Juichi Takahashi, Ph. D
CTSO of Digital Hearts

もうちょっと楽に単体テストしませんか？

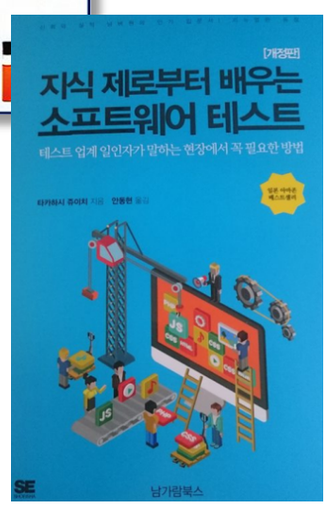
Books


SOFTWARE
PROJECT
MANAGEMENT
知識ゼロから学ぶ
ソフトウェアプロジェクト管理
SE


ソフトウェア品質を高める
開発者テスト
アジャイル時代の実践的・効率的なテストのやり方
◎著者＝情報工学博士 高橋 寿一
Developer's Test
BUG
ソフトウェア上流品質を上げまくって、
バグだらけ・死ぬほど働くのをやめませんか?
テスト界の第一人者による開発者テストの実践方法＋学術的根拠のエッセンス
単体テスト
リファクタリング
テストの自動化
アジャイル開発での品質担保
など、実例を交えてわかりやすく解説!
SE
SHOEISHA

テスト手法
高橋 寿一／瀬本 剛
技法を知っただけでは
テストはできない!
実践派を目指すあなたに贈る
知識ゼロから学ぶテスト管理術 虎の巻


知識ゼロから学ぶ
ソフトウェアテスト
【改訂版】
アジャイル・クラウド時代のソフトウェアテスト
SOFTWARE
TEST
信頼と実績ナンバーワンの
定番入門書! リニューアル版登場!
SE


[개정판]
지식 제로부터 배우는
소프트웨어 테스트
테스트 업계 입문자가 말하는 현장에서 꼭 필요한 방법
남가람북스

Who am I?

Microsoft US

↓

SAP Tokyo Lab

↓

Sony Co.

↓

Digital Hearts Co., CTSO

本日の主題

- Shift Left（単体テストで上流品質担保）しなければ、市場バグ流出はなくならない

- **正しく**Shift Leftすれば、あなたの仕事は圧倒的に減る

幻想

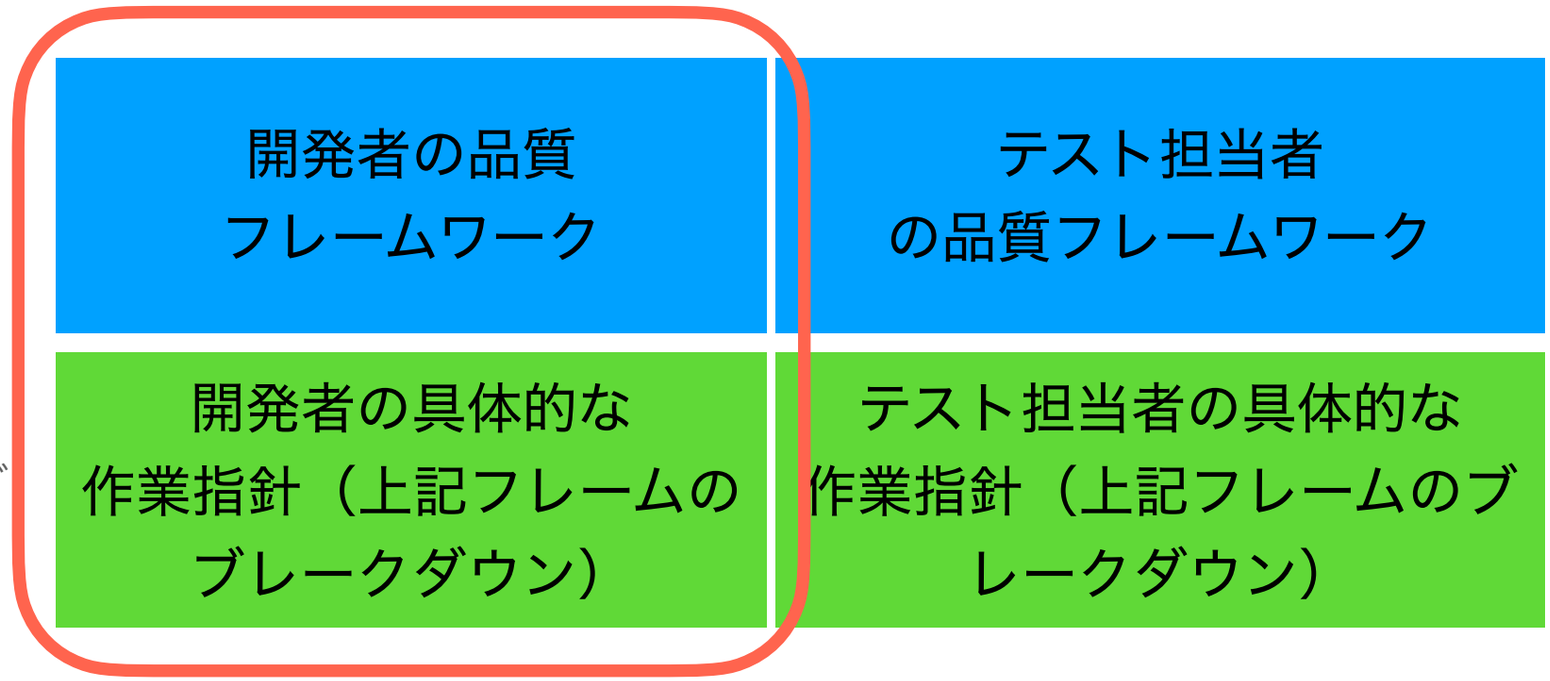
バグ数/LOC = 品質

Agile品質担保に必要な４つのBox

（ウォーターフォルでもAgileでも開発サイクルは短くなっている）

i.e)　各イテレーションでの アーキテクチャーはどうする？

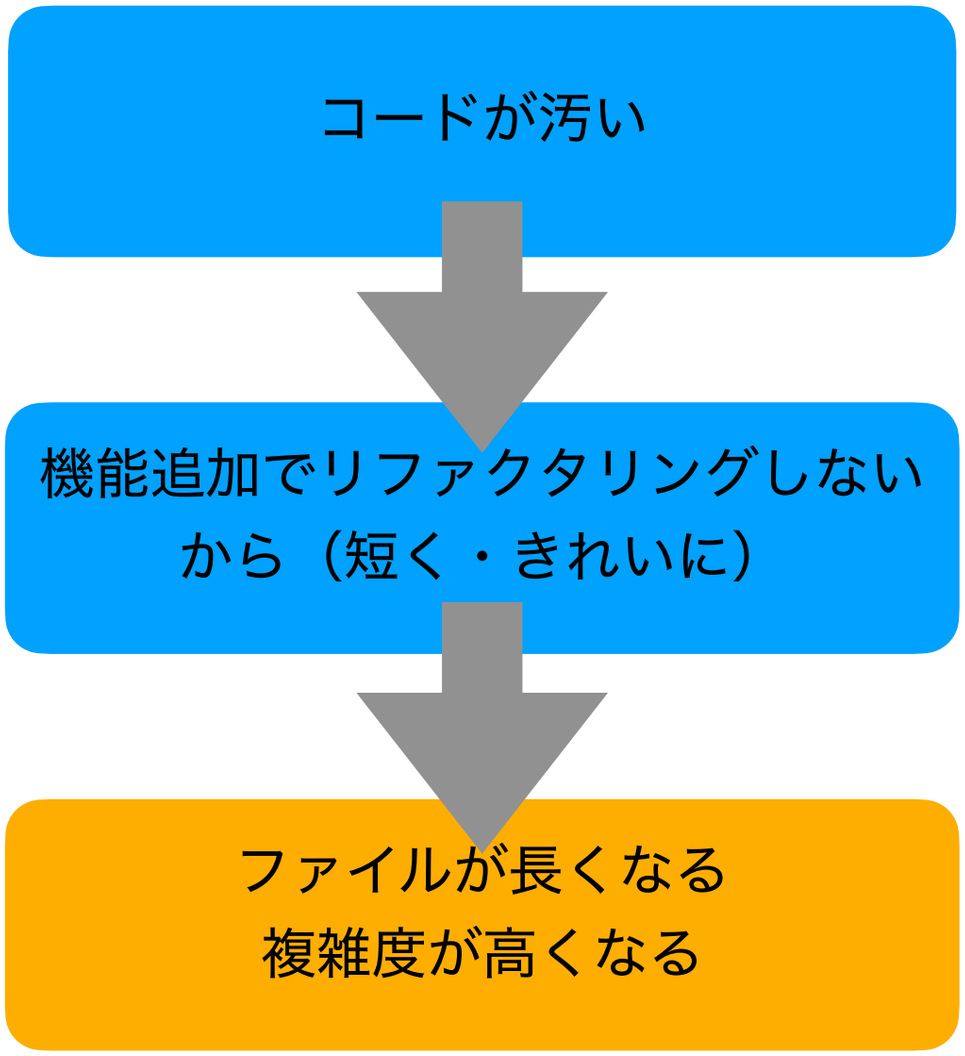
i.e) リファクタリング基準



i.e) Agile用のテスト計画書(IEEE829)的なもの

i.e) テストケースは書くの？
2wkで定量的な品質指標は？

寿一が勝手に想定する品質問題の本質
すべての日本の会社で



コードを書くスキルが十分ではない
コードを指導するマネージメントが少ない
単体テスト書くスキルがない

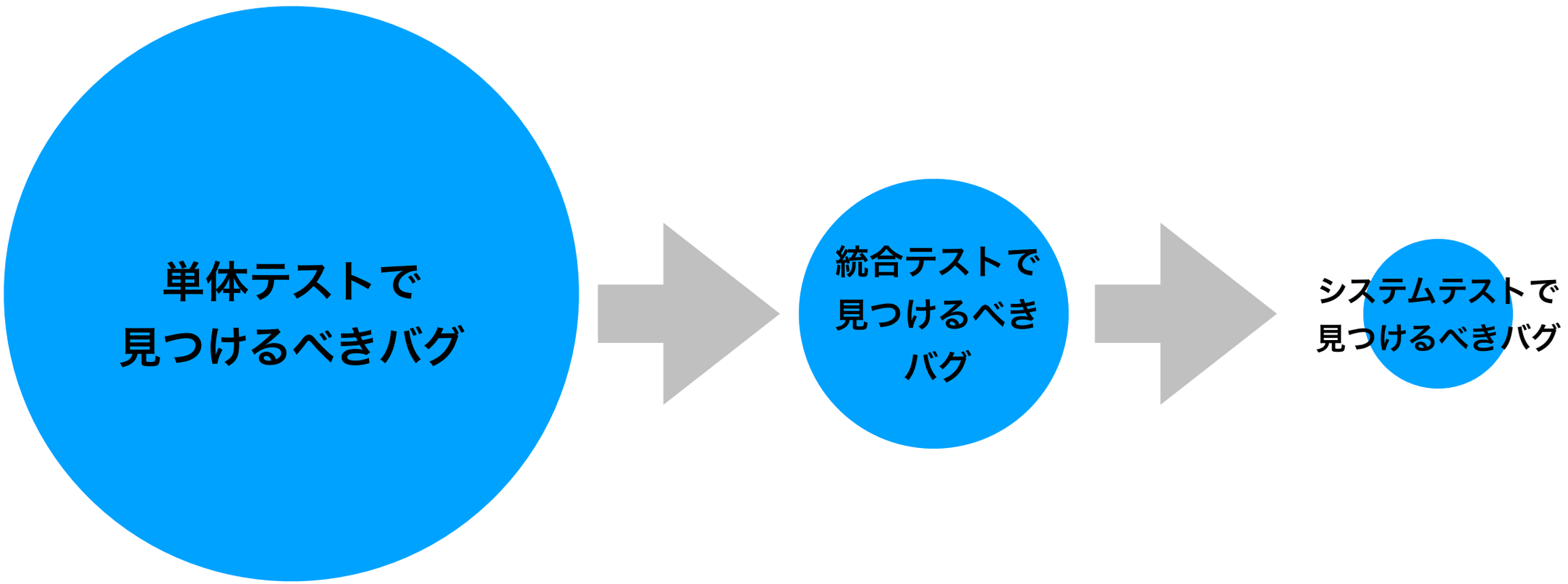
リファクタリングするスキルがない（Agile開発にとっては致命的）
リファクタリングに対するマネージメントの知識がない
リファクタリングしないので、単体テストができない（複雑度があがる）

