Qt Group

GUIDE

世界で最も信頼性の高いソフトウェアを構築する

リーダーのためのプレイブック

すべてのコードベースは、時間とともにソフトウェア内部品質が崩れていき、扱いにくくなります。問題は、そのような状態になるかどうかではありません。その進行を意図的にコントロールするのか、それとも手遅れになってから影響に直面するのかです。

本ガイドは、安全性や規格準拠が厳しく求められる業界、および競争の激しい業界の専門家インタビューから得られた知見を整理し、行動を起こす準備ができているリーダーのための実践的なフレームワークとしてまとめたものです。

R&Dへの投資には、目に見えにくい制約があります。調査によると、開発時間のおよそ50%は、変更を加える前に既存コードを理解することに費やされています。設計でも、実装でも、イノベーションでもありません。すでに存在しているものを把握するためだけに、多くの時間が使われているのです。

一方で、ソフトウェア内部品質の低下には、ある程度予測できるパターンがあります。

初期段階では、蓄積した複雑さはまだ管理できる範囲にあります。しかし、その影響はやがて指数関数的に大きくなり、組織は最終的に、問題対応に追われて前に進めなくなる限界点に達します。

その時点で、製品は事実上、寿命を迎えます。市場がなくなったからではありません。コードが複雑になりすぎて、安全に変更できなくなるからです。

本ガイドでは、成功するソフトウェア開発に共通するパターンを明らかにするため、専門家インタビューから得られた知見を整理しています。

そこから見えてくるのは、ソフトウェア開発組織の中に見えにくい対立を生む要因、従来のアプローチがうまく機能しない理由、そして信頼できるソフトウェアを作るために実際に有効な取り組みです。

本ガイドの内容

Part 1: 気付かないうちに進むソフトウェア内部品質の低下と、そのビジネスへの影響

Part 2: 信頼されるソフトウェアを継続的に作るための組織能力

Part 3: 信頼性の高いソフトウェアを構築するための8つの運用原則

対象読者:

R&Dをコストセンターではなく競争優位の源泉へと変えたい、戦略的な技術リーダー

本ガイドでわかること:

- スピードと品質を天秤にかける考え方が、なぜ誤った二者択一を生むのか
- ソフトウェア内部品質の低下が、どのように指数関数的に進むのか。そして、その進行を抑えるために何ができるのか
- 開発を、問題が起きてから対応する「火消し型」から、先回りして品質を作り込む「能力構築型」へ変えるための考え方
- 高い開発スピードと継続的な品質を両立するための運用原則

Part 1: 気付かないうちに進むソフトウェア内部品質の低下と、そのビジネスへの影響

理解にかかる見えないコスト

コードを変更するとき、調査データによれば、開発者は変更を加える前に、すでに存在するコードを理解するだけでおよそ50%の時間を使っています。

投資対効果を非常に単純化して計算すると、アーキテクチャ検証によって、この理解にかかる50%の工数を25%削減できます。この場合、全体ではおよそ10%の時間削減につながります。その時間は、より生産的な開発作業に使うことができます。

