Institutionen för datavetenskap
Department of Computer and Information Science

Examensarbete

Evaluation of Plugin Frameworks for the Jenkins Continuous Integration Build Server

av

Jens Christensen
Jonatan Ekstedt

LIU-IDA/LITH-EX-G--12/006--SE

2012.03.28

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Jens Christensen & Jonatan Ekstedt

Linköpings universitet
Institutionen för datavetenskap

Examensarbete

Evaluation of Plugin Frameworks for the Jenkins Continuous Integration Build Server

av

Jens Christensen
Jonatan Ekstedt

LIU-IDA/LITH-EX-G--12/006--SE

2012-03-28

Handledare: Tomas Hellberg, Autoliv

Examinator: Peter Dalenius, Institutionen för datavetenskap, Linköpings universitet

Sammanfattning

Begreppet *Continuous Integration* (CI) är idag centralt för många företag i deras produktutveckling. Kraven dessa företag ställer på denna programvara skiljer sig, beroende på vad de använder den till och hur deras miljö ser ut. Jenkins är en programvara som används för CI, det är öppen källkod och har ett brett stöd för pluginer. Det finns ett stort urval av pluginer redan idag, men det är inte säkert att specifika önskemål från företag uppfylls av dessa. Därför är det intressant att på ett snabbt sätt ta fram specifika pluginer för dessa företag.

Vi har i denna rapport utrett möjligheterna att utveckla pluginer till Jenkins i Ruby. Det senaste året har utveckling av pluginer i Ruby vuxit fram för Jenkins. Ramverket är fortfarande i ett tidigt stadium, men är utformat för att kunna falla tillbaka på det programmeringsspråk som Jenkins är skrivet i; Java. Det är på så sätt fullt möjligt att nu skriva pluginer i Ruby. Ruby är ett expressivt språk som är lätt att ta till sig, och den komplexitet som följer pluginutveckling i Java är till stor del gömd i Rubys ramverk.

Vår slutsats är att Ruby är tillräckligt moget för att användas till pluginutveckling för Jenkins.

Examensarbetet är uppdelat i två delar: en utvärdering av ramverken och deras verktyg för Ruby och Java, och en utvecklingsfas där vi fastställer vår analys. Den Rubyplugin som utvecklats kan ses som ett *'proof-of-concept'*, denna kan även användas som en slags mall vid framtida pluginutveckling vid Autoliv.

Abstract

The concept *Continuous Integration* (CI) is vital to many companies today in their product development. These companies may have specific demands on their CI-software, depending on how they are using it and what their development environment looks like. Jenkins is a software that is used for CI, it is open source and it has a wide support for plugins. There is a great selection of plugins available today, but it is not certain these plugins satisfy the specific requirements. It is therefore interesting to, in a quick way, develop plugins that meet these conditions.

In this report, we have evaluated the possibility to develop plugins for Jenkins in Ruby. In the last year or so, plugin development in Ruby has been growing to become a viable option. The framework is still at a very early stage, but it is constructed in such a way so that one can always fall back on the language Jenkins was made in; Java. Because of this it's definitely possible to write plugins in Ruby. Ruby is an expressive language and it is easy to learn, the complexity that comes with writing plugins in Java for Jenkins is largely hidden in the Ruby framework.

Our conclusion is that Ruby is ready to be used for plugin development for Jenkins.

This thesis is divided into two parts, an evaluation of the frameworks and the tools for Java and Ruby, and a development phase where we concrete our analysis. The Ruby plugin that is developed in this thesis can be seen as proof-of-concept, it can also be used as a kind of template for future plugin development at Autoliv.

Förord

Vi vill tacka Peter Dalenius som, förutom sin roll som examinator, fungerat som vår handledare på skolan och alltid haft en positiv attityd.

Vi vill också rikta ett tack till Tomas Hellberg som varit vår handledare på Autoliv och som hjälpt oss med Ruby och att förstå Jenkins.

We want to specially thank Charles Lowell who has answered our questions about Ruby and Jenkins.

Innehåll

1 Inledning 1
1.1 Syfte 1
1.2 Kravspecifikation 2
1.3 Målgrupp 2
1.4 Metod och källor 2
1.5 Struktur 3
1.6 Typografiska konventioner och språkkonventioner 3
1.7 Avgränsningar 3
2 Introduktion: Jenkins och dess ramverk 4
2.1 Continuous Integration och Jenkins 4
2.2 Ruby 5
2.3 Java 6
2.4 JRuby 6
3 Analys 7
3.1 Teoretisk analysdel 7
3.1.1 Skillnader 7
3.1.2 Ruby eller Java 9
3.1.3 Teoretiskt resultat 11
3.2 Praktisk analysdel 11
3.2.1 Tester av verktyg för Ruby 13
3.2.2 Tester av verktyg för Java 14
3.2.3 Java vs Ruby i Jenkins 14
3.2.4 Praktiskt resultat 15
3.3 Resultat och beslut 15
3.3.1 Beslutsgrund för Ruby 15
3.3.2 Anledning till att Java väljs bort 16
4 Utvecklingsprocess 17
4.1 Pluginutveckling i Ruby 17
4.2 Testning 21
4.3 Integration med Jenkins 22
4.4 Problemkällor under utvecklingen 24
5 Slutsats 25
Referensförteckning 26
Bilaga A 28
Bilaga B 32
Bilaga C 33

1 Inledning

Continuous Integration, CI, är ett koncept inom mjukvaruutveckling där medlemmar i ett team integrerar arbetet ofta. Vanligtvis gör varje person i utvecklarlaget det dagligen, vilket resulterar i flera integrationer per dag. Varje integration verifieras och projekt byggs och genomgår tester. På så sätt upptäcker man problem väldigt snabbt. (Fowler, 2006)

Jenkins är en programvara som bygger på detta koncept och det används av många företag (Jenkins, 2011c). Det är öppen källkod och har ett välutvecklat stöd för pluginer. I dagsläget finns drygt 400 pluginer tillgängliga (Jenkins, 2011a) för nedladdning för att utöka Jenkins funktionalitet eller underlätta användandet. Hur företag använder Jenkins kan skilja sig, och det finns därför en hel del specifika behov som inte täcks upp av de pluginer som redan existerar.

Jenkins är utvecklat i Java och majoriteten av de pluginer som finns tillgängliga är också skrivna i det språket. I dagsläget finns det endast ett fåtal registrerade pluginer skrivna i s k *pure Ruby*, ett annat språk. (Github, 2012)

Pluginutveckling i Ruby befinner sig i ett väldigt tidigt stadium, uppdateringar av verktyg kommer flera gånger i månaden och vissa delar av rapporten kan därför bli inaktuella väldigt fort. Vi har haft kontakt med Charles Lowell som utvecklar ramverket för JRuby, och Kohsuke Kawaguchi, skaparen bakom Jenkins, genom Internet Relay Chat (IRC) och Meeting Center (Cisco WebEx Meeting Center) där vi diskuterat dels vilka klasser som kan vara lämpliga att utöka samt rapporterat in buggar i ramverket och Jenkins.

1.1 Syfte

Syftet med detta examensarbete är att undersöka om ramverket för Ruby är moget att användas för utveckling av pluginer till Jenkins. Autoliv är intresserade av att utveckla pluginer till Jenkins för eget bruk och vår handledare och tillika Jenkins-expert är specifikt intresserad av Ruby. Eftersom ramverket befinner sig i en väldigt tidig fas, är det inte fastställt om det är lämpligt att använda sig av till något annat än experimentell utveckling. Den här rapporten ska ge insikt i huruvida ramverket är tillräckligt redo för att utveckla pluginer på ett enkelt och smidigt sätt. Rapporten ska också ligga till grund för framtida utveckling av pluginer och därför ska en eller flera pluginer utvecklas som *proof-of-concept*.

1.2 Kravspecifikation

De krav som ställts på examensarbetet är en utredning av Ruby som utvecklingsspråk för pluginer och framställning av en lämplig plugin som Autoliv behöver. Vidare så vill Autoliv att det ska finnas färdiga tester att köra på den plugin som vi skrivit samt att byggprocess och tester ska ske automatiskt i Jenkins när källkod uppdaterats. Dessutom vill Autoliv ha en snabbguide över hur pluginutveckling går till i det språk som vi valt, och väldokumenterad kod som de kan använda som mall för hur utvecklingen ska gå till. Guiden för pluginutveckling återfinns i Bilaga A sist i rapporten.

1.3 Målgrupp

Målgruppen är främst Autoliv som vill ha ett bra underlag för beslut angående fortsatt pluginutveckling, men också andra företag och personer som är intresserade av att utveckla egna pluginer för Jenkins.

1.4 Metod och källor

Arbetet delades upp i två delar: analys och implementation. En planeringsrapport skrevs första veckan i vilken vi satte en tydlig deadline för när utredningen skulle vara klar.

De första tre veckorna, själva analysdelen, bestod av en utredning av ramverken. Målet med utredningen var att få en djupare insikt i hur väl verktygen för dem olika ramverken fungerar. Vad händer när vi stöter på problem med Ruby? Går de att lösa eller är framsteg omöjligt fram till dess att ramverket byggts ut mer? Samtidigt bekantade vi oss med Jenkins, en programvara som har väldigt stor funktionalitet, och drygt ett hundratal s k *extension points*, där man kan bygga ut Jenkins funktionalitet genom pluginer.

Det finns inga böcker skrivna om pluginutveckling för Jenkins i Ruby, så vår informationskälla bestod främst av Internet där vi använde oss av dels Jenkins wikisidor, där det finns en speciell sektion för utveckling av pluginer i Ruby(Jenkins, 2011e), dels diverse forum och bloggar för att reda ut problem. Med anledning av detta kan vissa källor kännas mindre trovärdiga, tyvärr fann vi inte några andra alternativ.

För att få så uppdaterad och korrekt information som möjligt inledde vi konversationer med Jenkins huvudsaklige grundare Kohsuke Kawaguchi och Rubyutvecklaren Charles Lowell genom *IRC* och *Meeting Center*. Vi diskuterade vilka klasser i Jenkins kärna som är lämpliga att utöka och rapporterade in buggar i befintliga Ruby-klasser och Java-klasser i Jenkins.

