開発技術と開発プロセス

ライブラリ管理とビルド・リリースプロセス

NSMビジネスユニット
インフラ構築センター
佐 藤 雅 美

1．はじめに

住友信託銀行殿の年金管理システム開発では、プログラム・Webコンテンツ・帳票フォーム等、作成されたファイル総数は約7000ファイル・400万ステップ（コメント含む）にも上る。これらのファイルを各業務チームや個人で管理するには負担が大きく、また本番リリースする物の中に不要なファイルやデグレード*1したプログラムが紛れ込んでしまう等の問題がある。本プロジェクトでは、1～3人のビルドチームという独立したチームを作り、ライブラリ管理から本番リリースまでの作業フローを一括して行うことで、プロジェクト全体の作業の効率化を図り、リリースの精度を保ってきた。

本稿では、ビルドチームで担ってきた作業をもとに、.NETを使用した大規模アプリケーション開発にとって有効なライブラリ管理の手法を紹介する。また、運用を行うにあたって発生しうるトラブルやその回避策についても、あわせて紹介するので、他システムでライブラリ管理を行う際に参考にしていただきたい。

2．ライブラリ管理の概要

ライブラリとは、アプリケーションを動かすために必要な、様々な種類のファイルを一まとめにしたものの呼称である。ライブラリ管理では、それらのファイルを、ソース管理から本番環境への反映まで、一連の流れとしてサポートしている。具体的には、各開発者が作成したプログラムやWebコンテンツなどのソース管理から始まり、保管しているソースからのアプリケーションの組み立て（ビルド）、テスト環境・本番環境サーバーへのアプリケーションの配布（リリース）までを行っている。

3．ソース管理システム

表1は、本プロジェクトで作成したプログラムやWebコンテンツ等、ソース管理システムで管理されているファイルの統計である。この数値から、この開発プロジェクトの規模について、おおよそのイメージを掴むことができると思う。

表1　当プロジェクトで管理しているファイルの統計

管理ファイル	Webコンテンツ	プログラム	SPファイル	定義ファイル
162	463	3665	54	861
帳票フォーム	**設定ファイル**	**マクロツール**	**運用スクリプト**	**合計**
509	332	137	636	**6819**

この章では、このような大量のリソースを、効率よく、混乱を招かないよう運用していくために有効なツールについて紹介する。またその運用を始める上で、開発初期に注意しておくべき項目を挙げておくので、大規模な開発を行う際に参考にして頂きたい。

3.1　VSS製品概要

本プロジェクトでは、ソース管理のために、Microsoft製品のVisual SourceSafe（VSS）を使用している。VSSは、開発作業を行う際、複数の開発者からアクセスされるファイルのリソース管理、差分管理、バージョン管理などを行うツールである。開発者が更新したファイルに“チェック

*1）アプリケーションのリリース時に、古いプログラムが混入してしまうこと。

イン”という登録作業を行うと、VSSのデータベース上にファイルのバックアップが作成され、他のユーザーから利用可能になる。VSS上ではファイルの更新履歴が記録され、必要があれば古いファイルを復元することが可能であり、変更された内容をコードレベルで比較することができる。また、ファイルごとにアクセス権が設定でき、担当者以外が不意のファイル改変を行ってしまうようなトラブルを防ぐことができる。IDE*2として使用していたVisual Studio.NET（VS.NET）*3との親和性が良く、開発者はプログラムを修正しながらVS.NETから直接VSSのデータベースにアクセスできるため、VSSのGUIを立ち上げなくてもチェックインが行える。

3.2 ラベル機能

VSSの特徴として、ラベル機能がある。ラベルとは、VSSにチェックインされた任意のバージョンに名前を付けることによって、いつでも簡単に特定のバージョンのファイルを取り出すことができる機能である。本プロジェクトでは2種類のラベルの運用を行っており、統合テスト環境に毎日リリースしている最新のバージョンには「B+日付（例：B040401）」、本番環境にリリースしているバージョンには「Release」とラベリングしている。こうすることで、テスト未済みの最新プログラムが本番にリリースされてしまうことがないように制御している。

3.3 VSSのトラブルとその回避策

これまで述べてきたように、VSSにはさまざまな機能があり、長期的なプロジェクトで大量のファイルを管理するためには非常に有効なツールである。しかし、実際にプロジェクトで使用している間に、いくつかの問題が上がってきた。

