Testdriven utveckling

Kandidatuppsats
Namn: Sonja Joelsson
Insitution: Insitutionen för informationsteknologi
År: 2013

Innehållsförteckning

1. Inledning

Programvaruutvecklare söker ständigt effektivare och bättre sätt att producera mjukvara. En rätt ny metod är testdriven utveckling. Metoden är en programutvecklingsmetod, inte en testmetod. Kortfattat går testdriven utveckling på att skriva test innan man skriver produktionskod. Det är en agil metod men på vilket sätt är oklart, som önskas svar på. En frågeställning är om och hur det påverkar programutvecklingen om man skriver test innan man skriver kod. Anledningen till att metoden utvecklats och används är av intresse. Ytterligare en frågeställning är om metoden är mekanisk med fasta regler eller om den är flexibel. Som underlag för arbetet har använts litteratur, i vilka metoden beskrivits utförligt med illustrerande exempel.

I kapitel 2 beskrivs bakgrunden till testdriven utveckling samt extremprogrammering (eng. Extreme Programming, XP) eftersom testdriven utveckling är utvecklad från det. Avsikten är att ge en inblick i de tanksätt som krävs för att arbeta agilt. Ofta används testdriven utveckling inom extremprogrammering. Kapitel 3 behandlar mer specifikt vad metoden går ut på och hur man ska tillämpa den. Varje programmerare kan modifiera tekniken enligt personlig smak men huvudprinciperna ska ändå följas. För att belysa metoden presenteras konkreta exempel här. Java används i de flesta exempel. Eftersom testning är en väsentlig del i testdriven utveckling presenteras och beskrivs testning kortfattat i kapitel 3.1. JUnit har använts som grund för exemplen. En del av programmerarens vardag är att fastna i programutvecklingen utan att hitta medel att gå vidare. Inom testdriven utveckling finns det mönster och strategier för att komma vidare i produktionen. Ibland krävs det radikala lösningar medan det ibland räcker med mindre åtgärder. En del av dessa strategier behandlas närmare i kapitel 3.3. En viktig del inom testdriven utveckling är omfaktoriseringen, som beskrivs närmare i kapitel 3.3.4. Några för- och nackdelar med testdriven utveckling presenteras i kapitel 4.

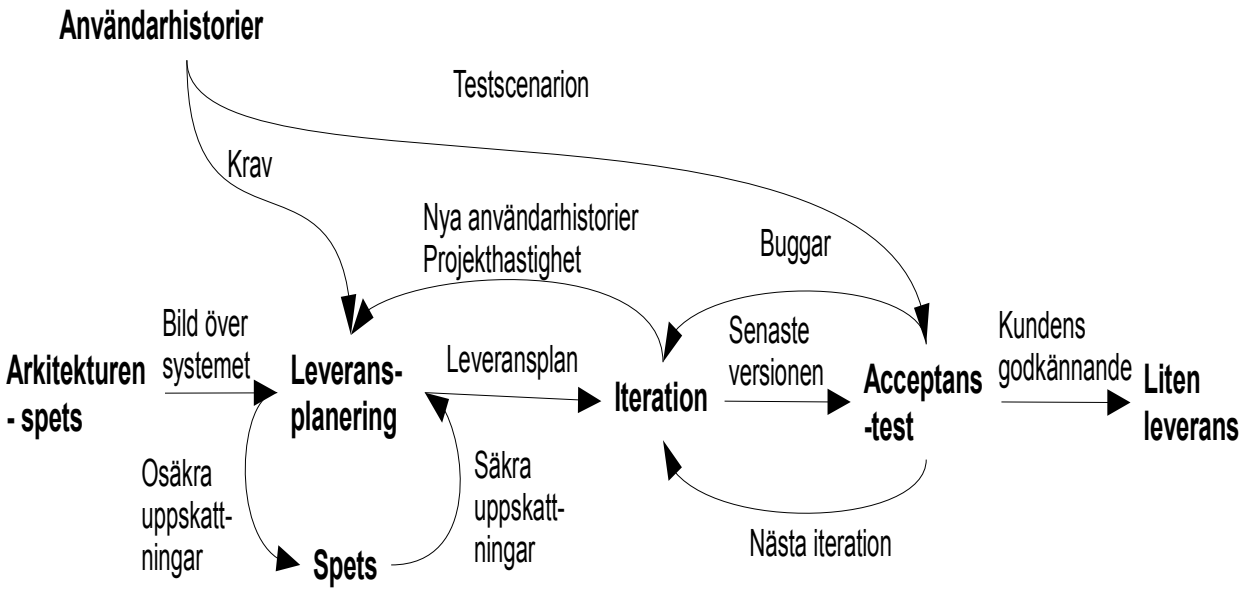
2. Bakgrund

Redan på 1960-talet hänvisar NASA till testdriven utveckling i deras dokumentation angående Mercuryprogrammet [8]. Ofta används testdriven utveckling inom extremprogrammering men den kan även tillämpas som en enskild metod. Att testa först eller i ett tidigt skede är inget nytt inom mjukvaruutvecklingen men testdriven utveckling för detta arbetssätt till en extrem nivå, där man alltid ska testa först. Det första projektet som använde extremprogrammering som utvecklingsmetod startade 1996 av Kent Beck. Han arbetade för Chrysler Corporation och hade som uppgift att göra ett splittrat löneberäkningssystem enhetligt.[4] Han utnyttjade de arbetssätt som präglar extremprogrammering idag, nämligen kommunikation mellan leverantör och kund, strävan till enkel och ren kod, respons genom att testa produkten under hela utvecklingstiden, korta leveranstider med möjlighet till förändringar i produkten, respekt gentemot varje medlem i teamet samt mod att anpassa sig till förändrade krav och teknik. Även om testdriven utveckling för det mesta anses som en särskild utvecklingsmetod går många arbetssätt och principer hand i hand med extremprogrammeringens. [5] Alla agila metoder följer det agila manifestet (se bilaga 1), varifrån de byggt ut fler specifika regler och sedvänjor för en viss metod.

