OSS を使用したインテグレーション作業の自動化

Automatic Integration Environment built by leveraging Open Source Software

山　田　暁　広

要　約　インテグレーションの自動化とは，ビルド（ソースコードのコンパイルを行い，最終的な実行可能ファイルを作成すること）やプログラムのユニットテストといったソフトウェア開発後期に行われるインテグレーション作業をコンピュータ上で自動的に実行するものである．インテグレーションの自動化により，インテグレーションにおける問題の早期発見と対応，手作業によるミスの低減，ひいてはソフトウェア開発プロジェクトでのリスク低減を図ることができる．

また，個々のプロジェクト状況に応じて低コストで柔軟にインテグレーションの自動環境を構築するには，オープンソースソフトウェアを組み合わせて利用することが得策である．

本稿では，インテグレーションの自動化の目的と背景を説明し，オープンソースソフトウェアを組み合わせて実現した T 社 PDM システム開発でのインテグレーションの自動化環境を紹介する．

Abstract　Automation of integration refers to automatically and electronically perform the integration works in the later phases of software development such as building (compiling the source codes and making the final executable file) and program unit testing. It allows early detection of integration problems and reduction of manual-handling errors, and thus eventually mitigates risks of software development project.

For the purpose of building the environment for automatic integration that flexibly responds to individual project status, the author attempted to leverage open source software. The use of open source software enables to build a low-cost and adaptable integration environment.

This paper expatiates on the purpose and backdrop of the automation of integration, and introduces the automatic integration environment that was built for a PDM system development for Company T by leveraging open source software.

1. は　じ　め　に

インテグレーションの自動化とは，開発対象システムのソースコードのコンパイル，モジュールの結合，単体テストといったシステム開発の後期に行われるインテグレーション作業をコンピュータ上で自動化するものである．また近年，CI（Continuous Integration，継続的インテグレーション[*1]）とも呼ばれるプラクティスが注目を浴びており，その実現方法としてインテグレーションの自動化が重要となっている．

インテグレーションは一般的に労力がかかる作業である．現在のソフトウェア開発では複数人のメンバーが並行して開発し，開発終盤でインテグレーションを行っている．このインテグレーションにおいて，コンパイルエラー，インタフェースの不一致による不具合，環境に依存した不具合などが表面化する．したがってソフトウェア開発において，インテグレーション作

業は一度ではなく複数回実行される．

こうしたインテグレーション作業を手作業で行った場合，多くの労力がかかる．プロジェクト規模が広がり，関係者が増えるほどインテグレーションが困難になり，プロジェクト全体のリスクにもなりかねない．

したがって，インテグレーション作業そのものの自動化が重要となる．自動化によりインテグレーションのコストを低減し，頻繁なインテグレーションを可能とする．その結果，インテグレーションに関連する問題の早期発見と対応，あわせてプログラム品質の見える化とプロジェクトリスクの低減が可能となるのである．

インテグレーションにおける問題は，T 社の PDM システム開発プロジェクトにおいても以下のような形で発生した．

- コンパイルエラーのまま総合テスト環境にリリースした
- インタフェースの不一致に気づかず後工程で発覚した
- 単体テストレベルでのデグレードの頻発

こうした課題に対して手作業による対応では労力がかかることもあり，インテグレーションの自動化をプロジェクトとして取り組むこととなった．当初，インテグレーションの自動化の実現方法として既製品を検討したが，

- プロジェクト自体に予算の制約があり，高価な既製品のインテグレーションツールを購入するには負担が大きい
- 既製品のインテグレーションツールによって，進行しているプロジェクトの開発プロセスを変えてしまうリスクが大きい

といったデメリットがあったため，既製品の採用を見送り，無料で自由に組み合わせることが可能なオープンソースソフトウェアを使って，インテグレーションの自動化環境を構築することとなった．

