Kodgenerering i CASE-verktyg – En undersökning hur CASE-verktyg uppfyller experters kodgenereringskrav

(HS-IDA-EA-01-601)

Martin Andersson (b97maran@student.his.se)
Institutionen för Datavetenskap
Högskolan i Skövde, Box 408
S-541 28 Skövde, SWEDEN

Examensarbete i datavetenskap 10 poäng under vårterminen 2001.

Handledare: Björn Lundell

Kodgenerering i CASE-verktyg – En undersökning hur CASE-verktyg uppfyller experters kodgenereringskrav

Examensrapport inlämnad av Martin Andersson till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för Datavetenskap.

28 maj, 2001

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

Signerat: _____

Kodgenerering i CASE-verktyg – En undersökning hur CASE-verktyg uppfyller experters kodgenereringskrav

Martin Andersson (b97maran@student.his.se)

Sammanfattning

Denna rapport undersöker krav, tagna från ett ramverk för evaluering av CASE-verktyg i ett kontextuellt sammanhang, i två representativa CASE-verktyg. Ramverket utnyttjar en modell som föreslagits av Lundell och Lings för att extrahera både krav och förväntningar som en organisation (www.it.volvo.com) hade på vad ett CASE-verktyg är och kan utföra.

Ramverket extraherar krav i ett organisationell kontext, dvs. utvärderingen utfördes innan verktyget som evaluerades användes i organisationen. Detta indikerar på att kraven inte är knutna till ett specifikt verktyg, samt att CASE-verktyg inte säkert stödjer dessa krav.

Resultatet för denna rapport är att viss semantisk förlust uppstod vid transformering av kod och modeller.

Nyckelord: CASE-verktyg, Kodgenerering, Reverse engineering, Round-trip engineering, Microsoft Visio 2000 Enterprise, TogetherSoft Together ControlCenter

Kodgenerering i CASE-verktyg – En undersökning hur CASE-verktyg uppfyller experters kodgenereringskrav

Martin Andersson (b97maran@student.his.se)

Abstract

This report evaluates demands, taken from a context dependent CASE-tool evaluation framework, in two representative CASE-tools. The framework used a model proposed by Lundell and Lings to extract both requirements and expectations that an organization (www.it.volvo.com) had about what a CASE-tool is and what it can do.

The framework extract requirements in an organizational context, i.e. the evaluation took place before the tool being evaluated was in real use in the oranizational setting. This implies that the demands are not tied to a specific tool, and current CASE-tools may not support these demands.

The outcome of this report is that some semantic errors were introduced when models and code were transformed.

Keywords: CASE-tool, Codegeneration, Reverse engineering, Round-trip engineering, Microsoft Visio 2000 Enterprise, TogetherSoft Together ControlCenter

Innehållsförteckning

Förord

Denna rapport har möjliggjorts tack vare flera personers råd och stöd. Framförallt vill jag tack min handledare Björn Lundell för värdefulla synpunkter, ett kritiskt granskande samt artiklar och annat material.

Jag vill också tacka Adam Rehbinder för tillåtelse att använda och återge delar av hans ramverk från dissertationen ”On Applying a Method for Developing Context Dependent CASE-tool Evaluation Frameworks”.

Tack även till Shahin Seifzadeh för tillstånd att utnyttja modeller från hans examensarbetet ”Kodgenereringsmöjligheter i Visio 2000 Enterprise”.

Skövde i maj 2001

Martin Andersson

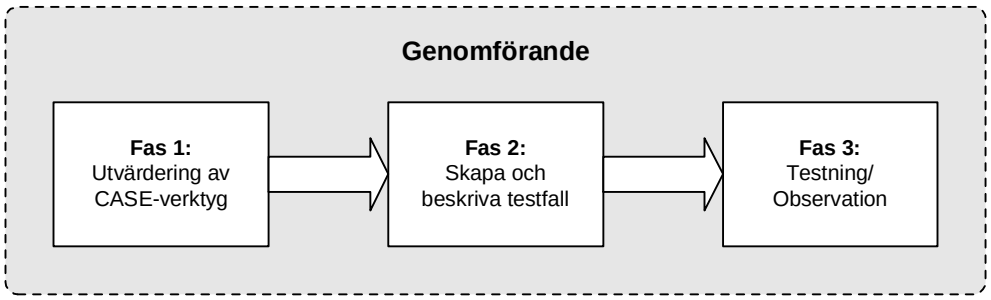
1 Introduktion

Syftet med denna rapport är att evaluera hur experters kodgenereringskrav uppfylls i dagens CASE-verktyg. Undersökningen försöker besvara hur kodgenerering och reverse engineering hanteras, samt huruvida det uppstår semantiska förluster under designtransformationen.

Rapport undersöker krav, tagna från ett ramverk (Rehbinder, 2000) för evaluering av CASE-verktyg i ett kontextuellt sammanhang, i två representativa CASE-verktyg. Dessa krav innehåller både förväntningar och krav som CASE-experter har på moderna CASE-verktyg.

De krav som valts att undersökas i rapporten är en delmängd av krav som har identifierats i ovanstående ramverk och tar upp kodgenereringskrav. Två representativa CASE-verktyg har valts ut och 13 testfall har skapats för att analysera dessa krav (figur 1.1).

Testfallen utnyttjar modeller från en rapport som undersöker kodgenererings-möjligheter i CASE-verktyget Visio 2000 Enterprise (Seifzadeh, 2000). Dessa modeller är i sin tur baserade på en modell som skapats av samma organisation som Rehbinder utnyttjade för att skapa hans ramverk.



Figur 1.1 *Plan för genomförande.*

Testfallen har sedan metodiskt utförts i CASE-verktygen. Testfall 8 – 13, som beskriver reverse engineering, har endast utförts i ett av verktygen då författaren inte haft tillgång till de verktyg som krävs för att utföra denna process i Microsoft Visio 2000 Enterprise Edition.

Efter att testfallen utförts och redovisats har en analys utförts. Detta steg analyserar hur resultatet från undersökningen uppfyller de krav som utnyttjats från Rehbinders ramverkt för att utforma problemformuleringen.

Resultatet för denna rapport är att vissa semantiska förluster uppstår vid transformering av kod och modeller. De båda CASE-verktygen uppfyller alltså inte alla de krav som har undersökts. Dock anser författaren att denna undersökning har gett en större insikt i vad de två verktygen kan utföra och att de kan utnyttjas som en hjälp i ett mjukvaruutvecklingsprojekt.

2 Bakgrund

Under 1970-talet fann ingenjörer att de flesta fel och buggar inträffade under analys-, planerings- och designfaserna i ett mjukvaruutvecklingsprojekt (Lewis, 1991: 15). Fisher (1991: 4-5) menar att så mycket som 64 procent av felen uppstår i dessa steg och endast 36 procent av felen upptäcks under implementationsfasen.

Detta visade på hur viktigt det är att utföra en noggrann analys i de första faserna av ett projekt. För att hjälpa till med denna process uppstod en utveckling av en helt ny kategori metodologier – föregångarna till CASE-verktyg. Dessa metoder stödde analys, modellering eller dokumentation av mjukvaran (Lewis, 1991: 3-4, 16).

2.1 Mjukvaruutveckling

Dagens CASE-verktyg kan stödja hela processen i ett mjukvaruutvecklingsprojekt eller endast en delmängd av dess faser. Enligt Humphrey (1991: 60) används CASE-verktyg främst för att förbättra kvaliteten och produktiviteten i en mjukvaru-utveckling. För att i följande kapitel kunna särskilja och definiera olika typer av CASE-verktyg så beskriver nedan vilka steg en mjukvaruutvecklingsprocess vanligtvis innehåller.

Flecher & Hunt (1993: 35) definierar mjukvaruutveckling som ”en integrerad mängd av metoder, procedurer och verktyg för att specificera, designa, utveckla och underhålla mjukvara”. I denna rapport likställs mjukvaruutveckling (eng. software engineering) med *en logisk följd av aktiviteter, som avser analys, konstruktion och införande av ett system för informationsbehandling*.

En annat vanligt förekommande term är *systemutveckling* eller *informationssystem-utveckling* (eng. system development, information system development). Enligt Andersen (1994: 48) ingår följande faser i systemutveckling:

- Analys
- Utformning
- Realisering
- Implementering

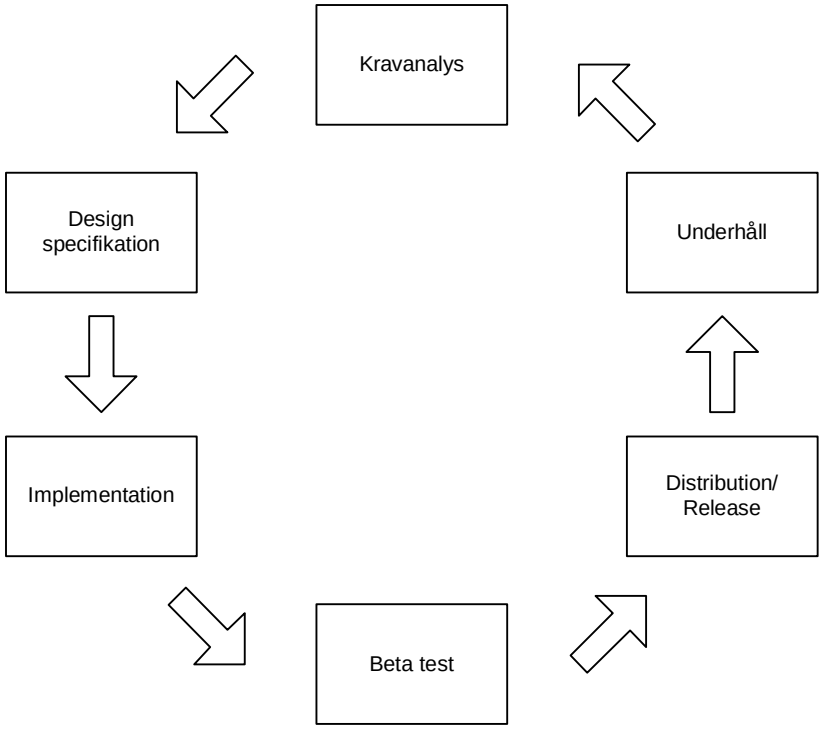
I denna rapport görs ingen skillnad på mjukvaruutveckling och systemutveckling, dock skiljer viss litteratur på dessa begrepp. Systemutveckling är processen att skapa ett informationssystem (Andersen, 1994: 9). Andersen (1994: 15) definierar ett informationssystem som ”…ett system för insamling, bearbetning, lagring, överföring och presentation av information”. King (1995: 4) och Andersen (1994: 15-16) menar att ett informationssystem kan innefatta människor och deras rutiner som en del i ett informationssystem.

Ordet mjukvaruutveckling antyder att man skapar *mjukvara*. Enligt Malmström *et al*. (1991: 424) är mjukvara synonym för *programvara* eller *software* och är *system och*

program i en dator. I detta arbete används system, mjukvara och programvara som synonymer.

2.1.1 Process

Enligt Fisher (1991: 7-13) kan mjukvaruutveckling ses som en cyklisk process som innehåller flera olika faser (figur 2.1). Fisher medger att processer vanligtvis har ett slut, men hävdar att vid mjukvaruutveckling krävs löpande underhåll av systemet, samt om den blir framgångsrik sker oftast en kontinuerlig utveckling.



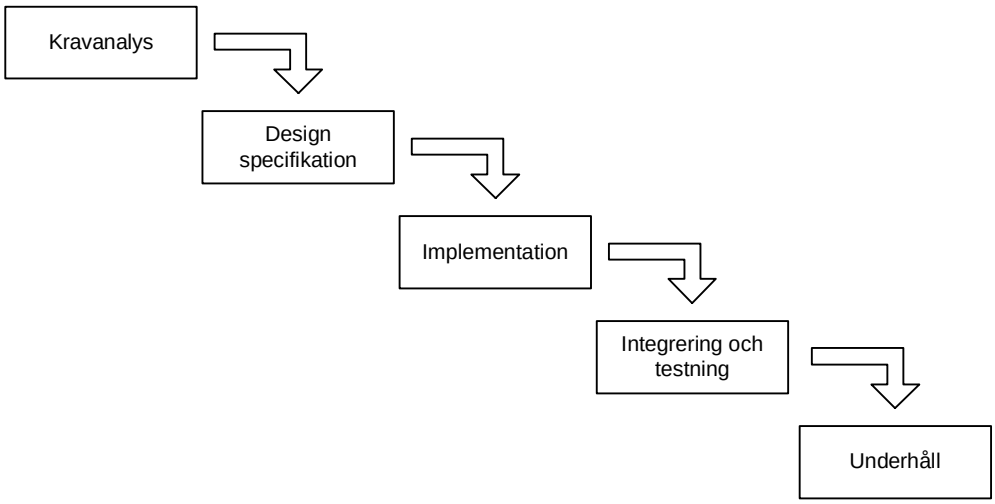
Figur 2.1 *Modell som beskriver mjukvaruutveckling som en cyklisk process.*

Stegen i Fishers modell utförs oftast linjärt efter varandra, men det förekommer att man går tillbaka till ett tidigare steg under projektets livstid. Vad man uträttar i de olika faserna i beskrivs nedan:

- **Kravanalys**. Skapa en kravspecifikation utifrån användarnas krav. Denna specifikation innehåller funktionalitetskrav, hårdvarukrav, krav på användargränssnitt, samt prestandakrav för mjukvaran.
- **Design specifikation**. Specificera moduler, datastrukturer, algoritmer m.m. för hur mjukvaran ska implementeras.
- **Implementation**. Implementera, testa och debugga de moduler som specificerats i designfasen.
- **Enhetstest och integrering**. Testning av varje modul, därefter sammanfogas modulerna och testas om huruvida de fungerar tillsammans och enligt specifikationen.

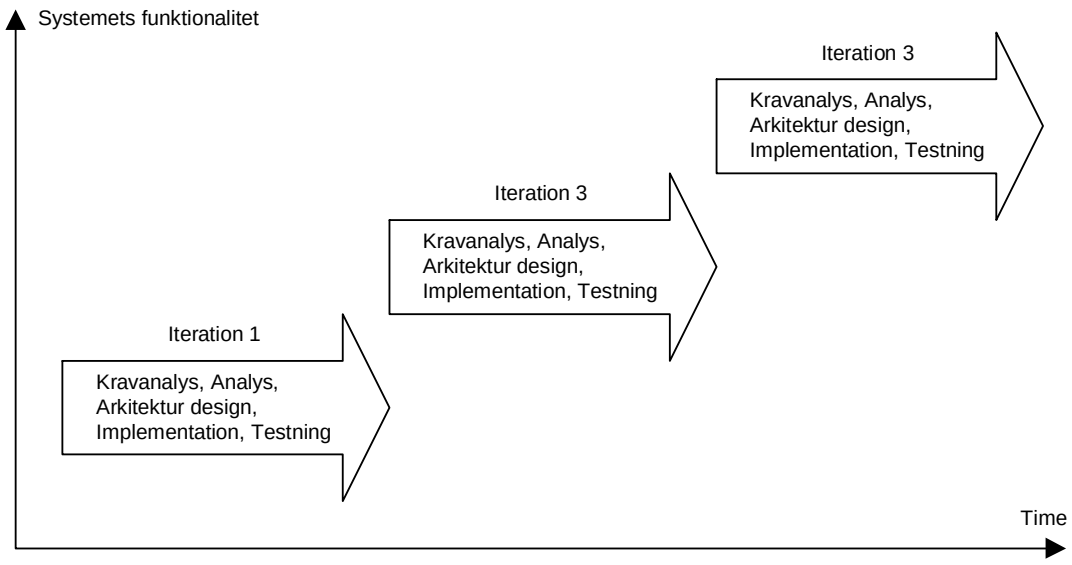
- **Beta test**. Utvärderar mjukvaran för att upptäcka buggar, prestandaproblem m.m.
- **Distribution/Release**. Distribuerar den färdiga mjukvaran till användarna.
- **Underhåll**. Rättar till buggar eller problem som upptäckts i den distribuerade mjukvaran.

Ovanstående modell kan enligt Fisher (1991) jämställas med den s.k. *vattenfall-modellen* som används av många utvecklare (figur 2.2).



Figur 2.2 *Vattenfallsmodellen.*

Eriksson & Penker (2000: 354-357) har en liknande modell för mjukvaruutveckling som Fisher, men menar att deras modell är tänkt att används iterativt (figur 2.3). Vid varje iteration utökas funktionalitet för systemet, vilket innebär att problem kan upptäckas kontinuerligt. Detta i sin tur medför att projektet kan kontrolleras bättre och fel upptäcks tidigare och hanteras bättre.



Figur 2.3 *Modell som beskriver mjukvaruutveckling som en iterativ process.*

Motståndare till denna modell (även kallad *evolutionär utvecklingsstrategi*) menar enligt Andersen (1994: 348-349) att uppdelningen av utvecklingsarbetet är ett slöseri med resurser, samt ger mindre effektiva tekniska lösningar.

2.1.2 Metoder

En *metod* är enligt Andersen (1994: 102) en detaljerad beskrivning för hur man löser ett visst problem. Om en metod används på ett problem av två personer bör de komma fram till samma resultat oberoende av varandra. En metod karakteriseras av (Andersen, 1994: 102):

- Användningsområde; på vilken typ av problem den kan tillämpas.
- Vilket arbete som ska utföras och ev. hur detta arbete bör organiseras.
- Vilka beskrivningstekniker som ska användas och hur.

En metod innehåller oftast, men inte alltid, beskrivningstekniker som lämpar sig för just denna metod (Andersen, 1994: 103).

Mjukvaruutveckling med hjälp av CASE-verktyg innebär oftast att man måste arbeta enligt den/de metod(er) som finns implementerade i verktyget (Cronholm, 1994: 6; Juric & Kuljis, 1999:1). Detta innebär att man måste ta hänsyn till befintliga metoder som används i en organisation när man ska välja ett CASE-verktyg.

Cronholm (1994: 47) delar in CASE-verktyg i två kategorier beroende på vilken grad av aktivt metodstöd de har:

- **Metodledning**. Verktyget kan guida utvecklaren i arbetet genom att t.ex. se till att olika steg i metoden utförs i rätt ordning.
- **Expertstöd**. Verktyget har en viss kunskap om arbetsprocessen och kan ge råd och tips på lösningar.

2.2 Modelleringsspråk och modeller

2.2.1 Modelleringsspråk

En *beskrivningsteknik* innehåller en uppsättning regler för hur verkligheten kan uttryckas med hjälp av en beskrivning (Andersen, 1994: 50, 103-104). Dessa regler kan beskriva:

- Vilka symboler som är tillåtna.
- Vilka symbolkombinationer som är tillåtna.
- På vilket sätt man använder text (naturligt språk) i anslutning till symbolerna.
- Användning av olika nivåer i beskrivningen (dvs. om beskrivningen är hierarkiskt uppbyggd eller ej).

En vanligt förekommande synonym för beskrivningsteknik är *modelleringsspråk*. Eriksson & Penker (2000: 17) delar upp ett modelleringsspråk i dess *notation*, symboler som används i modellen/beskrivningen, och dess *regler* som reglerar språket (Eriksson & Penker, 2000: 17).

Enligt SISU (1991) består ett modelleringsspråk av:

- En uppsättning modelleringsbegrepp.
- En grafisk notation.
- En verbal/textuell notation som kan vara mer eller mindre strukturerad och formell.

Unified Modeling Language, UML, är en notation som har blivit en standard inom mjukvaruutveckling och de flesta verktygen på marknaden har implementerat support för detta språk (Eriksson & Penker, 2000: 5; Larman, 1998: xv). Enligt OMG (1999: xi) används UML för att specificera, visualisera, konstruera och dokumentera mjukvara.

2.2.2 Beskrivningar och modeller

Under systemutvecklingsprocessen produceras olika typer av *beskrivningar* för att kunna analysera verkligheten (Cronholm, 1994: 41). I litteraturen används även *modeller* och *diagram* som en synonym för beskrivningar.

En beskrivning möjliggör att man kan få bättre insikt i verkligheten och tillåter att man selekterar ut de sidor av verkligheten som man vill analysera (Andersen, 1994: 50-52). Enligt Cronholm (1994: 41) kan man dela in beskrivningar in sex olika *beskrivningstyper*:

- Fri text.
- Strukturerad text.
- Formell notation.
- Listor.
- Matriser.
- Diagram.

UML innehåller nio olika diagramtyper, som används för att beskriver ett systems struktur, funktionalitet och beteende, dessa beskrivs kortfattat nedan (Eriksson & Penker, 2000: 5, 18-19):

- **Klassdiagram (class diagram)**. Beskriver strukturen av systemet.
- **Objektdiagram (object diagram)**. Exemplifierar möjliga objektkombinationer av ett specifikt klassdiagram.
- **Tillståndsdiagram (state chart diagram)**. Exemplifierar möjliga tillstånd av en klass (eller ett system).

- **Aktivitetsdiagram (activity diagram)**. Beskriver aktiviteter och händelser som sker i ett system.
- **Sekvensdiagram (sequence diagram)**. Beskriver en eller flera följder av meddelanden som sänds mellan objekt.
- **Kollaborationsdiagram (collaboration diagram)**. Beskriver samverkan mellan objekt.
- **Use-case diagram**. Illustrerar de funktionella kraven som existerar för ett system.
- **Komponentdiagram (component diagram)**. En speciell typ av klassdiagram som beskriver komponenterna i ett system.
- **Distributionsdiagram (deployment diagram)**. En speciell typ av klassdiagram som beskriver hårdvaran i ett system.

2.3 CASE

Vid slutet av 1980-talet lanserades begreppet CASE (Nilsson, 1995: 16). Det finns ingen enig definition av CASE (King, 1995: 4). Enligt IEEE (1990: 15) är CASE en akronym för *Computer-Aided Software Engineering*. En annan vanlig tolkning av CASE är *Computer-Aided System Engineering* (King, 1995: 4). Carnegie Mellon University (2001) definierar CASE som ”the use of computer-based support in the software development process.”

2.3.1 CASE-verktyg

Eriksson & Penker (2000: 88) och King (1995: 3) hävdar att CASE-verktyg ofta har setts som ”mirakelverktyg” av marknaden. Förväntningarna på CASE-verktyg har ofta varit högre än befintlig funktionalitet hos verktygen (Rehbinder, 2000: 1), vilket i sin tur har inneburit svårigheter vid införandet av CASE-verktyg (ISO/IEC, 1999: v). Författarna menar att förväntningarna på CASE-verktygen har varit en överdrift, de bör istället ses som en administrativ hjälp som förenklar skapandet av modeller.

CASE-verktyg förenklar enligt Andersen (1994: 109) mjukvaruutvecklingen. Verktygen låter utvecklarna koncentrera sig på systemets arkitektur snarare än implementationen (Fisher, 1991: 5). Enligt Cronholm (1994: 4-5) är följande olika motiv för att använda CASE-verktyg:

- Höja produktiviteten i utvecklingsprojektet.
- Förbättrad kvalitet på dokumentationen.
- Underlätta möjligheter till underhåll.
- Kortare projekttider.
- Ökad standardisering.

Eriksson & Penker (2000: 247) skiljer på CASE-verktyg inriktade på:

- Kravframställning.
- Analys och design.
- Verksamhetsmodellering.

Enligt Carnegie Mellon University (1999) är ett CASE-verktyg ”a computer-based product aimed at supporting one or more software engineering activities within a software development process”. Eriksson & Penker (2000: 247) definierar CASE-verktyg som “program tools that support the development of software systems”.

Fördelar med att utnyttja CASE-verktyg är (Andersen, 1994: 109; Fisher, 1991: 24-25):

- Modelleringsarbetet blir enklare; enklare och mindre arbetskrävande att utföra vissa förändringar.
- Man undviker vissa typer av fel; verktyget kontrollerar modellerna, antingen automatiskt eller på begäran.
- Färre fel och snabbare utveckling; utvecklarna och användarna tvingas till större disciplin tack vare att de formella reglerna i verktyget måste följas.
- Automatisk generering av kod samt synkronisering av kod och modell.
- Mer läsbar dokumentation; datorgenererade beskrivningar.
- Enklare att angripa mer omfattande problem än vid manuell dokumentation.

En av CASE-verktygens nackdelar är enligt Andersen (1994: 109) att de oftast är mycket dyra. En annan nackdel är att för formella verktyg kan begränsa handlingsfriheten och den kreativa processen.

2.3.2 Olika kategorier av CASE-verktyg

Andersen (1994: 109-111) delar in CASE-verktyg utifrån vilket slags stöd de ger:

- **Stöd till generella beskrivningar**. Har inga fördefinierade symboler eller stöd för en viss beskrivningsteknik.
- **Stöd till användning av en beskrivningsteknik**. Har fördefinierade symboler med åtanke på en bestämd beskrivningsteknik.
- **Stöd till användning av en metod**. Tar hänsyn till en metod kräver att beskrivningsarbetet utförs på ett visst sätt.
- **Stöd till egen beskrivning av metod eller beskrivningsteknik (CASE-skal)**. Innehåller inte en bestämd beskrivningsteknik eller metod, utan användaren kan själv lägga in sin(a) metod(er) och/eller beskrivningstekniker. Programmet måste alltså ”fyllas” med ytterligare innehåll för att kunna användas.
- **Stöd till en metod i analysfasen och automatisering av utformnings- och realiseringsfasen (integrerat CASE)**. Stödjer både analysarbetet och utformnings- och implementationsarbetet.

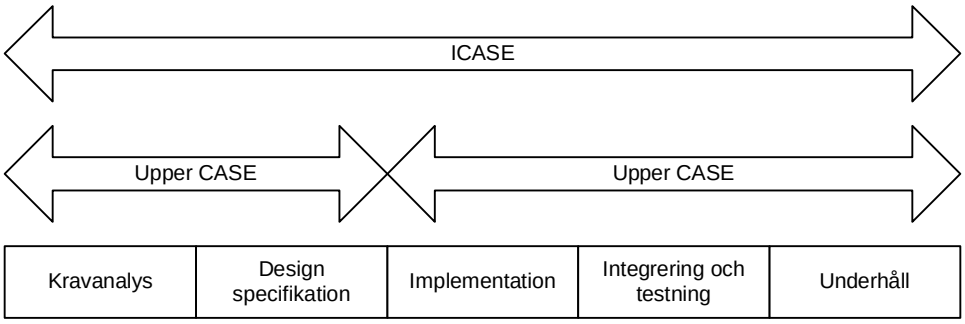
Process Improvement Associates (2001) delar upp CASE-verktyg i 8 olika kategorier beroende på vilka funktioner som verktyget stödjer (tabell 2.1). Denna tabell ger även en översikt vad olika CASE-verktyg kan utföra.

Kravanalys och design • Behavioristisk/data/objekt/funktionell modellering. • Kravvalidering/framställning. • Kravspårning.	**Konstruktion** • Applikationsgenerering. • Kodskelett generering. • GUI/UI utveckling. • Expertsystem utveckling. • Programmering
Testning • Prestanda analys. • Regressions testning. • Systemintegrations testning. • Testning och debuggning.	**Projektplanering och spårning** • Portfolio analys. • Förändringsanalys. • Organisationsplanering och modellering. • Processmodellering. • Kostnads och storleksvärdering. • Projektstyrning. • Flödeskontroll. • Gruppsupport.
Dokumentation • Dokumentering.	**Kvalitetskontroll, kodanalys och metrik** • Kvalitetskontroll. • Felsökning. • Reverse Engineering. • Språköversättning. • Metrik insamling/analys.
Konfigurationshantering • Konfigurationshantering.	**Integrerad mjukvaruutvecklingsmiljö** • Metamodell definition. • Modelltransformation. • Repository administration. • Repository export/import. • Verktygsutveckling/integration.

Tabell 2.1 *Olika kategorier av CASE-verktyg och dess funktion.*

Vidare definierar King (1995: 5-6) och Cronholm (1994:9) en vanligt förekommande indelning av CASE-verktyg (figur 2.4):

- **Upper CASE**. Verktyg för automatisering av analys fasen.
- **Lower CASE**. Verktyg för automatisering av implementeringsfasen.
- **Integrated CASE (ICASE)**. Verktyg som omfatta både upper och lower CASE verktyg.



Figur 2.4 *Indelning av CASE-verktyg enligt King.*

2.4 Krav på CASE-verktyg

Ett grundläggande krav som särskiljer CASE-verktyg från verktyg på lägre teknik-nivå, exempelvis ritverktyg eller papper och penna, är att CASE-verktyg innehåller en underliggande semantik medan ritverktyg endast kan erbjuda grafiska beskrivningar (Cronholm, 1994: 48). Fisher (1991: 32-33) anser att ett CASE-verktyg måste uppfylla följande kriterier:

- **Förenkla**. Dela upp krav och designspecifikationer i lätthanterliga delar.
- **Passa flera intressenter**. Verktygets output måste kunna förstås av flera olika typer av intressenter; både slutanvändare och utvecklare.
- **Spara tid och pengar**. Att använda verktyget måste vara billigare och effektivare i längden än att använda traditionell utveckling i implementations- och underhålls-fasen.
- **Producera kvalitativ och kontrollerbar designspecifikation**. Varje krav i implementationen måste kunna spåras och verifieras tillbaka till kravspecifika-tionen.
- **Stödja förändringar**. Krav och specifikationer skapade av verktyget måste kunna anpassas och då projektet förändras.
- **Visuellt stöd**. Verktyget måste kunna visa information grafiskt.

Vidare bör ett CASE-verktyg innehålla följande komponenter (Cronholm, 1994: 44, 49):

- Grafiska editorer för en eller flera beskrivningstekniker.
- Ett repository där allt som verktyget producerar kan lagras på ett strukturerat sätt.
- Fråge- och rapportgenereringsfunktion som möjliggör produktion av listor och rapporter utifrån innehållet i repository.
- Analys- och kontrollfunktioner av dokumentationen.
- Transformering t.ex. mellan olika beskrivningstekniker eller från design-beskrivning till kod.
- Export- och importfunktioner som utgör ett gränssnitt mot andra verktyg.
- Vissa funktioner för projektplanering.
- Metodstöd.

Rehbinder (2000) har utvecklat ett ramverk med krav på CASE-verktyg i samarbete med CASE-experter på Volvo IT i Skövde och Göteborg. I denna rapport finns krav definierade inom följande områden:

- CASE.
- CASE user facilities.
- CASE transparent facilities.
- Standards.
- Interoperability
- Tool migration.
- Comments.
- Repository.
- Documentation.
- Notational support.
- Components.
- Code generation.
- Database support.
- ISD life cycle support.

2.5 Kodgenerering

Kodgenerering är troligen den mest upphaussade funktion hos CASE-verktyg (Fowler, 2001) och är processen att automatiskt skapa mjukvara direkt från en design-specifikation (Fisher, 1991: 30). Enligt Aimar *et al.* (2001) innebär automatisk kodgenerering att man säkerställer konsistensen mellan design och implementations-stegen under mjukvaruutvecklingen.

Enligt Barclay & Padusenko (2001) innebär kodgenerering tids- och kostnadsvinster för ett mjukvaruutvecklingsprojekt, samt framställning av kod som är enklare att underhålla och är portabel mellan olika hårdvaruplattformar.

Olika CASE-verktyg stödjer olika "sorters" kodgenerering. Fowler (2001) menar att de flesta verktyg endast är s.k. *interface-generators* som skapar klassdefinitioner med attribut och operationer. Detta innebär alltså att endast interfacet till mjukvaran skapas, och implementationen, huvudkoden, måste skapas för hand. En synonym för *interface-generators* är *kodskelett*. Vidare hävdar Fowler (2001) att ett CASE-verktyg bör ge användaren valmöjligheter för hur koden ska genereras.

2.6 Reverse och round-trip engineering

Processen att skapa logiska modeller från exekverbar källkod kallas *reverse engineering* (Larman, 1998: 298). Denna process innebär att man tar en existerande källkodsfil och importerar den in i ett CASE-verktyg (Fisher, 1991: 146). Koden kan modifieras och förbättras i verktyget för att sedan generera ny källkod. Denna ”cirkulära” användning av kodgenerering och reverse engineering kallas ofta för *round-trip engineering* (Popkin, 2001).

2.7 Relaterade arbeten

Shahin Seifzadeh (Seifzadeh, 2000) har evaluerat kodgenereringsmöjligheter i Visio 2000 Enterprise Edition. Seifzadehs rapport undersöker hur Visio hanterar kodgenerering och reverse engineering med modeller med UML notation och programmeringsspråket C++.

Adam Rehbinder (Rehbinder, 2000) har skapat ett ramverk för evaluering av CASE-verktyg. I hans dissertation identifieras krav och förväntningar på CASE-verktyg från CASE-experter, och en delmängd av dessa krav utnyttjas i detta arbete. I ett av stegen som utförs i Rehbiders undersökning används även Visio 2000 Enterprise Edition för att evaluera kraven som framkommit (Rehbinder *et al.*, 2001).

