

質とスピード

～ソフトウェア開発の典型的な誤解を解く～

(2020 秋バージョン)

Nov 20, 2020 @ JaSST'20 Kyushu 基調講演

和田 卓人 (@t_wada)

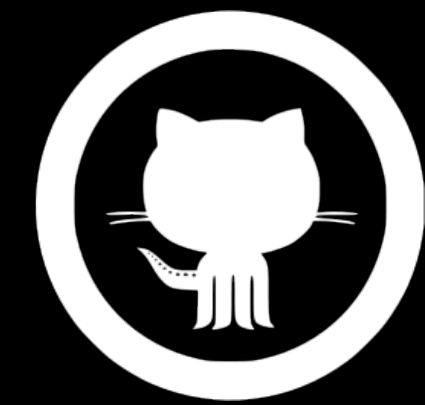
自己紹介



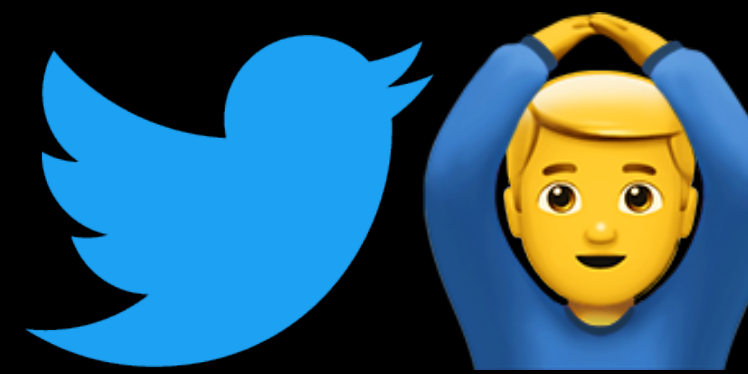
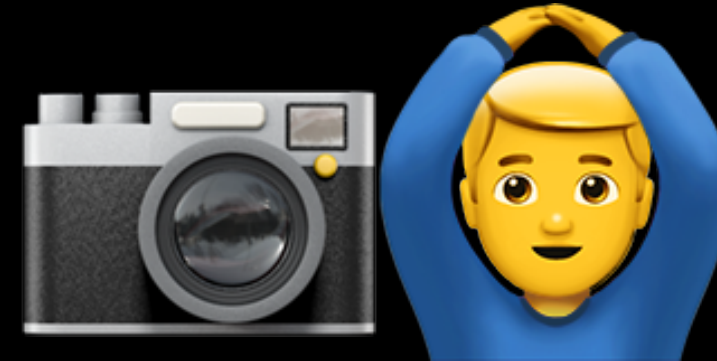
t-wada



t_wada

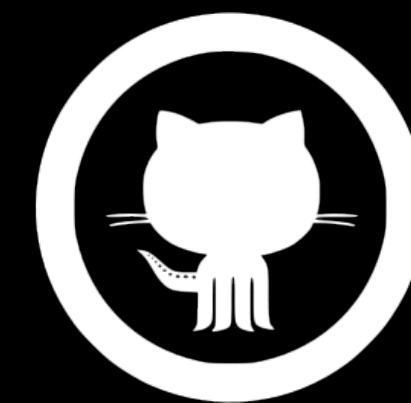


twada



#jasstkyushu

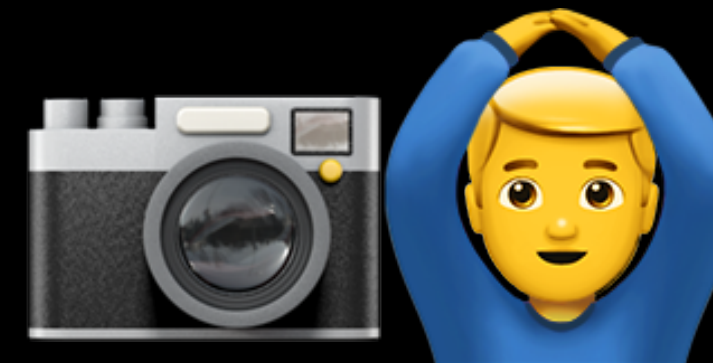
自己紹介



t-wada

t_wada

twada



#jasstkyushu

技術書の出版に関わっています



よろしくお願いします



illustrated by @nty_mno



#jasstkyushu

荒ぶる四天王があらわれた！



時間



予算



品質



スコープ

与えられた時間に対してやるべきことが多すぎる場合、どうすべきか？

- a) スコープを削る
- b) もっと人を増やす
- c) リリース日を延期する

👉 **d) 品質を犠牲にする**

『アジャイルサムライ』 p.86



品質を犠牲にすればスピードは得られる？

- ・短期的には得られる
- ・中期的には逆効果になる
- ・長期的には致命傷になる

「短期的/中期的」の境目は？

ここでいう「品質」とは？

「短期的/中期的」の境目は？

ここでいう「品質」とは？

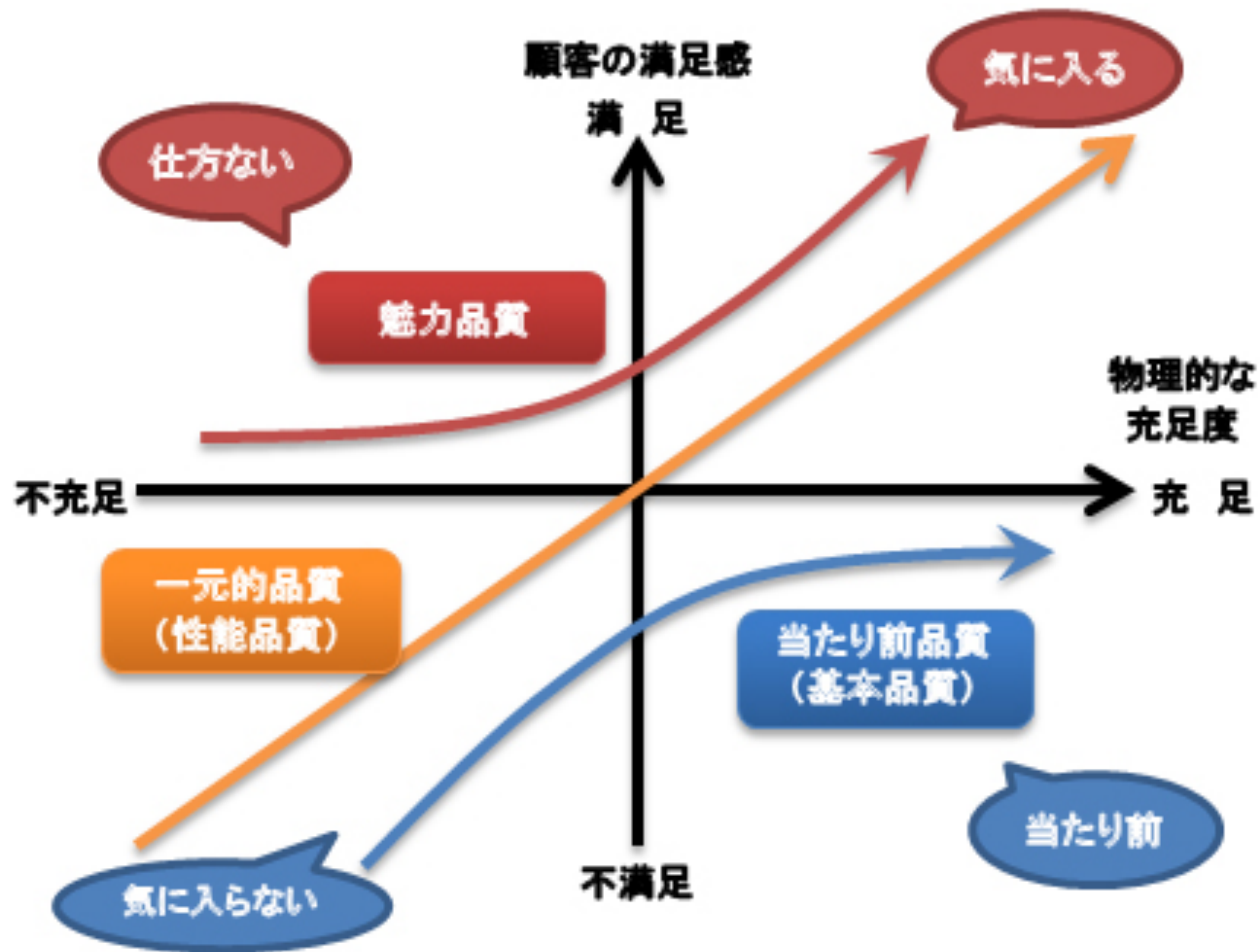


“品質とは誰かにとっての価値である”

-- Gerald Weinberg

"ilities"			
Accessibility	Functionality	Portability	Transactionality
Adaptability	Installability	Recoverability	Understandability
Administrability	Integrability	Reliability	Upgradeability
Analyzability	Internationalizability	Replaceability	Usability
Auditability	Interoperability	Reusability	... and lots more!
Availability		Scalability	
Calibrability		Security	
Changeability		Sensitivity	
Compatibility	Marketability	Stability	
Configurability	Maturity	Suitability	
Exploitability	Modifiability	Supportability	
Extensibility	Operability	Testability	
Flexibility	Personalizability	Traceability	

狩野モデル



- ・外部品質と内部品質
- ・利用時の品質と製品品質
- ・機能要件と非機能要件

External Quality:

Correctness, Usability, Efficiency, Reliability,
Integrity, Adaptability, Accuracy, Robustness

Internal Quality:

Maintainability, Flexibility, Portability,
Re-usability, Readability, Testability,
Understandability

ソフトウェアの品質を外部指標で特徴づける人は多い。正しいことをする、バグがない、速い、などだ。だが、それらはより深い原因の症状にすぎない。

本書で説明するソフトウェアの品質は**内部品質**である。内部品質を作り込んだ結果として、外部品質として定義される特性の実現に近づくことができる。**内部品質は結果ではなく原因**であり、良いソフトウェアが備えているべきものだ。

『レガシーコードからの脱却』 p.58



内部品質 (Internal Quality)

External Quality:

Correctness, Usability, Efficiency, Reliability,
Integrity, Adaptability, Accuracy, Robustness

Internal Quality:

自分たちの射程内にある
内部品質の方を犠牲にするという
判断をしがちなのではないか

Maintainability, Flexibility, Portability,
Re-usability, Readability, Testability,
Understandability

質とスピード

～ソフトウェア開発の典型的な誤解を解く～

和田 卓人 (@t_wada)

内部品質とスピード

～ソフトウェア開発の典型的な誤解を解く～

和田 卓人 (@t_wada)



ではどのような
内部品質を
犠牲に捧げたのか

内部品質 (Internal Quality)

External Quality:

Correctness, Usability, Efficiency, Reliability,
Integrity, Adaptability, Accuracy, Robustness

Internal Quality:

Maintainability, Flexibility, Portability,
Re-usability, Readability, Testability,
Understandability

保守性 (Maintainability)

External Quality:

Correctness, Usability, Efficiency, Reliability,
Integrity, Adaptability, Accuracy, Robustness

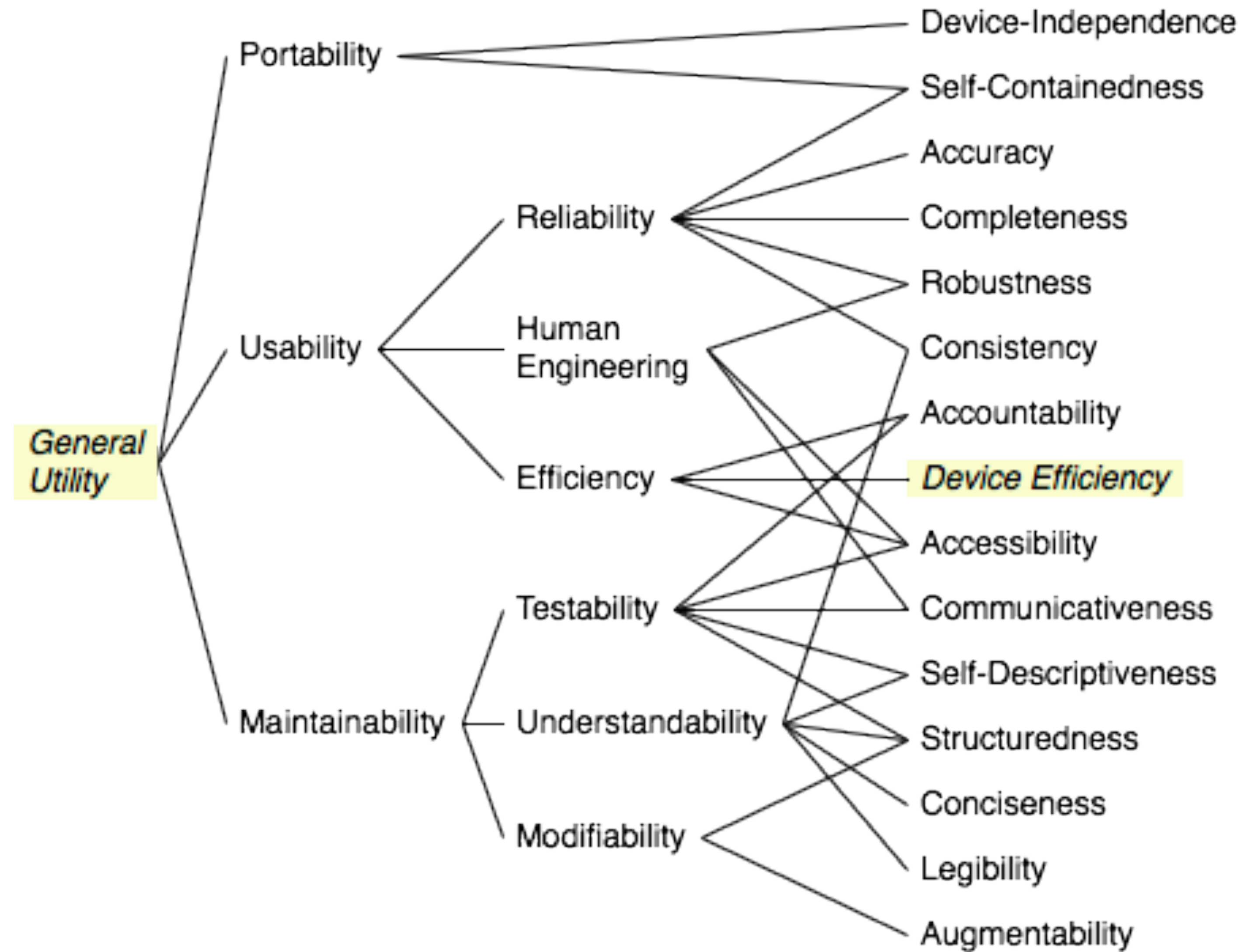
Internal Quality:

Maintainability, Flexibility, Portability,
Re-usability, Readability, Testability,
Understandability

