HÖGSKOLAN KRISTIANSTAD

Examensarbete
HT 2007
Lärarutbildningen

Fel i undervisningen

eller

Att lära av misstag

Författare
Maria Hedenby

Handledare
Lars-Olof Johansson
Örjan Hansson

www.hkr.se

(Denna sida har avsiktligt lämnats tom.)

Fel i undervisningen

eller

Att lära av misstag

Abstract

Arbetet syftar till att undersöka hur felsökning inom programmering används i undervisningen på gymnasiet och vilket stöd som finns i styrdokument och kurslitteratur.
En stor del av elevernas tid går åt till felsökning, ändå tycks det inte angripas som ett eget moment i undervisningen. Undersökningen grundar sig på två intervjuer med vardera två gymnasielärare och har bearbetats utifrån en kvalitativ metod samt studier av kurslitteratur och styrdokument. Under den andra intervjun presenterades ett förslag på en metod för felsökning. Resultatet av undersökningen blev att mycket lite stöd finns i kurslitteraturen trots att det finns som mål i kursplanerna. Lärarna har inte felsökning som ett eget moment i undervisningen och heller ingen strategi för hur de går till väga. Däremot skulle de gärna ha en metod att använda och tror att det skulle öka motivationen hos eleverna.

Författare Maria Hedenby

Datum:2007-12-14

(Denna sida har avsiktligt lämnats tom.)

Innehållsförteckning

1 INLEDNING 6

1.1 BAKGRUND 6
1.2 FELSÖKNINGSMETOD 9
1.3 STYRDOKUMENTEN 10
1.4 SYFTE 12
1.5 PROBLEMPRECISERING 12

2 METOD 13

2.1 INLEDNING 13
2.2 LÄROMEDLEN 13
2.3 INTERVJUERNA 14

3 LITTERATURGENOMGÅNG 16

3.1 KONSTRUKTIVISMEN 16
3.2 BRICOLAGE 17
3.3 MINIMALISM 17
3.4 PAPPER OCH PENNA ELLER DATOR OCH KOMPILATOR 17
3.5 OBJEKTORIENTERAD PROGRAMMERING 18
3.6 ANDRA ASPEKTER 18
3.7 SLUTORD 19

4 RESULTAT 20

4.1 SKOLVERKETS KURSPLANER 20
4.2 UNDERVISNINGSMATERIAL 20
4.3 INTERVJUERNA 21
4.3.1 Första intervjun 21
4.3.2 Andra intervjun 22

5 DISKUSSION 26

6 SLUTSATS 27

7 REFERENSER 29

8 BILAGOR 32

8.1 BILAGA 1 - FELSÖKNINGSMETOD 32
8.2 BILAGA 2 - FLÖDESSCHEMA 35
8.3 BILAGA 3 - INTERVJUUNDERLAG 36
8.4 BILAGA 4 – UNDERSÖKTA LÄROMEDEL 37

1 Inledning

1.1 Bakgrund

Under mina praktikperioder på gymnasiet har jag kunnat konstatera att många elever i programmering lägger ner mycket tid på att leta efter fel som inte har med det egentliga problemet att göra. De kunde till exempel tillbringa större delen av ett lektionspass med att leta efter ett program eller en fil som programmet skulle använda. I en annan situation kunde de stirra sig blinda på fel i sina program. Många gånger förstod de inte alls vad de skulle leta efter eller hur de skulle göra för att komma vidare. Ibland gjorde de planlösa försök att kopiera något ur boken i hopp om att det skulle lösa felet. När felet eventuellt var rättat, visade det sig ofta att de varken förstått varför det blivit fel eller varför det nu var rätt! Den bestående effekten verkade vara att de tappade den motivation de hade från början.

De flesta observationerna har jag gjort på en gymnasieskola med uttalad datainriktning. Alla elever har en egen laptop som de kan använda både i skolan och hemma. De har också tidigare haft datorkunskap på schemat. De lektioner jag har observerat är programmering A och B. (Programmering C – som också finns som kurs - gavs inte detta läsår på skolan.)

Man brukar skilja på tre olika typer av fel i programmering. Skansholm (2003) ger följande definition:

kompileringsfel	*Språkreglerna har inte följts. Upptäcks vid kompilering.*
logiskt fel	*Felaktig lösning av problemet. Ger felaktigt resultat.*
exekveringsfel	*Felsignal uppstår vid körning. Kan avbryta programmet. (Skansholm, 2003, s.143)*

Kompileringsfel är fel som uppstår när man ska översätta (kompilera) programmet så att datorn förstår instruktionerna. Varje programmeringsspråk har – precis som mänskliga språk - sin speciella syntax. Om kompilatorn som översätter programmet upptäcker sådana fel, ger den felmeddelanden och avbryter sedan översättningen för den förstår inte hur den ska tolka det som skrivits. Det blir ingen kod som kan köras på maskinen förrän dessa fel är korrigerade.

Nedan finns ett exempel på hur ett litet program kan se ut. Sist på rad tre har jag felvänt en ’måsvinge’ – den borde ha sett ut så här ’{’, men ser ut så här ’}’.

```
import javax.swing.*;
public class Hej {
        public static void main (String[] arg) }
                JOptionPane.showMessageDialog(null, "Hej!");
                System.exit(0);
        }
}
```

Vid kompileringen generar detta lilla syntaxfel följande felmeddelanden.

```
Hej.java:6: ';' expected
        public static void main (String[] arg) }
Hej.java:7: 'class' or 'interface' expected
                JOptionPane.showMessageDialog(null,"Hej");
Hej.java:8: 'class' or 'interface' expected
                System.exit(0);
Hej.java:9: 'class' or 'interface' expected
        }
Hej.java:11: 'class' or 'interface' expected
5 errors
```

Kompilatorn hittar fem fel. Det är dock bara ett fel i koden. Resten är följdfel då kompilatorn inte kan tolka vad som menas.

Logiska fel är fel som uppstår för att man har tänkt fel eller inte tänkt tillräckligt på olika situationer som kan uppstå. Det kan vara att kontrollera att data uppfyller vissa kriterier. En ålder får till exempel inte vara negativ. Ett annat exempel är att dividera med en variabel som antar värdet 0. Kompilatorn hittar inte dessa fel, utan felen visar sig genom ett felaktigt resultat.

Exekveringsfel är när programmet får en felsignal från datorn när det körs, ofta med följd att programmet abryts. Ett logiskt fel kan ligga bakom även dessa fel, men de kan också bero på yttre omständigheter.

Många problem verkar bero på dålig struktur – oordning rätt och slätt. Denna brist på struktur verkar i sin tur bero på okunskap snarare än på ovilja. Hur ska man organisera sina filer och program på bästa sätt? Vad ska man tänka på och när är det dags att börja? Äger man tre böcker behöver de knappast infogas i ett avancerat klassningssystem för att bli lätt åtkomliga. I ett bibliotek krävs det däremot att

böckerna är systematiskt ordnade efter en viss struktur för ett de ska gå att hitta någorlunda snabbt.
Andra problem verkar bero på att eleverna inte har en hållbar modell av datorn. Detta belyses närmare i litteraturgenomgången i kapitel 3.
En annan orsak är att programmeringsspråken inte överser med människors bristande noggrannhet. Minsta lilla fel i syntaxen eller logiken kan få oanade konsekvenser som visar sig på besynnerliga sätt.
Dessutom är datorsystem komplexa och påverkas av många parametrar. Den enklaste datormiljö idag består av ett stort antal programvaror som samverkar med varandra. Till synes lika miljöer kan skilja sig vad gäller uppdateringsnivå på del av programvaran.
Detta medför att felhantering ingår som en naturlig del i programmerarens vardag.