Liknande arbeten med att evaluera designtransformeringar i CASE-verktyg har utförts i en mängd olika verktyg och med olika förutsättningar och krav. Några exempel är Post & Kagan (2000) som evaluerar Rational Rose, Örn Kristinsson (1997) som

“OO-CASE tools: an evaluation of Rose”, Information and Software Technology, 42(6), pp. 383-388

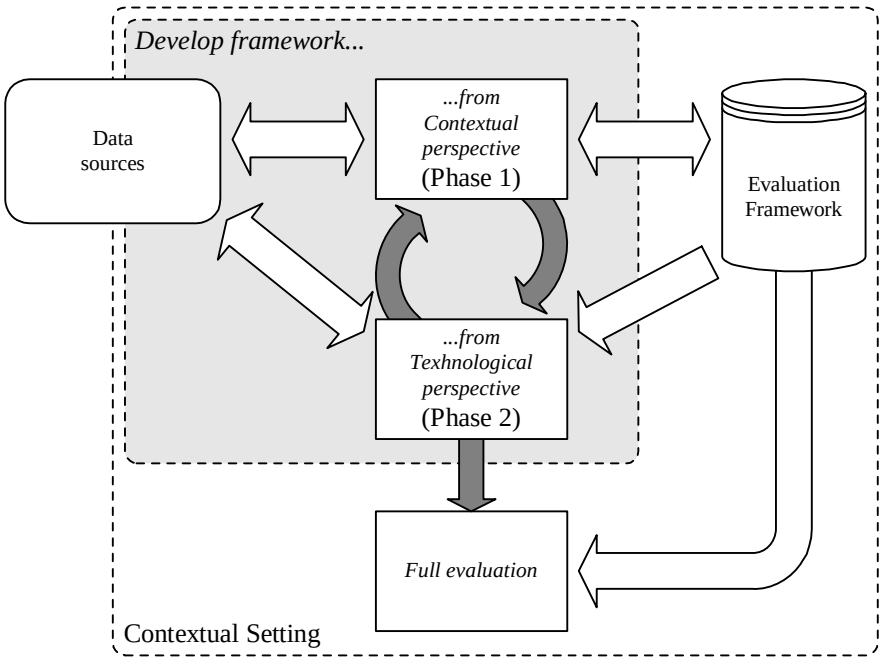
3 Problemprecisering

I detta kapitel preciseras en frågeställning utifrån det problemområde som har tagits upp i bakgrunden. Vidare beskrivs vilka begränsningar som har gjorts av problemområdet samt vilket resultat som författaren tror sig finna.

CASE-verktyg innebär enligt många författare (kapitel 2) fördelar såsom kostnads- och produktivitetsförbättring i en mjukvaruutveckling. Dock har organisationer historiskt sett fått svårigheter vid införandet av CASE-verktyg i sina informationssystem (ISO/IEC, 1999: v). Detta har lett till uppkomsten av olika metoder för evaluering av CASE-verktyg (Rehbinder, 2000: 2).

En av dessa metoder för evaluering av CASE-verktyg har föreslagits av Björn Lundell och Brian Lings (Lundell & Lings, 1999). Denna metod har använts i en dissertation av Adam Rehbinder (Rehbinder, 2000) för skapandet av ett ramverk med krav på CASE-verktyg. Dessa krav har framtagits med hjälp av CASE-experter på Volvo IT, en organisation med 2 500 anställda i Skövde och Göteborg (Rehbinder *et al*., 2001: 2).

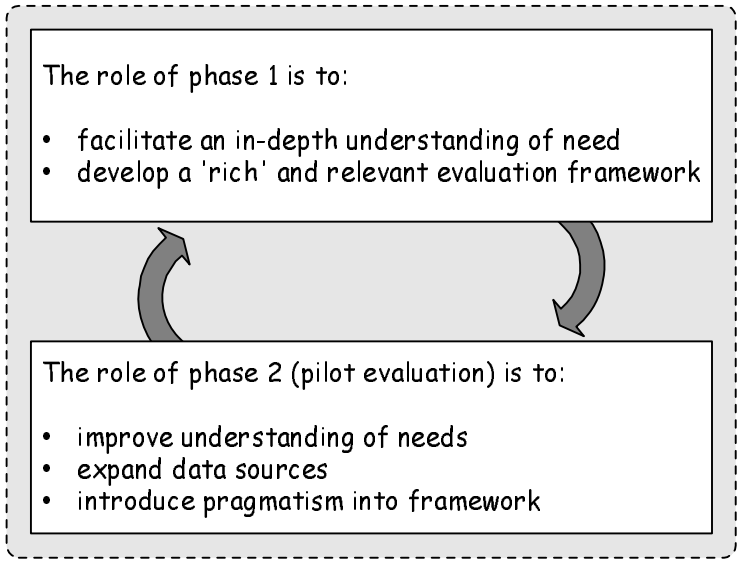
Lundell och Lings metod skiljer sig från existerande generella informationssystemsmetoder genom att stödja systematiskt utforskande av de tekniska aspekterna i en evaluering (Rehbinder *et al*., 2001: 3). Metod har även en ”grundande” approach för att skapa av ett ramverk för evaluering av CASE-verktyg (Rehbinder, 2000: 2). Detta innebär att kraven på verktygen får en organisationskontextuell grund, i motsats till a priori uppsatta krav.



Figur 3.1 *Faser och dataflöden i Lundell och Lings modell.*[1]

1 Rehbinder et al. (2000: 2, Figure 1).

Kortfattat kan Lundell och Lings metod beskrivas som en evolutionär process som består av datainsamling, analys och kodning (figur 3.1) (Rehbinder *et al*., 2000: 2). Under den första fasen insamlas och genereras dokumentation som ligger till grund till ett relevant evalueringsramverk för organisationen (figur 3.2). Denna fas fokuserar på krav och förväntningar som organisationen har på vad ett CASE-verktyg är och vad det kan utföra (Rehbinder, 2000: 6). Målet med fasen är även att får en större förståelse för behoven [av ett CASE-verktyg] som finns i organisationen (Rehbinder *et al*., 2000: 2).

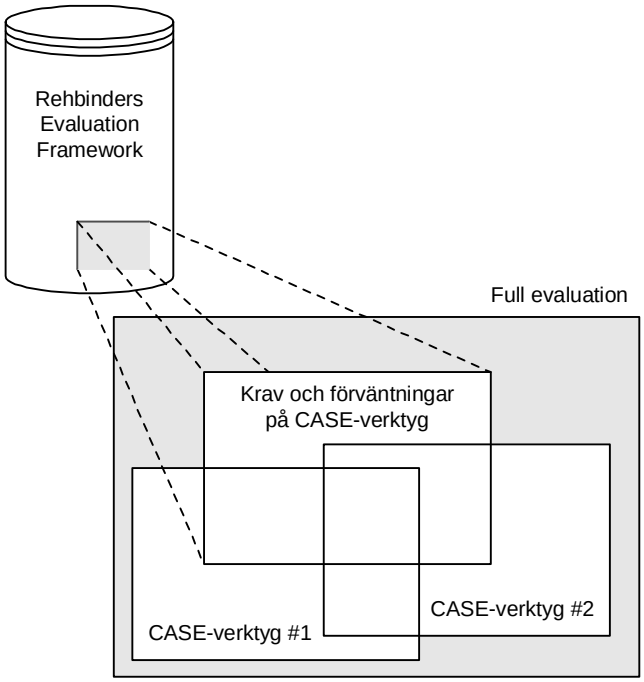


Figur 3.2 *De två fasernas roll i Lundell och Lings modell*.[2]

När de flesta kraven har framtagits och ramverket anses vara tillräckligt stabil, övergår metoden till fas 2 (Rehbinder, 2000: 7). Huvuddragen för den andra fasen är att utöka datakällorna, implementera pragmatism i ramverket, samt att få ytterligare förståelsen för organisationens behov (Rehbinder *et al*., 2000: 2; Rehbinder *et al*., 2001: 3). Detta utförs genom att utföra en pilotstudie där kraven från den första fasen evalueras med hjälp av ett CASE-verktyg (Rehbinder, 2000: 6). Om det framkommer ny information i denna fas kan detta medföra att processen går tillbaka till fas 1.

Då Rehbinder har använt ovanstående metod i sin dissertation[3] innebär detta hans ramverk innehåller krav som är framtagna med en organisationskontextuell grund. Denna rapport kommer att evaluera ett begränsat antal krav från Rehbinders dissertation i två moderna CASE-verktyg. Detta kan jämställas med att utföra en del av stegen i ”Full evaluation” från Lundell och Lings metod, med skillnad att denna rapport endast evaluerar en delmängd av kraven från ramverket (figur 3.3).

[2] Rehbinder *et al*. (2000: 3, Figure 2).
[3] Rehbinder (2000).



Figur 3.3 *Illustration av problemområdet.*

Kraven som denna rapport undersöker är, som tidigare nämnts, tagna från ett ramverk skapat av Adam Rehbinder (Rehbinder, 2000) och behandlar kodgenereringskrav för CASE-verktyg. Rehbinder (2000: 88) hävdar att en evaluering av CASE-verktyg kan baseras på krav som utvunnits från den första fasen[4]. I denna rapport kommer följande delmängd av Rehbinders ramverk att användas (Rehbinder, 2000: Appendix A):

Program skeleton from UML diagrams.

- Use cases, class diagrams, and sequence diagrams. From these it should at least be possible to auto generate stubs when work with the models is sufficiently concluded.

Code generation in any 3GL Language.

- The tool could furthermore generate code using any 3GL language, however Java is explicitly preferred.

Code generation tool independent.

- When CASE-tools generate code this code should not be CASE-tool specific. Thus when building code and when employing it in runtime the less influence the CASE-tool has the better.

Code platform independence.

- Important that the code generated is understandable and that it is independent of platforms and of the tool that generated it.

[4] Rehbinder (2000: 101-166)

Round-trip engineering

- Support for true round trip engineering where both models and the resulting implementation allows changes that are propagated in between.
- Visualising code by drawing models, being able to change the code, and being able to reverse engineer code. There should in this respect also be support for true round trip engineering.
- Desirable to go back from the implementation to the conceptual model even if the implementation is slightly altered so data has to be interchanged in between e.g. by support of a repository.

Reverse engineering

- Being able to reverse engineer code. There should in this respect be support for true round trip engineering.
- Round trip engineering should also be employed in reverse engineering.

3.1 Problemformulering

Syftet med denna rapport är att undersöka hur två representativa CASE-verktyg uppfyller empiriskt grundade kodgenereringskrav som har identifierats i en specifik organisation (Rehbinder, 2000). Utifrån de identifierade kraven har följande frågor hämtats:

- Vilket stöd ger verktygen för att skapa kodskelett från UML diagram? (Rehbinder, 2000: 150)
- Vilket stöd ger verktygen för att skapa verktygsoberoende kod? (Rehbinder, 2000: 155)
- Vilket stöd ger verktygen för att skapa plattformsoberoende kod? (Rehbinder, 2000: 155)
- Vilket stöd ger verktygen för reverse engineering? (Rehbinder, 2000: 156-157)

3.2 Avgränsning

Denna rapport undersöker kodgenereringskrav på CASE-verktyg och täcker endast en delmängd av de krav som återfinns i Rehbinders ramverk (Rehbinder, 2000). För att utvärdera verktygens förmåga till designtransformation och kodgenerering används modeller från ett examensarbete av Shahin Seifzadeh (Seifzadeh, 2000).

Seifzadeh har dels utnyttjat modeller som skapats på Volvo IT, och dels kompletterat med egna modeller för att undersöka kodgenereringsmöjligheter i Visio 2000 Enterprise (Appendix C). I Seifzadehs rapport användes programmeringsspråket C++, men i denna rapport kommer dessa modeller istället användas för att undersöka kodgenereringsmöjligheter i Java.

I verktygen kommer endast s.k. klassdiagram (se kapitel 2.2.2) med UML notation att användas för evaluering av verktygens kodgenererings- och reverse engineerings-funktioner. Klassdiagram kommer att skapas för att generera kodskelett och kod kommer att användas för att skapa klassdiagram.

Undersökningen kommer inte att evaluera all funktionalitet i verktygen, och kommer i största utsträckning använda sig av CASE-verktygens standardinställningar och inbyggda funktioner. Då verktygen medger konfigurering m.h.a. guider, templates, plugins, programmering osv. kommer endast detta att utnyttjas i mån av tid och om det förbättrar resultaten vid kodgenereringen. Denna rapport kommer alltså inte att gå igenom alla inställnings- och konfigureringsmöjligheter som finns i verktygen.

Denna rapport begränsar sig även till att utvärdera två representativa CASE-verktyg. För att evaluera plattformsoberoenden kommer Sun Java 2 SDK, Standard Edition, version 1.3.0_02 (Sun, 2001a) på operativsystemet Microsoft Windows Me (Microsoft, 2001a) att användas. För fullständig beskrivning av konfiguration hänvisas till Appendix B.

I CASE-verktyget Visio går det endast att utföra reverse engineering från projekt skapade i Microsoft Visual J++ (Microsoft, 2001b), Microsoft Visual C++ (Microsoft, 2001c) eller Microsoft Visual Basic (Microsoft, 2001d). Då jag inte har tillgång till dessa program, och då det inte går att ladda ner evalueringsversioner av dessa verktyg från Internet, kommer ingen evaluering av reverse engineeringsmöjligheter i Visio att utföras.

3.3 Förväntat resultat

Rehbinders ramverk har framställts med en metod som extraherar krav som intressenter önskar att ett verktyg ska kunna utföra oavsett vad befintlig teknik klarar av eller ej (Rehbinder *et al*., 2000: 13). Detta indikerar att vissa krav troligtvis inte kommer att stödjas fullt ut i verktygen som undersöks i denna rapport.

Tidigare studier om CASE-verktyg har visat på förlorad semantik vid kodgenerering (Post & Kagan, 1998; Seifzadeh, 2000; m.fl.). Vid kodgenerering, reverse enginering och designtransformation tror författaren att detta även kommer att uppstå i denna undersökning.

Resultatet från denna rapport kan troligtvis användas för att jämföra CASE-experters förväntningar på CASE-verktyg och vad verktygen klarar av att utföra i nuläget. Det kan även ge insikt i vad ett modernt CASE-verktyg kan utföra i avseende på kodgenerering, samt vilka begränsningar de har.

4 Möjliga tillvägagångssätt

I detta kapitel beskrivs lämpliga metoder för att undersöka problemet, samt vilken metod som valts. Därefter beskrivs hur denna metod planeras att användas i undersökningen.

4.1 Metodalternativ

4.1.1 Litteraturstudie

En litteraturstudie innebär, enligt Patel och Davidson (1994: 33-34), att studera befintliga dokument, dvs. sådan information som är nedtryckt eller har tryckts på något vis. Denna information kan bl.a. hittas i facklitteratur, rapporter, tidningar, mm.

För denna undersökning kan en litteraturstudie användas för att hämta information om de CASE-verktyg som ska evalueras. Enligt Dawson (2000: 69) innebär det stora svårigheter att utföra ett projekt utan tillgång till relevanta tekniska manualer.

Vid genomförande av en litteraturstudie är det väldigt viktigt att ha ett kritiskt förhållningssätt till det material som studeras. För manualer är det viktigt att komma ihåg att de inte källgranskas, så som det görs med vetenskapliga artiklar, och ska därför inte användas som en grund till arbetet (Dawson, 2000: 69).

4.1.2 Observation

Med observationsmetoden studerar man beteenden och skeenden i ett naturligt sammanhang i samma stund som de inträffar (Patel & Davidson, 1994: 74). Blaxter *et al*. (1996:158) definierar observationsmetoden som ”…watching, recording and analysing events of interest”. I en observationsstudie, enligt Johnson (2001), analyseras användbarhet hos ett system i dess kontext där det används.

Observationer är enligt Patel och Davidson användbara vid experiment och tester, vilket passar för denna undersökning. Vad man måste tänka på vid observationer är att de måste planeras systematiskt. Informationen som fås vid en observation måste även registreras systematiskt (Patel & Davidson, 1994: 74; Blaxter *et al*., 1996: 158).

4.1.3 Testning

Enligt Lewis, 1991: 3929 innebär *testning*

> ”*…the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results*”.

Enligt Andersen (1994: 469-470) utför man tester på programvara för att se hur väl det överrensstämmer med användarnas förväntningar. Dessa förväntningar är subjektiva och kan ha olika utgångspunkter:

- Förväntningar på produkten.
- Behov (krav) på produkten.
- Krav från en kravspecifikation.

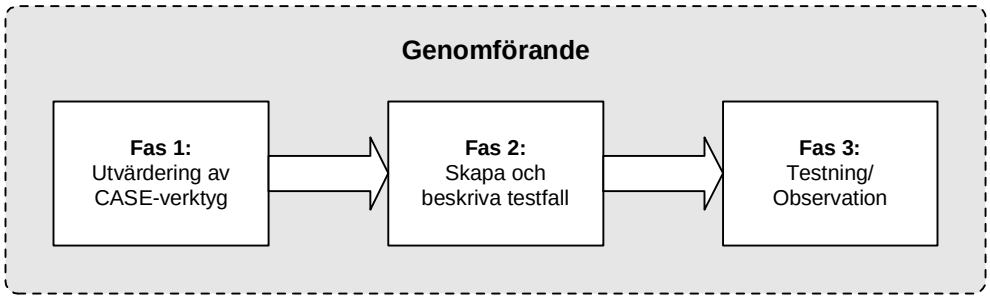
Denna undersökning utvärderar en delmängd av Rehbinders ramverk[5] som innehåller både krav och önskemål som har framställts i samarbete med CASE-experter, samt krav från organisationsspecifika dokument och kravspecifikationer. Detta innebär att alla tre utgångspunkter kommer implicit att testas.

4.2 Metodval

För att genomföra denna rapport kommer alla ovanstående metoder att utnyttjas. Manualer och dokumentation kommer att användas för att få kunskap om de CASE-verktyg som ska användas i undersökningen (TogetherSoft, 2000; TogetherSoft, 2001; Visio, 2000). Rapporten kommer dock inte att utföra en fullständig litteraturstudie på dessa dokument. Vidare kommer testning/observation av CASE-verktygen att utföras för att kunna besvara frågeställningen.

4.3 Plan för genomförande

Undersökningen kommer att utföras i sekventiella steg (faser), som beskrivs i detta kapitel (figur 4.1). I skapandet av denna plan har inspiration hämtats från ISO/IEC:s standard för evaluering och val av CASE-verktyg (ISO/IEC, 1995: 18).



Figur 4.1 *Plan för genomförande.*

Fas 1:

I den första fasen kommer olika CASE-verktyg att utvärderas. Utifrån uppställda kriterier kommer sedan två stycken att väljas och användas i följande faser. ISO/IEC:s standard (ISO/IEC, 1995: 16) hävdar att man bör välja CASE-verktyg utifrån de kriterier och krav som man har definierat. Utifrån denna rapports frågeställning får vi följande kriterier på CASE-verktyg.

[5] Rehbinder (2000).

Verktyget bör ge stöd för:

- skapandet av kodskelett från UML diagram.
- skapandet av verktygsoberoende kod.
- skapandet av plattformsoberoende kod.
- reverse engineering.

Följande kriterier kommer även att beaktas[6]:

- Företaget som utvecklar verktyget bör vara bland de marknadsledande inom sitt område och kontinuerligt stödja uppdateringar.
- Verktyget ska stödja UML, kodgenerering till Java, samt reverse engineering från källkod i Java.
- Tillgängligheten på verktyget (ska finnas en utvärderingsversion som kan laddas ner från webben).

Fas 2:

I denna fas skapas och beskrivs de testfall som ska genomföras i fas 3 samt hur de ska genomföras.

Fas 3:

Fas 3 testar testfallen på de utvalda CASE-verktygen och sammanställer resultaten.

[6] Vissa kriterier från Rehbinder (2000: Appendix C).

5 Genomförande

5.1 Utvärdering och val av CASE-verktyg

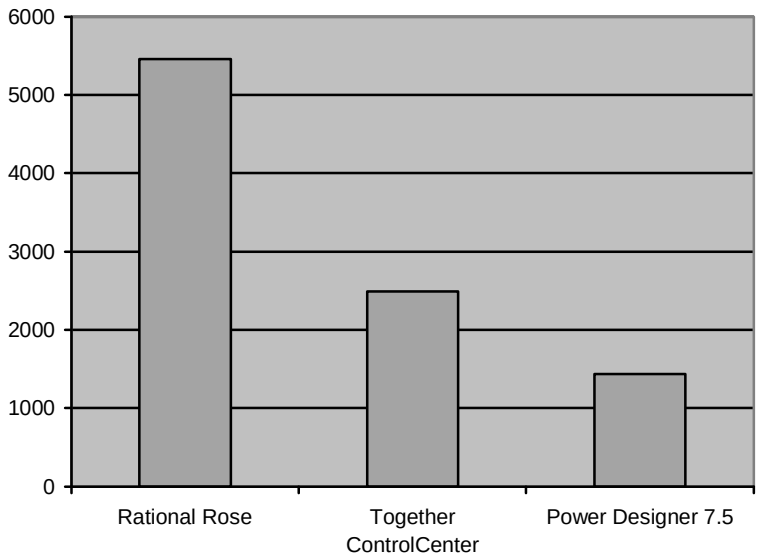
5.1.1 Möjliga CASE-verktyg

För att finna CASE-verktyg till denna undersökning använde jag mig av en omröstning, Readers Choise Award 2001, som utförs av Java Developer's Journal (JDJ, 2001a). Omröstningen utfördes mellan 15 januari till 31 maj, 2001 och var inte avslutad när denna rapport skrevs.

Enligt upphovsmännen är det Readers Choise Award "…the world's most widely participated industry award program" och man måste registrera sig för att få lägga sin röst (JDJ, 2001a). Vem som helst får registrera sig och rösta och man får bara rösta en gång. För att kontrollera vilka som röstar och undvika fusk, så måste man ange en korrekt e-post adress dit en bekräftelse skickas.

Den 22 april 2001 hade de tre högst placerade CASE-verktygen följande antal röster, av totalt 15 589 röster (JDJ, 2001b):

1. Rational Rose 2001 (5 462 röster)
2. Together ControlCenter (2 493 röster)
3. Power Designer 7.5 (1 434 röster)



Figur 5.1 *De tre högst placerade CASE-verktygen i Reader's Choise Award 2001*.

I Readers Choise Award 2000 intogs de tre översta placeringarna av följande CASE-verktyg (JDJ, 2000):

1. Rational Rose 2000 (3 285 röster)
2. Together/J (2 122 röster)
3. Power Designer 7 (1 180 röster)

5.1.2 Val av CASE-verktyg

Rehbinder (Rehbinder, 2000) samarbetade med organisationen Volvo IT (www.it.volvo.com) för att skapa det ramverk som kraven från detta arbete är hämtat ifrån. Från organisationens syn, var en av anledningarna för att skapa ramverket att utvärdera verktyget Visio 2000 Enterprise Edition (Rehbinder, 2000: 26; Seifzadeh, 2000: 13). Av denna anledning kommer ett av verktygen som används i undersökningen att vara Microsoft Visio 2000 Enterprise Edition (Appendix A).

För att välja ut ett andra verktyg att använda i undersökningen skapades en lista med information om Rational Rose och Together ControlCenter (Appendix A). Denna lista skapades med hjälp av riktlinjer från ISO/IEC:s standard för evaluering och val av CASE-verktyg (ISO/IEC, 1995: 16) och innehåller information[7] om verktygen som specificerats i kapitel 4.3.

Den största skillnaden på verktygen, enligt de uppsatta kriterierna i kapitel 4.3, var att man tillåts att evaluera en fullt funktionell demoversion av Together ControlCenter under en längre period än Rational Rose. Av denna anledning föll valet för det andra CASE-verktyget på Together ControlCenter.

5.2 Beskrivning av testfall

För att genomföra en systematisk undersökning måste dess observationer struktureras och registreras, antingen manuellt eller digitalt (Blaxter *et al.*, 1996: 158). Enligt ISO/IEC (1995: 18) bör nedanstående förberedelser genomföras innan en evaluering av CASE-verktyg:

- For each atomic subcharacteristic, define or select one or more metrics and define the details of their use.
- Set the rating levels and identify the means by which the levels will be generated or computed.
- Define the assessment characteristics for evaluation, establish what is acceptable, taking into consideration the rating levels previously defined and the context of use of the product.
- Identify and schedule all activities which must be performed as part of the evaluation process.

Då denna rapport utnyttjar krav från ett ramverk (Rehbinder, 2000) har vissa av stegen redan utförts. Denna undersökning kommer inte heller att använda sig av betygssystem i evalueringen, dock kommer ISO/IEC:s standard till viss del att utnyttjas för att analysera resultatet i denna rapport.

[7] Som är tagen från verktygens hemsida.

5.2.1 Kodgenerering

För att undersöka hur väl CASE-verktygen genererar kodskelett utifrån UML diagram kommer modeller från Shahin Seifzadehs rapport (Seifzadeh, 2000) att användas (Appendix D). Några av dessa modeller är hämtade från en modell skapad av Volo IT (Appendix D.7) vilket är intressant då kraven som evalueras i denna rapport kommer från nämnda organisation. För att evaluera relationstyper som Volvo ITs modell inte täcker har Seifzadeh även manuellt skapat två modeller (Seifzadeh, 2000: 22). Vad som kommer att undersökas i detalj är:

- Hur väl verktygen genererar kodskelett.
- Huruvida de genererade kodskelett är verktygs- och plattformsoberoende (se även kapitel 5.2.3).
- Vilka möjligheter man har att påverka resultatet m.h.a. inställningar, guider, mm.

Modellerna som används är tänkta att täcka grundläggande relationer som kan förekomma mellan klasser (Seifzadeh, 2000: 22). Totalt kommer sju testfall att skapas (testfall 1-7) med modeller som beskriver olika relationstyper. Modellerna har ändrats för att bättre stämma överens med Javas standard notation enligt Joy *et al*. (2000).

5.2.2 Reverse engineering

För att undersöka hur väl CASE-verktygen uppfyller kravet på reverse engineering kommer kod att skapas manuellt utifrån de utnyttjade modellerna i detta arbete (Appendix E). Denna kod kommer sedan att läsas in i verktyget och användas för att generera klassdiagram. Följande kommer att undersökas i detalj:

- Hur väl verktygen genererar klassdiagram från kod.
- Vilka möjligheter man har att påverka resultatet m.h.a. inställningar, guider, mm.

Koden är manuellt skapad m.h.a. verktyget UltraEdit version 8.00b (IDM, 2001) och har kompilerats med Suns Javakompilator, Sun Java 2 SDK, Standard Edition, version 1.3.0_02 (Sun, 2001a), samt kontrollerats med verktyget JavaPureCheck 4.1.1 (Sun, 2001b) för portabilitet.

Kod genererad från modeller i Visio (testfall 1-6) kommer även att utnyttjas. Genom att använda denna kod i Together får man en indikation på hur väl Visio genererar verktygsoberoende kod samt hur väl Together klarar av att hantera kod skapad av ett annat verktyg. Totalt har sex testfall (testfall 8-13) skapats för att evaluera CASE-verktygens förmåga till reverse engineering.

5.2.3 Oberoende kod

Vilket stöd verktygen ger för att skapa plattforms- och verktygsoberoende kod kommer att undersökas genom att evaluera hur väl koden överrensstämmer med standard Java (se även kapitel 5.2.2). I denna rapport definieras standard Java som

kod som följer Sun Microsystem, Inc. riktlinjer för ”100% pure Java” (Sun, 2001b). Kortfattat så innebär ”100% pure” Javakod (Meloan, 1997):

- no use of native method calls
- no external dependencies outside of the Java Core APIs

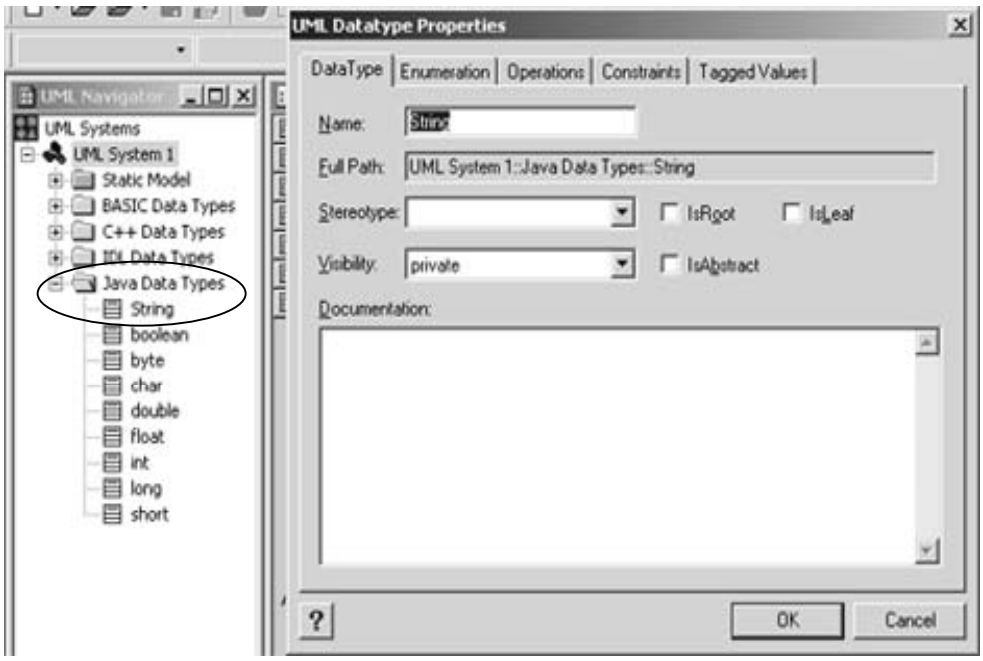
Dessa kriterier innebär att inga operativsystemspecifika anrop får utnyttjas och att koden inte får vara beroende av funktioner eller kod som inte ingår i Suns standard Java API.

För att kontrollera hur väl den genererade/skapade koden uppfyller Suns ”100% pure” kriterier kommer verktyget JavaPureCheck version 4.1.1 att användas (Sun, 2001b). Detta verktyg kontrollerar att Javakod är portabel och innebär alltså en kontroll huruvida koden är plattforms- och verktygsoberoende. Genererad kod kommer även att exekveras med Sun Java 2 SDK, Standard Edition, version 1.3.0_02 (Sun, 2001a) för att ytterligare kontrollera portabilitet.

5.3 Testning

5.3.1 Kodgenerering

Modellerna i nedanstående testfall har skapats manuellt i respektive CASE-verktyg med verktygens modelleringsfunktioner (Appendix D). Därefter har kod genererats från CASE-verktygen utifrån de skapade modellerna (Appendix D). Visio har ställts in för att generera kod för Java (det går att välja mellan C++, Java och Visual Basic, där C++ är standardinställning). I Visio har även en datatyp, String, skapats då den inte ingick i Visios standarddatatyper. Detta har gjorts genom att lägga till en ny UML Datatyp i Java Data Types katalogen (figur 5.2)



Figur 5.2 *Skapande av datatypen String i Visio.*

Visio skapar automatiskt metoder för att ändra och hämta variablers värden (s.k. get- och setmetoder). Denna funktion stängdes av för att förenkla analysen av de skapade kodskeletten samt förbättra läsbarheten. Skillnaden illustreras i figur 5.3. I Together kan man välja mellan att skapa variabler av typen *attribute* eller *property* (en klass kan innehålla båda typerna). Skillnaden mellan dessa är att variabler av typen *property*, i motsats till variabler av typen *attribute*, automatiskt skapar metoder för att ändra och hämta variablers värde (figur 5.4).

Med metoder

```
public class Class
{
   public Class()
   {
      super();
   }
   public final int getAttribute()
   {
      return mattribute;
   }
   public final void setAttribute(int the_mattribute)
   {
      this.mattribute = the_mattribute;
   }

   private int mattribute;
}
```

Utan metoder

```
public class Class
{
   private int mattribute;
}
```

Figur 5.3 *Skillnaden mellan att utnyttja Visios automatiska generering av metoder till attribut eller ej.*

Med attribute variabel

```
public class Class {
   private int attribute;
}
```

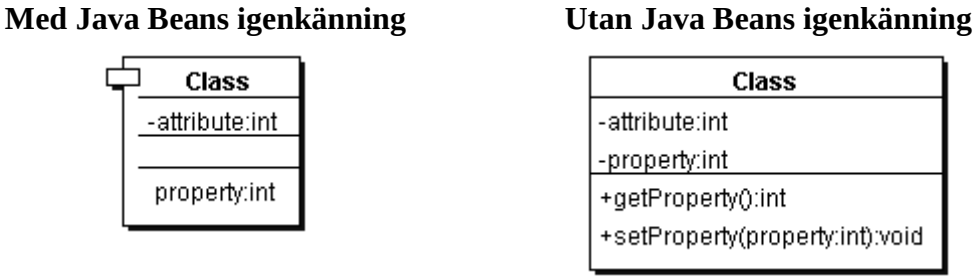
Med property variabel

```
public class Class {
   public int getProperty() {
      return property;
   }

   public void setProperty(int property) {
      this.property = property;
   }

   private int property;
}
```

Figur 5.4 *Skillnaden mellan att utnyttja attribute eller property variabler i Together.*

I Together ControlCenter har variabler av typen *attribute* används för att förenkla analysen samt se till att den genererade koden har samma förutsättningar som kod genererad i Visio. Together har en egenhet att se klasser som innehåller get- och setmetoder som Java Beans vilket påverkar den grafiska framställningen av modellerna (TogetherSoft, 2001). Denna skillnad, dvs. hur modeller illustreras i Together med igenkänning av Java Beans eller ej, kan ses i figur 5.5. I övrigt användes CASE-verktygens standardinställningar.

