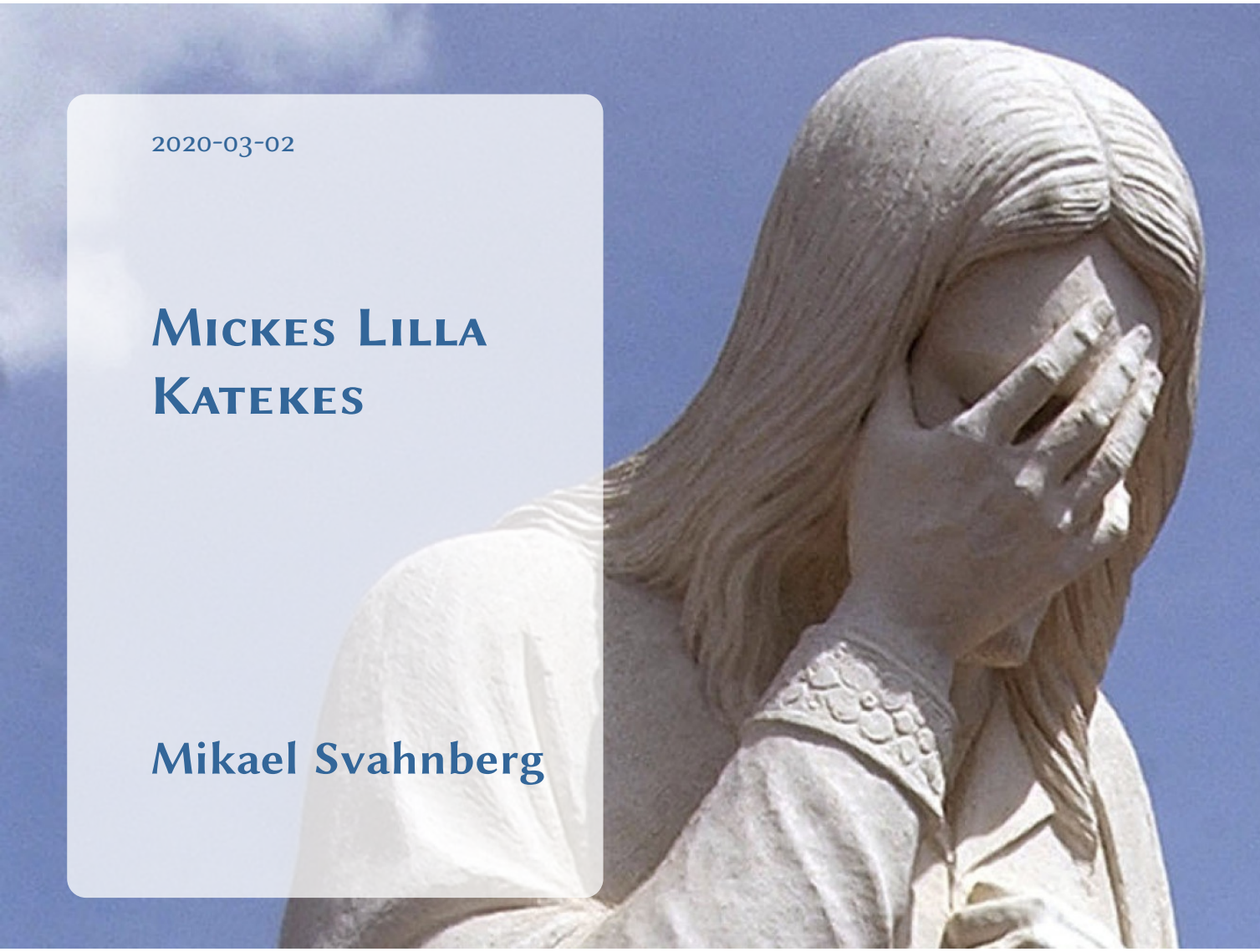
2020-03-02

Mickes Lilla Katekes

Mikael Svahnberg



Abstract

Det här dokumentet gäller kurserna PA1458, PA1459, och PA1460 vid Blekinge Tekniska Högskola, som på olika vis handlar om Objektorienterad Design.

