Göteborgs Universitet
Institutionen för informatik

Vilka principer bör styra utveckling av komponenter för återanvändning?

Öyvind Larsen

Abstrakt
Syftet med uppsatsen var att ta fram ett antal principer som enligt litteraturen bör styra utveckling av komponenter för återanvändning samt att validera principerna genom en undersökning i praxis. En litteraturstudie genomfördes för att hitta principer. En modell har skapats för att visa att principerna väl täcker alla viktiga områden som berör återanvändning av komponenter. Datainsamling gjordes med hjälp av en enkätundersökning på Ericsson Microwave Systems och dataanalys gjordes med hjälp av deskriptiv statistik i form av diagram. Uppsatsen slutar med att konstatera att ingen av principerna kunde förkastas genom enkätundersökningen, utan de förtjänar vidare forskning.

Handledare:
Informatik Thanos Magoulas
Ericsson Microwave Systems Björn Karlsson

Magisteruppsats 20 p
Vårterminen 2002

FÖRORD

Ett stort tack till handledarna Thanos Magoulas och Björn Karlsson.

Denna uppsats har jag gjort vid sidan av ett examensarbete som jag och David Sjöqvist gjort på divisionen för ytradar på Ericsson Microwave Systems (EMW) i Mölndal. Examensarbetet är dokumenterat i en rapport, Test Environment Simulation Tool, som är inlämnad på institutionen för Datavetenskap, Chalmers Tekniska Högskola/Göteborgs Universitet.

INNEHÅLLSFÖRTECKNING

1 INLEDNING 5

1.1 Bakgrund 5

1.2 Frågeställning 6

1.3 Syfte 6

1.4 Avgränsningar 6

2 TEORETISK REFERENSRAM 7

2.1 Komponentbaserad utveckling 7
2.1.1 Vad en komponent är 7
2.1.2 Komponentbaserad arkitektur 9

2.2 Återanvändning 10
2.2.1 Vad återanvändning är 10
2.2.2 Fördelar med återanvändning 11
2.2.3 Nackdelar med återanvändning 11
2.2.4 Typer av återanvändning av kod 12

2.3 Principer för utveckling av komponenter för återanvändning 13
2.3.1 Inledning 13
2.3.2 Återanvändningsanpassa systemutvecklingsmetoden 14
2.3.3 Basera återanvändning på produktfamiljer 15
2.3.4 Val av komponent för återanvändning 16
2.3.5 Använd objektorienterad systemutveckling 19
2.3.6 Bygg upp ett bibliotek av återanvändningsbara komponenter 19
2.3.7 Dokumentera komponenterna och återanvänd dokumentationen 20
2.3.8 Certifiera komponenterna 21
2.3.9 Låt komponenterna finansiera sig själva 22
2.3.10 Versions- och konfigurationshantering är nödvändig 22

2.4 Sammanfattning av teoretisk utredning 23
2.4.1 En modell för vägledning av komponentbaserad systemutveckling 23

3 METOD ... 26

3.1 Litteraturstudie ... 26

3.2 Enkätundersökning ... 26
3.2.1 Planering och genomförande av undersökningen ... 26
3.2.2 Val av undersökningsobjekt ... 26
3.2.3 Beskrivning av verksamheten på EMW ... 27
3.2.4 Arbete med återanvändning på EMW ... 28

4 RESULTAT ... 29

4.1 Separat redovisning av svaren för varje fråga ... 29
4.1.1 Återanvändningsanpassa systemutvecklingsmetoden ... 29
4.1.2 Basera återanvändning på produktfamiljer ... 30
4.1.3 Val av komponent för återanvändning ... 30
4.1.4 Använd objektorienterad systemutveckling ... 31
4.1.5 Bygg upp ett bibliotek av återanvändningsbara komponenter 31
4.1.6 Dokumentera komponenterna och återanvänd dokumentationen ... 32
4.1.7 Certifiera komponenterna ... 32
4.1.8 Låt komponenterna finansiera sig själva ... 33
4.1.9 Versions- och konfigurationshantering är nödvändig ... 33
4.1.10 Egna förslag på kandidatprinciper ... 34

4.2 Sammanfattning av svaren från hela enkätundersökningen ... 34
4.2.1 Översiktligt resultat ... 34
4.2.2 Medelvärdet för svaren på respektive fråga ... 35

5 DISKUSSION ... 36

5.1 Studien ... 36

5.2 Resultatet ... 36

5.3 Självkritik och framtida forskning ... 37

6 SLUTSATSER ... 38

7 REFERENSER ... 39

1 INLEDNING

1.1 Bakgrund

Det är uppenbarligen svårt att utveckla IT-baserade system. Att leverera ett system som uppfyller kundens krav, som håller sig inom kostnadsramarna och att dessutom leverera det i tid är väl i dagsläget närmast att betrakta som önsketänkande. ”Software crisis” är ett begrepp som kännetecknar detta fenomen som likt en förbannelse vilar över all systemutveckling sedan årtionden. För att lösa problemet och bli bättre på att utveckla system har metoderna hela tiden förbättrats. Omkring 1970 kom funktionsorienterade metoder och runt 1980 kom dataorienterade metoder inspirerade av databasdesign (Mathiassen, & Munk-Madsen, & Nielsen, & Stage, 1998). Slutligen kom objektorienterade metoder och programmeringsspråk.

Ett av de stora löftena som objektorientering gav, var att nu skulle man lätt kunna återanvända kod och slippa att ”uppfinna hjulet på nytt”. En klass skulle kunna användas, utan modifikation, i ett nytt projekt. Löftet har tyvärr inte infriats. I praktiken har det visat sig att klasser är alldeles för unika för varje projekt, vilket innebär att de utgör för små enheter för att kunna återanvändas (Sommerville, 2001). Komponenter är däremot rätt nivå för återanvändning.

I nästan alla verksamheter förekommer återanvändning av komponenter.
Husarkitekter sitter inte och ritar i detalj hur exempelvis ett fönster skall utformas, utan använder färdiga komponenter i sin konstruktion. När en ny bilmodell tas fram återanvänds motorer och mängder av andra komponenter. Likadant ser det alltså ut i nästan allt konstruktionsarbete, utom vad gäller utveckling av IT-baserade system.

Det är nu dags även för systemutvecklare att sluta slösa bort tid och pengar i varje systemutvecklingsprojekt för att istället börja återanvända komponenter. Ett exempel på vad lyckad återanvändning innebär ges i Software Reuse (Coulange, 1998). Utvecklingstiden för ett projekt var 15 289 timmar, om man inte återanvänt kod hade det tagit 40 352 timmar. Man hittade och rättade 184 fel, utan återanvändning hade man varit tvungen att hitta och rätta 459 fel. Lyckad återanvändning ger alltså snabbare, billigare och bättre programvara. Därför kan återanvändning av kod vara lösningen på ”software crisis”-problemet.

1.2 Frågeställning

Vilka principer bör styra utveckling av komponenter för återanvändning?

1.3 Syfte

Syftet med uppsatsen är att besvara frågeställningen genom att ta fram ett antal principer som enligt litteraturen bör styra utveckling av komponenter för återanvändning samt att validera principerna genom en undersökning i praxis.

1.4 Avgränsningar

Uppsatsen behandlar utveckling av programvarukomponenter, inte inköpta komponenter. Inte heller utveckling av komponenter avsedda att säljas. I framtiden kommer troligen marknadsplatser för handel med komponenter uppstå. Men de svårigheter detta medför kommer alltså inte behandlas i uppsatsen. Exempelvis kan nämnas problemet när en inköpt komponent behöver uppdateras och man själv inte har kontrollen över källkoden.

Istället för att återanvända kod så kan konceptuella beskrivningar av problem och dess lösning återanvändas. Dessa beskrivningar kallas på engelska design patterns. Design patterns kan användas i systemutvecklingsprocessen både vid framtagning av projektspecifika och återanvändningsbara komponenter. Design patterns, som introducerades i Design Patterns: Elements of Reusable Object-oriented Software (Gamma, & Helm, & Johnson, & Vlissides, 1994), kommer inte behandlas mer i denna uppsats.

2 TEORETISK REFERENSRAM

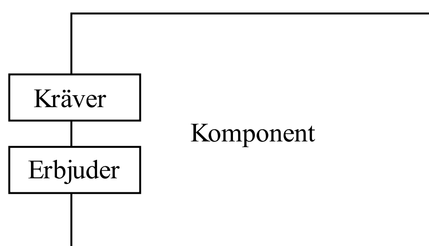
2.1 Komponentbaserad utveckling

2.1.1 Vad en komponent är

Komponenter är abstraktare än klasser och kan ses som självständiga tjänsteleverantörer. När ett system behöver en tjänst så anropar det en komponent som levererar tjänsten. Detta kan ske utan att systemet behöver veta var den anropade komponenten exekveras eller vilket programmeringsspråk komponenten är skriven i. För att det skall kunna fungera krävs en arkitektur som hanterar kommunikationen mellan komponenterna.

En komponent är en exekverbar(binär) enhet med endast explicita beroenden av omgivningen och den har inget bestående tillstånd (Szyperski, 2000).

För att en komponent skall vara oberoende måste den separeras från omgivningen och andra komponenter. Därför kapslar komponenten in sin implementation och interagerar med omgivningen genom ett gränssnitt. Komponenter definieras därmed av dess gränssnitt. Ofta har en komponent ett in-gränssnitt som specificerar vad komponenten kräver av systemet för att fungera samt ett ut-gränssnitt som specificerar de tjänster komponenten erbjuder. Gränssnitten måste vara standardiserade så olika komponenter kan interagera med varandra. Nedan visas en bild av en komponent.



