# システム操作インターフェイス最適化によるテスト自動化 ROI 向上

Improvement in test automation ROI by system manipulation interface optimization

石川達也

Ishikawa-tatsuya@codeer.co.jp

株式会社 Codeer

**発表要旨：**

本発表ではシステムテストを自動化する際に、アプリケーションをテストプログラムから操作するインターフェイスを最適化する手法と、それにより得られる効果を述べる。

一般的にシステムテストの自動化は、フロントエンドのアプリケーションの GUI 入力をエミュレートすることが多い。しかし、GUI は人間用に設計されており、プログラムからの操作に適していない。実際に問題になった点を挙げる。

・不安定なテストになる。(タイミング依存で成否が変わる。)

・技術的にブラックボックス性が高く失敗理由の解析が困難。

・プロダクト変更時のメンテコストが高い。

・テストシナリオの可読性が悪い。

・操作不可能なケースも多々ある。

問題が積み重なると、解消するためのコストが増大する。逆に、これらの問題が発生しないインターフェイスが用意できれば、十分な ROI を確保できるといえる。

本発表では、操作用インターフェイスとして GUI だけでなく API も利用する方法と、その際、ユーザー操作とは異なるインターフェイスを使うことに関してのトレードオフを実例を交え紹介する。また、テストシナリオから使うインターフェイスを洗練させる手法としてアプリケーションドライバ(Web でいうところの PageObject)も合わせて紹介する。

