アジャイルと反復開発

- 忍者式テスト 20 年の実践から -

Agile and iterative development

- Two-decade-long ninja testing -

キヤノンメディカルシステムズ株式会社
CANON MEDICAL SYSTEMS CORPORATION
○深谷 美和　　関 将俊 1)
○Miwa Fukaya　　Masatoshi Seki1)

Abstract Agile development has become widely spread all over the world. Iterative development, which is basis of it, and test-driven development have become also well known. On the other hand, it seems that there are cases where product development does not go well, although practices of agile development are going well. Hearing the situation of these kind of teams, we felt that practice of underlying iterative development had some problems. Our team has been developing x-ray computed tomography system with extreme programming. This paper describes “Ninja Testing” that is effective practice in iterative development, informs tips to make iterative development well from two-decade-long rich experience of agile development.

1. はじめに

アジャイル開発は世界中で広く普及し，その基礎となる反復開発，テスト駆動開発[1]（以降，TDD）もよく知られることとなった．その一方で，アジャイル開発の個々のプラクティスはうまくできているのに，製品開発がうまくいかないといったケースもあるようだ．そのようなチームの様子を聞いてみると，基礎となる反復開発の実践に問題があるように感じた．

私たちのチームは X 線 CT 装置をエクストリームプログラミング[2](以降，XP)で開発している．本稿では反復開発に有効なプラクティス「忍者式テスト」を紹介し，20 年以上にわたるアジャイル開発の豊富な経験から，反復開発をうまくやるためのヒントを伝える．

2. 大規模ソフトウェア開発の難しさ

ソフトウェア開発は,不具合の原因工程と不具合の発見工程（図 1）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く,テストも活用してソフトウェアを完成させる．

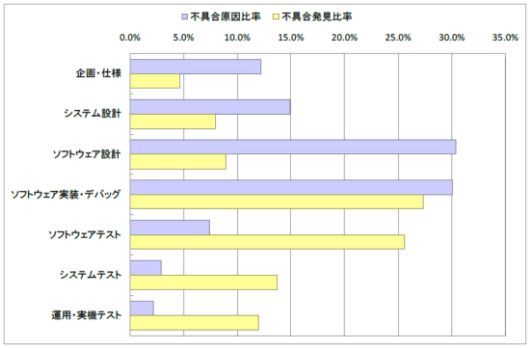

図 1 不具合の原因工程と不具合の発見工程[3]

キヤノンメディカルシステムズ株式会社
CANON MEDICAL SYSTEMS CORPORATION

栃木県大田原市下石上 1385 番地　Tel: 0287-26-6481　e-mail:miwa.fukaya@medical.canon
1385, Shimoishigami, Otawara, Tochigi Japan
1) キヤノンメディカルシステムズ株式会社
CANON MEDICAL SYSTEMS CORPORATION

【キーワード：】Software Development, Agile, XP, Software Test, TDD

最初の工程から確認までの期間が短い場合は問題が起きにくいが（図 2），大規模となり確認できるまで時間がかかるようになると（図 3），プロジェクトの終盤で次のような問題が発生する．ひとつは，不具合が見つかっても対策が間に合わない．もうひとつは，より良い仕様や設計を発見しても諦めざるを得ない．

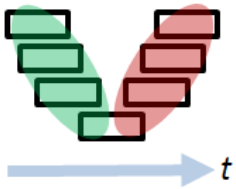

図 2 小規模開発

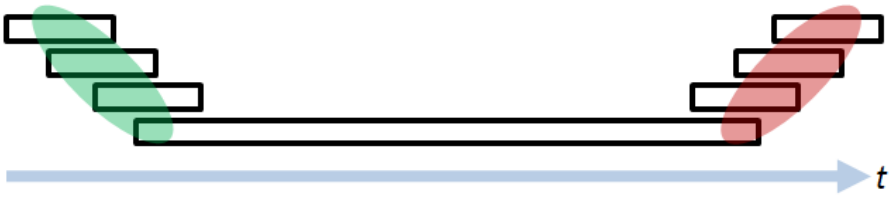

図 3 大規模開発

反復開発はこれらの問題を解決できる手法のひとつである．大規模の開発を「たくさんの小規模の開発の連続」と考えることで，プロジェクトの早い時期からテストを開始し，早く問題を見つけ修正することができる．また，プロジェクトの終盤であっても仕様や設計の調整が可能で，最後の最後まで製品を磨くことができる．（図 4）

