15-B-4

Friendly による内部 API を使ったシステムテスト自動化 [1]

1. 概要

本編では「Friendly[2] による内部 API を使ったシステムテスト自動化」の適用に関して、Windows アプリ操作ライブラリ Friendly により Windows アプリケーションのシステムテストを自動化した事例を紹介する。

大規模で継続的な開発を続けるアプリケーションでは、システムテスト自動化の必要性が増している。このような開発では、機能追加や不具合修正時の影響範囲が読みづらく、デグレードの危険性はあらゆるところに潜んでいる。そのため品質を保証するには、逸早く問題を検出しフィードバックするシステム、すなわち日々実行される自動化された回帰テストの存在が必要となる。しかし、そこには問題がある。実は人間の代わりに、プログラムにアプリケーションを操作させるということは、一般に思われているよりも、はるかに困難なプラクティスなのである。一般的にはアプリケーション操作にはキャプチャリプレイツールによるキーマウスエミュレート（図 15-B-4-1 参照）や UI オートメーションなどの UI 操作専門のライブラリ（図 15-B-4-2 参照）を使うことが多いのだが、少なくとも筆者は、それらでは効果的なテスト自動化を作成することができなかった。

そのためここでは、それらとは異なるアプリケーション操作技術として、内部 API（Application Programming Interface）（図 15-B-4-3 参照）をシステムテスト時にも用いる方法を紹介する。

内部 API は文脈により様々な意味を持つが、本編ではアプリケーション実装時に使用、作成するインターフェイス（フィールド、プロパティ、メソッド）を指す。これらは通常そのアプリケーションを実装するときにのみ使用可能なもので外部からは利用できない。しかし、テスト時にこれを使うことができれば、一般的な GUI の操作性向上はもちろん、コンポーネントテストで行うようなテスタビリティーを向上させる手法も使用可能となる。

Friendly は、まさにこの効果を狙うために作ったライブラリで、別プロセスの内部 API を使用可能にするものである。

実際の適用プロジェクトでは、日々数万件のシステムテストを安定実行させ、デグレード防止に力を発揮している。以降では、その具体的な活用方法も交え詳細に説明する。

---

[1] 事例提供: 株式会社 Codeer　石川 達也 氏

[2] Codeer 社が開発し OSS で公開している無料の Windows アプリ操作ライブラリ。
http://www.codeer.co.jp/AutoTest　より詳細なドキュメントを参照可能。