Bristen på litteratur på området ledde till att en praktisk analys genomfördes som del av beslutsgrunden. Vi skapade ett fåtal pluginer, i både Java och Ruby, prototyper av de pluginer som vi i den senare delen av examensarbetet skulle implementera, för att se vad som var praktiskt genomförbart och vilket stöd som faktiskt finns idag.

När analysdelen var klar diskuterade vi resultatet tillsammans med vår handledare då vi vägde för- och nackdelar och tog ett beslut. Därefter startade vi utvecklingen av pluginer för att fastställa vår analys och utredning.

Den metod vi valde för utveckling av pluginer var *Test driven development* (TDD), den har likheter med den agila metoden extreme programming men har på senare tid blivit ett eget koncept. (Wells, 2000) En närmare beskrivning om TDD kommer i kapitlet utvecklingsprocess.

1.5 Struktur

Den här rapporten har utformats med anvisningarna i ”Lathund för Rapportskrivning” i åtanke. Alla referenser är skrivna enligt Harvardsystemet vilket innebär att en parentes med författare och årtal skrivs ut efter en mening eller ett stycke som refererar till något. Referenserna återfinns i slutet av rapporten.

Detta kapitel uppföljs av en beskrivning av vad Continuous Integration och Jenkins är för något, varför företag använder det samt varför Ruby är intressant i sammanhanget. Därefter beskrivs vad som krävs för pluginutveckling, vilka verktyg som används och vad som skiljer Ruby och Java åt. I slutet av rapporten återfinns en slutsats som sammanfattar undersökningen och som återkopplar till syftet med examensarbetet.

1.6 Typografiska konventioner och språkkonventioner

Vi har valt att inte översätta de engelska ord som är vedertagna inom datatekniken. De engelska tekniska termer som kan vara nya för läsaren skrivs kursivt vid första förekomsten.

Vi har valt att använda “plugin” som benämning på programmen, snarare än “insticksmodul”, eftersom plugin är mer vanligt förekommande och vi anser att det är lättare att förstå. En plugin böjer vi i singular bestämd form till “pluginen”, och i plural “pluginer” samt “pluginerna”.

Kodexempel kommer att skrivas i `Courier New` och, för ytterligare förtydligande, ha en grå bakgrund för att man tydligt ska kunna urskilja källkod från övrig text.

1.7 Avgränsningar

Jenkins är väldigt stort, och har stor funktionalitet. Vi försöker förklara de delar av Jenkins som är relevanta ur ett pluginutvecklingsperspektiv, men vi går inte in mer på djupet än så.

2 Introduktion: Jenkins och dess ramverk

I detta avsnitt följer en djupare förklaring till Continuous Integration, Jenkins och ramverken som är inblandade.

2.1 Continuous Integration och Jenkins

När ett mjukvaruinriktat företag växer och antalet utvecklare blir större är integration av kod mellan olika utvecklingsteam ett stort problem som kan uppstå. När man jobbar med olika delar av programvaran på olika håll kan det skapa problem när de olika delarna ska slås ihop. Skillnaderna i hur man löst olika problem och gått till väga på olika sätt måste slipas ner så att delarna kan fungera tillsammans. Denna process kallas för integrering och kan vara riktigt smärtsam i stora projekt. I vissa fall tar integreringen mellan delarna mer tid än själva utvecklingen av de olika delarna. (Fowler, 2006)

Continuous Integration, CI, är ett koncept som ofta används inom mjukvaruutveckling i större projekt för att undvika detta integreringsträsk. Alla utvecklare ska enligt denna princip integrera sin kod i projektet varje dag, eller flera gånger om dagen, genom att lägga den i ett *repository* där alla andra utvecklares kod finns. Förutsättningen är att varje utvecklare ser till att koden är i ett sådant tillstånd att den kompilerar och att alla tester lyckas. Det gäller alltså för utvecklarna att se till att deras kod aldrig befinna sig mer än några minuter från detta tillstånd. (Roberts, 2009)

Man utgår från systemets aktuella tillstånd, *mainline*, genom att ta en kopia på projektet från repositoryt och jobba lokalt med denna. När man är klar med sina förändringar och koden byggts och testats uppdaterar man sin kod mot mainline, i.e. tar en ny kopia, eftersom denna kan ha förändrats medan man jobbat, och bygger och testar lokalt. Om förändringarna inte medför problem ska man genomföra en incheckning mot repositoryt så att ens förändringar blir en del av mainline. (Fowler, 2006)

Här kan olika företags CI-vägar skiljas. Antingen bygger och testar man sin kod med mainline manuellt, eller så tar en server hand om bygget och testningen automatiskt. (Fowler, 2006)

Det finns fördelar med att bygga manuellt, t ex att man då är närvarande under bygget och testningen och kommer förmodligen därmed att sträva efter snabba byggen i större grad än man skulle göra vid automatiska byggen. Man tenderar då att skriva effektivare och snabbare tester. En annan fördel är att man inte riskerar att få tag på kod som inte fungerar när man bygger. Vid automatiska byggen kan detta hända eftersom repositoryt inte blir låst när man integrerar sin kod. Då kan någon annan av misstag använda en version av ens kod som visar sig vara korrupt. (Shore, 2005)

Alternativet till manuell integrering är att använda en byggserver. Projektet byggs då automatiskt och testas direkt när någon checkar in sin kod mot repositoryt. Om personens ändringar har medfört nya problem i projektet får personen reda på det direkt och kan korrigera problemen. Många byggservrar har ett webbinterface som gör det enkelt att hantera servern. Detta interface har ofta någon form av visuell respons på hur bygget gick. En grön symbol motsvarar ett lyckat bygge, medan en röd symbol motsvarar ett misslyckat

(det finns populära alternativ, t ex fysiska lavalampor som lyser rött eller grönt, och där antalet bubblor motsvarar hur länge tillståndet varit så [Pragmatic Automation, 2005]).

CI är alltså en viktig beståndsdel i mjukvaruutveckling. De olika utvecklarnas kod kommer att vara relativt kompatibla under hela utvecklingsprocessen eftersom integrering sker så ofta. Därför kommer inga stora problem att uppstå vid integrering, utan endast mindre problem som går snabbt att lösa. På så vis finns alltid ett bygge av projektet och alla har en översikt över projektets hälsa.

Förutom att undvika långa manuella integreringsprocesser kommer CI att resultera i en mer smärtfri debuggprocess. Om enstaka buggar dyker upp emellanåt är dem betydligt lättare att fixa än om alla buggar visar sig på en gång, eftersom buggar ofta genererar fler buggar. I det fallet blir det svårt att hitta alla buggar i koden. Det är också ofta psykologiskt tyngre att debugga ett system med många buggar. (Fowler, 2006)

För att använda sig av CI behöver man ett versionshanteringssystem där man har sitt repository. I detta har alla sin senaste kompilerande kodversion. För att bygga och testa automatiskt behöver man även ett serverprogram som tar hand om byggena och testningen. Serverprogrammet kommer alltså att kommunicera med repositoryt i versionshanteringssystemet och, vid en ny incheckning, bygga projektet och meddela om resultatet. Jenkins är ett sådant serverprogram som bygger projektet automatiskt, skapat av Kohsuke Kawaguchi, som kan användas för att praktisera detta koncept och det används av väldigt många företag (Jenkins, 2011c). Jenkins ursprung ligger i Hudson, som var ett samarbete med Oracle. Hudson splittrades 2011 pga en dispyt mellan Oracle och den stora utvecklarbas som bidrog till Hudson. Eftersom Oracle ägde varumärket delades Hudson upp i två oberoende projekt. Båda anser det andra vara en fork (Robertsson, 2011) (Hudson, 2011). (Jenkins, 2011b)

Själva byggservern som kör Jenkins kallas *master*, och de datorer som kopplar upp sig mot mastern är *slavar*. Det är på slavarna som jobben körs.

2.2 Ruby

Ruby är ett helt objektorienterat språk som kom 1995. Det utvecklades av Yukihiro Matsumoto och interpretatorn är skriven i språket C. Ruby fick en stor användarbas i mitten av 2000-talet och är nu ett av de största programmeringsspråken i världen. (ruby-lang.org, 2012) Något som är intressant med Ruby är att allting är ett objekt, även siffror. Följande programkod är giltig i Ruby:

```
3.times { puts “Hello World!” }

Konsol: Hello World! Hello World! Hello World!
```

Matsumoto blandade delar av hans favoritspråk (Perl, Smalltalk, Eiffel, Ada och Lisp) för att forma ett nytt språk som balanserar funktionell och imperativ programmering och han har flera gånger sagt att han försöker göra Ruby naturligt, inte simpelt. Han ville också att Ruby skulle vara flexibelt, man kan lägga till metoder i t ex Numeric, som beskriver ett tal i Ruby, som man själv finner lämpliga.

```
class Numeric
  def plus(x)
    self.+(x)
  end
end

y = 5.plus 6
# y is now equal to 11
```

(ruby-lang.org, 2012)

2.3 Java

Programmeringsspråket Java behöver knappast någon närmare presentation. Det kom 1995, samtidigt som Ruby, och Java är idag ett väldigt stort och populärt språk. Javakompilerare producerar ingen plattformsspecifik kod utan snarare '*byte code*' för en Java Virtual Machine (JVM), vilket innebär att koden kan köras på i princip vilken plattform som helst som har JVM installerat. (Karandikar, årtal saknas)

2.4 JRuby

JRuby såg dagens ljus 2001 och är det som används för pluginutveckling till Jenkins som alternativ till Java. JRuby är en implementering av Ruby i Java (Campos, 2011). Det innebär att man har tillgång till Javas bibliotek och därmed möjlighet att göra fullgoda grafiska gränssnitt med JRuby. JRuby släpar alltid efter Ruby i stöd för senaste versionerna av Ruby, och i januari 2012 deklarerades det att stödet för 1.9.2 är fullgott (jruby.org, 2012), medan Ruby 1.9.3 kom redan oktober 2011 (ruby-lang.org, 2011).