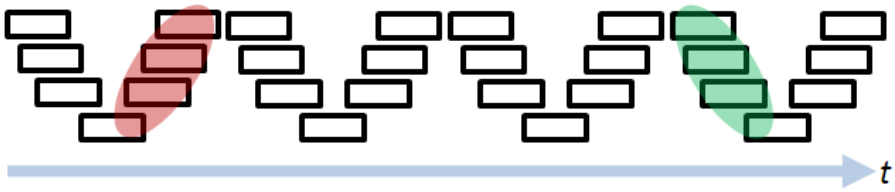

図 4 反復開発

反復開発の利点をいくつか述べる．

- 頻繁にテストするため問題がすぐに発見できる．修正も容易である．
- 発見した問題や違和感，知見を次の反復にフィードバックできる．ソフトウェアの設計，実装に関するものだけでなく，開発のやり方，製品のドメイン知識なども含まれる．
- 初期に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．リスクの高い機能から開発するといった作戦が使える．
- 一度に開発する機能を小さくし認知的負荷を軽減することで，機能の細部にまで目が届くようになる[4]．
- 機能そのものだけでなく機能とシステムの関わりについても詳細に議論できる．

反復開発でも一般的なバージョンアップ開発と同様に，システム全体をカバーするテストが非常に重要であり，これを反復ごとに行う必要がある．反復を重ね，システムの規模が大きくなるにつれ，テストケースの数も増えていく．本稿で述べる「忍者式テスト」は，この現実に正面から向き合うプラクティスである．

3. 忍者式テスト[5]

忍者式テストは名前に「テスト」が含まれているが，テストだけでなく開発全体の活動である．反復開発では大きな要求を小さな単位にわけて開発する．この単位をストーリーと呼び，チケットで管理する．ストーリーは数日で完成する大きさで，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，結合したシステムで確認できる必要がある．ストーリーはテストからはじめる．たとえば，どう試せばユーザーが困っていることが解決できたと分かるのか，どう試せば意図したとおりに作れたと言えるのか，を考える．そしてこれを問い続ける．

私たちはストーリーのチケットの中に確認方法となるテストケースを書く．開発とテストは分

かちがたい関係であるからストーリーとテストケースを同じ場所に書くのは自然なことである．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う．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，今日までに累積し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．前述したように反復開発ではシステム全体をカバーするテストを反復ごとに行う必要がある．したがって，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．このテストにパスすることが，今日のシステムがうまく動いている根拠になる．

なお，この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．

忍者式テストはテスト駆動開発を受け入れテストのレベルで行うプラクティスである．TDD ではまずテストコードから書くが，忍者式テストは受け入れテストから考えストーリーを実現する．

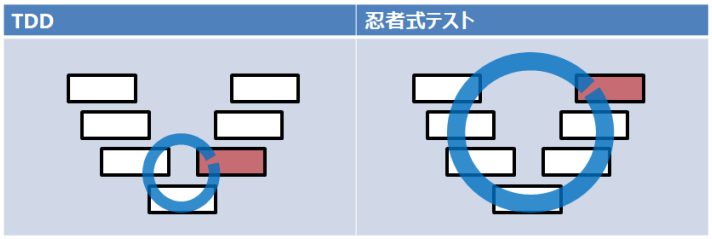

図 5 忍者式テストが対象とする範囲

4. 反復開発をうまくやるには

4.1 ストーリーを考えるときのヒント

アジャイル開発の個々のプラクティスはうまくできているのに，製品開発がうまくいかないといったケースがあるようだ．開発の様子を聞いてみると「じわじわ開発」になっていることが多い．プログラマーが V 字の底周辺（ソフトウェア設計，実装・デバッグ）ばかりを実施しているのが特徴で，複数のイテレーションを経て，テスト担当者，またはテストチームがまとめて受け入れテストを行う．（図 6）

