Harald Heimtun Soleng
og Eva Hadler Vihovde

PLETTFRI KODE

VINNER AV
Universitetsforlagets pris for beste lærebokidé



Hvordan forhindre feil i programvare?

Universitetsforlaget

Plettfri kode

Harald Heimtun Soleng og Eva Hadler Vihovde

Plettfri kode

Hvordan forhindre feil i programvare?

Universitetsforlaget

© Universitetsforlaget, Oslo 2023
ISBN: 9788215060217
1. utgave, 1. versjon, Universitetsforlaget 2022
E-utgaven baserer seg på 1. trykte utgave, Universitetsforlaget 2022

Elektronisk tilrettelegging: Universitetsforlaget
Omslagsdesign: Ellen Lorenzen
Illustrasjoner: Arnvid Moholt
Sats: Harald H. Soleng

Deler av denne boka baserer seg på ISTQB, Sertifisert Tester Pensum for grunnivå (Foundation Level) norsk versjon 28.04.2019 av Hans Schaefer, Berit Kristine Hatten, Ernst von Düring, Monika Stöcklein-Olsen, Tor Kjetil Moseng og Frans Dijkman som igjen er basert på Certified Tester Foundation Level 2018 V1.0. ISTQB har opphavsretten til pensum for grunnivå, og tillatelse er blitt gitt av ISTQB til å bruke dette materialet i boka.

Harald H. Soleng har mottatt støtte fra Det faglitterære fond.

Materialet i denne publikasjonen er omfattet av åndsverklovens bestemmelser. Uten særskilt avtale med Universitetsforlaget er enhver eksemplarfremstilling og tilgjengeliggjøring bare tillatt i den utstrekning det er hjemlet i lov eller tillatt gjennom avtale med Kopinor, interesseorgan for rettighetshavere til åndsverk.

www.universitetsforlaget.no
post@universitetsforlaget.no

Forord

Vårt mål med denne boka er å bidra til høyere kvalitet i programvare ved å øke bevisstheten om testingens betydning, samt tilføre utviklere grunnleggende kunnskap om testing. Vi gir råd og prinsipper for å skrive kode som er vedlikeholdbar og enkel å teste, og presenterer viktige prinsipper for det å lage utfyllende og dekkende tester. Boka er spesielt rettet mot datateknologistudenter, og kan med fordel brukes som lærebok i fag som programmering, programvareutvikling, testing av programvare, og ikke minst under arbeidet med bachelor- og masterprosjekter. Boka er imidlertid aktuell også utenfor utdanningsinstitusjonene. Alle involverte parter i et programvareprosjekt bør etter vår mening ha god forståelse for betydningen av testing. Både oppdragsgiveren og prosjektledelsen må gi ressurser og tid til tilstrekkelig kvalitetskontroll gjennom alle faser av utviklingen.

Selv om noe av innholdet i denne boka kan virke selvsagt, tilsier likevel erfaring at mange programmerere og selskaper burde ta kvalitetskontroll og testing mye mer alvorlig enn hva de gjør i dag. Vi håper at denne boka og eksemplene den inneholder, kan hjelpe programmerere og industrien til å forbedre utviklingsprosessen slik at færre feil slipper gjennom til det endelige produktet.

Til tross for at mange av testteknikkene vi gjennomgår i boka, kan bidra til å redusere så vel utviklingskostnader som utviklingstid, er det viktig å forstå at det å forhindre programvarefeil ikke er gratis. I begynnelsen vil det forsinke utviklingsprosessen, og det koster. Å ignorere risikoen og lette på testregimet koster imidlertid mye mer i det lange løp. Når feilene til slutt slår tilbake etter å ha blitt feiet under teppet, biter de hardere. Å fikse en bug i kvalitetssikringstadiet koster i størrelsesorden ti ganger mer enn å oppdage den under utviklingen. Å rette den etter at programmet er rullet ut, er typisk rundt hundre ganger dyrere.

Vi håper du finner boka interessant og nyttig. Skulle du komme over feil eller mangler, er vi takknemlige for kommentarer.

