

15-B-5

アジャイルプロセスにおける 実践的な品質向上施策の適用事例¹

1. 概要

サービスの早期リリース、システム利用者の満足度向上、開発コストの低減などを目的に、アジャイルプロセスによる開発を適用している。

変更を受け入れるアジャイル開発では、繰返しテストを実施し、品質確認を行っていくサイクルが必要となる。従来のウォーターフォールモデルのように開発後に品質を確認するプロセスでは、短納期開発の要求への対応は難しくなる。アジャイルプロセスを適用したが、品質確保のための施策が不十分であったことで、品質劣化による失敗プロジェクトとなったケースも聞くことがある。

本編では、アジャイルプロセスでよく適用されるプラクティスを採用することで、品質向上を実現したプロジェクトの事例を基に、実施した品質向上施策とその結果を紹介する。

本事例のプロジェクトは、エンタープライズシステム開発とインターネット上で提供するサービス開発であり、要件定義から基本設計までの上流工程と結合テスト以降のテスト工程をウォーターフォールモデルと同様に行うハイブリッドアジャイル[1]で実践した。

2. 取り組みの目的

2.1. 解決すべき課題

アジャイルプロセスでは、開発効率を上げるために作業の適切な省略・簡略を行うが、品質向上のための作業はしっかりと行う必要がある。作業項目が増えても自動化などで作業スピードを上げることで、開発効率を上げると同時に品質確保も行っていく。

ウォーターフォールモデルでは、ユーザー側のテスト工程以降に仕様齟齬による手戻り作業が発生したり、利用者の不満が上がってくる場合がある。その解決策として、アジャイルプロセスを適用する場合もあるが、アジャイルプロセスで品質確保のための施策が不十分なプロジェクトでは、次のようなケースが品質を下げるリスクとして考えられる。

- ・ 設計書を必要以上に省略・簡略化することにより、テストケース考慮漏れが発生し、

¹ 事例提供: 株式会社日立ソリューションズ 技術統括本部 英 繁雄 氏

テストが不十分となる。

- ・ 設計書を必要以上に省略・簡略化することにより、開発者に仕様が伝わらず実装されたシステムと仕様とに齟齬が発生する。
- ・ 変更を受け入れることにより、デグレードが発生する。
- ・ 変更を受け入れることにより、再テスト工数の膨張を抑止するため、テスト不十分となる。
- ・ 品質管理方法やテスト計画が不十分なことにより、問題発見が遅れる。

2.2. 目標

早期リリースや生産性向上だけでなく、開発途中やリリース後に発生する仕様変更や追加仕様に対応するために、アジャイルプロセスを適用したが、品質劣化によるトラブルケースとなることがあった。そのため、アジャイルプロセス適用時でも、ウォーターフォールモデルに劣らぬ品質を確保することを目標とし、次のような方針で行う。

- ・ シンプルで共通化された可読性と保守性に優れたプログラムコードにする。
- ・ テスト作業効率を上げる（テスト自動化）。
- ・ 変更に対するデグレードを防止する。

3. 取り組みの対象、適用技術・手法、評価・計測

3.1. 取り組みの対象

対象プロジェクトは、2012 年度に約 1 年の期間で、ハイブリッドアジャイルを適用したエンタープライズシステムの再構築プロジェクトの事例(以降、事例プロジェクト)である。

品質評価は、信憑性を確認するため、ほぼ同等の品質向上施策を適用した 5 つのプロジェクトの品質結果で評価した。

3.2. 適用技術・手法

(1) 開発プロセス

アジャイルプロセスと呼ばれる手法は多々あるが、代表的な Scrum の開発イメージは、図 15-B-5-1 のようになる[2]。

Scrum では、イテレーション（繰り返し）のことを Sprint と呼び、通常 1 回、2 週間から長くて 30 日間で実施する。Scrum は、役割、会議、規則および作業の仕方を提供する開発プロセスのフレームワークであり、1 チーム 7 人前後で編成されることが適当とされる。

Sprint のはじめに、要件一覧の Product Backlog から当該 Sprint で開発する候補を決め、具体的な作業レベルの Sprint Backlog へ詳細化する。Sprint 内では、実装作業を毎日繰り返す。Sprint の終わりには、Sprint レビューを行い完成したものは、正式または内部的に完成品としてデプロイを行う。

