Mittuniversitetet

MID SWEDEN UNIVERSITY

Metoder och verktyg i mjukvaruprojekt

Lektion 5 – Testning

Lektionen och den tillhörande inlämningsuppgiften skall ge en introduktion till mjukvarutestning!

Innehållsförteckning

1 Test-Driven Utveckling (*TDD*)

Test-Driven utveckling (TDD, Test-Driven Development) går ut på att produktionskoden testas under utvecklingensarbetets gång. Mer specifikt skriver man testkoden **före** produktionskoden - alltså koden som testas. Normalt är det programmeraren själv (istället för någon dedikerad testare) som skapar testerna.

En sak, som är viktig att komma ihåg när det gäller diskussionerna kring plus och minus nedan, är att man kan tillämpa tekniker på ett mer eller mindre lyckat sätt. Så att något beskrivs som ett plus garanterar inte att man får en viss positiv effekt, och att något beskrivs som ett minus garanterar inte att man får en viss negativ effekt. Det krävs omdömme för att använda olika tekniker på rätt sätt. Så för- och nackdelar handlar mer om möjligheter man kan försöka utnyttja till sin fördel och fallgropar man bör försöka undvika.

1.1 Plus

Genom att testerna skapas först, skrivs produktionskoden alltid som en reaktion på testerna, och förhoppningsvis (om det är bra tester) kan man lita på att kod som klarar testerna är korrekt.

Det kan vara värt att hålla isär 1) att ha tester, och 2) att ha testerna **först** .

Att ha en stor uppsättning (bra) testfall (1) som kan köras automatiskt, gör att man kan göra ändringar i design och implementation utan att behöva vara rädd för att förstöra. Givetvis använder vi ju versionshantering, och ändringar som leder till en återvändsgränd är enkla att kasta bort för att återgå till en tidigare version. Men detta löser ett separat problem. Vad gör du om du har testat programmet manuellt, och så städar du lite i koden? Fungerar programmet fortfarande? Det är ju inte meningen att ändringarna ska påverka funktionaliteten, men tänk om något gick snett. Hur kan du vara säker utan att upprepa testningen? Så testerna fungerar som ett slags skyddsnät som gör att vi inte behöver vara rädda för att något ska gå sönder om vi petar i koden.

Att skriva tester före implementationen (2) har flera positiva effekter. Vi måste designa produktionskoden så att den är testbar, och testbar kod är rent generellt (som tumregel) renare. Testkoden tvingar oss ofta att hålla oss till bra designprinciper. Bland annat för att den måste vara mer modulär, med fokus på gränssnitt, och med parametrar istället för hemliga magiska värden (dependency injection). Vi tvingas också ha en tydlig bild av vad komponenten ska göra och hur den ska användas, annars kan vi inte skriva testerna. Så det blir lättare att undvika den ganska naturliga tendensen att man bara sätter sig och börjar hacka kod i något hörn av vad som förhoppningsvis vid ett senare tillfälle visar sig vara programmet vi hade tänkt skapa.

Genom att skriva testet först tvingas vi ta ställning till vad rätt beteende faktiskt är. Vad är det `foo(13)` egentligen ska göra? Det är för lätt att titta på resultatet produktionskoden ger och tänka "jomen det ser väl rimligt ut", och anta att det är det testet ska kolla. En bra liknelse är när man löser problem i matematik eller fysik. Då är det värdefullt att först skaffa sig en uppfattning om vad man förväntar sig **innan** man tar fram lösningen i detalj. Finns det något minimum eller maximum som svaret **måste** ligga mellan? Är laddningen positiv eller negativ? Vem vet om jag kommer att lista ut rätt värde för vinkeln v, men jag vet i alla fall att, vad jag än kommer fram till, måste summan av vinklarna i triangeln vara 180 grader. Annars har vi något som är väldigt fel. Kan man enkelt räkna ut en approximation? Osv.

Till sist är vi tyvärr lata, och har vi skrivit kod som verkar fungera brukar det oftast kännas som att det inte är någon idé att skriva en massa tester. Gör vi det ändå, för att vi känner att vi bör ha gjort det, finns ändå

risken att det inte görs sådär jättenoga. Vi vet ju ändå egentligen att koden fungerar, eller hur? Skriver vi testerna först blir det lättare att inte slarva. Det påminner mig om michaelcthulhu, som kommenterade en gång att det hade varit mycket enklare att skippa en viss detalj, men eftersom den finns på ritningen måste den med. Så ritningen görs **innan** han är trött och otålig, så att det inte finns (lika stort) utrymme att slarva senare.

