ける「両刃の剣」かもしれません。

**■ アジャイル開発における品質確保・試験終了への考え方は、ウォーターフォール開発に比べて形式化されていません。**

アジャイル開発ではウォーターフォール開発と同じ試験終了ストーリーは使えません。 そして、試験終了の基準もなく、すべてがPOに任されているというのが現状でしょう。 では、アジャイル開発における試験の進め方、試験の終了判断はどうすれば良いでしょうか？





## 6.2 アジャイル開発での品質確保・試験終了の考え方

3章を読んだ方は、すでに理解されていると思いますが、アジャイル開発では以下の原則があります。

**■ アジャイル開発では品質維持責任も、発注者側(PO)にあります。**

品質確保も完成責任の一部です。 よって、品質についてもPOのコントロール配下にあります。

ウォーターフォール開発では形式として、「目標試験密度をクリアすること」、「バグ収束曲線が収束すること」、「バグ発生原因を分析し、対策が完了済みであること」、などと形式的に品質条件を指定することが可能でした。 アジャイル開発であってもPOがウォーターフォール開発と同様の基準を達成する進め方は可能です。 それも一つの方法でしょう。

しかし、ウォーターフォール開発と同じ進め方は本当に正しいのでしょうか。 アジャイル開発のPOは早期のサービスリース圧力と、品質確保のバランスの中で、もっと良い方法と試験の終了時期を考えていくべきでしょう。

**■ アジャイル開発では、一定レベルの品質の確保の後は、小さく利用してもらいながら品質を上げ、利用対象を拡大するアプローチをとります。**

アジャイル開発では、初期にまず小さい範囲のユーザに利用を開始してもらい、徐々に利用者の範囲を広げていくという考え方を素直に適用します。 ウォーターフォール開発とは異なり、試験を一旦終了しリリースした後であっても、追加で品質向上のための試験を実施することができますし、初期のユーザ試用によって発見され





た重大な懸点は早期に対応していくことが可能なのです。 アジャイル開発では以下の2つの考え方が大切になってきます。

【方針１】 重要な部分から品質を確保する考え方
【方針２】 継続的に品質を改善していく という考え方

それぞれについて更に考えていきましょう。





## 6.3【方針１】重要な部分から、品質を確保する

**■ テストの前に、まずは動作させることを優先しましょう。 テストは、動作するものが固まってから実施するものです。 早すぎるテストは悪です。**

テストは動作するものが固まってから実施するものです。 アジャイル開発では設計や仕様をダイナミックに変更させていくケースがあります。 そういったアーキテクチャの試行錯誤の工程でテストを優先してはいけません。 アーキテクチャを固めることは、テストをすることよりも明らかに優先されます。「コードを書いたら、まずはテストする」という考え方や、「UT 100%」という考え方の優先度を高く設定してはいけません。 コードを書いた瞬間こそ、テストコードを書きやすいというケースもあるでしょう。 その場合はソースコード中に「TODO」として、テストに関するメモを残すことに留めましょう。

**■ テストは、早めに大きなリスクを抽出するように順序を計画しましょう。**

テストは大きなリスクを早めにつぶすべき、それが考え方の軸となります。

- システム設計上の核となる部分、設計が難しい部分。
- システムのメインルート、ユーザ影響が大きい部分。
- 課金等に関わるもののうち根幹となる部分。
- 仕様が難しい部分。
- バグが発見されると改修するのに時間がかかり、代替手段（運用対処など）がない部分





**■ 1つのテストの中に、重要な部分と重要ではない部分が混在する場合には、テストも分割して各優先度に応じたタイミングで実施しましょう。**

大切な機能だから最初にテスト すべきと謳われた機能であっても、本当にその機能を全てテストするのか、を考え、重要な部分とそうでない部分が混在しているのであれば、重要でない部分の実施優先度を考え直しましょう。「ついでにやりたいから」という理由に対する優先度は低いと思いましょう。 一緒にやることで時間効率が高いのであればバランスを考えて判断しましょう。

**■ 機能ごとに、試験工程の実施時期を変えることは普通です。**

重要部分やシステムとして早期に動かすべき部分の結合試験と、周辺部分の結合試験のタイミングは当然異なって構いません。 機能の重要性が高いものの結合試験工程と、機能の重要性が低いものの製造工程が各々同時に進んでいることは平常状態です。

**■ 外部 API のテストは重要です。**

システム連携など、外部システムとのAPIを持つシステムであれば、その API に関するテストは重要です。 他システムに悪影響を与えないために、さらに、今後のリリースにおいても API を確実な互換性を維持するために、早期にテストセットを整備し、確実に試験すべきです。





**■ 画面系のテストは、ずっと後です。 利用者の意見を聞きながら変化・反映させる部分に対して、最初にテストしても無駄になります。**

外部との連携部分として、ユーザ画面もあるでしょう。 しかし、画面系について最初から試験密度を上げるテストを行っていけません。 できるだけ、UI 部分とバックエンドを REST (HTTP) 等で分離し、優先度を変えて試験を実施していきましょう。

