Webアプリケーション開発へのCI（継続的インテグレーション）の適用とその効果

第１ソリューション本部
ユビキタス・PCMソリューション部
第１ソリューション室
ITエンジニア
野村　晃
Hikaru　Nomura
hikaru-nomura@exa-corp.co.jp

第１ソリューション本部
ユビキタス・PCMソリューション部
第１ソリューション室
ITスペシャリスト
和田　宗靖
Muneyasu　Wada
muneyasu-wada@exa-corp.co.jp

第１ソリューション本部
ユビキタス・PCMソリューション部
第１ソリューション室
シニアプロジェクトスペシャリスト
七野　修一
Shuichi　Shichino
shuichi-shichino@exa-corp.co.jp

第１ソリューション本部
ユビキタス・PCMソリューション部
第１ソリューション室
ITエンジニア
椎原　実希
Miki　Shiihara
miki-shiihara@exa-corp.co.jp

第１ソリューション本部
ユビキタス・PCMソリューション部
第１ソリューション室
シニアアプリケーションスペシャリスト
中　宏水
Hiromi　Naka
hiromi-naka@exa-corp.co.jp

テクニカルコンピテンシー部
開発・品質技術室
アプリケーションスペシャリスト
武市　正人
Masato　Takechi
masato-takechi@exa-corp.co.jp

ウォーターフォール開発プロセスでは、開発中の欠陥は、埋め込まれてから発見に至るまでの期間が長いほど欠陥除去に多くの時間とコストを費やす。特に、上流工程で埋め込まれた欠陥の除去、仕様追加あるいは仕様変更は、品質劣化（デグレード）、スケジュール遅延、スコープクリープにつながる可能性がある。今回、筆者らはこれらの問題に対して、アジャイル開発および、そのプラクティスの1つであるCI（Continuous Integration：継続的インテグレーション）を導入することで対処した。本稿では、実プロジェクトにおけるCIの導入と、運用中に発生した課題とノウハウを紹介する。また、そのCI適用時の効果を品質・生産性という観点で考察する。

1. はじめに

ウォーターフォールに代表される従来型の開発手法では、品質保証において３つの大きな問題を抱えている。

１つ目の問題は、プロジェクトの最終工程まで、そのシステムが本当にお客様の要求を満たしているかどうかを確認する術がないことである。そのため、要件定義工程や設計工程での仕様漏れ、あるいはユーザと開発側の意識違いによる実装誤りがユーザテスト段階で発覚する。これらの仕様漏れや、実装誤りを修正することが、サービスを提供する上で必要と判断した場合に仕様変更となるが、この段階での仕様変更は、サービスイン後の大きな障害を引き起こす可能性が高い。また、サービスインに間に合わないという状況に陥る可能性もある。

２つ目の問題は、仕様や設計の不整合が開発終盤に発見されるリスクが高く、結合時に多大なコストがかかることである。通常の開発では、期間を短縮するため、モジュールを分割し、複数人の並行作業とすることが一般的である。モジュール実装者の責任の範囲は、モジュール単体での動作保証であるため、ソースコードは実装者間あるいは、実装者と設計者間での解釈のずれに起因する障害が内在した状態でリポジトリに保存される。これらの不具合はその後の結合テスト初期時に大量に検出されるのが一般的である。そのため、実装担当の人数、詳細設計書のあいまいさ、あるいはコーディング標準のあいまいさに比例し、結合テスト前に大量に不具合が検出されてスケジュールが遅延するリスクが高まる。

３つ目の問題は、実装者が担当外機能の仕様を把握しづらいということである。複数人で並行開発中の工程において、担当外機能の実装を完全に理解することは難しい。それにより潜在的な不具合が埋め込まれた場合、連鎖的に２つ目の問題を引き起こす要因となる。また、埋め込まれた不具合は、結合後に検出されることとなるが、埋め込んだ直後に検出された場合よりも、原因特定および修正負荷は増加する。

これらの３つの問題はスケジュールとリソースを逼迫させ、さらなる障害を埋め込む負の連鎖を招く。その結果、サービスイン後の重大障害、開発要員の疲弊やデスマーチを生み出すことになる。

１つ目の問題に対しては、スコープを細分化し、細分化した一部をインクリメンタルに開発し、ユーザが受け入れテストと要求のフィードバックをサイクリックに繰り返すアジャイル開発のアプローチがある[1]。このアプローチでは動作可能な製品をリリースし続ける必要があるため、CI（継続的インテグレーション）[2]の導入により作業の軽減を図ることができる。また２つ目、３つ目の問題に対してもCIの実施によりスケジュール遅延リスクの軽減、品質/生産性の向上が期待できる。

本稿では、これら３つの問題に着目し、Webアプリケーション開発へのCI導入事例を紹介する。またCI導入により得られた効果について考察する。

2. CI（継続的インテグレーション）とは

はじめに、Martin Fowlerによる「Continuous Integration」[3]を、CIの定義として記載する。

Martin Fowlerによると、CIとは、以下の４つの要素を満たし、テスト部分も含め、「全てが自動化された」「再現可能な」ビルドを、「日に何度も」実施する活動と定義している。