Båda verktygen innehåller funktioner och inställningar för att påverka hur den genererade koden ska se ut. Dock innehåller Together ControlCenter fler inställnings-möjligheter än Visio för hur kod ska formateras och genereras. I Togethers *templates* går det att uttrycka fler alternativa sätt hur koden ska genereras än i Visio samt skapa egna s.k. *patterns* (TogetherSoft, 2001). Genomgång av dessa inställningar ligger dock utanför denna rapports omfattning.



Figur 5.5 *Skillnad mellan framställning av modeller i Together.*

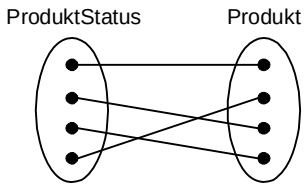
Generering av kod i Visio måste utföras manuellt och varje gång man ändrar i modellen måste ny kod skapas. I verktygets standarduppsättning indikeras ej om kod och modell är synkroniserade eller ej. I motsats stödjer Together ControlCenter round-trip engineering, kod uppdateras automatiskt och i realtid när modellen ändras och vice versa.

Inget av verktygen stödjer i dess standarduppsättning versionshantering, men stödjer användning av externa versionshanteringssystem som kan integreras i verktygen (Visio, 2000; TogetherSoft, 2000). I Visio kan detta vara ett sätt att säkerställa synkronisering mellan kod och modell, dock har externa versionshanteringssystem inte evaluerats eller använts i denna undersökning.

5.3.1.1 Testfall 1

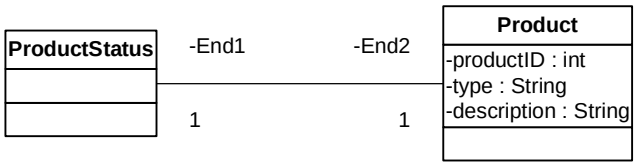
I detta testfall beskrivs ett ett-till-ett förhållande mellan två klasser. Relationen innebär att en instans av klassen `Produkt` känner till ett och endast ett `ProduktStatus` objekt (Seifzadeh, 2000:23). Dessutom uttrycks att en instans av

`ProduktStatus` känner till ett och endast ett `Produkt` objekt (Elmasri & Navathe, 1994: 52). Denna relation illustreras i figur 5.6[8].

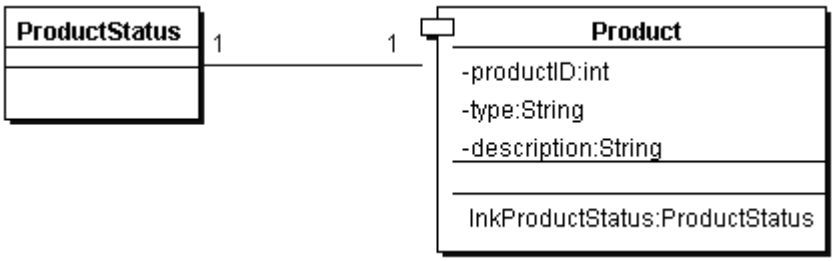


Figur 5.6 *Illustration av relationsförhållandet mellan klasserna i testfall 1.*

Enligt Elmasri & Navathe (1994: 698) kan denna modell implementeras i kod genom att införa en medlemsvariabel (relationsvariabel) i en av klasserna. Denna variabel ska refererar till en instans av den andra klassen. I detta testfall har modellen i de båda CASE-verktygen ritats så att relationsvariabeln hamnar i klassen `Product`, dvs. en riktad relation från klassen `Product` till klassen `ProductStatus`. Figur 5.7 visar den använda modellen i Visio och figur 5.8 visar modellen i Together.



Figur 5.7 *Ett-till-ett relation i Visio.*



Figur 5.8 *Ett-till-ett relation i Together.*

I Visio fick relationsslutpunkten `End1` ändras till en ”navigable” (figur 5.9). Detta för att ”tvinga” verktyget att, i klassen `Product`, generera en medlemsvariabel som refererar till en instans av klassen `ProduktStatus`. Enligt Visios hjälpfil innebär detta val följande:

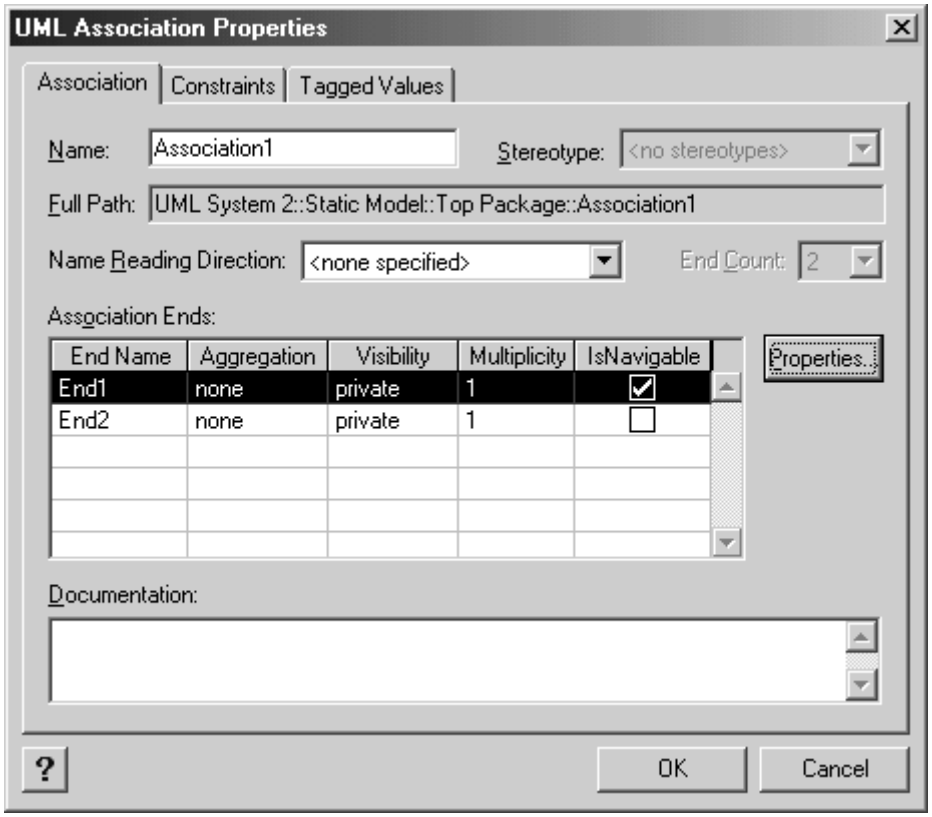
> ”Check [isNavigable] to indicate that navigation is supported toward the target instance”.

[8] Se Appendix C samt Elmasri & Navathe (1994) för genomgång av denna typ av notation.

I Together fick man manuellt ställa in att get- och setmetoder skulle skapas för att kunna få åtkomst samt ändra medlemsvariabeln `lnkProductStatus`, detta var inställt som default i Visio. Dessa metoder finnas för att `ProductStatus` och andra klasser ska få åtkomst till relationsvariabeln `lnkProductStatus` i klassen `Product`.

För att göra så att Together ControlCenter skapade ovanstående beskrivna metoder används ett template (kallas även *pattern* i Together) på relationsvariabeln. I praktiken innebär detta att man får upp en dialogruta där man får kryssa i om man vill skapa en get- eller setmetod för vald variabel. Det finns ett antal olika fördefinierade patterns och det hade även gått att skapa ett eget relationspattern som automatiskt skapar get- och setmetoder när man skapar en relation mellan två klasser.

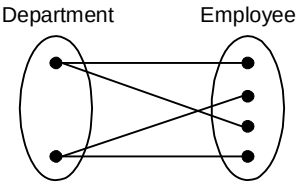
Båda verktygen lyckades dock att generera kodskelett som stämmer överens med modellen, dessa kodskelett redovisas i Appendix D.1.



Figur 5.9 *Inställning för relationen mellan klasserna ProductStatus och Product.*

5.3.1.2 Testfall 2

I detta testfall beskrivs ett ett-till-många förhållande mellan två klasser. Modellerna som användes illustreras i figur 5.12 och figur 5.13. Relationen innebär att en instans av klassen `Department` känner till flera `Employee` objekt, samt att en instans av klassen `Employee` känner till ett och endast ett `Department` objekt (figur 5.10).



Figur 5.10 *Illustration av relationsförhållandet mellan klasserna i testfall 2.*

Relationen är skapad som en riktad relation från klassen `Department` till klassen `Employee`. Enligt Elmasri & Navathe (1994: 698) går detta att lösa genom att skapa en relationsvariabel i klassen `Department` som kan innehålla en eller flera referenser till instanser av klassen `Employee`. I Java går det att implementera detta på flera olika sätt (Skansholm, 2000: 210-214, 235-242; Eckel, 2000: 407-409), tre av lösningarna är att:

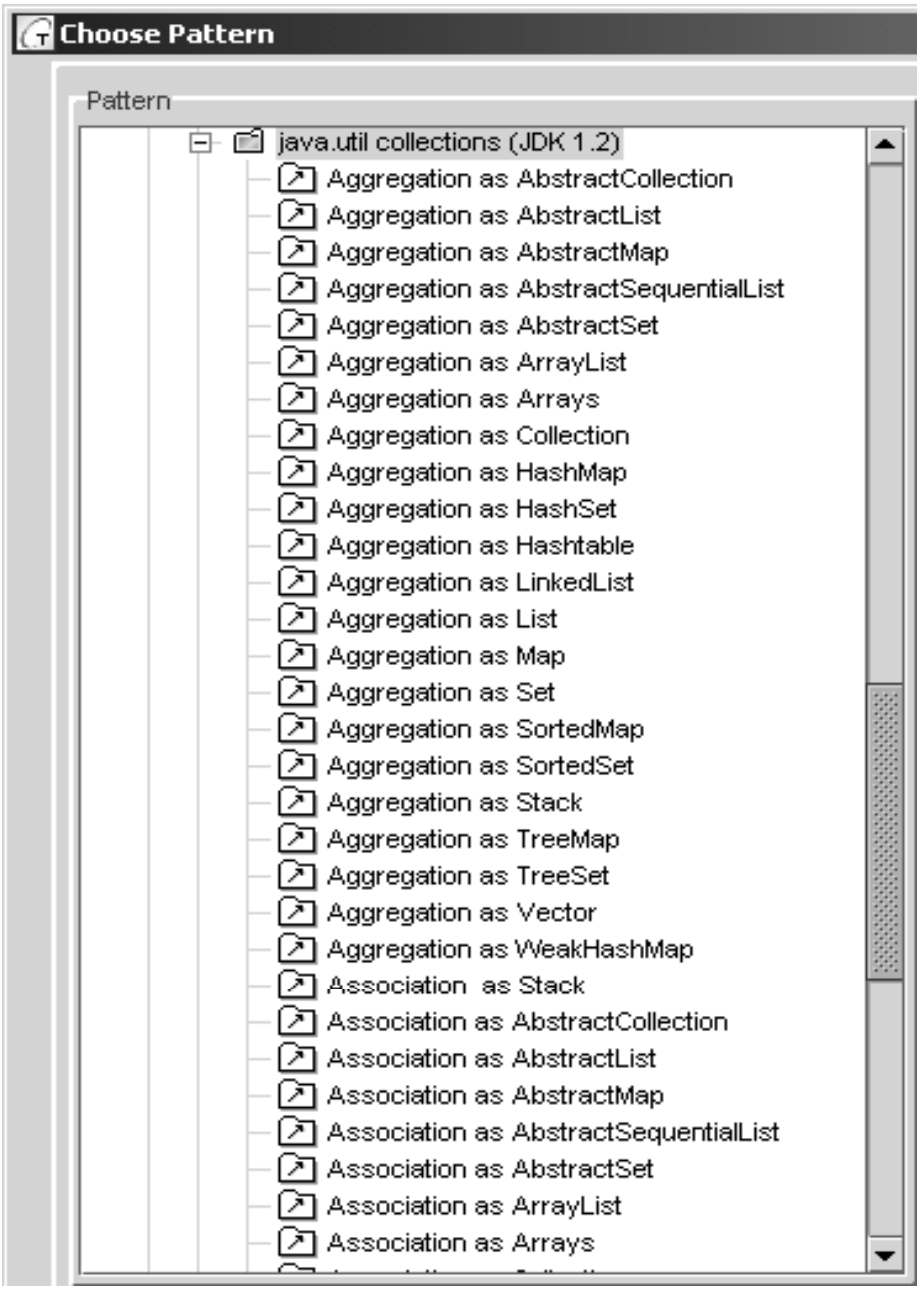
- Skapa en `Employee` array i klassen `Department`.
- Skapa en relationsvariabel av typen `Vector` i klassen `Department` som kan referera till en eller flera `Employee` objekt.
- Skapa en relationsvariabel av typen `Collection` i klassen `Department` som refererar till en eller flera `Employee` objekt

Den första lösningen är den mest effektiva lösningen (i form av snabbhet) för att lagra och hämta en sekvens av objekt (Eckel, 2000, 408). En nackdel med arrayer är att man måste specificera dess storleken vid implementationen, dvs. att den har en fast storlek (Eckel, 2000: 408). De två andra lösningarna har valt beroende på att de stöds i de båda CASE-verktygen, dock medför de en försämrad abstraktionsnivå samt är inte lika effektiva som arrayer (Eckel, 2000: 408, 450).

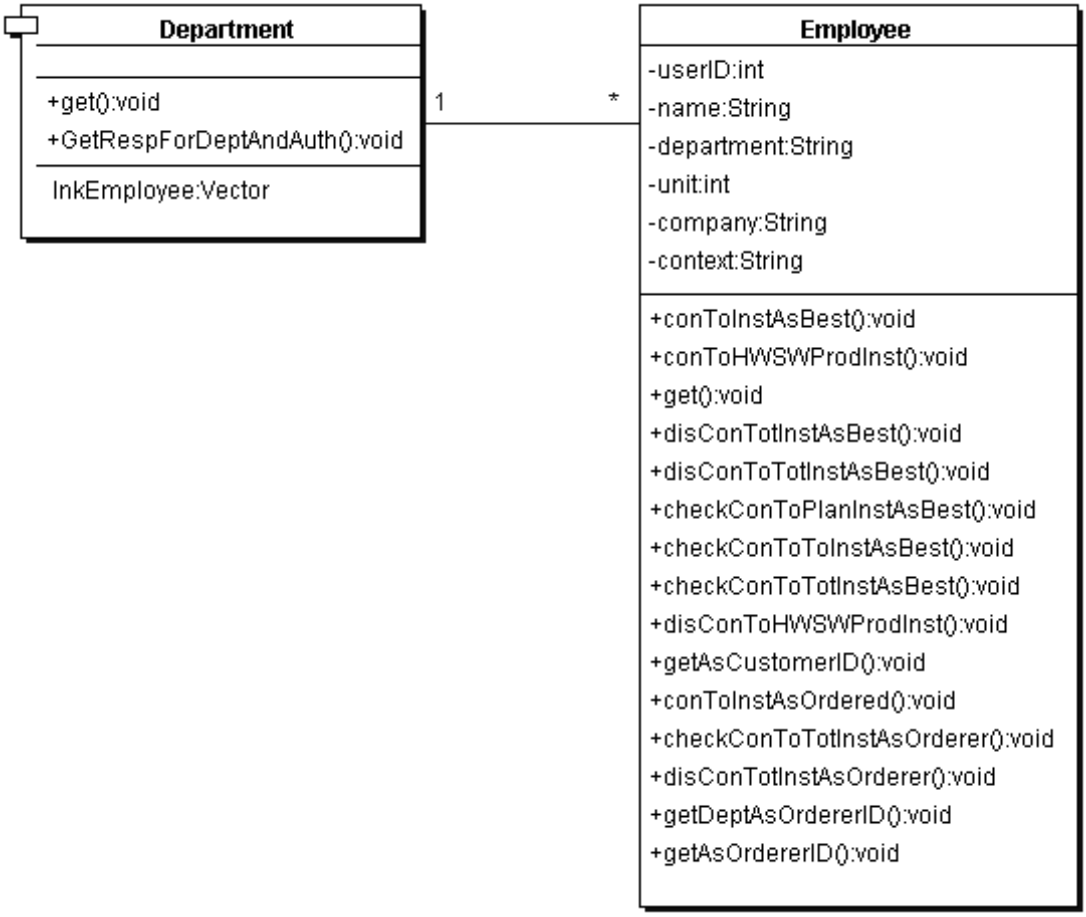
I Visio skapas en `Vector` för denna typ av relation automatiskt, det går dock endast att välja mellan att skapa en `Vector` eller en `Collection` i programmets inställningar. I Together går det m.h.a. patterns att välja mellan totalt 123 stycken olika datastrukturer (figur 5.11) från dels Java Development Kit 1.1 (Sun, 2001c), Java 2 Standard Development Kit (Sun, 2001a) samt JGL (Java Generic Library) 3.1 (ObjectSpace, 2001).

Några av dessa datastrukturer är dock dubbletter då de är de samma i Java Development Kit 1.1 och Java 2 Standard Development Kit men tillhör olika paket beroende på vilken version av kompilator man använder. Exempelvis så finns datatypen `Vector` i paketet `java.util` i Java 2 Standard Developent Kit och i paketet `com.sun.java.util.collections` i Java Development Kit 1.1.

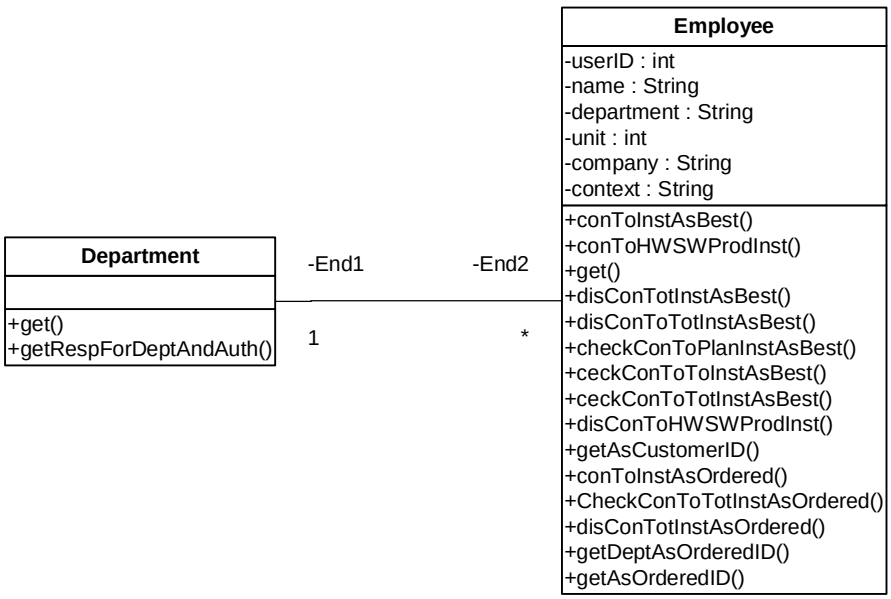
I modellen som skapats i Visio fick relationsslutpunkten `End2` ändras till en ”navigable” för att generera relationsvariabeln. I Together skapades associationen med en `Vector` för att efterlikna Visios inställningar. Vidare utnyttjades design pattern i Together för att skapa get- och setmetoder åt relationsvariabeln.



Figur 5.11 *Relationspattern i Together ControlCenter 4.2.*



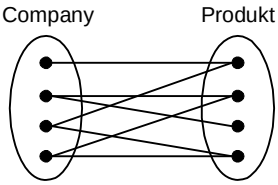
Figur 5.12 *Ett-till-många relation i Together.*



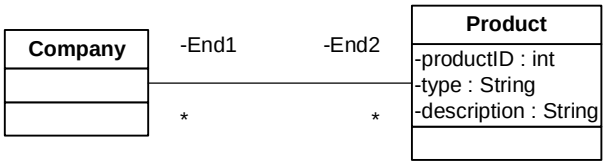
Figur 5.13 *Ett-till-många relation i Visio.*

5.3.1.3 Testfall 3

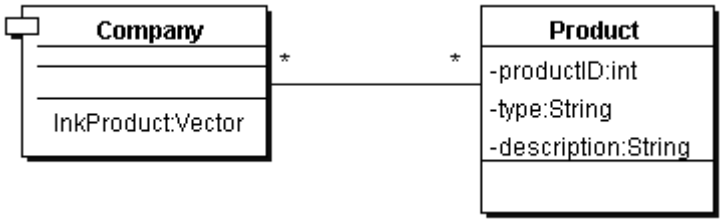
Detta testfall används för att beskriva ett många-till-många förhållande mellan två klasser. Modellerna som används illustreras i figur 5.15 och 5.16. Relationen innebär att en instans av klassen `Company` känner till en eller flera `Product` objekt och en instans av klassen `Product` känner till en eller flera `Company` objekt (figur 5.14).



Figur 5.14 *Illustration av relationsförhållandet mellan klasserna i testfall 3.*



Figur 5.15 *Många-till-många relation i Visio.*



Figur 5.16 *Många-till-många relation i Together*.

Att implementera denna relation innebär samma sak som i testfall 2, med den skillnaden att en relationsvariabel skapas i båda klasserna (Elmasri & Navathe, 1994: 698). Dessa relationsvariabler ska kunna referera till ett eller flera objekt av den refererade klassen. Detta innebär att referensvariabel i klassen `Company` ska kunna referera till en eller flera objekt av klassen `Produkt` och vice versa.

För att skapa relationsvariabler i Visio fick båda relationsslutpunkterna, `End1` och `End2`, i figur 5.15 ändras till att vara ”navigable”. Den genererade koden från CASE-verktygen redovisas i Appendix D.3.

5.3.1.4 Testfall 4

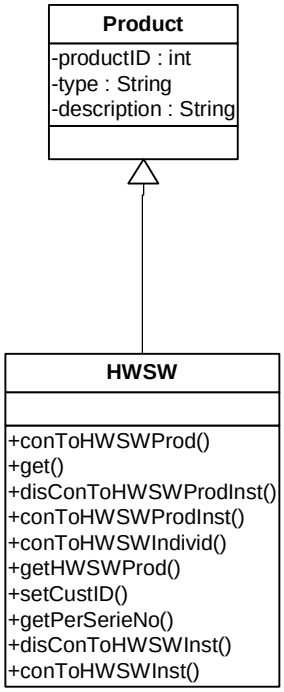
Modellerna i figur 5.18 och figur 5.19 används för att beskriva en arvsrelation mellan två klasser. I Java innebär ett arv mellan två klasser att subklassen (i detta fall klassen `HWSW`) ärver publika metoder och attribut från dess superklass (i detta fall klassen

`Product`) (Eckel, 2000: 38-44). För att arvsrelationen ska vara valid i den genererade koden, enligt denna rapports definition av standard Java, krävs det att CASE-verktygen genererar kod som definierar att klassen `HWSW` *extends* klassen `Product` enligt figur 5.17 (Eckel, 2000: 275).

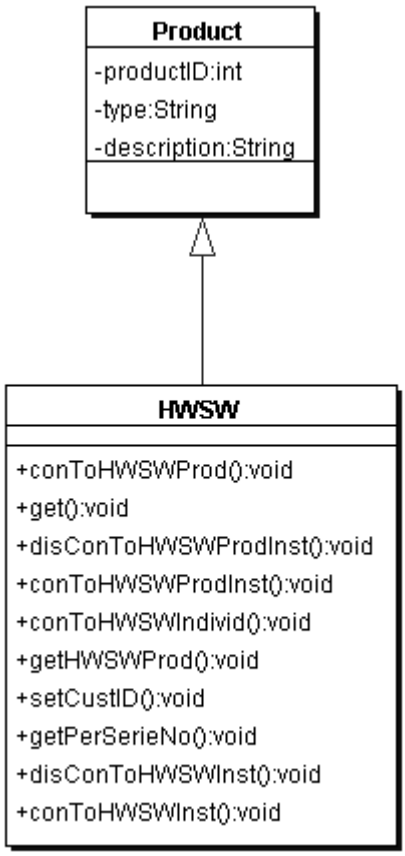
```
public class HWSW extends Product {
   :
```

Figur 5.17 *Kodexempel på valid arvsrelation mellan klassen Product och HWSW.*

De genererade kodskelett som CASE-verktygen producerade återfinns i Appendix D.4.



Figur 5.18 *Arvsrelation i Visio*.

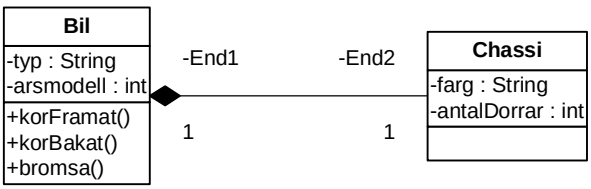


Figur 5.19 *Arvsrelation i Together*.

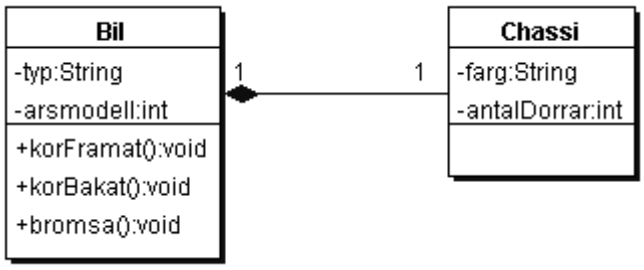
5.3.1.5 Testfall 5

I figur 5.20 och figur 5.21 illustreras modellerna som användes för att beskriva ett aggregatförhållande i kompositionsform mellan två klasser. Denna relationstyp kallas även komposition (Eckel, 2000: 271). Relationen innebär att en klass är en del av en annan klass. Detta kan beskrivas som att en klass ”består av” eller ”har en/ett” en annan klass, i detta fall kan relationen utläsas som att *en bil består av ett chassi*, eller att *ett chassi är en del av en bil* (Eckel, 2000: 37).

Enligt Eckel (2000: 271) kallas denna typ av relation för en komposition eftersom man utnyttjar existerande klasser, dvs. att en klass är en komposition av redan existerande klasser.



Figur 5.20 *Aggregatrelation i kompositionsform i Visio.*



Figur 5.21 *Aggregatrelation i kompositionsform i Together.*

För att skapa ett aggregatförhållande i kompositionsformat utnyttjades relationstypen *composition* som fanns som standard i båda CASE-verktygen. För att skapa en relationsvariabel i Visio måste man specificera att relationsslutpunkten `End2` är ”navigable”.

Ett exempel på en korrekt implementering av modellen är att när en instans av klassen `Bil` skapas så skapas även ett `Chassi` objekt som tillhör det nyss skapade `Bil` objektet. När sedan `Bil` objektet tas bort så kommer även `Chassi` objektet som det borttagna `Bil` objeketet refererade till, att tas bort (Eckel, 2000: 271-275). I Java löses detta genom att skapa en relationsvariabel av typen `Chassi` i klassen `Bil`, samt skapa ett `Chassi` objekt som denna relationsvariabel refererar till (figur 5.22).

```
:
private Chassi relation = new Chassi();
:
```

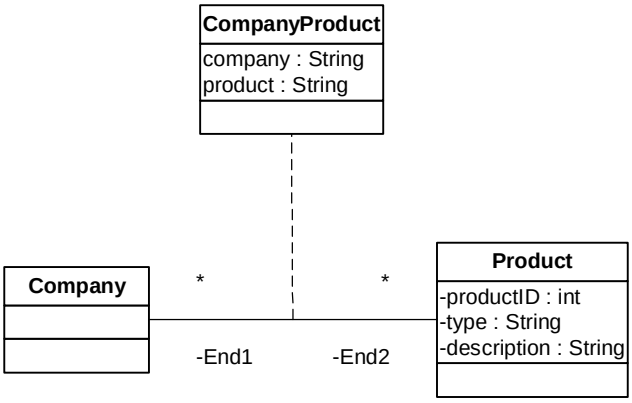
Figur 5.22 *Implementation av ett aggregatförhållande i kompositionsformat i Java enligt testfall 5.*

De kodskelett som verktygen genererade finns i Appendix D.5.

5.3.1.6 Testfall 6

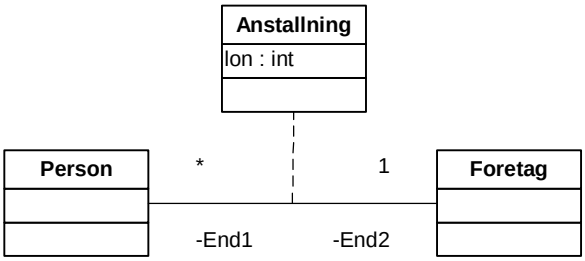
Detta testfall används för att beskriva en dubbelriktad association mellan två klasser, samt en tillhörande associationsklass till relationen (figur 5.23 och figur 5.24). En associationsklass används för att modellera en association som en klass och används då en association mellan två klasser själv har attribut (jGuru, 2000).

Ett exempel på en associationsklass illustreras i figur 5.25. Denna modell beskriver en ett-till-många association mellan klasserna `Person` och `Foretag` med associations-klassen `Anstallning` (Denzinger, 2001). Denna relation kan ha ett attribut, exempelvis `lon`. I detta fall går det att använda sig av en associationsklass med detta attribut istället för att skapa attributet i någon av klasserna `Person` eller `Företag` (jGuru, 2000).



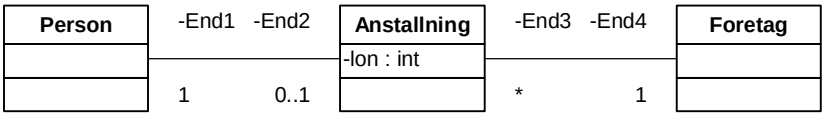
Figur 5.23 *Dubbelriktad association med en tillhörande associationsklass i Visio*.

Figur 5.24 *Dubbelriktad association med en tillhörande associationsklass i Visio*.



Figur 5.25 *Exempel på en associationsklass.*

Enligt Denzinger (2001) går det att beskriva ovanstående modell (figur 5.25) utan en associationsklass. Denna modell (figur 5.26) kommer dock inte att undersökas i detta testfall. För att undersöka hur Together klarar av att generera kod som beskriver denna typ av relation kommer figur 5.26 att utnyttjas som en referens (se kapitel 5.3.2.6).



Figur 5.26 *Exempel på ekvivalent representation av figur 5.25*.

För att implementera associationsklassen i kod innebär följande steg (Seifzadeh, 2000: 32). Jämför även med figur 5.26.

- Skapa en relationsvariabel i klassen `CompanyProduct` som refererar till ett en instans av klassen `Company`.
- Skapa en relationsvariabel i klassen `CompanyProduct` som refererar till en instans av klassen `Product`.
- Skapa en relationsvariabel i klassen `Company` som kan referera till en eller många instans(er) av klassen `CompanyProduct`.
- Skapa en relationsvariabel i klassen `Product` som kan referera till en eller många instans(er) av klassen `CompanyProduct`.