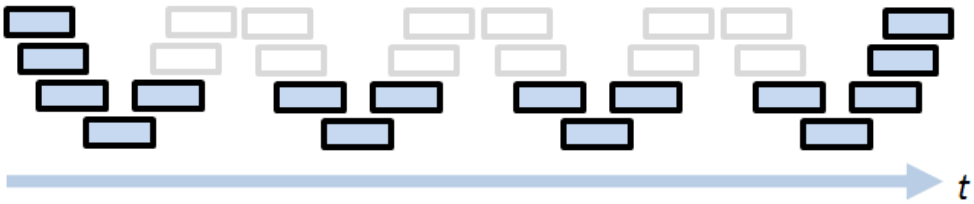

図 6 じわじわ開発

この状況はプログラミングに近い領域をイテレーションに区切っただけで，大きなひとつの V 字の開発と変わらない．反復していないので，前述した反復開発の利点を享受できない．ストーリーを入れ子にして複数のタスクで構成すると作業は進んでいるように見える．しかしタスクが終わってもストーリーとしては完結しておらず，各タスクの確からしさはわからないままだ．作業しているかどうかに興味があるチームだとタスクをこなすことに注力しがちである．再び，作業とロールに注目してみよう．プログラマーはプログラミング周辺ばかりを繰り返し，テスト担当者，またはテストチームがまとめて受け入れテストを行う．専門性を生かそうとした結果かもしれないが，システムと結合する難しさを最後まで避けているようにも感じる．本当にやらなければならないことが先延ばしになっている．

私たちのチームは今日の変更によってシステムがどのように変わるのか，よい製品となっているのかに興味がある．作業しているかどうかには興味がないので，結合するまで確からしさがわからないタスクの使用はやめた．その代わり，どんなに小さな変更であってもシステム全体で試せるようにストーリーを工夫する．ロールは二つある．すべての工程を行うプログラマーとプロ

グラミング以外のすべての工程を行うテスターだ．専ら特定の工程だけを行うロールではない．よって受け入れテストはプログラマーも毎日 1 時間実施する．

以降は「じわじわ開発」に陥ってしまい困っているチームへのアドバイスである．タスクだと思ったものをシステムで結合して試すにはどうしたらよいのか，という視点でタスク自体の再設計（単位，範囲など）をするとよい．ストーリーをどのように分割するのか具体的な例を用いて説明する．

ケーススタディ：レポートを表示する

既存のシステムにボタンを追加し，そのボタンを押すとレポート画面を表示するストーリーの例を見てみよう．（図 7）

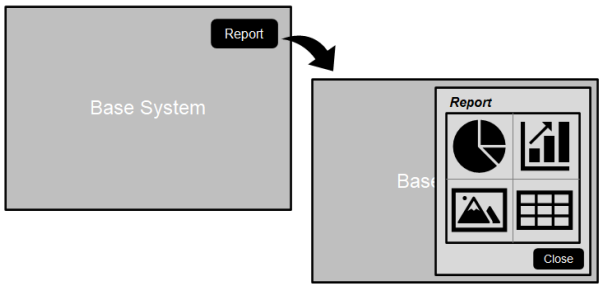

図 7 レポートを表示する

図 8 は，レポート機能を作ってからシステムに統合する作戦である．レポート機能自体に注目してストーリーを複数のタスクで構成し入れ子にする．部品を揃えて最後に既存システムと結合するスタイルであるため，各タスクの確からしさは最後までわからない．

