システムテスト自動化の技法

小井土亨

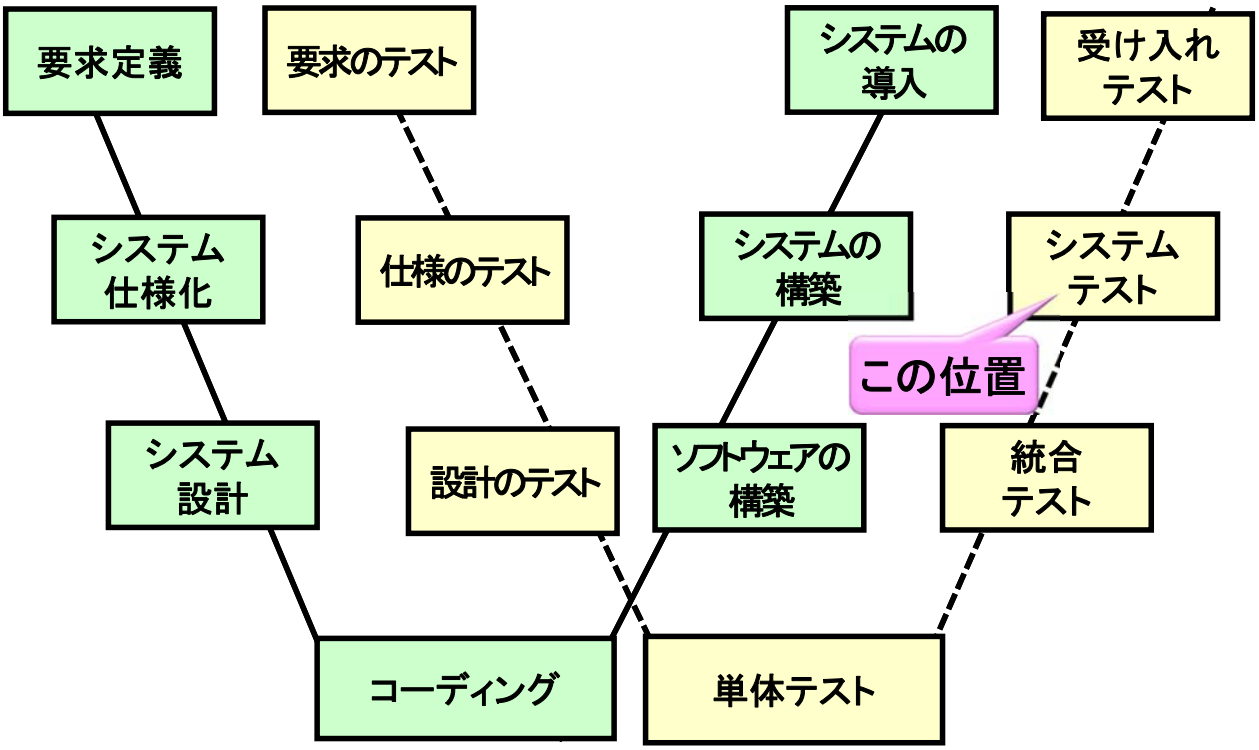
- アジェンダ
  - システムテスト
  - GUIテスト自動化
  - キーワード駆動テスト
  - システム自動テストのアーキテクチャ
  - システムテストの自動化について

2016.12.02

システムテスト

1.システムテスト
2.エンドツーエンドテスト
3.システムテストの自動化について

システムテスト
開発プロセス内のシステムテストの位置づけ

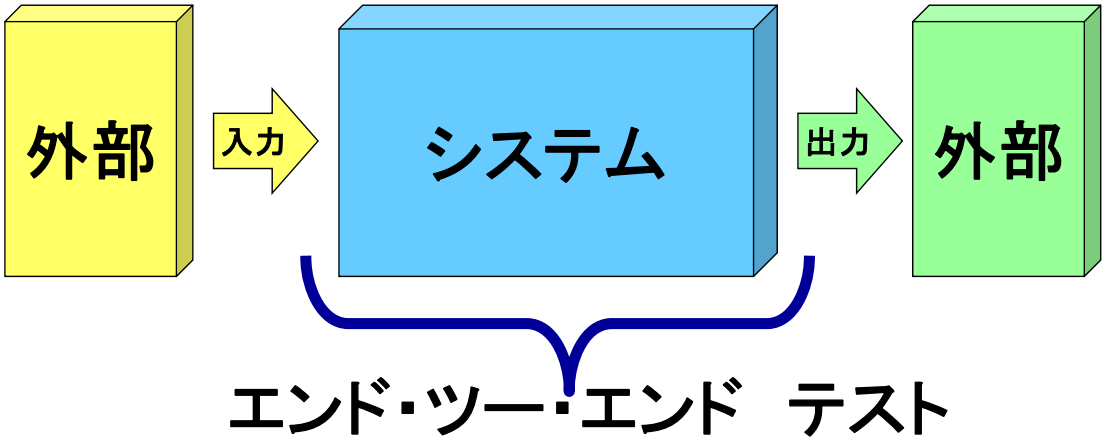


システムテスト
システムテストとは

- システムテストとは
  - システムが全体として正しく動作することをテストする
  - 様々な視点でのテストが必要、特に利用ユーザー視点
- システムテストの例
  - 運用テスト
    - 実運用に近い環境で、実際に運用するテスト
  - 操作性テスト
    - 利用ユーザー視点で、操作を行うテスト
  - 構成テスト
    - 様々な構成パターンでのテスト
  - マニュアルテスト
    - マニュアル通りに操作して問題なく動作することを確認するテスト

システムテスト
システムテスト　＝　エンドツーエンドテスト

- システムテストは、エンドツーエンド テスト
  - end to end　とは
    - 端から端まで
  - システムの端から端までをテストする
    - 全体として正しく動作することを確認するため



エンド・ツー・エンド　テスト

システムテスト
エンドツーエンドテストについて

- エンドツーエンドのシステムテスト
  - 外部からの入力に対応したシステムの動作によって、外部に対して行った結果をテストする
- なぜ、エンドツーエンドテストなのか
  - 部分に対するテストを積上げても、繋がっていることを保証できない

システムテスト
エンドツーエンドテストについて

- システムテストの結果
  - 外部への出力
    - 画面、通信、印刷、ファイル
  - 外部システムへの処理
    - データベース、外部のサービス
- エンドツーエンドテストの特徴
  - テストの実行時間が長いことが多い
  - テストの準備や結果判断は、システム外部も含めて行う必要がある