本稿ではインテグレーションの自動化の目的と背景，実際に T 社の PDM システム開発プロジェクトに適用したインテグレーションの自動化環境を紹介する．

2. 「インテグレーションの自動化」について

インテグレーション作業は定型的で単調な作業であり，単調な作業ほどミスを誘発しやすい．また，インテグレーションを繰り返し実行するのは労力が大きい．担当者を専任し，数時間かけて繰り返しイテレーションを実行することは非現実的である．したがって，継続的インテグレーションの実現においても，インテグレーションの自動化が必要である．

2.1 インテグレーションの自動化とは

継続的インテグレーションにおける主要な作業として，ビルドとユニットテストがある．この二つの作業を自動化することが，継続的インテグレーションには不可欠である．

また，自動化されたインテグレーションのプロセス内で副次的にプログラムの品質指標を取得することができる．つまり，インテグレーションの自動化によってプログラム品質の見える化という効果を得ることができる．

2.1.1　ビルドの自動化とは

ビルドとは，複数のソースファイルから実行可能なソフトウェアを構築する作業のことである．ソフトウェア開発では結合テストや統合テストといった節目でこのビルドを実施する．システムが本番を迎えるまでにはこうしたビルドを繰り返し，動作確認，不具合検出，修正を行っていく．こうした一連の作業を，コンピュータ上で自動的に実施するのがビルドの自動化である．

2.1.2　テストの自動化とは

ここで述べるテストとは，ユニットテストを示している．現在のJava開発ではプログラムコードとは別に，JUnit（3.1節で述べる）と呼ばれるオープンソースソフトウェアを使ってテストプログラムを作成し，ユニットテストを実行することが一般的である．JUnitは外部からバッチ的にテストプログラムを実行することが可能であり，そのテスト結果を自動的にレポートすることもできる．このようなJUnitの仕組みを使い，テストの自動化を行っている．

2.1.3　プログラム品質の見える化とは

「プログラムはどれくらい複雑か」，「規約に沿ったプログラムであるか」といったプログラムの品質指標を，インテグレーションの自動化の一環として確実に低コストで求め，明確に見えるようすることができる．つまりプログラム品質の見える化を行うことができる．

プログラム品質の見える化の対象として，一般的に使われる指標「テストカバレッジ」，「McCabeのサイクロマチック複雑度」について簡単に紹介する．

1）　テストカバレッジ

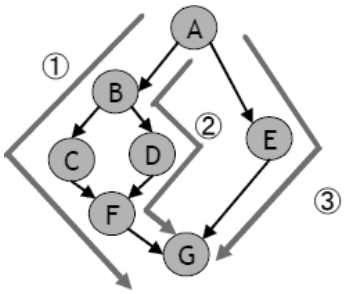
テストカバレッジはテストにより実行されたソースコードの網羅率のことである．つまりテストがもれなく実行されているかを確認する指標である．一般的には，表1のように定義され，C0が100%，C1を可能な限り100%に近づけることが単体テストの合格条件とされている．

表1　テストカバレッジの定義

略称	名称	内容
C0	命令網羅	ソースコード中の全命令を少なくとも1 回は実行する．
C1	分岐網羅	ソースコード中の全分岐のtrue とfalse の結果を少なくとも1回は通る．

2）　McCabeのサイクロマチック複雑度（CC（McCabe's Cyclomatic Complexity））

McCabeのサイクロマチック複雑度とはルーチン内での経路の数のことである（図1）．サイクロマチック複雑度が高くなればなるほど，そのルーチンが複雑になっていくことを示している．インテグレーションの自動化の一環として，このサイクロマチック複雑度を測定することが可能である．この指標が高いものについてはプログラムの改善を指示するなどしてプログラムの複雑化を抑制する．また，サイクロマチック複雑度は単体テスト項目数の算出基準となり，サイクロマチック複雑度の2倍から3倍が単体テスト項目数とされている．



