Inläsning
inför 18 och 21 november

Utvecklarnas testarbete

I ett agilt arbetslag så kan rollerna vara lite diffusare än i en mer traditionell organisation, men i det här avsnittet kommer vi att fokusera på de testoperationer som normalt sköts av utvecklarna. Men i ett agilt arbetslag utgår vi ju mycket från individuell kompetens och förmåga, och naturligtvis intresse.

Testdriven utveckling

Modultester och integrationstest är grunden till det som ibland brukar kallas testdriven utveckling. I princip är det här väldigt kostnadseffektivt eftersom vi vet precis vilken funktion vi anropar, när vi hittar någonting som behöver åtgärdas, jämfört med när applikationen gått hela vägen till testarna som då hittar någonting vid en manuell test. Vi identifierar även grundläggande tankefel som i värsta fall leder till att vi måste göra om delar av koden helt och eftersom metoden hjälper oss att upptäcka dessa problem tidigt så blir det färre saker som använder funktionen som vi måste ändra i när vi ändrar den felaktiga koden. Jämför vi med fallet där vi å andra sidan inte upptäckt problemet innan vi kom till manuell test av det färdiga programmet, i det fallet måste vi nysta upp hela kedjan från felindikationen till den funktion där felet inträffade. En annan fördel med den här typen av testning är att vi måste konkretisera gränssnitten och funktionerna på ett tidigare stadium, jämfört med om vi först utvecklar ett antal funktioner, tillräckligt många för att ha byggklossar nog att sätta ihop ett program som gör någonting värt att göra; ett program som genererar nytta för kunden. Detta ger oss en klarare väg i vårt fortsatta arbete. Å andra sidan medför testdriven utveckling att vi totalt sett måste producera mer kod för att uppnå samma slutresultat. Detta gäller ju naturligtvis framförallt när vi lyckas få allt att fungera, på första försöket utan att skriva några testfall, en situation som naturligtvis sällan är så vanlig som vi skulle hoppas på.

Generellt sett är det enklare att skriva testfall för mindre komplexa funktioner, detta låter kanske som ett problem, men i många sammanhang hjälper det oss att bryta ner problemet i mindre och enklare delar, vilket i sin tur underlättar både felsökning på kort sikt och underhållet på lång sikt. En annan generell iaktagelse är att vi kan stöta på liknande problem som de vi ofta drabbas av när vi använder globala variabler, använder vi sådana så frestas vi göra misstag med variabler som ändrats på något ställe vi glömt bort och som gör att programmet gör konstiga saker på något annat ställe. Funktioner som förutsätter att andra funktioner gjort någonting speciellt eller att de anropas i en viss ordning för att fungera, kan ge oss liknande problem. Med det inte sagt att det finns situationer där där både globala variabler och funktioner med inbördes beroenden kan vara en bra lösning.

https://en.wikipedia.org/wiki/Test-driven_development

En vanlig fallgrop med testdriven utveckling är dåliga testfall, den som skriver testfallen för att hen blivit instruerad att skriva testfall får ibland fel fokus. Oavsett om ett specifikt testfall skall visa att koden i ett visst fall förväntas ge ett feltillstånd eller om den skall ge ett fungerande resultat så måste vi först förstå exakt vad funktionen förväntas göra, sedan försöker vi definiera de viktigaste gränsfallen där vi går från ”fungerar” till ”fungerar icke”. Därefter kan vi skriva vettigare testfall. I nästa steg, när vi skriver koden som våra testfall skall testa måste vi försöka hålla rätt på gränsfallen och vår ide om vad funktionen skall åstadkomma, bör vi försöka undvika att tänka på exakt vad

testfallen gör för att inte låsa oss mentalt och producera kod vars enda mål är att generera önskade testresultat.

https://en.wikipedia.org/wiki/Behavior-driven_development

BDD är en vidareutveckling av TDD där man styr upp processerna så att vi dels gör saker i rätt ordning och dels flyttar vi fokus från själva testerna och börjar med att specificera vad varje funktion skall göra (behaviour), dels är man noga med att verkligen skriva testfallen först. Med BDD flyttar vi fokus från **HUR** till **VAD** någonting skall göra, när vi specificerar problemet. Sedan måste ju våra testfall ta hand om det levererade resultatet och tänker vi efter är just det enda vi kan förvänta oss av ett testfall, att det ser vad en funktion leverar, hur funktionen gör detta är i praktiken omöjligt att se den vägen och på det viset ser vi värdet av att fokusera på just vad och inte hur. Det senare är ju fortfarande intressant när vi skriver funktionen...

