Examensarbete

Analysera testbarhet på Resurs Bank

LTH Ingenjörshögskolan vid Campus Helsingborg
Institutionen för datavetenskap

Examensarbete:
Jonas Schrewelius
Nicklas Nilsson

© Copyright Jonas Schrewelius, Nicklas Nilsson

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden
Tryckt i Sverige
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2012

Sammanfattning

Resurs Bank är en nischbank baserad i Helsingborg med inriktning på finansiella tjänster mot företags- samt privatmarknaden. De har en IT-avdelning som utvecklar program och webbapplikationer. I dagsläget har de inga testare . Utvecklarna testar själva mjukvaran, detta i kombination med automatiska tester. Resurs Bank vill anställa testare som kan testa mjukvaran och behöver därför ett nytt sätt att hantera testflödet.

Målet med detta arbete är att ta fram ett arbetssätt där testarna är en egen enhet, att göra det nya testflödet effektivare än det nuvarande. Att hantera och rapportera buggar på ett bättre sätt, dokumentera relevanta testmetoder och sätta sig in i Resurs Banks nuvarande arbetssätt.

Det som detta examensarbete slutligen resulterat i är ett effektivt testflöde anpassat speciellt för Resurs Bank efter de program och det arbetssätt de använder. Program som Redmine för buggrapportering och Jenkins och Sonar för automatiska tester har integrerats i testflödet. Även kravet av Resurs Bank att testarna ska vara egen enhet är uppfyllt och har integrerats.

Detta examensarbetet har sammanfattningsvis resulterat i:
• En testplan lämplig för Resurs Bank.
• Testning av Butiksservice 2.
• Dokumentation om testmetoder lämpliga för Resurs Bank.
• Dokumentation om testtekniker lämpliga för Resurs Bank.
• 1 veckas testning är nu inkluderat i utvecklingsmetodiken.
• Övergång till projekthanteringssystemet Redmine.

Nyckelord: testflöde, testplan, testning, Resurs Bank, Redmine

Abstract

Resurs Bank is a niche bank based in Helsingborg, with focus on financial services for corporate and private markets. They have a department that deals with programming and maintaining applications that their customers use. Currently they have no employed testers. The developers are testing the code themselves in combination with automated tests. Resurs Bank wants to acquire testers that can test the software and therefore need a new test flow.

The goal of this work is to develop a flow where the testers are its own entity. Making the new flow more efficient than the current. Also to manage and report bugs in a better way.

The thesis finally resulted in an efficient test flow tailored specifically for Resurs Bank by the programs and the approach they use. Programs such as Redmine for bug reporting, Sonar for monitoring and Jenkins for automated tests were integrated in the test flow. Even the demand by Resurs Bank, that the testers should be an own unit, is met and has been integrated.

This thesis summarized resulted in:

- A test plan suitable for Resurs Bank.
- Testning of Butiksservice 2.
- Documentation of the methods suitable for Resurs Bank.
- Documentation of test techniques suitable for Resurs Bank.
- 1 week of testing is now included in the development methodology.
- Transition to the Project Management system Redmine

Keywords: test flow, testplan, testning, Resurs Bank, Redmine

Förord

Först och främst vill vi tacka Resurs Bank AB som gett oss möjligheten till detta examensarbete. Ett tack till vår handledare Vlado Palczynski på Resurs Bank, som sett till att vi aldrig stått stilla i arbetet och fått oss i kontakt med rätt personer inom Resurs Bank. Ett tack till Christian Fogel och Erik Cedergren, utvecklare och testare på Resurs Bank, för all hjälp inom testningen vi fått utföra. Ett extra stort tack till Sigurdur Birgisson som lyckades trycka in en intervju med oss trots sitt hektiska och upptagna schema. En intervju som gav oss väldigt viktig information och synpunkter. Även ett tack till Christin Lindholm, vår Examinator på LTH, för den kontinuerliga hjälp och feedback vi fått från början till slut.

Innehållsförteckning

1.1 Inledning 1
1.2 Bakgrund 1
1.3 Syfte och målsättning 2
1.4 Problemformulering 2
1.5 Avgränsningar 2
1.6 Målgrupp 3
2 Metod 4
2.1 Faser i examensarbetet 4
3 Analys 6
3.1 Bearbetning av insamlat material 6
3.2 Testflödesprototyp 6
3.3 Validering 7
4 Test strategi 8
4.1 Redmine 8
4.2 Buggrapportering 8
4.3 Testplan 9
4.4 Context-Free Questions 10
5 Resultat 12
5.1 Teknisk Bakgrund 12
5.1.1 Testnivåer 12
5.1.1.1 Enhetstestning 12
5.1.1.2 Integrationstestning 12
Big Bang Teknik 13
Inkrementell Teknik 13
5.1.1.3 Systemtestning 13
Test av hela systemet 13
Typer av Systemtestning 13
5.1.1.4 Acceptanstestning 21
Typer av Acceptanstestning 21
5.1.2 Box grupperingar 21
5.1.2.1 Black-box Test Metoder 22
Fuzz testning 22
Modellbaserad testning 22
Exploratory testning 24
Equivalence Partitioning 26
Boundary-Value Analysis 27
All-Pairs testning 27
5.1.2.2 White-box testmetoder 28
API testning 28
Code Coverage 29

Fault Injection 30
Mutation testning 30
Statisk testning 32
5.1.2.3 Gray-box testning 33
Spårbarhetsmatris 33
Ortogonal vektor testning 34
5.1.3 Jämförelse mellan box grupperingarnas testmetoder 35
5.2 Resurs Banks nuvarande utvecklingsmetodik 38
5.3 Butiksservice 2 40
5.3.1 Funktionalitet för Butiksservice 2 40
5.3.2 Befintlig testning av Butiksservice 2 41
5.3.3 Butiksservice 1 vs Butiksservice 2 41
5.4 Val av testmetoder 41
Exploratory testning 41
Input testning 41
API testning 42
GUI testning 42
Kompabilitetstestning 42
Prestanda, Laddning, Stress, Skalbarhet och Volymtestning 42
Säkerhetstestning 42
Installationstestning 42
Tillgänglighetstestning 42
Regressionstestning 42
Acceptanstestning 42
5.5 Resurs Banks nya testflöde 43
5.6 Testplan 44
5.7 Funna buggar av Butiksservice 2 46
6 Slutsats 47
6.1 Frågor från problemformulering 47
6.2 Framtida utvecklingsmöjligheter 48
7 Terminologi 49
8 Referenser 51
9 Bilagor 54
9.1 Intervjuer 54
9.1.1 Frågor kring nuvarande testning med Vlado (31/1-2012) ... 54
9.1.2 Intervju med Sigurdur (2/4-2012) 56
9.2 Context-Free Questions 57
9.3 Funna buggar 59

1.1 Inledning

Detta examensarbete är resultatet av ett samarbete mellan Lund Tekniska Högskola och Resurs Banks IT-avdelning och syftar till att underlätta deras kommande omstruktureringar. Resurs Bank är i ett expansivt skede och behöver omstrukturera utvecklingen av programvara. Då denna växer så vill de även lägga mer fokus på testning, därav behövs nya rutiner och ett nytt sätt att testa programvaran.
IT-avdelningen fokuserar främst på drift av befintliga system och utveckling av ny programvara och tjänster som ska komma att användas av Resurs Banks ombud. Denna programvaran är avsedd för Resurs Banks tjänster och är ofta någon form av webbapplikation.
Denna del av rapporten avser att klargöra för varför detta examensarbete behövs och vad Resurs Bank är för någonting.

1.2 Bakgrund

Resurs Bank är en nischbank med inriktning på finansiella tjänster mot företagsmarknaden och privatmarknaden. De har 200 anställda och huvudkontor på Väla i Helsingborg, samt ett kontor i Stockholm. Totalt har de drygt 1 000 000 kontokunder.

1977 etablerades företaget som Resurs Radio & TV.
1983 bildades Resurs Finans.
1989 såldes Radio- och Tv-butikerna och Resurs lade fokus på finans.
1998 etablerade de sig i Danmark.
2001 fick Resurs Bank bankoktroj och är idag ett av 32 svenska bankaktiebolag.
2002 etablerade de sig i Norge och Finland.
2008 förvärvade Resurs Bank Kaupthing Finans.
2010 planerades etableringen av företagets första filialkontor i Norge, som öppnade i januari 2011.
[1.0]

1.3 Syfte och målsättning

Syftet med arbetet är att underlätta infasningen av separat testning, då främst integrationstestningen, samt skapa ett bättre flöde för hela testprocessen. Detta kommer att realiseras genom skriftliga rutiner och testverktyg.
Målet är att ta fram ett flöde som hanterar Resurs Banks testning av programvara, från utvecklare till kund. Resultatet kommer att levereras i form av en rapport och det framtagna testflödet skall implementeras på Resurs Bank.

1.4 Problemformulering

Resurs Bank använder idag ett testflöde utan några dedikerade testare. I dagens läge agerar programmerarna som testare. Det Resurs Bank eftersträvar är ett bättre testflöde, effektivare arbetssätt och högre kvalitet.
En stor utmaning är att hitta det flöde som passar Resurs Banks sätt att arbeta. En del av arbetet är att sätta sig in i deras nuvarande arbetssätt, för att sedan hitta det testflöde som skulle passa in bäst utifrån det. De testmetoder som finns idag skall kartläggas och för- och nackdelar för dessa skall identifieras. Resultatet kommer att kontrolleras och implementeras på Resurs Bank.

Genom ovanstående formulering har följande frågeställning identifierats:

- **Vad är ett effektivt och bra testflöde för ett medelstort tjänsteföretag?**

Andra frågor som skall försöka besvaras:

- Vad är det optimala testflödet för Resurs Bank?
- Vilka rutiner kommer behövas?
- Vilka testmetoder är aktuella idag?
- Hur är företaget strukturerat idag?

1.5 Avgränsningar

Det är ett tidskrävande arbete att testa en applikation fullständigt.
Därav kommer endast de mest kritiska bitarna av programmet att väljas ut och testas. Delar av Resurs Banks nuvarande arbetssätt är svårt eller går inte att ändra på inom tidsramen för detta arbete även om en viss omstrukturering kommer påbörjas. Detta då en del system tar tid att byta ut då mycket information måste flyttas på ett smidigt sätt.

1.6 Målgrupp

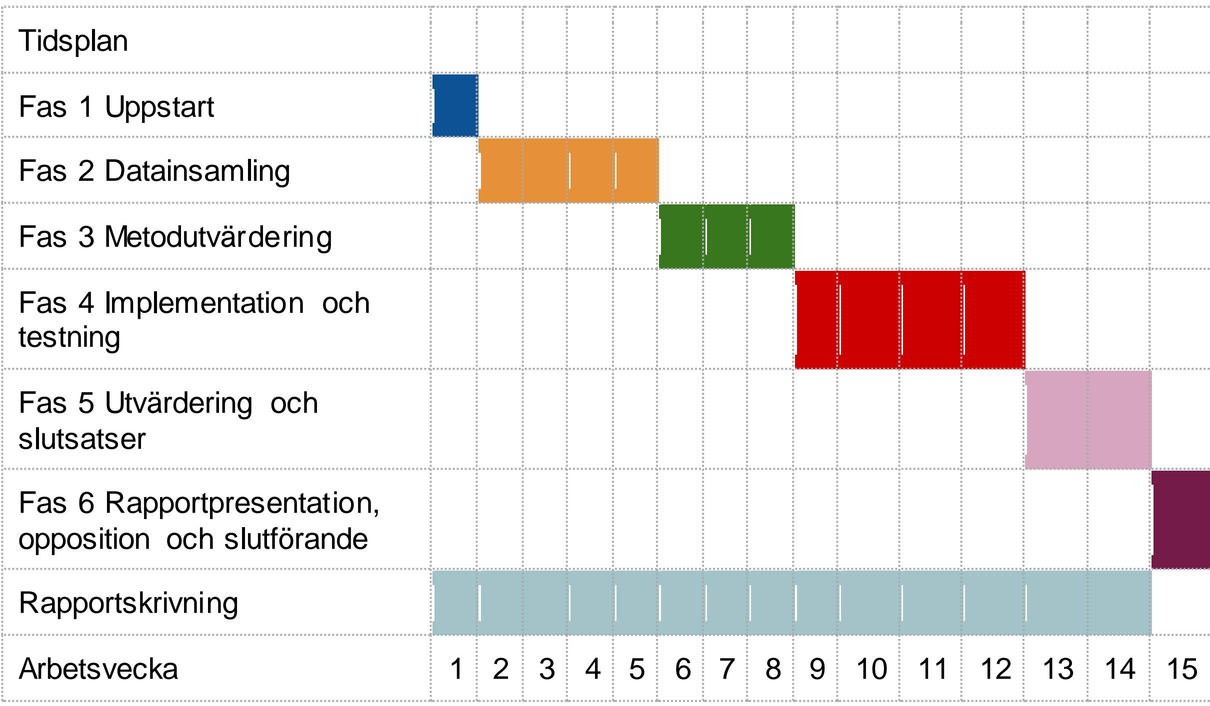
Rapporten riktar sig främst till Resurs Banks IT-avdelning men också till andra företag i liknande situation. Den riktar sig även till personer med en nyfikenhet eller intresse för programvarutestning.

2 Metod

Syftet med denna del av rapporten är att redovisa för vilka metoder som använts för att färdigställa arbetet. Information om Resurs Banks nuvarande system och testning har samlats in genom intervjuer, interna dokument och Resurs Banks intranät. Övrig information har samlats in från forskningsartiklar, arkiverade examensarbeten, bloggar och annan litteratur

2.1 Faser i examensarbetet

Tidsplanen och arbetsmodellen som valdes till arbetet är vattenfallsmodellen[25.0] då det kändes naturligt att slutföra de olika delarna i en viss följd. Tanken med vattenfallsmodellen är att varje steg ska vara helt klart och bedömas innan man går vidare till nästa steg. Med ett undantag att rapportskrivning sker parallellt under arbetets gång.



Figur 2-1. Beskriver projektets tidsplan i ett Gantt-schema med start den 8 februari.

Fas 1 Uppstart

Denna fas huvudsyfte är att sätta sig in i Resurs Banks nuvarande utvecklingsmetodik. Att undersöka vilka program som används för övervakning och testning idag samt vilket arbetsflöde en produkt har. Även att se vilken typ av företag Resurs Bank är och vilka tjänster dem erbjuder. Detta gav en bra förståelse om vad Resurs Bank är och hur de arbetar.

Fas 2 Datainsamling

I denna fas fastställs de mest relevanta testmetoder och en djupare studie kring dessa görs. Informationen samlas genom internet och diverse litteraturer. Det som hittas dokumenteras som “Teknisk bakgrund” då detta ligger till grund för vidare utvärdering och implementering. Även eventuella intervjuer tillåts.

Fas 3 Metodutvärdering

Genom resultatet från tidigare faser och i samarbete med utvecklarna väljs de lämpligaste test-metoderna och test-metodikerna ut.

Fas 4 Implementation och testning

I denna fas testas de arbetssätt som anses vara lämpliga för Resurs Banks arbetsmiljö. Författarna av detta arbete kommer även att agera testare under denna fas och på så sätt kunna lägga in egen feedback på hur arbetssättet fungerat, och modifiera det efter behov. Eventuella buggar rapporteras och lämplig mjukvara testas. Samtidigt bidrar utvecklarna och uppdragivaren med feedback om hur allt fungerar. Även eventuella intervjuer tillåts.

Fas 5 Utvärdering och slutsatser

Rapporten färdigställs och överlämnas till Resurs Bank och Lunds Universitet.

Fas 6 Rapportpresentation, opposition och slutförande

Rapporten och arbetet presenteras på Campus i Helsingborg och Resurs Banks intressenter och arbetet avslutas officiellt.

3 Analys

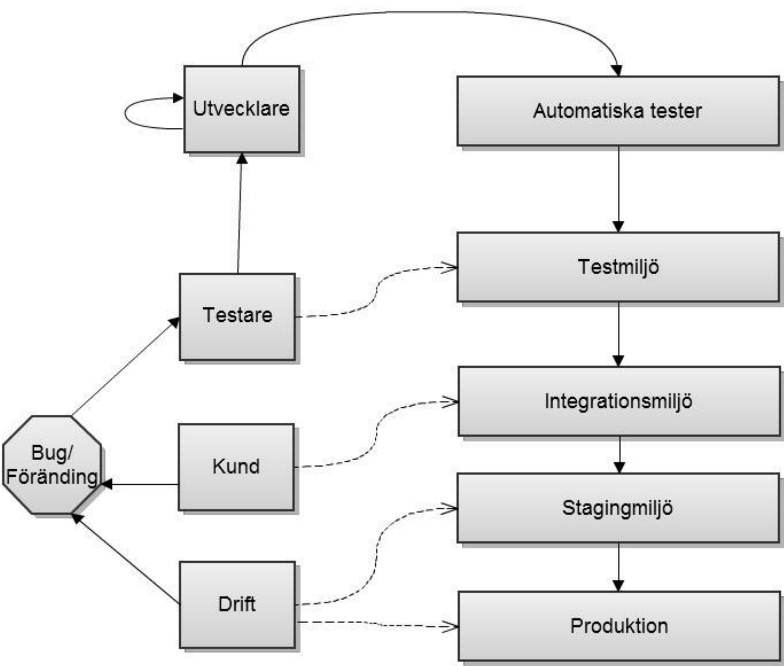
I denna del analyseras den information som samlats in under arbetets gång.

3.1 Bearbetning av insamlat material

Den insamlade information i framförallt kapitlet teknisk bakgrund har bearbetats genom sammaställningar och jämförelser. Testmetoderna och testteknikerna har testats på webbapplikationen Butiksservice 2. På detta sätt har "proof of concept" uppnåtts då det testats i skarpa miljöer.
Att författarna inte hade någon tidigare erfarenhet har gjort att detta blivit en viktig del av arbetet då det annars är en svår uppgift att avgöra om det material som samlats in är korrekt.

3.2 Testflödesprototyp

En testflödesprototyp konstruerades innan den mer detaljerade och slutgiltiga modellen blev framställd. Se Figur 3-1. Detta var bra för att säkerställa uppfattningen av Resurs Banks arbetssätt genom att visa utvecklarna en lättöverskådlig bild. För den slutgiltiga modellen se Figur 5-17 i kapitel 5.5 Resurs Banks nya testflöde.



