efterföljande iterationer består av nya versioner av dessa artefakter.

Hitta aktörer och användningsfall

Syftet med att identifiera användningsfall och aktörer är att:

- Begränsa systemet från dess omgivning.
- Ange vem och vad som kommer att interagera med systemet.
- Ange vilken funktionalitet som förväntas av systemet.
- Definiera vanliga termer i en ordlista.

Att identifiera aktörer och användningsfall är den mest avgörande aktiviteten för att fastställa kraven rätt. Aktiviteten består av fyra steg [2]:

- Hitta aktörerna.
- Hitta användningsfallen.
- Kortfattat beskriva varje användningsfall.
- Beskriva användningsfallsmodellen som helhet.

Stegen behöver inte utföras i någon särskild ordning och utförs ofta samtidigt. Resultatet av aktiviteten är en ny version av användningsfallsmodellen med nya och modifierade aktörer och användningsfall.

Prioritera användningsfall

Målet med denna aktivitet är att fastställa vilka användningsfall som behöver utvecklas i tidiga iterationer och vilka som kan utvecklas i senare. Resultatet fastställs i den arkitekturella vyn av användningsfallsmodellen.

Specificera användningsfall

I denna aktivitet beskriver man varje användningsfalls händelseförlopp i detalj. Man beskriver hur användningsfallet startar, slutar och interagerar med aktörer. Man beskriver också alla alternativa vägar genom användningsfallet. Målet är att göra en beskrivning som är noggrann och lätt att läsa. Resultatet uttrycks i text och diagram.

Bygga användargränssnittsprototyper

Man bygger användargränssnittsprototyper för att kunna designa användargränssnitt som låter användaren utföra användningsfallen effektivt. Resultatet av aktiviteten är en uppsättning skisser och prototyper som specificerar utseendet på användargränssnitten för de viktigaste aktörerna.

Strukturera användningsfallsmodellen

Användningsfallsmodellen struktureras för att [2]:

- Extrahera generella och delade beskrivningar av funktionalitet som kan användas av mer specifika användningsfallsbeskrivningar.
- Extrahera extra eller valbara beskrivningar av funktionalitet som kan utvidga mer specifika användningsfallsbeskrivningar.

För att minska redundans extraherar man händelser eller delar av händelser som är gemensamma för många användningsfall. Dessa händelser beskrivs i ett separat användningsfall, en generalisering, som sedan kan återanvändas av de ursprungliga användningsfallen. Generaliseringar används för att förenkla arbetet med och förståelsen av användningsfallsmodellen.

En utvidgning specificerar händelser som utökar den ursprungliga beskrivningen av ett användningsfall.

3.2 Analys

I analysen förfinar och strukturerar man kraven för att få en bättre förståelse av dem och för att åstadkomma en beskrivning som är lätt att underhålla och som ger struktur åt hela systemet.

Analysmodellen kan sammanfattas med följande punkter (jämför med användningsfalls-modellen):

- Beskrivs med utvecklarens språk.
- Är en inre vy av systemet.
- Struktureras med stereotypiska klasser och paket; ger struktur åt den inre vyn.
- Används i första hand av utvecklare för att förstå hur systemet ska utformas, dvs. designas och implementeras.
- Ska inte innehålla redundans eller inkonsekvenser bland kraven.
- Anger översiktligt hur man realiserar funktionaliteten inom systemet; fungerar som en första skiss av designen.
- Definierar användningsfallsrealiseringar.

Genom att begränsa frågeställningar och beslut i analysen förbereder man för och förenklar man de efterföljande design- och implementationsaktiviteterna.

3.2.1 Artefakter

Analysmodell

Analysmodellen består av analyspaket, analysklasser och användningsfallsrealiseringar– analys. Analyspaket organiserar analysmodellen i mer hanterbara delar, som representerar abstraktioner av delsystem eller hela lager i systemets design. En analysklass representerar en abstraktion av en eller flera klasser eller delsystem i systemets design [2]. Inom analys-modellen realiseras användningsfall av analysklasser och deras objekt.

Analysklass

Analysklasser fokuserar på funktionella krav och deras beteende definieras av ansvarigheter på en högre, mindre formell nivå. Analysklasser definierar attribut, även dessa är på en högre nivå. Typerna av attribut är ofta konceptuella och känns igen ifrån problemområdet, medan attribut i design- och implementationsklasser ofta är av programmeringsspråkstyper. Attribut som påträffas under analysen blir vanligtvis egna klasser i design och implementation. Analysklasser har relationer, fast dessa relationer är mer konceptuella än deras design- och implementationsmotsvarigheter.

Det finns tre grundläggande stereotyper av analysklasser: gränssnittsklasser, kontrollklasser och entitetsklasser. Varje stereotyp innebär specifik semantik, vilket resulterar i en kraftfull och konsekvent metod för att hitta och beskriva analysklasser. Dessa tre stereotyper är standardiserade i UML och varje stereotyp har sin egen symbol.

En gränssnittsklass används för att beteckna interaktion mellan systemet och dess aktörer. Interaktionen medför ofta att information tas emot från och presenteras till användare och externa system.

Entitetsklasser betecknar information som är långvarig eller bestående. De modellerar något fenomen eller koncept ifrån verkligheten, som till exempel en individ, ett objekt eller en händelse. Entitetsklasser reflekterar informationen på ett sätt som gynnar utvecklarna när de designar och implementerar systemet [2].

Kontrollklasser representerar koordination, händelseförlopp, transaktioner och kontroll av andra objekt. De används också för att representera komplexa beräkningar. Dynamiken i systemet finns i kontrollklasserna, eftersom de delegerar arbete till andra objekt (gränssnitts- och entitetsobjekt).

Användningsfallsrealisering–Analys

En användningsfallsrealisering–analys beskriver hur ett specifikt användningsfall utförs i termer av analysklasser och deras interagerande analysobjekt. En användningsfallsrealisering har en textbeskrivning av händelseförloppet, klassdiagram som skildrar dess deltagande analysklasser och interaktionsdiagram som skildrar realiseringen av ett särskilt flöde eller scenario av användningsfallet.

Analyspaket

Analyspaket organiserar artefakterna i analysmodellen i mindre, mer hanterbara delar. Ett analyspaket kan bestå av analysklasser, användningsfallsrealiseringar och andra analyspaket.

Arkitekturbeskrivning

Arkitekturbeskrivningen ska innehålla en arkitekturell vy av analysmodellen, som skildrar dess arkitekturellt betydelsefulla artefakter. Artefakter som kan anses vara betydelsefulla är: analyspaket och deras beroenden, viktiga analysklasser och användningsfallsrealiseringar som realiserar viktig och kritisk funktionalitet.

3.2.2 Arbetsflöde

Följande aktiviteter ingår i arbetsflödet för analys:

- Arkitekturell analys
- Analys av användningsfall
- Analys av klasser
- Analys av paket

Arkitekturell analys

Syftet med arkitekturell analys är att ange huvuddragen i analysmodellen och arkitekturen genom att identifiera analyspaket, uppenbara analysklasser och gemensamma specialkrav [2].

Analyspaket kan antingen identifieras i början som ett sätt att dela upp analysarbetet eller hittas under tiden som analysmodellen utvecklas till en stor struktur.

En första identifiering av analyspaket baseras naturligt på funktionella krav och problemområdet. Ett enkelt sätt att identifiera analyspaket är att tilldela ett antal användningsfall till ett specifikt paket och sedan realisera motsvarande funktionalitet inom paketet. Lämpliga allokeringar av användningsfall till ett specifikt paket är t.ex. användningsfall som stöder samma verksamhetsprocess, användningsfall som stöder samma aktör eller användningsfall som är relaterade via generaliseringar och utvidgningsrelationer.

Man ska definiera beroenden mellan analyspaket om deras innehåll har relationer till vartannat. Målet är dock att hitta paket som är relativt oberoende av och löst kopplade till varandra, vilket gör paketen lättare att underhålla.

Det är även lämpligt att i den arkitekturella analysen ta fram förslag över de viktigaste entitetsklasserna.

Gemensamma specialkrav är krav som förekommer under analysen och som är viktiga att fastställa inför kommande design- och implementationsaktiviteter. Exempel är restriktioner gällande säkerhet och feltolerans.

Analys av användningsfall

Man analyserar ett användningsfall för att identifiera de analysklasser som behövs för att realisera användningsfallet. Följande generella riktlinjer kan användas för att identifiera analysklasser [2]:

- Identifiera entitetsklasser genom att studera användningsfallsbeskrivningen och överväg sedan vilken information som måste innefattas och manipuleras i användningsfallsrealiseringen.
- Identifiera en central gränssnittsklass för varje mänsklig aktör och låt denna klass representera det primära fönstret i användargränssnittet som aktören interagerar med.
- Identifiera en primitiv gränssnittsklass för varje entitetsklass som hittades tidigare. Dessa klasser representerar logiska objekt som aktören interagerar med i användargränssnittet under användningsfallet.
- Identifiera en central gränssnittsklass för varje extern systemaktör och låt denna klass representera kommunikationsgränssnittet.
- Identifiera en kontrollklass som är ansvarig för att hantera kontroll och koordination av användningsfallsrealiseringen och förfina sedan denna kontrollklass enligt användningsfallets krav.

Analysklasserna samlas i ett klassdiagram. Klassdiagrammet används sedan för att visa relationerna i användningsfallsrealiseringen.

När man har funnit de analysklasser som behövs för att realisera användningsfallet, beskriver man hur deras motsvarande analysobjekt interagerar. För detta ändamål används samarbetsdiagram som innehåller de deltagande aktörinstanserna, analysobjekten och deras länkar. Objektinteraktion illustreras genom att man skapar länkar mellan objekten och fäster meddelanden till dessa länkar. Meddelandet ska ange syftet med interaktionen.

Analys av klasser

Syftet med att analysera en klass är att identifiera analysklassens ansvarigheter, attribut och relationer. Klassens ansvarigheter kan sammanställas genom att slå ihop alla roller som den spelar i olika användningsfallsrealiseringar. Attribut specificerar analysklassens egenskaper och antyds ofta av klassens ansvarigheter.

Analys av paket

Syftet med att analysera ett paket är att se till att analyspaketet är så oberoende av andra paket som möjligt, försäkra sig att analyspaketet uppfyller sitt syfte och beskriva beroenden så att följden av framtida förändringar kan uppskattas.

3.3 Design

I designen formas systemet så att det uppfyller alla krav. Resultatet av analysen (analysmodellen) förser designen med viktig data. Analysmodellen inför en struktur av systemet som man ska sträva efter att bevara så mycket som möjligt när systemet formas [2]. Syftet med design är att [2]:

- Skaffa sig en ingående förståelse över frågor gällande icke-funktionella krav och restriktioner relaterade till programmeringsspråk, komponentåteranvändning, operativsystem, distributions- och samtidighetsteknologier, databasteknologier, användargränssnittsteknologier, transaktionshanteringsteknologier och så vidare.
- Skapa en utgångspunkt för efterföljande implementationsaktiviteter genom att fastställa krav på enskilda delsystem, gränssnitt och klasser.
- Kunna dela upp implementationsarbetet i mer hanterbara delar.
- Fastställa större gränssnitt mellan delsystem tidigt i programvarans livscykel.
- Kunna visualisera och resonera kring designen genom att använda ett gemensamt notationssätt.
- Skapa en abstraktion av systemets implementation.

