Inför vecka 46

Innan ni ger er på veckans avsnitt kanske det kan vara på sin plats att belysa en trevlig kurs jag hittade, som Cem Kaner har gjort, en pionjär inom mjukvarutest:

https://www.youtube.com/playlist?list=PLT-y_nyQivZKwIT_UTABcIGozAbdtqSTb

Automatiska tester

- modultest
- integrationstest
- automatiska GUI-tester
- lasttestning
- testverktyg (verktyg som inte direkt utför testen men som hjälper till med testmiljön)
- api-tester
- fuzzing

Test har av många ansetts vara något närmast onödigt, enda anledningen vi testar program är att vi inte lyckats få våra programmerare att producera helt igenom fungerande program. Det är naturligtvis inga med några djupare tekniska insikter, som tänker i dessa banor. Men de som bestämmer på ett företag brukar kunna skapa konstiga förutsättningar för oss, speciellt om de inte riktigt begriper hur verksamheten fungerar. Ett annat närliggande problem är att testare ofta tjänar sämre, så när vi hanterar ett område som genererar stora utgifter och utförs av personal med lite lägre status upplevs gärna som en god kandidat för rationaliseringar och besparingar. Jämför gärna med städning. I många fall finns det nog möjlighet att göra besparingar, speciellt om man väljer att acceptera en lägre kvalitetsnivå. Men oavsett företagsledningens motiv så är det lika viktigt att vi har klar bild av hur det fungerar uppåt i organisationen och här har vi ett område som riskerar att utsättas för kostnadspress oavsett rådande tekniska förutsättningar.
En annan sak vi inte får glömma bort är hur vi ser på defekter, vad är en bug? Vi vill nog alla gärna ha en fri och snäll tolkning, gör programmet något dumt är det en bugg. Men precis som vi tenderar få problem med sena förändringar, när projektet närmar sig slutleverans, så minskar också intresset att vilja hantera programdefekter. När du argumenterar för att någonting borde vara en bugg, försök ta användarens parti, skulle beteendet förstöra för slutanvändaren? Är beteendet sådant att slutanvändaren skulle bli överraskad på ett negativt sätt? Svarar du ja här, då har du hittat en bugg!

modultester

Dessa utförs ju nästan alltid av programmerarna själva. Rent ide-mässigt är det här något relativt nytt. Någon gång under 1990-talet började det talas om att flytta lite av kvalitetsansvaret från test till utveckling, detta oavsett vilken organisationsmodell man arbetade efter. Agil utveckling var relativt ovanligt på 1990-talet. Sedan är det ju stor skillnad på hur pass välgjorda och omfattande testfallen är. Försök ta inspiration av fuzzing nedan, till dina modultester, du kanske inte behöver köra jättemånga framslumpade parametrar på dina funktioner, men det är ett sätt att undvika det olyckliga fallet att du skulle ha valt extremt snälla värden på din indata; värden som i princip inte kan ”bli fel”. Tanken är i alla fall att det är betydligt lättare att felsöka ett problem som hittats vid en modultest än ett problem som upptäckts av en slutanvändare (som ju är det andra extremfallet).

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

https://www.youtube.com/watch?v=cD4ThqhnAp0

.NET:

https://learn.microsoft.com/en-us/dotnet/architecture/maui/unit-testing

https://www.youtube.com/watch?v=HYrXogLj7vg

https://www.youtube.com/watch?v=Geq60OVyBPg

integrationstest

Integrationstest är steget efter modultesterna. När alla modulerna fungerar när de testas var och en för sig så kör man dem tillsammans, ungefär som de kommer att användas i det färdiga programmet. För utvecklarna är detta en viktig distinktion, mellan modul och integrationstest. Det här är ett exempel på testuppgifter som inträffar innan testarna normalt får börja med sina aktiviteter. Ett annat problem, med termen, är att det finns många saker som kan integreras i testarens värld, applikationer kan många gånger göras med olika databashanterare, det kan finnas olika fjärrsystem som exempelvis betaltjänster, logistiktjänster och annat som skall fungera ihop med den egna applikationen, dessa tester kan även de kallas integrationstest. För att inte tala om de system vi har system där våra program bara är en del i ett mycket större sammanhang och i en sådan situation integrationstestar vi när program och hårdvara körs tillsammans. Fast en del använder termen systemtest för att beskriva det steget. Men låt oss här använda integrationstest om steget efter modultesterna; den här kursen är ju trots allt en del av en programmeringsutbildning. Men vi måste vara medvetna om att ordet kan tolkas på flera sätt och att det långt ifrån är självklart vad en främmande person menar med termen.

