「テストからはじめよ」
～忍者式テスト 20 年の実践から～

深谷 美和
キヤノンメディカルシステムズ株式会社
miwa.fukaya@medical.canon

関 将俊
キヤノンメディカルシステムズ株式会社
masatoshi.seki@medical.canon

要旨

1999 年に出版された eXtreme Programming[1](以下 XP)に端を発したアジャイル開発は世界中で広く普及し，アジャイル開発の基礎となる反復開発，テスト駆動開発[2]も広く知られることとなった．

私たちのチームは X 線 CT 装置を XP で開発している．本稿では，私たちの 20 年の実践から，反復開発の利点と反復開発に有効なプラクティス「忍者式テスト」を説明する．

1. はじめに

Fig. 1 は Royce による大規模ソフトウェア開発のための実行ステップである．システム要求からコーディングまでの作り出すステップと，テスティングの確認するステップで構成される．

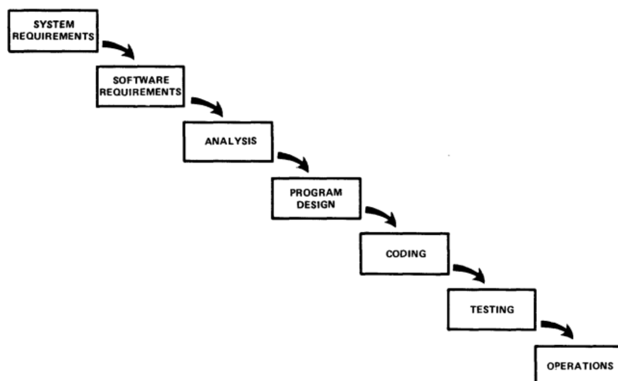

Fig. 1 Implementation steps to develop a large computer program for delivery to a customer.（参考文献[3] Figure 2.を引用）

Fig. 1 では確認するステップはテスティングのみである．テスティングを展開し，各ステップに対応するテストを配置すると V 字モデル(Fig. 2)となる．

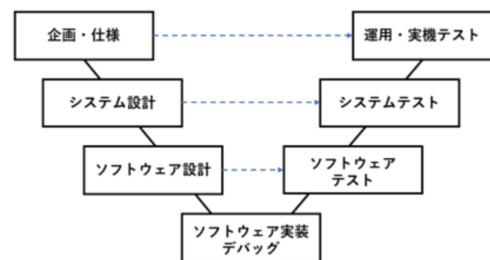

Fig. 2 V 字モデル（工程名は IPA[4]に合わせた）

実際のソフトウェア開発は各ステップを1回やって終わりではない．ステップは1つずつ進むが，問題が見つかるとn ステップ前に戻る(Fig. 3)．たとえば，ソフトウェアテストで問題が見つかり，ソフトウェア設計に戻った結果，さらにシステム設計まで戻るケースもある．

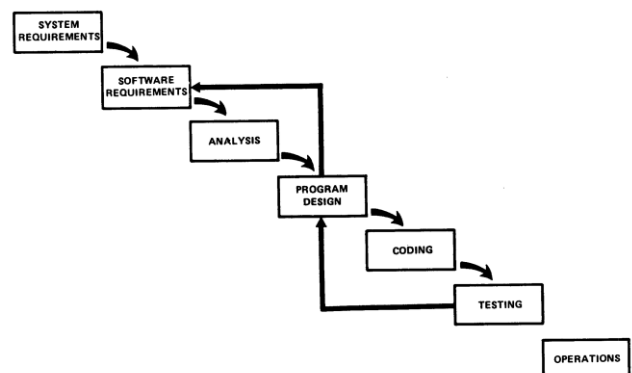

Fig. 3 Unfortunately, for the process illustrated, the design iterations are never confined to the successive steps.（参考文献[3] Figure 4.を引用）

このように作り出すステップで仮説を立て，確認するステップで検証する．これを繰り返すことで，開発は進んでいく．開発は単にモノを組み立てることではなく，製品を理解していく行為と言える．

これらのモデルは，各ステップの依存関係を示したものであるが，時系列のモデルと勘違いされることがある．たとえば，前のステップへ戻らないことを前提にしたスケジュールを立ててしまうケースである．「確認するステップ」で見つかる問題を修正し，それを確認する期間を確保していない．確保していたとしても「作り出すステップ」が遅延した場合のバッファと考えている．

