CI パイプラインでのテスト戦略とその実現方法

#stac2020

髙市 智章 (Tomoaki Takaichi)

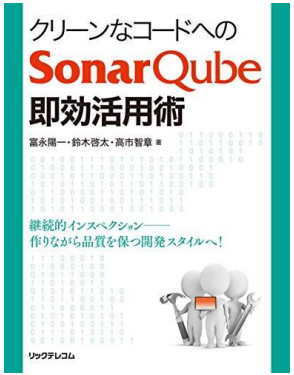
Dec, 5, 2020 ソフトウェアテスト自動化カンファレンス2020

自己紹介

@Takaichi00

tomoaki.takaichi.5

- 髙市 智章（タカイチ トモアキ）
- Java / Node でのシステム開発
- CI / CD
- Container / k8s
- アジャイル開発実践



共著: クリーンなコードへの SonarQube即効活用術

http://u0u0.net/RSvx

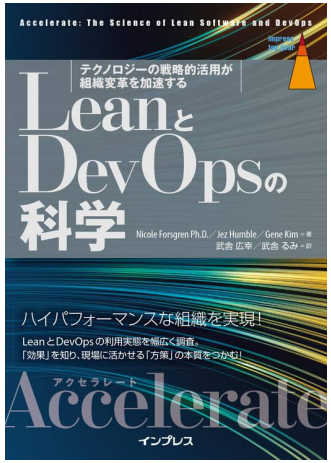
本日お話すること

- なぜ CI / CD /デプロイメントパイプラインが必要なのか
- デプロイメントパイプライントの中身
- デプロイメントパイプラインで実施するテスト戦略

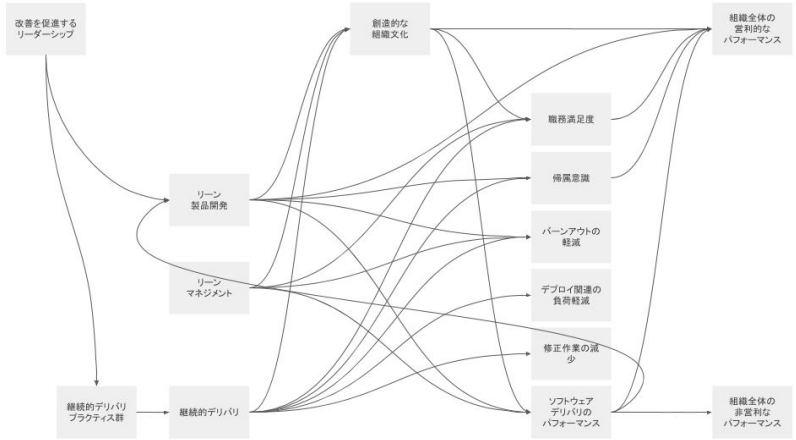
CI / CD の必要性

なぜ CI / CD を行う必要があるか

- 「LeanとDevOpsの科学」では、組織のパフォーマンス (収益性 / 市場専有率 / 生産性) は、「ソフトウェアデリバリのパフォーマンス」が予測要因の一つとされている



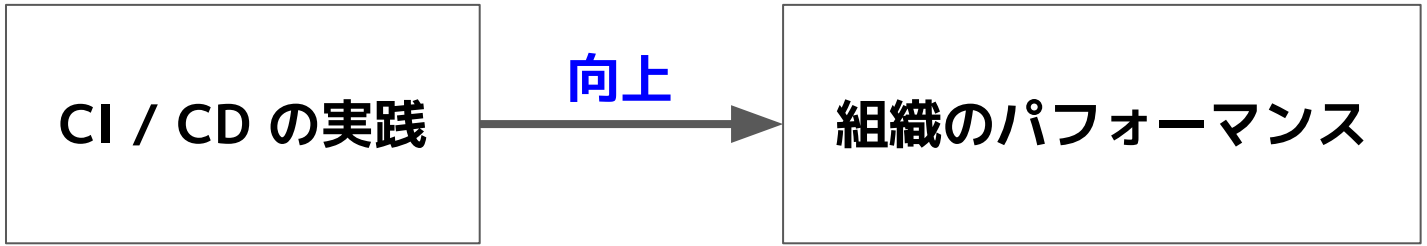
出典: https://img.ips.co.jp/ij/18/1118101029/1118101029-520x.jpg



出典: https://d2l930y2yx77uc.cloudfront.net/production/uploads/images/17728222/picture_pc_7ad9d39bff46bb8813d1c7c8812fa5c9.png

なぜ CI / CD を行う必要があるか

- 「ソフトウェアデリバリのパフォーマンス」を統計的に優位な形で改善できる24のケイパビリティの中に、「継続的インテグレーション (CI) の実装」「継続的デリバリ (CD) の実践」が含まれている
- 組織のパフォーマンス(収益性 / 市場専有率 / 生産性)を向上させるために、CI / CD の実践は効果的である



CI / CD を実践しないとどうなるか

- CI / CD を実践していないプロジェクトでは以下のような問題が往々にして起こりがち
  - 何日も main ブランチにマージされずに機能開発が並行して行われ、リリース間際にマージ作業。しかしコンフリクトが多発し、マージしてもシステムが動作しないで遅延。
  - リリース期限ギリギリに本番環境にデプロイ。しかし本番環境特有の設定値が考慮されておらず、外部システムと連携できなかったり、パフォーマンスに問題が生じて遅延。

