J2EE アプリケーション開発におけるビルド／テスト自動化への実践的アプローチ

北嵐 直樹‡

†日本アイ・ビー・エム（株） 〒105-0081 東京都中央区日本橋箱崎町 19-21
E-mail: †arashi@jp.ibm.com、

あらまし 開発フェーズにおける実践的なビルド／テストの自動化事例を紹介します。
キーワード テスト、自動化、OSS、ビルド

Empirical Approach of automated build and test in J2EE application development
－for developing high quality software－

Naoki Kitaarashi‡

†IBM Japan、 Ltd。 105-0088 Japan
E-mail: †arashi@jp.ibm.com

Abstract This paper shows empirical methods for automated build and test in customer application development。
Keyword Testing、 Automated、 OSS、 Build

1. はじめに

Kent Beck による XP の提唱以来テストに対する開発者の取り組み姿勢が変わってきている。テストはそれまでのコーディングの後に義務のように実施しなければならない作業から、開発者が積極的に、また幾分かは楽しみを持って取り組むべき作業になってきているのではないだろうか。特にプログラミング／UT における作業の効率化／品質向上へのアプローチは Java におけるオープンソース・ソフトウェア（以降 OSS と記述）の世界を中心に猛烈に広まってきている。

最近では、OSS は単なる商用製品の代理ではない。世界中のユーザーが手軽かつ無償で利用できるという特徴を活かして、OSS でしか実現できない様々なツールが生み出されてきている。JUnit や Ant はその代表とも言えるものだろう。開発時のビルド／テスト／デプロイの自動化も OSS 抜きには語れない分野であり、開発者は複数の OSS を組み合わせることで、品質の向上と作業の効率化を実現する自動化環境を容易に手に入れられるようになってきた。しかしながら、大規模なシステム開発の現場で、この OSS によるビルド／テスト／デプロイの自動化を実現する場合、選択可能な OSS の種類の多さ、実践的なガイドの不足、テスト・プロセスの未整備といった理由により、実現までのハードルは高い。

本論では、筆者が 1 年に渡る大規模お客様プロジェクトでの自動ビルド／テストの展開を通して得た、J2EE アプリケーション開発における実践的な自動化のアプローチについて述べていきたい。

2. 自動化の意義

まず始めにシステム開発における自動化の意義について確認しておきたい。自動化とは、単に自動化ツールを購入してテストを自動化することだけを指すわけではない。開発工程のうち何を自動化するのかを明確に定義することは、最近のオープン系の開発では極めて重要である。特に定型的な作業を適切に自動化することは、省力化／品質の安定化／属人性の排除／標準化の推進／プロセスの確立といった多様なメリットに繋がる。とくに、ビルド／デプロイ／メトリクスの測定は定型的な作業となるため、自動化の有力な候補として挙げられる。容易に自動化が可能な作業を手作業で行うことは、作業の非効率化だけでなく、作業の品質の面でも大きな問題を抱えていると言える。人間は創造的な作業には向いているが、定型的な作業には向いていない。逆にコンピューターは定型的な作業に向いていて、創造的な作業に向いていない。つまり、人手による作業とコンピューターを使った作業は補完関係にあると言える。

開発フェーズでの効果的な自動化タスクとして継続的インテグレーション[3]の実施が考えられる。継続

的インテグレーションとは、Martin Fowler により提唱されたアプローチであり、開発期間を通じて常に開発モジュールのインテグレーション、つまり、ソースコードのビルド作業を行うことを指す。この継続的ビルドの実行環境では、ビルドは単体テストが完了しモジュール統合に移る段階から始まるのではなく、開発作業期間を通じて常時ビルドが行われる。

Martin Fowler により提唱されたこのアプローチは、予防的テストの考えをベースにしている。ビルド作業では、全開発者のソースコードと依存するライブラリを組み合わせてコンパイル作業を行う。ソース間でインターフェースの不整合がある場合はコンパイルエラーが発生するため、ビルド作業を行うことはモジュール全体のインターフェースの検証を行うことになる。つまり、継続的ビルドを実施することで、インターフェース間の整合性に関する不具合を、ソースを修正した直後に発見できるようになる。不具合は、それが生み出されてから発見されるまでの時間が短ければ短いほど、少ないコストで修正することができる。従って、継続的ビルドを実施することで、モジュール間の整合性に関する不具合を最小限のコストで修正することができるようになると言える。