https://www.youtube.com/watch?v=zYj70EsD7uI

https://www.youtube.com/watch?v=Bq_oz7nCNUA

https://www.youtube.com/watch?v=yfP_v6qCdcs

https://www.youtube.com/watch?v=SQUI9Ixb790

Ofta utförs modultester via olika hjälpmedel integrerade i våra utvecklingsmiljöer. Fördelen med den typen av hjälpmedel är att vi får hjälp med en massa administration och modultesterna finns inte längre bort än att du kan köra igång dem med musen. Men du får också en styrande struktur, vissa saker går lättare att göra och andra blir jobbigare och du har alltid en inlärningskurva innan du blir produktiv. Det här är ju vanliga avvägningar inom vårt område.

https://www.testim.io/blog/xunit-vs-nunit/

https://xunit.net/

https://nunit.org/

https://www.lambdatest.com/blog/nunit-vs-xunit-vs-mstest/

Disciplinerad utveckling

Vi har en grupp metoder som hjälper oss att genomföra kodandet på ett noggrannare och mer ordningsamt sätt, inte så att vi inte alltid måste vara noga när vi programmerar. Men vi måste rikta blicken lite högre än mot det senaste kompileringsfelet om vi skall vara säkra på att utveckla system som fungerar i det långa loppet, över hela livscykeln. Ett sätt är övervaka själva kodandet genom så kallad parprogrammering. Detta är en metod som inte bara sätter ett större fokus på exakt vad som skrivs och varför, det är även ett utmärkt redskap för kunskapsöverföring när den som skriver tvingas förklara vad det hela går ut på och varför hen väljer att göra på något specifikt sätt. Detta går i båda riktningarna, en programmerare som inte är så bekant med en viss teknik (exempelvis databasprogrammering) och kan under överinseende från någon mer erfaren få möjlighet att nästan själv försöka skapa kod på ett lite ovant område; det vanligaste är nog att den som är mer erfaren ”leder”. Egentligen är väl termen kunskapsöverföring lite snäv här, parprogrammering kan vara lika mycket en metod att bygga fantastiska program som att bygga fantastiska arbetslag. Ett sätt att lära oss att sammarbeta även med personer vi inte alltid föredrar att umgås med bara för att vi tycker att de är trevliga. Tänk på fotboll (eller andra lagsporter), varje gång det närmar sig ett fotbolls VM så kommer det ett antal reportage om dels hur fantastiska våra landslagsspelare är, några går lite djupare och då får vi alltid höra saker om hur fantatiskt duktiga de är på att kastas in i ett nytt lag

och direkt ha en fungerande strategi för att effektivt arbeta tillsammans med främlingar. Vi förväntas klara av att kunna arbeta tillsammans med våra arbetskamrater och då programmering gärna blir en ganska privat syssla så kan vi enkelt dra oss undan och undvika de personer vi inte trivs med. Parprogrammering ger oss verktyg att hitta vägar förbättra vårt sammarbete även med dessa. Många studier visar att parprogrammering inte bara ger oss bättre kod, utan bättre arbetslag och ökad tillfredställelse med arbetet.
Vi fler kunskaper än de rent tekniska, det finns många områden som är komplicerade och där det är svårt att snabbt producera fungerande kod. Som färsk programmerare kan det ibland vara svårt att se skillnad på situationer där ens egen kunskap och erfarenhet inte riktigt räcker och situationer där problemområdet helt enkelt är besvärligt, i par med en erfaren programmerare smittar inte bara de rena kunskaperna av sig på den mer ”färska” programmeraren, djupare insikter om problemområdet kommer på köpet och med en någon att luta sig emot blir inte svåra saker lika svåra.
Många utvecklingsteam blir mer en grupp specialister som producerar olika delar till något gemensamt projekt, än en arbetsgrupp som tillsammans löser ett större problem; detta kan fungera bra på många sätt, men det kan även leda till att underhållet försvåras långsiktigt, att felsökningen försvåras av att de flesta i laget har begränsade kunskaper om de andras delar av systemet. Vi kommer aldrig ifrån ett visst mått av specialisering, men långsiktigt gynnas arbetet om vi kan ha personalrotation mellan systemets olika delar utan allt för stora problem.
Praktisk erfarenhet visar att vi kanske inte kan räkna med alla kodrader skall skrivas i par. Parprogrammering är en intensiv social process som i sig kräver energi och koncentration, för både den som kodar och den som ”bara” tittar på. Det är även en väg till djupare kommunikation och förståelse mellan alla i laget, om man vågar splittra paren och regelbundet pröva nya kombinationer så kanske vi ibland får lite mer ansträngande situationer socialt sett, men vi får också chans att överbrygga de små personliga egenheter och inkompatibiliteter som i en mer ”frivillig” social situation skulle tagits som motiv att inte försöka kommunicera och arbeta såpass nära tillsammans.
Rent praktiskt är det ovanligt att en ensamprogrammerande utvecklare kodar precis hela dagen, det är epost, möten och diverse andra mindre ärenden, som måste hanteras även under utvecklingsfaser då man producerar mycket kod varje dag. Detta är ju ytterligare ett skäl att inte försöka arbeta i par precis hela arbetsdagen, varje dag.
Men väl genomförd blir det en metod att bygga bättre arbetslag, sprida kunskap och för att skriva bättre kod. De som lärt sig den här metoden brukar dessutom säga att det skapar en högre grad av trivsel på arbetsplatsen.