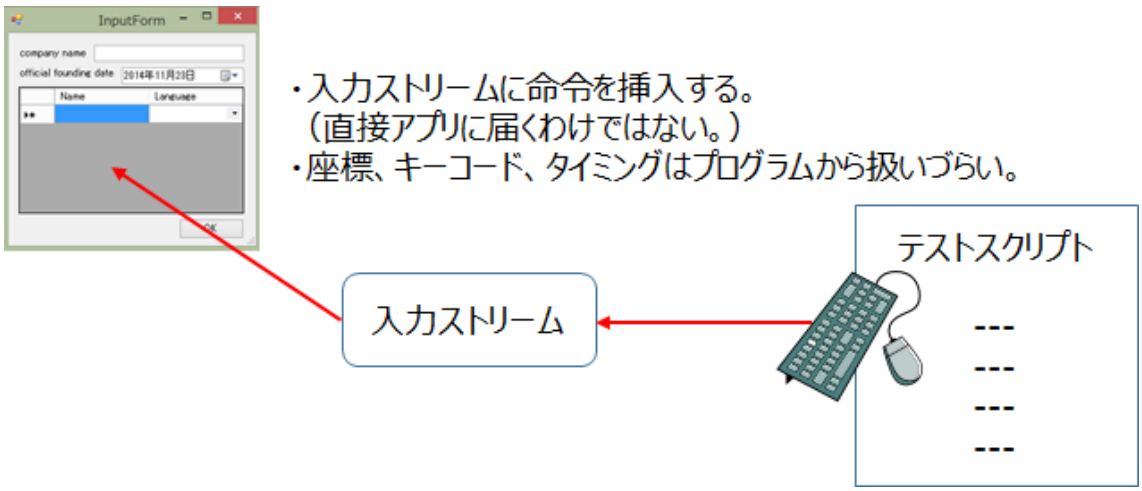

図 15-B-4-1　キーマウスエミュレートイメージ

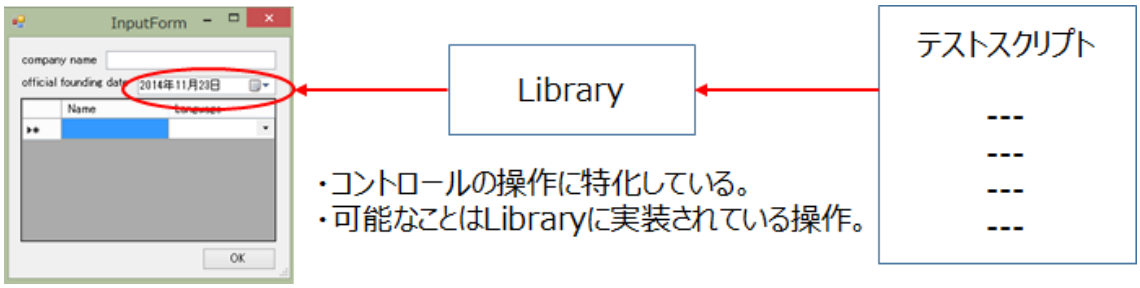

図 15-B-4-2　UI 操作ライブラリイメージ

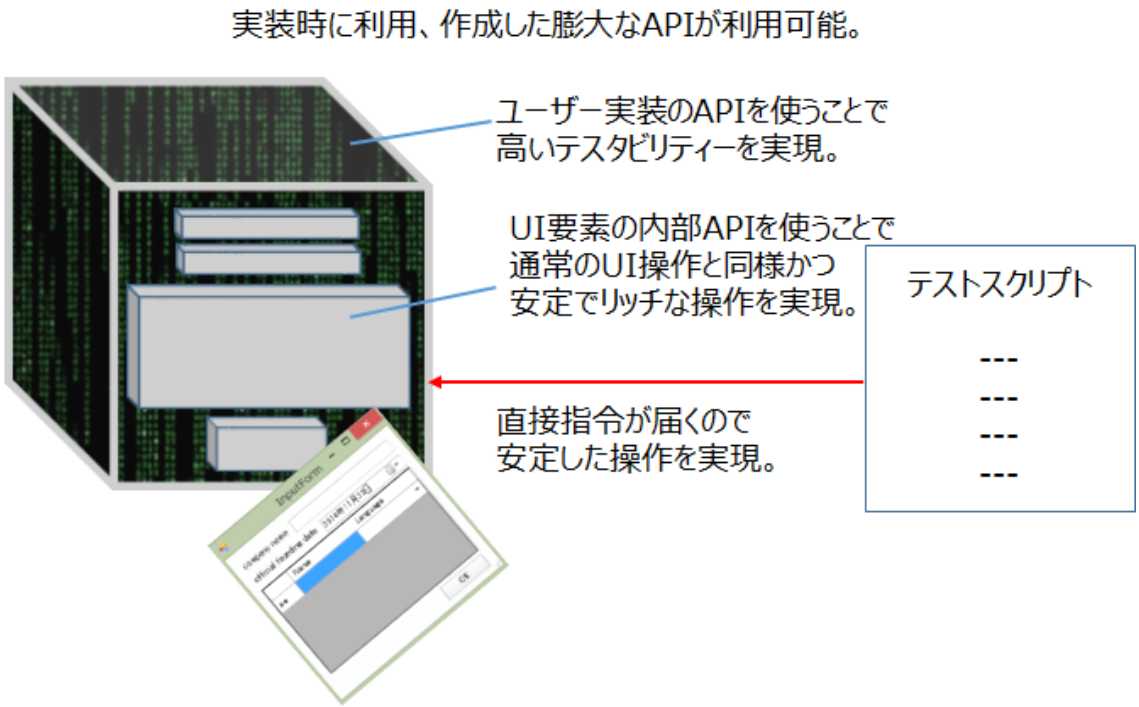

図 15-B-4-3　内部 API イメージ

2. はじめに

筆者は数年前から Windows アプリケーションのテスト自動化に取り組んできた。その中で如何にコストパフォーマンスの高いテスト自動化を実現するかを考え、内部 API を使う方法にたどり着いた。そして 2012 年には、それを汎用的に実現するためのライブラリ Friendly を開発、テスト自動化の普及に努めている。本編で解説する事例の環境は以下のものとなる。

- 対象のシステムは Windows アプリケーション
- システムテスト自動化プログラムの開発言語は C#[3]
- システムテスト自動化プログラムの開発環境は Visual Studio[4]
- 内部 API を外部から実行するために使用するライブラリは Friendly

本編で紹介する例は、上記のような環境であるが、この内部 API を使用するテスト自動化手法は様々な領域に適用できる可能性がある。Windows アプリケーション開発者以外も、それぞれのコンテキストに置き換えながら読んでいただければ幸いである。

3. 問題

大規模で継続的なアプリケーション開発ではテスト自動化、とりわけデグレードを防ぐための回帰テスト自動化の必要性が増している。

長い開発期間中には、多くの機能を追加、変更する必要があり、その際に共通モジュールの変更も行われるが、そこには常にデグレードの危険性が潜んでいるのである。

デグレードが大きなリスクとなるのは開発後期にそれが顕在化した場合である。以下のような状況が考えられる。

- 混入した時期が不明のため原因の特定が困難である
- デグレードの原因となったロジックを前提に次のロジックが書かれてしまった
- 既にその周辺の検証の多くが完了していて、修正した場合の修正確認コストが大きい

デグレードは、単なる不具合とは異なった性質を持っている。それは何らかの機能を追加、もしくは修正した影響で入った不具合であることだ。そのため、単純に修正したのでは改善のあった別の機能に深刻な影響を与える可能性がある。修正の前に混入時期や、その変更の意図を正確に把握しなければならない。また、往々にしてデグレードが発生するコードは影響範囲が大きい。（もともと別の機能追加、修正の影響を他の機能に与えるぐらいなので）より一層慎重な修正が必要となる。

