Detta examensarbete har utförts i samarbete med
Ericsson
Handledare på Ericsson: Thomas Kjellberg

Agilt regressionssystemtest

Agile Regression System Testing

Anwar Aodah
Bora Öcüt

Examensarbete inom
Datateknik, nätverk och säkerhet
Grundnivå, 15 hp
Handledare på KTH: Michael Lundvall
Examinator: Thomas Lindh
Skolan för teknik och hälsa
TRITA-STH 2012: 77

Kungliga Tekniska Högskolan
Skolan för teknik och hälsa
136 40 Handen, Sweden
http://www.kth.se/sth

Sammanfattning

Detta examensarbete utfördes på avdelningsnivån CCS på avdelningen PDU Platforms på Ericsson. Avdelningen ville förändra sin nuvarande arbetsprocess genom att övergå från Daily Build, att bygga och testa en gång om dagen, till ett mer kontinuerligt byggande och testande. CCS hanterar och tillhandahåller kontrollsystemtjänster för RNC applikationer och äldre RBS.

Målet med detta examensarbete var att automatisera testprocesser för att kunna utföra kontinuerliga tester. Första steget i arbetet var att ta fram ett specifikt användningsfall, ett use case, att arbeta mot. Detta skedde genom analys av felrapporter och detta ledde till att nodomstarter definierades som ett första användningsfall. När användningsfallet var definierat, skedde analys av plattform att använda för automatisering, samt vilka steg som behövde automatiseras. De var enligt följande, uppladdning med ftp, exekvering av filer med Telnet och nodkonfiguration med verktyget Cicc.

Arbetet organiserades enligt SCRUM, en agil arbetsmetod lämpad för utvecklingsprojekt.

Uppgiften hade några krav, kraven uppfylldes genom att automatisera exekveringen av automatiserade tester på ett testkort som hade konfigurerats med hjälp av verktyget Cicc. Automatiseringen skedde genom att skapa ett Expect skript med innehållet som nämnts ovan. Skripten skickades via Ericssons terminalmiljö till Gerrit, som användes för att verifiera och granska källkod. När skriptets källkod godkänts i Gerrit, användes verktyget Jenkins för att schemalägga körningar, samt presentera resultatet för användaren. Efter att koden hade godkänts skickades den senaste versionen av testfilerna till den centrala uppsamlingskatalogen.
Jenkins är inte kapabel av att exekvera och läsa resultat själv, därför skapades det en xml-fil som Jenkins kan anropa och läsa resultatet ifrån.

Uppgiften hade ett krav som skulle uppfyllas.

- Kan kvalitetsvinster uppnås genom att införa automatiserat kontinuerligt testande på avdelningsnivå?

Genom automatiseringen av byggandet och kontinuerlig exekvering av testandet på avdelningsnivå kunde man i första hand spara tid. Införandet av automatiserat kontinuerligt testande på avdelningsnivå gjorde att arbetet hade kommit igång för att få bättre kvalité genom att byggandet och exekveringen utökas. Detta ledde till kontinuerlig exekvering av fler testfiler istället för en gång om dagen.

Slutligen definierades tio testfall, där två av dessa testfall var att göra varma och kalla omstarter. Dessa testfall definierades för senare vidareutveckling.

Abstract

This thesis has been carried out at the department level CCS in the department PDU Platforms at Ericsson. The department wanted to change their current work process. They wanted to move from the Daily Build, where you build and test once a day, for a more continuous building and testing. CCS manages and provides control services for RNC applications and older RBS.

The goal of this project was to automate the testing processes to continuous test. The first step in the process was to develop a specific use case, a use case to work forwards. This was done by analyzing error reports and this led to node reboots defined as a first use case. Once the use case was defined, there was analysis of platform use of automation and the steps that needed to be automated. They were as follows: uploading to ftp, execution of files with Telnet and node configuration with the tool Cicc.

The work was organized as SCRUM, an agile methodology suited for development projects.

The task had some requirements, the requirement were met by automating the execution of automated testing on a test card that was configured using the tool Cicc. The automation was done by creating an Expect script with the content mentioned above. The scripts were sent via Ericsson terminal environment to Gerrit, which was used to verify and examine the source code. When the script´s source code was approved in Gerrit, the Jenkins tool was used to schedule runs and present the result to the user. After that the code had been approved, the latest version of the test files was sent to the central repository.

Through Gerrit the binary files was sent to a tool called Jenkins that builds the changes who was made. The response is sent back to Gerrit and finally when everything has gone perfectly. It sends the latest version of the files to the central repository.

The task had a requirement to be met.

- Can quality gains be achieved through the introduction of automated continuous testing at department level?

Through the automation of the building and continuous execution of testing at the division level could primarily save time. The introduction of automated continuously testing at department level meant that work had started to get better quality by building and executing are increased. This led to the continuous execution of the additional test files instead of once a day.

Finally it has been defined ten test cases, where two of these test cases are to make warm and cold reboots. These test cases are defined for later development.

Förord

Denna rapport kommer att sätta ett slut på våra studier på KTH. Det har varit väldigt lärorikt att göra detta examensarbete på Ericsson.
Vi vill tacka vår handledare på Ericsson, Thomas Kjellberg, mjukvaruutvecklare, för all stöd och hjälp vi har fått under projektets gång.
Vi vill även tacka Andrey Devyatkin, mjukvaruutvecklare, för sitt stöd under denna tid.

_____ _____

Anwar Aodah Bora Öcüt

Innehåll

1. Inledning

Arbetet går ut på att automatisera den nuvarande arbetsprocessen för ett mer agilt arbetssätt. Ett av de agila arbetsmetoderna är att arbeta med scrum, där arbetet är mer kontinuerligt och anpassad för eventuella arbetsförändringar. Mer om det agila arbetssättet kan man läsa i kapitel 4.

1.1 Bakgrund

I takt med att avdelningen PDU Platforms (Product Development Unit Platforms) förändrar sina arbetsprocesser för ett mer agilt arbetssätt, övergår man från Daily Build, att bygga och regressionstesta en gång om dagen, till Continuous Integration, bygga och testa kontinuerligt.

Med denna förändring vill CCS (Common Control Systems), avdelningsnivå på PDU Platforms, exekvera lågnivå tester av källkod på samma tid som deras högnivå tester för att reducera FST[1] (Fault Slip Through) och förbättra handläggningstiden för dem. CCS avdelningen hanterar och tillhandahåller kontrolsystemtjänster för RNC(Radio Network Controller) applikationer och äldre RBS(Radio Base Station).

1.2 Målformulering

Målet med detta examensarbete var att automatisera den nuvarande arbetsprocessen. Kravet från Ericsson var följande delmål:

- En analys av felrapportsstatistik, där minst ett område identifieras och är intressant att förbättra. Referenser till felrapporter inom området skall kunna presenteras.

- Definition av minst 10 testfall inom det analyserade området, som automatiskt exekveras för varje byggcykel (var tredje timme).
  Exekveringen skall använda existerande ramverk om möjligt.

- Automatiserad konfiguration av systemet, med hjälp av existerande verktyg, som automatiskt exekveras för varje byggcykel. Konfigurationen bör inte ta längre tid än 30 minuter. Konfigurationen skall inte använda GUI (Graphical User Interface). Konfigurationen skall sätta upp så stor del av systemet som verktyget tillåter.

- Modifiering av existerande testramverk för att kunna användas mot system i mindre lab.

  Examensarbetet kan även innehålla:

- Ytterligare testfall, i första hand för det identifierade området, men även för andra områden.

- Utökad konfiguration av systemet, för att möjliggöra konfiguration av större system.

Uppgiften har ytterligare ett krav:

- Kan kvalitetsvinster uppnås genom att införa automatiserat kontinuerligt testande på avdelningsnivå?

1.3 Avgränsningar

Under examensarbetets gång kommer man att arbeta efter projektmetoden scrum som beskrivs i kapitel 4, istället för den vanliga tidsplaneringen.
På företaget ska det analyseras felrapporter för att förbättra och vidareutveckla testsystemet. Man kommer att få tillgång till lokaler, datorer och testutrustningar, tillgång till Ericssons labbutrustning, testhårdvara och använda Ericssons terminalservermiljö. Detta examensarbete kommer i första hand att fokusera på att bygga och exekvera tester med hjälp av existerande testramverk. Testerna kommer att exekveras på testkort. Det som har valts bort är att det inte kommer att vara någon fokusering på byggandet av testladdmodulerna.