3 Analys

Analysdelen av arbetet bestod i att jämföra de två språk som var aktuella för pluginutveckling för Jenkins: Java och Ruby. Vår uppgift var att utvärdera de två ramverken för att se vilket av dem som är mest adekvat för Autoliv. Detta var tänkt att genomföras i en litteraturstudie av pluginutveckling för Jenkins och av språken för att få en uppfattning om hur de olika ramverken passar för Autoliv.

Vår handledare, Autolivs Jenkins-expert, har en uttalad preferens för Ruby, mycket på grund av dess expressivitet. Syftet med vårt examensarbete var därför att undersöka dem båda ramverken och möjligtvis verifiera att Ruby fungerar åtminstone tillräckligt bra för Autoliv att använda i fortsatt pluginutveckling, även om det skulle visa sig att Java har tydliga fördelar framför Ruby. Om vi eventuellt skulle komma fram till att Java ändå är överlägset Ruby, så skulle det vara ett helt acceptabelt resultat.

Första delen av analysen bestod av en teoretisk jämförelse mellan språken där de olika språkens generella för- och nackdelar, men även dess styrkor och svagheter gentemot pluginutveckling för Ruby, vägdes mot varandra.

Som nämnts ovan, under Metod och källor (avsnitt 1.4), finns väldigt lite dokumentation över pluginutveckling i Ruby. Därför fick även en praktisk del med implementation av en prototyp av pluginen vara en del av vår analys och därmed också påverka vår slutsats.

3.1 Teoretisk analysdel

Här kommer vi först ta upp lite olika områden som skiljer språken åt, sedan jämföra de mot varandra i en form av duell, och till sist komma fram till ett teoretiskt resultat.

3.1.1 Skillnader

Ruby är, till skillnad från Java, ett dynamiskt språk med dynamiskt typsystem, vilket innebär att värdenas typer kontrolleras vid körning istället för vid kompilering. Språket är också starkt typat precis som Java. Detta innebär att ett värde är av en specifik typ under hela sin existens: t ex, om 3 är en fixnum (heltal) kommer det alltid att vara det och kan inte hanteras som en textsträng. I ett dynamiskt typsystem har variablerna inte typer, de är endast namntaggar. Ett värde kan tillhöra vilken variabel som helst, oberoende av vilka värdetyper som tillhört variabeln tidigare. Variablerna är således inte låsta till att hålla en speciell typ av värden.

Expressivitet

Det dynamiska typsystemet gör Ruby mer expressivt än Java. Ett språks expressivitet mäts, enkelt uttryckt, i språkets förmåga att uttrycka saker kortfattat. T ex, av två likadana program, med samma funktionalitet, implementerade i Java respektive Ruby, är Rubyprogrammets filer avsevärt kortare, i avseende antalet kodrader och antalet tecken per kodrad.

Exempel 3.1 visar två exempel i dem olika språken, ett “Hello World”-exempel och ett exempel som skriver ut talen 1 till 3 i konsolen:

<table>
<tr><th>Java</th><th>Ruby</th></tr>
<tr><td><pre>public class HelloWorld {
  public static void main(String[]
args)
  {
    System.out.println("Hello
world!");
  }
}</pre></td><td><pre>puts "Hello world!"</pre></td></tr>
<tr><td><pre>for(int i = 1; i < 4; i++) {
  System.out.println(i);
}</pre></td><td><pre>for i in 1..3
  puts i
end</pre></td></tr>
<tr><td><pre>public class Test
{
  public static void main(String[] args)
  {
    int mInt = 5;
    printI(mInt);
    String mString = “hej”;
    printS(mString);
  }

  public static void printI(int arg)
  {
    System.out.println(arg);
  }

  public static void printS(String
arg)
  {
    System.out.println(arg);
  }
}</pre></td><td><pre>def putsVar var_
  puts var_
end

variable = “hej”
putsVar variable
variable = 5
putsVar variable
variable = “5”
# genererar exception
putsVar variable + 1

Konsol:
>hej
>5
>TypeError: String can't be coerced
into Fixnum
       + at
org/jruby/RubyFixnum.java:333
   (root) at test.rb:11
></pre></td></tr>
</table>

Exempel 3.1 Tre kodexempel i de respektive språken.

Som exemplet visar så behövs det ingen specifik main-metod i Ruby, allt man skriver utanför en klass eller modul sker i samma kontext som ett main-objekt, som kan nås genom `self` (`self => main`). Metoder som deklareras blir privata metoder i detta main-objekt. I Java måste man deklarera en klass, och denna måste också innehålla en main-funktion, annars vet inte Java vart den ska börja exekvera. Man måste ange att main är en offentlig(public) metod som man kan komma åt utanför klassen. Man måste också ange att main är statisk, vilket innebär att man inte måste instansiera en klass för att kunna komma åt main. Dessutom måste vi ange att main inte returnerar ett värde. Allt detta slipper man i Ruby.

I det tredje exemplet ser vi styrkan med det dynamiska typsystemet. Variabeln i Ruby kan innehålla både en textsträng och ett tal, men man kan inte addera till en variabel som innehåller ett tal som en textsträng eftersom språket inte är löst, utan starkt typat. Java-motsvarigheten behöver både olika variabler och metoder för de olika värdena, och metoderna måste anges som statiska om man inte ska behöva instansiera klassen Test.

Skriptsspråk

Ruby är ett tolkat skriptspråk, som tolkas direkt vid exekveringen. Det behöver alltså inte kompileras. Angående detta skriver Dave Thomas (2010, s xv) i inledningen till boken Programming Ruby:

> You can concentrate on solving the problem at hand, instead of struggling with compiler and language issues. That’s how it can help you become a better programmer: by giving you the chance to spend your time creating solutions for your users, not for the compiler.

Utvecklingsverktyg

En tydlig nackdel för Rubys del är utvecklingsverktygens oförmåga att förstå koden medan man programmerar (Fox, 2006):

> Despite Ruby's advantages, Java's static typing does give it one ability that leaves it as the preferred choice for large-scale projects: Java tools understand code at development-time. IDEs can trace dependencies between classes, find usages of methods and classes, auto-complete identifiers, and help you refactor code. Though parallel Ruby tools exist with limited functionality, they lack type information and so cannot perform all these tasks.

Denna nackdel märker man tidigt när man programmerar i Ruby, speciellt om man är van vid Java, och det beror på det dynamiska typsystemet.

Skalbarhet

James Gosling (2008), mest känd för att vara en av Javas skapare, säger angående Java: ”Doing the hard stuff is possible, doing the easy stuff is hard”. Han fortsätter med att kommentera Rubys “enkelhet” såhär: “It doesn’t scale very well […]. [It is] all about making the easy stuff easy but the hard stuff [is] impossible”. Detta är ett vanligt förekommande argument mot Ruby–att det inte skalar särskilt bra.

Många delar inte den uppfattningen, t.ex. Ezra Zygmuntowicz (2008), medgrundare på Engine Yard: “Languages don’t scale. Architectures scale […]. Ruby on Rails is just as scalable as any other platform, but there’s no free lunch.”

3.1.2 Ruby eller Java

Följande avsnitt är tänkt att förtydliga hur vi har tänkt när vi jämfört Ruby och Java, vi nämner en egenskap följt av det språk vi anser är bäst på den punkten.

Expressivitet och läsbarhet: Ruby

Ruby är ett mer expressivt språk jämfört med Java. Kodraderna blir färre och kortare. Det är dessutom lättare att snabbt sätta sig in i programkod och förstå

sambandet mellan olika klasser. Anledning är att den komplexitet som finns i Java göms med hjälp av s k *wrappers* som vi återkommer till (avsnitt 3.2). Detta innebär att man slipper undra vad som returneras från en metod–så länge man vet att det t ex är en lista så kan man anropa Rubyspecifika metoder på den listan.

Testning: Ruby

För att koden ska bli lätt att underhålla och för att någon ska våga redigera kodblock så behövs tester som går igenom hela pluginens funktionalitet. Med *RSpec* och *ci_reporter*, Rubybibliotek, blir testfallen enkla att skriva och de är kompatibla med *JUnit* (ramverk för unit testing i Java). Dessutom kan Jenkins med hjälp av *ci_reporter* använda sig av rapporten som skapas för att visa om testerna gick bra eller om de misslyckades.

Verktyg för pluginutveckling: Java

Även om verktygen för Rubyramverket fungerar någorlunda bra, går det inte att förneka att motsvarande verktyg för Java är bättre. Verktyget för Ruby, *jpi*, är fortfarande i *pre-alpha* och integrering av Ruby-pluginer i Jenkins var först möjlig i september 2011 (Lowell, 2011). Ytterligare en fördel med Java-verktyget, *Maven*, är att det finns mer dokumentation och hjälp att få om man skulle ha problem med det.

Utvecklarbas: Java

Eftersom pluginutveckling i Ruby är nytt, finns det ganska få utvecklare som gör pluginer i Ruby. I listan över registrerade pluginer på Jenkins hemsida så ser man tydligt hur Java dominerar. Det innebär att det finns färre pluginer att ta hjälp av när man försöker bygga sin egen Ruby-plugin. En Internetsökning resulterar möjligtvis i en relevant fråga, men oftast utan svar.

Dokumentation: Java

Java har till sin fördel Jenkins Java-doc som man använder sig av för att hitta lämpliga extension points (avsnitt 3.2) samt utforska de klasser man använder sig av vid pluginutveckling. Ruby har ännu ingen motsvarighet. Man använder fortfarande Jenkins Java-doc som referens men måste samtidigt ha källkoden för *jenkins-plugin-runtime*(Github, 2012) för att se vilka wrappers som finns för Ruby.

Plattform: Java

Java-utveckling fungerar i alla operativsystem som har Java Virtual Machine installerat, vilket är en klar fördel. Verktyget *jpi* fungerar för tillfället bara i Unix-baserade operativsystem.