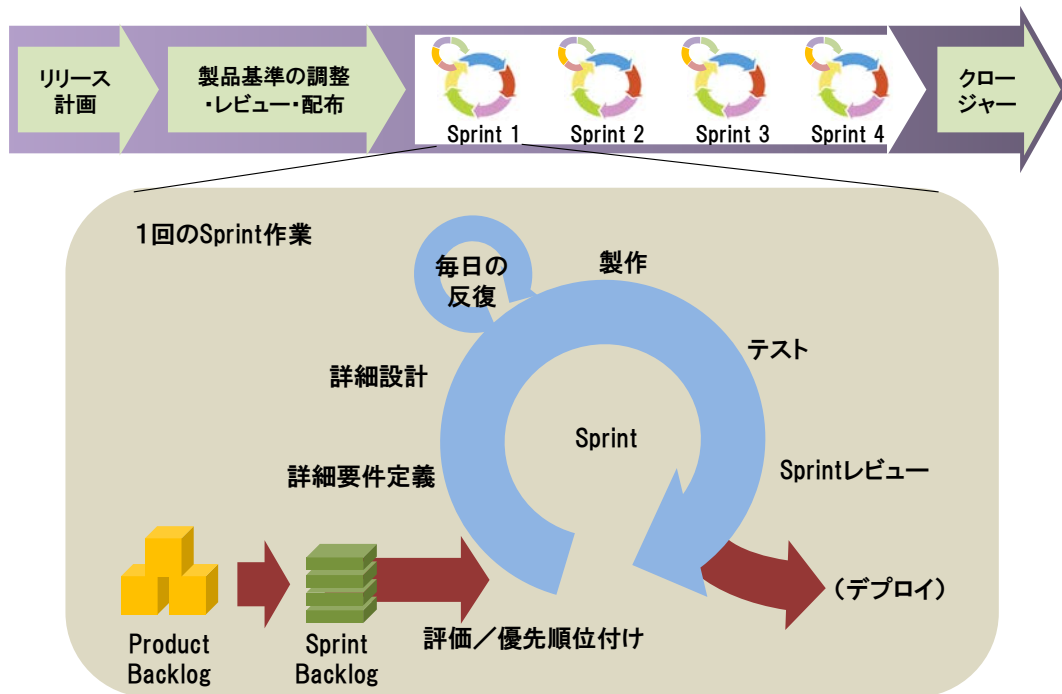


図 15-B-5-1 Scrum の開発の流れ

事例プロジェクトでは、既存システムのサポート切れに伴う再構築であったことから、一般的にアジャイルプロセスに求められる、優先度の高い機能から仕様を決めて、早期にリリースしていくという作業の必要はなかった。そのため、実際に動作するアプリケーションで定期的に仕様齟齬と操作性を確認しながら、可能な限り作業を簡略化・自動化して開発効率を上げることを目的とした。

事例プロジェクトでは、最後に一括してリリースすればよいため、品質向上のための結合テスト以降は、ウォーターフォールモデルと同様に行った。

いわゆる、ハイブリッドアジャイルとかウォーター・スクラム・フォールという方法である。

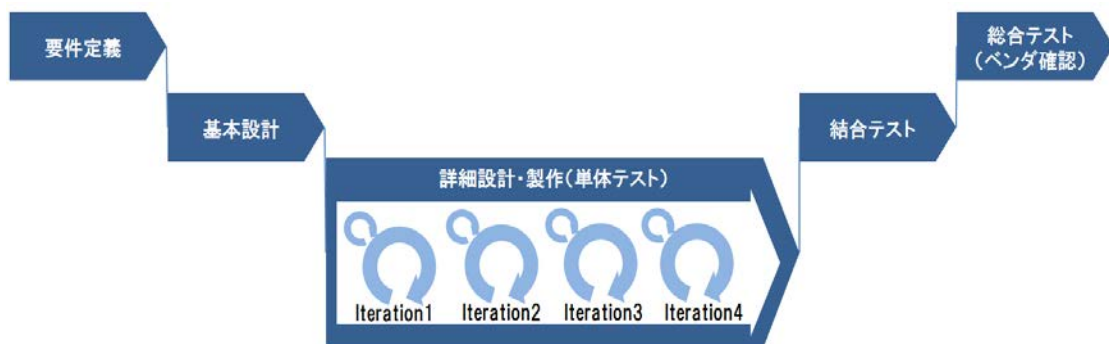


図 15-B-5-2 事例プロジェクトのハイブリッドアジャイルの流れ

ハイブリッドアジャイルの適用例として他にもさまざまなパターンが考えられるが、いくつかの例を次に挙げる。

- ① 要件定義など上流工程だけをアジャイルで反復開発を行う
- ② 基本設計から結合テストに相当する工程を同時にアジャイルで反復開発を行う
- ③ 要件定義から基本設計を同時にアジャイルで反復開発し、その後、詳細設計から製作（単体テスト）をアジャイルで反復開発を行う

(2) 品質確保のための適用技術・手法

本報告では、特にアジャイルプロセスでよく適用される品質向上のための施策にスポットを当てて説明する。品質向上の効果が期待できる技術的施策は、テスト駆動開発と継続的インテグレーションである。また、会議体で期待できるのはイテレーション計画と仕様確認レビューの実施である。

表 15-B-5-1 品質向上を目的とした適用技術・手法

No.	適用技術	概要
1	テスト駆動開発 (TDD : Test-Driven Development)	テストコードの作成とそれが動作するアプリケーションコードを、少しずつ繰り返して完成させていく方法
2	継続的インテグレーション (CI : Continuous Integration)	個々の開発者が完成したプログラムを常時結合し、ジョブを自動実行するための専用の実行環境のこと
3	イテレーション計画	各イテレーションの初めに、当該イテレーションで開発する機能を決定し、作業に分割する場
4	仕様確認レビュー	各イテレーションの最終日に、ステークホルダーに、成果物を確認してもらう場

3.3. 評価・計測

品質を確保するために、アジャイルプロセスで単体テストレベル（画面からの疎通テストを含む）まで開発した後、ウォーターフォールモデルと同様に結合テスト、総合テストを実施した。

アジャイルプロセスでは、品質が問題となるプロジェクトも見られるが、今回の品質向上施策を実施した場合、ウォーターフォールモデルに比べて、どの程度の品質確保ができたのかを評価した。

品質指標は、各テスト工程で、プログラムコードのステップ当りのテストケース密度とバグ抽出密度とした。ただし、テスト駆動開発対象部分である業務ロジック部分に対しては、テスト駆動開発は、常に正常状態を維持する開発手法のため、バグ抽出密度の計測は困難であり、コードカバレッジの網羅率で判断することにした。テスト駆動開発では、確認できない画面の操作や画面から業務ロジックとの接続確認は、画面からの疎通テストとし、単体テストの位置づけとした。

表 15-B-5-2 品質指標と品質目標値

テスト工程		品質指標	品質目標値
単体テスト	テスト駆動開発	コードカバレッジ	C0 メジャー※： カバレッジ 100%
	画面からの疎通テスト (テスト駆動開発非対象部分)	テストケース密度、 バグ抽出密度	ウォーターフォールモデル と同等の目標値を採用
結合テスト			
総合テスト			