1.1. ステップごとの問題発見能力の違い

ここで，IPA[4]を参照して，数字から分かる事実を二つ述べる．一つ目は不具合の原因の 73%が「作り出すステップ」にあるが，「確認するステップ」で 65%の不具合を発見している（Fig. 4）．不具合の原因工程と不具合の発見工程（Fig. 5）からも分かるように，企画・仕様，設計の問題をその工程で見つけるのは難しいのである．「作り出すステップ」がサボっているのではなく難しいという事実を関係者全員が真摯に受け止めなければならない．二つ目はテストは問題を見つける能力が高いということである．「作り出すステップ」で試さずに会議やレビューだけですべての問題を発見するのは高コストであり，試してみたほうが安いことを示している．これはソフトウェアの特徴のひとつである．

これを利用し「確認するステップ」をプロジェクトの初期の段階から実施すれば，早い時期に問題を発見する可能性が高まる．早い時期に問題を発見できれば修正コストも安くなる．

> 問題の発見が早ければ早いほどそれを修正できる可能性が高くなる．修正コストは開発の進捗に従って劇的に増加していく．統合とテストを最終段階で行うアプローチは，統合段階に到達しなければ実際の作業状況を認識できないという問題がある．[7]

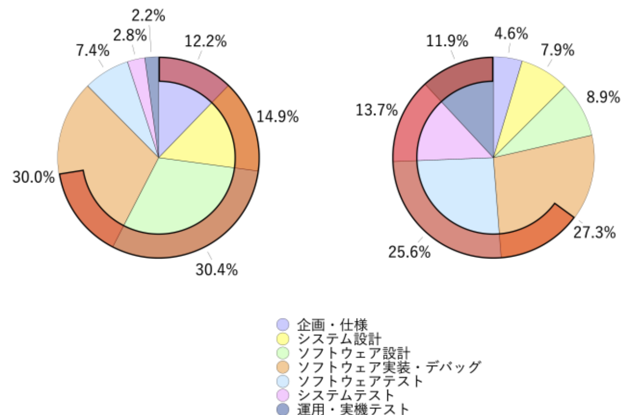

Fig. 4 不具合の原因工程の比率，発見工程の比率（参考文献[4]の図 Q5-4I, Q5-4II を引用，情報を追記）

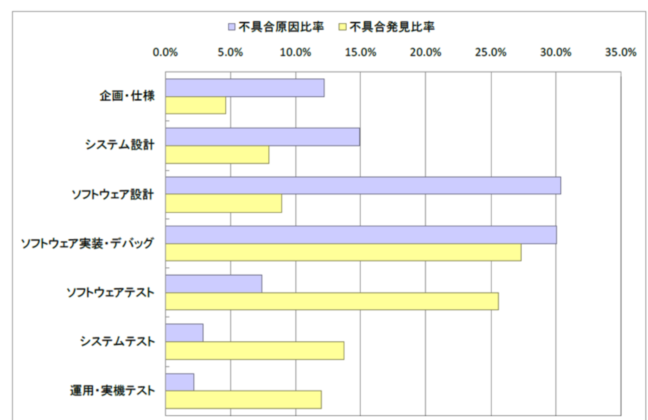

Fig. 5 不具合の原因工程と不具合の発見工程（参考文献[4]の図 Q5-4 を引用）

2. 実施内容・方法

反復開発の考え方と，忍者式テストを中心とした反復開発の実践の様子を述べる．

2.1. 反復開発

V 字モデルの後半の「確認するステップ」をプロジェクトの早い時期から実施するため，プロジェクト全体を n 回の V 字で構成する．これを反復開発と呼ぶ．各 V 字ではすべてのステップを行う．

初回の V 字に注目するとプロジェクトの初期に「確認

するステップ」が実行されるのが分かる（Fig. 6）．次の V 字には前回の V 字で発見した問題や違和感，知見をフィードバッグできる．初期の V 字で開発した機能は確実に搭載されるうえ，より長い期間，なんどもテストされることになる．

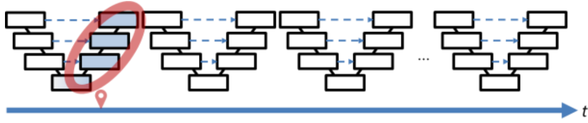

Fig. 6 反復開発

これを成果物の視点で言い換えると次のようになる．まず小さな動くシステムから始める（V.0.0）．そのシステムになんらかの機能を追加し，次の完結したシステム（V.0.1）を作る．これが 1 反復である．これを繰り返していく．レイヤーごとに部品をそろえて最後に結合するスタイルではなく，Fig. 7 のようにすべてのレイヤーを対象にその反復に必要な部分を変更して都度結合するスタイルである．

