Testning

1. Inledning

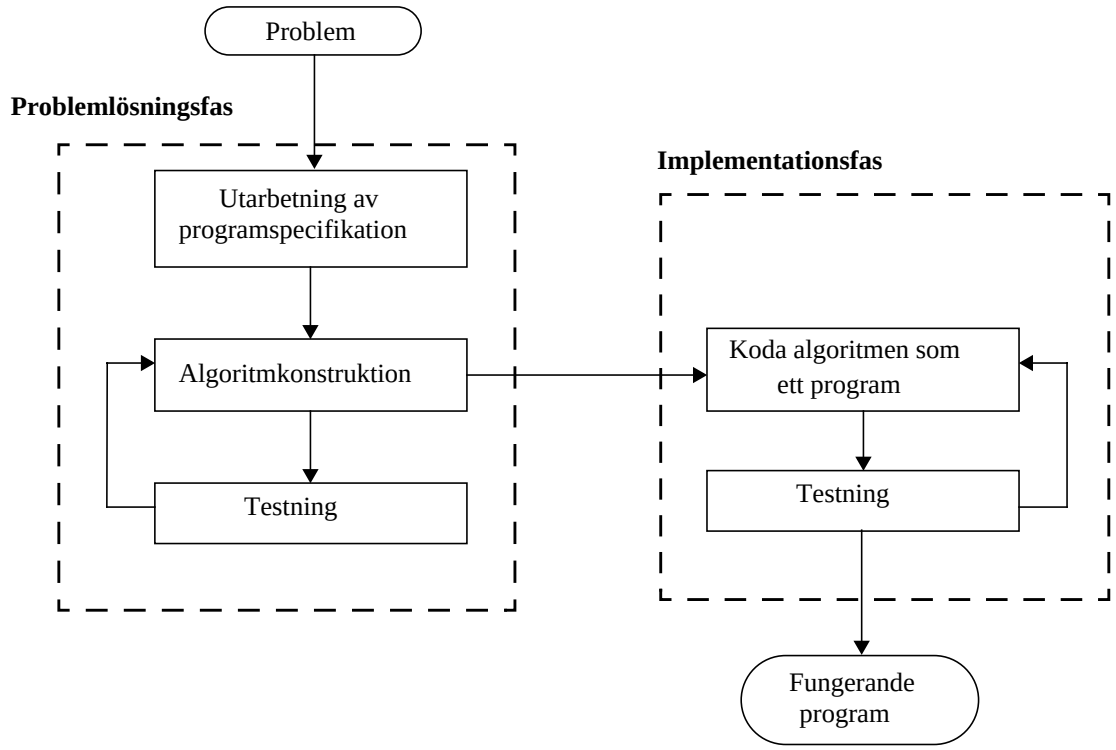
I all ingenjörsmässig verksamhet är testning en vedertagen metod för att fastställa om en hypotes, konstruktion eller produkt är korrekt och fungerar som avsett.

Datorprogram är ofta mycket komplexa företeelser och är därmed svåra att få att fungera korrekt. Testning är således en mycket viktig aktivitet under utvecklingen av datorprogram. Vissa uppskattningar gör gällande att cirka halva den totala tiden för ett programmeringsprojekt åtgår för att testa programmet och korrigera de fel som upptäcks.

Testning är ett gemensamt namn för en uppsättning tekniker som syftar till att verifiera att ett program verkligen gör vad det är tänkt att göra, dvs att programmet uppfyller sin specifikation. Till grund för all testning ligger således programspecifikationen. En dålig eller felaktig specifikation leder naturligtvis till ett undermåligt eller felaktigt program, oavsett hur utförligt man testar programmet.

Testning är ett sätt att minimera antalet fel i ett program. En viktig insikt är dock att *testning enbart kan påvisa förekomsten av fel, aldrig frånvaron av fel*! Om programmet ger riktiga resultat för en viss uppsättning testdata, kan vi endast dra slutsatsen att programmet är korrekt för just denna uppsättning testdata. Det finns dock inga garantier för att programmet är felfritt, utan enbart att våra tester inte kunnat påvisa några fel. Programmet kan därför mycket väl ge felaktiga resultat för andra uppsättningar av indata. Det är därför mycket viktigt att man väljer sina testdata med stor omsorg för att kunna fånga så många av felen som möjligt under testningen.

Under programutvecklingsarbetet skall testning göras både under problemlösningsfasen och implementationsfasen. Testning är en aktivitet som skall ses som en integrerad del i utvecklingsarbetet, för vilken både tid och resurser skall avsättas, och inte som något man sparar till sist och gör i mån av tid och lust.



Figur 1. Modell för programutveckling (perfekt).

