



LUNDS UNIVERSITET
Lunds Tekniska Högskola

EDAA35 Utvärdering av programvarusystem VT 2025

Martin Höst
Inst. Datavetenskap, LTH

16 december 2024

Innehåll

I	Teori	1
1	Inledning	3
1.1	Utvärdering av programvarusystem	3
1.2	Vetenskapliga undersökningar	4
1.3	Steg i en vetenskaplig undersökning	5
1.4	Innehåll i kompendiet	5
2	Definition av mål	7
3	Relaterat arbete och litteratursökning	9
3.1	Relaterat arbete	9
3.2	Vetenskapliga artiklar	10
3.3	Sökning efter vetenskapliga artiklar	12
3.4	Andra källor än traditionellt vetenskapliga	15
4	Forskningsmetodik	17
4.1	Inledning	17
4.2	Experiment	19
4.3	Kartläggning	21
4.4	Fallstudie	22
4.5	Aktionsforskning	23
4.6	Design av forskningsstudie	23
5	Mätningar av programvarusystem (metrics)	27
5.1	Inledning	27
5.2	Skalor	28
5.3	Objektiva och subjektiva mått	29
5.4	Direkta och indirekta mått	29
5.5	Vad man man mäter inom mjukvaruutveckling	30
5.6	Målorienterad mätning	31
5.7	Kvalitetsdimensioner	33
5.8	Exempel på mått	34
5.9	Verktyg	39
6	Analys	43
6.1	Sluppmässiga variationer och sannolikheter	43
6.2	Deskriptiva mått	46
6.3	Hantering av ”outliers”	48

6.4	Jämförande tester	50
7	Att rapportera resultat skriftligt	53
7.1	Disposition	53
7.2	Litteraturförteckning	56
7.3	Checklista	63
8	Exekveringsmiljö	65
8.1	Inledning	65
8.2	Exekveringssystem	65
8.3	Övriga anledningar	69
8.4	Exekveringstid	69
9	Prestandamätning	71
9.1	Egenskaper hos prestandamått	72
9.2	Exempel på prestandamått för datorsystem	72
9.3	Prestanda av program	74
9.4	Profileringsverktyg	77
9.5	Prestandamätning i sitt sammanhang	77
II	R	81
10	Språket R	83
10.1	Vad är R?	83
10.2	Grundläggande om språket	84
10.3	Objekt för att representera data	86
10.4	Urval av data	89
10.5	Operationer och vanliga funktioner	90
11	Programmering i R	95
11.1	Skript	95
11.2	Funtioner	95
11.3	Upprepning och villkor	96
11.4	Ett exempel på en funktion	97
11.5	Importera data från fil	98
11.6	Rita grafer	98
11.7	Några apply -funktioner	99
11.8	Ytterligare funktioner	103

Del I

Teori

Kapitel 1

Inledning

1.1 Utvärdering av programvarusystem

I många situationer är det viktigt att kunna utvärdera egenskaper hos programvarusystem. I utveckling av programvarusystem är det inte enbart viktigt att kunna definiera vilka egenskaper ett system ska ha, det är också viktigt att kunna kontrollera vilka egenskaper ett system verkligen har vid verifiering och validering. Under utvecklingens gång vill man ha möjlighet att mäta och kontrollera att systemet verkligen får de egenskaper man önskar.

Det finns flera sorters egenskaper som kan vara av intresse. Man kan t ex dela upp kraven på ett system i funktionella krav och "icke-funktionella" krav, även kallat kvalitetskrav. Funktionella krav beskriver hur systemet ska reagera på olika insignaler och indata och vad det ska svara med. Dessa krav beskriver både hur systemet ska fungera i "normalfall" och i "felfall" och kan t ex beskrivas i användningsfall eller en annan lista med krav i en kravspecifikation. De funktionella kraven kontrolleras vanligtvis i test, både under utvecklingen av programvaran och i slutet av utvecklingen [19].

Kvalitetskrav beskriver begränsningar på systemet, som t ex att svarstiden ska vara maximalt en viss tid, användbarheten ska vara på en viss nivå, systemet ska vara underhållbart vid framtida ändringar, etc. Att utvärdera kvalitetskrav kan ibland vara svårare än att testa funktionella krav. För det första är de inte lika "enkla" att bestämma i början av utvecklingen eftersom de beskrivs mer i form av begränsningar än de funktioner som man verkligen önskar sig. För det andra är de ofta mer ett värde på en flytande skala, t ex att svarstiden ska vara max 0,1 s, än att en funktion fungerar som den ska eller inte. Detta betyder att det krävs ett systematiskt angreppssätt att mäta denna typ av kvalitetsaspekter av programvara, vilket ett vetenskapligt angreppssätt till att utvärdera programvarusystem kan hjälpa till med.

Detta kompendium beskriver hur vetenskapliga undersökningar går till, från formulering av forskningsfrågor till analys av data med diskussion och formulering av slutsatser. Termen "vetenskapligt" kan låta ganska avancerat, precis som det kan låta att tala om "forskning". Vi menar dock att forskning görs i alla utvärderande studier där man efter hand tar reda på hur det ligger till med något man är intresserad av. På samma sätt krävs det ett vetenskapligt angreppssätt, även om detta inte betyder att det måste vara forskning på dok-

torsnivå eller liknande. I alla väl genomförda undersökande studier krävs det ett angreppssätt där man kan förklara varför man genomför de olika steg man genomför och hur de bidrar till att man kan svara på de frågeställningar man har, samt att man kan säga i vilken utsträckning de slutsatser man drar är giltiga och går att lita på.

Fokus i detta kompendium är alltså på kvalitetskrav och inte så mycket på funktionella krav, vilka täcks på ett naturligt sätt av traditionell testning. Tanken är att angreppssättet som presenteras ska kunna användas för att utvärdera programvarusystem.

Något som skiljer programvaruutveckling från annan ingenjörsmässig utveckling och produktion är den flexibilitet som finns under utvecklingen. Under utvecklingen och implementationen finns det möjlighet att göra förändringar i produkten förhållandevis sent, vilket ger en flexibilitet som också utnyttjas vid många tillfällen. Om denna flexibilitet utnyttjas rätt kan den ge stora fördelar och möjliggöra att målen för det som utvecklas fastställs i detalj under utvecklingens gång, vilket utnyttjas då man utvecklar med iterativa metoder vilket är en av grunderna i t ex agila metoder, se t ex [19]. Detta betyder att man efter hand som man utvecklar systemen kontrollerar att man verkligen utvecklat rätt och att man under hela utvecklingen skaffar sig förståelse för hur systemet verkligen ska fungera och justerar målen baserat på detta.

Iterativ utveckling kräver alltså att man regelbundet stämmer av att man är på rätt väg med hjälp av tester och utvärderingar. Den typ av utvärderingar som presenteras i detta kompendium är tänkta att kunna användas i samband med dessa utvärderingar. Utvärderingarna är också tänkta att kunna utföras mer fristående, t ex då man vill utvärdera eller jämför ett antal olika programvaror som löser samma problem. Om man t ex utvecklar ett program för att sortera ut de viktigaste kunderna i en kunddatabas baserat på ett antal kriterier så kan detta vidareutvecklas iterativt med nya kriterier, nya algoritmer, presentationssätt, osv. Vid varje ny version kan man då mäta kvalitetsegenskaper som t ex svarstid för att se att de inte blir sämre för varje version. Man kan givetvis också ha som mål att förbättra kvalitetsegenskaperna för varje version.

1.2 Vetenskapliga undersökningar

En ingenjörs arbete kan vara mycket mångfacetterat med konstruktion, problemlösning, och utveckling. En stor del av en ingenjörs arbete går ut på att göra undersökningar och utredningar och att presentera resultaten av dem. Det kan t ex vara att göra en förstudie till ett utvecklingsprojekt eller att göra en utredning om vilken hårdvara som ska användas i ett inbyggt system. I Högskoleförordningen fastställs det att en civilingenjör t ex ska ”visa förmåga att med helhetssyn kritiskt, självständigt och kreativt identifiera, formulera och hantera komplexa frågeställningar samt att delta i forsknings- och utvecklingsarbete och därigenom bidra till kunskapsutvecklingen” och ”visa förmåga att skapa, analysera och kritiskt utvärdera olika tekniska lösningar”. Dessutom ska en civilingenjör ”visa kunskap om det valda teknikområdets vetenskapliga grund och beprövade erfarenhet samt insikt i aktuellt forsknings- och utvecklingsarbete”. Detta betyder att en civilingenjör måste kunna genomföra undersökningar som är baserade på vetenskapliga metoder. Dessutom innebär det att en civilingenjör måste kunna ta till sig artiklar publicerade i vetenskapliga forum.

1.3 Steg i en vetenskaplig undersökning

Det finns givetvis ingen enkel steg-för-steg-process som gäller för alla vetenskapliga undersökningar. Arbetet sker ofta iterativt och efterhand som man upptäcker nya saker måste man gå tillbaka och göra om en del av de steg som man redan gjort. Trots det finns det en del huvudsakliga faser som man går igenom i denna iterativa process. Dessa presenteras i figur 1.1 och kan sammanfattas enligt följande:

- Definition av mål med studien. Detta resulterar ofta i en lista med konkreta forskningsfrågor som man besvarar i studien.
- Sammanfattning av relaterad forskning och litteratur. Detta resulterar i en litteratursammanfattning som blir en del av den rapport man skriver.
- Planering av forskningen, t ex val av grundläggande metod, tekniker för datainsamling, hur data ska analyseras, etc. Detta resulterar ofta i en metodbeskrivning som sammanfattas i rapporten.
- Genomförande av datainsamling enligt den metod man planerat.
- Analys av data enligt den plan man gjort
- Diskussion och sammanfattning av slutsatser. Här kan man diskutera de resultat man fått fram, jämföra dem med vad som är presenterat i litteraturen, och så vidare. Baserat på detta kan man formulera svar på de frågeställningar man presenterat.
- Muntlig och skriftlig presentation. De flesta studier sammanfattas i en rapport för att andra ska kunna ta del av vad man kommit fram till. Det är också vanligt att man presenterar resultaten muntligt.

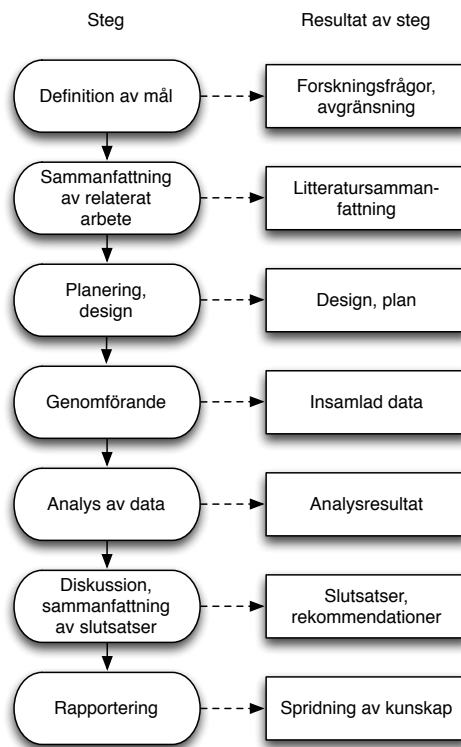
1.4 Innehåll i kompendiet

Stegen i figur 1.1 diskuteras närmare i kapitel 2 till kapitel 7. Detta är tänkt att ge en bas för att kunna göra undersökningar inom ett brett område av aspekter.

För att kunna göra undersökningar om specifika ämnen behöver man kunskap om de bakomliggande tekniska aspekterna. Ett viktigt område i detta kompendium är prestandamätning och mätning av exekveringstid av Java-program. I kapitel 8 och kapitel 9 presenteras den basen kort.

För att kunna utföra beräkningar och databehandling på ett effektivt sätt behöver man verktyg för det. Ett sådant verktyg är statistikspråket R, som presenteras i Del II av kompendiet. Kompendiet är tänkt att användas i en kurs med laborationer och projekt. Del ?? av kompendiet innehåller instruktioner för laborationerna.

Mycket av innehållet när det gäller analys och databehandling är tänkt att ge en förståelse för hur dessa aktiviteter utförs, samt behovet av dem. Tanken är dock inte att gå in på detaljer, eftersom varje område har hela kurser senare i utbildningen som omfattar dem. Ett exempel är statistik och kvantitativ analys. I detta kompendium finns det tillräckligt med material för att göra laborationerna och projektet som ingår i kursen. För att kunna lösa problem i allmänhet och få



Figur 1.1: Steg i vetenskaplig undersökning

en djupare förståelse för varför man gör som man gör krävs det dock kunskaper motsvarande de man får i en grundläggande statistikkurs.

Kapitel 2

Definition av mål

I början av en studie vill man för det mesta tänka ut och formulera i detalj vad man ska göra och vad målet med studien är. Det är givetvis inte möjligt att veta vad resultatet av studien blir, men man kan i alla fall ha en tanke om vad man vill undersöka.

En anledning till att man vill formulera vad man ska göra är att man vill se till att omfånget på det man ska göra varken är för stort eller för litet. Om man har en orimlig målsättning så kommer man inte att kunna bli klar eller i värsta fall inte ha några svar alls innan tiden är slut. Man kan alltså se till att det är ett rimligt omfång på uppgiften och att den inte kräver för mycket kunskaper som man själv inte har, även om man så klart kan lära sig nya saker efter hand (se t ex [8]). Att formulera vad man ska göra och vilka frågor man har ger också möjlighet att kommunicera med andra vad man ska göra. Att kunna beskriva vad man ska göra är en förutsättning för att man ska kunna få kommentarer och värdefulla synpunkter på arbetet, dvs en hjälp i att bestämma vad som ska göras. Man gör också studien i ett sammanhang, t ex som en del av sitt arbete, som en uppgift i utbildningen, som en del av ett forskningsprojekt, som en del av ett utvecklingsprojekt, osv. Sammanhanget påverkar givetvis också vad man tar sig an att göra. Det har alltså att göra med det sammanhang som diskuterades i kapitel 1.1. Om man utvecklar en produkt så kommer man kontinuerligt under utvecklingen att mäta på kvalitetsegenskaperna för att få kontroll över att de t ex inte försämrats under utvecklingen. I själva undersökningen kommer man då att ha som mål att utvärdera de egenskaper som man är intresserad av att följa under utvecklingen.

Det man själv är intresserad av är givetvis också en faktor som påverkar. Man kan alltså beakta till exempel följande då man bestämmer målen med en studie:

- Vad man har resurser att göra under med den tid man har tillgänglig och de resurser man har
- Vad man har kunskaper och kompetens att genomföra
- Vad man har för uppdrag och vilken omgivning arbetet genomförs i

Den första versionen av målbeskrivningen får man göra så gott man kan utifrån den kunskap man har. Man har någon idé om vad man ska göra baserat

på de övergripande frågor man har, den bakgrundskunskap man har, etc. Målet kan mycket väl anpassas efterhand som man arbetar vidare med att bestämma vilken metod och vilka mätningar man ska göra, samt när man tar fram och sammanfattar relaterat arbete och bakgrundsmaterial. Det är mycket vanligt att frågeställningen anpassas efter hand som man får mer kunskap i en studie.

Frågeställningen kan beskrivas och formuleras på olika sätt. Ett sätt är att helt enkelt beskriva den i löpande text. Man kan också beskriva målsättningen som konkreta forskningsfrågor, t ex

- "How game developing organizations test their products?"
- "Does game testing differ from the testing of conventional software products?"

från en studie om hur spelutvecklare testar sina produkter [23]. Numrera gärna dina forskningsfrågor, t ex "RQ1", "RQ2", så blir det lättare att referera till dem, t ex när du presenterar dina slutsatser senare i arbetet. Det är ibland också möjligt att beskriva målet med en studie med hjälp av GQM-ramverket som presenteras i kapitel 5.6.

Kapitel 3

Relaterat arbete och litteratursökning

3.1 Relaterat arbete

I en vetenskaplig undersökning är det viktigt att bygga på och relatera till vad andra gjort i tidigare undersökningar. Detta är en anledning till att det är viktigt att genomföra en litteraturstudie som en del av undersökningen. Målet med litteraturstudien är att sammanfatta det område som man bygger på och på så sätt kunna visa tydligare vad bidraget i den studie som man själv gör är.

Ofta gör man arbetet med litteraturstudien i början av en undersökning, som det visas i kapitel 1.3. Även om kanske det mesta av litteraturstudien görs tidigt i arbetet så fortsätter man under hela studien att leta material. Efterhand som man läser mer så hittar man nya källor som man lägger till och efterhand som man kommer på nya frågor baserat på vad man gjort så läser man nytt material som man lägger till. Det är alltså en iterativ process som pågår under hela studien.

Genom att sammanfatta området når man ett antal mål. Ett är att man kan sätta in sig i området och bättre förstå hur man själv ska genomföra sin undersökning och vad som redan är gjort. För det mesta känner man inte till allt som gjorts innan och det är därför bra att skaffa sig en bättre uppfattning om vad som gjorts av andra innan man själv börjar. Man kan lära sig mycket av vad andra gjort och på så sätt kunna göra en bättre undersökning själv.

Det är också viktigt för de som läser resultatet av ens undersökning att förstå hur den bygger vidare på vad andra gjort, vilket betyder att man måste sammanfatta resultatet av sin litteraturstudie så att de som läser den kan förstå vad som gjorts innan. Man måste alltså sammanfatta resultatet av sin litteraturgenomgång så att de som läser den kan förstå vad som gjorts innan och varför den studie som man själv presenterar är relevant och viktig.

När man väl har resultaten av sin egen undersökning kan man använda den kunskap man fick i litteraturgenomgången för att för att diskutera och jämföra sina resultat med andras. Det blir en viktig del av att dra slutsatser och förklara vad som är ny kunskap av en undersökning.

En väl genomförd litteraturstudie är viktig i alla undersökningar. Det är ganska självklart att den är viktig i en vetenskaplig forskningsstudie och om

man läser en forskningsartikel finns det i stort sett alltid en del som relaterar arbetet som gjorts i studien med annat arbete. Detta är även viktigt i en studie som görs t ex internt i ett företag. Här kan förmodligen en större del av det arbete man refererar till vara interna rapporter och dokument, men det är fortfarande viktigt att sätta det arbete man själv gjort i ett sammanhang.

En viktig del av att söka och sammanfatta relaterat arbete är att bedöma kvalitén av det material man hittar. Ett sätt att göra det är givetvis att läsa materialet själv och bilda sig en uppfattning om hur bra det är. Det är dock inte lätt att endast basera sig på detta, så det finns ett antal andra saker man kan ta med i bedömningen. En är var det material man hittat är publicerat. Om det är en artikel i en tidning så ska man veta att den kan vara skriven baserat på en viss nyhet och det är inte säkert att bakgrundsmaterial etc. är kontrollerat så uttömmande. Om man vet när den är skriven och i vilket syfte så kan man lättare bedöma hur relevant den är. Om det är en vetenskapligt granskad artikel kan man bedöma kvalitén till en viss del baserat på hur den är granskad.

3.2 Vetenskapliga artiklar

För vetenskapliga undersökningar som görs i forskningssyfte finns det en väl utarbetad process för granskning, så kallad "peer review", och publicering. Den skiljer sig åt något mellan olika vetenskapsområden, men i stora drag fungerar det på samma sätt i alla områden.

Forskare på universitet och på andra ställen som på företag gör forskning och publicerar den i t ex tidskrifter och på konferenser och som en del av detta granskas artiklarna av andra forskare inom ämnet för att avgöra om kvalitén är bra nog. Granskningen är alltså en viktig del i kvalitetssäkringen av forskningsresultat.

Artiklar publiceras på olika sätt, vilka kan sammanfattas enligt följande:

- **Tidskriftsartikel:** De mest mogna resultaten presenteras i tidskrifter. Tidskrifter utkommer t ex en gång per kvartal och varje nummer innehåller ett antal artiklar. Varje tidskrift har en "editorial board" med forskare som är ansvariga för granskningen. När en forskare skickar in en artikel till en tidskrift bjuder man in ett antal (t ex tre) andra forskare att granska artikeln. Efter granskningen tas ett beslut om artikeln ska accepteras, uppdateras av författarna och granskas om, eller refuseras. Granskningsprocessen för tidskriftsartiklar är normalt mer rigorös än för andra forum och manuskripten förbättras ofta i flera omgångar.
- **Konferensartikel:** På vetenskapliga konferenser träffas forskare och presenterar och diskuterar nya forskningsresultat inom sitt område. En konferens ges oftast en gång om året på olika platser i världen. Inför varje omgång av en konferens skapas en programkommitté bestående av forskare inom området. Forskare skickar in artiklar som bidrag till konferensen. Dessa bidrag granskas sedan av ett antal personer (t ex tre) från programkommittén. Baserat på deras kommentarer tas sedan ett beslut om artikels ska accepteras eller refuseras. Endast mindre ändringar av artikeln kan normalt sett ske efter att den är accepterad. De artiklar som accepterats samlas i "proceedings" för konferensen. Jämfört med en tidskrift har granskarna i de flesta fall inte lika lång tid på sig att granska artikeln och

författaren har inte samma möjlighet att uppdatera baserat på kommentarerna.

På många konferenser finns det möjlighet att skicka in "short papers", dvs kortare artiklar där kvalitetskraven är lägre. Det kan också hända att artiklar som skickas in som vanliga artiklar accepteras som korta artiklar i stället för att bli refuserade om det ligger på gränsen mellan att bli accepterade och refuserade.

Eftersom det fungerar olika i olika discipliner ska detta ses som ett normalfall för ämnet "computer science". Det kan dock finnas konferenser som inte har lika tydligt granskningsförfarande, vilket då medför att trovärdigheten för artiklarna blir lägre.

- Bokkapitel i samlingsverk: Ibland bestämmer sig forskare för att sammanställa ett samlingsverk inom ett ämne där olika författare skriver olika kapitel. Det kan fungera lite olika, men ofta kommer initiativet från de forskare som blir redaktörer. De tar kontakt med ett förlag om sin idé, dvs ett ämne och en idé om kapitel, och får den godkänd. Efter det bjuder de in olika författare att skriva de kapitel de har tänkt sig. De som är redaktörer bjuder även in granskare till de kapitel som ska ingå, det finns alltså en tydlig granskningsprocess med utsedda granskare till samlingsverk. Redaktörer till samlingsverk är ofta ansedda forskare inom ett område och kapitelförfattarna är utsedda av redaktörerna och även de ofta erkända forskare. Innehållet i denna typ av kapitel är ofta mer sammanfattande än traditionella forskningsartiklar och kan därmed vara bra att läsa om man vill sätta sig in i ett nytt ämne.
- Workshopartikel: En workshop fungerar ungefär som en konferens, dvs att forskare skickar in artiklar som granskas av en utsedd programkommitté. Kraven på att resultaten ska vara färdiga och väl underbyggda är dock något lägre än för konferenser. Man kan säga att det är lite nyare material som publiceras och presenteras på en workshop, men att det då är något mindre färdigt. Vid vissa stora konferenser anordnas det workshops i samband med konferenserna. Precis som på konferenser så publiceras ibland "short papers" på workshops.
- Poster med artikel: Vid vissa konferenser finns det möjlighet att presentera en poster med resultat. Dessa presenteras på konferensen och i det kommer ofta med en kort artikel i proceedings i för konferensen. Rent praktiskt så har forskaren med sig sin poster till konferensen och vid vissa tidpunkter i programmet förväntas han eller hon stå vid sin poster och diskutera innehållet med andra deltagare på konferensen.

Man kan säga att innehållet i de olika sorternas publikationer är olika moget. Som forskare kan man t ex börja att presentera helt nya idéer med en poster, sedan när man gjort mer forskning kan man presentera en konferensartikel och efter det när man gjort ännu mer forskning presentera resultatet i en tidskriftsartikel. Det är alltså generellt sett större krav på en tidskriftsartikel än på en konferensartikel osv., även om att det kan vara stora skillnader mellan kvalitetskraven för olika tidskrifter och olika konferenser. Det är normalt också mer meriterande för forskare att publicera t ex en tidskriftsartikel än en konferensartikel.

3.3 Sökning efter vetenskapliga artiklar

3.3.1 Databaser och förlag

Man kan söka efter vetenskapliga artiklar, dvs de typer av artiklar som presenteras i kapitel 3.2 på flera olika sätt. Ett sätt är att gå igenom innehållsförteckningen till tidskrifter och böcker från konferenser ("proceedings"). Detta är dock mycket tidsödande och det ovanligt att någon gör så.

För att underlätta för forskare så finns det ett antal databaser som samlar information om alla artiklar som publiceras. Dels finns det databaser från de förlag som publicerar tidskrifter och proceedings, t ex IEEE¹, ACM², Springer³ och Elsevier och dels finns det referensdatabaser som samlar information från flera olika förlag. Exempel på databaser som innehåller artiklar från flera förlag är Scopus⁴ och Web of Science Core Collection (WoS)⁵. Referensdatabaserna innehåller inte själva artiklarna som förlagens databaser gör, utan bara information om artiklar som t ex länkar till förlagens webbsidor om artiklarna.

Det ska poängteras att det inte är gratis för vem som helst att få tillgång till vetenskapliga artiklar eller att använda alla sökdatabaser som finns. Förlagen tar betalt för de artiklar de publicerar i tidskrifter och proceedings. De som tillhandahåller sökdatabaser för flera förlag tar också betalt för att man ska kunna använda dem.

För att forskare och studenter ska ha tillgång till artiklar så betalar de flesta universitet licensavgifter till flera av förlagen vilket innebär att man inte måste betala för varje enskild artikel som student eller anställd på universitetet. Det finns dock exempel på artiklar som inte omfattas av dessa avtal och som man därmed måste betala för om man vill ha. På Lunds universitet är det universitetsbiblioteket⁶ som har hand om licenserna. Via bibliotekets hemsida kan man också söka efter artiklar och böcker och det går att få reda på vilka databaser man har tillgång till via universitetet.

Rent praktiskt sköts kontroll av accessrättigheter oftast baserat på IP-nummer. Det betyder att om man sitter vid en dator på universitet och försöker hämta en artikel från ett förlag så går det bra. Ofta finns det en länk som ger tillgång till artikeln i pdf-format på förlagets sida som presenterar artikeln. Om man däremot sitter hemma så får man inte tillgång till artiklarna på samma sätt. I de flesta fall får man då istället en möjlighet att köpa artikeln. Om man sitter hemma och vill komma åt artiklarna så kan man ansluta sig till universitetet via VPN (med sin student-identitet) och därmed få tillgång till dem.

Det finns även en del artiklar som publiceras "open access", dvs gratis. För dessa artiklar har författarna i stället betalat för att få publicerat artiklarna. Många forskningsanslagsgivare, både i Sverige och i resten av världen, kräver idag att forskningsresultat ska presenteras "open access" och på så sätt vara tillgängliga för alla.

Det går också att söka med traditionella sökverktyg som t ex Google. Google och "Google Scholar"⁷ som är gjort för sökningar på akademiska artiklar är

¹<http://ieeexplore.ieee.org>

²<http://dl.acm.org>

³<http://link.springer.com>

⁴<http://www.scopus.com>

⁵<http://apps.webofknowledge.com>

⁶<http://www.lub.lu.se>, se LUBsearch

⁷<https://scholar.google.se>

förhållandevis bra på att hitta vetenskapliga artiklar och man får ofta länkar till förlagens sidor som presenterar artiklarna. En fördel med "Google Scholar" är att den är förhållandevis enkel att använda och att den även hittar artiklar som ligger utlagda på internet i pdf-format utanför förlagens databaser. Nackdelen är att de inte nödvändigtvis tar med alla artiklar och att sökresultaten ofta innehåller en stor mängd andra träffar som inte är relevanta. Den har inte heller så avancerade sökfunktioner som de traditionella databaserna och de är inte öppna med hur de samlar in och behandlar bibliografisk data från internet [6]. En annan fritt tillgänglig databas är DBLP⁸ som sköts av universitetet i Trier, Tyskland. Den går att använda för att söka på akademiska artiklar och författare särskilt inom området "computer science".

3.3.2 Hur man söker i artikeldatabaser

Även om det skiljer i detaljer hur man söker i olika databaser så är det på det store hela ändå ganska lika. Man brukar kunna välja mellan ett par olika sätt att söka, t ex "enkel sökning" där man helt enkelt matar in de ord man vill söka efter och "avancerad sökning" där man kan kombinera flera sökord och specificera tydligare i vilka fält man ska söka.