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

https://www.youtube.com/watch?v=QYCaaNz8emY

automatiska GUI-tester

På den senaste lektionen fick jag frågan om det inte används AI för att validera att ett automatiskt GUI test lyckats. Att jag inte hört talats om några sådana försök betyder ju verkligen inte att ingen försöker göra något sådant. Men som sagt, vi har en kulturell klyfta där många som går till test inte är tillräckligt duktiga programmerare, för att lyckas där karriärmässigt som sådana, medans utvecklarna då ofta ser testandet som kontorsarbetarens alternativ till dikesgrävning (eller annan lämpligt arbetsam och tung uppgift, utan kreativa inslag) vilket gör att vi allt för ofta saknar de skickliga programmerare som ofta skulle kunna utveckla testdelarna av utvecklingsprojekten.
En sak vi inte har talat om är de kommersiella verktyg som finns tillgängliga för att automatisera GUI-tester ofta är konstigt utformade, dels har de en uppsjö av funktioner som ser koola ut vid en demonstration; allt från att kunna läsa och skriva filer i vanliga format från Microsoft Office till funktioner gjorda för att hjälpa någon som inte förstår programmering att programmera. Det är ofta de senare funktionerna som gör verktygen mindre användbara. Låt oss tänka oss ett exempel, ditt verktyg har på något sätt lyckats identifiera den knapp som testaren vill att programmet skall klicka på som ett viktigt moment i ett automatiserat testfall. När så en ny version av programmet som skall testas dyker upp, så kanske inte längre vårt testverktyg förstår vilken knapp den skall klicka på. Om det då är otydligt hur testverktyget gjorde med den gamla versionen av programmet, så kan det vara jättebesvärligt att få den att hitta rätt igen med den nya versionen. (Observera, det jag beskriver här är ett extremt vanligt problem, inte bara något som ”skulle kunna hända”). Om sedan själva programmeringen av verktyget i stort är inlindad och luddig, med mycket inspelande av musrörelser och liknande, i stället för att skriva kommandon i filer, då har vi ett verktyg som programmeringsokunniga testare (som helst inte vill lära sig att programmera) inte vill använda.

