Java EEベースの大規模開発における単体テストの実践的アプローチ

Practical Approach of the Unit Test in Java EE based Large-scale Developing

二木隆彰，新井　敦

要　約　Java EE（Java Platform, Enterprise Edition）はミッションクリティカルな大規模情報システムの基盤技術として広く普及しており，様々な開発支援ツールが提供されている．大規模な情報システムの開発において，品質と開発生産性を向上させるための重要なポイントの一つに，開発時の単体テストがある．膨大な量のソースコードで構成される大規模な情報システムはテストケースの数も多く，リグレッションテストを繰り返し実施するため，ツールを効果的に活用して手作業を効率化するための単体テストのアプローチが必要となる．

また，多くの開発者が同時並行で作業する大規模開発では，成果物や作業の手順を定義することで作業プロセスを標準化した上で，開発環境の統一や検証作業の問題，リグレッションテストの簡素化など，実践する上でのいくつかの課題に対応する必要がある．

進化する開発技術を活用することを前提としたテストアプローチを作り上げて改善していくことが，システムの品質と開発の生産性を向上させることにつながる．

Abstract　As for Java EE (Java Platform, Enterprise Edition), it is widely spread as a basic technology of the mission-critical, large-scale information system, and being offered a variety of development supporting tools. In the development of a large-scale information system, there is a unit testing when developing in one of the important points to improve the quality and the development productivity. The large-scale information system composed of a huge amount of source code needs the approach of the unit testing to use the tool effectively so that the number of test cases may also repeatedly execute a lot of regression testing and to make the hand working more efficient.

Moreover, it is necessary to deal with some problems in the standardization of development environments, problems of verification working, and the simplification of regression testing etc., and practice in large-scale development that a lot of developers work parallel to each other after the work process is standardized by defining the procedures of the deliverables and work.

Developing and improving the testing approach based on the use of the evolving developmental technology will leads to the improved quality of the system and the enhanced productivity of development activity.

1. はじめに

社会全体の情報化が進む中，企業や官公庁の情報システムは重要な業務データや個人情報が蓄積され，様々な機能が盛り込まれて大規模化している．情報システムが対象とする業務の範囲やユーザ層が広がるにつれ，企業の競争優位や業務の効率化に及ぼす影響も大きくなるため，システム開発における品質と生産性に対する要求も高まってきている．

業務遂行に欠かせないミッションクリティカルな大規模情報システムの基盤技術として，Java EE（Java Platform, Enterprise Edition）が普及している．Java EEは，数多くの情報シ

ステムで採用されて改良と拡充を重ねており，ベンダー各社も様々なサーバソフトウェアやフレームワーク製品，開発支援ツールを提供している．また，オープンソースソフトウェア（OSS）の機能性と品質の向上もめざましく，Java EE の普及に拍車をかけている．

Java EE を基盤技術とした大規模な情報システムの開発において，品質と開発生産性を向上させるための重要なポイントの一つに，開発時の単体テストがある．膨大な量のソースコードで構成される大規模な情報システムはテストケースの数も多く，不具合修正や仕様変更による再テスト（リグレッションテスト）を繰り返し実施する．開発工程の早い段階で不具合を検出して修正することができれば，それ以降の結合テストや総合テストの工程での手戻りも少なくなる．また，多くの開発者が同時並行で作業する中で品質や開発生産性を高めるためには，設計から実装，テストまでの一連の作業工程において成果物や作業の手順を定義することで作業プロセスを標準化する必要がある．開発支援ツールの機能や特性を理解して適切に活用することで作業プロセスを工夫することができれば，作業を効率化させるだけでなく，機械的かつ網羅的にチェックすることで単純な人的ミスを防ぐことにもつながり，品質と開発生産性の両面で大きな効果がある．

本稿では，Java EE を基盤技術とした大規模なシステム開発プロジェクトで実際に適用している単体テストのアプローチを紹介し，それを実践する上での課題とその対応策を示す．

なお，日本ユニシスは，本稿で紹介するアプローチを前提とした開発標準 MIDMOST for Java EE[6]を提供しており，MIDMOST for Java EE には本稿で示す開発支援ツールや技術ノウハウが含まれている．

2. Java EE ベースのシステム開発における単体テストの方法

本章では Java EE を基盤技術としたシステム開発における単体テストの基本的な方法を説明する．まず，本稿で前提とする単体テストの定義と Java プログラム開発での支援ツールについて述べ，次に，Java EE を基盤技術としたシステム開発でのテストの対象範囲と具体的な手順について述べる．

