**MITTUNIVERSITETET**

Institutionen för informationsteknologi och medier (ITM)

Handledare: Martin Kjellqvist, Lisa Sällvin Mittuniversitetet

Författarens e-postadress: [mavn1500@student.miun.se](mailto:mavn1500@student.miun.se)

Utbildningsprogram: DT156G Applikationsutveckling i Java 7.5 HP

Omfattning: 6289 ord inklusive bilagor

Datum: 2017-03-17

|  |
| --- |
| [Examensarbete/projektrapport inom Datateknik/Elektroteknik A/B/C/D, kurs, X poäng]  Antons Skafferi  Utveckling av ett digitalt system för restaurangen  Mattias Vängman |

[Plats för ev. illustration]

Sammanfattning

Skrivs i slutet av rapportskrivandet.

Sammanfattningen fungerar som en varudeklaration av rapporten. Den ska underlätta en snabb genomgång av dokumentet och därför utgöra ett koncentrat av rapporten i sin helhet, det vill säga rymma allt från syfte och metod till resultat och slutsats. Exempel: ”Målet med denna undersökning har varit att besvara frågan… . Undersökningen har genomförts med hjälp av…. Undersökningen har visat att…”. Nämn inget stoff som inte behandlas i rapporten. Sammanfattning skrivs i ett stycke. 200-250 ord är en rekommendation. Hänvisningar till rapportens text, källor eller bilagor är inte tillåtet, utan sammanfattningen ska ”stå på egna ben”. Undvik såväl formler och matematiska symboler som kursiv och fet stil. Sammanfattningen kan avslutas med en uppräkning av nyckelord, som kan underlätta sökande efter rapporten i biblioteksdatabaser. Exempel:

**Nyckelord**: Människa-dator-interaktion, XML, .Net, C#.

Abstract

Stryka?

Abstract, det vill säga motsvarande sammanfattning på engelska, krävs i examensrapporter. Abstract skrivs i ett stycke.

**Keywords:**  Exempel: Human-computer interaction, XML, .Net, C#.

Förord

Jag vill tacka vår lärare Martin dels för att han varit tillgänglig men även för sitt engagemang och hans hjälpsamma sätt då vi som grupp fastnat på olika problem som uppstått i projektet.

Jag vill också tacka vår lärare Lisa för att du lärt oss om agila arbetsmetoder och förklarat hur det kan se ut på en arbetsplats.

Innehållsförteckning

Sammanfattning ii

Abstract iii

Förord iv

Terminologi vii

1 Inledning 1

1.1 Bakgrund och problemmotivering 1

1.2 Övergripande syfte 2

1.3 Avgränsningar 3

1.4 Konkreta och verifierbara mål 4

1.5 Översikt 5

1.6 Författarens bidrag (Enskilt) 5

2 Teori 6

2.1 Definition av termer och förkortningar (Stryk denna på slutet) 7

2.1.1 Exempel på rubriknivå 3 (Stryk denna på slutet) 7

2.2 Att referera eller citera (Stryk denna på slutet) 7

2.3 Källförteckning och källhänvisningar (Stryk denna på slutet) 8

2.4 Automatiskt numrerade källhänvisningar (Stryk denna på slutet) 10

2.5 Illustrationer (Stryk denna på slutet) 11

3 Metod 12

3.1 Studiebesök 12

3.2 Facebook Messenger 12

3.3 Git versionshantering 13

3.4 Agil metodik 15

3.4.1 Scrum 15

3.4.2 Todoist 17

3.4.3 Parprogrammering 17

3.4.4 Refactoring 18

3.5 Netbeans 19

3.5.1 GlassFish server 19

3.5.2 JSF (Javaserver Faces) 19

3.5.3 RESTful web service 19

3.6 Android Studio 20

3.6.1 Retrofit 20

3.7 Användbarhetstester 21

4 Konstruktion / Lösningsalternativ 22

5 Resultat 24

6 Diskussion (Individuell) 25

Källförteckning 27

Bilaga A: Dokumentation av egenutvecklad programkod 28

Exempel på underrubrik 28

Bilaga B: Matematiska härledningar 29

Bilaga C: Användarhandledning 30

Bilaga D: Resultatsammanställning av enkätundersökning (Anvädartest, t.ex) 31

Terminologi

(Eng**. Terminology / Abbrevations / Acronyms / Notation**.)

En eventuell förteckning över termer, förkortningar och variabelnamn med korta förklaringar placeras efter innehållsförteckningen. Ett alternativ är att upprätta ett sakregister sist i rapporten, som anger var i den löpande texten respektive begrepp förklaras. Observera att man måste förklara begrepp och förkortningar första gången de används i den löpande texten, även om rapporten har ett terminologiavsnitt.

Förkortningar och akronymer

Java EE Java Enterprice Edition. En samling teknologier och API som är designad för Enterprice.

JSF Java Server Faces. Java specificerat användargränssnitt för webbapplikationer.

REST Representational State Transfer. Beskriver hur tjänster för maskin till maskin kommunikation kan tillhandahållas.

URI Uniform Resource Identifier. Klassificeras som lokaliserare, likt Uniform Resource Locator.

XML Extensible Markup Language. Universiellt och utbyggbart märkspråk.

JSON JavaScript Object Notation. Motsats xml.

HTML Hyper Text Markup Language.

API Application Programming Interface.

Inledning

Hjälpa restaurangägare: Hemsida, schemapp, digitalt order/beställningssystem.

I tidigare utbildning kan du huvudsakligen ha kommit i kontakt med relativt väldefinierade problemställningar som formulerats av lärare och läromedel. Under projektkurser och examensarbete måste du som student själv lägga ned en hel del tankearbete på att definiera vad som egentligen är uppgiften. Denna analys redovisar du i rapportens inledande kapitel. Genom att beskriva vilket problem eller problemområde som du har valt att studera och ditt motiv för detta, ger du dessutom läsaren en allmän introduktion till rapporten.

Inledning eller Introduktion (välj en av rubrikerna) utgör kapitel 1, och kan t.ex. rymma nedanstående underrubriker.

Det inledande kapitlet motsvarar innehållet i den *projektplan* som ska presenteras några veckor efter det att examensarbetet påbörjats. Projektplanen kan även innehålla en tidplan för arbetet, och omnämna några av de källor du har för avsikt att läsa och referera (i kapitel 2), samt några ord om vilken metod (se kapitel 3) du har valt för att angripa problemet med.

Försök att komma till ”kritan” så snart som möjligt. För att behålla intresset behöver läsaren få veta vad ditt arbete handlar om redan inom några få meningar. Tänk på att den som snabbt vill skapa sig en uppfattning om arbetet ofta läser endast rapportens sammanfattning, inledning samt slutsatser, eftersom dessa delar vanligen skrivs på en hög nivå, utan allt för tekniska och matematiska detaljer.

## Bakgrund och problemmotivering

Hur ser arbetsgången på restaurangen ut (innan vårt projekt)

Att det finns Lunch, Alalcarte, spelningar på fredagar ibland, Att kocken vill kunna lägga ut egna menyer o.s.v. Personalen schemalägga sig.

I princip ska vi beskriva det Anders berättade för oss på första/andra träffen. Vilka var hans önskemål.

Spara tid/pengar

Bättre ordning/koll

Bättre arbetsflöde

I detta underkapitel ska du snabbt försöka skapa intresse hos läsaren för det problemområde du har valt att undersöka. Visa även att du inte bara är insatt i ditt smala tekniska problem, utan att du har förståelse för det sammanhang där ditt problem dyker upp, att du kan betrakta det även ur ett icketekniskt perspektiv, och att du känner till den praktiska nyttan av den teknik du undersöker eller av den kunskap din studie förväntas ge upphov till.

Det är vanligt att den första meningen innehåller en visionär formulering eller historisk åter­blick. Tänk emellertid på att du inte kan veta säkert hur framtiden kommer att te sig, utan bör uttrycka dina visioner på ett nyanserat och sakligt sätt för att framstå som trovärdig.

Exempel: ”Mänskligheten har under historien gång … . Användandet av Internet och mobiltelefoner har vuxit explosionsartat sedan… . Nästa steg i utvecklingen förväntas bli … . Detta kan leda till problem med … Inom denna studie undersöks om problemet kan lösas med hjälp av … . Denna kunskap kan bli särskilt värdefull om några år med tanke på att allt fler människor …, och på att det finns en växande efterfrågan på marknaden efter… ”.

En teknisk rapport som skrivs på uppdrag av ett företag kan t.ex. inledas: ”Inom organisationen finns ett ökande behov av … och samtidigt växande problem med … . Vi har därför fått i uppdrag att genomföra en förstudie om … . En lösning på detta problem är angelägen därför att den kan leda till avsevärd minskning av kostnader för …, ökade marknadsandelar inom … samt en förbättrad arbetsmiljö. ”

## Övergripande syfte

Så väl fungerande appar/hemsida/funktioner som möjligt. Så nära ett verkligt fungerande system som möjligt.

Lokalt fungerande i labmiljö.

Tillämpa/lära oss metoder för projektarbetsformen (Agilt och SCRUM t.ex)

Versionshanteringssystem (t.ex. GIT)

Vi bör kanske nämna något om Kursen mål: <https://www.miun.se/utbildning/Kurser/Sok-kursplan/kursplan/?kursplanid=16873>

Bättre förståelse för Android OS

Projektets övergripande syfte är en visionär beskrivning av den riktning i vilken du vill arbeta, av vad du hoppas att projektet ska resultera i i det långa loppet, samt av projektets motiv. Syftesformuleringen kan vara på hög nivå, dvs den behöver inte vara klart avgränsad eller konkret. Det kan vara ett mål som du kanske aldrig kommer att uppnå, eller inte säkert kan veta när du har uppnått.

Nyckeln till framgångsrik forskning är ofta att man lyckas formulera en intressant frågeställning. Projektets syfte kan vara att besvara en sådan problemfomulering. Problemformuleringen bör diskuteras i rapportens avslutande slutsatskapitel.

Exempel: ”Projektets övergripande syfte är att ge upphov till förklaringar till varför … ”. ”Projektets syfte är att jämföra teknik A med teknik B som lösning på behov C.” ”Projektets syfte är att identifiera generella principer för sambandet mellan X och Y”. ”Projektet syftar till att ge upphov till nya tekniska lösningsförslag inom följande problemområde: … .” ”Syftet är att ge upphov till ny kunskap inom organisationen om... .” ”Projektet syftar till att utgöra ett beslutsunderlag för …”

## Avgränsningar

JavaEE, JSF, Android för Appar etc

Tiden

Exempel: ”Studien har fokus på … .” ”Undersökningen är avgränsad till utvärdering av fall F1 och F2 … . Undersökningens slutsatser bör emellertid vara generellt giltiga för alla… .” ”I undersökningen negligeras inverkan av Z, därför att … .”

## Konkreta och verifierbara mål

Ett fungerande system (i tillräcklig grad) Även användarvänligt/bra gränssnitt

En databas, två hemsidor, tre appar.

Prioriteringslista (Se bilaga……nuvarande utan färg)

Få en nöjd kund (Vad innebär detta konkret?)

Målformuleringen är en konkretisering av ovanstående syftesformulering. De frågor som specificeras ska besvaras av rapportens resultat, och i dess avslutande slutsatser. Målformuleringen ska vara så konkret att det i efterhand ska gå att avgöra om den har uppfyllts, och syftar till att utgöra stoppkriterium för när arbetet är slutfört. Specificera de objektiva numeriska resultat du söker. Du kan ange vad x- och y-axlarna eller kolumnerna ska visa i de diagram och tabeller du har för avsikt att ta fram.