Det finns ett antal uppgifter att göra i varje kurs, och det kan vara svårt att få en överblick och en förståelse för vad man förväntas göra och vad som är syftet med varje uppgift. Jag har noterat att om syftet inte är helt klart så är det svårt att göra lagom lite på uppgifterna. Syftet med min lilla katekes är alltså att ge lite mer förklaringar runtomkring alla uppgifterna. (Som om /mer/ text var vad ni behövde)

Dokumentet växer i takt med att jag ser vilka frågor jag behöver besvara så se till att du har den senaste versionen.

Kapitel I

Blandade Frågor

1

Vilka är Uppgifterna?

Uppgifterna är lite olika i de olika kurserna; se nedanstående tabell för en översikt:

Assignment	Course		
	PA1458	PA1459	PA1460
Discussion of Quality Attributes			X
Instantiated Architecture Style			X
Create Detailed Use Cases	X		
List System Events	X		
WBS and Project Plan	X		
Discussion of GRASP Patterns	X	X	X
Interaction Diagrams	X	X	X
Class and Package Diagram	X	X	X
Refactored Class Diagram	X	X	X
Refactored Interaction Diagram	X	X	X
Create Test Plan			X
Unit Test for Use Case	X		X
Implementation	X		
Demo Implementation	X		

2

Vilket system skall jag göra uppgifterna om?

Du skall alltid använda det givna systemet (I skrivande stund är det en variant av spelet Mean Streets).

Jag kan ibland introducera *vad som ser ut som* olika system (t.ex. en turist-app). Anledningen till detta är att jag får ofta höra “vi kan ingenting om spel”. Därför vill jag visa på att även om jag har utgått från ett spel, så finns det många applikationer som är nästan lika dana som och fungerar nästan på samma vis som vissa spel. Mean Streets och turist-applikationen är med andra ord samma system. Ni behöver alltså inte välja system; det har jag gjort åt er.

3

Jag gillar inte den givna lösningen, måste jag följa den?

Absolut inte. Men det blir lite svårare för er.

Uppgifterna följer en designkedja från kravinsamling via objektorienterad analys och design till implementation och testning. Mitt mål har varit att ni alltid skall få tillräcklig input om steget innan för att ni skall kunna göra en viss uppgift, och den uppgiften är kanske högst 30% av vad det skulle innebära om ni gjorde hela systemet själva. Delar av de resterande 60% ger jag sedan som input till den uppgift som handlar om nästa steg.

Om ni väljer att göra om den givna lösningen själva på ett bättre sätt så blir ni beroende av att ni gör rätt hela vägen, och kan inte luta er tillbaka mot de färdiga del-lösningar som jag har gett er.

Ni får väldigt gärna göra detta men motivera någorlunda noggrant i text hur och varför ni har gjort som ni har gjort, så att vi har en möjlighet att förstå hur ni tänker.

Korollarium: ni kan alltså alltid tjuvkika lite på inputen till nästa uppgift för att få se exempellösningar.

Kapitel II

Uppgifter om Mjukvaruarkitekturer

I den här kursen så finns det inte utrymme att gå på djupet om mjukvaruarkitekturer, utan föreläsningarna och uppgifterna syftar mer till att ge en allmän överblick av området. Det huvudsakliga målet med en mjukvaruarkitektur är att få fram en potential för att uppnå en viss blandning av kvalitetsegenskaper. Därför börjar vi med att resonera om dessa, innan vi går vidare och faktiskt gör en arkitektur.

4
Discussion of Quality Attributes

All design, och arkitekturdesign i synnerhet, handlar om medvetna beslut. Det första steget är alltså att ta reda på vad som är viktigt. Om nu syftet med en arkitektur är att förenkla för att åstadkomma vissa kvalitetsegenskaper så måste man ju först ta reda på vilka egenskaper som är viktiga. Annars riskerar man ju att lägga en massa tid och pengar för att få till en egenskap som kanske inte är viktig.

En lista med kvalitetattribut är given, och för vart och ett av dessa skall ni resonera om de givna frågorna (är attributet relevant för det här systemet? Hur viktigt är

det? Hur skulle man kunna lösa det?).

Om den sista punkten "Strategies (e.g. tactics) you may use to address the quality attribute, and their impact on the software architecture.": Tactics hittar ni i boken "Software Architecture in Practice". Men syftet är att ni skall kunna på ett övergripande plan kunna fundera på olika sätt att lösa de utmaningar som vart och ett av kvalitetsattributen för med sig.

