15-B-11

安心なサービスの品質改善を実現する為の継続的システムテスト[1]

1. 概要

ウェブサービスの検索システムの開発においては、検索ランキングを継続的に改善していくための開発とテストプロセスを迅速かつ最小限のリスクで実施していく事が重要である。継続的インテグレーション（CI[2]）の構築等により一定の工数削減等に成功したが、一方で以下のような問題を抱えていた。（問題の詳細は、「2.1 解決すべき課題」 で後述する。）

- 手動のシステムテストでは依然、バグ検出時の手戻りが大きい
- 検索ランキングの評価をプロジェクト終盤まで実施できない

これらの問題を解決するため、機能性だけでなく運用性や可用性のテスト、また検索結果の評価等を行うシステムテストも自動化し、開発プロセスに組み入れる取り組みを行った。

システムテストの自動化では、アプリケーションのデプロイやテストの事前準備などの手順を自動化し、テストの自動実行ツールを開発した。また派生開発におけるソースコードと同じように発展性が求められるテストスクリプトについて、共有スクリプトの考え方を取り入れ、保守性の高いドメイン固有テスト言語（DSTL[3]）を定義した。

次に自動システムテストと開発プロセスを相互作用的に改善していく為の取り組みを実施した。これには欠陥特定にかかる時間の効率化やテストの失敗が機能やテストの実装に与える影響の最小化のための取り組みなどが含まれる。加えて、開発プロセスの改善を継続的に行う為のメトリクスの収集やフィードバックの仕組みを取り入れた。我々はこれらの取り組みを総称して継続的システムテストと呼んでいる。

これら一連の継続的システムテストについての取り組みにより、システムテストで検出された欠陥の修正日数の中央値が 5 日から 2 日へと改善された。また、スプリントの初期の段階で多くの欠陥を検出可能になった。

---

[1] 事例提供: 楽天株式会社 グループコアサービス部　荻野 恒太郎 氏

[2] Continuous Integration

[3] Domain Specific Test Language

2. 取り組みの目的

2.1. 解決すべき課題

ウェブサービスの検索システムを提供する上では、「業務継続に適切な品質」、「業務継続に必要なコスト」、「爆速のデリバリー」の開発体制で検索ランキングのアルゴリズムを継続的に改善して行く事が重要である。しかし、我々の開発プロジェクトはシステムテストの肥大化により、下記の２点のデリバリーに関わる問題が発生していた。

１つ目の課題は、プロジェクトの最後に手動でシステムテストを実行すると、機能、ソースコードの変更と検知された障害の対応関係が複雑になり過ぎ、結合バグの欠陥特定や修正に時間がかかってしまう事であった。検索システムでは一般的なウェブアプリケーションと異なり、ビッグデータの複雑な分散処理をクラウド上で実現する必要がある。そのため、機能性だけでなく図 15-B-11-1 に例示するような運用性や可用性等の非機能性のテストも重要になる。しかしこれらのテストでは、大規模なデータの復旧手順や、データの完全性や一貫性についてのテストが含まれるためその実行には時間がかかる。

２つ目の課題は、検索ランキングの作成が形態素解析、同義語辞書やスコア計算など複数のコンポーネントに相互作用する事にある。図 15-B-11-2 に検索結果の作成に関わるコンポーネントの例を示す。さらに、検索ランキングの作成アルゴリズムはデータやクエリにも依存するため、単体･コンポーネントテストレベルでの評価が難しい。すべてのコンポーネントが正しく動作している状態でなければ検索ランキングの評価は不可能であるが、システムテストの後では大きな手戻りが発生する。

実際、図 15-B-11-3 に示すように、我々のプロジェクトにおいてシステムテストは全体の 33%の工期をしめた。また図 15-B-11-4 に示すように、システムテストでの障害の検出から欠陥の特定、修正までには中央値で 5 日かかっており、単体･コンポーネントテストで見つかった欠陥の修正日数に比べ長い 。