しかし、この「理解にかかる見えないコスト」は、問題の一側面にすぎません。

より深刻なのは、コードベースが時間とともに自然に扱いにくくなっていくことです。しかも、この現象は予測可能で、危険なパターンをたどります。

ソフトウェア内部品質の低下が進む曲線を理解する

ソフトウェア内部品質の低下とは、複雑さが蓄積し、アーキテクチャへの理解が失われていくことで、コードベースが少しずつ扱いにくくなっていく状態を指します。

これは、あらゆるコードベースで常に起こる、避けられない現実です。

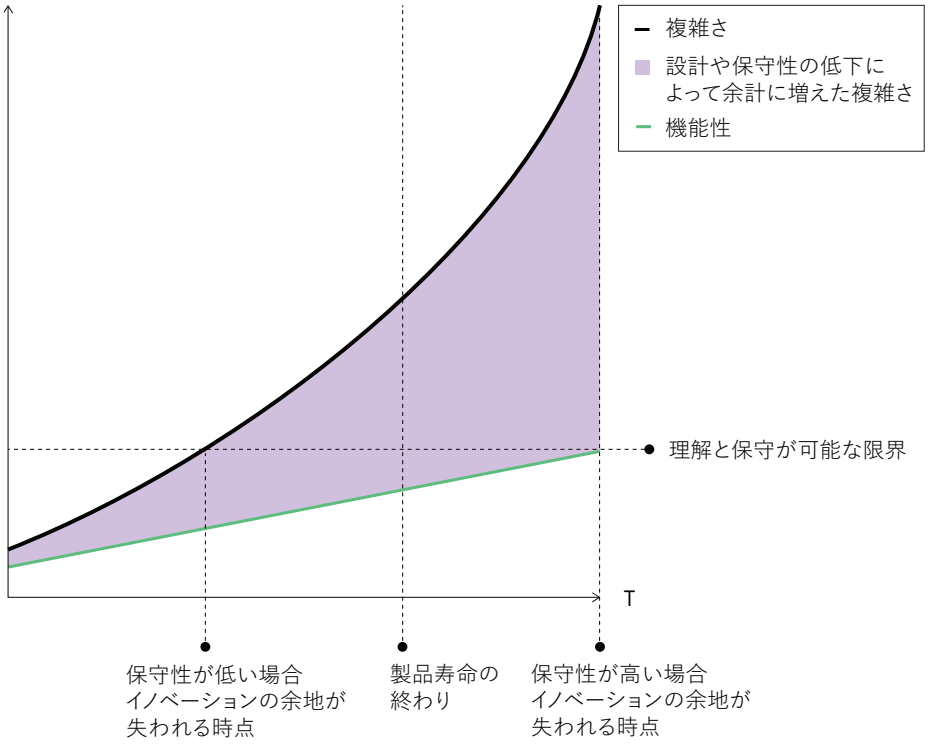


図: ソフトウェア内部品質の低下が進む曲線

時間の経過とともに機能は増えていきますが、それ以上の速さで複雑さが増大します。複雑さが「理解と保守が可能な限界」を超えると、コードベースは安全に変更しにくくなり、イノベーションの余地が失われていきます。保守性が低い場合、その限界には早く到達します。一方、保守性を高く維持できれば、製品寿命を延ばし、より長く価値を生み出し続けることができます。

Phase 1 - 管理可能な段階
製品ライフサイクルの初期段階では、ときどき行われる近道や場当たり的な対応も、まだ十分に管理できます。チームは開発スピードを維持でき、技術的負債が存在していても、開発の妨げになるほどではありません。

Phase 2 - 加速する段階
複雑さが積み重なり、その影響が大きくなり始めます。変更のたびにリスクが高まり、作業にかかる時間も増えていきます。システムは徐々に変更しづらくなり、一か所の修正が別の場所で予期しない不具合を引き起こすようになります。

Phase 3 - 限界点
組織は、問題への対応だけで手一杯になり、前に進めなくなる状態に達します。その時点で、その製品から得られる収益はほぼ頭打ちになります。

やがて製品は事実上の寿命を迎えます。市場から需要がなくなったからではありません。コードが複雑になりすぎて、安全に変更できなくなるからです。

なぜソフトウェア内部品質の低下は加速するのか

ある開発者が作り込んだ問題がチーム全体の開発に入り込むと、その人だけの問題ではなく、チーム全員の問題になります。

もし他の人が気付く前に修正できれば、その問題は存在しなかったも同然です。しかし問題が残ったままだと、解決されるまでチーム全員がその影響を受け続けます。場合によっては、それが数週間後、あるいは数か月後になることもあります。

これは、内部品質の低下が新たな技術的負債を生み、その技術的負債がさらに内部品質を悪化させるという連鎖が起こるためです。その結果、新たな問題が発生する頻度も高くなります。

コードが整理され、理解しやすい状態であれば、誤解が少なくなるため問題も発生しにくくなります。しかし、大量の技術的負債を抱えた扱いにくいコードを相手にしていると、問題はより高い頻度で発生します。

こうして終わりのない悪循環が生まれるのです。

さらに詳しく

スパゲッティコード — ソフトウェア開発で最も有名なアンチパターン

内部品質の低下が進む過程を説明するのが「ソフトウェア内部品質の低下曲線」です。しかし実際のコードでは、それはどのような姿をしているのでしょうか。

この記事では、複雑に絡み合った依存関係、あちこちに散らばったロジック、不明確な責任範囲などが静かに積み重なり、最終的にシステムが安全に変更できないほど脆弱になっていく様子を紹介します。

既存のコードベースを引き継ぎ、「なぜこんな状態になったのだろう」と感じたことがある人には、ぜひ読んでいただきたい内容です。

記事を読む ↗

重要な気付き:

スピードと品質のトレードオフが避けられないように見えるのは、ツールやプロセス、組織の能力が変えられないものだと考えている場合だけです。

実際には、それらは改善できます。

誤った二者択一 — スピードか品質か

「時間・予算・品質のうち、選べるのは2つだけ」

いわゆる「鉄の三角形」は、ソフトウェア開発において広く受け入れられている考え方です。

期限に間に合わなければ、市場投入のタイミングを逃してしまうかもしれません。そのため、多くの組織はどこかで妥協せざるを得ないと考えています。

しかし、このトレードオフが動かしようのないものに見えるのは、ツールやプロセス、組織能力が固定されていると考えている場合だけです。

実際には、それらは改善できます。

では、どうすればよいのでしょうか

その答えの一つが「ミニ・リファクタリング」です。

これは、開発者が日々の作業の中で行う、小さく継続的な改善です。

組織は、毎日あるいは一日に何度も、1分程度の小さなリファクタリングを実施できます。一つひとつは小さな改善ですが、それを積み重ねることで、時間と予算を確保して行う大規模なリファクタリングと同じような効果を得られます。

従来のアプローチ	継続的なアプローチ
技術的負債を放置し、大規模なリファクタリングが避けられなくなるまで蓄積させてしまう方法です。 その時点では、開発も進捗もすべて止めざるを得ません。コストが高く、影響も大きく、非常に負担の大きい対応になります。	ミニ・リファクタリングは、ほとんどコストをかけず、生産性への影響もほとんど見えない形で実施できます。 開発の生産性を妨げたり、作業を止めたりすることはありません。