3.3.1 Artefakter

Designmodell

Designmodellen är en objektmodell som beskriver den fysiska realiseringen av användnings-fall [2]. Designmodellen fungerar som en abstraktion av systemets implementation och förser därmed implementationsaktiviteterna med viktig data.

Designklass

En designklass är en abstraktion av en klass i systemets implementation. Språket som används för att specificera en designklass är detsamma som programmeringsspråket. Operationer, parametrar, attribut, typer, osv. specificeras alltså med vald programmeringsspråkssyntax. En designklass relationer och metoder har ofta direkta motsvarigheter i klassens implementation.

En designklass kan vara aktiv, vilket innebär att objekt av klassen upprätthåller sin egen tråd och exekveras samtidigt som andra aktiva objekt.

Användningsfallsrealisering–Design

En avändningsfallsrealisering–design beskriver hur ett specifikt användningsfall realiseras i termer av designklasser och deras objekt. En användningsfallsrealisering–design har en direkt koppling till en användningsfallsrealisering–analys.

En användningsfallsrealisering–design har en textbeskrivning av händelseförloppet, klassdiagram som skildras dess deltagande designklasser och interaktionsdiagram som skildrar realiseringen av ett särskilt flöde eller scenario av användningsfallet i termer av interaktioner mellan designobjekt. Vid behov kan diagrammen också skildra delsystem och gränssnitt involverade i användningsfallsrealiseringen.

En användningsfallsrealisering–design är en fysisk realisering av en användningsfalls-realisering–analys och hanterar dessutom de flesta icke-funktionella krav från användnings-fallsrealiseringen–analys.

Designdelsystem

Designdelsystem organiserar artefakterna i designmodellen i mer hanterbara delar [2]. Ett delsystem kan bestå av designklasser, användningsfallsrealiseringar, gränssnitt och andra delsystem. Ett delsystem kan dessutom tillhandahålla gränssnitt som representerar dess funktionalitet utåt, i termer av operationer. Delsystem i designmodellen har ofta direkta kopplingar till analyspaket i analysmodellen.

Gränssnitt

Gränssnitt skiljer funktionalitetsspecifikationen, i termer av operationer, från deras implementation, i termer av metoder [2]. Detta gör en klient som är beroende av, eller använder, ett gränssnitt oberoende av gränssnittets implementation. En särskild implementation av ett gränssnitt kan då ersättas med en annan implementation utan att man behöver göra ändringar i klienten.

De flesta gränssnitt mellan delsystem anses vara arkitekturellt betydelsefulla eftersom de definierar hur delsystemen tillåts att interagera.

Driftsättningsmodell

Driftsättningsmodellen är en objektmodell som beskriver den fysiska distributionen av systemet i termer av hur funktionalitet distribueras mellan noder [2]. Driftsättningsmodellen förser aktiviteter i designen och implementationen med viktig data, eftersom systemets distribution har en stor inverkan på dess design.

Varje nod representerar en datorresurs, ofta en processor eller liknande hårdvaruenhet. Noder har relationer som representerar kommunikationsmedel mellan dem, såsom Internet, intranät, eller bussar. Driftsättningsmodellen kan beskriva flera olika nätverkskonfigurationer, däribland test- och simulationskonfigurationer.

Arkitekturbeskrivning

Arkitekturbeskrivningen ska innehålla en arkitekturell vy av designmodellen, som beskriver designmodellens viktigaste artefakter. Viktiga artefakter i designmodellen kan vara: delsystem, deras gränssnitt och beroenden, aktiva klasser och användningsfallsrealiseringar– design som realiserar viktig och kritisk funktionalitet som behöver utvecklas tidigt i programvarans livscykel.

Arkitekturbeskrivningen ska också innehålla en arkitekturell vy av driftsättningsmodellen. På grund av dess betydelse, ska alla aspekter av driftsättningsmodellen visas i den arkitekturella vyn.

3.3.2 Arbetsflöde

Följande aktiviteter ingår i arbetsflödet för design:

- Arkitekturell design
- Design av användningsfall
- Design av klasser
- Design av delsystem

Arkitekturell design

Syftet med arkitekturell design är att ange huvuddragen i design- och driftsättningsmodellen och deras arkitektur genom att identifiera följande [2]:

- Noder och deras nätverkskonfigurationer.
- Delsystem och deras gränssnitt.
- Arkitekturellt betydelsefulla designklasser, t.ex. aktiva klasser.
- Generella designmekanismer.

De fysiska nätverkskonfigurationerna har ofta stor inverkan på programvarans arkitektur, däribland de aktiva klasserna och distributionen av funktionalitet mellan nätverksnoder.

Om en lämplig uppdelning av analyspaket hittades under analysen, kan man använda dessa paket för att identifiera motsvarande delsystem i designmodellen. Man kan dock behöva göra förändringar för att hantera frågeställningar relaterade till designen, implementationen och driftsättningen av systemet.

En del arkitekturellt betydelsefulla designklasser kan identifieras utifrån de arkitekturellt betydelsefulla analysklasserna i analysmodellen. Man identifierar också aktiva klasser genom att betrakta systemets samtidighetskrav.

Man identifierar generella designmekanismer genom att studera gemensamma krav. Resultatet kan yttra sig som designklasser, samarbeten eller delsystem. Krav som måste hanteras är ofta relaterade till begrepp som [2]:

- Bestående
- Transparent objektdistribution
- Säkerhetsmekanismer
- Felhantering
- Transaktionshantering

Design av användningsfall

Man designar ett användningsfall för att:

- Identifiera de designklasser och delsystem som behövs för att genomföra användningsfallet.
- Distribuera användningsfallets beteende till interagerande designobjekt och delsystem.
- Definiera krav på designklassernas operationer och delsystem och deras gränssnitt.
- Fastställa implementationskraven för användningsfallet.

Designklasserna identifieras genom att studera analysklasser och specialkrav i motsvarande användningsfallsrealisering–analys. Designklasserna samlas i ett klassdiagram, som visar relationerna i användningsfallsrealiseringen.

När man har klart för sig vilka designklasser som behövs för att realisera användningsfallet, beskriver man hur motsvarande designobjekt interagerar. För detta ändamål används sekvensdiagram, som innehåller de deltagande aktörinstanserna, designobjekten och överföringen av meddelanden mellan dem. Ordningsföljden i diagrammet ska vara i fokus, eftersom användningsfallsrealiseringen–design används som utgångspunkt när användningsfallet implementeras. En beskrivning av händelseförloppet kan användas för att komplettera sekvensdiagrammet.

Ibland är det också lämpligt att designa ett användningsfall i termer av delsystem och deras gränssnitt. Även här använder man klassdiagram och sekvensdiagram, dock med några små skillnader.

Till sist fastställer man krav för användningsfallsrealiseringen som identifierades under designen, men som ska hanteras under implementationen.

Design av klasser

Syftet med att designa en klass är att skapa en designklass som uppfyller sin roll i de användningsfallsrealiseringar och icke-funktionella krav som gäller för klassen. Följande aspekter av klassen skall tas i beaktning:

- Dess operationer
- Dess attribut
- Dess relationer
- Dess metoder (som realiserar dess operationer)
- Dess olika tillstånd
- Dess beroenden till generella designmekanismer
- Relevanta krav för dess implementation
- Realiseringen av gränssnitt som klassen ska tillhandahålla

Som ett första steg anger man huvuddragen för en designklass genom att använda sig av motsvarande analysklass. Metoden som används beror på analysklassens stereotyp. När man designar gränssnittsklasser är man beroende av den specifika gränssnittsteknologin. Att designa entitetsklasser som representerar bestående information antyder ofta användning av en specifik databasteknologi. När man designar kontrollklasser måste man ta hänsyn till distributionsfrågor, prestandafrågor och transaktionsfrågor.

Designklassens operationer beskrivs med programmeringsspråkets syntax. Viktiga data för att identifiera operationerna är motsvarande analysklass ansvar och specialkrav, gränssnitt som designklassen ska tillhandahålla och de användningsfallsrealiseringar–design i vilka klassen deltar. En designklass operationer måste stödja alla roller som klassen spelar i olika användningsfallsrealiseringar.

Även designklassens attribut beskrivs med programmeringsspråkets syntax. Ett attribut specificerar en designklass egenskap och kan ofta härledas från klassens operationer. De tillgängliga attributtyperna begränsas av programmeringsspråket.

Klassmetoder kan användas under designen för att specificera hur operationer realiseras. Man använder då antingen naturligt språk eller pseudokod. För det mesta specificeras dock inte metoder under designen, utan de skapas direkt med programmeringsspråket under implementationen.

Design av delsystem

Syftet med att designa ett delsystem är att se till att följande kriterier uppnås:

- Att delsystemet är så oberoende som möjligt av andra delsystem och deras gränssnitt.
- Att delsystemet tillhandahåller rätt gränssnitt.
- Att delsystemet uppfyller sitt syfte genom att på ett korrekt sätt realisera operationerna som definieras av dess gränssnitt.

Man definierar beroenden mellan ett delsystem och andra delsystem. Om de andra delsystemen tillhandahåller gränssnitt definieras beroenden gentemot dessa istället, eftersom det är bättre att vara beroende av ett gränssnitt än av ett delsystem. Ett delsystem kan då ersättas med ett annat delsystem, innehållandes en annan inre design, medan gränssnittet inte behöver ersättas i det fallet. Klasser i ett delsystem som är för beroende av andra delsystem bör övervägas att omlokaliseras, så att beroenden till andra delsystem minimeras.

3.4 Implementation

Under implementationen startar man med resultatet från designen och implementerar systemet i termer av komponenter; källkod, skript, binärfiler, exekverbara filer och liknande. Syftet med implementation är att [2]:

- Planera de systemintegreringar som behövs i varje iteration.
- Distribuera systemet genom att allokera exekverbara komponenter till noder i driftsättningsmodellen.
- Implementera de designklasser och delsystem som hittades under designen.
- Enhetstesta komponenterna och sedan integrera dem genom att kompilera och länka samman dem till en eller flera exekverbara filer.

3.4.1 Artefakter

Implementationsmodell

Implementationsmodellen beskriver hur element i designmodellen implementeras i termer av komponenter.

Komponent

En komponent är den fysiska motsvarigheten av modellelement. Komponenter har följande karakteristik:

- Det är vanligt att en komponent implementerar många element, t.ex. designklasser.
- Komponenter har samma gränssnitt som modellelementen de implementerar.
- Det kan finnas kompileringsberoenden mellan komponenter, som betecknar vilka komponenter som är nödvändiga för att kompilera en specifik komponent.

Implementationsdelsystem

Implementationsdelsystem organiserar artefakterna i implementationsmodellen i mer hanterbara delar. Ett delsystem kan bestå komponenter, gränssnitt och andra delsystem. Implementationsdelsystem är starkt relaterade till designdelsystem i designmodellen. Följande gäller för implementationsdelsystem i hänseende till motsvarande designdelsystem:

- Implementationsdelsystemet ska definiera motsvarande beroenden gentemot andra (motsvarande) implementationsdelsystem och/eller gränssnitt.
- Implementationsdelsystemet ska tillhandahålla samma gränssnitt.
- Implementationsdelsystemet ska definiera vilka komponenter, eller andra implementationsdelsystem inom delsystemet, som ska tillhandahålla gränssnitten som tillhandahålls av delsystemet själv.