Utvecklingstid: Ruby

Det går fort att skarpt testa pluginer skrivna i Ruby, man startar en Jenkins server med kommandot `jpi server`, och får då en instans av Jenkins med Ruby Runtime samt den plugin man utvecklar. Motsvarigheten till Java är en ganska lång kompileringsprocess, och vad vi har erfarit så måste man starta Jenkins och ladda upp pluginen inne i Jenkins, och sedan starta om Jenkins.

3.1.3 Teoretiskt resultat

När man följer diskussionen om Ruby och undersöker dem argument som finns för och mot Ruby respektive Java är Rubys skalbarhet ofta i centrum.

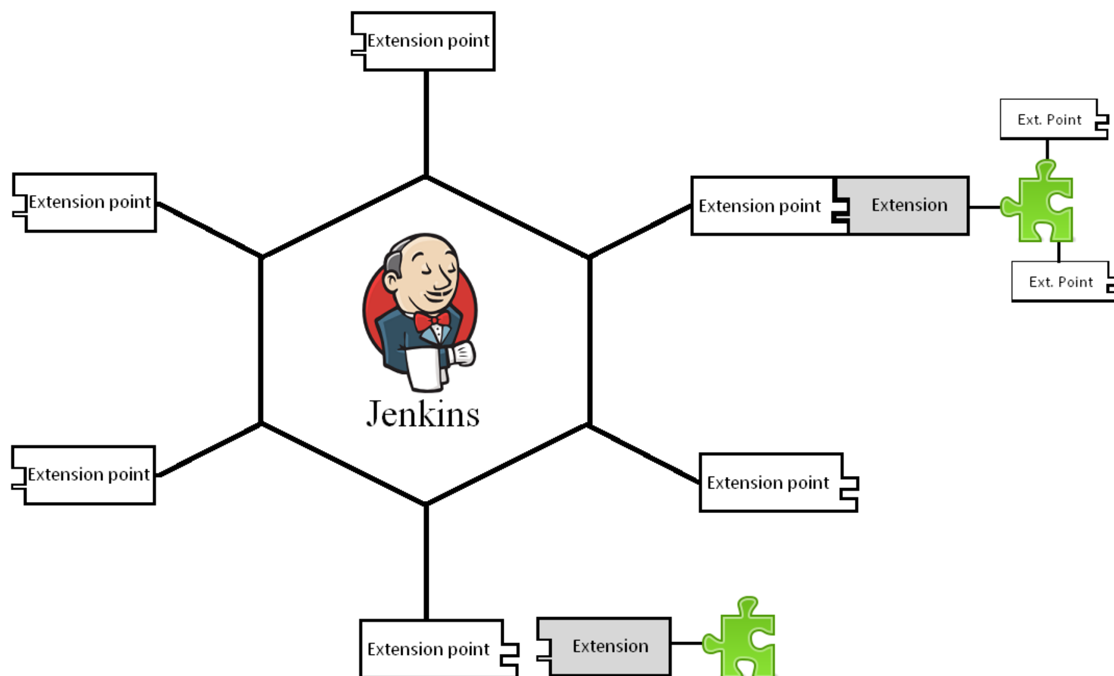
Huruvida Ruby skalar bra eller dåligt är en debatt som är intressant när man jämför språken, men inte särskilt relevant i vårt specifika sammanhang. När man utvecklar pluginer till Jenkins spelar det ingen roll hur bra språket skalar i större sammanhang eftersom en Jenkins master-server troligtvis aldrig kommer att ha i närheten av så mycket trafik att det skulle kunna bli ett problem.

Efter den teoretiska analysen kunde vi konstatera att Ruby har tydliga fördelar på flera områden. Men för att kunna avgöra om Ruby verkligen fungerar, utförde vi även en praktisk analys.

3.2 Praktisk analysdel

Under analysdelen av examensarbetet kändes det nödvändigt att skriva en plugin för att kunna utvärdera möjligheterna med Ruby. Därför utvecklade vi en prototyp till den plugin som varit på förslag från Autoliv, nämligen möjligheten att bli underrättad via email om en nod tappar kontakten med master-servern. Den första tanken vi hade var att pinga alla noder med jämna mellanrum för att se om de svarade, detta är dock ingen garanti för att noden faktiskt kör Jenkins fortfarande. Efter en del letande i Java-dokumentationen så fann vi en klass som gjorde precis det vi ville göra, `ComputerListener`, som har metoderna `onOffline` och `onOnline` bland annat. För att kunna lägga till konfigurationsalternativ för respektive nod, använde vi oss av klassen `NodeProperty`. Dessa två klasser fanns inte tillgängliga i Ruby när vi började skriva vår plugin, men efter kontakt med Charles Lowell och ett två-timmars möte i Meeting Center så var de färdiga att användas i *ren* Ruby.

Vi kommer att närmare beskriva utveckling av pluginer till Jenkins under avsnittet om utvecklingsprocessen (avsnitt 4), men för att kunna redogöra för denna praktiska analysdel behövs en snabb introduktion. Pluginutveckling för Jenkins handlar om att subklassa abstrakta klasser – extension points, fördefinierade i Jenkins – som kommer att instansieras och användas av Jenkins om pluginen finns i Jenkins plugin-katalog när Jenkins startar, om inte pluginen är avaktiverad inne i Jenkins. Varje extension point är en öppning in i Jenkins och en möjlighet att påverka Jenkins funktionalitet. För att bygga ut denna funktionalitet registrerar man en *extension*, alltså subklassar den abstrakta klassen. Man kan också definiera egna extension points i sina pluginer, som andra pluginer kan använda sig av. Figur 3.2 är en visuell representation av hur pluginer kopplas in i Jenkins.



Figur 3.2 Jenkins med dess extension points och pluginer.

När man registrerat en extension har man en ingång i Jenkins. Då vill man ofta, i en plugin, implementera olika Jenkins-specifika klasser – t ex `Computer` som representerar en dator: antingen mastern eller en slavinstans som är uppkopplad mot mastern. Många Jenkins-specifika Java-klasser har en Ruby-variant och en *proxy* som fungerar som ett gränssnitt till Ruby-varianten. M h a proxyn kan man komma åt det bakomliggande Ruby-objektet. På samma sätt har alla Ruby-klasser en *wrapper* som har en referens till motsvarande Java-objekt. Ruby-klasser kan med andra ord benämnas *Ruby wrappers*.

Vår missuppfattning var från början att det som avgör vad som är möjligt att göra i en plugin i Ruby är huruvida det, till motsvarande Java-klass, finns en wrapper skriven i Ruby eller inte. Utvecklingen av dessa Ruby wrappers drivs som ett hobbyprojekt av ett fåtal utvecklare på fritiden, det går därför ganska långsamt framåt. Det finns cirka hundra extension points i Jenkins och för närvarande finns ett tiotal wrappers i Ruby. Detta hade varit väldigt begränsande, men som tur är fungerar det inte så.

Den stora vändpunkten kom efter samtal med utvecklaren Charles Lowell(2012a), som ligger bakom ramverket för Ruby, när han förklarade att det i Ruby alltid går att falla tillbaka på javabiblioteket. Ramverket är nämligen utformat så att pluginutveckling alltid ska vara möjlig i Ruby även om det släpar efter motsvarigheten i Java. Här blir objektet `native` användbart.

Då man stöter på ett objekt till vilket det inte finns en wrapper i Ruby, får man en *opaque java wrapper* som har ett objekt som heter `native`. Objektet `native` refererar till det underliggande Java-objektet och tillåter anrop på Java-metoderna som objektet har. Även om detta ger merarbete, är det enklare att arbeta med Ruby än med Java i det här fallet. Vi återkommer till `native` senare i rapporten.

Genom kontakt med utvecklarna av Ruby-Jenkins-projektet skrevs två Ruby wrappers som vi kunde använda oss av. Det hade fortfarande varit möjligt att

använda Java-versionerna av klasserna genom att kalla på dem i Ruby-filen och sedan implementera dem metoder som finns i dem, det här gjorde dock vårt arbete enklare samt att vi fick mer insikt i hur dessa wrappers är byggda.

3.2.1 Tester av verktyg för Ruby

Autoliv använder sig av plattformen Windows 7 på alla sina datorer och deras servrar kör Windows Server. Det är därför viktigt att de pluginer som utvecklas går att köra i Windows. Det första vi gjorde var att installera dem verktyg som behövdes på den dator vi blivit tilldelade här på företaget. Som dem andra datorerna så kör den Windows 7.

Förutsättningar

För att skriva pluginer till Jenkins i Ruby behöver man Java Runtime Environment (JRE) och JRuby installerat. Jenkins wikisida rekommenderar att man installerar RVM (Ruby Version Manager). Den finns dock bara tillgänglig för Unix-baserade operativsystem. Till Windows finns Pik som motsvarar RVM, men eftersom vi bara skulle utveckla i JRuby och inte snabbt behöver byta mellan olika interpretatorer, så installerade vi JRuby direkt på hårddisken utan någon versionshanterare och satte *path*-variabeln att peka på vår installation. Därefter installerades dem gems som behövdes. Ett gem är en form av Rubybibliotek.

Det man behöver för att kunna bygga pluginerna är ett gem som heter *jpi* (Jenkins Plugin) som för tillfället är i *pre-alpha*-stadium. I Windows var det omöjligt för oss att få jpi att fungera, scriptet gav väldigt långa felmeddelanden. Vi upptäckte ganska snart att vi inte var de enda som hade problem med jpi i Windows, därför vände vi vår blick mot Linux. Vi startade med att installera en *Virtual Machine* i Windows och valet föll på Linux Mint 12. I vår Linux-distribution satte vi återigen upp JRuby, den här gången med RVM, och installerade dem gems som behövs. För att man snabbt ska komma igång med utvecklingen så finns en prototyp som heter *ruby-prototype*[1]. Meningen med denna är att man snabbt ska kunna sätta sig in i hur pluginutvecklingen ser ut i Ruby och den utnyttjar ett flertal extension points som troligtvis är intressanta för ändamålet.