保守性

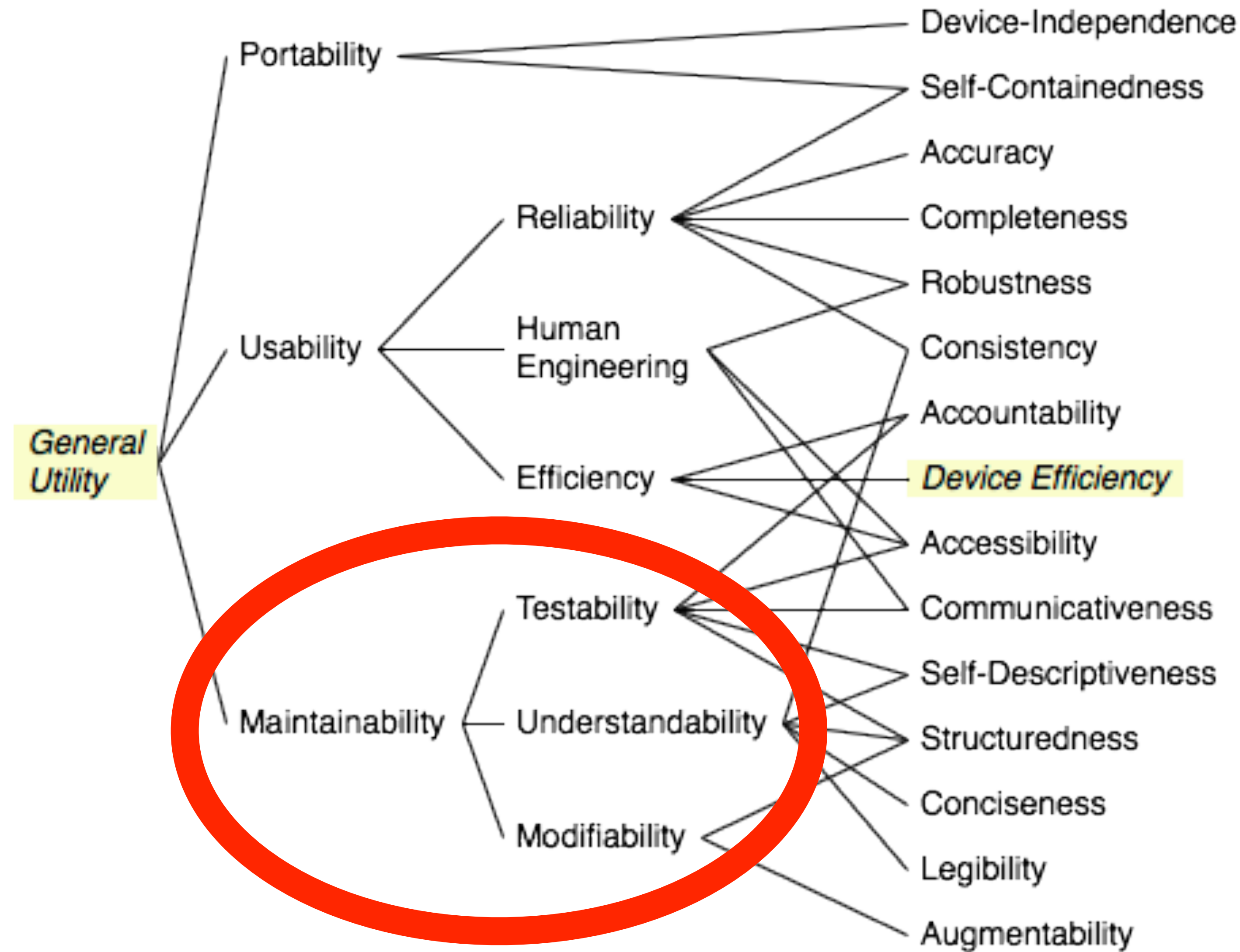
Maintainability

Boehm, Brown, and Lipow's 23 Quality Characteristics (1976)

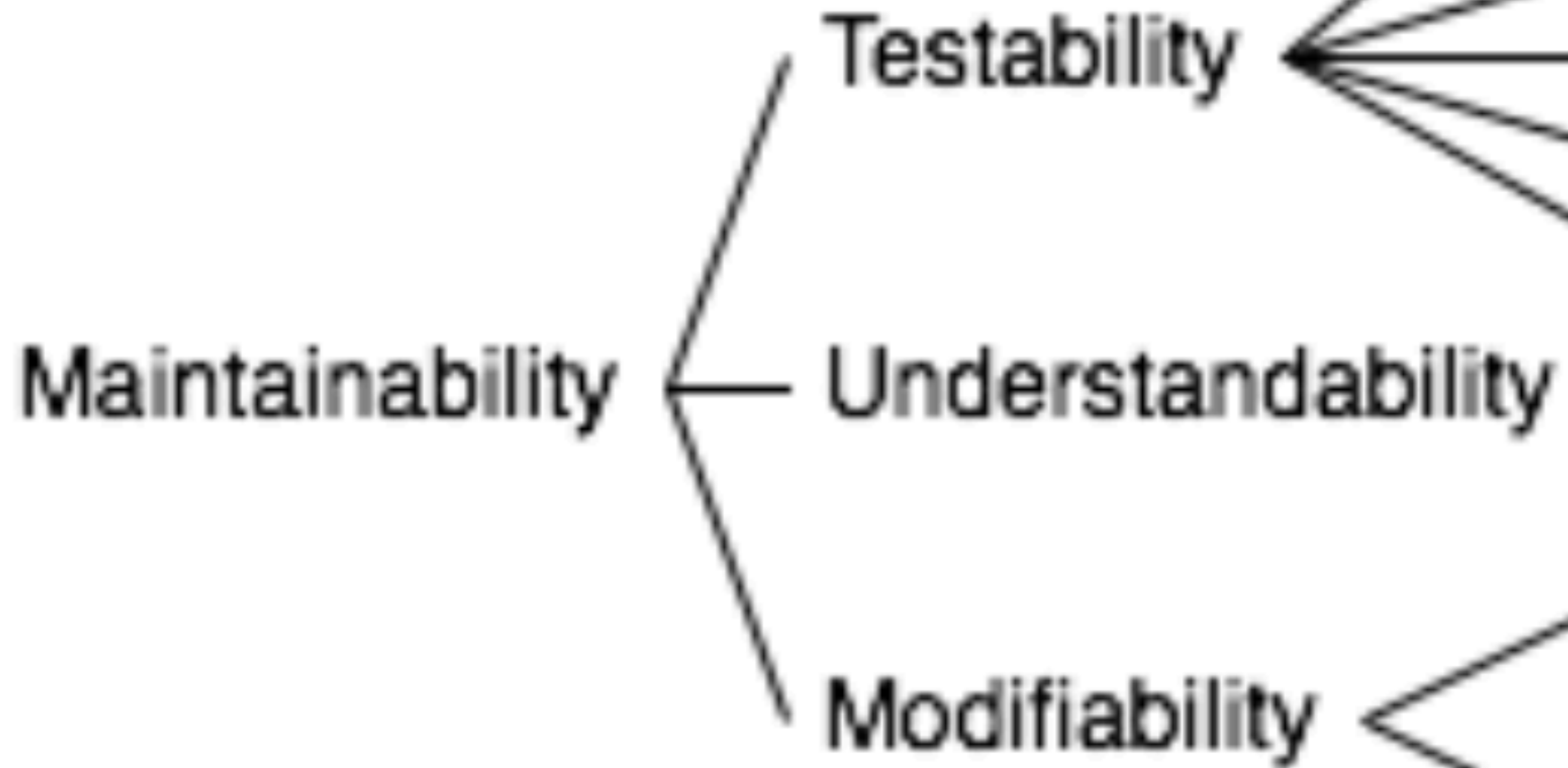


Boehm+Brown+Lipow1976 fig.1, corrected to match qualities named in text
"General Utility" and "Device Efficiency" are not defined there

Boehm, Brown, and Lipow's 23 Quality Characteristics (1976)



Boehm+Brown+Lipow1976 fig.1, corrected to match qualities named in text
"General Utility" and "Device Efficiency" are not defined there



- ・Testability: テスト容易性
- ・Understandability: 理解容易性
- ・Modifiability: 変更容易性

自動テストにおけるテストビリティ

- ・ 実行円滑性 (Operability)
 - ・ 自動実行が容易でかつ高速
- ・ 観測容易性 (Observability)
 - ・ テスト結果の取得・期待値比較を自動化しやすい
- ・ 制御容易性 (Controllability)
 - ・ 事前状態を制御し、テスト対象を自動操作しやすい
- ・ 分解可能性 (Decomposability)
 - ・ テスト対象の切り出し、テスト環境への部分置換が容易



内部品質とスピード

～ソフトウェア開発の典型的な誤解を解く～

和田 卓人 (@t_wada)



保守性とスピード

～ソフトウェア開発の典型的な誤解を解く～

和田 卓人 (@t_wada)

保守性を
犠牲に捧げると
どうなるか

「あとでクリーンにすればいいよ。先に市場に出さなければ！」

開発者はそうやっていつもごまかす。だが、**あとでクリーンにすることはない。市場からのプレッシャーは止まらない**からだ。「先に市場に出さなければ」ということは、後ろに競合他社が大勢いるということである。競合他社に追い抜かれないためには、これからも走り続けるしかない。

その結果、開発者はモードを切り替えることができない。次の機能、また次の機能、またまた次の機能を追加することになり、コードをクリーンにすることまで手が回らない。

そして、**崩壊が始まる。生産性がゼロに近づいていく。**

『Clean Architecture』 p.38





nekoya

@nekoya

フォローする



Web系企業でよく言う「今は大事な時期だからスピード優先で、コードの質は下げても構わん」みたいなもの、会社が存続している限りスピード優先なので、真に受けるとクソコードしか存在しない会社になるから常にちゃんとしたものを書いた方がいい。

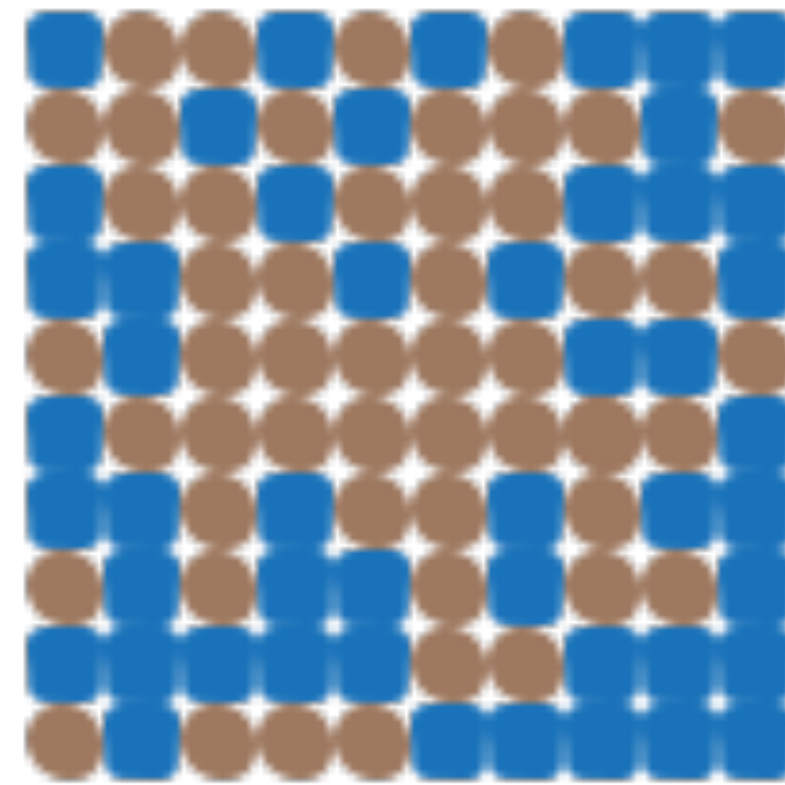
2:16 - 2014年8月6日

1,365件のリツイート 1,103件のいいね



保守性の低さがもたらすもの

*If we compare one system with **a lot of cruft**...*

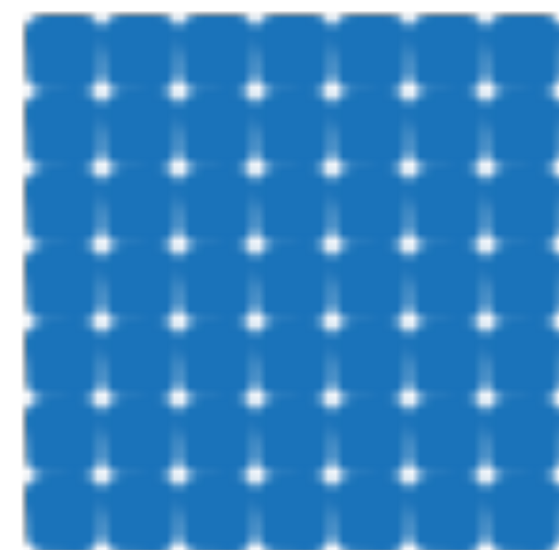


the cruft means new features take longer to build



this extra time and effort is the cost of the cruft, paid with each new feature

...to an equivalent one without



free of cruft, features can be added more quickly

**ひとつひとつの変更に
余計な時間がかかる**

保守性を犠牲にすればスピードは得られる？

- ・短期的には得られる
- ・中期的には逆効果になる
- ・長期的には致命傷になる

では
スピードを落とせば
保守性は上がる？

**V**

@voluntas

フォローする



経験上、十分時間を与えたから良いコードを書いてくる人は、スピード優先でも良いコードを書いてくる。クソコードを書いてくる人は結局クソコードしか書いてこない。

このツイートは表示できません @drillbits

毎度スピード優先だからとか言ってクソコード投げてるひとに十分な時間を与えたら果たしていいコードが返ってくるのだろうかという

18:14 - 2018年7月3日

546件のリツイート 771件のいいね



技術力のある人はある程度急いで作ったとしても一定以上の品質のコードを書き、意図的に品質を落としたとしても速度はあまり上がらない。

逆に、技術力が低い人が時間をかけて作ったとしてもその人の技術力以上の品質のコードは書けない。

クイック&ダーティの神話

今は急いでいるから、
早いけど将来困りそうなコードを書こう！



今は時間があるから
将来を見越したコードを書こう！



エンジニアリング
組織論への招待

不確実性に向き合う思考と組織のリファクタリング

Engineering
Organization
Theory

広木大地 著

技術的負債・経営との不和。
プロジェクトの理不尽。上がらない生産性。
そのすべての正体は
不確実性の扱い方の失敗にあった。

技術評論社

経営的な要求に合わせて、与えられた開発期間から柔軟に汚さと速さを選択するというような器用な芸当はほとんど不可能だといえます。これは、極めて基礎的なところであれば多少はできるかもしれませんが、実際には、ソースコードが汚くて遅い人もいれば、綺麗で速い人も多くいます。