Figur 1: *En komponent och dess gränssnitt.*

En komponent skall i största möjliga mån vara oberoende, men explicita beroenden av omgivningen är ofrånkomliga. En specifikation av komponentens beroenden skall innehålla dels gränssnittet som anger vad komponenten kräver av omgivningen, exempelvis kan komponenten kräva tillgång till filsystemet, dels vilken komponentarkitektur (se avsnitt 2.1.2) komponenten utvecklats för.

Komponenten skall inte ha inbyggd kunskap om typen eller identiteten hos användaren av dess funktionalitet. Kunskapen en komponent har om andra komponenter den använder skall begränsas till typen och gränssnittet komponenterna har. Andra komponenters identitet skall inte vara känd för en komponent förrän realiseringen av systemet som den är en del av.

Att komponenten saknar bestående tillstånd kan jämföras med klasser. Klasser är statiska beskrivningar av objekt och saknar tillstånd medan objekt är dynamiska och har bestående tillstånd. Det är väldigt viktigt att särskilja de statiska komponenterna från de dynamiska objekten. Dynamiska komponenter leder till problem med underhåll, konfigurering och versionshantering.

Tabellen nedan visar skillnader mellan objekt och komponent.

Egenskap	**Objekt**	**Komponent**
Instansieras (har identitet)	Ja	Nej
Tillstånd	Varaktigt	Flyktigt
Inkapsling	Tillstånd och beteende	Implementation

Tabell 1: *Objekt- och komponentegenskaper.*

Eftersom ett objekt har ett unikt tillstånd så behövs en unik identitet, så objektet kan identifieras trots tillståndsförändringar. En komponent utan varaktigt tillstånd kan inte skiljas från kopior av sig själv.

Att skilja mellan objekt och komponenter handlar alltså om att skilja mellan statiska egenskaper som gäller för en viss konfiguration, och dynamiska egenskaper för något datascenario. Om man vill ha ett objekt som komponent får man alltså ha en klass i binär form som man via ett gränssnitt anropar en metod som skapar ett objekt och returnerar en referens till objektet. Komponenten består då inte av objektet utan av klassen.

2.1.2 Komponentbaserad arkitektur

En komponentbaserad arkitektur är en specifikation av ett systems komponenter och kommunikationen mellan dem. För närvarande finns tre stora komponentarkitekturer, Object Management Groups CORBA, Suns Java och Microsofts COM.

Eftersom det redan skrivits så mycket om dessa arkitekturer har jag valt att ta en liten titt på Microsofts .NET plattform, .NET Framework, eftersom den nyligen har introducerats.

.NET Framework är en komponentarkitektur för flera språk som möjliggör samverkan mellan de olika programspråken och de olika språken har även gemensam felhantering. Förutom Microsofts egna språk, exempelvis Visual Basic och C#, finns stöd för bland andra COBOL, Eiffel, Fortran, Perl och Python.[1]

NET Framework består av ett omfattande klassbibliotek som är gemensamt för de olika språken och en virtuell maskin, Common Language Runtime (CLR), liknande javas virtuella maskin och Common Language Specification som beskriver hur ett språks kompilator skall generera Microsoft Intermediate Language (MSIL). De olika programmeringsspråken kompileras till MSIL och körs av CLR. Därför kan komponenter skrivna i olika programmeringsspråk användas för att bygga ett system.

Komponenter i .NET Framework kan kommunicera med varandra med hjälp av metadata (Nandu, 2001). Metadata är en egenskap som låter CLR känna till detaljer om en komponent. Metadata sparas vid kompilering och kan förfrågas vid körning. Genom en process som kallas reflection, som finns i klassen System.Reflection, kan en applikation undersöka en komponents metadata för att se vilka tjänster komponenten erbjuder. Metadata lagras i binärt format i komponenten. En komponent innehåller även information om dess version.

Det händer mycket inom komponentarkitekturområdet och det finns etablerade komponentarkitekturer. När man bygger system med komponenter måste man kunna sammankoppla dem, så man måste ha klart för sig vilken arkitektur man utvecklar sina komponenter för.

---

[1] Se http://msdn.microsoft.com/vstudio/partners/language/default.asp för en fullständig lista över de programmeringsspråk som stöds.

2.2 Återanvändning

2.2.1 Vad återanvändning är

Återanvändning är processen att utveckla nya mjukvarusystem från komponenter som är utvecklade för återanvändning. Troligen kan inte ett system byggas enbart med återanvändningsbara komponenter. Då får man även utveckla projekt-specifika komponenter. Allt som kan återanvändas utgör programvarutillgångar.

Följande kan vara programvarutillgångar:

1) Exekverbara program och programvaruverktyg.
2) Källkod.
3) Specifikationer för krav, design och arkitekturer.
4) Testdata och testspecifikationer.
5) Kunskap och information.

Det är alltså inte bara kod som kan återanvändas. Det som i denna uppsats avses med återanvändning är återanvändning av komponenter. För att kunna återanvända en komponent behöver man ha tillgång till krav- och designspecifikationer så man vet vad komponenten erbjuder och om den är lämplig att använda i det aktuella projektet. Så man kan se en återanvändningsbar komponent som bestående av komponenten i binär form (exekverbar fil) med källkod, kravspecifikation, designspecifikation och testdokumentation med testfall.

Återanvändning kan ses utifrån två perspektiv. Konsumentperspektivet innebär att system utvecklas med hjälp av återanvändningsbara komponenter. Utvecklarna är då konsumenter. Producentperspektivet innebär att komponenter utvecklas för att senare återanvändas. Utvecklarna producerar komponenter som görs tillgängliga för andra utvecklare att konsumera. En utvecklare kan i ett projekt vara både producent och konsument.

2.2.2 Fördelar med återanvändning

Att utnyttja de möjligheter som återanvändning erbjuder ger betydande förbättringar av programvaruutvecklingens produktivitet, kvalitet och kostnad.
De största fördelarna återanvändning ger är enligt Sommerville (2001):

- Ökar utvecklarnas produktivitet, då fler kodrader produceras per tidsenhet, eftersom de återanvända komponenterna redan är klara. Därmed krävs färre utvecklare.
- Ökar programvarans kvalitet genom att tillförlitligheten ökar. Programvarukomponenten återanvänds i många projekt och ju mer koden återanvänds, desto mer genomtestad blir den.
- Minskar utvecklingstiden och därmed leveranstiden.
- Minskar kostnaden för programvaruutveckling.
- Förenklar förvaltning och förändring av utvecklingsprojekt eftersom personal, verktyg och metoder lättare kan överföras mellan projekt.
- Effektiv användning av specialister som kan kapsla in sin kunskap i återanvändningsbara komponenter.
- Standarder kan upprätthållas genom att de implementeras i återanvändningsbara komponenter. Det kan gälla exempelvis standardiserade användargränssnitt.

2.2.3 Nackdelar med återanvändning

De största nackdelarna återanvändning ger är enligt Sommerville (2001):

- Brist på verktyg. CASE-verktyg stöder inte utveckling med åter-användning.
- Underhåll av ett komponentbibliotek. Att bygga upp ett komponent-bibliotek kan vara svårt. Det ställer krav på klassificering och hämtning av komponenter.
- Det räcker inte att det finns färdiga komponenter. Utvecklarna måste kunna hitta och förstå komponenterna i biblioteket för att kunna återanvända dem.

2.2.4 Typer av återanvändning av kod

Kopiering, funktionsåteranvändning och komponentåteranvändning är tre sätt att återanvända kod (Sommerville, 2001).

Kopiering
Intressanta delar kopieras in i en ny komponent. Det kan röra sig om allt ifrån enskilda källkodsrader till hela källkodsfiler. Detta är väldigt vanligt. Man ser en bra lösning och kopierar den. Men denna typ av kopiering är inte vad som avses med återanvändning av komponenter.

Funktionsåteranvändning
Återanvändning av funktioner, metoder och procedurer som ingår i de standardbibliotek som hör till de olika programmeringsspråken. De kan även ingå i företagsspecifika klassbibliotek. Exempelvis finns Open i Ada.Text_IO för att öppna en fil. Standardbiblioteksfunktioner används flitigt, det inser nog de flesta att det bara är dumt att skriva egna.

Komponentåteranvändning
Återanvändning av komponenter som kan variera i storlek från enskilda klasser upp till delsystem. Detta är den ”riktiga”, systematiska formen av återanvändning.

2.3 Principer för utveckling av komponenter för återanvändning

2.3.1 Inledning

Det finns ett samband mellan principer och praxis där principer påverkar de praktiska arbetssätten som i sin tur kan leda till nya principer. Principer organiserar, motiverar, förklarar och validerar de praktiska arbetssätten. I denna uppsats skall principer för utveckling av komponenter för återanvändning tas fram. Avsikten är att principerna skall vara till nytta vid utveckling av komponenter för återanvändning.

Jag kommer därför ta upp ett antal kandidater till principer för utveckling av komponenter för återanvändning.

Följande kriterier, hämtade från artikeln Fundamental principles of software engineering – a jorney (Bourke, & Dupuis, & Abran, & Moore, & Tripp, & Wolff, 2001), har använts för att identifiera kandidatprinciper:

- Grundläggande principer är mindre specifika än metoder och tekniker, så man kan välja en specifik metod eller teknik för att genomföra principen. En grundläggande princip bör ändå vara tillräckligt precis för att kunna användas som stöd.

- Grundläggande principer bör formuleras så att de är gällande längre tid än metoder och tekniker.

- Vanligen upptäcks grundläggande principer i praktiken.

- Grundläggande principer skall inte prioritera eller välja mellan olika lösningars kvalitet. Grundläggande principer skall identifiera och förklara de olika lösningarna medan utvecklingsprocessen skall göra avvägningen mellan de olika lösningarna.

Kandidatprinciperna skall sedan genom en undersökning valideras i praxis. Undersökningen kan även ge upphov till nya principer. Undersökningen kommer redovisas i resultatavsnittet och tolkning av resultatet kommer ske i diskussionsavsnittet.