Inspelade musrörelser brukar föresten gå fel redan när man byter skärmupplösning (eller att någon inställning eller funktion för HiDPI gjort att skärmbilden skalats om) eller när en uppdatering av operativsystemet gjort att default-positionen för ett nyöppnat fönster ändrat sig; ja eller för den delen om utvecklarna modifierat någon parameter för hur ett nytt fönster skapas. Vi kanske inte får missa att det är svårt att linda in programmeringshantverket på ett sådant sätt att en person som inte gillar att programmera ändå kan stå ut med det, så det är en fundamental tankevurpa att göra ett programmeringsverktyg för de som inte vill programmera. De saker som görs för att göra upplevelsen mindre programmeringslik, blir ofta saker som gör arbetet mer obegripligt och det är ju just den aspekten som de som inte gillar programmering brukar hänga upp sig på; varför gör inte mitt program som jag vill, varför är det så många små steg som måste göras när nästan alla tenderar att bli fel? En programmerare som tvingas använda ett sådant verktyg blir också frustrerad, men för att dels hårda saker, som att identifiera knappar, inmatningsfält och andra grafiska element döljs i lager på lager av konstigheter, samtidigt som den programmeringsdelen oftast ”barnanpassats” med musklick, drag och släpp och en massa ritprogramsliknande funktioner i stället för mödosam, men ändå begriplig hantering av textfiler med kommandon uppradade i tur och ordning. Om någon av er provat Scratch (https://scratch.mit.edu/) så vet ni ungefär vilka typer av förenklingar man jobbat med för att erbjuda ”programmering utan programmering”. Men jämfört med scratch så är QA-verktygen inte bara mycket tråkigare, utan även klumpigare och mindre flexibla. Scratch har varit en revolution för programmeringsundervisning bland barn. Mycket handlar alltså om att göra en produkt som kan säljas in till en lagom högt uppsatt chef, en som aldrig kommer i närheten av mjukvarutestandet mer än som en irriterande kostsam post i bokföringen och som alltid fördröjer utvecklingsprojekten, i situationer då alla trodde allt skulle vara klart.
Vidare kanske vi bör tänka på hur den rådande trenden med grafiska gränssnitt som sällan följer någon standard och med element som dynamiskt dyker upp, flyttas och försvinner, på ett sätt som ofta är obegripligt för användaren och som kan vara ännu mer utmanande för utvecklaren av automatiska tester. Att den här sortens tänkande ibland formaliserats gör det inte alltid lättare, Googles Material Design, med exempelvis knappar utan ram, har givit utvecklaren av automatiska tester två problem, dels syns inte gränsen för ett klickbart objekt om designfilosofin säger att det inte skall finnas några synliga gränser, dels inspirerar de programmeraren att pröva nya vägar i programmerandet. De grundläggande grafiska element som en windowsprogrammerare använde 1985 var inte förändrade på något grundläggande som skulle kunna göra någon förvirrad om en sådan utvecklare fick åka tidsmaskin till 2010 (ja eller en resa i andra riktningen med för den delen). Titta på Microsoft Teams där det hela tiden dyker upp knappar som lika fort försvinner igen, i somliga fall till och med när musen närmar sig. Hur automatiserar någon tester för ett sådant program? Är sedan knapparna inte skapade på standardmässigt sätt, commctrl.dll var länge ”motorn” som nästan alla program använde för att skapa GUI-elementen, då blir det svårt att få tag i det användaren ser på skärmen från testprogrammet. Att vi sedan riskerar att de moderna icke standardiserade elementen ändras oftare på ett tekniskt sätt, så att program som tidigare hittat en knapp inte längre kan identifiera denna, vilket naturligtvis är ett dagligt problem för testfallsutvecklaren och det är något som bara blivit svårare med moderiktiga användargränssnitt.

Den andra delen av ett automatiserat GUI-test är att avgöra om testfallet lyckades eller inte, om det var svårt att navigera grafiska element på skärmen för att genomföra fallet, då har vi ofta kommit till en ännu högre och mer arbetskrävande nivå, när vi utifrån en skärmbild skall avgöra om det gick bra eller inte. Ibland räcker det kanske med att se att vi inte fick upp några varnings-dialoger, men ofta har gränssnittsdesignern tänkt sig att användaren skall få återkoppling via subtila färgförändringar eller kanske genom byte av typsnitt, det kan även finnas små markeringar, som prickar i någon ikon. Oavsett vilket är dessa detaljer ofta svåra att hitta och extremt enkla för utvecklaren att trolla om så att testskriptet misstolkar utfallet. Med bättre verktyg kan vi göra kombinerade tester där vi genererar musklick och annan ”användarinput” samtidigt som vi kanske läser en databas eller tjuvlyssnar på API-anrop, eller så har vi kanske loggmeddelanden att utgå från. Oavsett vilket så är dessa ”sidokanaler” ofta betydligt enklare att analysera än att ”titta” på en

grafisk dialog, med ett testprogram. Det är på det här området som misslyckanden med automatiska tester brukar vara som tydligast, det är här vi hittar de verkliga tidstjuvarna.
Tyvärr är ju analysen av sidokanaler någonting som ofta kräver lite djupare insikter om programmering än vad vi förväntar oss av de personer som rekryteras som testare. Det kan även kräva mer sammarbete med utvecklarna än vad det finns tid för i projektet, det är länge sedan vi förväntade oss av ett program att det skulle gå att slå på loggmeddelanden, så att programflödet enkelt skulle kunna följas.
Jag säger inte att det är omöjligt att köpa ett verktyg för GUI-testning som fungerar och som ger er verksamhet ett lyft. Men risken är stor att ni slänger pengarna i sjön på något som kräver mycket arbete innan ni verkligen bevisat att det aldrig kommer att ge så mycket positiva resultat i ert utvecklingsarbete. Det är här fri programvara kommer in, ta applikationer som Linux, Apache, Nginx, Postfix, dnsmasq och många många fler. Det som ofta sägs är att de är ”fria” och att det står ”0” om någon lyckats hitta en prislapp. Men det vi söker är inte verktyg som är gratis, utan som fungerar och som hjälper oss i vårt dagliga arbete. På det här området är det dock så att det är här vi ofta hittar dem. I och med att utvecklingen inte drivs av ledningens krav att producera säljbara produkter utan av teknikers krav på att göra användbara sådana. Populära open source program inom testområdet (och inom andra områden) utvecklas för att någon ser funktioner som behövs för att göra programmen mer användbara. Ja kanske även ibland för att det helt enkelt är roligt att programmera av fri vilja och av glädje, inte för att någon chef kräver olika funktioner.
En sak att se upp med när det kommer till fria program, är projekt som slutat utvecklas, livaktig utveckling och en aktiv användargemenskap är viktiga kvalitetsstämplar på framgångsrika och projekt och en viktig nyckel till att snabbt hitta användbara verktyg.