Denna rapport kommer att fokusera på testning efter nodkonfiguration. Nodkonfigurationsprocessen kommer att behandlas i rapporten *Agile regression system testing*, 2012, skriven av Andreas Nordvall[B]

1.4 Syfte

Syftet med detta examensarbete är att analysera felrapportsstatistik för att få en vägledning om var de största kvalitetsvinsterna kan uppnås. Därefter är nästa steg att utveckla en automatiseringsprocess för att med hjälp av befintliga verktyg konfigurera ett system med en delmängd av systemets mjukvara, hämtad från officiellt byggda binärfiler, samt exekvera lämpliga testfall på systemet.

2. Nulägesbeskrivning

Avdelningen PDU Platforms på Ericsson består av flera mindre avdelningar, en av dessa är CCS avdelningen. CCS bygger och testar all sin källkod en gång om dagen som kallas Daily Test. Under det pågående testet byggs alla delar tillsammans och det testas för fel. Det kommer att bli en förändring som gör att CCS från att bygga och testa en gång om dagen kommer att övergå till ett mer agilt arbetsmetodik på Ericsson. Därför kommer det inte längre att användas Daily Test, istället kommer det att användas kontinuerliga tester.

Eftersom testet kommer att exekveras oftare, kommer CCS[1] avdelningen på Ericsson att kunna driva sina egna tester parallellt med det fullständiga byggandet. Genom att testa CCS skilt från de andra avdelningarna kommer det att minska på FST[1] och få en snabbare återkoppling. Hela testprocessen bör bli automatiserad.

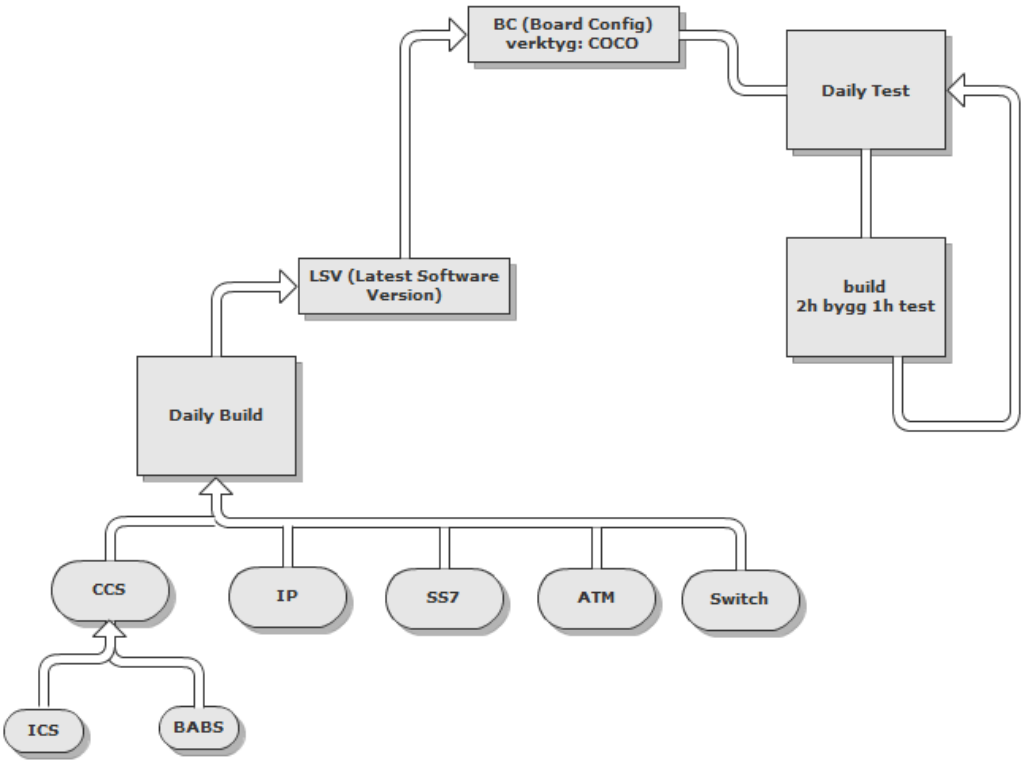


Bild 1 Systemarkitektur av PDU Platforms

[1] Appendix 1

3. Teoretisk referensram

Innan start av detta examensarbete, behövdes viss undersökning göras. FST skulle analyseras, för att bedöma hur mycket man kunde vinna på att förbättra testprocessen. Där uppstår också frågan om hur en lösning ska implementeras och i vilken miljö den ska användas.

3.1 Analys av Fault Slip Through

FST (Fault Slip Through) är fel som hittas senare i processen än de borde hittats. I varje steg från design till färdig produkt finns det tester som är tänkta för att hitta fel och buggar. Att upptäcka fel tidigt har flera fördelar som gör det lättare att hitta och åtgärda felen[16],[C]. Om man inte lyckas hitta felet i början kan det vara svårt att hitta anledningen till varför det inte funkar senare i processen då andra människor lägger in sin del av källkod. Detta kan medföra ett stopp för flera grupper från att fortsätta lägga till nytt, vilket innebär att denna process blir ekonomiskt dyrt och tar längre tid. Därför när ett fel hittas senare, klassificeras detta som FST[1]. På så sätt finns det statistik för att förbättra testprocessen så att fler fel hittas nästa gång.

Begränsningen gjorde att det kollades på olika felrapporter[2] som hade inkommit under de tre senaste månaderna på systemet CCS som består av tiotals delsystem, varav två av dessa delsystem är BABS[1] och ICS[1]. BABS tar hand om de grundläggande operativsystemfunktionerna, medan ICS ansvarar för den interna kontrollsignaleringen av
noderna.
Det som framgick från felrapporterna var vilka feltyper som uppstod. Efter detta delades dessa upp i separata grupper beroende på deras typ. Analysen bestod av totalt tretioåtta FST fel.

- Åtta mjukvarurelaterade fel. Noder som uppträdde felaktigt på grund av fel i källkoden.
- Fyra hårdvarurelaterade fel. Noder som uppträdde felaktigt på grund av fel i hårdvara.
- Fyra fel i kraschhantering och återhämtning. Dessa fel var loggfiler och kraschdumpningar som inte gjordes på rätt sätt.
- Sju fel med anknytning till omstarter av noden. De mesta berodde på att noderna tog för lång tid att starta om och ibland startade felaktig.
- Åtta diverse förbättringar. Det var inte fel egentligen utan de flesta av dem var källkods recensioner och FST gjordes för att säkerställa att granskningen gjordes.
- Fem för att lägga till eller på grund av saknad funktionalitet. Dessa var funktionaliteter som hade glömts bort att läggas till eller behövdes läggas till.
- Två FST dokumentation. Manualen var felskriven och var tvungen att redigeras.

[1] Appendix 1

Att förbättra testning av omstarter, borde vara lättare och reducera FST som första typfall. För att testa omstarter av noden, omstartas noderna några gånger. Både varma och kalla omstart. En varm omstart är för mjukvaran och är snabbare, medan de kalla omstarterna är för hårdvaran där den startar om. Det ska observeras att omstartstiden svaras inom en viss tid som är utsatt. Om det inte lyckas, är det något fel med omstarten vilket innebär att noden behöver undersökas och eventuellt åtgärdas.
Mjukvarurelaterade fel är svårare att hitta jämfört med omstarter. Mjukvarurelaterade problem är väldigt olika från varandra och det är inte möjligt att bara testa dem genom att bara göra en förändring. Därför har det kommit fram till att börja med att skriva testfall för omstarter.

3.2 Förutbestämd teknologi

Det som tidigare gjorts under avdelningarna på Ericsson var att man använde CC[1] som står för ClearCase, ett verktyg som är särskilt utformat för att lättare utveckla mjukvara i projektgrupper. Den har många egenskaper, som versionshantering, gemensamma gränssnitt, automatisering av informationsspridning, den kan ha kontroll över processutvecklingen och en av de kanske viktigaste sakerna är att man kan arbeta i grupper spridda över större geografiska områden. ClearCase stödjer versionshantering av
textfiler, källkod, bibliotek, binärer med mera.
Man använder ett verktyg som heter Cicc[1] (Core Integration node Control Center) för att konfigurera noderna, men konfiguration och testning borde vara kontinuerligt.
För det har man två alternativ. Antingen modifiera det nuvarande testramverket Daily Test eller använda Jenkins som används på en annan avdelning på Ericsson i samma byggnad. Hittills har det bara skapats testfall för att senare bygga och exekvera manuellt mot testkortet.

3.2.1 Daily Test

Daily Test är en process skapad av Ericsson för att testa källkoder varje dag. Under natten som Daily Test bygger all källkod kan det nästkommande dag se resultatet av byggandet. Daily Test ansågs vara svårt under detta arbete med kontinuerlig testning. Att kunna få en mindre lokal installation var mycket svårt, eftersom Daily Test har en hel del beroenden.