システムテスト ≒ GUIテスト

注：GUIテストとは、GUI操作を伴うテスト実行

システムテスト
ビジネスに追随するシステムとシステムテスト

- ビジネスに追随するシステムとは
  - ビジネス要件（エンドユーザーの要望）により、短期間で機能追加とリリースを繰り返すシステム
  - 最新の機器や環境に、短期間で対応するシステム
- ビジネスに追随するシステムのシステムテスト
  - リリース（機能追加）ごとに、システムテストを実施する必要がある
  - システムテストを実施する回数が増える

システムテストのコストが増大する

システムテスト
テストコスト増大への対応策

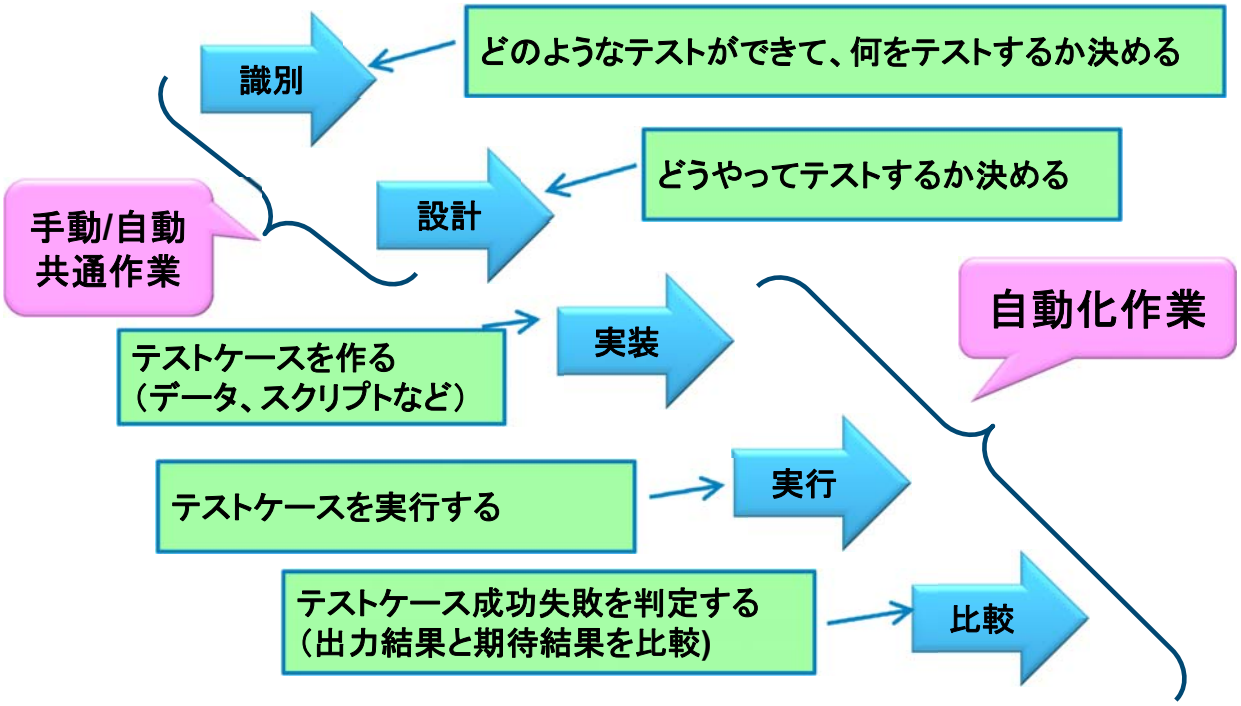
- 対応すべきシステムテスト
  - リリースごとに繰り返されるテスト
    - 前回までのリリースで提供した機能について、同じように動作することを確認するテスト
- 回帰テスト（リグレッションテスト）
  - 今回のリリースと前回のリリースで同じことを確認する
- 構成テスト
  - さまざま構成で同じテストを実行し、同じ品質であることを確認する