もちろん、継続的な改善を取り入れたからといって、鉄の三角形が魔法のように消えるわけではありません。

時間や予算の制約は依然として存在します。

しかし、自動化によって、その制約の中で実現できることは大きく広がります。

たとえば、

- 同じ投資額でより高い品質を実現する
- 同じ品質をより短期間で実現する
- 高い品質と持続的な開発スピードを両立する

といったことが可能になります。

適切なプロセスを整えれば、鉄の三角形の制約は乗り越えられるのです。

信頼性を伴わないスピードは、本当の意味でのスピードではありません。
それは単なる混乱です。

スピードも品質の一部である

一見すると矛盾しているように聞こえるかもしれません。しかし、ユーザーが本当に求めているものを考えれば理解できます。

コンシューマ向けアプリケーションでは、完璧な製品を最後に出すよりも、十分に動作する製品をいち早く市場へ投入する方が重要な場合があります。

しかし、本当にスピードと品質は別物なのでしょうか。信頼性を伴わないスピードは、本当の意味でのスピードではありません。それは単なる混乱です。

何か問題が発生すると、チームは製品サポートに多くの時間を費やすことになります。そして最終的には、すでにリリースした機能に戻って修正しなければならず、新しい価値を生み出すための開発時間が失われます。

だからこそ、最初から正しく作ることが重要なのです。

品質やテストへの投資は、後から発生する手戻りや再作業を減らすことで、十分に回収できます。

本当に問うべきなのは、「スピードか品質か」ではありません。どのようなスピードを選ぶのか、です。

- 近道によって得られるスピード。今は早く出荷できるが、将来の手戻りが増え続けるスピード。
- 信頼できる仕組みによって得られるスピード。安定してリリースを続け、何年にもわ

さらに詳しく

拡大するソフトウェアプロジェクトにおける技術的負債への対応

技術的負債は単なるコード上の問題ではありません。開発スピード、保守コスト、製品寿命に直接影響する、ビジネス上のリスクです。

この記事では、大規模かつ長期間運用されるコードベースにおいて、開発を止めることなく技術的負債を分類し、優先順位を付け、計画的に削減していく方法を解説します。

記事を読む ↗

たって開発スピードを維持できるスピード。

Part 2: 信頼されるソフトウェアを継続的に生み出す組織能力

有効なアーキテクチャとは、生きたものであり、継続的に検証・維持されるものである。

信頼されるソフトウェアを作る組織能力とは何か

「信頼されるソフトウェアファクトリー」とは、特定のプロセスフレームワークや認証制度、あるいはツール群のことではありません。

それは組織としての能力です。

すなわち、顧客、規制当局、監査人、そしてソフトウェアを開発・保守するチーム自身から信頼されるソフトウェアを継続的に提供する能力を意味します。

ここで「ファクトリー（工場）」という言葉を使うのには理由があります。優れた工場は、一部の優秀な個人の頑張りに依存しません。品質が自然に生み出される仕組みを構築します。

最も優れた工場では、不具合は日常的に発生するものではなく、まれな例外になります。なぜなら、生産システムそのものが不具合の発生要因を未然に防ぐよう設計されているからです。

品質を競争優位として捉えるための組織的な整合

品質を「コストを増やすための守りの活動」と考えるのではなく、「競争優位を生み出す資産」と捉えるようになると、組織の行動は大きく変わります。

「完了」の定義を見直す

「Done（完了）」の定義は組織によって異なります。

しかし多くの場合、その定義には開発面の目標や指標しか含まれていません。「品質目標はどこにありますか」と尋ねると、多くのチームは初めてその欠落に気付きます。

品質基準を含めずに「完了」を定義すると、品質は**誰か別の人の責任**になってしまいます。

一方、品質目標が完了条件の中で明確に定義されていれば、それを達成する**責任はチーム全員で共有**されます。

オーナーシップを育てる

チームが自分たちの仕事にオーナーシップを持っていれば、経営層や管理職は過度に不安を感じる必要がなくなります。

しかし、その不安がマイクロマネジメントや過剰な管理へとつながると、メンバーは自分の仕事に責任感や当事者意識を持てなくなります。

ただし、オーナーシップには難しい側面があります。

オーナーシップは、単に「与える」ことはできません。

何十年にもわたって細かな管理に慣れてきた組織では、オーナーシップは一夜にして生まれるものではありません。

自ら責任を持つことが当たり前になる文化を育てる必要があります。

構造を検証できるアーキテクチャの整合性

ソフトウェアアーキテクチャは、しばしば開発初期の設計作業として扱われます。アーキテクチャ図を作成し、レビューや承認を受け、その後は棚にしまわれてしまいます。

しかし、本当に有効なアーキテクチャは、生きたものであり、継続的に検証・維持されるものです。

現在では、アーキテクチャ検証ツールによってアーキテクチャの遵守状況を確認できます。コンポーネント間のやり取りは、設計時に意図した方法でのみ行われるべきです。これによって何が実現できるか考えてみてください。

