Clean Craftsmanship とシンプルな設計

Tech Base Okinawa 2023
2023年9月23日

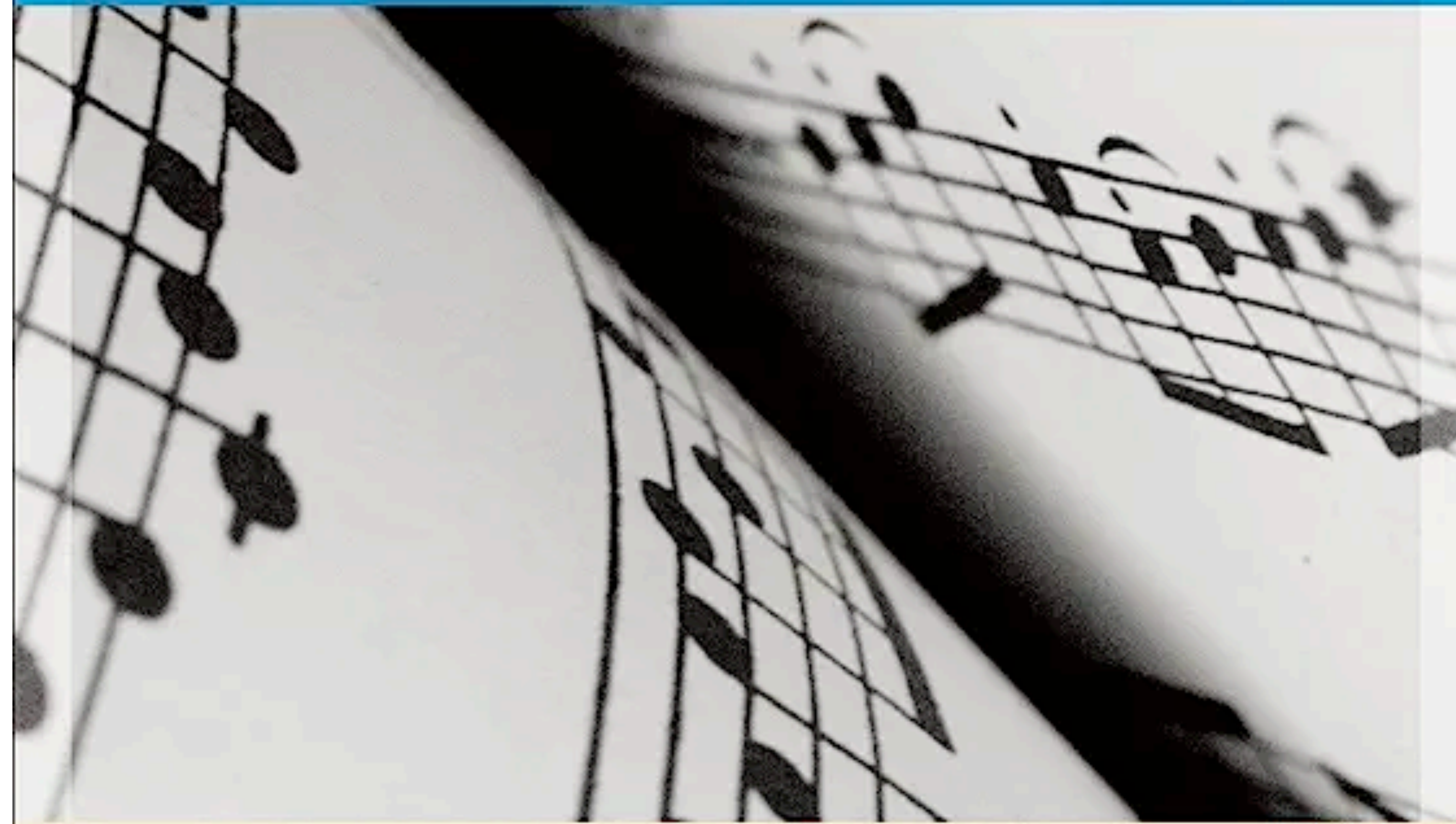
ワイクル株式会社
角征典 （かど まさのり） @kdmsnr
kado.masanori@waicrew.com

Waicrew
ワイクル株式会社

自己紹介

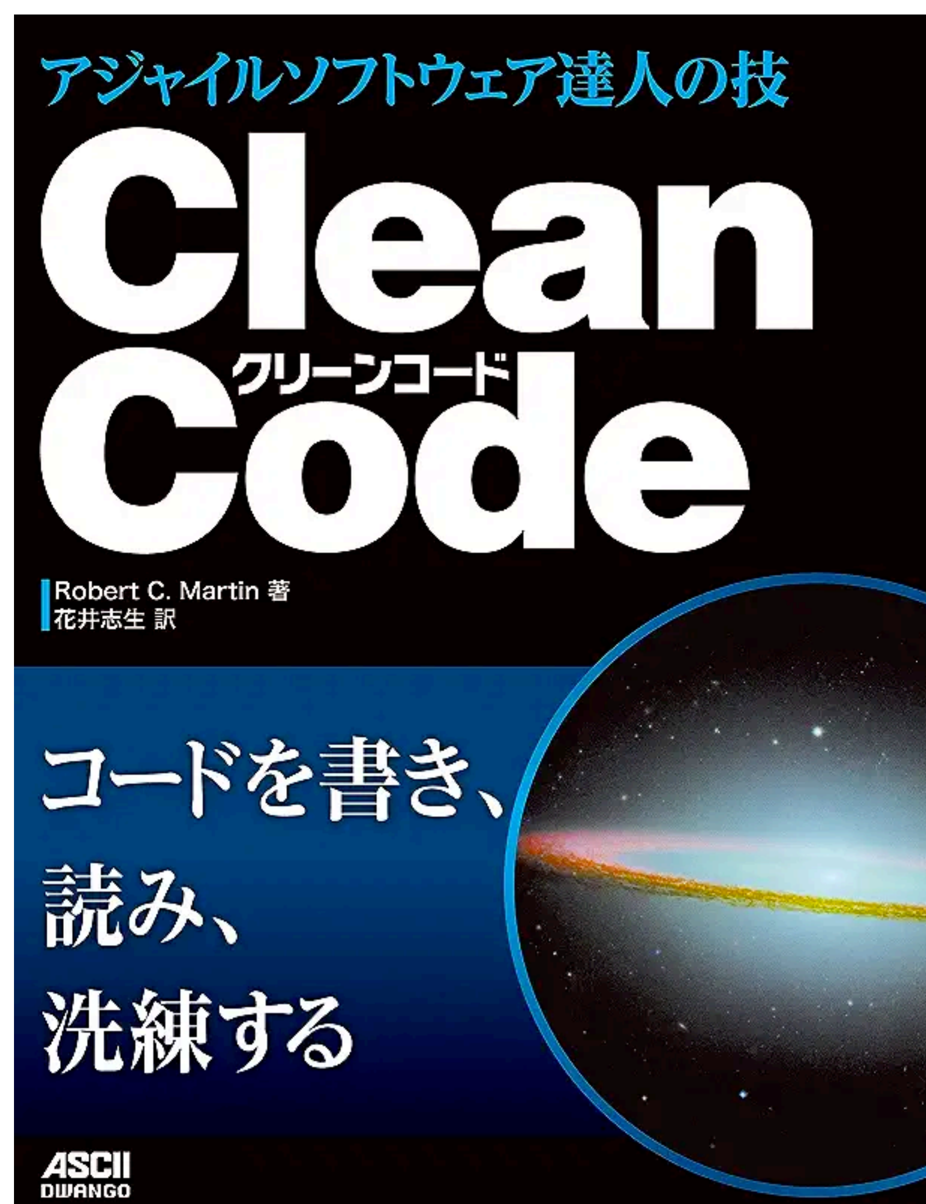
- 角 征典（@kdmsnr）
  - 技術書の翻訳・執筆 →
- ワイクル株式会社 代表取締
  - アジャイル開発／リーンスタートアップの導入支援
- 東京工業大学 環境・社会理工学院 特任講師
  - エンジニアのためのデザイン思考

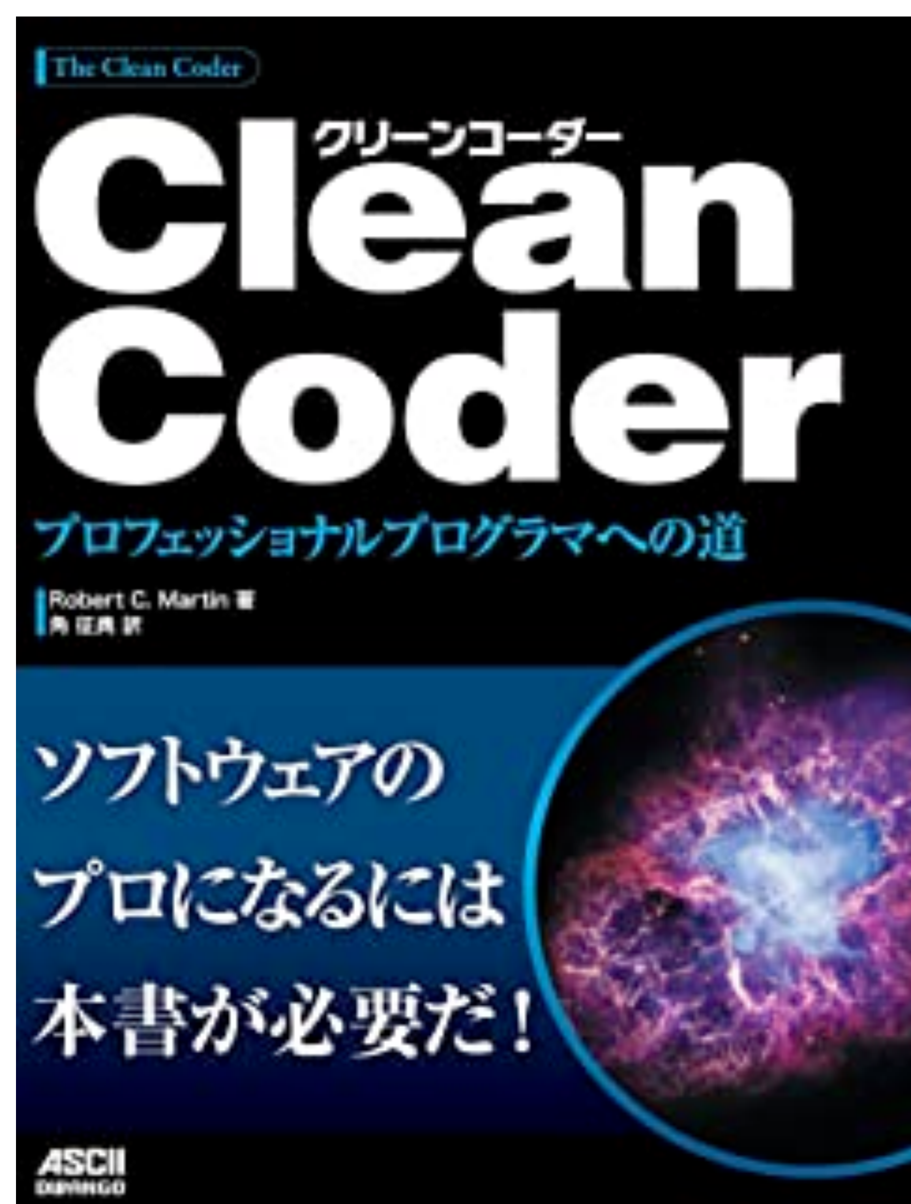
Waicrew ワイクル株式会社


/THEORY/IN/PRACTICE
リーダブルコード
より良いコードを書くための
シンプルで実践的なテクニック
累計発行部数20万部突破!
美しいコードを見ると感動する。優れたコードは
見た瞬間に何をしているかが伝わってくる。
（本書「はじめに」より）
発売から10年以上経った今も売れ続けているロングセラー
全プログラマ必読書!
O'REILLY
オライリー・ジャパン

Clean Architecture
クリーンアーキテクチャ
Clean
Architecture
達人に学ぶソフトウェアの構造と設計
Robert C. Martin 著
角 征典、高木正弘 訳
アーキテクチャの
ルールはどれも
同じである！
ASCII
DWANGO

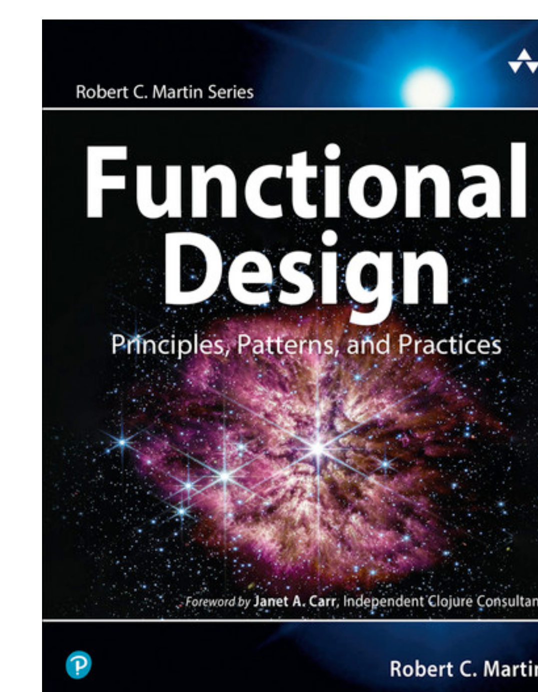
アンクル・ボブ（1952〜）

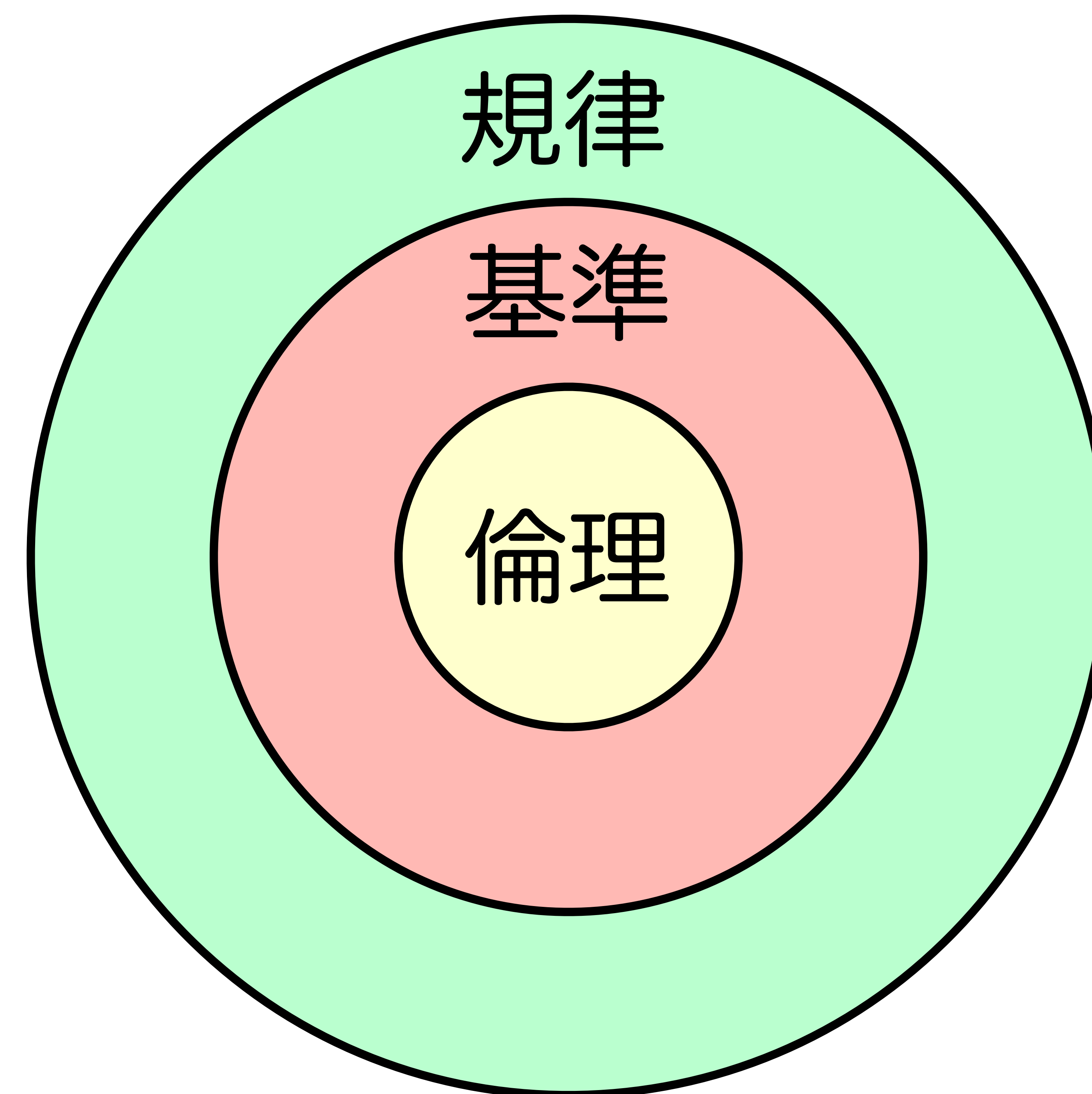




わたしの翻訳担当はここ

新刊が出るらしい





※同心円はClean Architectureのマネ（本には登場しない）

きょうお話すること

1. Clean Craftsmanship
2. 規律
3. シンプルな設計

1. Clean Craftsmanship

クラフトマンシップの定義（1）

- ソフトウェア開発者のコーディングスキルを重視するアプローチ。

  開発者の説明責任よりも金銭的な問題を優先させるなど、ソフトウェア業界の悪弊に対するソフトウェア開発者の反応である。

Source: Wikipedia

クラフトマンシップの定義（2）

- ソフトウェア開発者が自らのキャリアに責任を持ち、常に新しいツールやテクニックを学び、自分を高めていくことを選択するマインドセットである。

Source: Sandro Mancuso『The Software Craftsman』

Waicrew ワイクル株式会社

クラフトマンシップの定義（3）

- クラフトマンシップとは、何かをうまくやる方法を知っている状態である。それは、優れた指導と豊富な経験の結果である。
- 「クラフトマン」とは、特定の分野に関する高度なスキルを持ち、物事を成し遂げる人である。道具や業界に精通しており、仕事に誇りを持ち、仕事に対する尊厳とプロ意識を持って行動できると信頼されている人である。

