

# テスト自動化

2021-05-13

テストエンジニアリング 園

# 講義の目的

- テスト自動化の基礎的な知識の把握
- (E2E)テストの自動化を体感する

**なぜ、テストを自動化するのか？**

# 速い！安い！正確！

## ■ テスト自動化のメリット

### ● 速い

人間がテストを行うよりも早く結果を出せる

### ● 安い

プログラムで実行するので、人的工数がかからない

### ● 正確

うっかり、見落としといった人的ミスがなく正確

# 開発プロセスに対する効果

## ■ 開発物に対する素早いフィードバックが可能

問題に対処するための工数を削減できる

- 記憶を掘り起こす時間
- 経緯を確認する時間
- (他人が作った)プログラムの内容を把握する時間

**短期間で開発を行うAgile開発において  
素早いフィードバックを行えることは何よりのメリット**

# 活用例：CI（継続的インテグレーション）

CI：Continuous Integration

## ■ プログラムの変更後、アーカイブビルドとテストを自動的に行う

- 不具合にいち早く気づくことができる
- 変更点のマージ時にテストが必ず自動的行われる  
→ 不具合を他人に引き継がない

## ■ レポジトリ内を常に安全な状態に保つ

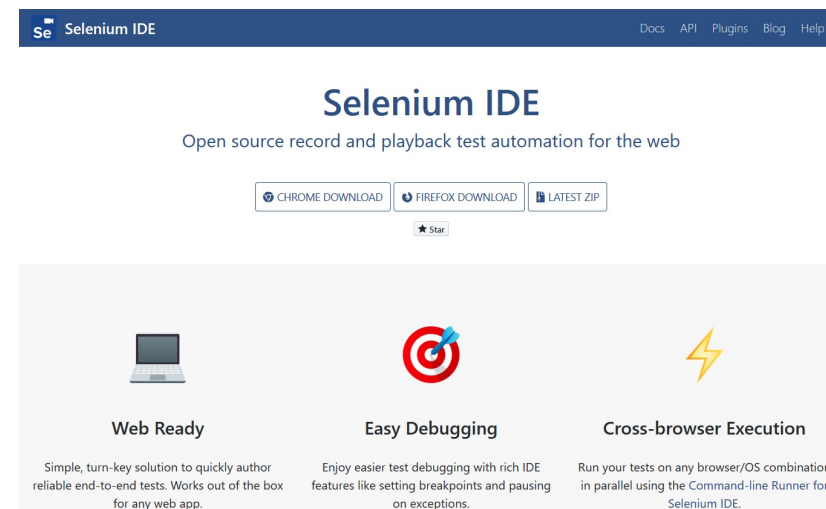
# E2E テストを自動化してみよう

E2E = end to end

# Selenium IDE

## ■ 公式サイト

<https://www.selenium.dev/selenium-ide/>



## ■ どんなツール？

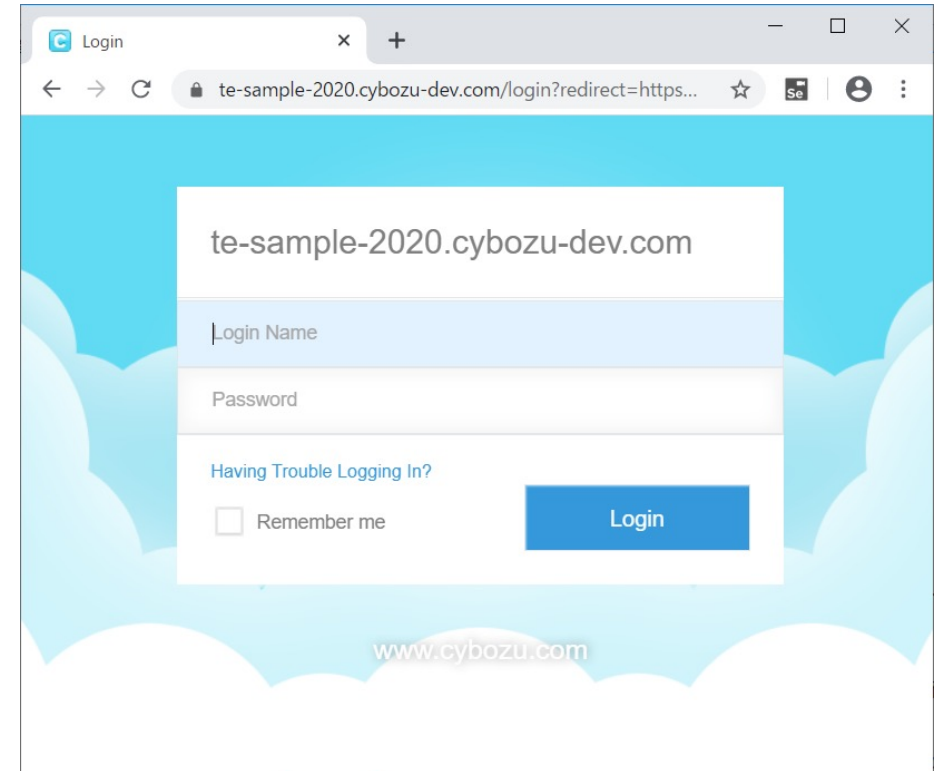
- ブラウザで行った操作を記録(Recode)して再現(Replay)する
- ブラウザのプラグインとして動作する  
Firefox / Chromeに対応
- 記録した操作をCUIで実行することも可能