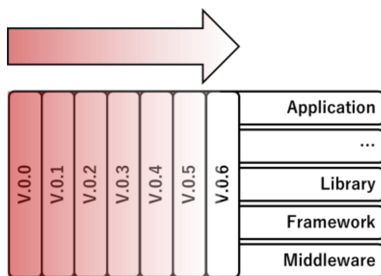

Fig. 7 作る順序（反復開発）

反復開発で反復ごとに結合してシステム全体を確かめるのは，製品のバージョンアップ開発のシステムテストと考え方は同じである．新しい反復により改定された仕様の確認と，それまでに搭載したものが問題なく動作することを確認するテストが必須になる．Fig. 7 でいうと V.0.6 時点での確認は V.0.6 で変更した内容だけでなく，V.0.0 から V.0.5 で搭載したものが問題なく動くことの確認も必要である．反復開発でも一般的なバージョンアップ開発と同様に，反復ごとにシステム全体をカバーするテストが非常に重要であり，これはシステムを安全に保つための生命線である．短い周期で反復すると，頻繁にシステムテスト，運用･実機テストを実施することになり負担が増えることが想像できる．本稿で述べる「忍者式テスト」は，このテストを行う活動である．

次に忍者式テストと，これを行うために必要なプラクティスについて説明する．

2.2. 忍者式テスト

設計する段階ですべての問題を発見することは難しく，またテストは問題を発見する能力が高いことを説明した．XP においても「テスト」は主役であり，あらゆる場合に登場する．

エクストリームプログラミング[1] 第 5 章 原則に自己相似形についての説明がある．

> 先に失敗するテストを書いてから，それを動かすという開発の基本的なリズムがある．これはあらゆる規模で作用する．

あらゆる規模というのは，プログラミングの場面だけではなく，ひとかたまりの機能であったり，大きな要求であったり，プロジェクトの進め方であったり，という意味である．

私たちのチームでは「うまくいったらどうなるの？」という問いかけからはじめる．うまくいったとしたらどうなるのか，それをどう試すのかを考えて，どうやら確かめることができそうだ，とわかってから実行する．確かめかたを考えるだけでも多くの問題を発見することができる．テスト駆動開発において，よいテストコードを考えることによる副次的な効果として，よいインターフェースが発見されたりすることとよく似ている．

これから述べる忍者式テストは「テストファーストプログラミング」「テスト駆動開発」の相似形である．プログラミングの場面でうまくいくアイデアを，要求･機能の場面へ拡張する．

エクストリームプログラミング[1]では多くの場面でテストが「たいくつなもの」「サボってしまうもの」「ストレスフルなもの」として登場し，プログラミングによって「自動化できるもの」と説明されている．なんとかしてプログラマーにテストを定着させようという努力が感じられる．

しかし私たちのチームのプログラマーにとって，テストはたいくつなものでもないし，サボるようなものでもないし，ストレスフルでもない．自動化できない領域も積極的にテストしていきたいと考えている．

2.2.1. システムを変更する最小単位＝ストーリー

ここでは私たちのチームのストーリーについて説明する．

反復開発では，開発全体をひとつの大きなかたまりで考えるのではなく，既存システムへの小さな変更の連続として考える．小さな変更ごとに V 字の全ステップを行う．

この小さな変更をストーリーと呼び，それらをチケットで管理する．ストーリーは，システムがどのように変わったのかをユーザーが目で見て分かるものでなければならない．つまり，システムテストで確認できることが求められる．これは「作り出すステップ」のフェーズから「確認するステップ」を考えるということである．このストーリーができるとユーザーは何が嬉しいのか，ユーザーが困っていることは解決できるのか，意図したとおりに作れたことはどう試すと分かるのか，といった議論からはじめる．ストーリーを考えるとき，私たちはテストから考えている．

XP と異なる点がある．XP ではストーリーという単位で計画し，ストーリーには複数のタスクが紐づいている．このタスクを変更の最小単位としている．タスクが完了してもストーリーが完了していない状態がある．一方，私たちはタスクを使用せず，ストーリーを変更の最小単位としている．システムテストで確認できるものだけを変更の単位と見なす．

2.2.2. ストーリーの大きさ

ストーリーの大きさ，粒度は，実際に開発しながら扱いやすい大きさに調整し続けていくものである．

私たちが扱うストーリーの多くは数日で完成する大きさである．小さな変更がシステムにどのような影響を及ぼすのかを議論しながら，V 字の各ステップを行き来して完成させる．この数日間は結合して試せない期間であり，開発のリスクとして全員が認識しておく．よって，ストーリーは小さいほうがよい．ただし，目で見て変化の様子が分かる程度の大きさは必要である．

大きな要求をストーリーに分割し開発の順序を計画する．この分割や順序が適切でない場合は，ストーリーがなかなか完成しない，試し方が思いつかない，制約が多すぎて本来のストーリーのテーマが試せないなど，さまざまなシグナルがあがる．シグナルに気づいたら，そのつど計画の見直しや，ストーリーの再構築を行う．