Nu följer i avsnitt 2.3.2 till 2.3.10 kandidatprinciperna.

2.3.2 Återanvändningsanpassa systemutvecklingsmetoden

Traditionella systemutvecklingsmetoder stöder inte återanvändning. För att nå bästa resultat med återanvändningsarbetet bör systemutvecklingsmetoden anpassas för återanvändning.

Återanvändning kan enligt Sommerville (2001) integreras i systemutvecklingsmetoden på två sätt.

1. Designa systemet först och leta sedan efter återanvändningsbara komponenter.
Metoden kan se ut så här:

- Design av systemarkitektur
- Specificera komponenter
- Sök efter återanvändningsbara komponenter
- Integrera de funna komponenterna

2. Sök först efter återanvändningsbara komponenter och basera designen på de tillgängliga komponenterna.
Metoden kan se ut så här:

- Specificera systemkraven
- Sök efter återanvändningsbara komponenter
- Modifiera kraven i enlighet med de funna komponenterna
- Arkitekturdesign
- Sök efter återanvändningsbara komponenter
- Designa systemet med de återanvändningsbara komponenterna.

I det första fallet kommer man först fram till vilka komponenter som behövs och tittar sedan efter vilka komponenter som kan återanvändas. De komponenter som saknas nykonstrueras.

I det andra fallet tittar man först vilka återanvändningsbara komponenter som finns och baserar designen på det. Det kallas återanvändningsdriven design. De komponenter som saknas nykonstrueras.

Coulange (1998) anser att en traditionell systemutvecklingsmetod bestående av faserna kravanalys, generell design, detaljerad design, kodning, enhetstest, integration och validering bör byggas ut med ytterligare två faser, anskaffning och arkivering. I anskaffningsfasen identifieras återanvändningsbara komponenter och i arkiveringsfasen lagras nykonstruerade komponenter för framtida återanvändning.

2.3.3 Basera återanvändning på produktfamiljer

Ett effektivt sätt att använda återanvändning baserar sig på produktfamiljer (Sommerville, 2001). En produktfamilj är en samling av applikationer med en gemensam arkitektur. Varje applikation är specialiserad på något sätt, men den gemensamma kärnan av produktfamiljen återanvänds när en ny applikation tas fram. Specialiseringen av en produktfamilj kan enligt Sommerville (2001) ske på olika sätt:

Plattformsspecialisering
Olika versioner av applikationen utvecklas för olika plattformar, exempelvis Windows eller Linux. Applikationens funktionalitet är oförändrad. Bara de komponenter som rör hårdvara och operativsystem ändras.

Konfigurationsspecialisering
Olika versioner av applikationen utvecklas för att hantera olika periferienheter. Funktionaliteten kan variera i förhållande till periferienheternas funktionalitet. De komponenter med gränssnitt mot periferienheterna ändras.

Funktionell specialisering
Olika versioner av applikationen utvecklas för kunder med olika krav. Till exempel kan ett bibliotekssystem specialiseras beroende på om det är ett kommunalt bibliotek eller ett universitetsbibliotek.

Att utveckla en applikation genom att anpassa en generisk version av applikationen innebär att en stor del av koden återanvänds. Processen att anpassa en applikationsfamilj för att skapa en ny applikation ser olika ut beroende på bland annat applikationsdomänen. En generisk process kan se ut så här:

1. Ta fram krav. Kan vara en vanlig kravframtagningsprocess, men normalt uttrycks kraven som de modifieringar som skall göras i förhållande till det redan existerande systemet i applikationsfamiljen.

2. Välj bästa applikationsfamiljemedlem. Kraven analyseras och den medlem i applikationsfamiljen som bäst uppfyller kraven väljs.

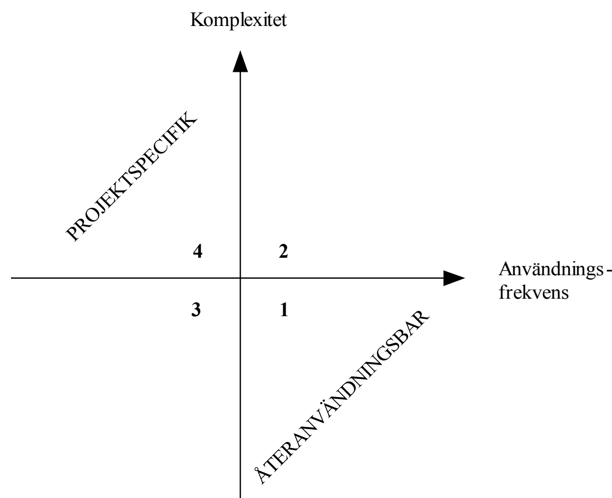
3. Omförhandla kraven. När man vet mer vilka ändringar som krävs, kan vissa krav omförhandlas för att minimera de nödvändiga ändringarna.

4. Anpassa existerande system. Existerande komponenter anpassas för att uppfylla de nya kraven och nya komponenter utvecklas.

5. Leverera den nya applikationsfamiljemedlemmen. Den nya applikationsfamiljemedlemmen levereras till kunden. Den nya medlemmen dokumenteras så den kan användas som underlag för utveckling av nya system i framtiden.

2.3.4 Val av komponent för återanvändning

En återanvändningsbar komponent skall kunna användas i flera projekt. För att avgöra om en komponent är lämplig att återanvända, och därmed skall utvecklas för återanvändning, kan följande modell ur Software Reuse (Coulange, 1998) användas.



Figur 2: *Klassificering av komponenter.*

Komponenter klassificeras i fyra grupper:

1. De med högre användningsfrekvens än den genomsnittliga användningsfrekvensen och mindre komplexitet än genomsnittet. Dessa komponenter är tveklöst återanvändningsbara.
2. De med högre användningsfrekvens än den genomsnittliga användningsfrekvensen och större komplexitet än genomsnittet.
3. De med lägre användningsfrekvens än den genomsnittliga användningsfrekvensen och mindre komplexitet än genomsnittet.
4. De med lägre användningsfrekvens än den genomsnittliga användningsfrekvensen och större komplexitet än genomsnittet. Dessa komponenter är inte lämpliga att återanvända, eftersom de är för projektspecifika.

Tröskelvärden kan användas istället för medelvärden.

Modellen utgår från en uppskattning av användningsfrekvensen, det vill säga hur många utvecklingsprojekt som kommer använda sig av komponenten, och ett mått på komponentens komplexitet. Att förstå mer vad komplexitet är, underlättar hanteringen av komplexiteten och gör det lättare att avgöra hur komplex en komponent är. Därför följer först en definition av orden komplicerad och komplex och därefter tas Rechers (1998) nio aspekter av komplexitet upp.

Cilliers (1998) har bra definitioner av komplicerad och komplex:

Komplicerad
”Om ett system, trots att det kan bestå av ett enormt antal komponenter, kan ges en komplett beskrivning genom sina individuella beståndsdelar, så är ett sådant system bara komplicerat. Jumbojets och datorer är komplicerade.”

Komplex
”I ett komplext system är interaktionen mellan systemets beståndsdelar och interaktionen mellan systemet och dess omgivning sådan att systemet som helhet inte fullt kan förstås bara genom att analysera dess komponenter. Hjärnan, naturliga språk och sociala system är komplexa.”

Recher (1998) anser att ju komplexare en uppgift är, desto mer resurser i form av tid, energi och ansträngning kräver den. Recher har nio aspekter av komplexitet som kan användas vid framtagande av komplexitetsmått:

1. **Beskrivningens komplexitet (Descriptive complexity).** Hur stor beskrivning som krävs för att ge en adekvat beskrivning av systemet i fråga. Ju mer komplext systemet man skall utveckla är, desto större designdokument krävs för att man skall kunna konstruera systemet. Beskrivningen kan också vara komplex genom att den är så vag att man inte riktigt vet vad som skall utvecklas. Ständiga ändringar ökar också komplexiteten för utvecklarna. När systemet är färdigutvecklat visar beskrivningens komplexitet hur komplext systemet blev.

2. **Genereringens komplexitet (Generative complexity).** Hur mycket instruktioner som krävs för att konstruera systemet i fråga. Denna komplexitet är beror starkt på utvecklarnas kompetens. Erfarna utvecklare behöver inte lika utförliga instruktioner som oerfarna. Utvecklare kan minska denna komplexitet genom att kräva mer instruktioner.

3. **Lösandets komplexitet (Computational complexity).** Mängden tid och ansträngning som krävs för att lösa problem. Även denna komplexitet beror på utvecklarnas kompetens. Det är bra om man kan identifiera de delar med högst lösningskomplexitet, så man kan lägga sin intellektuella kraft där. Det kan exempelvis handla om en algoritm för att hitta bästa transportväg. Ett sätt att hantera lösandets komplexitet är att man samlar projektgruppen för att lösa problemet gemensamt.

4. **Mängdens komplexitet (Constitutional complexity).** Antalet ingående element eller komponenter i systemet. Det kan handla om antalet klasser och antalet delsystem, gränssnitt med mera. Att systemet består av flera delar underlättar arbetsfördelningen. I ett IT-baserat system behöver mängden i sig inte vara ett problem, eftersom de olika komponenterna ofta kan utvecklas fristående. Interaktionen mellan komponenter kan däremot vara komplex. Jämför Cilliers (1998) definition ovan.

5. **Variationens komplexitet (Taxonomical complexity).** Antalet av varierande ingående element i systemet. Ett system kan ha få ingående komponenter, men många varianter. Ett exempel kan vara ett system som konverterar mellan olika filformat som kan ha väldigt många varianter på ”läs fil” och ”skriv fil”. Ett sådant system är ju inte så mängdkomplext men väldigt variationskomplext.