継続的にビルド作業を行うにはビルド作業の自動化が不可欠であり、現在では各種の OSS を組み合わせることで、リポジトリを監視し、コードに変更があった場合に自動的にビルドが走る環境を準備することができる。

3. 各種 OSS の紹介

Java アプリケーションの開発において、各種タスクの自動化の実現にはツールとしての OSS の存在が必要不可欠である。ここでは 4 章でのプロジェクト事例の紹介に先立って、利用した OSS について簡単に説明しておく。

■Ant

C における make と同様に、java 開発で各種タスクの自動化に使用されるツールが Ant である。Java 開発の現場では既にデファクト・スタンダードな位置付けであり、build.xml と呼ばれる XML で記述された定義ファイルを実行することで開発に関連するあらゆるタスクを自動化することができる。Ant ではコンパイル、ファイルのコピー／削除といった個々の作業がタスクとして定義されており、これらのタスクを組み合わせることで一連の作業を定義していく。あらかじめ多くのタスクが標準で提供されるだけでなく、Java プログラムとして簡単にユーザー定義のタスクを追加できるため、最近の OSS では Ant タスクを標準で提供しているものが多い。このため Ant ベースの自動化タスクに各種 OSS を容易に組み込むことができる。

■Maven

簡単に言えば Maven は Ant の代わりに利用可能なビルド／テスト／デプロイを自動化するツールである。Ant が一連の作業を 1 から XML として記述するのに比べて、Maven ではビルド／テスト／デプロイのスクリプトが事前に準備されており、開発者はソースコードの場所とオプションを指定するだけで、簡単に作業の自動化を図ることができる。Ant をプログラミング言語とすれば、Maven は ERP のようなパッケージソフトと考えると、両者の機能の違いが分かりやすいだろう。単に作業の自動化を行うたけでなく、最終的に Web のレポートサイトを構築するように構成されていることも Maven の大きな特徴である。Maven での各作業はプラグインという単位で構成されていて、各プラグインは HTML レポートを生成するように作成されている。Maven 本体は各プラグインが生成した様々なレポートを 1 つの Web サイトとしてまとめることができるため、面倒な設定をすることなく、JUnit によるテスト結果やカバレッジ測定結果等を、HTML レポートとして公開可能だ。

■CruiseControl

継続的インテグレーションの提唱者である Martin Fowler が所属する ThoughtWorks 社より、無償で提供されているビルドの自動化をサポートするツールである。CruiseControl はサーバー上に常駐するプログラムであり、指定されたリポジトリを常時監視して、ソースコードに変更があると自動的にビルド用のスクリプトを実行する。そして、ビルドの結果をチェックして、e メールによる通知を行ったり、ビルド結果を Web サイト上で公開したりする機能を提供する。CruiseControl 自体はビルド機能を提供せず、Ant または Maven を呼び出すことでビルド作業を実施する。CruiseControl を使用することで、Ant や Maven で作成したビルド／テスト用のスクリプトをサーバー上で自動実行し、その結果を容易に監視することができるようになる。

■CheckStyle

CheckStyle はルールに従って Java のソースコードに対する静的インスペクションを行うツールである。Javadoc と API の整合性が取れているか、変数名が標準に従っているか、未使用の変数／import が定義されていないか等、CheckStyle に標準で定義されている様々なルールを取捨選択して、ソースコードをインス

ペクションできる。各ルールにはオプションを指定できるため、しきい値、名前の書式、チェック対象といった項目を柔軟にカスタマイズ可能だ。CheckStyle は Eclipse のプラグインの他、Ant、Maven にも対応しているため、作業の自動化も容易である。

■Cobertura

Cobertura はオープンソースのカバレッジ・ツールとして有名な JCoverage の後継とも言うべきツールである。本家 JCoverage GPL 版のメンテナンスが止まってしまったことを受けて、JCoverage のソースコードを元に新たに開始されたのが Cobertura プロジェクトである。使い勝手や機能は JCoverage とほぼ同様であるため、JCoverage に慣れたユーザーなら違和感なく使用できるだろう。既知の JCoverage の不具合が修正されていること、McCabe のサイクロマチック複雑度を算出する機能が追加されていること等が、JCoverage との主な違いである。