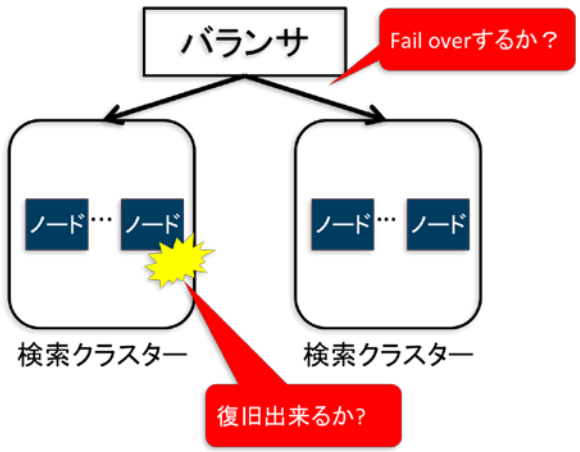


図 15-B-11-1　運用性のテストの例

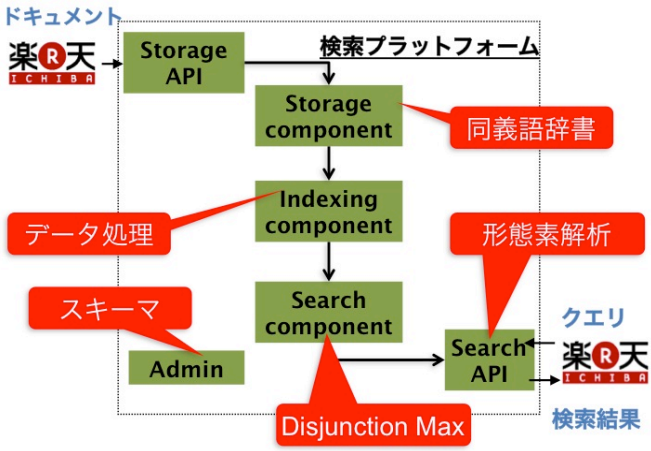


図 15-B-11-2　検索結果の生成に関わるコンポーネント

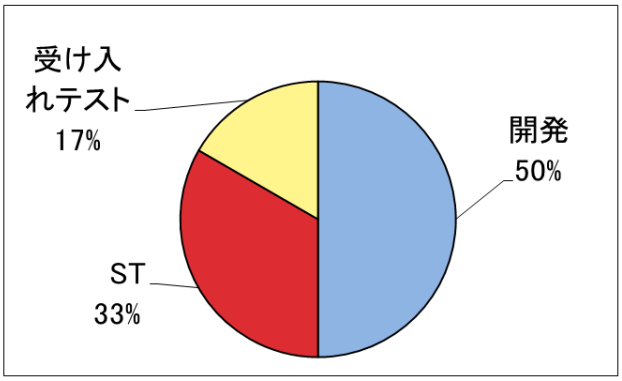
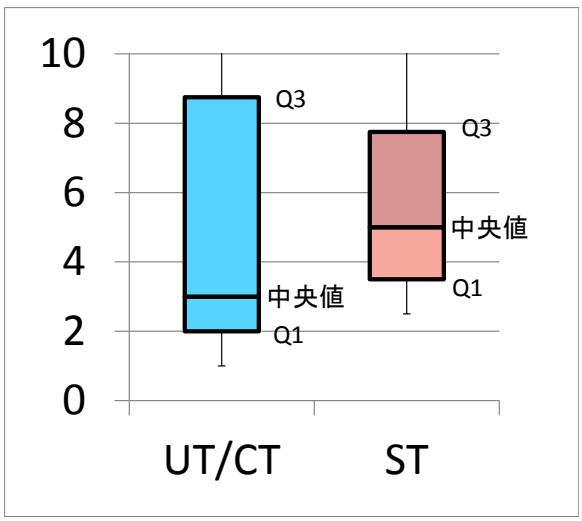


図 15-B-11-3　開発プロジェクト中の各工程の割合



UT（単体テスト）、CT（コンポーネントテスト）、ST（システムテスト）

図 15-B-11-4　各テストレベルでの障害の検出から欠陥の修正までの日数

2.2. 取り組みの目標

これらの課題を解決し、要求分析からリリースまでのリードタイムを改善する事が我々の取り組みの目的である。この目的の達成のため、２つの目標を設定した。

第一の目標はシステムテストでの障害の検出から欠陥の修正までの期間を短縮する事である。自動化によりテスト実行の工程は大幅に短縮可能であるが、テスト全体のプロセスとしてみるとテスト設計や欠陥特定など、まだまだ人手で行わなければならない部分は多い。それらの中でも特に障害のレポートから欠陥の特定、修正までのプロセスを改善する事ができればテスターだけでなく開発者にも大きなメリットとなる。システムテストにおいても、このバグ修正日数の中央値を、単体・コンポーネントテストと同レベルの 3 日へと改善する事を目標とした。

第二の目標は、欠陥の早期検出である。スプリントやバーンダウンチャートなどアジャイルな方法論の導入によりスプリントで完了可能なストーリー数やベロシティなどのソフトウェア開発の予測性が向上しプロジェクトはある程度管理可能になるが、欠陥の予測はまだまだ難しい。特にプロジェクトの後半で予期しない大量の欠陥が検出される事がある。この問題を解決するため欠陥を早期検出する事を目標とした。

これらの目標を達成するため、継続的にコンポーネントの結合、機能性と非機能性ならびに検索ランキングのテストをする継続的システムテスト環境を導入した。

3. 継続的システムテスト

3.1. コンセプト

継続的システムテストは、継続的インテグレーションにより自動化したビルド、単体テストやコンポーネントテスト等のプロセスに加え、機能性や非機能性のシステムテストも自動化する事で、開発期間中に継続的に実行していこうというアジャイルな開発プロセスの考え方の１つである[1][2][3]。継続的システムテストのコンセプトを図 15-B-11-5 に示す。自動化する事により、それまでプロジェクトやスプリントの最後に実行していたシステムテストを設計や実装、単体テストと並行して実行する。

