自社開発 Java フレームワークを
長期的に維持・利用するための，
テストの自動化について

ヤマトシステム開発株式会社

■ 執筆者Ｐｒｏｆｉｌｅ ■

尾崎　隼都

2000 年　ヤマトシステム開発（株）入社
　　システム開発業務担当
2005 年　現在 システム技術グループ所属
　　Web 基盤技術担当

■ 論文要旨 ■

自社で開発した Java フレームワークが社内に普及していくにしたがい，長期的に維持・利用する必要が出てきた．それに伴い，業務システムで使用する複数アプリケーション・サーバ上でのテスト・コスト増大，機能拡張時のリスク，マルチスレッドにおける不具合の検出が困難などの問題が発生するようになってきた．

この問題を解決するために，エクストリーム・プログラミングと，テスト駆動型開発をヒントに，自社フレームワークに対するテストを自動化した．

その結果，複数アプリケーション・サーバ上でのテスト・コスト削減，機能拡張時のリスク低減，マルチスレッドにおける不具合の検出を容易にして改善を図った．

今後は，テストプログラム作成におけるコスト削減について調査し，テストプログラムのコードカバレッジ率 100%を達成していく．そして，今回のテストの自動化における成果を，業務システムのテストの自動化に活かしていきたいと思う．

■ 論文目次 ■

■ 図表一覧 ■

１．はじめに

１．１　当社概要

当社は，ヤマトグループの中で『e-ビジネス事業』を担当する情報サービス会社である．ヤマトグループの情報サービス業務を担当するだけではなく，そこで蓄積したノウハウをもとに，情報・通信・物流を組み合わせた企業向け ASP サービスや，情報システム開発などの情報サービス事業を展開している．

私の所属するシステム技術グループは，最新の技術動向からシステム開発や運用に役立つ技術を調査・検証し，社内への普及を推進している．最近では，IC タグを利用したトレーサビリティ・システムの作成・検証をはじめ，仮想マシン技術を利用したサービスの適用実験，LibertyAlliance を利用したシングル・サインオンの検証，Microsoft .NET Framework 及び Java 用アプリケーション・フレームワークの企画・開発・運営などの取り組みを行っており，これら技術の社内への普及を推進している．

１．２　Web システム化への流れ

2001 年にヤマト運輸株式会社の基幹システムを汎用機からオープン・システムへ移行するというプロジェクトが開始され，システム基盤の Web システム（Java）化が推進された．この Web システム化への流れの中で，Web システム用のフレームワークが社内に確立されていなかったため，新たに自社で開発し，社内で開発するシステムに適用してきた．

社内で開発した Web システム用のフレームワーク（以下，自社フレームワークという）の目的は，一般的にいわれるフレームワークの目的と同じように，『処理の枠組みをあらかじめ定めて作業を効率化し，成果物の品質を向上させること』である．

２．自社フレームワークの概要

２．１　適用想定システム

自社フレームワークは，Java をベースとしたシステムに適用することを想定しており，Java2 Platform Standard Edition 1.2（以下，J2SE1.2 という）以降，Java2 Platform Enterprise Edition 1.2（以下，J2EE1.2 という）以降で動作するように設計・実装されている．また，特定の業務に依存しないように設計・実装されており，主に下記のようなシステムに対して適用される．

- Servlet/JSP[①]を使用した Web システム
- バッチ処理を実行する Java アプリケーション

[①] Java 言語を用いた Web アプリケーションの開発技術．現在，広く世界で利用されている．

２．２　開発時期

自社フレームワークは，2001 年に初期バージョン（Ver 1.0）が開発された．それ以降，図 1 のようにバージョンアップを続け 2003 年にマイナー・バージョンアップを終了した．その後，動作させるミドルウェア環境（アプリケーション・サーバ）の組み合わせが増加したが，自社フレームワーク自体の機能は拡張していない．

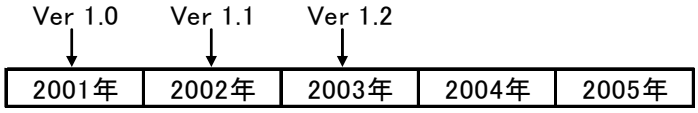

図１　自社フレームワークの開発年表

２．３　提供機能

自社フレームワークが提供している機能は，業務システムを開発する場合に必要になる汎用的な機能に絞っている．下記に，提供している機能の概要を示す．

- Web 画面制御機能
- データベース操作・制御機能
- ユーティリティ機能
- 開発・運用ログ出力機能
- Web 画面・ユーティリティに対してのテスト実行機能