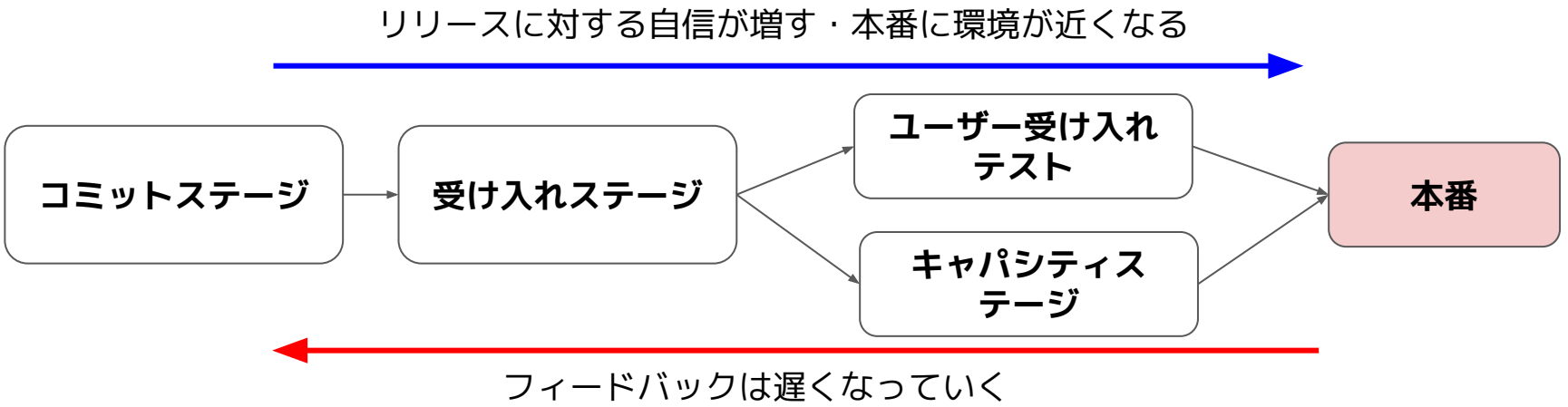
CI / CD のメリットは自動化だけではない

- 継続的にマージし、CI を実施することで常に動作するソフトウェアを維持する
  - トランクベース開発と組み合わせて実施することでより効果を発揮する
- 継続的にデプロイを行い、リリースによるリスクを早期に発見し対処する
- CI / CD を含めた、ソフトウェアをバージョン管理にチェックインしてからユーザーの手に渡すまでのプロセスをデプロイメントパイプラインと呼ぶ

デプロイメントパイプラインの流れ

デプロイメントパイプラインのステージ

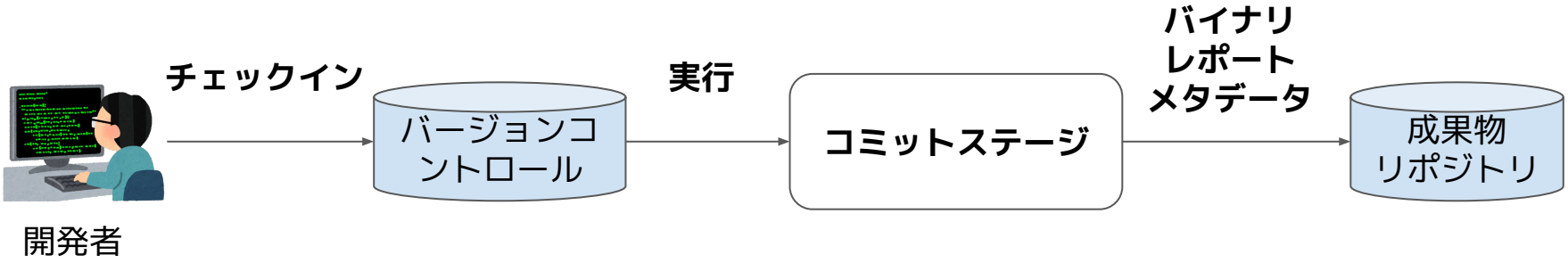
- 「継続的デリバリー」の本では、デプロイメントパイプラインはいくつかのステージに分割されると記載されている
- ステージが進むにつれてリリースの自信が増し、環境がどんどん本番に近づく一方、フィードバックは遅くなる



コミットステージ

コミットステージの概要

- コミットステージは、開発者がチェックインしたタイミングではじめに実行されるステージ
- 本番にふさわしくないビルドを排除し、**そもそもアプリケーションが壊れていないか**ということを素早く知ることが目的



コミットステージの内容

- コミットステージでは以下のようなことを実施する

1. 「コミットステージテスト」を実行する
2. ソースコードの解析を行い、健全性をチェックする
3. ビルドを実行する
4. すべて問題なければ、生成されたリリース候補となる生成バイナリをアップロードする

コミットステージテスト

- 素早く実行できるように最適化されたテスト一式のこと
  - ほとんどはユニットテストになる (xUnit 系)
  - ユニットテスト以外のテストも少し選び出し、このステージに含める (コンポーネントテストなど)
- コミットステージが通れば、アプリケーションが実際に動くのだと強い自信を持って言えるようにする