リリースされる新しい機能の最初のテストは繰り返さない

システム（GUI）テストを自動化する

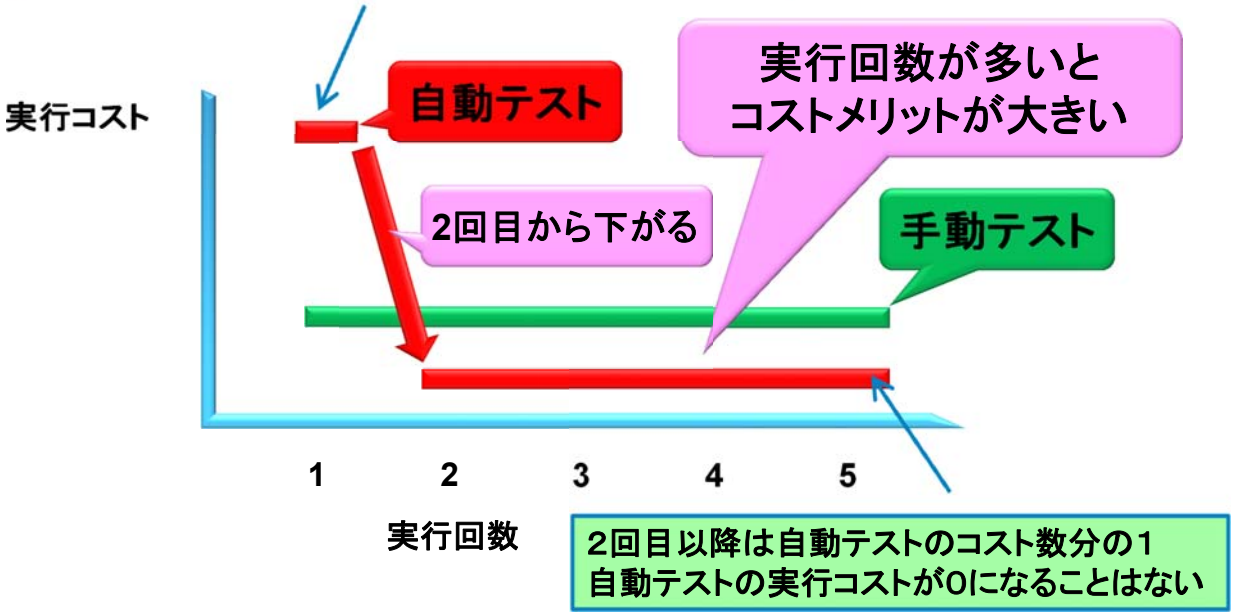
システムテスト
テスティングと自動テスト

- テスティング活動



システムテスト
テストケース実行コスト

初回はテストケースを作成するコストが必要なので
自動テストを比較すると数倍のコストが必要



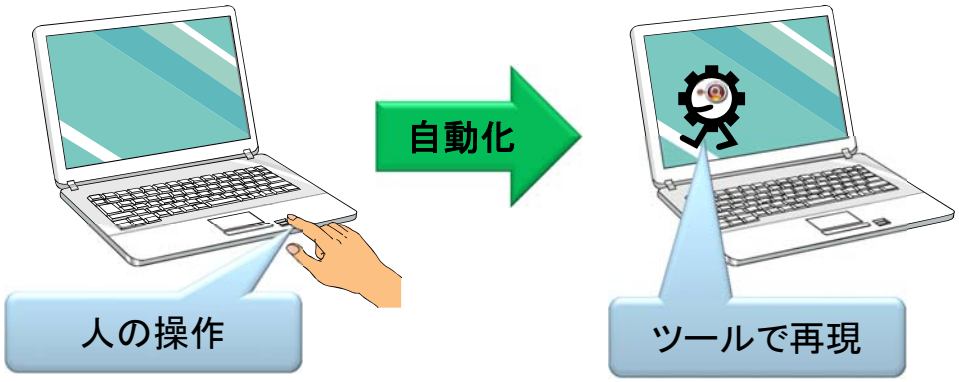
2回目以降は自動テストのコスト数分の1
自動テストの実行コストが0になることはない

GUIテスト自動化

1. キャプチャーリプレイ
2. キーワード駆動テスト

GUIテスト自動化
GUIテスト自動化とは

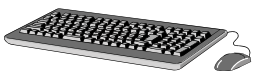
- GUIテスト自動化とは
  - GUIを通して操作した場合と同等になる様にツールで操作を再現することで行うテスト
- ツールの種類
  - キャプチャーリプレイ型やキーワード型などがある



GUIテスト自動化
キャプチャーリプレイとは

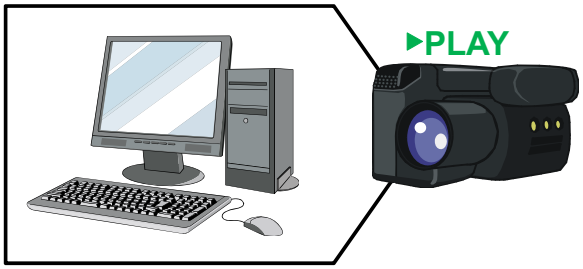
- キャプチャーリプレイとは
  - OSやツールが提供する、ユーザー操作の記録と再生機能

実際に操作した内容をツールで記録



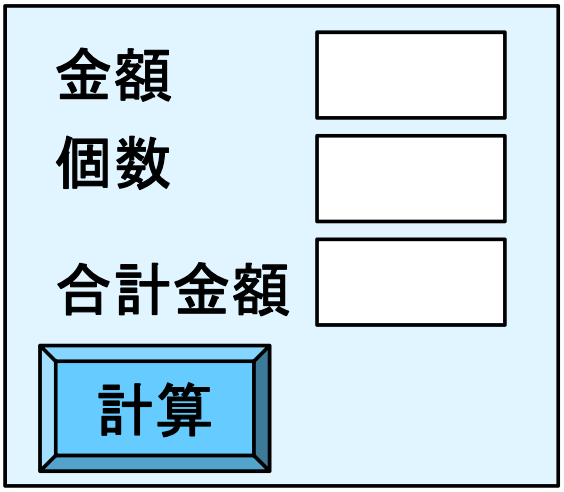
キーボード操作
マウス操作

記録した内容をツールで再生



- キャプチャーリプレイの課題
  - テストケースを作成するためには、操作する必要がある
  - 記録時と再生時で、処理時間が異なった場合、目的通りに再生されないことがある
  - テスト対象の変更された場合、作り直しが必要となる場合がある

GUIテスト自動化
キャプチャーリプレイの例



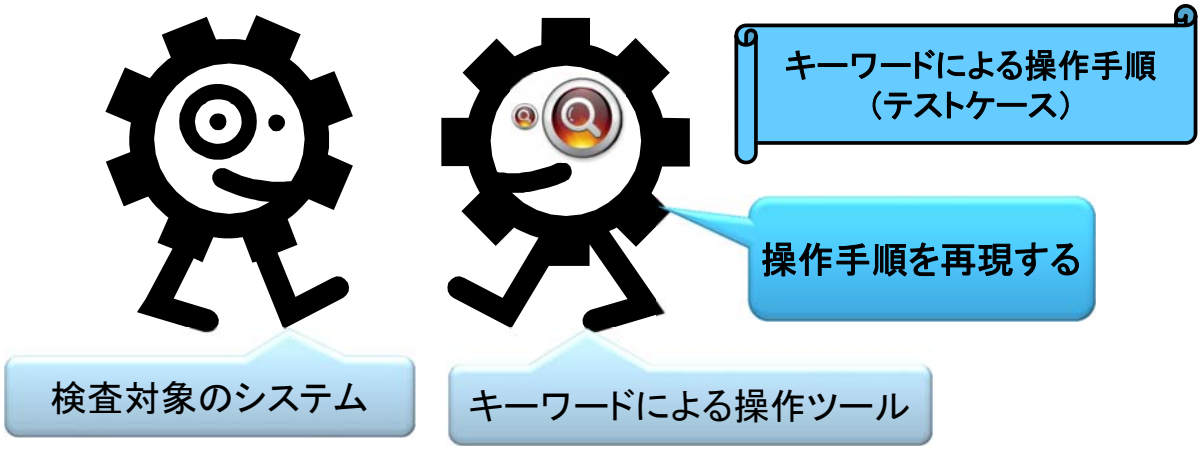
- 記録結果
  - マウス移動:10,100
  - マウス左クリック
  - キー入力:1000
  - キー入力:{Tab}
  - キー入力:5
  - 待ち:1
  - マウス移動:100,10
  - マウス左クリック

計算処理を1秒待つ

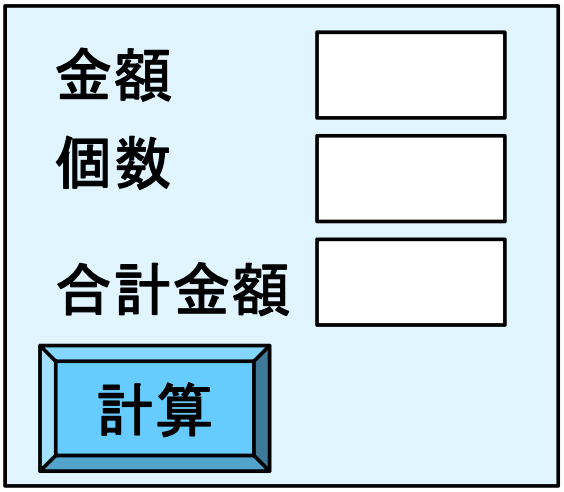
- 課題
  - 場所が変わると正常動作しなくなる
  - 処理時間が変わると正常動作しなくなる
  - 実行結果のチェックができない
  - テストの内容が分からない

