Automatiserad testning av användargränssnitt i SharePoint

Automated UI Testing in SharePoint

Daniel Borg
Anders Elfström

KTH Information and
Communication Technology

Examensarbete inom information- och programvarusystem, grundnivå
Högskoleingenjör

Degree Project in Information and Software Systems
First Level

Stockholm, Sweden 2014
Kurs II121X, 15hp

TRITA-ICT-EX-2014:57

Automatiserad testning av användargränssnitt i Sharepoint

Examensarbete inom Informations- och Programvaruteknik, grundnivå
Högskoleingenjör
Kungliga Tekniska Högskolan
Stockholm, 2014

Sammanfattning

Företag arbetar ofta efter hårda krav från kunder där lösningar måste levereras på ett tidseffektivt sätt och samtidigt hålla en hög kvalitet. Detta i form av felfria och robusta system vilket delvis kan åstadkommas med hjälp av testning. Kraven på snabb leverans och hög kvalitet är däremot motpoler till varandra; snabb leverans genomförs ofta på bekostnad av kvalitet och tvärtom. Agila arbetsmetoder med tidiga och frekventa leveranser har ändrat på detta, men kräver samtidigt en ständig kvalitetsförsäkring under arbetets löptid. Under utveckling av mjukvara enligt dessa metoder förekommer därför en kontinuerlig kvalitetssäkringsprocess för att säkerställa att produkten dels uppfyller vad kravspecifikationen avser samt att levererad produkt håller en hög tekniskt kvalitet i form av buggfri och robust kod som innehar stor pålitlighet för framtiden. Då manuell testning är en kostnad- och resurskrävande metod har automatiserad testning blivit ett aktuellt alternativ för ökad effektivitet och hållbar utveckling.

Målet med det här arbetet har varit att för Precio Systemutveckling AB utreda möjligheterna för en implementation av automatiserad testning av användargränssnitt, i en hos företaget redan existerande och etablerad utvecklingsprocess. Arbetet har genomförts med en inledande förstudie om testning med fokus på varför det är en extra viktig faktor i dagsläget. Detta följs av ett avsnitt där existerande teori och teknik för testning i generell mening avhandlats, följt av en närmare studie på hur automatiserad testning är rekommenderad att utföras ur ett perspektiv från utveckling av produkter inom Microsoft-teknologi.

Arbetet avslutas med att visa en implementation av automatiserad testning i skarp miljö hos Precio.

Nyckelord: Automatiserad testning, Testning, .Net, C#, SharePoint, Continuous Integration.

Abstract

Software companies work under a strict pressure from customers where solutions must be delivered in a timely manner as well as providing high quality and value. The products should be robust and without errors, which partially can be accomplished by testing during the development process. The requirement for an early delivery and a high quality does not always come hand in hand, but with the increased use of agile software development methods, this can be achieved. During agile development of software, there is a continuous process to ensure the quality of what is being developed, both to make sure that all the functional requirements are fulfilled, but also that the code behind is robust and dependable for the future. Since manual testing can be both time and resource consuming, automated testing has become a modern alternative to increase productivity and to maintain a sustainable development process.

The goal of this thesis work has been to investigate the possibilities of implementing a solution for automated UI testing in an already existing development process at the company Precio Systemutveckling AB. The work has been conducted in three steps, starting with a literature study about testing in general, followed by an extensive research into suitable tools and technology for testing that exists today. After this, a deeper look was made at what the recommended solutions for implementing automated testing in a Microsoft-oriented enviroment were.

The work was concluded with an actual implementation of automated testing on premise at Precio.

Keywords: Exempel: Automated testing, Testing, .Net, C#, SharePoint, Continuous Integration, GUI testing, UI.

Förord

Det här examensarbetet har utförts hos Precio Systemutveckling AB i Stockholm, och vi vill rikta ett stort tack till dem för vänligheten och möjligheten till detta. Särskilda tack riktas till våra handledare, Fredrik Gullberg och Johan Schedin Jigland, för den tid och oersättliga hjälp de avsatt till vårt arbete. Utan er hade inte detta varit möjligt!

Vi vill även tacka vår examinator Anders Sjögren på KTH för de värdefulla tips och den vägledning vi fått.

Innehållsförteckning

Terminologi

Tekniska begrepp

Agent – Nämns i sambandet ”bygg-agent”, där den svenska översättningen blir *ombud*. Har till uppgift att utföra arbeten den tilldelats.

Agil utveckling – Agil utveckling är en iterativ och inkrementell utvecklingsform med många kontinuerliga leverabler, med avsikt att genomföra arbetet i ett nära samarbete med kund eller slutanvändare. En agil utvecklingsform öppnar upp för löpande ändringar i kravspecifikationer och tillåter förändring i större utsträckning än andra etablerade utvecklingsmetoder.

Backlog – Ett redskap i form av en loggbok med översikt över uppgifter tillhörandes en aktiv iteration under utvecklingen.

Checka in kod – Att synkronisera kod till en för alla utvecklare gemensam server.

Hårdkodad – Programkod där statiska värden används som inte kan ändras under exekvering.

Testning – Verifierar att programkod fungerar och gör vad den är ämnad att göra. Om programkod fungerar inkorrekt ska tester upptäcka detta.

Regressionstestning – En testningsmetod med fokus på att hitta fel och buggar i äldre kod efter att en förändring av systemet har genomförts. Testar funktionalitet som tidigare fungerade som tänkt.

Smoke testing – En uppsättning tester där man med minimal ansträngning försöker få det testade systemet att fungera, för att utesluta eventuella grundläggande problem. Integreras ofta med kompilering av system.

Acceptanstestning – En typ av test som har till syfte att avgöra huruvida en slutprodukt uppfyller de konkreta mål och krav som kravspecifikationen avser.

Sprint – En arbetsperiod som sträcker sig över en fixerad tid i projekt och innehåller olika uppgifter som ska utföras.

Syntax – Regler för hur programkod ska skrivas inom ett särskilt programmeringsspråk.

.NET – Ett ramverk och en del av Windows som möjliggör utveckling och exekvering av särskild programkod.

Ramverk – En arkitektur och paketering av olika tekniker som hör samman.

C# – Programmeringsspråk inom .NET-ramverket.

Interface – En definition gemensam för flera komponenter som specificerar särskild funktionalitet som måste implementeras.

Assembly – En modul eller paketerad del av kompilerad kod.

DLL – En assembly inom .NET och ett delat programbibliotek.

GUI – Graphical User Interface. Det grafiska användargränssnittet som exponeras för slutanvändaren.

CI – Continuous Integration. Beskrivs utförligt i avsnitt 2.4.2.

HTML – Hyper Text Markup Language. En standard för hur webbsidor skrivs.

Språkliga förklaringar

Texten innehåller i särskilda fall ord som till synes hör till det engelska språket, exempelvis ”deployment” eller ”release”. Dessa är etablerade ord även inom den svenska utvecklingsbranschen och skrivs därför ut som de är.

I texten förekommer ordet ”bygge” i särskilda sammanhang. Detta syftar på kompilering och sammanställning av kod, och används ofta inom mjukvaruutveckling.

1 Introduktion

1.1 Bakgrund och problemmotivering

Precio Systemutveckling AB är ett företag som arbetar med utveckling av nya system baserade på teknologi från Microsoft, med fokus på Microsoft SharePoint. Företaget är beläget i Stockholm, och vill med det här arbetsuppdraget utreda möjligheterna för automatiserad testning på närmare håll.

Inom den moderna mjukvarubranschen är system idag benägna att kontinuerligt förändras som en effekt av agil utveckling, vilket direkt påverkar användargränssnitt kopplade till dessa. Detta gör det viktigt att under hela utvecklingsprocessens löptid säkerställa att användargränssnittet alltid fungerar och uppför sig på det sätt som det är tänkt att göra.

Genom att tillämpa testning av användargränssnitt tidigt i utvecklingsfasen kan man i senare skeden snabbt upptäcka huruvida ny implementerad funktionalitet har en påverkan på existerande funktioner och dess tillhörande gränssnitt. Testning innehar även en viktig roll för att säkerställa kvaliteten på en utvecklad mjukvara och är oftast en del av kvalitetssäkringen i ett projekt.

Testning av användargränssnitt går ut på att genomföra uppgifter eller scenarion mot användargränssnittet för att sedan jämföra resultatet med förväntad utdata. Detta går att genomföra manuellt, då en särskild testare utför uppgifterna mot systemet för att sedan undersöka resultatet. Manuell testning är dock en resurskrävande och kostsam process som numera går att ersätta med automation. Ett automatiserat test går att genomföra upprepade gånger utan någon yttre påverkan, vilket gör det mer kostnadseffektivt och tillförlitligt än manuell testning.

I denna rapport läggs fokus på kvalitetssäkring i form av testning av användargränssnitt, och undersöker hur och i vilken utsträckning testningen kan automatiseras med avseende på generella projekt på den valda plattformen Microsoft SharePoint. Eftersom detta arbete utreder möjligheterna för en implementation av automatiserad testning tillämpbar på projekt genomförda enligt agila utvecklingsmetoder som använder en kontinuerlig integrationsmodell, inleds även rapporten med en kortare introduktion till agil utveckling och arbetsprocessen ”Continuous Integration”, ibland förkortat CI.

1.2 Övergripande syfte

Arbetet utreder om det är möjligt att implementera en etablerad lösning för automatiserad testning av gränssnitt, och i sådana fall hur detta underlättar och påverkar olika moment som återfinns i en existerande agil arbetsprocess. Detta undersöks genom att konstruera en metod som sedan testas mot ett under arbetets gång uppsatt SharePoint-projekt.

1.3 Avgränsningar

Då området som utreds är av ett stort omfång är det som rapporten avhandlar begränsat till testning av användargränssnitt, GUI (Graphical User Interface), med fokus på Microsoft SharePoint. Övrig programvara förutsätts även den vara implementerad med teknik från Microsoft, om inte annat anges. Rapporten innehåller även en generell beskrivning av testning för att ge läsaren en förståelse för olika viktiga koncept.

1.4 Konkreta och verifierbara mål

Precio, företaget där arbetet är genomfört, bygger många komplexa webbaserade system, där flertalet av dessa har ett lika komplext användargränssnitt. Att testa hela användargränssnittet manuellt är oftast inte ett kostnadseffektivt alternativ. Genom att introducera automatiserad testning av användargränssnitt, framförallt i samband med processtillämpningen CI, strävar Precio mot att höja kvaliteten samt reducera kostnader på samma gång.

Existerande teknik och arkitektur hos Precio, bygger på en agil arbetsmetodik och den ovan nämnda processen som kallas ”Continuous Integration”. De konkreta mål för arbetet som existerar är att undersöka hur automatiserad testning kan integreras som steg i deras utvecklingsprocess där vi i slutet av arbetet har en fungerande prototyp

som går att använda och tillämpa i olika godtyckliga projekt. Prototypen är tänkt att vara en testningsmodul med fokus på GUI-testning.

För att underlätta arbetet har ett antal delmål etablerats för att bättre specificera vad som behöver undersökas och konstrueras. Dessa presenteras i följande avsnitt.

1.4.1 Delmål

Utifrån de övergripande mål som ovan omnämnts har ett antal mindre delmål tagits fram för att vi enklare ska kunna strukturera upp arbetet i projektet. Delmålen ligger till grund för vad som ska undersökas och refereras till i rapporten under motsvarande moment.

- Sammanställa en kravspecifikation över önskad prototyp
- Definiera vad GUI-testning innebär
- Undersöka och hitta ett lämpligt verktyg för testning av GUI
- Hitta en lämplig arbetsmetod för utförande av arbetet
- Hitta en lämplig lösning för automatiserad testning av GUI utifrån kravspecifikationen
- Hitta ett lämpligt sätt att implementera lösningen i Continuous Integration

2 Bakgrundsmaterial

Det som den initiella förstudien resulterat i presenteras här nedan. Denna ligger sedan till grund för val av tekniker och konstruktionsutförandet.

2.1 SharePoint

För att bättre förstå det arbete och tekniken som Precio använder i sin utveckling beskrivs här kortfattat vad Microsoft SharePoint är.

2.1.1 Vad är SharePoint och vad används det till?

SharePoint är en plattform utvecklad av Microsoft där man samlat olika verktyg som tillsammans utgör en samling av funktioner som underlättar samarbete inom projekt, dokument-, data- och informationshantering. Utöver detta kan man även hantera deltagare och deras engagemang inom olika åtaganden.

SharePoint per definition är ett komplext CMS (Content Management System) som skräddarsys utifrån de krav som tillfället efterfrågar. Många företag ser idag SharePoint som ett enkelt verktyg för att hantera information på ett så kallat intranät. Med detta kan användare utan avancerad teknisk kunskap skapa, hantera och administrera information i form av dokument, sidor eller dylikt på ett enkelt sätt. Med de verktyg som SharePoint i dagsläget erbjuder är det även enkelt att binda samman delar i ett företag för att på ett användarvänligt sätt administrera verksamheten [1].

Underlättande av samarbete inom verksamheten

SharePoint gör det möjligt för verksamheter att enkelt samarbeta på dokument eller annan information som berör särskilda områden. Genom olika grupper av deltagare kan man ge personer särskilda åtkomsträttigheter för innehåll, och även göra det möjligt för deltagare att skapa nytt, ändra eller ta bort innehåll. På detta sätt kan exempelvis olika avdelningar inom ett företag ha åtkomst till olika projekt och informationskällor. På så sätt är SharePoint ett system som främjar möjligheterna för samarbete mellan flera personer eller avdelningar inom en verksamhet.

Interoperabilitet

SharePoint har en stark koppling till andra produkter från Microsoft, i huvudsak Microsoft Office och dess tillhörande program. Detta möjliggör kommunikation mellan SharePoint och Office, exempelvis i form av att

kunna ange dokument skapade i Excel som en underliggande datakälla, eller att snabbt kunna redigera dokument skapade i Word direkt via SharePoint.

2.2 Testning

2.2.1 Vikten av testning

Mjukvaruindustrin präglas av hård konkurrens och vikten av hög kvalitet på produkter som levereras är mycket stor. Under utveckling av mjukvara förekommer därför en kontinuerlig ”Quality Assurance”-process, QA, för att ständigt säkerställa dels att produkten uppfyller vad kravspecifikationen avser, samt att produkten håller en tillräckligt hög kvalitet. Utöver detta måste utvecklaren försäkra sig om att det som utvecklas håller en hög teknisk kvalitet i form av robust kod fri från buggar och andra fel. Testning av mjukvara bör därför ha en stor roll i utvecklingsprocessen för att undvika att särskilda risker infaller.

Risker som på en ett indirekt sätt kan motverkas eller hanteras med hjälp av testning kan vara felaktig uppskattning av tid, brist på erfarenhet inom det område produkten utvecklas i eller sena ändringar i kravspecifikationen. Samtliga tre av dessa risker beskrevs i en undersökning redan 1991 [2], men återfinns som påtagliga risker inom mjukvaruprojekt även i senare undersökningar [3] [4]. Man kan tänka sig att kod som är skriven av mindre erfarna utvecklare har en större risk att innehålla fel eller buggar än utvecklare med flera års erfarenhet. Med hjälp av testning kan man identifiera dessa fel i ett tidigt skede och på så sätt minska risken för att man vid ett senare tillfälle måste spendera tid på att felsöka, eller att tiden för iterationen eller uppgiften löper ut.

På ett liknande sätt kan ett tidigt oupptäckt fel i mjukvaran även vara en påverkande faktor för huruvida framtida implementering av funktionalitet fungerar eller inte. Testning kan därmed vara en viktig faktor för att avstyra att fastställda perioder för särskilda moment överskrids tidsmässigt. På samma sätt kan testning hjälpa till att säkerställa att sena implementationer av ny funktionalitet fungerar tillsammans med existerande system.

2.2.2 Manuell & automatiserad testning

Tidigare genomförd forskning inom mjukvarutestning har visat att företag i mjukvaruindustrin investerar stora summor på infrastruktur för detta. Det är dessvärre inte alltid dessa uppfyller de önskade krav man

har på tekniken, och ett exempel på detta är den ofantliga summan av nästan 60 miljarder dollar som den årliga kostnaden för felaktig mjukvara uppgick till i USA år 2000 [5]. Detta innebär att en förbättrad metodik och infrastruktur av mjukvarutestning kan ha en stor ekonomisk påverkan på den framtida industrin [6].