Tyvärr har många studenter svårt att inse vikten av att göra rigorösa testningar och att ha en underbyggd strategi för hur testningen skall utföras. De tror att när de väl lyckats kompilera sitt program och kan exekvera det, att de i stor sett är klara med sitt arbete. Men detta är dock långt i från sanningen, mycket arbete återstår innan man kan verifiera att programmet fungerar som det var tänkt.

Syftet med dessa anteckningar är att påpeka vikten av att testa sina program, samt att presentera ett antal olika tekniker för hur man genomför testning på ett effektivt och framgångsrikt sätt.

2. Testning av det färdiga programmet

Syftet med testning är att övertyga sig om att det färdiga programmet gör det som utlovas i programspecifikationen. Vid arbetet med programspecifikationen är det därför lämpligt, förutom att specifiera vad programmet skall göra, att även utarbeta en *testplan* som anger hur det färdiga programmet skall testas. Testplanen ligger till grund för *leveranstestning*, dvs de test som programmet måste genomgå och passera för att kunna tas i drift.

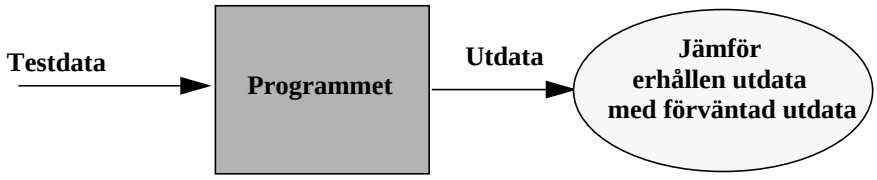
En testplan är en strategi för att övertyga sig själv och andra om att programmet ger korrekta resultat. När kommersiella programprodukter utvecklas är det beställaren som utarbetar testplanen och bestämmer vilka testfall som programmet måste klara av för att programmet skall bli godkänt för leverans. Om beställaren inte utvecklar en lämplig testplan utan istället gör planlösa tester och kanske t.o.m. överlåter till leverantören att demonstrera sin produkt med tillrättalagd indata, är risken stor att många viktiga testfall förbises. Effekten blir att situationer, som borde blivit testade innan leverans, dyker upp och orsakar oväntade fel när programmet har tagits i drift.

När man läser en första kurs i programmeringsteknik utvecklar man inga kommersiella program och man har ingen beställare till de program man utvecklar. Därför har man heller ingen beställare som specificerar en testplan för programmet, utan man får utarbeta testplanen själv. Testplanen skall utarbetas parallellt med programspecifikationen och är en viktig del av problemanalysen. Vid arbetet med testplanen är det lämpligt att man låtsas vara en blivande köpare av programmet och frågar sig själv: "Vilka tester kräver jag för att bli övertygad om att programmet verkligen beter sig som säljaren påstår?".

2. 1. Black-box testning

Vid arbetet med programspecifikationen - då testplanen som ligger till grund för att testa det färdiga programmet skall utarbetas- finns ingen kännedom om hur själva implementeringen av det färdiga programmet ser ut. Den enda kännedom som finns är vad programmet skall kunna göra, dvs vilken funktionallitet programmet skall ha.

Testmetoder där man inte har någon kännedom om hur programmet är implementerat kallas för *black-box testning*. Vid denna form av testning är det endast programmets in- och utdata som beaktas. För den som utarbetar testfallen, som skall ingå testplanen, är programmet en *svart låda* (*eng: black-box*) vilken skall tillhandhålla vissa funktioner. Uppgiften är att försöker finna en uppsättning testfall som visar att programmet har den funktionallitet som utlovas. Vid själva testningen körs programmet för var och ett av de fastställda testfallen och resultatet från körningarna jämförs med det förväntade resultatet för respektive testfall. Överstämmer det erhållna resultatet med det förväntade resultatet för samtliga testfall godkänns programmet för leverans. I annat fall måste felen i programmet rättas till och samtliga testfall göras om på nytt. Det är nödvändigt att alla testfallen återupprepas när ett observerat fel blivit åtgärdat, eftersom de ändringar som gjorts i programmet för att rätta till det upptäckta felet kan introducera nya fel som kan få konsekvenser för tidigare genomförda testfall.



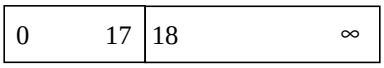
Figur 2. Black-box testning

En strategi för att välja vilka testfall som skall utföras på ett program skulle naturligtvis kunna vara att ta med alla möjliga indatauppsättningar som kan ges till programmet, d.v.s. att göra en s.k. *uttömmande testning*. Detta är dock en praktisk omöjlighet. Betrakta ett sådan trivialt program som att läsa in och beräkna summan av en serie om maximalt 100000 heltal. Antalet tal som ingår i serien kan således variera från 0 upptill 100000 tal. Varje enskilt tal i serien kan dessutom ha värden mellan Integer.MAX_VALUE och Integer.MIN_VALUE. Tillsammans ger detta en ofantlig mängd olika möjliga kombinationer på indata till programmet. Därför är det således ogörligt att genomföra testkörningar på samtliga dessa indatauppsättningar.