2.2.3. チケットでストーリーとテストケースを紐づける

ストーリーはチケットで管理されている．チケットには一般的な項目（タイトル，概要，担当者，ステータスなど）の他に，テストケースを書く．これは，新しいストーリーを搭載したことによるシステムの具体的な変化と，その確認方法である．テストケースはストーリーと同一のチケットに書くため自動的に紐づく．

また，開発中に見つけたバグの修正もシステムへの変更と考え，ストーリーと同様に扱い，チケットで管理する．バグ管理システムで別管理ということはしない．よって，ストーリーと同じように，バグを修正したことが分かる具体的な変化と確認方法をテストケースとしてチケットに書き，バグとテストケースを紐づける．

テストケースはストーリーを設計，実装したプログラマーが記載する．1 つのストーリーは複数のプログラマーで担当することが多く，テストケースの作成もその過程で行う．V 字の期間中にもたらされたさまざまな議論やテスターから示された具体的なシナリオなどもテストケース作成の入力になる．

テストケースを記載するタイミングはプログラマーに任せているが，いくらソースコードをコミットしていてもテストケースがないと完成とは見なさない．チケットには必ずテストケースをつける．

初回のシステムテストはテストケースを記載したプログラマー自身によって実施される．その後は，チームのプログラマーやテスターによる容赦ないテストが行われる．その際，システムの動作だけでなく，同時にテストケースの内容が適切かどうかもテストする．テストケース自体も疑っているのである．

これらのチケットにはテストケースの他に，要求の背景や，日々のシステムの変化の様子（以降，開発日記）が書かれている．ToDo リストではなく，具体的な作業内容の記録であり，なぜそのような設計や実装にしたかの根拠，実験した内容や結果，懸念点や解決法やそれを選択した理由，プログラムコード片など多岐にわたる．開発日記は朝会で V 字の期間中，前回との差分（たいていは前日分）を全員で読み合わせている．その様子は設計レビューに近い．

2.2.4. システムがうまく動いている根拠

反復開発では，反復ごとにシステム全体をカバーするテストが非常に重要である．

今日のシステムはこれまで作ってきたすべてのストーリ

ーとすべてのバグの修正からなっている．ということは，システム全体をカバーするテストとは，今日までに累積されたすべてのチケットに書かれたテストケースであると言える．

そこで私たちは毎日すべてのチケットをテストし直すことにした．すべてのチケットのテストケースをパスすることが，今日のシステムがうまく動いている根拠になるからである．毎日ストーリーは増え続け，試すべきテストケースも増えていく．しかしそれでも私たちはテストし続けるのである．

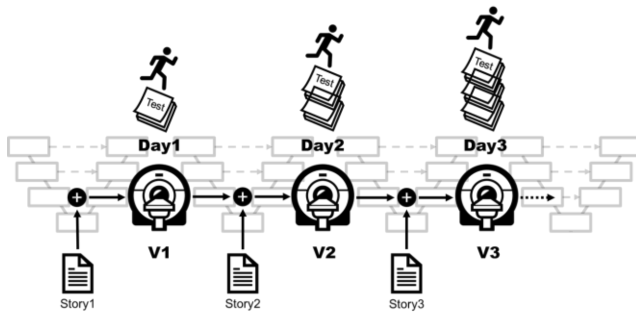

Fig. 8 忍者式テストの日々

忍者式という名前は，忍者が毎日成長する麻や竹の上を飛び越える修行に由来している．毎日増えるテストケースの束を毎日全部やり直す様子が似ているのである．

2.3. 手動テストを採用している理由

私たちのテストは，ストーリーに書かれているテーマを起点として「本当によいシステムになっているのか」について，本物のシステムを使って確かめる過程である．操作手順のような基本シーケンスだけでなく，チケットに書かれた開発日記（要求の背景，開発の経緯など）を読み直し，実際に今日のシステムで操作して，問題がないか，さらには，本当によいものになっているのかを実験する．単に不具合を見つけたいだけでなく，使いづらさや誤解をまねく仕様や手触りの良し悪しなども含めた理想との違いを知りたいのである．その理想さえも市場や環境の変化に応じて変わっていく可能性がある．

ストーリーに書いてあるテストを実施して分かるのは“テストケースに書かれている前提や操作において”うまく動くか，動かないかである．変更による意図しない副作用を検出するためのリグレッションテストとしての効果はある．しかし，私たちが手に入れたい問題を発見するには，このテストケースだけでは足りない．そのために，書かれているテストケースを出発点として，さらに新しいテストケースを考えながらテストしなくてはならない．これらを実現するには手動でのテストが適している．