Detta underkapitel skrivs vanligen efter det att du har genomfört teoristudien i kapitel 2, och revideras ofta under projektets gång. Det förekommer att den konkreta problemformuleringen placeras efter teoristudien, eftersom det annars kan vara svårt för läsaren att förstå de begrepp du använder. Nackdelen med en sådan disposition är emellertid att läsaren kan tappa intresset för ämnet, till följd av att det dröjer så länge innan du som författare kommer till kärnpunkten.

Exempel på problemformulering för en vetenskaplig rapport:

”Undersökningen har som mål att besvara följande frågor:

P1: Vilken betydelse har teknik A i jämförelse med teknik B för prestandamåttet Y vid olika värden på parameter X, för fall F1 och F2? För matematiska definitioner av X och Y, se kapitel 3, Modell.

P2: Vilken vinst ger … .”

Exempel på målformulering för en teknisk rapport:

“Undersökningens mål är att föreslå en lösning på följande tekniska problem: … . Undersökningen har vidare som mål att verifiera att lösningsförslaget uppfyller användbarhetskriterier, samt att utvärdera förslaget med avseende på prestandamått Y.”

Allt för tekniska detaljer senareläggs till konstruktionskapitlets tekniska kravspecifikation.

## Översikt

Beskriv kort rapportens disposition. Exempel: ”Kapitel 2 beskriver...”.

## Författarens bidrag (Enskilt)

Beskriv vilken del av arbetet som du själv har gjort, och vad du har fått hjälp med t.ex. av kollegor på företaget. Ange om du har redovisat någon del av arbetet under tidigare kurser eller examensarbeten. Utförs arbetet i grupp kan rapporten redovisa hur huvudansvaret för arbetets delar har fördelats mellan författarna. Givetvis ska alla medförfattare vara insatta i hela arbetet.

Teori

Vad kan labbarna bidra med här?

Beskrinving av Agila metoder, Scrum samt de teorier/metoder som beskrivs i lab 1-3.

XProgramming

Rapportens teoristudie, ibland kallat bakgrundsmaterial, ska innehålla fakta som krävs för läsarens förståelse för den fortsatta rapporten. Du sammanfattar här vad som tidigare är skrivet inom ditt område, t.ex. i uppslagsverk, vetenskapliga artiklar, kurslitteratur, tidskrifter, examensarbeten, dokument på webben, tekniska rapporter och standarder. Förklara pedagogiskt med konkreta exempel och många illustrationer. Skriv på en nivå så att någon med liknande utbildning som du kan förstå texten.

Visa att du har kännedom om sammanhanget och bakgrunden till ditt arbete, och inte bara om det arbete du själv har genomfört. Förklara gärna syftet med den teknik du beskriver, och inte bara hur tekniken fungerar. På D-nivå ska du visa att du har kännedom om forskningsfronten inom området, för att säkerställa att ditt arbete har ett visst nyhetsvärde. Men gå inte för långt ifrån ditt forskningsproblem. Ditt uppdrag är inte att skriva en lärobok som innehåller sådant som kan läsas på annat håll. Det är viktigt att hitta en lämplig balans mellan bakgrundsmaterial och dina egna resultat.

Rubriken kan gärna vara ett ämne, till exempel ”GSM-standarden” eller ”Forskningsläget inom område X”.

Om teoristudien är kort kan den i stället ingå i kapitlet Inledning.

Om din metod är att genomföra en kritisk litteraturstudie behövs normalt inte ett separat kapitel med bakgrundsmaterial, utan referaten av källorna sammanställs då i resultatkapitlet. Din kritik av källorna och dina argument för en egen uppfattning placeras i slutsatskapitlet.

## Definition av termer och förkortningar (Stryk denna på slutet)

Termer och förkortningar som är viktiga för läsarens förståelse av den fortsatta framställningen förklaras i detta kapitel. Första gången du i den löpande texten använder ett begrepp eller en förkortning ska du förklara det, även om det dessutom finns definierat i ett terminologiavsnitt. När begrepp introduceras skrivs de med *kursiv* stil.

Första gången en förkortning (förk.) används skrivs den inom parentes efter dess förklaring, såsom exemplifieras i denna mening.

Använd svenska termer så långt det är möjligt. Se svenska datatermgruppens rekommendationer på webbplatsen

<http://www.nada.kth.se/dataterm/>.

Du kan upprätta ett sakregister i slutet av rapporten, som anger på vilka sidor respektive begrepp definieras och diskuteras. För att lägga till ett begrepp i sakregistret markerar du begreppet och trycker på tangenterna Alt+Shift+X, eller väljer Infoga -> Referenser -> Index och tabeller -> Index ->Mark Index.

### Exempel på rubriknivå 3 (Stryk denna på slutet)

Undvik för många rubriknivåer.

## Att referera eller citera (Stryk denna på slutet)

Du *refererar* när du sammanfattar eller återger en text med egna ord.

Ex: Forsslund [3] förespråkar mer berättande rubriker i tekniska rapporter och menar att man särskilt i underrubrikerna kan ge viktig information.

Du *citerar* när du ordagrant återger en fras, en mening eller ett stycke. I normalfallet refererar man istället för att citera källor. Du kan använda direkta citat om du har speciella skäl, t.ex. om du vill återge vedertagna definitioner av begrepp, när du tycker att en författare formulerat sig på ett särskilt träffande sätt, när du behöver stöd av en auktoritet, eller när du vill visa att en författare har fel.

Korta citat omges med citationstecken. Att citera Strömqvist kan vara en passande illustration i detta sammanhang: "Det må vara svårt att skriva, men det är roligt också" [2].

Långa citat kan återges i form av *blockcitat*. Textmassan placeras då på sidan utan citationstecken, men med indrag, dvs något förskjutet åt höger, och med mindre teckenstorlek. Källan anges i direkt anslutning till citatet.

Det här är ett blockcitat vilket innebär indragning, mindre teckenstorlek, rak vänstermarginal, inte nödvändigtvis rak högermarginal, och inga citationstecken. Blockcitatet kan tillämpas vid mer än ungefär 50 ord. Blockcitat avslutas alltid med källhänvisning [2].

## Källförteckning och källhänvisningar (Stryk denna på slutet)

Att kopiera in en text utan att ange dess källa betraktas som plagiat och därmed allvarligt fusk.

En källförteckning (eng. references) upprättas i slutet av rapporten för att ge läsaren en samlad upplysning om samtliga källor som du refererar, citerar eller av annat skäl hänvisar till i den löpande texten. Källan ska anges så noggrant att läsaren ska kunna kontrollera den, om den finns tillgänglig via bibliotek eller på Internet. Det förekommer även att muntliga källor och annan korrespondens inkluderas i källförteckningen, men det är ovanligt i tekniska rapporter.

Använd vederhäftiga källor, gärna författade av auktoriteter på området. Privata hemsidor och studentuppsatser har låg tillförlitlighet som källor, i synnerhet om studentuppsatsen har lägre nivå (A, B, C eller D) än det egna arbetet. Var källkritisk, särskilt mot kommersiella försäljningsargument.

Ta endast med källor i förteckningen som du refererar eller citerar i den löpande texten. Samtliga källor som tas upp i källförteckningen ska vara kopplade till rapporten genom hänvisning i den löpande texten, enligt *Vancouver-systemet,* som är vanligt förekommande i rapporter i tekniska ämnen.

Enligt Vancouver-systemet ordnas källförteckningen i den ordning källorna återges i den löpande texten, och källhänvisningen anges i texten med en siffra inom hakparenteser, t.ex. [1] eller [2, 3]. De anges även i denna ordning i källförteckningen. Exempel på källhänvisning: Enligt Eriksson [2] kan dynamiska SFN ge betydande prestandavinster.

Exempel på post i källförteckning enligt Vancouver-systemet: (Observera att förteckningen inte placeras här, utan i slutet av dokumentet.)

[1] M. Eriksson, ”Dynamic Single Frequency Networks”, *IEEE Journal on Selected Areas in Communications (J-SAC)*, vol. 19, nr. 10, 2001, s. 1905-1914.

Eftersom information på webben kan revideras ofta, och eftersom webblänkar kan upphör att fungera, måste datum anges då du själv hämtade information från webbsidan. Vid webbaserade källor krävs ibland anvisningar för hur källan kan hittas. Tänk på att kvaliteten på materialet på Internet varierar.

Exempel på källförteckning:

[4] Post- och telestyrelsen (PTS), ”Ansluta trådlöst”,   
http://www.pts.se/internetsakerhet . Klicka på Webbkarta -> Ansluta trådlöst. Publicerad 2005-04-11. Hämtad 2005-05-20.

Ibland förekommer även att man använder det s.k. Harvard­systemet­. Då anges källhänvisningen i den löpande texten med efternamn, tryckår inom parentes, och sidhänvisning inom parentes, t.ex. (Andersson, 1996, s. 3), eller ”enligt Andersson (1996:3) …”. Källförteckning ordnas i bokstavsordning efter författarens efternamn, och kan vara uppdelad i följande underrubriker: Muntliga källor, Skriftliga källor, Webbaserade källor, Övriga källor.

Formatera källförteckningen på ett konsekvent sätt. I slutet av denna mall finns källförteckning i enlighet med Vancouver-systemet med exempel på hur du anger referenser till bokkapitel, vetenskapliga artiklar, artiklar i dagspressen, muntliga källor, rapporter, kataloger, manualer samt tekniska standarder.

## Automatiskt numrerade källhänvisningar (Stryk denna på slutet)

Du kan utnyttja ordbehandlingsprogrammet för att automatiskt numrera källförteckningen och källhänvisningarna enligt Vancouver-systemet. I Microsoft Word kan detta göras på två alternativa sätt. I slutet av mallen finns exempel på källförteckningar skapad med den första metoden.

Den första metoden innebär att du skriver in källans beskrivning i källförteckningen i slutet av dokumentet, i en automatiskt numrerad lista. Därefter markerar du var i den löpande texten källhänvisningen ska placeras, och väljer menyalternativet Infoga -> Korsreferenser -> Numrerat objekt. Exempel på en sådan källhänvisning: Se [19]. Fördelen med denna metod är att källförteckningen kan ha valfri ordningsföljd.

Det andra sättet är baserat på MS-Wordfunktionen Slutkommentarer, som är en form av fotnoter. Välj funktion Infoga -> Referenser -> Fotnoter -> Slutkommentarer. Om du vill hänvisa till samma referens vid flera tillfällen, måste du använda funktionen Infoga -> Korsreferenser enligt ovan. Om du utgår från en annan MS Word-mall än denna, måste du ändra slutkommentarssiffrans stilformat så att den inte blir upphöjd. Fördelen med denna metod är att källförteckningstexten (slutkommentaren) följer med om du kopierar det textavsnitt som innehåller källhänvisningsnumret till ett annat dokument. En annan fördel är att källförteckningen automatiskt får samma ordningsföljd som hänvisningarna, i enlighet med Vancouver-systemet. En tredje fördel är att om du håller musen över källhänvisningen visas slutkommentaren (dvs källförteckningens text) i ett popup-fönster. Problemet är att källförteckningen antingen kan placeras i slutet av dokumentet eller i slutet av en s.k. sektion - ingen annan möjlighet till placering finns. För att kunna placera källförteckningen innan första bilagan måste du se till att första t.o.m. sista kapitlet, tillsammans med källförteckningen, bildar ett enda avsnitt genom att radera den avsnittsbrytning som ligger strax innan källförteckningen. Notera att detta kan få effekter på utformningen av källförteckningens sidhuvud.