では、デグレードを無くすことは可能なのだろうか。おそらく、それは困難であると思われる。ただ、混入した後すぐにそれを検出することができれば、少なくとも前述したデグレードによるリスクを大きく軽減できると考える。

本編で論じるテスト自動化は、この問題への対策を主な目的としている。

---

[3] Microsoft 社が開発したプログラミング言語

[4] Microsoft 社が開発した統合開発環境

4. 回帰テストに求められる要件と技術的な課題

デグレードのリスクを軽減するために、回帰テストが満たすべき要件はなんであろうか。適用事例のプロジェクトでは以下のものと定義した。

- リスクを軽減できるだけの質と量を備えたテストケースが実装されていること
- デイリーで回帰テストを実行可能であること

本項ではこれらについて考察し、そこに潜む技術的な課題を明らかにする。

4.1. リスクを軽減できるだけの質と量を備えたテストケースが実装されていること

たとえ回帰テストを自動化したとしても、そのテスト内容が適切なものでなければ意味はない。(目的が不適切、ケース数が少なすぎて成功しても品質の判断に使えない等)

実はこの要件は、テスト自動化を実践しているプロジェクトでも、実現できていないところも多い。テストケースの設計自体は手動でも実践しているので、既に各プロジェクトにノウハウは溜まっている。その上で、自動化ができないのは、人間が実行している「操作」「判断」をプログラムに置き換える「技術」が、ボトルネックになっているためである。そのため、本当に自動化して日々品質を確保したいテストケースと、実際に自動化できたケースが同じにならないのである。

4.2. デイリーで回帰テストを実行可能であること

せっかく自動化したテストであるので一日一回は実行したいものである。仮にデグレードが発生しても次の日に検出できれば、リスクは少なく修正できる。

しかし、ここにも問題はある。それは、日々不特定のテストが失敗する不安定な自動テストを作りこんでしまった場合で、そのような状況では失敗原因の調査コストが膨大になり、結果的に運用コストが増大するのである。

また、もっと致命的なのはテスト自体への信頼性がなくなっていくことである。失敗しても、まともに調査しなくなるし、テスト作成、保守へのモチベーションが下がる。

最悪の場合は、不具合検出率の低さとも相まって、逆にプロジェクトの足を引っ張る存在に成り下がってしまい、デイリーで実行するどころか、回帰検査を継続することも困難になってくる。

特に、目的はリスクを軽減できるだけの十分な量のテストを実行させることなので、大量のテストを日々安定して実行するための、自動テストで対象アプリケーションをコントロールする「操作技術」が重要となってくる。

また、テストをデイリーで実行するのにはもう一つ重要な意味がある。それはテストの陳腐化を防ぐことである。自動テストは、プロダクトのアプリケーションと協調動作するプログラムである。プロダクトのアプリケーションは開発期間中、日々成長し変化していく。当然､テストに影響がでる変更の場合はその変更をテストコードにも反映させなければならい。これにはチーム間の連携が重要となってくる。そして、その連携が十分にできていることを

確認する意味でも自動テストを日々実行し、追従できていることを確認することが重要である。時には連携がうまくいかずテスト失敗となって表れることもあるが、それが発生してすぐならばメンテナンスも簡単にできる。しかし、長期間放置して、連携が上手くいかない部分が多数溜まってくると、それを正常運用できる状態まで戻すのは困難な作業になると考える。

繰り返すが、日々安定してテスト実行するためには、自動化によるアプリケーションの「操作技術」が非常に重要なのである。

4.3. 技術的課題「自動化によるアプリケーションの操作技術」

本項での考察からわかるように、技術的課題は対象のアプリケーションの操作技術である。ポイントは十分な操作性があること、安定していることである。次項からはアプリケーション操作技術に関して考える。

5. 一般的なテスト自動化のアプリケーションの操作方法に対する考察

一般的に Windows アプリケーションを外部から操作するために使われる手法は、キーマウスエミュレートか UI オートメーション（もしくはその両方）になる。筆者はこれらの手法では前項で定義した自動化回帰テストへの要件を満たすことができなかった。その理由に関して考察してみる。

5.1. キーマウスエミュレート

キーマウスエミュレートはプログラムから人間の代わりにキー、マウスの信号を送る操作方法である。これは、一見理に適っているようだが、いくつかの問題を含んでいる。

まず、安定動作させることが非常に困難になってくる。キーマウス操作と言うのは、あくまで人間向けに提供されている操作方法であり、プログラムからこれを実行するのは困難であるのだ。なぜなら、命令を入力する先は OS 層になり、直接対象アプリケーションに働きかけられるわけではない。ここに不安定さが入り込む余地がある。また、プログラムと人間では、その速度感が違いすぎる。高速に動作させると、不都合が生じる場合も多い。

さらに、その自動化プログラムは可読性、メンテナンス性を欠いたものになる傾向がある。特にキャプチャリプレイツールを使った場合はそれが顕著になる。メンテナンス性を欠いた大量のテストプログラムを長期間保守していくのは正しい選択とは言えない。

5.2. UI オートメーション

UI オートメーションは、UI 要素をプログラムから操作するためのマイクロソフト標準のインターフェイスである。元々プログラムからの使用を目的としたインターフェイスであるので前述の問題はクリアできている。しかし、難易度は若干高い。また自由度は低く対応していない UI 要素も多い。特に各プロジェクトで UI 要素を自作する場合、これへの対応を入