4. 適用事例

この章では、筆者が参画する大規模 J2EE アプリケーション開発プロジェクトで稼働している自動ビルド／テスト環境について説明する。

ビルド・サーバーとしては 1 台の PC サーバーを用意している。プロジェクトの規模やサーバー上で実行させるタスクにも依るが、サーバー機にはさほどスペックは要求しない。本プロジェクトでも当初はノートブックをサーバーとして使用していたが、スペック的な問題は特になかった。

対象となるクラス数は 2005 年 8 月時点で約 1000 クラス、それら全てに対して JUnit によるテストケースの作成を義務づけている。開発対象のアプリケーションは基幹システム向けの基盤フレームワークであり、J2EE 1.3 、 Struts1.1, IBM WebSphere Application Server6.0, DB2 v8.1, MQ v5 といった技術や製品を使用する。JUnit 系テストフレームワークとしては、JUnit, StrutsTestCase for JUnit, DbUnit, Cactus を使用している。特徴としては基盤系であるため画面系の処理が少なく、業務系のアプリケーションと比べて JUnit よる単体テストに向いていることが挙げられる。

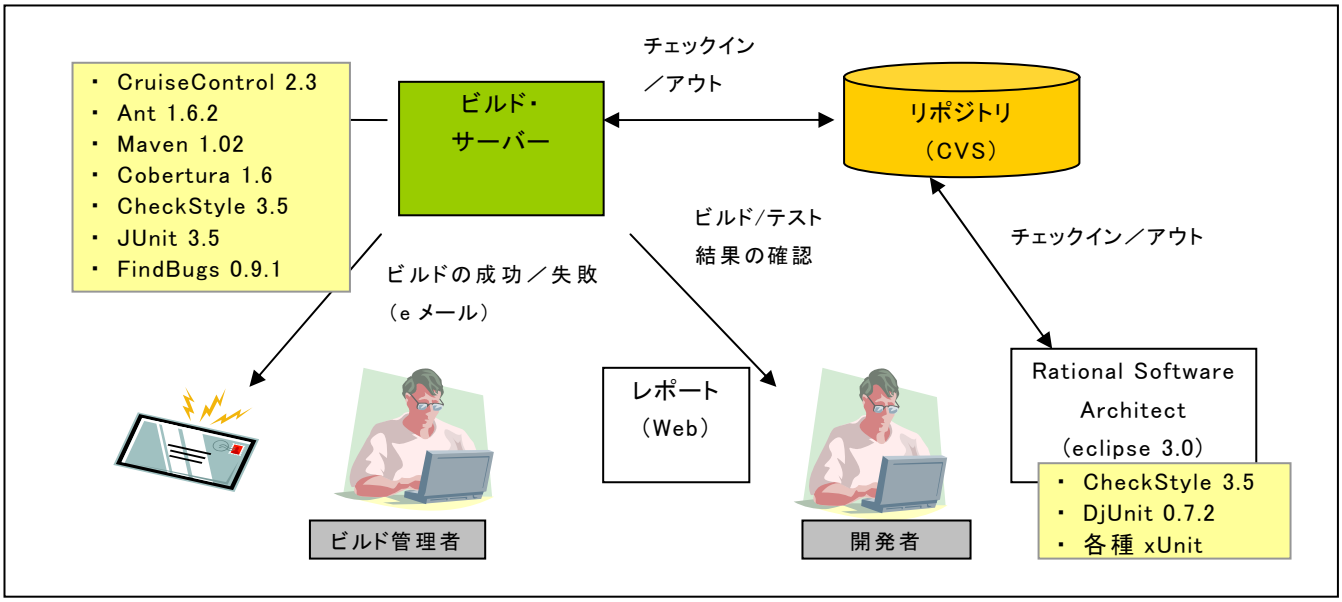
本プロジェクトでは、単体テストの完了基準として下記の 3 つの項目を定めている。

- カバレッジ基準の達成（DjUnit、Cobertura 使用）
- CheckStyle による静的コードインスペクション
- JUnit によるテストケースの作成とサーバーでの自動実行の成功

上記の基準を確実に守らせるために、開発環境であるクライアント側とサーバー側で同じチェックを実施することにした。具体的には、開発実施時は開発環境である IBM Rational Software Architect（以降、RSA）上でプラグインを実行することで、作成したテストケースが完了基準を満たしていることを確認する。一方、リポジトリにチェックインされたソースコードに対しては、ビルド・サーバー上で定期的に自動タスクを実行させ、全てのソースコードが単体テスト完了基準を満たしていることを確認することとした。