Även om jag varnar för motsatsen ovan, så finns det säkert användbara ”köpeprogram”, även inom det här området. Är du ute efter en specifik funktion, som du inte hittar bland de fria alternativen, så kan ju även detta vara en viktig indikation på att du hittat ett område där det kan löna sig att betala.

Här följer några populära exempel på fria program för gui-testning:

https://www.selenium.dev/
https://github.com/SeleniumHQ/selenium

http://appium.io/

https://testsigma.com/
https://github.com/testsigmahq/testsigma

http://cerberus-testing.org/
https://alternativeto.net/software/cerberus-testing/about/

Här har ni några guldkorn från ’tuben:

https://www.youtube.com/watch?v=QtZ4yV49RtA
https://www.youtube.com/watch?v=c91XBjfODks

https://www.youtube.com/watch?v=j7VZsCCnptM
https://www.youtube.com/watch?v=IQ_KpBC8fwo

lasttestning

Samtidigt som lasttest är viktigt så måste vi komma ihåg att resultatet är extremt beroende av både hur vi testar och av hur vår miljö ser ut. Hos slutanvändarna kommer det att se annorlunda ut, men vi kan vara övertygade att det som går långsamt hos oss i vårt lab, knappast kommer att vinna poäng på snabbhet ute hos kund. Lasttest i labmiljö ger indikationer hur väl ni lyckats prestandaopterimera systemet under test, speciellt i jämförelse med tidigare versioner. Vill ni däremot ta reda på hur väl en webbtjänst står pall för trycket från användarna så talar vi om avancerade tester med stora krav på både nät- och serverinfrastruktur. Är din tjänst ”stor”, alltså en webbtjänst tänkt att kunna klara många samtidiga användare; jämför med Skatteverkets hemsida, de populärare shoppingsajterna, informationstjänster som Hitta och Eniro, för att nämna några saker som är ”stora”; sajter som är tänkta att klara tiotusentals anslutningar per sekund. Går vi sedan en par tiopotenser uppåt så räknar Google, Facebook, Twitter, Amazon, Ali-express och liknande tjänster med att klara anstormningar på över en miljon anslutningar per sekund; att bygga upp en testmiljö för sådana saker ligger bortom normala förmågor även om säkert specialister som Apica Systems (https://www.apica.io/) kan rå på den sortens tjänster med. Att sedan dessa specialister även kan genomföra storskaliga lasttester i produktionsmiljö utan att störa för slutanvändare gör ju sådana tjänster mycket tilltalande.

Inte oväntat finns ett stort utbud av verktyg för lasttest av webbtjänster, skall du testa något mer specialiserat protokoll, får du räkna med egen programmering från grunden; en strategi som nog ofta fungerar väl även med webb-applikationer. Eftersom prestandatester är en såpass specialiserad aktivitet är det ändå troligt att detta är ett område där det oftare lönar sig att själv göra sina verktyg. Kanske inte alltid från grunden, till populära programmeringsspråk så brukar det finnas en uppsjö av färdig kod både för nätprogrammering i allmänhet, men naturligtvis även för att automatisera webbtjänster. Sedan krävs avancerade och genomtänkta anläggningar, både beträffande nät- och serverkonfiguration.