Läraren i programmering står inför ett antal problem. Datorn är vår nya medierande artefakt. ”Medierande redskap bjuder motstånd för den oinvigde, och det är övervinnandet av detta motstånd som är en del av approprieringens kärna. När vi väl bekantat oss med ett medierande redskap, tappar vi ofta förmågan att se vad som är speciellt med det. Redskapet ”naturaliseras” och vi tror att det vi ser är uppenbart också för andra.” (Säljö, 2005, s. 146)
Detta får mig osökt att tänka på följande ordförklaring:
”Self-Evident, *adj*.Evident to one’s self and to nobody else.”(Bierce, 1985, s.289)

Ett annat problem som läraren ställs inför är miljön som eleverna arbetar i. Alla elever har inte samma miljö, så läraren måste kunna anpassa undervisningen därefter. Då det inte kommer att kunnat säkerställas att miljön är samma och då inte att läraren i detalj kommer att kunna känna till hur den enskilde elevens miljö ser ut, måste lärarna få ett stöd så att eleverna kan komma vidare på egen hand.

Under mina första datalektioner som lärarstuderande besökte jag en gymnasieklass som arbetade i en miljö och med ett språk som jag aldrig ens hört talas om. De hade fått uppgifter av sin lärare som de arbetade enskilt med. Många fastnade med sina fel och jag var uppriktigt förvånad över hur lätt jag kunde hjälpa dem att komma vidare bara genom att ge dem en generell metod att arbeta efter. På så vis kunde de själva

lösa sina problem. Programmering är ju bland annat en konst i att kunna växla mellan helhet och detaljer, mellan det triviala och det viktiga och att kunna se vad som är väsentligt med detsamma och vad som kan lämnas därhän för tillfället. Metoden hjälpte eleverna dels att få struktur på sin kod och sina filer, dels att ringa in ett fel som uppstått. Är programmet lättläst kan eleven koncentrera sig på vad programmet ska utföra snarare än att förstå kod som är kryptiskt skriven. Enkla medel som bland annat formattering och namnstandard ger överblick och tar bort fokus på detaljerna till förmån för helheten. Läraren kan å sin sida koncentrera undervisningen på programmering och felhantering som modell.

1.2 Felsökningsmetod

Mitt förslag på hur en enkel felsökningsmetod kan se ut beskrivs kortfattat nedan. Metoden bygger på många års erfarenhet inom IT-branschen, där en stor del av IT-avdelningarnas tid och budget går åt till felsökning och rättning av program. En mer utförlig beskrivning finns i bilaga 1.

Formattering av program

Indragningar brukar göras på så sätt att man lätt ser vilka kodavsnitt som hänger ihop eller är på samma nivå.

Exempel:
```
if (pos = 1) {
    // de satser som ska utföras om villkoret är sant dras in en bit
} //end if (pos = 1)
```

Samla variabeldeklarationer i början av ett program eller i början av en modul / metod / subrutin[1].

Samla moduler på att ställe i programmet och gärna i slutet. Låt deklarationer och liknande börja programmet följt av huvudflödet och sedan moduler.

Skilj olika moduler åt med en kommentarsrad som bara består av till exempel streck så är det lättare att se var de börjar och slutar.

Kommentering av program

Alla kommentarer bör vara tydliga men korta och enkla att ändra vid uppdatering av programmet.

Namnstandard / versionshantering

[1] En modul är förenklat (liksom metod och subrutin) en benämning på ett program i programmet. Det finns även andra benämningar av likartade fenomen. I fortsättningen kommer jag att använda benämningen modul, men motsvarande gäller även för de andra benämningarna.

Allt som ska namnges, från bibliotek via program och klasser till variabler, bör få beskrivande namn.

Modularisering av program

Ett program bör delas upp i moduler så att inget överstiger 1-2 skärmsidor. Huvudflödet bör inte heller överstiga denna storlek. På så sätt ökar överskådligheten och därmed förståelsen.

Modularisering underlättar också felsökning då det är lättare att kontrollera resultatet av en mindre mängd kod. Genom huvudflödet ska logiken framgå.

Flödesschema

Ett flödesschema (se exempel i bilaga 2) ska beskriva händelseförloppet av ett program. Det kan göras på olika detaljnivå alltefter behov.

Utskrift av mellansteg i program

Utskrift från ett program under tiden det körs kan göras på olika sätt och med olika syften. Man kan dels skriva ut att man befinner sig på ett speciellt ställe i programmet, dels kan man skriva ut innehållet i en variabel.

Uteslutning av kod (bortkommentering) i program

Man kan vid behov tillfälligt kommentera bort en modul för att begränsa sin felsökning.

Skrivbordstest av program

Ibland kan det underlätta att sätta sig med en utskriven version av programmet och själv gå igenom instruktion för instruktion, samtidigt som man antecknar mellanresultat av variabler.

Strukturering av bibliotek/mappar, filer och program

Försök hitta en hierarkisk organisation av bibliotek, filer och program.

1.3 Styrdokumenten

Läroplanerna är hämtade från Lärarens handbok (2005). (Kursplanerna behandlas i resultatgenomgången i kapitel 4.1.) Läroplanen för det obligatoriska skolväsendet (Lpo 94) säger: ”Skolan skall sträva efter att varje elev utvecklar sitt eget sätt att lära.” (Lärarens handbok, 2005, s. 14)

Nedan följer ett antal punkter som är tagna ur 1994 års läroplan för de frivilliga skolformerna (Lpf 94)**.**

- ”Elevernas kunskapsutveckling är beroende av om de får möjlighet att se samband. Skolan ska ge eleverna möjligheter att få överblick och

sammanhang, vilket fordrar särskild uppmärksamhet i en kursutformad skola. Eleverna skall få möjlighet att reflektera över sina erfarenheter och tillämpa sina kunskaper.” (Lärarens handbok, 2005, s. 40)

- ”Varje elev skall få stimulans att växa med uppgifterna och möjlighet att utvecklas efter sina förutsättningar.” (Lärarens handbok, 2005, s. 40)
- ”Läraren skall utgå från den **e**nskilde elevens behov, förutsättningar, erfarenheter och tänkande.” (Lärarens handbok, 2005, s. 45)
- ”Läraren skall i undervisningen skapa en sådan balans mellan teoretiska och praktiska kunskaper som främjar elevernas lärande.” (Lärarens handbok, 2005, s. 45)

En lärare som kan hjälpa eleverna förstå och korrigera sina egna fel, reflektera över sina erfarenheter och tillämpa sina kunskaper får eleverna att utveckla förmågan att se samband. Detta gör att eleverna växer med uppgiften och växlar mellan teori och praktik på sitt eget, personliga sätt. Undervisning som riktar in sig på att eleverna ska förstå hur de själva kan angripa och lösa sina fel och på så sätt komma vidare har därmed ett starkt stöd i styrdokumenten.

1.4 Syfte

Många år inom IT-branschen har gett mig liknande erfarenhet av fel vid programmering som vid gymnasieskolorna. En stor del av IT-avdelningarnas tid och budget går åt till felsökning och rättning av program. Därför kan man ställa sig frågan: Hur ser undervisningen i felhantering i programmering ut på gymnasieskolan?

Mer precist syftar den här uppsatsen till att undersöka om det finns stöd för felhantering i programmeringsundervisningen på gymnasiet och hur verksamma lärare arbetar med felhantering.

1.5 Problemprecisering

Mina frågeställningar blir:

- Hur tas felsökning upp i gymnasieutbildningens kursplaner för programmering?
- Hur ser stödet ut för felsökning i läromedlen?
- Hur arbetar verksamma lärare med felsökning som moment i undervisningen på gymnasiet?
- Hur ser verksamma lärare på en felsökningsmetod för gymnasieelever?

2 Metod

2.1 Inledning

Jag har gjort en fallstudie baserat på kvalitativa data. En fallstudie är enligt Andersen (1998):

> en empirisk undersökning
> - som belyser ett samtida fenomen inom det verkliga livets ramar
> - där gränserna mellan fenomenet och det sammanhang i vilket det ingå inte är självklart och
> - där det är möjligt att använda flera informationskällor för att belysa fenomenet (s. 129)

Det kvalitativa data kommer från två olika källor. Primärkällan kommer från intervjuer och sekundärkällan från läromedel som används i programmeringsundervisningen på gymnasiet.