RSA 上では、静的コードインスペクションのために CheckStyle プラグインを、JUnit テストケース実施時のカバレッジ測定のために DjUnit プラグインを使用し

図 1 自動ビルド／テスト環境



た。CheckStyle に適用するルールは、デフォルトで提供されている Sun の標準ルールを修正し、必要最低限なものだけを定義している。

サーバー上では、CruiseControl を使用することで継続的なビルド／テスト環境を実現させている。CruiseControl はソースコード・リポジトリ（CVS）を監視し、コードに変更があった場合は Ant ビルドファイルまたは Maven プロジェクトを実行して、ビルド／テストを実行する。最近の OSS は eclipse のプラグインと Ant/Maven ベースのタスクの両方を提供しているものが多いため、このクライアント側とサーバー側で同じチェックを行うのには OSS の利用が適していると言えるだろう。

本プロジェクトでの自動実行環境は Maven と Ant を組み合わせたものである。当初、Maven のみで環境構築を試みたが、ソース／テストケース・リポジトリの数、プラグイン実行の失敗、デプロイ用に準備している Ant スクリプトとの共有等の理由により現在の Ant と Maven を併用する形に落ち着いた。Ant と Maven の使い分けの方針は次の通りである。

- コンパイル用に準備している Ant がある場合はそれを使用（二重メンテの回避）
- Maven のプラグインが正常に動作しない場合やサポートしていない場合、同等の Ant タスクで代替
- Maven のプラグインとオプション設定だけでは、機能的に満たされない場合、Ant タスクを使ってカスタマイズを行う

Maven のプラグインが正常に動作しないケースは、一部あり、対象クラス数が多い場合に OutOfMemory が発生するというものである。また、Maven のプラグインが提供するデフォルトのレポートをカスタマイズしたい場合や、Maven のプラグインが提供しないオプション設定を行いたい場合は、Maven のプラグインを独自にカスタマイズするのではなく、Ant タスクとして同様な機能（タスクの実行とレポートの作成）を作り込んだ。表 1 に Maven および Ant で実行しているタスクを示す。自動実行させているのはビルド作業の他、JUnit による単体テストの自動実行、Cobertura を使ったカバレッジの測定、CheckStyle、PMD、FindBugs による静的コードインスペクションの実施、Javadoc の生成、CVS 変更ログの作成を行っている。これらのタスクの実行結果は全て HTML レポート化し、Maven のレポートサイトとして、全開発者に公開している。Maven は日本語化されていないため、デフォルトでは全て英語のサイトが公開される。これでは一般のユーザーには使いにくいと判断したため、レポートのトップページだけは独自に日本語化を行った。各タスクの実行頻度は、jar ファイルのビルドのみリポジトリでの更新の度とし、JUnit による単体テストと各種チェックタスクの実施は 1 日 2 回としている。これは 1 回の実行にタスク全体で 1 時間半ほどかかるためである。

タスク名	Maven /Ant	実行間隔
リポジトリからのチェックアウト	Ant	更新の度
ビルド（コンパイル,jar 作成）	Ant	更新の度
カバレッジレポート	Ant	2 回/日
単体テスト（JUnit）	Ant	2 回/日
コードインスペクション（CheckStyle）	Maven	2 回/日
CVS 変更ログ一覧作成	Ant	2 回/日
PMD（コードインスペクション）	Maven	2 回/日
FindBugs（コードインスペクション）	Maven	2 回/日
Javadoc 生成	Ant	2 回/日
ソースコード公開(Xref)	Maven	2 回/日

表 1 実施したタスクの種類と頻度

自動実行の結果は成功／失敗にかかわらず CruiseControl の機能を使用して HTML 形式の e メールをビルド担当者に送るように設定している。自動実行は様々な原因で失敗する。一番多いのはモジュール変更によるコンパイルエラーだ。担当者がいちいちサーバーの実行結果をチェックしなければ実行に失敗したことが分からないと、対応が遅れがちである。自動実行時のエラーは迅速に解決すべきであり、これを実現するためには、PULL ではなく PUSH 形式で実行結果をチェックできるようにしておくべきである。

5. 効果

本章ではビルド／テストの自動化を実施したことによる具体的な効果について述べていく。

まず継続的ビルドを実施することにより、リリース間際にビルドエラーが発生することを防止できるようになったことが挙げられる。あるソースコードへの変更が他チームのソースにコンパイルエラーを引き起こす、つまりインターフェースの不整合を、CVS へのチェックイン後に即時に発見でき、それを修正できるため、チーム間の依存関係のよる問題を最小限に抑えることができた。