(1)構成管理の徹底

全てのソースコードが格納され、誰でも最新の（そして過去の）ソースを取り出すことが可能な、ただ１つの出入口を設定すること

(2)ビルドプロセスの自動化

ビルドプロセスを自動化して、誰でも、コマンド１つでソースからシステムを作り出すことができること

(3)テスト作業の自動化

テスト作業を自動化して、コマンド１つでいつでもテストスィートを実行できるようにすること

(4)リリースビルド

現行の実行可能モジュールで、もっとも適切であると自信を持てるものを、誰でも入手可能であるようにすること

また、CIの導入により、以下の恩恵を受けることができ

ると考えられる。

(1)問題（お客様との仕様ギャップの解消）の早期発見

プロセスを自動化することにより、開発初期から頻繁にモジュールの結合やシステムテストを実施し、いつでも動作可能なモジュールを取り出すことができる。その結果、プロジェクト終盤にならないと検出しにくく、さらに、お客様が操作することでようやくわかる仕様上の問題を早期に発見できる。

(2)モジュール結合時初期に発生する問題の減少

継続的インテグレーションを維持するためには、その日の内に、ビルド破壊、テスト未実施、テストNGとなるコードを検出して除去する必要がある。このプロセスを継続することで、少なくとも動作可能（テスト可能）なモジュールをいつでもリリースできる状態になる。その結果、結合着手時によく見られる、以下の２つの大きな問題が発生する可能性を低減する。

①ビルドエラーのためテストモジュールが作成できない
②モジュール単体では動作するが、結合すると動作しない

(3)デグレードリスクの低減

複数のテストを一括して自動実行することにより、ソフトウェアに加えられる変更が、他の部分に与える影響を継続的に検証できる。その結果、影響範囲の確認を見誤り、「問題の修正が他の問題を引き起こす」という負の連鎖を回避できる。

CIの４つの要素は、個々に見ると一般的なプロジェクトにおいて既に実施しているものである。ただし、これら要素を個々に実施するのではなく、統合的に管理することで製品の製造工程を円滑に運営することが可能となる。本章以降、この統合管理システムをCIシステムと呼び、このシステムの実装と効果について述べる。

3. CIシステムの実装

本章では、CIにおける４つの要素を満たしたCIシステムの実プロジェクトへの適用事例と構築方法を述べる。

3.1. プロジェクトの概要

適用したプロジェクトは、アジャイル開発によるWebアプリケーションの開発事例である。設計、開発、テスト、製品リリーステストまでを１回のイテレーションとした。１イテレーションの期間は２～３週間とした。

開発の際、お客様から開発チームに提示される要求仕様は、ユーザ操作の手順を記述しており、開発側の製品リリーステストと、お客様の受け入れテストのテスト仕様を兼ねている。お客様は、リリースされた製品の利用結果を、開発チームにフィードバックし、次のイテレーションへの要求に反映した。

図１にイテレーション毎の開発サイクルの流れを示す。

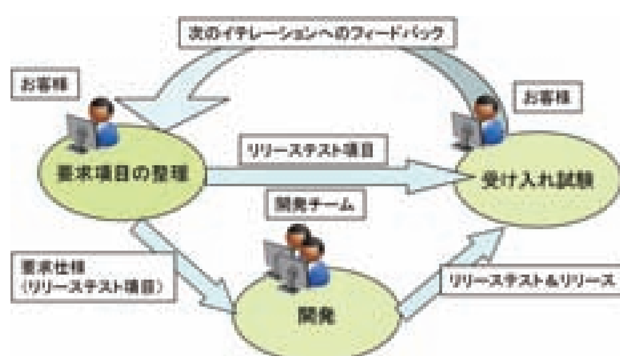

図１　イテレーション毎の開発サイクルの流れ

3.1.1. CIの要素と適用ツール

プロジェクトにCIを適用する上では、CIの４つの要素を満たし、かつ開発チームがフィードバックを受けるための仕組みが必要である。本プロジェクトでは、CIの各要件それぞれについては、各種のツールを導入し、作業の自動化を図った。さらに、各種ツールにより自動化された個々の作業を統合的に管理し、その結果をレポートとしてフィードバックするまでの一連の流れを自動化するために、継続的インテグレーションツールと呼ばれるツールを導入した。

表１に、適用したツールの一覧を示す。

表１　CIの要素と適用ツール

CI の要素	適用ツール	概要
構成管理の徹底	バージョン管理システム	ソースコードやその他のデータを管理する
ビルドプロセスの自動化	統合ビルドツール	複数のソースコードやライブラリを制御、統合してコンパイルなどを実行し、実行ファイルを生成する
テスト作業の自動化	テスティングフレームワーク	ユニットテスト(単体テスト)自動化のためのフレームワーク
リリースビルド	リリーステストツール	Web ブラウザを使って Web アプリケーションをテストするツール
CI の統合管理	継続的インテグレーションツール	バージョン管理システムをサポートし、統合ビルドツールで管理されているタスクを実行する。タスクの実行結果をレポートする機能を持つ

3.1.2. CIシステムの概要

CIの４つの要素に適用した各種ツールを継続的インテグレーションツールで統合管理することで、一連のプロセスを自動実行するCIシステムを構築した。図２にCIシステムの構成を示す。