Eftersom uttömmande testning är utesluten, måste man med användning av ett begränsat antal testfall demonstrera programmets frånvaro av fel på ett så övertygande sätt att man kan *anta* att programmet är korrekt. Då testning är en mycket tidsödande aktivitet är det dessutom viktigt att antalet testfall som erfordras för att komma till denna övertygelse kan minimeras. Det är således mycket viktigt att man väljer sina testfall med stor omsorg.

Den metod som används för att bestämma vilka testfall som behöver genomföras för att bli övertygad om att programmet är korrekt, bygger på att indela de möjliga indatavärdena till programmet i ett antal olika *ekvivalens kategorier*. Inom varje ekvivalens kategori samlas de indatavärden som förväntas ge likartad utdata och som testar samma egenskaper hos programmet. När man bestämmer vilka testfall som skall ingå i en testplan är det därför tillräckligt att ett eller ett par värden ur varje ekvivalens kategori medtages.

Vi illustrerar indelningen i ekvivalens kategorier med hjälp av ett mycket enkelt exempel. Antag att vi har ett program som läser åldern för en person och skriver ut om personen får rösta eller inte. Röståldern är 18 år. I detta fall kan vi indela de möjliga indatavärdena till programmet i två ekvivalens kategorier - en kategori som innehåller värdena 0 till 17 och en kategori som innehåller värdena 18 till ∞. Anledningen är att vi för samtliga värden inom den första kategorin förväntar oss att erhålla utdatan "får inte rösta" från programmet och för den andra kategorin förväntar oss få utdatan "får rösta". Den förväntade utdatan inom varje ekvivalens kategori är således lika, medan den förväntade utdatan för varje ekvivalens kategori är olika. Vi kan rita följande diagram över ekvivalens kategorierna:



För att testa de olika ekvivalens kategorierna räcker det med två testfall, t.ex.

Test nr	Indata	Förväntat resultat
1	12	Får inte rösta
2	21	Får rösta

För att bli övertygad om att programmet är korrekt räcker dock inte dessa testfall. Testfallen undersöker inte den *kritiska punkten* vid gränsen mellan de båda kategorierna. Det är just vid denna gräns, där det är en övergång från ett tillstånd till ett annat, som det är stor risk att programmeraren kan göra fel. Därför behövs även följande testfall:

Test nr	Indata	Förväntat resultat
3	17	Får inte rösta
4	18	Får rösta

Därmed har vi i princip vår testplan klar. Det finns dock ytterligare en ekvivalens kategori som vi ännu inte har beaktat, nämligen kategorin som innehåller *ogiltig* data. Ett programs beteende skall i så stor utsträckning som möjligt vara *förutsägbar*, även då ovana användare beter sig felaktigt åt vid användningen av programmet. Testplanen skall således även innehålla testfall som täcker in felaktig användning av programmet, t.ex. oväntade tangentryckningar på tangentbordet. Om det av programspecifikationen inte framgår hur ogiltig data skall hanteras, kommer detta att upptäckas under utarbetningen av testplanen och kan därmed åtgärdas tidigt i utvecklingsfasen.

I vårt exempel kan en användare ge en negativ ålder på en person eller ge indata som överhuvudtaget inte är ett heltal, varför vi även skall inkludera testfallen:

Test nr	Indata	Förväntat resultat
7	-10	Felutskrift
6	0	Får inte rösta
7	-1	Felutskrift
8	xx	Felutskrift

Vår kompletta testplan kommer därför att innehålla följande testfall:

Test nr	Indata	Förväntat resultat
1	12	Får inte rösta
2	21	Får rösta
3	17	Får inte rösta
4	18	Får rösta
7	-10	Felutskrift
6	0	Får inte rösta
7	-1	Felutskrift
8	xx	Felutskrift

Denna uppsättning testfall är tillräcklig för att bli övertygad om att programmet fungerar korrekt.

Sammanfattning:

För att få en lämplig uppsättning testfall skall man förfara enligt nedan:

1. Samla indata med samma egenskaper i olika ekvivalens kategorier.
2. Välj en representant från varje ekvivalens kategori att ingå i testfallen.
3. Ta med ett testfall för varje värde som utgör en gräns mot en annan ekvivalens kategori.
4. Ta med testfall som innehåller ogiltig data.