**■ 真にエンドの利用者に使ってもらうフェーズのリリースになってきたら、ユーザに失望されないレベルの品質が大切です。**

真のエンドユーザに失望されて、今後の開発や利用で協力を得られないレベルに陥るのでは問題があります。 エンドユーザはアジャイル開発にも、不完全なシステムの利用にも慣れていないかもしれません。 あらかじめ目的と品質レベルを通知し納得してもらって使ってもらいましょう。

**■ エンドの利用者に使ってもらった後のリリースでは、デグレード（先祖がえり）を避けましょう。**

一旦、利用してもらった後に、改善版を再度リリースして使ってもらう際には、デグレードしないように注意し試験を実施しましょう。 リグレッション・テストの整備を実施し始めるタイミングになっています。 あわせて構成管理（各サブシステムのバージョン管理等）も確実に実施していきましょう。





**■ その後のリリースでは、テスト不足を取り繕うことで忙殺されることのない品質レベルを目指しましょう。**

品質が悪いものを大規模ユーザに対してリリースすると、多数の課題が出てくるかもしれません。 品質強化の作業で開発メンバを酷使してはいけません。 アジャイル開発では開発メンバのモチベーションの維持が重要なポイントでした。 大規模ユーザへのリリースの前には、十分な総合試験と自動リグレッション・テストの整備を行いましょう。 リリース後の品質対応の苦労を知る優秀な開発メンバを含む開発チームは、きっと品質を上げてからリリースした方が楽だと思っているハズです。





## 6.4【方針２】継続的に品質を改善する（自動リグレッション・テストの整備）

アジャイル開発で作られるシステムは常に成長し続けるシステムです。 永遠のβ版かもしれません。

そこで、重要になるのが自動リグレッション・テスト(回帰試験) の整備と実施です。

**■ リグレッション・テストの整備とその自動化率の向上は計画に入れておきましょう。**

アジャイル開発のテストにおいてリグレッション・テストは必須です。 それは、アジャイル開発の**属人化リスクを避けるための重要な手段**でした。 リグレッション・テストの整備はついでの作業でできるレベルではありません。 あらかじめ計画の中に組み込んで置く必要があります。 リグレッション・テストの自動化環境の整備やCI 環境の整備は、システム本体開発の遅延対策用のバッファ（たとえば CI は後にする、自動化は後にするなど）として活用することもできます。

なお、自動リグレッション・テストの整備についても 100%主義を持ち込んではいけません。 たとえば、異常状態からの復旧のテストを100%自動化するのは投資対効果の面から良い判断とは言えないケースもあります。 また、リグレッション・テストの整備は、システム本体開発よりは優先度が下がることは意識しましょう。 リグレッションコードの整備を優先しすぎてシステムの進化を止めすぎると、アジャイル開発で得るはずだった競争力が低下して、開発プロジェクト自体の停止に至ることもありえます。





**■ リグレッション・テストの拡充によって、過去のリリースより品質が向上していることを積み上げて保証していきましょう。**

リグレッション・テストの整備も段階的に実施していくことになります。 自動化の度合いも徐々に向上させていくことになるでしょう。

リリースの都度リグレッション・テストを完了させていることによって、過去のリリース以上の品質であることを保証していきます。 開発者もサービス提供者も安心して次の機能開発に進んでいくことができるようになります。

**■ リグレッション・テストの整備のメリット／デメリット**

メリットは以下となります。

- 属人性の削減ができる。 維持管理費用の低減になる。
- OSS のセキュリティパッチ対応やバージョンアップの際にも、自信を持って対応できる。

一方、デメリットは以下になるでしょう。

- 機能作成時に、テストコード作成の手間が増える。
- 将来の機能変更の際に、テストコードのメンテの手間が増える。

**■ テストコードは適切なフレームワークを利用し、シンプルにして維持管理しやすくしましょう。**

上記のデメリットに対応するためには、テストコード自体の維持管理を考えなければいけません。 適切なフレームワークを活用し、できるだけシンプルに作らなけれ





ばいけません。 テストコードが複雑すぎて修正もできない、テスト自体が正しいか判断できない、ということになっては本末転倒です。

**■ CI 環境の整備のタイミングは遅くても良いと思います。**

自動リグレッション・テストが完成後、一度動作したシステムに対して継続的インテグレーション(CI)環境の実現と実施は非常に有効です。 Build&Test を毎晩自動実行していれば、互換性の崩れを翌日に検出できます。 けれどもリグレッション・テストの実行が自動化されていることがより重要であって、その日々実行の環境はそこまで優先度が高くありません。 実施するタイミングは、一度、商用サービスリースしてからでも良いと筆者は考えます。 当初の開発計画にはCI環境の構築までを含めておき、品質問題が発生した際には強化試験に置き換えるバッファとして使うのがバランスの良い考え方だと思います。

**■ TDD の価値は無いと思います。 多くの TDD は、実は単体試験駆動開発(UTDD)だからです。**

「テスト駆動開発(TDD)」、それは、非常に魅力的な視点を提供しています。 筆者もウォーターフォール開発の際に、設計書はテスト項目で書こうと提案したこともありました。