Ett specialfall av lasttestverktyg som kan vara mycket användbara är inspelningsverktyg, som sniffar trafiken när någon ansluter manuellt och sedan kan sådana sessioner editeras och återanvändas så att det inte krävs så mycket specialprogrammering. Detta kanske låter mer primitivt än användbart. Men om vi tänker efter inser vi att även en ”misslyckad” session, en session där exempelvis inloggningen eller något annat viktigt steg misslyckas, så kan den ändå ge intressanta fingervisningar om ditt systems prestandanivå. Speciellt om du ser hur svarstiderna varierar över olika lastnivåer.

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

https://www.youtube.com/watch?v=-dI_gKRePRo

https://www.youtube.com/watch?v=KECr2BujqtM

testverktyg

En ofta förbisedd del av vår arsenal för automatisering av tester är program som inte direkt används för att genomföra testerna, men som utför tidsödande rutinuppgifter som bör genomföras innan vi får en miljö där det är relevant att testa. Några exempel är automatisk installation, automatisk konfiguration av databaser, automatisk generering av testdata, automatisk konfiguration och installation av testdatorerna. Automatisk konfiguration av nätverksutrustning. Listan kan säkert göras längre, men det viktigaste för er att ta med er från den här kursen är att det här är ett område

som ofta ger stor arbets- och tidsvinst med liten insats. För det mesta är det här saker som ingen talar om och som vi allt för ofta glömmer bort, du har chans att bli arbetslagets hjälte genom att slå ihop några små kodrader, som sedan spar arbetstid varje vecka, för de som testar.
Det här området bör beaktas redan innan vi egentligen sätter igång med utveckling av själva produkten. En sak som gör den här sortens verktyg så arbetsbesparande är att de ofta kan användas hela projektet igenom utan större förändringar, när en flyttad knapp i ett formulär kan vara något som orsakar mycket arbete för den som vill underhålla ett automatiserat testfall, så kan genererandet av testdata, automatiska installationer eller liknande ofta ske på precis samma sätt, oavsett hur kreativ någon varit med användargränssnittet den senaste veckan.
En annan orsak till varför det inte talas så mycket om just det här ämnet kan vara att dessa verktyg varierar mycket i funktion; de som testar spel kanske har liten nytta av ett verktyg som genererar high-score listor, medans de som testar bokföringsprogram ofta har stor nytta av hjälp att generera allt från nya artiklar, lagerposter till fakturor och betalningsordrar.
Till den här gruppen saker kanske vi även bör räkna diverse saker som vi kan låna från systemtekniker, framför allt verktyg för att automatisera installationer, för virtuella maskiner och verktyg för konfigurationshantering på både servrar och skrivbordsmaskiner (rent tekniskt sett är detta samma sak, men frågar du teknikerna som jobbar med det, så använder de oftast helt olika verktyg beroende på om de kör skrivbordsdatorer, servrar eller virtuella maskiner).
Här följer några exempel utan några rekomendationer eller värderingar, men alla skulle kunna vara användbara i storskaliga testkonfigurationer:

https://puppet.com/
https://kubernetes.io/
https://en.wikipedia.org/wiki/Salt_(software)
https://learn.microsoft.com/en-us/windows/configuration/provisioning-packages/provisioning-how-it-works
https://learn.microsoft.com/en-us/windows/win32/msi/standard-installer-command-line-options

API-tester

Precis som testverktygen ovan är det här något som ofta helt hoppas över i planeringen. API-testning skiljer sig från modultesterna genom att vi här använder samma kommunikationsprotokoll som klientprogrammet använder, så kan vi testa de funktioner som vi annars skulle testat via antingen manuella eller automatiska GUI-tester. Men vi slipper alla de problem och fallgropar som automatiserad GUI-testning medför. Det här kräver naturligtvis lite bättre programmerings-kunskaper än vad vi kan förvänta oss från den genomsnittlige testaren. Å andra sidan är detta ett mer långsiktigt arbete eftersom dessa funktioner ofta ändras mindre än användargränssnittet som ju hela tiden står under ”kritik” från användare och deras företrädare. Det här är alltså ett område där vi enkelt kan öka volymen av tester med en relativt liten arbetsinsats. Det här kan vara en intressant nivå att testa på när vi närmar oss fuzzing (se nedan). Närmar oss? Det jag menar är vi kanske inte behöver köra fullskaligt med alla tänkbara parametrar, som vid fuzing, för att ändå öka den totala testvolymen till extrema nivåer utan att testa precis alla tillåtna parametrar som existerar.
En annan fördel med de här testerna, jämfört med automatiserade användargränssnittstester är att vi slipper leta efter olika subtila förändringar i grafiska element i skärmdumpar, i stället kan vi titta på svarsvärden från API:et vi testar och utifrån detta se om det är ”flipp” eller ”flopp”.
Det här är precis som testverktygen ovan ett område som ofta glöms bort fastän det kan vara en mycket lönsam väg att testa snabbt och mycket.
Tyvärr är det här lite nytt så jag har inte hittat några vettiga länkar.
Kom ihåg att när du som utvecklare gjort klart modultesterna (unit testing) så går man ofta till nästa steg där vi testar flera funktionsanrop tillsammans, för att se att de går att använda som det är tänkt. Detta kallas ibland för ”integrationstest” vilket tyvärr är en term som även används till många andra saker, därför skall du inte bli förvånad om du hittar någon som använder termen ”API-test” till

