Institutionen för datavetenskap
Department of Computer and Information Science

Examensarbete

Testramverk för distribuerade system

av

Jakob Pogulis

LIU-IDA/LITH-EX-G--13/010--SE

2013-06-16

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköpings universitet
Institutionen för datavetenskap

Examensarbete

Testramverk för distribuerade system

av

Jakob Pogulis

LIU-IDA/LITH-EX-G--13/010--SE

2013-06-16

Handledare: Peter Dalenius
Examinator: Peter Dalenius

Testramverk för distribuerade system

Jakob Pogulis[1]
jakob@pogulis.se

Institutionen för Datavetenskap
Linköpings universitet

16 juni 2013

[1]Denna uppsats är det avslutande momentet på min utbildning Innovativ Programmering vid Linköpings Universitet. Under utbildningens gång har jag lärt mig mycket mer än vad jag vid utbildningens början trodde fanns kvar för mig att lära inom programmering och datavetenskap. Bland många mycket duktiga adjunkter, lektorer och professorer vill jag framför allt tacka Peter Dalenius, Anders Haraldsson och Johan Åberg för fantastisk undervisning och råd kring mina studier.

Sammanfattning

When developing software that is meant to be distributed over several different computers and several different networks while still working together against a common goal there is a challenge in testing how updates within a single component will affect the system as a whole. Even if the performance of that specific component increases that is no guarantee for the increased performance of the entire system. Traditional methods of testing software becomes both hard and tedious when several different machines has to be involved for a single test and all of those machines has to be synchronized as well.

This thesis has resulted in an exemplary application suite for testing distributed software. The thesis describes the method used for implementation as well as a description of the actual application suite that was developed. During the development several important factors and improvements for such a system was identified, which are described at the end of the thesis even though some of them never made it into the actual implementation. The implemented application suite could be used as a base when developing a more complete system in order to distribute tests and applications that has to run in a synchronized manner with the ability to report the results of each individual component.

Innehåll

Figurer

Tabeller

Listningar

Kapitel 1

Inledning

Pitch Technologies är ett företag som arbetar med mjukvara för simulering. De utvecklar ett flertal applikationer och tjänster för att underlätta arbetet med simulering, däribland Pitch pRTI[1], Pitch Visual OMT[2] och Pitch Developer Studio[3].

Simuleringar som körs med Pitch pRTI eller andra implementationer av High Level Architecture (HLA), en standard för simuleringar[4] kräver ett flertal olika applikationer och tjänster som dels gör anspråk på stora delar av systemets resurser, och dels ställer höga krav på kort responstid. Detta medför att olika applikationer och tjänster ofta exekveras på olika maskiner och kommunicerar över ett lokalt nätverk (LAN) eller över internet vid en fullskalig simulering. Simuleringar är inte begränsade till en specifik domänrymd utan kan röra i princip vad som helst, ett konkret exempel där HLA och Pitch Technologies produkter använts är militärövningen Viking där flera länder under flera dagar genomfört en gemensam övning med stöd av simulering [12].

Då förhållanden vid olika simuleringar skiljer sig avsevärt beroende på bland annat hårdvara, implementation av federater och nätverket är det svårt, och framför allt tidskrävande, att återskapa dessa förhållanden för att på ett korrekt sätt testa prestanda eller återskapa fel med konventionella metoder för testning och felsökning av mjukvara och programkod.

Vid utveckling av Pitch pRTI och Pitch Developer Studio, eller implementation av federater[5], är det därför svårt att mäta prestandaförändringar mellan olika version eller att aktivt arbeta för att förbättra prestandan på dessa produkter.

Med hjälp av Pitch pRTI går det att koppla samman tusentals entiteter i samma simulering oavsett om dessa kommunicerar över LAN eller WAN [9]. Även om en distribuerad simulering med ett hundratal system anses vara en stor simulering finns det utrymme för förbättringar vad gäller prestandan och möjligheten att koppla samman ytterligare system i en och samma simulering. Därför är det av stort intresse att kunna testa förändringar i prestanda på

[1]Servermjukvara som implementerar Run-Time Infrastructure i enlighet med IEEE 1516 standarden.

[2]Mjukvara för att skapa och underhålla HLA-modeller som används vid simulering.

[3]Utvecklingsmiljö med tillhörande bibliotek för att underlätta implementation av HLA-modeller.

[4]IEEE 1516-2010 [9]

[5]Klienter i ett system som implementerar HLA.

ett distribuerat system och då framför allt olika versioner av Pitch pRTI och Pitch Developer Studio för att kunna analysera hur tillägg och förändringar i produkterna påverkar prestandan beroende på vilken konfiguration de olika noderna i distributionen har.

1.1 Syfte

Kandidatarbetet syftar till att ta fram en prototyp av ett system där det går att testa samt analysera prestandaförändringar i produkter och applikationer som kommunicerar över nätverk.

1.2 Frågeställning

Att analysera produkter som används för att koppla samman stora distribuerade system kräver antingen avancerad heuristisk eller möjligheten att skapa en situation som återspeglar produktens användning i verkligheten och sedan titta på resultatet av denna användning. Vid ett test gäller det dels att kunna övervaka prestanda på den centrala server som klienter ansluter till, dels att övervaka prestanda på varje enskild klient för att kunna se skillnader mellan olika konfigurationer och versioner av samma produkt.

I Hur skall ett testfall som ska distribueras över en eller flera klienter beskrivas?

II Hur skall en grupp av testfall distribueras över ett nätverk för att maximera användningen av hårdvara?

III Hur skall en grupp av testfall distribueras över ett nätverk med variabelt antal klienter så att resultatet är tillförlitligt och jämförbart med tidigare resultat?

IV Hur skall testfall formuleras för att representera verkligheten utan att antalet testfall blir ohanterligt stort?

1.3 Avgränsningar

Examensarbetet har omfattat 16 högskolepoäng och pågått från Mars till och med Maj. Arbetet har ägt rum på Pitch Technologies i Linköping. Den slutliga produkten har inte något eget grafiskt gränssnitt i enlighet med önskemål från beställaren utan använder sig av TeamCity, ett Continuous Integration System, i den mån information måste visualiseras.

På grund av begränsad tillgång till hårdvara och virtuella maskiner genomfördes aldrig tester på fler än tre olika maskiner, en avgränsning som kom till under arbetets gång.

1.4 Målgrupp

Den som läser denna rapport antas ha grundläggande kunskaper om nätverk, distribuerade system samt programmering. Studenter vid en utbildning med inriktning på datavetenskap eller programmering, ett eller två år in i utbildningen bör ha tillräckliga kunskaper för att förstå materialet.

1.5 Språkbruk

Även om denna rapport är författad på svenska så finns det ett antal termer som saknar korrekt svensk översättning och termer som inom området de facto refereras till på engelska. Framför allt gäller detta distribuerade system och implementationer av High Level Architecture i synnerhet; dessa termer kommer därför att förekomma på engelska. Förkortningar kommer att skrivas inom parentes efter den fullständiga utskrivningen av ordet första gången de förekommer och därefter refereras till enbart som förkortning.

1.6 Disposition

I Kapitel 1 beskrivs syftet med arbetet och rapportens struktur. I Kapitel 2 beskrivs bakgrunden till arbetet och rapporten. I Kapitel 3 beskrivs den teori som legat till grund för arbetet. I Kapitel 4 beskrivs den metod som används under arbetets gång samt vilken metod som användes för programvaruutveckling. I Kapitel 5 beskrivs de resultat som arbetet lett till och sammanfattande information kring användning av systemet. I Kapitel 6 diskuteras den metod som används från Kaptel 4 samt de resultat arbetet lett till från Kapitel 5.

Kapitel 2

Bakgrund

Arbetet grundar sig i en tidigare projektbeskrivning inom företaget av ett ramverk för kontinuerlig prestandatestning i en exekverad miljö med projektnamnet CoRuPT[1]. Dokumentet beskriver mycket översiktligt en idé för ett ramverk avsett för kontinuerlig och persistent prestanda och skalbarhetstestning av Pitch pRTI.

Avsikten med projektet var att bygga en infrastruktur för att på ett enkelt och praktiskt sätt verifiera att prestanda och skalbarhet gällande Pitch pRTI kontinuerligt förbättras. En av de stora skillnaderna mellan de produkter som implementerar RTI har varit prestanda och samtidigt som de som utvecklar produkter för simulering strävar efter att uppfylla specifikationer i så hög grad som möjligt så prioriterar kunden i många fall prestanda över extra funktionalitet.

Projektet var tänkt att användas kopplat till en Continuous Integration Server (Sektion 3.4) och vid körning generera resultat som enkelt kan jämföras med fördefinierade värden för vad som är acceptabelt och tydligt visa på skillnader mellan försämrade resultat och kraftigt förbättrade resultat. Tabell 2.1 illustrerar ett exempel på möjlig utdata från ett sådant system.

Tabell 2.1: Exempel på möjligt resultat från CoRuPT

Test Case	Value	Norm	Status
Joins per sec 10 Federates	48	45	BAD
Joins per sec 100 Federates	8	12	OK
Mutual Updates per sec 10 Federates	1300	1200	BAD
Mutual Updates per sec 100 Federates	800	1100	EXCELLENT

[1]Continuous RunTime Performance Test Framework Project

Kapitel 3

Teori

Arbetet har i mångt och mycket bestått av programutveckling som skett i form av en iterativ process där beställaren stått för en stor del av kunskapen kring deras specifika behov och varit delaktig i att avgöra åt vilket håll utvecklingen skulle fortsätta. I detta kapitel finns det översiktsinformation kring den teori som ligger till grund för arbetet, teori kring den design som valts under utvecklingsprocessen och teori kring den terminologi som valts för att förklara arbetet i uppsatsen.

3.1 High-Level Architecture

Specifikationen för High-Level Architecture beskriver bland annat en uppsättning regler för hur klienter som deltar i en simulering ska kommunicera med varandra [9]. Dess huvudsakliga syfte är dels att öka interoperabilitet mellan olika programvaror för simulering, dels att underlätta återanvändning av olika simuleringsmodeller mellan olika simuleringar [11].

Federater inom samma federation kommunicerar med varandra genom att skicka meddelanden till den programvara som implementerar Run-Time Infrastructure specifikationen[1] (RTI). Det är sedan RTI som ansvarar för att dessa meddelanden levereras till alla andra federater som har specificerat att de är i behov av den typen av information. Kommunikationen går att likna vid ett Observer Pattern där instanser av flera olika klasser kan prenumerera på vissa typer av meddelanden men också skicka iväg meddelanden av en specifik typ. Det är sedan upp till den klass som ansvarar för prenumerationerna att meddela de prenumererande klasserna att ett meddelande har skickats ut.