I många databaser kan man dessutom skriva söksträngar som man matar in i fritextfält. Det ger möjlighet att på ett tydligt sätt formulera exakt vilka söktermer man ska använda, hur de ska kombineras med logiska uttryck, samt i vilka fält man ska söka. Exakt hur dessa frågor ser ut beror lite på vilken databas man söker i, men i grunden kan man ange sökord och vilka fält man vill leta i och kombinera ihop sökorden med logiska villkor som "and" och "or". Om man t ex använder databasen IEEE-Explore och söker efter artiklar som skrivits av författaren Basili och som handlar om antingen "Experience Factory" eller GQM så kan man använda följande söksträng:

```
"Authors":Basili
AND
("Abstract":Experience Factory OR "Abstract":GQM)
```

Här söker vi alltså efter artiklar som skrivits av en viss författare och som i sammanfattningen (abstract) innehåller minst en av termerna "Experience Factory" och "GQM". Med denna sökning hittar man alla artiklar som uppfyller villkoret hos IEEE:s förlag.

Ett annat exempel på en söksträng, från en "systematisk litteraturstudie" [16], är

```
((open?source} wn ALL) OR
(opensource wn ALL) OR
(libre wn ALL) OR
(OSS wn ALL) OR
(FLOSS wn ALL))
AND
((proprietary wn ALL) OR
(commercial wn ALL) OR
(non?open?source} wn ALL) OR
(non?opensource} wn ALL))
AND
```

⁸<http://dblp.uni-trier.de>

```
((empirical* wn ALL) OR
(experiment* wn ALL) OR
({case?study} wn ALL) OR
(survey wn ALL))
```

Här har man sökt efter artiklar som handlar om empirisk data från användandet av öppen källkod inom kommersiell utveckling, dvs artiklar som innehåller något om öppen källkod, kommersiell utveckling och empirisk data. Sökningen är gjord i en annan databas så formen är inte exakt samma som hos IEEE, t ex så står inte fältet man söker i innan termen, utan efter termen med ett "wn". Man använder också "wildcards" som betyder ett (?) eller ett obegränsat antal (*) tecken, vilka som. Här har man sökt i alla fält, dvs "ALL".

Exemplet ovan är ett exempel på att sökningar, när man söker efter relaterade arbeten till sin forskning, ofta blir på formen

$$\bigwedge_i (\bigvee_j t_{ij})$$

dvs ett antal ämnen med **AND** mellan sig ($\bigwedge_i$) där varje ämne i innehåller ett antal synonymer eller liknande termer ($t_{i1}, t_{i2}, \dots$) med **OR** mellan sig ($\bigvee_j$). Man söker oftast inte bara på ett ämne, som t ex "öppen källkod". Det skulle ge väldigt många träffar och förmodligen är man intresserad av artiklar som handlar om mer än ett ämne, dvs en kombination av olika ämnen. I varje ämne finns det inte bara ett ord som används av författare, utan man måste leta efter flera ord för varje ämne, dvs synonymer och liknande termer.

Man brukar även kunna ange andra logiska villkor, som att datum ska vara senare än ett visst värde, att någon term inte ska finnas med etc. Det ska dock påpekas att man inte kan var hur precis som helst i sina sökningar. Kitchenham et al. [25, kap 25.5.2.2] föreslår t ex att man i litteraturstudier ska använda förhållandevis enkla söksträngar eftersom terminologin i ämnet programvaruutveckling inte är så väl utvecklad att det är möjligt att hitta en precis söksträng som hittar all relevant litteratur utan att också identifiera en stor mängd irrelevanta artiklar. Det är alltså en balansgång mellan att snäva in söksträngen och då missa relevanta artiklar och att ha en bredare söksträng och hitta stora mängder irrelevanta artiklar.

Ofta provar man sig fram lite med exakt vilken söksträng man ska använda. När man hittar några artiklar kan man hitta synonymer i t ex sammanfattningarna och man kan på så sätt få mer kunskap om hur söksträngen ska se ut. När man hittat en bra söksträng kan det vara en bra idé att spara den till senare tillfällen då man kanske vill göra om sökningen eller om man vill göra en liknande sökning i en annan databas.

3.3.3 Alternativ till artikeldatabaser

Det finns alternativ, eller kanske snarare komplement, till att söka i databaser för att hitta vetenskapliga artiklar. När man läser artiklar är det också naturligt att studera vilka referenser de har och även läsa de artiklar som man tror är intressanta av dem. I de flesta artiklar finns ju ett avsnitt med relaterad forskning, där det ofta finns bra referenser att läsa vidare i. På så sätt hittar man nya artiklar baserat på de man läser, se t ex [38]. Ofta kan man identifiera viktiga artiklar inom området som många artiklar refererar till. Då kan det vara extra viktigt att läsa och referera dessa artiklar.

När man identifierar artiklar genom att studera referenser i artiklar ska man vara medveten om att man kommer längre och längre bak i tiden ju längre ”kedjan” av artiklar blir. Av naturliga skäl så refererar ju en artikel aldrig till senare artiklar.

Ofta kan man identifiera viktiga författare från de artiklar man hittar. När man ser att flera av de artiklar som refereras är skrivna av en eller ett par författare så kan det vara värt att söka efter fler artiklar av dessa författare.

Ett ytterligare alternativ är att leta baserat på relevanta forum. Om man t ex ser att en viss konferens innehåller relevanta artiklar kan det vara värt att titta igenom flera årgångar av konferensen för att se om det finns fler relevanta artiklar.

Ibland gör man också sökningar på vilka artiklar som refererar de intressanta artiklar man hittat. Det kräver dock att man kan söka på det hållet.

3.3.4 Vilka databaser ska man välja?

Det går såklart inte att ge något entydigt enkelt svar på vilken eller vilka databaser man ska använda. Det beror lite på vilken ambitionsnivå man har och i många studier räcker det att söka i en eller två databaser, särskilt om man kombinerar sina sökningar med att följa referenser. Alla referensdatabaser är förhållandevis bra, men alla artiklar finns inte i alla databaser. Cavacini [6] undersökte 2014 hur god täckning de olika databaserna har inom ”computer science” genom att prova att leta efter ett urval av artiklar. Han valde först 240 artiklar från 2011–2013 från sitt universitet och letade sedan efter dessa i databaserna Scopus, WoS, DBLP och INSPEC. Han såg att 34,45% av artiklarna fanns i alla databaserna medan de andra artiklarna inte fanns i alla databaserna. 12,95% av artiklarna fanns inte i någon databas. Den databas som innehöll flest artiklar var Scopus (75,86%) och den som innehöll minst antal av artiklarna var INSPEC (53,39%). Den databas som innehöll flest unika artiklar var DBLP, där 4,14% av artiklarna fanns endast i DBLP.

Det finns alltså skillnader mellan databaserna och om man verkligen vill göra så uttömmande sökningar som möjligt bör man söka i flera databaser. Man hittar dock många relevanta artiklar även om man nöjer sig med en referensdatabas.

3.4 Andra källor än traditionellt vetenskapliga

Det finns såklart andra källor än vetenskapliga artiklar man kan ha nytta av att läsa och att referera. Förutom vetenskapliga artiklar enligt kapitel 3.2 så finns det ett antal andra typer av källor som man ofta använder sig av, t ex:

- Läroböcker och andra fackböcker.
- Examensarbeten, licentiatavhandlingar och doktorsavhandlingar
- Tekniska rapporter från universitet eller företag
- Artiklar från facktidskrifter
- ”White papers”, vilka ofta kan ses som marknadsföringsmaterial
- Webbsidor

Här är det viktigt att reda ut källans trovärdighet, samt arbetets kvalitet och relevans för sammanhanget. Läroböcker och andra fackböcker är ofta faktagranskade och skrivna av personer som många har stort förtroende för inom ämnet. Det är vanligt att man refererar till böcker i vetenskapliga rapporter.

Licentiatavhandlingar och doktorsavhandlingar bygger ofta på arbete som även presenteras i vetenskapliga artiklar. Om man vill referera denna information kan man överväga att läsa och referera till de enskilda vetenskapliga artiklar som presenteras i avhandlingen. Arbetet som presenteras i avhandlingar är vetenskapligt granskat både genom granskningen av artiklar och examinationen av själva avhandlingen.

Examensarbeten betraktas inte som lika mogna som traditionella vetenskapliga artiklar, men det är ganska vanligt och vedertaget att referera till examensarbeten.

När det gäller tekniska rapporter kan de t ex innehålla specifik information som inte är av vetenskaplig karaktär eller material som av andra anledningar inte lämpar sig för vetenskaplig publikation. Man får ofta själv som läsare göra en bedömning av trovärdighet och validitet.

Ogranskade webbsidor är generellt sett inte att rekommendera som grund för vetenskapliga arbeten. Det finns givetvis mycket material som är bra, men det finns också mycket material av lägre kvalitet. Om man använder sig av material från ogranskade sidor måste man på något sätt försäkra sig om att kvalitén på materialet är bra nog. Wikipedia är ett öppet uppslagsverk på internet som innehåller en stor mängd information. Denna information har inte granskats i traditionell mening utan vem som helst får lägga till vilken information som helst. Det betyder att många av de mer kända termerna har förhållandevis välskrivna beskrivningar, inte minst eftersom det finns så många läsare som skulle reagera om någon skrev något felaktigt. För mindre kända termer kan man inte räkna med att lika många personer läser och därmed ”granskar” materialet. Ett vanligt angreppssätt är att inte bara läsa innehållet på Wikipedia, utan även studera de referenser som finns där för att hitta mer beständiga källor.

Kapitel 4

Forskningsmetodik

4.1 Inledning

När man pratar om vetenskapliga metoder, eller så kallade forskningsmetoder, så kan man mena olika saker. Grundtanken är att man vill säkerställa att de undersökningar och utredningar man gör bygger vidare på befintlig kunskap och att man kan lita på resultaten. Detta kan man nå på flera olika sätt och Glass har definierat ett antal olika sätt att forska inom ”software engineering” [13]:

- den vetenskapliga metoden (eng. *scientific*) där omvärlden studeras och man bygger en modell av den som kan studeras. Det kan till exempel vara en simuleringsmodell som modellerar de aspekter av verkligheten som man är intresserad av.
- den ingenjörsmässiga metoden där nuvarande lösningar studeras och man föreslår och utvärderar förbättringar. Fokus är alltså på att förbättra en nuvarande situation, men med en tydlig validering så att man vet att det verkligen blir förbättringar.
- den empiriska metoden där en modell föreslås och omvärlden utvärderas utifrån denna i t ex experiment och fallstudier. Fokus är alltså på att empiriska undersökningar.
- den analytiska metoden där en formell teori föreslås och utvärderas med empirisk data. Fokus är alltså på att analytiskt härleda modeller, men man kan utvärdera dem med empirisk data.

Orsaken till att man använder flera metoder inom ”software engineering” är att området är brett och det finns flera olika sorters frågor. I vissa situationer är man kanske intresserad av att göra prestandaanalyser av en systemarkitektur som är under utveckling och ännu inte finns och då kanske det är lämpligt att använda en simuleringsmodell. I andra fall kanske man vill göra förbättringar av t ex en mjukvara och då kan ett typiskt ingenjörsmässigt angreppssätt enligt ovan är mest lämpligt. Det kan också finnas situationer då man t ex vill utvärdera en ny utvecklingsprocess i ett projekt och då kanske man ser att en fallstudie är det bästa sättet.

Det finns fyra vanliga metoder inom tillämpade vetenskapsområden och empirisk forskning [17]:

- Kartläggning (eng. survey)
- Experiment, eller även kallat "kontrollerat experiment" (eng. controlled experiment)
- Fallstudie (eng. case study)
- Aktionsforskning (eng. action research)

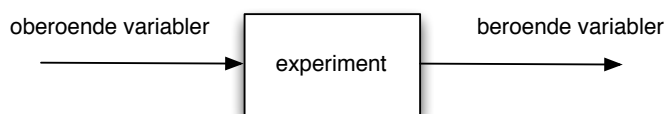
Dessa förklaras närmare nedan. Något som är gemensamt för alla metoderna är att man bygger på befintlig kunskap i sin undersökning och att man redogör för hur man kommer fram till sina slutsatser så att en läsare kan förstå hur tillförlitliga resultaten är. Detta möjliggör sedan att en läsare kan bedöma kvaliteten och giltigheten av resultaten, samt att det är möjligt för andra att replikera undersökningen, dvs genomföra en liknande studie och kontrollera om man får samma resultat.

Metodik är alltså det grundläggande arbetssätt man väljer för sitt projekt. Metodiken sätter upp ramarna för hur man går till väga, men den beskriver inte i detalj exakt vad och hur man ska göra i varje steg, eller exakt när varje steg är planerat i tiden. Metodiken är snarare en hjälp till att komma från en övergripande målsättning till ett svar på frågorna man har.

För att genomföra sin studie måste man välja en lämplig vetenskaplig metod, eller kombinationer av sådana. Utifrån dessa gör man sedan en konkret plan för sitt undersökningsarbete. Detta diskuteras närmare i kapitel 4.6 om design av studier. För olika metoder kan man använda olika slags "verktyg" för t ex datainsamling och analys. I projektet i denna kursen (EDAA35) kommer mättekniken i stor utsträckning att handla om att mäta exekveringstid av java-program.

Man pratar ofta om skillnaden mellan *kvantitativa* och *kvalitativa* studier. Då menar man att man i kvantitativa studier baserar analysen på kvantitativ data som man mäter under studien, t ex mätning av tid, antal fel, komplexitet och liknande. Då man pratar om kvalitativa studier menar man att man inte har kvantitativ data utan kvalitativ data, dvs data i form av t ex intervjuer som man skriver ner efter intervjuer och analyserar. Ofta förknippar man kartläggningar och experiment med kvantitativ data och fallstudier och aktionsforskning med kvalitativ data. Man kan också skilja mellan fix och flexibel design. I en fix design planerar man hela studien från början och har ganska liten möjlighet att ändra efter hand. Detta är ganska typiskt för kvantitativ forskning där man inte vill ändra eftersom det gör att man inte kan räkna alla datapunkter t ex i ett experiment eller en kartläggningsstudie. Med en flexibel design har man möjlighet att ändra och tekniken går ut på att man ska upptäcka nya saker och förhålla sig till dem under forskningens gång. Detta är ganska typiskt för kvalitativ analys. Det är dock inget som hindrar att man använder kvalitativ data i en fix design eller kvantitativ data i en flexibel design.

De metoder som nämns ovan är kanske lättast att förstå om man jämför undersökningar där människor är inblandade. Om man t ex studerar hur man gör testning av java-program och om man vill säga något om hur bra olika metoder är i olika sammanhang kan man genomföra olika sorters studier med olika mål. Man skulle t ex kunna genomföra en kartläggningsstudie för att ta reda på vilka metoder som används i industrin. Man skulle efter det t ex kunna göra ett experiment för att förstå vilken metod som hittar flest fel för en viss



Figur 4.1: Oberoende och beroende variabler i ett experiment

typ av kod. När man sedan vet det så skulle man kunna göra en fallstudie där man undersöker mer i detalj varför den ena metoden hittar fler fel än den andra och t ex vad det innebär för resten av utvecklingen på företag där den används. I fallstudien skulle man kunna kombinera observationer av vad som händer i testningen med intervjuer av de som utför själva testningen.

Fokus i detta kompendium är på utvärderingar och den metod som ligger närmast är förmodligen kontrollerade experiment, inte minst eftersom denna metod generellt sett används i både studier med människor och mer tekniska studier. Montgomery [29] fokuserar t ex helt på tekniska experiment, som att jämföra hållfasthet hos olika material. Nedan följer korta beskrivningar av några av metoderna.

4.2 Experiment

Termen ”experiment” används ganska allmänt om undersökningar i största allmänhet. När man preciserar och använder termen ”kontrollerat experiment” så menar man dock en jämförande studie där man tar reda på effekten av ett antal variabler på ett antal andra variabler, genom en isolerad studie där man försöker ta bort effekterna av alla andra variabler.

4.2.1 Den grundläggande idén med experiment

I ett experiment vill man alltså undersöka effekten av en eller flera andra variabler på en eller flera andra variabler. De variabler som man vill undersöka effekten av kallas *oberoende variabler* och de variabler som man undersöker effekten på kallas *beroende variabler*, se figur 4.1. Ibland kallas även oberoende variabler för *faktorer*.

Ett exempel på oberoende variabel i ett tekniskt experiment kan t ex vara valet av kompilator och ett exempel på en beroende variabel kan t ex vara tiden det tar att exekvera programmet som kompilerats med kompilatorn.

Ett lite mindre tekniskt exempel på oberoende variabel kan vara granskningsteknik för att granska koden för att hitta fel i den. Denna variabel kan i detta exempel ha två nivåer, antingen genom att göra sitt bästa baserat på sin erfarenhet (dvs en *ad-hoc*-teknik) eller en teknik där de som granskar får en checklista som de ska använda under granskningen. I detta exempel har alltså den oberoende variabeln två nivåer (*ad-hoc* och checklista). Beroende variabler kan t ex vara antalet fel som varje granskare hittar och tiden det tar för varje granskare. Själva experimentet kan sedan t ex genomföras genom att 20 personer delas in i två grupper och ena hälften gör individuell *ad-hoc*-granskning och andra hälften gör individuell granskning med checklista. Det betyder att man

Tabell 4.1: Exempel på experiment

Person	Oberoende variabel	Beroende variabel 1	Beroende variabel 2
Person 1	ad-hoc	antal fel person 1	tid person 1
Person 2	ad-hoc	antal fel person 2	tid person 2
...	...	...	...
Person 10	ad-hoc	antal fel person 10	tid person 10
Person 11	checklista	antal fel person 11	tid person 11
Person 12	checklista	antal fel person 12	tid person 12
...	...	...	...
Person 20	checklista	antal fel person 20	tid person 20

skulle få data enligt tabell 4.1. I analysen kan man sedan jämföra medelvärdena för antal fel och tiden det tar för de två olika metoderna. Då finns det statistiska metoder för att ta reda på om en eventuell skillnad verkar bero på en verklig underliggande skillnad eller bara beror på slumpmässiga variationer.

Idén med ett experiment är att man ska ändra enligt oberoende variabler, men allt annat ska vara så lika som möjligt. I exemplet ovan ska t ex erfarenheten hos de som utför granskningen vara så lika som möjligt, granskningen ska ske av samma program med samma fel i för alla och granskningen ska ske under så lika förutsättningar som möjligt. Man gör alltså experimentet i en "laboratoriemiljö" där man ändrar bara det som man vill ändra och man kan hålla så många andra faktorer som möjligt så lika som möjligt.

Experiment bygger på *randomisering*, dvs att slumpen tar ut effekten av individuella skillnader. I exemplet ovan sa vi att personerna skulle ha samma erfarenheter, vilket i praktiken givetvis är omöjligt. Personer har inte exakt samma erfarenheter, men om man väljer de två grupperna slumpvis så kommer de individuella skillnaderna att tas ut av att grupperna i genomsnitt har samma erfarenhet av att granska kod. Den som genomför och planerar experimentet måste planera så att man har med tillräckligt många personer i experimentet för att man ska få tillräckligt många datapunkter för att skillnaderna ska ta ut varandra. Detta diskuteras ytterligare i kapitel 6.

4.2.2 Olika typer av experiment

Det exempelexperiment som presenterades ovan är mycket enkelt och det går givetvis att definiera mer avancerade experiment. Med en oberoende variabel kan man t ex undersöka mer än två nivåer. Man skulle t ex kunna jämföra tre olika granskningstekniker. Man skulle också kunna undersöka mer än en oberoende variabel. T ex skulle man även kunna ändra vilket programspråk som granskas. Då skulle man ha två olika oberoende variabler, granskningsteknik (t ex med alternativen ad-hoc och checklista) och programspråk (t ex med alternativen Java och C++).

I experiment där människor ska delta, t ex i granskningsexperimentet som beskrivits här är det ofta svårt att hitta tillräckligt många människor för att man ska kunna dra säkra slutsatser. Man kan då ibland låta varje person utsättas för mer än en behandling. I exemplet med en oberoende variabel med nivåerna ad-hoc och checklista skulle varje person kunna göra två granskningar, en med varje metod. Givetvis måste man då tänka på att personerna inte granskar

samma kod två gånger, eftersom de då lär sig vilka fel som finns. I stället kan de arbeta med två likvärdiga program. Man brukar också rekommendera att hälften av personerna först använder den ena tekniken och den andra hälften först använder den andra tekniken, för att ta bort effekten av ordningen. Det senare kan vara viktigt eftersom man ofta kan tänka att de som deltar tröttnar och gör ett bättre jobb i den första granskningen än i den andra.

Ju mer avancerad experiment-design man väljer desto mer avancerad analys måste man göra. Ett experiment med en oberoende variabel med två nivåer och en beroende variabel är förhållandevis enkelt att analysera, medan ett experiment med flera oberoende och beroende variabler där de som deltar gör mer än en sak kan vara förhållandevis krävande att analysera och ställa krav på mer kunskap om statistiska metoder.

Några vanliga experimenttyper är (t ex [40, 39]):

- En oberoende variabel med två nivåer, varje person gör en sak
- En oberoende variabel med två nivåer, varje person gör båda sakerna
- En oberoende variabel med mer än två nivåer
- Mer än en oberoende variabel

Hur dessa analyseras diskuteras närmare i kapitel 6.

4.3 Kartläggning

Kartläggningar (eng. survey) genomförs ofta för att uttala sig om en grupp personers åsikter, för att ta reda på hur en grupp personer genomför en uppgift, eller liknande. Idén är att man först identifierar målgruppen (eng. target population) som man vill uttala sig om och sedan bestämmer vilken representativ delmängd (eng. sample) man frågar för att kunna uttala sig om målgruppen [24, 34].

Exempelvis skulle man kunna genomföra en kartläggningsstudie för att ta reda på vilka testmetoder som används i Sverige. Målgruppen skulle i så fall vara alla som arbetar med testning i Sverige. Men skulle så klart kunna tänka sig att man frågade alla dessa personer hur de genomför testning, men det skulle förmodligen bli för arbetskrävande. Man måste därför välja ett representativt urval.

Data samlas in genom att det urval man valt svarar på ett antal frågor. Ofta sker detta genom ett webb-formulär eller ett frågeformulär på papper, men det kan också genomföras genom intervjuer, t ex över telefon. Inte minst eftersom man får in ganska många svar vill man ha slutna frågor med svarsalternativ, vilket är förhållandevis lätt att analysera jämfört med fritextsvar.

Validiteten av en kartläggningsstudie påverkas både av hur väl man lyckats formulera frågorna och hur väl man lyckats sampla målgruppen. När det gäller frågorna gör man som sagt ofta ganska korta frågor med svarsalternativ, vilket gör att det finns en risk att de som svarar inte riktigt förstår vad de handlar om, vilket kan vara fallet om frågorna inte är mycket väl utarbetade. Ofta försöker man prova frågorna på en kontrollgrupp för att hitta eventuella problem med dem innan man skickar ut dem till alla som man vill ha svar av. Det är också vanligt, i alla fall inom områden där denna typ av forskning är vanlig, att man

bygger på redan använda och utprovade frågeuppsättningar när man tar fram sina frågor [24].

När det gäller hur väl man lyckas sampla målgruppen så beror detta både på hur väl man lyckas välja ut sampel och på svarsfrekvensen. Man väljer ofta ut vem man ska fråga genom att göra ett slumpvis val av personer. Eventuellt kan man dela in målgruppen i olika grupper och välja slumpvis ur varje grupp för att försäkra sig om att alla grupper verkligen kommer med i urvalet, om det finns en naturlig gruppindelning och man tror att de kommer att svara olika. Ibland kan man även göra ett urval baserat på vilka personer som är tillgängliga (eng. convenience sample). Det ger inte samma validitet, men kan t ex användas i en testomgång om man provar frågorna innan man skickar ut dem till den riktiga gruppen.

Ett exempel på en kartläggningsstudie presenteras av Neill m.fl. [30]. De ville ta reda på hur kravhantering fungerar i industrin och ställde frågor t ex om hur man gör prototyper, hur man tar reda på kraven för en produkt och i vilken utsträckning formella analysmetoder används. De gjorde sitt urval av personer genom att skicka ut frågor till "a database of prospective, current, and past graduate students of the Penn State Great Valley School of Graduate Professional Studies". Samplingsstrategin är alltså en typ av "convenience sample", men de har redogjort för domän, företag, etc. för de som svarade så det finns viss möjlighet att avgöra till vilken målgrupp resultaten kan generaliseras.

4.4 Fallstudie

Yin [41] definierar en fallstudie enligt

"A case study is an empirical enquiry that

- investigates a contemporary phenomenon within its real-life context, especially when
- the boundaries between phenomenon and context are not clearly evident."

Denna definition visar på två viktiga saker med fallstudier, dels att det ofta är svårt att skilja en företeelse från sin omgivning, och dels att man i en fallstudie accepterar och bygger på det och därför studerar företeelsen i sin omgivning. Det betyder att man, istället för att man som i ett experiment försöker studera en företeelse utan att omgivningen får påverka, studerar en företeelse tillsammans med sin omgivning. Om man t ex vill studera hur man arbetar med testning så studerar man ett verkligt fall och t ex försöker förstå varför man testar som man gör under de förutsättningar man har i det fallet. Det betyder att man inte får någon möjlighet att säga något baserat på medelvärde av flera olika fall, utan man drar sina slutsatser genom att förstå det fall man studerar på djupet.

I en fallstudie försöker man ofta samla in data från flera olika håll för att kunna lita på sina slutsatser. Vanliga exempel på datakällor är t ex intervjuer, observationer, resultat från fokusgrupper, samt data som redan finns tillgänglig i rapporter, protokoll och databaser. Det rör sig alltså ofta om kvalitativ data, men det går också att använda kvantitativ data.

Mer information om fallstudier i allmänhet finns t ex i [41] och [34] och mer information om fallstudier om mjukvaruutveckling finns t ex i [35].

Tabell 4.2: Jämförelse av forskningsmetoder (baserat på [40])

Faktor	Kartläggning	Fallstudie	Experiment
Kontroll, genomförande	Nej	Nej	Ja
Kontroll, mätning	Nej	Ja	Ja
Kostnad	Låg	Medium	Hög

4.5 Aktionsforskning

Aktionsforskning har vissa likheter med fallstudier. Båda angreppssätten studerar en händelse i sin kontext och kan bygga på både kvalitativ och kvantitativ data. Man kan dock säga att aktionsforskning har ett tydligare fokus på problemlösning, dvs att man utgår från ett problem som man försöker lösa och på så sätt förbättra något som finns idag. Forskaren balanserar alltså mellan att både hjälpa till att förbättra något och på samma gång studera i vilken utsträckning man verkligen får förbättringar. Detta är en förhållandevis vanlig situation för en ingenjör.

4.6 Design av forskningsstudie

4.6.1 Val av forskningsmetod

Som en del av en forskningsstudie eller en annan typ av undersökning måste man bestämma sig för vilken forskningsmetodik man ska använda, samt planera mer i detalj hur den ska gå till. Detta kan man kalla för att man ”designar” sin studie. Den forskningsfråga man har påverkar givetvis i stor utsträckning vilken metod man ska använda, men även vilken tillgång till data man har.

En jämförelse av metoder finns i tabell 4.2. Denna fokuserar på vilken kontroll man har vid genomförandet, vilken kontroll man har när det gäller hur mätningar ska göras och vilken kostnaden är.

När det gäller kontrollen vid genomförandet betyder detta i vilken utsträckning forskaren kan bestämma vem som ska göra vad av deltagarna och när och hur detta ska göras. Ett experiment kan ses som ett ”laboratorieexperiment” och forskaren har stora möjligheter att bestämma vad som ska göras, det är ju i stort sett grundidén med ett experiment. Man har som forskare inte alls samma möjlighet att påverka i en fallstudie eller en kartläggning. Här måste forskaren rätta sig efter den verklighet som undersökningen görs i. Nackdelen är att det ibland är svårare, men fördelen är att undersökningen därmed ”automatiskt” genomförs under realistiska förhållanden.

Kontroll vid mätning avser i vilken mån forskaren kan påverka vilka mätningar som görs och hur och när de mäts. I en fallstudie och ett experiment har forskaren förhållandevis stor möjlighet att bestämma vad som ska mätas. I en fallstudie har man dessutom stor möjlighet att ändra efter hand som man ser behov av nya mätningar. Orsaken till att vi säger att det är låg kontroll för kartläggningar är dels att vi inte kan veta exakt när och hur mätningarna görs och dels att det inte går att ändra vilka mätningar man gör när undersökningen väl börjat. Forskaren kan (och ska) dock bestämma vilka frågor man ställer.

Kostnaden är en uppskattning av hur mycket resurser som krävs, både av

forskaren och deltagarna. I detta fall är kartläggningen av lägst kostnad eftersom det rör sig endast om att fylla i en enkät, vilket i alla fall förmodligen ger lägst kostnad per deltagare. Kostnaden för experiment är satt som högst eftersom hela studien görs just för forskningens skull. I en fallstudie studerar man ju oftast en företeelse som ändå hade skett.

Dessa jämförelser är ungefärliga och vilket resultat man får beror mycket på hur man definierar faktorerna. Jämförelsen ska alltså inte tas som någon absolut sanning som alltid gäller i alla fall.

