第3章 測れない生産性を求められる重圧

TODO ： 全体的に図解やダッシュボードの詳細部分はあとで追加予定

3-1 技術的負債の蓄積と長期的な開発速度の関係

バグ対応の忙しさの中での警鐘

リリースから1週間が経過した。湊のチームは相変わらず緊急バグ対応に追われていた。

　Slackの通知音が鳴り止まない。お客様からの報告が次々と舞い込む。ログインエラー、データ表示の不具合、画面のフリーズ。リリース直後の混乱は収まらず、むしろ新たな問題が次々と発見されている。

　山田テックリードが疲れた声で言った。

「また同じようなバグだな。根本的な問題を直さないと、この繰り返しが続くよ」

湊は画面を見つめながら答えた。

「わかってます。でも今は対応に追われて...」

「いつまでこれが続くんだろう？」

佐藤が小さく呟いた。チーム全体が疲労感に包まれている。リリース後1週間、毎日のようにバグ対応に追われ、新規開発は完全に停止している。

山田がコーヒーカップを置き、湊の方を向いた。

「湊さん、今のやり方、持続可能だと思う？」

チームの空気が一瞬凍りついた。誰もが感じていた違和感を、山田が言葉にした。

「技術的負債が蓄積している。それを無視して新機能を追加し続けると、開発速度はどんどん低下する。過去のデータを見ると、技術的負債の蓄積で開発速度が年間20-40%低下している」

湊は返答に困った。確かに最近、同じようなバグが繰り返し発生している。修正しても別の場所で同じ問題が起きる。モグラ叩きのような状態だ。

「でも、どうすれば...」

「まずは現状を把握することからだね。技術的負債がどれだけ蓄積しているか、それが開発速度にどう影響しているかを可視化する必要がある」

山田の言葉は的確だった。でも、それをどう可視化すればいいのか。湊にはまだ答えが見えていなかった。

技術的負債の現実と向き合う

ランチタイム。湊のチーム5名が社内カフェテリアに集まっていた。普段は別々に食べることが多いが、今日は自然と同じテーブルに座った。

山田が箸を置き、真剣な表情で話し始めた。

「このコードベース、3年前と比べて開発速度が半分以下になってる」

「半分以下...ですか？」

佐藤が驚いた様子で聞き返した。

「うん。同じ規模の機能開発に、以前は3日かかっていたのが、今は1週間かかる。コードが複雑になりすぎて、どこを触っても影響範囲が広がるからね」

山田はノートPCを開き、過去のプロジェクトデータを見せた。グラフには明確な下降トレンドが示されている。

「新機能追加に3日→今は1週間。バグ修正のたびに別の場所が壊れる。テスト追加が困難な構造になっている」

他のメンバーも頷いた。誰もが同じ問題を感じていた。

「勤怠管理モジュールと経費精算モジュールの連携が複雑すぎる。一箇所を修正すると、予想外の場所でエラーが出る」

「プロジェクト管理モジュールも同じ。データベースの構造が複雑で、新しい機能を追加するのが大変」

湊はメンバーの話を聞きながら、改めて問題の深刻さを実感していた。

「でも、これって生産性指標に現れるんですか？」

湊の質問に山田は首を横に振った。

「現れないんだよ。だから問題なんだ。コード行数は増えているし、コミット数も増えている。でも、実際の開発速度は低下している。見た目の指標と実際の生産性が乖離している」

「見えない生産性の低下...」

湊は呟いた。技術的負債の影響は、従来の生産性指標では測れない。でも、確実に開発速度を低下させている。

「これをどう説明すればいいんだろう」

湊の疑問は、チーム全体の疑問でもあった。

解説：技術的負債が開発速度に与える影響

湊のチームが直面した問題は、多くの開発チームが経験する典型的な課題です。

なぜ技術的負債が開発速度を低下させるのか

技術的負債が蓄積すると、以下の問題が発生します：

- **コードの複雑度の増加**: 循環的複雑度が10を超えるコードが増えると、理解と修正に時間がかかる
- **結合度の高さ**: モジュール間の依存関係が密で、一箇所の変更が広範囲に影響する
- **テストカバレッジの不足**: リファクタリング時の安全性が確保できず、変更を躊躇する
- **ドキュメントの不足**: コードの意図が不明確で、理解に時間がかかる