**キーワード：**

システムテスト自動化、テストプログラム設計、アプリケーションドライバ、システム公開 API

**想定している聴衆**

システムテスト自動化に興味を持つ人、Windows アプリ開発関係者

**発表者の紹介**

株式会社 Codeer 代表取締役

Microsoft MVP for C#

Windows アプリテスト自動化歴 9 年

Windows アプリ操作用ライブラリ Friendly 開発者

# システム操作インターフェイス最適化による テスト自動化ROI向上

株式会社Codeer

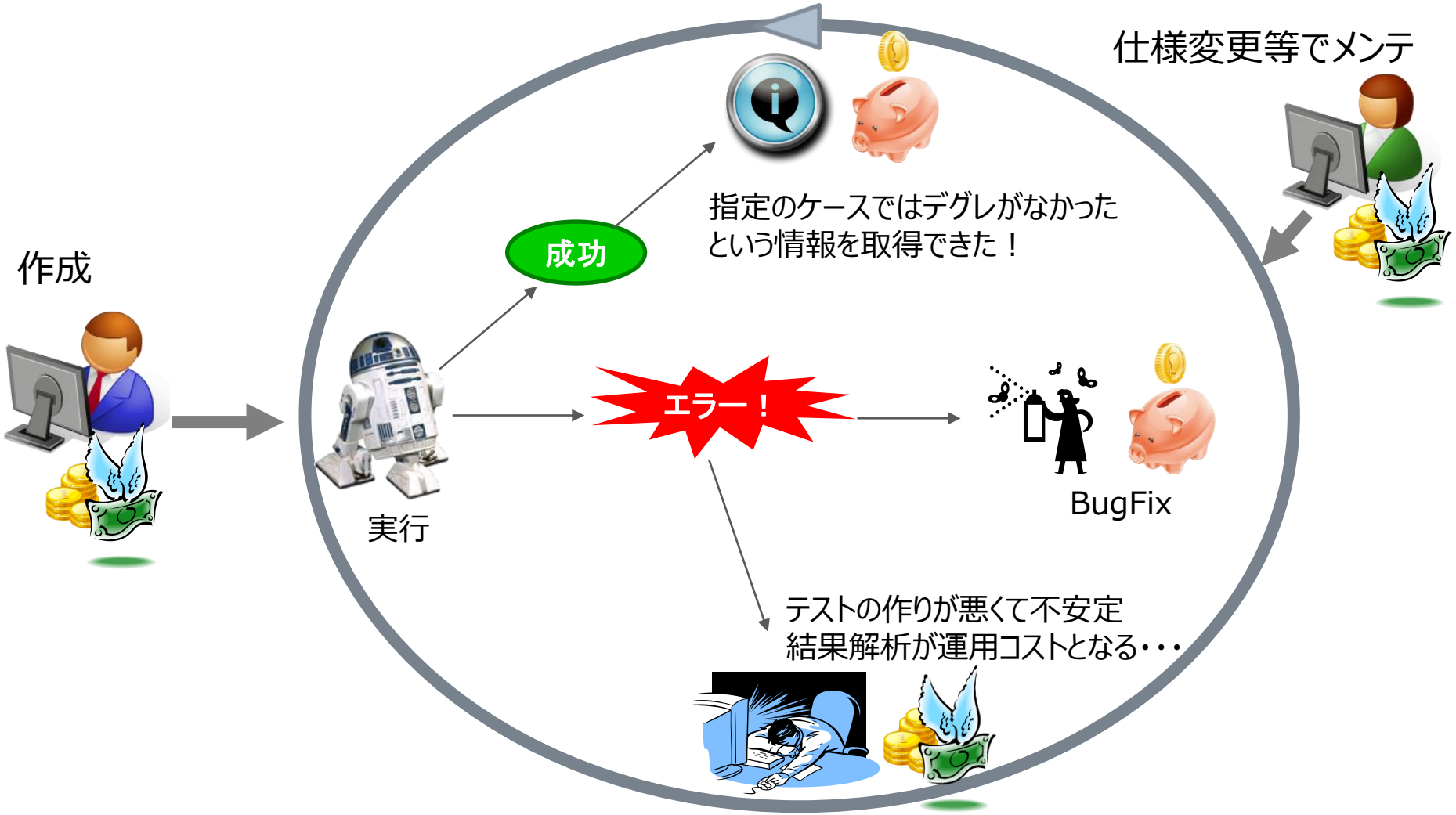
〇石川達也

e-mail：ishikawa-tatsuya@codeer.co.jp

# ご相談を受けた企業様の悩みで多いもの

システムテスト自動化
やったことあるんだけど、
効果が出なくて・・・

# 作業とROI要素を分析



# 課題①　自動実行不安定

運用コストが想定以上に発生している。
回帰テストなので、ランニングコスト化。

エラー！

モチベーション低下
テストに対する不信感も

テストの作りが悪くて不安定
結果解析が運用コストとなる・・・

# 課題②　自動化テストケース数が少ない

技術、工数の都合で
作れるケースに限界がある

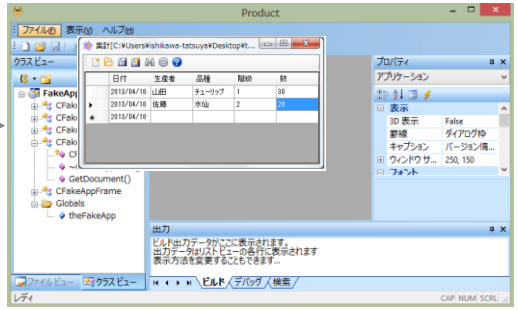
成功

指定のケースではデグレがなかった
という情報を取得できた！
→**でも・・・**

テスト成功時の情報の価値が低い
・デグレ検出できないケースが多い。
・成功しても次のアクションに繋げづらい。
(リファクタリング、リリース判断等)