4.6.2 Planering av datainsamling och dataanalys

När man bestämt den grundläggande metoden måste man göra en mer utförlig planering av hur studien ska gå till. Man kan t ex bestämma vilken experimentuppställning man ska ha i ett tekniskt experiment, hur man ska göra mätningar etc. I ett experiment där människor är inblandade måste man bestämma vem som ska göra vad, när det ska ske, vilka instruktioner som ska delas ut etc.

Det är också bra att planera hur den data man samlar in ska analyseras. Dels så vet man hur analysen ska gå till och dels så minskar risken att man genomför datainsamling som resulterar i data som är svår att analysera.

4.6.3 Validitet av studien

I samband med att man gör en undersökning är det viktigt att man reflekterar över vilken validitet studien har och försöker få den så bra som möjligt under de förutsättningar man har. Det är också viktigt att man är öppen med vad man gjort för validiteten i studien och vilka eventuella hot man ser mot den.

Man kan klassificera validitetshot på flera olika sätt, se t ex [34, 40], beroende på vilken typ av studie det är. Här väljer vi en enkel variant där vi bara skiljer på två olika sorters hot mot validitet, interna hot och externa hot.

Interna hot kan sägas innefatta allt som påverkar resultatet utan att den som gör undersökningen är medveten om det. Här kan man exempelvis beakta om de mätningar man gör påverkar resultatet på något sätt, vilket då gör att man inte kan lita på resultatet. Om man jämför två saker, som t ex exekveringstiden hos två olika implementationer av samma algoritm, kan man också beakta om de mätts under samma förutsättningar. Ett enkelt exempel är om man har mätt den ena implementationen på en snabbare dator än den andra så är det ett hot mot validiteten.

Om man gör experiment där människor deltar finns det ett antal hot som kan påverka, t ex att den ena gruppen, t ex kontrollgruppen, känner sig som "under dogs" och därmed presterar annorlunda än den andra gruppen, att deltagare tröttnar och därmed kan prestera sämre i slutet av en undersökning än i början. Det finns också en risk att personer i två olika grupper har för olika kompetens och därmed presterar olika bra, särskilt om valet av vem som ska göra vad i ett experiment inte är helt slumpmässigt.

Externa hot handlar om hot mot *generaliserbarheten*. Om man i en studie har kommit fram till en sak så är det naturligt att ställa sig frågan om man hade kommit fram till samma sak i en annan situation där man

undersökt samma sak. Om man t ex jämför exekveringstiden hos två olika implementeringar av samma algoritm så kan resultatet påverkas av vilken indata man har. Man kan då diskutera vilket resultat man fått med annan indata och vilken indata som förmodligen ger samma resultat.

Generellt sett kan man studera extern validitet i experiment som involverar människor t ex utifrån

- vilka personer som var med i studien, dvs hur representativa är personerna för den population som man vill uttala sig om,
- vilket material som användes i studien, dvs hur representativa är materialet som man gör studien med, jämfört med ”verkligt” materia. Om man t ex vill uttala sig om hur bra en metod är så är det lämpligt att prova den på ett riktigt system och inte ett för enkelt system
- vilken historisk erfarenhet personer hade i studien när de gjorde studien.

De två typerna av hot ovan kan alltså användas som en ”checklista” för att definiera och lösa validitetshot i designfasen av en studie. Dessutom kan den användas för att diskutera validitetshot och deras lösningar i den rapport man skriver som resultatet av en studie.

Kapitel 5

Mätningar av programvarusystem (metrics)

5.1 Inledning

När man gör utvärderingar av programvarusystem är det givetvis viktigt att kunna göra mätningar på dessa system, eller som DeMarco uttrycker det

”You cannot control what you cannot measure” [9]

Området ”software metrics” har utvecklats till ett eget ämnesområde inom ”software engineering” och det finns flera böcker som fokuserar på detta, t ex [10], [22]. Mätningar är grunden för empiriska studier och utan mätningar är det inte möjligt att förstå t ex hur bra ett system är under dess utveckling. Man kan mäta många olika saker under utvecklingen och man kan mäta på flera olika sätt. Målet med detta kapitel är att reda ut vilka olika alternativ som finns.

Först kan man skilja mellan *mätningar* och *mått*. Mätning är det man gör för att koppla en siffra eller en symbol till ett attribut för en entitet i den verkliga världen. En entitet kan vara ett objekt, som t ex en person eller artefakt (t ex kod), eller en händelse (t ex testprocessen). Man vill beskriva något intressant attribut för entiteten vilket definieras som ett attribut. Exempel på attribut är storlek, produktivitet, och kostnad. Ett mått på storlek för ett kodstycke är t ex antalet kodrader. Vid en mätning räknar man helt enkelt antalet kodrader på något sätt som man definierat.

Ett grundläggande krav på ett definierat mått är att det ska bibehålla ”den empiriska observationen”, t ex att om ett kodstycke A är större än ett kodstycke B, så ska även längdmåttet av kodstycke A vara större än längdmåttet för kodstycke B.

Man kan definiera ett mått för ett attribut på flera olika sätt. Till exempel kan man mäta längd i olika enheter som meter, tum, etc. Varje mappning mellan attribut och mått, dvs sätt att mäta, kallas en *skala*. Det är också möjligt att omvandla mellan olika skalor, t ex från meter till fot. Om en omvandling bibehåller förhållandena mellan objekten så kallas det en *tillåten omvandling* eller tillåten transformation.

5.2 Skalar

De vanligaste skalorna är nominal-, ordinal-, intervall- och ratioskala, som kan förklaras kort enligt följande:

- **Nominalskala:** Detta är en klassificering av attributen för entiteter. Alla omvandlingar som bibehåller att man fortfarande kan göra en "en-till-en mappning" är tillåtna om man vill behålla mätningarna på nominalskala. Exempel på nominalskalor inom programvaruutveckling är t ex felklassificeringar, utvecklartyper, etc. I någon studie skulle man t ex kunna dela upp utvecklare i "front-end utvecklare" och "back-end utvecklare". En tillåten omvandling skulle kunna vara att kalla dem "gränssnittsutvecklare" och "databasutvecklare" i stället.
- **Ordinalskala:** Med denna skala kan man använda de mätvärden man får när man mäter på entiteter för att rangordna entiteter. Man kan därför säga att skalan är mer kraftfull än nominalskalan, eftersom man inte kan göra det med en nominalskala. Exempel på ordningskriterier som kan användas är "större än", "bättre än", "mer komplex", osv. Alla transformationer som bibehåller ordningen är tillåtna, dvs alla $F(M)$ som är strängt växande, och alltså uppfyller $M_1 > M_2 \Rightarrow F(M_1) > F(M_2)$. Ett exempel på ett mått på ordinalskala är bedömningar av kvalitet enligt "låg", "medium" eller "hög". En tillåten transformation skulle kunna vara att i stället för att säga "låg", "medium" eller "hög" så kan man säga 1, 2 och 3. Ett annat exempel kan vara att man klassificerar fel som uppkommer i testning som "triviala", "normala" eller "kritiska".
- **Intervallskala:** Med denna skala är det relevant att jämföra värden och titta på skillnaden mellan värden, men de absoluta värdena är inte lika meningsfulla. Precis som med ordinalskalan går värdena att sortera och man kan också prata om den relativa skillnaden mellan avståndet mellan värden. Man kan t ex säga att skillnaden mellan entitet A och B är dubbelt så stor som skillnaden mellan entitet C och D . Möjliga transformationer är $F(M) = \alpha M + \beta$, där M är ett mått på ett attribut och M' det transformerade måttet. Denna skala är förhållandevis ovanlig vid mätning på programvarusystem, men exempel från dagliga livet är temperatur i grader Celcius eller Fahrenheit.
- **Ratioskala:** Med denna skala är det precis som för intervallskalan relevant att jämföra värden och titta på skillnaden mellan värden och dessutom finns det en nollpunkt som också gäller i den fysiska världen. Det är alltså relevant att säga att t ex värde A är 30% större än värde B . Möjliga transformationer är $F(M) = \alpha M$, där M är ett mått på ett attribut och M' det transformerade måttet. Exempel på mått på ratioskala är exekverinstid och programlängd i rader kod, samt temperatur i grader Kelvin.

Ibland talar man dessutom om absolutskala, vilket är ett specialfall av ratioskalan, när det inte är relevant att göra några transformationer alls, t ex för programlängd i antal rader kod [10].

Tabell 5.1: Sammanfattning av skalor

Skala	Tillåtna transformationer
Nominal	alla som tillåter "en-till-en mappningar"
Ordinal	alla strängt växande funktioner
Intervall	$F(M) = \alpha M + \beta$
Ratio	$F(M) = \alpha M$
Absolut	—

Skalorna sammanfattas i tabell 5.1. Här är det lämpligt att förtydliga vad vi menar med att transformationer är "tillåtna". Med det menar vi att en transformation är tillåten så länge den inte orsakar att vi får en annan, "lägre", skala. Om man t ex har en mätning på ratioskala och transformerar den med $F(M) = \alpha M + \beta$ där $\beta > 0$ så kan man ju inte längre säga t ex att ett värde A är 30% större än värde B .

5.3 Objektiva och subjektiva mått

De mätningar man gör kan vara antingen subjektiva eller objektiva.

Med ett objektivt mått menar vi såklart att det är objektivt, dvs det påverkas inte av hur den som mäter värdet tolkar det. Ett objektivt mått skulle i teorin kunna mätas flera gånger av olika forskare och värden skulle bara skilja sig åt pga det mätfel som sker under mätningen. Exempel på ett objektiva mått är kodstorlek och exekveringstid, eller medeltid för rättning av fel efter att de hittats i testfasen av ett projekt. Objektiva mått är ofta på ratioskala eller nominalskala.

Ett subjektivt mått bygger på människors subjektiva åsikter och bedömningar. Exempel på subjektiva mått är kodkvalitet på en skala "dålig", "acceptabel", eller "bra", samt bedömning av användbarhet efter användningen av ett nytt användargränssnitt. Subjektiva mått är ofta på ordinalskala, t ex efter ett frågeformulär personer fått fylla i om hur bra de tycker en sak är, t ex enligt ovan.

5.4 Direkta och indirekta mått

Mått kan vara direkta eller indirekta. Med direkta mått menar man sådana som man kan mäta direkt och som inte kräver att man mäter något annat. Exempel på direkta mått är antalet rader kod, antalet fel som upptäcks i acceptanstest och arbetstid i kodfasen. Dessa exempel går ju att mäta "enkelt" utan att mäta andra saker.

Indirekta mått är härledda från ett andra mått, antingen direkta eller indirekta. Exempel på indirekta mått är feldensitet, t ex uträknat som antalet upptäckta fel delat med antalet rader kod, och produktivitet hos utvecklare. Dessa går alltså inte att mäta direkt, utan man måste bestämma sig för hur man ska räkna ut dem från andra mått. Ofta kan man välja mellan olika sätt att räkna ut dem, vilket gör att man måste ta beslut om hur man ska mäta.

Tabell 5.2: Exempel på interna och externa mått inom programvaruutveckling

Mätobjekt	Typ	Exempel
Process	Intern	arbetstid
	Extern	kostnad
Produkt	Intern	antal rader kod, antal fel
	Extern	kvalitet, exekveringstid
Resurs	Intern	erfarenhet hos personal
	Extern	produktivitet hos personal

Det är alltså upp till den som gör undersökningen att definiera hur måtten ska räknas ut.

I undersökningar är man ofta intresserad av storheter som mäts som typiska indirekta mått. Man kan ex vara intresserad av storheter som produktivitet och kvalitet. Dessa måste man då formulera i termer av direkta mått som man kan mäta för att få en uppfattning om dem. På samma sätt är det ganska ofta som man inte är väldigt intresserad av enstaka direkta mått. T ex så säger ju inte antalet rader kod så mycket om man endast får veta det eller antalet fel man hittat i granskning av en kodfil. Om man bara får reda på att det finns tre fel i en fil så är det såklart bra att ta bort de felen, men det säger mer om kvaliteten om man jämför med storleken. De flesta skulle nog hålla med om att en fil som är 10 rader med tre fel har sämre kvalitet än en fil med tre fel som är 2000 rader.

Detta är relaterat till en annan uppdelning man gör, mellan interna och externa mått. Interna mått kan mätas direkt på en enhet utan att mäta på någon annan enhet, t ex antalet rader kod. Externa mått innehåller mätningar av hur enheten relaterar till andra enheter. T ex är tillförlitlighet ett externt mått eftersom det inte går att mäta direkt på ett program eftersom man måste ta hänsyn till andra enheter, som vem som använder det och hur de använder det. Det kan t ex ofta vara skillnad på hur en erfaren användare använder ett program och hur en nybörjare använder det.

5.5 Vad man man mäter inom mjukvaruutveckling

Vi har i detta kapitel sett ett antal exempel på mått av på olika saker, t ex kod och processer. Mätningar kan göras på en mängd olika typer av objekt, både på processer, produkter och resurser.

- Process: man kan mäta på den utvecklingsprocess som finns för att utveckla programvara. Exempel på mått är tid och kostnad.
- Produkt: Man kan mäta på de produkter som utvecklas. Man kan t ex mäta antal rader kod och tillförlitlighet.
- Resurs: Man kan mäta på de resurser som finns för att utveckla programvara, t ex personer och verktyg.

I tabell 5.2 visas ett antal exempel på interna och externa mått.

5.6 Målorienterad mätning

Som vi såg i kapitel 5.5 kan man mäta på många olika saker. När man genomför en studie är det därför frestande att mäta på "så mycket som möjligt" för att vara säker på att man inte missar något som kan vara intressant att analysera i framtiden. Även i ett företag kan det vara frestande att samla på sig mätningar för att kunna göra bra analyser i framtiden. Det har dock visat sig att den strategin inte fungerar så bra. Även om man mäter många saker så är det ändå troligt att man missar det man verkligen behöver mäta om man inte tänker efter noga vad det egentligen är man behöver mäta. Dessutom är det svårt att använda data i framtiden om man inte vet exakt under vilka omständigheter de är mätta och det är svårt att använda gammal data eftersom den ofta är mätt under andra omständigheter än de man har när man gör analysen.

Det finns ett antal utmaningar med att göra mätningar. Vi vill t ex att mätningar som görs i olika projekt ska vara jämförbara med varandra, vilket inte alltid är fallet om man inte definierar tydligt vad man mäter varje gång man gör mätningar. Dessutom vill man såklart att de mätningar man gör ska vara tillförlitliga och pålitliga. Till detta kommer att det ofta är svårt att genomföra mätningar om de som verkligen utför mätningarna, eller blir mätta på, inte upplever att mätningarna används till något i organisationen. Några av lösningarna till dessa utmaningar är att bara göra de mätningar som verkligen behövs och att då verkligen definiera noga vad som verkligen ska mätas. På så sätt undviker man stora mängder data som ingen använder och man kan ha resurser att verkligen beskriva och definiera mätningarna, vilket kan vara av nytta i senare mätningar i andra områden av organisationen.

För att lösa dessa problem har man definierat olika sätt att mäta "målorienterat", dvs att man tar fram sina mått baserat på de måtmål man har. Ett exempel på ett målorienterat angreppssätt är *Goal Question Metrics* (GQM) metoden (t ex [4]).

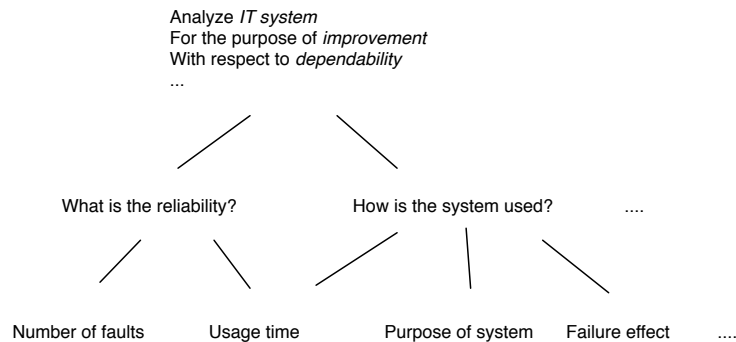
Med GQM härleds mätningar genom att man först definierar mål, sedan frågor som svara på de mål man har och sedan, baserat på det, de detaljerade måtten som ska mätas. För detta har man tagit fram följande "mall":

Analyze	<i>Object(s) of study</i>
for the purpose of	<i>Purpose</i>
with respect to their	<i>Quality focus</i>
from the point of view of the	<i>Perspective</i>
in the context of	<i>Context.</i>

Här ska man själv ska fylla i de kursiverade delarna för sitt måtmål. Baserat på dessa mål kan man sedan lista de frågor man måste ställa sig för att svara på målen och sedan definiera de mått man måste mäta för att svara på frågorna.

De olika delarna kan förklaras enligt följande:

- *Object(s) of study* (objekt): Detta är det objekt man studerar, t ex programkoden. Det kan vara antingen en produkt, en process eller en resurs.
- *Purpose* (målsättning): Detta är målet med att göra mätningarna. Ett mål kan vara att förstå hur något fungerar och skaffa sig en "baseline". Ett annat mål kan vara att förbättra något.



Figur 5.1: Exempel på GQM-modell

- Quality focus (kvalitetsfokus): Detta är det kvalitetsfokus man har, t ex tillförlitlighet, exekveringstid, kostnad, predikterbarhet.
- Perspective (perspektiv): Med detta menas från vems perspektiv mätningarna definieras, t ex utvecklarens, projektledarens, företagsledningens eller forskarens.
- Context (kontext): Detta beskriver i vilket sammanhang mätningarna gjorts, t ex i vilket utvecklingsprojekt och i vilket företag de gjorts.

Ett exempel på ett mål för att mäta exekveringstid kan formuleras enligt följande:

Analysera	<i>appen Tetris</i>
med målsättningen att	<i>förstå</i>
med avseende på	<i>exekveringstid</i>
ur följande synvinklar	<i>normal användare, forskare</i>
i följande sammanhang	<i>enkel telefon i prisklassen X.</i>

Ett ytterligare exempel för IT-system och pålitlighet skulle kunna vara

Analyze	<i>IT system X</i>
for the purpose of	<i>improvement</i>
with respect to their	<i>dependability</i>
from the point of view of the	<i>user</i>
in the context of	<i>business process Y</i>

Detta kan sedan brytas ned i frågor och konkreta mätningar som i figur 5.1. Här är målet överst, under det kommer frågorna och underst kommer de konkreta mätningar. Man ritat det ofta som ett träd, men eftersom vissa mätningar är gemensamma för flera frågor så är det inte ett riktigt träd. Det ligger utanför fokus för detta kompendium att gå igenom GQM i detalj, utan detta ska ses mer som en introduktion till ämnet.

GQM-mallen fungerar i många fall även bra som mall för att definiera målsättningen med empiriska studier och är ofta ett bra hjälpmedel att använda i ett första steg när empiriska studier planeras [40].

5.7 Kvalitetsdimensioner

Som vi sett i tidigare delkapitel finns det många olika mätningar som kan definieras och det är viktigt att välja vilka man ska genomföra, gärna baserat på en målorienterad metod. Det finns ett antal standarder som beskriver vad som menas med produktkvalitet. ISO9126 beskriver kvalitetsdimensioner för programvarusystem och innehåller dimensionerna funktionalitet, tillförlitlighet, användbarhet, effektivitet, underhållbarhet, portabilitet, uppfyllandegrad och kvalitet i användande [17, 18].

Funktionalitet: Denna dimension beskriver hur väl systemet uppfyller användarnas uttalade och inte uttalade behov. Dimensionen delas upp följande områden: *Lämplighet* anger i vilken utsträckning systemet innehåller en lämplig mängd funktioner, *noggrannhet* anger hur korrekta resultaten är, *samverkansförmåga* beskriver hur väl systemet kan samverka med andra system och *säkerhet* beskriver hur väl systemet skyddar användarnas information.

Tillförlitlighet: Denna dimension beskriver hur driftsäkert systemet är och vilken risk det finns för mjukvarufel ska uppträda då det används. Efterhand som programvara används hittas fler och fler fel som införts i utvecklingen, vilket man säger ger en mer mogen programvara. *Mognadsgrad* anger hur mogen programvaran är i detta avseende och beskriver alltså i vilken utsträckning man tagit bort fel ut programmet och hur länge man kan köra programmet utan att något fel uppstår. *Feltolerans* beskriver hur väl systemet kan fortsätta att exekvera om något fel i programmet skulle uppstå och *återhämtningsförmåga* beskriver i vilken utsträckning systemet kan starta om på ett bra sätt efter att ett fel uppstått.

Användbarhet: Detta beskriver hur intuitivt systemet är och lätt det är att använda för en användare. *Begriplighet* beskriver hur lätt det är för en användare att förstå hur ett system ska användas i olika situationer, *lärbarhet* hur lätt det är att lära sig de funktioner som är nya för användaren och *handhavande* hur lätt det är för användaren att handha, t ex att styra och få information från systemet. *Attraktivitet* beskriver den subjektiva tillfredsställelsen i att använda systemet. Exempelvis påverkas attraktiviteten av den grafiska layouten.

Effektivitet: Programvaran använder ett antal resurser som minne och exekveringstid. Dessa resurser kan användas mer eller mindre effektivt. *Tidsbeteende* beskriver hur lång tid programmet tar på sig att genomföra de beräkningar och operationer det ska utföra och *resursutnyttjande* beskriver hur väl programmet utnyttjar t ex primärminne och diskutrymme.

Underhållsbarhet: Denna dimension beskriver hur enkelt eller svårt det är att ändra systemet när nya funktioner läggs till eller fel som hittats rättas. *Analyserbarhet* beskriver hur lätt det är att hitta orsaken till fel och hur lätt det är att göra en påverkansanalys för att bestämma vilka delar av ett system som måste ändras vid en ändring. *Ändringsbarhet* beskriver hur lätt eller svårt det är att rent konkret gå in i koden och ändra och kompilera om den. Vid ändringar finns det alltid en risk att de medför nya problem. *Stabiliteten* beskriver hur stor risk det är för detta. *Testbarhet*

beskriver hur lätt eller svårt det är att verifiera att införda förändringar är korrekta.

Portabilitet: Detta beskriver hur lätt det är att flytta programmet från en miljö till en annan, t ex mellan två olika operativsystem. *Anpassningsbarhet* beskriver hur lätt det är att anpassa systemet till en annan miljö medan *installationsbarhet* anger hur lätt är det att installera systemet i den omgivning där det ska exekvera. *Samexistens* beskriver hur lätt är det för systemet att samverka med andra system i en omgivning.

Uppfyllandegrad: Detta anger hur väl systemet uppfyller standarder och lagar och förordningar.

Kvalitet i användning: Denna övergripande dimension beskriver hur systemet bidrar i ett större sammanhang där användare samverkar med systemets olika delar i en helhet, t ex en integration av delar som består av programvara, elektronik, eller mekanik.

5.8 Exempel på mått

Det finns ett antal vanliga mått som mäts på programkoden, dvs statiska kodmått. Med statisk menar vi här att de mäts utan att exekvera programmet. Här i detta kapitel visar några exempel, men det finns ganska många fler. Först går vi igenom ett mått för storlek, dvs ett ganska enkelt produktmått som kan användas för i stort sett alla program. Sedan går vi igenom ett produktmått för ”cyklomatisk komplexitet” som kan användas för kod på metodnivå. Detta är alltså ett lite mer avancerat mått än storlek, men fortfarande ett ”standardmått” som i stort sett alltid nämns när man diskuterar produktmått. Efter det går vi igenom två olika uppsättningar mått som används inom objektorienterad programmering, först ”CK-måtten” som är en uppsättning mått för att mäta hur komplext ett system är och sedan Martin’s mått på instabilitet för system.

Hur ger vi alltså exempel på ett antal produktmått. Det ska betonas att det finns avsevärt fler exempel på produktmått och att det givetvis finns exempel på vanliga processmått och resursmått också, men det ligger lite utanför denna kursen att ge en komplett uppsättning mått. En bra sammanfattning har t ex gjorts av Fenton [10].

5.8.1 Storlek

Man vill ofta veta hur stort ett program är. Det tar t ex generellt sett längre tid att utveckla och underhålla ett stort program än ett litet program. Inte minst i planeringen av utvecklingsprojekt är man därför intresserad av att veta hur stora program är som man redan har utvecklat och hur stora program som man ska utveckla förväntas bli för att kunna jämföra. Om man får i uppgift att ändra i ett program så är det förmodligen svårare i ett stort program än i ett litet program.

Det vanligaste sättet att mäta programlängd är att räkna antalet rader kod (LOC – Lines Of Code). Detta kan låta enkelt, men det finns ett antal val man måste göra och man måste bestämma sig för exakt vad man ska mäta för att måtten av olika program ska vara gjorda på samma sätt.

För det första kan man konstatera att det inte är självklart att man kan jämföra storleken av program skrivna i olika språk. Om inte en rad i de olika språken innebär lika mycket funktionalitet så är det inte lätt att tolka och jämföra mätningarna. Vi antar därför att mätningarna är gjorda av program skrivna i samma språk.

Man kan också diskutera vilka rader man ska räkna. Man kan t ex räkna blankrader eller inte. Oftast väljer man att inte räkna blankrader.

Nästa fråga rör om man ska räkna kommentarsrader eller inte. De tillför ju ingen funktionalitet så det skulle tala för att man inte räknar dem, och det vanligaste är att man inte räknar dem. Väl skrivna kommentarer kan dock antas bidra positivt till hur lätt det är att förstå programmet.

Beroende på språk måste man sedan bestämma hur man räknar för olika alternativ. T ex måste man bestämma om man ska räkna varje "{" och "}" om de står på egna rader. Man kan också dela upp rader med radbrytningar helt enkelt för att de blivit för långa för att få plats i den texteditor som man använder. Då måste man bestämma hur man ska göra vid räkningen av dem. Om man väljer att mäta storlek som rader kod så är det lämpligt att kombinera det med en kodningsstandard för att alla program ska räknas på samma sätt [15].

Som storlek kan man också räkna antalet satser, vilket är förhållandevis likt antalet rader. Även här måste man definiera exakt vad man mäter. T ex har man gjort en definition för programmet JavaNCSS, se kapitel 5.9. Studera gärna hur de har gjort.

5.8.2 Komplexitet

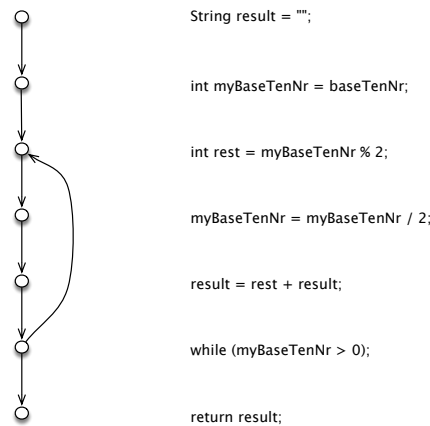
Förutom storleken är man ofta intresserad av hur komplext ett program är, inte minst då man försöker avgöra hur svårt eller lätt ett program är att underhålla.

Det finns många försök att modellera och mäta komplexitet och i förlängningen underhållbarhet. Ett av de första och mest spridda måtten är *McCabe's* cyklo-matiska komplexitet. Det beräknas genom att modellerar programmet som en flödesgraf, där noderna representerar satser och bågarna representerar flödet mellan satserna. Komplexiteten beräknas sedan som antalet "oberoende vägar", vilket beräknas som

$$v(F) = e - n + 2$$

där e är antalet bågar och n är antalet noder. Antalet oberoende vägar är ett grafteoretiskt begrepp och vi går inte igenom i detalj varför det beräknas exakt på detta sättet.

Följande exempel visar en metod i en java-klass som omvandlar (positiva) heltal till binära tal:



Figur 5.2: Flödesgraf för uträkning av cyklomatisk komplexitet

```

public String toBinary(int baseTenNr) {
    String result = "";
    int myBaseTenNr = baseTenNr;
    do {
        int rest = myBaseTenNr % 2;
        myBaseTenNr = myBaseTenNr / 2;
        result = rest + result;
    } while (myBaseTenNr > 0);
    return result;
}

```

Flödesgrafen för detta program visas i figur 5.2. Det finns 7 noder och 7 bågar, vilket betyder att $n = 7$ och $e = 7$, dvs $v(F) = e - n + 2 = 7 - 7 + 2 = 2$. McCabe's cyklomatiska komplexitet för denna metod är alltså 2.

Man kan ta reda på detta mått antingen genom att räkna antalet noder och bågar, eller så kan man helt enkelt för varje metod börja på värdet 1 och öka med ett för varje `if`, `for`, `while`, `catch`, etc. För `switch`-satser får man öka med antalet möjliga val.

Man kan diskutera i vilken utsträckning detta mått verkligen mäter komplexitet av ett program och det finns många som är kritiska till att det motsvarar verklig komplexitet. T ex kan man konstatera att två `if`-satser efter varandra har samma påverkan på den uppmätta komplexiteten som två `if`-satser som är nästlade. Det finns också ett antal andra mått som har föreslagits för att mäta komplexitet.

