アンクル・ボブに学ぶ クラフトマンシップ

ミイダス株式会社 様

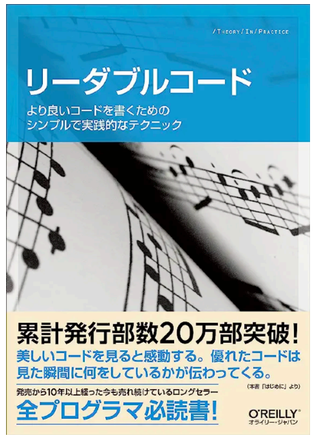
2025年11月28日

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

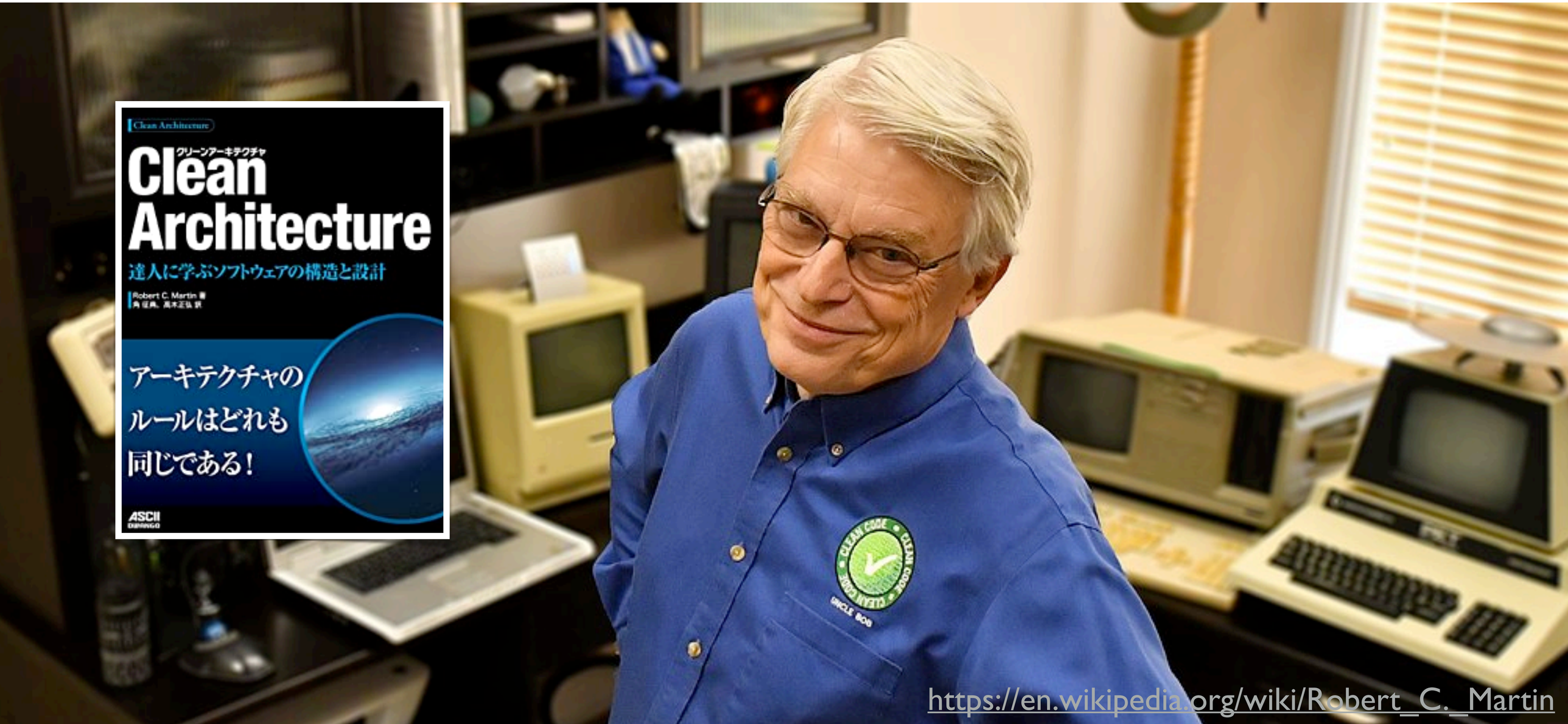
講師の紹介

- 角 征典（@kdmsnr）
  - 技術書の翻訳・執筆 →



- ワイクル株式会社 代表取締役
  - アジャイル開発／リーンスタートアップの導入支援
- 東京科学大学 非常勤講師
  - エンジニアのためのデザイン思考

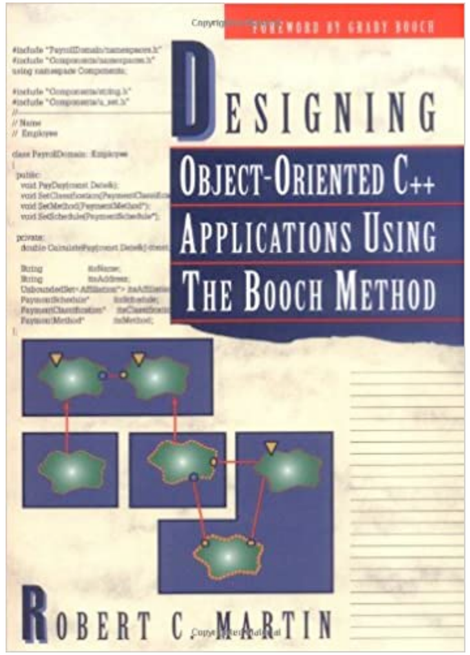
アンクル・ボブ（1952~）

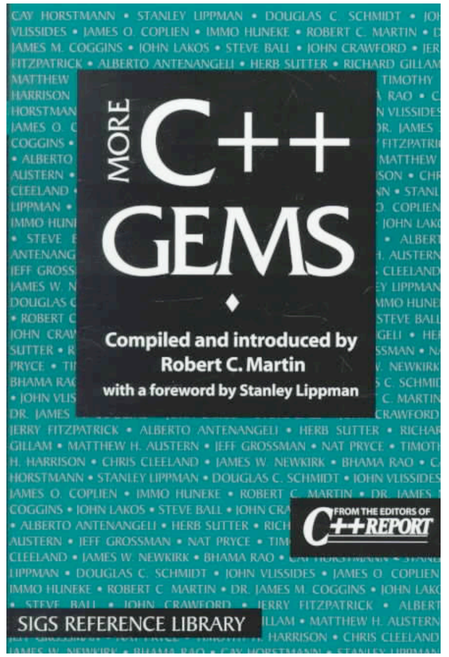


https://en.wikipedia.org/wiki/Robert_C._Martin

アンクル・ボブの歴史（前半）

- 1952年生まれ
- 1970年：高校卒業後に就職
- 1972年(?)：結婚、解雇、再就職
- 1976～1988年：Teradyne社（大企業）
- 1986～1990年：同僚のスタートアップにジョイン
  - この頃にC++とオブジェクト指向に出会う
- 1990～1991年：フリーランスとしてRationalで仕事
  - Grady BoochとUMLツール開発
- 1991～2010年：Object Mentor社（起業）
  - 受託開発、コンサルティング、教育
- 1996～1999年：『C++ Report』誌編集長

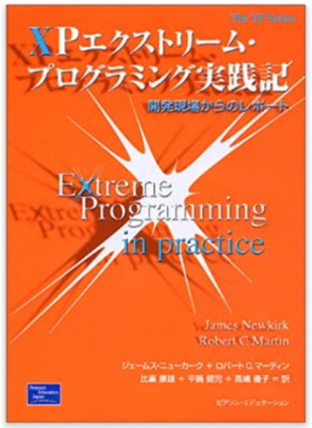


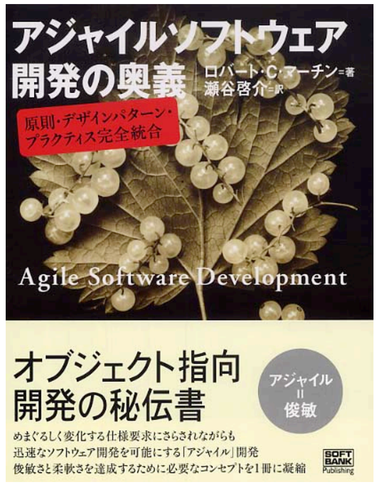


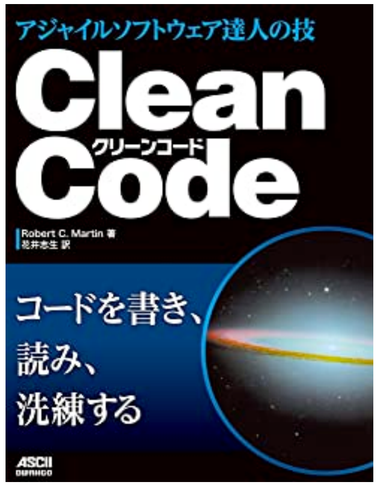
アンクル・ボブの歴史（後半）

- 1999年：Kent BeckとTDDに出会う
- 2001年：アジャイルマニフェストの起案
- 2001年：ドットコムバブル崩壊
- 2008年：リーマン・ショック
  - ・Object Mentor社が倒産
- 2010年：動画制作を開始

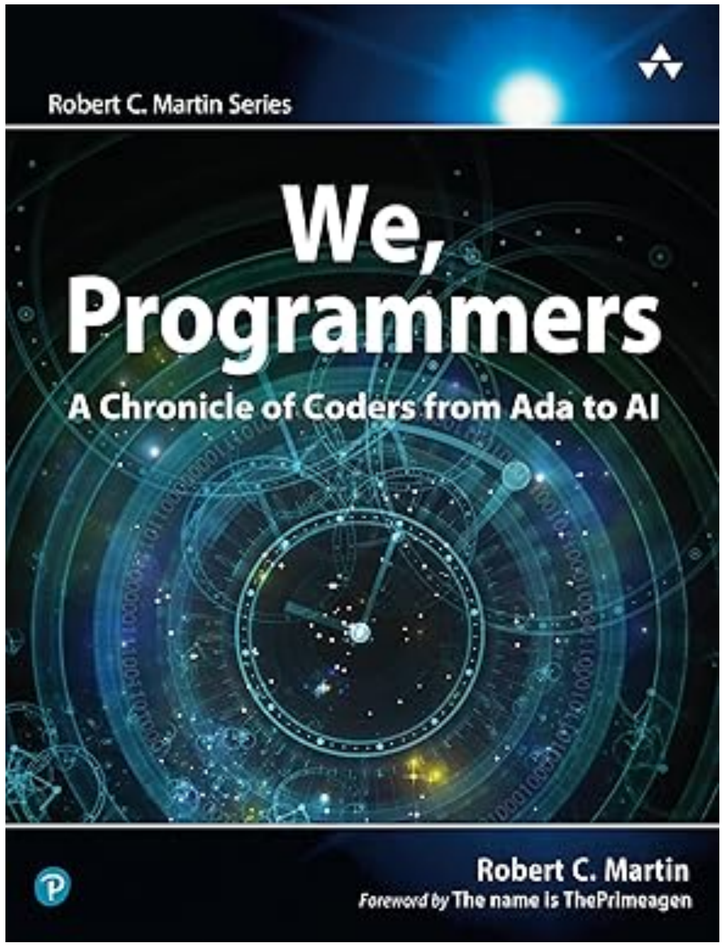
書籍執筆は継続中...