れることは、現実的には稀であろう。

5.3. 共通の問題

これらの操作方法に関して、共通することは操作インターフェイスの制約が大きいことである。例えば、何らかのコントロールの操作が困難であれば、それがボトルネックとなり、本来やりたいテスト自体が自動化できなくなってしまう。

もう一つ、視点がブラックボックス的なのである。別プロセスのアプリケーションを思い通りに操作したい場合、本来相手の特性を知ってプログラムを書く必要がある。相手のアプリケーションが要求を受けた後どのように動作するのか知っていなければ安定して操作することはできない。

特に GUI というのは、通常は人間が操作することを前提に作られる。これはサードパーティーから提供された UI 要素だけでなく、自分達で画面を実装する時も同じである。これをプログラムの速度で操作した場合、その反応がブラックボックスでは読みにくい。

キーマウスに対する情報の取得方法は様々であるし、その反応も様々である。イベントに対しても同期で処理するものもあれば、非同期で処理を実行するものもある。タイミングをずらして遅延実行するものもあれば、タイマーを使って一定の間隔をあけてから実行するものもある。

そのような GUI アプリケーションを外部からブラックボックス的な視点を元にした手法で操作してそれを安定させるのは困難であると言える。

6. 技術的課題への対応方針

前項で挙げた課題に対してどのように対応していけば良いのであろうか？結論から言うと、筆者は以下二点の方針で対応した。

- 内部 API を使う
- アプリケーションドライバを使う

6.1. 内部 API を利用するアプローチ

ここで対象をプログラムとして考えることで別の道が開ける。操作するときに、開発時に使った内部 API を呼び出すようにする。内部 API とはプログラムの内部実行で利用されている API を指す。C#などのオブジェクト指向プログラミングで言えば、フィールド・プロパティ・メソッドなどである。これらのインターフェイスに直接アクセスできれば、非常に安定したテストを書くことが可能となり、さらにはテストの対象領域も大きく広がっていく。しかし、通常は別プロセスの内部 API を直接操作することはできない。そのため、これまで Windows アプリケーションのシステムテストでは内部 API を利用するというアプローチは一般的ではなかった。

6.1.1. Friendly を用いた内部 API へのアクセス

Windows アプリケーションという領域に対して、これを解決するために作られたのが、ライブラリ Friendly である。Friendly を用いることで、対象となるアプリケーションに定義された内部 API に外部プロセスからアクセスし、値の取得／設定、操作の実行などを自由に行うことができるようになる。Friendly により、これまで困難であった内部 API を用いたテストが、実現可能になったのである。

6.1.2. 内部 API を用いるメリット

プログラム作成時と同様の API が使えるので、目的の操作を実現するための十分なインターフェイスの利用と安定した操作が可能になる。

これらだけでも十分なメリットだが、もっと大きな価値がある。それは、システムテストにもホワイトボックス的な視点を導入できることだ。内部 API が使えるということは、コンポーネントテストと同様のアプローチが使えることを意味する。つまり、そこで培ってきたテスタビリティー向上のプラクティスが輸入できるのである。

これは、テストの自動化ができる範囲の拡大を意味している。通常の方法では、あるテストケースを自動化しようとして、そこにボトルネックとなる操作があれば、そのテストケースは自動化を諦め手動で実行することになる。しかし、この手法であればボトルネックとなる部分は捨てて、それ以外の箇所のテストは自動化できるのである。

これはパラメタライズしたいテストケースに対して非常に有効である。大部分は自動で実行して、最後にボトルネックとなる操作のみ一回手動で実行するということができる。往々にして、リスクが高く、回帰テストで保障したい部分は「GUI の操作」ではなく、結合したシステムの動作なのである。つまり、「GUI の操作」がボトルネックになるなら、それを外して、別のインターフェイスを使っても十分な効果が得られるのである。

6.2. アプリケーションドライバを利用するアプローチ

この手法は Web アプリケーションでは Page Object[5] と呼ばれている手法に似ている。これは内部 API を使わない場合でも有効なのだが、使う場合は必須となる。

内部 API を使う手法は、対象のアプリケーションの内部の設計、仕様を知っている必要がある。これはどうしても開発チームでなければ有効に使うことができない。しかし、テストケース自体はテストチームで実装することが効率的である。

また、内部 API はホワイトボックス的な手法であるが、このレベルのテストはブラックボックス的に書きたいというジレンマもある。

それを、アプリケーションドライバが解決する。テスト自動化に必要な「操作」の部分と「テストシナリオ」の部分を分離するのである。アプリケーションドライバは「操作」の部分である。ホワイトボックス的な視点を有効に利用して、確実に動作するように実装する。

---

[5] Web アプリケーションのテスト自動化のための設計手法。各ページの内部設計を隠蔽する。

それにより「テストシナリオ」の部分は外部仕様の言葉だけで記述できるようになる。

これはテストメンテナンス上も非常に有効な方法である。テスト自動化のためのプログラムもソフトウェアである。つまり、一般的なソフトウェア同様、様々な設計手法を用いてその品質を向上させるべきである。