ソースコードの解析

- SonarQube などの静的解析ツールを実行し、コードカバレッジ、Bug の恐れがあるコード、循環的複雑度などの統計情報を計測する
- **閾値を超えた場合はパイプラインを失敗させる**

項目	失敗させる設定例
コードカバレッジ	70%未満
Bugs の深刻度 (Severity) が Major 以上の項目	1個以上
コードの重複率	5%以上

脆弱性チェック

- 利用している商用ソフトウェアや OSS に脆弱性がないかを検査し、脆弱性があるものの利用を検知したらパイプラインを失敗させる
- SourceClear のようなツールを利用する

リリース候補となるバイナリを生成する

- コミットステージのすべてのステップが成功したときのみ、今後の受け入れステージや本番環境で利用するバイナリを成果物リポジトリ (Artifactory など) にアップロードする
- リリース候補となるバイナリを生成するのはコミットステージの最後ただ一度きりにする
  - デプロイ環境ごとにビルドはしない
  - バイナリの再生成の際に何らかの変更が紛れ込んでしまうというリスク
  - 再コンパイルに時間がかかりフィードバックが遅くなる

コミットステージのプラクティス

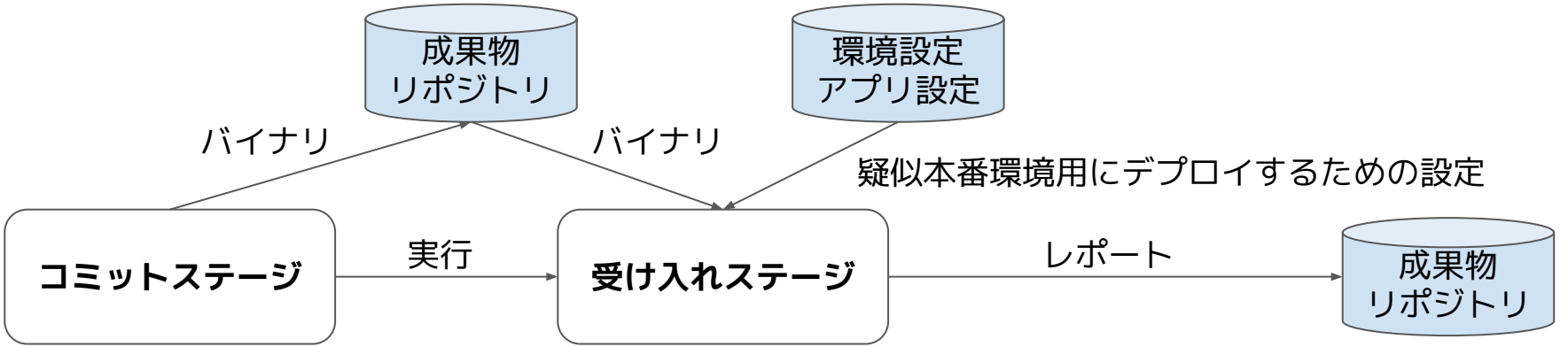
- CI はツールでなくプラクティスである。以下のような規律をチームで守ることで CI は実現される

1. 定期的に main ブランチにチェックインをする (最低でも1日2回)
2. ビルドが壊れているときはチェックインしない
3. コミットテストが通るまで次の作業を始めない
    a. 変更の記憶があるうちに素早くエラーを修正したい
4. テストが失敗してもビルドは続ける
    a. テスト以外にも失敗要因はあるかもしれない
5. 5分以内で終わるようにする (10分以上かかってはいけない)

受け入れステージ

受け入れステージの概要

- コミットステージで生成されたバイナリを**擬似本番環境で実行**させ、顧客の期待する価値をシステムがデリバリーできていることを検証する



受け入れステージの目的

- コミットステージテストでは検証できないような、ビジネス視点での検証を本番とほぼ同じように動いているアプリケーションに対して行うことが目的

1. ユーザーの求める価値を提供しているか?
2. 環境や設定ファイルに起因する問題はないか?
3. 新しい変更によって既存のふるまいにバグが生じていないか?

受け入れステージのテスト

- 受け入れステージのテストは、受け入れ基準に基づき、エンドユーザーにとって価値があるテストでなければならない
- Cucumber のような Garkin 記法という自然言語を用いてテストケースを記載できるツールを利用することもある