3.2 Designmönster

Ett designmönster är en generell lösning på ett vanligt problem inom programmering och mjukvaruutveckling. Till skillnad från bibliotek eller gränssnitt är det inte färdig kod tänkt att användas för ett speciellt syfte utan en beskrivning av hur ett specifikt problem kan lösas. Till skillnad från algoritmer så beskriver ett designmönster inte hur en viss beräkning ska genomföras på ett effektivt sätt utan hur mjukvara ska designas för att undvika vanliga problem och enklare kunna implementera och underhålla mjukvaran.

3.2.1 Observer pattern

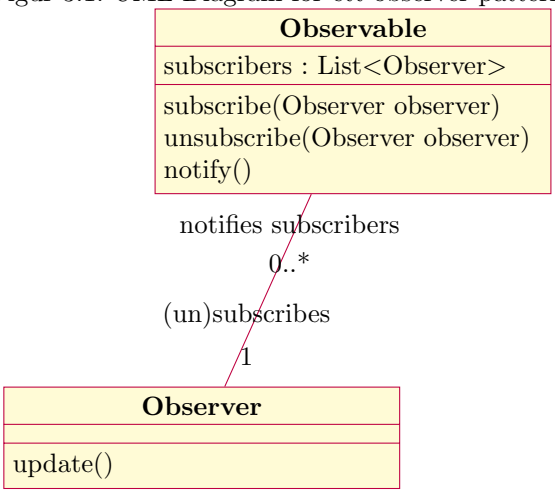
Observer pattern är ett designmönster vars huvudsakliga uppgift är att förmedla information om att en specifik händelse har inträffat till de komponenter (observatörer) som tidigare anmält intresse för den specifika händelsen. När det observerade objektet utför en viss uppgift, kommer i ett visst tillstånd eller tilldelas viss information meddelas detta till de komponenter som har registrerat intresse, vilket ger dem möjlighet att agera utifrån händelsen.

Den stora fördelen med denna typ av designmönster är att det tillåter lösa kopplingar (loose-coupling) mellan olika objekt. Detta innebär att komponenter enkelt kan läggas till, förändras eller tas bort helt och hållet utan att påverka de andra komponenterna eller programmet som helhet. När det observerade objektet väl är implementerat kommer det aldrig att behöva ändras för att tillåta ytterligare komponenter att agera utifrån dess information, förutsatt att den information som redan tillhandahålls är tillräcklig för samtliga komponenter [6].

För att ge ett exempel på detta, om vi föreställer oss att vi implementerar en enkel modell av en bil. När bilen bromsar kommer ett flertal oberoende moduler att agera på denna händelse, bromsplattor kommer att trycka mot hjulet och bromslampor kommer att tändas. Bromspedalen skulle i en sådan modell med fördel kunna implementeras som ett observerbart objekt medan

[1] IEEE 1516-2010 [9]

Figur 3.1: UML Diagram för ett observer-pattern



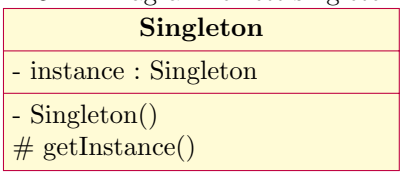
bromsplattor och bromsljus skulle kunna implementeras som observatörer. Eftersom det endast finns lösa kopplingar mellan bromspedalen, bromsplattorna och bromsljusen så kommer varken bromsplattorna eller bromsljusen påverkas ifall den andra modulen skulle sluta att fungera. Vill vi sedan implementera en ljudsignal som låter i samband med inbromsning kan denna implementeras helt oberoende av de andra modulerna.

Listning 3.1: Kodexempel för ett observer-pattern

```
public abstract class Observer {
  public abstract void notify();
}

public abstract class Observable {
  private List<Observer> subscribers;

  public Observable() {
    subscribers = new ArrayList<Observer>();
  }

  public void subscribe(Observer observer) {
    subscribers.add(observer);
  }

  public void unsubscribe(Observer observer) {
    subscribers.remove(observer);
  }

  public void notify() {
    for (Observer subscriber : subscribers) {
      subscriber.notify();
    }
  }
}
```

3.2.2 Singleton pattern

Figur 3.2: UML Diagram för ett singleton-pattern



Singleton pattern är ett designmönster som karaktäriseras av att det endast kan finnas en instans av den klass som implementerar designmönstret. Alla försök att skapa ytterligare instanser av en klass som implementerar detta designmönster resulterar i en referens till samma faktiska instans. Detta designmönster är användbart då det bara behövs, och bara ska existera, en enda instans av ett objekt för att till exempel koordinera olika händelser eller tillstånd i ett system.

Den stora fördelen med denna typ av designmönster är just att det garanterar att endast en instans existerar, och dessutom garanterar att denna instans är åtkomlig från vilken del av programmet som helst. Detta är fördelaktigt när funktionalitet så som Thread pools, cache, objekt som hanterar användarpreferenser, loggning med mera [6]. Även om det finns flera andra designmönster som skulle kunna användas för denna typ av funktionalitet kan en singleton i många fall underlätta implementation ifall behovet av att komma åt en specifik resurs uppkommer långt efter designfasen.

Det har riktats kritik emot användande av detta designmönster. Den huvudsakliga kritiken riktar sig mot att användande av designmönstret singleton inför en ny, global, referens som i många fall inte skulle vara nödvändig och skulle kunna undvikas med en bättre design.

Listning 3.2: Kodexempel för ett singleton-pattern

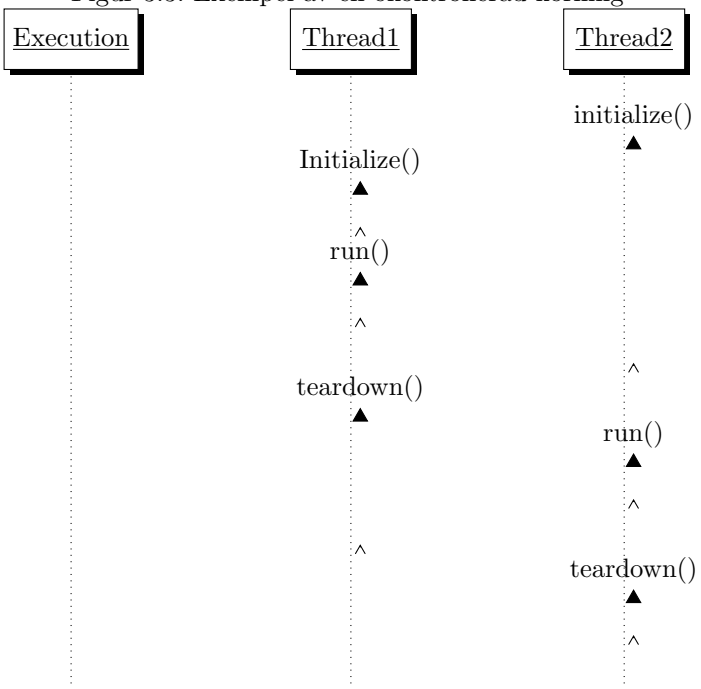
```
public class Singleton {
  private static Singleton instance = null;

  private Singleton() {

  }

  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }

    return instance;
  }
}
```

3.2.3 Semafor

Eftersom en funktion eller instruktion i ett högnivå-språk i regel innehåller flera instruktioner på processor-nivå och trådar och processer exekverar i en icke-deterministisk ordning går det inte att säga huruvida en funktion i en tråd avslutats innan en funktion i en annan tråd påbörjat exekvering. Det är viktigt att poängtera att funktioner i detta sammanhang inte endast avser metoder utan även programkonstruktioner så som villkors-satser och tilldelningar. På grund av detta icke-deterministiska beteende finns det risk för att ett konkurrenstillstånd uppstår (race condition).

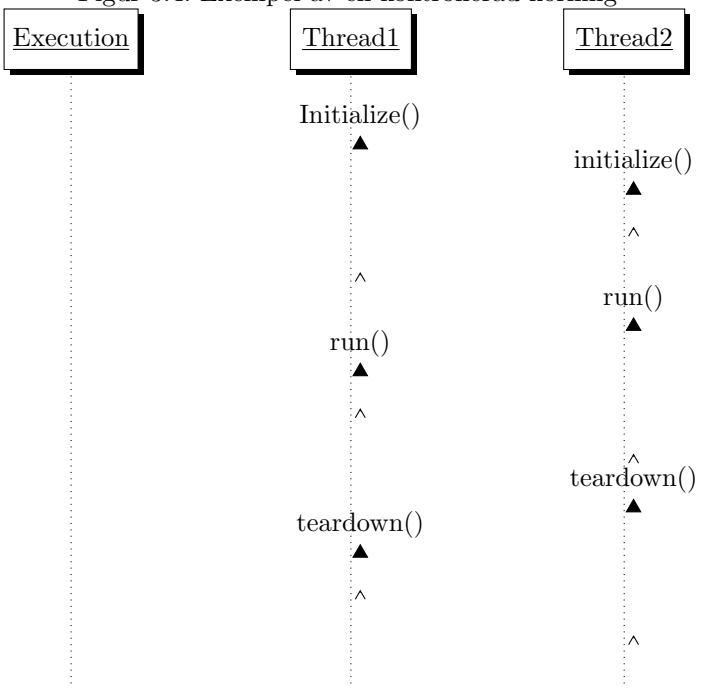
En semafor är en mekanism i ett operativsystem eller exekveringsmiljö som tillhandahåller möjligheten att begränsa åtkomst till resurser eller funktionalitet mellan olika trådar. Metoder som opererar på semaforer är implementerade som atomära funktioner i operativsystemet eller exekveringsmiljön för att garantera att ingen annan tråd eller process än den som opererar på semaforen vid en given tidpunkt har möjlighet att exekvera några instruktioner.

Figur 3.3: Exempel av en okontrollerad körning



Tittar man till exempel på Figur 3.3 så är det tydligt att ifall `Thread2` är beroende av `Thread1` i metoden `run()` är det inte säkert att detta beroende kan uppfyllas eftersom metoden `run()` i `Thread1` exekveras samtidigt som metoden `teardown()` i `Thread2`. Tittar man istället på Figur 3.4 så ser man att metoderna `initialize()`, `run()` och `teardown()` kommer att exekveras parallellt med varandra, metoden `initialize()` kommer att avslutas i samtliga trådar innan metoden `run()` påbörjas och metoden `run()` kommer att avslutas i samtliga trådar innan metoden `teardown()` påbörjas.