## Illustrationer (Stryk denna på slutet)

Samtliga illustrationer (bilder, figurer, diagram, tabeller) i rapporten ska vara numrerade och försedda med en kort figur- eller tabelltext. Därtill ska i anslutning till texten anges källhänvisning varifrån illustrationen är hämtad, om den inte är av egen produktion. Exempel:

  
Figur 2.1: Systemöversikt. (Källa: [1].)

Storage

Documents

Samtliga illustrationer ska vara kopplade till rapporten genom hänvisning i den löpande texten. Hänvisningarna skrivs på engelska med begynnande versal, och på svenska med begynnande gemen Exempel: ”Systemet illustreras av blockschemat i figur 2.1”, ”According to Figure 2.1 ...”, ”Av tabell 3.5 framgår …” etc. På svenska skrivs figur med litet f, medan Figure skrivs med stort F på engelska.

I ordbehandlingsprogrammet Microsoft Word kan du åstadkomma automatisk figur- och tabellnumrering genom att välja menyalternativet Infoga -> Referens -> Beskrivning. För automatisk hänvisning till en figur, välj Infoga -> Referens -> Korsreferens. Ange referentyp Figur. Figurtext förfigur 2.1och hänvisningar till figuren är skapade med hjälp av dessa funktioner.

Metod

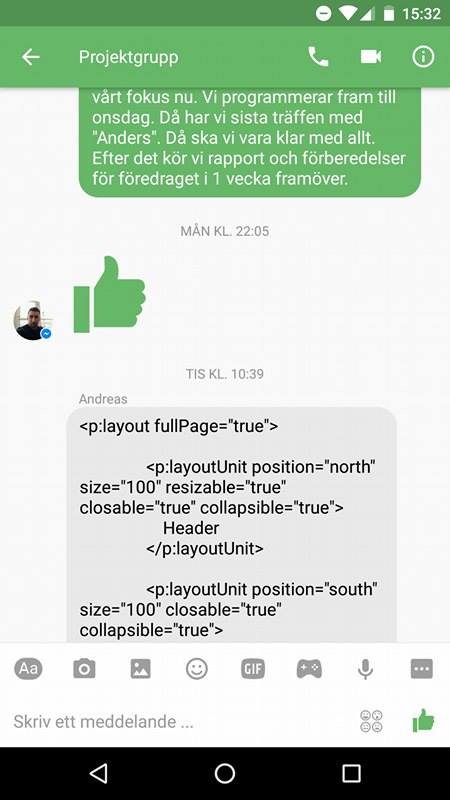
Kapitel 3 representerar metoder som har använts för att utföra detta projekt. Det börjar med studiebesök, därefter digitala och tekniska verktyg och slutligen användbarhetstester.

## Studiebesök

För att skapa en bild om hur ett digitalt system för en restaurang fungerar utfördes ett studiebesök på restaurangen Pinchos i Sundsvall. Pinchos restaurangkedja använder sig av ett digitalt system där kunden själv får lägga en beställning med hjälp av en mobilapplikation. När beställningen blivit lagd kommer instruktioner upp på en skärm i köket. Största skillnaden på Pinchos system och detta projekt är att beställningar sköts av en anställd på restaurangen. Därför var ett studiebesök på Pinchos ett stort bidrag till att projektets applikationer ligger så nära verkligheten som möjligt.

## Facebook Messenger

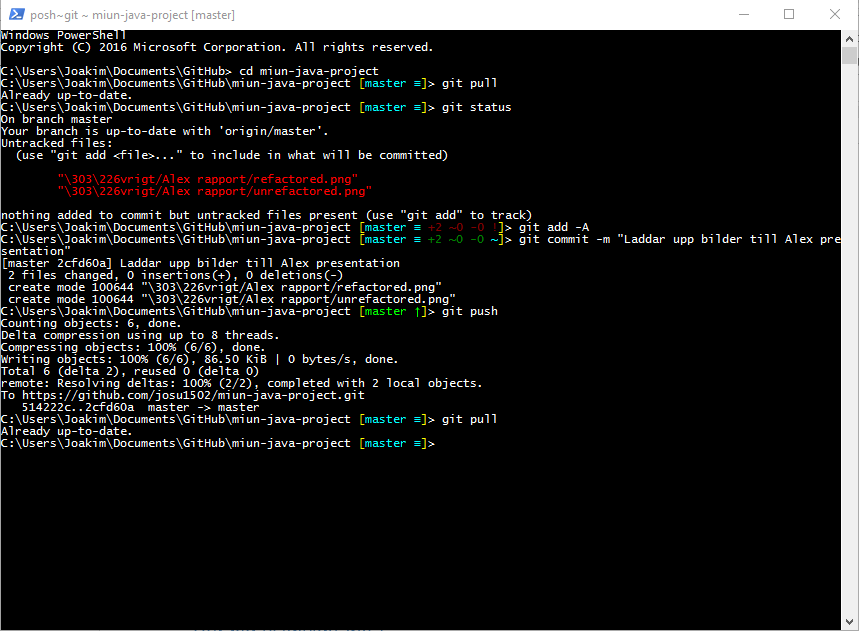
Till en början skapades en gruppchat i Facebook Messenger. Messenger användes för att få en strukturerad kommunikation inom projektgruppen och dela viktig information som till exempel tid och plats för projektmöten, dela programmeringskod, dela länkar till hjälpande webbsidor samt allmän diskussion kring projektet.



Figur x.x: Bilaga på kommunikation förd över messenger

## Git versionshantering

För att underlätta projektets arbetsflöde användes Git. Git är ett versionshanteringssystem som möjliggör hantering av olika versioner i ett stort projekt, som visar alla ändringar som är gjord i programkoden under arbetsflödets gång. Det möjliggör även att gå tillbaka versioner samt ångra ändringar om det skulle krävas. Git är världens mest använda versionshanterare. Det var därför ett enkelt val att använda.



Figur x.x: Skrämavbildning från Git shell.

Git använder sig av en central server för att centralisera projektet och göra det tillgängligt för flera användare, där varje individ som har tillgång till projektet kan ladda upp eller ladda hem ändringar. I detta fall användes Github som server. En fördel med Github är att det är gratis, men nackdelen är att projektet blir ”open source”, det vill säga att vem som helst har tillgång till källkoden.

Viktigast av allt är att Git skapar en behaglig arbetsmiljö för en projektgrupp att jobba med samma projekt. Git meddelar användaren om en kollision har uppstått, det vill säga att två personer har ändrat i samma kodrad.

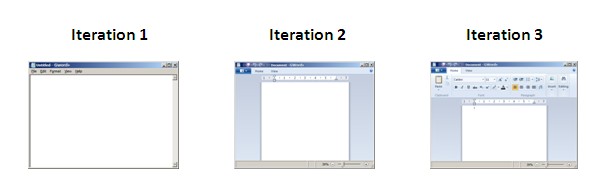
För ett så pass stort projekt med flera utvecklare är ett versionshanteringssystem, som Git, en nödvändighet. Git är anpassat för att fungera bra tillsammans med agila arbetsmetoder, se kapitel 3.4.

http://www.makeuseof.com/tag/git-version-control-youre-developer/

Nämn nått om Gitshell och vanliga kommandon.

## Agil metodik

Att jobba Agilt betyder att projektgruppen arbetar mot att ta fram en fungerande produkt i taget som sedan byggs på med fler funktioner. Iterationen upprepas tills ett slutresultat är nådd. Den vanligaste arbetsmetoden som inte är Agil är vattenfallsmodellen. Det innebär att produkten ska gå igenom bestämda steg tills den är klar, i slutet. Vattenfallsmodellen kommer ursprungligen från tillverkningsindustrin där det är svårt att göra förändringar i en färdig produkt. Detta gäller ej mjukvaruutveckling därför passar det bättre att jobba på ett Agilt arbetssätt då det gör det lätt att ändra på en färdig produkt.



Figur x.x: Förklarande bild för agil metodik.(Källa:[x])

<http://istqbexamcertification.com/what-is-agile-methodology-examples-when-to-use-it-advantages-and-disadvantages/>

För att få en bra struktur och en bra sammanhållning inom projektgruppen är den Agila arbetsmetoden en stor fördel.

Det finns ett flertal metoder att använda för att underlätta den Agila metodiken. För just programmering finns en metod som kallas för extreme programming. Till extreme programming tillhör parprogrammering och refactoring. Alla metoder som använts i detta projekt finns under kapitel 3.3.1-3.3.3.

### Scrum

Scrum är en metodik för systemutveckling som underlättar och möjliggör att projektet förändras på ett strukturerat sätt. I scrum används inte den typiska kravspecifikation utan istället används en backlogg som kontinuerligt uppdateras med önskemål. Detta innebär att önskemål som finns från början av projektet kan falla bort eller uppdateras med tidens gång. En backlogg är med andra ord en prioriteringslista med önskemål om hur den slutgiltiga produkten skall se ut.

När en projektgrupp använder sig av scrum finns det olika roller inom gruppen. Dessa roller har olika ansvarsområden och utföranden. Ett scrum team har en scrum master, scrum masterns jobb är att coacha teamet samt se till att arbetsflöde flyter på med bästa kapacitet. Inom scrum arbetar teamet själv och kommer gemensamt fram till vilka arbetsuppgifter var och en inom gruppen skall tilldelas. Scrum teamet jobbar mot en produktägare. Det sker ett gemensamt beslut mellan scrum team och produktägaren över vad backloggen skall innehålla. Produktägaren har mandat till att sätt en prioritering på denna backlogg.

Efter att en backlogg är färdigställd ställs det samman en sprint backlogg. Denna backlogg sätts samma av scrum teamet och dess scrum master. En sprint backlogg används för att dela upp projektet till mindre delar. Detta gör det lättare för utvecklarna att fokusera på konstruktionen av mindre delar inom ett projekt. En sprint sträcker sig vanligtvis över en två veckors period.

Under en sprint använder sig scrum teamat av något som heter daily scrum. Det är ett kort oftast ett stående möte som varar max femton minuter. På detta möte går man igenom vad som skedde igår, vad som skall ske den dagen och finns det några hinder.

Efter avslutad sprint hålls en sprintgenomgång där teamet visar projektets nuvarande status för produktägare, kunder och eventuellt andra intressenter.