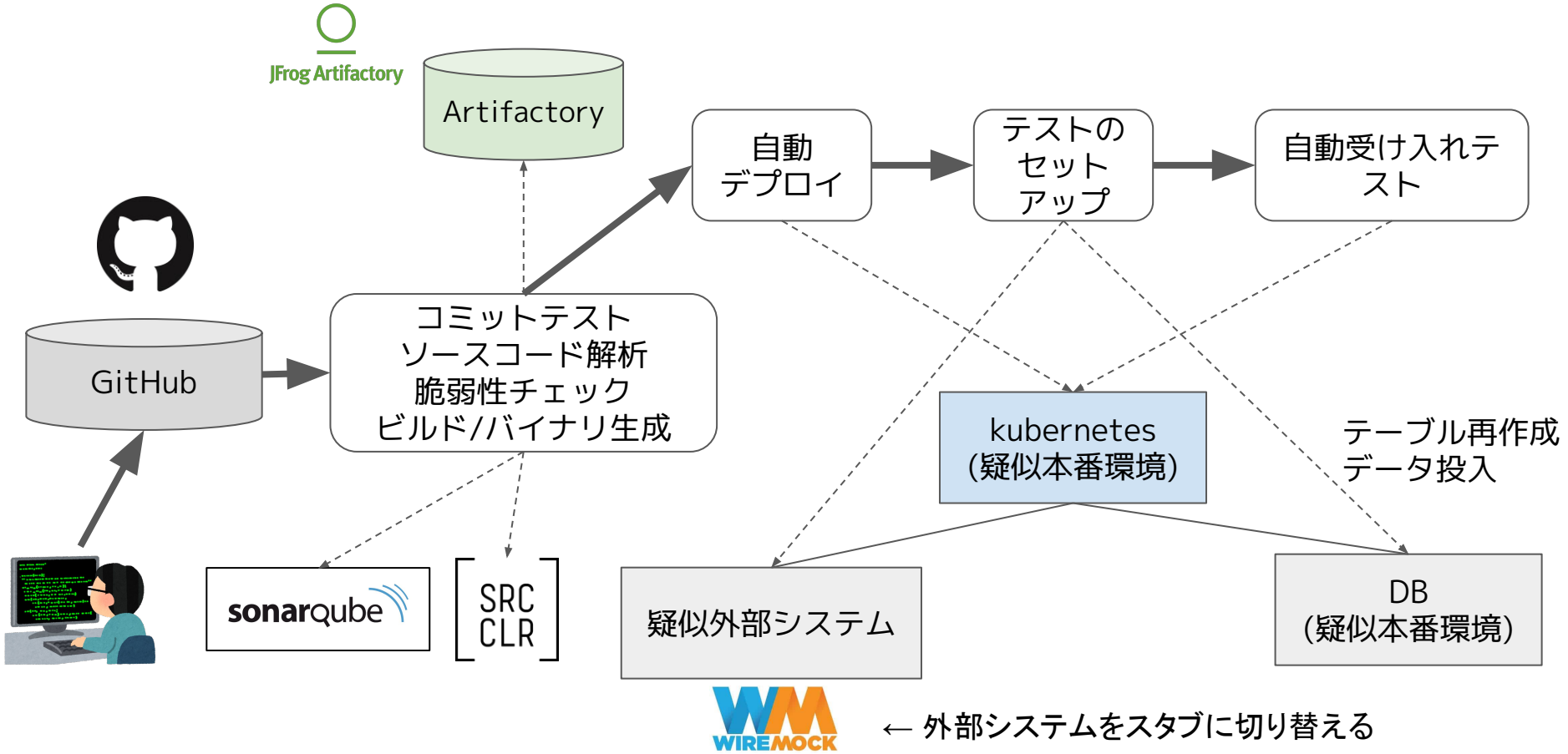
```
# language: ja

@Developing
機能: 商品を購入することができる

  背景: ユーザーがログインしている
    前提名前が"たろう"であるユーザーがログインしている

  # コメント
  @sample
  シナリオ: 商品を検索して購入することができる
    前提商品検索ページにアクセスする
    かつ商品検索バーに"水"を入力する
    ならば検索結果一覧に以下が表示される
```

受け入れステージのプラクティス

1. 受け入れステージが失敗したら即座にチーム全体で修正する
2. 自動受け入れテストは自分たちの開発環境で実行できるようにする
    a. 失敗した場合開発者のマシンで再現できなければならない
3. 外部システムとすべて統合された環境で実行しない
    a. テストダブルを利用し、外部システムのふるまいはこちらで定義できように
4. 受け入れテストは実行可能な仕様として機能するよう、ビジネスの言葉(ユビキタス言語)で表現する
5. 本番環境と同じプロセスでデプロイを行う
    a. 自動デプロイメントのテストという意味合いもあるため

コミットステージと受け入れステージの例

Kubernetes で実行されるアプリケーション の場合



デプロイメントパイプラインを実装するタイミング

- デプロイメントパイプラインのプロセスは、プロジェクトの最初のうちに整備したほうが、イテレーションを何度か行ったあとに整備するのに比べて遥かにコストが低い
  - プロジェクトの初期からデプロイメントパイプラインが整備されていることで、自動テストや自動受け入れテストが前提となる開発になり、結果よい設計で開発が進み保守性が上がる
  - イテレーション0 のような準備期間で、動くスケルトンをもとにデプロイメントパイプラインを実装するのが理想
- プロジェクトの途中に導入する場合は、最も一般的で価値が高く、重要なユースケースから徐々に導入する

テスト戦略

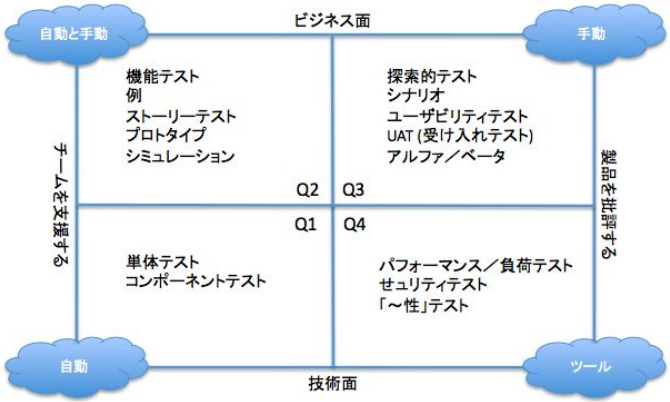
どのようにテストを設計するか?