Till exempel: Om ett problem är att det tar för lång tid för systemet att svara på frågor (Performance – Response Time), så kan man komma åt detta på olika sätt. Först och främst måste man ju fundera på *varför* detta är ett problem – är det för att det kommer för många frågor samtidigt, eller för att varje fråga är för komplex, eller är det för att det tar tid att få fram ett svar? När man vet detta så kan man lösa det på olika vis: Begränsa mängden samtidiga anrop, ge stöd för att förenkla frågorna, ha förututräknade svar, köra flera instanser av programmet på t.ex. moln-servrar, eller byta till en push-notis-arkitektur. Alla de här är exempel på *strategier* för att åstadkomma kvalitetsattributet.

Det viktiga är alltså inte var ni får strategierna ifrån (vilken bok ni tittat i), utan att ni kan resonera om ett par olika sätt att lösa de utmaningar som respektive kvalitetsattribut för med sig.

5
Instantiated Architecture Style

En *Komponent* är en logisk byggsten till ert system. Om man tittar på systemet man skall bygga, och bara försöker spåna fram vad man behöver för att kunna bygga det, så har man troligen de viktigaste komponenterna.

Exempel: Skall jag bygga en webbshop, så behöver jag en produktkatalog, en kundvagn, en kund-databas, och ett sätt att hantera betalningar. Där är mina komponenter (Jag behöver också en massa extra krångel om att visa produktkatalogen som en websida, en databashanterare, och så vidare, men det behöver man inte oroa sig för det första man gör).

Ett *paket-diagram* i UML är egentligen tänkt att fokusera på vilka paket som man skall organisera upp programkoden i (*packages* i Java, olika underkataloger i filsystemet i C++, kanske namespaces). Men diagrammet får göra dubbel tjänst genom att helt enkelt bli en grafisk representation av komponenterna som ni har kommit fram till behövs.

Det finns några syften med uppgiften:

- Ni skall öva lite på att se en väldigt övergripande produktbeskrivning och fundera på vilka komponenter, vilka byggstenar, man kan tänkas behöva.
- Ni skall fundera på arkitekturstilar, och hur olika stilar kan vara mer eller mindre lämpliga för systemet ni skall bygga.
- Ni skall se hur man gör om en generellt beskriven arkitekturstil till en konkret arkitektur med hjälp av de komponenter man har bestämt sig för att ha.
- Ni "testar" era komponenter och de ansvarsområden som ni gett respektive komponent (eller paket) genom att sortera in koncepten från domänmodellen i "rätt" paket/komponent.

- Ni övas på att göra medvetna val om hur ni tar hand om bestämda kvalitetsegenskaper genom den övergripande strukturen (arkitekturen) på ett system.

Kapitel III

Uppgifter om Objektorienterad Analys och Projektplanering

6
Create Detailed Use Cases

Ett use case går i grund och botten ut på att man skall täcka av ett helt arbetsflöde (ur användarens perspektiv) mot systemet. En viktig färdighet här är att man tänker på vad användaren säger till systemet, och vad systemet svarar, och hur man därmed ser arbetsflödet som ett slags dialog mellan användaren och systemet.

7
List System Events

Ni får ett antal use cases givna “Assignment Input: Examples of Detailed Use Cases”. De use cases som ni skall arbeta med finns med bland dessa.

I Use Cases beskriver vi med vanligt språk (t.ex. Engelska, Svenska, eller Latin) dialogen mellan användare och system. Men för att komma vidare med designen och att till syvene og sidst faktiskt implementera systemet så behöver vi översätta detta till något som datorn begriper, nämligen metodanrop.

Vi tänker oss därför att vi särar på systemet i sig självt och systemets gränssnitt (till exempel ett grafiskt gränsnitt med fönster och knappar och annat vackert). Användaren interagerar med hjälp av gränssnittet, som *översätter* till

metodanrop som anropas i systemet. Ju bättre vi är på att hålla isär dessa två delar, desto bättre, för då blir inte gränssnittet beroende av att veta en massa detaljer om hur systemet fungerar internt, och systemet kan strunta fullständigt hur gränssnittet är byggt.

Så i gränssnittet så använder vi oss av text-strängar och integers och sån't, och det blir systemets uppgift, när den har fått rätt strängar i ett systemanrop, att leta rätt på eller skapa lämpliga objekt som kan samarbeta för att lösa uppgiften och ge ett svar i form av en sträng, ett värde, eller en händelse, tillbaka till gränssnittet igen.