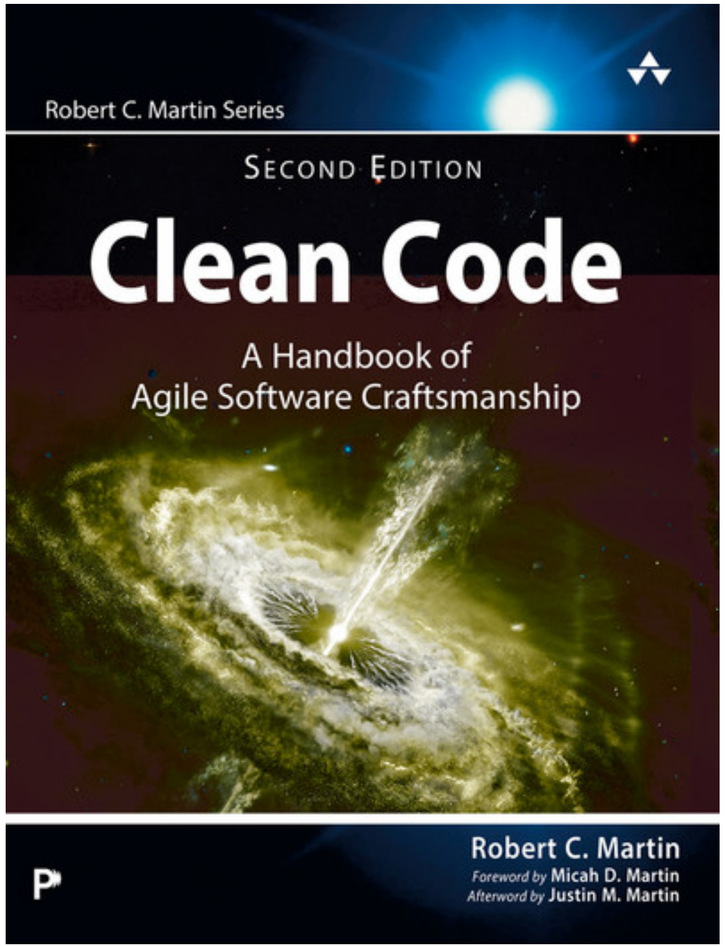






次々と出る新刊（日本語版は翻訳中）





きょう取り上げる書籍


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

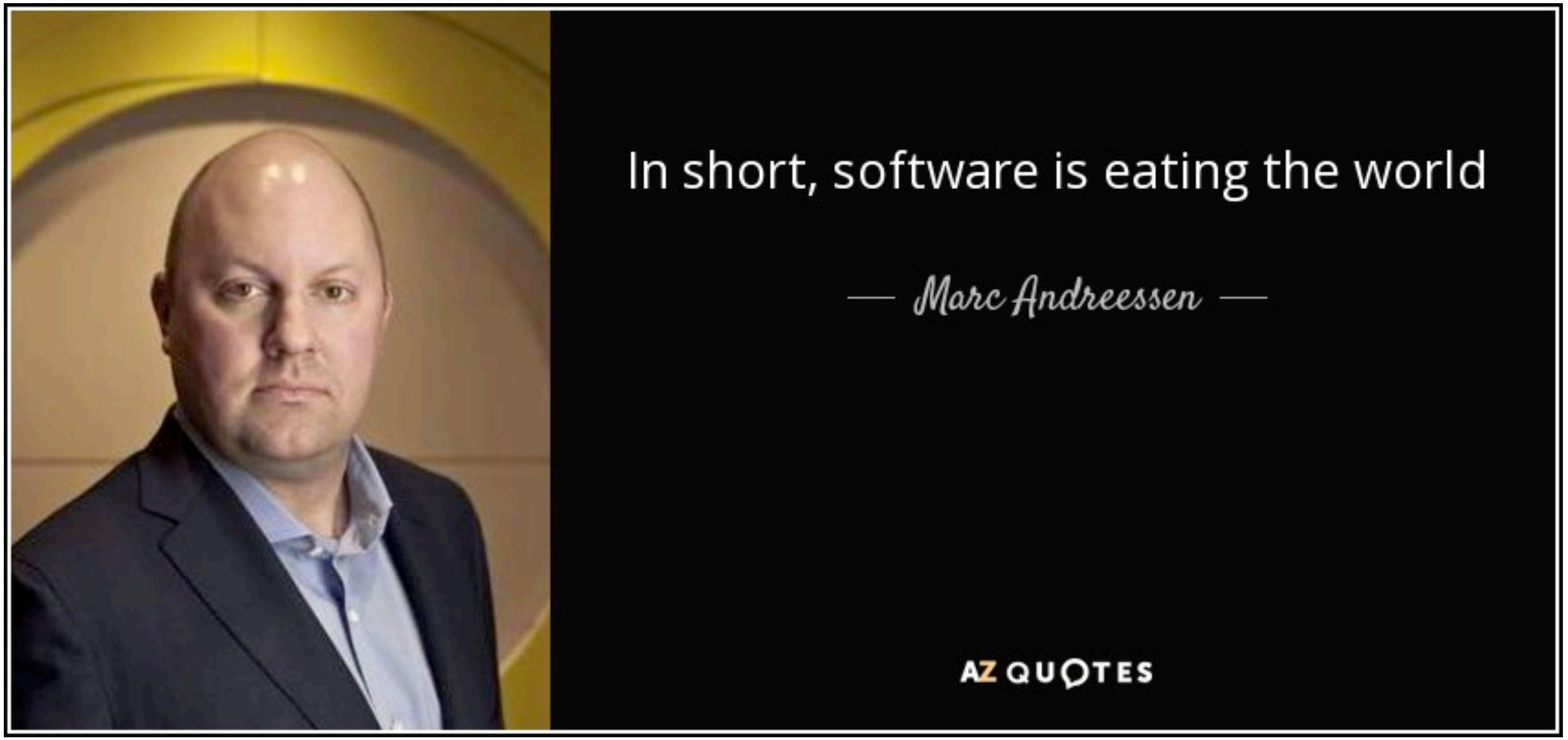

Clean Craftsmanship
クリーンクラフトマンシップ
Clean Craftsmanship
規律、基準、倫理
Robert C. Martin 著
角 征典 訳
より良く働き
生産性を高め
自分が書いたものに
誇りを持つ！
ASCII DWANGO

きょうお話すること

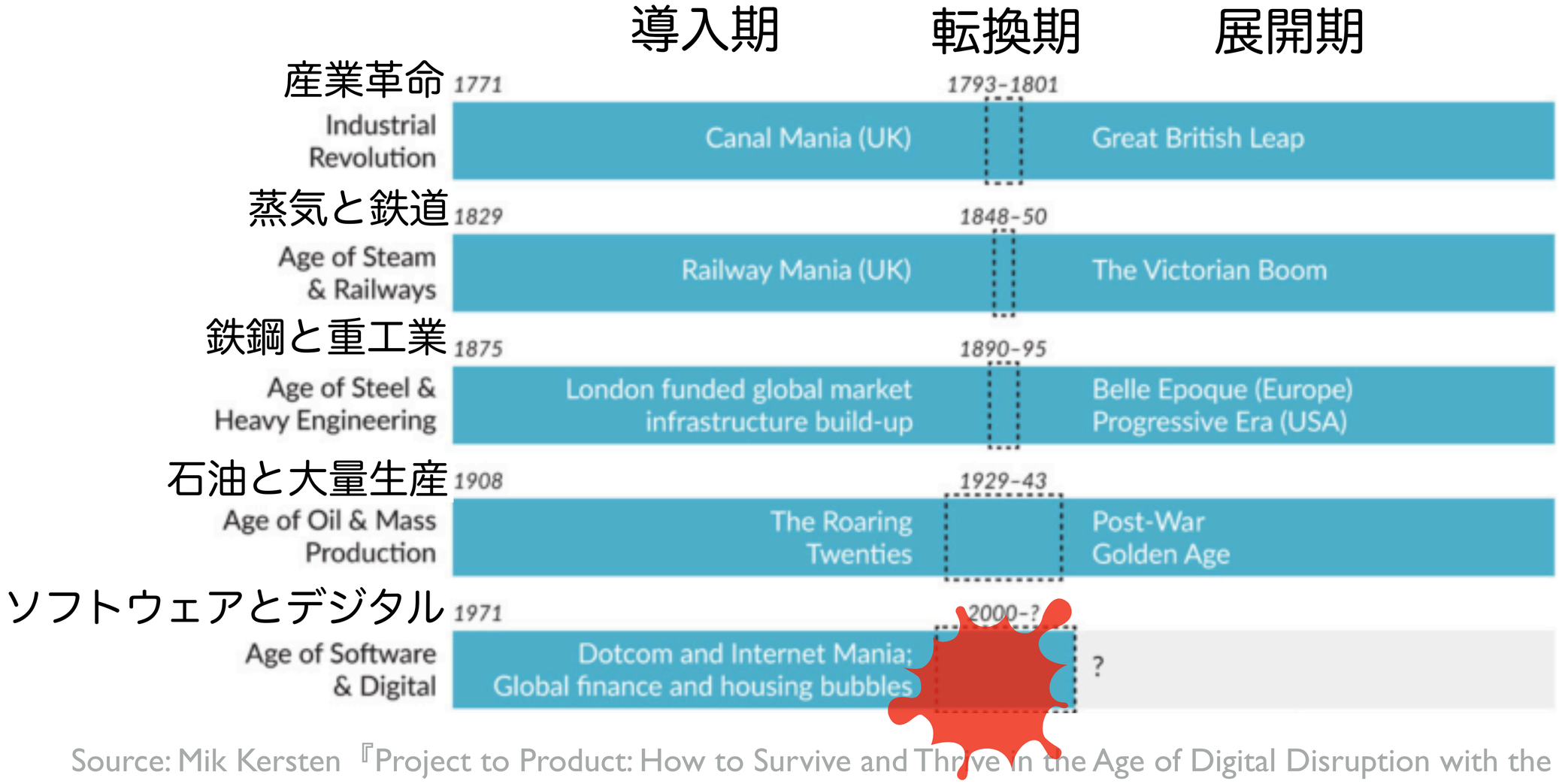
1. クラフトマンシップ
2. 倫理と基準
3. 規律
4. シンプルな設計
5. SOLID原則
6. クリーンアーキテクチャ

1. クラフトマンシップ

ソフトウェアが世界を支配する時代（2011）



5つの技術革命と3つのフェーズ



Source: Mik Kersten 『Project to Product: How to Survive and Thrive in the Age of Digital Disruption with the Flow Framework』

アジャイルソフトウェア開発宣言（2001）

https://agilemanifesto.org/iso/ja/manifesto.html

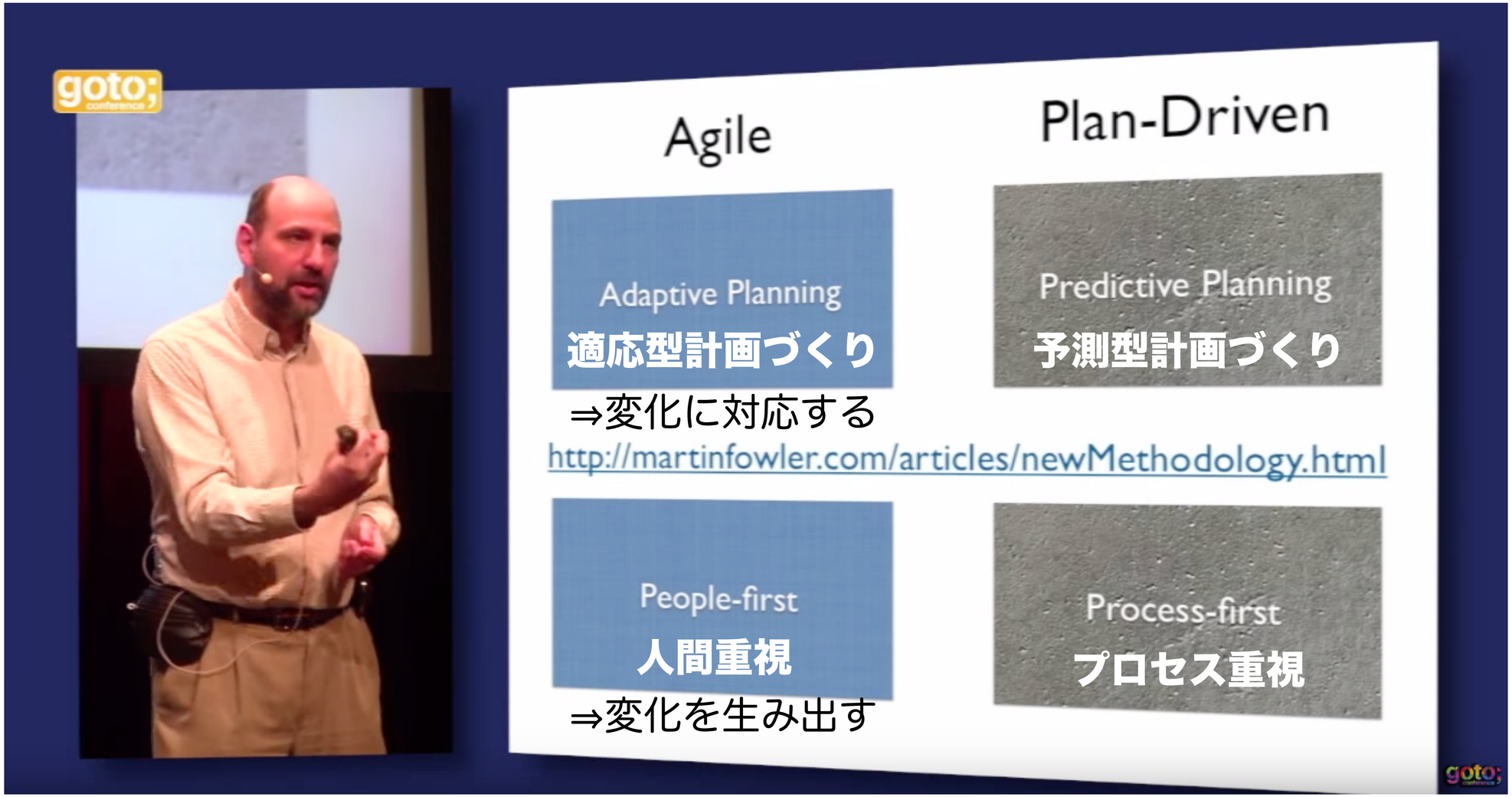
アジャイルのシンプルな定義

Agile is the ability to create and respond to change.