5.8.3 Mått för objektorienterade program

De exempel på mått som presenterats ovan, dvs LOC och cyklomatisk komplexitet, är användbara för i stort sett alla typer av program. De är inte anpassade för objektorienterad programmering, men de kan användas även för denna typ av program. Eftersom objektorienterade program är uppdelade i klasser och i metoder (och paket) så brukar man sammanställa t ex storlek i antalet rader

per klass eller antalet rader per metod. För cyklomatisk komplexitet så brukar man beräkna det per metod i klasserna. På så sätt kan man bedöma vilka metoder som är komplexa som de metoder med högt värde, och vilka klasser som är komplexa som de klasser där metoderna har hög komplexitet.

Förutom de generella mått som formulerats så finns det ett antal mått som är framtagna för just objektorienterade program. En uppsättning mått kallas CK-måtten efter dess upphovsmän (Chidamber och Kemerer) och en annan är måtten på instabilitet som är framtagna av Robert Martin.

CK-måtten

Chidamber och Kemerer föreslår följande mått (t ex [36], [7] eller [10]):

- WMC (Weighted Methods per Class): Här räknar man antalet metoder i varje klass och viktar värdena med metodernas komplexitet. En enkel metod kan få värdet 1, medan en mer komplex metod får ett större värde. Detta mäts för varje klass och ju större WMC är desto mer komplex antas klassen vara. Man har inte sagt vilket komplexitetsmått som ska användas, men att vilket mått som helst som är på intervall-skala går bra [7].
- DIT (Depth of Inheritance Tree): Detta är djupet av arvskedjan för en klass, dvs hur många klasser det finns ovanför klassen i arvskedjan som potentiellt kan påverka klassen. Ju djupare trädet är desto mer komplext antas programmet vara, eftersom ju fler nivåer det finns, desto fler ärvda klasser måste en utvecklare förstå.
- NOC (Number Of Children): För varje klass mäter man antalet sub-klasser. Ett högt värde på NOC betyder att många klasser är beroende av klassen.
- CBO (Coupling Between Object classes): Man säger att klasser är kopplade om de använder metoder eller publika variabler av varandra. CBO för en klass är definierat som antalet andra klasser som är kopplade till klassen. Ett högt värde på CBO indikerar en högre komplexitet eftersom en ändring i en klass kan innebära att det är nödvändigt att ändra även i de klasser som de har hög koppling med. CBO för en klass antas också påverka hur omfattande testning som måste göras då ändringar görs i klassen.
- RFC (Response For a Class): Detta definieras som antalet metoder i en klass och antalet metoder som dessa metoder kan anropa. Lite mer formellt definieras det i [7] som

$$RFC = |RS|$$

där RS är mängden av responser för klassen, dvs

$$RS = \{M\} \bigcup_{\forall i} \{R_i\}$$

och $\{R_i\}$ är mängden av alla metoder som kan anropas av metod i och M är mängden av alla metoder i klassen. Tanken är att om ett anrop till en klass kan medföra många ytterligare anrop så ökar detta komplexiteten och andra aspekter som t ex hur mycket testning som måste göras.

- LCOM (Lack of Cohesion in Metrics) Detta är ett mått på sammanhållning (eng. cohesion), eller rättare sagt ett mått på bristen på sammanhållning i en klass. Här har man valt att mäta detta genom att undersöka hur många metoder som inte har gemensamma variabler med andra metoder i klassen och jämföra med hur många metoder som har gemensamma variabler. Värdet av LCOM har diskuterats en del och det är inte säkert att det verkligen tillför något i en analys av ett verkligt system [36].

I flera av de mätverktyg som finns för att analysera mjukvara så ingår det funktioner för att mäta CK-måtten. Måtten försöker svara på hur komplext ett system med sina klasser är och hur svårt det är att arbeta med dem. Om man t ex har underhållsbarhet eller komplexitet som övergripande mål så kan dessa mått vara värda att undersöka.

Martin's mått på instabilitet

En annan uppsättning mått som kan användas för objektorienterade system har tagits fram av Robert Martin (t ex [28]). Målet är att kunna analysera hur "stabil" en design är baserat på kopplingen mellan olika paket. Här antar man att mjukvaran är uppdelad i paket som i sin tur är uppdelad i klasser.

Ju färre paket ett paket beror på desto stabilare är det, eftersom det inte är lika troligt att det måste ändras pga av ändringar i andra paket. Som ett mått på "utåtledande" definierar Martin "efferent coupling"

C_e = antalet klasser i paketet som beror på klasser i andra paket

I vissa fall mäter man detta som antalet klasser utanför paketet som klasser i paketet beror på, t ex i verktyget JDepend¹.

En annan viktig aspekt är hur stort "ansvar" ett paket har för andra paket och han definierar därför "afferent coupling" som

C_a = antalet klasser i andra paket som beror på klasser i paketet

Tanken är att om många andra klasser beror på klasser i paketet så är paketet förmodligen stabilt eftersom det kan medföra så mycket problem att ändra i paketet. Som ett mått på instabilitet, I , definierar han

$$I = \frac{C_e}{C_a + C_e}$$

$I = 0$ är ett maximalt stabilt paket där inga klasser i paketet beror på paket utanför klassen, men det finns klasser utanför paketet som beror på klasser i paketet. $I = 1$ är ett maximalt instabilt paket där inga klasser utanför paketet beror på klasser i paketet, men där klasser i paketet beror på klasser utanför paketet.

Han kombinerar detta med ett mått, A , som mäter i hur stor utsträckning klasserna i ett paket är abstrakta

$$A = \frac{\text{antalet abstrakta klasser i paketet}}{\text{antalet klasser i paketet}}$$

¹<http://clarkware.com/software/JDepend.html> alt. <https://github.com/clarkware/jdepend>

För både A och I gäller att de är mellan 0 och 1.

Man menar att en instabil klass ändrar sig mycket, vilket betyder att det är bra om inte för många andra klasser beror på den. På samma sätt menar man att en klass som många beror av bör vara stabil. Man menar att det finns en "main sequence" som beskrivs av den rätta linjen

$$I + A = 1 \quad \begin{array}{l} 0 \leq I \leq 1 \\ 0 \leq A \leq 1 \end{array}$$

som tyder på att man har en bra avvägning mellan hur abstrakta klasserna i ett paket är och hur instabila de är.

Ibland vill man analysera hur bra en design är med avseende på stabilitet och hur abstrakta klasser är. Det kan man göra genom att undersöka hur långt designen ligger från Martin's "main sequence". Om vi kallar avståndet $D(I, A)$ så är det ganska lätt att visa att det är

$$D(I, A) = \left| \frac{1}{\sqrt{2}}(I + A - 1) \right| \quad \begin{array}{l} 0 \leq I \leq 1 \\ 0 \leq A \leq 1 \end{array}$$

5.9 Verktyg

Det finns många verktyg för att mäta på källkod, både kommersiella verktyg och verktyg som levereras enligt en licens för öppen källkod. Nedan går vi igenom några exempel på verktyg.

5.9.1 JavaNCSS

JavaNCSS² "Non Commenting Source Statements (NCSS)". Man får tillgång till programmet genom att lägga till ett antal `.jar`-filer till sin `classpath`. På LTH:s datorer kan man göra det med kommandot

```
> export PATH=$PATH:/usr/local/cs/EDAA35/javancss-32.53/bin
```

Man kan exekvera NCSS t ex såhär:

```
> javancss -all *.java
```

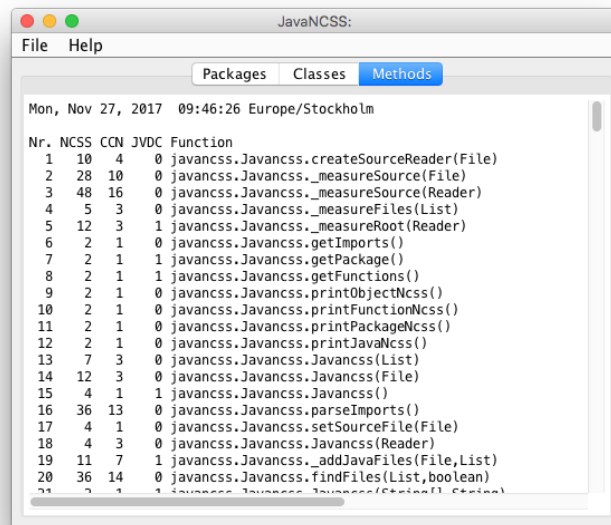
om man står i biblioteket med `.java`-filerna i sitt projekt. Man kan också starta ett grafiskt användargränssnitt t ex såhär:

```
> javancss -recursive -all -gui
```

där `-recursive` berättar att den ska analysera alla `java`-filer som ligger i underbibliotek och `-gui` att ett fönster som t ex i figur 5.3 ska startas. I fönstret kan man välja att antingen studera metrics per paket, per klass, eller per metod. Om man väljer att se per metod, som i figur 5.3 så får man t ex reda på antalet rader (NCSS) och cyklomatisk komplexitet (CCN). Om man skrollar ned så får man en sammanställning av medelvärden för hela programmet.

När detta kompendium skrevs så fungerade JavaNCSS fortfarande inte för lambda-funktioner.

²<http://www.kclee.de/clemens/java/javancss/index.html>



Mon, Nov 27, 2017 09:46:26 Europe/Stockholm

Nr.	NCSS	CCN	JVDC	Function
1	10	4	0	javancss.Javancss.createSourceReader(File)
2	28	10	0	javancss.Javancss._measureSource(File)
3	48	16	0	javancss.Javancss._measureSource(Reader)
4	5	3	0	javancss.Javancss._measureFiles(List)
5	12	3	1	javancss.Javancss._measureRoot(Reader)
6	2	1	0	javancss.Javancss.getImports()
7	2	1	1	javancss.Javancss.getPackage()
8	2	1	1	javancss.Javancss.getFunctions()
9	2	1	0	javancss.Javancss.printObjectNcss()
10	2	1	0	javancss.Javancss.printFunctionNcss()
11	2	1	0	javancss.Javancss.printPackageNcss()
12	2	1	0	javancss.Javancss.printJavaNcss()
13	7	3	0	javancss.Javancss.Javancss(List)
14	12	3	0	javancss.Javancss.Javancss(File)
15	4	1	1	javancss.Javancss.Javancss()
16	36	13	0	javancss.Javancss.parseImports()
17	4	1	0	javancss.Javancss.setSourceFile(File)
18	4	3	0	javancss.Javancss.Javancss(Reader)
19	11	7	1	javancss.Javancss._addJavaFiles(File, List)
20	36	14	0	javancss.Javancss.findFiles(List, boolean)

Figur 5.3: JavaNCSS (resultat för mätning på källkoden till JavaNCSS)

5.9.2 PMD

Ett annat verktyg som kan användas för att räkna antalet rader och cyklomatisk komplexitet är PMD³. Se t ex "Installation and basic CLI usage" under "User Documentation"⁴ på deras hemsida för information om hur man installerar och använder systemet.

Verktyget är egentligen till för att man ska sätta "tröskelvärden" för att få meddelanden om någon klass eller metod överskrider ett visst värde. Om man vill ha värdet för alla klasser och metoder kan man därför sätta tröskelvärdet till 1, så att alla mätvärden hamnar över värdet och därmed presenteras.

Verktyget är skrivet i Java och finns till både Windows och Linux/Unix (och Mac). Man kör det t ex med följande kommando i Linux:

```
> ./run.sh pmd -d <bibliotek> -R <regler>.xml
```

där <bibliotek> är roten till den källkod i Java som man vill analysera och <regler> är en xml-fil där man specificerar bland annat vilka mått man vill mäta. Titta gärna i dokumentationen vad som ska vara med i xml-filen och vilka mått som kan mätas. Det finns möjlighet att specificera vilket format resultatet ska vara med en -f-parameter, men om man inte gör det blir det textformat.

För att mäta rader kod (NCSS) och cyklomatisk komplexitet per klass och metod kan följande xml-fil användas:

```
<?xml version="1.0"?>
<ruleset name="EDAA35"
```

³<https://pmd.github.io/pmd>

⁴https://pmd.github.io/pmd/pmd_userdocs_installation.html

```

xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://pmd.sourceforge.net/ruleset
    /2.0.0 https://pmd.sourceforge.io/ruleset_2_0_0.xsd">

<description>
    Rules in EDAA35
</description>

<!-- Here you add the rules you want to use -->

<!-- Report LOC -->
<rule ref="category/java/design.xml/NcssCount">
    <properties>
        <property name="methodReportLevel" value="1" />
        <property name="classReportLevel" value="1" />
        <property name="ncssOptions" value="" />
    </properties>
</rule>

<!-- Report cyclomatic complexity -->
<rule ref="category/java/design.xml/CyclomaticComplexity">
    <properties>
        <property name="classReportLevel" value="1" />
        <property name="methodReportLevel" value="1" />
        <property name="cycloOptions" value="" />
    </properties>
</rule>

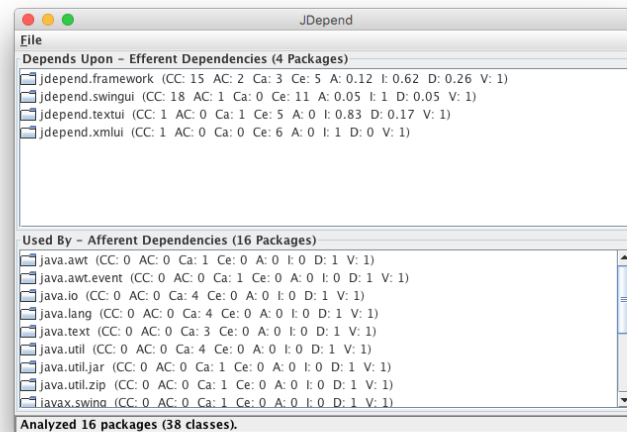
</ruleset>

```

5.9.3 JDepend

Det finns flera verktyg som mäter Martin's mått på instabilitet. Ett exempel är JDepend⁵ som kan användas för att analysera java-program. Om man t ex startar det för JDepends egen kod får man ett fönster som i figur 5.4. CC är antalet klasser i paketen, medan Ca, Ce, A, och I är mått enligt kapitel 5.8.3. Vi går dock inte in i detalj på vilken information som presenteras här.

⁵<https://github.com/clarkware/jdepend>



Figur 5.4: JDepend (resultat för mätning på källkoden till JDepend)

Kapitel 6

Analys

Följande steg är vanliga i analys av data:

1. Deskriptiv analys för att förstå mer om den data man har. Detta kan t ex innebära att man ritar grafer för att förstå datan.
2. Identifiering och behandling av "outliers". Här identifierar man eventuella datapunkter som man av någon anledning inte litar på.
3. Statistisk analys, t ex för att visa på skillnader mellan behandlingar i experiment

Innehållet i stegen förklaras närmare i delkapitlen 6.2, 6.3 och 6.4 nedan. Innan dess går vi först igenom lite information om slumpmässiga variationer och sannolikheter i delkapitel 6.1.

6.1 Slumpmässiga variationer och sannolikheter

I den typ av mätningar som görs i kursen räcker det oftast inte att göra en mätning och sedan använda det resultatet för att dra slutsatser. Om man t ex mäter exekveringstiden för en funktion i ett Java-program en gång blir resultatet ett värde. Om man sedan mäter en gång till för samma funktion med samma indata blir det förmodligen inte exakt samma värde. Detta beror på ett antal faktorer som att Javas exekveringssystemet inte beter sig på samma sätt varje gång och att processorns (eller processorkärnans) kapacitet delas mellan olika processer i datorn där programmets process bara är en av flera processer. Det kan t ex också bero på att tillgång till I/O-enheter tar olika lång tid vid olika tillfällen.

Det finns alltså ofta naturliga variationer i mätningar. Om man mäter "samma sak" n gånger får man alltså en mätserie av olika tal

$$x_1, x_2, \dots, x_n$$

6.1.1 Sannolikhet

I olika situationer kan man mena olika saker med termen "sannolikhet" (t ex [33]). Man kan använda termen på ett subjektivt sätt, t ex att "Sannolikt vinner

MFF på söndag”. Detta är alltså en subjektiv bedömning. Ibland gör man också en kvantitativ bedömning av sannolikheten som t ex att det är ”1 till 4 att de vinner turneringen”.

Ett annat sätt att använda termen är i mer matematiska termer. Om man t ex har en ”perfekt kortlek” som är ”perfekt blandad” så är sannolikheten $1/4$ att man drar ett hjärter om man drar ett slumpvis valt kort. Detta sätt att betrakta händelser med lika sannolika händelser etc. gör det möjligt att bestämma den matematiska, teoretiska sannolikheter för olika händelser. Om sedan verkligheten stämmer med dessa sannolikheter beror på hur väl antaganden man gjort stämmer med verkligheten.

Den tredje typen är empirisk sannolikhet baserad på relativa frekvenser. Här skattas sannolikheten att en händelse inträffat genom att studera hur många gånger den inträffat jämfört med hur många försök man gjort, dvs hur många gånger den skulle kunna inträffat. Sannolikheten är alltså en skattning som blir bättre och bättre ju fler försök man gör.

Det finns alltså med detta resonemang tre olika typer av sannolikheter, subjektiv, teoretisk och empirisk. Dessa används ofta tillsammans. Man har en tanke om vilken vilken teoretisk sannolikhet som beskriver en händelse, men man observerar den empiriska.

Sannolikheter beskrivs ofta numeriskt med ett värde mellan 0 och 1, där 0 betyder att en händelse aldrig inträffar och 1 betyder att en händelse inträffar vid varje försök. En sannolikhet beskrivs ofta med bokstaven P , t ex den teoretiska sannolikheten att man får klave om man singlar slant som $P(\text{klave}) = \frac{1}{2}$.

6.1.2 Fördelningar

När man tittar på flera utfall av en händelse så kan man se ett mönster i vilka resultat man får. Om man t ex genererar 10 slumpstal mellan 1 och 10, där alla tal är lika sannolika, kan man få resultat enligt histogrammet i figur 6.1.

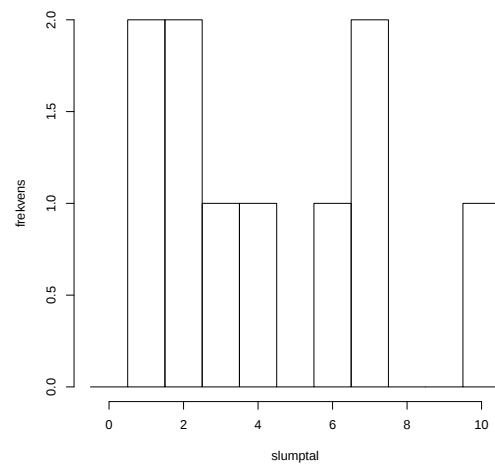
Man kan se att det t ex fanns två ettor en trea och ingen femma. Orsaken till att det är så stora skillnader är att det är så få slumpstal vi tittar på. Det är t ex givetvis inte ett generellt mönster att det finns dubbelt så många ettor som treor.

Om man däremot drar 5000 slumpstal får man ett histogram som i figur 6.2. Detta histogram börjar bli lite mer användbart och man kan ana att slumpstalen kommer från en fördelning där alla tal mellan 1 och 10 är lika sannolika. Hade vi dragit väldigt många slumpstal så hade vi sett att de kom från en ”rektangelfördelning” med slumpstal mellan 1 och 10.

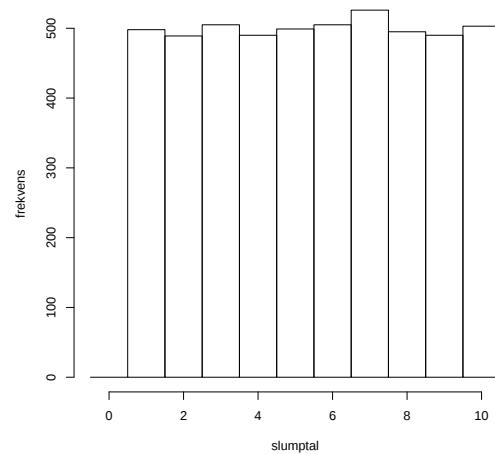
Det finns alltså en teoretisk fördelning bakom de observationer man gör. Denna fördelning kan beskrivas av en täthetsfunktion som beskriver hur sannolikt det är att få ett värde från ett visst intervall. T ex ser den ut som en rektangel för en rektangelfördelning, dvs det histogram vi skulle fått om vi hade haft väldigt många slumpstal enligt ovan.

Det finns både diskreta och kontinuerliga fördelningar. En diskret fördelning beskriver sannolikheten av olika värden från en lista av värden, som t ex de 10 värdena enligt ovan. En kontinuerlig fördelning beskriver situationen att man kan få vilket värde som helst, men olika värden, eller intervall av värden, är olika sannolika, som t ex längden hos olika personer.

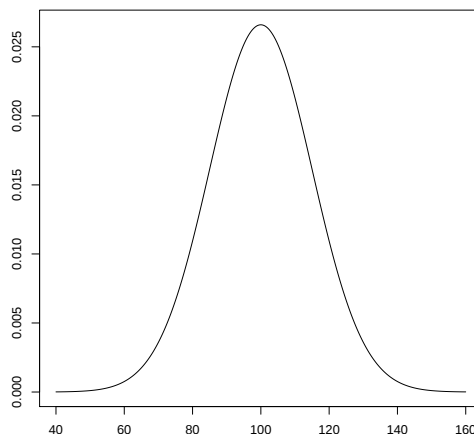
Många saker som man mäter på i naturen följer en normalfördelning. Denna ser ut som en klocka där de flesta värdena ligger nära värdenas medelvärde,



Figur 6.1: Histogram för 10 slumptal



Figur 6.2: Histogram för 5000 slumptal



Figur 6.3: Normalfördelning

men där det också finns värden längre från medelvärdet. Normalfördelningens täthetsfunktion visas i figur 6.3. Denna fördelning kan beskrivas av formeln

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

där μ är medelvärdet och σ är ett mått på spridningen (standardavvikelsen enligt nedan). Detta är alltså en kontinuerlig fördelning. Fördelningen beskrivs helt genom medelvärdet och standardavvikelsen, dvs om man vet dessa två storheter kan man räkna ut täthetsfunktionen i vilken punkt som helst.

6.2 Deskriptiva mått

6.2.1 Mått på centralitet

Ett vanligt mått på centralitet är *medelvärde*. För en talserie beräknas detta som summan av talen delat med antal tal, dvs

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

om man har n tal. Detta kallas också för aritmetiskt medelvärde, men i de fall man bara skriver 'medelvärde' så är det denna typ av medelvärde man menar.

Ett annat mått som kan användas är *medianen* som fås genom att först sortera alla talen i ordning och sedan, om det är ett udda antal tal, ta det tal som har lika många tal före och efter sig. Om det är ett jämnt antal tal tar man medelvärdet av de två mittersta talen.

Ytterligare ett mått är *typvärde* (ibland kallas det också modalvärde och på engelska heter det 'mode'). Typvärdet är det vanligaste värdet i en dataserie.

T ex om vi har dataserien 4, 3, 9, 5, 1, 5, 4, 4, 8, 9 så får man följande deskriptiva mått på centralitet:

- Medelvärde: $(4 + 3 + 9 + 5 + 1 + 5 + 4 + 4 + 8 + 9)/10 = 5,2$
- Median: $(4+5)/2 = 4,5$
- Typvärde: 4

6.2.2 Spridningsmått

Ett vanligt mått på en datamängds spridning är *standardavvikelsen*

$$s = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2}$$

Detta mått bygger på det kvadratiska avståndet mellan varje datapunkt och medelvärdet av alla datapunkter. Orsaken till att man ska dela med $(n-1)$ och inte n behandlas i senare kurser. Lite informellt kan man dock se att den n :te skillnaden, $x_n - \bar{x}$, kan räknas ut om man vet medelvärdet, $\bar{x}$, och de andra $n-1$ skillnaderna, dvs det finns $n-1$ frihetsgrader [26].

Ett annat mått är *variationsvidd* (eng. range). Detta är avståndet mellan den största datapunkten och den minsta datapunkten, dvs

$$R = \max_i x_i - \min_i x_i$$

6.2.3 Osäkerhetsmått – konfidensintervall

När man skattar värden som t ex medelvärdet finns det en osäkerhet i vad "det verkliga" medelvärdet är. Med "det verkliga" medelvärdet menar vi då medelvärdet av den underliggande fördelning som vi antar att värdena kommer från. Om vi t ex antar att värdena kommer från en normalfördelning med ett visst medelvärde och en viss standardavvikelse så tänker vi att värdena kommer från en fördelning som beskrivs av formeln för normalfördelningen, men där vi inte vet medelvärdet och standardavvikelsen. Detta är alltså de "verkliga värdena" som vi vill skatta baserat på våra observationer.

Om man t ex har en enda mätning är det ganska uppenbart att det "medelvärde" vi får med denna mätning inte alls är en bra skattning av det verkliga medelvärdet. Har vi t ex 10 mätningar är medelvärdet nog lite bättre som skattning, men som vi såg med de 10 slumpalen i kapitel 6.1.2 så behöver det inte ge en så bra förståelse av den underliggande fördelningen.

För att visa på skattningosäkerheten av de parametrar man skattar så beskriver man ofta ett *konfidensintervall* som innehåller den parameter man skattat. Man kan kräva olika konfidensgrad av ett konfidensintervall, t ex 95% eller 99%.

Exempelvis räknas ett konfidensintervall för ett medelvärde ut genom att först räkna ut medelvärdet och enligt kapitel 6.2.1. Detta är ju en skattning av medelvärdet och därför kan man göra ett konfidensintervall genom att använda följande funktion i R:

```
confidenceInterval(x, confidenceLevel=0.95)
```

Tabell 6.1: Användandet av deskriptiva mått för olika skalor

Mått	Nominell	Ordinal	Intervall	Ratio
Medelvärde	ok	ok	ok	ok
Median			ok	ok
Typvärde				
Varians		ok	ok	ok
Variationsvidd			ok	ok

som man på LTH:s datorer får tillgång till genom att skriva

```
source("/usr/local/cs/EDAA35/R_resources.R")
```

Innebörden av detta intervall är att det verkliga medelvärdet av den bakomliggande fördelningen med en sannolikhet enligt konfidensgraden ligger inom konfidensintervall. Detta kommando antar mätningarna kommer från en normalfördelning, skattar medelvärdet enligt kapitel 6.2.1 och standardavvikelsen enligt kapitel 6.2.2 och, baserat på detta, räknar ut ett konfidensintervall. Exakt hur det fungerar beskrivs i senare kurser i statistik [5].

Det som påverkar storleken på ett konfidensintervall för medelvärdet är

- Spridningen: ju större spridning det är i datan desto större blir konfidensintervall
- Antal datapunkter: ju fler datapunkter man har desto mindre blir konfidensintervall
- Konfidensgraden: ju större konfidensgrad man väljer desto större blir konfidensintervall

6.2.4 När ska man använda vilket mått?

Alla mått är inte relevanta för alla skalor. I figur 6.1 anges vilka mått som är lämpliga att använda för olika skalor.

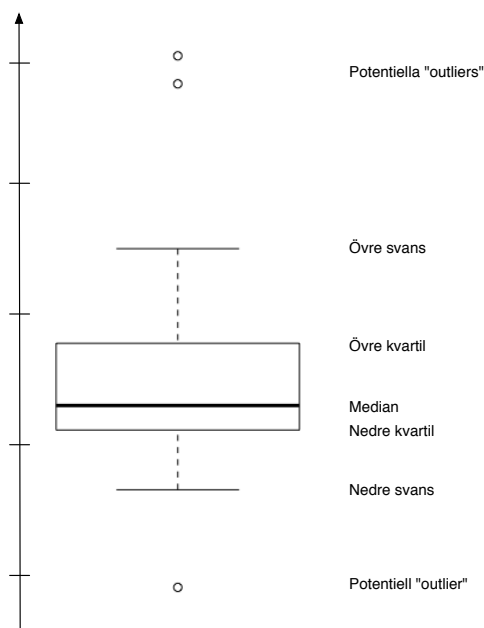
6.3 Hantering av ”outliers”

När man samlar in data så hamnar man ofta i situationen att man har ett antal datapunkter som man inte litar på. De kan t ex vara uppmätta på fel sätt eller så kan mätningarna av någon annan anledning ha blivit förstörda. I många fall kan man identifiera dem genom att värdena är orimligt höga eller orimligt låga, men alla värden som är höge eller lägre än de andra värdena är givetvis inte ”outliers”.

När man har data är det alltså ofta nödvändigt att först identifiera vilka potentiella ”outliers” man har och sedan bestämma vad man ska göra med den.