2.1 単体テスト

本稿で前提とする Java EE を基盤技術としたシステム開発における単体テストは，開発したソースコードが仕様通りに動作することを確認するテストであり，最小単位のモジュールに対して，入力と出力，内部処理の全てのパターンについてテストケースを定義し，仕様通りに正しく実装されていることを確認することを目的とした作業である．

一般的なテストの技法として，モジュール内部の構造や処理手順を考慮せずに設計書に定義された入出力（外部仕様）に着目したブラックボックス技法と，モジュールの構造やソースコードを前提にデータフローと制御フローの観点でコードが正しく動くことを確認するホワイトボックス技法とがある．テストケースの定義は，まず，設計書を基にブラックボックス技法でテストケースを挙げ，次に，ホワイトボックス技法により内部構造を確認しつつ，ブラックボックス技法で挙げたテストケースでは通過しないソースコード部分をテストするためのテストケースを追加していく手法が一般的である[1][5]．

こうしたテスト設計の視点としては，エクストリーム・プログラミングのプラクティスの一つに挙げられているテストファーストや，その提唱者である Kent Beck 氏が著した「Test

Driven Development」で示しているテスト駆動型開発のテストパターンが有名である[2].

2.2 Java プログラム開発と単体テスト支援ツール

Java プログラムの開発においては，開発作業を効率化するための支援ツールとして統合開発環境 Eclipse[8]が広く普及している．Eclipse は開発者向けのオープンソースソフトウェアであり，コード入力やコンパイルエラー修正の支援機能や強力なデバッグ機能等を備えているほか，単体テスト自動化ツール JUnit をはじめ，ソースコードの静的チェック処理や解析レポートを出力するプラグインも豊富に提供されている．ソースコードの品質とテスト作業の効率を向上させる様々な機能があり，Java によるシステム開発の現場で広く普及している．

一般的な Java プログラムの開発では，開発者は Eclipse 上でソースコードを作成して JUnit によりテストを実行する．JUnit によるテストでは，テスト対象のモジュールを呼び出して出力結果をチェックする処理をテストドライバとして実装する．一般に一つのテストケースに対して一つのテストドライバを作成するため，あるモジュールのテストでは複数のテストドライバを実行することになる．JUnit は，複数のテストドライバを一括で実行し，テスト結果の合否を集計して表示する機能を具備しており，リグレッションテストを繰り返す単体テストにおいては，これを採用することにより作業効率の大幅な改善が可能となる．

また，全てのソースコードを網羅的にチェックし，コーディングスタイルの規約違反やエラーの可能性が高いコードを検出する静的コード解析ツールも，膨大なソースコードを開発する大規模システム開発においては有効なツールである．

2.3 システムのアーキテクチャ

次に，Java EE を基盤技術とするシステムのアーキテクチャについて説明する．

一般的に Web ブラウザをクライアントとするような情報システムでは，クライアントからのリクエストを受け付け，画面上で入力された値を解析して業務処理を実行し，その結果として次の画面を表示する．業務処理のロジックでは必要に応じてデータベースを参照し更新する．この一連の処理を実現するための実行構造や，システム内部の構成要素とその関係を定義したものがアーキテクチャである．

Web ベースの情報システムにおけるアーキテクチャの基本的な考え方として，ここ数年で MVC2 モデル[*1] が定着してきている．図 1 に示すように，システム内部の役割を，業務ロジック（Model）と画面の入出力処理（View），これらの実行制御（Controller）とに機能分割することで，それぞれの機能の独立性を高めて依存関係を最小化することができ，大規模システムにおける分散・並行開発を可能にしている．