3. Testning av algoritmer

När man utvecklat en algoritm skall algoritmen givetvis testas innan man tar itu med att koda algoritmen som ett program. Gör man inte denna testning kommer eventuella fel i algoritmen att finnas kvar under implementationsarbetet, vilket leder till att tid och arbete läggs ner som inte kommer till någon nytta. Ju senare i utvecklingsarbetet man upptäcker ett fel ju svårare och dyrare är det att lokalisera och korrigera felet. Därför skall man lägga ner stora ansträngningar på att finna eventuella fel redan i algoritmkonstruktionen. Kan ett fel konstateras under algoritmkonstruktionen istället för under arbetet med implementationen sparas mycket tid och arbete. Varje timme som man lägger ned på testning under algoritmkonstruktionen tar man mångfalt igen vid testningen under implementationsarbetet.

Att testa en algoritm tillgår på så sätt att man manuellt går igenom algoritmen och utför de olika algoritmstegen. De hjälpmedel som behövs är papper och penna. Syftet är att verifiera att algoritmen inte innehåller några *logiska fel*, dvs. att man inte gjort något tankefel när man konstruerat algoritmen.

Under testning av algoritmen kan det visa sig att programspecifikationen är felaktig eller ofullständig. I sådana fall måste man naturligtvis backa tillbaks till programspecifikationen och göra nödvändiga modifieringar eller kompletteringar.

4. Testning av kod

När man konstruerat och testat sin algoritm är det dags att börja med kodningsarbetet och översätta algoritmen till ett program. Sedan översättningen är gjord är nästa steg att kompilera programmet. Under kompileringen upptäcker kompilatorn sådana fel som handlar om att man använt konstruktionerna i programspråket på ett felaktig sätt. Programmeraren har helt enkelt inte följt programspråkets grammatik, varför kompilatorn inte förstår vad programmeraren menar. Denna typ av fel kallas för *kompileringsfel*. Vid ett kompileringsfel erhålles felmeddelanden från kompilatorn och de är vanligtvis triviala att avhjälpa.

Alla språk, även programspråk, har en *syntax* och en *semantik*, varför kompileringsfelen i sin tur kan indelas i *syntaktiska fel* och *semantiska fel*. Syntaxen anger *hur* konstruktionerna i språket ser ut och semantiken anger konstruktionernas *betydelse*. Typiska exempel på syntaktiska fel är att man glömt att avsluta en sats med ‘;’ eller att

man i en jämförelse skrivit ‘=’ istället för ‘==’. Vanligt förekommande semantiska fel är att man glömt att deklarera variabler som används i programmet, att man glömt initiera ett värde till en variabel eller att man försöker tilldela en variabel av en viss datatyp ett värde av en annan datatyp.

När programmet inte innehåller några kompileringsfel kan kompilatorn skapa ett exekveringsbart program. Detta betyder dock inte att programmet är korrekt i den meningen att programmet verkligen gör det som programmet var tänkt att göra. Det enda det betyder är att det finns ett program som går att exekvera.

I ett exekveringsbart program kan det förekomma två typer av fel - *exekveringsfel* och *logiska fel*. Dessa typer av fel uppstår då programmet körs och kan inte upptäckas under kompileringen.

Ett exekveringsfel uppkommer på grund av att det någonstans i programmet under exekveringen sker en evaluering av ett uttryck som resulterar i att ett värde erhålls som ligger utanför det giltiga definitionsområdet för uttryckets datatyp. Felet uppträder vanligtvis inte varje gång programmet körs utan endast då vissa specifika indatavärden ges till programmet.

Exempel 1 Betrakta nedanstående programsatser

```
double tal1, tal2, res;
. . .
String indata = JOptionPane.showInputDialog("Ge första talet: ");
tal1 = Double.parseDouble(indata);
indata = JOptionPane.showInputDialog("Ge andra talet: ");
tal2 = Double.parseDouble(indata);
res = tal1 / tal2;
```

Om vi läser in värdet 12.5 till variabeln tal1 och värdet 2.5 till variabeln tal2 kommer uttrycket tal1 / tal2 att evalueras till 5.0 och allt fungerar som vi tänkt oss. Om vi däremot läser in värdet 12.5 till variabeln tal1 och värdet 0.0 till variabeln tal2 kommer det i uttrycket tal1 / tal2 att inträffa en division med 0, varför resultatet av uttrycket beräknas till ∞ och därmed blir större än det största tal som kan lagras i datatypen **double**.