ちなみに、テスト自動化は自動化のためのプログラムを作成する作業である。そこには必ずプログラムの専門的な知識が必要になってくる。もちろん、テストチームがそれを持っていても良いのだが、そこは身近にいる開発チームと協力するのが効率的である。それを可能にするのにこの手法は非常に役に立っている。

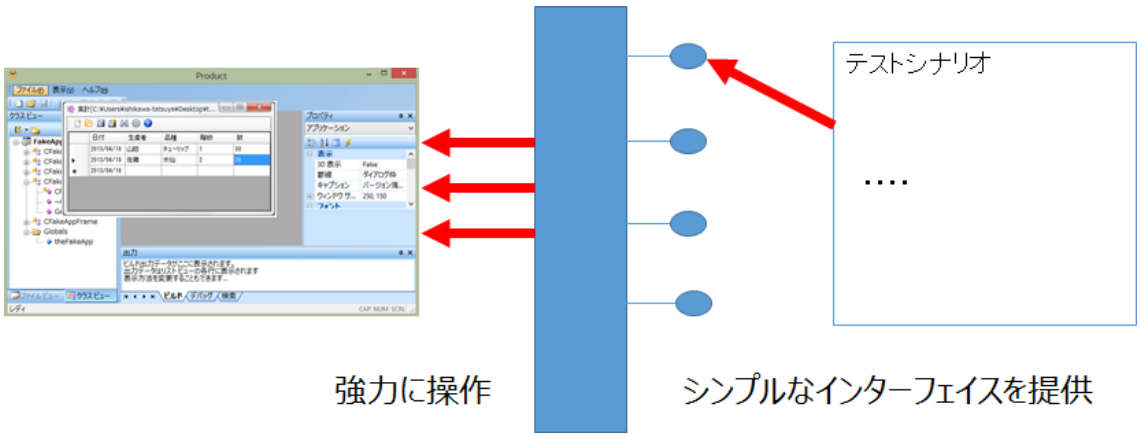

図 15-B-4-4　アプリケーションドライバ

7. 具体的な対応

7.1. アプリケーションドライバの機能

先にアプリケーションドライバ使用法の具体例に関して説明する。テストシナリオから使うのはアプリケーションドライバであり、内部 API ではない。

内部 API は非常に有効な手段ではあるが、それは「操作」というコンテキストに対してである。システムテストのテストシナリオは外部仕様から作成されるべきである。つまり、「操作」を隠蔽したアプリケーションドライバのインターフェイスが使われるのである。

実際にアプリケーションドライバを実装する際に持たせた機能は以下の通り。

- アプリケーションのコントロール機能
- 画面のコントロール機能
- ショートカット機能

7.1.1. アプリケーションのコントロール機能

これも実はさらに３つに分かれる

- アプリケーション起動、再起動、終了の機能
- アタッチ機能

・　フリーズ対策機能

(1)　アプリケーション起動、再起動、終了の機能

先に述べたように、大量のテストを実行する必要がある。その際に 1 ケースごとにアプリケーションの起動、終了、再起動を実行していては実行時間が無駄に伸びる。一度起動した状態で、複数個のテストを実行するようにしたい。しかし、前回のテスト状態に依存して次の実行結果が変わるのは避けなければならない。初期状態に戻すことは必要である。これはトレードオフである。完全な初期状態に戻すならば、再起動をかけるのがよい。しかし、それは実行時のコストがかかる。

であるので､成功し続ける限りは同一のインスタンスで実行するという設計にした(もちろんテストケースごとに初期状態には戻す)。失敗があった場合は、アプリケーションの状態が不正なものになっている可能性があるので再起動するようにした。成功し続ける場合も内部的には状態が不正になっている可能性はあるが、そこはトレードオフで許容することにした。

(2)　アタッチ

テストを実際に実行するためにアプリケーションと繋げる処理だが、ここにも一つ工夫がある。それは、自動で操作されるアプリケーションのデバッグである。前述の機能を使って、テスト時に新たなインスタンスを起動するだけでは、デバッグが非常に不便である。デバッガから起動したアプリケーションでテストが実行できれば、問題が発生した場合の解析が効率的にできる。そのため、どちらも使える設計にした。

システムテストで不具合を検出できた場合、その解析は一般的には困難である。単体テストと違って一つのテストケースでもカバーしている範囲が広いからである。

その難易度を下げるのは、混入から発見までの時間の短さ、トレーサビリティー、解析時に使用できるツールである。デバッガで得られる情報は非常に多い。これを有効活用できる設計は重要である。

(3)　フリーズ対応

アプリケーションのフリーズ対応も重要であった。デグレードの種類によっては、アプリケーションがビジー状態になることや、想定外のモーダルダイアログでテストが止まるケースがある。そのような場合にはアプリケーションを強制終了しなければならない。でなければ、後続のテストが実行できない。それをアプリケーションドライバの責務にした。

具体的には 1 ケースの実行時間を N 分以内とし、それを超える場合は、アプリケーションを終了させ、そのテストケースを NG にした。この時間制限は細かく決めると運用が困難になるので、プロジェクトで統一して、それを超える時間のテストケースは設計を変更するようにした。