Vi vil først og fremst takke Universitetsforlaget og juryen i kåringen av «Beste lærebokidé 2021» for å ha gitt oss muligheten til å skrive og utgi boka. Dernest takker vi Roy M. Istad for konstruktive kommentarer og innspill. Harald H. Soleng takker Norsk faglitterær forfatter- og oversetterforening for støtte og Norsk Regnesentral for en sabbatsperiode i et tidlig stadium av arbeidet med boka. Vi takker Storm Jaran Bruvoll Westlie, Maja Wawrzyniak Soleng og Daniel Bakkelund for å

ha tatt seg tid til å lese manuskriptet og for å ha kommet med nyttige kommentarer og forslag.

Oslo, oktober 2022
Harald Heimtun Soleng og *Eva Hadler Vihovde*

Innhold

Forord **i**

1 Dødelige tegn og kommafeil **1**
1.1 Kjente katastrofer 1
1.2 Feil spart er penger tjent 4
1.3 Leserveiledning 4

2 Kilder til feil **7**
2.1 Dårlig problemforståelse 7
2.2 Kryptisk kode 8
2.3 Dårlig fundamentering 9
2.4 Å stole på debuggere 9
2.5 Behandle symptomene i stedet for sykdommen 10
2.6 Refaktorering 11
2.7 Oppsummering 11

3 Syv prinsipper for testing **13**
3.1 Prinsipp 1. Testing kan ikke bevise at koden er feilfri 13
3.2 Prinsipp 2. Fullstendig testing er umulig 14
3.3 Prinsipp 3. Tidlig testing sparer tid og penger 15
3.4 Prinsipp 4. Feil samler seg i klynger 15
3.5 Prinsipp 5. Pass på sprøytemiddel-paradokset 16
3.6 Prinsipp 6. Testing er avhengig av konteksten 16
3.7 Prinsipp 7. Feilslutning vedrørende fravær av feil 17
3.8 Oppsummering 17

4 Arbeidsprosessen **19**
4.1 Utviklingsprosessen for programvare 19
4.2 Versjonskontroll 21
4.3 Oppsummering 22

5 Testing i produktets livssyklus **25**
5.1 Sentrale begreper og testaktiviteter 25
5.2 Aktivitetene i henhold til V-modellen 30
5.3 De fire testnivåene 32
5.4 De fire testtypene 38
5.5 Oppsummering 40

6 Å forhindre feil **41**
6.1 Fagfellevurdering av koden 41
6.2 Statisk kildekodeanalyse 42
6.3 Tilstandskontroll 44
6.4 Versjonskontroll 44
6.5 Verktøy for å finne minnefeil 45
6.6 Enhetstester 45
6.7 Integrasjonstesting 49
6.8 Systemtester 50
6.9 Akseptansetester 50
6.10 Kontinuerlig integrasjon og automatisk testing 51
6.11 Avsluttende bemerkninger om bug-feller 51

7 Solid og testbar kode **53**
7.1 Enkeltansvarsprinsippet: En klasse, ett ansvar 53
7.2 Åpen/lukket-prinsippet 56
7.3 Liskovs substitusjonsprinsipp 56
7.4 Prinsippet om seperasjon av grensesnitt 57
7.5 Prinsippet om inversjon av avhengigheter 57
7.6 Avhengighetsinjeksjon 60
7.7 Veien til solid kode 60
7.8 Oppsummering 62

8 Testprosessen **63**
8.1 Plan og kontroll 64
8.2 Analyse og design 65
8.3 Testimplementering og utførelse 69
8.4 Evaluering av utgangskriterier og rapportering 71
8.5 Avsluttende testaktiviteter 72
8.6 Avsluttende ord 73

9 Statiske testteknikker **75**
9.1 Granskninger 76
9.2 Statisk analyse 77
9.3 Oppsummering 77

10 Spesifikasjonsbaserte teknikker **79**
10.1 Svart boks-testing 80
10.2 Ekvivalensklasseinndeling 80
10.3 Grenseverdianalyse 83
10.4 Grenseverdianalyse og ekvivalensklasser 84
10.5 Beslutningstabelltesting 85
10.6 Tilstandsbasert testing 92
10.7 Testing basert på brukstilfeller 96
10.8 Oppsummering 100

11 Strukturbaserte testteknikker **101**
11.1 Testdekning 101
11.2 Oppsummering 106

12 Erfaringsbaserte testteknikker **109**
12.1 Feilgjetting 110
12.2 Utforskende testing 110
12.3 Oppsummering 113

13 Kildekodedokumentasjon **115**
13.1 Ekstraherbar dokumentasjon 115
13.2 Litterær programmering 116
13.3 Hvorfor dokumentere? 117