3.3 Tilläggsteknologi

Teknologin som har börjat användas och kommer att användas av alla avdelningar på Ericsson som sitter och arbetar med programmering kommer att övergå till Git. Git kommer att ersätta CC mer och mer.

Det som skulle göras var att hela testandet med bygg och exekvering av testerna skulle automatiseras. Detta skulle göras genom att skapa en wrapper med skriptspråket Expect. Tidigare har det gjorts så att alla skickar in sina ändringar via Git, där de senare läggs ihop till en gemensam källkod. Men detta har bara gjorts en gång om dagen. Istället ska

[1] Appendix 1

det övergås till ett nytt system som heter CI[1] (Continuous Integration), CI fokuserar på att integrera och genomföra tester av plattformsmjukvaran.

3.3.1 Jenkins

Jenkins[12] är en öppen källkods applikation för automatiserad testning. Som används av en annan avdelning på Ericsson för deras tester. Att arbeta med Jenkins blev mycket lättare. Det används i en aktiv miljö, så den sattes upp utav Ericsson. Jenkins kan exekvera ett källskript, därför skrevs det ett skript. Efter att Jenkins har exekverat skript, läser den resultatet från en xml-fil. Att få tillträde till minilabbet med Jenkins var enkelt. Allt som behövdes göras var att öppna en minilabb brandvägg för Jenkins.

3.3.2 Expect

Expect[A] är ett skriptspråk, används som ett verktyg av Linux/Unix och Windows för att automatisera tester av interaktiva applikationer. Dessa applikationer uppmanar användaren att skriva in tangenttryckningar som svar. Med hjälp av Expect, kan det skrivas ett enkelt skript för att automatisera dessa interaktioner. Expect är speciellt utformad för att samverka med interaktiva program.

3.3.3 Git

Git[11] är en öppen källkods versionshanteringssystem designad för att anpassa arbetsmetodiken från små till stora projekt. Git är extremt snabbt för att kunna hantera den stora volymen källkod och ändringar som Linux kärnan kräver.

4. Scrum

Scrum grundades av två personer Jeff Sutherland och Ken Schwaber 1995. Men innan var det japanska managementforskarna Hirotaka Takeuchi och Ikujiro Nonaka som använde denna metod som liknar sporten rugby. Där målet med denna sport är att hela laget spelar tillsammans för att föra bollen uppför planen. Denna typ av arbetsform använde japanerna till en mer stafettliknande process som nu kallas sprintar.

4.1 Vad är Scrum?

Inom mjukvarubranschen används scrum metodiken[14]. Scrum är agil metod som fokuserar mer på vad som skall utvecklas än hur det rent praktiskt programmeras. Scrum består av scrumteam och deras roller, aktiviteter och regler. Varje beståndsdel i ramverket har sitt eget syfte och är väsentlig för en bra användning av scrum. Med scrum bestämmer själva teamet hur mycket man klarar av att göra under en viss period, sprint.

4.2 Varför Scrum?

Varför scrum och inte vanlig tidsplanering?
Skillnaden mellan scrum och vanlig tidsplanering är att på tidsplanering planeras det från start vecka till slut vecka. På scrum planeras det i två veckors sprintar. Där det istället finns dagliga scrum möten på femton minuter, som hålls med tre frågor.

– Vad har gjorts (färdigt)?
– Vad skall göras?
– Vad hindrar?

Som består av en sprint backlog på varje vecka. Sprint backlog innebär att efter varje avklarad moment sätts det ett kryss på det och går vidare till nästa moment.

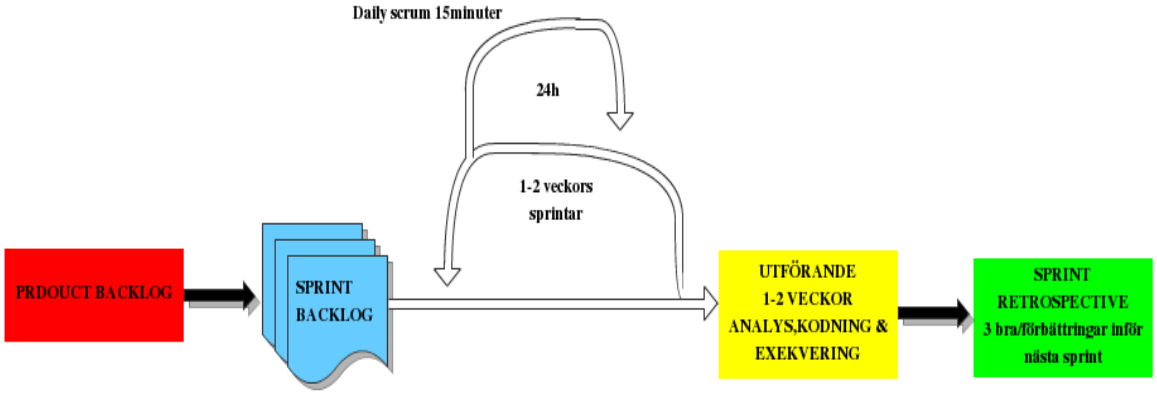


Bild 2 Scrum proceduren

5. CLT

CLT[1] (Component Level Test) är för CCS CT (CCS Component Test) på det specifika ramverket och definierar hur tester organiseras och exekveras[4]. CLT är en del av den Test-support komponent och är delad i två separata testkategorier, CT[1] och MCT [1] (Multi Component Test). CT använder MTE (minimal testmiljö) medan MCT använder en realtids miljö där CLT läses in som en separat TLM[1](Test Load Module).

5.1 Grundläggande begrepp som ingår i CLT

5.1.1 MTE

Test skrivna för CT är exekverade i en minimal testmiljö, MTE[1]. Denna testmiljö består av ett enkelt operativsystem och bara de kärnfunktioner som behövs för att få testmiljön operativ och kunna kommunicera med omvärlden.

5.1.2 CT

Idén bakom komponent tester i CLT är att isolera komponenter för att bli testade och därefter är den kallad som IUT[1](Implementation Under Test), så mycket som möjligt. Meningen med denna strategi är att kontrollera IUTs externa beroenden och därför dess in och utkommande data. Komponent gränssnitt är C funktionsgränssnitt.

5.1.3 MCT

Tester skrivna för MCT är exekverade genom MTE eller via en testladdmodul. TLM är laddad tillsammans med laddmodulen som är målet för testning och tester är exekverade på samma sätt som CT. MCT använder ett verkligt mål nod eller en checkpoint, som används med CPPemu. CPPemu[1] är en emulator som används på Ericsson.

5.2 Organisering av tester i CLT

Testerna har skrivits i språket C. Ett test är den minsta enheten hanterad av CLT, det vill säga den minsta enheten som kan godkännas eller underkännas. Tester är grupperade i testfall. Testfall är grupperade in i testsekvenser.

[1] Appendix 1

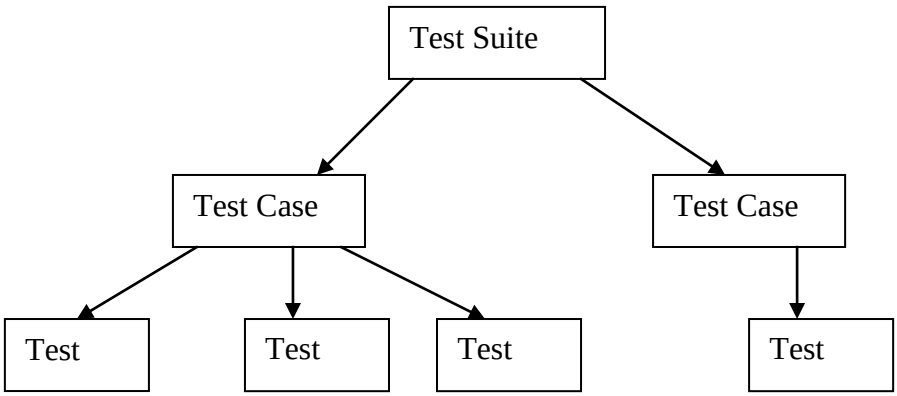


Bild 3 CLT hierarkin

5.3 Strukturering av ett test i CLT

Tester delas normalt upp i tre steg[4].

1. Förberedelse – förberedelse görs för att få systemet och IUT i det tillstånd från vilket testet kan börjas.
2. Exekvering – testet är exekverad genom att förse IUT med några stimuli, alltså att det simuleras.
3. Analysering – resultatet av exekveringen är analyserad genom att verifiera att stimuli gav förväntat resultat.

