Avdelning för datavetenskap

Mattias Hedman och Fredrik Johansson


# Utvärdering av enhetstestning för Palasso

## Evaluation of unit testing on Palasso

Datavetenskap

Examensarbete 15hp


| | |
|---|---|
| Datum/Termin: | 2009-06-05 |
| Handledare: | Johan Garcia |
| Examinator: | Martin Blom |
| Löpnummer: | C2009:06 |



Karlstads universitet 651 88 Karlstad

Tfn 054-700 10 00 Fax 054-700 14 60

Information@kau.se www.kau.se

# Utvärdering av enhetstestning för Palasso

**Mattias Hedman Fallquist & Fredrik Johansson**

© 2008 Mattias Hedman, Fredrik Johansson och Karlstads universitet

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

___________________________________________

Mattias Hedman Fallquist & Fredrik Johansson

Godkänd, 20090605

___________________________________________

Handledare: Johan Garcia

___________________________________________

Examinator: Martin Blom

# Sammanfattning

I Karlstad utvecklas det marknadsledande löne- och PA-stytemet för den statliga sektorn Palasso. För att få så hög kvalité på Palasso som möjligt genomförs omfattande testning. Testning genomförs för att säkerställa att mjukvaran gör det den ska göra. Dock har testningen varit på en högre nivå. För att fortsätta utvecklas har gruppen som utvecklar Palasso börjat att snegla på de nya utvecklingsmetoderna som har blivit populära inom dataindustrin de senaste åren. Dessa utvecklingsmetoder bygger på att använda sig av test på så låg nivå som möjligt, så kallade enhetstest.

För att undersöka möjligheterna för Palasso att använda enhetstest, har två olika delar undersökts av detta arbete. Först genomfördes en studie av de befintliga ramverk som finns för enhetstest. Därefter valdes det ramverk som var bäst lämpad för Palasso. Därefter genomfördes en experimentdel för att se om det var möjligt att införa enhetstest på Palasso. Detta gjordes med det valda testramverket. Denna del delades upp i två metoder. I metod 1 togs en modul av Palasso och skrevs om för att visa på hur koden måste vara uppbyggd för att enhetstestning ska kunna införas. I metod 2 infördes enhetstest på befintlig kod för att bevisa att det var möjligt att införa enhetstest och att små ändringar gör koden mer testbar.

Det ramverk som valdes till det bäst lämpade för Palasso blev TestNG. Detta för att det var lätt att använda och att det var skalbart. Genom experimentdelen kunde slutsatsen dras att det var möjligt att införa enhetstest på Palasso. Det ska dock införas i samband med nyutveckling eller när ändringar av den befintliga koden ska genomföras. Införandet av enhetstest ger många fördelar därför är slutsatsen att det vore Palassossystemets kvalité till gagn att använda sig av enhetstest.

# Evaluation of unit testing on Palasso


## Abstract

The market-leading wage and PA-.system for the Swedish public sector Palasso is developed in Karlstad. In order to achieve the highest quality of code , the developing team behind Palasso has been conducting extensive testing. Testing is done to ensure that the software does what it should do. However, the testing has been done at a high level. In order to continue to evolve, the group that develops Palasso has started to look at the new development methods that have become popular in the computer industry in recent years. These methods are based on the use of tests as low as possible so-called unit tests.

To examine the possibility for Palasso to use unit testing, the study has been divided into two parts. First, a study of existing frameworks to find the best suited one for Palasso. Then, by using the selected framework, a second study is conducted to see if it is possible integrate unit testing with the Palasso system. This part was divided into two methods.

In the first method a existing functionality was redeveloped to show how the code should be structured and implemented to be suitable for unit tests. In the second method unit testing was introduced on legacy code to prove that it was possible to introduce unit test and that small changes can make the code more testable.

The best suited framework for Palasso was TestNG. TestNG was chosen because it was easy to use and it had scalability. The second study showed that it is possible to integrate unit tests with the Palasso system. But it should be introduced when new modules are developed or when existing code is modified. The introduction of unit testing provides many advantages. It is concluded that the quality of the Palassos system would benefit from using unit testing.

# Innehållsförteckning

**1 Översikt ... 1**

1.1 Sammanfattning ... 1

1.2 Dokumentstruktur ... 2

**2 Bakgrund ... 3**

2.1 Validering och verifiering ... 3

2.1.1 Inspektion

2.1.2 Testning

2.1.3 Komponenttestning och enhetstestning

2.2 Utvecklingsprocess ... 6

2.2.1 Traditionell

2.2.2 Scrum

2.2.3 TDD

2.3 Testning i dag hos Palassogruppen på Logica ... 10

2.3.1 Komponenttestning

2.3.2 Integrationstestning

2.3.3 Systemtestning

2.3.4 Acceptanstestning

2.4 Uppdrag ... 11

**3 Förstudie ... 12**

3.1 Allmänt om testverktyg ... 12

3.2 JUnit ... 13

3.3 TestNG ... 13

3.4 JTiger ... 16

3.5 Unitils ... 17

3.6 Krav ... 18

3.7 Diskussion och slutsats ... 20

**4 Experiment-översikt ... 22**

4.1 Palassosystemet ... 22

4.2 Systemuppbyggnad ... 22

4.3 Moduluppbyggnad ... 23

4.3.1 MVC

4.3.2 Klient/Server

4.4 Inledning till experiment .......... 25
4.5 Förklaring av metoder .......... 25
**5 Experiment - Metod 1 .......... 26**
5.1 Översikt .......... 26
5.2 Design .......... 26
5.2.1 DatabaseHandler
5.2.2 Modul
5.2.3 TDD
5.3 Utveckling .......... 30
5.3.1 SQLConnection
5.3.2 Row
5.3.3 Table
5.3.4 ObjectTable
5.3.5 Server
**6 Experiment - Metod 2 .......... 39**
6.1 Tekniker för att få test på plats .......... 39
6.1.1 Införa subklass för testning
6.1.2 Extrahera och skriva över anrop
6.1.3 Parameteriserar konstruktorn
6.1.4 Extrahera interface
6.1.5 Skapa och använda fakeobjekt
6.2 Beskrivning av vald modul .......... 44
6.2.1 SprakModule
6.2.2 SprakController
6.2.3 SprakView
6.2.4 Relationer
6.3 Genomförande av experiment .......... 48
6.3.1 Avgränsning
6.3.2 Sammanfattning av införandet av enhetstester för SprakController
6.3.3 Ändringar av SprakController för ökad testbarhet
**7 Diskussion .......... 53**
7.1 Det valda ramverket .......... 53
7.2 Metod 1 .......... 54
7.2.1 Att skriva relevanta testfall
7.3 Metod 2 .......... 56
7.3.1 Testning
7.3.2 Dokumentation
7.3.3 Tid
7.3.4 Ändringar
7.3.5 Sammanfattning
**8 Slutsats .......... 61**
8.1 Testverktyg .......... 61
8.2 Införande av enhetstest .......... 61

**Referenser** ........ **63**

**A Appendix** ........ **65**

# Figurförteckning

Figur 1: Traditionell utvecklingsprocess (vattenfallsmodellen[3]) ....................................7
Figur 2: V-modellen.........................................................................................................10
Figur 3: Standardfunktioner enhetstestramverk ..............................................................12
Figur 4: Standardnotering................................................................................................14
Figur 5: Gruppering ........................................................................................................14
Figur 6: Uppbyggnad ......................................................................................................17
Figur 7: Testfall objekt.....................................................................................................17
Figur 8: Mock objekt Unitils ...........................................................................................18
Figur 9: MVC ..................................................................................................................23
Figur 10: Klient/Server ....................................................................................................24
Figur 11: Palasso server....................................................................................................24
Figur 12: DatabaseHandler...............................................................................................27
Figur 13: Arvstruktur för Object table ..............................................................................27
Figur 14: DatabaseHandler flöde.......................................................................................28
Figur 15: Flöde i applikationen ........................................................................................29
Figur 16: Konstruktor-test.................................................................................................30
Figur 17 Inledande klass ...................................................................................................30
Figur 18 Inledande test......................................................................................................31
Figur 19 Metod före omstrukturering ................................................................................31
Figur 20 Klass efter omstrukturering.................................................................................32
Figur 21 Test av stängningsfunktion ..................................................................................32
Figur 22 Stängning av socket..............................................................................................33
Figur 23 Test med dataprovider..........................................................................................33
Figur 24 Equals-funktion ...................................................................................................34
Figur 25 Test equalfunktion ...............................................................................................34
Figur 26 Test statement.......................................................................................................35
Figur 27 Add test ...............................................................................................................36

Figur 28 Testklass efter omstrukturering ....................................................................37
Figur 29 Beroende testfall ....................................................................38
Figur 30 Normal testning ....................................................................39
Figur 31 Införd subklass för test ....................................................................40
Figur 32 Exempel på problem ....................................................................40
Figur 33 Ändrad metod till möjlighet att skriva över ....................................................................41
Figur 34 Metod som skriver över ....................................................................41
Figur 35 Utan möjlighet till dependency injection ....................................................................41
Figur 36 Med parameteriserad konstruktor ....................................................................41
Figur 37 Exempel på beroende ....................................................................42
Figur 38 Införande av interface med nytt namn ....................................................................42
Figur 39 Införande av interface med konkreta klassens gamla namn ....................................................................43
Figur 40 SprakModules hierarki ....................................................................45
Figur 41 SprakControllers hierarki ....................................................................46
Figur 42 SprakViews hierarki ....................................................................47
Figur 43 Klassernas relationer ....................................................................48
Figur 44 SprakController innan ändring ....................................................................51
Figur 45 SprakController efter införandet av SprakAdapter ....................................................................51
Figur 46 SprakController med FakeSprakAdapter ....................................................................52
Figur 47: Exempel get funktion ....................................................................57

## Tabellförteckning

Tabell 1: Funktionalitet ........................................................................................ 20
Tabell 2: Mer funktionalitet ................................................................................ 21

# 1 Översikt

## 1.1 Sammanfattning

Inom datavetenskapen finns det ett område som rör utvecklandet av mjukvara detta kallas för Software Engineering. Inom detta område finns en del som behandlar om mjukvara beter sig som önskat. Detta område kallas för testning och enhetstestning är en del av detta område.

För att genomföra enhetstestning används 2 delar:

- Enhetstest
- Enhetstestningsverktyg

Enhetstestning är testning av de minsta beståndsdelarna av mjukvara. Detta arbete handlar om att undersöka användandet av enhetstester.

För att lätt kunna använda sig av enhetstester har flera olika verktyg utvecklats. Dessa har olika funktionalitet och fördelar. Dock har de alla grundfunktionen att kontrollera om test går igenom eller inte. Detta arbete kommer att fokusera på att välja ut det bäst lämpade verktyget och införande av ramverket för ett speciellt företag, Logica.

Logica är ett av de största IT-företagen i Europa. I och med sitt uppköp av det svenska företaget WM-data blev de också ägare till lönesystemet Palasso som är specialiserat på den statliga sektorn. Ca 200 000 personer får sin lön av Palassosystemet varje månad. För att hela tiden se till att systemet har en så hög kvalité som möjligt söker gruppen hela tiden nya vägar för att uppnå det. Under de senaste åren har det uppkommit nya sätt att utveckla mjukvara, där flertalet fokuserar på ett systematiskt användande av enhetstest. Detta har gjort att Logicas Palassogrupp blivit intresserade av enhetstester.

Detta är motivet för detta arbete, där flera olika delar av enhetstestning kommer att undersökas. Då få av utvecklarna inom gruppen har använts sig av enhetstest innan behövs en utvärdering både av olika ramverk och hur enhetstestning går till. Först ligger fokus på att ta fram den bäst lämpade testverktyget för Palasso. Detta skall göras genom en studie av de olika ramverk som finns och vilka för- och nackdelar som de för med sig. Palasso är ett stort och gammalt system som består av flera olika tekniker. Dock går Palasso mot att bli mer Javabaserad. På grund av detta kommer arbetet att jobba mot att införa enhetstester på kod som är i programmeringsspråket Java.

Genom att studera dessa nya utvecklingsmetoder som blivit populära inom dataindustrin de senaste åren och specifikt hur de använder sig av enhetstester ska vi komma fram till hur och om det är möjligt att införa enhetstester på Palasso.

För att kunna undersöka möjligheten att enhetstesta denna del av Palasso kommer två olika metoder att användas. Den första metoden går ut på att visa hur nyutveckling av moduler skulle kunna se ut i Palasso med enhetstester. Den andra metoden är att försöka få enhetstest på redan skriven kod och därefter genomföra ändringar som ökar möjligheten för enhetstester. I den första metoden valdes en Klient-Server modul med fokus på server-delen. I den andra metoden valdes en modul uppbygg av MVC-mönstret. Genom detta experiment framkom det att det var möjligt att införa enhetstester och att det för med sig fördelar som dokumentation och design.

## 1.2 Dokumentstruktur

Kapitel 2 handlar om bakgrunden till experimentet, vad testning är och vilka olika typer det finns samt olika metoder för utveckling av programvara. Där återfinns också information om Palasso och Logica.

Kapitel 3 handlar om förstudien som gjordes. Den handlar om olika typer av enhetstestramverk, vad som de har gemensamt och inte. Där finns även valet av enhetstestramverk.

Kapitel 4 innehåller en översikt över experimentet som utfördes.

Kapitel 5 handlar om metod 1, vilket beskriver hur Palasso skulle kunna använda TDD och enhetstester vid utveckling av nya moduler.

Kapitel 6 handlar om metod 2, vilket beskriver hur införandet av enhetstester på befintlig kod i Palasso.

Kapitel 7 innehåller diskussion om de olika delarna i rapporten.

Kapitel 8 innehåller våra slutsatser av detta arbete.

# 2 Bakgrund

Software engineering kan definieras som vetenskapen för hur mjukvara ska utvecklas. Enligt Ian Sommerville består den av följande fem stora delar[1]:

- Krav
- Design
- Kritiska system
- Validering och verifiering
- Management

De olika delarna behandlar olika aspekter som spelar roll när man ska utveckla mjukvara. Inom dessa områden är det validering och verifiering som denna uppsats kommer att behandla.

## 2.1 Validering och verifiering

Validering och verifiering definieras enligt Sommerville[1] som:

- Validering – Skapar vi rätt produkt?
- Verifiering – Skapar vi den på rätt sätt?

Validering är att säkerställa att en produkt utför det som har bestämts att den ska kunna utföra enligt de specifikationer som har satts, vanligtvis att en kund vill ha en produkt som utför en viss uppgift.

Verifiering är att en produkt utför dessa operationer enligt det sätt som utvecklaren har tänk att den ska kunna göra.

Inom valideringen och verifieringen finns två större tekniker som används. Dessa är inspektion och testning[1][5].

### 2.1.1 Inspektion

Inspektion innebär att gå igenom programmet och kontrollera de olika delarna i utvecklingen bland annat källkoden, specifikationer och design. Detta innebär en kontroll av de delar som man kan kontrollera, utan att exekvera den specifika mjukvaran[1].

### 2.1.2 Testning

Testning innebär att exekvera mjukvara med vissa inmatningsvärden och sedan kontrollera att mjukvaran beter sig som det är tänkt och att det resultat som man får ut av testet matchar det resultat som väntades [1][6].

Inom testning finns flera olika sätt att se på hur de olika delarna av testningen ska delas upp, vilka framgår av skillnaderna i beskrivningen[1][5][7]. Anledningen till skillnaderna kan vara att de olika delarna inom validering och verifiering är oklara vad gäller vilka delar som hör till testningen och vilka som hör till inspektion då dessa aktiviteter stödjer varandra. Till exempel för att kunna skriva bra test måste man veta vad man ska testa och för att veta det måste man veta hur det är tänkt att det ska fungera enligt kraven. I denna uppsats kommer dock den ovanstående definitionen att vara den gällande.

Testning av mjukvara kan ske på olika nivåer. Enligt International Software Testing Qualifications Board[7] är dessa följande.

- Komponenttestning
- Integrationstestning
- Systemtestning
- Acceptanstestning

Översättningen till svenska har skett med hjälp av SSTBs ordlista[8].

**Komponenttestning**

Är testning på låg nivå där det fokuseras på de minsta delarna i mjukvaran. Detta kan vara en klass, en metod eller en funktion. Målet är att kunna testa alla de små delarna i isolation för att se att de fungerar som de ska. Denna testning körs vanligtvis av programmeraren som skriver den koden som testas.

**Integrationstestning**

Är test på mellannivå där exekveras test för att kontrollera att de olika enheterna inom ett system fungerar med varandra, till exempel att de olika gränssnitten mellan både mjukvara och hårdvara fungerar som de ska.

**Systemtestning**

Är test på hög nivå, dessa innefattar att testa att alla de delar som innefattas i ett system fungerar tillsammans. Dessa fokuserar på att se till att de krav som ställts på systemet blir uppfyllda. Detta genom att till exempel använda ”användarfall”, där försöks det att genomföra uppgifter ur en användares perspektiv.

**Acceptanstestning**

Denna testning är till största del inte till för att hitta fel utan istället ett tillfälle för kunden att kunna kontrollera att systemet beter sig som kunden tänkt sig.

### 2.1.3 Komponenttestning och enhetstestning

Komponenttestning har inom datavärlden också andra namn. På svenska talar man om också om enhetstestning och på engelska om unit testing men dessa har samma betydelse[8]. Intresset för enhetstestning eller unit testning ökar mer och mer inom datavärlden[9]. En av anledningarna till detta kan vara nya sätt att utveckla mjukvara som till exempel. eXtreme programming, som lägger mycket vikt vid just enhetstestning[10]. Som påpekas i Robert C Martins bok[12], används inte bara testningen som enbart test på att mjukvaran fungerar utan också som dokumentation. Dokumentation i den formen gör att testerna visar hur koden kan användas. En annan aspekt av enhetstestning är att den fungerar som ett skyddsnät[12]. Om du ändrar eller rättar fel i mjukvarukoden kan man senare kontrollera att man inte har förstört gammal funktionalitet genom att kunna exekvera alla enhetstester som kontroll.

Enhetstestning ger med andra ord mycket feedback till utvecklaren på ett snabbt och enkelt sätt. Detta bidrar till att öka kvalitén samtidigt som de ovan nämnda fördelarna kommer på köpet.

För att enhetstestning ska kunna användas på ett bra sätt finns vissa praktiska regler som ingår i begreppet[11]. Exempel på krav:

- Enhetstester ska vara automatiserade så att de är så lätta som möjligt att exekvera.
- Enhetstester skrivs i samband med att koden skrivs.
- Enhetstester ska finnas för alla enheter som finns i systemet.
- Enhetstester ska vara så enkelt skrivna som möjligt.
- Enhetstester ska ge ett resultat som indikerar om testet gick igenom eller inte
- Alla de enhetstester som finns ska gå igenom innan ny kod skrivs.
- Alla testfall av enhetstesterna ska underhållas och användas av alla utvecklare som gör ändringar i koden.