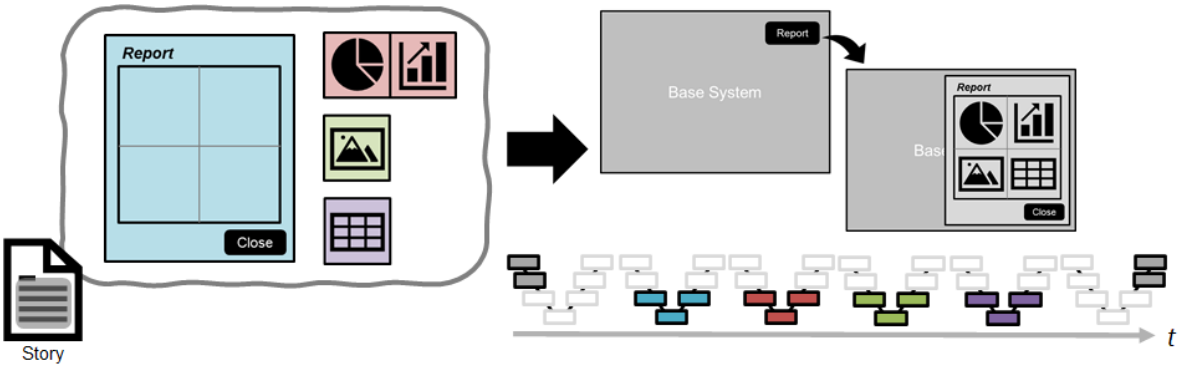

図 8 レポート機能を作ってからシステムに統合する作戦

私たちはストーリーを考えるとき「システムで試せる一番小さな変化はどこか」を探す．このケースでは「既存のシステムにボタンをつけて中身のない画面を表示する」となる．（図 9）

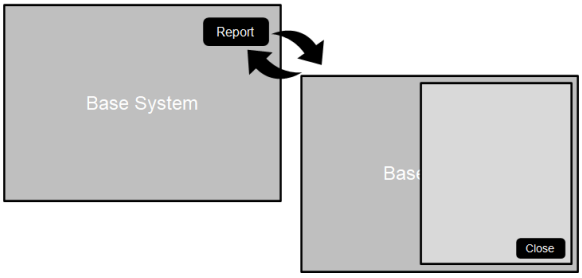

図 9 システムで試せる一番小さな変化

これほどの小さな変化に価値があるのかと不安になるかもしれない．私たちは，次のように考える．

- 中身のない画面を表示してもシステムが壊れないことは価値である
- 最初からシステムに統合した状態で開発できるのは価値である
- 懸念点を実際にシステムで確かめられたことは価値である

✓ システム側が機能を拡張できるようになっているか	✓ 画面(ウィンドウ／ダイアログ)が動く／動かない
✓ どんなときに起動できるのか	✓ スクリーンセーバーなどが動いたときの挙動
✓ 終了したらどんな画面になっているべきか	✓ キーボードショートカットの割り当て
✓ 起動に関わる設定ファイル(システム全体、機能別)	✓ 他の機能との並行動作
✓ 起動方法をどうするのか(メニュー、ボタン)	✓ 機能の起動中にできることはなにか
✓ 二重起動を許す(制御方法)／許さない	✓ 起動中にシステムを終了したらどうなるのか
✓ ウィンドウのZオーダー、モーダル／モードレス	✓ システムを再起動したときどうなってほしいか

図 10 懸念点の例

ユーザーに見えているものだけで考えると開発の実体に合ってないことがある．ストーリーを記述するときユーザーの言葉をそのまま使わず，既存のシステムや実装技術や製品のドメインの視点で解釈しなおすとよい．最初のストーリーで中身のない画面ができたので，次はレポートの要素ごとのストーリーを 1 つずつ実装する．1 つ実装するたびにシステムと結合した状態で確認する．結合は混乱を伴うが，少しずつ結合することで混乱を小さくできる．図 11 に私たちのストーリーの作り方を示す．

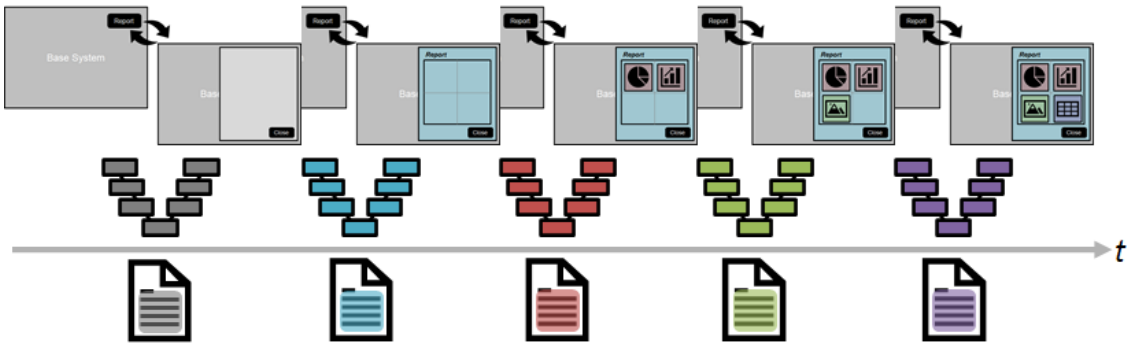

図 11 私たちのストーリーの作り方

私たちは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ストーリーを完成させる．どんなに小さくても，そのストーリーのテーマの中での完璧を目指し，かつ，システムが破綻しないように作り続けることは，いつでも出荷可能な状態を保つことにつながる．

