Hemtentamen ETSA01

Thomas Strahl
D11
910612-5835
120523

U1 Test hela vägen

Enhetstester:
Utförs av utvecklarna/programmerarna. Enhetstester utförs i början av hela testperioden och de är det första faktiska testerna på klasserna och koden. De testar sin kod, utan krav på dokumentation angående om testfallet var lyckat eller inte. Varje klass ska fungera innan den skickas vidare till integrationstesterna. Enhetstester kan utföras på många olika sätt, men det utförs nästan alltid på White-box-test metoden där man har koden fram för sig dvs. motsatsen till Black-box-test där koden inte syns. Black-box-test skulle kunna beskrivas med ett svart rum som har ett litet hål där man kan sticka in lappar. Det är okänt vad som finns i rummet, men det kommer ut svarslappar ur samma hål som kan jämföras med det förväntade svaret. Black-box-test är inte alltid optimala då man gärna vill testa så mycket kod som möjligt på så få testfall som möjligt. Det blir svårt för utvecklaren att utföra sådana tester vilket är varför olika former av White-box.tester utförs istället, eller statiska tester eftersom klasserna inte alltid är exekverbara. En mycket bra, men ibland tidskrävande test-metod är Branch coverage. Tidskrävande beroende på klassens storlek och tidskomplexiteten på dess metoder. Där går utvecklaren igenom varje ”gren” i programkoden och kontrollerar att den gör vad som var tänkt. Varje gren innefattar booleans, if- och else-satser samt while-satser med eventuella if-satser i sig. Alla de här vägarna ska kontrolleras och testas. Även om branch coverage tar tid att genomföra kommer noggrann och heltäckande test av källkod och metoder att innebära sparad tid i slutet. Då systemtesterna kräver mindre tid eftersom koden är nära in på felfri. Branch coverage är en metod som kan användas under alla nivåer av tester utom acceptanstester. Anledningen till det beskriver jag under rubriken Acceptanstester.

En annan testmetod som är oerhört tidskrävande vid mer komplexa program är parvis tester. Parvis tester går ut på att varje enskild inmatning av parameter kommer generera ett enskilt testfall som måste testas för att upp nå helt ”combinatorial testing”. Det blir väldigt opraktiskt att utföra på större system då man gärna vill täcka mesta möjliga kod på få testfall.

Integrationstester:
Utförs av utvecklarna. På den här nivån tas de färdiga klasserna från enhetstesterna och sätts ihop till små undergrupper av det färdiga systemet. Det huvudsakliga målet med den här fasen är att testa om klasserna fungerar och interagerar med varandra som planerat. Med andra ord testas interface och strukturen eller designen på hela systemet vid den här tidpunkten. Varje test på den här nivån ska dokumenteras i någon form av testprotokoll för framtida användning och för att veta var felet finns så det kan korrigeras. Dokumentering av test kan vara mycket givande för organisationer pga. att återkommande fel kommer att noteras och organisationen och dess anställda lär sig av sina misstag. Det kan också vara ut av värde om någon risk som underskattades och hade höga konsekvenser på kostnad av projektet eller tidsåtgången. Då kan det undvikas i framtida projekt. Båda är nyckelaspekter inom projekt.

Genom att skapa och använda en kravtäckningsmatris kan utveckalarna komma fram till när designen är korrekt eftersom testfallen har täckt sina krav. Kravtäckningsmatriser kan också vara bra för att upptäcka när regressionstester måste utföras på grund av att ett fel eller en ändring i koden har genererat fler fel i en annan del av programmet. Det underlättar att avgöra var delen är och varför eftersom de täcks av samma krav eller testfall. När eventuella fel har korrigerats och dokumenterats i testprotokollet och utvecklarna anser att designen är korrekt och alla testfall har täckt de tänkta kraven anses den här nivån som färdig och all kod kan skickas vidare till testarna för att verifiera systemet och sätta ihop den färdiga produkten.

Systemtester:
Utförs av testarna. Systemtester kan utföras med hjälp av Black-box-tester eller White-box-tester beroende på vad testarna känner är mest lämpligt eller om det har specificerats i testplanen vilka typer av tester som ska utföras. För att se om systemet uppfyller kvalitativa krav kan så kallade stresstester utföras på systemet. Där utsätts det för hög belastning och ser hur det presterar under de förhållandena. Om ett kvalitativt krav för en webserver var att den ska klara av 100000 användare på samma gång, men inte uppnår det i stresstestet måste det korrigeras innan systemet kan levereras till kunden.