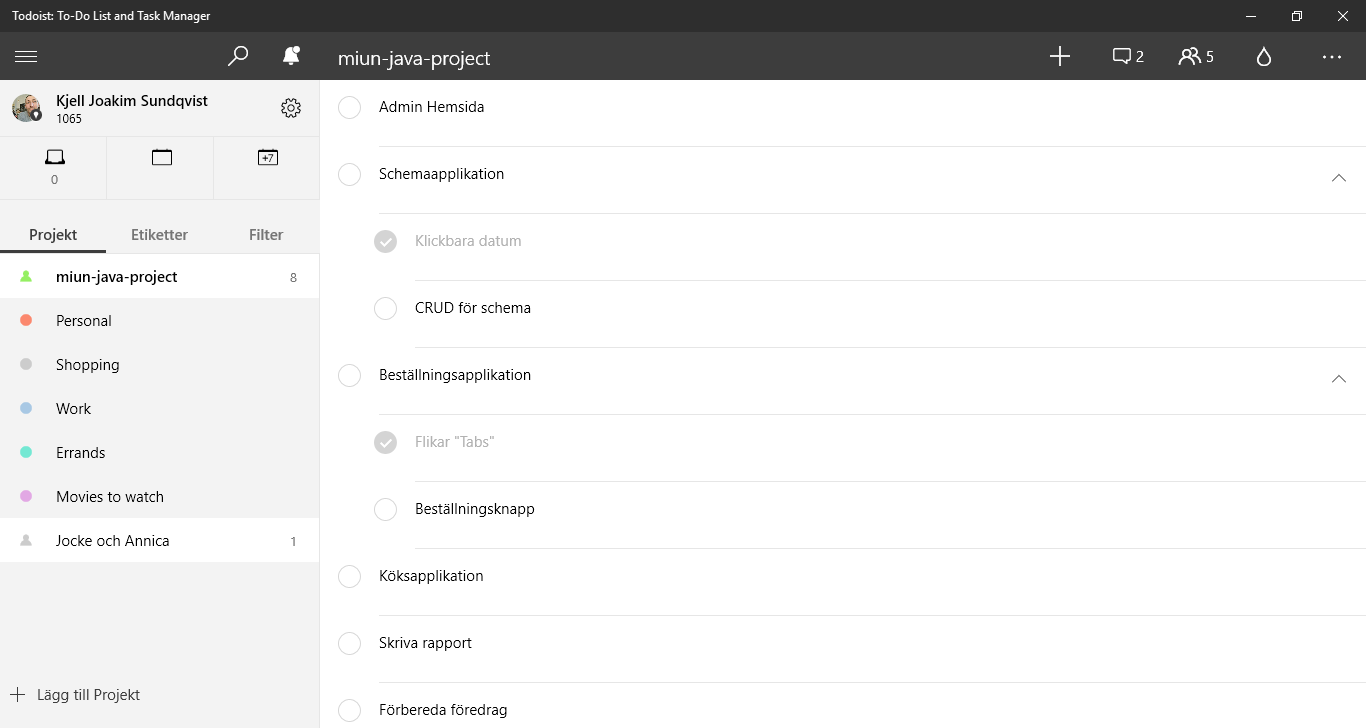
Vid en avslutad sprint har team, master och produktägaren en återblick över föregående sprint. Vid denna återblick analyseras sprinten för att se vad som kan förbättras inför kommande sprintar.

När återblicken blivit slutförd sker en sprint planning där produktägaren, scrum mastern och teamet går igenom backloggen igen för att se om det eventuellt justeringar. Därefter går scrum master och teamet igenom backloggen för att sammanställa en ny sprint backlogg.

<http://www.happiness.se/artiklar/vad-ar-scrum>

### Todoist

Todoist är ett verktyg som hjälper användare att strukturera upp arbetsuppgifter samt vad som finns att göra, till exempel inom ett projekt. Användare kan lägga till nya arbetsuppgifter och även bocka av avklarade uppgifter. Verktyget fungerar på Windows, Mac, Android, Ios, Windows phone och i webbläsaren. Inom projektet användes detta verktyg för att sammanställa en sprint backlogg som hela projektgruppen hade tillgång till, se kapitel 3.4.1.



Figur x.x: Skrämavbildning från projektet.

Detta verktyg fungerar bra tillsammans med sprintar i arbetsmetoden scrum. Se kapitel 3.3.1.

### Parprogrammering

Parprogrammering är ett smidigt och lärande sätt att programmera på. En person sitter vid tangentbordet och tar emot kommandon från någon som sitter bakom med en bättre överblick. Dessa två personer får då olika fokus. Den ena fokuserar på koden och programmets struktur och den andra på kod semantiken. Arbetsuppgifterna skiftar med jämna mellanrum och personerna byter arbetsroller. Vanligt är att en timer ställs in och det sker ett byte av arbetsuppgifter då klockan ringer. Under projektets gång tillämpades parprogramering främst då större problem uppstått eller då det började bli tidsbrist eftersom parprogrammering effektiviserar inlärningskurvan.



Figur x.x: Joakim och Mattias parprogrammerar.

### Refactoring

Huvudsyftet med refactoring är att göra den ursprungliga koden mer simpel. När borttagning av redundans, borttagning av oanvända funktioner och förnyelse av gammal kod utförs, kallas detta för refactoring. Med refactoring sparar man tid i det långa loppet och gör koden mer läsbar för nya användare. Resultatet utan refactoring är svårläst kod där nya programmerare inte vågar skriva om kod, med rädsla för att förstöra någon typ av funktionalitet. Vid användning av refactoring bör inte programmeraren vara rädd för förändring i funktionalitet eller utseende av kod.

(<http://www.extremeprogramming.org/rules/refactor.html> )

Refactoring fungerar bra i kombination med parprogrammering då två personer kan göra en refactor mer effektivt än bara en person.

<https://atlaz.io/blog/refactoring-in-extreme-programming/>

## Netbeans

Netbeans är i grund och botten ett fullt utvecklat integrerad utvecklingsmiljö för Java, men Netbeans stödjer även fler programmeringsspråk. Plattformen är ett open-source projekt och är gratis. Netbeans är den största utvecklingsmiljön för Java. Utvecklingsmiljön stödjer flertalet funktioner och tillägg som har använts inom detta projekt. Netbeans erbjuder ett stort bibliotek av färdiga mallar för webbapplikationer och gör utvecklingen av webbapplikationer lättsammare, speciellt för detta projekt. Alla funktioner och tillägg som använts i projektet finns inom kapitel 3.5.1-3.5.3.

### GlassFish server

GlassFish server är open-source och är en server som är kompatibel med Java EE. Servern är utvecklad av både Oracle och GlassFish community. Java EE referensimplementering är en del av GlassFish och det gör att de senaste versionerna av GlassFish oftast har stöd för de senaste funktionerna i Java EE. För tidiga användare av nya Java EE versioner är GlassFish ett utmärkt val. Då projektet ska vara utvecklat i Java ställer det ett krav på att servern kan klara av Java EE och därför passar GlassFish väldigt bra som val. Detta projekt har använt sig av GlassFish server 4.0.

https://glassfish.java.net/roadmap.html

### JSF

JSF är ett standard Java-ramverk för att bygga webbapplikationer i Java. Det förenklar utvecklingen genom att tillhandahålla ett komponentcentrerat användargränssnitt. Ett krav är att servern som webbapplikationen kommer byggas i klarar av Java EE. Som tidigare nämnt klarar GlassFish server av detta.

http://www.oracle.com/technetwork/topics/index-090910.html

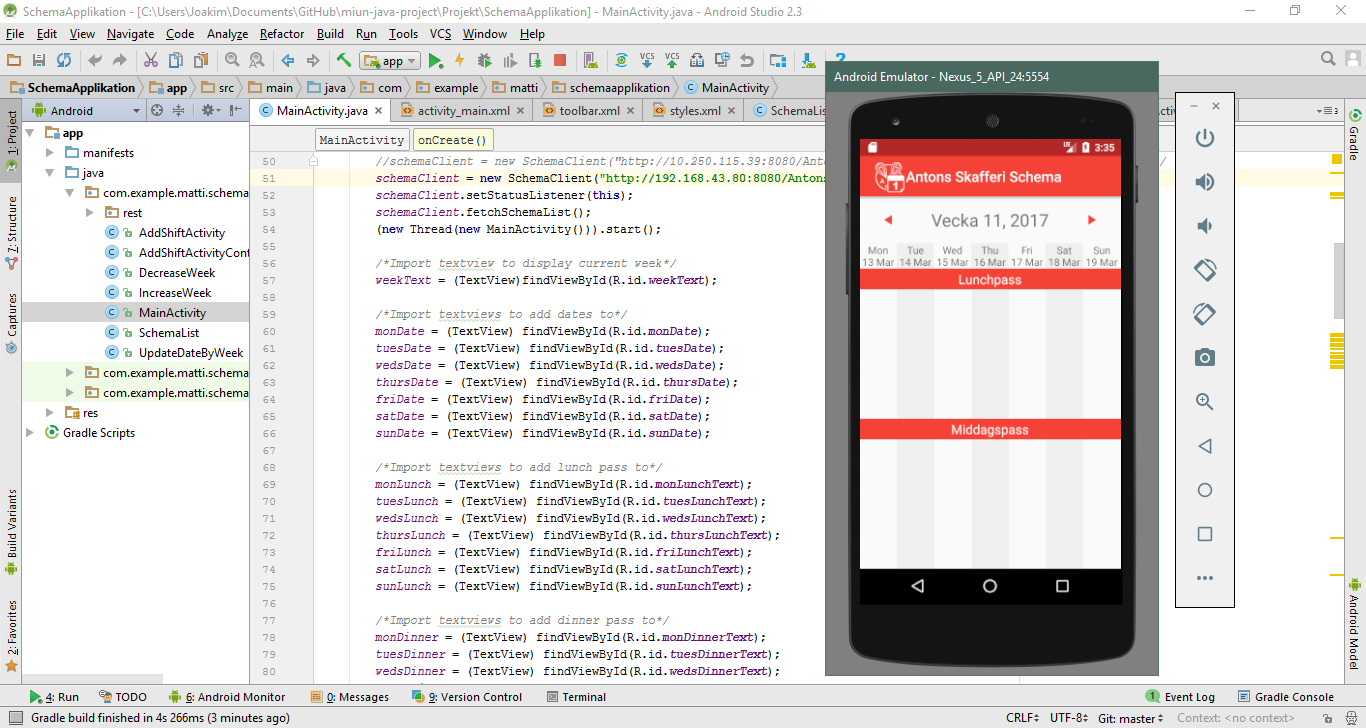
### RESTful web service

RESTful web service sätter upp en URI som identifierar en webbresurs. Webbresurserna manipuleras av kommandon som specificeras över internet. Webbresursen kan representeras i olika filformat, till exempel XML, JSON, HTML med flera. Endast fyra metoder är tillgängliga i ett RESTful web service. Vilket är GET, PUT, POST och DELETE. Detta krävs för att få en koppling mellan databasen och en Android applikation. För att förenkla denna process i Android applikationerna används Retrofit som går att läsa om i kapitel 3.6.1.

https://netbeans.org/kb/docs/websvc/intro-ws.html

## Android Studio

Android Studio är en integrerad utvecklingsmiljö som används för att utveckla Android applikationer. Utvecklingsmiljön använder sig av ett flexibelt Gradle-baserat byggsystem och erbjuder även testkörning via inbyggda emulatorer. Detta förenklar utveckling av Android applikationer avsevärt. Där av passade det in väldigt bra att använda i detta projekt.



Figur x.x: Skrämavbildning från Android Studio.

https://developer.android.com/studio/intro/index.html

### Retrofit

Retrofit är ett REST-baserat API för Android Studio som används för att koppla upp sig mot en databas med Android applikationer. Retrofit är kopplat till ett xml-dokument som är genererat av RESTful web services i Netbeans. Retrofit API ger tillgång till fyra kommandon för att sköta alla transaktioner till och från en databas. Kommandona är GET, POST, PUT och DELETE, som är vanliga kommandon i ett REST-baserat API. GET hämtar data från databasen, PUT uppdaterar data i databasen, POST lägger till ny data i databasen, DELETE tar bort data från databasen. Detta möjliggör simpel kod jämfört med egenskriven kod och är en stor del i projektet.

## Användbarhetstester

Syftet med användbarhetstester är att låta representiva användare utföra olika uppgifter då utvecklarna observerar hur utförandet av dessa uppgifter sker. Dessa observationer blir sedan sammanställda till ett underlag som används för att vidareutveckla produkten i fråga. På grund av att projektet bestod av två webbsidor och tre mobilapplikationer var användbarhetstester passande att utföra.

Konstruktion / Lösningsalternativ

Protyper: Dummies av hemsidan (mockups) av de olika apparna hemsidorna.

Hur vi gjprde för CRUD till hemsidan från början, och sedan hur vi utvecklade det.

Exempel på frera versioner av schemaappen.

Nämna olika metoder som vi funderade på och vilka val vi gjorde när det gäler de olika apparna/hemsidan.