この場合，経路は三つあるので，サイクロマチック複雑度は３．

図１　サイクロマチック複雑度

2. 2　インテグレーションの自動化の必要性

手作業にはミスがつきものである．開発中のプログラムやバグのあるプログラムをリリースするといったインテグレーションの問題がソフトウェア開発の現場では頻発する．手作業で行う限りこうしたミスが発生しうる．ならば，可能な限り手作業を排除し，自動化することによってインテグレーションで発生する問題を取り除くことが必要である．

またソフトウェア開発プロジェクトの規模が大きくなると，ユニットテストだけで数千個，数万個という膨大なテストを行うことになる．これらのテストをすべて手作業とすると，かなりの労力がかかることになる．こうした作業を自動化すれば，テスト作業効率を向上できる．また自動化されたインテグレーションでは，プログラムの品質についても自動的にレポーティングされ，効率よい状況の把握と対策が可能となる．

こうして自動化されたインテグレーションは，早く頻繁に実施することが可能となる．一般的にソフトウェア開発プロジェクトにおいてインテグレーションは一度の実行で済むことはなく，何度も実行される．コンパイルエラーやインタフェースの不一致といったインテグレーションに関する問題は，何度もインテグレーションを繰り返すことで問題を解消するのが現実的である．したがって，インテグレーションを円滑に行うことが必要であり，そのためにインテグレーションの自動化は不可欠なのである．

3.　オープンソースによるインテグレーション環境の構築

インテグレーションの自動化を実現するソフトウェアは販売されているが，高価であり，プロジェクトの採算，または購入手続きを考えればすぐに導入できるものではない．またそうした市販のソフトウェアがプロジェクトの流儀や開発プロセスに一致するかどうかもわからない．

インテグレーションの自動環境の導入コストとプロジェクトへの柔軟な適合という問題を解消するため，オープンソースソフトウェアを組み合わせて実現することとした．

本章では，実際に T 社の PDM システム開発プロジェクトで導入した，オープンソースソフトウェアを組み合わせたインテグレーションの自動環境について紹介する．なお，紹介するインテグレーションの自動化環境は T 社固有の要件は無く，汎用性を持った環境である．

3.1　使用したオープンソースソフトウェアについて

使用したオープンソースソフトウェアについて簡単に紹介する．

1）　Subversion[*3]

バージョン管理システムである．時間とともに変化するファイルやディレクトリを管理する．Subversionで開発対象のすべてのソースコードを一括管理する．

2）　Ant[*4]

Javaプログラム構築用に開発されたインテグレーションツールである．Antではビルドなどインテグレーションの手順をXMLで記述し，その内容に沿ってビルド，テストなどのインテグレーションのタスクを自動的に実行する．

3）　JUnit[*5]

Javaで開発されたプログラムにおいてユニットテストの自動化を行うためのソフトウェアである．

4）　CheckStyle[*6]

Javaソースファイルが決められた規約に従っているかをチェックするソフトウェアである．また，前述のサイクロマチック複雑度といったソースコードの品質指標を計測することができる．

5）　FindBugs[*7]

プログラムを動作させずに，バグの存在する可能性があるソースコードを自動的に指摘するソフトウェアである．指摘結果は図2のようにHTMLでレポーティングすることができる．

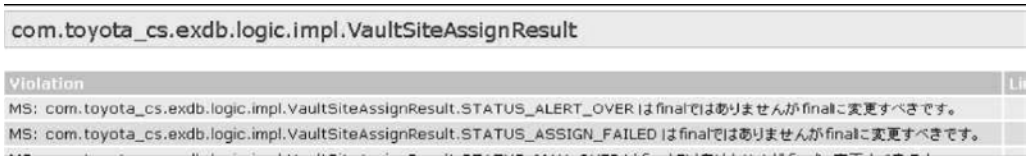


図2　FindBugsによるレポーティング

6）　dJUnit[*8]