På den här nivån kan testerna ta ett tag då många oväntade fel kan uppkomma. Tidsåtgången är likvärdig till den i acceptanstesterna (någon vecka till någon månad). För att effektivisera den här nivån kan regressionstester vara lämpliga då de hittar eventuella nya fel efter någon korrigering av defekter.

Här fortlöper testerna tills man når sitt stopkriterium. Vilket är de skede då man är nöjd med systemet och det presterar som planerat. Med andra ord har alla testfall utförts och lyckats, systemet stödjer alla krav i kravspecifikationen. Vid det här stadiet är systemet redo att leveraras till kunden för att de ska kunna utföra sina acceptanstester.

Acceptanstester:
Utförs av kunden för att se om det färdiga systemet möter deras krav och förväntningar dvs. validera systemet. Sista nivån ibland testerna. Förmodligen en av de sista sakerna som sker i projektet om allt går som planerat och kunden inte ändrar sig.

Acceptanstester sker med hjälp av Black-box-tester med ”riktig” data för att se om allt fungerar och utförs enligt kundens beställning. Den här nivån brukar vara relativt tidskravande, givetvis beroende på projektets storlek kommer tidsåtgången variera kraftigt, men uppskattningsvis mellan några veckor till några månader.

Som tidigare nämnt är inte Branch coverage relevant att använda på den här nivån av den anledningen att kunden inte har tillgång till ”insidan” av systemet. Och som tidigare nämnt går branch coverage ut på att testa alla programmets grenar. En annan anledning till att det inte är relevant är för att många gånger är kunderna inte nämnvärt insatta i programmering då de har anställt ett företag för att göra det åt dem. Även om de hade haft möjligheten att testa systemet med White-box-testing hade det inte varit så givande för dem om de inte förstår koden. Kunden vill bara veta om systemet möter deras beställning.

När den här nivån är avklarad har projektet nått sitt slutskede och den färdiga produkten är redo att sättas i arbete.

U2 Hög standard på programkod

De fyra kodingsstandarderna som Jalote beskriver är: Naming Conventions, Files, Statements och Commenting and Layout.
Det är regler för hur metoder och parametrar ska namnges, organiseras, hur saker ska deklareras samt hur koden ska formateras och komenteras. Många gånger har olika företag sina egna kodningsstandarder vilket kan ha sina för och nackdelar.

a)
Fördelar med att ha kodningsstandarder inom företag är att det underlättar för alla anställda att läsa samt förstå koden. Denna fördel underlättar när koden ska underhållas eller förbättras samt under testning av program. Vilken av utvecklarna som helst kan identifiera problemet fortare och lokalisera sig i koden på ett bättre sätt om kodningsstandard tillämpas. Om organisationen följer den här standarden blir det mycket lättare att sätta sig in i äldre kod då det är känt vilket upplägg som använts.

Det finns även en baksida till det här, då vi alla kommer från olika bakgrunder och utbildningar är det omöjligt att alla har lärt sig att programmera på samma sätt eller följer samma standard inte minst skillnaden mellan programmeringsspråk. Det här medför att när nyanställda ska lära sig en ny standard kan det ta ett tag och företagets standard kommer förmodligen att ändras hos just den här individen dvs. den kommer inte tillämpas till hundra procent. Speciellt om det är stor tidsskillnad mellan utbildningarna, då IT-världen rusar fram i utveckling och företag vill ligga i framkant kör man inte med samma standard idag som för 20år sedan. En annan mycket viktig sak att tänka på är om ett företag anställer ett konsultbolag för att utföra en del av projektet så kommer konsulterna med all säkerhet att använda sig av en annan standard. Då har man ”två delar” av ett system som ska sättas ihop med olika standarder. Detta gör det mycket komplicerat för båda parter att få en djupare förståelse av koden. I den här typen av situationer kanske det ska bestämmas i förväg vilken standard som ska användas för att undvika förvirring.

b)
Som beskrivits i **a) delen** är två stora faktorer som ändrar organisationens kodningsstandard tid och ursprung. Med det menas att allt ändras med tiden och gamla program kodades inte med samma standard som används idag. I den perfekta världen borde alla högskolor och universitet lära ut samma standard till alla studenter, men så är inte fallet. Beroende på utbildning/ursprung kommer den ”inlärda” standarden att variera. Organisationer anställer inte bara personer från en skola eller en världsdel utan den som är mest lämpad för jobbet. Därför finns det många anställda med olika ursprungsstandarder vilket kommer att färga organisationens standard.