これらが連鎖することで、開発速度が段階的に低下します。過去のデータでは、技術的負債の蓄積により開発速度が年間20-40%低下することが確認されています。

技術的負債の予兆検知

技術的負債の蓄積は、以下の兆候で早期に発見できます：

- **ビルド時間の延長**: 5分だったビルドが10分、20分と延びていく
- **バグ発生率の増加**: 新機能追加のたびにバグが増える
- **開発速度の低下**: 新機能追加に3日かかっていたのが1週間、2週間と延びる
- **コードレビュー時間の増加**: レビューに時間がかかり、承認までに数日かかる

これらの予兆を見逃すと、事業失敗につながる可能性が高いです。技術的負債の蓄積が市場投入遅れを引き起こし、競争優位を失うことになります。

技術的負債の可視化の重要性

技術的負債を可視化することで、以下の効果が得られます：

- **問題の早期発見**: 予兆を検知し、早期に対応できる
- **投資判断の材料**: 技術的負債の返済にどれだけの時間とコストがかかるかを把握できる
- **共通理解の形成**: チーム全体で問題を共有し、改善への意識を高められる

詳細な改善手法については、章の後半で具体的に見ていきます。

3-2 「今は動くから」の先にある持続不可能な開発環境

データ収集への挑戦

その夜、湊は一人オフィスに残っていた。他のメンバーは帰宅し、静まり返ったフロアで湊は過去のプロジェクトデータを整理していた。

Git履歴、JIRAチケット、バグ報告数。過去3年分のデータを集計し、スプレッドシートに整理していく。

「開発速度の推移...バグ発生率の推移...」

湊はデータを見つめながら、パターンを探していた。技術的負債の影響を数値で示せないか。それができれば、チーム全体で問題を共有できるかもしれない。

数時間後、湊はいくつかの発見をしていた。

「開発速度：3年前と比べて50%低下。バグ発生率：月5件から月15件に増加。技術的負債の蓄積による年間20-40%の開発速度低下を確認」

スプレッドシートには明確な相関関係が示されていた。コードの複雑度が増すにつれて、開発速度は低下し、バグ発生率は増加している。

「これが技術的負債の影響なのか...」

湊はグラフを見つめながら考えた。でも、これだけでは不十分だ。どう伝えれば、みんなが理解してくれるのか。どう説明すれば、技術的負債の返済に時間を割いてもらえるのか。

孤独な作業と挫折感

数日間、湊は孤独な分析作業を続けていた。データは集まるが、「どう伝えれば理解してもらえるのか」という疑問が消えない。

作成した資料を見直しては修正の繰り返し。グラフを追加し、説明を書き直し、また見直す。でも、どこか物足りない。

「これって本当に意味があるのかな？」

湊は自己不信に陥っていた。一人で作業を続けても、誰も見てくれないかもしれない。でも、「何もしないより、やってみる価値はある」という決意だけは揺らがなかった。

リビングのテーブルには空き缶が2本だけ。先日の失敗直後と比べると、少し落ち着いてきた証拠だ。でも、窓際のパキラはまだ元気がない。水をやったばかりだが、葉が少し垂れている。

湊はノートPCを閉じ、深く息を吸った。

「明日、もう一度山田さんに相談してみよう」

「今は動くから」の先にある問題

翌日のランチタイム。湊は山田に作成した資料を見せた。

「山田さん、これを見てもらえますか？」

山田は画面を見つめながら、時々頷いた。

「おお、これは面白い。開発速度とバグ発生率の相関関係が明確に示されているね」

「でも、これだけでは不十分な気がして...」

湊の不安そうな声に、山田は少し考えてから答えた。

「確かに、データだけでは伝わらないかもしれない。でも、これは重要な第一歩だ。『今は動くから』という言葉の先に、どんな問題が待っているかを示している」

「『今は動くから』...」

「うん。多くのチームが『今は動くから、後でリファクタリングする』と言って先送りにする。でも、その『後で』は永遠に来ない。そして、気づいた時には手がつけられない状態になっている」

山田の言葉は、湊のチームの現状を的確に表現していた。

「このデータが示しているのは、『今は動くから』という判断の先にある、持続不可能な開発環境だ。開発速度が50%低下し、バグ発生率が3倍になる。これが続けば、いずれ開発が止まってしまう」