2.1 Extremprogrammering - en översikt

Mjukvaruutveckling enligt extremprogrammering präglas av goda vanor, vilka utvecklingsteamet ska följa. De följer enkla regler kring planering, ledarskap, design, kodning och testning. "Användarhistorier" (eng. user stories) krävs för att utvecklinsteamet ska veta vad de ska utveckla. Kunden/användaren/kravställaren står för grunden till användarhistorierna och ska således beskriva vad som önskas av systemet. När kraven och användingshistorierna är klara kan teamet börja planera utvecklingsprocessen. Ett extremprogrammeringsteam består av kunder, programutvecklare och chefer. Planeringen går ut på att tidsuppskatta varje användarhistoria. En användarhistoria ska göras inom 1-3 veckor. Varje användarhistoria delas upp i mindre uppgifter, som ska kunna göras på ungefär 1-3 dagar. Kunden kan när som helst ändra eller lägga till krav som tidigare ställts på systemet.

Arbetssättet inom extremprogrammering består av parprogrammering. Testdriven utveckling kan tillämpas men är inte nödvändigt. Inom extremprogrammering ska produktionskoden testas från och med samma dag som projektet startar. Den som har lagt till en funktion i produktionskoden ska även testa hela systemet med de automatiska testerna för att vara säker på att den nyss tillagda funktionen inte har orsakat funktionsfel i andra delar av systemet. Om systemet inte fungerar ska programmeraren åtgärda felen ända tills systemet fungerar igen. En viktig del i extremprogrammering är att alltid ha en exekverbar version av systemet, därför ska ingen utvecklare lägga till produktionskod som inte fungerar. En ytterligare viktig del i extremprogrammering är omfaktorisering. Principen om snygg och ren kod är viktig för att andra utvecklare enkelt ska förstå vad koden gör. Som utvecklare ska man reflektera över koden och kontrollera att det inte finns duplikat i koden. Systemet byggs upp stegvis genom kontinuerliga tillägg och förändringar i systemet med målet färdig produkt.[3] Nedan en illustration över utvecklingsprocessen inom extremprogrammering.



Figur 1. Översikt av ett extremprogrammeringsprojekt. [6]

3. Testdriven utveckling

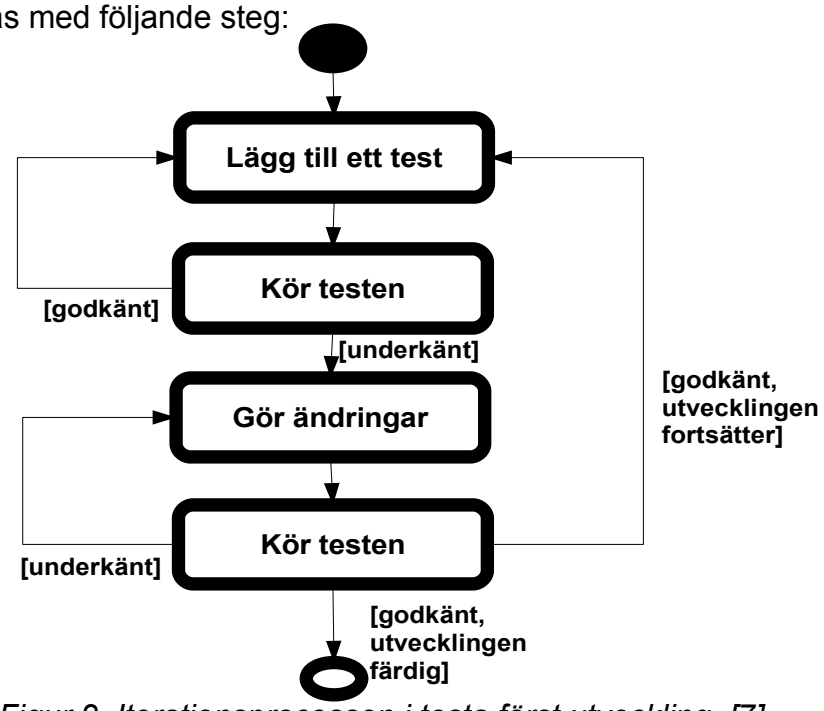
Testdriven utveckling är en inkrementell utvecklings-, design- och programmeringsmetod. Metoden går ut på att skriva ett test för en viss funktion som ingår i ett krav och därefter skriva programkoden. Testet ska misslyckas efter att det har skrivits eftersom enbart nya funktioner ska läggas till i systemet. Testdriven utveckling är en iterativ metod där programkoden bearbetas ända tills testet ger önskat resultat, d.v.s. exekverar utan felmeddelanden. När testet har lyckats ska koden ännu omfaktoriseras, d.v.s. städas upp och reduceras från duplikat. All kod, både produktionskoden och testkoden, ska vara så ren och enkel som möjligt.