Retrofit/skapa eget API

Vi valde XHTML ist. För HTML för attdet var lättare att integrera Java då.

Konstruktionsavsnitt ingår ofta i tekniska rapporter, men inte alltid i vetenskapliga rapporter. Här genomför du din analys av problemställningen och formulerar en teknisk kravspecifikation. Här beskriver du de viktigaste principerna i de lösningsalternativ som du föreslår, utformar och senare i rapporten kommer att utvärdera. Beskrivningen placeras ibland före, men oftast efter metod- /modellkapitlet.

Tänk på att läsaren sällan är intresserad av alltför detaljerad dokumentation av datorprogramkod, algoritmer, kretsscheman, användarhandledning, med mera. Sådana detaljer placeras med fördel i bilagor, om de över huvud taget inkluderas.

Under din tidigare universitetsutbildning har du i huvudsak fått små uppgifter som vanligen har tagit några minuter eller timmar att lösa. Ett examensarbete eller en projektkurs kan ibland kännas som en oöverstiglig uppgift därför att den är så omfattande, om du inte vet i vilken ände du ska börja. Ett sätt att underlätta stora projekt är att använda *top-down-metoden*, d.v.s. successivt dela upp problemet eller konstruktionen i allt mindre delproblem eller delsystem, och ange kravspecifikation, problemanalys och lösningsförslag för var och en av delarna. Till slut har du identifierat små och konkreta uppgifter av liknande karaktär som du har mött under din tidigare utbildning.

Det är emellertid inte alltid praktiskt möjligt att arbeta enligt top-down-metoden, eftersom problemet kan vara för komplext och du inte ser hela problembilden från början. Många gånger måste du kanske växla mellan top-down- och *bottom-up-metoden*. Den senare innebär att du utgår från delar som du redan har, samt från enkla problem som du redan vet hur du ska angripa, och som du förväntar sig att du kommer att ha nytta av senare under projektet. Du utökar successivt dessa delar till allt större system och problem, och strävar efter att gå i riktning mot projektets mål.

Top-down-metoden har fördelen att den ger rapporten en god struktur, vilket underlättar för läsaren. Dokumentationen följer därför ofta top-down-metoden. Du kan således dela upp konstruktionsdelen i flera kapitel, och ge dem namn efter delproblem och delsystem, t.ex. ”Kravspecifikation”, "Algoritmer", "Användargränssnitt", "Programdokumentation", "Prototyp" och "Implementering".

Resultat

Bilder på färdiga appar

Vad vi lyckats med

Resultat från användbarhetstester

Resultatkapitlet ingår när du har genomfört en systematisk undersökning, t.ex. en utvärdering av ett datorprogram som du har utvecklat, vilket krävs inom examensarbeten på C- och D-nivå. I resultatkapitlet redovisas objektiva resultat av en empirisk undersökning, t.ex. en utvärdering av ett datorprogram som du har utvecklat. Tänk på att eventuella kommentarer i detta kapitel endast får vara av förtydligande art. Dina egna synpunkter och subjektiva (personliga) kommentarer hör hemma i kapitlet Slutsatser/Analys/Diskussion.

Sträva efter att redovisa resultaten, till exempel enkät-, test-, mät-, beräknings- och/eller simuleringsresultat, så överskådligt och lättbegripligt som möjligt. Resultaten presenteras med fördel i diagram- eller tabellform. Redovisning av intervjuer kan bestå av sammanfattningar, eventuellt kompletterade med några konkreta exempel.

Omfattande resultat, till exempel fullständiga sammanställningar av enkätresultat, stora tabeller och långa matematiska härledningar, placeras med fördel i bilagor.

Diskussion (Individuell)

Vad vi inte lyckats med och varför.

Svårighet i att utveckla när man från början inte har kunskaper som krävs för att göra jobbet (Java EE, GIT, Agilt etc.)

Vad man lärt sig

Vad man kunnat göra bättre

Vad man själv gjort under projektet, mer konkret och utförligt.

Vad har funkat bra/dåligt

Hur var det att jobba i grupp/projekt.

Önskemål: t.ex: genomgång på viktiga moment i kursen som CRUD för Android eller hanteringen av bilder till hemsidan.

Personliga motgångar

Efter de objektiva resultaten följer kapitlet Slutsatser/Analys/Diskussion (välj en rubrik), där du presenter dina egna slutsatser, din subjektiva uppfattning, samt kritiskt analyserar resultatens tillförlitlighet och generaliserbarhet.

Om denna del är omfattande kan den indelas i flera kapitel eller underkapitel, t.ex. ett analys- eller diskussionskapitel med förklaringar till och kritisk granskning av resultaten, ett slutsatskapitel där de viktigaste resultaten och slutsatserna presenteras, samt ett avsnitt med förslag på fortsatt arbete inom området.

Att återknyta till undersökningens syftes- och målformulering hör till det viktigaste i detta kapitel.

Ge gärna utrymme åt svaren på följande frågor: Vad är projektets nyhetsvärde och viktigaste bidrag till forskningen eller teknikutvecklingen? Har projektets mål uppnåtts? Har uppdraget utförts? Vad är svaret på den inledande problemformuleringen? Har resultatet blivit det väntade? Är slutsatserna generella, eller gäller de bara under vissa förutsättningar? Vilken betydelse har metod- och modellvalet för resultaten? Har nya frågor väckts på grund av resultatet?

Den sista frågan inbjuder till möjligheten att ge förslag till andra, anknytande undersökningar, d.v.s. förslag dels till åtgärder och rekommendationer, dels till fortsatt forskning eller utveckling för den som vill bygga vidare på ditt arbete.

I tekniska rapporter på uppdrag av företag presenterar du här den rekommenderade lösningen på ett problem, och kan göra en konsekvensanalys av lösningen ur tekniskt såväl som icke-tekniskt perspektiv, till exempel i fråga om ekonomi, miljö och förändrade arbetsrutiner. Kapitlet innehåller då rekommenderade åtgärder samt förslag på vidare utveckling eller forskning, och utgör således beslutsunderlag för uppdragsgivaren.

Källförteckning

Här följer exempel på hur en källförteckning kan utformas enligt Vancouver-systemet. Den är automatiserad enligt metoden numrerad lista och korsreferenser, som beskrivs i kapitel 2.4.

1. M. Eriksson, ”Dynamic Single Frequency Networks”, *IEEE Journal on Selected Areas in Communications (J-SAC)*, vol. 19, nr. 10, 2001, s. 1905-1914. (Exempel på referens till artikel i vetenskaplig tidskrift)
2. S. Strömquist, *Skrivboken*. 5 uppl. Malmö: Gleerups., 2005 (Exempel på referens till bok)
3. L. Forsslund, ”Rapportering av forskningsresultat - ett rationaliseringsobjekt”. *Industriell teknik*, 22, 1969, s. 361-363. (Exempel på referens till artikel i tidskrift)
4. Post- och telestyrelsen (PTS), ”Ansluta trådlöst”. Se   
   www.pts.se/internetsakerhet . Klicka på ”Webbkarta” och ”Ansluta trådlöst”. Publicerad 2005-04-11. Hämtad 2005-05-20. (Exempel på referens till webbsida)
5. N. Bie, ”Minspel på Internet”, *Dagens Nyheter* 7 juli, 1997, sidan 5. (Exempel på referens till artikel i dagstidning)
6. E. Eriksson, ”Inledning till kunskap”, Fiktiva högskolan, rapport nr XYZ-102, 2003, 120 sidor. (Exempel på referens till annan rapport)
7. AB Benzlers, ”Växelboken. Kuggväxlar, kuggremmar, kilremmar”, Mecman Hydraulik, SKF. Huvudkatalog. (Exempel på referens till katalog)
8. Microsoft Corporation, 1992: ”Microsoft Word Användarhandbok. Ordbehandlingsprogram för Macintosh Version 5.0.” (Exempel på referens till manual)
9. Svenska Datatermgruppen, ”Information om datatermer”,  
   http://www.nada.kth.se/dataterm/  
   Hämtad 1998-08-20 (Exempel på referens till webbsida)

Bilaga A: Uppgifterna för användbarhetstestet

Användartester

Uppgifter/frågor om den publika hemsidan:

* Vilken gatuadress har restaurangen?
* Vilka öppettider gäller på restaurangen?
* Om du skulle äta middag på restaurangen vilka rätter skulle du då välja?
* Vilken lunch serveras på torsdag?
* Är det några band som ska spela på restaurangen?

Uppgifter/frågor om administrationshemsidan:

* Lägg till en ny lunchrätt på valfri dag
* Ändra i en befintlig middag
* Lägg till en kommande spelning på restaurangen
* Ta bort en kommande spelning på restaurangen

Uppgifter/frågor om schemaapplikationen:

* Boka in dig på tre olika pass (tex två lunchpass och ett middagspass)
* Du blir sjuk och kan inte jobba ett av passen du bokade in dig på. En arbetskamrat lovade att ta över ditt pass, hur gör du då?
* Om du var anställd på restaurangen, hur lång tid skulle det ta att lära sig applikationen?

Uppgifter/frågor om Beställnings- och köksapplikationen:

* Lägg en beställning (Beställningsapplikationen)
* Du har börjat lagt en beställning och kommer på att du skrev fel bordsnummer, hur ändrar du? (Beställningsapplikationen)
* Vad gör du när en maträtt är färdiglagad? (Köksapplikationen)
* Dagen är slut och du vill rensa bort historiken, vad gör du då? (Köksapplikationen)
* Om du var anställd på restaurangen, hur lång tid skulle det ta att lära sig applikationerna?

Bilaga B: Prioriteringslista

Prioriteringslista Anders

Anders önskemål om funktioner/tjänster med prioritering (1-10), prioritet 1 = högst prioritet.

**Publika hemsidan**

|  |  |  |
| --- | --- | --- |
| Nr. | Prioritet | Funktion/Tjänst |
| 1 | 1 | Websida (publik) (fixa scrollen) |
| 2 | 8 | Meny på websida |
| 3 | 5 | Info om kommande spelningar på websida |
| 4 | 6 | Bildgalleri från tidigare spelningar, på websida |
| 5 | 1 | Övrig info på websida, t.ex. kontakt, vägbeskrivning, öppettider etc. |

**Hemsida, admin**

|  |  |  |
| --- | --- | --- |
| Nr. | Prioritet | Funktion/Tjänst |
| 6 | 1 | Websida (för Anders admin) Ändra meny lunch (helfärdig) |
| 7 | 8 | Websida (för Anders admin) Ändra meny à la carte (Worddokument) |
| 8 | 5 | Websida (för Anders admin) Ändra info om kommande spelningar |
| 9 | 6 | Websida (för Anders admin) Ändra i bildgalleri från tidigare spelningar |
| 10 | 9 | Websida (för Anders admin) Ändra övrig info |

**Applikationer, övrigt**

|  |  |  |
| --- | --- | --- |
| Nr. | Prioritet | Funktion/Tjänst |
| 11 | 2 | Schema (app)Enkel Prototyp med layotförslag. |
| 12 | 5 | Schema (app)Avancerad |
| 13 | 3 | App för att ta upp beställningar |
| 14 | 4 | App för att visa info om beställningar i köket (kocken) |
| 15 | 7 | Sammanställning för notan |

Grönmarkerad text betyder att funktion/tjänst är färdigutvecklad.

Rödmarkerad text betyder att funktion/tjänst **inte** är färdigutvecklad.