湊は深く頷いた。確かに、このままでは持続不可能だ。

「でも、どうすれば...」

「まずは、この問題をチーム全体で共有することだ。一人で抱え込まず、みんなで考えよう」

山田の言葉に、湊は少し希望を感じた。

解説：「今は動くから」の先にある持続不可能性

湊がデータ分析で発見した問題は、多くのチームが直面する典型的な課題です。

なぜ「今は動くから」が問題になるのか

「今は動くから、後でリファクタリングする」という判断は、短期的には合理的に見えます。しかし、以下の理由で問題が発生します：

- **「後で」は永遠に来ない**: 新機能開発が優先され、リファクタリングの時間が確保されない

- **負債の雪だるま式増大**:　技術的負債が蓄積し、将来的な開発速度が大幅に低下する
- **修正コストの増大**:　後から修正するほど、コストが指数関数的に増加する

　データでは、技術的負債の蓄積により開発速度が年間20-40%低下することが確認されています。また、バグ発生率も増加し、リリース後の障害対応に追われることになります。

持続不可能な開発環境の兆候

　以下の兆候が見られたら、持続不可能な開発環境に陥っている可能性があります：

- **開発速度の継続的な低下**:　同じ規模の機能開発に、以前の2倍以上の時間がかかる
- **バグ発生率の増加**: 新機能追加のたびにバグが増える
- **修正の連鎖**: 一箇所を修正すると、別の場所で問題が発生する
- **テスト追加の困難**: 新しいテストを追加することが困難になる

　これらの兆候を見逃すと、いずれ開発が止まってしまう可能性があります。

持続可能な開発への転換

　持続可能な開発を実現するには、以下の取り組みが必要です：

- **技術的負債の可視化**: 問題を数値で示し、チーム全体で共有する

- **リファクタリング時間の確保**:　開発時間の10-20%を技術的負債の返済に充てる
- **継続的な改善**:　「ボーイスカウトルール」のように、コードを触るたびに少しずつ改善する
-

　詳細な手法については、章末の「手法3：測れない価値の可視化ダッシュボード」を参照してください。

3-3 エンジニアとしての説明責任を果たす方法

山田との偶然の出会い

夜10時。湊が資料を修正していると、背後から声がかかった。

「まだいたの？何やってるの？」

　振り返ると、山田が立っていた。

「あ、山田さん。まだ残ってたんですか」

「うん、ちょっと用事があって。湊さんは？」

「実は、先日のデータ分析をさらに深掘りしていて...」

湊は恐る恐る作成した資料を見せた。山田は画面を見つめながら、時々「なるほど」と呟いた。

「これは面白い。開発速度とバグ発生率の相関関係が明確に示されているね」

「でも、これだけでは不十分な気がして...」

「いや、十分だよ。でも、もう少しコスト換算もできそうだね」

山田は資料を指差しながら続けた。

「例えば、開発速度が50%低下しているということは、同じ機能を開発するのに2倍の時間がかかる。つまり、開発コストが2倍になる。これを金額に換算すれば、経営層にも伝わりやすいかもしれない」