I figur 5.27 illustreras kodskelett som har implementerat ovanstående kriterier. Detta kodexempel visar endast relationsvariabler och implementerar inte attribut eller metoder i klasserna. Den genererade koden från CASE-verktygen återfinns i Appendix D.6.

```
class Company {
   private CompanyProduct[] lnk;
}

class Product {
   private CompanyProduct[] lnk;
}

class CompanyProduct {
   private Company lnkComp;
   private Product lnkProd;
}
```

Figur 5.27 *Kodexempel som beskriver relationerna från modellen i testfall 6.*

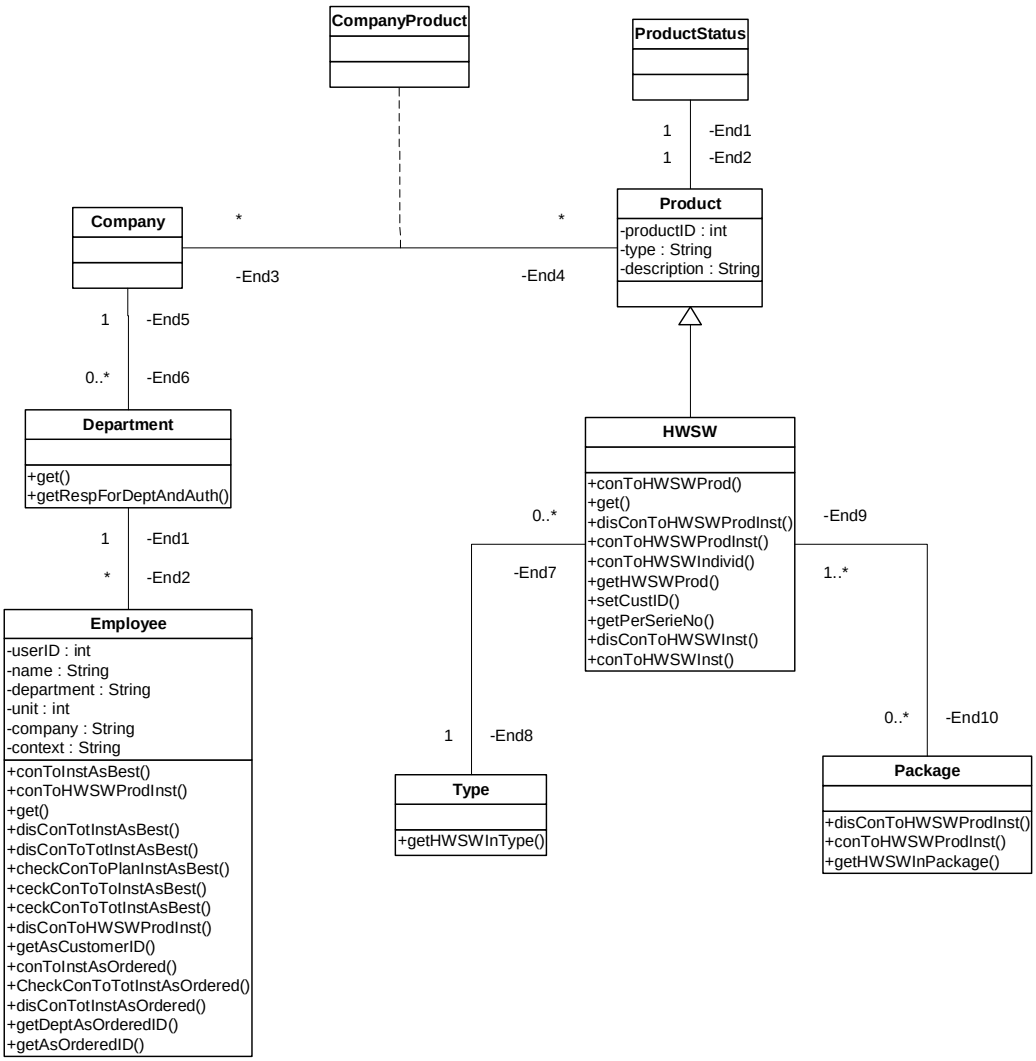
5.3.1.7 Testfall 7

I detta testfall undersöks hur CASE-verktygen genererar kod utifrån den fullständiga modellen som skapats av Volvo IT (figur 5.28 och figur 5.30). Intressant är att undersöka hur verktygen klarar av att hantera en lite större modell som innehåller fler än en relation. Då tidigare modeller (testfall 1-6) undersöker delmängder av denna modell kommer följande klasser att undersökas mer detaljerat:

- `Product`
- `Company`
- `HWSW`

Detta urval har gjorts beroende på att i denna modell måste dessa klasser innehålla mer information än i föregående testfall. Klassen `Department` kommer inte att innehålla fler relationsvariabler då relationen mellan `Company` och `Department` endast genererar en relationsvariabel i klassen `Company`. Relationerna mellan `HWSW` och `Type`, samt mellan `HWSW` och `Package` är en ett-till-många relation respektive en många-till-många relation. Då dessa typer av relationer redan har tagits upp i tidigare

testfall kommer de inte heller att analyseras mer ingående. Dock redovisas all genererad kod i Appendix D.7.



Figur 5.28 *Modell från Volvo IT i Visio.*

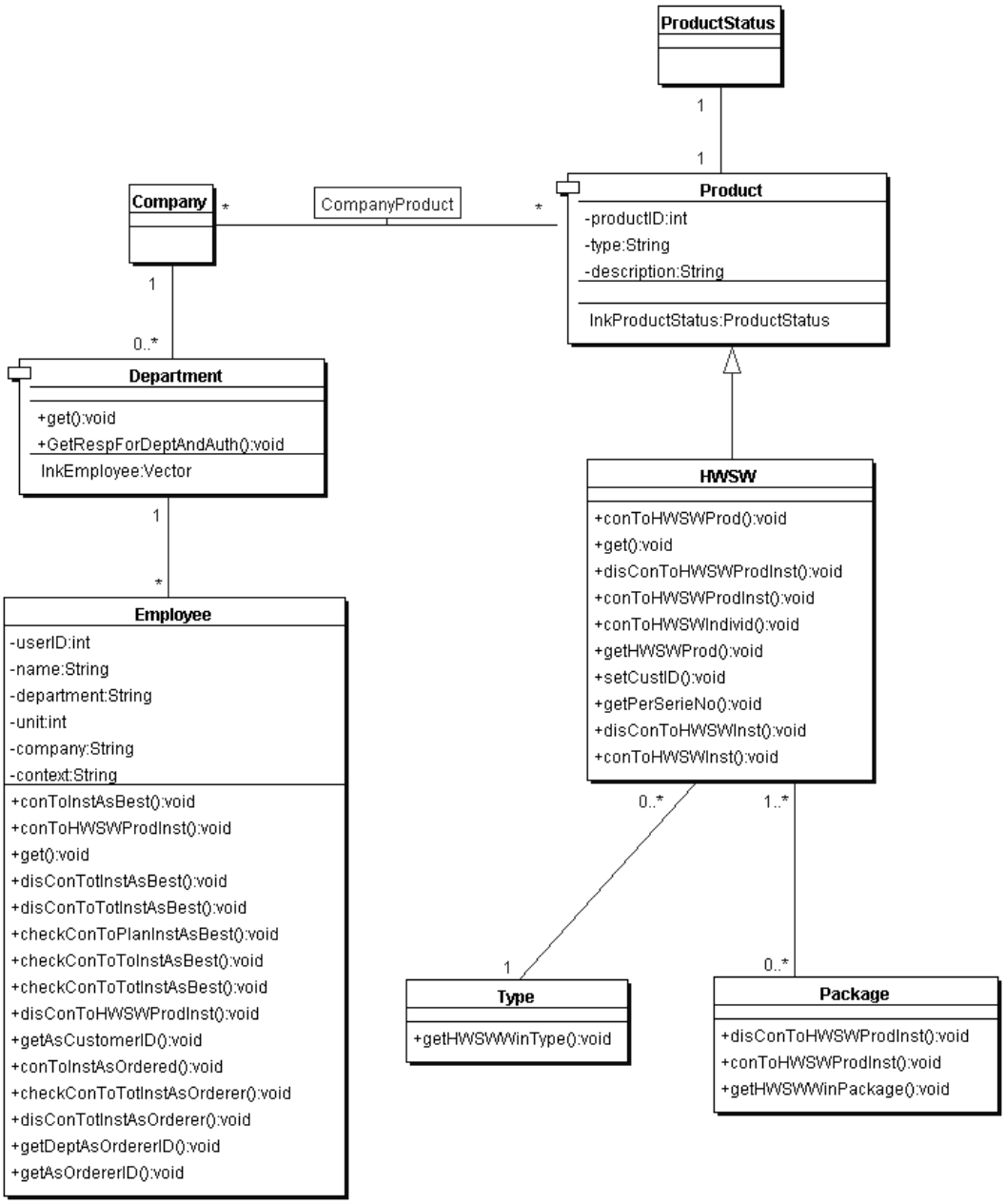
Principer som gåtts igenom i tidigare testfall kommer att utnyttjas för att utvärdera de genererade kodskeletten. Klassen `Product` har i denna modell tre relationer, dessa relationer är beskrivna i mer detalj i testfall 1, 4 och 6 (kapitel 5.3.1.1, 5.3.1.4 och 5.3.1.6). Utifrån dessa testfall ska klassen `Product` innehålla följande relationsvariabler (figur 5.30):

- En relationsvariabel som kan referera till en instans av klassen `ProductStatus`.
- En relationsvariabel som kan referera till en eller många instanser av klassen `CompanyProduct`.

Ett exempel på en implementation av ovanstående relationer illustreras i figur 5.29.

```
class Product {
   private ProductStatus lnk1;
   private CompanyProduct[] lnk2;
}
```

Figur 5.29 *Relationsvariabler i klassen Product.*



Figur 5.30 *Modell från Volvo IT i Together.*

Till klassen `Company` finns det en ett-till-många relation från klassen `Department` samt en dubbelriktad association till klassen `Product` med en tillhörande

associationsklass. En möjlig implementation beskrivs i figur 5.31, och har utnyttjat testfall 1 och 6 för att definiera följande kriterier:

- En relationsvariabel som kan referera till en eller många instans(er) av klassen `Department`.
- En relationsvariabel som kan referera till en eller många instans(er) av klassen `CompanyProduct`.

```
class Company {
   private Department[] lnk1;
   private CompanyProduct[] lnk2;
}
```

Figur 5.31 *Relationsvariabler i klassen Company.*

Utifrån tidigare testfall får man att klassen `HWSW` ska innehålla följande för att kunna anses valid (figur 5.32):

- En relationsvariabel som kan referera till en eller många instans(er) av klassen `Package`.
- Ärva ifrån klassen `Product`.

```
class HWSW extends Product {
   private Package[] lnk;
}
```

Figur 5.31 *Relationsvariabler i klassen Company.*

5.3.2 Reverse engineering

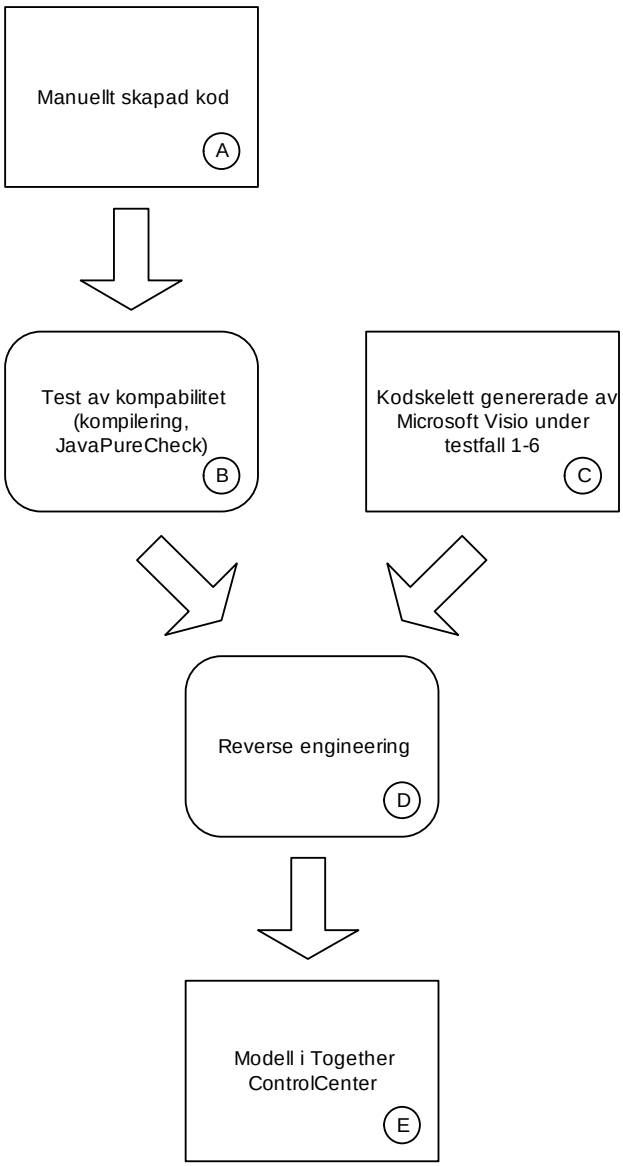
För att evaluera Togethers förmåga till reverse engineering har kod skapats manuellt i verktyget UltraEdit (IDM, 2001) (figur 5.32 – A). De kodexempel som används är översatta från Seifzadehs rapport (Seifzadeh, 2000) som är skrivna i C++ och utgår ifrån modellerna i testfall 1-6. Koden har kompilerats och exekverats i Sun Java 2 Platform, Standard Edition, version 1.3.0_02 (Sun, 2001a) samt kontrollerats med verktyget JavaPureCheck (Sun, 2001b) för att garantera kompabilitet med standard Java enligt denna rapports definition (figur 5.32 – B).

Koden har sedan använts för att skapa modeller i Together ControlCenter 4.3. För att utföra en reverse engineering i Together skapar man helt enkelt ett nytt projekt i samma katalog som källkodsfilerna finns i. Det går även att manuellt kopiera källkodsfilerna till katalogen där projektet skapats, då Together automatiskt utför en reverse engineering på dessa filer (figur 5.32 – D).

Vidare har de kodskelett som genererats i Visio under testfall 1-6 utnyttjats för att undersöka hur väl Visio genererar verktygsoberoende kod (figur 5.32 – C). Denna process ger även en antydan av hur väl Together klarar av att hantera kod skapad av

ett annat verktyg. De genererade modellerna i Together har även jämförts med de ursprungliga modellerna från Visio för att finna eventuella semantiska förluster.

Kodskelett som genererades av Visio under testfall 1-6 innehåller en rad i koden som specificerar att klassen tillhör paketet `Top_Package`, dvs. följande sats finns överst i varje kodskelett; `package Top_Package;`

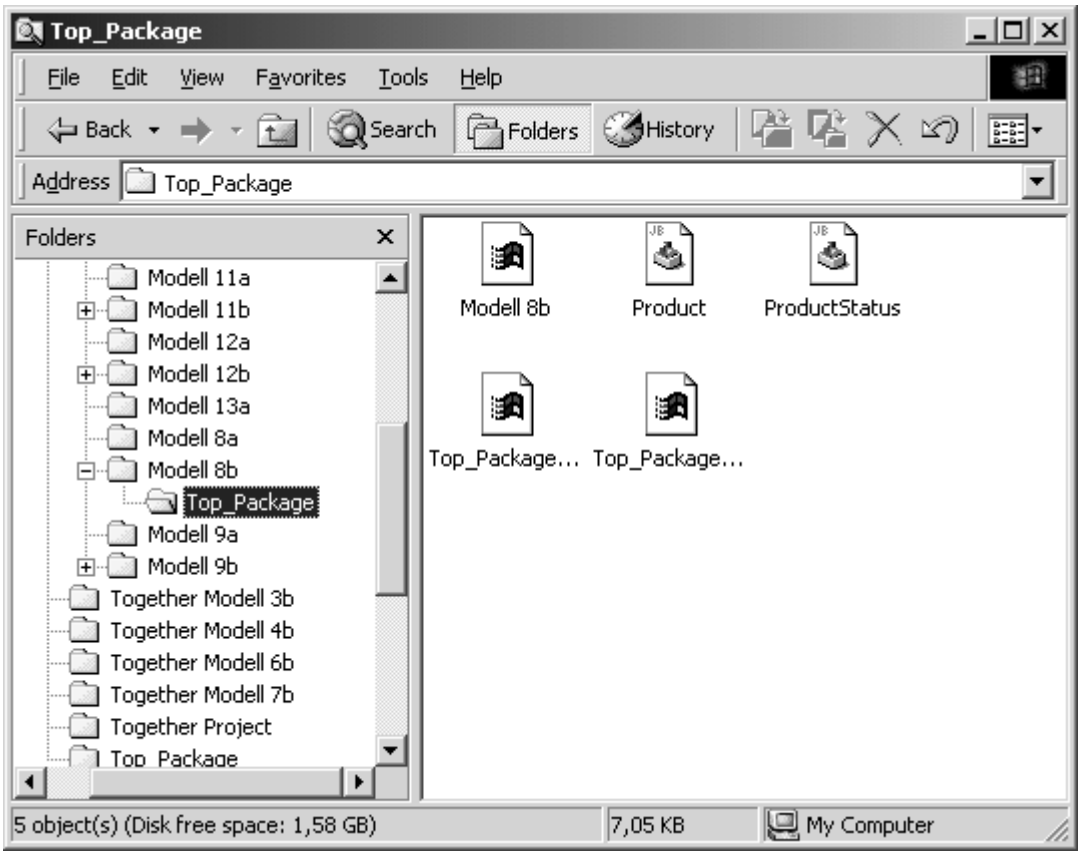


Figur 5.32 *Figur över utförda steg i reverse engineeringsevalueringen.*

Enligt Java ska källkodsfiler som tillhör ett visst paket lagras i en katalog med samma namn som paketsnamnet (Eckel, 2000: 247-250). Together klarar ej av att skapa denna katalog automatiskt vid reverse engineering av kod, vilket innebar att en katalog med namnet `Top_Package` fick skapas i projektkatalogen i Together (figur 5.33). Det går att skapa ett paket på två olika sätt i Together:

- Manuellt genom att skapa en katalog i projektkatalogen (figur 5.33).
- Skapa (rita) ett `pakage` med Togethers inbyggda modelleringsfunktioner.

I denna undersökning gjordes denna procedur genom att manuellt skapa en katalog, detta för att i största möjliga mån låta Together hantera reverse engineerings-processen.



Figur 5.33 *En katalog med namnet Top_Package fick skapas för projekt som utnyttjade kodskelett genererade i Visio i reverse engineeringsprocessen. Denna figur illustrerar att en katalog med namnet Top_Package har skapats för projektet med namnet Modell 8b.*

Modellerna som verktyget genererat har därefter granskats för att se om semantik har förlorats under transformationen, dvs. huruvida de stämmer överens med koden som använts vid reverse engineeringsprocessen.

Genererade kodskelett ifrån Together har dock inte utnyttjats. Detta beroende på att Together stödjer round-trip engineering och använder källkoden för att rita upp modeller. Modellerna är alltså automatiskt synkroniserade med källkoden (Together, 2000).

5.3.2.1 Testfall 8

Detta testfall beskriver en ett-till-ett relation mellan klasserna `Product` och `ProductStatus`. Egenproducerad kod som används för att generera modeller

återfinns i Appendix E.1. Kodskelett genererad av Visio i testfall 1 (Appendix D.1) har även använts för att skapa modeller. Koden är uppbyggd så att en instans av klassen `Product` känner till ett `ProductStatus` objekt.

En korrekt modell utifrån källkod ska innehålla klasserna `Product` och `ProductStatus`. Mellan klasserna ska det finnas en associationsrelation med kardinaliteten 1 på båda ändarna av relationen (se exempelvis modeller i testfall 1).

5.3.2.2 Testfall 9

I Appendix E.2 återfinns den egenproducerade kod som har använts i detta testfall. Från testfall 2 har även genererad kod från Visio utnyttjats (Appendix D.2). Koden beskriver en ett-till-många relation mellan två klasser, `Department` och `Employee`. Som tidigare har beskrivit i rapporten går relationen att implementeras i Java genom att skapa en relationsvariabel i klassen `Department` som kan referera till en eller flera `Employee` objekt.

I detta testfall utnyttjas klassen `Vector` för att skapa relationsvariabel (se även kapitel 5.3.2.3 för utnyttjande av klassarrayer). En modell som överensstämmer med den utnyttjade koden ska innehålla två klasser, `Department` och `Employee`, samt en associationsrelation mellan dessa med kardinaliteten 1 på `Department` sidan och kardinaliteten * på `Employee` sidan (se exempelvis modeller i testfall 2).

5.3.2.3 Testfall 10

I detta testfall genereras modeller utifrån kod specificerad i Appendix E.3 samt kodskelett som genererats av Visio under testfall 3 (Appendix D.3). Koden beskriver en dubbelriktad många-till-många relation mellan klasserna `Company` och `Product` och innebär att en instans av klassen `Company` känner till flera `Product` objekt och vice versa. Kardinaliteten för relationen ska även vara * på båda ändarna (se exempelvis modeller i testfall 3).

Koden i Appendix E.3 utnyttjar klassen `Vector` för att skapa relationsvariabeln i klassen `Company` som kan referera till flera `Product` objekt, samt en relationsvariabel i klassen `Product` som är en `Company` array. Denna lösning har valts för att kunna undersöka om Together skiljer på olika typer av relationsvariabler och illustreras i figur 5.34.

```
class Company {
   private Vector relationVariable1;
}

class Product {
   private Company[] relationVariable2;
}
```

Figur 5.34 *Relationsvariabler i klassen Company och klassen Product.*

5.3.2.4 Testfall 11

Detta testfall beskriver en arvsrelation mellan klasserna `Product` och `HWSW`, där klassen `Product` är superklass och `HWSW` är subklass, dvs. `HWSW` är en arvinge till `Product`. För att utföra detta testfall skapades kod i programmet UltraEdit (Appendix E.4) som sedan användes i Together för att skapa en modell över klasserna och arvsrelationen mellan dem. Reverse engineering i CASE-verktyget har även utförts på kod som genererades i Visio under testfall 4 (Appendix D.4).

5.3.2.5 Testfall 12

Detta testfall används för att evaluera reverse engineering i Together av kod som beskriver en aggregatrelation i kompositionsform mellan klasser. En aggregatrelation i kompositionsform innebär att en klass är ”en del” av en annan klass. Detta testfall använder två klasser, `Bil` och `Chassi`, för att illustrera denna typ av relation. Relationen utläses:

- Ett chassi är en del av/tillhör en bil.

`Bil` klassen kan ses som den ”större” klassen i relationen. En korrekt implementering av denna relation är att skapa kod som, när ett `Bil` objekt skapas, så skapas även en instans av klassen `Chassi` tillhörande `Bil` objektet. Detta kan utföras på följande sätt:

- Skapa en relationsvariabel i klassen `Bil`, som kan referera till ett `Chassi` objekt, därefter skapas en instans av klassen `Chassi` som relationsvariabeln refererar till.

Skapandet av en instans av klassen `Chassi` kan utföras på flera olika ställen i koden, i denna rapport har exempel 1 utnyttjats (figur 5.35):

- På samma rad som relationsvariabeln skapas (figur 5.35).
- I konstruktorn (figur 5.36).

```
class Bil {
   private Chassi relationVariable = new Chassi();
   :
}
```

Figur 5.35 *Implementering av en aggregatrelation i kompositionsform på en rad*.

```
class Bil {
   private Chassi relationVariable;
   :
   public Bil() {
      relationVariable = new Chassi();
   }
   :
}
```

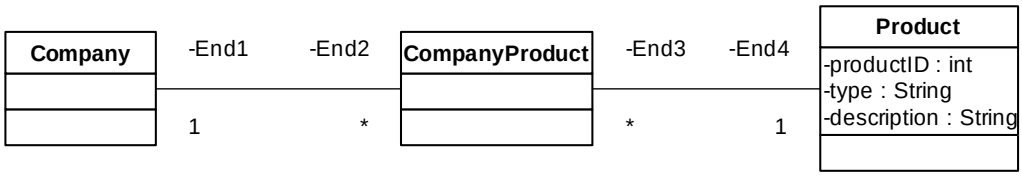
Figur 5.36 *Implementering av en aggregatrelation i kompositionsform mha konstruktor*.

5.3.2.6 Testfall 13

I detta testfall har kod skapats för att beskriva en dubbelriktad association mellan två klasser, med en tillhörande associationsklass till relationen (Appendix E.6). Kodskelett genererad från Visio i testfall 6 har inte evaluerats, då verktyget inte genererade associationsklassen `CompanyProduct` (se kapitel 6.1.6). För att implementera kod som beskriver en associationsklass användes följande steg (Seifzadeh, 2000: 32):

- Skapa en relationsvariabel i klassen `CompanyProduct` som refererar till ett en instans av klassen `Company`.
- Skapa en relationsvariabel i klassen `CompanyProduct` som refererar till en instans av klassen `Product`.
- Skapa en relationsvariabel i klassen `Company` som kan referera till en eller många instanser av klassen `CompanyProduct`.
- Skapa en relationsvariabel i klassen `Product` som kan referera till en eller många instanser av klassen `CompanyProduct`.

En korrekt modell för ovanstående krav illustreras i figur 5.37. Denna modell kommer även att användas som referens vid analysen av reverse engineeringen i kapitel 6.2.6.



Figur 5.37 *Modell över en implementering i Java av en dubbelriktad association mellan två klasser där relationen har en associationsklass knuten till sig.*

6 Resultat

6.1 Kodgenerering

Gemensamt för alla testfallen var att de uppfyllde ”100% pure” Java (Meloan, 1997; Sun, 2001b) och gick att kompilera i Suns kompilator (Sun, 2001a). Genererade kodskelett med tillhörande modeller återfinns i Appendix D.

6.1.1 Resultat av testfall 1

Som tidigare nämnts lyckades båda CASE-verktygen att generera kodskelett som överensstämmer med modellerna. Skillnaden mellan verktygen är att Visio krävder att man explicit anger att relationen är riktad så att en medlemsvariabel skapas för att behålla semantiken i relationen.

Together i sin tur kräver att man ställer in att metoder ska skapas för åtkomst till relationsvariabeln i klassen `Product`. Dessa metoder måste dock inte finnas för att relationen ska vara valid. Om metoderna ska vara med eller ej beror på vilken funktionalitet och skyddsnivå som programmet ska ha.

Visio skapade även viss överflödig kod, exempelvis importsatsen av klassen `ProductStatus` i klassen `Product`. Denna sats är redundant då båda klasserna finns i samma paket (Sun, 2001d). Visio genererar även konstruktorer som endast anropar superklassens konstruktor. Även detta är överflödigt då en klass utan konstruktor automatiskt utför detta anrop (Sun, 2001e).

6.1.2 Reslutat av testfall 2

Båda CASE-verktygen genererade kod som överensstämmer med modellen. Inget av verktygen klarar, i sitt standardutförande, att generera klassarrayer, utan använder vektorer eller andra generella datastrukturer (s.k. *containers*). Att utnyttja generella datastrukturer innebär en försämrad abstraktionsnivå, dvs. att objekt tappar sin identitet (Eckel, 2000: 54, 450). Vid implementation av kod leder detta till att man i många fall måste använda sig av casting på objektet/objekten som är lagrade i relationsvariabeln, vilket kan innebära en ökad exekveringstid (Eckel, 2000: 54-55) och ej optimal kod (se exempel figur 6.1).

Enligt Eckel (2000: 51) finns det dock även fördelar med att lagra objekt i vektorer eller andra generella datastrukturer:

- En container kan utökas/förminskas dynamiskt, dvs. objekt kan läggas till i en containern och utrymme för dessa skapas dynamiskt.
- En container kan lagra olika typer av objekt.
- En container innehåller generella metoder för att ta bort, lägga till och hämta objekt.

Med generell datastruktur (Vector)

```
import java.util.Vector;

public class Class {
   public void invokeMetodInAssociationVariable(int pos) {
      if(lnkAnotherClass.elementAt(pos) != null) {
         ((AnotherClass)lnkAnotherClass).aMethod(); //Casting
      }
   }

   /** @associates <{AnotherClass}> */
   private Vector lnkAnotherClass;
}
```

Med klassarray

```
public class Class {
   public void invokeMetodInAssociationVariable(int pos) {
      if(lnkAnotherClass(pos) != null) {
         lnkAnotherClass[pos].aMethod();
      }
   }

   /** @associates <{AnotherClass}> */
   private AnotherClass[] lnkAnotherClass;
}
```

Figur 6.1 *Skillnaden mellan att utnyttja en klassarray eller en generell datastruktur*.

Att utnyttja klassvariabler istället för generella datastrukturer går att utföra i Together genom att skapa ett pattern (figur 6.2) eller att manuellt namnge dess datatyp (i detta fall klassnamnet `Employee`, figur 6.3). Fördelen med att skapa en relationspattern är att man får en generell relation som kan utnyttjas för flera klasser (Together, 2000: 184). För fullständig beskrivning av hur man skapar patterns hänvisas till Togethers manualer. Information om hur detta kan utföras i Visio har inte kunnat identifieras av författaren.

Pattern

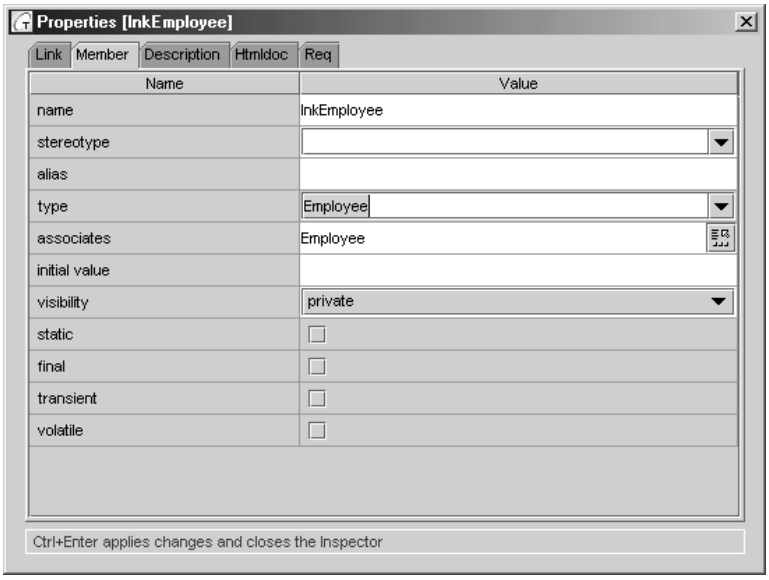
```
/**@associates <{%Dst%}>*/
private %Dst%[] %Name%;
```

Genererad kod (utifrån figur 6.1)

```
/** @associates <{AnotherClass}> */
private AnotherClass[] lnkAnotherClass;
```

Figur 6.2 *Ett pattern i Together som skapar en klassarray*.

En fördel som Together har över Visio är att kommentarer skapas som beskriver till vilken klass associationen härrör. Detta gör det enklare att förstå relationer när man läser källkod. Skillnaden mellan Visio och Together illustreras i figur 6.4. Dock går detta att förbättra i Visio genom att manuellt döpa om relationsslutpunkterna (figur 6.4)



Figur 6.3 *Manuell inställning av datatyp för relationsvariabel*.

Visio

```
private java.util.Vector mEnd1 = null;
```

Visio (med manuellt omdöpt relationsslutpunkt)

```
Private java.util.Vector mEmployee = null;
```

Together

```
/**
 * @associates <{Employee}>
 */
  private Vector lnkEmployee;
```

Figur 6.4 *Kommentering av relationsvariabler i Visio och Together*.

6.1.3 Resultat av testfall 3

Visio genererade ett kodskelett som uppfyller kraven på en många-till-många relation, dvs. skapade en relationsvariabel i båda källkodsfilerna. Together ControlCenter skapar dock endast en relationsvariabel i en av klasserna i modellen och genererar alltså endast en relationsvariabel i en av källkodsfilerna (Appendix D.3). En av anledningarna för detta är att Together anser att alla relationer har en client och supplier. Vilken av klasserna som är client och vilken som är supplier bestäms av hur man väljer att rita relationen mellan klasserna. Relationer i detta verktyg ritas genom att klicka och dra muspekaren från en klass (client) till en annan (supplier).

För att lösa ovanstående problem får man manuellt skapa en relationsvariabel i supplierklassen (figur 6.5). För att undvika att Together ritar två relationer mellan klasserna får denna variabel inte ha kommentaren ”`/** @associates <{Class1}> */`” ovanför dess implementation. Detta beror på Togethers sätt att hantera kopplingen mellan modell och källkod. Vissa fördefinierade tecken och ordkombinationer i

källkoden används av Together för att definiera hur modellen är uppbyggd. Dock är denna konstruktion uppbyggd på ett sätt som inte inverkar på standard Java och innebär alltså ”100% pure” Javakod.

```
/* Generated by Together */

public class Product {
   private int productID;
   private String type;
   private String description;

   /**
    * Manuellt inskriven relationsvariabel
    */
   private Vector lnkCompany;
}
```

Figur 6.5 *Manuellt skapad relationsvariabel i Together.*

6.1.4 Resultat av testfall 4

Båda CASE-verktygen genererade kodskelett där arvsrelationen var korrekt implementerad. För att generera dessa kodskelett krävdes inga inställningar eller användning av templates mm vare sig Visio eller Together.

6.1.5 Resultat av testfall 5

Ingen av CASE-verktygen lyckades att skapa kodskelett som gick att kompilera. Visio skapade en relationsvariabel, samt genererar kod som skapar ett `Chassi` objekt som relationsvariabeln refererar till. Denna kod innehåller dessvärre syntaxfel och författaren kunde inte finna information om att det går att konfigurera verktyget så att detta fel inte inträffar. Detta fel illustreras i figur 6.6, Visio genererar inte parenteser efter klassnamnet vilket resulterar i kompileringsfel.

```
// Kod genererad av Visio
private Chassi mEnd2 = new Chassi;

//Kod genererad av Together
private Chassi lnkChassi;

// Exampelkod med rätt syntax
private Chassi variabelNamn = new Chassi();
```

Figur 6.6 *Illustration av kod genererad av Visio och Together.*

Together genererar en relationsvariabel men skapar inget objekt som denna variabel kan referera till (figur 6.6). För att lösa detta går det att skapa ett pattern i Together (figur 6.7). Detta pattern kan sedan användas på relationen för att skapa en syntaktiskt

korrekt implementering av ett aggregatförhållande i kompositionsformat mellan två klasser.

```
/**@link aggregationByValue*/
private %Dst% %Name% = new %Dst%();
```

Figur 6.7 *Pattern i Together som genererar ett korrekt aggregatförhållande i kompositionsformat.*

6.1.6 Resultat av testfall 6

Vid generering av kodskelett för detta testfall lyckades inget av CASE-verktygen att skapa en komplett implementering av dess modell. Detta kan bero på att i Java måste man implementera modeller som beskriver associationsklasser med vanliga klasser eller attribut (Kaitanen, 2001)[9].

Författaren kunde inte finna information, vare sig i dokumentationen, på verktygens webbplats eller i hjälpfilerna, om att det går att generera kod för associationsklasser eller hur man ska översätta modeller till kod i respektive verktyg som ersätter associationsklasser.