Bilaga C: Laboration 1

**Laboration 1 – Java Code Conventions**

**---------------------------------------------------**

**Författad av:** *Gabriel Afram, Thomas Astner, Alex Darborg, Andreas Edin, Adam Hjernquist, Joakim Sundqvist* och *Mattias Vängman*

**Inledning**

Kod konventioner är bra att kunna eftersom det underlättar en programmerare att förstå andras kod då det blir mer lättläst. Java använder två filtyper, java source samt java bytecode. Suffixen för dessa filtyper är .java och .class. Rekommenderade namn för att bygga makefiler är GNU make för att bygga vårt program. En textfil med README är också bra att ha med för att ge instruktioner till nya projekt arbetare. Filer som är större än 2000 rader anses vara besvärliga och skall undvikas.

**Java Source Files**

Varje java source fil innehåller antingen en singel publik klass eller ett interface. När privata klasser och interfaces är associerade med en publik klass så kan man lägga dem i samma source fil som den publika klassen ligger i.

**Java source file har följande ordning:**

* Kommentarer
* package java.awt;

**Paketen kan se ut såhär:**

import java.applet.Applet;  
import java.awt.\*;  
import java.net.\*;

* Klass och interface deklarationer

**Klass & Interface struktur:**

* Klass eller interface dokumentation kommentarer.
* Klass eller interface påstående.
* Klass eller interface implementations kommentarer.
* Klass (static) variabler, publika först sedan skyddade och sist privata variabler.
* Instans variabler, publika först sedan skyddade och sist privata variabler.
* Konstruktorer.
* Metoder.

När klassvariabler deklareras, börjar man med dem publika variablerna och sedan dem skyddade och sist dem privata.

**Indentering/Indrag**

Fyra mellanslag borde användas som enhets mått för ett indrag. Undvik rader som är längre än 80 karaktärer då det kan ställa till problem för terminaler och verktyg. Om du får längre än 80 karaktärer bör du bryta upp den i mindre bitar.

**Grundprinciper för att bryta upp en rad:**

* Bryt efter ett kommatecken
* Bryt före en operator
* Föredrar higher-level breaks före lower-level breaks
* Justera den nya raden precis under uttrycket från den föregående raden.
* Om ovanstående grundprinciper leder till förvirrande kod, då underlättas det med 8 mellanslag.

Ett Java program kan ha två olika typer av kommentarer: implementationskommentarer och dokumentationskommentarer.

/\*....\*/ eller // använder man för implementations kommentarer medans /\*\*...\*/ används för dokumentations kommentarer.

Dokumentationskommentarer som kallas "doc comments" finns bara i Java. Implementationskommentarer är menade till att kommentera koden eller för att kommentera en specifik implementation. Dokumentskommentarer är menade att beskriva specifikationen av koden från ett implementations-fritt perspektiv, det ska kunna läsas av utvecklare som inte nödvändigtvis har koden framför sig. Kommentarer bör användas till att ge en översikt av koden och bara innehålla information som är relevant till att läsa och förstå programmet. Undvik att kommentera för mycket, saker som är uppenbara i koden behöver inte kommenteras, det gör bara att koden blir svårläst, försök hellre att göra koden mer lättläst än att lägga in kommentarer till allt. Det finns fyra olika stilar av implementations kommentarer som ett program kan ha: block, single-line, trailing och end-of-line.

**Block kommentarer:** Används för att ge en beskrivning av filer, metoder, data strukturer och algoritmer. De borde användas i början av varje fil och förre varje metod, de kan även användas inne i en metod men den ska då vara indenterad på samma nivå som koden den beskriver. Exempel på en block kommentar:

/\*  
 \* Här är en block kommentar  
 \*/

**Single-line kommentarer:** Kan förekomma på en enkel rad som är indenterad till samma nivå som koden som följer. Om inte kommentaren kan skrivas på samma rad borde den följa block kommentar formatet. Exempel på en single-line kommentar:

If (condition) {  
 /\* Handle the condition. \*/  
 ....  
}

**Trailing kommentarer:** Väldigt korta kommentarer som förekommer på samma rad som koden den beskriver ska hålla ett tillräckligt långt avstånd till koden. Är det fler korta kommentarer i samma stycke av kod så ska de vara indenterade till samma nivå. Exempel på en trailing kommentar:

if(a == 2) {  
 return true; /\* special case \*/  
} else {  
 return isprime(a); /\* works only for odd a \*/  
}

**End-OF-Line kommentarer:** När man använder sig av // så kommer kommentaren att fortsätta enda tills det blir en nyrad. Exempel på End-Of-Line kommentarer:

if(foo >1) {  
 // Do a double-flip  
 ...  
} else {  
 return false; //Explain why here  
}

**Dokumentations kommentarer:** beskriver Java klasser, gränssnitt, konstruktorer, metoder och fält. Varje dokumentations kommentar borde förekomma precis innan deklarationen. Exempel:

/\*\*  
\* The Example class provides ..  
\*/  
class Example { ...  
  
Notera att klasser och gränssnitt inte är indenterade medan deras medlemmar är det. Första raden (/\*\*) för klasser och gränssnitt är inte indenterad, raderna nedanför är indenterade så det hamnar i linje med \*-tecknet. Medlemmar, inkluderat konstruktorer har fyra blanksteg på första raden och fem på de andra. En dokumentationskommentar borde inte vara positionerad inne i en metod eller konstruktor, för Java associerar dokumentations kommentarer med den första deklarationen efter kommentaren.

En deklaration per rad rekommenderas då det fungerar bättre om man vill göra en kommentar för en specifik deklaration. Aldrig ska både en variabel och en metod deklareras på samma rad. Inte heller bör olika typer av variabler deklareras på samma rad. Det går bra att använda indentering mellan variabeltyp, variabelnamn och kommentar. Se exempel nedan:

int level; // indentation level  
int size; // size of table  
Object currentEntry; // currently selected table entry

Lägg alla deklarationer i början av ett block. Vänta inte med deklarationen tills den ska användas. Detta kan försvåra läsbarheten hos en annan användare. Det enda undantaget är i “for”-iterationer, där räknaren kan deklareras i metoden. Undvik att deklarera en ny variabel med samma namn i ett inre block. Om det är möjligt bör variabler initieras direkt vid deklarationen. Det vill säga ge den ett värde när den skapas. Vid kodning av klasser och interface bör det inte vara något mellanrum mellan metodnamn och första parentesen “(”, första måsvingen förekommer på samma rad och sista måsvingen som avslutar metoden på en egen rad. Metoder separeras med ett blankt radbyte.

Varje rad bör innehålla max ett påstående. Varje returvärde för en metod bör vara parantesfri.

Alla ”if”, ”else if” och ”else” -satser bör ha denna form:

if (*condition*) {  
 *statements*;  
} else if (condition) {  
 *statements*;  
} else {  
 *statements*;  
}

Alla ”if”-satser ska ha måsvingar, även om den bara innehåller ett påstående.

En for-loop skall innehålla initiering, tillstånd, uppdatering och påstående/påståenden. En tom for-loop bör innehålla initiering, tillstånd och uppdatering. Vid användning av kommaoperatorn i initiering eller uppdatering av en klausul undvik att använda mer än tre variabler. om det behövs använd separata påståenden innan for-loopen påbörjats. En while-loop skall innehålla tillstånd och påstående en while-loop deklareras på följande sätt:

while (condition) {  
 statements;  
}

Vid användning av while-loop utan statement deklareras den på detta sätt.

while (condition);

Do-while loop är lik en while loop enbart att den är omvänd och statementet kommer innan deklarationen av while-loopen. Do-while loopen deklareras på följade sätt.

do {  
 statements;  
} while (condition);

Implementation av switch statements ser ut på följande sätt

switch (condition) {  
case ABC:  
 statements;  
 /\* falls through \*/  
case DEF:  
 statements;  
 break;  
case XYZ:  
 statements;  
 break;  
default:  
 statements;  
 break;  
}

Varje switch statement bör innehålla ett default case. Varje case som inte innehåller en break skall innehålla en kommentar (/\*Falls through\*/) för att visa att den faller igenom. Se exempel ovan.

try-catch statement deklareras på följande sätt.

try {  
 statements;  
} catch (ExceptionClass e) {  
 statements;  
}

Blankrader förbättrar läsbarheten av koden, det blir som styckesindelning av koden. Man delar upp koden i stycken för den kod som är relaterad till varandra. Två blankrader skall alltid användas under följande omständigheter, mellan delar av en källfil och mellan klass och gränssnittsdefinitioner. En blankrad bör alltid användas under följande omständigheter, mellan metoder, mellan lokala variabler i en metod och metodens första statement. Samt för en kommentar innan ett block eller på ensam rad. En blankrad bör implementeras mellan logiska sektioner inuti en metod för att förbättra läsbarheten. Ett nyckelord följt av en parentes bör separeras med ett mellanslag. Ett mellanslag bör ej användas mellan en metods namn och dess öppningsparetens. Detta bidrar till att särskilja sökord från metodanrop. Ett mellanslag bör förekomma efter ett komma i argumentlistor. Alla binära operatorer utom operatorn (.) bör skiljas från sina operander med ett mellanslag. Unära aktörer som increment och decrement bör ej skiljas från dess operand.

**Konventioner för namngivning**

För att göra programkod lätt att läsa och förstå finns det konventioner (överenskommelser) som bör följas. Om detta görs blir det lätt för läsaren av koden att avgöra om en identifierare är en variabel, klass eller en metod t.ex. Nedan följer en kortfattad beskrivning av hur konventionen för respektive identifierare ser ut.

**Klass:** Namnet utgörs av ett substantiv. Namnet börjar alltid med stor bokstav. Om namnet är sammansatt av flera ord börjar varje delord med stor bokstav. Inga blanksteg används. Några exempel:

class Hus

class SommarStuga

Interface: Namnges på samma sätt som klasser.

**Metod:** Namnet utgörs av ett verb. Om namnet är sammansatt av flera ord är första ordet ett verb. Namnet inleds med små bokstäver, sedan inleds varje delord i namnet med stor bokstav. Några exempel:

rulla();

getNumberOfRooms();

**Variabel:** Namnet inleds med liten bokstav. Om namnet består av flera ord inleds varje delord med stor bokstav. Variabelnamn bör vara korta och meningsfulla. Namnet bör vara utformat så att det är lätt att komma ihåg. I allmänhet undviks variabelnamn som består av endast en bokstav, förutom för kortlivade variabler, t.ex. i en for-loop. Bokstäverna i, j, k, m och n betecknar vanligtvis heltal och d, d, e betecknar vanligtvis tecken (characters). Några exempel:

int i;

string firstName;

**Konstant:** Namnet skrivs med stora bokstäver. Om namnet består av flera ord separeras de av understreck. Ett exempel:

Int MAX\_ANTAL\_RUM = 8;

**Tillgänglighet och referens till variabler**

Variabler och instanser bör inte göras publika om det inte särskilt krävs.

När man refererar till ett objekt är det viktigt att skilja på klassvariabler (statiska) och på instansierade objekt. Ex:

klassMetod(); //OK

EnKlass.klassMetod(); //OK

ettObjekt.klassMetod(); //EJ OK!

**Tilldelning av variabler**

Undvik att tilldela flera variabler samma värde i ett kommando, det blir svårläsligt. Ex:

firstName = secondName = ”no name”; // FEL

Även inbäddade tilldelningar bör undvikas, ex:

a = ((b = c + d)-e); // FEL

bör skrivas:

b = c + d;

a = b – e; // RÄTT

Bilaga D: Laboration 2

**Laboration 2 – Grundläggande begrepp och principer**

**---------------------------------------**

**Författad av:** *Gabriel Afram, Thomas Astner, Alex Darborg, Andreas Edin, Adam Hjernquist, Joakim Sundqvist* och *Mattias Vängman*

**Kravspecifikation (SRS)**

En kravspecifikation är en beskrivning av det mjukvarusystem som utvecklas, vilket arbetas fram av både kund och entreprenör samt fastställer att båda parter är överens om hur slutprodukten ska se ut. Det planlägger funktionella- och icke-funktionella krav samt fastställer vilka typer av användarinteraktioner som mjukvaran måste stödja.