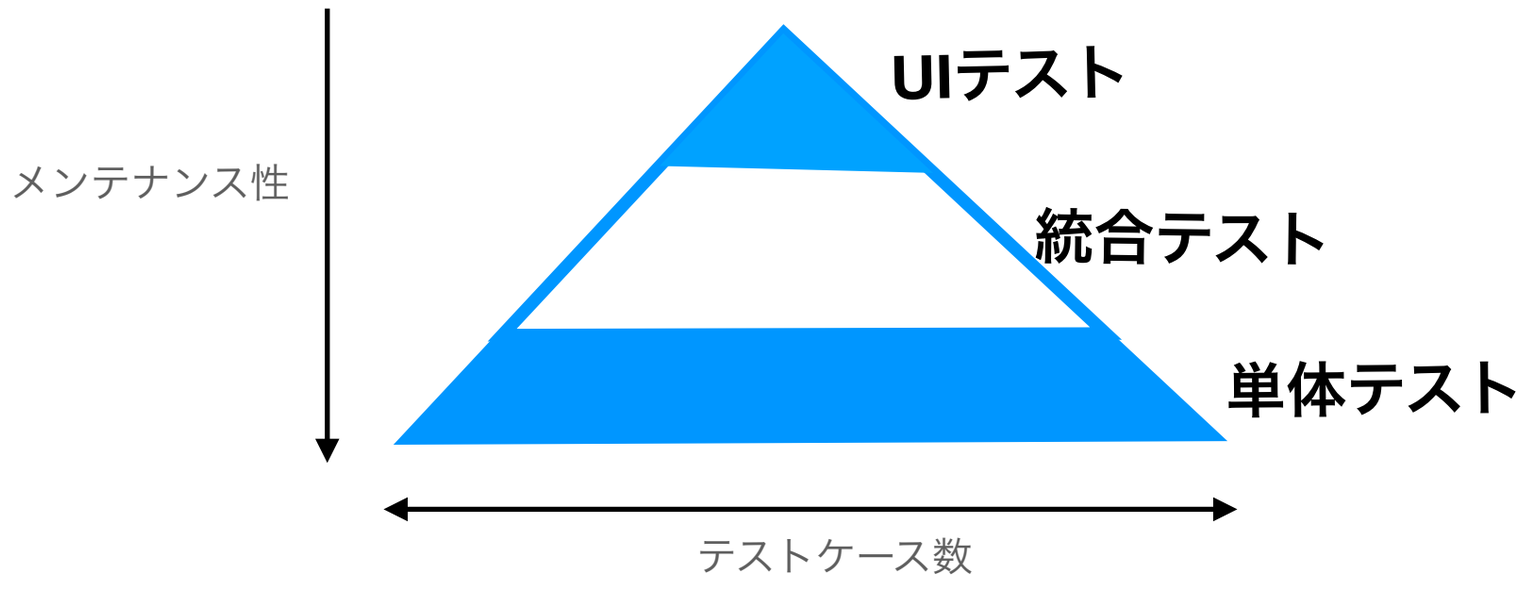
バグが多発する原因のTop1, 2（論文で証明済）

アジャイルではこの悪いサイクルが加速する？
ウォーターフォル開発より時間と費用のかかるアジャイル？

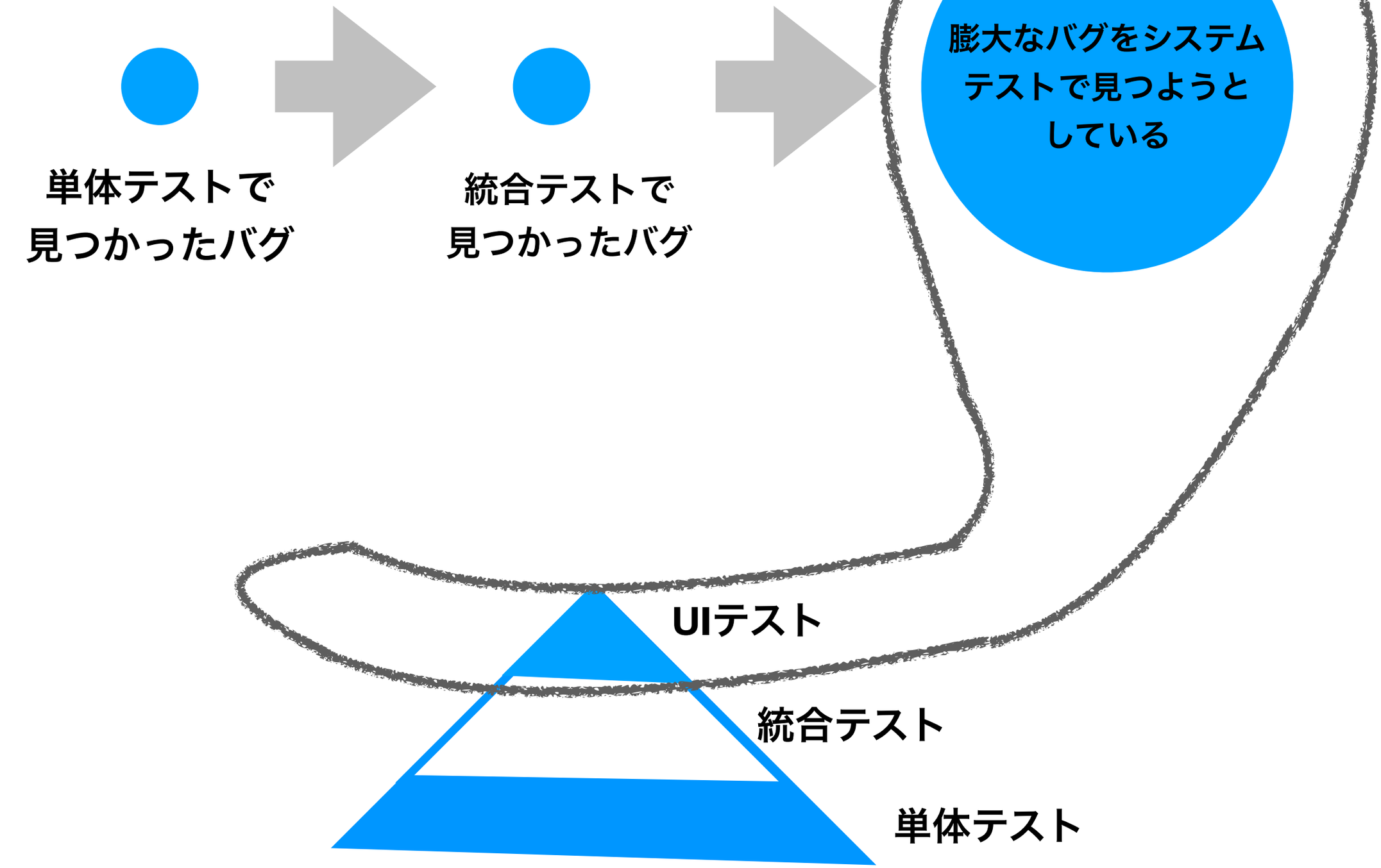
Googleでは

良い自動化


単体テストで
見つけるべきバグ
統合テストで
見つけるべき
バグ
システムテストで
見つけるべきバグ


メンテナンス性
UIテスト
統合テスト
単体テスト
テストケース数

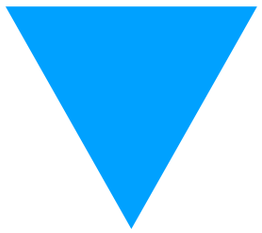
悪い自動化


単体テストで
見つかったバグ
統合テストで
見つかったバグ
膨大なバグをシステム
テストで見つようと
している
UIテスト
統合テスト
単体テスト

私がxxx時代の
ソフトウェア品質の課題

- コードが汚く、単体テストができない（やる気がおきない）。統括もエンジニアも見て見ぬ振り

- コードが汚く、そして長いので単体テスト（コンポーネントレベルの品質担保）が戦略的にできないので、システムとして品質保証できない



リファクタリング！リファクタリング！
自動単体テスト！自動単体テスト！

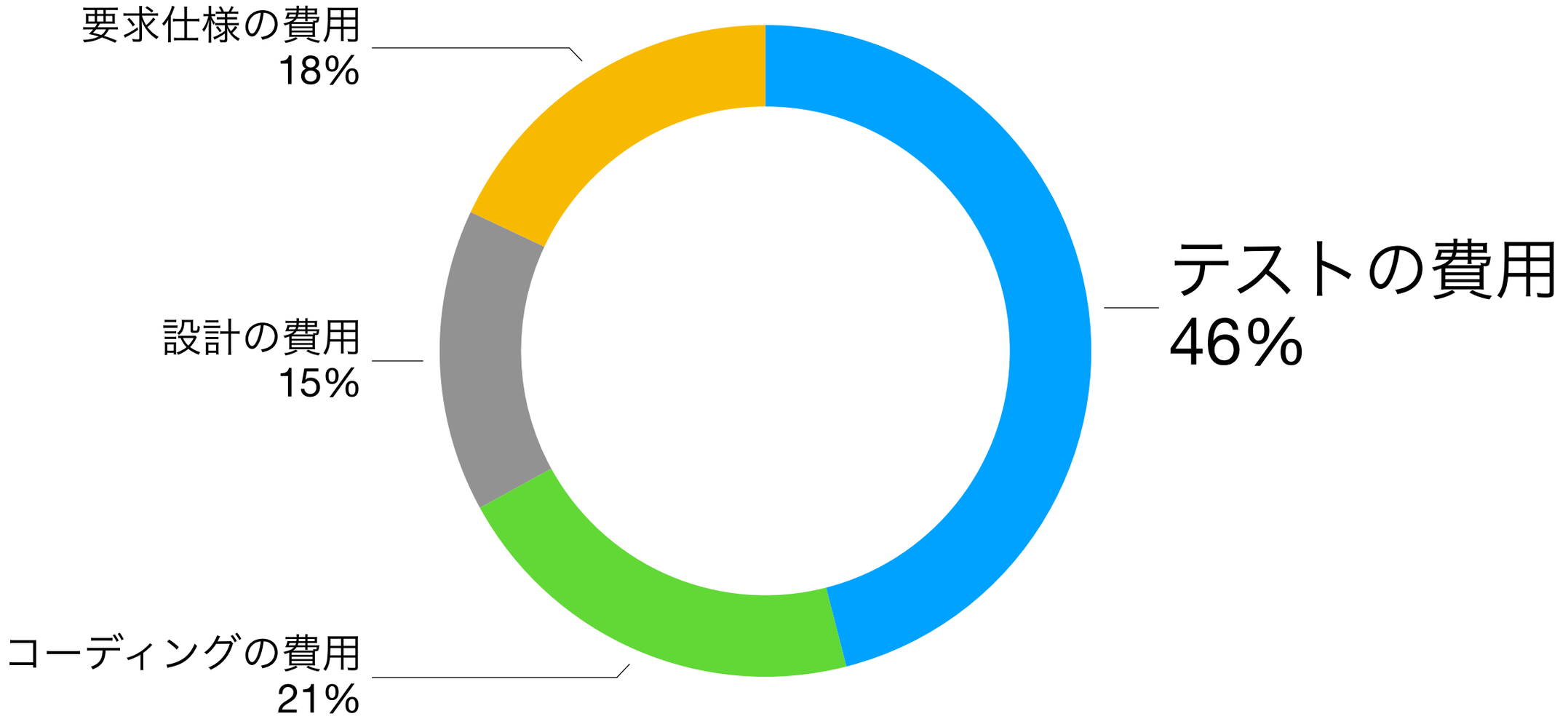
今日話す品質の定義

- バグはすべて取ることができない

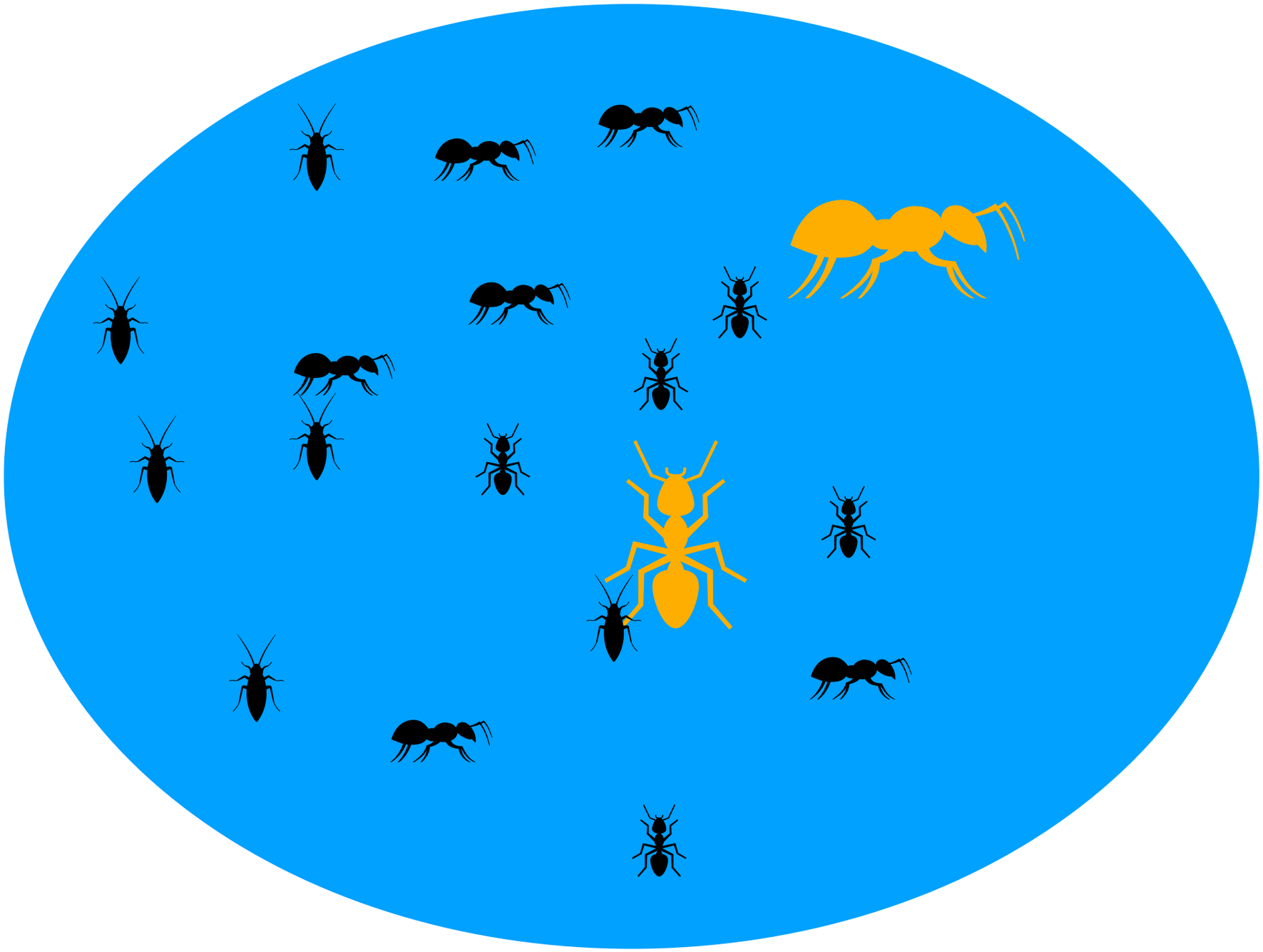
- 有限な時間の中で最適品質を探す、最適品質とは

  - **膨大なシステムテストは無意味。上流で効率的なテストを行う**
  - 品質はビジネス。早いタイミングでユーザの求めるソフトウェアを出すことが重要。要求仕様が変更され、コードが数十行変更されても、その日にリリースできる仕組みが必要
  - 最大公約数的なユーザ操作において、致命的なバグを起こさない（Operational Profile）
  - バグったとしても、ソフトウェア全体がいきなり死なない（ウオッチドッグ・Netflix的アプローチ）

品質にかける費用はここ３０年変わってない



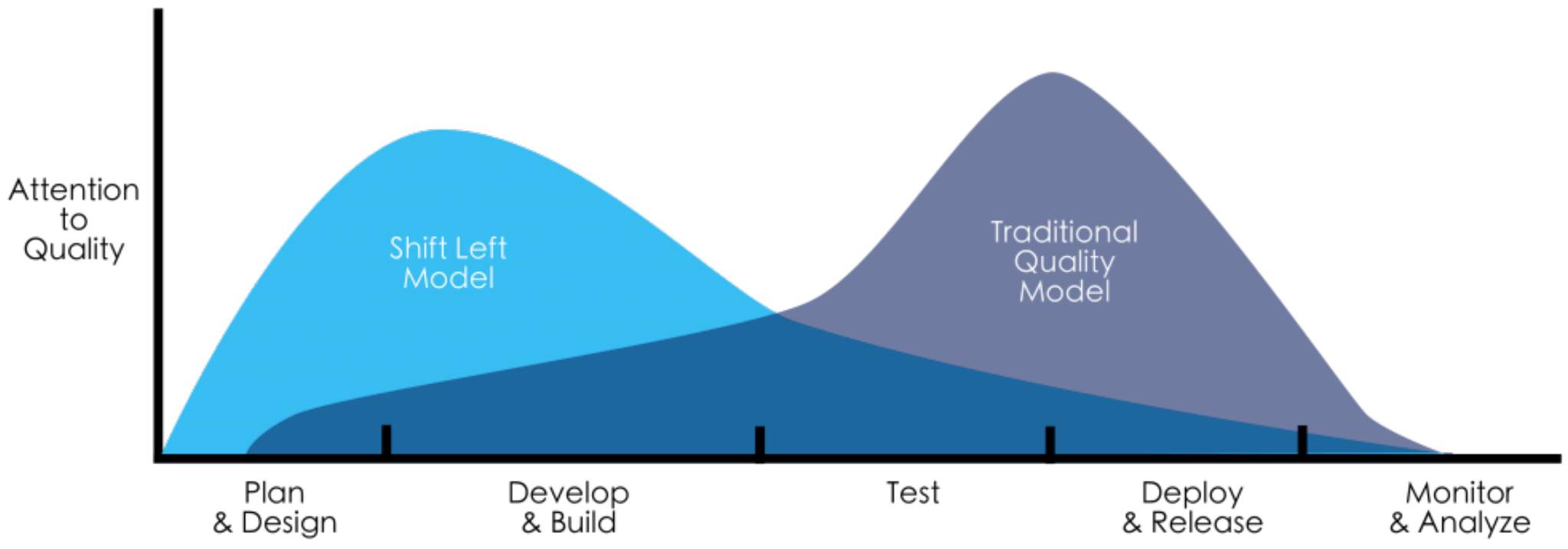
大きなバグの海



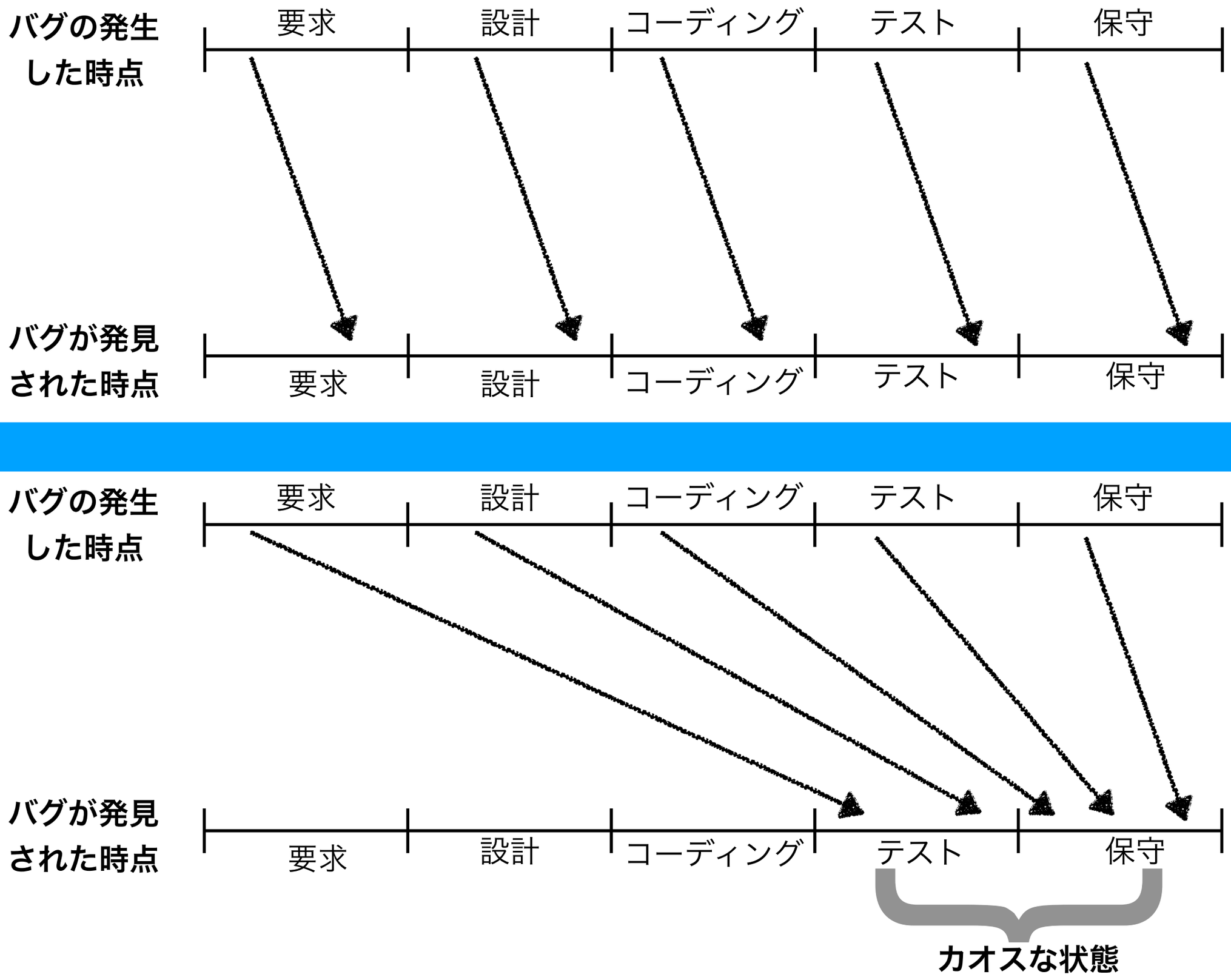
大きなバグを見逃さない
限られた時間内で多くの仕事をする

- **NG**: 組み合わせテスト（組み合わせのバグはほとんど出ない）
- **NG**: UIをなぞるだけの自動化テスト（遅すぎる）
- **OK**: 単体テストでの境界値テスト（網羅率を計測でき品質指標になる）
- **OK**: 単体テストの自動化