データベース操作関連の機能は，開発・運用時のシステム監視を意識して，パフォーマンス・ログなどの機能を強化している．

３．今までの問題

自社フレームワークの社内への適用を開始してから４年が経過し，適用数が約 70 システムに増えるにつれて，次のような問題を抱えるようになってきた．

３．１　複数アプリケーション・サーバ上でのテスト・コスト増大

自社フレームワークは，J2SE1.2・J2EE1.2 以降をサポートするアプリケーション・サーバ上で動作させるため，各アプリケーション・サーバ製品ごとにテストを行う必要がある．（現在，社内で主に使用されているアプリケーション・サーバ製品の種類は，バージョンを含めると 12 種類に及ぶ）

通常，Java で作成されたソフトウェアは，移植性が高いといわれている．しかし，アプリケーション・サーバ上で安定して動作するかどうかは，それぞれの環境で確認する必要がある．筆者は，あるアプリケーション・サーバ上でテストした際，特定のオペレーティング・システム環境下の Java 実行環境に問題があり，動作しなかったという経験をした．

このような問題が，システム開発中や運用中に発生した場合，業務の進行や運用に多大な影響を与えてしまう．よって，問題が発生する前に，動作させるアプリケーション・サ

ーバ上でテストを行っておく必要がある.

３．２　自社フレームワーク機能拡張時のリスク

自社フレームワークは，開発者・運用者のニーズに沿って継続的に機能拡張を行っており，その際，変更された部分だけをテストすることが多い．この場合，変更していない部分に対して影響があるかどうかは，テストされないことになってしまい，機能を拡張することに対してリスクが高くなる.

３．３　マルチスレッドにおける不具合の検出が困難

自社フレームワークは，Web システム用のフレームワークとして使用されることを想定している．そのため，マルチスレッド[①]環境で安定して動作する必要がある.

しかし，マルチスレッド環境で安定して動作することを検証するには，意図して競合状態を作り出す必要があり，困難を伴う．実際，検証作業の際は，通常より厳密なソースコード検証と，動作検証テストを繰り返し行った．そのため，検証作業に非常に時間が掛かってしまった.

４．ソフトウェア開発の歴史からの教訓

前章で取り上げた問題を解決するにあたって，ソフトウェア開発の歴史から教訓を得られるのではないかと考えた．さまざまな開発技法を調査して，エクストリーム・プログラミング[②](以下，XP という)と，テスト駆動型開発[③]からヒントを得られるのではないかと考えた．それらの開発技法は，ソフトウェア開発の歴史からさまざまな経験を実践として取り入れている．それらの考え方の中で，効果がありそうな実践について考えてみる.

XP とテスト駆動型開発の技法の中で，中核とも言えるものに『テストの自動化（テスティング）』がある．『テストの自動化』とは，作成したモジュールに対してのテストを行うプログラム（以下，テストプログラムという）を作成し，テストを自動化する技法である．テストプログラムを簡単に説明すると，テスト対象の関数に対してある値を入力して，期待した結果が得られるかどうかを調べるためのプログラムが定義してあるものである.

前章で挙げた問題に対して，テストを自動化することで期待できる効果について考えてみる.

４．１　複数アプリケーション・サーバ上でのテスト・コストへの効果

一つ目の効果として，複数アプリケーション・サーバ上での自社フレームワークのテスト自動化による，テスト・コスト削減が期待できる.

---

① 一つのアプリケーションがスレッドと呼ばれる処理単位を複数生成し，個々のスレッドが平行して複数の処理を行う事．複数スレッドを協調するように制御する必要があるため，プログラミングが難しい.

② ソフトウェア開発手法の一つ．ウォータ・フォール型のように各工程を順番に実施していくことよりも，常にフィードバックを行い修正・再設計していくことを重視している.

③ プログラムを作成する前に，テストプログラムを先に作成するという開発方法．テストを行った後に再設計を行い，それを繰り返していく事で品質を高めていく.

