 第3章 測れない生産性を求められる重圧

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る。

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み、ログインエラー、データ表示の不具合、画面のフリーズと、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。

山田が、画面から目を離して疲れた声で口を開いた。

「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る。

「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、湊の方を向く。

「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おり、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。

疲弊するチーム

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。窓際のテーブルに五人分のトレイが並ぶ。

技術的負債の現実と向き合う

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、しばらく黙ってから真剣な表情で話し始める。

「開発速度、3年前と比べて半分以下に落ちてるんだ」

「半分以下...ですか？」

佐藤が驚いた様子で聞き返す。

「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。スピードを優先してテストを書かないことが、影響範囲を見えなくしている。テストがないと、変更がどこまで影響するかわからない。だから、一箇所を修正すると、予想外の場所でエラーが出る」

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。

「先日開発した機能、過去に似たような機能を開発したときは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」

他のメンバーも頷く。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」

山田は少し間を置き、最後にこう口を開く。

「そうだね。ドメイン駆動設計を採用しているんだけど、根本的にはイベントストーミングでのモデリングができていない。ドメインモデルが曖昧なまま開発が進んで、積み重なった技術的負債と膨れ上がった設計の乱雑さが問題になっている」

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り、ため息をついてから答える。

「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」

単純な数値だけでは実態が歪むこと、指標を評価に直結させるとかえって行動がゆがむ——そんな話が、以前読んだどこかにあった。湊はそれを思い出していた。

「見えない生産性の低下...」

湊は呟く。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。開発側と事業側で測っているものが違う。だから数字を出しても「良くなった」と伝わらない。問題は遅れの理由や次の見通しを説明できないことだ。能力不足ではなく、説明に使える情報が足りていないのだと、湊は感じていた。

解説：技術的負債が開発速度に与える影響

この節では、技術的負債が開発速度に与える影響、開発組織と事業側の視点の違い、そして可視化の重要性を整理します。

ストーリーで描かれる「重圧」

なぜ「測れない生産性を求められる」が重圧になるのか。**何が起きているかが見えないうちは**、その問いが構造的な課題としてのしかかります。

- **バグ対応に追われ新規開発が止まっている** 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。山田の「今のやり方、持続可能だと思う？」という問いに、誰もが感じていた違和感が言語化される

- **技術的負債の現実** ストーリーで山田が語ったように、開発速度が年間20-40%低下し、同じ規模の機能に以前は3日・今は1週間かかる。バグ修正のたびに別の場所が壊れ、テスト追加が困難な構造になっている
- **見た目の指標と実際の生産性の乖離**　コード行数は増えているしコミット数も増えている。しかし実際の開発速度は低下している。従来の生産性指標では測れないが、確実に開発速度を低下させている
- **説明できないもどかしさ**　開発組織が見ているもの（リードタイム、デプロイ頻度など）と事業責任者・PdMが見ているもの（計画信頼性・予測可能性）は異なる。「なぜ遅れたのか」を説明できないことが問題だと気づき始めている

この重圧の背景には、**技術的負債が開発速度を確実に低下させているのに、従来の生産性指標では測れず説明できない構造**があります。何が起きているかを可視化する必要性に直面しているのです。

開発組織と事業責任者・PdMの視点の違い

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。

開発組織が見ているものと、事業責任者・PdMが見ているものは異なります。

開発組織が見ているもの

- リードタイム、デプロイ頻度、変更失敗率、MTTR
- これらは開発組織の内部状態・流動性・安定性を測る指標

事業責任者・PdMが見ているもの

- 計画信頼性・予測可能性
- 欲しいタイミングで欲しいものが出るか
- 計画は信じて立ててよいか

- 意思決定の前提として使えるか

Four Keysなどの指標は開発組織の内部安定性を測りますが、事業計画の信頼性は直接測りません。これが、Four Keysが改善しているにもかかわらず、事業責任者・PdMには「良くなった実感」がないギャップの原因です。