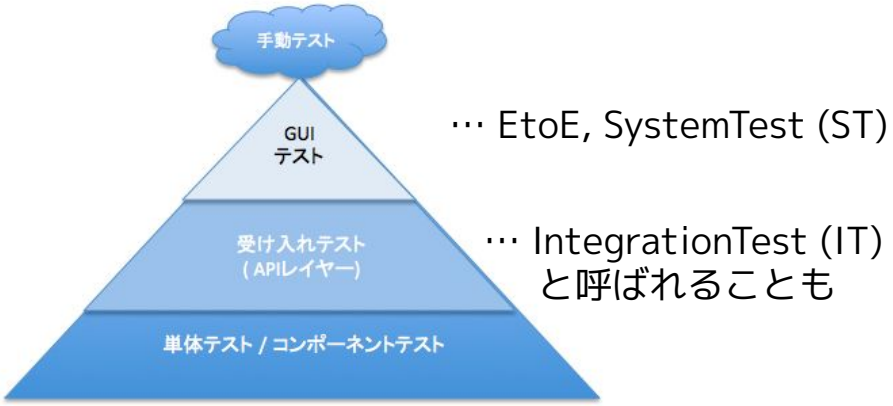
- **コミットステージ / 受け入れステージで実施するテストはどのように設計すればいいだろうか?**

コミットステージ	受け入れステージ
• 素早く実行できる • そもそもアプリケーションは起動するか? • 開発者視点のテスト	• 疑似本番環境で実行する • 機能的な価値を満たしているか? • 顧客視点のテスト

アジャイルテストの4象限 / テストピラミッド

- 「実践アジャイルテスト」では、**アジャイルテストの4象限 / テストピラミッド**を用いてソフトウェアにおけるテストにはどのような種類があり、どう自動化すると良いかが説明されている

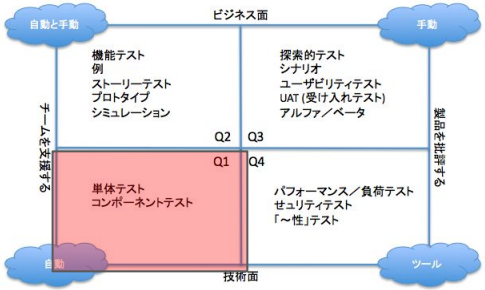


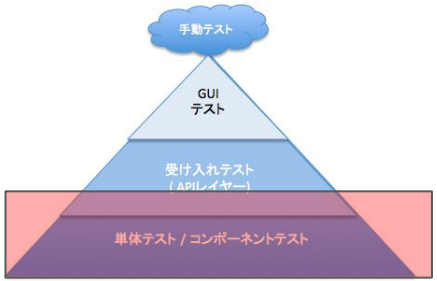


出典: https://notta55.hatenablog.com/entry/2015/05/03/161631

コミットステージで実施するテスト

- コミットステージでは主に Q1, 単体テスト/コンポーネントテストを実施する
- xUnit 系のテストで、クラスや関数に対してテストを実施することが多い (JUnit, PHPUnit など)
- 境界値試験など多くのテストケースが発生するテストはなるべくコミットステージで実施する

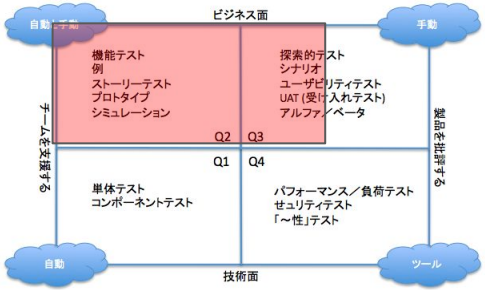


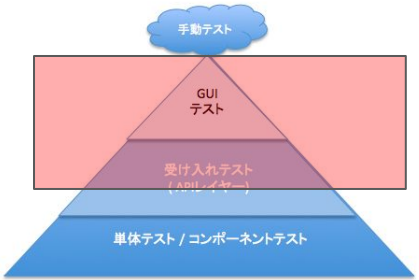


出典: https://notta55.hatenablog.com/entry/2015/05/03/161631

受け入れステージで実施するテスト

- 受け入れステージでは主に Q2, Q3, API レイヤー, GUI に対してのテストを実施する
- これらのテストは実行に時間がかかり、特に GUI のテストは少しの変更で壊れやすいため、単体テストほど厚くテストをしない
  - 境界値試験のようなテストはあまり実施しないようにする

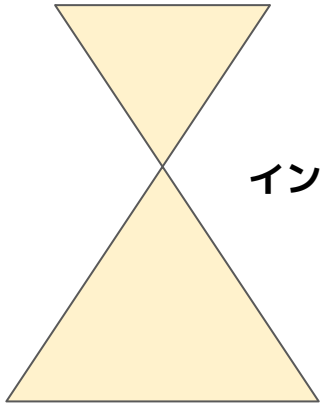




出典: https://notta55.hatenablog.com/entry/2015/05/03/161631

テストアワーグラス (砂時計)