Testning av mjukvara går att genomföra på flera olika sätt, och man skiljer vanligtvis mellan manuell och automatiserad testning. Det förstnämnda innebär som namnet antyder att alla testscenarion utförs manuellt av en testare som jämför det förväntade utfallet av testet med verkligheten. Det är upp till den ansvarige testaren att avgöra när utfallet är godkänt för ett lyckat genomfört test. I flera fall kan manuella tester vara att föredra framför automatiserade tester; exempelvis kan projektet som testas vara av sådan liten skala att det inte upplevs som ekonomiskt och resursmässigt motiverat att investera i verktyg som möjliggör automatiserad testning. Likväl är automatiserade tester just automatiska, vilket innebär att manuella tester har en större sannolikhet att komma närmare och identifiera verkliga problem eller fel som en slutanvändare skulle kunna beröras av. Manuell testning kan även vara ett flexibelt alternativ när man i en snabb takt vill implementera ny funktionalitet till en produkt, då det kan vara tidskrävande att skriva tester som ska automatiseras.

Ser man istället till automatiserad testning så finns det många faktorer att ta i beaktning inför ett beslut mellan de två alternativen. Just att testerna är automatiserade gör det möjligt att köra dem repetitivt flera gånger om. Detta i sin tur eliminerar en av nackdelarna med manuell testning vilket är att det är tidskrävande och saknar stimulans för testaren i att gång på gång utföra samma testscenario. En utredning genomförd 2005 [7] uppskattar att alla test i ett mjukvaruprojekt körs minst fem gånger, medan en fjärdedel av alla test i projektet körs över 20 gånger. När detta sätts i relation till att tester överlag är mer använt i dagsläget än för snart tio år sedan bekräftar man att repetition av tester är ett stort argument och faktor för att gå från manuellt utförda till automatiserade tester. I en automatiserad testmiljö har man även fördelen att samtliga inom utvecklingsprojektet kan följa testningen då resultatet är synligt direkt i systemet. Fördelar såsom denna kan vara en bidragande faktor till bättre samarbete mellan projektmedlemmar.

Som ovan nämnt är det inte alltid motiverat att investera resurser och pengar i att implementera automatiserade tester till en

utvecklingsprocess. Verktyg för automatiserade tester kan vara dyra, och väl inköpta måste tid investeras för att valuta för pengarna ska ges. Avsevärt mycket tid kan placeras på att skriva tester, och körningen av dessa kan ibland uppgå till flera timmar. En jämförelse av olika verktyg för automatiserad testning och vad man bör tänka på vid val mellan dessa återges i avsnitt 2.3.

För ett företag att besluta sig för att gå vidare med automatiserad testning bör en tydlig motivering till detta finnas. Studier har visat att egenskaperna för den produkt som testas har en direkt påverkan på huruvida automatiserade tester är en applicerbar metod att tillämpa eller ej. [5] Samma studie klargör utifrån en observation av fem företag som tillämpar automatiserad testning att det optimala fallet för automatiserad testning borde vara när det implementeras som ett standardiserat ramverk med en stabil och konsistent plattform. Detta med tydliga resultat för utvecklarna att tolka, och utan att mänskliga ingripanden behöver göras i större utsträckning.

2.2.3 Olika typer av test

Test kan struktureras och genomföras på många olika sätt. Enligt [8] är enhetstest, komponenttest och systemtest bra tillvägagångssätt för att täcka in de tillfällen då testning bör utföras för att säkerställa korrekthet i den utvecklade produkten.

Enhetstestning

Enhetstestning innebär att tester skrivs och anpassas för specifika metoder eller funktioner i ett system. Denna typ av test syftar till att verifiera den lägsta detaljnivån av systemets funktionalitet.

Komponenttestning

Programkod består i de allra flesta fallen av ett antal metoder som kommunicerar med varandra och tillsammans skapar en särskild funktionalitet i ett system. Denna typ av funktionalitet kallas för en komponent, som även den behöver testas för att verifiera att alla delmetoder fungerar på ett korrekt sätt tillsammans.

Systemtestning

Här integreras de olika delkomponenterna i ett system till det kompletta systemet. Med syfte att verifiera att komponenterna kan interagera med varandra utan att fel uppstår.

GUI-testning

En metod där testerna och dess resultat baseras på interaktion via gränssnittet. Denna typ av test behandlas noggrannare i avsnitt 2.3.

2.3 GUI-testning

2.3.1 Motivering

Gränssnitt i webbapplikationer kan vara komplexa med möjlighet att utföra många olika uppgifter. Eftersom användaren interagerar med webbapplikationen genom användargränssnittet behöver detta ändras för att återge feedback som talar om för användaren att något händer. Detta exempelvis genom att en ny vy visas. I arbetets sammanhang syftar GUI-testning till att verifiera att det är rätt vy som visas efter att en viss interaktion har skett.

Även om bakomliggande kod för GUI kompilerar felfritt kan fel uppstå i gränssnittet efter användarens interaktion. Detta innebär att GUI-testning kan ses som en form av systemtestning, men i de flesta fall klassificeras det som en egen typ av testning.

2.3.2 Vad kan man testa?

Möjligheterna till vad som testas i ett användargränssnitt är breda och testningen kan därför göras omfattande. Enklast är kontroll av funktionalitet hos gränssnittets knappar och länkar för att se till att dessa genererar rätt resultat. Testning kan däremot tas ytterligare ett steg längre, exempelvis test för att valideringen av det data som matas in i formulär fungerar eller för gränssnittets interaktionsdesign och dess kognitiva egenskaper.

I den här rapporten begränsar vi oss till relativt enkla tester som kan utföras mot användargränssnittet, tester som enbart verifierar användargränssnittets vyer och inte någon egentlig funktionalitet. Detta eftersom målet med denna rapport är att hitta en lösning för att automatisera GUI-test och inte hur GUI-test bör utformas. Fortsättningsvis när vi talar om GUI-test i rapporten är det denna typ av GUI-testning som avses.

2.3.3 Testverktyg

För att genomföra GUI-test krävs först och främst ett verktyg som definierar och genomför testerna i fråga. Detta kan vara en integrerad del av ett utvecklingsverktyg eller ett verktyg från en tredjepart. Verktygen har alla sina för- och nackdelar i programvarustöd och funktionalitet.

Tabell 1 nedan visar ett urval av lämpliga verktyg som kan användas för GUI-test och som är tillämpbara för Precios ändamål. Observera att särskilda testverktyg måste användas i kombination med ett testramverk, exempelvis NUnit [9] eller MSTest [10], för att testerna ska kunna köras.

Verktyg	Skriptspråk		Webbläsare			Pris
	C#	Visual Basic	Internet Explorer	Chrome	Firefox	
Microsoft Coded UI	✓		✓	Använder Selenium plugin	Använder Selenium plugin	Visual Studio Premium/Ultimate
HP QuickTest Pro		✓	✓	✓	✓	Licensierad
Selenium	✓		✓	✓	✓	Gratis
Silk Test	✓		✓	✓	✓	Licensierad
Telerik Test Studio	✓		✓	✓	✓	Licensierad

Tabell 1 - Testverktyg som kan användas för GUI-test. Information hämtad från respektive verktygs hemsida.

Gemensamt för alla ovanstående verktyg är att testerna implementeras med två metoder, antingen inspelade test eller kodade test. Vi har valt att lägga fokus på att utreda hur de fungerar i fallet för kodade test.

2.3.4 Inspelade test

Det här är det enklaste tillvägagångssättet för att implementera automatiserad testning av GUI. Det finns inget krav på att personen som skapar test behöver vara utvecklare, utan fokus läggs enbart på interaktion med användargränssnittet.

För att skapa ett inspelat automatiserat test startas en inspelningssession där testaren manuellt interagerar med användargränssnittet. Baserat på interaktionen genereras ett skript som kan användas för att spela upp GUI-testet på nytt. Uppspelningen genomför då samma interaktion som gjordes manuellt under inspelningssessionen.

En nackdel med inspelade test är att de skript som genereras är hårdkodade och därför inte kan anpassa sig efter ändringar i gränssnittet. En förändring av bakomliggande gränssnittskod kan därför innebära att inspelade test blir ogiltiga och att de måste göras om på nytt. Ett inspelat test går att jämföra med vad som ofta kallas för ”macro” inom utveckling, dvs. kod som är skriven och kan spelas upp repetativt.

2.3.5 Manuellt kodade test

Här programmeras testskripten manuellt istället för att genereras. Kodens syntax varierar baserat på det verktyg som används. Många verktyg har anpassats för integrering med etablerade programmeringsspråk som exempelvis C# och använder då dess syntax.

Kodade test har en fördel i att de kan göras dynamiska och är därmed mer anpassade för ändringar. I längden innebär det att mindre tid behöver spenderas på att uppdatera gamla tester som blivit ogiltiga. En ytterligare fördel är att testerna inte begränsas till att använda endast applikationens gränssnitt utan även kan utnyttja resurser i operativsystemet eller annan programkod.

2.3.6 Val av testverktyg

De senaste åren har ett antal verktyg för GUI-testning tagits fram. Detta innebär att valet av verktyg inte är självklart då alla har sina för- och nackdelar. I första hand måste verktyget ha stöd för den plattform man eftersträvar att testa, samtidigt är det framtidsmässigt bra att välja ett verktyg som stöder flera olika plattformar och utvecklingsverktyg. Oavsett om verktyget är gratis eller licensierat är inlärningstiden alltid en kostnad, därför är detta bra att ta i beaktning vid val av verktyg.

Vidare är det viktigt att välja ett verktyg som kontinuerligt utvecklas. Det lanseras ständigt nya versioner av webbläsare, operativsystem och utvecklingsmiljöer. Detta ställer krav på att utvecklingsverktyget uppdateras i samma tempo för att kunna användas på bästa sätt. Det bör åtminstone ges stöd för den aktuella versionen av målplattformarna. Samtliga av dessa faktorer bör alla tas i beaktning vid val av verktyg för testning.

2.3.7 GUI-testning mot SharePoint

SharePoint är ett ramverk för webbapplikationer och körs således enbart i en webbläsare där HTML används för att visualisera användargränssnittet. För att utföra GUI-test mot en SharePoint applikation krävs därför inget särskilt SharePoint-stöd. För att testa SharePoint krävs enbart ett verktyg som kan utföra test mot webbsidor.

2.4 Agil utveckling & Continuous Integration

2.4.1 Agila arbetsmetoder

Inom mjukvaruindustrin tillämpas flera olika typer av arbetsmetoder, där agila utvecklingsprocesser under de senare åren fått ett starkt fäste.

Nämnvärda agila utvecklingsmetoder är exempelvis extreme programming (XP), funktionsdriven utveckling (FDD), och Scrum, där den sistnämnda är den mest använda metoden bland agila utövare världen över (2013) [11].

Att arbeta enligt en agil arbetsmetodik innebär att man delar upp projekt i iterationer, som vardera sträcker sig över en kortare period, ofta inte längre än några veckor. Dessa iterationer bryts sedan ned till mindre uppgifter som genomförs av en grupp deltagare som alla har till uppgift att hantera kravanalys, design, programmering och olika typer av testning. Uppgifterna genomförs med minsta möjliga planeringstid, där man istället har dagliga möten för att gå igenom vad som genomfördes föregående dag och var fokus kommer att ligga under dagen. En iteration i sin helhet avslutas sedan i de flesta fall med ett möte med samtliga av projektets intressenter där man går igenom vad iterationen avhandlat och där man demonstrerar den för stunden aktuella versionen av produkten.

Agil utveckling leder till att man vid frekventa tillfällen kan lansera en ny version av en produkt [12], även om produkten vid tillfället saknar delar av den funktionalitet som den initiella kravspecifikationen avser. Detta resulterar i många fall till ökad kundnöjdhet tack vare kontinuerliga lanseringar, och att kunden enkelt kan följa utvecklingsprocessen av produkten.

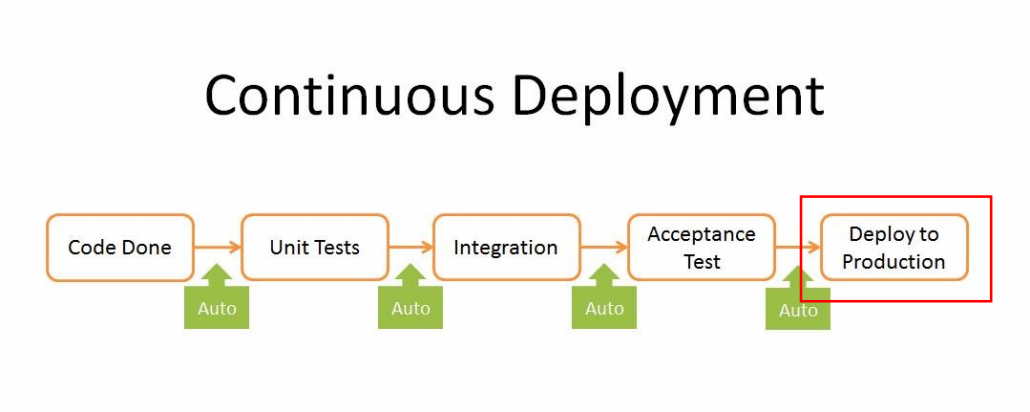
Agila arbetsmetoder är ofta direkt kopplade till särskilda processer som möjliggör för det agila tillvägagångsättet. När det gäller just utvecklingsprocessen och produktion av kod är CI den mest använda processen [11] för att snabbt kunna bygga och integrera ny kod med existerande kodbas.

2.4.2 Continuous Integration – CI

Continuous Integration är en tillämpning av ett agilt arbetsätt som tillåter utvecklare att kontinuerligt och ofta integrera kod till mjukvara på ett enkelt och i de flesta fall fullt automatiserat sätt.

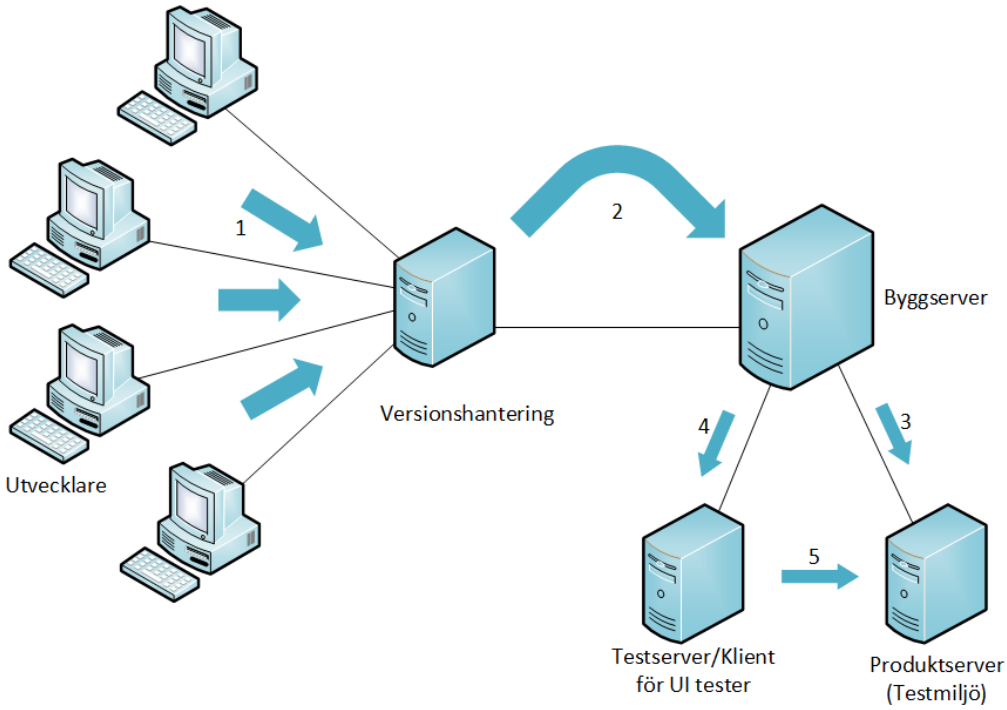
CI ska inte förväxlas med vad som kallas för ”Continuous Deployment” (CD), figur 1, vilket innebär att slutkunden vid jämna mellanrum erhåller en ny uppdaterad version av produkten. Detta kan ses som en förlängning av CI där man lagt till steget som utgör release och leverans

till kund.



Figur 1 - Continuous deployment, inte att förväxla med CI.

CI grundar sig på observationen att ju längre ett steg i en utvecklingsprocess pågår och avviker tidsmässigt från andra steg, desto mer tid spenderar enskilda utvecklare på individuellt arbete utan att synkronisera kod med övriga utvecklare. Detta resulterar i sin tur i att integrationen av olika utvecklares kod blir svårare ju mer tid som går. Lösningen till problemet är att integrera kod mer frekvent och kontinuerligt, även vid små ändringar. Tillämpning av detta får samtidigt inte sänka kvaliteten på kod vilket har lett till att CI ofta implementeras som en ”bygg & testa”-process, där varje incheckning av ny kod testas utförligt och ger direkt återkoppling till utvecklaren om resultatet av testerna.