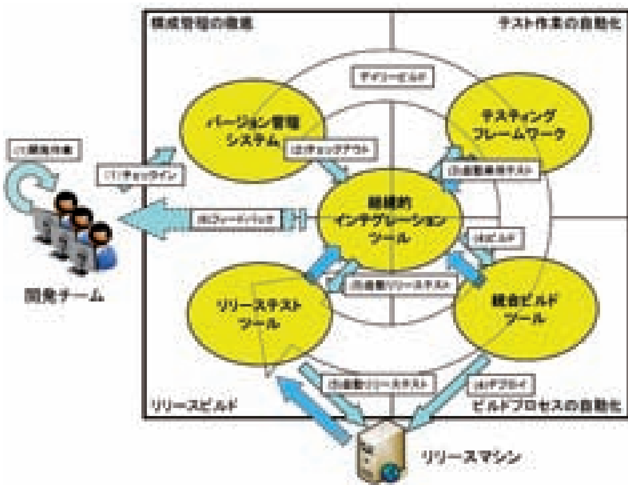

図２　CIシステム

一般的なCIシステムでは以下のツールを組み合わせた構成が適用されることが多い。

(1)バージョン管理システム
(2)統合ビルドツール
(3)テスティングフレームワーク
(4)継続的インテグレーションツール

今回適用したCIシステムでは、上記ツールにリリーステストツールを統合することで、リリーステストの自動化をCIのプロセスに含めたことが特徴となる。

CIのプロセスにリリーステストを含めることで個々の開発者による構成管理上のエラーと実装上のバグのほか、要求仕様に対する不適合も検出できるなどのメリットが生まれる。一方で、開発の初期段階ではリリーステストを適用する一連の機能が揃っていないため、デイリービルドが動作しないことが懸念された。しかしながら、今回のプロジェクトでは、要求仕様がリリーステスト項目として開発期間前に提供されていたため、図３のように開発の初期段階ではモックをあてはめ、未実装部分があっても、CIが可能な状態を維持した。できるだけモックを単一な仕組みで実現するためDI(Dependency Injection)を利用し、テストしたい箇所にエクセルからデータを流し込む仕組みを自作した。

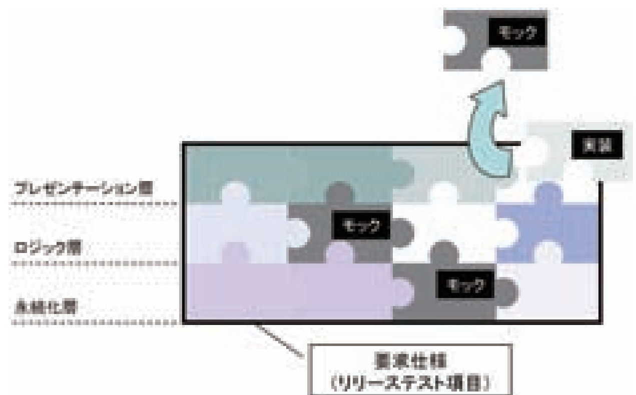

図３　モックと実装のイメージ

これにより、単体テストが完了したモジュールを段階的にモックと入れ替えていくことにより、CIのプロセスを中断せずに開発を進行させることができた。

3.1.3. CIシステムにおける製品製造プロセス

以下に、図２のCIシステムを使用した製品製造プロセス（デイリービルド）の流れを説明する。

(1)開発作業/チェックイン

開発チームは、日々の開発作業において以下の作業を実施する。

① 個々の開発環境でコーディング（機能の実装）

② 機能を実装したソースコードの単体テスト

開発者は一連の作業を完了した際に、成果物であるソースコードをバージョン管理システムにチェックインする。

(2)チェックアウト

継続的インテグレーションツールはデイリービルドの開始時に最新のソースコードをバージョン管理システムからチェックアウトする。

(3)自動単体テスト

継続的インテグレーションツールは、チェックアウトした最新のソースコードを、同じく最新の単体テストコードを用いて単体テストを全て実行し、結果をレポートする。

(4)ビルド/デプロイ

継続的インテグレーションツールは、全ての自動テストを通過した最新のソースコードからリリース可能なモジュールをビルドする。また、統合ビルドツールは、ビルドしたリリースモジュールを自動的にリリースマシンに対してデプロイする。

ビルドおよびデプロイの作業により、リリースマシン上のモジュールを常に最新の状態とし、また常にリリースできる状態に保つ。

(5)自動リリーステスト

継続的インテグレーションツールは、リリーステストツールを利用して全てのリリーステストを実行する。自動リリーステストにより、リリースマシン上のモジュールが全ての要求仕様を満たしているかを検証できる。

(6)フィードバック

継続的インテグレーションツールは、デイリービルドの結果をレポートとして開発チームにフィードバックする。

開発チームは、１日のサイクルの中で、開発作業に対するフィードバックを得ることができ、コンパイルエラー、単体テストで検出されたバグ、要求仕様を満たしていないバグなどを早期に発見し、ソースコードを小さな単位で修正できる。