『エンジニアリング組織論への招待』 p.255

つまり
トレードオフではない



チーフ

@katzchang

フォローする



何かにつけてトレードオフ構造を見出して
「要はバランス」という結論で満足してしま
う人を揶揄して「要はバランスおじさん」と
呼ぶことがあります。

8:18 - 2019年9月9日

120件のリツイート 411件のいいね



1

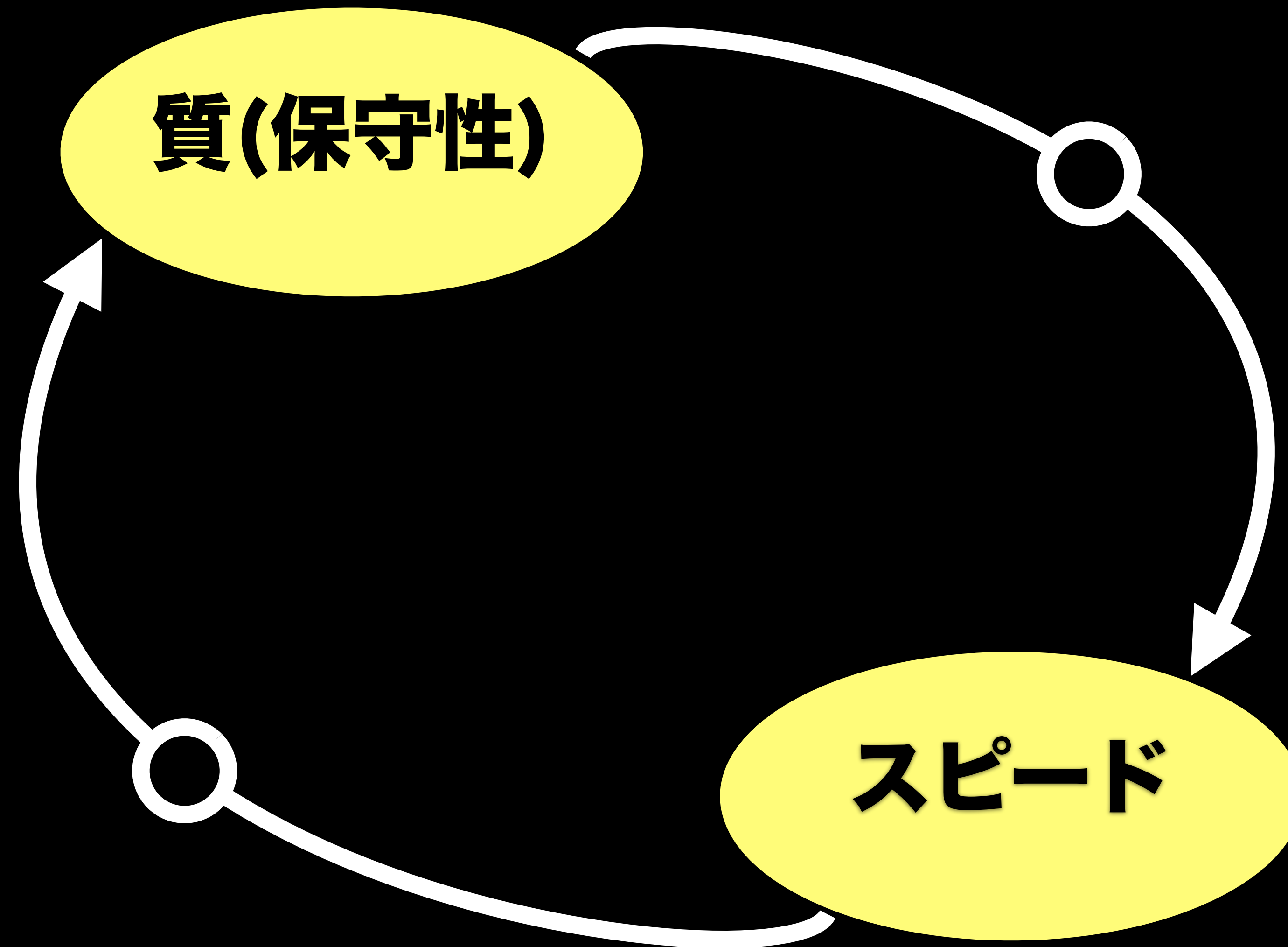


120



411

トレードオフと考えるのは典型的な誤解



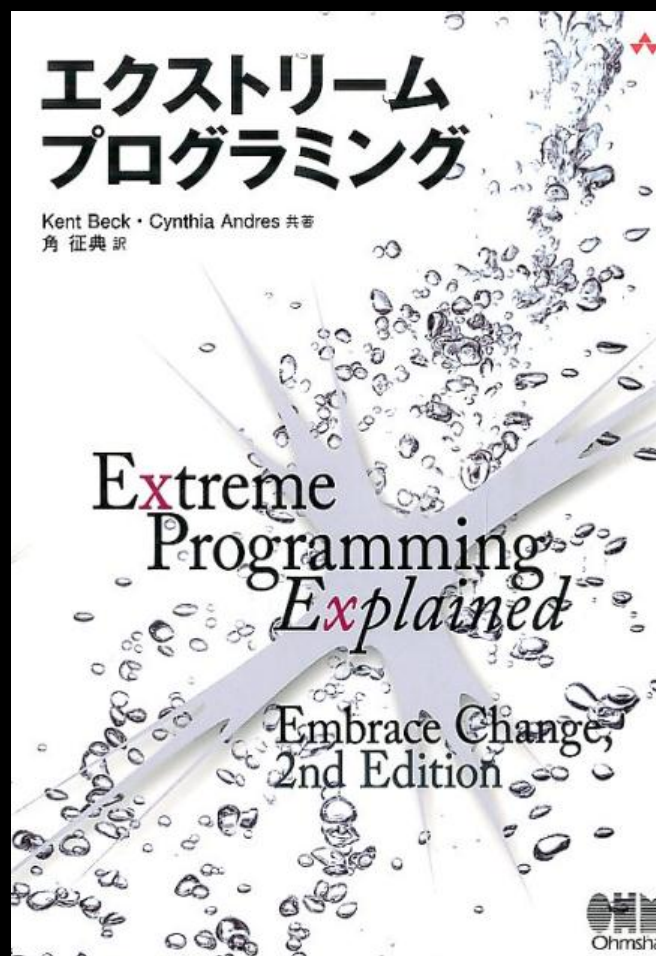
白丸付き矢印: 負の接続 = 根元が増えれば先は減る。根元が減れば先は増える

品質を犠牲にするのは、効果的なコントロール方法ではない。品質は制御変数ではない。

低品質を受け入れることで、プロジェクトが速くなることはない。
高品質を要求することで、プロジェクトが遅くなることもない。

むしろ品質を高めることで、デリバリーが高速になることが多い。
品質基準を下げてしまうと、デリバリーが遅くなり、
予測できなくなってしまう。

『エクストリームプログラミング』 p.30



先ほどの開発者のごまかしは、崩壊したコードを書けば長期的には遅くなるものの、短期的には速度が上がるという考え方にもとづいている。

このことを信じている開発者は、崩壊したコードを書くモードから、いずれどこかでクリーンにするモードに切り替われる、という（ウサギとカメの）ウサギのような自信を持っている。