・本番用のReleaseというラベリングを行ったのに、正しいバージョンのファイルが取得できない。
・特定のファイルがチェックインできなくなった。
・特定のファイルの履歴が見られなくなった。

これらの問題の原因はVSSのデータベースの仕様にあり、回避できないようである。VSSのデータベースの実体は、主にプレーンテキストで管理されている。SQL ServerやOracleのような本格的なデータベースとは違い、トランザクション機能がないため、複数の開発者が同時にチェックインやラベリングをしたり、チェックイン途中でVSSをログオフしたりしてしまうと、テキストファイルがロックされたり、書き込み途中でデータが消えてしまうといった問題が時々発生する。ラベリングは、1種類のラベルに対して、1つのテキストファイルで管理されているため、Releaseというラベルを開発者全体でラベリングすると、特に障害が発生しやすい状況になる。そのため、VSSにはデータベースのメンテナンスを行うAnalyzeというツールが用意されている。Analyzeツールを実行すると、データベースの整合性を検証し、問題があれば修復を行う。ただし、このAnalyzeツールにも、長いファイル名（33バイト以上）のファイルは修復できず、逆にファイル破損させてしまうことが報告されている（参照URL1）。この問題はVSSのパッチを適用することで回避できるが、パッチはMicrosoftがプライベートフィックス（一般公開してない）として提供しているもののため、特別なサポートを受ける必要がある。33バイト以内のファイル名であれば問題ないので、初期設計の段階でファイル名にはIDを使用するなどの制限を設けてもよい。ファイル名やフォルダ名が長いと、普段利用するときでもVSSの制約事項に掛かってしまう可能性がある。詳細は、MSDN*4の情報を参考にすると良い（参照URL2）。

参照URL1：『analyze.exe -f で一部の日本語ファイル名が不正に処理される』
http://support.microsoft.com/default.aspx?scid=kb;ja;JP418298
参照URL2：『Visual SourceSafe の名前付け規則と制限事項』
http://www.microsoft.com/japan/msdn/library/default.asp?url=/japan/msdn/library/ja/guides/html/vsgrfssugrnaming_syntax_conventions_and_limitations.asp

4．ビルドプロセス

4.1 日次ビルド作業

ビルドとは、プログラムファイルを、dllやexeといった実行モジュールに変換する作業である。開発者の端末で、

＊2）統合開発環境（Integrated Development Environment）の略。
エディタ、コンパイラ、デバッガなど、プログラミングに必要なツールがそろっている開発環境のこと。
＊3）Microsoft Visual Studio .NETの略で、Microsoftが提供する統合開発環境のこと。
開発言語はVB、C++、C#などが使用できる。
＊4）Microsoft Developer Networkの略。
Microsoftが提供する、開発者向けサポートサービス。同社Webサイト上ではMSDN Onlineが公開されており、様々な開発者向け情報を無償で閲覧できる。

修正したプログラムの単体ビルドを行っているが、他のプログラムとの連携や、整合性の確認のためには、プログラム全体をビルドしなければならない。プログラム全体のビルドには時間がかかり、ビルド処理中の端末では他の作業を行えないため、開発者個人で行うには負荷が高い。この負荷から開発者を解放するため、ビルドチームは朝と夕方の1日2回、ビルド専用端末を用いて、プログラムの全体のビルドを行っている。ビルドされたライブラリは統合テスト環境にリリースされ、開発者は、プログラムの動作について、他のプログラムとの連携まで確認を行うことができる。以下は、日次ビルド作業のワークフローである（図1）。

①開発者は、毎日決められた時間までに、修正したファイルをVSSにチェックイン、ラベリングする。
②VSSから、VSSに登録されているすべてのファイルをビルド専用端末に取得する。このとき、VSS上の最新のファイル（最新版）と、本番リリース用にReleaseとラベリングされたファイル（Release版）の2種類を2台のビルド専用端末にそれぞれ取得する。
③ビルド端末で、プログラム全体のビルドを行う。
④ビルド処理中、最新版とRelease版のファイル比較を行い、差分が出ているファイルについて、開発者に差分一覧をメールで展開する。またビルドエラーが出た場合は、エラーの情報を開発者に知らせ、修正を促す。
⑤ビルドが完了すると、InstallImageと呼ばれているリリース物一式を、バッチ管理サーバーにアップロードする。
⑥リリース作業を開始するため、統合テスト環境が使用できなくなる旨をメールでアナウンスする。
⑦バッチ管理サーバーのInstallImageから、最新版、Release版共に、統合テスト用サーバーにリリースする。
⑧リリース作業が完了し、統合テスト環境が再開したことを開発者にメールでアナウンスする。