Gränssnitt

Man kan använda samma gränssnitt i implementationsmodellen som i designmodellen för att specificera de operationer som implementeras av komponenter och implementations-delsystem. En komponent som realiserar, och därmed tillhandahåller, ett gränssnitt måste på ett korrekt sätt implementera alla operationer som definieras av gränssnittet. Ett implementationsdelsystem som tillhandahåller ett gränssnitt måste innehålla komponenter eller andra delsystem som tillhandahåller gränssnittet.

Arkitekturbeskrivning

Arkitekturbeskrivningen ska innehålla en arkitekturell vy av implementationsmodellen, som skildrar dess arkitekturellt betydelsefulla artefakter. Artefakter i implementationsmodellen som kan anses vara arkitekturellt betydelsefulla är: delsystem, deras gränssnitt och beroenden, komponenter som implementerar arkitekturellt betydelsefulla designklasser, exekverbara komponenter och komponenter som är generella, centrala och implementerar allmänna designmekanismer som många andra komponenter är beroende av.

Integrationsbyggnadsplan

Ett bygge (eng. build) är en exekverbar version av systemet. Den funktionalitet som ska implementeras i en specifik iteration är ofta alltför komplex för att integreras i ett enda bygge. Istället kan en följd av byggen behövas inom iterationen. En integrationsbyggnadsplan beskriver sekvensen av byggen som behövs i en iteration. Mer specifikt beskrivs följande för varje bygge:

- Den funktionalitet som förväntas att implementeras i bygget. Detta är en förteckning över användningsfall och/eller scenarion eller delar av dem. Förteckningen kan också referera till andra kompletterande krav.
- Vilka delar av implementationsmodellen som berörs av bygget. Detta är en förteckning över de delsystem och komponenter som fordras för att implementera funktionaliteten.

3.4.2 Arbetsflöde

Följande aktiviteter ingår i arbetsflödet för implementation:

- Arkitekturell implementation
- Integration av system
- Implementation av delsystem
- Implementation av klasser
- Utförande av enhetstest

Arkitekturell implementation

Syftet med arkitekturell implementation är att ange grunddragen i implementationsmodellen och dess arkitektur genom att:

- Identifiera arkitekturellt betydelsefulla komponenter, t.ex. exekverbara komponenter.
- Allokera komponenter till noder.

Att identifiera implementationsdelsystem och deras gränssnitt är mer eller mindre trivialt, eftersom motsvarande designdelsystem och gränssnitt identifierades under designen. Den stora utmaningen under implementationen är att i ett implementationsdelsystem skapa komponenter som implementerar motsvarande designdelsystem.

För att identifiera exekverbara komponenter betraktar man de aktiva klasserna som hittades under designen och tilldelar en exekverbar komponent per aktiv klass. Om det finns några aktiva objekt allokerade till noder i design- eller driftsättningsmodellen, så ska komponenterna placeras på samma noder som motsvarande aktiva klasser.

Integration av system

Syftet med systemintegration är att [2]:

- Skapa en integrationsbyggnadsplan som beskriver de byggen som behövs i en iteration och kraven för varje bygge.
- Integrera varje bygge innan det utsätts för integrationstester.

Varje bygge ska lägga till funktionalitet till föregående bygge genom att implementera hela användningsfall eller scenarion av dem. Integrationstesterna baseras sedan på dessa användningsfall. Det är av stor betydelse att identifiera rätt krav som ska implementeras i ett bygge och att lämna resten av kraven till framtida byggen.

Implementation av delsystem

Syftet med att implementera ett delsystem är att se till att delsystemet uppfyller sin roll i varje bygge. Ett delsystem uppfyller sin roll när de krav som ska implementeras i det aktuella bygget och de krav som påverkar delsystemet implementeras på ett korrekt sätt av komponenter och andra delsystem inom delsystemet.

Implementation av klasser

Man implementerar en designklass i en filkomponent, vilket innebär att man:

- Anger en filkomponent som ska innehålla källkoden. Det är vanligt att man implementerar många designklasser i en enda filkomponent.
- Genererar källkod från designklassen och dess relationer.
- Implementerar designklassens operationer i termer av metoder. Att implementera en operation innebär att välja en passande algoritm och gynnande datastrukturer.
- Ser till att komponenten tillhandahåller samma gränssnitt som designklassen.

Utförande av enhetstest

Syftet med att utföra enhetstest är att testa de implementerade komponenterna som enskilda enheter. Följande typer av enhetstestning görs [2]:

- Specifikationstestning – verifierar enhetens externa, observerbara beteende.
- Strukturtestning – verifierar enhetens interna implementation.

Det kan även förekomma andra tester för vissa enheter, t.ex. test av prestanda, minnesanvändning, belastning och kapacitet.

Specifikationstestning verifierar en komponents beteende utan att betrakta hur beteendet implementeras. Specifikationstesterna undersöker vilka utdata en komponent returnerar givet viss indata och ett visst tillstånd. Antalet kombinationer av möjliga indata, tillstånd och utdata är ofta väldigt omfattande, vilket gör det opraktiskt att testa alla enskilda kombinationer. Istället delas indata, utdata och tillstånd in i ekvivalenta klasser. En ekvivalent klass är en uppsättning värden för indata, tillstånd eller utdata för vilka ett objekt antas att uppföra sig likadant. Genom att testa en komponent för varje kombination av de ekvivalenta klasserna av indata, utdata och tillstånd kan man uppnå nästan samma effekt som att testa alla enskilda kombinationer av värden.

Strukturtestning verifierar att en komponent internt fungerar som avsett. Under strukturtestning ska man testa all kod, vilket betyder att varje sats måste exekveras minst en gång. Man ska också testa de mest intressanta vägarna genom koden, vilket omfattar de vanligaste, de mest kritiska, de minst kända och andra vägar som förknippas med höga risker.

3.5 Test

I arbetsflödet för test verifierar man resultatet från implementationen genom att testa varje bygge. Syftet med test är att [2]:

- Planera integrationstester och systemtester för varje iteration. Integrationstester fordras för varje bygge inom iterationen, medan systemtester endast behövs i slutet av iterationen.
- Designa och implementera testerna genom att skapa testfall som specificerar vad som ska testas, skapa testprocedurer som specificerar hur testerna ska genomföras och om möjligt skapa exekverbara testkomponenter som automatiserar testerna.
- Utföra de olika testerna och systematiskt hantera resultatet från varje test. Byggen som innehåller fel testas igen och skickas eventuellt tillbaka till andra arbetsflöden så att felen kan rättas till.

3.5.1 Artefakter

Testmodell

Testmodellen beskriver huvudsakligen hur exekverbara komponenter i implementationsmodellen testas genom integrations- och systemtester. Testmodellen kan också beskriva hur specifika aspekter av systemet ska testas, t.ex. om användargränssnittet är användbart och konsekvent eller om systemets användarhandbok uppfyller sitt syfte. Testmodellen är en samling av testfall, testprocedurer och testkomponenter [2].

Testfall

Ett testfall specificerar *ett* sätt att testa systemet, däribland vad som ska testas och under vilka villkor. Vad som ska testas kan i praktiken vara vilket systemkrav som helst. Vanliga testfall är [2]:

- Ett testfall som specificerar hur man testar ett användningsfall eller ett specifikt scenario av ett användningsfall. Ett sådant testfall antyder oftast ett ”black-box”-test, dvs. ett test av systemets externa beteende.
- Ett testfall som specificerar hur man testar en användningsfallsrealisering–design eller ett specifikt scenario av realiseringen. Ett sådant testfall antyder oftast ett ”white-box”-test, dvs. ett test av systemets interna interaktion.

Testprocedur

En testprocedur specificerar hur man utför en eller flera testfall eller delar av dem. En testprocedur kan vara instruktioner för hur man manuellt utför ett testfall, eller en beskrivning över hur man interagerar med ett testautomatiseringsverktyg för att skapa exekverbara testkomponenter.

Testkomponent

En testkomponent automatiserar en eller flera testprocedurer eller delar av dem. Testkomponenter kan utvecklas genom att använda ett skriptspråk, ett programmeringsspråk eller ett testautomatiseringsverktyg. Testkomponenter används för att testa komponenterna i implementationsmodellen genom att tillhandahålla testdata, kontrollera och övervaka exekveringen och rapportera testresultaten.

Testplan
En testplan beskriver teststrategierna, resurserna och tidsschemat.

Defekt
En defekt är en systemavvikelse, såsom ett symptom av ett programvarufel.

Testutvärdering
En testutvärdering är en utvärdering av testresultaten.

3.5.2 Arbetsflöde
Följande aktiviteter ingår i arbetsflödet för test:

- Planering av test
- Design av test
- Implementation av test
- Utförande av integrationstest
- Utförande av systemtest
- Utvärdering av test

Planering av test
Syftet med att planera ett test är att för varje iteration:

- Skapa en teststrategi
- Uppskatta kraven, t.ex. resurser
- Planera tidsschemat

Eftersom inget system kan testas fullständigt, identifierar man testfall, procedurer och komponenter som ger bäst avkastning i termer av förbättrad kvalitet.

Design av test
Syftet med att designa test är att:

- Identifiera och beskriva testfall för varje bygge.
- Identifiera och strukturera testprocedurer som specificerar hur man utför testfallen.

Integrationstestfall verifierar att komponenterna interagerar korrekt med varandra efter att de har integrerats i ett bygge. De flesta integrationstestfall kan härledas från användningsfalls-realiseringar–design, eftersom användningsfallrealiseringarna beskriver hur klasser och objekt och därmed även hur komponenter interagerar.

Systemtester testar att systemet fungerar korrekt i sin helhet. Många systemtestfall kan hittas genom att betrakta användningsfallen, speciellt deras händelseförlopp och specialkrav.

Implementation av test
Syftet med att implementera test är att automatisera testprocedurer genom att skapa testkomponenter utifrån dem. Inte alla testprocedurer kan automatiseras. Testkomponenterna

använder ofta stora mängder av indata och producerar stora mängder av utdata som resultat av testerna.

Utförande av integrationstest

Under denna aktivitet utförs integrationstesterna för varje bygge i en iteration. Integrationstestning uträttas i följande steg:

1. Integrationstesterna genomförs manuellt eller med testkomponenter.
2. Testresultatet jämförs med det förväntade resultatet.
3. Eventuella defekter rapporteras för utvärdering.

Utförande av systemtest

Man utför systemtester för varje iteration och fastställer testresultatet. Systemtestning utförs på motsvarande sätt som integrationstestning.

Utvärdering av test

Man utvärderar resultatet av testarbetet inom en iteration genom att jämföra resultatet med målen i testplanen.

4 Studie av Design Patterns

I det här kapitlet beskriver jag först själva konceptet *design patterns* och presenterar sedan de designmönster som jag har valt att applicera på faktureringssystemet.

4.1 Design Patterns

Design patterns, eller designmönster, beskriver enkla och eleganta lösningar till specifika problem i objektorienterad programvarudesign [3]. Lösningarna har utvecklats med tiden, då utvecklare har kämpat för högre återanvändbarhet och flexibilitet i sin programvara. Designmönster hjälper utvecklare att återanvända lyckade designer och arkitekturer genom att bygga ny design på tidigare erfarenhet. En utvecklare som är bekant med dessa mönster kan direkt applicera dem i sina lösningar utan att själv behöva upptäcka dem.

