# Kvalitetssikring

## Test (Karsten)

Formålet med tests er at identificere fejl i koden (defects)—og de findes selvom vi kan føle os nok så overbeviste om at den er fejlfri.

*”A defect is the algorithmic cause of a failure: some code logic that is incorrectly implemented”* [Christensen, s. 16]

Slipper fejl i koden forbi os, risikerer vi at vores program fejler under brug (failure).

*”A failure is a situation in which the behavior of the executing software deviates from what is expected”* [Christensen, s. 16]

Det kan være en farlig situation for vores image som udviklere hvis programmet fejler ude hos kunden, og det skader indtrykket af vores produkt. Dertil kommer at fejlretning bliver betydeligt vanskeligere at foretage når først produktet er hos kunden efter release. Det kan fx være en udfordring at skulle reproducere en fejl alene baseret på brugerobservationer hos kunden. Vi ønsker derfor at opdage fejl så hurtigt som muligt i udviklingsprocessen. Jo hurtigere de opdages, des færre fejlretninger skal vi foretage, og vi undgår helst at ende med en stor kodebase som viser sig afhængig af fejlagtig funktionalitet på et dybere niveau.

### Hvad er en test?

Testen er et værdifuldt værktøj og kilde til høj moral idet den giver tillidsvækkende feedback til os som udviklere. Vi kan skrive ny kode eller refaktorisere eksisterende kode og straks—hvis testen er automatiseret—få bekræftet at vores kode fungerer efter hensigten.

*”Testing is the process of executing software in order to find failures”* [Christensen, s. 15]

Vores tests beviser ikke at vores program er fejlfrit (en fuldkommen og udtømmende test er urealistisk), men gode tests reducerer risikoen for failures i takt med at defects identificeres og rettes. Formår vi at lave omfattende tests på alle centrale dele af vores program, kan vi hvile i forsikringen om at vi leverer et produkt af høj kvalitet—i hvert fald hvad løsningens korrekthed angår.

### Test cases

Når vi tester, arbejder vi med konkrete test cases. En test case udgør et specifikt, afgrænset scenarie hvor en testenhed kontrolleres i forhold til et forventet output som respons på et konkret sæt af input.

*”A test case is a definition of input values and expected output values for a unit under test”* [Christensen, s. 16]

En testenhed (unit) er i bred forstand en del af et system som vi i testsammenhæng betragter som en hel, atomisk enhed. Hvad en unit i en given test case dækker over, afhænger af testens omfang og kan spænde fra et lille metodefragment, over subsystemer og helt til det komplette system.

*”A unit under test is some part of a system that we consider to be a whole”* [Christensen, s. 17]

Der vil typisk være flere test cases rettet mod en given unit. Samlingen af de relaterede test cases udgør en test suite som søger at definere en dækkende test.

*”A test suite is a set of test cases”* [Christensen, s. 16]

### Hvordan vi tester

Det er muligt—men sjældent udført i praksis—at teste koden manuelt. I den manuelle test er det en person som styrer gennemgangen af en test case. Input angives manuelt til programmet hvorefter output verificeres. Det er en tidskrævende form for test, men kan af og til være nødvendig, fx ved brugertest.

*”Manual testing is a process in which suites of test cases are executed and verified manually by humans”* [Christensen, s. 17]

Mere anvendelig er den automatiserede test hvor der skrives egentlig testkode hvis ansvar det er at udføre test cases og verificere at faktisk output stemmer overens med det forventede.

*”The test code is the source code that defines test cases for the production code”* [Christensen, s. 19]

Automatiseret test tillader dermed en nem og hurtig udførelse af test suites, og derfor ønsker vi naturligvis at automatisere vores tests så vidt muligt.

*”Automated testing is a process in which test suites are executed and verified automatically by computer programs”* [Christensen, s. 18]

Vi får dermed to sæt af kode: produktionskode og testkode. De to sæt holdes skarpt adskilt da testkode ikke er en del af produktet. Den separate testkode er dog afhængig af produktionskoden i det omfang at de enkelte test cases er koblet til produktionskoden. Vigtigt er det at produktionskoden intet kender til testkoden.