従来のやり方では、アーキテクチャ文書を作成し、その後に開発者が実装を進めます。そして時間が経つにつれて、文書と実装は少しずつ乖離していきます。どれほど乖離しているのかは、大規模な監査や重大な障害が発生するまで誰にも分かりません。

自動化という解決策

ドキュメントを更新し続けることにはコストがかかります。

しかし、更新しないことのコストはさらに大きくなります。

紙や静的な文書ではなく、機械が読み取れる形式でアーキテクチャを管理し、その上でアーキテクチャ検証を行うことができます。これは自動化できる活動の一つです。

機械可読なアーキテクチャとは、ドキュメントとコードを自動的に照合できることを意

味します。両者にずれが生じた場合、それを数か月後ではなく、その場で検知できます。

ドキュメントのパラドックス

もし、動作するソフトウェアとドキュメントのどちらか一方しか残せないとしたら、どちらを選ぶでしょうか。

多くの人はソフトウェアを選ぶでしょう。しかし実際には、ドキュメントの方が重要です。

優れたドキュメントが残っていれば、ソフトウェアは作り直せます。しかし、ソフトウェアだけが残りドキュメントがなければ、システムを理解する手掛かりが失われます。

それは行き止まりです。

現場での検証結果

あるアーキテクトは、自分自身で興味深い実験を行いました。

リファクタリングに必要な工数を見積もったのです。最初の見積もりは「おそらく数週間程度」でした。

しかしアーキテクチャ検証を実行してみると、その見積もりは大きく外れていることが分かりました。実際の依存関係は想像以上に複雑で、見えていなかったのです。自分のコードを熟知している専門家であっても、システム構造を正確に把握できていませんでした。

そして彼らは次の結論に至りました。

「専門家ですらアーキテクチャ全体を頭の中で把握できないのであれば、他の誰にもできるはずがない」

ROIを重視する意思決定者のための考え方

アーキテクチャ検証によって、コードを理解するために費やされる時間（全体の約

さらに詳しく

静的コード解析とアーキテクチャ検証がもたらす隠れたROI

ソフトウェア品質の経済的な価値は、何か問題が起きるまでは見えにくいものです。この記事では、不具合を未然に防ぐことでどれだけのコスト削減につながるのかを定量的に説明します。対象となるのは単純な修正工数だけではありません。その後に発生するデバッグ、テスト、本番環境での対応、さらにはアーキテクチャ上の問題対応まで含まれます。経営層やマネジメント層に対して品質投資のビジネス価値を説明する際に役立つ実践的な内容です。

記事を読む ↗

50%）を25%削減できるとすると、開発全体では約10%の時間短縮につながります。

監査対応の効率化

監査の初回レビューで、指摘事項が200件以上ある場合と20〜50件程度しかない場合とでは、その後に必要となる工数やコストは大きく異なります。

指摘事項が少なければ、監査対応は短期間かつ低コストで完了します。

一方で、多数の指摘事項がある場合は、それぞれの修正だけでなく、原因の分析や関連する問題の切り分けも必要になります。

100件を超える相互に関連した問題を解決する作業は、はるかに困難です。

製品寿命を延ばす

製品の寿命を当初の想定より長く延ばせた期間は、そのまま利益につながります。

なぜなら、大きな開発投資はすでに回収済みだからです。たとえば、製品寿命を10年から11年へ1年延ばせれば、その投資から得られる利益は10%増加することになります。

ミニ・リファクタリングによってソフトウェア内部品質の低下を緩やかにできれば、組織が限界に達する時期を後ろへずらすことができます。

つまり、より長期間にわたって収益を生み出し続けられるということです。

運用の自動化によって品質を日常業務に組み込む

締切が迫ると、人によるプロセスには近道が入り込みます。**自動化されたプロセスは入り込みません。**

開発初日でもリリース前日でも、同じルールで同じように実行されます。

信頼できるソフトウェアを構築するための3つ目の要素は、品質チェックを極めて自動化し、即座にフィードバックが得られる状態にすることです。

理想は、「正しいやり方を選ぶ方が、間違ったやり方を選ぶより簡単である」状態です。

自動化の優先順位をどう決めるか

自動化できるのであれば、自動化すべきです。

理由は単純です。

自動化された作業の実行コストは非常に低く、プレッシャーがかかった状況でも実施漏れが発生しないからです。一度仕組み化してしまえば、人が常に意識し続ける必要はありません。

ここで重要なのは「プレッシャーがかかった状況でも」という点です。

締切が迫ると、人によるプロセスには近道が入り込みます。しかし、自動化されたプロセスは違います。開発初日でもリリース前日でも、同じルールで同じように実行されます。

ただし、自動化を急ぎすぎることにも注意が必要です。

組織によっては、本来その製品には不要なものまで自動化するために、何千時間もの工数を費やしてしまうことがあります。

原則はシンプルです。必要性が確認されたものを自動化します。推測だけで自動化しません。

まずは規制や規格によって求められているもの、そして過去の不具合から頻繁に発生することが分かっている問題から始めるべきです。

その後、本当に価値のあるチェック項目を見極めながら、自動化を段階的に拡大していきます。

ただし、静的コード解析だけは最初から導入すべきです。

静的コード解析は、人間がレビューする前の段階で一般的なエラーや品質問題を検出する自動化技術です。その他の取り組みは後から追加しても構いません。最初からすべてを導入しようとすると、組織は必要な成果を出せなくなるからです。

