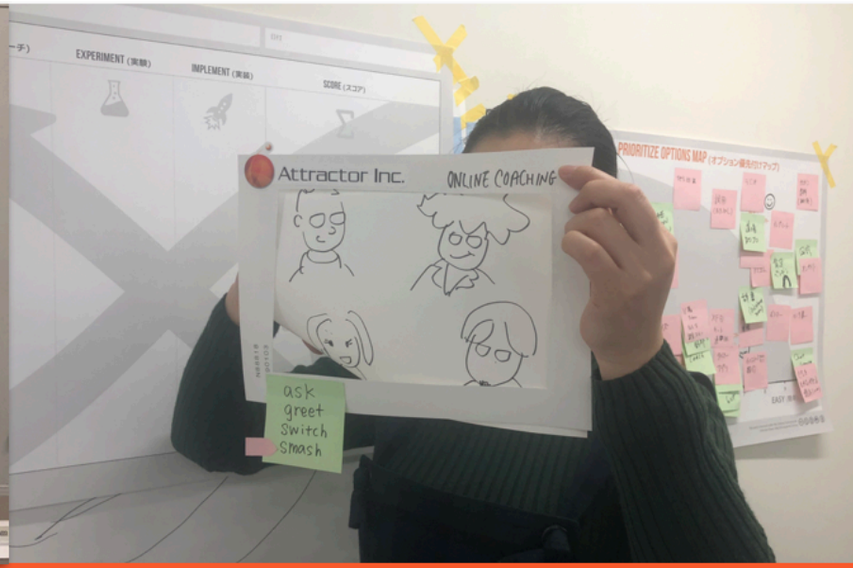
レガシーコードからの脱却

2020/2/14

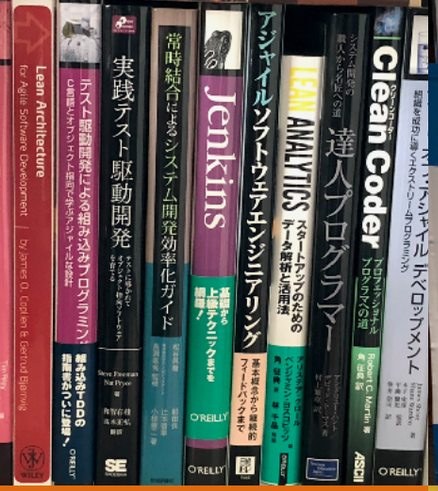
株式会社アトラクタ
吉羽龍太郎 (@ryuzee)

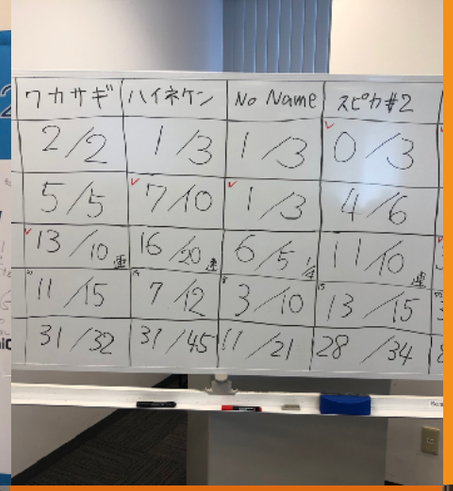
- 初版: 2019/10/03 AWS DevDay
- 改訂: 2019/10/31 EOF2019
- 改訂: 2020/02/14 Developers Summit 2020

株式会社アトラクタについて



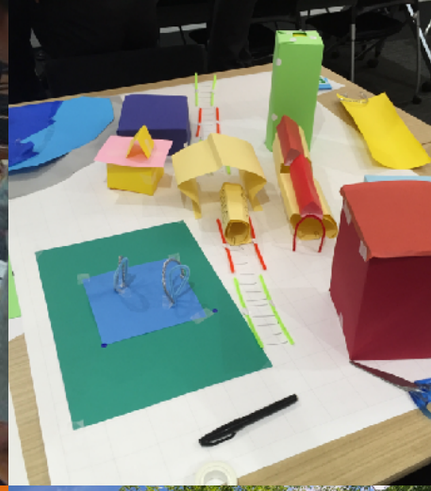
- 社名：株式会社アトラクタ　英文表記：Attractor Inc. / https://www.attractor.co.jp
- 設立：2016年12月
- 所在：東京都港区
- 開発プロセスに関するコンサルティングやトレーニングを提供
- アジャイル開発 / DevOps / チーム育成 / クラウドコンピューティング / ドメインモデリングなどが専門領域





登壇

各種イベントでの登壇



コーチング

アジャイル開発やDevOps
に関するオンサイトコーチ
ング

オンサイト
トレーニング

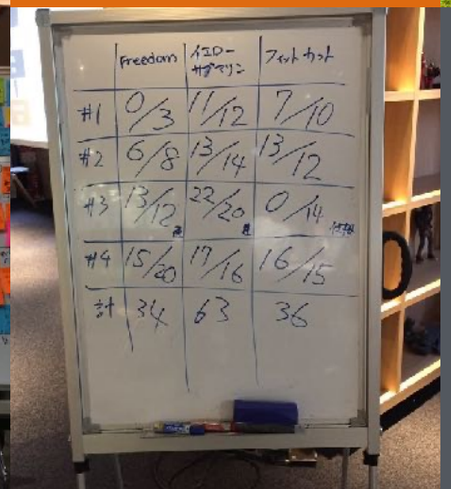
アジャイル開発や組織づくり
に関するオンサイト
トレーニング

執筆・翻訳

技術書やドキュメントの
執筆や翻訳

認定研修

認定スクラムマスター研修
等の主催

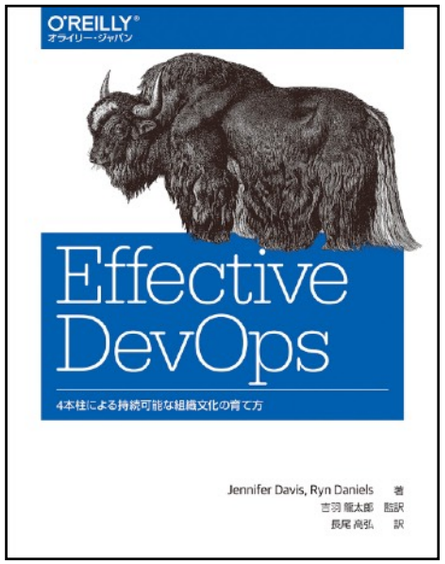


などなど

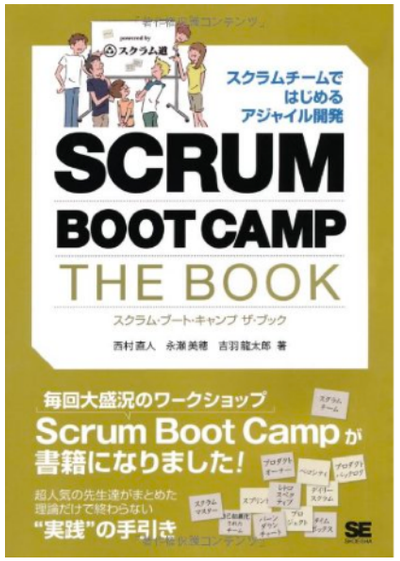
自己紹介

- 吉羽龍太郎 (@ryuzee)
- 株式会社アトラクタ取締役CTO/アジャイルコーチ
  - 野村総合研究所、Amazon Web Servicesなどを経てアトラクタを創業
- 開発プロセス/アジャイル開発/DevOps/クラウドコンピューティングが専門
  - Scrum Alliance Certified Team Coach (CTC)
    - コーチングを受けることで認定スクラムマスター取得可能
  - Microsoft MVP for Azure

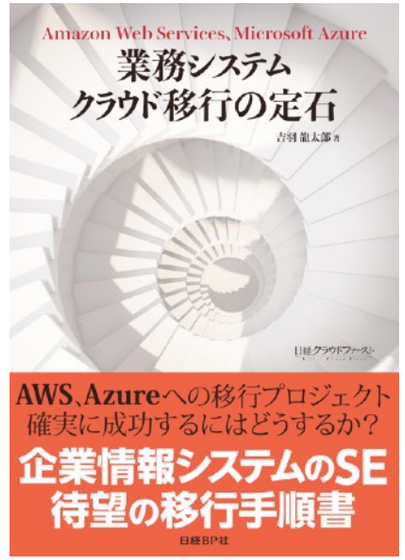
著書・訳書（買ってね）


O'REILLY
オライリー・ジャパン
Effective DevOps
4本柱による持続可能な組織文化の育て方
Jennifer Davis, Ryn Daniels 著
吉羽 龍太郎 監訳
長尾 高弘 訳

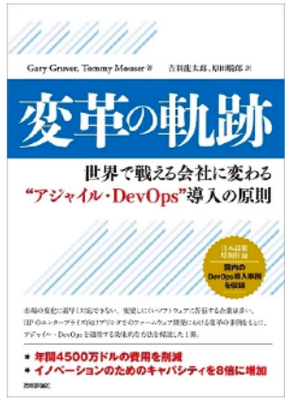

役職も部署もない
全員主役の
マネジメント
Joy, Inc.
ジョイ・インク
How We
Built a Workplace
People Love
リチャード・シェリダン 著
喜びを追求しながら事業を継続できる会社
あなたは、誰を
喜ばせますか？
日本交通 川鍋会長 推薦！

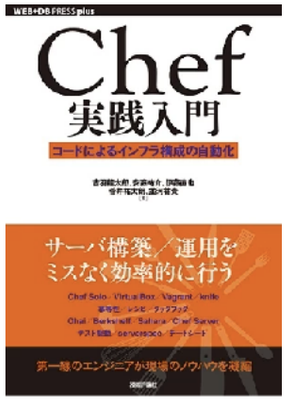

スクラムチームで
はじめる
アジャイル開発
SCRUM
BOOT CAMP
THE BOOK
スクラム・ブート・キャンプ ザ・ブック
西村直人 永瀬美穂 吉羽龍太郎 著
毎回大盛況のワークショップ
Scrum Boot Camp が
書籍になりました！
超人気の先生達がまとめた
理論だけで終わらない
"実践"の手引き
SE