JUnitで作成したテストプログラムのカバレッジを測定するソフトウェアである．

3.2　フィードバックデバイス

フィードバックデバイスとは，工場等で設備や環境の状態を人間に伝達する装置のことである．フィードバックデバイスの代表的な例として，自動車生産ラインでのアンドン[*9]がある．アンドンは異常状態を第三者に知らせにくい生産ラインにおいて，異常状態を他者に伝えることを目的としている．

インテグレーションの自動化に伴い，インテグレーションそのものがコンピュータ上で完結するため，作業そのものの状況が見えにくくなる．そのためインテグレーション作業の状況を第三者に伝達するフィードバックデバイスが必要となる．

インテグレーション作業のフィードバックデバイス実現について，T社PDMシステム開発プロジェクトでの取り組みについて簡単に触れる．T社PDMシステム開発プロジェクトでの

フィードバックデバイスは，ビルド，テストといったタスクごとにその状況をコンピュータディスプレイに表示することで実現した．（図 3）

図 3　PC ディスプレイを使用した実際のフィードバックデバイス

ビルドやユニットテストといったタスクの自動実行中に異常が発生すれば，画面上で赤く警告表示される．問題がなければ緑で表示されるようになっている．

このフィードバックデバイスの実装は，Microsoft Windows に標準で搭載されている HTA（HTML Application）を使用している．HTA ファイルをダブルクリックすると，ツールバーやアドレスバーのない IE（Internet Explorer）のウィンドウが起動し，HTML による画面表示や JavaScript の実行が可能となる．つまり，Web システムで使用される HTML 技術をクライアント PC だけで実現させる機構が HTA である．HTA の実装には HTML と JavaScript を用いる．この HTA の JavaScript からインテグレーションのタスクが実行され，タスクの状況に応じてフィードバックを行う仕組みとなっている．

3. 2. 1　フィードバックデバイスの実装例

HTA を使ったフィードバックデバイスの実装コードを以下に例示する．なお，HTA は 拡張子 hta のテキストファイルにコーディングし，そのファイルをダブルクリックすると実行することができる．

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="ja">
<head><title>Ant Controller</title>
<HTA:APPLICATION
    APPLICATIONNAME="Ant Controller" BORDER="thick" BORDERSTYLE="raised"
    CAPTION="yes" CONTEXTMENU="yes" ID="oHTA" INNERBORDER="no"                    1)
    MAXIMIZEBUTTON="yes" MINIMIZEBUTTON="yes" NAVIGABLE="no"
    SCROLL="yes" SCROLLFLAT="no" SELECTION="yes" SHOWINTASKBAR="yes"
    SINGLEINSTANCE="yes" SYSMENU="yes" VERSION="0.0.1" WINDOWSTATE="Maximize" />

<script type="text/javascript" src="constant.js"></script>
<script type="text/javascript" src="build.js"></script>
<script type="text/javascript">
  // 監視の開始
  function Start() {
    // 割込み設定
    TimerId = setTimeout("Interval()",1000);
  }
  // 監視中(割込み処理)                                                          2)
  function Interval() {
```

```
    main();
    TimerId = setTimeout("Interval()",18000000);
  }
  function main() {
    init();
    if(!build()) return;
    if(!checkstyle()) return;
    -- 略 --
  }
  function init() {
    document.getElementById("build").style.backgroundColor = "#FFFFFF";
    document.getElementById("arch_usol").style.backgroundColor="#FFFFFF";
  }
  window.onload=Start;
</script>
</head>
<body><table width="100%" BORDER="1" cellspacing="0">
  <tr><td align="center" height="150px" class="main" id="build"> ソースコードのビルド </td></tr>
  -- 以下, 略 --
```

3)

1)　画面サイズ，ボーダー線の有無など HTA の初期定義を行う

2)　Javascript のタイマー機能を使い，一定間隔でインテグレーションのタスクを実行する．この例では 3 時間に一度実行する

3)　インテグレーションのタスクを実行する JavaScript のメソッドをコールする

HTA は HTML と JavaScript で実装されていることが，上に示したソースコードからわかる．

上記の 3）でコールされるメソッドの実装例は次の通りである．ここではソースコードをビルドするタスクを実行するメソッドを例示している．