このワークフローの中で、最新版とRelease版の差分情報を開発者に展開しているが、これによって、次の確認を行うことができる。

・現在本番環境にリリースされているプログラムが最新のものであるか。
・開発者がReleaseのラベリングをし忘れていないか。
・VSSのデータベースに不整合が起こっていないか。Releaseのラベリングが正常に行われたか。

4.2 ビルドスクリプト

日次のビルド作業は、その手順のほとんどに対して作業の効率化と確実性の向上を図り、スクリプトを用いて自動化されている。ビルドスクリプトに必要な処理の詳細は次の通りである。

1）今回のビルドのバージョン番号、ビルドの種類（最新版・Release版/一次・二次）の入力を要求する。
2）ビルドに必要なサードパーティ製品などのコンポーネントが登録されているか、マシンの環境を確認する。
3）ビルド処理を阻害してしまうリアルタイムウィルスチェックサービスを停止する。
4）VSSからビルドの種類に合ったソースファイルを取得する。
5）ビルドで使用するソースをディスク上にコピーし、物理的な履歴を残す。
6）ビルドが不要な画面コンテンツや帳票フォームなどのファイルをInstallImageの中にコピーする。
7）VS.NETで用意されているDevenvコマンドでソリューション全体をビルドする。
8）ビルド終了後にログファイルを確認し、エラーがあれば修正を促すメッセージを出力する。

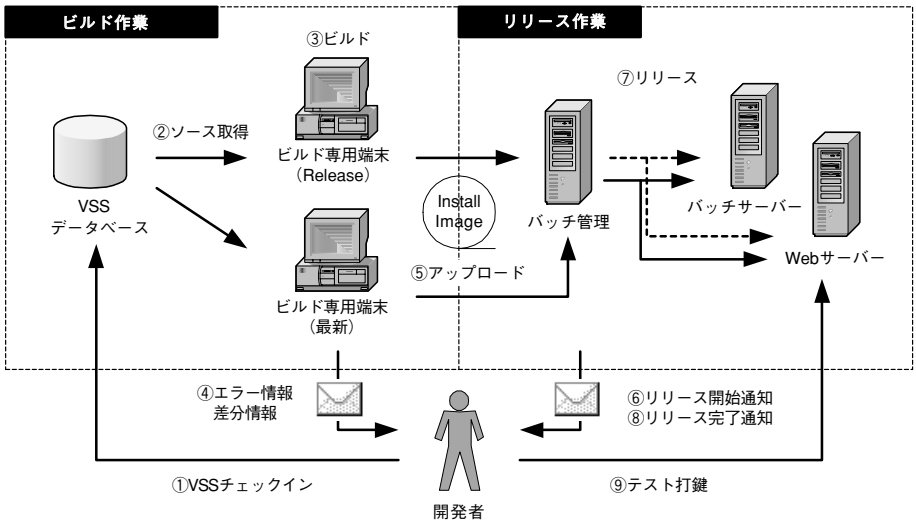

図1　日次作業ワークフロー

9）ビルドで生成された実行ファイル類（exe、dll、pdb）*5をInstallImageの中にコピーする。
10）停止させていたウィルスチェックサービスを再開する。
11）サービス登録する実行ファイルからセットアップファイルを作成し、InstallImageの中にコピーする。
12）ウィルスチェックをしたInstallImageを成果物管理サーバーにアップロードする。

これ以降の処理はリリーススクリプトが行っている。

ここに挙げた手順はビルド１セットの内容である。実際には最新版とRelease版、また一次リリース・二次リリースなどフェーズが重複した時期があり、さらに本番環境への緊急リリース要件が重なり、最大５セットのビルド作業を同時並行で行っていた。しかし、ビルドスクリプト自身に汎用的な処理をさせるような改良を行ってきたため、ビルドマシンは並行度の数だけ必要だが、オペレーションにかかる時間は、並行度に関わらず１人のビルド担当者で２時間足らずの作業量に収まっている。Devenvコマンドが１時間半ほどかかるので、その間ビルド担当者は、開発者へ展開するためのソース差分情報のファイル作成などの作業を並行して行っている。