Figur 2 - Typisk implementation av continuous integration.

Figur 2 visar ett exempel på typiskt implementation av CI, där utvecklare arbetar individuellt och sedan checkar in kod (1) till en gemensam server som sköter versionshantering. Vid incheckning av kod sker ingen typ av testning, och huruvida olika utvecklares kod integrerar med varandra är i det här stadiet okänt. En incheckning av kod utlöser dock oftast per automatik ett nytt bygge av all kod (2) där det framgår om koden integrerar korrekt eller inkorrekt med övrig existerande kod. Olika typer av tester kan vara kopplade till detta bygge, men då denna rapport fokuserar på testning av GUI visar figuren istället hur byggservern i nästa steg installerar den byggda koden till en server där produkten körs (3). Tester mot GUI kan sedan initieras (4) för att låta en dedikerad server/klient genomföra dessa (5). Rollen som testservern i exemplet ovan har går att jämföra med den roll en person har som interagerar med en webbapplikation. Testerna som utförs är en form av automatiserad acceptanstestning, dock ej något som bör ersätta den slutgiltiga acceptanstestningen, förutsatt att man inte arbetar fullt ut efter en ATDD-modell (Acceptance Test Driven Development).

Som utvecklare inom ett team som tillämpar CI följer man oftast principerna att aldrig checka in kod som inte byggs lokalt utan fel, inte checka in otestad kod, och att aldrig checka in kod när servern inte lyckats

bygga den senast incheckade versionen av koden. Detta leder till att agila utvecklingsteam sköter sig själva i stor utsträckning, utan att man behöver sätta upp en policy på företagsnivå kring hur kod ska hanteras.

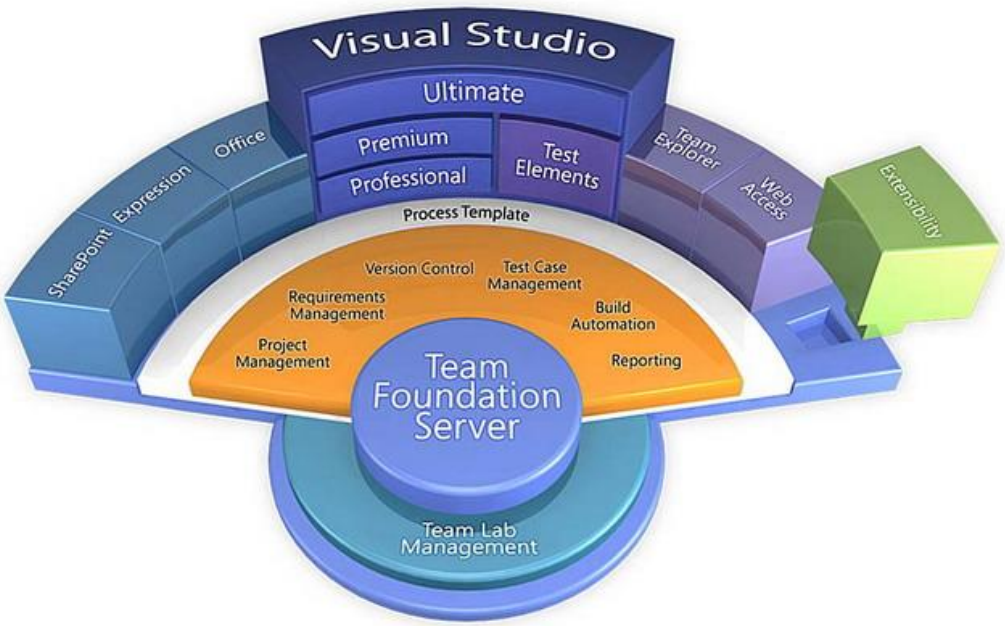
2.4.3 Arkitektur över Microsoft Team Foundation Server

Precio tillämpar i dagsläget en form av Scrum-process på sina projekt, vilket innebär att de arbetar enligt en agil utvecklingsmodell. Då Precio är ett företag inriktat på utveckling av mjukvara som bygger på Microsofts plattformar använder de sig även av verktyg utvecklade av Microsoft för att underlätta arbetet. I grunden finner man Microsoft Team Foundation Server, TFS, som är kärnan till all form av samarbete utvecklare emellan. Kopplat till detta finns olika servrar som sköter olika uppgifter inom livscykeln för mjukvaran som utvecklas. Som utvecklingsverktyg används Microsoft Visual Studio för att kopplingen mellan utvecklare och TFS ska vara så effektiv som möjligt.

I nedanstående rubriker beskrivs de olika delarna av TFS och hur arkitekturen kan tänkas se ut enligt Microsofts rekommendationer. Den existerande arkitekturen för hur detta är implementerat hos Precio finns även beskrivet i den sista underrubriken.

TFS & TFS-server

TFS är kärnan i Microsofts plattform för att hantera livscykeln för mjukvaruapplikationer. TFS är en plattform för samarbete som möjliggör för gemensam utveckling inom ett och samma projekt, där man har tillgång till bland annat versionshantering för kod, rapporthantering, kravsamling, projekthantering, testning och mycket mer. Detta illustreras i figur 3 i det orangea området. Genom att använda TFS sammankopplat med Microsoft Visual Studio kan man skapa en så kallad ”Sprint Backlog”, där alla olika moment för den aktuella sprinten innefattas. Momenten bryts sedan ned i konkreta uppgifter som tidsestimeras, och tilldelas olika deltagare inom projektet. Tack vare detta finns på så sätt alltid en god överblick av vem som för tillfället arbetar med vad. Detta beskrivs även mer utförligt i avsnitt 4.2.



Figur 3 - Kopplingen mellan TFS och utvecklingsverktyg och plattformar. [Källa: Microsoft]

TFS är installerat på en separat server dit enskilda datorer för utvecklare sedan är anslutna via Team Explorer (lila området i figur 3). Från Team Explorer kan utvecklare ansluta till utvalda projekt som återfinns i en projektsamling, förutsatt att de är behöriga för detta. Man kan via Team Explorer tilldela utvecklare olika rättigheter tillhörandes olika projekt, för att på så sätt styra vem som har tillgång till vad och i vilken omfattningen utvecklaren kan ändra och redigera projektet.

I varje existerande projekt har man tillgång till versionshantering där man checkar in kod till TFS-servern. Vid en incheckning av kod kan man även låta det aktiveras ett nytt bygge på byggservern som initieras av TFS-servern. Detta sker med hjälp av ett automatiserat arbetsflöde som utlöser ett anrop mellan TFS-servern och byggservern.

TFS-servern är alltid den centraliserade delen som sköter kopplingar mellan olika andra servrar och arbetsmoment. Det är denna som sköter kommunikationen mellan exempelvis byggserver och testserver.

Byggserver & Byggdefinition

För att kunna tillämpa CI med hjälp av TFS krävs att en server dedikerad för bygge av incheckad kod integreras i arkitekturen. Som namnet antyder har servern till uppgift att bygga den senast incheckade koden

och rapportera tillbaka resultatet av detta. Tack vare att man använder Visual Studio kan man få en tydlig bild av resultatet från ett specifikt bygge.

En byggserver består av två essentiella moduler; dels en bygg-controller som har till uppgift att ta emot byggförfrågningar från TFS-servern, och en eller flera bygg-agenter. En bygg-agent har till uppgift att genomföra själva bygget av kod, och tilldelas arbeten av bygg-controllern.

En byggserver kan bestå av en eller flera bygg-agenter, förutsatt att en bygg-controller är installerad och sammankopplad med dessa på en annan server. Flera bygg-agenter kan vara fördelaktigt i situationer då man behöver kunna bygga kod parallellt eller vill öka skalbarheten av byggen. Figur 4 visar en översiktsbild på de olika komponenterna.

För att kunna installera en bygg-controller och tillhörande agenter på en server krävs en version av dessa som är kompatibel med den version av TFS som används. Det finns inga krav på att använda någon specifik version av Visual Studio.

I TFS Team Explorer har man i varje projekt möjlighet att skapa en så kallad byggdefinition, i vilken man specificerar huruvida ett bygge av incheckad kod ska ske vid särskilda tillfällen. Exempelvis kan man ställa in i definitionen att ett bygge ska initieras vid varje incheckning av kod, eller vid särskilda tillfällen på dygnet. En vanlig tillämpning här är så kallade ”nightly builds”, där byggservern under natten bygger all incheckad kod sedan det senaste bygget.

Testserver

På ett liknande sätt som en byggserver kan man i TFS-arkitekturen integrera en separat server för att utföra tester. Detta möjliggör enhetstestning, GUI-testning och belastningstestning från en fristående server som anropas på samma sätt som mot en byggserver. Figur 2 visar en tydlig bild på hur kopplingen dessa emellan fungerar.

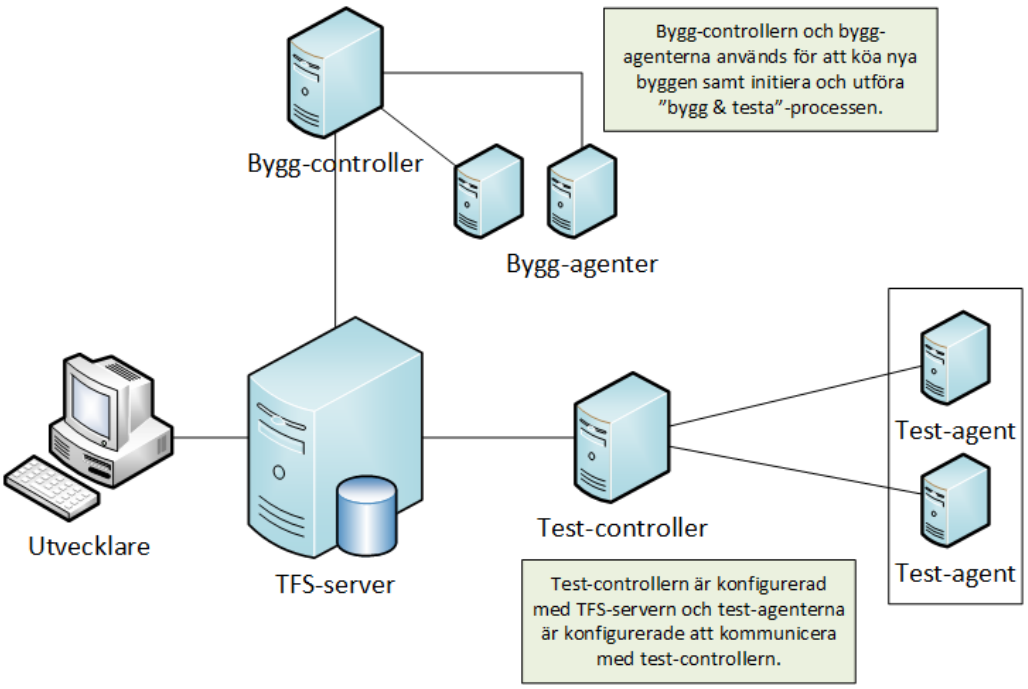
För att en testserver ska kunna användas behövs det en installerad test-controller med en eller flera tillhörande test-agenter. En test-controller har i likhet med en bygg-controller till uppgift att ta emot förfrågningar för att genomföra tester, för att sedan delegera dessa som uppgifter till olika test-agenter. En test-agent kan vara installerad på samma server som en test-controller, eller på en egen server förutsatt att den är

konfigurerad att kommunicera med test-controllern. På så sätt kan man distribuera de olika test-agenterna till olika testmiljöer för att få testningen så skräddarsydd som möjligt.

Till skillnad från byggservern med installerade bygg-controllers krävs det att den version av test-controller som är installerad på testservern är av samma version som den underliggande TFS-servern [13]. Även Visual Studio måste vara av samma version som den installerade test-controllern för att dessa ska kunna kommunicera med varandra. Detta kan leda till problem mellan icke kompatibla produkter för företag som använder olika versioner av särskilda verktyg.

Arbetsflöde inom TFS & CI

TFS-servern utgör den styrande servern i en utvecklingscykel där man tillämpar CI. Det är via denna som man sammankopplar de olika fristående servrarna till en större arkitektur, och som möjliggör automatiseringen av bygge och testning.

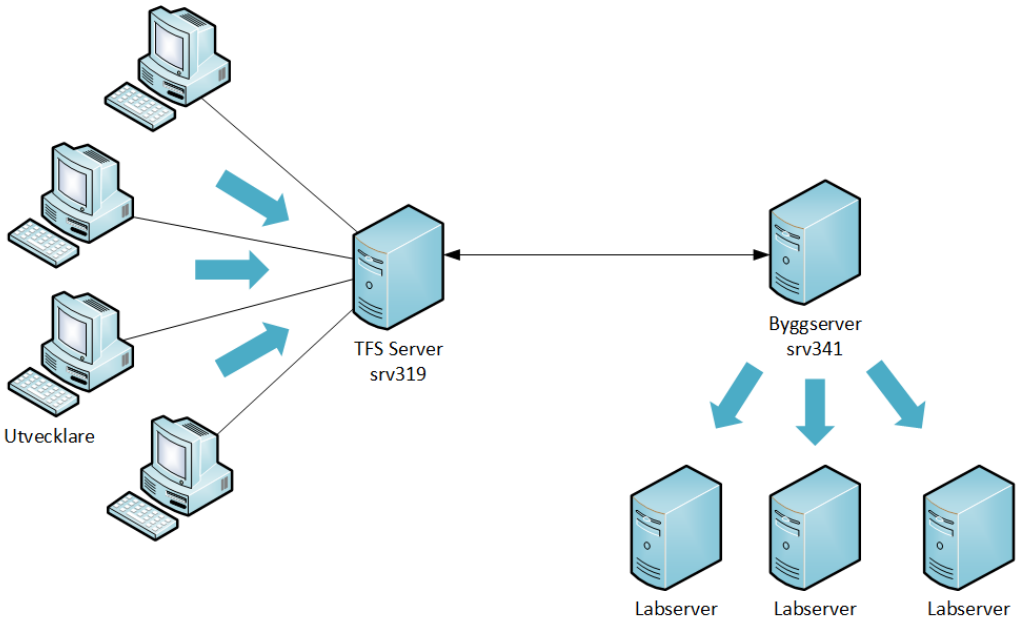


Figur 4 - Microsoft rekommenderade arkitektur för TFS & CI.

Ovanstående figur 4 visar hur det är tänkt att man kan använda Microsoft Team Foundation Server för att skapa en helautomatiserad lösning från incheckning av kod, till bygge och testning.

Existerande implementation av TFS & CI hos Precio

Precio använder i dagsläget TFS för att hantera projekt och versionshantering av kod, och har även en struktur för hur CI implementeras i olika projekt. En byggserver finns uppsatt för att hantera dagliga byggen och installation av projekt till labservermiljöer. Det existerar dock inte någon etablerad lösning för hur testning ska genomföras, varken för enhetstestning eller GUI-testning, och om en utvecklare vill genomföra någon form av testning är det helt upp till denne att själv se till att det blir utfört.



Figur 5 - Illustration över existerande arkitektur hos Precio.

Jämför man figur 4 med figur 5 ser man att det i den sistnämnda figuren helt saknas någon form av implementation av testserver.