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
- **テスト不足による影響範囲の不明確化**　ストーリーで山田が話した通り、テストがないと影響範囲が予測できず、一箇所の修正で予想外の場所でエラーが発生する
- **ドメインモデルの曖昧さ**　ドメイン駆動設計を採用していても、イベントストーミングでのモデリングができていないと、ドメインモデルが曖昧なまま開発が進む。これにより「積み重なった技術的負債」と「膨れ上がった設計の乱雑さ」が発生し、開発速度が大幅に低下する

技術的負債の根本原因の一つとして、ドメインモデルの曖昧さがあります。ドメイン駆動設計を採用していても、イベントストーミングでのモデリングができていないと、以下の問題が発生します。

- **ビジネスロジックの散在**　ビジネスロジックがコード全体に散在し、どこに何があるかわからなくなる
- **設計の乱雑さ**　ドメインの境界が不明確で、モジュール間の依存関係が複雑になる

- **影響範囲の不明確化**　一つの変更がどこまで影響するか予測できず、予想外の場所でエラーが発生する
- **技術的負債の蓄積**　曖昧なモデルの上に機能を追加し続けることで、技術的負債が雪だるま式に増大する

イベントストーミングの価値

イベントストーミングは、ドメインエキスパート、エンジニア、プロダクトマネージャーなど、異なる職種が集まって、ビジネスイベントを中心にドメインモデルを可視化する手法です。これにより

- **共通理解の形成** 全員が同じドメインモデルを共有し、認識のずれを防げる
- **設計の明確化**　ドメインの境界とエンティティの関係が明確になり、設計の乱雑さを防げる
- **影響範囲の可視化**　イベントの流れを追うことで、変更の影響範囲を事前に把握できる

事業への貢献が実証されているシステムこそ、ドメイン駆動設計の価値が高い

すでにユーザーに価値を提供しているシステムは、長期的な持続可能性が重要です。ドメイン駆動設計とイベントストーミングを活用することで、積み重なった技術的負債と膨れ上がった設計の乱雑さを刷新し、将来の開発速度を大幅に向上させることができます。これは、新規システムよりも既存システムの方が、ドメイン駆動設計の投資対効果が高い理由です。

- **知識の属人化**　特定のメンバーにしか理解できないコードや設計が増えると、そのメンバーが不在の際に開発が停滞する。コードの意図が不明確で、設計判断の背景が共有されていないため、新しいメンバーがコードベースを理解するのに時間がかかる。メンバーの離脱リスクにより、技術的負債がさらに増大する
- **依存関係の管理不足**　古いライブラリやフレームワークのバージョンアップができていないと、セキュリティパッチが適用されず、リスクが蓄積する。将来的なメジャーアップデートの負荷が増大し、技術的負債として残る

- **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。GitClearの分析（「Coding on Copilot」2024 Data Report／「AI Copilot Code Quality」2025、1.5〜2億行規模のコードベース）では、変更量が増えると**Churn**（書いてすぐ消される行）が倍増し、リファクタリングの減少、コピー・クローンの急増（重複ブロックは2年前比で約10倍等）が報告されています。短期的なデリバリー速度は上がる一方、中長期的な技術負債を加速させる可能性が指摘されているため、コード行数やコミット数が増えても実際の開発速度や維持性は低下しうるという示唆があります。

生産性の測定に関する体系的レビュー（Petersen 2011）では、単純な成果／投入比（SLOC／工数等）は歪みを生むことが繰り返し指摘され、**メトリクスを評価に直結させるとゲーミングを誘発する**とされています。山田が言った「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」という乖離は、こうした研究と整合的です。DORAメトリクスはスピード（デプロイ頻度・変更リードタイム）と安定性（変更失敗率・復旧時間）の対でフローを捉える最小セットを提供し、SPACEフレームワークは開発者生産性をSatisfaction／Performance／Activity／Communication／Efficiencyの多次元で扱い、指標は緊張関係（trade-off）込みの束で扱うべきだとしています。単一指標に依存しない多面的な把握が、技術的負債の議論においても重要です。