だが、それは**事実誤認**である。

事実は、**短期的にも長期的にも、崩壊したコードを書くほうがクリーンなコードを書くよりも常に遅い。**

『Clean Architecture』 pp.38-39



短期的にも長期的にも、
崩壊したコードを書くほうが
クリーンなコードを書くよりも
常に遅い。

-- Bob Martin



『Clean Architecture』 p.39



最高の開発者がいちばんきれいな開発者であることに気づいたとき、私はびっくりした。速いプログラマーは雑なプログラマーだと思っていたからだ。

だが、実際は**正反対**だった。私が会った中で最速のプログラマーは、コードを扱いやすいように保つことに特に注意を払っていた。

『レガシーコードからの脱却』 p.49



コードを書く速さとコードのきれいさに関連があると認識した
あとでも、私はその2つのあいだの**因果関係**を見つけるのに時
間を要した。

コードの品質を高く保っていた「**にも関わらず**」速いのではない。
コードの品質を高く保っていた「**からこそ**」速いのだ。

このことを理解したら、ソフトウェア開発に対する
見方が変わった。

『レガシーコードからの脱却』 pp.49-50



コードの品質を高く保っていた
「**にも関わらず**」速いのではない。
コードの品質を高く保って
いた「**からこそ**」速いのだ。

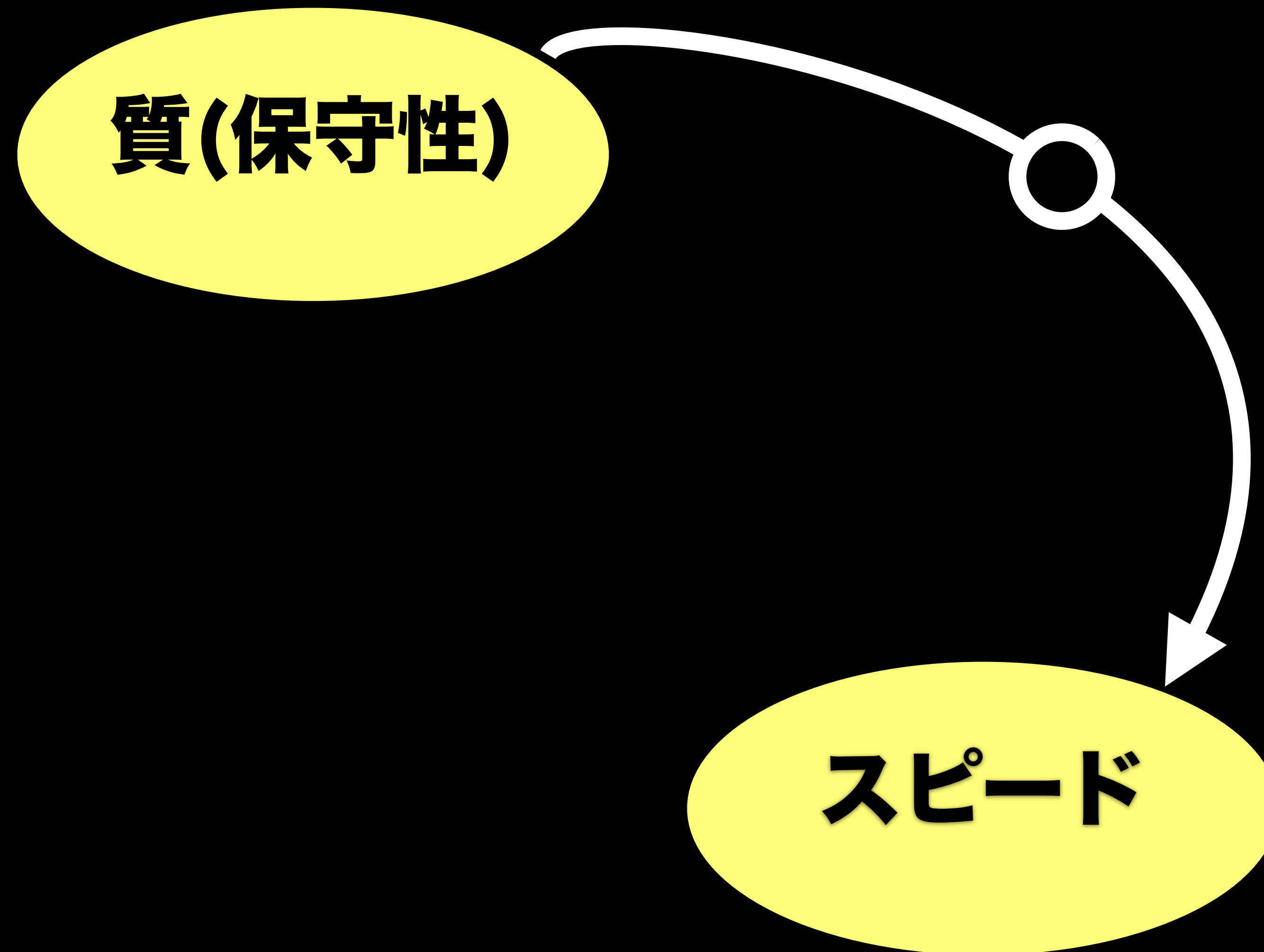
-- David Scott Bernstein



『レガシーコードからの脱却』
pp.49-50

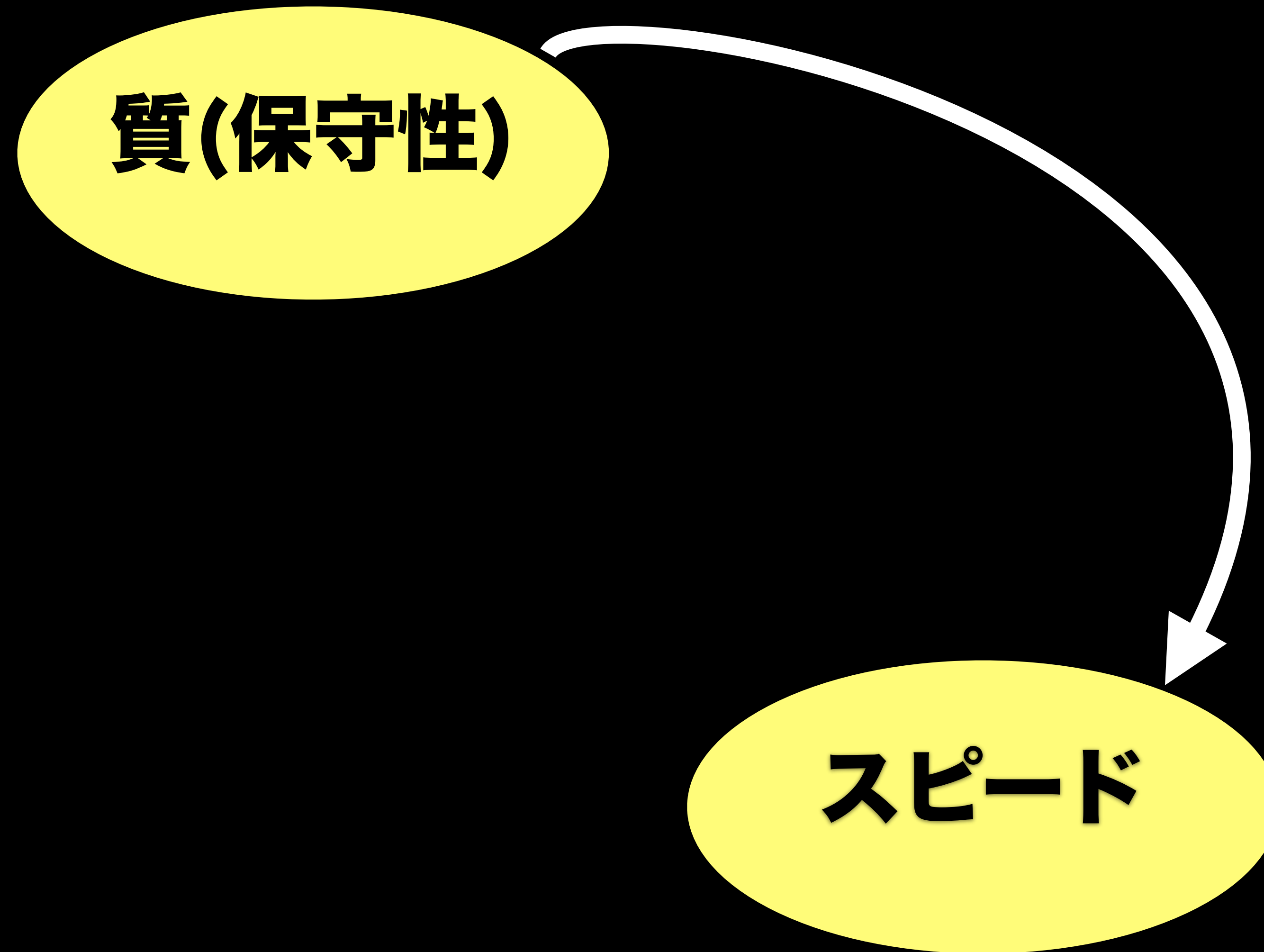


コードの品質を犠牲にしたから速いのではなく



白丸付き矢印: 負の接続 = 根元が増えれば先は減る。根元が減れば先は増える

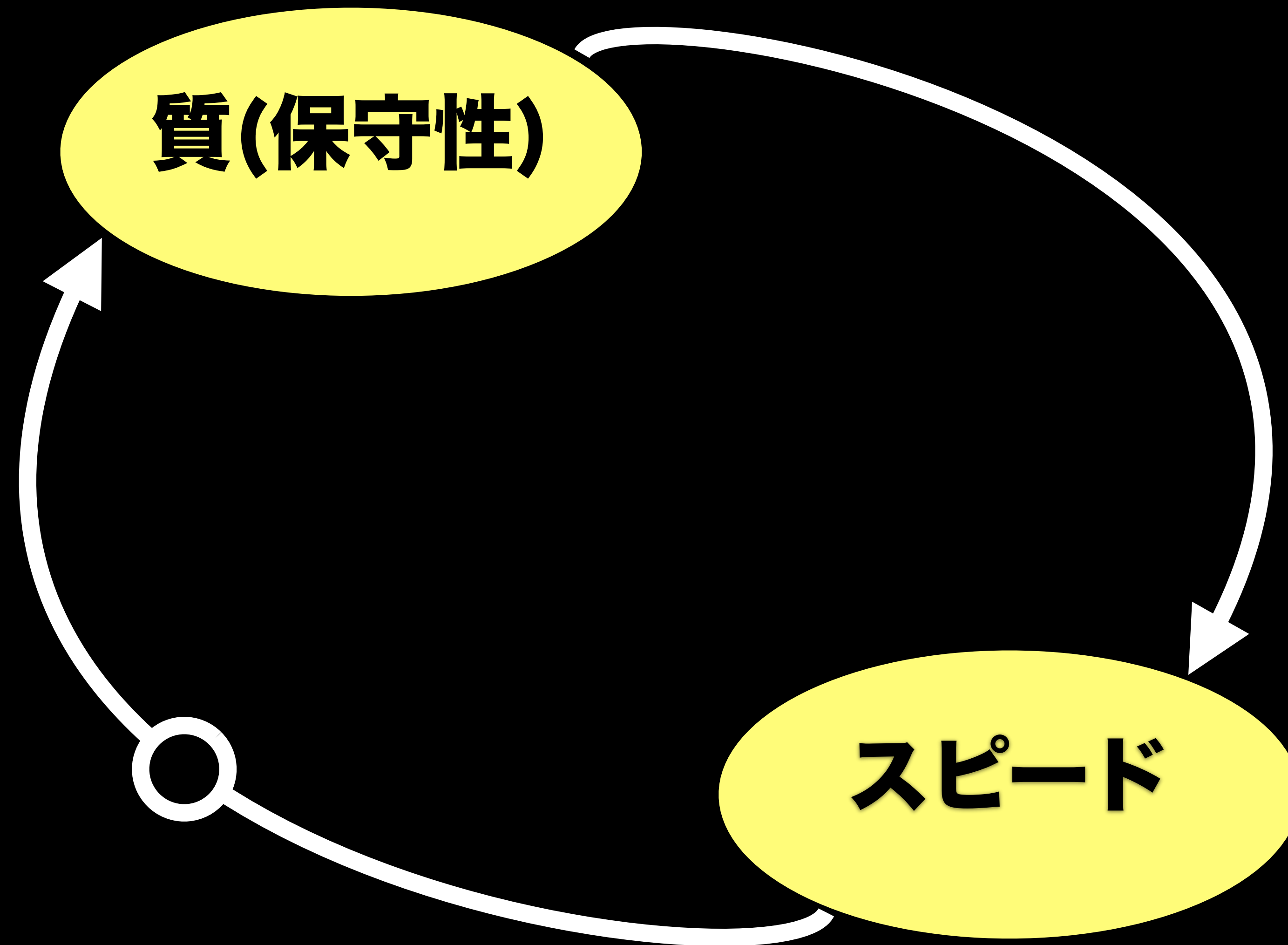
コードの品質を高く保っていた「からこそ」速い



矢印: 正の接続 = 根元が増えれば先も増える。根元が減れば先も減る

では