Från början var uppsatsen tänkt att beskriva en undersökning om elever kunde utveckla sin förståelse för programmering genom att tillämpa en metod för felsökning. Metoden (se bilaga 1) bygger på en lektionsserie om felsökning som jag tidigare har tagit fram. Av olika anledningar visade det sig sen att jag inte längre kunde ha tillgång till elevgruppen. I stället ändrade jag infallsvinkeln till att undersöka hur det pedagogiska stödet ser ut för felsökning.

- Hur tas felsökning upp i gymnasieutbildningens kursplaner för programmering?
- Hur ser stödet ut för felsökning i läromedlen?
- Hur arbetar verksamma lärare med felsökning som moment i undervisningen på gymnasiet?

2.2 Läromedlen

Undersökningen omfattar tjugo läromedel som används i undervisning på gymnasiet. När jag har undersökt läromedlen har jag dels gått igenom innehållsförteckningen, dels sökt i registret / indexet då sådant har funnits. Jag har letat efter ord som har med fel att göra. Sedan har jag letat upp avsnittet inne i boken och läst vad som står där. Jag tittade särskilt på;

- om de tog upp olika typer av fel
- om de gav exempel på dessa feltyper

- om de beskrev och förklarade orsaker och sammanhang till felen eller
- om de beskrev något sätt att angripa problemen.

Det är möjligt att det kan stå något om felsökning i andra avsnitt i boken. I så fall har det inte kommit med i min undersökning. Min avsikt har varit att undersöka om man explicit tar upp felsökning i läromedlen.

2.3 Intervjuerna

Jag har intervjuat två lärare i två olika omgångar. Lärarna arbetar på en gymnasieskola med uttalad datainriktning. Båda lärarna undervisar i programmering A, B och C. Båda lärarna har undervisat i flera år och har tidigare erfarenhet inom IT-branschen.

Intervjuerna har varit semistrukturerade, vilket kännetecknas av att de är ”varken ett öppet samtal eller ett strängt strukturerat frågeformulär” (Kvale,1997, s. 32). Detta för att ge lärarna så mycket utrymme som möjligt och inte i onödan styra deras tankar.

Vid båda tillfällena har jag före intervjun upplyst om det övergripande syftet och hur materialet kommer att behandlas. Jag upplyste också om att det var tänkt som en diskussion där deras synpunkter och tankar var det viktiga och frågorna endast en vägledning för att hålla den röda tråden. Intervjuerna har spelats in på band och skrivits ut.

I den första intervjun undersökte jag hur lärarna tog upp frågor rörande felsökning, vad de använde för eventuellt stöd samt hur eleverna reagerar vid fel. Som stöd för mitt minne hade jag sammanställt ett antal frågor som jag använde som en checklista för mig själv (se bilaga 3). Själva samtalet rörde sig mer fritt kring felsökningens olika aspekter.

I den andra intervjun som var av explorativ karaktär, ställdes de inför faktiska felsituationer och ombads beskriva hur de skulle angripa problemet i en klassrumssituation. Explorativa undersökningar ”syftar till att utforska förhållanden eller fenomen som är mindre kända eller kanske helt okända (Andersen, 1998, s. 18).Med vissa ledande frågor försökte jag få lärarna att utveckla sitt resonemang och få fram om de omedvetet använde sig av en metod för att leda eleverna på rätt spår. Det kunde till exempel vara att inta rollen som elev för en stund. Däremot försökte jag låta bli att lägga orden i munnen på dem eller styra dem då det inte var nödvändigt. (Ibland var det nödvändigt att påminna om att intervjun handlade om hur läraren skulle gör för att hjälpa en elev i en liknande situation snarare än att här och nu lösa

problemet.) När felsituationerna var genomgångna, presenterade jag min metod och diskussionen fortsatte med den som utgångspunkt (se bilaga 1). Huvudsyftet med en intervju kan enligt Kvale (1997) vara antingen empiriskt för att samla in information eller teoretiskt det vill säga att pröva konsekvenserna av en teori. Den andra intervjun som bestod av två delar kan sägas vara empirisk i sin första del och teoretisk i den andra.

Intervjuerna genomfördes i ett av skolans grupprum med var lärare för sig. Vid den andra intervjuomgången hade vi tillgång till en laptop för att kunna titta på programmen som utgjorde diskussionsunderlaget – och eventuellt kompilera eller testa dem. Eftersom det skulle likna en klassrumssituation var det viktigt att någorlunda kunna ge motsvarande förutsättningar. Då jag har både praktiserat och vikarierat på skolan känner båda lärarna mig ganska väl vilket gjorde intervjusituationen avspänd.

3 Litteraturgenomgång

Trots att man brukar få åtminstone ett fel varje gång man skriver ett program – litet som stort, nybörjare som expert – är det svårt att finna litteratur som behandlar ämnet. Jag har därför valt att i stället granska litteratur som tar upp möjliga orsaker till att det blir fel. Det handlar mycket om grundförståelse för datorn och för utveckling av program för datorn.

Att det kan vara svårt att lära sig programmera kan man ana då mellan 25 och 80 % av förstaårsstudenterna inte avslutar kursen. Orsaken är att studenterna inte klarar att skapa en modell av vare sig uppgiften eller datorn (Guibert, 2005). Att en grundförståelse för uppgiften är viktig vid lärandet framgår också av Herskin's (1999, s.45): ”Utgångspunkten för hela förståelseprocessen är att användaren förstår problemet, dvs. vad de ska använda funktionen till. Om deltagarna inte har förstått behovet, kan de inte säga ”aha” till lösningen.” Att inte ha en hållbar modell av datorn innebär att de inte förstår den feedback som datorn ger, allra helst då de arbetar i avancerade miljöer som i sig bygger på komplexa modeller (Guibert, 2005).

3.1 Konstruktivismen

Konstruktivismens grundtanke är att kunskap byggs upp successivt genom att ny information kombineras med tidigare erfarenheter och skapar på så sätt en modell av verkligheten. Eftersom det inte finns någon absolut sanning kan denna modell inte vara fel, däremot kanske den inte är hållbar. Det är lärarens roll att hjälpa eleven att bygga upp hållbara modeller. För att kunna hjälpa eleven måste läraren förstå vilken logisk struktur som eleven bygger sin modell på (Ben-Ari, 2001). Detta stämmer väl överens med Lpf94: ”Läraren skall utgå från den enskilde elevens behov, förutsättningar, erfarenheter och tänkande.” (Lärarens handbok, 2005, s. 45). Den modell som eleven har, hållbar eller ej, ligger alltså till grund för elevens vidare kunskapskonstruktion. Det är därmed viktigt att de grundläggande modellerna lärs ut från början innan man lär ut abstraktioner eller låter datorns verklighet helt ta över (Ben-Ari, 2001).

Lärandet måste vara aktivt och inom datavetenskap innebär det osvikligt att det blir fel, så kallade buggar. Konsekvenserna av konceptuella missuppfattningar visas omedelbart av datorn (Ben-Ari, 2001). Liksom vid abstraktioner är det viktigt att den

grundläggande, mentala modellen finns innan eleverna ger sig i kast med konkreta uppgifter som ett aktivt lärande förutsätter.

En av fördelarna, vid till exempel laborationer, är att man lär på egen hand. Med rätt typ av uppgift uppmuntras reflektion och utforskande vilket underlättar uppbyggnaden av en hållbar modell (Ben-Ari, 2001).

Konkret, handfast lärande förespråkas också av bricolage[2] och minimalism.