変化を生み出し、変化に対応する能力のこと

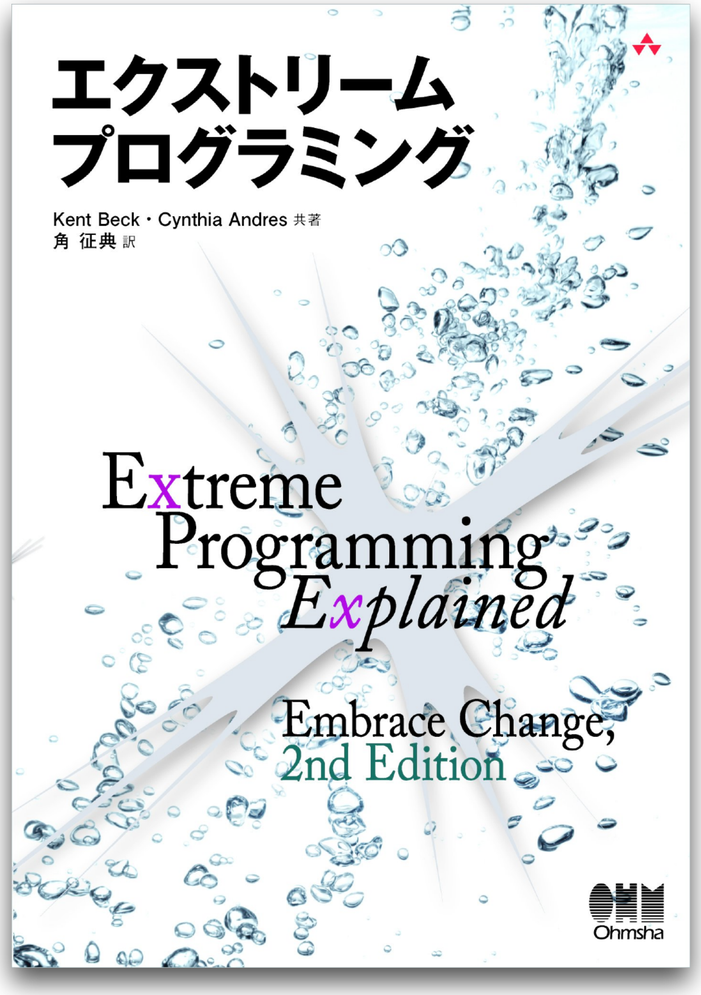
https://www.agilealliance.org/agile101/

アジャイルの特徴的な2つの能力

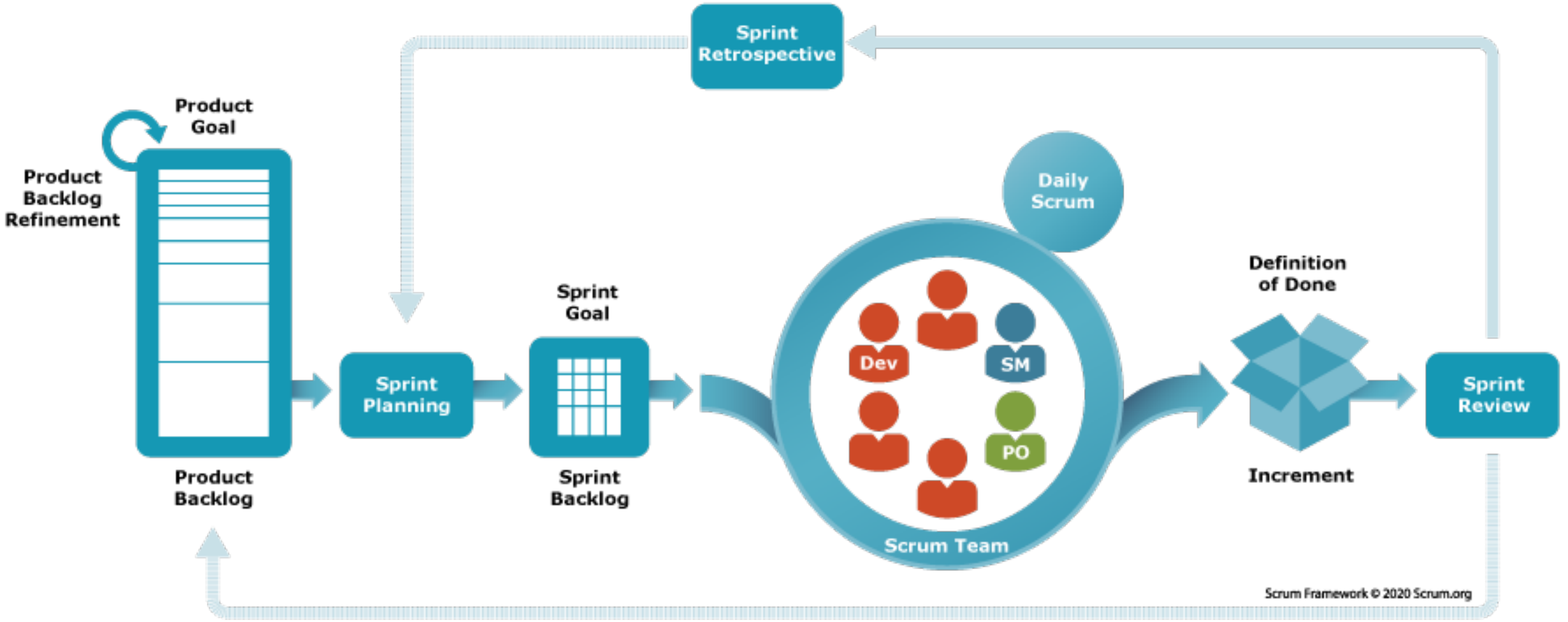


https://www.youtube.com/watch?v=TgdFA72crHM

当初はXPの時代


エクストリーム
プログラミング
Kent Beck・Cynthia Andres 共著
角 征典 訳
Extreme
Programming
Explained
Embrace Change,
2nd Edition
Ohmsha

次第にスクラムが台頭


Sprint Retrospective
Product Goal
Product Backlog Refinement
Product Backlog
Sprint Planning
Sprint Goal
Sprint Backlog
Daily Scrum
Dev
SM
PO
Scrum Team
Definition of Done
Increment
Sprint Review
Scrum Framework © 2020 Scrum.org

ヘロヘロスクラム（2009）

FlaccidScrum

29 January 2009

Martin Fowler

AGILE
AGILE ADOPTION
BAD THINGS

There's a mess I've heard about with quite a few projects recently. It works out like this:

- They want to use an agile process, and pick Scrum
- They adopt the Scrum practices, and maybe even the principles
- After a while progress is slow because the code base is a mess

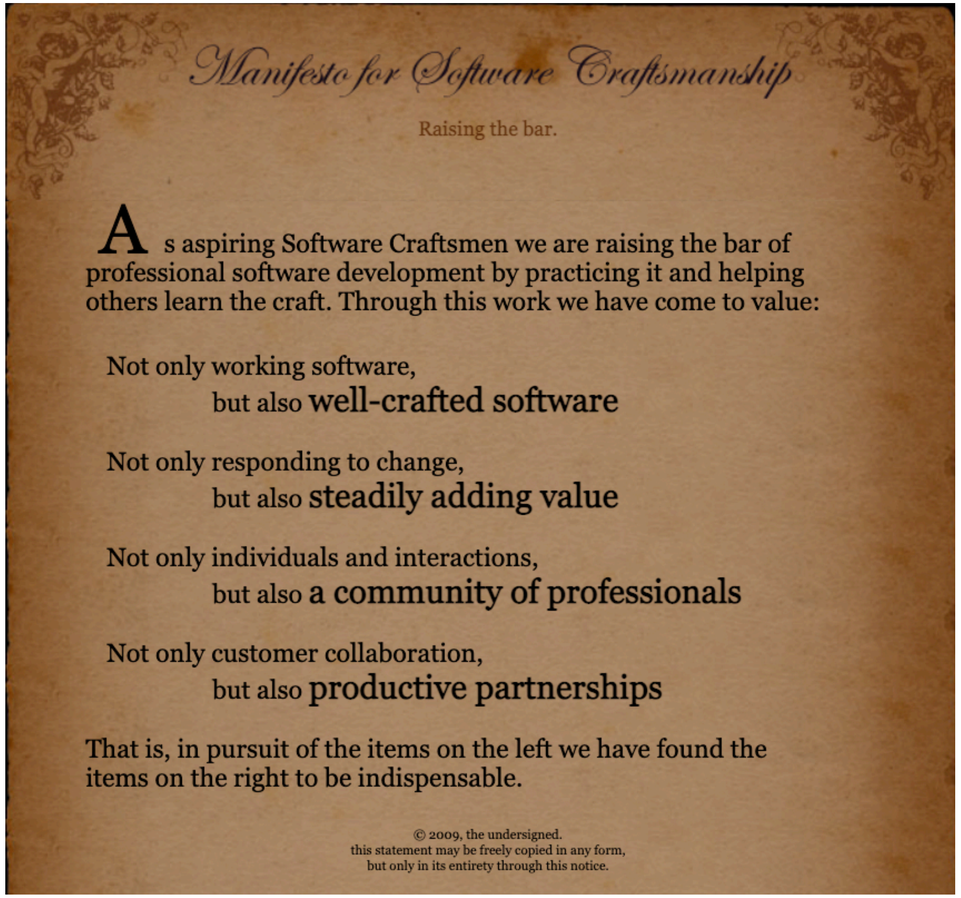
https://martinfowler.com/bliki/FlaccidScrum.html
日本語訳：https://bliki-ja.github.io/FlaccidScrum/

アジャイルの二日酔い 

- 多くの組織では、アジャイルとスクラムが同義語になっている。
- アジャイルコーチは、テクニカルプラクティスをコーチできるほどの技術スキルを持っていない。エンジニアリングについて話すこともほとんどない。
- アジャイルと開発者はお互いに離れようとしている。

『Clean Agile』 Bob C. Martin, Sandro Mancuso

ソフトウェアクラフトマンシップ宣言（2009）



Manifesto for Software Craftsmanship

Raising the bar.

As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

Not only working software,
but also **well-crafted software**

Not only responding to change,
but also **steadily adding value**

Not only individuals and interactions,
but also **a community of professionals**

Not only customer collaboration,
but also **productive partnerships**

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

私たちは意欲的なソフトウェアクラフトマンとして、ソフトウェアクラフトマンシップの実践あるいは専門技術の学習の手助けをする活動を通じて、プロとしてのソフトウェア開発の水準を引き上げようとしている。この活動を通して、私たちは以下の価値に至った。

動くソフトウェアだけでなく、精巧に作られたソフトウェアも
変化への対応だけでなく、着実な価値の付加も
個人との対話だけでなく、専門家のコミュニティも
顧客との協調だけでなく、生産的なパートナーシップも

すなわち、左記のことがらを追求するなかで、
右記のことがらも不可欠であることがわかった。

https://manifesto.softwarecraftsmanship.org/

クラフトマンシップの定義(?)