# 課題①②の共通問題　対象アプリ操作インターフェイス

ココ



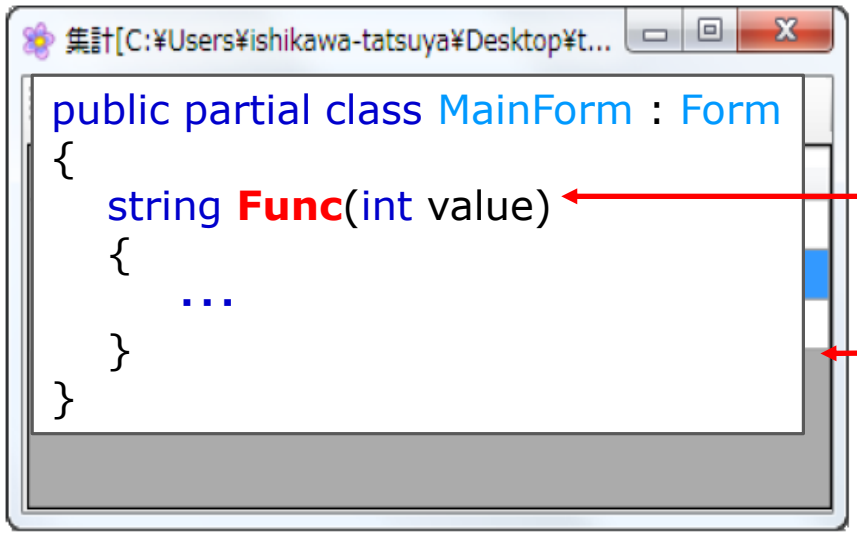

「簡単に」「思い通りに」
操作するためのインターフェイスがあれば、
多くのテストケースを自動化できる。
また、安定して動作させることも可能。

Friendlyを導入

http://www.codeer.co.jp/AutoTest

# Friendly紹介

基本機能
・内部メソッドを実行させる
・DLLインジェクション

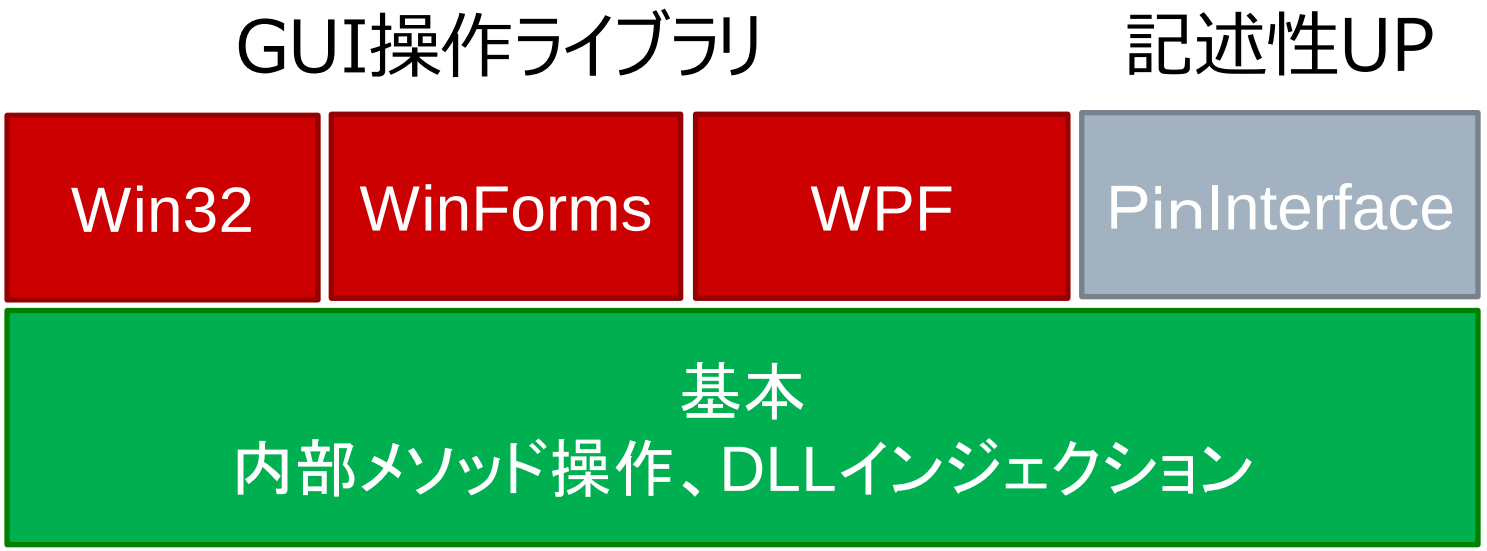



```
//テストプロセス
public void Test()
{
  //呼び出し
  form.Func(3);

  //Dllインジェクション!
  WindowsAppExpander.
    LoadAssembly(app,GetType().Assembly);
}
```