Eftersom metoden bygger på att utföra test innan kodning får programmeraren respons varje gång ett test exekveras. På så sätt upprätthåller programmeraren information om systemets nuvarande tillstånd och får uppgifter, från testresultaten, om vad som ytterligare måste åtgärdas för att testet ska lyckas. Metoden bygger på en utveckling där testning och kodning fortlöper hand i hand under hela utvecklingsprocessen. [1] Testdriven utveckling kan beskrivas som en metod där man testar först med tillägget omfaktorisering [7]. Eftersom testdriven utveckling är en utvecklinsmetod och inte en testmetod krävs det att övrig testning, som t.ex. integrations-, system- och acceptanstester, ytterligare måste göras.

Iterationsprocessen kan beskrivas med följande steg:

1. Skriv ett test
2. Kör testen för att se dem misslyckas
3. Ändra/Lägg till kod
4. Kör testen för att se dem lyckas
5. Omfaktorisera

[2]



Figur 2. Iterationsprocessen i testa först utveckling. [7]

3.1 Testning

Det finns olika typer av tester inom programvaruutveckling. *Enhetstester* testar enskilda komponenter för att kontrollera att de gör det de ska. Testdriven utveckling bygger på enhetstester. *Regressionstester* testar att redan implementerade delar fungerar efter att ändringar eller tillägg gjorts i dem. *Integrationstester* testar att olika delar fungerar tillsammans. Modultester testar att beroende komponenter fungerar tillsammans. *Delsystemtester* testar att modulgrupper fungerar tillsammans. *Systemtester* testar att hela applikationen fungerar. *Acceptanstester* testar att programmet gör vad kunden/kravställaren önskar. [3]

En testklass och testmetod benämns genom att inkludera ordet test i klassnamnet, t.ex. `testMultiplication()`. Klassens metoder betecknas med märkkod innan metodens början för att indikera dess syfte. Exempel på märkkoder som används i JUnit:

- @Test - Indikerar att metoden är ett test.
- @Test(expected = ExceptionClass.class) - Indikerar att ett undantag ska testas.
- @Test(timeout = tid_i_millisekunder) - Indikerar att ett test ska avbrytas om det exekverar längre än tid_i_millisekunder.
- @Ignore - Inidkerar att metoden inte ska exekveras.
- @Before - Indikerar att metoden ska exekveras innan varje testmetod.
- @After - Indikerar att metoden ska exekveras direkt efter varje testmetod.
- @BeforeClass - Indikerar att metoden ska exekveras innan hela testklassen, d.v.s. innan någon av metoderna exekveras.
- @AfterClass - Indikerar att metoden ska exekveras efter att alla metoder i klassen har exekverat.
- @RunWith(Parameterized.class) - Indikerar att testklassen använder sig av olika värden.
- @Parameters - Ger parametervärden som indata till testmetoder. Returnerar element i en array som används som testvärden.

JUnit har en färdig testklass vid namn `Assert`, som innehåller olika testmetoder. Assert kommer från engelskan och betyder bl.a. påstå, vilket kan underlätta förståelsen av vad metoderna gör. Utöver de inbyggda metoderna kan man skapa egna. Nedan följer en lista på `Assert`- metoder och vad de testar:

- `fail(String)` – Skriver ut ett felmeddelande. Kan t.ex. användas på ställen i testmetoder dit exekveringen inte är meningen att nå eller i testmetoder som inte ännu är implementerade.
- `assertTrue(String message, boolean condition)` – Om `condition` är sant så lyckas testet anars misslyckas det och `message` skrivs ut.
- `assertFalse(String message, boolean condition)` – Om `condition` är falskt så lyckas testet annars misslyckas det och `message` skrivs ut.
- `assertEquals(String message, expected, actual)` – Om `expected` och `actual` är lika så lyckas testet annars misslyckas det och `message` skrivs ut.
- `assertEquals(String message, expected, actual, tolerance)` – Om `expected` och `actual` är lika med högst `tolerance` som skillnad så lyckas testet annars misslyckas det och `message` skrivs ut. T.ex. `expected = 0.35, actual = 0.37` med `tolerance = 0.03` resulterar i lyckat test medan `tolerance = 0.02` resulterar i misslyckat test.
- `assertNull(String message, object)` – Om objektet inte är `null` så returneras ett felmeddelande.
- `assertNotNull(String message, object)` – Om objektet är `null` så returneras ett felmeddelande.
- `assertSame(String message, expected, actual)` – Kontrollerar att båda variablerna hänvisar till samma objekt. Om inte returneras felmeddelandet.
- `assertNotSame(String message, expected, actual)` – Kontrollerar att båda objekten hänvisar till olika objekt. Om inte returneras felmeddelande.

Exempel. En testklass som använder `assertTrue()` *för att testa om talet 5 är större än talet 10.*

```
import static org.junit.Assert.*;
import org.junit.*;
public class TestClass{
     @Test
     public void testMetod(){
     int aNumber = 5;
     assertTrue("fail",aNumber > 10);
     }
}
```

Förutom testmetoder med beteckningar av märkkod kan även testsviter användas. I en testsvit kan man specificera vilka testmetoder och testklasser man vill exekvera samt spara uppgifter om exekveringen i ett särskilt resultat. Först skapar man en testsvit genom att skapa en ny testsvit och sedan samlar man alla testmetoder och -klasser i sviten.