今までは，複数アプリケーション・サーバ上でのテストを手動で行っていた．そのため，テストの実行や実行結果の検証など，多くのことを手動で行っていたため時間が掛かってしまった．テストを自動化することで，それらのテスト時間の削減が期待できる．

４．２　自社フレームワーク機能拡張時のリスクへの効果

二つめの効果として，自社フレームワークに対する機能拡張を低リスクで実施できることが期待できる．

テストの自動化によって，修正した部分を含め全体に対するテストが行えることで修正ミスの検出が可能になる．それによって，機能拡張に対するリスクを最小限に押さえることが可能になる．

４．３　マルチスレッドにおける不具合の検出への効果

三つめの効果として，自社フレームワークのマルチスレッドにおける不具合の検出が期待できる．

機能を拡張した際，再度マルチスレッドにおける競合状態の検証を行う必要がある．この検証作業は，競合状態を作り出すことが困難なため，テストに非常に時間が掛かり，テスト作業自体にも困難なことが多い．

その検証作業の際，テストの自動化によってフレームワーク全体を繰り返しテストすることで競合状態を作り出しやすくなり，不具合を検出することが容易になる．

５．テスト自動化の仕組み

ここまで自社フレームワークの現在の問題と，その解決方法の可能性について述べた．ここからは，今回適用したテストの自動化の仕組みについて説明する．

テストの自動化を実現するテストツールとして，単体テスト向けのフレームワークである JUnit がよく知られている．JUnit は，簡単に使えて Java プログラムをテストできるテスト・フレームワークである．しかし，今回はテストプログラムの繰り返し実行や，マルチスレッドによる実行をより簡単に実現するため，新たにテスト実行用の仕組みを作成した．

５．１　テスト方式

今回は，テスト方式として一般的に知られている『テスト・ドライバ[①]のみを使用する方式』と，『テスト・ドライバとテスト・スタブ[②]を併用する方式』で行った．

(1)　テスト・ドライバのみを使用する方式

この方法は，主に自社フレームワークのユーティリティ系モジュール[③]のテストに使用する．

図２は，『テスト・ドライバのみを使用する方式』の概念図である．それぞれのモジュールが起動される順番を示している．

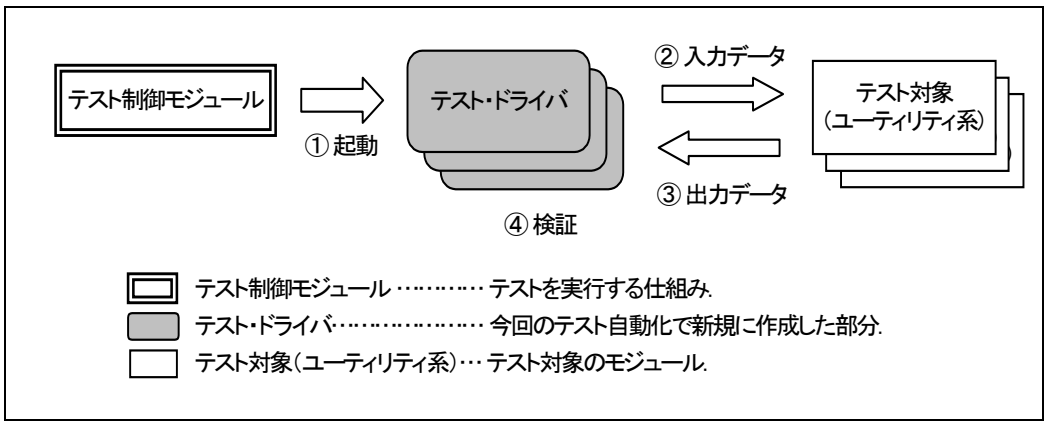

図２　テスト・ドライバのみを使用する方式の概念図

(2)　テスト・ドライバとテスト・スタブを併用する方式

この方法は，自社フレームワークが提供しているサーブレット系モジュール（Web 用モジュール）に対してテストを実行する．サーブレット系モジュールに対してのテストでは，HTTP で通信を行う必要があるため，Apache Jakarta Project[④] が提供している HTTP Client を使用してテストを行っている．

図３は，『テスト・ドライバとテスト・スタブを併用する方式』の概念図である．それぞれのモジュールが起動される順番を示している．

---

① 呼び出す側のプログラムを模倣するテスト用プログラム.
② 呼び出される側のプログラムを模倣するテスト用プログラム.
③ メインの機能に対して，文字列操作のような補助的に役立つ有効な機能.
④ Apache HTTP Server で有名な，Apache Software Foundation で運営されているプロジェクトの一つ.