Figur 3.4: Exempel av en kontrollerad körning



Listning 3.3: Kodexempel för användning av en semafor

```
public abstract class Something {
  private static Semaphore s = new Semaphore(5);

  public static void doSomething() {
    s.aquire(2);
    Thread.sleep(5000);
    s.release(2);
  }
}

public class myThread extends Thread {
  @Override
  public void run() {
    while (true) {
      Something.doSomething();
    }
  }
}

public class Application {
  public static void main(String[] args) {
    (new myThread()).start();
    (new myThread()).start();
    (new myThread()).start();
  }
}
```

3.3 Testning

Termen testning har många olika betydelser för olika personer vilket kan beskrivas som fem olika faser. *Fas 0*, där det inte finns någon upplevd skillnad mellan testning och felsökning utan testning endast ses som ett hjälpmedel för att felsöka. *Fas 1*, där syftet med testning anses vara att visa att programvaran fungerar. *Fas 2*, där syftet med testning till skillnad från fas 1 anses vara att visa att programvaran faktiskt inte fungerar enligt specifikationen. *Fas 3*, där syftet med att testa programvara inte nödvändigtvis är att bevisa något, utan att minska den upplevda risken att programvaran inte fungerar över en acceptabel nivå. *Fas 4* där det upplevda syftet med testning har gått från att vara ett verktyg för att uppnå något annat till att i sig bli ett självändamål som resulterar i att bättre programvara blir utvecklad utan att särskilt mycket testning är nödvändig [4].

Att testa mjukvara definieras av IEEE som "Processen att exekvera ett system eller en komponent under specifika förhållanden, observera eller spela in resultatet och [baserat på resultaten] göra en bedömning av någon eller några aspekter av systemet eller komponenten"[2] [8]. De specifika förhållanden som omnämnds i definitionen är de förhållanden som beskrivs i, och utgör, ett testfall.

Det finns två distinkta metoder för att testa programvara, **Whitebox testing** som beskrivs i Sektion 3.3.1 och **Blackbox testing**, utöver dessa två nämns ibland även ***Greybox testing*** som då refererar till ett ospecificerat snitt av de tidigare nämnda metoderna. Då både programsviten och uppsatsen till stor del handlar om Whitebox Testing beskrivs detta ytterligare i Sektion 3.3.1 medan Blackbox testing i sammanhanget inte är särskilt relevant. Till dessa metoder finns dessutom ett flertal tekniker, mätvärden och tillvägagångssätt för att genomföra den faktiska testningen av programvara som inte kommer att presenteras i denna uppsats, den intresserade läsaren kan hitta mer information i Lee Copelands bok *A Practitioner's Guide to Software Test Design* [5]

3.3.1 Whitebox testning

Whitebox testing är ett metod där testningen baseras på implementationen av programvaran eller komponenten som testas snarare än specifikationen. Eftersom testaren har vetskap om implementationen fokuserar whitebox testing på att testa de olika vägar kontrollflödet och dataflödet kan ta genom en komponent eller mellan olika komponenter för att på så vis komma åt saker som inte nödvändigtvis står specificerat i specifikationen.

3.4 Continuous Integration System

Continuous Integration (CI) är en typ av system som används inom mjukvaruutveckling för att sammanfoga olika utvecklares bidrag till kodbasen med en ytterligare möjlighet att vidta åtgärder som att kompilera, exekvera tester, skicka mail med mera när kodbasen förändras. Skillnaden på ett vanligt byggsystem så som Make, Ant eller Maven och ett CI-system är framför allt automatise-

[2]Fritt översatt från Engelska

ringen av arbetet med att bygga och regressionstesta den kod som kopplas till systemet.

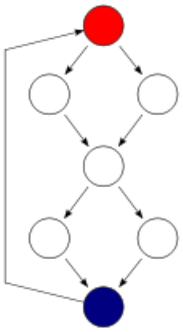
Jenkins[3] är ett av de mer kända och väl använda system för Continuous Integration och innehåller bland annat stöd för att ansluta till revisionshanteringssystem så som subversion, git, mercurial och CSV, genomföra unit-testing och skicka mail eller sms ifall något inte skulle fungera när kod från olika utvecklare sammanfogas. Utöver den inbyggda funktionaliteten finns det ett stort utbud av plugins och moduler med över 400 officiellt stödda plugins [1]. För en grundläggande förståelse för continuous integration kan Jenkins vara en bra utgångspunkt då det både är väl använt, enkelt att själv installera och bygger på öppen källkod. I projektet har TeamCity[4] använts då det redan var en del av beställarens infrastruktur.

3.5 Cyklomatisk Komplexitet

Cyklomatisk komplexitet (Cyclomatic Complexity) är ett mätetal för källkod som syftar till att indikera hur komplext ett stycke kod är baserat på antalet möjliga ingångar och utgångar från stycket. Genom att representera källkoden som ett kontroll-flödes-diagram (Control Flow Graph, CFG) där varje hörn/nod i den riktade grafen motsvarar ett stycke kod som alltid exekveras sekventiellt och varje kant/båge representerar utgången från ett stycke kod och engången till ett annat [10].

Eftersom ett stycke kod med en kant från en senare del av koden till en tidigare del av koden potentiellt har ett oändligt antal vägar från påbörjad exekvering till avslutad exekvering beräknas cyklomatiskt komplexitet baserat på simpla vägar som tillsammans utgör alla möjliga vägar genom koden. Det cyklomatiska numret $v(G)$ för en graf G med n noder, e kanter och p sammankopplade komponenter (vägar tillbaka) beräknas enligt $v(G) = e - n + p$ [10].

Figur 3.5: CFG föreställande två if-else satser[4]



I Figur 3.5 är det antalet noder 7, antalet kanter 9 och antalet sammankopplade komponenter 1. Det cyklomatiska numret blir således $v(G) = 9 - 7 + 1 \Leftrightarrow$

[3] `http://jenkins-ci.org/`
[4] `http://www.jetbrains.com/teamcity/`
[4] Hämtad från `http://en.wikipedia.org/wiki/` 2013-05-07

$v(G) = 3$. Cyklomatisk komplexitet visar på ett relativt enkelt sätt vilka delar av källkod som innehåller stora delar av den logik som är viktig för systemet. I uppsatsen redovisas den cyklomatiska komplexiteten som en referenspunkt till vilka delar av systemet som är särskilt relevanta för den funktionalitet som krävs när det kommer till att distribuera och exekvera testfall.

Kapitel 4

Metod

Då projektet inte hade någon tydlig specifikation eller tydliga leverabler har arbetet skett med relativt tät kontakt med kunden som har kommit med åsikter och tankar på hur programsviten bör förändras och vidareutvecklas för att passa deras behov i så stor utsträckning som möjligt. På grund av detta arbetssätt har Scrum (Sektion 4.1) och Test-driven utveckling (Sektion 4.2) använts i den mån det gått att följa under rådande omständigheter. Programsviten har utvecklats som ett enmans-projekt vilket begränsat möjligheterna till att dela upp roller inom Scrum. Test-driven utveckling har valts som arbetsmetod för att garantera att det finns testfall för att verifiera fortsatt funktionalitet vid förändrade villkor från kunden.

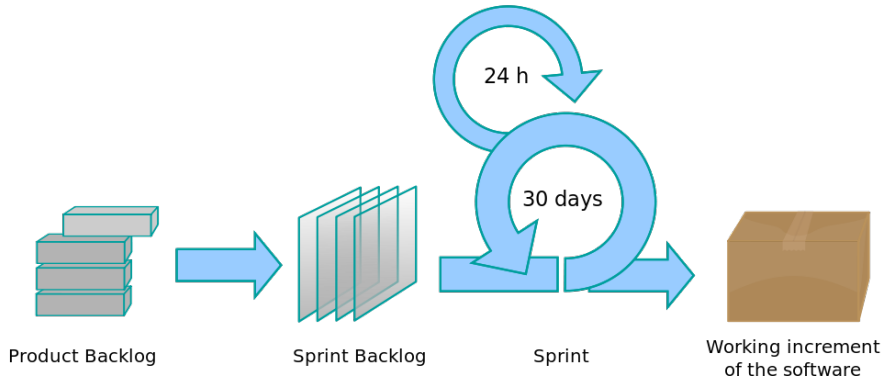
4.1 Scrum

Scrum är en agil utvecklingsmetodik som först beskrivs som ett holistisk tillvägagångssätt för att utveckla programvara med sex karaktäristika; inbyggt instabilitet, självorganiserande projektgrupper, utvecklingsfaser som överlappar varandra, multipelt lärande (multilearning), diskreta kontroller och organisatoriskt överföring av kunskap [14].

Det finns tre huvudsakliga roller inom Scrum och ytterligare några mer informella roller och uppgifter. **Produktägaren** (Product Owner) är den person som representerar kundens perspektiv och ansvarar för att projektgruppen levererar det kunden faktiskt har efterfrågat. **Utvecklingsgruppen** (Development Team) är den grupp som är ansvarig för att leverera en (oftast) fungerande produkt som är färdig att driftsättas i slutet av varje utvecklingscykel. **Scrum-Master** är den person som ansvarar för kontakt med kunden och andra intressenter för projektet så att utvecklingsgruppen kan arbeta ostört och klara av att leverera en färdig produkt i tid.

Utvecklingsprocessen är uppdelad i mindre cykler (sprints) där varje cykel innehåller en ny uppsättning fuktionalitet och förändringar (sprint backlog) som ska levereras i slutet av cykeln.