3 Metod

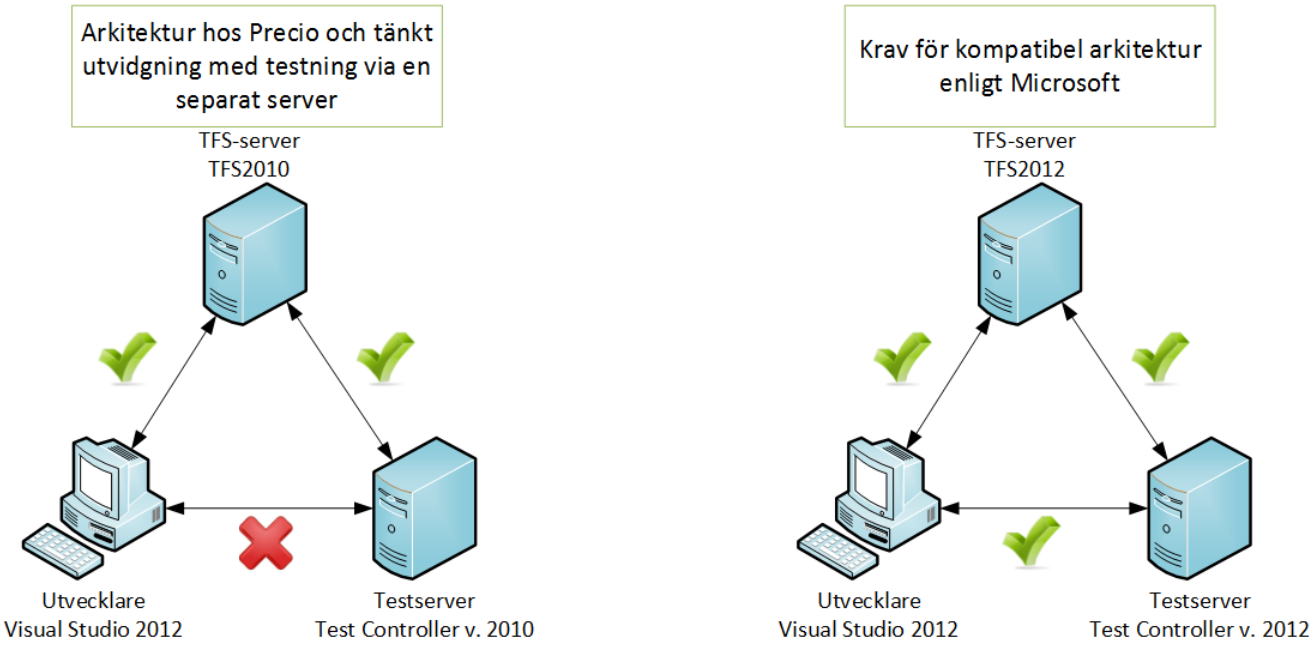
3.1 Arbetsmetod

Ett av våra delmål har varit att finna en lämplig arbetsmetod för genomförande av projektet. Det som är tänkt att konstruktionsdelen av arbetet ska leda fram till har varit vagt definierat, och högst benäget för ändringar i kravspecifikationen. Då Precio tillämpar en form av Scrum-metod för sina projekt har vi haft möjlighet att följa det i närhet, vilket även lett till att valet av arbetsmetod fallit på Scrum för vår arbetsmetodik. Det motiveras lämpligen även med de delmål vi har haft uppsatta, som till stor del varit att undersöka och anpassa arbetet efter det bäst funna alternativet, vilket således gör att Scrum är en utmärkt arbetsmetod för oss då den stödjer en kontinuerligt förändrad kravbild.

3.2 Val av teknik

När det kommer till val av teknik har vi under förstudiens gång bildat oss en god uppfattning om hur man med hjälp av TFS kan implementera en helautomatiserad lösning för GUI-testning. Eftersom den automatiserade testningen är tänkt att vara en av pelarna i CI har vi även implementerat CI på det SharePoint-projekt vi arbetat med. Detta för att förstå processen i hur det installeras och konfigureras i Precios arkitektur.

Dessvärre har upptäcktes även att särskilda krav som ställs för att inkludera automatiserad testning med TFS i CI inte uppfylls i den existerande utvecklingsmiljön där testningen ska implementeras. I förstudien av arkitekturen för TFS, se avsnitt 2.4.3, kan man läsa att ett av kraven för att installera en test-controller bland annat är att samma version av TFS, test-controller och Visual Studio används för att samtliga komponenter ska kunna kommunicera med varandra. Då Precio använder sig av TFS version 2010, och Visual Studio version 2012 som utvecklingsverktyg är det därför inte möjligt för oss att implementera automatisering enligt den modellen. Figur 6 illustrerar vad som skulle krävas för en lösning på detta.



Figur 6 - Illustration över icke-kompatibla versioner av mjukvara till vänster, och en korrekt implementation till höger.

Vi kommer av den anledningen istället implementera en annan modell av automatiserad testning som även den är sammankopplad till existerande CI-process. Den bygger på att använda etablerade verktyg och ramverk för GUI-testning för att genomföra själva testerna, och sedan använda script för att initiera testningsprocessen på ett automatiserat sätt. Utifrån den jämförelse av existerande verktyg för GUI-testning, se tabell 1 i avsnitt 2.3.3, har vi valt att använda oss av Selenium som testverktyg för vår konstruktion. Detta för att Selenium dels är öppen källkod och gratis att använda, men även för att tester enkelt går att skriva i ren C#-kod i stil med vanliga enhetstest i Visual Studio. Det finns även en viss kännedom om Selenium hos anställda hos Precio, vilket motiverar valet av detta än mer. Som tidigare nämnt i avsnitt 2.3.7 behöver man inte heller något särskilt verktyg för att kunna genomföra GUI-testning mot SharePoint, utan att det räcker med ett verktyg som klarar av testning av webbsidor. För att köra testerna skrivna i Selenium kommer vi använda ramverket NUnit 2.0 [9] som i sin tur bygger på ramverket xUnit som även ligger till grund för JUnit, SUnit och andra enhetstestningramverk.

All utveckling kommer att ske i Visual Studio 2012, skrivet i programspråket C#. Vi kommer att skriva automatiseringsscripten i Windows PowerShell-format [14] då dessa går att anropa på ett enkelt sätt via TFS och Visual Studio.

4 Konstruktion

Den här delen av rapporten beskriver den konstruktion för automatiserad testning som vi tagit fram och implementerat. Notera att den frågeställning som ursprungligen presenterats i avsnitt 1.2 redan besvaras i stora drag i förstudien, och att detta avsnitt är ett förslag på hur automatiserad testning kan implementeras.

Här redogörs hur en konstruktion av automatiserad testning baserad på Selenium och NUnit tillsammans med script kan implementeras i en miljö där CI redan är tillämpat, men som saknar stöd för GUI-testning. De verktyg och den teknik som används finns beskrivet i avsnitt 3.2.

4.1 Kravspecifikation

Den kravspecifikation som arbetats efter har tagits fram av oss som genomfört arbetet tillsammans med intressenter i form av handledare på Precio. Detta då det inledningsvis inte existerat någon konkret målbild av vad en möjlig implementation skulle innebära. Istället har en vision av vad som är möjligt och önskvärt vuxit fram under arbetets gång och därmed legat till grund för en kravspecifikation. Kravspecifikationen är uppdelad i funktionella respektive icke-funktionella krav samt en övergripande beskrivning av vad visionen för slutresultatet är. Värt att notera är att kravspecifikationen i vårt fall varit relativt oklar gällande önskad funktionalitet för en möjlig implementation av automatiserad testning av GUI.

4.1.1 Vision

Precio bygger många komplexa webbaserade system, där flertalet av dessa har ett lika komplext användargränssnitt. Att testa hela användargränssnittet manuellt är oftast inte kostnadseffektivt och Precio söker därför en automatiserad lösning för att genomföra testfasen. Detta skulle innebära att mer tid kan investeras i utveckling för ökad kvalitet samt en reducering av kostnader för kontinuerlig manuell testning. Önskemålet för en implementation av automatiserad testning är att den ska vara inbyggd i existerande CI-process och att den ska vara enkel att tillämpa på godtyckliga projekt. Utöver detta ska det vara enkelt för enskilda utvecklare att skriva egna testfall som sedan automatiseras utan att större ingrepp i konstruktionen måste göras.

4.1.2 Funktionella krav

- Implementationen **ska** kunna köra tester mot en i konfigurationsfilen specificerad SharePoint-applikation
- Implementationen **ska** kunna köra skräddarsydda tester paketerade i DLL-filer skapade av enskilda utvecklare
- Användaren **ska** kunna konfigurera testerna via en XML-baserad konfigurationsfil

4.1.3 Icke-funktionella krav

- Implementationen **ska** vara enkel att installera på godtyckliga projekt
- Det **ska** vara enkelt för användaren att använda implementationen för egna skrivna testfall

4.2 Arbetsutförande

För att uppnå delmålen i projektet har vi valt att fokusera arbetet i tur och ordning på olika områden som var för sig knyter an till olika mål. Inledningsvis har tid investerats i att klargöra vad för typ av lösning som i dagsläget saknas hos Precio, och vad Precio önskar att arbetet ska leda fram till. Detta har genomförts med en initiell kravformulering för att bättre kunna strukturera upp arbetet. Arbetet efter detta har sedan följts av två faser, en förstudie respektive en konstruktionsfas. Förstudien har genomförts för att angripa de problem- och frågeställningar som etablerats och hur detta genomfördes beskrivs nedan.

4.2.1 Fas 1 – Förstudie

Förstudien genomfördes för att ge svar på olika delmål hörandes till arbetet. Samtliga delar av förstudien finns beskrivna i avsnitt 2, ”Bakgrundsmaterial”, men hur dessa genomförts redogörs här nedan för hur arbetet med dessa utfördes.

Literatturstudie

Förstudien har innefattats av en djupgående analys av existerande material inom de olika områden vårt arbete berör. Dels har vi sökt och läst åtskilliga avhandlingar som publicerats och som handlat om just olika typer av testning, men framförallt med fokus på GUI-testning. Det råder en oklar bild om vad som definieras fullt ut som just GUI-testning vilket gjorde att vi i det här stadiet fick ta ställning och tolka det hela på

ett sätt som gjorde det användbart och implementerbart i vår lösning. Mycket material var inte heller till någon användning då det enbart avhandlade konstruktionen av inspelade tester, vilket vi i ett tidigt skede valde bort för att istället fokusera på kodade tester.

SharePoint

I och med att prototypens slutgiltiga mål för Precios del varit att tillämpa det på projekt för SharePoint, upplevde vi i ett tidigt skede att det var viktigt för oss att förstå och få en uppfattning om hur SharePoint fungerar och hur tester kan utföras gentemot det. Detta hörde inte till ett specifikt delmål, men kändes essentiellt för att arbetet skulle kunna fortgå.

Verktyg för GUI-testning och automatisering

Som en del för att ge svar på frågor och finna lösningar till våra problemformuleringar har mycket av förstudiens tid investerats i att undersöka existerande produkter för testning av GUI. En utförlig jämförelse med perspektiv på olika faktorer såsom kostnader, tillämpningsområden, produktstöd etc. genomfördes här för att till slut resultera i en god överblick av produkter, från vilken en slutsats och ett val av verktyg kunde genomföras.

En viktig faktor utöver hur verktygen kan användas var även hur möjligheterna för automatisering såg ut. I och med att vi i detta skede inte hade en färdig bild av hur integrationsprocessen skulle se ut var vi tvungna att lämna detta öppet för vidare beslut senare under projektets gång.

Den här delen av förstudien har legat till grund för de beslut kring teknik vi tagit inför konstruktionen.

Continuous Integration

Det stora övergripande målet har varit att undersöka hur en lösning för automatiserad testning går att integrera med en existerande processmodell, vilket i fallet för Precio inneburit CI. För att förstå bättre hur det fungerar har vi därför tillämpat en likadan process på ett eget projekt, där vi installerat allt från början till slut. I och med detta fick vi utöver en god förståelse för CI även en bra inblick i hela utvecklingscykeln, och idéer om hur vår automatiserade testning skulle gå att implementera.

Detta skedde genom att simulera en liknande form av projekt som de projekt som Precio i vanlig ordning genomför.

4.2.2 Fas 2 – Konstruktion

Konstruktionsfasen har utförts i iterationer helt enligt Scrum-metodik. Lösningsförslag och att delmål såsom val av teknik var uppfyllda bekräftades inför en implementation i existerande processmodell för att konstruktionsfasen skulle gå att genomföra. Varje steg utfördes sedan med uppbackning av en Sprint backlog, där större uppgifter bröts ned till mindre moment som enklare kunde utföras steg för steg. Vi upplevde i ett tidigt skede att det kändes viktigt att testa alla olika delar utförligt innan de kopplades samman med övrig konstruktion. Exempelvis lades mycket tid och fokus på att etablera en kontakt mellan olika servrar på ett hållbart sätt, där anslutningen inte riskerades att brytas och på så sätt även avbryta testningen.

Att arbeta enligt den här metoden har underlättat för vårt arbete, då det varit enkelt att angripa de problem som vi varit tvungna att lösa. Mycket av det som vi konstruerat är skräddarsytt och har därför förändrats under konstruktionstiden; något som valet av arbetsmetod underlättat för vår del.

4.3 Implementation av GUI Test

Eftersom varje utvecklingsprojekt har olika utseende och implementationer av GUI blir det problematiskt att skapa en automatiserad lösning med färdiga test som komplett kan testa ett gränssnitt. Därför har vi i vår lösning valt att använda två olika typer av test, standardtester och egenskrivna tester.

4.3.1 Standardtester mot SharePoint

Detta är generella testfall som ska fungera på alla SharePoint-webbsidor. Testerna utnyttjar standardelement i SharePoints HTML som alltid har samma namn och som då kan användas för att identifiera specifika händelser i gränssnittet. Detta öppnar upp för möjligheten att utveckla generella SharePoint-test. Exempelvis finns standardfelkoden 404 för webbprotokollet HTTP, som används när en webbsida inte kan hittas. I SharePoint visas detta med ett standardelement som kan identifieras med hjälp av Selenium.

4.3.2 Egenskrivna tester

Här kan utvecklare skriva egna GUI tester för specifika projekt. För mer information om detta, se arkitekturdokument i bilaga A.

4.4 Testning med Selenium & NUnit

Vi har valt att utföra all form av testning med hjälp av Selenium som är ett automatiseringsramverk för webbläsare. Med hjälp av detta kan man skapa automationer som används inom testning. Selenium erbjuder användaren möjlighet att skapa både inspelade och kodade tester. En fördel med Selenium när det gäller kodade tester är att det i flera etablerade programmeringsspråk finns implementationer av det domän-specifika språket som används för automatiseringen, bland annat i C#. Tack vare detta är det möjligt att skriva tester i det språk man känner sig mest bekväm att arbeta i eller det som passar bäst in för projektet, vilket i vårt fall inneburit just C#.

Det går inte att använda Selenium som fristående komponent, utan som det är beskrivet i avsnitt 2.3.3 måste man använda sig av ett testramverk för att de testfall man skrivit ska utföras. I vår konstruktion använder vi NUnit 2.0 som testramverk.

4.4.1 Att skriva tester med Selenium

Selenium använder sig av WebDriver API [15], vilket är ett öppet interface för automation av webbläsare och som även är helt neutralt till olika programmeringsspråk. Med hjälp av WebDriver API kan man exempelvis lokalisera specifika element på en webbplats genom särskilda anrop. I Selenium utgår man alltid från en implementation av en WebDriver, där den motsvarar en specifik instans av en webbläsare. Således kan man i ett testfall skapa flera olika instanser av WebDriver som får representera olika webbläsare. Ett enkelt exempel visas nedan i form av C#-kod där man använder sig av Selenium för att skapa en instans av webbläsaren Firefox som sedan instrueras att navigera till Google.

```
public void GoToGoogle()
{
    //1. Skapa en instans av webbläsaren Firefox
    IWebDriver driver = new FirefoxDriver();
    //2. Säg åt Firefox att navigera till URL'en för Google.
    driver.Navigate().GoToUrl("http://www.google.se");
}
```

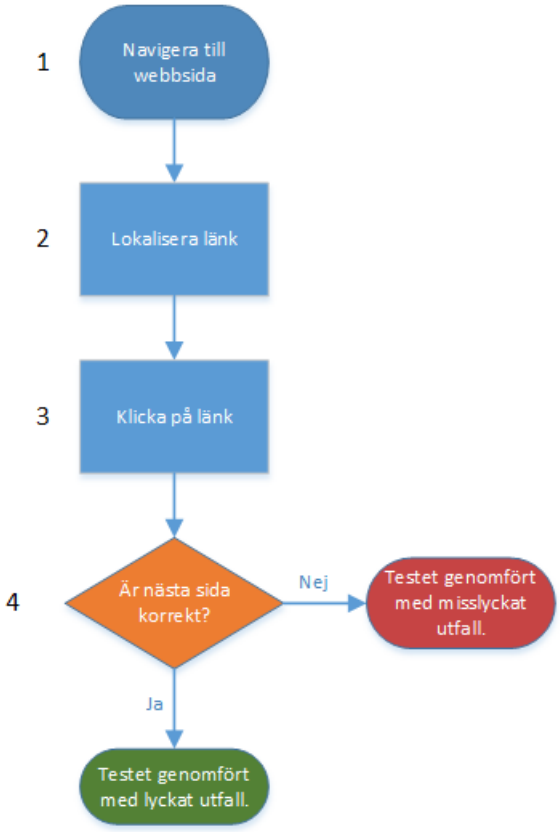
På detta sätt kan man enkelt skapa olika testfall för att simulera olika scenarion som en användare i vanliga fall skulle genomföra. Vill man exempelvis testa att en länk fungerar och att den skickar användaren till rätt plats kan ett test utformas enligt konceptet som illustreras i figur 10.