Språket i ett SRS-dokument är skrivet på ett sådant sätt att kunden ska kunna förstå vad som står utan vidare teknisk expertis, vilket innefattar både text och analytiska modeller såsom UML (ex. Use Case Diagram) och ER-Diagram.

Kravspecifikationen låter entreprenören estimera det ungefärliga priset och hur lång tid ett projekt kommer att kräva samtidigt som det fungerar som ett hanteringssystem för vad kunder har att förvänta sig. Om kundens krav förändras under utvecklingsprocessen kan priset och tiden som krävs att kunna beräknas på nytt utifrån dem nya förutsättningarna.

**Kravfunktionalitet**

Ett **funktionellt krav** är vad som definierar ett systems funktion eller funktionen av en komponent i systemet; hur det krav som kunden ställer i kravspecifikationen faktiskt fungerar – vad det **gör**. Dessa kan exempelvis beskrivas med hjälp av Use Case diagram och/eller textuella beskrivningar. Ett **icke-funktionellt krav** används i konjunktion med funktionella krav för att beskriva hur systemet i stort ska fungera, såsom prestanda-, underhålls-, migrations- och användbarhetskrav.

Ett funktionellt krav definierar vad ett system ska göra medan ett icke-funktionellt krav definierar hur systemet ska vara. Ett funktionellt krav utrycks som specifika funktioner såsom exempelvis ”systemet skall beräkna [någonting]” eller ”systemet ska hämta information från [databas]”. Ett icke-funktionellt krav kan utryckas som ”systemet skall vara säkert” eller ”systemet skall vara stabilt”, osv, i en mer ospecificerad ton.

**Prototyp**

En prototyp inom mjukvaru-utveckling är en ofärdig version av ett system under utveckling. Prototypen används under utvecklingens gång som test-system för att försöka visualisera det färdiga systemet/projektet och förbättra kommunikation mellan deltagare i arbetet genom att ge en gemensam bild av slutprodukten samt vart i arbetsprocessen man befinner sig.

**Single Responsibly Principle**

En funktion, klass eller modul ska bara förändras/modifieras på grund av **en** (och endast **en**) anledning. Denna princip används för att underlätta underhåll och organisering av system samt snabba upp/bibehålla utvecklingstakten.

Anta att ett program läser in en fil och gör matematiska kalkyler baserade på informationen i filen. Här kan exempelvis två händelser ge anledning till att källfilerna måste redigeras; att filformatet förändras eller att den matematiska kalkylen är ineffektiv/gammal. Dessa områden förändras alltså på grund av helt olika anledningar, d.v.s. input och prestanda. Dessa två förändringar ska enligt **SRP** vara två enskilda ansvarsområden (responsibilities) och skall därför skrivas i skilda klasser/moduler för att underlätta underhåll (uppdatering) av systemet.

**Open Closed Principle**

Funktioner, klasser eller moduler ska vara öppna för utvidgning/utbyggnad **utan att** ursprungskoden behöver skrivas om. Denna kraftfulla princip är direkt relaterad till objektbaserad programmering och **arv** (inheritance) där vi ska kunna bygga subklasser till redan skrivna klasser utan att huvudklassen behöver modifieras. Denna princip är även direkt relaterad till SRP eftersom OCP underlättar underhåll och vidare utveckling eftersom vi behöver göra färre förändringar vid uppdatering.

**Liskov Substitution Principle**

Om X är en subtyp av Y så måste instanser av objektet X ersätta objekt av typen Y – utan att förändra någon utav dem önskvärda egenskaperna hos Y. LSP handlar om valet mellan att skriva en subklass eller använda en annan strategi för att uppnå den önskvärda funktionaliteten. Med andra ord så kan vi alltså inte i programkod säga att en boll är en punkterad boll, en fyrkant är en rektangel eller att ett fordon är ett flygplan, utan tvärtom (en rektangel är en fyrkant).



Bilaga E: Laboration 3

**Laboration 3 – Designmönster**

**---------------------------------------**

**Författad av:** *Gabriel Afram, Thomas Astner, Alex Darborg, Andreas Edin, Adam Hjernquist, Joakim Sundqvist* och *Mattias Vängman*

**Factory Pattern**

**Avsikt**

Factory-mönstret ”producerar” objekt med ett dynamiskt urval av olika subklasser. Ett interface definierar **hur** ett objekt skapas men subklasser till interfacet avgör vilken klass som ska användas för varje enskild instans.

**Motivation**

Låt oss föreställa oss ett scenario där vi har ett system som använder sig av abstrakta klasser som definierar och upprätthåller relationer mellan objekt. Vi har även en typ av ansvarsområde för att skapa dessa objekt. Detta system vet alltså bara (under exekvering) **när** ett nytt objekt ska skapas men inte exakt av vilken **typ**. Factory-mönstret kan då implementeras via en s.k. Factory-metod där ett interface definieras för att skapa objekt men låter subklasser stå för typ-skillnaderna.

**Tillämplighet**

Factory-mönstret är användbart när en klass inte kan förutsäga vilken typ av objekt den måste skapa.

**Struktur**

**Airplane**

**Car**

**MainClass**

**+create() : void**

**+create() : void**

**+main() : void**

**<<interface>>**

**ToyFactory**

**Toy**

**+getToy() : Toy**

**+create() : void**

**Deltagare**

* **MainClass;** den klass som hanterar alla instanser av Toy.
* **ToyFactory;** fungerar som själva factory-metoden.
* **Toy;** använder ett gemensamt interface med dem olika Toy-typerna Car och Airplane.

**Samarbete**

ToyFactoryDemo använder ToyFactory-klassen (Factory-metoden) för att skapa och ge tillbaka instanser av Toy-objekt. Den information som ToyFactory får av ToyFactoryDemo används utav Toy och dem olika Toy-typerna för att ge tillbaka korrekt typ av Toy-objekt, d.v.s. Car eller Airplane.

**Konsekvenser**

Det kan vara besvärligt att implementera Factory-mönstret i ett redan fungerande system; många fundamentala moduler kan behöva skrivas om. Fabriken måste användas på ett sådant sätt att objekten tillhör samma familj. Om dem inte tillhör samma basklass/interface så kan inte Factory-mönstret användas.

**Implementering**

ToyFactory kan vara abstrakt och därför kräva att implementationen definieras i subklasser. Om ToyFactory är konkret så kan implementeringen kan vi istället flexibelt åsidosätta default-metoderna och använda den implementation som används i subklasserna.

**Kodexempel (Java)**

public interface Toy {

void create();

}

public class ToyCar implements Toy {

@Override

public void create() {

//Do stuff

}

}

public class ToyAirplane implements Toy {

@Override

public void create() {

// Do stuff

}

}

public class ToyFactory {

public Toy getToy(String type) {

if (type == "CAR") {

return new ToyCar();

} else if (type == "AIRPLANE") {

return new ToyAirplane();

}

}

}

// A main class to use the factory pattern

**Relaterade mönster**

* **Abstract Factory**, vilket ofta använder implementationer av factory-metoder.

**Model-View-Controller**

**Avsikt**

MVC används för att separera ett systems ansvarsområden, framför allt från användargränssnittet. Dessa ansvarsområden delas upp i Model, View och Controller.

**Motivation**

I en mjukvaruarkitektur där användare använder ett interface och hanterar en samling data så är det praktiskt att kunna separera dessa ansvarsområden för att underlätta underhåll och utveckling.

**Tillämplighet**

Ett MVC-system implementeras i tre fundamentala steg; en Model som definierar vad informationen representerar, en View som beskriver hur informationen ska representeras för användaren (exempelvis ett användargränssnitt) och en Controller som bearbetar och besvarar händelser samt hanterar ändringar i Model och View.

**Struktur**

Controller

+setSID(int) : void

+getSID() : int

+print(Model, View) : void

+ setSID()

View

Model

+setSID(int) : void

+getSID() : int

+print(Model, View) : void

**Deltagare & sammarbete**

* **Model**; själva dataobjektet där informationen finns.
* **View**; det gränssnittsmedium som användaren utnyttjar.
* **Controller**; den mekanism som hanterar instanser av Model och view.

**Konsekvenser**

Istället för att koncentrera sig på att använda den aktuella informationen inuti Model så ligger MVC’s fokus främst på att presentera data, vilket inte alltid är optimalt ur ett prestandaperspektiv.

**Kodexempel**

public class Model {

private int SID;

public Model(int SID) {

this.SID = SID;

}

public int getSID() {

return SID;

}

public void setSID(int SID) {

this.SID = SID;

}

}

public class View {

public void print(int SID) {

System.out.println("The SID is " + SID);

}

}

public class Controller {

private final Model model;

private final View view;

public Controller(Model model, View view) {

this.model = model;

this.view = view;

}

public void setSID (int SID) { model.setSID(SID); }

public int getSID () { model.getSID(); }

public void print() { view.print(model.getSID()); }

}

public class MVCDemo {

public static void main(String[] args) {

int SID = 3;

Model model = new Model(SID);

View view = new View();

Controller controller = new Controller(model, view);

controller.print(); // -> OUTPUT: “The SID is 3”

SID++;

controller.setSID(SID);

controller.print(); // -> OUTPUT: “The SID is 4”

}

}

**Relaterade mönster**

* **Observer Pattern**; används ofta tillsammans med MCV för att hålla View- och Controller-modulerna synkroniserade.

**Iterator pattern**

**Avsikt**

Att i en sekvens kunna nå en samling objekt utan att veta någonting om den underliggande representationen.

**Motivation**

I dem fall vi behöver skapa egna typer av listor, arrays, osv, så behöver vi även en metod som kan ge oss tillgång till objekten utan att känna till den interna strukturen och hur dem internt är representerade; vi är bara intresserade av den aktuella informationen. Därför kan vi lägga allt itererande ansvar (åtkomst och placering) i ett iterator-objekt.

**Tillämplighet**

I situationer där flera olika typer av objektsamlingar håller information och vi behöver ett mönster som ger oss åtkomst parallellt så är det användbart att implementera ett gemensamt interface för iterator-mönstret som sedan kan användas av objekt av liknande typer som ShoppingCart. Det är då enkelt att tillämpa detta mönster på nya typer av samlingar/Collection’s.

**Struktur**

**<<interfaces>>**

**Iterator**

+next() : object

**Collection**

+createIterator() : Iterator

+hasNext() : bool

**ShoppingCart**

+main() : void

**IteratorDemo**

**CartIterator**

-products : String[]

-pos : int

+CartIterator() : Iterator

+createIterator() : Iterator

**Deltagare & sammarbete**

* **Iterator**; det första interfaces för hur vi söker igenom objektstrukturen.
* **CartIterator**; håller position i varje iterering och ger tillbaka Iterator-objektet till ShoppingCart.
* **Collection**; skapar och håller instanser av Iterator.
* **ShoppingCart**; det objekt som kapslar in hela instansen av det som ska sökas igenom.

**Konsekvenser**

Potentiella problem kan upstå i flertrådade miljöer när Collection hämtar Iterator-objekt från olika trådar om designen inte är trådsäker. Det innebär ytterliggare komplexitet och svårigheter under implementering.

**Implementering**

Mönstret kan flexibelt implementeras med hjälp av två interface så att flera olika samlingstyper kan användas samtidigt.