GUIテスト自動化
キーワード駆動テスト

- キーワード駆動テスト
  - 論理的なキーワードを用いて記述されたテストケースをツールが解釈し、テスト対象を操作するテスト
  - 操作として、値の設定以外に、状態や値の取得を提供する



GUIテスト自動化
キーワード駆動の例



- 操作手順

対象	値	操作
金額	1000	Set
個数	5	Set
計算		Click
合計金額		Get

- 利点
  - テストの内容が分かるため、多くの人が作成できる
  - 場所や処理時間に影響されない
    - 待ち設定が必要ないのは、ボタンの押下処理を同期処理で実現しているため
  - 実行結果を取得し確認できる

GUIテスト自動化
キャプチャーリプレイとキーワード駆動テスト

比較項目	キャプチャーリプレイ	キーワード駆動テスト
導入しやすさ	○ツールの導入コストのみ	×ツールだけではなく、検査対象のシステムの対応が必要
テストケース作成工数	△操作して記録するため、作業工数の削減は難しい	○テストケースの作成は容易 テストケースの自動生成も可能
システム変更への対応	×対応可能な場合もあるが、多くの場合作り直しが必要となる	△項目追加などは、変更せずに利用できる場合も多い
異なった環境での再現性	×実行時間や座標位置などが変わると再現できなくなる場合が多い	○環境の違いなどをツールが吸収して、再現できる場合も多い
テスト結果の確認	×テスト結果の確認は、他の方法を用意する必要がある	○テスト結果を取得することで自動判定が可能

キーワード駆動テスト

1.キーワード駆動テスト
2.問題と解決策
3.推奨構造

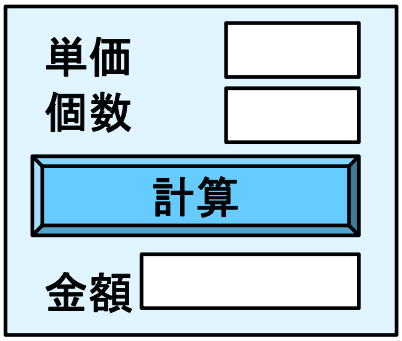
キーワード駆動テスト
データ駆動とキーワード駆動

- データ駆動スクリプト
  - テストスクリプトとデータを分離し、スクリプトがファイルからデータを読み込んで実行する
  - 例：キャプチャで作成したスクリプトから値の部分を変数にし、外部ファイルから読み込む
- キーワード駆動スクリプト
  - スクリプトに、テストのすべての情報を簡易な方法で記述
  - 論理的なキーワードを用いて記述されたテストスクリプトをツールが解釈し、テスト対象を操作するテスト
  - 操作として、値の設定以外に、状態や値の取得を提供する

キーワード駆動
実行結果のチェック方法

- 方式①
  - キーワード駆動にチェック機能を用意する
- 方式②
  - キーワード駆動では、値の取得だけを行い、判定は別途行う
- 利点
  - 方式①　実行時すぐに判定できる
  - 方式②　システム外部の状態（DBなど）も含めて判定が可能