1. I det första steget gäller att man navigerar till den URL där länken som ska klickas är placerad.

2. Nästa steg är att lokalisera länken. Detta kan ske med olika metoder, ex. identifikation av CSS-klasser.

3. Klicka på länken.

4. Undersök huruvida den nya sidan är den sida man ville förflyttas till. Detta kan göras genom att söka efter element på sidan som man vet existerar om man hamnat rätt.



Figur 7 - Flödesschema över hur ett test skulle kunna utföras konceptuellt.

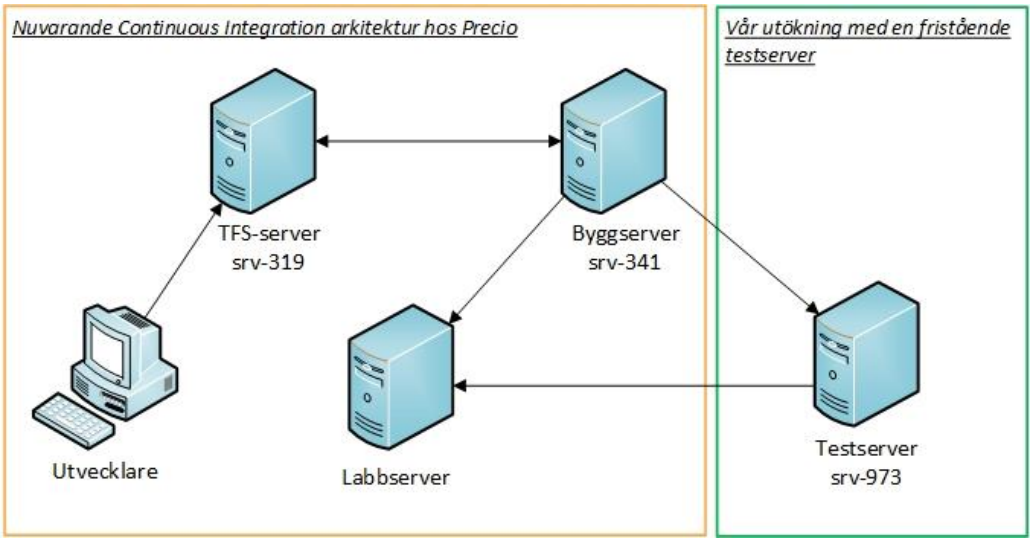
Som synes är det enkelt att konstruera olika typer av testfall genom att bara konceptuellt föreställa sig vad en användare skulle göra för att manuellt testa detta. Översatt till C#-kod blir ovanstående konceptuella test även det enkelt att följa och förstå, vilket visas med nedanstående kodsekvens.

```
public void TestNextPageLink()
{
    //1. Navigera till webbsida
    driver.Navigate().GoToUrl("http://www.precio.se");
    //2. Lokalisera länk med hjälp av CSS-klassen ”nextPage”
    IWebElement link = driver.FindElement(By.CssSelector("a.nextPage"));
    //3. Klicka på länken för att skickas vidare
    link.Click();
    //4. Avgör om länken förflyttat webbläsaren till korrekt sida
    Assert.AreEqual("Precio" ,driver.FindElement(By.ClassName("title")));
}
```

4.5 Integration av automatiserad testning

4.5.1 Integration av testserver i nuvarande arkitektur

För att automatisera testning introducerar vi ytterligare en server dedikerad enbart för utföra automatiserad GUI-testning i projekt som använder CI. Figur 11 visar den resulterande arkitekturen.



Figur 8 - Vår utökade implementation av CI där en testserver utför automatiserade tester.

Detta innebär att vår lösning blir en fristående komponent från Precios nuvarande arkitektur som enkelt kan implementeras efter behov. Servern finns alltid tillgänglig, men sammankopplingen är inte alltid där.

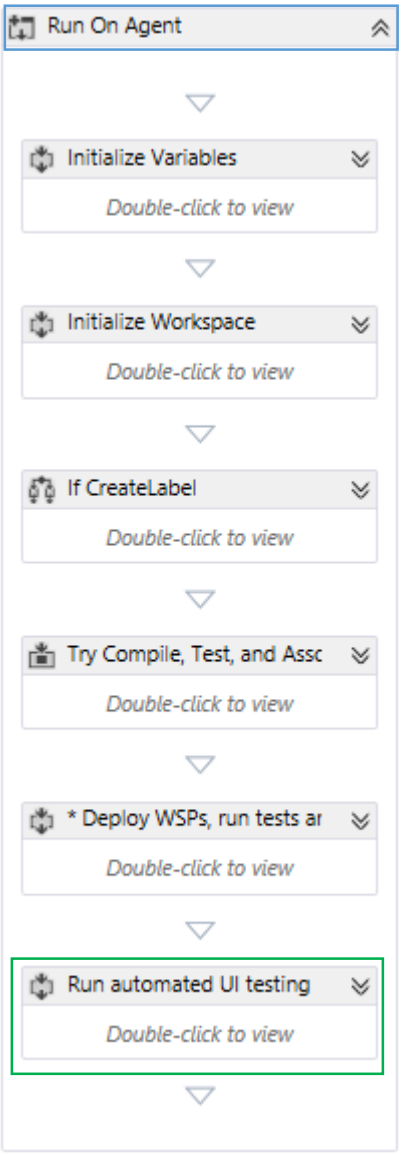
4.5.2 Automatiseringssekvens

I vår lösning inkorporeras testservern i CI genom att lägga till ett sista anropssteg efter att all kod har installerats på labservern. I detta steg anropas testservern som i sin tur utför ett antal definierade GUI-tester

mot labservern. Denna implementation förklaras mer detaljerat i nästa avsnitt.

4.6 Automatisering med hjälp av script

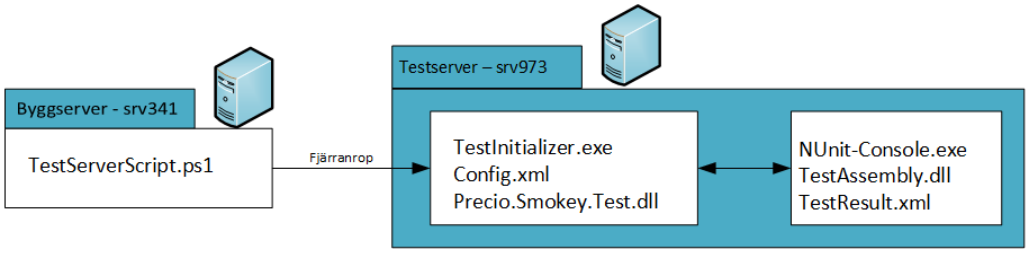
I och med att vi inte kunnat använda TFS och Microsoft rekommenderade teknik för implementation av automatisering, har vi varit tvungna att implementera en fristående lösning som fungerar på ett snarlikt sätt. Gemensamt för alla projekt hos Precio är att de byggs på en dedikerad byggserver. Beroende på vilken byggdefinition som används anropas ett script som installerar SharePoint-applikationen till en labserver. Byggdefinitionen består av en sekvens av steg att utföra där nästlade sekvenser eller steg kan placeras in. Det steg som vi genomfört ändringar i är där en sekvens av händelser utförs på bygg-agenten, som visas i figur 12. Längst ned i sekvensen har vi lagt till steget ”*Run automated UI testing*”, som har till uppgift att anropa det script som initierar testningen av GUI.



Figur 9 - Byggdefinitionens steg

Den sekvens av anrop som initieras presenteras i figur 13 följt av en förklaring av varje steg. Notera att detta inte är på en djup detaljnivå, utan endast ger en överblick av kommunikationsflödet mellan olika servrar och komponenter. Är man intresserad av att veta mer av hur scripten är utformade hänvisar vi till arkitekturdokumentet i bilaga A.

Det första steget är att från byggservern via TFS anropa ett lokalt skript, ”*TestServerScript.ps1*”, som finns i hemkatalogen på byggservern. Det här skriptet har endast till uppgift att etablera en anslutning till testservern där resterande komponenter är placerade. En av dessa komponenter är



Figur 10 - Kommunikationsflöde för initialisering av testning.

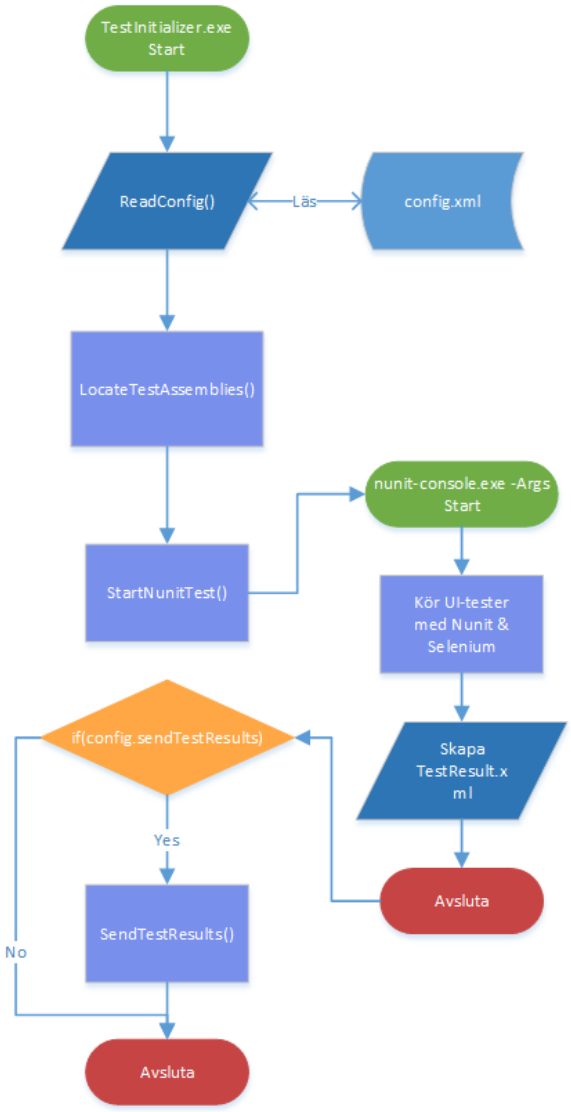
”*TestInitializer.exe*” som används för att initialisera testningsprocessen. De olika komponenterna förklaras mer detaljerat i efterföljande avsnitt.

4.6.1 TestServerScript.ps1

Det här är ett PowerShell-skript som är placerat på byggservern. I detta finns användarinformation för att kunna etablera en nätverkssession via PowerShell till testservern. På testservern startas i sin tur ”*TestInitializer.exe*” som sköter allt som har med testningen att göra.

4.6.2 TestInitializer.exe

Det här är en konsol-applikation som startar testerna. För att kunna genomföra tester måste varje projekt ha en egen katalog på testservern på platsen `C:\Test`. Katalogen innehåller konfigurationsfiler respektive de DLL-filer som utgör testerna som ska genomföras. Hur applikationen fungerar och vad för arbete den utför syns i flödesschemat i figur 14.



Figur 11 - Flödesschema över de uppgifter som TestInitializer.exe utför.

När TestInitializer.exe startas med anrop från byggservern läser den av en konfigurationsfil som innehåller information om vilken SharePoint-applikation som ska testas och vilka eventuella användaruppgifter som behövs för att få åtkomst till applikationen. I konfigurationsfilen finns även specificerat vilka URL:er som ska testas för standardfel och även vilka utvecklare som ska notifieras med ett testresultat via mail.

Nästa steg är att lokalisera och indexera de DLL-filer som finns i en särskild katalog ämnad för dessa. Där återfinns alltid minst en DLL-fil som utgör standardtesterna. Utvecklare kan placera egenskrivna och kompilerade tester i DLL-format i den här katalogen för att även få dem utförda och inkluderade i testresultatet.

När alla testassemblies är lokaliserade startas en ny process av NUnit som utgör det testramverk som genomför själva testerna.

Ramverket anropas med DLL-filerna som parametrar, och utför alla tester som de innehåller. En XML-fil innehållandes resultat från de utförda testerna skapas och processen stängs. Applikationen TestInitializer.exe återfår då kontrollen och skickar resultatet till de utvecklare som finns specificerade i konfigurationsfilen.

4.7 Paketering

Konstruktionen för automatiserad testning är paketerad utifrån de icke-funktionella krav som definierats i kravspecifikationen, se avsnitt 4.1.3. Dessa ställer krav på en enkel installation och användning, vilket resulterat i att alla nödvändiga filer paketerats tillsammans i en färdig katalogstruktur. Filerna och katalogerna har sedan placerats i en .ZIP-fil som går att packa upp till den katalog på testservern som tillhör det projekt som ska testas. Utöver detta måste en konfigurationsfil modifieras, samt att projektets byggdefinition måste utökas med en sekvens i Visual Studio. Beskrivningar om hur detta genomförs finns i arkitekturdokumentet, se bilaga A.

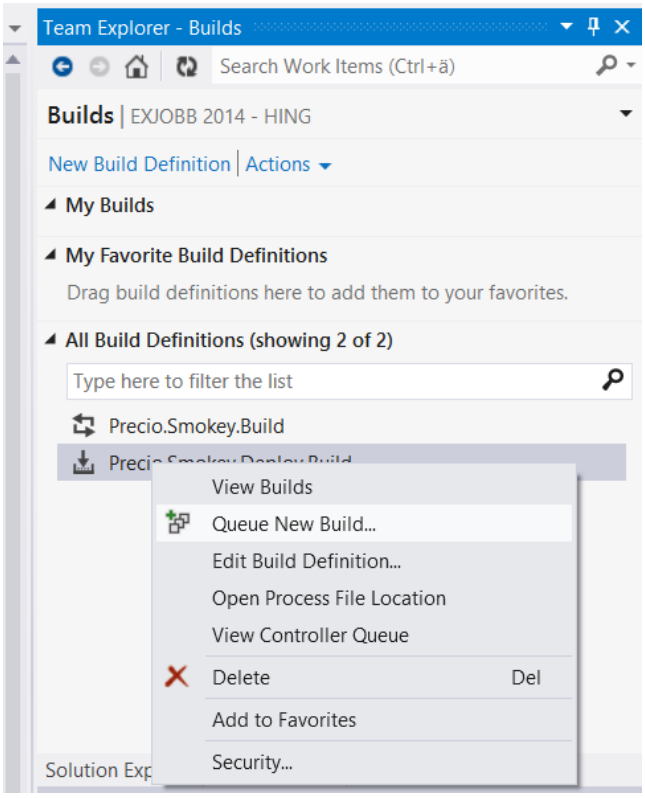
5 Resultat

5.1 Slutresultat

Resultatet av det här arbetet är en färdig implementation för automatiserad testning av GUI i en existerande och etablerad utvecklingsprocess. Utvecklare erbjuds nu möjlighet att använda implementationen för testning av sina projekt, där arbetsflödet beskrivs nedan.

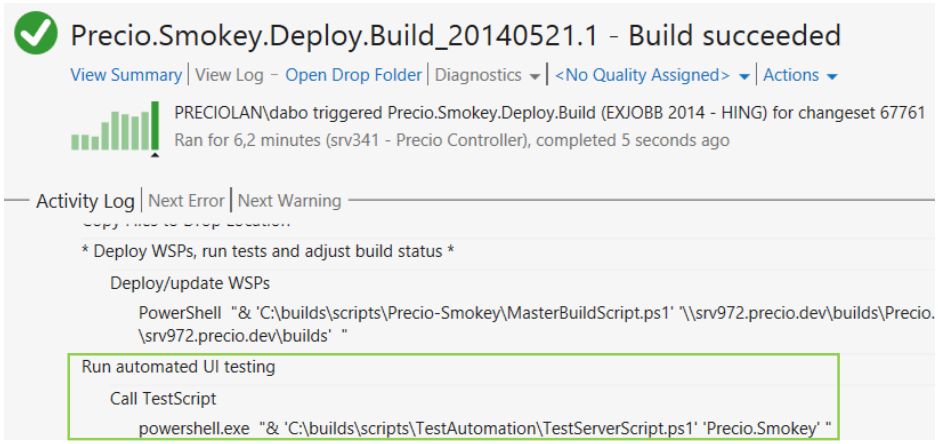
1. Utvecklare checkar in kod till TFS.

2. Utvecklare väljer att manuellt köa ett nytt bygge via ”Team Explorer” i Visual Studio. Se figur 15. Detta kan även köras per automatik vid schemalagda tider.



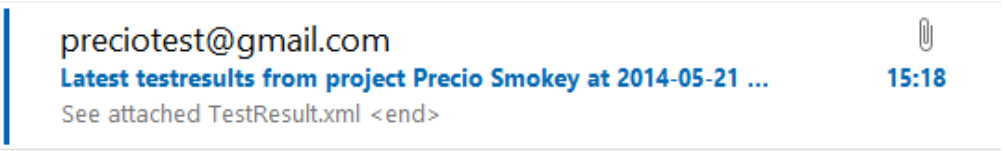
Figur 12 - Manuell köläggning av nytt bygge.