Kanban
in
Action
カンバン仕事術
チームではじめる見える化と改善
Marcus Hammarberg
Joakim Sundén
O'REILLY
オライリー・ジャパン


Amazon Web Services, Microsoft Azure
業務システム
クラウド移行の定石
吉羽 龍太郎 著
日経クラウドファースト
AWS、Azureへの移行プロジェクト
確実に成功するにはどうするか？
企業情報システムのSE
待望の移行手順書
日経BP社

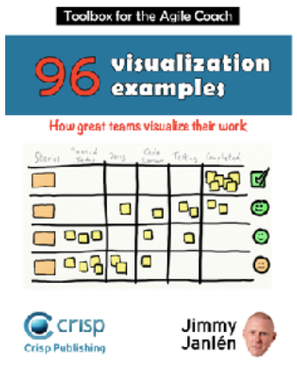

Amazon
Web Services
企業導入ガイドブック
AWSでシステムを構築する前に
知っておきたいことのすべて。


Gary Gruver, Tommy Mouser 著
吉羽龍太郎、原田騎郎 訳
変革の軌跡
世界で戦える会社に変わる
"アジャイル・DevOps"導入の原則
年間4500万ドルの費用を削減
イノベーションのためのキャパシティを8倍に増加


WEB+DB PRESS plus
Chef
実践入門
コードによるインフラ構成の自動化
サーバ構築／運用を
ミスなく効率的に行う
第一線のエンジニアが現場のノウハウを凝縮


サーバ/インフラ
エンジニア
養成読本
DevOps編
開発者と運用者の壁を取り除く
DevOpsのススメ
Ansibleによる
サーバ環境の構築
継続的
インテグレーション
入門


Software in
30 Days
スクラムによるアジャイルな
組織変革"成功"ガイド
組織変革を
目指す
マネジメントの
ためのスクラム入門
ASCII


Toolbox for the Agile Coach
96 visualization examples
How great teams visualize their work
crisp
Crisp Publishing
Jimmy
Janlén


CakePHPで学ぶ
継続的
インテグレーション

【宣伝】新しい書籍が出ます!!

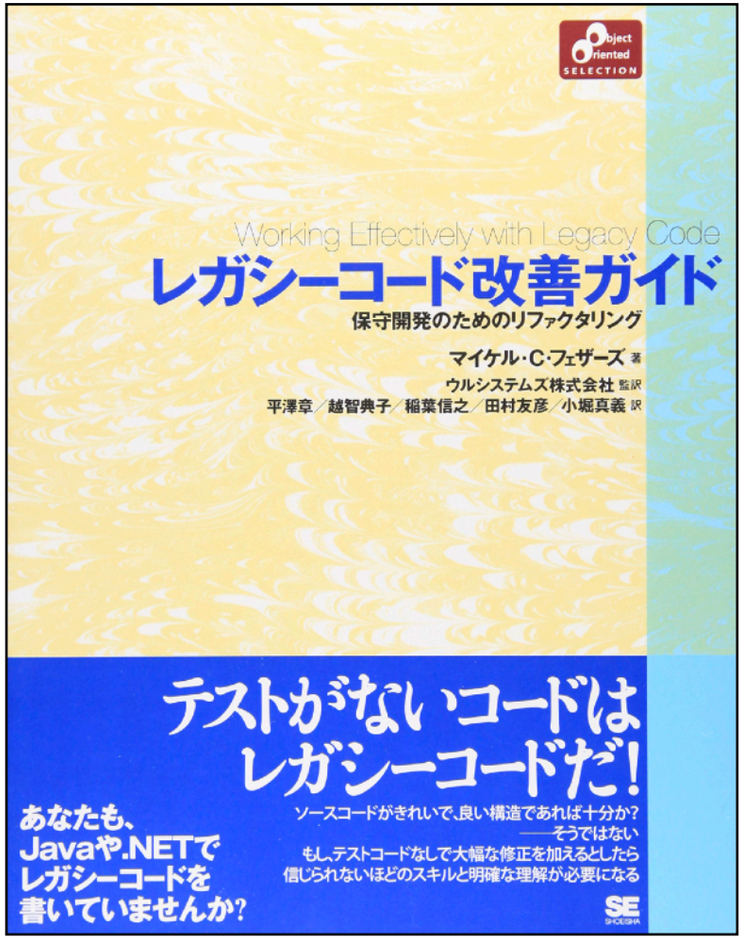
- 『みんなでアジャイル』
- レガシーコードからの脱却の訳者4人による翻訳
- 2020/3/19 発売予定
- こちらもよろしくお願いします!!

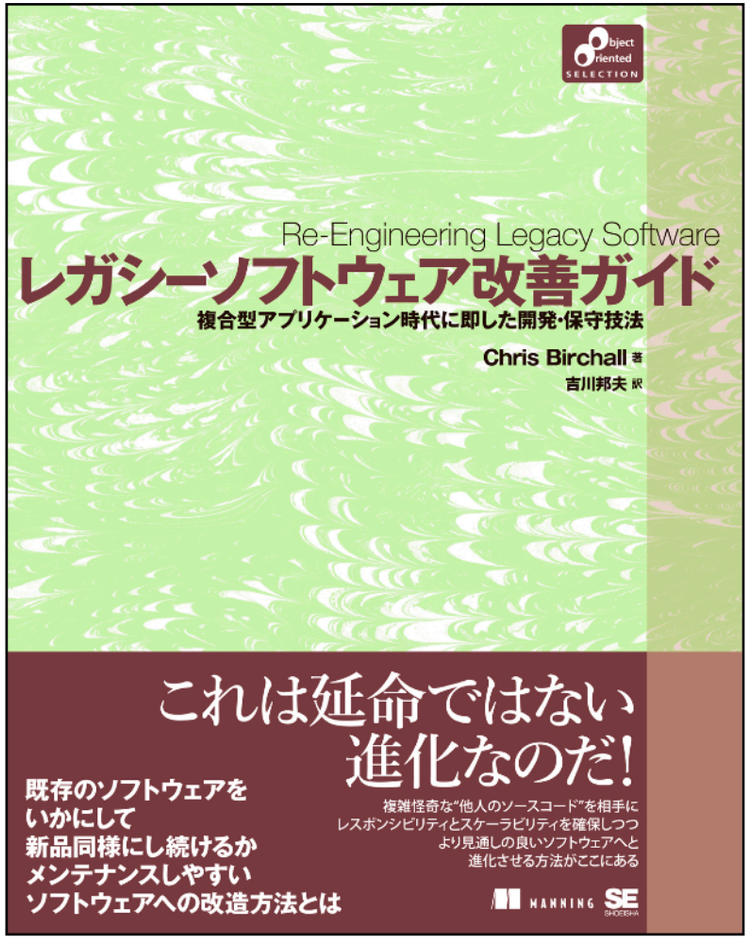
今日のお話

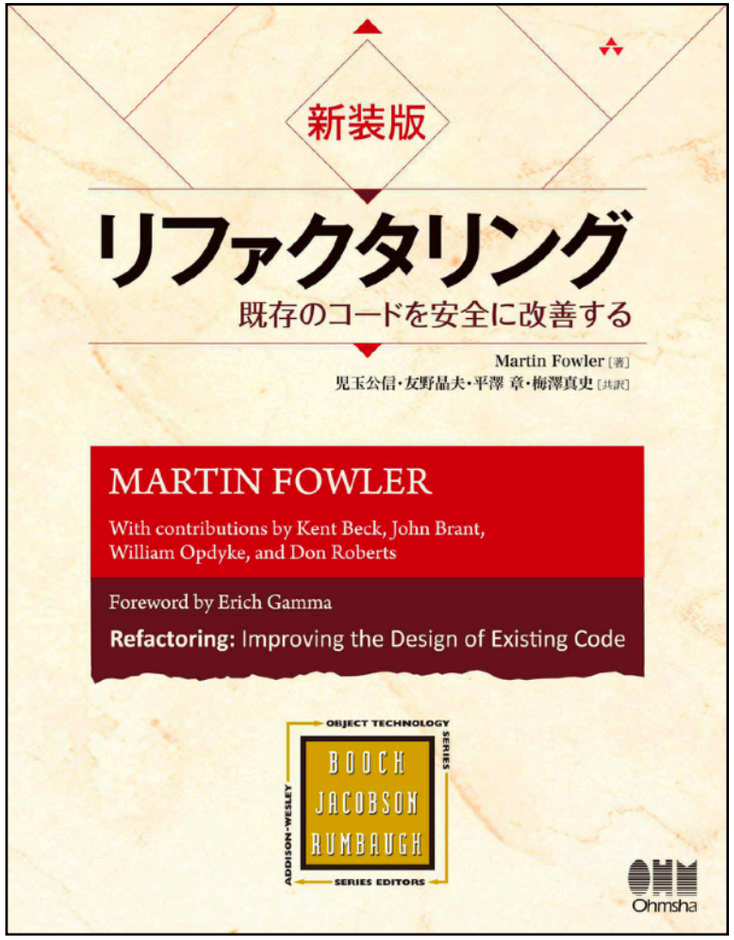
- レガシーコードからの脱却 ―ソフトウェアの寿命を延ばし価値を高める9つのプラクティス
- 2019/9/19 発売 3刷御礼

ITエンジニア本大賞 2020
技術書部門大賞!!
ありがとうございます！

レガシー本といえば


Working Effectively with Legacy Code
レガシーコード改善ガイド
保守開発のためのリファクタリング
マイケル・C・フェザーズ 著
ウルシステムズ株式会社 監訳
平澤章／越智典子／稲葉信之／田村友彦／小堀真義 訳
テストがないコードは
レガシーコードだ!
あなたも、
JavaやNETで
レガシーコードを
書いていませんか?
ソースコードがきれいで、良い構造であれば十分か?
——そうではない
もし、テストコードなしで大幅な修正を加えるとしたら
信じられないほどのスキルと明確な理解が必要になる
SE


Re-Engineering Legacy Software
レガシーソフトウェア改善ガイド
複合型アプリケーション時代に即した開発・保守技法
Chris Birchall 著
吉川邦夫 訳
これは延命ではない
進化なのだ!
既存のソフトウェアを
いかにして
新品同様にし続けるか
メンテナンスしやすい
ソフトウェアへの改造方法とは
複雑怪奇な"他人のソースコード"を相手に
レスポンシビリティとスケーラビリティを確保しつつ
より見通しの良いソフトウェアへと
進化させる方法がここにある
MANNING
SE


新装版
リファクタリング
既存のコードを安全に改善する
Martin Fowler [著]
児玉公信・友野晶夫・平澤 章・梅澤真史 [共訳]
MARTIN FOWLER
With contributions by Kent Beck, John Brant,
William Opdyke, and Don Roberts
Foreword by Erich Gamma
Refactoring: Improving the Design of Existing Code
OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS
Ohmsha

レガシーコードとは?