4.3　大量リソースビルドのトラブルと回避策

前項で述べた全体ビルド作業は、統合テストの初期段階では困難を極めていた。統合テストを開始するまで、開発者は、クラスを単体でビルドするために、１クラス１プロジェクトファイルの構成（図２）でプログラミングを行っていた。この構成はクラス単位にモジュール（dll）を作成するので、他のプログラムの影響を受けることなくテストを行いやすい。また他のクラスを参照したい場合も、明示的にクラス名＝dllファイル名になっているのでわかりやすい。

しかし、統合テストを開始するにあたって、すべてのプロジェクトファイルを１ソリューションにまとめてみたところ、ソリューションを開くだけで、丸２日かかってしまう事が判明した。それぞれのプロジェクトファイルの参照関係が複雑すぎて、依存関係の解決にマシンの処理能力が追いつけなかったのである。またソリューションファイルを開くことに時間がかかるだけでなく、ビルド処理中にメモリー不足などが発生してしまい、全体のビルドができない。事実、本プロジェクトでは全体のビルドができなかったため、一次カットオーバーの直前までサブシステムごとにソリューションを分割し、小さな単位で少しずつdllファイルを作成する代替手段を用いて総合テストまでを行っていた。この方法を用いると、dll間の参照関係がわからないままビルドを行うため、１つのソリューションを何度もビルドしなおさなければならず、ビルドするたびに丸一日もの時間がかかってしまう。さらにVS.NETの不具合が重なり、ビルドコマンドが途中でメモリー不足を起こしても、コマンドが正常に中断されない（画面上ではあたかも処理中のように見える）ため、ビルド担当者は目視でCPUの使用状況を24時間監視していた。

この問題を解決するため、参照関係が似たクラスをグルーピングし、複数クラスを１つのプロジェクトファイル（図２参照）に統合するためのツールを２ヵ月かけて作成することになった。プロジェクトファイルの数を減らし、参照関係の複雑さを解消するのである。このツールの完成によって、約2000近くあったプロジェクトファイルが120ほどに集約され、全体のソリューションファイルを開く時間は２日間から１時間に短縮された。

大規模なアプリケーションを開発する際には、単体テストと統合・総合テストのフェーズでクラス-プロジェクトファイルの管理構成を変更することは、開発効率を上げるために有効である。しかし、開発初期の段階で、ある程度クラスのレイヤ構成を考慮しておかなければ、後の統合作業はほとんど不可能である。

今回行ったプロジェクトファイル統合でも、統合後のプロジェクトファイルが循環参照してしまったり、Webコンテンツと実行ファイルの関係まではツールで補うことがで

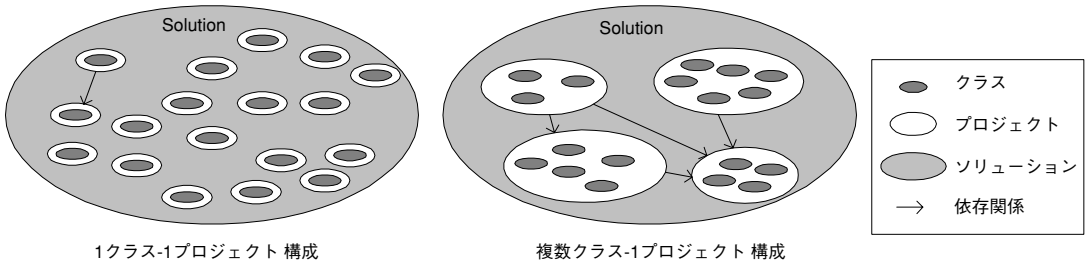

図2　クラスープロジェクト構成

*5）EXEはMS-DOSやWindowsで実行できるプログラムが収められたファイルのこと。EXEファイルは内部で、汎用性の高いプログラムを部品化したDLL（Dynamic Link Libraryの略）を呼び出して処理を行なっている。PDB（Program Data Base）はEXE、DLLごとに作成され、プログラムのデバッグ情報を含んでいる。