*”The production code is the code that defines the behavior implementing the software’s requirements”* [Christensen, s. 18]

Automatiserede tests har desuden den sidegevinst at motivere til udførelse af regressionstest ofte, så vi hele tiden sikrer at nye tilføjelser til og ændringer i produktionskoden ikke får uventede, negative konsekvenser.

*”Regression testing is the repeated execution of test suites to ensure they still pass and the system does not fail after a modification”* [Christensen, s. 17]

### Test framework

Når vi implementerer vores test cases i testkoden, er det naturligt at benytte et test framework. Frameworket består af en mængde af fælles funktionalitet til udførelse af test. Det øger produktiviteten at vi ikke selv skal udarbejde de fundamentale dele af testkoden, og ikke mindst bidrager brugen af et framework til at holde risikoen for defects i testkoden på et minimum. Andre nyttige funktioner er typisk muligheden for opstilling af statistikker over udførte tests.

Udbuddet af test frameworks er stort. Udbredt til Java er frameworket JUnit som vi benytter i dette projekt. Centralt i test frameworket er assert-mekanismen som sammenligner forventet med faktisk værdi og melder fejl hvis vores antagelse ikke holder. Vi kan supplere frameworkets standardbesked med vores egen for en mere detaljeret tilbagemelding.

## Testparadigmer (Karsten)

For at sikre kvaliteten af vores program mest muligt kan vi teste efter særlige testparadigmer. Testparadigmerne beskriver hver især en bestemt tilgang til løsning af testopgaven på en måde så testen vurderes tilstrækkelig. Vi kan ikke teste udtømmende da det vil kræve et urealistisk stort antal test cases. Der er derfor behov for kriterier for tilstrækkelighed der lader os begrænse antallet af test cases og samtidig sikre at dækningen af dem er fyldestgørende.

### Blackbox

Blackbox-test, også kendt som functional testing, beskriver en tilgang til testenheden som et lukket system. Her kender vi i princippet intet til hvordan systemet fungerer; kun hvad dets funktion er. Vi tester systemet på grundlag af forventninger og specifikationer, og kan således opstille en liste af inputs hvortil der skal svare et bestemt output. Her er kriteriet for tilstrækkelighed at hver enkelt delfunktion er testet mindst én gang. Kombinationer af input som behandles forskelligt tester hver deres delfunktion.

Blackbox-test er et alternativ til whitebox-test, eller structural testing, som omvendt tilgår testenheden som et åbent system. Her kan anvendelsen af kodemetrikker til automatiseret analyse på en række forskellige parametre bl.a. danne grundlag for udarbejdelsen af dækkende tests samt forbedring af produktionskoden gennem refaktorisering.

Der findes forskellige testteknikker til opdeling af testforløbet i bestemte aktiviteter. I blackbox-test findes en vifte a teknikker som hver især dækker systemet på forskellig vis. Hver teknik afspejler en specifik måde at drive testen på.

### Datadrevet test

I datadrevet test er målet at identificere de sammenhænge mellem data som tilsammen vil teste alle dele af systemet. Med udgangspunkt i projektets case har vi i kraft af forretningsreglen vedrørende renteberegning en beskrivelse af sådan sammenhænge. Der gælder følgende:

* Kreditværdighed skal være A, B eller C
* Kreditværdighed A tillægger +1 procentpoint
* Kreditværdighed B tillægger +2 procentpoint
* Kreditværdighed C tillægger +3 procentpoint
* Udbetalingsprocenten skal være på mindst 20 %
* En udbetalingsprocent under 50 % tillægger +1 procentpoint
* En løbetid på mere end 36 mdr. tillægger +1 procentpoint

Hertil føjer vi følgende begrænsninger for frasortering af værdier som ikke giver mening i forbindelse med beregningen:

* Løbetiden skal være minimum 1 md.
* Udbetalingsprocenten skal være under 100 %
* Dagsrenten skal være og *(en mere restriktiv begrænsning vil helt sikkert være fornuftig)*

Med denne behandlingsregel defineret benytter vi nu en fast fremgangsmåde for datadrevet test:

1. Identificér det gyldige og det ugyldige værdiområde
2. Opdel de to områder i ækvivalensklasser
3. Beskriv grænseværdier

[Vinje 5.4.2]

#### Værdiområder

Tabel NNN viser input til renteberegningen og deres gyldige værdiområder.