14 Praktisk testing **119**
14.1 Testrammeverk 119
14.2 Kontraktsrammeverk 120
14.3 Enhetstester i praksis 123
14.4 Eksempel på bruk av statisk assert 126

15 Testingens psykologi **129**
15.1 Utvikling versus testing — to ulike tankesett 129
15.2 Uavhengig testing 130
15.3 Kunsten å formidle og motta negativ kritikk 132
15.4 Testorganisering 134
15.5 Oppsummering 135

16 Automatisert testing **137**
16.1 Hva er et testverktøy? 137
16.2 Potensielle fordeler ved bruk av testverktøy 138
16.3 Risikofaktorer knyttet til bruk av testverktøy 139
16.4 Testautomatisering 140
16.5 Valg av testverktøy 144

16.6 Oppsummering 145

17 Testguide for studentprosjekter **147**
17.1 Generelle råd 148
17.2 Startfasen 149
17.3 Programmets fundament – kravspesifikasjonen 151
17.4 Analyser testforholdene og design testtilfellene 152
17.5 Dynamisk testing gjennom fire testnivåer 153
17.6 Regresjonstesting 155
17.7 En siste test 155
17.8 Testing kan ikke bevise at koden er feilfri 156
17.9 Avsluttende ord 156

A En fikspunktimplementering **157**
A.1 Flyttall og fikspunkttall 157
A.2 Implementering av fikspunktklasse 159
A.3 Enhetstester 174
A.4 Testdekning for FixedPoint-klassen 186
A.5 Matematiske funksjoner 197
A.6 Testdekning for matematiske funksjoner 217
A.7 Kildekodedokumentasjon 226

Norsk–engelsk ordliste **227**

Engelsk–norsk ordliste **231**

Bibliografi **235**

Stikkordregister **239**

Kapittel 1

Dødelige tegn og kommafeil

> Å feile er menneskelig, å tilgi er guddommelig.
>
> *Alexander Pope*

Siden datamaskinen vil utføre ordre slavisk, og brukerne neppe er guddommelige, så kan du som programmerer ikke forvente å slippe unna med feil. Desto verre er at noen brukere synes å glemme at datasystemer er menneskeskapte og derfor kan inneholde feil. Derfor kan noen trafikanter finne på å sette bilen i autopilotmodus og lese avisen mens sensorer og dataprogrammer overtar styringen.

Vanligere er det at datafeil fører til irritasjon. Faktisk er et av de største irritasjonsmomentene på norske arbeidsplasser datasystemer som ikke virker. Det kan være programmer som er unødvendig trege eller som regelrett krasjer og fører til tapt arbeid.

Historien har imidlertid flere eksempler på katastrofale konsekvenser av datafeil. Noen av de mest kjente er nevnt nedenfor.

1.1 Kjente katastrofer

Eksemplene som følger, viser at tilsynelatende bagatellmessige feil i dataprogrammer kan få alvorlige følger.

Mariner 1-krasj

Mariner-programmet (1959–1963) til NASA skjøt opp en serie interplanetariske romsonder for å undersøke Mars, Venus og Merkur. I 1962 feilet oppskytingen av Mariner 1 på grunn av en programmeringsfeil[1]. Forfatteren av spesifikasjonsdokumentet for styringssystemet overså symbolet for gjennomsnitt, en stripe, i en håndskrevet formel, slik at $\bar{\dot{r}}_n$ ble erstatet med $\dot{r}_n$. I stedet for å korrigere ut fra variasjoner i en gjennomsnittsverdi, forsøkte styringssystemet å korrigere for hver lille tilfeldige variasjon. Alle de meningsløse korreksjonene endte med at raketten havarerte [2].