# Selenium IDE – demo –

## ■ 「ログインを自動化」するデモ

1. プロジェクト名を決める
2. 操作をレコーディングする
3. Assertionを設定する
4. 自動で実行する



**このツールを使えば簡単にテストを自動化できます。**  
**どんどんテストを自動化して作業効率を上げましょう。**

**ご清聴ありがとうございました。**

end?

**こんな感じで  
意気込んでテストを自動化した結果  
使われなくなった例が多く存在します**

**なぜ、自動化したテストを使わないの？**

# 自動化したテストを使わなくなる理由

## ■ 動かなくなった

- 製品の仕様変更により動かなくなった
- ブラウザのバージョンなどの外部変化により動かなくなった

## ■ メンテナンスできない

- メンテナンスできる人がいない

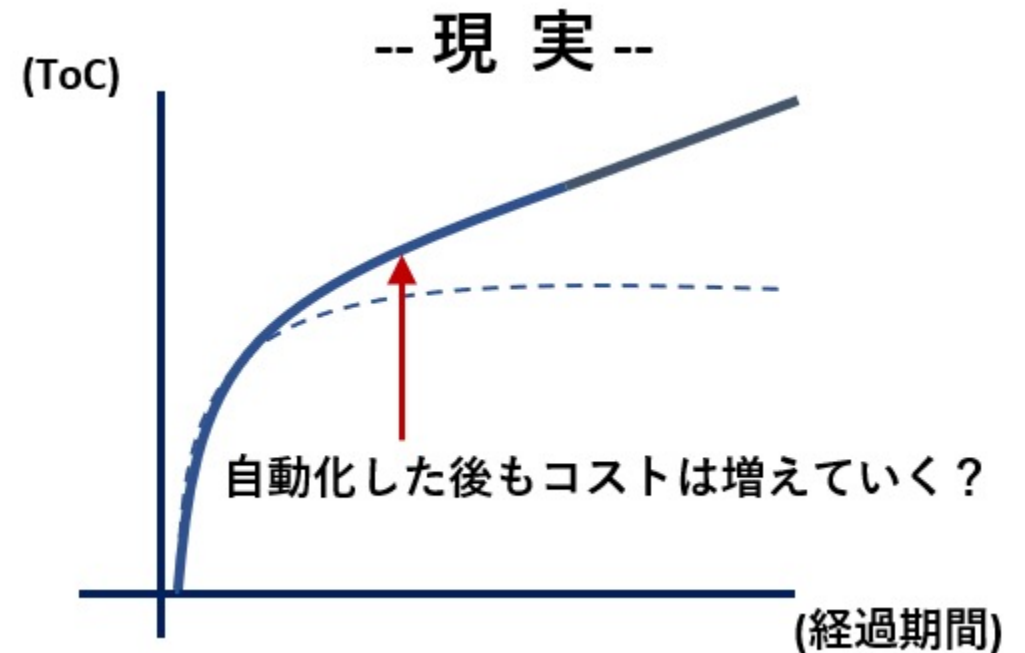
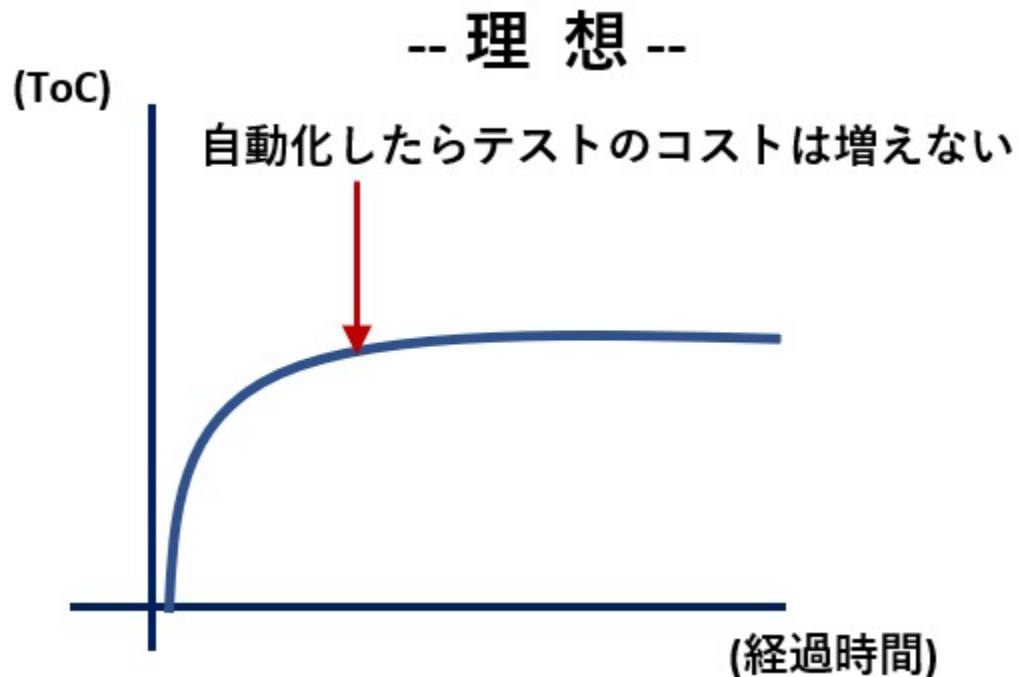
## ■ 工数が確保できない