※ C0 メジャー：命令網羅率（ステートメントカバレッジ）で、コード内の全てのステートメントを少なくとも1回は、実施する。

4. 取り組みの実施、実施上の問題、対策・工夫

4.1. 実施内容

(1) 品質確保のためのプロセス

事例プロジェクトでは、ウォーターフォールモデルで行っていた品質を確保するための従来の作業を、アジャイルプロセスを適用したイテレーションの中で、図 15-B-5-3 のような手順で実施した。対象工程は、ウォーターフォールモデルの詳細設計・製作（単体テスト含む）工程である。

単体テストまでは、テスト駆動開発や継続的インテグレーション、画面からの疎通テストとお客様による仕様確認レビューを実施することで、品質に対する施策は厳しくなっているといえる。もちろん、ウォーターフォールモデルであっても、このように行うことは可能だ。ただし、アジャイルプロセスでは、変更に対応することを前提としており、変更による修正作業への影響を少なくするために、設計書の修正影響を少なくしたり、再テストの自動化などで修正スピードのアップを行ったりする。

単純に、アジャイルプロセスだからと言って、設計書作成を省略しただけでは、品質を低下させる危険性もあるため、品質を確保するための作業は、図 15-B-5-3 のような流れで行った。

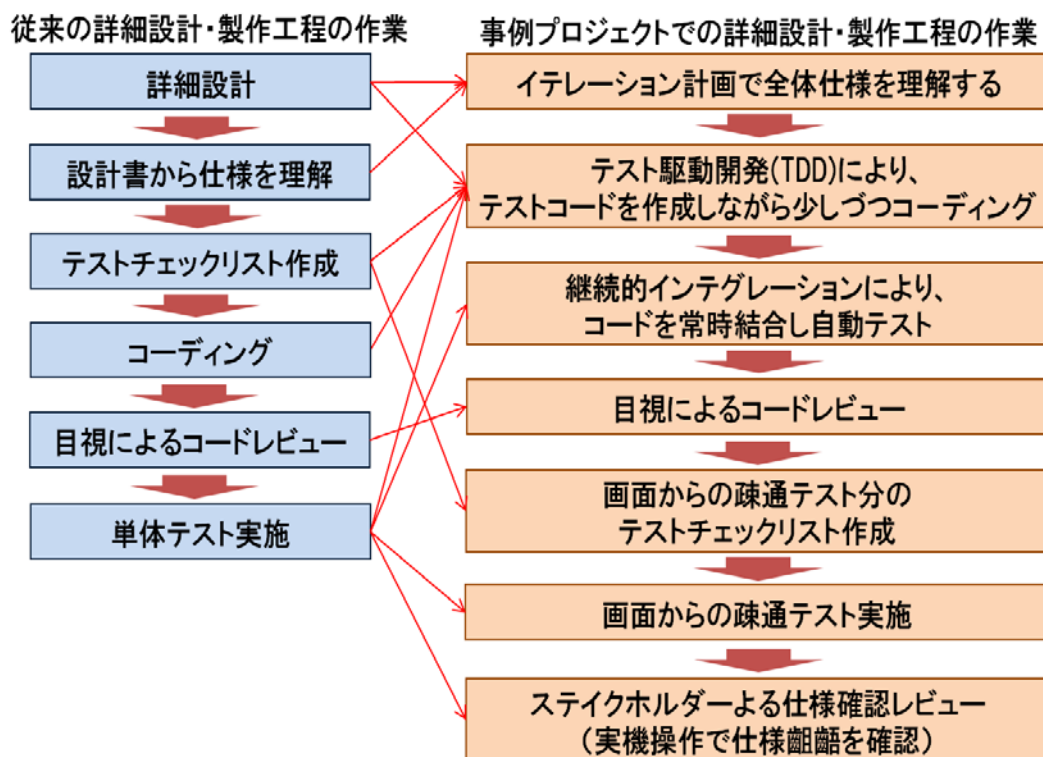


図 15-B-5-3 品質確保のために実施した作業例

4.2. 具体的な取組み内容、活動

品質確保のために実施した作業を具体的に説明する。

(1) イテレーション計画

イテレーション計画とは、各イテレーションの初めに、今回のイテレーション中に開発する機能を決定する場のことである。

開発チームは、イテレーション計画の時間内で、開発する機能を作業レベルのタスクに分割し、直観的に作業に必要な工数を見積る。工数を見積るために、開発する機能の仕様を理解する必要があり、全員で見積りのための議論をすることで、全員がこのチームで今から開発するものの仕様を理解することができる。

イテレーション計画のタイムボックスは、Scrum では、2 週間イテレーションで 4 時間、30 日間のイテレーションで 8 時間が目安であり、イテレーション期間に比例した時間が目安である。短時間に当該イテレーションで開発する機能の全体仕様を全員が理解することが重要であり、効率よく実施し、細かい仕様は開発しながら考えていく。