Det går att modellera associationsklasser i Together, men inte att lägga till attribut och metoder i dessa. I Together används endast en kommentar i koden för att beskriva att relationen har en associationsklass, vilket är skälet att det inte går att skapa några attribut i denna klass (figur 6.8). Visio stödjer modellering av associationsklasser bättre än Together, det går att både lägga till attribut och metoder på samma sätt som i en vanlig klass.

```
/**
 * @associationAsClass CompanyProduct
 */
```

Figur 6.8 *Kommentar i Together som beskriver att relationen har en associationsklass.*

Varken Together eller Visio lyckades att generera kodskelett för associationsklassen `CompanyProduct`. I Together skapades även endast en relationsvariabel i en av klasserna, vilket beror på att verktyget ser alla relationer som riktade (se även kapitel 6.1.3).

Resultatet av detta testfall visar på att inget av de undersökta CASE-verktygen klarar att generera kod från associationsklasser. I klasserna `Company` och `Product` produceras även inga relationsvariabler som refererar till klassen `CompanyProduct`. De ovanstående begränsningarna innebär alltså en förlorad semantik mellan kod och modell.

[9] För exempel hur associationsklasser kan implementeras i Java se Kaitanen (2001).

6.1.7 Resultat av testfall 7

Båda CASE-verktygen skapar samtliga relationsvariabler som behövs i klassen `HWSW` enligt definition i kapitel 5.3.1.7. Vidare genereras kod som definierar att klassen `HWSW` är en subklass till klassen `Product`.

I klassen `Product` genererar båda verktygen en relationsvariabel till klassen `ProductStatus`. Dock hade både Visio och Together samma problem som redogörs i kapitel 6.1.6, dvs. att generera kod för associationsklassen.

Som definierats i kapitel 5.3.1.7 ska det skapas två relationsvariabler i klassen `Company`. Dessa variabler har korrekt implementerats av båda verktygen (Appendix D.7).

Resultatet av detta testfall visar att det inte framkommit fler semantiska förluster än vad som har påvisats i tidigare utförda testfall.

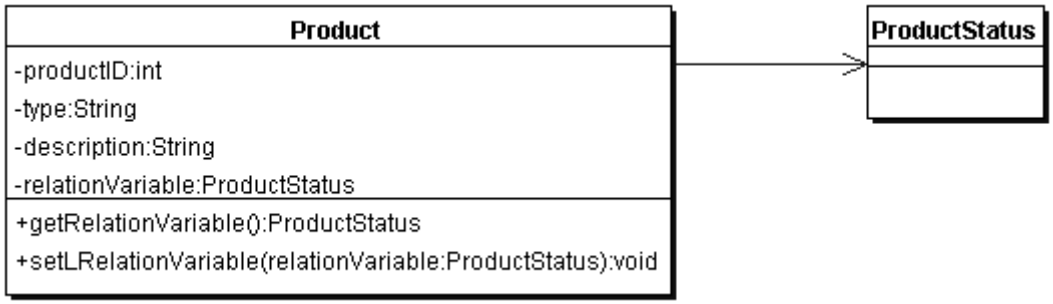
6.2 Reverse engineering

Modellerna i nedanstående testfall har skapats i Together ControlCenter 4.3 utifrån kod i Appendix E och kodskelett från Visio i Appendix D. I Together går det att ställa in via alternativ m.m. hur reverse engineering ska utföras samt hur genererade modellerna ska formateras. Denna rapports resultat baseras på standardinställningar som verktyget har vid en nyinstallation.

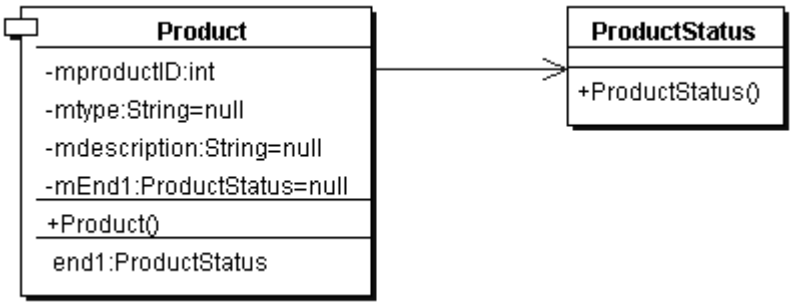
I figurerna nedan ritas relationer med en pil. Detta beror på att Together ritar alla relationer som definieras i källkoden med en relationsvariabel som har ett namn som inte börjar med texten `lnk`, som en pil (Together, 2000: 81). Detta är dock endast en visuell skillnad enligt dokumentationen och denna funktion går att stänga av eller konfigureras på ett annat sätt (Together, 2000: 81).

6.2.1 Resultat av testfall 8

I figur 6.9 illustreras den genererade modellen som verktyget skapade från koden i Appendix E.1, och figur 6.10 visar den genererade modellen som skapades från kod-skelett genererad av Visio (Appendix D.1).



Figur 6.9 *Modell genererad av Together från kod specificerad i Appendix E.1.*



Figur 6.10 *Modell genererad av Together från kodskelett genererad i Visio (Appendix D.1).*

Gemensamt för de båda modellerna är att relationernas kardinalitet inte modelleras vid en reverse engineering. I modellen skapad utifrån koden i Appendix E.1 (figur 6.9), dvs. utifrån den manuellt skapade koden klarade Together att generera modeller som är ekvivalenta med koden. Relationsvariabeln `relationVariable` samt get- och setmetoder för denna visas i modellen. Detta beror på att Together endast anser att variabler som börjar med texten `lnk` är relationsvariabler och ska döljas i modellen (om relationsvariabeln hade haft ett namn som börjar med texten `lnk` så hade denna inte visats i modellen, se även kapitel 5.3.1, figur 5.5).

Modellen som Together genererade från kodskelettet i Appendix D.1 transformerades inte korrekt av verktyget och innehåller semantiska fel. Av figur 6.9 kan man utläsa att Together modellerar en variabel av typen *property* med namnet `end1` av typen `ProductStatus`. Detta beror på att CASE-verktyget tolkar metoder med namn som börjar med texten ”get” eller ”set” som metoder som hämtar och ändrar en medlemsvariabel (Together, 2000: 264).

Om det t.ex. finns en metod med namnet `getIncome()` så tolkas detta av Together att det finns en variabel med namnet `income` och att nämnda metod är knuten till denna variabel. Detta innebär att Together skapar en felaktig modell vid reverse engineeringsprocessen i detta fall. Dock lägger inte Together till någon kod för att skapa denna variabel, utan håller den ursprungliga koden intakt.

Författaren har funnit två olika sätt som hade kan utföras för att undvika att verktyget utför, i detta fall, en felaktig tolkning:

- Döpa om relationsvariabeln eller get- och setmetoder så att de matchar varandra enligt ovanstående definition, dvs. om relationsvariabeln har namnet `relationVariable` ska get- och setmetoden för denna variabel ha namnet `getRelationVariable()` respektive `setRelationVariable()`. I detta fall hade Together utfört en korrekt reverse engineering om relationsvariabeln `mEnd1` i klassen `Product` hade döpts om till `end1` eller om get- och setmetoden hade döpts om till `getMEnd1()` respektive `setMEnd1()`.
- Döpa om get- och setmetoden så att de inte börjar med texten ”get” eller ”set”.

6.2.2 Resultat av testfall 9

De skapade modellerna illustreras i figur 6.12 och figur 6.13. För modellen som genererats från kodskelett skapade i Visio uppstod samma problem med get- och setmetoderna som i testfall 8. Ingen av modellerna lyckades att skapa relationen mellan klasserna.

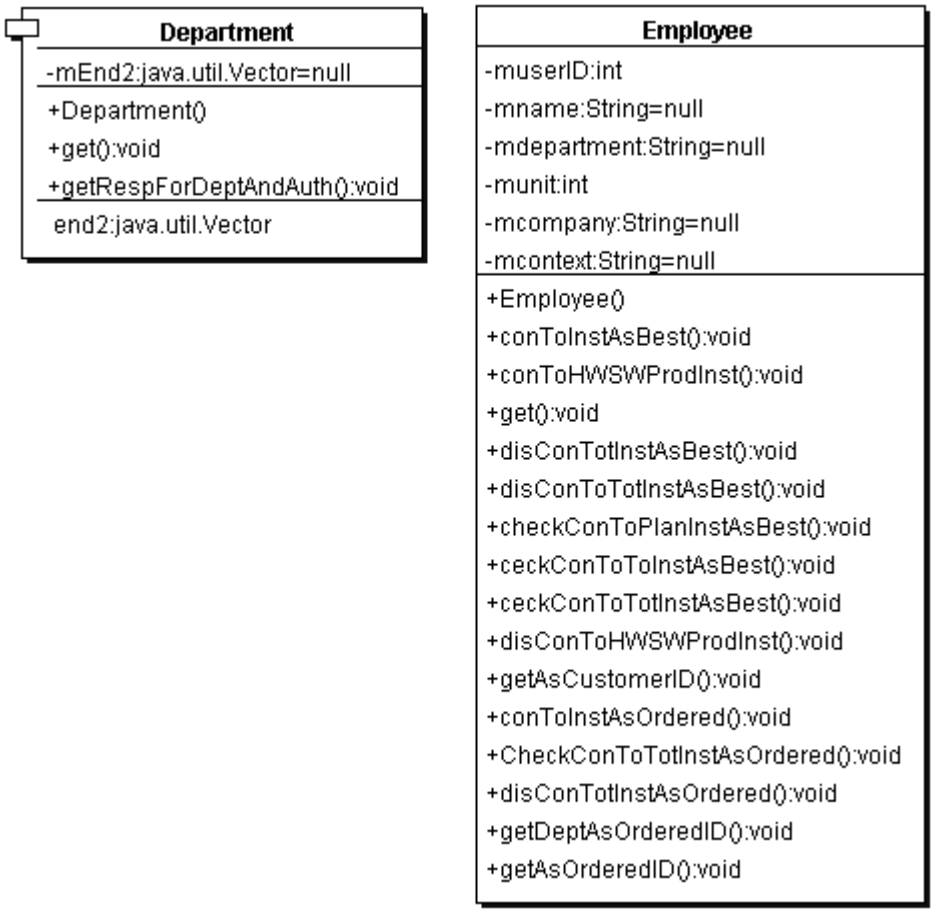
Författaren kunde inte finna någon direkt information varför ett-till-många relationer ej skapas i Togethers manualer, dock framgår det i manualen att man måste kommentera i källkoden att en variabel relaterar till en annan klass (Together, 2000: 77). Detta utförs automatiskt i verktyget när man utnyttjar dess designverktyg och templates mm, men om en relationsvariabel ska läggas till manuellt i koden måste den föregås med en kommentar (figur 6.11).

```
/** @associates <{Class2}>*/
private Vector lnkClass2;
```

Figur 6.11 *Exempel på en relationsvariabel i klassen Class1 som refererar till klassen Class2.*

Relationsvariabler som är av samma typ som klassen som variabeln refererar till behöver dock inte föregås med en kommentar enligt figur 6.11, vilket även skulle skapat en korrekt relation mellan klasserna (se exempelvis testfall 8). Vidare så genererades ingen kardinalitet för relationerna i modellerna.

Figur 6.12 *Modell genererad av Together från kod specificerad i Appendix E.2.*



Figur 6.13 *Modell genererad av Together från kodskelett genererad i Visio (Appendix D.2).*

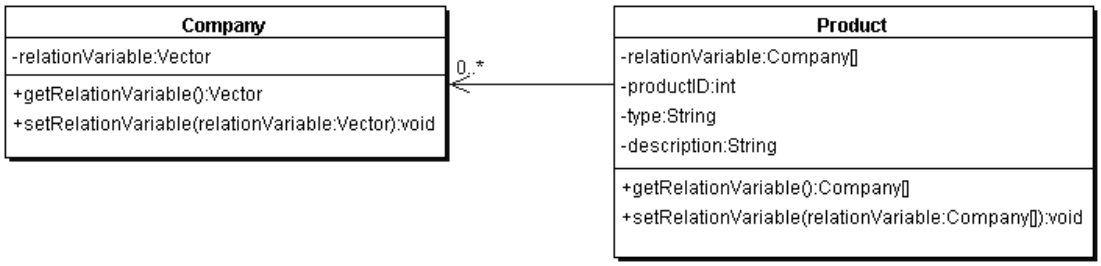
6.2.3 Resultat av testfall 10

Together lyckades ej att transformera koden som har genererats i Visio (Appendix D.3) utan att semantiska förluster uppstod (figur 6.14). Som framgår av modellen genererades ingen relation mellan klasserna `Company` och `Product`. Detta beror på samma anledning som i testfall 2, dvs. att get- och setmetoderna inte följde namnkonventioner som definierats i Together tillsammans med relationsvariabeln (se även kapitel 6.2.3).

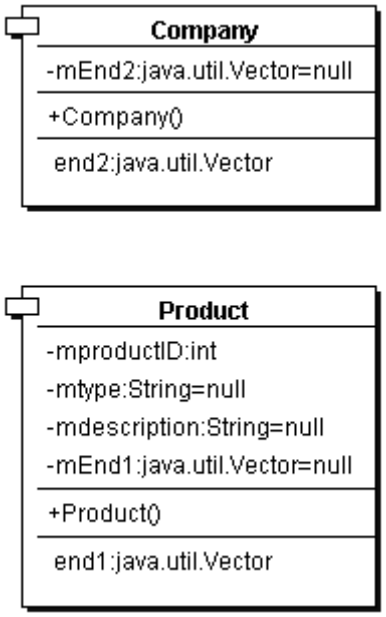
Reverse engineering utfördes även på kod som författaren själv skrivit (Appendix E.3). Modellen som CASE-verktyget genererade återfinns i figur 6.15. I denna kod utnyttjades klassen `Vector` för att skapa en relationsvariabel i klassen `Company` som kan referera till flera `Product` objekt. I klassen `Product` skapades en `Company` array som utnyttjas till att vara referensvariabel och kan referera till flera `Company` objekt.

Den genererade modellen (figur 6.14) skapar en relation mellan klasserna `Company` och `Product`, dock är relationen endast riktad åt ett håll och är alltså inte en dubbelriktad relation. Detta problem, att Together ser alla relationer som ”riktade” med en clientklass och en supplierklass har diskuteras i kapitel 6.1.3 och tas även upp i

följande stycke. Verktyget modellerade även endast kardinalitet för relationen åt ett håll (figur 6.14).



Figur 6.14 *Modell genererad av Together från kod specificerad i Appendix E.3*.



Figur 6.15 *Modell genererad av Together från kodskelett genererad i Visio (Appendix D.3).*

I detta testfall har två alternativ implementationssätt undersökts som beskriver en dubbelriktade många-till-många relation. De två alternativen utnyttjar olika kombinationer av relationsvariabler för att implementera relationen enligt följande:

- en relationsvariabel i båda klasserna som är av typen `Vector` (figur 6.15).
- En relationsvariabel i klassen `Product` som är av typen `Company` array och en relationsvariabel i klassen `Company` som är av typen `Vector` (Figur 6.14).

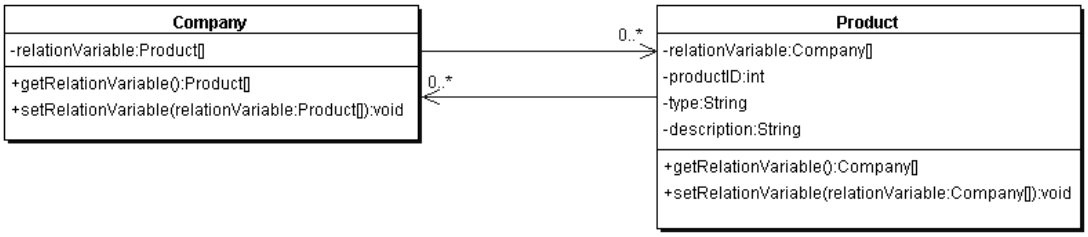
För att utvärdera hur Together omvandlar kod som innehåller relationsvariabler i båda klasserna i form av en array som kan referera till instanser av klassen `Company`, så har koden i Appendix E.3 ändrats enligt figur 6.16. Resultatet av reverse engineering i Together av denna förändring redovisas i figur 6.17.

Utifrån den ändrade koden i figur 6.16 genererar Together två enkelriktade relationer mellan klasserna `Company` och `Product` (figur 6.17). Önskvärt vore om verktyget

hade skapat en dubbelriktad relation, hur, och om, detta kan utföras i Together har dock inte författaren kunnat finna någon information om.

```
class Company {
      private Product[] relationVariable;

      public Product[] getRelationVariable() {
            return relationVariable;
      }

      public void setRelationVariable(Product[] relationVariable) {
            this.relationVariable = relationVariable;
      }
}
```

Figur 6.16 *Förändring av kod från Appendix E.3.*



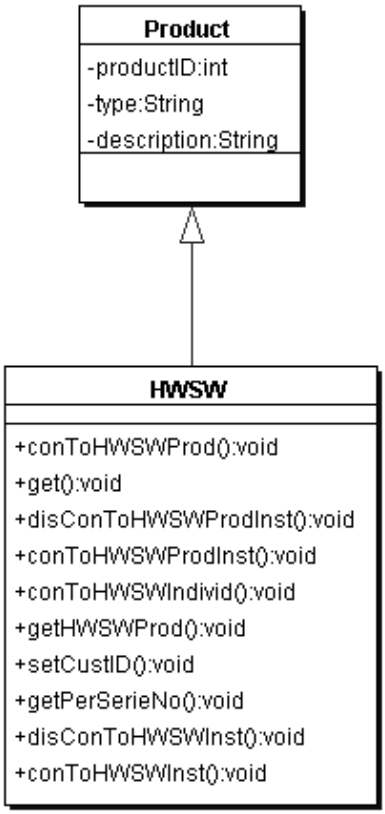
Figur 6.17 *Resulterande modell utifrån kod i figur 6.16.*

6.2.4 Resultat av testfall 11

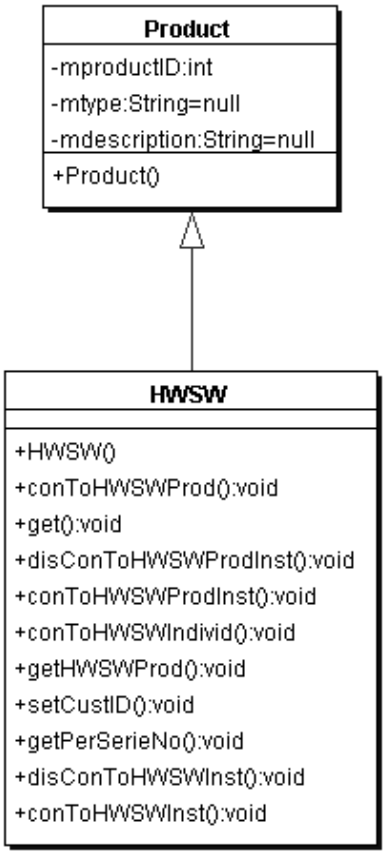
Två modeller skapades av Together med dess reverse engineeringsfunktion, en modell utifrån kod i Appendix E.4 och en modell utifrån kodskelett genererad av Visio under testfall 4 (Appendix D4). Dessa modeller presenteras i figur 6.18 och figur 6.19.

Koden som använts beskriver en arvshierarki mellan klasserna `Product` och `HWSW` där klassen `HWSW` är en subklass till klassen `Product`. Med UML notation visas denna relation med en pil från klassen `HWSW` som pekar på klassen `Product`.

Together genererade korrekta modeller, både för kod specificerad i Appendix E.4, samt för kod genererad av Visio i testfall 4 (Appendix D.4). Alla metoder och variabler finns med i modellerna och inga semantiska förluster uppkom under reverse engineeringsprocessen.



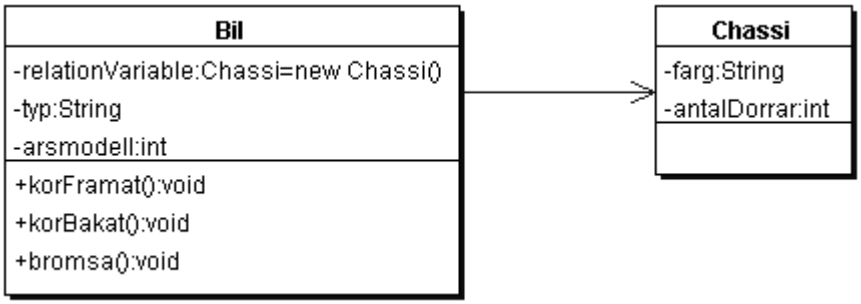
Figur 6.18 *Modell genererad av Together från kod specificerad i Appendix E.4.*



Figur 6.19 *Modell genererad av Together från kodskelett genererad i Visio (Appendix D.4).*

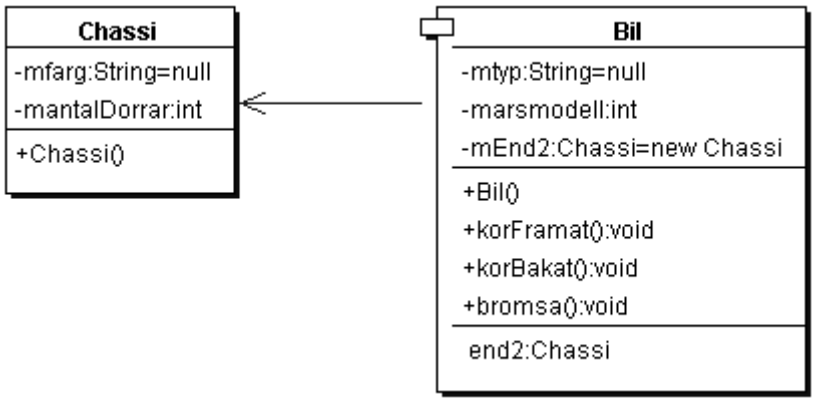
6.2.5 Resultat av testfall 12

I figur 6.20 illustreras den genererade modellen som verktyget skapade från koden i Appendix E.5, och figur 6.21 visar den genererade modellen som skapades från kodskelett genererad av Visio (Appendix D.5).



Figur 6.20 *Modell genererad av Together från kod specificerad i Appendix E.5*.

Together lyckades inte att skapa en aggregatsrelation i kompositionsform mellan klasserna `Bil` och `Chassi` från vare sig kod från Appendix E.5 eller kodskelett genererad av Visio i testfall 5 (Appendix D.5). Författaren har inte funnit någon information i Togethers dokumentation hur aggregatrelationer i kompositionsform översätts vid reverse engineering, men har funnit att i verktyget utnyttjas kommentarer för att ansvara för att en aggregatrelation ritas ut mellan två klasser (figur 6.22). Om denna kommentar tas bort från koden ändrar Together relationen till en associationsrelation, vilket stämmer överens med resultatet i detta testfall.



Figur 6.21 *Modell genererad av Together från kodskelett genererad i Visio (Appendix D.5).*

```
/** @link aggregationByValue */
```

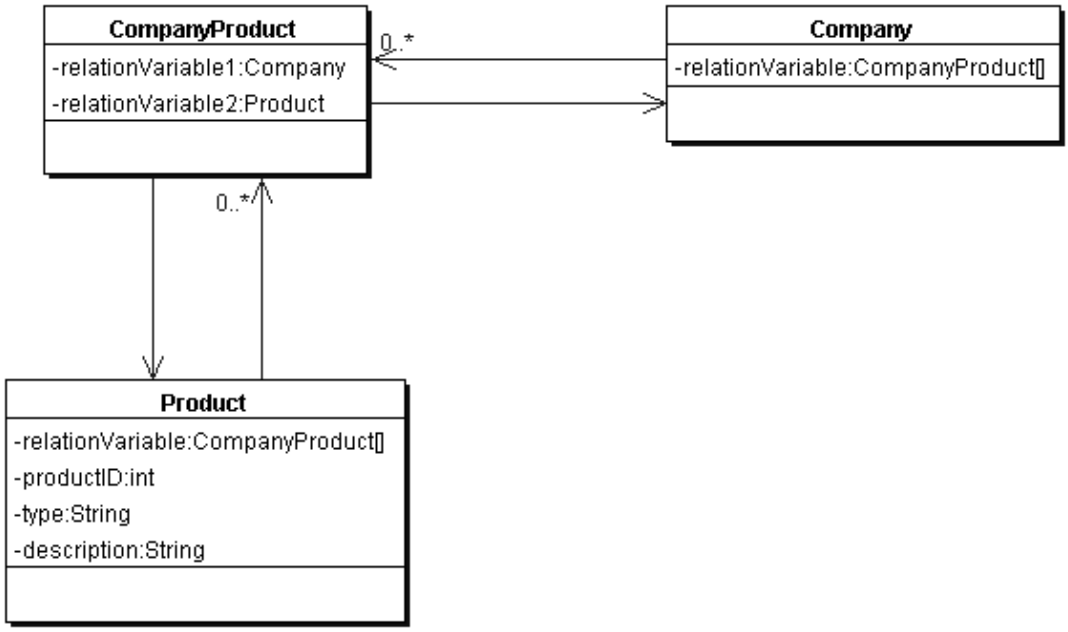
Figur 6.22 *Kommentar som måste föregå en relationsvariabel som beskriver en aggregatrelation i kompositionsform i Together ControlCenter 4.2.*

Together skapade även en associationsrelation, och inte en aggregatrelation, av kodskelettet som genererades av Visio i testfall 4 (Appendix D.5). Då denna kod

innehöll syntaktiska fel, som identifierades i kapitel 6.1.5, visar detta på att Together inte kontrollerar huruvida en relationsvariabel initieras eller ej. Detta anser författaren även vara ett bevis på att Together måste utnyttja kommentarer i koden för att kunna identifiera aggregatrelationer i kompositionsform mellan klasser.

6.2.6 Resultat av testfall 13

I figur 6.23 illustreras den genererade modellen som Together genererade från koden i Appendix E.6. Som uppmärksammats i tidigare kapitel (kapitel 6.1.3 och kapitel 6.2.3) genereras dubbla relationer mellan två klasser för att illustrera en dubbelriktad association. Together modellerade även endast ut kardinaliteten för en av relationerna mellan två klasser.



Figur 6.23 *Modell genererad av Together från kod specificerad i Appendix E.6.*

7 Analys

I denna rapport har det framkommit att vissa semantiska förluster uppstår vid kodgenerering och reverse engineering. I Appendix F sammanställs de testfall som har undersökts, vilka brister som transformeringen har, samt vilka åtgärder som kan utföras i verktygen för att förbättra resultatet.

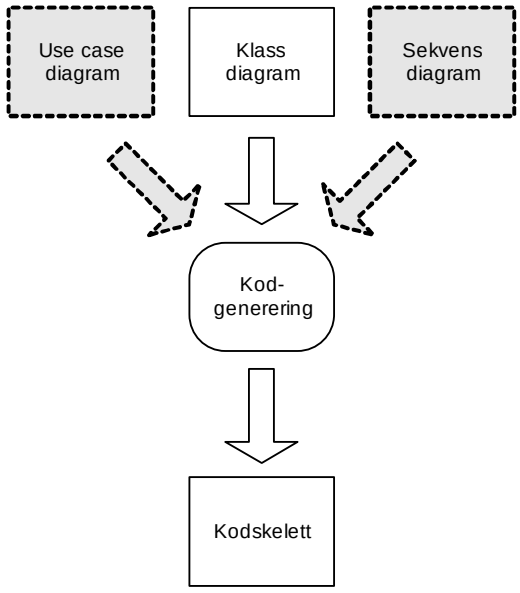
I detta kapitel kommer resultaten analyseras för att se hur de uppfyller och täcker Volvo ITs krav som tas upp i kapitel 3. Tillslut så jämförs resultatet med rapportens problemformulering (kapitel 3.1) för att kunna analysera och ge en sammanfattning på vilka svar som har framkommit i undersökningen.

7.1 Krav från Volvo IT

Rehbinder har skapat ett ramverk (Rehbinder, 2000: Appendix A) som innehåller krav på CASE-verktyg. Dessa krav har specificerats av CASE-experter från organisationen Volvo IT i Skövde och Göteborg. Denna rapport har utnyttjat en delmängd av dessa krav för att evaluera kodgenereringsfunktioner i två CASE-verktyg; Microsoft Visio 2000 Enterprise Edition och TogetherSoft Together ControlCenter.

7.1.1 Program skeleton from UML diagrams

- *Use cases, class diagrams, and sequence diagrams. From these it should at least be possible to auto generate stubs when work with the models is sufficiently concluded.*



Figur 7.1 *Diagramtyper som har utnyttjats respektive ej utnyttjats vid evaluering av verktygens kodgenereringsfunktioner.*

Denna undersökning har endast evaluerat verktygens förmåga att skapa kodskelett från klassdiagram med UML notation (figur 7.1). De modeller som har använts täcker grundläggande relationer som kan förekomma mellan klasser.

Resultaten i testfall 1 – 7 visar att inga semantiska förluster uppstår vid transformeringen av klassers metoder och attribut till källkod i Java. Dock klarar ingen av verktygen att korrekt översätta alla de typer av relationer mellan klasser som har undersökts.

Together ControlCenter har större möjlighet att påverka resultatet av en kodgenerering än Visio 2000 Enterprise Edition. Nackdelen med Together är att alla relationer mellan två klasser är riktade, med en supplier- och klientklass, vilket försvårar modelleringen av dubbelriktade relationer.

7.1.2 Code generation in any 3GL Language

- *The tool could furthermore generate code using any 3GL language, however Java is explicitly preferred.*

Båda verktygen klarar av att generera kodskelett i Java. Visio stödjer även kodgenerering till språken C++ och Visual Basic och Together stödjer kodgenerering till C++ och Corba IDL. I denna undersökning har endast kodgenereringsmöjligheter till Java undersökts.

7.1.3 Code generation tool independent

- *When CASE-tools generate code this code should not be CASE-tool specific. Thus when building code and when employing it in runtime the less influence the CASE-tool has the better.*

Både Together och Visio genererar kodskelett som uppfyller ”100% pure Java” (Sun, 2001b). Kodskeletten gick att kompilera i Sun Java 2 SDK, Standard Edition, version 1.3.0_02 förutom i ett testfall, testfall 5, där kod genererades som innehöll syntaktiska fel (kapitel 6.1.5).

7.1.4 Code platform independence

- *Important that the code generated is understandable and that it is independent of platforms and of the tool that generated it.*

Together skapar kommentarer i koden som beskriver relationers kardinalitet, och i vissa fall även till vilken klass en relationsvariabel refererar till. Som standard namnger Together relationsvariabler bättre än Visio (figur 7.2), vilket medför att koden blir enklare att tolka. Det går dock att döpa om relationsvariablerna i både Visio och Together manuellt för att förbättra läsbarheten av koden.

För att kontrollera att de genererade kodskeletten var plattformsoberoende utnyttjades verktyget JavaPureCheck 4.1.1 (Sun, 2001b). Detta verktyg kontrollerar att koden följer Suns riktlinjer för ”100% pure Java”. All genererad kod uppfyller dessa riktlinjer och kan alltså exekveras på de plattformer som Sun har implementerat en Javainterpretator till.

```
/**
 * Kod genererad från Visio
 */
public class Department {
   private java.util.Vector mEnd2 = null;
}

/**
 * Kod genererad från Together
 */
public class Department {
   /**
    * @associates <{Employee}>
    * @clientCardinality 1
    * @supplierCardinality *
    */
   private Vector lnkEmployee;
}
```

Figur 7.2 *Skillnad mellan kommentering och namngivning av relationsvariabler i Together och Visio.*

7.1.5 Round-trip engineering

- *Support for true round trip engineering where both models and the resulting implementation allows changes that are propagated in between.*
- *Visualising code by drawing models, being able to change the code, and being able to reverse engineer code. There should in this respect also be support for true round trip engineering.*
- *Desirable to go back from the implementation to the conceptual model even if the implementation is slightly altered so data has to be interchanged in between e.g. by support of a repository.*

Stöd för round-trip engineering har endast evaluerats i Together då författaren inte hade tillgång till verktyg som krävs för att utföra reverse engineering och round-trip engineering i Visio (kapitel 3.2).

Together utnyttjar källkodsfilen för en klass för att synkronisera modell och kod, samt uppdaterar dessa automatiskt och i realtid om någon förändring sker i antingen i modellen eller i koden. Det går även att manipulera koden i ett externt program då Together automatiskt kontrollerar om några förändringar har skett i källkodsfilen.

Detta innebär även att Together inte utnyttjar ett repository (TogetherSoft, 2000: 26). Verktyget stödjer dock integration av ett externt program som tillhandahåller ett repository och versionskontroll (TogetherSoft, 2000: 110).