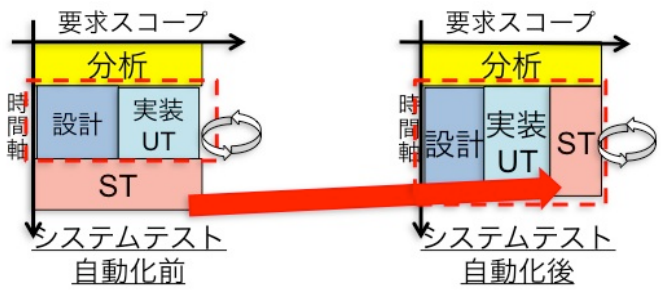


図 15-B-11-5　継続的システムテストのコンセプト

システムテストを継続的に繰り返し実行する事で、ソースコードの変更によって欠陥が混入されてから、その欠陥がシステムテストを失敗させるまでの時間が短縮される。これによりシステムテストが失敗した時に、どのソースコード変更が原因かを推定しやすくなる。すなわち、ソースコードの変更とシステムレベルの欠陥の追跡性が改善される事で、欠陥特定や修正にかかる時間が短縮される事が期待される。また、プロジェクトの初期からシステムテストを実行する事で、早期からの欠陥検出が可能になる事が期待される。

3.2. 継続的システムテストの実現

継続的システムテスト環境を実現するため、既にあった継続的インテグレーション環境を拡張しシステムテストの自動化を行った。

継続的インテグレーションでは単一のソースコードレポジトリに開発者がソースコードをコミットする事により、変更のコンフリクトのリスクを小さくする。またビルドサーバー上で定期的にビルド、単体テストを自動実行する事により、それらをより確かなものにする[4]。

このビルド、単体テストの後工程として手動で実施していたシステムテストを自動化する（図 15-B-11-6）。初めに、我々のドメインである検索や分散システムのテストを記述する事に特化した DSL[4]、すなわち ドメイン固有テスト言語（DSTL）を定義した。また、そのテストを自動実行するテストツールの開発を行った。次にプロダクト、テストそれぞれの安定バージョンを管理する為のスモークテストとバージョン管理の仕組みを取り入れた。また、テスト環境でのスモークテストの前工程として開発者がシステムテストを自動実行する為のテスト環境を開発者側にも用意した。

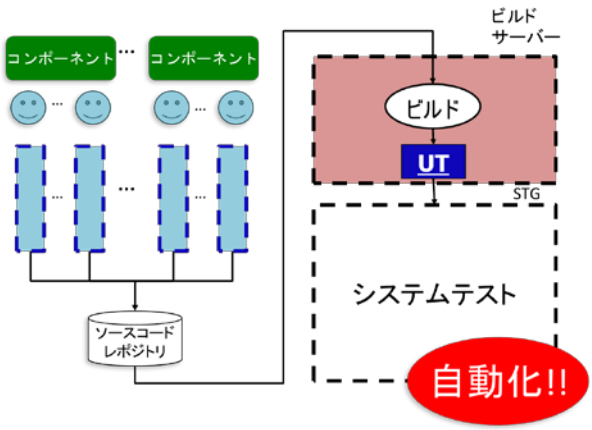


図 15-B-11-6　継続的インテグレーションから継続的システムテストへ

3.3. 継続的システムテストの開発プロセス

継続的システムテストの実現により、それまで実装の後工程として独立に実施されていたシステムテストを日々の開発プロセスに取り込んだ。継続的システムテストの開発プロセスを図 15-B-11-7 に示す。継続的システムテストでの開発プロセスは 、

[4] Domain Specific Language

・　開発者による実装とテストの実行
・　ビルドサーバーによるビルドと単体テストの実行
・　バージョン管理以降のシステムテストの実行

の３つに大きく区分できる。

開発者は開発環境で実装、単体テストの実行と必要なシステムテストを実行する。次にスモークテストにより単純な結合バグがない事を確認し、ソースコード変更をソースコードレポジトリにマージする。

ビルドサーバーはソースコードが変更されたタイミングと、毎晩定時にビルドと単体テストを自動実行する。次にスモークテストを実行し、成功したらそのバージョンの記録、失敗したら開発者への通知を行う。

次にビルドサーバーは最新の安定バージョンをテスト環境にデプロイし、すべてのシステムテストを自動実行する。システムテストには機能テストならびに、運用性や検索ランキング等の非機能テストと大規模データに対するテストが含まれる。システムテストが失敗した場合、開発者への通知が行われる。

