課題図書レポート『継続的インテグレーション入門』

2023/2/28

本社システム課　町田

■本書を選定した理由

CIの意義やプラクティスに対する理解を深めるため。

■学んだこと(まとめ)

●用語の定義

* ビルド…ソフトウェアの生成、テスト、インスペクション、デプロイを行う一連の活動。
* インテグレーション…個々のソースコードが全体として機能するかを確認するため、1つに統合すること。

●CIの構成要素

　本書では、CIの構成要素として以下の5つが紹介されていた。これらの一連の処理を特殊な操作なく、自動もしくはコマンド1つで実行できるようにすることがCIの肝である。

* ソースコードのコンパイル
* データベースのインテグレーション
* テストの実施
* インスペクションの実施
* ソフトウェアのデプロイ

●CIの価値

　CIは以下の価値をもたらす。

* リスクを軽減する
  + 欠陥の検出と修正を迅速に行えるようになることでリスクの低減につながる。
* 繰り返しが多い手作業を削減する
* いつでも、どの環境にもデプロイできるソフトウェアを生成する
* プロジェクトの可視性を改善する
  + プロジェクトの様々な傾向の把握、効果的な判断を行うきっかけになる。
* ソフトウェア製品に対する開発チームの自信を深める

●CIでのプラクティス

　CIの中心的なプラクティスを以下に列挙する。

* 頻繁にコードをコミットする
  + インテグレーションを早めに、かつ頻繁に行うことがCIの原則である。そのため、開発者は頻繁にコードをコミットする必要がある。
* ビルドが失敗したコードをコミットしない
* 失敗したビルドをすみやかに修復する
* 開発者テストを自動化する
  + NUnitなどのフレームワークを使用してテストを作成することで、テストの実行を自動化できる。
* すべてのテストとインスペクションを合格させる
* プライベートビルドを実行する
  + 自身のローカル環境でインテグレーションビルドと同様のビルドを行い、成功することを確認してから変更ファイルをコミットすることで、CIサーバ上のインテグレーションビルドが失敗する危険性を小さくできる。
* ビルドが失敗したコードを取得しない

●CIによるリスクの軽減

　CIが軽減する領域とその対策を以下に列挙する。

* デプロイできないソフトウェア
  + デプロイ可能なソフトウェアをいつでもビルドできるようCIシステムを使用する。バージョン管理リポジトリからすべてのソフトウェア資産を取得する、反復可能なビルドプロセスを作り上げる。
  + ソフトウェア資産とは、ソフトウェアの開発・運用・保守に必要なすべての電子データを指す。
* 欠陥検出の遅れ
  + ソフトウェアに変更が加えられるたびに、開発者テストを含めたビルドを実行することで早期に欠陥を検出できる。
* プロジェクトの「見える化」不足
  + 定期的にビルドを実行することで、ソフトウェアの状態をいつでも把握できるようになる。
* 低品質なソフトウェア
  + ソフトウェアに変更が加えられるたびに、テストやインスペクションを実行する。そこで複雑度、コードの重複、コードの網羅率などの要素を確認することで、コードに混入した欠陥の検出がしやすくなる。

●変更を起点としたビルドの実行

　ビルドの実行に際して押さえておくべきプラクティスを以下に列挙する。

* ビルドを自動化する
  + IDEから独立したビルドスクリプトを作成する。ビルドツールとしてはjava用のAnt、.Net用のNAntがある(実務ではMSBuildを使用)。
* コマンド1つでビルドを実行する
  + 最新ソースコードの取得からビルド全体の実行までを、1つのコマンドで起動できなければならない(必要なツールがあることが前提)。そのために、すべてのソフトウェア資産はバージョン管理リポジトリに入れておく必要がある。
* ビルドスクリプトをIDEから分離する
  + 自動化されたビルドを実行するのに、IDEが必要とならないようにするべきである。理由は以下の2つである。
    - 開発者全員が同じIDEを使用していると限らないから。
    - 開発者が使うものと同じビルドスクリプトがCIサーバで使用可能である必要があるから。
* ソフトウェア資産を集中化する
  + 依存関係の問題を減らすために、すべてのソフトウェア資産をバージョン管理リポジトリで一元管理する。これにより、異なるマシンに移した際にビルドが失敗する可能性を減らせる。(「すべて」とは具体的にどの範囲を指すのかは各プロジェクトで決定する必要がある。)
* 一貫したディレクトリ構造を作成する
  + すべてのソフトウェア資産を取得できるようディレクトリの構造を一貫性のあるものにする。(例えば、実装に関するファイルは実装ディレクトリ配下に配置する、など)
* 失敗しやすいビルドプロセスから始める
  + フィードバックが早いほど、問題の修正も早く行えるため、失敗する可能性の高い順にビルド作業を実施する。
* 複数環境へのデプロイに対応する
  + 自動化された同じビルドを、開発環境、テスト環境、QA環境、本番環境など複数の環境でビルド作業を実施する。
* インテグレーションマシンを使用する
  + ビルドを実施するために専用のマシンを1台使用する。インテグレーションを実施する環境では、過去のビルド成果物に影響を受けないよう環境を初期化する必要がある。
* CIサーバを使う
  + CruiseControlなどのCIサーバを使う。CIサーバは自動的にバージョン管理リポジトリの変更を監視し、インテグレーションビルドを別のマシン上で実行させる。