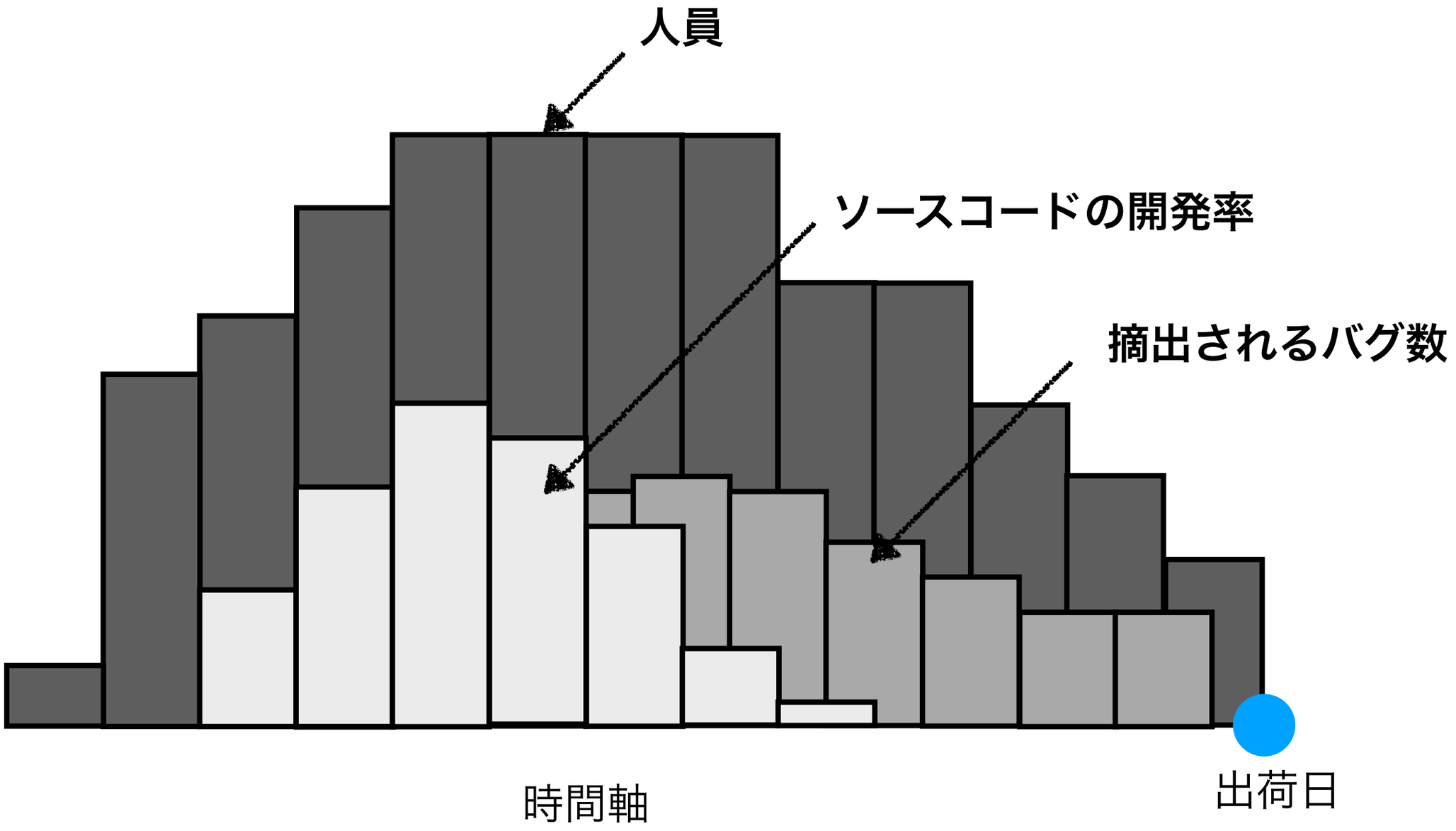
Shift Leftとは？



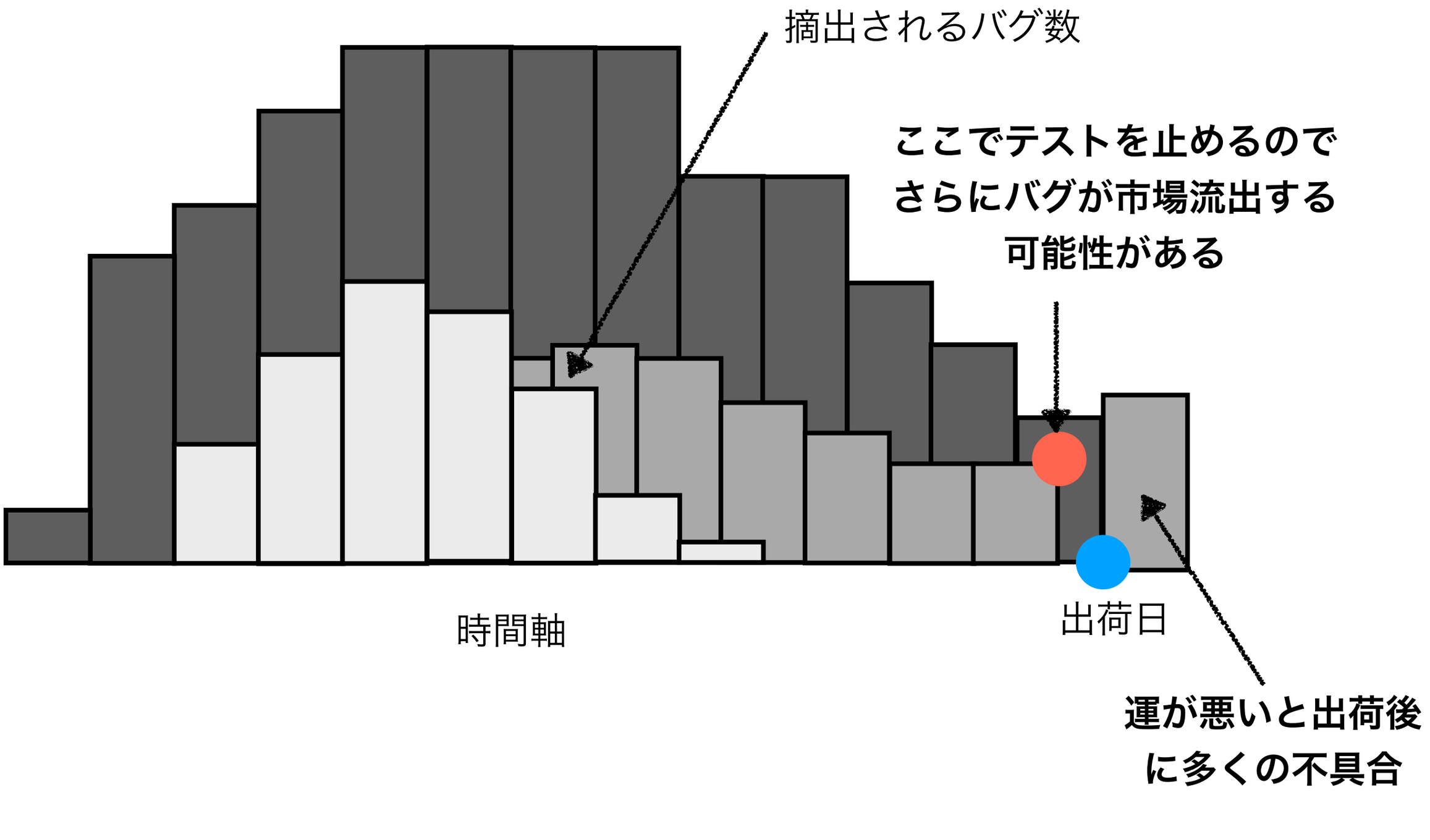
Caper Jonesの言うカオス



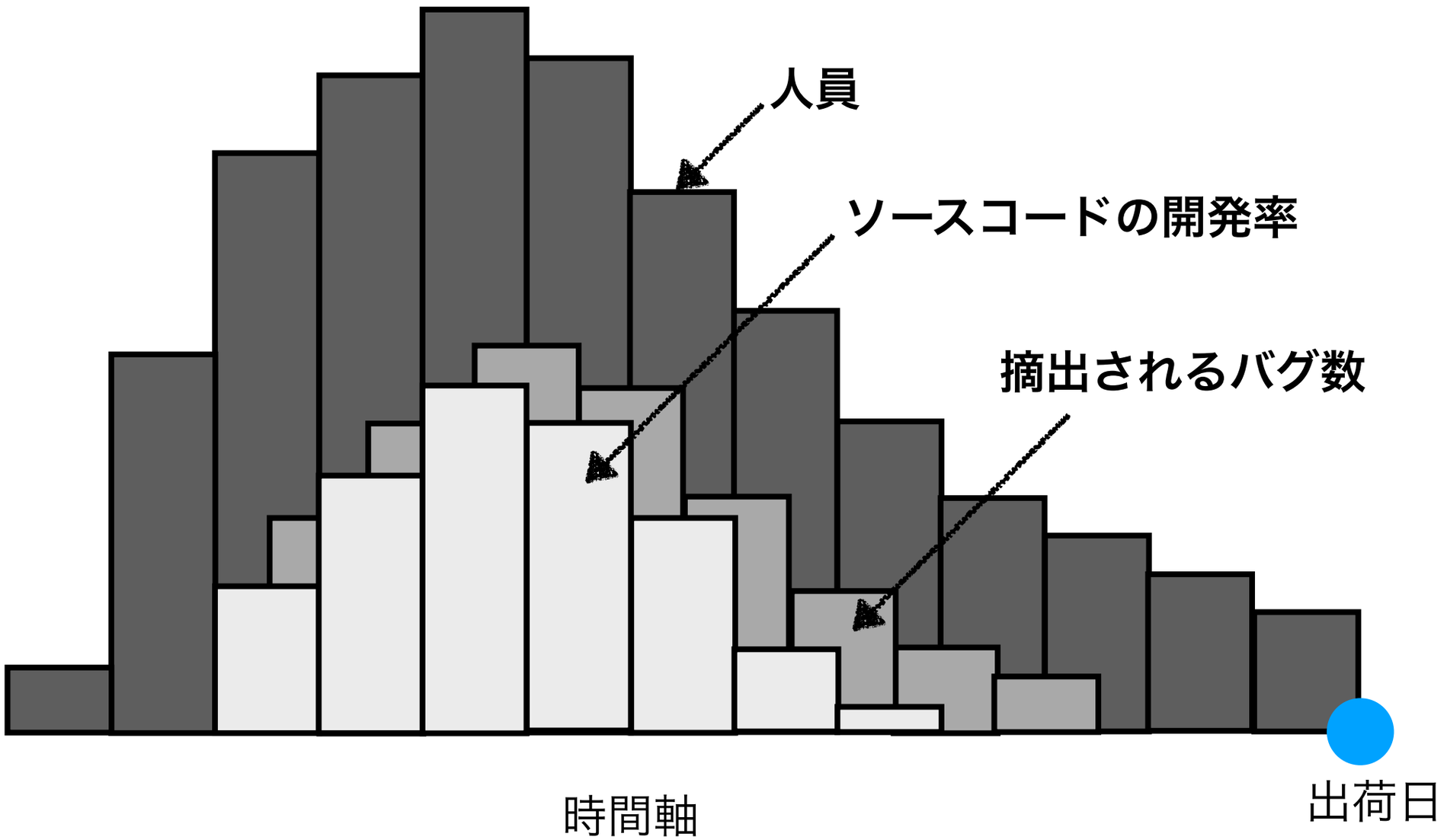
レイリー曲線



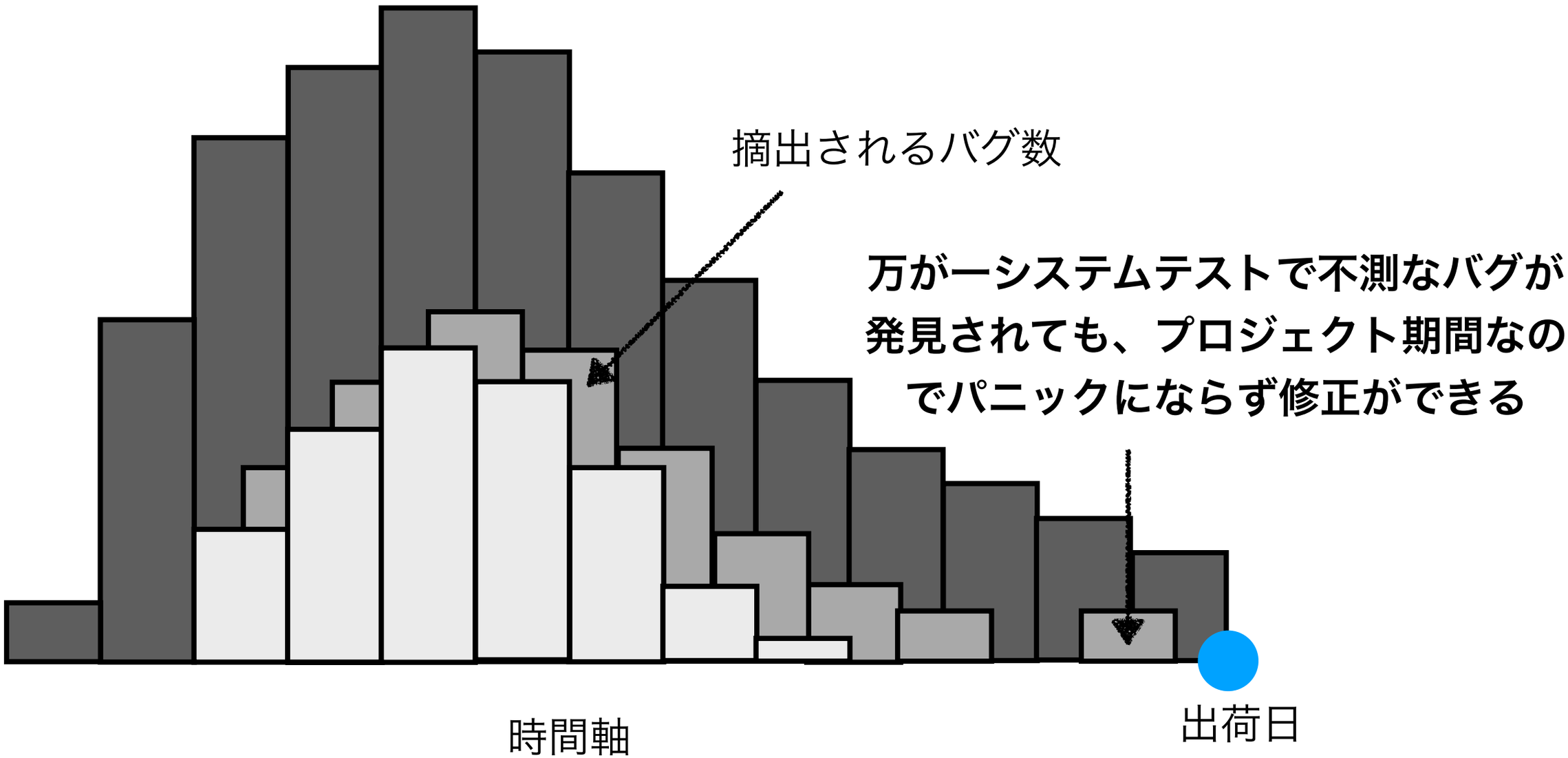
リスクがある


摘出されるバグ数
ここでテストを止めるので
さらにバグが市場流出する
可能性がある
時間軸
出荷日
運が悪いと出荷後
に多くの不具合

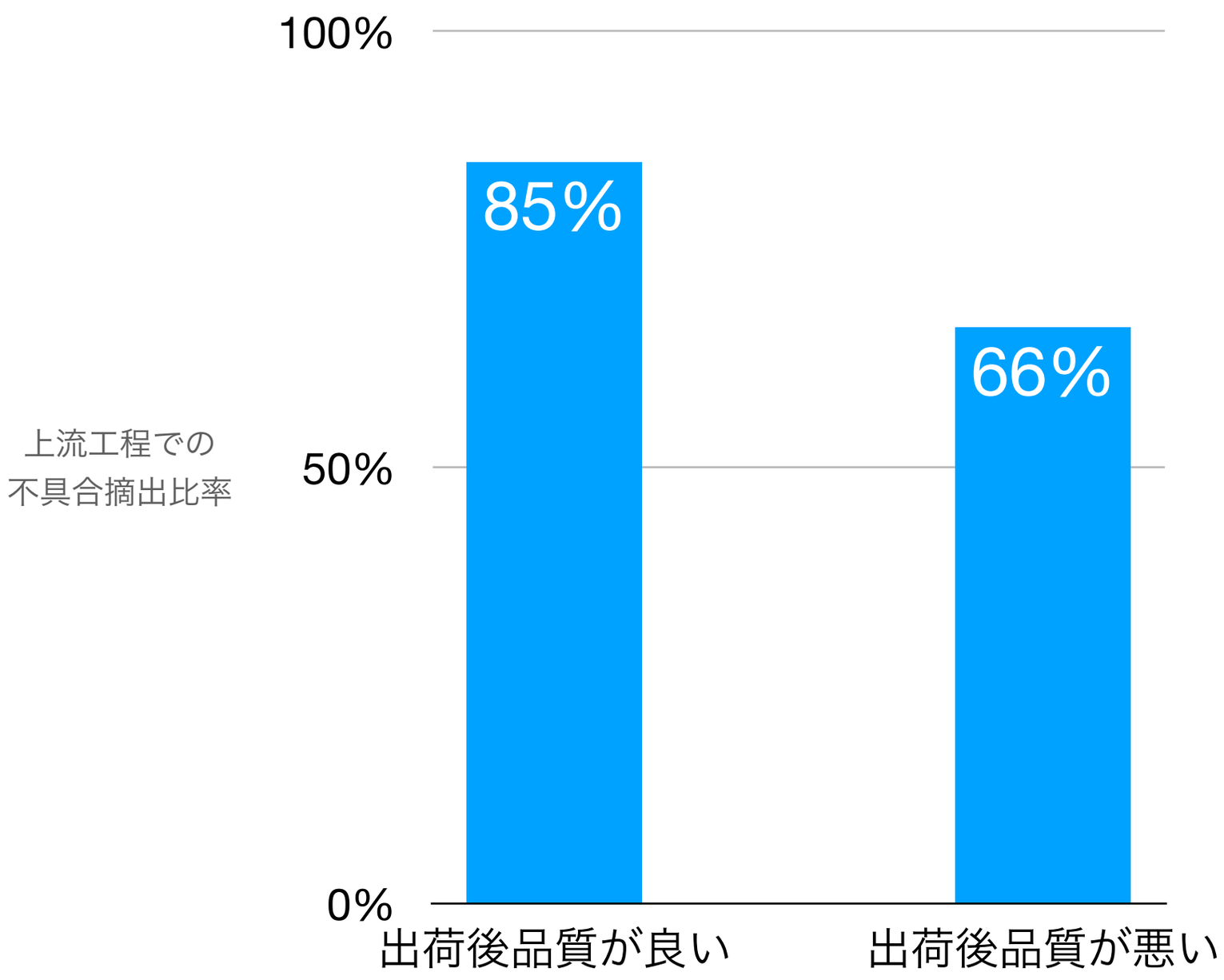
リスクがない



Cont.