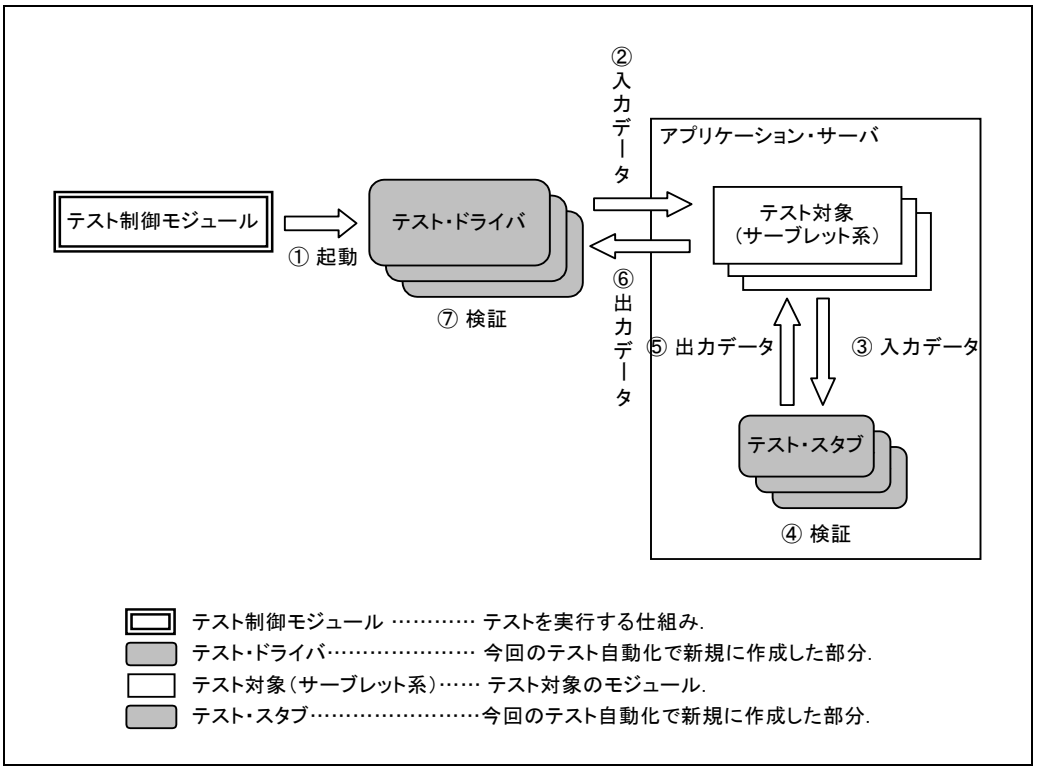

図３　テスト・ドライバとテスト・スタブを併用する方式の概念図

この二つのテスト方式で使われているテスト・ドライバとテスト・スタブの中では，assert 関数を使用して，結果・状態の検証を行う．assert 関数には、引数として真偽値を指定する．

```
assert(真偽値);
```

assert 関数は具体的には下記のように使用する．下記は，add 関数に引数で指定された二つの数値を足すという assert の例である．

```
assert(add(１， ２) == ３);
```

上記の例は，『add(１， ２)を実行した結果は，３でなくてはならない』と説明できる．この例では，テスト対象関数の『add(１， ２)』を実行した結果が『３』以外の場合，テストは失敗したと判断される．

このように，テスト・ドライバとテスト・スタブの中では，テスト対象のモジュールが出力する結果に対して assert 関数を使用している．

５．２　テスト実行方法

テストの実行は，『テスト・ドライバを単体で動作させて検証する方法』と，『複数のテスト・ドライバを一括で動作させて検証する方法』の二つで行った．

(1)　テスト・ドライバを単体で動作させて検証する方法

この方法は，テスト・ドライバを作成した直後に GUI のテスト制御モジュール（**図４**）を使用して，正常に動作するかを確認することを目的としている．その際，正常に動作するかを検証すると同時に，テスト対象プログラムのコードカバレッジ率[①]の測定を行った．