- 『レガシーコード改善ガイド』による定義
  - **レガシーコードとはテストのないコード**
  - 「テストのないコードは悪いコードである。どれだけうまく書かれているかは関係ない。どれだけ美しいか、オブジェクト指向か、きちんとカプセル化されているかは関係ない。テストがあれば、検証しながらコードの動きを素早く変更することができる。テストがなければ、コードが良くなっているか悪くなっているのかが本当にはわからない」
  - きれいなコードは有益だが、それだけでは不十分

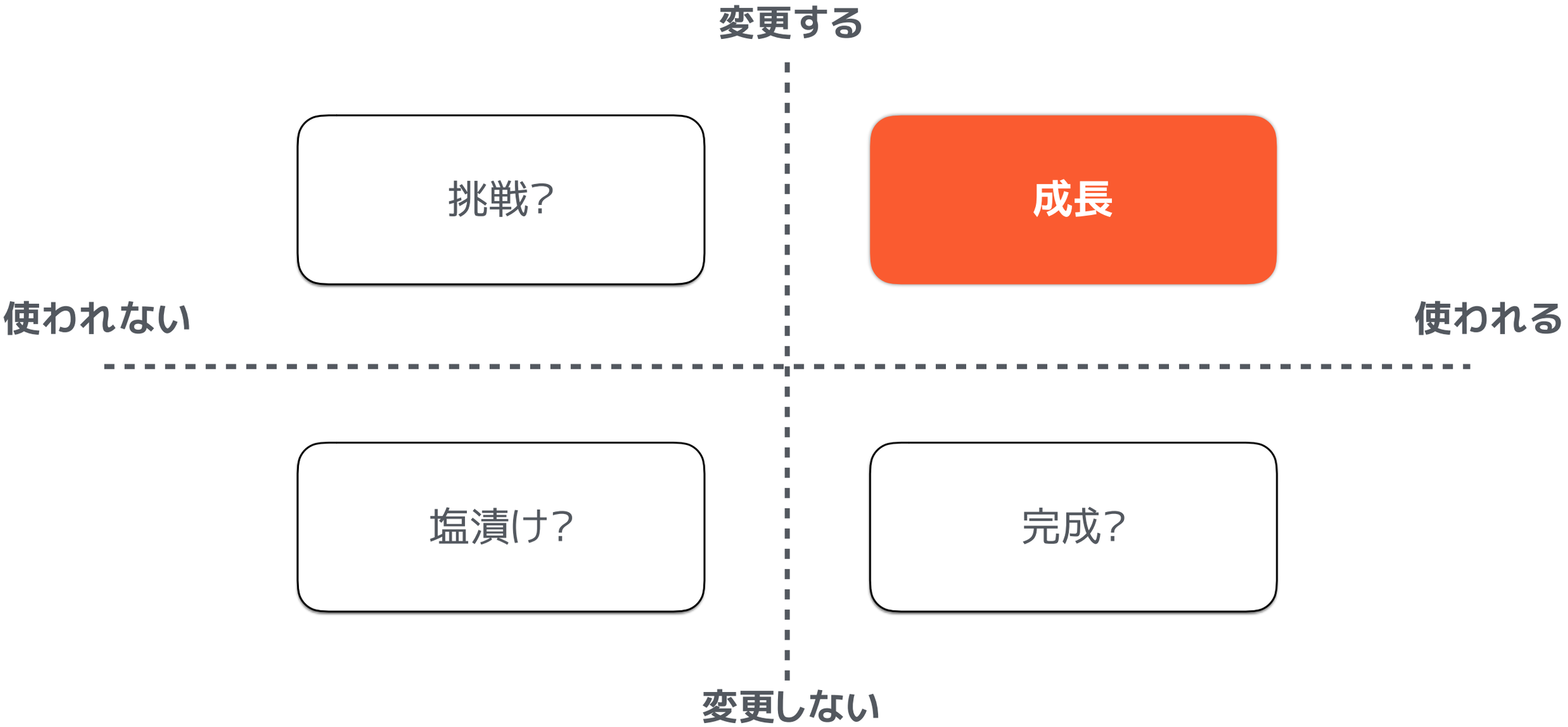
レガシーコードとは

- 『レガシーソフトウェア改善ガイド』による定義
  - **保守または拡張が困難なコード**
  - 「保守または拡張が困難な既存のプロジェクトなら、なんでもレガシーと呼ぶことにしている。ここでの話題は、単にコードベースだけではなく、プロジェクト全体であることに注意していただきたい」

レガシーコードとは

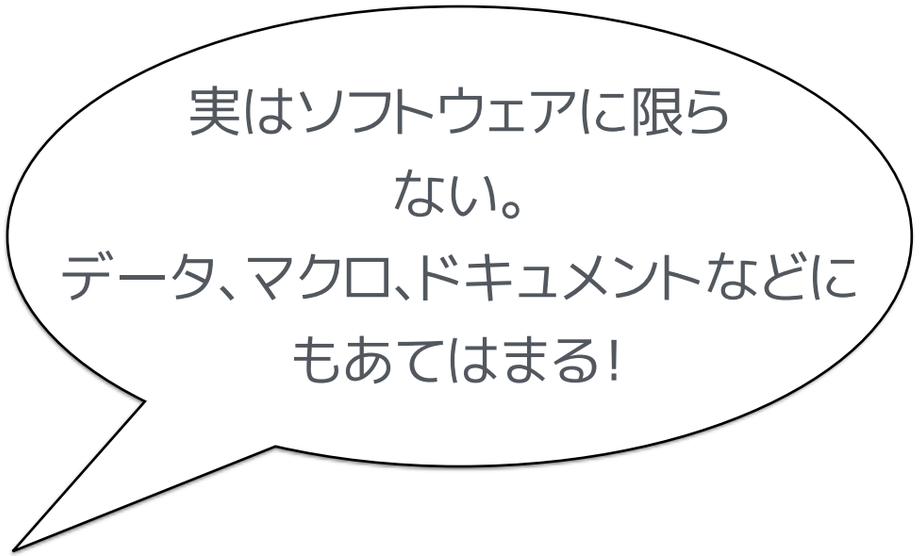
- つまり定義はさまざま
  - 同じ用語を使っていても、中身の認識に相違がでることが多い
  - 議論のときはまず**共通認識は何かを確認する**とよい
- 本セッションでは以下のように定義する
  - **理由は問わず修正、拡張、作業が難しいコード**
  - つまり保守に多額のお金がかかる

ソフトウェアと変更



使われるソフトウェア

- **使われるソフトウェアは変更が必要になる**
  - 必要な変更をすべて予測するのは無理
  - よって**変更可能となるように書くべき**
  - その逆がレガシーコード(修正、拡張、作業が難しいコード)
  - 必要になったときに対応できるエンジニアリングプラクティスが必要
  - 変更のコスト(ソフトウェアの保有コスト)を下げたい



ところが実際の時間の使い方には問題が…

『ソフトウェアがバグに悩まされる理由の1つは、増し続ける複雑さである。千行単位で計測されてきたソフトウェア製品のサイズは、今や百万行単位で計測されるようになっている。ソフトウェア開発者はすでに**ほぼ80%の開発コストを障害の特定と修正のために使っている**。それでも、ソフトウェアほど不具合が多い状態で出荷されている製品はほぼない。』

国立標準技術研究所（NIST）, 2002

ここにも10億ドル、あそこにも10億ドル

- ソフトウェアの障害が米国経済に年間600億ドルのコストになっている
  - 世界の180か国のうち、GDPが600億ドル以下の国は70%
  - マーク・ザッカーバーグとジェフ・ベゾスの総資産合計がおよそ600億ドル
- 失敗のコストは恐ろしく高い