Nesten-ulykke i bemannet romrakett

Mercury-programmet var det første bemannede amerikanske romprogrammet. Programmeringsspråket som ble brukt den gang, het Fortran, og i koden som beregnet rakettens bane, fantes følgende linje:

```
    DO 17 I = 1.100
       :  :
17  CONTINUE
```

i stedet for

```
    DO 17 I = 1,100
       :  :
17  CONTINUE
```

Hadde ikke punktumet blitt rettet til et komma, ville en katastrofe skjedd. For det blotte øye var forskjellen mellom punktum og komma bare en piksel på dataskjermen, men i språket Fortran er det forskjellen mellom tilordning av en verdi til en variabel kalt `DO17I` (siden Fortran kan ignorere mellomrom) og det å gjenta koden frem til merkelappen `17` hundre ganger [3]. Dypt inne i et dataprogram kan derfor en liten tegnsettingsfeil få fatale konsekvenser. Heldigvis ble denne feilen funnet i tide.

Ulik bruk av desimaltegn i ulike språk er for øvrig en hyppig kilde til feil; engelskspråklige bruker desimalpunkt og komma som tusenskilletegn, mens man på de fleste språk gjør det omvendt (jf. tabell 1.1). På norsk brukes desimalkomma, men innen norsk akademia er likevel desimalpunkt på fremmarsj. Siden de fleste programmeringsspråk er basert på engelsk, er det desimalpunkt som gjelder i kildekode.

Tabell 1.1: Desimal- og tusenskilletegn

Engelsk	one million	1,000,000.00
Norsk	en million	1.000.000,00

Tap av Mars-sonden Phobos 1

I september 1988 mistet russerne sin Phobos 1 Mars-sonde på grunn av en feil i kommandoer som ble lastet opp i begynnelsen av måneden. Vanligvis skulle all kommunikasjon med romsonden gå gjennom en automatisk syntakskontroll som sjekket at kommandoene fulgte de spesielle språkreglene som gjaldt for datamaskinen i sonden, men korrekturlesersystemet var i ustand, og operatøren måtte laste opp kommandoene uten denne sjekken. Dessverre resulterte dette i at Phobos 1 mottok en *avslutt-kommando* som resulterte i at datamaskinen ble skrudd av og

det ble umulig å få kontakt med romsonden igjen [4].

Ariane 5-krasj til en halv milliard dollar

Som kjent er datamaskinens minne kun i stand til å lagre informasjon binært, som tall i totallsystemet. Heltall kan lagres på ulike måter i dataminnet. Er man sikker på at tallet f.eks. er mellom 0 og 255, klarer man seg med åtte siffer i totallsystemet. Da kan man bruke det som kalles et 8-biters usignert tall.

Den 4. juni 1996 krasjet en Ariane 5-rakett på grunn av datakonvertering fra et 64-biters desimaltall til et 16-biters heltall med fortegn. Da tallet ble større enn 32 767, som er det største 16-biters signerte heltallet, fikk man det som kalles heltallsoverflyt. Datatypen var ikke i stand til å lagre det riktige tallet og dermed mistet man styrings- og høydeinformasjon [5]. Resultatet var at raketten skar ut av banen og eksploderte (figur 1.1).

Raketten og nyttelasten var verdsatt til 500 millioner dollar. I dette tilfellet var det bare materielle tap, men andre ganger har programmeringsfeil gitt dødelige resultater.



Figur 1.1: Ariane 5-raketten eksploderte som følge av heltallsoverflyt. Foto: ESA.

Menneskeliv tapt da luftvernmissil feilet

Programmeringsfeil har også ført til tap av menneskeliv. Et eksempel var 25. februar 1991 da et Patriot-missil ikke klarte å avskjære et irakisk Scud-missil, som deretter traff en amerikansk base og drepte 100 soldater og skadet 100 andre. Feilen skyldtes aritmetiske feil i tidsberegninger [6]. Multiplisering av den interne klokketiden med 1/10 for å få tiden i sekunder ble gjort i et 24-biters register med tall med et fast antall desimaler. Dette gav en avrundingsfeil på $0{,}000\ 000\ 095$ s hvert tiendedels sekund. Etter 100 timer operasjonstid hadde den interne klokken tikket $100\ \text{t} \cdot 60\ \text{min/t} \cdot 60\ \text{s/min} \cdot 10\ \text{(tiendedels sekund)/s} = 3\ 600\ 000$ tiendedels sekunder. Da hadde feilen akkumulert til

$$3\ 600\ 000\ \text{tiendedels s} \cdot 0{,}000\ 000\ 095\ \text{s/(tiendedels s)} = 0{,}34\ \text{sekunder.}$$

Dette var nok tid til at Scud-missilet hadde beveget seg mer enn en halv km slik at Patriot-missilet mistet målet. I virkeligheten skulle imidlertid ikke målets posisjon avhenge av absolutt klokketid, men av tiden mellom to radarpulser, så avrundingsfeilen skulle uansett vært neglisjerbar. Problemet var at en mer nøyaktig funksjon for å oversette klokketiden til flyttall hadde blitt introdusert i det 20 år gamle systemet slik at det kunne hanskes med hurtigere, moderne mål, men man hadde oversett å introdusere den nye funksjonen overalt. En subtraksjon av to tider beregnet på ny og gammel måte gav den store feilen på $0{,}34$ sekunder.