スピードから質への
影響はどうか

ではスピードから質(保守性)への影響はどうか



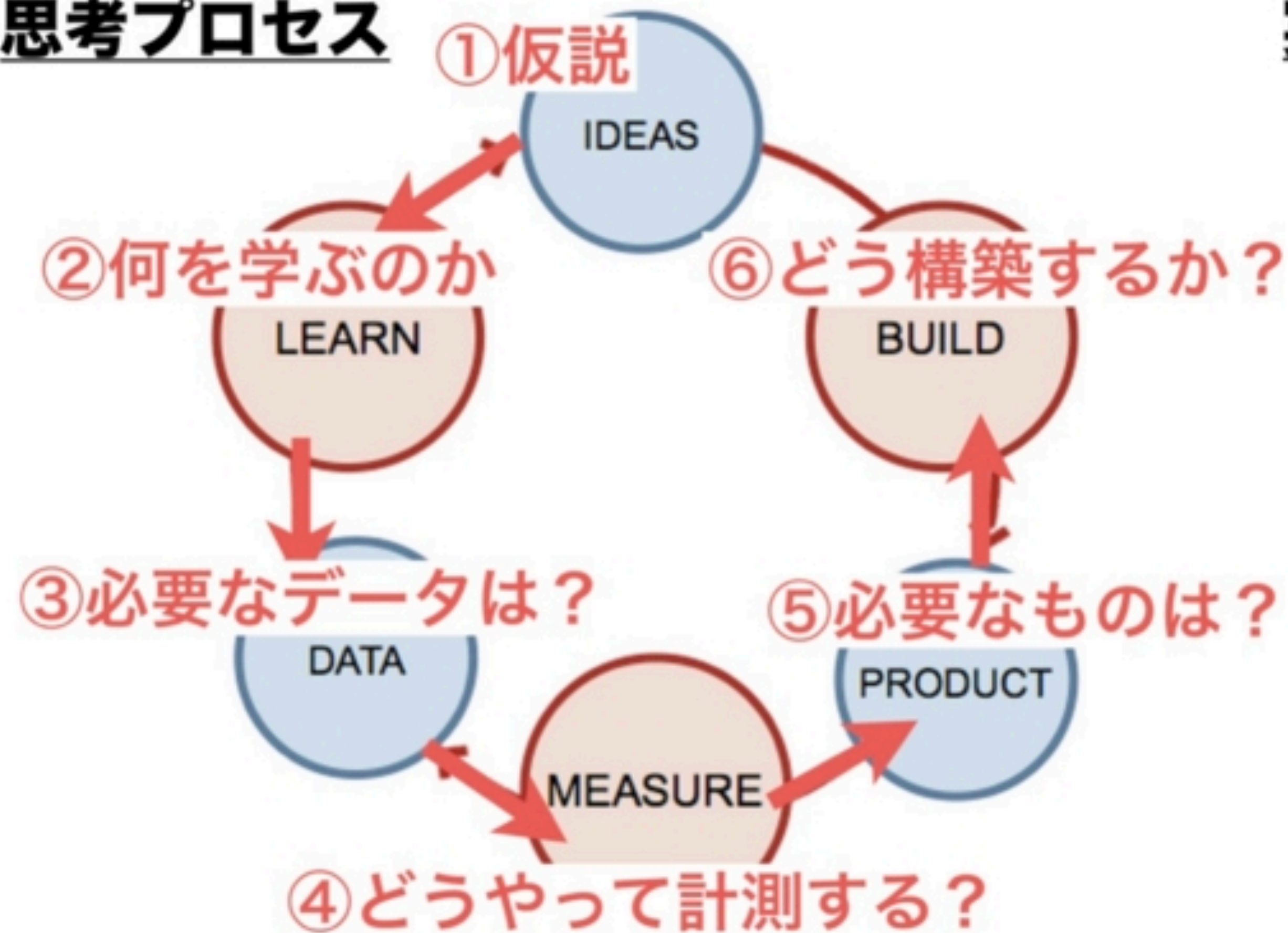
品質は劣化すればリードタイムの増加に跳ね返る

品質は悪いと基本的に手戻りを生むので速度に跳ね返る。手戻ってる時間は学びをうまない時間。品質を下げるという判断は学びの速度低下を許容すること。

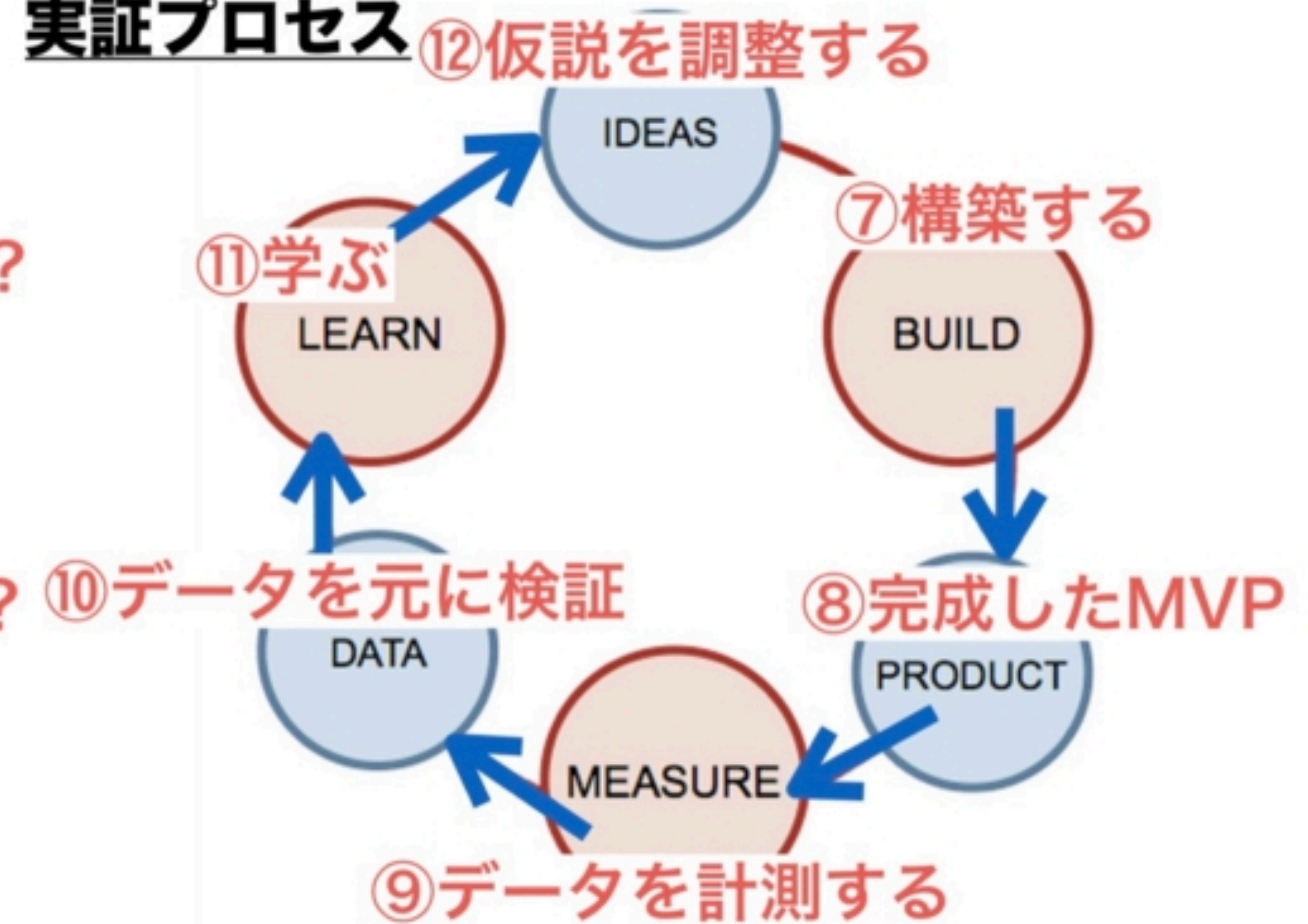
従来の鉄板(だと誤解されていた)だった品質すてて速度あげようは品質は劣化すれば手戻りが発生するだけで、結局はリードタイムの増加に跳ね返るのでやめましょう。

リードタイムが増加すると仮説検証プロセスが回らない

思考プロセス



実証プロセス



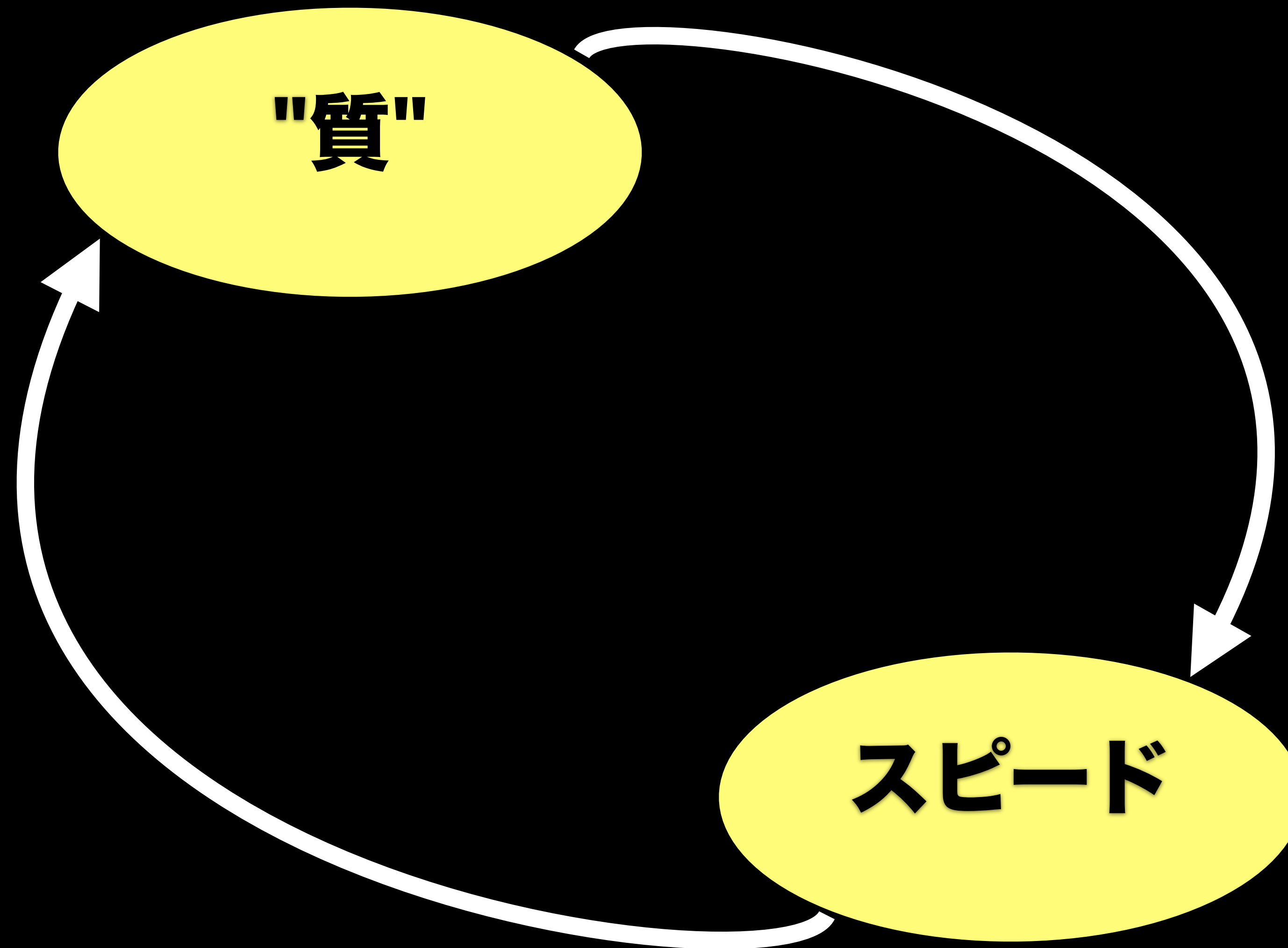
本当の関係

本当の関係

- ・内部品質がスピードを生み
- ・スピードが学びのループを生み
- ・学びのループが外部品質を生み
- ・外部品質が競争力を生み
- ・競争力が売上を生み
- ・売上が内部品質を育む

※良いものを作れば売れるというわけではないので、ここはそう簡単でも単純でもない

本当の関係



矢印: 正の接続 = 根元が増えれば先も増える。根元が減れば先も減る

「質v.s.スピード」という概念は根本的に間違っていると思う。

だって素早く開発をしなくては環境、あるいは自分の環境の理解の変化にソフトウェアがついてこれず、ソフトウェアが解決すべき問題が解決できなくなり、必然的に質が落ちてしまう。

逆に、質の高いソフトウェアを書かなくては、なにかある度にインフラが崩壊し、素早く開発をすることができなくなってしまった。インフラの崩壊は、やる気を削ぐので特にたちが悪い。

Evan Priestley



質v.s.スピードという二律背反の関係は、
局所的なものでしかない。

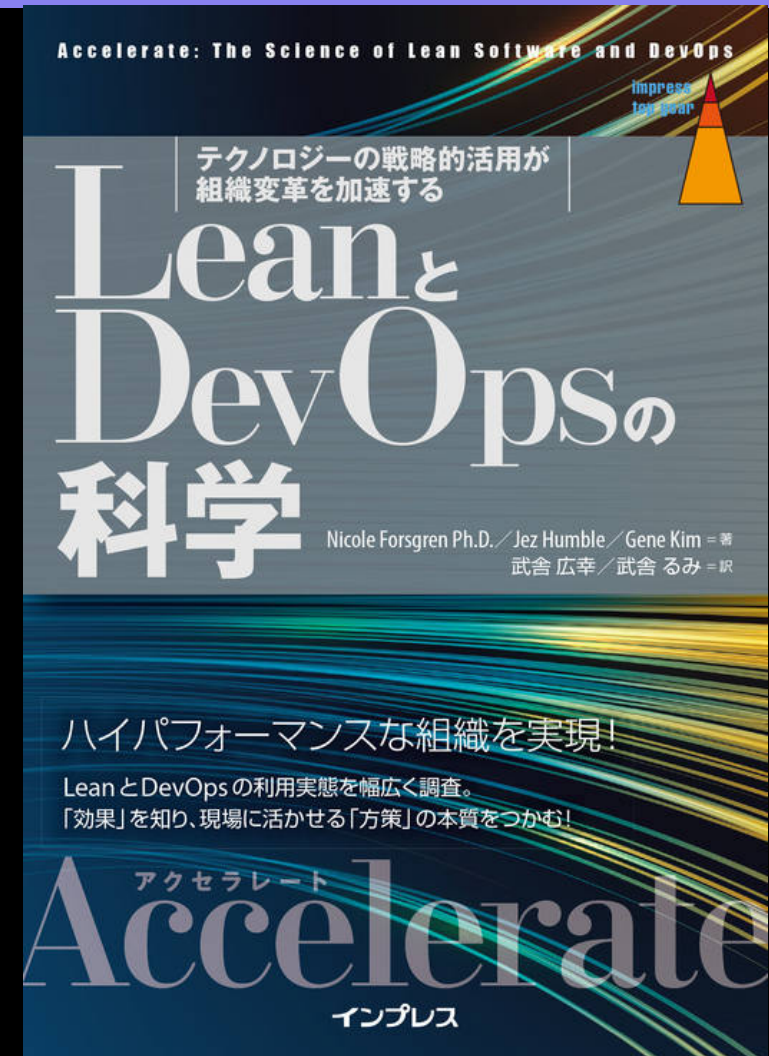
大域的には、片方を犠牲にした場合、知
らぬうちにもう一つも犠牲にしていると
いうことをお忘れなく。