Efter en hel del problem lyckades vi bygga prototyp-pluginen och installera det i Jenkins. Tyvärr fungerade fortfarande inte pluginen och efter att ha pratat med utvecklaren bakom prototypen, Charles Lowell(2012b), stod det klart att vidareutvecklingen av ramverket för Ruby är orsaken till att det inte längre fungerar och utvecklaren underhåller inte heller prototypen. Det går dock att använda vissa delar av den för att se hur man implementerar olika extension points i Jenkins.

Eftersom prototyp-pluginen är ganska stor och innehåller väldigt många olika komponenter så tänkte vi att en enklare plugin var en bra inkörsport för att förstå hur de olika delarna hängde ihop. När man skapar ett nytt pluginprojekt i Java, skapas en ”Hello World”-plugin, den kändes lämplig att skriva om till

[1] https://github.com/cowboyd/jenkins-prototype-ruby-plugin

Ruby. Det fungerade utan problem och vi experimenterade en del med den pluginen innan vi gick vidare.

Problemkällor

Autoliv använder sig av en proxy som orsakade en hel del problem för oss när vi skulle bygga pluginer. Verktyget jpi försöker nämligen hämta den plugin som Ruby-pluginer är beroende av (Ruby Runtime), något proxyn inte släppte igenom. Eftersom vissa problem kan vara svåra att spåra tillbaka till proxyn, och sådan felsökning kan ta onödig tid, fick vi låna en brygga för att ansluta till Autolivs WiFi för att slippa de problemen. Senare under arbetet fick vi dock som krav att utvecklingsverktygen måste fungera bakom Autolivs proxy, för att pluginutvecklingen ska kunna ske på samma sätt som deras övriga projekt. Vi lyckades lösa de problem vi hade till en början genom att använda en alternativ proxy som enligt vår handledare är mindre restriktiv. Vi upptäckte dessutom att vi inte behöver använda jpi för att testa vår plugin, det går att starta en instans av Jenkins och ladda upp pluginen dit, och på så vis testa. Det tar lite längre tid än det enkla kommandot `jpi server`, men det fungerar utan problem.

3.2.2 Tester av verktyg för Java

Det verktyg som rekommenderas av Jenkins-ci för Javautveckling heter Maven och fungerar på ungefär samma sätt som jpi. Man skapar ett nytt projekt med kommandot `mvn hpi:create`, som genererar en mappstruktur komplett med tillhörande filer för pluginen. Maven är som förväntat mycket mer stabilt än motsvarande för Ruby, och i motsats till pluginer skrivna i Ruby, behöver inte Jenkins någon stödplugin för att kunna köra pluginen. Pluginer skrivna i Java måste dock kompileras, en process som tar ganska lång tid (upp till en minut även för väldigt små pluginer), detta kan bli ganska frustrerande när man testar små ändringar i pluginen. Detta beror till stor del på att Maven, som bygger pluginen, utför tester som involverar en uppstart av Jenkins, som tar en stund att genomföra. För att kompilera samt paketera källkoden till en plugin, kör man kommandot `mvn package`.

3.2.3 Java vs Ruby i Jenkins

Anledningen till att det utvecklas verktyg för pluginutveckling i Ruby är för att antalet kodrader minskas otroligt mycket jämfört med Java, samt att integrationen med Jenkins-klasserna är enklare. I Java måste man ange om det är en extension (`@extension`) man beskriver eller om man överskuggar metoder (`@Override`), det behöver och kan man inte specificera i Ruby-filerna.

I Java består en plugin av *.pom-, *jelly- och *.java-filer. I Ruby består den av *.pluginspec-, *.erb-, Gem- och *.rb-filer. Pom- och pluginspec-filen är en beskrivning av pluginen – vem som skrivit den samt en länk till repositoryt i vilket koden finns för beskådning. Specifikt för Ruby är att ett beroende måste anges i denna fil, eftersom Ruby-pluginer alltid är beroende av minst *Ruby Runtime*. Ruby har dessutom en extra fil som heter Gemfile. Den innehåller

dem gems som pluginen använder sig av och de kommer att bindas till pluginen när det byggs. Jelly-filerna innehåller xml-liknande syntax som beskriver de fält som ska finnas inne i Jenkins för konfigurering av pluginen. Erb-filen gör samma sak men är kodad i Ruby som en skriptfil.

I Bilaga B och C kan man se skillnaderna mellan en enkel implementation av klassen *ComputerListener* i samarbete med klassen *NodeProperty*, i Java och Ruby.

3.2.4 Praktiskt resultat

Efter att ha konstaterat att det går att bygga pluginer på Unix-baserade plattformar och skrivit en prototyp i Ruby så vågar vi säga att pluginutveckling i Ruby är att rekommendera.

3.3 Resultat och beslut

Beslutet togs i samråd med vår handledare. Vi lade fram argument för och emot de båda ramverken. Den enda betydande nackdelen vi kunde se med Ruby är att utvecklingen måste ske i en virtuell maskin, eftersom Autoliv uteslutande använder Microsoft Windows. Vår uppfattning är dock att det är acceptabelt, och om vi får spekulera så tror vi att jpi snart kommer att fungera på alla plattformar.

Vår handledare kom också med förslaget under mötet att vi, istället för att utveckla flera pluginer, skulle skriva testfall för den plugin vi redan har i prototypstadie, och automatisera bygget och testningen i Jenkins. Anledningen till detta är att Autoliv planerar att fortsätta pluginutvecklingen på detta sätt och vill veta hur väl det fungerar.

3.3.1 Beslutsgrund för Ruby

Vi anser att Ruby går att använda som utvecklingsspråk för pluginer, det finns inga hinder som utgör så stora problem att det finns anledning att välja Java.

Samtliga områden inom vilka Java hade fördel kommer antagligen med tiden att jämnas ut mellan Java och Ruby. Däremot kommer Ruby alltid att ha de fördelar vi nämnt över Java.

När man behöver använda en klass till vilken det än inte finns en Ruby wrapper kan man, som sagt, alltid falla tillbaka till javabiblioteket. Detta innebär att det mesta man kan göra i Java-pluginer, kan man också göra i Ruby-pluginer (enligt Lowell(2012a) kan man möjligtvis stöta på problem med *generics* i Ruby). Ibland måste man dock ta hjälp av Java.

Att använda Java i Ruby är varken krångligt eller tidskrävande. Därför ser inte vi bristen på Ruby wrapper som något problem.

3.3.2 Anledning till att Java väljs bort

Det finns fördelar med att skriva pluginer i Java, t ex att det finns väldigt många pluginer tillgängliga att ta hjälp ifrån. Verktygen är också stabilare att använda än motsvarigheten för Ruby. Men den extra komplexitet som tillkommer i Java är det som fäller avgörandet, eftersom pluginutvecklingen på Autoliv är tänkt att vara små korta sidoprojekt för att snabbt få ytterligare funktionalitet eller annat så lämpar sig Ruby generellt bättre.

4 Utvecklingsprocess

I detta avsnitt kommer vi att redogöra för utvecklingen av den plugin vi implementerade för Autoliv.

Som nämnts i avsnitt 3.2 har många Java-klasser en Ruby wrapper som gör det möjligt att komma åt ett Ruby-objekt från en Java-klass. På samma sätt har Ruby-klasserna Java-proxies som gör åtkomst till Java-klasserna möjliga. Här kommer vi att närmare beskriva denna teknik, som vi har använt oss av under utvecklingen.

4.1 Pluginutveckling i Ruby

När man utvecklar pluginer i Ruby behöver man känna till lite olika begrepp: proxy, wrapper och Stapler. Vi kommer att ta upp dem i detta avsnitt.

Proxy - getTarget

Pluginer skrivna i Ruby måste inkludera ett gem som heter *Jenkins-plugin-runtime*. Detta gem innehåller klasser skrivna i Ruby, bland annat två klasser för varje klass i Java: en som motsvarar den klass som finns i javabiblioteket och en proxyklass som kopplar ihop Ruby-klassen med motsvarande Java-klass. Exempel 4.1 visar Ruby-versionen av `ComputerListener.` (En modul, *module*, är en s k namnrymd, *namespace*. Allt som befinner sig mellan `module` och motsvarande `end` ingår i den modulen, som har strängen efter `module` som namn. Om man inkluderar en modul i en klass så har den klassen tillgång till allt i modulen.)

```
module Jenkins::Slaves
  module ComputerListener
    extend Jenkins::Plugin::Behavior
    […]

    def online(computer, listener)
    end

    def offline(computer)
    end
    […]
  end
end
```

Exempel 4.1 Ruby-version av ComputerListener

`ComputerListener` har dels metoderna `online` och `offline` deklarerade som anropas när en dator ansluter till eller kopplas ifrån mastern. Parametern `computer` är den dator som ansluter till mastern. Exempel 4.2 visar motsvarande proxyklass, `ComputerListenerProxy` (den inkluderar modulen `Proxy`).

```
module Jenkins::Slaves
  class ComputerListenerProxy < Java.hudson.slaves.ComputerListener
    include Jenkins::Plugin::Proxy
    […]

    def onOnline(computer, listener)
      @object.online(import(computer), import(listener))
    end

    def onOffline(computer)
      @object.offline(import(computer))
    end
  end
end
```

Exempel 4.2 ComputerListenerProxy

`ComputerListenerProxy` funkar som en adapter, eller proxy, mellan Java-klassen och Ruby-klassen. Dess uppgift är att vidarebefordra anrop på Java-klassens metoder till motsvarande metoder i Ruby-klassen. Ruby-metoderna heter lite annorlunda. Metodanropen `onOnline` och `onOffline` på Java-klassen vidarebefordras som anrop av `online` respektive `offline` på `@object`. Detta objekt, som finns i Jenkins::Plugin::Proxy, är en instansvariabel som innehåller en referens till Ruby-objektet.