Den här ger oss många praktiska tips och inblickar i intressanta erfarenheter från parprogrammering i praktiken:
https://www.citerus.se/effektiv-parprogrammering-i-utvecklingsorganisationer/

Wikipedia har som vanligt en bra övergripande
https://en.wikipedia.org/wiki/Pair_programming

Ett litet föredrag från någon som arbetat med det här länge:
https://www.youtube.com/watch?v=u_eZ-ae2FY8

Eller varför inte ett lite mer teoretiskt orienterat föredrag:
https://www.youtube.com/watch?v=aItVJprLYkg

Kodanalys

En annan väg att arbeta direkt med källkoden för att förbättra den är så kallad statisk analys, att försöka utreda kodens funktion utan att köra den. Det finns verktyg för detta, som analyserar koden automatiskt, dessa verktyg har varierande nivå av funktion och användbarhet. Mer om dessa längre ned.

När parprogrammering på många sätt är en teambuildingövning, så är kodrevision något som fungerar bäst om vi redan har ett fungerande lag där vi alla vågar vara oss själva och där vi tillåts göra fel och där vi inte behöver vara bäst hela tiden. Det finns storföretag som har organiserad kodrevision med specifika kodrevisorer vars enda uppgift är att läsa andras program och kommentera och betygsätta programmen. Detta upplevs naturligtvis som en minuskarriär av de flesta programmerare och jag tror vi kan utgå från att det är mycket svårt att finna duktiga programmerare motiverade att göra ett gott arbete i en sådan roll.
Vad som däremot kan vara mycket gynnsamt för utvecklingsarbetet, är att tillsammans titta på viktiga delar av programmet vi jobbar med och i samband med detta försöka hitta förbättringar och naturligtvis även sprida kunskap om hur den delen av koden fungerar. Om alla är väl förberedda och om både ”revisorerna” och den ”skyldige” programmeraren alla kan hålla en avslappnad och icke konfliktorienterad dialog så kan det här bli en effektiv väg att hitta problem och att sprida kunskap. Som teambuildningövning är inte kodrevision lika självklar som parprogrammering, men har arbetslaget kommit upp i en rimlig mognadsnivå så kan det lyfta arbetet till en högre nivå.
Vi får inte heller glömma att kodrevision (eller för den delen ”code review”) ibland kan betyda lite andra saker än de jag försöker förklara här, många gånger låter man någon (eller några) erfarna programmerare har ansvar för det vi kallar ”pull requests” i git; när en utvecklare vill ladda upp ny kod för att ersätta tidigare kod. Då får den ansvarige programmeraren titta på koden, läsa vidhängande kommentarer och antingen godkänna och dra in koden i utvecklingsträdet eller skicka en uppmaning om någon form av förändring av koden. Det här är ju viktiga aktiviteter de med. Men kommunikationen tenderar att bli kortfattad och vi saknar ofta den typ av dialog som vi ofta får när vi använder kodrevision som en mer social aktivitet, som parprogrammering fast ofta i en lite större grupp än två.
Sedan får vi inte glömma det agila manifestet som säger att direkt kommunikation är effektivast i längden, det går säkert att revidera kod utan att mötas tillsammans, men om alla läser koden först och sedan träffas för att prata igenom det hela så kan arbetet hållas effektivt både när det kommer till själva studiedelen och när vi sedan pratar igenom koden. Programmeraren får gärna skicka med en förklaring om vad programmet gör och varför, programkoden förklarar sedan hur programmet gör vad det nu är programmet gör.