```
function build() {
  var buildbatch = BUILD_CMD;
  var WshShell = new ActiveXObject("WScript.Shell");
  var colEnv = WshShell.Environment("Process");
  colEnv.Item("CLASSPATH") = BUILD_CLASSPATH;
  colEnv.Item("CATALINA_HOME") = CATALINA_HOME;
  colEnv.Item("ANT_HOME") = "C:¥¥java-root¥¥apache-ant-1.6.5";
  colEnv.Item("ANT_OPTS") = "-Djava.util.logging.config.file=D:¥¥gha_builds¥¥logging.properties";
  colEnv.Item("PATH") = "C:¥¥j2sdk1.4.1_07¥¥bin; C:¥¥java-root¥¥apache-ant-1.6.5¥¥bin";
  WshShell.run(buildbatch,1,true);
  return build_end() ;
}
```

1)

2)

1)　外部プログラムを実行する WScript.Shell オブジェクトをインスタンス化する

2)　run メソッドにより，3.1 節で述べた Ant を外部プログラムとして実行する．

つまり，HTA 内の JavaScript より WScript.Shell の run メソッドを使って外部プログラムとして前述の Ant を実行しているのである．

3.3　インテグレーションの自動環境の構成と実行の流れ

前述のオープンソースとフィードバックデバイスを使い，インテグレーションの自動化環境

を構築した．システム要件の概要は表2，図4のとおりである．

表2　インテグレーションの自動化環境のシステム要件

ハードウェア	Dell PowerEdge 1400
CPU	PentiumⅢ-800MHｚ(FSB133・FC-PGA)
OS	Microsoft Windows XP
Memory	ECC 133MHz SDRAM 512MByte
LAN	Intel EtherExpress+ Pro 100
HDD	Ultra3/LVD SCSI10,000回転 36GB
Java	JDK 1.3