摘出されるバグ数
万が一システムテストで不測なバグが
発見されても、プロジェクト期間なの
でパニックにならず修正ができる
時間軸
出荷日

上流テストを十分行いバグを検出したグループと上流で十分バグを検出しなかったグループの比較



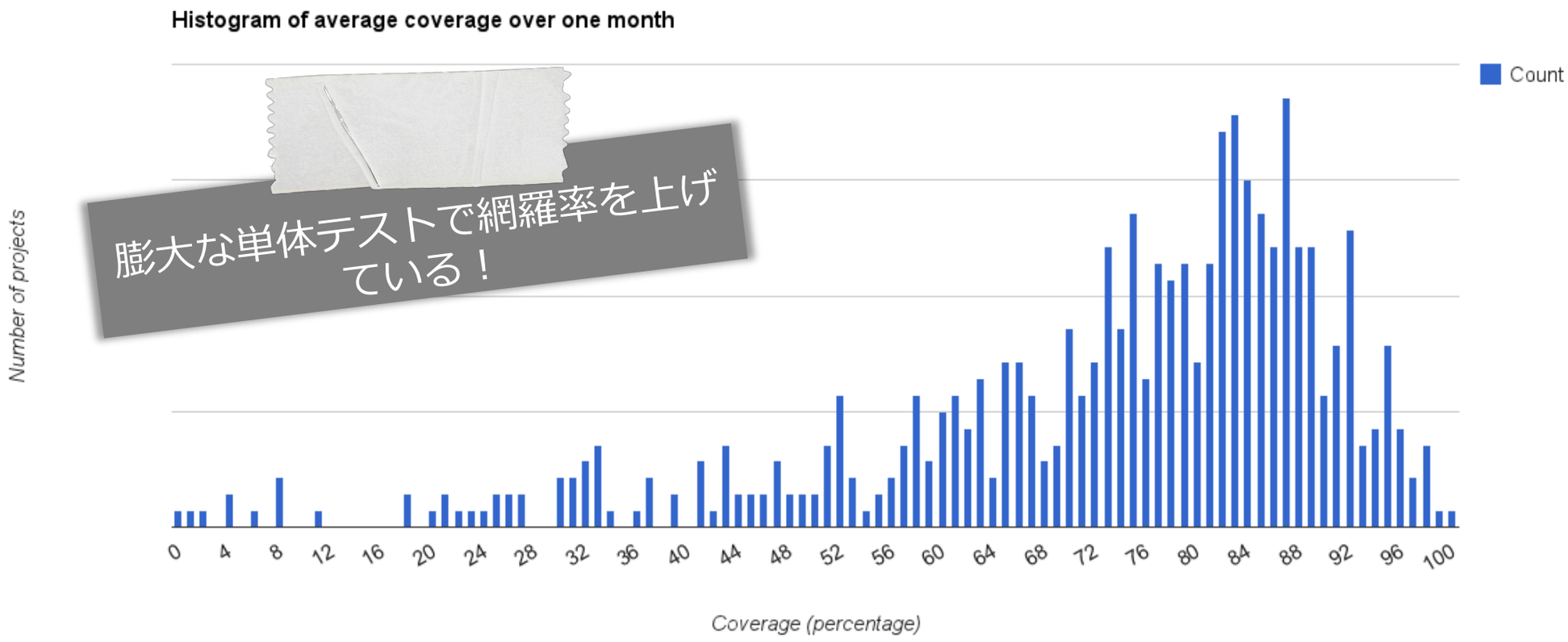
Dailyで上流品質を担保するための３つのこと

- 単体テスト
- リファクタリング
- 要求仕様のテストケース展開

Dailyで品質を担保するための３つのこと

- 単体テスト
- リファクタリング
- 要求仕様のテストケース展開

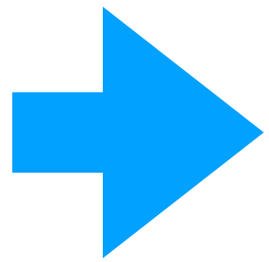
2017年Googleの網羅率データ
中央値は78%



膨大な単体テストで網羅率を上げている！

Google単体テスト基準

- 網羅率60%： 許容範囲（acceptable）
- 網羅率75%： 推奨（Commendable）
- 網羅率90%： 模範的(exemplary)



どの組織もこっから
初めましょう！

Google単体テスト基準

- 低いコード網羅は大きいエリアがテストされていない(low code coverage number does guarantee that large areas of the product are going completely untested)
- 高いコード網羅だからって品質が高いわけじゃない。もし高品質を確認したい場合はmutation testを使うのも手です（A high code coverage percentage does not guarantee high quality in the test coverage.）
- よく変更されるコードは網羅されるべき（Make sure that frequently changing code is covered）
- レガシーコードの網羅率をほっとくのはいいけど、でも少しずつ網羅率をあげていきましょう。you can adopt the ‘boy-scout rule’ (leave the campground cleaner than you found it)

SKIP

75% Code Coverage

大変ですよね？

Assert突っ込むだけじゃ達成しませんよね？

つらい単体テストは

２：８の法則を使いませんか？

あとテスト手法使うと網羅率あがりまっせ！

２割の部分に８割以上のバグが潜んでいます。

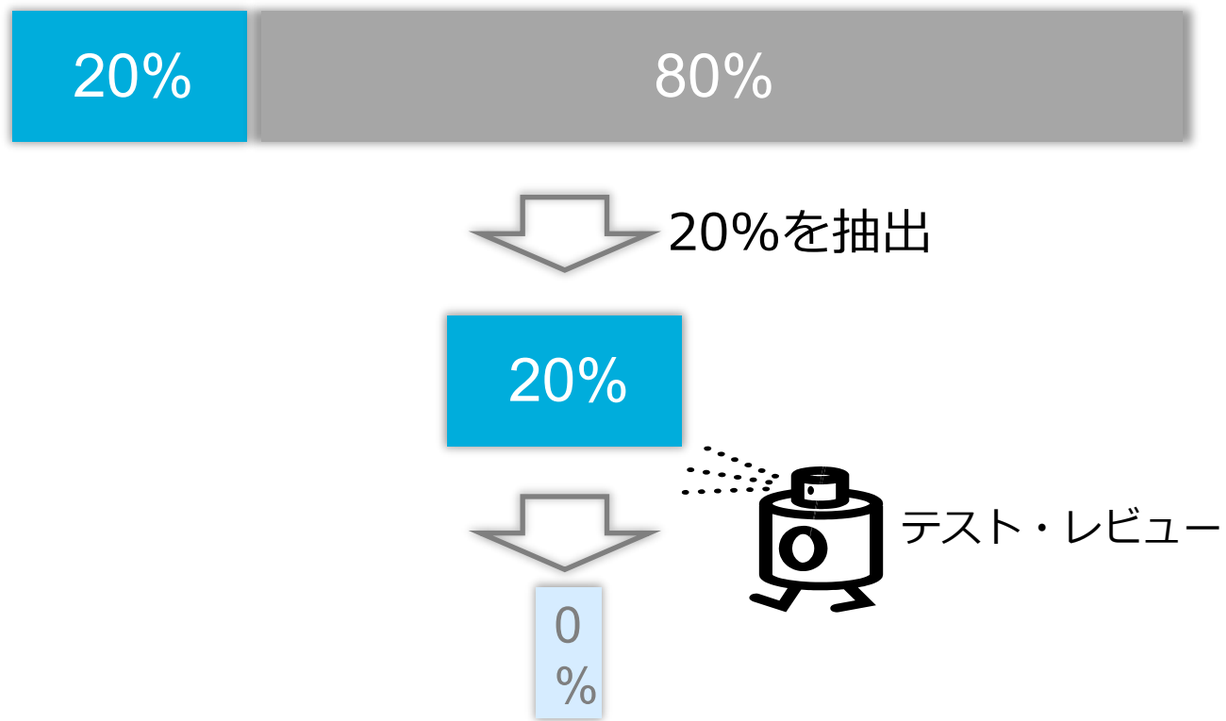
どこにバグが潜んでいるか？

Top 1: よく変更されるファイル

Top 2: 長いファイル

Top 3: 複雑度が高い

基本はファイル単位、関数単位ではない。ファイル単位で単体テスト計画を立てる必要がある。


20%
80%
20%を抽出
20%
テスト・レビュー
0
%

２割の部分に８割以上のバグが潜んでいます。

どこにバグが潜んでいるか？

Top 1: よく変更されるファイル

Top 2: 長いファイル

Top 3: 複雑度が高い

基本はファイル単位、関数単位ではない

HotSpot概念

BugCache

Predicting Defects

Sung Kim • MIT
Tom Zimmermann • Saarland University
Jim Whitehead • UC Santa Cruz
Andreas Zeller • Saarland University

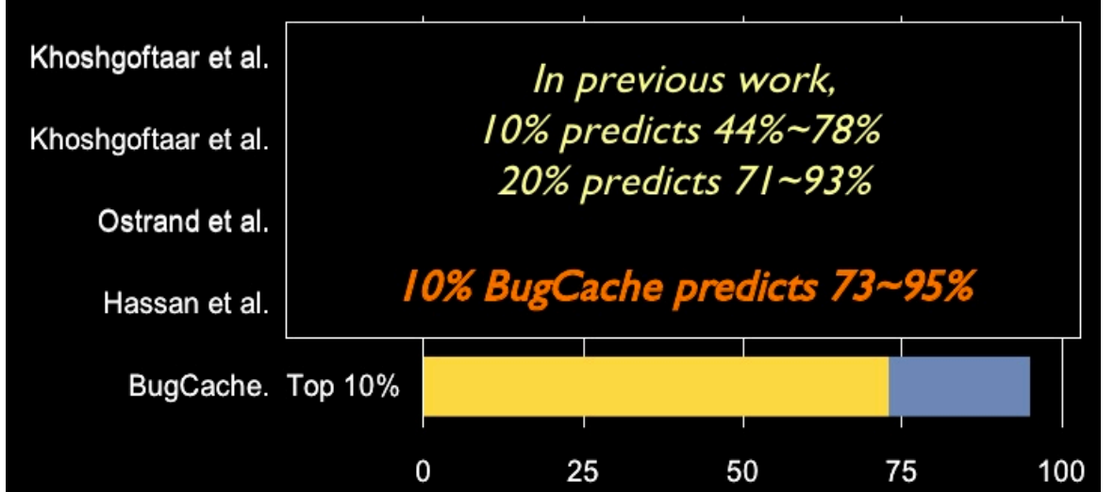
Function Level Optimal Hit Rates

Project	Function	Hit rate	Block	Pre-fetch	Replace
Apache 1.3	2,113	62%	15%	17%	BUG
Columba	8,428	68%	57%	20%	BUG
Eclipse	33,214	72%	20%	4%	BUG
JEdit	5,489	49%	85%	8%	BUG
Mozilla	8,203	55%	41%	14%	LRU
PostgreSQL	8,659	59%	29%	17%	LRU
Subversion	3,693	46%	71%	14%	BUG

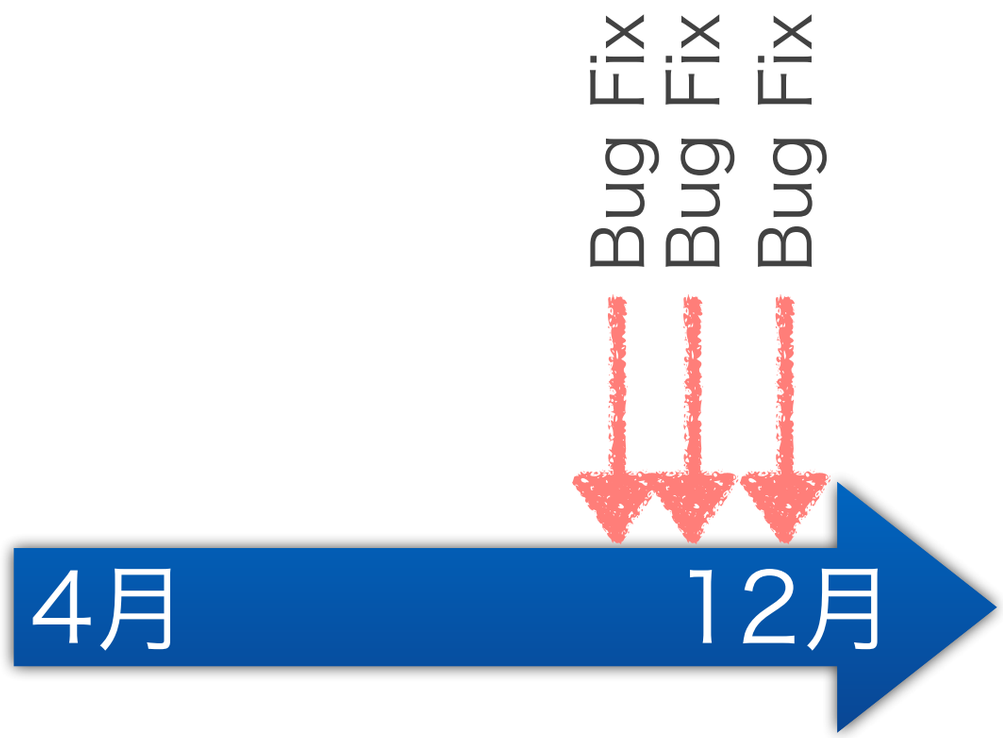
Cache size = 10% of all functions/methods