Figur 3-1 Prototyp av testflödesmodell

3.3 Validering

Mycket av den information som inhämtats till detta examensarbete har hämtats från bloggar av kända personer inom testning. Även intervjuer och föreläsningar av testare har gett en bra bas för vad testning är och hur man gör. Eftersom det inte finns något direkt rätt och fel sätt att testa på, utan att olika metoder fungerar bra till olika situationer, så kan man aldrig veta vad som fungerar bra och vad som fungerar dåligt förrän man satt igång med testningen. Testning kommer alltid att vara något man får anpassa under arbetets gång.

Enligt Runeson och Hösts artikel "Guidelines for conducting and reporting case study research in software engineering" innebär en kombination av kvalitativa och kvantitativa forskningsmetoder att författaren får en bättre förståelse av de studerade materialet. Det faktum att arbetet användt både kvalitativa intervjuer och en kvantitativ undersökning med flera separta intervjuer stärker därför arbetetes giltighet. [26.0]

4 Test strategi

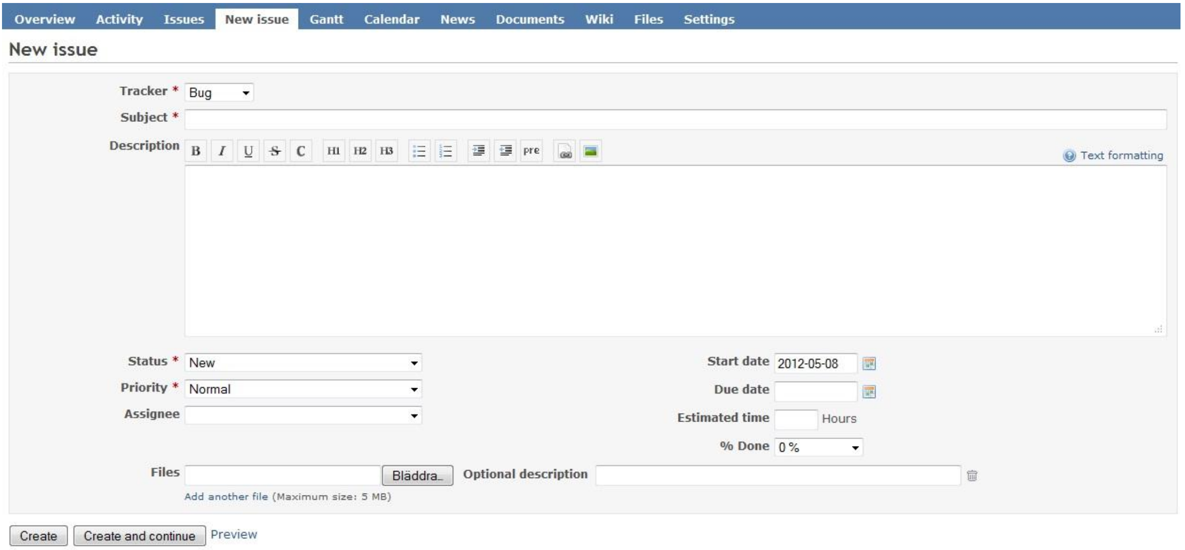
Test strategier varierar ofta väldigt mycket mellan företag och det finns ingen utformad standard. Detta debatteras kraftigt bland framträdande testare och många upplever att dokumentation ofta kostar mer än vad det ger, även om de är överens om att en del information måste dokumenteras. [23.0]
I detta kapitel diskuteras en lämplig test strategi för Resurs Bank, där fokus ligger kring webbapplikationen Butiksservice 2.
Dokumentation sker löpande genom testningens gång, och registreras i Redmine. Buggar och de delar av programvaran som testats kommer att dokumenteras. Testplanens syfte är att skapa en överblick över vad som testats i projektet.

4.1 Redmine

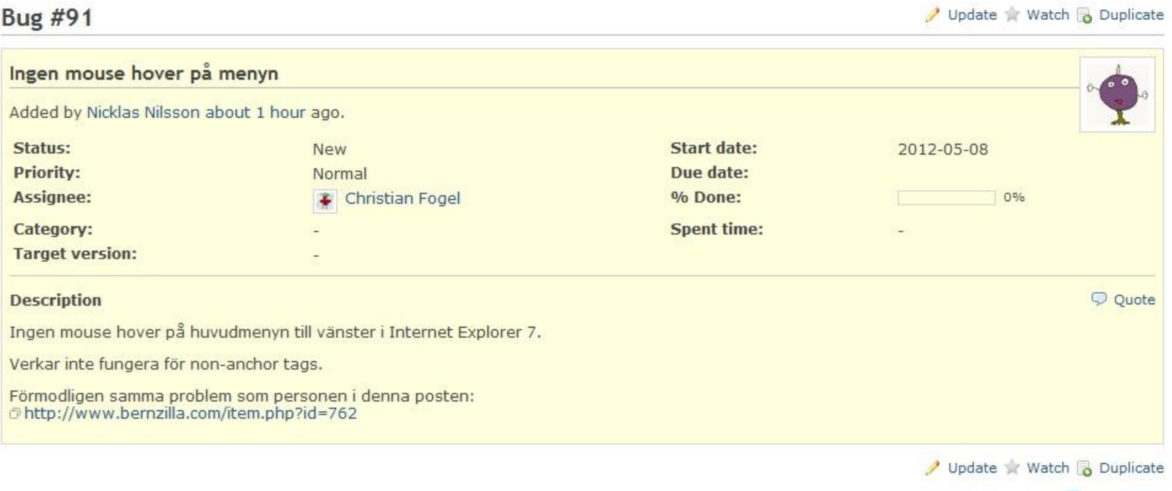
Redmine är ett gratis open source, webbaserat projekt och bugghanterings system som används på Resurs Bank. Detta kommer att utnyttjas till all dokumentation. Till Redmine finns ett stort antal plugins. Bland annat ett plugin för att göra mindmaps vilket skulle kunna användas till testplanen.

4.2 Buggrapportering

Buggrapporteringen sker genom Redmine. Då används en förinställd mall på hur en buggrapport ska se ut. Se Figur 4-1. Denna rapportering sker av testare och ibland även av utvecklarna.
Följande detaljer kan läggas in i buggrapporten:
Tracker – Vilken tracker man vill lägga ärendet i, som i detta fall är ”Bug”.
Subject – En kort passande rubrik för problemet.
Description – En längre beskrivning av problemet, gärna om hur man kan återskapa buggen. Ingen onödig eller orelevant information.
Status – Status på aktuell buggrapport. New/In Progress/Resolved/Feedback/Closed.
Priority – Vilken prioritet buggen har.
Assignee – Den som fått ärendet.
Start date – Datum för när buggrapporten blev tillagd.
Due date – Datum för när buggen bör vara fixad
Estimated time – Estimerad tid för åtgärdande av bugg.
% done – Procentuellt tal på hur mycket som är åtgärdat.
Files – Möjlighet att bifoga filer till buggrapporten. T.ex. bilder, film eller ljud relevant till buggen.
Optional description – Alternativ beskrivning.



Figur 4-1 Buggrapportsmall



Figur 4-2 Exempel på buggrapport

4.3 Testplan

Ett sätt att hantera testplanen är i form av en mindmap. De delar som inte passerat testerna markeras med ett rött kryss och de delar som passerat testerna markeras med en grön bock. Testplanen uppdateras kontinuerligt och ger på så vis alltid en aktuell bild över testningen i projektet. Denna metodik är kraftigt inspirerad av Christin Wiedemanns föredrag från Swedish Workshop on Exploratory Testning 9-10 april 2011 och skulle kunna vara lämplig att använda på Resurs Bank. Den är lätt att uppdatera genom programmet FreeMind och lättöverskådlig då man tydligt ser statusen för testningen.

4.4 Context-Free Questions

Context-Free Questions är frågor man kan ställa sin uppdragsgivare samt utvecklarna av produkten för att få en bättre insyn till vad det är man testar, vad meningen och målet med produkten är.
En del frågor har under arbetets gång redan besvarats och en del var inte aktuella för Butiksservice 2. Se kapitel 9.2 Context-Free Questions

Frågorna ställdes till, och besvarades av Christian Fogel, en utvecklare av tjänsten.

- Who is my client?

– Utvecklarna av produkten.

- Who is the customer of the product?

– Ombuden hos Resurs Bank.

- Who are the other stakeholders?

– Ombuden hos Resurs Bank, integratörer, PC-kassa - programvaran som används av ombuden och kommunicerar med webbservicen, slutkunderna.

- What problems are you aware of that would threaten the value of this product or service?

– Att inte all funktionalitet fungerar eller finns.

- How much time do I have?

– Inga tidsramar.

- How long before the next release or deployment?

– 1 vecka, sedan 2 veckors intervall.

When do you want reports or answers?

– Några dagar innan deploy.

- Is there another one like it?

– Butiksservice 1, som nuvarande används, men ska komma att ersättas.

- Could you draw me a diagram of how it works?

– Det fungerar ungefär såhär (Christian ritar upp ett diagram, återskapat i kapitel 5.3 Butiksservice 2)

- How would I recognize a problem?

– Webbservicen ger felsvar, exceptions, Soapfaults.
– WebbGUI ger felmeddelande i GUIt och service error.

- Who built this thing?

– Kalle Tjärnlund, Erik Cedergren, Christian Fogel. Utvecklare på Resurs Bank.

- Can I talk to them?

– Ja. Att vi arbetar i samma rum gör det lätt att snabbt nå rätt folk.

- Have they ever built anything like this before?

– Alla har jobbat med liknande arbete förut.

- Who else knows something about this?

– Anette Casparson projektledare på Resurs Bank, och beställaren Anders Engstedt

- Has anyone else tested this?

– Unit tester och enstacka SoapUI tester, men ingen djupare testning har funnits.

- What information is available to me?

– Information angående produkten och projektet finns lagrat på Lime, Confluence, docs och subversion.

- Can I talk to the technical support people?

– Anette.

För övriga Context-Free Questions se Bilaga 9.2.

Denna intervju gjordes för att få information angående Butiksservice 2. Då intervjun gjordes så sent in i arbetet så fanns där ingen ny information att få in. Det rekommenderas därför att Context-Free Questions ställs i början av arbetet för att få en bättre start.

5 Resultat

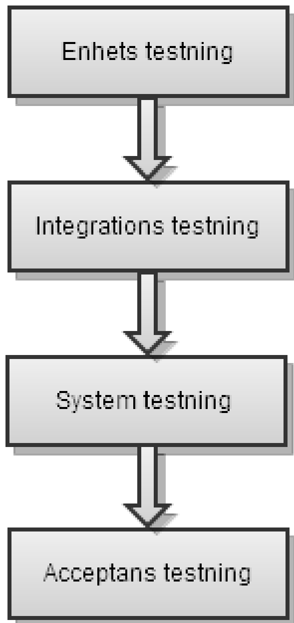
I detta kapitel beskrivs samtliga delar av resultatet utförligt. Viktiga val i arbetet motiveras och resultatet sammanställs.

5.1 Teknisk Bakgrund

Syftet med denna del av rapporten är att skapa en sammanfattning av de mest relevanta testmetoder och testtekniker som gåtts igenom inför implementeringen av arbetet.

5.1.1 Testnivåer

Tester blir ofta grupperade beroende på var de appliceras i utvecklingsprocessen (se Figur 5-1) eller beroende på hur noggranna de är. [21.0]



Figur 5-1 Testgrupperingar.

5.1.1.1 Enhetstestning

Detta är en nivå av mjukvarutestning där varje komponent/enhet testas var för sig. Syftet är att säkerställa att varje komponent fungerar som den ska. Detta utförs oftast av utvecklarna men testarna kan även hjälpa till vid tidsbrist.

5.1.1.2 Integrationstestning

I denna nivå sätts individuella komponenter/enheter ihop i grupper och testas. Den säkerställer att data överförs korrekt mellan enheterna. Enheterna sammansatta kallas för assemblages. Man kontrollerar även att dem fungerar

korrekt med hårdvaran. Dessa tester utförs ofta av både utvecklare och testare. Två vanliga tekniker beskrivs nedan.

Big Bang Teknik

En teknik där man väntar till alla enheter är klara innan man börjar testa. Detta är ofta tidskrävande. Det är också svårt att spåra rotorsaken till felen.

Inkrementell Teknik

Testar enheter direkt efter att dem är klara.

5.1.1.3 Systemtestning

Systemtestning av mjukvara eller hårdvara görs på ett färdig implementerat system för att utvärdera systemets överensstämmelse med kravspecifikationen. System testning faller in i kategorin black box testning och kräver därför ingen vetskap om koden eller logiken bakom mjukvaran. Man använder alla implementerade enheter som passerat integrationstesterna och applicerar hela lösningen på produktionslika servrar.

Test av hela systemet

System testning utförs på hela systemet med fokus på funktionskravsspecifikationen (FRS) och systemkravsspecifikationen (SRS). Man testar inte bara designen utan även beteendet av systemet och de förväntade kraven från kunden. Det är också meningen att testa mjukvarans/hårdvarans gränser och även att överbelasta.[12.0]

Typer av Systemtestning

Följande testmetoder bör man fundera på att använda under systemtestningen. Även om alla organisationer inte är överens om vilka metoder som bör användas i system testningen kan dessa metoder ses som en bra ram att börja med. [22.0]

Användargränssnittstestning

Användargränssnittstestning är när man testar mjukvarans användargränssnitt, och ser till så att de stämmer överrens med vad som skrivits i specifikationerna, men även att användarna är nöjda.

För att få fram en bra mängd testfall för användargränssnitt så måste man se till så att all funktionalitet har täckts, och att alla möjliga val på användargränssnittet har valts. Att testa detta på ett användargränssnitt är en väldigt stor uppgift. Ett exempel är Microsofts lilla program Wordpad, som har 325 olika val, så förstår man att det blir det väldigt många val på stora program att testa. Ett problem med GUI testning är när det gäller regressionstester. Eftersom mjukvaran kan ändra utseende för olika versioner så måste regressionstester skrivas om. Detta medför att man försöker autogenerera så många av testerna som möjligt.

För att slippa skriva om GUI testerna när ändringar sker så går man "under the hood". Istället för att testa med t.ex. en knapp så använder man knappens event för att nå samma funktion. På så sätt slipper man oroa sig ifall knappen ändrar plats i en ny version.

Användarbarhetstestning

Användarbarhetstestning är när man testar mjukvaran direkt mot användarna. Det är en bra teknik för att få input från användarna. Det är en black-box metod där man observerar användarna för att hitta fel och förbättringar. Områden att studera under tester kan vara t.ex.:

- Performance - Hur lång tid tar det för användaren att slutföra en uppgift?
- Accuracy - Hur många misstag gör användaren?
- Recall - Hur mycket kommer användaren ihåg efter testerna?
- Emotional response - Hur tycker användaren att det fungerat?

Prestandatestning

Prestandatestning är när man bedömer mjukvarans prestanda. Det kan vara antingen kvantitativa eller kvalitativa tester. De kvantitativa testerna fokuserar på saker så som svarstider, hur lång tid mjukvaran tar på sig att utföra en funktion. De kvalitativa testerna fokuserar på delar så som tillförlitlighet, skalbarhet och kompatibilitet. Prestandatester sker ofta i samband med stresstester.

Kompabilitetstestning

Kompabilitetstester är en del av den icke-funktionella testningen. Det görs för att se hur pass bra mjukvaran fungerar i olika datormiljöer. Olika datormiljöer kan vara:

- Kompabilitet med kringutrustning, som skrivare, läsare etc.
- Kompabilitet med olika operativsystem, som Unix, Windows etc.
- Kompabilitet med olika databaser, som Oracle, DB2 etc.
- Kompabilitet med olika systemprogram, som webbservrar, meddelandesystem etc.
- Kompabilitet med olika webbläsare, som Firefox, Internet Explorer etc.

Felhanteringstestning

Felhanteringstestning, eller exception handling, är en väldigt viktig del att testa för att få en stabil mjukvara. I programmeringsspråk finns det ofta funktioner som inte alltid returnerar ett "säkert" svar. Vilket i sin tur leder till en krasch ifall detta undantag inte hanteras korrekt. Ofta hanterar en så kallad exception handler undantaget. Det den gör är att spara undan information i det tillfället som undantaget sker, och leder sedan programmet vidare till en annan plats. Denna information kan användas vid ett senare tillfälle då man lyckats lösa undantaget.

Java använder sig utav så kallat throw och catch. Då ett undantag sker så gör java en throw på detta, och sedan catch, där den loggar information om undantaget.
Bra metoder att testa Error handling med är fault injections och mutation testning. För mer genomförlig förklaring se Fault Injection och Mutation Testning i kapitel 5.1.2.2.

Lasttestning

Denna teknik används för att se hur ett system presterar under hög belastning, men även under vanlig körning. Syftet är också att hitta eventuella flaskhalsar i systemet. Lasttestning är en typ av icke-funktionell testning.

Då man istället påfrestar systemet över dess normala rytm kallas detta för stress testning, då stressas systemet ofta så hårt att man förväntar sig att det kraschar eller att fel genereras.

Lasttestning refererar ofta till att man simulerar antalet användare som man tror kommer använda programmet, för att se så det inte blir överbelastat. Denna metod lämpar sig bäst för system med flera användare med en client/server modell så som webapplikationer, även om det används för annan mjukvara också.

Lasttestning låter dig mäta din Quality of Service prestanda baserat på faktiska kundbeteende. De flesta lasttestnings verktyg använder samma princip. Då en kund besöker programmet eller websidan så sparas kommunikationen och sedan genereras integrationsskript baserat på den datan. Sedan upprepar en belastnings-generator skripten, som kan modifieras med olika parametrar innan testet vid behov. Sedan övervakas och samlas information in under körningen från både mjukvaran och hårdvaran och en rapport genereras. Denna typ av testning appliceras oftast på en produktionslik testmiljö innan den sätts i produktion.

Volymtestning

Volymtestning går ut på att testa mjukvaran med en viss mängd data. Denna mängd kan vara databasens storlek eller storleken på en interface-fil beroende på målet med testningen. Detta för att testa hur mjukvaran hanterar reella mängder data. Denna typ av testning tillhör icke funktionella tester.

Stresstestning

Denna typ av test avser att kontrollera stabiliteten av ett givet system. Man testar utöver systemets gränser, ofta tills det kraschar för att se var bristningsgränsen ligger.
Man testar systemets robusthet, tillgänglighet och felhantering under höga påfrestningar. Målet med dessa tester är att säkerställa att mjukvaran inte kraschar på grund av otillräckliga dataresurser (så som minne eller hårddisks brist), ovanligt hög “concurrency” eller DOS-attacker.

Säkerhetstestning

Säkerhetstestning sker för att kontrollera så att systemet är tillräckligt säkert, att all data skyddas, men även så att dess funktionalitet inte hindras av det. Det finns en del olika områden som samknyts med säkerhet, så som sekretess, integritet, autentisering och tillgänglighet.