Source: Bob C. Martin『Clean Craftsmanship』

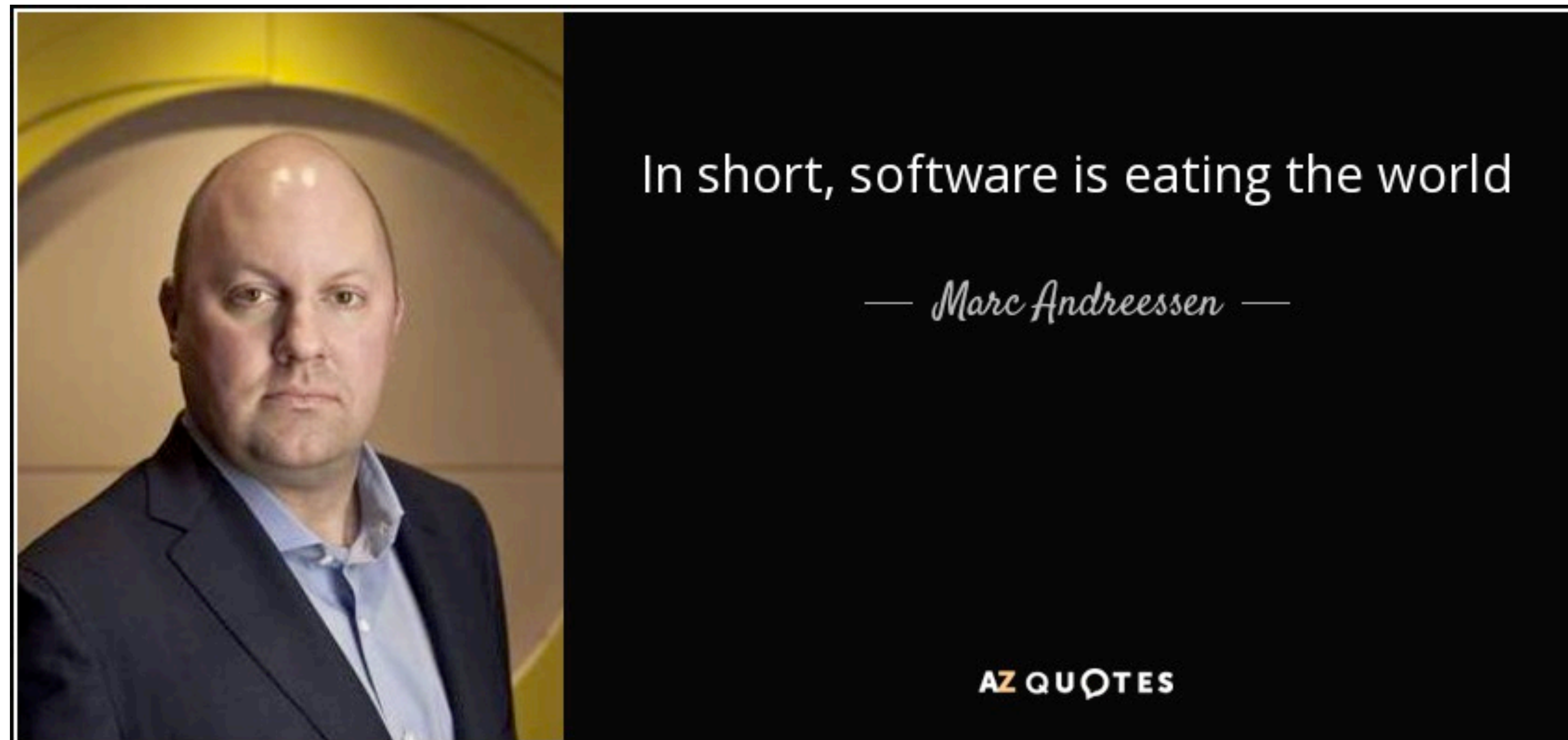
👉「個人」としてのプロ意識

ソフトウェアが世界を支配する時代（2011）

社会はまだそのことを理解していない。

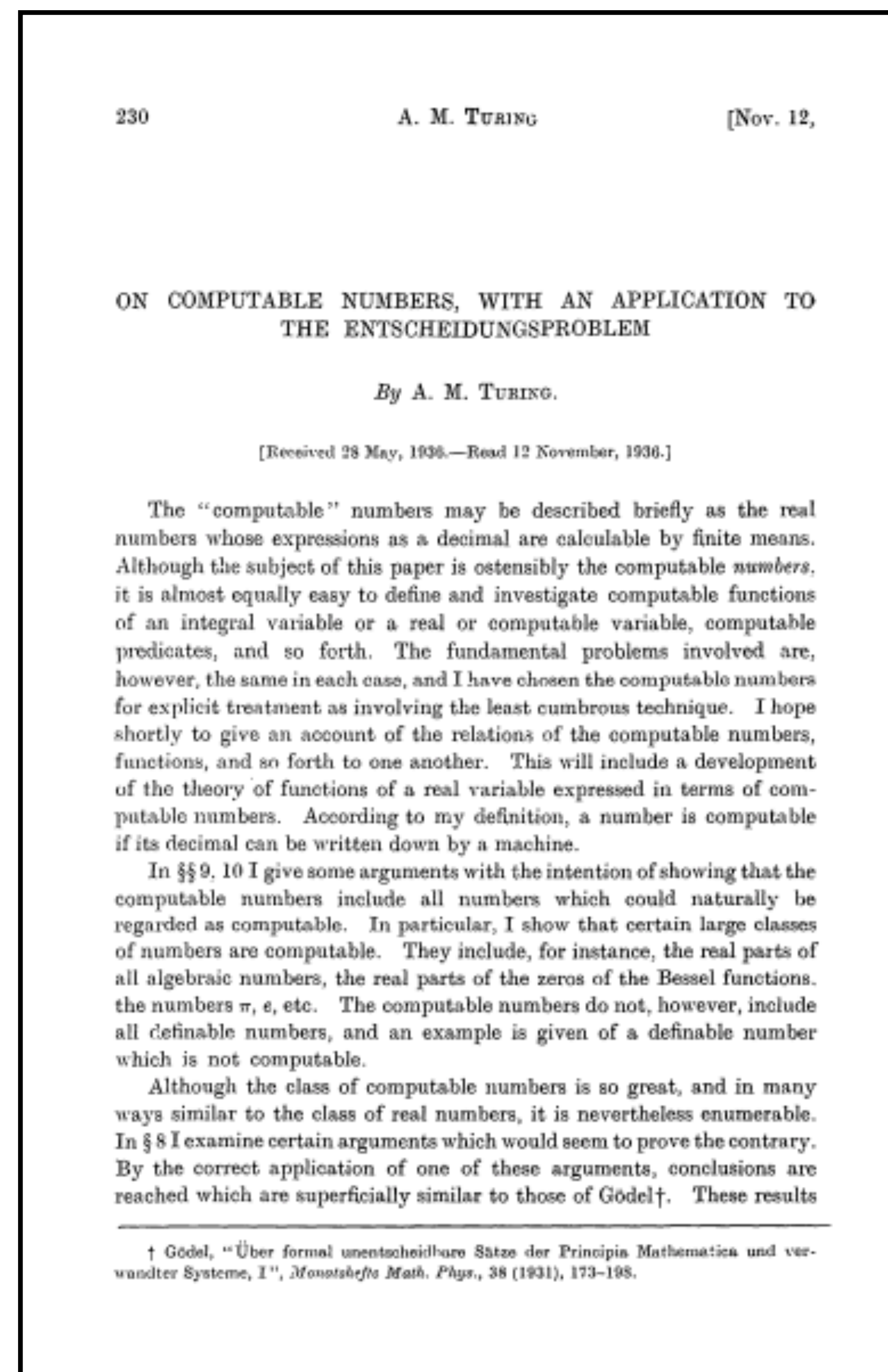
我々プログラマーもそのことを理解していない。

Bob C. Martin『Clean Craftsmanship』



「ソフトウェア」の歴史は浅い

チューリングマシン（1936）、ACE（1945）



230 A. M. TURING [Nov. 12,

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

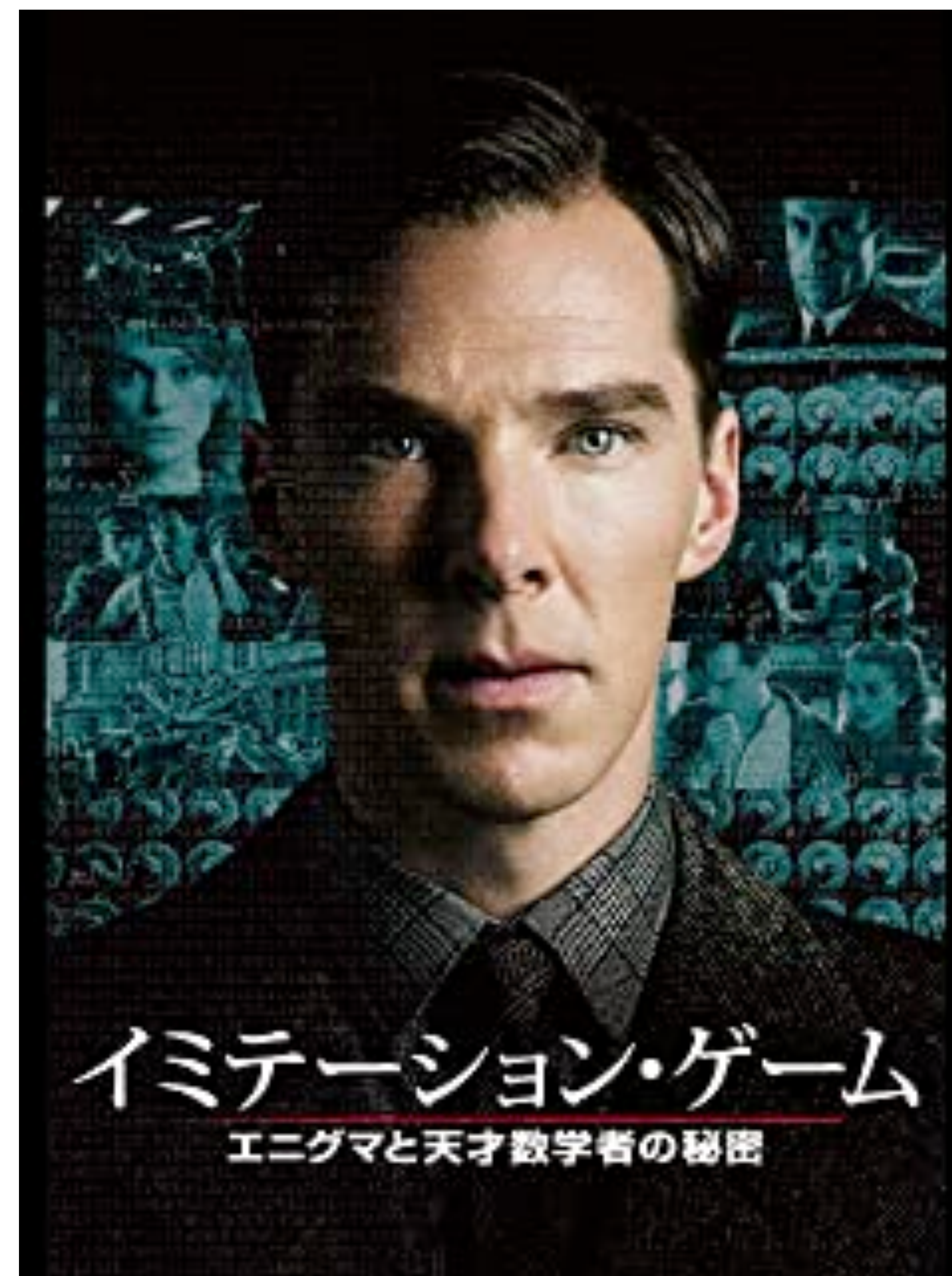
The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

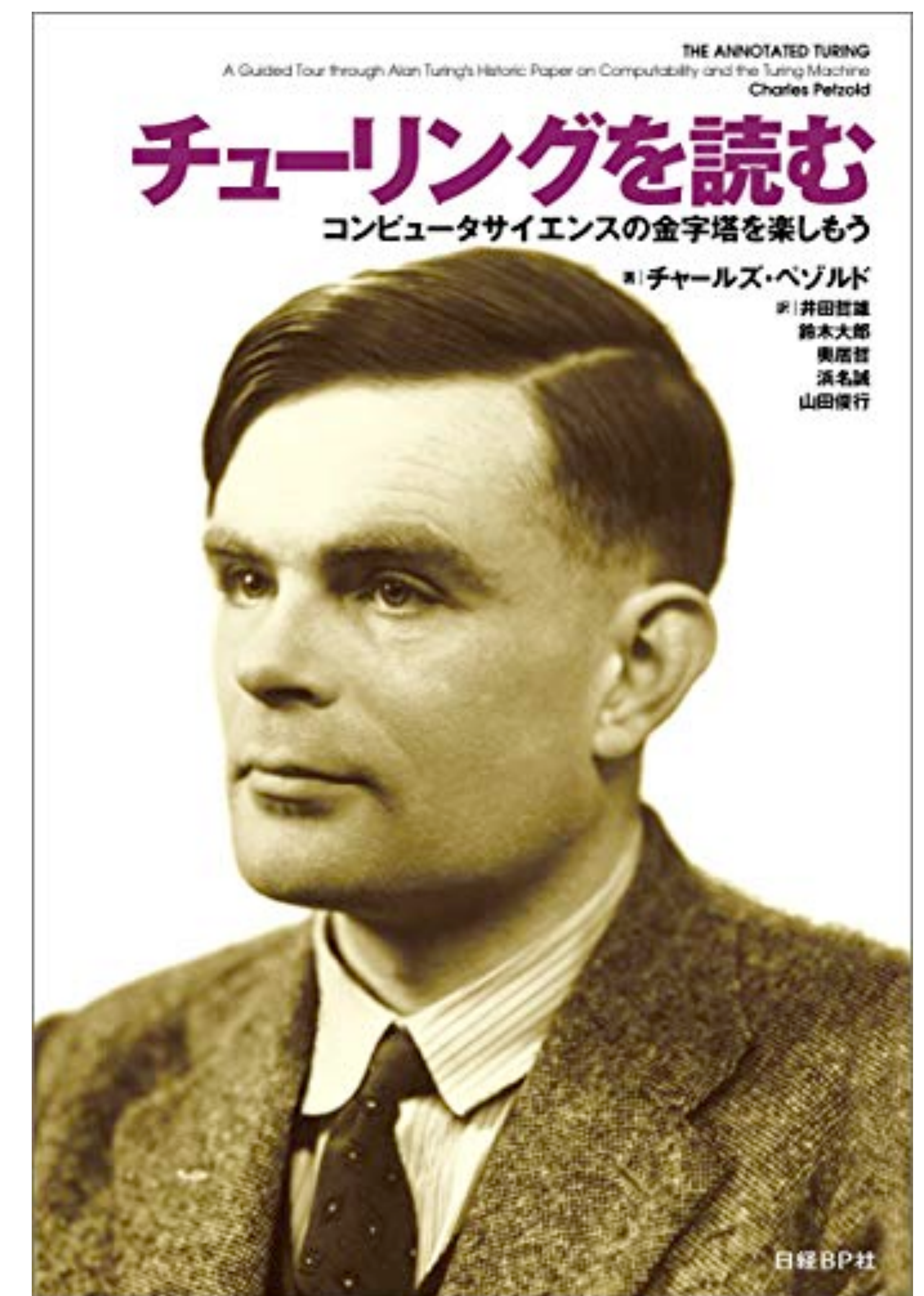
In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers π, e, etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In § 8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel†. These results

† Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", *Monatshefte Math. Phys.*, 38 (1931), 173–198.

「計算可能数とその決定問題への応用」(1936)





正確には1958年のテューキーの論文が初

これからは（問題を）計算可能な形にする
能力のある数学者がかなりの人数必要になるはずだ。
困難となるのは、我々が何をしているのかを
見失わないために、適切な**規律**を維持することである。

―アラン・チューリング

A.M. Turing's ACE Report of 1946 and Other Papers - Vol. 10,
"In the Charles Babbage Institute Reprint Series for the History of Computing", (B.E. Carpenter, B.W. Doran, eds.).
The MIT Press, 1986.

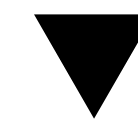
能力や規律を持たないプログラマーの増加

プログラマーの人数
数億人（10^8）▶

チューリングから75年▲

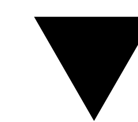
Clean Coder (Clean Coders Video Series) by Robert C. Martin

$$log_2 10^8 \approx 27$$
$$75 \div 27 \approx 2.8$$



約3年で

プログラマーが倍!?