6.3.1 Identifiering av potentiella ”outliers”

Man kan som sagt ofta identifiera ”outliers” genom att titta på storleken av alla värdena och om det är något värde som ”sticker ut” så kan man titta på detta



Figur 6.4: Boxplot

lite extra för att ta reda på om det verkligen är ett "outlier"-värde eller inte. Ett sätt att göra detta är att rita en "boxplot" med värdena, se figur 6.4.

De värden som finns i grafen kan förklaras enligt följande [10]:

Median Medianen av alla värden

Övre kvartil (u) Medianen av alla värdena över medianen

Nedre kvartil (l) Medianen av alla värdena under medianen

Övre svans Det största värdet som inte är större $u + \frac{3}{2}d$, där $d = u - l$ är boxstorleken

Nedre svans Det minsta värdet som inte är mindre $l - \frac{3}{2}d$, där $d = u - l$ är boxstorleken

Potentiella "outliers" De värden som är större än övre svansen eller mindre än nedre svansen

Boxplottar används inte enbart för att identifiera "outliers". De används också ofta för att få en förståelse för hur data är fördelat och om fördelningen är symmetrisk eller skev.

6.3.2 Vad ska man göra med potentiella "outliers"

När man väl har identifierat "outliers" är nästa fråga vad man ska göra med dem. Man måste alltså bestämma om man ska slänga datapunkterna eller om man ska behålla dem.

Om man kan konstatera att datapunkterna helt enkelt är felaktiga så är det förmodligen ett ganska enkelt beslut att slänga dem och eventuellt försöka samla in ny data.

Om man kan konstatera att datapunkterna är korrekta så ska man givetvis inte slänga dem.

Om man inte är säker på om datapunkterna är korrekta eller inte så får man väga fördelar mot nackdelar för att bestämma om man ska behålla dem. Om man behåller dem så tar man risken att de är felaktiga. Om man tar bort dem tar man risken att missa information som man egentligen har tillgång till om de är korrekta.

6.4 Jämförande tester

I många fall står man inför situationen att man vill jämföra två eller flera olika uppsättningar med mätningar. Det kan t ex vara att man gjort två olika lösningar, A och B, på samma problem och vill veta vilken som är snabbast. Om man då mäter en gång så kan man få att A tar 1,30 sekunder och B tar 1,32 sekunder. Som vi sett med konfidensintervallen innan så är det inte alls säkert att vi kan dra den generella slutsatsen att A är snabbare än B bara baserat på denna enda mätning. Det är ju mycket möjligt att resultatet blir annorlunda om man mäter en gång till. Man måste alltså mäta tillräckligt många gånger för att kunna dra en slutsats som man kan lita på.

Det finns sätt att dra slutsatser baserat på jämförande mätningar. För dessa har man definierat följande hypoteser

- Nollhypotes, ofta kallad H_0 : Denna hypotes säger att det inte finns någon skillnad mellan de olika mätningarna.
- Alternativhypotes, ofta kallad H_a : detta är motsatsen till nollhypotesen och den statistiska analysen går ut på att undersöka om det är möjligt att förkasta nollhypotes och säga att den alternativa hypotesen gäller.

Att utföra hypotestest innebär risker att göra ett antal fel. Två viktiga fel är

- Typ-I-fel: Denna typ av inträffar då en statistisk analys säger att det finns en skillnad men att det i verkligheten inte gör det. Sannolikheten att förkasta H_0 även om det i verkligheten inte finns någon skillnad kallas ibland för α . Denna sannolikhet vill man hålla nere. T ex vill man kunna säga att sannolikheten att man dragit fel slutsats är max 5% eller max 1%, vilket man också kallar ett tests *signifikansnivå*.
- Typ-II-fel: Denna typ av fel inträffar då en statistisk analys säger att det inte finns en skillnad men att det i verkligheten gör det.

Man vill givetvis hålla nere sannolikheten för båda typerna av fel, men man kan konstatera att det är värre att göra ett typ-I-fel än ett typ-II-fel. Ett typ-I-fel innebär ju att man drar slutsatser fast man inte borde göra det, medan ett typ-II-fel "bara" innebär att man konstaterar att man inte hade underlag att dra några slutsatser.

Tabell 6.2: Exempel på hypotestest för några vanliga designtyper

Design	Parametriskt test	Icke-parametriskt test
En oberoende variabel med två nivåer	t-test	Mann-Whitney
En oberoende variabel med två nivåer, upprepade mått	"paired t-test"	Wilcoxon
En oberoende variabel, mer än två nivåer	ANOVA	Kruskal-Wallis

De finns flera olika sorters statistiska test uttänkta som man kan använda när man gör jämförande tester. De kan delas upp i *parametriska* och *icke-parametriska*. Parametriska tester förutsätter bland annat att det man mäter varierar enligt någon viss fördelning, normalfördelning, och att skalan på intervall eller rationivå. Om skalan är på en lägre nivå, dvs ordinalskala, går det inte att använda tekniker som bygger på att avståndet mellan mätresultat är relevant och inte bara ordningsföljden.

Tabell 6.2 visar exempel på olika tester som kan utföras.

Exempelvis kan ett t-test utföras enligt följande i R:

```
> a <- c(2, 3, 4, 6.4, 3.5, 7, 3, 4.5, 4, 6)
> b <- c(9, 11, 18, 14, 16, 13.5, 11, 6, 14, 13, 12)
> t.test(a, b)
```

Welch Two Sample t-test

```
data: a and b
t = -7.3129, df = 14.982, p-value = 2.574e-06
alternative hypothesis: true difference in means is not
equal to 0
95 percent confidence interval:
-10.538604 -5.781396
sample estimates:
mean of x mean of y
4.34 12.50
```

Här testas alltså om det finns en signifikant skillnad mellan de underliggande medelvärdena för talserierna a och b. Svaret man får säger först vad värdena av "t" och "df" är. Detta är parametrar till den fördelning som testet baseras på (t-fördelningen), vilka förklaras närmare i senare kurser.

"p-value" beskriver på vilken signifikansnivå man kan dra slutsats om skillnad på. Om man t ex vill ha signifikansnivån 95% så betyder det att detta värde ska vara mindre än 0,05. I detta fallet är värdet avsevärt mindre än 5%, vilket gör att man kan dra slutsatsen att det finns en signifikant skillnad mellan de underliggande medelvärdena för talserierna. Efter det beskrivs vilken "alternative hypothesis" som är undersökt, dvs H_a enligt ovan. Efter det beskrivs ett 95% konfidensintervall för skillnaden mellan talserierna och efter det medelvärdena för talserierna.

Som i de flesta andra statistikprogram så finns flera inbyggda test i R. Se t ex hjälpen för `t.test` och `wilcox.test`.

Kapitel 7

Att rapportera resultat skriftligt

När man gjort en undersökning eller annan studie vill man givetvis berätta om resultatet för andra. Ett viktigt sätt att göra detta är att skriva en rapport, dvs en teknisk rapport som beskriver resultaten man kommit fram till. Andra möjliga sätt att publicera skriftligt är artiklar till vetenskapliga tidskrifter eller konferenser, populärvetenskapliga artiklar, webbsidor, etc. De första påminner i viss utsträckning om tekniska rapporter, så fokus i detta kapitel är på tekniska rapporter.

En rapport innehåller inte bara själva resultaten man kommit fram till. Det hade varit ganska svårt att förhålla sig till en rapport som bara berättade resultat utan att säga någon om hur man kommit fram till dem. Man går därför igenom även andra saker än själva resultaten, som t ex hur resultaten relaterar till redan genomfört arbete, vad målen med studien var, vilken metod man använt för att komma fram till resultaten, etc.

Att man har med alla dessa delar gör att läsaren kan förstå sammanhanget som studien gjorts i och dessutom skapa sig en bild av validiteten av resultaten. Det ökar trovärdigheten och gör att läsare kan lita på resultaten. Dessutom finns det möjlighet för andra att repetera studien och se om de får samma resultat, dvs replikera studien.

Det är alltid en bra idé att försöka definiera vilken målgrupp man har med en rapport. Är det t ex en rapport som vänder sig till allmänheten så får man ta hänsyn till det när man skriver och det blir inte samma innehåll som om man vänder sig till kolleger på ett företag. Om man vänder sig till andra personer på samma företag som man själv arbetar på så kan man räkna med att de känner till mer bakgrundsmaterial, eller med referenser kan ta reda på detta, än om man vänder sig till allmänheten.

7.1 Disposition

Rapporten kan disponeras på olika sätt, det beror t ex på traditionen inom det ämne där man presenterar sina resultat. Man kan dock tänka på att ha med följande delar i rapporten:

- Titel
- Kort sammanfattning (eng. abstract): Man brukar ofta ha en sammanfattning som täcker alla delar av rapporten.
- Inledning: I inledningen är det viktigt att sätta rapporten i sitt sammanhang, t ex genom att förklara varför arbetet behövs och vilken behov den täcker.
- Bakgrund och relaterat arbete: Bakgrundsmaterial och relaterat arbete sammanfattas, se t ex kapitel 3.1.
- Frågeställning: De frågeställningar som behandlas och besvaras i rapporten sammanställs. Detta kan t ex ske i form av ett antal listade forskningsfrågor, men kan också beskrivas i vanlig text, se kapitel 2.
- Metodbeskrivning: Här beskriver man vilken metod man haft och hur man gjort för att svara på frågorna man ställt sig. Man kan t ex gå igenom vilken övergripande metod man använt, vilka mätningar som gjorts hur de gjorts, hur man har analyserat mätningarna, etc. Det är också bra att förklara varför man valt de mätningar man valt, varför man analyserar på det sätt man gjort, etc. när detta är lämpligt att beskriva.
- Validitetsdiskussion: Man kan ha med ett stycke som beskriver hur man tagit hand om validitetshot till studien, se t ex kapitel 4.6.3.
- Presentation av utfört arbete och resultat: Man presenterar de resultat man mätt eller fått på annat sätt, och om det är lämpligt hur det gick till att genomföra studien.
- Diskussion: Man vill ofta tolka och diskutera de mätningar man gjort, vilket givetvis är möjligt. Det är dock ofta bra att vara tydlig med vad som är rapportering av uppmätta resultat (dvs förra punkten) och vad som är ens tolkningar och diskussion.
- Slutsatser: Slutsatserna presenteras ofta i ett eget kapitel. Slutsatserna ska svara på de frågeställningar man listat för studien.
- Fortsatt arbete: Som en del av arbetet kan man dra slutsatser om vilket fortsatt arbete som bör göras.
- Litteraturförteckning

Den struktur som beskrivs ovan innehåller de viktiga delarna i en rapport. Det är dock viktigt att påpeka att strukturen i detalj kan skilja sig från ämnesområde till ämnesområde. Om man t ex tittar på forskningsartiklar från olika områden kan man se att de olika delarna kan komma i olika ordning och de kan ha olika rubriker eller vara sammanslagna till större kapitel eller uppdelade i mindre kapitel. Det är på samma sätt möjligt att den exakta strukturen i framtida kurser kan skilja sig åt något. Det är alltid viktigt att ta reda på vilket format som förväntas för de rapporter man skriver, vilket gäller både i kurser och i senare arbete.

Rapporten delas in i kapitel, stycken etc., ofta numrerade t ex som i detta kompendium. Kapitlen kan t ex följa de innehållsdelar som presenteras ovan,

Tabell 7.1: Innehåll, presentation av experiment, anpassad och nerkortad från [20].

Kapitel	Exempel på innehåll
Titel	–
Strukturerad sammanfattning	Bakgrund, Mål, Metod, Resultat, Begränsningar, Slutsatser
Inledning	Problemformulering, Forskningsmål, Kontext
Bakgrund	Undersökt teknologi, Alternativa teknologier
Planering	Sampling, Experimentmaterial, Uppgifter för deltagare, Hypoteser, Design, Analysmetoder
Genomförande	Förberedelser, Avvikelser från plan
Analys	Deskriptiv statistik, Hypotestestning
Diskussion	Utvärdering av resultat och implikationer, Validitetshot, Diskussion om generalisering av resultat, "Lessons learned"
Slutsatser och fortsatt arbete	Sammanfattning, Påverkan, Fortsatt arbete
Referenser	–

men det är givetvis möjligt att slå samman delar till ett kapitel eller dela delar till flera kapitel. Man brukar inte ha så många kapitel som ovan. T ex slår man ofta samman frågeställning och metodbeskrivning i ett kapitel. Validitetsdiskussionen lägger man ofta med metodbeskrivningen eller med diskussionsdelen.

Om man har delarna i ungefär den ordning som listas ovan får man en vanlig disposition för empirisk forskning inom programvarusystem. Man presenterar denna ofta som en "IMRAD"-struktur (Introduction, Method, Results and analysis, Discussion) som ofta rekommenderas och kan ses som en minsta lista över kapitel man bör ha med. Man kan även jämföra t ex med tabell 7.1 från [20]. Man kan se att även den dispositionen följer punktlistan ovan ganska väl men att den är anpassad till kontrollerade experiment.

7.1.1 Språket och textens uppbyggnad

Försök använda ett ganska formellt språk. Använd i alla fall inte slanguttryck eller talspråk. Tänk på att en läsare bildar sig en uppfattning om kvalitén på innehållet också efter hur väl texten är formulerad. Använd gärna de hjälpmedel som finns i texteditorer och ordbehandlingsprogram för stavningskontroll och grammatikhjälp. Att få sin rapport granskad av någon annan innan den är färdig kan rekommenderas. Det ger ofta bra synpunkter både på innehållet och hur lätt den är att förstå.

Var sparsam med förstärkningsord. Det är t ex inte tydligt vad som menas med att en exekveringstid är "våldigt lång", jämfört med en "lång" exekveringstid. Ange då hellre hur lång den verkligen var.

Ofta skriver man med ett passivt språk, t ex "exekveringstiden mättes" i stället för t ex "vi mätte exekveringstiden", men det händer att man skriver i

aktiv form också. Det kan vara bra att se hur andra skriver inom området innan man börjar skriva på ett visst sätt.

Texten bör delas in i kapitel med underkapitel, ungefär som i detta kompendium. Normalt numrerar man kapitel och stycken. Man brukar inte ha mer än tre rubriknivåer.

Figurer och tabeller numreras och alla figurer och tabeller ska refereras minst en gång i texten.

7.2 Litteraturförteckning

Man använder referenser för att referera till andra källor. Det finns många olika sätt att referera, t ex "Harvard-systemet", "notsystemet" och "siffersystemet" [3]. Harvard-systemet bygger på att man presenterar författarnamn och årtal inom parenteser i den löpande texten för att man ska hitta rätt referenspost i referenslistan. Siffersystemet bygger på att man numrerar alla referenser i referenslistan och sedan refererar med numren i löptexten, dvs så som referenser fungerar i detta kompendium. Notsystemet bygger på att man har noter, ungefär som fotnoter, vanligtvis i sidfoten.

Man använder de olika systemen i olika utsträckning i olika discipliner och tillfällen. Siffersystemet används ofta på tekniska och naturvetenskapliga konferenser och det är det system man ofta använder om man skriver i LaTeX och BibTeX. I tidskrifter och i böcker använder man ofta ett parentessystem, vilket också kan stödjas med LaTeX och BibTeX. Notsystemet används ofta i humanistiska ämnen och behandlas inte i detta kompendium.

7.2.1 Vilken information ska finnas med?

Oavsett vilket format på referenser man använder så är det viktigt att vara tydlig med var information kommer ifrån, samt att ange detta så fullständigt att en läsare kan hitta den källa man refererar till. En läsare som vill hitta den källa man refererar till ska kunna göra det baserat på den information som man ger. Det betyder att det är olika information som krävs för olika sorters referenser. För en bok behöver man titel, författare, förlag och årtal, medan man för en tidskriftsartikel typiskt behöver titel, författare, namn på tidskrift, volym, nummer, förlag och årtal. Nedan ges exempel på vilken information som är lämplig att ge för ett antal vanliga källtyper:

- Bok: författare, publiceringsår, titel, upplaga, förlag.
- Tidskriftsartikel: författare, publiceringsår, artikelns titel, tidskriftens namn, volym och nummer för tidskriftsnumret, sidnummer för artikeln.
- Konferensartikel: författare, publiceringsår, artikelns titel, konferensens namn, geografisk plats och datum för konferensen, sidnummer för artikeln.
- Teknisk rapport: författare, publiceringsår, rapportens titel, rapport-nummer, utgivande institution.
- Information från internet: författare eller utgivande institution, publiceringsår, titel, datum för senaste access, URL.

7.2.2 Hur ska informationen presenteras?

Som det beskrivs ovan så finns det många olika sätt att formatera referenslistan. Förlag har ofta sina egna riktlinjer och ofta finns det riktlinjer i mallar för t ex tekniska rapporter på ett företag eller examensarbeten på en institution. Då är det helt enkelt bara att följa dem. En grundläggande regel är dock att alltid vara så konsekvent som möjligt. Om man t ex skriver namnen på en författare med förnamnet först och efternamnet efter det så ska man inte ha namnen i andra ordningen för någon annan författare någon annan stans i referenslistan, om man anger hela förnamnet för en författare så ska man göra det för alla författare och inte bara initialerna för en annan författare, eller om man anger året sist i en referens så ska man göra det för alla referenser.

Referenslistan presenteras oftast i bokstavsordning för författarna, som t ex referenserna i detta kapitel. Ibland väljer man dock att presentera dem i den ordning som de refereras i texten.

Ett exempel på riktlinjer för siffersystemet kommer från IEEE och deras "Templates for Transactions"¹. Exempel på hur man kan presentera referenser i referenslistan enligt deras regler (på engelska) är

- Bok: W.-K. Chen, *Linear Networks and Systems*. Belmont, CA, USA: Wadsworth, 1993.
dvs författare, titel, var förlaget finns, förlag, samt årtal.
- Tidskriftsartikel: E. P. Wigner, "Theory of traveling-wave optical laser," *Phys. Rev.*, vol. 134, pp. 635—646, 1965.
dvs författare, artikelns titel inom citationstecken, tidskriftens namn kursiverad, volym, sidnummer, samt årtal.
- Konferensartikel: D. B. Payne and J. R. Stern, "Wavelength-switched passively coupled single-mode optical network," in *Proc. IOOC-ECOC*, Boston, MA, USA, 1985, pp. 585—590.
dvs författare, artikelns titel inom citationstecken, vilken konferens artikelsamling (eng. "proceedings") artikeln finns i, var konferensen höll, samt årtal.
- Teknisk rapport: R. J. Hijmans and J. van Etten, "Raster: Geographic analysis and modeling with raster data," R Package Version 2.0-12, Jan. 12, 2012. [Online]. Available: <http://CRAN.R-project.org/package=raster>
dvs författare, rapportens titel inom citationstecken, beskrivning, datum och årtal, samt URL om rapporten finns tillgänglig på internet.

Studera gärna informationen från IEEE för mer information om detaljer för fler typer av referenser. Här i detta kompendium har vi tagit bort några detaljer från referenserna, men de är fortfarande korrekta. Man kan observera att det finns många förkortningar i referenserna, vilket är bra om man vill spara plats i den text man skriver. Ofta skriver man dock ut hela namnen, t ex på de tidskrifter man refererar.

¹<http://ieeeauthorcenter.ieee.org/create-your-ieee-article/use-authoring-tools-and-ieee-article-templates/ieee-article-templates/>

7.2.3 Verktyg: Ordbehandlare som t ex Word

Om man skriver i en ordbehandlare som t ex MS Word och om man skriver en förhållandevis kort rapport och vill ha en referenslista enligt nummersystemet så kan man ofta klara sig bra genom att helt enkelt skriva referenslistan som en numrerad lista i slutet av rapporten och sedan referera till denna i texten. Om man "hårdkodar" numren på referenserna kan det dock bli jobbigt att lägga in referenser mitt i referenslistan eftersom man då måste ändra numren på alla referenser som refererar någon källa efter den man lagt in. En lösning i MS Word är att använda korsreferenser till en numrerad lista som uppdateras automatiskt.

Om det går kan man också använda parentesssystemet, vilket kan vara lättare eftersom det inte kräver att man håller reda på nummer i en lista. Det finns också särskilda verktyg som kan användas, t ex EndNote², för att hålla reda på referenser i en databas och få dem i valfritt format.

7.2.4 Verktyg: LaTeX och BibTeX

Om man skriver i LaTeX så finns det två vanliga sätt att hålla reda på referenser, den enklaste med `thebibliography`-omgivningen och den lite mer avancerade metoden med BibTeX-verktyget³.

Om man använder `thebibliography`-omgivningen så skriver man alla referenserna i denna typ av omgivning i slutet av rapporten enligt

```
\begin{thebibliography}{widest-label}
  \bibitem{cite_key}
    literature citation ...
  ....
\end{thebibliography}
```

och man citerar med kommandot `\cite{cite_key}` i texten. Det finns alltså en `bibitem` för varje referens i referenslistan och LaTeX håller reda på numren till en så att det blir rätt när man kompilerar. För mer information se t ex [31] eller [11].

Det är givetvis bra att LaTeX håller reda på nummer i referenserna till en, men man måste själv fortfarande sortera dem i bokstavsordning och man måste själv se till att man är konsekvent och beskriver alla referenser på samma sätt. Ett verktyg som kan underlätta är BibTeX. Om man använder detta så lagrar man alla referenser i en databas, en `.bib`-fil. Denna fil innehåller alla referenser enligt BibTeX format, se figur 7.2. Alla poster börjar med ett `@`-tecken och en referenstyp, t ex `book`, `article`, eller `inproceedings`. Sedan följer en referens-tag och efter det ett antal informationsfält som är lite olika för olika referenstyper. En bra beskrivning finns på den engelska Wikipedia-sidan för BibTeX⁴. Referensposterna i `.bib`-filen kan komma i vilken ordning som helst. Ett utdrag ur en `.bib`-fil kan t ex se ut som i figur 7.1.

Mer om hur `.bib`-filen ska skrivas och exakt hur referenserna i den ska formateras enligt BibTeX format finns t ex på CTANs sidor⁵. Några vanliga refe-

²<http://endnote.com>

³Man kan även använda det nyare BibLaTeX-systemet som innehåller fler funktioner och är mer avancerat än BibTeX. I grunden används det dock på ungefär samma sätt och databasen (dvs `.bib`-filen) ser likadan ut.

⁴<http://en.wikipedia.org/wiki/BibTeX>

⁵<https://www.ctan.org/pkg/bibtex> [37]

```

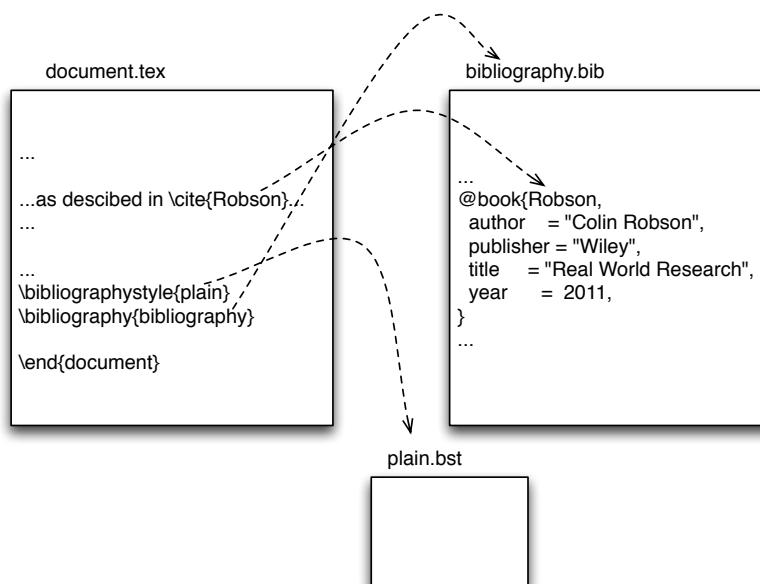
...
@inproceedings{Georges07,
  author      = {Andy Georges and Dries Buytaert and
                Lieven Eeckhout},
  title       = "Statistically Rigorous Java Performance Evaluation",
  booktitle   = "Proceedings of the 22nd annual ACM SIGPLAN conference on
                Object-oriented programming systems and applications
                (OOPSLA '07)",
  year        = 2007,
  pages       = {57-76},
}

@article{Glass94,
  author      = {Robert L. Glass},
  title       = {The Software Research Crisis},
  journal     = {IEEE Software},
  volume      = 11,
  pages       = {42-47},
  year        = 1994,
}

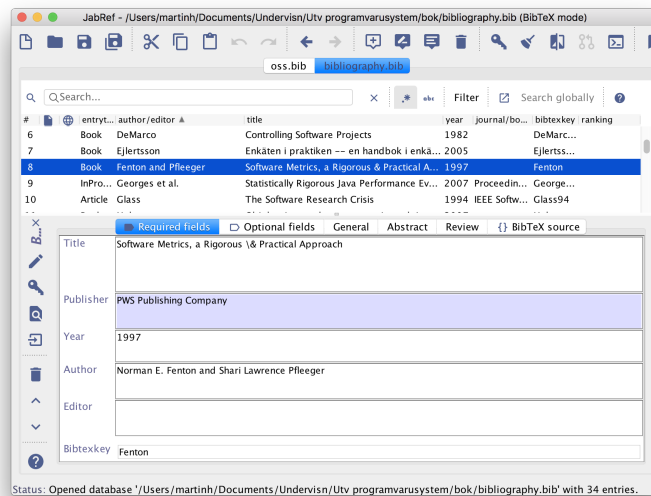
@book{Humphrey95,
  author      = {Watts S. Humphrey},
  title       = {A Discipline for Software Engineering},
  publisher    = {Addison-Wesley},
  year        = 1995,
}
...

```

Figur 7.1: Utdrag ur en .bib-fil



Figur 7.2: BibTeX, exempel



Figur 7.3: Exempel på verktyg för att arbeta med .bib-filer

renstyper beskrivs i tabell 7.2. Man kan såklart arbeta med sina .bib-filer i en vanlig texteditor, men det går också att använda något av de verktyg som finns för att underlätta arbetet. Ett exempel på denna typ av verktyg är JabRef⁶, se figur 7.3. Med det kan man arbeta i ett grafiskt användargränssnitt och man får hjälp med vilken information som behövs för vilken referenstyp.

Formatet för att ge information till de element som krävs kan t ex ses i figur 7.2. Några saker att tänka på är:

1. Man kan ge informationen inom citationstecken, t ex `author = "Colin Robson"`. Man kan också använda {-parenteser, t ex `author = {Colin Robson}`.
2. Namn i författarlistan anges ungefär som man brukar ange namn i "vanlig text", t ex på någon av formerna
 - Colin Robson
 - Robson, Colin
 - C. Robson
 - Robson, C.
3. Om det finns flera namn i författarlistan så lägger man in ett `and` mellan varje författare, t ex `author = {Andy Georges and Dries Buytaert and Lieven Eeckhout}`. Observera att man inte kan avgränsa med komma-tecken mellan författarna, det tror ju BibTeX är avgränsara mellan olika namn på en person, se punkt 2.

⁶<http://www.jabref.org>

Tabell 7.2: BibTeX-element för några vanliga referenstyper (baserat på [37])

Referenstyp	Fält
@book	Obligatoriska: <code>author/editor</code> , <code>title</code> , <code>publisher</code> , <code>year</code> Frivilliga: <code>volume/number</code> , <code>series</code> , <code>address</code> , <code>edition</code> , <code>month</code> , <code>note</code> , <code>key</code>
@article (artikel i tidskrift)	Obligatoriska: <code>author</code> , <code>title</code> , <code>journal</code> , <code>year</code> , <code>volume</code> Frivilliga: <code>number</code> , <code>pages</code> , <code>month</code> , <code>note</code> , <code>key</code>
@inproceedings (konferensartikel)	Obligatoriska: <code>author</code> , <code>title</code> , <code>booktitle</code> , <code>year</code> Frivilliga: <code>editor</code> , <code>volume/number</code> , <code>series</code> , <code>pages</code> , <code>address</code> , <code>month</code> , <code>organization</code> , <code>publisher</code> , <code>note</code> , <code>key</code>
@incollection (bokkapitel i samlingsverk)	Obligatoriska: <code>author</code> , <code>title</code> , <code>booktitle</code> , <code>publisher</code> , <code>year</code> Frivilliga: <code>editor</code> , <code>volume/number</code> , <code>series</code> , <code>pages</code> , <code>address</code> , <code>month</code> , <code>organization</code> , <code>note</code> , <code>key</code>
@techreport	Obligatoriska: <code>author</code> , <code>title</code> , <code>institution</code> , <code>year</code> Frivilliga: <code>type</code> , <code>number</code> , <code>address</code> , <code>month</code> , <code>note</code> , <code>key</code>
@mastersthesis (examensarbete)	Obligatoriska: <code>author</code> , <code>title</code> , <code>school</code> , <code>year</code> Frivilliga: <code>type</code> , <code>address</code> , <code>month</code> , <code>note</code> , <code>key</code>
@phdthesis (doktorsavhandling)	Obligatoriska: <code>author</code> , <code>title</code> , <code>school</code> , <code>year</code> Frivilliga: <code>type</code> , <code>address</code> , <code>month</code> , <code>note</code> , <code>key</code>
@misc ("övrigt")	Obligatoriska: — Frivilliga: <code>author</code> , <code>title</code> , <code>howpublished</code> , <code>month</code> , <code>year</code> , <code>note</code> , <code>key</code>

- Om man vill berätta för BibTeX att inte följa sina vanliga regler för stor och liten bokstav (dvs de regler som finns i `.bst`-filen) så kan man göra det med `{}`-parenteser inuti textsträngen, t ex `title = {The Not So Short Introduction to {LATEX2e}}` (som vi gjort för referens [31] i referenslistan i detta kompendium).