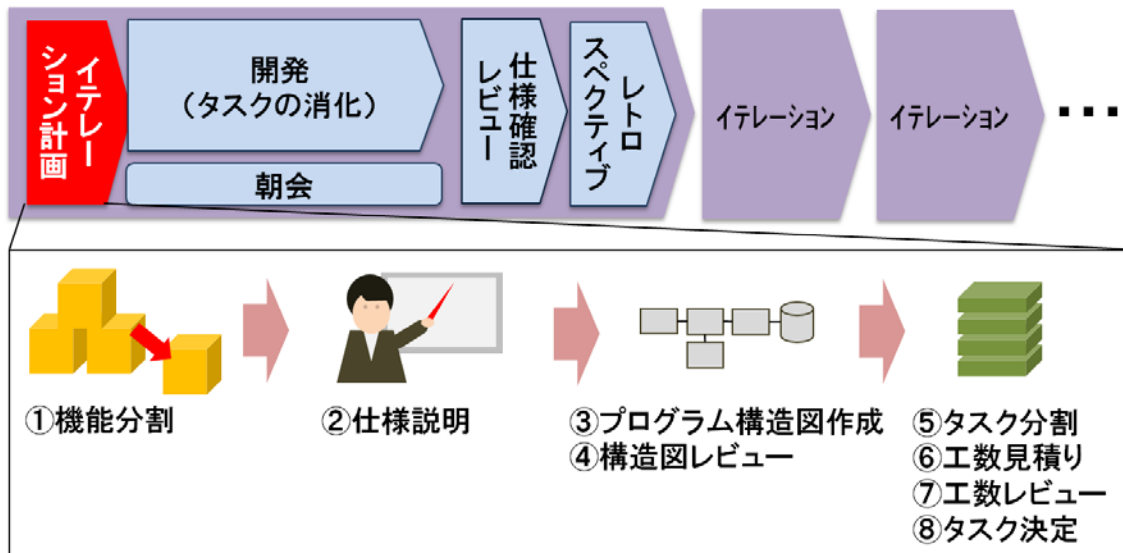


図 15-B-5-4 イテレーション計画の実施例

事例プロジェクトでは、次の手順でイテレーション計画を実施した。必ずしもこの手順で行う必要はないため、自身のプロジェクトに適した手順を考えていただければと思う

- ① 機能分割
- ② 仕様説明（仕様理解）
- ③ 2～3 人のチームに分かれてプログラム構図作成
- ④ 開発チーム全員で、プログラム構図をレビュー
- ⑤ タスクに分割
- ⑥ 2～3 人のチームに分かれてタスク消化の工数見積り
- ⑦ 開発チーム全員で、タスク見積り工数レビュー
- ⑧ 当該イテレーション内で実施するタスクの決定

事例プロジェクトでは、ホワイトボードなどを用いて、実装する機能ごとにプログラム構図を手書きで描きながら検討した。プログラム構図を基に、実装に必要な作業（タスク）に分割し、タスク単位の工数見積りまで実施した。そのため、30 日イテレーションで行ったが、イテレーション計画に 2 日かかったこともあった。

(2) テスト駆動開発

テスト駆動開発は、プログラミング時に、テストコードの作成と、それをクリアする処理の実装を少しずつ交互に行いプログラムを完成させていく方法である。テストコードとは、開発する処理のインプットとアウトプットを明確にしたテストを行うためのテストプログラムのことである。テストプログラムを実行することで、アプリケーションが入力データに対して、正しく結果を返しているかを確認することができる。

テスト駆動開発の手順は、はじめに仕様に沿ったテストコードを少しだけ作成し、そ

のテストコードに基づいて動くプログラムを作成する。このようにテストコードを先に作成することで、余計な処理の実装を抑制したり、テスト実施漏れを防いだりする効果がある。

テスト駆動開発は、数分の短いサイクルで行うことで、開発者は繰り返しプログラムコードの見直しと仕様の再考を行うことになるため、仕様の間違いに気づいたり、プログラムコードが洗練されたりするなど、品質向上に効果を発揮する。

さらに、作成したテストコードは、後に修正作業によって、デグレードが発生していないことを確認する場合にも利用できるため、変更対応をすることを前提とした場合や、稼働後にもエンハンスを何度も行っていくような場合には、一度作成したテストコードを再利用することで、保守効率の向上にも繋がる。

テスト駆動開発は、図のような作業サイクルを回し、プログラムが望みどおりの動作をするまで、この手順を繰り返す。

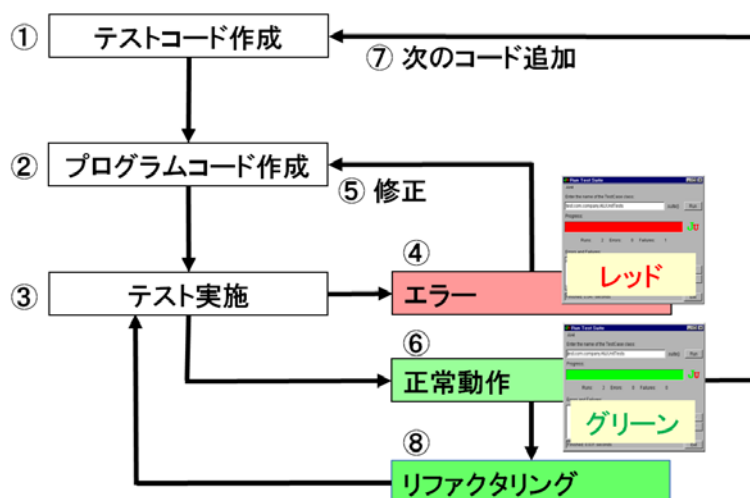


図 15-B-5-5 テスト駆動開発の作業手順

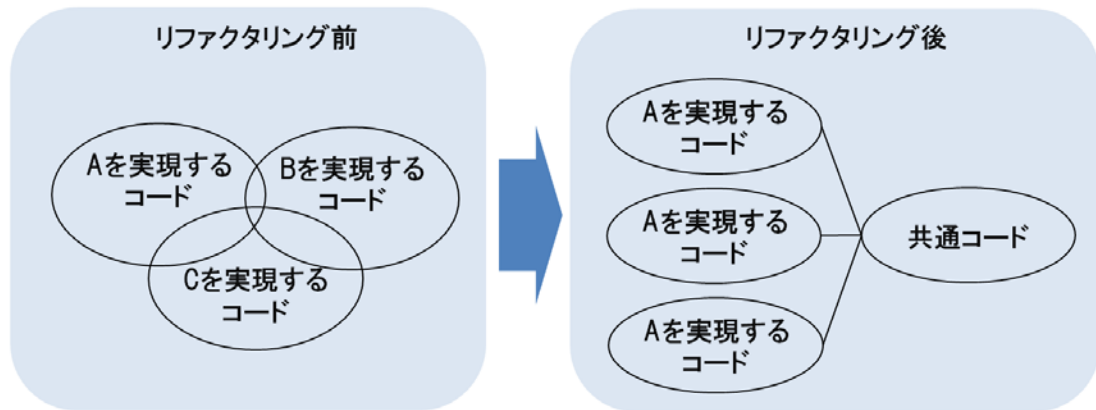
- ① テストコードを作成（追加）する
- ② テストコードに対するプログラムを作成（追加）する
- ③ テストを実施する
- ④ エラーの場合 → レッド（テストツール上のテスト結果の色）
- ⑤ プログラムコードを修正する
- ⑥ 再テスト後、正常動作の場合 → グリーン（テストツール上のテスト結果の色）
- ⑦ 次のテストコードを追加する
- ⑧ プログラムコードを綺麗にする → リファクタリング