- Sekretess - En säkerhetsåtgärd som skyddar utlämnande av data till andra än dem det är avsedd för.
- Integritet - En åtgärd som låter mottagaren fastställa att den information som den får är korrekt
- Autentisering - Bekräftelse av identiteten hos en person eller produkt.
- Tillgänglighet - Se till så att informations- och kommunikationstjänster är tillgängliga när de förväntas att vara det.

Det finns en mängd olika sätt att utföra säkerhetstester, några kan vara:
Sårbarhetsskanning - Letar efter kända säkerhetsproblem med hjälp av automatiska verktyg för att matcha villkor med kända sårbarheter.

- Penetreringstest - Simulerar en attack från en skadlig mjukvara.
- Säkerhetsanalys - Analyserar risken i en specifik funktion.
- Säkerhetsomdöme - Kontrollerar så att mjukvaran håller de standarder som sätts på den typen av mjukvara.

Scalability Testning

Scalability testning sker för att se hur mjukvaran kan hantera stora expanderingar. Det kan vara direkt i mjukvaran, så som datavolymen. Men även indirekt till mjukvaran, så som datorer kopplade till mjukvaran. Prestanda, skalbarhet och tillförlitlighet betraktas som en helhet av programvarans kvalitet. [13.0]

Sanity testning

Ett sanity test är enkelt test för att snabbt se om ett resultat från en uträkning är rimligt.
Poängen med dessa tester är att skapa ett antal testfall med uppenbart felaktiga resultat, inte att fånga upp alla tänkbara fel. Fördelen med dessa test jämfört med fullständiga tester är snabbheten. Dem används som en grov kontroll av mjukvarans funktionalitet.

Både smoke och sanity tester har gemensamt att om något av dem inte godkänns så är det inte lönsamt att fortsätta med testningen. Många företag använder automatiska sanity tester i kombination med unit tester. Ibland används det också som ett verktyg då man manuellt felsöker mjukvaran. Ett komplett system innehåller ofta många mindre subsystem mellan indata och utdata. När ett komplett system inte fungerar kan sanity tester vara användbara för att se vilket subsystem man skall angripa. [14.1]

Det klassiska “Hello World” programmet används ofta som en typ av sanity test. Om man inte kan kompilera eller exekvera det programmet så är det oftast problem med konfigurationen av utvecklingsmiljön. Om det fungerar så ligger problemet oftast i det riktiga programmet.
Ett mer vanligt användningsområde är för att kontrollera funktioner i program där man skickar in argument och förväntar sig ett visst resultat.
Ett annat exempel av sanity tester är ett banksystem där man inte skall kunna ta ut mer pengar än vad som finns på kontot. [14.0]

Smoke testning

Smoke tester refererade från början till fysiska tester som gjordes på slutna rör-system för att leta efter läckor. Detta begrepp har sedan applicerats på många olika industrier, där ibland mjukvaruindustrin.
Inom programmering och mjukvaruutveckling, är det ett krav att smoke tester är godkända för att ens försätta med andra tester eller installationer. Meningen är att avslöja enkla fel som ändå är allvarliga nog för att avbryta en release. En mängd testfall som täcker den viktigaste funktionaliteten väljs ut och körs, detta för att se att grund funktionaliteten fungerar i programmet. T.ex. ställs ofta väldigt grundläggande frågor så som "Startar programmet?", "Öppnas en ruta då programmet körs?" eller "händer det något om man klickar på starta knappen?". Syftet är att säkerställa att inte mjukvaran är så illa skadad att vidare testning är onödig. Smoke tester utförs både av utvecklare och även av testare innan vidare testning utförs. [15.0] [15.1]

Microsoft påstår att efter "code reviews" är smoke testning den mest kostnadseffektiva testmetoden som finns. I Microsofts fall används smoke tester för att validera mjukvaruförändring precis innan den körs genom source control systemet. [15.2]

Smoke tester kan både utföras genom manuella och automatiska tester. Då man använder automatiska tester så körs dem ofta samtidigt som projektet byggs.
Smoke tester kan grovt kategoriseras som funktionstester eller enhetstester. Då dem körs på ett enskilt bygge kan dem ibland beskrivas som ett "build verifications test". [15.3]

Ad hoc testning

Ad hoc-testning är en form av mjukvarutestning som utförs utan en detaljerad plan, och utan krav på dokumentation. Testerna körs ofta endast en gång, vilket även orden "ad hoc" refererar till. Ofta utförs denna metod som en form av improviserad testning. Testaren försöker hitta fel med alla tänkbara hjälpmedel som verkar lämpliga just då. Ad hoc-testning kan ses som en del av exploratory testning. [16.0]

Ad hoc-testning har varit hårt kritiserad för att inte vara strukturerad, vilket också stämmer, men detta är även metodens styrka då man kan hitta fel mycket snabbt.

Motsatsen till denna metod är regressionstestning som syftar till att se om ett specifikt fel inte inträffar. Där har man tydliga steg och ett förväntat resultat.

Tillförlitlighetstestning

Att ha ett tillförlitligt system är viktigt inom alla områden. Meningen med reliability testning är att se hur pass tillförlitligt systemet är, man vill se hur långt man faktiskt kan gå innan man får ett allvarligt fel, och ha möjligheten att få ett mätvärde på det.

När ett fel hittas skickas det till utvecklarna som fixar detta. Sedan testas den nya versionen igen. Sedan jämförs resultaten, så som fel/transaktion eller fel/timme, för att bestämma mjukvarans genomförbarhet och för att kolla så att mjukvaran håller de tillförlitlighetskrav som kunden satt. [17.0]

Installationstestning

Installationstestning är en sorts kvalitétsförsäkran i mjukvaruindustrin som fokuserar på vad kunderna kommer att behöva för att kunna installera och få mjukvaran att fungera som det är tänkt. En del påstår att installations testning är en del av de viktigaste testerna som görs då installationen är den första interaktionen som kunden har med produkten.

Då möjliga sätt att installera mjukvara på ständigt ökar så har denna typ av testning även fått en viktigare roll.

Beroende av vilken typ av installations testning som utförs så kommer det finnas många faktorer som spelar in, så som vilket operativsystem används och hur produkten skall distribueras. [18.0]

Underhållstestning

Underhållstestning används för att identifiera och diagnosera utrustningsproblem eller att se att en reparation blivit lyckad. Bra testrutiner sparar testresultaten och åtgärder som vidtagits. Dessa utvärderas sedan för att se en trend som sedan används som bas för hur frekvent dessa tester skall utföras.

Tillgänglighetstestning

Denna typ av testning är till för att kontrollera att produkten är tillgänglig för alla tänkbara användare av den. Det finns många anledningar till att utföra dessa tester. Typiska tillgänglighetsproblem kan kategoriseras i fyra grupper.

Motorik
Som att inte kunna använda en mus eller ett tangentbord, eller att kunna göra känsliga rörelser.

Nedsatt hörsel
Som nedsatt eller helt utan hörsel

Nedsatt syn

Kognitiva förmågor
Lässvårigheter, dyslexi eller problem att minnas.

Utvecklare kan se till att produkten är tillgänglig genom unit tester och kod granskning. Testare ser till att produkten är tillgänglighets kompatibel genom funktions testning. I många fall används checklistor som innehåller information om vad som ska testas, hur det ska testas och statusen för produkten inom de olika grupperna. Typiska testfall för tillgänglighets kompabilitet kan se ut som följande exempel.

- Se till att alla funktioner finns tillgängliga via tangentbordet
- Se till att information är synlig då visningsläget är ändrat till hög kontrast läge.
- Se till att skärmläsnings verktyg kan läsa all text som är tillgänglig, även bilders alternativa texter.

Det finns många verktyg för dessa ändamål och det räcker oftast inte med att man bara använder ett. Då man använder Windows plattformen så är mycket av detta inbyggd genom Microsoft Active Accessibility (MSAA). [19.0]

Regressionstestning

Regressionstesters syfte är att hitta fel efter en större kodändring skett i mjukvaran. Dessa ändringar kommer ofta i samband med att mjukvaran slutat fungera och stora nya ändringar krävs för att få den att fungera igen. Man kör de tester som fanns med paketet från start för att se att inte äldre fel kommit tillbaka. Hur noga regressionstesterna görs avgörs ofta av hur hög arbetstakt man håller och hur stor risk man anser att den nya funktionaliteten utgör. Regressionstester förekommer ofta som oväntade konsekvenser av en större mjukvaruförändring och bör hållas automatiska, testfokus bör vara på den nya koden.

5.1.1.4 Acceptanstestning

Acceptanstestning används för att säkerställa att alla krav för en produkt är uppfyllda innan den slutliga produkten färdigställs. Detta är en form av black-box test som ofta utförs av kunden och kallas då för UAT (user acceptance testning). User storys anses aldrig vara klara förrän dem passerat acceptans testerna.
Ett smoke test används som acceptans test på en produkt innan den går in i huvudtest fasen.

Typer av Acceptanstestning

Alpha Testning

En liten del av de anställda eller potentiella kunderna testar mjukvaran. Detta sker ofta i ett skede då mjukvaran i princip är färdig och ses ofta som ett internt acceptanstest innan betatestningen börjar.[20.0]

Beta Testning

Betatestningen börjar efter alphatestningen och kan ses som ett externt acceptanstest.

Tidiga versioner av mjukvaran, beta versioner, släpps till en grupp personer utanför utvecklingsteamet. Detta för att hitta så många fel som möjligt innan programvaran släpps. Man låter ofta allmänheten ansöka om att få bli beta-testare men ibland släpps även betaversioner helt öppet till allmänheten för att få maximal feedback från så många framtida användare som möjligt.

5.1.2 Box grupperingar

Testmetoder inom mjukvarutestning delas ofta in i grupper, till största del beroende på om man har tillgång till koden och algoritmerna i programvaran. Dessa grupper heter Black-box, White-box och Gray-box och illustreras nedan. (se Figur 5-2)

Jämförelse mellan testmetodgrupperingar

Funktioner	White-box	Black-box	Gray-box
Test av programkod	Ja	Nej	Delvis
Ritningar och arkitektur tillgängligt	Ja	Nej	Delvis
Algoritm testning	Ja	Nej	Nej
Fullständig testning	Ja	Nej	Delvis
Metoder som liknar en angripares/hackares	Nej	Ja	Ja
Testas av slut-användare	Nej	Ja	Ja
Utförs av testare	Ja	Ja	Ja
Utförs av utvecklare	Ja	Delvis	Delvis

Figur 5-2 Egengjord tabell med jämförelse mellan testmetodgrupperingar.

5.1.2.1 Black-box Test Metoder

Black-box testning behandlar mjukvaran som om den var en svart låda. Testaren har ingen kunskap om hur den är implementerad och testar därför bara funktionaliteten. Kan appliceras på alla test nivåer.

Fuzz testning

Fuzz testning är en ”negative-testning brute force-metod” och används för att testa säkerheten i program. Det används för att testa ifall programmet kan hantera undantag utan att krascha, snarare än att se ifall det uppför sig korrekt. Fuzz testning hittar bara väldigt simpla fel, men de är oftast väldigt allvarliga, som kunnat utnyttjas av obehöriga om de inte rättas till.
De fel som fuzz testning hittar är oftast sådana som en mänsklig testare förbiser, eller skulle missa att skapa testfall för.

Termen härstammar från Barton Miller, University of Wisconsin 1988, och var från början ett skolprojekt. Projektets syfte var att testa pålitligheten på UNIX-program genom att skicka massvis med slumpmässig data till programmet tills det kraschade. 1995 utvecklades testet ytterligare för att kunna testa GUI-baserade program. [2.0]
Ett av de tidigare tecknen på Fuzz Testning är “The Monkey”, som var ett Macintosh program, utvecklat av Steve Capps 1983. Det användes för att leta efter fel i MacPaint. [2.1]
Fuzz testning sker hel eller halv automatiskt och skickar ogiltig, oväntade eller slumpmässig data som indata till mjukvaran. Sedan övervakas programmet för att hitta minnesläckor eller systemkrashar.

Modellbaserad testning

Modellbaserad testning är en metod där man ritar upp en liten del av systemet, de olika tillstånden man vill att den ska följa och kopplar samman dessa. Utifrån detta kan man sedan skapa testfall för den delen av systemet. Testerna som skapas utifrån detta anses vara funktionella tester. Eftersom testerna skapas utifrån en modell, och inte själva koden, så anses model-based testning vara en form av Black-box testning. Dock finns det undantag då man faktiskt kan involvera koden till de olika tillstånden.

Modellerna för modellbaserad testning skapas före eller parallellt med utvecklingen. Modellerna kan dock även skapas utifrån ett komplett system. Modellerna skapas oftast manuellt, men det finns språk som kan skapa dessa automatiskt utifrån koden.

Det finns olika sätt att använda sig utav modellbaserad testning, så som

“Online testning”, “Offline generation of executable tests” och “Offline generation of manually deployable tests”. [3.0]

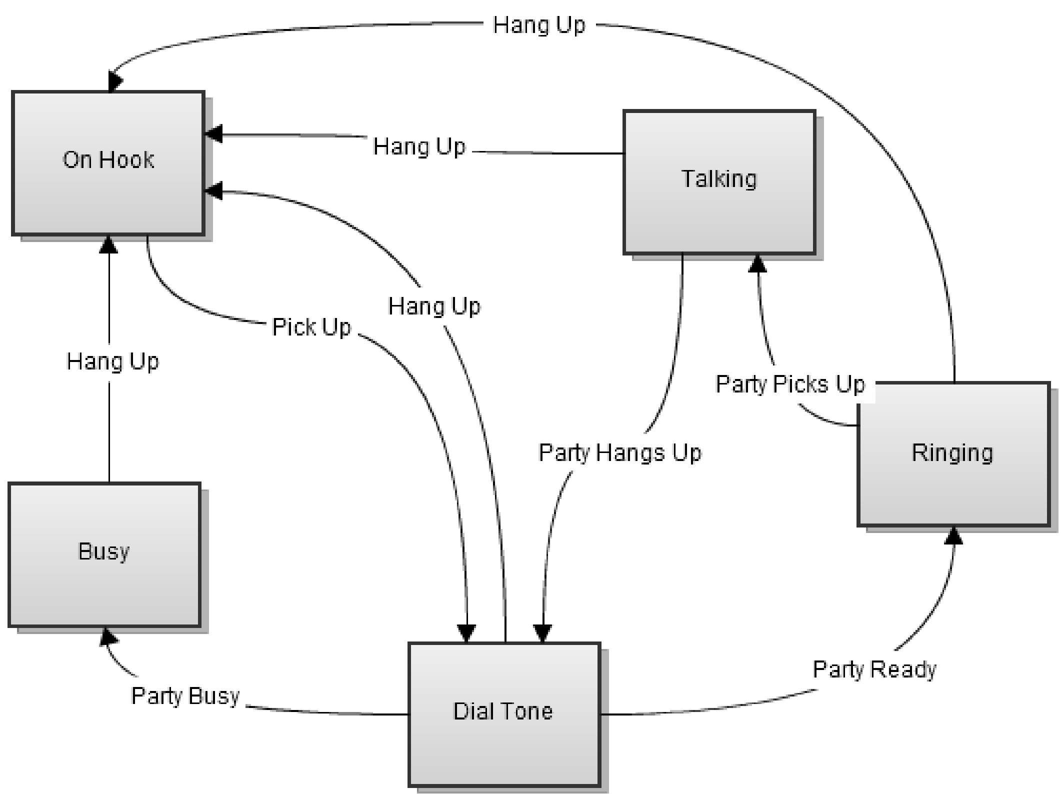
- Online testning - är när ett verktyg kopplas upp online till systemet och testar det dynamiskt.
- Offline generation of executable tests - är när ett verktyg skapar testfall i ett maskinläsbart språk som senare kan användas för automatiska tester.
- Offline generation of manually deployable tests - är när ett verktyg skapar testfall i ett läsbart format, t.ex. .pdf, som senare kan användas för manuella tester.

Effektiviteten som modellbaserad testning ger beror mestadels på den automation som den erbjuder. Om verktyget ger automatiska tester, och modellen är väldefinierad, så kan testningen erhållas helt mekaniskt. När ett verktyg skapar testfall så letar den efter körbara vägar i modellen. En möjlig körväg blir då ett testfall. Beroende på hur komplext systemet är, och hur många olika vägar man kan ta i modellen, så kan antalet testfall bli väldigt många. För att hitta relevanta körsvägar att testa så måste verktyget bli guidat. För detta finns det en mängd olika tekniker:

Test case generation by:

- Theorem proving - Testfallen väljs utifrån de logiska uttryck som anger systemets beteende.
- Constraint logic programming and symbolic execution - Testfallen väljs utifrån de krav och begränsningar som beskriver systemet.
- Model checking - Testfallen väljs utifrån de egenskaper som beskriver modellen som testfallen kommer ifrån.
- Using an event-flow model - Testfallen väljs utifrån de händelser i flödesmodellen som beskriver alla möjliga vägar ett system kan ta sett från ett GUI-perspektiv.
- Using a Markov chains model - En användning eller statistik-metod där man jämför alla möjliga användningsfall med de fall som en användare vanligtvis skulle använda. På så sätt kan man härleda testfallen till att inrikta sig på de viktigaste delarna först.

- Figur 5-3 visar ett exempel på en modell som kan användas för testning.



Figur 5-3. Modellbaserad testning *där man kan se relationer mellan funktioner, och på så sätt bedömma testningsfall. Bild inspirerad från källa. [3.1]*

Exploratory testning

Ses ofta som en black-box teknik även om en del påstår att det kan appliceras på alla tekniker och i vilket utvecklingsstadie som helst. Det är en självständig teknik där kvalitén på testningen är väldigt beroende av testarens färdigheter. Den moderna definitionen av exploratory testning säger att den är ett angreppssätt och inte en teknik för testning. [4.1]

Exploratory testning myntades först av Cem Kaner 1983. Ad hoc testning var på tidigt 90-tal förknippat och ofta detsamma som slarvig och ovarsam testning. Som ett resultat av detta började en grupp testare (idag Context-driven School) använda exploratory testning mer frekvent. Den första publikationen av denna teknik gjordes av Cem Kaner i sin bok Testning Computer Software och utökade detta kapitel i boken Lessons Learned in Software Testning. [4.2] [4.3]

Testningens kvalitet är väldigt beroende av testarens skicklighet och färdighet i att uppfinna testfall och hitta svaga punkter i programvaran. Desto mer testaren vet om testmetoder och programvaran, desto bättre kvalitet kommer testningen att hålla. För att ytterligare förklara så är dess antites skriptad testning. Med skriptad testning förväntar man sig ett visst resultat vilket man inte gör i exploratory testning där man istället håller alla förväntningar öppna. Testaren utforskar mjukvaran genom att observera, konfigurera och navigera runt i funktionerna för att sedan utvärdera produktens beteende och rapportera det som ser ut som fel.