I det dokument man skriver anger man vilken `.bib`-fil som ska användas med kommandot `\bibliography` där man vill ha referenslistan, dvs i slutet av dokumentet. I samband med det anger man också med kommandot `\bibliographystyle` vilken stil-fil (`.bst`-fil) som ska användas. I figur 7.2 anges filen `plain.bst` som är en av de standardstilar som följer med alla installationer. Om man använder en av standardfilerna så hittar systemet den utan att man lägger den i sin arbetskatalog, men om man använder en `.bst`-fil som inte ingår i ens LaTeX-installation så måste man lägga den så att systemet hittar den, t ex i samma katalog som den kompillerade `.tex`-filen. Stil-filen beskriver hur referenslistan ska formateras, dvs vilken ordning referenserna ska listas i, hur varje post ska formateras, om det ska vara hela namn eller bara initialer, etc. De förlag, som t ex IEEE, som har regler och riktlinjer för hur referenser ska formateras tillhandahåller ofta `.bst`-filer som man kan använda för att få rätt format. Det är ganska ovanligt att man själv skriver `.bst`-filer.

När man använder BibTeX så gör man det ofta med följande sekvens:

```
> latex document
> bibtex document
> latex document
> latex document
```

Först kör man LaTeX, vilket skapar en hjälpfil som beskriver vilka referenser som finns i dokumentet från `\cite`-kommandona. Sedan kör man BibTeX för att skapa en ny hjälpfil (`.bbl`) med bibliografisk information, ungefär som när man använder `thebibliography`-omgivningen. BibTeX skapar bara referenser för de poster i `.bib`-filen som man verkligen refererar. Sedan kör man LaTeX igen för att få in referensinformationen från den andra hjälpfilen och för att få rätt nummer på alla referenser i texten.

Användningen av BibTeX visar på att det finns ett antal olika sorters information man måste hålla reda på och besluta om när det gäller referenshantering

- Man måste bestämma vilka referenser man ska referera var i sin rapport, dvs den information man ger i `\cite`-kommandon
- Man måste presentera och hålla reda på rätt sorts information om varje typ av referens, dvs den information som man lagrar i `.bib`-filen
- Man måste presentera informationen i referenserna på ett enhetligt sätt, på rätt format enligt de instruktioner man kan ha fått, i referenslistan, dvs den information man ger i `.bst`-filen

7.2.5 Parentessystemet

Vi går inte igenom parentessystemet i detalj här utan hänvisar till andra källor, t ex [3] och [1]. Kort kan man dock säga att man refererar i texten genom att ange författare och årtal för att läsaren ska kunna hitta rätt referens i referenslistan som då inte är numrerad. Man kan referera antingen direkt till författaren eller till verket som sådant. Ett exempel på det första alternativet är

”Robson (2011) presents how experiments are conducted.”

och ett exempel på en referens till verket som sådant är

”Experiments are often conducted in research (Robson, 2011).”

Här kan man hitta referensen i referenslistan eftersom det bara finns en bok av Robson från 2011. Om det är mer än en författare anger man alla i referenslistan, men max tre i referensen i texten:

”Software metrics are important (Fenton and Pfleeger, 1997).”

”Oetiker et al. (2016) show how LaTeX can be used to write technical reports.”⁷

Om man har mer än en referens i referenslistan av samma författarkonstellation från ett år så måste man skilja dem åt på något sätt, annars kan man inte veta vilken referens man menar. Man brukar göra det genom att sätta en bokstav, a, b, c, efter året både i referensen och i referenslistan, t ex ”(Robson, 2011a)” respektive ”(Robson, 2011b)” om det hade funnits två referenser av Robson från 2011 i referenslistan.

Om man vill använda BibTeX för att göra referenser enligt parentessystemet så kan man använda `natbib`-paketet. I stället för att citera med `\cite{}` som vanligt så använder man `\citet{}` för att referera direkt till författaren eller författarna och `\citep{}` för att referera till verket som sådant.

⁷”et al.” är en förkortning av ”et alia”, som betyder ”med andra” på latin

7.3 Checklista

Följande checklista kan användas för att bedöma kvalitén av en rapporterad studie.

1. Är målsättningen med den rapporterade studien tydlig? Detta kan t ex beskrivas som ett antal forskningsfrågor.
2. Är den övergripande metoden som används tydligt beskriven och motiverad?
3. Är det tydligt vilken data som samlats in och hur den har samlats in?
4. Presenteras tillräckligt mycket av den data som samlats in?
5. Är analysmetoderna rätt valda och lämpliga för studien?
6. Är analysen korrekt genomförd och kan man lita på resultaten?
7. Finns det en beskrivning av hur validitetshot tagits om hand och en diskussion om eventuella kvarvarande validitetshot?
8. Är slutsatserna tydliga och svarar de på den målsättning som finns med studien? (T ex, om det finns forskningsfrågor, besvaras alla frågor?)
9. Görs referenser till trovärdiga källor i tillräckligt utsträckning (dvs det finns inga påståenden som saknar grund i rapporten)?
10. Är språket lätt att förstå och av tillräckligt hög kvalitet?
11. Följer rapporten en lämplig struktur (i denna kurs enligt kapitel 7.1)?

Kapitel 8

Exekveringsmiljö

8.1 Inledning

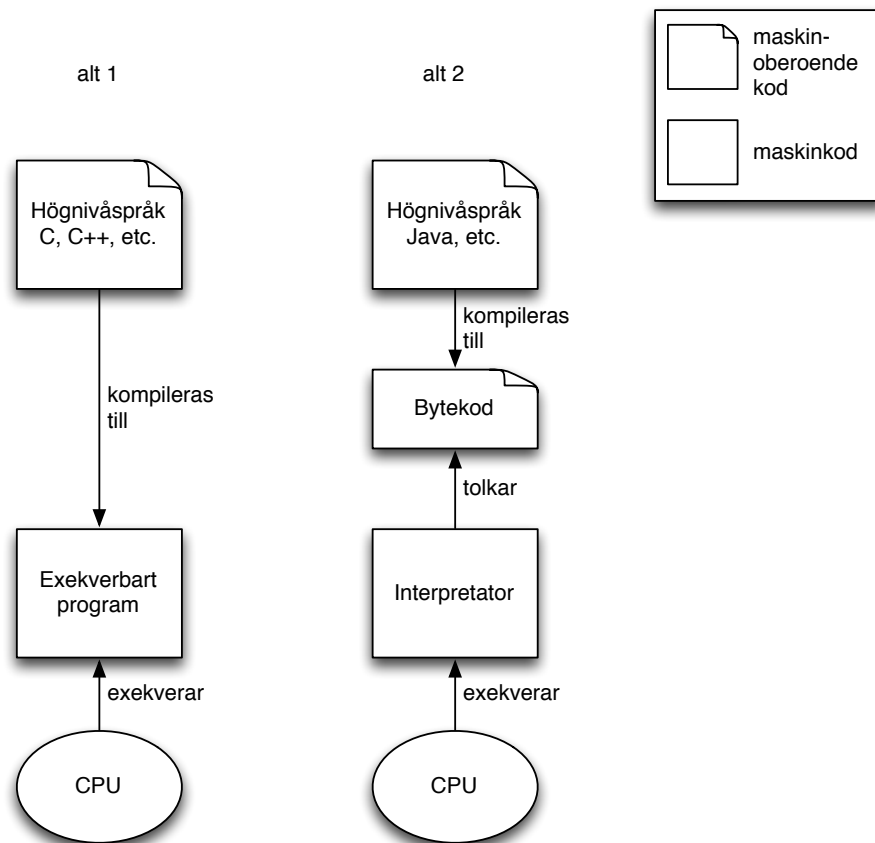
En rimlig fråga man kan ställa sig är varför ett program inte tar lika lång tid att exekvera varje gång. För att förstå detta måste man veta lite om hur program exekveras på datorn och vad som mer händer i datorn.

8.2 Exekveringssystem

Det som exekverar program i en dator är CPU:n, eller egentligen en av kärnorna i CPU:n om det finns mer än en kärna. På ett mycket förenklat sätt kan man säga att den läser maskinkodsinstruktioner i minnet och utför dem. Dessa maskinkodsinstruktioner är olika för olika processorer, dvs olika för olika datorer. Det är t ex olika sorters processorer i en mobiltelefon och en laptop. Dessutom måste koden vara anpassad till just den enhet den exekveras på med rätt uppsättning minne och andra saker.

Maskinkodsinstruktionerna är anpassade till datorn och inte särskilt lätta att förstå för människor. Man kan programmera i Assembler, vilket är en läsbar form av maskinkod, men det är inte särskilt vanligt. Man programmerar i stället i högnivåspråk som är gjorda för att människor ska förstå dem på ett bra sätt. Eftersom programmen som är skrivna i högnivåspråk behöver de inte anpassas lika mycket efter vilken CPU de ska exekveras på. Eftersom de inte är lika CPU-beroende säger man att de är mer ”portabla”. Ett av målen med Java har t ex varit att få ett portabelt program, vilket medför att man ofta kan flytta ett Java-program från en plattform, dvs kombination av CPU och operativsystem, till en annan utan att göra några förändringar. Man kan alltså säga att program skrivna i högnivåspråk är mer portabla än program skrivna i maskinspråk. Det är också olika hur portabla olika högnivåspråk är. Portabilitet är t ex en tydligare drivkraft för Java än för C. Eftersom maskinkod måste vara anpassad till den typ av dator den exekveras på säger man att den är CPU-beoende och det är alltså inte möjligt att enkelt flytta den från en typ av dator till en annan.

Eftersom en dator bara kan exekvera maskinkod så måste innehållet i programmen som är skrivna i ett högnivåspråk omvandlas till exekverbar maskinkod på något sätt.



Figur 8.1: Omvandling från högnivåspråk till maskinkod

Två olika angreppssätt visas i figur 8.1. I alternativ 1 omvandlas högnivåprogrammet till maskinkod med en kompilator. Det är t ex så som program skrivna i C och C++ kompileras och exekveras. En fördel med detta angreppssätt är att maskinkodsprogrammet blir helt självständigt och kan exekveras på datorn utan några andra program. En nackdel är att programmet inte blir så portabelt utan måste kompileras för varje plattform [2].

Ett annat alternativ visas som alternativ 2 i figuren. Här kompileras högnivåspråket i stället till ett format, bytekod, som kan sägas vara mitt emellan högnivåspråket och maskinkod. Det finns sedan en "interpretator" som kan tolka denna kod och på så sätt utföra instruktionerna. Detta betyder att den kompilerade koden är förhållandevis portabel och kan köras på många olika sorters datorer. Det som är datorberoende är interpretatorn som det måste finnas en för varje plattform.

Det ska sägas att det också finns ett antal varianter av alternativ 2, t ex att interpretatorn tolkar högnivåspråket direkt, utan att man kompilerar till någon bytekod. Detta är t ex hur R fungerar. Om man inte kompilerar till en bytekod, utan låter interpretatorn läsa högnivåkoden så blir det långsammare.

Alternativ 2 som i figuren motsvarar hur Java fungerade i början när Java var nytt. Fördelen med detta alternativ är att koden blir mycket portabel och går att flytta mellan många olika plattformar. Nackdelen är att exekveringen blir förhållandevis långsam, eftersom det är en interpretator som måste tolka instruktionerna i bytekoden och utföra dem. Det hade blivit snabbare om koden hade kompilerats till maskinkod och exekverats som ett eget program.

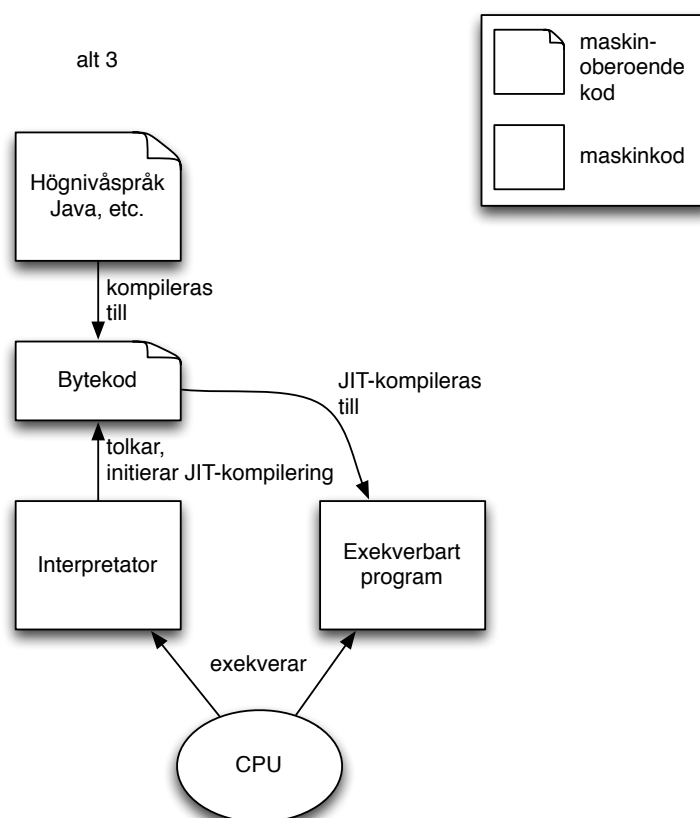
För att komma tillrätta med att alternativ 2 är långsamt har man infört vad man kallar "just-in-time"-kompilering (JIT), vilket presenteras i figur 8.2.

Precis som i alternativ 2 kompileras högnivåspråket till bytekod som tolkas av en interpretator. Skillnaden är att interpretatorn kan beordra att små delar av bytekoden ska kompileras till maskinkod precis innan ("just-in-time") instruktionerna ska genomföras och så kan de exekveras som ett vanligt program i stället. Fördelen med detta angreppssätt är att programmet fortfarande är lika portabelt, men nu avsevärt snabbare än med alternativ 2. Idag används JIT-kompilering i de flesta Java exekveringssystem, dvs interpretatorerna för Java.

En ytterligare fördel med alternativ 3 är att eftersom koden kompileras så kan optimeringar av koden genomföras. T ex kan kompilatorn byta ut loopar mot konsekutiva instruktioner när detta är lämpligt. Det finns flera olika sätt att optimera kod med en kompilator och det skulle leda för långt att gå igenom ens en bråkdel här, utan detta får skjutas till senare kurser¹. JIT-kompilatorn kan dock inte göra all optimering på en gång. Dels skulle det ta lång tid och dels så har den inte all kunskap om hur programmet körs i början. Den vet t ex inte vilka klasser med en viss superklass som används om den bara vet vilken superklass det är. Kompilatorn kan optimera lite för varje gång en del av programmet körs. Det betyder att ett program blir snabbare och snabbare för varje gång det exekveras. Optimeringar görs fram till en punkt när inte fler optimeringar lönsas sig. Detta kallas adaptiv optimering, dvs att kompilatorn anpassar optimeringen till den aktuella programkörningen.

I Java är det som sagt standard att använda JIT-kompilering med optimeringar vid exekvering. Man kan dock stänga av JIT-kompileringen om man vill

¹t ex EDAN65 Kompilatorer och EDA230 Optimerande kompilatorer



Figur 8.2: "Just-in-time"-kompilering

med flaggan `-Xint` till kommandot `java`, t ex

```
> java -Xint fil
```

Teknikerna för JIT och adaptiv optimering togs ursprungligen fram för dynamiskt evaluerade språk som Smalltalk och Lisp, men används idag, som nämnts ovan, i Java och andra dynamiska språk som C# och JavaScript.

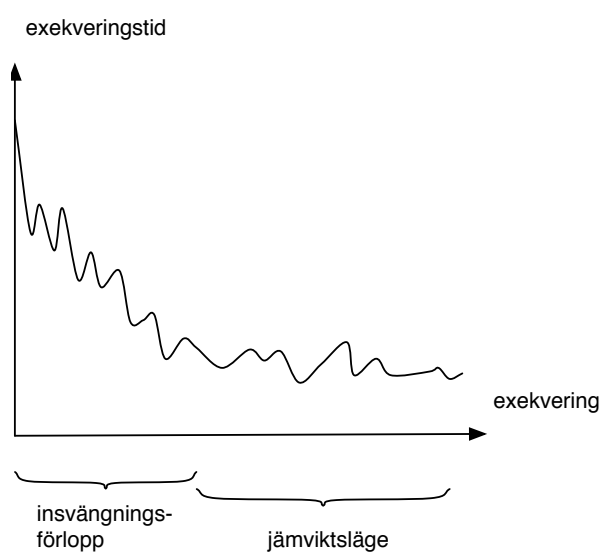
8.3 Övriga anledningar

En annan anledning till att program tar olika lång tid att exekvera olika gånger man exekverar det är att det finns "garbage collection". När ett program körs så krävs det minne. Det allokeras t ex det minne då nya objekt skapas med `new`. Det minne som reserverats kan släppas igen när det inte används mer. Detta sköts av "garbage collection" som då och då går igenom det minne som reserverats och ser vad som kan släppas igen. Detta tar givetvis tid, vilket kan ses genom att andra saker som görs av programmet då och då blir långsammare.

Det finns fler anledningar till att det tar olika lång tid olika gånger ett program exekveras. Inte minst så finns det fler program i datorn som exekveras och processorn delas mellan olika program.

8.4 Exekveringstid

Det tar alltså olika lång tid för ett java-program att exekvera olika gånger. Bland annat på grund av "just-in-time"-kompilatorn går det förhållandevis långsamt första gången en del av ett program körs, men sedan snabbare och snabbare om koden körs flera gånger. Om man har en del av ett program som körs flera gånger, t ex en sorteringsfunktion eller inmatningsfunktion så tar den lite längre tid första gången än den tar när den körts flera gånger. När den körts många gånger planar exekveringstiden ut runt ett jämviktsvärde, även om det runt detta jämviktsvärde finns slumpmässiga variationer. Detta kan illustreras schematiskt som i figur 8.3. Figuren försöker åskådliggöra att det går snabbare och snabbare tills ett jämviktsläge uppnås. Exakt hur det ser ut i ett riktigt program beror på det programmet och hur det exekveras. Detta är bara en illustration.



Figur 8.3: Insvängningsförlopp för exekveringstid

Kapitel 9

Prestandamätning

Det finns flera olika aspekter av prestanda av mjukvara, t ex hur lång tid ett program tar att exekvera, hur mycket minne det kräver, eller hur mycket ström ett inbyggt system drar då ett program körs. En viktig aspekt som behandlas mer ingående i detta kompendium är exekveringstid.

Denna typ av prestanda märker man tydligt när man använder en dator och man vill gärna att det ska gå så snabbt som möjligt när man använder datorn. När man t ex startar webb-läsaren vill man inte att det ska ta för lång tid, när man startar själva datorn så vill man att den med sitt operativsystem ska komma igång så snabbt som möjligt, etc. Om man tittar på den tekniska prestanda som presenteras för datorer så är det ofta angivet hur snabba de är i termer av klockfrekvens, dvs antal klockcykler som processorns inbyggda klocka gör per sekund, t ex "4,1 Ghz". Denna frekvens påverkar givetvis hur många instruktioner processorn kan exekvera per sekund. Är detta då tillräckligt för att skapa sig en bild över hur snabb en dator och ett program på en dator är? Nej, det är det ju givetvis inte. Det är ju inte tydligt hur många klockcykler varje instruktion kräver eller hur mycket som "görs" i varje instruktion. Dessutom exekveras ofta mer än en instruktion åt gången i en processor, vilket också gör det svårt att förstå vad två olika klockcykler på olika datorer betyder.

Man kan givetvis diskutera och mäta prestanda för olika system eller delar av system. Man kan t ex betrakta prestanda hos en dator, enligt diskussionen ovan. Detta är det vanliga om man ska införskaffa en dator för ett ändamål som dator att ha hemma, kontorsdator, som en del i ett helt system, etc. Man kan också betrakta prestanda hos ett program. Ett program kan utvecklas för att bli snabbare eller långsammare beroende på hur man designar det med olika val av algoritmer och tekniska lösningar. Det är främst denna aspekt som kommer att diskuteras i detta kompendium. En typ av program som har betydelse för prestandan är Java:s exekveringssystem som påverkar hur snabbt andra program kan exekvera. Ofta är man också intresserad av prestandan hos ett helt system bestående av datorer, nätverk och andra komponenter. Här talar man ofta om egenskaper som svarstid och genomströmning. Man kan alltså dela upp system i olika nivåer:

- Övergripande systemnivå: Detta är den översta abstraktionsnivån som behandlar prestanda av hela system bestående av datorsystem, nätverk, databaser, etc. Exempel kan vara telekommunikationssystem, en bil med

CPU:er och nätverk, etc. Övergripande systemprestanda beskrivs t ex i termer som svarstid, antal avklarade uppgifter per tidsenhet etc. och påverkas av delsystemens prestanda.

- Datorsystemnivå: Denna nivå betraktar prestandan av ett datorsystem, t ex en dator för kontorsbruk. Här använder man t ex mått som klockfrekvens.
- Programnivå: Denna nivå betraktar prestandan av ett datorprogram, eller en del av ett datorprogram, som t ex ett ordbehandlingsprogram, en implementering av en sorteringsalgoritm, en kompilator, eller liknande. Även en exekveringsmiljö för Java kan räknas hit.

Frågan är då vad vi menar med ett bra mått på prestanda och vilka mått som brukar användas.

9.1 Egenskaper hos prestandamått

Lilja [26] listar ett antal egenskaper man eftersträvar hos prestandamått främst på datorsystemnivå, t ex:

- Linjäritet: Man har sett att människan ofta uppskattar linjära mått. Det vill säga, om en mätningen ger ett dubbelt så bra resultat som en annan mätning så ska det också betyda att det system som det bättre måttet kommer ifrån ska vara dubbelt så bra som det andra systemet. Det är dock inte helt nödvändigt att mått ska vara linjära, utan ibland kan även andra mått användas, t ex logaritmiska mått som dB i andra sammanhang.
- Tillförlitlighet: Med tillförlitlighet menar vi att om ett mått säger att ett system är bättre än ett annat så ska det verkligen vara det också i praktiken när det används.
- Lätt att mäta: Om mått är svåra att mäta i praktiken innebär det stora risker för felmätningar, vilket givetvis inte är bra. Det är därför bra om de mått man definierar inte är för svåra att mäta i praktiken.
- Konsistens: Detta innebär att de enheter som mäts med ett mått är de samma och jämförbara för olika system. Dvs, om två olika system får samma värde med måttet så ska de också vara lika bra.
- Oberoende: Definitionen av måttet ska vara oberoende av vilka system som undersöks. Detta är viktigt eftersom många tillverkare av datorer är måna om att deras datorer ska komma väl ut i jämförelser och att måtten därmed inte ska vara bättre eller sämre för olika tillverkares datorer.

9.2 Exempel på prestandamått för datorsystem

Det finns ett antal exempel på prestandamått som föreslagits och använts.

9.2.1 Klockfrekvens

Detta är som beskrivs innan ett mått på hur många klockcykler som processorns inbyggda klocka gör per sekund. Detta är givetvis relaterat till hur många instruktioner som utförs per tidsenhet, men det tar inte hänsyn till hur många klockcykler som krävs för varje instruktion. Måttet används för datorsystem, främst eftersom det är så enkelt att redogöra för för olika datorsystem. Man ska dock vara medveten om att en stor mängd andra aspekter påverkar upplevelsen av att använda en dator, som t ex lagringsenhetens snabbhet, minnets snabbhet, programmets snabbhet, etc. Om man jämför två datorer ska man alltså vara medveten om att måttet inte är linjärt eftersom en dator med 10% snabbare klockfrekvens inte behöver upplevas som just 10% snabbare.

9.2.2 MIPS (Million Instructions Per Second)

Detta mått används för att beskriva hur många instruktioner en processor i ett datorsystem exekverar per sekund. Detta kommer till rätta med ett av problemen med att mäta klockfrekvens, dvs att olika instruktioner kräver olika många klockcykler. Problemet med MIPS-måttet är dock att varje instruktion inte genererar lika mycket nyttigt arbete i olika sorters processorer.

9.2.3 MFLOPS (Millions of floating-point instructions per second)

Detta är ett mått på snabbheten av ett datorsystem som istället för att fokusera på endast instruktioner fokuserar på hur många flyttalsoperationer som datorn kan genomföra per sekund.

Med flyttal menas ett reellt tal med begränsat antal siffror [14], t ex `float` och `double` i Java. Detta är ett mer "rättvist" mått än MIPS eftersom flyttalsoperationer är mer lika mellan olika datorer än vad instruktioner är. Nackdelen är att man endast studerar hur ett datorsystem beter sig just för flyttalsoperationer och inte för något annat som kanske är mer relevant för den användning man tänkt sig. För t ex en webbserver eller en kontorsdator är kanske inte detta den mest troliga användningen.

9.2.4 Testuppsättningar

Eftersom användning av datorsystem innefattar många olika sorters användning så har man tagit fram uppsättningar av testprogram som kan användas för prestandautvärdering av datorsystem.

En uppsättning definieras av SPEC¹ (The Standard Performance Evaluation Corporation). De definierar ett antal testramverk som består av olika program som körs och för vilka exekveringstiden mäts. Tanken är att man ska exekvera en standarduppsättning av olika program som är typiska för en användning av datorn och sedan räkna ut ett medelvärde av tiden det tar. De levererar hela ramverk som innehåller program, inklusive funktionalitet för att göra mätningar av exekveringstid.

¹<https://www.spec.org>

Ett ramverk från SPEC är anpassat för prestandamätning av Java exekveringsmiljöer, `SPECjvm2008`. Det innehåller program ("benchmarks") för att utvärdera exekveringstid för

- kompilering av java-filer
- komprimering av data
- kryptering och dekryptering av data
- användning av en java-databas för affärssystem
- mp3-avkodning
- numeriska metoder, baserat på SciMark 2.0²
- serialisering och av-serialisering av Java-objekt
- grafisk visualisering
- XML-hantering

Dessutom innehåller ramverket ett program för att mäta hur lång tid de ingående programmen tar första gången de körs.

Man kan se att SPEC innehåller flera olika sorters program och att de försökt täcka in olika sorters användning av exekveringsmiljöer. Det hade inte räckt till att endast undersöka t ex hur snabbt man kan göra sortering med en miljö, utan det är viktigt att täcka flera olika användningsområden. Man kan också se att man har tänkt att delar utanför exekveringsmiljön inte ska påverka resultatet i för stor omfattning, som t ex att skriva till disk.

9.3 Prestanda av program

9.3.1 Inledning

Exekveringstid är en viktig egenskap hos mjukvara som utvecklas och man ställer ofta krav på hur lång den får vara för produkter som ska utvecklas, se t ex [21]. Det är därför viktigt att kunna mäta hur lång exekveringstid som krävs av ett program eller en del av ett program.

När man ska mäta exekveringstid av ett program eller del av ett program måste man först bestämma om man är intresserad av hur lång tid det tar första gången man kör det eller om man är intresserad av hur lång tid det tar efter insvägningsförloppet. Om det program man ska mäta på bara ska köras en gång per uppstart av programmet av de som ska använda programmet så är det mest intressant att veta hur lång tid det tar första gången. Om man däremot ska mäta på en del av ett program som kommer att köras många gånger varje gång programmet startats av de som använder det så är man förmodligen mer intresserad av hur lång tid det tar efter insvägningsförloppet.

²<http://math.nist.gov/scimark2/>

9.3.2 Praktisk mätning

Det finns olika sätt att mäta exekveringstid, men ett vanligt sätt är att helt enkelt läsa av klockan i datorn före och efter det som man vill mäta tiden för. Ett sätt att göra detta är enligt följande:

```
...
long startTime = System.nanoTime();
// measured code
long endTime = System.nanoTime();
long measuredTime = endTime - startTime;
...
```

Här använder man alltså den statiska metoden `.nanoTime()` i klassen `System`³. den returnerar "the current value of the most precise available system timer, in nanoseconds". Värdet som sådant säger alltså ingenting, men det går att använda skillnaden mellan två värden. Man skulle också kunna använda metoden `currentTimeMillis()`, som returnerar "the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC", men det ger inte samma noggrannhet som `.nanoTime()`.

Den tid som mäts på detta sätt innehåller även tiden det tar att göra själva mätningen, dvs tiden det tar att exekvera `System.nanoTime()`. Man kan kompensera för detta, men här antar vi att den tiden är så kort att det inte behövs.