De 23 ursprungliga designmönstren finns framför allt beskrivna i en bok skriven av Erich Gamma, Richard Helm, Ralph Johnson och John Vlissides (1995). De uttrycker innebörden av design patterns enligt följande:

> ”The design patterns in this book are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.” (Gamma, Helm, Johnson och Vlissides 1995, s. 3)

Författarna delar in dessa mönster i tre typer:

- Creational patterns
- Structural patterns
- Behavioral patterns

Creational patterns hanterar processen att skapa instanser. Programmet blir mer flexibelt genom att själv kunna avgöra vilka objekt som behöver skapas för ett givet fall [4].

Structural patterns beskriver hur klasser eller objekt kan sättas samman till större strukturer [4].

Behavioral patterns definierar kommunikation mellan objekt och hur flödet kontrolleras i ett komplext program [4].

Författarna delar också in designmönstren i klassmönster och objektmönster. Klassmönster beskriver relationer mellan klasser och deras subklasser. Dessa relationer skapas med hjälp av arv och är därmed statiska. Objektmönster behandlar objektrelationer som kan ändras under körningstid och är alltså mer dynamiska.

4.2 Command Pattern

Det primära syftet med command pattern är att skilja användargränssnittets objekt från händelserna de initierar. Dessa objekt ska vara helt separerade ifrån varandra och ska inte behöva veta hur andra objekt fungerar.

Användargränssnittet tar emot ett kommando ifrån användaren och ber ett command-objekt att utföra uppgiften. Användargränssnittet behöver inte veta vilka uppgifter som ska utföras.

Detta gör det möjligt att förändra händelsekoden utan att behöva ändra klasserna som innehåller användargränssnittet.

Man kan också använda command-objekt för att be programmet utföra kommandon när resurser finns tillgängliga. Slutligen kan man använda command-objekt för att minnas operationer, så att man kan stödja ångra-kommandon.

4.3 Mediator Pattern

Mediator pattern används för att förenkla kommunikation mellan klasser. Ju mer varje klass behöver känna till om andra klassers metoder, desto mer invecklad kan klasstrukturen bli. Detta gör programmet både svårare att läsa och svårare att underhålla. Dessutom kan det bli besvärligt att ändra i programmet, eftersom en förändring kan beröra kod i flera andra klasser.

Mediator pattern löser problemet genom att främja lösare koppling mellan klasser. Detta åstadkoms genom att mediator är den enda klassen som är medveten om de andra klasserna i systemet. Varje klass informerar mediator när en händelse inträffar och mediator meddelar då de klasser som berörs av händelsen. Mediator-klassen är alltså den enda klass som behöver ändras när en av de andra klasserna ändras eller när nya klasser tillkommer.

Mediator-mönstrets främsta tillämpningsområde är visuella användargränssnitt, men kan även användas i andra sammanhang där man vill underlätta komplex kommunikation mellan flera objekt.

4.4 Façade Pattern

Façade pattern omsluter ett antal komplexa klasser och tillhandahåller ett förenklat gränssnitt till dessa klasser. Denna förenkling kan i vissa fall reducera flexibiliteten i de underliggande klasserna, men tillhandahåller vanligtvis all funktionalitet som behövs. Mer sofistikerade användare kan fortfarande komma åt de underliggande klasserna och metoderna.

4.5 Iterator Pattern

Iterator pattern är ett av de enklaste och mest frekvent använda av designmönstren. Med iterator kan man förflytta sig igenom en lista eller samling av data, utan att behöva känna till detaljer om datans interna representation. Man kan också definiera särskilda iterators som uträttar någon form av beräkning på datan och endast returnerar specificerade element från datasamlingen.

Iterator pattern är användbart eftersom det tillhandahåller ett definierat sätt att förflytta sig igenom en uppsättning av dataelement, utan att exponera vad som händer inne i klassen. Iterator är ett gränssnitt och implementeras följaktligen på ett lämpligt sätt för gällande data. I Java finns redan ett iterator-gränssnitt definierat.

5 Tillämpning av Unified Process och Design Patterns vid integrering av system

Precis som i kapitlet om Unified Process har jag även här valt att dela in kapitlet i de fem arbetsflödena. Jag kan därmed presentera fullständiga diagram tillhörande varje arbetsflöde. Naturligtvis har arbetet skett i iterationer i de olika faserna, där arbete har uträttats i olika grad i de fem arbetsflödena. I avsnittet om design visar jag också klassdiagram för de design-mönster som jag har implementerat i faktureringssystemet.

5.1 Kravhantering

Kunden (Create) önskade en applikation som visar samtliga upparbetade intäkter för en viss period. Genom att ange ett datumintervall ska programmet visa en översikt över fakturorna för perioden. Man ska sedan kunna välja att visa detaljerad information över varje faktura.

Create har tre olika uppdragstyper:

- Löpande
- Fastpris
- AM (Application Management)

Ett uppdrag som är löpande innehåller en eller flera aktiviteter, där varje aktivitet är knuten till en konsult, en aktivitetstyp och en prislista.

Ett uppdrag som är fastpris innehåller en eller flera fakturaplaner, där varje fakturaplan helt enkelt består av ett fast belopp som ska betalas ett visst datum.

AM är en kombination av de två andra uppdragstyperna och kan således innehålla både aktiviteter och fakturaplaner.

Applikationen ska visa en faktura per uppdrag och eftersom det finns olika uppdragstyper, kan det förekomma fakturor med helt olika innehåll. En faktura med uppdragstyp löpande ska bland annat innehålla samtliga aktiviteter för uppdraget, rapporterade tider för varje aktivitet, samt pris för varje rapporterad tid. En rapporterad tid kan ha olika tidkoder, t.ex. normaltid eller övertid, och för varje tidkod gäller ett visst pris i prislistan.

En annan viktig detalj är att det till varje uppdrag är kopplat ett eller flera lead. Om det förekommer uppdrag med flera lead så ska det visas en faktura per uppdrag och lead, förutsatt att fakturan innehåller någon form av intäkt (aktivitet med rapporterade tider eller fakturaplan).

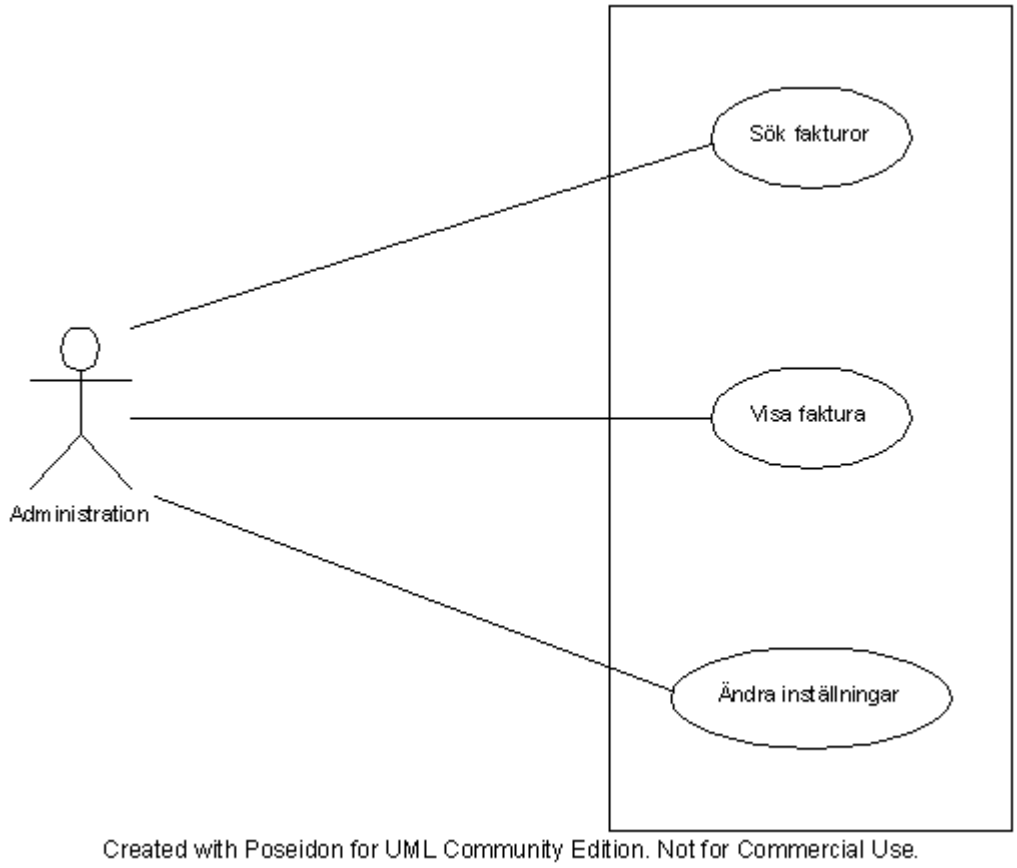
Utifrån ovanstående krav kunde jag urskilja två huvudsakliga användningsfall:

1. Sök fakturor
2. Visa faktura

Sök faktura – Genom att välja ett datumintervall ska applikationen visa samtliga fakturor för perioden.

Visa faktura – Genom att klicka på en faktura ska applikationen visa detaljerad information över fakturan.

I figur 2 visas de användningsfall som jag har valt att implementera i applikationen. Applikationen kallar jag för Invoice Manager.



Figur 2 Användningsfallsdiagram

5.2 Analys

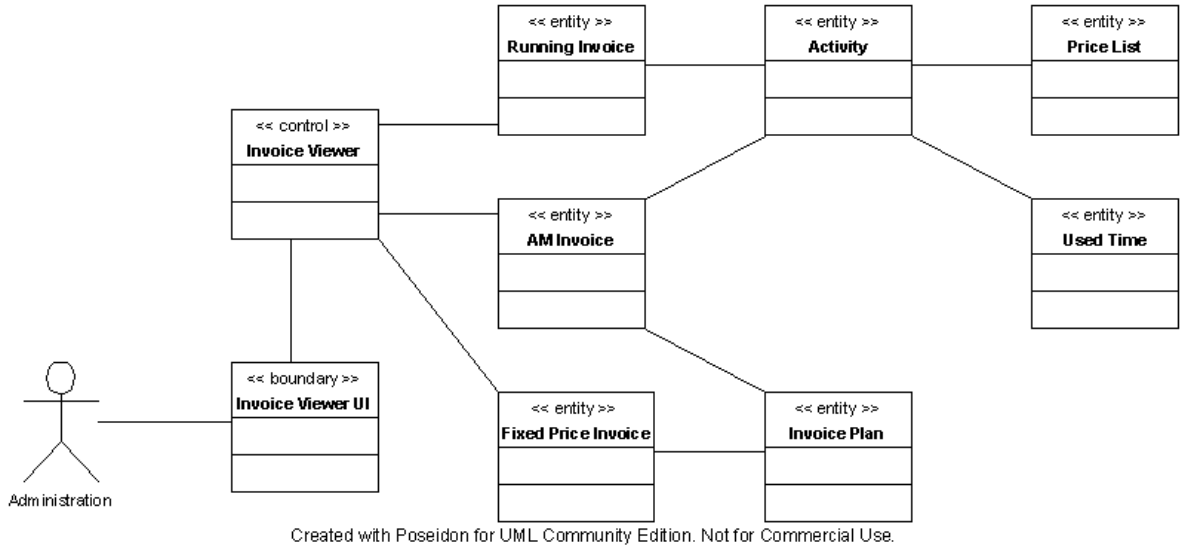
Då användningsfallen stöder samma aktör och verksamhetsprocess (fakturering), har jag under analysen endast identifierat *ett* analyspaket. Analyspaketet kallar jag för Invoice Management.