Med åldern kan personer också ha svårt att lära om och använda en ny standard. Det medför i värsta fall att vissa delar av organisationen kodar efter en delvis ”gammal” standard och andra delar efter den nya. Detta skapar irritation och förvirring, vilket innebär både oanade kostnader och längre tid innan projektet färdigställs.

c)
En metod för att lösa stora delar av problemet med hade varit att dokumentera vilken standard som används vid vilken tid i ett öppet dokument för alla berörda. Sedan när standarden uppdateras på grund av någon anledning skulle dokumentet uppdateras, men spara den gamla standarden också. Ungefär som en tidslinje. Sen skulle alla klasser och program också

dokumenteras när de skrevs första gången och ha kvar den standarden även vid uppdateringar. På så sätt kunde anställda kolla i dokumentet och se vid vilken tidpunkt något skrevs och veta vilken standard som använts.

Jag hade även haft ett team av testare som vid systemtest såg till att standarden följs. Där alla i teamet följer kodningsstandarden till punkt och pricka. Det skulle medföra att systemet inte mötte sitt stopkriterium innan standarden följts.

En annan metod för att lösa problemet är att diskutera fram en ”ny” standard in för ett projekt, dvs. bestämma inom gruppen vilken standard som ska användas och dokumentera vilken som har valts. Det betyder att programmvaruutvecklarna kommer vara mer till sin rätt och inte slösa onödig tid på att rätta sig efter en standard då den standard som används har röstats fram av dem själva.

Det når tillslut en punkt där det måste bestämmas om tidsåtgången för att säkerställa kodningsstandarden eller längden på projektet är viktigast. Båda är mycket kostsamma processer.

U3 Samma sak på olika sätt
a)
För ett program med funktionen att spela musik t.ex. Itunes och Realplayer måste man kunna spela upp musik. Ett användarfall för uppspelning av musik kan tänkas se ut på följande sätt:
Låt1 – Artist:X Titel:Y

Användarfall:

Användare utför uppspelning av en låt, normalt fall:

Aktör: Användare.

Main success scenario:

1. Användaren söker upp en låt för uppspelning och väljer den.
2. Användaren trycker på play-knappen.
3. Programmet känner igen låten och spelar upp den.

Exception scenarios:

1: Önskad låt finns inte i programmet.

- Låten visas inte bland sökresultatet.

2: Programmet spelar inte låten.

- Användaren trycker på ”play” igen.

Testfall:

Pre-condition: - Programmet är redo att användas. Låt1 finns på datorn och inlagd i programmet.

Post-condition: - Programmet är redo att användas. Låt1 spelas upp.

1. Sök upp låt1 i sökfältet.
2. Välj låt1.
3. Tryck på ”Play”.
4. Låt1 spelas upp.

Manualtext:

Uppspelning av låtar:
1. Sök på en låt eller artist i sökfältet.
2. Välj den önskade låten.
3. Tryck på Play-knappen.

b)
Användarfallet: Är tänkt att bestämma hur produkten ska fungera. Om de till exempel ska meddela användaren när ett fel har inträffat. De ska beskriva hur en uppspelning går till för att i ett senare skede testa om produkten uppfyller dessa krav.

Testfallen: Ska vara väldigt precisa och inte lämna något till slumpen eller fantasin. De ska beskriva exakt hur testet ska gå till, precis vilken data som ska matas in och vad som ska hända när testet är klart. På så sätt förstår testaren av SUT-system under testing precis när ett test har lyckats eller misslyckats och behöver inte tänka själv. De ska följas slaviskt!

Manualtext: Ska förklara för användaren vad funktioner gör och hur användaren ska ändra inställningar eller utföra saker i systemet/programmet.

Även om de här fallen beskriver samma ska så skiljer de sig åt för att de är avsedda för olika personer. Användarfallen är till för utvecklarna, testfallen är till för testarna och manualen är till för användaren av varan.

U4 Spårbarhet

a)

Genom att använda en kravtäckningsmatris för att se vilka testfall som täcker vilka krav och vice versa. Kan regressionstest underlättas eller i bästa fall bli onödiga givetvis beroende på nivån av spårbarheten. Eftersom när ett fel har upptäckts kan matrisen användas för att se vilka fler krav eller testfall som har drabbats. Sedan är det bara att leta upp dem i koden, vilket är enkelt med en vettig design och korrigera felen. Regressionstester är till för att hitta dessa typer av fel(fel som uppkommer när något har ändrats på ett annat ställe i koden), vilket kunde göras med matrisen istället. Med andra ord kommer tester och granskningsprocessen att underlättas med hjälp av en kravtäckningsmatris. Återkommande fel och fel som bygger på varandra är lättare att spåra.