3.2 Bricolage

Bricolage innebär ett angreppssätt med trial-and-error. Man testar olika sätt och ser vad som händer. Det överensstämmer med det konstruktivistiska synsättet genom att olika elever lär på olika sätt. Det gynnar nybörjare genom att uppmuntra konkret tänkande. Däremot är det inte effektivt som metod när man kommit lite längre även om de flesta praktiserar bricolage ibland. Då konstruktivismen menar att god programmering måste bygga på en hållbar modell, måste bricolage komplettera en formell metod och inte ersätta den (Ben-Ari, 2001).

3.3 Minimalism

Minimalismen förespråkar som konstruktivismen och bricolage ett aktivt lärande med realistiska uppgifter. Gemensamt med konstruktivismen har man också synen på betydelsen av förkunskaper. Till skillnad från bricolagets mer ostrukturerade angreppssätt, utnyttjar minimalismen fel och felhantering som ett pedagogiskt redskap och fel och konceptuella missuppfattningar är inte symptom på misslyckanden. Däremot leder avsaknaden av hållbara modeller till svårigheter att klara av avancerade uppgifter. En annan skillnad är att minimalismen fokuserar på problemlösningen medan konstruktivismen fokuserar på problemlösandet (Ben-Ari, 2001).

3.4 Papper och penna eller dator och kompilator

En studie som jämför lösandet av uppgifter med papper och penna eller med dator och kompilator, visar att eleverna föredrog dator-kompilator trots att det tog längre tid (Gomm, Hailey Waters, Feng, 2003-2004). Den motivation det innebär att få använda datorn bör utnyttjas om än inom kontrollerade former.

[2] Betyder ungefär fixa. I franskan används uttrycket ofta för hobbyfixaren hemma.

”Den individ som kan läsa, befinner sig i många sammanhang i en annan situation än den som inte kan, när det gäller deltagandet i många olika verksamheter.” (Säljö, 2005, s.206). Med datorernas inträde och framväxt blir ovanstående påstående lika relevant för den som inte kan hantera en dator. Nu ska vi inte bara kunna läsa utan även kunna tolka ett otal symboler. GUI (Graphical User Interface) och WYSIWYG (What You See Is What You Get) kan verka intuitiva och därmed lätta att använda för nybörjaren. Men, den modell som till exempel ikoner bygger på, är det inte säkert att studenterna har med sig i bagaget (Ben-Ari, 2001).

3.5 Objektorienterad programmering

Vid val av programmeringsspråk och programmeringsmiljö, finns det anledning att tänka sig för. Objektorienterad programmering (OOP) har fördelen av att uppmuntra eleven att se till helheten genom en strategisk ansats. Har de däremot inte fått klart för sig den underliggande modellen innan de konfronteras med abstraktionerna inom OOP, går det emot Piaget’s syn om att abstraktioner kommer efter assimilering, dvs att man måste tillgodogöra sig modellen innan man kan förstå abstraktioner. Läraren måste vara försiktig och hitta en medelväg mellan dessa för att inte eleven ska bilda konceptuella missuppfattningar (Ben-Ari, 2001).

3.6 Andra aspekter

En annan aspekt är om det är bättre att arbeta i grupp eller individuellt. En studie visar att teamwork har en positiv effekt på resultatet (då åtminstone en är duktig i programmering), men att de som var duktiga inte fick ut så mycket av det och att det var krävande för läraren (Van der Vyver, Lane, 2003).

Vid universitetet i Porto (Portugal) går man löpande igenom sin utbildning inom informationssystem i syfte att förbättra den. Bland annat finner man att avståndet mellan IT- och affärsfolk är för stort. Det för med sig att organisationsteori och problemlösningsmetoder bör införas i utbildningen (Lopes, Morais, 2002). Som Guibert (2005) konstaterar krävs en hållbar modell även av problemets struktur för att undvika fel.

En fenomenografisk studie över förstaårsstudenter, delar upp dem i grupper beroende på deras sätt att lära sig programmera. Dessa kan sammanfattas enligt följande:

1. Following (betyg mot minsta möjliga ansträngning)
2. Coding (syntax ock kod)

3. Understanding and integrating
4. Problem solving
5. Participating or enculturation

De två första fokuserar mer på detaljerna och de tre sista på helheten. Detta får givetvis konsekvenser för hur utbildningen bör bedrivas om studenterna ska utvecklas till en mer sofistikerad nivå (Bruce, Buckingham, Hynd, mcMahon, Roggenkamp, Stoddley, 2004).

3.7 Slutord

Visserligen sägs det att ”det är de små detaljerna som gör det”, men inte bara konstruktivismen har en förkärlek för konceptuella modeller (dvs vilka delar som ingår och hur de hänger ihop till exempel ett flödesschema).

- ”Att förstå är att se mönster” (Gärdenfors, 2001)
- ”Om man blandar detaljer och principer, så riktas deltagarnas uppmärksamhet osvikligen mot detaljerna.” (Herskin, 2001, s.19) (vilket gör att man inte förstår helheten)
- ”Begreppslig kunskap kan förflyttas mellan olika situationer och en väsentlig komponent i vårt kunnande blir vanan att se hur något skall konstitueras för att vi skall förstå det eller kunna vidta någon åtgärd.” (Säljö, 2005)

4 Resultat

4.1 Skolverkets kursplaner

I kursplanen för programmering A står det att eleven skall:

- kunna analysera programmeringsuppgifter och formulera strukturerad pseudokod samt konstruera enkla algoritmer (Skolverket).
- kunna systemera och strukturera programmeringsarbetet samt skriva enkla program och felsöka källkod (Skolverket).

I kursplanen för programmering B står det att eleven skall:

- kunna analysera programmeringsuppgifter och formulera strukturerad pseudokod (Skolverket).
- kunna skriva program och felsöka källkod (Skolverket).

I kursplanen för programmering B står det att eleven skall:

- kunna analysera programmeringsuppgifter och formulera objektorienterad pseudokod (Skolverket).
- kunna skriva program och felsöka källkod (Skolverket).

4.2 Undervisningsmaterial

En intressant aspekt var att se hur läromedlen hanterar problemet. Jag har undersökt 20 läromedel som kan användas för undervisning på gymnasiet (se bilaga 4). Av dessa 20 böcker är det endast fem som berör felsökning i den mening som denna uppsats handlar om.I stort sett alla böcker behandlar felhantering med betydelsen avbrottshantering. Med det menas vad programmet ska göra om något oförutsett inträffar då det körs. De flesta programmeringsspråk har en generell rutin som kan användas för sådana situationer. Det kan till exempel vara att spara undan vissa data och skriva ut någon typ av felmeddelande. Dessa rutiner har karaktären av ’rädda vad som räddas kan’ men ger egentligen inte någon större vägledning om felorsaken. Logiska fel som inte skulle ge upphov till ett avbrott i programmet fångas inte upp. Det handlar alltså framförallt om att ta hand om fel som har uppstått, inte om att hitta dem.

Nedan tar jag upp de fem böcker som har med något om felsökning och hur de tar upp det.

- Abelli, B. (2004). *Programmeringens grunder* är en bok som behandlar programmering på ett mer generellt plan. Den är inte i första hand en bok i något programmeringsspråk utan ger exempel på syntax och konstruktioner i flera av de vanliga språken. På fem sidor beskriver den de tre vanliga feltyperna, ger några små exempel och förklarar vad en debugger[3] är.
- Gyulai, N. (2002). *JAVA från A till O: Programmering, kurs A, B och C* har tre övningar som går ut på att man ska finna olika typer av fel. Till två av övningarna finns en ledtråd. Däremot ges ingen beskrivning av hur man bör gå tillväga för att ringa in felet.
- Jonsson, T. (2003). *Programmering i fokus: Grundläggande programutveckling i Visual Basic.NET* ger ett exempel och beskriver debuggern på åtta sidor.
- Linton, G., & Malmback, C. (2003). *Programmering A Visual Basic* tar på tre sidor upp de olika feltyperna och beskriver debuggern något. Dessa sidor omfattar också den felhantering som beskrevs ovan.
- Skansholm, J. (2003). *Java – första steget* är den enda av de böcker som jag har undersökt som i någon egentlig mening tar upp felsökning. På ett tiotal sidor beskrivs de olika feltyperna med exempel och en beskrivning av möjliga orsaker. Inte heller här finns någon metod för hur man kan underlätta felsökning genom att ringa in sina fel. Jag vände mig också direkt till läromedelsförlagen som borde ha den mest aktuella informationen och eventuellt bimaterial där felsökning kunde finnas. Ingen har svarat.