きなかったりしたため、最終的には人の手で何百ものファイルを１つひとつ修正しなければならなかった。結果的に統合作業は成功しているが、こういった事態を避けるために、コンポーネントやレイヤ構造の設計には充分に注意が必要である。Microsoftが提唱する設計の指針の内容については、参照URL3から確認できる。

参照URL3：『.NET のアプリケーション アーキテクチャ：アプリケーションとサービスの設計』
http://www.microsoft.com/japan/msdn/net/bda/AppArchCh2.asp

5．リリースプロセス

本プロジェクトで行っているリリースの仕組みには次の特色がある。

・多種多様なテスト内容に対応するため、「面」という概念がある。
・リリースしたものに不具合があった場合、即時復旧するための手段として、リリース物を「世代管理」している。
・処理時間を短縮するため、処理の性質によって実行タイミングを変えている。
・バッチ管理サーバーから一括制御するために、各アプリケーションサーバー上でリモート操作している。

この章では、これらの特色をふまえた上で、リリース処理について紹介する。

5.1　面の概念

開発者が動作確認を行う際、必要に応じてデータベースを更新したり、テーブルのレイアウトを変更したりといった要件が出てくる。しかし、大人数のプロジェクトで共有のテスト環境を使用してテストを行っていると、他の人のテストにとって必要なデータを削除してしまったり、テスト結果を書き換えられてしまったりといった問題も出てきてしまう。少人数で開発を行っていれば、事前にテスト計画について話し合い、調整することで回避できるが、数十人単位でテストを行う場合は難しい。そこで、「面」という概念を採り入れることになった。面とは、同じライブラリを１つのサーバーに幾つもコピーし、アプリケーションが使用するデータベースのコネクションストリングやキューの名前だけ変えて、複数のアプリケーションを同時に立ち上げる考え方である。統合テストのフェーズでは、各サブシステムのチーム単位に面を用意し、他サブのチームの状況を気にすることなくテストできるようになっている。総合テストに至っては、面を複数持つことで、幾つもテストシナリオを同時に流すことができるため、総合テスト期間を短縮することが可能だ。

面の実体は、ライブラリフォルダを複数コピーし、そのフォルダ名に連番を振ったものだ。最後に、それぞれのフォルダに、その面で使用する設定が書かれた設定ファイルを撒いている。Webサーバーには、仮想ディレクトリもしくはサイトを面数分準備しておき、それぞれの面に対応したライブラリフォルダを仮想ディレクトリパスに指定することで、同じアプリケーションを多面で動作させることができる。バッチサーバーではライブラリフォルダにあるバッチ制御の実行ファイルをサービス登録している。同じサービスを１台のサーバーに複数個登録することは不可能なので、複数台あるバッチサーバーをそれぞれの面に割り当ててサービスを登録する、もしくはサービス登録の代わりに、同じ処理を行ってくれるアプリケーションを常時起動させておくことで、多面運用することができる。

ただし、本プロジェクトでは、Webサーバーでいくつものサイトを同時に立ち上げて使用すると、突然Webアプリケーションが動かなくなるといった障害が発生した。その原因はサーバー構成などの環境によるものなのか、製品によるものなのかは不明だが、本プロジェクトでは、経験値として、１つのサーバー上で７サイト以上は立ち上げないようにして、障害を回避している。

5.2　世代管理

本プロジェクトでは、統合環境から本番環境まで、定期的にリリース作業を行っている。ライブラリをリリースするたびに、仕様変更や障害に対応した修正プログラムが稼動し始めるが、そこで新たな障害が発生しうる可能性がある。最悪の状況では、Webアプリケーションが動かず、オンライン障害を引き起こしてしまうことも考えられる。万が一、新たにリリースしたプログラムで障害が発生してしまった場合、その対応策として「プログラムを修正し、再びリリースを行う」ことがまず思い浮かぶが、プログラムを修正するには、障害が発生した場所を特定し対応策を決めなければならないため、復旧までに時間がかかってしまう。特にオンライン障害では、解決までの時間がそのまま障害の大きさに繋がるので、一刻も早い復旧が望まれる。そのため、最も復旧までの時間を短くできる方法として、「今まで動いていたものに戻す」という手段が考えられた。

もちろん、1から全バッチ、Webサーバーにライブラリをリリースし直すとその時間が無駄になってしまうため、一度リリースしたものを、できる限りリリース直後の状態で残してサーバー上に保存しておく。復旧するときは、Webサーバーの仮想ディレクトリパスやサービスに登録されている実行パスを前のライブラリフォルダパスに戻すだけなので、復旧までの時間を短くすることができる。

