OOAD – Objektorienterad analys och design

1. Inledning

1.1 Om detta kompendium

Texten i detta kompendium har legat till grund för kapitel 10, om objektorienterad systemutveckling, i boken Objektorienterad programmering och Java (Per Holm, Studentlitteratur). Texten här är dock mera omfattande och utförlig än texten i det kapitlet.

1.2 Analys, design och implementation

Vid utveckling av stora programsystem kan man inte bara läsa igenom kravspecifikationen och sedan skriva programmet. Om man försöker tillämpa den ”metoden” vid programutveckling kommer man att råka i stora svårigheter. Vägen från kravspecifikation till färdigt program är svår, speciellt som man har som krav att program inte bara skall fungera utan också vara lätta att förstå för att senare modifieringar skall underlättas.

Man brukar dela in arbetet med att utveckla ett program i fyra faser: analys, design, implementation och testning. I analysen är den främsta uppgiften att förstå och beskriva det bakomliggande problemet, något som kan vara nog så svårt när det är en komplicerad uppgift som skall lösas. Bara att sätta sig in i problemområdet är normalt en omfattande uppgift. Observera att det i analysen är *problemet* som skall beskrivas, inte datorlösningen till problemet.

I designen utgår man från de erfarenheter som man skaffat sig under analysen och bygger vidare på dem. Här börjar man tänka på att programmet skall implementeras på dator och hur det skall byggas upp i stora drag. Man bestämmer vilken hårdvara som behövs för att köra programmet, hur kommunikation med andra datorer skall ske, hur data skall lagras i databaser eller på filer, vilka datastrukturer och algoritmer som programmet skall använda, osv.

Implementationen är själva programskrivningen. Om man har gjort en ordentlig analys så har man förstått problemet, om man har gjort en ordentlig design så vet man nu i grova drag hur man skall lösa det. Att skriva programmet borde alltså vid det här laget vara en överkomlig uppgift, men också implementation är svårt – det är inte bara ”kodning” utan också en hel del kvalificerat arbete. Dessutom visar det sig ofta att brister i analys och design inte blir påtagliga förrän man kommer till implementationen. Man kan då behöva gå tillbaka och göra om en del av det man gjort under analysen och designen.

En fjärde viktig fas i systemutvecklingen är testning av programmet. Man testar naturligtvis programkomponenter allteftersom man anser att de är färdiga, men det behövs också tester av det färdiga programmet. Man bör dock ha i minnet att man med testning bara kan påvisa fel i programmet, inte bevisa att det är korrekt.

1.3 Traditionella metoder för systemutveckling

Det finns ett stort antal metoder för systemutveckling som har använts under lång tid. Dessa traditionella metoder har det gemensamt att de ser de tre faserna i systemutvecklingen som avgränsade delar: man börjar med analysen, när den är klar så vidtar designen och inte förrän den är klar så påbörjar man implementationen. Man brukar kalla sådana metoder för ”vattenfallsmetoder”. Metoder av det slaget tar inte hänsyn till att man lär sig mer om systemet under utvecklingstiden och att man kan behöva gå tillbaka till en tidigare fas och göra om en del arbete. Dessutom är det i traditionella metoder inte ovanligt att övergången mellan faserna är svår att göra. Det kan vara olika team som är ansvariga för de olika faserna och det kan till och med vara så att de olika teamen använder olika metoder i sitt arbete och att metoderna inte passar ihop.

Ett ännu viktigare kännetecken på de traditionella systemutvecklingsmetoderna är att de är *funktionsinriktade*. Detta är i viss mån en naturlig följd av att kravspecifikationer oftast skrivs på ett funktionsinriktat sätt: ”Utveckla ett system för X, som klarar av att göra Y1, Y2, …, YN”. Y1, Y2, … är alltså de funktioner som programmet skall ha. I en funktionsinriktad systemutveckling koncentrerar man både analys och design på funktionerna, som bryts ner i mindre och mindre (och mer och mer lätthanterliga) delfunktioner. I implementationen kommer funktionerna att motsvaras av metoder.

Erfarenheten har visat att en systemutveckling som görs på detta sätt leder till program som är svåra att modifiera och underhålla. Om en ny funktion skall läggas till eller en existerande funktion skall ändras så medför det ofta genomgripande ändringar i programmet.

1.4 Objektorienterade metoder

I objektorienterade metoder för systemutveckling utgår man inte från programmets tilltänkta funktioner. I stället utgår man från de *data* som skall hanteras av programmet och identifierar objekt som är ansvariga för handhavandet av data. På objekten kan man utföra operationer för att modifiera data, men det är objekten själva som är ansvariga för *hur* modifieringen skall göras. I stället för ett stort antal underprogram som alla arbetar på samma mängd av data har man objekt som vart och ett ansvarar för sina egna data. Under exekvering av programmet samarbetar objekten genom att de ber varandra om att få operationer utförda. För att lägga till en ny funktion i systemet behöver man oftast bara lägga till en ny operation i ett objekt, eller åtminstone bara ändra ett fåtal objekt.

Tyngdpunkten i all objektorienterad systemutveckling ligger på *modellering*. Systemutvecklingens uppgift är att skapa en modell av det ”verkliga” systemet. Byggstenarna i modellen är objekt och klasser. Modellen skall vara gemensam för utvecklingens alla faser. Den första analysmodellen kommer att förfinas under designen, och man kommer att förfina designmodellen under implementationen, men man kan under hela processen känna igen den modell som man arbetar med.

Man brukar karakterisera objektorienterade systemutvecklingsmetoder som iterativa och inkrementella, i motsats till de traditionella vattenfallsmetoderna. Att en metod är iterativ innebär att det är fullt tillåtet och till och med rekommendabelt att man går tillbaka och modifierar det man tidigare har utvecklat. Att metoden är inkrementell innebär man oftast gör sådana modifikationer genom tillägg till det tidigare. När man använder en objektorienterad metod är det dessutom lätt att göra ändringar: att lägga till en ny operation på ett objekt är enkelt, att lägga till nya objekt (klasser) är lika enkelt. Det räcker dock inte att bara lägga till nya operationer och klasser: de nya operationerna och klasserna skall också utnyttjas någonstans i programmet (operationerna skall utföras, objekt av klasserna skall skapas). Även sådana förändringar är enklare att göra i objektorienterade metoder än i de traditionella metoderna.