開発期間中は、開発作業に対するフィードバックを受け、早期に修正を施すことを目的として、１日２回、以下の時間帯にデイリービルドを実施した。

① 13:00～（午前中の作業終了後）

② 18:00～（１日の作業終了後）

3.1.4. CI適用時のコード実装者の開発サイクル

図４に一連の開発サイクルの流れを示す。

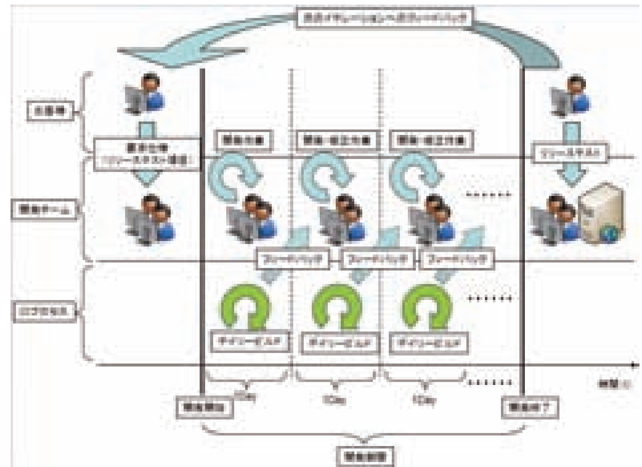

図４　開発サイクルの流れ

(1)日々の開発サイクルでは、リリーステストを含めたデイリービルドを実施する

(2)デイリービルドの結果は、電子メールで成功、あるいは特定のモジュールのコンパイルが失敗、テストが失敗などの記録（ビルドレポート）として開発チームにフィードバックされる。その結果、エラーが検出された場合には即座に開発チームによる修正が施される

(3)「開発作業　－　デイリービルド　－　フィードバック　－　修正作業」のサイクルを開発期間の終了まで継続することにより、段階的かつ細かい単位で機能の実装と修正が繰り返され、品質を保証でき、また、リリースできる状態を保ったまま開発を終了できる
(4)お客様へリリースされたシステムは、日々のリリーステストと同じ項目で受け入れテストが実施される
(5)お客様はリリースされたシステムから新たな要求や修正を検討し、次のイテレーションに向けて開発チームへのフィードバックを作成する

上記のサイクルを繰り返し、当初の機能を全て実装した段階で１つのイテレーションが完結する。

このように、自動化されたCIプロセスをデイリービルドとして日々の開発サイクルの中で繰り返して実行するアプローチをとることで、早期にバグを検出し、リリースできる状態を保った。

3.2. CIシステムソフトウェア構成

本プロジェクトでは、表２で示すソフトウェアを利用して、CIシステムを構築した。採用したソフトウェアは、いずれもオープンソースのソフトウェアである。Subversion®*1およびMaven2™*2は、社内での実績が豊富な点、Selenium*3は、ブラウザに依存しない点が採用した主な理由である。また、継続的インテグレーションツールには、CruiseControl*4と同等の機能を持つものにHudson*5やContinuum™*6があったが、本プロジェクトでは、お客様からの指定によりCruiseControlを採用した。

図５にCIシステムのソフトウェア構成図を示す。

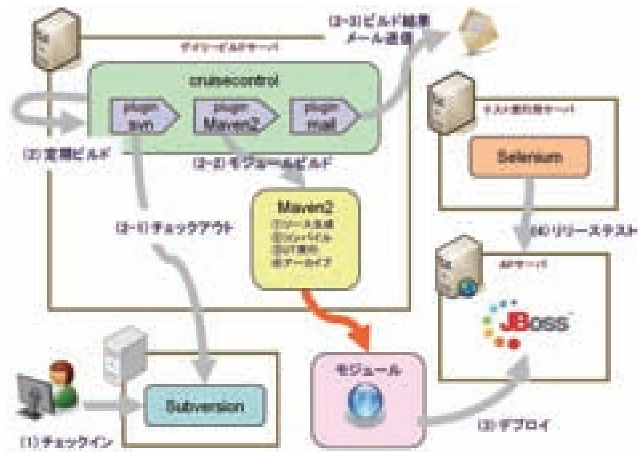

図５　CIシステム ソフトウェア構成図

3.3. リリーステストの自動化

製品製造後のリリーステストは動作可能なモジュールを本番の擬似環境上に配備し、Webブラウザを使って手動で実施することが一般的である。しかし、今回のように製品リリースの回数が多い場合、手動によるリリーステストは時間的にもリソース的にも困難である。そこで、本プロジェクトではSeleniumを用い、手動の操作手順（テストケース）を記録し、Webブラウザ上で自動実行するようにした。

表２　CIシステム ソフトウェア構成

CIツール名	ソフトウェア名	概要
バージョン管理システム	Subversion	ソースコードなどのバージョン管理システム
統合ビルドツール	Maven2	ビルドプロセスを再利用するためのフレームワーク。テストプロセスもビルドプロセスの一貫として提供される Ant™と比較し、主要な相違点は、以下である。 ・基本から外れている部分のみ設定すればよい(Convention over Configuration) ・依存関係の自動解決 ・各種plug-inを使用することにより、プロジェクト独自の処理を簡単に実現できる ・Antを再利用して呼び出すことも可能
テスティングフレームワーク		
リリーステストツール	Selenium	Webブラウザを使用し、Webアプリケーションをテストするツールである
継続的インテグレーションツール	CruiseControl	継続的インテグレーションを実現するための基盤となるフレームワーク。コードリポジトリからのチェックアウト、ビルド、メールによるビルド結果通知などを定期的に実行することが可能。Subversion,Maven2をはじめ、Git、Rakeなど各種構成管理・ビルドツールと連携するための豊富なplug-inが提供されている

