B-032

ワークフローシステム構築におけるテスト方法の提案

A Test Method for Workflow System

沖宗 賢一†　　鈴木 善昭†
Kenichi Okimune　　Yoshiaki Suzuki

1．はじめに

近年、電子化された文書の承認行為を電子化するワークフローシステムの需要が高まっており、ワークフローを扱うアプリケーションを開発する機会も従来のシステムソフトウェア開発に比べ増えている。

一般にワークフローを用いたアプリケーション開発では、ワークフローに関わるユーザーの権限やフローの経路の種類によってさまざまなパターンを確認する必要がある。

これらの確認すべきパターンは開発するアプリケーションの種類によって千差万別であり、またアプリケーションの規模によっては膨大な数の試験を実施しなければならなくなる。

そこで本論文ではワークフローシステムの構築において低コストで品質を確保するためのテスト方法について提案する。

2．課題

これまでさまざまなワークフローアプリケーションを開発してきたが、出荷後に細々とした不具合が絶えず発生していた。その原因を分析したところ開発プロセスで定められた所定のテストはきちんと行っているものの、特定の状態でしか発生しない現象を開発時のテストフェーズで検出できていないためであることが分かった。

ワークフローシステムは設定された承認者による承認行為の進捗を管理するものである。通常のシステムでは単純な承認だけでも承認行為実施するユーザの権限や所属する組織など、フローを決定する要素が多く存在する。このため各承認案件にはそれぞれ多数の状態が存在することになる。例えばフローの一単位であるタスクのステータスだけをとってみても、「処理可能」、「処理中」、「処理済」などの状態が存在しており、これらがそれぞれの承認者の属性によって振る舞いを変えてしまうことがあるため確認すべき状態の種類は多岐に渡ることになる。

このように試験を実施すべき状態が非常に多く発生する可能性があるため、従来の単純なマトリックス試験方案だけではどうしても試験漏れが発生してしまうことがわかった。

このことを解決するためにはワークフローの状態を加味したテスト方法を考案する必要がある。また単純にすべての状態を加味したテストを行う場合にはテスト工数が増大するため最小限のコストでテストを行う工夫が必要となる。

3．ソフトウェア構成

今回開発したシステムは図1のようになっている。

申請、承認などワークフローシステムに必要な機能はワークフローコアと呼ぶ部分で用意された API を呼び出すことで実現している。ワークフローコア部分はステータスの管理に RDMSを利用している。

† 東芝 IT ソリューション（株）

通常われわれがワークフローシステムと呼ぶ部分は、ワークフローコアの機能を呼び出す上位アプリ部分であり、この部分は Albatross と呼ばれるわれわれが独自に開発し利用しているフレームワーク上で動作することを前提に設計されれている。

本論文では主にワークフローコア部分の動作確認方法について焦点をあてる。

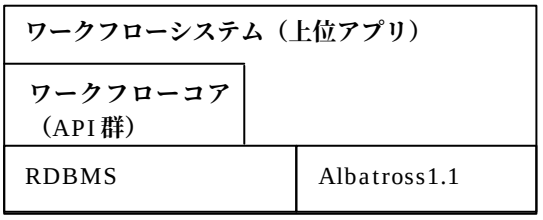

図 1　ソフトウェア構成図

4．テスト方法（API 単位の品質確認）

ワークフローコア部分が用意する個々の API の品質を確認する方法を考える。このときワークフローコア部分の基本設計時に決定した各 API 仕様を個別に確認する必要がある。ところが前述の通りワークフローシステムでは同一試験条件でもそれまでに呼び出された API の動作結果によって確認が必要となる状態が異なってしまう。このため個別の API の確認作業においてテストデータを与える条件をワークフローの状態と存在得るパターンを洗い出し、それぞれの状態およびテスト条件を組み合わせた上で動作を確認することが必要となる。

（例：「差戻し」API）

	状態	本人/代理	タスク種	戻し先
1	処理中	本人	承認	申請
2	一時保存	本人	承認	申請
3	引戻し	本人	承認	申請
4	保留	本人	承認	申請
：	：	：	：	：
35	処理中	代理	承認	1つ前
36	一時保存	代理	承認	1つ前
37	引戻し	代理	承認	1つ前
38	保留	代理	承認	1つ前

同一試験条件でも状態毎に動作を確認

図 2　状態を加味したテストパターン

これらの状態と試験データのパターンに関する組合せは製造開始前の設計段階で検討しテスト仕様書としてまとめた。製造段階ではこれらテスト仕様を満たすようにプログラム作成作業を行った。このように事前にテスト仕様を準備することにより製造担当者による品質のばらつきを防止した。

