WHITE PAPER | 2015 年 9 月

モデル・ベースのテストを使用したビヘイビア駆動開発

2015 年 9 月

Huw Price
CA Technologies

ca technologies

目次

セクション 1

基本原則としての明確さとコラボレーション

ビジネスと IT のイニシアチブの関係が緊密になり、ビジネス要件を技術プロジェクトに直接移譲できるように伝えることは必要不可欠になっています。また、組織にとって顧客に価値を提供できるソフトウェアはますます重要になり、IT チームは変化するビジネス・ニーズにより速く、より低コストで対応できるソフトウェアを完全にテストした上で提供することが求められています。

しかし、ソフトウェア要件が曖昧であれば、エンドユーザやビジネス・アナリストが希望するソフトウェアに対する IT チームの理解度が低下し、その結果、ソフトウェアが組織にもたらす価値も低下します。

Dan North 氏が「ビヘイビア駆動開発」を提唱するようになったのは、要件の解釈が妥当であっても、最初の意図と異なる結果になるというソフトウェア開発おける不満が原因でした。同氏は常に「混乱と誤解」を経験し、「言ってくれればいいのに」と思う状況があまりに多いことから、ソフトウェア開発は「袋小路」のようだと考えるようになりました [1]。

一般的な IT プロジェクトで直面する課題を考えると、このような不満が開発者とテスト担当者だけのものでないのは明らかです。平均して、開始時に要件が明確なプロジェクトはわずか 4%[2] にすぎず、要件の不明確さは不具合の原因の 56% を占めています [3]。このような不具合は後から発見されることが多く、実運用中に検出されるため、要件の段階で発見された場合に比べてその修正には 50 倍もの時間がかかります [4]。要件の曖昧さに起因する作業のやり直しはソフトウェア・プロジェクトの失敗または長期化の原因の 60% を占め、実際、バグ修正作業の 82% は不十分な要件に起因するものであり [5]、毎年 2,500 億ドルもの無駄なコストが発生しています [6]。このような事実が動機となって、North 氏は誤解を招く「すべての「落とし穴」を回避できるようなわかりやすい方法で、テスト駆動開発（TDD）を紹介しなければならない」と考えるようになりました。

ドメイン駆動開発の「ユビキタス言語」の考えに触発されたビヘイビア駆動開発（BDD）では、ビジネスと IT のコラボレーションを重視します。IT チームは変更の多いビジネス要件に迅速に対応することが求められるため、一次的利害関係者と二次的利害関係者（ビジネス目標とアプリケーションの動作を重視する利害関係者と期待した成果と動作を実装する利害関係者）の両方が理解できる共通の言語を使用し、その言語を準形式化することで開発とテストの対象となるシステムのロジックへの変換を簡略化するという考え方には共感できます。

また、開発で問題なく実装するために、BBD では「ビヘイビア駆動要件」をどれほど正確に形式化できるかが鍵になります。この文書ではそれをテーマに、ビジネスと IT 間のコラボレーションと明確さを促進するというその基本原則を、開発ライフサイクルで最も効果的に実践する方法について考察します。

まず、一般に言われている要件の収集プロセスを見直して、適切なツールを使用することで、形式のモデル化が BDD にもたらす相乗効果について説明します。また、完全な文書化（「最小限の文書化」の逆）は変化への迅速な対応を妨げるという思い込みに対する反論として、形式のモデル化による要件の完全性の確保についても説明します。要件の完全性は明確さと共に、一次的利害関係者が期待するソフトウェアの動作の実現可能性を最大化するためには必要不可欠です。

セクション 2

不完全性

前述の BDD の基本原則から、要件はビジネスと IT の両方が理解できるだけでなく、テスト担当者と開発者が使用する技術要件として実装できることも重要です。ソフトウェアから直接価値を引き出す一次的利害関係者と要件について話し合い、アプリケーションに期待する動作の具体的な例や実際のシナリオを提供してもらって文書化すれば、ビジネスが必要とするシステムを確実に要件に反映させることができます。