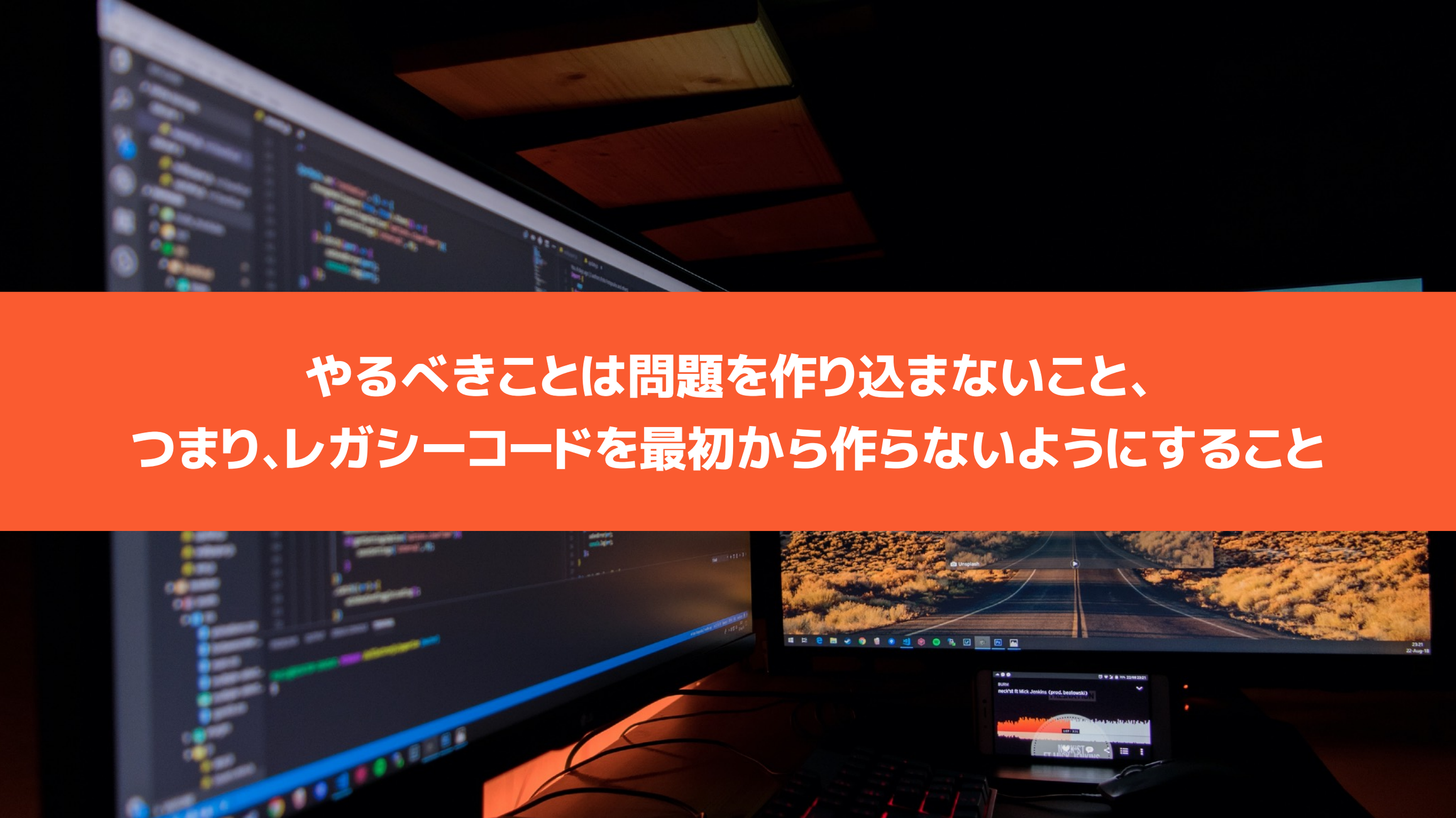
なぜこのような問題が起こるのか

- 急ぎすぎている
- たくさん作ろうとしすぎている
- 読みやすさより書きやすさを優先している
- ……
- つまり**品質を犠牲**にしている

品質は検査では上がらない

Quality comes not from inspection, but from improvement of the production process..

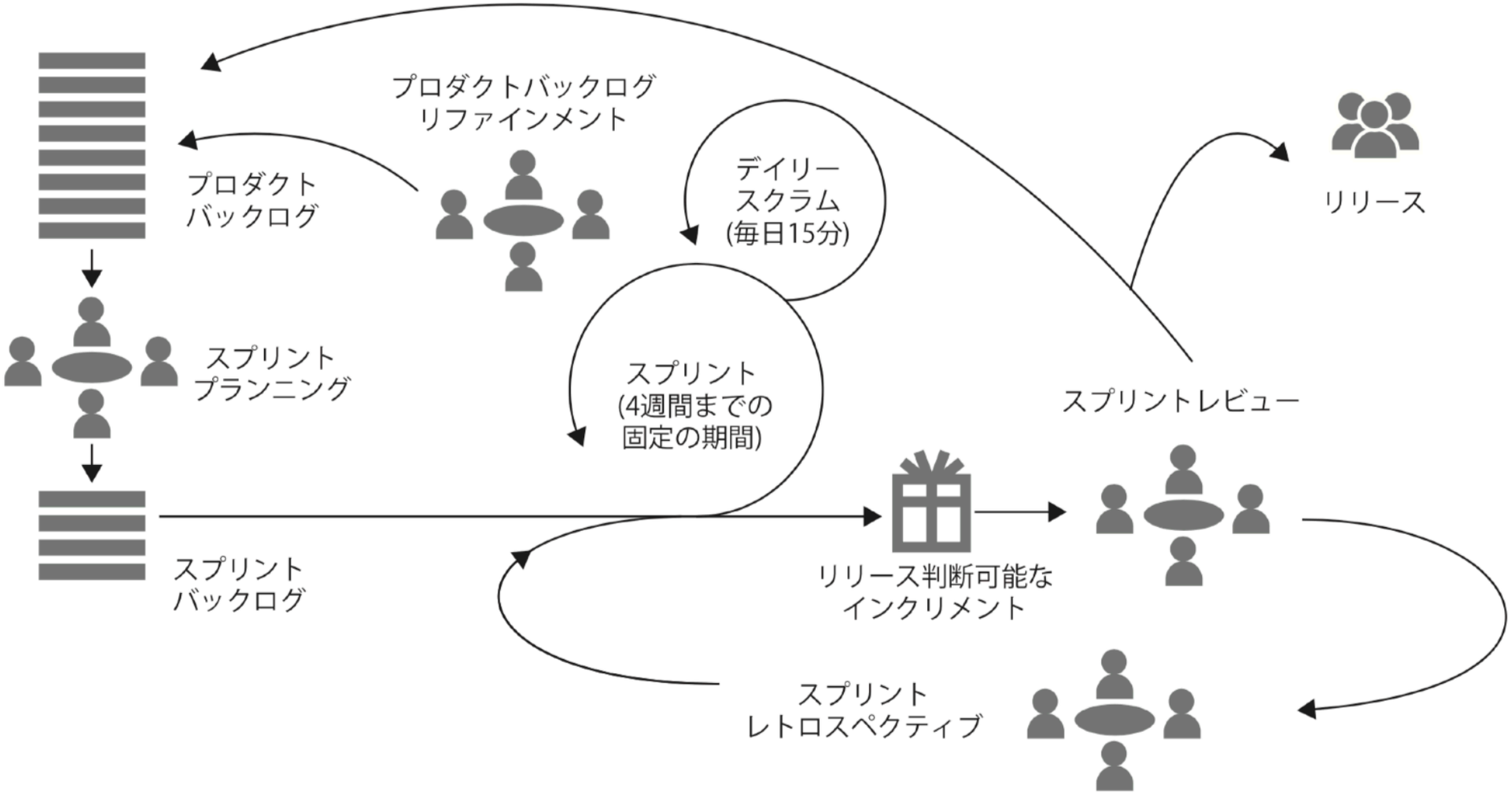
Edwards Deming



やるべきことは問題を作り込まないこと、
つまり、レガシーコードを最初から作らないようにすること

ではどうすればいいのか？
→ 開発プロセスに着目する

Scrum


プロダクトバックログ
リファインメント
プロダクト
バックログ
デイリー
スクラム
(毎日15分)
リリース
スプリント
プランニング
スプリント
(4週間までの
固定の期間)
スプリントレビュー
スプリント
バックログ
リリース判断可能な
インクリメント
スプリント
レトロスペクティブ

エクストリームプログラミング(XP)

初版(1999)【12】	2nd Edition(2004)【24】		現在【19】	
	主要プラクティス (13)	**導出プラクティス (11)**	**共同のプラクティス (4)**	**管理者のプラクティス (5)**
計画ゲーム	全員同席	本物の顧客参加	イテレーション	責任の受け入れ
短期リリース	チーム全体	インクリメンタルなデプロイ	共通の用語	援護
メタファ(比喩)	情報満載のワークスペース	チームの継続	オープンなワークスペース	四半期ごとの見直し
シンプルな設計	いきいきとした仕事	チームの縮小	ふりかえり	ミラー
テスト	ペアプログラミング	根本原因分析		最適なペース
リファクタリング	ストーリー	コードの共有	**開発のプラクティス (6)**	
ペアプログラミング	週次サイクル	コードとテスト	テスト駆動開発	**顧客のプラクティス (4)**
共同所有	四半期サイクル	単一のコードベース	ペアプログラミング	ストーリー
継続した統合	ゆとり	デイリーデプロイ	リファクタリング	リリース計画
40時間労働	10分ビルド	交渉によるスコープ契約	ソースコードの共同所有	受け入れテスト
オンサイトのユーザ(顧客)	継続的インテグレーション	利用都度課金	継続的インテグレーション	短期リリース
コーディング規約	テストファーストプログラミング		YAGNI	
	インクリメンタルな設計			

ソフトウェアが生み出す成果を決める要素

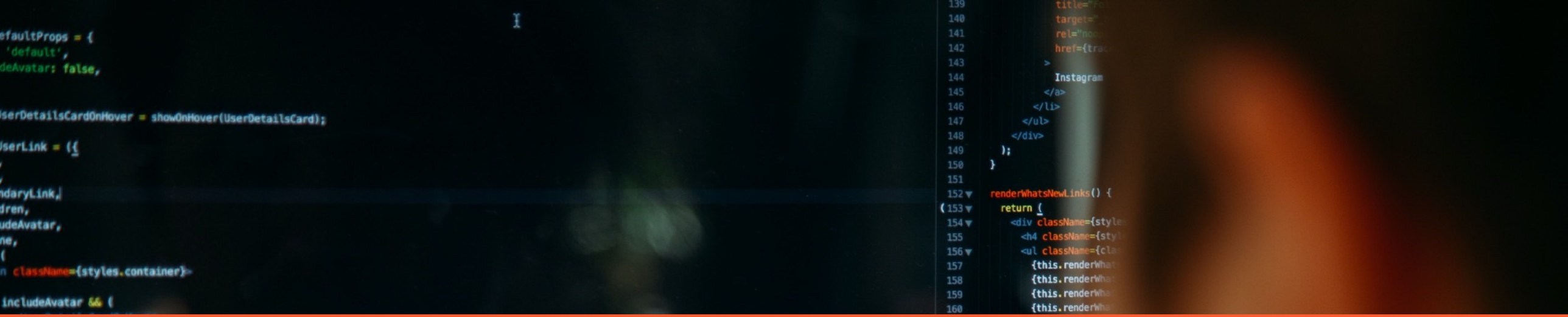
問題設定力 × **開発力** × **チーム力**

- 適切な問題設定
- 明確なビジョン
- マーケットの理解
- 取捨選択
- 良いプロダクトバックログ
- 優先順位付け
- ステークホルダー管理
- リスク･コスト管理
- …