I tabell 2 visar jag de analysklasser jag identifierade för användningsfallen ”Sök fakturor” och ”Visa faktura”. Jag visar också vilken stereotyp varje analysklass har, samt vilket användningsfall det tillhör.

Tabell 2 Analysklasser för användningsfallen "Sök fakturor" och "Visa faktura".

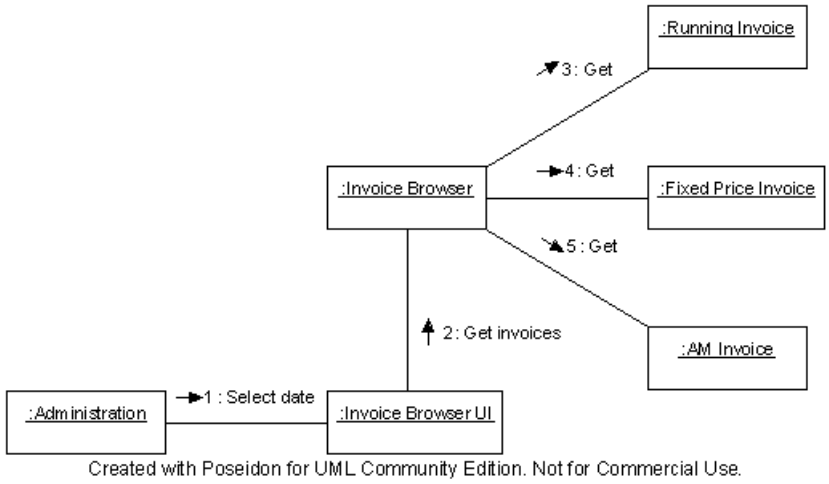
Analysklass	Stereotyp	Användningsfall
Invoice Browser UI	Gränssnitt	Sök fakturor
Invoice Viewer UI	Gränssnitt	Visa faktura
Running Invoice	Entitet	Båda
Fixed Price Invoice	Entitet	Båda
AM Invoice	Entitet	Båda
Activity	Entitet	Visa faktura
Price List	Entitet	Visa faktura
Used Time	Entitet	Visa faktura
Invoice Plan	Entitet	Visa faktura
Invoice Browser	Kontroll	Sök fakturor
Invoice Viewer	Kontroll	Visa faktura

Analysklasserna och deras relationer för användningsfallet ”Visa faktura” skildras av klassdiagrammet i figur 3.



Figur 3 Klassdiagram för användningsfallet "Visa faktura".

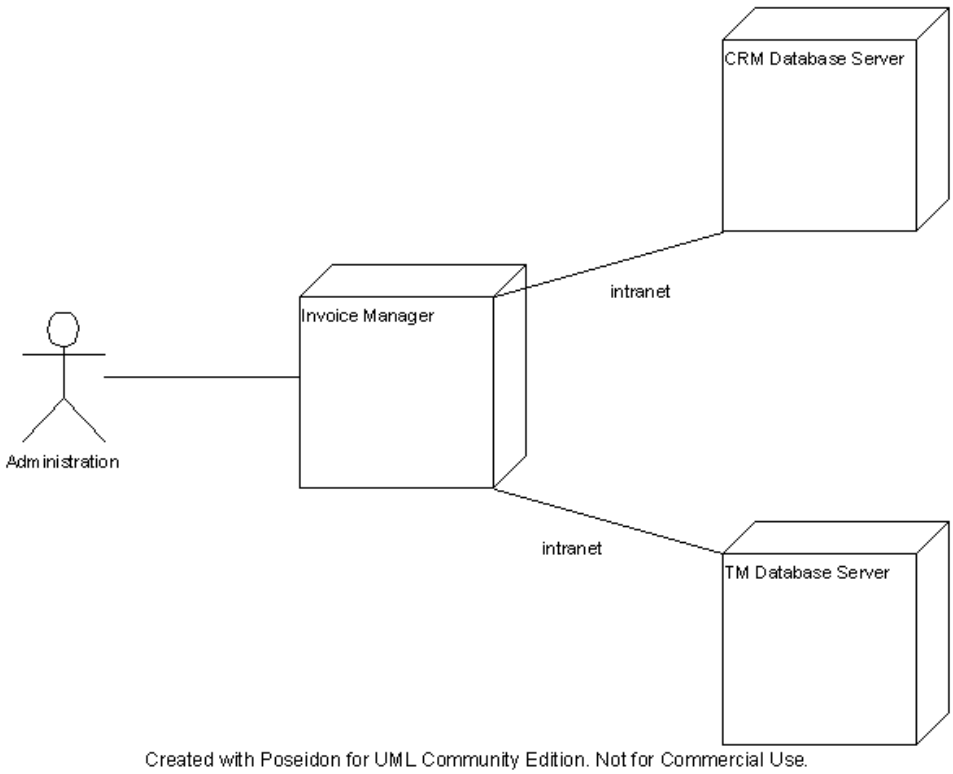
I figur 4 nedan visar jag hur ett samarbetsdiagram för *en* realisering av användningsfallet ”Sök fakturor” kan se ut.



Figur 4 Samarbetsdiagram för en realisering av användningsfallet "Sök fakturor".

5.3 Design

I figur 5 visas driftsättningsdiagrammet för faktureringssystemet. Klienten kommunicerar direkt med CRM- och TM-systemets databaser. Under utvecklandet av programmet fanns klienten och databaserna på samma dator.



Figur 5 Driftsättningsdiagram för Invoice Manager.

Utifrån analyspaketet Invoice Management skulle jag i designen kunna identifiera ett motsvarande delsystem. Jag har dock valt att dela in designmodellen ytterligare, nämligen i tre olika delsystem:

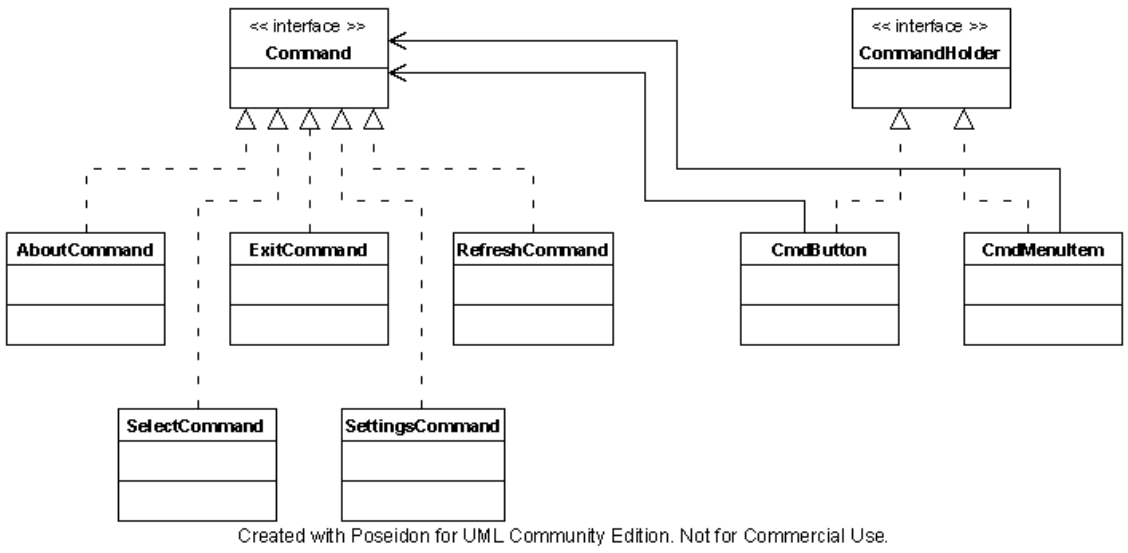
- gui – innehåller användargränssnittsklasser.
- function – består av klasser som hanterar kommunikation med databaserna.
- model – innehåller klasser som representerar objekt från verkligheten.

När det gällde designmekanismer, så var det begreppet *bestående* som var aktuellt. Mer precist så gällde det kommunikationen mellan applikationen och databaserna. JDBC (Java Database Connectivity) är ett API (Application Programming Interface) i Java som kan användas för att koppla en godtycklig databas till ett Java-program. För att det ska fungera krävs det också en drivrutin mellan gränssnittet och databasen. Drivrutiner för de vanligaste databashanteringssystemen finns att tillgå på Internet.

Efter den arkitekturella designen tyckte jag att det var lämpligt att tillämpa designmönstren på faktureringssystemet. Jag fann som sagt fyra intressanta mönster, nämligen:

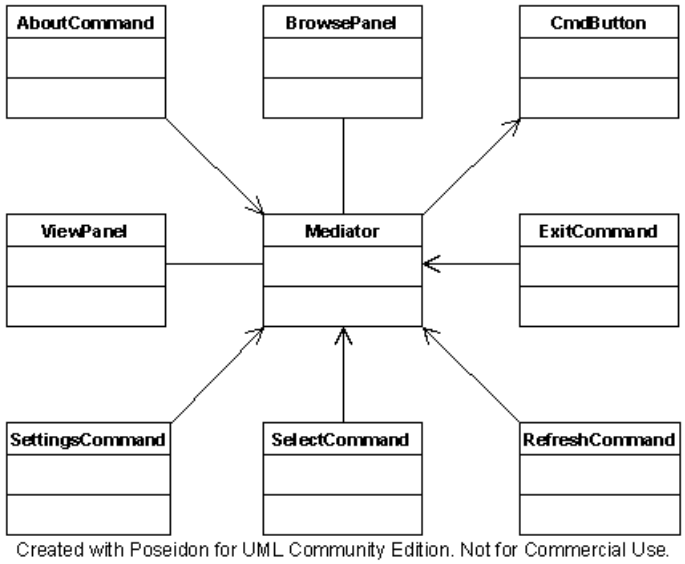
- Command pattern
- Mediator pattern
- Façade pattern
- Iterator pattern

I de följande klassdiagrammen visar jag de klasser och relationer i faktureringssystemet som är inblandade i respektive designmönster. I figur 6 visas klassdiagrammet för command pattern. Det jag la märke till med command pattern och som jag inte tidigare nämnt, var att det var väldigt enkelt att lägga till ny funktionalitet efterhand i programmet. När man lagt till en ny funktion är det bara att koppla valfritt användargränssnittsobjekt till funktionen.