7.1.2. 画面の操作機能

画面の UI 要素は外部仕様と捉えることが可能である。しかし、何段階か抽象化することも考えられる。例えば、次の画面は複数のテストシナリオの書き方が考えられ、その書き方によってアプリケーションドライバの公開するインターフェイスも変わってくる。

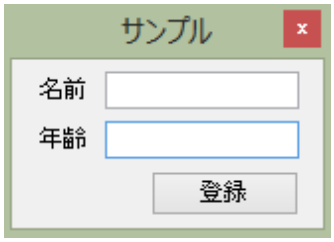

図 15-B-4-5　画面例

コード 1. ①GUI コントロールをテストシナリオに公開する

```
//名前をテキストボックスに入力する
appDriver.TextBoxName.EmulateChangeText(“山田太郎”);
//年齢をテキストボックスに入力する
appDriver.TextBoxAge.EmulateChangeText(“30”);
//登録ボタンを押す
appDriver.ButtonEntry.EmulateClick();
```

コード 2. ②項目と操作手順をテストシナリオに公開する

```
//名前を入力する
appDriver.SetName (“山田太郎”);
//年齢を入力する
appDriver.SetAge(30);
//登録する
appDriver.Entry();
```

コード 3. ③目的のみをテストシナリオに公開する

```
//個人情報を登録する
var data = new PersonalData()
{
      Name = “山田太郎”,
      Age = 30
};
appDriver.EntryPersonalData(data);
```

どのインターフェイスを採用するかは、それぞれの対象の特性によって決定した。

例えば、③の方法であれば、GUI コントロールの種類が変わったとしても、テストシナリオを書き直す必要はない。新規作成の機能で、その画面の目的は定まっているものの詳細な仕様は流動的な場合はこの手法が有効である。

しかし、抽象化されているので、詳細な手順のテストはできない（例えば入力順序など）。また、経験的にはテストシナリオを書く人間は①の方が直感的だと感じるようである。

アプリケーションドライバを実装する場合も、もっとも目的に合致した方法を選択できる。例えば、③の方法であれば、操作に使うレイヤは無理に GUI オブジェクトでなくとも、もっと下のレイヤのメソッド呼び出しにすることも可能である。

7.1.3. ショートカット機能

Windows アプリケーションの多くは複雑な画面遷移を持つ。そして、各テストケースにおいて本質的ではない画面の操作が必要になってくることも多い。また、それは複数のテストケースで使い回されることがほとんどである。そのような場合は共通化して定義しておき、目的以外の処理はそれを使って簡単に済ませてしまうとよい。

7.1.4. 作業分担

この設計方針は作業分担の上でも非常に効果があった。開発チームがアプリケーションドライバを実装し、テストチームがテストシナリオを記述できる。技術的な部分はアプリケーションドライバで吸収されるので、テストチームは簡単にテストシナリオを記述することができた。それぞれの長所を生かした作業分担ができたと言える。

テスト自動化を進める際にテストチームだけでやり遂げようとするケースも見かけるが、どのようなツール、ライブラリを使ってもテスト自動化には技術的な側面が存在する。それに関しては開発チームと密に連携して解決してくのが現実的な解である。

ちなみに、テストシナリオも C#を使ってプログラムを書いている。もちろんテストチームはテストのスペシャリストであり、プログラムは本職ではない。それでも書けたのは、テストシナリオの性質と開発環境が関係している。

テストシナリオ自体は、複雑な記述をすべきではなく、処理を上から並べていくだけのスタイルが望ましい。つまり、高度なプログラミングテクニックは必要としない。

また、テストシナリオの開発環境として Visual Studio を採用したのだが、アプリケーションドライバが公開するインターフェイスの命名規約をいくつか定めておくことで、後はインテリセンスの強力な機能により、書くべきコードが簡単に選択できる。これらの機能さえあれば、特にキーワード駆動のようなノンプログラムスタイルでなくともテストチームでテストシナリオを記述できることが分かった。

7.2. 内部 API の利用方法

内部 API を使えば、非常に強力で安定した操作を実現できる。それでは、内部 API はど

こで使うべきだろうか。先に述べたようにテストシナリオは外部仕様を元に記述するので、テストシナリオでは使用しない。内部 API はアプリケーションドライバを実装するために使うのである。

7.2.1. UI 要素を操作する

内部 API を使うとはいえ、システムテストであるので、できるだけ上位レイヤから操作できる方が望ましい。UI 要素の API を使用すると、一般的な GUI 操作とほぼ同等の効果を得ることができる。逆に言えば一般に提供されている Windows 標準の UI オートメーションでも記述できる部分も多い。では、この層の操作に内部 API を使うメリットは何であったかというと、次のものである。

- 画面、及び画面要素の特定
- サードパーティー、独自の UI 要素の操作
- ホワイトボックス的な視点での操作

(1)　画面、及び UI 要素の特定

画面、及び UI 要素の特定は GUI アプリケーション操作にとって非常に重要な処理である。特にメンテナンス性を考えた場合、座標ではなく、ID などでアプリケーションの設計によってプロトコル化できるものが望ましい。内部 API を使う手法では、フィールド名称で取得できるし、もしできない場合でも取得用のメソッドを作成して特定手段を明確化することができる。