6. **Organiseringens komplexitet (Organizational complexity).** Antalet möjliga sätt att arrangera de olika elementen på i systemet. Olika komponenter kan ju ha olika grad av sammankoppling med andra komponenter. En administratör i ett logistiksystem skall ju exempelvis kunna göra allt som en kund, transportör och leverantör kan göra. Det är viktigt att tänka på vid utformningen av systemet.

7. **Hierarkiens komplexitet (Hierarchical complexity).** Antalet hierarkiska nivåer i systemet. Här kan det handla om arkitektur, exempelvis en treskikts arkitektur bestående av gränssnitt, funktion och modell. Även klassernas hierarki kan avses. En välutbyggd klasshierarki ökar komplexiteten samtidigt som den möjliggör återanvändning av kod.

8. **Operationskomplexitet (Operational complexity).** Antal sätt något kan fungera på i systemet. Ett program som bara exekveras och sedan lämnar ifrån sig ett värde har låg operationskomplexitet. Ett händelsebaserat system som styrs av användarens val har hög operationskomplexitet. Det kan även röra sig om att en viss funktion i systemet, exempelvis en beställning, kan ske på flera olika sätt: automatiskt, manuellt etc. Ju mer funktionalitet, desto större operationskomplexitet.

9. **Lagbundenhetens komplexitet (Nomic complexity).** Mängden ”lagar” som påverkar systemet. I ett logistiksystem skulle det kunna vara regler för hur en leverantör skall väljas utifrån pris och leveranstid. Det rör sig om saker som måste tas i beaktande vid utveckling av systemet och som ökar komplexiteten.

Många komplexitetsmått utgår bara från mängdens komplexitet i form av antal attribut och metoder ett objekt har, antal rader kod och så vidare. Man bör försöka få med så många som möjligt av de nio komplexiteterna i det komplexitetsmått man väljer att använda.

2.3.5 Använd objektorienterad systemutveckling

Objektorienterad systemutveckling kan användas vid all systemutveckling, men passar bäst vid utveckling av system för att övervaka, administrera eller styra ett problemområde. Ett problemområde är den del av datasystemets omgivning som administreras, övervakas eller styrs. Då ett datasystem har ett statiskt problemområde, det kan gälla verktygssystem som till exempel ordbehandlare, är det ingen större skillnad mellan objektorienterad analys och design och andra metoder. (Mathiassen, & Munk-Madsen, & Nielsen, & Stage, 1998).

Många likställer återanvändning med objektorientering, men det går att jobba med återanvändning även med andra tekniker. Bernard Coulange har i sin bok Software Reuse (Coulange, 1998) jämfört och rangordnat olika tekniker för återanvändning. De tekniker som jämfördes var kopiering, användning av pre-processor, bibliotek, paket, objekt, generella paket och objekt samt objektorientering. Teknikerna utvärderades enligt dessa kriterier: produktivitet, underhållbarhet, pålitlighet, utbyggbarhet och anpassningsbarhet. Kopiering var sämsta tekniken med 15 av 60 poäng. Objektorientering var bästa tekniken där användning av Ada paket (i Ada görs objekt med hjälp av paket) fick 52 poäng. Eiffel var det bästa objektorienterade programmeringsspråket och fick 60 poäng.

2.3.6 Bygg upp ett bibliotek av återanvändningsbara komponenter

Utan ett bibliotek av återanvändningsbara komponenter finns inga komponenter att återanvända. Komponentbiblioteket skall göra komponenterna tillgängliga för utvecklarna. Alla bör ha tillgång till komponenterna. Däremot skall inte vem som helst kunna lagra komponenter i biblioteket, eftersom risken då är stor att dåligt verifierade och dokumenterade komponenter lagras.

Någon form av administratör behövs för att sköta biblioteket och regler behövs för lagring av komponenter, exempelvis att endast certifierade komponenter får lagras, så inte vem som helst lägger in komponenter som kanske inte skall finnas i biblioteket.

Biblioteket är ett programvaruverktyg som lagrar komponenter, har sökfunktion och versionshantering. Under utvecklingsarbetet jobbar man mot biblioteks-verktyget där man ”checkar ut” en fil. När man är klar och har ”checkat in” filen lagrar verktyget ändringarna inkrementellt, det vill säga bara ändringarna lagras varje gång och inte hela filen.

Biblioteket bör ha en sökfunktion som gör det möjligt att hitta rätt komponent.
Sökfunktionen bör kunna söka i dokumentationen för en komponent, alltså kravspecifikationen, designspecifikationen, testspecifikationen och källkods-kommentarerna.

Det skall också på ett enkelt sätt med hjälp av verktyget gå att hämta ut rätt version av en komponent. Biblioteket skall kunna hantera olika versioner av komponenterna, eftersom olika projekt kan kräva anpassningar av en komponent vilket leder till nya versioner. Det är därför väldigt viktigt att biblioteksverktyget kan hantera de olika versionerna på ett bra sätt.

2.3.7 Dokumentera komponenterna och återanvänd dokumentationen

För att kunna återanvända komponenter räcker det inte med att passande komponenter finns tillgängliga. Man måste förstå hur de fungerar för att kunna återanvända dem. För att kunna återanvända en komponent behöver man ha tillgång till kravspecifikationen och designspecifikationen så man vet vad komponenten erbjuder och om den är lämplig att använda i det aktuella projektet. Så man kan se en återanvändningsbar komponent som bestående av komponenten i binär form (exekverbar fil) med källkod, kravspecifikation, designspecifikation och testdokumentation med testfall. Samtliga dessa delar bör återanvändas.

I avsnitt 16.2 i Objektorienterad analys och design (Mathiassen, & Munk-Madsen, & Nielsen, & Stage, 1998) beskrivs hur dokument skall utformas. I avsnittet finns även mallar för analys- och designdokument. Kravspecifikation, designspecifikation och testspecifikation bör vara standardiserade i form av dokumentmallar som beskriver innehållet på rubriknivå. Det underlättar både framställandet och läsningen av dokumenten. Dokumentationen av en komponent utgörs av kravspecifikation, designspecifikation, testspecifikation och källkodskommentarer.

Kravspecifikation
Kravspecifikationen beskriver den funktionalitet en komponent skall erbjuda. För att veta vad som skall utvecklas behövs en bild av de uttryckta och implicita behov en kund har. Dessa behov kallas krav.

Designspecifikation
Designspecifikationen beskriver hur komponenten skall uppfylla kraven som ställs i kravspecifikationen. Genom analys av kraven tas systemets interna struktur fram, det vill säga arkitektur, komponenter, gränssnitt och data för systemet definieras. Designspecifikationen är en komplett beskrivning av systemets nedbrytning i mindre delar och kravens fördelning på dessa.

Testspecifikation
Testspecifikationen beskriver hur testningen genomförts, valet av testfall och vilka testmetoder som använts. Även resultat från testverktyg som exempelvis kontrollerar kodtäckning, kan ingå.

Källkodskommentarer
Coulange (1998) anser att det överst i källkodsfilerna skall finnas en kommentar bestående av tre delar:

- Nyckelord. Här skrivs de ord som har relation till komponenten. Nyckelorden gör det lättare att klassificera komponenten och underlättar sökningar.
- Sammanfattning. En kort beskrivning av vad komponenten gör. Sammanfattningen används för att avgöra om komponenten är lämplig att använda i det aktuella projektet.
- Beskrivning. Fullständiga anvisningar för användning av komponenten som beskriver hur man använder komponenten

Förutom en huvudkommentar överst bör funktioner, globala variabler och konstanter kommenteras. Kommentarer bör i övrigt finnas där en kort beskrivning kan underlätta förståelsen. En bra guide till hur man kommenterar kod ges i avsnitt 1.6 i The Practice of Programming (Kernighan & Pike, 1999).

Programmeringsspråket Java har ett bra sätt att generera standardiserad dokumentation av kod från kommentarer i koden. Javadoc genererar html-sidor med beskrivningar av klasser som ser ut precis som Javas egna API.[2]

2.3.8 Certifiera komponenterna

Det är alltid viktigt att testa och verifiera sin kod. Detta gäller i ännu högre utsträckning för återanvändningsbara komponenter. Annars kommer inte återanvändning att fungera, eftersom utvecklare inte vågar använda de återanvändningsbara komponenterna. Dessutom riskeras flera projekt om en återanvändningsbar komponent är felaktig.

Enligt Coulange (1998) är det viktigt att certifiera en komponent. Det innebär att utföra ett antal kontroller för att så långt som möjligt kunna säkerställa att komponenten är felfri. Man verifierar att funktionaliteten är den avsedda och att önskad prestanda uppnås. Även dokumentationen verifieras så att den stämmer överens med koden. Det Coulange kallar certifiering är egentligen bara vanlig verifiering, möjligen noggrannare genomfört. Man skall i alla fall se till så att de komponenter som görs tillgängliga för utvecklare är pålitliga.

---

[2] Läs om javadoc på: http://java.sun.com/j2se/javadoc/index.html.

2.3.9 Låt komponenterna finansiera sig själva

Det är dyrare att utveckla generella, återanvändningsbara komponenter. Det är främst två orsaker till detta. Storleken blir större för en återanvändningsbar komponent än en projektspecifik komponent. Dessutom blir utvecklingskostnaden per kodrad högre för en återanvändningsbar komponent (Mili, & Mili, & Yacoub, & Addy, 2002).

En återanvändningsbar komponent skall användas i flera projekt. Därför bör också kostnaden för att ta fram komponenten belasta flera projekt.

En lösning är att skapa en internprissättning för komponenterna. När ett projekt använder en komponent betalar de för komponenten, precis som de skulle ha gjort om de köpt den från annat håll. På samma vis köper komponentbiblioteket skapade komponenter från projekten, vilket på så sätt kompenserar den merkostnad projektet har haft för att skapandet av komponenten. Därmed belastas inte en kund med hela utvecklingskostnaden. Det skapas helt enkelt en intern marknadsplats för komponenterna.