**Dessa olika regler har också sina förklaringar:**

Automatiserade tester hjälper utvecklaren att genomföra dessa. Detta i sin tur ger utvecklaren möjligheten att exekvera dessa tester oftare, jämfört med manuell testning av funktioner. Detta ger fördelar speciellt när många funktioner har testats innan. Svårigheter att exekvera testfall gör att testen inte körs lika ofta, vilket får till följd att när ett fel uppstår kan det vara länge sedan som felet infördes i koden och orsaken till felet blir svårare att lokalisera.

Enkelheten bakom enhetstester ligger i att det inte ska bli mer komplicerat att använda enhetstester utan att de ska finnas som stöd för utvecklaren. Är testen enkla blir också resultaten lätta att tolka.

För att enhetstestningen ska fungera fullt ut måste testningen vara så heltäckande som möjligt. Detta för att när ett fel lokaliseras, ska kunna få indikationer på var felet skulle kunna vara. Om ett system bara är delvis enhetstestat kommer de luckor som då uppstår att skapa förvirring när felet ska lokaliseras. Finns det enhetstest för alla enheter kan tiden som läggs på att hitta fel minskas.

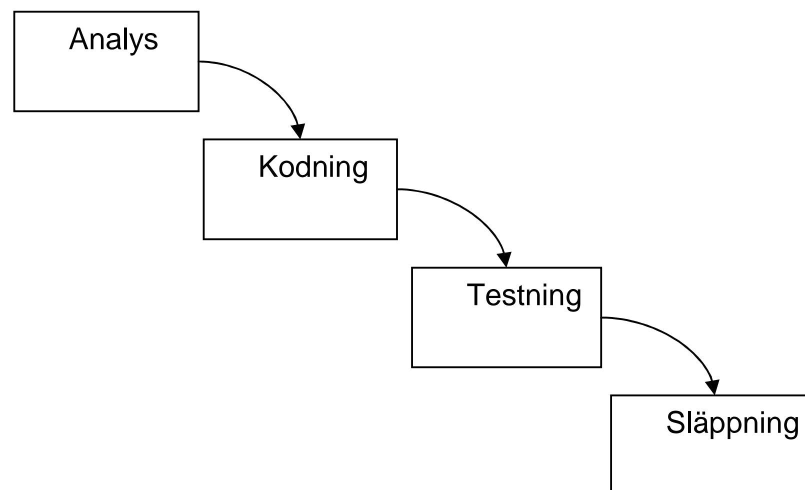
Endast när alla tidigare enhetstester fungerar som det ska är det dags att fortsätta att utveckla nya funktioner. Att skriva nya funktioner utan att tidigare tester går igenom är inte bra, då de saker som ger småfel i början ger till följd att utvecklaren ser lättare fel som uppstår och de små fel som finns följer med i utvecklingen och ger koden sämre kvalité.

## 2.2 Utvecklingsprocess

Testning av programvara har historiskt sett levt en undanskymd tillvaro enligt Michael A. Cusumano och Stanley A. Smith[4]. Till en början var testningen enbart för att hitta fel i de instruktioner som skulle genomföras, debugging. Med debugging lades fokus på att säkerställa att produktens instruktioner var korrekt skrivna. Däremot ansågs det inte nödvändigt att kontrollera att instruktionerna gav önskad funktionalitet.

Anledningen till att verifikationen av programmet inte ansågs nödvändig var att de inte hade komplex struktur och var inte med dagens mått mätt särskilt stora. När programmens komplexitetsgrad ökade började dock programmerarna inse att det var nödvändigt att kunna säkerställa att programmen mötte de krav som ställdes på dem.

### 2.2.1 Traditionell




*Figur 1: Traditionell utvecklingsprocess (vattenfallsmodellen[3])*

Utvecklingen av programvara har traditionellt sett varit att först skriva funktionaliteten och sedan testa det färdiga resultatet. Utvecklingen har setts som en rak linje utan någon möjlighet att gå tillbaka till de steg som varit innan, denna modell heter vattenfallsmodellen. Detta har lett till att det i vissa fall har spenderats mycket pengar på utveckling av programvara som inte har kunnat utföra den funktion som den var avsedd att utföra, eller att den inte mötte kundens krav. Detta resulterade i en produkt som kasserades[3].

Mjukvaruföretag upptäckte för 20 år sedan att man hade det problemet. Det berodde på att denna modell inte är flexibel nog att klara förändringar i kraven som kunden ställer på programvaran. Om det blev fel i början blev det tvunget att börja om för att det inte fanns några vägar tillbaka i denna utvecklingsprocess[4].

För att kunna utveckla programvara på ett kostnadseffektivt sätt var feedback tvunget att ske mycket tidigare i utvecklingsprocessen. Genom att testa programvara och få feedback från kunden i samband med utvecklingen kan missförstånd mellan kunden och utvecklaren motverkas.

### 2.2.2 Scrum

Till skillnad från det traditionella sättet att utveckla programvara är Scrum en modell där utvecklingen sker stegvis. Det är en utvecklingsprocess som är iterativ, det vill säga att den är cyklisk, där varje cykel är ett komplett projekt med krav från kunden, design, implementation, testning och sedan demonstration för kunden. Inom Scrum kallas en sådan iteration för en Sprint. Kunden blir alltså involverad under hela utvecklingen och kan komma med feedback under tiden.

Eftersom kunden är involverad tidigt i utvecklingen är det lätt att göra ändring på funktionalitet som har gjorts innan, då fel oftast upptäcks i sprinten och kan då ändras till nästa. I början av ett Scrum projekt ger kunden en produktlist (Product-backlog)[13], vilket är en kravlista där alla specifikationer står i prioriteringsordning. Ett projekt är färdigt när alla punkter på produktlista är avprickade. Denna produktlista är inte statisk utan kan ändras under projektets gång. Enligt Kniberg[13] ger detta en produkt som är mycket närmare det kunden vill ha än med vattenfallsmodellen.

För att kunna använda Scrum krävs det kontroll av funktionalitet och korrekthet av koden i samband med utvecklingen då tidsutrymmet är mycket kort och man har inte tid att ändra alla fel i slutet av sprinten. När ett kodblock har utvecklats måste det kontrolleras[14]. Test driven development är ett sätt att lösa detta på.

### 2.2.3 TDD

De utvecklingsprocesser som har uppkommit senast, till exempel Scrum, har integrerats med TDD, test driven development[13]. Idén bakom TDD är att för att kunna motivera att en funktionalitet behövs måste det bevisas och det sker genom ett test för den funktionaliteten skrivs innan den blivit implementerad. Om testet inte går igenom behövs den och implementeras annars behövs den inte och den lämnas.

De tester som skrivs är små och testar funktionaliteten för en viss subrutin, kodblock som kan anropas för exekvering, eller en klass. Denna process är liksom Scrum iterativ och i varje cykel genomförs dessa steg:

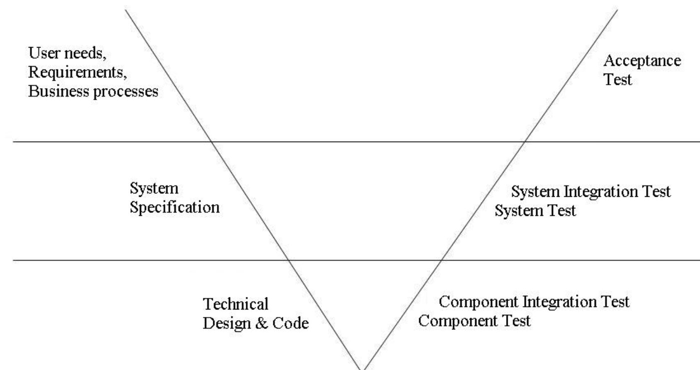
- Skriv ett enhetstest
- Testkör alla enhetstester och kontrollera om det nya testet genererar fel
- Skriv kod
- Testkör alla test och kontrollera att alla är godkända
- Omstrukturering (Refactoring[19]) – Förbättring av befintlig kod utan att ändra det externa beteendet
- Börja om

Denna metod bygger på att utvecklingen sker i små steg och utvecklarna ska varken göra mer eller mindre än det som krävs för att klara testerna. Ett begrepp som är stort inom TDD är YAGNI, (”You ain't gonna need it”), vilket i detta sammanhang betyder ”varför utveckla något som du inte vet att du kommer att behöva”. Anledningen till att utvecklingsprocessen ser ut så här är att man fokuserar på att generera så effektiv kod och programvara som möjligt och skriva funktionalitet som inte är nödvändig är ineffektivt och resurskrävande, enligt Kniberg[13].

TDD är alltså inte en testprocess utan en utvecklingsprocess. En stor anledning till att den har tagits fram är just att tester i utvecklingsfasen har varit bristfälliga eller inte existerat hos många företag. Detta har i många fall lett till att kod har levererats, vars funktionalitet är tvetydig och det har varit svårt att se vad som levereras till sluttestarna. I normalfall görs det bara sluttester, systemtester och acceptanstest på de flesta företagen, enligt Jan Säll, Palassogruppen Logica.

## 2.3 Testning i dag hos Palassogruppen på Logica

Idag använder sig Logica av den så kallade V-modellen, som består av ett flöde där vissa tester matchas mot olika delar i utvecklingen av mjukvara[16].




*Figur 2: V-modellen*

De delar som Palassogruppen fokuserar på i denna modell är de övre delarna som handlar om att testa hela systemet och att validera systemet.

### 2.3.1 Komponenttestning

Idag används inte enhetstestning över huvudtaget i Palassogruppen på Logica. På utvecklingsnivån finns det inte några bestämda krav på hur testningen ska utföras. Det är upp till varje enskild utvecklare att ta ansvar för att kodkvalitén är så hög som möjligt. Anledningen till att enhetstester inte har använts har varit att man inte sett det nödvändigt att testa på den nivån. Bakgrunden till detta ligger i att testningen och utvecklingen har varit skilda delar där var del bara ska ha fokus på sitt område.

Fokus ligger i dag på testning av hela systemet, systemtestning, och testning mot krav som kunden har ställt på produkten, acceptanstest.

### 2.3.2 Integrationstestning

Denna testning ligger mellan enhetstestning och systemtestning. Denna del är ligger på samma nivå i V-modellen som enhetstestning där det inte sker någon testning, trots att det formellt sker testning på alla nivåer i V-modellen. Då ingen testning sker i detta steg ligger det på utvecklarens ansvar att se till att kodkvalitén är så hög som möjligt och att de gränssnitt som används fungerar med varandra.

### 2.3.3 Systemtestning

I denna fas börjar testningen av Palassosystemet på Logica. Denna fas är den lägsta nivån av testningen som görs på ett uttalat sätt. Om det skulle ske ett fel i denna fas kommer felet att rapporteras till utvecklarna som får åtgärda felet.

### 2.3.4 Acceptanstestning

Detta är det sista steget av de testningsfaser som de har i Palassogruppen på Logica. Detta steg görs av testare som ser till att kunden kan genomföra de operationer som systemet ska kunna klara av.

## 2.4 Uppdrag

Logica har börjat använda sig av Agila metoder[13] för att utveckla mjukvara där det fokuseras mycket på just enhetstestning. I och med detta behöver de stöd för att kunna välja vilket ramverk för enhetstestning som de ska använda sig av.

Vårt uppdrag har två delar:

- I en första delen ska vi titta på befintliga ramverk för enhetstestning och utifrån de krav som Logica har ställt ska en utvärdering göras. Detta ska sedan leda till ett val av det bäst lämpade ramverket.
- Den andra delen består av att vi ska försöka integrera verktyget med Palassos utvecklingsmiljö. Därefter kommer vi att enligt två metoder integrera enhetstester med Palasso. Detta kommer att leda till en slutsats om det går att införa enhetstester eller ej.

# 3 Förstudie

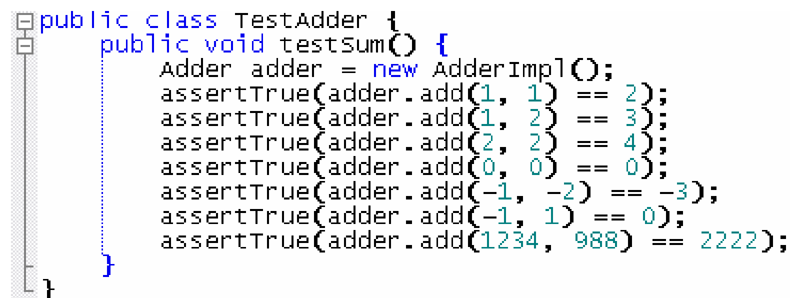
För att kunna välja vilket testverktyg som är det bäst lämpade för Palasso genomförs en förstudie av befintliga ramverk. Studien är uppdelad i 3 delar:

1. Allmänt om testverktyg
2. Individuella studier av olika testverktyg
3. Diskussion och val av verktyg

Det valda testverktyget kommer sedan att användas i kommande experiment.

## 3.1 Allmänt om testverktyg

Ett testverktyg, eller testramverk, tillhandahåller subrutiner för att jämföra objekt med förväntade egenskaper, eller värden med förväntade värden. Dessa funktioner kallas för Assertfunktioner, de ger svar som är sanna eller falska.




*Figur 3: Standardfunktioner enhetstestramverk*

Dessa funktioner har till uppgift att underlätta för utvecklaren att använda sig av enhetstestning, då de är definierade och implementerade i testramverken. Assertfunktioner är vanligaste funktionen som används, då det mest intressanta i ett system är att se om objekt har förväntade egenskaper.

För att det ska vara så lätt som möjligt att använda enhetstestningsramverken följer det med en exekverare, vilket har som uppgift att exekvera de olika testfallen. I samband med detta hjälper ramverket till med att samla ihop de resultat som testfallen givit för att sedan presentera detta för utvecklaren.

Dessa ramverk tillhandahåller också, beroende på ramverk, funktionalitet för att styra testningen, till exempel att sätta samman tester och skapa testsviter. Dessa är klasser som har egenskapen att vara testklasser och en eller flera av dessa bildar just testsviter.

Det finns många olika ramverk för enhetstestning inom Javautveckling där de följande är de mest intressanta. Detta på grund av att de täcker stora områden och är inte specialiserade på precis en uppgift. De ramverk som behandlas i förstudien är:

- JUnit
- TestNG
- JTiger
- Unitils

I följande sektioner följer beskrivning av dessa.

## 3.2 JUnit

Den äldsta och mest använda ramverket där det också finns flera olika tillägg för att kunna få de funktioner som man önskar. Denna text fokuserar dock på användandet i grunduppförandet av ramverket. I grunduppförandet av JUnit finns det en enkel uppsättning av funktioner för att göra asserts. JUnit har också noteringar vid de olika testfallen för att kunna bestämma vad som är ett test och vad som är stödjande funktioner. Dessa funktioner bygger upp och river ner de miljöer som testet ska verka i genom att till exempel instansiera objekt och förbereda data. JUnit kan hantera exceptions som kastas genom att man gör en notering av att detta kommer att ske. Du kan i JUnit gruppera test med hjälp av testsviter för att kunna exekvera dessa tillsammans. Detta sker dock på klassnivå och inte på individuella testfall.

Nackdelar med JUnit är att det är väldigt grundläggande och saknar en del av de funktioner som andra ramverk har. Dock finns det tilläggsprogram för att kunna utöka JUnit med de funktioner man anser behöva.

## 3.3 TestNG

TestNG är ett testverktyg som bygger på JUnit. Det som skiljer mellan JUnit och TestNG är att TestNG kommer med mer funktionalitet som enligt dess utvecklare Cédric Beust[18] gör det mer kraftfullt, lättare att använda och är avsett enbart för Java. Det skapades enligt Beust för att han var frustrerad över de begränsningar som JUnit har och på det sätt han var tvungen att ta sig runt dem.

Det tillhandahåller de vanliga testfallen som JUnit har att erbjuda, det vill säga de vanliga Assertfunktionerna, och finns som plug-in till de flesta Java IDE, vilket gör det lätt att

installera. TestNG kan även exekvera JUnit tester, samt ta ett JUnit test som input och översätta det till TestNG och sedan exekvera det i sin tur. Detta på grund av att de bygger på samma grund och som nämnts är TestNG en vidareutveckling av JUnit.

TestNG går att styra med hjälp av en XML-fil där utvecklaren definierar reglerna för hur verktyget ska exekvera testerna och egenskaperna för varje test.

En klass behöver inte ärva egenskaper från TestNG för att få tillgång till dess funktioner utan det räcker med att importera biblioteket i projektet. För att TestNG ska kunna veta vad som är testfall använder det istället noteringar för att definiera tester eller subrutiner som har med testning att göra.

```
@Test
public void testCon() throws ClassNotFoundException, SQLException{
```

*Figur 4: Standardnotering*

Den viktigaste egenskapen som TestNG har är möjligheten att kunna gruppera test efter deras funktion och möjligheten att säga vilka tester som ska exekveras. Detta sker genom att skapa testgrupper, vilket bilden nedan illustrerar.

```
@Test(groups = {"all", "get"})
public void testGetNames() throws SQLException, ClassNotFoundException{
    testDB.queryAllNames();
}
```

*Figur 5: Gruppering*

Vid exekveringen av testningen kan utvecklaren sedan ange vilken testgrupp som ska exekveras. TestNG erbjuder även möjligheten att exekvera flera grupper samtidigt. En anledning till att utvecklare vill kunna välja vilka testfall som ska exekveras är att i stora system skulle det ta allt för lång tid att köra alla tester, vilket skulle påverka utvecklingen. Testfall som utvecklare vet fungerar är inte alltid nödvändiga att exekvera utan det är mer intressant att köra de som inte är okej.

TestNG klarar även av att parallellexekvera tester, vilket påskyndar test processen avsevärt. Ett sätt att tillämpa detta på är att använda gruppering och sedan köra testgrupperna parallellt.

Utvecklare som är beroende av viss data för att få giltiga testfall har möjlighet att använda datatillhandahållare(dataproviders), vilka är metoder som har till uppgift att ge testfunktioner eller testklasser värden som ska behandlas[18]. Dessa datatillhandahållare kan generera data från olika typer av källor, allt från databaser till hårdkodning. Det går att definiera valfritt

antal datatillhandahållare, vilket ger möjlighet att beroende på testfall ge dem olika data från olika källor.

Testverktyget erbjuder även möjlighet att skapa beroenden mellan testfall. Låt säga att en utvecklare ska skriva tester mot en databas. För att överhuvudtaget kunna arbeta mot databasen behöver utvecklaren vara uppkopplad mot den, vilket testfall med förfrågningar är beroende av. I TestNG kan man genom noteringar, säga att för att detta test ska få exekveras måste ett annat test vara godkänt. Detta gör det lättare för utvecklaren att hitta fel, om ett test skulle fallera. Då får utvecklaren reda på om det är det nya testet eller det test som den är beroende av som var orsaken till att det inte lyckades.