The Problem

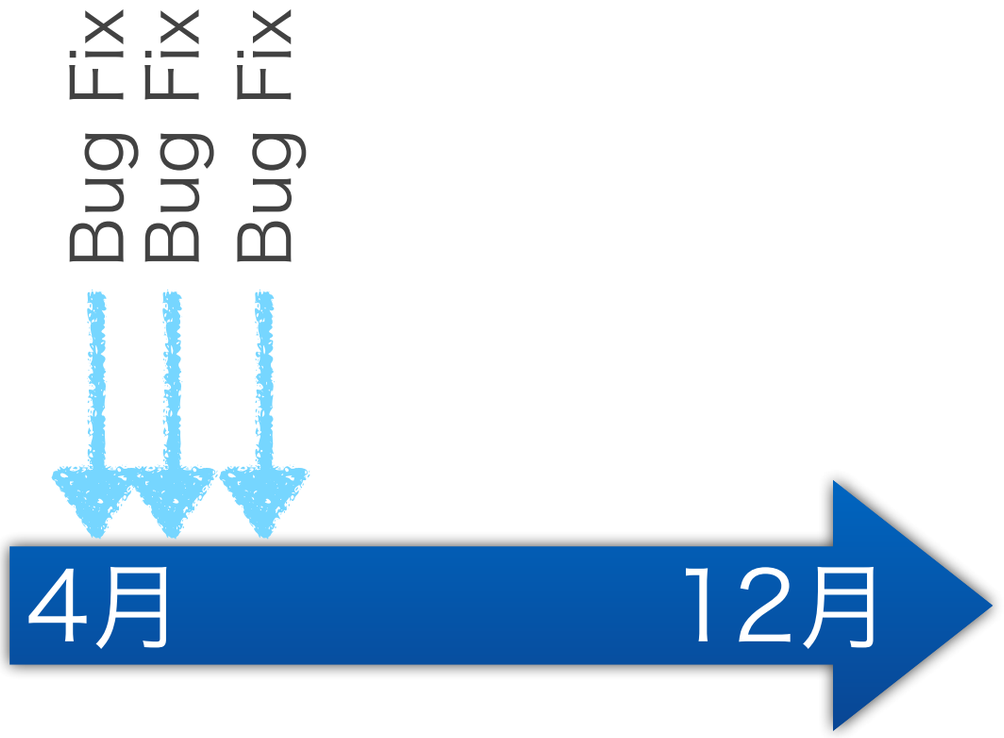
Related Work



SKIP

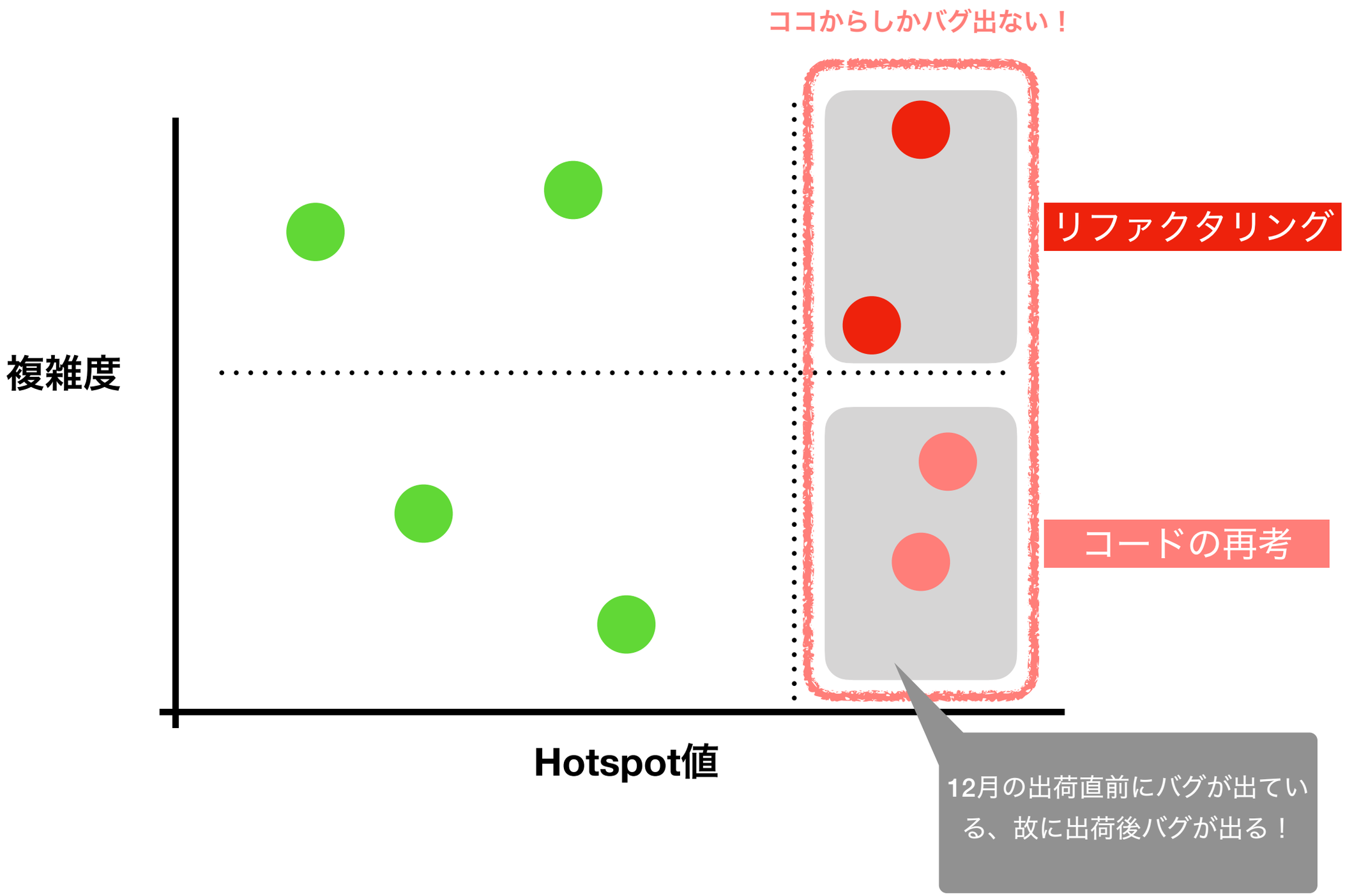


Hot Spot値　高

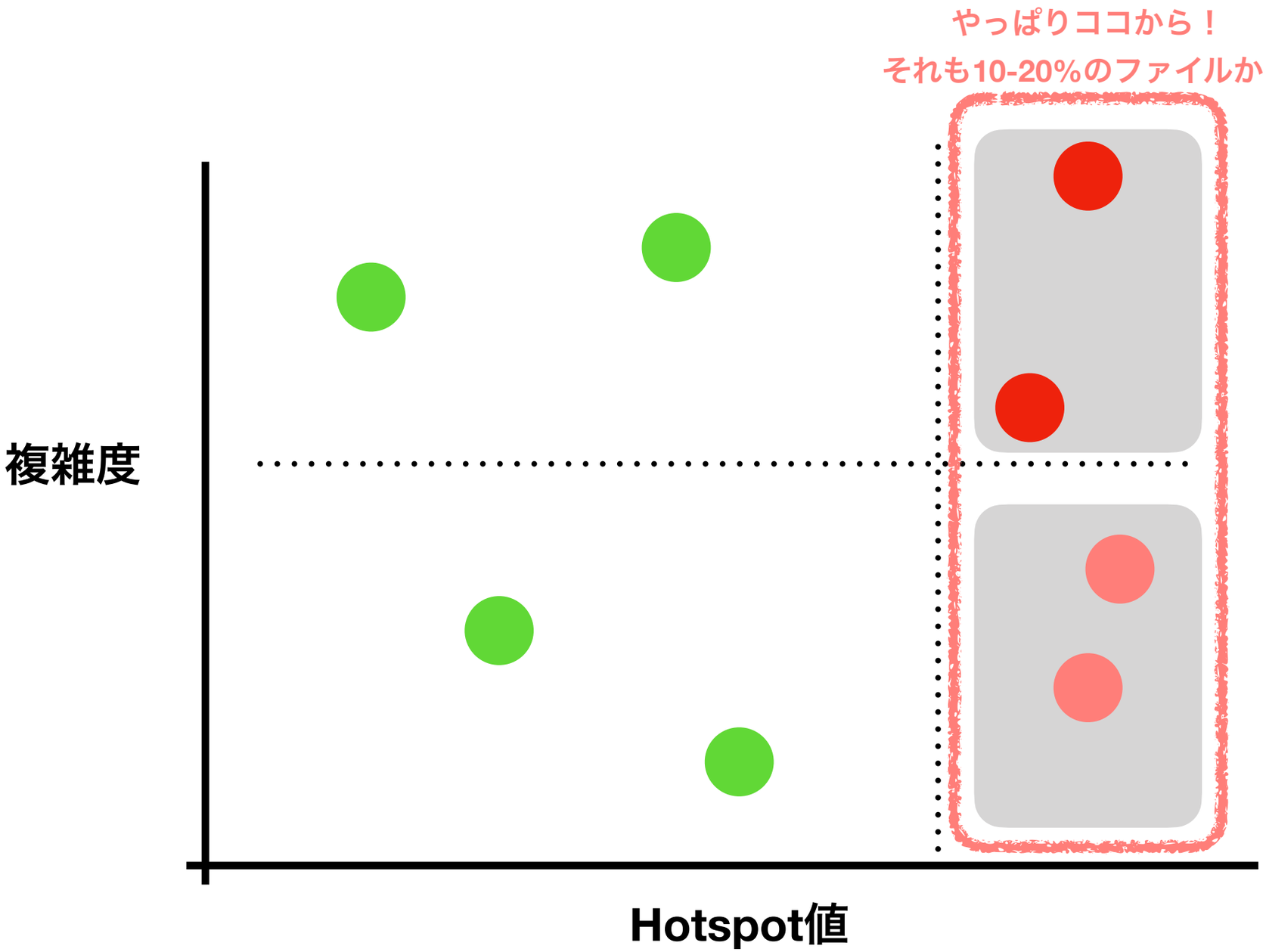


Hot Spot値　低

Quality Measurement Criteria



出荷後のバグはどこから出るか？



２割の部分に８割以上のバグが潜んでいます。

どこにバグが潜んでいるか？

Top 1: よく変更されるファイル

Top 2: 長いファイル

Top 3: 複雑度が高い

基本はファイル単位、関数単位ではない

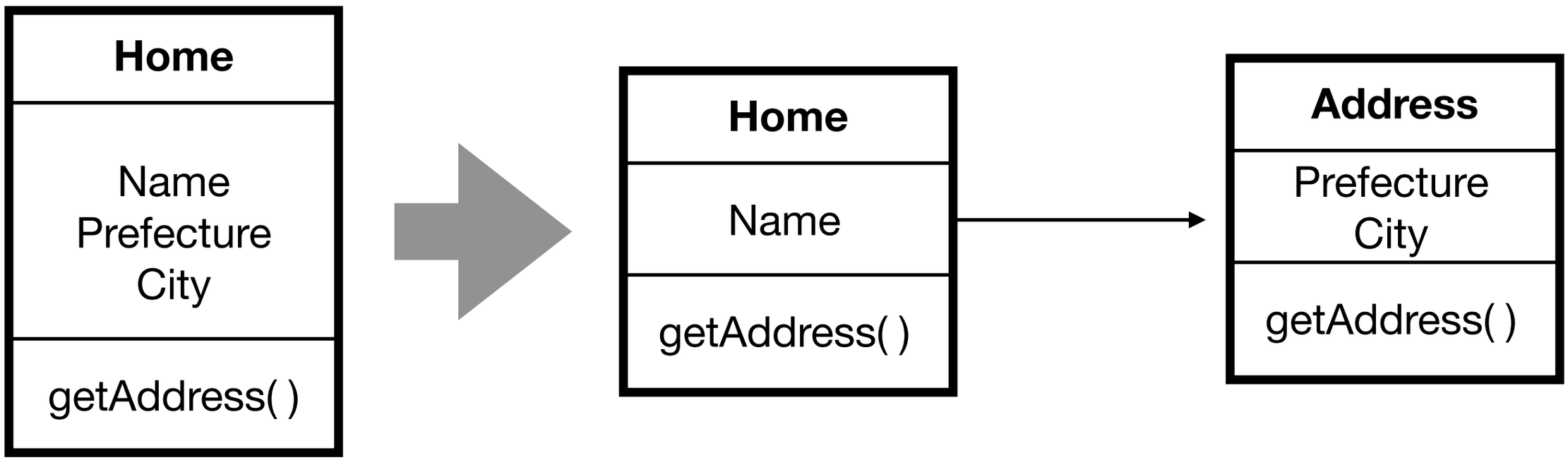
乱暴にファイルをぶった切る！

Weighted Methods Per Class(WMC)

$$WMC = \sum_{i=1}^{n} Ci$$

SKIP

Martin Fowlerのクラス抽出



SKIP

２割の部分に８割以上のバグが潜んでいます。

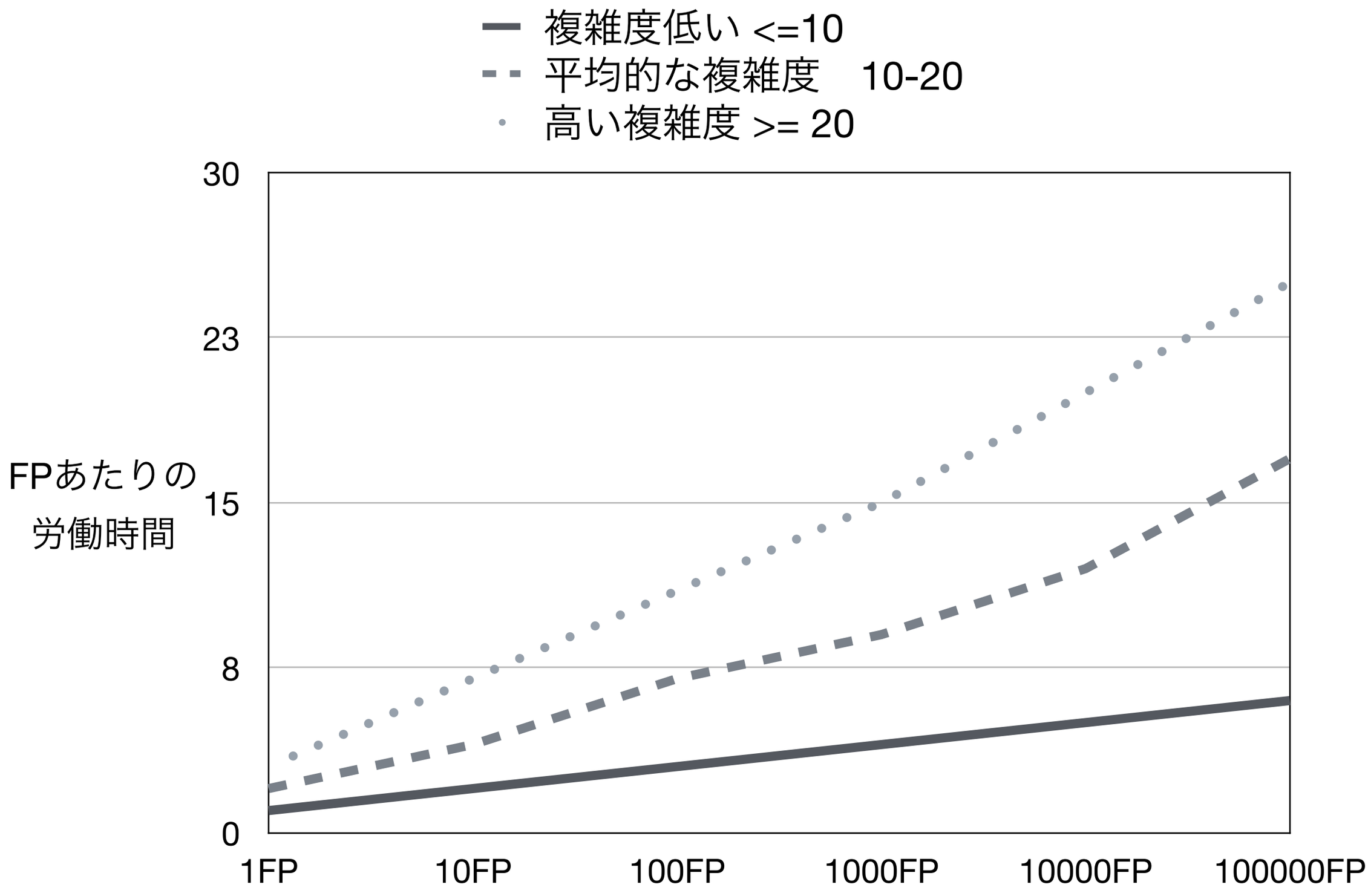
どこにバグが潜んでいるか？

Top 1: よく変更されるファイル

Top 2: 長いファイル

Top 3: 複雑度が高い

基本はファイル単位、関数単位ではない


複雑度低い <=10
平均的な複雑度　10-20
高い複雑度 >= 20
30
23
15
8
0
FPあたりの
労働時間
1FP
10FP
100FP
1000FP
10000FP
100000FP

複雑なコードのバグ修正成功確率

循環的複雑度	複雑さの状態	バグ混入確率
10以下	非常に良い構造	25%
30以上	構造的なリスクあり	40%
50以上	テスト不可能	70%
75以上	いかなる変更も誤修正を生む	98%