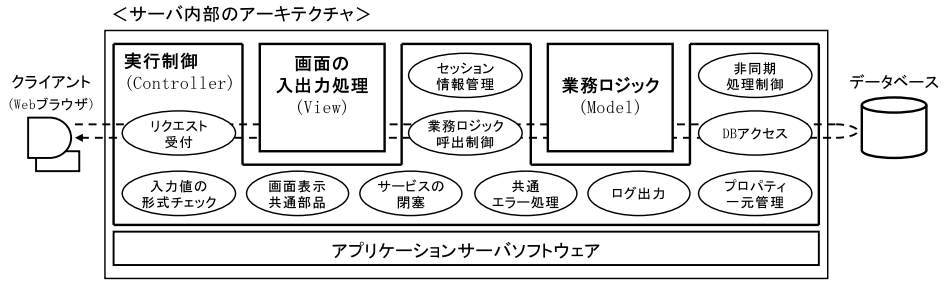


図 1　MVC2 モデルのアーキテクチャ構造

2. 4　単体テストの対象とテスト実施順

MVC2 モデルにおいて，実行制御（Controller）の機能部分はシステムが対象とする業務分野に関わらず共通であり，一般にフレームワークと呼ばれて汎用化されている．フレームワークを適用することで，新たに開発するシステムで実装してテストする範囲は「画面の入出力処理」部分と「業務ロジック」部分および，これらから呼び出される共通的な業務ロジックをライブラリ化したモジュールである「業務共通ライブラリ」に限定することができる．

また，呼び出し先のスタブ（完成していないプログラム部分の代用）を作成する手間を省くため，呼び出される下位のモジュールから単体テストを実施し，上位のモジュールのテストではテスト済みのモジュールを呼び出すボトムアップ方式のテストを実施する．

これらの呼び出し関係とテストの実施順を図 2 に示す．

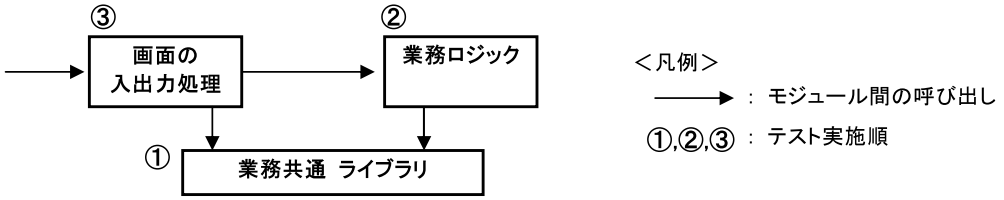


図 2　モジュールの呼び出し関係とテスト実施順

2. 5　モジュールごとの単体テスト

2. 5. 1　モジュール単体テストの手順

モジュールの単体テストの基本的な手順は，テスト対象のモジュールを構成するソースコードを解析する静的テストと，実際に動作させて実行結果を確認する動的テストに分けられ，図 3 に示す手順となる．

また，図 2 で示したモジュールの種類によって実施するテスト作業は異なる．他のモジュールから呼び出される業務共通ライブラリや業務ロジックの動的テストは JUnit テストが中心であり，補完的にデバッガテストを追加する．JUnit からの実行が難しい画面の入出力処理は打鍵テストが中心となる．この関係を表 1 に示し，それぞれのテスト手順の詳細を次項以降で説明する．

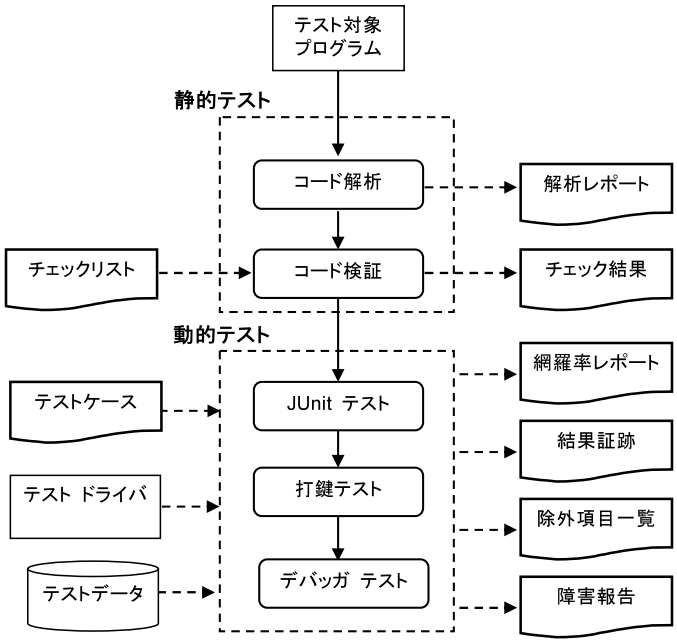


図 3　単体テストの基本的な手順

表 1　モジュールの種類によるテスト作業の違い

モジュールの種類	静的テスト		動的テスト		
	コード解析	コード検証	JUnitテスト	打鍵テスト	デバッガテスト
業務共通ライブラリ	○	○	○	―	○
業務ロジック	○	○	○	―	○
画面の入出力処理	○	○	―	○	○