Ett vanligt sätt att testa sekvenser av händelser är att dela på minnet så att processen kan skriva till två minneskretsar samtidigt, alltså genomförandet och utvärderingsfasen. Med detta menar man att testet skickas till utvärdering och tvärtom. Varje del av sekvensen är en utvärdering när testet utvecklas och när sekvensen når sitt slut. Då alla delar har blivit verifierade och testet har nått sitt slut.

5.4 Skrivandet av ett enkelt testfall

För att skriva ett enkelt test utgörs det genom flera faser, som deklaration, registrering, loggning och synkronisering av testerna.

5.4.1 Deklarera tester

Varje test måste vara deklarerad[4] mellan makron CLT_DECL_TEST och CLT_END_TEST. Varje test har ett namn och en beskrivning om vad den skall utföra.

```
CLT_DECL_TEST(first_restart,"Text example")
{
        /*Test code here*/
}
CLT_END_TEST
```

Bild 4 Deklarering av tester

5.4.2 Registrera tester

Testerna är inte automatisk kända av CLT, bara för att de är deklarerade. Alla tester som CLT borde exekvera måste vara uttrycklig registrerad. Funktionen för registrering är clt_register_tests. Eftersom varje test måste tillhöra ett testfall, skapas ett testfall som registreras för testerna. Testfall skapas med clt_create_test_case funktionen och tester adderas med clt_add_test funktionen[4].

```
void
clt_register_tests(CLT_TestSuite *ts)
{
  CLT_TestCase *tc;
  tc=clt_create_test_case(ts,"Testcase description");
  clt_add_test(tc, first_restart);
}
```

Bild 5 Registerering av tester

I CLT är varje test ansluten med ett utlåtande. Innan ett test exekveras, är utlåtandet okänd, men efter exekveringen är den antingen *"pass"* eller *"failed"*.

5.5 Bygga och exekvera ett testfall

För att bygga ett testfall skapas en testkatalog under källkatalog med alla tillhörande bibliotek och källkoder och lägger in sin c-fil med en build.spec, build.spec är en byggfil. En build.spec skapar användaren själv genom att skriva in alla adresserna till de mappar som ska användas, även inkluderingen av adressen till din skapade fil läggs in här. Efter att dessa två delar är klara, blir nästa steg att bygga mappen genom kommandot bs –cello.
För att exekvera testfallet, blir första steget att använda FTP för att lägga upp <filnamn>.ppc filen till noden. Efter det utförs en inloggning till noden med hjälp av Telnet. Väl inne skriver kommandot pgrun <filnamn>.ppc &[5]. & tecknet används för att exekvera programmet i bakgrunden.

```
$ pgrun xjobb.ppc &
$ clt run 1
```

"Clt run 1" kommer att få visa om testet har lyckats genom ett svar som antingen är PASS eller FAIL. Svaret kommer att skrivas ut på terminalen. Det finns ytterligare flera olika kommandon som kan användas. Kommandot som används för att se hela test suiten är:

```
$ clt list
```

6. Implementering

6.1 Skapa skript och exekvera

Ett enkelt skript för att ansluta till Telnet.

```
#!/usr/local/bin/expect --

spawn telnet <ip-adress>

expect "username: "
send "username\r"
expect "password: "
send "password\r"
expect "$"
```

Bild 6 automatisering av Telnet

Exekveringen görs via kommandot:

```
-> expect filnamn
```

6.2 Wrappers för Jenkins

Jenkins är inte kapabel att exekvera och läsa resultat själv. Därför skapades det wrappers, som Jenkins kan anropa och när den är klar kommer den att få ut resultatet från ett xml-fil som är skapad av en wrapper.
Expect användes för att skapa en wrapper, eftersom detta program hade använts förr. Den första wrappern[1] var för att skapa ett xml-fil och exekvera Cicc för att konfigurera noderna. Nästa wrapper[2] var ett main för att kunna anropa subrutinerna från de andra skripten, Mainwrappern innehåller källkod för att kalla på testnamn som exekverades på varje nod. Wrappern[3] som skulle anropa Telnet och exekvera testerna, hade en uppgift, att exekvera testet och se vilka som hade fått *"PASS/FAIL"*. En del av wrapperns innehåll:

```
send "pgrun $testname &\r"
expect "$"
send "clt list\r"
expect "$"
send "clt run\r"
expect {
        "PASS" {
                set respons "test PASS"
        } "FAIL" {
                set respons "test FAIL"
        }
}
```

[1] Appendix 2
[2] Appendix 4
[3] Appendix 3

Beroende på om tester är *”PASS/FAIL”* sätter den en respons och skickar det till main-wrappern som skickar ett felmeddelande till xml-filen.

Att automatisera innebär att skript gör det som utfördes manuellt. Det behövdes en konfigurationsfil för wrappern, så detta skapades. Konfigurationsfilen innehöll nod namn, rutt till testerna på Telnet, namnet av vilket test som skulle exekveras och den innehöll även en rutt till vart testladdmodulen låg. Här nedan kommer ett exempel:

{
enolans_split
/d/loadmodules
xjobb.ppc
/vobs/cpp/src/babs_ss/control_test_dm/continous_ccs_test/bin
}

I detta fall är det en nod som exekveras på ett test. Testet ligger under d/loadmodules mappen och heter xjobb.ppc. Vid behov kan det läggas till ytterligare filer. Nodnamnet används som ett argument för att kalla på Cicc[B].

Konfiguration och testning av noderna exekveras en efter en. Efter att noden är konfigurerad och testad går den vidare. Efter varje procedur i skriptet kommer resultatet att läggas upp på en lista. Allting i listan kommer senare att skrivas till xml-filen. Om det skulle uppstå några fel kommer den att skrivas på listan och skriptet kommer att gå vidare till nästa nod. Efter att alla noder har blivit testade kommer skriptet att skapa xml-filen.

6.3 XML-fil för Jenkins

Om Jenkins ska veta om konfigurationen och testning gick bra, kommer den att läsa xml-filen som är definierad i Jenkins för resultat. Jenkins har en plug-in installerad så att den stöder XML(Extensible Markup Language). Testrapporter är kompatibla med Junit [15] och dessa ser ut så här:

```
- <testsuites errors="4" failures="2" hostname="test" name="AutoCicc" tests="3" time="..." timestamp="2012-07-13T13:01:39">
  - <testsuite errors="2" failures="2" tests="2" id="" name="Configure_Node">
    - <testcase classname="configure" name="enolans_split">
      + <error type="Error"></error>
      + <failure type="Failure"></failure>
      </testcase>
    - <testcase classname="configure" name="qthokjeSplit">
      + <error type="Error"></error>
      + <failure type="Failure"></failure>
      </testcase>
    </testsuite>
  - <testsuite errors="2" failures="0" tests="1" id="" name="Test_Node">
    - <testcase classname="testfall" name="testfall">
        <error type="Error"> [ ftp: connect: No route to host ftp> ] </error>
        <error type="Error"> [ftp: connect: No route to host ftp> ] </error>
      </testcase>
    </testsuite>
  </testsuites>
```

Bild 7 xml-fil generingen

Junit är ett testramverk för Java applikationer. Men det som spelar roll är att Jenkins kan bli informerad om eventuella fel som uppstår och varför dessa uppstod när filerna laddades upp med ftp eller få information om det fungerade att exekvera Telnet och testerna. Exemplet ovan indikerar misslyckad och fel typerna på att det har uppstått två fel, när man ska koppla upp sig mot ftp, båda gångerna fås samma meddelande om att det inte finns någon rutt till värden.
Failure används för att indikera varningar och om testerna skulle misslyckas att exekvera. Skillnaden mellan *Failure* och *Errors* är att när det sker en *failure*, fortsätter skript sin process till slutet, medan då det inträffar ett *error* stoppas skript direkt.

6.4 Se resultat i Jenkins

Efter att wrapper har exekverats kommer Jenkins att läsa xml-filen och resultatet kan ses med en webbläsare.