Läs gärna om Tuckmans teori om arbetsgrupper och om Belbins lagroller, om du är nyfiken på hur ett arbetslag fungerar har du ett par korta klipp här som förklarar båda:
https://www.youtube.com/watch?v=K8xdzBfX6x0
https://www.youtube.com/watch?v=yUidm6-3G54
(observera att detta inte är huvudtema för kursen, men dessa modeller hjälper oss att förstå hur lagarbete fungerar i praktiken).

Lite tips om kodrevision:

En ganska personligt färgad video om praktiska erfarenheter av kodrevision:
https://www.youtube.com/watch?v=1Ge__2Yx_XQ

Ett lite mer strukturerat föredrag på området:
https://www.youtube.com/watch?v=0t4_MfHgb_A

Analysverktyg

Det äldsta analysverktyget för automatisk kodanalys heter lint och har funnits sedan 1978. Tekniken har ju naturligtvis gått framåt sedan dess. Vi kan förvänta oss två saker av ett sådant verktyg, oavsett om det är modernt eller inte. Dels brukar de ge någon form av ”chefsinformation”, övergripande information som kanske inte direkt hjälper någon men som ändå ger en fingervisning om det analyserade programmet; vi kan då få reda på någon form av mått på programmets storlek,

komplexitet och mängd kommentarer. Programmets fysiska storlek, alltså som det ser ut i den fil som programmeraren skrivit, är ju beroende av formattering och antal kommentarer, så för att mäta den objektivt måste verktyget förstå programmeringsspråkets syntax och skala bort onödig formatering. Sedan kan verktyget analysera komplexiteten, hur många funktioner har vi? Hur djupa anropskedjor finns det? Hur många variabler? Hur många globala variabler? Hur många källkodsfiler och så vidare. Sedan försöker verktygen också hitta olika typer av problem, variabler som inte används, eller variabler som används i situationer där vi inte kan vara säker på om variabeln tilldelats ett värde innan variabeln används. Till det har vi ju alla problem med pekare och minnesanvändning i språk som exempelvis C. Ett program som hjälper till att tala om var man slarvat med sina pekare, eller vid allokering av minnet, kan vara guld värt, men rent generellt är ofta den här sortens program inte jätteanvändbara. AI kanske förändrar den saken i framtiden.

https://en.wikipedia.org/wiki/Static_program_analysis
https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

Reglerade industrier

Det finns många områden i dagens värld som är hårt styrd av lagar och regler, framför allt brukar vi tänka på läkemedel, energiproduktion, miljöstörande verksamhet, fordon, transporter, militär och telekommunikation är några exempel på industrier som har lagkrav som medför att vi blir tvungna att anpassa våra arbetsrutiner till sådana regler. Det gemensamma för dessa områden är olika typer av revision, validering och certifiering. Det hela handlar om att visa att det finns detaljerade rutiner och arbetsflöden för alla delområden i ett programmeringsprojekt. Alla dessa industrier har ju andra saker än just kod som sina främsta produkter och de har naturligtvis en massa regler inom det området som de måste uppfylla. Men nu fokuserar vi på programutvecklingen.
Många gånger väljer man fortfarande vattenfallsliknande modeller för utvecklingen, för att de dels passar bra mentalt i en industri styrd av ingenjörsarbete, i de flesta storskaliga fysiska projekt, så måste arbetet planeras i en viss sekvens för att vara meningsfullt. Man bygger grunden innan man bygger taket på ett hus, man monterar motorn innan motorhuven monteras på en bil och så vidare. Problemet med programutveckling är att vi inte har fysiska begränsningar på samma sätt som med ett konkret föremål och därför är det mindre klart vilket som är den bästa ordningen att göra olika saker på, sedan är det svårt att dela en vision om hur ett mjukvarusystem skall fungera och hur det skall användas, med alla inblandade. Därför kan vi vara helt säkra på att en utvecklingsmodell utan återkoppling från användare och andra intressenter alltid leder till besvikelser; inte alltid misslyckanden, men det kommer alltid att finnas intressenter som hade önskat att få en annan produkt. Det är där den agila utvecklingen kommer in, med sina utvecklingscykler där alla intressenter kan konfirmera att projektet går i rätt riktning innan arbetet fortsätter. Inom den här typen av industrier kan värdet av ett extremt väldokumenterat utvecklingsprojekt vara betydligt viktigare än problemet med att inte alla är nöjda med slutresultatet. Det hindrar inte många medicintekniska företag från att arbeta agilt idag. Men det är inom den här typen av verksamheter vi hittar störst konservatism beträffande arbetsmetoder.
Så vi kommer att finna företag som kräver minutiös dokumentation av mjukvarans alla delar, både före koden skrivits, men även efteråt. Arbetslagets rutiner kan regelbundet inspekteras och ni kommer då att få redogöra för hur alla steg i utvecklingskedjan går till och framför allt hur ni dokumenterar den. Ibland arbetar man även med att eliminiera onödiga funktioner, för att få en mindre kodbas att felsöka, kvalitetssäkra och underhålla långsiktigt. Trots perfektionistiska och byråkratiska rutiner kan det även finnas krav på långsiktig lagring, i vår del av världen är det generella kravet för lagring av information som kan vara kritisk för hälsan 35 år, att jämföra med bokföring där vi ganska nyligen sänkte lagringskraven från 10 till 7 år.
Går vi sedan in på storskaliga militära utvecklingsprojekt så kan budgeten vara så astronomisk och visionerna så fantasifulla att utvecklingen blir nästan gränslös i sin ambition. Ett omtalat exempel är