この仕組みは、古いフォルダをいくつかサーバーに残しておくため、世代管理と呼ばれている。世代管理を行うた

め、過去のものと重複しないよう、ライブラリのフォルダ名にバージョン番号を振っている。保存する世代数を任意に決めることができ、保存されていたフォルダはリリースがあるたびに、順次古いものから削除されていく。詳しい内容は次の5.3.の章で述べるが、世代管理を行うことによってリリース時間を短縮することも可能である。面と世代管理を採り入れると、図3のようなフォルダ群ができる。

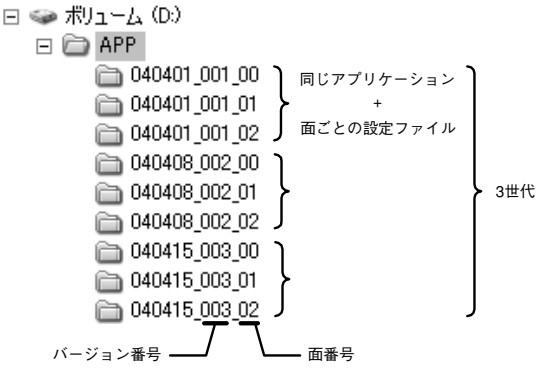

図3　面と世代管理のフォルダ体系

5.3　リリース処理の内容と実行タイミング

リリース処理の内容は、大きく「配布」と「切替」の2つに分かれている。それぞれの処理内容は次の通りである。

●配布部分

1）2世代以上前のライブラリを削除
2）リリース先サーバーのドライブの空き容量の確認
3）新しいライブラリをコピー、面展開
4）配布したライブラリが正しくコピーできているか、ファイルサイズとファイル数を確認

●切替部分

1）バッチ制御のサービス登録（バッチサーバー）
2）IIS[*6]仮想ディレクトリのローカルパス切り替え（Webサーバー）
3）Webサイトに対し、HTTP Requestを投げて、Webアプリケーションの動作確認
4）ASP.NETのワーカプロセスが動作するための空き容量の確認
5）ジョブ管理システム（JP1）のジョブの参照パスの切り替え

リリース処理では、バッチ制御サービスを停止したり、Webサービスを停止したりしなければならないため、リリースを行う時間帯は、アプリケーションがすべて停止していることが条件である。しかし、リリースにかける時間が長くなると、それだけ全体の処理が遅くなるため、リリース時間は可能な限り短くすることが望ましい。そのため、リリース内で行っている処理を稼動中のアプリケーションに影響がある範囲/ない範囲で分割し、「配布部分」では、現在動いているアプリケーションに影響がない処理に留められている。この処理で行われていることは、各アプリケーションサーバーに新しいライブラリのフォルダが作られるだけなので、オンライン中か夜間バッチ中かを問わず、リリース物の準備が整い次第、実行している。

「切替部分」では、バッチ制御サービスを停止したり、Webサイトを停止したりしなければならないため、実行できるタイミングが限られ、実際にはオンラインと夜間バッチの境目で、切替部分を実行している。こうして処理を分割した結果、リリースにかかる時間は1時間だが、全体処理に対しての影響は30分以下に抑えることができる。

5.4　一括制御とリモート操作

また、これらのリリース処理はすべてバッチ管理サーバー上で一括制御されており、それを可能にしているのがRemote Shell Service （RSH）の機能である。バッチ管理サーバーから、リリース処理のスクリプトを実行するよう、RSHのコマンドを各アプリケーションサーバーに対して投げる。各アプリケーションサーバーでは、Remote Shell Serviceというサービスが稼動しており、バッチ管理サーバーからのRSHコマンドを受け取る。コマンドを受け取ったRemote Shell Serviceはバッチ管理サーバーに置いてあるInstallImageのフォルダにアクセスし、その中にあるリリーススクリプトを自分のマシン上で実行する。スクリプトが実行されているのはそれぞれのアプリケーションサーバー上なので、大量ファイルのコピーなどの負荷を分散でき、処理にかかる時間を短縮している。また、細かな権限の考慮が不要でサービスの設定・再起動などの操作も容易に行うことができる。