2.4. 問題を見つけたらどうするか

理想の製品との違いを見つけたときにどうするかについて述べる．

仕様やふるまいについて，ちょっとした違和感や気になる挙動を見つけた場合は，良い仕様とはどのようなものかを議論する．必要と判断したらストーリーとしてチケットを発行する．

不具合の場合は見かけの現象によらず，原因の特定までを最優先で行う．システムにどのような影響があるかは発見された現象だけではわからない．いくら軽微な現象に見えても，重大な障害を引き起こす可能性のある実装になっている場合もあるからだ．原因を特定する過程で，どのチケットで問題が入ったのかを調べ，さらに既存のテストケースで検出できなかった理由も明らかにする．

テストケース自体の問題を発見する場合もある．テストケースの過不足の他に，ストーリーが増えることによって既存のテストケースが影響を受けるときがある．新しいストーリーが仕様を上書きし，操作手順などが変わった場合は，現在のシステムにあわせてテストケースを修正する．また上書きによってテストケースが不要になったら，その旨をチケットに記載し，テスト対象から外す．

毎日すべてのチケットをテストしながらテスト自体も見直し，テストケースを最新の状態にしている．

2.5. 規模に対する工夫

忍者式テストをはじめた 20 年前はチケットの数も少なく 1 日ですべてのテストケースを確かめることができていた．しかし，開発が進みチケットが増えるにつれテストケースも増加し，1 日で回せなくなってきた．そこで，一定期間でテストケースをすべて回す作戦に切り替えた．効果の高いテストケースを優先的に，かつ，一定の期間ですべてのテストケースを試せるアルゴリズムを開発した．まだテストされていない新しいストーリー，修正したばかりのチケットのテストケースを最優先とし，前回のテスト結果がパスしたテストケースの頻度を徐々に落とす，といった具合である．また，開発の状況に応じて機能ごとに出現頻度を調整している．

このアルゴリズムを用いると，新しい機能，問題のあっ

た機能を優先にしつつ，システム全体をある期間でテストすることができる．アルゴリズムにより抽出された今日のテストスイートを「本日のおすすめテスト」と呼んでいる．

Fig. 9 に，私たちの 20 年分の主たる開発におけるテスト実施の記録を元に作成した散布図を示す．

散布図の横軸は営業日，縦軸はチケットの番号である．ひとつひとつの点がテストの実施を示している．水色はテスト結果が OK，赤色は NG になったものである．なお，1 つのチケットには複数のテストケースが書かれているため，テストケースの数ではない．ここから読み取れることを，いくつか述べる．

(1) 開発初期はすべてのチケットのテストを実施しているため色付けが濃い．開発が進みチケットが増えていくと色付けがだんだん薄くなる．全体を見ると，まんべんなくすべてのテストケースを実施している．（Fig. 10）

(2) 営業日と完成したチケットの数がリニアに増えている．システムが複雑になり，要求も高度になっているが，チケットの増え方は一定である．これは同じペースで開発していることを示している．XP の仮説のとおり，規模が大きくなっても変更コストは一定である．（Fig. 11）

(3) グラフの稜線のあたりに赤色が散見される．これは新しいチケットがしばしば NG になっていることを示している．作る前のレビュー，V 字の期間中に議論を重ねてもなお，実際に触って分かることがある事実と，さらに，テストも手を緩めず実施していることを示している．（Fig. 12）

(4) 3800 日目くらいから下半分に薄い部分が増えるが，次世代の製品にシフトした時期と一致している．事業計画に従って，テストの注力点を調整しているのが履歴からも見て取れる．その後，徐々に全体を試すように回復している．（Fig. 13）

20 年前に作成したテストケースを読み，開発日記で当時どのようなことが起きたのかを知り，今日のシステムで試すことは私たちの日常である．これはまた自分が書いた開発日記やテストケースが後世に渡って何度も読まれ，試され続けるものであると実感する瞬間でもある．

2.6. プログラマーによるシステムテスト

私たちのチームではプログラマーに毎日1時間，システムテストの時間を割り当てている．前述した「本日のおすすめテスト」にしたがって，最新のシステムでテストする．

問題を発見するという本来の効果の他に，副次的な利点がある．それは，システムを学習または再学習し，製品を知り，理解を深められる点である．

私たちの製品は開発規模が大きい．製品について知っていることもあれば知らないこともある．プログラマーによるシステムテストは，知らないことを学習する機会になるし，知っていることを再学習する機会になる．過去の開発は記憶が薄れがちである．それをはっきりと思い出し，自身の記憶を再構築するのである．

実際にテストしている様子を観察してみると，単にテストケースを実行するだけでなく，開発日記を読んでいる．これは，対象のストーリーがどのような意図，経緯で作成したのか、当時の開発で起きていたことを知ろうとしている．バグ由来のテストケースの場合は，過去にどのような経緯でバグが混入し，それの原因と対処を読み取るであろう．