Figur 4.1: Modell över Scrum-processen[1]

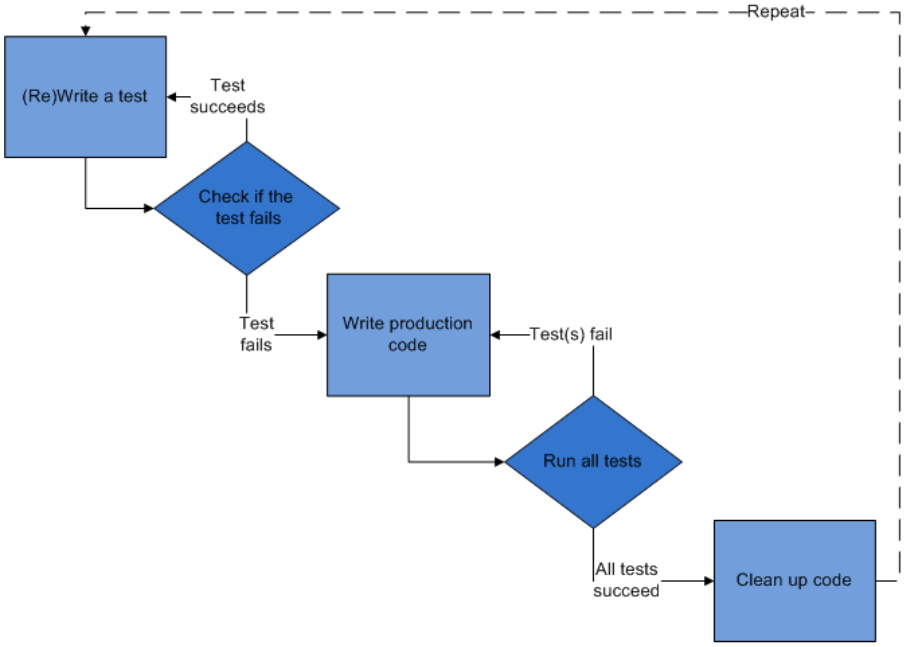


4.2 Test-Driven Development

Test-driven utveckling (Test-Driven Development, TDD) baseras på att först beskriva en del av funktionaliteten som ett test och sedan implementera den beskrivna delen av funktionalitet med testet som specifikation. När samtliga test passerar utan varningar eller fel skrivs ytterligare test som beskriver ny funktionalitet och processen återupprepas tills dess att all den behövda funktionaliteten har blivit implementerad och testad.

[1] Hämtad från `http://en.wikipedia.org/wiki/` 2013-05-06

Figur 4.2: Modell över processen för test-driven utveckling[2]



Trots att grunden inom test-driven utveckling endast specificerar att test ska skrivas före funktionalitet implementeras så får det till konsekvens att utvecklare tvingas tänka på hur funktionaliteten eller komponenten kommer att användas, och först efter att de beskrivit testfall som är menade att testa den tänkta funktionaliteten faktiskt implementerar något. Detta innebär i realiteten att det är en teknik för design såväl som implementation och främjar utveckling av komponenter som är enklare att underhålla och enklare att testa än i annat fall [13]. Undersökningar har visat på att individuella utvecklare och projektgrupper som inkorporerar test-driven utveckling i sitt arbete både skriver fler testfall, blir mer produktiva och producerar bättre programkod [7].

[2]Hämtad från `http://en.wikipedia.org/wiki/` 2013-05-06

Kapitel 5

Resultat

Arbetet har resulterat i en programsvit bestående av fyra applikationer; Pitch Performance Master, Pitch Performance Slave, Pitch Performance Development och Pitch Performance Framework. Pitch Performance Master och Pitch Performance Slave är de två applikationer som är ämnade att användas för testning av distribuerade system, Pitch Performance Framework är det bibliotek som används för att skriva de testfall som senare ska exekveras med hjälp av de förstnämnda. Pitch Performance Development är ett stödverktyg som tagits fram för att underlätta utveckling och testning av de testfall som skrivs, syftet med Pitch Performance Development är att en utvecklare inte ska behöva koppla upp en hel testmiljö och se till att applikationer, bibliotek och filer finns tillgängliga på nätverket för att manuellt kunna kontrollera ifall ett testfall fungerar som det är tänkt eller inte.

Tabell 5.1: Rader kod per applikation

Applikation	Rader kod	Klasser	Metoder
PP Master[1]	1 507	7	55
PP Slave[2]	206	3	10
PP Framework[3]	287	3	17
PP Development[4]	241	3	10
Communication Handler	821	23	55
Scenario	1 769	16	90
Settings	315	4	17
Totalt	5 146	59	254

[1]Pitch Performance Master
[2]Pitch Performance Slave
[3]Pitch Performance Framework
[4]Pitch Performance Development

5.1 Systemöversikt

Pitch Performance Master är den applikation i programsviten som står för distribution och koordinering av applikationer och data nödvändigt för att genomföra ett test, testsviter och scenarion. **Pitch Performance Slave** är den applikation i programsviten som står för exekvering av ett enskilt scenario. I systemet är ett test en grupp instruktioner som definierar vilka komponenter som behöver hämtas och vad som ska exekveras på en specifik instans av Pitch Performance Slave; ett scenario är en grupp av tester och en testsvit är en grupp av scenario. Dessa begrepp finns ytterligare förklarade i Sektion 5.6.

Ett exempel, om än något haltande, skulle kunna vara relationen mellan flygledningen och ett antal flygplan där flygledningen motsvarar Pitch Performance Master medan flygplanen motsvarar Pitch Performance Slave. Även om det är flygplanen som genomför det faktiska arbetet med att flytta passagerare från position A till position B så är det flygledningen som avgör när (om) ett flygplan får lyfta, koordinerar höjd mellan olika flygplan, godkänner färdriktning och avgör när (om) ett flygplan får landa.

Eftersom antalet Pitch Performance Slave som behövs för ett givet test varierar och antalet fysiska maskiner som finns tillgängliga för att exekvera ett givet test också kan variera är nätverket uppbyggt enligt samma princip angående lösa kopplingar som observatörsmönstret beskrivet i Sektion 3.2.1. Pitch Performance Master konfigureras med en lista av möjliga Pitch Performance Slave instanser och deras IPv4 adress och kontrollerar sedan innan exekvering av varje testsvit vilka av dessa instanser som finns tillgängliga och dessutom vilka av dessa instanser som matchar de krav som är specificerade för varje testfall.

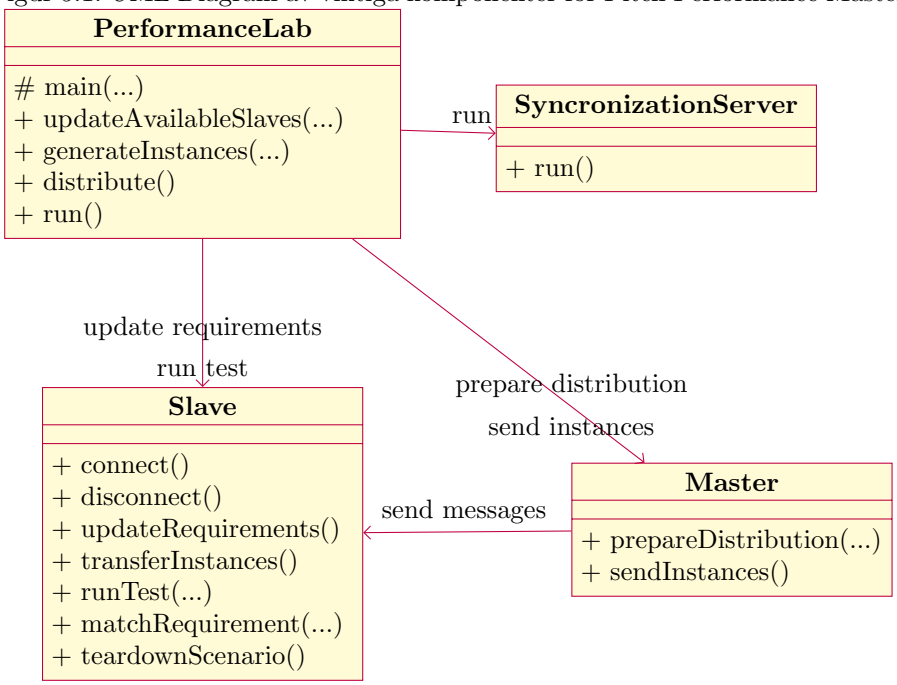
För att varje testfall skall exekveras på ett deterministiskt sätt finns det anledning att specificera vilken typ av hårdvara och miljö som ska användas vid exekvering av ett visst testfall.

För att ge ett exempel så skulle resultatet för ett test där syftet är att mäta hur många anrop per sekund en servermjukvara kan hantera innan den överbelastas kraftigt differentiera ifall servermjukvaran vid ett tillfälle exekverades på en Raspberry Pi ansluten till nätverket med hjälp av ett 3G modem och vid ett annat tillfälle exekverades på en MacMini ansluten till nätverket med en vanlig TP-kabel. Resultatet av test exekverade med så olika förutsättningar skulle inte alls vara jämförbart och det skulle vara mycket svårt, för att inte säga omöjligt, att dra slutsatser kring huruvida förändringar i servermjukvaran påverkat presentandan till det positiva eller negativa.

Pitch Performance Framework är det bibliotek i programsviten som innehåller de stödfunktioner och den initialisering som behövs för att skriva testfall som sedan ska exekveras med hjälp av Pitch Performance Slave via Pitch Performance Master. Modellen för att skriva testfall med hjälp av Pitch Performance Framework är baserat på JUnit och CppUnit där `initialize` (`setup`), `run` (`test*`) och `teardown` ansvarar för att initialisera testet, exekvera testet, och städa upp efter testet.

5.2 Pitch Performance Master

Figur 5.1: UML Diagram av viktiga komponenter för Pitch Performance Master



Pitch Performance Master är det program i programsviten som distribuerar och koordinerar test till Pitch Performance Slave, beskrivet i Sektion 5.3. Pitch Performance Master är den huvudsakliga applikationen som också innehåller majoriteten av all funktionalitet för programsviten.

Tabell 5.2: Sammanfattning av Pitch Performance Master

Klass	Metod	Rader kod	CC[5]
PerformanceLab	main	37	10
ProChoice	process	29	9
CMH[6]	receiveMessage	17	7
CMH[6]	reportPhase	37	6
ProChoice	setMode	13	5
Slave	matchRequirement	20	5
PerformanceLab	run	36	4
Master	prepareDistribution	22	4
PerformanceLab	updateAvailableSlaves	19	3
PerformanceLab	generateInstances	18	3

[5]Cyklomatisk Komplexitet (Cyclomatic Complexity)
[6]CoordinationMessageHandler

Algoritm för exekvering av Pitch Performance Master

I Upprätta en anslutning till samtliga Pitch Performance Slave.

II För varje testsvit:

III Hitta en passande slav för varje scenario.

IV Skicka alla scenarion till respektive slavar.

V Starta en instans för nätverkssynkronisering.

VI För varje testfall:

VII Vänta tills samtliga slavar har anslutit för synkronisering.