På de flesta företag används en kombination av exploratory testning och skriptad testning. Enligt många, där ibland Cem Kaner och James Markus Bach, är exploratory testning mer ett mindset än en testmetod.
Hur mycket som dokumenteras skiljer sig ofta åt kraftigt, från att man endast dokumenterar fel till att man även dokumenterar alla test.
Vid par testning skapas ofta testfallen tillsammans, sedan utförs testerna av en person medans den andra dokumenterar dem. [4.4]

Det är inte ovanligt att man använder diverse hjälpmedel, som att ta skärm dumpar eller spela in video samtidigt som mjukvaran testas. Det finns även hjälpmedel som snabbt genererar de test- scenarios som är av intresse.
En av de största fördelarna med denna typ av testning är att det krävs ytterst lite förberedelser för att komma igång, fel hittas fort och det är ofta mer intellektuellt stimuleraande för testaren än vad skriptade tester är. En annan fördel är att testaren efter hand kan styra och anpassa testerna till de delar som verkar mest kritiska. Detta gör att man ofta hittar det allvarligaste felen fort.
En nackdel är att testerna ofta skapas under testningen och inte kan bli granskade under utvecklingsprocessen och därmed förhindra tidigt i utvecklingsfasen. En annan nackdel är att det ofta blir svårt att visa exakt vilka tester som man kört.

Det finns både för och nackdelar då man återvänder till tester man kört tidigare med exploratory testning. Det är svårt att återskapa exakt samma scenario som tidigare vilket kan bli ett problem om man vill återskapa ett fel. Men det medför även att man ibland upptäcker nya fel vid denna typ av regressions testning. Exploratory testning lämpar sig bäst om specifikationer och krav inte är färdigskrivna eller om det är tidsbrist i projektet. Den är även bra för att kontrollera att föregående testning hittat de mest kritiska felen. [4.0]

Equivalence Partitioning

Equivalence Partitioning, eller Equivalence Class Partitioning, är en teknik som används för att minska antalet testfall på klasser. Det används mest för att dela in indata, men kan även implementeras på utdata. Equivalence Partitioning är dock ingen fristående testteknik, utan måste kompletteras utav Boundary-Value Analysis.

Ett exempel är att det skulle ta väldigt lång tid att testa alla siffror från 1-100. Man kan därför dela in dem i olika partitioner som troligen ger samma utfall, och endast testa ett fall av varje partition. Teoretiskt sett räcker det att testa varje partition en gång för att se att programmet uppför sig som det ska, då flera test av samma partition inte skulle ge ett annat resultat eller hitta andra fel. Enligt figur 5-4 nedan kan man se hur Equivalence Partitioning fungerar, där ett program använder månadens siffra som input (1-12, januari - december), och all annan data anses vara ogiltig data. I det fallet räcker det med tre testfall, ett för under det giltiga området, ett för i det giltiga området och ett för över det giltiga området.

```
     ... -2 -1  0 1 ............. 12 13  14  15 .....
   --------------|------------------|---------------------
invalid partition 1     valid partition    invalid partition 2
```

Figur 5-4. Ett exempel på partitionering.

Ett annat sett att använda Equivalence Partitioning är att hitta så kallade "Dirty Test Cases" bland redan existerande testfall. Om man tar exemplet ovan, så skulle en oerfaren testare omedvetet skriva testfall för alla tillåtna indata (1-12), och glömma bort och testa den ogiltiga datan. På så sett skulle det finnas många onödiga testfall, och viktiga testfall skulle saknas.

Följande riktlinjer kan följas för att skapa Equivalence testklasser:
Om indata anger ett intervall, definiera minst en giltig och två ogiltiga klasser.

- Om indata kräver ett specifikt värde, definiera en giltig och två ogiltiga klasser.
- Om indata anger en medlem av en uppsättning, definiera en giltig och två ogiltiga klasser.
- Om indata anger en boolean, definiera en giltig och en ogiltig klass. [5.0]

Boundary-Value Analysis

Används i samband med Equivalence Partitioning. Med Boundary-Value Analysis testar man minsta och högsta värdet i varje partition, då dessa är områden där fel oftast inträffar. Även här kan testningen utföras både på in- och utdata. Om man jämför med Equivalence Partitionings tre testfall på Figur 5-5 nedan, så kommer Boundary-Value Analysis testfall istället hamna på 0,1 och 12,13, vilket är fyra testfall. Boundary-Value Analysis behöver inte ha ogiltiga partitioner.

```
      ... -2 -1  0 1 .............. 12 13  14  15 .....
     --------------|-------------------|---------------------
invalid partition 1     valid partition    invalid partition 2
```

Figur 5-5. Ett exempel på partitionering.

Följande riktlinjer kan följas för att skapa Boundary-Value testfall:

- Undre gräns - 1
- Undre gräns
- Undre gräns + 1
- Övre gräns - 1
- Övre gräns
- Övre gräns + 1
  [6.0]

All-Pairs testning

All-pairs testning, eller pairwise testning, är en testmetod som för alla par av indata testar alla möjliga kombinationer av paren. Antalet tester uppgår till O(nm), där n och m är antalet möjligheter för varje parameter. De enklaste felen i en mjukvara uppstår oftast när en parameter är fel. De näst enklaste uppstår när två parametrar samverkar med varandra, vilket all-pairs testning testar. Fel som uppstår när tre eller flera parametrar samverkar är betydligt mindre vanliga, och betydligt dyrare att hitta genom omfattande tester.

Precis som de flesta andra testtekniker så kombineras All-pairs testning med andra tekniker då det är omöjligt för en teknik att hitta alla fel. All-pairs testning kombineras ofta med t.ex. unit testning och fuzz testning.

5.1.2.2 White-box testmetoder

White-box testning är motsatsen till black-box testning. Testaren har tillgång till all logik och algoritmer bakom programvaran och kan därför testa den interna strukturen.

API testning

API, eller Application Programming Interface, är en mängd funktioner och operationer som andra program kan exekvera mot mjukvaran. GUIn är väldigt sällan involverad i API testning, vilket gör det väldigt komplext att sätta upp de inledande miljöerna. På så sätt måste man ha ett sätt att kolla så att systemet är redo att testas när man handskas med API testning. Boundary-Value Analysis och Equivalence Partitioning är de tekniker som brukar användas med API testning.

Ifall ett API testfall lyckas är baserat på outputen. Några typer av API testfall:

- Output baserat på input - Denna typ är väldigt simpel, där man kontrollerar outputen baserat på inputen.

- Output som ej returnerar något - Denna typ kontrollerar de funktioner som ej returnerar något. Exempelvis borttagnings-funktioner, där man kan behöva gå in i databasen för att kontrollera att elementen tagits bort.

- Output som triggar andra API/evenemang/avbrott - Om outputen triggar andra API, evenemang eller avbrott så ska APIet spåra dessa.

- Output som uppdaterar datastruktur - Ungefär som utdata som ej returnerar något, där uppdatering av datastrukturen ska kontrolleras direkt i databasen.

- Output som modifierar resurser - Om outputen uppdaterar databas, ändrar register, dödar processer eller liknande, så ska detta kontrolleras.

API testning VS Unit testning

- API tester ägs av testaren, medan Unit tester ägs ut utvecklaren.

- API testning och Unit testning är båda på kodnivå, vilket gör att de båda kan konstrueras med samma verktyg.

- Unit tester konstrueras för att kontrollera varje modul, så som en klass eller funktion. Medan API tester konstrueras för att kontrollera ett paket med flera moduler i, ett så kallat bygge, och kontrollera integrationen

mellan dessa. API testerna tar även hänsyn till vad en slutanvändare skulle använt för input till de olika funktionerna.
[7.0]

Code Coverage

Code Coverage är ett sätt att mäta hur stor del av koden som täcks upp av testfall. Det mäts i procent.

Code coverage var en av de första metoder som uppfanns för systematisk mjukvarutestning. Code coverage nämndes för första gången 1963 i tidningen Communication of the ACM av Joan C. Miller och Clifford J. Maloney. [8.0] För att mäta hur pass stor del av mjukvaran som täckts av tester finns det en del kriterier att följa: [8.1]

- Function coverage - Har varje funktion blivit anropad?
- Statement coverage - Har varje nod i mjukvaran blivit testad?
- Decision coverage - Har varje möjlig väg i mjukvaran gåtts igenom?
- Condition coverage - Har varje booleskt uttryck blivit testad både för sant och falskt?
- Condition/decision coverage - Både Decision och Condition coverage ska vara uppfyllda.

Antag att denna funktion ska testas med code coverage. (se Figur 5-6)

```
int foo (int x, int y)
{
    int z = 0;
    if ((x>0) && (y>0)) {
        z = x;
    }
    return z;
}
```

Figur 5-6. Code Coverage exempel

Funktionen kommer då uppfylla:
Function coverage - Om funktionen har körts åtminstone en gång.
Statement coverage - Om funktionen körs som t.ex. foo(1,1), då alla rader kommer exekveras i detta fall.
Decision coverage - Om funktionen körs som t.ex. foo(1,1) och foo(1,0), då första körningen kör genom hela funktionen, medans andra körningen ej gör det. Condition coverage - Om funktionen körs som t.ex. foo(1,1), foo(1,0) och

foo(0,0). Då de två första fallen ger (x>0) sant, och den tredje ger falskt. Samtidigt som första fallet ger (y>0) sant, medans de två sista ger falskt.

Fault Injection

Fault Injection är en metod som testar kodvägar som annars inte brukar följas. Det används ofta vid stresstest och är en viktig del av testning för att få ett robust system. I dagsläget finns det ett antal verktyg som automatiskt gör sådana tester. [9.0]

Fault Injection introducerades under 1970-talet och användes från början till att inducera fel på hårdvara. Denna typ av tester kallades då Hardware Implemented Fault Injection (HWIFI) och var ett sätt att simulera hårdvarufel i ett system. De första hårdvarutesterna kunde vara så lätta som att kortsluta anslutningar på kretskort och observera effekterna. Det gjordes mest för att studera hur pass pålitligt hårdvaran för systemet var. Strax därpå upptäckte man att Fault Injections även kunde användas på mjukvara, Software Implemented Fault Injection (SWIFI). [9.1]

Fault Injection använder sig utav Time Based Triggers och Interrupt Based Triggers för att utlösa en injektion. Time Based Triggers aktiveras efter en specificerad tid, medans Interrupt Based Triggers aktiveras då ett specifikt exception eller en specifik del i mjukvaran körts.

Fault Injection använder en mängd olika tekniker för att testa mjukvaran, t.ex.

- Corruption of memory space: Denna teknik korrumperar RAM, processorregistret och Input/Output.
- Syscall interposition: Denna teknik fångar upp systemanrop som mjukvaror gör, och korrumperar dessa.
- Network Level fault injection: Denna teknik korrumperar, tar bort eller slänger om paket som finns i nätverket.

Mutation testning

Mutation testning är en metod att testa testfall. Man byter då ut operatörer i koden för att se ifall testfall detekterar detta och ger fel svar. Syftet med detta är att få fram effektiva testfall och hitta svagheter i testdatan. Fuzz Testning är en typ av Mutation testning. [10.0]
Mutation testning introducerades först 1971 av Richard Lipton. Första testverktyget med mutation testning implementerades först 1980 av Timothy

Budd. 2004 expanderades Mutation Testning ytterligare av Certess Inc. då de implementerade metoden att även testa hårdvara. Det som sker då är att den kollar på skillnader i utsignaler. [10.1]
Mutation testning sker genom att välja ut operatorer i koden som testas, och byta ut dessa en åt gången. En sådan utbytning kallas “Mutant”. Om testfallet sedan märker att resultatet är fel, då sägs det att mutanten är “dödad”. Då funkar det som de ska.

En mutation kan se ut enligt följande exempel:
En bit programkod i C++ ser ut enligt Figur 5-7 nedan.

```
if (a && b) {
    c = 1;
} else {
    c = 0;
}
```

Figur 5-7. Normalt uttryck.

En mutant kan då se ut enligt Figur 5-8 nedan, då && ersatts med ||.

```
if (a || b) {
    c = 1;
} else {
    c = 0;
}
```

Figur 5-8. Muterat uttryck.

För att döda ovanstående mutant borde följande punkter uppfyllas:

- Få fram olika resultat mellan originalet och mutanten. T.ex. skulle ett testfall med a = 1 och b = 0 få fram detta.
- Testfallet kontrollerar resultatet på variabeln c.

Mutation testning kan klassas i två typer, svaga mutationer och starka mutationer. Med svaga mutationer räcker det att första punkten är uppfylld, men det med starka krävs att båda punkterna är uppfyllda. Starka mutationer är kraftfullare, då det garanterar att testfallet fångar problemen. Svaga mutationer är mer relaterade till code coverage och kräver mindre datorkraft än starka.

Det finns även så kallade Equivalent mutants, som är ett stort problem inom mutation testning. Många mutanter kan skapa ekvivalent mutanter. Nedan visas ett exempel:

Följande bit programkod(se Figur 5-9) kan ha en ekvivalent mutant.

```
int index = 0;

while (…)
{
    …;
    index++;

    if (index == 10) {
        break;
    }
}
```
Figur 5-9. Normal equivalent mutant.

En boolesk mutant ersätter == med >=, och skapar följande kod (se Figur 5-10).

```
int index = 0;

while (…)
{
    …;
    index++;

    if (index >= 10) {
        break;
    }
}
```
Figur 5-10. Muterad Equivalent mutant.

Enligt exemplet så är det omöjligt att hitta ett testfall som skulle döda mutanten. Mutanten är då ekvivalent med den normala koden. Att hitta equivalent mutants är det största hindret inom användningen av mutation testning. Att kolla om en mutant är ekvivalent eller inte kan kräva en stor insatts, även för små program. [10.2]

Statisk testning

Statisk testning är en typ av test där man inte använder själva mjukvaran, motsatsen till dynamisk testning. Det är ingen metod för att få någon detaljerad testning, utan den används mestadels för att få förståelse och finna eventuella fel i kod, algoritmer och dokument. Denna typ av testning används oftast av den programmerare som skrivit koden. Statiska tester kan bli automatiserade. Mjukvaran blir då analyserad av en kompilator som anger mjukvarans giltighet. Fel som hittas i detta steg blir på så sätt billigare att reparera än om de hittats i en senare testmiljö. Både programmerare och testare kan göra denna typ av test.

5.1.2.3 Gray-box testning

Gray-box testning är en blandning mellan Black-box och White-box testning, där man delvis har tillgång till koden. Målet är att hitta fel som kan uppkomma på grund av felaktig struktur eller användning av programmet. Gray-box testning är användbart för webbapplikationer.

Spårbarhetsmatris

En spårbarhetsmatris är ett dokument, oftast gjort i form av en tabell, som ställer två dokument mot varandra för att kontrollera många till många relationer mellan dessa. Används ofta när det gäller högnivåkrav. [11.0]

Den enklaste användningen av spårbarhetsmatris är att ta identifierarna från det ena dokumentet och lägger dem i den översta raden, för att sedan ta identifierarna från det andra dokumentet och lägga dem den vänstra kolumnen. När en identifierare från den vänstra kolumnen kan relateras till identifieraren i den övre raden sätts ett kryss i deras korsning.
Sedan adderas värdena för varje rad och kolumn ihop. Ett värde på noll visar att ingen relation finns, och man diskuterar ifall det ska rättas till. Stora värden kan antyda på för komplex relation, och man diskuterar ifall det ska förenklas.

Exempel på Traceability Matrix illustreras i Figur 5-11.

Requirement Identifiers	*Reqs Tested*	REQ1 1.1	REQ1 1.2	REQ1 1.3	REQ1 2.1	REQ1 2.2	REQ1 2.3	...
Test Cases	*11*	2	2	1	2	3	1	...
Tested Implicitly	*11*							
1.1.1	2	x				x		
1.1.2	2	x	x					
1.1.3	3			x	x	x		
1.1.4	2		x		x			
1.1.5	2					x	x	
...	...							

Figur 5-11 exempel på spårbarhetsmatris.

Ortogonal vektor testning

Ortogonal vektor testning är ett systematiskt och statistiskt sätt att testa på. Det används när antalet inputs är relativt få, men för många för att göra en heltäckande testning. Ortogonala vektorer kan tillämpas i användargränssnittstestning, systemtestning, regressionstestning, konfigurationstestning och prestandatestning.

Varje vektor har följande egenskaper:

- Varje vektor skiljer sig från varje annan vektor i sekvensen.
- Varje vektors signal kan separeras lätt.
- Varje vektor är statistiskt oberoende av varandra.

Ett system med tre parametrar, där varje parameter kan ha tre värden. En fulltäckande testning skulle då kräva 27 testfall. Ortogonala vektorer kan maximera testtäckningen med minimalt antal testfall, genom att välja testfallen effektivt och enhetligt. De ortogonala vektorerna skulle då vara som Figur 5-12 visar.

Test Case	Parameter 1	Parameter 2	Parameter 3
1	1	1	3
2	1	2	2
3	1	3	1
4	2	1	2
5	2	2	1
6	2	3	3
7	3	1	1
8	3	2	3
9	3	3	2

Figur 5-12 Ortogonal vektor testning exempel.

Single Mode Faults: Single mode faults inträffar endast på grund utav en parameter. I exemplet ovan, om testfall 7,8 och 9 blev fel, så skulle detta bero på Parameter 1.

Double Mode Faults: Double mode fault inträffar när två parametrar agerar med varandra

Multimode Faults: Fel med fler än två parametrar kallas Multimode Faults.

5.1.3 Jämförelse mellan box grupperingarnas testmetoder

På nästa sida följer en jämförelse på för- och nackdelar med varje testmetod i de olika boxgrupperna, där grön färg indikerar fördel och röd färg indikerar nackdel.