I den här uppgiften så läser vi våra use cases och försöker se vilka systemanrop (metodanrop) som faktiskt blir resultatet av dialogen mellan användaren och systemet. Vi tittar än så länge inte djupare än så, vi vill bara veta vilka systemanrop som tillsammans behövs för att bygga just detta use case.

Normalt uttrycker man detta i ett systemsekvensdiagram, men det är bara ett specialfall av sekvensdiagram som vi kommer komma till senare i kursen så ni lär er inte något nytt av att göra ännu ett diagram. I stället så räcker det med att ni listar systemhändelserna.

8
WBS and Project Plan

Det pågår just nu ett skifte i hur man ser på planering av utvecklingsprojekt. Å ena sidan finns den "gamla" skolan att man skall bryta ner allt i smådelar (en Work Breakdown Structure) och sedan göra en detaljerad tidsplan med ett GANTT-schema. Å andra sidan finns den agila synvinkeln att man bara fokuserar på det som skapar värde till kunderna, och bryter ner detta till lagom stora jobb att planera med. Det här är en ganska stor förändring i synsätt, men många delar av industrin tvekar och försöker i viss mån stå med ett ben i vardera lägret.

Den här uppgiften är just en sådan blandning mellan en gammal WBS och en nyare syn på estimat. WBS:en är baserad på modernare input (use cases i det här fallet;

det hade lika gärna (kanske ännu hellre) kunnat vara user stories), men resulterar i samma fokus på *arbetet* som skall utföras snarare än *värdet* som skapas. Det här efterfrågas fortfarande, och framför allt behöver ni få en förståelse för hur man bryter ner och storleksuppskattar arbetet som behövs för att skapa ett visst värde (ett visst use case). Med mer erfarenhet av detta så kommer ni bygga upp en vana i att uppskatta arbetets omfattning för en viss user story eller ett use case, men det här är de första stegen.

Samtidigt vill jag inte släppa iväg er med tron att det är bara såhär det går till i industrin (WBS, tidsestimat, långa och detaljerade planer). Därför introducerar jag också *story points* och hur man skulle kunna kombinera sin uppskattning av arbetets omfattning med en storleksuppskattning som i grund och botten försöker vara frånkopplad någon direkt översättning till de klassiska *person-timmarna*. Kopplat till detta blir då också begreppet *velocity* som något som är individuellt för en viss person eller en viss grupp. Planeringen blir alltså individanpassad snarare än den klassiska synen att en viss uppgift alltid tar en viss tid att genomföra.

Så betrakta den här uppgiften som ett mellansteg mellan klassisk plan-driven projektstyrning och moderna individ- och värdebaserade projektformer.

Kapitel IV

Uppgifter om Objektorienterad Design

9
Discussion of GRASP Patterns

Syftet med den här uppgiften är att ni skall fundera på de givna frågorna. Om ni gör det, så kommer ni ha gjort ett antal medvetna val som leder till en bättre design i nästa uppgift (Interaction Diagrams).

Vi kommer alltså att se i Interaction Diagrams - uppgiften hur ni har tänkt, och därför behöver ni inte lämna in er diskussion om GRASP patterns. Den är till för er och för att ni skall få hjälp att fokusera tankarna.

Jag upprepar: Discussion of GRASP Patterns skall *inte* lämnas in. Lägg inte tid på att skriva världens vackraste rapport här.

10

Interaction Diagrams

I den här uppgiften fortsätter vi designkedjan. Ni har klarat vägen från Use Cases till Systemsekvensdiagram, och nästa steg är att titta på vilka objekt som behövs och hur de måste samarbeta för att kunna ge rätt svar (och rätt följdverkningar genom resten av systemet) till varje anrop till systemet från systemsekvensdiagrammet.

Här övas ni i att fundera lite djupare ner i systemet; vad måste systemet faktiskt göra för att kunna ge rätt svar tillbaka, och vilka objekt är mest lämpade för att göra varje liten del. På många vis så går den här uppgiften till själva pudelns kärna i objektorienterad design, eftersom det är här ni behöver fundera på vilka objekt ni behöver och vad de skall göra.

Ett knep är att alltid försöka fundera på hur det skulle fungera i verkligheten; vilka "objekt" samarbetar i problemdomänen? Prova att rollspela: Var ett varsitt objekt och försök lösa systemanropet tillsammans utan att någon av er gör för mycket eller något som ni inte är bekväm med (i rollen som ett objekt av en viss klass, naturligtvis).