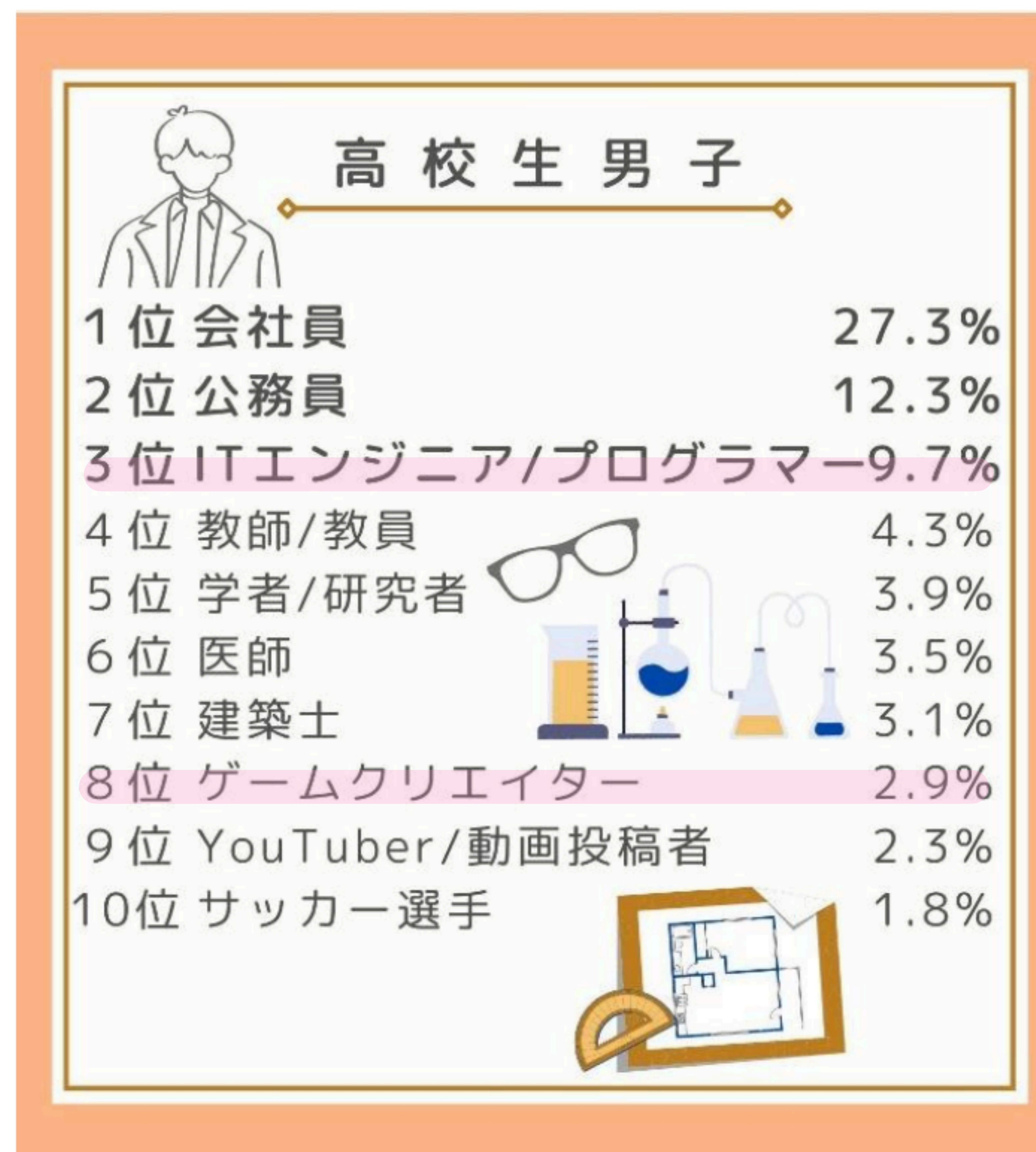


いつも半数が

#駆け出しエンジニア

Waicrew
ワイクル株式会社

高校生の「大人になったらなりたいもの」（2023）



出典：第一生命／第34回「大人になったらなりたいもの」調査結果

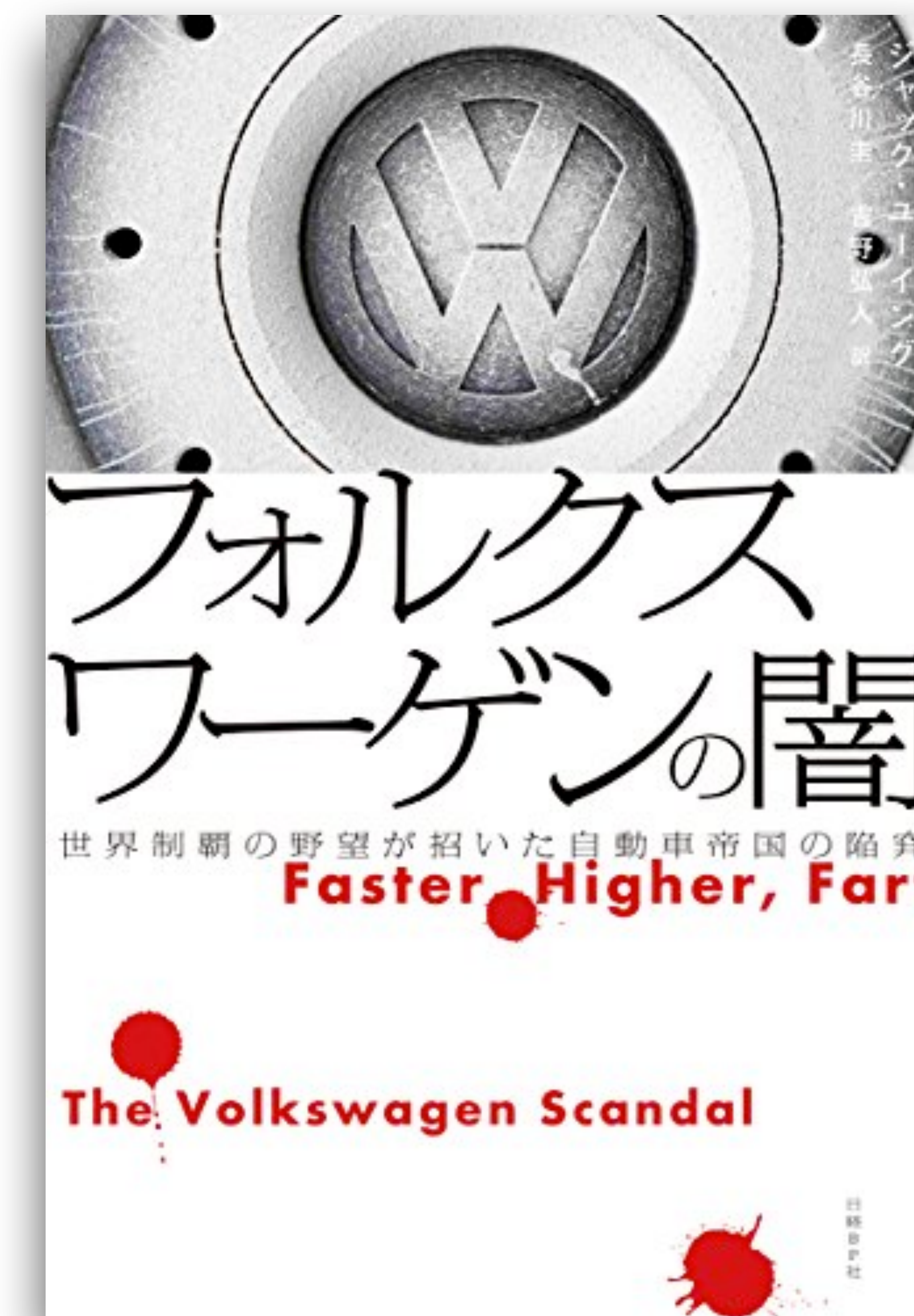
Waicrew
ワイクル株式会社

👉「個人」のプロ意識や規律を伝えていく必要がある

規律を持たなかったことによる問題

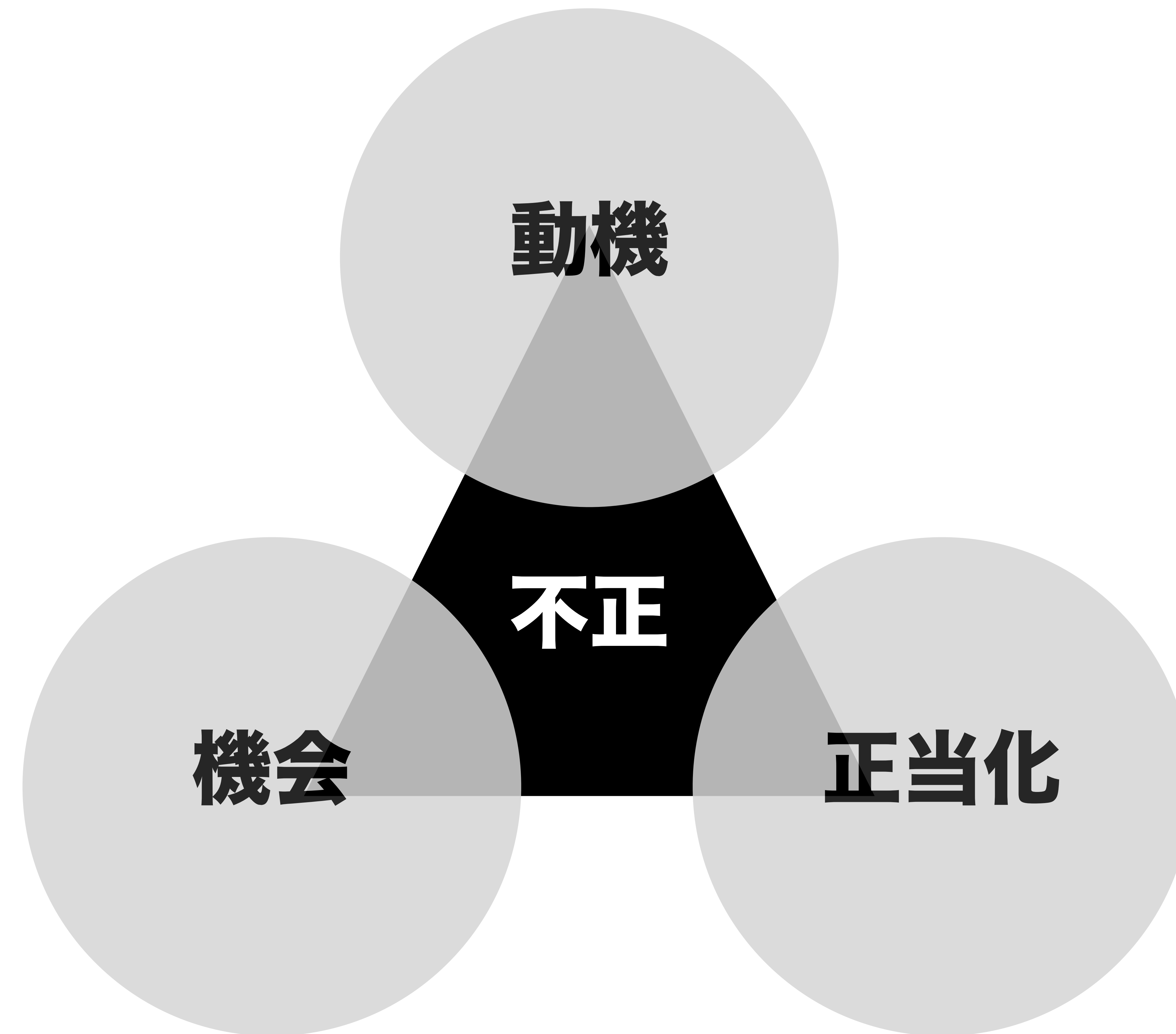
- VW社が競合他社に比べて簡素な排ガス対策装置で対応できていたことに、かねて疑問を抱いていたからだ。疑問の答えは、優れた技術にあったわけではなく、「いかさま」だった。排ガス対策装置の作動を弱める「デフィート・ソフト」を使い、公式試験のときにだけ、「ジェッタ」などに搭載する排気量2.0Lのディーゼルエンジンの排ガス成分が規制値内に収まるようにした。通常の走行時は、規制値を大幅に超える有害な排ガス物質を垂れ流す。

https://xtech.nikkei.com/dm/atcl/news/16/080808697/



クレッシーの「不正のトライアングル」

神戸製鋼品質不正問題(*)の分析例



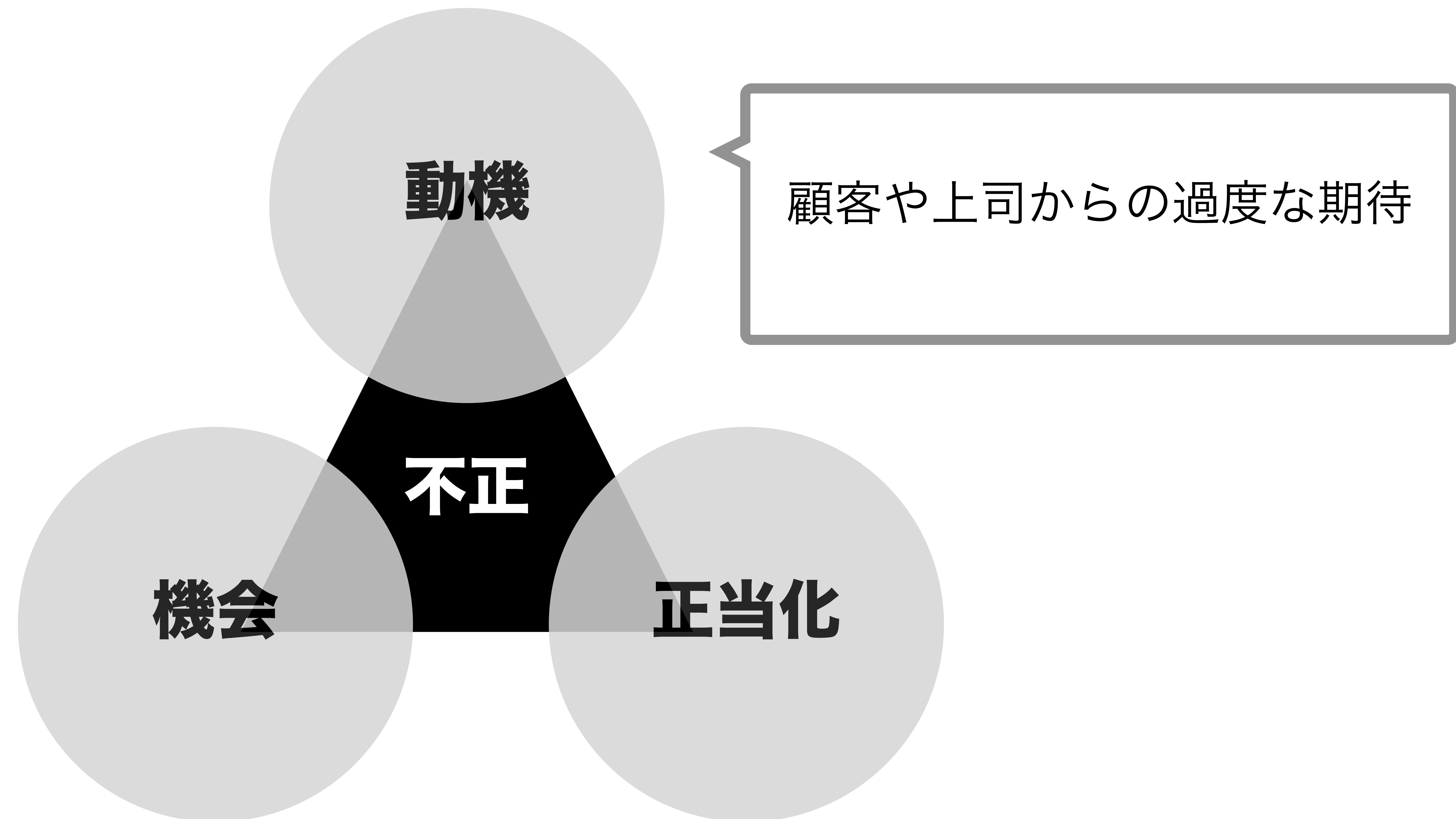
飯野 大介, 小松原 明哲, 2B1-2　不正のトライアングルによる組織的不正の評価について,
人間工学, 2019, 55 巻, Supplement 号, p. 2B1-2, 公開日 2019/07/31, Online ISSN 1884-2844, Print ISSN 0549-4974,

Waicrew ワイクル株式会社

*: https://www.kobelco.co.jp/releases/1199080_15541.html

クレッシーの「不正のトライアングル」

神戸製鋼品質不正問題(*)の分析例

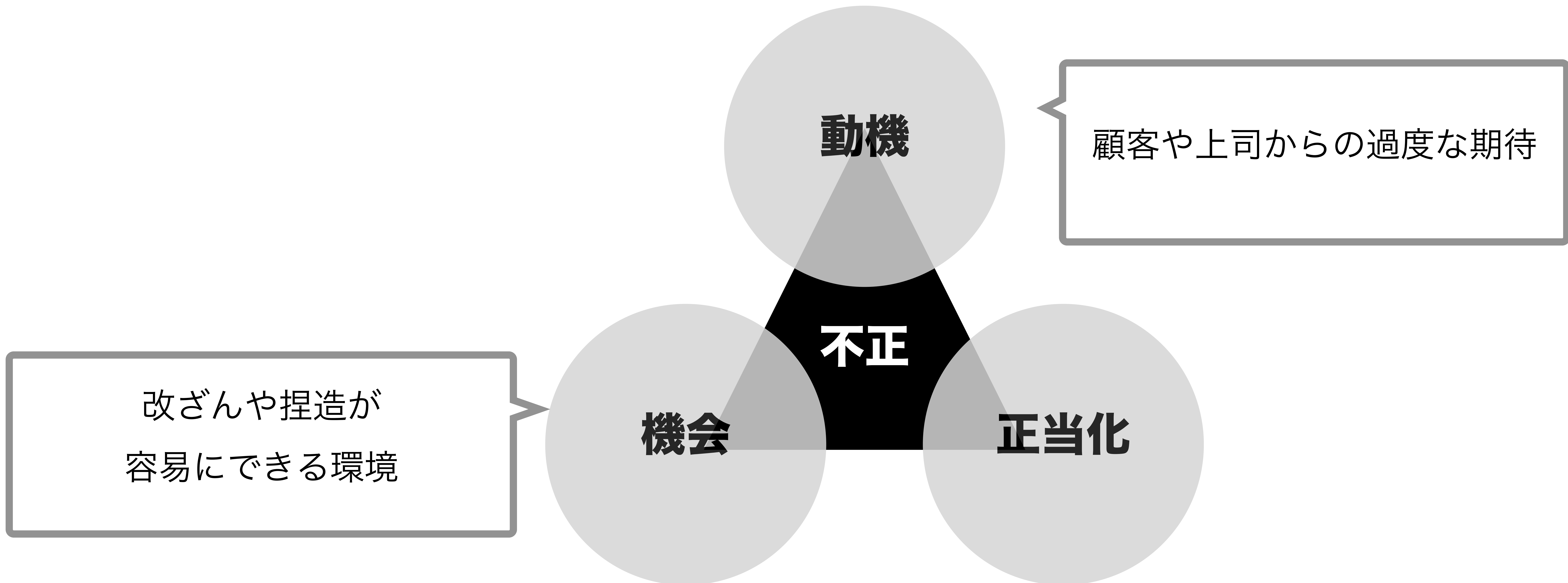


飯野 大介, 小松原 明哲, 2B1-2　不正のトライアングルによる組織的不正の評価について,
人間工学, 2019, 55 巻, Supplement 号, p. 2B1-2, 公開日 2019/07/31, Online ISSN 1884-2844, Print ISSN 0549-4974,

*: https://www.kobelco.co.jp/releases/1199080_15541.html

クレッシーの「不正のトライアングル」

神戸製鋼品質不正問題(*)の分析例



飯野 大介, 小松原 明哲, 2B1-2　不正のトライアングルによる組織的不正の評価について,
人間工学, 2019, 55 巻, Supplement 号, p. 2B1-2, 公開日 2019/07/31, Online ISSN 1884-2844, Print ISSN 0549-4974,

*: https://www.kobelco.co.jp/releases/1199080_15541.html

クレッシーの「不正のトライアングル」

神戸製鋼品質不正問題(*)の分析例

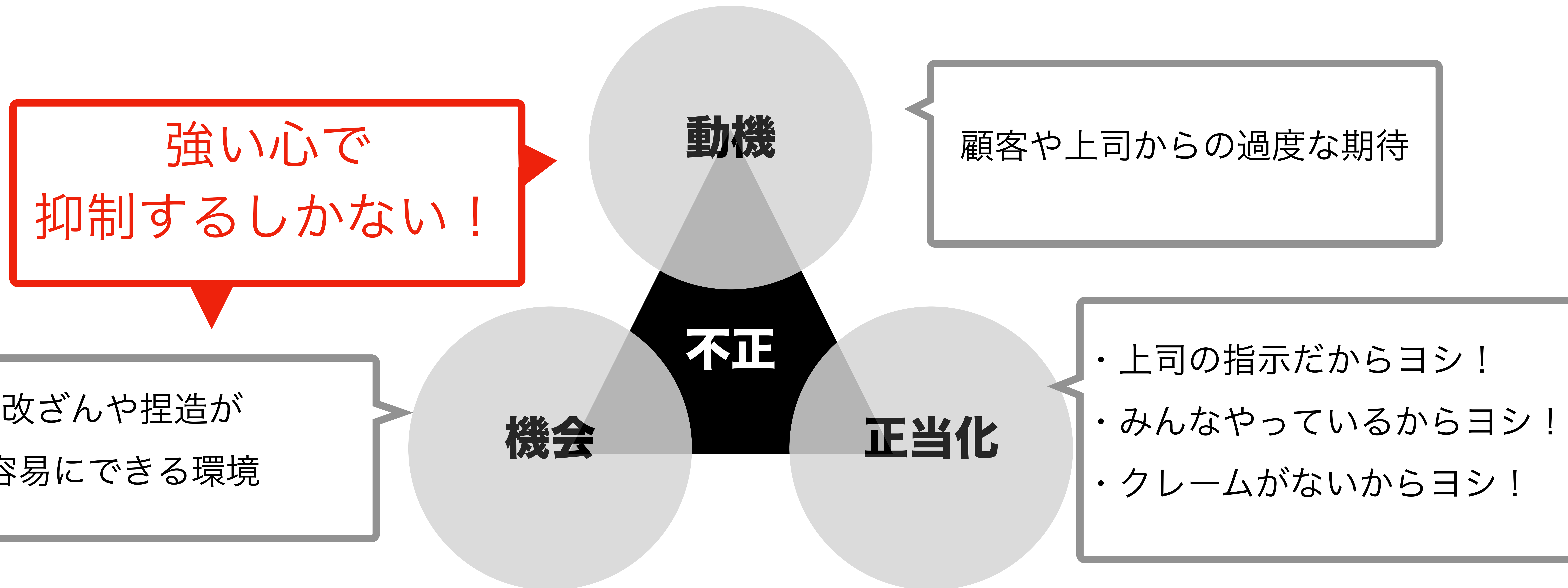
飯野 大介, 小松原 明哲, 2B1-2　不正のトライアングルによる組織的不正の評価について,
人間工学, 2019, 55 巻, Supplement 号, p. 2B1-2, 公開日 2019/07/31, Online ISSN 1884-2844, Print ISSN 0549-4974,

Waicrew ワイクル株式会社

*: https://www.kobelco.co.jp/releases/1199080_15541.html

クレッシーの「不正のトライアングル」

神戸製鋼品質不正問題(*)の分析例



飯野 大介, 小松原 明哲, 2B1-2　不正のトライアングルによる組織的不正の評価について,
人間工学, 2019, 55 巻, Supplement 号, p. 2B1-2, 公開日 2019/07/31, Online ISSN 1884-2844, Print ISSN 0549-4974,