この開発プロセスにより、小規模な機能追加や単純なバグ修正については、実装からシステムテストまでを最短で１日で実行する事が可能になった。

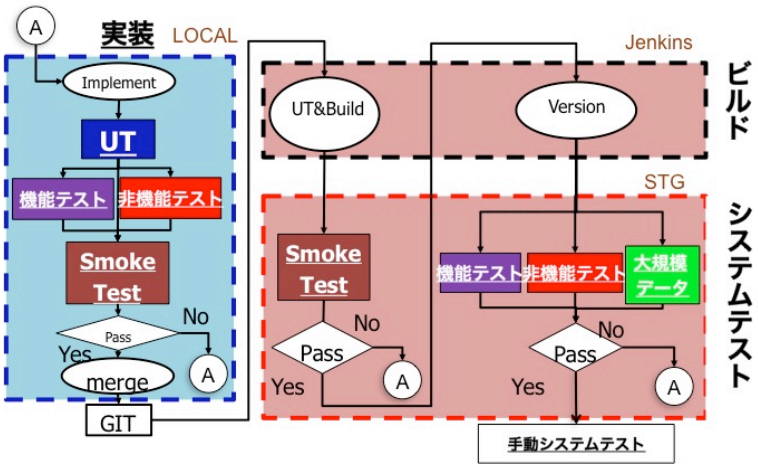


図 15-B-11-7　継続的システムテストの開発プロセス

4. 継続的システムテストの効果を大きくする為の工夫

「3.継続的システムテスト」では継続的システムテストの実現方法と開発プロセスについて説明した。「4.継続的システムテストの効果を大きくする為の工夫」では継続的システムテストによって得られる効果をより大きくする為の品質特性の工夫について述べる。

4.1 ではテストスクリプトの保守性、発展性に関する工夫を、4.2 では開発とテストの相互作用に関する品質についての工夫をそれぞれ説明する。

4.1. テストスクリプトの保守性と発展性

我々のプロジェクトで取り扱っているウェブベースの検索サービスでは、ユーザーからの要望や新しい検索アルゴリズムに対応し、常にサービス改善を行う必要がある。そのため、ソースコードに保守性と発展性が求められるのと同様に、テストスクリプトにおいても保守性と発展性が求められる[5][6]。例えば新規の要求に対しソースコードの変更は数行で済むが、テストスクリプトの変更には数千行かかるという状況は可能な限り避けたい。

テストスクリプトの発展性を確保するため、共有スクリプトと同義の工夫を取り行った。まず「システムのデプロイ」、「システムの起動」、「リクエストの送信」などのロジックと、「対象のシステムのホスト名」、「リクエスト中の値」などのデータを切り分けて管理するようにした。これにより、システムのデプロイ方法が変更された場合に、すべてのスクリプトを変更する必要がなくなり、ロジックとして管理している部分を変更するだけで済むようになる。

また副次的な効果として、テストデータの管理のために必要なテスト対象システムについての知識が少なくてすむ。そのためシステムの専門家ではない検索の専門家が自動テストを作成する際の障壁が低くなるという効果も得る事ができた。

4.2. 開発とテストの相互作用に関する工夫

継続的システムテストにおいてシステムテストは開発から独立しておらず、開発プロセスの中の一つとして実施される。そのためテスト単独の工夫だけでなく、開発側と効果的に相互作用する事も求められる。開発者を自動テスト環境の利用者と捉え、以下の品質特性についての工夫を行った。

4.2.1. 機能要求、ソースコード、テストケースの追跡可能性

テスト自動化の欠点として、障害の解析や欠陥特定により多くの時間がかかる事が挙げられる事がある[6]。テストの失敗から障害を解析し、ソースコード中の欠陥を特定するまでの時間の短縮には、機能要求、ソースコード、テストケースの追跡可能性を向上させる事が重要である。

テストケースが失敗した場合、まずその原因がテストスクリプトと機能要求に起因するかどうかの調査を行う。その時、機能要求やテスト観点に対して 1 対 1 の関係になるようにテスト設計を行っていれば調査が容易になる。

テストの失敗がテストスクリプトと機能要求の欠陥に起因していなかった場合、次にソースコード中の欠陥の存在を疑う。自動システムテストを日次で行っている環境では、テスト実行の直前のソースコード変更の中に欠陥が含まれている可能性が高いと推定できる。そのため、 ソースコード変更の単位が小さくなればなるほど、欠陥特定を容易に行えるようになる。

つまり、図 15-B-11-8 の A に示すように、テストケース、機能やテスト観点とソースコード変更の対応関係が整理されていない場合、テスト失敗の原因のトリアージと欠陥特定に時

間がかかってしまう。機能やソースコード変更を小規模化し、テストケースとの対応関係を整理する事によって追跡可能性が改善する。

これはテスト駆動開発において、最小単位の機能に対しテストと機能の実装とテスト実行を繰り返す事で、ソースコード変更の小規模化や機能とテストケースの追跡可能性を確保する事と同義であると考えられる[7]。