En annan indelning av systemutvecklingsmetoder är att karakterisera metoder som ”top-down” eller ”bottom-up”. Top-down innebär att man börjar från helheten och efterhand bryter ner

problemet i mindre och mindre delar. Bottom-up innebär att man först försöker hitta lösningar till delproblem, som man sedan sätter samman till en lösning av hela problemet. Mycket grovt kan man säga att traditionella metoder är top-down-metoder, objektorienterade metoder bottom-up-metoder. I en objektorienterad metod börjar man ju med att identifiera objekt och klasser och inte förrän man gjort det bekymrar man sig över hur objekten skall samarbeta. Dock arbetar man även i objektorienterade metoder top-down i vissa fall.

En fördel med objektorienterade metoder som ofta framhålls är att de kan leda till *återanvändbara* programkomponenter, dvs klasser som senare kan utnyttjas i andra program. Generellt användbara klassbibliotek är målet. När det gäller objektorienterad programmering har man kommit en bit på vägen; det finns klassbibliotek i flera programspråk t ex Java och C++. Att återanvända analys- och designresultat är ett svårare problem. Under de senaste åren har dock forskningen om ”design patterns” [Gam95, Bus96, Gra98] visat på lösningar till även detta problem. Ett designmönster är en lösning till ett ofta förekommande designproblem.

1.5 Metodernas uppbyggnad

En metod för systemutveckling har två komponenter: dels en notation, som talar om hur de storheter som man hanterar skall betecknas, dels en metodik, som talar om vilka steg man skall gå igenom för att få det önskade resultatet. Man kan jämföra detta med matematiken: när man lär sig integrera lär man sig dels en passande notation för integraler, dels metodiken för hur man bär sig åt för att beräkna integralerna.

I analys och design är notationen både grafisk och textmässig. Man kan t ex använda rektanglar för att beteckna klasser, linjer och pilar för att åskådliggöra relationer mellan klasser och mellan objekt. Man behöver naturligtvis också kunna namnge de storheter man hanterar; det gör man med vanlig text.

Eftersom slutmålet för utvecklingen är att producera ett program verkar det kanske onödigt att gå omvägen över en separat notation för analys och design, när man ändå till sist måste översätta alltihop till programkod. Man kan i och för sig tänka sig att utnyttja ett programspråk även under analys- och designarbetet. Detta är dock inte lämpligt, eftersom programspråken innehåller alldeles för många detaljer för att vara användbara på denna höga abstraktionsnivå. Det som främst krävs vid analys och design är överblick, inte detaljer.

Notationen i en systemutvecklingsmetod är viktig. Jämför återigen med matematiken: man skulle inte komma långt utan en notation i vilken man kan uttrycka sina beräkningar. Av en notation kräver man att man skall kunna uttrycka allt som man behandlar i metoden (klasser, objekt, relationer mellan klasser och mellan objekt, mm) på ett koncist sätt. Man måste också kunna hantera de använda symbolerna på ett effektivt sätt. Vid utveckling av små system kan man nöja sig med att rita på papper eller med ett vanligt ritprogram. Stora system kräver ett datoriserat verktyg som är specialutvecklat för metoden (om man i metoden representerar klasser med rektanglar och relationer mellan klasser med linjer mellan rektanglarna, måste t ex linjerna följa med när man flyttar en rektangel). Sådana verktyg brukar kallas CASE-verktyg (CASE = Computer Aided Software Engineering). Avancerade CASE-verktyg kan också generera ”programskelett” utgående från designmodellen.

Det allra mest väsentliga i en systemutvecklingsmetod är dock inte notationen, utan metodiken. Detta borde egentligen vara självklart, men det finns en hel del exempel på metoder som nästan enbart består av notation som understöds av ett CASE-verktyg. Metodiken bör inte bara bestå av ett antal goda råd utan av klara och tydliga instruktioner. Allra bäst vore det om man när man följde instruktionerna garanterat kom fram till ett färdigt program som uppfyllde alla krav som man kan ställa. Så är det dock tyvärr inte i någon metodik, utan sunt förnuft, kreativitet och framför allt erfarenhet krävs också. Detta gäller ännu mer i analys och design än i programmering på lägre nivå.

1.6 Några etablerade metoder för OOAD

Objektorienterad analys och design är ett förhållandevis nytt område. De specifika problem som man försöker lösa, t ex att hitta lämpliga klasser i systemet, har man dock sysselsatt sig med sedan man började med objektorienterad programmering i slutet av 1960-talet. I de moderna metoderna har man formaliserat lösningsmetoderna i lägre eller högre grad, och man bygger framför allt på en stor mängd erfarenhet av hur man löser problem av detta slag.

Utvecklingen av metoder påbörjades i början av 1990-talet. Varje metod introducerade sin egen notation och gav mer eller mindre handfasta regler för systemutvecklingen. En av de första metoderna var Responsiblity-Driven Design, RDD, utvecklad av Rebecca Wirfs-Brock [Wir90]. Metoden är lättfattlig och enkel, och boken är fortfarande en bra introduktion till objektorienterad systemutveckling.

De metoder som fick störst praktisk och kommersiell användning var Object Modeling Technique, OMT, av James Rumbaugh [Rum91] och Object-Oriented Design av Grady Booch [Boo94]. Båda dessa metoder är innehållsrika och tar hänsyn till många olika aspekter på systemutvecklingen. Särskilt OMT har också en väldefinierad metodik.

Bland de första metoderna kan också nämnas Peter Coads och Edward Yourdons Object-Oriented Analysis/Design [Coa91a, Coa91b]. Upphovsmännen hade tidigare utarbetat icke-objektorienterade metoder för systemutveckling. Coad-Yourdons metod är dock för enkel för att den skall kunna utnyttjas i stora projekt.

Ivar Jacobson (svensk) utvecklade under sin tid vid Ericsson en systemutvecklingsmetod som senare fick namnet Objectory. Denna metod skalades ner och beskrevs i boken Object-Oriented Software Engineering [Jac92].

De ”tre stora” inom OOAD-området (Booch, Rumbaugh och Jacobson) arbetar sedan 1995 i samma företag, Rational Corporation. De har tillsammans utvecklat en notation med namnet UML, Unified Modeling Language (introduktioner finns t ex i [Eri98, Poo99], en mera utförlig beskrivning i [Boo98]), som antagligen kommer att ersätta den rika flora av notationer som tidigare funnits.