1.2 Minus

TDD brukar ibland utvecklas till Test-Driven **Design** (istället för development), och en del kritiserar TDD för att det är en ursäkt att hoppa in och börja koda utan att tänka till. Frågan är då vad det hjälper att ha tester när de testar fel sak, och produktionskoden som "drivs" av testerna inte riktigt gör det den ska ur ett vidare perspektiv. Med andra ord, hur ska komponenterna passa in i en arkitektur när det inte finns någon arkitektur, och har man tjänat något på att behöva ändra både produktionskoden och testkoden när den arkitekturen växer fram? Det är väldigt lätt att få en falsk känsla av att utvecklingen går automatiskt. Det är trevligt när det känns som om arbetet går av sig självt, men inte när man senare inser att det gick åt fel håll. En bra liknelse är *hill climbing* som leder till ett lokalt maximum.

Det är intressant att läsa den ursprungliga artikeln som presenterade vattenfallsmodellen, och se att den egentligen var mer agil och inte alls så rigid, och fokuserad på att bestämma alla detaljer i början, som den modell som sedan kom att dominera under lång tid. Så de moderna ("agila") metoder som är populära idag, och som är en slags motpol till vattenfall, skulle kunna ses som den mer korrekta implementationen av vattenfall. På samma sätt som vattenfallsmodellen anammades i en förenklad och förvrängd version och tillämpades på ett överdrivet strikt sätt, skulle man kunna oroa sig för att agila metoder förenklas och förvrängs och tillämpas på ett överdrivet slappt sätt. Så man skulle kunna oroa sig för att man går från "allt måste planeras och specificeras i förväg och skrivas i sten" till "ingen planering behövs, och sånt där 'arkitektur'-tjafs är bara till för gamlingar som inte har förstått att vattenfall hör hemma på 70-talet."

En del kritiserar också TDD som ett sätt att kompensera för ett dåligt programmeringsparadigm som kräver att man staplar flera rader testkod på varje rad produktionskod för att ha en chans att producera pålitlig mjukvara. Medan Java enligt många mått idag är det mest populära programmeringsspråket har det ett ganska strikt men primitivt typsystem. Dynamiskt typade språk som Ruby och Python har varit populära som ett alternativ där inte kompilatorn lägger sig i, och kod som fungerar fungerar. Men när kod inte fungerar går det bara att upptäcka det när programmet körs. Så testning är den enda kvalitetskontrollen som finns. Bortsett från typsystemet är det imperativ/objekt-orienterad programmering som har varit dominant i industrin, och standard tillvägagångssättet är att all data "kapslas in" i objekt, med getters och setters. Så i stort sett all data (och därmed komponenters beteende) kan ändras när som helst, vilket gör tillståndsgrafen gigantisk och ökar antalet möjliga interaktioner som måste testas (ungefär som globala variabler). Så med andra ord, "TDD är bara en krycka som används istället för att programmera på ett vettigare sätt." Eller för att ta ett verkligt citat: "Mutable objects is the new spaghetti code" -- Rich Hickey.

Förra året var det några studenter som var kritiska till TDD i sina rapporter på projektuppgiften. En kommentar som dök upp flera gånger i olika versioner var att det var svårt med testningen. Minst en tyckte att idén att skriva testerna först kändes "helt fel", för att man då skapar begränsningar på koden istället för att låta den växa fram efter behov. Det stämmer att det kan kännas besvärligt att skriva tester för kod som ännu inte finns, eftersom man inte vet hur den ser ut eller fungerar; den finns ju inte ännu. Så jag tar upp det här "minuset" som en slags förvarning, och för att i förebyggande syfte förklara varför jag inte tror att det är ett

verkligt problem. Ett agilt arbetssätt lutar redan åt det flexibla eller rent av lite improviserade hållet till att börja med, även *med* TDD med i bilden. Om man däremot inte har en tillräckligt tydlig bild av vad man vill att koden ska göra för att kunna skriva ett test för det tolkar jag det som att man behöver tänka efter några minuter, eventuellt vid en whiteboard. Med andra ord, koden är redan begränsad av vad den ska göra - vilken funktionalitet den ska implementera. Det är en "begränsning" som man inte kan undvika. Vad är poängen med att odla kod som **inte** begränsas av den funktion den ska fylla?

2 Två olika varianter

Det finns (minst) två olika varianter av TDD.

Observera att jag har valt att kalla dem för "körbar specifikation" och "red-green-refactor" för att det var passande beskrivningar, inte för att det är vedertagna termer.

2.1 Variant 1 – körbar specifikation

Ett alternativ är att man, givet en specifikation för en komponent, skapar en i stort sett komplett uppsättning testfall som täcker in hela komponenten. Man har då en lista med testfall som man sedan skriver implementationen "mot" så att testerna lyckas ("blir gröna"). På så sätt har man en fullständig, körbar, specifikation som ger ett tydligt "tumme upp" när komponenten är klar. Är något test "rött" är det något som inte är (korrekt) implementerat.