Alla testfall sparas sedan i två olika rapporter i XML-format, där det ena rapporterar vilka tester som fallerade och vilka som var godkända samt annan nyttig information, som till exempel hur lång tid varje test tog.

TestNG kommer med många attribut från början, vilket enligt Cédric Beust[18] ska underlätta för de utvecklare som använder sig av TDD. Utöver dessa attribut finns det plug-ins framför allt för att underlätta för utvecklare som använder databaser och även plug-ins för att skapa fuskobjekt, simulering av riktigt objekt. Dessa fuskobjekt är till för att till exempel simulera databaser.

Nackdelen som vi ser det med TestNG är just för att det kommer med väldigt många attribut kan det vara svårt för utvecklare att använda sig av det på ett optimerat sätt. Eftersom man har möjlighet att använda så pass mycket funktionalitet kan testsviterna bli mer komplicerade än vad de behöver vara.

### 3.4 JTiger

Detta är ett ramverk som bara finns till Java 1.5 och lägger vikten vid att kunna genomföra alla test till hundra procent. Det ska kunna genomföra allt från enkla assert-funktioner till avancerade klasser med arv. JTiger kan sedan generera rapporter i olika format med testresultaten.

Likt andra ramverk använder sig JTiger av setup- och teardownfunktioner[20] för att initialisera och för att riva testkörningen. Dock kan utvecklaren inte specificera när dessa ska exekveras utan JTiger gör det innan testklassen startas.

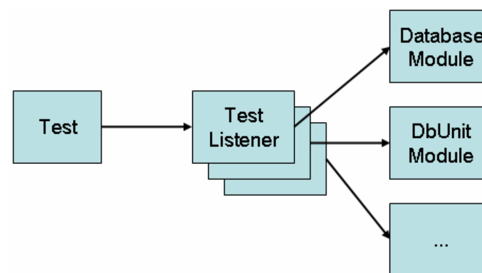
JTiger har en mer avancerad felrapportering än de flesta andra testramverk, då den kan generera sex olika tillstånd. Av dessa sex har tre av dem med felhantering att göra, då den säger antingen att det var fel i setup av testfunktionen eller i teardown av den eller om det var testet i sig som var orsaken till felet.

Ramverket, likt TestNG, använder noteringar för att kunna styra hur testfall ska exekveras, genom att gruppera testerna efter gruppnamn. I JTigers fall grupperas testfall efter klasser av testfall. Även likt TestNG kan utvecklare skapa beroenden mellan testfall och test avbryts om beroende testfall inte returnerar sant.

Nackdelen med JTiger är ett det inte är integrerat med någon IDE utan kräver manuell installation, vilket är betydligt mer arbetskrävande än JUnit eller TestNG. Dock på grund av att JTiger inte är integrerad i någon IDE leder det till att fel rapporteringen blir mer exakt då den olikt JUnit inte kan exekvera testfall som i sin tur exekverar funktioner som är fel.

## 3.5 Unitils

Detta ramverk är ett försök att hantera enhetstestningen på ett lättare sätt genom att bygga vidare på DBUnit, vilket är ett testramverk anpassat för databashantering, och integrerar med JUnit, TestNG, EasyMock[21], Spring[22], Hibernate[23] och Java Persistence API[24]. Detta projekt samlar de verktyg som behövs för att kunna testa olika saker inom ett ramverk. Nedan följer bara information om Unitils och inte de externa verktyg som används då deras funktionalitet inte beror på ramverket.




*Figur 6: Uppbyggnad*

Unitils är en utbyggnad av de befintliga ramverken som JUnit och TestNG och lägger där till sina egna testexekverare för dessa olika ramverk och inkluderar också tilläggsprogram för dessa som DBUnit och EasyMock. Du kan genom detta både använda dig av JUnit och TestNG för att köra dina tester eller välja att använda dig av Unitils egna exekverare för dessa ramverk. Unitils kommer också med lite mer avancerade asserts-metoder som assertReflectionEquals, som kan hantera objekt som innehåller flera attribut. Istället för att kontrollera att det bara är samma sorts objekt kontrolleras att alla fälten i objektet är likadana till exempel.

```
Person person = new Person("Kalle");
Person person2 = new Person("Börje");
```

*Figur 7: Testfall objekt*

Ovanstående skulle vid en assert ge att det var lika, dock skulle assertReflectionEquals inte godkänna detta på grund av att attributet namn inte är lika. Detta sker utan att utvecklaren behöver specificera vilket attribut som ska kontrolleras. Det går i samband med detta också bestämma hur kontrollen ska ske om den ska ta hänsyn till saknade värden eller default-värden.

Unitils bidrar också med funktioner för att lättare använda sig av de verktyg som det är integrerat med. Dessa är databastestning med hjälp av verktyget DBUnit och användning av

mockobjekt, objekt som simulerar det verkliga objektets beteende[27], med hjälp av verktyget EasyMock.

Ramverket har en modul för att kunna sätta in mockobjekt i andra objekt, genom att bestämma vilket attribut i den klassen som mockobjektet ska representera. Nedan följer kod för en insättning av mockobjektet personMock i klassen UserService´s attribut person.

```
@InjectInto(property="person")
private Mock<Person> personMock;

@TestedObject
private UserService userService;
```

*Figur 8: Mock objekt Unitils*

Nackdelar med Unitils är just det att den samlar allt på samma ställe och det är inte säkert att man vill använda sig av alla delar som Unitils beror på.

## 3.6 Krav

För fortsatt utvärdering ska vi fokusera på ett av ovan nämnda ramverk. För att kunna göra ett urval har vi satt upp ett antal egenskaper som vi anser vara viktiga för ramverken och som har godkänts av Palassogruppen. Kraven sattes med hänsyn till den utvecklingsmiljö som de använder och hur mycket extra arbete det blir för utvecklarna.

Krav på ramverk:

- Eclipse-kompatibelt
- Öppen källkod
- För utveckling i Java
- Snabb exekvering av test
- Skalbar
- Möjlighet att konfigurera testexekvering
- Lätt att installera i Eclipse
- Lätt att sätta upp testmiljön
- Lätt att använda
- Lättförståligt
- Möjlighet att generera rapporter
- Möjlighet att testa med databaser

Noteringar till kraven där behov finns:

Eclipse-kompatibelt: Då Palassogruppen till mycket stor del utvecklar med hjälp av detta IDE (Integrated Development Environment) så var det viktigt att det går att använda tillsammans.

Öppen källkod: Palassogruppen har som policy att använda sig av öppen källkod därför att det minskar kostnaderna för användningen.

För utveckling i Java: då Palasso i dagsläget utvecklas i Java ligger fokus att hitta verktyg för just det språket.

Skalbar: Verktyget ska kunna användas i det enklaste testfallet men med möjlighet för att användas med mer komplicerade funktioner.

Möjlighet att konfigurera testexekvering: att utvecklaren ska kunna bestämma hur testfallen ska exekvera i förhållande till ordning och vilka som ska köras.

Lätt att sätta upp testmiljön: detta innebär hur mycket arbete som behövs från det att man installerat ramverket tills dess att man kan börja skriva test. Är det lätt att sätta upp testmiljön går det snabbare för utvecklaren att komma igång med utveckling.

## 3.7 Diskussion och slutsats

För att få en bild av ramverken följer nedan en tabell över de funktioner som ramverken har:

| | JTiger | JUnit | TestNG | Unitils |
|---|---|---|---|---|
| Eclipse-kompatibel | x | x | x | x |
| Öppen källkod | x | x | x | x |
| För Java utveckling | x | x | x | x |
| Snabb exekvering av test | x | x | x | x |
| Skalbar | x | x | x | x |
| Möjlighet att konfigurera | x | | x | x |
| Lätt att installera i Eclipse | | x | x | |
| Lätt att sätta upp testmiljön | | x | x | x |
| Lätt att använda | | x | x | |
| Lätt förståligt | | x | x | x |
| Rapportgenerering | x | | x | x |
| Möjlighet att använda med databas | x | x | x | x |

*Tabell 1: Funktionalitet*

Då många av de kraven som finns med i kravlistan blir mötta av flera eller alla av ramverken har vi fått utöka kravlistan med mer specifika krav för att få en bra bild av vad som skiljer dem åt. Funktionalitet som valdes här gjordes för att den underlättar för utvecklarna att arbeta med ramverket. Detta illustreras i tabellen nedan.

| | JTiger | JUnit | TestNG | Unitils |
|---|---|---|---|---|
| **Assert-metoder** | x | x | x | x |
| **Enkel gruppering** | x | x | x | x |
| **Beroenden mellan test** | x | | x | x |
| **Avancerad gruppering** | x | | x | x |
| **Dataförsörjare** | | | x | x |
| **Differentierad felrapportering** | x | | x | x |
| **Parallell exekvering av test** | | | x | x |
| **Hantering av ”Exceptions”** | x | x | x | x |

*Tabell 2: Mer funktionalitet*

Som tabellen ovan illustrerar finns det flera av ramverken som har liknande funktionalitet.

Av dessa verktyg valdes det att fokusera på TestNG i fortsättningen, detta på grund av att den innehåller flest egenskaper och har ett mycket enkelt gränssnitt. Funktionaliteten som följer med TestNG är mycket omfattande, dock är det inte för utvecklaren ett måste att använda sig av allt. Tack vare att det är skalbart är det enkelt att använda och komma i gång med.

JUnit, som är en standard för enhetstestning idag, anser vi inte har tillräckligt med funktionalitet, dock är det väldigt enkelt att använda och installera då det är förinstallerat i flera Java IDE:er. Dock väger bristen på funktionalitet störst och kan inte anses vara tillräckligt i jämförelse TestNG.

Egenskaperna som JTiger har motsvarar mångt och mycket TestNG, däremot är installationen och uppsättningen väldigt komplicerad i jämförelse med alla andra testverktyg. Därför faller detta verktyg bort.

Unitils är ett jämbördigt verktyg till TestNG, då Unitils använder sig av både TestNG och JUnit. Däremot är de egenskaper som Unitils tillför själv inte någon utveckling av dessa två verktyg. Då Unitils är en samling befintliga verktyg ser vi inte någon anledning att använda sig av det, eftersom vi anser det mer viktigt att kunna skräddarsy verktyget efter det behov som finns.

# 4 Experiment-översikt

Det vi kommer att göra är att försöka anpassa Palasso för testning, då det i dag inte är skrivet på ett sådant sätt att det är särskilt testbart. Vi kommer att genomföra detta genom att använda oss av två olika metoder. Testningen kommer att ske genom att vi använder oss av TestNG som testramverk.

## 4.1 Palassosystemet

Palasso är ett personal- och löneadministrationssystem. Det har till uppgift att sköta löneuträkningar och -utbetalningar samt även personaladministration, som anställning, ledigheter, arbetsuppföljning m.m. system är marknadsledande för den statliga sektorn. Ungefär 200 000 människor får sin lön genom Palasso varje månad.

Systemet består av ett antal fristående moduler som var och en har en viss funktionalitet. Modulerna är sammankopplade genom en portal och i botten ligger en databas, för mer läsning se Palassosidan på Logica[25].

## 4.2 Systemuppbyggnad

I dag är Palasso skrivet i flera olika språk Cobol, New Era, Visual Basic och Java. Det sker en emigration till att enbart använda Java som kommunicerar med en relationsdatabas. De håller även på att gå ifrån tjocka klienter till tunna, vilket innebär att de flyttar all logik från klienten till server-sidan. Vi kommer enbart att fokusera på Java delen, vilket är den stora delen i systemet.

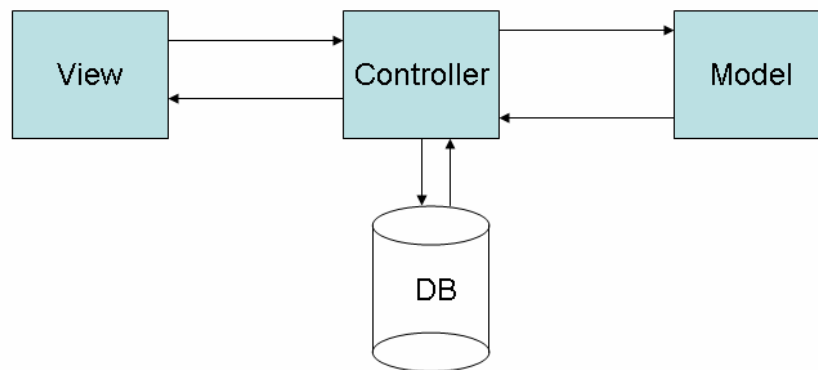
Systemet har en väldigt komplex uppbyggnad, då det har blivit utvecklat under mycket lång tid. Komplexiteten ligger i uppbyggnaden av systemet, då varje modul ärver alla sina egenskaper från superklasser, som i sin tur ärver många egenskaper från klasser ytterligare en nivå upp i klasshierarkin. Storleken på Palasso och strävan att få alla moduler enhetliga leder även det till att systemets uppbyggnad ökar i komplexitetsgrad.

## 4.3 Moduluppbyggnad

Palasso består av moduler, som nämnts ovan. Varje modul fyller en viss funktion till exempel ledighet, löner.

Modulerna är uppbyggda på flera olika sätt. De två största är MVC (model view Controller) och klient/server, vilket illustreras av bilderna nedan.

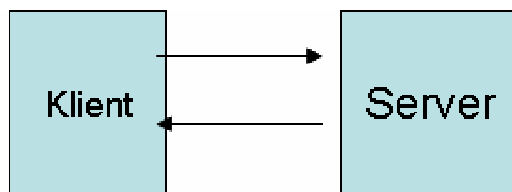
### 4.3.1 MVC




*Figur 9: MVC*

MVC är det designmönster som Palasso satsar på att införa i hela systemet och ska vara den arkitektur som används. Den består av en kontroller(controller) som skapar en vy(view) och en modell(model). Vyn är en representation av modellen. En vy är bara ett användargränssnitt som inte har någon egen logik. Det den gör är att skicka sitt tillstånd till kontrollen som uppdaterar modellen och även uppdaterar objekt som är beroende av modulen. I Palasso sköts även databasuppdateringar av controller objektet. Detta skiljer från den formella bilden av MVC, där modellen uppdaterar databasen.

### 4.3.2 Klient/Server

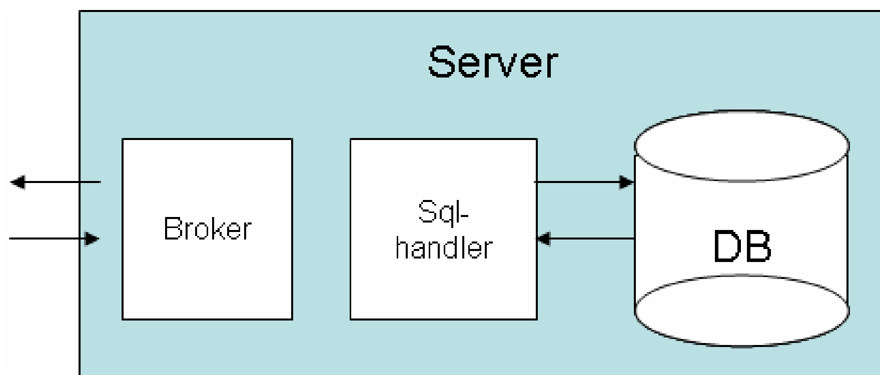



*Figur 10: Klient/Server*

Denna arkitektur är en ren Klient/Server. Klienten är det som användaren har tillgång till och kan både vara tunn och tjock beroende på fall. De fall som vi använder är tunna, då det är den lösning som Palasso strävar efter.

En tunn klient är en klient som inte har någon logik utan bara skickar händelser vidare till en server. Servern är det objekt som innehåller logik och som utför operationer. En tjock klient innehåller egen logik och skickar färdiga resultat till servern.

I Palasso är klienten ett GUI (grafiskt användargränssnitt) där användaren matar in data som sedan skickas till servern. Servern i Palasso är uppbyggd enligt bilden nedan.




*Figur 11: Palasso server*

Palasso-servern består av tre delar: Broker, sqlhandler och en databas. Brokern är den del som sköter kontakten mellan klienten och servern. Det är en form av interface, som visar vilka funktioner som finns tillgängliga på servern.

SQLhandler är den del som sköter kommunikationen mellan databasen och resten av systemet. Den gör det genom att använda ren SQL, vilket är ett frågespråk avsett för databashantering, och även ett eget utvecklat ramverk kallat Butler.

Butler har till uppgift att istället för att returnera tabeller som inte är typdefinierade, vilket SQL förfrågningar gör, returnera objekt som representerar specifika tabeller i databasen med alla relationer som den har. Butler är den metod som huvudsakligen används idag.

## 4.4 Inledning till experiment

I dag är Palasso-koden väldigt svår att enhetstesta, då den är uppbyggd med klasser som har mycket beroenden i sig. Varje klass ärver från superklasser som i sin tur ärver egenskaper från klasser högre upp i hierarkin. Detta skapar problem då dessa superklasser inte är testade. De ärvda funktionerna kan vi inte vara säkra på att de används överhuvudtaget då olika moduler använder sig av de ärvda egenskaperna på olika sätt, till exempel vissa metoder skrivs över, andra inte i subklasserna. Då beroendena är tvetydiga är det svårt att se hur arvet används och bidrar till att försvåra testningen.

Beroendena är väldigt hårt bundna till varje metod i klasserna. Detta gör även det väldigt svårt att testa klasserna i isolation, vilket är det vi vill göra när vi enhetstestar.

På grund av dessa argument måste vi, för att kunna visa hur testning ska ske, även visa hur utvecklare bör skriva för att få testbar kod. Detta kommer vi att försöka visa med två olika metoder.

## 4.5 Förklaring av metoder

Det första sättet vi kommer att göra det på är att vi väljer en modul och skriver om den från början mot en annan databas. Här används TDD se kapitel 2.2.3. Detta för att på ett enkelt sätt ska kunna visa på hur en modul ska kunna skrivas med fokus enbart på funktionaliteten. Genom att ta bort resten av Palassologiken från en vald del av en viss modul förenklas uppbyggnaden. I de klasser som det är tänkt genomföra denna metod på, fyller inte beroendena som finns till Palasso någon större funktion och går att simulera.

De klasser som är bra kandidater till ovan nämnda tillvägagångssätt är de moduler som följer klient/server-modellen, och just server-delarna i dessa moduler är särskilt bra lämpade för detta. De är lämpade för att de har tydliga begränsningar i ansvar och det är lätt att se flödet i dem.

Den andra metoden som vi har tänkt använda oss av är att skriva om befintlig kod i Palasso, utan att extrahera den från systemet. Detta kommer vi att göra på moduler som följer MVC-mönstret.

Anledningen till att vi vill visa hur man eventuellt skulle kunna ändra i befintlig kod utan att skriva om den från början, med avseende på testning, är att bevisa att det går att införa enhetstestning utan några större ändringar i systemet.