Metod	Jämförelse mellan Black-box testmetoder			
Fuzz Testning	Felen som hittas kan ofta lätt utnyttjas av en attackerare, och är därför bra om dem upptäcks	Ökar säkerheten	Hittar de fel som en mänsklig testare skulle förbise	Ökar robustheten
	Liten chans att hitta fel då man använder slumpartade värden som input	Hittar ofta bara simpla fel		
Model-based Testning	Automatiska tester	Dynamisk testning	Mestadels automatiska tester	Modellerna som används för testfall kan skapas före och parallellt med utvecklingen
	Modellerna kan skapas automatiskt utifrån kod med hjälp av verktyg	En väldefinierad modell, kombinerat med automatisk testning, kan ge en helt mekanisk testning	Kan bli väldigt många testfall	
All-Pairs Testning	Hittar enkla fel	Hittar lätt fel på par av inparametrar	Antar att alla par är möjliga, och testar därför alla möjliga parbildningar	
Boundary-Value Analysis	Bra för att hitta användar input/gränsnitts problem	Lätt och tydligt att definiera testfall	Litet antal testfall behövs	Testar inte all möjlig indata
	Testar inte beroenden mellan kombinationer av indata			
Equivalence Partitioning	Designad så att all input data endast har en equivalence partition	Litet antal testfall behövs	Testar inte all möjlig indata	Inga riktlinjer för hur man skall välja indata
Exploratory Testning	Ingen dokumentation behövs	Ingen tidigare kunskap om applikationen behövs	Testarna kan hjälpa till med andra uppgifter tidigt in i utvecklingsprocessen	Lite förberedelser behövs och det går fort att komma igång
	Intellektuellt stimulerande för testaren	Testaren kan använda deduktivt resonemang från föregående resultat för att leda sig fram till nya testfall	Kvalitén på testningen är väldigt beroende av testarens färdigheter	Svårt att återskapa testfall
	Lätt att tappa fokus på grund av dess ostrukturerade arbetssätt	Testerna skapas oftast under testningen		

Metod	Jämförelse mellan White-box testmetoder			
API Testning	Säkerställer att APIet fungerar	Säkerställer att integrationen mellan olika bygger fungerar	Svårt att se ifall systemet är redo för API testning	Komplext att sätta upp de inledande miljöerna
Code Coverage	Automatiskt mätverktyg	Procentuellt tal på hur mycket av koden som testats	En del fel tenderar till att förbli oupptäckta trots stor code coverage	
Fault Injection	Kan automatiseras	Ökar robustheten i systemet	Kan användas på fel sätt av hackare	Real-life fel är svåra att upptäcka
Mutation Testning	Ökar kvaliteten på testfallen	Equivalent mutants		
Static Testning	Kan automatiseras	Fel kan hittas tidigt och blir då inte lika kostsamma som senare i projeketet	Snabbt och ger 100% coverage	Behöver inte köra mjukvaran
	Hälften av de fel funna genom dynamisk testning kan hittas tidigare med statisk testning	Många fel syns bara när mjukvaran kör		

Det som kan urskiljas ur tabellen är att Black-box metoderna oftast inte kräver någon hög tekniskt kännedom utan kan utföras av någon utan teknisk erfarenhet vilket inte är fallet med White-box metoderna. Black-box metoderna kan i större grad automatiseras än vad White-box metoderna kan. Den mest populära metoden idag är Exploratory testning som används i stor grad på de flesta företagen. Detta syns även i tabellen då det var lätt att hitta många för och nackdelar om den.

5.2 Resurs Banks nuvarande utvecklingsmetodik

Syftet med denna del är att beskriva Resurs Banks nuvarande utvecklingsmetodik. Denna information har framställts i samråd med utvecklarna på IT-avdelningen.

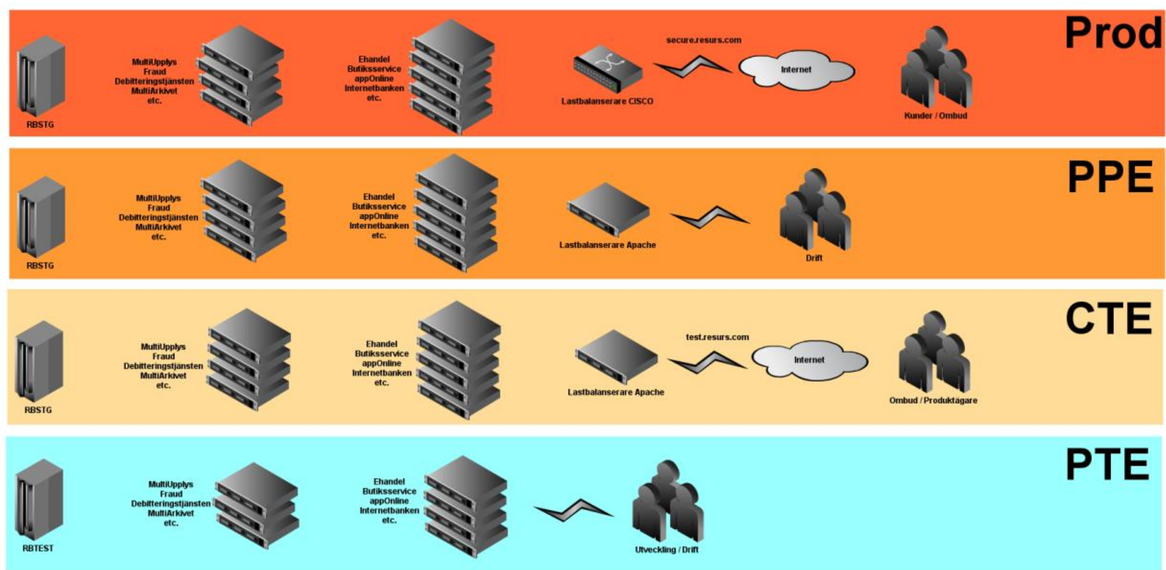
Resurs Banks utvecklare arbetar idag både med utvecklingen och testningen av programvaran och har begränsad tid och resurser att utföra en fullständig testning. 10 personer arbetar som utvecklare och 5 personer med drift.

Utvecklingsteamet arbetar enligt den Agila metoden Scrum[24.0] med modifikationen att de inte har några dagliga möten då den mesta utvecklingen sker självständigt.

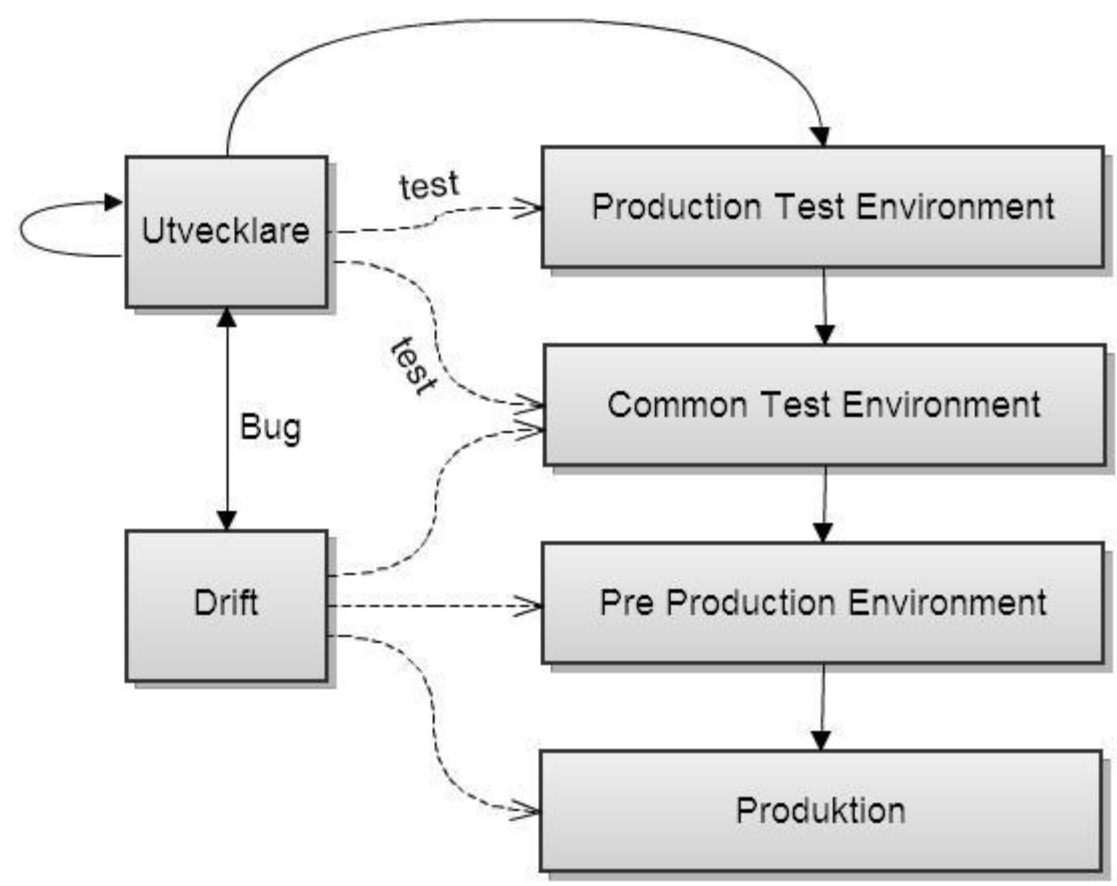
Två veckors utvecklingscykler används och man försöker alltid sätta produkten i drift på tisdagar. Detta med undantag för till exempel Butiksservice 2 som inte körs med iterationer. Ett driftmöte hålls ofta på torsdagen innan programvaran sätts i produktion för att se att allt är färdigt, och att drift paketet är klart.

Paketet läggs på *staging* servern, PPE (*Pre production environment*), där drift-personalen tar över. Figur 5-13 till 5-15 visar nuvarande arbetsflöde och de arbetsmiljöer som aktivt används på Resurs Bank. Figur 5-13 och 5-14 är hämtade från Resurs Banks intranät medan Figur 5-15 är egengjord.

Figur 5-13 Illustration av Resurs Banks arbetsmiljöer.



Figur 5-14 Illustration av Resurs Banks arbetsmiljöer.



Figur 5-15 Illustration av Resurs Banks arbetsmiljöer och arbetsflöde.

5.3 Butiksservice 2

De testmetoder och rutiner som utformats från detta examensarbete kommer att appliceras på projektet Butiksservice 2. I dagsläget används Butiksservice 1, men ska framöver ersättas av Butiksservice 2. Butiksservice är en tjänst där Resurs Banks ombud kan utföra kreditupplysningar på kunder, för att sedan ansöka om kredit. (se Figur 5-16 för tekniskt uppbyggnad)

Figur 5-16. Figuren visar en teknisk beskrivning om hur Butiksservice 2 är uppbyggt. Datavägar är ritade som heldragen linje och streckad linje representerar testning.

5.3.1 Funktionalitet för Butiksservice 2

Kommunikationen till Frontend servern kan ske genom två sätt, webbläsare med HTTP protokoll eller klient med SOAP/HTTP protokoll. Klienten består av en programvara som anropar webservicen, ofta för att skapa nya konton och utföra debiteringar.
Därefter sker kommunikationen vidare till Backend servern, som kontrollerar om ombudet har tillåtelse att gå in och göra en kreditupplysning.
Informationen om vad ett ombud får göra finns lagrade i databasen. Databasen går endast att ändra genom Backoffice. Backend är kopplat till Banksystemet, Multiupplys och Debiteringen. En kreditupplysning sker genom Multiupplys. Banksystemet innehåller all information om Resurs Banks ombud och kunder. En godkänd kreditupplysning möjliggör att konto skapas för kunden och att detta konto debiteras.

5.3.2 Befintlig testning av Butiksservice 2

Testning av Butiksservice 2 sker just nu på tre sätt, genom Selenium, SoapUI och Unit tester.

- Selenium testar de funktioner som sker från webbläsare.
- SoapUI Suite testar de funktioner som sker från klienter.
- Unit tester finns i stor utsträckning på Java koden.
- XSD, eller XML Schema, är bestämda mallar på hur informationen som kommer in ska vara formaterad (regular expressions), det finns en mindre mängd tester på denna del.

5.3.3 Butiksservice 1 vs Butiksservice 2

Butiksservice 1 hanteras helt utan versionshantering vilket gör det svårt att följa utvecklingen. Detta då historik för utvecklingen inte sparas och det blir svårt att se tillbaka vid funna buggar. Koden ändras även direkt på live-servern, vilket kan generera nya problem. Det saknar även en egen datamodell, och körs direkt ovanpå banksystemet. I Butiksservice 2 har versionshantering införts och en ny datamodell har byggts upp. Man använder även olika miljöer för testning innan det läggs ut på liveservern. Dokumentationen för Butiksservice 2 är dessutom betydligt bättre än för Butiksservice 1 då dokumentation på Resurs Bank värderas högre nu än då Butiksservice 1 utvecklades.

5.4 Val av testmetoder

De mest kritiska delarna att testa identifierades och därefter valdes de verktyg och tekniker som behövdes för att säkerställa kvalitén.
Lämpliga testmetoder till Butiksservice 2 valdes genom diskussion med utvecklarna, analys av kapitlet 5.1.3 Jämförelse mellan box grupperingarnas testmetoder och 4.4 Context-Free Questions.

Exploratory testning

En nästintill standard approach som bör användas på de flesta projekt. Ett bra sätt att hitta fel snabbt utan större förberedelser.

Input testning

Equivalence Partitioning, Boundary-Value Analysis, All-pair testning och Fault injection. Bra testning för webbsidan, för att säkerställa att felaktig input inte accepteras.

API testning

Testning för webbservicen.

GUI testning

Test av webbsidans användargränssnitt.

Kompabilitetstestning

Behövs för att säkerställa att tjänsten fungerar i rätt webbläsare.

- IE7 och uppåt
- FF 3 och uppåt
- Chrome, senaste versionen (uppdaterar sig själv)
- Safari, senaste versionen (uppdaterar sig själv)

Prestanda, Laddning, Stress, Skalbarhet och Volymtestning

Behövs för att säkra att tjänsten fungerar vid hög belastning.

Säkerhetstestning

För att testa säkerheten i tjänsten.

Installationstestning

Endast intern testning då Resurs Bank sköter installationen mot ombuden.
Även test för PC-Kassa och liknande applikationer som ombuden använder.

Tillgänglighetstestning

Krav enligt Butiksservice 1. Ombuden har ofta dåliga datorer som står i mörka miljöer vilket medför höga krav på designen av applikationen.

Regressionstestning

Skall ske efter stora förändringar i programvaran.

Acceptanstestning

Viktig del av testningen för att se att alla krav är uppfyllda.

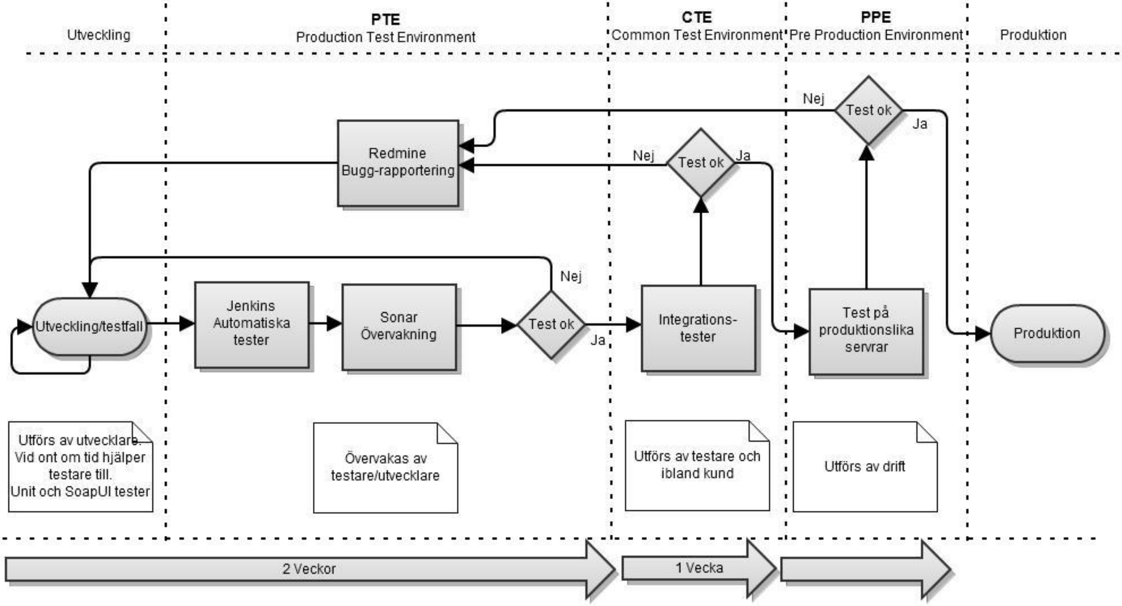
5.5 Resurs Banks nya testflöde

Testflödet visar hur kod och funktioner övervakas och testas från de att de börjat utvecklas tills de sätts i produktion.

Funktioner och unittester skrivs i utvecklingsstadiet av utvecklarna. Dessa unittester körs automatiskt i Jenkins och väldigt detaljerad data av testerna genereras från Sonar, som övervakar testningen. Om ett test fallerar, så ser utvecklarna detta och åtgärdar problemet. Om funktionen fungerar som förväntat, och unittesterna gått genom, så skickas paketet till Common Test Environment.

Efter en två veckorsiteration överlämnas paketet till testarna, vars uppgift är att testa integrationen mellan dessa funktioner, och redan existerande system, se till så att det fungerar som de ska och hitta eventuella buggar. De buggar som hittas av testarna rapporteras i Redmine, ett system där projektstatus rapporteras. Utvecklarna kan i Redmine se buggrapporterna och utifrån den informationen åtgärda problemet.
När ett paket är fungerande så skickar testarna det till driftpersonalen, som installerar paketet för produktion. Om oupptäckta fel ändå lyckats ta sig förbi testarna och hittas här, så rapporteras även det i Redmine.(se Figur 5-17)

Fördelarna ligger i att där är mer tid åt testningen, vilket kommer säkerhetsställa och ge högre kvalité på produkterna. Nackdelen ligger i att det är mer tidskrävande, men att det i slutändan kommer det vara värt den extra tiden då produkten får en högre kvalité.



Figur 5-17 Illustration av det nya testflödet.

5.6 Testplan

Testplanen visar de delar som ska och har testats. Författarna av detta examensarbete valde att använda sig av, och rekommenderar testare att använda, en mindmap som testplan då detta ger en klar bild över vad som har och ska testas, och även nuvarande status av testningen. Struktureringen av mindmappen sker lite på magkänsla, alltså att man lägger in det man tror kommer behövas testa, och man väljer de mest kritiska bitarna först. Eftersom det är väldigt flexibelt att strukturera en mindmap, så kan man lätt lägga till och ta bort testområden under testningens gång. Programmet som används för att skapa dessa mindmaps heter FreeMind och är open source.

Testplanen är strukturerad i tre huvudkategorier:

- Webbsida - all testning som involverar webbsidan, alltså testa tjänsten genom olika webbläsare.
- Webbservice - all testning som involverar webbservicen, alltså testa de olika anropen som görs med PC-kassa och SOAP.
- Prestanda - all testning som involverar prestandatestning av tjänsten.