第1章では、Martin Fowlerによるソフトウェアの「外部品質」と「内部品質」の分類を紹介しました。ここでは、その分類を前提に、内部品質とコストの関係を述べていきます。

高品質ソフトウェアは実際には安く作れる

「高品質なものは高価」というトレードオフがあるが、ソフトウェアの内部品質についてはこの法則が当てはまりません。Martin Fowler（"Is High Quality Software Worth the Cost?" 参照）によれば、**高品質なソフトウェアは実際には安く作れる**ことが示されています。

理由は以下の通りです。

1. **将来の変更コストの削減**　内部品質が高いコードは、将来の機能追加や修正が容易で、長期的なコストが低くなる
2. **開発速度の維持**　内部品質を維持することで、開発速度の低下を防ぎ、結果として開発コストが削減される
3. **バグ修正コストの削減**　テストカバレッジが高いコードは、バグの早期発見が可能で、リリース後の修正コスト（30-100倍）を避けられる

つまり内部品質への投資は「コスト」ではなく「将来のコスト削減への投資」として位置づけるべきです。

本質的な問題

本質的な問題は第1章で触れた通り、遅れを説明できないことと、説明可能な情報が欠けている状態です。

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
- **依存関係の古さ**　使用しているライブラリやフレームワークのバージョンが古く、セキュリティパッチが適用されていない。メジャーバージョンが2つ以上遅れている
- **知識の属人化**　特定のモジュールや機能について、特定のメンバーにしか質問できない状況が増える。コードレビューで「なぜこの設計にしたのか」という質問に答えられる人が限られる

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、後述のワークシート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

あの夜、見えない価値を可視化する方法について山田さんに相談しようと決めた。その意志を、数字で現状を把握する形で実行した。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、天井の照明だけがついた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。

バグ報告数は社内のバグ管理システムから、月ごとの発生件数を集計した。リリース日とバグ報告日の関係も確認し、リリース直後にどれだけのバグが発見されたかを時系列で追った。

スプレッドシートにデータを入力しながら、湊は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。

去年の同じ時期、ある機能の開発には平均して5日かかっていた。でも、今年は7日かかる。同じ規模の機能なのに、開発時間が1.4倍になっている。3年前と比べると、開発速度は50%低下していた。

バグ発生率も月5件から月15件に増加している。特に、リリース直後の1週間で発見されるバグの割合が、3年前は全体の20%だったのが、今は40%を超えている。

スプレッドシートにグラフを描くと、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。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できた。

データが語る現実

「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る。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。答えはまだ見えない。投資が増えても生産性の数字に表れない、個人の努力と組織の成果のあいだにはギャップがある——そんな論点が、以前目にした文章にあったことを思い出した。

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。毎晩、オフィスに一人残り、データを集計し、グラフを作成し、説明文を書いた。でも、「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を繰り返し、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。湊は資料を印刷して机に広げ、何度も読み返した。でも、経営層やPMに説明するには、まだ抽象的すぎる気がした。

「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。山田さんに相談した時は励まされたが、実際にチームに共有するとなると、また別の不安が湧いてくる。もし、みんなが「そんなの当たり前だよ」と一蹴したらどうしよう。もし、データの解釈が間違っていたらどうしよう。

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。今回は違う。データという武器がある。それをどう使うかは、自分次第だ。

自己不信の深まり

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、深く息を吸う。

「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。社内カフェテリアの騒がしさのなか、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、時々頷く。