# 5 Experiment - Metod 1

## 5.1 Översikt

I denna del kommer vi att visa hur programvaruutveckling med hjälp av TDD kan ske och hur utvecklarna av Palasso kan använda enhetstestning för att säkerställa kodkvalitet. Detta genom att ge exempel på vilka fall som är viktiga att testa och vilka som inte är det samt visa hur de kan gå tillväga för att skapa testfall och testmiljöer som genererar så säkra testsvar som möjligt.

Metoden för att visa detta kommer att gå ut på att ta en modul ur Palasso och göra en enklare form av den genom att använda TDD och enhetstestning. Detta visar hur enhetstestning används på det mest effektiva sättet och visar vikten av att ha allt i ett system testat. Applikationen som kommer att utvecklas kommer att följa samma struktur som Palasso dock i en enklare form då den i detta fall inte är nödvändig för att visa vikten av testbar och testad kod. Den modul som har blivit vald är Organisation då den är den enklaste uppbyggd och inte har många beroenden.

Testerna som kommer att utföras kommer att bli en blandning av rena enhetstester och integrationstestning. Anledningen till integrationstestningar är att mycket av funktionaliteten ligger i kommunikationen mellan applikationen och databasen. Det är mycket enklare att testa kopplingen och förfrågningar genom att anropa databasen istället för att mocka ut den. Mocka, skapa mock objeckt se kapitel 3.5, ut databasen skulle i detta fall vara för komplext för ändamålet.

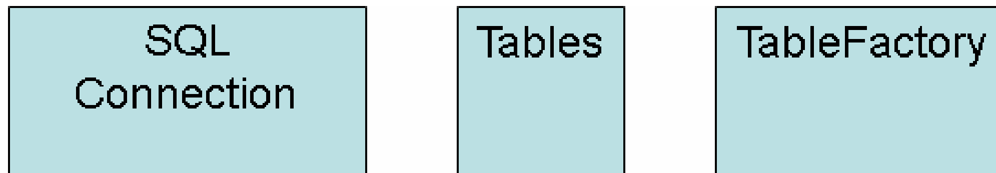
## 5.2 Design

Systemet kommer att bestå av två stora delar. Den ena är DatabaseHandler, som namnet antyder sköter databas-funktionaliteten. Den andra delen är varje modul, vilket är funktionaliteten och gränssnittet för användaren mot ett visst område i databasen, till exempel en person-tabell, organisation-tabell eller en samling av tabeller.

DatabaseHandler kommer att bli en form av Palassos ramverk Butler, vilket generar tabeller från databasen som objekt. Butler är ett eget utvecklat databashanteringssystem likt linq[26] i C#.

### 5.2.1 DatabaseHandler

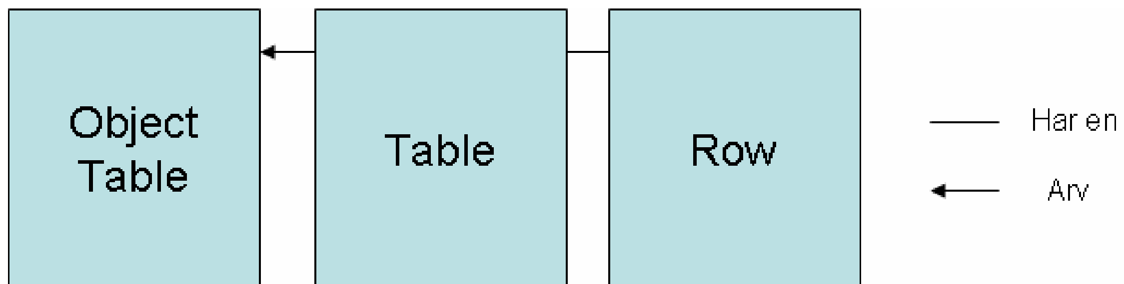
Denna del består av tre huvuddelar, vilket bilden nedan illustrerar.




*Figur 12: DatabaseHandler*

SQLConnection har till uppgift att tillhandahålla uppkopplingar mot olika typer av databaser. Den vet hur den sätter upp, tillhandahåller och stänger en uppkoppling.

Tables har till uppgift att skapa, hämta och sätta objekt som representerar en relation i en databas. Varje tabell är uppbyggd genom att ärva en grundtabell, vilket är en container för data. För varje tabell i databasen finns motsvarande klass i applikationen. Dessa klasser vet hur data läses från och skrivs till databasen och uppdaterar sig själv beroende på vad som passerar den. Bilden nedan visar hur tabellklasserna är uppbyggda.

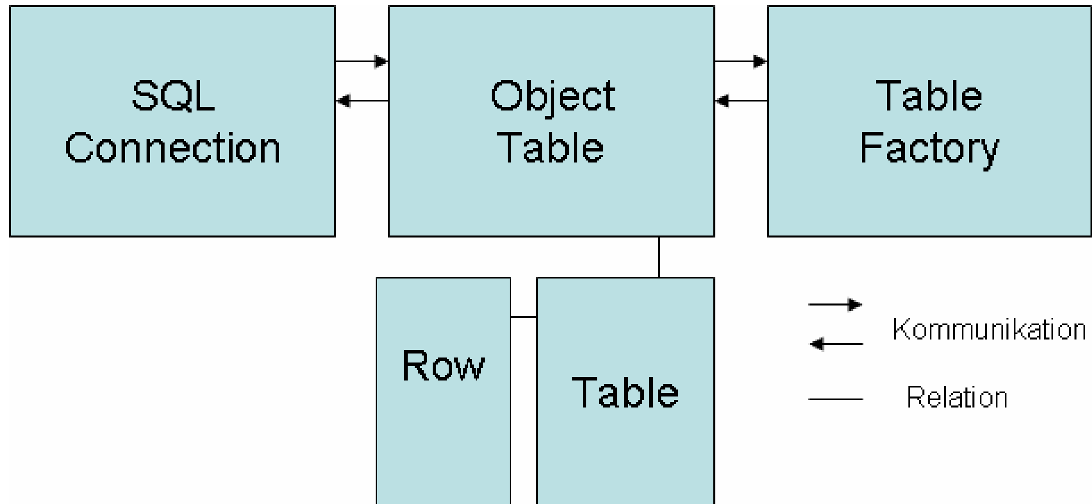



*Figur 13: Arvstruktur för Object table*

Kommunikationen mellan databasen och tabellklasserna sker genom SQL-frågor.

TableFactory har till uppgift att tillhandahålla tabeller. Den gör det genom att en modul frågar efter en viss tabell och beroende på om den finns eller inte returneras tabellen som frågades efter. Om den inte har den returneras null.

Flödet i DatabaseHandler ser ut som följer




*Figur 14: DatabaseHandler flöde*

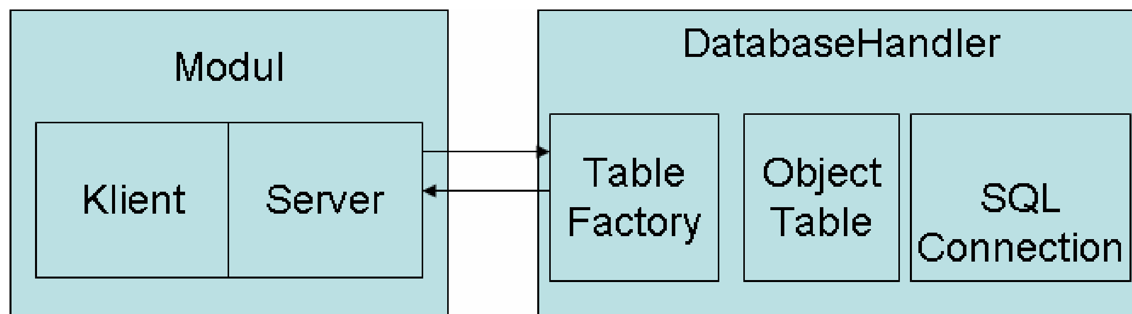
### 5.2.2 Modul

I Palasso är en modul uppbyggd enligt designmönstret Klient/Server eller MVC. Denna del kommer att fokusera enbart på klient/server.

I en sådan design sköter klienten interaktionen med användaren, ritar upp GUI och tar hand om händelser från det. Logiken, alltså beräkningar och annan manipulation av data, sköts av servern för att avlasta klientdatorerna och för att få logiken centraliserad vilket underlättar underhållet av systemen.

I denna metod kommer bara server-delen att utvecklas då brist av tid och GUI-tester inte omfattas av denna avhandling.

I systemet kommer manipulationen av data i tabellobjekten att skötas av servern som sedan uppdaterar databasen genom objekten. Flödet genom systemet kommer se ut enligt bilden nedan.




*Figur 15: Flöde i applikationen*

### 5.2.3 TDD

För att verkligen visa fördelarna med enhetstestning kommer utvecklingen att ske genom TDD, Test Driven Development. Testfallen kommer alltså att skrivas först och sedan implementationen för att klara testet. Studier har visat att utveckling genom TDD genererar mindre onödig kod, just för att implementation inte sker förrän det har visats att den är nödvändig. Enhetstestningen och integrationstestningen kommer att ske genom att använda ramverket TestNG, kapitel 3.

Beskrivningen kommer att ske genom förklaringar av klasser och deras funktionalitet samt testfall där det är intressant att visa hur och varför testfallen fungerar och behövs.

## 5.3 Utveckling

### 5.3.1 SQLConnection

Databasen som används är en MySQL-databas, en öppen källkodsdatabas[30], vilket innebär att uppkopplingen kommer att behöva egenskaper för den. Klassen MYSQLCon kommer att tillhandahålla just en sådan uppkoppling. Den första egenskapen klassen behöver är att kunna etablera en uppkoppling mot databasen. Detta kan ske på två sätt, antingen sätts egenskaperna i konstruktorn och så har objektet dem under hela sin livslängd eller så sätts de innan varje uppkoppling. Genom att sätta dem innan varje uppkoppling blir uppkopplingen mer flexibel och mer anpassningsbar. I denna metod kommer andra varianten att användas.

När klassen MYSQLCon skapas kommer den att skapa en tom uppkoppling, dvs konstruktorn kommer att skapa en Connection som är null. Sedan när vi sätter egenskaperna kommer den inte att vara null utan antagit ett värde. Värdet spelar ingen roll utan allt som behövs är att den inte är null.

Detta testfall kommer alltså först anropa konstruktorn där vi förväntar oss att Connection är null. Detta innebär att vi måste skapa två funktioner, konstruktor och en funktion för att hämta uppkopplingen. Testfallet för detta ser ut som följer:

```
@Test
(groups = "konstruktor")
public void construct(){

    MYSQLCon con = new MYSQLCon();
    Assert.assertNull(con.get());

}
```

*Figur 16: Konstruktor-test*

I detta stadium kommer constructor inte att göra något och get-funktionen kommer att returnera null.

```
public class MYSQLCon{

    public MYSQLCon(){}

    public Connection get(){
        return null;
    }
}
```

*Figur 17 Inledande klass*

Testet gick igenom och utvecklingen kan fortsätta. Nästa steg är att skapa en set-funktion som skapar uppkopplingen. Testfallet för detta ser ut som följer:

```
@Test
(groups = "set")
public void construct(){

    MYSQLCon con = new MYSQLCon();
    con.set();
    Assert.assertNotNull(con.get());

}
```

*Figur 18 Inledande test*

Set-funktionen ser ut som följer:

```
public void set() throws ClassNotFoundException, SQLException{

    Connection mycon = null;
    if(mycon == null){

    Class.forName("com.mysql.jdbc.Driver");

        mycon = DriverManager.getConnection("jdbc:mysql:///pal",

        "***", "***");

    }

}
```

*Figur 19 Metod före omstrukturering*

Exekveringen av testfallet godkändes inte eftersom get() returnerade null. Genom att använda omstrukturering kommer klassen att ändras, vilket visas på nästa sida.

```
public class MYSQLCon {
    public MYSQLCon(){
        mycon = null;
    }
    public void set() throws ClassNotFoundException, SQLException{
        if(mycon == null){
            Class.forName("com.mysql.jdbc.Driver");
            mycon = DriverManager.getConnection("jdbc:mysql:///pal",
                    "***", "***");
        }
    }
    public Connection get(){
        return mycon;
    }
    private Connection mycon;
}
```

*Figur 20 Klass efter omstrukturering*

Efter att testet har exekverats godkänns det och utvecklingen kan fortsätta. Det som behövs nu är att kunna stänga uppkopplingen när den inte används. För att testa detta sätts det upp en uppkoppling och sedan hämtas den och kontrollerar att den inte är null. Efter det stängs den och till sist kontrolleras att den är null.

```
@Test(groups="set")
public void closeWithSet() throws ClassNotFoundException, SQLException{
    MYSQLCon con = new MYSQLCon();
    con.set();
    Assert.assertNotNull(con.get());

    con.close();

    Assert.assertNull(con.get());
}
```

*Figur 21 Test av stängningsfunktion*

```
Public void close(){
    if(mycon != null)
        mycon.close();
    mycon = null;
}
```

*Figur 22 Stängning av socket*

Testet gick igenom och MYSQLCon är klar, funktionaliteten som behövs av klassen finns nu och är testad, vilket innebär att det är säkert att den fungerar. Eftersom den är testad är det fritt fram för andra klasser att använda den och ärva från den, eftersom det nu går att säkerställa att fel som uppkommer inte beror på denna klass.

### 5.3.2 Row

Denna klass är en container-klass för en rad i en tabell, den skapar alltså ett objekt av en tabellrad. Den är generisk, det vill säga att den kan lagra ett obestämt antal kolumner i tabell. Den struktur som har valts att använda är ArrayList som kan lagra Object. ArrayList är en array som har egenskaper för att hämta, sätta och jämföra med andra ArrayList eller objekt. Object-noder används för att kunna lagra vilken typ av data som helst i strukturen.

Det första som gjordes vara att skapa ArrayListan och en räknare som räknade antalet kolumner.  I konstruktorn skapas en ny tom ArrayLista och sätter storleken till 0. Sedan skapades funktionalitet för att lägga till ett objekt i raden och hämta raden. Testningen av att lägga till ett objekt gjordes med dataproviders, vilket är en funktionalitet som finns i TestNG och den tillhandahåller data till en funktion. Testfallet för att lägga till ser ut som följer:

```
@DataProvider(name = "addprov")
public Object[][] addprovider(){
    return new Object[][]{ {"Hålger"}, { "54223"}, {"Karlstad vägen 2"}, {"054000000"}};
}

@Test(groups ="add", dataProvider = "addprov")
public void Add(String namn){
    int num = row.columnCount();
    row.add(namn);
    Assert.assertEquals(row.columnCount(), num+1);
}
  Row row;
```

*Figur 23 Test med dataprovider*

Eftersom row är en lista som innehåller objekt definieras en egen eqauls-funktion. Denna funktion tar ett objekt och jämför klassens värde med det värde som den får in. Det detta objekt behöver göra är att jämföra data mot klassens data men även kontrollera objektets rad mot en annan rad. Funktionen ser ut som följer.

```
@Override
public boolean equals(Object o)
{
    try{
        Row ar1 = (Row)o;
        if(toString().compareTo(ar1.toString()) == 0)
        return true;
    } catch(Exception ex){
        for(Object ro: row){
            if(ro.toString().compareTo(o.toString()) == 0)
                return true;
        }
    }
    return false;
}
```

*Figur 24 Equals-funktion*

Testningen av denna funktion sker helt enkelt genom att fylla upp den med värden och göra 4 assert-anrop. Första för att kontrollera att object finns, sedan att lista finns och till sist om varken objet eller lista finns.

```
@Test(groups = "equal")

public void EQ(){
    Row t = new Row();
    ArrayList<Object> a = new ArrayList<Object>();
    a.add("Hej");
    a.add("Bengt");
    t.add(a);
    Assert.assertTrue(row.equals(t));
    // elementen i listan är samma
    Assert.assertTrue(row.equals("Bengt"));
    Assert.assertFalse(row.equals("då"));
    Assert.assertFalse(row.equals(new ArrayList()));
}
```

*Figur 25 Test equalfunktion*

Row-klassen är enkel. Allt den kan göra är att returnera och tilldela data den har. Dessa klasser är inte nödvändiga att enhetstesta, då dessa funktioner är väldigt elementära och enkla att göra. Dock är det intressant att testa equal och andra jämförelsefunktion om dessa finns i en klass. Dessa funktioner besitter mer logik som kan vara felande.

### 5.3.3 Table

Table är likt Row en klass som bara tilldelar och returnerar data som den har. Funktionaliteten den har är nästan exakt lika som Row. Det som skiljer den åt är att dess ArrayList består av Row, vilket gör att Table blir en lista av listor. Den kan lagra ett obestämt antal av Row. Testningen av den tillför inga nya kunskaper utan den har skett exakt som för Row.

### 5.3.4 ObjectTable

ObjectTables är klasser som ärver från Table. Det de har som uppgift är att sätta, hämta och uppdatera data i sin instans av Tableklassen från databasen.

Av dessa ObjectTables skapas OrgTable vilket är en klass som kommunicerar med tabellen orgtab i databasen.

Den första funktionalitet som läggs till är för att lägga till en post. Det kommer att fungera så att vald data läggs in i databasen och sedan uppdateras Table-instansen. Det första som måste göras är att skapa en SQL-sträng som talar om var data ska lagras och vilken data som hör till vilket attribut.

```
@Test(groups = "add_tests")
public void addStatmentTest(){
    Row newRow = new Row();
    newRow.add("Coca Cola"); //företags namn
    newRow.add("Cola gatan 2"); // address
    newRow.add("8877-2211"); //org nummer
    String sql = new OrgTable().createAddStatment(newRow);
    Assert.assertTrue(sql.startsWith("insert "));
    Assert.assertTrue(sql.endsWith(";") );
    System.out.println(ot.addStatement(newrow));
}
```

*Figur 26 Test statement*

TestNG klarar att hantera utskrifter i testramverket till skillnad från JUnit.

När SQL-strängen är testad och visar sig korrekt är det dags för att skapa funktionen för att lägga till data i databasen. För att kommunicera med den används MYSQLCon, vilken redan är testad och fungerande.

Första steget är att etablera uppkopplingen, genom att anropa set-funktionen i MYSQLCon. Sedan anropas get() för att hämta ett Connection-objekt och som sedan skapar ett statement genom createStatement. Det statement som ges används sedan för att uppdatera relationen genom executeUpdate med addStatement som parameter, vilket är SQL-strängen. När det är klart anropas stängningen av Statement och MYSQLCon.