このような開発の「外部」の作業によって曖昧さは解消されますが、要件とその抽出方法から、どのように実装してシステムのテストと開発を行うのかという疑問が起こります。

Llyr Wyn Jones 氏が「A Critique of Testing」で論じているように、要件の実装は情報の変換のプロセスとみなすことができます。情報はある形式で入力されてから、別の形式で出力されます。たとえば、開発者はコードを作成して要件を 1 つのソフトウェアに変換します。ソフトウェア開発ライフサイクル（SDLC）は、ユーザがビジネス･アナリストと共同で要件と使用事例を作成し、開発者がコードを作成し、テスト担当者が手作業でテスト･ケースとテスト・スクリプトを作成するという情報の流れのようなものと考えることができます。これらの段階のそれぞれで情報が引き継がれ、異なる形式に「変換」されます。

要件をアクティブな情報とする考え方をベースに、Llyr Wyn Jones 氏は「不確実性」という概念を用いて曖昧さを定義しています。同氏によると、指示の内容の不確実性によって要件の実装方法は複数になるため、不確実性は異なる情報が出力される原因になります。また、不完全性は、システムの必要な情報が思い込みのまま放置されることを原因としています [7]。そのため、どちらも「不明確さ」につながり、一次的関係者と二次的関係者間の誤解につながります。

つまり、不完全性は曖昧さと同様に、一次的関係者が意図したソフトウェアの動作を実現する可能性を低下させます。不完全性は、Dan North 氏が解決を試みた開発における誤解の可能性を高め、BDD の形式に至った考え方とは全く対照的です。

セクション 3

ビヘイビア駆動要件における不完全性

BDD では要件の形式化、ソフトウェアの開発とテスト、評価のための一次的関係者へのレポート提出という情報の流れは、Jones 氏の言う「フィードバック・ループ」で起こります。この場合、不確実性のリスクは、常に意図したとおりのソフトウェアが提供されるわけではなく、また、要件は常に進化するために、その「ループ」が無制限に繰り返されることです。

ビジネス目標と希望する動作に基づいて要件とテスト・シナリオを抽出することによって、テストでどのような結果になるべきかが強調されます。つまり、例外やエラーのない状態が第一の焦点になります。これは、シナリオ作成用に提案されている自然言語で見ることができます。

システムのロジックをブール演算子で考える場合、NOT、OR、AND の 3 つに絞り込むことができます（これら 3 つを組み合わせた XOR も含む）。これは、IF … THEN ステートメントに置き換えて考えることができます。

また、ガーキンでは「IF」、「AND」、「THEN」を中心にシナリオの最初の条件と設定の節、実際に存在する「トリガー」（「IF」）、およびシナリオの予測される結果（THEN）を記述します。各ステップでは AND によって複数の前提またはトリガーを指定できます。この形式でテスト･シナリオを収集する場合、OR が明らかに存在しません。また、これは Rspec で使用される別の自然言語、たとえば、一連の「When … Then」ステートメントの形式によるシナリオではより明確になります [8]。つまり、このような自然言語ではトリガーが存在しない場合にどうなるかは考慮されません。

ただし、この「OR」によって示される判断はシステムのロジックの基本であり、それぞれが異なるパスになる可能性があるため、テストの一部として強調する必要があります。これは特にネガティブ・パスに顕著で、テスト作業の約 80% を占めています。このような予測に反する結果は、トリガーが存在しない場合、つまり、「IF」が満たされない場合に発生しますが、このような予測に反する結果と異常値はシステムが破綻する大きな原因になります。

BDD は、プログラムの設計時に不足していたパスを特定し、いつ、どのような条件で予測に反する結果になったかを説明するのにはよい方法だと言われます。BDD は結局、反復のプロセスですが、そのために「フィードバック・ループ」が短縮されるということは、このようなネガティブ・パスをより速く処理できるため、テストをライフサイクルのより早い段階に移動することができます。そのため、「不確実性の活用」が提案されることさえあります。[9]