Jacobson, Booch och Rumbaugh har också varit delaktiga i framtagandet av en systemutvecklingsmetod som nu har namnet Rational Unified Process [Jac99]. Denna metod tar hänsyn inte bara till programutveckling utan också till hela utvecklingsprocessen med t ex kravspecifikation och testning. Iterativ utveckling betonas i metoden.

2. Registerkort – en enkel metod för OOAD

2.1 Registerkort – notation

I detta kapitel skall vi som inledning till studiet av objektorienterade metoder för systemutveckling närmare beskriva en enkel metod för OOAD. Metoden är nog för enkel för att kunna användas i praktiken men kan fungera bra på små och enkla problem. Dessutom innehåller metoden många av de moment som återkommer i andra, mer avancerade metoder. Av dessa liknar metoden mest Rebecca Wirfs-Brocks metod Responsibility-Driven Design, RDD.

I metoden används en mycket enkel notation. Man hanterar bara klasser, och man dokumenterar varje klass på ett registerkort. I litteraturen kallas korten CRC-kort, där CRC står för Class-Responsibility-Collaborators [Bec89]. Fördelen med registerkort är att de är enkla att hantera och att man lätt kan ändra det man skriver på korten. Relationer mellan klasser kan man åskådliggöra antingen genom att man skriver namnen på de berörda klasserna på korten eller t ex genom

att man tejpar upp korten på en tavla och ritar linjer mellan korten. Allt i denna metod är mycket informellt och några fasta regler för hur korten skall se ut finns inte.

Man kan till exempel använda registerkort av följande utseende:

Registerkort – framsida

Klassnamn	Superklass
Anvarsområde Vad klassen beskriver och en översiktlig beskrivning av klassens uppgifter.	
Samarbetar med Relationerna till andra klasser.	

På framsidan av kortet noterar man alltså klassens namn och namnet på dess superklass (om den har någon). Dessutom ger man en kortfattad beskrivning av klassens ansvarsområde och av klassens relationer till andra klasser.

Baksidan på registerkortet innehåller en mera detaljerad beskrivning av klassen, med en förteckning över operationer och attribut:

Registerkort – baksida

Operationer En förteckning över operationerna, på lämplig detaljnivå. Till att börja med kanske man bara anger operationernas namn, senare kan man också ange parametrar.
Attribut En förteckning över attributen. Även denna kan man göra mer eller mindre noggrann.

På ett kort skall man alltså notera det som man tycker är väsentligt att komma ihåg om motsvarande klass. Det bör understrykas att man inte fyller i hela kortet på en gång: till att börja med kanske man bara noterar klassnamnet. I takt med att man under analysarbetet får mer kunskap om klassen lägger man till information på kortet. Det är naturligtvis också tillåtet att kasta bort kort, om man upptäcker att klassen inte behövs, och att radera och ändra information.

2.2 Registerkort – metodik

I den metodik som utnyttjas i registerkortsmetoden skiljer man inte så noga på analys och design. Dock är det fördelaktigt om man försöker skjuta designdetaljer framför sig så länge som möjligt.

Följande uppgifter ingår i metodiken:

1. Finn objekt.
2. Klassificera objekten dvs finn klasser.
3. Finn relationer mellan objekt och mellan klasser.
4. Gruppera klasser dvs dela upp systemet i delsystem.
5. Generera och gå igenom scenarier dvs följ exekveringen av viktiga användningsfall och kontrollera att ingenting saknas i modellen.
6. Specificera operationer och attribut.
7. Verifiera modellen.

Vi har redan påpekat att objektorienterad systemutveckling är en iterativ och inkrementell process. Man behöver alltså inte strikt följa den angivna ordningen, utan man kan mycket väl gå fram och tillbaka mellan momenten.

En mycket viktig uppgift vid OOAD är att finna lämpliga operationer och attribut till klasserna. Denna uppgift är utspridd över flera av momenten – man finner operationer både när man klassificerar objekt (moment 2) och när man går igenom scenarier (moment 5).

Man är i de flesta OOAD-metoder överens om att de uppgifter som beskrivs i de sju punkterna måste lösas. I de följande avsnitten går vi igenom punkterna i tur och ordning, förklarar vad de innebär och ger några anvisningar för hur man kan lösa uppgifterna.

2.2.1 Finn objekt

Uppgiften här är att finna de objekt som förekommer i det verkliga systemet och som också skall finnas i modellen. Utgångspunkten är kravspecifikationen och den allmänna kunskap man har om problemområdet. Om man inte har någon sådan kunskap måste man skaffa sig den, om inte annat för att man skall kunna läsa och förstå kravspecifikationen. Man studerar alltså det verkliga system som skall modelleras, läser böcker som behandlar området, talar med dem som skrivit kravspecifikationen och med tilltänkta användare, osv.

Att finna objekt är normalt ett jobb av ”brainstorm”-karaktär som det är fördelaktigt att utföra i grupp. Man diskuterar och föreslår objekt utan att ha några förutfattade meningar om hur modellen skall se ut. Inte förrän man har en lista över möjliga objekt börjar man fundera på vilka av dessa som verkligen skall ingå i modellen.

Man kan leta efter objekt bland:

- Fysiska storheter, t ex maskiner, lokaler, fordon, människor, datorer. Det är på detta slags objekt som man allra först bör koncentrera sin uppmärksamhet, av två anledningar: dels är det sådana objekt som alldeles säkert finns i det verkliga systemet, dels är sådana objekt oftast lätta att finna.
- Tänkta storheter, som kanske inte har någon fysisk existens, t ex fönster i ett fönstersystem, konton och transaktioner i ett banksystem. Dessa objekt kan vara väl så viktiga i systemet. Datastrukturer som man för in för att underlätta designen tillhör denna kategori av objekt.

En enkel metod att hitta objekt är att gå igenom kravspecifikationen och leta efter substantiv i texten. Metoden har fått utstå mycket kritik, men som en metod bland andra fungerar den bra. Den får dock användas med urskillning, eftersom kravspecifikationer ofta är mycket långa. Att gå igenom hundratals sidor text och mekaniskt leta efter substantiv är säkert inte givande.