Oljeplattform sank og 2,3 milliarder kroner gikk tapt

I august 1991 sank Sleipner A-plattformen i Gandsfjorden etter at et av plattformbeina sprang lekk. Ulykken skyldtes alvorlige feil i en numerisk styrkeberegning [7]. Beregningene undervurderte skjærkreftene med 47 % i veggene til plattformbeina. Dette resulterte i en sprekk, slik at plattformen sank [8]. Heldigvis slapp alle de 22 om bord fra det med livet i behold, men forsikringsutbetalingene ble på 2,3 milliarder kroner [9].

1.2 Feil spart er penger tjent

Selv om ikke alle datamaskinfeil har katastrofale konsekvenser, har de fleste brukerne erfart irriterende datafeil som har kostet tid og penger. For utviklere er det spesielt irriterende når kode man har skrevet, feiler på grunn av feil i eldre kode. Å feilsøke og rette kode man ikke er kjent med, er både tidkrevende og risikabelt, og kan til og med føre til nye feil. Derfor er det i alles interesse at utviklere fokuserer på å unngå feil, og at de bruker metoder som reduserer feilrisikoen til et minimum.

Før vi kan sikre oss mot at det lages slike feil, må vi for det første forstå hvordan feil oppstår, og dernest hvordan vi kan finne og rette disse før de leder til katastrofer. De neste kapitlene vil bidra til dette.

1.3 Leserveiledning

Denne boka vender seg mot alle som er involvert i programvareutikling. Noen kapitler er mest relevante for programmererne, mens andre er rettet mer mot profesjonelle testere. Prosjektledere og oppdragsgivere vil også ha nytte av boka.

Kapitlene 1 til 6 bør leses av alle. Kapittel 7, som handler om å skrive solid og testbar kode, er hovedsakelig rettet mot utviklere som selv skriver kode. Eksemplene i dette kapitlet er basert på programmeringsspråket `C++`, men de fleste av prinsippene er overførbare til andre programmeringsspråk. Kapitlene 8 til 12 går gjennom testprosessen og de ulike testteknikkene og krever ingen forkunnskaper i programmering. Kapittel 13 og kapittel 14 omhandler kildekodedokumentasjon og praktisk implementering av tester og er mest relevant for de som selv skal skrive kildekode. Kapittel 15 om testingens psykologi bør leses av alle, mens

kapittel 16 om automatisering er nok mest relevant for testere og prosjektledere. Testguiden i kapittel 17 er myntet på studenter og andre som har små programvareprosjekter. I appendiks A utvikler vi en `C++20` klasse for å håndtere fikspunkttall. Dette tillegget passer for lesere med god kunnskap til `C++` som vil se en detaljert implementasjon og enhetstestsuite som benytter seg av teknikkene som er omtalt i boka.

Kapittel 2

Kilder til feil

> Hvem som helst kan ta feil, men bare tåpen holder fast ved sin feiltagelse.
>
> *Marcus Tullius Cicero*

Tenk på programvare som en problemløser. For en spesiell klasse av problemer skal programvaren være i stand til å finne en løsning. Feil i programvare fører enten til feil løsning eller ingen løsning i det hele tatt. Selv om programmet produserer riktig resultat, kan det likevel bryte med kravene som oppdragsgiver har satt, dersom resultatet blir funnet først etter lang tid. Ofte er treg programvare bare et irritasjonsmoment — en timeglassmarkør som varer for lenge — men i sanntidsapplikasjoner kan treghet være livstruende. Hva om du kommer ut for en bilulykke og programvaren i bilen din er så treg at kollisjonsputene ikke utløses i tide?