キーワード駆動テスト
データ駆動とキーワード駆動の例



- データ駆動

自動スクリプト

```
TestBoxA.SetTest(tanka)
TestBoxB.SetTest(kosu)
```

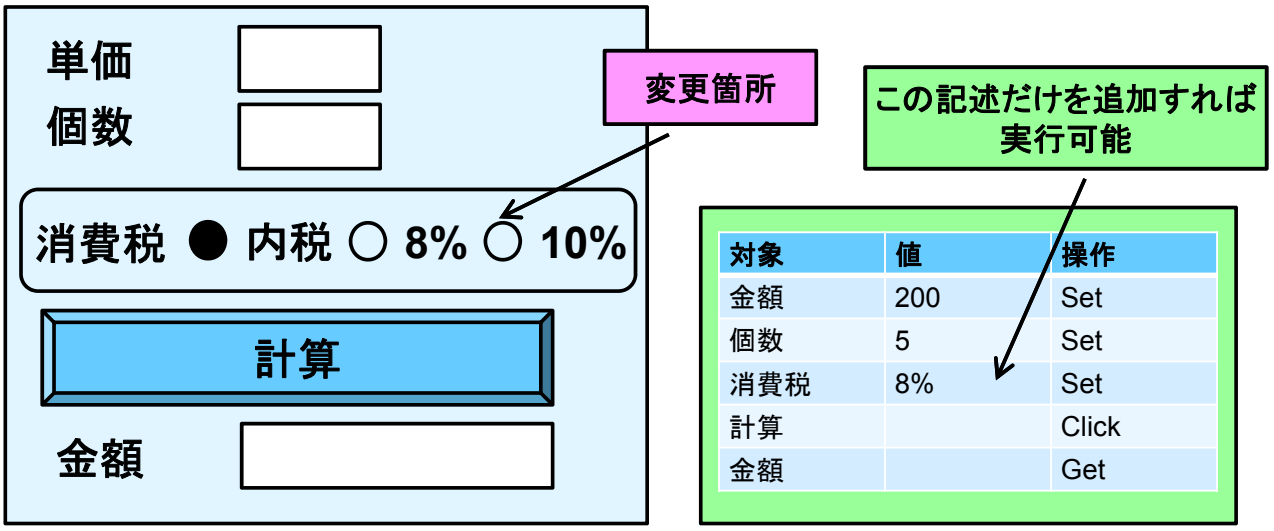
読み込み

単価	個数
200	5

- キーワード駆動

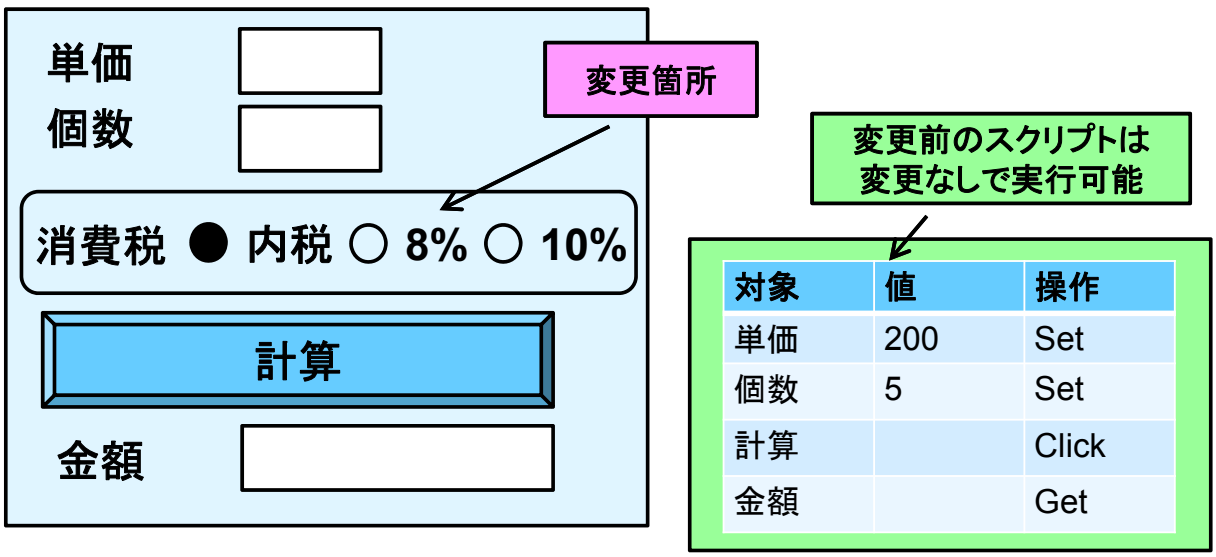
対象	値	操作
単価	200	Set
個数	5	Set
計算		Click
金額		Get

キーワード駆動テスト
キーワード駆動のシステム変更への対応例



対象	値	操作
金額	200	Set
個数	5	Set
消費税	8%	Set
計算		Click
金額		Get

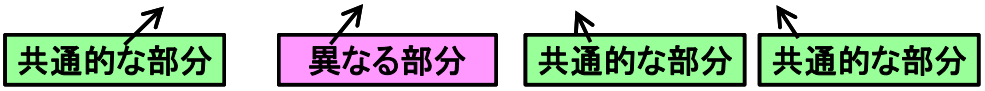
キーワード駆動テスト
キーワード駆動のシステム変更への対応例



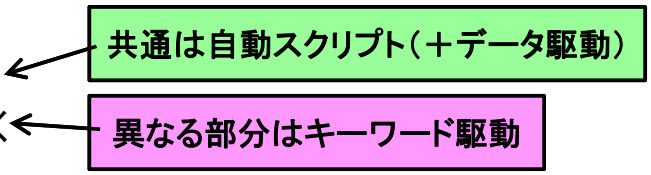
対象	値	操作
単価	200	Set
個数	5	Set
計算		Click
金額		Get

キーワード駆動テスト
キーワード駆動に起因する問題

- キーワード駆動は、制御処理(IF文など)の記述が難しい
  - 問題
    - 同じ部分を含んだスクリプトが大量に作成される
    - テストスクリプトに、他のテストスクリプトと共通的な部分と、他のテストスクリプトと異なる部分(テストバリエーション)が混在する
  - 例
    - ログイン ＞ 商品選択 ＞ 購入 ＞ ログアウト



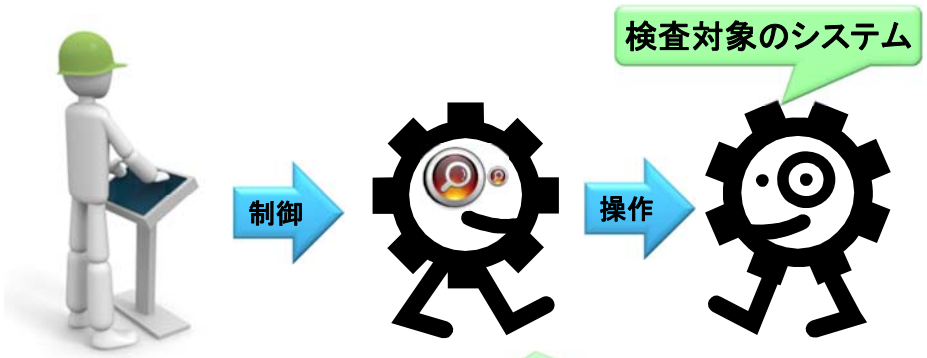
  - 解決策
    - テストスクリプトを分離する
    - バリエーションは作りやすく



キーワード駆動テスト
システムテストに起因する問題

- 準備と結果はシステムだけでは完結しない
  - 問題
    - システムテストの準備や結果判定は、データベースなどの外部システムも考慮する必要がある
  - 解決策
    - テストの準備に、外部システムも含める
    - テストの結果判定に、システムの状態も含める
    - システムを自動で動作させる部分、準備と結果判定部分を分離する

キーワード駆動テスト
自動テストシステムの推奨構造



制御システム
PowerShellなどで作成

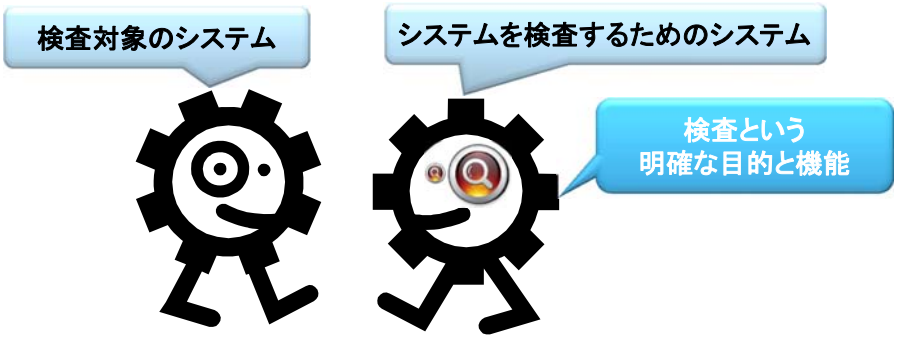
自動操作システム
自動化ライブラリを使用してシステムを構築
テストスクリプトは、各種言語や
テキストファイルなどで記述する

システムテスト自動化のアーキテクチャ

1.システムテスト自動化の問題
2.システムテスト自動化のアーキテクチャ
3.システムテスト自動化のパターン

システムテスト自動化のアーキテクチャ
自動テストとは

- 自動テストはシステム
  - 自動テストは、検査を実行するシステム
- 自動化されたシステムテストとは
  - システムをテストするという目的を持ったシステム