- メンテナンスの量が多すぎて手が回らない

# 自動化したら、それ以上のコストがかからない？

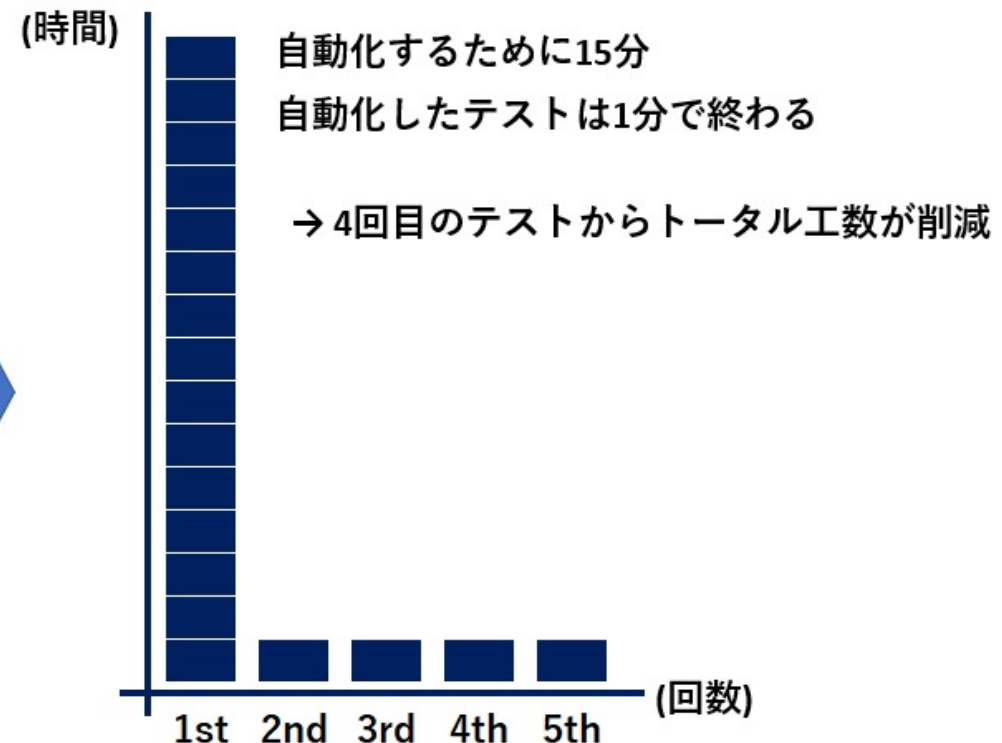
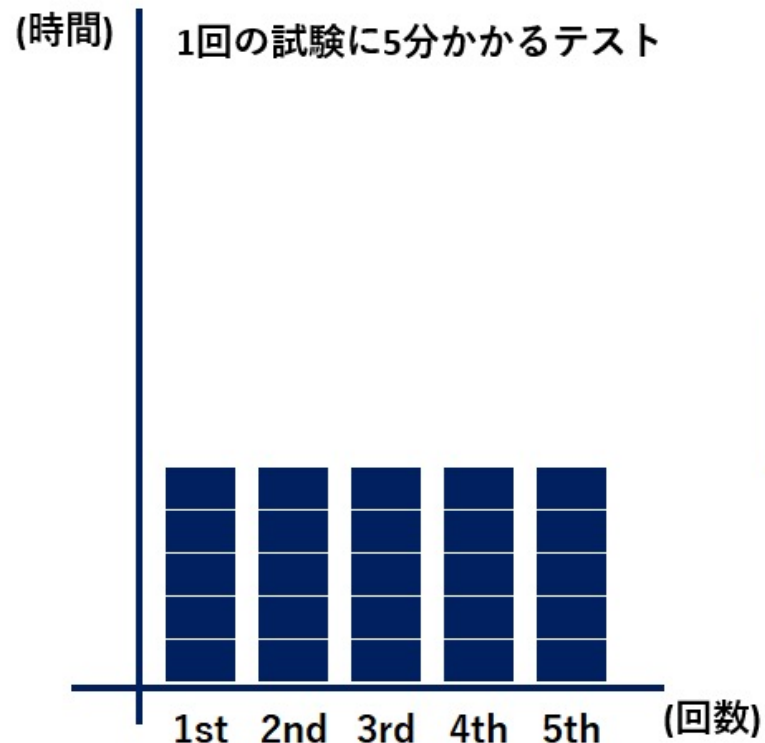
## ■ テスト内容に変化がなくても、メンテナンスを迫られる