すでに説明したように、不具合をテストで検出されるまで放置するとコストだけでなく時間もかかり、実際に短期間の反復は多くの場合、ミニ・ウォーターフォールになり、テストは後回しにされるか、実行されなくなります。また、不具合の可観測性の問題はより深刻です。シナリオの予測される結果（Then）は検証可能な状態でなければならず、テストは予測どおりの結果になること、およびそれが正しいトリガーと正しい設定節によって引き起こされたことを確認する作業です。ただし、アド・ホック・ベースのテストの場合、複数の不具合が互いを相殺することもあるため、結果が予測された原因によって引きこされたことを確認する方法がありません。

セクション 4

モデル・ベースのテストと BDD

完全性を向上させ、目的どおりのソフトウェアを提供するために、BDD では「AND」に加えて「OR」の概念も導入します。BDD とガーキンなどの言語を個別に示したシナリオは、システムのロジックを反映するために接続する必要があります。そのためには、BDD のテストでは、実行されるはずのトリガーが実行されない場合、つまり、ネガティブ・パスの場合はどうなるのかも検討します。

テストを作成して実行する場合、テスト担当者は通常、単純なモデルを作成します。ネガティブ・テストでは、たとえば、設定節があるのにトリガーが機能しなかった場合の対応を考えます。また、BDD では、2 つのテスト・シナリオが同じステップを共有する場合、それらは「OR」によって黙示的に接続されます。たとえば、1 つのトリガーのみを共有して他のトリガーを共有しない場合、システムのロジックでは判断になります。

ただし、アドホックでモデル化する場合、テストはテスト担当者と開発者のそれぞれが考えたシナリオだけにしか対応していない可能性があります。BDD では、ユーザ・ストーリーが個別に直線のユニットとして示された場合がこれに該当しますが、システムのロジックにおける相関関係が反映されません。そのため、単純なシステムでも入力と出力の可能な組み合わせは数千に及び、1 人の人が記憶して正確に組み合わせることができる数量を超えているため、テストの機能カバー率は一般的に 10 ～ 20% と低く抑えられています。

BDD でテスト用のモデルが必要な場合、体系的な方法で行わない理由はないため、明確かつ完全な要件の文書を作成します。ネガティブ・パスと予測に反する結果を含め、テストでシステム内のすべての可能なパスに対応するなら、このような体系化は必要です。

セクション 5

BDD の一部としてのフローチャートのモデル化

以下では、開発にモデルを組み込む方法を提案していますが、この方法では前述の BDD の原則に対して相乗効果がもたらされます。

1. 具体例を使用した仕様：要件がいまだに動作によって決まり、利害関係者が希望するシステムの説明から抽出されることは注目すべきことです。ビジネス部門が形式化した要件は、リバースエンジニアリングによって、または一次的関係者自身がフローチャートを作成します。後者は、フローチャートをユビキタス言語として扱うことで可能になります。Bloor Research の Philip Howard 氏によると、フローチャート・モデルとは、テスト担当者と開発者の両方が必要とするシステムに関するすべての関数型論理によってビジネス部門を「茫然自失」させることなく実証できるものです [10]。

開発者とテスト担当者が既存の BDD の要件を使用している場合、プロセスはシナリオの重複するステップを接続することが中心となり、その後、システムのロジックで判断（OR）を構成します。そのためには、モデル作成者はシステムのロジックの観点から考え、完全性を向上させますが、それと同時に CA Agile Requirements Designer（旧 Grid-Tools Agile Designer）を使用すると、破損したパスを特定し、可能なシナリオの不足しているパスのヒントが提供されます。それによって、不足しているパスやネガティブ・パスがフィードバック・ループが完了する前に特定されるため、必要な反復の回数を削減できます。

2. モデル化のシナリオ：ユーザ・ストーリーは実際にはテストと開発にはレベルが高すぎます。開発およびテストするシナリオは、シナリオのステップがフローチャートのプロセスと判断のブロックを構成し、シナリオがシステムのロジック全体を通るパスになるフローチャートでモデル化します。シナリオはユーザ・ストーリーから抽出しているため、すべてのシナリオをモデル化すると、すべてのユーザ・ストーリーがモデル化されます。