Mindmappen använder följande ikoner för att representera en viss status:

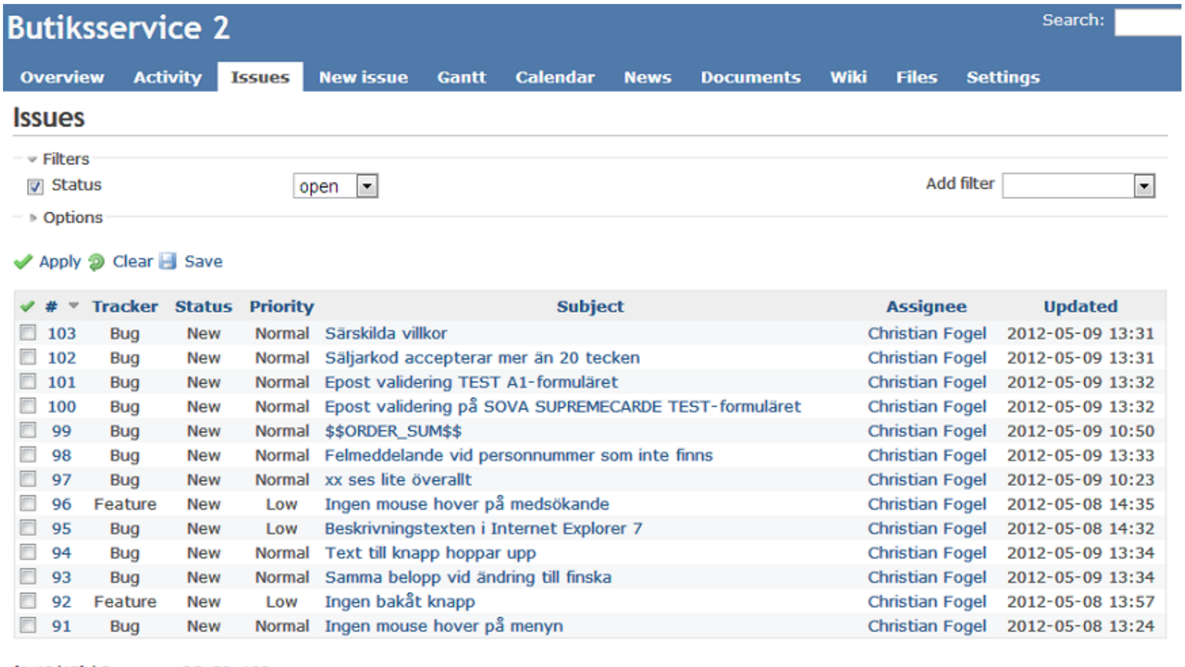
- Grön bock - fullt passerad testning i det området.
- Rött kryss - funnen bugg/buggar i det området.
- Gult papper med penna - notering i fritext.

Även ikoner för Firefox, Chrome, Safari och Internet Explorer används för att se vad som testats och i vilken webbläsare. När en funktion är fullt testad ersätts dessa ikoner av en grön bock eller rött kryss beroende på om funktionen passerat testningen eller ej. (se Figur 5-18)

Figur 5-18 Butiksservice 2 testplan, där testfall mot Webbservicen och Webbsidan har ritats upp, och även enstaka Prestandatester.

5.7 Funna buggar av Butiksservice 2

Testningen begränsades till funktionen ”Skapa konto” i Butiksservice 2, detta på grund av tidsbrist. De fel som hittades dokumenterades i Redmine enligt Figur 5-19. För mer information om vardera bugg se bilaga 9.3 Funna buggar.



#	Tracker	Status	Priority	Subject	Assignee	Updated
103	Bug	New	Normal	Särskilda villkor	Christian Fogel	2012-05-09 13:31
102	Bug	New	Normal	Säljarkod accepterar mer än 20 tecken	Christian Fogel	2012-05-09 13:31
101	Bug	New	Normal	Epost validering TEST A1-formuläret	Christian Fogel	2012-05-09 13:32
100	Bug	New	Normal	Epost validering på SOVA SUPREMECARDE TEST-formuläret	Christian Fogel	2012-05-09 13:32
99	Bug	New	Normal	$$ORDER_SUM$$	Christian Fogel	2012-05-09 10:50
98	Bug	New	Normal	Felmeddelande vid personnummer som inte finns	Christian Fogel	2012-05-09 13:33
97	Bug	New	Normal	xx ses lite överallt	Christian Fogel	2012-05-09 10:23
96	Feature	New	Low	Ingen mouse hover på medsökande	Christian Fogel	2012-05-08 14:35
95	Bug	New	Low	Beskrivningstexten i Internet Explorer 7	Christian Fogel	2012-05-08 14:32
94	Bug	New	Normal	Text till knapp hoppar upp	Christian Fogel	2012-05-09 13:34
93	Bug	New	Normal	Samma belopp vid ändring till finska	Christian Fogel	2012-05-09 13:34
92	Feature	New	Low	Ingen bakåt knapp	Christian Fogel	2012-05-08 13:57
91	Bug	New	Normal	Ingen mouse hover på menyn	Christian Fogel	2012-05-08 13:24

(1-13/13) | Per page: 25, 50, 100

Figur 5-19 Lista över de fel som arbetet resulterat i för Butiksservice 2. En del fel anses ej som buggar, utan är mer förslag på förbättringar.

6 Slutsats

Resultatet från denna rapport har överlag varit lyckat enligt Resurs Bank och de har nu en framtida grund för mjukvarutestning att utgå från och vidareutveckla vid behov. Det har varit en intressant och givande uppgift och de resurser och hjälp som behövts har alltid funnits tillgängliga. Inga av författarna hade någon tidigare erfarenhet av att arbeta som testare. Mycket information har bearbetas från bloggar, intervjuer och andra källor på internet. Att samla information om hur ett testflöde normalt ser ut och hur man testar av en produkt manuellt har tagit mycket tid.
En hel del fel hittades (se 5.7 Funna buggar av Butiksservice 2) även om endast en liten del av webapplikationen Butiksservice 2 har hunnit testats. Det tar lång tid att testa en applikation fullständigt och oftast testas endast de mest kritiska bitarna.

Detta arbete har resulterat i:
• En testplan lämplig för Resurs Bank.
• Testning av Butiksservice 2.
• Dokumentation om testmetoder lämpliga för Resurs Bank.
• Dokumentation om testtekniker lämpliga för Resurs Bank.
• 1 veckas testning är nu inkluderat i utvecklingsmetodiken.
• Övergång till projekthanteringssystemet Redmine.

Målet har från början varit att besvara frågorna i problemformuleringen, detta gås igenom i nästa del av slutsatsen.

6.1 Frågor från problemformulering

- **Vad är ett effektivt och bra testflöde för ett medelstort tjänsteföretag?**

Det finns ingen mall för hur ett bra testflöde är, utan det måste skräddarsys efter företaget och anpassas efterhand.

- Vad är det optimala testflödet för Resurs Bank?

Ett bra testflöde för Resurs Bank har uppnåtts. Även om det finns utrymme för förbättring. Se 5.5 Resurs Banks nya testflöde.

- Vilka rutiner kommer behövas?

Rutiner för buggrapportering, test planering och kommunikation. Något som täcks upp bra av denna rapport

- Vilka testmetoder är aktuella idag?

Unit tester skrivs av utvecklare. Manuella tester utförs av testarna genom en test-approach som är stark inspirerad av Exploratory testning.

- Hur är företaget strukturerat idag?

Detta tas upp i kapitlet 5.2 Nuvarande utvecklingsmetodik och besvaras väl.

6.2 Framtida utvecklingsmöjligheter

Det finns en del sätt att förbättra testmetodiken som nu finns på Resurs Bank. Författarna av detta arbete är även övertygade om att mindre detaljer kontinuerligt kommer dyka upp och åtgärdas och hoppas att Resurs Bank fortsätter att arbeta utifrån det jobb som lagts ner på detta examensarbete. Att vidare överföra information från det gamla systemet Lime till det nya Redmine är något som kommer behöva göras i framtiden, även att fortsätta testningen av Butiksservice 2 och senare andra applikationer. Detta har tyvärr fått prioriterats bort på grund av tidsbrist. Att de framtida testarna för Resurs Bank delar erfarenheter, testidéer och deltar i utbildningar och seminarier inom mjukvarutestning är viktigt för att hålla sig uppdaterade och säkerställa kvalitén i testningen.

7 Terminologi

Nischbank	**Ett litet bankaktiebolag som erbjuder ett mer begränsat utbud av tjänster.**
Bankoktroj	Ett företag som har tillstånd av regeringen att bedriva bank- eller fondbörsverksamhet.
Filial	En självständig underavdelning under en annan inrättning av samma slag.
Operativsystem	En samling datorprogram som syftar till att underlätta användandet av en dator.
Plattform	Struktur för databearbetning och funktioner som bildas genom en kombination av maskinvara och programvara och hur information lagras.
Mindmap	Även kallat tankekarta, en teknik för att göra anteckningar, ofta i flera färger, baseras på relationer mellan inlärda element.
Open source	Öppen källkod. avser oftast datorprogram där källkoden är tillgänglig att använda, läsa, modifiera och vidaredistribuera för den som vill.
Plugin	Ett insticksprogram är ett datorprogram som inte körs fristående, utan installeras som en del av ett annat program. Insticksprogrammet tillför det andra programmet ny funktionalitet.
Scrum	Metodik för systemutveckling. Se http://sv.wikipedia.org/wiki/Scrum för mer information.
Deploy	Att flytta en produkt från ett tillfälligt eller utveckling tillstånd till ett permanent eller önskat tillstånd.
Staging server	En server som används som ett tillfälligt stadium för att testa nya

	eller reviderade webbsidor innan de blir publika.
Integration	Sammanställning av sinsemellan olika delar till en komplexare helhet.
Frontend	Beteckna för den bearbetning som sker nära användaren.
Backend	Beteckna för den bearbetning som sker nära basen.
Selenium	Program som används för att skapa och utföra automatiska webbsidtester.
SoapUI	Program som används för att testa webbservice.
SoapUI Suites	En mängd SoapUI tester.
Lime	System för projekthantering.
Confluence	Resurs Banks intranät.
XML	Universellt språk för format på dokument.
XSD	XML-Schema som klargör tillåtna element och attribut för en XML.
Regular Expressions	En notation för att beskriva vissa mängder av strängar. Ett uttryck består av en sträng som följer särskilda syntaxregler.
Assemblages	En samling av datamoduler.
Jenkins	Continuous Integration System.
Sonar	System för testövervakning.
Intranät	Ett privat datornätverk inom ett företag eller en organisation.

8 Referenser

[1.0]
- Resurs Bank http://www.resurs.se/om+ /omoss.jsp. (06-02-2012)

[2.0]
- Michael Sutton, Adam Greene, Pedram Amini (2007). Fuzzing: Brute Force Vulnerability Discovery. ISBN 0321446119.

[2.1]
- Folklore http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt (15/2-2012)

[3.0]
- Mark Utting, Bruno Legeard (2007). "Practical Model-Based Testning: A Tools Approach". ISBN 978-0-12-372501-1.

[3.1]
- Goldpractice http://goldpractice.thedacs.com/practices/mbt/images/image002.gif

[4.0]
- James Bach (2003). "Exploratory Testning Explained". (21/2 - 2012)

[4.1]
- Cem Kaner (2008). "A Tutorial in Exploratory Testning". (21/2 - 2012)

[4.2]
- Cem Kaner, James Bach, Brett Pettichord (2001). "Lessons Learned in Software Testning". ISBN 0-471-08112-4.

[4.3]
- Cem Kaner, Jack Falk, Hung Quoc Nguyen (1999). "Testning Computer Software". ISBN 0471358460.

[4.4]
- Cem Kaner, James Bach (2004). “Exploratory & Risk Based Testning”.

[5.0]
- TestningGeek http://www.testninggeek.com/software-testning-introduction-to-equivalence-partitioning. (13/2-2012)

[6.0]
- TestningGeek http://www.testninggeek.com/boundary-value-analysis. (13/2-2012)

[7.0]
- Scribd http://www.scribd.com/doc/9808382/Introduction-to-API-Testning. (2012-02-20)

[8.0]
- Joan C. Miller, Clifford J. Maloney (1963). "Systematic mistake analysis of digital computer programs". ISSN 0001-0782.

[8.1]

- Glenford J. Myers (2004). "The Art of Software Testning". ISBN 0471469122.

[9.0]

- J. Voas (1997). "Fault Injection for the Masses". ISSN: 0018-9162.

[9.1]

- J. V. Carreira, D. Costa, J. G. Silva (1999). "Fault Injection Spot-Checks Computer System Dependability". ISSN: 0018-9235.

[10.0]

- George Mason University http://cs.gmu.edu/~offutt/rsrch/papers/practical.pdf. (14/2-2012)

[10.1]

- George Mason University http://cs.gmu.edu/~offutt/rsrch/papers/mut00.pdf. (14/2-2012)

[10.2]

- Phyllis G. Frankl, Stewart N. Weiss, Cang Hu (1997). “All-uses versus mutation testning: An experimental comparison of effectiveness”.

[11.0]

- Westfallteam http://www.compaid.com/caiinternet/ezine/westfall-bidirectional.pdf. (20/2 - 2012)

[12.0]

- IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries; IEEE; New York, NY.; 1990.

[13.0]

- André B. Bondi (2000). "Characteristics of scalability and their impact on performance". ISBN 1-58113-195-X.

[14.0]

- Hassan Ahmed, Ken Zhang (2006). "Using Decision Trees to Predict the Certification Result of a Build". ISBN:0-7695-2579-2

[14.1]

- Mariusz A. Fecko, Christopher M. Lott (2002). "Lessons learned from automating tests for an operations support system"

[15.0]

- Cem Kaner, James Bach, Brett Pettichord (2001). "Lessons Learned in Software Testning". ISBN 0-471-08112-4.

[15.1]

- Elfriede Dustin, Jeff Rashka, John Paul (1999). "Automated Software Testning -Introduction, Management, and Performance". ISBN 0201432870.

[15.2]

- "Guidelines for Smoke Testning". (22/2 - 2012)

[15.3]

- Steve McConnell. "How to: Configure and Run Build Verification Tests (BVTs)". (22/2 - 2012)

[16.0]

- Satisfice http://www.satisfice.com/articles/et-article.pdf. (20/2 - 2012)

[17.0]

- Robdavispe http://www.robdavispe.com/free2/software-qa-testning-test-tester-2249.html. (23/2 - 2012)

[18.0]

- TestningGeek http://www.testninggeek.com/installation-testning. (21/2 - 2012)

[19.0]

- TestningGeek http://www.testninggeek.com/accessibility-testning. (21/2 - 2012)

[20.0]

- Erik van Veenendaal. "Standard glossary of terms used in Software Testning". (20/2 - 2012)

[21.0]

- Software Testning Fundamentals http://softwaretestningfundamentals.com/software-testning-levels/. (17/2 - 2012)

[22.0]

- Computer.org "SWEBOK Guide - Chapter 5". (20/2 - 2012)

[23.0]

- Thetesteye http://thetesteye.com/conferences/SWET2Abstracts.pdf.
- http://christintesting.blogspot.se/ (20/3 - 2012)

[24.0]

- Mountain Goat Software http://www.mountaingoatsoftware.com/topics/scrum. (20/3 - 2012)

[25.0]

- Uppsala Universitet, Sven-Olof Nyström http://www.it.uu.se/edu/course/homepage/oopjava/st09/notes/f09-vattenfall.html. (22/2 - 2012)

[26.0]

- Guidelines for conducting and reporting case study research in software engineering http://www.springerlink.com/content/t22r8165q7h31636/ .(22/2- 2012)

[27.0]

DevelopeSense

- http://www.developsense.com/blog/2010/11/context-free-questions-for-testing/ (22/2- 2012)

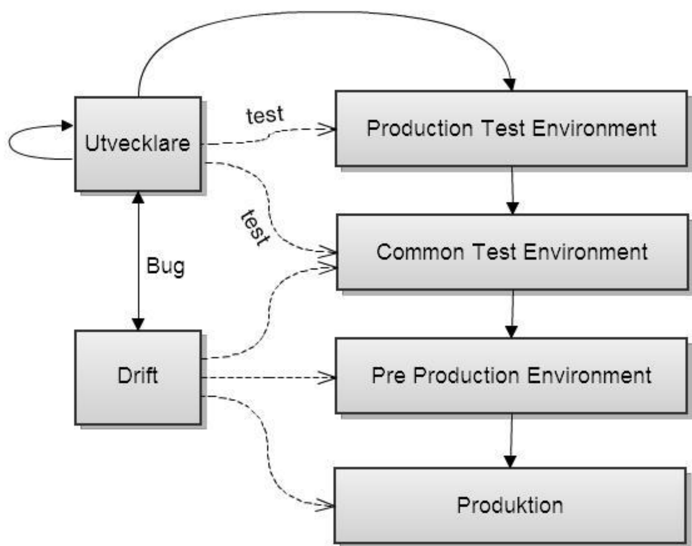
9 Bilagor

9.1 Intervjuer

9.1.1 Frågor kring nuvarande testning med Vlado (31/1-2012)

Denna intervju skedde med Vlado Palczynski, handledare till detta examensarbete på Resurs Bank. Enbart slutna frågor användes. Intervjun skedde på ett strukturerat sätt. [26.0]

- Hur ser er nuvarande utvecklingsmetod/ testmodell ut?
  Vlado förklarar på whiteboard tavlan vilket sedan resulterar i bilden nedan.