(2)　サードパーティー、独自の UI 要素の操作

一般的に操作しづらいサードパーティーやプロジェクトで独自に作成した UI 要素も、内部 API を使えば操作できることがほとんどである。なぜなら、通常プログラムを作成するためには、使いやすくリッチなインターフェイスが提供されているからである。一般的にはこのような UI 要素の操作は諦めるか、もしくはキーマウスエミュレートにするしかなかったが、内部 API を使えば確実に動作するインターフェイスを低コストで作成することができる。

(3)　ホワイトボックス的な視点を持つ

重要なポイントは、この場合でもホワイトボックス的な視点を持つことである。それにより、UI 要素を操作した後の動作を正確に把握し、もしそれで不安定になる処理が書かれているのであれば、後述の GUI の一層下のレイヤの API を操作するという手法を選択することができる。また、操作中に思い通りに動作しない場合も、慌てずに解析することができる。

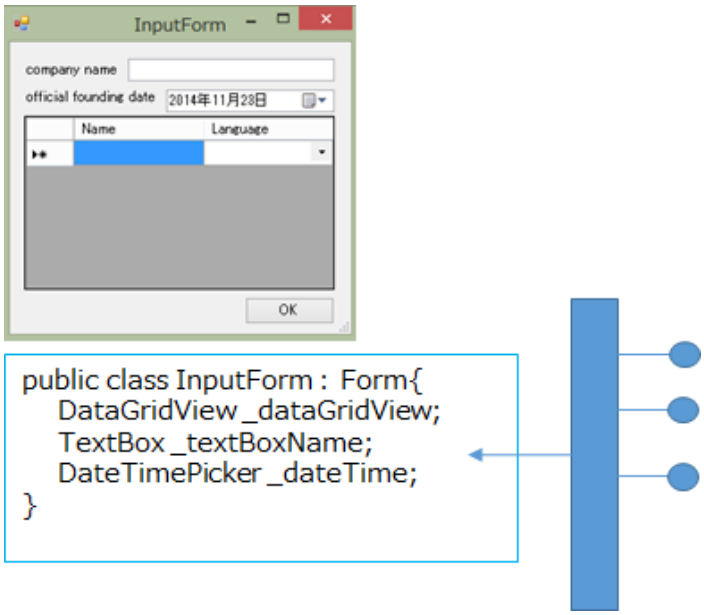

図 15-B-4-6　UI 要素を操作する

7.2.2. GUI の一層下のレイヤのメソッドを操作する

先ほどの UI 要素を操作する手法は通常の手法を少し発展させただけである。捨てているレイヤはキーマウスがアプリケーションまで届く層だけなので一般的な UI テストと同等の操作の効果を得ることができる。UI 要素との結合からテストでき、ほとんどはその手法で通常の手法より低コストで自動化を実現できるだろう。

しかし、いくつかのケースで、上手く操作できないものや、費用対効果が悪いものがある。その場合は、もう一層下のレイヤから操作を開始すればよい。

UI 要素は、大部分は外部から買ってきたものである。そのため、内部 API を使ったとしても、自由度にも限界がある。

しかし、ここから先は、それぞれのプロジェクトの完全な管理下になっている。テスタビリティーが足りなければ、それを向上させることが可能なのである。また、本当にテストしたい対象は、ここから先のレイヤであることが多い。UI の一層下というところがポイントで、自分達で書いたコードはほとんどテストできるうえ、さらにこのあたりは開発後期で調整を入れてもリスクが少ないことがほとんどなのである。例えば、以下のような場合に、このパターンを使った。

- マウス、キーの状態を直接参照するプログラム
- D&D（Drag & Drop）
- OS 提供の GUI を表示するプログラム

例を一つ示す。操作対象のアプリケーションに、ボタンを押すとフォルダー参照ダイアログ（OS 提供のダイアログ）が表示され、そこで選択されたパスを使って次の処理を実行する機能があった。

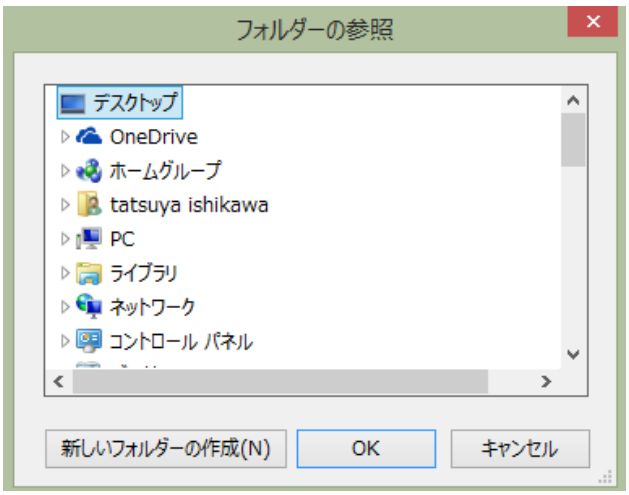

図 15-B-4-7　フォルダー参照ダイアログ

コード 4.　ボタンを押すとフォルダー参照ダイアログが表示される

```
void OnButtonClick(object sender, EventArgs e)
{
    using (var dlg = new FolderBrowserDialog())
    {
        if (dlg.ShowDialog() == DialogResult.OK)
        {
            FindTargetData(dlg.SelectedPath);
        }
    }
}

//デグレードが発生していないか日々確認したい処理
void FindTargetData (string path)
{
・・・
}
```