Exempel. En testsvit där alla metoder i klassen `TestClass` *ska exekveras och metoden* `test1` *från* `TestClassName` *ska exekveras. Testerna i sviten exekveras med* `run` *och resultatet läggs i* `result`*.*

```
public class TestSuiteExample{
     TestSuite suite = new TestSuite();
     suite.addTest(TestClass.class)
     suite.addTest(new TestClassName("test1"));
     TestResult result = new TestResult();
     suite.run(result);
}
```

[3]

3.2 Iteration och implementering

Testdriven utveckling kännetecknas av små byggstenar som stegvis läggs till i produkten. Produktionskoden ska kunna utökas med korta instruktioner men den kan även byggas på med större steg. Även om det är bra att ha en vision av produktens helhet är det inte nödvändigt för den enskilde programmeraren. Designen bildas stegvis medan tester och produktionskod implementeras. Programmerarens huvuduppgift är att fokusera på den

funktion som för tillfället ska läggas till. Utvecklaren ska fundera på vad testet ska åstadkomma och hitta på exempel för testet. Processen ska inte ta alltför länge. Om det går flera timmar åt att fundera vad testet ska göra så måste testen delas upp i mindre, mer kontrollerbara, delar eftersom det är meningen att man snabbt ska implementera test och kod som sedan exekveras. Parprogrammering är ett alternativ för att effektivisera arbetet. När det är klart vad testet ska göra så ska man skriva enhetstestet för funktionen.

Exempel. Ett program som ska kunna hantera flera olika valutor. Programmet ska kunna addera och multiplicera två tal, som har olika valutor, och ge resultatet i den ena valutan.

För att lösa problemet måste vi först göra en lista på vad som skall göras. Som exempel använder vi 5 dollar och 10 franc, som ska kunna adderas med varandra med beaktande av valutakursen 2:1. En vanlig multiplikation ska även kunna utföras, där exemplet blir 5 dollar multiplicerat med 2. Listan blir för tillfället enligt följande:

```
$5 + 10 CHF = $10 om kursen är 2:1
$5 * 2 = $10
```

När vi vet vad vi vill att programmet ska göra skriver vi det första testet. Det kan se ut på följande sätt:

```
public void testMultiplication(){
     Dollar five = new Dollar(5);
     five.times(2);
     assertEquals(10,five.amount);
}
```

Testet ovan kommer inte att exekvera eftersom det inte finns någon produktionskod alls i det här skedet. Nästa steg går ut på att iterativt lägga till produktionskod och exekvera testet ända tills testet godkänns. För att ovannämnda test ska kunna exekvera krävs en klass `Dollar`, en konstruktor `Dollar(int)` , en metod `times(int)` och ett fält `amount`. Produktionskoden kunde se ut enligt följande:

```
class Dollar{
     int amount;
     public Dollar(int amount){}
     void times(int multiplier){}
}
```

Testet exekverar men ger inte det förväntade resultatet 10 eftersom `amount` inte har tilldelats något värde i klassen `Dollar`. Det här går att åtgärda genom att tilldela `amount` värdet 10.

```
class Dollar{
     int amount = 10;
     ...
}
```

Testet kommer att godkännas och ge rätt resultat men lösningen är inte generell. Enligt principer inom testdriven ska kod generaliseras om det finns fler än ett exempel på vad testet ska kunna utföra(triangulering). I det här fallet vill vi att programmet ska kunna multiplicera två godtyckliga tal, vilket betyder att antal exempel för vad testet ska kunna göra är fler än ett. Vi måste generalisera koden. Koden innehåller duplikat, som måste elimineras. Talet 10 har vi själva räknat ut genom att multiplicera 5 * 2. Nästa steg kunde vara följande:

```
...
int amount;
void times(int multiplier){
     amount = 5 * 2;
}
...
```

De här stegen kan tänkas väldigt små men en del av testdriven utveckling är kunskapen och möjligheten att kunna ta små steg för att kunna ta stora. Stegen behöver inte vara små men programmeraren ska kunna ta korta utvecklingssteg vid behov. Nästa steg är att eliminera duplikat av 5 och 2, alternativt tänka sig att eliminera konstanter. Talen finns redan i `testMultiplication()`. Produktionskoden för `times(int multiplier)` kan ändras till följande:

```
void times(int multiplier){
     amount = amount * 2;
}
```

Koden innehåller ännu duplikat och konstanter, som kan elimineras:

```
void times(int multiplier){
     amount = amount * multiplier;
}
```

Efter ytterligare eliminering av duplikat kunde koden se ut enligt följande:

```
...
Dollar(int amount){
     this.amount = amount;
}
void times(int multiplier){
     amount *= multiplier;
}
...
```

Nu innehåller programmet en funktion för att utföra punkt två från listan. Vi kan således stryka den punkten. Medan vi har skrivit koden har vi märkt att vi måste lägga till några punkter på listan, såsom avrundning. Den nya listan ser då ut enligt följande:

```
$5 + 10 CHF = $10 om kursen är 2:1
~~$5 * 2 = $10~~
Gör "amount" till privat
Bieffekter av Dollar?
Avrundning
```