4.3 Intervjuerna

4.3.1 Första intervjun

Båda lärarna har i stort samma uppfattning om felsökning som moment, varför jag inte skiljer på intervjuerna utan sammanfattar vad som kom ut av dem. De konstaterar att det inte finns något stöd för felsökning i de läroböcker de använder sig av. De använder sig inte heller själva av någon metod för felsökning i undervisningen. Man använder en ad hoc-lösning när felen dyker upp.

En grupp starka elever har goda kunskaper med sig när de börjar och är vana att hantera fel på egen hand. De är motiverade och vana vid felhantering och tar därför

[3] En debugger är ett mer eller mindre sofistikerat verktyg för att hitta programmeringsfel.

tag i problemen själva. De använder sig ofta av bricoalge och att hitta felorsaker på Internet där de diskuterar problem på olika forum. Motpolen är den grupp elever som inte anstränger sig och som i grunden inte är särskilt intresserade. Det stora flertalet elever vet inte hur de ska komma vidare, men skulle nog försöka om de bara visste hur. En brist i grundförståelse för datorer verkar vara orsaken. De tror att datorn kan mer än vad den egentligen kan. Språk där man direkt kan göra grafiska program (Visual Basic) underlättar motivationen eftersom de kan se resultat ganska omgående. Miljöer som ger mycket stöd har fördelen av att man kommer igång snabbt, men nackdelen att man inte får full förståelse för hur det egentligen fungerar. Får man dessutom problem med själva miljön är det i stort sett omöjligt för en elev att komma vidare själv. Däremot slipper eleven fördjupa sig i vissa delar (som IO) från början utan kan koncentrera sig på de andra bitarna (som ofta är mindre komplicerade och dessutom hänger ihop med det problem som de ska lösa). De starka eleverna hjälper ofta andra elever, men utan att eleven som fick hjälp egentligen förstod varför det blev fel.

Båda lärarna försöker få eleverna att använda sig av någon slags metod för att klara av fel, men saknar stöd för det. De tror båda att det skulle underlätta för både eleverna och dem själva om det fanns en mer formell metod de kunde använda sig av. Båda lärarna påpekar också att den lektionstid som är avsatt för kursen inte räcker till för att lära sig programmera. Det krävs att eleverna fortsätter hemma (alla har egna datorer).

4.3.2 Andra intervjun

Under den andra intervjun ställdes lärarna först inför realistiska felsituationer och ombads att beskriva hur de skulle ha agerat i en klassrumssituation. Jag växlar mellan rollen som tänkt elev och rollen som diskussionsledare. Programmen är skrivna i Java och består av tre separata filer. Det är i princip samma program men med olika typer av fel och olika grad av läsbar kod. Programmet tillåter två spelare att spela luffarschack.Därefter visade jag dem min metod för felsökning och diskussionen fortsatte med den som underlag.Som framkom av den första intervjun använder ingen av lärarna någon metod för felsökning i undervisningen. Ställda inför felsituationerna i den andra intervjun reagerar de på olika sätt.

Lärare 1 försöker att lösa felet själv för att eleven ska kunna komma vidare i programmet. Så småningom kommer det fram mer generella tips om hur man kan gå tillväga.Läraren är rädd för att eleverna ska tappa motivationen om de inte snabbt kommer vidare.

Lärare 2 försöker få eleverna att själva lösa problemet genom att ge dem tips på vad det skulle kunna röra sig om för typ av fel. Ganska snart kommer det dock fram generella tips på hur man kan gå tillväga. Läraren menar att eleverna inte lär sig något om de inte själva löser problemen.

Situation 1

Programkoden i situation 1 är ostrukturerad, utan kommentarer men med ganska väl valda variabelnamn. Programmet innehåller kompileringsfel.

Lärare 1 försöker att lösa felet själv. Vad felet beror på är inte uppenbart genom att bara titta på koden och efter ett tag efterlyser läraren ”en klar och tydlig kommentar”. Efter ytterligare studier av koden ber läraren om att den ska formatteras för att öka läsbarheten.

Lärare 2 förklarar för eleven hur man brukar angripa kompileringsfel och ber eleven att själv försöka lösa felet. Då eleven (dvs jag) återkommer påpekar läraren att: ”Det är lite rörigt här.”. Efter att ha förtydligat vad rörigheten består i, säger läraren: ”Så jag ber väl om att fixa till den formatteringen först. För då hittar dom förhoppningsvis felet.”

Situation 2

Programkoden i situation 2 är nu formatterad men utan kommentarer. Felet är en typ av miljöfel – koden är i sig riktig men delarna hänger inte ihop som de ska. Programmet går att kompilera men det kommer inte upp någon spelplan. Endast lärare 2 blev ställd inför den här situationen på grund av tekniska problem med lärare 1:s maskin.

Lärare 2 ber direkt eleven att kontrollera namnen på filerna och att de ligger i rätt mapp. Sedan försöker läraren att själv förstå vad felet består av för att kunna leda eleven på rätt spår.

Situation 3

Programkoden i situation 3 har blivit kommenterad och modulariserad. Problemet består av två logiska fel som gör att programmet ibland inte reagerar som det borde. I övrigt går det bra att köra programmet och spela spelet. Jag förklarar hur felen yttrar sig.

Lärare 1 försöker lösa problemet genom att studera koden. Läraren efterlyser efter en stund fler kommentarer och ett flödesschema för att få grepp om helheten. Föreslår sedan hur man kan gå vidare med hjälp av flödesschemat. Efter ytterligare en stund menar läraren att det hade underlättat med ett verktyg för att ringa in problemet.

Lärare 2 börjar med att hoppas att det har tillkommit kommentarer för att öka förståelsen. Vi tittar i koden tillsammans och läraren försöker förstå vad som är problemet. Däremot vägrar läraren att leta i koden för: ”Jag ska inte behöva leta i din kod.” Sökandet fortsätter men läraren inser snart att tiden inte räcker till och föreslår då olika sätt för eleven att komma vidare om än inte metodiskt.

Genomgång av metoden

Lärare 1 menar att det är nödvändigt att uppfylla momenten i metoden för att klara av mer komplicerade program. ”Bara för man kan se vilka pusselbitar som passar ihop.” Det betyder dock inte att eleverna gör på det viset. De förstår inte att ett tecken vänt på fel håll kan ställa till problem i programmet. ”Ja och vad gör det för skillnad? Det ska väl inte göra någon skillnad.” Det betyder i sin tur att de inte klarar av att rätta ens elementära fel och tappar motivationen då de inte kommer framåt. Läraren påpekar också hur viktigt det är att göra koden läslig och överskådlig samt att kommentera syftet med programmet och de olika kodavsnitten. Eftersom programmering är väldigt detaljerat kan det vara svårt att förstå helheten om koden inte är strukturerad. Förståelsen för detaljerna ökar om man kan sätta in dem i ett sammanhang.