VIII Vänta tills samtliga slavar har genomfört initialiseringsfasen.

IX Vänta tills samtliga slavar har exekverat sitt scenario av testfallet.

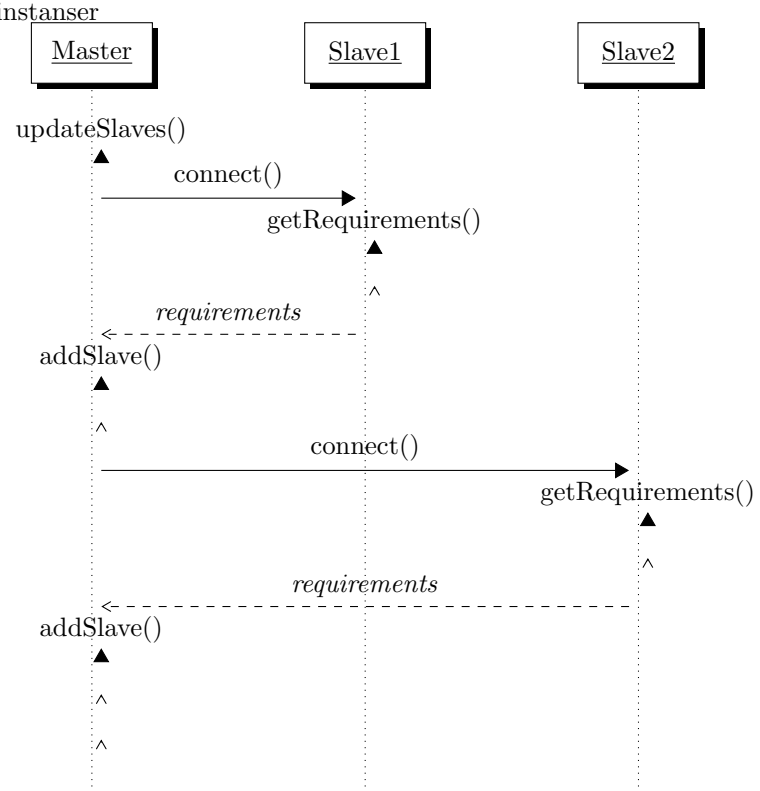
X Vänta tills samtliga slavar har städat upp efter sitt scenario.

XI Avsluta anslutningen till samtliga slavar.

5.2.1 Koppla upp nätverket inför test

Pitch Performance Master läser först in en lista med information om *namn* som identifierar den specifika instansen, *IPv4 adress* till den maskin som exekverar den specifika instansen, *krav* som uppfylls av den maskin och den exekveringsmiljö där den specifika instansen exekveras. Därefter försöker Pitch Performance Master upprätta en anslutning till varje Pitch Performance Slave som finns beskriven i den lista som lästes in och kontrollerar ifall några krav har tillkommit, ändrats eller försvunnit från instansen. De instanser av Pitch Performance Slave som är kontaktbara och dessutom kan tillhandahålla en lista med krav läggs till i ett register med tillgängliga Pitch Performance Slave instanser.

Figur 5.2: Pitch Performance Master initiala kontakt med Pitch Performance Slave instanser



5.2.2 Distribuera test-scenarion

För att hitta en passande Pitch Performance Slave till varje scenario går Pitch Performance Master igenom varje slav för varje scenario tills dess att någon Pitch Performance Slave matchar kraven för det scenario som kontrolleras. Kraven för ett scenario specificeras som fritext och ifall en instans av Pitch Performance Slave inte alls har det specifika kravet beskrivet antar Pitch Performance Master att kraven inte kan uppfyllas. I de fall då det finns flera instanser av Pitch Performance Slave som uppfyller kraven så väljs den instans med färst scenarion.

Eftersom vissa scenarion måste köra på hårdvara dedicerad för just det scenariot kan ett scenario kräva en reserverad instans av Pitch Performance Slave och sedan neka följande scenarion från att samköra på den reserverade instansen. Eftersom vissa testsviter och testfall kan kräva tusentals scenarion måste reservation specificeras explicit och dessutom beskrivas bland de första scenarion i testfallet.

Ifall att Pitch Performance Master inte lyckas matcha varje scenario till en Pitch Performance Slave så kommer testet att avbrytas.

Listning 5.1: Pitch Performance Master distribution av scenarion

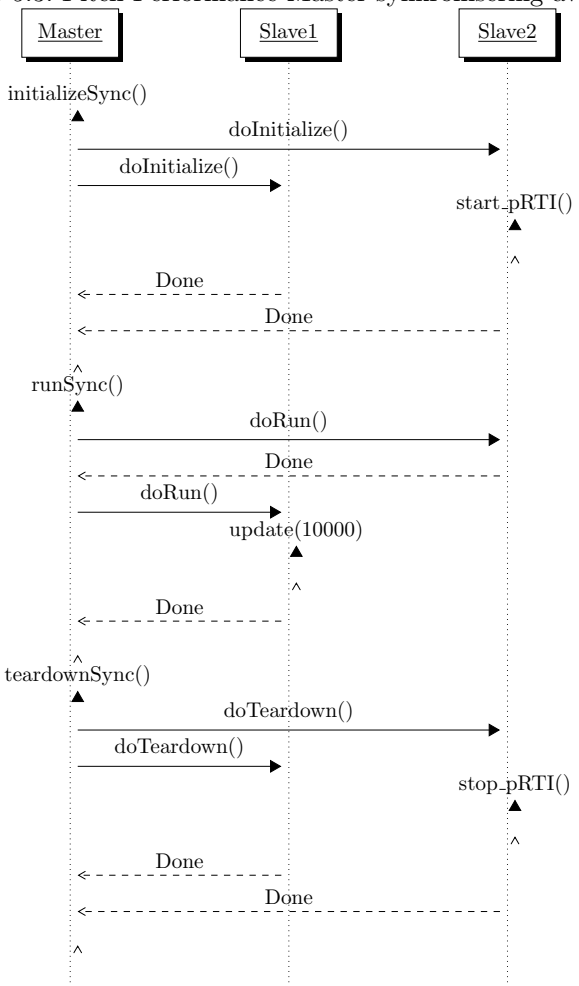
```
public class Master {
  public boolean prepare(List<Scenario> instances) {
    int count = 0;
    for (Scenario instance : instances) {
      Slave match = null;
      for (Slave slave : slaves) {
        if (slave.isMatch(instance)) {
          if (slave.size() < match.size()) {
            match = slave;
          }
        }
      }
    }
  }
}
public class Slave {
  public boolean canAccept(Scenario scenario) {
    if (scenario.getReserved()) {
      return !getReserved() && instances.isEmpty();
    }

    return !getReserved();
  }

  public boolean matchReq(String name, String val) {
    if (val == null) {
      return hasReq(name);
    }

    Iterator it = requirements.iterator();
    boolean result = false;
    while (!result && it.hasNext()) {
      Parameter p = (Parameter) it.next();
      if (p.getName().equals(name)) {
        result = p.getValue().equals(val);
      }
    }

    return result;
  }

  public boolean isMatch(Scenario scenario) {
    boolean result = this.canAccept(scenario);
    Iterator it = scenario.getRequirements().iterator();
    while (result && it.hasNext()) {
      Parameter p = (Paramter) it.next();
      result = matchReq(p.getName(), p.getValue());
    }

    return result;
  }
}
```

5.2.3 Synkronisera test-scenarion

Eftersom de applikationer som huvudsakligen var avsedda att testas kräver olika mycket tid för att genomföra varje fas av ett test och applikationerna dessutom beror på varandra i mycket stor grad har det varit viktigt att kunna synkronisera varje distinkt fas av ett testfall; initialisering där resurser allokeras, anslutningar upprättas och inledande beräkningar genomförs, exekvering där det specifika testfallet genomförs och slutligen nedrivning (teardown) där anslutningar avslutas och resurser frigörs.

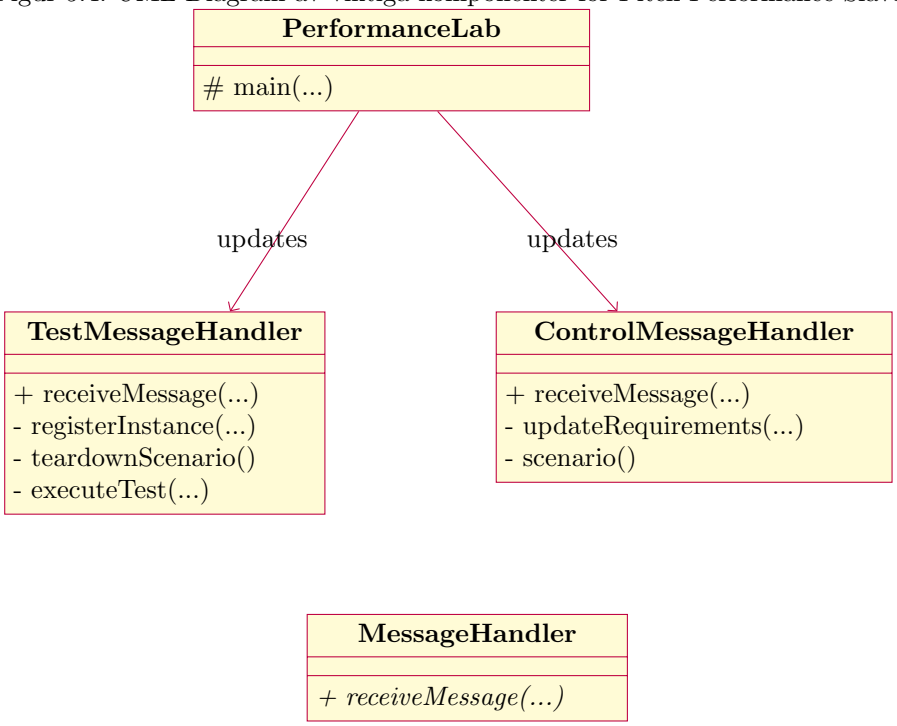
Ett realistiskt exempel på ett testfall skulle kunna vara att starta upp en pRTI på en dedicerad server och sedan starta upp 10 federater som ansluter till pRTI och skickar 10 000 uppdateringar var innan de kopplar från och testet är färdigt. För ett sådant testfall blir det givetvis väldigt viktigt att den Pitch Performance Slave som ansvarar för att starta upp pRTI har möjlighet att göra detta redan under initialisering så att den finns tillgänglig för att ta emot anslutningar när federaterna väl påbörjar sin exekveringsfas.

