課題図書レポート『継続的インテグレーション入門』

2023/2/26

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サーバー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、開あhつしゃ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
* 失敗しやすいビルドプロセスから始める
* 複数環境へのデプロイに対応する
* インテグレーションマシンを使用する
* CIサーバを使う
* 手動インテグレーションビルドを実行する
* 短時間でビルドを実行する
* 段階的にビルドを実行する

●境界

　境界とは、例えば自分たちが実装するコードの領域とサードパーティのコードの領域などが該当する。サードーパーティのコードをビジネスロジックに分散的に実装すると、サードパーティコードの変更の影響を広範囲に受けることになる。サードーパーティ用のラッパークラスを用意したり、アダプタを用意するなどして境界を設けることで、サードーパーティの変更による影響を最小化すべきである。

●単体テスト

* TDD三原則
  + 失敗する探偵テストのコードを書く前に、製品のコードを書いてはならない。
  + コンパイルが通り、適切に失敗する単体テストができるまでは、次の単体テストを書いてはならない。
  + 現在失敗している単体テストが通るまで、次の製品コードを書いてはならない。
* F.I.R.S.T
  + 高速である(Fast)
    - テストは高速である必要がある。なぜなら、TDDではテストを頻繁に実行するためである。
  + 独立している(Independent)
    - テストは互いに関連すべきではない。なぜなら、あるテストの失敗が後続のテストの失敗につながり、テスト後半部の問題を隠してしまうためである。
  + 再現性がある(Repeatable)
    - テストはどんな環境でも再現可能である必要がある。なぜなら、動かあに環境があるということは、テストが失敗する言い訳をいつも抱えているということを意味するためである。
  + 自己検証可能(Self-Validating)
    - テストは成功か失敗かのどちらかを出力すべきである。なぜなら、テストが自己検証可能でないと、成功・失敗の判定が属人的になったり、長時間を要する手作業による検証が必要になる可能性があるためである。
  + 適時性がある(Timely)
    - テストは必要な時にすぐ書ける必要がある。なぜなら、製品コードの実装の後にテストを書こうとすると、テストが困難な設計になってしまっている可能性があるためである。

●クラス

　クラスは小さくすべきである。大きいクラスは、責務が多かったり、依存関係が多いなど、変更に弱くなる理由を持っている可能性がある。

* 単一責務の原則(SRP)
  + クラスやモジュールは変更の原因となるものが1つでなければならない。
* 凝集性
  + あるメソッドが操作するインスタンス変数が多ければ多いほど、凝集性が高いといえる。

●システム

　システムを使うことと、構築することを分離すべきである。

* mainの分離
  + 構築に関するすべての局面を、mainあるいはmainと名付けられたモジュール(いずれも開始処理)へ移動し、システム内の残りの部分は、すべてのオブジェクトが適切に生成され、関連付けられているという前提のもとに設計する手法。
* ファクトリ
  + 特定のオブジェクトを生成するタイミングをアプリケーションが制御することを可能とさせつつ、生成に関する詳細をアプリケーションコードから分離する手法である。
* 依存性注入
  + オブジェクトの利用と生成を分離する手法である。
  + 多くの場合は、セッタメソッドあるいはコンストラクタの引数を通して依存性を解決する。

●優れた設計を生み出す4つの原則

* 全テストを実行する
  + 設計作業を通して、意図通りに動作するシステムを生成する必要がある。そのためには、システムが意図通りに動作しているかを検証できるテストが必要である。
* 重複がない
  + 重複は、余計な作業、余分な危険、不必要な複雑さをもたらす。同じあるいは似ている処理を関数にくくりだす、多態性を利用するなど、重複の排除に尽力すべきである。
* プログラマの意図が表現されている
  + 変更を行う際にバグを埋め込む可能性を最小化するには、システムの動作を理解可能とすることが重要である。そのため、コードには書き手の意図が明快に表現されている必要がある。
* クラスとメソッドを最小化する
  + 必要最低限のクラスとメソッドを用意するよう尽力すべきである。ただし、この原則は他の原則よりも優先度は低い。

■重要だと思ったこと(業務で活かすこと)

　ソフトウェア開発において、関心事の分離と重複の排除が重要なのだと思いました。

　関心事の分離に関して、サードパーティコードとの境界を明確にすることや単一責務の原則など、関心事を局所化することにより、修正箇所を最小化する営みが変更に強いシステムの構築につながるのだと思いました。

　重複の排除が重要であることは先日のハッカソンにおいても実感しました。ほとんど同じような処理が分散して実装されていると、その処理に変更があった際に修正漏れが発生しやすくなると思います。そうした際には、メソッドに処理をくくりだすことで修正箇所を局所化できると思います。また、オブジェクト指向の三大要素の一つである多態性を利用して重複を排除することも重要だと思います。

　今後の業務における実装では、特に以下のことを意識します。

* 変数や関数の命名
  + 他人が読んで意図が伝わる名前や処理の流れになっているか
* 重複
  + 実装した内容が既存の実装と重複する部分がないか