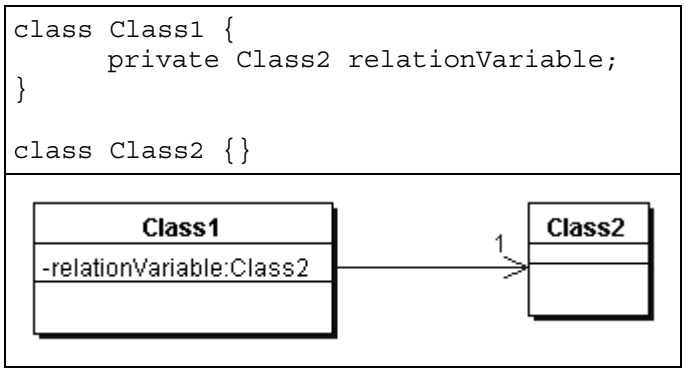
7.1.6 Reverse engineering

- *Being able to reverse engineer code. There should in this respect be support for true round trip engineering.*
- *Round trip engineering should also be employed in reverse engineering.*

I reverse engineeringsprocessen lyckades inte Together, förutom i testfall 10, att generera kardinalitet för relationerna mellan klasserna. I testfall 10 används en array för att beskriver en många-till-många associationsrelation mellan klasserna. Av resultatet från testfall 8-13 så kan man se att verktyget endast klarar av att modellera kardinalitet för relationer som använder en array.

Författaren anser att reverse engineering av kardinalitet även hade kunnat implementeras i verktyget för ett-till-ett relationer genom att utnyttja följande tumregler (se även figur 7.3):

- Då en variabel har en datatyp som refererar till en befintlig klass ska relationen ha en kardinalitet av typen 1.

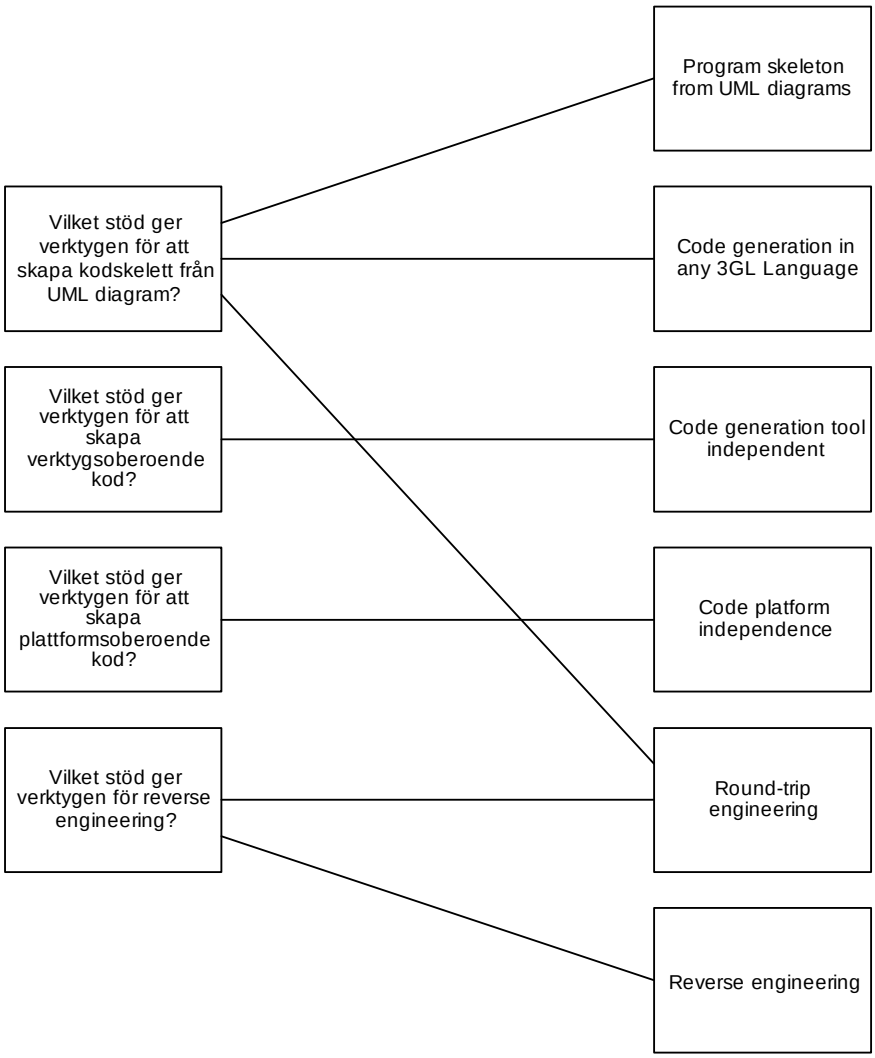


Figur 7.3 *Kodexempel och resulterande modell som visar hur Together hade kunnat utföra en reverse engineering på kod som beskriver en ett-till-ett relation som även tar hänsyn till kardinaliteten.*

Together hade även viss svårighet att utföra en reverse engineering på kod som genererats i Visio under testfall 1-7. Som framgår i tidigare kapitel är en av anledningen för detta att Together utnyttjar kommentarer och namnkombinationer för att generera modeller från kod (se exempelvis kapitel 6.2.1).

7.2 Frågor från problemformuleringen

Detta kapitel relaterar de resultat som har identifierats i rapporten i relation till frågorna som ställdes i problemformuleringen (kapitel 3.1). Då dessa har tagits upp i föregående kapitel används en figur för att schematiskt beskriva sambanden mellan frågorna i problemformuleringen och kraven från Volvo IT (figur 7.4).



Figur 7.4 *Samband mellan krav från Volvo IT och frågorna i problemformuleringen.*

Denna undersökning ger svar på de frågor som ställts i problemformuleringen, vilket illustreras i figur 7.4. Rapporten ger en större insikt i vilken funktionalitet de två undersökta verktygen har samt vilka fördelar, skillnader, samt nackdelar de har vid designtransformeringar.

8 Slutsatser och fortsatt arbete

8.1 Diskussion

Denna rapport har utnyttjat modeller från Shahin Seifzadehs rapport (Seifzadeh, 2000). Seifzadeh undersökte kodgenereringsmöjligheter i Visio 2000 Enterprise med programmeringsspråket C++. I denna rapport har en nyare version av CASE-verktyget Visio använts och programmeringsspråket Java har använts.

Av två anledningar är det intressant att jämföra denna rapports resultat med resultaten från Seifzadehs rapport. Dels så får man en indikation om det skiljer på hur CASE-verktyget hanterar kodgenerering i olika programmeringsspråk (Java och C++), och dels om det går att finna att verktyget har ändrat sin funktionalitet mellan versionerna.

I Seifzadehs rapport lyckades Visio att utföra testfall 5, att generera kod från en modell som beskriver ett aggregatförhållande i kompositionsform mellan två klasser, bättre än i denna undersökning. Författaren anser att detta kan bero på att Visio hanterar kodgenerering till C++ bättre än kodgenerering till Java. Detta antagande grundas på att som standard så är Visio inställd för att generera C++ kod, samt att namngivning av metoder och attribut följer C++ konventioner.

De övriga kodgenereringstestfallen fick samma resultat i Seifzadehs rapport som i denna. Beroende på att författaren inte hade tillgång till Microsoft Visual J++ jämförs inte reverse engineeringsresultaten från Seifzadehs rapport med resultaten i denna rapport.

Seifzadeh (Seifzadeh, 2000) tar inte upp några ytterligare funktioner i Visio som inte gåtts igenom i denna undersökning. Skillnaden mellan den version av Visio som undersöks i denna rapport och versionen som används i Seifzadehs rapport är inte stora. Detta grundar författaren på att Microsoft köpte Visio från Visio Corporation och har endast släppt en s.k. service release. Denna service release innehåller enligt Microsoft (Microsoft, 2001):

- A closer integration with the Microsoft Office suite of products and a more consistent Office look and feel.
- Minor program improvements.

Nästa version av Visio, Microsoft Visio 2002, kommer inte att innehålla funktioner för kodgenerering och reverse engineering. Dessa funktioner kommer att flyttas över till Microsoft Visual Studio.NET (PR Newswire, 2001; Windows-Help.NET, 2001). Microsoft (Microsoft, 2001e) kommer att fortsätta att stödja och sälja Microsoft Visual J++ men har skapat något som de kallar ”Jump to .NET” som ska hjälpa utvecklare att flytta över sina Javaprojekt till Microsofts .NET platform.

Microsoft Visio 2000 Enterprise Edition stödjer endast kodgenerering och reverse engineering för Microsoft Visual C++, Microsoft Visual Basic och Microsoft Visual J++. Författaren tror att då Microsoft överför dessa funtioner till Microsoft Visual Studio.NET så kommer inte designtransformationer att stödjas för Microsoft Visual J++. Microsoft har skapat ett nytt programmeringsspråk, C# (uttalas C-sharp), som bygger på C och C++ och är tänkt att vara huvudprogrammeringsspråket i .NET

plattformen. Enligt författaren är detta är en indikation på att Microsoft kanske inte kommer att ha en produkt som stödjer designtransformation för Java.

8.2 Slutsatser

Detta arbete har undersökt kodgenererings- och reverse engineeringsmöjligheter i två representativa CASE-verktyg. Vid transformering mellan kod till modell och tvärt om uppstår i vissa fall semantiska förluster. De två verktygen innehåller avancerad funktionalitet och inställningsmöjligheter. Dock framgår det av denna rapport att i CASE-verktygens standardutförande går det inte att utföra alla funktioner som CASE-experter ställer på moderna CASE-verktyg.

Författaren anser dock att de båda verktygen är kompetenta och kan utnyttjas som stöd i ett mjukvaruutvecklingsprojekt. Författaren anser att Together har fler fördelar än Visio tack vare dess förmåga till round-trip engineering samt utnyttjandet av patterns och templates. Mest fördel får man om man utnyttjar Together från början i ett systemutvecklingsprojekt och utför all kodning i Together. Om man utför reverse engineering på kod som skapats för hand eller i ett annat verktyg måste man se till att ha döpt attribut och metoder enligt de namnkonventioner som Together förutsätter.

Together har även den fördelen att verktyget finns till flera plattformer än Visio (se Appendix A) samt finns i en fri version som inte innehåller alla avancerade funktioner. Visio har i sin tur fördelen av att ha bättre integration med övriga program från Microsoft.

Denna undersökning visar på att moderna CASE-verktyg inte kan ses som automatiska verktyg som tar hand om designtransformationer fullständigt. De två undersökta CASE-verktygen kan utnyttjas för att skapa modeller och kodskelett, men kan inte fullständigt automatisera processen att utföra designtransformationer i ett mjukvaruutvecklingsprojekt. I många fall behöver kod respektive modeller utformas utformas för att passa verktyget, vilket inte kan ses som optimalt.

Författaren anser sig ha kunnat svara på frågeställningarna från problem-formuleringen, vilket illustereras i figur 7.4. Reverse engineering och round-trip engineering har dock inte evaluerats i verktyget Visio då författaren inte har haft tillgång till krävda verktyg (se även kapitel 3.2).

Resultatet av denna undersökning stämmer även väl överrens med det förväntade resultatet som författaren identifierade i kapitel 3.3, dvs. att vid vissa design-transformationer uppstår semantiska förluster.

8.3 Förslag till fortsatt arbete

I ett framtida arbete skulle vara intressant att utvärdera hur Microsoft Visio 2000 Enterprise Edition hanterar reverse engineering, samt hur kommande verktyget Microsoft Visual Studio.NET implementerar kodgenerering och reverse engineering. I TogetherSoft Together ControlCenter är ett förslag till fortsatt arbete att evaluera hur

kodgenerering kan utföras från use-case och sekvensdiagram. Ett intressant arbete vore även att utvärdera huruvida Microsoft Visio kan utnyttja Visual Basic for Applications för att programmera Visio till att utföra designtransformationer som inte lyckades att utföras enligt denna rapport.

Referenser

Aimar, A., Khodabandeh, A., Palazzi, P. and Rousseau, B. (2001) "A Configurable Code Generator for OO Methodologies", ⟨http://webmaker.web.cern.ch/WebMaker/examples/CHEP94_codegene_1/www/codegene_1.html⟩ (15 April, 2001).

Andersen, E. S. (1994) *Systemutveckling – principer, metoder och tekniker*, 2:a uppl., Studentlitteratur, Lund.

Barclay, S. and Padusenko, S. (2001) Faculty of Education Computer Science, - Palette of Units, "CASE-Tools", ⟨http://educ.queensu.ca/~compsci/units/casetools.html⟩ (15 April, 2001).

Blaxter, L., Hughes, C. and Tight, M. (1996) *How to research*, Open University Press, Buckingham.

Carnegie Mellon University (1999) "What is a CASE Environment?", ⟨http://www.sei.cmu.edu/activities/legacy/case/case_whatis.html⟩ (5 April, 2001).

Cronholm, S. (1994) *Varför CASE-verktyg i systemutveckling? – En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer*, Humaniora och Samhällsvetenskap FHS-rapport 5/94, Linköpings Universitet, Linköping.

Dawson, C. W. (2000) *The Essence of Computer Projects: A Student's Guide*, Prentice Hall, Harlow.

Denzinger, J. (2001) "Association Classes", ⟨http://sern.ucalgary.ca/~denzinger/lecture12.pdf⟩ (8 Maj, 2001).

Eckel, B. (2000) *Thinking in Java*, 2nd ed., Prentince Hall PTR, Upper Saddle River, New Jersey.

Elmasri, R. and Navathe, S. B. (1994) *Fundamentals of Database Systems*", 2nd ed., The Benjamin/Cummings Publishing Company, Inc., California.

Eriksson, H-E. and Penker, M. (2000) *Business Modelling with UML: Business Patterns at Work*, John Wiley & Sons, Inc., New York.

Fisher, A. S. (1991) *CASE: Using Software Development Tools*, 2nd ed., John Wiley & Sons, Inc., New York.

Flecher, T. and Hunt, J. (1993) *Software Engineering and CASE: Bridging the Culture gap*, McGraw-Hill, Inc., New York.

Fowler, M. (2001) "Techniques for Object Oriented Analysis and Design: CASE Tools", ⟨http://www.aw.com/cseng/titles/0-201-89542-0/techniques/casetools.htm⟩ (15 April, 2001).

Humphrey, W. S. (1991) "CASE Planning and the Software Process" I Yeh, R. T. (Ed.) *CASE Technology*, Kluwer Academic Publishers, Massachusetts, 1992, sid. 59-75.

IDM Computer Solutions, Inc. (2001) "Text Editor - HEX Editor - HTML Editor - Programmers Editor - UltraEdit", ⟨http://www.ultraedit.com/⟩ (1 Maj, 2001).

Institute of Electrical and Electronics Engineers, Inc. (1990) *Glossary of Software Engineering Terminology*, ANSI/IEEE Std. 610.12-1990. IEEE, cop., New York.

ISO/IEC (1999) *Software Engineering – Guidelines for the adoption of CASE tools*, TR14471, ISO/IEC JTC1/SC7 Secretariat, Canada.

JDJ (2001a) "Java Developer's Journal - Readers' Choise Award", ⟨http://www.sys-con.com/java/readerschoice2001/⟩ (21 April, 2001).

JDJ (2001b) "Java Developer's Journal - Readers' Choise Award: Best Java Modeling Tool", ⟨http://www.sys-con.com/java/readerschoice2001/liveupdatemodeling2.cfm⟩ (21 April, 2001).

jGuru (2000) "What is an association class?", ⟨http://www.jguru.com/faq/view.jsp?EID=100819⟩ (8 Maj, 2001).

Johnson, C. (2001) "Basic Research Skills in Computing Science", ⟨http://www.dcs.gla.ac.uk/~johnson/teaching/research_skills/basics.html⟩ (17 April, 2001).

Joy *et al*. (2000) "The Java Language Specification", 2nd edition, ⟨http://java.sun.com/docs/books/jls/index.html⟩ (17 April, 2001).

Juric, R. and Kuljis, J. (1999) "Building an Evaluation Instrument for OO CASE Tool Assessment for Unified Modelling Language Support", I *Proceedings of the 32nd Hawaii International Conference on System Sciences*, IEEE, IEEE Computer Society Press, January, 1999.

Kaitanen, J. (2001) "Implementation Variations from UML", ⟨http://www.vtt.fi/tte/papers/j-uml/j-uml_example11.htm⟩ (18 Maj, 2001).

King, S. F. (1995) *Using and Evaluating CASE Tools: From Software Engineering to Phenomenology*, Warwick Business School, University of Warwick, Coventry.

Larman, C. (1998) *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Pretice-Hall PTR, New Jersey.

Lewis, T. G. (1991) *CASE: Computer-Aided Software Engineering*, Van Nostrand Reinhold, New York.

Lundell, B. and Lings, B. (1999) "Validating Transfer of a Method for the Development of Evaluation Frameworks", I Brown, A. and Remenyi, D. (Eds.) *Sixth European Conference on Information Technology Evaluation*, Brunell University, Uxbridge, UK, 4th-5th November, 1999, sid. 255-263.

Malmström, S., Györki, I. och Sjögren, P. A (1991) *Bonniers svenska ordbok*, 5:e uppl., Bonniers Fakta Bokförlag AB, Stockholm.

Meloan, S. (1997) "Getting There: 100% Pure Java Certification", ⟨http://java.sun.com/features/1997/dec/100certif.html⟩ (1 Maj, 2001).

Microsoft Corporation (2001a) "Microsoft Windows Millenium Edition", ⟨http://www.microsoft.com/WindowsMe/⟩ (2 Maj, 2001).

Microsoft Corporation (2001b) "Microsoft Visual J++", ⟨http://msdn.microsoft.com/visualj/⟩ (8 Maj, 2001).

Microsoft Corporation (2001c) "Microsoft Visual C++", ⟨http://msdn.microsoft.com/visualc/⟩ (8 Maj, 2001).

Microsoft Corporation (2001d) “Microsoft Visual Basic”, ⟨http://msdn.microsoft.com/vbasic/⟩ (8 Maj, 2001).

Microsoft Corporation (2001e) “Jump to .NET Q & A”, ⟨http://msdn.microsoft.com/visualj/jump/faq.asp⟩ (25 Maj, 2001).

Microsoft Corporation (2001f) “Microsoft Visio 2000 Service Release 1 (SR-1) Update”, ⟨http://office.microsoft.com/Downloads/2000/vis2000SR1ddl.aspx⟩ (25 Maj, 2001).

Nilsson, A. G. (1995) “Utveckling av metoder för systemarbete – ett historiskt perspektiv”, I Dahlbom, B. (Ed) *The Infological Equation – Essays in Honor of Börje Langefors*, Gothenburg Studies in Information Systems, Report 6, Göteborgs universitet, 1995.

ObjectSpace, Inc. (2001) “JGL Libraries”, ⟨http://www.objectspace.com/products/voyager/libraries.asp⟩ (4 Maj, 2001).

OMG (1999) *OMG Unified Modeling Language Specification version 1.3, June 1999*, Object Management Group, Inc., USA.

Patel, R. och Davidson, B. (1994) Forskningsmetodikens grunder: att planera, genomföra och rapportera en undersökning, 2:a uppl., Studentlitteratur, Lund.

Popkin Software (2001) “Build Class Diagram through Iterative Design – Round-trip Engineering for Java and C++”, ⟨http://www.processimprovement.com/resources/case_list.htm⟩ (17 April, 2001).

Post, G. and Kagan, A. (2000) “OO-CASE tools: an evaluation of Rose”, *Information and Software Technology*, 42(6), sid. 383-388.

PR Newswire (2001) “Microsoft Visio 2002 Now Available for Customer Evaluation”, Yahoo Finance, ⟨http://biz.yahoo.com/prnews/010306/sftu059.html⟩ (21 Maj, 2001).

Process Improvement Associates (2001) ”CASE tool taxonomy”, ⟨http://www.processimprovement.com/resources/case_list.htm⟩ (5 April, 2001).

Rehbinder, A. (2000) *On Applying a Method for Developing Context Dependent CASE-tool Evaluation Frameworks*, HS-IDA-MD-00-012, Institutionen för datavetenskap, Högskolan i Skövde, Skövde.

Rehbinder, A., Lings, B., Lundell, B., Burman, R. and Nilsson, A. (2000) *Developing a Framework for Pre-usage Evaluations of CASE-tools: a field-study*, Techical Report, HS-IDA-TR-00-002, Institutionen för datavetenskap, Högskolan i Skövde, Skövde, 3:e November.

Rehbinder, A., Lings, B., Lundell, B., Burman, R. and Nilsson, A. (2001) “Observations from a Field Study on Developing a Framework for Pre-Usage Evaluation of CASE Tools”, I Russo, N. L., Fitzgerald, B. and DeGross, J. L. (Eds.) *New Directions in Information Systems Development*, Kluwer Academic Publishers, Boston, sid. 211-220.

Seifzadeh, S. (2000) *Kodgenereringsmöjligheter i VISIO 2000 Enterprise*, HS-IDA-EA-00-109, Institutionen för datavetenskap, Högskolan i Skövde, Skövde.

SISU (1991) *Modelleringsansater för begrepps- och datamodellering – Beskrivning och försök till jämförelse*, Rapport nr 16, Svenska Institutet för Systemutveckling, Kista.

Skansholm, J. (2000) *Java From the Beginning*, Addison-Wesley, Harlow, England.

Sun Microsystem, Inc (2001d) "The Java Tutorial – Using Package Members", ⟨http://java.sun.com/docs/books/tutorial/java/interpack/usepkgs.html⟩ (18 Maj, 2001).

Sun Microsystem, Inc. (2001a) "Java 2 Platform, Standard Edition, version 1.3", ⟨http://java.sun.com/j2se/1.3/⟩ (2 Maj, 2001).

Sun Microsystem, Inc. (2001b) "100% Pure Java Certification Program", ⟨http://java.sun.com/100percent/⟩ (1 Maj, 2001).

Sun Microsystem, Inc. (2001c) "Java Development Kit, version 1.1.*", ⟨http://java.sun.com/products/jdk/1.1/⟩ (4 Maj, 2001).

Sun Microsystem, Inc. (2001e) "The Java Tutorial – Providing Constructors for Your Classes", ⟨http://java.sun.com/docs/books/tutorial/java/javaOO/constructors.html⟩ (18 Maj, 2001).

TogetherSoft (2000) *Together Documentation Set*, Together Corporation, USA.

TogetherSoft (2001) "A Practical Guide to Getting Started with Together ControlCenter", ⟨ http://a1612.g.akamai.net/f/600/1325/9d/164.109.49.29/files/services/tccguide/index.html⟩ (2 Maj, 2001).

Visio. (2000) *User's Guide for Microsoft Visio 2000 Enterprise Edition*, Visio Corporation, Seattle.

Windows-Help.NET (2001) "Microsoft Visio 2002 Now Available for Customer Evaluation", ⟨http://www.windows-help.net/features/visio2002.html⟩ (21 Maj, 2001).

Appendix A: Beskrivning av CASE-verktyg

Microsoft Visio 2000 Enterprise Edition

Företag:	Microsoft Corporation
Produktnamn:	Microsoft Visio 2000 Enterprise Edition
Version:	2000 Service Release 1
Hemsida:	http://www.microsoft.com/office/visio/
Utvärderingsversion:	ingen

Systemkrav (Intel):

Operativsystem:	Windows 95/98/ME/NT/2000
CPU:	Pentium
Minne:	80 Mb, 96Mb (NT/2000)
Hårddiskutrymme:	130 Mb

Stöd för Java:	Ja
Stöd för UML:	Ja
Stöd för kodgenerering:	Ja

Rational Rose

Företag:	Rational Software Corporation
Produktnamn:	Rational Rose
Version:	2001
Hemsida:	http://www.rational.com/products/rose/index.jsp
Utvärderingsversion:	15 dagar

Systemkrav (Intel):

Operativsystem:	Windows 95/98/NT 4/2000
CPU:	Pentium
Minne:	64 Mb
Hårddiskutrymme:	100 Mb

Systemkrav (Unix - Sparc):

Operativsystem:	Solaris 2.5.1/Solaris 2.6/Solaris 2.7/HP-UX 10.20/HP-UX 11.0/AIX 4.3.2/IRIX 6.5.5/Tru64 UNIX 4.0f
CPU:	SparcStation 20
Minne:	64 Mb
Hårddiskutrymme:	270 Mb

Stöd för Java:	Ja
Stöd för UML:	Ja
Stöd för kodgenerering:	Ja

Together ControlCenter

Företag:	TogetherSoft Corporation
Produktnamn:	Together ControlCenter
Version:	4.2
Hemsida:	http://www.togethersoft.com/us/products/index.html
Utvärderingsversion:	30 dagar

Systemkrav (Intel):

Operativsystem:	Windows 95/98/NT 4/2000
CPU:	Pentium 233 MHz
Minne:	128 Mb
Hårddiskutrymme:	95-210 Mb

Systemkrav (Unix - Intel):

Operativsystem:	RedHat Linux 5.2
CPU:	Pentium 233 MHz
Minne:	128 Mb
Hårddiskutrymme:	75 Mb

Systemkrav (Unix - Sparc):

Operativsystem:	SunOS Release 5.7 (Solaris7)
CPU:	Sparc Ultra-5 333 MHz
Minne:	128 Mb
Hårddiskutrymme:	75 Mb

Stöd för Java:	Ja
Stöd för UML:	Ja
Stöd för kodgenerering:	Ja

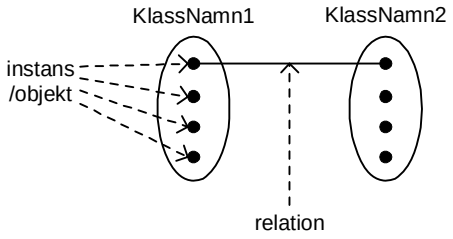
Appendix B: Hård- och mjukvarukonfiguration

Operativsystem:	Microsoft Windows Millenium (http://www.microsoft.com/WindowsMe/)
CPU:	Pentium II 300 MHz (http://www.intel.com/PentiumII/home.htm)
Minne:	192 Mb
Java kompilator:	Sun Java 2 SDK, Standard Edition, version 1.3.0_02 (http://java.sun.com/j2se/1.3/)
CASE-verktyg:	Microsoft Visio Enterprise Edition SR 1 (6.0.2072) (http://www.microsoft.com/office/visio/)
	TogetherSoft Together ControlCenter, version 4.2 (http://www.togethersoft.com/)
Text-editor	IDM UltraEdit, version 8.00b (http://www.ultraedit.com/)

Appendix C: Notation för relationsmodeller

Detta Appendix går igenom notationen som används för att beskriva relationer mellan instanser av klasser och är hämtad från Elmasri & Navathe (1994).

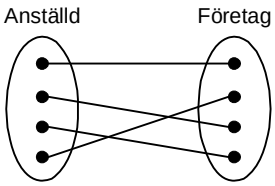
En *instans* (även *entitet*, *objekt*) av en klass ritas som en punkt, och kan endast tillhöra en klass. Detta illustreras genom att instanser ritas i en oval som representerar instansens typ, dvs. från vilken klass den härrör ifrån. Instanser från olika klasser kan ha en relation mellan sig. Detta ritas som en linje mellan instanserna (figur C.1).



Figur C.1 *Generell beskrivning av notation.*

För att beskriva *kardinalitet*, dvs. antalet relationsinstanser en entitet kan deltaga i, kan en instans ha en relation med noll, en eller flera andra instanser (Elmasri & Navathe, 1994: 52). Om en relation har en kardinalitet av typen ett-till-ett innebär detta att en instans från klass X endast känner till en instans från klass Y och en instans från klass Y endast känner till en instans från klass X. Detta illustreras i figur C.2 och relationen kan utläsas som:

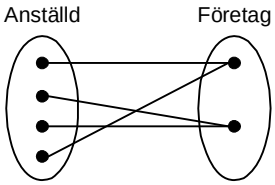
- En anställd kan endast arbeta på ett företag.
- Ett företag kan endast ha en anställd.



Figur C.2 *Ett-till-ett förhållande*.

En relation med en kardinalitet av typen ett-till-många innebär att en instans från klass X känner till en eller flera instanser från klass Y, men en instans från klass Y känner endast till en instans från klass X. Figur C.3 illustrerar denna typ av relation och kan utläsas som:

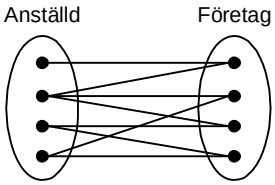
- En anställd kan endast arbeta på ett företag.
- Ett företag kan ha en eller flera anställda.



Figur C.3 *Ett-till-många förhållande*.

En relation med en många-till-många kardinalitet innebär att en instans från klass X känner till en eller flera instanser från klass Y, och en instans från klass Y känner till en eller flera instanser från klass X. Detta illustreras i figur C.4 och relationen kan utläsas som:

- En anställd kan arbeta på ett eller flera företag.
- Ett företag kan ha en eller flera anställda.

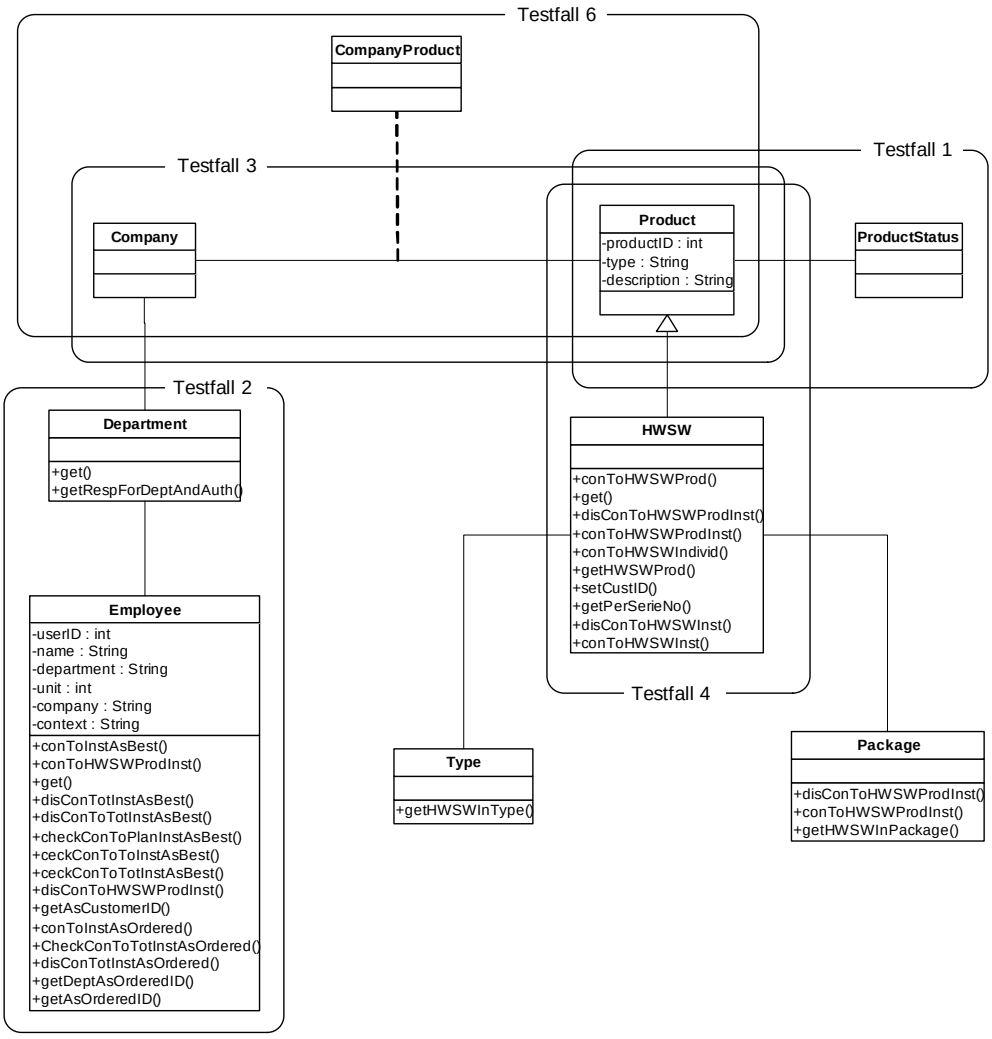


Figur C.4 *Många-till-många förhållande*.

Appendix D: Kodgenerering

Följande modeller har använts för att undersöka vilket stöd verktygen gav för att skapa kodskelett från UML diagram. Modellerna är ursprungligen skapade av Volvo IT och Shahin Seifzadeh (Seifzadeh, 2000: Bilaga 1) och har endast ändrats för att uppfylla Javas standard notation enligt Joy *et al*. (2000). Efter varje modell listas de kodskeletten som skapades utifrån modellen i CASE-verktyget.

I nedanstående figur illustreras selektionen av testfallen från modellen skapad av Volvo IT.

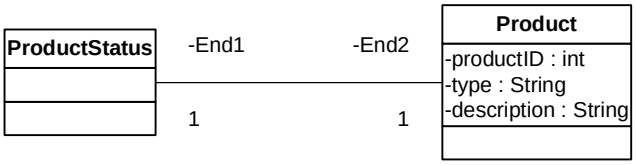


D.1 - Testfall 1

Modellerna representerar en dubbelriktad association med ett-till-ett förhållande mellan klasserna `ProduktStatus` och `Produkt`. Utläses:

- För varje `ProductStatus` finns det en och endast en `Product`.
- För varje `Product` finns det en och endast en `ProductStatus`.