*: https://www.kobelco.co.jp/releases/1199080_15541.html

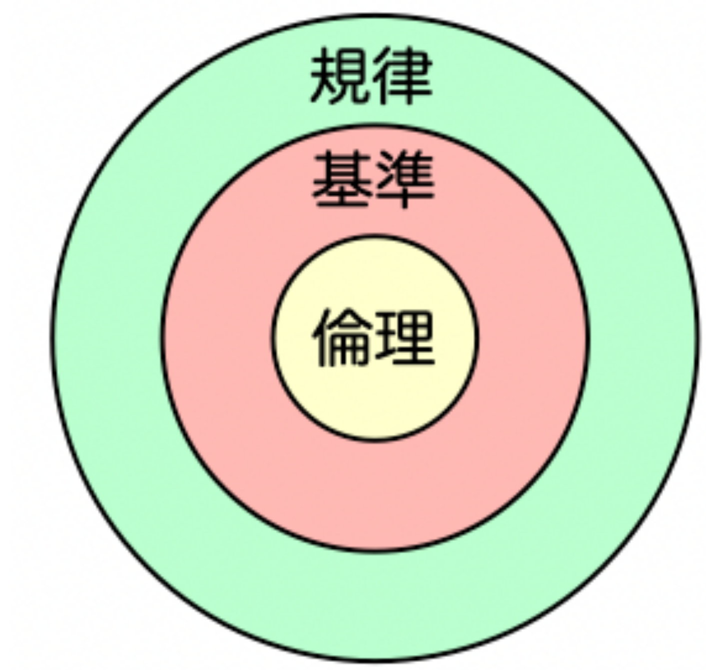
倫理を持つエンジニアの証「鉄の指輪」

“鉄の指輪は、カナダで教育を受けた多くのエンジニアが、
その職業に伴う義務や倫理を象徴し思い出させるものとして
身につけている指輪である”

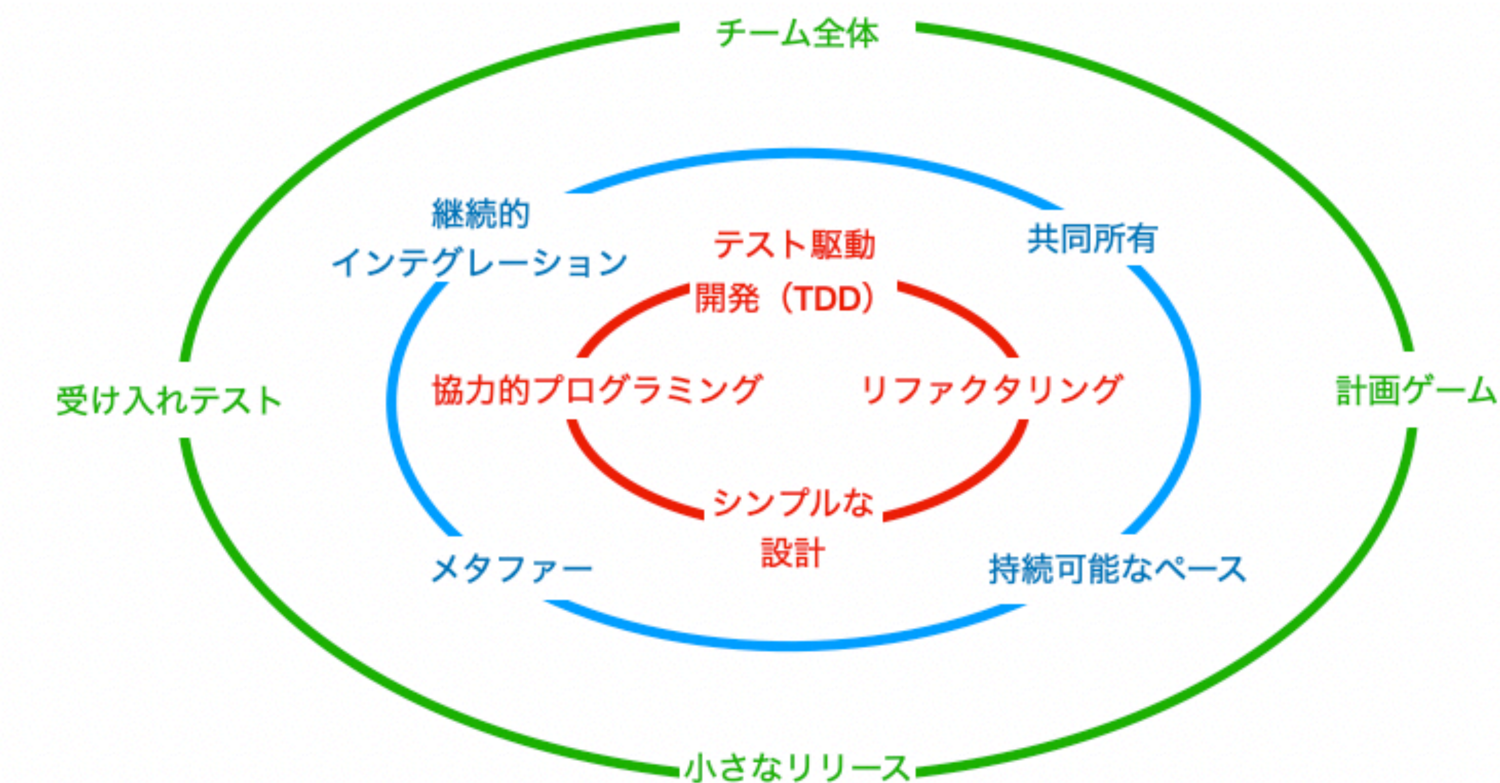
https://en.wikipedia.org/wiki/Iron_Ring

プログラマーが自分で正しさを決めるだけでは不十分である。これからは、規律、基準、倫理が必要になるはずだ。我々は、それらをプログラマーである自分たちで定義するべきなのか、それとも知らない誰かに強制されるべきなのかを決めなければならない。