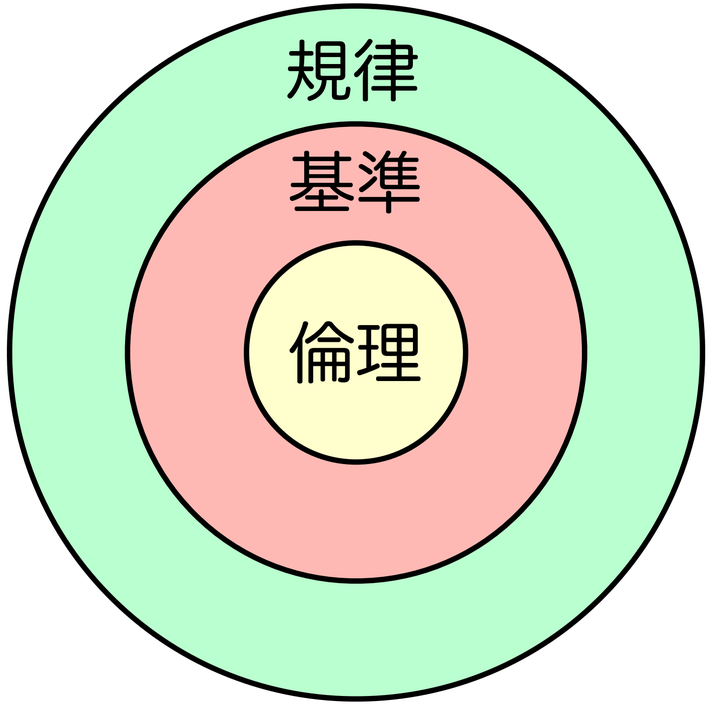
- 「クラフトマン」とは、特定の分野に関する高度なスキルを持ち、物事を成し遂げる人である。道具や業界に精通しており、仕事に誇りを持ち、仕事に対する尊厳とプロ意識を持って行動できると信頼されている人である。

『Clean Craftsmanship』 Bob C. Martin

クラフトマンシップの定義（その他）

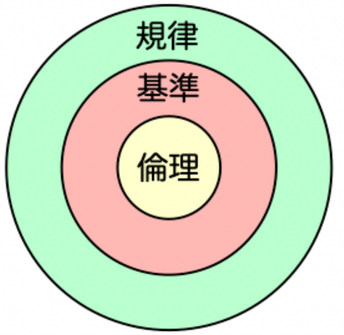
- ソフトウェア開発者のコーディングスキルを重視するアプローチ。開発者の説明責任よりも金銭的な問題を優先させるなど、ソフトウェア業界の悪弊に対するソフトウェア開発者の反応である。（Wikipedia）
- ソフトウェアクラフトマンシップは、熟練への長い旅です。ソフトウェア開発者が自らのキャリアに責任を持ち、常に新しいツールやテクニックを学び、自分を高めていくことを選択するマインドセットです。ソフトウェアクラフトマンシップは、ソフトウェア開発に責任、プロフェッショナリズム、プラグマティズム、そしてプライドを取り戻すものです。（Sandro Mancuso『The Software Craftsman』）

クラフトマンシップの3つの要素

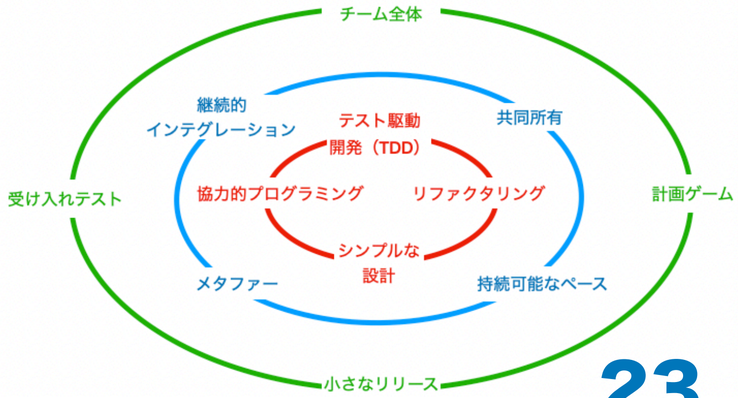


※同心円はClean Architectureのマネ（本には登場しない）

倫理、基準、規律



- 倫理
  - 3つのテーマ（無害、誠実、チームワーク）
- 基準
  - 品質、生産性、勇気（倫理の3つのテーマに対応）
- 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



2. 倫理と基準

医師の倫理「ヒポクラテスの誓い」

表1　ヒポクラテスの誓い

医神アポロン、アスクレピオス、ヒュギエイア、パナケイア、およびすべての男神・女神たちの御照覧をあおぎ、つぎの誓いと師弟契約書の履行を、私は自分の能力と判断の及ぶかぎり全うすることを誓います。

この術を私に授けていただいた先生に対するときは、両親に対すると同様にし、共同生活者となり、何かが必要であれば私のものを分け、また先生の子息たちは兄弟同様に扱い、彼らが学習することを望むならば、報酬も師弟契約書もとることなく教えます。また医師の心得、講義そのほかすべての学習事項を伝授する対象は、私の息子と、先生の息子と、医師の掟てに従い師弟誓約書を書き誓いを立てた門下生に限ることにし、彼ら以外の誰にも伝授はいたしません。

養生治療を施すに当たっては、能力と判断の及ぶ限り患者の利益になることを考え、危害を加えたり不正を行う目的で治療することはいたしません。

また求められても、致死薬を与えることはせず、そういう助言も致しません。同様に婦人に対し堕胎用のペッサリーを与えることもいたしません。私の生活と術ともに清浄かつ敬虔に守りとおします。

結石の患者に対しては、決して切開手術は行わず、それを専門の業とする人に任せます。

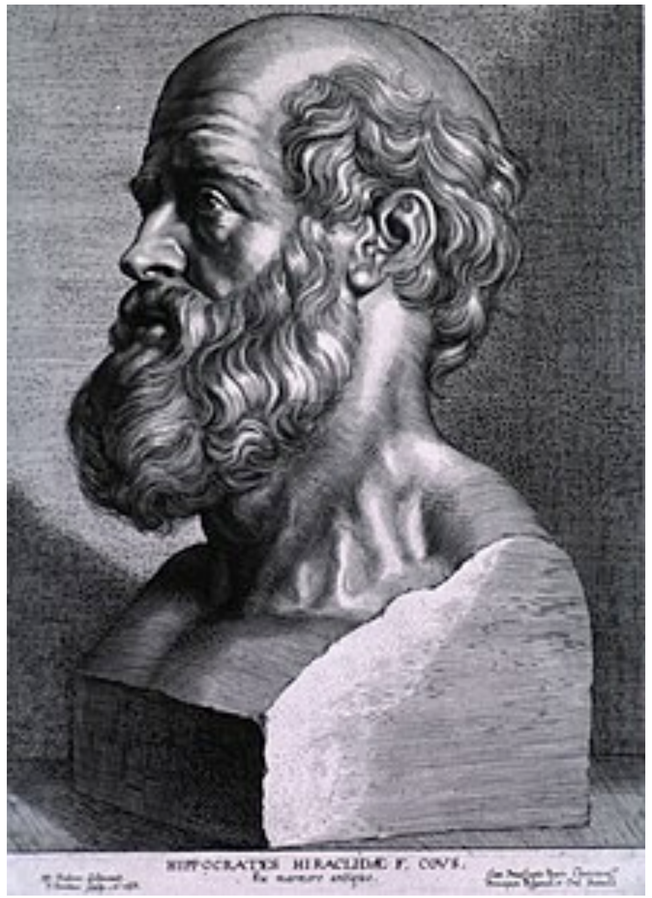
また、どの家にはいって行くにせよ、すべては患者の利益になることを考え、どんな意図的不正も害悪も加えません。とくに、男と女、自由人と奴隷のいかんをとわず、彼らの肉体に対して情欲をみたすことはいたしません。

治療の時、または治療しないときも、人々の生活に関して見聞きすることで、およそ口外すべきでないものは、それを秘密事項と考え、口を閉ざすことに致します。

以上の誓いを私が全うしこれを犯すことがないならば、すべての人々から永く名声を博し、生活と術のうえでの実りが得られますように。しかし誓いから道を踏み外し偽誓などをすることがあれば、逆の報いをうけますように。

（大槻マミ太郎訳：誓い．小川鼎三編、ヒポクラテス全集、第1巻、エンタプライズ、東京、1985;580-582より引用）

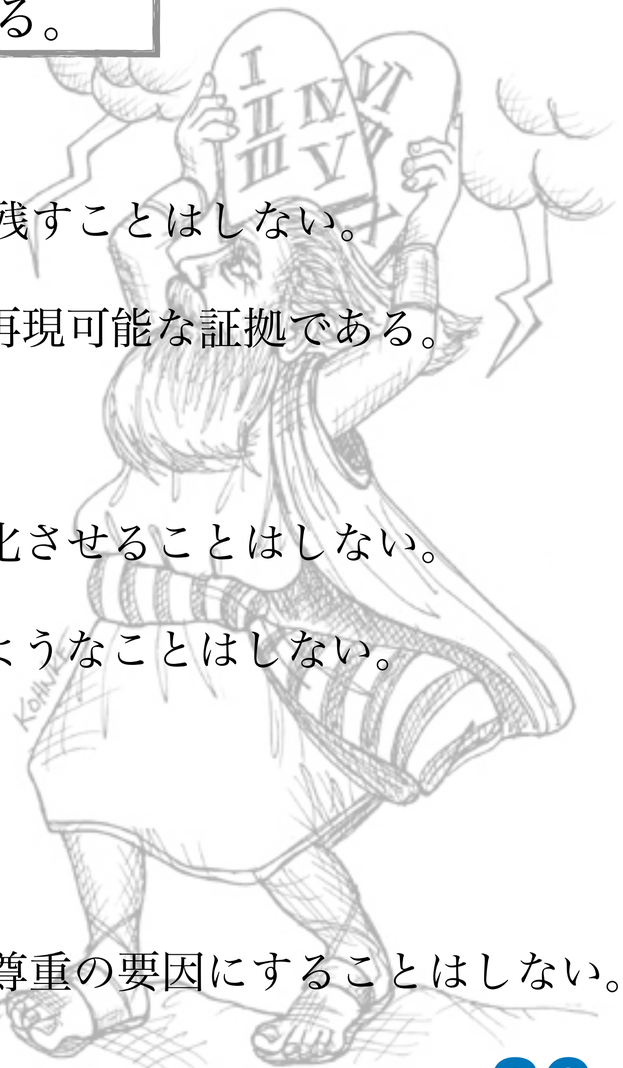
https://www.med.or.jp/doctor/rinri/i_rinri/a06.html



出典：Wikipedia（CC 表示-継承 4.0）

プログラマーの誓い

コンピュータープログラマーの職業の名誉を守り、維持するために、
私は自分の能力と判断の限りにおいて、以下のことを約束する。



1. 私は、有害なコードを作らない。
2. 私が作るコードは、常に私の最高傑作である。振る舞いや構造に欠陥のあるコードを故意に残すことはしない。
3. 私は、コードが正常に動作する証拠をリリースごとに用意する。それは、迅速で、確実で、再現可能な証拠である。
4. 私は、誰かの進捗を妨げないように、小さく何度もリリースする。
5. 私は、あらゆる機会において、恐れることなく執拗に私の作品を改善する。決して作品を劣化させることはしない。
6. 私は、私や誰かの生産性を高めるために、できる限りのことをする。決して生産性を落とすようなことはしない。
7. 私は、他の人が私をカバーできるように、私が他の人をカバーできるように努める。
8. 私は、規模と精度の両方を正直に見積もる。合理的な確実性がないときには約束をしない。
9. 私は、仲間のプログラマーの倫理、基準、規律、スキルを尊重する。その他の属性や特性を尊重の要因にすることはしない。
10. 私は、私の技術の学習と向上を怠らない。

プログラマーの誓い

コンピュータープログラマーの職業の名誉を守り、維持するために、
私は自分の能力と判断の限りにおいて、以下のことを約束する。

1. 私は、有害なコードを作らない。
2. 私が作るコードは、常に私の最高傑作である。振る舞いや構造に欠陥のあるコードを故意に残すことはしない。
3. 私は、コードが正常に動作する証拠をリリースごとに用意する。それは、迅速で、確実で、再現可能な証拠である。

無害／品質

4. 私は、誰かの進捗を妨げないように、小さく何度もリリースする。
5. 私は、あらゆる機会において、恐れることなく執拗に私の作品を改善する。決して作品を劣化させることはしない。
6. 私は、私や誰かの生産性を高めるために、できる限りのことをする。決して生産性を落とすようなことはしない。