- 機能や仕様の変化
- ブラウザ・OSの変化などの外的要因



# 自動化したテストは繰り返さないといけない

- テストを自動化するためには多くのコストが必要
- 繰り返し使うことで、そのテストにかかる ToC を下げる。





テスト自動化の恩恵を享受するには、

自動化したテストを **繰り返し長く使う** 必要がある。

繰り返し長く使うためには、

作成やメンテナンスにかかる**コストを下げる**

ことを意識したほうが良い

# 「作成・メンテナンスにかかるコストを下げる」

## 低コストで試験する

# 作成・メンテナンスのコストが高くなる原因は？

## ■ 初期実装コストが高い

- 技術的には自動化が可能だが、作成工数が高い  
⇒ 作成工数が高くないテストだけを自動化する

## ■ 仕様変更が多い機能

- いわゆる「枯れていない」機能  
⇒ 変更が少ないテストだけを自動化する

## ■ 外部仕様の変更

- OSやブラウザ・ライブラリの変化等  
⇒ 回避できない変化

# メンテナンスの発生を抑えてコストを下げよう

## ■ 変更が少ない“枯れている”機能

- 重要度の高い機能
- 繰り返しテストが実行される機能
- 自動化する難易度が低い機能

**自動化にかかるコストを削減するにも限界がある。**

**テスト自体のコストを削減できないか？**

# 「テストにかかるコストを下げる」

## 低コストで試験する

# 開発工程毎のテスト

## ■ 実装前の仕様検討

- 仕様を検討して、不具合の作りこみを防止する

## ■ Unitテスト(単体テスト)

- 関数やメソッドなど、最小単位のプログラムに対するテスト

## ■ インテグレーション(結合テスト)

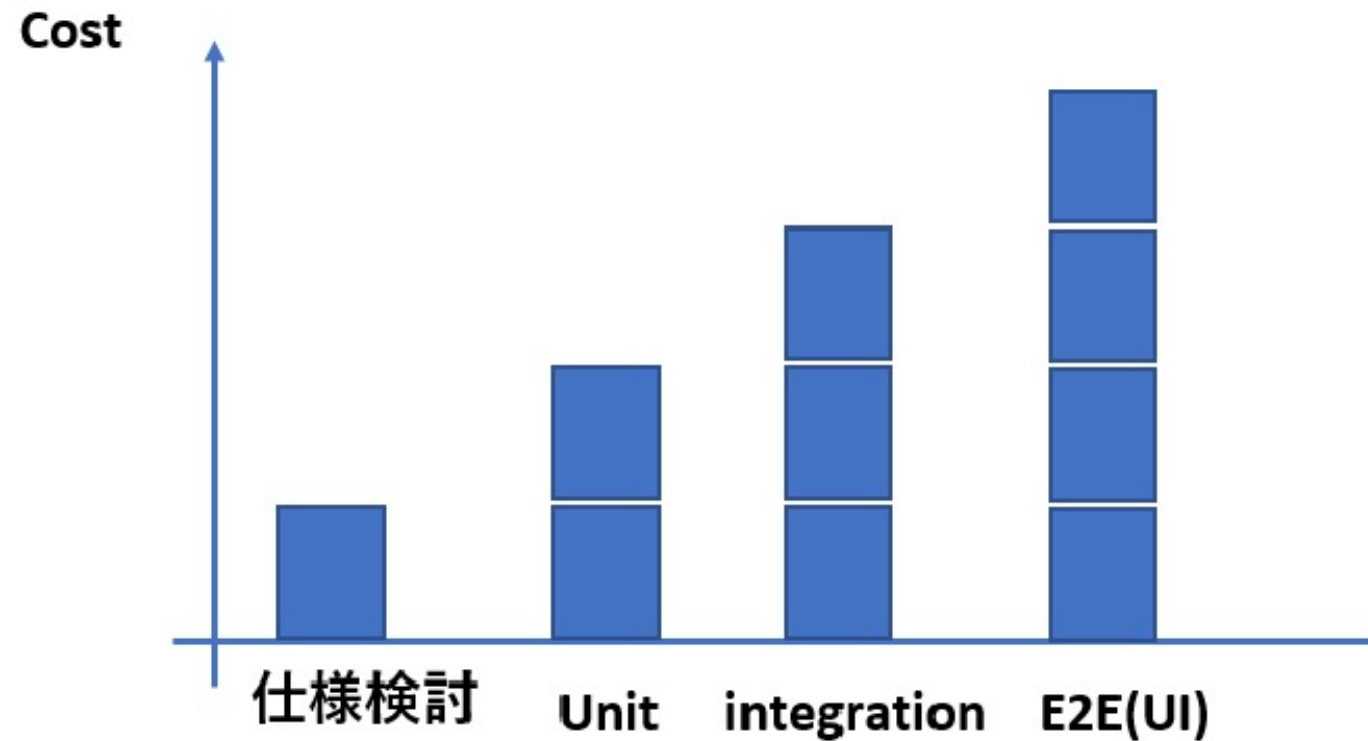
- 機能に対するUIを用いないテスト

## ■ E2Eテスト(UIテスト)

- ユーザー操作に最も近い、ブラウザを用いたテスト

# 開発工程とテストにかかるコスト

- 工程が進むにつれて、テストに必要なコストは増加する