2.3.10 Versions- och konfigurationshantering är nödvändig

Ett systems konfiguration består av en uppsättning komponenter i en viss version. Ett projekt utvecklar ett system med en viss konfiguration.

Komponenter vidareutvecklas när fel upptäcks och korrigeras när ny funktionalitet behövs. Vid större ändringar skapas en ny version. Sådan anpassning av en komponent leder till att komponenten existerar i flera olika versioner. Om ändringarna gör att komponenten inte längre är kompatibel med den tidigare komponenten skapas istället en ny komponent.

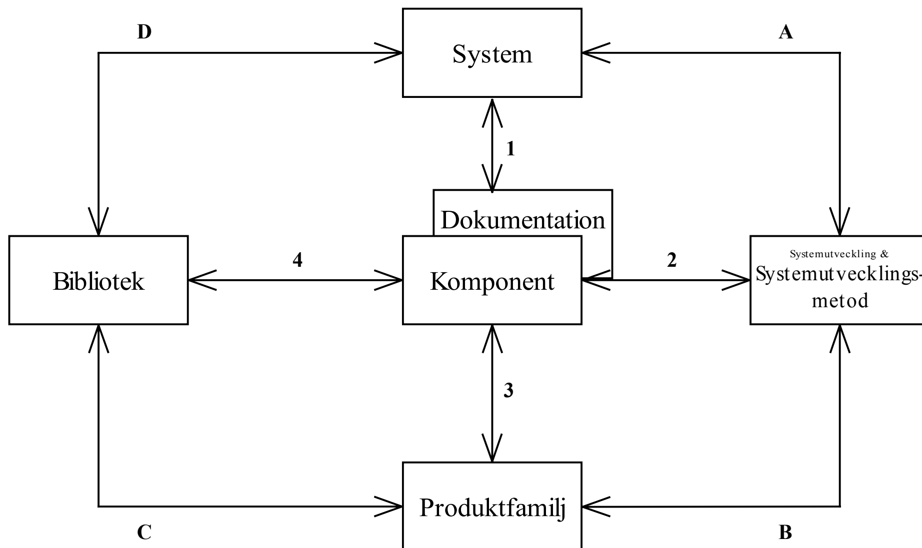
Anledningen att man måste jobba med olika versioner av en komponent är att när en ändring av en komponent behöver utföras, så används komponenten redan i flera projekt. Dessa projekt skulle inte uppskatta att vi plötsligt ändrade en komponent så de fick nya problem att lösa. Därför kan de fortsätta som vanligt medan vi skapar en ny version där vi gör ändringarna. När sedan nya projekt behöver komponenten kan de använda den senaste versionen som förhoppningsvis är felfriare eller har utökad funktionalitet. Enligt Coulange (1998) kan inte återanvändning fungera ifall inte ett system för konfigurationshantering används.

För att underlätta för framtida utvecklare är det lämpligt att ha en uppdaterad historik över de ändringar som gjorts. Revisionshistoriken kan exempelvis vara en kommentar överst i koden.

2.4 Sammanfattning av teoretisk utredning

2.4.1 En modell för vägledning av komponentbaserad systemutveckling

För att sammanfatta och strukturera teoriutredningen har en modell tagits fram. Avsikten är att modellen skall kunna användas som vägledning vid komponentbaserad systemutveckling.



Figur 3: *En modell för vägledning av komponentbaserad systemutveckling.*

I modellen finns de huvudsakliga delar principerna berör alltså: komponent och dokumentation, av komponent, produktfamilj, systemutveckling och systemutvecklingsmetod, bibliotek, samt system.

I modellen återfinns följande förhållanden:

1. Ett system utvecklas med komponenter och då behövs även dokumentationen av komponenterna. En komponent utvecklas och dokumenteras för att ingå i ett system. Komponenten kan antingen vara specifik för ett visst system eller återanvändningsbar för att ingå i flera olika system.
2. Systemutvecklingsmetoden vid komponentbaserad utveckling är annorlunda än vid traditionell systemutveckling. Arbetsfördelningen underlättas vid komponentbaserad utveckling eftersom flera personer ofta kan arbeta parallellt vid utveckling av komponenter. Komponenten och dess dokumentation påverkar systemutvecklingen när ett system utvecklas.
3. En komponent utvecklas för och hör sedan till en viss produktfamilj. En komponent kan ingå i flera produktfamiljer, antingen i samma version eller i olika versioner för varje produktfamilj. Produktfamiljen består av ett antal komponenter som kan sättas samman till ett system som blir en

befintlig medlem av produktfamiljen eller som med tillägg av nya komponenter kan bli en ny medlem av produktfamiljen.

4. Komponenter och deras dokumentation i olika versioner ingår i biblioteket. Vid utveckling av en komponent kan komponenter och deras dokumentation återanvändas ur biblioteket. Biblioteket lagrar och håller reda på komponenter och deras dokumentation i olika versioner.

A. System utvecklas med hjälp av en systemutvecklingsmetod. Systemutvecklingsmetoden avgör hur systemet utvecklas och hur bra det blir genom exempelvis verifieringskrav.

B. Systemutvecklingsmetoden måste vid utveckling för produktfamiljer analysera och hitta de delar som är gemensamma för produktfamiljen så att dessa delar kan bli återanvändningsbara komponenter.

C. Produktfamiljer styr bibliotekets struktur genom att biblioteket ordnar komponenterna efter den produktfamilj de tillhör.

D. Biblioteket styr vilken konfiguration ett nytt system får, det vill säga vilken version av varje komponent som används vid utvecklingen av systemet. Redan vid analysfasen av ett nytt systemutvecklingsprojekt kan sökningar i biblioteket visa vilka komponenter som finns tillgängliga så designen av systemet kan baseras på de tillgängliga komponenterna. Ett färdigt system kan uppdateras med en ny komponent eller en ny version av en komponent från biblioteket.

Även Rechers (1998) nio komplexiteter (se 2.3.4) ingår i modellen och sambandet mellan principerna och komplexiteterna framgår av tabellen nedan.

Princip	**Kort sammanfattning**	**Förhållande som avses i modellen**
Återanvändningsanpassa systemutvecklings-metoden.	Systemutvecklingsmetoden byggs ut med ytterligare faser där återanvändning beaktas.	Beskrivningens komplexitet mellan Systemutvecklings-metod och Komponent.
Basera återanvändning på produktfamiljer.	Ett nytt system utvecklas genom att anpassa ett färdigt system som är medlem av en produktfamilj.	Genereringens komplexitet mellan Komponent och Produktfamilj.
Val av komponent för återanvändning.	En komponent väljs att utvecklas för återanvändning utifrån parametrarna komplexitet och användningsfrekvens.	Lösandets komplexitet mellan System och Komponent.
Använd objektorienterad systemutveckling.	Objektorienterad systemutveckling är bäst för utveckling av återanvändningsbara komponenter.	Mängdens komplexitet mellan Systemutvecklings-metod och Komponent.

Bygg upp ett bibliotek av återanvändningsbara komponenter.	Ett uppbyggt bibliotek av återanvändningsbara komponenter krävs för att det skall finnas något att återanvända.	Variationens komplexitet mellan Komponent och Bibliotek.
Dokumentera komponenterna och återanvänd dokumentationen.	Utförlig dokumentation av komponenter i form av kravspecifikation, designspecifikation, testspecifikation och källkodskommentarer krävs för att man skall kunna återanvända komponenterna.	Organiseringens komplexitet mellan Komponent och Bibliotek.
Certifiera komponenterna.	Återanvändningsbara komponenter måste vara pålitliga och felfria. Därför verifieras funktionalitet, prestanda och dokumentation.	Hierarkiens komplexitet mellan Komponent och Bibliotek.
Låt komponenterna finansiera sig själva.	Kostnaden för att utveckla en återanvändningsbar komponent bör fördelas på samtliga projekt som använder komponenten.	Operationskomplexitet mellan Systemutvecklings-metod och Komponent.
Versions- och konfigurationshantering är nödvändig.	Anpassningar av komponenter leder till att flera versioner uppstår av varje komponent och detta måste hanteras.	Lagbundenhetens komplexitet mellan System och Komponent.

Tabell 2: *Samband mellan principer och komplexiteter.*

3 METOD

3.1 Litteraturstudie

En omfattande litteraturstudie gjordes för att hitta principer för utveckling av återanvändningsbara komponenter. Litteraturstudien var alltså utöver vad som ingår i den inledande litteraturgenomgång som finns i inledningen av Backmans (1998) forskningshjul. Vid valet av de nio principer jag kom fram till i litteraturstudien användes de fyra kriterierna i avsnitt 2.3.1. Det var även viktigt att de var så utförligt beskrivna att det framgick varför principen var viktig, det vill säga att det fanns argument för att använda principen vid utveckling av komponenter för återanvändning, samt att det även framgick hur en användning av principen skulle kunna gå till.

3.2 Enkätundersökning

3.2.1 Planering och genomförande av undersökningen

Jag valde att göra en kvantitativ undersökning i enlighet med Management Research (Easterby-Smith, & Thorpe, & Lowe, 2002). Datainsamling gjordes med hjälp av en enkätundersökning och dataanalys gjordes med hjälp av deskriptiv statistik i form av diagram.

Vid utformning av enkäten valdes slutna frågor, det vill säga frågor med färdiga svarsalternativ, för rangordning av kandidatprincipernas betydelse enligt skalan ingen, liten, viss, stor och avgörande som motsvarades av siffrorna 1 till 5, där 1 motsvarade ingen betydelse. Avslutningsvis fanns en öppen fråga där egna förslag på kandidatprinciper kunde ges. Frågeformuläret distribuerades med avsnitt 2.3 i denna uppsats, som beskriver kandidatprinciperna.