規律、基準、倫理



- 規律
  - テスト駆動開発、リファクタリング、シンプルな設計、協力的プログラミング（ペア + モブ）、受け入れテスト

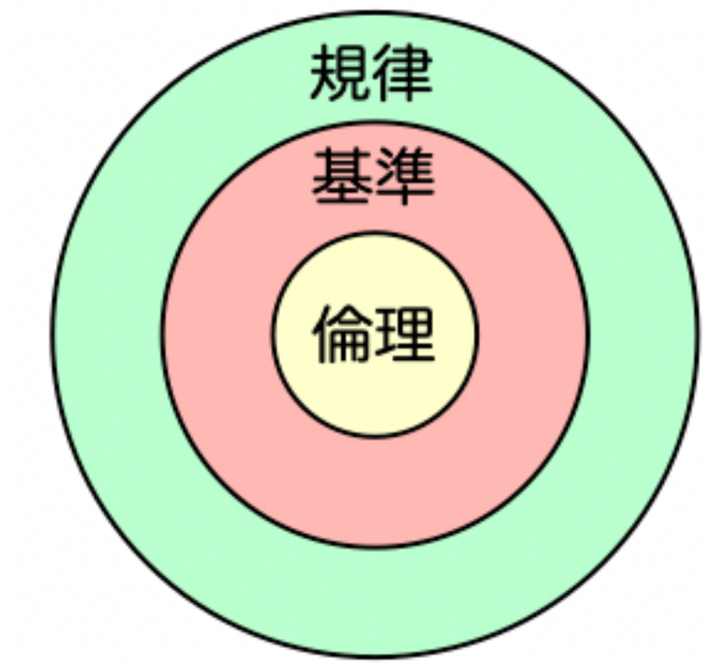


- 基準
  - 生産性、品質、勇気
- 倫理
  - プログラマーの誓い（十戒）
    - 3つのテーマ（有害、誠実、チームワーク）

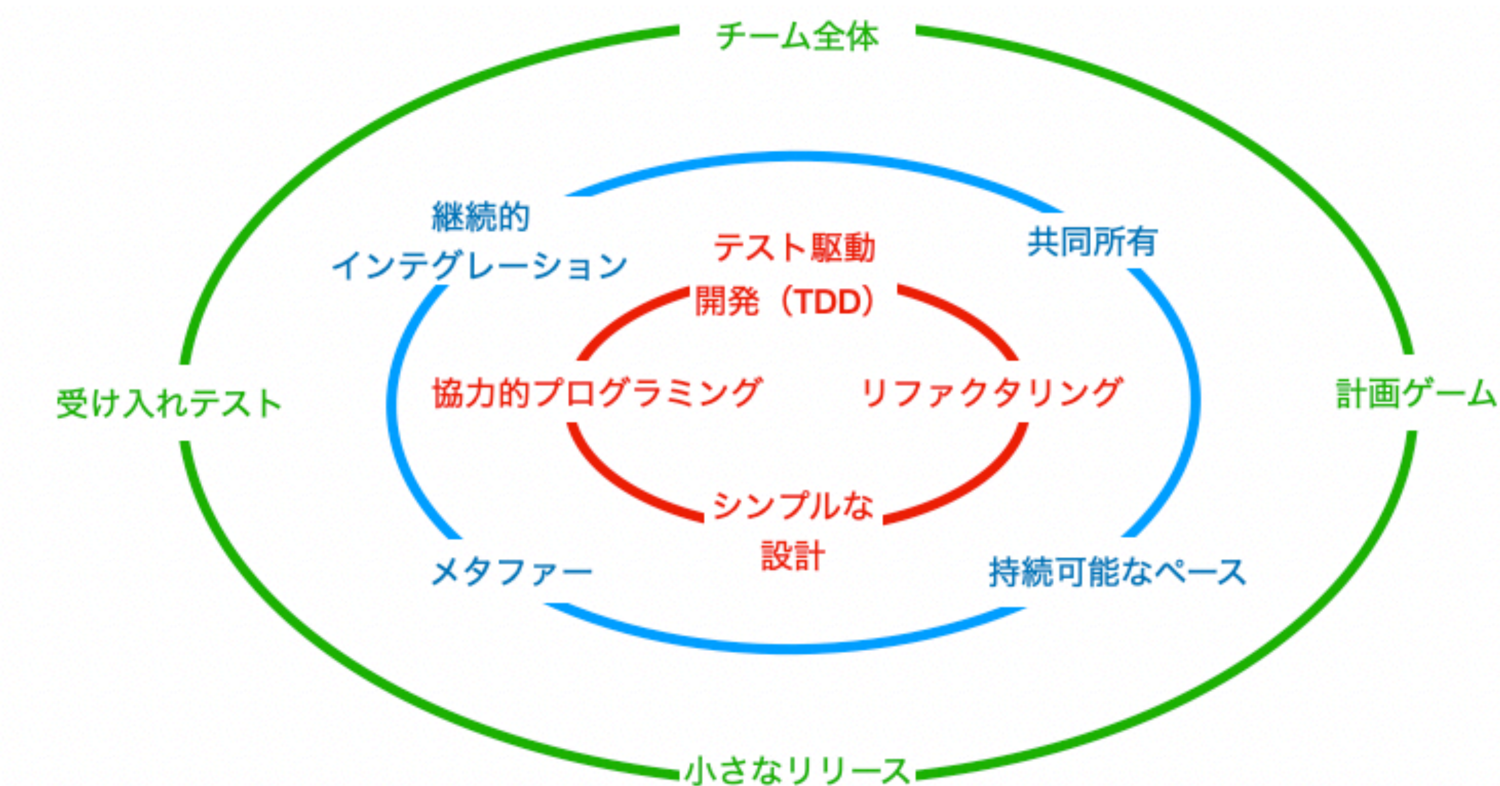
👉「業界」としてのプロ意識

2. 規律

規律、基準、倫理

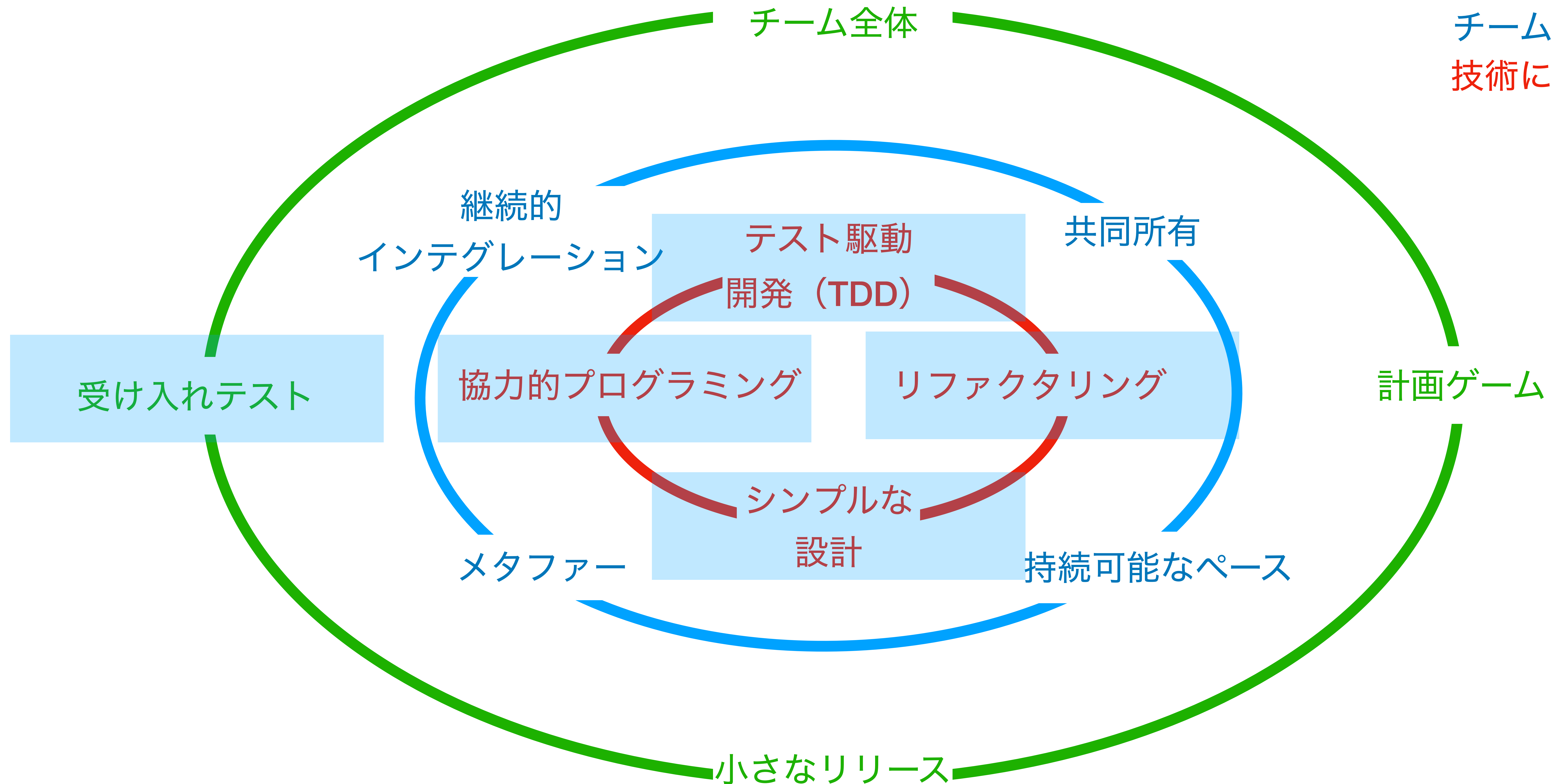


- 規律
  - テスト駆動開発、リファクタリング、シンプルな設計、協力的プログラミング（ペア + モブ）、受け入れテスト



- 基準
  - 生産性、品質、勇気
- 倫理
  - プログラマーの誓い（十戒）
    - 3つのテーマ（有害、誠実、チームワーク）

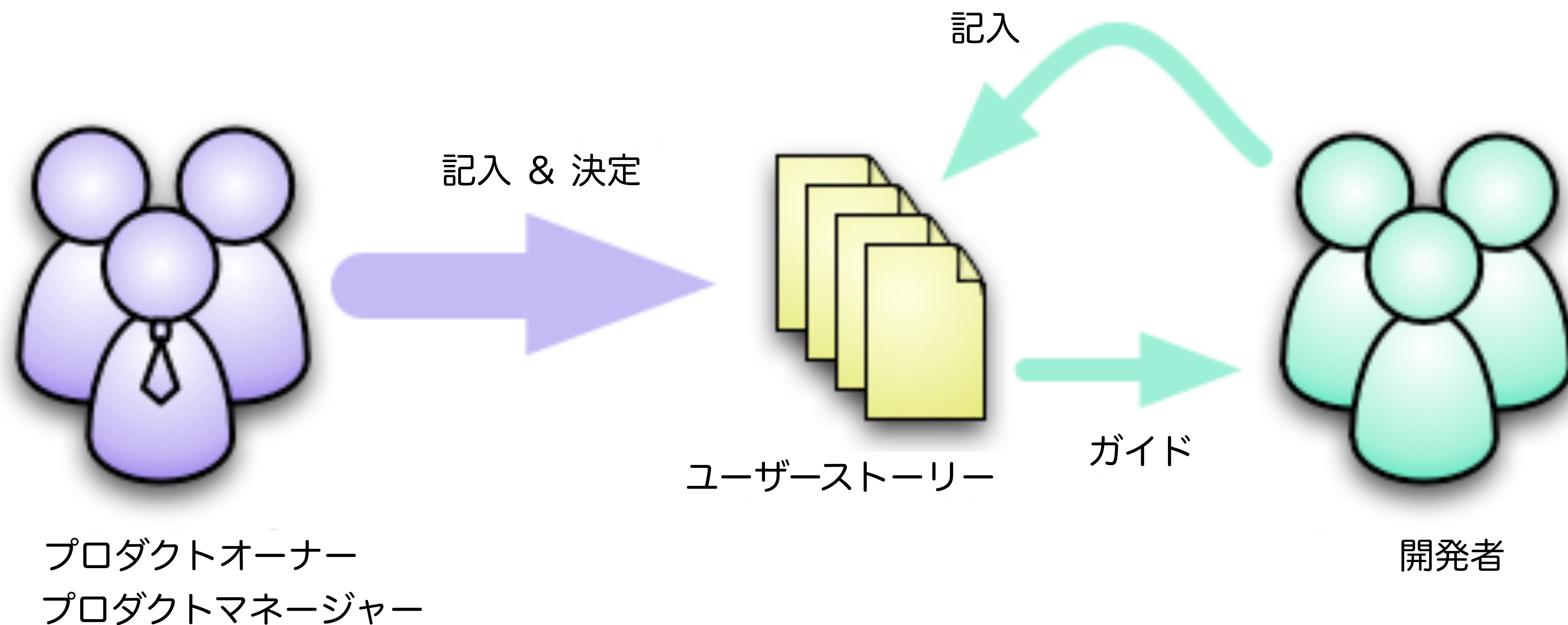
プログラマーの「規律」



Bob Martin『Clean Agile』図1.8 サークルオブライフ

受け入れテスト

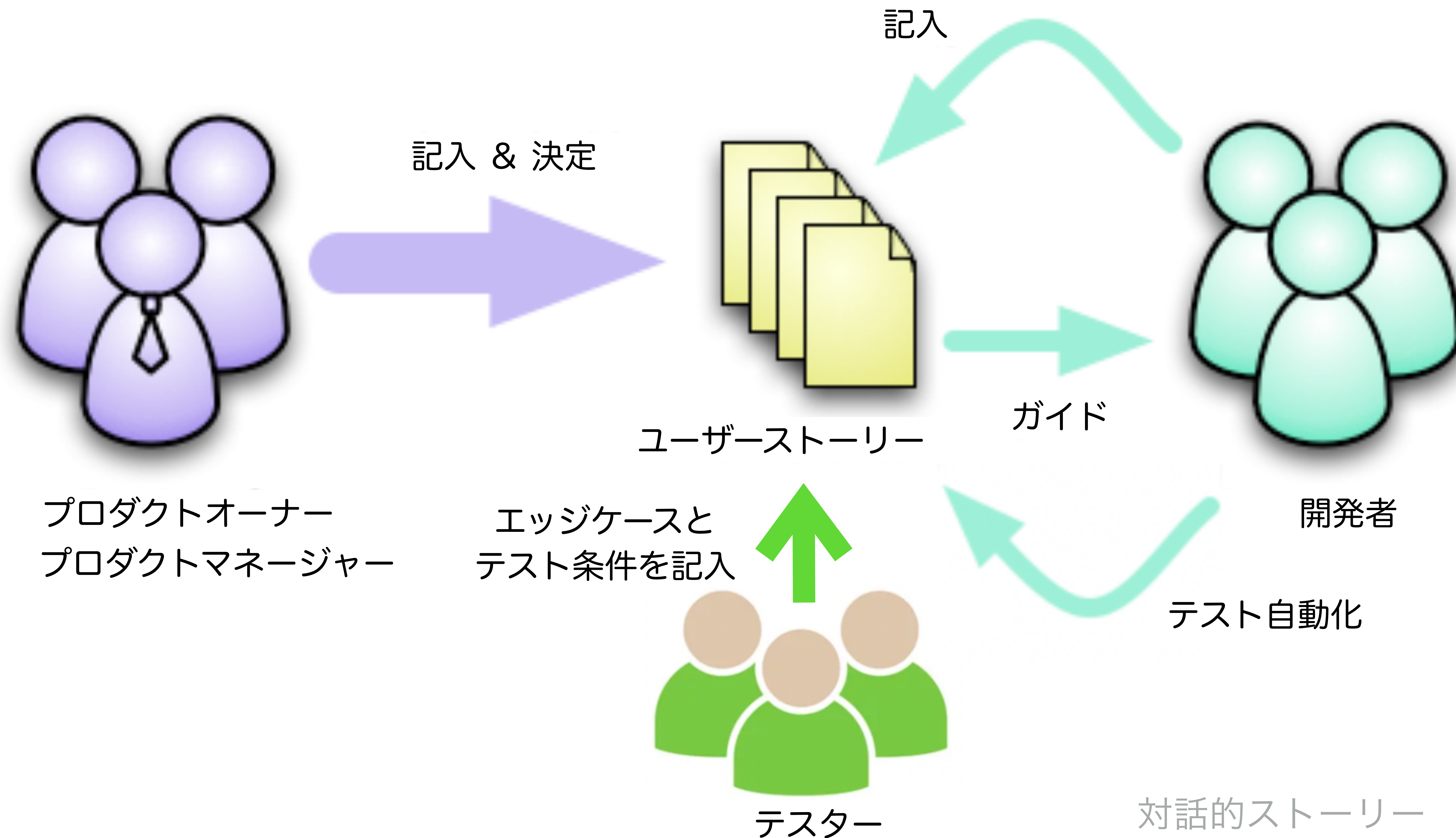
ストーリーを中心に置く



対話的ストーリー

https://bliki-ja.github.io/ConversationalStories/

ストーリーから受け入れテストへ



対話的ストーリー

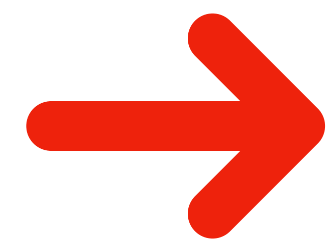
https://bliki-ja.github.io/ConversationalStories/

協力的プログラミング

ペアプログラミング

- Fred Brooks「大学院生時代(1953-56)にはじめてペアプログラミングをやってみた」
- Richard P. Gabriel「MIT人工知能研究所時代（1972-73）一般的に実施されていた」
- Larry Constantine「1980年代の"ダイナミックデュオ"という開発方法」
- Jim Coplien「1995年、ベル研究所におけるペアによる開発（組織パターン）」

出典：『Pair Programming Illuminated』Laurie Williams, Robert R. Kessler



昔からあるプログラミング方法だった

その後、モブプログラミングに引き継がれていく

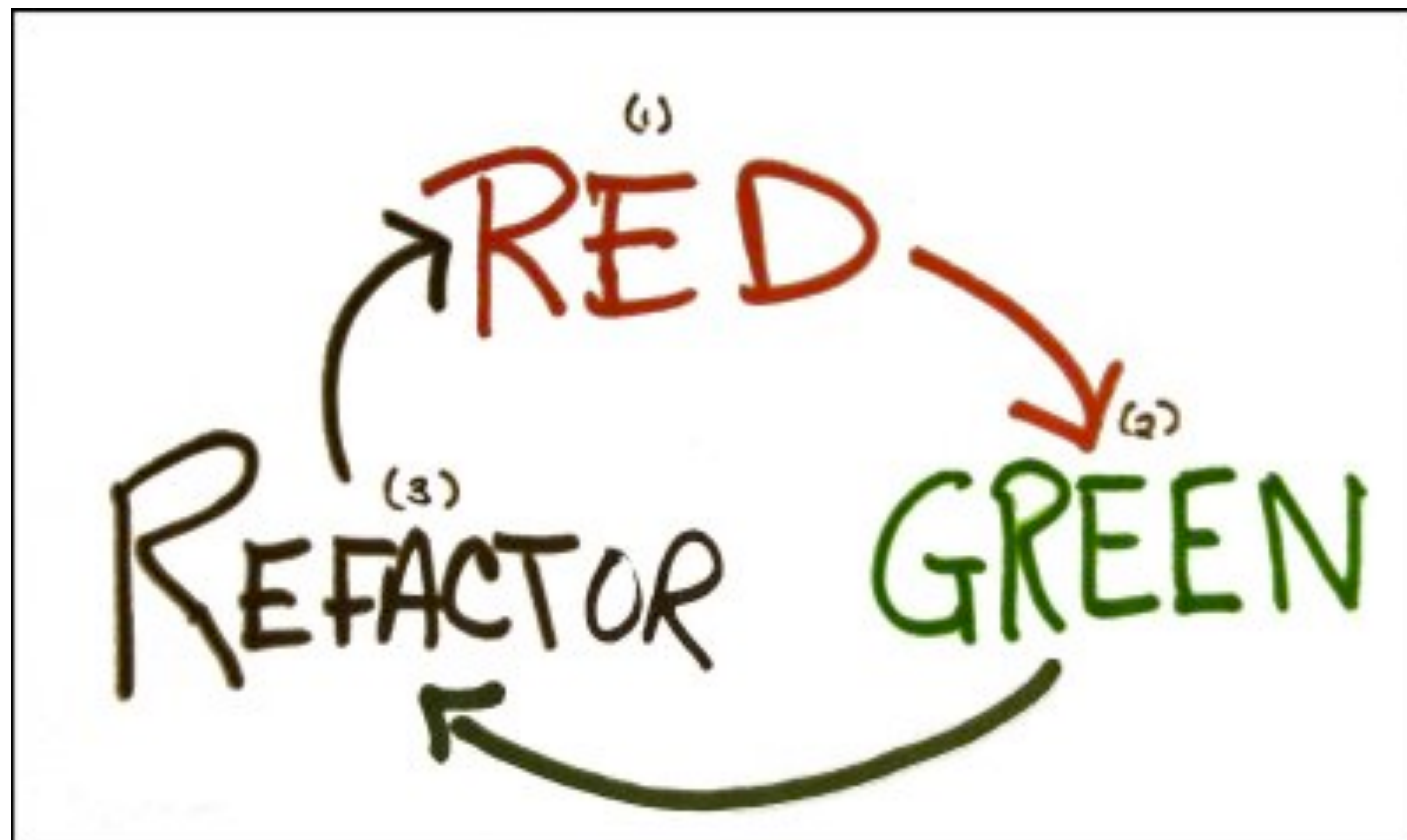
『Clean Coder』Bob C. Martin

TDD + リファクタリング

Waicrew
ワイクル株式会社

TDDのRGRサイクル

(1) 失敗するテストを書く



(3) コードをクリーンにする

(2) テストをパスさせるコードを書く

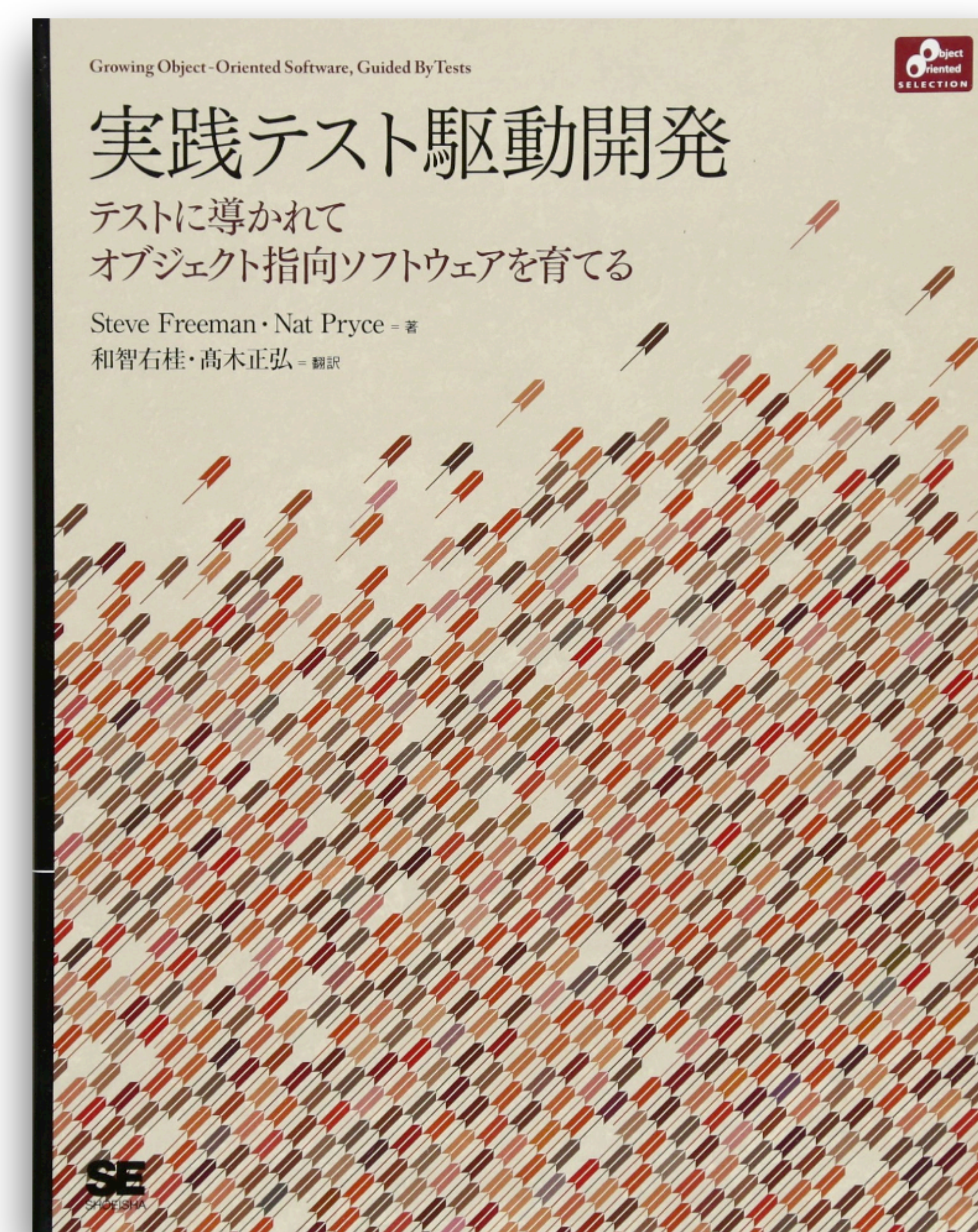
https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html

TDDはプロの前提条件

TDDを実践していなければ、プロのソフトウェア開発者になれない。

私は本気だ。というより、それが真実になりつつある。

Bob C. Martin『Clean Craftsmanship』



※TDDが唯一の解ではないとも言っている

補助線としてのニコマコス倫理学

アリストテレスの徳（卓越性）

- 思考の徳（知的徳）
  - ・エピステーメー（なぜを知る / 科学）
  - ・テクネ（いかにを知る / 工学）
  - ・フロネーシス（何をなすべきかを知る / 実践知, 状況判断）

参考：野中郁次郎・竹内弘高『ワイズカンパニー』

アリストテレスの徳（卓越性）

- 思考の徳（知的徳）
  - ・エピステーメー（なぜを知る / 科学）
  - ・テクネ（いかにを知る / 工学）
  - ・フロネーシス（何をなすべきかを知る / 実践知, 状況判断）
- 性格の徳（倫理的徳）
  - ・勇気、節制、友愛、正義など

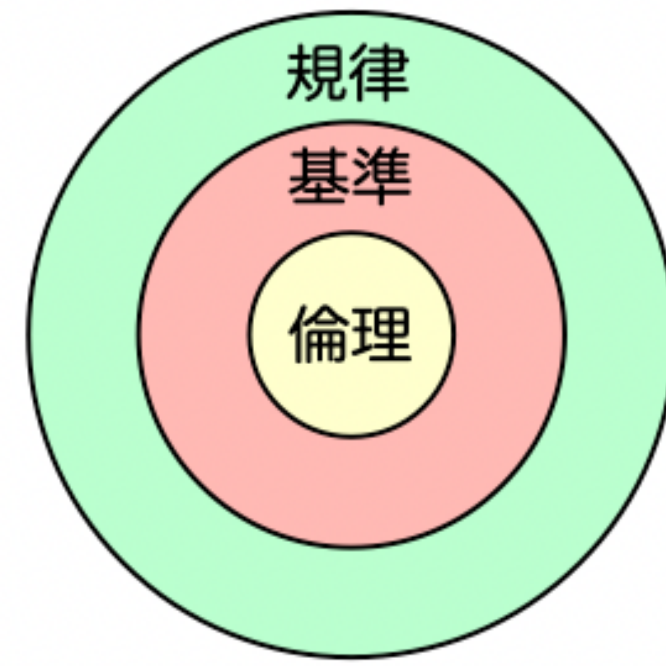
参考：野中郁次郎・竹内弘高『ワイズカンパニー』

日々の活動（習慣）が倫理を育む

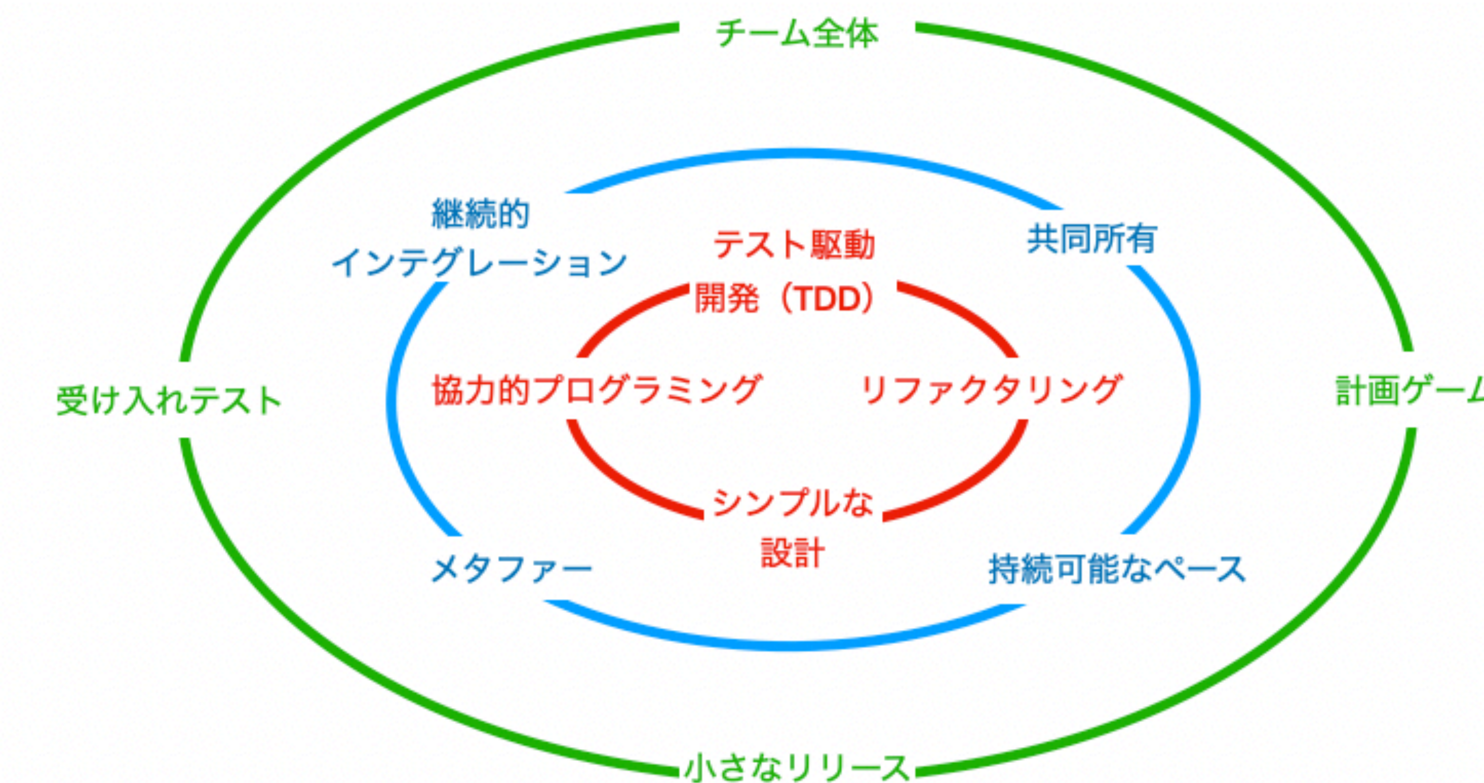
- "上手に家を建てることから人はすぐれた建築家になり、下手に建てることから劣悪な建築家になる"
- "要するに一言でいえば、同じような活動の反復から、人の性格の状態が生まれるのである"