ストーリー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．

4.2 バグの修正を後回しにしない

テストとは，エラーをみつけるつもりでプログラムを実行する過程である[6]．もちろんエラーを見つけたら終わりではない．私たちはバグを見つけて直すためにテストする．バグ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のだ．

バグを見つけたらどうするか．私たちは原因が分かるまでストーリーの開発を止める．このとき見かけの現象だけで深刻さを判断しない．バグは問題の一部分が運良く見えただけに過ぎず，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．

バグの修正を最優先にしてもストーリーの出来上がりは遅くならない．後述するが，私たちのチームが 20 年間同じペースで開発できていることからも分かる．

バグの修正を後回しにすると，結果的に開発の進みが遅くなる．バグが混入した状態で開発するので，プログラマーは問題を避けて開発しなければならず，プログラムコードが複雑になりミスをまねきやすくなる．テスト担当者も有効なテストができない．バグが修正されたときに本来やるべきだったテストを実施するのも難しくなる．そしてバグを修正するまで関連する問題が報告され続けるだろう．

チームで扱える心配の量は一定であり，いまある問題を解決しなければ，そこを解決した先にある高度な問題を見つけることはできない．あなたのチームの問題が増えたのはストーリーの実現を急ぎ過ぎているせいかもしれない．

4.3 ペースの違いからくる不満をどう考えるか

大きなシステムを複数のサブシステムで構成し，それぞれにチームを割り当てて開発することがある．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．となり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，次のように考えている．

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．イテレーションがきれいに揃うことよりも，早く確認できることが大切だと考えているからだ．ペースを合わせるために待ち合わせをするより，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，結合し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，となり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．

4.4 規模と向き合う

私たちのチームはより早く現物で確認して問題を放置しないのを是としている．ゴールを先に考え，これを実現できるように全員が行動している．そのために結合を先延ばしにしがちなタスクの使用を禁止した．バグが見つかったら原因が分かるまでストーリーを止めた．チームの規模が大きくなった際には，イテレーションをきれいに揃えるよりも早く確認できることを優先した．さまざまな場面において「より早く現物で確認するにはどうしたらよいか？」を軸に，完結した反復を考えることがうまくやるコツだ．大きくて複雑な問題を小さくて完結した大量の問題に変換するのである．

次に，量が増え続けても一日で処理したいというジレンマを解決した事例を紹介する．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．直近の変更は早く確かめたいが，全てのテストケースを確かめたいとも思っていた．

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する．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，注目している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し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．