このような学習を毎日繰り返すことで，製品やシステムに対する理解や知識を積み重ねている．テストケースに書かれていないことを試せる，そして，未知の問題を見つけられるのは，これらの学習の成果だと思う．この成果は製品開発のすべてのフェーズに活かされているだろうと想像する．

2.7. 反復開発と規模と組織

反復開発を実施してきた経験からさまざまなことに気づいた．ここでは，規模と組織に関するテーマを挙げ，それらについての考え方を述べる．

2.7.1. スケールアップと実際のヒント

反復開発の V 字が十分小さくなると，二つの点から，スケールしやすくなる．なお，ここでのスケールとは増員分だけ規模が増やせることである．ひとつは，時期による負荷の偏りがない点，もうひとつは1チームで複数バージョンの並行開発が可能な点である．

反復開発ではプロジェクトの期間中ずっと各ステップの作業が発生する．そのため，プロジェクトの前半だけ忙しい，あるいは終盤だけ仕事が発生するという人員はい

なくなり，負荷の偏りがなくなる．ステップごとの職種，たとえば，要求だけを考える人であっても，毎日出来上がってくる製品が期待どおりか確かめなくてはならず，毎日新しい要求について考える必要がある．また開発領域についても同様で，事前に大きな部品だけを作る，ということはなく，その V 字に必要な部分だけ作り，結合して確認するようになるため，プロジェクト期間中でロードの偏りが発生しない．つまり増員分だけこなせる規模が増やせるのである．ただし「要求する人＝要求を確認する人」の増員を怠ると，「作る係」をいくら増やしても開発規模を増やせる効果はない．要求の確認がボトルネックとなるからである．

システムはストーリーの集まりに分解され，ひとつのキューで管理される．チームは同時に数ストーリーを開発することができ，その様子はまるでマルチスレッドでひとつのキューのジョブを処理していくかのようである．これを拡張して，ひとつのキューに複数バージョンのストーリーを積むことで，自然とチームは複数バージョンに対応できるようになる．それぞれの状況に応じて投入するメンバーを調整できるうえ，バージョン間での問題の共有が行える．メンバーをバージョンに固定する方式では，バージョンのリーダーがメンバーを独占して使いたがるため，相互の協力が難しくなる．

私たちのチームはこの複数バージョンを開発できる方式で，複数の製品の並行開発を行っている．まったく別の製品であっても 1 つのチームで扱うことが可能なのである．

2.7.2. サブシステム別チーム

大きなシステムでは複数のサブシステムから構成され，それぞれにチームを割り当てて開発を行うことが多い．その場合，チーム間の開発ペースの違いにより衝突が起こりがちである．隣のチームのリリースが遅い，質問の返事がこないなどといった不満がたまる．これは，次のように考えるとよい．

1チームの開発でも同時に複数のストーリーを開発している．完成するタイミングはそれぞれ異なるが，そのタイミングをイテレーションの区切りなどに合わせていない．忍者式テストによるシステムテストによって，毎日完成を確認するからだ．なるべく長い期間テストできるように早めにコミットしたほうが喜ばれる．このようにチームの中でもペースの違いは発生し，それは許容されている．

この構造はチームとチームの間でも同様である．異なるペースを混ぜても，結合された状態で確認するという原則を守っていれば，1 チームの開発とそれほど変わらない．他のチームからの変更が届くのが，自分たちの予想よりも早かったり遅かったりするかもしれないが，自チームのメンバーの作業が遅延するのと同様である．

イテレーションの切れ目にみんなの完了がきれいに揃うことは重要でないので，隣のチームのペースを無理に変える必要はない．タイミングを合わせると効率がよい場面では自分たちが合わせればよい．

となりのチームは遅いのだろうか．「遅い」のではなく，実はタイミングが合わないことへの不満なのではないか．となりのチームもサボってないのでイライラしない．となりのチームの人たちも同じチームの人たちのように考えよう．相手がなぜその時に出せないのか，相手の都合を聞いてみよう．

さて，あなたのチームはどこまでか．同じチームの人のようにとなりのチームに気をかけていると，果たしてどこから違うチームなのかわからなくなる．チームの境界が曖昧になることは悪いことではない．自チームだけの成功は製品としては失敗なのである．

2.7.3. 工程別／ロール別のチーム

工程別／ロール別にチームを作るケースもあるだろう．たとえば，設計するチームと実装するチーム，開発するチームとテストチームといった具合である．その場合，お互いの領域に踏み込めない雰囲気になりがちである．前後の工程のチームに対して遠慮したり，不信感を持ったりしてしまう．自分たちはうまくできているが前後の工程に問題がある，といった対立構造が生まれやすい．