3. Den automatiserade testningen körs som en del av byggsekvensen efter att webapplikationen installerats. Utvecklaren kan följa detta i byggets aktivitetslogg, se figur 16.



Figur 13 - Aktivitetslogg för bygge.

4. När alla tester har genomförts skickas ett mail med testresultatet, se figur 17.



Figur 14 - Mailnotifikation med testresultat.

6 Diskussion

6.1 Resultat

6.1.1 Validitet

Sett till de ursprungliga målen kan vi konstatera att det arbete vi utfört uppfyller dem, och att det resulterat i ett svar på frågan huruvida en implementation av automatiserad testning av GUI är möjlig eller inte, vilket är att det är fullt möjligt. Vi har även lagt fram ett svar på hur en implementation skulle kunna tänkas se ut, både ur ett perspektiv av Microsoft-produkter, men också en helt fristående lösning.

Precio har efterfrågat ett system som gör det möjligt för utvecklare att testa SharePoint-applikationer efter standardiserade testfall. Tanken var att vi från första början skulle ha utformat ett bibliotek av tester som testar just efter fel som alltid kan uppstå oavsett vilket projekt som testas. Problematiken med detta har dock varit att tiden för vårt arbete inte räckt till i den mån man önskat, då vi under arbetets gång på grund av oförutsägbara händelser och tekniska upptäckter har behövt ändra vår kravspecifikation eller särskilda mindre teknikval. I det här fallet kanske vi inte når ända fram, i och med att det inte finns en färdig samling med tester att köra. Samtidigt uppfyller vi dock målet om att låta utvecklare skriva egna testfall som sedan utförs av den automatiserade testningen vilket gör att systemet fortfarande kan komma till användning i ett framtida skede.

6.1.2 Reliabilitet

Eftersom strukturen på Microsofts programvaror hos Precio inte uppfyllde versionskraven som ställs av Microsoft för att använda deras framtagna lösning för testautomatisering blev vår implementation en alternativ lösning. Det är en lösning skräddarsydd efter Precios utvecklingsprocess och som använder verktyg som utvecklare på företaget sedan tidigare är vana vid att arbeta med. Den enda faktiska skillnaden mellan vår implementation och Microsofts lösning är att testresultatet levereras till användaren via e-post medan användaren får feedback direkt i Visual Studio med Microsofts lösning. Självklart är det sistnämnda kraftfullare bakom kulisserna men för användaren blir detta den enda märkbara skillnaden.

Man kan se vår konstruktion som ett exempel på hur en alternativ lösning kan användas vid problem som ovan nämnda versionskrockar av

mjukvara. Nästan alla företag inom mjukvarubranschen drabbas av problemen av olika versioner av mjukvara, då olika interna projekt och kunder sätter olika krav. Det är för ett företag som Precio inte hållbart i längden, av ekonomiska skäl, att ständigt uppdatera mjukvara till senaste versioner.

Då vår lösning är fristående från den nuvarande programvarustrukturen hos Precio finns det dock inget krav på synkronisering mellan programvaruversioner. Detta innebär att när Precio i framtiden uppdaterar sin mjukvara, exempelvis Team Foundation Server, kommer vår implementation fortfarande fungera. Emellertid kan testmotorn, ”*TestInitializer.exe*”, behöva uppdateras om det kommer en ny version av NUnit som saknar bakåtkompabilitet eller har ny funktionalitet som utvecklare använder vid tester som skrivs. Detta är däremot ingen avancerad åtgärd.

Uppdateringar är något som sköts av programdistributören vid användning av en kommersiell produkt, vilket kan vara en fördel framför vår implementation som inte har någon löpande vidareutveckling. Om så önskas måste Precio själva ta ansvar för detta.

En sista frågeställning är huruvida det utöver Microsofts rekommenderade implementation finns ytterligare bättre alternativ, men det har inte varit något aktuellt att jämföra då Microsoft är den dominerande aktören för automatisering i utveckling av just .NET.

6.2 Metod

Vi har arbetat efter en agil arbetsmetodik, mycket just för att det är det tillvägagångssätt som är etablerat och i bruk hos Precio när man arbetar med projekt och utveckling av ny mjukvara. Vi anser att det har fungerat bra, framförallt med åtanke på att vi under arbetets gång tvingats ta ställning till förändringar i kravspecifikationen, vilket agila arbetsmetoder är anpassat för att hantera.

På det sätt som Precio arbetar utför man morgonmöten inom varje projekt, där varje deltagare redogör för vad de gjort och vad de ska göra. Detta har även vi gjort i form av morgonmöten med våra handledare där vi kunna reflektera över vad som gått bra och dåligt, samtidigt som vi kunnat få den hjälp vi behöver. Det har fungerat bra, och har ofta resulterat i att vi aldrig suttit med obesvarade frågor under dagen, utan istället har kunnat fokusera på sådant vi haft kunskaper till. Vi har även

under dessa morgonmöten sett över innehållet i backlogen, vilket är den egentliga agendan under dessa sammanträden. Backlogen har varit ett bra verktyg för att hålla koll på arbetets status, även om vi kanske borde varit mer aktiva med att uppdatera den, exempelvis med hur mycket tid som kvarstår för särskilda uppgifter.

Backlogen har även använts för att koppla olika incheckningar av kod till särskilda uppgifter. Även här kunde vi ha varit mer aktiva och checkat in kod oftare. Det hände att vi var tvungna att se över vilken version som var den aktuella, och sedan checka in kod i olika ordningar.

Överlag har metoden för arbetet fungerat mycket väl, och vi har fått en större förståelse för hur agil utveckling kan genomföras i större skala på ett framgångsrikt sätt.

Efter varje sprint har vi tillsammans med våra handledare haft ett utvärderingsmöte. Detta har möjliggjort både för oss som utfört projektet och för våra handledare att framföra vad som har fungerat bra och vad som har fungerat dåligt. På så sätt har vi undvikit att göra om samma misstag under varje sprint.

6.3 Hållbar utveckling

Inom mjukvaruutveckling, precis som i alla tänkbara områden, innebär hållbar utveckling att ta vara på de resurser som finns att tillgå, och att inte utnyttja dem på ett sätt där de framstår som outtömliga. Att vidareutveckla och underhålla mjukvarusystem är en tidskrävande process, och därmed en viktig aspekt i konceptet för hållbar utveckling. Att ta detta i beaktning redan under utveckling bäddar för att det i framtiden ska vara enklare och mindre tidskrävande att utföra detta.

Ser man till verktyg för att underlätta hållbar utveckling så är automatiserad testning ett bra och viktigt sådant, vilket knyter an till vårt projekt på ett direkt sätt. Dels erbjuder automatiserad testning snabbare utveckling för företag då ingen manuell testning måste genomföras utan allt istället sker per automatik under utvecklingens gång. Detta resulterar i att fel upptäcks och åtgärdas tidigt, vilket förhindrar oplanerade förseningar som kan kosta mycket pengar och resurser.

Under det här projektets gång har vi mer eller mindre omedvetet tillämpat en form av hållbar utveckling. Dels har vi använt oss av versionshantering som måste ses som ett verktyg för hållbar utveckling.

Versionshanteringen har tillåtit oss att under arbetets gång arbeta parallellt på flera versioner av kod, för att sedan slå dem samman till en version. Samtidigt har detta möjliggjort för enkel ”code review” av andra utvecklares kod, vilket kan vara en bidragande faktor till att skära ned på kostnader.

Den konstruktion som är resultatet av vårt arbete är av en sådan utformning att det är enkelt att vidareutveckla den för utökad funktionalitet. Man kan här göra en referens till SOLID-akronymen inom objekt-orienterad design, där O (Open/closed principle) handlar om hur mjukvaran ska vara öppen för utvidgning men stängd för modifikation. Även här kommer versionshantering in i bilden då det möjliggör för andra utvecklare att snabbt ansluta sig till utvecklingen och bidra med egen kod.

Tanken med resultatet av vårt projekt är i grund och botten att det ska fungera som ett redskap under en längre period, och att det ska vara något som Precio kan använda även efter att en lång tid har passerat, något som vi tagit i beaktning bland annat när det gäller versioner av testningsverktyg.

6.4 Sociala och etiska aspekter

Vårt projekt är genomfört enlig en agil arbetsmetod, vilket kan föra särskilda etiska tankar och aspekter på tal. Agil utveckling är, till skillnad från många andra utvecklingsmetoder, värdedriven. Detta innebär att utvecklingen skapar värde hos kunden på ett kontinuerligt sätt, både i form av tillhandahållning av en efterfrågad produkt, men även i det sätt som interaktionen mellan kund och utvecklare sker.

Vår morala skyldighet som utvecklare av mjukvara handlar just i förstahand om att skapa värde hos kunden. Den agila utvecklingsmetoden öppnar upp för detta utan någon form av slöseri av resurser varken för utvecklare eller kund. Samtidigt får inte detta ske på bekostnad av involverade parters välmående, exempelvis på grund av en dåligt genomförd tidsuppskattning av ett projekt, vilket leder till hög stress för utvecklarna och kvalitetskompromisser i slutprodukten.

Agil utveckling tillämpar även en form av ärlighet genom hela projektet. Det är möjligt att för vem som helst, när som helst se vad som är färdigställt, eller vad som är försenat. Ärligheten bör även tillämpas vid

nya affärer, då man antingen genomför, eller inte genomför. Det existerar aldrig ett scenario då man endast **försöker** utveckla något.

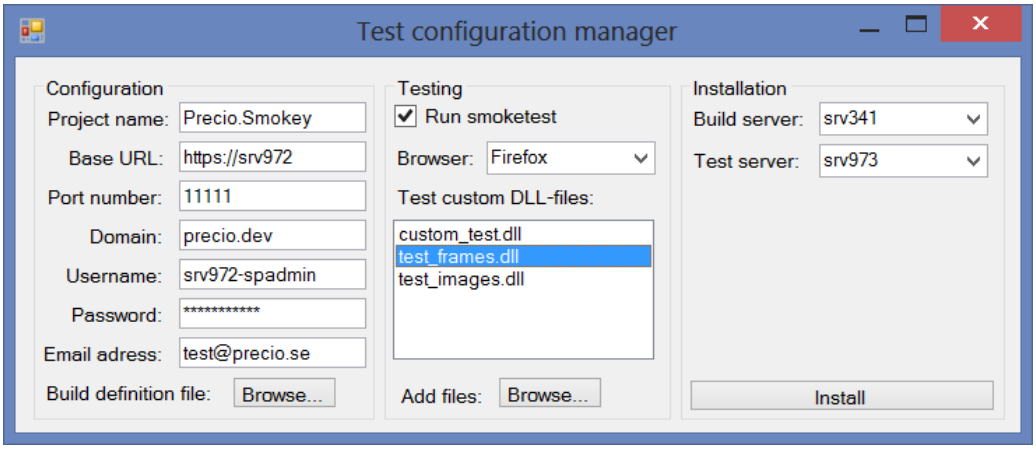
Kent Beck, skaparen av utvecklingsmetoden Extreme Programming (XP), sammanfattar det hela kring etik inom agil utveckling väldigt väl i boken ”Extreme Programming Explained: Embrace Change” [16]. Genom att lägga fokus på värdena i god kommunikation, återkoppling, respekt för varandra och att hålla saker enkelt kan man uppnå en bra harmoni och hög etisk nivå inom mjukvaruutveckling.

7 Framtida funktionalitet

Det färdiga resultatet av vår konstruktion kan ses som en ”proof of concept”, och tillsammans med Precio har vi kommit fram till att mycket utökning av funktionalitet är möjlig för att göra ett mer fulländat testningssystem. Nedan listas de mest diskuterade idéerna över vad framtida arbete kan resultera i.

7.1 Förenklad installation och konfiguration

Resultatet av arbetet som det ser ut i dagsläget kräver att man installerar och konfigurerar den automatiserade testningen manuellt för varje nytt projekt det ska tillämpas i. Detta skulle gå att förenkla genom att låta konfigurationen skötas via ett gränssnitt, exempelvis via en fristående applikation där användaren fyller i den information som behövs. Ett förslag på hur det skulle kunna se ut visas i figur 18.



Figur 15 - Exempel på vidareutveckling för konfiguration av automatiserad testning.

Det som applikationen skulle ersätta är steget att skapa en ny katalog på testservern med projektets namn, och sedan kopiera in nödvändiga filer till denna. Applikationen skulle även kunna modifiera den byggdefinitions-fil som användaren i dagsläget måste lägga till ett steg i. Till sist skulle även en konfigurationsfil, config.xml, kunna skapas dynamiskt av applikationen utifrån ifylld information.

Alla dessa förslag är något som skulle kunna utvecklas av någon som förstår hur konfigurationen måste ske.

7.2 Presentation av testresultat

Just nu presenteras testresultatet i en XML fil som skickas via e-post till specificerade mottagare. Innehållet i XML filen är relativt omfattande vilket gör den besvärlig att avläsa efter exekvering av ett stort antal testfall. Ett bättre presentationsformat vore därför lämpligt att implementera.

7.3 Koppling av konfiguration till specifika byggen

I dagsläget existerar inte någon koppling mellan olika byggen och de test som genomförs via konfigurationen. En framtida idé är att sammankoppla konfigurationen med versionshanteringen, för att man vid senare skeden enklare ska kunna följa vilken typ av testning som tidigare genomförts och även för att enklare kunna konfigurera testningen direkt från utvecklingsverktyget Visual Studio.

7.4 Detaljerad konfiguration

En efterfrågan hos Precio har varit att kunna specificera olika subadresser till SharePoint-applikationer som test sedan utförs gentemot. Detta existerar i nuvarande version, men skulle kunna vidareutvecklas för att tillåta specificering av vilka tester som genomförs mot vilka adresser. I grund och botten handlar detta om att definiera en bättre modell för konfiguration av testerna.

8 Källförteckning

[1] S. Fox, C. Johnson och D. Follette, ”Beginning SharePoint® 2013 Development,” Indianapolis, John Wiley & Sons, Inc., 2013, pp. 3-5.

[2] B. Boehm, ”Software risk management: principles and practices,” *Software, IEEE,* vol. VIII, nr 1, pp. 32-41, 1991.

[3] G. Paré, C. Sicotte, M. Jaana och D. Girouard, ”Prioritizing Clinical Information System Project Risk Factors: A Delphi Study,” *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual,* p. 242, 2008.

[4] W.-M. Han och S.-J. Huang, ”An empirical analysis of risk components and performance on software projects,” *Journal of Systems and Software,* pp. 42-50, 2007.

[5] O. Taipale, J. Kasurinen, K. Karhu och K. Smolander, ”Trade-off between automated and manual software testing,” *International Journal of System Assurance Engineering and Management,* vol. II, pp. 114-115, 2011.

[6] P. Gregory Tassey, ”The Economic Impactsof Inadequate Infrastructure for Software Testing,” RTI Health, Social, and Economics Research, Triangle Park, NC, 2002.

[7] S. Berner, R. Weber och R. K. Keller, ”Observations and lessons learned from automated testing,” i *27th international conference on Software engineering*, New York, NY, USA, 2005.

[8] I. Sommerville, ”Software Testing,” i *Software Engineering Ninth Edition*, Pearson, 2011, pp. 210-219.

[9] C. Poole, J. Terr och S. Busoli, ”NUnit 2.0,” [Online]. Available: http://www.nunit.org/. [Använd 14 Maj 2014].

[10] Microsoft, ”MSTest.exe Command-Line Options,” [Online]. Available: http://msdn.microsoft.com/en-us/library/ms182489.aspx. [Använd 15 Maj 2014].

[11] VersionOne, ”7th Annual State of Agile Development Survey,” Versionone.com, http://www.versionone.com/pdf/7th-Annual-State-of-Agile-Development-Survey.pdf (Hämtad 2014-04-24), 2013.

[12] D. Goodman och M. Elbaz, ""It's not the pants, it's the people in the pants" Learnings from The Gap Agile Transformation – What Worked, How We Did it, and What Still Puzzles Us," i *Agile 2008 Conference*, 2008.

[13] Microsoft, "Upgrading Test Controllers from Visual Studio 2010," Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/library/hh707968(v=vs.110).aspx. [Använd 13 Maj 2014].

[14] "Scripting with Windows PowerShell," Microsoft, [Online]. Available: http://technet.microsoft.com/en-us/library/bb978526.aspx. [Använd 3 Juni 2014].