複雑な関数の修正怖いので、リファクタリングしたくない。。。
でもどっちにしろ致命的なバグはでるので早めのリファクタリング

ちゃんと単体テスト書いてから、リファクタリングしましょう！

Dailyで上流品質を担保するための３つのこと

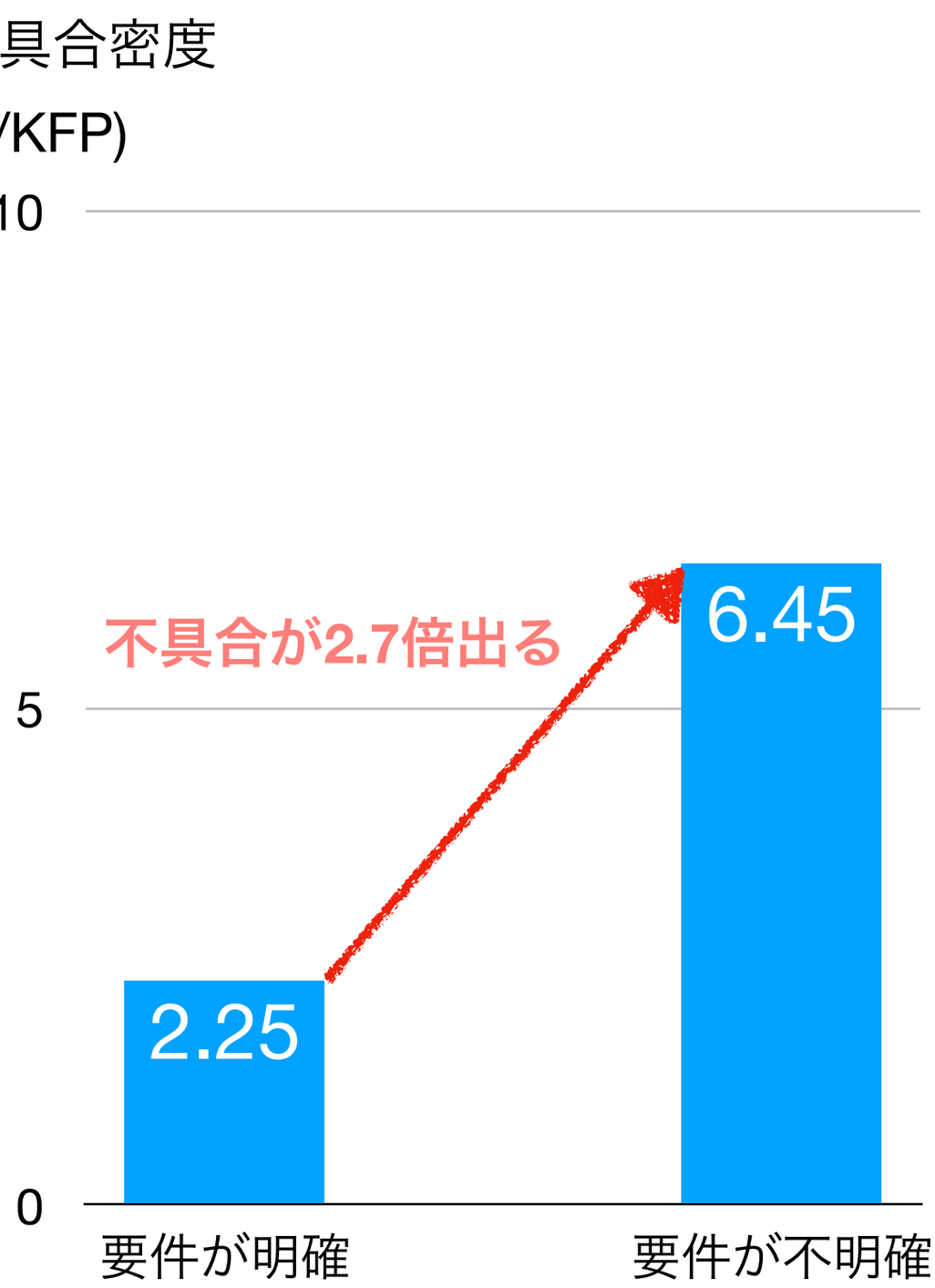
- 単体テスト
- リファクタリング
- 要求仕様のテストケース展開

要求仕様

- 要求仕様は品質を担保する上で重要なドキュメントです
  （大きいバグになる）

- 要求仕様とテストケースは明示的にリンクを貼りましょう

要件定義を明確に行ったグループと行わなかったグループの不具合比較

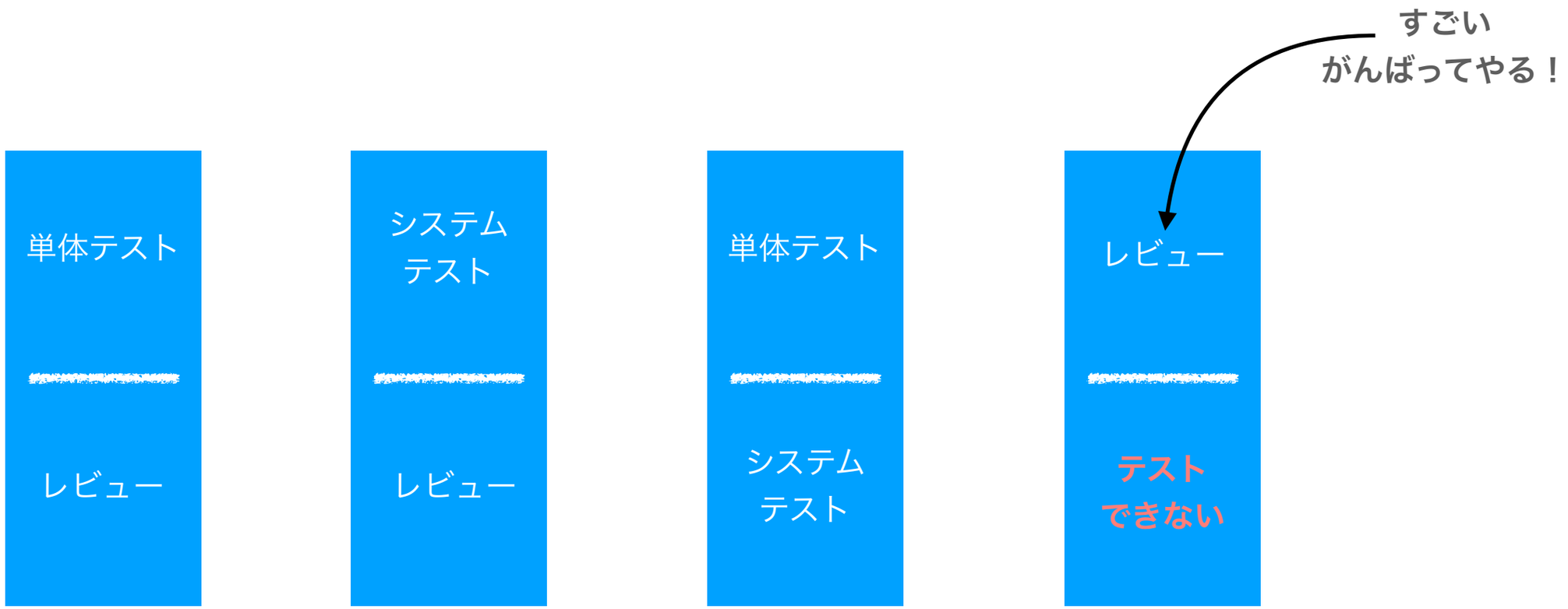


品質保証の実際１

心構え編

品質保証とは

- ２つ以上の品質保証手法によって、リスクのあるコンポーネントをテストする
- リスクなければ、テストしなくていい（かな？）。
- リスクのあるコンポーネントとは。Hotspotが高い、新規追加、セキュリティ等々



リーダーが認識し、リスクヘッジする

テストができない箇所は特定しておく

- Race Condition(データベース・スレッドセーフ)
  プログラミング言語やシステムの違いによるrace condtionバグを理解し、race condtionのバグを出さない工夫することはmustです。
- Memory Leak
- Security
- その他

テストできない箇所はレビューの重みをつけておく

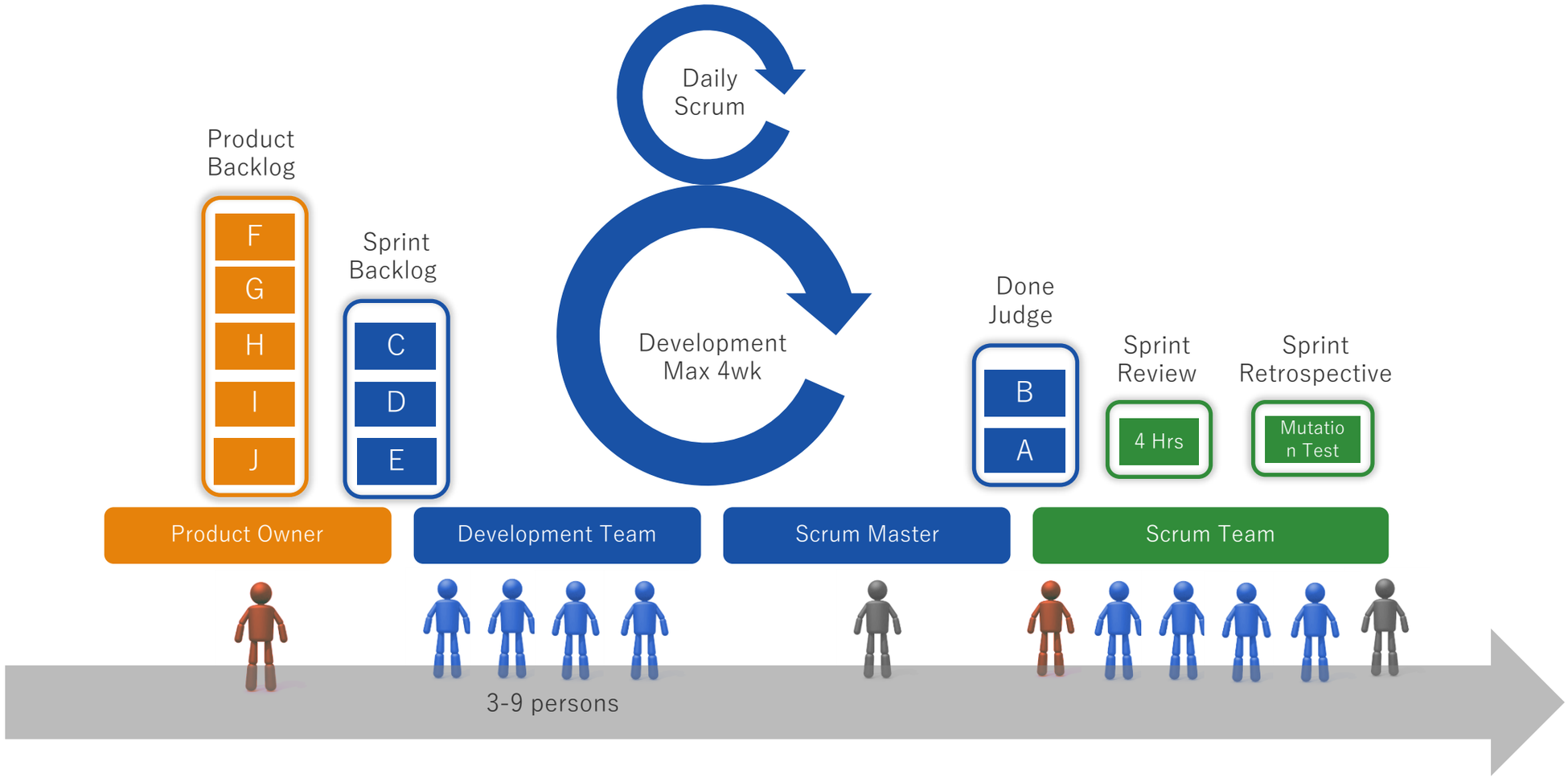
品質保証の実際２

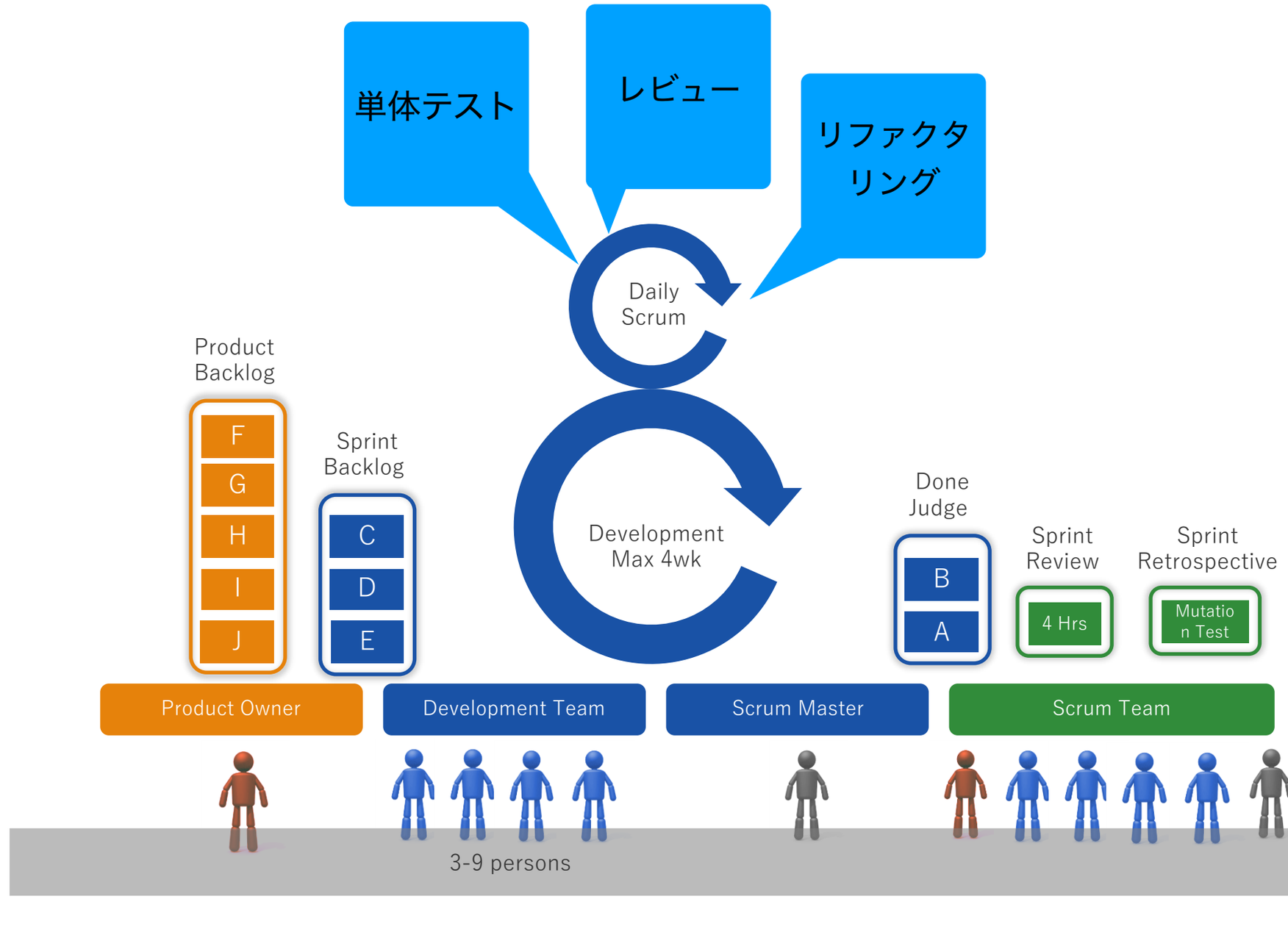
Daily 品質の担保

テストは基本すべて自動化

すべての自動テストは最低その日に終了

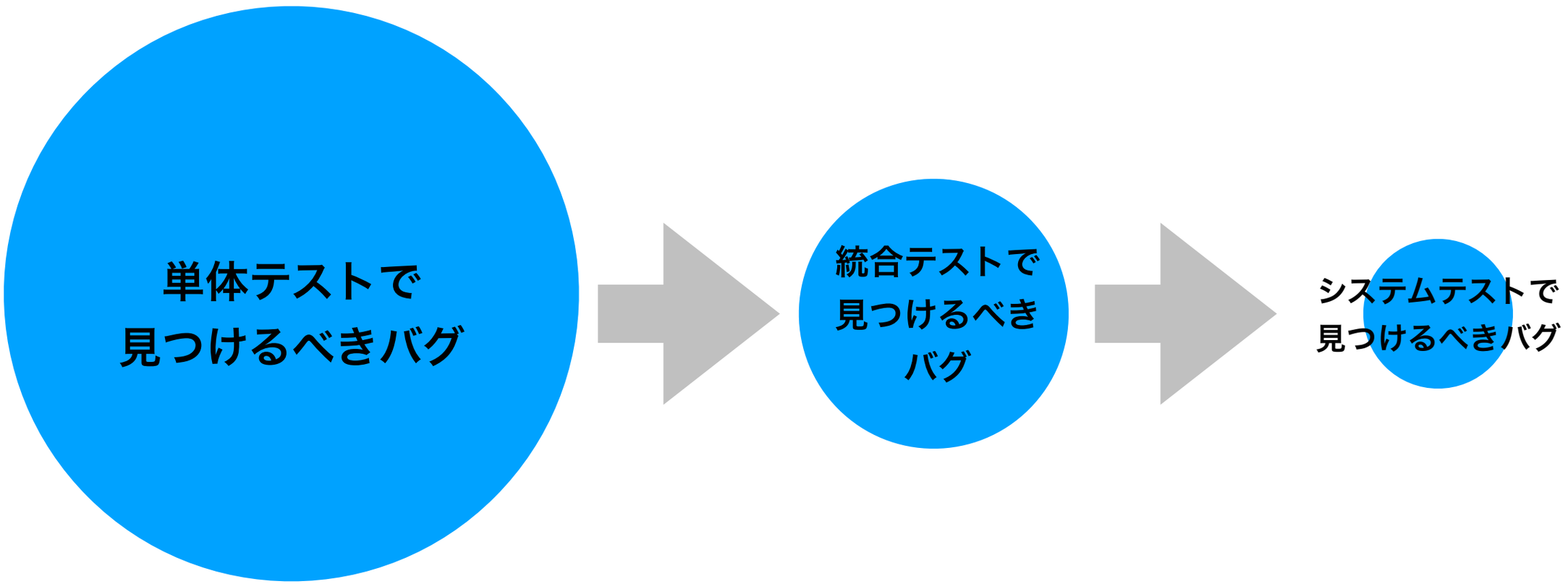
現代では：その日に品質担保できないものは、出荷後確実に市場問題となる、市場問題って複雑なパフォマンスやセキュリティテストの問題ではないですよね？

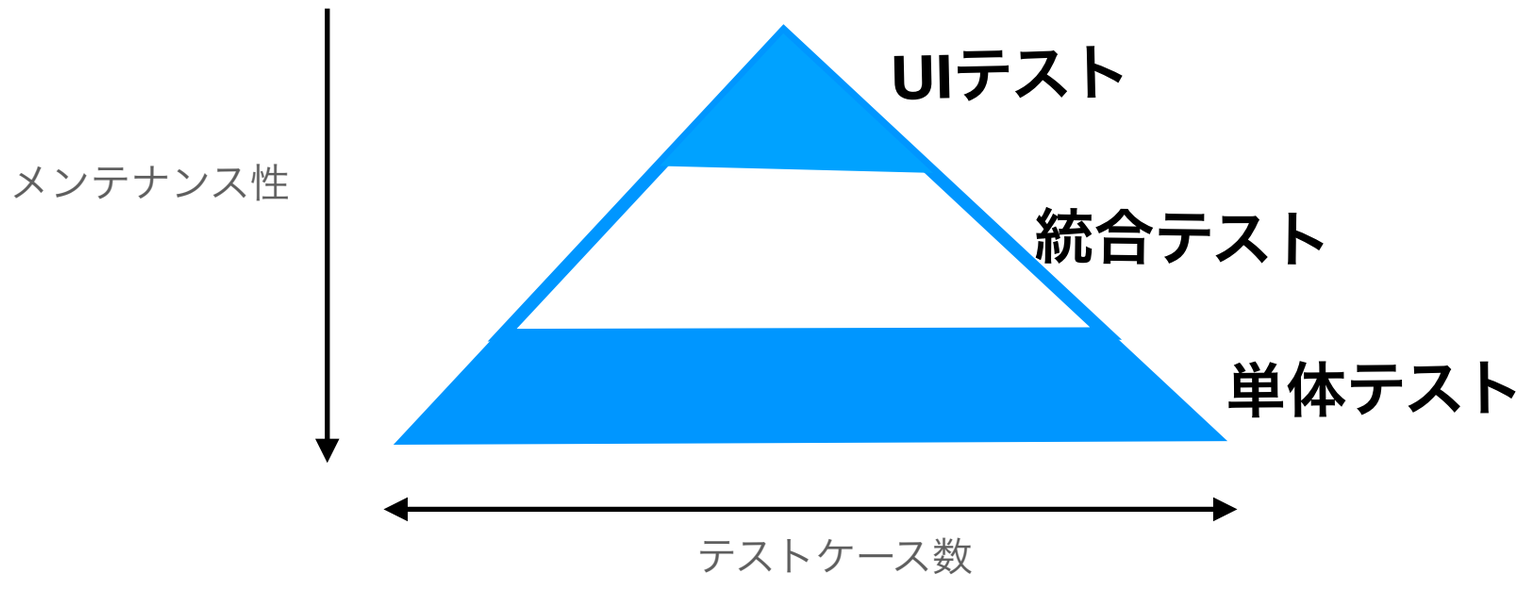

Product Backlog
F
G
H
I
J
Sprint Backlog
C
D
E
Daily Scrum
Development Max 4wk
Done Judge
B
A
Sprint Review
4 Hrs
Sprint Retrospective
Mutatio n Test
Product Owner
Development Team
Scrum Master
Scrum Team
3-9 persons