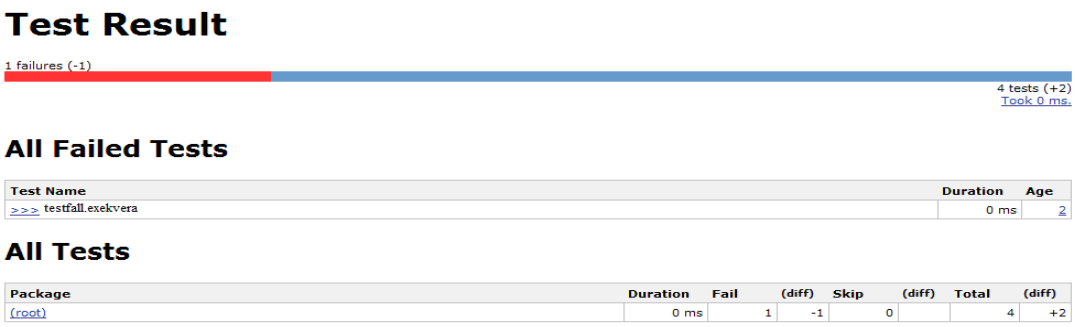


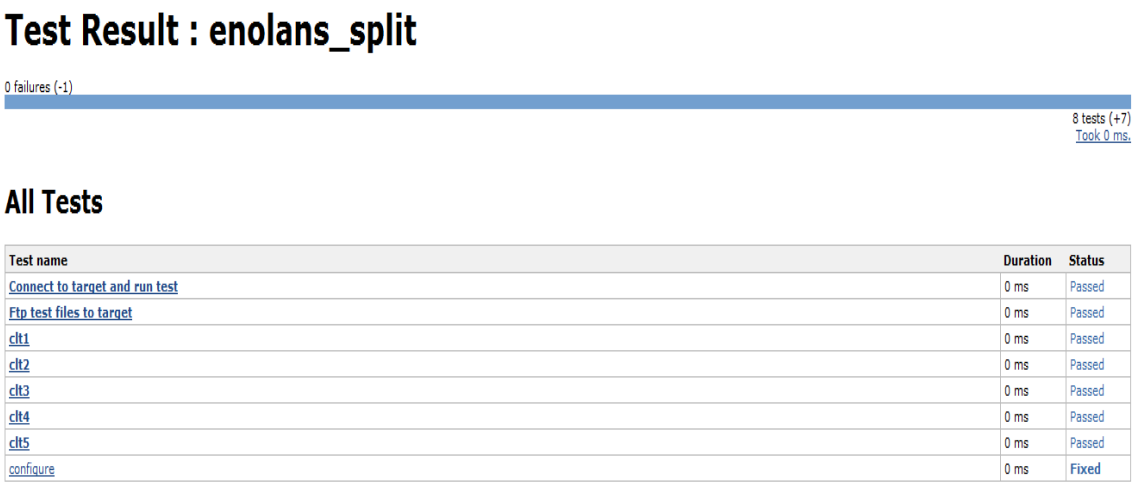
Bild 8 Resultat från Jenkins. Det har uppstått ett fel.

Så här ser resultat ut i Jenkins. Skripten försökte ansluta sig till IP-adressen via Telnet, men misslyckades, vilket skapade ett fel. Genom att klicka på testfall.exekvera kan anledningen ses till varför det misslyckades.

Bild 9 Jenkins felmeddelande

Här är det nu möjligt att se att ett fel har uppstått, eftersom Telnet hade ett fel när den skulle koppla upp sig till IP-adressen 10.67.30.110, men där den rätta IP-adressen var 10.67.30.112. Eftersom den inte lyckades med uppkopplingen kommer testet inte att gå vidare med skriptet och istället kommer det att avslutas och skicka felmeddelandet till xml-filen.

Bilden nedan är resultat på noden enolans_split. *"Connect to target and run test"* misslyckas bara om skript på något sätt inte skulle vara tillgänglig att exekvera alla tester.*"Ftp test files to target"* misslyckas om testfilerna inte kan skickas till noden. clt1 till clt 5 är tester som var felfria. Konfigurationstestet är för Cicc, det står *"fixed"* eftersom den misslyckades på förra exekveringen.



Test Result : enolans_split

0 failures (-1)

8 tests (+7)
Took 0 ms.

All Tests

Test name	Duration	Status
Connect to target and run test	0 ms	Passed
Ftp test files to target	0 ms	Passed
clt1	0 ms	Passed
clt2	0 ms	Passed
clt3	0 ms	Passed
clt4	0 ms	Passed
clt5	0 ms	Passed
configure	0 ms	Fixed

Bild 10 Resultat från Jenkins, PASSED

7. GIT

Git[11] är en källkods versionshanteringssystem. Git är byggt för att anpassa arbetsmetodiken i stora och små öppet källkodsprojekt. Med Git saknas det en central arkiv, istället kan vem som helst skapa en egen kopia. Till Git finns det även program för att skicka och ta emot ändringar i form av patchar via e-post.

7.1 Start av Git

För att komma igång med Git, är första steget att registrera sig i Gerrit[10] där uppladdning och byggandet av sina ändrade filer sker. Detta görs genom att logga in med sitt användarnamn och lösenord.

7.2 Dagligt arbetsflöde

En designers dagliga arbetsflöde[9] omfattar daglig användning av Git som används för gemensamma scenarion, skripter, källkoder, samt lösningar på hur dessa skall lösas.

7.2.1Utveckling av din uppgift eller en funktion på lokal branch

Varje gång vid start av ett nytt terminalskal, måste kommandot *”source setup.sh”* på katalogen /repo/<user-id>/cpp skrivas in. Detta används för att förbereda miljön som ska användas.
Innan igångsättningen av sitt arbete i Git, ska det ses till att det finns en *”local branch”* att arbeta på. En *”local branch”* skall vara baserad på den senaste *”commit”* från *”master branchen”* eller vilken annan uppströms *branch* som helst. *”Commit”* är den senaste informationen som beskriver vad filen har modifierats med. Efter det används kommandot *”git push”* för att lägga till sina lokala ändringar. När detta är klart, börjas det med att först gå till super katalogen för att ladda ner de senaste ändringarna från den centrala uppsamlingskatalogen. Kommandona som används:

```
$ cd /repo/<user-id>/cpp
$ git pull
$ git submodule update
```

Kommandot *”git pull”* kommer att ladda ner den senaste *commit* som finns på super katalogen. och *”git submodule update”* kommer att kolla upp motsvarande *commits* för alla submoduler. Kommandon för att komma fram till submodulen:

```
$ cd /repo/<user-id>/cpp/<submodule>
$ git checkout –b <branch name> origin/master
```

7.2.2 Utvecklingen av en funktion

Efter alla dessa kommandon, är det dags att börja göra sina ändringar på sin källkod. Varje gång det har skett en förändring på filen som har jobbats med, ska kommandot *”git add <ändrad fil>”* användas. *Add* används för att säga till Git att ändringar på en specifik fil ska bli inkluderad in i nästa commit. Det kan läggas flera filer till en och

samma kommentar. Detta görs genom kommandot *”git commit”,* detta sparar ändringarna som har gjorts till den lokala uppsamlingskatalogen.

git add <filnamn>
git add <filnamn>
git commit <kommentarer>

7.2.3 Dela dina ändringar med andra

Alla ändringar som nyligen har gjorts på en lokal branch, kommer bara att synas under din egen uppsamlingskatalog, även kallat ”repository” och kommer inte att kunna ses utav andra. För att detta ska ses utav alla måste man *”pusha”.* Detta kommer att ske genom några kommandon. Först byter man katalog till sin egen repo. Sen skrivs kommandot *”git checkout <local branch name>”*. Efter det nerladdas de senaste ändringarna med kommandot *”git fetch”* från den centrala uppsamlingskatalogen, därefter används *”git rebase –i <remote name>/<upstream branch>”,* som används för att flytta de senaste ändringarna från den lokala uppsamlingskatalogen till toppen av hierarkin. Slutligen kommer det viktigaste kommandot *”git push origin HEAD:refs/for/master”*, som skickar källkoden till Gerrit.

8. Gerrit

Just denna process är specifik för PDU Platforms. Efter användningen av kommandot *"git push origin HEAD:refs/for/master"* och att allting har gått bra, skrivs det ut en URL-länk på terminalen, denna URL-länk skickas även till mail adressen som skrevs i början av registreringen till Gerrit. URL-länken är till Gerrits hemsida med specifik adress för varje användarens uppladdning. Inloggning sker med sitt användarnamn för att ha tillåtelse för att kunna se ändringarna som har gjort.

Längst upp sidan finns det information om vem som har gjort ändringen, på vilken *”branch”* den är på och vilket datum och tid den laddades upp.*” Commit”* är det meddelande som visar om vilken förändring som har gjorts.

Bild 11 Status

8.1 Jenkins

Jenkins är ett program som övervakar utförandet av upprepade arbeten, som att bygga ett programvaru-projekt. Tillsammans med Gerrit kan Jenkins användas med kontinuerlig integration, detta gör att det blir lättare att integrera ändringar i projekt och för användare att erhålla en ny test. Med automatiseringen kan ökning av produktiviteten ske.
Jenkins kommer att automatiskt att börja bygga de ändringar som har gjorts och det kommer att ta skickas en e-post när byggandet börjar och avslutas. Via Gerrit kan det klickas på kommentaren, där det är en URL-länk till Jenkins hemsida och där kan det se hur det ser ut.