→ TDD（RGRサイクル）の繰り返しから、良いプログラマの状態が生まれるのである

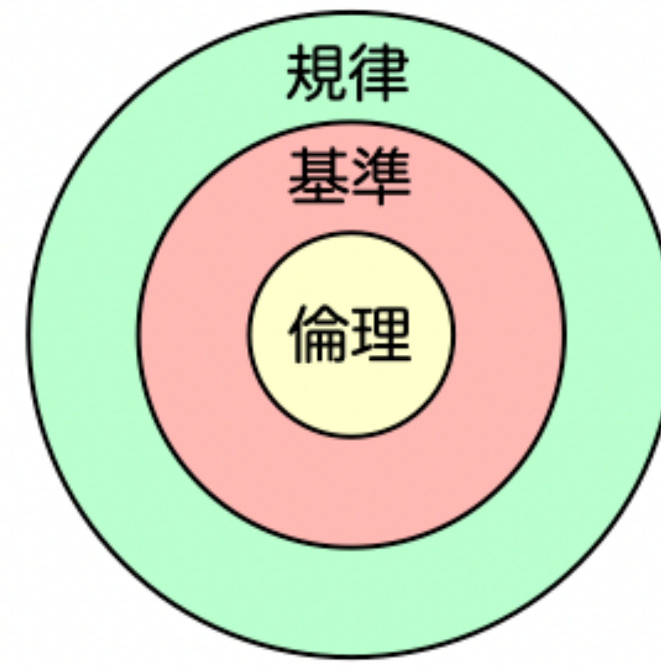
TDDから倫理が生まれる



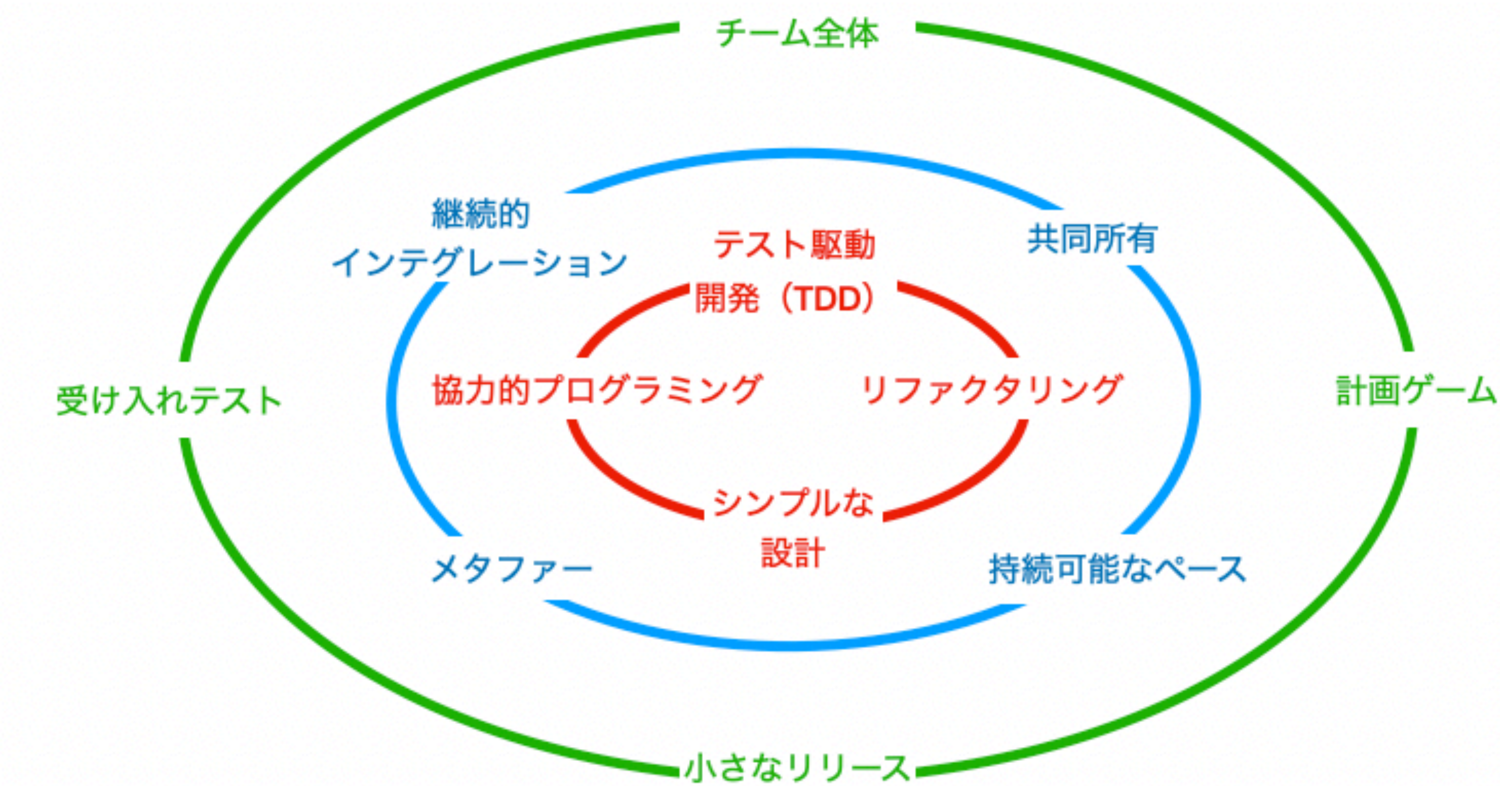
- 規律
  - テスト駆動開発、リファクタリング、シンプルな設計、協力的プログラミング（ペア + モブ）、受け入れテスト
- 基準
  - 生産性、品質、勇気
- 倫理
  - プログラマーの誓い（十戒）
    - 3つのテーマ（有害、誠実、チームワーク）



TDDから倫理が生まれる



- 規律
  - テスト駆動開発、リファクタリング、シンプルな設計、協力的プログラミング（ペア + モブ）、受け入れテスト
- 基準
  - 生産性、品質、勇気
- 倫理
  - プログラマーの誓い（十戒）
    - 3つのテーマ（有害、誠実、チームワーク）



クリーンコーダーの証「グリーンバンド」

テストがパスした「グリーン」を意味する

http://butunclebob.com/ArticleS.UncleBob.GreenWristBand

シンプルな設計

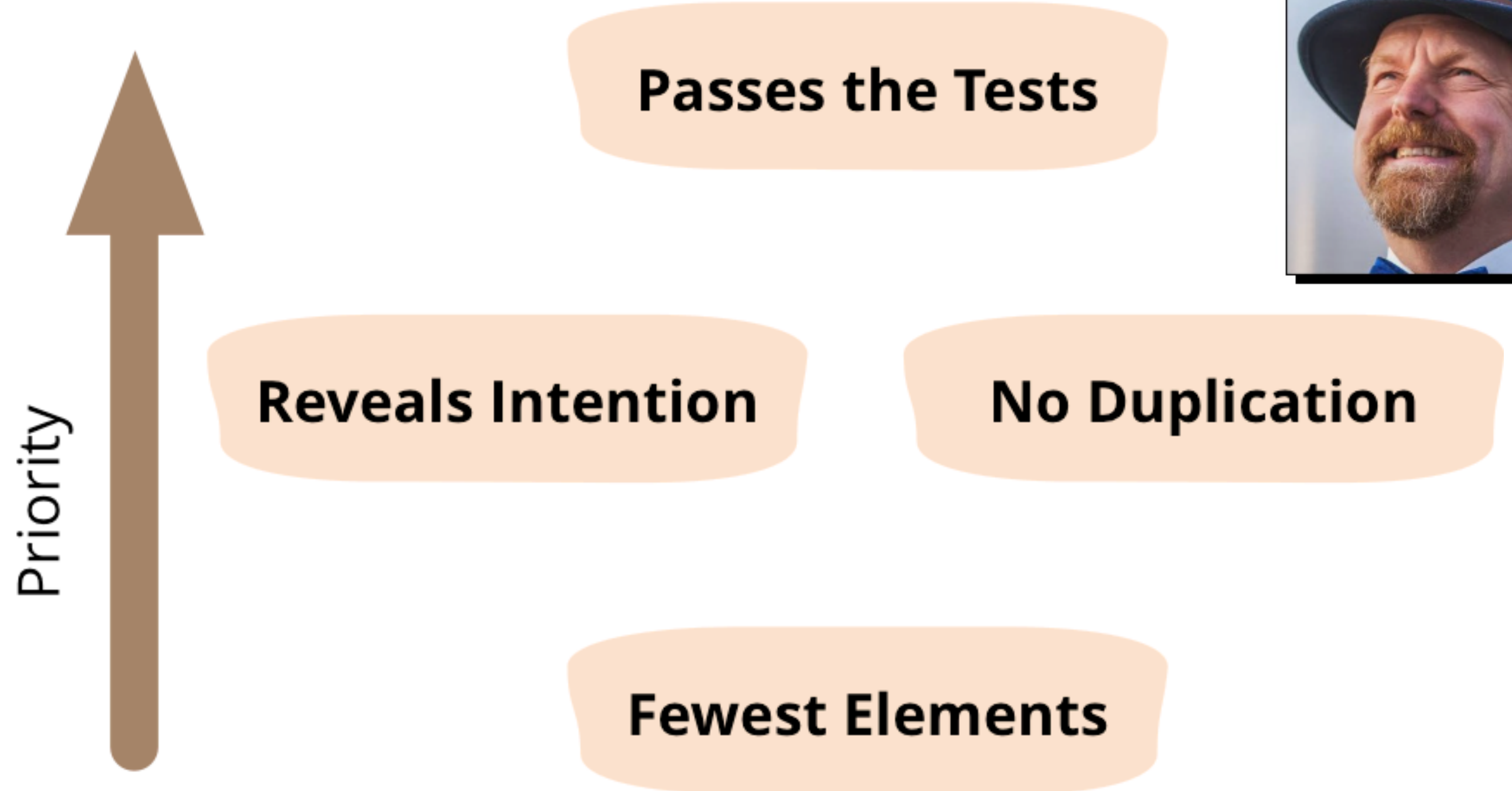
シンプルな設計　第6章

Bob C. Martin『Clean Craftsmanship』

3. シンプルな設計

Kent Beckの設計のルール

1. テストをパスさせる
2. 意図を明らかにする
3. 重複を排除する
4. 要素を最小限にする

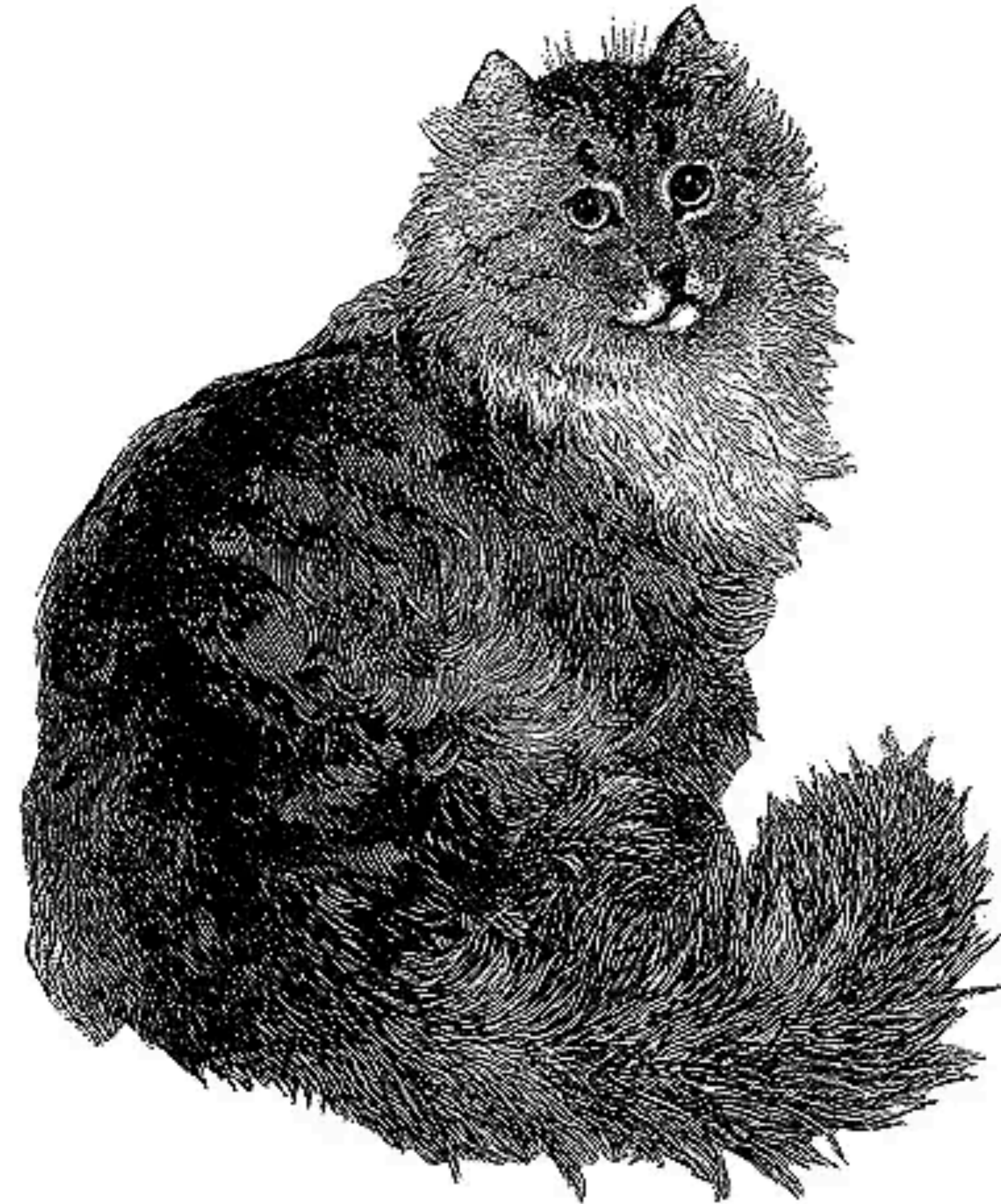


https://bliki-ja.github.io/BeckDesignRules/

O'REILLY®

Tidy First?

A Personal Exercise in Empirical Software Design



Kent Beck

ソフトウェアの設計とは？

Kent Beck『Tidy First?』

ソフトウェアの設計とは「人間関係のエクササイズ」である

Kent Beck『Tidy First?』

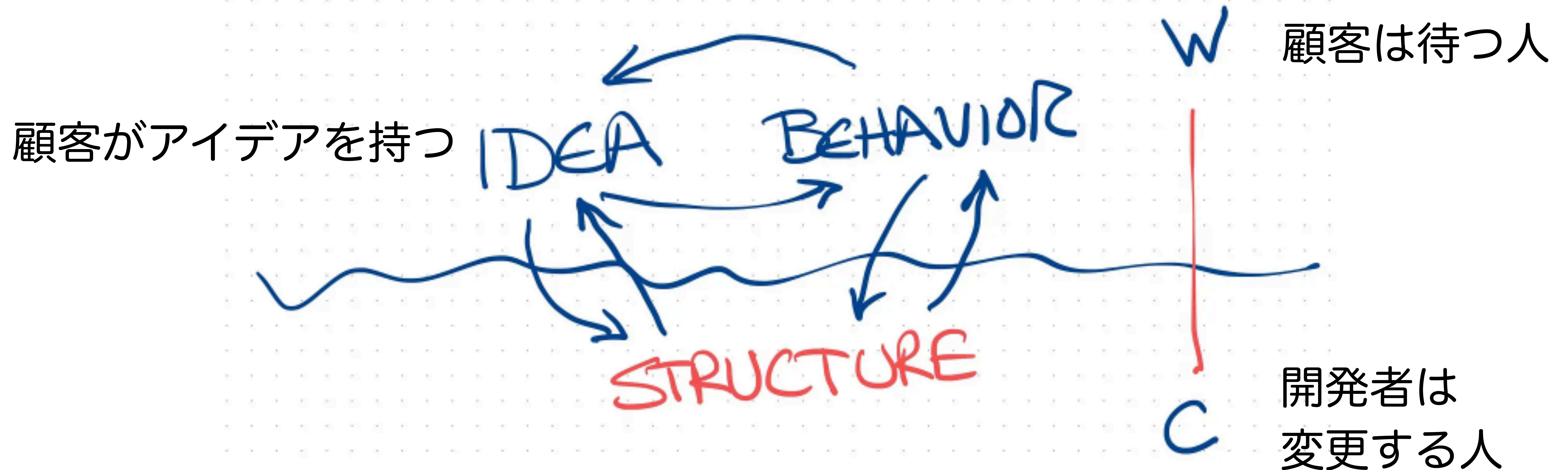
Waicrew
ワイクル株式会社

ソフトウェアの設計とは「人間関係のエクササイズ」である... 🤔 ???

Kent Beck『Tidy First?』

WとCの人間関係

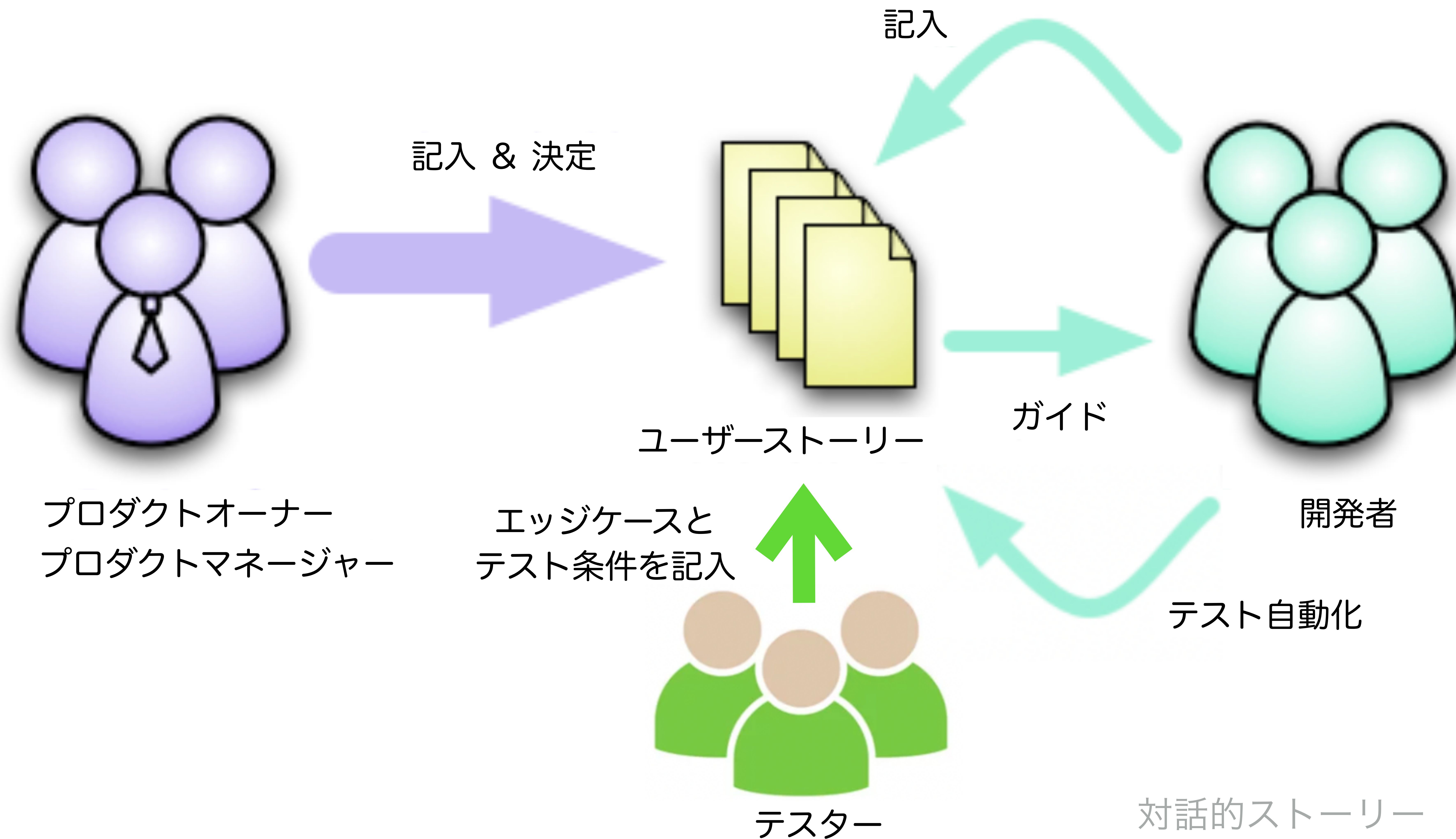
顧客は振る舞いを期待/理解する



ソフトウェアを理解/実装できるのは開発者だけ

https://tidyfirst.substack.com/p/behavior-change-revenue-versus-structure

振る舞い ≈ ユーザーストーリー



対話的ストーリー

https://bliki-ja.github.io/ConversationalStories/

振る舞いと構造のバランス

- 開発者が振る舞いを追加すれば、顧客は喜ぶ（= 収益になる）
- 顧客は「同じような機能」ならば「同じようなコスト」で追加したいと思う

Waicrew
ワイクル株式会社

振る舞いと構造のバランス

- 開発者が振る舞いを追加すれば、顧客は喜ぶ（= 収益になる）
- 顧客は「同じような機能」ならば「同じようなコスト」で追加したいと思う
  - だけど、構造の状態によって振る舞いの追加コストは変化する可能性がある
  - 「いやあ、先月までなら簡単に追加できたんですけどね...💦」