自分の工程の良し悪しは製品で確認するとよい．どの工程を担当していても，良い製品ができたかが根拠となる．前後の工程のチームがうまくいくように気をつかうべきである．あなたの工程だけが成功しているという状況はありえないのである．

3. 結果・考察

忍者式テストを始めて最初に起きたことは，難しそうな問題も躊躇せずに修正できるようになったことである．不具合や性能などの実装レベルの問題に積極的に対応できるようになった．次に起きたのは，直せる幅が広がったことだ．理想の製品をイメージして，それとの差分も積極的に探しだす者が現れた．数年後には全員がずっと「良い製品とは何か」を問いながら開発できるようになった．こ

れが忍者式テストの一番大きな効果である.

忍者式テストの副次的な効果として学習機会の創出がある. 毎日のテスト時間, 朝会のチケットの読み合わせなどさまざまな場面で, 製品の仕様, 設計, 実装など, 製品に関する質疑応答が絶えず行われる. これは圧倒的な量のドリル学習に相当する. これにより, ひとりひとりの脳内にシステムのあらゆることが配線され, 製品を動かしたときに瞬時におかしさに気づくようなことが起きる. 誰かのちょっとした疑問が別の誰かの思考を刺激する. チームはひとつの巨大な脳を持った生き物のようになる.

4. おわりに

事前に分かることと試してみて分かることがある. 私たちはこの事実に真摯に向き合っている. あらゆる場面で「うまくいったらどうなるの？」という問いかけからはじめる. うまくいった状態がイメージできると, 期待通りでないことを認識できる. 認識できれば修正できる. これが私たちのあらゆる行動の元となるカタチだ. 問題が見つかると皆で喜び, 見つけた人は感謝される. 早く失敗して良い製品に仕上げていくことに全員の目が向いている.

本稿では, 反復開発の利点とプラクティス「忍者式テスト」を説明した. 忍者式テストは XP を拡張するプラクティスである. XP はその名前から受ける印象とは裏腹にビジネスとテクノロジーの調和を述べているが, 忍者式テストも同様にテストを起点とした開発全体の有り様を語っているのである.

テストからはじめよ.

参考文献

[1] Kent Beck, Cynthia Andres, 角 征典[訳]. エクストリームプログラミング: オーム社, 2015

[2] Kent Beck, 和田 卓人[訳]. テスト駆動開発: オーム社, 2017

[3] Dr. Winston W. Royce: Managing the Development of Large Software Systems: 1970

[4] 独立行政法人 情報処理推進機構. 2012 年度「ソフトウェア産業の実態把握に関する調査」: 2013

[5] Glenford J.Myers, Tom Badgett, Todd M.Thomas, Corey Sandler, 長尾 真[監訳], 松尾 正信[訳]. ソフトウェア・テストの技法 第 2 版: 近代科学社, 2006

[6] Tom DeMarco, Timothy Lister, 伊豆原 弓[訳]. 熊とワルツを - リスクを愉しむプロジェクト管理: 日経 BP 社, 2003

[7] 落水浩一郎, "WATERFALL Model 再考 ソフトウェア開発管理とソフトウェア開発方法論の融合について", SEAMAIL. 2006, Vol.14, No.9-10. 岸田孝一編. ソフトウェア技術者協会.

[8] 中島滋, “Ninja Testing at Toteka03”, slideshare, 2014-10-04, https://www.slideshare.net/ledsun1/03-39903732.