しかし、現在の筆者の考えでは、アジャイル開発において TDD は良くない考え方の側にいます。 TDD と言った場合には、単体試験を前提に語られているように思います。 要求仕様や基本設計等に対応するレベルの TDD であれば優先度を上げていくことは理解できますが、単体試験レベルを志向する TDD は良くないアプローチです。





**■ TDD は全体のアーキテクチャの決定より個別プログラムコードの品質を優先している点において、早期サービスリリースを優先するスタイルとは合いません。**

筆者の考えでは、単体試験の優先度はアーキテクチャの決定より圧倒的に低いと考えています。 まずは全体のアーキテクチャ設計をユーザからのフィードバックを受けながら確かめ修正していくことが優先されます。 将来捨てるかもしれないプログラムコードに対して、品質を求めていくことは優先順位が間違っています。 アジャイル開発ではUT100%よりも、サブシステム間のインタフェースを試しながら確定する方が重要なのです。

**■ 全体アーキテクチャが固定化された後の機能追加開発であれば、システムテスト駆動開発(STDD)は有効です。**

単体試験レベルに対する TDD ではなく、結合試験レベルに対する TDD が有効になることは間違いありません。 その場合の TDD は外部インタフェース(API)の試験と、ユースケースシナリオ試験になります。 ATDD（受入試験の TDD）もこれに近いかもしれません。





## 6.5 それでも、納期が厳しく品質も求められる場合の対応（例）

それでも、納期が厳しく品質が求められる開発の場合もあるでしょう。 その場合にはどのように進めるべきでしょうか？

残念ながら、科学的な手法はありません。 ウォーターフォール開発スタイルでの開発だった場合とアジャイル開発での比較をしながら考えていくのも、一つの方法だと思います。

従来のウォーターフォール開発で開発する場合のV 字モデルを参照しつつ、KKD 法によって、たとえば、一旦以下のような基準（仮）とします。

0／5 の時期 ・・・ 仕様の提示、設計開始
1／5 の時期 ・・・ 仕様の FIX
1／2 の時期 ・・・ 製造の完了、試験工程開始
納期 1ヶ月前 ・・・ 最低限の納品物が存在する
4／5 の時期 ・・・ 商用環境試験開始
5／5 の時期 ・・・ 納品

これを参考として、あなたのプロジェクトの計画や実績との比較分析をしてはどうでしょうか。 たとえば、1/2 の時期で製造工程は完了しているでしょうか。 普通のアジャイル開発であれば、既に先行開発部分を利用者に使い始めてもらっていておかしくない時期でしょう。 しかし、あなたのプロジェクトではまだ製造工程が完了していないとしたら、それは納期に対して大きな遅延かもしれません。 開発機能の一部を落としてでも試験に入っていかなければ品質確保は難しいのかもしれません。 逆





に先行する部分に対する試験が十分進んでいれば、±0 であって遅延はないのかもしれません。 この分析によって、今後、アジャイル開発で受け付けられる仕様変更の量を見極めるのも一つの方法でしょう。

この例は、あくまでも、他に頼るべき基準が無くて困っている場合ですが・・・。









# 7. 開発担当者の心得

受注者全員が日々、全体に優先度を考え行動すべし。
真のゴールとPOの価値観を理解し、チームの現状を把握し、
自らの状況を適切な相手に適切なタイミングで伝え、
次の作業案をもって優先度を都度 PO と相談し、
技術者としての高い付加価値を提供し続けるべし。

## 7.1 真のゴール、POの価値観、チームの状況を理解し、PO の判断に近づく努力をする

**■ PO を全力で支援する姿勢を持ちましょう。**

**【ルール】 アジャイル開発では、PO に最も負担が集中します。**

アジャイル開発では、ステークホルダとの仕様調整や各種の判断がPO の責任となり負担が集中します。 時には、PO の判断やレビュー待ちなどでプロジェクトのスピードが減速することもあるでしょう。 そんなアジャイル開発をより効率的に実施する





には、開発メンバは何をすればよいでしょうか？

**■ PO の判断に近い判断ができるようになって、主体性を高めましょう。**

それには各開発メンバが、PO に近い判断ができるようになるのが理想です。 POの考える真のゴールを正しく理解し、PO と同じ価値観をもって、今のチームの状況を見ながら判断していく練習を積み重ねましょう。 そうやって PO と同じ判断で作業が進められるようになれば、PO からの修正の指示は減少し、最終的にはあなた自身が自立して動けるようになるでしょう。

**■ PO の人物モデルを持って、自らを成長させていく。**

では、PO の判断に合わせるには、具体的には何をすれば良いでしょうか？ 4 章で示したように、まずは、相手の人物モデルを自分の中に持つことがコツの1つです。そして、各種の状況における PO の判断を見て、その人物モデルの価値観での判断と比較してみましょう。 異なっていれば、その人物モデルを補正し成長させます。
その判断の差が発生した理由が理解できないときは、PO に確認しましょう。「こういう判断もあったと思うのですが、なぜその判断では無かったのですか？」と。

**■ PO に求められている役割・責任や心得も理解しましょう。**

3 章、5 章で示した PO の役割・責任や心得等を理解して、PO の判断に近づきましょう。