# 例) ユーザー名とパスワードの確認

※ユーザー名とパスワードが一致していればエラーになることを確認

## ■ Unitテスト(単体テスト)

- 関数にユーザー名とパスワードを渡してエラーになるか確認

## ■ インテグレーション(結合テスト)

- (テストなし)

## ■ E2Eテスト(UIテスト)

- (テストなし)

# 例) ユーザー名とパスワードの確認

※ユーザー名とパスワードが一致していればエラーになることを確認

## ■ Unitテスト(単体テスト)

- (テストなし)

## ■ インテグレーション(結合テスト)

- APIにユーザー名とパスワードを渡してエラーになるか確認

## ■ E2Eテスト(UIテスト)

- (テストなし)

# 例) ユーザー名とパスワードの確認

※ユーザー名とパスワードが一致していればエラーになることを確認

## ■ Unitテスト(単体テスト)

- (テストなし)

## ■ インテグレーション(結合テスト)

- (テストなし)

## ■ E2Eテスト(UIテスト)

- ブラウザから入力してエラーが出ることを確認

# テストに必要な環境

## ■ Unitテスト(単体テスト)

- 環境作成の必要なし(関数の引数に値を設定)

## ■ インテグレーション(結合テスト)

- APIが動作する環境を作成
- ユーザーデータ

## ■ E2Eテスト(UIテスト)

- APサーバーを用意
- アーカイブを用意してインストール
- ユーザーデータ

テストにかかるコストを削減するために

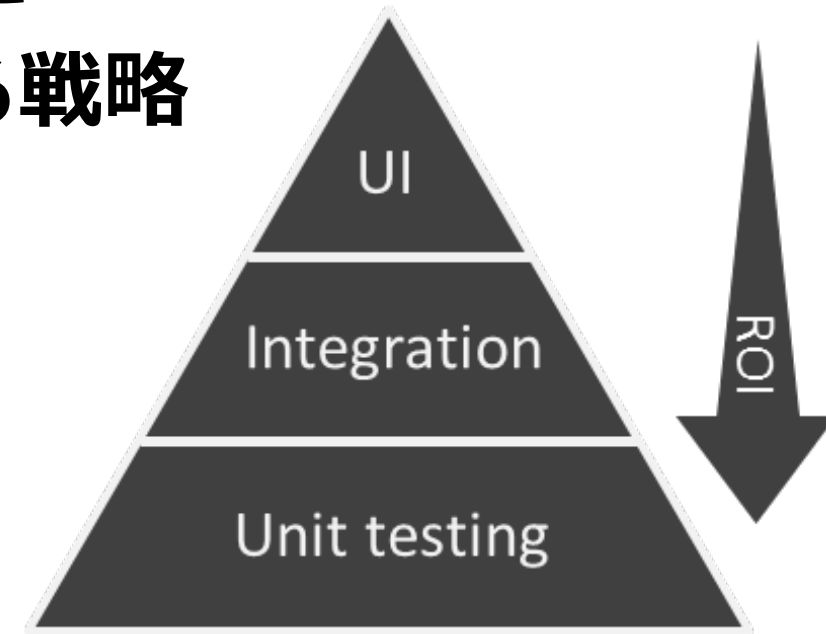
**適切なプロセスで適切なテストを行う**

ことが重要

# 『テストピラミッド』という概念

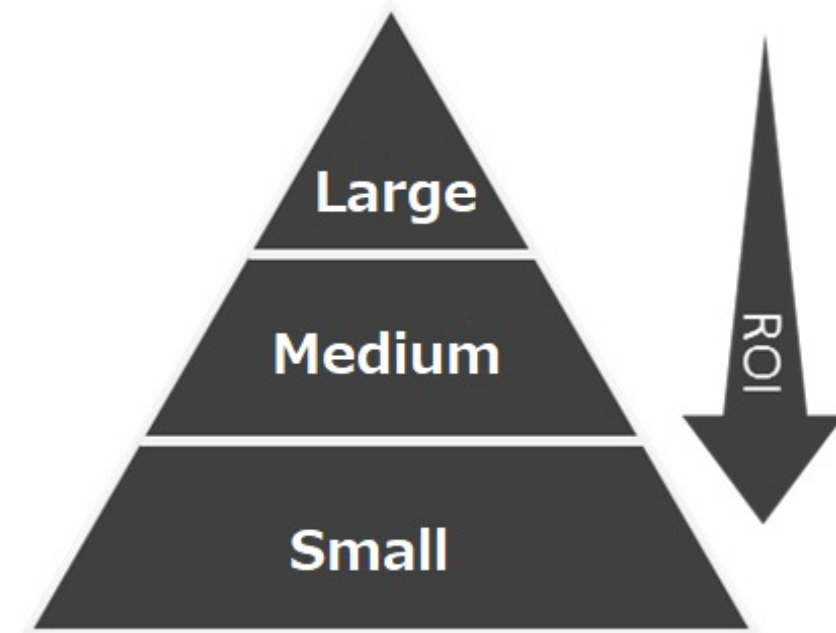
- テスト実行コストが低い層のテストを厚く行うことで全体のコストを抑える戦略
- 効率のよい開発(テスト)を行う上で重要な概念
- Mike Cohn氏が提唱したモデル