- 近年コンピューターリソースが向上し、GUI を含むシステムテストや DB アクセスのコストが低下したことで生まれた考え方

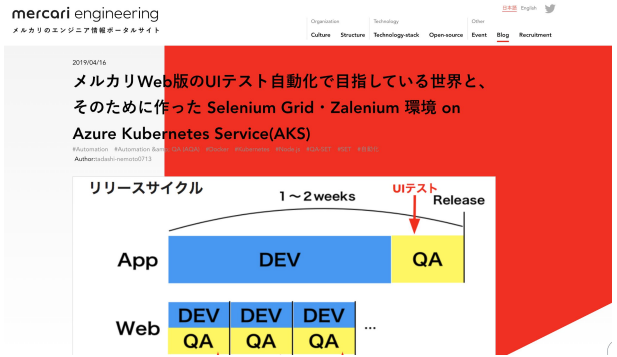


テストピラミッドの欠点:

- システムテストが少ないため、「なぜその機能が必要なのか」を忘れてしまう
- インテグレーションテストがあっても、今意味がある機能なのかがわからないためメンテナンスコストが高くなってしまう

GUI テストの実行時間を短くする取り組み

- GUI テストは一般的に時間がかかるが、**並列実行などで工夫すれば時間を短縮**することができる
- 例えばメルカリ様では、Selenium Grid をベースに作られた Zalenium と Kubernetes を組み合わせることで、**従来では1時間かかっていた GUI テストを 6~7 分で実施**することに成功



「メルカリWeb版のUIテスト自動化で目指している世界と、そのために作った Selenium Grid・Zalenium 環境 on Azure Kubernetes Service(AKS)」(https://engineering.mercari.com/blog/entry/2019-04-16-060000/)

システムの特徴によってテスト戦略は変わる

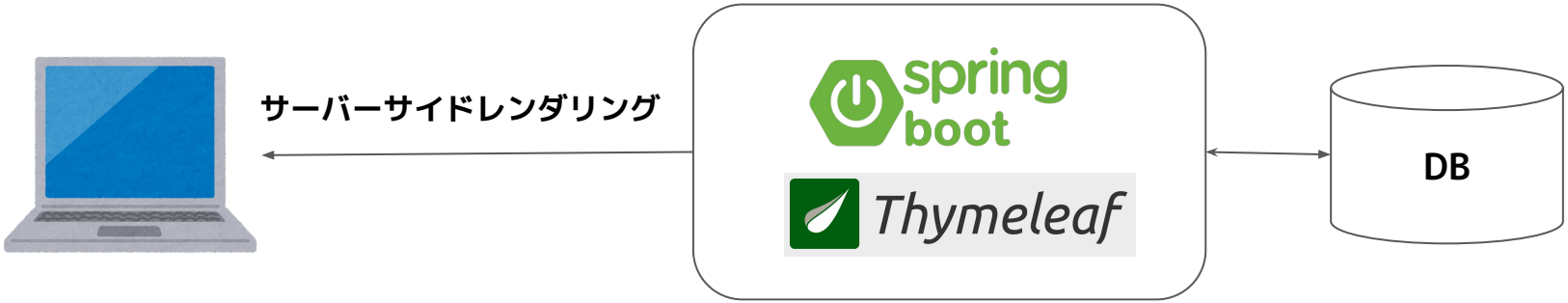
- 一般的なテストピラミッドの考え方は持ちつつも、システムの特徴にあったテスト戦略を考える
- 例) GUI を持たないアプリケーションでは、EtoE 試験を厚くしテストアワーグラスのような構造をとる
- 例) GUI テストが高速に実行できる基盤、ナレッジがあるので、GUI テストを厚くしテストアワーグラスのような構造をとる

テスト戦略の具体例をを考える

※発表者の見解であり、正解ではありません

例: 簡単な GUI をもつ Java (SpringBoot) の管理システム

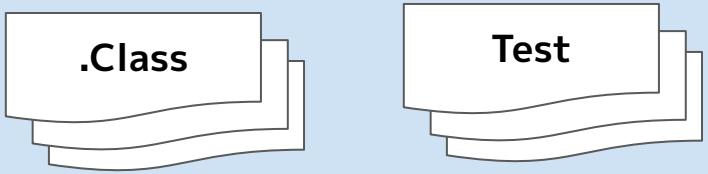
- SpringBoot、Thymeleaf を用いてサーバーサイドレンダリングで簡単な GUI を提供し、画面操作を通じて DB の値を操作する管理システムの場合



例: 簡単な UI をもつ Java (SpringBoot) の管理システム

コミットステージ

ユニットテスト



コンポーネントテスト

(クラスを結合させて動作検証する)