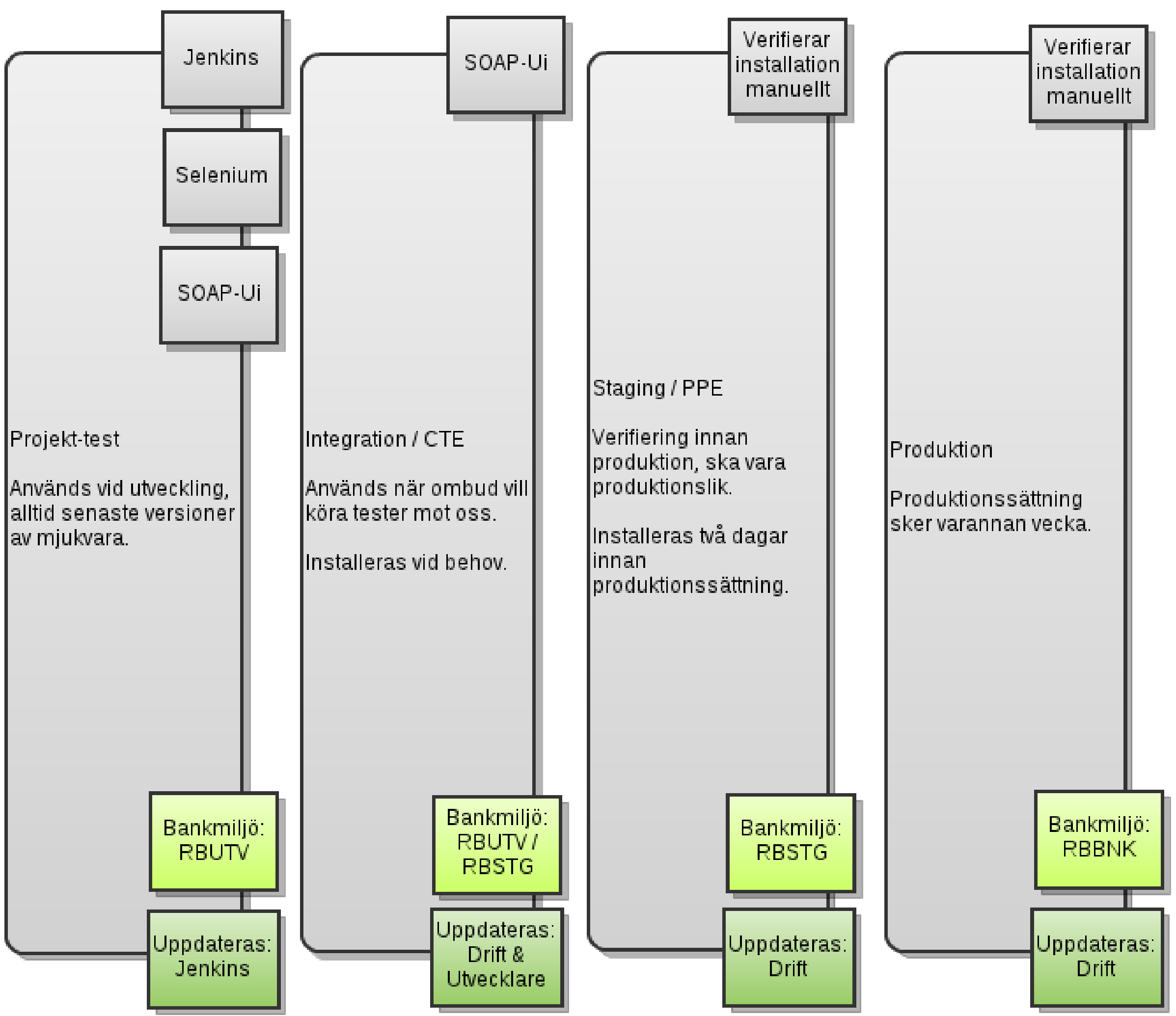


- Hur skall detta arbete levereras?
  – Genom implementering och rapport

- Antalet anställda som detta ska anpassas för?
  – Jag tror vi har 10 utvecklare 5 drift och 2 testare

- Vilka funktioner eftersträvas?
  – Ett bra flöde. En röd tråd, testare måste kunna se vad som ändrats.
  Finns det något program som gör detta?
  – Det vet jag inte

- Vad innefattar de olika lagerna?
  – Jag återkommer med en skiss (resulterade i figuren på nästa sida)



- Är Selenium, SoapUI och Jenkins fasta eller dem kan bytas ut?
  – Dem är fasta, försök istället att peka om dem och anpassa dem för testare.

Denna intervju gjordes för att få en bättre bild över hur Resurs Banks nuvarande utveckling och testning ser ut.

9.1.2 Intervju med Sigurdur (2/4-2012)

Denna intervju skedde med Sigurdur Birgisson, en välkänd testare som arbetar på JayWay och har en lång erfarenhet av testning. Även information utöver frågorna tillkom, sidenotes. En kombination av sluta och öppna frågor användes. Intervjun skedde på ett semi-strukturerat sätt. [26.0]

- Vilka populära testverktyg används idag?
  – Selenium, SoapUI. Ta även en titt på Testmaker.

- Exporteras Selenium testfall från en till flera webbläsare?
  – Ja, men det blir oftast fel. Att scripta testfallen manuellt istället för Record and Play gör att de kan användas på flera olika webbläsare.

- Hur verkar våra metoder? Några att rekommendera? Några att skippa?
  – Vilka metoder som används är beroende på hur projektet man testar ser ut. Efter att ni undersökt applikationen bättre så kommer ni nog se vilka metoder ni kommer att behöva använda.

- Finns det några buggrapporteringstips? Mjukvaror? Resurs Bank använder sig av Redmine.
  – Kolla upp vad som behövs vid buggrapportering, vilken information en utvecklare behöver för att rätta en bugg? Hur ofta buggar skall rapporteras? Vad händer vid stora buggar? Hantering av buggar generellt?

- Testplan?
  – Lägg energi på vad som är viktigt att testa. Egendesignade checklistor kan vara bra. Mindmaps är ett bra hjälpmedel. Var försiktiga med att dokumentera för mycket, då glöms testningen ofta av.

- Dokumentation?
  – Beroende på vad kunden vill ha. Det lönar sig oftast inte att dokumentera testfallen. Skriv om det som man har testat istället.

- Sigurdurs tips om Bloggare
  Michael Bolton
  James Bach
  Gojko Adzic
  Elisabeth Henricsson
  Gregg Pollack (Ruby/Selenium)
  Christin Wiedemann

- Sidenotes

www.thetesteye.com
Software quality Characteristics (.pdf)
Context free questions
Swet2 abstracts (.pdf)
Eurostar keynotes
happytestning.se

Denna intervju gjordes för att få en bättre inblick om hur testning planeras och utförs. Även för att få lite tips angående testning som ska kunna hjälpa att utföra examensarbetet.

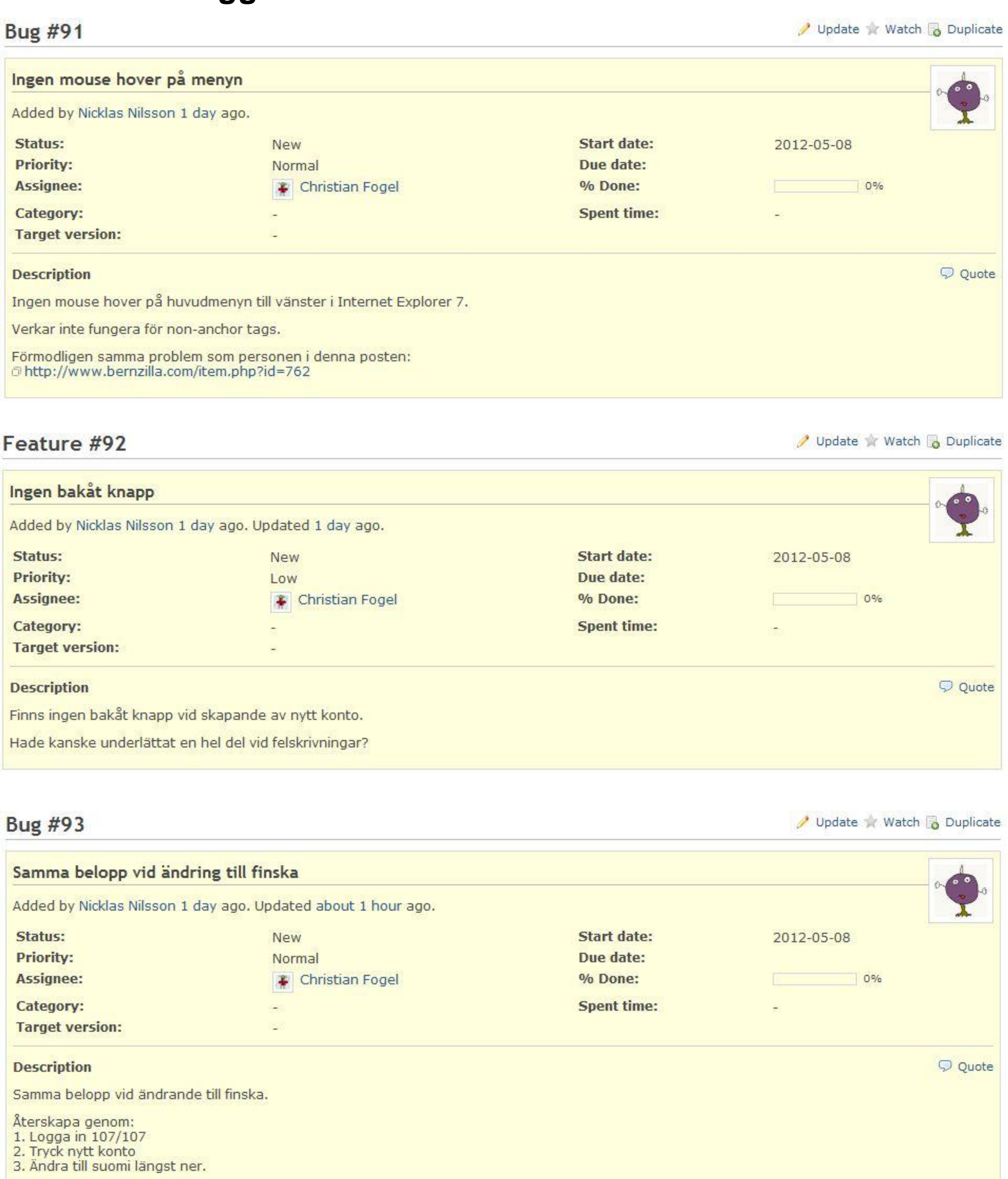
9.2 Context-Free Questions

- Are you my only client?
- What is my mission?
- What else might be part of my mission?
- Do you want a quick, practical, or deep answer to the mission or question you have in mind?
- How long before the end of this testning or development cycle?
- How do you want me to provide them? How often?
- When were you thinking of shipping or deploying this product or service?
- What else do you want me to deliver?
- How do you want me to deliver it?
- This thing I'm testning… could I have it myself, please?
- Are there more than that?
- Is that all there are?
- How is this one expected to be the same or different from the other ones?
- Here's what I believe I see in front of me. What else could it be?
- Here's what I'm thinking right now. What else might be true? What if the opposite were true?
- Could you describe how it works?
- I think I'm seeing a problem. Why do I think it's a problem? For whom might it be a problem?
- What does this thing depend upon?
- What tools or materials were used to construct it?
- Are they easy to talk to? Helpful?
- Is there anyone that I should actively avoid?
- Who's the *best* person to ask about this?
- Who are the local experts in this field?
- Who are the acknowledged experts, even if they don't work here?

- Can I see their results, please?
- Who else is on my test team?
- What skills and competencies are expected of me?
- What other skills and competencies can be found on the test team? Elsewhere?
- What skills and competencies might we be lacking?
- Is there more information available?
- Where could I find more information? Is that the last source you can think of?
- In what other forms could I find information?
- Is that all the information there is? Is there more? Are there more rules? Requirements? Specifications?
- If information is in some way wanting, what can I do to help you discover or develop the information you need?
- What equipment and tools are available to help with my testning?
- What tools would you like me to build? Expect me to build?
- Is there some data that is being processed by this thing?
- Can I have some of that data?
- Can I have a description of the data's structures?
- What are your feelings about this thing?
- Who might feel differently?
- How might they feel?
- What do customers say about it?
- (How do I feel about this thing?)
- Who can we trust? Is there anyone that we should distrust?
- Is there anything that you would like to prohibit me explicitly from doing?
- Are there any other questions I should be asking you?

[27.0]

9.3 Funna buggar



Bug #91

Update Watch Duplicate

Ingen mouse hover på menyn

Added by Nicklas Nilsson 1 day ago.

Status:	New	**Start date:**	2012-05-08
Priority:	Normal	**Due date:**	
Assignee:	Christian Fogel	**% Done:**	0%
Category:	-	**Spent time:**	-
Target version:	-		

Description Quote

Ingen mouse hover på huvudmenyn till vänster i Internet Explorer 7.

Verkar inte fungera för non-anchor tags.

Förmodligen samma problem som personen i denna posten:
http://www.bernzilla.com/item.php?id=762

Feature #92

Update Watch Duplicate

Ingen bakåt knapp

Added by Nicklas Nilsson 1 day ago. Updated 1 day ago.

Status:	New	**Start date:**	2012-05-08
Priority:	Low	**Due date:**	
Assignee:	Christian Fogel	**% Done:**	0%
Category:	-	**Spent time:**	-
Target version:	-		

Description Quote

Finns ingen bakåt knapp vid skapande av nytt konto.

Hade kanske underlättat en hel del vid felskrivningar?

Bug #93

Update Watch Duplicate

Samma belopp vid ändring till finska

Added by Nicklas Nilsson 1 day ago. Updated about 1 hour ago.

Status:	New	**Start date:**	2012-05-08
Priority:	Normal	**Due date:**	
Assignee:	Christian Fogel	**% Done:**	0%
Category:	-	**Spent time:**	-
Target version:	-		

Description Quote

Samma belopp vid ändrande till finska.

Återskapa genom:
1. Logga in 107/107
2. Tryck nytt konto
3. Ändra till suomi längst ner.

Rubriker i formuläret ändras till finska med belopp kvarstår trots att Finland använder Euro.

Bug #94

Update Watch Duplicate

Text till knapp hoppar upp

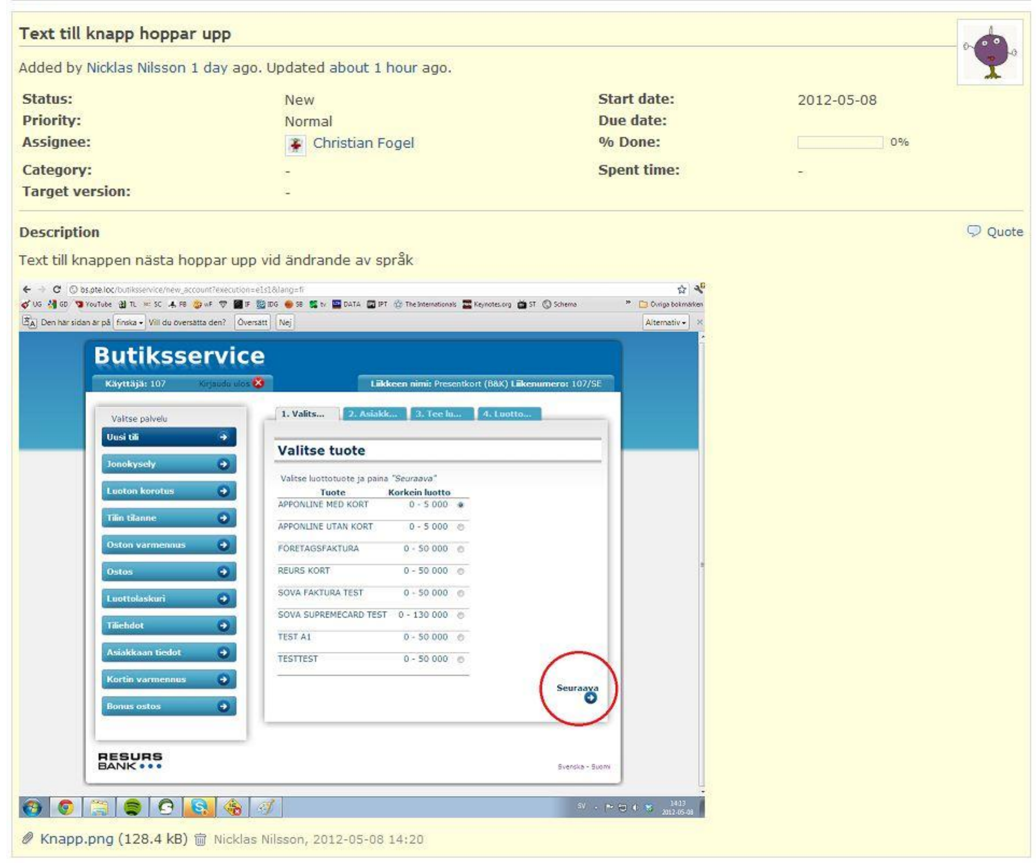
Added by Nicklas Nilsson 1 day ago. Updated about 1 hour ago.

Status:	New	**Start date:**	2012-05-08
Priority:	Normal	**Due date:**	
Assignee:	Christian Fogel	**% Done:**	0%
Category:	-	**Spent time:**	-
Target version:	-		

Description

Quote

Text till knappen nästa hoppar upp vid ändrande av språk



Knapp.png (128.4 kB) Nicklas Nilsson, 2012-05-08 14:20

Bug #95

Update Watch Duplicate

Beskrivningstexten i Internet Explorer 7

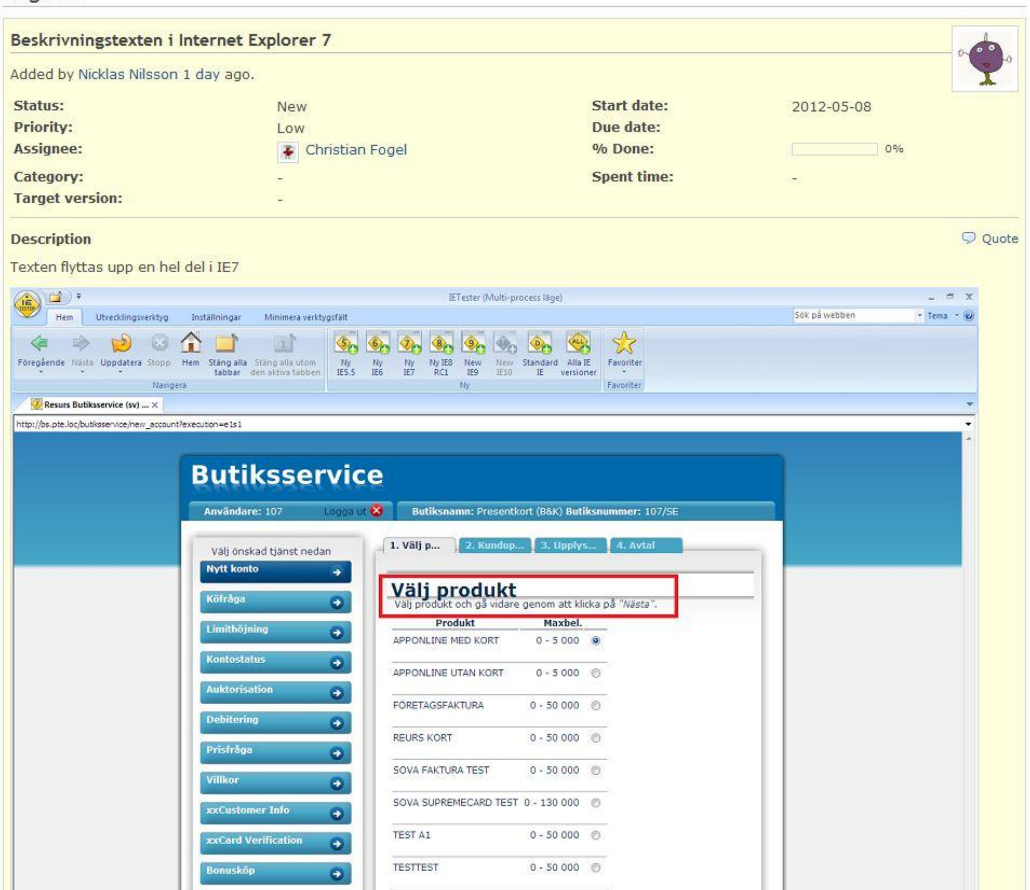
Added by Nicklas Nilsson 1 day ago.