誠実／生産性

7. 私は、他の人が私をカバーできるように、私が他の人をカバーできるように努める。
8. 私は、規模と精度の両方を正直に見積もる。合理的な確実性がないときには約束をしない。
9. 私は、仲間のプログラマーの倫理、基準、規律、スキルを尊重する。その他の属性や特性を尊重の要因にすることはしない。
10. 私は、私の技術の学習と向上を怠らない。

チームワーク／勇気

無害／品質 😈

ソフトウェアが社会に害を与える時代

「ミスター・ベイツvsポストオフィス」(2024)

英国史上最大規模の冤罪スキャンダルを描いた衝撃作

https://www.mystery.co.jp/programs/mr_bates_vs_the_post_office/

イギリス郵便局冤罪事件

- 2000年に運用開始した会計システム「ホライゾン」に欠陥
- 不正経理などの罪で郵便局長1000人以上が起訴される
  - 有罪判決：700人
  - 刑務所に収監：236人
  - 自殺者：4人（→ 13人）
- 賠償協議開始（2025年3月9日）、救済措置を勧告（2025年7月9日）

https://www.mystery.co.jp/column/k78ys6nr6j0f4hpj.html
https://jp.reuters.com/economy/QOIKQXZ36FNJDBFDEYX5JTU5MM-2025-07-09/
https://www.bloomberg.co.jp/news/articles/2025-07-09/SZ3UQ0T0G1KW00

フェニックス給与システム（カナダ）

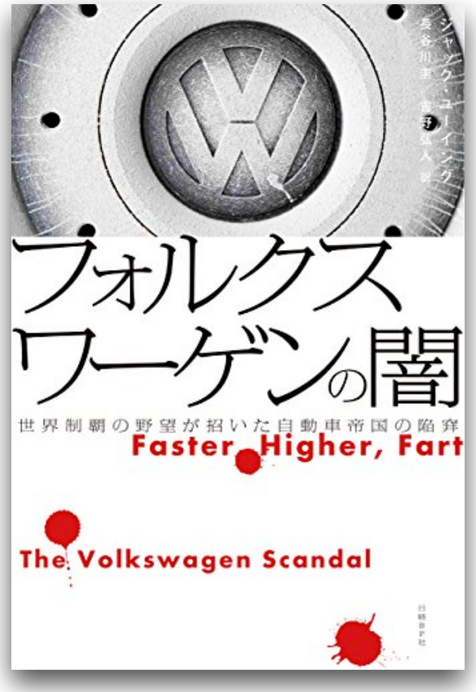
- 給与計算パッケージのカスタマイズ
- カナダ連邦政府職員43万人のうち、約70%が給与計算の誤りを経験
  - 2023-2024年度においても、3分の1が給与計算の誤りを経験
- カナダ政府は2026年6月までに未解決案件を処理すると約束
  - 残っている未解決案件は349,000件以上（2025年3月）
- 財政的損失は、これまでに51億カナダドル（約36億米ドル）に達している

https://spectrum.ieee.org/it-management-software-failures

フォルクスワーゲン排ガス不正問題

- 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/



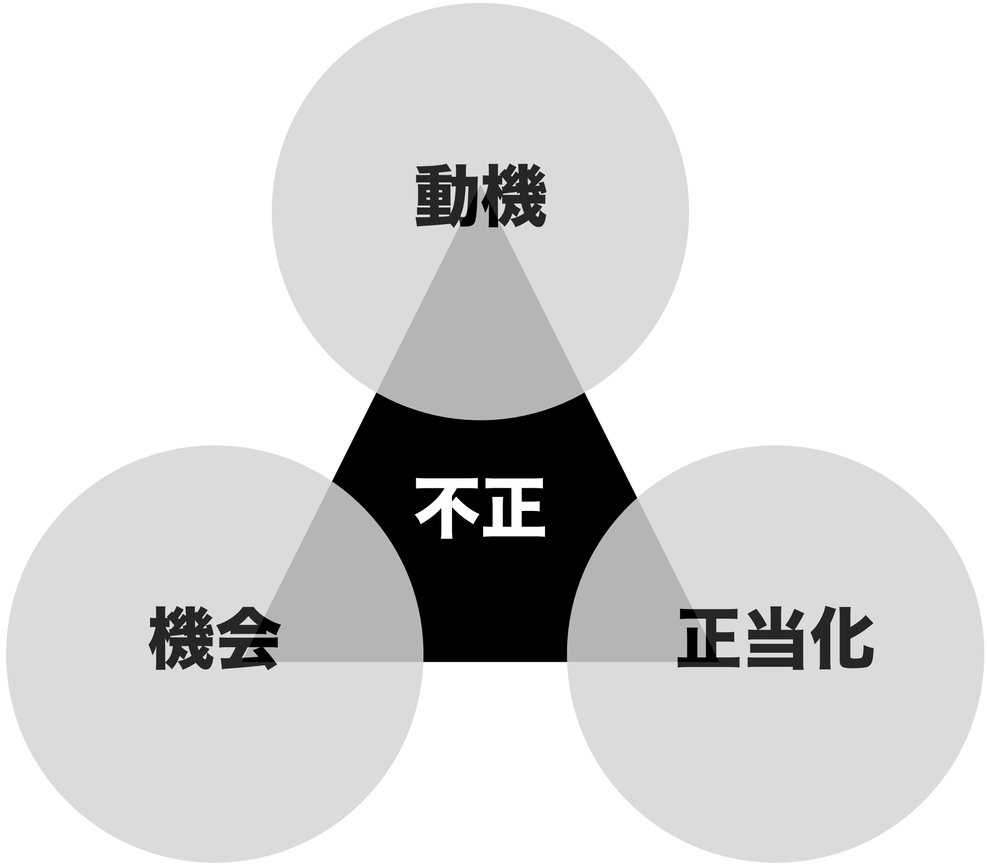
約束1：有害なコードを作らない

- フォルクスワーゲンのプログラマーが破ったのはこのルールだ。
- 彼らのソフトウェアは雇用主（フォルクスワーゲン）に利益をもたらしたかもしれない。だが、社会一般には害をもたらした。
- 我々プログラマーは、そのようなことを決してやってはならない。

『Clean Craftsmanship』 Bob C. Martin

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

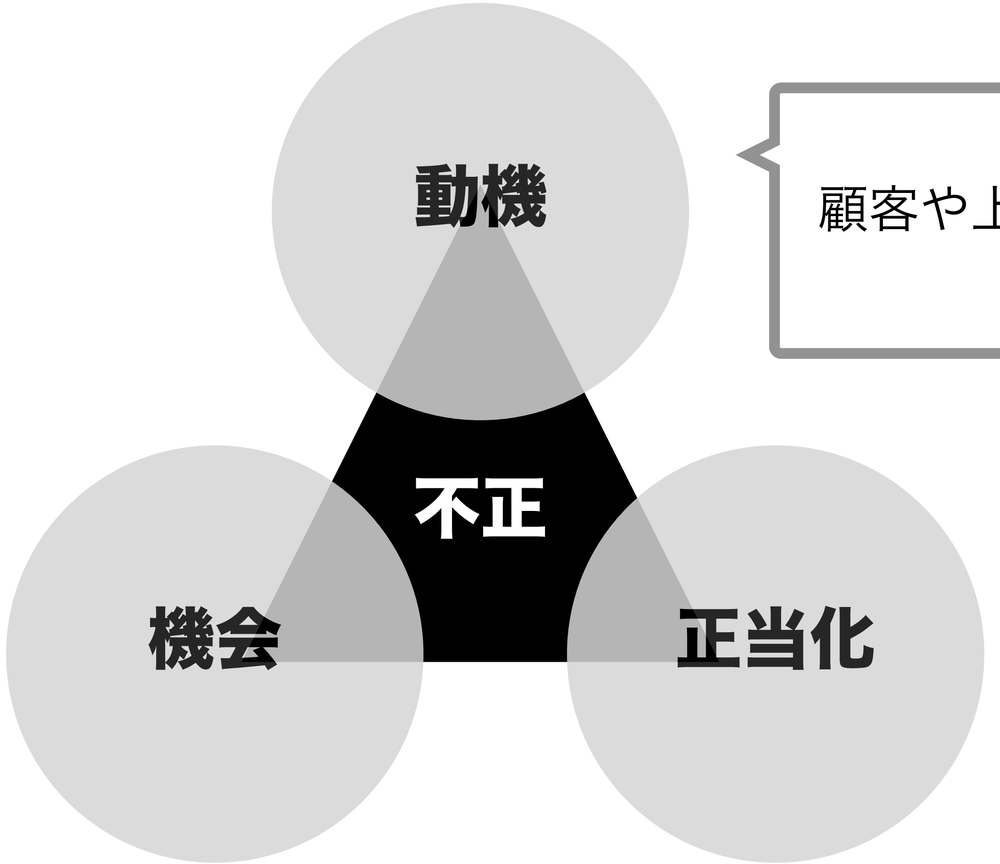


飯野 大介, 小松原 明哲, 2B1-2　不正のトライアングルによる組織的不正の評価について,
人間工学, 2019, 55 巻, Supplement 号, p. 2B1-2, 公開日 2019/07/31, Online ISSN 1884-2844, Print ISSN 0549-4974,