Eftersom MYSQLCon är testad och funktionaliteten är säkerställd behövs det inte testas något specifikt här utan allt som behövs är att veta att funktionen körs utan att kasta undantag, execeptions.

```
Test(groups = "add_tests")
public void addtest(){
        Row newRow = new Row();
        newRow.add("Coca Cola"); //företags namn
        newRow.add("Cola gatan 2"); // address
        newRow.add("8877-2211"); //org nummer
        new OrgTable().add(newRow);
}
```

*Figur 27 Add test*

Testet går igenom. Det som är en viktig del i TDD är att alltid kontrollera om det går att strukturera om koden. En tumregel är att inte ha samma funktionalitet på två olika ställen. Detta gäller både för klassen som implementeras och för testklassen. Det som går att göra i testklassen är att flytta ut skapandet av OrgTable och Row till en setup-funktion. Det är en funktion som körs innan en Testsvit, testgrupp, metod eller klass. Bilden på nästa sida visar hur klassen ser ut efter omstrukturering.

```
public class TestOrgTable {
    @BeforeGroups("addStatement", "add")
    public void setUpAdd()
        throws ClassNotFoundException, SQLException
    {
        ot = new OrgTable();
        newrow = new Row();
        newrow.add("Cola AB");
        newrow.add("Cola vägen 2");
        newrow.add("666");
    }
    @AfterGroups("addStatement", "add")
    public void teardownAdd()
        throws ClassNotFoundException, SQLException
    {
        ot.remove(newrow);
        newrow = null;
        ot = null;
    }
    @Test(groups = "addStatement")
    public void testAddStatement(){
        String sql = new OrgTable.createAddStatement(newrow);
        Assert.assertTrue(sql.startsWith("insert "));
        Assert.assertTrue(sql.endsWith(";"));
        System.out.println(sql);
    }
    @Test(groups = "removeStatement")
    public void testRemoveStatement(){
        String sql = new OrgTable.createRemoveStatement(newrow);
        Assert.assertTrue(sql.startsWith("delete "));
        Assert.assertTrue(sql.endsWith(";"));
        System.out.println(sql);
    }
    @Test(groups = "add")
    public void testAdd()
        throws ClassNotFoundException, SQLException
    {
        ot.addObjectRow(newrow);
    }
    OrgTable ot = null;
    Row newrow = null;
}
```

*Figur 28 Testklass efter omstrukturering*

Remove lades även till i OrgTable som tar bort den skapade raden ur databasen. Denna funktion ska exekveras efter att testerna har körts, vilket händer om man använder AfterGroups.

Testningen av get-funktioner sker på liknande sätt som för add. Det första som testas är SQL-strängen. Det som skiljer är dock att testa att det är rätt data som returneras. Detta är det som blir en form av integrationstestning. Nu spelar det roll vad applikationen kopplar upp mot och vilken relation samt vilka attribut som ska returneras, klassen beror alltså på en annan del av systemet. För att testa detta är det viktigt att säkerställa att databasens tillstånd är exakt samma vid varje exekvering.

Ett sådant testfall visar exemplet nedan.

```
@Test(groups = "removeAll")
public void removeAllTest()
    throws ClassNotFoundException, SQLException
{
    ot.removeAll();
    Assert.assertEquals(ot.getAll().size(), 0);
}

@Test(groups= "getall", dependsOnGroups="removeAll")
public void getAllTest(){
    testadd();
    ArrayList<Object> orglist=ot.getAll();
    Assert.assertEquals(orglist.size(), 1);
}
```

*Figur 29 Beroende testfall*

Det sätt som visas här för att säkerställa tillståndet är att använda dependsOn-funktioner, vilket är en del av TestNG-ramverket. Det dependsOn gör är att säga att denna eller dessa funktioner ska köras innan denna funktion. Genom att säga att testfunktion är beroende av removeAll-gruppen, kommer databasen vara tömd innan den körs eller om removeAll inte går igenom kommer inte getAllTest att köras alls. Efter att den är tömd är det bara att lägga in data som behövs för testet och på så sätt säkerställa att testet går rätt till.

### 5.3.5 Server

Server-delen har logiken, det vill säga att den bearbetar data och sköter uträkningar. Denna del är den som är lättast att köra enhetstester på, då den är beroende av redan testad kod. Det den gör är bara att behandlar data och returnera resultatet upp eller ner i systemhierarkin. Testerna här blir tester av logik, vilket är enkla assert-funktioner.

# 6 Experiment - Metod 2

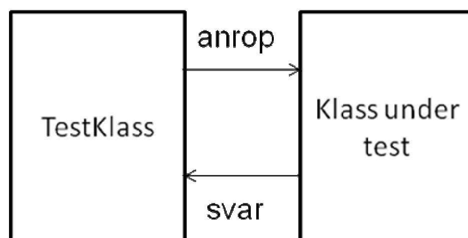
## 6.1 Tekniker för att få test på plats

När man ska införa test på redan skriven kod kan det uppstå problem. Ett exempel på det kan vara att koden inte går att köra utan att vissa förutsättningar finns.

Kod som tros vara lätt att testa i isolation kan i verkligheten vara fylld av beroenden på klasser som man inte hade en aning om innan man började skriva testen. För att kunna köra koden i isolation måste en del eller alla beroenden antingen skrivs om,(låtsas objekt) eller tas bort. I samband med detta har vissa tekniker visat sig vara effektiva för att få tester på plats. Nedan följer beskrivningar av de som används flitigast i detta experiment. Dessa tekniker och många fler finns att läsa om i Michael C. Feathers bok[17].

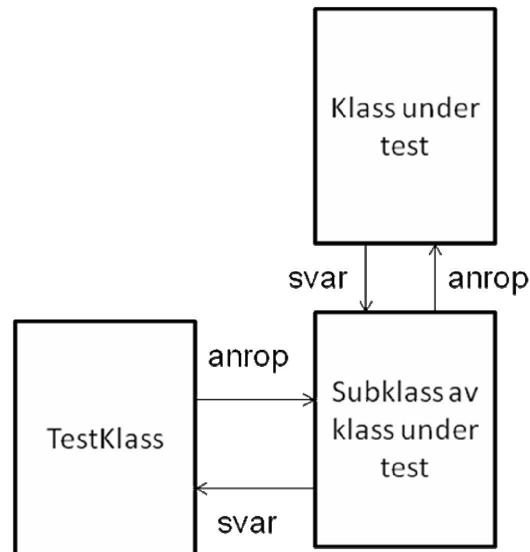
### 6.1.1 Införa subklass för testning

Vanligtvis brukar man använda sig av de publika metoderna som finns i en klass för att testa koden. I vissa fall visar det sig dock vara svårt att göra detta. Till exempel när man vill testa en metod längre ner i anropshierarkin än just de publika metoderna. För att lösa detta och kunna komma åt metoderna från ett test kan det visa sig vara nödvändigt att göra en subklass av klassen som ska testas. Sedan kan man göra anropen till superklassen genom subklassen då subklassen har större befogenheter att göra anrop till superklassen. Nedan följer bilder på de olika sätten att anropa metoder i klasser för testning.




*Figur 30 Normal testning*

Vanlig kommunikation mellan testklass och klass under test




*Figur 31 Införd subklass för test*

Kommunikation mellan testklass och klass under test genom en subklass av klass under test.

### 6.1.2 Extrahera och skriva över anrop

Detta är en av de tekniker som har använts mest under detta experiment. Detta för att det är ett lätt och effektivt sätt att begränsa vad som ska köras i ett test. Detta uppnås genom att skriva över med en metod som inte behöver returnera något och/eller inte innehåller någon kod. Men också inför möjligheten att få in testdata till metoderna. Ett exempel på detta kan vara kod som ser ut som nedan:

```
public int getNumberOfBooks(){
    BookList bookList = BookShelf.getInstance().getAllBooks();
    return booklist.size();
}
```

*Figur 32 Exempel på problem*

Som man kan se på denna kod använder sig metoden sig av en Singleton-klass, klass som har ett globalt tillstånd[29], för att få tag i bookList. Denna slutsats kan dras av det statiska anropet till getInstance(). Om man inte kan sätta instansen av denna Singleton så blir denna kod mycket svår att testa. Lösningen på detta blir att flytta ut Singleton-anropet till en egen metod. Därefter skrivs denna metod över i en subklass som istället ger tillbaka en fakeVariant av en bookList som sedan kan användas i körningen av koden. Resultatet för koden ovan blir:

```
public int getNumberOfBooks(){
    BookList booklist = getListOfAllBooks();
    return booklist.size();
}

protected BookList getListOfAllBooks(){
    return BookShelf.getInstance().getAllBooks();
}
```

*Figur 33 Ändrad metod till möjlighet att skriva över*

Där getListOfAllBooks sedan skrivs över i subklassen till:

```
public BookList getListOfAllBooks(){
    return fakeBookList;
}
```

*Figur 34 Metod som skriver över*

Denna utflyttning av beroende och ansvar kan också hjälpa till att minska duplicering i koden då fler metoder kan använda den skapade metoden.

### 6.1.3 Parameteriserar konstruktorn

Ett annat vanligt problem är instansiering av objekt inne i en konstruktor. I fallet nedan visas ett exempel av detta. För att kunna testa hanteringen av denna Handler vill vi ersätta den med ett fakeobjekt. Dock är det en privat medlem av klassen så det går inte att komma åt den. Detta löses genom att man lägger till en till konstruktor som tar Handlern som en parameter.

```
public void Server(){
    myHandler = new RequestHandler();
}
```

*Figur 35 Utan möjlighet till dependency injection*

Detta ändras till:

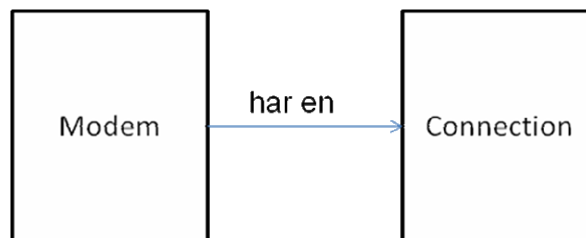
```
public void Server(){
    this(new RequestHandler());
}
public void Server(Handler handler){
    myHandler = handler;
}
```

*Figur 36 Med parameteriserad konstruktor*

Denna ändring ökar möjligheten till att testa klassen. Genom införandet av den andra konstruktorn blir det möjligt att skicka in en handler vid testning. Dock påverkar detta inte de klasser som använder sig av den gamla konstruktorn då de anropen fungerar precis som vanligt. Detta är ett klassiskt fall av möjligheten av dependency injection, där klassen beror på de objekt som man kan skicka in i objektet[28].

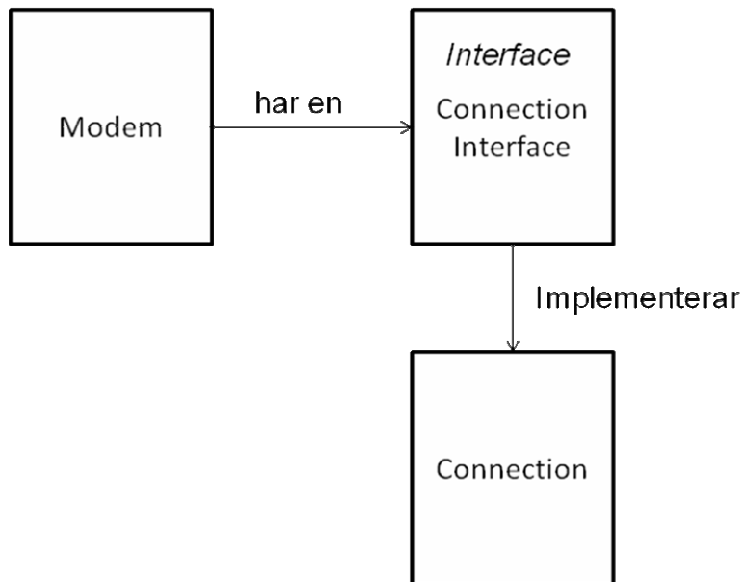
### 6.1.4 Extrahera interface

Ett vanligt problem när klasser ska testas är beroenden till konkreta klasser, då det är svårt att byta ut dessa klasser till fakeobjekt för att kunna genomföra test. Det man då kan göra är att extrahera ett interface till den klassen som ställer sig i vägen för testet. Detta kan man göra på två olika sätt. Antingen skapas först en klass som döps till t ex problemklassInterface som innehåller alla de metoder som problemklassen har. Eller så ändras den konkreta klassen till ett interface av samma namn som klassen tidigare hette. Därefter får en konkret klass implementera detta interface. Detta ger lite olika fördelar. Skapas ett nytt interface med ett nytt namn sätter det kravet på klassen att ändra alla typreferenser till denna nya typ. Detta är inte nödvändigt med senare versionen, där det istället blir skapandet av objektet som kommer att behöva ändras. Nedan följer bilder som illustrerar de olika fallen. Klassen Modem är beroende av den konkreta klassen Connection. Struktur från start:



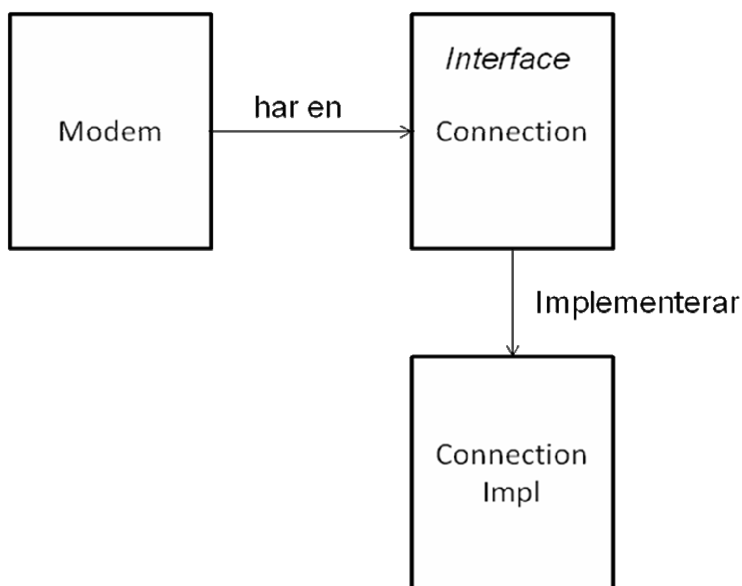

*Figur 37 Exempel på beroende*

Metod 1. Skapar ett interface med nytt namn.




*Figur 38 Införande av interface med nytt namn*

Metod 2. Skapar interface med den konkreta klassens gamla namn.




*Figur 39 Införande av interface med konkreta klassens gamla namn*

Dessa ändringar ger fördel om Modem skulle behöva utvecklas till att ta olika sorter av Connection. Då behöver de nya sorterna bara implementera interfacet och inga ändringar behövs göras i Modem. Andra fördelar med detta är att Modem-klassen inte behöver kompileras varje gång som ändringar sker i den konkreta klassen så länge som interfacet inte ändras.

### 6.1.5 Skapa och använda fakeobjekt

Det finns ett flertal olika ramverk för att skapa och verifiera fakeobjekt eller Mockobjekt. Dessa ramverk är lätta att använda och bidrar till att minska jobbet vid testning. Dock är dessa specialiserade på att skapa objekt efter ett interface. Detta blir ett problem när det inte finns ett interface som kan användas. Då kan till exempel extrahera interface vara ett effektivt alternativ. Dock finns det också tillfällen då detta inte heller är lämpligt. Exempel på detta kan vara när objekt redan har en arvstruktur eller när det har att göra med klasser som man inte vill ändra på. Det som kan vara lösningen för att få fakeObjekt på plats då är att skapa ett fakeObjekt som sedan ärver av den klass man vill göra ett fakeobjekt av. Det blir då möjligt att använda sig av objektet då det kan castas till att vara av typen för sin superklass. Under experimentet har flera sådana objekt kommit till användning.

Dessa har skapats enligt följande mönster

1. Skapa ny klass som ärver av den klass som önskas ersättas med en fakevariant vid test.
2. Försöker att skapa en instans av denna klass.
3. Om problem vid steg 2 gå igenom koden för superklassen eller superklasserna, särskilt vad som sker i konstruktorn för dessa klasser.
4. Kontrollera vad det är i klassen som kommer anropas av under körning av testet och se till att fakeobjektet kan fånga dessa anrop.
5. Köra testet med fakeobjektet castat som sin superklass och sedan verifiera att de anrop som skulle ske också har skett.

## 6.2 Beskrivning av vald modul

Då vi inte var insatta i de olika delarna av systemet eller hur dessa fungerade, gjordes valet av vilken del av Palasso som bör vara lämplig för enhetstester i samråd med några utvecklare i Palassogruppen.

Den modul som har valts är en liten MVC-modul. Det som modulen utför utifrån ett användarperspektiv är möjligheten att ställa in tre olika saker med hjälp av ett grafiskt användargränssnitt. Dessa tre är:

- Vilket språk som ska gälla i systemet.
- Vilket skal som skall användas.
- Vilken nivå det ska vara på loggningen i systemet.

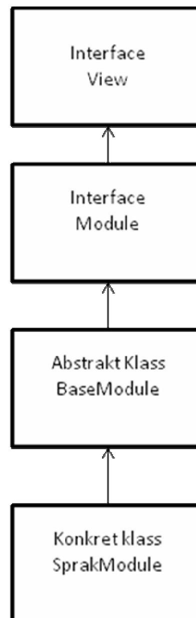
Modulen ligger i paketet Sprak och består av tre klasser:

- SprakModule
- SprakController
- SprakView

De olika klasserna har olika ansvarsområden och olika uppbyggnad. Nedan följer en beskrivning av dessa.

#### 6.2.1 SprakModule

SprakModule innehåller enbart en konstruktor. Anledningen till att klassen kan bestå av så lite beror på att den ärver alla sina ytterligare egenskaper av sin superklass. Hela hierarkin för denna klass är som bilden nedan visar:



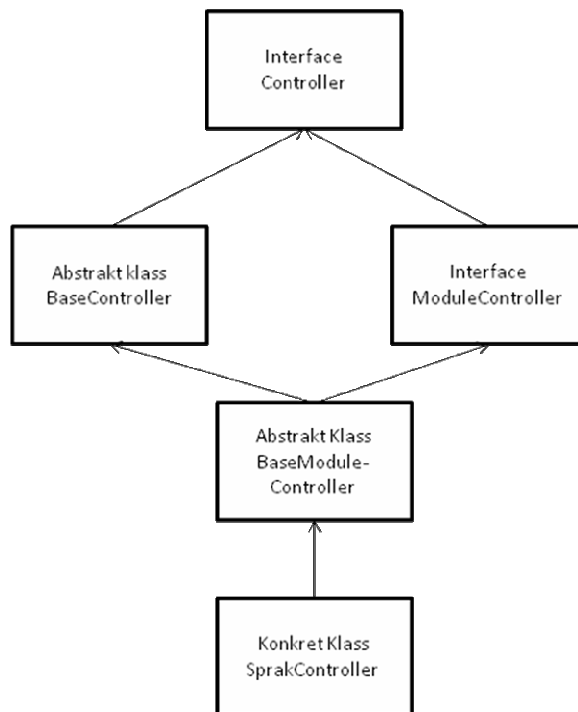

*Figur 40 SprakModules hierarki*

Det som blir tydligt när man ser denna hierarki är att alla metoder för klassen finns i BaseModule, på grund av att både View och Module är interface som inte består av några implementeringar alls utan enbart är ett gränssnitt som sedan implementeras av implementerade klasser.