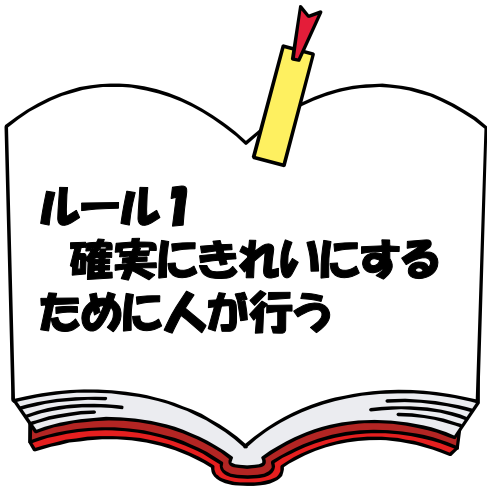


システムテスト自動化のアーキテクチャ
システムテスト自動化の問題

- テストするシステムやプロジェクトによって、自動化の要求は異なる

システムテスト自動化のアーキテクチャ
ルールと指針を決めて、目的にあった自動化

- ルール
  - 判断するための具体的なルールを用意する
- 指針（品質特性）
  - 複数のルール間で整合性を取るためには、指針が必要



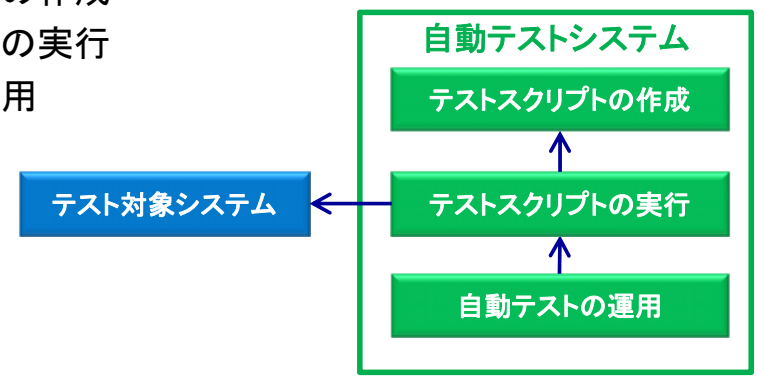
システムテスト自動化のアーキテクチャ
システムアーキテクチャとは

- 品質特性
  - システムとして統一された品質を達成するために、システムとして求められる品質特性と対象（ｽﾃｰｸﾎﾙﾀﾞｰ）を決める
- ルール（仕組み）
  - 複数のメンバー間で統一された品質を達成するために、具体的なルールを用意する
  - 品質特性を強制する仕組みを作る

➢システムアーキテクチャとは
システムの構造や構造化の原則
➢システムアーキテクチャの目的
システムの品質特性を強制（支配）する構造を構築する

システムテスト自動化のアーキテクチャ
自動テストのシステム構成

- 自動テストのシステム構成
  - 二つのシステム
    - テスト対象システム
    - 自動テストシステム
  - 自動テストシステムのサブシステム
    - テストスクリプトの作成
    - テストスクリプトの実行
    - 自動テストの運用



システムテスト自動化のアーキテクチャ
システムごとに異なった関心事

- 自動テストの観点からの関心事
  - テスト対象システム
    - テストしやすいシステムであること
  - テストスクリプトの作成
    - 効率よく作成できること
  - テストスクリプトの実行
    - 確実に実行できること
  - 自動テストの運用
    - 上手にテストの運用を行うこと
- 各システムと品質特性
  - システムごとに異なった品質特性がある

システムテスト自動化のアーキテクチャ
テスト対象システムに要求される品質特性

- 「テスト対象システム」に対する品質特性
  - 操作性（実行性）
    - 自動テストのプログラムから対象のプログラムを実行し、制御できること
  - 確認性
    - 対象のプログラムを実行した結果が正確に確認できること
  - 再現性
    - テストを行う特定の状況を再現できること

システムテスト自動化のアーキテクチャ
自動テストシステムに要求される品質特性

- 「テストスクリプトの作成」に関する品質特性
  - 理解性
    - テストスクリプトが読みやすく理解しやすいこと
    - テストスクリプトを容易に作成することができること
    - テストスクリプトができるだけ多くのメンバーが作成できること
  - 効率性
    - 多様で効果的なテストスクリプトを適切なコストで作成できること
  - 保守性
    - テストスクリプトはテスト対象の変更に素早く対応する必要がある
    - テストスクリプトの変更が容易であること
  - 拡張性
    - テストスクリプトの機能を必要に応じて拡張できること

システムテスト自動化のアーキテクチャ
自動テストシステムに要求される品質特性

- 「自動テストの実行」に関する品質特性
  - 安定性
    - テストスクリプトを安定して実行できること
    - 同じテストスクリプトを何度でも安定して実行できること
  - 信頼性
    - テスト結果の判定が信頼できること
  - 移植性
    - 異なった環境でテストを実行できること

システムテスト自動化のアーキテクチャ
自動テストシステムに要求される品質特性

- 「自動テストの運用」に関する品質特性
  - 効率性
    - 必要な自動テストを効率的に運用することができること
  - 柔軟性
    - 状況に合わせて、柔軟な運用ができること
  - 障害許容性
    - 対象のプログラムで予測しないエラーが発生しても、他のテストスクリプトが継続的に実行できること
  - 並列性
    - 複数の環境で効率的にテストを並列して実行できること

まとめ
システムテストの自動化

- 目的を明確にする
  - テストしたいシステムやプロジェクトによって目的は異なる
- ルールと指針（品質特性）を決める
  - ルールと指針によって、的確な自動化が行える
- アーキテクチャを構築する
  - 対象システム、自動テストシステムは異なった指針が必要
  - 品質特性を強制するものがアーキテクチャ

まとめ
システムテストの自動化について

1.システムテスト自動化の方針
2.各工程で行うべきこと

システムテストの自動化について
システムテスト自動化の方針

- エンドツーエンドを目指す
  - 100%のエンドツーエンドでなくても、有効と判断できれば自動化する
- カイゼンを継続する
  - 自動化できる部分から自動化を開始する
  - 自動テストからフィードバッグを得る
  - テストの効果を向上させるためには、カイゼンが必要
  - 理想と現実を近づけるためには、カイゼンする
- カイゼン対象に聖域なし
  - 自動化に必要であれば、あらゆる可能性を検討する

> 完全性を待たず、改善を作業に取り入れる
> （完全な状態だけを求めると、十分になる機会を失うことになる）
> XP第2版

システムテストの自動化について
開発の初期段階で行うべきこと

☑開発ガイドラインに「テストの自動化を追求する」と記述する

- 理由
  - メンバー全員が最初からテストの自動化に取り組むことを宣言するため
  - システムテストを効率的に自動化するためには、様々な障害が内在する、それをアイデアで乗り越えるため
- やるべきこと
  - テスト容易性を高くすることに注力する
  - テストを自動化するために必要であれば、プログラムの内部的な機能として実装する（エンドユーザーには、必要ない機能であっても）

システムテストの自動化について
分析・設計段階で行うべきこと

☑テスト容易性を追求した設計を行う
☑設計の基本「Don't repeat yourself」を守る（重複を避ける）
☑GUIでしか実行できない機能をできる限り作らない