図 12 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アルゴリズムの効果をいくつか述べる．

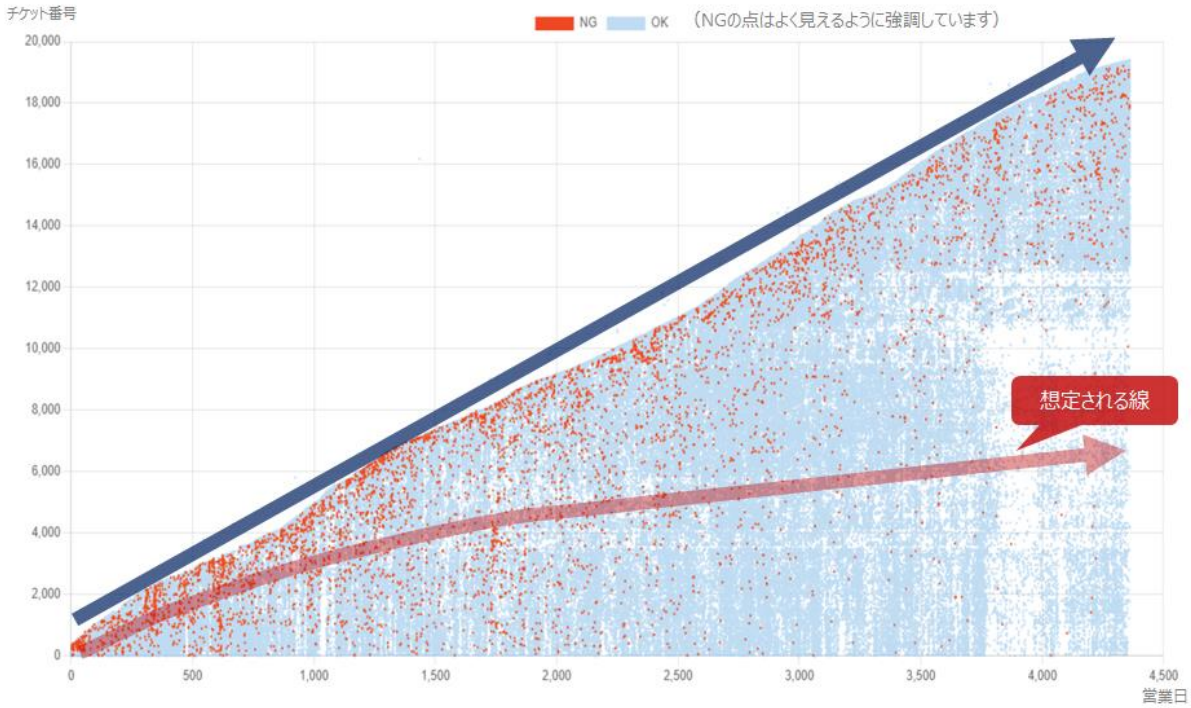

図 12 テスト実施の記録（20 年分）

(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．

(2) 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．本物を目の前にすると，初期のゴール設定より高いゴールを求めてしまうことが多い．

(3) 3,800 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．

5. 考察

図 12 を用いてチームの開発能力について考察する．営業日と完成したチケットの数がリニアに増えている点に注目してほしい．

一般的にシステム開発は複雑・大規模になるにつれ修正が難しくなり，競争の激しい製品では日々要求も高度になる．そのため，開発のペースは赤色のカーブのように徐々に鈍化すると想定される．しかし実績では 20 年間同じペースで開発が進んでいる．開発の難しさに釣り合うように，チームの開発能力が向上していると考えられる．XP の仮説のとおり，規模が大きくなっても変更コストを一定にできることも示している．

なぜこのようなチームになれたのか，20 年間をふりかえってみよう．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だ．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．これは毎日の受け入れテストにより，もしミスをしても私たちは見つけられる，直せるという安心感からだ．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「良い製品とは何か」を問いながら開発するようになり，あとからチームに参加した開発者にも同様の変化が起きた．これが忍者式テストの一番大きな効果であり，この効果が開発能力の向上に寄与している．

6. まとめ

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，反復開発をうまくやるためのヒントを伝えた．結合して試すのを後回しにしないのが勘所である．後回しを誘う入れ子を避け，「システムで試せる一番小さな変化はどこか」を探す．すぐ結合して試せるよう小さなストーリーに解釈しなおす．バグの修正を最優先にしても開発のペースは遅くならない．十分に大規模で複雑になると完成のペースが揃わなくなるが無理に同期しなくてもよい．20 年以上にわたるアジャイル開発の経験をとおして，今もっとも伝えたいことである．

7. 参考文献

[1] Kent Beck, 和田卓人[訳], テスト駆動開発, オーム社, 2017
[2] 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
[3] 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」調査報告書, pp. 78, 2013, https://www.ipa.go.jp/digital/chousa/kumikomi/hjuojm000000lps0-att/000026799.pdf
[4] Felienne Hermans, 水野貴明[訳], 水野いずみ[監訳], プログラマー脳 ～優れたプログラマーになるための認知科学に基づくアプローチ, 秀和システム, 2023
[5] 深谷美和, 関将俊, 「テストからはじめよ」～忍者式テスト 20 年の実践から～, 2023, https://www.sea.jp/ss2023/download/15-ss2023.pdf
[6] 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