SprakModule används främst av denna modul vid uppstarten av densamma. Då anropas metoden start som finns i den abstrakta klassen BaseModule. I den metoden sker kontroll av de möjliga inparametrar som skickas med vid anropet. Till exempel finns möjlighet att skicka med en container för det grafiska användargränssnittet eller att skicka med ett bestämt skin som bestämmer utseendet för det grafiska gränssnittet.  Om inga värden skickas med, kommer metoden att hämta standardvärden för dessa inställningar.

### 6.2.2 SprakController

I SprakController ligger logiken för denna modul. Controller ärver en del av sina egenskaper från BaseModuleController. Dock använder sig inte SprakController av så många av dessa metoder vilket ger en viss avskildhet mellan SprakController och BaseModuleController. Hela arvhierarkin för SprakController visas i bilden nedan:



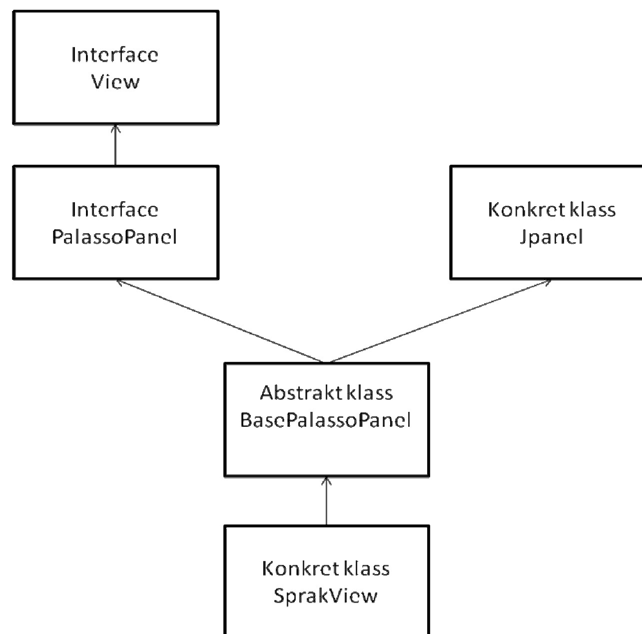

*Figur 41 SprakControllers hierarki*

SprakController har metoden StartModule som startar modulen. Detta kan delas upp i olika delar:

- Hämta data från databasen. Den inloggade användarens inställningar för språk, skin och loggnivå. Den hämtar också de olika valbara språk skin och loggnivåer som finns i systemet.
- Sätter Comboxarna i SprakView till att innehålla det data som hämtas i varje kategori.
- Hämtar och gör inställningar på den container som ska innehålla sprakView.
- Skapar en SprakView.
- Efter starten av modulen har SprakController också ansvar för sparande av data och avslutning av modulen.

### 6.2.3 SprakView

I denna klass finns den grafiska presentationen av modulen. Den tar emot data utifrån och visar på det på skärmen i form av en ComboBox för varje kategori (språk, skin och loggnivå) och två knappar: en för att spara och en för att avsluta modulen. SprakView innehåller också den privata klassen ButtonPanel som används för att skapa de två knapparna. SprakView ärver en del egenskaper av sina förfäder. Nedan följer en bild över SprakViews arv:

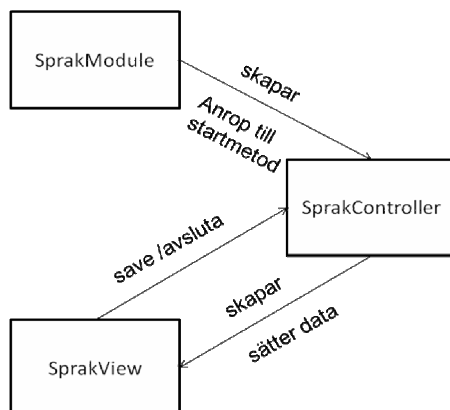



*Figur 42 SprakViews hierarki*

Det som är intressant i denna hierarki är att SprakView genom BasePalassoPanel ärver från JPanel som är en Java swing komponent som används i skapandet av grafiska användargränssnitt. Detta ger SprakView vissa egenskaper vilket är bra att veta när denna klass ska testas.

### 6.2.4 Relationer

För att få en bild över de olika klassernas relation till varandra följer nedan en bild över detta:




*Figur 43 Klassernas relationer*

## 6.3 Genomförande av experiment

Genomförandet är indelat i tre olika steg, där varje steg är en del av en cykel. Dessa steg kommer att göras metodvis. När sista steget är taget för en metod börjar man om från början med nästa.

**Steg 1**

Införandet av tester på befintlig kod med så små ändringar av koden som möjligt.

**Steg 2**

Skriva om koden för att göra den enklare testbar. Hela tiden under detta steg ska man kunna köra de test som blev skrivna vid steg 1, för att vara säker på att man inte förstör något beteende i koden.

**Steg 3**

Göra omstrukturering av koden för att få den enhetlig och i linje med de ändringar som införts i steg 2. Också under detta steg ska testen från steg 1 kunna köras för att försäkra sig att kodens beteende inte ändras.

### 6.3.1 Avgränsning

I och med att logiken i modulen ligger i SprakController kommer enhetstesterna att utgå från denna klass, för att senare ingripa de delar i SprakView som kan testas. Då metoderna som används i SprakModule inte ligger i detta paket utan finns i sina superklasser kommer dessa metoder inte att bli föremål för enhetstestning då det finns en stor risk att ändringar i dessa klasser kommer att påverka andra klasser som ärver av den. Detta gör att dessa klasser inte kommer att omfattas av införandet av enhetstest som inriktar sig på det specifika beteendet av just denna modul.

### 6.3.2 Sammanfattning av införandet av enhetstester för SprakController

Denna klass är intressant ur ett testningsperspektiv då den har med många av de delar som gör kod svår att testa. Här följer dessa:

- Singleton-anrop
- Statiska anrop
- Beroenden på konkreta klasser
- Små möjligheter att skicka in objekt i klassen.

I detta fall har metoden stått i centrum för testningen. Målet har hela tiden varit att kunna köra en metod i klassen och därefter verifiera vad som har gjorts när metoden har exekverats av testet. Då det inte har funnits några specifikationer att gå på har klassen i sig själv varit specifikation över vad som ska ha gjorts, vilket ofta är det enda valet då enhetstest ska införas på redan skriven kod[17].

Den arbetsgång som använts är:

1. Försök att göra en metod som ska testas
2. Låt koden tala om vad som saknas för att koden ska kunna köras.
3. Inför den saknade omgivningen.
4. Börja om från 1 tills det att metoden går igenom
5. Verifiera att koden gör det som den ska göra.

Det som ska tilläggas till detta är att enhetstester av redan skriven kod inte blir tester i meningen av att hitta buggar i systemet, i alla fall inte primärt, utan istället är till för att dokumentera vad koden faktiskt gör. Detta ger ett skyddsnät när ändringar ska göras i klassen så att man inte ändrar något beteende som ska bevaras.

För mer detaljer över hur införandet av enhetstester se appendix A.

### 6.3.3 Ändringar av SprakController för ökad testbarhet

**Införandet av subklass**

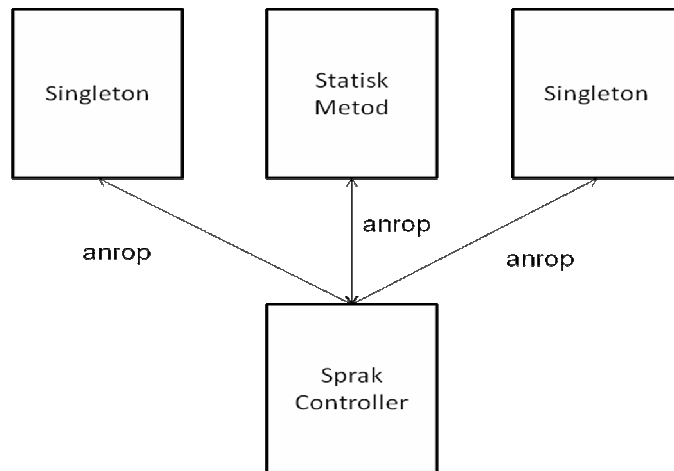
För att kunna köra vissa metoder i SprakController infördes en subklass av denna som namngavs till SprakControllerForTest. Genom denna klass infördes möjligheten att skriva över metoder och på ett enkelt sätt kunna få in testdata in i metoderna utan att det påverkade koden i klassen.

**Extrahera interface för SprakView**

När testningen började var det ett stort problem att få till ett fakeobjekt istället för SprakView, på grund av att typen som användes var just en SprakView och det gick inte att använda någon annan typ. Därför valdes att extrahera ett interface till SprakView som kallades SprakViewInterface som innehöll alla metoder som finns i SprakView. Därefter var det inga problem att få in ett fakeObjekt istället för SprakView. Valet att inte döpa interfacet till SprakView var på grund av att slippa ändra skapandet av SprakView. Denna ändring innebar bara ändring på en returvärdestyp och ändringen av fältet SprakView i SprakControllern.

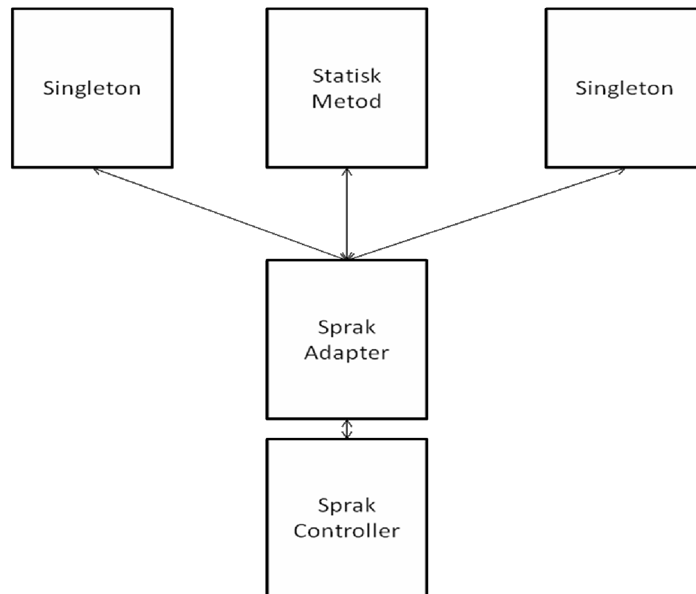
**Adapter**

När alla tester hade gått igenom gjordes en större ändring i och med införandet av en adapter. SprakController hade många direkta anrop till olika Singletons och statiska anrop. Detta innebar problem. Därför infördes en adapter som SprakController sedan använder. Adaptern implementerar ett interface så att det ska vara lätt skapa ett fakeobjekt som kan användas vid test. Adaptern i sig hämtade dock bara den post som skulle hämtas från sin Singleton. Nedan beskrivs ändringen i bilder.



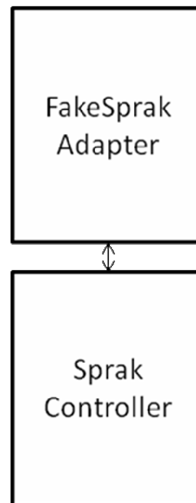

*Figur 44 SprakController innan ändring*

Ändrades till:




*Figur 45 SprakController efter införandet av SprakAdapter*

Vid test kan sedan SprakAdapter bytas ut till ett fakeobjekt enligt nedan




*Figur 46 SprakController med FakeSprakAdapter*

Denna ändring minskar antalet beroenden mellan SprakController till resten av systemet, vilket gör att ändringar som sker på utomstående objekt inte kommer att påverka SprakController. Ändringen minskar också jobbet med att underhålla koden då det är lättare att se vilka delar av resten av systemet som SprakController använder sig av.

**Möjlighet till dependency injection**

I och med införandet av SprakAdaptern (se ovan) så infördes också möjligheten att kunna skicka in en Adapter som SprakController kunde använda sig av. Genom att parameterisera konstruktorn och skapa en andra konstruktor som tog en Adapter som en parameter blev det lättare att förbereda klassen då den skulle testas.

**Omstrukturering**

En del omstruktureringar genomfördes för att minska ner dupliceringen av kod i klassen där det fanns möjlighet för det. Sedan gjorde också vissa ändringar med flera mindre metoder istället för några få stora, för att koden skulle bli lättare att förstå för nästa person som ska läsa den.

# 7 Diskussion

## 7.1 Det valda ramverket

Avhandlingen var uppdelad i två moment. Det första bestod av att ta fram det enhetstestramverk som skulle passa Palasso bäst efter de principer som vi tog upp i kapitel 3. Denna del bestod av att läsa på om de ramverk som vi hittade och som rekommenderades av andra personer genom forum för utvecklare som håller på med Scrum och TDD samt genom att studera dokumentationen över dessa ramverk.

Enhetstestramverk överlag är väldigt lika varandra när det kommer till funktionalitet. Funktionen ligger i att testa om värdet som en funktion returnerar motsvarar ett förväntat värde eller att en viss funktion går att exekvera utan några fel, till exempel kastar undantag. Det vi kom fram till och som vi tog upp i de slutsatser som vi drog efter förstudien (se kapitel 3) var att oavsett vilket ramverk som används finns dessa grundfunktioner.

Det som har varit intressant var att studera de extra funktioner som följer med olika ramverk. En intressant upptäckt är att det, JUnit, ramverk som används mest och som förespråkas i utbildningar och av experter inom området, är ett ramverk som kommer med ett mer begränsat utbud av funktioner. Det intressanta är att det finns mycket mer sofistikerade ramverk vars funktionalitet i grundutförande är betydligt bättre än det som JUnit erbjuder. Dessa ramverk är även betydligt lättare att anpassa efter behov och användningsområde. Man kan fråga sig varför JUnit förespråkas? Vi hävdar att anledningen är att det är det äldsta ramverket och därför finns det mer dokumentation över det, vilket gör att nya användare är mer villiga att använda det framför de andra ramverken.

Ramverket som valdes, TestNG, detta på grund av att det var det mest kompetenta. Det var överlägset JUnit i funktionalitet, se kapitel 3, och lika enkelt att installera och komma igång med som JUnit. Dessutom är det mycket enkelt att konfigurera, genom att använda en konfigurerings XML-fil. Det går även, som vi har nämnt och visat, gruppera tester efter funktionalitet, användningsområde eller på något annat sätt som utvecklaren tycker passar. Genom att gruppera testfallen går det att optimera testexekvering och vinna prestanda.

Det vinnande skälet till att TestNG valdes var för att det går att använda sig av databeroende testning på flera enkla sätt. Detta passar Palasso bra då det är ett system som kommunicerar med en databas, se kapitel 2. Genom att använda dataproviders eller XML-filer

går det att köra databeroende tester mycket enkelt och stabilt med säkra svar. Det som gör testen säkra är just att data blir statisk och lätthanterlig.

Som vi har nämnt tidigare går det att använda plug-ins, vilket gör att det till exempel går att använda JMock för att mocka ut exempelvis databaser.

Sammanfattningsvis efter att ha använt verktyget under experimentdelen, se kapitel 4-6, har det levt upp till förväntningarna. Den funktion som har varit mest användbar har varit grupperingsfunktionen. Den har verkligen underlättat arbetet, delvis på grund av att grupperingar av testfall snabbade upp exekveringen men framför allt att det underlättar testskrivandet och gör testklasserna mycket friare.

Det ger en frihet att döpa testfunktioner helt fritt, funktionerna behöver inte som i JUnit ha ett namn som innefattar ordet test. Den största vinsten är dock att det ger homogena testklasser. Genom att gruppera testfunktioner kommer alla egenskaper i den gruppen att delas mellan funktionerna. Vid databeroende tester är detta väldigt användbart genom att det går att koppla data till en viss grupp, vilket gör att testerna får samma data.

Grupperingar gör det även möjlighet att initialisera testfunktioner specifikt, vilket gör att databeroende testers data inte påverkar varandra genom att de sätts upp på olika sätt.

## 7.2 Metod 1

Denna del tog upp hur Palasso skulle kunna skriva och använda sig av enhetstester och TDD när de utvecklar moduler eller annan ny funktionalitet.

Metoden skulle vara allt för dyr och tidskrävande för Logica att införa på befintlig kod och vid vidareutveckling av redan existerande moduler eller klasser. Detta beror på att hela systemet måste anpassas för enhetstestning och testklasser för alla klasser i systemet måste implementeras.

Anledningen till att hela system måste vara enhetstestbart är att vid utveckling av ny eller befintlig kod blir testerna för det väldigt opålitligt om den har beroenden till kod som ej är testad. Vid fel till exempel blir det svårt att säga om felet beror på den nya eller den gamla koden.

Dock finns det ett sätt som de skulle kunna använda metoden med. Den är vid vidareutveckling av systemet genom att införa enhetstester stegvis. I samband med omarbetning av kod införs det enhetstestning på den kod som den har beroende med.

Det som skulle vinnas med att införa enhetstester vid detta moment är att utvecklaren får en klar bild över tillståndet som koden har och kommer att ha efter omarbetningen. Detta

genom att innan ändringar sätta upp relevanta testfall för koden och genom den kontrollera att den fungerar. Genom att sedan exekvera testfallen under omarbetningens gång får utvecklaren hela tiden information om något har blivit brutet i systemet. Tack vare att utvecklaren får information direkt kommer det att minska felsökningstiden och korta ner tiden som omarbetningen kräver.

TDD som användes i denna metod vid utvecklingen av modulen skulle vara svår att införa för Palasso.

Det som TDD skulle kunna användas till dock är utveckling av nya moduler eller ny logik som inte har allt för många beroenden till andra klasser. Ett exempel på det är en klass som översätter ett svenskt ord till ett engelskt genom att ta in en sträng och returnera en annan. Eftersom en sådan klass inte har beroenden till andra delar som till exempel Butler, skulle det gå att införa. Alltså helt fristående funktionalitet skulle mycket väl gå att utveckla genom TDD. Även delar som inte har allt för hårt bundet beroende med andra delar skulle kunna gå att utveckla på detta sätt. Dock blir testerna inte alltför pålitliga då de är knutna till ej testad kod.

Att införa enhetstester på redan befintlig kod anser vi vara svårt. Det är för svårt att skriva relevanta testfall. Svårigheten är att se vilka delar i en funktion som är relevant för testningen och vad som förväntas få ut av den.

### 7.2.1 Att skriva relevanta testfall

Det svåra, som vi har skrivit i stycket ovan, med att använda enhetstestning ligger i att skriva testfall som är relevanta och testar på rätt sätt.

Är det relevant att skriva ett testfall för en subrutin som skriver data till en databas? Vad finns det för relevanta enhetstestfall för det?