Enligt ovan beskrivna tillvägagångssätt fortsätter programutvecklingen i testdriven utveckling ända tills produktionskoden är färdig för leverans. Processen ska vara smidig och snabb. Först ska grunderna för testet skrivas, som kan bestå av en enkel historia. Sedan kan man lägga till gränssnitt, klasser och metoder i testet. Testet ska exekveras så fort som möjligt. Testresultaten ger respons och berättar vad som måste göras till näst. Om det är självklart vilken kod som ska implementeras så ska den skrivas. Om det tar en minut att implementera den självklara koden så ska en snabbare lösning skrivas, som kan kodas på några sekunder. Den enklaste och snabbaste lösningen till att få testet att lyckas ska implementeras eftersom målet är att snabbt få ett fungerande test. När testet lyckas så ska koden omfaktoriseras för att få en snygg, fungerande, ren och generell kod. [2]

3.3 Mönster - Strategier

Det finns en del strategier och mönster man kan följa vid tillämpning av testdriven utveckling. Varje test ska vara isolerat från övriga tester, med andra ord ska ett test inte vara orsaken till att ett annat test misslyckas. Varje test ska kunna exekveras enskillt eller i

grupper. Innan man börjar skriva test ska man göra en lista över de test man tror är nödvändiga för önskat resultat. En viktig sak är att inte gå vidare i utvecklingen innan man vet var i utvecklingen man befinner sig just nu. Det går eventuellt att hålla allt man ska och vill göra i huvudet, men en sådan strategi blir lätt ogreppbar. Ett bättre tillvägagångssätt är att göra en lista över vad som just nu behövs och t.ex. rita upp en översikt över projektet som kan hängas på väggen. Projektet kan delas in i test som ska implementeras just nu och test som kan göras senare. Listan över det som ska göras just nu är bättre att hålla kort eftersom målet med testdriven utveckling är att snabbt implementera test och snabbt få dem att lyckas.

Vid traditionell mjukvaruutveckling där man testar produktionskoden sist kan det hända att testningen inte görs ordentligt eller i tillräckligt stor utsträckning p.g.a. tidsbrist och stress. Om utvecklingsteamet inför en regel om att alltid testa först undviker man eventuellt denna stressfaktor. Vad gäller själva testen och innehållet i dem kan det löna sig att skriva `Assert`- metoderna innan övrig kod i testmetoderna. Vad som känns bekvämast och smidigast är dock upp till varje programmerare.

Exempel. Ett test för kommunikation mellan två system över en socket. Metoden testar att socketen är stängd efter att programmet utfört det begärda och att strängen "abc" har lästs från det andra systemet. Först lägger vi till `Assert`*-metoderna:*

```
testCompleteTransaction(){
      ...
      assertTrue(reader.isClosed());
      assertEquals("abc",reply.contents());
}
```

Sedan fyller vi ut metoden med nödvändig kod för att testet ska lyckas. Först öppnar vi servern och kontaktar den, sedan läser vi in data varefter testen utförs:

```
testCompleteTransaction(){
      Server writer = Server(defaultPort(),"abc");
      Socket reader = Socket("localhost",defaultPort());
      Buffer reply = reader.contents();
      assertTrue(reader.isClosed());
      assertEquals("abc",reply.contents());
}
```

Den indata som används i testen ska vara lättanvänd och lättförståelig. Alltför långa strängar eller orealistiskt stora tal är onödiga och svåra att läsa. Om möjligt ska antalet indata för testerna hållas låga. Det är onödigt att testa tio olika tal om det räcker att testa tre. Variabelnamn och metodnamn ska relatera till syftet och funktionen anknytna till dem. Beskrivande namn bidrar till enkel och förståelig kod. Indatan för testerna kan plockas från verkligheten för att få så realistiska tester som möjligt. De tester som skrivs är inte enbart menade för datorer utan även för andra programmerare. Meningen är att underlätta arbetet för den programmerare som kommer att arbeta med koden till näst. Testen och koden ska fungera som huvudsaklig dokumentation.

Exempel. Vi vill utföra valutaomvandling där provisionen för omvandlingen är 1.5 % och kursen mellan USD och GBP är 2:1. Vi vill skriva ett test för att omvandla 100 USD till GBP. Resultatet borde bli 50 GBP - 1.5 % = 49.25 GBP. Det finns olika sätt att skriva testet och nedan följer två av dem:

1)

```
...
Bank bank = new Bank();
bank.addRate("USD","GBP",STANDARD_RATE);
bank.commission(STANDARD_COMMISSION);
Money result = bank.convert(new Note(100,"USD"),"GBP");
assertEquals(new Note(49.25,"GBP"),result);
...
```

2)

```
...
Bank bank = new Bank();
bank.addRate("USD","GBP",2);
bank.commission(0.015);
Money result = bank.convert(new Note(100,"USD"),"GBP");
assertEquals(new Note(100/2*(1-0.015),"GBP"),result);
...
```

Båda versionerna fungerar men vi ska inte enbart tänka på vad som fungerar eller inte. Vi ska tänka på programutvecklaren som ser koden för första gången och då är version 2 mer beskrivande än version ett. T.ex. tvingas programmeraren fundera på vad `STANDARD_RATE` och `STANDARD_COMMISSION` är i version 1. [2]

Det finns flera strategier för att få misslyckade test att lyckas. Några exempel på dessa strategier är den självklara implementeringen, "fejka" (eng. Fake It) och triangulering. När en programmerare vet vad som skall kodas används den självklara lösningen. Om det inte är självklart vad som ska implementeras så används fejka, som går ut på att returnera en konstant istället för en variabel. Med små steg framåt ersätter programmeraren konstanten med variabler och på så sätt bildas den riktiga koden. Om designen är oklar kan triangulering användas. Triangulering går ut på att generalisera kod enbart om det finns fler än ett exempel för testet. [1].