Exempel 4.3 visar Jenkins::Plugin::Proxy.

```
module Jenkins
  class Plugin
    module Proxy
      extend Plugin::Behavior
      […]
      def initialize(plugin, object, *super_args)
        super(*super_args) if defined? super
        @plugin, @object = plugin, object
        @pluginid = @plugin.name
      end

      def getTarget
        @object
      end
      […]
    end
  end
end
```

Exempel 4.3 Jenkins::Plugin::Proxy

Proxyns `initialize` får dels parametern `object`, det Ruby-objekt som proxyn representerar. Detta tilldelas `@object`. Det finns en metod i `Proxy`, nämligen `getTarget`, som returnerar `@object`. M h a denna metod kan man komma åt ett Ruby-objekt bakom en proxy.

Wrapper - native

Vi beskrev `wrappers` och `native` i den praktiska analysdelen(avsnitt 3.2). Alla Ruby-objekt har en wrapper. Den ser ut som i exempel 4.4.

```
class Jenkins::Plugin
  module Wrapper
    extend Behavior

    attr_reader :native

    def initialize(*args)
      @native, = *args
    end
  end
end
```

Exempel 4.4 Wrapper

`Wrapper` har instansvariabeln `native` som är det Java-objekt som Ruby wrappern representerar.

När en klass finns tillgänglig i *Jenkins-plugin-runtime* så går det att inkludera den på normalt vis i Ruby och på så vis implementera den:

```
class MyComputerListener
  include Jenkins::Slaves::ComputerListener
```

Om den wrappern inte finns måste man ärva direkt från javabiblioteket. Ruby-klass med javabibliotek:

```
class MyComputerListener < Java.hudson.slaves.ComputerListener
```

I detta fall måste man dessutom registrera Ruby-klassen under klassens implementation, för att Jenkins ska hitta den:

```
class MyComputerListener < Java.hudson.slaves.ComputerListener
  […]
end Jenkins.plugin.register_extension MyComputerListener.new
```

Metodnamn i en wrapper är förkortade för att bättre överensstämma med Ruby-konventionerna.

Java: `void onOnline(Computer c, TaskListener listener)`

Ruby: `def online(computer, taskListener)`

Om det inte finns en Ruby wrapper för klassen så måste metodnamnen i Ruby-filen motsvara det i Java-klassen. Varken metoden eller parametrar behöver ha en angiven typ, utan det är endast metodnamnet som måste motsvara Java-varianten:

```
def onOnline(computer, taskListener)
```

Oavsett om det finns en wrapper eller inte så använder man *pure* Ruby när man skriver sina klasser. Alla objekt som inte har en wrapper får ett s k *opaque Java object*, vilket innebär att det är ett javaobjekt som paketerats för

Ruby. För att kunna kalla på någon av objektets metoder så måste man först kalla på `native`.

Stapler

Jenkins använder sig av något som heter *Stapler*. De klasser som har en grafisk vy är bundna till en URL med hjälp av Stapler. Klassen Hudson som är en *singleton*, är bunden till "/"-URL:en (root) och de övriga objekten är bundna utifrån denna URL. För att visa ett exempel så finns metoden `Hudson.getJob("foo")`, objektet som returneras från denna metod binds till URL:en `/job/foo/`. (Jenkins, 2011d)

Stapler har en referens till ett Javaobjekt, men konfigurationen för noden är däremot lagrad på Ruby-sidan. Därför behöver Stapler veta var den ska hämta konfigurationsvyn. För att göra det anropar man `getTarget`. Denna metod, som finns definierad i Jenkins::Plugin::Proxy, returnerar Ruby-objektet `@object`, som nämnts under rubriken "Proxy - getTarget". På så sätt vet Stapler vilket objekt som har vyn den ska hämta.

Vi har implementerat en klass som heter `EmailNodeProperty`. Den lägger till ett fält i varje nods konfigurationsvy i Jenkins i vilket man kan skriva in en eller flera emailadresser. Om man konfigurerar en nod i Jenkins och fyller i fältet `email` och sparar konfigurationen så instansieras `EmailNodeProperty`. Alla datorer som är uppkopplade till mastern har en instans av `Computer`. Varje dator har även en instans av `EmailNodeProperty` om detta email-fält är ifyllt. Exempel 4.5 använder alla begrepp som vi har förklarat i detta avsnitt och knyter ihop dem. Detta exempel hämtar en specifik nods konfigurationsvärden som finns lagrade i Jenkins xml-fil *config.xml*.

```
[…]

1 list = mComputer.native.getNode().getNodeProperties()
2 mEmailNodeProperty = list.getProperty("EmailNodeProperty")
3 emailNodeProp = mEmailNodeProperty.getTarget()
4 email = emailNodeProp.email

[…]
```

Exempel 4.5 `native` och `getTarget` in action.

Koden i exempel 4.5 hämtar denna ifyllda emailadress från datorn `mComputer`. Här har vi kallat på metoder från Java-klassen `Computer` med Ruby-kod genom att först anropa `native` på `mComputer`. Java-klassen `Computer` har ingen Ruby-wrapper så objektet `mComputer` är ett opaque Java object som bara har medlemmen `native` för att man ska kunna kalla på Java-klassen `Computer`'s metoder.

1. Vi lagrar resultatet i en lista över den specifika konfiguration som noden har. Denna lista kommer delvis att innehålla vår `EmailNodeProperty` som har en proxy.
2. Vi hämtar `EmailNodeProperty` ur listan.
3. Vi anropar `getTarget()` på `EmailNodeProperty` för att tala om för Stapler vilket objekt som har vyn i vilken datan finns som vi vill åt.

4.Emailadressen hämtas ur objektet.

4.2 Testning

Pluginutveckling i Ruby går snabbt och det krävs inte särskilt mycket kod för att få en fungerande plugin. Men för att få kod som är lätt att underhålla så behöver man ett flertal tester, för att på så sätt kontrollera att ändringar av den inre kodstrukturen inte förstör något. Autoliv har som önskemål att automatisera bygget och testningen av pluginen i Jenkins, men vi återkommer till det senare i detta kapitel.

Som vi nämnde tidigare använde vi oss av Test Driven Development (TDD). TDD har ett evolutionärt förhållningssätt till utveckling som går ut på att skriva test först, skriva tillräckligt med kod för att testet ska lyckas och sedan förfina och faktorisera (*refactor*) koden. Denna process repeteras till den punkt då programvaran är färdig. Figur 4.1 visar grafiskt hur det kan se ut.

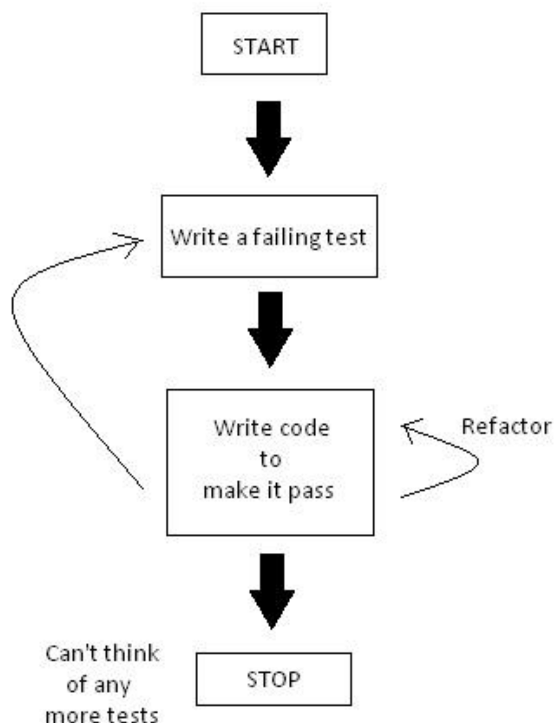


Fig. 4.1 Flödesschema över TDD

I exempel 4.6 kan man se några av de tester som vi skrev för vår plugin.

```
describe MyComputerListener do
  before :each do
    @mcl = MyComputerListener.new()
    Net::SMTP::restoreState
  end

  it "should send an email if the node is configured" do
    @mcl.offline(ConfiguredComputer.new("ThisIsASlave"))
    Net::SMTP::emailSent?.should == true
  end
  […]
  # motsatsen testas också
  […]
  it "should not send an email if smtp related values are not set" do
    @mcl.offline(NotConfiguredComputer.new("ThisIsASlave"))
    Net::SMTP::areValuesSet?.should == false
  […]
end
```

Exempel 4.6 mycomputerlistener_spec.rb-filen där alla tester är definierade.

Testbatteriet börjar med att skapa en ny instans av klassen `MyComputerListener` och sedan anropa metoden `restoreState` i SMTP-klassen (vår egen SMTP-klass som används för testningen), detta anrop nollställer alla variabler och sätter `emailSent` till `false`. Detta block kommer också att köras innan varje test, som `before :each do` indikerar. Syntaxen är ganska självbeskrivande: ett test börjar med `it` följt av en textsträng som beskriver vad testet förväntar sig för resultat. Denna textsträng skrivs ut om testet misslyckas. Därefter använder vi vår instans av `MyComputerListener` och anropar `offline`-metoden med en ny instans av `Computer`. Denna `Computer` är också en mock-klass som har väldigt begränsad funktionalitet jämfört med originalet. `offline`-metoden kommer att hämta konfigurationen och skicka ett email och det vi testar är slutresultatet; med en rätt konfigurerad `Computer`, anropades `Net::SMTP.start?`, `emailSent?.should == true`.

4.3 Integration med Jenkins

För att Autoliv ska kunna jobba med pluginutveckling på samma sätt som de utvecklar annan programvara så måste det integreras i Jenkins. Det innebär ett eget repository, en egen nod (slav) samt ett jobb som sköter bygget och testningen av pluginen. Den dator vi har arbetat med på Autoliv kör Linux Mint 12 i en virtuell maskin med Windows 7 som värdoperativsystem. Programvaran som Autoliv använder för att automatiskt ladda upp kod till sitt repository är Windowsbaserat, vilket innebär att vi måste dela en katalog mellan Windows och Linux för att kunna få till automatiken. I figur 4.2 kan man se ett av de skript som utförs i jobbet som testar och bygger vår plugin.