Efter genomgång av metoden säger lärare 2 att: ”...jag är helt överens med dig...” men ”...det är jättesvårt att få eleverna att inse det...”. Vidare kommenterar läraren hur viktigt det är och att eleverna ofta gör samma typ av fel många gånger innan de inser att: ”Oj vad enkelt det blev när man gör som du säger!”. Eftersom läraren i princip hade nämnt alla momenten i min metod under genomgången av felsituationerna, diskuterade vi nyttan av att ha en metod på papper. Läraren

efterlyser något som man kan visa. ”Det är så här vi ska arbeta.” Läraren menar vidare att det är lätt att glömma påpeka saker. Till exempel menar läraren att: ”..när man har kört en kurs många gånger så tycker man det är självklart att man ska formattera på ett visst sätt. Så man säger inte det till eleverna. Man tycker att det borde ni väl veta. Det har vi ju hållit på med i flera år. Men det är ju jag som har hållit på med det i flera år inte eleverna.” Vidare framkommer att eleverna ofta bara provar sig fram planslöst. När de sedan eventuellt får det att fungera har de varken förstått varför det fungerar eller varför det blev fel. Läraren tror också att de skulle få ett större intresse för programmering om de hade lättare för att själva komma vidare med sina fel. ”Får man dom verkligen att jobba så här så kanske dom får en annan förståelse och då kommer nog intresset på ett annat sätt.”

5 Diskussion

Att det blir fel när man programmerar är svårt att undvika. Därför vore det bra om man kunde utnyttja misstagen i undervisningen. Då många fel beror på missuppfattningar av antingen modellen av datorn eller av uppgiften, måste vi redan på ett tidigt stadium inrikta oss på att bygga upp en hållbar modell. Ett planlöst sökande efter fel som man varken förstår orsaken till eller lösningen av, medför osvikligen tappad motivation. Tyvärr går det att komma alldeles för långt med klena konceptuella modeller, det vill säga utan att ha begreppen klart för sig och hur helheten ser ut. Det kanske inte visar sig som fel med det samma men det kan bli mycket svårupptäckta (och svårkorrigerade) fel långt senare. Det är inte heller ovanligt att man kringgår ett fel i stället för att leta reda på och rätta grundorsaken. Detta brukar också straffa sig i längden. Därför är det viktigt att undervisningen i programmering tar upp felsökning och felkorrigering på ett riktigt sätt.

Att eleverna inte från början har en hållbar konceptuell modell av datorn är inte så konstigt. Det finns inget intuitivt med datorer. Olika programmeringsspråk hanterar samma sak på olika sätt. Ett språk initierar variabler när de deklareras medan andra inte alls gör det utan lämnar initieringen till programmeraren. Ben-Ari (2001) menar till exempel att vid tilldelning av variabler tror elever att den tilldelande variabeln töms på sitt värde (som en låda). Det finns inget som säger att det inte skulle kunna vara så, och hur ska då en nybörjare i programmering kunna förstå något annat?

Vidare tar Ben-Ari (2001) upp bricolage och minimalism. Precis som det i intervjuerna kom fram att mer grafiska språk ökade motivationen för att man ser snabba resultat, kan man i viss utsträckning tillåta bricolage och minimalism. Detta bör dock göras under kontrollerade former för att eleven inte ska bygga upp felaktiga modeller. Speciellt kan man få se upp med starka elever som ofta har lyckad erfarenhet av att programmera ostrukturerat och är svårmotiverade att göra på annat sätt. Det kan blir den (o)ordning vi får leva med då det ofta är eleverna som går vidare (och sätter normen).

Intervjuresultatet stämmer bra överens med Bruce m fl (2004) fenomenografiska studie. Vissa elever är bara där för att få betyg mot minsta möjliga ansträngning, andra deltar redan aktivt i programmerarens underbara värld.

När man väl vet om det i huvudsak är en fråga om motivation eller kunskap (eller både och), kan man lättare gå vidare för att se hur man kan hjälpa eleven på bästa sätt. Ska vi som lärare kunna hjälpa eleverna till hållbara modeller, måste vi förstå just den elevens förutsättningar och anpassa oss efter dem. Vilket är precis vad styrdokumenten strävar efter. Med en metod för att hjälpa eleven till självhjälp, blir det lärarens roll att visa eleven på en lämplig nivå och att kontinuerligt validera att eleven skapar den hållbara modell som vi är ute efter. Därmed kan mer tid läggas till uppbyggnaden av dessa modeller snarare än att planlöst leta efter samma fel om och om igen. Som lärare gäller det också att hitta balansen mellan de motivationshöjande konkreta uppgifterna och den teoretiska bakgrunden. Jag finner det anmärkningsvärt att så få läromedel tar upp detta moment. De som har tagit upp felsökning har inte gjort det i någon större utsträckning. Enligt min mening borde det vara ett eget kapitel med sina egna, speciella övningsuppgifter som ger eleverna en adekvat träning och läraren ett stöd för undervisningsmomentet.

Det hade varit önskvärt att ta del av fler lärares praktiska arbete ute på fältet. Intervjuer med endast två lärare kan förstås vara väldigt missvisande. Därför bör man vara försiktig med de slutsatser som dras utifrån intervjumaterialet.

Sammanfattningsvis kan man nog ändå säga att trots att kursplanerna explicit har felsökning som kursmål och trots att så stor del av lektionstiden går åt till felsökning, har det väldigt lite stöd i kurslitteraturen. Av intervjuerna framkom att lärarna gärna skulle arbeta mer med felsökning men saknar en metod för det.

6 Slutsats

Syftet med undersökningen var att ta reda på hur felsökning inom programmering tas upp i undervisningen på gymnasiet och vad det finns för stöd för det som moment i undervisningen. De frågor jag ställde var:

- Hur tas felsökning upp i gymnasieutbildningens kursplaner?
- Hur ser stödet ut för felsökning i läromedlen?
- Hur arbetar verksamma lärare med felsökning som moment i undervisningen på gymnasiet?
- Hur ser verksamma lärare på en felsökningsmetod för gymnasieelever?

Styrdokumenten är klara på den punkten och har felsökning som ett av delmålen på alla programmeringskurserna. Av tjugo undersökta kursböcker finns det bara några få som berör felsökning. De intervjuade lärarna arbetar aktivt med felsökning då det dyker upp på lektionerna. De har dock ingen egentlig metod att arbeta efter och tar inte heller upp felsökning som ett eget avsnitt i undervisningen. Ställda inför min metod anser båda att det skulle vara ett bra stöd och förhoppningsvis öka motivationen för eleverna genom att de kan hjälpa sig själva att komma vidare. Även för läraren själv skulle det vara ett stöd då de får ett material att bygga undervisningsmomentet på.

7 Referenser

Abelli, B. (2004). *Programmeringens grunder*. Lund: Studentlitteratur.

Andersen, I. (1998). *Den uppenbara verkligheten: Val av samhällsvetenskaplig metod.* Lund: Studentlitteratur.

Ben-Ari, M. (2001). Constructivism in computer science education. *Journal of Computers in Mathematics and ScienceTeaching 20*(1), 45-73.

Bierce, A. (1985). *The Enlarged devil's Dictionary.*Harmondsworth, Middlesex, England: Penguin Books Ltd.

Bruce, C., Buckingham, L., Hynd, J., mcMahon, C., Roggenkamp, M., Stoddley, I., (Editor: Knight, L.). (2004). Ways of experiencing the art of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education*, 3, 143-160.

Burman, S., & Ohlsson, I. (2003). *Java: För nybörjare*. Lindesberg: SKOLelektronik.

Ek, J. (1996). *JAVA - programmering: En introduktion.* Upplands Väsby: Pagina.

Forsberg, A. (2003). *Programmering i C#*. Lund: Studentlitteratur.

Franzen, T. (2003). *JAVA från grunden.* Lund: Studentlitteratur.

Galjic, F. (2004). *Programmeringsprinciper i Java*. Lund: Studentlitteratur.

Gomm, S., Hailey Waters, S., Feng, J. (2003-2004). Testing computer programming skills: Paper and pencil versus a computer and compiler. *Journal of Educational Technology Systems, 32*(1), 123-135.

Gratte, I. (2001). *Java; Programmering A*. Stockholm: Liber.

Gratte, I. (2002). *Java; Programmering B*. Stockholm: Liber.

Gratte, I. (2003). *Grunderna i Java*. Stockholm: Liber.

Gratte, I. (2003). *Java; Programmering C*. Stockholm: Liber.