人とツールの役割分担

機械は、形式的なルールへの適合を確認することが得意です。

自動化された静的解析は、

- 関数の引数の数
- 結合度（Coupling）
- コード重複

といった項目を正確にチェックできます。

一方、人間が得意なのは意味的な妥当性の判断です。人はコードの意図や文脈を理解しながら判断できます。

たとえば、

- この抽象化は適切か
- この問題はどれほど緊急か
- 今すぐ修正すべきか、それとも後回しにしても安全か
- これは本当に問題なのか、それともドキュメントを修正すべきなのか

といった判断は人間の役割です。

現代の開発者に求められる役割

今日の開発者には、コードを書くことだけでなく、品質にも責任を持つことが期待されています。

開発者は、

- ユニットテスト
- コードレビュー
- CI/CDパイプラインのテスト自動化

などを日常的に行っています。

実際には、現在の開発者が行っている仕事の多くは何らかの形で品質に関わっています。純粋なコーディング作業は、その一部に過ぎないかもしれません。

開発者に品質への責任を持たせることは、単純に仕事を増やすことではありません。重要なのは、品質活動のために別のプロセスへ切り替えることなく、高品質な開発を実現できるツールや仕組みを提供することです。

さらに詳しく

コードとしてのアーキテクチャ：開発者に優しいアーキテクチャ検証アプローチ

多くのソフトウェアアーキテクチャは、Wikiやホワイトボードの写真、あるいは担当者の頭の中にしか存在していません。そして、実装されたコードが本当にそのアーキテクチャに従っているかを自動的に確認する手段もありません。

この記事では、「Architecture as Code」という考え方を紹介します。アーキテクチャをバージョン管理可能なPythonコードとして表現し、CIゲートとして自動的に検証できるようにするアプローチです。新規プロジェクトにも既存システムにも適用できます。

記事を読む ↗

QAの新しい役割

従来の品質保証担当者は、今後ますます「監査人との橋渡し役」に近い役割を担うようになります。組織には、製品やプロジェクト全体における品質状況を把握し、それを監査人に説明できる人材が必要です。なぜなら、組織は監査を通過しなければならないからです。

しかし、品質が重要なのは監査のためだけではありません。

ソフトウェア品質は、組織全体で取り組むべきものです。その考え方を突き詰めると、「全員がある意味で品質エンジニアである」という状態になります。もちろん、単にそう宣言するだけでは実現しません。誰もが品質活動に参加できるようにするためのツールや仕組みが必要です。

Part 3: 信頼性の高いソフトウェアを構築するための8つの運用原則

これらの原則は、「信頼されるソフトウェアを継続的に生み出す組織能力」を日々の実践へ落とし込むためのものです。

信頼できるソフトウェアを構築する組織と、技術的負債を積み上げ続ける組織。その違いを生み出す共通パターンがここにあります。

従来型のアプローチ	継続的なアプローチ
問題が起きるまで品質を後回しにし、発生後に大規模で高コストな対策を行う。	小さく継続的な投資によって品質を維持し、大きな問題の発生を未然に防ぐ。

品質は開発プロセスの最後に追加するものではありません。リリース直前に確認するだけのものでもありません。

品質は、開発ライフサイクル全体を通じて継続的に維持されなければならないのです。

原則1: 品質をイベントではなく継続的な活動にする

品質は開発プロセスの最後に追加するものではありません。リリース直前に確認するだけのものでもありません。

品質は、開発ライフサイクル全体を通じて継続的に維持されるべきものです。

分かりやすい例で言えば、虫歯になってから歯科医院で長時間の治療を受けるよりも、毎日2〜3分歯を磨く方がはるかに効果的です。

問題が起きてから大掛かりな対策を講じることもできますが、本当に目指すべきなのは、日々の習慣によって問題の発生そのものを防ぐことです。

品質ゲートの実践

品質ゲートとは、テスト工程へ進む前に満たすべき条件を全員が共有するためのチェックポイントです。準備が整っていれば、確認は数分で終わります。整っていなければ、問題が広がる前に発見できます。

品質ゲートが開発プロセスに組み込まれると、それはボトルネックではなくなります。チームはその価値を理解し、自然に活用するようになります。

導入時の考え方

継続的な改善は、すぐに大きな成果を生むわけではありません。

コードを書いてから半年も経てば、詳細な内容は忘れてしまいます。自分で書いたコードであっても、他人のコードのように感じることがあります。ドキュメントや整理されたアーキテクチャはこの問題を解決しますが、その効果は後になって現れます。だからこそ、先に投資が必要なのです。

ここで陥りやすいのが、「すべてを一度に導入しようとする」ことです。これは失敗への近道です。まずは規制や規格で要求されている項目と、自社の不具合履歴から見えている問題パターンから始めるべきです。その後、本当に価値のあるチェックを見