Bild 12 Bygga i Jenkins

I Jenkins finns det ett antal länkar som kan användas, en av dem är *”console output”* länken som visar bygg loggan om den pågående processen.

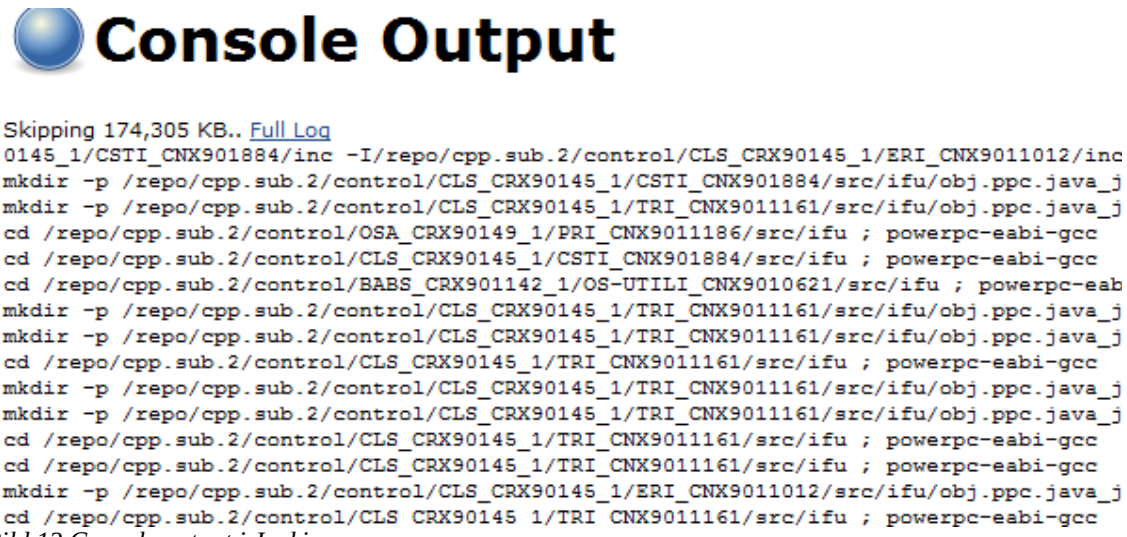


Bild 13 Console output i Jenkins

När det är klart med byggandet i Jenkins, kommer den automatiskt att uppdatera den verifierade statusen enligt byggresultatet.

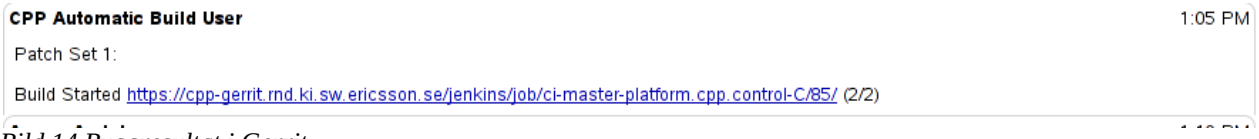


Bild 14 Byggresultat i Gerrit

Om byggandet lyckades kommer verifieringsflaggan att sättas till +1 i kodgranskningen, om den misslyckas kommer den att sätta verifieringsflaggan till ett rött kryss.

Reviewer	Verified	Code-Review
Bora Öcüt		+1
Anwar Aodah		✓
CPP Automatic Build User	✓	+1

Bild 15 Code Review

Men det ska inte glömmas att verifiera kodgranskningen, detta kommer då att göras av användarna som bjuds in via *”add reviewer”* knappen.
När nu både Jenkins bygget är felfri och källkoden är granskad, ska det slå ihop ändringarna in i den centrala uppsamlingskatalogen. Även för detta finns det en enkel knapp som kallas *”Submit patch set”*. Via klickandet på knappen kommer Gerrit att slå ihop dina ändringar in i den centrala uppsamlingskatalogen. Om det har lyckats kommer den att se ut såhär:

Owner	Bora Öcüt
Project	platform/cpp/control
Branch	master
Topic	
Uploaded	Jul 18, 2012 3:21 PM
Updated	Jul 19, 2012 8:54 AM
Status	**Merged**
Can Merge	Yes

Bild 16 Merge

Annars kommer den i kommentarer att visa vad som fattas, vad som ska göras för att få det att fungera.

9. Resultat

Examensarbetet resulterade i automatisk exekvering av testerna på testkortet. Skriptspråket Expect användes för att skapa wrappers. Innan testerna utförs, konfigureras
noderna med hjälp av Cicc[B]. Examensarbetet har resulterat i följande punkter:

9.1 CLT

Under CLT testramverket skapades det ett enkelt testfall. Testfallet skrevs genom att uppnå vissa krav som behövdes inom CLT. Dessa var deklaration och registrering av testerna. Byggandet och exekvering skedde genom kommando som har nämnts i kapitel 5.5. Vid skapandet av ett testfall skapades det en build.spec fil. Denna fil var för att testfallet skulle veta var testet var baserad, vilken miljö den skulle exekveras i och vilken typ det var.

Problem som uppstod vid exekvering av testfall var följande:

9.1.1 Problem som har uppstått

- build.spec bygger inte, hittar inte innehållet i MTE
- Kan inte byggas och exekveras via sitt eget användarnamn.
- Kan inte kompilera med hjälp av kommandot cc filnamn.

9.1.2 Problemlösning

- I build.spec har ändring av namn för det exekverade utdatat och källadress där filerna ligger skett. Bortkommenterat källkoder som inte behövdes för att få igenom exekveringen.
- Överföra de filer som har skapats till Ericssons egen källmapp. Byggandet exekveras i källmappen och exekvering utförs via Telnet.
- Använda kommandot bs –cello istället.

9.2 Automatisering av testning och konfiguration

Jenkins anropar wrappern inne i källskripten. Wrapperskript läser en konfigurationsfil som innehåller information på nodnamnen och vilket test som skall exekveras på dem. Varje nod är konfigurerad och testad en efter en. Om skriptet misslyckades under konfigurationen eller när testet läggs upp, kommer den att ta bort testet och rapportera ett fel senare på Jenkins. Om testfallet är misslyckat kommer det att rapporteras till Jenkins. Därefter om alla testade noder lyckades, kommer resultatet att skrivas till xml-filen. Xml-filen kan läsas av en webbläsare

För att kunna automatisera testning skapades det wrappers. För att exekvera Cicc och skapa xml-filen skapades en *configwrapper*[1], för att bygga och exekvera testerna via ftp och Telnet skapades en *testwrapper*[1].

[1] Appendix 2

I *mainwrappern*[2] skapades det variabler som anropade subrutinerna i de andra wrappern. Variablerna som anropades var för byggning och exekvering. Efter skapandet av alla wrappers användes Ericssons terminalmiljö. I terminalen anges de kommandon man ska använda i Git. För att lära sig Git, fick gruppen genomgå en tre dagars intensiv kurs på Ericsson.

9.3 Gerrit

Efter angivningen av kommandona på Ericssons terminalmiljö och allt gick felfritt skickades det en URL-länk som är hemsidan till Gerrit, där kodgranskning fick göras, alltså granska om källkoden stämmer. Samtidigt som URL-länken skickas, börjar Jenkins med att bygga de förändringar som har gjorts. Dessa bygganden tog cirka fyrtio minuter. Efter att den hade byggts klart fick Gerrit ett svar som antingen var *”successful”* eller *”failed”*.

Felen som troligtvis kan förekomma är att källkoden inte är granskad eller verifierad. Byggandet i Jenkins har misslyckats. Eller så kan det uppstå en konflikt när det ska slå ihop alla wrappers. För att lösa denna måste kommandot *”git rebase –i origin/master”* användas. Ta hand om konflikterna och ladda upp en ny ändring.

När allting har gått felfritt. då är det dags för *”merge”,* alltså ladda upp ändringarna på den centrala uppsamlingskatalogen. Detta gjordes med hjälp av en knapp i Gerrit, *”submit patch”*. Efter detta kan övriga användare hämta de senaste versionerna av nuvarande filerna som nu ligger uppe.