När ni är klara, så glöm inte att ta ett steg tillbaka och fundera på low coupling och high cohesion: Går det flytta ansvar om vissa saker till andra objekt och få bättre cohesion? Kan man flytta ansvar om vissa saker och därmed få ett lösare kopplat system?

Att designa är en process, man gör något, sedan gör man om, och om, och om igen.

11

Class and Package Diagram

Den här uppgifen är mycket enklare än vad ni tror.

Först, komponenter. Några av er har redan gjort detta som en del av “Instantiated Architecture Style” - uppgiften. För er andra så är det kanske lite nytt. Jag klipper och klistrar från vad jag skrev i den uppgifen:

En *Komponent* är en logisk byggsten till ert system. Om man tittar på systemet man skall bygga, och bara försöker spåna fram vad man behöver för att kunna bygga det, så har man troligen de viktigaste komponenterna.

Exempel: Skall jag bygga en webbshop, så behöver jag en produktkatalog, en kundvagn, en kund-databas, och

ett sätt att hantera betalningar. Där är mina komponenter (Jag behöver också en massa extra krångel om att visa produktkatalogen som en websida, en databashanterare, och så vidare, men det behöver man inte oroa sig för det första man gör).

Ett *paket-diagram* i UML är egentligen tänkt att fokusera på vilka paket som man skall organisera upp programkoden i (*packages* i Java, olika underkataloger i filsystemet i C++, kanske namespaces). Men diagrammet får göra dubbel tjänst genom att helt enkelt bli en grafisk representation av komponenterna som ni har kommit fram till behövs.

Det här är de två första delarna av uppgiften:

- Att öva på att se en väldigt övergripande produktbeskrivning och fundera på vilka komponenter, vilka byggstenar, man kan tänkas behöva.
- Att "testa" era komponenter och de ansvarsområden som ni gett respektive komponent (eller paket) genom att sortera in koncepten från domänmodellen i "rätt" paket/komponent.

För den sista delen så vänder vi lite på problemet. Här får ni två stycken interaktionsdiagram givna. Ni skall läsa

ut vilka klasser och vilka metoder som används i dessa interaktionsdiagram. Med den här kunskapen så kan vi ta våran konceptuella modell och börja göra om den till ett riktigt klassdiagram. Några av domänkoncepten används i interaktionsdiagrammen – de skall vi alltså uppgradera till riktiga klasser och fylla på med de metoder som används. Andra klasser som användes var kanske helt nya, då får vi lägga till dem i vårt klass-och-paketdiagram.

Det är alltså egentligen två olika diagram ni gör här – det ena är en logisk "gissning" om vilka komponenter som behövs och hur problemdomänen kan mappas till dessa komponenter, och det andra är att ni konkret vet vilka klasser och metoder som behövs för att implementera de två givna interaktionsdiagrammen. För att spara på antalet inlämningar ni måste göra så har jag bara valt att slå samman dem till en och samma uppgift.

På så vis så får ni också lite insikt i hur man tar domänkoncept och "uppgraderar" till klasser, och därmed ser ni lite tydligare hur man alltid strävar efter att programmet man bygger skall likna verkligheten så mycket som möjligt i objektorienterad design.

En liten detalj kvar: Eftersom det de facto är två olika diagram som vi har slagit ihop till ett, så behöver vi (för att lättare kunna rätta) och ni (för att lättare kunna se kopplingen mellan domänkoncept och klasser) kunna se

vad som är vad. Därför ber vi er att markera på något sätt vilka klasser som faktiskt är klasser som ni tog fram ur interaktionsdiagrammen, och vilka koncept som fortfarande bara är koncept.

Jag vill också avsluta med att påminna om att det slutgiltiga diagrammet är en tvåhövdad chimär som man aldrig skulle se i verkligheten. Man skulle förmodligen nöja sig med klasserna från interaktionsdiagrammen.

12

Refactored Class Diagram

Det man får ut av att göra klassdiagram av interaktionsdiagram är den minsta möjliga samlingen klasser som behövs för att lösa just den givna uppgiften. Det säger ingenting om hur lätt eller svårt det blir att underhålla systemet. I den här uppgiften så tar vi oss an att göra systemet lättare att underhålla genom att tillämpa olika design patterns.