3.3.1. Seleniumによるテストの自動化

SeleniumはWebブラウザを使用したWebアプリケーションのテストをサポートするオープンソースのツールである。これを利用すればWebブラウザの操作をテストケースとして記録し、テストの段階で操作を自動実行できるようになる。Seleniumは以下の３つのプロダクトから構成されている。

(1)Selenium Core

Seleniumの中核プロダクト。Webブラウザ操作や検証コマンドなどの機能を提供する。機能はJavaScript™で実装されている。

(2)Selenium IDE

ブラウザ操作を記憶してSeleniumのテストケースを生成するプロダクト。Firefox®プラグインとして提供されている。Selenium IDEを利用すれば、テストシナリオに合わせてFirefox上で画面操作するだけでテストケース作成できる。ただしSeleniumテストケースはWebブラウザの実装に依存性があるため、マルチブラウザ対応とするには、各ブラウザでの動作テストが必要となる。

(3)Selenium Remote Control

リモートのサーバに対してテスト実行を可能にしたプロダクト。Jetty[*7]という軽量なJava™のWebサーバを利用したSelenium Serverを内包しており、Selenium Server上でSelenium Coreを動作させてテストケースを実行する。テストケースの記述はSelenium Coreが対応しているHTMLだけでなく、Java、C#、Perl、PHP、Python、Rubyでも可能である。

これらのプロダクトを利用して自動化したリリーステストの全体フローを図６に示す。

以下、図６に沿って自動化の全体フローを解説する。

(1)Seleniumテストケース作成

作成済みのテストシナリオを基にSelenium IDEによりテストケースを作成する。

(2)デプロイ

Subversionから最新のテストケースと実行モジュールを取得し、テスト実行用マシンにはSeleniumテストケースを配置して、APサーバにはモジュールをデプロイする。

(3)テスト実行

テストを自動的に実行するため、JUnitテストケースにSeleniumServer起動、Seleniumテストケース実行といった操作を記述する。JUnitを使う理由は、Seleniumテストケースの実行結果をJUnitと同様の形式でも出力し、実行結果を容易に把握するためである。作成したテストケースを実行してサーバが起動すると、指定したブラウザが自動的に立ち上がり、Seleniumテストケースに記述した操作が実行される。全てのテストケースが実行されるとHTML形式のテストの通過あるいは未通過を記録してレポートが出力され、テスト完了となる。

このように、リリーステストを自動化することにより、作成したモジュールが要求仕様を満たしているかどうかの確認を人に依存しない形で、かつ繰り返し実施するようにした。

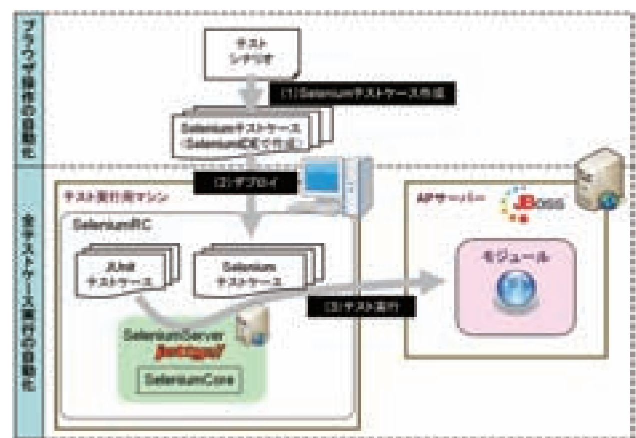

図６　リリーステスト自動化の全体フロー

3.3.2. Seleniumの応用

本節では、作成したSeleniumテストケースを２つの非機能テスト、長期安定化テストとペネトレーションテストに応用した事例を述べる。

(1)長期安定化テストへの応用

Seleniumテストケースの自動実行を実現したことにより、長期安定化テストへの応用が可能となった。

長期安定化テストとはシステムが長時間にわたって使用され続けた場合、メモリリークなどの不具合が発生し

ないことを確かめるテストである。

今回のプロジェクトではSeleniumテストケースの自動実行をループさせ、JConsole（Java仮想マシンのパフォーマンス・リソース消費を監視するツール）でテスト環境のメモリを監視するテストを実施した。テストはプロジェクトメンバーのいない週末をはさむ形で実施され、24時間休まずにリリーステスト用の操作を繰り返した結果、問題が発生しないことを確認できた。

(2)ペネトレーションテストへの応用

ペネトレーションテストとは動作しているシステムに攻撃をしかけてセキュリティ上の弱点を発見するテストであり、ペネトレーションテストツールとしてratproxy[*8]やParos[*9]などが有名である。これらのツールを使用する場合にもSeleniumは利用できる。これらペネトレーションテストツールはプロキシサーバとして動作し、ユーザがWebブラウザの操作を一旦記憶後、ペネトレーション用のHTTPリクエストを自動生成し、システムをチェックする。