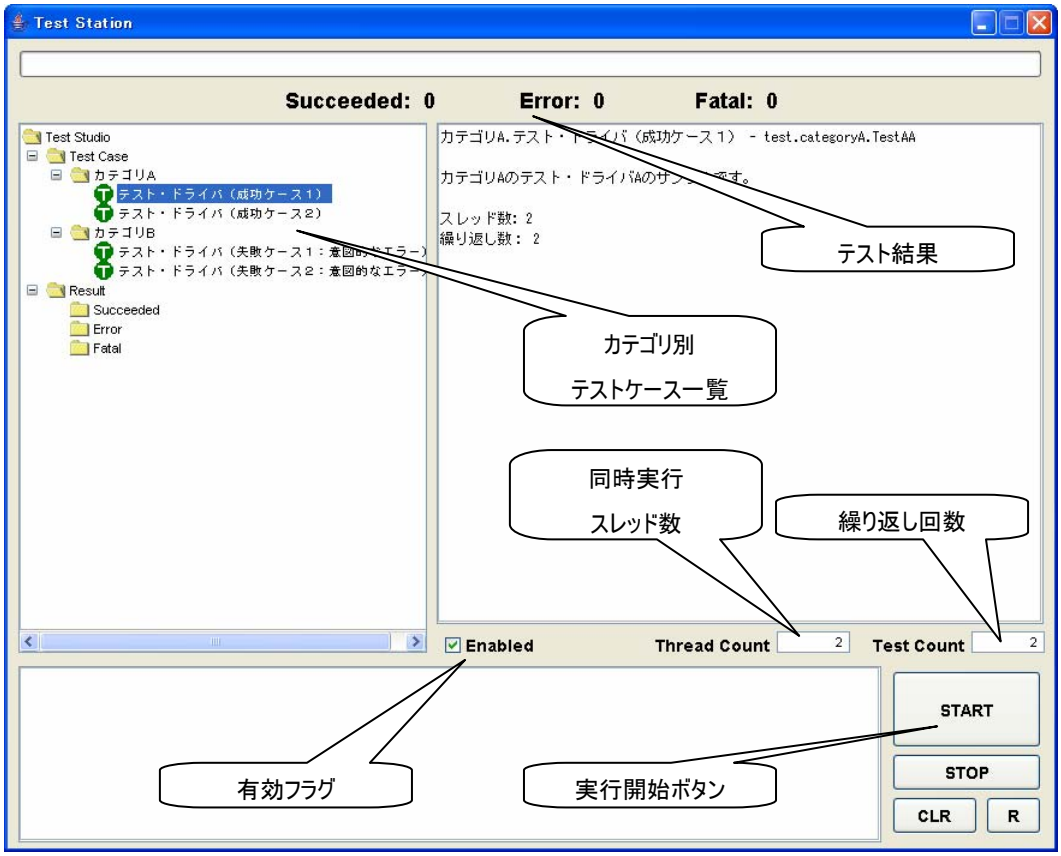

図４　テスト制御モジュール

(2)　複数のテスト・ドライバを一括で動作させて検証する方法

この方法は，テスト・ドライバが単体で動作することを確認した後に，繰り返し実行することと，マルチスレッドで動作させて検証を行うことを目的としている．テスト・ドライバを繰り返しマルチスレッドでテストするのは，耐久性テストと，競合状態テストを行うためである．今回作成したテスト・ドライバとテスト・スタブは，単体テスト，耐久性テスト，競合状態テストを考慮して同時に平行して動作させられるように実装されている．

耐久性テストとは，データベース接続管理プログラムのテストや，ファイル管理プログラムのテストなどを繰り返し動かして，データベース・コネクションや，ファイルなどの

---

① 命令網羅率とも言われる．プログラムの全命令数における，実行済み命令数の割合で，品質を知る一指標．

資源が適切に開放されているかを確認するためのテストである．資源が適切に管理されていない場合，オペレーティングシステムのプロセスなどの資源管理の限界に達するため，テスト・ドライバは異常終了する．耐久性テストを行うことで，長時間動作できることを確認できる．

競合状態テストとは，マルチスレッド環境下でモジュールが安定して動作するかを確認するためのテストである．通常，マルチスレッド環境下で競合状態を発生させることは難しい．そのため，繰り返し長時間に渡ってテストプログラムを動作させることで，競合状態を発生させやすくする．競合状態が発生しやすい環境でテストすることで，マルチスレッド環境下で安定して動作するかを検証できる．

５．３　テスト結果の検証方法

ここまで，テストの実行方法について説明してきた．ここからは，テストの実行結果の検証方法について説明する．

(1)　テストプログラムを単体で動作させて検証する方法

この方法では，自社フレームワークのテスト用 GUI のテスト制御モジュールを利用してテスト実行する．テスト結果は，**図５**のように画面の上部に表示されるようになっており，テストの成功や，失敗した箇所が簡単に分かるようになっている．