Dataregistrering gjordes enligt boken Från datainsamling till rapport (Dahmström, 2000). Enkätens svar sammanställdes i en datamatris i Microsoft Excel med svaren på frågorna i raderna och svarspersonerna i kolumnerna. Därefter genererades deskriptiv statistik i form av diagram utifrån siffrorna i matrisen. Diagrammen redovisas i resultatavsnittet.

3.2.2 Val av undersökningsobjekt

Jag gjorde examensarbetet Test Environment Simulation Tool (se förord sida 2) på EMW och där väcktes mitt intresse att skriva uppsats om återanvändning av kod. Att göra undersökningen på EMW föll sig därför naturligt, då jag hade möjlighet att göra en undersökning där.

Fem personer från två olika avdelningar inom EMW valdes ut av min handledare på EMW, Björn Karlsson. Detta tillvägagångssätt för att välja lämpliga personer att besvara enkätundersökningen valdes för att säkerställa att endast kvalificerade personer med erfarenhet av återanvändning berördes.

3.2.3 Beskrivning av verksamheten på EMW

Ericsson Microwave Systems AB, divisionen för ytradar, tillverkar radarstationer, som är mycket tekniskt komplicerade produkter. Projekten är i regel flera år långa och involverar många människor.

Ett radarsystem bryts ner i många olika delsystem, där vissa av dem bara hanterar hårdvara, medan andra delsystem även hanterar mjukvara. En radar är en väldigt komplex produkt som består av många olika delar som samverkar för att nå ett gemensamt mål. De olika delarna är i stora drag antenn med sändare, mottagare, signalbehandling och databehandling.

Antenn
Antennens och sändarens uppgift är att verka som omvandlare mellan elektromagnetisk vågutbredning i fria rymden och ledningsbundna elektromagnetiska vågor. I de flesta radartillämpningar används samma antenn för sändning och mottagning. Förenklat brukar man säga att en sändare på en antenn skickar ut energi som sedan reflekteras på föremål som befinner sig i luften, på marken eller till havs beroende på vilket område radaroperatören önskar söka av.

Mottagare
Mottagaren har till uppgift att fånga upp de svaga signaler som reflekterats på ett mål. Mottagaren förstärker sedan signalen till en nivå lämplig för signalbehandling. I mottagarens uppgift ingår också att hålla det internt alstrade bruset på en låg nivå.

Signalbehandling
Signalbehandlingen har bland annat till uppgift att urskilja signaler från mål från övriga signaler så som brus och klotter, sammanlagra målpulser över ett tidsintervall samt fatta beslut vad som är mål och vad som inte är mål med en viss säkerhet.

Databehandling
Databehandlingens uppgift är att se till så man kan följa flera mål i bäring och avstånd under spaning, att kontinuerligt kunna följa ett mål samt att testa och övervaka. Det är i databehandlingen som man avgör vad som skall synas på skärmen. Delsystemet databehandling är ett delsystem med i stort sett bara hantering av mjukvara. Detta delsystem svarar bland annat för en av huvudfunktionerna i radarn: att med ledning av radarns inmätningar skapa, hantera och utvärdera målföljen. Denna funktion, följefunktionen, måste uppfylla många och hårda krav, både funktionella krav och prestandakrav.

3.2.4 Arbete med återanvändning på EMW

I denna genomgång av Ericsson Microwave Systems arbete med återanvändning avses delsystemet Databehandling för produktfamiljen Giraffe Mk VI. EMW har jobbat med återanvändning sedan 1996 och har nått 80 % återanvändning, det vill säga 80 % av programvarukoden i ett nytt system består av återanvänd kod.

En introduktion till arbetet med återanvändning inom databehandlingen ges i processbeskrivningen ”Reuse within data processing sub systems for Giraffe Mk VI” (Ericsson dokumentnr 24/155-FEA 202 203/2 Uen, 2002). Dokumentet ingår i MOOSE, Methodology for Object-Oriented Software Engineering.

MOOSE är en objektorienterad systemutvecklingsmetod. Metoden är framtagen av Ericsson Microwave Systems på enheten för databehandling och används vid all systemutveckling inom databehandling. MOOSE omfattar bland annat delarna Kravinsamling, Kravanalys, Systemkonstruktion, Programkonstruktion, Kodning, Programvaruintegration, Verifieringsförberedelser och Verifiering. MOOSE är en iterativ process där arbetet går ut på att först ta fram ett system med begränsad funktionalitet där alla steg i processen gås igenom. Därefter byggs funktionaliteten för systemet ut i olika steg, inkrement, där alla delarna i processen återigen upprepas. Alla utvecklare har en MOOSE-pärm som innehåller utförliga beskrivningar av de olika delarna i metoden. De finns även lätt åtkomliga på EMWs interna nätverk.

EMW använder CORBA för att koppla samman komponenter. Alla komponenter skall ha sina gränssnitt definierade enligt CORBA 2.0 IDL standard.

Ericssons dokument för kravspecifikation och designspecifikation är standardiserade. Instruktioner för framtagningen av dessa dokument finns i MOOSE.

Kodningsstandard finns med namngivningsregler, exempelvis skall variabelnamn inledas med T_ för typ och C_ för konstant.

Det finns även kodningsmallar med ett huvud som anger vad det är för fil samt med utförliga kommentarer för varje funktion och procedur.

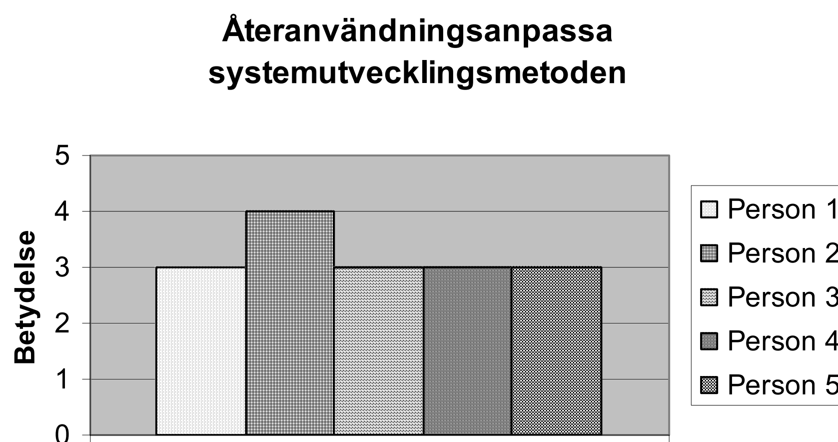
EMW använder biblioteksverktyget Rational ClearCase. Det fungerar delvis som en vanlig filhanterare där varje komponent har en mapp med undermappar för Requirement, Design, Test, Interface, Source etc. ClearCase håller även reda på versioner och vem som har en viss fil ”utcheckad”. Alla ändringar lagras inkrementellt, det vill säga bara ändringarna lagras varje gång inte hela filen.

4 RESULTAT

4.1 Separat redovisning av svaren för varje fråga

4.1.1 Återanvändningsanpassa systemutvecklingsmetoden

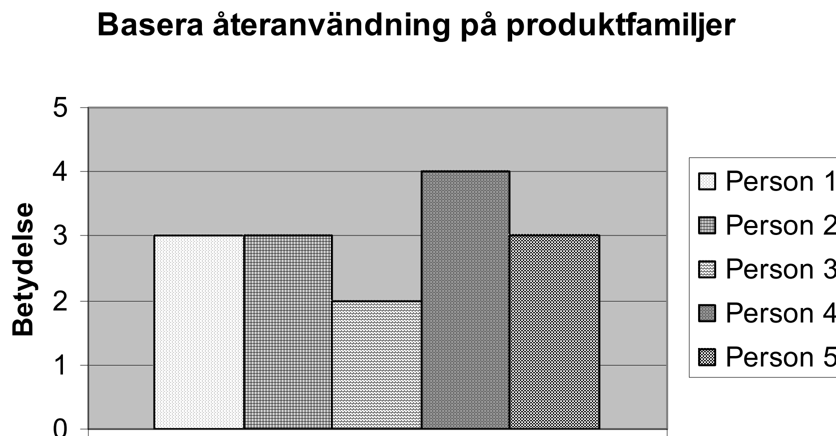
Vilken betydelse har kandidatprincipen, **återanvändningsanpassa systemutvecklingsmetoden,** för utveckling av komponenter för återanvändning?



Figur 4: *Betydelsen kandidatprincipen, återanvändningsanpassa systemutvecklingsmetoden, har för utveckling av komponenter för återanvändning.*

4.1.2 Basera återanvändning på produktfamiljer

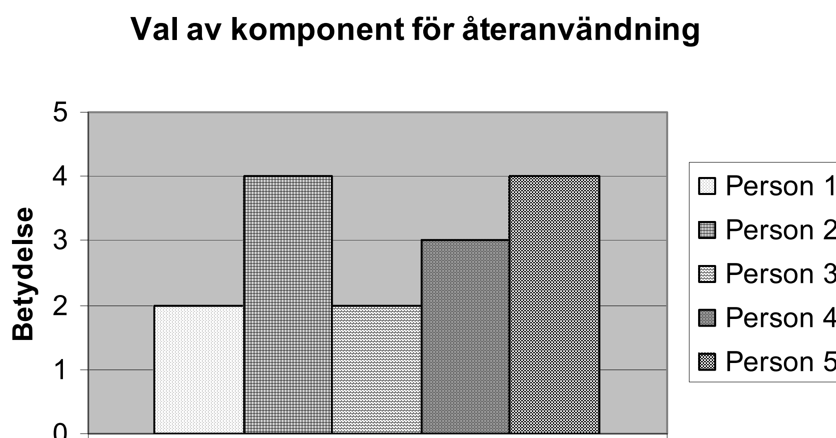
Vilken betydelse har kandidatprincipen, **basera återanvändning på produktfamiljer,** för utveckling av komponenter för återanvändning?