*: 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



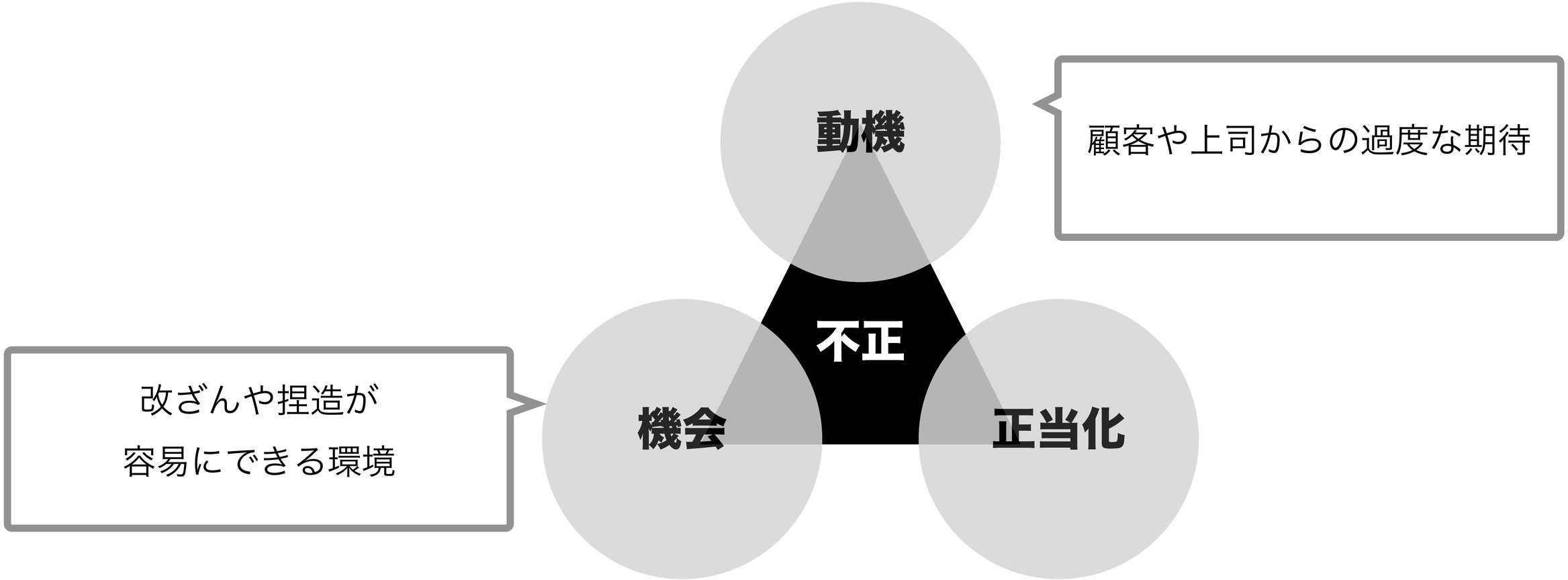
顧客や上司からの過度な期待

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

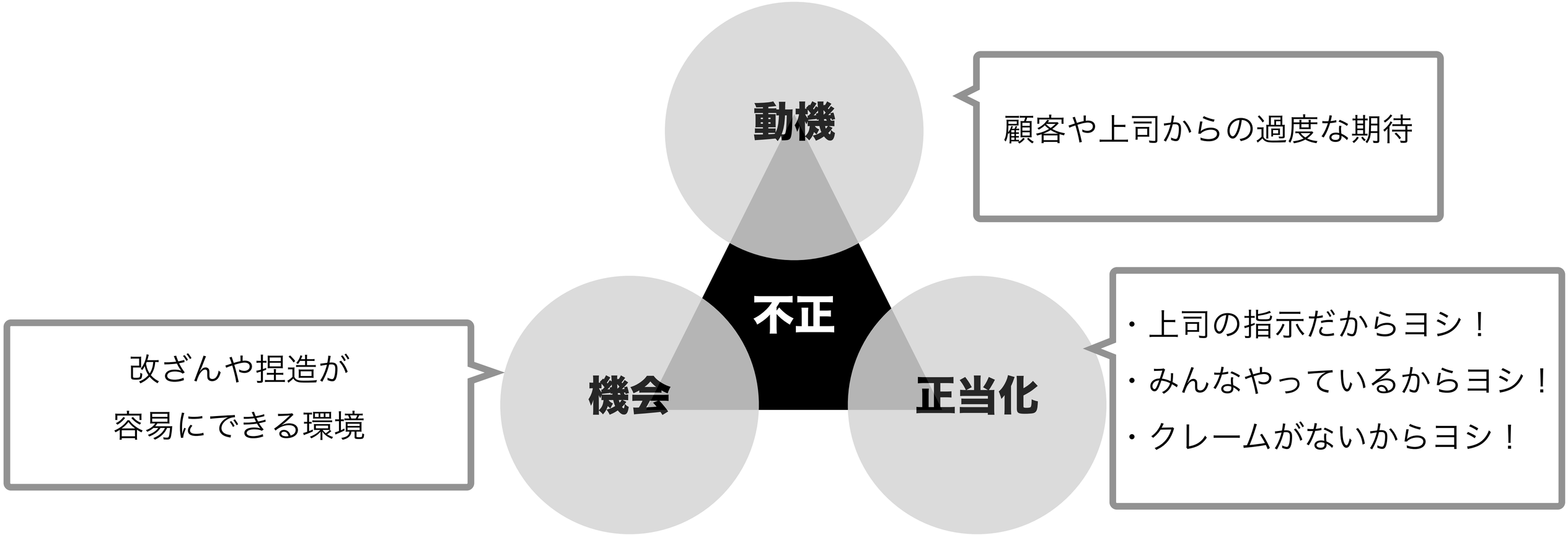


飯野 大介, 小松原 明哲, 2B1-2　不正のトライアングルによる組織的不正の評価について,
人間工学, 2019, 55 巻, Supplement 号, p. 2B1-2, 公開日 2019/07/31, Online ISSN 1884-2844, Print ISSN 0549-4974,

*: 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

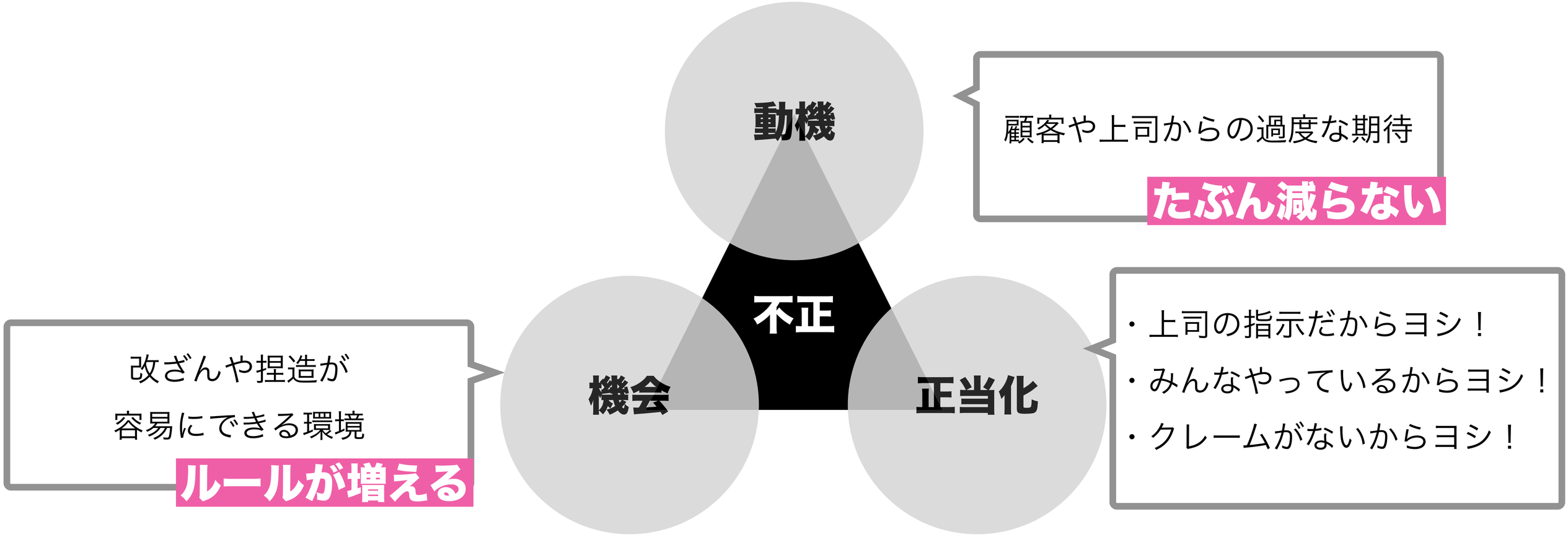


飯野 大介, 小松原 明哲, 2B1-2　不正のトライアングルによる組織的不正の評価について,
人間工学, 2019, 55 巻, Supplement 号, p. 2B1-2, 公開日 2019/07/31, Online ISSN 1884-2844, Print ISSN 0549-4974,

*: 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

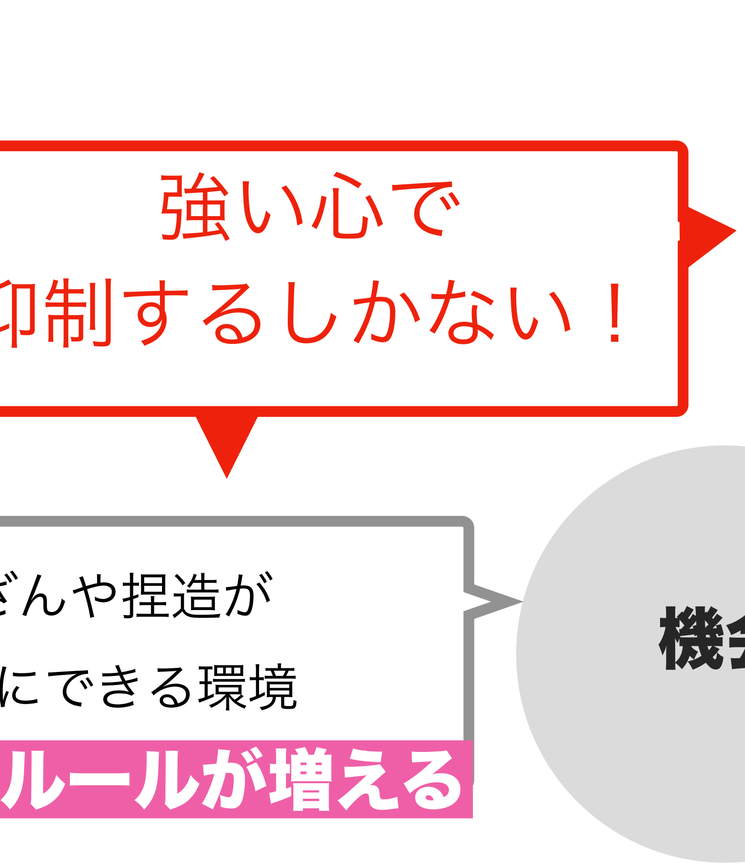


飯野 大介, 小松原 明哲, 2B1-2　不正のトライアングルによる組織的不正の評価について,
人間工学, 2019, 55 巻, Supplement 号, p. 2B1-2, 公開日 2019/07/31, Online ISSN 1884-2844, Print ISSN 0549-4974,

*: 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



強い心で
抑制するしかない！

顧客や上司からの過度な期待

たぶん減らない

改ざんや捏造が
容易にできる環境

ルールが増える

・上司の指示だからヨシ！
・みんなやっているからヨシ！
・クレームがないからヨシ！

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

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

ミラー・テストで判断する

- 倫理についての原則はただ一つだけである。

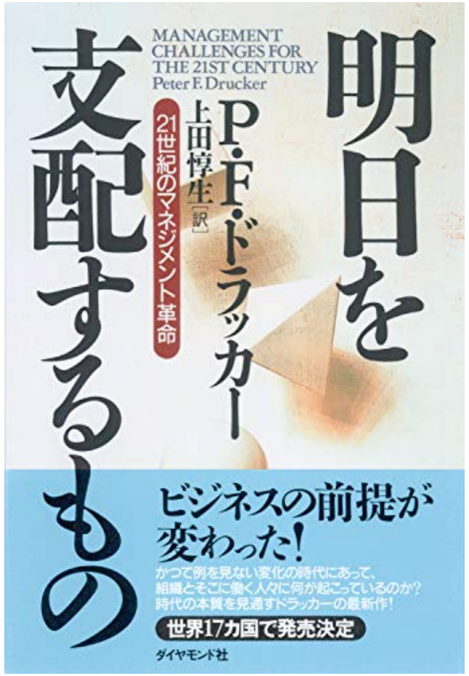
  判断の方法は簡単である。

  それをミラー・テストという。

  倫理の問題とは、

  朝、髭を剃るとき、あるいは口紅を塗るとき、

  どのような顔を見たいかというだけの問題である。



ミラー・テストで判断する

- 仕事から家に帰ったとき、鏡に映った自分を見て

  「今日はいい仕事をした」と言うだろうか？

  それとも、すぐにシャワーを浴びるだろうか？

約束2：欠陥のあるコードを残さない

- 意図しないバグを発生させない
- ただし、コードを動かすことだけがゴールではない
- コードの「構造」の品質にも責任を持つのがプログラマーである
  - ・構造を簡単に変更できてこそ「ソフト」ウェアである

『Clean Craftsmanship』 Bob C. Martin

約束3：コードが正常に動作する証拠を用意する

- 「わたしの環境では動きました」ではダメ
- 数学的に証明する必要はないが、実証的な証拠を残しておくべき
- つまり、テストコードを書こう!!
    - ・AIコーディングの時代はさらに重要になっている（らしい）

『Clean Craftsmanship』Bob C. Martin

誠実／生産性 😇

ボーイスカウト・ルール

それは、「来た時よりも美しく」です。たとえ自分が来た時にキャンプ場が汚くなっていたとしても、そしてたとえ汚したのが自分でなかったとしても、綺麗にしてからその場を去る、というルールです。（snip）「モジュールをチェックインする際には、必ずチェックアウト時よりも美しくする」となります。

https://プログラマが知るべき97のこと.com/エッセイ/ボーイスカウト-ルール/

約束4：小さく何度もリリースする

- ちょっとずつ変更（改善）できるようにする
  - 「来た時よりも美しく」
- 求められたときにすぐにリリースできるように環境を整えておく

『Clean Craftsmanship』 Bob C. Martin

約束5：執拗に作品を改善する

- プログラマーが作成するのはコードだけではない。設計、ドキュメント、スケジュール、計画なども作成する。これらも継続的に改善すべき「作品」だ。
- じゃあどうするのか？
- これも同じく、ちょっとずつ変更を加える
  - 「来た時よりも美しく」

『Clean Craftsmanship』 Bob C. Martin

約束6：生産性を高める

- 速く進みたければ、うまく進むしかない
  - 自動化しよう
  - 集中力を高めよう
  - 時間管理をうまくやろう（ポモドーロテクニックがおすすめ）
- 一気にうまくやるのは難しいので、ちょっとずつ変更を加える
  - 「来た時よりも美しく」（3回目）