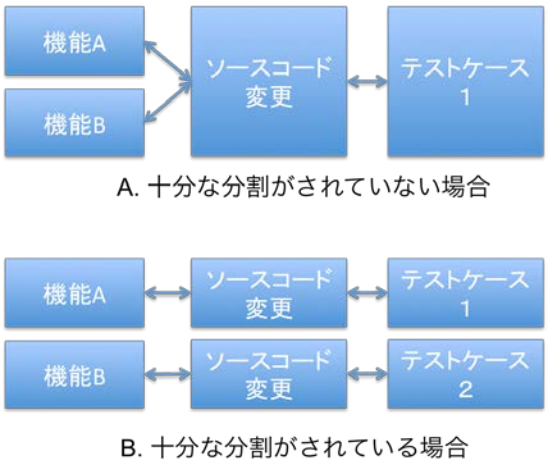


図 15-B-11-8　機能、ソースコードとテストケースの追跡可能性

4.2.2. テスト実行環境の移植性とテスト結果の信頼性についての工夫

開発者とテスターが自動テスト環境をそれぞれ有効利用するためには、テスト実行環境の移植性とテスト結果の信頼性が重要になる[6]。テスト実行環境の移植性を確保する事により、開発者がソースコード変更のコミット前に、テスターのテスト環境とは独立したテスト実行環境でシステムテストを実行することが可能になる。また、テスト結果の再現性を担保する事により、テスターによるテスト実行では失敗するが、開発者によるテスト実行では成功するといった混乱を防ぐ事ができる。

移植性と信頼性を担保するため、まずテスト実行環境の構築手段を仮想化技術により自動化した。これによりテスター、開発者の区別なく同じ環境をコマンド一つで構築できるようになった。また、テストスクリプトからテスト実行環境の情報を分離して独立に管理する事によってテストシナリオの移植性を確保した。

加えて、テストスイートの実行時にアプリケーションやデータの削除によってテスト環境の初期化を行う。テストスクリプト内にはデプロイやデータ投入等のテストの事前準備の手順がすべて自動化、記述されている。

これらの工夫により、「いつでも」「誰でも」「どの環境でも」コマンド一つでテスト環境の構築、事前準備とテストを自動実行し、同じテスト結果を得る事が可能になる。

4.2.3. 開発プロセスの堅牢性

テスト自動化の品質特性として堅牢性が挙げられる[6]。継続的システムテスト環境でも開発プロセスの堅牢性についてのリスクが２つある。１つ目はテスト自動実行ツールも常に拡張開発しているため、バグの混入によりテストの実行自体ができなくなるリスクである。２つ目はシステムテストと平行してソースコードの変更が常に行われているため、システムテストが実行不可能になるような結合バグが混入されるリスクが常にある。これらのリスクを軽減するため、スモークテストによる安定バージョンの管理とテストの優先順位付けを行っている。

システムテストの実行前に自動スモークテストを実行し、安定バージョンのみをシステムテストの対象とする事でテストの堅牢性を向上できる[8]。これにより、例えば機能 A の欠陥による機能 B への影響を軽減している。また、自動テスト実行ツールについても同様のバージョン管理の仕組みを取り入れる事で、ツールに潜むバグの影響を軽減している。

また、「3.継続的システムテスト」で示した図 15-B-11-7 の開発プロセスの通り、開発者はソースコード変更のコミットの前にスモークテストを実行する。この仕組みにより開発者は深刻な結合バグをソースコードのコミットの前に気付く事ができる。

スモークテストで実行されるテストケースはテスト結果の履歴により定期的に更新する。テスト環境の安定度への影響と、実行時間のバランスを考慮しながら優先順位付けをする。

4.2.4. プロセスの継続的な改善

継続的システムテストでは、システムテストを開発から独立したプロセスとして取り扱わず、実装からシステムテストまでを一貫したプロセスとして取り扱う。そのため自動化されたシステムテストが開発に与えている影響をモニタリングし、定期的にプロセス改善に役立てる事が重要である。我々のプロジェクトでは、開発、プロダクトおよび欠陥（バグ）に関するメトリクスを継続的に収集しプロジェクトにフィードバックしている。表 15-B-11-1 に収集しているメトリクスの一部を示す。

これらのメトリクスを分析する事によってプロジェクトの進捗状況の把握に役立てる事ができる[9]。また、欠陥密度やテスト密度を定期的に計算する事で、多すぎず少なすぎない量のテストを実施できているかを評価する為の目安になる。

表 15-B-11-1　収集しているメトリクスの例

グループ	メトリクス名	単位
開発メトリクス	日次のコミット数	回数
	日次のコミットサイズ	行数
プロダクトメトリクス	日次の LOC	行数
	日次の変更 LOC 日次の追加 LOC 日次の削除 LOC 日次の変更無し LOC	行数
	日次の変更ファイル 日次の追加ファイル 日次の削除ファイル 日次の変更なしファイル	ファイル数
バグメトリクス	日次のプロダクトの検出バグ数	個数