Modell 1a: Visio



ProduktStatus.java (kodskelett 1)

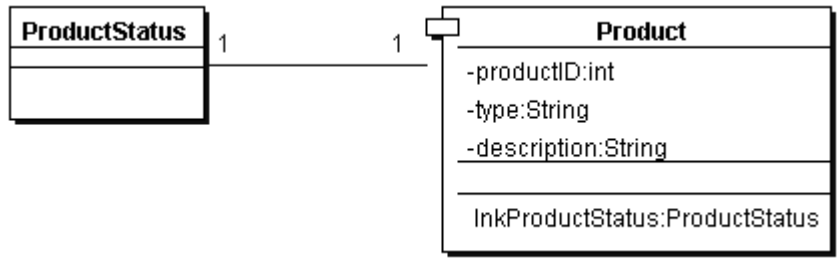
```
/* Static Model */
package Top_Package;


public class ProductStatus
{
   public ProductStatus()

   {
      super();
   }

}
/* END CLASS DEFINITION ProductStatus */
```

Product.java (kodskelett 2)

```
/* Static Model */
package Top_Package;


import Top_Package.ProductStatus;
public class Product
{
   public Product()

   {
      super();
   }
   public final ProductStatus getEnd1()
   {
      return this.mEnd1;
   }
   public final void setEnd1(ProductStatus the_mEnd1)
   {
      this.mEnd1 = the_mEnd1;
   }
```

```
   private int mproductID;

   private String mtype = null;

   private String mdescription = null;

   private ProductStatus mEnd1 = null;

}
/* END CLASS DEFINITION Product */
```

Modell 1b: Together



ProduktStatus.java (kodskelett 3)

```
/* Generated by Together */

public class ProductStatus {
}
```

Produkt.java (kodskelett 4)

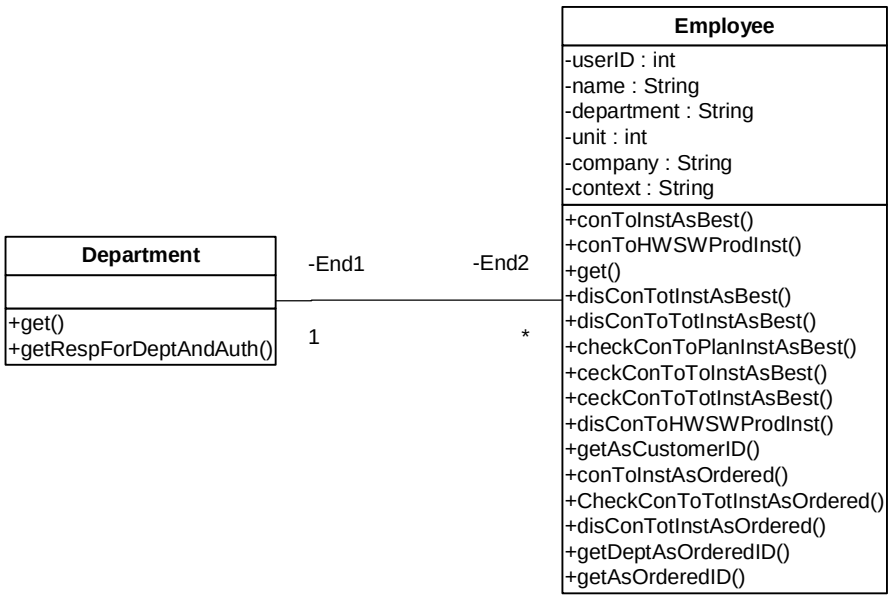
```
/* Generated by Together */

public class Product {
   public ProductStatus getLnkProductStatus() {
      return lnkProductStatus;
   }

   public void setLnkProductStatus(ProductStatus lnkProductStatus) {
      this.lnkProductStatus = lnkProductStatus;
   }

   private int productID;
   private String type;
   private String description;

   /**
    * @clientCardinality 1
    * @supplierCardinality 1
    */
   private ProductStatus lnkProductStatus;
}
```

D.2 - Testfall 2

Modellerna representerar en dubbelriktad association med ett-till-många förhållande mellan klasserna `Department` och `Employee`. Utläses:

- För varje `Department` finns det en eller flera `Employee`.
- För varje `Employee` finns det en och endast en `Department`.

Modell 2a: Visio



Department.java (kodskelett 5)

```
/* Static Model */
package Top_Package;


import Top_Package.Employee;
public class Department
{
  public Department()

  {
    super();
  }
  public final void get()
  {

  }
  public final void getRespForDeptAndAuth()
  {

  }
  public final java.util.Vector getEnd2()
  {
    return this.mEnd2;
```

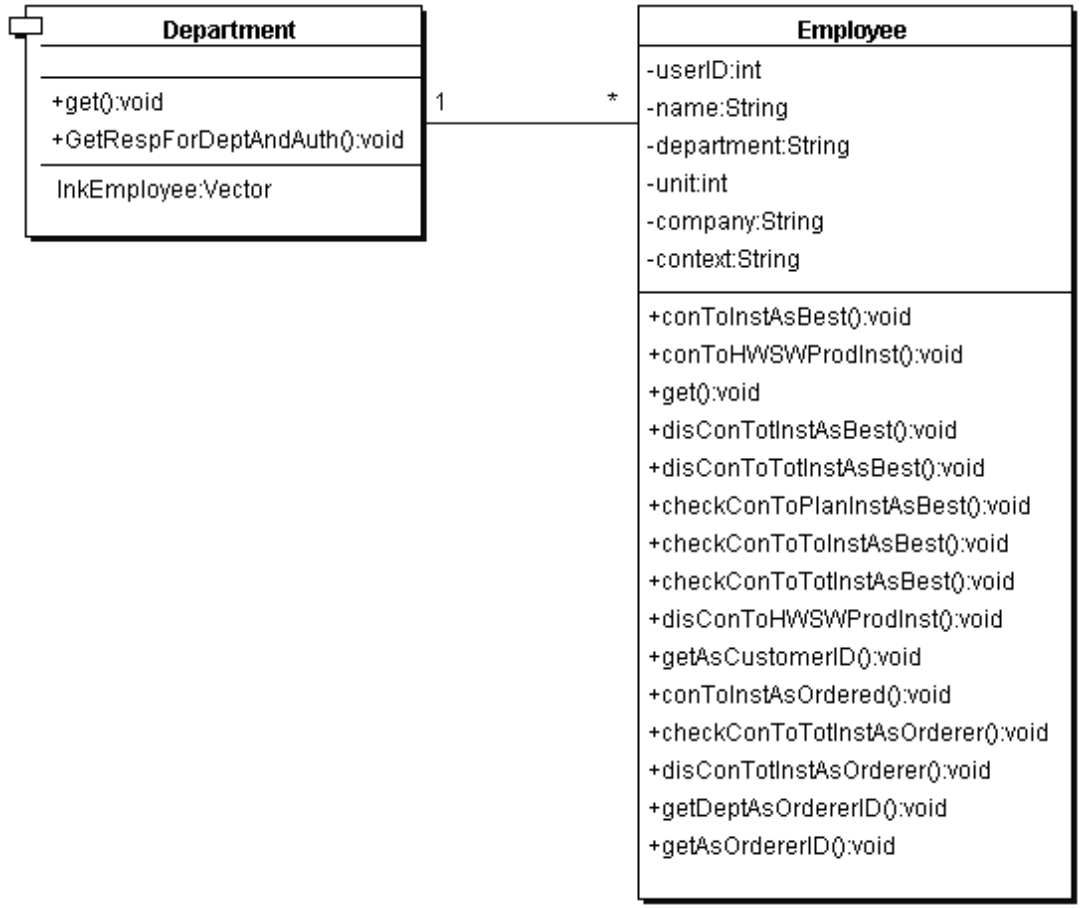
```
   }
   public final void setEnd2(java.util.Vector the_mEnd2)
   {
     this.mEnd2 = the_mEnd2;
   }

   private java.util.Vector mEnd2 = null;

}
/* END CLASS DEFINITION Department */
```

Employee.java (kodskelett 6)

```
/* Static Model */
package Top_Package;


public class Employee
{
   public Employee()

   {
     super();
   }
   public final void conToInstAsBest()
   {

   }
   public final void conToHWSWProdInst()
   {

   }
   public final void get()
   {

   }
   public final void disConTotInstAsBest()
   {

   }
   public final void disConToTotInstAsBest()
   {

   }
   public final void checkConToPlanInstAsBest()
   {

   }
   public final void ceckConToToInstAsBest()
   {

   }
   public final void ceckConToTotInstAsBest()
   {

   }
   public final void disConToHWSWProdInst()
   {

   }
```

```
    public final void getAsCustomerID()
    {

    }
    public final void conToInstAsOrdered()
    {

    }
    public final void CheckConToTotInstAsOrdered()
    {

    }
    public final void disConTotInstAsOrdered()
    {

    }
    public final void getDeptAsOrderedID()
    {

    }
    public final void getAsOrderedID()
    {

    }

    private int muserID;

    private String mname = null;

    private String mdepartment = null;

    private int munit;

    private String mcompany = null;

    private String mcontext = null;

}
/* END CLASS DEFINITION Employee */
```

Modell 2b: Together



Department.java (kodskelett 7)

```
/* Generated by Together */

import java.util.Vector;

public class Department {
   public void get() {
   }

   public void GetRespForDeptAndAuth() {
   }

   public Vector getLnkEmployee() {
      return lnkEmployee;
   }

   public void setLnkEmployee(Vector lnkEmployee) {
      this.lnkEmployee = lnkEmployee;
   }

   /**
    * @associates <{Employee}>
    * @clientCardinality 1
    * @supplierCardinality *
    */
   private Vector lnkEmployee;
}
```

Employee.java (kodskelett 8)

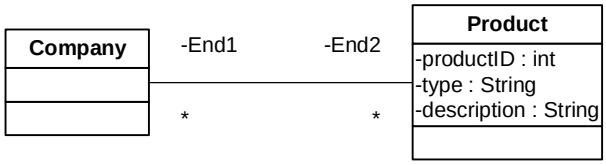
```
/* Generated by Together */

public class Employee {
   public void conToInstAsBest() {
   }

   public void conToHWSWProdInst() {
   }

   public void get() {
   }

   public void disConTotInstAsBest() {
   }

   public void disConToTotInstAsBest() {
   }

   public void checkConToPlanInstAsBest() {
   }

   public void checkConToToInstAsBest() {
   }

   public void checkConToTotInstAsBest() {
   }

   public void disConToHWSWProdInst() {
   }

   public void getAsCustomerID() {
   }

   public void conToInstAsOrdered() {
   }

   public void checkConToTotInstAsOrderer() {
   }

   public void disConTotInstAsOrderer() {
   }

   public void getDeptAsOrdererID() {
   }

   public void getAsOrdererID() {
   }

   private int userID;
   private String name;
   private String department;
   private int unit;
   private String company;
   private String context;
}
```

D.3 - Testfall 3

Modellerna representerar en dubbelriktad association med många-till-många förhållande mellan klasserna `Company` och `Product`. Utläses:

- För varje `Company` finns det en eller flera `Product`.
- För varje `Product` finns det en eller flera `Company`.

Modell 3a: Visio



Company.java (kodskelett 9)

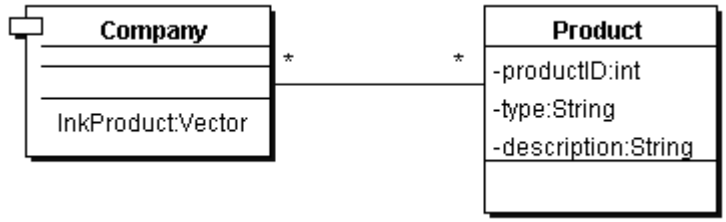
```
/* Static Model */
package Top_Package;


import Top_Package.Product;
public class Company
{
  public Company()

  {
    super();
  }
  public final java.util.Vector getEnd2()
  {
    return this.mEnd2;
  }
  public final void setEnd2(java.util.Vector the_mEnd2)
  {
    this.mEnd2 = the_mEnd2;
  }

  private java.util.Vector mEnd2 = null;

}
/* END CLASS DEFINITION Company */
```

Product.java (kodskelett 10)

```
/* Static Model */
package Top_Package;


import Top_Package.Company;
public class Product
{
  public Product()
```

```
   {
     super();
   }
   public final java.util.Vector getEnd1()
   {
     return this.mEnd1;
   }
   public final void setEnd1(java.util.Vector the_mEnd1)
   {
     this.mEnd1 = the_mEnd1;
   }

   private int mproductID;

   private String mtype = null;

   private String mdescription = null;

   private java.util.Vector mEnd1 = null;

}
/* END CLASS DEFINITION Product */
```

Modell 3b: Together



Company.java (kodskelett 11)

```
/* Generated by Together */

import java.util.Vector;

public class Company {
   public Vector getLnkProduct1() {
     return lnkProduct1;
   }

   public void setLnkProduct1(Vector lnkProduct1) {
     this.lnkProduct1 = lnkProduct1;
   }

   /**
    * @associates <{Product}>
    * @clientCardinality *
    * @supplierCardinality *
    */
   private Vector lnkProduct1;
}
```

Product.java (kodskelett 12)

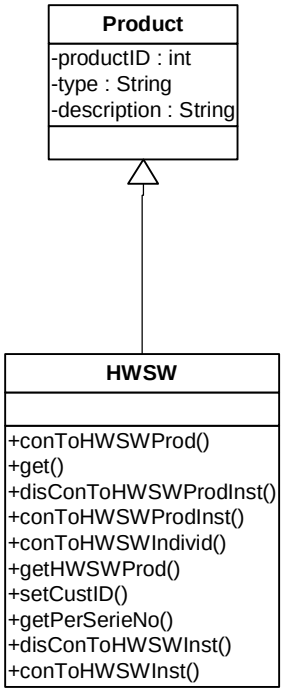
```
/* Generated by Together */

import java.util.Vector;

public class Product {
  private int productID;
  private String type;
  private String description;
}
```

D.4 - Testfall 4

Modellerna representerar en arvsrelation mellan klasserna `Product` och `HWSW`. Utläses:

- `Product` är en superklass till `HWSW`.
- `HWSW` är en arvinge (subklass) till `Product`.

Modell 4a: Visio



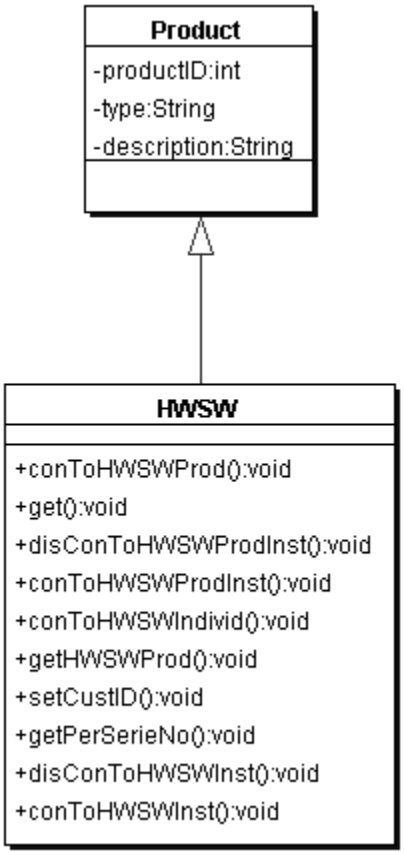
Product.java (kodskelett 13)

```
/* Static Model */
package Top_Package;


public class Product
{
   public Product()

   {
      super();
   }

   private int mproductID;

   private String mtype = null;

   private String mdescription = null;

}
/* END CLASS DEFINITION Product */
```

HWSW.java (kodskelett 14)

```
/* Static Model */
package Top_Package;


import Top_Package.Product;
public class HWSW extends Product
{
   public HWSW()

   {
      super();
   }
   public final void conToHWSWProd()
   {

   }
   public final void get()
   {

   }
   public final void disConToHWSWProdInst()
   {

   }
   public final void conToHWSWProdInst()
   {

   }
   public final void conToHWSWIndivid()
   {

   }
   public final void getHWSWProd()
   {

   }
   public final void setCustID()
   {

   }
   public final void getPerSerieNo()
   {

   }
   public final void disConToHWSWInst()
   {

   }
   public final void conToHWSWInst()
   {

   }

}
/* END CLASS DEFINITION HWSW */
```

Modell 4b: Together



Product.java (kodskelett 15)

```
/* Generated by Together */

public class Product {
   private int productID;
   private String type;
   private String description;
}
```

HWSW.java (kodskelett 16)

```
/* Generated by Together */

public class HWSW extends Product {
   public void conToHWSWProd() {
   }

   public void get() {
   }

   public void disConToHWSWProdInst() {
   }

   public void conToHWSWProdInst() {
   }
```

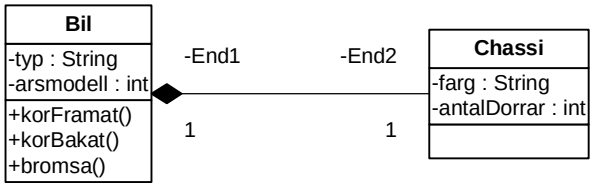
```
    public void conToHWSWIndivid() {
    }

    public void getHWSWProd() {
    }

    public void setCustID() {
    }

    public void getPerSerieNo() {
    }

    public void disConToHWSWInst() {
    }

    public void conToHWSWInst() {
    }
}
```

D.5 - Testfall 5

Modellerna representerar en aggregatsrelation i kompositionsform mellan klasserna `Bil` och `Chassi`. Utläses:

- En `Bil` har ett `Chassi`.
- Ett `Chassi` är en delmängd av (tillhör) en `Bil`.

Modell 5a: Visio



Bil.java (kodskelett 17)

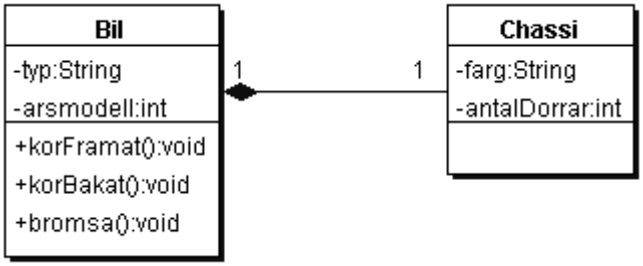
```
/* Static Model */
package Top_Package;


import Top_Package.Chassi;
public class Bil
{
  public Bil()

  {
    super();
  }
  public final void korFramat()
  {

  }
  public final void korBakat()
  {

  }
  public final void bromsa()
  {

  }
  public final Chassi getEnd2()
  {
    return this.mEnd2;
  }
  public final void setEnd2(Chassi the_mEnd2)
  {
    this.mEnd2 = the_mEnd2;
  }

  private String mtyp = null;

  private int marsmodell;

  private Chassi mEnd2 = new Chassi;
```

```
}
/* END CLASS DEFINITION Bil */
```

Chassi.java (kodskelett 18)

```
/* Static Model */
package Top_Package;


public class Chassi
{
   public Chassi()

   {
      super();
   }

   private String mfarg = null;

   private int mantalDorrar;

}
/* END CLASS DEFINITION Chassi */
```

Modell 5b: Together



Bil.java (kodskelett 19)

```
/* Generated by Together */

public class Bil {
   public void korFramat() {
   }

   public void korBakat() {
   }

   public void bromsa() {
   }

   private String typ;
   private int arsmodell;

   /**
    * @link aggregationByValue
    * @supplierCardinality 1
```

```
    * @clientCardinality 1
    */
   private Chassi lnkChassi;
}
```

Chassi.java (kodskelett 20)

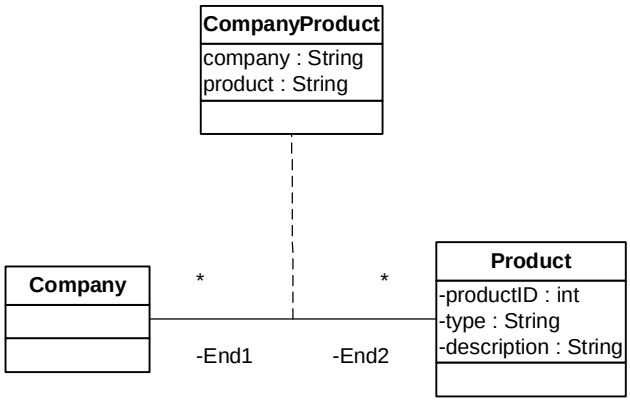
```
/* Generated by Together */

public class Chassi {
   private String farg;
   private int antalDorrar;
}
```

D.6 - Testfall 6

Modellerna representerar en dubbelriktad association med många-till-många förhållande mellan klasserna `Company` och `Product` med associationsklassen `CompanyProduct`. Utläses:

- För varje `Company` finns det en eller flera `Product`.
- För varje `Product` finns det en eller flera `Company`.

Modell 6a: Visio



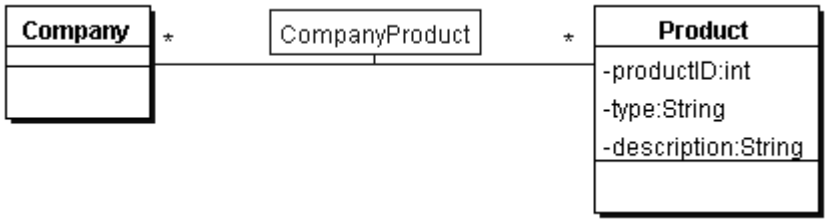
Company.java (kodskelett 21)

```
/* Static Model */
package Top_Package;


import Top_Package.Product;
public class Company
{
  public Company()

  {
    super();
  }
  public final java.util.Vector getEnd2()
  {
    return this.mEnd2;
  }
  public final void setEnd2(java.util.Vector the_mEnd2)
  {
    this.mEnd2 = the_mEnd2;
  }

  private java.util.Vector mEnd2 = null;

}
/* END CLASS DEFINITION Company */
```

Product.java (kodskelett 22)

```
/* Static Model */
package Top_Package;


import Top_Package.Company;
public class Product
{
   public Product()

   {
      super();
   }
   public final java.util.Vector getEnd1()
   {
      return this.mEnd1;
   }
   public final void setEnd1(java.util.Vector the_mEnd1)
   {
      this.mEnd1 = the_mEnd1;
   }

   private int mproductID;

   private String mtype = null;

   private String mdescription = null;

   private java.util.Vector mEnd1 = null;

}
/* END CLASS DEFINITION Product */
```

Modell 6b: Together



Company.java (kodskelett 23)

```
/* Generated by Together */

import java.util.Vector;

public class Company {
   /**
    * @associates <{Product}>
    * @associationAsClass CompanyProduct
    * @clientCardinality *
    * @supplierCardinality *
    */
   private Vector lnkProduct;
```

```
}
```

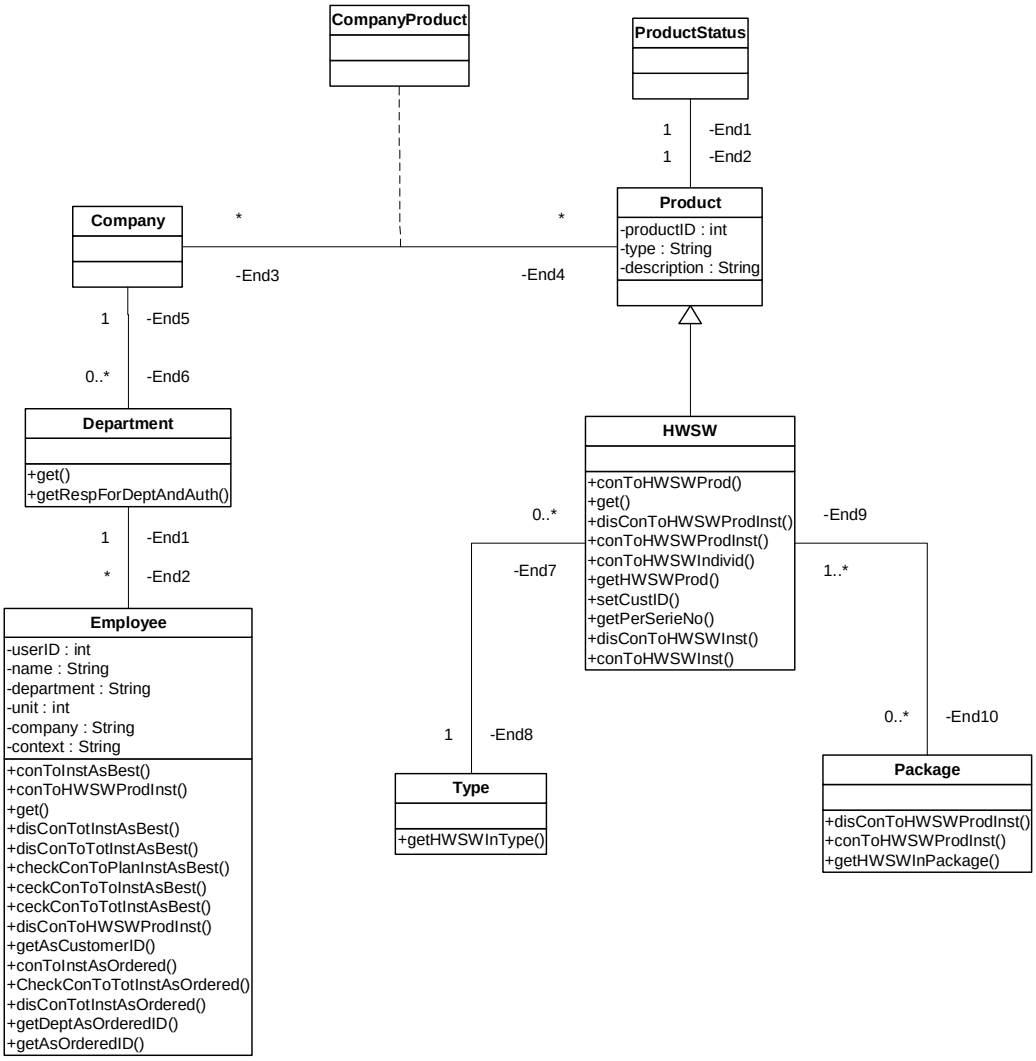
Product.java (kodskelett 24)

```
/* Generated by Together */

public class Product {
   private int productID;
   private String type;
   private String description;
}
```

D.7 - Testfall 7

Modell från Volvo IT (Seifzadeh, 2000: 100).

Modell 7a: Visio



Company.java (kodskelett 25)

```
/* Static Model */
package Top_Package;


import Top_Package.Department;
import Top_Package.Product;
public class Company
{
  public Company()

  {
    super();
  }
  public final java.util.Vector getEnd6()
  {
    return this.mEnd6;
```

```
   }
   public final void setEnd6(java.util.Vector the_mEnd6)
   {
     this.mEnd6 = the_mEnd6;
   }
   public final java.util.Vector getEnd4()
   {
     return this.mEnd4;
   }
   public final void setEnd4(java.util.Vector the_mEnd4)
   {
     this.mEnd4 = the_mEnd4;
   }

   private java.util.Vector mEnd6 = null;

   private java.util.Vector mEnd4 = null;

}
/* END CLASS DEFINITION Company */
```

Department.java (kodskelett 26)

```
/* Static Model */
package Top_Package;


import Top_Package.Employee;
public class Department
{
   public Department()

   {
     super();
   }
   public final void get()
   {

   }
   public final void getRespForDeptAndAuth()
   {

   }
   public final java.util.Vector getEnd2()
   {
     return this.mEnd2;
   }
   public final void setEnd2(java.util.Vector the_mEnd2)
   {
     this.mEnd2 = the_mEnd2;
   }

   private java.util.Vector mEnd2 = null;

}
/* END CLASS DEFINITION Department */
```

Employee.java (kodskelett 27)

```
/* Static Model */
package Top_Package;


public class Employee
{
  public Employee()

  {
    super();
  }
  public final void conToInstAsBest()
  {

  }
  public final void conToHWSWProdInst()
  {

  }
  public final void get()
  {

  }
  public final void disConTotInstAsBest()
  {

  }
  public final void disConToTotInstAsBest()
  {

  }
  public final void checkConToPlanInstAsBest()
  {

  }
  public final void ceckConToToInstAsBest()
  {

  }
  public final void ceckConToTotInstAsBest()
  {

  }
  public final void disConToHWSWProdInst()
  {

  }
  public final void getAsCustomerID()
  {

  }
  public final void conToInstAsOrdered()
  {

  }
  public final void CheckConToTotInstAsOrdered()
  {

  }
  public final void disConTotInstAsOrdered()
  {
```

```
   }
   public final void getDeptAsOrderedID()
   {

   }
   public final void getAsOrderedID()
   {

   }

   private int muserID;

   private String mname = null;

   private String mdepartment = null;

   private int munit;

   private String mcompany = null;

   private String mcontext = null;

}
/* END CLASS DEFINITION Employee */
```

HWSW.java (kodskelett 28)

```
/* Static Model */
package Top_Package;


import Top_Package.Package;
import Top_Package.Product;
public class HWSW extends Product
{
   public HWSW()

   {
      super();
   }
   public final void conToHWSWProd()
   {

   }
   public final void get()
   {

   }
   public final void disConToHWSWProdInst()
   {

   }
   public final void conToHWSWProdInst()
   {

   }
   public final void conToHWSWIndivid()
   {

   }
```

```
  public final void getHWSWProd()
  {

  }
  public final void setCustID()
  {

  }
  public final void getPerSerieNo()
  {

  }
  public final void disConToHWSWInst()
  {

  }
  public final void conToHWSWInst()
  {

  }
  public final java.util.Vector getEnd10()
  {
    return this.mEnd10;
  }
  public final void setEnd10(java.util.Vector the_mEnd10)
  {
    this.mEnd10 = the_mEnd10;
  }

  private java.util.Vector mEnd10 = null;

}
/* END CLASS DEFINITION HWSW */
```

Package.java (kodskelett 29)

```
/* Static Model */
package Top_Package;


import Top_Package.HWSW;
public class Package
{
  public Package()

  {
    super();
  }
  public final void disConToHWSWProdInst()
  {

  }
  public final void conToHWSWProdInst()
  {

  }
  public final void getHWSWInPackage()
  {

  }
```

```
  public final java.util.Vector getEnd9()
  {
    return this.mEnd9;
  }
  public final void setEnd9(java.util.Vector the_mEnd9)
  {
    this.mEnd9 = the_mEnd9;
  }

  private java.util.Vector mEnd9 = null;

}
/* END CLASS DEFINITION Package */
```

Product.java (kodskelett 30)

```
/* Static Model */
package Top_Package;


import Top_Package.Company;
import Top_Package.ProductStatus;
public class Product
{
  public Product()

  {
    super();
  }
  public final ProductStatus getEnd1()
  {
    return this.mEnd1;
  }
  public final void setEnd1(ProductStatus the_mEnd1)
  {
    this.mEnd1 = the_mEnd1;
  }
  public final java.util.Vector getEnd3()
  {
    return this.mEnd3;
  }
  public final void setEnd3(java.util.Vector the_mEnd3)
  {
    this.mEnd3 = the_mEnd3;
  }

  private int mproductID;

  private String mtype = null;

  private String mdescription = null;

  private ProductStatus mEnd1 = null;

  private java.util.Vector mEnd3 = null;

}
/* END CLASS DEFINITION Product */
```

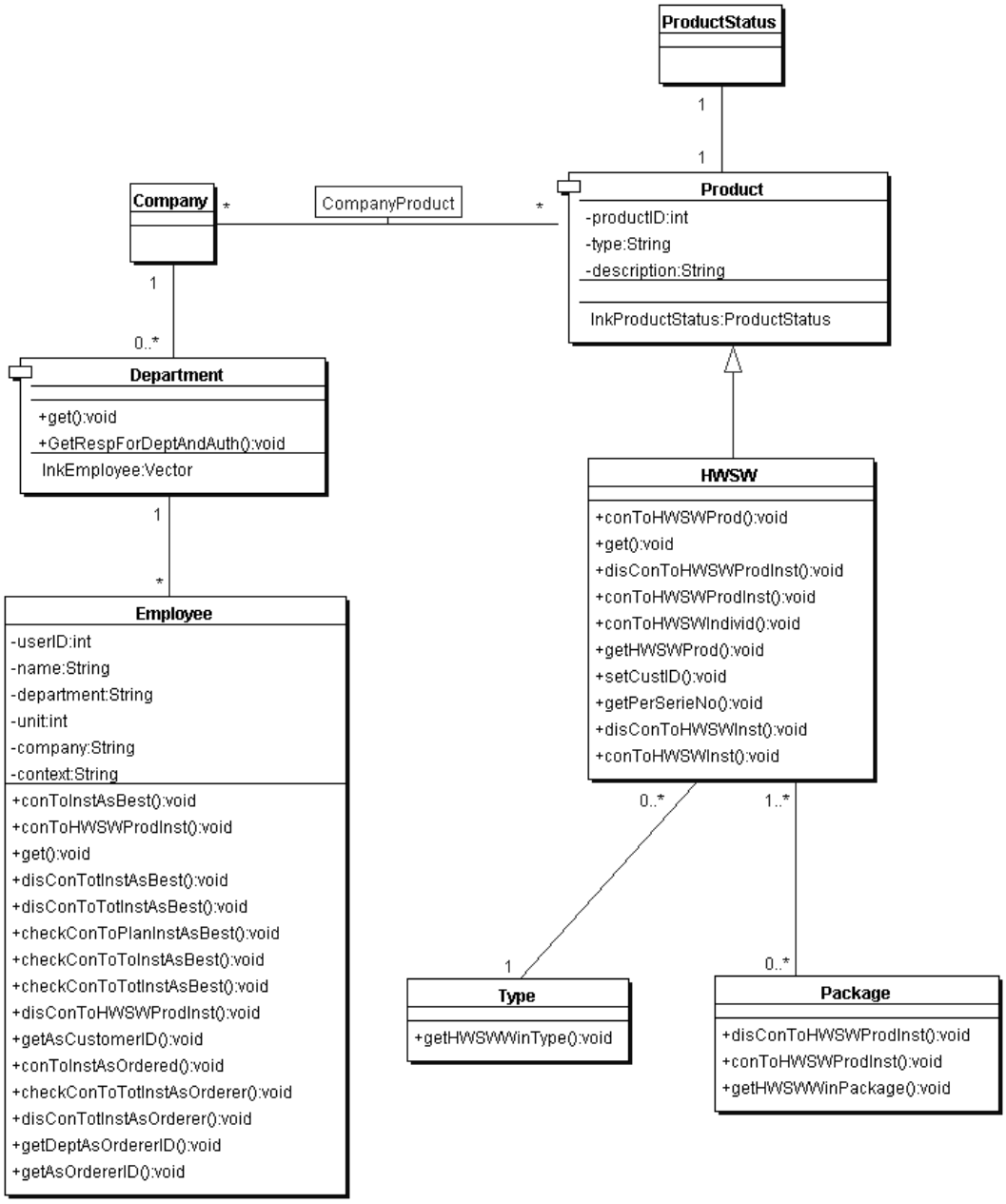
ProductStatus.java (kodskelett 31)

```
/* Static Model */
package Top_Package;


public class ProductStatus
{
   public ProductStatus()

   {
      super();
   }

}
/* END CLASS DEFINITION ProductStatus */
```