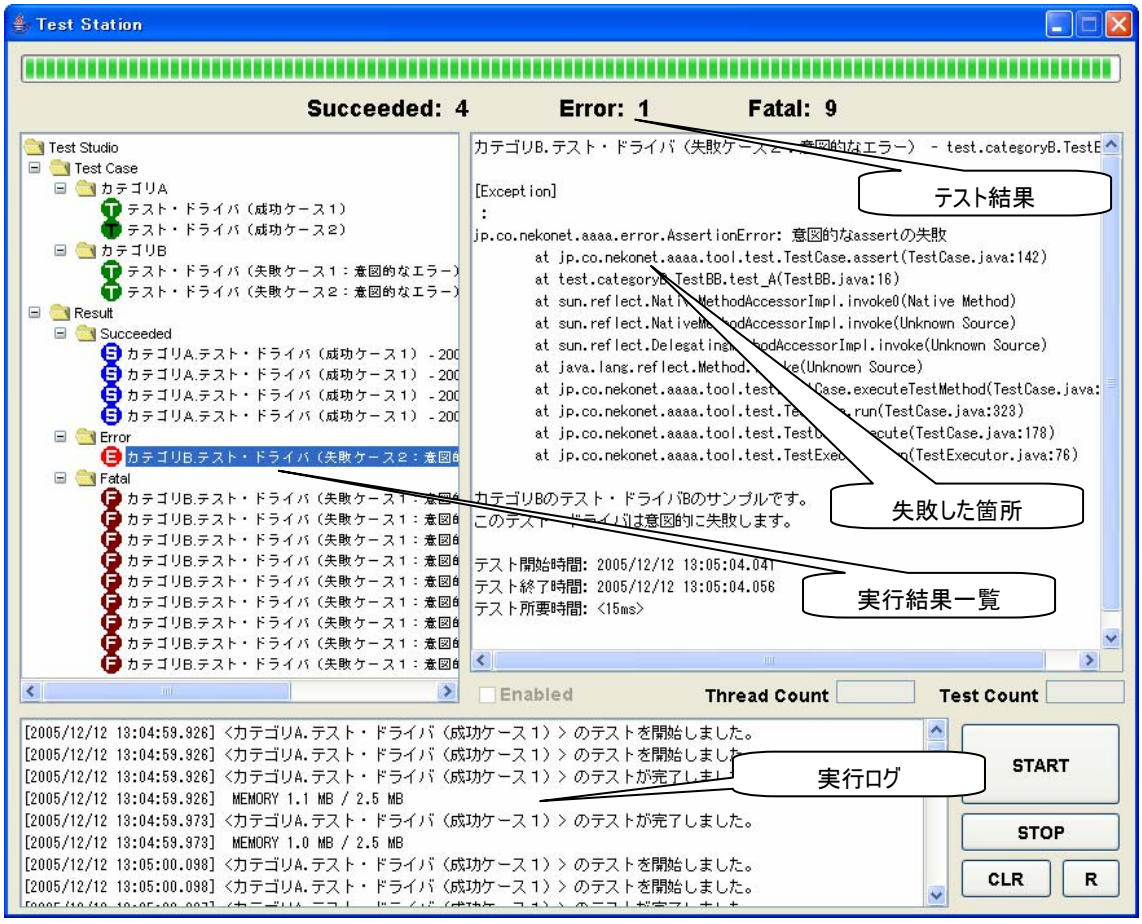

図５　テスト結果の表示画面

(2)　複数のテストプログラムを一括で動作させて検証する方法

この方法では，バッチ・ファイルやシェル・スクリプトを使ってテストを実行する．一括テストの終了時に，**図６**のような成功・失敗の数がログに出力されるため，結果の確認は簡単にできる．テストが失敗した時は，ログに失敗したテストプログラムの情報が出力されているため，それらの情報を調べて原因を調査する．一括テストの段階でテストが失敗する原因は，メモリの枯渇や，ネットワークの異常などの資源や環境に起因することが多い．

```
[java] [Test Studio - Count Report]
[java] TOTAL     = 1324
[java] SUCCEEDED = 1321
[java] FAILED    = 2
[java] FATAL     = 1
```

出力ログの説明

TOTAL	テストプログラム実行回数. (TOTAL = SUCCEEDED + FAILED + FATAL)
SUCCEEDED	成功回数. (FAILED と FATAL が発生せず，テスト･ドライバが成功した回数)
FAILED	失敗回数. (1テスト･ドライバ内で assert が失敗した回数.)
FATAL	例外発生回数. (1テスト･ドライバ内で例外が発生した回数.)