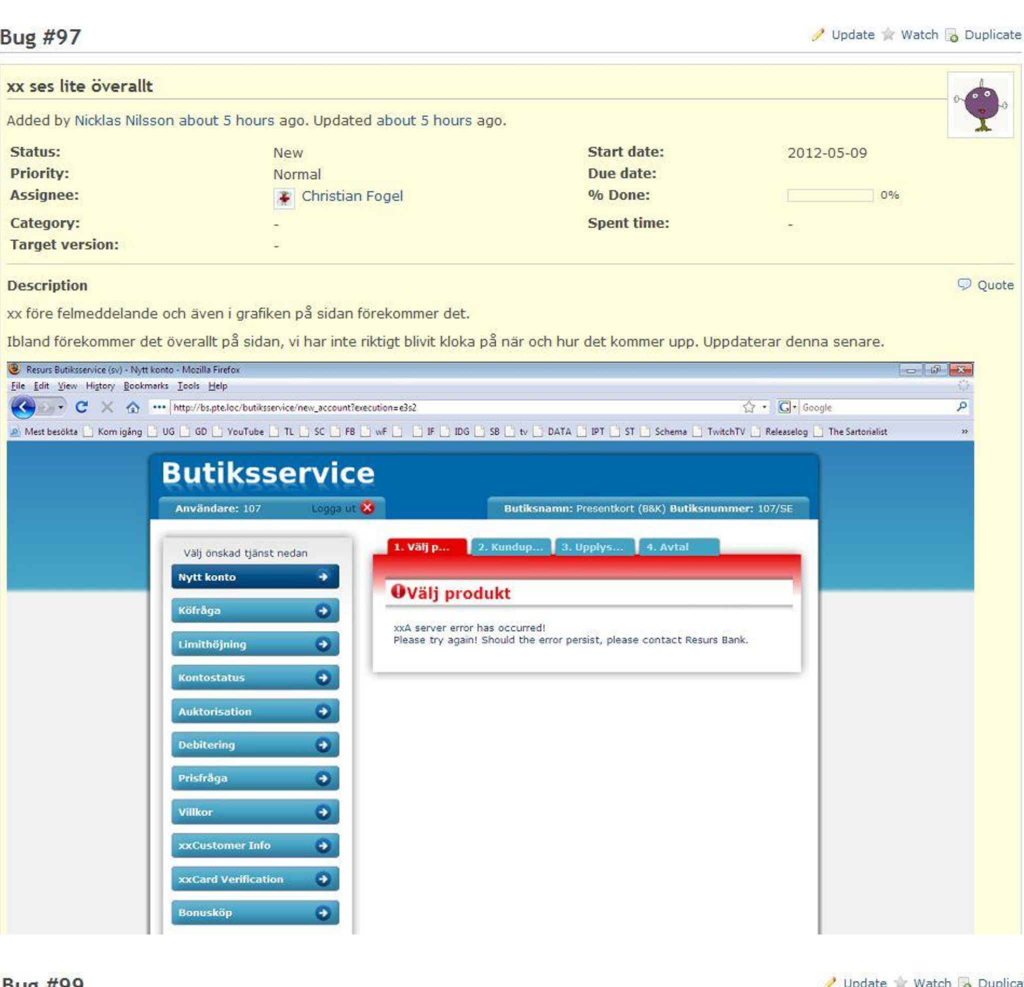
Status:	New	**Start date:**	2012-05-08
Priority:	Low	**Due date:**	
Assignee:	Christian Fogel	**% Done:**	0%
Category:	-	**Spent time:**	-
Target version:	-		

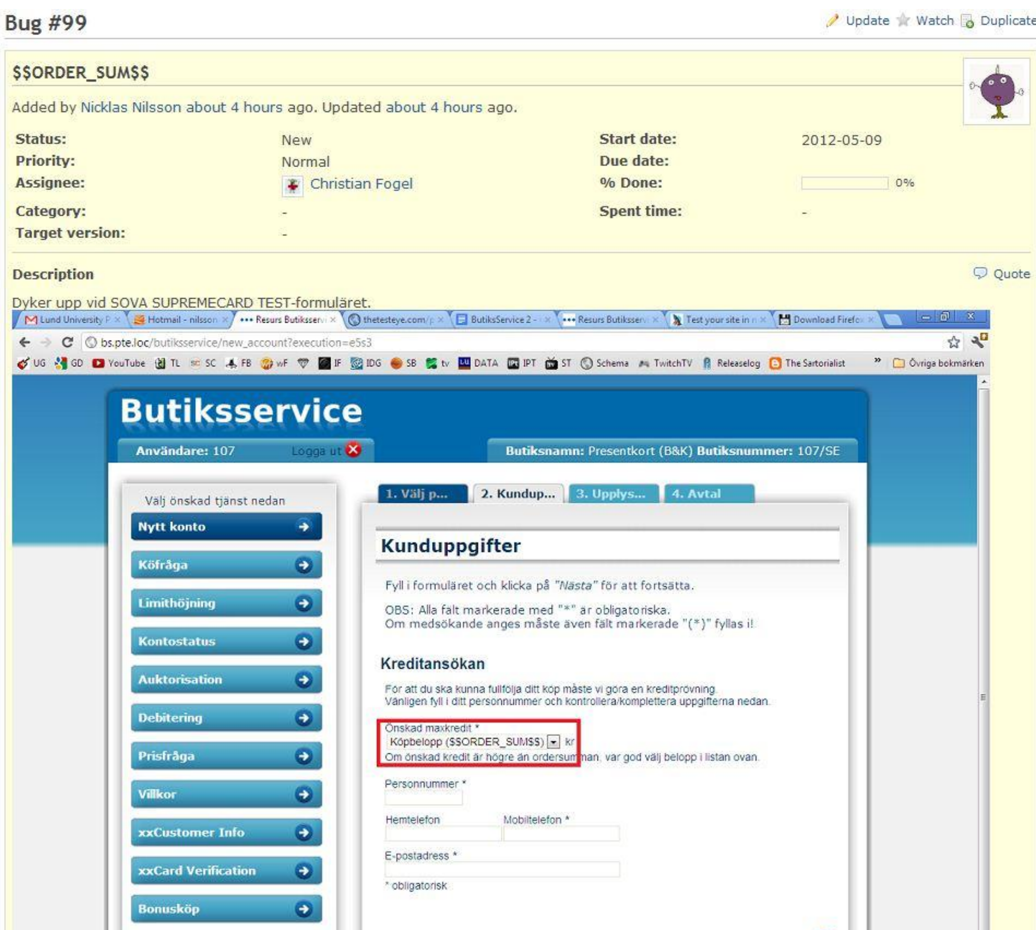
Description

Quote

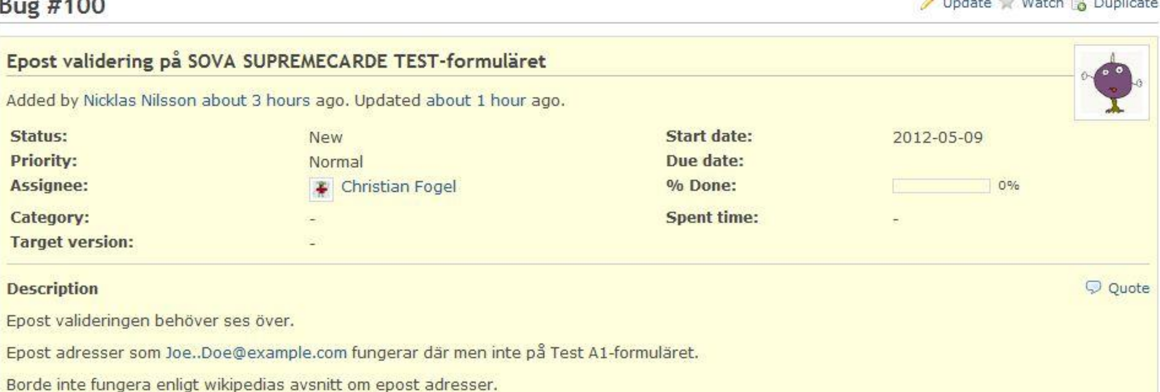
Texten flyttas upp en hel del i IE7




Bug #97
Update Watch Duplicate
xx ses lite överallt
Added by Nicklas Nilsson about 5 hours ago. Updated about 5 hours ago.
Status: New
Priority: Normal
Assignee: Christian Fogel
Category: -
Target version: -
Start date: 2012-05-09
Due date:
% Done: 0%
Spent time: -
Description
Quote
xx före felmeddelande och även i grafiken på sidan förekommer det.
Ibland förekommer det överallt på sidan, vi har inte riktigt blivit kloka på när och hur det kommer upp. Uppdaterar denna senare.
Butiksservice
Användare: 107
Logga ut
Butiksnamn: Presentkort (B&K) Butiksnummer: 107/SE
Välj önskad tjänst nedan
Nytt konto
Köfråga
Limithöjning
Kontostatus
Auktorisation
Debitering
Prisfråga
Villkor
xxCustomer Info
xxCard Verification
Bonusköp
1. Välj p...
2. Kundup...
3. Upplys...
4. Avtal
Välj produkt
xxA server error has occurred!
Please try again! Should the error persist, please contact Resurs Bank.


Bug #99
Update Watch Duplicate
$$ORDER_SUM$$
Added by Nicklas Nilsson about 4 hours ago. Updated about 4 hours ago.
Status: New
Priority: Normal
Assignee: Christian Fogel
Category: -
Target version: -
Start date: 2012-05-09
Due date:
% Done: 0%
Spent time: -
Description
Quote
Dyker upp vid SOVA SUPREMECARD TEST-formuläret.
Butiksservice
Användare: 107
Logga ut
Butiksnamn: Presentkort (B&K) Butiksnummer: 107/SE
Välj önskad tjänst nedan
Nytt konto
Köfråga
Limithöjning
Kontostatus
Auktorisation
Debitering
Prisfråga
Villkor
xxCustomer Info
xxCard Verification
Bonusköp
1. Välj p...
2. Kundup...
3. Upplys...
4. Avtal
Kunduppgifter
Fyll i formuläret och klicka på "Nästa" för att fortsätta.
OBS: Alla fält markerade med "*" är obligatoriska.
Om medsökande anges måste även fält markerade "(*)" fyllas i!
Kreditansökan
För att du ska kunna fullfölja ditt köp måste vi göra en kreditprövning.
Vänligen fyll i ditt personnummer och kontrollera/komplettera uppgifterna nedan.
Önskad maxkredit *
Köpbelopp ($ORDER_SUM$) kr
Om önskad kredit är högre än ordersumman, var god välj belopp i listan ovan.
Personnummer *
Hemtelefon
Mobiltelefon *
E-postadress *
* obligatorisk

Bug #100



Update Watch Duplicate

Epost validering på SOVA SUPREMECARDE TEST-formuläret

Added by Nicklas Nilsson about 3 hours ago. Updated about 1 hour ago.

Status:	New	**Start date:**	2012-05-09
Priority:	Normal	**Due date:**	
Assignee:	Christian Fogel	**% Done:**	0%
Category:	-	**Spent time:**	-
Target version:	-		

Description Quote

Epost valideringen behöver ses över.

Epost adresser som Joe..Doe@example.com fungerar där men inte på Test A1-formuläret.

Borde inte fungera enligt wikipedias avsnitt om epost adresser.

Bug #101

Update Watch Duplicate

Epost validering TEST A1-formuläret

Added by Nicklas Nilsson about 2 hours ago. Updated about 1 hour ago.

Status:	New	**Start date:**	2012-05-09
Priority:	Normal	**Due date:**	
Assignee:	Christian Fogel	**% Done:**	0%
Category:	-	**Spent time:**	-
Target version:	-		

Description Quote

Om vanlig epost standard följs så enligt den på wikipedia så borde adresser som simplewith+symbol@example.com accepteras men det gör den inte.

Finns fler exempel.

Bug #102

Update Watch Duplicate

Säljarkod accepterar mer än 20 tecken

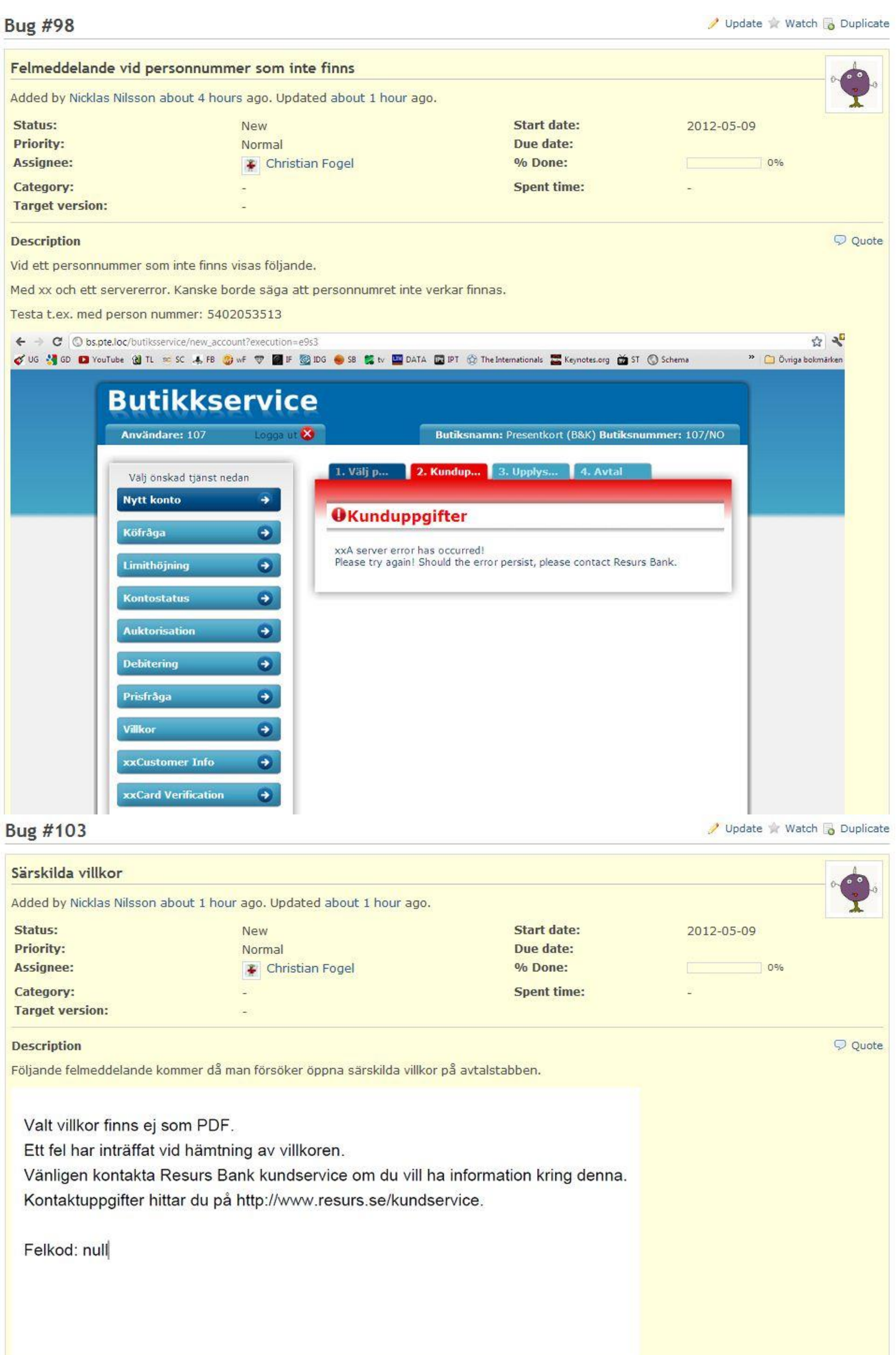
Added by Nicklas Nilsson about 2 hours ago. Updated about 1 hour ago.

Status:	New	**Start date:**	2012-05-09
Priority:	Normal	**Due date:**	
Assignee:	Christian Fogel	**% Done:**	0%
Category:	-	**Spent time:**	-
Target version:	-		

Description Quote

på test A1-formuläret.

Skall det kunna ske enligt http://confluence/display/it/Butiksservice+2+-+Kund-FAQ



Bug #98

Update Watch Duplicate

Felmeddelande vid personnummer som inte finns

Added by Nicklas Nilsson about 4 hours ago. Updated about 1 hour ago.

Status:	New	**Start date:**	2012-05-09
Priority:	Normal	**Due date:**	
Assignee:	Christian Fogel	**% Done:**	0%
Category:	-	**Spent time:**	-
Target version:	-		

Description

Quote

Vid ett personnummer som inte finns visas följande.

Med xx och ett servererror. Kanske borde säga att personnumret inte verkar finnas.

Testa t.ex. med person nummer: 5402053513

Bug #103

Update Watch Duplicate

Särskilda villkor

Added by Nicklas Nilsson about 1 hour ago. Updated about 1 hour ago.

Status:	New	**Start date:**	2012-05-09
Priority:	Normal	**Due date:**	
Assignee:	Christian Fogel	**% Done:**	0%
Category:	-	**Spent time:**	-
Target version:	-		

Description

Quote

Följande felmeddelande kommer då man försöker öppna särskilda villkor på avtalstabben.

Valt villkor finns ej som PDF.
Ett fel har inträffat vid hämtning av villkoren.
Vänligen kontakta Resurs Bank kundservice om du vill ha information kring denna.
Kontaktuppgifter hittar du på http://www.resurs.se/kundservice.

Felkod: null

SärskilldaVillkorFinnsEjSomPDF.png (44.2 kB) Nicklas Nilsson, 2012-05-09 13:27