○　：実施する作業

2.5.2　静的テスト

静的テストはソースコードを実行せずに確認するテストである．コード解析ツールやチェックリストによって潜在的なバグや標準化違反を検出して修正する．

1）　コード解析

Eclipse に組み込んだ以下のプラグインにより全ての Java ソースコードを網羅的にチェックし，コーディングレベルの誤りや不具合，標準化違反を検出して修正する．

・CheckStyle[9]：コードの記述形式（コーディングスタイルの規約違反）をチェックする

・FindBugs[10]：エラーの可能性が高いコードを検出する

2）　コード検証

標準化規約や実装方式等に関するチェック項目を挙げたチェックリストを予め用意し，他の開発者も含めてソースコードをレビューすることで，コード解析ツールだけでは検出できない複合的なバグや非効率なコード，プロジェクト独自の規約に対する違反を目視で検出する．

2.5.3　動的テスト

動的テストは，実際にソースコードを実行して結果を検証するテストである．

一つ一つのテストケースを実行するテストであり，テストケースに定義された入力値で実際に呼び出して実行し，期待された出力結果であることを確認する．テストケースを実行する方法として以下の三つのパターンがある．

1） JUnit テスト

テストケースに従ってテストドライバを実装し，Eclipse 上で JUnit により実行する．業務ロジックやデータのチェック処理などのサブルーチンを呼び出すためのテストドライバを実装して JUnit で実行してテストする．

2） 打鍵テスト

JUnit によるテスト実施が難しい機能部分は，手作業で入力してテストする．例えば，画面からの入力を受け付けて次画面の表示内容を確認するようなテストケースはテストドライバの実装が難しいため，Web ブラウザから手作業で入力操作し，その実行結果の画面を目視で確認する．

3） デバッガテスト

JUnit テストと打鍵テストでも実施が難しい機能部分は，Eclipse のデバッガ機能を使ってテストする．例えば，発生したエラーを処理する部分は，ファイル I/O や OS に起因するシステム障害を発生させる必要があり，このための設定や環境を作り出すことが難しい．このようなテストケースの場合は，Eclipse のデバッガ機能を使ってテストする．Eclipse 上でテスト対象のコード部分にブレイクポイントを設定してステップ実行することができ，Eclipse 上の操作によって仮想的にエラーを発生させることもできる．

デバッガテストでも実施できない部分は，テスト除外項目として一覧表に列挙し，重点的なコードレビューを実施することでコードの正当性を確認する．

また，システム開発におけるテストの対象には，JSP や JavaScript，Shell スクリプトなど Java 以外の言語で実装したモジュールもあり，これらには Eclipse 上でのコード解析や JUnit テストは実施できないため，レビューによるコード検証と画面やコマンドラインから打鍵によるテストを実施する．

2. 5. 4　単体テスト手順に関する考察

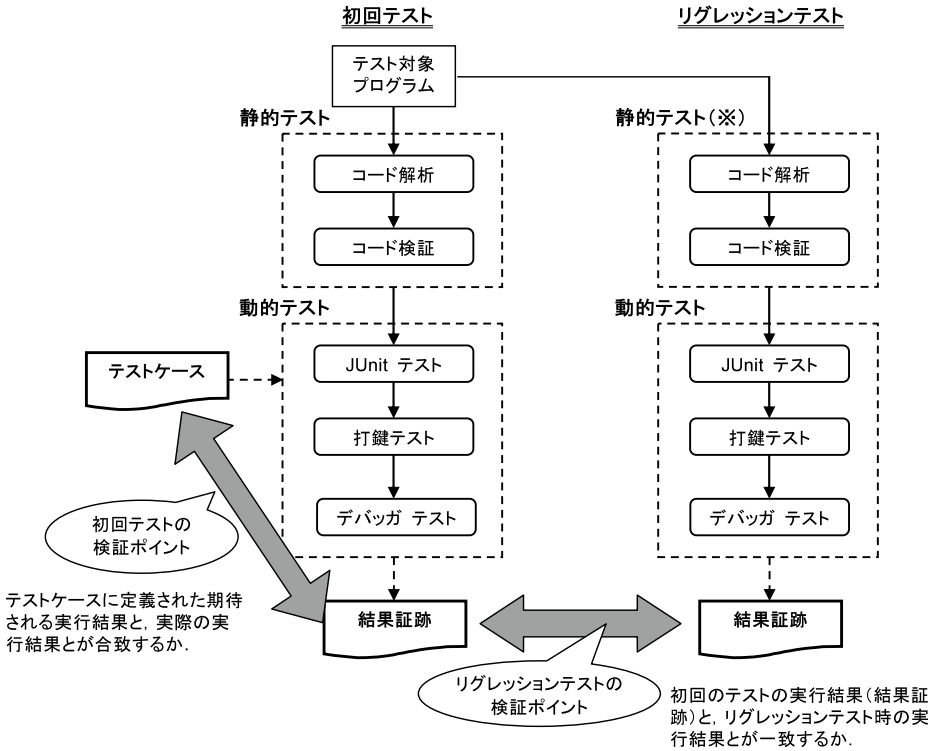
このように単体テストの工程を多段階のステップに分けることは，一見すると開発作業の手間を増やしているように見えるが，ツールを効果的に活用して手作業の効率化を図り，テストの品質を向上させるために必要な手段である．ツールで機械的かつ網羅的にチェックできるところはツールに任せ，それに合格している前提で目視確認ができれば，目視で確認すべき項目や考慮すべき範囲を限定することができ，手作業によるテストを効率化することができる．ツールは単純なチェックしかできないが，網羅性と処理スピード，単純な見落としのなさにおいて優れている．一方，複数の項目間の関連や意味的な整合性を確認することは，人による目視の方が優れている．ツールによる自動化と人による目視をうまく組み合わせて手順やチェック内容を考慮することで，高い品質と効率的な作業を実現できる．