内部メソッドを呼び出せるので、操作自由度が高い。
単体テストのように、プロダクトコードを変更して
テスタビリティーを向上させる(操作性を向上させる)ことが可能。

# Friendlyの上位ライブラリ紹介




Friendly.Windows.NativeStandardControls
Friendly.FormsStandardControls
Friendly.WPFStandardControls
Friendly.PinInterface

安定性した操作が可能。
高い拡張性がある。

# 対策　GUI操作をFriendlyの上位ライブラリに変更

| 操作対象 | メリット | デメリット | 技術 |
|---|---|---|---|
| ①ＯＳ層<br>（Key，Mouse） | ・ある意味万能<br>・ユーザー操作に最も近い | ・不安定<br>・低速 | ・SendInput |
| ②GUIコントロール層 | ・安定<br>・高速<br>・リッチなインターフェイス | ・Key,Mouseとの結合は<br>カバーしない<br>・提供されていない<br>コントロールもある | ・Friendly<br>・UIAutomation<br>・一部のWin32API |

・テストの主目的はOSと各GUIライブラリの結合ではない
・安定性、実装容易性を重視

課題①②へ効果

# 対策　操作可能なGUIを増やした

Friendly上位ライブラリがサポートしていないコントロールに関して、
（サードパーティー提供コントロール、自作コントロール）
通常のプログラムに使用するインターフェイスとFriendlyを使って、
操作ライブラリを作成。

| 各プロジェクトのコントロール使用状況 | |
|---|---|
| プロジェクトA | 全て標準。 |
| プロジェクトB | 表に関して、サードパーティーのコントロールを使用。 |
| プロジェクトC | 9割以上の画面にサードパーティーもしくは自作のコントロールがあった。 |

→ B、Cでは、これにより安定した操作を
簡単に実装できるようになった。

課題①②へ効果

# 対策　自動化困難で効果の低い部分の分離

```
//ここの結合は不安が少ない
void Event(object sender, EventArgs e)
{
   EventCore(PointToClient(Control. MousePosition));
}
//これをFriendlyで呼び出す
void EventCore(Point mousePosClient)
{
  //ここから先のロジックをテストしたい。
}
```

プロダクトを変更。
効果の高い部分のみ
呼び出せるようにした。

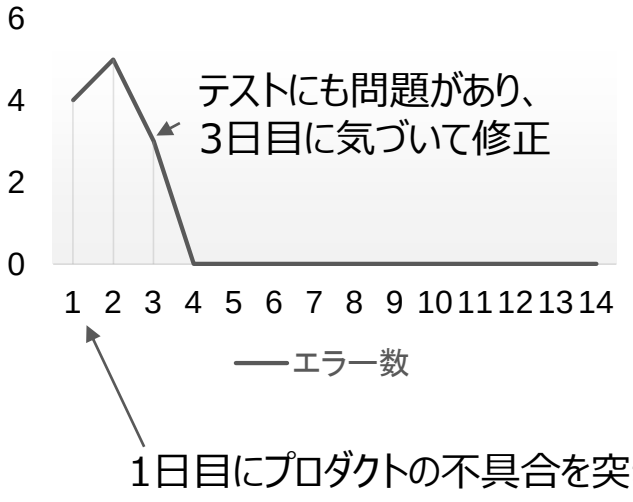
・キー、マウス直接参照
・D&D
・OS提供のGUI
etc…

課題②へ効果

# 対策　ユーザー実装による不安定の対応

| 原因 | 対応 |
| --- | --- |
| タイミング依存の不具合 | 不具合修正 |
| タイマ、非同期処理、<br>描画イベントを利用した実装 | テスト時は同期にする |
| | 確実に同期がとれる情報を参照する |