2.1.1 Plus

Den innebär ett väldigt enkelt arbetssätt som inte kräver särskilt mycket disciplin eller att man behöver hålla reda på olika steg. En fördel är att man tvingas använda en fiktiv komponent i testerna, vilket kan avslöja brister i specifikation/arkitektur tidigt. En till fördel är att man har en väldigt konkret checklista för vad som behöver göras. Så man riskerar varken att man glömmer bort viktig funktionalitet, eller att man ödslar tid på att finslipa en komponent genom att lägga till funktionalitet som kanske egentligen inte behövs.

2.1.2 Minus

En nackdel är att man kan vänja sig vid att ha ett stort antal "röda" testfall. Ofta är en komponent beroende av andra komponenter, och testerna kan först bli gröna efter att en eller flera andra komponenter blivit färdiga. Alternativt tvingas man till att utveckla bottom-up.

En av poängerna med agil utveckling är att man har minst information tidigt i projektet och att man inte vill låsa sig för vissa detaljer från början. Så beslut som fattas så sent som möjligt fattas med mesta möjliga informationen som underlag. Det resonemanget skulle tala för att man inte vill skriva specifikationen i sten från början, och att då översätta den specifikationen till testfall, som kanske aldrig kommer till nytta, skulle bara innebära bortslösad tid.

2.2 Variant 2 – red-green-refactor

En annan variant, som har populäriserats av Kent Beck, passar konceptuellt ihop ännu bättre med det iterativa arbetssättet som utmärker agil utvecklingsmetodik. Man skriver fortfarande testerna först, men mer "parallellt" med produktionskoden. Både test- och produktionskoden skrivs i små portioner, där testkoden alltid ligger ett steg före.

Den strikta versionen kräver en hel del disciplin, och man försöker då hålla sig i en väldigt kort loop som brukar beskrivas som "red-green-refactor". Man får bara skriva produktionskod för att göra ett test grönt, och man får bara skriva så mycket testkod så att man har ett testfall som misslyckas, vilket inkluderar misslyckad kompilering (för att produktionskoden som behövs inte ännu existerar). Genom att hålla sig till de här två enkla reglerna tvingar man sig själv till att hela tiden hålla test- och produktionskod synkade.

"Red" syftar på det misslyckade testet. Testet ska bli rött först, även när det finns tillräckligt med produktionskod för att koden ska kunna kompileras och testet ska kunna köras. Sedan görs testet grönt med hjälp av en fake-implementation, ofta med ett hårdkodat värde. Till sist ersätts det hårdkodade värdet med en "riktig" implementation, efter vilket testet förhoppningsvis fortfarande är grönt.

Värdet i de två första stegen är att vi på så vis säkerställer att testet faktiskt testar det vi tror. Vi vill ju inte ha buggar i testkoden! Om testet blir grönt oavsett hur implementationen ser ut, då är det inget bra test! Om testet förblir rött även när vi fuskar och manuellt fyller i rätt svar, då är det också något som är galet!

När vi har gått igenom loopen en gång fortsätter vi på nästa loop:

- lite testkod, som troligtvis inte går att kompilera (red)
- lite produktionskod som ger fel resultat (red)
- en fake-implementation med rätt resultat (green)
- en riktig implementation (refactor)

2.2.1 Plus

Test coverage maximeras eftersom man nästan garanterar att varje rad kod som skrivs testas - hade koden inte varit nödvändig för att göra ett test grönt hade den ju inte skrivits.

Även själva kodandet blir iterativt och "agilt". Produktionskod skrivs bara som reaktion på ett rött test, som i sin tur bara skapas som reaktion på ett krav på funktionalitet. På samma sätt som man med iterativ utveckling har kort tid (några veckor) mellan fullt fungerande versioner av mjukvaran, har man med den här typen av TDD kort tid mellan fungerande kod. Man slipper alltså jonglera halvfärdiga ändringar under en längre tid och arbeta med kod som inte klarar testerna eller kanske inte ens kompilerar. Med andra ord, man skulle potentiellt kunna göra en commit med några minuters mellanrum (det skulle förmodligen inte vara meningsfullt med tanke på att den funktionalitet man håller på att lägga till inte är fullständig, men det skulle i alla fall inte vara kod som inte alls fungerar). Man skulle kunna likna det med att lösa en ekvation steg för steg istället för att hålla en stor del av manipulationen i huvudet och sedan behöva oroa sig för om man verkligen hanterade den där kombinationen av negativ exponent och rationellt uttryck rätt.

> *Jag misstänker att bergsklättring skulle kunna vara en ännu bättre liknelse, men jag känner inte till det tillräckligt bra för att ge mig på det. Om det är någon som har erfarenhet tar jag tacksamt emot ett förslag.*

2.2.2 Minus

Det krävs mycket disciplin för att systematiskt hålla sig till red-green-refactor-loopen utan att hoppa över steg eller börja göra ändringar lite här och där.

Det här är också den variant som kräver minst eftertanke och inbjuder mest till planlöst "klättrande" mot ett lokalt maximum.

I laboration 3 får du arbeta vidare med testning med hjälp av ramverket Catch!