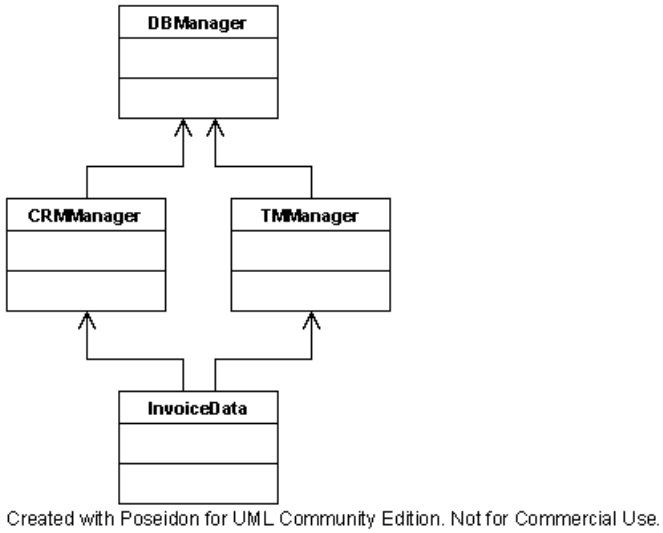
Figur 6 Klassdiagram för Command Pattern

I figur 7 skildras klassdiagrammet för mediator pattern. Som man ser i diagrammet är mediator-klassen central, då den i princip sköter all kommunikation mellan de andra klasserna. En sak som jag la märke till vid användning av mediator pattern tillsammans med command pattern i faktureringssystemet, var att kommandona till största delen utgjordes av kommunikation mellan objekt och därför skulle denna kod ligga i mediator-klassen. De olika kommando-klasserna kom därför oftast till att bara innehålla ett anrop av en metod i mediator.



Figur 7 Klassdiagram för Mediator Pattern

I figur 8 visas klassdiagrammet för façade pattern, där klasserna DBManager, CRMManager och TMManager alla fungerar som en fasad.

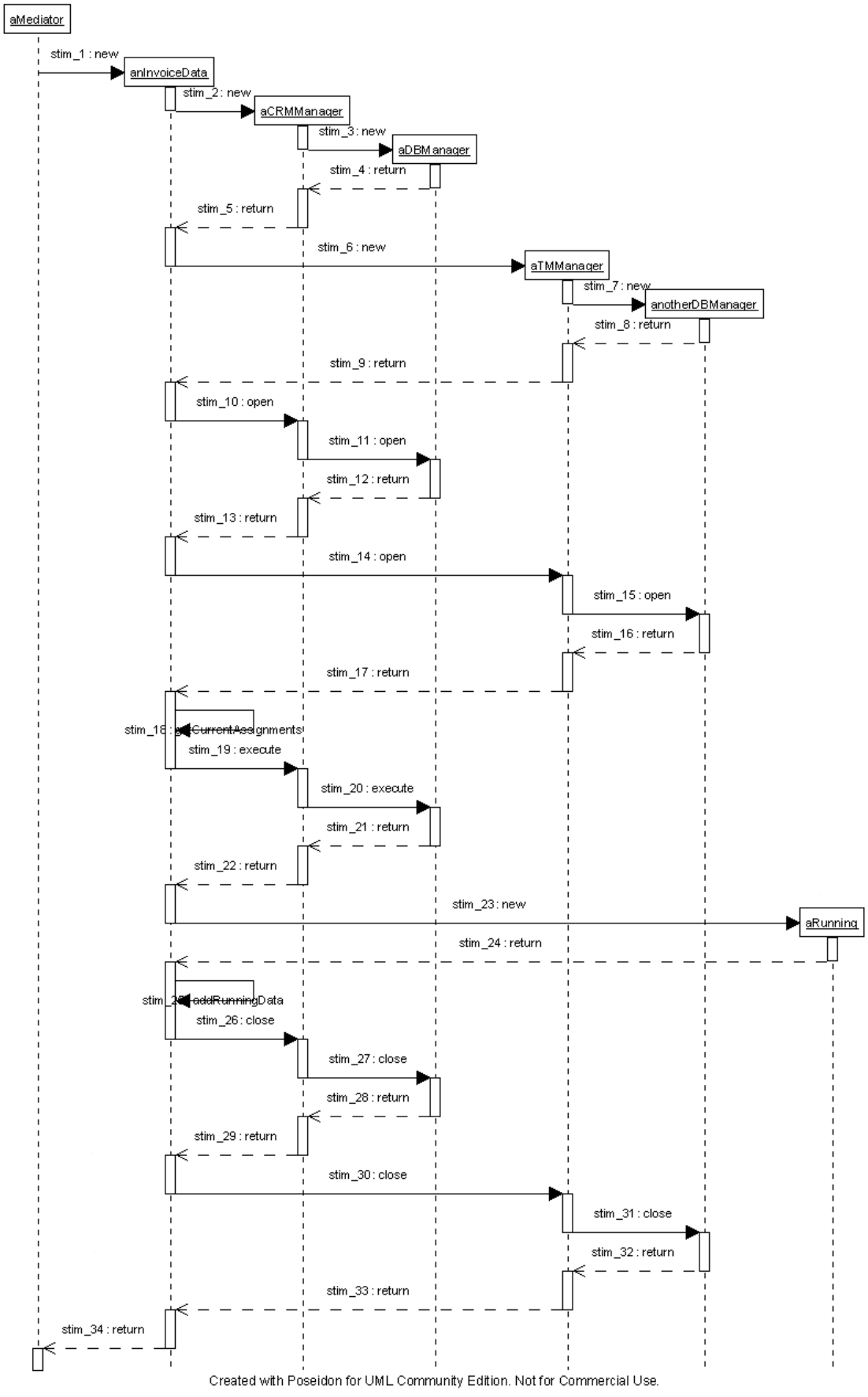


Figur 8 Klassdiagram för Façade Pattern

Jag har valt att utelämna klassdiagram för iterator pattern, då jag tycker det är tillräckligt att endast nämna vilka klasser som använder sig av designmönstret. Följande klasser itererar data genom att använda sig av ett iterator-gränssnitt: InvoiceData, InvoiceTableModel, ViewAMPanel, ViewFixedPricePanel, ViewRunningPanel, Activity, AM och Running.

Det sista jag gjorde i designen var att skapa sekvensdiagram för användningsfalls-realiseringarna. I figur 9 visar jag hur ett sekvensdiagram kan se ut för *ett* scenario av användningsfallsrealiseringen ”Sök fakturor”.

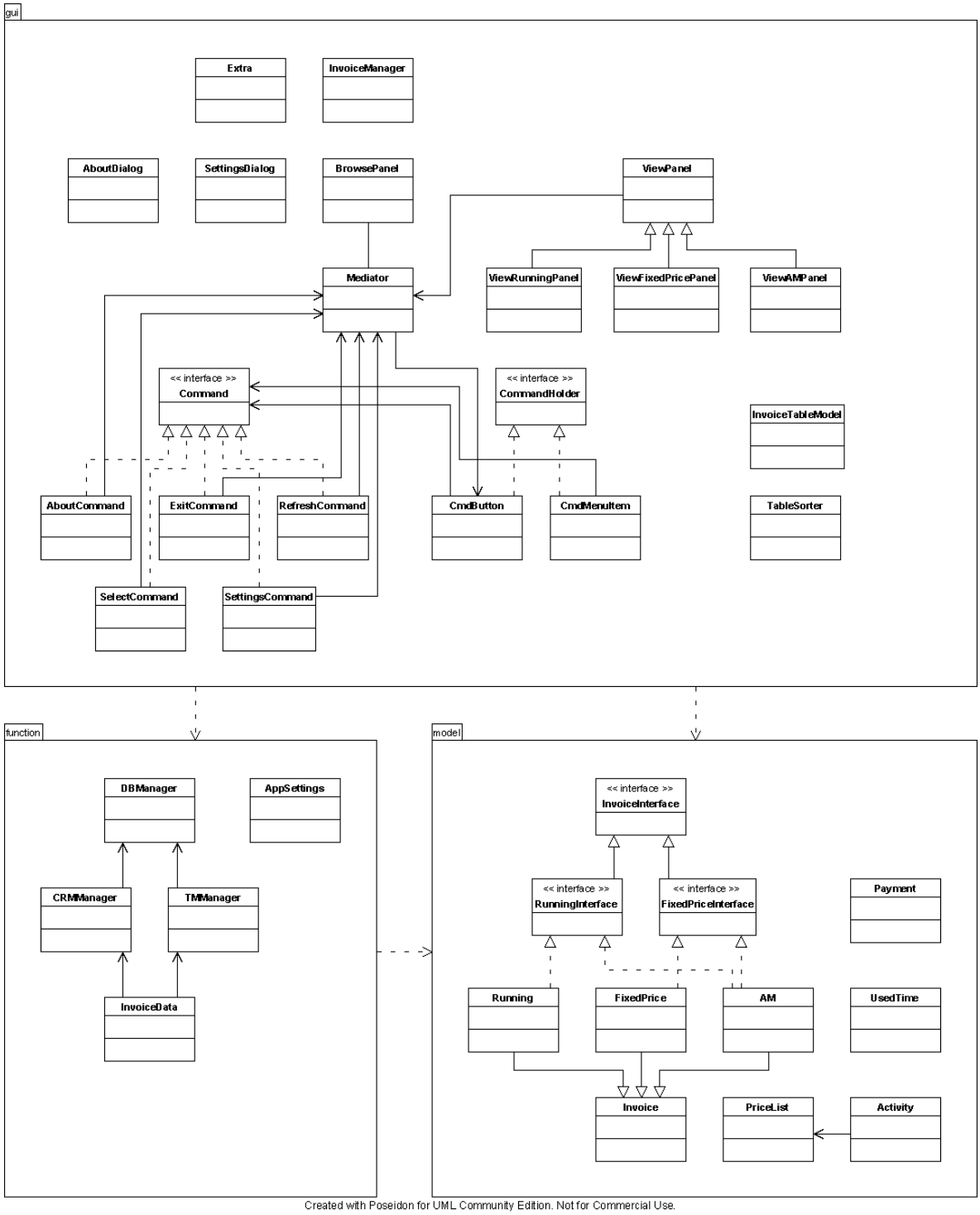
Sekvensdiagrammet är något förenklat, då ett komplett sekvensdiagram skulle kunna innehålla hundratals objekt och interaktioner mellan dem.



Figur 9 Sekvensdiagram för användningsfallsrealiseringen ”Sök fakturor”.

5.4 Implementation

Under detta arbetsflöde har själva programmeringsarbetet ägt rum och jag nöjer mig därför med att visa det slutgiltiga klassdiagrammet över hela faktureringssystemet, se figur 10. Klassdiagrammet innehåller alltså faktureringssystemets samtliga klasser och deras relationer. Jag har dock valt att utelämna relationen *beroende* (eng. dependency) mellan klasserna, eftersom klassdiagrammet annars skulle bli alldeles för komplext och klottrigt, och därmed svårt att överblicka. Faktureringssystemet implementerades i programmeringsspråket Java.



Figur 10 Slutgiltigt klassdiagram över hela faktureringssystemet.

5.5 Test

På grund av tidsbrist hann jag endast utföra tre testfall (systemtester). Varje testfall verifierar *ett* scenario av användningsfallet ”Visa faktura”. Det ena testfallet beskriver jag nedan. De andra två finns i bilaga A. Testfallen utfördes manuellt genom att direkt lägga till data i databaserna, då jag ej hade tillgång till själva systemen.

Testfall 1:

Input:

- Det finns ett lead i CRM-databasen med följande data:

Lead ID	1
Leadstatus	Affär
Leadnamn	Testfall 1
Ordernummer	1

Leadet är kopplat till uppdraget med uppdragsnummer 1.