Waicrew ワイクル株式会社

振る舞いと構造のバランス

- 開発者が振る舞いを追加すれば、顧客は喜ぶ（= 収益になる）
- 顧客は「同じような機能」ならば「同じようなコスト」で追加したいと思う
  - だけど、構造の状態によって振る舞いの追加コストは変化する可能性がある
  - 「いやあ、先月までなら簡単に追加できたんですけどね…💦」
- 開発者が構造を変更しても、その時点では顧客は喜ばない（それはそう）

振る舞いを
追加する……！追加するが……
今回 まだ その時の指定までは
していない（AA省略）

振る舞いと構造のバランス（cont.）

- 開発者が構造を変更しても、その時点では顧客は喜ばない（それはそう）

振る舞いと構造のバランス（cont.）

- 開発者が構造を変更しても、その時点では顧客は喜ばない（それはそう）
  - だが、振る舞いの追加を容易にすることはできる（= オプションの提供）
  - そのために顧客はプレミアム（オプション価格）を支払う必要がある

👉 ソフトウェアの設計とは

「人間関係のエクササイズ」を通じて

振る舞いと構造（収益とオプション）

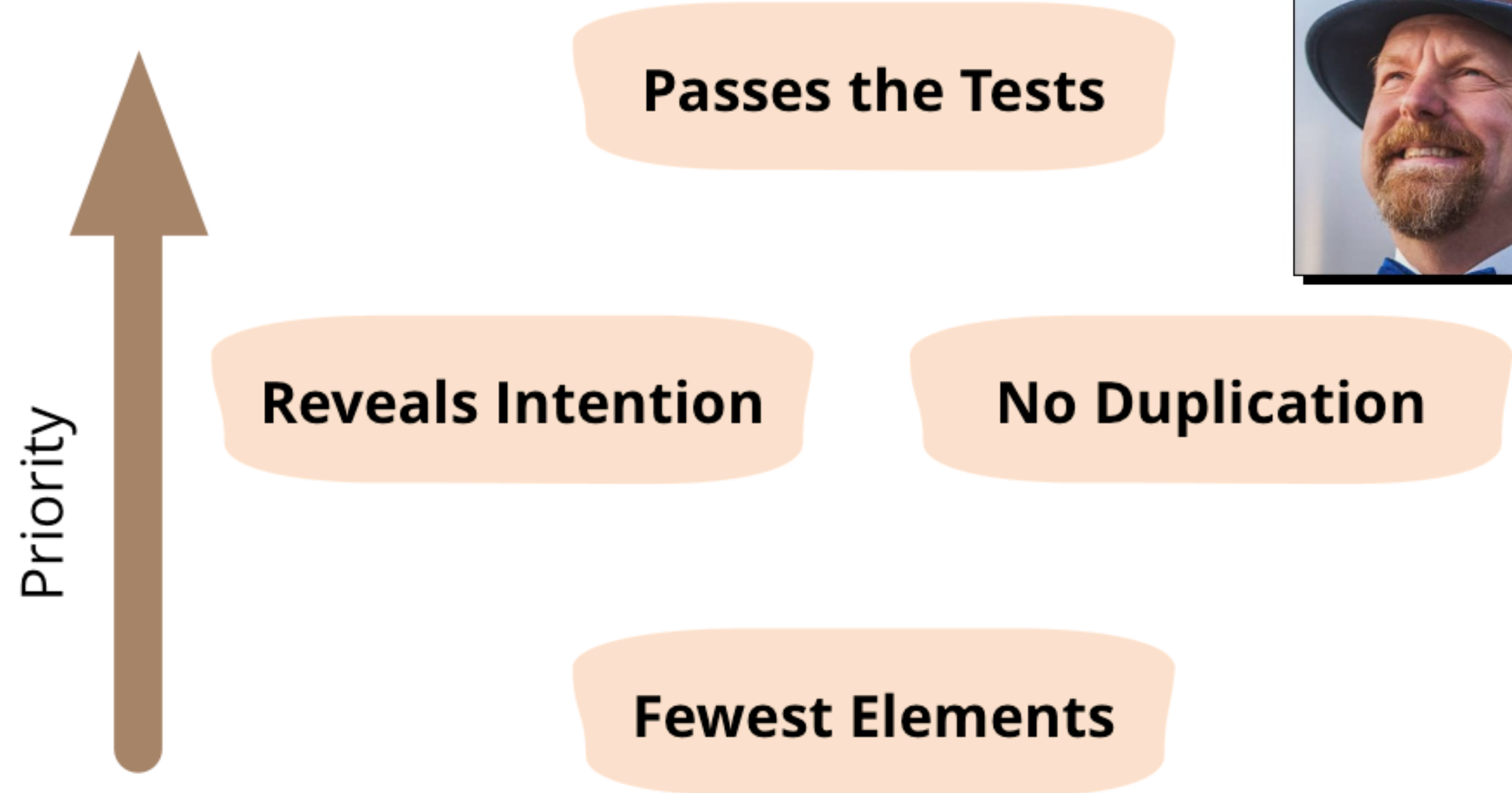
のバランスを取ることである

Kent Beck『Tidy First?』を参考にした

Paicrew ワイクル株式会社

Kent Beckの設計のルール

1. テストをパスさせる
2. 意図を明らかにする
3. 重複を排除する
4. 要素を最小限にする



https://bliki-ja.github.io/BeckDesignRules/

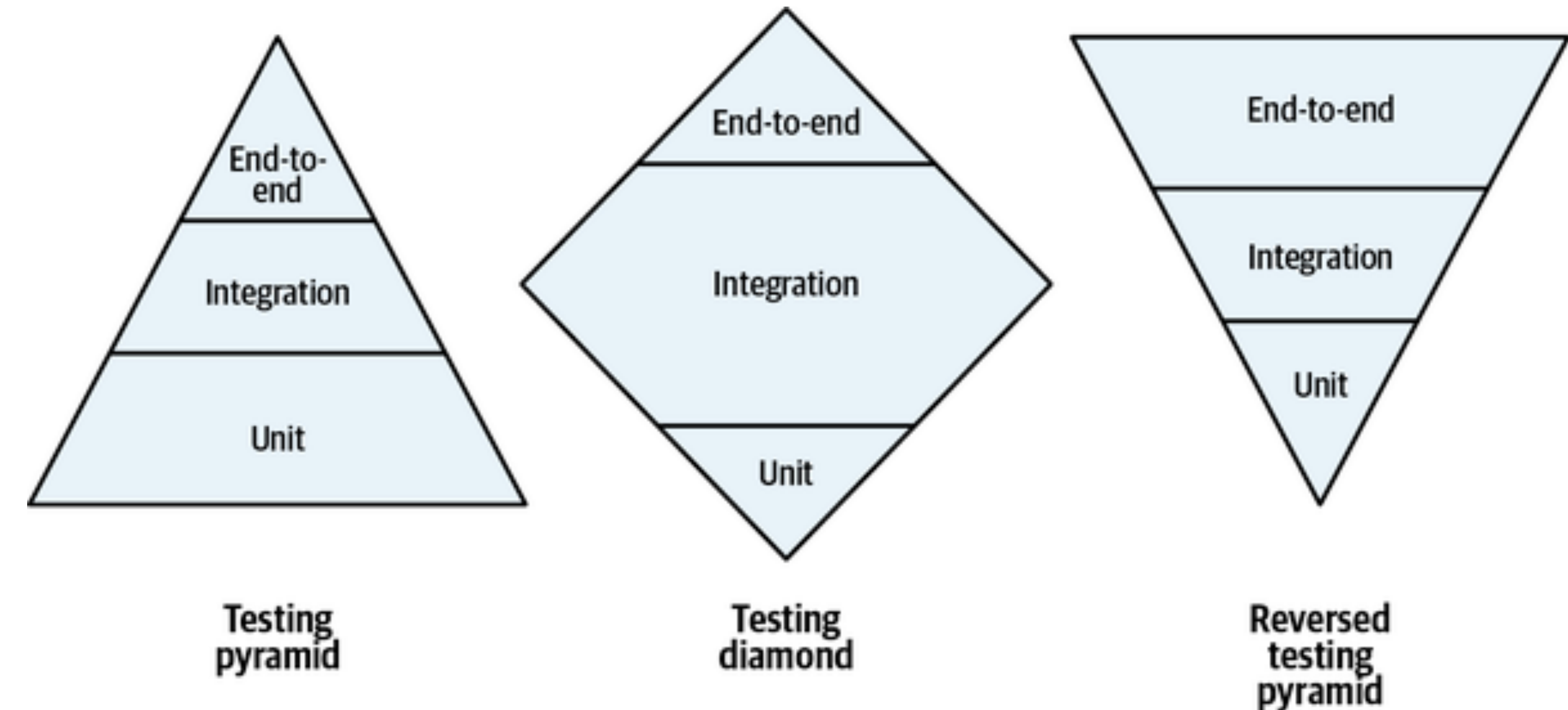
1. テストをパスさせる

- テストのカバレッジを高めやすいのが良い設計という話

・TDDとはまた違う（テストファーストや自動化を意味しない）

1. テストをパスさせる

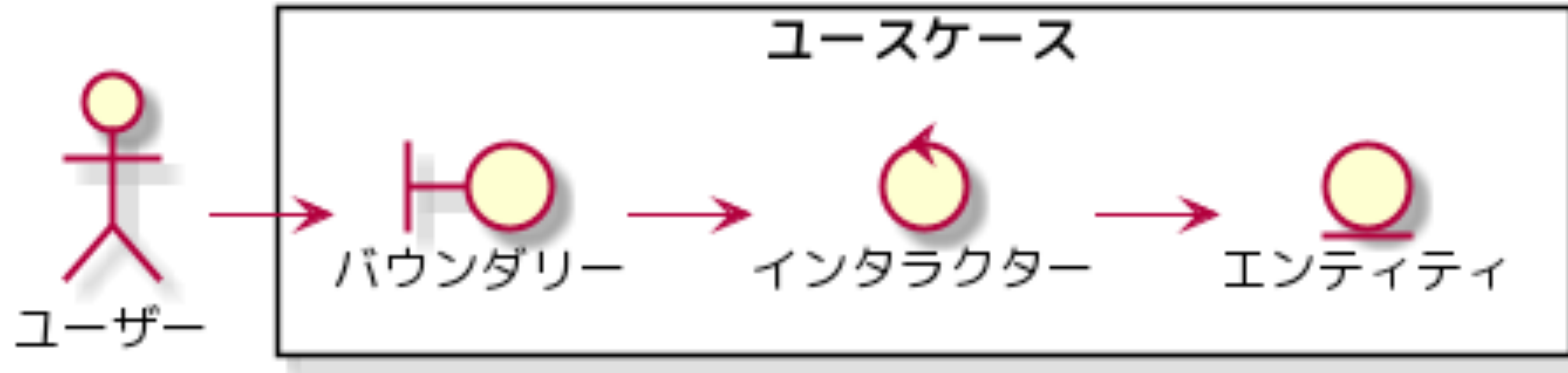
- テストのカバレッジを高めやすいのが良い設計という話
  - ・TDDとはまた違う（テストファーストや自動化を意味しない）
- 理想：方針（ビジネスロジック）は高く、詳細（UIやDB）は低く
  - ・ビジネスロジックが集まっていれば、
    主にユニットテストを重視すればいい



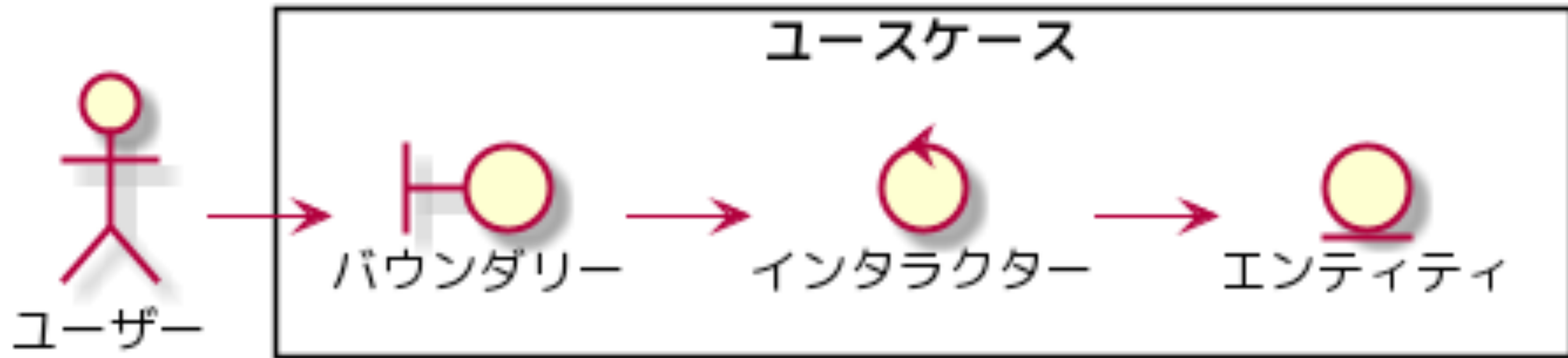
Vlad Khononov『Learning Domain-Driven Design』

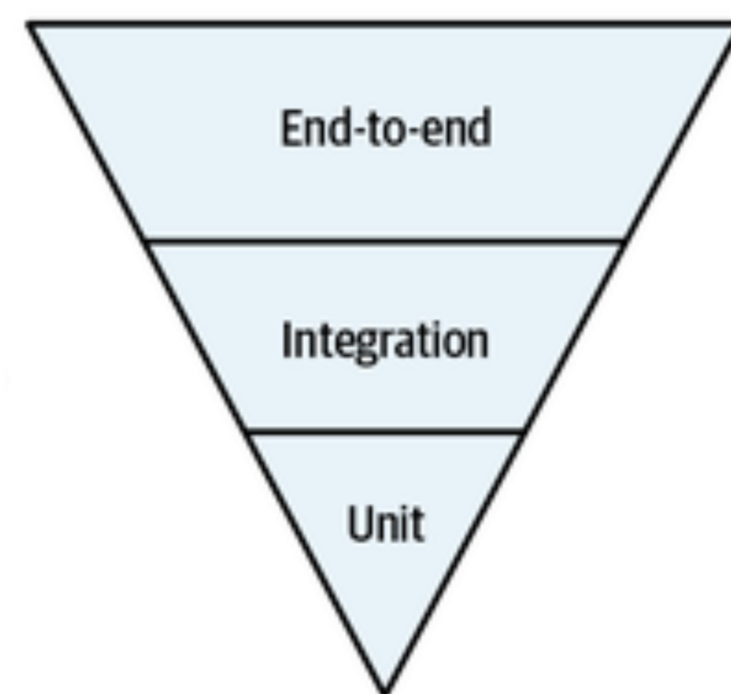
大事なのは振る舞い（ユースケース）

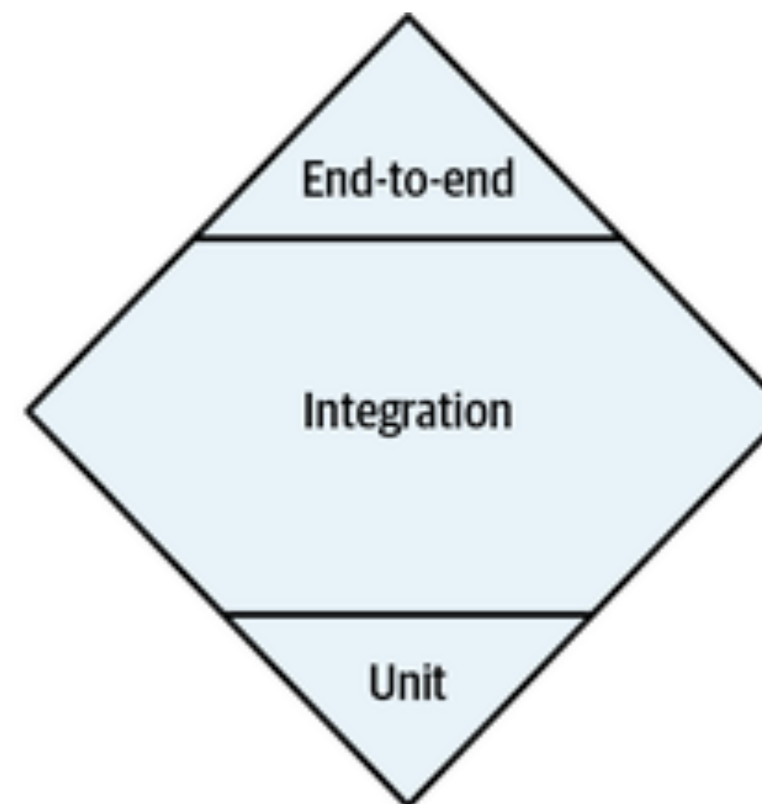
ユースケースを分割する（BCE)

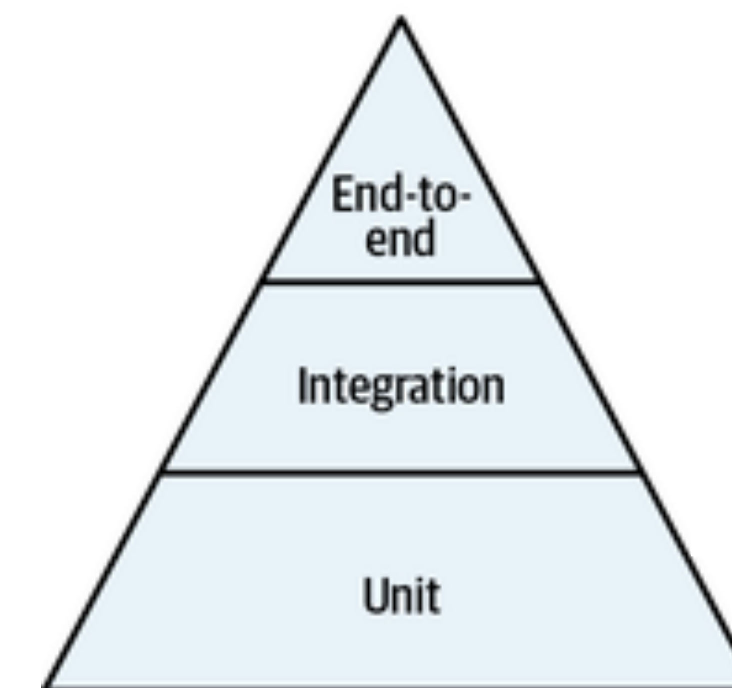


ビジネスロジックの場所とテスト戦略









Vlad Khononov『Learning Domain-Driven Design』

2. 意図を明らかにする

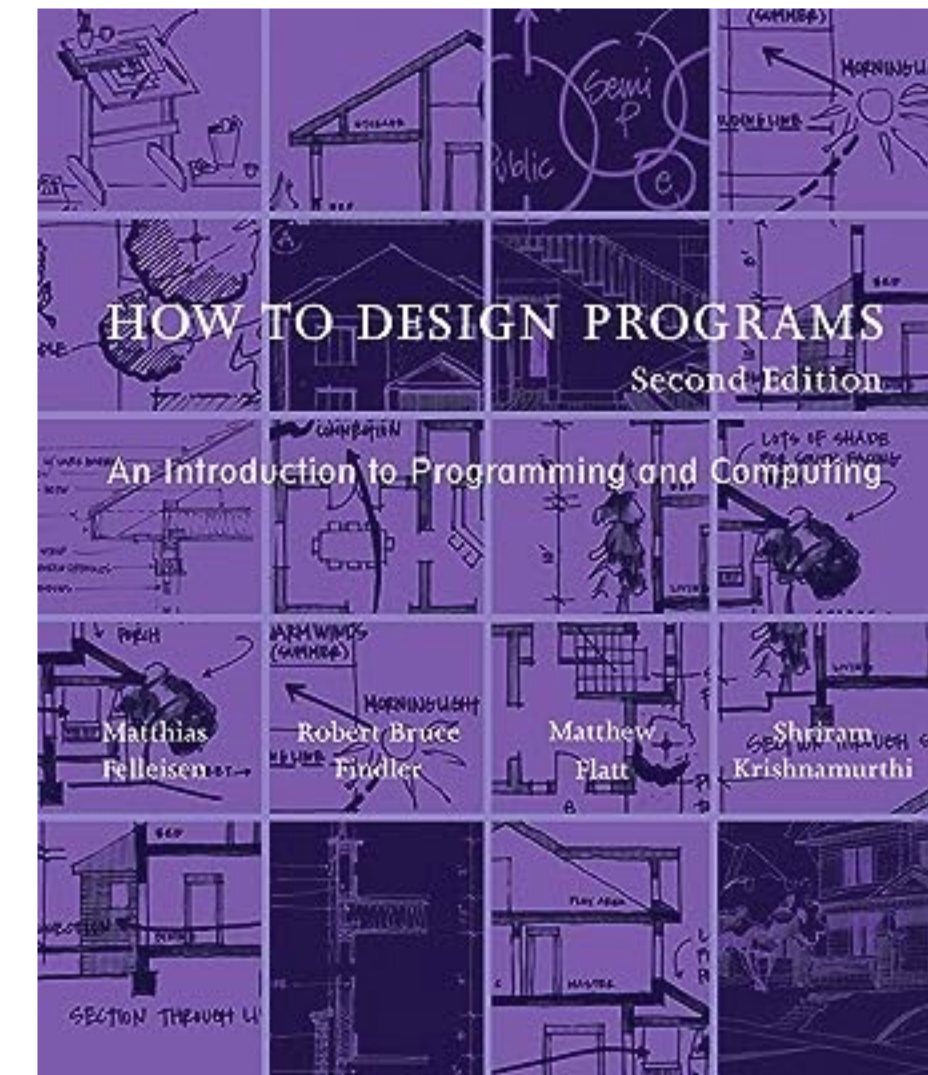
- ソフトウェアの設計は「人間関係のエクササイズ」なので…
  - 自分のために意図を明らかにする
  - 同僚のために意図を明らかにする
  - 顧客のために意図を明らかにする