MockMVC

※ MockMVC … アプリケーションサーバ上にデプロイすることなくSpring MVC の動作を再現できる

受け入れステージ

Cucumber を使い自然言語でテストケースを記載

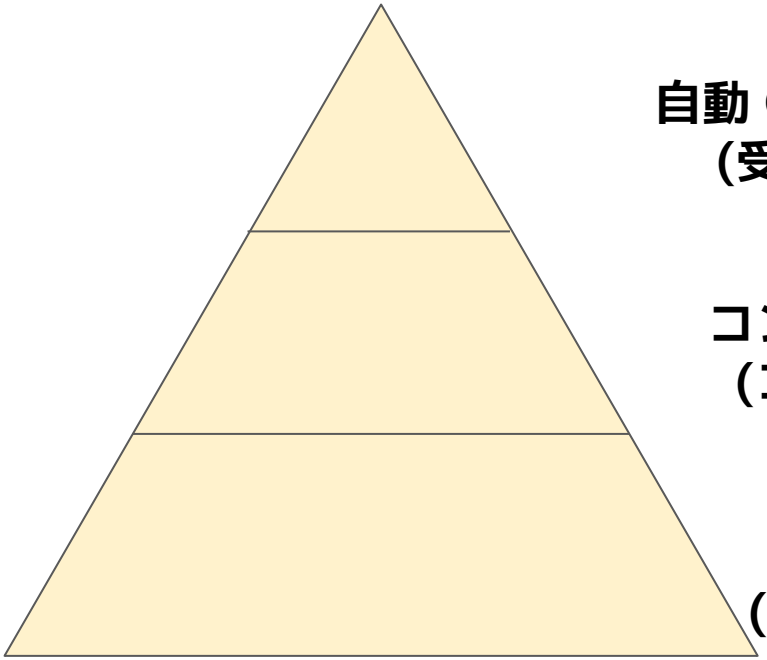
Selenide を使って 自動 GUI テストを実行

Selenide
CONCISE UI TESTS IN JAVA

cucumber

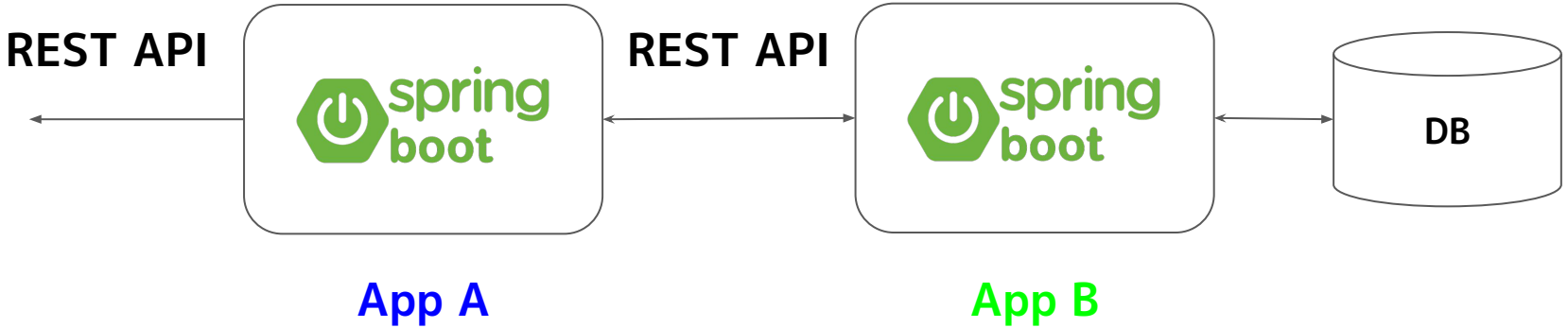
例: 簡単な UI をもつ Java (SpringBoot) の管理システム

- それぞれのテストの割合はテストピラミッドの考え方に近づける



例: 複数の REST API からなるシステム

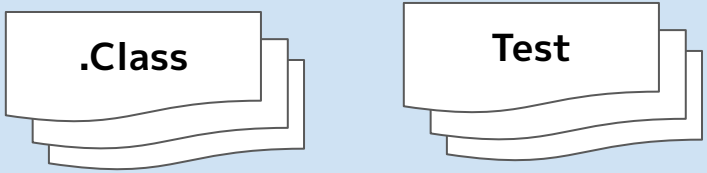
- REST API を提供する複数の SpringBoot アプリケーションからなるシステムの場合