När man har fått en lista över möjliga objekt gäller det att besluta om vilka av objekten som skall vara kvar i modellen. Man går igenom sin lista och kontrollerar för varje objekt följande punkter:

- Objektet skall beskriva någonting meningsfullt i modellen. Här får man tänka på att olika synsätt på en modell kan leda till att olika objekt kommer att vara relevanta.
- Objektet skall i normalfallet ha både tillstånd (som beskrivs av attribut) och operationer. Man bör kunna svara på frågan om vad ett objekt *är*, inte bara vad det gör. Man har dock ofta ett ”huvudprogramobjekt” med en enda operation vars enda uppgift är att skapa de andra objekten och sätta igång kommunikationen mellan dem. Även i andra fall kan det vara praktiskt med sådana ”kontrollobjekt”.
- Objektet skall beskriva en enda sak. Stora objekt med många uppgifter brukar man kunna dela upp i flera mindre objekt med väldefinierade uppgifter.
- Man bör inte omedelbart förkasta ”små” objekt, t ex ett objekt som bara är ansvarigt för att hantera ett enda talvärde. Det är möjligt att man senare, under implementationen, kommer att finna att man inte behöver något objekt för talet utan lika gärna kan implementera det som en vanlig variabel, men det är också möjligt att man kommer att bygga ut objektet med flera värden och flera operationer. Då kan det vara bra att ha kvar det som ett objekt.

2.2.2 Klassificera objekt

För att finna klasser går man igenom listan över objekt och bestämmer för varje objekt vilken klass den tillhör. Även om det bara finns ett enda objekt av ett visst slag så måste man definiera en klass för objektet. Man måste dock se till att man inte definierar olika klasser för objekt som är så lika så att de kan tillhöra samma klass men ha olika värden på attributen. (Här avses inte objekt som är lika i vissa avseenden men olika i andra. Sådana likheter beskriver man med ärvning, se avsnitt 2.2.3).

Man bör tänka efter noga när man namnger klasserna. Av namnet skall framgå vad klassen beskriver. Detta gör dels att modellen blir lättare att förstå, men också att det fortsatta arbetet underlättas. När man t ex skall definiera relationer mellan klasser (avsnitt 2.2.3) och upptäcka fall av ärvning, måste ju ”är-en”-relationen gälla. Om en klass då har ett olämpligt namn kan det hända att man aldrig upptäcker en sådan relation.

Om det är möjligt, bör man redan under analysen och designen definiera klasserna så att en framtida utbyggnad av programmet underlättas. Man kan alltså definiera klasser som är mer generellt användbara än vad det aktuella problemet kräver. Om man gör så, kan man också uppnå att klasserna blir möjliga att återanvända i andra program.

För att dokumentera en klass noterar man dess namn på ett registerkort. Vid det här laget bör man också kunna fylla i fältet ”Ansvarsområde” på kortet (se avsnitt 2.1). Det räcker om man skriver en kort mening om detta. Man kan på kortet också notera om objekt av klassen finns under hela programexekveringen eller om de skapas och försvinner efter en kort stund, samt hur många objekt av klassen som kan existera samtidigt.

Man kan också redan nu börja tänka på vilka operationer och attribut som klasserna skall ha. Om man utgår från verkliga objekt så är det lätt att skriva en lista över saker som man brukar göra med objekten. Man får dock komma ihåg vad som skall göras i den aktuella tillämpningen så att inte listan av operationer blir alltför omfattande. Genom att utgå från egenskaperna hos verkliga objekt kan man också definiera attribut i klasserna.

Beskrivningen av operationer och attribut behöver inte vara fullständig i detta skede. Till exempel kan man vänta med att specificera vilka parametrar som operationerna skall ha.

2.2.3 Finn relationer

Det finns två slags relationer mellan objekt:

Aggregering	Ett objekt består av (är sammansatt av) andra objekt. Typexemplet är en bil, som består av ett chassi, fyra hjul, en motor, osv. För att upptäcka relationen undersöker man om ”består-av”-relationen gäller.
Association	Ett objekt känner till ett annat objekt. Detta är en betydligt mera generell relation, eftersom objekt kan känna till varandra på många olika sätt. Här kan man testa relationer som ”har-en”, ”kommunicerar-med-en”, ”använder-en”, ”äger-en” osv. (Relationer som ”har-två”, ”äger-många”, osv förekommer naturligtvis också.) Exempel: en människa ”äger-en” bil, en dator ”kommunicerar-med” en annan dator.

I tidigare kurser har vi slagit samman dessa båda relationer till en: sammansättning, ”har-en”. För tydlighetens skull är det dock fördelaktigt att skilja mellan relationerna. Man kan också tänka sig ytterligare uppdelningar; i en del OOAD-metoder betraktas t ex ”använder” som ett eget slag av relation.

Mellan klasser finns det bara en möjlig relation:

Ärvning	En klass är en specialisering (subklass) av en annan klass (superklass). Eller omvänt, superklassen är en generalisering av subklassen. Relationen mellan subklass och superklass beskrivs av ”är-en”-relationen. Typexemplet är återigen en bil, som ”är-ett” fordon.

För att upptäcka att ärvning är den lämpliga relationen kan man givetvis testa ”är-en”-relationen mellan samtliga klasser i systemet. Vanligare är dock att man utnyttjar ”är-en”-relationen när man redan har upptäckt ärvningen, för att kontrollera att det verkligen är ärvning som det är frågan om. För att upptäcka ärvning brukar man i stället testa följande:

Specialisering	Man upptäcker att ett objekt av en viss klass är en mer speciell form av ett objekt av en annan klass (en bil är ett mer speciellt slags fordon).
Generalisering	Man upptäcker att man har flera liknande klasser som har en gemensam nämnare, en superklass (ett fordon är ett begrepp som fångar likheterna mellan bilar och bussar).

Man bör vara noga med att se till att ärvningen verkligen uttrycker en hierarkisk klassificering. I undantagsfall kan det dock vara befogat att utnyttja ärvning bara för att få tillgång till önskade egenskaper hos en superklass. Ett exempel på detta är när man utnyttjar ärvning för att ge ett objekt egenskapen ”kan lagras i en lista”.

Det kan mycket väl vara så att man definierar en klass utan att man har för avsikt att skapa objekt av klassen i fråga. Sådana klasser, som ligger på en hög nivå i den hierarkiska klass-strukturen och bara används som superklass till andra klasser, kallas abstrakta klasser. Det är ofta de abstrakta klasserna som är återanvändbara.

I vår registerkortsmodell åskådliggör vi relationerna på lämpligt sätt, t ex genom att notera relationerna på korten. Ett mer åskådligt sätt är att tejpa upp korten på en tavla och rita linjer mellan korten.