[15] S. Stewart och D. Burns, "WebDriver API," W3C, 15 May 2014. [Online]. Available: https://dvcs.w3.org/hg/webdriver/raw-file/default/webdriver-spec.html. [Använd 15 Maj 2014].

[16] K. B. w. C. Andres, i *Extreme Programming Explained: Embrace Change*, 2005.

Bilaga A: Arkitekturdokument

Arkitekturdokument för automatiserad testning av GUI

Architectual document

Daniel Borg
Anders Elfström

Innehållsförteckning

1 Introduktion

1.1 Syfte

Detta dokument beskriver arkitektur och implementation av automatiserad testning av gränssnitt i SharePoint hos företaget Precio Systemutveckling AB. Dokumentet behandlar även hur den utvecklade lösningen driftsätts och underhålls.

1.2 Målgrupp

Dokumentet är ämnat för personer med teknisk kunskap inom IT och utveckling. Läsaren förväntas känna till hur ”Continuous Integration” används och fungerar i företagsmiljö. Mer om detta i projektrapporten ”Automatiserad Testning av användargränssnitt i SharePoint”.

1.3 Mål

Med hjälp av detta dokument ska en extern utvecklare kunna använda och underhålla lösningen för automatiserad testning av användargränssnitt i SharePoint.

1.4 Relaterade dokument

Anders Elfström och Daniel Borg, ”Automatiserad Testning av användargränssnitt i SharePoint”.

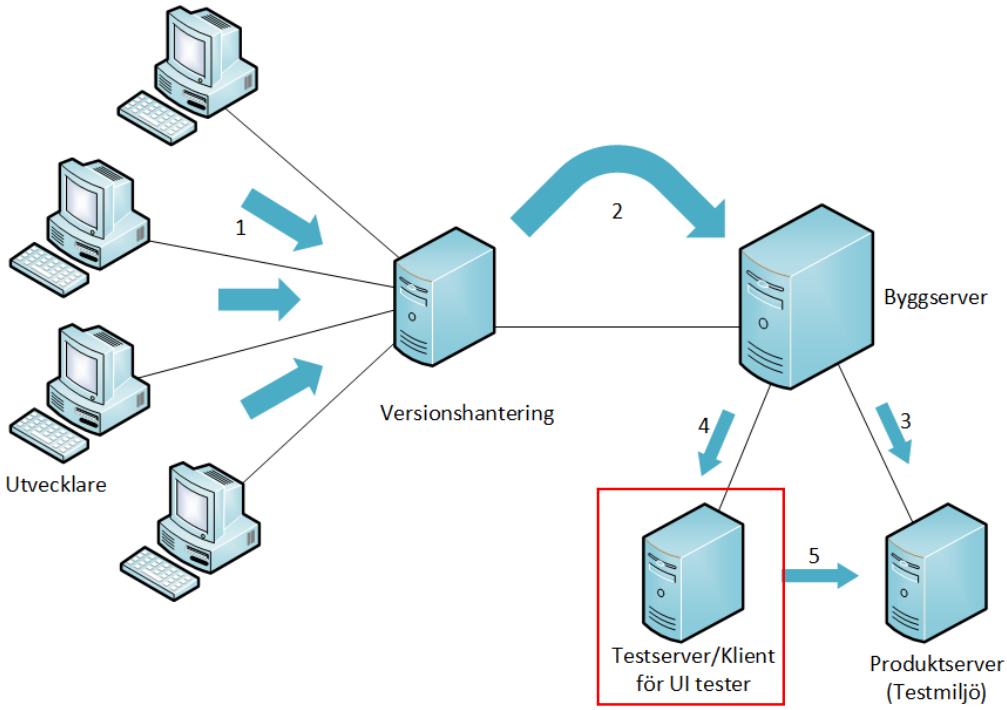
2 Systemets uppgift

2.1 Systemets sammanhang & kontext

Precio, företaget där arbetet är genomfört, bygger många komplexa webbaserade system, där flertalet av dessa har ett lika komplext användargränssnitt. Att testa hela användargränssnittet manuellt är oftast inte ett kostnadseffektivt alternativ. Genom att introducera automatiserad testning av användargränssnitt, framförallt i samband med processtillämpningen ”Continuous Integration”, strävar Precio mot att höja kvaliteten samt reducera kostnader på samma gång.

Systemet har till uppgift att lösa problemet med avsaknaden av automatiserad testning i en existerande implementation av continuous integration. Detta med exklusivt avseende på en utvecklingsmiljö baserad på Microsoft Team Foundation Server och utveckling inom Microsoft .NET-technology.

Systemet kommunicerar direkt med Microsoft Team Foundation Server i samband med kompilering och bygge av incheckad kod från enskilda utvecklare. Testningen aktiveras med anrop från en server dedikerad till bygge, efter att ett sådant genomförts och installation av den applikation som ska testas är skett.



Figur 16 - En högnivå-översikt över systemets integration i existerande arkitektur.

I figur 19 ovan syns olika parters deltagande i processen, där systemet integreras på enskild server och anropas i steg 4.

2.2 Systemets gränssnitt

Systemet erbjuder endast en tjänst i form av ansvaret att tillhandahålla automatiserad testning av fördefinierade testfall alternativt skräddarsydda testfall utformade av enskilda utvecklare. Tjänsten är aktiverad att utföras då inledande initieringsscript finns installerat på byggserver i berört projekt/applikation som ska testas.

2.3 Icke-funktionella krav

Systemet har icke-funktionella krav som faller in på den kvalitetsaspekt det bedöms inom i form av underhåll, drift, användarvänlighet, portabilitet etc. Nedan listas de två krav som tagits fram i relation till detta, tillsammans med uppdragsgivare och utvecklare.

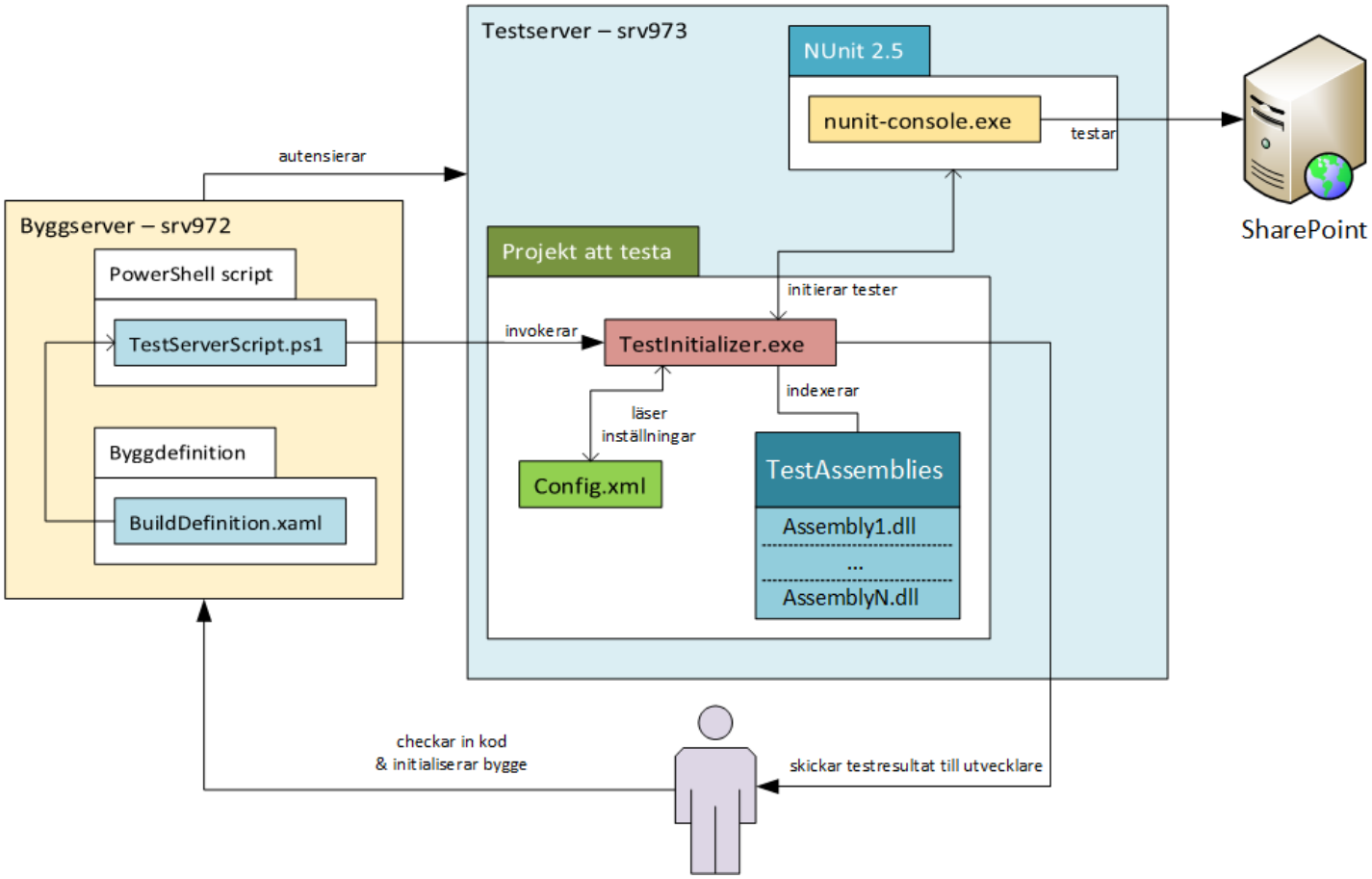
1. Systemets implementation och tjänst ska vara enkel att installera på godtyckliga projekt som faller inom ramen för användningsområdet.

2. Det ska vara enkelt för en användare att använda systemets implementation för att låta det utföra skräddarsydda testfall som faller inom kraven för utformning av dessa.

3 Systemets struktur

3.1 Övergripande strukturvy

Systemets komponenter är distribuerade över två servrar där byggservern måste konfigureras för kommunikation med testservern. Detta sker i en byggdefinition som är unik och individuell för varje enskilt bygge av applikationer. På byggservern finns även ett skript som anropas för att initiera testningsprocessen. Övriga komponenter är placerade på en dedikerad testserver som anropas från byggservern. Figur 20 illustrerar de olika komponenterna i existerande struktur.



Figur 17 - Struktur över olika komponenter i systemet och dess placering.

3.2 Komponenter

Nedan beskrivs de individuella komponent från avsnitt 3.1 mer detaljerat.

Komponent	Byggdefinition
Placering	En .xaml fil som specifieras i projektetes ”Build Process Template” på Team Foundation Server.
Samarbetar med	*TestServerScript.ps1*
Ansvar	Invokera *TestServerScript.ps1* som ett sista steg i continuous integration.
Anteckningar	Förklaras mer detaljerat separat(?)
Problem	

Komponent	TestServerScript.ps1
Placering	srv341\Scripts\TestAutomation
Samarbetar med	*TestInitializer.exe*
Ansvar	Anropar *TestInitializer.exe* för att initiera och köra tester.
Anteckningar	Tar emot en parameter som talar om namnet på projektkatalogen där *TestInitializer.exe* är lokaliserad.
Problem	

Komponent	TestInitializer.exe
Placering	srv973\Test\{Projektkatalog}\TestRunner
Samarbetar med	*NUnit-console.exe, Config.xml, TestAssemblies, TestResult.xml*
Ansvar	Initierar och kör tester som ligger i *TestAssemblies* katalogen med hjälp av *NUnit-console.exe* baserat på konfigurationen i *Config.xml*. Testresultatet skrivs till *TestResult.xml*. Om angivet i *Config.xml* kan resultatet mailas till specificerade användare.
Anteckningar	Se figur 3 och 4 för diagramöversikt.
Problem	- Använder just nu en Gmail-adress för att skicka ut testresultat via mail.

Komponent	NUnit-console.exe
Placering	srv973\Program Files (x86)\NUnit 2.6.3\bin
Samarbetar med	
Ansvar	Det ramverk som används för att exekvera GUI tester.
Anteckningar	
Problem	

Komponent	Config.xml
Placering	srv973\Test\{Projektkatalog}\Config
Samarbetar med	*TestInitializer.exe*
Ansvar	Specificerar den information behövs för att testa en SharePoint site.
Anteckningar	Beskriv vad som måste finnas med i denna fil
Problem	

Komponent	TestAssemblies
Placering	srv973\Test\{Projektkatalog}\TestAssemblies
Samarbetar med	*TestInitializer.exe*.
Ansvar	I denna katalog placeras de tester som ska köras av *TestInitializer.exe*. Utvecklare kan skriva egna tester enligt den syntax som Selenium och NUnit använder och sedan placera de dll filer som genereras efter att ett projekt har kompilerats/byggts i denna katalog. *TestInitializer.exe* kommer då upptäcka dessa dll filer och köra dem.
Anteckningar	
Problem	

Komponent	TestResult.xml
Placering	srv973\Test\{Projektkatalog}\TestRunner
Samarbetar med	*NUnit-console.exe*
Ansvar	Denna fil genereras av *NUnit-console.exe* efter att alla tester har exekverats och innehåller testresultaten.
Anteckningar	
Problem	

4 Interaktion med systemet

4.1 Integration av egenskrivna testfall

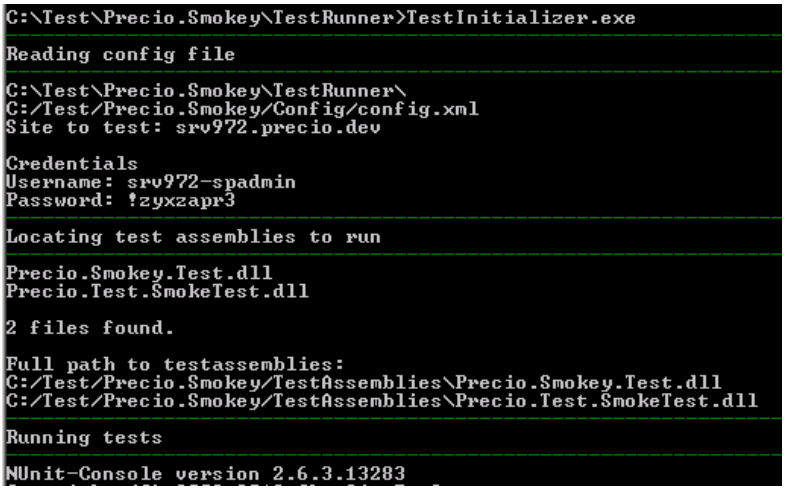
Följande avsnitt beskriver hur en utvecklare kan skriva egna gränssnittstester och sedan integrera dem i den automatiserade testprocessen. Figur 3 illustrerar flödet för att skapa ett eget GUI test och integrera det som ett automatiserat test.

Figur 18 - Flöde för att integrera egna test i den automatiserade lösningen.

Skapa ett helt nytt projekt i Visual Studio och lägg till referenser för NUnit och Selenium. Skriv dina test enligt den syntax som NUnit och Selenium använder.

När testerna är skrivna, bygg projektet och navigera till den katalog där projektets DLL-filer ligger, vilket normalt är under *"{Ditt C#-projekt}\bin\Debug"*. Kopiera .dll filen och placera i projektets TestAssemblies katalog på testservern, *"srv973\Test\{Projektkatalog}\TestAssemblies"*.

Sista steget är att manuellt köra *"TestInitializer.exe"* för att säkerställa att testerna integreras på ett korrekt sätt med den automatiserade lösningen. Figur 22 visar ett exempel på hur *"TestInitializer.exe"* körs manuellt för ett projekt och de .dll-filer som hittas som ska köras som GUI test. Skulle något fel uppstå kommer detta skrivas ut i konsolapplikationen.