I cykelgarageprojektet från kursen fanns det till viss del nytta av kravtäckningsmatrisen. Designen hade några brister och det syntes tydligt om ett krav/use case inte täcktes. Vilket kunde korrigeras i efterhand, själva täckningen av krav hade förmodligen inte hittats om inte kravtäckningsmatrisen gjorts, men den felaktiga desingen hade visat sig förr eller senare. Dock anser jag att en sådan nivå av spårbarhet kan vara onödig i ett mindre projekt som det här faktiskt var. Då matrisen faktiskt inte avslöjade nämnvärt mycket om designen utan egentligen bara ett krav som inte täcktes. Sätts sedan tidsåtgången i perspektiv till vad den gav för utdelning så var det en mindre bra disposition av tiden. Kravtäckningsmatriser är mycket tidskrävande. Det tog många timmar per person att färdigställa matrisen.

Slutligen blir frågan att ställa till projektmedlemmarna är tiden spenderad på att göra matrisen värd den tid man sparar på eventuella fel som kan uppkomma senare vid ändringar, eller tar den mer tid än den sparar?

b)

Den största nackdelen med spårbarhet i allmänhet är den tid det kräver. Speciellt om spårbarhets nivån: krav-källkod tillämpas till varje rad kod. Då framgår det precis var något händer och exakt vad som berörs av det. Det som kostar mest med spårbarhet är de förlorade arbetstimmarna och den förlorade arbetskraften om den typen av noggrannhet inte var nödvändig. Tid och kraft som kunde lagts på andra delar av projektet.

Projekt där spårbarheten är ofantligt viktig är mjukvaruprojekt inom vapenindustrin. Där får inget lämnas till slumpen och varje förändring ska kunna hittas, samt visa var den förändringen påverkar andra delar av mjukvaran. Den typen av mjukvara ska vara mycket stabil och robust dvs. helt felfri kod.

U5 Statistik testing
Granskningsprocessen bör gå till på följande sätt:
Planering → Introduktion → Individuell granskning → Granskningsmöte → Omarbete →Uppföljning

Under granskningen finns det olika personer som är med och förklarar vad de har tänkt eller om det är något som är otydligt. Exempel på personer som borde vara delaktiga under granskningen: moderator, författare, sekreterare och granskare. Det är viktigt att författaren är med och förklarar vad han/hon har tänkt när dokumentet skrevs. Annars kan det bli konflikt mellan dokumenten och missförstånd kan lätt uppkomma.

Individuell granskning kan ske på olika sätt. Ett av sätten är att granskaren antar en roll t.ex. användare eller annan lämplig roll, sedan använder granskaren denna roll för att se om han/hon kan komma på något fel på de sättet. Det går också att följa en mall som organisationen har skapat med kriterier och viktiga punkter för att underlätta granskningen.

Tidsspannet över den här typen av processer varierar mycket beroende på projektet storlek. Det finns dock generella tidsåtgångar för uppskattning av hur lång tid en process som den här skulle kunna ta.
Individuell granskning uppskattad tidsåtgång
Programkod: ca. 150 rader kod/timme
Krav/kravspecifikationen: ca. 5 sidor/timme
Testplanen: ca. 4 sidor/timme
Design dokument: ca. 4 sidor/timme

Det går givetvis inte att gå direkt från introduktionen till den individuella granskningen då det kräver lite tid att låta informationen sjunka in och tänka igenom om det finns något alvarligt fel med dokumentet som strider mot ett krav eller helt enkelt inte är genomförbart. Därför måste det finnas ett tidsspann på minst en dag mellan varje delprocess enligt mig-(erfarenhet från projektet i kursen).

Fördelar med statiska tester är att fel skriven kod eller felaktig design kan upptäckas tidigt; även innan programmet är exekverbart. Eftersom dynamiska tester bygger på att systemet är körbart kan den typen av tester inte utföras för än ett senare skede. Vilket medför en mycket kostsam process om och när ett fel upptäcks eftersom då måste i värsta fall hela systemet och designen omarbetas. Med statiska tester kan det undvikas och upptäckas mycket tidigare, vilket är de största fördelarna med statisk testning enligt mig.