- ドメイン知識
- アーキテクチャ設計力
- 開発言語
- 性能
- セキュリティ
- インフラストラクチャー
- 自動化
- 品質･テスト
- …

- 開発プロセス習熟
- 心理的安全性
- 透明性
- 検査と適応
- ムダをなくす
- 学習
- リーダーシップ
- オーナーシップ
- …

	XP	Scrum
共同のプラクティス	反復	スプリント
	共通の用語	(XPを活用)
	開けた作業空間	(XPを活用)
	頻繁なふりかえり	スプリントレトロスペクティブ
開発のプラクティス	テスト駆動開発	(XPを活用)
	ペアプログラミング	(XPを活用)
	リファクタリング	(XPを活用)
	ソースコードの共同所有	(XPを活用)
	継続的インテグレーション	(XPを活用)
	YAGNI	(XPを活用)
管理者のプラクティス	責任の受け入れ	コミットメント
	援護	スクラムマスター
	四半期ごとの見直し	スプリントレビュー
	ミラー	デイリースクラム、スクラムボード
	最適なペースの仕事	(XPを活用)
顧客のプラクティス	ストーリーの作成	プロダクトバックログ
	リリース計画	スプリントプランニング
	受け入れテスト	受け入れ基準
	短期リリース	リリース判断可能なインクリメント



レガシーコードを最初から作らないようにするには
具体的にはどうすればよいか？

レガシーコードを作らない9つのプラクティス

1. やり方より先に目的、理由、誰のためかを伝える
2. 小さなバッチで作る
3. 継続的に統合する
4. 協力しあう
5. CLEANコードを作る
6. まずテストを書く
7. テストでふるまいを例示する
8. 設計は最後に行う
9. レガシーコードをリファクタリングする

ScrumとXPからプラクティスを抽出。
全部やらなければいけないわけではないが
各項目は相互に関係性がある

- 今日はそのうちいくつかを紹介します

1

やり方より先に目的、理由、誰のためかを伝える

役割の違い

- プロダクトオーナーの責任
  - ビジネス価値を最大化する
  - プロダクトビジョンを周りに理解させる
  - プロダクトの結果責任
  - プロダクトバックログの順番の最終決定
  - ステークホルダーをマネージする
  - 予算を管理する
  - リリース日を決める
  - 開発チームの成果物の受け入れ可否

など…

- 開発チームの責任
  - リリース判断可能なプロダクトを作る
  - 設計、開発、テストをする
  - 決められた品質を満たす
  - 合意したプロダクトバックログ項目を完成させるように最善を尽くす
  - プロダクトオーナーにフィードバックする
  - 集中して仕事に取り組む
  - 問題があればそれを明らかにする
  - 常に改善する

など…

WhatとHowを分離する

- 何をほしいか、なぜ欲しいか（What）は顧客やプロダクトオーナーの領域
- やり方は開発者の領域（How）
  - 物事のやり方は1つではなく、やり方ごとにトレードオフがある
  - やり方を明示されると選択や交渉の余地が減る
  - 結果として手続き的なコードになりがち
- 双方が創造的に協調することで、無駄な時間や機能が減る
- **作る上で重要なのは、まず「コンテキスト」を共有・理解すること**

ユーザーストーリー

- 「何が」「なんのために」「誰のために」存在するかを1文で表したもの
  - (例) 映画ファンとして、チケットをオンラインで購入したい。そうすれば劇場で列に並んで待つ必要がない

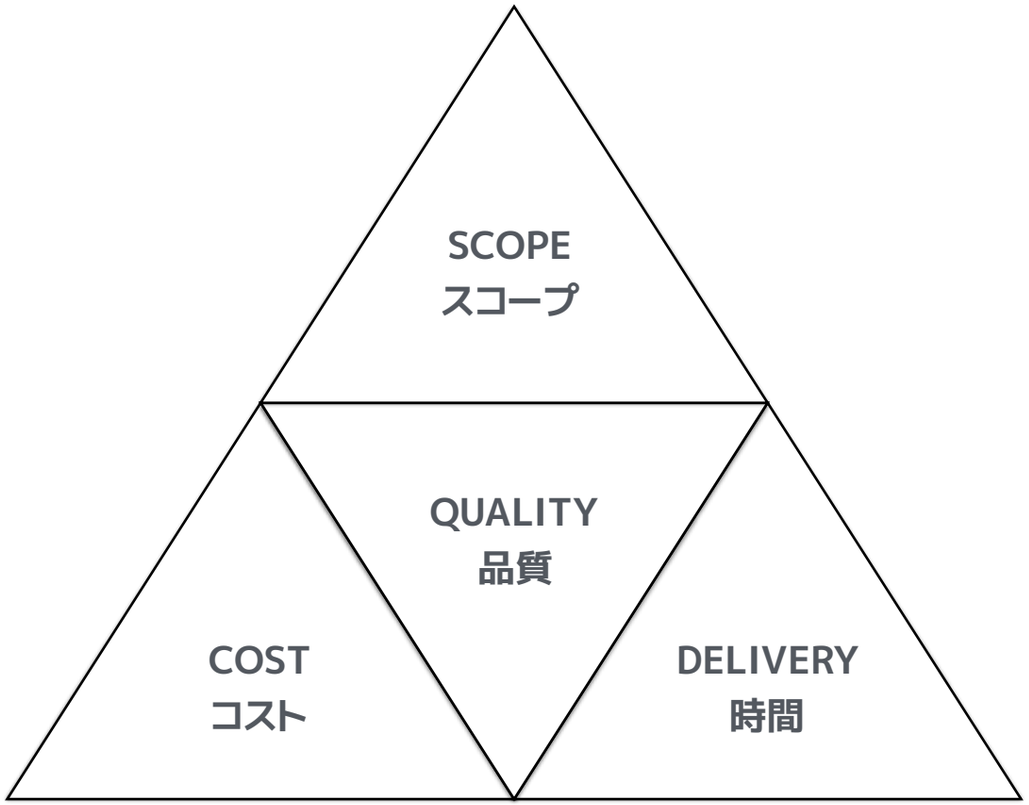
ユーザーストーリー

- 機能について会話できるくらいの辛うじて十分なドキュメント(仕様書ではない)
  - 会話によってソフトウェアを作るための理解を深める
  - 開発プロセスを円滑にするのは会話
- 知識は詳細なドキュメントではなく、コード(テストも含む)にまとめるべき
- ストーリーが**限定的であることで、テスト可能になる** (受け入れ基準と自動化)
  - 実例によるふるまいのテストが可能に
- シンプルに始めて追加はあとで行う (=> 2. 小さなバッチで作る)
  - **漸進的に進めることで、良い設計が浮かび上がってくる**