説明のすべての側面はフローチャートに容易に移譲できます。たとえば、プロセスのブロックで実行する利害関係者を指定し、どのように「顧客が ATM から預金の引き出す」か記述します。フローチャートのパスには、BDD のユニット・テストを反映し、役割、必要な機能、利点または予測される結果を指定します。

メソッドを組み合わせているため、各シナリオには最初から最後まで他と異なる独自のパスはありません。その代わりに、複数のシナリオまたは機能の関数型論理のコンポーネントがフローチャートの特定の部分で組み合わされます。これは、ガーキンなどの自然言語を使用して収集した BDD 要件からテストを作成する場合と、システムの中核的な関数型論理をモデル化した場合の重要な相違を反映しています。モデル化した場合、複数のシナリオが個別のユニットとして示されるのではなく、システム内で 1 つに統合されているため、単一テスト・ケースで複数のシナリオに対応できます。このようにユーザ･ストーリーを 1 つの完全なシステムとして統合することは、不確実性を低減する重要なステップです。

複数のシナリオの場合、たとえば、システムを単一のフローチャートとしてモデル化すると、1 つの「Given（前提）」に統合されます。この「Given」はサブフロー (複数のパスの一部) になり、その後、判断ブロックで示されます。

同様に、パスの複数の「When（条件）」は判断ブロックまたはプロセス・ブロックで示すことができます。そこで複数のシナリオを統合し、複数のシナリオでトリガーを共有する場合、それらは同じブロックを通過できます。その途中にある判断はそれぞれ、そこから新しい 1 つまたは 1 セットのパスにつながり、システムが完全にモデル化されるまで、すべてのネガティブ・パスを含めて続きます。たとえば、2 つのシナリオがすべてのトリガーのうち 1 つだけ共有していない場合、最後の判断ブロックまで同じパスをたどり、最後の判断ブロックで分岐します。

3. フィードバック・ループ：モデルの検証。システムがユーザとビジネス・アナリストによってモデル化された場合、モデルの検証は不要なため、BDD は次のステップに進めます。この例ではフィードバックの遅延は実際には非常に短くなりますが、それについては後述します。

システムがテスト担当者と開発者によってモデル化された場合も同様に、モデルの検証は短時間で簡単に終えることができます。CA Agile Requirements Designer では、使用事例を使用できます。フローチャートの各パスはそれぞれ異なる使用事例を示しますが、これらはシステムのモデル化後のテスト・ケースまたはシナリオに相当します。検証は 1 セットの使用事例を通過するだけですが、これらはプレーン・テキストの一般的な用語とフローチャートの両方で表示され、システムが意図されたとおりに設計されているか確認できます。

4. フィードバック・ループ：モデルの妥当性確認。フローチャートはシステムの関数型論理をベースにしているため、テスト・ケースは自動的にそこから抽出されます。前述のように、システムを通過するパスはそれぞれがテスト・ケースを構成します。CA Agile Requirements Designer では可能なパスが自動的に特定され、その機能カバー率のメトリクスが正確に計算されます。

さらに、無効なテストや冗長なテストを削除し、可能なテスト･ケースの重複を排除できるため、最小限のテスト･ケースで最大限の機能カバー率を実現します。前述のように、複数のシナリオのロジックは統合できるため、最小限のテスト・ケースを使用してシナリオのセットをテストできます。たとえば、15 の可能なシナリオは 3 つのテスト・ケースを使用してテストできます。Agile Requirements Designer で提供されるさまざまなアルゴリズムを使用すると、機能カバー率が最大化されるパスを特定できます。たとえば、ある例では CA Agile Requirements Designer によって可能なテスト数が 326 からわずか 17 にまで削減され、その 17 個で 100% の機能に対応できました。

このようなパスを保存すると、エクスポートして実行することができます。これは手動でできますが、HP ALM/QC などのテスト管理ツールにテスト・ケースをエクスポートして、テスト・データと予測される結果にリンクできます。あるいは、パスを自動化されたテスト・スクリプトとしてエクスポートして、さまざまな自動化エンジンで実行することも可能です。いずれの場合も、使用しているテスト管理ツールによって「編集可能な文書」およびレポートが作成されます。