Type.java (kodskelett 32)

```
/* Static Model */
package Top_Package;


import Top_Package.HWSW;
public class Type
{
   public Type()

   {
      super();
   }
   public final void getHWSWInType()
   {

   }
   public final java.util.Vector getEnd7()
   {
      return this.mEnd7;
   }
   public final void setEnd7(java.util.Vector the_mEnd7)
   {
      this.mEnd7 = the_mEnd7;
   }

   private java.util.Vector mEnd7 = null;

}
/* END CLASS DEFINITION Type */
```

Modell 7b: Together



Company.java (kodskelett 33)

```
/* Generated by Together */

import java.util.Vector;

public class Company {
  /**
   * @associates <{Product}>
   * @clientCardinality *
   * @supplierCardinality *
   * @associationAsClass CompanyProduct
   */
  private Vector lnkProduct;
```

```
   /**
    * @associates <{Department}>
    * @supplierCardinality 0..*
    * @clientCardinality 1
    */
   private Vector lnkDepartment;
}
```

Department.java (kodskelett 34)

```
/* Generated by Together */

import java.util.Vector;

public class Department {
   public void get() {
   }

   public void GetRespForDeptAndAuth() {
   }

   public Vector getLnkEmployee() {
      return lnkEmployee;
   }

   public void setLnkEmployee(Vector lnkEmployee) {
      this.lnkEmployee = lnkEmployee;
   }

   /**
    * @associates <{Employee}>
    * @clientCardinality 1
    * @supplierCardinality *
    */
   private Vector lnkEmployee;
}
```

Employee.java (kodskelett 35)

```
/* Generated by Together */

public class Employee {
   public void conToInstAsBest() {
   }

   public void conToHWSWProdInst() {
   }

   public void get() {
   }

   public void disConTotInstAsBest() {
   }

   public void disConToTotInstAsBest() {
   }
```

```
  public void checkConToPlanInstAsBest() {
  }

  public void checkConToToInstAsBest() {
  }

  public void checkConToTotInstAsBest() {
  }

  public void disConToHWSWProdInst() {
  }

  public void getAsCustomerID() {
  }

  public void conToInstAsOrdered() {
  }

  public void checkConToTotInstAsOrderer() {
  }

  public void disConTotInstAsOrderer() {
  }

  public void getDeptAsOrdererID() {
  }

  public void getAsOrdererID() {
  }

  private int userID;
  private String name;
  private String department;
  private int unit;
  private String company;
  private String context;
}
```

HWSW.java (kodskelett 36)

```
/* Generated by Together */

import java.util.Vector;

public class HWSW extends Product {
  public void conToHWSWProd() {
  }

  public void get() {
  }

  public void disConToHWSWProdInst() {
  }

  public void conToHWSWProdInst() {
  }

  public void conToHWSWIndivid() {
  }
```

```
   public void getHWSWProd() {
   }

   public void setCustID() {
   }

   public void getPerSerieNo() {
   }

   public void disConToHWSWInst() {
   }

   public void conToHWSWInst() {
   }

   /**
    * @associates <{Package}>
    * @clientCardinality 1..*
    * @supplierCardinality 0..*
    */
   private Vector lnkPackage;
}
```

Package.java (kodskelett 37)

```
/* Generated by Together */

public class Package {
   public void disConToHWSWProdInst() {
   }

   public void conToHWSWProdInst() {
   }

   public void getHWSWWinPackage() {
   }
}
```

Product.java (kodskelett 38)

```
/* Generated by Together */

public class Product {
   public ProductStatus getLnkProductStatus() {
      return lnkProductStatus;
   }

   public void setLnkProductStatus(ProductStatus lnkProductStatus) {
      this.lnkProductStatus = lnkProductStatus;
   }

   private int productID;
   private String type;
   private String description;

   /**
    * @supplierCardinality 1
```

```
    * @clientCardinality 1
    */
   private ProductStatus lnkProductStatus;
}
```

ProductStatus.java (kodskelett 39)

```
/* Generated by Together */

public class ProductStatus {
}
```

Type.java (kodskelett 40)

```
/* Generated by Together */

public class Type {
   public void getHWSWWinType() {
   }

   /**
    * @clientCardinality 1
    * @supplierCardinality 0..*
    */
   private HWSW lnkHWSW;
}
```

Appendix E: Reverse engineering

Följande testfall har använts för att undersöka Together ControlCenter 4.2 stöd för reverse engineering. Koden har skapats manuellt i editorn UltraEdit 8.00b (www.ultraedit.com). I varje testfall redovisas även modeller som har genererats av CASE-verktyget från kod som specificerats i detta appendix. De modeller som skapats av verktyget från kodskelett genererade av Visio redovisas även under varje testfall.

E.1 - Testfall 8

Följande kod representerar en association med ett-till-ett förhållande mellan klasserna `ProductStatus` och `Product`.

Product.java (kod 1)

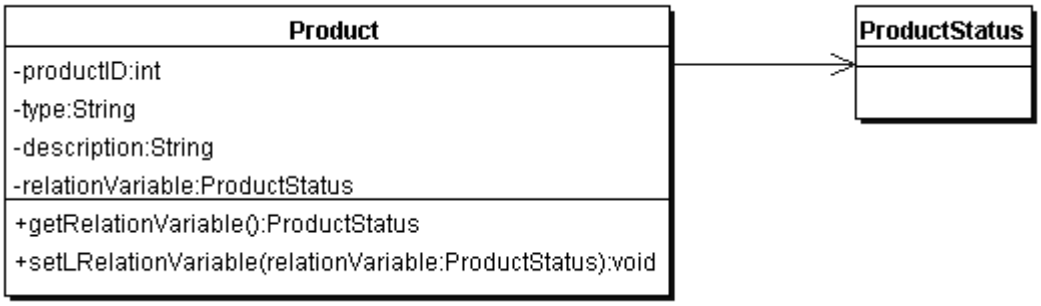
```
class Product {
  private int productID;
  private String type;
  private String description;
  private ProductStatus relationVariable;

  public ProductStatus getRelationVariable() {
    return relationVariable;
  }

  public void setLRelationVariable(ProductStatus relationVariable) {
    this.relationVariable = relationVariable;
  }
}
```

Product.java (kod 2)

```
class ProductStatus {}
```

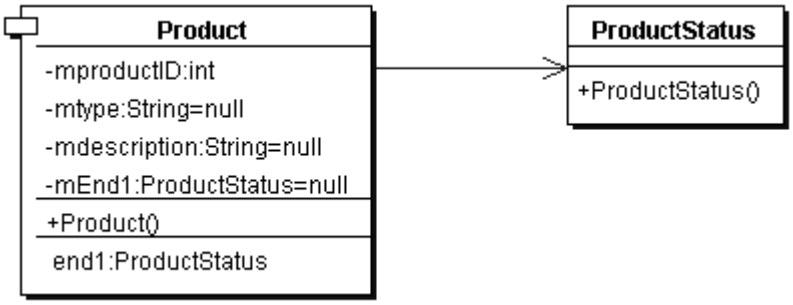
Modell E.1.1

Modell som Together skapat utifrån ovanstående kod.



Modell E.1.2

Modell som Together skapat utifrån kod genererad av Visio i testfall 1 (Appendix D.1).



E.2 - Testfall 9

Följande kod representerar en association med ett-till-många förhållande mellan klasserna `Department` och `Employee`.

Department.java (kod 3)

```
import java.util.Vector;

class Department {
  private Vector relationVariable;

  public Vector getRelationVariable() {
    return relationVariable;
  }

  public void setRelationVariable(Vector relationVariable) {
    this.relationVariable = relationVariable;
  }

  public void get() {}

  public void GetRespForDeptAndAuth() {}
}
```

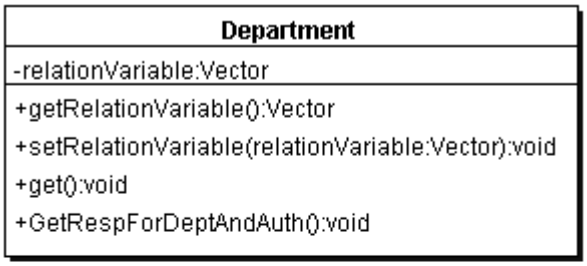
Employee.java (kod 4)

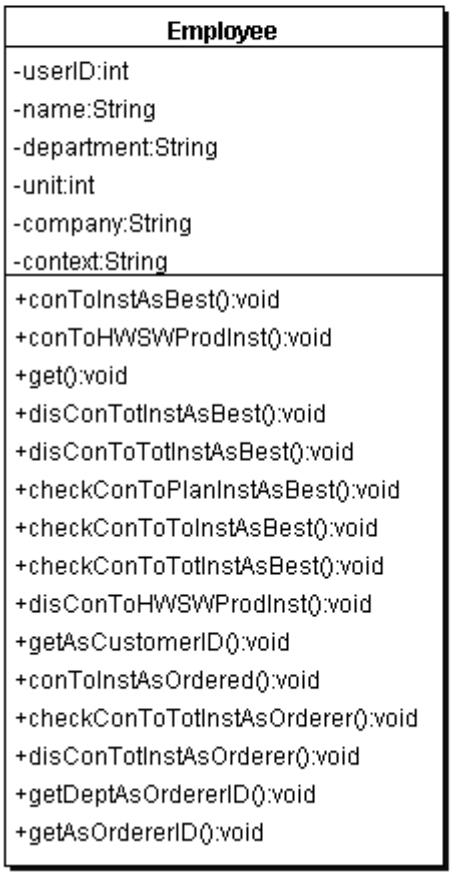
```
class Employee {
  private int userID;
  private String name;
  private String department;
  private int unit;
  private String company;
  private String context;

  public void conToInstAsBest() {}

  public void conToHWSWProdInst() {}

  public void get() {}

  public void disConTotInstAsBest() {}

  public void disConToTotInstAsBest() {}

  public void checkConToPlanInstAsBest() {}

  public void checkConToToInstAsBest() {}

  public void checkConToTotInstAsBest() {}

  public void disConToHWSWProdInst() {}

  public void getAsCustomerID() {}

  public void conToInstAsOrdered() {}
```

```
  public void checkConToTotInstAsOrderer() {}

  public void disConTotInstAsOrderer() {}

  public void getDeptAsOrdererID() {}

  public void getAsOrdererID() {}
}
```

Modell E.2.1

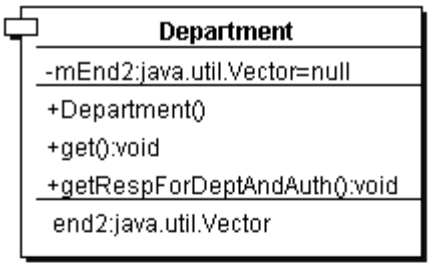
Modell som Together skapat utifrån ovanstående kod.

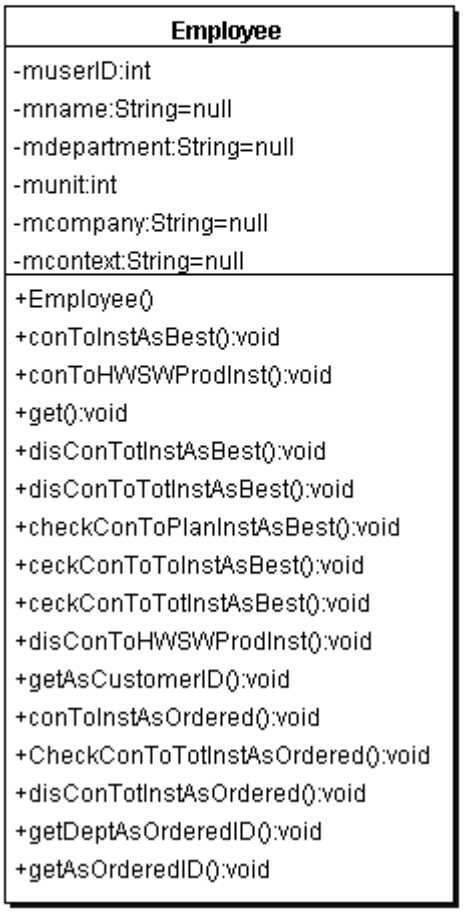




Modell E.2.2

Modell som Together skapat utifrån kod genererad av Visio i testfall 2 (Appendix D.2).





E.3 - Testfall 10

Följande kod representerar en association med många-till-många förhållande mellan klasserna `Company` och `Product`.

Company.java (kod 5)

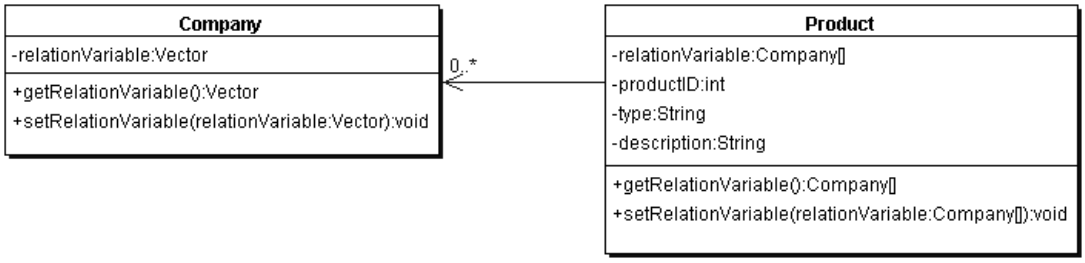
```
import java.util.Vector;

class Company {
  private Vector relationVariable;

  public Vector getRelationVariable() {
    return relationVariable;
  }

  public void setRelationVariable(Vector relationVariable) {
    this.relationVariable = relationVariable;
  }
}
```

Product.java (kod 6)

```
class Product {
    private Company[] relationVariable;
  private int productID;
  private String type;
  private String description;

    public Company[] getRelationVariable() {
     return relationVariable;
  }

  public void setRelationVariable(Company[] relationVariable) {
    this.relationVariable = relationVariable;
  }

}
```

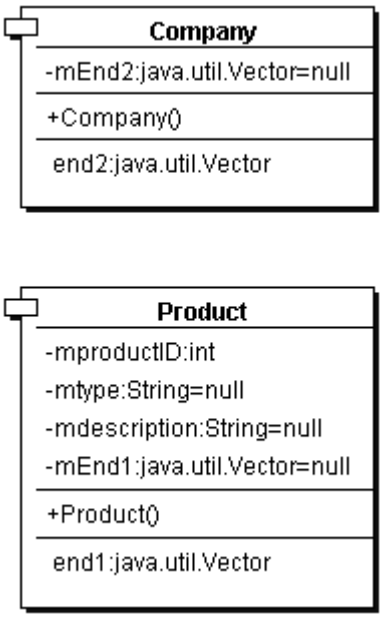
Modell E.3.1

Modell som Together skapat utifrån ovanstående kod.



Modell E.3.2

Modell som Together skapat utifrån kod genererad av Visio i testfall 3 (Appendix D.3).



E.4 - Testfall 11

Följande kod representerar ett arv mellan klasserna `HWSW` och `Product`, där `HWSW` är subklass till `Product`.

Product.java (kod 7)

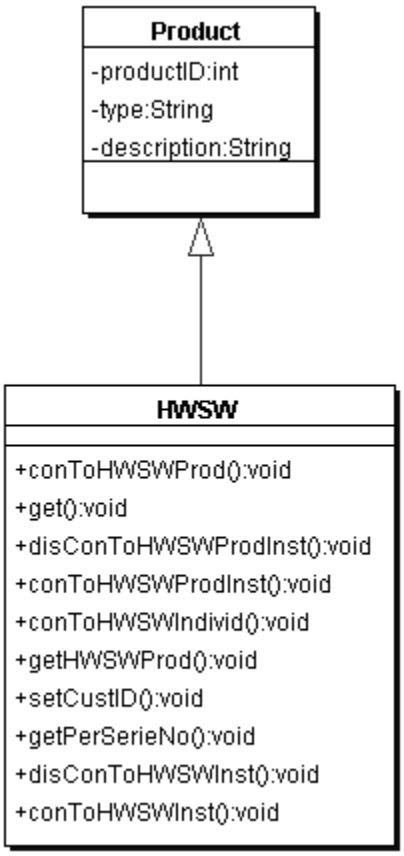
```
class Product {
  private int productID;
  private String type;
  private String description;
}
```

HWSW.java (kod 8)

```
class HWSW extends Product {
  public void conToHWSWProd() {}

  public void get() {}

  public void disConToHWSWProdInst() {}

  public void conToHWSWProdInst() {}

  public void conToHWSWIndivid() {}

  public void getHWSWProd() {}

  public void setCustID() {}

  public void getPerSerieNo() {}

  public void disConToHWSWInst() {}

  public void conToHWSWInst() {}
}
```

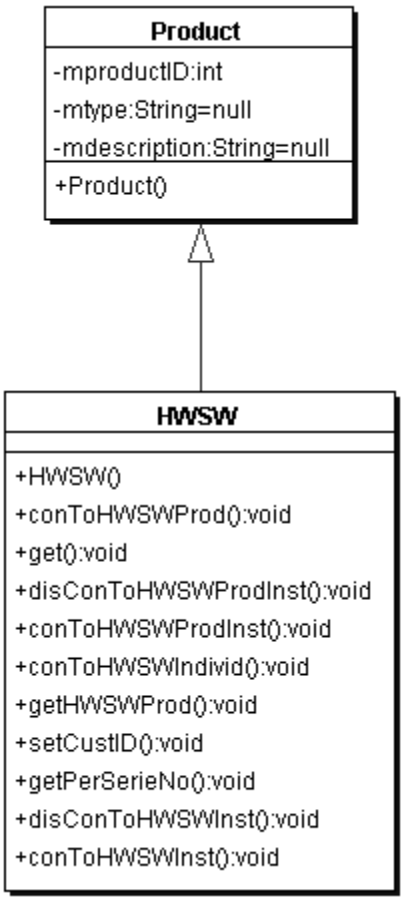
Modell E.4.1

Modell som Together skapat utifrån ovanstående kod.



Modell E.4.2

Modell som Together skapat utifrån kod genererad av Visio i testfall 4 (Appendix D.4).



E.5 - Testfall 12

Följande kod representerar en aggregatrelation i kompositionsform mellan klasserna `Bil` och `Chassi` som beskriver att ett chassi är en del av en bil.

Bil.java (kod 9)

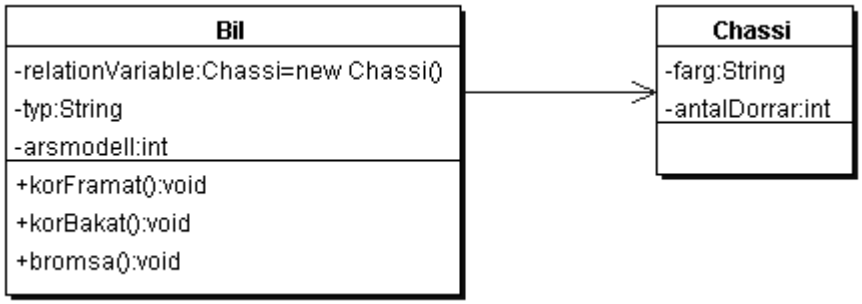
```
class Bil {
  private Chassi relationVariable = new Chassi();
  private String typ;
  private int arsmodell;

  public void korFramat() {}

  public void korBakat() {}

  public void bromsa() {}
}
```

Chassi.java (kod 10)

```
class Chassi {
  private String farg;
  private int antalDorrar;
}
```

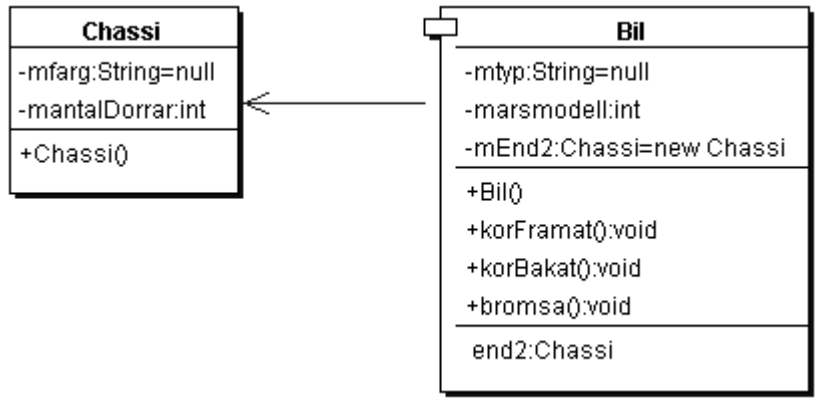
Modell E.5.1

Modell som Together skapat utifrån ovanstående kod.



Modell E.5.2

Modell som Together skapat utifrån kod genererad av Visio i testfall 5 (Appendix D.5).



E.6 - Testfall 13

Följande kod representerar en dubbelriktad association mellan två klasser (`Company` och `Product`), med en tillhörande associationsklass till relationen (`CompanyProduct`).

Company.java (kod 11)

```
class Company {
  private CompanyProduct[] relationVariable;
}
```

Product.java (kod 12)

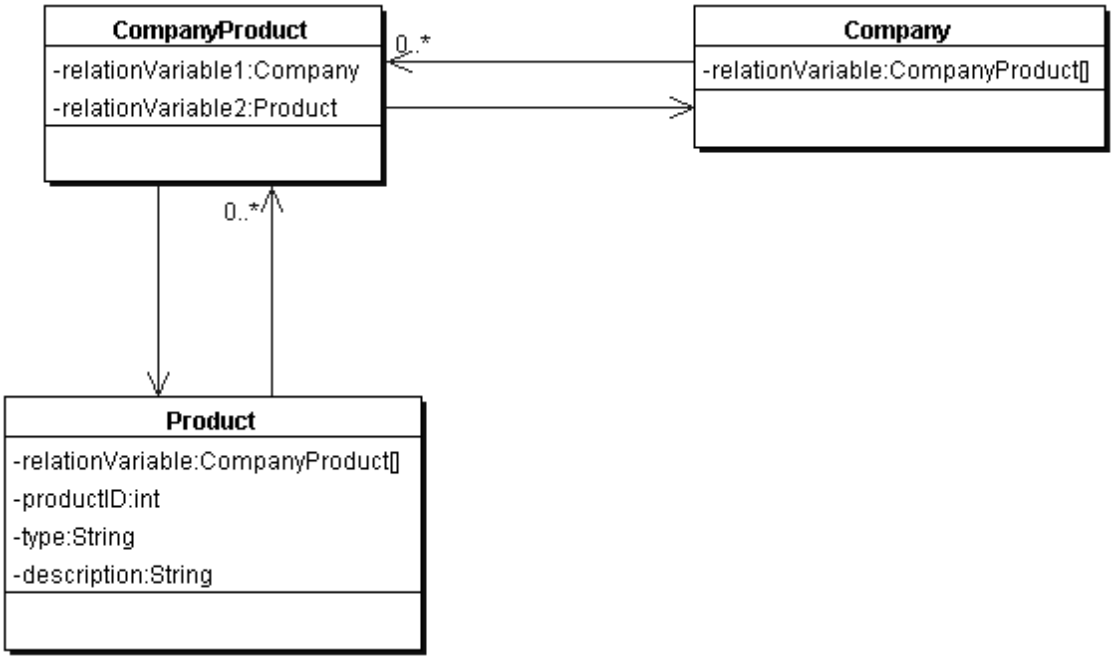
```
class Product {
  private CompanyProduct[] relationVariable;
  private int productID;
  private String type;
  private String description;
}
```

CompanyProduct.java (kod 13)

```
class CompanyProduct {
  private Company relationVariable1;
  private Product relationVariable2;
}
```

Modell E.6.1

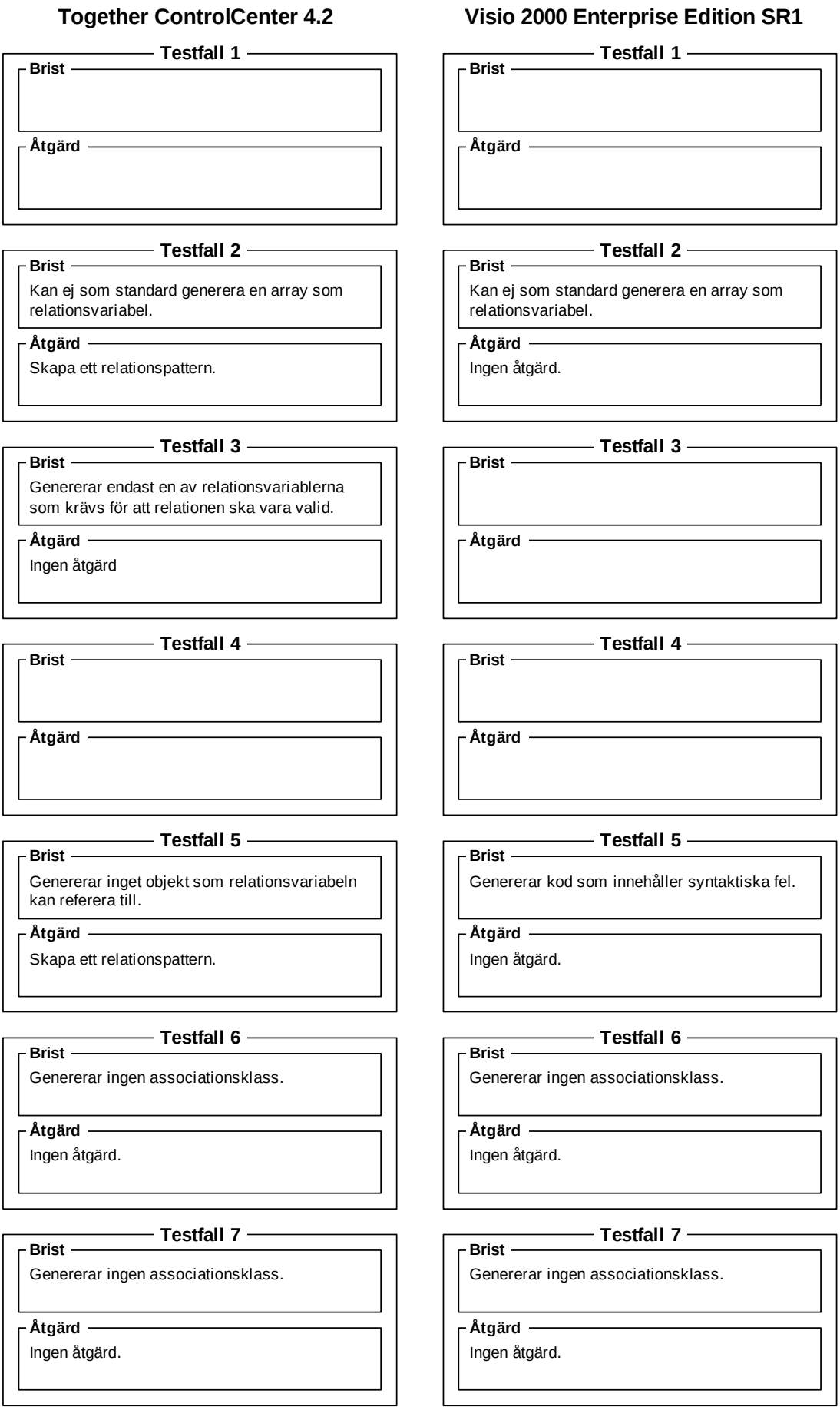
Modell som Together skapat utifrån ovanstående kod.



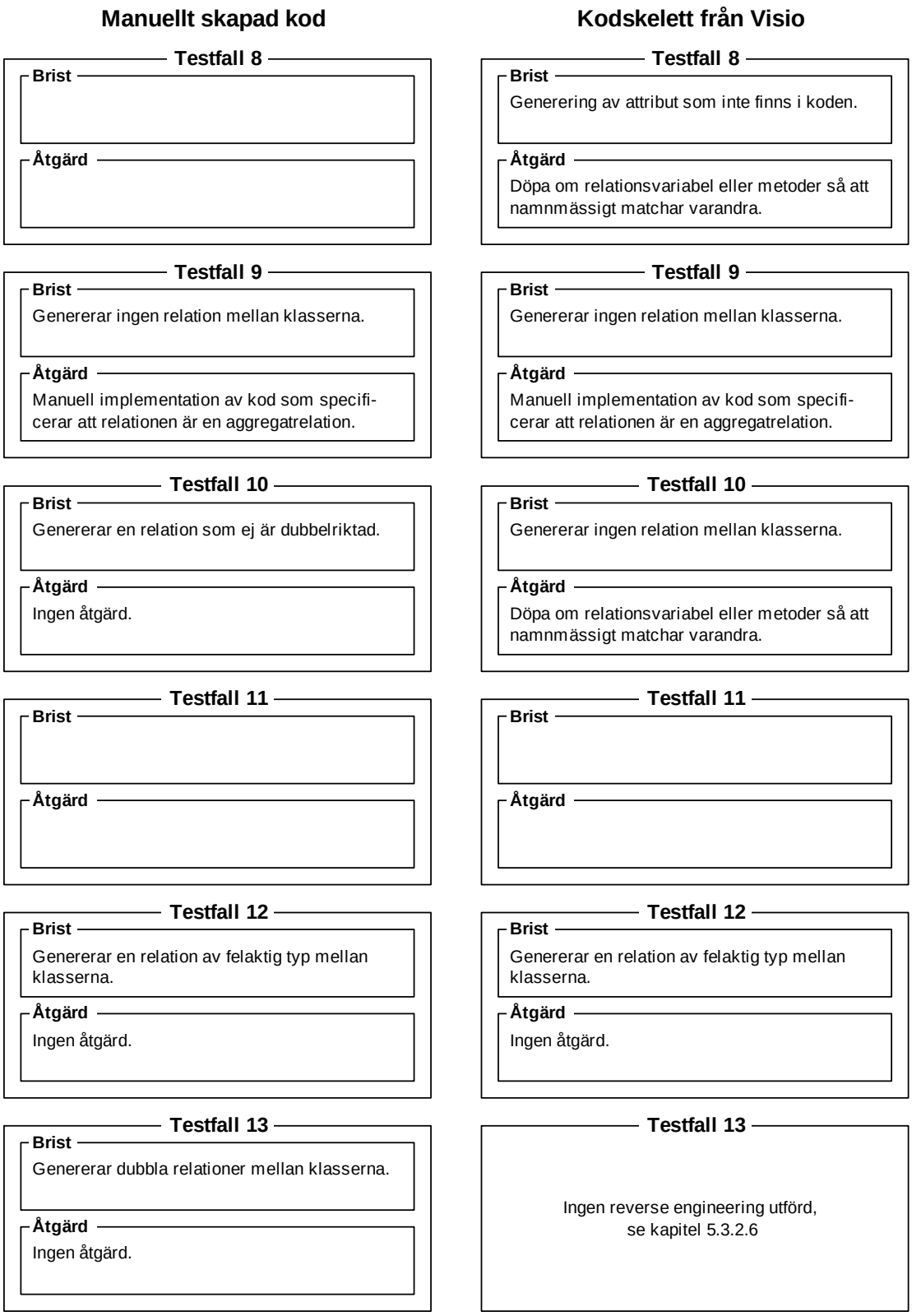
Appendix F: Sammanställning av testfallen

I detta appendix sammanställs resultaten från de testfall som utförts. För varje testfall beskrivs vilka brister som funnits samt eventuella åtgärder som utförts för att komma runt dessa brister.

Figur F.1 illustrerar testfallen från kodgenereringsdelen och figur F.2 illustrerar testfallen från reverse engineeringsdelen.



Figur F.1 *Sammanställning av kodgenereringsresultatet i Together och Visio.*



Figur F.2 *Sammanställning av reverse engineeringsresultat i Together*.