正しく動作すれば、①に戻り繰り返す

テスト駆動開発は、レッド（バグあり）、グリーン（正常）、リファクタリングのリズムを繰り返して開発を行う。

リファクタリングとは、外部から見たときの振る舞いを保ちつつ、理解や修正が容易になるように、ソフトウェアの内部構造を変化させることであり、一言で表現すれば「綺麗なプログラムコードにする」ことである。重複するプログラムコードの共通化やプログラムのインターフェースを変えずにプログラム内部の機能を成長させていく方法である。

共通化可能なコードを整理する



インターフェースを変えずに中味を進化させる

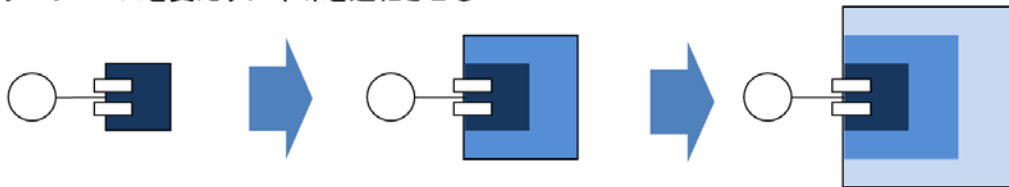


図 15-B-5-6 リファクタリングのイメージ

事例プロジェクトでは、テスト駆動開発を適用したが、この開発リズムに慣れず戸惑う開発者もいた。

(3) 継続的インテグレーション

継続的インテグレーションとは、個々の開発者が完成したプログラムを開発チームの共用サーバに常時結合し、設定した任意のジョブを自動実行するための専用環境である。

多くの場合、プログラムコードのビルド、テスト、コード解析などを自動的に行うように設定している。継続的インテグレーションでは、動作するソフトウェアを維持するため、バグを即時に発見でき、再確認作業の効率化と品質確保に効果がある。

継続的インテグレーションの実施手順は、次のとおりである。

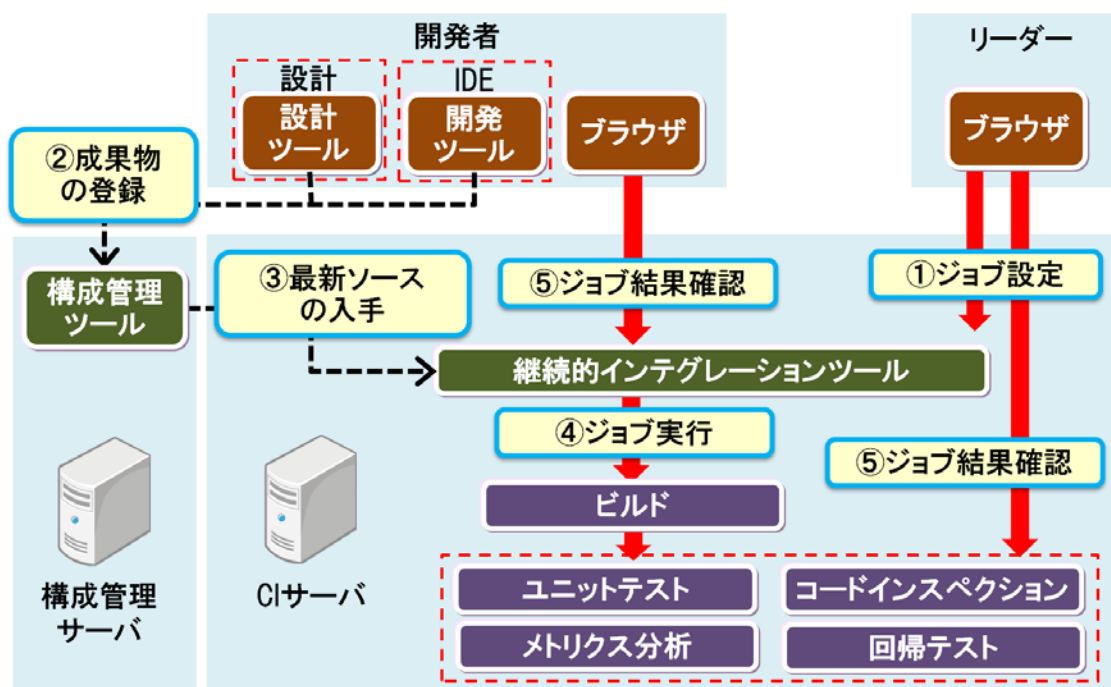


図 15-B-5-7 継続的イテレーションの実施例

- ① リーダーは、継続的インテグレーションで実行するジョブを設定する。
- ② 開発者は、個々の開発環境上で完成したプログラムコードを構成管理サーバにコミットする。
- ③ 継続的インテグレーションサーバは、構成管理サーバから最新のプログラムコードを取得する（自動化可能）。
- ④ 継続的インテグレーションサーバは、設定されたジョブ（ビルド、テスト、コード解析など）を自動実行する。
- ⑤ リーダーと開発者が結果を確認する。

継続インテグレーションで実行するジョブに、回帰テストツールの実行も設定することができる。回帰テストも自動化することで、プログラム修正時に行う再テストを自動化できるため、再テスト工数の低減とデグレード防止を期待することができる。