Figur 5.3: Pitch Performance Master synkronisering av test



5.3 Pitch Performance Slave

Figur 5.4: UML Diagram av viktiga komponenter för Pitch Performance Slave



Pitch Performance Slave är den applikation i programsviten som står för exekvering av ett eller flera scenario i ett testfall. Pitch Performance Slave svarar på kommunikation från Pitch Performance Master, beskriven i Sektion 5.2, och agerar utifrån dessa kommandon och instruktioner. I Tabell 5.3 finns en sammanfattning över de tio metoder med högst cyklomatisk komplexitet för applikationen Pitch Performance Slave.

Tabell 5.3: Sammanfattning av Pitch Performance Slave

Klass	Metod	Rader kod	CC[7]
TestMessageHandler	registerInstance	9	5
TestMessageHandler	receiveMessage	14	4
CMH[8]	receiveMessage	14	4
PerformanceLab	main	30	3
TestMessageHandler	executeTest	6	2
PerformanceLab	initializeXStream	4	1
TestMessageHandler	teardownScenario	3	1
CMH[8]	scenario	1	1
CMH[8]	updateRequirements	2	1
CMH[8]	reboot	1	1

När Pitch Performance Slave startar så registreras en `ControlMessageHandler` som tar hand om grundläggande kommandon som skickas från Pitch Performance Master; `UpdateRequirement`, `Scenario`, `Reboot`. Därefter registreras en `TestMessageHandler` som tar hand om test-specifika kommandon från Pitch Performance Master. Slutligen öppnas en nätverkssocket för att lyssna efter inkommande anslutningar från Pitch Performance Master.

När en anslutning upprättats med Pitch Performance Master och ett meddelande tas emot så kommer den eller de instanser som registrerat ett intresse för meddelandet att få tillgång till det.

När meddelandet `UpdateRequirement` skickas så kommer `ControlMessageHandler` att ladda den lokala konfigurationsfilen med uppfyllda krav och skicka tillbaka dessa till Pitch Performance Master.

När meddelandet `Scenario` skickas så kommer `ControlMessageHandler` att skicka tillbaka en *ping* till Pitch Performance Master för att bekräfta att meddelandet tagits emot, `TestMessageHandler` kommer att registrera scenariot som en del av testet den aktuella instansen av Pitch Performance Slave ska genomföra och därmed ladda hem allt material i form av applikationer, bibliotek och filer som krävs för scenariot, exekvera de applikationer som krävs för scenariot (till exempel kompilatorer) och slutligen skicka ett meddelande till Pitch Performance Master om att scenariot blivit bearbetat.

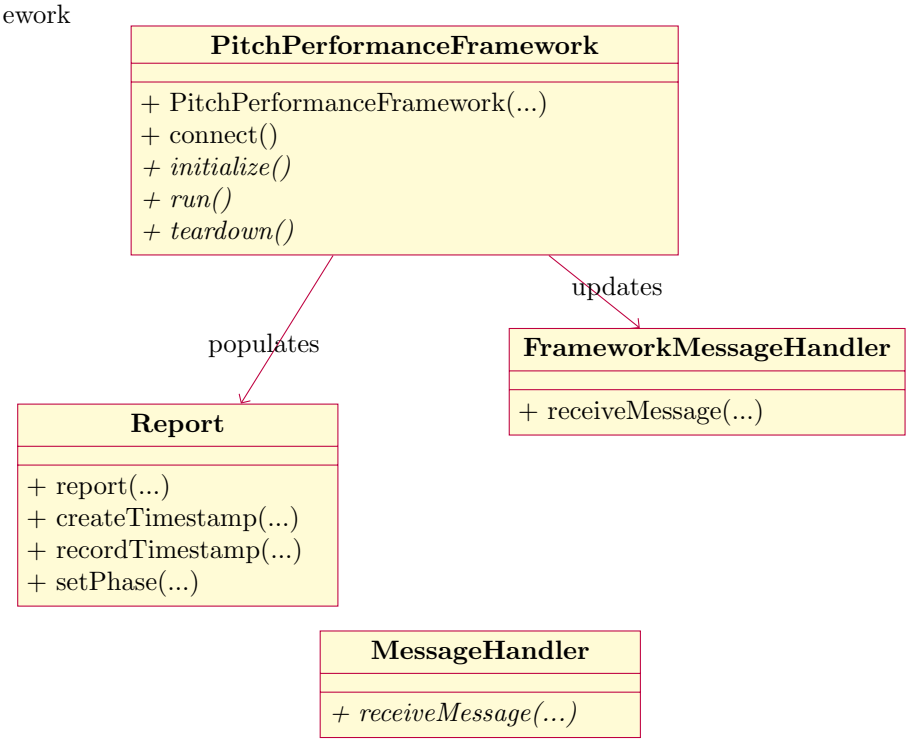
När meddelandet `ExecuteTest` skickas så kommer `TestMessageHandler` att för varje scenario som tillskrivits den aktuella instansen av Pitch Performance Slave exekvera scenariot i en separat tråd. När ett scenario exekveras kommer det att upprätta en anslutning till Pitch Performance Master synkroniseringsserver som finns beskriven i Sektion 5.2.3 och varje scenario kommer därefter att genomgå samtliga faser av testet, förutsatt att allt går som förväntat.

På grund av begränsningarna beskrivna i Sektion 1.3 har det varit nödvändigt att samköra flera instanser av Pitch Performance Slave på samma hårdvara och operativsystem. Utöver ofrånkomliga problem så som lagringsutrymme, arbetsminne (RAM) och processorkraft har en sådan miljö inte uppvisat några särskilda problem i jämförelse med en miljö där endast en instans av Pitch Performance Slave exekveras.

[7] Cyklomatisk Komplexitet (Cyclomatic Complexity)
[8] CoordinationMessageHandler

5.4 Pitch Performance Framework

Figur 5.5: UML Diagram av viktiga komponenter för Pitch Performance Framework



Pitch Performance Framework är det bibliotek som används för att skriva test som sedan ska exekveras med hjälp av programsviten. Metoderna `initialize()`, `run()` och `teardown()` är abstrakta metoder och måste implementeras för varje test som avses exekveras med hjälp av Pitch Performance Lab. Det finns ett flertal intressanta mätvärden för varje del av ett test, däribland allokerat minne, processoranvändning och tidsåtgång. Pitch Performance Framework har inbyggt stöd för att kontrollera denna typ av parametrar, även om enbart tidsåtgång implementerades till fullo under projektets gång.

Tabell 5.4: Sammanfattning av Pitch Performance Framework

Klass	Metod	Rader kod	CC[9]
FrameworkMessageHandler	receiveMessage	31	10
PitchPerformanceFramework	internalRun	19	2
PitchPerformanceFramework	connect	13	1
PitchPerformanceFramework	internalInitialize	5	1
Report	recordTimestamp	3	1
Report	report	2	1
Report	addParameter	1	1
Report	constructHeader	1	1
Report	createTimestamp	1	1

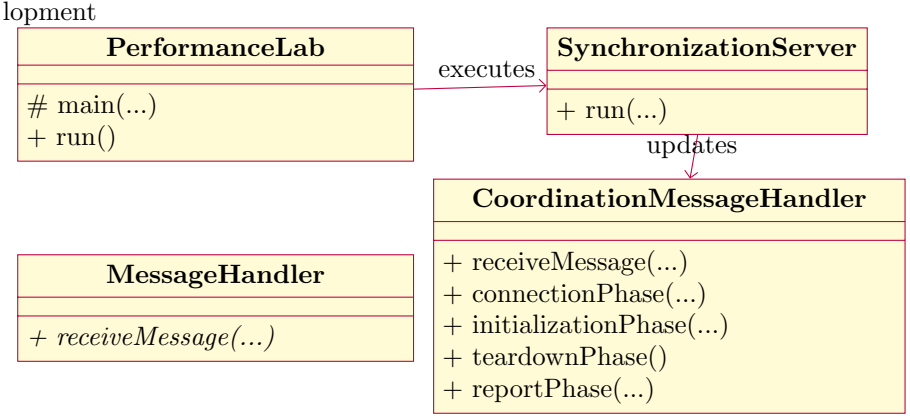
Metoden `initialize()` är ämnad att användas för att koppla upp stödtjänster, upprätta anslutningar till externa resurser och allokera resurser som är relevanta för att genomföra ett specifikt test, metoden `run()` är ämnad att användas för att genomföra det faktiska testet och metoden `teardown()` är ämnad att användas för att stänga ner eventuella stödtjänster, koppla från eventuella externa resurser och rensa upp kvarvarande objekt.

Efter att ett test genomförts till fullo sammanställer Pitch Performance Framework automatiskt en partiell rapport över de mätvärden som kontrollerats för de scenario som exekverats och skickar tillbaka denna rapport till Pitch Performance Master.

Till exempel så skulle ett test där utvecklaren avser testa hur snabbt det går att genomföra 10 000 registreringar av objekt till pRTI från n olika maskiner kunna utformas så att metoden `initialize()` används för att upprätta en anslutning till pRTI, metoden `run()` skapar 10 000 objekt samt registrerar dessa och metoden `teardown()` avregistrerar objekten och avslutar anslutningen till pRTI. En rapport för tidsåtgång för metoderna `initialize()`, `run()`, `teardown()` skulle sedan generaras och återrapporteras till Pitch Performance Master där en sammanställning för samtliga instanser som varit delaktiga i testet skulle ske.

5.5 Pitch Performance Development

Figur 5.6: UML Diagram av viktiga komponenter för Pitch Performance Development



Pitch Performance Development är ett utvecklingsverktyg som tagits fram i samband med resterande komponenter för att underlätta utvecklingen av testfall. Pitch Performance Development utför samma uppgift som Pitch Performance Master, beskriven i Kapitel 5.2, applikationen skiljer sig framför allt på två punkter då den för det första inte läser in några testfall och fördelar scenarion utan istället blint väntar på att någon skall ansluta och kommunicera som att den vore en Pitch Performance Master. För det andra accepterar den bara att ett enda test ansluter för synkronisering, vilket i realiteten innebär att ingen synkronisering är nödvändig.