- Det finns ett uppdrag i CRM-databasen med följande data:

Uppdragstyp	Löpande
Uppdragsnummer	1
Uppdragsnamn	Testfall 1

- Det finns en aktivitet i CRM-databasen med följande data:

Aktivitetstyp	Löpande
Konsult	Anders Andersson
Prislista	1
Från-datum	2006-01-01
Till-datum	2006-02-28

Aktiviteten är kopplad till leadet med ID 1.

- Det finns ytterligare en aktivitet i CRM-databasen med följande data:

Aktivitetstyp	Löpande
Konsult	Per Persson
Prislista	1
Från-datum	2006-01-01
Till-datum	2006-01-31

Aktiviteten är kopplad till leadet med ID 1.

- Det finns en prislista, prislista 1, i CRM-databasen med följande innehåll:

Tidkod	Pris
Normaltid	500
Beordrad övertid	600
Restid	400

- Följande tider finns rapporterade i TM-databasen:

Konsult	Tidkod	Uppdrags-nummer	Datum	Timmar	Aktivitet
Anders Andersson	Normaltid	1	2006-01-01	8	Löpande
Anders Andersson	Normaltid	1	2006-01-02	8	Löpande
Anders Andersson	Normaltid	1	2006-01-03	8	Löpande
Anders Andersson	Beordrad övertid	1	2006-01-03	4	Löpande
Anders Andersson	Normaltid	1	2006-01-05	8	Löpande
Anders Andersson	Restid	1	2006-01-05	2	Löpande
Anders Andersson	Normaltid	1	2006-01-09	8	Löpande
Anders Andersson	Normaltid	1	2006-01-10	8	Löpande
Anders Andersson	Normaltid	1	2006-01-11	8	Löpande
Anders Andersson	Restid	1	2006-01-11	2	Löpande
Per Persson	Normaltid	1	2006-01-02	8	Löpande
Per Persson	Beordrad övertid	1	2006-01-02	2	Löpande
Per Persson	Normaltid	1	2006-01-03	8	Löpande
Per Persson	Beordrad övertid	1	2006-01-03	2	Löpande
Per Persson	Normaltid	1	2006-01-04	8	Löpande
Per Persson	Normaltid	1	2006-01-05	8	Löpande
Per Persson	Normaltid	1	2006-01-09	8	Löpande
Per Persson	Beordrad övertid	1	2006-01-09	2	Löpande
Per Persson	Normaltid	1	2006-01-10	8	Löpande
Per Persson	Beordrad övertid	1	2006-01-10	4	Löpande

- Valt datum i applikationen är från och med 2006-01-01 till och med 2006-01-31.

Resultat:

Förutom korrekt lead- och uppdragsinformation ska applikationen visa följande:

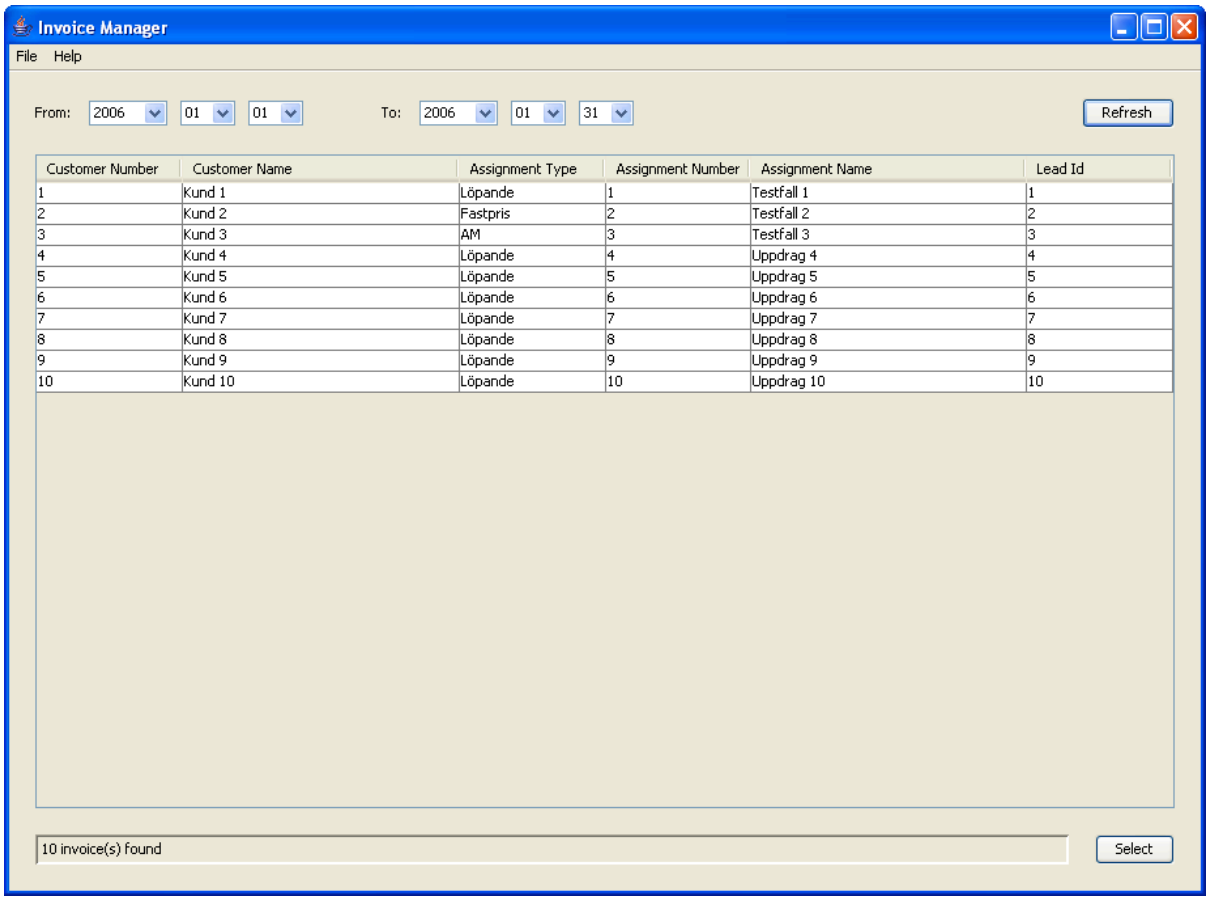
```
Activity: Löpande, Anders Andersson, 2006-01-01 - 2006-02-28, Prislista 1
  Beordrad övertid
    v01, 2006   4H, 600 = 2400
  Normaltid
    v52, 2005   8H, 500 = 4000
    v01, 2006   24H, 500 = 12000
    v02, 2006   24H, 500 = 12000
  Restid
    v01, 2006   2H, 400 = 800
    v02, 2006   2H, 400 = 800

Activity: Löpande, Per Persson, 2006-01-01 - 2006-01-31, Prislista 1
  Beordrad övertid
    v01, 2006   4H, 600 = 2400
    v02, 2006   6H, 600 = 3600
  Normaltid
    v01, 2006   32H, 500 = 16000
    v02, 2006   16H, 500 = 8000
```

6 Resultat

I detta kapitel presenterar jag resultatet av mitt arbete i form av beskrivningar och bilder av de användargränssnitt som ingår i faktureringssystemet, eller ”Invoice Manager”.

6.1 Sök fakturor

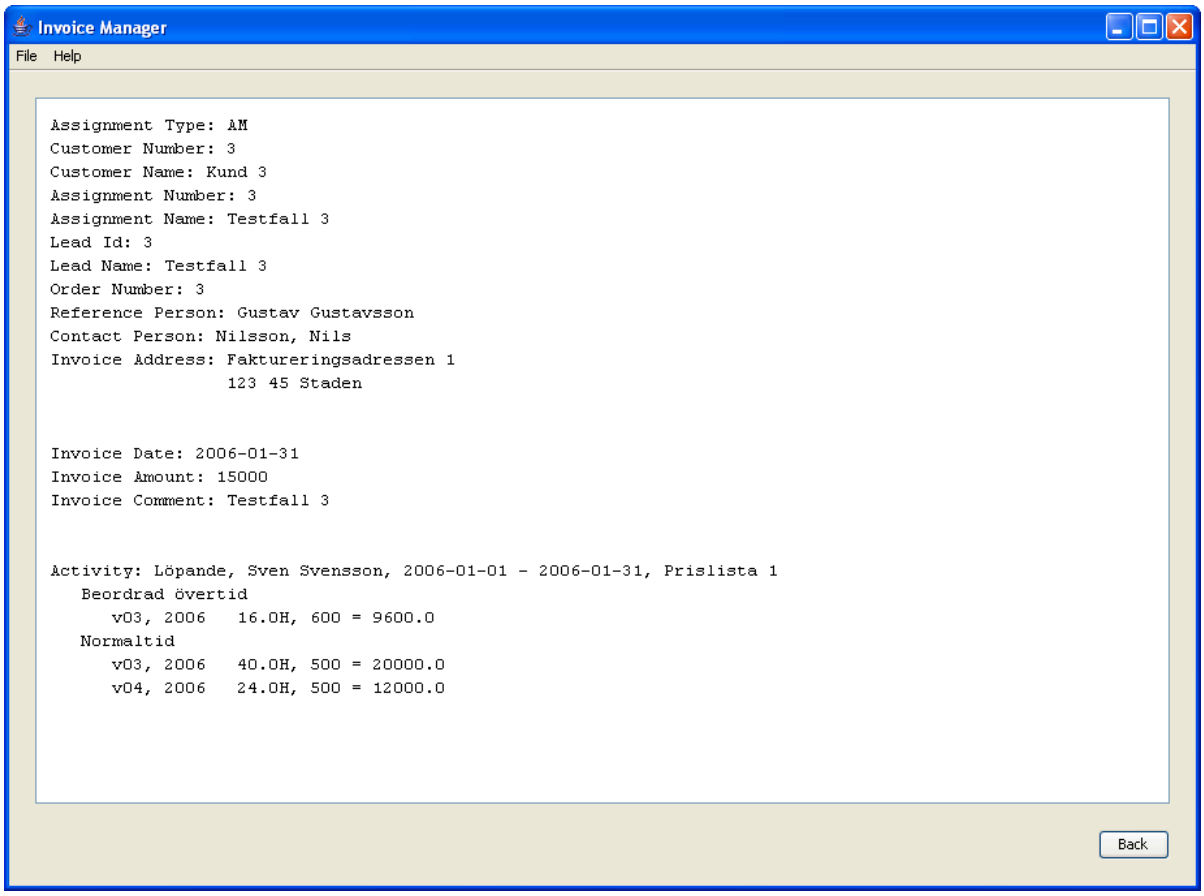
I figur 11 illustreras användargränssnittet för användningsfallet ”Sök fakturor”. Användaren (administrationen) väljer ett datumintervall överst i fönstret och trycker sedan på knappen ”Refresh”. Samtliga fakturor för perioden visas då i en tabell. Varje faktura presenteras som en rad i tabellen och anges med kundnummer, kundnamn, uppdragstyp, uppdragsnummer, uppdragsnamn, samt lead ID. Längst ner, i en statusrad, visas även hur många fakturor som hittades. Tabellen kan sorteras efter valfri rubrik genom att helt enkelt klicka på rubriken. Man kan sedan visa specifik information över en faktura genom att klicka på en rad och sedan trycka på knappen ”Select”.