この自動化されたプロセスは Cucumber のプロセスと一見似ていますが、フィードバックの遅延がはるかに短く、必要なフィードバック・ループも少ない点で異なります。まず、テスト・ケースが自動生成されるため、手作業でテスト・ケースを定義する必要がなくなります。Cucumber の場合、Ruby で定義するために手作業でガーキンを変換する必要がなくなります。たとえば、ある例では、CA Agile Requirements Designer を使用して、90 分で 108 のテスト・ケースを作成し、機能カバー率 100% を実現できました。これにはフローチャートの設計時間も含まれます。フローチャートは簡単に微調整して再利用できるため、実際に削減される時間はさらに多くなります。これについても後述します。

また、開発者には検証済みの明確で完全な要件が提供されるため、最初に意図したとおりのソフトウェアを提供できる可能性が高くなります。前述のように、曖昧さは不具合の原因の 56% を占めていますが、CA Agile Requirements Designer では不具合の最大 95% が防止されます。テスト・ケースによる 100% 機能カバー率によって、開発からテストに渡される不具合も、最初に検出される可能性が高くなります。再作業に費やす時間だけでなく、テストにかかる時間も削減されます。テストの複製によってテストの時間が 30% 削減されるのに加え、テストの最適化によってすべてのシナリオに対応するのに必要なテストの合計数も大幅に削減されます。

セクション 6

変化への迅速な対応の簡略化

BDD 環境ではフローチャートのモデル化は時間がかかる、また、テストを何度も繰り返した方が時間を効果的に使えたのではないかという反対意見もあるでしょう。

まず、このような反対意見は、テスト担当者と開発者がフローチャートを自ら作成している場合のみを指している点に注意してください。ビジネス・アナリストとエンドユーザがガーキンなどの言語を使用するよりもフローチャートを好むようなことは、ほとんどありません。また、たとえ技術チームが BDD の要件をリバースエンジニアリングする必要があったとしても、フローチャートの作成にかかる時間は一般的に短時間です。フローチャートを作成してしまえば、要件が変更された場合でも簡単に再使用できます。これは、完全に文書化すると変更には迅速に対応できないという思い込みとは対照的です。

CA Agile Requirements Designer では、ユーザは関数型論理の新しい要素を追加するだけで、フローチャートを迅速かつ簡単に変更できます。破損したテスト・ケースはその後自動的に特定および修正され、重複または冗長なテストは削除されます。CA Agile Requirements Designer ではその後、最大限の機能カバー率の実現に必要な新しいテスト・ケースが生成されます。

それによって、最初の作業であるフローチャートの作成の価値が最大化され、変更されたときに手作業ですべてのテスト・ケースを確認する無駄な時間を排除できます。ある組織では、変更要求後 2 分で既存のフローチャートを変更しています。CA Agile Requirements Designer によって影響のあった 3 つのテスト・ケースが自動的に特定および修正され、残りの 64 はそのまま維持されました。この例が示すように、フローチャートのモデル化によって、変更の影響を受けるシステムの側面のみが再テストされるため、フィードバック・ループの効率が向上します。

つまり、フローチャートを作成するのにかかる時間よりも、不完全性や曖昧さによって必要になる再作業とデバッグ、テスト・ケースの作成、実行、変更要求の実装が不要になって節約できる時間の方が多いということです。このようなことから、完全な文書化はビヘイビア駆動開発の考え方と正反対なのではなく、むしろ、その実装の改善に役立つものと考えることができます。

セクション 7

まとめ

フローチャートのモデル化ではその基本原則を妥協することなく、BDD に完全な文書を導入する方法が提供されます。また、フローチャートは利害関係者の説明から直接導き出すことができるだけでなく、ユビキタス言語が提供され、テスト・チームと開発チームが必要とするシステムについて関数型言語に書き換えることができるため、ビジネスと IT の両方が理解できます。そのため、フローチャートによってテスト・サイクルが短縮され、テストで手作業が減少するため、「フィードバックの遅延」の防止にも役立ちます。