2. 6　リグレッションテスト

大規模なシステムでは内部の処理構造は複雑で相互に関係しており，一つのソースコードの

修正が想定外の機能部分に影響して不具合となることもある．このため，テスト対象のソースコード，もしくは，そこから呼び出す先のモジュールのソースコードを修正した際には，リグレッションテストを実施する．特に業務共通ライブラリは，システム全体の整合性を考慮して設計仕様の変更や機能を追加することも多く，頻繁にソースコードが更新されるため，それらを呼び出すモジュールはその都度リグレッションテストを実施する必要がある．

リグレッションテストでは初回のテストと同じテストを実施するが，初回テストとは検証のポイントが異なる．初回テストではテスト結果がテストケースの期待結果と一致するかを中心に検証するが，リグレッションテストでは再テストの結果と初回テストの結果を突き合わせて比較し，テスト結果に差異がないことを中心に検証する．図 4 にその概要を示す．



※リグレッションテストの対象のソースコードに変更がなければ，静的テストは省略する．

図 4　初回テストとリグレッションテスト

このように，初回テストとリグレッションテストとで検証ポイントが違う理由は，テスト結果の検証作業の難易度にある．

テストの実行結果の証跡は，データベース上の更新レコードやファイルに出力されたログ，画面に表示された情報である．これに対して，テストケースに定義された期待される実行結果は，システムが対象とする業務の知識やデータモデルの理解を前提とした文章で記述されており，これらを解釈してテストの結果証跡から合格であると判断するためには，付随する業務データも含めた確認作業と相応の業務知識を必要とし，テスト結果の検証は簡単な作業ではない．

リグレッションテストでは，初回テストで合格であると判定した結果証跡が揃っているため，これとリグレッションテスト時の結果証跡を突き合わせて比較することで，比較的容易にテスト結果を検証することができる．

3. 大規模開発における課題と対応策

本章では，これまでに紹介した単体テストのアプローチを大規模システム開発で実践する上での課題とその対応策について示す．

大規模なシステム開発では，多くの開発者を必要とする物理設計とプログラミング，テストの工程において，必要なタイミングで必要な数の開発者を調達するプロジェクト制組織が導入されており，経験やスキルが異なる多くの開発者が同時並行で作業を進めることで様々な問題が発生する．

本稿で対象としている大規模なシステム開発の例として，実際のプロジェクトのプロフィールを表2に示す．

表2　大規模なシステム開発のプロフィール

	プロジェクトA	プロジェクトB
開発期間	30ヶ月	25ヶ月
物理設計/プログラミング/単体テスト期間	14ヶ月	11ヶ月
総ステップ数［単位：千Step］	7,237	1,758
単体テスト対象ステップ数	926	628
画面数	527	183
帳票数	156	10
クラス数	9,042	3,714
DBテーブル数	464	140