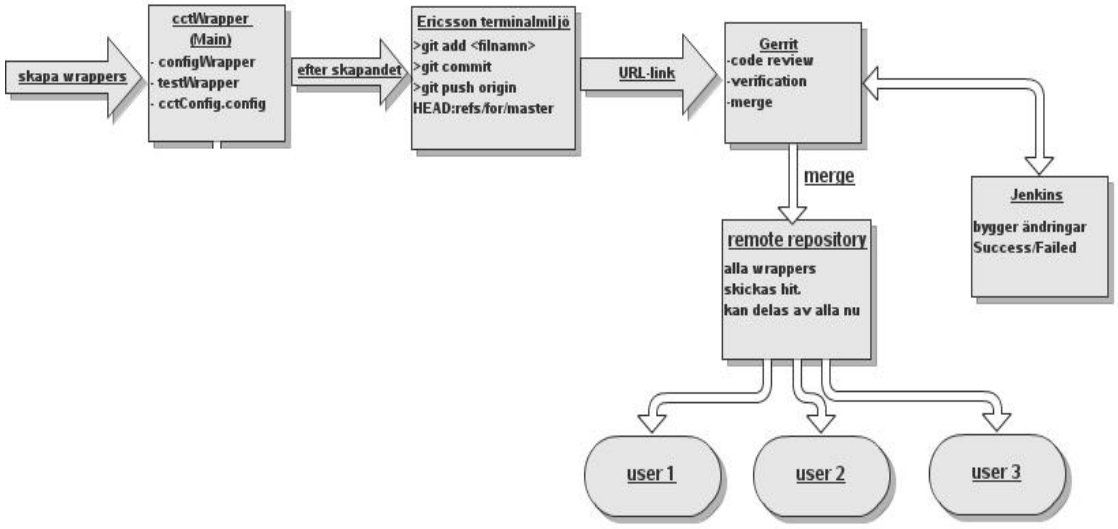


Bild 17 uppbyggandet av hela examensuppgiften

[1] Appendix 4
[2] Appendix 3

10. Slutsats

Målet med ”Agila regressionssystemtest” var att automatisera arbetsprocessen på avdelningen CCS på Ericsson. Examensarbetet hade några mål som skulle uppfyllas. Genom att analysera felrapportstatistiken, framgick det att det skulle börjas med att förbättra omstart av noderna.
Med hjälp av skript som skrevs, lyckades automatisering av testandet ske genom att ladda upp och bygga testladdmodulen. Testladdmodulen innehöll testfall som hade tester i sig. Genom att exekvera testerna fick man resultatet ”PASS” som var målet, som betydde att man hade lyckats med testerna. Tidsreducering på noderna hade optimerats[B].

Projektgruppen har under projektets gång lärt och fördjupat sig inom området exekvering av testfall, genom att skriva ett skript med hjälp av skriptspråket Expect. Detta hjälpte projektgruppen att exekvera och bygga testladdmodulen automatiskt.

Uppgiften hade ett krav som skulle uppfyllas:

- Kan kvalitetsvinster uppnås genom att införa automatiserat kontinuerligt testande på avdelningsnivå?

Genom automatiseringen av byggandet och kontinuerlig exekvering av testandet på avdelningsnivå kunde man i första hand spara tid. Införandet av automatiserat kontinuerligt testande på avdelningsnivå gjorde att arbetet hade kommit igång för att få bättre kvalité genom att byggandet och exekveringen utökas. Detta ledde till kontinuerlig exekvering av fler testfiler istället för en gång om dagen.
Varje testladdmodul kunde innehålla flera testfall och dessa testfall kunde innehålla ytterligare hundratals tester i sig.

Ett annat mål var att definiera tio testfall, dessa definitioner gjordes och finns att läsa under ”Rekommendationer” kapitel 11.

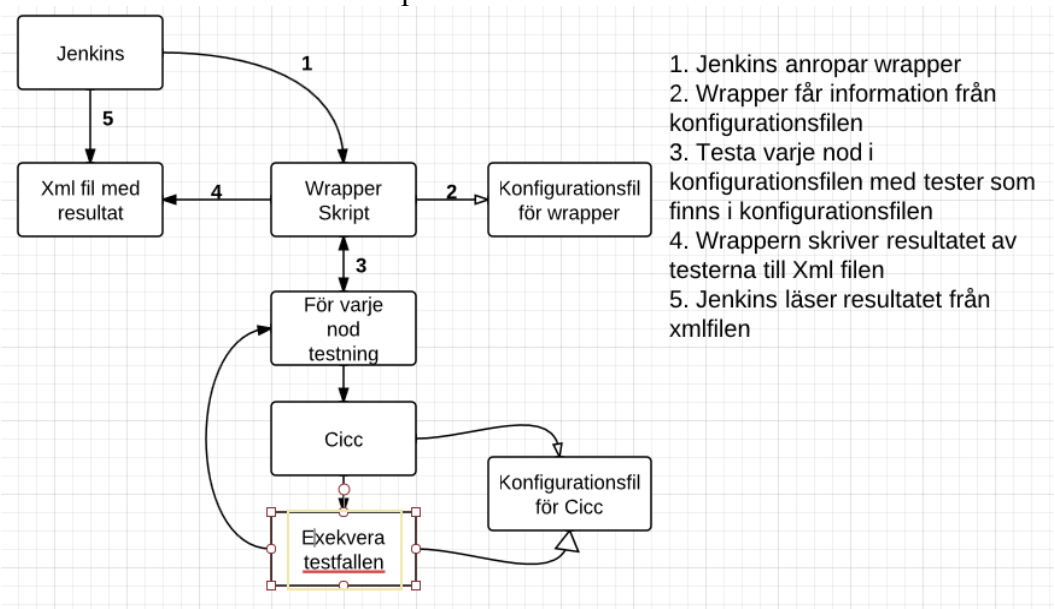


Bild 18 automatisering av CCS avdelningen

.

11. Rekommendationer

Detta projekt byggde på en existerande färdigbyggd testladdmodul. Men vår rekommendation är att skriva skript som bygger testladdmoduler vid varje testexekvering. Definition av tio testfall har gjorts som finns numrerade under, nästa steg för detta arbete är att använda dessa vid en vidare utveckling. Dessa testfall var:

1. Starta om kortet med Cold Restart, och ta tid på hur lång tid det tar att komma till start utskriften.
2. Starta om kortet med en Warm Restart, och räkna hur lång tid det tar att komma till 'Login Server Ready'-utskriften.
3. Starta om kortet med en Cold W Test Restart, och räkna hur lång tid det tar att komma till WERA-utskriften.
4. Skicka ett antal (1000) signaler till bipservern, upprepa en gång per minut i femton minuter. Om kortet inte har kraschat efter fem minuter är det tillräckligt stabilt.
5. Kontrollera om EDD drivrutinen har den senaste versionen, annars uppdatera till den senaste.
6. Kontrollera om RMD ID dupliceras under en omstart, om detta händer ta bort en av RMD ID.
7. Kontrollera om EPB1 MTE har rätt konsolhastighet, annars ange rätt hastighet.
8. Verifiera att BM genererar en krasch av kortet, tio minuter efter sista kontakten med System Managern.
9. Skriv en felhantering till CBM3-CS-FT som rensar RAM, då du gör en varm omstart, eftersom RAM räcker inte till.
10. Gör 1000 omstarter på CBM3 kortet och se om USB enheten hittas vid samtliga.

Källförteckning

Tryckta

[A]. Libes, Don, Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs, O´Reilly Media Inc, 1995

[B] Nordvall, Andreas, Agile Regression System Testing, 2012

[C] McConnell, Steve (2004). Code Complete (2nd ed.). Microsoft Press. p. 29. ISBN 0-7356-1967-0

Interna

[1]. Kaipainen, Harry, *Operational description*, 2012-05-29
http://internal.ericsson.com/page/hub_net/unit/unit_01/u_14/operational_descr/index.js (2012-07-19)

[2]. *Analyze of FST*
http://cdmweb.ericsson.se/WEBLINK/ViewDocs?DocumentName=EAB%2FFTP-08%3A0627&Revision=BE
2012-06-01

[3]. Sandberg, Mats, *DTE Build Support User Guide*, 2009-06-04
http://cdmweb.ericsson.se/TeamCenter/controller/ViewDocs?DocumentName=1553-LXA1191352&Latest=true&Approved=true (2012-06-04)

[4]. Ericsson, *Users´s Guide for Test Support*, 2011-02-07
http://cdmweb.ericsson.se/TeamCenter/controller/ViewDocs?DocumentName=1553-CXA1102809&Latest=true (2012-06-06)

[5]. Olsson L, Johan, *Build upload and run*, 2011-03-17
https://ericoll.internal.ericsson.com/sites/RBS_Management/Wikis/Build%20upload%20and%20run.aspx (2012-06-08)