「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、山田は箸を置いて少し考えてから答える。

「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なのだと、改めて実感する。

「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

ストーリーで描かれる「重圧」

データはあるが伝え方がわからない。そんな状態のとき、「投資判断につなげたい」という重圧はとりわけ重くのしかかります。湊が経験したのも同じ構造からです。

- **データが示す深刻さ** 過去3年分のデータで、開発速度の50%低下とバグ発生率の増加（月5件から月15件）の相関を発見した。技術的負債の影響を数値で示すところまでは届いている
- **伝え方の壁**　「どう伝えれば理解してもらえるのか」「技術的負債の返済に時間を割いてもらうための説明方法がわからない」という疑問が消えない
- **孤独な作業と挫折感**　一人で分析を続け、資料を作り直しては見直す繰り返し。誰も見てくれないかもしれないという不安と、それでもやり続ける決意のあいだで揺れている
- **「今は動くから」の先**　データが「持続可能な開発ではない」ことを示している。このままでは開発が止まってしまう可能性を肌で感じ始めている

この重圧の背景には、**問題をデータで示せても、ROIや説明の仕方がわからず投資判断につながらない構造**があります。持続不可能性を認識したうえで、どう転換するかを模索する段階にあるのです。

なぜ「今は動くから」が問題になるのか

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。

「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

- **テスト不足による影響範囲の不明確化**　ストーリーで山田が話した通り、テストがないと影響範囲が把握できず、予想外の場所でエラーが発生する。これが持続不可能性を加速させる
- **ドメインモデルの曖昧さ**　ドメイン駆動設計を採用していても、イベントストーミングでのモデリングができていないと、ドメインモデルが曖昧なまま開発が進む。これにより「積み重なった技術的負債」と「膨れ上がった設計の乱雑さ」が発生し、持続不可能性が加速する

技術的負債の蓄積と開発速度低下の関係は、本節3-1の解説で述べた通りです。コードの変更量や行数が増えてもリファクタリングが減り重複が増えると、中長期的には開発速度が落ちるという知見は、GitClearの大規模分析（参考文献参照）でも報告されています。「今は動くから」と先送りにすると、そうした負のスパイラルに陥りやすいのです。

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」（「来た時よりも美しく」を合言葉に、コードを触るたびに少しずつ改善する）を実践する

詳細な手法については、章末の測れない価値の可視化ダッシュボードのワークシート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

山田からの後押し

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。フロアには湊のデスクの明かりだけがついている。

「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。山田は湊の肩越しに画面を見つめながら、時々「なるほど」と呟く。

「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」

「データは揃ったんですが、経営層やPMにどう見せれば伝わるか、まだ自信がなくて…」

「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る。

「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」

湊は深く頷く。でも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。

開発生産性の本質は、部長や山田さんとの話のなかで何度も触れてきた、約束の信頼性だ。技術的負債の返済は、その信頼性を高める投資だと、湊は考えた。

「なぜ遅れたのか」を説明するためのデータ分析の重要性に、湊は改めて気づいた。

緊張の高まり

山田はコーヒーを淹れながら、湯気の立つカップを手に口を開いた。

「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、同じ調子で続ける。

「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」

湊は深く頷く。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。会議室の窓から午後の光が差し込み、湊は手に汗をかきながら資料を準備していた。

プロジェクターに接続し、スライドを開く。5名のチームメンバー全員が湊を見ている。

「すみません、5分だけ時間をください」

湊は一度息を吸い、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」

湊はグラフを指差しながら続ける。時系列グラフには、明確な下降トレンドが示されている。

「去年の同じ時期、ある機能の開発には平均して5日かかっていました。でも、今年は7日かかります。同じ規模の機能なのに、開発時間が1.4倍になっています」

佐藤が画面に身を乗り出した。

「これ、本当ですか？自分たちの感覚と一致してる...」

別のメンバーも小さく呟いた。

「確かに、最近開発が遅くなってる気がしてた」

湊は同じ調子で続ける。

「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、さらに続ける。

「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。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」

山田が画面を見つめ、しばらく黙ってから口を開いた。

「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」

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。具体的には、Gitのコミットログから機能開発にかかった時間を抽出し、JIRAチケットの作成日と完了日を比較して、実際の開発期間を計算しました」