3.3.1 Strategier vid röd balk

När det är dags att skriva det första testet väljer man ett av testen från listan, som man gjort tidigare. Vilket test man väljer är upp till en själv men en strategi är att välja det som verkar ha en självklar lösning. Man kan välja att skriva test uppifrån ner eller nerifrån upp. Uppifrån ner går ut på att skriva test för listpunktens hela funktion först och sedan implementera detaljer. Nerifrån upp går ut på att bygga upp funktionen med en detalj i taget ända tills hela funktionen är färdig. Varje programmerare kan hitta sitt eget tillvägagångssätt. Vid val av nästa test är det värt att fundera på var i systemet nästa funktion ska finnas. Testen ska vara realistiska så att man vet var i systemet de ska finnas samt veta vilka indata och utdata som behövs för funktionen.

Vid tillämpning av testdriven utveckling ska man tänka på vad man vill lösa istället för hur. Ett sätt att komma in i sådana tankebanor är att beskriva realistiska exempel på vad man vill att funktionen ska utföra. T.ex. kan man beskriva ett fall så här: "Om jag har en NN och lägger till en XX så borde resultatet vara YY". Detta abstrakta tankesätt kan tillämpas på högre och lägre nivåer i programutvecklingen.

Ibland kan utvecklaren lida av för många idéer, vilket inte heller är enbart fördelaktigt. Man kommer på allt fler funktioner till systemet och det är svårt att inte börja arbeta med dem direkt. Det är dock viktigt att fokusera på den uppgift som för tillfället ska utföras för att ha kontroll över arbetet och för att nå ett professionellt, fungerande resultat. Man kan skriva ner idéerna för att senare arbeta med dem och lägga till dem till testlistan.

Agila metoder handlar inte enbart om att planera och implementera kod utan även om att teammedlemmarna ska trivas och må bra. Ibland stöter en utvecklare på hinder, t.ex.

kommer inte vidare i programutvecklingen. Det är inte meningen att någon ska programmera tolv timmar i sträck utan pauser. Att tvångsmässigt försöka hitta en lösning orsakar enbart frustation och en trött programmerare tenderar att bli ännu tröttare och därmed mindre produktiv. Det är alltså viktigt att ha normala arbetstider, pauser, lediga veckoslut och normala semesterdagar.

Om produktionskoden är alltför komplex och ogreppbar är en lösning att börja helt från början. Ibland börjar programmeringen bra men blir alltmer rörig och oklar ju mer den bearbetas. När man inte vet hur man ska gå vidare är en lösning att stryka hela koden. Ett sätt att undvika rörig kod är att parprogrammera. Då kan man byta programmerare med jämna mellanrum för att få en färsk syn på koden.

3.3.2 Strategier vid grön balk

Fejka för att få testet att lyckas. Metoden går ut på att först returnera en konstant för att få testet att lyckas. Efter att balken är grön istället för röd så ändrar man gradvis konstanten till en variabel.

Exempel. Stegvis operation med metoden fejka

Steg 1:

```
return "1 run, 0 failed"
```

Steg 2:

```
return "%d run, 0 failed" % self.runCount
```

Steg 3:

```
return "%d run, %d failed" % (self.runCount, self failureCount)
```

Det kan tänkas onödigt att göra dessa små steg. Men det är även ett sätt att ha bättre kontroll över vad man gör. Det är bättre att ha ett exekverbart test än ett som inte gör det. Efter varje steg körs testet och man får direkt respons på det man skrivit. Det är enklare att rätta fel som gjorts mellan två korta testkörningar än att försöka hitta fel när ett helt program eller system är programmerat. Metoden om att först fejka tester har även psykologisk effekt, det känns bättre att se en grön balk än att se en röd. När balken är grön vet man var man just då befinner sig i programutvecklingen. Att utveckla stegvis gör även att fokusen ligger på det som görs nu, inte på framtida problem. Den kod som skrivs

med fejkmetoder ska även omfaktoriseras för att få en ren och lättförståelig kod.

Triangulering är en annan metod för att komma från röd balk till grön balk. Metoden går ut på att generalisera kod enbart om det finns fler än ett exempel på vad testet ska utföra.

Exempel. En funktion som ska kunna addera två tal implementeras med triangulering.

```
public void testSum(){
     assertEquals(4,plus(3,1));
}
private int plus(int tal1, int tal2){
     return 4;
}
```

Vid triangulering måste vi lägga till ett exempel:

```
public void testSum(){
     assertEquals(4,plus(3,1));
     assertEquals(7,plus(3,4));
}
```

När testet innehåller två exempel måste produktionskoden generaliseras. Då kan metoden `plus()` ändras på följande sätt:

```
private int plus(int tal1, int tal2){
     return tal1 + tal2;
}
```

Triangulering kräver minst två Assert-metoder i testklassen för att förfarandet ska följa reglerna. Metoden är mest användbar då det är mycket oklart hur implementeringen ska göras korrekt enligt metoderna fejka eller den självklara implementeringen.

Den självklara implementeringen är ytterligare ett sätt att få tester godkända. Det sättet används som första alternativ, vilket betyder att om det är självklart vad ett test ska göra så kodas testet och produktionskoden med självklar kod. Ovannämnda exempel kunde lätt implementeras direkt med självklar kod. Om det känns självklart vad som ska skrivas så ska man skriva det, förutsatt att det går snabbt. Om det inte går snabbt så ska man använda någon av de andra metoderna för att så snabbt som möjligt exekvera testet. Vid användning av denna metod ska man vara rätt säker på vad som ska skrivas för att få testet att godkännas. Om man ändrat på testet några gånger utan att det godkänns ska man ändra strategi. Syftet är att snabbt få ett exekverbart och godkänt test.