Exempel 2: Betrakta nedanstående programsatser

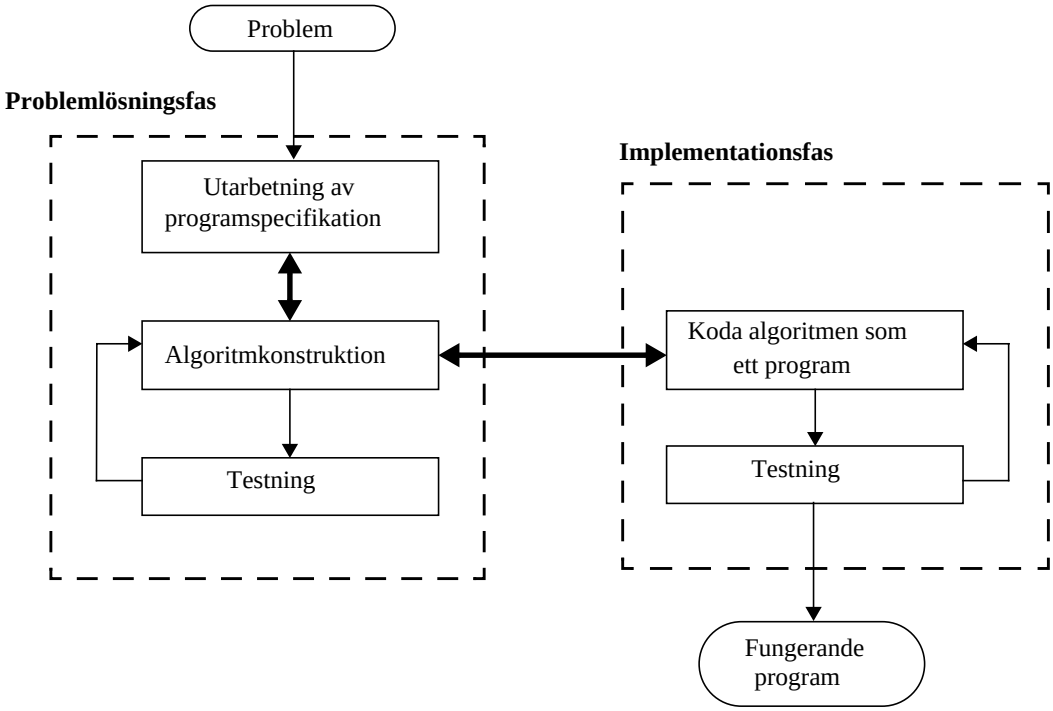
```
int [] falt = new int[10];
int antal;
. . .
String indata = JOptionPane.showInputDialog("Ge antalet tal: ");
antal = Double.parseDouble(indata);
for (int i = 0; i < antal; i = i + 1) {
      indata = JOptionPane.showInputDialog("Ge tal nr " + (i+1) + ": ");
     falt[i] = Double.parseDouble(indata);
}
```

Om vi läser in ett värde som är mindre eller lika med 10 till variabeln antal fungerar programmet som vi tänkt oss. Om vi däremot läser in ett värde som är större än 10 till variabeln antal kommer ett exekveringsfel att inträffa vid satsen falt[i] = Double.parseDouble(indata) när vi genomlöper det 11:e varvet av **for**-satsen, eftersom den styrande variabeln i då har värdet 10 och det i fältet falt inte finns något fältelement med index 10.

När ett exekveringsfel inträffar och felet inte fångas upp som en exceptionell händelse, kommer exekveringen av programmet avbrytas på ett onormalt sätt. Exekveringsfelen kan ibland vara ganska svåra att upptäcka, eftersom de nödvändigtvis inte uppträder vid varje programkörning. Då de inträffar får man dock felutskrifter från kompilatorn, vilket underlättar att lokalisera felen.

Logiska fel är rena tankefel hos programmeraren. Exekveringen av programmet lyckas, men programmet produ-

cerar inte den utdata man förväntat sig. Logiska fel skapas oftast redan när man konstruerar sin algoritm. Om så är fallet måste man gå tillbaka till algoritmkonstruktionen och korrigera felaktigheterna i sin algoritm. Som tidigare påpekats kan felet i algoritmen i sin tur bero på felaktigheter eller brister i programspecifikationen. När ett exekveringsfel uppkommer kan det i värsta fall betyda att programspecifikationen måste omarbetas. Därför är den modell för programutveckling som angavs i figur 1 en idealliserad modell. En mer realistisk programutvecklings modell ges i figur 3.



Figur 3. Modell för programutveckling (realistisk).

Om algoritmen är korrekt men programmet innehåller ett logiskt fel har detta uppkommit p.g.a. att algoritmen kodas på ett felaktigt sätt i programspråket. Logiska fel är många gånger mycket svåra att lokalisera, eftersom man inte får något stöd i form av felmeddelanden. Den enda indikation som erhålles är att programmet producerar felaktiga resultat eller i sämsta fall inget resultat överhuvudtaget.