Før vi kan håpe på å redusere antallet og alvorlighetsgraden til programvarefeil i koden som vi skriver, trenger vi å forstå hvorfor programvarefeil oppstår. Når de vanlige kildene til problemene er kjent, er det lettere å ta grep for å ta ondet ved roten.

Som Cicero påpekte, kan hvem som helst feile, og derfor kan programvarefeil oppstå hele veien i utviklingsprosessen, helt fra man først innser at det finnes et problem som trenger en programvareløsning, til de siste stegene i programmeringen. Det kan være feil i problemforståelsen, feil i spesifikasjonen og sist, men ikke minst, i implementeringen. Feilene kan oppstå på mange måter. I dette kapitlet går vi gjennom hyppig forekommende kilder til feil.

2.1 Dårlig problemforståelse

En forutsetning for korrekt problemløsning er å ha tilstrekkelig forståelse av problemet. Dette kan virke selvsagt, men uheldigvis er det ikke uvanlig i industrien at man kaster seg over problemet og begynner problemløsning før man vet hva

problemet er, og enda mindre hva kriteriet for suksess vil være. Det hender at oppdragsgiver mangler forståelse for problemet som skal løses, og dermed blir spesifikasjonen uklar. Altfor ofte starter store programvareprosjekter uten at de involverte har forståelse av de grunnleggende sidene ved problemet. Selv om den fundamentale forståelsen vanligvis er der, er ofte spesifikasjonen vag og mangelfull når det gjelder testbare krav. Når prosjektteamet omsider når en bedre forståelse, bør prosjektet ta kostnaden ved å rette dårlige designbeslutninger fattet på sviktende grunnlag tidlig i den forvirrede oppstartsfasen. Hvis ikke vil disse designfeilene forplante seg i programvare som blir vanskelig å vedlikeholde og videreutvikle. Dette vil uvilkårlig lede til programvarefeil i sluttproduktet.

En annen kilde til feil er programmering mot et *bevegelig mål*. Problemet oppstår når spesifikasjonen endres sent i utviklingsløpet, og utviklerne må hacke modifikasjoner i siste øyeblikk. Dette kan skje om ikke sluttbrukerens behov har vært tilstrekkelig ivaretatt i den opprinnelige spesifikasjonen, og man finner ut at designet må endres mot slutten av utviklingen. Selvfølgelig må man være i stand til å gjøre små justeringer helt opp til endelig kodeleveranse, men noen ganger kan tilsynelatende små forandringer tvinge frem større designendringer. I slike tilfeller er det bedre å ta kostnaden med å redesigne, heller enn å arbeide seg rundt designbegrensninger med *ad hoc* snarveier bare for å få siste endring til å virke. Selv om det kan implementeres raskt, er slike løsninger ofte ineffektive, lite elegante, og til og med vanskelige for andre å forstå, og leder typisk til bug-infisert kode som blir vanskelig å vedlikeholde. Jo senere spesifikasjonsendringer kommer, dess større er risikoen for å produsere hekkeplasser for programvarefeil.

Når et team av smarte mennesker står fast og ikke kommer videre i et prosjekt, er det vanligvis fordi problemet er dårlig formulert, og ikke fordi teamet er svake på problemløsning. I slike situasjoner har man all grunn til å begynne den kreative prosessen med å definere problemet uten å falle tilbake på de opprinnelige uklare problembeskrivelsene og vage suksesskriteriene. Først når problemet er veldefinert og testbare kriterier for suksess foreligger, vil teamet kunne se målet med prosjektet. Fordi å forstå er å oppnå, gir dette en utrolig samlende energi og et bestemt driv mot det å fullføre prosjektet.

2.2 Kryptisk kode

Enhver programmerer bør huske en ting: Du skriver ikke kode for kompilatoren. Ganske riktig kommer du ingen vei før kompilatoren aksepterer det du har skrevet, men med mindre du skriver små trivielle programmer, vil de virkelige brukerne av din kode være fremtidige kollegaer som får i oppgave å utvide og vedlikeholde systemet. Hvis de ikke kan forstå koden, så er det godt mulig du heller ikke gjorde det. Selv om en programmerer må være svært presis når det gjelder å forholde seg til datamaskinspråkets grammatikk, så er innholdet fritt. Misvisende variabel- og funksjonsnavn er ingen hindring for en kompilator, men kryptisk kode er ofte