<https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid>



## 補足)

- テストの種別にあまり意味はない。
- 大切なのは、  
低コストで行えるテストを充実させ  
高コストのテストを少なくする戦略
- Small,medium,Largeと分類して、  
自分たちにあわせた分類を定義するとよい



# 適切なタイミングでテストを行う

## ■ どの段階のテストも必要なテスト

- どの段階のテストが悪い、という話ではない。

## ■ 各段階のテストの特性を理解することが重要

- 何を目的としたテストなのか？
- テスト準備・テストにかかる工数の違い



# 例) ユーザー名とパスワードの確認

※ユーザー名とパスワードが一致していれば**エラー画面に遷移する**

## ■ Unitテスト(単体テスト)

- (テストなし)

## ■ インテグレーション(結合テスト)

- (テストなし)

## ■ E2Eテスト(UIテスト)

- **エラー時にエラー画面にリダイレクトされていること**

# 「チームで取り組む」

# テストはテスターがやるもの？

## ■ プログラマとテスターが組織的に分かれていた時代

- テストはテスターが実施するもの
- テスト自動化は、プログラマの詳しい人が手掛けていた
  - テスター側でメンテナンスができない
  - 新規自動化もメンテナンスも、プログラマの工数頼り
- 協力体制を構築するところから開始
  - 協力を得られなかったり工数の融通ができなかったりした結果  
自動化したテストが活用されないことも多かった

# 製品品質の向上はチーム全体の課題

## ■ チームでなければできないことがある

- 特定の工程だけで品質を向上させるのは限界がある  
例) テストピラミッドの概念の実現

## ■ あなたにしかできないことがある

- プログラマだから実装できるUnitテスト
- UIスペシャリストだからわかるユーザビリティ的な指摘

# 「人」に依存しない体制づくり

## ■ 「人」はいなくなる

推進する人が異動や退職でいなくなる

## ■ 独りで作れても、独りで維持はできない

独りで「熱」を維持するのは難しい

自分の工数は無限ではない

## ■ チームのメンバーを巻き込む

**品質向上をチームの課題と考えて  
テストやテスト自動化に取り組む『文化』  
を構築していくことが重要**

# チームで共有しよう

## ■ テスト自動化の方針

- 自動化の方法・ツール
- 誰が、どのテストを、どのタイミングで実装するのか？

## ■ テスト自動化の工数

- 作成・メンテナンスに必要な工数

## ■ テスト自動化の知識

# チームに合わせたツール選び

## ■ 製品とツールの相性

- 開発言語
- テスト内容

## ■ 目的に合わせたツール

- 静的解析
- 動的テスト

## ■ チームメンバーのスキルと習得難易度



自動化したテストを「**繰り返し長く使う**」  
ためには、  
適切な段階で適切なテストを自動化する  
という  
文化をチーム内で育成していく  
ことが重要



**休憩（～14:10）**



## E2E テストを自動化しよう（実践）

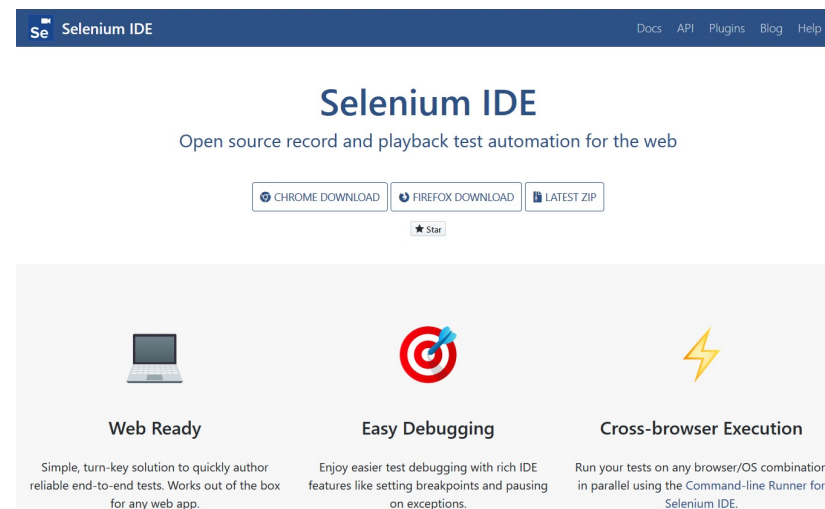
# E2Eテストの特徴

- **エンドユーザーの操作を再現する**
- **データなどを事前に準備する必要がある**
- **簡単に壊れやすい**

# Selenium IDE

## ■ 公式サイト

<https://www.selenium.dev/selenium-ide/>



## ■ どんなツール？

- ブラウザに対する操作を記録(Recode)して再現(Replay)する
- 利点：GUI操作で初心者でもわかりやすく簡単に作成可能
- 難点：オブジェクト指向ではないので、修正コストが高い