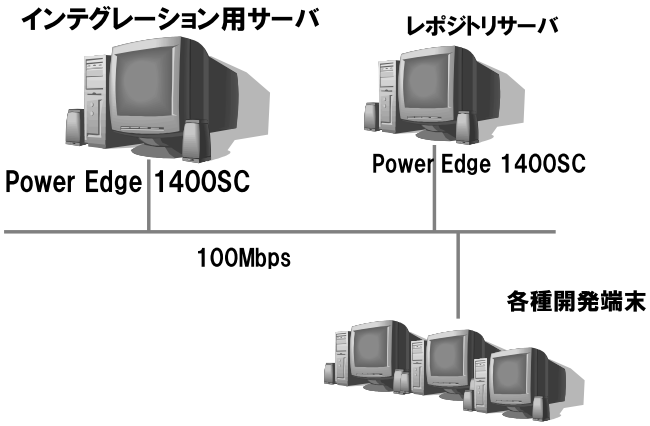


図4　システム構成図

自動化されたインテグレーション作業の流れは次のようになる（図5）．

1)　ソースコードをソースコードレポジトリ（Subversion）より取得する
2)　ソースコードをコンパイルし，実行可能なモジュールを生成する
3)　JUnitによりモジュールのユニットテストを実行する
4)　CheckStyle，FindBugsにより品質指標の取得，静的不具合の検出を行う

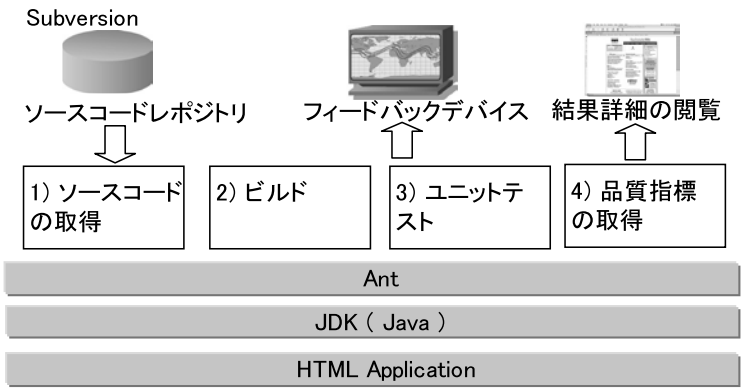


図5　PCディスプレイを使用した実際のフィードバックデバイス

1)から4)までの一連のタスクは3時間ごとに一回自動的に実行されるようにWindowsのタ

スクマネージャによりスケジューリングしている．タスクの状態はフィードバックデバイスで常時，確認できるようになっており，また品質指標など詳細な情報はWebブラウザから結果レポートを参照することが可能となっている．

インテグレーションの自動環境のモジュール構造は図6のようになっている．

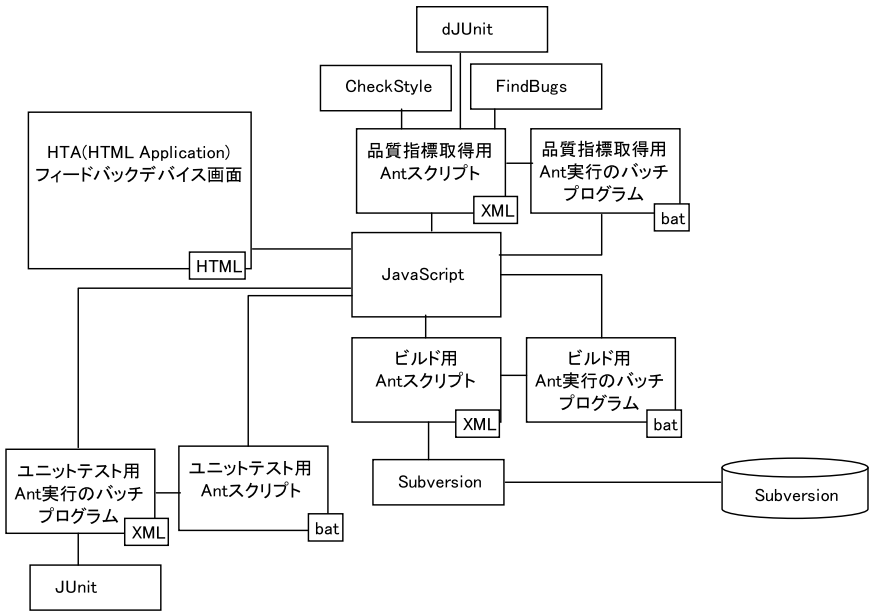


図6　モジュール構造図

3.2.1項のHTAで実装されたフィードバックデバイス画面よりJavaScriptがインテグレーション作業を行うタスクを実行する．タスクはAntのスクリプト（以降，Antスクリプトと記す）で実装されている．Antスクリプトより，前述の各オープンソースソフトウェアを実行している仕組みとなっている．

3.3.1　Antスクリプトの実装例

本項ではインテグレーションの自動環境を実現するAntスクリプトについて，ソースコードのビルドとサイクロマチック複雑度を算出するタスクの実装を例示する．

なお，例示するにあたり本論では，以降の本文のなかで，オープンソースソフトウェアのインストール先を表3のように記述する．

表3　インストール先

インテグレーション環境の導入場所	INTEG_HOME
Antのインストール先	ANT_HOME

Antからソースコードのレポジトリである Subversion に接続するために，「SvnAnt[*10]」というオープンソースのライブラリを使用する．このライブラリを使い[*11]，Subversionからソースコードをチェックアウト（ダウンロード）するAntスクリプトは以下のようになる．