det utvecklingsprojekt som till slut fram till ett sameuropeiskt projekt att ta fram ett modernt jaktplan, Eurofighter. Innan det projektet startat hade man i Storbritannien jobbat med ett experimentflygplan inom det området, några år efter vårt egen JAS, flygplan som får sin stabilitet via datorer och program var något nytt och där valde man att ta fram en helt ny mikroprocessor, som validerades mot en specifikation, för att kunna bevisa att eventuella programfel inte kom från felaktig hårdvara.

https://en.wikipedia.org/wiki/VIPER_microprocessor

Inom transportsektorn har vi allt från system att hantera trafikledningssystem, flygkartor, flyghinder, start och landningsprocedurer, riskanalys, flygplatssäkerhet till arbetet kring själva fordonen. Alla dessa hanteras på liknande sätt som jag beskrivit ovan, med minutiös dokumentation i alla steg, med regelbunden revision av arbetsrutiner och naturligtvis även med revision av så väl manualer, specifikationer som själva programkoden. I både järnvägs-, luft- och rymdfartsbrancherna så har vi ofta tredubblade styrsystem, tre styrdatorer (ibland tre olika modeller) med var sitt program som var och ett är utvecklat separat av olika arbetslag. När sedan en operation skall genomföras måste minst två av systemen vara överens för att operationen skall gå vidare till fordonets hårdvara. Järnvägsbranchen är lite speciell, där väljer man ofta att slå till bromsarna när inte systemen är överens, inom flyget kan detta vara en sämre strategi.
Miljöstörande och riskabel industri, från traditionell tung kemisk industri till energiproduktion; olja, gas och kärnkraft är alla exempel på områden som drivs av omfattande säkerhetsrutiner. Precis som medicin och flyg så kan potentiella skadeståndskrav vara en viktigare faktor än direkt juridiska begränsningar. Går det fel är risken stor att de drabbade förväntas få ersättning. Speciellt i länder där en viss typ av industri är vanlig så finns ofta seriösa och detaljerade bestämmelser kring vad man får och inte får göra. Generellt sett kan sägas att vi i Europa har ett annat grundläggande synsätt än i exempelvis USA, hos oss brukar man helt enkelt förbjuda oönskade utsläpp, ja på en del områden arbetar vi förstås med utsläppsrätter, men det gäller då saker som är svåra att undvika, exempelvis koldioxid som restprodukt från förbränning. När sedan ett företag släpper ut något som de inte borde, så kontaktar de berörda myndigheter och berättar att nu råkade det rinna ut stora mängder natriumhydroxid i vårt avlopp så att kommunens reningsverk kan förbereda sig på att ta hand om detta. I USA däremot handlar arbetet normalt om att tysta ner och mörklägga så att man inte blir påkommen med att överskrida någon gräns som verksamheten fått sig pålaggd. I sådana fall kan det både bli höga böter och krav på att tillfälligt stänga verksamheten.
Oavsett dessa administrativa ramar så medför det här att vi får liknande krav som inom de områden där man har mer direkta krav på just mjukvaruutvecklingen, det dokumenteras, det testas och programmeras; allt med hårda krav på att varje steg skall vara spårbart och dokumenterat.
Eftersom vi här talar om tillverkningsindustri, program och maskiner (med mjukvaruinnehåll) för sådan industri, så finns det nästan alltid många företag som gör liknande saker på liknande sätt. Därför finns en uppsjö av nischade företag med specialapplikationer för dessa industrier. Ibland är företagen små och arbetar kanske lite mindre strikt än de borde, men blir det fel vet alla inblandade att risken är stor för att någon drabbas av orimligt höga (ur synvinkeln hos den som tvingas betala) skadeståndskrav.

https://www.team-consulting.com/insights/medical-software-development-101/