[6]. Ericsson, *Jenkins User guide*, 2012-01-23
https://ericoll.internal.ericsson.com/sites/NDO_EM/Documents/Transfer/Tools/Jenkins.ppt#263,8,User Linux Build (2012-06-16)

[7]. Jönsson, Peter, *Git at PDU Platforms*, 2012-07-13
https://wikimgw.rnd.ki.sw.ericsson.se/display/git/Git+at+PDU+Platform (2012-06-17)

[8]. Virtanen, Jussi, *Git*, 2012-06-18
https://wikimgw.rnd.ki.sw.ericsson.se/display/git/Git (2012-07-16)

[9]. Nimac, Luka, *Daily Workflow*, 2012-28-06
https://wikimgw.rnd.ki.sw.ericsson.se/display/git/Daily+Workflow (2012-07-16)

[10]. Arkhipov, Sergey, *Get CPP Repositories*, 2012-07-18
https://wikimgw.rnd.ki.sw.ericsson.se/display/git/Get+CPP+Repositories (2012-07-16)

Externa

[11]. Chacon, Scott, Pro GIT, 2009-07-29
http://git-scm.com/ (2012-07-20)

[12]. Kawaguchi, Kohsuke, *Meet Jenkins – Jenkins – Jenkins Wiki*, 2012-04-27
https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins (2012-06-07)

[13]. A Bridge to Computer Science, *UNIX Commands Guide*
http://www.cs.brown.edu/courses/bridge/1998/res/UnixGuide.html#simple
(2012-07-05)

[14]. Wikipedia, *Scrum (development)*, 2012-07-30
http://en.wikipedia.org/wiki/Scrum_%28development%29 (2012-06-04)

[15] Jenkins CI, *JUnit Attachments Plugin - Jenkins - Jenkins Wiki*, 2012-07-13
https://wiki.jenkins-ci.org/display/JENKINS/JUnit+Attachments+Plugin

[16]. Software Testing, Economics, 2012-08-03
http://en.wikipedia.org/wiki/Software_testing#Economics (2012-08-05)

Mjukvara

[17]. https://cpp-gerrit.rnd.ki.sw.ericsson.se/
2012-07-18

[18]. https://cpp-gerrit.rnd.ki.sw.ericsson.se/jenkins/
2012-07-06

Muntliga

[19]. Thomas Kjellberg, Software Developer, Ericsson

Appendix 1

BABS	Ett delsystem på CS (hanterar mycket grundläggandeoperativsystemfunktioner)
bs -cello	Används för att bygga testladdmodul, bs = build structure
CC	ClearCase
CCS	Common Control System
CI	Continuous Integration
CICC	Core Integration node Control Center
CLT	Component Level Test
CPPemu	CPP emulator
CS	Common System
CT	Component Test
CTU	CLT Test Utilities
FST	Fault Slip Through
GUI	Graphics User Interface
ICS	Interprocess Communication System
IUT	Implementation Under Test
LSV	Latest Software Version
MCT	Multi Component Test
MTE	Minimal Test Environment
PDU Platforms	Product Development Unit Platforms
RBS	Radio Base Station
RNC	Radio Network Controller
TLM	Test Load Module

Appendix 2

```
    #Start writing the xml.
    puts $input "<?xml version='1.0' encoding='UTF-8'?>"
    puts $input "<testsuites errors=\"$error_total\" failures=\"$failure_total\"  hostname=\"cctWrapper\"
    name=\"$name\" tests=\"[expr "[llength $result_list]"]\" time=\"...\" timestamp=\"$timestamp\">"
    # Create he configure testcase first
    foreach nodename $nodenames {
    puts $input "<testsuite errors=\"$n_error($nodename)\" failures=\"$n_failure($nodename)\"
    tests=\"$n_test($nodename)\" id=\"\" name=\"$nodename\">"
    foreach  result $result_list {
        #result_list index 0 = Name 1 = Type 2 = Error 3 = Failure
        if {[lindex $result 0] == "$nodename"} {
        set teststage [lindex $result 1]
        if { [lindex $result 2] == "" && [lindex $result 3] == "" } {
            puts $input "<testcase classname=\"$nodename\" name=\"$teststage\" />"
        } else {
            puts $input "<testcase classname=\"$nodename\" name=\"$teststage\" >"
            foreach error_message [lindex $result 2] {
            puts $input "<error type=\"Error\">"
            puts $input "\[$error_message\]"
            puts $input "</error>"
            }
            foreach failure_message [lindex $result 3] {
            puts $input "<failure type=\"Failure\">"
            puts $input "\[$failure_message\]"
            puts $input "</failure>"
            }
            puts $input "</testcase>"
        }
        }
    }
    puts $input "</testsuite>"
    }
    puts $input "</testsuites>"
    close $input
    return 0
}
```

Appendix 3

```
##############################
#
#       MAIN
#
##############################
set nodename "ScriptStart"
# There should be a config file. Read it!
set type "ReadConfigFile"
set config [read_config]
if {$config == "1"} {
    set errors "Failed to read the configfile!"
    lappend nodenames $nodename
    generate_testcase_result $nodename $type $errors $failures
    if {[create_xml $result_list $nodenames]} {
        exit 1
    }
    exit 0
}
#Lappend the result of the file reading

foreach element $config {
    set respons ""
    set errors ""
    set failures ""
    set nodename [lindex $element 0]
    lappend nodenames $nodename
    set testpath [lindex $element 1]
    set testname [lindex $element 2]
    set testlocation [lindex $element 3]

    set errors ""
    set failures ""
    set type "configure"
```

```
    #Ftp the test files to the node.
    #Now we wait for the node to start the login server
    puts "Sleep 30\n"
    sleep 30

    set type "Ftp test files to target"
    if { [send_files_to_target] } {
        lappend errors "FTP failed: $respons"
        generate_testcase_result $nodename $type $errors $failures
        return 1
    }
    generate_testcase_result $nodename $type $errors $failures

    #Connect to the node and run the tests.
    set type "Connect to target and run test"
    if { [connect_to_target 2] } {
        lappend errors "Connecting to target  failed: $respons"
        generate_testcase_result $nodename $type $errors $failures
        continue
    }
   generate_testcase_result $nodename $type $errors $failures
}

if {[create_xml $result_list $nodenames]} {
    exit 1
}
```

Appendix 4

```
proc run_test { } {
    global update_config
    global shell
    global testpath
    global testname
    global n_test
    global tests
    global node_connection_info
    global line
    global nodename

    set prompt "\\\$"
    set n_test 0
    set timeout 15

    #Here starts the execute of the testnames, use the command pgrun to run the test
    #for every test in the testcase list run the "clt run <n>" command

    if {[prepare_for_test]} {
        #Error! Error message is set in prepare_for_test
        return 1
    }
    # Check how many tests there is and run them!
    send "clt list\r"
    expect "est suite*$prompt"
    set variabel $expect_out(buffer)
    foreach line [ split $variabel "\n"] {
        if { [ regexp "(\[0-9])" $line] } {
            incr n_test
            set type "clt$n_test"
            set errors ""
            set failures ""
            #Wait a moment
            sleep 1
            send "\r"
            expect "$prompt"
            send "clt run $n_test\r"
```

```
expect {
    "PASS" {
        generate_testcase_result $nodename $type $errors $failures
    }
    "FAIL" {
        lappend failures "Test number $n_test failed!"
        generate_testcase_result $nodename $type $errors $failures
    }
    "UNKNOWN" {
        lappend failures "Test number $n_test is UNKNOWN!"
        generate_testcase_result $nodename $type $errors $failures
    }
    "Issuing RESTART_TEST_REQUEST" {
        #It's a restart test! Go to a procedure to see how long the restart takes.
        if {[restart_test $n_test $type]} {
            return 1
        }
        #Its all good. Login so the other tests can continue
        set password $update_config(target_ftp_password)
        send "\r"

        expect {

            "username:" {
                exp_send "[exec whoami]\r"
                exp_continue
            }

            "password:" {
                # Add an exta return to get a prompt of some kind
                exp_send "$password\r\r"
                exp_continue
            }

            "\\\$" {
                #Prepare the node so running more test is possible after a restart
                if {[prepare_for_test]} {
                    #Error! Error message is set in prepare_for_test
                    return 1
                }
            }

            "Entering server port" {
                exp_send "\r"
            }

                timeout {
                    error_user "\n*** ERROR: Connection to terminal server timed out.\n"
                    return 1
                }

                eof {
                    error_user "\n*** ERROR: Connection to terminal server failed : $expect_out(buffer)\n"
                    return 1
                }
            }
        }

    }
}
puts "There is $n_test test(s)"
return 0
}
```