極めながら、自動化を段階的に追加していきます。

改善を一つ導入する。
効果を測定する。
そして次の改善を行う。

この繰り返しが重要です。

また、必要以上の取り組みを行わないことも重要です。実際には、自社製品には不要な規格への対応に何千時間も費やしてしまう組織もあります。適切なスコープ設定こそが無駄を防ぎます。

原則2: 繰り返し行うものはすべて自動化する

自動化の最大の強みは、プレッシャーの影響を受けないことです。

開発初日であっても、リリース数時間前であっても、自動化されたチェックは同じ厳密さで実行されます。

その結果、他の部分で妥協が発生しても、品質が一定レベル以下に落ちることを防げます。自動化は品質を安定させ、開発プロセスそのものに組み込みます。

原則3: アーキテクチャは生きたドキュメントでなければならない

アーキテクチャ文書は実装と常に一致している必要があります。ずれる可能性があるなら、必ずずれます。

開発が進むにつれてシステムは変化します。人は元のドキュメントを更新し忘れます。すべてを手作業で維持し続けることは現実的ではありません。

解決策は、より厳格な運用ルールではありません。ここでも答えは自動化です。

機械可読なアーキテクチャには二つの意味があります。

1. アーキテクチャがツールで解析可能な形式で表現されていること
2. ツールが実装とアーキテクチャの整合性を自動的に検証できること

原則4: 品質を全員の責任にする

品質は一つの部門だけの責任ではありません。開発プロセス全体を通じて、ソフトウェアに関わるすべての人が担うべきものです。

これは、「開発者がコードを書く」「QAが不具合を見つける」という従来型モデルからの大きな変化です。

ただし、それを実現するには品質活動へのハードルを下げる必要があります。開発者が日常的に、そして頻繁に確認できるようにすることが重要です。リアルタイムのIDEチェックは、Wordのスペルチェックのようなものです。

別のツールを起動したり、別システムでチケットを作成したりする必要があると、それは負担になります。一方で、普段使っているツールの中でリアルタイムに確認できれば、それは自然な作業の一部になります。

原則5: 早く失敗し、より早く学ぶ

開発初期に不具合を見つけるコストを1とすると、

- 後工程で見つけるコストは10倍
- 本番環境で見つけるコストは100倍
- 顧客先で見つかるコストは1,000倍

になります。

実際には、それぞれさらに1,000倍して考えてもよいかもしれません。

失敗そのものは避けられません。重要なのは、どれだけ早く発見し、そこから何を学ぶかです。

その経済的な影響は明確です。

開発の早い段階で不具合を見つけるコストを1とすると、後工程で見つけるコストは10倍、本番環境で見つけるコストは100倍、顧客先で発見されるコストは1,000倍になります。実際には、これらの倍率はさらに1,000倍して考えるべきかもしれません。

このような指数関数的なコスト増加を考えると、早期発見は単に望ましいだけでなく、経済的にも不可欠です。

メトリクスを確認するときは、成功したパイプラインだけでなく、失敗したパイプラインにも注目してください。たとえば、ある組織が12万件のパイプラインを実行したことを誇っていたとします。しかし、そのうち4分の1が失敗していたとしたら、それは重要な事実を示しています。つまり、その仕組みが3万件もの問題を早い段階で検出し、修正できているということです。

これはむしろ良い知らせです。問題が早期に表面化しているため、品質は着実に改善されているからです。

失敗したビルドは、システムが意図どおりに機能している証拠でもあります。

問題が連鎖的に広がる前に検出できており、しかも修正コストがまだ低いうちに対処できているからです。

建設に例えるなら、基礎工事のミスは、壁を建て終わって建物全体が崩れてしまった後ではなく、できるだけ早い段階で発見したいはずです。

ソフトウェア品質も同じです。組織は、不具合やミス、品質上の問題を、より大きく深刻な問題へと発展する前に、できるだけ早く、そして最も低コストな段階で発見しなければなりません。

透明性を土台にする

完璧さよりも透明性の方が重要です。

もしチームが、品質をある程度犠牲にしてでも開発スピードを優先することに明確に合意しており、そのトレードオフを全員が理解しているのであれば、不具合が発生しても大きな対立にはなりません。

問題の多くは、関係者の認識や期待値のずれから生じます。たとえば、開発チームが開発スピードの最大化を重視している一方で、マネジメントが「絶対に不具合のない品質」を期待している場合、両者の間には避けられない摩擦が生まれます。そして、その過程で誰かが責任を問われることになります。

透明性は、このような問題を防ぎます。最初の段階で期待値を共有し、認識を揃えておくことができるからです。

失敗から学ぶ文化をつくる

人は必ずミスをします。だからこそ、組織には失敗から学ぶ文化が必要です。ミスが起きること自体は避けられませんし、それは特別なことではありません。

人は必ずミスをします。

だからこそ、組織には失敗から学ぶ文化が必要です。

ミスが起きること自体は避けられませんし、それは特別なことではありません。

重要なのは、ミスが発生したかどうかではなく、その経験からチームが学べるかどうかです。一度ミスをするのは自然なことです。場合によっては避けられないこともあります。しかし、同じミスを繰り返すのであれば、それは個人の問題ではなく、組織や仕組みの問題です。