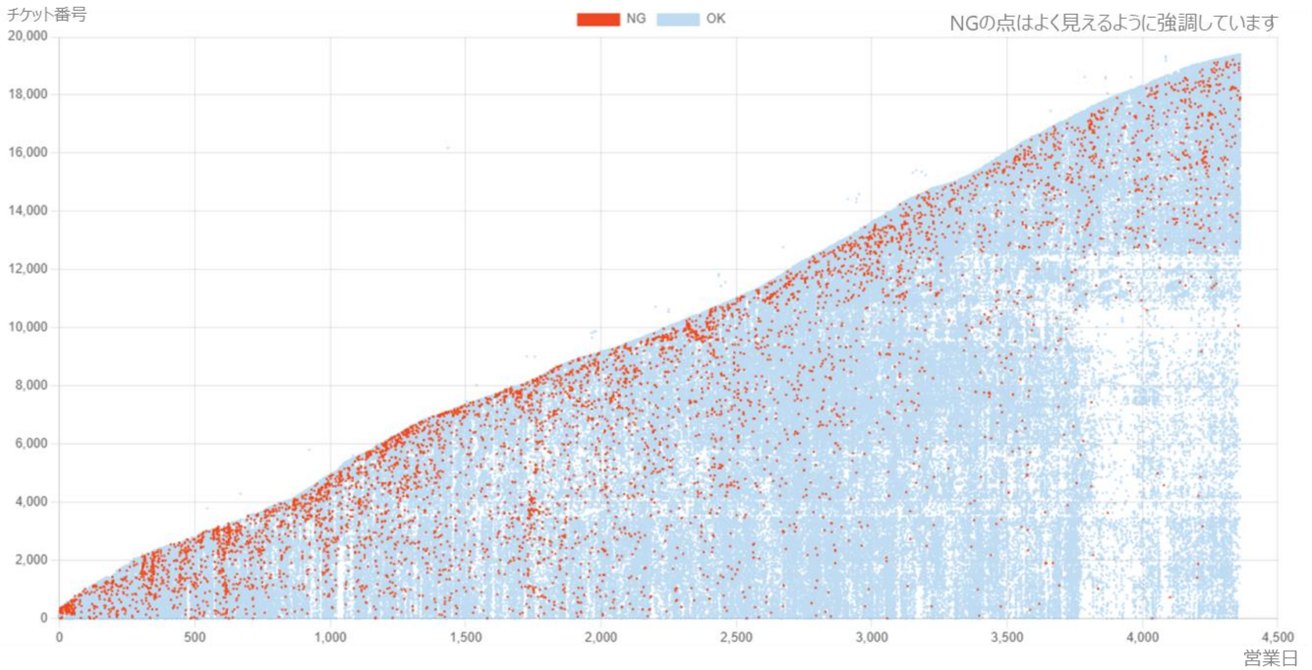

Fig. 9 テスト実施の記録（20 年分）

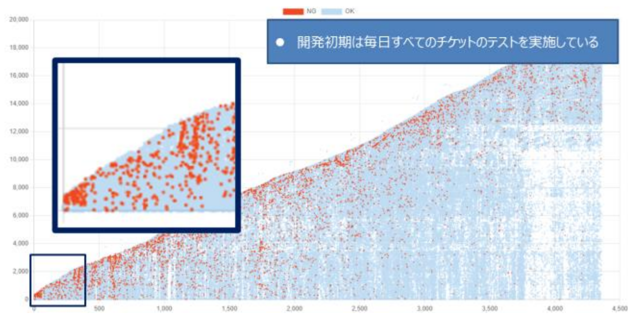

Fig. 10

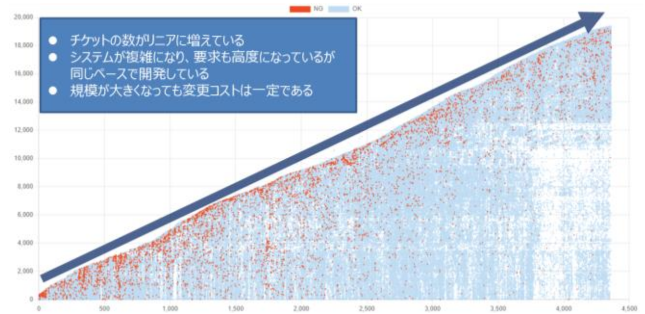

Fig. 11

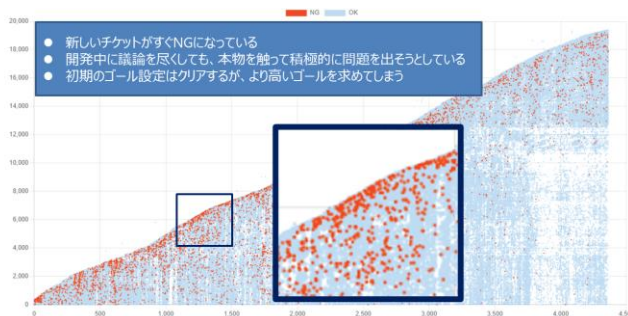

Fig. 12

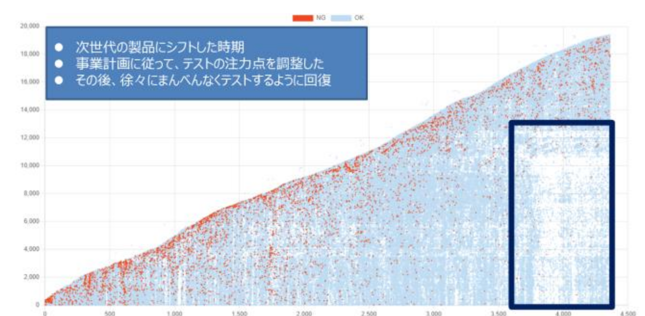

Fig. 13