5. テスト方法（API間の品質確認）

前述の方法では個別の API 毎の品質確認はできるが複数の API を組み合わせた時のみに発生する、複合的なシステム不具合をチェックすることはできない。このような検査項目を漏れなく確認するためにワークフローコア部分が用意するすべての API を組み合わせた結合テストを計画する必要がある。ところが API の組合せを単純に行うと、膨大な数の試験パターンを実施する必要が生じてしまい時間的にもコスト的にも実現が不可能となってしまう。

ここで試験パターンを網羅しつつ試験コストを押さえるにはどのようにすべきかを考えてみる。

通常ある API を呼んだ時に発生するシステム不具合は、その API 呼び出し以前に動作する API の影響を受ける可能性がある。例えば図３のように API-A→API-B→API-C の順に API を呼び出す場合、API-C の動作は API-A、API-B の影響を受ける可能性がある。ところが、API-B→API-A→API-C の順に API を呼び出す場合を考えた場合、API-C の動作に影響を及ぼす可能性があるのは API-A、API-B のいずれかになる。仮に API-A に問題があり、それが API-C にのみ影響がある場合、両方のテストパターンを実行する必要はなく、一方のみを実行すれば十分であることがわかる。

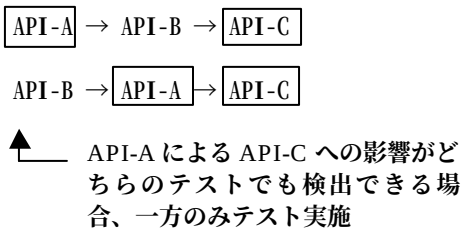

図 3　シナリオテストの不要パターン

開発したワークフローコア部分の提供する API 群の結合試験を実施するにあたり、前述のように必要となるテスト対象を残しつつ不要と考えられるテストパターンをふるい落としていけば、無駄なテスト作業を省きつつ、ワークフローコアが用意した全ての API を確認するテストシナリオを準備することができるため、必要とするテスト項目を網羅しつつ試験工数を押さえることができるようになる。

6. ツール活用によるテスト工数の削減

今回、確認作業を効率的に行うために JUnit というテストツールを活用した[1]。入力パラメータをあらかじめ JUnit で定義しておけばテストは自動実行され、返り値の妥当性もあらかじめ定義した内容で自動チェックされるため、無人テストを実行することが可能になるため試験を効率的に実施できるようになる。

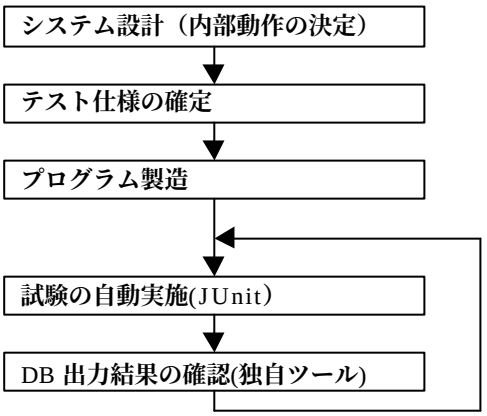

図 4　JUnit による自動テスト

ところがテストシナリオによっては処理結果を返り値だけではなく DB へ反映しなければならない場合がある。このとき JUnit をそのままで利用しただけでは DB への反映結果を自動確認することができない。そこで今回はテスト用に独自に作成したテストツール（JavaServlet）を作成することにより JUnit の利便性を上げることができる。その結果、入力パラメタを与えることによりワークフローコアが提供する API の戻り値および DB への書き込み結果を自動的に確認できるようになったため、前述のシナリオパターンをより効率的に利用できる環境を構築することができる。

7. 結果

ワークフローを扱うシステムを開発する時に問題となるテストパターンの多さに対して、必要なパターンを残しつつ最小限のコストでテストを実施する方法を提供することができた。ツール活用、シナリオテストの不要パターン削除により、テストにかかる時間を当初予定から 50%程度削減することができた。そのため予定期間内で同様のテストを繰り返し実施しすることが可能となったため、普段見逃しがちな些細な不具合項目に対しても入念にチェックをすることができるようになる。

8. おわりに

今回の論文では、開発したアプリケーション全体のうちの主に共通機能にあたるワークフローコアの部分に対する試験方法について述べた。現在、ワークフローコア部分の開発は完了し、ワークフローコアを使用した上位アプリを現在開発しているが、この部分の動作確認においても今回の開発で利用した手法を取り入れテストパターンを絞り込むことができている。

また、今回テスト用に作成したツールは試験効率をあげるだけでなくサンプルプログラムとして API の開発効率を上げるためのツールとして利用できるなど副次的な効果をあげている。

今回の開発で得たノウハウは今後の開発に活かし、安価で高品質なワークフローシステムの提供を行っていきたい。

参考文献
[1] JUnit 説明サイト http://www.junit.org