Guibert, N., Guittet, L., Girard, P. (2005). A study of the efficiency of an alternative programming paradigm to teach the basics of programming. *WCCE (World Education), IFIP. Cape Town, South Africa.*

Gyulai, N. (2002). *JAVA från A till O: Programmering, kurs A, B och C*. Solna:Ekelunds Förlag AB.

Gärdenfors, P. (2001). *IT och lärande.* Delegationen för IT i skolan. Hämtad 15 december 2006, från http://www.itis.gov.se/studiematerial/kopia/db/itl_grund/440.html.

Herskin, B. (2001). *IT-undervisning: med användaren i centrum.* Lund: Studentlitteratur.

Holm, P. (1999). *Objektorienterad programmering och Java*. Lund: Studentlitteratur.

Hrastinski, S. (2004). *Programmering B Visual Basic.NET*. Malmö: Gleerups.

Jonsson, T. (2003). *Programmering i fokus: Grundläggande programutveckling i Visual Basic.NET*. Lund: Studentlitteratur.

Linton, G., & Malmback, C. (2003). *Programmering A Visual Basic*. Malmö: Gleerups.

Lopes, F., Morais, P., (Editor: Cohen, E., Cope, C.). (2002). Lessons learned from the teaching of IS development. *Journal of Information Technology Education, 1(* 2), 103-112.

Lärarförbundet (2005). *Lärarens handbok: Läroplan för det obligatoriska skolväsendet, förskoleklassen och fritidshemmet.(Lpo 94)*. Stockholm; Lärarförbundet.

Lärarförbundet (2005). *Lärarens handbok: Läroplan för de frivilliga skolformerna.(Lpf 94)*. Stockholm; Lärarförbundet.

Nilsson, U., & Silborn, B. (2001). *Programmering A Java*. Malmö: Gleerups.

Silborn, B. (2003). *Programmering B Java*. Malmö: Gleerups.

Skansholm, J. (2005). *Java direkt med Swing*. Lund: Studentlitteratur.

Skansholm, J. (2003). *Java – första steget*. Lund: Studentlitteratur.

Skansholm, J. (2004). *Java – andra steget*. Lund: Studentlitteratur.

Säljö, R. (2005). *Lärande & kulturella redskap: Om lärandeprocesser och det kollektiva minnet.* Stockholm: Norstedts Akademiska Förlag.

Utbildningsdepartementet (2002), Gymnasieskolan, Kursplaner.
Hämtad 15 december 2006, från
http://www3.skolverket.se/ki03/front.aspx?sprak=SV&ar=0708&infotyp=2&skolform=21&id=0&extraId=

Van Der Vyver, G., Lane, M. (2003). Using a team-based approach in an IS course: An empirical study. *Journal of Information Technology Education, 2*, 393-406..

8 Bilagor

8.1 Bilaga 1 - Felsökningsmetod

Det flesta av stegen nedan är det bäst att göra från början när ett program ska skrivas. Är det inte gjort eller man tar över ett program från någon annan är det ett bra sätt att lära känna programmet genom att utföra dem. Det blir ett sätt att bringa struktur både i programmet och sina tankar, vilket brukar underlätta förståelsen. Det uppmuntrar också till reflektion över vad programmet egentligen gör och hur det görs.

Formattering av program

Indragningar brukar göras på så sätt att man lätt ser vilka kodavsnitt som hänger ihop eller är på samma nivå.

Exempel:
```
if (pos = 1) {
    // de satser som ska utföras om villkoret är sant dras in en bit
} //end if (pos = 1)
```

Samla variabeldeklarationer i början av ett program eller i början av en modul / metod / subrutin[4].

Samla moduler på att ställe i programmet och gärna i slutet. Låt deklarationer och liknande börja programmet följt av huvudflödet och sedan moduler.

Skilj olika moduler åt med en kommentarsrad som bara består av tex streck så är det lättare att se var de börjar och slutar.

Kommentering av program

Det är bäst att kommentera från början när programmet skrivs. Ett avsnitt man väntar med att skriva till senare kan kommenteras där det hör hemma även om inte koden finns än. Alla kommentarer bör vara tydliga men korta och enkla att ändra vid uppdatering av programmet. Felaktiga kommentarer är nog sämre än inga alls.

I början av programmet skriver man kort vad programmet gör med eventuella parametrar.

Alla variabler som det inte är uppenbart för alla vad de används till (därför att de har förklarande namn) bör kommenteras vid deklarationen.

Alla moduler, klasser eller liknande bör kommenteras med kort förklaring.

Huvudflödet i programmet bör kommenteras där det börjar.

[4] En modul är förenklat (liksom metod och subrutin) en benämning på ett program i programmet. Det finns även andra benämningar av likartade fenomen. I fortsättningen kommer jag att använda benämningen modul, men motsvarande gäller även för de andra benämningarna.

Alla kodavsnitt som inte är väldigt uppenbara bör kommenteras.
Där något avslutas (tex loop, villkor, modul, klass) skriver man explicit vad som avslutades i en kommentar.

Exempel:
```
if  (pos = 1) {
    //  här följer ett antal satser som ska utföras om villkoret är sant
} //end if (pos = 1)
```

Namnstandard / versionshantering

Försök hitta en enkel namnstandard för dina bibliotek / mappar / filer / program. Antalet växer snabbt och det är inte lätt att veta vad som ”test1”, ”test2” eller ”test3” egentligen gör. Lättare är det då tex att ta reda på vad program ”Uppgift1-3” som läggs i bibliotek ”ProgrammeringB” gör. (Det är bara att titta i boken.) Namn som ”MyProgram” som man dessvärre ofta ser i kursböcker, talar inte heller om vad programmet är till för. Snart har man många egna program som ska skiljas åt.
Arbetar man med större dokument eller program, kan det vara lämpligt att införa en manuell versionshantering, så att man direkt kan se vilken version som används. För program som används av andra program (och alltså måste heta på ett särskilt sätt) kan man versionsnamnge den gamla versionen. Då kan aktuell version heta samma sak hela tiden. Har stora förändringar gjorts som man behöver ta tillbaka, kan man lätt få tag på en tidigare version genom namnbyte.
Allt som ska namnges, från bibliotek via program och klasser till variabler, bör få beskrivande namn.

Modularisering av program

Ett program bör delas upp i moduler så att inget överstiger 1-2 skärmsidor. Huvudflödet bör inte heller överstiga denna storlek. På så sätt ökar överskådligheten och därmed förståelsen.
Modularisering underlättar också felsökning då det är lättare att kontrollera resultatet av en mindre mängd kod. Genom huvudflödet ska logiken framgå.

Flödesschema

Ett flödesschema (se exempel i bilaga) ska beskriva händelseförloppet av ett program. det kan göras på olika detaljnivå alltefter behov. Finns inte ett flödesschema på den detaljnivå man behöver, kan det skrivas i efterhand för att underlätta förståelsen och eventuell upptäckt av logiska brister i programmet.

Utskrift av mellansteg i program

Utskrift från ett program under tiden det körs kan göras på olika sätt och med olika syften.
Man kan dels skriva ut att man befinner sig på ett speciellt ställe i programmet, dels kan man skriva ut innehållet i en variabel. Att skriva ut var man befinner sig kan göras för att säkerställa att man passerar ett vissa ställen. Ett effektivt sätt är att lägga in en utskrift av modulnamnet så fort programmet kommer in i modulen. Då kan man kontrollera att så verkligen sker (eller konstatera att det inte sker). Det ger en god bild av flödet i programmet. På liknande sätt är det lätt att kontroller om programmet verkligen kommer in i vissa avsnitt som till exempel föregås av ett villkor.
Att skriva ut mellanresultat av variabler är också ett effektivt sätt att försäkra sig om att man befinner sig på rätt spår eller att det nu har blivit fel någonstans.
Vissa av dessa utskrifter kan man låta finnas kvar i programmet, men kommentera bort dem, för att kunna använda sig av dem vid senare behov.