Det är tre färdigheter som övas här:

- När jag ger er namnet på ett design pattern så kan ni söka fram hur det ser ut och sedan översätta till något konkret i just det här systemet.
- Ofta har man kanske inte namnet, utan bara ett problem; då behöver ni kunna slå i “design-pattern-

katalogen” och få fram vilket/vilka designmönster som skulle kunna lösa problemet.

- Till sist så bör man kunna känna igen när man utan ursprunglig tanke har råkat använda sig av ett design pattern, och att då kunna i efterhand annotera och eventuellt justera i designen så att man gör det till ett medvetet beslut.

13
Refactored Interaction Diagram

Här bryter jag mina principer att ni skall aldrig vara beroende av något ni själva har gjort för att kunna komma vidare i uppgifterna, men det spelar inte så stor roll i just det här fallet.

Det finns (som vanligt) några olika syften med den här uppgiften:

- Att kunna läsa "baklänges" från de ansvarsområden man har gett klasser och skapa interaktionsdiagram den här vägen.
- Att inse hur design patterns inte nödvändigtvis bara

ändrar i klassdiagrammen utan också hur interaktionen mellan objekt förändras av att man tillämpar ett visst design pattern.

Det finns säkert också en läxa här någonstans om vikten av att hålla all design-dokumentation uppdaterad när man gör refactoring på sitt system (oavsett om man gör det med koden eller med designdokumentation), men den läxan lämnar jag som en övning till läsaren.

Kapitel V

Uppgifter om Implementation och Testning

14

Create Test Plan

Om ni $SÖKMOTOR:ar på testplaner så kommer ni hitta fin-fina mallar för hur man skall formattera sitt testplans-dokument. Men genom att fylla i en sådan mall så glömmer man bort att faktiskt fundera på *vad* det är man fyller i. Så i stället så har jag lyft ut essensen av en testplan till ett antal frågor. Med svaren på dessa frågor så skulle det bli en barnlek för er att fylla i ett testplansdokument (och därför begär jag inte in det).

Formattering och sådant är inte viktigt; det är era svar på frågorna som är viktigast.

Syftet med uppgiften är att ni skall fundera på hur ni kan se till att alltid ha med er testning som en naturlig del av ert design- och utvecklingsarbete. Lika naturligt som

det är för er att använda konfigurationshanteringsverktyg, er favorit-utvecklingsmiljö, de inbyggda standardbiblioteken i ert programspråk, osv., lika naturligt skall det vara att tänka på testning och hur ni bygger system som är testbara.

15
Unit Test for Use Case

Den här uppgiften har några olika syften:

- Att sätta upp en utvecklingsmiljö för att göra enhetstester är mer eller mindre icke-trivialt, men har man gjort det en gång så är det mycket lättare nästa gång. Genom att ni gör det i den här kursen så räknar jag med att det är mer sannolikt att ni gör det i nästa kurs, eller i era projekt.
- Enhets-testning är en så grundläggande del av mjukvaruutveckling idag och med objektorienterad design så finns det så många enkelt identifierbara enheter (use cases, paket, klasser) att testa, så det går nästan inte att säga att man kan objektorienterad

systemutveckling utan att man faktiskt kan genomföra enhetstestning.

- Det är inte alltid lätt att se hur en viss design skall översättas till körbar kod, så att bygga en del av designen ger en viktig förståelse för hur den här översättningen kan gå till. Vissa av er (beroende på kurskod) har också en implementation att göra, men för er andra så är det här så nära implementationen ni kommer, och ni får en viktig övning i att översätta ett tidigt designarbete till mätbara resultat av ett körande system.

Ni behöver implementera *lite* av systemet för att åtminstone kunna ge meningsfulla svar tillbaka på de testade systemanropen, men ni behöver inte gå på djupet och implementera interaktionsdiagrammen hela vägen ner.

16

Implementation

Här implementerar vi slutligen två use cases, och upptäcker vad vi borde ha tänkt på i vår design. Det här innebär att i nästa design ni gör så är det ett par fällor mindre att ramla i.

Det är inte ett fullständigt system som efterfrågas, men jag vill se något som faktiskt följer designen och använder de klasser och metoder som ni har designat fram tills nu (eller som ni får givna som assignment input).

Redovisning av den här uppgiften är dels inlämning av koden, men framför allt en demo och diskussion av den; detta bokas separat (se Demo Implementation).

17
Demo Implementation

Detta är en fortsättning på Implementations-uppgiften, där ni visar ert system för en lärare och svarar på frågor om er kod och dess design.