図６　一括テストの結果ログ

図６の一括テストの結果は，耐久性テストと競合状態テストを行った時の結果であるため TOTAL 件数が大きくなっている．

６．評価・結果

今回，テストを自動化することによって，前章で取り上げた問題を解決することができた．また，想定していなかったプラスの効果や，改善していくべき新たなる課題も浮き彫りになった．

６．１　複数アプリケーション・サーバ上でのテスト・コスト削減

複数アプリケーション・サーバ上でのテストを繰り返して実行することが可能になった．テストの自動化により，**図７**のようにアプリケーション・サーバ上でのテスト・コストを削減することができた．

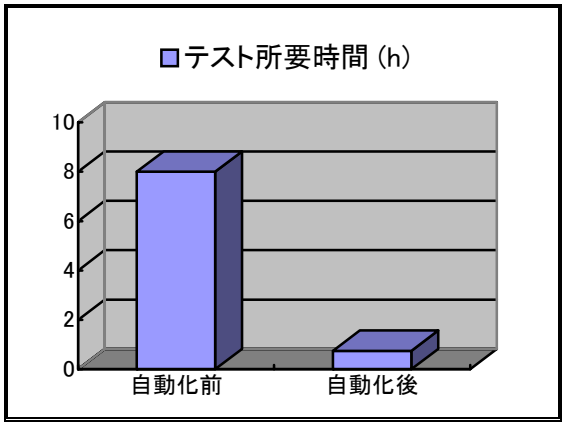

図７　テスト所要時間

図７の自動化前に掛かった時間は，手動で行える範囲のテストを，八時間掛けて行ったという意味である．一方，自動化後では，フレームワークのすべてのモジュールに対して通常のテストを行っている．

自動化後の方が短時間でテストが行えるということだけではなく，自動化後ではより多くのテスト項目のテストを実施することができた．

６．２　自社フレームワーク機能拡張時のリスク低減

フレームワークの品質を維持するための仕組みが確立され，機能を拡張する時に自動化されたテストを実行することで，プログラムの修正ミスを容易に発見することが可能になった．

今回のテストの自動化後に，アプリケーションから頻繁に使用されるデータベース・コネクション生成機能の動作速度を最適化した．その際，自動化されたテストを利用する事で，インターフェースの互換性の確認と，動作の確認を行うことができた．

６．３　マルチスレッドにおける不具合の検出

自社フレームワークのテストの自動化によって，競合状態テストや耐久性テストを行うことが容易になった．これにより，将来，機能拡張した際のマルチスレッドに対する安全性を確認することが容易になった．

今回のテストの自動化後に，一週間ほどフレームワークのテストを連続で実行した．その際，データベースのデッドロックと，マルチスレッドにおける同期処理の問題でテストが失敗した．調査した結果，それらの問題は，今回作成したテスト・ドライバとテスト・スタブに問題があるということが判明した．このように，短時間動作させただけでは発生せず，長時間連続でテストしないと発生しない問題も存在する．

６．４　その他

(1)　自社フレームワークに対する仕様書の多層防衛化

自社フレームワークの動作仕様に対する仕様定義が補強され，より明確化した．テストの自動化前までは，仕様を定義するために UML[①]と JavaDOC[②]による資料は作成していた．この UML と JavaDOC の仕様定義に加え，動作結果を定義したテストプログラムの追加によって，三つの視点から仕様を定義し多層防衛化を行うことができた．

(2)　テストプログラムの作成に掛かるコスト

テストプログラムの作成は，**図８**のようにテスト対象のプログラムのステップ数に対して，テストプログラムには同等（約 0.9 倍）のステップ数が必要になった．つまり，自社フレームワークを開発する際に，テストプログラムの作成まで含めて計画した場合，テストプログラムの作成を含めない場合に比べて約２倍のコストが掛かるということである．

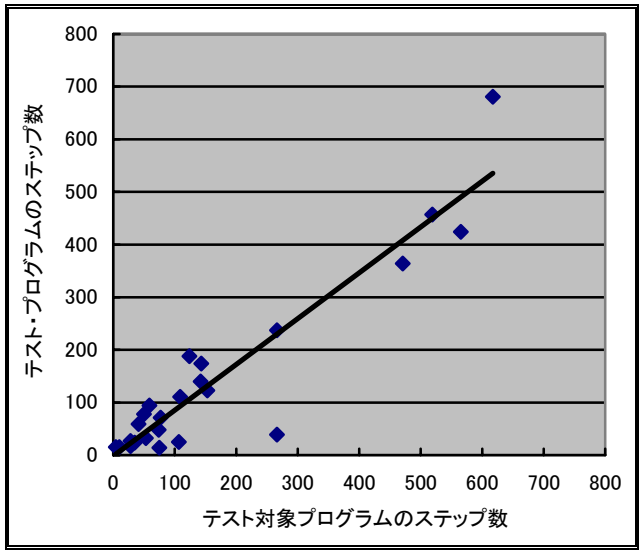

図８　テストプログラムとテスト対象のステップ数の関係

① オブジェクト指向のソフトウェア開発における，プログラム設計書の統一表記法．
② Java のソースファイルから生成された，モジュールのリファレンス・マニュアル．