山田が頷いた。

「なるほど。データの根拠が明確だから、説得力があるね」

別のメンバーがしばらく黙ってから質問した。

「でも、どうすればいいんですか？」

「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。佐藤がメモを取りながら頷いている。

「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、湊は資料に目を落とし、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。先ほどお話ししたように、開発コストが2倍になっている。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ります。例えば、開発時間の20%を返済に充てれば、将来的には速度が戻り、コスト削減につながると思います」

「それなら、説得力があるかもしれないね」

山田が頷いた。他のメンバーも同意の表情を見せた。

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。データの集計方法とか、教えてほしい」

「一緒に改善策を考えよう。技術的負債の返済、どこから始めればいい？」

メンバーの声が耳に残るなか、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

ストーリーで描かれる「重圧」

どう変えるかを考え始めるときに現れやすいのが、「説明責任を果たすにはどうすればいいか」という重圧です。湊が経験した重圧も、ここから生じています。

- **山田の後押し**　「チームに共有してみたら？」という言葉で、一人で抱え込まずチーム全体で問題を共有する重要性に気づく。エンジニアとして説明責任を果たすことの意味を実感している
- **勇気ある提案** チームミーティングで開発速度50%低下は開発コスト2倍になるというコスト換算で説明し、メンバーから「もっと詳しく話を聞かせて」「一緒に改善策を考えよう」という反応を得る
- **説明の難しさ**　専門用語の壁、数値化の困難、時間的制約で、技術的課題を非技術者に説明することの難しさを経験してきた。説明できないと技術的負債の返済に時間を割いてもらえない
- **転換点**　データの可視化とコスト換算によって、説明責任を果たす具体的な方法にようやく手が届いた。チーム全体で共有することで、次の一歩を踏み出そうとしている

この重圧の背景には、**説明責任を果たさないと投資判断が得られず問題が悪化する一方で、可視化・コスト換算・チーム共有によって説明責任を果たす道筋が見え始めている**という構造があります。転換の入り口に立っているのです。

なぜ説明責任を果たすのか

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欠かせない取り組みです。

エンジニアとして、技術的な課題を説明する責任があります。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

開発生産性の再定義

開発生産性の本質は第1章で述べた通り、約束の信頼性にあります。単にコードを書く速度やコミット数ではなく、約束した期日に正確に成果物を届けられるかという信頼性を意味します。

エンジニアとしての説明責任を果たすことは、この信頼性を高めるために不可欠です。なぜなら、技術的負債が蓄積している状態では、予測不可能なバグや遅延が発生し、約束を守れなくなるからです。

「投資が増えても生産性統計には見えない」という指摘は、Solow（1987）の「コンピュータ時代はどこにでも見えるが、生産性統計には見えない」やBrynjolfssonの「生産性パラドックス」の整理に代表され、ITの世界で古くから論じられてきました。IT投資が増えても労働生産性成長率が低迷した要因として、測定問題・タイムラグ・利益の再分配・IT管理の失敗が論じられ、後続研究では**組織変革と補完的に導入した場合**にプラスのリターンがあることが示されています。NBERの「Firm Data on AI」では、約70%の企業がAIを使用しているにもかかわらず、80%超が過去3年間で雇用・生産性に影響がなかったと回答しており、経済学者はSolowパラドックスの再来と位置づけています。個人レベルの生産性向上と組織レベルの成果向上の間には構造的なギャップがあり、組織の適応（プロセス再設計、レビュー体制、品質管理）が追いついていないためと解釈できます。

このギャップを説明する一つの枠組みが**アムダールの法則**です。並列化できる部分だけを高速化しても、全体の速度は「並列化できない部分の割合」で上限が決まります。開発では、実装・コーディング・ドキュメント作成などはAIで高速化しやすい一方、顧客の課題の特定・設計判断・要件定義・レビュー・承認などは人間の判断が中心で並列化しにくい領域です。本質的な仕事が全体の30%を占める場合、それ以外を10倍速くしても全体は約2.7倍にしかなりません。50%を占める場合は約1.8倍程度です。だから「コーディングが速くなった」という体感があっても、組織全体のスループットは同じ倍率では伸びず、「測れない」「数字に現れない」と感じる構造的な理由があるのです。