- 理由
  - テスト容易性を追求することで、設計品質が向上する
  - システムの重複が多いと、同じようなテストをたくさん用意しなければならなくなる
  - GUIを使用しないAPIを用意できれば、自動化が容易になる
- やるべきこと
  - TDDを導入し、内部品質を向上させる

システムテストの自動化について
システムテストの作成について

☑テストケースは、メンバー全員で作成する

- 理由
  - システムテストは、様々な視点で行うことが有効
  - メンバーの様々な視点でテストケースを作成する
  - システムテストでは、準備や結果比較も容易ではない
    - 様々なツールや技術を組合わせることが必要となる
- やるべきこと
  - テストケースの記述方法の抽象度を上げる
    - テストケースがテスト目的の視点で記述できる
  - テストケースの作成コストを低くする

システムテストの自動化について
システムテストの運用で行うべきこと

☑自動化したテストを効率的・効果的に運用するためにはコストが必要
☑テスト結果からフィードバッグで運用を改善する

- 理由
  - システムテストの実行には、時間が掛かるため、効果的に運用するためには様々な工夫が必要となる
  - 自動化はゴールではなく、新たな自動化へスタート
  - テスト結果からのフィードバックには、様々な情報が含まれている
- やるべきこと
  - テストケースを色分けして、実行計画を作成する
  - テストの実行は、夜間や休日を有効に利用する
  - テスト結果を分析し、新たなテストケースを作成する

システムテストの自動化について
まとめ

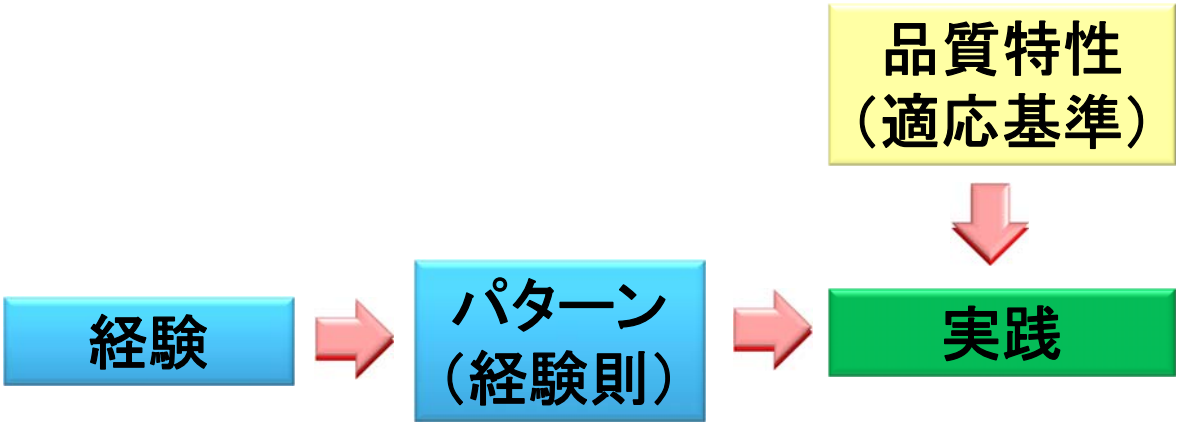
☑自動化を目的にしない
☑システムテストをすべて自動化しない

- 理由
  - テストすることが目的ではない
  - 人の手によるテストは非常に重要である
  - テストの自動化にはコストが掛かるので、効果のあるところから自動化を開始する
  - テスト結果を予想することが難しいのでカイゼン（再計画）が有効
- やるべきこと
  - システムのリスクを考慮し、自動化することで効果のあるところから自動化に取り組む
  - できるだけ早い段階から開始し、システムに対するフィードバックを得る機会を作る

補足資料
パターン集

1.「自動化されたシステムテスト」のパターン
2.「テスト対象システム」のパターン
3.「テストスクリプトの作成」のパターン
4.「テストスクリプトの実行」のパターン
5.「自動テストの運用」のパターン

経験とパターン



「自動化されたシステムテスト」のパターン

1.「目的はChecking」

「自動化されたシステムテスト」のパターン
「目的はChecking」

- 問題
  - 自動化したシステムテストの良い使い方が分からない
  - システムテストを自動化してコスト的に問題ないかわからない
- 解決策
  - 自動化したシステムテストは、ある時点やある環境と現在や他の環境とを比較して、壊れていないことを確認するものである
  - 比較する状況の発生と回数と自動化するコストを比較してコストメリットを考える
- 効果
  - 的確に壊れていることや問題があることが発見できる
  - Checking以外のテストに人手を回すことができる

「テスト対象システム」のパターン

1. 同期で操縦
2. ソフトウェアで処理するためのID

「テスト対象システム」のパターン 「同期で操作」

- 問題
  - テストしたい処理の処理時間が不定期で、事前に待ち時間を想定できない
  - 長い待ち時間で実行すると無駄な待ちが発生して、実行時間が長くなってしまう
- 解決策
  - テスト対象のシステムに同期通信で外部から処理を呼び出す機能を実装する
- 効果
  - 処理時間を気にしないでテストスクリプトを作成できる
  - 待ち処理が必要ないため、高速にテストを実行することができる
- 注意
  - 確認ダイアログなどが表示されると処理が止まる

「テスト対象システム」のパターン 「ソフトウェアで処理するためのID」

- 問題
  - テスト結果を取得したいのにIDがなく処理できない
  - IDが実行ごとに異なりテストスクリプトで処理できない
- 解決策
  - 結果を表示するような場合でも、必ずIDを用意する
  - 固定的なIDを用意する
    - 自動テストだけで利用する連番IDなども検討する
- 効果
  - 自動テストで操作しやすく、確実に実行できるようになる
  - テストスクリプトの共通化が可能となる
- 注意
  - 人間が理解しやすいIDとソフトウェアが処理しやすいIDは異なることが多いため、複数のIDが必要となる

「テストスクリプトの作成」のパターン

1. シナリオをユースケースで分析
2. 粒度の調整
3. テストスクリプトに共通変数
4. 正しいのは以前の自分
5. 時間への挑戦

「テストスクリプトの作成」のパターン 「シナリオをユースケースで分析」

- 問題
  - テストスクリプトに重複部分が多く、作成や保守コストが掛かる
  - テストスクリプトを作成するために、テストシナリオを分析したいが方法わからない
- 解決策
  - 似ているテストシナリオを基にユースケースを作成し、ユースケースごとに、テストスクリプトの作成方法を決定する
- 効果
  - 重複がなくなり、作成や保守コストを下げることができる
- 例
  - ECサイトでの商品の購入
    - ユースケース的には、以下の3つから構成される
      - ログイン ＞ 商品の選択 ＞ 決済
    - 「ログイン」と「決済」は重複されることが多い、また、処理フローはある程度限定される
    - 「商品の選択」は検索やキャンセル、個数の変更などバリエーションが多い