本章では，こうした大規模なシステム開発プロジェクトにおいて，2章で述べた単体テストのアプローチを実践する上での課題を挙げて，その対応策を示す．

3.1 開発環境の統一

3.1.1 課題

多くの開発者が同時並行で作業を進める場合，開発者ごとの開発環境の差異に起因するトラブルが発生する．

システムを構成する個々のソースコードは，開発者それぞれのPC上で実装して単体テストまで実施する．次に，それぞれの開発者が作成した膨大な数のソースファイルを最終的に一つの実行環境上に集めて結合テストを実施する．このとき，開発者ごとのPCの開発環境の違いによって，単体テストは通過したが，結合テストで不合格になることがある．ここでいう開発環境とは，開発者の各PC上でのフォルダ構成やシステム変数の設定値，Java EEの一般ライブラリや業務共通ライブラリのバージョンである．大規模な開発になると1年以上の開発期間を要するため，その間にミドルウェア製品のライブラリもバージョンアップされる．最近ではインターネット上から最新のライブラリをダウンロードできるが，プロジェクトとしては実績のある安定版を採用するケースもある．

こうした開発環境の差異によってテスト結果にも違いが生じる場合がある．大規模開発で開発者が増えるほど，開発者ごとの開発環境に差異が生じることが多くなり，開発環境の違いに

起因するトラブルも発生しやすくなる．開発環境に起因するトラブルは，多くの要素が複合的に関係した特殊な状況下で発生するために原因調査にも時間を要することが多い．

3.1.2　対応策

開発環境の設定手順や使用するツールやライブラリのバージョンをルールとして明文化して開発者に周知徹底するだけでなく，開発者の PC 上の権限を制御して設定を変更できないようにしたり，外部メディアやインターネットの接続を制限するなど，自由にツールをインストールできないようにする対策も必要である．

また，プロジェクト内で開発する業務共通ライブラリは，システム開発を続ける段階で仕様変更や不具合の修正によって更新され，定期的に配布される．このバージョン管理を徹底するとともに，配布と開発者の PC への取り込み手順の簡易化，開発環境にインストールされているライブラリのバージョンをチェックするツールの提供などで，開発者の作業負担を減らす工夫が必要である．

3.2　JUnit による自動テスト

3.2.1　課題

テストケースに従ってテスト対象のモジュールを呼び出して実行し，その実行結果を確認するようなテストでは，テストドライバを実装して Eclipse から JUnit で実行するが，その実装と検証には以下に挙げる二つの課題がある．

一つ目の課題は，テストドライバの実装負荷である．一般に，入力値の組み合わせや内部の処理パターンは複数あり，一つのテスト対象に対してテストケースは複数存在する．このため，テストケースごとに作成するテストドライバのコード量は多くなり，テスト対象のモジュールを実装するよりも多くの工数を費やすこともある．リグレッションテストまで含めると，テストドライバを実装して JUnit でテストを自動化するメリットは大きいが，テストの準備には相応の手間がかかる．

もう一つの課題はテストドライバの検証である．実装したテストドライバは，テスト対象のモジュールの仕様を正しくテストできるか，呼び出す際の入力値の作成や出力結果を検証するテストドライバのソースコードに不具合がないかを検証する必要がある．入力値のパターンや業務ロジックが比較的単純であればテストドライバの検証は容易だが，結果の値が単一ではなく，DB 上の複数レコードの更新やログの出力，複雑なデータ構造のオブジェクトである場合には，テストドライバが正しいことの検証が難しくなる．テストドライバの検証を怠ると，テストケースを間違った仕様で実装し，JUnit 上のテスト結果では合格だが，意図したテストを実施できていないテストケースが生じる．

大規模開発においては，必ずしも全ての開発者が対象の業務処理やデータに精通しているわけではなく，こうした誤り（テスト結果は正常であるが，テストの設計と実装が間違っている）が起きやすく，業務仕様に精通した要員によるテストドライバの検証が必要となる．しかし，テストドライバの検証には，Java のプログラミングや JUnit の動作原理の知識も必要であり，開発チームの業務担当者や顧客側担当者には，テストドライバの検証はさらに難しいことになる．

3.2.2　対応策

一つ目の課題であるテストドライバの実装負荷については，特に有効な対応策はなく，開発スケジュールに反映させる必要がある．これを軽減するアイデアとして，テストケースの入出力情報の定義ファイルを読み込ませて JUnit を自動で実行させるツールが考えられる．そのようなツールができれば，開発者は入出力情報の定義ファイルの作成だけでテストが実行できるため，作業負荷は大幅に軽減される．テストケースには様々なパターンがあり，その全てに対応できるツールを作成することはできないが，一部のテストケースだけでもツールを適用することができれば，一定の効果は得られる．