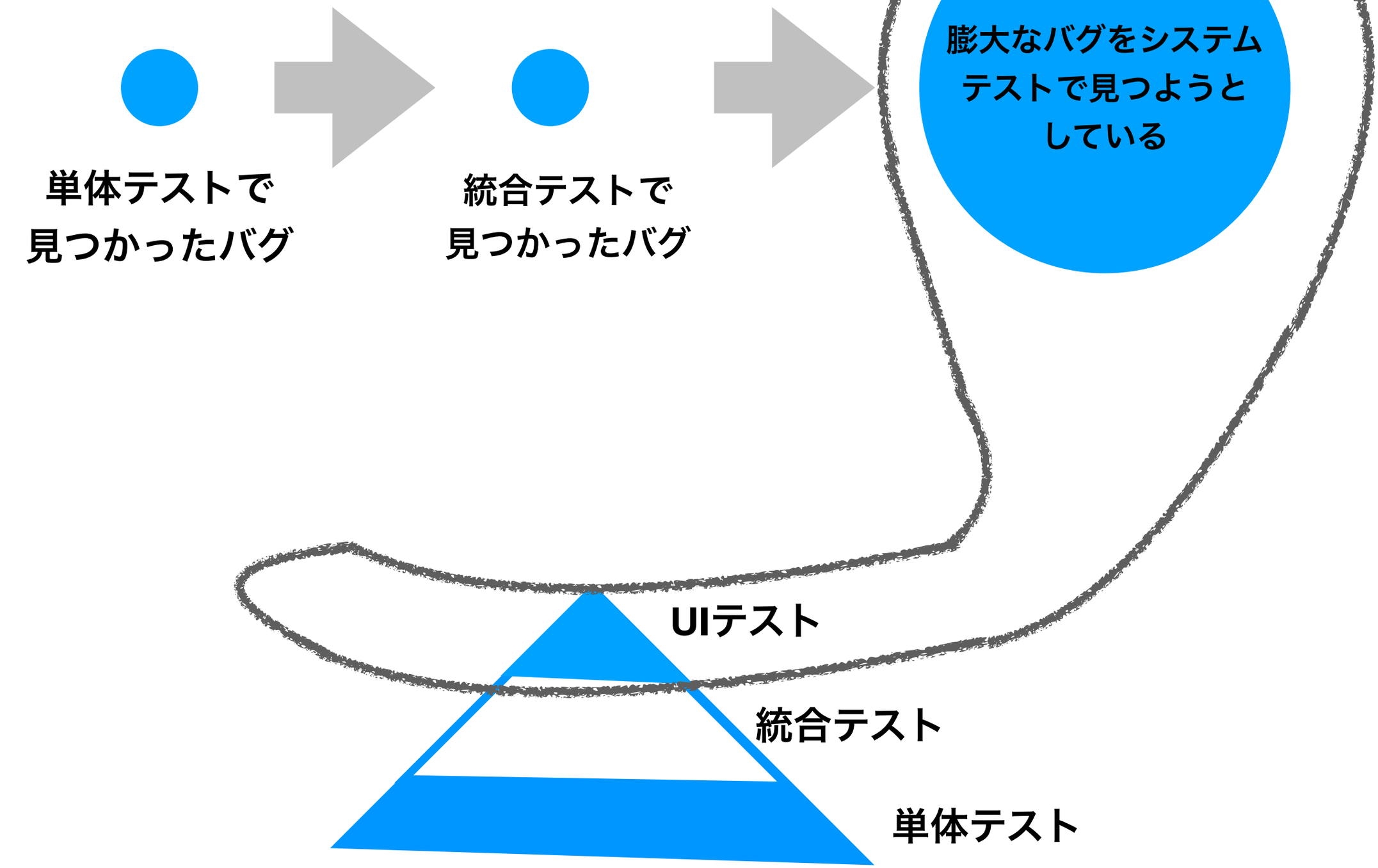
毎日定量的品質のわからないCI/CDって意味あります？

良い自動化


単体テストで
見つけるべきバグ
統合テストで
見つけるべき
バグ
システムテストで
見つけるべきバグ


メンテナンス性
UIテスト
統合テスト
単体テスト
テストケース数

悪い自動化

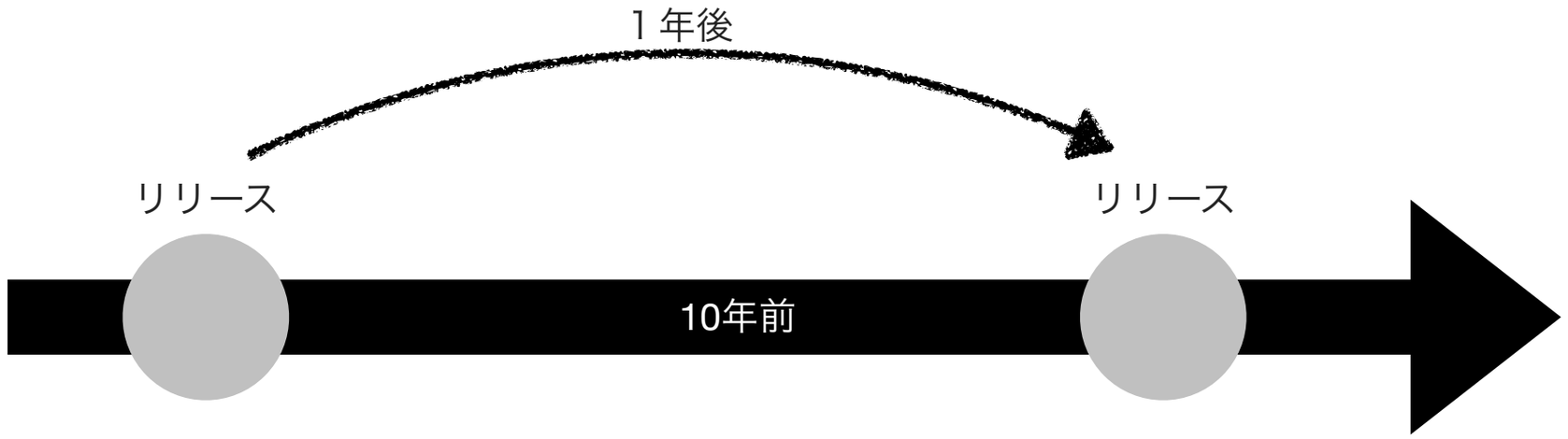

単体テストで
見つかったバグ
統合テストで
見つかったバグ
膨大なバグをシステム
テストで見つようと
している
UIテスト
統合テスト
単体テスト

あなたの自動化大丈夫ですか？

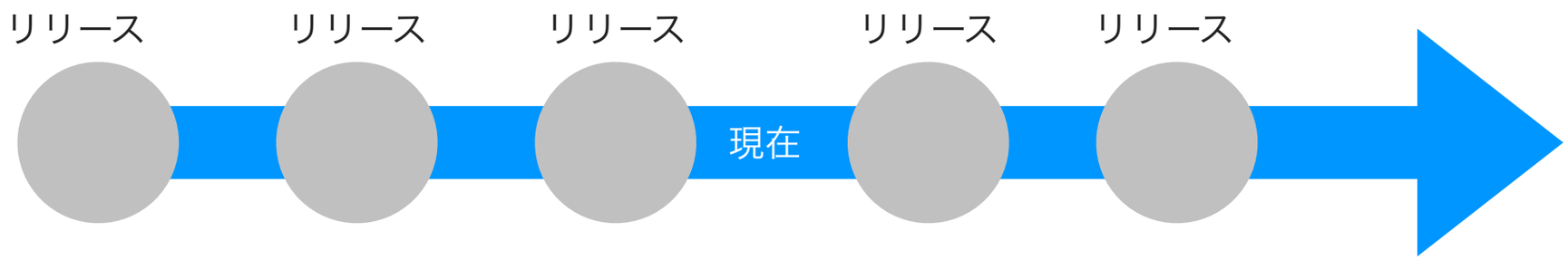
自動化健全性チェックリスト

- 網羅率あがってます？
- スピード十分あります？

- 自動化ですべてのバグは見つからないことを認識しています？UIなぞる自動化がすべてだと思ってませんか？



毎月アップデート



あなたテストは耐えられます？

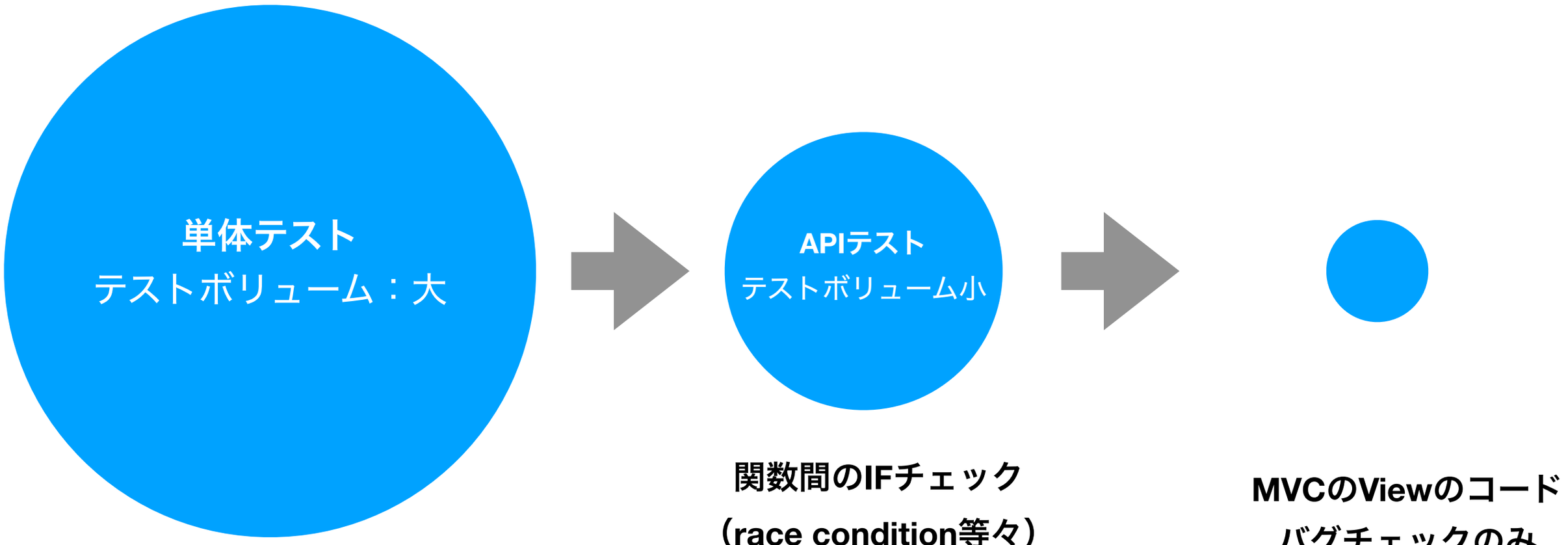
システムテストでUIをただなぞってる自動化してませんか？

- 境界値バグ — 自動単体テスト
- 状態遷移バグ — 自動APIテスト
- Race Conditionバグ — レビュー
- セキュリティバグ — レビュー
- 非機能バグ — 要求仕様段階でverification pointを明記