形式のモデル化によって明確で完全なビヘイビア駆動要件が提供されるため、最初の要件により忠実なソフトウェアを提供できる可能性が高くなります。また、ビジネス要件に適合するソフトウェアを提供するのに必要なフィードバック・ループの合計数が減少するため、繰り返しの多い面倒な再作業が不要になり、技術革新に集中できます。さらに、変更にも迅速かつ簡単に対応できるため、ソリューションを継続的に改善して、顧客に提供する価値を最大化できます。

セクション 8

著者について

約 30 年に及ぶキャリアを持つ Huw Price は、米国とヨーロッパの複数のソフトウェア会社で技術アーキテクトの責任者として、多国籍の銀行、大手公共事業および医療会社の高度なアーキテクチャ設計をサポートしてきました。長年にわたってテスト自動化ツールを専門に取り組み、ソフトウェア業界で使用されるテスト・モデルの交代を促す多数の革新的な製品を発表し、QA Guild の「IT Director of the Year 2010」に選ばれています。現在は国際的に有名なイベントで講演を行うほか、「Professional Tester」や「CIO Magazine」をはじめとする多数の技術雑誌に執筆しています。

最後に設立したベンチャー企業の Grid-Tools は 2015 年 6 月に CA Technologies に買収されましたが、同社は約 10 年にわたって大規模組織のテスト戦略アプローチの再定義を手がけていました。Grid-Tools は、Price の明確なビジョンに基づくアプローチとリーダーシップによってデータ中心の強力なアプローチをテストに導入し、Price が発案した「データ・オブジェクト」、「データの引き継ぎ」、「中央のテスト・データ・ウェアハウス」などの新しいコンセプトを発表しています。

ca.com/jp/でCA Technologiesにアクセスしてください。

CA Technologies（NASDAQ:CA）は、企業の変革を推進するソフトウェアを作成し、アプリケーション･エコノミーにおいて企業がビジネス・チャンスを獲得できるよう支援します。ソフトウェアはあらゆる業界であらゆるビジネスの中核を担っています。プランニングから開発、管理、セキュリティまで、CA は世界中の企業と協力し、モバイル、プライベート･クラウドやパブリック･クラウド、分散環境、メインフレーム環境にわたって、人々の生活やビジネス、コミュニケーションの方法に変化をもたらしています。詳細については **ca.com/jp** をご覧ください。

1 Dan North「Introducing BDD (2006)」ダウンロード：2015 年 6 月 3 日、http://dannorth.net/introducing-bdd/.

2 「Chaos Manifesto 2013」（Standish Group、2013 年）ダウンロード：2015 年 7 月 3 日、http://www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf.

3 「Requirements Based Testing Process Overview」（Bender RBT、2009 年）ダウンロード：2015 年 5 月 3 日、http://benderrbt.com/Bender-Requirements%20Based%20Testing%20Process%20Overview.pdf.

4 「Why testing should start early in software development life cycle?」（Software Testing Class、2012 年）ダウンロード：2015 年 6 月 3 日、http://www.softwaretestingclass.com/why-testing-should-start-early-in-software-development-life-cycle/.

5 Bender RBT「Requirements Based Testing Process Overview」

6 Kathleen Barret「Business Analysis:The Evolution of a Profession」（IIBA、2013 年）ダウンロード：2015 年 5 月 3 日、http://www.iiba.org/Careers/Careers/Business-Analysis-The-Evolution-of-a-Profession.aspx.

7 Erik Kamsties「Engineering and Managing Software Requirements」（Springer、2005 年）250 ページの「Understanding Ambiguity in Requirements」参照

8 この例は http://en.wikipedia.org/wiki/Behavior-driven_development#Story_versus_specification 参照

9 Dan North「Embracing Uncertainty (Goto Con、2013 年） ダウンロード：2015 年 6 月 3 日、http://gotocon.com/dl/goto-chicago-2013/slides/DanNorth_EmbracingUncertainty.pdf

10「Test Case Generation」ダウンロード：2015 年 6 月 3 日、http://www.agile-designer.com/wpcms/wp-content/uploads/2014/12/Bloor-Market-Report-Test-Case-Generation1.pdf