例: 複数の REST API からなるシステム

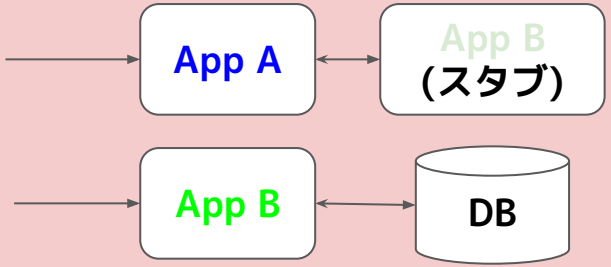
コミットステージ

ユニットテスト

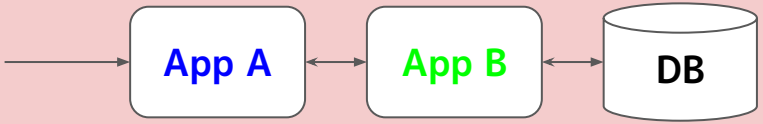


受け入れステージ

API ごとの受け入れテスト

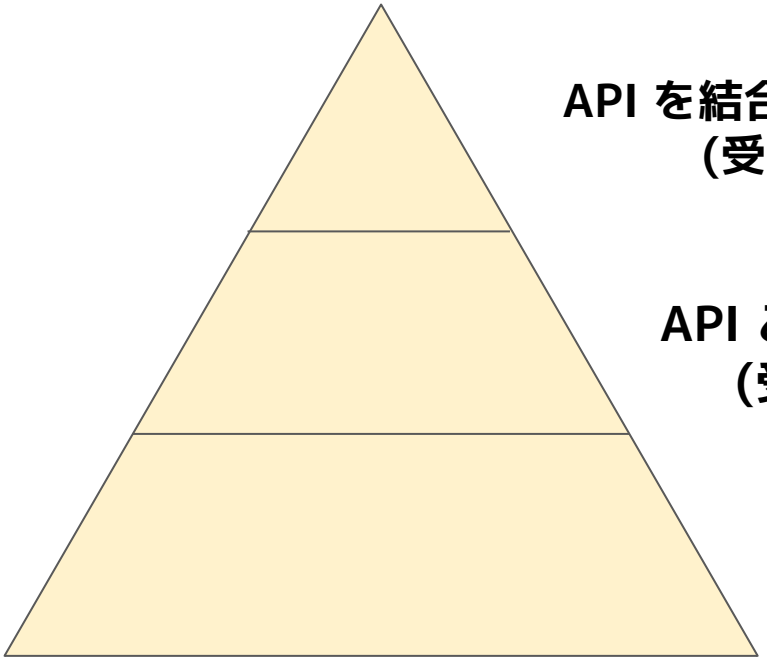


API を結合させた受け入れテスト



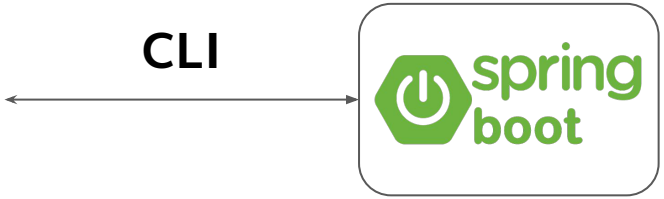
例: 複数の REST API からなるシステム

- それぞれのテストの割合はテストピラミッドの考え方に近づける



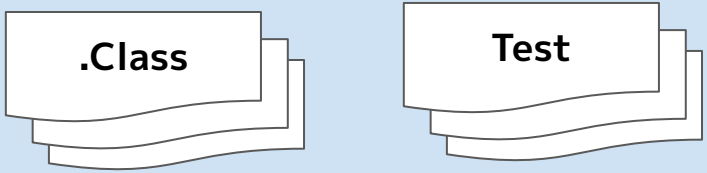
例: CLI アプリケーション

- Spring Boot CLI を用いて CLI アプリケーションを作成する



コミットステージ

ユニットテスト



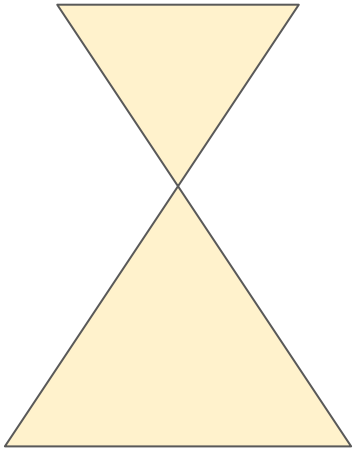
受け入れステージ

自動受け入れテスト

受け入れ基準となるテストケースを元に、
実際に CLI を実行して動作を検証する

例: 複数の REST API からなるシステム

- CLI アプリケーションであり、自動受け入れテストのコストテストの割合は低くなるため、それぞれのテストの割合はテストアワーグラスの考え方に近づける



自動受け入れテスト
(受け入れステージ)

ユニットテスト
(コミットステージ)

テスト戦略の具体例をを考える

※発表者の見解であり、正解ではありません

さいごに

まとめ

- 今回の発表ではデプロイメントパイプラインに関するプラクティスと自分が経験した具体的な方法論についてご紹介した
- 皆様が携わっている業務に少しでもお役に立てれば幸い

まとめ

- デプロイメントパイプラインについての具体的な内容・方法論はあまり情報として出回っていない(気がする)
  - 特にキャパシティテストやセキュリティテストの自動化など
- システムの特徴や状況によって、多種多様なデプロイメントパイプラインが存在するはず

→皆様のデプロイメントパイプラインも

是非教えて下さい！！

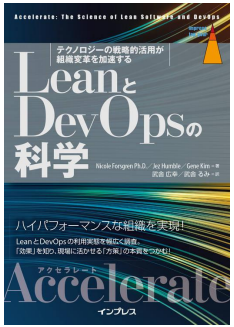
参考書籍

実践アジャイルテスト

出典: https://images-na.ssl-images-amazon.com/images/I/51TkJaY4jPL._SX400_BO1,204,203,200_.jpg

継続的デリバリー

出典: https://m.media-amazon.com/images/I/51v3XabFauL._SX260_.jpg

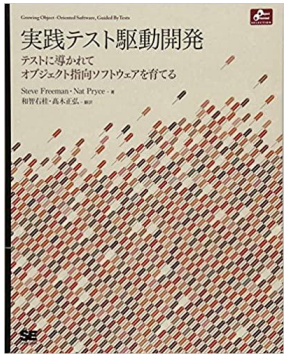


LeanとDevOpsの科学

出典: https://img.ips.co.jp/ij/18/1118101029/1118101029-520x.jpg

テスト駆動開発

出典: https://images-na.ssl-images-amazon.com/images/I/51hsd-b1RTL._SX350_BO1,204,203,200_.jpg



実践テスト駆動開発

出典: https://images-na.ssl-images-amazon.com/images/I/61UOZrEZR6L._SX398_BO1,204,203,200_.jpg

ご清聴ありがとうございました