Figur 11 Användargränssnitt för användningsfallet "Sök fakturor".

6.2 Visa faktura

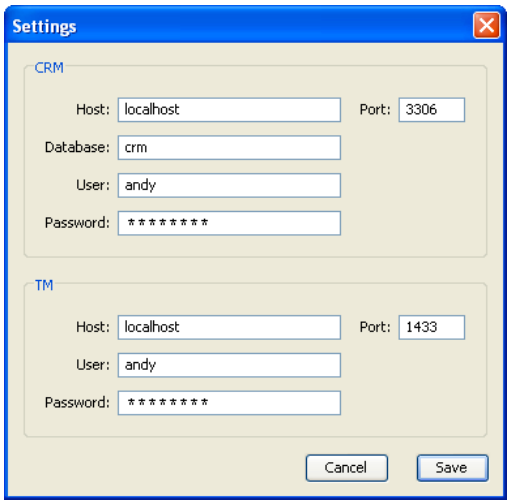
Om man har valt att visa en faktura, presenteras samtlig information över fakturan i fönstret. Hur det kan se ut visas i figur 12. Överst i fönstret visas information som är gemensam för alla fakturor, t.ex. uppdragstyp, uppdragsnummer, kundnummer, ordernummer och faktureringsadress. Efter det visas eventuella fakturaplaner som ligger inom det valda datumet (gäller uppdragstyp fastpris och AM). Sist visas de aktiviteter som innehåller rapporterade tider för det valda datumet (gäller löpande och AM). Tiderna delas in i tidkoder och under varje tidkod presenteras arbetad tid för varje vecka, vilket pris som gäller för tidkoden, samt summan. Finns det flera aktiviteter presenteras de efter varandra. Vill man återgå till överblicken över fakturorna, trycker man på knappen ”Back”.



Figur 12 Användargränssnitt för användningsfallet "Visa faktura".

6.3 Inställningar

Det är även nödvändigt att man kan ändra inställningar för databaserna i programmet. I figur 13 visas det användargränssnitt, en dialogruta, som används för att göra dessa inställningar. Här kan man bland annat ange var databaserna finns lokaliserade (via ip eller domännamn), samt ändra användare och lösenord för respektive databas. Dialogrutan öppnas genom att man i menyn väljer ”File” och sedan ”Settings”.



Figur 13 Användargränssnitt för inställningar.

7 Diskussion

I det här kapitlet berättar jag först lite om integrering av system, sedan beskriver jag de slutsatser jag har dragit och till sist ger jag förslag till fortsatt utveckling av programmet.

7.1 Integrering av system

En stor del av arbetet vid integrering av system går åt till att sätta sig in i de befintliga systemen, både ur ett användarperspektiv och ur ett utvecklingsperspektiv. Det är viktigt att man har en klar förståelse över hur de nuvarande systemen fungerar för att kunna genomföra en lyckad integrering.

Under utvecklandet av faktureringssystemet fick jag först och främst skaffa mig en förståelse över faktureringsprocessen på Create. Detta gjorde jag genom ett antal diskussioner med vice VD. Jag fick också sätta mig in i hur tidrapporterings- och CRM-systemets databaser var uppbyggda, då programmet skulle kommunicera direkt med databaserna. Det här gjorde jag genom att noga studera testdatabaser för respektive system (i brist på dokumentation), samt genom diskussion med en anställd på Create som varit delaktig i utvecklandet av CRM-systemet.

7.2 Slutsatser

Förutom att utveckla ett faktureringssystem som underlättar faktureringsprocessen för administrationen på Create in Lund AB, så skulle jag även efter genomförandet av arbetet kunna svara på frågan:

- Hur kan Unified Process och design patterns användas vid integrering av system?

Unified Process är en väldigt omfattande process, som innehåller väldigt många artefakter. På grund av tidsbrist fick jag därför välja ut och endast använda de artefakter som jag ansåg vara mest väsentliga, samt avgränsa arbetet i de olika aktiviteterna. Det uppkom inga argument under arbetets gång som talar för att Unified Process är lämpligare att använda än andra utvecklingsprocesser vid integrering av system. Snarare tyckte jag att det saknades aktiviteter och artefakter som passade just för integreringen. Det kändes också som att integrerings-arbetet var något avskilt ifrån själva utvecklandet av applikationen. Eftersom jag tvingades att välja bort artefakter, samt begränsa arbetet i de olika aktiviteterna, anser jag att Unified Process är en programvaruutvecklingsprocess som är lämplig att använda vid utveckling av stora eller mycket stora system. Vid utveckling av mindre system, som i mitt fall, bedömer jag att Unified Process istället är onödigt tidskrävande och att andra modeller, t.ex. vattenfalls-modellen, med fördel kan användas framför Unified Process.

Utav de 23 designmönster som jag studerade, fann jag fyra mönster som jag hade användning för i faktureringssystemet. Jag kunde inte heller här finna någon koppling mellan något särskilt designmönster och integreringen. Dock anser jag att designmönster med fördel kan användas vid utveckling av alla (objektorienterade) programvarusystem, stora som små, och att det är nyttig kunskap att ha med sig som programvaruutvecklare.

7.3 Fortsatt utveckling

Då utvecklingsarbetet i hög grad har varit tidsbegränsat, har jag tvingats att endast utveckla och implementera den mest grundläggande funktionaliteten i faktureringssystemet. Faktureringssystemet är alltså långt ifrån färdigutvecklat och får mer ses som en prototyp. Dock anser jag att applikationen, i sin nuvarande form, utgör en god grund för systemets fortsatta utveckling. Nedan följer några förslag till vidareutveckling, som bland annat uppkommit under diskussion, men som jag ej har hunnit med att implementera:

- Inloggningsfunktion – för att hindra obehöriga ifrån att använda systemet. Ett gränssnitt för inloggning, där användaren anger sitt användarnamn och lösenord, bör vara det första som dyker upp vid start av applikationen.
- Utskriftsfunktion – för att administrationen direkt ska kunna skriva ut fakturorna på papper från applikationen.
- Integritetstest – dvs. en funktion som kontrollerar att vissa tabeller och värden stämmer överens med varandra i de olika databaserna. Detta för att minska risken för felaktig data vid användning av systemet.
- Vidareutveckling av användargränssnittet, då, som jag tidigare nämnt, ingen fokus har lagts på det.

Referenser

[1] Sommerville, I. (2004), *Software Engineering 7th ed.*, Addison-Wesley, ISBN 0-321-21026-3.

[2] Jacobson, I., Booch, G. & Rumbaugh, J. (1999), *The Unified Software Development Process*, Addison-Wesley, Reading, Massachusetts, ISBN 0-201-57169-2.

[3] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, ISBN 0-201-63361-2.

[4] Cooper, J. W. (2000), *Java design patterns: a tutorial*, Addison-Wesley, Boston, ISBN 0-201-48539-7.

Övrig litteratur

Franzén, T. (1999), *Java från grunden*, Studentlitteratur, Lund, ISBN 91-44-01028-1.

Skansholm, J. (2002), *Java direkt med Swing*, Studentlitteratur, Lund, ISBN 91-44-02228-X.

Speegle, G. D. (2002), *JDBC: Practical Guide for Java Programmers*, Academic Press, USA, ISBN 1-55860-736-6.

Bilaga A: Testfall

Testfall 2:

Input:

- Det finns ett lead i CRM-databasen med följande data:

Lead ID	2
Leadstatus	Affär
Leadnamn	Testfall 2
Ordernummer	2

Leadet är kopplat till uppdraget med uppdragsnummer 2.

- Det finns ett uppdrag i CRM-databasen med följande data:

Uppdragstyp	Fastpris
Uppdragsnummer	2
Uppdragsnamn	Testfall 2

- Det finns en fakturaplan i CRM-databasen med följande data:

Datum	2006-01-31
Belopp	10000
Kommentar	Testfall 2

Fakturaplanen är kopplad till uppdraget med uppdragsnummer 2.

- Det finns ytterligare en fakturaplan i CRM-databasen med följande data:

Datum	2006-02-28
Belopp	20000
Kommentar	Testfall 2

Fakturaplanen är kopplad till uppdraget med uppdragsnummer 2.

- Valt datum i applikationen är från och med 2006-01-01 till och med 2006-02-28.

Resultat:

Förutom korrekt lead- och uppdragsinformation ska applikationen visa följande:

Invoice Date: 2006-01-31
Invoice Amount: 10000
Invoice Comment: Testfall 2

Invoice Date: 2006-02-28
Invoice Amount: 20000
Invoice Comment: Testfall 2

Testfall 3:

Input:

- Det finns ett lead i CRM-databasen med följande data:

Lead ID	3
Leadstatus	Affär
Leadnamn	Testfall 3
Ordernummer	3

Leadet är kopplat till uppdraget med uppdragsnummer 3.

- Det finns ett uppdrag i CRM-databasen med följande data:

Uppdragstyp	AM
Uppdragsnummer	3
Uppdragsnamn	Testfall 3

- Det finns en aktivitet i CRM-databasen med följande data:

Aktivitetstyp	Löpande
Konsult	Sven Svensson
Prislista	1
Från-datum	2006-01-01
Till-datum	2006-01-31

Aktiviteten är kopplad till leadet med ID 3.

- Det finns en prislista, prislista 1, i CRM-databasen med följande innehåll:

Tidkod	**Pris**
Normaltid	500
Beordrad övertid	600
Restid	400

- Följande tider finns rapporterade i TM-databasen:

Konsult	**Tidkod**	**Uppdrags-nummer**	**Datum**	**Timmar**	**Aktivitet**
Sven Svensson	Normaltid	3	2006-01-16	8	Löpande
Sven Svensson	Normaltid	3	2006-01-17	8	Löpande
Sven Svensson	Normaltid	3	2006-01-18	8	Löpande
Sven Svensson	Normaltid	3	2006-01-19	8	Löpande
Sven Svensson	Normaltid	3	2006-01-20	8	Löpande
Sven Svensson	Beordrad övertid	3	2006-01-21	8	Löpande
Sven Svensson	Beordrad övertid	3	2006-01-22	8	Löpande
Sven Svensson	Normaltid	3	2006-01-23	8	Löpande
Sven Svensson	Normaltid	3	2006-01-24	8	Löpande
Sven Svensson	Normaltid	3	2006-01-25	8	Löpande

- Det finns en fakturaplan i CRM-databasen med följande data:

Datum	2006-01-31
Belopp	15000
Kommentar	Testfall 3

Fakturaplanen är kopplad till uppdraget med uppdragsnummer 3.

- Valt datum i applikationen är från och med 2006-01-01 till och med 2006-01-31.

Resultat:

Förutom korrekt lead- och uppdragsinformation ska applikationen visa följande:

```
Invoice Date: 2006-01-31
Invoice Amount: 15000
Invoice Comment: Testfall 3


Activity: Löpande, Sven Svensson, 2006-01-01 - 2006-01-31, Prislista 1
  Beordrad övertid
    v03, 2006   16H, 600 = 9600
  Normaltid
    v03, 2006   40H, 500 = 20000
    v04, 2006   24H, 500 = 12000
```