(3) コードカバレッジ率 100% は重要であるがすべてではない

今回，コードカバレッジ率 100%を達成するように，テストプログラムを作成してきた．しかし，100%の達成が難しい場合や，100%を達成したとしてもテスト対象プログラムの正当性が判断できない場合があった．

コードカバレッジ率 100%の達成が難しい例として，通信例外などの処理がある．自社フレームワークでは，データベース操作制御機能を提供しており，制御プログラムの中で通信例外を考慮した例外処理が含まれている．こういった場合，実際に通信エラーを発生させて例外処理をテストする必要があるが，意図的に例外を発生させることは非常に難しい．

コードカバレッジ率 100%を達成したとしても，テスト対象のプログラムの正当性が判断できない場合の例として，０除算の回避処理がある．０除算に対する回避処理を実装し忘れていた場合，コードカバレッジ率 100%だったとしても，０除算を行ってしまいバグは発生してしまう．

７．今後の課題

７．１　テストプログラムの作成に掛かるコストの削減

テストの自動化によって得られるメリットも大きいが，テストプログラムの作成にコストが掛かるというデメリットがあった．

今後，テストプログラムの作成に掛かるコストをどのようにして削減していくかが課題になると思われる．テストプログラムを作成しやすいモジュールの構造や，作業者のテストプログラム作成作業に対するスキルレベルの向上などを意識しながら，コスト削減について調査を行う必要があると考えている．

７．２　コードカバレッジ率 100%の達成

今回，テストプログラムを作成する際に，コードカバレッジ率 100%を目指してきた．しかし，データベースとの通信時における例外処理などについて，実際に通信エラーを発生させることが困難であったため，テストができなかった．そのため，重要な検証項目であるコードカバレッジ率 100%を達成することができなかった．

そのエラーを発生させることが難しいという問題に対して，モックオブジェクトを利用してテストするという手法がある．モックオブジェクトとは，テスト対象が使用するオブジェクトの振る舞いを偽装するためのプログラムである．この手法を使用すれば，データベース通信時におけるエラーを発生させることが可能になり，すべての例外処理をテストすることが可能になる．

今後は，このモックオブジェクトを利用した手法を適用して，コードカバレッジ率 100%を目指しテストプログラムを補強していきたいと思う．

８．終わりに

今回，自社フレームワークに対してのテストの自動化は，非常に効果が高いことが分かった．テストを自動化することはメリットも多いが，テストプログラムの作成という作業に時間とコストがかかる．

今回，業務システムとは違う種類のモジュールに対してテストを自動化した．自社フレームワークにはユーザー・インターフェースが存在しないことで，テストプログラムの作成に掛かる時間も業務システムより少なかったのではないかと思う．

今後，今回の成果を活かして業務システムにテストの自動化を推進していきたい．そのために，試験的なプロジェクトを立ち上げ，業務システムにテストの自動化を適用した場合のコストや効果について調査を行う必要があると思われる．その試験的なプロジェクトの結果を分析して，自動化によって効果の大きい業務システムを選択しながらテストの自動化を慎重に推進していきたいと思う．

参考文献

［１］STEVE McCONNELL：“CODE COMPLETE -完全なプログラミングを目指して-”，株式会社 ASCII，1994 年.

［２］Kent Beck：“XP エクストリーム・プログラミング入門 – ソフトウェアー開発の究極の手法”，株式会社ピアソン・エデュケーション，2000 年.

［３］Cem Kaner，James Bach，Bret Prettichord：“ソフトウェアテスト 293 の鉄則”，日経 BP 社，2003 年.

［４］Kent Beck：“テスト駆動開発入門”，株式会社ピアソン・エデュケーション，2003 年.

［５］サイバービーンズ株式会社：“JUnit によるテストファースト開発入門”，ソフトバンクパブリッシング株式会社，2004 年.