2.2.4 Gruppera klasser

Att gruppera klasser i ”delsystem” är oftast inte aktuellt i så små problem som registerkortsmetoden är avsedd för. I större system kan det dock vara nödvändigt att göra modellen mindre komplex genom att samla klasser som har ett gemensamt användningsområde till ett delsystem.

Ett enkelt exempel på uppdelning i delsystem är att man skiljer mellan applikationsspecifika klasser och klasser som är till för mer allmänna uppgifter, t ex hantering av kommunikationen med användaren (fönstersystem etc) och hantering av grundläggande datastrukturer.

I registerkortsmodellen kan man tänka sig att lägga korten i olika högar och ha en hög för varje delsystem. Mera givande är kanske att beskriva delsystemen med särskilda kort.

2.2.5 Generera och gå igenom scenarier

Ett scenario är en följd av händelser som inträffar i systemet som följd av en (yttre) påverkan. Ibland kallar man scenarier för användningsfall. Man skall tänka igenom vilka viktiga användningsfall som finns i systemet och definiera ett scenario för varje sådant fall.

Om man skall modellera ett system av bankomater och bankdatorer har man bl a scenarierna ”användaren tar ut pengar” och ”användaren begär kontouppgift”. I varje scenario går man igenom vilka händelser som inträffar: användaren stoppar in sitt bankomatkort i bankomaten, bankomaten frågar efter kod, användaren trycker in sin kod, bankomaten verifierar koden hos bankdatorn, bankomaten frågar efter hur mycket pengar som skall tas ut, användaren trycker in beloppet, bankomaten vidarebefordrar begäran om uttag till bankdatorn, osv.

Det kan vara svårt att veta hur detaljerat man skall beskriva scenarierna. Det gäller att lägga sig på en ”lagom” nivå, men avvägningen här är svår och beror på om man studerar hela systemet eller någon detalj i det. När man studerar hela bankomatsystemet är beskrivningen ovan nog lagom detaljerad; när man detaljstuderar själva bankomaten skall man kanske dela upp händelsen ”användaren trycker in sin kod” i flera delhändelser, en för varje siffertangent som användaren trycker ned.

I modellen kommer händelserna att motsvaras av operationer. (Bankomatsystemet är dock inte ett så bra exempel på detta, eftersom händelserna ibland måste skickas över ett nätverk. Till exempel måste ju bankomaten kontakta bankdatorn för att verifiera en kod.)

En stor svårighet i all programmering är att ta hand om de felsituationer som kan inträffa. Det är viktigt att man genererar scenarier också för sådana fall (”bankomatkortet fastnar i kortläsaren”, ”användaren trycker in en felaktig kod”, ”användaren har inte tillräckligt mycket pengar på sitt konto”, ”förbindelsen med bankdatorn kan inte kopplas upp”, osv).

Ett scenario kan beskrivas i text, som vi gjort i vårt exempel, men man kan också rita diagram av olika slag. I registerkortsmetoden nöjer vi oss med den enkla textbeskrivningen.

För varje scenario går man sedan igenom vad som händer i modellen när de olika händelserna i scenariot inträffar. Man kontrollerar att alla objekt som kommunicerar med varandra verkligen är knutna till varandra med relationer och att alla operationer som skall utföras är definierade.

Det är ofta givande att arbeta i grupp när man går igenom scenarierna. Man börjar med att fördela registerkorten mellan sig. Vid varje steg i ett scenario avgör man vilken klass som bör vara ansvarig för att motsvarande operation utförs, och kontrollerar om denna operation finns med på rätt registerkort.

Vid genomgången av scenarierna kommer man troligen att upptäcka många brister i modellen. Man kan behöva lägga till nya klasser, lägga till eller ändra relationer, lägga till eller ändra operationer.

2.2.6 Specificera operationer och attribut

När man med hjälp av genomgången av scenarierna börjar bli någotsånär säker på att den modell man utvecklar är vettig, om än inte fullständig, så är det dags att ägna sig åt detaljerna i klasserna. Man specificerar nu utförligt vilka parametrar som konstruktorerna och metoderna skall ha.

För varje operation bör man också kontrollera följande punkter:

- Se till att operationen utför en logiskt sammanhängande uppgift. Om det är en omfattande uppgift bör man undersöka om det inte går att dela upp operationen i flera mindre deloperationer.
- Var misstänksam om operationen har många parametrar. Det är ofta möjligt att dela upp också en sådan operation.
- Om klassen ingår i en klasshierarki: kontrollera så att operationen ligger i rätt klass. Kan operationen flyttas uppåt i hierarkin så att den blir mer generell? Gäller operationen verkligen för alla subklasser eller skall den flyttas nedåt i hierarkin?

På liknande sätt går man igenom attributen och bestämmer deras typer. Som tidigare nämnts bör man ifrågasätta om en klass som inte har några attribut verkligen skall vara en klass. Man kan också ha i minnet att klasser med väldigt många attribut ibland kan delas upp i flera klasser (naturligtvis under förutsättning att det är naturligt i modellen att göra så).

Här kan vi passa på att nämna att implementationen av relationer mellan objekt egentligen är ett kapitel för sig. I både aggregering och association skall ju ett objekt ha en relation till ett annat objekt. Vanligen implementerar man en sådan relation med en referensvariabel som refererar till det andra objektet. Det finns dock andra möjligheter.

På registerkorten noterar man de ändringar och utökningar som man har gjort.

2.2.7 Verifiera modellen

I detta moment skall man verifiera och dokumentera den modell som man kommit fram till. Vid verifikationen går man på nytt igenom samtliga scenarier som man tidigare definierat, nu på en mer detaljerad nivå. Återigen ägnar man särskild uppmärksamhet åt de scenarier som beskriver felsituationer, eftersom dessa ofta är de mest komplexa.

Om den modell man utvecklat är liten kan med nöja sig med registerkorten och den textmässiga beskrivningen av scenarierna som dokumentation av modellen. Om det är en stor modell behövs antagligen en mer formaliserad beskrivning.

När allt detta är klart kan man börja implementera modellen. Som vi redan tidigare har sagt så skall objektorienterad systemutveckling dock vara inkrementell, så det är inget fel om man redan tidigare har implementerat delar av modellen för att övertyga sig om att den fungerar. Om man tidigt programmerar en prototyp så lär man sig antagligen mycket om systemet. Man bör dock inte bli alltför bunden till prototypen utan vara beredd att kasta bort den om den visar sig alltför otymplig som grund för det fortsatta arbetet. Om man envist fortsätter att bygga vidare på prototypen så kan utvecklingsarbetet bli svårt i fortsättningen.