例) 不安定な処理の周辺のテストを86ケース投入してから安定するまで

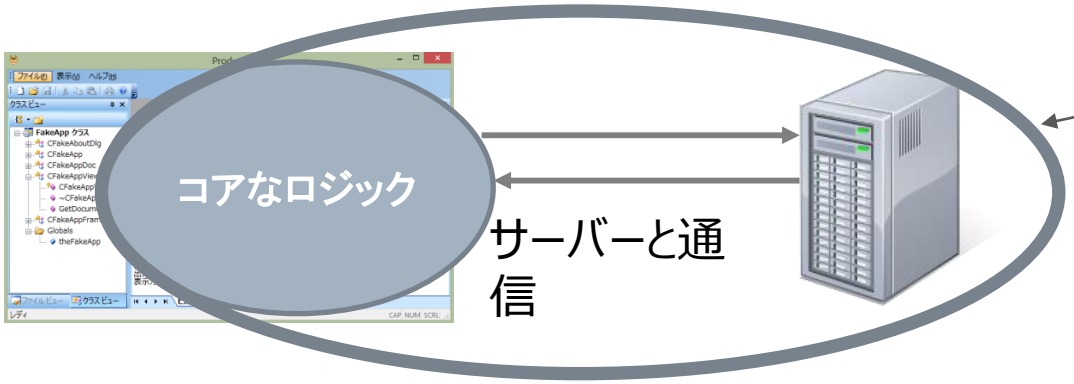



CI環境投入前に単体では
全ケース成功することを確認している。

プロダクト自体タイミング依存の不具合があったので修正。
テストも安定性を考慮した記述方法に変更。
4日目からはこのテストの運用コストは発生していない。

課題①へ効果

# 対策　実装工数が大きいものはGUI操作を省略



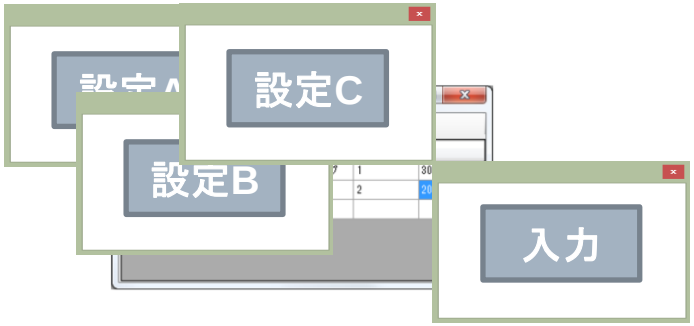

例）
システムテスト自動化を実施したいが、
GUI操作で、実行するには困難で諦めていた。

しかし、分析の結果、品質を確認したい部分は
GUIではなかったので、内部メソッドを利用。
約5千ケースが自動化された。

課題②へ効果

# 対策　実行時間がかかりすぎるものはGUI操作を省略

例）
文字列のパースをする処理があり、
それはアプリの様々な設定によりパース結果が変わる。
しかも、レガシーコード。（単体化が困難）
求められるテストケースは15万ケース。

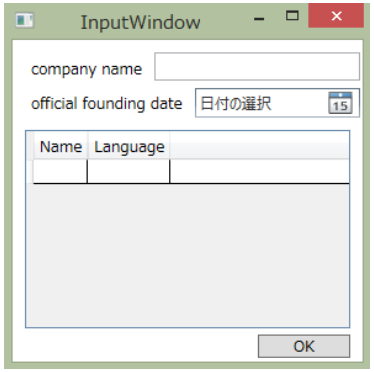



| 種類 | 実行平均時間 | 合計 |
|---|---|---|
| 手動 | 30秒 | 1250時間 |
| 自動GUI操作 | 2秒 | 83時間 |
| 自動内部メソッド | 10ミリ秒 | 24分→Dailyで回帰化可能な時間に！ |