Följande exempel är en utmärkt illustration av logiska fel. Rötterna till en andragradsekvation $Ax^2 + Bx + C = 0$ bestäms av formeln:

$$\frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Antag att ett kodavsnitt ur ett program för att beräkna rötterna till en andragradsekvation har följande utseende:

```
rotuttryck = Math.pow(b, 2) - 4 * a * c;
if ((rotuttryck > 0) && (a != 0)) {
    rot1 = -b + Math.sqrt(rotuttryck) / (2 * a);
    rot2 = -b - Math.sqrt(rotuttryck) / (2 * a);
}
else {
    . . .
}
```

Programavsnittet innehåller ett logiskt fel. Formeln för att beräkna rötterna har i programmet inte blivit översatt på ett korrekt sätt! Således är satserna

```
rot1 = -b + Math.sqrt(rotuttryck) / (2 * a);
rot2 = -b - Math.sqrt(rotuttryck) / (2 * a);
```

felaktiga. Orsaken är att operationen ‘/’ har högre prioritet än operationerna ‘+’ och ‘-’, varför det korrekta utseendet på satserna skall vara

```
rot1 = (-b + Math.sqrt(rotuttryck)) / (2 * a);
rot2 = (-b - Math.sqrt(rotuttryck)) / (2 * a);
```

Vid jämförelser mellan värden av datatypen **double** är det stor risk att göra fel om man inte är försiktig. På grund av lagringsprincipen för reella tal gäller t.ex. att det decimala talet 0.2 inte kan lagras exakt. Detta medför alltså att en evaluering av summan 0.2 + 0.2 + 0.2 + 0.2 + 0.2 inte blir *exakt* lika med 1.0 (men dock mycket nära 1.0). Denna egenskap gör att programavsnittet nedan är felaktigt:

```
double x = 0.0;
while (x != 2.2) {
    . . .
    x = x + 0.2;
}
```

Avsikten med programavsnittet är att bearbeta de reella värden som består av serien 0.0, 0.2, 0.4, . . . , 1.6, 1.8 och 2.0. Problemet är dock att villkoret x != 2.2 i **while**-satsen alltid kommer att vara uppfyllt. Detta betyder att vi får en evighetsloop p.g.a. att negationen för avbrottsvillkoret, d.v.s. x == 2.2, aldrig kommer att inträffa. För att programavsnittet skall bli korrekt måste vi ändra villkoret i **while**-satsen. För reella tal skall man aldrig jämföra på exakthet utan på tillräcklig noggranhet. En korrekt version av programavsnittet har följande utseende:

```
final double EPS = 0.0000000001;
. . .
x = 0.0;
while (x <= 2.0 + EPS) {
    . . .
    x = x + 0.2;
}
```

4. 1. Granskning

En metod för testning är granskning. Vid granskning körs inte programmet, utan testningen går helt enkelt till på så sätt att man manuellt läser programkoden för att försöka hitta felaktigheter. Denna metod fungerar bäst om det är en annan person än den som skrivit programmet som gör granskningen. Detta p.g.a. av att man lätt blir "blind" då man granskar något som man själv skrivit.

Vid granskning behövs tillgång till

- programspecifikationen
- en lista av programkoden.

När man genomför en granskning finns det vissa saker som är ganska enkla att kontrollera och som alltid skall utföras av granskaren mer eller mindre mekaniskt:

- instansiering av variabler
- korrekt instansiering och avslutning av loopar
- korrekta parametrar vid anrop av metoder.

En annan kontroll som skall göras är att undersöka programmets logiska uppbyggnad, d.v.s. kontrollera att de algoritmer som används i programmet är korrekta. Detta görs genom att man utför en handexekvering av programmet.

Vid granskning skall man även passa på och kontrollera att

- meningsfulla namn används på identifierare
- programmet är indenterat på lämpligt sätt
- programmet innehåller lämpliga kommentarer
- logiken i programmet är lättbegriplig och inte innehåller "smarta lösningar".

Även om det primära syftet med granskningen inte är att kontrollera programmeringsstilen, är nästan alltid en dålig programmeringsstil en "källa" till fel.

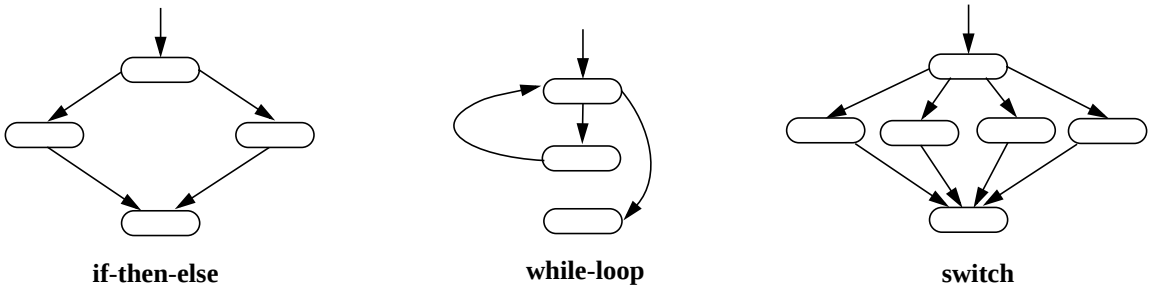
Många undersökningar har visat att granskning är en mycket effektiv metod och fångar vissa typer av fel effektivare än att exekvera programmet!