## ■ 公式サイト

<https://autify.com/ja>

## ■ どんなツール？

- ブラウザに対する操作を記録(Recode)して再現(Replay)する
- 利点：GUI操作で初心者でもわかりやすく簡単に作成可能  
UIの変更をAIである程度追隨して自動でテストを修正する
- 難点：オブジェクト指向ではないので、修正コストが高い



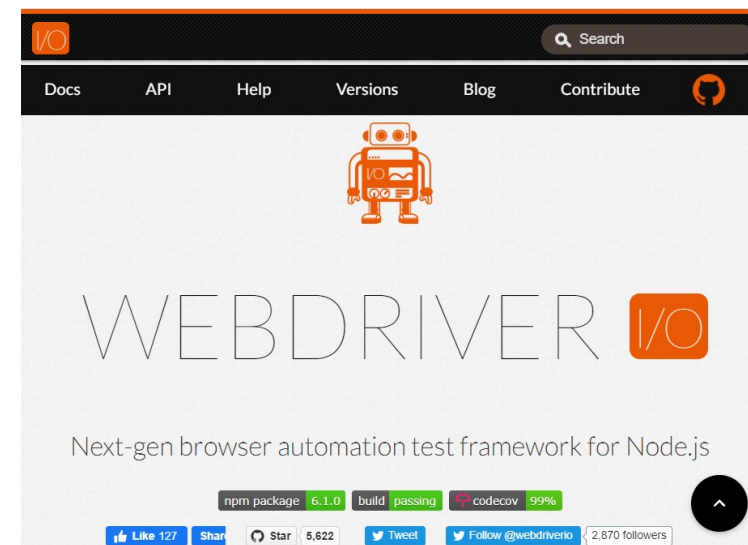
# WebDriver IO

## ■ 公式サイト

<https://webdriver.io/>

## ■ どんなツール？

- Selenium WebDriver をNode.js上で動作させるフレームワーク
- 利点：レポート・Assertionツールを組み合わせで柔軟なテストが可能
- 難点：プログラムベースのため習得のハードルがやや高い



# TCB (Test Common Base)

## ■ 公式サイト

<https://github.dev.cybozu.co.jp/pages/te/tcb-wiki/>

## ■ どんなツール？

- 内製(TEチーム作成)のWebDriverIOベースのテストツール
- 利点：ページオブジェクト指向でメンテナンスコストが低い
- 難点：プログラミングベースなため、習得のハードルが高め



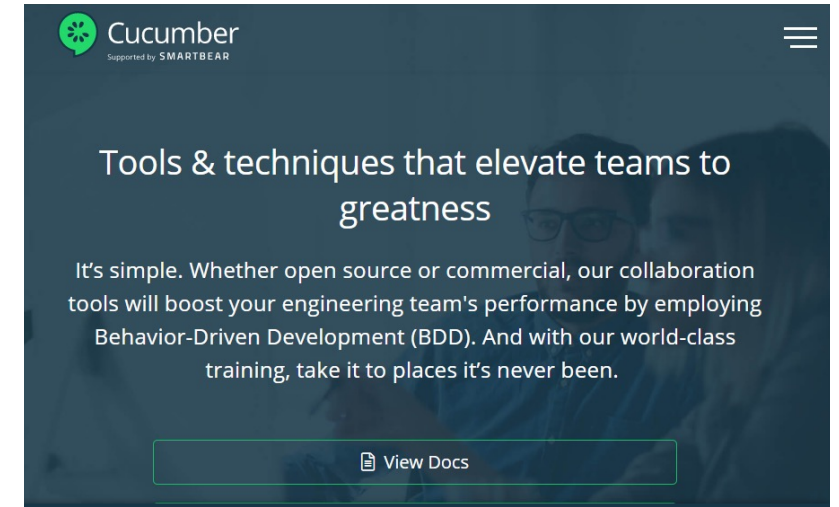
# Cucumber (Gherkin)

## ■ 公式サイト

<https://cucumber.io/>

## ■ どんなツール？

- テスト手順(シナリオ)を自然言語(日本語)で記載する
- 利点：テスト内容などの把握が容易になる
- 難点：シナリオと駆動部を別々のレイヤーで管理するコストが高い