回帰テストツールは、テスト実施時にキーボードやマウスからアプリケーションに対して行った操作と、画面項目の確認内容を記録して、操作記録ファイルを作成する。そのファイルを再生すると自動で画面操作を再現することができる。ただし、画面仕様が安定してから適用しないと、テストケースの再作成に時間がかかりすぎることもあるため、効率を考慮して適用時期を検討したほうがよい。事例プロジェクトでは、結合テスト以降の適用とした。

(4) 目視によるコードレビュー

アジャイルプロセスでは、よくペアプログラミングという方法が用いられる。ふたり

でペアとなり、1台のパソコンに向かって、いっしょに開発する方法だ。ひとりがナビゲーターで、もうひとりがドライバーの役割となる。ドライバー役の開発者がコーディングすると同時に、ナビゲーター役がコードレビューをする。

ふたりでチェックしながら、開発を行うためプログラム品質がよくなると期待される方法である。

事例プロジェクトでも、ペアプログラミングは実施したが、プロジェクトが進むにつれて、開発者のプログラムレベルが標準化していくと、ひとりで開発した方が効率がよいと判断したため、途中からは技術的リーダーが、各開発者のコーディングしたプログラムをひとりでコードレビューするように開発者の状況に合わせて切り替えるようにした。

(5) 画面からの疎通テスト

テストコードが対象とするプログラムは、業務ロジックやデータベースアクセス処理といった部分になる。また、テストコードは、テストケースのチェックリストと重複作業となるため、テストコード作成対照部分のチェックリストは作成しないようにした。

ただし、これはテストケースを作成しないため、仕様の実装漏れが発生する危険性がある。

事例プロジェクトでは、実装漏れを防ぐことと、テストコードの対象外である画面内処理や画面との接続テストを、画面からの疎通テストとして実施することで、テスト十分性を担保するようにした。

(6) 仕様確認レビュー

仕様確認レビューとは、アジャイル開発などで各イテレーションの最後日に、仕様に意見を述べることができるステークホルダーから数名の代表に参加してもらい、実際に動くアプリケーションで仕様を確認してもらう場である。これにより、実装漏れや仕様齟齬の早期発見、利用者観点から見た操作性の確認を行うことができる。実装したものの仕様に問題がなければ、その機能は完成である。変更要望があれば、変更の可否と優先順位を考慮し、変更が必要なものは、次のイテレーション以降で変更対応するようにする。

仕様確認レビューは、開発中に定期的に実際に動作するもので確認することができるため、仕様齟齬や操作性の問題を早期に発見することができ、品質向上にも繋がり手戻り作業を最小限に抑えられる。

事例プロジェクトでは、プロジェクトルームに発注側の仕様決定者が常時同席することはできなかった。そのため、イテレーションごとに、成果物を発注側がレビューする「仕様確認レビュー」という場を4時間ほど設けた。時間が少し足りなかったため、予備として各イテレーション終了後に1週間ほどの期間で、ステークホルダーが、いつでも実機で操作確認できるように確認用の環境を準備した。

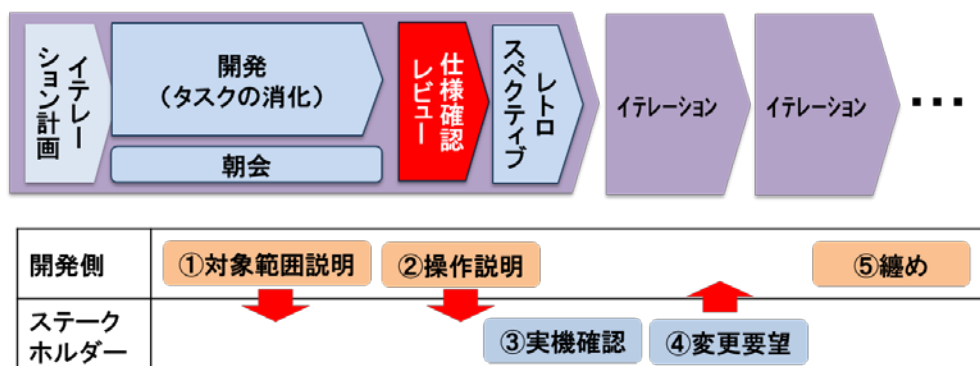


図 15-B-5-8 仕様確認レビューの実施例

- ① 開発側が、今回確認する機能概要を説明する
- ② 開発側が、実機による機能説明する
- ③ ステークホルダーが、実機で確認する
- ④ 開発側が、仕様に対するステークホルダーの意見を確認する
- ⑤ 仕様に問題のない機能は完了を宣言する

4.3. 生産物、途中成果物の変更、改善等

(1) 設計書の削減

事例プロジェクトでは、詳細設計書に相当するドキュメントは、できる限り省略し、プログラム構造を整理するために必要なものは、ホワイトボードや紙に手書きでメモ書きにした。設計書を省略・簡略化することで、作業量の削減だけでなく変更による設計書の修正時間を削減することができた。

(2) 再実行の効率化

事例プロジェクトでは、デグレード防止のための再テストを自動的に実行できるようにした。そのため、テストコードを作成し、テストコード対象ロジック部分以外の単体テストレベルでのテストケースのチェックリストの作成は、行わないようにした。

(3) 開発途中の仕様確認

アジャイルプロセスでは、イテレーションという短い期間で、ステークホルダーと仕様の確認を行っていく。そのため、一定のペースで仕様確認が取れたものが完成されていくため、ウォーターフォールモデルのように全てが完成してから仕様齟齬による手戻り作業が発生するのを防ぐのに効果的である。イテレーティブ開発やインクリメンタル開発と同様のように思えるが、設計書の削減や再実行の効率化を行う点でアジャイルプロセスとして分類される。

4.4. 途中ででの主なトピックス

(1) テストコード作成は無駄ではないか？

テストコードの作成作業は、保守におけるデグレード防止や再テストの効率化に役立つ

つと期待される。開発者は、最初、テストコードの作成時間が生産性の面では無駄だと思いがちだが、開発途中に受け入れる変更や保守時の再テストを考えれば、再テストコードを再利用可能なため、生産性を落とすとは言い切れない。