『Clean Craftsmanship』Bob C. Martin

チームワーク／勇気 👥

約束7：他の人が私をカバーできるように、私が他の人をカバーできるように努める。

- チームで仕事をするなら、チーム全体に知識を広げよう。
- 知識を広げる最善の方法は、一緒に（ペアかモブで）働くことである。
- 車を運転しているときに他のドライバーを怒鳴りつけたことはないだろうか？これは「フロントガラス効果」と呼ばれる。フロントガラスを挟むと、他人のことをバカ、マヌケ、敵だと見なすようになる。
  - フロントガラス効果を回避するには、1年に数回は物理的な部屋に集まる。

『Clean Craftsmanship』Bob C. Martin

約束8：正直に見積もる

- 「とりあえずやってみます」には気を付けよう
  - 「とりあえず」がそのまま使われてしまうぞ！
  - 無理なものは最初から無理と言ったほうが誠実
    - 無理なときは代替案を提示できると良い
- 見積りは確率分布（幅）なので、修正版PERT法を使ってみてはどうか？
  - 平均値：(2 × 最有力 + (最良 + 最悪) / 2) / 3
  - シグマ：(最良 - 最悪) / 6

約束9：仲間を尊重する

- そのまま！
- 人を属性で判断しない（性別、人種、宗教など）
- 能力や規律を持たない人が入ってきても歓迎してあげよう
  - むしろ、そういう人たちを育てる責任が私たちにはある
  - クラフトマンシップを継承していく必要がある

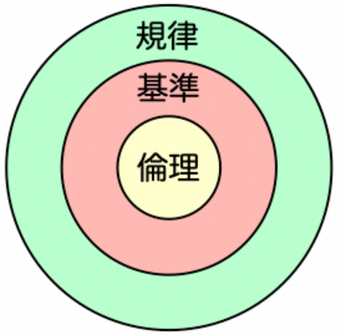
『Clean Craftsmanship』Bob C. Martin

約束10：技術の学習と向上を怠らない

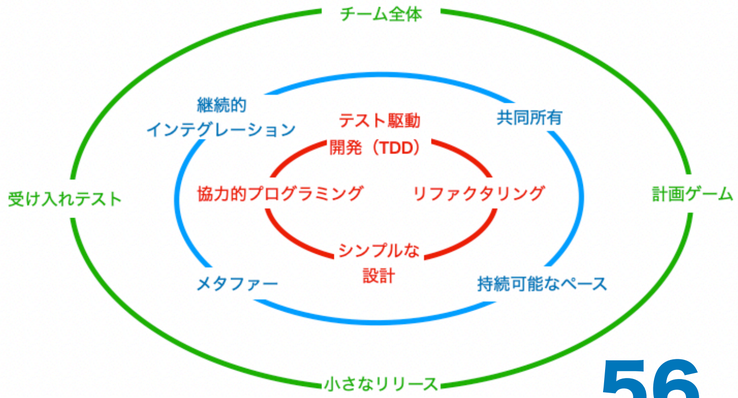
- プログラマーは学習をやめてはならない。学ぶべき領域は事実上無限だ。我々の業界は過去何十年かけて急速に変化している。これからもその変化はしばらく続くだろう。あなたはそれに追いつく必要がある。
- 書籍やブログを読み続けよう。ビデオを観続けよう。カンファレンスやユーザーグループに参加し続けよう。研修に行き続けよう。学習を続けよう。
- 過去の偉大な作品に注目しよう。1960年代、1970年代、1980年代に書かれた書籍は、素晴らしい洞察と情報があふれている。

『Clean Craftsmanship』 Bob C. Martin

倫理、基準、規律



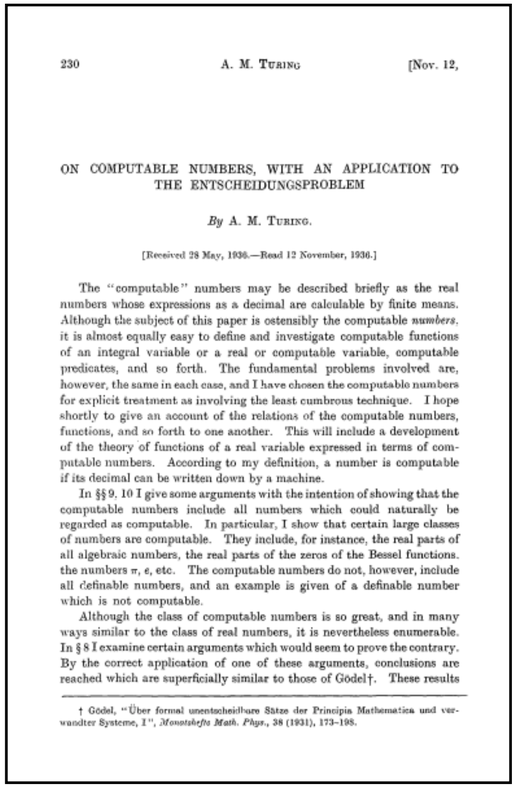
- 倫理
  - 3つのテーマ（無害、誠実、チームワーク）
- 基準
  - 品質、生産性、勇気（倫理の3つのテーマに対応）
- 規律
  - テスト駆動開発、リファクタリング、シンプルな設計、協力的プログラミング（ペア + モブ）、受け入れテスト



3. 規律

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.]

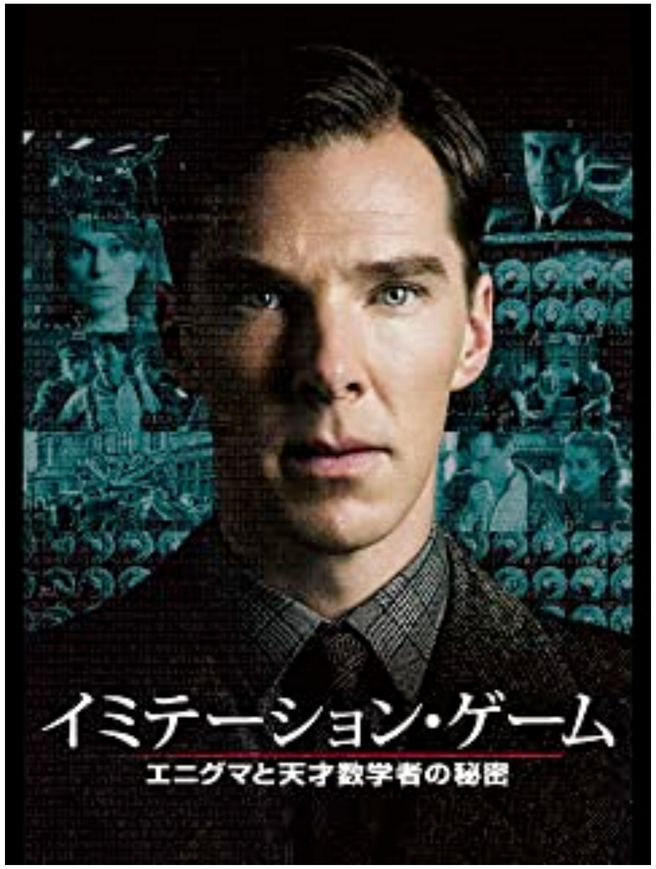
The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable 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.

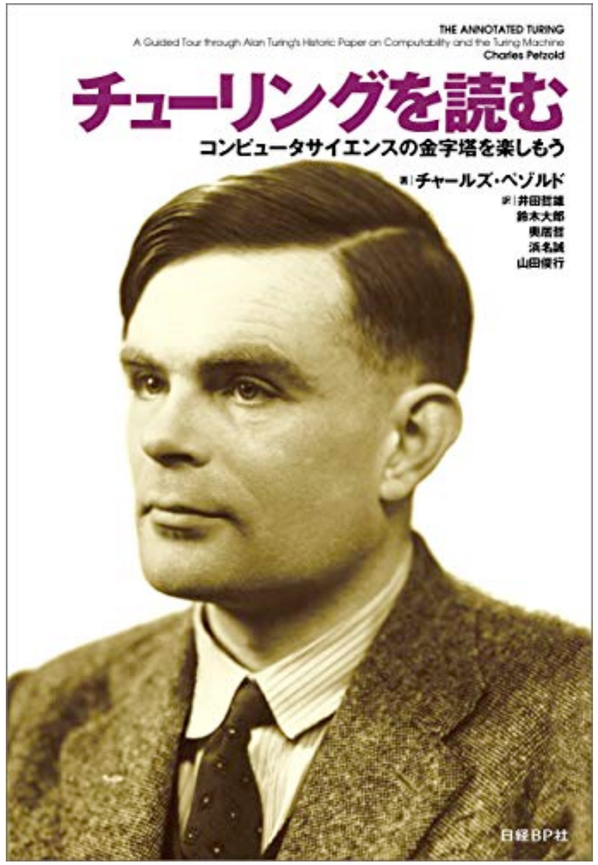
In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, 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)



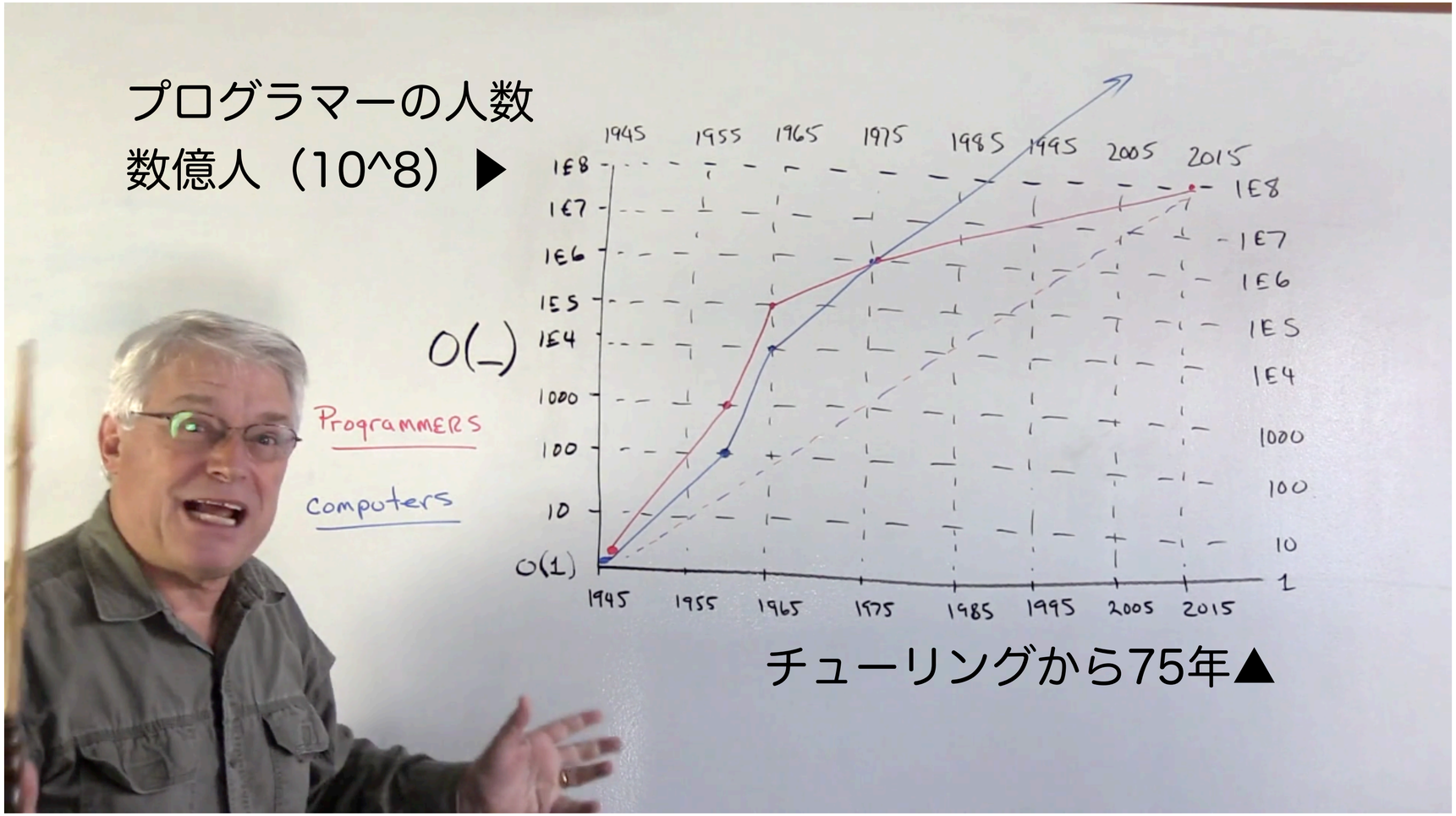


これからは（問題を）計算可能な形にする
能力のある数学者がかなりの人数必要になるはずだ。
困難となるのは、我々が何をしているのかを
見失わないために、適切な**規律**を維持することである。