UI 要素のボタンクリックでテストを書くと、その後のフォルダー参照ダイアログも処理しなければならない。フォルダー参照ダイアログは OS や個人設定によってその GUI 表示が変わるので、非常に自動化しづらい画面である。しかも OS 提供であり、プロジェクトで作成したものではないので、日々テストするメリットは少ない。本当に毎日デグレードがないかチェックしたいのは、この例では FindTargetData メソッド以下の処理である。このような

場合は、ボトルネックでかつ価値の低い部分は飛ばして FindTargetData メソッドを呼び出してテストを実行するようにした。

7.2.3. GUI をまったく使わずに操作する

先ほどの例は、GUI を操作では実現困難、もしくはコストが合わない場合に GUI を削る処理を紹介した。この例では、もう少し別の理由から GUI 操作を削った例を紹介する。

目的は、その実行速度である。というのも GUI 操作は実行速度面で低速なのである。もちろん、内部メソッドを使用し、確実な操作で処理を実行する場合、かなり高速に処理を実行できる。なぜなら無駄な待ち合わせ処理を入れる必要性がないからである。

しかし、品質を確保するために大量のパラメタライズが必要になった場合は、これでも間に合わない。GUI への表示反映はプログラムの時間からすると圧倒的に遅い。つまり、このような場合は、実行速度上のボトルネックとなる部分を取り除いてテストを実行することが考えられる。

以下は、約 15 万通りのパラメタライズが必要となったテストケースの例である。GUI を操作する方法では高速に実行しても 1 ケース 2 秒で計算上では終了までに 83 時間必要となった。これくらいであれば、テスト実行マシンを分散して実行することも考えられなくはないが、目的に合わない箇所のためにコストをかけるのは得策ではない。このテストに GUI の操作は重要な意味を持たなかったため、それを排除してトータル 24 分で終わるようにした（実際は速度チューニングのために DLL インジェクションなどの他のテクニックも併せて使っている）。

表 15-B-4-1　テスト実行時間

種類	実行時間平均	合計
手動	30 秒	1,250 時間
自動 GUI 操作	2 秒	83 時間
自動 GUI なし操作	10 ミリ秒	24 分

7.2.4. 内部 API を使う上で気を付けること

内部 API を使うと、操作上、実行上のボトルネックとなる処理を避けてテストを実行することが可能となる。しかし、気を付けなければならないことは、省いた部分を管理し、その部分は手動で抑えられるようにしておくことである。

あとは当然のことだが、避ける部分が重要な部分やリスクの高い部分であっては意味がない。重要度の低い部分だから避けることが可能なのであり、重要な部分は毎日テスト可能な状態にしておくことに価値があるのである。

8. 成果

本事例の手法を適用したプロジェクトでは、それぞれの現場で、自動化された数万ケースのテスト実行が日々行われている。

多くの日には、全てのテストケースは成功し、デグレードなくプロジェクトが進行していることが確認できている。これらの場合は、成功しているという情報を確認すればよいだけでその日の運用コストは、ほとんど発生しない。デグレードの発生は、次の日には検出され、即時に対応が行われる。

重要なのは、日々大量なテストケースを実行できる高速性と、頻繁にテスト実行が中断しない安定性である。これがあるので十分な量のテストを実行でき、かつ安定性は信頼性につながりテスト自動化を継続するモチベーションとなる。

ただ、成果で注意しておかなければならない点が一つある。それは不具合検出率である。実は回帰テストの不具合検出率はそれほど高くない。と言うのは、デグレードに関しては、発生したときのリスクや回収コストが高いとはいえ、毎日発生するようなものではないからである。デグレードが毎日発生しているとすれば、解決すべき問題はもっと別のところにある。

不具合検出率で回帰検査を評価してしまうと、費用対効果が低いということになってしまう。しかし、そうではない。テスト自動化の本来の目的は、「不具合を見つけること」だけではなく、「不具合がないことを確認すること」であると考える。これらの回帰検査は、自動であれ手動であれ、必要なので実施されなければならない。しかし手動で実行するとコストが大きいので繰り返し実行するのは現実的ではない。自動化するからこそ、日々実行することができ、デグレードのリスクを軽減してプロジェクトを進めていくことができるのである。この点をステークホルダー間で共有しておくことが重要である。

9. 今後の課題

今回は Windows アプリケーションのシステムテスト自動化という領域に対し、内部 API を使うという手法を適用した。

システムテストが必要なソフトウェアは他にも多く存在し、多くは GUI 操作ベースでテストが実行されている。この手法は Windows アプリに限らず、様々な分野で応用が可能だと感じている。

今後は、様々なアプリケーションに対してこの手法を適用し、効果を計測していきたいと考えている。

参考文献

[1] David Farley, Jes Humble : 継続的デリバリー　信頼できるソフトウェアリリースのためのビルド・テスト・デプロイメントの自動化、アスキー・メディアワークス、2012.3.14