(2) テスト駆動開発の価値がなかなか理解されなかった

イテレーション開始前に、プロジェクト教育としてテスト駆動開発の説明も実施したのだが、初めの頃はアプリケーションプログラムを先に作成して、テストコードを後で作成する人もいた。これは、アプリケーションプログラムが間違っていた場合に、間違ったプログラムに合わせて、テストコードを作成する危険性があるため、正しい手順で行うことの価値を再認識してもらった。

また、テストコードを多く書いてしまってから、アプリケーションプログラムをコーディングする数時間という長いサイクルで開発していた場合もあった。これは、かえってバグ修正時間がかかってしまうこともあり、短いサイクルで行うことの価値を再認識してもらった。

(3) 回帰テストの修正は大変だった

回帰テストの適用時期は、当初はイテレーション内で単体テストとしての位置づけでも行っていたが、仕様確認レビューで画面に対する変更要望の比率が多く、回帰テスト用のテストケースを作り直すのに予想以上に時間がかかった。

そのため、事例プロジェクトでは、回帰テストは画面仕様が安定する結合テスト以降に適用するようにした。

4.5. 人材育成、意識改善等

(1) 早期スキル向上効果

アジャイルプロセスの特徴でもあるチーム開発は、チーム内で助け合うことでお互いのスキル向上に効果的であった。特に新メンバーが加わる時は、ペアプログラミングによるスキル・トランスファーやチームに早く馴染むことができ、コミュニケーションもよくなる。

(2) 品質意識の向上

テスト駆動開発や継続的インテグレーションを適用することで、製作と同時にテストも行い、テストが毎日自動的に繰り返されるため、開発者の品質に対する意識が製作時にも自然と高くなったと思われる。

5. 達成度の評価、取り組みの結果

5.1. 達成度の評価

変更要望を受け入れることを前提としたアジャイルプロセスでは、プログラムの修正を伴うため、何度も再テストが必要になる。

アプリケーションを常に動く状態にしておくことで、実装時に簡単な動作確認をすぐに行えたことが、不良の早期検出に役立った。また、アプリケーションが実際に動く状態を維持することで、結合テストを行う時点で、各モジュール間のインターフェース不良が原因で動作確認が行えなくなるような問題を解消することもできた。

継続的インテグレーションツールによる自動ビルド・自動テストの内容は Web ブラウザから簡単に確認できるため、エラーが発生すれば、すぐに誰かが気付ける状態であり、誰のコミットによってエラーが発生したのかもすぐにわかるため、自分が修正したコードをコミットする際には、品質をしっかりと意識するようになった。

事例プロジェクトから得られた各作業の品質に対する効果を表 15-B-5-3 各施策の評価結果にまとめる。

表 15-B-5-3 各施策の評価結果

実施作業	品質面の効果
イテレーション計画	仕様の全員理解
テスト駆動開発	きれいなコード 不要ロジックの作り込み防止 テストコード再利用
継続的インテグレーション	デグレード防止 バグの早期発見 コードインスペクション テスト自動化による頻繁な再テスト実施 品質に対する意識向上
目視によるコードレビュー	コーディングルールの遵守
画面からの疎通テスト	テスト漏れ防止
仕様確認レビュー	仕様齟齬確認 実装漏れ確認 操作性の早期確認 完成後の手戻り作業防止

5.2. 取り組みの結果

(1) 品質

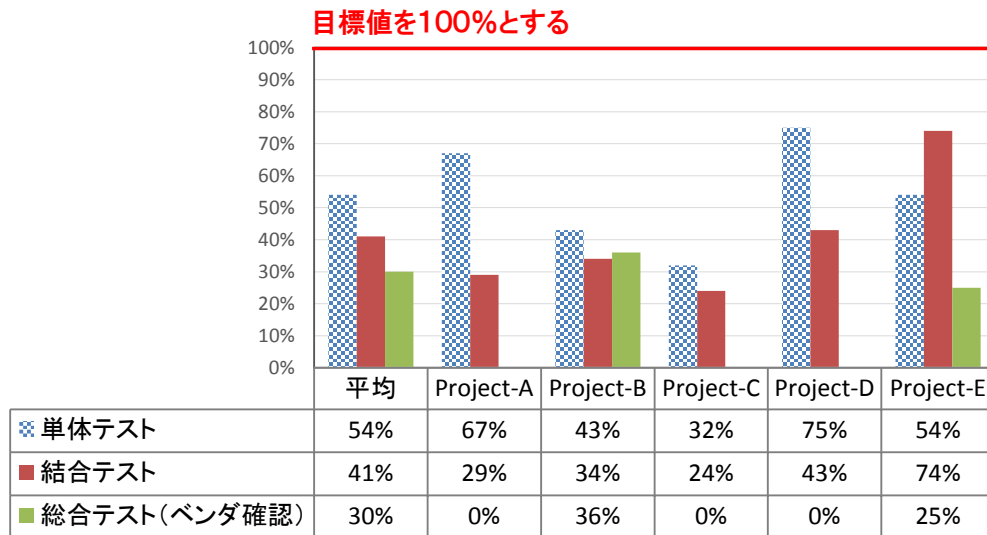
品質に対する効果を評価する上で、今回の事例プロジェクトの結果だけでは、判断できないこともあり、同様の手順で実施した5つのプロジェクトで評価する。

図 15-B-5-9 に示すように、各プロジェクトがウォーターフォールモデルで標準としているバグ摘出目標値（コードステップあたりのバグ摘出件数）に対して、事例プロジェクトから得られた結果をバグ摘出実績比率とする。5つの事例プロジェクトのバグ摘出実績比率の平均では、単体テスト 54%、結合テスト 41%、総合テスト（ベンダ確認）30%と大幅に減らすことができた。バグ摘出実績比率は、各部門で設定しているウォーターフォールモデルのバグ摘出目標値を 100%とし、それに対する実績件数を意味する。