① テスト対象システムが生成するURLを記憶し、リクエストを送信してチェック
② システムをブラウザから操作した際の通信内容を解析してチェック

このように、Webブラウザを手動で操作する箇所を、Seleniumで代替することで、低コストで網羅性が高いテストが実施可能となる（図７）。今回のプロジェクトではリリーステスト用Seleniumテストケースをratproxy、Parosのチェックに応用し、Webアプリケーションとしてセキュリティ上問題がないことを確認した。

図７　Seleniumを利用したペネトレーションテストツールの実行

4. CIの導入効果とベタープラクティス

本章では、これまでに述べたCIシステムをプロジェクトに適用して得られた効果と、CIシステムをより効率的に運用するためのベタープラクティスについて述べる。

4.1. CIシステムの導入効果

本節ではCIシステム導入効果を品質および生産性という観点で計測し、その導入効果を考察する。

比較対象とする従来型開発プロジェクト（CI未導入ウォーターフォール型プロジェクト）として、新規Webアプリケーション開発で開発規模および最終品質水準が類似したものを選定した。またCIシステムは導入していないもののCIの４つの要素を個々に実施しているものを選定した。

4.1.1. 品質に関する考察

CI未導入ウォーターフォール型プロジェクトとの障害密度実績値の比較を図８に示す。値は障害密度[障害件数/KLOC]を算出しCIプロジェクトを１とした相対値で提示する。値が小さいほど品質が良いことを意味する。

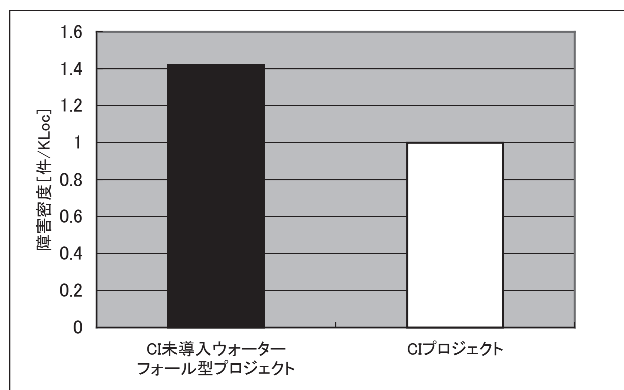

図８　結合テストにおける障害密度の比較

図８に示すように障害密度はCIプロジェクトの方が小さい結果となった。CIプロジェクトでは、サービスイン後も重篤な障害は発生していない。このことから、ウォーターフォール型プロジェクトと比較し、品質が良いと言える。

次に、原因別の障害分析結果を図９に示す。

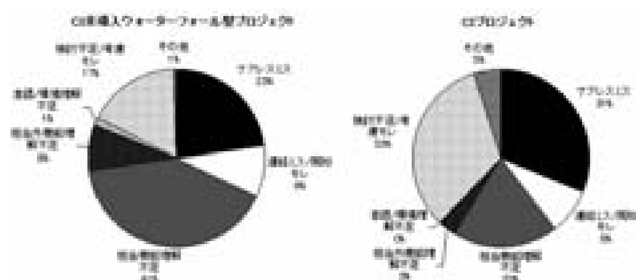

図９　原因別の不具合の比較

図９に示す通り、担当機能理解不足、担当外機能理解不足による障害が減少したことがわかる。これは、構成情報が常に正しい、すなわち製品製造可能な状態が実装フェーズで既に構成管理されており、開発者は、担当外の機能についてもどの様な動作を期待するか把握することが容易なためである。このことは、CIにおける大きな利点であり、担当者は、実装後、自分の加えた変更が期待した範囲にのみ影響を与えているかを確認することも可能となり、担当機能理解不足、担当外機能理解不足によるデグレーションを避けることができたためである。一方、ウォーターフォール開発では、製品を製造可能な構成管理の状態が、結合テスト工程の後半以降に達成されることから理解不足による障害が顕著に減少したと考えられる。

4.1.2. 生産性に関する考察

次に、生産性について評価する。CI未導入ウォーターフォール型プロジェクトとの開発生産性の実績値比較を図10に示す。生産性は[KLOC/人月]を算出しCIプロジェクトを１とした相対値とした。数値が大きいほど生産性が高いことを意味する。

図10では、全フェーズの生産性はほぼ両プロジェクトとも同等であった。一般的にアジャイル開発ではウォーターフォールに比べ生産性が上がると言われているが、今回のプロジェクトでは顕著には現れなかった。これは、後述する評価基準に対する課題のためと考えられる。

次にCI適用フェーズの生産性を抜き出して評価する。図11に示すように、CI未導入ウォーターフォール型プロジェクトの方が良好であるという結果を得た。この結果は、今回のプロジェクトでは改善すべきプロセスが含まれていることを示唆している。そこで、CI未導入ウォーターフォール型プロジェクトで実施していない２つの作業を抽出し、