「テストスクリプトの作成」のパターン 「粒度の調整」

- 問題
  - テストスクリプトを作成できるメンバーが限定される
  - テストスクリプトの品質が安定しない
  - テストスクリプトの作成と保守コストが掛かる
- 解決策
  - テストスクリプトを複数の粒度で実装できるようにする
  - 粒度は以下のようになる
    - ドメイン言語 ＞ スクリプト言語 ＞ プログラム言語
    - 例：タブ区切りのテキスト ＞ PowerShell ＞ C#
- 効果
  - バリエーションが必要な部分の粒度を大きくすることで、テストスクリプト全体の作成と保守コストを下げることができる
  - 粒度の大きい部分の作成は習熟度があまり必要ないためチームメンバー全員が作成することができるようになる
  - 粒度の大きい部分は、ツールとの組み合わせで自動生成が可能である

「テストスクリプトの作成」のパターン 「テストスクリプトに共通変数」

- 問題
  - 環境を変えるとテストスクリプトが正しく動作しない
  - 環境に合わせてテストスクリプトを変更すると保守コストが爆発する
- 解決策
  - テストスクリプトに共通変数を導入する
  - 環境に依存する部分は、テストスクリプトではなく共通のファイルに定義する
  - 自動テストの実行時に、共通定義内の値でテストスクリプトを変更する
- 効果
  - テストスクリプトを書き換えることなく、待ち時間や実行年月日などを実行時に設定することができる
  - 環境ごとのテストスクリプトを作成する必要がなくなる

「テストスクリプトの作成」のパターン 「正しいのは以前の自分」

- 問題
  - システムテストの正しい結果を事前に用意するのにコストが掛かる
  - 実行できたかどうかだけで、成否を判断している
- 解決策
  - ある時点、ある環境の結果を正解とする
  - 検査時は、検査の結果と正解を比較して、成否を判断する
- 効果
  - 実行結果を正解とすることで、容易に結果を用意できる
  - 正解とした結果の内容を確認しないと、間違った結果を正解としてしまう可能性がある

「テストスクリプトの作成」のパターン
「時間への挑戦」

- 問題
  - システムで利用する時間が実行ごとに異なるため、結果が必ず異なってしまう
  - システムの環境によって実行時間が異なるため、安定して実行できない
- 解決策
  - 時間取得部分を一か所にして、モック化する
  - 結果判断から時間を除外する
  - 同期で処理を行う
- 効果
  - システムにより結果判断が可能となる
  - 安定した自動テストの実行が可能となる

「テストスクリプトの実行」のパターン

1. 非同期で操作
2. スクリプトで実行
3. 結果はテキストファイル

「テストスクリプトの実行」のパターン 「非同期で操作」

- 問題
  - テストしたい処理の中で、確認メッセージや確認入力があり、同期では処理が止まってしまう
- 解決策
  - APIなどを使用して、テスト対象のシステムを非同期で外部から処理を呼び出す
  - 確認処理の呼出し部分を非同期で行う、確認処理は同期で行う
  - 確認処理を同期で処理することで、状態の変更待ちなどが可能となる
- 効果
  - 同期では止まってしまう処理を実行できるようになる
- 注意
  - 待ちを時間で行う場合、事前に時間を想定する必要がある

「テストスクリプトの実行」のパターン 「スクリプトで実行」

- 問題
  - システムテストを実行するためのデータベースなどの設定を行えない
- 解決策
  - システムテストの実行に、スクリプト言語を導入する
  - スクリプト言語を利用して、データベースなどの設定を行う
- 効果
  - スクリプト言語は、システム間の連携のための機能が豊富で、容易に他システムとの連携ができる

「テストスクリプトの実行」のパターン
「結果はテキストファイル」

- 問題
  - システムテストの実行結果の判定が自動化できない
  - システムテストの実行結果が環境などに依存してしまう
- 解決策
  - 画面の値を文字列として取得できる機能を用意する
  - 画面を画像として残すのではなく、テキストファイルで保存し、期待値のファイルと比較し判定を行う
- 効果
  - テスト結果の比較制度が上がり、的確な判定を行うことができる
  - 日付や時間などを除外して比較することができる
  - 期待値を事前に用意することができる

自動テストの運用」のパターン

1. 自動テストの運行スケジュール
2. 自動テストの並行実行
3. 自動テストのランキング

「自動テストの運用」のパターン 「自動テストの運行スケジュール」

- 問題
  - テストの実行時間が長く、効果的なテストの実行ができない
- 解決策
  - 夜間や休日を考慮して実行スケジュールを作り、運用する
  - 全てのテストスクリプトを実行対象とするのではなく、変更に関連したテストスクリプトを選択する方法を確立する
- 効果
  - 開発作業を行っていない間に実行することで、効果的なフィードバックが可能となる
- 注意
  - テストスクリプトの実行でエラーが発生することを想定した運用環境を作る

「自動テストの運用」のパターン 「自動テストの並行実行」

- 問題
  - テストの実行時間が長く、効果的なテストの実行ができない
- 解決策
  - テストセット（複数のテストスクリプト）単位で並列実行できる環境を構築する
  - 実現例
    - キューシステムを導入し、同じテストセットを実行しないようにする
    - 成功リスト、エラーリスト、実行中リストを用意して、並列実行を行う
- 効果
  - 実行時間のトータル時間より短い時間でのフィードバックが可能となる
- 注意
  - 各テストセットの実行時間をできるだけ均一とする
  - テストセット間の関係をなくし、テストセットの独立性を実現する

「自動テストの運用」のパターン
「自動テストのランキング」

- 問題
  - テストスクリプトの数が多く、効果的なテストの実行ができない
- 解決策
  - テストスクリプトにランキングを付けて、実行に強弱を付ける
  - ランキングの正しさを判断する基準を用意する
- 効果
  - 効果のあるテストを優先的に実行することができる
- 注意
  - ランクの低いテストスクリプトを実行していれば、見つかっていた問題を見逃してしまう可能性がある

参考文献

- システムテスト自動化標準ガイド
  - Mark Fewster, Dorothy Graham著　テスト自動化研究会　訳
- XP　エクストリーム・プログラミング入門 変化を受け入れる
  - Kent Beck
- 実践テスト駆動開発
  - Steve Freeman・Nat Pryce
- トヨタ生産方式　脱規模の経営をめざして
  - 大野 耐一