5. 評価

5.1. 事前評価

2.2 で挙げた目標に対する評価の前に、継続的システムテストのコンセプトを実現できているかどうかについての事前評価を取り上げる。

5.1.1. 継続的なソースコード変更と欠陥の検出

継続的システムテストのコンセプトである機能追加と欠陥の検出を継続、平行して行う開発プロセスが実現できているかどうか調べた。図 15-B-11-9 にプロジェクト期間 283 日の累積コミット数と累積検出バグ数を示す。このプロジェクトは

- 大きな機能要件に対応する A
- 断続的な小さな要件に対応する B
- システムレベルのリファクタリングを行っていた C

の 3 つの開発フェーズがあった。一年を通して累積検出バグ数、累積コミット数ともに増加しており、開発とテストのプロセスがともに継続して進捗していた事が分かる。

また、断続的な小さな要件に対応していた B の期間ではバグ検出の増加量が他の 2 つの期間に比べ小さい。これは 1 つのコンポーネントに閉じた小さな機能が多かったので、そもそもシステムテストで検出される結合バグが少なかったからだと考えられる。

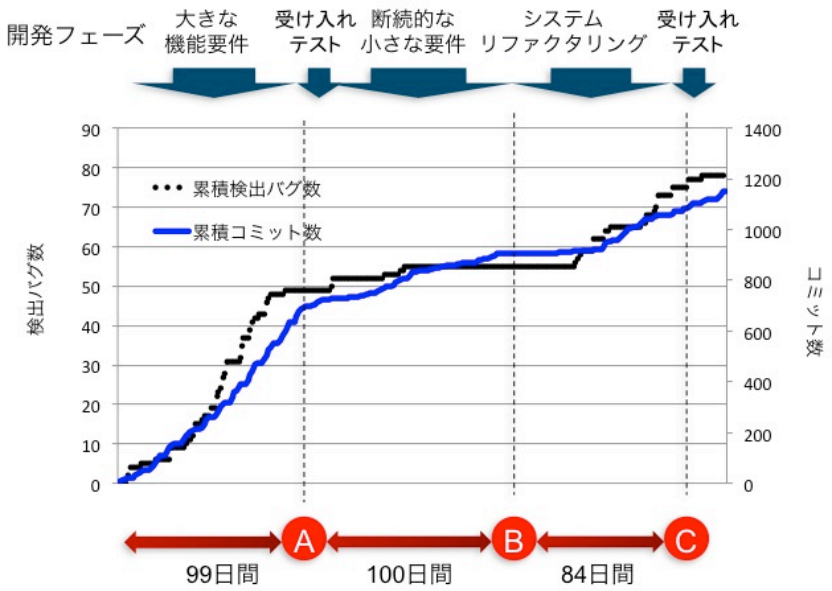


図 15-B-11-9　累積検出バグ数と累積コミット数

5.1.2. テスト密度とバグ密度

自動化したシステムテストにより十分に欠陥の検出が行えているか調べた。テスト密度とバグ密度の値を上記の開発フェーズ A、B、C で算出し、IPA が提供する「ソフトウェア開発データ白書 2012-2013」の業界標準の統計値との比較を行った[10]。この資料では、新規開発と改良開発において異なる統計データが提供されている。我々の開発プロジェクトでは A が新規開発、B と C が改良開発にそれぞれ相当する。テスト密度、バグ密度それぞれの結果を図 15-B-11-10 と図 15-B-11-11 に示す。

テスト密度の結果を見ると、テスト密度はほぼ業界標準内の値である事が分かる。また、A、B、C とフェーズを経るごとに上昇している。これは自動テスト環境の成熟とともに自動テストの追加が容易になっているためである。一方で C では業界標準の範囲よりも若干高い。テストを作りすぎていないかについて注意する必要がある。

バグ密度は A、B、C を通して業界標準の値の中にある。興味深いのは機能の追加を行っていた B の期間よりも、システムのリファクタリングを行っていた C の期間の方がバグ密度が高い事である。システムレベルのリファクタリングについても自動テスト本来のリグレッションテストとしての役割をきちんと果たしていた事が推測できる。