また、「AIを入れれば誰でも生産性が上がる」「ミクロの効率化がマクロの成果に波及する」といった通説は、実証研究では必ずしも支持されていません。California Management Review（2025）のメタ分析では、AI導入と集計レベルの生産性向上の関係が一貫しないこと、高精度だが不完全な自動化により誤判断が増加しうることなどが報告されています。AI導入＝生産性向上と短絡せず、湊のようにデータで可視化し、組織の適応（プロセス・レビュー・品質管理）とセットで考えることが重要です。

真の生産性向上は技術ではなく人材とプロセスの変革に70%の投資を要する（BCGの10-20-70）という示唆と整合します。そのため、こうしてデータで可視化し、コスト換算で説明し、チーム全体で共有して投資判断につなげる取り組みが重要になります。

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。

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押さえておくべき取り組みです。本節で扱う指標（開発速度・バグ発生率・コード複雑度など）は、DORAメトリクス（デプロイ頻度・変更リードタイム・変更失敗率・復旧時間。DORA metrics guide 参照）やForsgren et al.（2021）のSPACEフレームワーク（活動量だけでなく満足度・効率・フロー状態を扱う）と整合的であり、単一のoutput/input比に依存しない多面的な測定がPetersen（2011）の体系的レビューでも推奨されています。以下では、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

本章では、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。数字や指標だけでは本質的な課題は見えない。次の章では、湊がチームを巻き込んで本音の「痛み」を出し合い、小手先の指標に頼らず本質的な問題を特定していく過程を描きます。

一人で抱え込まずチーム全体で問題を共有し改善に取り組む。次の章でその先を描く。

参考文献

本章でデータ・指標を論じる際に参照した文献を、本文での言及順に挙げる。

- **Petersen, K.** (2011). "Measuring and predicting software productivity: A systematic map and review." *Information and Software Technology*. 単純な output/input 比（SLOC/工数等）は歪みを生むこと、メトリクスを評価に直結させるとゲーミングを誘発することが体系的レビューで指摘されている。
- **GitClear.** "Coding on Copilot" 2024 Data Report / "AI Copilot Code Quality" 2025（業界ホワイトペーパー）。1.5〜2億行規模のコードベース分析。Churn（書いてすぐ消される行）の倍増、リファクタリングの減少、コピー・クローンの急増、中長期的な技術負債の加速が報告されている。
- **Solow, R.** (1987). "We'd better watch out." *New York Times Book Review*. 「コンピュータ時代はどこにでも見えるが、生産性統計には見えない」という指摘。IT投資と労働生産性成長率の乖離の古典的議論。
- **Brynjolfsson, E.** 等、IT生産性パラドックス関連研究。測定問題・タイムラグ・利益の再分配・組織変革との補完的導入でプラスのリターンが示された後続研究。
- **NBER.** "Firm Data on AI" 等。AIを使用する企業の約70%がAIを利用している一方、80%超が過去3年間で雇用・生産性に影響がなかったと回答。個人と組織のギャップ、Solowパラドックスの再来として経済学者に位置づけられている。
- **DORA.** DORA metrics guide. https://dora.dev/guides/dora-metrics/ Four Keys（デプロイ頻度・変更リードタイム・変更失敗率・復旧時間）により開発の流れと安定性を対で捉える最小指標セットを提供。
- **Forsgren, N. et al.** (2021). "The SPACE of Developer Productivity." 開発

者生産性を Satisfaction / Performance / Activity / Communication / Efficiency の多次元で扱い、活動量だけの増加は長時間労働や悪いシステムの力技で悪化しうると警告している。

- **Martin Fowler.** "Is High Quality Software Worth the Cost?". https://martinfowler.com/articles/is-quality-worth-cost.html 内部品質とコストの関係（高品質なソフトウェアは実際には安く作れる）を論じている。