```
<?xml version="1.0"?>
<project name="pdm" default="main" basedir=".">
  <property file="build.properties" />
  <path id="svnant.classpath">
    <fileset dir="${ANT_HOME}/lib/">
      <include name="*.jar" />
    </fileset>
  </path>
  <!-- SVNANT のタスク -->
  <typedef name="svn" classname="org.tigris.subversion.svnant.SvnTask"                    1)
classpathref="svnant.classpath"/>

  <target name="checkout" depends="clean" description="チェックアウトターゲット">
    <svn username="${svnant.repository.user}" password="${svnant.repository.passwd}">
      <checkout url="${svn.gwing2pdm.url}" destPath="${svn.dest}" />                      2)
    </svn>
  </target>
```

1） Subversion に接続するタスクを定義する．classpathref 属性にて前述のファイルを指定する．

2） Subversion からソースコードをチェックアウト（ダウンロード）する

こうしてチェックアウトされたソースコードは Ant スクリプトでコンパイルを行う．コンパイルを行う Ant スクリプトを以下に例示する．コンパイルには Ant で標準実装されている javac タスクを使う．

```
<target name="main" depends="buildcopy" description="メインターゲット">
  <javac srcdir="${src.dir}" destdir="${classes.dir}" source="1.4" fork="yes"
    memoryMaximumSize="512m" debug="true" optimize="true" encoding="UTF-8">
      <classpath refid="main.compilation.classpath" />
  </javac>
</target>
```

つぎに，3. 1 節で述べた CheckStyle を用いて，プログラムのサイクロマチック複雑度を算出する Ant スクリプトを例示する．まず，http://checkstyle.sourceforge.net/ より checkstyle-4.3.zip をダウンロードする．ダウンロードした zip ファイルを解凍し，checkstyle-all-4.3.jar を取り出す．そして，その jar ファイルをクラスパスに含めるようにする．本例では INTEG_HOME/lib というフォルダを作成の上で jar ファイルをコピーし，クラスパスに含まれるようにしている．

```
<?xml version="1.0"?>
<project name="pdm" default="main" basedir=".">
  <!-- checkstyle タスク                    -->                                        1)
  <taskdef resource="checkstyletask.properties" classpath="${INTEG_HOME}/lib/checkstyle-all-4.3.jar"/>
<!-- 循環性複雑度取得の実行 -->
  <target name="cc" depends="clean" description="循環性複雑度の取得">
    <checkstyle config="${cc.config}" failureProperty="checkstyle.failure" failOnViolation="true">   2)
      <classpath refid="main.compilation.classpath"/>
```

```
        <fileset dir="${src.dir}">
          <exclude name="com/XXXXX/jp/pdm/PMCY9000/AMCYDAB/**/*"/>
        </fileset>
        <!-- XML 形式のレポートを出力します -->
        <formatter type="xml" toFile="${cc.report}"/>
      </checkstyle>
      <style in="${cc.report}"  out="${cc.report.html}" style="${checstyle.style}" />
    </target>
```

3)

1)　CheckStyle 用のタスクを定義する．

2)　CheckStyle 用タスクを実行しサイクロマチック複雑度を算出する．求めたサイクロマチック複雑度の情報を XML ファイルに出力する．

3)　2)で生成された XML を HTML に変換する．

2)でサイクロマチック複雑度を算出する際に外部のコンフィグレーションファイルを読み込む．詳細は割愛するが，そのコンフィグレーションファイルを以下に例示する．

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Puppy Crawl//DTD Check Configuration 1.1//EN"
"http://www.puppycrawl.com/dtds/configuration_1_1.dtd">
<module name="Checker">
  <module name="TreeWalker">
    <module name="com.puppycrawl.tools.checkstyle.checks.metrics.CyclomaticComplexityCheck">
      <property name="max" value="10"/>
      <property name="severity" value="info"/>
    </module>
  </module>
</module>
```