Build

Execute shell

Command

```
#!/bin/bash -x
source $HOME/.bash_profile
rvm jruby

rake all
jpi build
```

See the list of available environment variables

Add build step

Post-build Actions

- [ ] Publish duplicate code analysis results
- [ ] Scan for compiler warnings
- [ ] Scan workspace for open tasks
- [x] Activate Chuck Norris

Figur 4.2 Ett urklipp från jobbet som testar och bygger pluginen.

Beroende på vilket operativsystem som skriptet ska köras på, lägger man till ett lämpligt *build step*. I *Execute Shell*, som syns i figuren, anropar vi skript som testar och bygger vår plugin. Det som syns nedanför *build steps* är saker man vill att Jenkins ska utföra efteråt. Det kan t ex vara att Jenkins ska meddela någon via email om jobbet gick dåligt, publicera en HTML-rapport eller kommunicera med versionshanteringssystemet. Det finns dessutom inställningar för saker den ska utföra innan jobbet startar med kompilering av kod, utförande av tester och annat. Vi har t ex angett att *workspace*, platsen på den lokala hårddisken som koden kopieras till för att byggas, ska raderas innan den går vidare med bygget av pluginen.

Eftersom den startar ett helt nytt skal i Linux så måste vi ange vart den hittar de kommandon som vi anropar (t ex `jpi`). Det vi måste ange är att *Ruby Version Manager* (RVM) ska köras som en funktion, vilket ligger lagrat i `bash_profile`, och sedan säga att RVM ska köra JRuby med `rvm jruby`. Först då känner skalet till vad `rake all` och `jpi build` faktiskt betyder.

Versionshanteringssystemet som Autoliv använder heter MKS Integrity, där lagrar Autoliv sin källkod, och även vår kod ligger där. Källkoden är skrivskyddad så länge man inte *checkar ut* sin kod, denna information lagras i MKS och när man *checkar in* koden, anger man en kort beskrivning på vad man har gjort för ändringar i koden. I Jenkins kan man lägga till en *pollning* av versionshanteringssystemet, då upptäcker Jenkins automatiskt när en förändring har skett och ett jobb schemaläggs. När slaven som ska köra jobbet är tillgänglig (online eller om den inte gör något annat), startar jobbet automatiskt. Om jobbet misslyckades eller om något test misslyckades, så kan man ange att Jenkins ska skicka ett email till personen/personerna som är ansvariga för projektet.

4.4 Problemkällor under utvecklingen

När vi använder Ruby som ramverk för våra pluginer så introduceras ytterligare problemkällor. Den plugin som Ruby-pluginer är beroende av, Ruby Runtime, kör den version av JRuby som var aktuell då pluginen byggdes. När man kör JRuby i ett konsolfönster kan man ange vilken version av Ruby som JRuby ska använda sig av, 1.8 eller 1.9. För att undvika eventuella problem med en specifik version av Ruby, har det diskuterats(Lowell, 2012c) om man ska ha möjlighet att ange om JRuby ska använda Ruby 1.8 eller 1.9, i pluginspec-filen. Från Ruby Runtime 0.8 och framåt så är man låst till Ruby 1.9.

Vidare upptäckte vi att `ComputerListener`-klassen i Jenkins *core* innehöll en allvarlig bugg. Metoden `onOffline` kallades aldrig på när en nod tappade kontakten med servern genom abrupt avstängning (om noden stängs av utanför Jenkins kontroll, eller att någon “rycker sladden”), och eftersom wrapper-klassen i Ruby i själva verket anropar metoden i javabiblioteket så följer buggen med i Ruby. Vi rapporterade in detta till Kohsuke som fixade denna bug.

5 Slutsats

Vi har genomfört en teoretisk och praktisk analys av ramverket för Java och Ruby, fokus hamnade på Ruby eftersom det var det ramverk som vår handledare var mest intresserad av. Vi har kommit fram till att Ruby är fullt möjligt att använda för pluginutveckling. Det är ett snabbt sätt att utveckla pluginer och man behöver inte ha lika stor kännedom kring Jenkins klassbibliotek. Komplexiteten på Java-sidan är gömd inne i de Ruby wrappers som man använder, vilket gör att utvecklingen tenderar att gå lite fortare. Detta är positivt ur Autolivs perspektiv eftersom tanken är att man snabbt ska kunna ta fram en plugin när behovet uppstår.

Vi skrev en prototyp av den plugin som Autoliv hade som önskemål och byggde sedan vidare på den allt eftersom vår handledare kom med nya förslag på vad pluginen skulle kunna göra. Eftersom vi använde oss av testdriven utveckling så finns ett testbatteri som testar funktionaliteten av pluginen som den ser ut idag. Det innebär att Autoliv själva kan faktorisera och förfina koden och samtidigt testa att funktionaliteten inte har ändrats.

Förhoppningen är också att Autoliv kommer att kunna utveckla egna pluginer, m h a den guide som finns som bilaga A. Den innehåller dels installationsinstruktioner som beskriver hur man kommer igång, och dels ett avsnitt om de problem som man kan stöta på medan man utvecklar och testar, och lösningen på dessa.

Om vi hade ytterligare 10 veckor hade vi dels kunnat fortsätta utveckla pluginer, men också hjälpa till med utvecklingen av Jenkins-plugin-runtime, t ex att skriva s k wrappers för klasser som ännu inte existerar. Det tror vi hade varit väldigt roligt och samtidigt en lämplig förlängning av arbetet.

Referensförteckning

Campos, D. (2011) *About JRuby*. Tillgänglig på Internet:
https://github.com/jruby/jruby/wiki/AboutJRuby [Hämtad 26 jan 2012]

Fox, J. (2006) *Ruby for the Java world*. Tillgänglig på Internet:
http://www.javaworld.com/javaworld/jw-07-2006/jw-0717-ruby.html
[Hämtad 26 jan 2012]

Fowler, M. (2006) *Continuous Integration.* Tillgänglig på Internet:
http://martinfowler.com/articles/continuousIntegration.html [Hämtad 26 jan 2012]

Github (2009) Jenkins på github. Tillgänglig på Internet:
https://github.com/jenkinsci [Hämtad 24 feb 2012]

Github (2012) *jenkins-plugin-runtime.* Tillgänglig på Internet:
https://github.com/cowboyd/jenkins-plugin-runtime/tree/master/lib/jenkins
[Hämtad 7 mars 2012]

Gosling, J. (2008) *Ruby can't scale as good as Java.* [video online] Tillgänglig på:
http://www.youtube.com/watch?v=uq25JnHrF14 @2:16 [Hämtad 26 jan 2012]

Hudson (2011) *The Future of Hudson.* Tillgänglig på Internet:
http://hudson-ci.org/docs/news.html#future [Hämtad 24 feb 2012]

Java, The Java Programming Language (1997) Tillgänglig på:
groups.engin.umd.umich.edu/CIS/course.des/cis400/java/java.html
[Hämtad 23 feb 2012]

Jenkins (2011a) *Jenkins, An extendable open source continuous integration server*.
Tillgänglig på Internet:
http://jenkins-ci.org [Hämtad 24 feb 2012]

Jenkins (2011b) *Hudson's future.* Tillgänglig på Internet:
http://jenkins-ci.org/content/hudsons-future [Hämtad 24 feb 2012]

Jenkins (2011c) *Who is using Jenkins.* Tillgänglig på Internet:
https://wiki.jenkins-ci.org/pages/viewpage.action?pageId=58001258
[Hämtad 24 feb 2012]

Jenkins (2011d) *Architecture*. Tillgänglig på Internet:
https://wiki.jenkins-ci.org/display/JENKINS/Architecture [Hämtad 2 mar 2012]

Jenkins (2011e) *Extend Jenkins.* Tillgänglig på Internet:
https://wiki.jenkins-ci.org/display/JENKINS/Extend+Jenkins [Hämtad 6 mar 2012]

Jruby.org (2012) *JRuby 1.6.6 Released.* Tillgänglig på Internet:
http://jruby.org/2012/01/30/jruby-1-6-6.html [Hämtad 6 mar 2012]

Karandikar, K. (årtal saknas) *Java in education.* [PDF] Tillgänglig på Internet:
http://www.edb.utexas.edu/minliu/multimedia/PDFfolder/JAVA_I~1.PDF
[Hämtad 7 mar 2012]

Lowell, C. (2011) *Ruby for Jenkins Goes Pre-Alpha*. The Frontside Blog.
Tillgänglig på Internet:

http://blog.thefrontside.net/2011/09/13/ruby-for-jenkins-goes-pre-alpha/
[Hämtad 13 feb 2012]

Lowell, C. (2012a) Utvecklare bakom Jenkins ramverk för Ruby, IRC-konversation 9 februari 2012.

Lowell, C. (2012b) Utvecklare bakom Jenkins ramverk för Ruby, IRC-konversation 2 februari 2012.

Lowell, C., Kawaguchi, K. (2012c) Diskussion i Meeting Center med Lowell och Kawaguchi, 16 feb 2012

Pragmatic Automation (2005) *Bubble, Bubble, Build's In Trouble.*
Tillgänglig på Internet:
http://www.pragmaticautomation.com/cgi-bin/pragauto.cgi/Monitor/Devices/BubbleBubbleBuildsInTrouble.rdoc
[Hämtad 6 mars 2012]

Roberts, M. & Cameron, D. (2009) *What is Continuous Integration.*
Tillgänglig på Internet:
http://confluence.public.thoughtworks.org/display/CCNET/What+is+Continuous+Integration [Hämtad 6 mars 2012]

Robertsson, I. (2011) *Oracle forks Jenkins.* Tillgänglig på Internet:
http://www.artima.com/weblogs/viewpost.jsp?thread=317610 [Hämtad 24 feb 2012]

Ruby-lang.org (2011) *Ruby 1.9.3 p0 is released*. Tillgänglig på Internet:
http://www.ruby-lang.org/en/news/2011/10/31/ruby-1-9-3-p0-is-released/
[Hämtad 6 mar 2012]

Ruby-lang.org (2012) *About Ruby*. Tillgänglig på Internet:
http://www.ruby-lang.org/en/about/ [Hämtad 6 mar 2012]

Shore, J. (2005) *Continuous Integration is an Attitude, Not a Tool.*
Tillgänglig på Internet:
http://jamesshore.com/Blog/Continuous-Integration-is-an-Attitude.html
[Hämtad 1 mar 2012]

Thomas, D., Fowler, C & Hunt, A. (2010) *Programming Ruby 1.9 – The Pragmatic Programmer's Guide.* The Pragmatic Bookshelf.