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Type | Gyldigt værdiområde | Særlige, ugyldige værdier |
| Dagsrenten | double |  | Double.NaN |
| Kreditværdighed | Rating enum |  | null |
| Udbetalingsprocent | double |  | Double.NaN |
| Løbetid | int |  |  |
|  |  |  |  |

*Tabel NNN: Input og værdiområder for renteberegning*

#### Ækvivalensklasser

Tabel NNN viser de identificerede ækvivalensklasser. Vi vælger at lade ækvivalensklasser med samme ugyldige input sammenfatte til én. Dermed får vi reduceret i alt klasser til blot .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Klasse# | Dagsrente | Kreditværdighed | Udbetalingspct. | Løbetid / mdr. |
| 1 | 1.00% | A | 50.00% | 37 |
| 2 |  |  |  | 36 |
| 3 |  |  |  | 0 |
| 4 |  |  | 20.00% | 37 |
| 5 |  |  |  | 36 |
| 6 |  |  | 19.99% | 37 |
| 7 |  |  | 100.00% | 37 |
| 8 |  |  | NaN | 37 |
| 9 |  | B | 50.00% | 37 |
| 10 |  |  |  | 36 |
| 11 |  |  | 20.00% | 37 |
| 12 |  |  |  | 36 |
| 13 |  | C | 50.00% | 37 |
| 14 |  |  |  | 36 |
| 15 |  |  | 20.00% | 37 |
| 16 |  |  |  | 36 |
| 17 |  | D | 50.00% | 37 |
| 18 |  | null | 50.00% | 37 |
| 19 | NaN | A | 50.00% | 37 |
| 20 | POS\_INFTY | A | 50.00% | 37 |
| 21 | NEG\_INFTY | A | 50.00% | 37 |
|  |  |  |  |  |

*Tabel NNN: Identificerede ækvivalensklasser*

#### Grænseværdianalyse

*”Behovet for test af grænseværdier bygger på erfaringer. Der begås oftere fejl i grænseområderne end andre steder”* [Vinje, s. 224]

Visse grænser afspejles allerede i ækvivalensklasserne. Øvrige grænseværdier vi må supplere vores test suite med, er:

* En udbetalingsprocent på 49.99 %
* En udbetalingsprocent på 99.99 %
* En løbetid på 1 md.

#### Test cases

Vi lader os inspirere direkte af ækvivalensklasserne når vi skal definere vores test cases. Således ser TC-01 ud som i tabel NNN. De øvrige kan findes i bilag NNN.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TC# | INPUT |  | OUTPUT |  |
| 1 | Dagsrente | 1.00% | Rente | 3.00% |
|  | Kreditværdighed | A |  |  |
|  | Udbetalingspct. | 50.00% |  |  |
|  | Løbetid | 37 |  |  |
|  |  |  |  |  |

*Tabel NNN: TC-01*

Knap halvdelen af de i alt 24 test cases drejer sig om ugyldigt input, helt som ventet.

*”Det er en god tommelfingerregel, at halvdelen af alle testcases skal beskæftige sig med det forbudte, det usandsynlige og det umulige”* [Vinje, s. 222]

## TDD – Test Driven Development (Karsten)

Til implementering af vores test cases og udvikling af produktionskoden til renteberegning anvendte vi Test Driven Development (TDD). TDD er en agil praksis relateret til Test-First Programming-koncepter fra Extreme Programming (XP). Her handler det om at skrive unit testen først for derefter at implementere koden der skal testes. Denne fremgangsmåde øger testbarheden af kildekoden helt naturligt idet den implementeres med testen for øje. Effekten af dette er bl.a. øget tillid til implementationen i kraft af grundige tests, og det letter bekymringer om refaktorisering. Desuden sørger metoden for at fokus fastholdes under implementering så kodens funktionalitet begrænses til at opfylde de definerede test cases—og ikke mere end det. Feature eller scope creep er et kendt problem i branchen, og her kan TDD være en løsning.