2

小さなバッチで作る

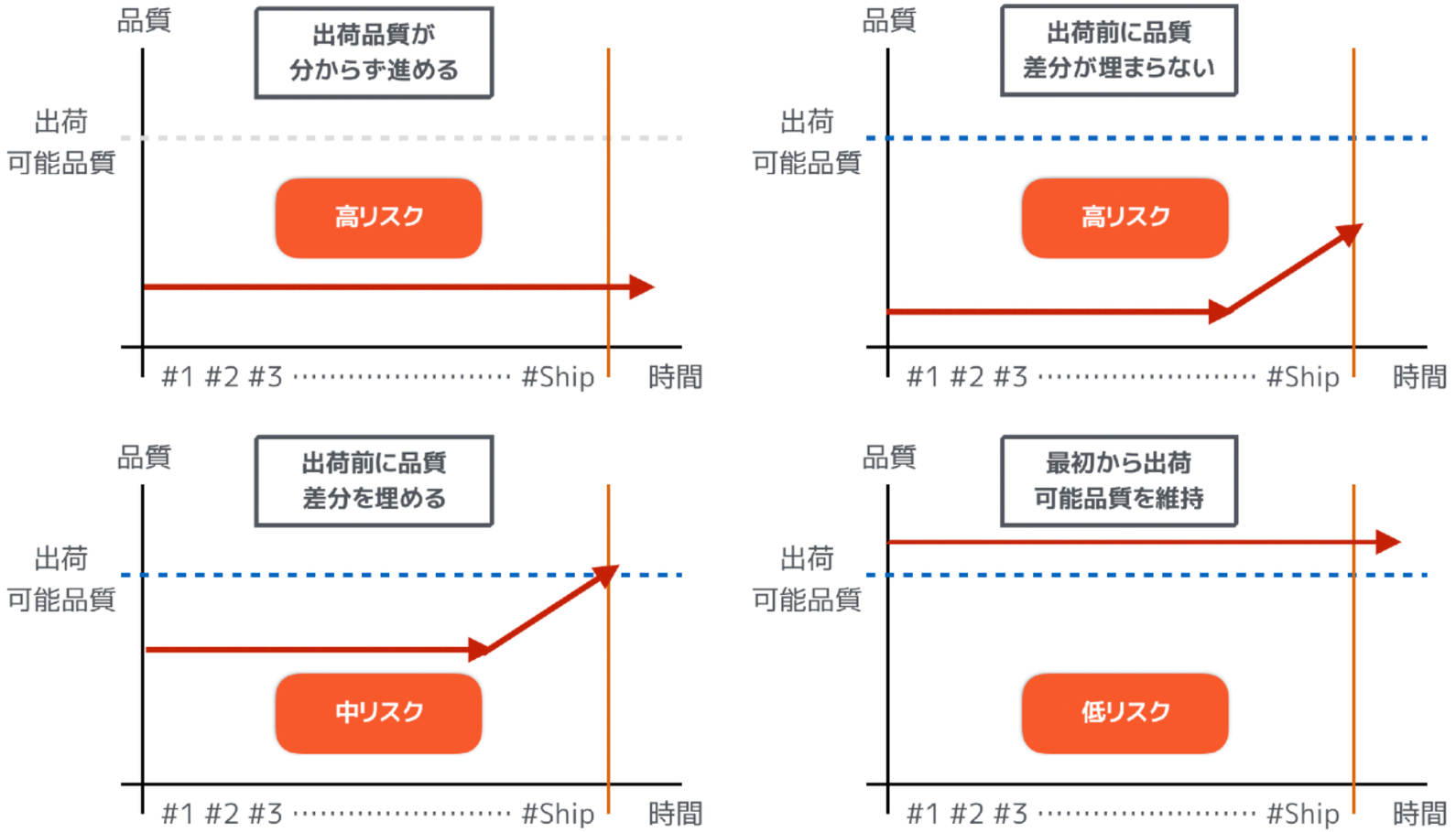
鉄の三角形



- 三角形の要素
  - Q => 品質 (Quality)
  - C => コスト (Cost)
  - D => 時間 (Delivery)
  - S => スコープ (Scope)
- すべてのものを同時には満たせない

QCDSの何を調整するか

- 品質(Q)を調整する(下げる)と、あとから手直しに何倍もの時間を使うことになる

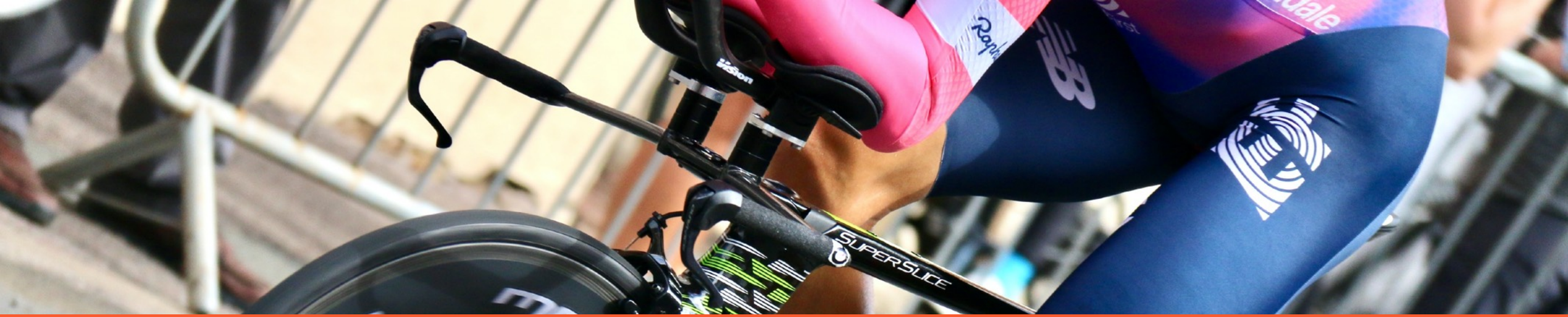


QCDSの何を調整するか

- コスト(C)を調整するとは、すなわち人を増やすことに繋がる
  - だが、人を増やしても、オーバーヘッドによって成果はリニアには増えない
  - 人月の神話「遅れているプロジェクトに人を追加すると、更に遅れる」
- 時間(D)を調整してもよいが、多くのビジネスは「時間」が重要
- つまり、**いちばん調整すべきなのは「スコープ」**
  - およそ半分の機能はほとんど使われない
  - スコープを調整して、価値あるものから順番に進める

タイムボックスによるアプローチ

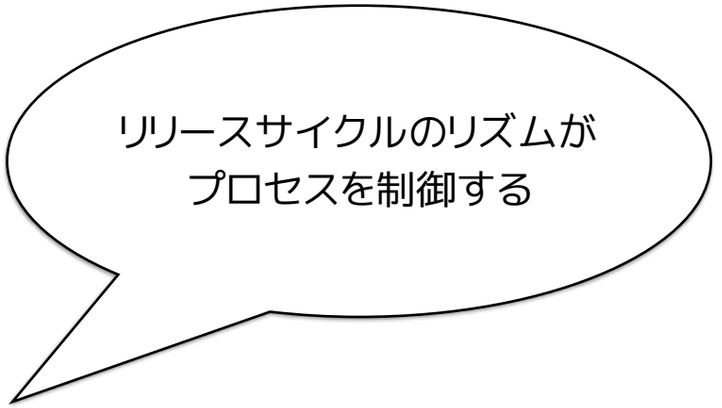
- タイムボックス
  - 固定の期間(リズム)の中でタスクに取り組む
  - タスクの分割になれるまでは機能しやすい
  - ScrumやXP
- 長い期間で大きなウソをつくのではなく、短いイテレーションで小さなウソをつく