# E2E テストツールを選ぶポイント

## ■ 「誰に」対してわかりやすいツールを選ぶか？

- 実装者のスキル
- テスト結果の視認性

## ■ メンテナンス性をどう考えるか？

- 作りやすさを優先するのか？
- (ページオブジェクトの)変更への対応力を優先するのか？

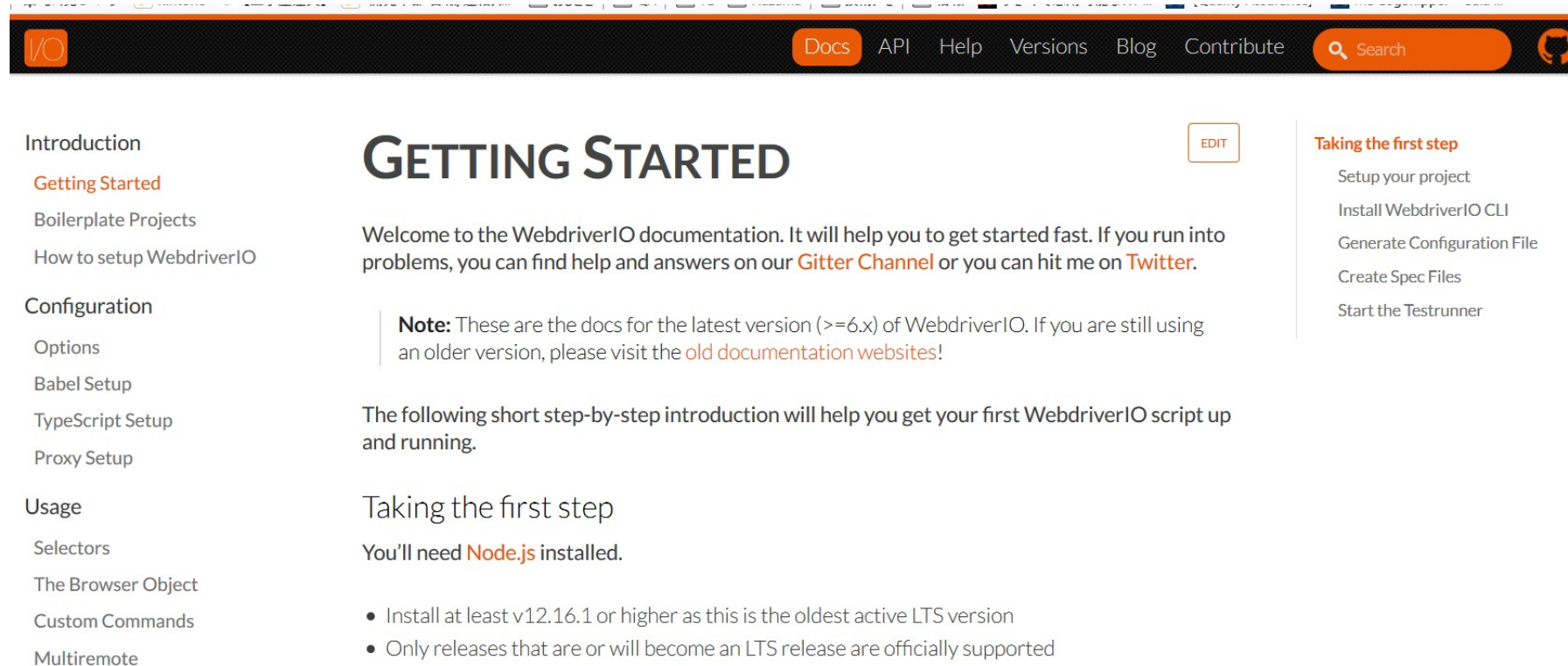
**チームメンバーの  
構成や人数、スキルによって最適解は異なる**

# テストを自動化してみよう (プログラム編)

# WebDriverIOで自動化する

## ■ チュートリアルページ

<https://webdriver.io/docs/gettingstarted.html>



The screenshot shows the 'GETTING STARTED' page of the WebdriverIO documentation. The page has a dark navigation bar with links for Docs, API, Help, Versions, Blog, and Contribute, along with a search bar and a GitHub icon. The main content area is divided into three columns. The left column contains a sidebar with links to Introduction, Getting Started, Boilerplate Projects, How to setup WebdriverIO, Configuration, Usage, and Multiremote. The middle column features the 'GETTING STARTED' heading, a welcome message, a note about the latest version, and a list of steps for taking the first step. The right column contains a 'Taking the first step' section with links to Setup your project, Install WebdriverIO CLI, Generate Configuration File, Create Spec Files, and Start the Testrunner.

**GETTING STARTED** EDIT

Welcome to the WebdriverIO documentation. It will help you to get started fast. If you run into problems, you can find help and answers on our [Gitter Channel](#) or you can hit me on [Twitter](#).

**Note:** These are the docs for the latest version ( $\geq 6.x$ ) of WebdriverIO. If you are still using an older version, please visit the [old documentation websites](#)!

The following short step-by-step introduction will help you get your first WebdriverIO script up and running.

**Taking the first step**

You'll need [Node.js](#) installed.

- Install at least v12.16.1 or higher as this is the oldest active LTS version
- Only releases that are or will become an LTS release are officially supported

**Taking the first step**

- Setup your project
- Install WebdriverIO CLI
- Generate Configuration File
- Create Spec Files
- Start the Testrunner