そのためには、表面的な原因だけを見て終わらせてはいけません。ミスや不具合が発生したとき、「コードが間違っていた」「単純なタイプミスだった」といった説明だけで片付けてはいけません。

その背景にある原因を掘り下げる必要があります。

なぜ起きたのでしょうか。

- 金曜日の午後で、早く帰ろうとしていたのではないか
- 他の開発者からの割り込みが頻繁に発生していたのではないか
- 非現実的なスケジュールによる過度なプレッシャーがなかったか

こうした組織的・構造的な要因を理解することが、再発防止につながります。そして、チームが本当に改善できるのも、こうした根本原因に対してです。

ミスは個人の失敗として扱うのではなく、チーム全体で学ぶ機会として活用すべきです。開発者が自分のミスや、その背景にあった状況について共有することで、チーム全体が学びを得られます。

一人のミスから得られた教訓が、チーム全体の再発防止策になるのです。

原則6: 複雑さは最大の敵である

人間の認知能力には限界があります。そして、ソフトウェアはその限界を前提として設計されなければなりません。

コードを書くという作業は、人間が頭の中で扱える情報量の限界に常に挑戦する仕事です。私たちは、ワーキングメモリの容量や集中力の持続時間に制約されており、細かな情報を忘れてしまうこともあります。また、複数の抽象化レベルを同時に理解しながら考えることは容易ではありません。ソフトウェア開発は、本質的に認知負荷の高い作業なのです。

さらに、ソフトウェアの複雑さは、人間が完全に把握できる範囲を本質的に超えています。

ソフトウェアは、人類が生み出した人工物の中でも最も複雑なものの一つと言えるでしょう。あらゆるコンポーネントが他のコンポーネントと接続でき、あらゆる要素が相互に関係し得ます。設計の選択肢は事実上無限に存在し、その無限の組み合わせが避けがたい複雑さを生み出します。

では、どうすればよいのでしょうか。

その答えが、モジュール化と明確なインターフェースです。これらはソフトウェアアーキテクチャの基本原則です。

複雑なシステムを理解しやすい単位へ分割することで、人間の認知負荷を軽減します。しかし、こうした原則は、継続的に維持され、守られて初めて効果を発揮します。そのために欠かせないのが、アーキテクチャ検証です。

原則7: コンプライアンスは目的ではなく、良い開発の結果として実現される

コンプライアンスは、開発の最後になって追加するものではありません。優れたソフトウェア開発プロセスを実践した結果として、自然に実現されるべきものです。

コンプライアンスは、開発の最後になって追加するものではありません。
優れたソフトウェア開発プロセスを実践した結果として、自然に実現されるべきものです。

ソフトウェア品質の管理は、組織が「やるか、やらないか」を選べるものではありません。それは事業を継続するための必須要件です。監査をスムーズに通過するための最も確実な方法は、開発の初期段階から日々の業務の中にコンプライアンスを組み込むことです。

なぜなら、監査に合格できなければ、認証や資格を失うだけでなく、契約の維持や市場への参入そのものが難しくなる可能性があるからです。その影響は大きく、場合によっては事業に深刻なダメージを与えることもあります。

もちろん、アーキテクチャ検証ツールだけで認証プロセスそのものを不要にできるわけではありません。しかし、それらのツールは認証や監査対応に伴う負担を大幅に軽減できます。

組織は、設計書として定義されたアーキテクチャと実際のコードが整合しているかを、リアルタイムで継続的に確認できます。これは、本ガイドの前半で説明した「実態を反映し続けるコンプライアンス」、すなわち継続的コンプライアンスを実現するための重要な仕組みです。

原則8: AIは良いことも悪いことも増幅する

AIは、ソフトウェア品質の原則そのものを変えるわけではありません。AIは、組織がすでに持っている基盤を増幅するだけです。

そのため、AIを活用しようとする組織は、ある皮肉な現実に直面します。AIが必要とするものは、多くの組織が「スピード重視」の名の下に手放してしまったものだからです。AIを効果的に活用するためには、次のような情報が必要です。

- 要求仕様書
- 設計仕様書
- 構造化された知識や設計情報

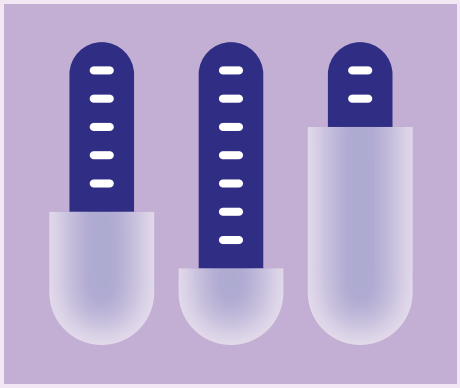
しかし、これらは短期的な開発スピードを優先する中で、十分に維持されなくなりがちな成果物でもあります。