något annat än det som vi syftar till i den här texten; att använda API:er för att enkelt och effektivt kunna testa saker som är svåra att genomföra via automatiserad GUI-tester, eller automatiserad GUI-testfall som är känsliga och som tenderar att gå sönder vid små förändringar i det testade programmet eller i testmiljön. Den typen av API-tester ofta det enda fruktbara alternativet till flera saker när storskalig automattestning skall genomföras, men tyvärr saknas ofta kompetens bland utvecklarna av automatisk testfall för att göra de nödvändiga API-anropen och dessutom har de fått ”barnanpassade” och fördummande testverktyg som inte heller klarar av sådana saker.

fuzzing

Det här är ett relativt nytt område, betydligt vanligare än både ”verktygen” och API-testerna, men ändå ovanligt. Storskaliga API-tester närmar sig fuzzing rent konceptuellt. Men vad är det som menas med fuzzing? Det är när testaren slumpar fram testdata och försöker generera alla tänkbara input till program och API:er för att hitta värden som genererar fel. Det är naturligtvis svårt att avgöra vad som är ett fel när ett program körs med väldigt ”ovanliga” indata, men programkrascher brukar definitivt räknas som fel.
Flera stora företag, däribland Google, har egen säkerhetsforskning som även riktar sig mot andras program, ibland ser man i sitt nyhetsflöde om hur någon misstänkt att Google försöker misskreditera exempelvis Microsoft eller Apple genom att publicera säkerhetsproblem som de hittat i dessa företags program och tjänster. Det är fullt möjligt att det finns en sådan agenda, men faktum kvarstår att de regelbundet publicerar information om säkerhetsproblem de funnit på liknande sätt när de undersökt fria program där det rimligen inte kan finnas någon marknadsstrategi från Googles sida att skrämma oss från att använda fria program. Ett viktigt verktyg som Googles säkerhetsforskare brukar tala om i dessa sammanhang är fuzzing.
Skall vi utgå från det som skrivs på nätet så är fuzzing betydligt vanligare bland de säkerhetsforskare som försöker hitta säkerhetshål i olika program, jämfört med användningen inom QA hos programutvecklare. Min gissning är att det här kan vara en mycket fruktsam metod att se till att inte våra modultester blir snälltester som bara testar ofarliga saker. Vi kanske inte behöver ta efter säkerhetsforskarna och köra extremt många anrop, men om inte annat kan du se det såhär, med bra pseudoslumptal, så behöver du själv inte anstränga dig med att ”uppfinna” indata och du kan därför kosta på dig mer omfattande tester utan allt för omfattande testplanering.

https://en.wikipedia.org/wiki/Fuzzing
https://owasp.org/www-community/Fuzzing

https://www.youtube.com/watch?v=7KWPiRq3ZYI

Det här är ett roligt klipp, här hittar upphovsmännen ett stort säkerhetshål när de gör videon som dels gör att de måste vänta med att släppa sitt klipp tills problemet är åtgärdat och dels att de måste presentera sin information på ett sådant sätt att det inte hjälper dem som vill använda deras upptäckt att ”busa” med på system som ännu inte blivit uppdaterade:

https://www.youtube.com/watch?v=O3hb6HV1ZQo

Väl mött nästa vecka, ställ gärna frågor på Teams om du undrar över något, jag har inte alltid näsan i datorn men jag brukar kolla av teams dagligen.

Lycka till,
Roland