Figur 5: *Betydelsen kandidatprincipen, basera återanvändning på produktfamiljer, har för utveckling av komponenter för återanvändning.*

4.1.3 Val av komponent för återanvändning

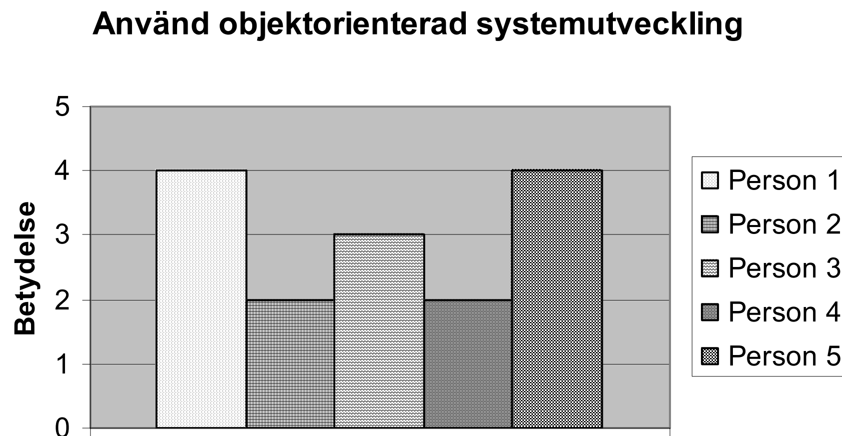
Vilken betydelse har kandidatprincipen, **val av komponent för återanvändning,** för utveckling av komponenter för återanvändning?



Figur 6: *Betydelsen kandidatprincipen, val av komponent för återanvändning, har för utveckling av komponenter för återanvändning.*

4.1.4 Använd objektorienterad systemutveckling

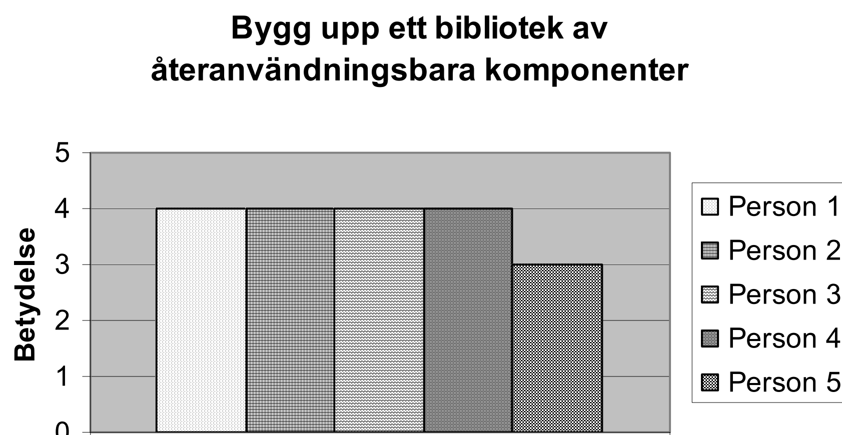
Vilken betydelse har kandidatprincipen, **använd objektorienterad systemutveckling,** för utveckling av komponenter för återanvändning?



Figur 7: *Betydelsen kandidatprincipen, använd objektorienterad systemutveckling, har för utveckling av komponenter för återanvändning*.

4.1.5 Bygg upp ett bibliotek av återanvändningsbara komponenter

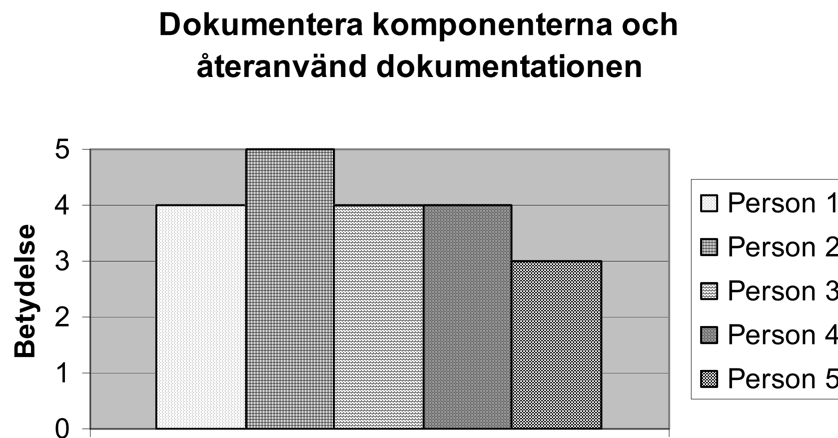
Vilken betydelse har kandidatprincipen, **bygg upp ett bibliotek av återanvändningsbara komponenter,** för utveckling av komponenter för återanvändning?



Figur 18: *Betydelsen kandidatprincipen, bygg upp ett bibliotek av återanvändningsbara komponenter, har för utveckling av komponenter för återanvändning.*

4.1.6 Dokumentera komponenterna och återanvänd dokumentationen

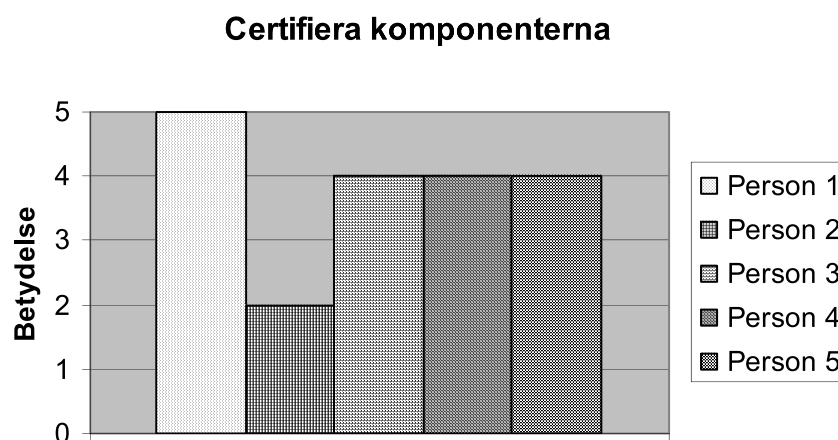
Vilken betydelse har kandidatprincipen, **dokumentera komponenterna och återanvänd dokumentationen,** för utveckling av komponenter för återanvändning?



Figur 9: *Betydelsen kandidatprincipen, dokumentera komponenterna och återanvänd dokumentationen, har för utveckling av komponenter för återanvändning.*

4.1.7 Certifiera komponenterna

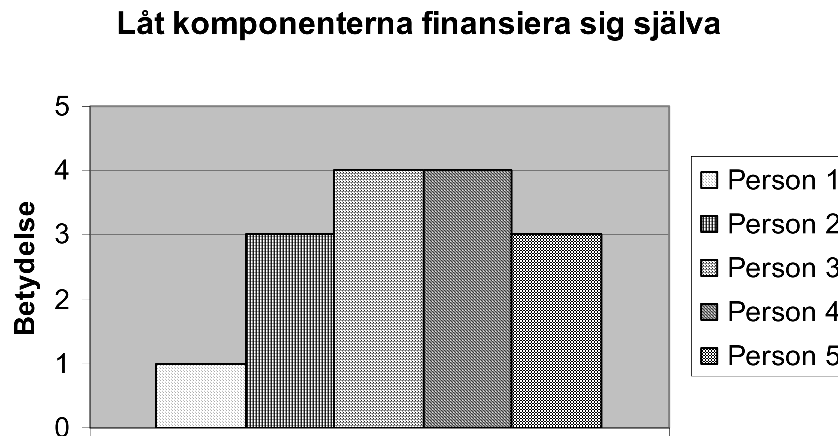
Vilken betydelse har kandidatprincipen, **certifiera komponenterna,** för utveckling av komponenter för återanvändning?



Figur 10: *Betydelsen kandidatprincipen, certifiera komponenterna, har för utveckling av komponenter för återanvändning.*

4.1.8 Låt komponenterna finansiera sig själva

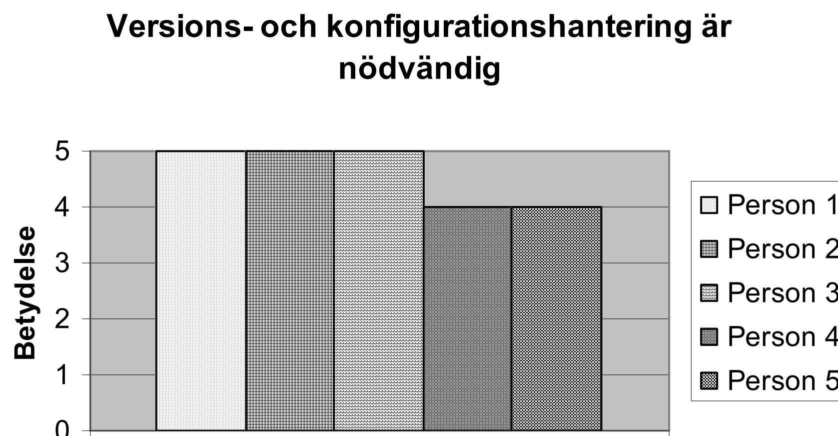
Vilken betydelse har kandidatprincip 8, **låt komponenterna finansiera sig själva,** för utveckling av komponenter för återanvändning?



Figur 11: *Betydelsen kandidatprincipen, låt komponenterna finansiera sig själva, har för utveckling av komponenter för återanvändning.*

4.1.9 Versions- och konfigurationshantering är nödvändig

Vilken betydelse har kandidatprincip 9, **versions- och konfigurationshantering är nödvändig,** för utveckling av komponenter för återanvändning?