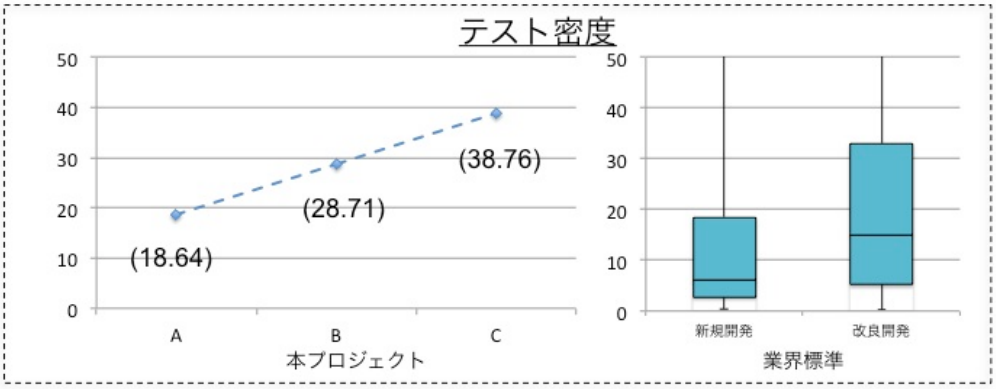


図 15-B-11-10　テスト密度

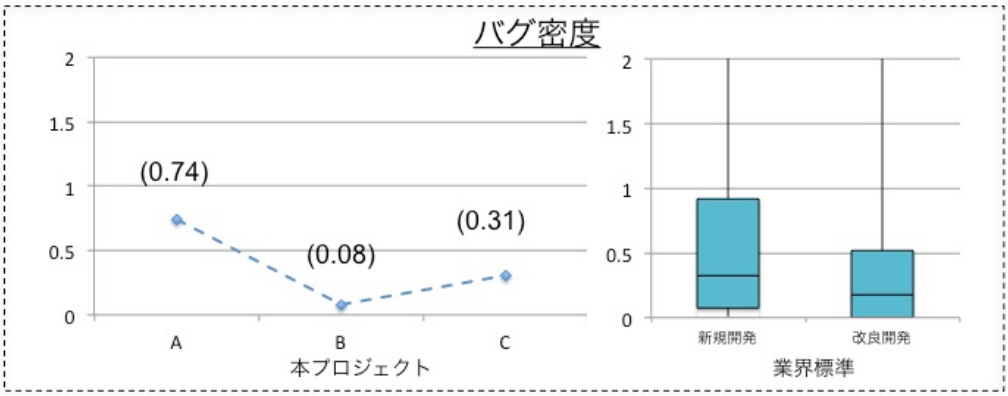


図 15-B-11-11　バグ密度

5.2. 目標に対する評価

2.2 で設定したバグ修正日数の短縮と、プロジェクトの早期からの欠陥の検出の評価を行った。

5.2.1. バグ修正日数

図 15-B-11-12 に示すように、システムテスト自動化前後で、バグの修正日数の中央値が 5 日から 2 日へと大きく改善し、目標である 3 日を上回った。これは多くのバグについて、バグ混入の翌朝にテストが失敗し、バグ票の作成とバグの修正が行われ、ナイトリービルドで修正が確認されている事を示している。

この結果が得られた理由として、継続的なシステムテストの実行によってソースコード変更とテスト結果の追跡性が大幅に向上し、テストを失敗させたソースコードの変更部分の特定が容易になった事が大きいと考えられる。また、開発者が QA[5]側との調整を気にせず自分たちのタイミングでシステムテストを実行できるようになった事も影響している。

[5] Quality Assurance

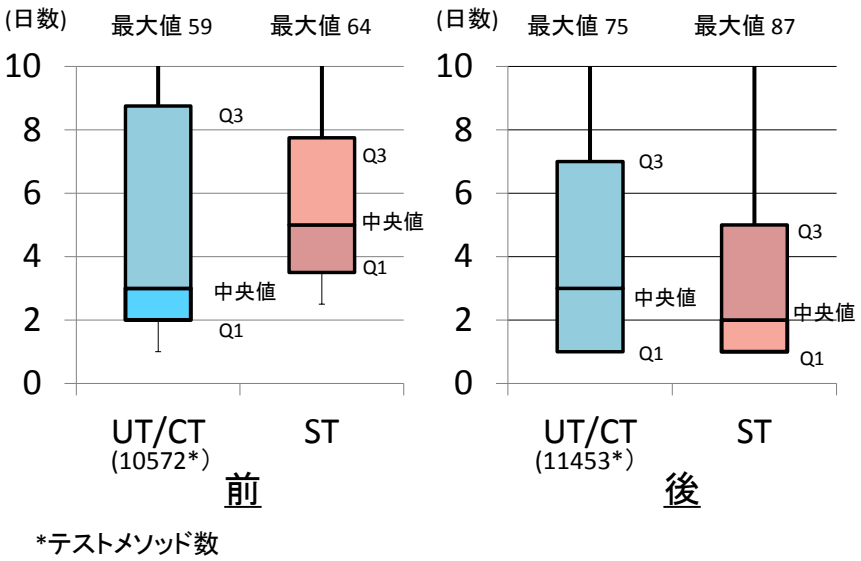


図 15-B-11-12　システムテスト自動化前後でのバグの修正日数

5.2.2. バグカーブ

プロジェクトのフェーズごとのバグ曲線について、累積コミット数を横軸にとったものを図 15-B-11-13 に示す。フェーズ A ではバグがプロジェクト開始から終了直前まで見つかっているが、フェーズ C では各イテレーションの開始直後に大部分のバグを検出する事に成功している。これは前述した開発とテストの相互作用的なプロセス改善の総合的な結果であると考えられる。