次のテストドライバの検証の課題に対する対応策として，テスト証跡（入出力情報，DB ダンプ，ログ）の検証に関しては，目視で確認することが挙げられる．目視による検証は大変な作業負荷にはなるが，検証者の多くにとって，テストドライバのソースコードを理解して確認するよりも，現実的で確実な方法である．さらに，テストドライバはテスト対象のモジュールを呼び出して実行するだけとなり，ソースコードの実装の手間も少なくなる．

ただし，目視による作業を効率化するためには，以下のような作業を自動化するテスト作業支援ツールを提供する必要がある．

・出力されたログの該当部分を切り出してファイルに保存する
・結果画面のイメージをキャプチャしてファイルに保存する
・テスト証跡を検証者が見やすい形に編集する
・更新された DB テーブルを対象に，差分のレコードを抽出する

JUnit はテストを一括で実行するための仕組みとしては有効であり活用すべきだが，テスト設計や結果検証の作業品質を担保するためには，目視での検証と，それを考慮した仕組みが必要である．

3.3　リグレッションテストの簡易化

3.3.1　課題

不具合や仕様変更によるコード修正が行われた場合，関連するモジュールのリグレッションテストを実施するが，大規模開発ではテスト対象のモジュール数と繰り返されるリグレッションテストの回数が多く，何回も繰り返されるリグレッションテストの度にひととおりのテストケースを手作業で実行していたのでは，テストの実施と結果検証に膨大な作業負荷がかかってしまう．

3.3.2　対応策

まず，リグレッションテストは極力，JUnit 上で実行できるようにすべきである．そのためには，モジュール分割や入出力仕様の設計，ソースコード実装の段階から JUnit で実行できることを意識して進めるべきである．JUnit 上で単体テストの実行を自動化するためには，テストドライバの実装など，いくつかの手間がかかるが，リグレッションテストを繰り返すことを考えると，単体テストの実行と集計が自動化されるメリットは大きい．

次に，テスト結果の検証作業を効率化させるための工夫も必要である．2 章で述べた通り，リグレッションテストでは再テストの結果と前回のテスト結果を突合し，差異があるかを検証する．この検証作業のために，呼び出し引数や結果の構造体オブジェクトをダンプ出力する仕

組みや，前回のテスト結果の証跡と今回の証跡とを比較してその差異を検出するツールを提供するなどの工夫が必要である．

3.4　テストデータ

3.4.1　課題

テストの入力となる業務データの質は，テストの精度に大きな影響を与える．

テストを実施するためには，マスターテーブル上のレコードや設定値などの定数，二次的に参照する業務データなど，テスト対象のモジュールを動作させるために必要な最低限の入力データを用意する必要がある．しかし，対象業務の知識や経験が豊富な開発者は少なく，開発者が想定する業務データと実際の業務データとは大きく違う場合がある．開発者の思い込みで作成した誤った業務データによるテストは意味がないばかりでなく，誤った業務処理を実装することにもなる．さらに，開発者がそれぞれ作成したテストデータが間違っていると，モジュール間の呼び出しインターフェイスで思わぬミスマッチを引き起こす．

3.4.2　対応策

基本的な業務処理を動作させるために必要な最小限の業務データのセットは，開発者がそれぞれ作成するのではなく，開発プロジェクト全体で用意して開発者に提供する必要がある．それぞれの開発者は提供された業務データのセットに対してデータを追加し，テストケースに応じて必要なバリエーションを増やしていく．

基本的な業務データのセットを共通で提供することは，開発者のテスト準備の作業を軽減するだけでなく，開発者の対象業務の理解を向上させることにもつながる．

ただし，実際の業務データは顧客情報の漏洩にもつながるため，一部をマスキングしたデータやランダムに加工処理した業務データを使うことが一般的である．

3.5　テストの証跡の電子化

3.5.1　課題

単体テストの実行結果の証跡には，テスト実行前後の画面の表示イメージや DB 上のデータの差分，出力されたログなどがあり，大規模なシステム開発で全てのテストケースの証跡を集めて紙に印刷すると膨大な量になる．さらに開発現場では，多くの開発者が数少ないプリンタに同時に出力することで競合し，作業時間の無駄にもなる．