Figur 12: *Betydelsen kandidatprincipen, versions- och konfigurationshantering är nödvändig, har för utveckling av komponenter för återanvändning.*

4.1.10 Egna förslag på kandidatprinciper

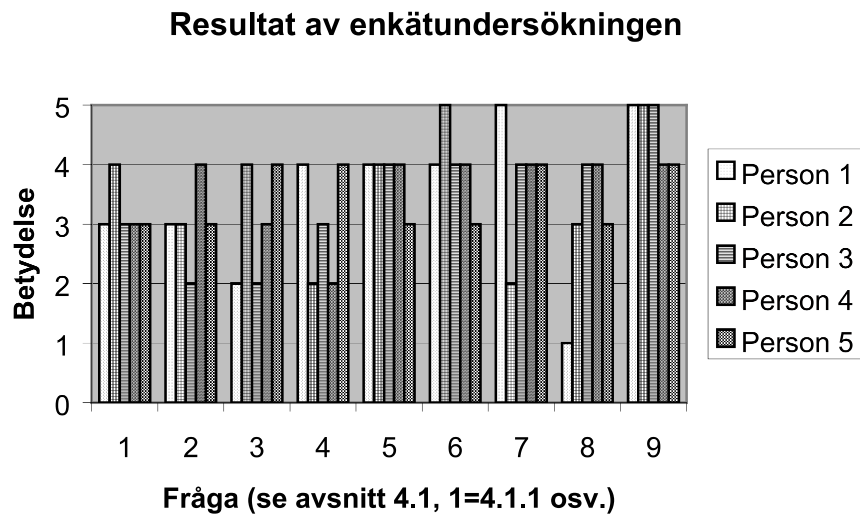
Egna förslag på kandidatprinciper?

Ingen av svarspersonerna lämnade förslag på någon kandidatprincip.

4.2 Sammanfattning av svaren från hela enkätundersökningen

4.2.1 Översiktligt resultat

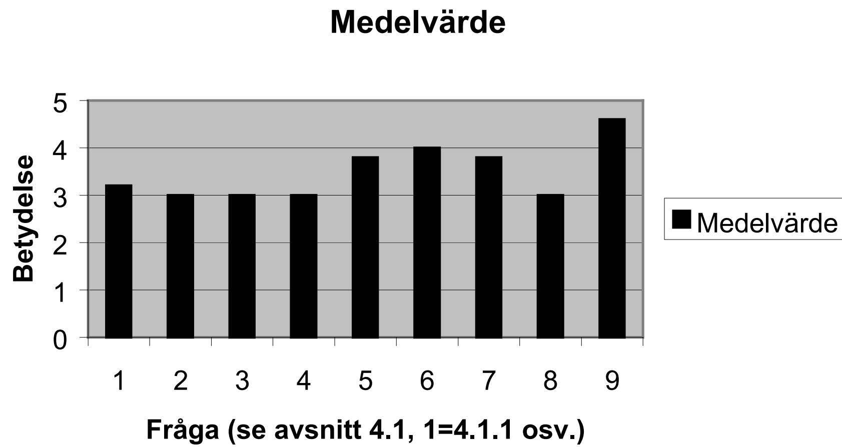
I diagrammet nedan visas betydelsen svarspersonerna gav de nio olika kandidatprinciperna.



Figur 13: *De nio kandidatprincipernas betydelse för utveckling av komponenter för återanvändning.*

4.2.2 Medelvärdet för svaren på respektive fråga

I diagrammet nedan visas medelvärdet av betydelsen svarspersonerna gav de nio olika kandidatprinciperna.



Figur 14: *Medelvärdet för kandidatprincipernas betydelse för utveckling av komponenter för återanvändning.*

Medelvärdet på en fråga beräknades genom att först summera samtliga personers värden för principens betydelse och sedan dividera summan med antalet (fem) personer som besvarade enkätundersökningen.

5 DISKUSSION

5.1 Studien

Hantering av komponentbaserad utveckling är ett nytt område. Det finns många synpunkter och många goda idéer och många bittra erfarenheter och misslyckade fall. Samtidigt har min litteraturstudie visat att det saknas någon dominerande och vägledande modell. Detta gör arbetet svårare att avgränsa och det gör att resultatets godhet blir osäkrare att bedöma.

Denna studie är explorativ eftersom min sökning inte kunde resultera i någon etablerad modell. Detta faktum ledde till en relativt omfattande litteraturstudie som resulterade i nio grundläggande principer. En vidare analys av dessa principer skapade förutsättningar för design av en modell för hantering av komponentbaserad utveckling. Jag anser att modellen har fungerat bra som ett struktureringsinstrument och kan betraktas som såväl valid som trovärdig. I detta avseende bedöms validitetens godhet utifrån det faktum att modellen på ett överblickbart sätt sammanfattar de principer som litteraturstudien har genererat. I samma anda kan modellen betraktas som trovärdig eftersom den empiriska delen inte skapat behov av att införa någon extra dimension eller att ta bort någon dimension. Sammanfattningsvis kan modellen betraktas som utgångspunkt för vidare granskning och förbättring.

5.2 Resultatet

Ser man till medelvärdet för varje fråga som diagrammet i avsnitt 5.2.2 visar så fick frågorna om principerna basera återanvändning på produktfamiljer, val av komponent för återanvändning, använd objektorienterad systemutveckling och låt komponenterna finansiera sig själva ett medel på tre och frågan om återanvändningsanpassa systemutvecklingsmetoden hamnade på ett medel strax över tre. Frågan om dokumentera komponenterna och återanvänd dokumentationen fick fyra i medel och frågorna bygg upp ett bibliotek av återanvändningsbara komponenter och certifiera komponenterna hamnade strax under fyra. Frågan om versions- och konfigurationshantering är nödvändig var den fråga som samtliga tyckte hade stor eller avgörande betydelse för utveckling av komponenter för återanvändning.

Ser man varje fråga för sig så var svarspersonerna ganska överens om vilken betydelse kandidatprincipen har för utveckling av komponenter för återanvändning i frågorna om återanvändningsanpassa systemutvecklingsmetoden, bygg upp ett bibliotek av återanvändningsbara komponenter och versions- och konfigurationshantering är nödvändig. I övriga frågor rådde större oenighet.

Att uttala sig generellt om de olika kandidatprincipernas betydelse utifrån enkätundersökningen är svårt. Vad man kan säga är att ingen av principerna helt förkastades av svarspersonerna.

Det är också svårt att uttala sig om vad skillnaderna i svaren beror på. Svarspersonerna arbetar i och för sig på två olika avdelningar inom EMW, men skillnaderna verkar mer bero på individerna än på arbetsställe.

En förhoppning fanns att den öppna frågan i enkätundersökningen där svarspersonerna uppmanades att lämna egna förslag på kandidatprinciper skulle leda till ytterligare principer. Så blev det inte.

Sammanfattningsvis kan man säga att ingen av kandidatprinciperna kunde avfärdas, utan samtliga principer förtjänar vidare forskning.

5.3 Självkritik och framtida forskning

Vad som ytterligare kunde ha gjorts i denna uppsats och vad som även kan göras i framtida forskning är att ha med flera principer och att utföra undersökningen på flera företag.

Jag tycker ändå, med hänsyn till begränsningarna för en magisteruppsats, att dels teoriavsnittet är tillräckligt stort och därför utrymme för ytterligare principer saknas samt dels att tidsramarna för uppsatsen gör att det inte med framgång går att genomföra undersökningen på flera företag.

6 SLUTSATSER

Syftet med uppsatsen var att ta fram ett antal principer som enligt litteraturen bör styra utveckling av komponenter för återanvändning samt att validera principerna genom en undersökning i praxis.

Genom litteraturstudien togs nio kandidatprinciper fram. En modell utvecklades även för att visa sambandet mellan principerna. Dessa principer validerades i en enkätundersökning. Ingen av kandidatprinciperna förkastades i enkätundersökningen. Därför kan samtliga kandidatprinciper ses som principer för utveckling av återanvändningsbara komponenter.

Avslutningsvis anser jag att syftet med uppsatsen uppfyllts och att principerna förtjänar vidare forskning.

7 REFERENSER

Backman, J. (1998). *Rapporter och uppsatser*. Lund: Studentlitteratur.

Bourke, P., & Dupuis, R., & Abran, A., & Moore, J.W., & Tripp, L., & Wolff, S. (2001). *Fundamental principles of software engineering – a jorney*. The Journal of Systems and Software.

Cilliers, P. (1998). *Complexity and postmodernism: understanding complex systems*. London: Routledge.

Coulange, B. (1998). *Software Reuse*. London: Springer.

Dahmström, K. (2000). *Från datainsamling till rapport*. Lund: Studentlitteratur.

Easterby-Smith, M., & Thorpe, R., & Lowe, A. (2002). *Management Research*. (2nd ed.). London: Sage.

Ericsson Microwave Systems AB. Dokumentnr 24/155-FEA 202 203/2 Uen. (2002). *Reuse within data processing sub systems for Giraffe Mk VI*.

Gamma, E., & Helm, R., & Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-oriented Software*. Reading, Mass.: Addison-Wesley.

Kernighan, B.W., Pike, R. (1999). *The Practice of Programming*. Reading, Mass.: Addison-Wesley.

Mathiassen, L., & Munk-Madsen, A., & Nielsen, P.A., & Stage, J. (1998). *Objektorienterad analys och design*. Lund: Studentlitteratur.

Mili, H., & Mili, A., & Yacoub, S., & Addy, E. (2002). *Reuse-Based Software Engineering: Techniques, Organization and Controls*. New York: Wiley.

Nandu, S. (Ed.). (2001). C# .NET Web Developer's Guide. Rockland, Mass.: Syngress Media

Recher, N. (1998). *Complexity*. New Brunswick, New Jersey: Transaction Publishers.

Sommerville, I. (2001). *Software Engineering* (6th ed.). Harlow: Pearson.

Szyperski, C. (2000). Components and the Way Ahead. In G.T. Leavens & M. Sitaraman (Eds.), *Component Based Systems* (pp 1–20). Cambridge: Cambridge University Press.