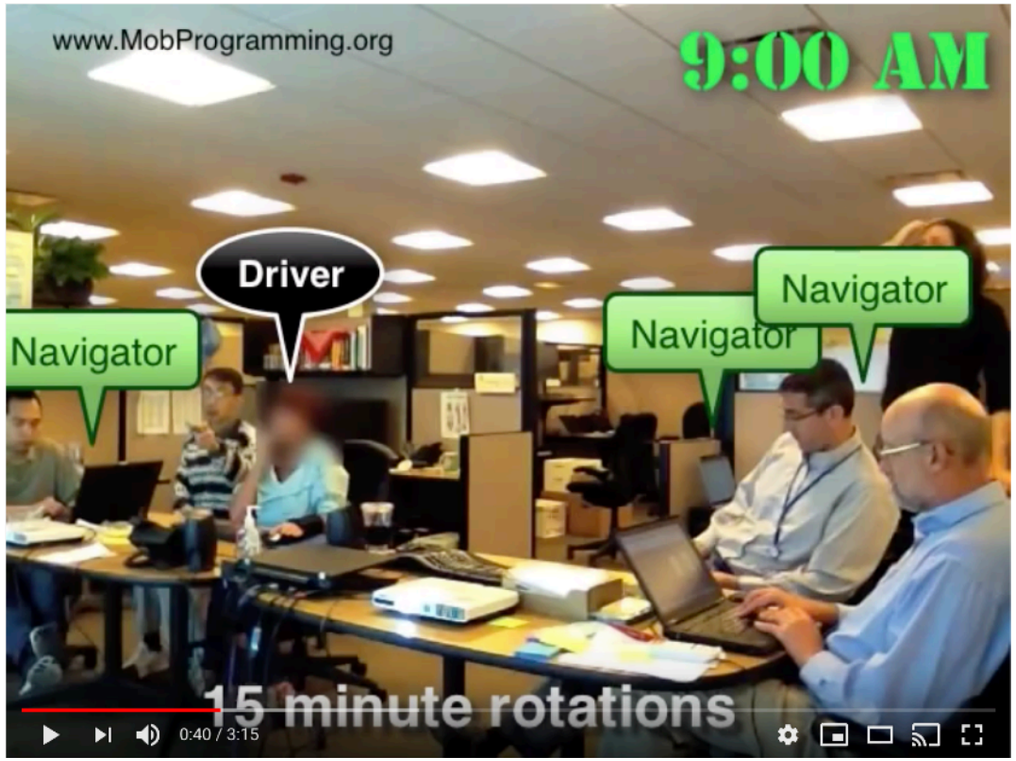
同じリズムを刻むことで負担を少なくして、異常に気づく
日々の仕事も同じ

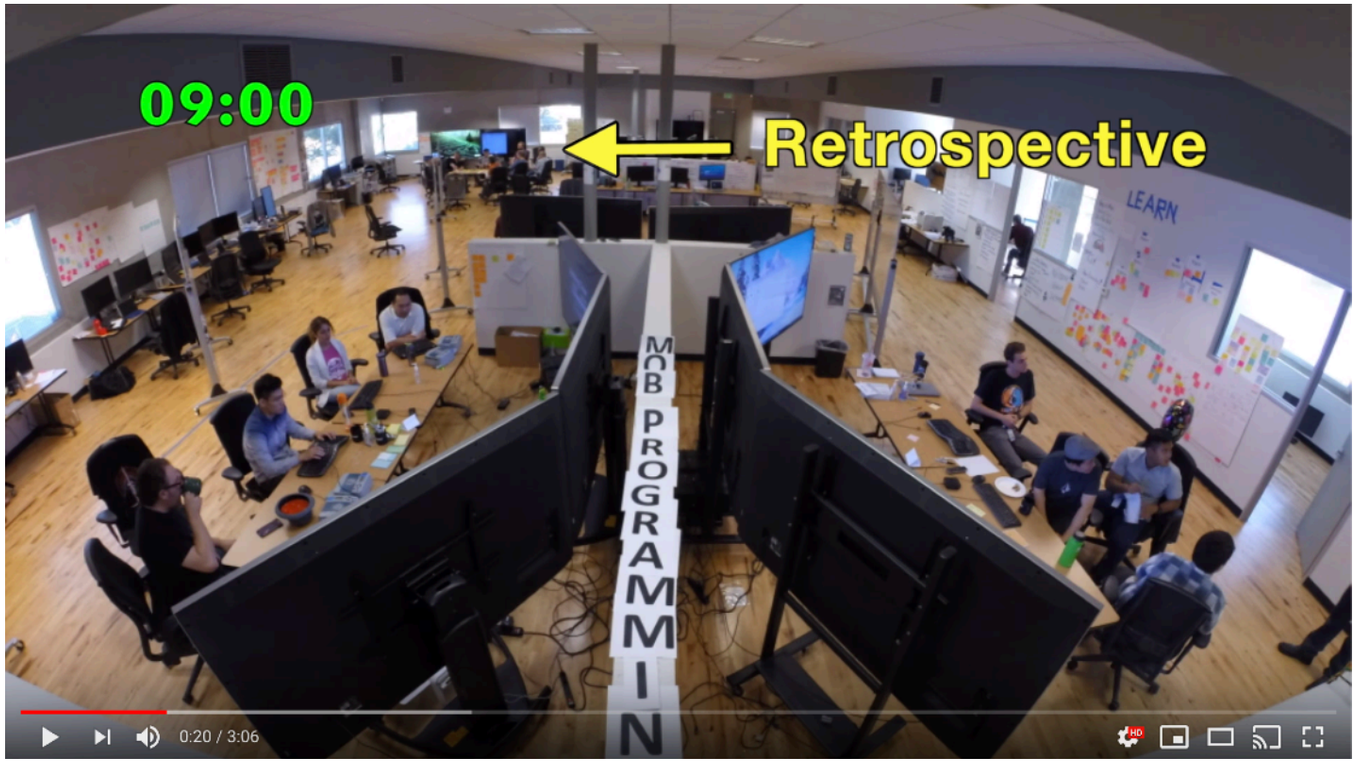
リソース効率とプロセス効率

- 1サイクルのリズムが長くなると、リソース効率化を目指しやすい
  - 同時に着手するものが増える / タスク切換えが増える
  - 役割分担が増える
  - 最後にテストフェーズや統合フェーズが増える
- 1サイクルのリズムを短くすると、プロセス効率があがる
  - 同時に着手するものを減らす必要(つまり1個流しのようなもの)



モブプログラミング = 究極の1個流し





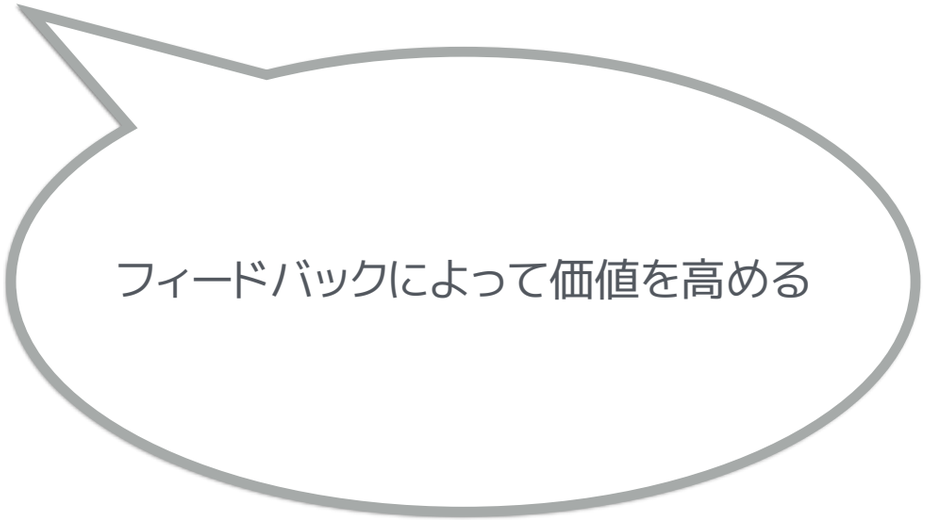
https://www.youtube.com/watch?v=p_pvsIS4gEI
https://www.youtube.com/watch?v=dVqUcNKVbYg

ソフトウェアの評価

- ソフトウェアの評価は、顧客にとって価値あるものにもとづいて評価すべき
  - 価値は「完成」して初めて価値になる
  - 本当に価値が実現できているか？
  - 価値実現までの時間が期待したとおりか？
  - 部分最適化を避ける
- **効率より効果**

フィードバックサイクル

- **小さなバッチのほうがフィードバックの回数は増える**
  - スプリントレビュー、レトロスペクティブ、継続的統合…
  - 顧客やPOと同席することで、フィードバックの回数を増やす
  - ビルドを高速化する
- フィードバックへの対応をサポートする文化が必要
  - 価値やリスクに応じて取捨選択する
  - フィードバックをバックログに取り込む



4

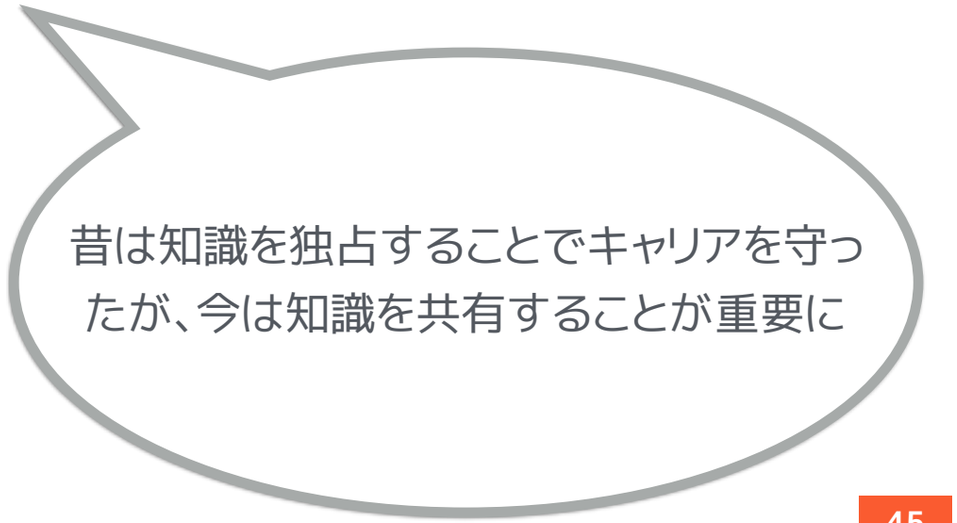
協力しあう

コミュニケーションと相互作用

- ソフトウェア開発は社会的な活動で、
  多くの**コミュニケーションと相互作用**を伴う
- 一緒に働くことでお互いから学ぶ（継続的学習）
- そのためにはオープンな空間が必要

ペアプログラミング

- 共通認識の形成（私だけのコード、私だけの秘伝の手順などをなくす）
- 良いプラクティスに従いながらお互い学び合う。新人の速度があがる
  - 常にメンターであれ、常にメンティーであれ
- 多少時間がかかっても、複雑な問題を解決できるし、不具合も減少する
- 割り込みの削減
- 変化型
  - バディプログラミングやモブプログラミング
- **効率より効果**



ペアプログラミング

- 共通認識の形成（私だけのコード、私だけの秘伝の手順などをなくす）
- 良いプラクティスに従いながらお互い学び合う。新人の速度があがる

プログラミングだけに限らず、すべての作業でペアは有効!!

- 変化型
  - バディプログラミングやモブプログラミング
- **効率より効果**

昔は知識を独占することでキャリアを守ったが、今は知識を共有することが重要に

ペアプログラミングの7つの戦略

- **とりあえず1度試しにやってみる**
- ドライバー(キーボードを操作する人)とナビゲーター(全体を見通す人)が参加する
- 役割は頻繁に交代する
- エネルギーを十分に保つ
- すべてのペアの組み合わせを試してみる
- 詳細はチームで決める
- 成果を測定してみる

レトロスペクティブ(ふりかえり)の7つの戦略

- 小さな改善を繰り返す
- 人を責めずに、プロセスを責める
- なぜなぜ5回(5 Why)
- 根本原因を直す
- 全員の意見を聞く
- 改善に必要な権限を与える
- 進ちょくを測る

5

CLEANコードを作る

品質は検査では上がらない（再掲）

Quality comes not from inspection, but from improvement of the production process..

Edwards Deming

よくないコードの特徴

- 重複したコード
- 長すぎるメソッド
- 巨大なクラス
- 長すぎるパラメータリスト
- 変更の偏り
- 変更の分散
- 特性の横恋慕
- データの群れ
- 基本データ型への執着
- スイッチ文
- パラレル継承
- 怠け者クラス
- 疑わしき一般化
- 一時的属性
- メッセージの連鎖
- 仲介人
- 不適切な関係
- クラスのインタフェース不一致
- 未熟なクラスライブラリ
- データクラス
- 相続拒否
- コメント

『新装版 リファクタリング―既存のコードを安全に改善する』より

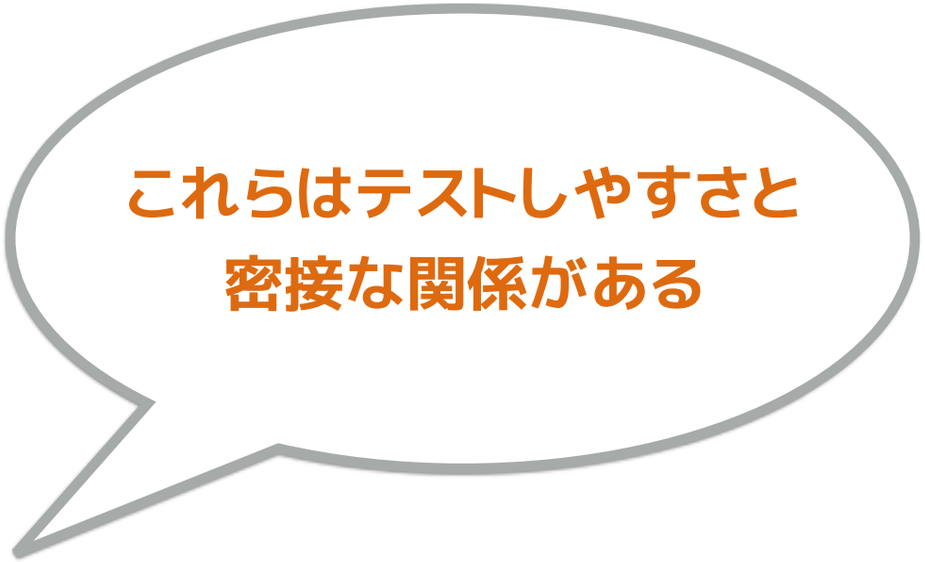
- よくない例をあげると数限りなし…
    - チームで理解して、よくない例を避ける必要はある
- 一方で、**従うべきガイドラインも必要**

CLEANコードとは?