Evan Priestley



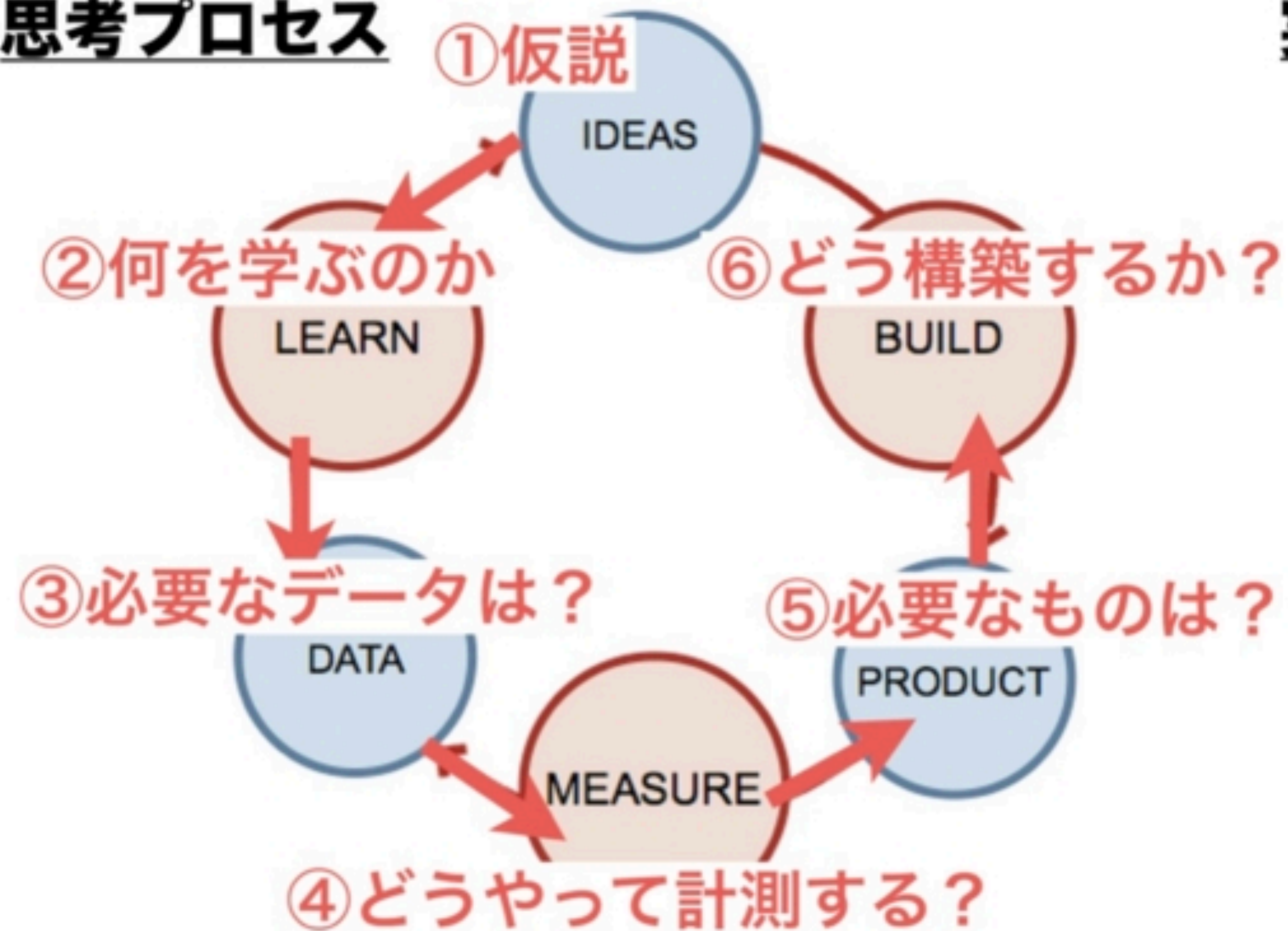
現代における質とスピード: 4つのキーメトリクス

- ・リードタイム
- ・デプロイ頻度
- ・MTTR(平均修復時間)
- ・変更失敗率

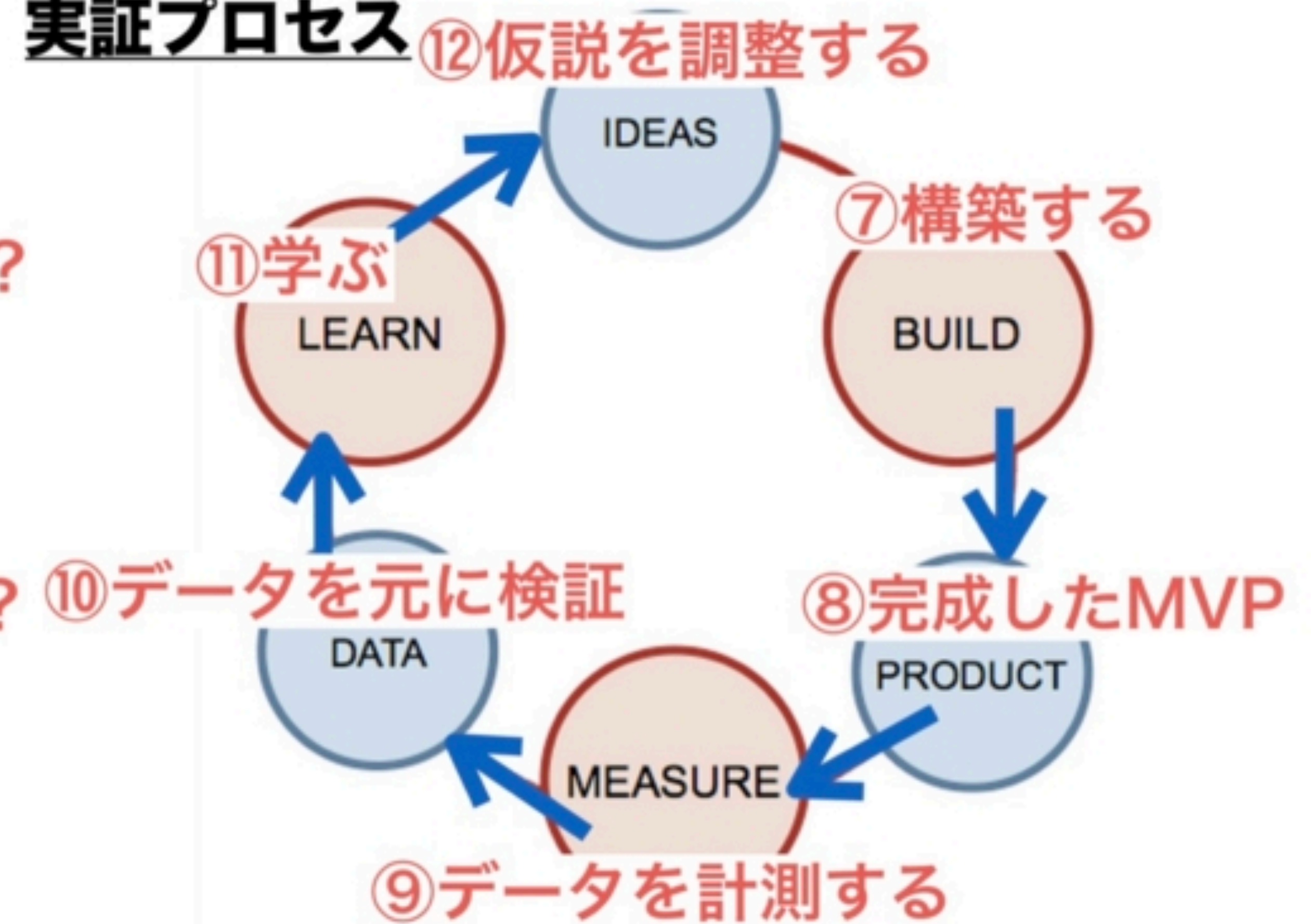


リードタイムとデプロイ頻度: 仮説検証プロセスを迅速に回す

思考プロセス

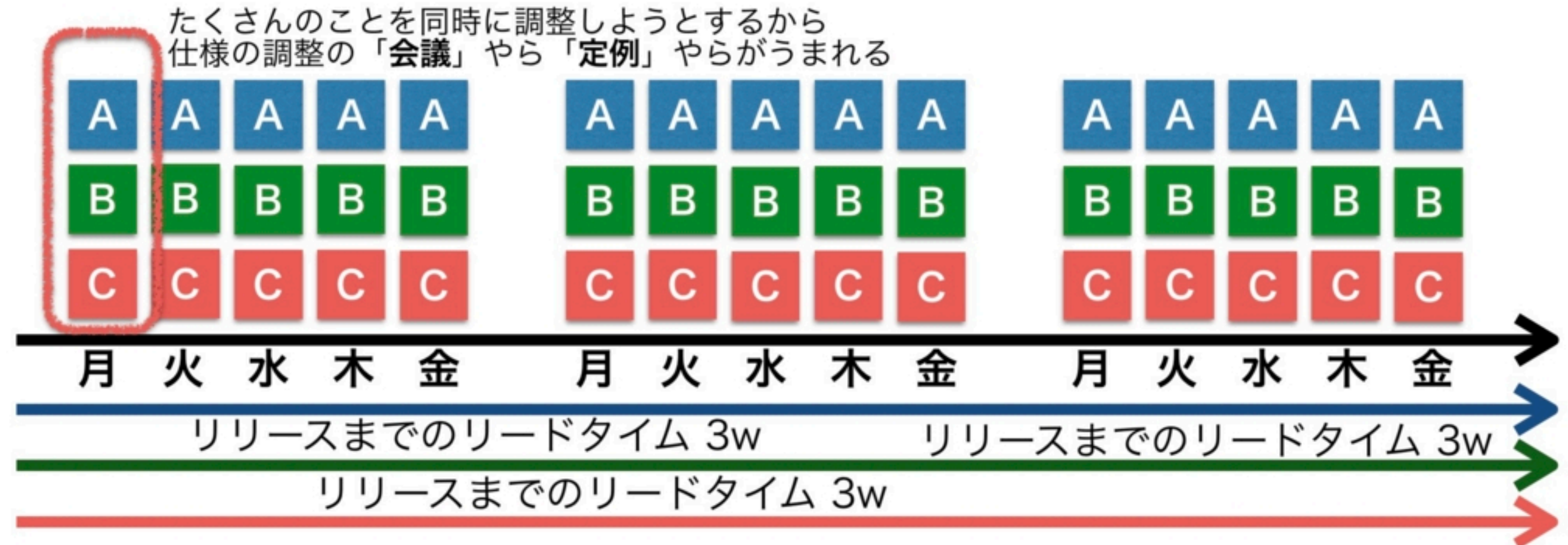


実証プロセス

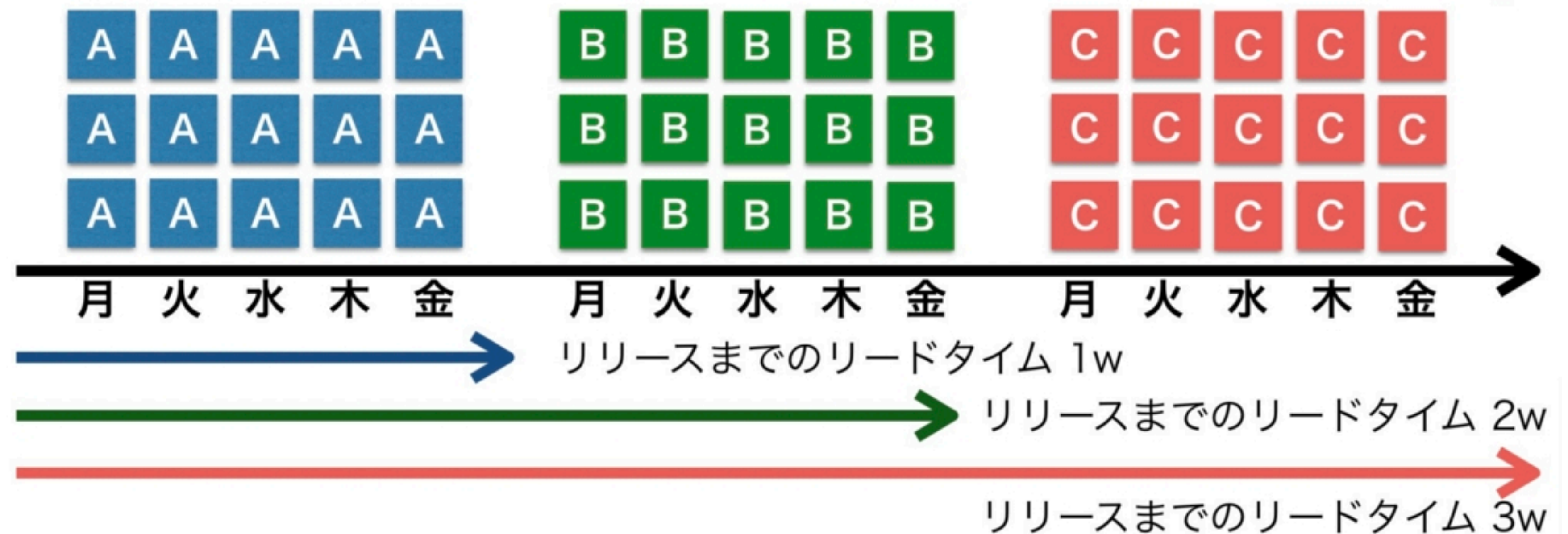


リードタイムとデプロイ頻度: フロー効率とリソース効率

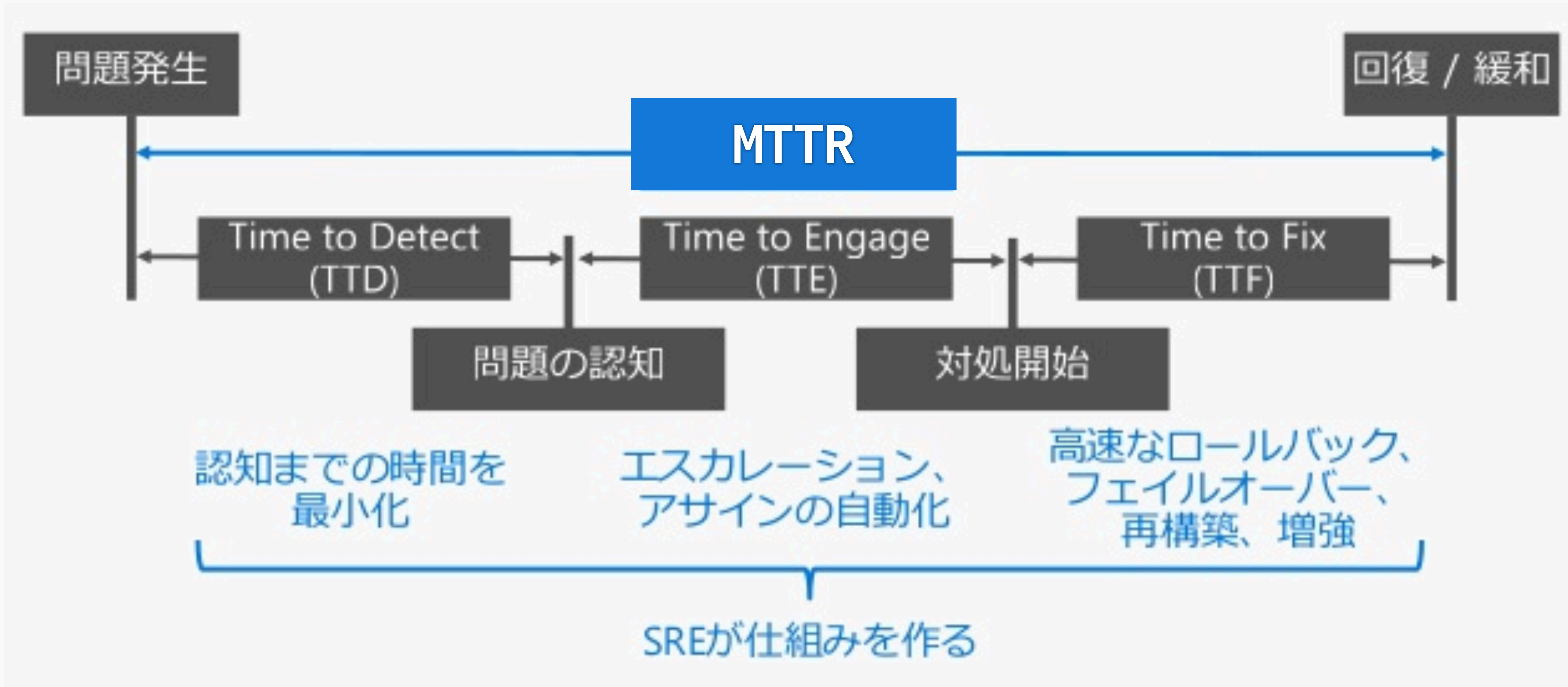
リソース効率



フロー効率



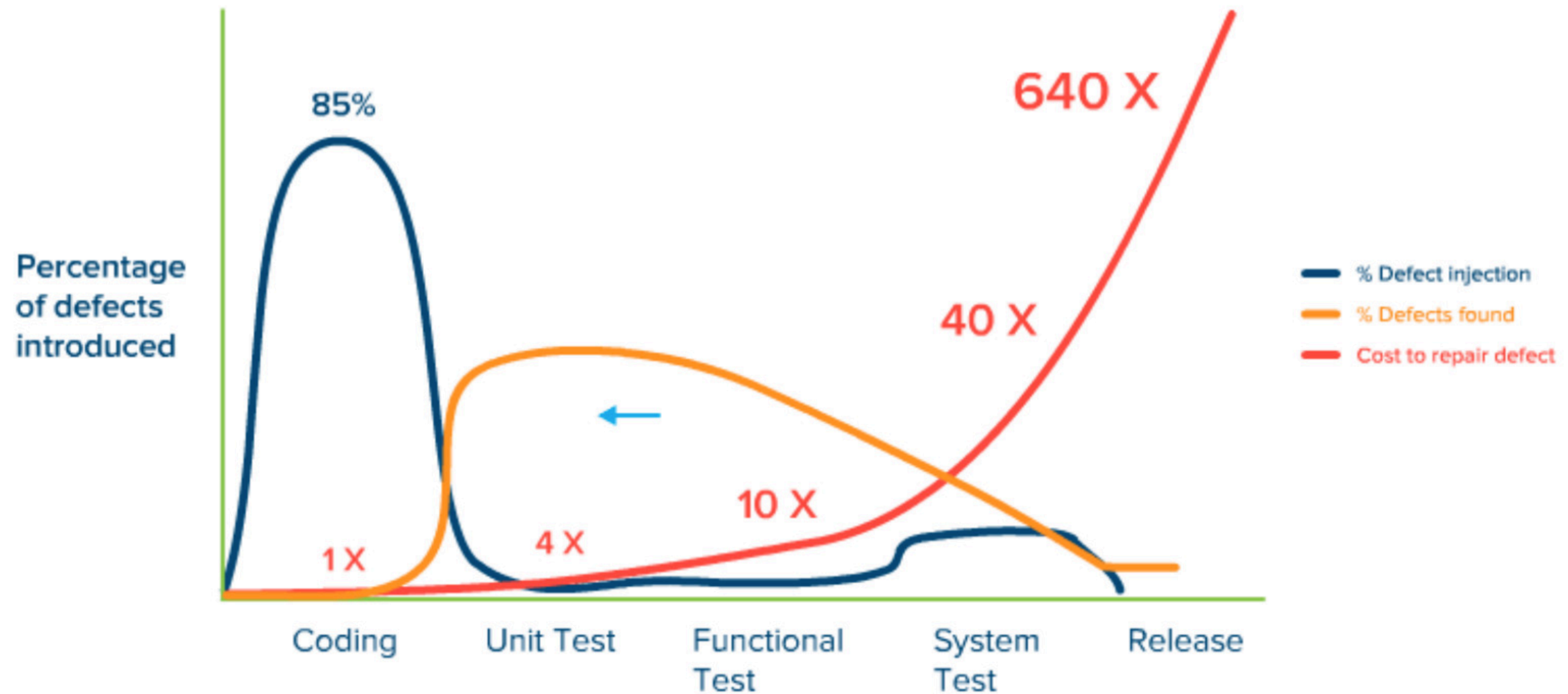
MTTR: Mean time to Recovery/Repair



MTBFを捨てるのではなく品質をプロセスで作り込む

問題発見の時期をシフトレフトし継続的に取り組む

つまりテスト自動化を始めとした技術基盤が欠かせない



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

ACCELERATE
State of DevOps
2019

Sponsored by



<https://cloud.google.com/devops/state-of-devops/>

4つのキーメトリクス: 2019年の調査 (3万件以上の回答をクラスター分析)

	エリート	ハイパフォーマー	ミディアム パフォーマー	ローパフォーマー
リードタイム	1日未満	1日から1週間	1週間から1ヵ月	1ヵ月から6ヵ月
デプロイ頻度	オンデマンド (1日複数回)	1日1回から週1回	週1回から月1回	1ヵ月から6ヵ月
MTTR	1時間未満	1日未満	1日未満	1週間から1ヵ月
変更失敗率	0 - 15%	0 - 15%	0 - 15%	46 - 60%

- ・開発速度と品質はトレードオフの関係ではない
- ・組織間の差はかなり大きく、さらに開いている (2016 - 2019)
- ・圧倒的な差は継続的デリバリやDevOpsへの組織的な投資の差

4つのキーメトリクス: エリートクラスとローパフォーマークラスの差

	エリート	エリートとローパフォーマーの差	ローパフォーマー
リードタイム	1日未満	106倍	1ヵ月から6ヵ月
デプロイ頻度	オンデマンド (1日複数回)	208倍	1ヵ月から6ヵ月
MTTR	1時間未満	1/2604	1週間から1ヵ月
変更失敗率	0 - 15%	1/7	46 - 60%

- ・開発速度と品質はトレードオフの関係ではない
- ・組織間の差はかなり大きく、さらに開いている (2016 - 2019)
- ・圧倒的な差は継続的デリバリやDevOpsへの組織的な投資の差