3. Registerkort – användning på ett exempel

3.1 Problemet

Detta enkla problem bygger på ett exempel i avsnitt 10.5 i boken Objektorienterad programmering och Java. Den modell som vi kommer att komma fram till liknar mycket den lösning som där presenteras. Problemet är följande:

Nim är ett spel i vilket två personer deltar. Spelet börjar med att ett antal stickor fördelas på tre högar av godtycklig storlek. De båda spelarna turas därefter om att ta en eller två stickor, efter eget val, från någon av högarna. Den som tar den sista stickan har vunnit spelet.

Skriv ett program där en spelomgång mellan en människa och datorn genomförs. ”Människospelaren” skriver sina drag på tangentbordet. ”Datorspelaren” väljer alltid den hög som innehåller minst antal stickor och tar två stickor ur den om högen innehåller mer än en sticka. Annars tar han den enda stickan.

För att hålla nere omfånget på texten visar vi inte registerkortens utseende under arbetets gång. Det man skriver på korten beskrivs i stället i texten.

3.2 Analys och design

3.2.1 Finn objekt

Kravspecifikationen är här kortfattad och tydlig, så det är inte svårt att sätta sig in i problemet. Genom att läsa specifikationen och leta efter substantiv får vi omedelbart en lista med möjliga objekt:

> Spel, person, antal, sticka, hög, storlek, spelare, val, program, spelomgång, människa, dator, människospelare, drag, tangentbord, datorspelare.

Många av dessa substantiv kommer inte att motsvaras av klasser i lösningen till problemet. Vi förkastar följande klasskandidater:

person	Samma som spelare, som är ett bättre namn
antal	Attribut (antal stickor i spelet)
storlek	Attribut hos högklassen (antal stickor i högen)
val	Substantiverat verb, något som utförs under spelet
program	Tillhör inte referenssystemet
spelomgång	Något som utförs
människa	Samma som människospelare
dator	Samma som datorspelare
drag	Något som utförs

3.2.2 Klassificera objekt

Varje objekt skall beskrivas av en klass. Vi inför följande klasser och beskriver kortfattat deras uppgifter (i fortsättningen använder vi engelska namn för alla storheter som vi inför):

NimGame	Själva spelet med stickhögarna, ”spelplanen”. Håller ordning på högarna, ser till att spelarna kan ta stickor osv. Bara ett objekt av denna klass skapas.
Pile	En stickhög (tre objekt). Håller ordning på en hög med stickor, ser till att man kan lägga dit och ta bort stickor.

Pin	En sticka (”ett antal” objekt). Utför egentligen ingenting, bara läggs i högar och tas ur högar.
Player	En spelare (två objekt). Spelar spelet enligt någon strategi, tar i varje drag stickor från en hög.
HumanPlayer	En människa (ett speciellt slags spelare).
ComputerPlayer	En dator (också ett speciellt slags spelare).
Keyboard	Tangentbordet, varifrån den mänsklige spelarens drag läses (ett objekt).

Som nämnts kan man redan i detta moment börja tänka på operationer och attribut. Även om man från beskrivningen ovan direkt kan se många lämpliga operationer och attribut (”lägga dit stickor”, ”ta bort stickor”, osv), så väntar vi med en fullständig beskrivning till senare (3.2.5).

3.2.3 Finn relationer

Vi börjar med den uppenbara klassrelationen att både människan och datorn ”är-en” spelare. Player skall alltså vara superklass till HumanPlayer och till ComputerPlayer och kommer att vara en abstrakt klass – vi kommer bara att skapa objekt av subklasserna.

Det är en onödig begränsning att en mänsklig spelare nödvändigtvis skall ge sina drag via ett tangentbord. I en mer utvecklad version av programmet kan man tänka sig att denna inmatning i stället görs med musen, t ex genom ett menyval. Det är enkelt att förbereda för detta redan nu, genom att vi inför en abstrakt klass Input, en ”allmän inenhet”, och låter Keyboard vara en subklass till denna. När vi vill föra in musen skriver vi en ny subklass till Input.

När det gäller relationer mellan objekten finns exempel både på aggregering och på association. Aggregeringsrelationer:

NimGame – Pile	Spelplanen ”består-av” tre högar.
Pile – Pin	En hög ”består-av” ett antal stickor.

Associationer:

Player – NimGame	En spelare ”väljer-en-hög” på spelplanen.
Player – Pile	En spelare ”tar-stickor” från en hög.
HumanPlayer – Input	En människa ”matar-in-drag” via inmatningsenheten.

Det kan ifrågasättas om associationen mellan Player och Pile behövs. Det är fördelaktigt om den kan tas bort, eftersom man då minskar antalet beroenden mellan klasserna. Vi diskuterar detta närmare i 3.2.5.

Här är det lätt att finna relationerna. I mer komplicerade problem, med många klasser, är det en betydligt svårare uppgift.

3.2.4 Gruppera klasser

Detta moment är inte meningsfullt här, eftersom man inte kan vinna något genom att gruppera klasserna i ett så här litet system.

3.2.5 Generera och gå igenom scenarier

Det viktigaste scenariot i detta problem är ”spela en omgång”. Det finns också specialfall (felsituationer), t ex att en hög är tom när spelaren vill ta en sticka eller att spelaren väljer en hög som inte finns. Vi utelämnar dock dessa scenarier i denna första analys. ”Spela en omgång” kan beskrivas enligt följande:

1. Fördela stickorna på högarna.
2. Den förste spelaren gör ett drag.
3. Den andre spelaren gör ett drag.
4. Punkterna 2 och 3 upprepas tills alla stickorna är slut.

Vi skall nu gå igenom punkterna och kontrollera att alla objekt som behövs finns i modellen. Samtidigt försöker vi tänka ut vilka operationer som behövs i objekten, utan att för tillfället bekymra oss om parametrarna.

1. Det måste vara NimGame-objektets uppgift att fördela stickorna, eftersom detta objekt har ansvaret för hela spelplanen. Operation på NimGame-objektet: setUpNewGame.
2. Operation på HumanPlayer- eller ComputerPlayer-objektet: makeMove. Eftersom alla slags spelare skall kunna göra drag placerar vi operationen i superklassen Player (abstrakt metod).
3. D:o.
4. För att ta reda på om stickorna är slut för vi in operationen morePins i klassen NimGame.