フェーズ C では、システムテストでのバグを早期に発見、修正する事により、イテレーションの中盤以降は検索ランキングの改善に集中できるようになった。継続的な改善を安心して実施できる環境を開発者に提供できる事が、継続的システムテストの最大のメリットであると考えられる。

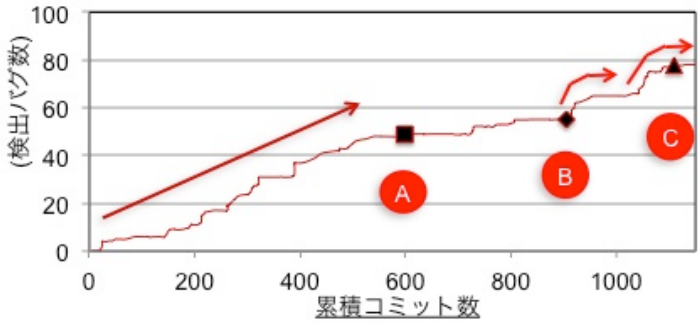


図 15-B-11-13　バグカーブ

6. まとめと今後の取り組み

ウェブベースの検索システム開発プロジェクトにおけるデリバリーの問題を改善するために、継続的システムテストを導入した。システムの運用性や可用性、また検索結果に関するテストを含むシステムテストを自動化し、開発プロセス全体のプロセス改善を行った。継続的システムテストの効果をより大きなものにするため、テストスクリプトの発展性や開発側と相互作用するための品質について工夫を実施した。

結果として、システムテストで検出された欠陥の修正日数は中央値で 5 日から 2 日に改善された。また、総合的な開発プロセス改善の結果として、早期からの欠陥の検出が可能になった。

この事例においてシステムテスト自動化の導入がうまくいった理由として、システムテストを開発プロセスの一部として捉えた事が挙げられる。一般的にシステムテスト自動化失敗の原因として開発と QA の対立構造が挙げられる事が多い。本事例では開発プロセスの改善に重点を置いたため開発者の協力が得られやすかった。また、テスト自動化に取り組んだチームはもともとプロダクト側の設計や実装を行っていたメンバーである事もプラスに働いていたと考えられる。

また、継続的システムテスト導入による開発プロセスの変化について、メトリクスによる客観的なフィードバックを定期的に行っていた事も大きい。メトリクスによるフィードバックにより、機能分割が適切かどうか、またテストを作りすぎていないか等について一定の目安を得る事ができる。

一方で、継続的システムテストをより効果的なものにしていくための課題も残っている。

第一の課題は、信頼度成長曲線のようなテストの十分性を評価する為の仕組みが、継続的にシステムテストを実施している環境では確立されていない事である。テスト自動化が成熟している環境ではテストケースの追加が容易であるため、テスターが直感と経験から不安を感じた部分については多めにテストを追加する。しかしテストの経済学の面から考えると余分なテストが追加されている可能性がある。

また第二の課題は、テスト設計とテストスクリプトの作成はテスター個人のスキルに依存しており、属人性が高いということである。テスト設計からテストスクリプト作成までのプロセスの一般化と自動化がこの問題を解決するためのヒントになると考えられる。

参考文献

[1] 平鍋健児: ソフトウェア工学の分岐点における、アジャイルの役割, 第 30 回ソフトウェア･シンポジウム,2010、http://sea.jp/Events/symposium/ss2010/talks/hiranabe.ppt

[2] 荻野恒太郎 他 3 名: システムテストの自動化による大規模分散検索プラットフォームの開発工程改善、JaSST2014 Tokyo 予稿集、2014、
http://jasst.jp/symposium/jasst14tokyo/pdf/D2-2.pdf

[3] 荻野恒太郎: 継続的システムテストについての理解を深めるための開発とバグのメトリクスの分析, ソフトウェア品質シンポジウム 2014 発表資料、2014
http://www.juse.jp/sqip/symposium/2014/timetable/files/happyou_A1-3.pdf (2015/11/18)

[4] Martin Fowler: Continuous Integration,
http://martinfowler.com/articles/continuousIntegration.html

[5] Ciraci, Selim and Broek, Pim van den: Evolvability as a Quality Attribute of Software Architectures, In: International ERCIM Workshop on Software Evolution 2006, pp.pp.29-31, France, 2006.

[6] Mark Fewster, Dorothy Graham 著、テスト自動化研究会 訳：システムテスト自動化標準ガイド、翔泳社、2014.12

[7] Jonathan Ramusson 著、西村直人,角谷信太郎 監訳、近藤修平,角掛拓未 訳：アジャイルサムライ −達人開発者への道−、2011 年 7 月

[8] David Farley、Jez Humble 著、和智右桂,、高木正弘 訳：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

[9] SQuBOK 策定部会：ソフトウェア品質知識体系ガイド - SQuBOK Guide、オーム社、2007.12

[10] 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）: SECBOOKS ソフトウェア開発データ白書 2012-2013、2012.10.1