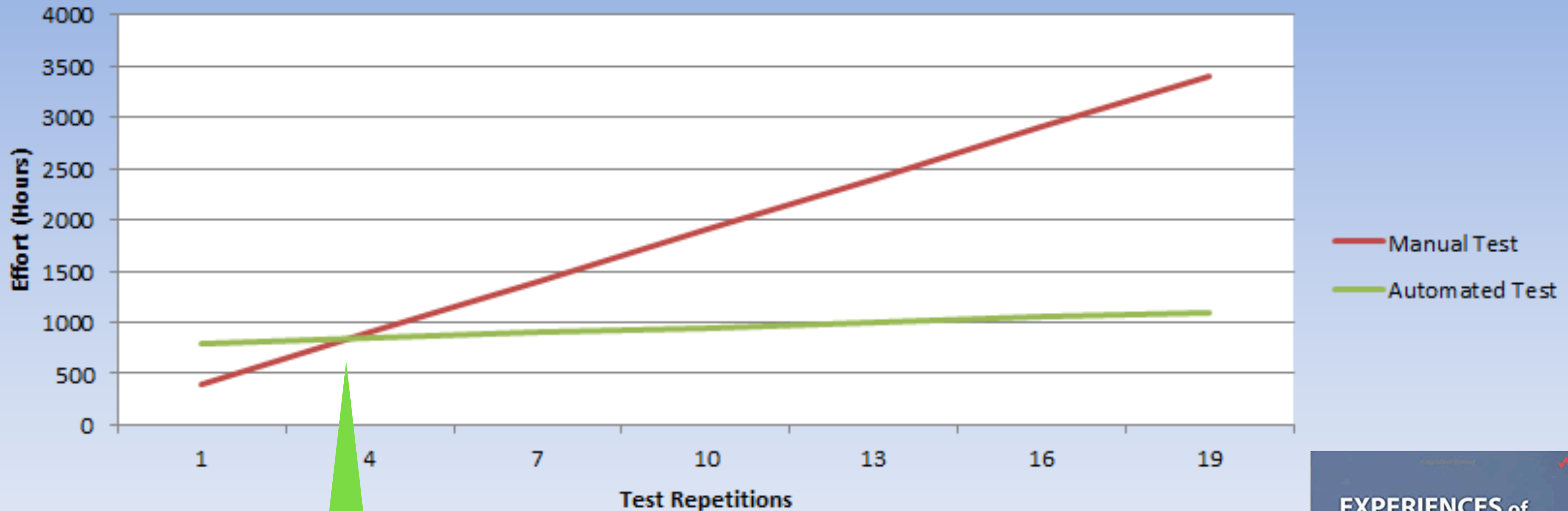
保守性を犠牲にすればスピードは得られる？

- ・短期的には得られる
- ・中期的には逆効果になる
- ・長期的には致命傷になる

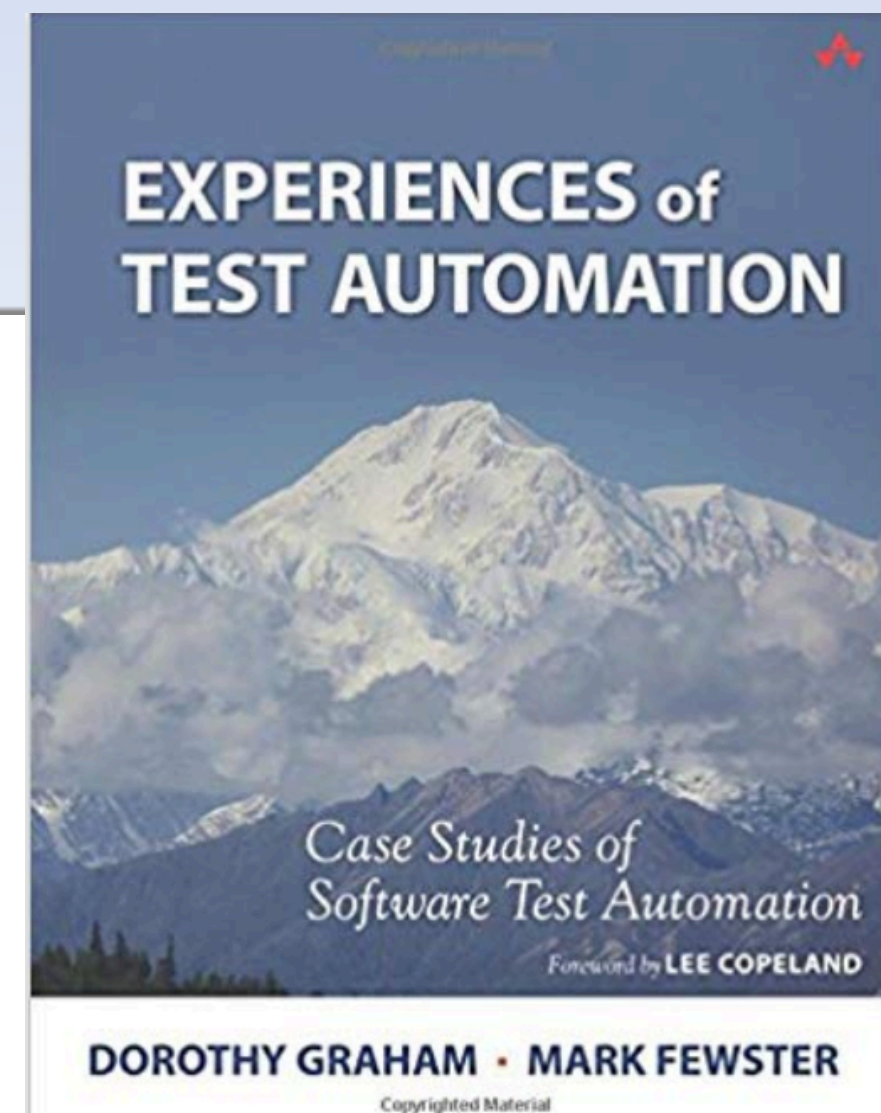
「短期的/中期的」の境目は？

ここでいう「~~質~~質」とは？

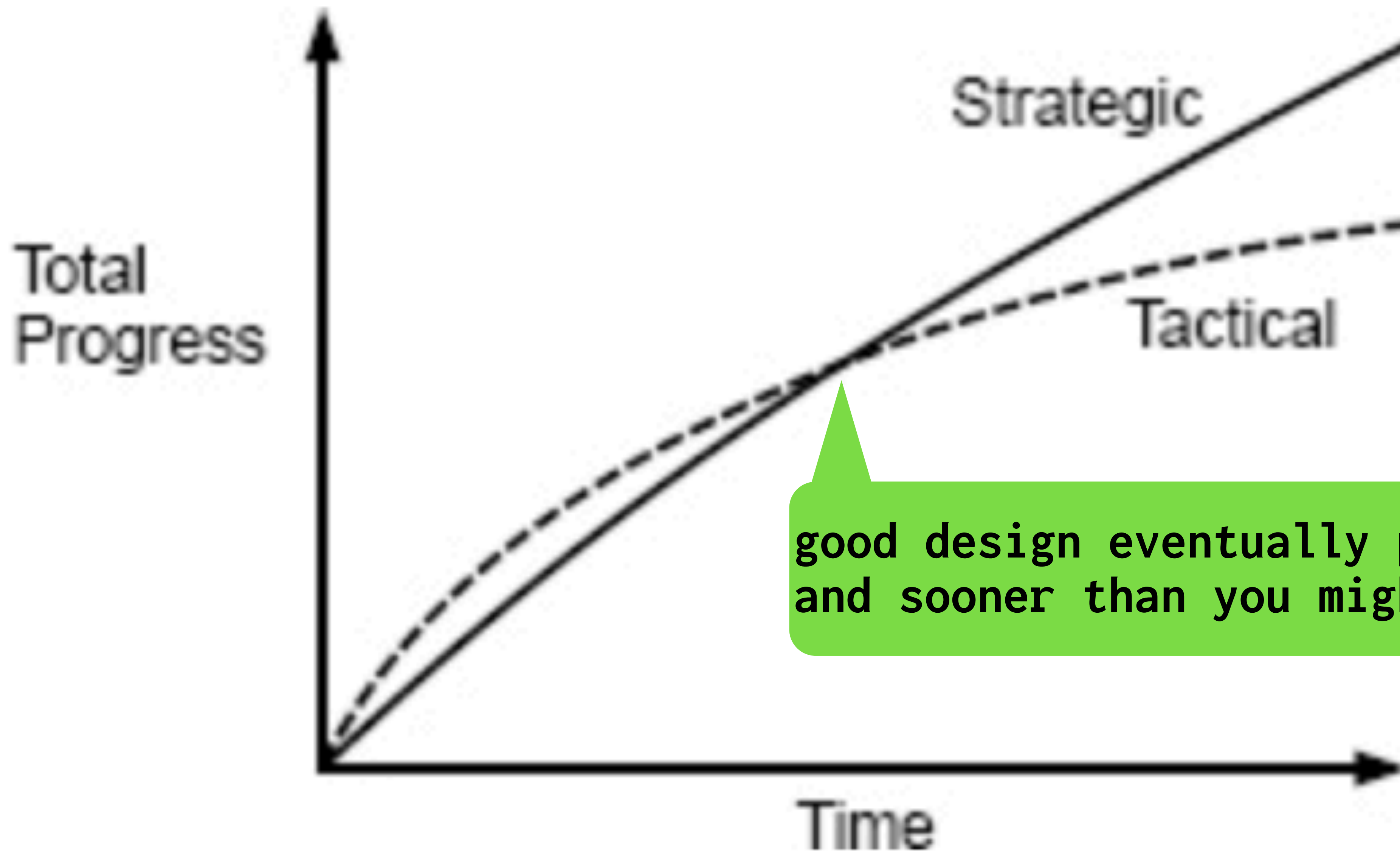
テスト自動化の損益分岐点は「4回」



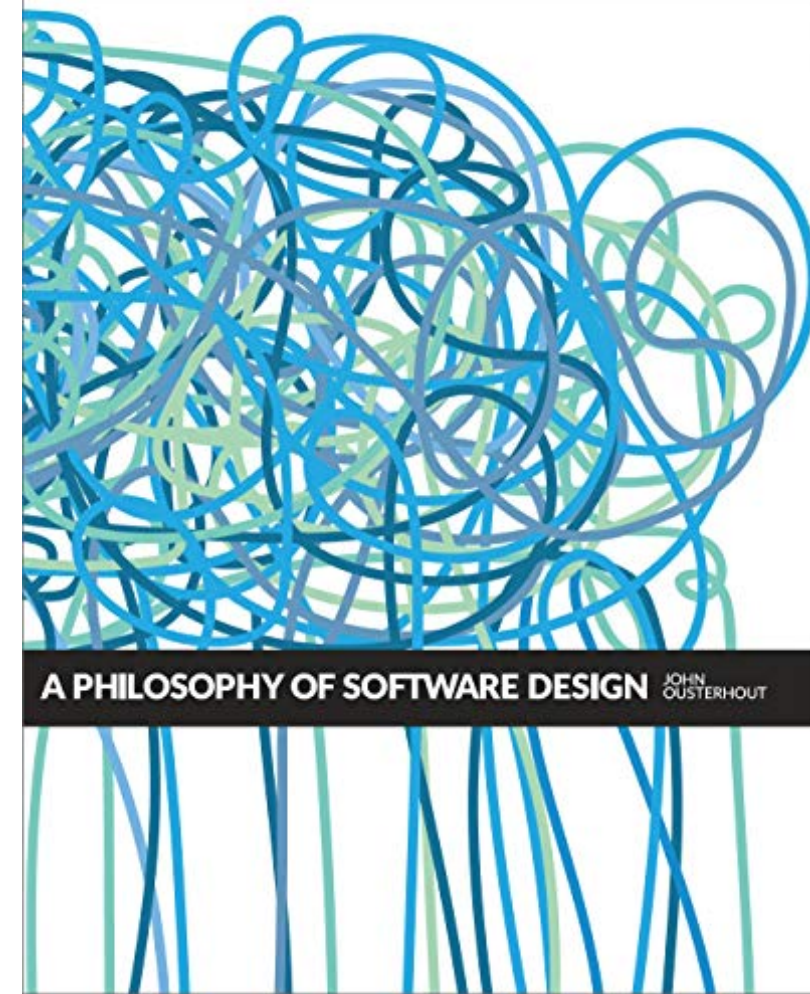
およそ4回で
手動テストと自動化されたテストの
コストが逆転する



A Philosophy of Software Design



good design eventually pays for itself,
and sooner than you might think.

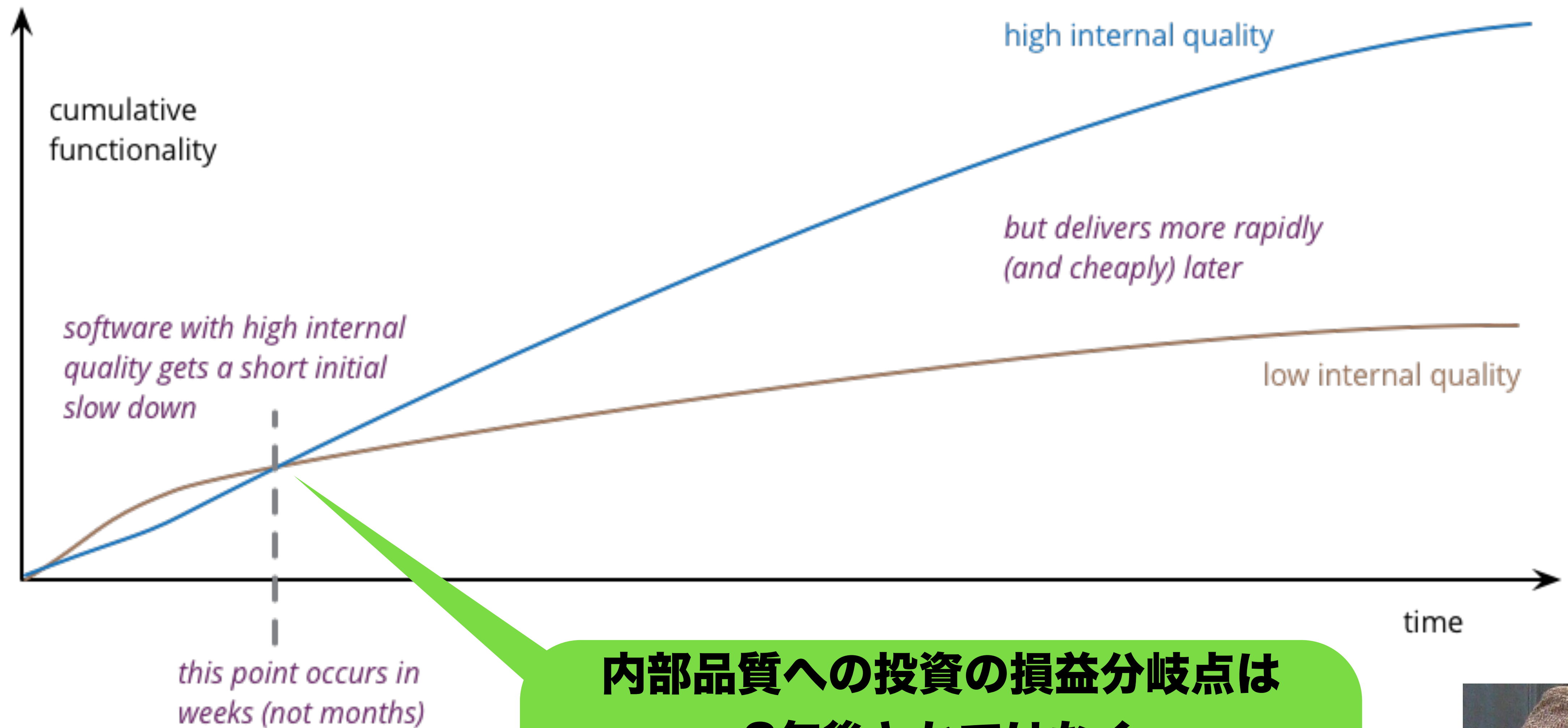


木こりのジレンマ



illustrated by @mtty_mno

内部品質への投資の損益分岐点は1ヶ月以内に現れる



**内部品質への投資の損益分岐点は
3年後とかではなく
1ヶ月以内に現れる**



保守性を犠牲にすればスピードは得られる？

- ・短期的には得られる
- ・1ヶ月後には逆効果になる
- ・長期的には致命傷になる

1ヶ月以内であれば、内部品質への投資の受益者は自分たち自身であり、つまり道徳や矜持の話ではなく損得の話である

キャディ株式会社様の事例

リファクタリング用のリソースを純増した訳ではない。**sprint 内の 20-30% を投資すると決めた**。最初はベロシティが落ちたが、**1-2ヶ月経った後は明確にデリバリー速度が高まった**。（例を挙げると、1ヶ月かかると言われていた規模の開発が5日で終わるようになった）

たまたま参加したイベントで **t_wada** さんの「**質とスピード**」に関する講演を聞き、もともと悩んでいたポイントだったので吹っ切ることができた。



荒ぶる四天王はスコープを削れ



時間



予算



品質



スコープ

与えられた時間に対してやるべきことが多すぎる場合、どうすべきか？

👉 a) スコープを削る

b) もっと人を増やす

c) リリース日を延期する

d) 品質を犠牲にする

『アジャイルサムライ』 p.86



ここから延長戦

ではスピードおよび質は
何とトレードオフなのか

"スピード"を上げるのに有効なのが"教育"を無くすことだと思ってて、おっさんが寄ってたかって開発すれば早くリリース出来るけど、若者が育つ・チャレンジする機会が無くて組織としては死ぬ、というのを体感中

 n593977 のコメント 2019/11/01 13:17

スピードというか分母の時間に注目するとよいのかも。学習とか調査とかにかける時間を減らすことの功罪について

 takahashim のコメント 2019/11/01 12:40

質を上げればスピードも上がるのだとしても、質が技術力によって限定されるのであれば、技術力を底上げするしかないという結論になりそう。

 iwamot のコメント 2019/11/03 20:59

では、品質と速度についてのトレードオフが意識される
とき、実際には何と何が秤にかけられているのか。

それは各個人ではなくプロダクト全体の品質と速度が秤
にかけられているのではないか。言い換えれば、プロダ
クトの品質を支えるために必要なメンバーの成長とその
成長のために必要なフィードバックや学習の時間が秤に
かけられているのではないかと思う。

本物の力をつけるには
時間がかかる

ソフトウェアの開発に最初から最後まで関わるという経験はとても貴重だったんじゃないだろうか。なぜなら、プロジェクト開始時のダメなデザインのしっぺ返しを、後で自分でモロに受けるからだ。

当初ぼくは、ほとんど考えずコードを「アンダーエンジニアリング」していた。後々これを過度に修正してしまい、全てを「オーバーエンジニアリング」し作り込みすぎてしまった。2年半という時間はこういった失敗を経験するには十分な時間で、辞めるころにはシステムのデザインの塩梅が、かなりわかるようになっていたと思う。

Evan Priestley



一番重要で一番やっかいなスキルはシステムを設計するための判断力だ。
限りなくシンプルなデザインというのはなかなか教えられるものではなく、
大方経験を重ねて覚えるものだ。

この「判断力」は、プログラマーにとって非常に重要なのだが、そう簡単に教えられるものでもない。ぼくが知る限り、判断力をつける一番の方法は、自分で設計したシステムを長い間メンテすることだと思う

Evan Priestley

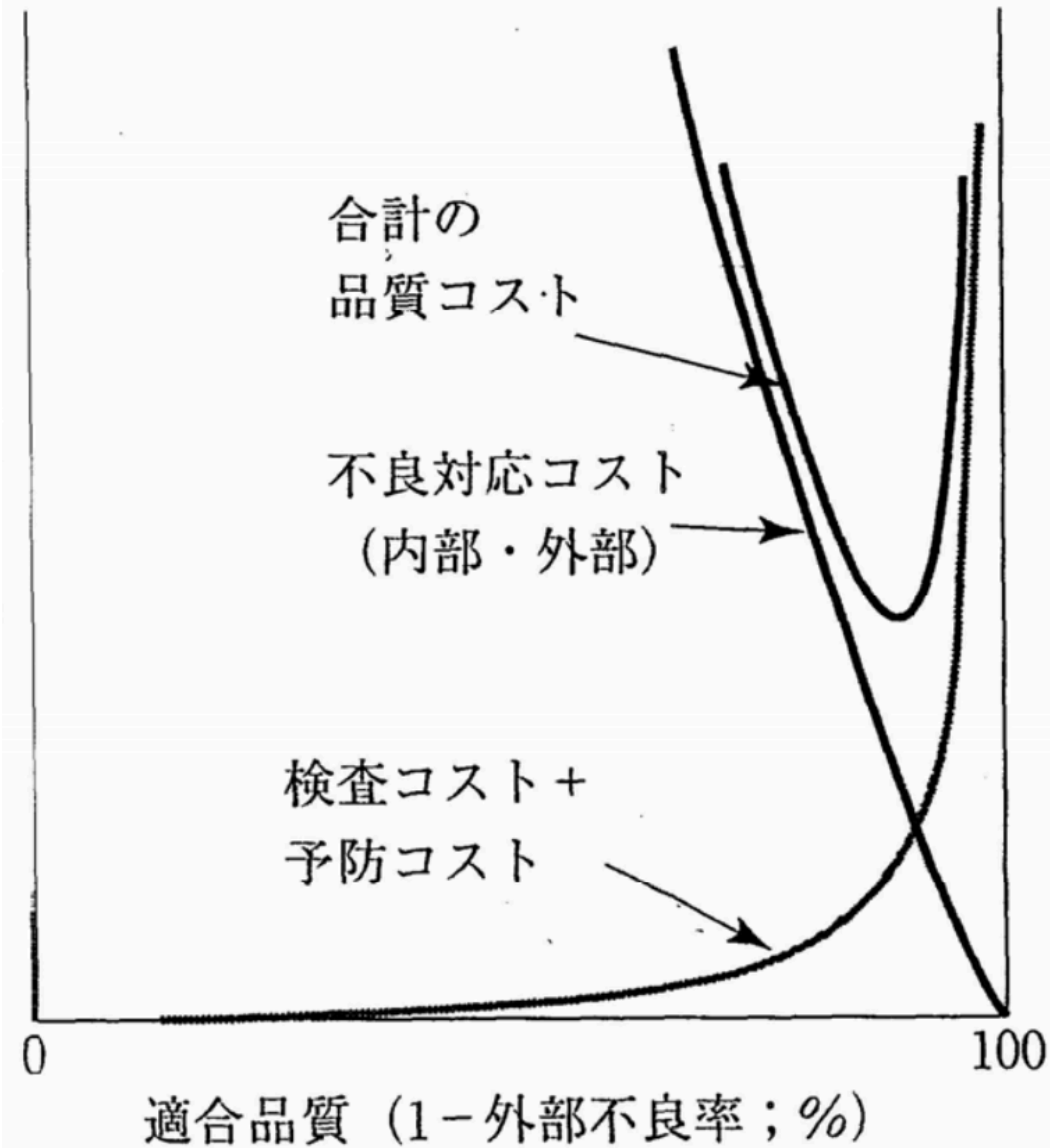


品質とコストは
トレードオフか

Philip Crosby "Quality is Free": 品質アップはコストアップかダウンか

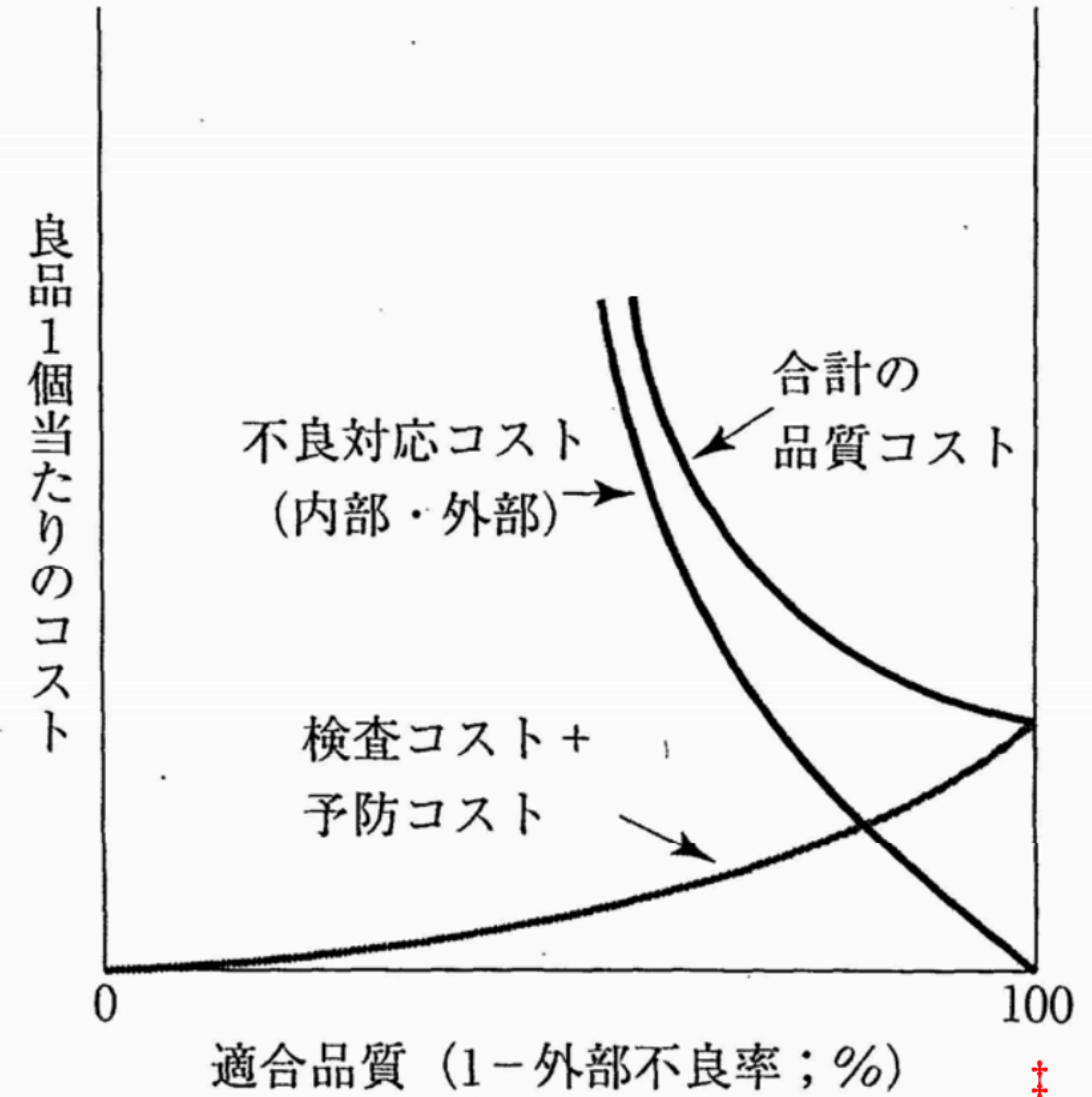
品質のコスト：2つの考え方

a. 伝統的な考え方（最適の不良率を想定）



コストアップ説

b. 新しい考え方 (不良率ゼロを最適と考える)