あれ？組み合わせテストは？

SKIP

テストケースの重み基準



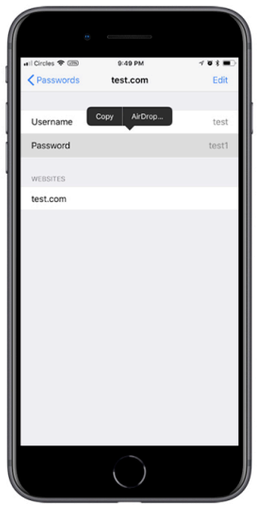
CI/CDでより多くのバグを発見
（エラーハンドリング含め）
網羅率測定

関数間のIFチェック
（race condition等々）
網羅率測定（なし）

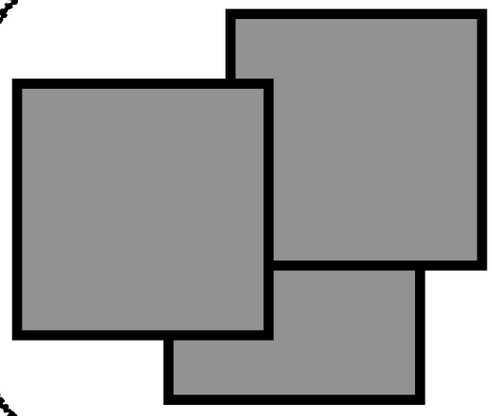
MVCのViewのコード
バグチェックのみ

テストスピード

単体テストがお得



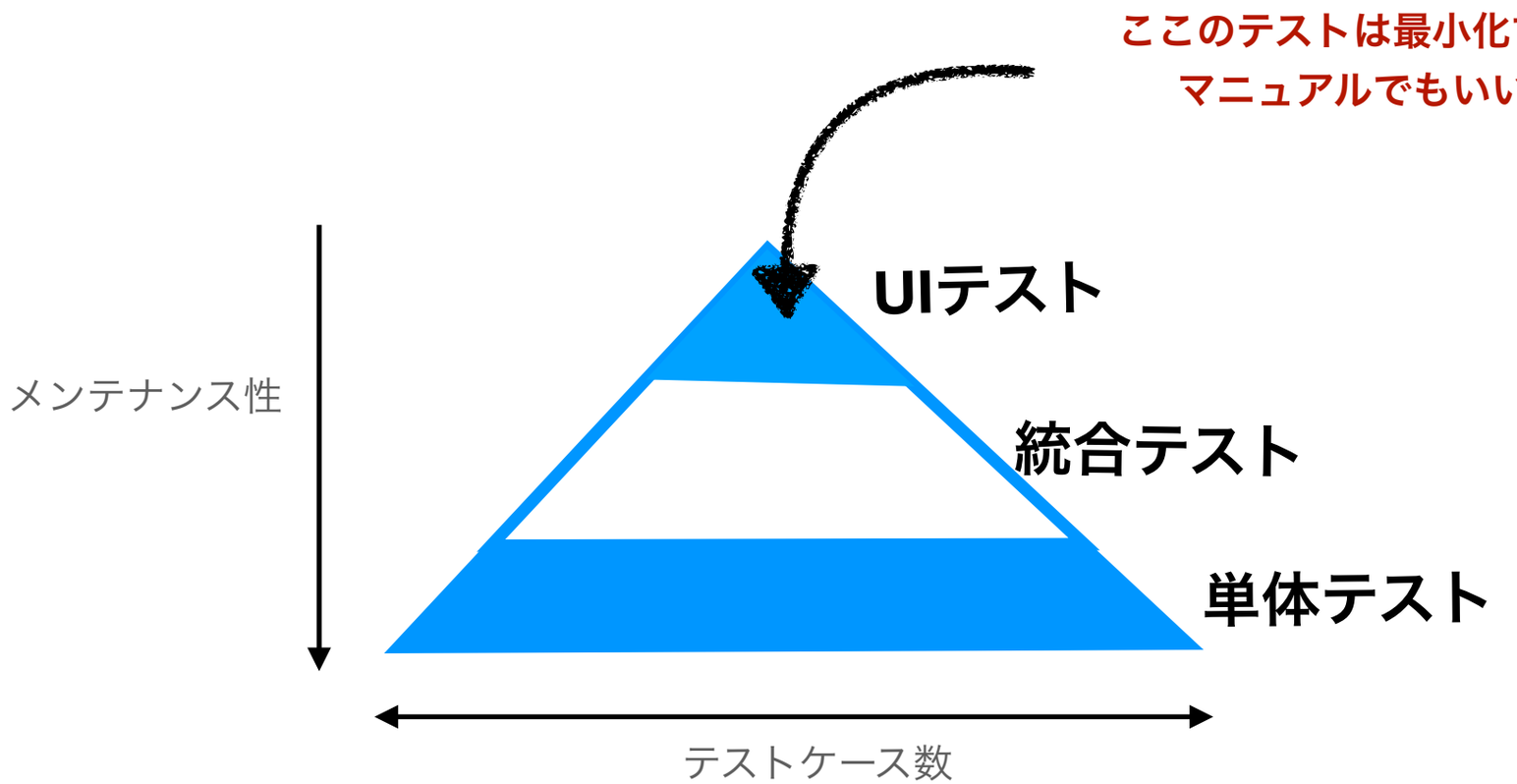
システムテストテスト（自動化）
60分



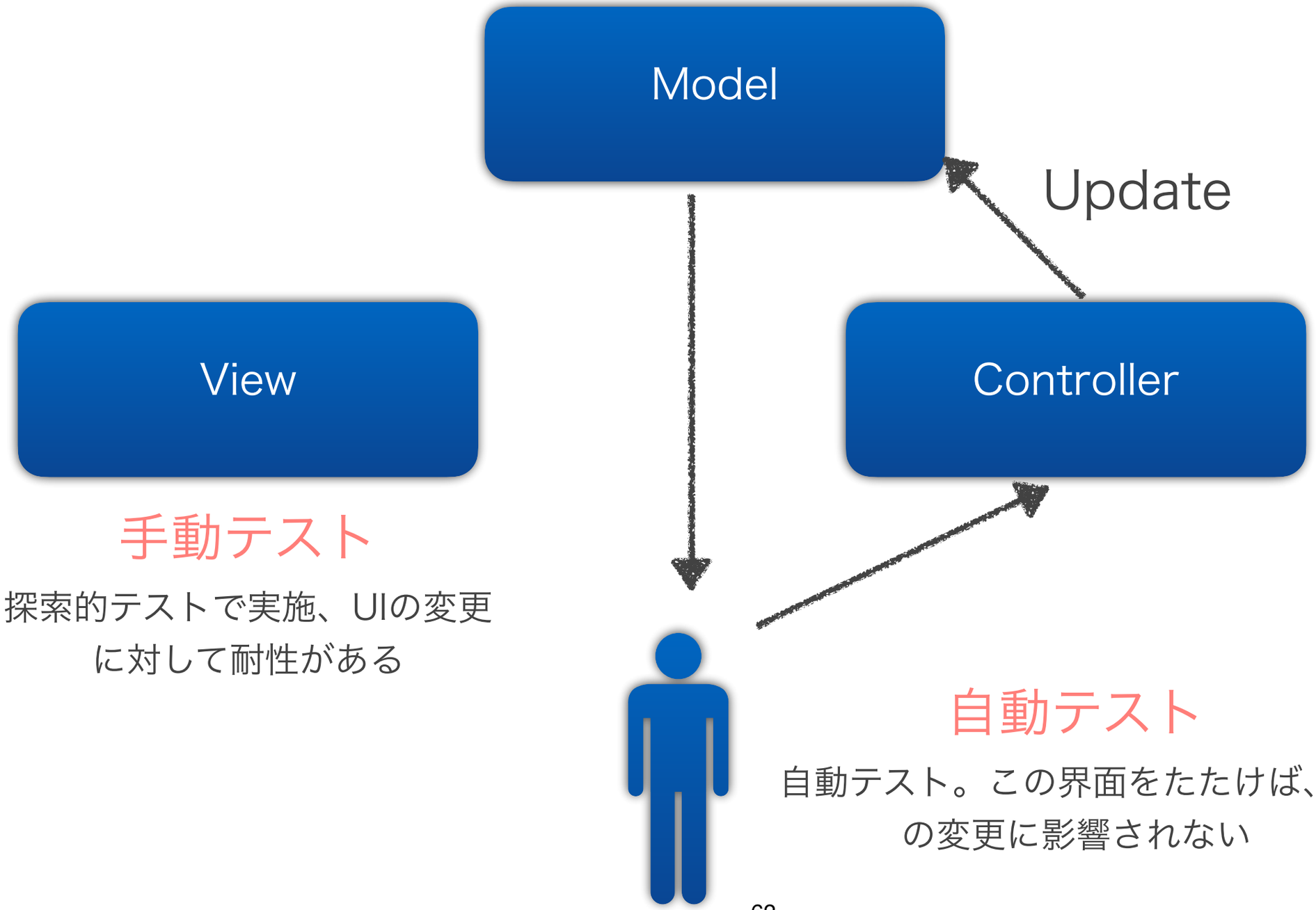
統合テスト
10分

単体テスト
３分

テストスピード


ここのテストは最小化する！
マニュアルでもいい。
UIテスト
統合テスト
単体テスト
メンテナンス性
テストケース数

MVC分離でUIテストの最小化・テスト時間の最小化



SKIP

Viewテストを分離する２つの理由

技術的な理由

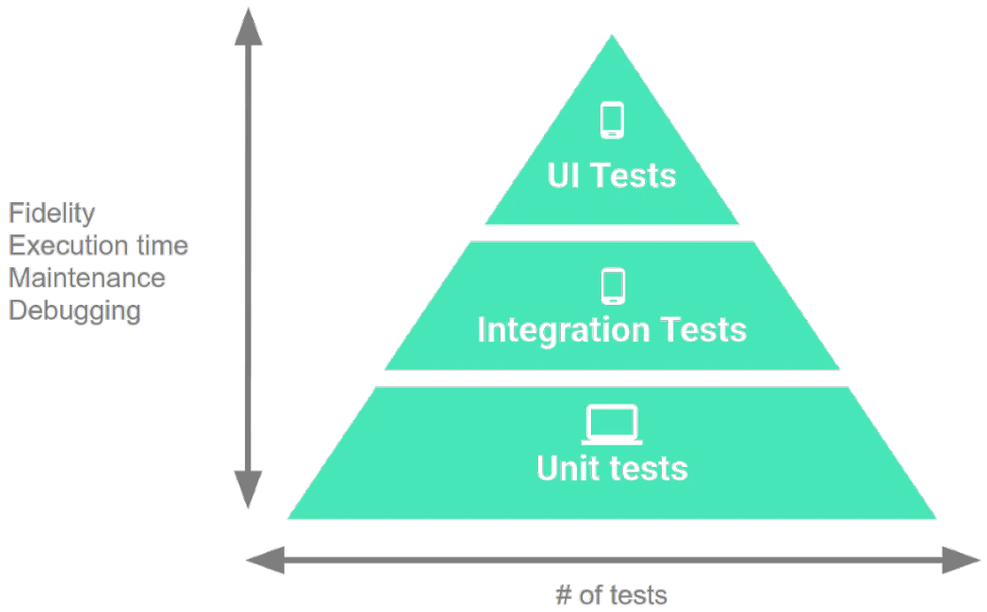
- Viewのテストは時間がかかり、UIの変更は頻繁に起こる
- Viewテスト単体テスト化が困難で、コストのかかるシステムテストになるため
- コード上でviewとコントロールを分離する(例: javascriptに計算を持たせない)

不条理な理由

- いくら事前に説明しても、企画と上司はものができてからUI変更を命令する
- 企画と上司は出荷直前でもUI変更はできると信じ、それを命令する

SKIP

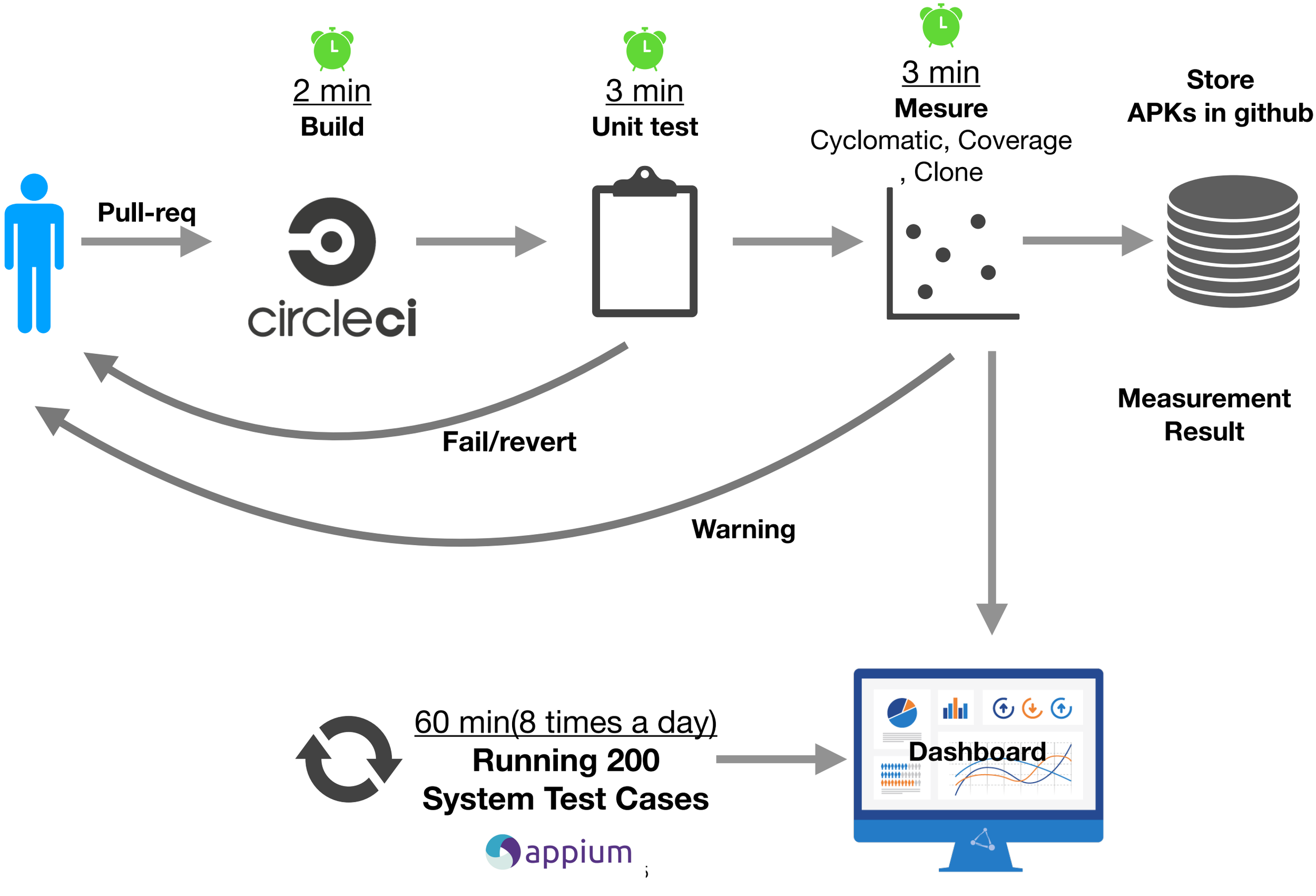
事例) Testing Strategies



60 min(10 times a day)
Running 400
System Test Cases

You Write code,
You Test code by JUnit as **100% Coverage**
(only you changed one, not for legacy code)

事例) テストスピード


2 min
Build
3 min
Unit test
3 min
Mesure
Cyclomatic, Coverage
, Clone
Store
APKs in github
Pull-req
circleci
Fail/revert
Warning
Measurement
Result
60 min(8 times a day)
Running 200
System Test Cases
appium
Dashboard

本日の主題

- Shift Left（上流品質担保）しなければ、市場バグ流出はなくならない

- **正しく**Shift Leftすれば、あなたの仕事は圧倒的に減る

Next Step：まずはアクションを起こしてみませんか？

- 新規に書く関数だけでも、単体テスト書いてCI/CDフレームワークに入れませんか？Commitごとのビルド＆API test
- 品質分析はどうでしょうか？なぜ後半にたくさんのバグを見つかってるか分析してみませんか？分析コストはかかりますが、総バグ数は半減すると思いますが。

Appendix

Testing Scale at Google

- 4.2 million individual tests running continuously
  - Testing runs before and after code submission
- 150 million test executions / day (averaging 35 runs / test / day)
- Distributed using internal version of bazel.io to a large compute farm
- Almost all testing is automated - no time for Quality Assurance
- 13,000+ individual project teams - all submitting to one branch
- Drives continuous delivery for Google
- 99% of all test executions pass

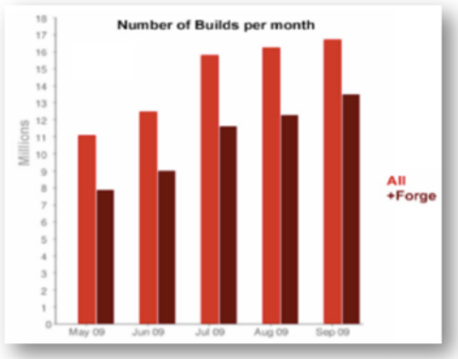
Google

SKIP

Testing at Google

Test & Build System Fast Facts

- Recall, Google's emphasis on speed – 20+ code changes occur per minute
  - Code churn - **50%** of code changes every month
- 65k builds/day
- 20M builds/year



Source: http://googletesting.blogspot.com/2010/04/googles-innovation-factory-and-how.html

Testing at Google

Test & Build System Fast Facts

- 120k test suites in code base
- 7.5M test suites run/day
- 120M individual tests run/day
- Every affect test run after each code change
- Tests run on all major OS/browser pairs

SKIP

Quality
Cost
Deliver

2000年でQCDの研究開発はすでに終わっている

SKIP

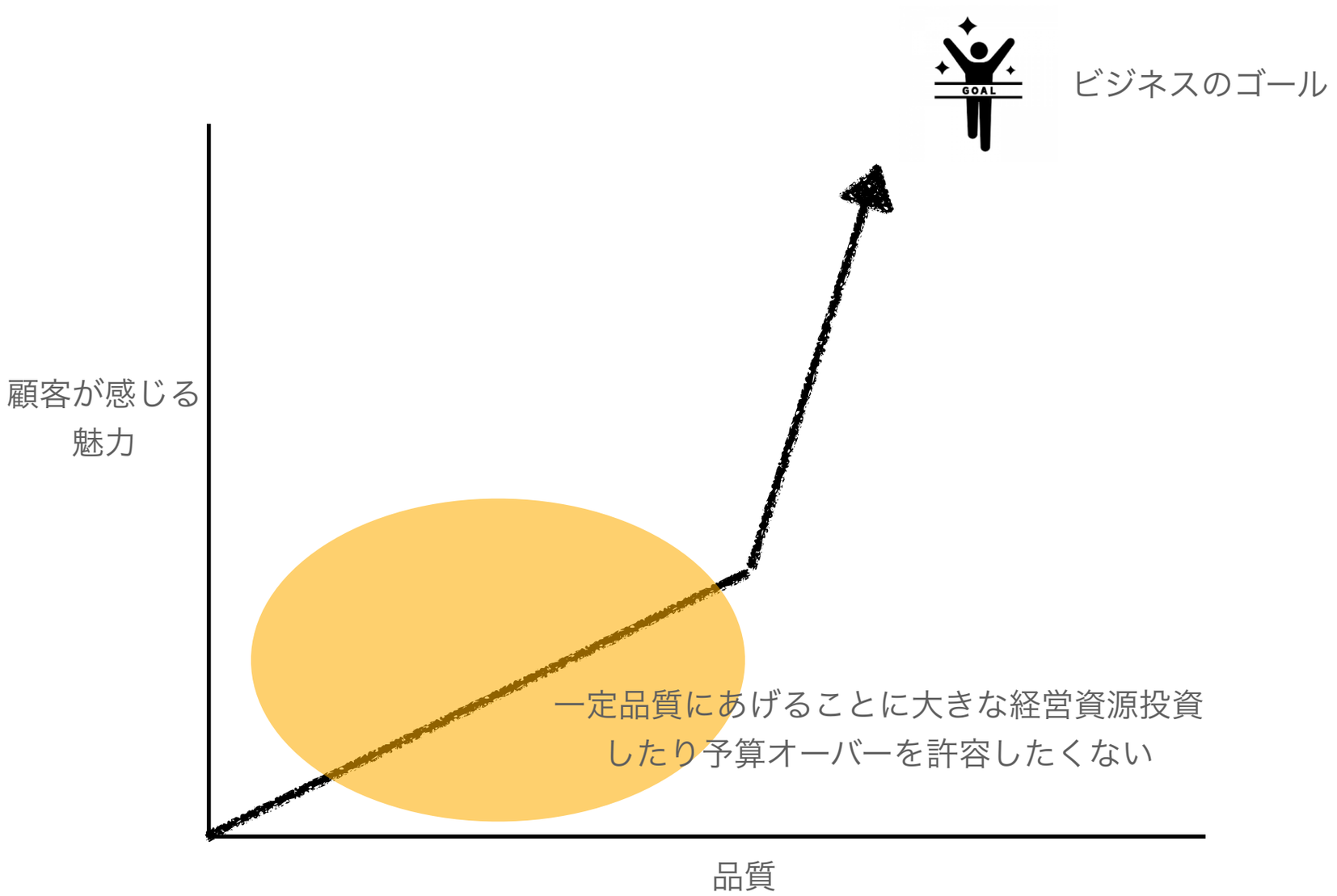
QCD+

QCDはとっとと終わらせて＋にいきませんか？

でも今日は＋の話ではなくQCDをとっとと終わらせる話

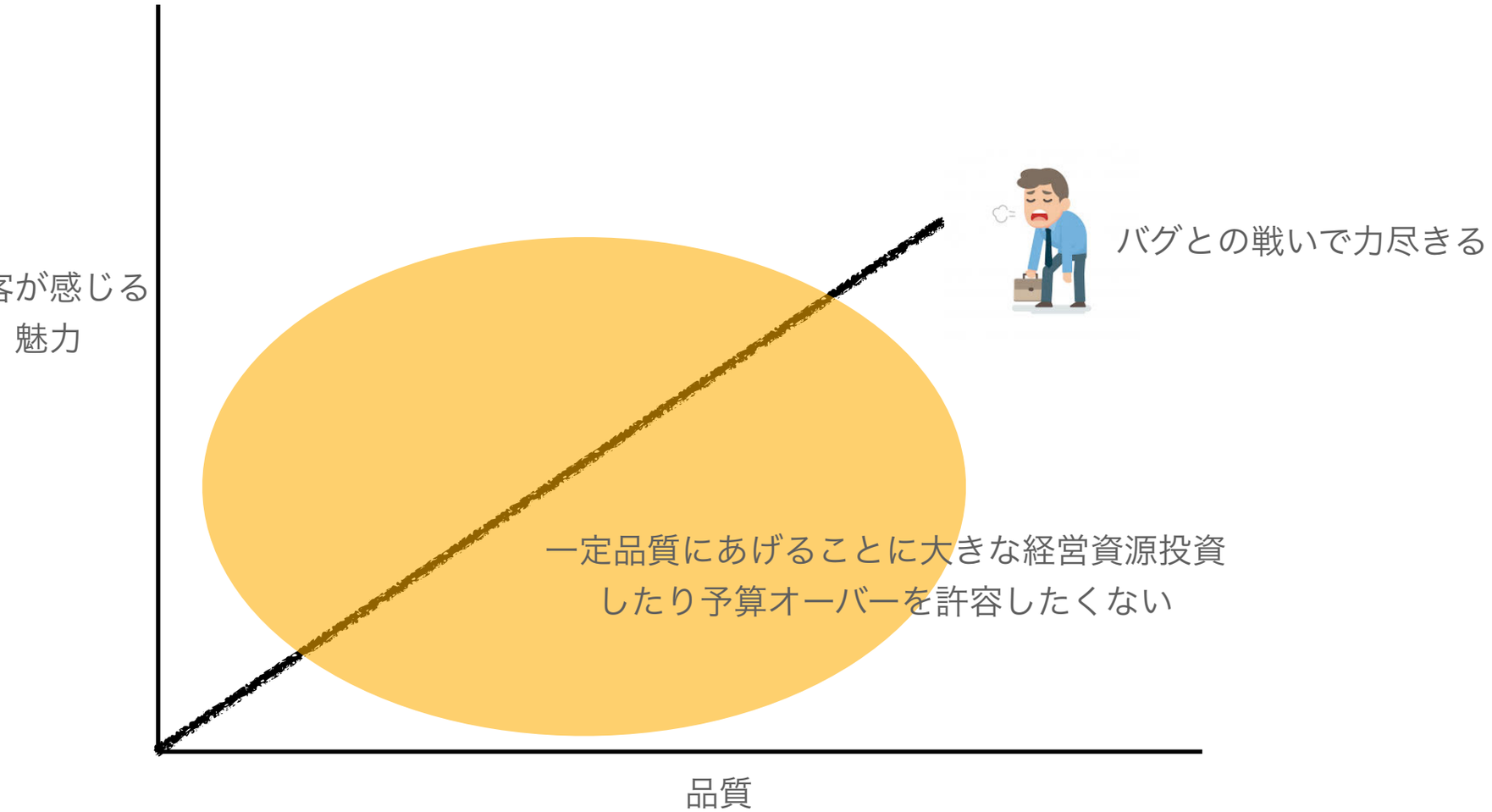
SKIP

多くのIT経営者の考え
（やめちゃったけどOさんもそう言っていた）



SKIP

多くの会社


顧客が感じる
魅力
品質
バグとの戦いで力尽きる
一定品質にあげることに大きな経営資源投資
したり予算オーバーを許容したくない

SKIP