算出したサイクロマチック複雑度は HTML に出力され，ブラウザで参照可能となっている．最後にその出力された HTML イメージを示す（図 7）．

図 7　サイクロマチック複雑度の HTM レポート

4. お　わ　り　に

インテグレーションの自動化環境は以下のメリットがある．

1)　ビルドの自動化により，コンパイルエラーのないソフトウェアが存在し，いつでもソフトウェアを動作させることができる．

2)　回帰テストが自動で行われるためデグレードの心配が低減した

3)　プログラム品質の指標が手に入り，対応策が立てやすい

4） フィードバックデバイスを通して，不具合の発生を明確に知ることができた

そして，インテグレーションを早く・何度も実施できることにより，インテグレーションに関わる問題に対してすばやく対応できるようになった．これがインテグレーションの自動化の最大のメリットであった．

インテグレーションの自動化環境を構築するにあたり，オープンソースソフトウェアを組み合わせたからこそ，こうしたメリットを享受できた．また，3章で例示したように少ないコード量で実装・構築できることも大きなメリットのひとつである．

現在，インテグレーションの自動化を行うオープンソースソフトウェア[*12]も登場し，日本ユニシスグループにおいてもインテグレーションツールの整備が開始されている．今後は一からインテグレーション環境を構築することは少なくなると予想される．提供・標準化されたインテグレーション環境をプロジェクトにフィットさせることが課題になる．したがってインテグレーションの自動環境を整備するエンジニア，とくにアーキテクトは，プロジェクトの状況，プロセスを確実に把握しておくことが重要になる．もちろん，オープンソースソフトウェアの情報収集，評価を随時行っておくことも必要である．そして，インテグレーションの自動化によって，さらなる効率・品質が向上できるように，多くのアイデア，仕組みを取り入れていくことが課題となる．

最後にシステム開発にてご助力いただいたプロジェクトメンバーの全員に感謝の意を表する．システム開発の後期に行われるインテグレーションの作業に対して，本稿の情報が参考になれば幸いである．

＊1 マーチン・ファウラー氏とマシュー・フォウメル氏による，記事『Continuous Integration』（http://www.objectclub.jp/community/XP-jp/xp_relate/cont-j）にて広く知られるようになった．
＊2 一般的に Java では，JDK に含まれる javac コマンドを使用し，コンパイルを行う．
＊3 http://subversion.tigris.org/
＊4 http://ant.apache.org/
＊5 http://www.junit.org/index.htm
＊6 http://checkstyle.sourceforge.net/
＊7 http://findbugs.sourceforge.net/
＊8 http://works.dgic.co.jp/djwiki/Viewpage.do?pid=@646A556E6974
＊9 トヨタ生産方式の一要素で，行灯が由来となっている．生産ラインで異常を即座に知らせるランプであり，ライン従事者は異常があった場合即座にアンドンを点灯させる．
＊10 http://subclipse.tigris.org/svnant.html
＊11 http://subclipse.tigris.org/svnant.htm から svnant-1.0.0.zip ファイルをダウンロードする．ZIP ファイルを解凍し，svnant.jar，svnClientAdapter.jar，svnjavahl.jar を ANT_HOME/lib フォルダにコピーする．
＊12 代表的な CI ツールとして「Apache Continuum」（http://maven.apache.org/continuum/），「CruiseControl」（http://cruisecontrol.sourceforge.net/）といったオープンソースソフトウェアがある．

参考文献 ［1］ デビッド・トーマス，アンドリュー・ハント，マイク・クラーク，「達人プログラマー ソフトウェアに不可欠な基礎知識」，株式会社アスキー，2000年11月 P.326～P.340
［2］ 角谷信太郎，「継続的インテグレーション対応ツール・カタログ」，月刊 JavaWorld，IDG ジャパン，2006年6月号，2006年6月，P98～100

執筆者紹介　山　田　暁　広（Akihiro Yamada）
1998年関西ソフト・エンジニアリング(株)入社．アパレル系顧客でメイン・フレームのSEサービスを経て，2001年からJavaによるWebシステム構築プロジェクトを中心に従事．現在，USOL関西(株)システムサービスプロジェクト第2グループ所属．