[9] Cyklomatisk Komplexitet (Cyclomatic Complexity)

Tabell 5.5: Sammanfattning av Pitch Performance Development

Klass	Metod	Rader kod	CC[10]
CoordinationMessageHandler	receiveMessage	13	6
CoordinationMessageHandler	initializationPhase	14	2
CoordinationMessageHandler	runPhase	14	2
CoordinationMessageHandler	connectionPhase	14	2
CoordinationMessageHandler	teardownPhase	14	2
PerformanceLab	main	7	2
SynchronizationServer	run	32	2
PerformanceLab	initializeXStream	6	1
PerformanceLab	run	5	1

Ett testfall exekverar därför min minimal fördröjning mellan de olika faserna, och en testklass utvecklad i Java som ärver från Pitch Performance Framework, beskrivet i Kapitel 5.4, kan exekveras direkt från en utvecklingsmiljö så som Netbeans[11], Eclipse[12] eller IntelliJ IDEA[13].

5.6 Konfiguration

Eftersom programsviten är ämnad att fungera tillsammans med ett Continuous Integration System och kontrolleras utan eget gränssnitt har konfiguration varit centralt för att testfall enkelt ska kunna utvecklas och användas i en automatiserad mijlö. Då beställaren använder sig av TeamCity för att automatisera testning och kompilering av system och subversion för att hantera olika revisioner av programkod har det varit naturligt att fokusera på dessa två plattformar även för den programsvit som utvecklats i detta projekt.

Konfigurationen består av tre olika delar, konfiguration för tillgängliga Pitch Performance Slave, konfiguration för en testsvit och rapport till TeamCity. Konfigurationen för tillgängliga Pitch Performance Slave och för testsviter sker manuellt medan programsviten tar hand om rapportering till TeamCity på egen hand. Samtliga konfigurationer skrivs i XML[14].

5.6.1 Konfiguration för Pitch Performance Slave

Varje Pitch Performance Slave definieras som ett eget block och identifieras med ett namn som definieras i attributet `identifier`. Eftersom olika instanser av Pitch Performance Slave kan köras på miljöer med olika förutsättningar så som trådlöst nätverk, operativsystem, kompilator-mjukvara, fysisk plats med mera har det inte varit möjligt att på förhand definiera en specifik uppsättning konfigurationsmöjligheter. Varje instans kan därför innehålla noll (0) eller flera parametrar (`param`) med attributen `name` och `value`. Ett exempel på konfiguration kan ses i Listning 5.2.

[11] Utvecklingsmiljö utvecklad av Oracle (`https://netbeans.org/`)
[12] Utvecklingsmiljö utvecklad av The Eclipse Foundation (`http://www.eclipse.org/`)
[13] Utvecklingsmiljö utvecklad av JetBrains (`http://www.jetbrains.com/idea/`)
[10] Cyklomatisk Komplexitet (Cyclomatic Complexity)
[14] Extensible Markup Language

Listning 5.2: Konfiguration av tillgängliga Pitch Performance Slave

```
<pitch>
  <slave identifier="HOPE">
    <param name="name" value="HOPE" />
    <param name="location" value="linkoping" />
    <param name="os" value="Windows␣Vista" />
    <param name="java" value="true" />
    <param name="prti" value="true" />
  </slave>

  <slave identifier="LIPSUM">
    <param name="name" value="LIPSUM" />
    <param name="location" value="malmo" />
  </slave>
</pitch>
```

5.6.2 Konfiguration av testsvit

En testsvit är en egen fil som beskriver hur ett (1) eller flera `scenario` ska genomföras. Varje scenario definieras som ett eget block och identifieras med ett namn som definieras i attributet `identifier`, utöver ett namn specificeras även hur många kopior av det specifika scenario som ska distribueras till Pitch Performance Slave med attributet `instances` och ifall scenariot kräver en reserverad instans av Pitch Performance Slave eller ifall flera scenario kan exekveras parallellt på samma fysiska eller virtuella maskin med attributet `reserved`.

Varje scenario innehåller ett block `requirements` som specificerar vilka krav som ställs på de instanser av Pitch Performance Slave som ska exekvera det specifika scenariot, detta block med krav specificeras likt konfigurationsmöjligheterna beskrivna i Sektion 5.6.1.

Eftersom samtliga testsviter och därmed scenario exekveras via ett automatiserat system och varje instans av Pitch Performance Slave inte kan förväntas upprätthålla ett bibliotek (repository) av applikationer och komponenter som kan behöva testas finns möjligheten att specificera noll (0) eller flera platser att hämta innehåll för ett specifikt scenario. Innehåll beskrivs av ett block `content` med attributen `protocol` och `path`, vidare kan detta block innehålla parametrar (`param`) som specificerar uppgifter relevanta för det specifika protokollet så som användarnamn och lösenord. Protokoll för subversion och lokala filer (`file://`) implementerades under projektet, åtkomsten till lokala filer gäller även delade nätverksresurser och implementerades framför allt för att underlätta utveckling av testsviter där applikationen eller komponenten under test inte ännu skickats in till något revisionshanteringssystem.

Slutligen innehåller ett scenario ett (1) eller flera test. Varje test definieras som ett block och identifieras med ett namn som definieras i attributet `identifier`, utöver ett namn specificeras även den relativa sökvägen från det innehåll i scenariot till den exekverbara filen som utgör det faktiska testet. Varje test innehåller dessutom en (1) eller flera parametrar, till skillnad från parametrar för tidigare beskrivna konfigurationer kräver ett test-block en parameter med `name` *entrypoint* som specificerar vilken klass som skall exekveras. Vidare krävs det att resterande parametrar endast har attributet `name` och dessa parametrar kommer att skickas vidare till den av *entrypoint* specificerade klassen

som in-parametrar från terminalen.

Listning 5.3: Konfiguration av Testsvit

```
<pitch>
  <scenario identifier="someFederate" instances="3" reserved="false">
    <requirements>
      <param name="location" value="linkoping" />
    </requirements>

    <content protocol="local" path="\\HOPE\someFederate" />

    <test identifier="smallJoin" path="someFederate/myfed.jar">
      <param name="entrypoint" value="se.pitch.somefederate" />
      <param name="192.168.1.30" />
      <param name="20" />
    </test>

    <test identifier="BigJoin" path="someFederate/myfed.jar">
      <param name="entrypoint" value="se.pitch.somefederate" />
      <param name="192.168.1.30" />
      <param name="200" />
    </test>
  </scenario>

  <scenario identifier="pRTI" instances="1" reserved="true">
    <requirements>
      <param name="location" value="linkoping" />
      <param name="prti" value="true" />
    </requirements>

    <content>
      <application filename="prti.exe">
        <param name="Xms" value="512" />
        <param name="Xmx" value="4092" />
      </application>
    </content>
  </scenario>
</pitch>
```

Kapitel 6

Diskussion

Den programsvit som har tagits fram under det här arbetet bör i flera avseenden ses som en första prototyp av ett system för att genomföra testning av distributerade system. Eftersom det initialt inte funnits någon särskilt detaljerad kravspecifikation utan kraven och behoven vuxit fram parallellt med utvecklingen av programsviten så har vissa designval och förenklingar genomförts vilket innebär att det finns utrymme för att förbättra designen såväl som att optimera algoritmer och implementationer.

Allt eftersom programsviten vidareutvecklades och krav på att kunna skriva och genomföra testfall från programvara och bibliotek som fanns lagrade på flera olika ställen och var åtkomliga genom ett flertal olika protokoll växte fram så blev det snabbt tydligt att konfiguration för genomförande av ett specifikt testfall kräver många stödtjänster och verktyg för att inte upplevas som hindrande för utvecklaren. Tittar man på applikationen `ftp` som finns tillgänglig i de flesta UNIX miljöer så finns det 22 olika argument för konfiguration där vissa dessutom tar egna argument för ännu mer detaljerad konfiguration. Tittar man på applikationen `subversion` som finns tillgänglig i de flesta UNIX miljöer och dessutom är en stor del av den infrastruktur som beställaren använder finns där 32 olika kommandon som alla tar egna argument.

Att för en så pass specifik applikation som denna programsvit försöka inkorporera all önskvärd konfiguration för alla applikationer, protokoll och system som kan tänkas användas i ett företag som sysslar med mjukvaruutveckling kan inte anses vara en hållbar lösning med tanke på den massiva expansion av verktyg, programmeringsspråk, tjänster och hårdvara som pågår just nu. Även om den huvudsakliga funktionaliteten mellan en utvecklingsmiljö (Integrated Development Environment, IDE) och den programsvit som utvecklats i detta projekt skiljer sig avsevärt så finns det ett, för diskussionen, viktigt snitt mellan de båda i det att även en utvecklingsmiljö måste hantera problem med att sammanfoga källkod från olika källor och kompilera med de beroenden som introducerats av utvecklaren.

Några av de stora utvecklingsmiljöerna så som NetBeans och Eclipse stödjer flera olika programmeringsspråk och olika modeller inom dessa språk, även utan den extra komplexiteten som stöd för flera programmeringsspråk innebär så finns det väldigt många parametrar som måste tas i beaktning.

Problematiken kring att på ett effektivt sätt sätta samman bibliotek och källkod är på ingalunda sätt nytt utan har existerat minst lika länge som mjuk-

varuutveckling. Blickar man tillbaka till de första försöken till att lösa detta problem så är systemet `Make` ett bra exempel som vid första utgåvan 1977 säkerligen underlättade arbetet avsevärt. Även om `Make` fortfarande används i stor utsträckning så har applikationerna vuxit enormt och beroenden på tredjeparts bibliotek har ökat.

På senare tid har det kommit ytterligare verktyg för att underlätta arbetet med att sammanfoga och bygga mjukvara, bland de många olika system som har föreslagits och introducerats till marknaden finner jag det särskilt intressant att nämna `Ant` som introducerades 2000 [2] och `Maven` som introducerades 2002 [3]. Den stora skillnaden mellan `Make`, `Ant` och `Maven` är att de i olika grad fokuserat på problemet att hantera kod från olika källor, `Make` fokuserar framför allt på att sammanfoga bibliotek och källkod som existerar på den maskin där det exekveras, medan `Ant` och `Maven` tillhandahåller möjligheten att hämta bibliotek, källkod och andra artifakter från externa källor.

Då både `Ant` och `Maven` är system som en utvecklare förutsätts känna igen och vara tillräckligt bekväm med för att kunna konfigurera och det dessutom finns en stor mängd information kring de båda systemen att tillgå på internet är dessa bra alternativ och utgångspunkt för system som dels är starkt beroende av möjligheten att sammanfoga bibliotek och källkod från olika källor och olika protokoll, dels är för små för att på egen hand tillhandahålla den funktionalitet som är nödvändig för att ge utvecklare en bra användarupplevelse.