4. 2. White-box testning

När man testar ett program eller en programkomponent under implementationsfasen har man kännedom om hur implementationen är gjord. Testmetoder som drar nytta av att implementationen är känd kallas för *white-box testning*. Vid white-box testning används kunskapen om programmets strukturella uppbyggnad som utgångspunkt för valet av testdata. Det man eftersträvar vid white-box testning är att köra programmet med en uppsättning testfall som valts på så sätt att varje sats i programmet blir exekverad under testningen. Således gäller det att kartlägga de olika exekveringsvägarna som finns i programmet och försäkra sig om att man väljer en uppsättning testfall som leder till att varje exekveringsväg blir genomlöpt av minst ett testfall.

Exekveringsvägarna i ett program styrs av olika styrsatser. Alla styrsatser evaluerar villkorsuttryck och beroende på utfallet av evalueringen kommer styrsatsen att dirigera exekveringen olika vägar genom programkoden.

När man analyserar vilka olika exekveringsvägar det finns i ett program kan ett hjälpmedel vara att rita ett flödesdiagram över programmet. Ett flödesdiagram består av noder som representerar beslut och bågar som representerar flödesstyrning. Flödesdiagrammen för **if**-satsen, **while**-satsen och **switch**-satsen framgår av figur 4. För övriga styrsatser i språket kan motsvarande diagram ritas upp.



Figur 4. Flödesdiagram för **if**-satsen, **while**-satsen och **switch**-satsen.

Vid valet av testdata gäller det t.ex. för varje **if**-sats i programmet att förvissa sig om att villkoret i satsen evalueras till **true** i minst ett testfall och till **false** i minst ett testfall. Genom att täcka in båda alternativen har vi försäkrat oss om att båda exekveringsvägarna genom **if**-satsen kommer att exekveras under testningen. Motsvarande gäller för alla andra former av styrsatser i programmet.

Problemet med att testa alla möjliga exekveringsvägar är att stora program vanligtvis har ett mycket stor antal exekveringsvägar, varför det blir ett orealistiskt stort antal testfall som måste göras. För stora programmeringsprojekt blir det därför omöjligt att i praktiken göra en fullständig white-box testning.

4. 3. Avlusning

Under testning kan man påvisa att programmet innehåller fel. För att få ett korrekt program måste dessa fel avlägsnas från programmet. Processen att lokalisera och korrigera ett fel kallas för *avlusning* (*eng:debugging*).

Vissa fel kan vara mycket svåra att lokalisera. Detta gäller framför allt logiska fel, eftersom vi vid denna typ av

fel inte får något stöd från kompilatorn i form av felutskrifter. Därmed har vi i allmänhet inga egentliga ledtrådar, som ger information om var felet inträffat. Många gånger kan det bli nödvändigt att stega igenom sitt program och exekvera en sats i taget med hjälp av en *debugger*. När man använder en debugger kan man följa vilken exekveringsväg programmet tar och studera hur värdena på variablerna i programmet förändras under exekveringens gång. Det man koncentrerar sig på när man kör sitt program i en debugger är att kontrollera att exekveringen tar den förväntade vägen och att variablerna ändras under exekveringen på ett korrekt sätt. Att ett felaktigt resultat erhålls från ett program beror alltid på att exekveringen tagit en oväntad väg eller att en variabel erhållit ett felaktigt värde.

För en nybörjare kan det vara ganska tidskrävande att lära sig att använda en debugger på ett effektivt sätt. Dessutom är de program man utvecklar förhållandevis små. Ett alternativ till att använda en debugger är därför att själv lägga in kontrollutskrifter i sitt program för att följa programmets exekveringsväg och studera hur värdet av kritiska variabler förändras.

Det finns här återigen anledning att påminna om att det är nödvändigt att lägga ner stor möda när man konstruerar sina algoritmer, annars riskerar man att avlusningen av programmet blir mycket mödosamt och tidsödande.

4. 4. Testning av komponenter och system.

Mycket små program kan testas i sin helhet. Men så snart ett program blir något större och innehåller flera programenheter blir allt i alla avseenden genast mycket mer komplext, även testning och avlusning.