Uteslutning av kod (bortkommentering**)** i program

Man kan vid behov tillfälligt kommentera bort en modul för att begränsa sin felsökning.
Dels kan det göras med en modul som man vet fungerar och inte behöver titta på för tillfället (och som då inte har någon väsentlig betydelse för exekveringen av programmet). Dels kan det göras för att utesluta en modul som man vet inte fungerar för att kontrollera om resten fungerar. På så vis är det ganska lätt att ringa in problemet till ett begränsat avsnitt som kan undersökas i detalj.

Skrivbordstest av program

Ibland kan det underlätta att sätta sig med en utskriven version av programmet och själv gå igenom instruktion för instruktion, samtidigt som man antecknar mellanresultat av variabler.

Strukturering av bibliotek/mappar, filer och program

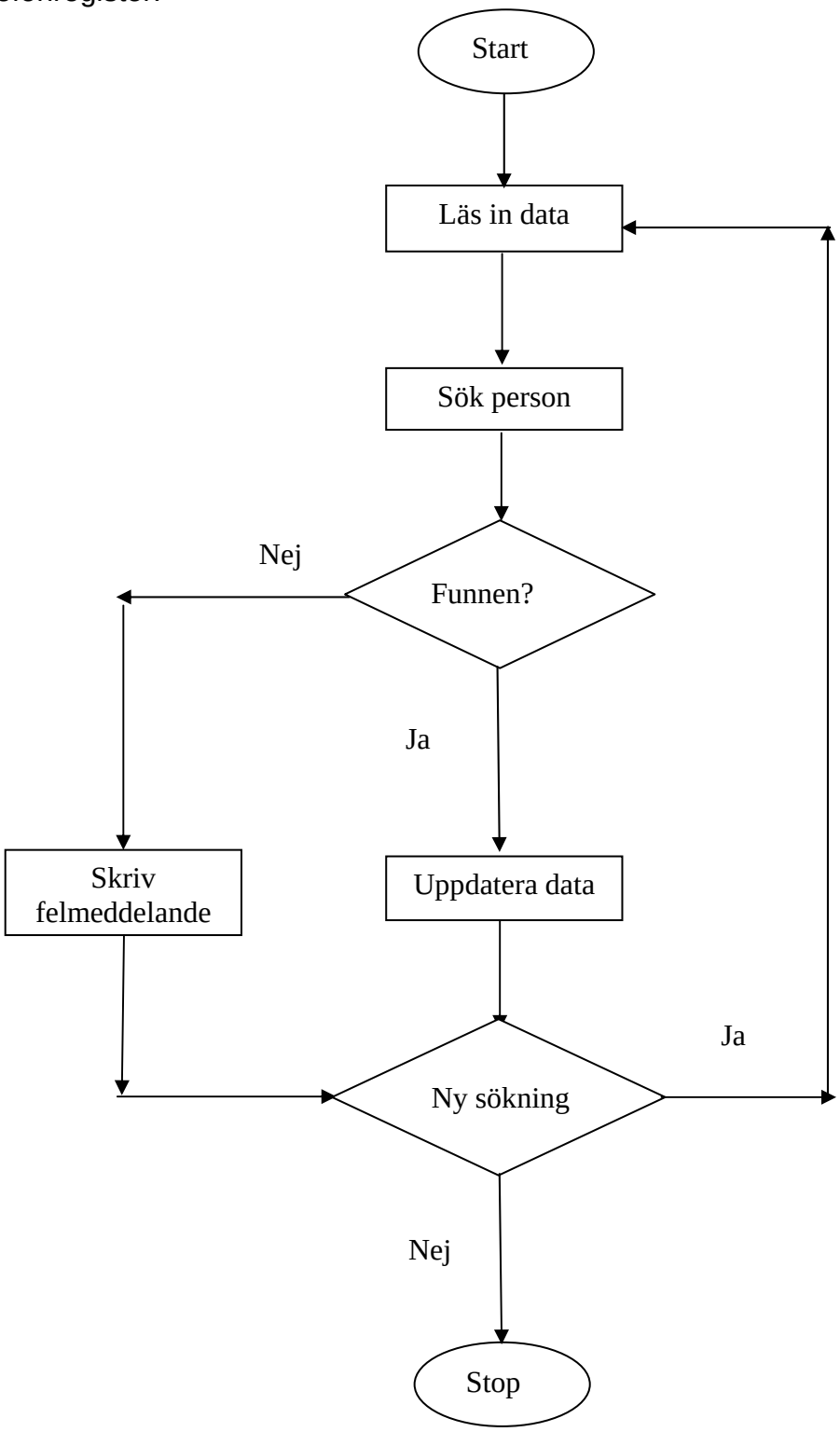
Försök hitta en hierarkisk organisation av bibliotek, filer och program så att:

- de är lätta att hitta och veta vad de innehåller
- relativt lätt kan utökas (utan total omstrukturering varje gång)
- det är lätt att knyta ihop program med filer (och eventuellt andra program) det använder sig av.

Anpassa strukturen för aktuellt behov, men med tanke på en viss flexibilitet för utvidgning omorganisation.

8.2 Bilaga 2 - Flödesschema

Exempel på ett enkelt flödesschema för till exempel uppdatering av ett telefonregister.



8.3 Bilaga 3 - Intervjuunderlag

Moment /stöd

- Tar du upp felsökning som ett speciellt avsnitt i undervisningen? Om ja – hur?
- Finns felsökningsstöd;
    - i den lokala kursplanen ?
    - i läroböckerna?
    - i bimaterial (lärarstöd och dylikt)?
    - någon annanstans?
- Är felsökning något ni pratar om kolleger emellan?
- Diskuterar ni andra kursmoment?

Metod

- Vad gör du när eleverna har fått ett fel?
- Vad gör eleverna när det blir fel?
    - Frågar läraren direkt
    - Börjar felsöka själva; Om ja – hur?
    - Inget / tappar motivationen
    - Annat
- Angriper du / eleverna felsökningen efter ett speciellt mönster / metod? Om ja – hur?
- Drar du / eleverna några slutsatser efter (en) felsökning (tex indragningar är bra, kommentering, flödesschema)?
- Hur reagerar eleverna på felen? (tappar motivationen och ger upp ==> sporras)

Felen

- Är det samma typ av fel som upprepas? Om ja – vilken typ?
- Vilken typ av fel verkar vanligast?
- Vad beror de på?
- Vem löser oftast felet – läraren / eleven / elevkompis?
- Får de många fel?

8.4 Bilaga 4 – Undersökta läromedel

- Abelli, B. (2004). *Programmeringens grunder*.
- Burman, S., & Ohlsson, I. (2003). *Java: För nybörjare*.
- Ek, J. (1996). *JAVA - programmering: En introduktion.*
- Forsberg, A. (2003). *Programmering i C#*.
- Franzen, T. (2003). *JAVA från grunden.*
- Galjic, F. (2004). *Programmeringsprinciper i Java*.
- Gratte, I. (2001). *Java; Programmering A*.
- Gratte, I. (2002). *Java; Programmering B*.
- Gratte, I. (2003). *Grunderna i Java*.
- Gratte, I. (2003). *Java; Programmering C*.
- Gyulai, N. (2002). *JAVA från A till O: Programmering, kurs A, B och C*
- Holm, P. (1999). *Objektorienterad programmering och Java*.
- Hrastinski, S. (2004). *Programmering B Visual Basic.NET*.
- Jonsson, T. (2003). *Programmering i fokus: Grundläggande programutveckling i Visual Basic.NET*.
- Linton, G., & Malmback, C. (2003). *Programmering A Visual Basic*.
- Nilsson, U., & Silborn, B. (2001). *Programmering A Java*.
- Silborn, B. (2003). *Programmering B Java*.
- Skansholm, J. (2005). *Java direkt med Swing*.
- Skansholm, J. (2003). *Java – första steget*.
- Skansholm, J. (2004). *Java – andra steget*.