Som vi ser i kapitel 8 så tar det olika lång tid för ett java-program att exekvera olika gånger. Bland annat på grund av "just-in-time"-kompilatorn går det förhållandevis långsamt första gången en del av ett program körs, men sedan snabbare och snabbare om koden körs flera gånger. Till slut planar tiden ut runt ett jämviktsvärde, även om det runt detta jämviktsvärde finns slumpmässiga variationer.

9.3.3 Exekveringstiden första gången man kör ett program

Om man är intresserad av att veta hur lång exekveringstiden är första gången man kör ett program kan man mäta enligt följande (se t ex [12]):

1. Mät exekveringstiden för programmet med olika uppstarter av exekveringsmiljön varje gång programmet körs. Detta resulterar i p mätningar x_i där $1 \leq i \leq p$.
2. Beräkna medelvärdet

$$\bar{x} = \frac{1}{p} \sum_{i=1}^p x_i$$

och konfidensintervallet för medelvärdet enligt kapitel 6.2.3.

Uppstart av exekveringsmiljön innebär att man startar java-programmet, t ex med kommandot `java` i en terminal.

³t ex <https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/System.html>

9.3.4 Exekveringstid, jämviktsvärde

Om man vill veta jämviktsvärdet så måste man mäta och se hur långt insvägningsförloppet är, dvs hur många gånger man måste köra delen av programmet för att tiden ska variera runt jämviktsvärdet och inte längre sjunka mot jämviktsvärdet. Man måste alltså hitta var gränsen går mellan insvägningsförloppet (se figur 8.3) och jämviktsvärdet och slänga bort mätningarna man gjort på insvägningsförloppet innan man räknar ut medelvärdet.

En annan svårighet när man gör dessa mätningar är att det inte säkert blir samma jämviktsvärde varje gång man startar exekveringsmiljön. Detta innebär att man måste starta exekveringsmiljön flera gånger och varje gång slänga mätningarna från insvägningsförloppet och räkna ut jämviktsvärdet. Det betyder att en metod som följande kan användas [12]:

1. Studera p olika uppstarter av exekveringsmiljön, där man vid uppstart i gör s_i mätningar av samma sak. Låt x_{ij} beteckna resultatet av mätning j vid uppstart i . Antag att man vill använda sig av k mätningar per uppstart av exekveringsmiljön i analysen.
2. För varje uppstart av exekveringsmiljön, i , bestäm var tröskeln mellan insvägningsförloppet och jämviktsläget är. Kalla denna tröskel r_i , dvs insvägningsförloppet är slut efter den r_i :te mätningen, vilket betyder att det finns $s_i - r_i = k$ mätningar från jämviktsläget.
3. Bestäm medelvärdet, $\bar{x}_i$, för varje uppstart av exekveringsmiljön

$$\bar{x}_i = \frac{1}{k} \sum_{j=s_i-k+1}^{s_i} x_{ij}$$

4. Beräkna medelvärdet

$$\bar{\bar{x}} = \frac{1}{p} \sum_{i=1}^p \bar{x}_i$$

och konfidensintervallet för medelvärdet enligt kapitel 6.2.3.

I steg 2 finns det statistiska metoder att ta hjälp av vid bestämningen av tröskelvärdet, vilka dock inte går igenom här. Det går också att manuellt titta på kurvorna och bestämma var gränsen går. Man måste dock vara medveten om att gränsen inte behöver gå vid exakt samma ställe för alla uppstarter. Om man väljer att göra lika många mätningar i alla uppstarter, dvs s_i är samma för alla i , så måste man göra tillräckligt många mätningar för att det ska fungera för alla uppstarter.

9.3.5 Indata

Hur lång tid ett program, eller en del av ett program, tar beror i stor utsträckning på vilken indata som ges till programmet. Om man t ex har ett program för att hitta kortaste bilvägen på en karta så är det rimligt att det går mycket snabbare att hitta mellan Lund och Malmö än mellan Lund och Skövde. Det är också rimligt att det tar olika lång tid att sortera en helt osorterad lista och en nästan sorterad lista.

När man utvärderar prestanda eller jämför prestanda av två olika program så är det därför viktigt att man gör det för olika indata. Jämför t ex med de olika program som ingår i SPECjvm2008 ovan. Där har man valt att inkludera flera olika program för att jämförelser inte bara ska spegla hur det förhåller sig i ett visst läge.

9.4 Profileringsverktyg

Det finns flera verktyg framtagna för att analysera exekvering av program. Dessa verktyg kallas "profilerare" eller profileringsverktyg (eng. profilers). För java-program analyserar de vad som händer i exekveringsmiljön och man kan följa hur mycket minne som används, hur många klass-instanser som finns, hur mycket processor-kraft som används av olika klasser, etc.

Det finns både verktyg som säljs separat och verktyg som är av typen öppen källkod. Ett exempel på ett verktyg är "Java VisualVM" som levereras med jdk från version 6. Det kan startas med unix-kommandot

```
unix> jvisualvm
```

om det är installerat. Ett exempel på användning kan ses i figur 9.1. Man kan t ex använda verktyget genom att först starta det och sedan starta den java-applikation som man vill analysera. När man startat en java-applikation syns den i listan till vänster i fönstret. I figuren är det bara en java-applikation igång, "VisualVM", dvs verktyget självt. Om man dubbelklickar på applikationen man vill analysera får en ny flik i högra delen av fönstret, dvs den flik som visas i figuren. Här kan man välja mellan t ex

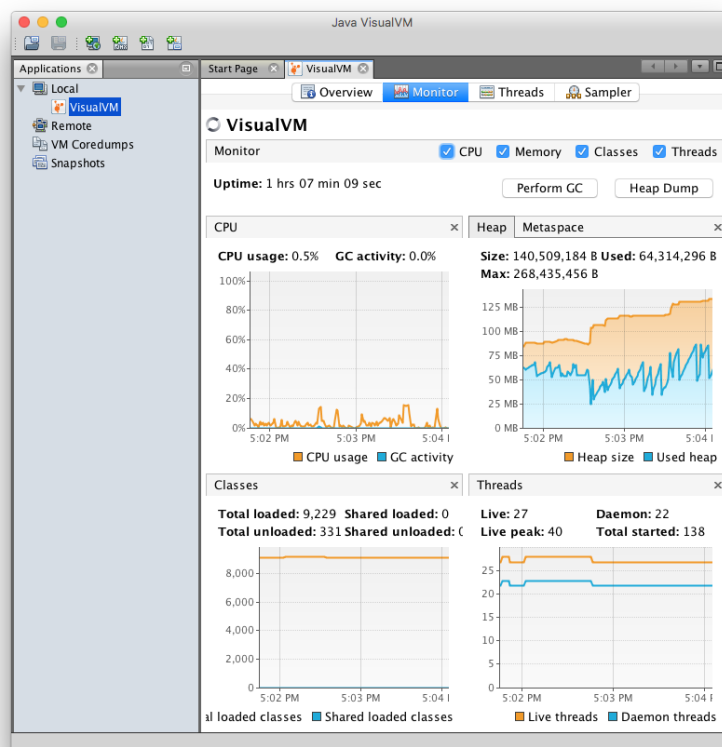
- "Overview": där man kan få en översikt
- "Monitor": där man kan följa CPU-åtgång, minnesåtgång, antalet klasser, och antalet trådar för hela det analyserade programmet
- "Threads": där man kan studera enskilda trådar i programmet
- "Sampler": där man kan studera minnesåtgång och CPU-åtgång per klass

Verktyget kan användas på ytterligare sätt, undersök gärna vad som finns i olika menyalternativ.

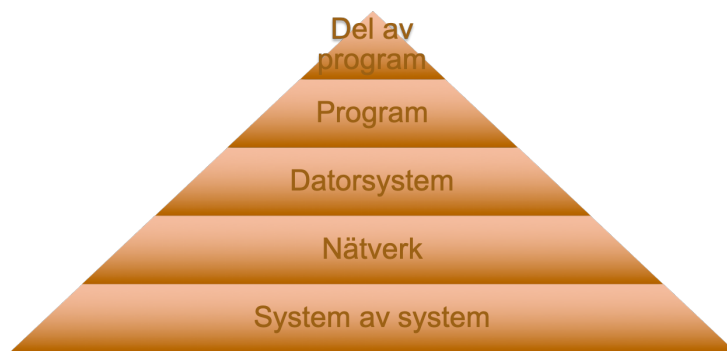
9.5 Prestandamätning i sitt sammanhang

Vi har här berättat hur man mäter prestanda av den del av ett program. Man kan förstås diskutera varför det är viktigt och det appliceras i ett lite större sammanhang. Den typ av delar som vi studera nu sätts samman till hela program. Detta visas i figur 9.2 där varje del i pyramiden är en del av flera av nivån under. Dvs, "program" är sammansatt av flera "del av program". Man kan såklart dela upp system och system av system i olika nivåer på olika sätt, men detta är ett sätt att se det.

Ofta är man intresserad av att mäta prestandan av ett helt system, t ex i termer av svarstid för en webbtjänst. Denna tid beror av flera tider i systemet, t ex kommunikation över nätverk, tid att hämta data i en databas, tid i själva



Figur 9.1: Ett exempel på ett profileringsverktyg, ”Java VisualVM”



Figur 9.2: System av system – ett sätt att se det

webbservern, etc. Hela svarstiden kan delas upp i dessa olika delar och var och en av de delprojekt som arbetar med att utveckla tjänsten kan få i uppdrag att ligga under ett visst värde för sin del. På så sätt är det viktigt att hålla reda på exekveringstiden för varje del, både t ex under utvecklingen och integrations- och systemtest.

Del II

R

Kapitel 10

Språket R

10.1 Vad är R?

R är ett programspråk för att analysera data och för statistisk analys. Det publiceras som öppen källkod, vilket betyder att det är gratis att ladda ned och att det även är möjligt att studera hur det är implementerat, samt att bidra till dess utveckling. Språket är förhållandevis väl använt i forskning och utveckling och det finns tillgängligt för alla vanliga plattformar (Windows, Linux, Mac)¹. Det används på ett sätt som liknar t ex Matlab, men det är lite mer fokuserat på statistisk analys, vilket passar bra i denna kursen.

Historiskt så har språket sitt ursprung i språket S som började utvecklas av "Bell laboratories" redan på 70-talet. R började utvecklas på 90-talet och det finns idag paket med funktionalitet för en stor mängd funktioner och det utvecklas hela tiden nya paket.

En fördel med att språket är så väl använt är att det finns mycket hjälp på nätet, både på R-projektets egna sidor och på andra sidor, som t ex "stack overflow". Ofta kan man söka på det man undrar om och hitta förhållandevis bra svar. Man ska dock vara medveten om att vissa sidor behandlar statistikproblem som kan vara svåra att förstå om man inte kan teorin som behandlas.

R är ett interpreterande språk som kan köras direkt i terminalen. Det sker alltså ingen kompilering som t ex i Java (eller Scala). Detta liknar en del andra språk som t ex Matlab och Python.

R används typiskt till:

- Importering av data: Det finns många färdiga funktioner för att importera data från t ex filer och databaser.
- Sortering och urval: När man väl importerat data till R så är det förhållandevis lätt att sortera och filtrera ut den information som man är intresserad av med olika kriterier.
- Statistisk analys: Det finns många färdigdefinierade funktioner för att göra statistisk analys, som t ex statistiska hypotestest och andra analyser. Det finns också många funktioner för att enkelt tillverka grafiska presentationer som t ex olika sorters diagram.

¹se r-project.org

I denna kursen kommer vi att använda R för att analysera mätdata från de laborationer vi gör och i projektet. I detta kompendiet går vi igenom några grundläggande funktioner och hur man använder R på ett enkelt sätt. Det finns en hel del mer avancerade funktioner i R som vi inte går igenom här. I stället hänvisar vi till någon av de utmärkta böcker och kompendium som finns att tillgå, t ex [32]. R har också en bra hjälp-funktion inbyggd som man kommer åt med funktionen `help()`.

10.2 Grundläggande om språket

Språket är, som nämns ovan, interpreterande, så man kan köra kommandon direkt från terminalen, som t ex

```
> 5+5
[1] 10
```

Här räknar man ut att $5 + 5 = 10$. '[1]' betyder att interpretatorn svarar med en lista med tal (i detta fall endast ett tal) och att det första talet på raden är nummer 1. Man behöver inte ha semikolon, men semikolon kan användas mellan kommandon om man vill ha mer än ett kommando på en rad.

Språket är dynamiskt typat, så interpretatorn försöker själv räkna ut vilken typ användaren menar. T ex:

```
> a <- 1
> b <- "hej"
> a
[1] 1
> b
[1] "hej"
```

Här förstår interpretatorn att `a` är ett heltal och `b` är en textsträng. Tilldelning görs vanligen med symbolen '`<-`', men även '`->`' fungerar om man skriver i andra ordningen, t ex '`10 -> c`'. Tecknet '`=`' kan också användas i stället för '`<-`', men det används inte så ofta.

10.2.1 Vektorer

Man kan lagra flera värden i en vektor, t ex

```
> a <- 1:10
> a
[1] 1 2 3 4 5 6 7 8 9 10
> b <- c(a, 2, 3)
> b
[1] 1 2 3 4 5 6 7 8 9 10 2 3
```

Här skapas först en vektor med värdena 1–10. Sedan används funktionen `c()` ("combine") för att kombinera ihop vektorn med talen 2 och 3 till en ny vektor `b` med 12 element. Vektorer kan också skapas genom att direkt kombinera ihop element med `c()`, t ex `c(1, 2, 12)`, eller genom att använda funktionen `vector()`.

Man kan ta reda på längden av en vektor med funktionen `length()`:

```
> length(5:10)
[1] 6
```

Ett enkelt sätt att komma åt innehållet i en vektor är att ange positionen inom parenteserna `[ ]`. T ex

```
> a <- 1:4
> a[2] <- 10
> a
[1] 1 10 3 4
> a[3]
[1] 3
```

Det första värdet finns alltså vid position 1, det andra vid position 2, osv.

Vektorerna kan användas i beräkningar och funktioner, t ex

```
> 5*c(2, 4)
[1] 10 20
> nchar(c("hej", "R"))
[1] 3 1
```

där `nchar()` är en funktion som returnerar antalet tecken i en sträng.

Man kan namnge elementen i en vektor och man kan ta reda på namnen med funktionen `names()`, t ex:

```
> daylength <- c(4222.6, 2802.0, 24, 708.7)
> names(daylength) <- c("Mercury", "Venus", "Earth", "Moon")
> daylength
Mercury Venus Earth Moon
4222.6 2802.0 24.0 708.7
> names(daylength)
[1] "Mercury" "Venus" "Earth" "Moon"
```

Om man anropar mer avancerade funktioner som resulterar i vektorer (t ex `table()` och `tapply()` som går igenom senare i kompendiet) så får man ofta vektorer med namn på elementen.

10.2.2 Grundläggande typer

Det finns ett antal grundläggande typer ("atomic modes") av dataelement, vilka vi kallar typer i detta kompendium:

- **numeric** – används för att representera numerisk data, både heltal (som kallas **integer**) och flyttal (som kallas **double**)
- **character** – används för att representera text i strängar
- **logical** – används för att representera boolsk data med värdena **TRUE** och **FALSE** (T och F kort).
- **complex** – används för att representera komplexa tal

En vektor kan bara innehålla en sorts grundläggande typer

```
> a <- c(1, 2, 3)
> b <- c("Lund", "Malmo")
> a
[1] 1 2 3
> b
[1] "Lund" "Malmo"
> c(a, b)
[1] "1" "2" "3" "Lund" "Malmo"
```

Här är `a` en numerisk vektor och `b` en vektor med textelement. När vi kombinerar ihop dem omvandlar R till en vektor med textelement eftersom de måste vara av samma typ. Detta kallas för implicit omvandling.

Man kan också göra explicita omvandlingar, t ex

```
> as.numeric(c("11", "22"))
[1] 11 22
```

Man kan använda t ex funktionen `mode()` för att ta reda på vilken typ man har

```
> mode(c("text1", "text2"))
[1] "character"
```

För denna typ av omvandling finns det även t ex funktionen `as.character()` och `as.logical()`.

10.3 Objekt för att representera data

I R säger man att det finns olika sorters objekt för att representera data. Några viktiga objekt är:

- Vektorer – för att lagra en serie data. All data måste vara av samma typ. Se kapitel 10.2.1.
- Matriser – matriser för att lagra data i två dimensioner. All data måste vara av samma typ.
- 'Array' – för att lagra data i mer än två dimensioner. All data måste vara av samma typ.
- Listor – listor för att lagra en lista med data där dataelementen kan vara av olika typ och bestå av olika sorters objekt för att representera data
- Dataramar – för att lagra data i en tvådimensionell struktur i rader och kolumner. Kolumnerna kan vara av olika typ.
- Faktorer – för att lagra faktordata, dvs data på nominalskala som t ex används för att beskriva olika nivåer av en oberoende variabel i ett experiment.

Funktionen `str()` är bra att använda om man vill ta reda på vilken typ av objekt man har. Nedan går vi igenom några av objekten lite närmare.

10.3.1 Matriser

Matriser kan skapas med

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
        dimnames = NULL)
```

Funktionen har alltså fem argument (`data`, `nrow`, `ncol`, `byrow`, `dimnames`). Det som står efter likhetstecknen är "defaultvärden" som kommer att användas om man inte anger något värde, se kap 11.2. Funktionen kan t ex användas såhär:

```
> m <- matrix(c("A", "B", "C", "D"), 2, 2)
> m
      [,1] [,2]
[1,] "A"  "C"
[2,] "B"  "D"
```

Här skapar vi alltså en matris med 2 rader och två kolumner, som vi fyller med data ("A" – "D"). Notera att R fyller matrisen kolumnsvis med den data vi angett (eftersom `byrow` är falsk). Man kan komma åt data genom att ange positionen:

```
> m[2,2]
[1] "D"
> m[,2]
[1] "C" "D"
```

Man anger alltså först raden och sedan kolumnen. Om man inte anger något värde alls så tolkar R det som hela raden/kolumnen.

10.3.2 Listor

Listor kan skapas med `list(...)`, där `'...'` betyder är en lista med objekt:

```
> L <- list("A", matrix(1:4, 2, 2))
> str(L)
List of 2
 $ : chr "A"
 $ : int [1:2, 1:2] 1 2 3 4
> str(L[1])
List of 1
 $ : chr "A"
> str(L[[1]])
chr "A"
```

Här skapar vi först en lista med två element, en textsträng och en matris. Sedan undersöker vi listan med `str()` och ser att den innehåller två element. Efter det tar vi ut det första elementet med `[`, vilken ger en lista (med ett element) med elementet. För att få det element som verkligen finns på första platsen så tar man ut det med `[[`. När man indexerar listor så använder man oftast `[[` och inte `[`.

Man kan namnge elementen i en lista och använda namnen med `$` för att komma åt elementen:

```
> L2 <- list(Cities = c("Lund", "Malmo"), Populations = c(
  (87000, 280000))
> L2$Cities
[1] "Lund" "Malmo"
```

Med denna listan ger alltså `L2$Cities` samma resultat som `L2[[1]]`. Det är ofta en fördel att använda namnen, eftersom det är lättare att hålla reda på namnen än på vilken position viss data befinner sig. Man kan, ungefär som för vektorer, ta reda på namnen med funktionen `names()` och man kan också sätta nya namn:

```
> names(L2)
[1] "Cities"          "Populations"
> names(L2) <- c("stader", "invanare")
> L2
```

```
$stader
[1] "Lund" "Malmo"

$invanare
[1] 87000 280000
```

10.3.3 Dataramar

Denna typ av objekt är mycket kraftfull för att lagra och behandla data. Man kan skapa en ny dataram med funktionen `data.frame()`, t ex

```
> name <- c("Paul", "Bodil", "Karla", "Diana")
> age <- c(25, 30, 22, 22)
> persons <- data.frame(name, age)
> persons
  name age
1 Paul  25
2 Bodil 30
3 Karla 22
4 Diana 22
```

Nu kan man använda data för att t ex räkna ut medelvärde med `mean(persons$age)`. `$` fungerar alltså på ungefär samma sätt som för listor och ger här den kolumn som man angett namnet på. En mycket användbar funktion för en dataram är `summary()`, som ger en sammanfattning av innehållet:

```
> summary(persons)
      name      age
Bodil:1  Min.    :22.00
Diana:1  1st Qu.:22.00
Karla:1  Median  :23.50
Paul :1  Mean    :24.75
        3rd Qu.:26.25
        Max.    :30.00
```

Resultatet anpassas alltså efter innehållet. För nominell data får man reda på hur många element det finns av varje sort och för numerisk data (ratio) får man t ex reda på medelvärde och medianvärde. Ungefär som för listor så kan man komma åt namnen på kolumnerna med `names()` (eller `colnames()`).

Man kan indexera med `[ ]`, ungefär som för en matris, för att komma åt en del av en dataram:

```
> persons[3,] # third row
  name age
3 Karla  22
> persons[,1] # first column
[1] Paul  Bodil Karla Diana
Levels: Bodil Diana Karla Paul
> persons[-c(2, 3),] # all rows except the second and third
  name age
1 Paul  25
4 Diana 22
> persons[, "age"] # column name as index
[1] 25 30 22 22
> persons[1] # first column
```

```

      name
1 Paul
2 Bodil
3 Karla
4 Diana
> persons$name # first column
[1] Paul Bodil Karla Diana
Levels: Bodil Diana Karla Paul

```

'#' är kommentarstecknet i R, så det som står efter det på raden är bara en kommentar. '-' betyder ungefär "alla utom". Man kan se i koden ovan att första kolumnen innehåller namnen som en faktor (eftersom R skriver ut nivåerna). Som man kan se i exemplen ovan så kan man göra ungefär samma sak på olika sätt. Om man t ex vill komma åt första kolumnen så kan man göra det med `persons[, 1]` (alla rader i första kolumnen), `persons[1]` (första kolumnen), `persons$name` (kolumnen 'name'), `persons["name"]` (kolumnen 'name') och `persons[, "name"]` (alla rader i kolumnen 'name').

R gör om textdata till faktorer om man inte anger med en parameter att det inte ska göras. Om man vill ha namnen som textsträngar kan man ge ett argument `stringsAsFactors = FALSE`, se hjälpfunktionen. Man kan också omvandla kolumnen till textelement t ex med

```
> persons$name <- as.character(persons$name)
```

Några ytterligare funktioner som är användbara för dataramar presenteras i kapitel 10.5.

Som vi såg i kapitel 10.2.2 så kan man omvandla till de grundläggande typerna med "as.-funktioner". Man kan också omvandla t ex matriser och listor till dataramar med `as.data.frame`-funktionen:

```

> as.data.frame(list(col1 = c(1, 3), col2 = c("x1", "x2")))
  col1 col2
1     1  x1
2     3  x2

```

10.4 Urval av data

Det finns ett antal sätt att indexera och välja ut data ur olika sorters objekt. Vi har sett tidigare hur man kommer åt värden ur en vektor

```

> a <- 2*(1:8)
> a[4]
[1] 8
> a[c(2, 3)]
[1] 4 6
> a[-5]
[1] 2 4 6 8 12 14 16

```

Man kan alltså indexera med det eller de värden man vill komma åt, eller med ett minustecken när man vill komma åt alla värden utom de man anger.

Man kan också ange en logisk vektor som anger om man vill ta med värdet eller inte

```

> a[c(T, T, T, T, F, T, T, F)]
[1] 2 4 6 8 12 14

```

Om man anger färre logiska värden än vad det finns värden i objektet man indexerar så kommer den logiska vektorn man anger att repeteras så att man fyller ut till tillräckligt många värden. För att få vartannat värde kan man alltså skriva

```
> a[c(T, F)]
[1]  2  6 10 14
```

Man kan utnyttja detta sätt att indexera om man vill skriva uttryck för vilka datapunkter man vill ha med. Man kan ju beräkna uttryck som

```
> a<10
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

Alltså kan man t ex få alla värden som är mindre än 10 med `a[a<10]`. Ett exempel på detta sätt att indexera är om man vill lista namnen på alla personer som är yngre än 25 år i `persons` enligt ovan:

```
> persons[persons$age<25, "name"]
[1] "Karla" "Diana"
```

När det gäller urval vill man ibland ta bort datapunkter där man inte har komplett data. I R finns det två vanligen förekommande beteckningar på saknad eller felaktig data, `NA` som är ett typiskt värde efter att ha matat in data från en fil eller liknande och det saknas data och `NaN` (not a number) som t ex är resultatet av `0/0`. Man kan kontrollera om en datapunkt är saknad med funktionen `is.na`, t ex

```
> naData <- suppressWarnings(c(10, as.numeric("X"), 0/0))
> naData
[1] 10 NA NaN
> is.na(naData)
[1] FALSE  TRUE  TRUE
```

där `suppressWarnings` används för att inte få varningar från `as.numeric` som vi helt medvetet använder med felaktig data i exemplet. Om man har data i t ex en dataram så kan man använda `complete.cases()` för att få reda på vilka observationer (dvs rader) som är kompletta, dvs inte innehåller `NA` eller `NaN`.

10.5 Operationer och vanliga funktioner

Nu när vi sett hur man får in data i R och hur man utför enkla funktioner är vi redo att sammanfatta de operationer som finns och vanliga funktioner. Operatorerna i R sammanfattas i tabell 10.1 (som kan fås genom att skriva `?Syntax` i R). Det finns många färdiga beräkningsfunktioner i R. Några av dem beskrivs i tabell 10.2. För att få mer information om en funktion använder man hjälpfunktionen, t ex `help(sum)` (eller `?sum`). För att få reda på vilka grundläggande funktioner som finns kan man skriva `library(help = "base")` och för att få reda på alla funktioner i det grundläggande statistikpaketet kan man skriva `library(help = "stats")`.

I tabell 10.2 ser vi att `'...'` och `'na.rm = FALSE'` förekommer ofta. Punkterna `'...'` betyder ungefär att funktionen kan ta emot ett variabelt antal argument. Argumentet `'na.rm = FALSE'` betyder att det finns ett 'defaultvärde' som säger att R inte ska ta bort datapunkter som saknas i beräkningarna. Ofta vill man

Tabell 10.1: Operatorer i prioritetsordning (* = behandlas inte i detta kompendium och inte uppenbar betydelse)

:: :::	åtkomst av variabler i 'namespace' *
\$ @	komponent, @*
[[[	indexering
^	exponent
- +	unär plus och minus
:	sekvens
%any%	special-operator *
* /	multiplikation, division
+ -	binär addition och subtraktion
< > <= >= == !=	jämförelser
!	negation
& &&	och
	eller
~	för att beskriva modeller *
-> ->>	tilldelning, tilldelning i 'enclosing environment'*
<- <<-	tilldelning, tilldelning i 'enclosing environment'*
=	tilldelning (motsvarar '<-')
?	hjälp

Tabell 10.2: Några användbara funktioner för beräkningar

sum(..., na.rm = FALSE)	summa
prod(..., na.rm = FALSE)	produkt
cumsum(x), cumprod(x)	kumulativ summa, produkt
mean(x, trim = 0, na.rm = FALSE, ...)	medelvärde
median(x, na.rm = FALSE)	median
var(x, na.rm = FALSE)	varians
sd(x, na.rm = FALSE)	standardavvikelse
exp(x)	e^x
log(x, base = exp(1)), log10(x), log2(x)	logaritmer
max(..., na.rm = FALSE)	max
min(..., na.rm = FALSE)	minimum
range(..., na.rm = FALSE)	'range', dvs min och max
which.max(x), which.min(x)	vilket värde som är max, min
sqrt(x), abs(x)	$\sqrt{x}$, $ x $
sin(x), cos(x), tan(x)	$\sin()$, $\cos()$, $\tan()$ (radianer)
round(x, digits = 0)	avrundning (digits decimaler)
floor(x), ceiling(x)	avrundning nedåt, uppåt
factorial(x), choose(n, k)	$x!$, $n!/((n-k)!k!)$

Tabell 10.3: Några användbara funktioner för att arbeta med dataobjekt

<code>nrow(a)</code>	antal rader i <code>a</code>
<code>ncol(a)</code>	antal kolumner i <code>a</code>
<code>rbind(a, b)</code>	sammanslagning där raderna i <code>b</code> kommer efter raderna i <code>a</code> , dvs en radvis sammanslagning.
<code>cbind(a, b)</code>	sammanslagning där kolumnerna i <code>b</code> kommer efter kolumnerna i <code>a</code> , dvs en kolumnvis sammanslagning.
<code>summary(a)</code>	sammanfattar innehållet i <code>a</code> , t ex medelvärde och median
<code>str(a)</code>	sammanfattar <code>a</code> 's struktur
<code>head(a)</code>	de första raderna i <code>a</code>
<code>tail(a)</code>	de sista raderna i <code>a</code>
<code>sort(a)</code>	sortera <code>a</code>
<code>order(a)</code>	index som sorterar vektor <code>a</code> , dvs <code>a[order(a)]</code> motsvarar <code>sort(a)</code>
<code>table(a)</code>	räknar hur många element i <code>a</code> det finns av varje värde
<code>unlist(x)</code>	omvandlar listan <code>x</code> till en vektor
<code>append(x, values, after = length(x))</code>	lägger in vektorn <code>values</code> i vektorn <code>x</code> efter position <code>after</code> (om <code>after==0</code> så hamnar <code>values</code> först)

att R ska ta bort saknade värden och man sätter då detta argument till `TRUE` i anropet. Mer information om argument till funktioner finns i kapitel 11.2.