Svaret på det är att det är relevant. Dock är kommunikationen med databasen inte det, hur databasen skriver data och om den skriver rätt hör inte till subrutinens ansvar. Det som är relevant att testa är i dessa fall SQL-strängar, kontroll att syntaxen är rätt samt att rätt tabell eller tabeller anropas. Det är det ansvar en sådan subrutin har och inget mer, all annan funktionalitet ligger i andra klasser och bibliotek. Även om det är ett litet ansvar är det nog för att testa.

Detsamma gäller för funktioner som läser från databas, alltså hämtar tabeller, rader eller kolumner. Allt som den sköter är att se till att rätt tabell anropas. Resten är, som för add-rutiner, upp till andra delar av systemet och andra bibliotek.

Det vanligaste att testa för är extremvärden, tex. min och max värden för arrayer. Det går inte att säga allmänt går testning till på ett visst sätt och testfallen går inte heller att säga hur de ska se ut. Allt beror på vad en viss klass har för uppgift och vad som är viktigt i den klassen. Det allmänna är dock att testa extremfall och fall som inte ska inträffa. Om en funktion ska kasta ett undantag, se då till att det finns test som visar att undantaget kastas.

## 7.3 Metod 2

För att få en överblick på de aspekter som finns av enhetstestning och kodning kommer slutsatsen och diskussionen av denna metod att vara uppdelad i olika delar:

- Testning
- Dokumentation
- Tid
- Ändringar

Först följer diskussion om varje aspekt sedan följer slutsats av båda metoderna.

### 7.3.1 Testning

Det är svårt att veta vad det är man testar när det kommer till att få in test på redan skriven kod. De testfall som skrivs testar de olika vägar som koden kan exekveras och verifierar att detta görs på ett korrekt sätt.

I controller-klassen som testades fanns stora beroenden i början av experimentet. På grund av detta var det svårt att exekvera koden. Detta blev då det största problemet av alla.

Vad behöver klassen för att den ska kunna köras?

Innan det är möjligt att testa klassen får testen fungera som indikator på vad det är som klassen behöver för att fungera. Detta kommer oftast att ge null pointer exceptions i Javakod. Genom detta tillvägagångssätt får varje rad kod sitt tillfälle att säga till om det är något den saknar för att fungera. Därefter får testen se till att tillfredställa de krav som koden har för att köras. Det är först efter detta är gjort som det är möjligt att testa koden i riktig mening. Då kan testdata skickas in eller sättas för att sedan hämtas av koden som ska testas och testet kan verifiera att testdata kommer till rätt plats eller att koden reagerar på förväntat sätt.

Behovet av att möta kraven från koden, med beroenden till andra objekt, gjorde att det blev ett måste att skapa fake-objekt. Dessa skulle sedan kunna användas för att ersätta de riktiga objekt som klassen skulle använda sig av eller samarbeta med. Ett problem med detta är att om det skulle ske en ändring i dessa fake-objekt som används vid testningen kan detta leda till

fel i testningen och då det inte är klassen som testas som innehåller fel. Det naturliga att göra vid dessa fall blir att kontrollera koden som testas. Det är inte förrän detta kan uteslutas som felsökningen fokuserar på att kontrollera fakeobjekten.

Dessa fakeobjekt blev till slut en stor samling objekt som skulle kombineras på olika sätt. Detta gjorde att det ibland var svårt att hålla ordning på de olika objekt som skulle användas på olika ställen.

Ett annat problem, som gör felsökning av testfall svåra, är då testfall ger fel indikationer. Ett testfall som borde ge fel ger istället godkänt av testet. Detta kan vara svårt att förstå. Vad det är som gjorde att det gick igenom? Detta kan precis som ovan nämnda problem med fakeobjekten ligga på flera olika platser, antingen i testet eller i koden. Problemet från början ligger i att testen i mycket speglar förståelsen av koden som ska testas hos den som skriver testet. En mindre förståelse och testen kan vara helt värdelösa. Ta till exempel en metod som har en if-sats och som returnerar en integer. Metoden kommer alltid att returnera en integer. Om ett test då fokuserar på att bara få ut ett värde ur metoden är det kanske inte ett heltäckande test.

```
public int getUserNumber(){
        Int number;
if(something){
            DoStuff
        }
        else{
            doOtherStuff
        }
        return number;
}
```

*Figur 47: Exempel get funktion*

Koden inuti if-satsen körs aldrig av testet och testas därför inte. För att testen ska vara bra kräver det stor förståelse av koden, annars finns det en risk att fel missas som skulle ha varit lätta att upptäcka om testen varit mer täckande.

### 7.3.2 Dokumentation

På grund av att det var svårt att få testerna på plats som var målet med denna metod har dokumentationsaspekt varit av sekundär karaktär. Dock ska det sägas att testen som kom på plats ger en bra bild över vilka delar som behövs för att koden i klassen ska kunna köras och vilket beteende som är det förväntade. När då nästa person läser koden och har testen som stöd blir testen istället en dokumentation som ger direkt information om hur koden fungerar. Denna dokumentation är, rätt använd, en riktigt bra dokumentation, till skillnad från de dokument som sammanställs för att ge en bild över hur systemet ska se ut och hur det ska bete sig. Där koden ska omvandlas genom en person som kommer att färga dokumentationen med

sin syn på hur systemet fungerar. Risken finns också att dokumentationen inte alls kommer att stämma överens med bilden som en annan person behöver för att få samma förståelse. Dokumentationen som fås av enhetstest är av en annan karaktär. Där speglar koden som ska testas genom de enhetstest som exekverar denna kod. Denna dokumentation är levande på ett helt annat sett. Körs testerna regelbundet kommer de att säga till när en ändring har gjort att dokumentationsaspekten av testen har blivit gammal. Vilken annan dokumentation gör det? Ingen, i alla fall inte på detta konkreta sätt och lättförståeliga sätt som enhetstester gör.

#### 7.3.3 Tid

Det tar tid att införa enhetstest på det sätt som genomförts i denna metod, då testning av kod kräver förståelse av den kod som ska testas. Den tid det tar från det att det bestäms vilken del som ska enhetstestas tills dess att testfallen börjar komma på plats kan vara mycket lång. Detta beror självklart på flera olika variabler, hur mycket av systemet som är känt innan, hur dokumentationen ser ut, hur duktig personen är på att förstå kod, hur koden är skriven.

Dock ska det tilläggas att det inte tar tid att köra dessa test. Ett enhetstest ska gå fort att köra annars är det inte ett enhetstest(se kapitel 3).

Frågan är då om det är lönsamt rent tidsmässigt att införa enhetstest. Där kan det vara svårt att få ett entydigt svar. Om den tiden som kommer att läggas på enhetstest istället läggs på att utveckla andra testningsmetoder skulle det ge samma eller bättre resultat. Eller om tiden att skriva enhetstest minskar ner tiden det kommer ta att rätta till de fel som uppkommer. Denna fråga är flitigt diskuterad inom dataindustrin. Det svåra ligger i att se vad som ger olika fördelar. Är det enhetstesterna som har gjort att det tar mindre tid att rätta till fel eller är det att koden är lätt att skriva? Har programmerarna blivit bättre på att sätta sig in i redan skriven kod eller är det enhetstesterna som dokumentation som gjort att ändringar och ny funktionalitet går fortare att genomföra?

Antalet variabler som påverkar hur lönsamt det är att införa enhetstest rent tidsmässigt är många. Det är svårt att se riktigt vad det är som ger vad. Detta ger den enskilde betraktaren förmånen att välja vad den tror är anledningen till de tidsförbättringar som har gjorts i samband med införandet av enhetstester.

#### 7.3.4 Ändringar

I samband med att enhetstest ska komma på plats på redan skriven kod sker det ofta ändringar av koden för att det ska vara möjligt. Är dessa ändringar av positiv eller negativ karaktär?

Ändringar som görs för att införa test sker oftast för att bli av med beroenden till andra delar av systemet. En klass som inte har många beroenden behöver inte lika ofta ändras då andra delar av systemet ändras. Detta är en klar fördel. Om det ska till en ändring finns det ingen lust till att ändra mer än just den platsen som ska ändras. Om det då finns beroenden av det som ändrades kan ändringen fortplanta sig genom ett helt system genom att nästan alla andra klasser i systemet var beroende av just den ändringen som gjordes.

När beroenden sedan är brutna är det en stor fördel för att underlätta testningen att införa abstraktioner genom att skapa interface som klassen kan bero på. Detta skapar ett skikt mellan klassen och den implementering som den använder sig av. Denna ändring skapar dock ett ökat antal filer som programmeraren ska hålla ordning på och förstå. Men när en ändring sker i en implementeringsklass utan att gränssnittet utåt ändras så kommer detta inte att påverka den klass som använder sig av interfacet i fråga, vilket ger till exempel kortare byggtid då klassen inte behöver kompileras om.

Möjligheten att skicka in beroenden i en klass, så kallat dependency injection, är något som används flitigt för att få tester på plats. Detta används för att lätt få in testdata till klassen eller för att få in ett fakeobjekt. Dock kan det tvinga andra delar av systemet att vara tvungen att veta vad just denna klass behöver för att fungera. Någonstans måste systemet sättas ihop om alla klasser beror på det man skickar in i dem. Vilken klass ska då börja? Dependecy injection skapar möjligheter för att kunna bygga ihop och ersätta delar av system på ett enkelt sätt. Genom att använda sig av abstraktioner i samband med detta kan det ge fördelar av att systemet blir mer flexibelt. Detta medför att när nya ändringar skall göras så kommer dessa att vara lättare att genomföra då dessa inte har lika stor påverkan.

#### 7.3.5 Sammanfattning

Det är svårt och det tar tid att införa enhetstest på kod som inte är skriven för att den ska enhetstestas. Detta bör enbart införas då ändringar ska göras i den befintliga koden och då som stöd för att ändringar inte förstör funktionalitet som ska bevaras. Även om det är svårt så går det att införa tester på nästan all kod, vilket har visats genom experimentet.

Sedan ska det tilläggas att det kan vara lättare eller svårare att få test på plats om man utgår ifrån att koden redan är skriven.

För att det ska vara lätt att införa enhetstest ställs vissa krav på kodens struktur. Koden bör inte vara beroende av konkreta klasser utan bör vara beroende av abstraktioner, till exempel beroenden på interface inom Java. Möjligheten till dependecy injection gör en klass mycket mer testbar jämfört med om klassen har alla sina beroenden inuti sig. Det intressanta med

dessa krav är att de stämmer överrens med de krav som uppfylls av kod som är lätt att underhålla och lätt att ändra. Ändringar som införs för att få test på plats ökar alltså kvalitén på systemet. Ändringar som annars inte hade varit tydliga blir med hjälp av enhetstesten enkla att se.

Enhetstester bör kanske inte främst användas som tester utan mer som en parameter på vilken hälsa koden har. Är det svårt att införa enhetstester är det ett tecken på att koden inte mår bra. Självklart bidrar enhetstester till att de små detaljerna i koden fungerar som det är tänkt. För att testa enskilda klasser eller enbart en metod är det ett bra och effektivt verktyg. Istället för att vara tvungen att skriva en driver för metoden skrivs bara ett enkelt enhetstest, som sedan kan användas flera gånger. Men om det är svårt att skriva detta enhetstest är det ett varningstecken på att koden mår dåligt ur flexibilitetssynvinkel.

Dock blir det svårt att säga vad som egentligen testas då man inför tester på ett befintligt system. Utgår man ifrån att det inte är något fel i koden kommer testerna enbart att dokumentera att koden gör det den gör. Utifrån detta kan det sedan dras slutsatser om testerna avspeglar det önskade beteendet eller ej. Men det är en annan frågeställning. Istället kan dessa enhetstester ses som test och dokumentation i ett, som har information om hur koden fungerar och om den fungerar med fördelen att denna typ av dokumentation säger när den har blivit föråldrad.

I slutet av testningen var antalet fake-objekt alldeles för stort för att det skulle vara lätt att förstå. Detta är en sidoeffekt av klasser med stora beroenden. Detta går dock att lösa, genom att klassernas ansvarsområde minskas och delas upp i flera klasser kommer antalet fakeobjekt som skall användas av varje enskild testfil att minska. Genom detta blir det lättare att se beroenden mellan de olika klasserna. Detta kan ses som ytterligare en fördel av enhetstestning. Utan testen skulle detta inte ha framkommit.

Det effektivaste sättet att använda sig av enhetstest är att använda sig av dem i samband med utvecklingen av mjukvaran. Då är förståelsen av koden som störst hos programmeraren. Detta ger mindre tid för inlärning av de funktionerna som ska testas, vilket ger bättre test och då också bättre kod. Den största fördelen med att använda sig av enhetstest är att det ger så pass många fördelar i ett och samma verktyg. Bra dokumentation, bra insikt i hur koden fungerar, ett skyddsnät för ändringar, ett stöd i felsökningar. Om detta sedan ska jämföras med den tid det tar att skriva testen och att hålla dessa uppdaterade är det ett lätt val. Den tid som enhetstester kommer att spara genom färre fel och kortare utvecklingstid och lättare ändringar är betydligt större än den tid det tar att skriva testfallen. Då testfallen skrivs i samband med utvecklingen.

# 8 Slutsats

## 8.1 Testverktyg

Det testramverk som valdes till att vara bäst lämpad för Palasso blev TestNG. De egenskaper som gjorde att det valdes var att det var skalbart och lätt att använda. I samband med att vi använde oss av ramverket i experimentdelarna förstärktes detta.

## 8.2 Införande av enhetstest

Det går att införa enhetstest på Palasso. Denna slutsats kan vi dra av de experimentmetoderna som vi har genomfört, se kapitel 5 och 6. De fördelar som införandet av enhetstest skulle ge är:

- Dokumentation
- Skyddsnät
- Debuggning
- Optimera design
- Bevisad funktionalitet

Genom att införa enhetstest skapas också en dokumentation av systemet. Den består av hur och vad som fungerar i systemet. Denna dokumentation är levande då den säger till när den behöver uppdateras.

Genom att enhetstesterna dokumenterar kodens beteende kan det användas som ett skyddsnätt vid ändringar och vidareutveckling av befintlig kod, så att önskad funktionalitet inte försvinner.

För att hitta fel i kod är enhetstestning ett bra verktyg, då det ger möjlighet att testa på metodnivå i systemet.

Enhetstester hjälper till att göra designbrister i ett system tydliga. Detta genom att testerna ger en överblick om hur systemet fungerar.

Genom att ha test för att en metod finns och fungerar bevisas detta.

I Palassos fall blir enhetstestning aktuellt i samband med vidareutveckling och ändringar i befintlig kod. Även vid nyutveckling bör enhetstestning användas. Genom att använda denna strategi kommer hela systemet successivt bli enhetstestat.

Att införa enhetstester på befintlig kod skulle innebära en allt för hög arbetsbörd och kostnaderna skulle sannolikt bli allt för höga.

Slutsatserna av detta arbete är att det går att införa enhetstester på Palasso successivt. De fördelar som det för med sig väger tyngre än den tid och kostnad som krävs för att införa enhetstester.

# Referenser


[1] Ian Sommerville. 2001. Software Engineering. Upplaga 6. Pearson Education limited.

[2] Wikipedia.org. Software testing, 20 mars, 2009. http://en.wikipedia.org/wiki/Software_testing#cite_note-5, 23 mars, 2009.

[3] Dr. Winston W. Royce. 1970. Managing the development of large software systems http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf , 23 mars 2009.

[4] Michael A. Cusumano & Stanley A. Smith. 1995. Beyond the waterfall: software development at Microsoft. Sloan School of Management, Massachusetts Institute of Technology.

[5] NASA verification and validation - http://satc.gsfc.nasa.gov/assure/agbsec5.txt. 23 mars 2009.

[6] Douglas Bell. 2000. Software engineering – a programming approach. Pearson Education limited.

[7] International Software Testing Qualifications Board - Certified Tester Foundation Level Syllabus 1 Juli 2005.

[8] Sigrid Eldh m fl .SSTB ordlista Engelsk/Svensk Version 2.0, 13 december 2008, http://sstb.se.loopiadns.com/pub_documents/SSTB_ordlista_E_S_2_0.pdf 23 mars 2009

[9] Matin Fowler, Mock Aren't Stubs, 2 januari 2007, http://martinfowler.com/articles/mocksArentStubs.html, 23 mars 2009.

[10] Unit tests, Extreme Programming: A gentle introduction, 17 februari 2006, http://www.extremeprogramming.org/rules/unittests.html, 23 mars 2009

[11] Randy A. Ynchausti. .Integrating Unit Testing Into A Software Development Team's Process,1 maj 2001, http://www.agilealliance.org/system/article/file/1072/file.pdf, 23 mars 2009

[12] Robert C Martin. 2002. Agile software development. Prentice Hall.

[13] Henrik Kniberg. 2007. Scrum and XP from the Trenches. InfoQ.

[14] Wikipedia.org Subrutin, 12 mars 2009. http://sv.wikipedia.org/wiki/Funktion_(programmering), 23 mars 2009

[15] Wikipedia.org. Test-driven development. 17 mars 2009, http://en.wikipedia.org/wiki/Test-driven_development, 23 mars 2009

[16] TestGrip. Gaining control on IT quality and processes through test policy and test organisation. Rik Marselis, Jos van Rooyen, Chris Schotanus, Iris Pinkster. 2007, Logica cmg.

[17] Michael C. Feathers 2005 Working effectively with legacy code, Prentice Hall PTR

[18] Officiella TestNG webbplatsen, 31 mars 2009, http://testng.org/doc/, 26 Maj 2009.

[19] Wikipedia.org Code refactoring. 5 juni 2009 , http://en.wikipedia.org/wiki/Refactoring, 9 juni 2009.

[20] Officiella JTiger webbplatsen. http://jtiger.org/. Senast besökt 9 juni 2009.

[21] Officiella EasyMock webbplatsen. 24 maj 2009, http://easymock.org/, 9 juni 2009.

[22] Wikipedia.org Spring Framework. 22 maj 2009, http://en.wikipedia.org/wiki/Spring_Framework, 9 juni 2009.

[23] Officiella Hibernate webbplatsen. 3 juni 2009, https://www.hibernate.org/, 9 juni 2009.

[24] Wikipedia.org Java Persostence API. 3 juni 2009, http://en.wikipedia.org/wiki/Java_Persistence_API, 9 juni 2009.

[25] Säljsida för Palasso. http://www.logica.se/palasso/400007833. Senast besökt 9 juni 2009.

[26] Wikipedia.org Language_Integrated_Query . 19 maj 2009, http://en.wikipedia.org/wiki/Language_Integrated_Query, 9 juni 2009.

[27] Wikipedia.org Moch object. 3 juni 2009 http://en.wikipedia.org/wiki/Mock_object, 9 juni 2009.

[28] Wikipedia.org Dependency injection. 22 maj 2009, http://en.wikipedia.org/wiki/Dependency_injection, 9 juni 2009.