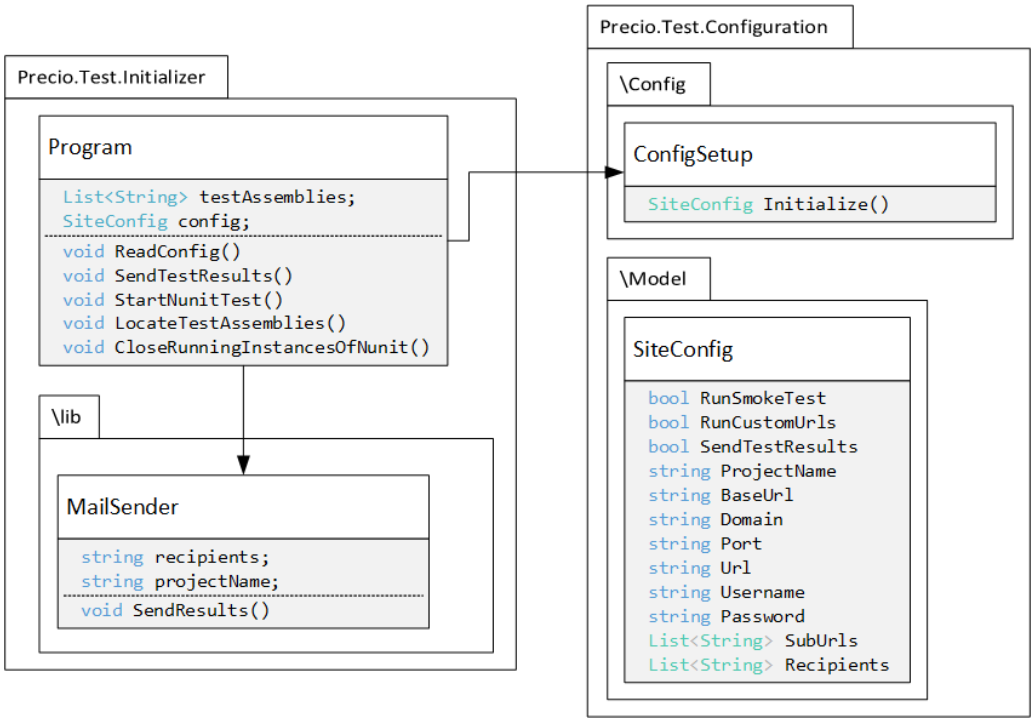


Figur 19 - En manuell körning av TestInitialzer.exe från kommandotolken för tester i projektet ”Precio.Smokey”.

Om testerna körs som önskat i konsollapplikationen fungerar integreringen av det egenskrivna testet och är nu automatiserat.

5 Övriga vyer

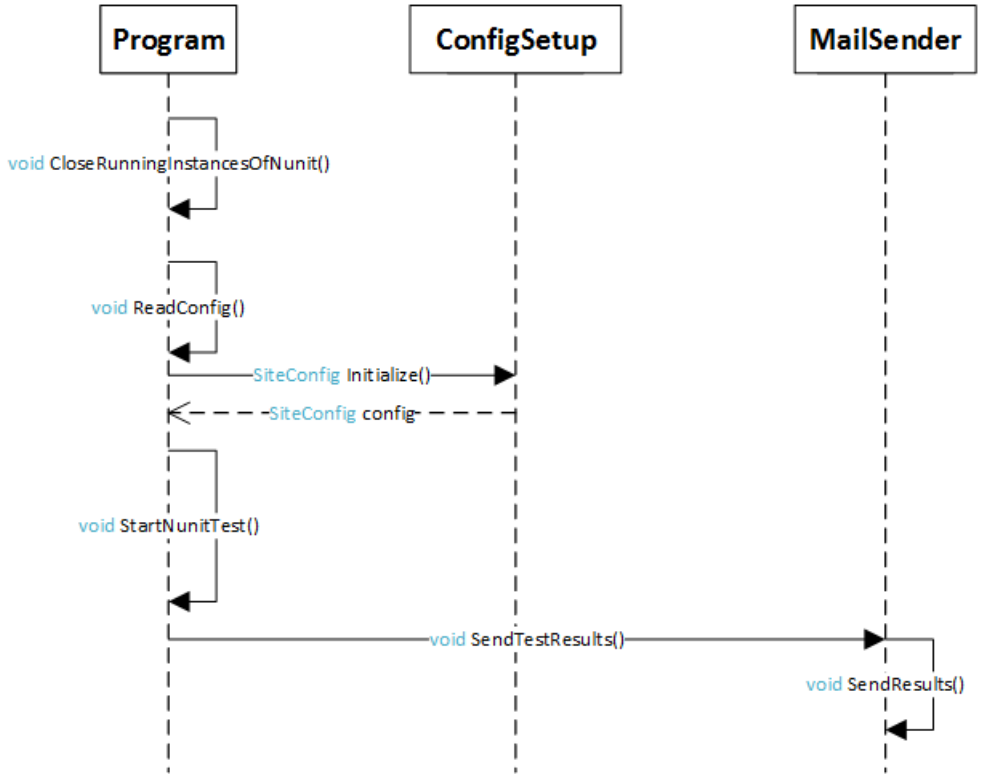
5.1 Processvy



Figur 20 - UML-diagram för TestInitializer. Programmet utgörs av två klassbibliotek som interagerar med varandra och är uppdelade i olika paket för logisk avgränsning.

Ovan illustreras implementation av mjukvara i form av ett UML-diagram för den logiska komponenten TestInitializer.exe. Applikationen är uppdelat i två klassbibliotek som interagerar med varandra och där klassen ”Program” utgör ingångspunkten för exekvering. TestInitializer.exe anropas via skript i MicroSoft PowerShell från dedikerad byggserver, se avsnitt 5.2 för detaljer.

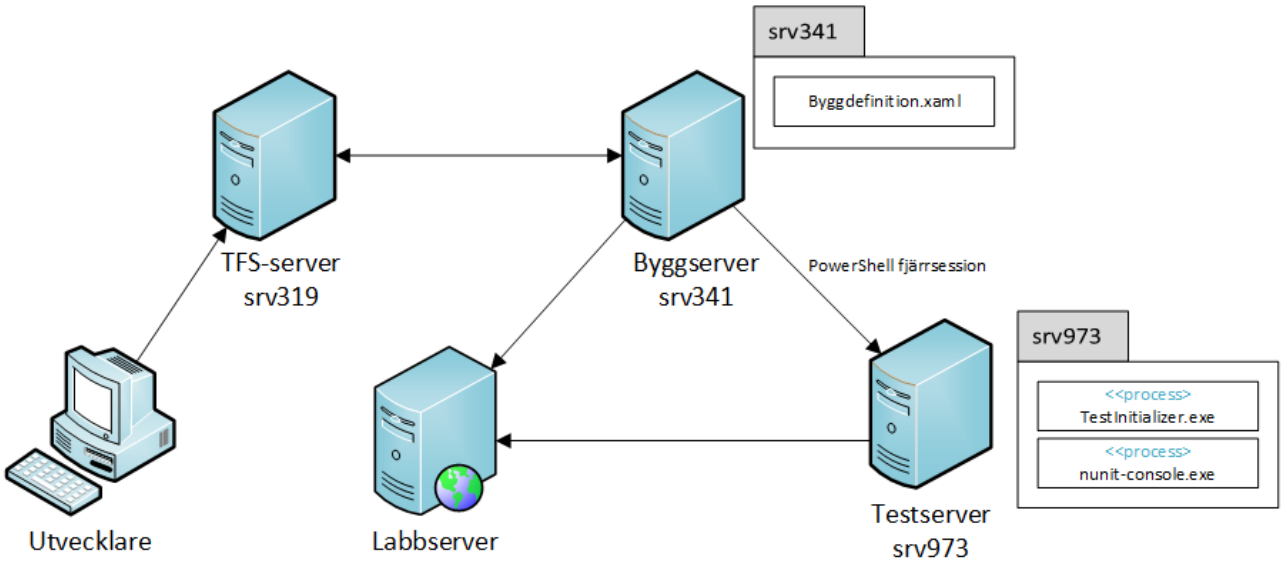
Anropskedjan internt för TestInitializer.exe återges i figur 24 i form av ett sekvensdiagram.



Figur 21 - Sekvensdiagram över intern kommunikation i TestInitializer.exe.

Sekvensvyn ovan illustrerar de anrop som sker internt i TestInitializer.exe. Detta förutsatt att programmet exekverats lokalt via ett fjärranrop från dedikerad byggserver.

5.2 Hårdvaruvy



Figur 22 - Översikt av systemimplementation på hårdvara.

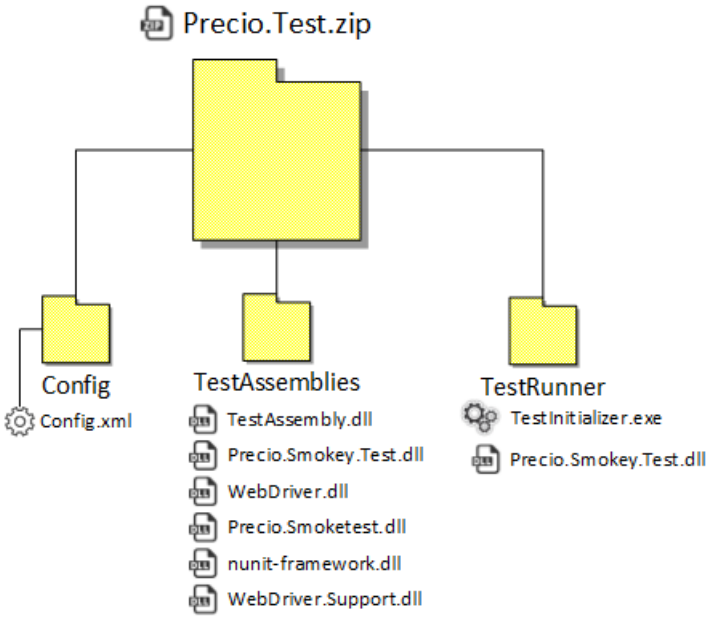
Figur 25 illustrerar hur olika komponenter är distribuerade på hårdvara i arkitekturen.

6 Paketering & Installation

6.1 Paketering

Systemet är paketerat i en .ZIP-fil där allt är samlat för enkel installation. Filen består av tre kataloger och en instruktionsfil för hur byggdefinitionen ska konfigureras för anrop från byggserver till testserver.

Figur 26 visar innehållet av .ZIP-filen som den är paketerad och hur katalogstrukturen även kommer att vara efter installation.



Figur 23 - Översikt av hur systemets olika delar är paketerade inför installation.

6.2 Installation

6.2.1 Installationskrav

För att installera systemet krävs det åtkomst till den byggdefinitionsfil som projektet som ska testas hör till, och till testservern för att kunna skapa en projekttillhörande katalog. Testservern är sedan tidigare förinställd för att acceptera anrop och skrivning till objekt i katalogen `\\srv973.precio.dev\Test` och vidare konfiguration för detta före installation ska inte vara nödvändigt.

Det är även förutsatt att projektet som ska testas har en färdigkonfigurerad implementation av continuous integration aktiverad där slutsteget är installation till labmiljö eller produktionsmiljö.

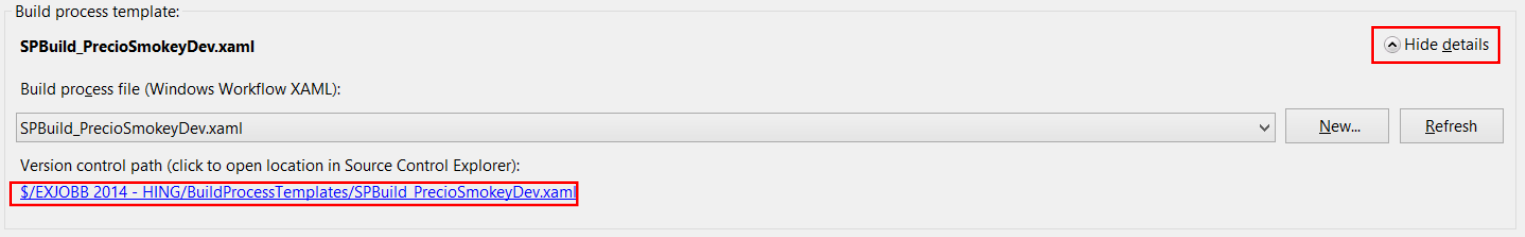
6.2.2 Konfiguration av byggdefinition

Byggdefinitionen för ett projekt baserat på continuous integration med Microsoft Team Foundation Server, är en sekvens av processer eller händelser som utlöser varandra. Slutsteget i en byggdefinition för ett SharePoint-projekt är en installation till en testmiljö. Följandes detta ska byggdefinitionen efter lyckad installation utföra tester.

För att lägga till en process i den övergripande byggsekvensen gör man följande.

1. Lokalisera korrekt byggdefinition i Visual Studio via ”Team Explorer” -> ”Builds” -> ”Edit Build Definition”.

2. Navigera till fliken ”Process”. Under ”Build Process Template”, utvidga rutan genom att klicka på pilen och sedan på den blåmarkerade .xaml-filens sökväg.



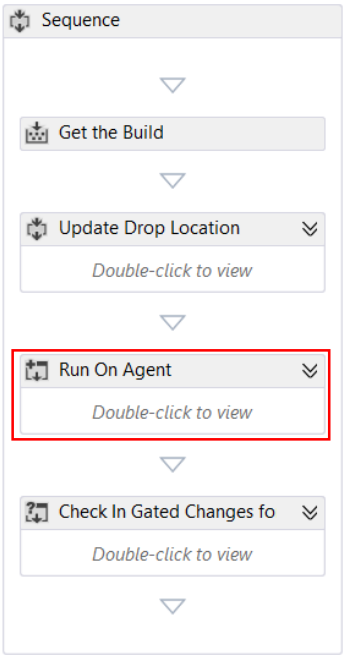
Figur 25 - Lokalisering av byggdefinition.

3. När diagramvyn för byggdefinitionen öppnats, öppna AgentScope-komponenten som heter ”Run On Agent”.

4. Efter det sista steget i ”Run On Agent”, lägg till en ny sekv från Toolbox av typen **`System.Activities.Statements.Sequence`**. Döp sekvense någon lämpligt, ex. ”Run Automated Testing”.

5. I den nya sekvensen, lägg till från Toolbox ett objekt av typ **`Microsoft.TeamFoundation.Build.Workflow.Activities.Process`**. Döp objektet till något lämpligt, ex. ”Call TestSc

6. Expandera det nyskapade objektet. Under ”Handle Standa Output”, dra från Toolbox ett objekt av typen **`Microsoft.TeamFoundation.Build.Workflow.Activities.BuildMessage`** till området där det står *”Drop activity here”*.



Figur 24 - Steg i byggdefinition att lägga ändra i.

7. Under ”Handle Error Output”, dra från Toolbox en ny sek av samma typ som i steg 4. Lägg sedan till i sekvensen ett av typen **`Microsoft.TeamFoundation.Build.Workflow.Activities.WriteBuildError`**

8. Markera objektet ”Call TestScript” (steg 5). Under egenskaper, lokalisera attributen ”Arguments”, ”FileName”, och ”WorkingDirectory”.

9. Under ”Arguments”, fyll i följande:
```
String.Format(" ""&
'C:\builds\scripts\TestAutomation\TestServerScript.ps1' '{0}'
"" ", "Precio.Smokey")
```
Ersätt exemplet ”Precio.Smokey” med ett namn för ditt projekt. **OBS!** Detta namn är även det namn du måste använda för katalogen på testservern för att det ska fungera.

10. Under ”FileName”, fyll i följande:
```
"powershell.exe"
```

11. Under ”Working Directory”, fyll i följande:
```
"C:\builds\Scripts\TestAutomation"
```

6.2.3 Installation på testserver

När konfigurationen av byggdefinitionen tillhörandes det projekt som ska testas måste testservern konfigureras för att kunna genomföra testning.

1. Navigera till `\\srv973.precio.dev\Test` med Windows Explorer.

2. Skapa en ny katalog med **samma** namn som du specificerat i steg 9 i avsnitt 6.2.2, ex. ”Precio.Smokey”.

3. I katalogen `\\srv973.precio.dev\Test` finns .ZIP-filen Precio.Test.zip. Packa upp innehållet till din nyskapade katalog.

6.2.4 Konfiguration av config.xml

I katalogen ”Config” finns en .xml-fil med namnet config.xml. Denna fil måste innehålla information om den applikation som ska testa. Nedanstående fält är vad som är krav på att fylla i:

```
<baseurl>http://srv972.precio.dev</baseurl>
<port>11111</port>
<domain>precio.dev</domain>
<username>srv972-spadmin</username>
<password>password</password>
```

`baseurl` – URL till webbapplikation som ska testas, se exempel ovan.
`username` – Användarnamn för att logga in och granska webbapplikationen.
`password` – Lösenord för åtkomst av webbapplikation.

6.2.5 Inkludering av tester

För att tester ska köras måste katalogen ”TestAssemblies” innehålla .dll-filer med testfall. Från början inkluderas en .dll-fil med namnet

"Precio.Test.SmokeTest.dll" som innehåller sökning efter standardiserade SharePoint-fel. För att inkludera egenskrivna tester, se avsnitt 4.1.1. Notera att filen *"Precio.Smokey.Test.dll"* **inte** innehåller testfall utan nödvändig logik för att testerna ska kunna utföras.

7 Drift & Underhåll

7.1 Versioner av mjukvara

Vid eventuella uppdateringar av mjukvara krävs kompatabilitet mellan NUnit och Selenium. Förutsatt att filnamna vid installation av NUnit förblir oförändrade så påverkas inte testningen.