När ett program innehåller flera enheter, såsom underprogram och klasser, kan man prata om ett *programsystem*, där de ingående enheterna bygger upp systemet. Har man ett programsystem, är det nödvändigt att varje enhet testas separat från övriga enheter innan enheten integreras i programsystemet. Anledningen till att testa de enskilda enheterna innan man testar systemet är naturligtvis att man därigenom har större möjlighet att lokalisera och åtgärda uppkomna fel. Om ett fel hittas finns felet givitvis i den enhet som testas. Testas alla enheter samtidigt kan man troligen konstatera att det existerar ett fel, men man har ingen aning om var felet finns lokaliserat. Även om enheterna i programsystemet testas separat måste givetvis det kompletta programsystemet genomgå testning för att kontrollera att de olika enheterna samverkar på ett korrekt sätt.

Att testa en programkomponent kallas för *enhetstesting* och att testa det kompletta programsystemet kallas för *systemtesting*. Enhetstestning bör utföras med en kombination av granskning, white-box testning och black-box testning. Vid systemtestning används lämpligen black-box testning.

Att testa varje klass och metod separat kan verka vara en tidskrävande metod, men faktum är att metoden är mycket tidseffektiv. Den tid som sparas genom att ett fel kan lokaliseras snabbt innebär att den totala tiden det tar att få ett korrekt program är kortare än vad som blir fallet då hela programmet testades som en enhet.

Det finns två olika metoder för att testa de enskilda enheterna i ett programsystem, *bottom-up* och *top-down*. Dessa metoder skiljer sig i fråga om i vilken ordning programenheterna utvecklas. Vid bottom-up utvecklas och testas de minsta och mest grundläggande enheterna först, varefter dessa kan användas som komponenter i större och mera kraftfulla enheter. Vid bottom-up testas och godkänns således varje enhet innan den används i någon annan enhet. Tanken med detta är att man därigenom har större möjlighet att lokalisera och åtgärda felen. Om alla komponenter som ingår i enheten är felfria (önsketänkande) måste ju felet givetvis ligga i enheten själv. Vid bottom-up testas enheterna genom att man skriver särskilda testprogram för varje enhet.

Buttom-up testning är ofta en utmärkt metod att använda vid utveckling av relativt små program eller för att testa mindre delar av större program. Vid testning av stora program är emellertid den bästa strategin att använda sig av top-down testning. Orsaken till detta är att stora program utvecklas enligt top-down principen, varför de också lämpligast testas enligt samma princip. Top-down kan sägas vara motsatsen till bottom-up. I bottom-up utvecklas de minsta och mest detaljerade enheterna först, medan dessa enheter utvecklas sist i top-down. Istället utvecklas de största och mest abstrakta enheterna först. Det som är tilltalande med top-down är att man under utvecklingsarbetet startar på systemnivån och arbetar sig successivt ner till de minsta komponenterna. Därigenom läggs fokus på systemet och inte komponenterna. Detta är naturligtvis en klar fördel, eftersom det är systemet vi skall konstruera och det är systemets behov som bestämma vilka komponenter som är lämpliga att använda. Därigenom kan man testa den grundläggande konstruktionen av systemet och korrigera eventuella brister mycket tidigt i utvecklingsarbetet.

Hur kan man då testa ett program innan man skrivit de delar som ingår i programmet? Svaret på detta är att vi skriver mycket enkla och ofullständiga versioner av de delar som behövs och använder dessa förenklade versioner vid testningen. Ett program som använder sådana förenklade programenheter kallas för *stubbprogram* (*eng: stub programs*) och de förenklade programenheterna kallas för *stubbar* (*eng: stubs*). Ett stubbprogram är ett komplett, kompileringsbart och exekverbart program, men det utför inga egentliga beräkningar.

Metoden gör det möjligt att avgöra om man delat in programmet i lämpliga programenheter. Om så inte är fallet måste man backa tillbaks och finna en lämpligare uppdelningen av enheterna i programmet. Top-down tillåter att man testar den exakta specifikationen av vad varje klass och metod skall göra innan man fortsätter och i detalj bestämmer hur man skall göra implementationen av de enskilda klasserna och metoderna.

Normalt användes en kombination av top-down och button-up testning.

4. 4.1. Inkrementell utveckling

Det är vedertagen praxis att utföra enhetstester och systemtest. Men om alla enheter förs samman på en gång vid systemtestet kan det vara svårt att lokalisera eventuella fel som uppstår. Ett alternativt tillvägagångssätt till denna "big-bang"-test är att successivt utöka systemet med nya programdelar och utföra testningar allteftersom programdelarna integreras i systemet. Denna metod kallas för inkrementell programmering och stegen i modell är

1. Implementera en liten del av programsystemet.
2. Testa och avlusa det som är möjligt att testa.
3. Utöka programsystemet med en ny programkomponent.
4. Repetera från steg 2 till programsystemet är komplett.

Konsten är att identifiera vilken del av programsystemet man skall börja med och i vilken ordning de olika programdelarna skall infogas till systemet.