3.5.2　対応策

テスト結果の証跡は Excel ファイルや PDF[*2] の電子ファイルとして開発環境のファイルサーバ上に決められたルールで保管することで，検証者は紙に印刷せずに PC 上で確認することができる．PDF ファイルで保存した証跡は，Acrobat[*3] の注釈機能により指摘内容を PDF ファイル上に追記することができ，証跡の検証結果をチーム内で共有する上でも大きな効果がある．さらに，電子印鑑機能で検証完了時の検証印にも利用できる．

テスト証跡を電子的に管理することで検索性が向上し，出力された業務データやログから該当箇所を容易に探すことができる．紙資源の節約と印刷時間をなくすだけでなく，それを活用して工夫することで作業効率の向上にもなる．

4. お わ り に

近年のシステム開発においては，開発支援ツールやアーキテクチャの技術が進歩したことで，テストのアプローチも複雑になっている．2章で示したように，ツールと手作業をうまく組み合わせたテストの手順を定義し，アーキテクチャ上のモジュールの種類や特性を考慮してテストのアプローチを工夫する必要がある．ツールによるチェック作業の機械化やその自動化は作業効率の向上や単純ミスの防止といった恩恵をもたらす．しかし，ツールで確認できる範囲や検出できる不具合は限られており，ツールの恩恵を最大限に享受しつつ，手順化された手作業を的確に実施する必要がある．

また，大規模なシステム開発では，多くの開発者が同時並行で作業することを前提に，開発環境やテストの検証方法，テストデータについて，テストの計画と準備が重要である．リグレッションテストの自動化やツールによる検証作業の支援は大きな効果があり，システム開発全体の品質と生産性に与える影響も大きいだけに，プロジェクトの状況に合せた実践的なアプローチが求められる．

コンピュータの処理速度や記憶容量の向上は目覚しいものがあり，それに合わせて支援ツールや開発環境も急速に発展している．一昔前には想像もしなかったような開発手法を可能にしており，進化する開発技術にキャッチアップし，それを活用することを前提としたアプローチや開発環境を作り上げて改善していくことが，システムの品質と開発の生産性を向上させることにつながる．

＊1 MVC2は，Javaベースの Webアプリケーションを構築するための開発モデルである．Smalltalk-80（スモールトーク）で確立された MVC モデルを基本として JavaEE の技術要素に対応させ，システム内部構造を Model-View-Controller に分けて実装する考え方を示している．

＊2 PDF は，Adobe Systems 社が開発した電子文書のフォーマット Portable Document Format の略称である．同社が提供するソフトウェア Adobe Reader で参照できる．

＊3 Acrobat は，Adobe Systems 社が提供する，PDF ファイルを編集するためのソフトウェアである．

参考文献

［1］ G. J. Myers 他，ソフトウェアテストの技法第2版，近代科学社，2006年8月

［2］ Kent Beck, Test Driven Development, Addison-Wesley Professional, Nov. 2002

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

［4］ ソフトウェア・テスト PRESS Vol.1-4，技術評論社，2005年6月－2007年1月

［5］ 大塚俊章，荻野富二夫，ソフトウェアテスト技術，ユニシス技報，日本ユニシス，Vol.27 No.2 通巻93号，2007年8月

［6］ 新井敦，日本ユニシスにおける開発標準の策定と適用への取り組み，ユニシス技報，日本ユニシス，Vol.27 No.2 通巻93号，2007年8月

［7］ Mary Poppendieck，Tom Poppendieck，平鍋健児，高嶋優子，佐野建樹訳，リーンソフトウエア開発，日経 BP 社，2004年7月

［8］ Eclipse, http://www.eclipse.org/

［9］ CheckStyle, http://checkstyle.sourceforge.net/

［10］ FindBugs, http://findbugs.sourceforge.net/

執筆者紹介　二　木　隆　彰（Takaaki Futatsugi）

2007 年日本ユニシス(株)入社．Java EE ベースのシステム開発のアーキテクトとして，開発プロジェクトの技術支援を手がけている．現在，総合技術研究所 OSS センターに所属．

新　井　　敦（Atsushi Arai）

1992 年 筑波大学卒業．同年 日本ユニシス(株)入社．電力・ガス・通信の顧客システム構築を中心に，開発プロジェクトを手掛ける．情報処理技術者 アプリケーション・エンジニア．企業派遣により，早稲田大学大学院アジア太平洋研究科 MOT コース（国際経営学修士課程）を 2004 年に卒業．現在，総合技術研究所 OSS センターに所属し，社内プロダクトの開発と，開発プロジェクトへの技術支援を担当．