En till många är en metod som kan användas då man arbetar med mängder av objekt, t.ex. en räcka. Metoden går ut på att först skriva test och produktionskod för ett element och sedan utöka testet och produktionskoden med mängder.

Exempel. En funktion ska addera varje tal i en räcka med varandra. Vi vill att funktionen ska kunna addera en räcka med talen 5 och 7.

Först gör vi en testklass för ett element, i vårt fall 5.

```
public void testSum(){
     assertEquals(5,sum(5));
}
private int sum(int value){
     return value;
}
```

Nästa steg går ut på att ändra koden så att den kan ta emot en räcka. Koden blir enligt följande:

```
public void testSum(){
     assertEquals(5,sum(5,new int[] {5}));
}
private int sum(int value,int[] values){
     return value;
}
```

Sedan kan `sum(int, int[])` modifieras så att additionen utförs och `value` kan elimineras:

```
private int sum(int[] values){
     int sum = 0;
     for(int i = 0; i < values.length; i++){
          sum += values[i];
     }
     return sum;
}
```

Testmetoden kan sedan ändras så att `value` elimineras och funktionen kan addera talen 5 och 7:

```
public void testSum(){
     assertEquals(12,sum(new int[] {5,7}));
}
```

3.3.4 Omfaktorisering

Omfaktorisering är en väsentlig del inom testdriven utveckling. Det handlar om att ändra på systemets design i små steg. Inom testdriven utveckling omfaktoriserar man enbart de test som fungerar. Som exempel kallas förändring av en konstant till en variabel för omfaktorisering inom testdriven utveckling. Den del kod man vill omfaktorisera ska isoleras från övrig kod, med andra ord ska man enbart fokusera på den delen.

Om två koddelar ser liknande ut ska man gradvis förena dem. De ska inte förenas innan de ser helt likadana ut. Små steg krävs för att vara säker på att systemet inte ändrar sig p.g.a. det som ändras på. Som exempel på kod som kan förenas är två loopar, vars struktur liknar varandras. Ett annat exempel är två villkorssatser som påminner om varandra. Ett tredje exempel är liknande metoder och klasser. Om det går att göra ovannämnda exempel likadana utan att testresultaten ändras så ska man göra dem identiska och sedan eliminera den ena koden.

När man vill ändra en kod från en representation till en annan ska man inte radera den ursprungliga koden innan man börjar med den nya utan man ska först duplicera den och därefter radera den gamla. Innan man raderar den gamla koden ska det finnas två varianter av den kod man vill byta ut. Exempel på metoden finns i exemplet ovan om räckor där man vill övergå från ett element till en mängd av element.

Man kan göra en lång och komplicerad metod mer lättläst genom att plocka ut delar som utför samma uppgift och skapa en ny metod av delarna. Exempel på sådana koddelar är en loop, delar av en loop och villkorssatser. Kontrollera att det inte finns tilldelningar av tillfälliga variabler utanför den del som ska plockas ut. Sedan kan man kopiera koddelen till en ny metod och kompilera den. Varje tillfällig variabel eller parameter i den ursprungliga metoden ska läggas till som parameter i den nya metoden. Sedan ska man kalla på den nya metoden från den ursprungliga. Ibland börjar koden kännas alltför splittrad och komplicerad, då kan man gå åt andra hållet, d.v.s. ersätta ett anrop med det riktiga metodinnehållet.

Om ett nytt test kan sammankopplas med ett redan befintligt kan det löna sig att skapa ett gränssnitt som innehåller gemensamma funktioner. Exempel på ett dylikt fall är en befintlig metod `Rektangel` där en metod `Oval` ska läggas till. Som metodnamnen syftar på är de båda funktioner för att skapa former. Produktionskoden blir tydligare och enklare om dessa

metoder ingår i ett gränssnitt, som t.ex. kan kallas `Form`. I testdriven utveckling ska man först deklarera gränssnittet, sedan ska den existerande klassen implementera gränssnittet, nödvändiga metoder ska läggas till i gränssnittet och de variabler som kan ska flyttas till gränssnittet.

Ibland lönar det sig att flytta en metod från en klass till en annan. Om det i en metod skickas information flera gånger till samma objekt kan det löna sig att flytta funktionen till en egen metod i den påkallade klassen. Funktionen ska kopieras till en metod i den andra klassen innan den raderas från den ursprungliga. Ett exempel illustrerar förfarandet:

```
class Shape(){
     ...
     int width = bounds.right() - bounds.left();
     int height = bounds.bottom() - bounds.top();
     int area = width * height;
     ...
}
```

Funtionen skickar information till objektet `bounds` flera gånger, vilket kan förenklas genom att skapa en egen metod `area()` i klassen `Rectangle`:

```
class Rectangle(){
     ...
     public int area(){
          int width = this.right() - this.left();
          int height = this.bottom() - this.top();
     return width * height;
}
```

Metoden i `Shape` skulle då se ut så här:

```
...
int area = bounds.area();
...
```

En metod som kräver många variabler och parametrar lönar det sig att göra till ett eget objekt. Man ska skapa ett objekt med samma parametrar som metoden har. Objektet ska skapas medan metoden ännu existerar. Metodens lokala variabler blir objektets instansvariabler. I den nya klassen ska en metod vid namn `run()` skapas och metoden ska ha samma innehåll som den ursprungliga metoden. I den ursprungliga metoden ska

sedan ett nytt objekt skapas och `run()` ska köras därifrån.