[29] Wikipedia.org Singelton pattern. 3 juni 2009, http://en.wikipedia.org/wiki/Singleton_pattern, 9 juni 2009.

[30] Officiella MYSQL webbplatsen. 2009 http://www.mysql.com/, 9 juni 200.

# A Appendix

**Målet att kunna införa enhetstest för att köra kod i Controller klassen och verifiera att det detta görs.**

Controllern är lätt att instansiera då den inte har någon inparameter till konstruktorn eller anropar någon metod i denna.

Där emot blir det lite mer kompliserat då man börjar kolla på olika metoder. Controllern består av dessa metoder:

Public void Save()

Public void startModule(Map params)

Private UserprefsRecord getUserprefsRecord()

Private void fillLocales()

Private void fillSkins()

Private void fillLogLevel()

Private void fillComboboxes()

Public void exitApplication()

Private void clearCashBeforeTerminate()

De första sakerna som man kan observera är de publika metoderna som finns I klassen. Detta pga att när man enhetstestar kollar utifrån klassen vad den gör.

Det första blir att försöka att köra startModule(Map params). Detta görs genom att anropa med nullvärde för att kunna köra. **Testkör**

NullpointerException: inget object från getView i BaseController.

Vid raden:

sprakFrame = ((JFrame)((BaseModule)getView()).getPalassoContainer());

Sätt att lösa detta: det finns möjlighet att sätta in ett objekt I controllern genom setView i BaseController.

View är ett tomt interface, så det är inte svårt att skapa ett objekt som implementera denna. Däremot ska det vara möjligt att ge tillbaka ett objekt som kan castas som en Jframe för att detta ska funka.

Genom att göra en fakeView och sätta den i controllern blir vi av med det nullproblemmet. **Testkör**

ClassCastException: view funkar inte som BaseModule, byter till en Fake Module istället som ärver av BaseModule **Testkör**

Den första raden är i denna metod nu avklarad. Därefter kommer direkt en metod med ett anrop till en singleton objekt som hanterar databas anrop. För att kunna fortsätta blir det nödvändigt att göra en subklass av detta objekt för att kunna skriva över en del av de metoder vi vill undvika att göra anrop till: Skapar därför ControllerForTest som äver alla engenskaper från SprakController

Problem med detta är att metoderna som är i SprakController är privata för att undvika detta ändrar vi dessa till protected så att man kan skriva över dessa i subklassen. Detta blir den första ändringen i klassen för att göra den testbar. **Ändring**

**TestKör**

Nu änländer testet till nästa rad som skapar en SprakView som ser ut som nedan

itsSprakView = new SprakView(this, userprefsRec );

Det som försöker testas är anropen i metoden inte att kontrollera att SprakView funkar. För att undvika att få problem vid detta steg finns det två ganska enkla saker att göra. Antigen ska man försöka att få konstruktorn att ta fakeargumenten och vara nöjd med det, dock har SprakView anrop till metoder inne i sin konstruktor. Vilket gör det till en otrevlig sidoeffekt eftersom det inte är det som ska testas. Det andra möjligheten är att extrahera anropet till en protected method och skriva över denna i subklassen och returnera **null** vilket verkar vara enklare. Det andra alternativet väljs och skapar metoden makeSprakView() som anropas det ovan nämnda raden blir till

itsSprakView = makeSprakView();

**Ändring**

Än så länge inte en genomkörd metod.

Nästa problem ligget i en try catch där ett anrop till fillComboboxes. Skriver över detta anrop i subklassen. Den metoden som nu anropas kontrollerar bara att anropet görs.

Efter detta kommer det ett anrop till itsSprakView som tidigare sattes till null med hjälp av överskrivning.

```
itsSprakView.updateView();
```

Detta ger nullpointerException då null inte har nån metod updateView att anropa.

Nu ska man försöka att skapa ett objekt för att simulera en SprakView. Detta göras genom att skapa en subklass av SprakView och låter MakeSprakView sända en sådan till metoden som testas.

Nu har det blivit ett större problem då skapandet av FakeView ger fel i sin superklass konstruktor där det sker ett anrop till en singleton som inte är instansierad. Sätt att lösa detta skulle kunna vara att skapa ett interface för SprakView som man kan göra en fakeView av detta för att slippa beroendet av den konkreta klassen view. Då finns det två sätt att göra detta:

Antingen skapar man ett interface som heter SprakView och döper om SprakView till t ex SprakViewImpl, detta kommer dock att påverka alla ställen i koden som skapar en SprakView. Och sedan låta SprakViewImpl implementera det interfacet. Fördel med detta är att det inte behövs ändra något på fältet i SprakController då den är till typen SprakView.

Det andra sättet skulle vara att skapa ett interface som heter SprakViewInterface som sedan både SprakView och FakeView skulle kunna implementera. Det ger fördel att inte behöva vara orolig för de ställen som skapar en SprakView men måste dock byta typen på Viewfältet i SprakController så att den tar en SprakViewInterface istället.

**Ändring**

Då målet är att störa koden så lite som möjligt så verkar det bättre att göra ett interface som innehåller alla metoder som finns i SprakView och byta typen på handtaget i SprakController. Då sprakview är en privatmedlem av SprakController behöver man inte vara orolig för att ändringen komme att påverka något utanför SprakController.

Det som behövdes ändras var deklartionen i SprakController och en castning där detta handtag inte kunde skickas utan att castas till en SprakView. Nu är det möjligt att skapa en FakeView som kan användas. En ändring till blir i metoden som skapar SprakView där return fältet byter från SprakView till SprakViewInterface.

**Testkör**

NullpointerException: Anrop till den den Jframe som hämtades från fakemodule där null returneras.

Väljer att skapa en fakePalassocontainer som fakemodulen skulle kunna sända tillbaka.

**Testkör**

nullpointerException: anrop till userPrefsRec som sattes till null vid anrop på getUserprefsRec. Då UserprefsRecord är en databasklass är det inte lätt att instansiera. Valet blir att göra är att extrahera en metod av det anropet och skriver över detta i testklassen under.

**Ändring**

If((NullSafe.toString(userPrefsRec.get(”skin”), ””).trim()).equals(“aquathemepack”))

Till

If(skinIsAquathempack())

**Metoden går att köra igenom nästa steg är att kontrollera att den anropar det den ska.**

Skriver ett nytt test som heter testVerifyStartModule

För att kunna kontrollera att anropen görs behövs hjälpmetoder i de fakeobjekt som används.

Detta görs genom att registrera alla anrop till klasserna och kontrollera att antalet anrop stämmer med de antal som står i koden. Vilket görs med implementering av en lista i varje fakeObjekt.

Den första idén var att klasserna själva skulle kolla om antalet anrop stämde men då det finns förhoppning om att det ska gå att använda sig av objekten för vidare testning. Så sker kontrollen av antalet anrop vid testet.

Nästa steg blir att kontrollera om det är möjligt att köra nån mer metod direkt genom uppbygganden av FakeObjekten. Det finns en som heter fillComboBoxes som som bara anropar metoder inom den egna klassen som var lätta att skriva över och testa. Nu finns tre test, 2 för StartModule och en för fillComboBoxes.

Den enda logiken som finns kvar i denna motod är en if satts som bestämmer dimension som ska vara på jFramen beroende på vilken skin som gäller. Där ska anrop ske och kontroller att den reagerar på rätt på olika värden.

**Testkörning**

Skriver först testen som inte går igenom då det inte finns några metoder i subklassen därefter implementerars metoderna för att sätta om det ska vara denna aquaskin eller inte.

**Testkörning**

Det går igenom, skriver nu test för varje rad av de anrop i koden som görs. Väljer att skriva ett test för varje rad för att minska risken för fel.

**Testkörning**

Allt går igenom. =)

I metoden startModule finns det en rad som blir svår att testa då det är ett anrop till en metod som är deklarerad till att vara static. Vilket gör den lite svår att skriva över, det man skulle kunna göra att att byta ut den till ett metodsanrop innom klassen och skriva över det under. Men då vi inte har haft några problem men detta anrop vet vi inte hur vi ska lösa detta än. Det som görs på objektet i detta anrop är ansvar som ligger på en annan klass därför påverkar det inte. Dock vill vi vara konsekventa så vi väljer att extrahera en metod och skriva över anropet i testklassen. Kollar att testerna går igenom. Ett test failar pga ändring i ett test, fixar det och alla test går igenom.

Testar att kommentera bort en av raderna i metoden och se om testerna uppfattar detta och två tester indikerar på att något skett.

Nästa steg är att testa att metoden hantera exceptions som den ska.

Väljer att kasta exception vid två olika ställen i koden.

Det första är där Exception fångas i metoden och där ett felmeddelande skrivs ut.

Det andra stället blir ett exception som kastas vidare till anroparen av metoden alltså testet. Där det kotrolleras att det är rätt sorts Exception som kastats.

Nu finns test för alla rader kod i StartModule Metoden. Dock finns det inte test för de metoder som skapats i försöket att göra StartModule testbar. Därför blir nästa steg att testa dessa metoder. De som skapades och skrevs över var:

Protected SprakViewInterface makeSprakView()

Protected skinIsAquathemepack(()

Protected void setCenterFrame()

Dessa tre innehåller delar som är svåra att testa, en är ett statisk anrop, det andra är en skapning av ett objekt och det tredje är ett anrop på ett objekt som är svår att ersätta med en fake.

För att testa det statiska anropet ändras metoden till att ta in framen som en parameter som senare vidarebefodras till den statiska metoden. Testet kontrollerar om det har skett några ändringar av positionen då det är detta som den statiska metoden sätter.

Som nästa steg blir den metod som använder sig av det objektet som är svårt ersätta med en fake. Lösningen på detta blir extrakt metod på just den lilla delen och skriver över den så att logiken kan testas ovan kan testas. Genom dessa två åtgärder är det två stycken metoder som inte har blivit testade. Det som görs i dessa är ett anrop till en svårt objekt och skapandet av ett annat.

Det objektet som är svårt att ersätta med ett fakeobjekt är ett userprefsRecord som är en klass för mappning mot databasen. För att kunna testa resten av metoderna i klassen bör vi komma på ett sätt att kunna runt beroendet av denna klass pga att det finns ett flertal anrop till just denna klass.

Det som var problematiskt med denna klass är att den beror på en Tableklass för att kunna fungera. De metoder som finns i UserprefsRecord klassen är privata så det går inte att skriva över dessa.

Däremot går det att göra det i table klassen.Därför skapas en faketable klass som därefter gör det möjligt att skapa en fakeUserprefsRecord. Det man skulle kunna göra var att ändra på

userprefsRecordklassen men detta känns inte okej då det inte har nått med SprakController att göra.

Skriver ett test som kör igenom metoden som hämtar data från ett UserprefRecord. Vid testet har userprefsRecordet byts ut mot en fake. Därefter kontrollerar att det som hämtats stämmer överrens med testdatat.

**TestKörning:** Funkar fint alla test går igenom

Nu är det bara skapandet av den nya SprakView som inte har testats i StartModul vilket vi anser vara helt okej. Om det är något fel med detta finns det bara en metod som behöver kontrolleras.

Nu blir det att fokusera på att försöka testa de andra metoderna. Nästa metod är en som heter save() som gör ett anrop till SprakView och ett anrop till modellen som finns i sprakView som är av typen UserprefsRecord.

Skriver ett test som fångar anropen från koden och sedan verifierar att detta stämmer med antal anrop.

**Testkörning:**Funkar fint, men för att kunna sätta de privata fälten i SprakController så kördes startModule först. Detta gjorde att fakeSprakView och FakeUserprefRecordet måste rensas från de anrop som gjorts innan körning av metoden save(). Så att testen är på save och inte på startModule.

När nu detta är klart väljer vi att snygga till koden i StartModule. I och med att metoden har test för allt bör det vara lugnt att ändra i den. De ändringar som görs är att extrahera metoder från denna för att få en lättare överblick över vad metoden gör. Mellan varje ändring körs testerna så att ändringarna inte ändrar kodens externa beteendet på något sätt. Det är detta som kallas för refactoring.

Nästa steg är en lite mer komplicerad metod som hämtar en tabel och fyller ett UserprefsRecord. De delar som är lite komplicerade är de två första raderna som den första hämtar en table från databas genom en singleton den andra hämtar användar id genom en statisk metod. Detta går att lösa genom att man tar och extraherar dessa som metoder och skriver över dessa. Detta görs med dessa två rader. Pga att de ska returnera något så skapas en metod för varje rad.

Skriver över dessa metoder i testKlassen så att de returnerar null när det är ett objekt och returnerar 0 när det var ett nummer. För att komma vidare.

Nästa sak testen fastnade på var att det objekt som skulle hämtas från databasen var ett tabelobjekt. Nu blir det att försöka att få till en fakevariant på för att kunna fortsätta testningen. Detta visade sig vara mycket lättare sagt än gjort. Detta pga att det gjordes en del

i denna konstruktor och dessa berode på andra objekt som gjorde det hela mycket krångligare. Dock efter att ha hittat felet som var ett anrop till en metod som var möjlig att skriva över i en subklass så fungerar det.

Nu går metoden igenom och det finns test som verifierar att koden gör det den ska. När exception kastas går den rätt i logiken som är nu och rätt data kommer på rätt ställen. För att kunna testa detta har tre nya metoder extraherats med en rad i varje. Detta var för att slippa anrop till statiska metoder, singletons eller skapning av objekt som sen använder sig av singleton.

Det första som är ett statiskt anrop till en metod som använder sig av en singleton för att hitta användar id. Detta ska gå att komma runt. Det svåra är att hitta ställen där man kan stoppa in värden och se till att den hämtar rätt värde. Tyvärr sätter både det statiska anropet och singletonen stop för att hitta något innan detta. Men till slut hittas ett ställe där det är möjligt att stoppa in en session som senare raden av kod kan hämta. Sedan verifieras värdet i testet.

Nästa metod är ett skapande av ett objekt som hämtar en String som representerar en locale vilket visar sig vara svårt. Pga att anrop som snurrar i väg in i butler(databasramverk) och variabler och instanser som behöver sättas. Som det ser ut blir det svårt att skriva ett enhetstest för denna metod pga att det blir testning av en massa andra objekt på vägen.

I dagsläget finns två metoder som inte har bra test:

Protected SprakViewInterface makeSprakView()

Protected String getLocale()

Och nu blir det ännu en metod.

Protected getUserprefsTable som hämtar en tabel från databasen genom singleton.

Då storleken på SprakControllerForTest har börjat att bli lite för stor för att vara lättbegriplig har vi börjat att fundra ut en lösning på hur man ska bli av med beroendet av databasanropen i SprakController och samtidigt minskar ned storleken på SprakControllerForTest.

För att göra testningen lättare och få grep om vilka delar som består av anrop till andra delar av systemet så bestämmer vi oss för att skapa en Adapter för de singleton anrop och de statiska anrop som finns i de methoder som finns i Controllern. Börjar med att skapa ett interface och väljer sedan att skapa två olika objekt som implementerar detta interface. Varav det ena är den skarpa versionen och den andra är en fakeAdapter som vi använder oss av vid testning. Det fina i detta är att det gör möjligheten att förflytta de överskrivna metoderna i ControllerForTest till FakeAdaptern och de faktiska singletonanropen till den riktiga

adaptern. Detta ger både kontroll över singletonanropen och ökar möjligheterna för testning. Medan dessa ändringar införs körs testen mellan varje steg för att vara säker på att inte något annat går sönder.

Ser över de test som skrivits så långt. Det finns vissa test som kör samma kod bara att de kollar olika saker i slutet för att minska ner detta så väljer vi att försöka att kunna kontrollera händelserna för varje metod som har kört istället för som tidigare att kolla varje rad. Detta gör att antalet test minskar drastiskt. I och med detta städas också sprakControllerFortest upp. Där några metoder har blivit förlegade i och med utvecklingen av testen.

Metoder som är kvar att testa är metoder som hämtar data från databasramverket och fyller Comboboxar som finns i SprakView med det datat.

Dessa metoder är över skrivna i testklassen. Därför införs en metod i testklassen som anropar metoden för att kunna köra metoden. Den första metoden som ska testas är fillLocales. Stöter på ett anrop till en singleton vid första raden. Gör en extract method och kopierar denna metod till sprakAdapter. Sedan skrivsmetoden om i SprakController till att använda sig av sprakAdapter.

För att kunna köra metoden måste en fakeLocaleTable skapas. Detta objekt använder vi sedan för att kunna komma in med testdata i metoden. Får in detta objekt i metoden genom anropet till adaptern. För att underlätta testningen av denna metod görs en metod för att sätta vilken sprakview som ska anropas i SprakController. **ändring**

I metoden används en listIterator som går igenom de olika records som finns i databasen. I och med detta används interfacet listiterator. Detta är bra då det ger möjlighet att använda ett ramverk(easymock) för att lätt skapa ett mockobjekt. Mockobjekt är som ett fakeobjekt dock med skilnaden att mockobjekt har stöd för att verifiera anrop.

Metoden går igenom. Skapar ett fakeRecord. Lyckas att fylla comboboxen men ett item från det fakeRecordet och kontrollera att det har hänt.

Nästa test är ett där comboboxen fylls med två stycken items. Det funkar fint. I metoden finns det ett statiskt anrop till en NullSafe objekt. Därför skrivs ett test till att returnera null från recordet för att kolla vad som händer. Lite förvånande ger detta ger ett nullpointer exception. Vilket vi anser vara lite konstigt då Nullsafe borde ha gjort nått det. (**bugg?**)

Skriver test för fillSkin som är nästa metod samma problem som förut med nullsafe metoden men utöver detta funkar det fint och metoden gör det den ska. Det som är lite intresant med fillLocale och fillskin är att testen ser nästan identiska ut. Kan det vara så att de gör samma sak?(**fundering).** Kan det ligga en dupplisering och lura här kanske ?

Låter det vara så länge då det finns en metod till som ser nästan likadan ut. Det denna gör är att fylla i alla lognivåer som finns för användaren. Skriver test för denna metod också och ser att de funkar som de ska. Mycket att det som finns i testet ser ut som de har gjort i de två andra metoderna som fyller comboboxar.

Nu när det finns test för dessa metoder också tar vi och gör lite refactoring på testen för att de inte ska vara så svåra att förstå och tar bort så mycket av dupliseringen som möjligt.

Nästa metod är exitApplication() som anropar clearCashBeforeTerminate() och gör dispose på sprakFrame. Dessa metoder använder sig av singleton och statiska anrop för att rensa cache vid två ställen i systemet. Gör extrahera metode på dessa anrop sedan används sprakAdaptern för att få bort beroendet på Singletonen och det statiska anropet. Vi lyckas i samband med det att få in test på det som händer i metoderna.

ALLA Metoder testade

I slutet ändras sättningen av sprakadapter så att den sätts i en konstruktorn i stället för att sättas i startmodule. Detta för att det ska vara lättare att sätta den när man testar.