こうした基盤を維持しているチームは、AIを効果的に活用できます。一方で、要求や設計に関する情報が整理されていない組織は、AIの能力を十分に引き出すことができません。どれほど優れたプロンプトを書いたとしても、コンプライアンスをプロンプトだけで実現することはできません。

コンプライアンスは、適切な開発プロセスと品質管理の積み重ねによって達成されるものであり、AIが代わりに保証してくれるものではないのです。

改善状況を測定するための指標

本当に改善できているかどうかは、適切な指標を継続的に確認することで判断できます。



1. マージリクエストの頻度

開発者が週に一度しかマージしていない場合、それは問題の兆候かもしれません。作業をまとめてから一度に統合しているため、フィードバックが遅れ、開発の進捗やチーム間の連携を妨げる可能性があります。マージは単なるコード統合ではありません。それはチーム内のコミュニケーションであり、品質を少しずつ積み上げていくための重要な活動です。頻繁にマージが行われている状態は、開発プロセスが健全に機能していることを示しています。

2. パイプライン失敗率

12万件のパイプラインを実行し、そのうち25%が失敗している組織は、必ずしも問題を抱えているわけではありません。むしろ、その3万件の失敗は、多くの問題を早い段階で発見し、すぐに修正できていることを意味します。品質が向上するのは、問題が修正コストの低い段階で表面化するからです。失敗したビルドは、問題が大きく広がる前にシステムが検出できている証拠でもあります。

3. 監査指摘件数の推移

監査の初回レビューで指摘される問題の件数を継続的に追跡してください。指摘件数が200件以上から20〜50件程度まで減少しているのであれば、継続的コンプライアンスの取り組みが実際に機能し始めていると考えられます。この指標は、品質活動が「監査前だけの確認作業」から「日常的な取り組み」へ移行できているかを示します。

4. コード理解に費やす時間

四半期ごとに、チームが既存コードの理解に費やす時間と、新機能開発に費やす時間を比較してください。コードを理解するための時間が増えているのであれば、複雑さが増大している可能性があります。逆に、その時間が減少しているのであれば、アーキテクチャ検証や品質改善の取り組みが効果を発揮していると考えられます。

内部品質の低下は避けられない。だからこそ品質を仕組みにする

さまざまな業界で積み重ねられてきた長年の経験から、次の5つの事実が見えてきます。

1. ソフトウェア内部品質の低下は止められない

すべてのコードベースは、時間とともにソフトウェア内部品質が崩れていきます。そして、その進行が自然に止まることはありません。マネジメントが望まないからといって防げるものでもありません。むしろ、納期やコストのプレッシャーはその進行を加速させます。追い込まれたチームほどミスが増え、問題はさらに蓄積していきます。やがてその影響は指数関数的に拡大し、新機能の開発ではなく問題対応に開発リソースの大半が費やされるようになります。

2. 個人の頑張りよりも仕組みが重要である

必要なのは、さらに長時間働くことではありません。

重要なのは、

- 自動化
- 継続的な品質管理
- 実態を反映し続けるドキュメント
- チーム全体での責任共有

といった仕組みです。

品質を一部の専門家だけのものではなく、誰もが維持できる状態にすることが重要です。そして、自動化できるものは積極的に自動化すべきです。

3. 品質問題のコストは想像以上に大きい

不具合を開発初期に発見するコストを1とすると、

- 後工程での発見は10倍
- 本番環境での発見は100倍
- 顧客先での発見は1,000倍

のコストがかかります。

現実には、それぞれさらに1,000倍して考えるべきかもしれません。一方で、製品寿命を延ばすことは直接的な利益につながります。たとえば製品寿命を10年から11年へ延ばせれば、その投資から得られる利益は10%増加します。

4. 文化と組織能力は同時に育てなければならない

「オーナーシップを持とう」と宣言するだけでは、当事者意識は生まれません。何十年にもわたってマイクロマネジメントが続いてきた組織文化は、一朝一夕には変わらないからです。品質を組織全体の活動にするためには、誰もが無理なく品質活動に参加できるツールや仕組みが必要です。

5. AI時代だからこそ基礎が重要になる

AIはソフトウェア品質の原則を変えるものではありません。むしろ、要求仕様や設計仕様といった基礎の重要性を改めて浮き彫りにしています。

AIが効果的に機能するためには、

- 要求仕様書
- 設計仕様書
- 構造化された知識

が必要です。

また、コードを誰が書いたかに関係なく、品質やコンプライアンスに対する責任は開発者と組織に残ります。どれほど優れたプロンプトを書いたとしても、コンプライアンスをプロンプトだけで実現することはできません。

これらを理解し実践できる組織は、単に成功するだけではありません。

「スピードか品質か」という古い対立や、「品質を犠牲にしたスピード」「スピードを犠牲にした品質」といった非効率な選択に縛られ続ける組織に対して、大きな競争優位を築くことができます。

次に取るべき行動

今から行動を始めることで、その効果は時間とともに積み重なっていきます。まずは、本ガイドで紹介した原則の中から一つを選び、今四半期の間に確実に実践してください。その効果を測定し、成果を確認したうえで、適用範囲を広げていきましょう。

賢く構築し、継続的に改善していきましょう。

www.qt.io/axivion