またトータルで 1000 を超えるクラスを開発しているため、各開発者の環境にモジュールをソースコード単位で配布するのは、煩雑かつ非効率である。継続的ビルドにより最新の jar ファイルを共有ライブラリにチェックイン可能であったため、各開発者はクラスではなく jar を自分の開発環境に取り込むことで開発作

業を実施できた。

JUnit による自動テストの実施は、変更によるデグレードの防止、大小様々なリファクタリングの促進に繋がっている。これは、リグレッション・テストを実施することで、ソースコードへの変更が既存の機能に影響を与えていないことを容易に検証可能であったためである。通常、開発が進み、実装した機能の依存関係が複雑になるにつれて、変更には細心の注意が必要となるものである。実際、ささいな変更が思いもよらぬ所で影響を及ぼしていることがあるものだ。加えて、プログラミングの品質を維持するためには、不具合の修正だけでなく設計の変更を伴うリファクタリングの実施も不可欠な作業と考える。今回の開発では、リグレッション・テストの自動化を実現できたからこそ、必要なリファクタリングを実施できたと言える。

クライアント側開発環境での各種テスト（UT、カバレッジ、コードインスペクション）の実施と同じテストをサーバーで実施することで、各テストの実施を徹底することができたことも特筆すべき効果の 1 つとして挙げられる。UT の場合、各開発者がきちんと単体テストを実施しているかを、どのように確認するかが課題の 1 つとして挙げられる。リーダーによる目検でのチェックだと開発規模が膨らんだ場合は目が行き届かず、開発者のモラルまかせになってしまうケースが多いものだ。今回の開発では単体テストの完了基準として定義した項目については、サーバー側での自動テストにパスすることを最終確認とした。これにより、管理者側で UT の抜け／漏れを確実にチェックすることができ、アプリケーションの品質の維持に役立った。

最後にツールによるカバレッジ測定の効果について触れておく。カバレッジ測定は昔から UT で必ず実施したいテストとして挙げられているものだが、ツールのサポートなしには実現が難しいテストの 1 つである。カバレッジ測定のレベルとしては、C0（行網羅）、C1（分岐網羅）、C2（パス網羅）が挙げられるが、ツールがサポートするのは通常 C0 もしくは部分的な C1 のみである。カバレッジが 100%であったとしても行網羅の場合は、全てのコードが実行されたことを示すに過ぎない。従って、例えツールを使ってカバレッジを測定してみても、そのソースが取り得る全ての条件の組合せをテストしたとは言えない。つまり、論理的なテストケースの網羅性までは、ツールでは測れないと言える。

筆者は、理想を言えば人手によるテストケースのレビューは必ず行うべき作業と考える。これは、上記に挙げた通り、テストケースの論理的な網羅性は機械では測れないからである。しかしながら、現実的な話として全てのテストケースを人手でテストするには膨大な労力が必要なことも事実である。そこで折衷案として、ツールによるカバレッジ測定を基本として、重要なクラスのみ人手によるレビューを実施する案が考えられる。ツールを使って行網羅レベルのカバレッジを測定することで、明らかなテスト漏れがないことを機械的に保証することが可能だ。残りは、ある程度フォーカスを絞ってテストケースが足りているかどうかをチェックすれば良く、少ないワークロードで必要な品質を確保することができるだろう。人手による確認では、エラーが発生する確率が高い箇所や考慮漏れが出やすい箇所についてテストケースが充足しているかをチェックする。具体的には、境界値のテスト、異常系のテスト、マルチスレッドでの排他制御のテストなどが挙げられる。

ツールによるカバレッジテストを自動化することで、テストケースの品質を少ないコストで検証できることは、テストの自動化における分かりやすい効果と言えるだろう。

6. まとめ

ビルド／テスト／デプロイ作業の自動化の実現は、アプリケーションの品質や作業効率の面で大きな効果を生み出すものだが、これを 1 から構築することは少なくないワークロードとスキルを要求される。利用できるツールが多いことは開発者の選択肢を広げるという意味では良いのだが、逆に組合せの検証作業を繁雑にしている。また実践レベルのガイドとなると、あまり見あたらないため、現状では開発者自身が独力で自動化環境を構築しなければならない。筆者も候補となった様々なツールを試し、試行錯誤を重ねながら現在の環境を作り上げた。長期のプロジェクトであればこそ、こうした作業に時間をかけることができたが、短期間のプロジェクトの場合そんな余裕はないだろう。