**Kodexempel**

public interface Iterator {

public boolean hasNext();

public Object Next();

}

public interface Collection {

public Iterator createIterator();

}

public class ShoppingCart implements Collection {

private final String products[] = {"Apples", "Cake", "GPU"};

@Override

public Iterator createIterator() {

return new CartIterator();

}

private class CartIterator implements Iterator {

private int pos;

@Override

public boolean hasNext() {

if (pos < products.length) {

return true;

}

return false;

}

@Override

public Object Next() {

if (this.hasNext()) {

return products[pos++];

}

return null;

}

}

}

**Relaterade mönster**

* **Composite Pattern**; iteratorer används ofta rekursivt för att färdas igenom sammansatta strukturer.

**Command Pattern**

**Avsikt**

Tanken med Command-mönstret är att kunna skapa kommandon som är insvept/inkapslat i ett objekt för att bl.a. kunna exekvera åtgärden på ett kontrollerat sätt. På så vis kan uppgifter/requests utföras vid ett specifikt tillfälle men också tas bort eller dupliceras.

**Motivation**

En kund vill på en restaurang kunna för-, varm- och efterrätt på en och samma gång. Kunden ska samtidigt kunna ångra sig innan beställningen läggs. Om kunden exempelvis skulle vilja göra det via en dator-applikation så behöver dessa kommandon kunna utföras på ett strukturerat sätt. Med hjälp av Command-mönstret så kan varje beställning motsvara ett kommando där varje kommando placeras i en lista som senare skall utföras, samtidigt som denna lista kan modifieras under tiden.

Receiver

OrderMeal

**Struktur**

CommandPatternDemo

+buy() : void

+sell() : void

+execute() : void

+main() : void

Sell

Invoker

Buy

-que : ArrayList<OrderMeal>

-meal : Receiver

+execute() : void

+Sell() : void

+execute() : void

+Buy() : void

+execute() : void

+Buy() : void

-meal : Receiver

**Deltagare och samarbete**

* **OrderMeal**; ett interface för körning av kommandon.
* **Receiver**; definierar hur operationerna utförs.
* **Invoker**; säger åt kommandot att utföra sitt ärende. Detta objekt håller även listan med dem kommandon som ska utföras.
* **Buy & Sell**; implementerar OrderMeal och håller en Reciever.

**Kodexempel**

public interface OrderMeal {

public void execute();

}

public class Receiver {

public void buy() {

System.out.println("A product was bought");

}

public void sell() {

System.out.println("A product was sold");

}

}

public class Invoker {

private final ArrayList<OrderMeal> que = new ArrayList();

void newOrder(OrderMeal ordermeal) {

que.add(ordermeal);

}

public void executeCommands() {

que.forEach((orderMeal) -> {

orderMeal.execute();

});

}

}

public class Buy implements OrderMeal{

private final Receiver meal;

@Override

public void execute() { meal.buy(); }

public Buy(Receiver meal) { this.meal = meal; }

}

public class Sell implements OrderMeal {

private Receiver meal;

@Override

public void execute() { meal.sell(); }

public Sell(Receiver meal) { this.meal = meal; }

}

public class CommandPatternDemo {

public static void main(String[] args) {

Receiver r = new Receiver();

Buy WhatToBuy = new Buy(r);

Sell WhatToSell = new Sell(r);

Invoker i = new Invoker();

i.newOrder(WhatToBuy);

i.newOrder(WhatToSell);

// execute the list of commands

i.executeCommands();

}

}

**Relaterade mönster**

* **Composite Pattern**; kan användas tillsammans med Command för att gruppera existerande kommandon för att skapa macros.

**Singleton Pattern**

**Avsikt**

Singleton-mönstret förhindrar att flera än en instans av ett objekt kan skapas.

**Motivation**

I situationer där flera moduler i ett system vill kunna hantera och modifiera en gemensam resurs eller ett gemensamt objekt så kan det bli problematiskt om nya instanser av ett objekt tillåts att skapas. Därför kan Singleton-mönstret implementeras för att förbjuda skapandet av nya instanser.

ForeverAlone

**Struktur**

-PoorSoul : ForeverAlone

+test() : void

+get() : ForeverAlone

**Deltagare och sammarbete**

* **ForeverAlone**; singleton-objektet som instansieras. Klassen definierar hur objektet används för att hämta den specifika instansen (via en get-metod) samt hålla all eventuell inkapslad information.

**Konsekvenser**

Singleton-mönstret bryter mot SRP (Single Responsibility Principle) eftersom det kontrollerar sitt eget skapande och sin egen livscykel.

**Kodexempel**

public class ForeverAlone {

private static ForeverAlone PoorSoul = new ForeverAlone();

private ForeverAlone(){}

public void test(){ System.out.println("The object exists."); }

public static ForeverAlone get(){ return PoorSoul; }

}

public class SingletonPattern {

public static void main(String[] args) {

ForeverAlone singletonObject = ForeverAlone.get();

singletonObject.test();

}

}

**Relaterade mönster**

* **Abstract Factory**; implementeras ofta som singletons för att försäkra om att endast en factory kontrollerar objekt-skapande.

**Template Method Pattern**

**Avsikt**

En mall som håller det standardmönster en algoritm använder. På så vis kan subklasser skriva över och modifiera särskilda event i algoritmen.

**Motivation**

En algoritm kan behöva förändras beroende på vilken data den försöker använda, exempelvis en sorteringsalgoritm. Sorteringsalgoritmen kan behöva sortera en Array<String> liksom en Array<Int>. Därför kan Template Method Pattern standardisera en mall som har respektive subklasser för respektive uppgift som behöver utföras.

**Struktur**

SomeAlgorithm

TemplateMethodPatternDemo TemplateMethodPatternDemo

**<<interface>>**

+main() : void

+stepOne() : void

+stepTwo() : void

+start() : void

**extends**

Sort

-list : int[]

+Sort() : void

+stepOne() : void

+stepTwo() : void

**Deltagare och Sammarbete**

* **SomeAlgorithm**; standardklassen som definierar formatet på hur algoritmerna är uppbyggda.
* **Sort**; den specifika subklassen som skriver över SomeAlgoritm, en sorteringsalgoritm i detta fall.

**Kodexempel**

public abstract class SomeAlgorithm {

abstract void stepOne();

abstract void stepTwo();

public final void start(){

stepOne();

stepTwo();

}

}

public class Sort extends SomeAlgorithm {

private final int[] list;

public Sort (int[] list) {

this.list = list;

}

@Override

void stepOne() {

System.out.println("Original list: " + Arrays.toString(list));

Arrays.sort(list);

}

@Override

void stepTwo() {

System.out.println("Sorted list: " + Arrays.toString(list));

}

}

public class TemplateMethodPatternDemo {

public static void main(String[] args) {

int[] list = {3, 1, 7, 4, 2, 8};

SomeAlgorithm sort = new Sort(list);

sort.start();

}

}

**Relaterade mönster**

* **Factory Pattern**; metoderna/funktionerna kallas ofta template-metoder.
* **Strategy**; template-metoder använder arv för att växla vissa delar av en algoritm (se nästa mönster).

**Strategy Pattern**

**Avsikt**

Detta mönster används för att definiera och individuellt inkapsla en familj av algoritmer för att kunna använda olika algoritm-varianter under exekvering, omväxlande.

**Motivation**

I särskilda situationer kan ett program under exekvering behöva en annan strategi kring hur en algoritm ska utföras. Ett program kan exempelvis behöva använda olika resurser beroende vilken typ av webbläsare som används, vilket underliggande operativsystem som är närvarande, osv.

**Struktur**

Strategy

**<<interface>>**

StrategyPatternDemo

+Operation(int[]) : int[]

+Option(int[]) : int[]

+main() : void

SortAscending

SortDescending

+Operation(int[]) : int[]

+Option(int[]) : int[]

+Operation(int[]) : int[]

+Option(int[]) : int[]

**Deltagare & sammarbete**

* **Strategy**; det interface som definierar vilka metoder som familje-algoritmerna ska stödja.
* **SortAscending & SortDescending**; subklass och familjeimplementation till Strategy.

**Konsekvenser**

Klienter måste förstå skillnaden mellan strategierna som finns tillgängliga innan ett lämpligt alternativ kan väljas, för att undvika problem.

**Kodexempel**

public interface Strategy {

public int[] Operation(int[] list);

public int[] Option(int[] list);

}

public class SortAscending implements Strategy {

@Override

public int[] Operation(int[] list) {

Arrays.sort(list);

return list;

}

@Override

public int[] Option(int[] list) {

return list;

}

}

public class SortDescending implements Strategy {

@Override

public int[] Operation(int[] list) {

Arrays.sort(list);

list = Option(list);

return list;

}

@Override

public int[] Option(int[] list) {

for(int i = 0; i < list.length / 2; i++) {

int temp = list[i];

list[i] = list[list.length - i - 1];

list[list.length - i - 1] = temp;

}

return list;

}

}

public class StrategyPatternDemo {

public static void main(String[] args) {

Strategy Ascendinglist = new SortAscending();

Strategy Descendinglist = new SortDescending();

int[] list = {4,2,7,1,3,2};

list = Ascendinglist.Operation(list);

System.out.println("Ascending sort: \t" + Arrays.toString(list));

list = Descendinglist.Operation(list);

System.out.println("Descending sort: \t" + Arrays.toString(list));

}

}

**Relaterade mönster**

* **Template Method Pattern**; dessa två mönster kan användas i konjunktion för att kombinera familje-metoder och strategier med templates (mallar).

**Facade Pattern**

**Avsikt**

Facade används för att skapa en ”fasad” som gömmer ett mer komplicerat interface. Detta är användbart för att göra koden enklare att använda, förstå, läsa och för att gömma komplexitet.

**Motivation**

Vid tillfällen där användaren av ett program inte behöver/får veta hur programmet fungerar så är det användbart att skapa en s.k. fasad. Ett exempel på en fasad är gränssnittet i ett program eller ett operativsystem; allt användaren behöver veta är **vad** resultatet blir av interaktionen, inte **hur**. På så vis så är det samtidigt ett mönster som är svårt att låta bli att implementera om systemet har ett användargränssnitt.

**Struktur**

Component

FacadePatternDemo

**<<interface>>**

+run() : void

+main() : void

CPU

GPU

Facade

+run() : void

-CPU : Component

-GPU : Component

+run() : void

+start() : void

**Deltagare & Sammarbete**

* **Component**; ett interface som definierar hur varje komponent ser ut.
* **CPU & GPU**; implementationer av Component, utför olika uppgifter relaterade till deras funktionalitet.
* **Facade**; den centrala klassen i fasad-mönstret. Döljer underliggande komplexitet genom att låta Facade-instanser använda en start-metod.

**Kodexempel**

public interface Component {

public void run();

}

public class CPU implements Component{

@Override

public void run() {

System.out.println("CPU is running...");

}

}

public class GPU implements Component {

@Override

public void run() {

System.out.println("GPU is running...");

}

}

public class Facade {

private final Component CPU;

private final Component GPU;

public Facade() {

CPU = new CPU();

GPU = new GPU();

}

public void Start(){

CPU.run();

GPU.run();

}

}

public class FacadePattern {

public static void main(String[] args) {

Facade System = new Facade();

System.Start();

}

}

**Real uses**

Dem flesta system använder någon form utav Facade-liknande mönster. Ur en användares perspektiv i ett operativsystem eller en webbläsare så fungerar gränssnittet som en Facade eftersom vi inte vet eller får någon information om hur dem underliggande processerna till vår interaktion med interfacet faktiskt fungerar.