Wells, D. (2000) *Code the unit test first*. Tillgänglig på Internet:
http://www.extremeprogramming.org/rules/testfirst.html [Hämtad 7 mar 2012]

Zygmuntowicz, E. (2008) *Ruby on Rails expert: Rails scales; Twitter shouldn't taint Ruby.* Tillgänglig på Internet:
http://www.zdnet.com/blog/btl/ruby-on-rails-expert-rails-scales-twitter-shouldnt-taint-ruby/8878 [Hämtad 15 feb 2012]

Bilaga A

En guide till pluginutveckling i Ruby.

Instruktioner för installation:

- Se till att ha en Unix-baserad plattform. Inte MS Windows.
- Öppna en terminal.
- Installera Ruby Version Manager (RVM) med kommandot:
  ```
  bash -s stable < <(curl -s
  https://raw.github.com/wayneeseguin/rvm/master/binscripts/rvm-installer )
  ```
- ```
  echo '[[ -s "$HOME/.rvm/scripts/rvm" ]] && . "$HOME/.rvm/scripts/rvm" # Load RVM
  function' >> ~/.bash_profile
  ```
- Ladda om din `.bash_profile` med kommandot:
  ```
  source ~/.bash_profile
  ```
  (Fullständiga instruktioner på http://beginrescueend.com)

- Installera senaste JRuby: `rvm install jruby`
- Instruera RVM att använda JRuby: `rvm use jruby`
- Installera de gems som behövs för att bygga Ruby-pluginer:
  ```
  gem install jpi
  ```
  Test- och Jenkinsrelaterade verktyg:
  ```
  gem install rspec
  gem install ci_reporter
  ```

Instruktioner för att skapa en ny plugin:

```
my-first-plugin
/     models
      /     mComputerListener.rb
      /     mNodeProperty.rb
/     views
      /     m_node_property.erb
/     pkg
      /     my-first-plugin.hpi
      /     vendor
            /     […]
/     my-first-plugin.pluginspec
/     Gemfile
```

Figur A1 Pluginens katalogstruktur

- Skapa ett nytt projekt: `jpi new my-first-plugin`
  Pluginkatalogen my-first-plugin skapas med två filer, *my-first-plugin.pluginspec* och *Gemfile*.
- Skapa två underkataloger, *models* och *views*
- I katalogen models skapar man sina Ruby-filer, exempelvis *my-plugin.rb*
- I katalogen views behövs en underkatalog som är döpt efter klassnamnet i *my-plugin.rb*. Om din klass använder sig av en vy-extension
```

(exempelvis `NodeProperty` eller `BuildWrapper`) och din klass heter MyPluginClass, så måste katalogen heta my_plugin_class. I denna katalog ska config.erb placeras.
ERB-filen innehåller deklarationer av de textboxar(fält) och checkboxar som ska dyka upp i vyn som ERB-filen beskriver. Här kommer ett exempel på hur ERB-filen kan se ut:

```
<%
   f = taglib("/lib/form")
   f.block do
      f.entry(:title => 'Email address',
              :field => 'email',
              :description => 'White space separated emails to notify if this
node goes offline') do
         f.textbox
      end
   end
%>
```
Exempel A1 ERB-fil

`:title` är den titel som står ovanför boxen

`:field` är boxens id med vilket man kan komma åt boxen programmatiskt

`:description` är den beskrivande textsträng som står under boxen

`f.textbox` bestämmer om det ska vara en textbox eller checkbox

Var i Jenkins som vyn kommer att vara beror på vad Ruby-klassen som vyn tillhör implementerar. Om den implementerar en BuildWrapper hamnar den post build actions i jobbkonfigurationen, om det är en NodeProperty hamnar den i nodkonfigurationen.

- Under tiden du skriver din plugin så är det lämpligt att ha tillgång till Jenkins-plugin-runtime-biblioteket för att se vilka wrappers som finns tillgängliga, samt Java-Doc för Jenkins, för de klasser som saknas i Jenkins-plugin-runtime.
- För att bygga pluginen ställer du dig i pluginkatalogen och kör kommandot: `jpi build`
  Det skapar en katalog som heter *pkg* i pluginkatalogen. I denna finner du *my-first-plugin*.hpi. Denna HPI-fil kan laddas upp i Jenkins, `Manage Jenkins > Manage Plugins > Advanced`, och kommer att vara aktiv efter en omstart av Jenkins, förutsatt att Ruby-Runtime också är förinstallerat.
- Alternativet är att, i pluginkatalogen, köra: `jpi server`
  Då startas en Jenkins-server med enbart pluginen installerad tillsammans med Ruby-Runtime som behövs för att köra den. Detta är det snabbaste sättet att testköra pluginen på.

Testning:

- Skapa en katalog som heter t ex spec och skapa i den en *_spec.rb-fil där du skriver dina testfall.

- Skapa hjälpfiler till din spec-fil, t ex om man inte vill att den ska använda sig av den riktiga Net::SMTP så kan man skapa mock-klasser som den klass du vill testa kallar på istället för den verkliga Net::SMTP-klassen.
- I root-katalogen till din plugin skapar du filen *Rakefile* och riktar den till de filer du vill testa.

```
require 'rspec/core/rake_task'
require 'ci/reporter/rake/rspec'

RSpec::Core::RakeTask.new(:all => ["ci:setup:rspec"]) do |t|
  t.pattern = 'spec/*_spec.rb'
end
```

Exempel A2 Rakefile

När Rakefile exekveras kommer den att köra alla tester i `spec/*_spec.rb`.

- Öppna en terminal och skriv `rake all`, testerna utförs och du får direkt svar vilka tester som eventuellt misslyckades.

Låt Jenkins utföra jobbet:

- Se till att koden finns i ett repository i t ex MKS.
- Starta ett nytt jobb, på en lämplig slav.
- Lägg till pollning av versionshanteringssystemet.
- Lägg till ett *build step* → Execute shell och skriv

```
#!bin/bash -x
source $HOME/.bash_profile
rvm jruby

rake all
jpi build
```

- Kryssa i *Archive the artifacts*, och ange → **/*.hpi
- Fyll i att du vill publicera JUnit test results (ci_reporter) → **/reports/*.xml
- Spara och kör jobbet för att se att det gick bra.
- Fr o m nu kommer Jenkins polla repositoryt och märka om det uppdaterats, då schemaläggs ett jobb som kommer att köras så fort slaven blir ledig.

Kända problemkällor:

- Konfigurationsfönstret där pluginen ska bygga ut funktionalitet, visar detta fel: Status code null, Stack trace not available.
  → Felaktig syntax i *.erb-fil eller felaktig katalogstruktur.
- Om man förändrar något som får Jenkins att krascha och det därför inte går att avaktivera pluginen så kan man, i `$JENKINS_HOME/plugins`, manuellt radera JPI-filen samt katalogen som tillhör pluginen.
- Om din plugin förändrar `config.xml`, och det orsakar krasch, ofta `hudson.util.IOException2: Unable to read */config.xml`, radera dem raderna manuellt från `config.xml` som ligger i `$JENKINS_HOME`. Filen `config.xml` har en tagg <slave> under <slaves> för varje registrerad slav. Denna tagg kan ha subtaggen <nodeProperties> som i sin tur har en subtagg för varje alternativ som är aktiverat under *NodeProperties* på nodkonfigurationssidan i Jenkins. T ex, om du

manuellt har tagit bort en `NodeProperties`-plugin(enligt punkten ovan) som förut exekverat och <nodeProperties> därför har subtaggen <ruby-proxy-object>(i detta fall), så tar du bort den subtaggen(med dess eventuella subtaggar).

```
[…]
<slave>
  <name>foo</name>
  <mode>NORMAL</mode>
  […]
  <nodeProperties>
    <ruby-proxy-object>
      <ruby-object ruby-class="Jenkins::Slaves::NodePropertyProxy"
pluginid="nodeofflineemailer">
        […]
        <object ruby-class="class_name" pluginid="plugin_name">
          <email pluginid="name" ruby-class="String">email</email>
        </object>
      </ruby-object>
    </ruby-proxy-object>
  </nodeProperties>
</slave>
[…]
```

Exempel A3 config.xml

Bilaga B

Enkel implementation av ComputerListener i Java, hämtar värdet från en NodeProperty.

```
package org.sample.jenjondev;

import java.io.IOException;
import java.util.logging.Level;
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;

import hudson.Extension;
import hudson.model.Computer;
import hudson.model.TaskListener;
import hudson.slaves.ComputerListener;
import hudson.model.Node;
import hudson.util.DescribableList;
import hudson.model.Descriptor;
import hudson.model.Describable;
import hudson.slaves.NodeProperty;
import hudson.slaves.NodePropertyDescriptor;

@Extension

public class MyComputerListener extends ComputerListener {
  public void onOnline(Computer computer, TaskListener listener){
    Node node = computer.getNode();

    DescribableList<NodeProperty<?>, NodePropertyDescriptor>
    dList = node.getNodeProperties();
    Iterator<NodeProperty<?>> iterator = dList.iterator();
    while (iterator.hasNext()) {
      NodeProperty<?> next = iterator.next();
      MyNodeProperty nodeProp = (MyNodeProperty)next;
      System.out.println(nodeProp.getEmail());
    }
  }
}
```

Bilaga C

Samma klass som i bilaga B, men implementerat i Ruby.

```
class MyComputerListener
  include Jenkins::Slaves::ComputerListener
  include Jenkins::Plugin::Proxy

  def online(computer, taskListener)
    list = computer.native.getNode().getNodeProperties()
    nodeProp = list.find {"EmailNodeProperty"}
    nodeProp = nodeProp.getTarget()
    puts nodeProp.email
  end
end
```