Resterande del av diskussionen har delats upp i de specifika frågor som beskrivits i Sektion 1.2, där Paragraf I innehåller en diskussion kring hur ett distribuerat testfall skall beskrivas, Paragraf II innehåller en diskussion kring hur en grupp testfall skall distribueras med fokus på hårdvara, Paragraf III innehåller en diskussion kring hur testfall skall distribueras med fokus på jämförbara och tillförlitliga resultat och Paragraf IV innehåller en diskussion kring hur testfall ska beskrivas utan att mängden blir ohanterligt stor.

I ***Hur skall ett testfall som ska distribueras över en eller flera klienter beskrivas?*** För att på ett så enkelt sätt som möjligt kunna beskriva ett test där flera maskiner, eller instanser av Pitch Performance Slave är inblandade används en gruppering i form av ett scenario. Terminologin som valts kan i efterhand förefalla något felaktig vilket visat sig problematiskt när det kommer till att beskriva ett testfall. Formatet (XML) i sig är väldigt vanligt förekommande inom området och utgör i sig inget hinder för en utvecklare, däremot kräver det en väldigt noggrann struktur med många upprepningar för att åstadkomma relativt enkla konfigurationer. Problematiken med att beskriva ett testfall är framför allt kopplat till den mängd olika möjligheter som måste tillhandahållas för att arbetet inte skall vara, eller upplevas, begränsande.

Det bör vara så enkelt som möjligt att beskriva vad som skall genomföras och helst av allt så ska det inte kräva särskild kunskap om systemet eller programmering i allmänhet. Med den implementerade konfigurationen som utgångspunkt kan man tänka sig ett antal förbättringar som skulle underlätta konfigurationen avsevärt. Möjligheten att referera till innehåll så som bibliotek, applikationer och test-filer med namn istället för att explicit specificera protokoll och plats skulle underlätta såväl utveckling av tester som beskrivning av tester avsedda att exekveras via ett automatiserat system. Tittar man till exempel på hur `Maven` konfigureras så specificeras beroenden med en grupp, ett id och versionsnummer

där gruppen specificerar vilket paket komponenten tillhör, id specificerar vilken komponent det gäller och versionsnummer specificerar vilken version av komponenten som är relevant. Det finns med andra ord inget behov för användaren att känna till vem som utvecklat komponenten, var komponenten finns eller hur den är åtkomlig.

Med stöd för att referera till innehåll med namn och dessutom kunna referera till delar av testet med namn skulle samma testfall som beskrivs i Listning 5.3 istället kunna beskrivas som illustrerat i Listning 6.1. Bortser man från den struktur som krävs när testfallen beskrivs med hjälp av XML skulle samma testfall kunna beskrivas enligt nedan.

"`START pRTI AS prti ON RESERVED`"
"`START 3 someFederate AS smallJoin PARAMETERS prti.ip 20`"
"`START 3 someFederate AS bigJoin PARAMETERS prti.ip 200`"

Listning 6.1: Konfiguration av Testsvit med referenser

```
<pitch>
  <scenario identifier="testSomeFederate" instances="3" reserved="false">
    <test identifier="smallJoin" test="someFederate">
      <param name="pRTI.ip" />
      <param name="20" />
    </test>

    <test identifier="BigJoin" test="someFederate">
      <param name="pRTI.ip" />
      <param name="200" />
    </test>
  </scenario>

  <scenario identifier="pRTI" instances="1" reserved="true">
      <application name="prti">
        <param name="Xms" value="512" />
        <param name="Xmx" value="4092" />
      </application>
  </scenario>
</pitch>
```

II ***Hur skall en grupp av testfall distribueras över ett nätverk för att maximera användningen av hårdvara?*** Även om tanken på att kunna använda existerande hårdvara så som skrivbordsdatorer, laptops och servrar för att köra instanser av Pitch Performance Slave och utnyttja dessa då de inte används till annat är mycket intressant i sig så kommer en sådan implementation med en hel uppsättning egna problem av såväl teknisk och etisk natur, i detta fall visade det sig inte heller vara relevant då utgifter för hårdvara dedicerad för uppgiften att exekvera distribuerade testfall är förhållandevis liten för ett företag.

För diskussionens skull kan det däremot ändå vara intressant att vidröra något av denna problematik. Vid första anblick kan det förefalla trivialt att avgöra ifall en dator som i vanliga fall används till något annat är passiv eller används under ett givet tidsintervall, det skulle exempelvis vara enkelt att titta på den tid som passerat sedan mus och tangentbord använts samt titta på vilka processer som för närvarande är aktiva och hur mycket processorkraft

och RAM dessa upptar. Problemet blir däremot avsevärt mycket svårare i den händelse då en maskin bedöms vara passiv vid testets början men bedöms vara aktiv vid testets slut. Ifall användaren kommer tillbaka till maskinen efter att ett test påbörjats så kommer användarens användning av maskinen tveklöst att påverka till tillgängliga processorkraften och i värsta fall, medvetet eller omedvetet, genomföra en eller flera handlingar som påverkar resultatet av testet. Att undvika detta problem kräver dels tekniskt avancerade lösningar för att kunna låsa ute användaren från maskinen och dessutom blockera indata från enheter så som mus och tangentbord, dessa förändringar i sig skulle kunna påverka utfallet av ett test och dessutom påverka effektiviteten hos användaren samt den upplevda kontrollen av den egna arbetsmiljön.

En ytterligare intressant fråga som snarare är av etisk natur är huruvida det är etiskt försvarbart för ett företag eller en organisation att medvetet försöka maximera användning av datorer som ur ett energiperspektiv inte är optimerade till att alltid vara igång. I detta fall blir problematiken mycket mer komplex då det kommer ner till individuella maskiner och dess energiförbrukning. Vad som däremot kan sägas att det finns maskiner speciellt avsedda för att köras under längre perioder utan en anmärkningsvärt hög energiförbrukning.

III ***Hur skall en grupp av testfall distribueras över ett nätverk med variabelt antal klienter så att resultatet är tillförlitligt och jämförbart med tidigare resultat?*** På denna punkt har projektet lett till en helt acceptabel lösning där varje testfall kan specificera de krav som finns och varje instans av Pitch Performance Slave konfigureras för att meddela Pitch Performance Master vilka krav den specifika instansen uppfyller. Beroende på hur tillförlitligt och jämförbart ett visst test måste vara kan kraven specificeras mer eller mindre strikt. Ett test (1) där nätverket är centralt kan enkelt specificera vilken plats instanserna måste befinna sig på och vilken typ av nätverksanslutning som måste vara tillgänglig medan ett test (2) där beräkningskapacitet är centralt enkelt kan specificera att en viss typ av processor måste användas. Samtidigt som detta kan specificeras på detaljnivå behöver användaren inte specificera information som inte är relevant för testet.

När det kommer till den faktiska distributionen av testfall så fördelas antalet instanser så jämnt som möjligt inom den delmängd av Pitch Performance Slave som uppfyller kraven för det specifika testet. Instanser av Pitch Performance Slave med minst antal registrerade test prioriteras förutsatt att de uppfyller kraven vilket ger en så jämn distribution som möjligt. Systemet tar däremot ingen hänsyn till den globalt bästa lösningen, då det är en girig algoritm som används skulle det vara möjligt att förbättra resultaten genom att göra en omfördelning där antalet tilldelade instanser från den första fördelning tas i beaktan.

IV ***Hur skall testfall formuleras för att representera verkligheten utan att antalet testfall blir ohanterligt stort?*** Denna fråga har en stark anknytning till Paragraf I då problematiken med en stor mängd testfall endast existerar ifall konfigurationen av testfall är komplex eller tidskrävande. Den implementerade möjligheten att specificera antalet instanser av ett scenario underlättar förvisso hanteringen och minskar storleken på varje enskilt scenario, även om ytterligare förenklingar kan implementeras.

Behovet av att duplicera en redan existerande konfiguration för att till ex-

empel exekvera pRTI i samband med ett nytt test skulle kunna undvikas genom att introducera möjlighet för att referera till, eller inkludera mindre konfigurationer. Detta skulle ytterligare minska storleken och komplexiteten på varje enskild konfiguration samtidigt som det skulle underlätta underhåll av testen då en uppdatering av en komponent skulle innebära ändringar i en enda konfiguration istället för ändringar i samtliga konfiguration som nyttjar den specifika komponenten.

Sammanfattningsvis så skulle jag vilja påstå att det finns goda möjligheter att utveckla ett system för att exekvera distribuerade testfall såväl som andra distribuerade applikationer över ett nätverk likt det som utvecklats under detta arbete. Ett sådant system skulle ha mycket att vinna på en stark koppling till något av de större byggsystem som redan finns då mycket av den funktionalitet som krävs även finns i dessa byggsystem. Maven är en särskilt stark kandidat då det finns möjlighet för såväl publika som privata bibliotek (repositories) vilket skulle underlätta distribution såväl som uppdateringar av programvara avsevärt.

Litteraturförteckning

[1] http://jenkins-ci.org/. [online; accessed 2013-05-09].

[2] http://ant.apache.org/faq.html#history. [online; accessed 2013-05-09].

[3] http://maven.apache.org/background/history-of-maven.html. [online; accessed 2013-05-09].

[4] Boris Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, 1984.

[5] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House Publishers, 2004.

[6] Eric Freeman and Elisabeth Freeman. *Head First Design Patterns*. O'REILLY.

[7] Maurizio Morisio Hakan Erdogmus and Marco Torchiano. On the Effectiveness of the Test-First Approach to Programming. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, January 2005.

[8] IEEE Computer Society. IEEE Standard Glossary of Software Engineering Terminology, 1990.

[9] IEEE Computer Society. IEEE Standard for Modelling and Simulation (M&S) High Level Architecture. Technical report, Institute of Electrical and Electronics Engineers, 2010.

[10] Thomas J. McCabe. A Complexity Measure. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, SE-2, April 1976.

[11] Pitch Technologies. The HLA Tutorial. http://www.pitch.se/images/files/tutorial/TheHLAtutorial.pdf.

[12] Pitch Technologies. Viking 08. http://www.pitch.se/services-and-solutions/viking-08, 2008. [online; accessed 2012-12-25].

[13] Christoph Steindl. Test Driven Development. http://cf.agilealliance.org/articles/system/article/file/1423/file.pdf. [online; accessed 2013-05-06].

[14] Hirotaka Takeuchi and Ikujiro Nonaka. The new new product development game. *Hardvard Business Review*, 1986.