Særlige værdier i TDD (og XP i øvrigt) er 1) software produceres ved at skrive kode, 2) tag små skridt, og 3) hold fokus. Ved anvendelsen af TDD opnår vi en række fordele:

* Ren kode der virker, opbygget ad små skridt og refaktoriseret undervejs.
* En struktureret proces for hvordan vi kan starte implementering for et problem. Vi kan anvende konkrete Green/Red Bar-mønstre der sætter os i gang frem for at arbejde ud fra en mavefornemmelse.
* Hurtig feedback er et boost til tillid og moral. Hurtige, små iterationer nedbryder problemet i små, overkommelige skridt som er hurtigt løst. Hver green bar er en succes der bringer os et skridt tættere på at have løst opgaven, og får vi red bar, ved vi at fejlen næsten med garanti må ligge i den nye kode tilføjet i den aktuelle iteration.
* Stærkt fokus på pålidelig software. Med vores tests kan vi have tillid til programmet selv efter omfattende refaktorisering. Det gør op med ”if it ain’t broke, don’t fix it”-mentaliteten hvor softwareudvikling styres af frygten for at introducere defects.
* Uafhængigt af øvrig dokumentation udgør vores test cases i sig selv detaljeret, up-to-date dokumentation af hvordan vores klasser anvendes (især med Evident Data-mønstret).

Figur NNN viser det typiske workflow i TDD.

|  |  |
| --- | --- |
| [Christensen, s. 14] beskriver rytmen i TDD:   1. Quickly add a test 2. Run all tests and see the new one fail 3. Make a little change 4. Run all tests and see them all succeed 5. Refactor to remove duplication | Skåret ud i pap er workflowet:   1. Tilføj test case til testliste 2. Kør test suite   *Hvis red bar*   * 1. Ændr kildekode   2. Gå til 2.   *Hvis green bar*   1. Tjek for koderedundans   *Hvis koderedundans*   * + 1. Refaktoriser kildekode     2. Gå til 2.   *Hvis ingen redundans*   1. Implementer ny test case 2. Gå til 2. |
|  |  |

*Figur 1: TDD workflow*

### Fake It (’Til You Make It)

Et Green Bar-mønster der anvendes til hurtig progression når test suites fejler, er Fake It. Her implementerer vi blot den simplest mulige løsning for at bestå en test case, fx ved returnering af en konstant med den forventede værdi eller en null-reference.

Forløbet i figur NNN efter red bar bliver da:

*Hvis red bar*

1. Ændr kildekode
2. Tilføj ny test case til testliste
3. Gå til 2.

Her anvender vi Triangulation og tilføjer en ny test case til testlisten som skal sikre at den naive løsning senere fejler. Fake It hjælper os således til at tage små skridt og holde fokus på at komme hurtigst muligt videre.

### Triangulation

Et Green Bar-mønster der ligger i tæt relation til Fake It er Triangulation. Med Triangulation kan vi opbygge abstraktion gradvist vha. adskillige datapunkter. Således kan skiftevis brug af Fake It og Triangulation bidrage til gradvis konstruktion af en kompleks algoritme i takt med at anvendelsen af Fake It resulterer i nye test cases med nye datapunkter.

### One Step Test

One Step Test er et eksempel på et Red Bar-mønster som anvendes når test suiten lykkes. Udgangspunktet her er at gå til implementering af den test case som umiddelbart virker nemmest. Der er fokus på hurtige iterationer, og implementering af test og løsning bør kun tage ganske få minutter. Begynder en implementering at vare længere, kan Fake It træde til som en måde at komme videre på. Ved test og udvikling af løsningen til renteberegning var One Step Test udgangspunktet for implementering af de test cases der blot forventer en Exception på ugyldigt input.

### Andre relevante testmønstre

* Obvious Implementation
  + Test er en afvejning af cost vs. benefit. Der opstår typisk situationer hvor de enkelte skridt i Fake It og Triangulation synes så små at det bedre kan betale sig at implementere en åbenlys løsning ad én omgang.
* Evident Tests
  + Vi forsøger så vidt muligt at undgå defects i vores tests ved at holde testkoden åbenbar og så simpel som muligt. Ideelt set består testkoden udelukkende af assignments og assertions; løkker og andre komplekse kontrolstrukturer undgås. [Christensen, s. 23]
* Evident Data
  + Vi viser hensigten med datapunkterne og tydeliggør hvad der er forventet og faktisk output. Det dokumenterer testen i sig selv så vel som enheden der testes.