●継続的データベースインテグレーション

　継続的データベースインテグレーション(CDBI)とは、プロジェクトのバージョン管理リポジトリに対する変更に伴い、開発に利用しているデータベースとテストデータを再構築するプロセスである。

　以下に、CDBIのプラクティスを列挙する。

* データベースインテグレーションを自動化する
  + データベースに対するアクティビティ(データベースのドロップ、作成、データの登録、カラムの属性と制約の変更、ストアドプロシージャの変更など)を自動化するスクリプトを作成する。
* ローカル環境でデータベースサンドボックスを使う
  + 開発者は全員、SQLスクリプトで作成できる自分専用のデータベースを使うべきである。これにより、他の開発者に影響を与えることなくデータベースを変更できる。
* バージョン管理リポジトリを使ってデータベース資産を共有する
  + DDLとDMLをバージョン管理リポジトリへコミットする。これにより、データベースの作成とテストデータ作成の際に、他の開発者も同じスクリプトを利用できる。
* 開発者にデータベースを変更する権限を与える
  + 開発者にDDLとDMLを変更できる権限を与え、バージョン管理リポジトリへコミットさせる。こうすることで、DBAへのデータベースに対する作業の集中を避けることができる。

●継続的テスト

* 単体テストを自動化する
  + NUnitのようなテストフレームワークを使用することが推奨される。単体テストはファイルやデータベースなどの外部への依存性を持たない。
* 結合テストを自動化する
  + NUnitのようなテストフレームワークを使用することが推奨される。単体テストとは異なり、外部との依存関係を持つことがある。また、オブジェクトの連携が必要になるため、単体テストより実行時間が長くなる。
* システムテストを自動化する
  + 完全なシステムを稼働させてテストを行うため、実行時間がかかる。通常、複数のコンポーネントを使用してテストが行われる。
* 機能テストを自動化する
  + ユーザーの観点からアプリケーションの機能性を確認する。Selenium(Webアプリケーション用)やAbbot(GUIアプリケーション用)を利用することで自動化できる。このテストが最も実行時間がかかる。
* 実行時間が短いテストを先に実行する
  + 単体テストは実行時間が最も短いため、最初に実行させるようにする。
* 1つのテストケースに1つのアサーション
  + 1つのテストケースに1つのアサーションを限定することで、テスト失敗時の原因を効率よく特定できるようにする。

●継続的インスペクション

　インスペクションとは、予め決められた標準にしたがってコードを分析するアクティビティを指す。(コーディング規約に従っているか、コードに重複がないか、など)

　このインスペクションの自動化に関するプラクティスを以下に列挙する。

* コードの複雑度を下げる
  + 高い複雑度を持つコードを特定するために自動インスペクションツールを利用し、ソースコートのサイクロマティック複雑度を下げる。このインスペクションツールを自動ビルドの中で実行する。
* 継続的なデザインレビューの実施
  + アーキテクチャの結合度をレポートするツール(NDependなど)と取り入れることで、アーキテクチャのもろさにつながる可能性のあるパッケージ/アセンブリを発見しやすくなる。
* コードの複製を減らす
  + コードの重複が多い部分を特定するツールを使用してソースコード中の重複コードの量を減らす。
* コード網羅率を評価する
  + NCoverなどのツールを利用し命令網羅率と分岐網羅率を測定する。測定結果を、テストが不十分な部分の特定に利用する。

●継続的デプロイ

　継続的デプロイとは、利用可能な状態のソフトウェアをいつでもどこでも最小限の労力でリリースすることを可能にすることを指す。

　以下に、継続的デプロイのプラクティスを列挙する。

* 動作するソフトウェアを常にリリースできるようにする
  + コンパイル、テスト、インスペクション、パッケージ、そしてデプロイを含むビルドプロセスを完全に自動化し実行することで、常に動作するソフトウェアをリリースできる。
* リポジトリの資産にラベル付けをする
  + バージョン管理リポジトリにあるファイルにラベル付けをする。これにより、特定のファイルのまとまりが同じグループに属していることを明確化し、ファイルの特定と追跡が容易になる。
* 個々のビルドにラベルをつける
  + バージョン管理リポジトリで、ビルド結果のバイナリにラベル付けをする。これにより、特定の機能、欠陥、要求などを関連付けられる。
* すべてのテストを実行する
  + 単体テスト、結合テスト、システムテスト、機能テストを含むすべてのテストを実行し、すべてが成功することを確認する必要がある。
* リリースのロールバックを可能にする
  + 問題の発生に備えて、バージョン管理リポジトリにコミットされるべきではなかった変更をロールバックできるようにしておく。

■重要だと思ったこと(業務で活かすこと)

本書のほとんどがソフトウェアを運用する上で重要だと思いましたが、特に継続的テストの重要性を実感しました。テストを自動で実行しそのフィードバックをすぐに得られる環境があれば、プロジェクトをかなり円滑に進められるのだろうと思います。

　現在の歩掛チームでは、積算チェッカーでGaia10とGaia Cloudの金額を合わせるということを行っていますが、積算チェッカーのソースが古くなっており、チェック以前に自動積算で失敗しているケースが少なくないです。積算チェッカーの自動積算実行部に対するテストコードが実装されておりCIが定期的にそのテストを自動で実行していれば、積算チェッカーの自動積算がなぜ失敗しているのかを調査する時間を省けたのではないかと感じています。プロジェクトの計画時点(この時点に限らずですが)でCIに任せられることはないか、といった視点を今後持つよう心がけます。