コストダウン説

品質“実質無料”とは？

1. 品質を向上させることで手戻りを減らすことができます。
 - 開発後半でのバグ、本番障害
2. 手戻りを減らす、つまりやり直しを減らすことができる。
3. 「品質向上のためのコスト」が「品質向上の結果減るやり直しにかかるコスト」を上回ることにより、結果的に品質が“実質無料”となります。

どのくらい
設計に投資するか

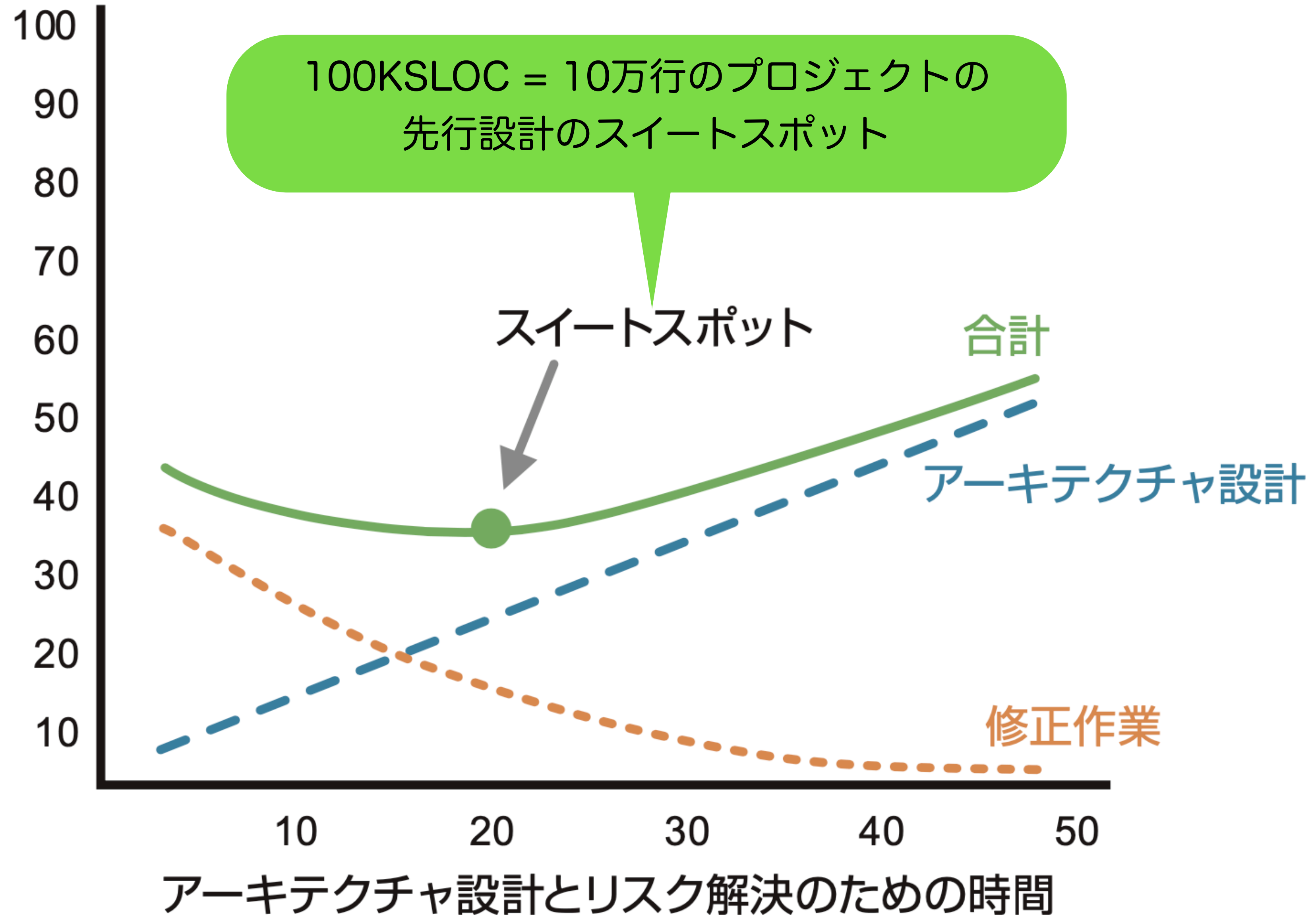
アーキテクティング: いつ、どれだけ？

全体スケジュールに
追加された時間

初期のアーキテク
チャ設計とリスク
解決にかかった時間

修正にかかった
時間

全体としてかかった
時間



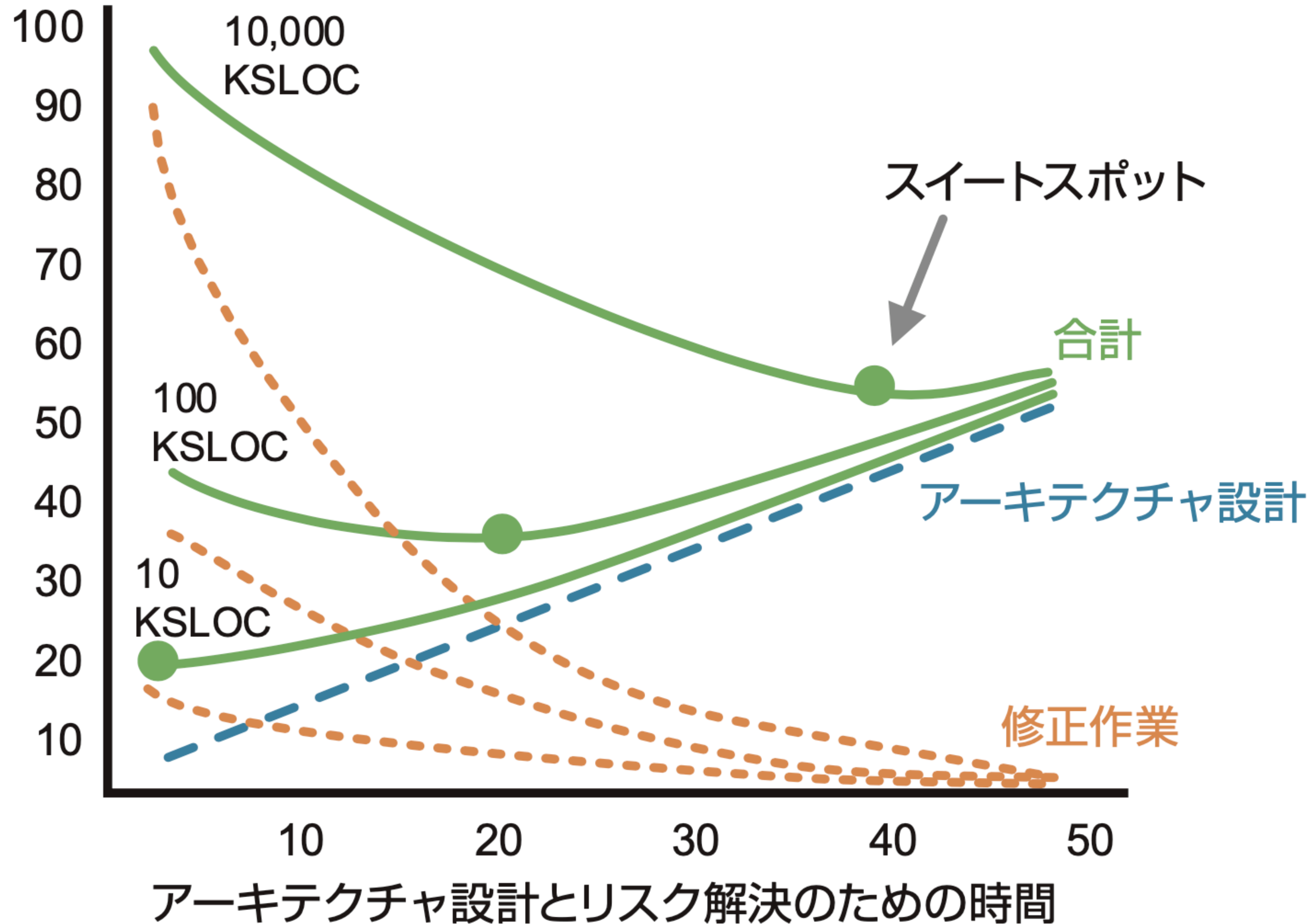
ソフトウェアの規模がスイートスポットに与える影響

全体スケジュールに追加された時間

初期のアーキテクチャ設計とリスク解決にかけた時間

修正にかかった時間

全体としてかかった時間

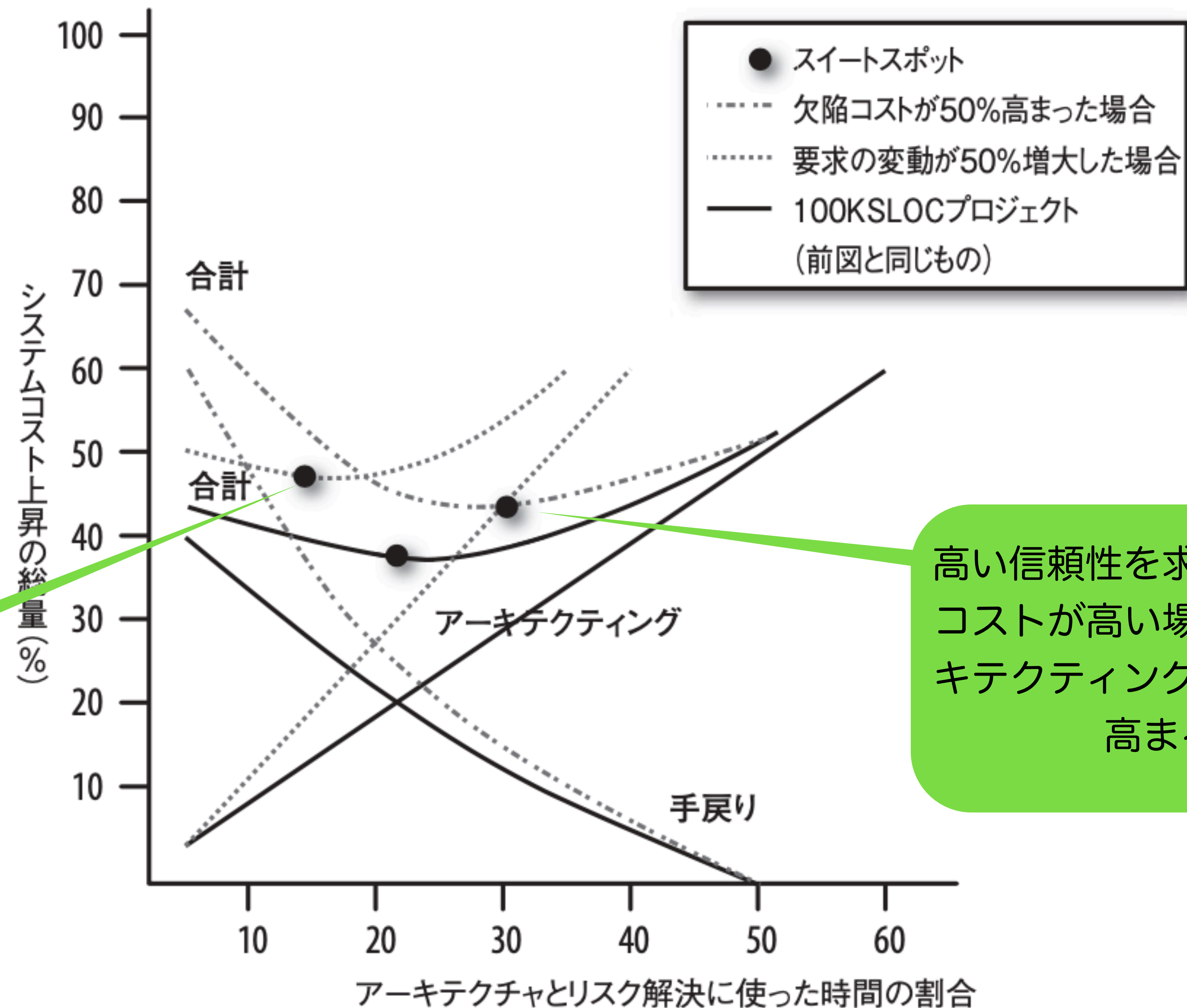


ソフトウェアの規模がスイートスポットに与える影響

- ・ソフトウェアシステムが大きくなればなるほど、前払いのアーキテクチャ設計から受ける恩恵は大きくなる
- ・小規模（1万行）のソフトウェアシステムでは、前払いのアーキテクチャ策定からはほとんど恩恵を受けられない
- ・アーキテクチャ設計に少ししか投資しないと、強烈なしっぺ返しが来ることが予想される
- ・アーキテクチャに投資すればするほど、必要な手直しは少なくなる



要求の変動や欠陥コストがスイートスポットに与える影響



要求の変動が大きい場合は、急速な変化に対応するための分析や文書化の手戻りのため、先行設計の効果は下がる

高い信頼性を求められ欠陥コストが高い場合は、アーキテクティングの必要性が高まる

ご清聴ありがとうございました

- ・ 「品質とスピードはトレードオフの関係にある」は大きな誤解
- ・ 「品質」の名のもとに犠牲にされるのは内部品質の特に保守性（テスト容易性、理解容易性、変更容易性）
- ・ 実際には保守性を高めればスピードは上がるし、保守性を落とせばスピードは下がる
- ・ スピードを落としても保守性は上がらないし、スピードを落とすと仮説検証プロセスが回らない
- ・ 内部品質への投資の損益分岐点は意外と早く(1ヶ月以内)やってくる
- ・ 1ヶ月以内ということは受益者は自分たち自身であり、つまり道德や矜持の話ではなく損得の話である
- ・ スピードおよび質とトレードオフなのは教育、成長、多様性への投資