単体テストレベルでバグ抽出率が下がっているが、ウォーターフォールモデルと同様

のテストを行った結合テスト以降でも、大幅にバグ抽出率が下がっていることから、単体テストでのバグ抽出に問題もなく、コーディング段階から高い品質が維持されているといえる。

アジャイルプロセスでよく用いられる施策は、品質向上に効果的なものも多くあり、作業の簡略化だけでなく、それを補う開発テクニックを備えることが、アジャイルプロセスで品質を劣化させないためには必要である。



※ Project-B,E のみ総合テストまでの結果を把握

図 15-B-5-9 バグ抽出目標値に対するバグ抽出実績比率

(2) 生産性

事例プロジェクトでは、詳細設計から製作（単体テスト）までの部分的な工程範囲をアジャイルプロセスの対象としているため、一般的にアジャイルプロセスに期待するほどの生産性向上とはならない。事例プロジェクトでは、詳細設計から製作（単体テスト）までの工程範囲で、設計書の省略・簡略化と自動化の効果で約 90%の工数に削減できた。この値は、本プロジェクトでの結果であり、必ずしも他プロジェクトで同等の効果が得られるとは限らないため、数値の扱いには注意していただきたい。

結合テストと総合テスト（ベンダ確認）工程においては、バグ抽出率が下がっているため、プログラム修正時間全体の短縮が期待できる。

品質向上により生産性向上効果を、次のような前提条件を設定して想定してみる。

バグ抽出率低下による工数削減効果を IPA/SEC 発行の「ソフトウェア開発データ白書 2014-2015」[3]の新規開発時の工数比率のデータ（図 15-B-5-10 の括弧内のパーセント数値）を参考に想定してみる。

- ・ 結合テスト想定条件

テストケース消化時間：バグ修正時間 = 60% : 40%

バグ摘出件数：41%

想定結合テスト作業時間： 60% × 41% = 76.4%

想定結合テスト作業： 17.3% × 76.4% = 13.2%

- ・ 総合テスト（ベンダ確認）想定条件

テストケース消化時間：バグ修正時間 = 60% : 40%

バグ摘出件数：30%

想定結合テスト作業時間： 60% × 30% = 72.0%

想定結合テスト作業： 13.0% × 72.0% = 9.4%

アプリケーション開発の新規開発時の基本設計から総合テスト（ベンダ確認）までの全体工数を 100%とし、各工程比率に対してハイブリッドアジャイルでの工数削減率を適用してみると、全体で 87%に工数を削減できたと推定できる。この値は、「ソフトウェア開発データ白書 2014-2015」の工数比率の平均値と事例プロジェクトから得たデータ、および前提としたテストケース消化とバグ修正の時間比率から算出したものであり、必ずしも他プロジェクトで、同等の効果が得られるとは限らないため、数値の扱いには注意していただきたい。

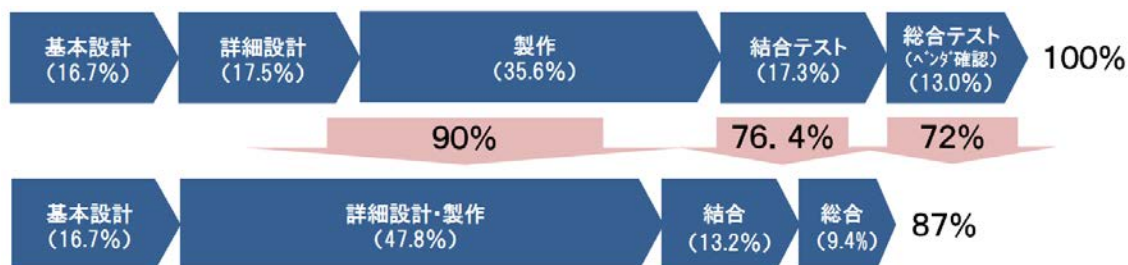


図 15-B-5-10 工程別工数比率（新規開発）

6. 今後の取り組みと考察

アジャイルプロセスでも、品質向上施策をしっかりと行っていれば、生産性を向上しつつ品質向上も行うことができた。事例プロジェクトで適用した品質向上施策は、ウォーターフォールモデルでも適用することは可能である。しかし、ウォーターフォールモデルのままで今回の施策を適用しても、生産性向上は期待できない。

アジャイルプロセスが適用できないと諦めていたプロジェクトに対して、ハイブリッドアジャイルで少しでも生産性向上に繋がっていくという選択肢もあったと思う。

ハイブリッドアジャイルの課題は、変更要望に対する修正作業の効率化によるコスト削減

やリスク予防の効果は得られるが、大幅な開発コストの削減と短期リリースの効果は、あまり大きくは期待できない点である。

Scrum を代表とする一般的なアジャイルプロセスで、結合テストと総合テスト（ベンダ確認）を明確な工程として定義せずに行う場合は、品質担保の方法が課題である。

アプリケーションの重要度やプロジェクトの特徴に合わせて、一般的なアジャイルプロセスとハイブリッドアジャイルを使い分けていくのがよいと思う。

参考文献・引用転載

- [1] 英繁雄 ほか3名（著）、長瀬嘉秀（監修）：ハイブリッドアジャイルの実践、リックテレコム、2013
- [2] Michael James (CollabNet,Inc.) : Scrum Reference Card, CollabNet,Inc., 2010
- [3] ソフトウェア開発データ白書 2014・2015、P227 図表 8-1-13、独立行政法人情報処理推進機構 技術本部 ソフトウェア高信頼化センター（IPA／SEC）、2014

掲載されている会社名・製品名などは、各社の登録商標または商標です。

独立行政法人情報処理推進機構 技術本部 ソフトウェア高信頼化センター（IPA/SEC）