「コスト換算...ですか」

「うん。技術的負債の返済に時間をかけることは、長期的なコスト削減につながる。でも、それをどう説明すればいいかが難しいんだよね」

湊は深く頷いた。確かに、技術的な課題を財務的な言葉で説明できれば、理解してもらいやすいかもしれない。

山田からの後押し

山田はコーヒーを淹れながら言った。

「湊君、これをチームに共有してみたら？」

「でも、みんな忙しいし...」

「むしろ、みんなが感じてることを数字で示してるじゃない。これは価値のあることだよ」

山田の言葉に、湊は少し勇気をもらった。

「来週のチームミーティングで時間をもらおう。5分でいいから、この発見を共有してみて」

「でも、うまく説明できるかな...」

「君の作った資料で十分伝わるよ。データが明確だからね」

山田はコーヒーを一口飲み、続けた。

「エンジニアとして、説明責任を果たすことは大事だ。でも、一人で抱え込む必要はない。チーム全体で考えれば、きっと道は開ける」

湊は深く頷いた。一人で抱え込むのではなく、チーム全体で問題を共有する。それが第一歩なのかもしれない。

チームミーティングでの勇気ある提案

翌週水曜日のチームミーティング。湊は手に汗をかきながら資料を準備していた。

「すみません、5分だけ時間をください」

湊は緊張しながら、作成した資料を画面に映した。

「見えない生産性の可視化について、お話ししたいことがあります」

チームメンバーは最初、ざわめいていた。でも、湊がデータを説明し始めると、徐々に真剣な表情に変わっていった。

「開発速度が3年前と比べて50%低下しています。バグ発生率も月5件から月15件に増加しています。これは技術的負債の蓄積による影響だと考えられます」

グラフを見せながら、湊は続けた。

「技術的負債の返済に時間をかけることは、短期的には開発速度が低下するように見えます。でも、長期的には開発速度の低下を防ぎ、コスト削減につながります」

湊は資料を指差しながら続けた。

「開発速度が50%低下しているということは、開発コストが2倍になる。技術的負債の返済に時間をかけることで、長期的なコスト削減につながる」

チームメンバーは最初、ざわめいていた。でも、湊がデータを説明し始めると、徐々に真剣な表情に変わっていった。

佐藤が手を挙げた。

「湊さん、このデータって、どうやって集めたんですか？」

「Git履歴とJIRAチケット、バグ報告数を集計しました。過去3年分のデータを分析して、パターンを探しました」

山田が頷いた。

「データで見ると、確かに深刻だね。このままでは、さらに開発速度が低下する可能性が高い」

他のメンバーも頷いた。誰もが同じ問題を感じていた。

「でも、どうすればいいんですか？」

別のメンバーが質問した。

「まずは、技術的負債の返済時間を確保することだと思います。開発時間の10-20%をリファクタリングに充てる。それから、継続的な改善を心がける。コードを触るたびに、少しずつ綺麗にする」

湊の提案に、チームメンバーは真剣に聞き入っていた。

「これは良い提案だと思う。でも、PMや部長にどう説明すればいいんだろう」

山田の質問に、湊は少し考えてから答えた。

「コスト換算で説明するのはどうでしょうか。先ほど説明したように、開発速度が50%低下しているということは、開発コストが2倍になる。技術的負債の返済に時間をかけることで、長期的なコスト削減につながる」

「それなら、説得力があるかもしれないね」

山田が頷いた。

ミーティング後、複数のメンバーから個別に声をかけられた。

「もっと詳しく話を聞かせて」「一緒に改善策を考えよう」

湊は実感した。一人じゃできないことも、チームならできる。チーム全体で問題を共有し、改善に取り組む。それが持続可能な開発への第一歩なのかもしれない。

解説：エンジニアとしての説明責任を果たす方法

湊がチームミーティングで行った提案は、エンジニアとしての説明責任を果たす上で重要な取り組みです。

なぜ説明責任が重要なのか

　エンジニアとして、技術的な課題を説明することは重要な責任です。しかし、以下の理由で説明が困難になることがあります：

- **専門用語の壁**: 技術的な課題を非技術者に説明するのが困難
- **数値化の困難**: 技術的負債の影響を数値で示すのが難しい
- **時間的制約**: 説明のための時間が確保されない

　しかし、説明責任を果たさないと、以下の問題が発生します：

- **投資判断の困難**: 技術的負債の返済に時間を割いてもらえない
- **認識のずれ**: エンジニアと経営層の間で認識のずれが生じる
- **問題の悪化**:　技術的負債が蓄積し、将来的な開発速度が大幅に低下する

説明責任を果たすための方法

　説明責任を果たすには、以下の方法が有効です：

- **データによる可視化**: 技術的負債の影響を数値で示す
- **コスト換算**: 技術的な課題を財務的な言葉で説明する
- **段階的な説明**:　一度に全てを説明せず、段階的に理解を深めてもら

う

- **チーム全体での共有**:　一人で抱え込まず、チーム全体で問題を共有する

湊が行ったように、データを集計し、グラフで可視化することで、問題を明確に示すことができます。また、開発速度の低下をコスト換算することで、経営層にも理解してもらいやすくなります。

次のステップ

説明責任を果たした後は、以下のステップに進みます：

- **改善計画の策定**: 技術的負債の返済計画を立てる
- **時間の確保**: 開発時間の10-20%をリファクタリングに充てる
- **継続的な改善**:　「ボーイスカウトルール」のように、継続的に改善する

詳細な手法については、章末の「手法3：測れない価値の可視化ダッシュボード」を参照してください。

手法3：測れない価値の可視化ダッシュボード

湊が行ったように、技術的負債の影響を可視化することは、持続可能な開発を実現する上で重要な取り組みです。このセクションでは、具体的な手法を紹介します。

ダッシュボードの目的

目的: 技術的負債の影響を数値で示し、チーム全体で問題を共有する

効果:

- 問題の早期発見
- 投資判断の材料
- 共通理解の形成

測定すべき指標

開発速度に関する指標

- **機能開発時間**: 同じ規模の機能開発にどれだけ時間がかかるか
- **バグ修正時間**: バグ修正にどれだけ時間がかかるか
- **コードレビュー時間**: コードレビューにどれだけ時間がかかるか

品質に関する指標

- **バグ発生率**: 新機能追加のたびにどれだけバグが発生するか
- **テストカバレッジ**: テストがどれだけコードをカバーしているか
- **ビルド時間**: ビルドにどれだけ時間がかかるか

コードの複雑度に関する指標

- **循環的複雑度**: コードの複雑度がどれだけ高いか
- **結合度**: モジュール間の依存関係がどれだけ密か
- **コード重複率**: コードの重複がどれだけあるか

ダッシュボードの作成方法

ステップ1: データの収集

以下のデータを収集します：

- **Git履歴**: コミット数、変更行数、開発時間
- **JIRAチケット**: タスクの完了時間、バグ報告数
- **CI/CDパイプライン**: ビルド時間、テスト実行時間
- **コード品質ツール**: SonarQube、CodeClimateなどのメトリクス

ステップ2: データの分析

収集したデータを分析し、以下のパターンを探します：

- **開発速度の推移**: 時間の経過とともに開発速度がどう変化しているか
- **バグ発生率の推移**: 時間の経過とともにバグ発生率がどう変化しているか
- **コードの複雑度の推移**: 時間の経過とともにコードの複雑度がどう変化しているか

ステップ3: 可視化

分析したデータをグラフで可視化します：

- **時系列グラフ**: 開発速度、バグ発生率、コードの複雑度の推移
- **相関関係グラフ**: 開発速度とバグ発生率の相関関係
- **コスト換算グラフ**: 開発速度の低下をコストに換算

ステップ4: ダッシュボードの共有

作成したダッシュボードをチーム全体で共有します：

- **週次レビュー**: 週に1回、ダッシュボードをレビューする
- **月次報告**: 月に1回、経営層に報告する
- **四半期レビュー**: 四半期ごとに、改善計画を立てる

実践のステップ

ステップ1: データ収集の開始

- Git履歴、JIRAチケット、バグ報告数を集計
- 過去3年分のデータを分析
- パターンを探す

ステップ2: 可視化の試行

- スプレッドシートでグラフを作成
- 開発速度とバグ発生率の相関関係を可視化

- コスト換算を試みる

ステップ3: チームでの共有

- チームミーティングで資料を共有
- フィードバックを収集
- 改善計画を立てる

ステップ4: 継続的な改善

- 週次でデータを更新
- 月次で経営層に報告
- 四半期ごとに改善計画を見直す

チェックリスト

実施前:

- [ ] データ収集の方法を決定した
- [ ] 測定すべき指標を決定した
- [ ] ダッシュボードの目的を明確にした

実施中:

- [ ] データを収集し、分析した
- [ ] グラフで可視化した
- [ ] チーム全体で共有した

実施後:

- [ ] 週次でデータを更新している
- [ ] 月次で経営層に報告している
- [ ] 四半期ごとに改善計画を見直している

第3章のまとめ

第3章では、湊が技術的負債の影響を可視化し、チーム全体で問題を共有する過程を描きました。

学んだポイント

1. **技術的負債の可視化**: 問題を数値で示すことで、チーム全体で問題を共有できる
2. **持続不可能な開発環境の兆候**: 開発速度の低下、バグ発生率の増加などの兆候を早期に発見する重要性
3. **エンジニアとしての説明責任**: 技術的な課題を説明し、投資判断の材料を提供する責任
4. **チーム全体での問題共有**: 一人で抱え込まず、チーム全体で問題を共有することの重要性

次章への展望

湊は技術的負債の影響を可視化し、チーム全体で問題を共有することに成功しました。第4章では、湊がチームを巻き込んで、小手先の指標に依存せずに本質的な問題を特定していく過程を描きます。

一人で抱え込むのではなく、チーム全体で問題を共有し、改善に取り組む。それが持続可能な開発への第一歩となることを学びました。