時間がかかりすぎるとDailyの回帰環境に組み込めない。
分散させることも可能。実際そうしたケースもある。
ただ、分散には大がかりな仕組みが必要になる。
シンプルに速くなれば開発者のローカルPCでも実行可能。

課題②へ効果

# 課題③　テスト作成に内部仕様の知識が必要




いくつかテストケースを
実装した段階で気づきがあった。

```
class InputWindow : Window {
  TextBox _name;
  DatePicker _founding;
  DataGrid  _member;
  Button _ok;

  public void Func() { ・・・ }
}
```

```
//テストコード
//変数名を知っている必要がある
var _name = new WPFTextBox(window._name);
var _founding = new WPFDatePicker(window._founding);
var _member = new WPFDataGrid(window._ member);
var _ok = new WPFButtonBase(window._ok);

//内部メソッドを知っている必要がある
Window.Func();
```

# 対策　アプリケーションドライバを導入


①のインターフェイスは、技術的要素がどうしても入る。また、内部仕様も現れる。
そういったものは、テストシナリオからは使いたくない。

| | 記述すること | 実装に必要な知識 |
|---|---|---|
| アプリケーション<br>ドライバ | 内部仕様に依存する処理 | 内部仕様 |
| | 複雑な処理 | C# |
| | | 基本的なプログラム設計能力 |
| テストシナリオ | テストケース<br>（できるだけシンプルに） | 外部仕様 |
| | | 基本的なC#の知識 |

書籍「継続的デリバリー」参照

# 対策　アプリケーションドライバを導入

テストケース作成

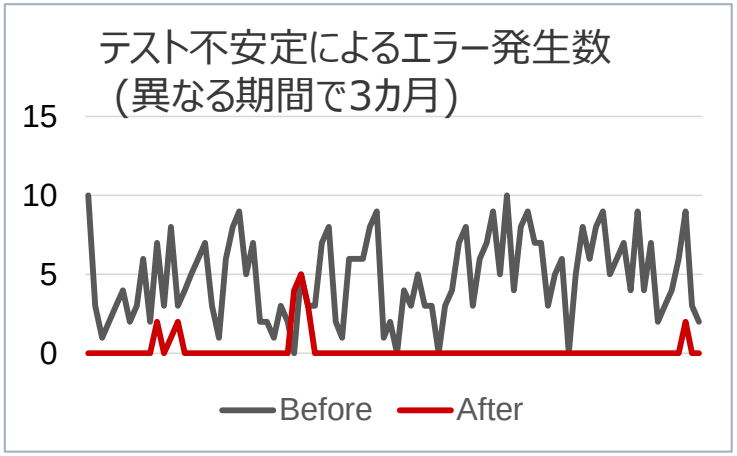
プロダクト開発者以外に負荷を分散できるようになった。
結果、テストケース開発効率が上がった。
また、ケース追加時に安定化された操作を
容易に使いまわせるようになった。

テストケースメンテ

仕様変更、内部実装変更時の対応箇所が最小化された。
コード量が減った。行数で約35%減。
テストシナリオの可読性Up。

課題①②③へ効果

# A社改善の成果




課題①に対して

改善前は、ほぼ毎日結果を解析して、プロダクトに問題がないことを確認する必要があった。

（半自動）改善後は、不安定なテストが混入することはあるが即座に対応して安定化させることができた。


課題②に対して

テストケース数を増やすことに成功。

部分的にリリース、リファクタリング実行の判断に使うことができるようになった。

最終に実施する手動テスト数も減らすことに成功。

# まとめ

・対象アプリ操作インターフェイス

目的に応じて最適な操作方法を選ぶことにより、
運用コストを下げることができた。
また、操作の幅が広がりテストケース数も増やすことができた。

・アプリケーションドラバ

自動化はテストプログラムの作成である。
そのため、設計、実装の品質により作成、メンテコストは変わる。
アプリケーションドライバを利用することにより、
テストプログラムの品質を上げ、
テストケース追加、メンテコストを抑えることができた。
また、作成可能なメンバーが増えたので結果として
テストケース数増加にもつながった。