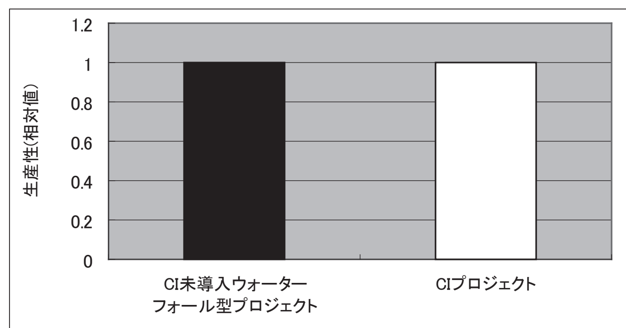

図10　開発生産性の比較

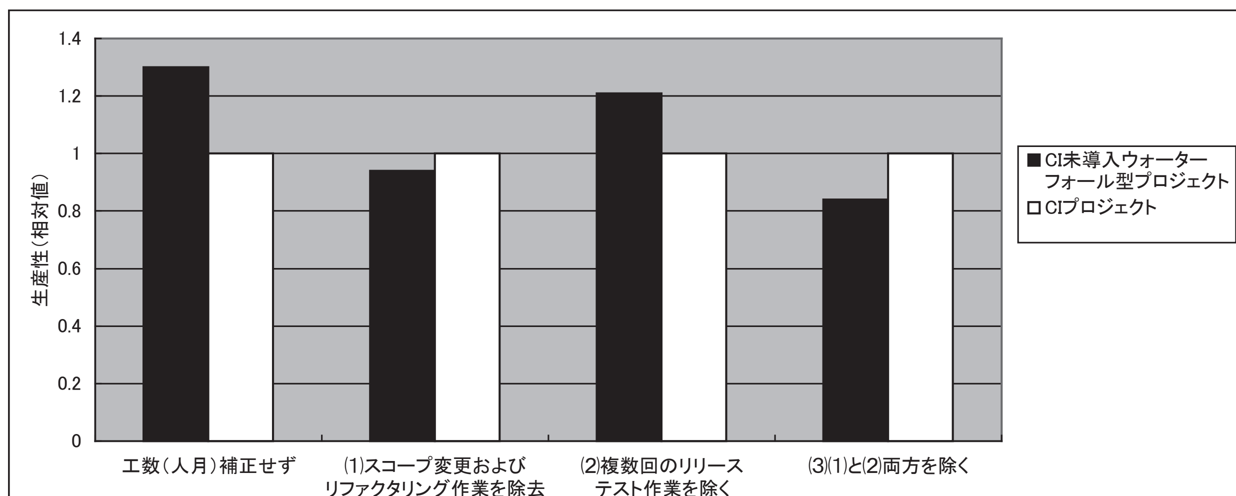

図11　CI適用フェーズの比較（アジャイル工数の補正）

CIの生産性について再評価したものについて３パターンの補正を行った。その結果、（1）のパターン（スコープ変更やリファクタリング作業を除去した）および、（1）のパターンを含む（3）のパターンでは、逆に生産性が上がる結果となった。このことは、CIシステム導入により複数回のリリーステスト作業が効率よく実施できているが、スコープ変更に耐えるようリファクタリングしたことが生産性を下げる要因となっていることを意味する。スコープ変更とリファクタリング作業はアジャイル開発では必須の作業のため、CIのフェーズに限定して生産性を上げるためには、複数回のリリーステストをさらに効率よく実施する必要がある。

4.1.3. 定性的な効果（スケジュール遅延リスクの低減）

次に、定性的な効果について考察する。CI未導入ウォーターフォール型プロジェクトでは、結合テスト初期にビルドできないという状況がしばしば発生していた。そのため、テストの開始が遅れ、テスト期間を圧迫することがあった。また、結合テスト中においても、障害改修したモジュールを結合し、テスト用モジュールを作成する作業と、テストモジュールのリリーステスト作業が、人手により行われていたことが、リリース担当者の負担を増大させた。

これらの問題は、CI導入で減らすことができ、短いイテレーション期間内にスケジュールどおり結合テストができた。また、リリース担当者の作業負担も大きく軽減された。

4.1.4. メトリクスの考察

次に、メトリクスについて考察する。今回は評価基準として、生産性をKLOCで評価した。一般的には、変更時の設計・製造・テストの生産性は、新規開発に比べて大幅に低くなる傾向にあるため、スコープ変更とリファクタリング作業を繰り返すアジャイル開発にとって開発期間を通したKLOCは不利な指標値である。

こうしたことから、今後、アジャイル開発の生産性を適切に評価するためには、イテレーション毎のKLOCの集計と、コードの変更理由のタグ付けが必要と考える。

また、今回対象としたCI未導入ウォーターフォール型プロジェクトとCIプロジェクトでは開発時期に違いがあり、CIプロジェクトで採用したフレームワークは、CI未導入ウォーターフォール型プロジェクトで採用しているフレームワークと比較すると、機能も洗練されて充実している。こうしたフレームワークの進化や設計技術、実装技術の向上により、少ないコード量で多くの機能を実現できる。KLOCでの生産性評価は、こうした条件の違いにより、表面的に生産性が低く見えてしまう危険性がある。これらを回避するためには、ファンクションポイントや、ユースケースポイントによる評価が有効と考える。

以上より、CIは、はじめに挙げた３つの問題、「アジャイル開発におけるリリース作業負荷」、「結合時のスケジュール遅延リスク」、「機能理解不足による品質低下および不具合対応負荷」を解消し、生産性の向上と品質の改善、および結合時の遅延リスクの解消に有効なアプローチであることを確認した。