Dessa operationer är dock inte de enda som behövs. Vi måste gå igenom scenariot mer i detalj och undersöka om de operationer som vi infört behöver utnyttja andra operationer som vi inte ännu definierat.

Så är det t ex i setUpNewGame. Här skall ett antal stickor läggas i varje hög. Då behövs en operation putPin i Pile för att lägga en sticka i en hög. Om vi vill att samma spelplan skall kunna utnyttjas för flera spelomgångar måste vi också ha en operation clear i Pile för att ta bort eventuella stickor från högen.

När en spelare skall göra ett drag väljer han ut en hög och tar stickor från högen. Det verkar behövas två operationer för detta: spelaren utför en operation getPile på NimGame-objektet och därefter takePins på det utvalda Pile-objektet. Enligt 3.2.3 finns det en association mellan Player och NimGame och en mellan Player och Pile, så det är fullt möjligt att göra på det sättet. Vi kan dock minska antalet associationer om vi slår samman dessa båda operationer till en operation, som både väljer hög och tar stickor från högen. Om vi gör så behövs inte längre associationen mellan Player och Pile.

Vi inför därför operationen takePins i klassen NimGame. Operationerna skall ha högens nummer och antalet stickor som parametrar. Innan spelaren kan ta stickor måste han kontrollera att det verkligen finns så många stickor som han önskar ta i den utvalda högen, så vi inför också operationen possibleMove i klassen NimGame.

På detta sätt går vi igenom alla operationer. Till slut kommer vi fram till följande lista av operationer:

NimGame	setUpNewGame	Fördela stickorna på högarna
	possibleMove	Tag reda på om ett visst drag är tillåtet
	takePins	Tag stickor från en av högarna
	morePins	Undersök om det finns fler stickor att ta
Pile	clear	Töm högen
	putPin	Lägg en sticka i högen
	takePins	Tag stickor från högen
Pin	–	Inga operationer!
Player	makeMove	Gör ett drag
HumanPlayer	–	(ärver makeMove från Player)
ComputerPlayer	–	(ärver makeMove från Player)
Input	getPileNbr, getNbrPins	Läs ett drag
Keyboard	–	(ärver getPileNbr och getNbrPins från Input)

Det ser egendomligt ut att klassen Pin inte har några operationer. Visserligen tas det stickor från högarna, men det är operationer på Pile-objekten. Detta tyder på att klassen Pin inte behövs, och vi tar bort den ur modellen. Situationen skulle ha varit annorlunda om stickorna hade haft några egenskaper, t ex varit numrerade, och dessa egenskaper hade varit väsentliga i spelet.

En annan egendomlighet är att vi verkar ha tappat bort själva spelandet! Vi har i modellen inget objekt som är ansvarigt för att en spelomgång (=scenariot) genomförs. Vad vi har gjort är att definiera ett antal klasser som är nödvändiga för att genomföra en omgång enligt scenariot.

Detta problem träffar man ofta på. Ibland är lösningen att ”distribuera ut kontrollen” bland de deltagande objekten. I detta exempel skulle det innebära att varje spelare, när han gjort ett drag och fortfarande inte har vunnit, ser till att motspelaren gör sitt drag. Det som då återstår är att någon måste sätta igång spelet. En annan metod är att ”centralisera kontrollen” genom att låta ett särskilt objekt, ”kontrollobjekt”, vara ansvarigt för att de steg som behövs genomlöps. Om man har många komplicerade uppgifter som skall lösas kan det behövas flera kontrollobjekt. I vårt exempel skulle vi ha ett Controller-objekt, som sätter igång spelet och ser till att de båda spelarna omväxlande får göra sina drag. (Alternativt kan man naturligtvis se till att dessa uppgifter löses i huvudprogrammet, om man använder ett programspråk där det begreppet finns.)

Den sista metoden, med särskilda kontrollobjekt, anses av många inte vara riktigt rumsren när man sysslar med objektorienterad systemutveckling, eftersom man ju då gör en ”funktionsinriktad” del av systemet. Metoden har dock sina fördelar och vi kommer i fortsättningen att välja denna metod om vi tycker den verkar mest lämplig. Så är det i det aktuella exemplet. – Vårt problem kan härledas till ett val vi gjorde i 3.2.1, att förkasta Spelomgång som ett möjligt objekt. En sådan klass skulle haft precis det ansvarsområde som kontrollobjekt-klassen nu får.

Vi går alltså tillbaka några steg och inför en ny klass:

Controller	play	Spela en omgång
	winner	Tag reda på vem som vann spelomgången

3.2.6 Specificera operationer och attribut

Vi beskriver nu alla operationer fullständigt,. För att vi skall känna igen oss skriver vi specifikationer av klasserna på liknande sätt som i boken Objektorienterad programmering och Java, även om detta kanske är att föregripa implementationen väl mycket. Normalt är man inte så detaljerad i designfasen. Vi visar också de viktigaste attributen i klasserna.

Observera att vi har implementerat alla associationer med referensvariabler. Spelplanen ”består-av” tre stickhögar och klassen NimGame har därför attributet Pile[ ] piles. En spelare ”spelar-ett” Nim-spel och klassen Player har därför attributet NimGame myGame.