När man ska lägga till parametrar i en metod så ska parametrarna först läggas till det eventuella gränssnittet och sedan till metoden varefter koden ska kompileras. Kompileringsfelmeddelandena används som hjälp för att ändra på anrop av metoden. En metodparameter kan flyttas till en konstruktorparameter genom att först lägga till parametern till konstruktorn, sedan lägga till en instansvariabel med samma namn som parametern och lägga till den i konstruktorn. Steg för steg ska man sedan ändra `parameter` till `this.parameter`, radera referenser till parametern, radera sedan parametern från metoden och alla anrop, eliminera överflödiga `this` från referenserna och namnge variabeln enligt användningsändamål.

4. För- och nackdelar med testdriven utveckling

Fördelar med testdriven utveckling är att programmeraren satsar på att skriva bra test, vilket underlättar för nästa programmerare att förstå koden. Dokumentation för program kan vara bristfällig eller föråldrad medan produktionskoden och testen inte är det. Testen kan således fungera som dokumentation. Följande programutvecklare kan köra test, ändra på dem m.m. för att förstå vad testen gör. En annan fördel är den snabba respons som fås genom att testen exekveras kontinuerligt. Kodtäckningen är alltid 100 % och antalet buggar minskar i allmänhet med 40-90 %. Programutvecklare uppmuntras att skriva kod som är enkel att testa.

Nackdelar med testdriven utveckling är att den tar ungefär 10-30 % längre tid att utveckla programvaran. En annan nackdel är att det är svårt att testa vissa koder. Om designen inte är klar från början av programutvecklingen kan man vara tvungen att gå tillbaka och ändra på ett tidigare gjort test. Det här förlänger utvecklingstiden. Om den önskade funktionen är komplex kan det vara svårt att skriva ett test för den. Ytterligare en nackdel är att mängden kod blir mer än om man inte följer testdriven utveckling eftersom varje del kod är testad. Om någon funktion ska ändras i programmet så ska testet och produktionskoden ändras, vilket medför förlängd implementeringstid.

Som både för- och nackdel kan tänkas att testdriven utveckling garanterar att enheterna är nära buggfria men metoden garanterar inte att varje modul och enhet fungerar tillsammans i applikationen. Ändå är det enklare att rätta till fel anknytna till att delarna inte fungerar med varandra med vetskapen om att varje enhet fungerar som den ska. [9][1]

5. Sammanfattning

Testdriven utveckling är en programutvecklingsmetod som kräver en tankeomställare hos de programmerare, som är vana med att testa sist. Det krävs disciplin att hålla fast vid grundregeln om att alltid testa först. Man ska fokusera på vad istället för hur. Metoden tenderar vara flexibel med undantaget att testet alltid ska skrivas först. I övrigt finns det tydliga strategier för hur man kan komma vidare i programutvecklingen om man har fastnat. Metoden är extrem och kräver mod att våga tillämpa, speciellt då utgångspunkten är att ens kod ska misslyckas i början. Tankar om att en röd balk betyder personligt misslyckande måste förkastas vid tillämpning av testdriven utveckling.

De små, korta stegen man som testdriven utvecklare ska kunna ta kan tänkas svåra att acceptera eftersom de verkar för små och enkla. En annan aspekt är att slutresultatet av koden blir renare och tydligare om den är uppbyggd med mindre steg. Kraven om omfaktorisering bidrar även till prydligare kod. Ytterligare en anledning till att metoden används kan vara att ingen redan existerande funktion kan läggas till i produktionskoden. På så vis undviks dubbelarbete.

Traditionella utvecklingsmetoder som betonar dokumentation och kommentarer i koden kan glömmas. I testdriven utveckling är det koden och testen som är dokumentationen. Om koden behöver kommentar så är den inte välgjord. Metoden kan tänkas mer effektiv vid tillämpning av parprogrammering eftersom olika aspekter då beaktas under arbetets gång. Testdriven utveckling kan ses som en växande, framtidens utvecklingsmetod.

Källor

Bok

[1] Lech Madeyski, Test-Driven Development: An Empirical Evaluation of Agile Practice, Springer, 2010

[2] Kent Beck, Test-Driven Development: By Example, Addison-Wesley, 2003

[3] Björn Eiderbäck, Extremprogrammering, Studentlitteratur AB, 2007

[8] Craig Larman, Iterative and Incremental Development: A Brief History, IEEE Computer, 2003

Nätpublikation

[4] Lee Copeland, http://www.computerworld.com/s/article/66192/Extreme_Programming?taxonomyId=063, 3.12.2001

[5] Don Wells, http://www.extremeprogramming.org/, 28.9.2009

[9] Michael Feathers, Steve Freeman, http://www.infoq.com/presentations/tdd-ten-years-later, 18.8.2009

Figurer

[6] Figur 1. Don Wells, www.extremeprogramming.org/map/project.html, 2000

[7] FIgur 2. Scott W. Ambler, http://www.agiledata.org/essays/tdd.html, 2012

Bilaga 1. "Manifesto for Agile Software Development", http://www.agilemanifesto.org/, 2013