4.2. CIのベタープラクティス

前節に示した通り、CIを適用した結果、多くの恩恵を受けたが、今回のプロジェクトは初めてのCI導入プロジェクトであり、運用にあたっては問題も発生した。それらの問題を解決するための対策をCIのベタープラクティスとして示す。

(1) ビルドが成功している状態を保つために開発者の意識を改革する

CIの導入初期には、ビルド失敗が頻発することがある。今回のプロジェクトでも導入初期には、この問題が発生した。この問題は、不完全なソースコードを不用意にチェックインし、エラーメッセージが頻発することにより、誰もそのメッセージに注意を向けなくなるという、CIに対する開発者の意識の問題が根底にある。しかし、今回のプロジェクトでは、毎日のミーティングや、イテレーション終了時のレトロスペクティブなどで継続的に注意を促すことで、徐々にこの問題を改善できた。

(2) テストデータをテストコードから分離する

アジャイルのように段階的に仕様が確定していく開発方法を採用する場合、仕様確定に追随して常に成功するテストケースを維持し続けることは、なかなか難しい。今回のプロジェクトでは、任意のテストデータをエクセルに定義することにより、データベースにデータをロー

ドしてから単体テストを実施するテストフレームワークを用意した。このフレームワークを使用することにより、テストデータのメンテナンスが容易になった。

しかし、エクセルデータをテストケース毎に用意していたため、データベースのカラム追加などテーブル定義の変更により、大量にエクセルデータを変更する必要が生じた。開発が進むにつれてエクセルデータ数も増え、それに対するメンテナンスコストも増加した。そこで、必要最低限のエクセルデータのみをテスト毎に用意し、それ以外は共通のマスタデータを使用するようにテストフレームワークを修正した。これにより、エクセルデータのメンテナンスコストを削減させることができた。

(3)機能限定的なテストシナリオを作成する

今回のプロジェクトでは、リリーステストとしてSeleniumを採用した。その結果、リリース直前でも大量のテストケースを繰り返し実行可能となり、テストの自動化により、テスト実行の省力化が可能となった。

その一方で、アジャイル開発では、イテレーション毎にお客様フィードバックがあり、それには次期開発部分のスコープの変更などが含まれている。スコープの変更は、仕様追加や仕様変更などを含み、作成済みの画面にも影響を及ぼした。　SeleniumはHTMLのコードをテストするツールであるため、画面の修正はそのままテストケースの修正となる。

今回のプロジェクトでは、リリーステストシナリオが機能横断的に作られているため、１つの画面への変更要求に対し、複数のテストケースの変更が発生し、それがメンテナンスコストの大幅な増加へとつながった。メンテナンスコストを減少させるためには、機能横断的なテストケースではなく、１画面につき１テストケースといったような、機能毎に分割されたテストケースを作成すると効果的である。ただし、データの更新機能とそのデータの参照機能といったように機能間に関連性がある場合は、必要に応じて機能横断的なテストシナリオを用意しておくと品質の向上に役立つと考えられる。

5. おわりに

システム開発中の品質保証とコストの低減を両立する手法としてCIを紹介し、アジャイル開発プロセスを適用した実プロジェクトでの事例を通して、CIが、１章で述べた従来型開発プロセスでの問題点を改善するために有効なアプローチであることを確認した。さらに本稿で提案したCIを実践することにより以下の様な効果が得られた。

(1)品質の向上
(2)製作時および結合テスト時の生産性向上
(3)スケジュール遅延リスクの低減

また、プロジェクトにより得られたノウハウや課題から、CIを効果的に実践するためには以下が重要であることがわかった。

(1)ビルドが成功している状態を保つために開発者の意識を改革する
(2)テストデータをテストコードから分離する
(3)機能限定的なテストシナリオを作成する

CIの考え方およびCIシステムは、ウォーターフォールの製造／単体テストフェーズや結合テストフェーズに適用しても問題のないプラクティスである。今後、我々は様々な開発に適用し、成果をフィードバックしていきたい。

参考文献

1) Venkat Subramaniam, Andy Hunt : 「アジャイルプラクティス」, オーム社,2007
2) Paul M Duvall, Steve M matyas, Andrew Glover : 「継続的インテグレーション入門」,2009
3) Martin Fowler:「Continuous Integration」, http://www.martinfowler.com/articles/continuousIntegration.html,2006

※使用ツールの参考URL

*1 Subversion
http://subversion.tigris.org/

*2 Maven
http://maven.apache.org/

*3 Selenium
http://seleniumhq.org/

*4 CruiseControl
http://cruisecontrol.sourceforge.net/

*5 Hudson
http://hudson-ci.org/

*6 Continuum
http://continuum.apache.org/

*7 Jetty
http://www.mortbay.org/jetty/

*8 ratproxy
http://code.google.com/p/ratproxy/

*9 Paros
http://www.parosproxy.org/

Subversion は、The Subversion Corporation の登録商標である。
Ant、Maven および Continuum は、Apache Software Foundationの商標である。
Firefox は、Mozilla Foundationの登録商標である。
Java およびすべてのJava関連の商標及びロゴは、米国及びその他の国における米国Sun Microsystems, Inc.の商標または登録商標である。
その他の会社名、製品名およびサービスは、それぞれ各社の商標または登録商標である。