*自分*のために意図を明らかにする

- テストコードも含めて自分にとってわかりやすくしておく
  - そうすれば、変更したいときに該当箇所が見つけやすくなる
- 未来の自分は他人「誰がこんなの書いたんだよ！？（俺だ）」
  - このへんは『リーダブルコード』という本もありますね（丸投げ）

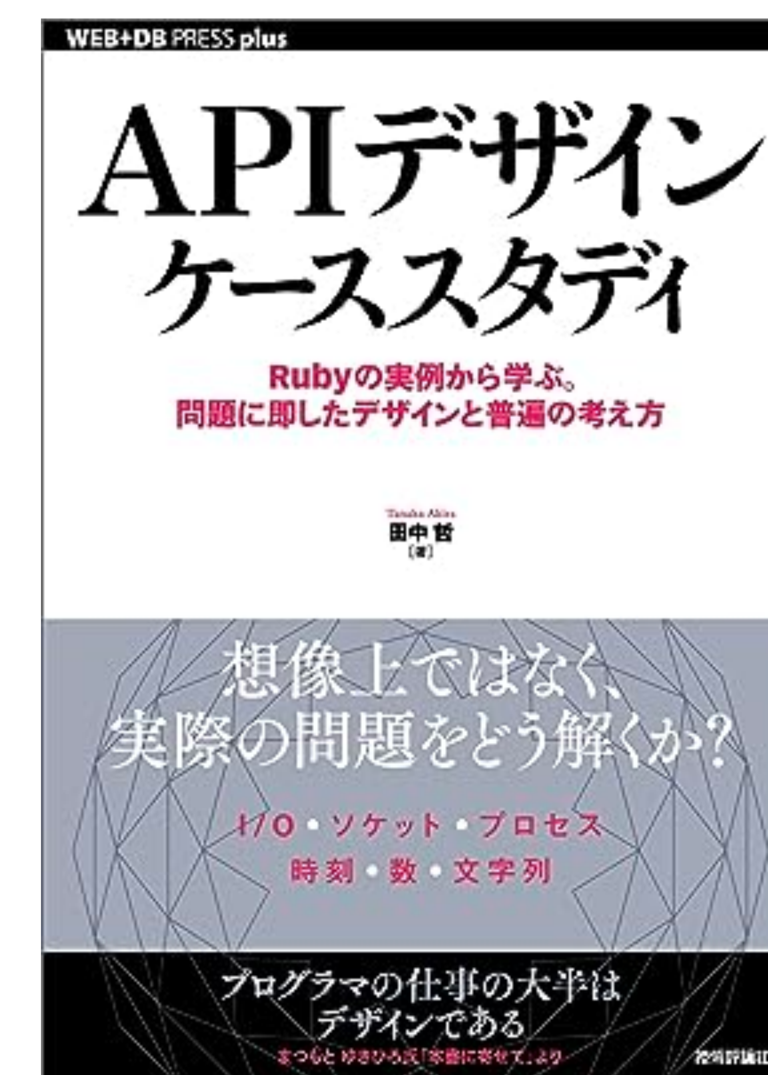
参考：デザインレシピ

- 問題分析からデータ定義へ
- 目的文、シグニチャ、ヘッダ
- 目的を説明する具体例
- テンプレート
- 定義（本体）
- テストコード



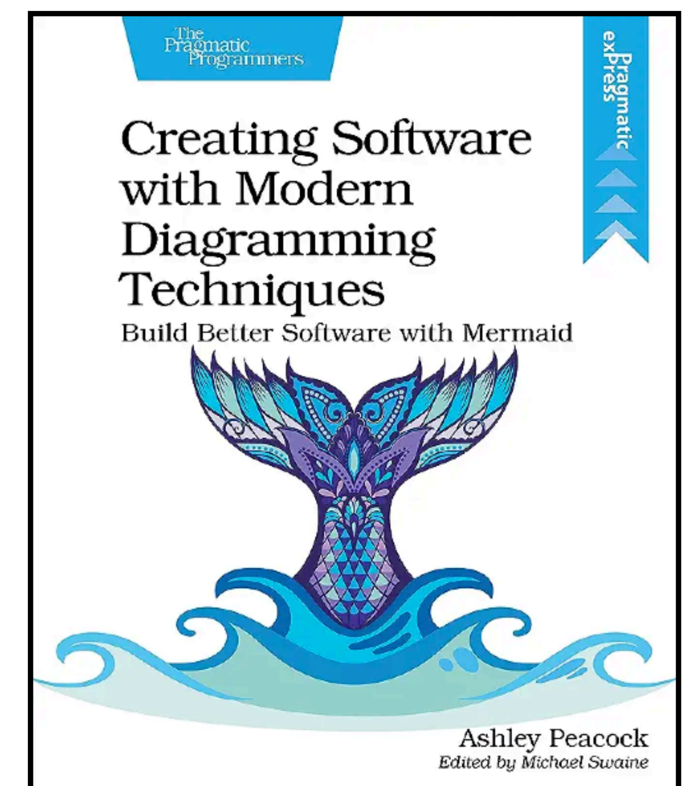
*同僚*のために意図を明らかにする

- 引き継ぎやレビューをしてもらいやすくなる
  - ・一人ひとりの理解や責任の範囲が広くなる → チームとして成長する
- 意図のわかりやすさよりも慣習のほうが大事なこともある
  - ・チームのルールを決める
  - ・標準APIを読む
  - ・UNIX文化に触れる
  - ・実際のコードを読む（or Copilotで支援してもらう）



*顧客*のために意図を明らかにする

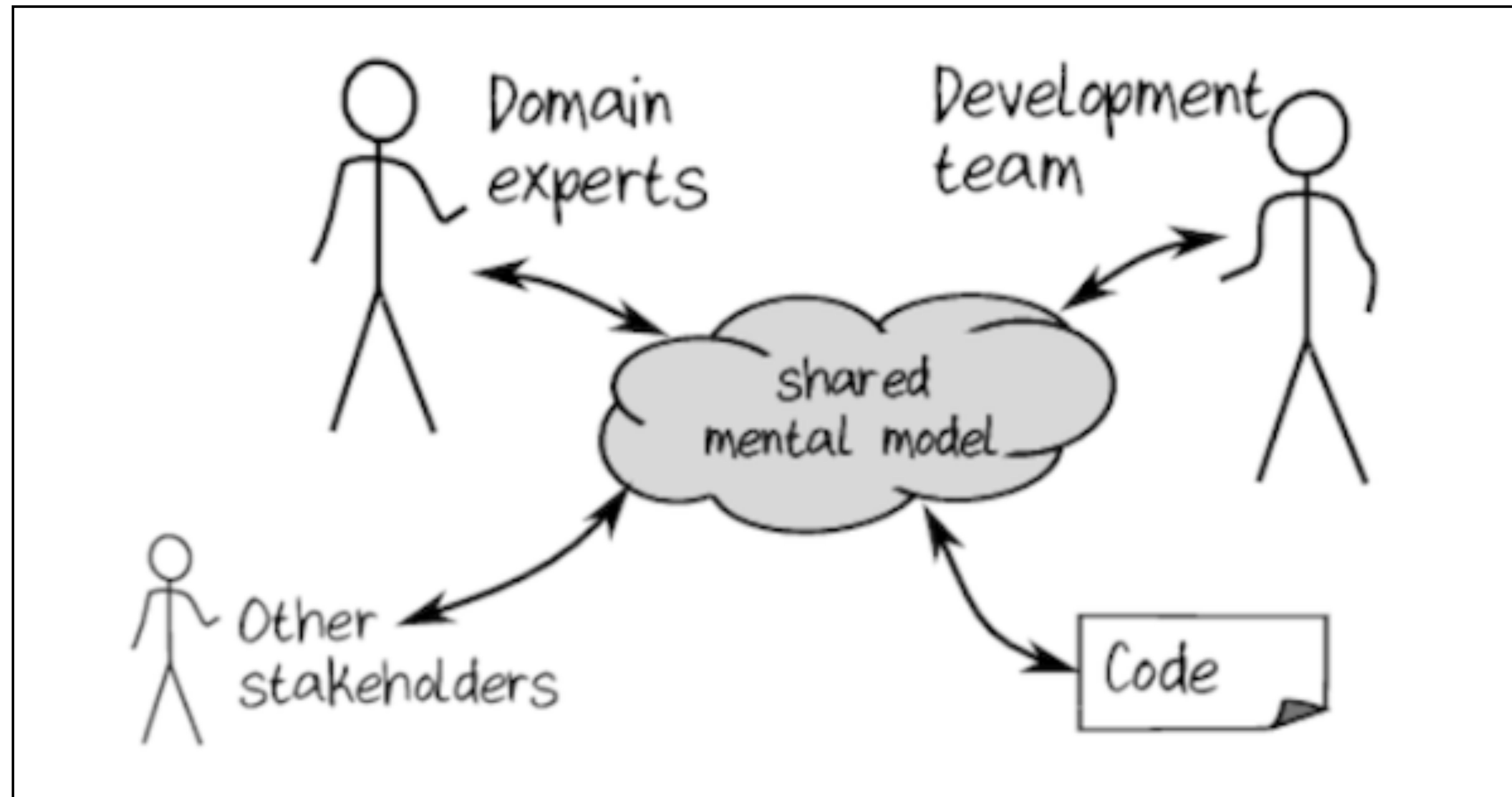
- 顧客と意思疎通しやすくしておく
  - ・ユーザーストーリーやユースケースはひとつの方法
- もちろん顧客にコードを読んでもらう必要はない
  - ・重要なのは構造ではなく振る舞い（n回目）
- Mermaidで振る舞いの流れ（e.g. シーケンス図）を描くとよい



Sequence diagrams, the only good thing UML brought to software development
<https://www.mermaidchart.com/blog/posts/sequence-diagrams-the-good-thing-uml-brought-to-software-development>

*顧客*のために意図を明らかにする（cont.）

共有メンタルモデルを作成する



『Domain Modeling Made Functional』

3. 重複を排除する

- DRY原則「すべての知識はシステム内において、単一、かつ明確な、そして信頼できる表現になっていなければならない」（『達人プログラマー』）

3. 重複を排除する

- DRY原則「すべての知識はシステム内において、単一、かつ明確な、そして信頼できる表現になっていなければならない」（『達人プログラマー』）
  - ・コードをコピペすんなという話ではない
  - ・DRY原則はコード以外にも適用される（p. 40）
    - ・ドキュメント、データ構造、APIのバインディング
      - ・知識をコードに落とし込んでおくとよい

例：ビジネスルールの置き場所

- 「ライフゲーム」のルールを考えてみる（Wikipediaより引用）
  - 誕生：死んでいるセルに隣接する生きたセルがちょうど3つあれば、次の世代が誕生する
  - 生存：生きているセルに隣接する生きたセルが2つか3つならば、次の世代でも生存する
  - 過疎：生きているセルに隣接する生きたセルが1つ以下ならば、過疎により死滅する
  - 過密：生きているセルに隣接する生きたセルが4つ以上ならば、過密により死滅する

ライフゲームの基本ルール

誕生	生存（維持）	死（過疎）	死（過密）

実装で重複を排除するよりも…

```
nextState(liveNeighbors) {
  if (this.isAlive() && (liveNeighbors < 2 || liveNeighbors > 3)) {
    this.nextState = State.DEAD;  // Rules 3 and 4
  } else if (this.isDead() && liveNeighbors == 3) {
    this.nextState = State.ALIVE;  // Rule 1
  } else {
    this.nextState = this.currentState;  // Rule 2 and all other cases
  }
}
```

Waicrew
ワイクル株式会社

…ルールの記述をコードに残す

```
nextState(liveNeighbors) {
  if (this.isDead() && liveNeighbors == 3) { // Rule 1
    this.nextState = State.ALIVE;
  }
  else if (this.isAlive() &&
     (liveNeighbors == 2 || liveNeighbors == 3)) { // Rule 2
    this.nextState = State.ALIVE;
  }
  else if (this.isAlive() && liveNeighbors <= 1) { // Rule 3
    this.nextState = State.DEAD;
  }
  else if (this.isAlive() && liveNeighbors >= 4) { // Rule 4
    this.nextState = State.DEAD;
  }
  else {
    this.nextState = this.currentState;
  }
}
```

※意図の反映は重複に勝る

4. 要素を最小限にする

- これまでの3つのルールに当てはまらないものは削除する

4. 要素を最小限にする

- これまでの3つのルールに当てはまらないものは削除する
- たとえば、柔軟性を確保するために要素を追加しない

4. 要素を最小限にする

- これまでの3つのルールに当てはまらないものは削除する
- たとえば、柔軟性を確保するために要素を追加しない
  - ・インターフェイスを挟もうとするクリーンアーキテクチャは？
  - ・「要素を最小限にする」の優先順位は最も低い
    - ・意図が明確に伝わるなら、重複や余計な要素があっても構わない

4. 要素を最小限にする

- これまでの3つのルールに当てはまらないものは削除する
- たとえば、柔軟性を確保するために要素を追加しない
  - ・インターフェイスを挟もうとするクリーンアーキテクチャは？
  - ・「要素を最小限にする」の優先順位は最も低い
    - ・意図が明確に伝わるなら、重複や余計な要素があっても構わない
  - ・「柔軟性を確保したい」は（ある意味）未来を予測しようとしている
    - ・未来を予測できるならOK … だけど？

未来を予測しない by Kent Beck

- 変化を予測すればするほど、変化を起こすのが難しくなる
  - 予測する → 設計が悪くなる → 予測する → 設計が悪くなる …

『Understanding the Four Rules of Simple Design』

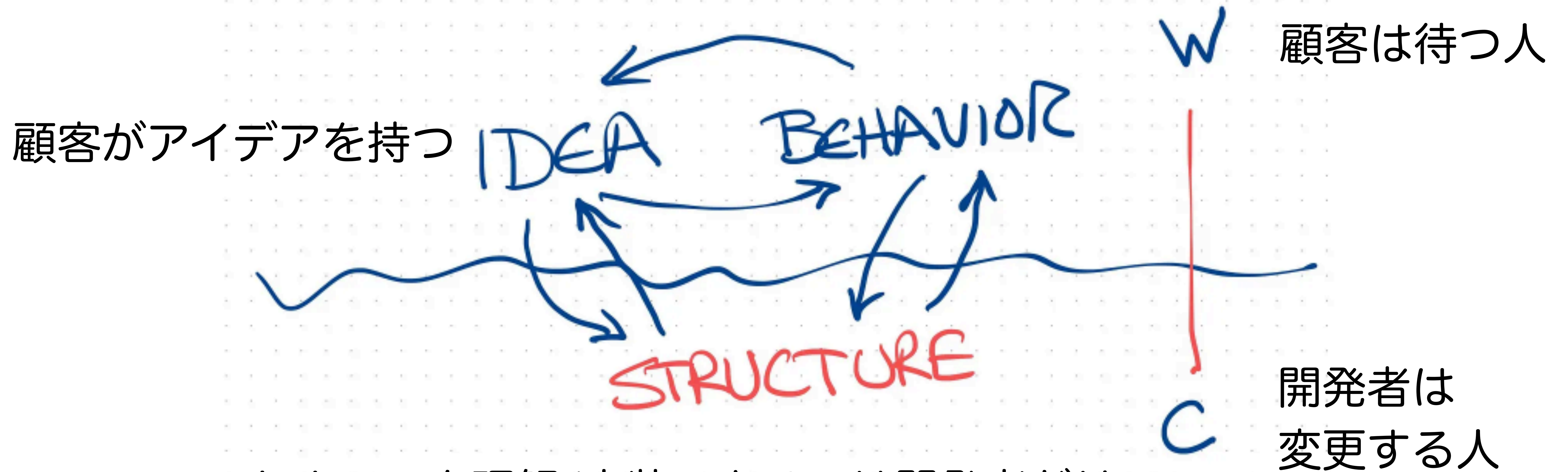
Waicrew ワイクル株式会社

未来を予測しない by Kent Beck

- 変化を予測すればするほど、変化を起こすのが難しくなる
  - ・予測する → 設計が悪くなる → 予測する → 設計が悪くなる ...
- 現時点で要求されていない設計要素を排除してみよう
  - ・とりあえず、半年以上先のことを予測するのをやめてみる → よくなった
  - ・3か月なら？ 1か月なら？ 1週間なら？ 1日なら？ → どんどんよくなった

『Understanding the Four Rules of Simple Design』

WとCの人間関係のエクササイズ



https://tidyfirst.substack.com/p/behavior-change-revenue-versus-structure

まとめ

まとめ

- Clean Craftsmanship
  - ソフトウェア開発者の社会的責任の増加
  - 個人で頑張るだけでなく、次世代の人たちに伝えるための業界のルールも必要になっていく
- 規律
  - 受け入れテスト、協力的プログラミング、TDD+リファクタリング、シンプルな設計
  - 普段の行動や習慣（規律）が倫理につながるはず
- シンプルな設計
  - 設計とは 「人間関係のエクササイズ」を通じて、振る舞いと構造（収益とオプション） のバランスを取ること
  - テストをパスさせる、意図を明らかにする、重複を排除する、要素を最小限にする
  - 未来を予測せずに、変化に適応できるようにコードをアレンジする

読んでみてください！