- よいソフトウェアの土台となる5つのコード品質のこと
- 「オブジェクトは、特性が明確に定義されていて、はっきりした責務を担い、実装は隠ぺいされているべきだ。オブジェクトの状態は自分自身が管理し、オブジェクトの定義は一度だけにすべきだ。」

CLEANコードとは?

- Cohesive (凝集性)
- Loosely Coupled (疎結合)
- Encapsulated (カプセル化)
- Assertive (断定的)
- Non redundant (非冗長)



Cohesive(凝集性)

- それぞれの部品は1つのことだけを扱う
- クラスが1つの責任に集中する
- つまり名前をつけられるアイデアや概念になる
  - **名前重要**
- 複雑なものはコンポジション
  - 概念をネストする

Loosely Coupled(疎結合)

- オブジェクト間の関係を明確な意図をもった状態に保つ
- サービスを直接呼び出すのではなく中間層を経由する
  - コードにつなぎ目を入れておく
- 全てが悪なわけではない
  - 意図的な結合と不慮の結合
  - 不慮の結合はコード品質が低いときによく現れる
- なんでもできる神APIを避ける
- **再利用という名目のもとにコードの品質を犠牲にしてはいけない**

Encapsulated(カプセル化)

- 実装の詳細は外から見えなくなっている
- オブジェクト指向の最大の利点
- インターフェースと実装を切り離す
- アウトサイドインプログラミング
  - **コンシューマー(呼び出し側)の観点で機能を設計する**
  - **何をやっているかを示す名前をつけて、どう動くかは隠す**
  - 全体と詳細を行き来するが、まずは全体から始める
- 公開しているものを隠すより、非公開のものを後から公開するほうが簡単

Assertive(断定的)

- 自分自身の責任は自分で管理する
- オブジェクトがフィールドなどを持つなら、それらを管理するふるまいも持つ
- **好奇心旺盛すぎてはいけない**
  - ほかのオブジェクトの状態を頻繁に参照するのを避ける
  - 特性の横恋慕、不適切な関係

Non redundant(非冗長)

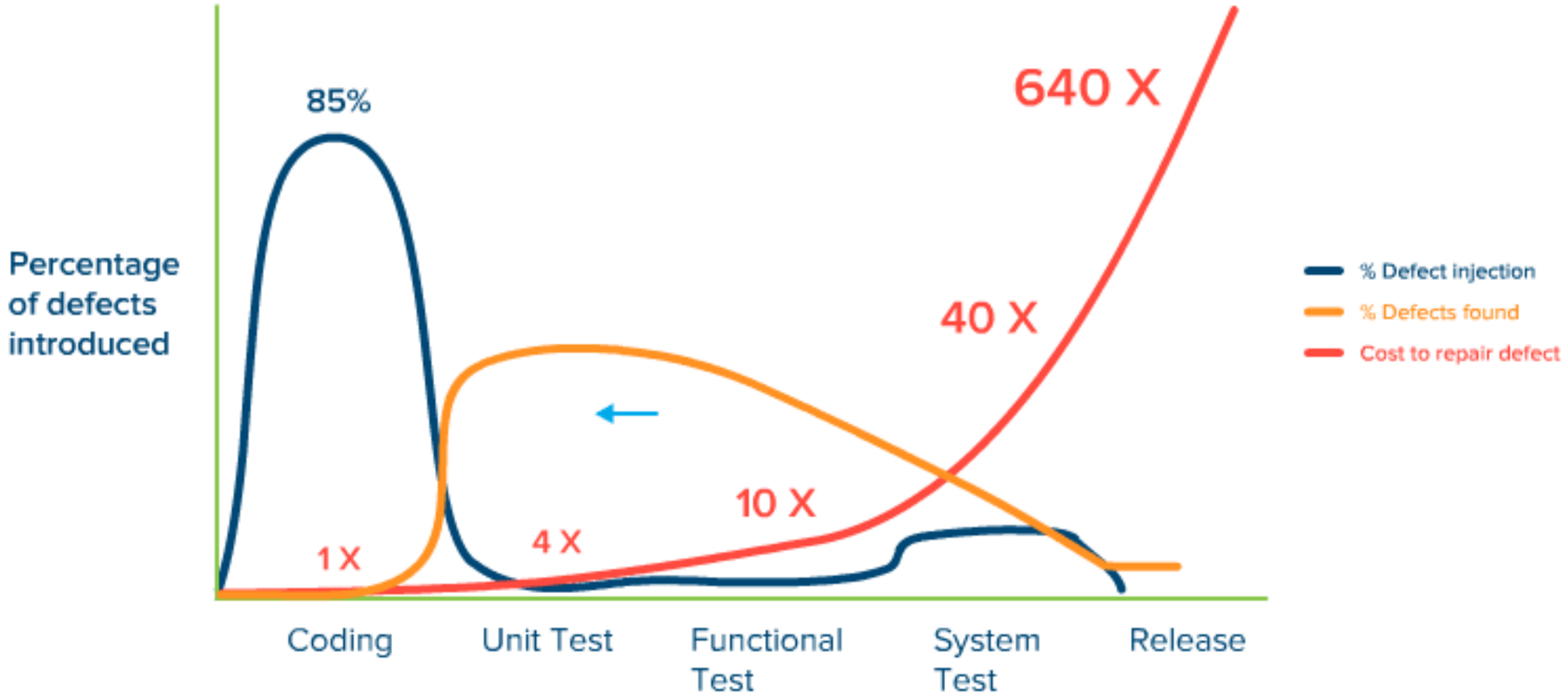
- **同じことを繰り返してはいけない(DRY原則)**
- 意図的に冗長性を組み込むことはあるが、それはあくまで意図による
- 冗長さは状態やふるまいだけに限らない
    - 冗長なテスト、冗長な概念、冗長な解釈、冗長なプロセス…
- コードが違うから冗長でないというわけではない

CLEANのメリット

テストしやすさが
設計や実装の品質を計測する
基準になる

- (C) 凝集性があれば、理解もバグを見つけるのも簡単
- (L) 結合度が低ければ、副作用が減り、テストや再利用、拡張が簡単
- (E) カプセル化されていれば、呼び出し元が実装の詳細を知らなくてもよいように維持できる。あとから変更するのも簡単
- (A) 断定的であることは、ふるまいを配置する場所が依存データがある場所であることを示す
- (N) 冗長でなければ、バグ修正や変更は1箇所で1回だけやればよい

CLEANとテストでバグ発見の時期をシフトレフトする



Capers Jones, Applied Software Measurement: Global Analysis of Productivity and Quality

明日のベロシティのために今日品質を上げる

- **速度は日々の積み重ねによってしか実現できない**
  - CLEANコードは理解しやすく取り組みやすい
  - 高品質のコードは拡張しやすい
  - 高品質のコードはデバッグしやすく保守しやすい
  - 結果的に総所有コストが下がる
  - 逆の状態が「技術的負債」


https://flickr.com/photos/joanbrebo/38646019434


https://flickr.com/photos/joanbrebo/38646019434

すばやく働くということは「きれいに働く」ということ

これって5Sだった

- 整理 => いらないものを捨てる
- 整頓 => 決められた物を決められた場所に置き、いつでも取り出せる状態にする
- 清掃 => 常に掃除をする
- 清潔 => 3S（上の整理・整頓・清掃）を維持し職場の衛生を保つ
- 躾　 => 決められたルール・手順を正しく守る習慣をつける

8

設計は最後に行う

ソフトウェアは書く回数より読む回数の方が多い

あくなき追求

- いかにバグを取るかではなく、いかに保守しやすいかという設計に注意を払う
  - 良いコードとは変更しやすいコード
- 変更しにくいコードをみつけて、それらを取り除いていく
  - カプセル化の欠如、継承の過度の利用、具体的すぎる実装、インラインコード、依存性、自身によるオブジェクトの新規生成
- **テスト**があれば安全にコードをクリーアップできる
- コードが動作して、テストのサポートがある状態から設計を良いものにする
- コーディングとクリーニングは分離する

持続可能な開発

死んだコードを消す	コメントアウトされたコード、呼び出されないコード、使わない機能は全部消す。存在しても注意が散漫するだけ
名前を更新する	メソッドやクラスの名前を意図のわかる良い名前に更新する。開発を進めて分かったことが増えると機能が変わる。今やっていることを反映する名前に
判断を集約する	判断を集約し、1 回だけで済むようにする。冗長なコードも削除できる
抽象化する	すべての外部依存性には抽象を作成し利用する。モデルに欠けているエンティィは作成する
クラスを整頓する	利用範囲においてモデ ルが完全であることを確認する。そうすることで、正しいふるまいと正しい属性を持つようにクラスを整頓できる

持続可能な開発

死んだコードを消す	コメントアウトされたコード、呼び出されないコード、使わない機能は全部消す。存在しても注意が散漫するだけ

ソフトウェア開発はあとから分かることが多い
それを随時反映していく

抽象化する	すべての外部依存性には抽象を作成し利用する。モデルに欠けているエンティィは作成する
クラスを整頓する	利用範囲においてモデ ルが完全であることを確認する。そうすることで、正しいふるまいと正しい属性を持つようにクラスを整頓できる

レガシーコードからの脱却 (まとめ)

- 使われるソフトウェアは変更が必要になるので、コードは変更可能になるように書くべき
- プロダクトオーナーは、目的、必要な理由、誰のためのものかが伝わるようにする
- 要求は意味のある小さな単位にして、重要なものから作る
- 小さなバッチで関係者が協力しながら作っていく
- コードはテストコードも含めてCLEANを維持し、常に品質を満たすように作っていく
- 必要に応じて設計を見直したりコードを見直したりする。良い設計は創発する


Attractor Inc.