Observera också att operationerna makeMove i Player och getMove i Input är abstrakta. Att det skall vara så är självklart, t ex skall ju olika slags spelare kunna spela på olika sätt.

```
class NimGame {
  private Pile[] piles;                // de tre högarna

  /** skapa ett Nim-spel med tre högar */
  public NimGame();

  /** fördela stickor på högarna */
  public void setUpNewGame();

  /** undersök om det är möjligt att ta m stickor från hög nr k */
  public boolean possibleMove(int m, int k);

  /** undersök om det finns stickor kvar i någon hög */
  public boolean morePins();

  /** tag m stickor från hög nr k */
  public void takePins(int m, int k);
}
```

```
class Pile {
  private int nbrPins;       // antalet stickor i högen

  /** skapa en hög (från början tom) */
  public Pile();

  /** töm högen */
  public void clear();

  /** lägg en sticka i högen */
  public void putPin();

  /** tag m stickor från högen */
  public void takePins(int m);
}

abstract class Player {
  private String myName;     // spelarens namn
  protected NimGame myGame;  // spelet som spelaren spelar

  /** skapa en spelare med namnet nm som
      spelar spelet game */
  protected Player(String nm, NimGame game);

  /** tag reda på spelarens namn */
  public String getName();

  /** gör ett drag (abstrakt operation) */
  public abstract void makeMove();
}

class ComputerPlayer extends Player {
  /** skapa en datorspelare med namnet nm som
      spelar spelet game */
  public ComputerPlayer(String nm, NimGame game);

  /** gör ett drag enligt ”datorreglerna” */
  public void makeMove();
}

class HumanPlayer extends Player {
  private Input input;       // enhet för inmatning av drag

  /** skapa en människospelare med namnet nm som spelar
      spelet game. Dragen läses från inmatningsenheten in */
  public HumanPlayer(String nm, NimGame game, Input in);

  /** gör ett drag som läses från inmatningsenheten */
  public void makeMove();
}

abstract class Input {
  /** skapa en inmatningsenhet */
  public Input();

  /** läs ett högnummer */
  public abstract int getPileNbr();

  /** läs antal stickor */
  public abstract int getNbrPins();
}

// Keyboard är en subklass till Input där de abstrakta operationerna implementeras

class Controller {
  private Player player1,player2; // de båda spelarna
  private NimGame theGame;       // spelet

  /** p1 och p2 spelar game */
  public Controller(Player p1, Player p2, NimGame game);

  /** spela en omgång */
  public void play();

  /** tag reda på vem som vann */
  public Player winner();
}
```

3.2.7 Verifiera modellen

I detta moment övertygar vi oss om att alla objekt och operationer som vi behöver finns med i modellen, genom att i detalj gå igenom scenariot i 3.2.5. Vi kontrollerar också att parametrarna till operationerna verkar vara korrekta.

När det gäller dokumentation av modellen kan vi anse att beskrivningen i 3.2.6 är tillräcklig.

3.2.8 Implementation av modellen

Som redan tidigare nämnts kan det inträffa att man upptäcker brister i modellen i samband med implementationen. Detta inträffade också i det aktuella problemet, trots att problemet är litet och välkänt för författaren till detta kompendium. Vid implementationen upptäcker vi följande (i och för sig hade nog upptäckterna gjorts redan i samband med verifieringen av modellen, om denna gjorts tillräckligt noggrant):

1. I kravspecifikationen sägs att ”ett antal” stickor skall fördelas på tre högar. Detta antal måste naturligtvis specificeras någonstans, men det finns inte med i modellen. För att lösa problemet kan man ge operationen setUpNewGame i klassen NimGame en parameter nbrOfPins. Om man gör så bör också operationen play i klassen Controller ha samma parameter.

   En annan lösning är att ge konstruktorn i klassen NimGame denna parameter. Då kan man dock inte använda spelplanen för att spela flera spel med olika antal stickor, vilket förefaller vara en onödig begränsning.
2. I operationen morePins i klassen NimGame måste man kunna ta reda på antalet stickor i varje hög. Det måste alltså finnas en operation getNbrOfPins i klassen Pile.
3. När en dator skall göra ett drag behöver den, för att den skall kunna spela enligt någon vettig strategi, kunna ta reda på antalet stickor i varje hög. Vi inför därför operationen getNbrOf-Pins(int k) i klassen NimGame.
4. Av samma anledning måste antalet stickor i varje hög presenteras för en mänsklig spelare. Presentationen görs naturligtvis genom någon sorts utskrift av de tre antalen. Vi förberedde ju för användning av olika in-enheter genom att införa den abstrakta klassen Input, och på liknande sätt bör vi nu införa en klass Output och specialisera den till klassen Screen.

   Klassen Output skall ha en (abstrakt) operation printNumbers(n1, n2, n3), där n1-n3 är de tre antalen som skall skrivas ut. Operationen kan t ex utföras först i operationen makeMove i klassen HumanPlayer. För att HumanPlayer skall komma åt utenheten lägger vi till Output-objektet i parameterlistan till konstruktorn. Vi för då in en relation mellan HumanPlayer och Output och måste gå tillbaka till analysmodellen och dokumentera relationen.

3.3 Sammanfattning

Det är ingen större skillnad mellan den modell som vi här har kommit fram till och motsvarande modell i avsnitt 10.5 i Objektorienterad programmering och Java. Där har man dock inte tagit med klasserna Keyboard och Input, eftersom man inte tyckte att de tillhörde problemet.

Vi kan notera att klassen Pile här bara hanterar ett enda heltal (antalet stickor i högen). Det är dock ändå befogat att behålla klassen. Dels är ju ”hög” ett viktigt begrepp i det verkliga problemet, dels blir programmet lättare att modifiera om man t ex inför numrerade stickor.

4. Referenser

Bec89	A Laboratory for Teaching Object-Oriented Thinking, K. Beck och W. Cunningham. SIGPLAN Notices, October 1989.
Boo94	Object-Oriented Design with Applications, G. Booch. Benjamin-Cummings 1994 (2nd edition).
Boo98	The Unified Modeling Language User Guide, G. Booch, J. Rumbaugh, I. Jacobson. Addison-Wesley 1998.
Bus96	Pattern-Oriented Software Architecture – A System of Patterns, F. Buschmann m fl. John Wiley & Sons 1996.
Coa91a	Object-Oriented Analysis, P. Coad och E. Yourdon. Prentice-Hall 1991.
Coa91b	Object-Oriented Design, P. Coad och E. Yourdon. Prentice-Hall 1991.
Eri98	UML Toolkit, H-E. Eriksson och M. Penker. John Wiley & Sons 1998.
Gam95	Design Patterns – Elements of Reusable Object-Oriented Software, E. Gamma m fl. Addison-Wesley 1995.
Gra98	Patterns in Java, M. Grand. John Wiley & Sons 1998.
Jac92	Object-Oriented Software Engineering – A Use Case Driven Approach. I. Jacobson m fl. Addison-Wesley 1992.
Jac99	The Unified Software Development Process, I. Jacobson, G. Booch, J. Rumbaugh. Addison-Wesley 1999.
Poo99	Using UML: Software Engineering with Objects and Components, R. Pooley och P. Stevens. Addison-Wesley 1999.
Rum91	Object-Oriented Modeling and Design, J. Rumbaugh m fl. Prentice-Hall 1991.
Wir90	Designing Object-Oriented Software, R. Wirfs-Brock m fl. Prentice-Hall 1990.