6．ライブラリ管理システム

開発フェーズでは、プログラムの修正は開発者の判断に任されるが、本番カットオーバー以降の保守フェーズに入ると、プログラムの修正は厳正に管理されなければならない。この章では、確実にリリース物を管理する方法と、作業効率を向上させる手段について説明する。

6.1　ライブラリ管理ワークフロー

ライブラリ管理では、保守フェーズからプログラムの修正は案件として扱われ、仕様変更や障害対応と紐付けて管理される。案件として管理される項目は次の通りである。

・仕様変更・障害番号/案件名

＊6）Internet Information Serverの略で、Microsoftのインターネットサーバーソフトウェア。

・修正責任者/修正担当者
・修正リソース
・本番リリース予定日/実施日

また、これらの案件内容を審査しながらリリースするため、ワークフローも整えた。修正開始前にお客様の承認を得ることで、修正方針、リリース予定の妥当性を確認しながら作業を進める。また、本番リリースを行う前に、修正内容とテスト結果を提出し、再度お客様からの承認を得なければならず、これにより確実に動作確認済みのリソースがリリースされる仕組みになっている。このワークフローは、処理を滞りなく行うため、メールで進められている。ライブラリ管理者は、お客様の承認を確認して、修正のために担当者に対してVSSアクセス権限をつけたり、本番リリースのための準備作業をしたり、適宜対応を行う（図４）。

6.2　ライブラリ管理の省力化

6.1で述べたワークフローでは、申請メールが１通届くごとに、ライブラリ管理者にとって、管理一覧データベースへの反映やアクセス権限の操作、メールの送信など、様々な作業が発生する。その中で、最も手間のかかる項目として、VSSのアクセス権操作が挙げられる。各ユーザーのアカウント管理は、VSSアドミニストレーターというツール上で、VSSのどのアカウントにどのフォルダへのアクセスを許可するか、という設定を行うのだが、VSSに登録されているユーザーやリソースが多い場合、処理に時間がかかってしまい、なかなか作業が捗らない。

そこで、手作業を少しでも減らすために、VSSオートメーション（参照URL4）という、Microsoftが提供するAPIを使用したプログラムを作成した。このAPIはVSSがインストールされている端末上で使用できる。WSH（Windows Script Host）の簡単なスクリプトに埋め込むことで、申請用のワークシートからユーザーアカウントと修正リソースの情報を読み込み、自動的に権限操作を行うことができる。また、申請内容のデータベース反映やメール文書の作成などもすべて自動化し、更なる省力化を実現した。本プロジェクトでは多い日で１日に40件近くもの申請が上がったので、これらの自動化は大変有効であった。

参照URL4：『Visual SourceSafe 6.0 オートメーション』
http://www.microsoft.com/japan/msdn/ssafe/techmat/vss6auto/

７．今回の評価と他プロジェクトへの応用

7.1　今回の評価

冒頭で述べたとおり、ここに書かれたすべての作業は３人以下の少人数で行っている。３人の作業者が必要だったのは、チーム発足当初の運用面が整備されていない時期のみで、この記事を書いている現時点では、作業のすべてを１人で行うことも可能になった。現在では、ライブラリ管理の複雑なワークフローを、開発者にかかる負担を減らしながら、混乱なく運用することができている。これらは、ライブラリ管理を開発作業や運用の片手間に行うのではなく、最初から全体ビルドやリソース管理に専念するためのチームを組み、運用環境を整備してきたことの成果である。

また、そのチームの役割としてリリース機能まで採り入れることで、アプリケーションの仕組みに大きな変更が入っても、開発者をサポートしながら柔軟に対応することができ、確実にリリースを行うことができた。開発フェーズの移行時期などはライブラリ管理全体の作業量が突発的に増えるが、作業の効率化を進めてきた結果として、作業工数を大きく増やすことなく吸収する柔軟性を持ち合わせることに成功している。

7.2　今後の課題

本プロジェクトでのライブラリ管理システムは、開発状況やテスト環境の状況によって出てきた要件に対して、徐々に機能追加されてきた。厳密なライブラリ管理を行い始めたのは、１次カットオーバー直後の保守フェーズからである。そのため、ライブラリ管理として必要な機能を実装することはできているが、開発・テストのフェーズから