I tabell 10.3 presenteras några funktioner som är användbara för att arbeta med dataobjekt som dataramar och matriser. I tabellen presenteras inte alla argument till funktionerna. För en komplett beskrivning hänvisas t ex till hjälpfunktionerna.

Flera av funktionerna har vi redan gått igenom, men några är nya. För att lägga till rader eller kolumner i t ex dataramar kan man använda `rbind()` och `cbind()`, t ex (med `persons` enligt sidan 88)

```
> rbind(persons, data.frame(name="Ivar", age=51))
  name age
1 Paul  25
2 Bodil 30
3 Karla 22
4 Diana 22
5 Ivar  51
```

Sortering kan göras med `sort()`-funktionen, t ex enligt

```
> a <- c(2, 10, 3, 3, 6)
> sort(a)
[1] 2 3 3 6 10
> sort(a, decreasing=T)
[1] 10 6 3 3 2
```

Flera funktioner fungerar med olika typer, t ex kan `max`, `min` och `sort` användas även med t ex strängar:

```
> sort(c("b", "a", "c"))
[1] "a" "b" "c"
> min(c("b", "a", "c"))
```

```
[1] "a"
```

Om man vill få reda på index-ordningen som sorteringen ska ske i så kan man använda `order()`

```
> order(a)
[1] 1 3 4 5 2
```

dvs först kommer index 1, sedan index 3, osv.

För att räkna antalet element av varje värde kan `table()` användas

```
> table(a)
a
 2  3  6 10
 1  2  1  1
```

som också kan användas för ta fram en ”kontingenstabell” (Contingency Table). För att se ett exempel på detta så kan vi titta på exempeldata mängden `mtcars` som innehåller 11 aspekter av 32 bilmodeller med årsmodell 1973-74. Det är alltså en dataram med 11 kolumner och 32 rader. Om vi är intresserade av antal växlar och antal cylindrar så finns de i två av kolumnerna:

```
> str(mtcars[c("cyl", "gear")])
'data.frame': 32 obs. of 2 variables:
 $ cyl : num 6 6 4 6 8 6 8 4 4 6 ...
 $ gear: num 4 4 4 3 3 3 3 4 4 4 ...
```

En kontingenstabell kan fås med

```
> table(mtcars[c("cyl", "gear")])
      gear
cyl   3   4   5
 4     1   8   2
 6     2   4   1
 8    12   0   2
```

Då kan man t ex i den nedersta raden se att det i datamängden finns 12 åttacylindriga bilmodeller som var treväxlade, ingen som var fyrväxlade och 2 som var femväxlade.

Kapitel 11

Programmering i R

11.1 Skript

Vi har tidigare gått igenom de grundläggande funktionerna i R. Nu ska vi se hur man gör lite mer avancerade saker som att skriva funktioner och hela program. Hittills har vi gjort alla beräkningar i terminalen men när vi ska göra lite mer avancerade saker kan det vara bra att göra beräkningarna i skript-filer. En skript-fil är helt enkelt en textfil med instruktioner som körs i R som om de hade matats in från terminalen.

Om man `tex` har en fil `myscript.R` med följande innehåll

```
a <- "Hej"
print(a)
```

så kan man köra skriptet med kommandot `source()`:

```
> source("myscript.R")
[1] "Hej"
```

När man gör detta är det viktigt att man i R är i samma arbetsbibliotek som skriptet ligger i. Man kan ta reda på vilket arbetsbibliotek man är i med funktionen `getwd()` och man kan bestämma vilket arbetsbibliotek man ska vara i med `setwd()`. I detta kompendium markerar vi att den R-kod vi visar finns i skript och inte på R:s normala kommandorad genom en ram runt koden som ovan. Notera dock att all kod som ges i skript kan ges på kommandoraden i stället.

11.2 Funtioner

Ofta vill man definiera funktioner som returnerar ett värde. I R skriver man en funktion enligt

```
function(arglist) body
```

där `function` är ett reserverat ord, `arglist` en lista med kommaseparerade argument där varje argument antingen är en enkel symbol, ett uttryck som `'symbol = default_value'` för att ange 'defaultvärden', eller `'...'`. De tre punkterna `'...'` betyder att det kan finnas en variabel mängd argument, vilket används `tex` när

funktionen anropar en annan funktion. I detta kompendium går vi inte igenom hur man definierar denna typ av argument.

`body` är vilket giltigt R-uttryck som helst, där flera uttryck kan slås samman inom `{ }`. Funktioner kan definieras som anonyma (om de t ex används i ett `apply()`-uttryck eller annan ”högre ordnings funktion”), men ofta låter man en variabel tilldelas funktionen.

Ett enkelt exempel på en funktion är

```
> f <- function(x) x^2
> f(1:10)
[1] 1 4 9 16 25 36 49 64 81 100
```

där vi först definierar en funktion som vi kallar `f` med argumentet `x` och sedan anropar den med värdena 1 – 10. Ett aningen mer avancerat exempel på en funktion är

```
# Convert EUR to SEK
convertToSEK <- function(EUR, rate = 9.2) {
  inSEK <- rate * EUR
  inSEK
}
```

Denna funktion har två argument, `EUR` och `rate`. Argumentet `rate` har ett ’defaultvärde’, dvs ett värde som används om man inte anger något värde när man anropar funktionen. Funktionens resultat anges på sista raden. Man hade även kunnat ange resultatet med `return(inSEK)`, som fungerar ungefär som `return` i andra språk. Funktion kan t ex anropas enligt följande:

```
> convertToSEK(10)
[1] 92
> convertToSEK(10, rate = 10)
[1] 100
> convertToSEK(10*1:10, 9.2)
[1] 92 184 276 368 460 552 644 736 828 920
```

11.3 Upprepning och villkor

Här presenteras några vanliga programmeringskonstruktioner genom exempel som förhoppningsvis är självförklarande.

Upprepning/iteration:

```
cities <- c("Malmo", "Lund", "Trelleborg", "Ystad")
for (city in cities) {
  to_print <- paste(city, "is a city")
  print(to_print)
}
```

’while’:

```
while(sum > 0) {
  sum <- sum -1
  print(sum)
}
```

Villkor:

```
if (city != "Malmo") {
  numberOther <- numberOther + 1
  print("Not Malmo")
} else
  print("This is Malmo")

if (a>b)
  b <- c
```

11.4 Ett exempel på en funktion

Ett lite mer avancerat exempel på en funktion är insättningssortering. Om man t ex vill skriva en funktion för att sortera vektorer med insättningssortering kan man göra det såhär:

```
insertionSort <- function(x) {
  if (length(x)==1) return(x)
  result <- x[1]
  for (i in 2:length(x)) {
    j <- length(result)
    while (j>0 && x[i]<result[j])
      j <- j-1
    result <- append(result, x[i], after=j)
  }
  result
}
```

där funktionen `append` kanske inte är helt självklar, men den lägger in `x[i]` i `result` efter position `j`, se tabell 10.3.

Vi kan prova vår funktion t ex enligt följande:

```
> unsorted <- order(runif(10))
> unsorted
[1] 6 4 1 9 8 10 5 2 3 7
> insertionSort(unsorted)
[1] 1 2 3 4 5 6 7 8 9 10
```

Här tillverkar vi först en vektor med osorterade heltal genom att använda funktionen `order` på en vektor med 10 slumpstal mellan 0 och 1 (`runif`). Funktionen `insertionSort` är avsevärt långsammare än R:s inbyggda `sort`-funktion. För att undersöka hur mycket långsammare vår funktion är kan vi använda funktionen `microbenchmark`. Denna funktion finns i paketet `microbenchmark` som inte finns bland standardfunktionerna i R. Innan man använder det så måste man först installera det, vilket görs med funktionen

```
> install.packages("microbenchmark")
```

Sedan måste man, för varje gång man startar R specificera att den ska leta efter även detta paket, vilket man gör t ex med funktionen `library`. Funktionen `microbenchmark` kör de uttryck man anger 100 gånger och mäter exekveringstiden för varje körning i mikrosekunder. För att jämföra kan vi alltså köra följande:

```

> library(microbenchmark)
> unsorted <- order(runif(60))
> res <- microbenchmark(insertionSort(unsorted), sort(
  unsorted))
> summary(res)[c("expr", "mean")]
              expr      mean
1 insertionSort(unsorted) 865.22849
2      sort(unsorted)    28.68284

```

`res` är en dataram med två variabler, `expr` och `time`. Eftersom `expr` är en faktor så ger `summary(res)` medelvärde, max, min etc. av `time` för varje värde som `expr` kan anta. Här väljer vi att bara skriva ut kolumnerna `expr` och `mean` av `summary(res)` för att spara plats.

Man kan se att det är stor skillnad i tidsåtgång för vår sorteringsfunktion och R:s vanliga sorteringsfunktion. Skillnaden blir ännu tydligare om man sorterar större vektorer.

11.5 Importera data från fil

Det finns flera funktioner för att läsa in data från externa källor som t ex från textfiler. En användbar funktion för att läsa in data från en kommaseparerad fil till en dataram är

```

read.csv(file, header = TRUE, sep = ",", quote = "\"",
  dec = ".", fill = TRUE, comment.char = "", ...)

```

Med `anger` man helt enkelt sökvägen för filen (`file`) och, om man vill, om det finns en rubrikrad, vilket tecken som separerar element i filen, etc. Man kan ange många fler argument, se t ex hjälpfunktionen för funktionen. Det finns flera ytterligare funktioner för att importera data och man kan också använda t ex funktionen `read.table()` som är mer generell än `read.csv()`.

11.6 Rita grafer

Det finns många funktioner för att rita grafer i R. I följande skript visar vi ett exempel på hur man först ritar en graf med en kurva (`sales_A`), sedan lägger till ytterligare en kurva (`sales_B`) och sedan en förklarande ruta (eng. legend).

```

time <- 2000:2005
sales_A <- c(10, 8, 7, 7, 5, 6)
sales_B <- c(4, 8.5, 7.5, 8, 9, 11)
y_range <- range(1, sales_A, sales_B)

# Plot sales_A in a diagram
plot(time, sales_A,          # X and Y data
  type = "o",               # lines between points
  lty = 1,                  # line type solid
  pch = 21,                 # points as circles
  main = "Sales",           # heading
  xlab = "Year",            # X axis label
  ylab = "Sales (MSEK)",    # Y axis label
  ylim = y_range,          # range covers both curves
)

```

```
# Add plot of sales_B
lines(time, sales_B,          # X and Y data
      type = "o",            # lines between points
      lty = 2,                # dashed line
      pch = 22,               # square points
      col = "black"           # black line
)

# Add legend to plot
legend(2003.5, 3,             # legend placement
      c("Prod A", "Prod B"), # legend texts
      lty = c(1, 2),         # line types
      pch = c(21, 22)        # point types
)
```

För att rita grafen används funktionen `plot()`. Argumentet `type` anger vilken typ av kurva det ska vara där "o" betyder en kurva med punkter. Andra exempel på värden på `type` är "p" ("points"), "l" ("lines") och "h" ("histogram like"). Argumenten `lty` och `pch` anger vilken typ av linjer och punkter det ska vara och argumenten `main`, `xlab` och `ylab` anger grafens rubrik och rubrikerna på axlarna. `ylim` specificerar hur stort område y-axeln ska täcka. Funktionen `lines()` används för att lägga till kurvor till den senast ritade grafen och har ungefär samma argument som `plot()`. Resultatet kan ses i figur 11.1.

Man kan ange många ytterligare argument till funktionerna, t ex för att ändra färgen och typsnittet på texten, tjockleken på linjerna, hur strecken ska kopplas ihop med punkterna, osv. För mer information, se t ex hjälpfunktionerna.

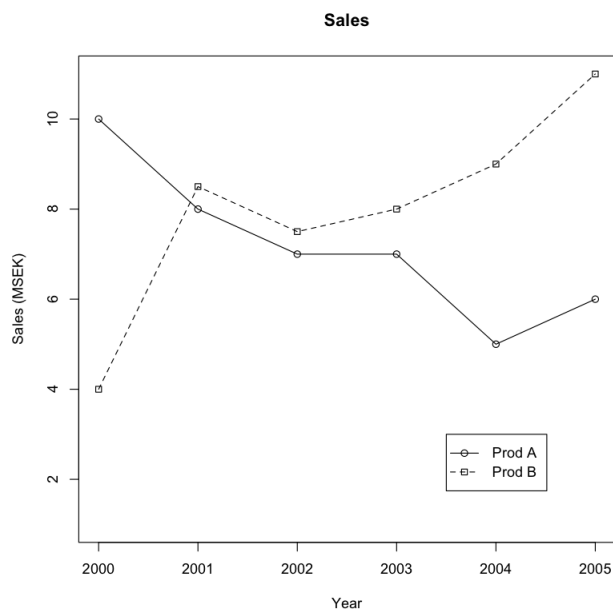
Några ytterligare funktioner för att rita grafer är t ex

- `boxplot(x)` – Ritar en boxplot. Om `x` är en vektor så får man en figur med en enkel boxplot. Om `x` är en dataram så får man en figur med en boxplot för varje kolumn i `x`.
- `barplot(x, beside=FALSE)` – Ritar ett stapeldiagram. Om `x` är en vektor så får man en figur med "ett vanligt" stapeldiagram. Om `x` är en dataram så får man en figur där staplarna är grupperade enligt kolumnerna, där argumentet `beside` anger om staplarna i grupperna ska lagras på varandra eller vara sidan om varandra.
- `hist(v)` – Ritar ett histogram för datan i vektorn `v`
- `pie(v)` – Ritar ett tårtdiagram med storleken på tårtbitarna enligt vektorn `v`.

Man kan ange många ytterligare argument till funktionerna. För mer information, se t ex hjälpfunktionerna.

11.7 Några apply-funktioner

I detta delkapitel går vi igenom några högre ordningens funktioner, dvs funktioner som tar funktioner som parametrar, som `lapply()` och andra `apply`-funktioner. `apply`-funktionerna gör det möjligt att iterera över datastrukturer utan att använda t ex `for`-loopar.



Figur 11.1: En graf

11.7.1 lapply() och sapply()

Som vi antydde i kapitel 11.2 så är funktioner "first-class citizens" i R, vilket betyder att man kan göra allting med funktioner som man kan göra med andra variabler. Det betyder t ex att man kan tilldela dem till variabler och man kan skicka med dem som argument till funktioner, där det senare används t ex i funktionerna `lapply()` och `sapply()`:

```
> x <- list(elem1 = c(1, 2, 4), elem2 = 50:100)
> lapply(x, mean)
$elem1
[1] 2.333333

$elem2
[1] 75

> sapply(x, mean)
      elem1      elem2 
2.333333 75.000000
```

Här skapas först en lista med vektorer (`x`). Sedan appliceras funktionen `mean()` på varje element i listan med hjälp av funktionen `lapply()`.

Resultatet av `lapply()` är en lista. Ibland vill man ha ett mer lättöverskådligt svar och då kan man använda funktionen `sapply()` i stället, som fungerar på samma sätt som `lapply()`, men som ger ett mer användarvänligt svar.

Man kan givetvis skicka med funktioner som man själv definierat och man kan skicka med anonyma funktioner:

```
> meanPlusOne <- function(x) mean(x) + 1
> sapply(x, meanPlusOne)
      elem1      elem2
3.333333 76.000000
> sapply(x, function(x) mean(x) + 2) # anonymous function
      elem1      elem2
4.333333 77.000000
```

Om man anropar `lapply` med en dataram så kommer den funktion man anger att köras för varje kolumn i dataramen, och resultatet kommer att presenteras som en lista. Om vi först definierar en dataram som

```
df <- data.frame(a=c(1, 3, 9), b=c(3, 5, 9))
> df
  a b
1 1 3
2 3 5
3 9 9
```

och sedan en funktion för att skala, där vi vill att alla värden i en vektor ska transformeras så att det minsta får värdet 0 och det största värdet 1 så kan vi göra det såhär:

```
> myscale <- function(x) (x-min(x))/(max(x)-min(x))
> myscale(c(1, 5, 2))
[1] 0.00 1.00 0.25
```

Om vi nu vill transformera varje kolumn i vår dataram så kan vi göra det med funktionen `lapply`. Resultatet blir dock en lista så om vi vill få en dataram som resultat så får vi omvandla listan t ex med funktionen `as.data.frame`:

```
> df <- as.data.frame(lapply(df, myscale))
> df
      a      b
1 0.00 0.000000
2 0.25 0.333333
3 1.00 1.000000
```

11.7.2 split(), tapply() och några fler funktioner

Funktionen `split()` kan användas för att dela upp t ex en dataram i en lista med nya dataramar enligt en faktor. I följande exempel delas `results` upp i två nya dataramar, en för varje namn i `name`, dvs "LH" och "MH". Sedan använder man `sapply()` för att räkna ut medelbetyget för varje namn.

```
> name <- c("MH", "MH", "LH", "MH", "LH")
> grade <- c(3, 4, 3, 4, 5)
> results <- data.frame(name, grade)
> splitted <- split(results, results$name)
> splitted
$LH
  name grade
3   LH     3
5   LH     5

$MH
```

```

      name grade
1     MH      3
2     MH      4
4     MH      4

> sapply(splitted, function(y) mean(y$grade))
      LH      MH
4.000000 3.666667

```

Det finns en annan funktion, `tapply()`, som gör ungefär samma sak som `split()` och `sapply()` tillsammans:

```

> tapply(results$grade, results$name, mean)
      LH      MH
4.000000 3.666667

```

Funktionen anropas enligt

```
tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

där `X` typiskt är en vektor, `INDEX` är en vektor, eller en lista med flera vektorer, med lika många faktorelement som det finns element i `X` och `FUN` är en funktion som beräknas för varje grupp av värden i `X`.

Det finns ett antal ytterligare "apply-funktioner" som är användbara, t ex

- `mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)` – Applicera en funktion `FUN` med mer än ett argument (...), dvs en multivariat version av `sapply`.
- `apply(X, MARGIN, FUN, ...)` – Applicera en funktion `FUN` över en array `X`. `MARGIN` är en vektor som anger över vilken/vilka dimensioner funktionen ska appliceras.
- `by(data, INDICES, FUN, ..., simplify = TRUE)` – Ungefär som `tapply()`, men i stället för att ange en vektor `X` anger man en dataram `data`
- `aggregate(x, ...)` – kan sägas vara en mer generell version av `tapply` som fungerar på fler sorters dataobjekt och ger resultat som är uppställt på ett sätt som ibland ger bättre överblick

För mer information, se t ex hjälpfunktionen.

11.7.3 Några ytterligare funktioner för funktionsprogrammering

Som vi sett ovan så kan R användas för att programmera på ett funktionsorienterat sätt. Det är därför på sin plats att nämna för den som är intresserad att även de vanliga metoderna *filter*, *map* och *reduce/fold* finns i R:

```

> a <- 1:10
> Filter(function(x) x>5, a)
[1] 6 7 8 9 10
> unlist(Map(function(x) x^2, a))
[1] 1 4 9 16 25 36 49 64 81 100
> Reduce(function(x, y) x+y, a)
[1] 55

```

För mer information se t ex hjälpfunktionerna eller [27].

11.8 Ytterligare funktioner

I detta kompendium har vi främst gått igenom de funktioner som har med programmering att göra och de beräkningsfunktioner som behövs i kursen. Det är viktigt att påpeka att det finns många ytterligare områden av funktioner som erbjuds av R, t ex

- Statistikfunktioner, som t ex statistisk test
- Slumptal: för i stort sett alla fördelningar finns det funktioner för slump-talsgenerering, täthetsfunktion, fördelningsfunktion, etc
- Funktioner för maskininlärning

Det finns även många paket av funktioner som utvecklats av användare av R. Många av dessa distribueras av CRAN¹.

¹<https://cran.r-project.org/web/packages/>

Litteraturförteckning

- [1] Anglia Ruskin University. Guide to Harvard style of referencing, 2017. <https://libweb.anglia.ac.uk/referencing/harvard.htm> (besökt 2017-10-11).
- [2] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.
- [3] Jarl Backman. *Rapporter och uppsatser*. Studentlitteratur, 2006.
- [4] Victor R. Basili och Dieter Rombach. The the TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, 1988.
- [5] Gunnar Blom, Jan Enger, Gunnar Englund, Jan Grandell och Lars Holst. *Sannolikhets teori och statistik teori med tillämpningar*. Studentlitteratur, 2005.
- [6] Antonio Cavacini. What is the best database for computer science journal articles? *Scientometrics*, 102:2059–2071, 2015.
- [7] Shyam R. Chidamber och Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
- [8] Christian W. Dawson. *The Essence of Computing Projects, a Student's Guide*. Prentice Hall, 2000.
- [9] Tom DeMarco. *Controlling Software Projects*. Yourdon Press, New York, 1982.
- [10] Norman E. Fenton och Shari Lawrence Pfleeger. *Software Metrics, a Rigorous & Practical Approach*. PWS Publishing Company, 1997.
- [11] Per Foreby. Att skriva rapporter med LATEX. Datordriftgruppen LTH, <http://www.ddg.lth.se/perf/handledning/handledning.pdf>, 2006.
- [12] Andy Georges, Dries Buytaert och Lieven Eeckhout. Statistically rigorous java performance evaluation. I: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA'07)*, sid. 57–76, 2007.
- [13] Robert L. Glass. The software research crisis. *IEEE Software*, 11:42–47, 1994.

- [14] Per Holm. *Objektorienterad programmering och Java*. Studentlitteratur, 2007.
- [15] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.
- [16] Martin Höst och Alma Orucevic-Alagic. A systematic review of research on open source software in commercial software product development. *Information & Software Technology*, 53(6):616–624, 2011.
- [17] Martin Höst, Björn Regnell och Per Runeson. *Att genomföra examensarbete*. Studentlitteratur, 2006.
- [18] ISO. ISO/IEC 9126-1:2001(E) International Standard Software Engineering Product Quality Part 1: Quality Model, 2001.
- [19] Pankaj Jalote. *A Concise Introduction to Software Engineering*. Springer, 2008.
- [20] Andreas Jedlitschka, Marcus Ciolkowski och Dietmar Pfahl. Reporting experiments in software engineering. I: Forrest Shull, Janice Singer och Dag I.K. Sjøberg, redaktörer, *Guide to Advanced Empirical Software Engineering*, sid. 201–228. Springer London, 2008.
- [21] Enrico Johansson och Martin Höst. Performance prediction based on knowledge of prior product versions. I: *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR05)*, 2005.
- [22] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison Wesley, andra utgåvan, 2003.
- [23] Jussi Kasurinen och Kari Smolander. What do game developers test in their products? I: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, sid. 1–10, 2014.
- [24] Barbara A. Kitchenham och Shari L. Pfleeger. Personal opinion surveys. I: Forrest Shull, Janice Singer och Dag I.K. Sjøberg, redaktörer, *Guide to Advanced Empirical Software Engineering*, sid. 63–92. Springer London, 2008.
- [25] Barbara Ann Kitchenham, David Budgen och Pearl Brereton. *Evidence-based software engineering and systematic reviews*. CRC press, 2016.
- [26] David J. Lilja. *Measuring Computer Performance, a Practitioners Guide*. Cambridge University Press, 2000.
- [27] Thomas Mailund. *Functional Programming in R*. Apress, 2017.
- [28] Robert Martin. OO design quality metrics, an analysis of dependencies. Technical report, Object Mentor, 1994.
- [29] Douglas C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 1991.
- [30] Colin J. Neill och Phillip A. Laplante. Requirements engineering: The state of the practice. *IEEE Software*, 20(6):40–45, november 2003.

- [31] Tobias Oetiker, Hubert Partl, Irene Hyna och Elisabeth Schlegl. The not so short introduction to LATEX2e, v. 5.06. <https://tobi.oetiker.ch/lshort/lshort.pdf>, 2016.
- [32] Emmanuel Paradis. R for beginners. University of Montpellier, France, 2005.
- [33] Colin Robson. *Experiment Design and Statistics in Psychology*, 3:rd ed. Penguin Group, 1994.
- [34] Colin Robson. *Real World Research*. Wiley, 2011.
- [35] Per Runeson och Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.
- [36] Ian Sommerville. *Software Engineering*, 10:e uppl. Pearson, 2016.
- [37] Wikipedia. BibTeX – Wikipedia, the free encyclopedia, <https://en.wikipedia.org/wiki/BibTeX>, 2017. hämtad 2017-11-01.
- [38] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. I: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, 2014.
- [39] Claes Wohlin och Martin Höst. Special section: Controlled experiments in software engineering. *Information and Software Technology*, 43:921–924, 2001.
- [40] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell och Anders Wesslén. *Experimentation in Software Engineering*. Springer, 2012.
- [41] Robert K. Yin. *Case Study Research, Design and Methods*, 3:rd ed. Sage publications, 2003.

Sakregister

- afferent koppling, 38
- agil utveckling, 4
- aktionsforskning, 23
- användbarhet, 33
- beroende variabel, 19
- BibTeX, 58
- bokkapitel, 11, 61
- boxplot, 49, 99
- centralitet, 46
- CK-mått, 37
- cyklomatisk komplexitet, 35
- dataram (R), 88
- DBLP, 13
- deskriptiv analys, 43
- direkta mått, 29
- disposition, 53
- doktorsavhandling, 15
- effektivitet, 33
- efferent koppling, 38
- examensarbete, 15
- exekveringstid, 75
- experiment, 19
- externa hot, 24
- externa mått, 30
- fallstudie, 22
- fix metod, 18
- flexibel metod, 18
- forskningsfrågor, 8, 54
- forskningsmetodik, 17
- funktion (R), 95
- funktionella krav, 3
- fördelning, 44
- Google Scholar, 12
- GQM, 31
- grafritning (R), 98
- Harvard-systemet, 62
- histogram, 99
- hypotestest, 50
- högskoleförordningen, 4
- icke-funktionella krav, 3
- icke-parametriska test, 51
- IEEE Explore, 12
- IMRAD, 55
- indirekta mått, 29
- interna hot, 24
- interna mått, 30
- interpretator, 67, 84
- intervallskala, 28
- iteration (R), 96
- iterativ utveckling, 4
- Java VisualVM, 77
- JavaNCSS, 39
- JIT, 67
- just in time kompilering, 67
- kartläggning, 21
- klockfrekvens, 73
- komplexitet, 35
- konferensartikel, 10, 56, 57, 61
- konfidensintervall, 47
- kontrollerat experiment, 19
- kvalitativ metod, 18
- kvalitetsdimensioner, 33
- kvantitativ metod, 18
- lista (R), 87
- litteraturförteckning, 56
- litteraturstudie, 9
- LOC, 34
- LUBsearch, 12
- Martin-måtten, 38
- maskinkod, 65
- matris (R), 86

- McCabe's komplexitet, 35
- medelvärde, 46
- metodbeskrivning, 54
- metrics, 27
- MFLOPS, 73
- MIPS, 73
- målbeskrivning, 7
- målgrupp, 21
- målorienterad mätning, 31
- nominalskala, 28
- normalfördelning, 46
- notsystemet, 56
- oberoende variabel, 19
- open access, 12
- optimering, 67
- ordinalskala, 28
- outliers, 48
- paket (R), 97
- parentessystemet, 62
- peer review, 10
- PMD, 40
- portabilitet, 34, 67
- poster, 11
- processmått, 30
- produktmått, 30
- profileringsverktyg, 77
- R, 83
- randomisering, 20
- ratioskala, 28
- referensdatabas, 12
- relaterat arbete, 9, 54
- replikering, 18
- resursmått, 30
- sannolikhet, 43
- Scopus, 12
- short paper, 11
- siffersystemet, 56
- sortering (R), 92
- SPEC, 73
- spridningsmått, 47
- standardavvikelse, 47
- stapeldiagram, 99
- storleksmått, 34
- subjektiva mått, 29
- söksträng, 13
- t-test, 51
- testning, 4
- tidskriftsartikel, 10, 56, 57, 61
- tillförlitlighet, 33
- typvärde, 46
- tårtdiagram, 99
- underhållsbarhet, 33
- urval (kartläggning), 21
- validitet, 21, 24
- validitetshot, 24
- variationsvidd, 47
- vektor (R), 84
- vetenskaplig artikel, 10
- Web of Science, 12
- Wikipedia, 16
- workshopartikel, 11