より多くのプロジェクトで自動化環境を実現するためには、もっと手間なく環境を構築できるようにする必要がある。そのためには、使用するツールの推奨セットを決め、環境構築のための適切なガイドを提供することが求められる。また、最初は小さく初めて、それを徐々に大きくしていくといったアプローチが効果的と考える。具体的には、最初はビルド作業の自動化だけを行い、それから JUnit によるテストの自動化、Maven レポートサイトの構築を実現していくのである。Maven を使ったレポートサイトの構築に比べれば、ビルドの自動化は比較的容易に実現可能である。

JUnit よる単体テストの自動化、CheckStyle による静的コードインスペクション等のテスト作業の自動化は管理者の努力だけでは効果的な運用はできない。開発

者を巻き込み、サーバーで発生したエラーは修正しなければならないという習慣を植え付けることが大事である。もし、サーバー上のテストに失敗しても、それをそのままで放置してしまう、つまりエラーが起こっていることを常態化させてしまうと、開発者はサーバー上でのエラーを無視してしまうようになる。この常態になってしまうとサーバー上でのチェックは形だけのものになってしまう。後でまとめてエラーを修正する案も考えられるが、ワークロードの確保が難しいため現実的ではないだろう。エラーがあっても修正しないのでは、せっかくサーバー上で自動テストを行っても、その効果が半減してしまう

自動ビルド／テストを効果的に運用していくには、エラーはすぐに修正するという習慣を開発チーム全体に持たせることが必要である。これを実現する手段として次の 3 つの作業が考えられる。

- サーバーでのテスト結果を全開発者に報知する
- ビルドエラーの結果を関係者（チームリーダー格のユーザー）に送る
- サーバー上でのビルド／テストに失敗した場合は、迅速に管理者から担当者に修正依頼を行う

始めのうちは、開発者は進んでサイトのチェックや修正を行わないはずなので、管理者側が権限を持ってエラーの修正を依頼する。サーバー側でエラーが起こった場合はすぐに修正を行うという体制を継続していくと、開発者は自発的にサーバーでのエラーを修正するようになっていく。こうなると管理者側での手間をかけずにビルド／テスト環境を効果的に運用していくことができるようになる。

自動化の中でもっとも実現が困難なのが、JUnit によるテストの自動化である。全てテストケースを JUnit で作成することは可能でも、ソースに対する変更の度にテストケースを更新し続けること、テストケースをサーバー上で自動実行させることには大きな労力が伴い、実現までの道のりは遠い。特に JUnit テストケースをサーバー上でも実行できるように開発するためには、ガイド／ユーティリティの準備と開発者に対する教育が欠かせない。開発者にまかせて作成したテストケースの場合、様々な理由によりサーバー上では自動実行できないクラスが数多く含まれるものであるからだ。

工数的／開発者のスキル的な理由で全てのクラスに対して単体テストの自動化が難しければ、クラスに優先順位を付けて重要なクラスのみを自動化対象にするといったアプローチも考えられる。本プロジェクトでも、JUnit による自動テストの実施率はまだ全体の 70%ほどであり、残りの 30%については今後組み込んでいく予定となっている。

今後の展望としては、現在のビルド／テストの自動化に加えて、サーバーへのデプロイ作業を自動化したいと考えている。本プロジェクトではアプリケーション・サーバーとして IBM WebSphere Application Server（以降 WAS と記述）を使用している。WAS ではデプロイの自動化のために各種 Ant タスクやコマンドが準備されているので、これらを組み合わせることで、現在、手作業で実施しているサーバーへのデプロイ作業のかなりの部分を自動化できると見込んでいる。

本論では、筆者が現在実践している自動ビルド／テスト環境の事例を紹介し、その効果と考慮点について述べた。本論が、日々プロジェクトでの品質管理に携わっている開発者の参考となれば幸いである。

文　献

[1] Rick D Craig、体系的ソフトウェアテスト入門、日経 BP 社、2004 年 10 月.

[2] Mike Clark 、 Pragmatic Project Automation 、 Pragmatic Bookshelf、 2005.2.

[3] 継続的インテグレーション、 http://www.martinfowler.com/articles/continuousIntegration.html