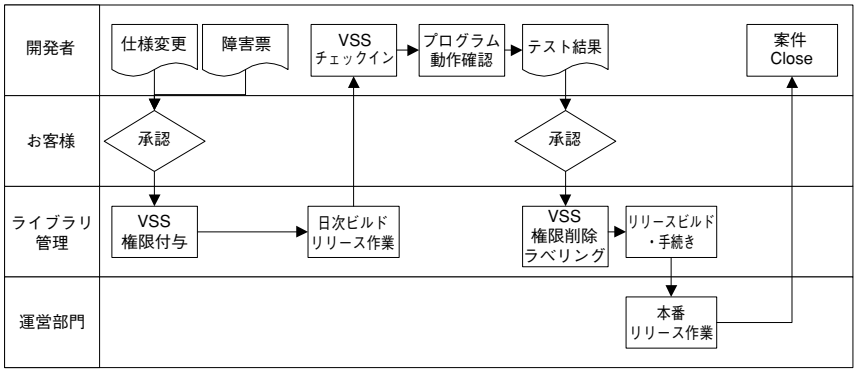

図4　ライブラリ管理ワークフロー

実施されていた障害・仕様変更管理との結びつきが弱い。現在はライブラリ管理の案件一覧を見ながら、人の手によって障害管理表などとの紐付けを行っている状況である。この状況を改善するために、今後プロジェクト全体で管理している情報をデータベース化し、それらの情報を一元管理するためのシステムを構築していかなければならない。実際にそのようなシステムの開発を試みているが、既存情報の入力が追いつかなかったり、ワークフローとシステムの仕様がかみ合わなかったりしたため、まだ実際に運用する段階までたどり着いていない。今後は外部のワークフローツールの利用なども考慮に入れ、各管理システムとの連携を図っていきたい。

7.3 他プロジェクトへの応用

ライブラリ管理は大規模なシステムを開発する際に必須だが、開発のかなり初期の段階からライブラリ管理に対する要件定義や、運用の構想がなされなければ、適切に運用していくことは難しい。その際、考慮するポイントとして、次の項目が挙げられる。

・ソース管理に利用するツールの選定
　ソース管理ツールを使用するか
　最新版・Release版のような考慮が必要か
・ビルドサイクル
　サイクルが短ければ、ビルドが間に合わない
　サイクルが長ければ、テスト進捗が遅れる
・テスト環境の運用体系
　面・世代管理の考慮などは必要か
・ライブラリ管理とプログラム修正案件の関連性
　仕様変更や障害とソース管理の紐付けが必要か
　その他ドキュメント管理なども紐付けを行うか

これらの要件が決定する時期が早いほど、運用体制を整えやすく、自動化できる部分も見出せるので、ライブラリ管理、ひいてはプロジェクト全体の成功に大きく貢献することができる。

そして、前述したように専門のチームを作って運用することになった場合、発足当初は、担当者の１人に、開発言語のアーキテクチャに詳しい者を招き入れることが望ましい。作業の効率化などは後追いで覚えた知識でこなせるが、初期に発生するビルドのトラブル対応や開発者のサポートを行う際には、ある程度の知識がなければ無用な混乱を招くことになる。これは、本プロジェクトでライブラリ管理を始めた初期段階に、担当者のVS.NETの利用経験が十分でなかったためビルド障害の解決に時間がかかり、少なからずプロジェクト全体のスケジュールを遅れさせてしまったことからの反省でもある。

このチームが見舞われる障害は、常に全体へも影響するので、そのことを念頭に担当者を選定しなければならない。ただし、開発作業とは別の、「運用」という視点も必要なため、開発作業との掛け持ちは避けた方が良い。

8．終わりに

本稿で述べたような管理システムは、小規模の開発案件であれば必要性が低いだろう。しかし、規模の大小に関わらず、「管理をする」ということは、製品の品質を保ち、開発環境をより良く運用していくことに繋がる。また、今回、ライブラリ管理の運用を行ってきたことで、いったんワークフローを整えてしまえば人手はかからず、同じような.NETプロジェクトへの流用も可能であることがわかった。小さなプロジェクトでもライブラリ管理を採り入れておくと、新たなプロジェクトの立ち上げ時の導入が容易になる。したがって小さなプロジェクトであってもライブラリ管理プロセス・体制を固めることは有効であると考える。本稿がよりよいプロジェクトの推進に寄与することができれば幸いである。