―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



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

▼

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

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

SYSTEM FAILURE

その後、またビジョンが変わった。1999年の映画『マトリックス』では、主人公はプログラマーであり、世界の「救世主」だった。実際、彼の神々しいパワーは「コード」を読み解く能力に由来するものだった。 『Clean Craftsmanship』 Bob C. Martin

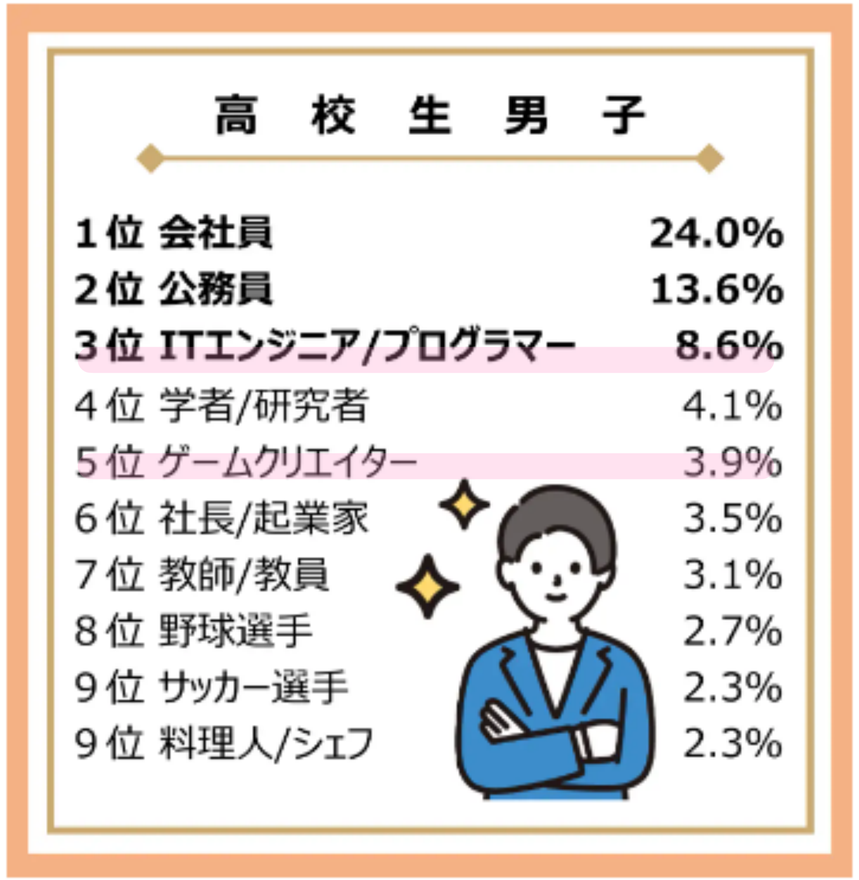
世界で最も影響力のある100人（2013）

Notch and Jeb (Minecraft)

https://time100.time.com/2013/04/18/time-100/slide/markus-persson-and-jens-bergensten/

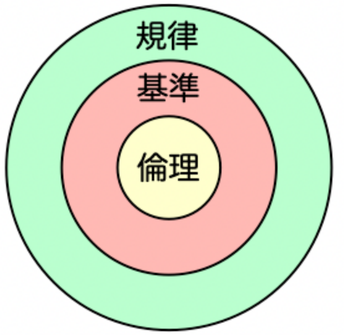
補足：他にも リーナス（2003）、パンジー創業者（2005）、ヴィタリック・ブテリン（2021）など

高校生の「大人になったらなりたいもの」（2025）

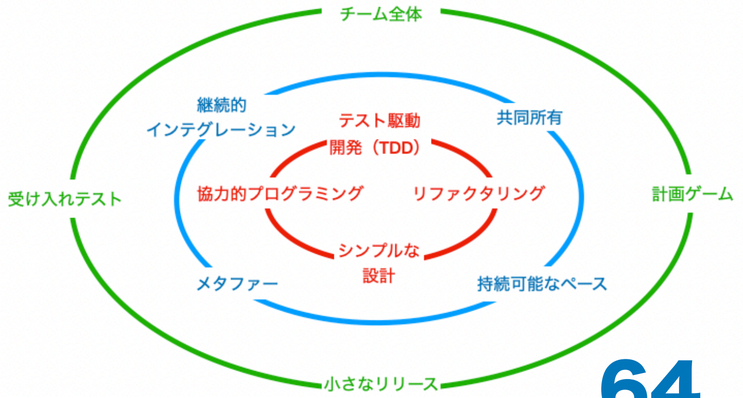


出典：第一生命／第36回「大人になったらなりたいもの」調査結果

倫理、基準、規律

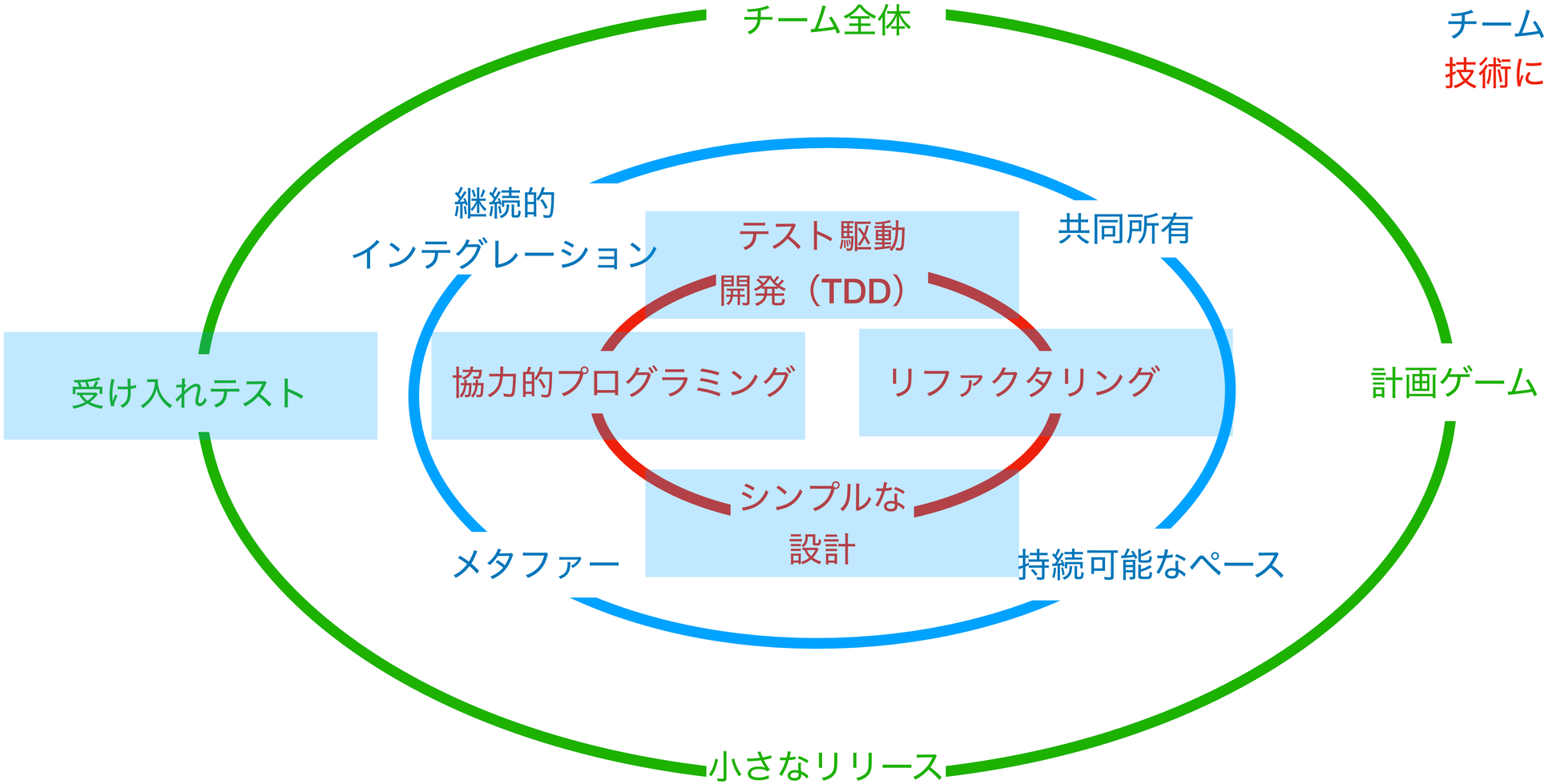


- 倫理
  - 3つのテーマ（無害、誠実、チームワーク）
- 基準
  - 品質、生産性、勇気（倫理の3つのテーマに対応）
- 規律
  - テスト駆動開発、リファクタリング、シンプルな設計、協力的プログラミング（ペア + モブ）、受け入れテスト



プログラマーの「規律」はXPの手法

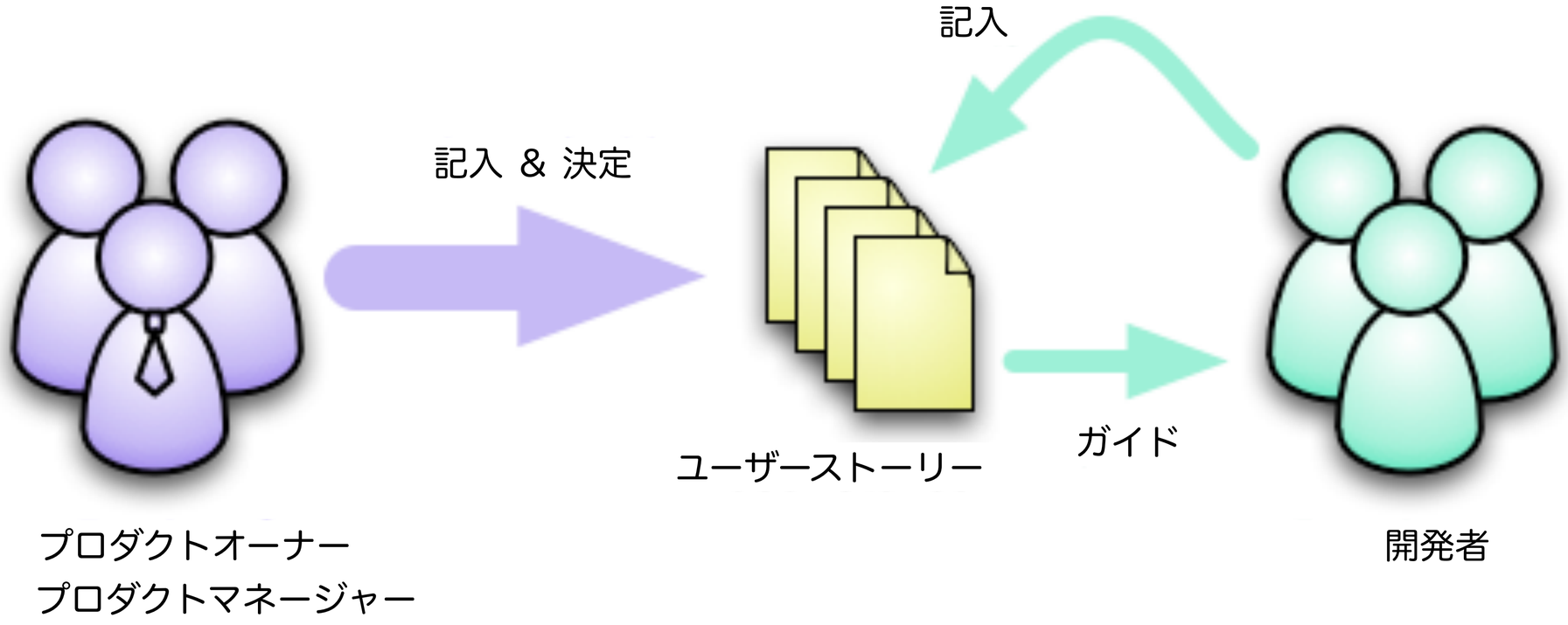
ビジネスに関すること
チームに関すること
技術に関すること



Bob Martin『Clean Agile』図1.8 サークルオブライフ

受け入れテスト

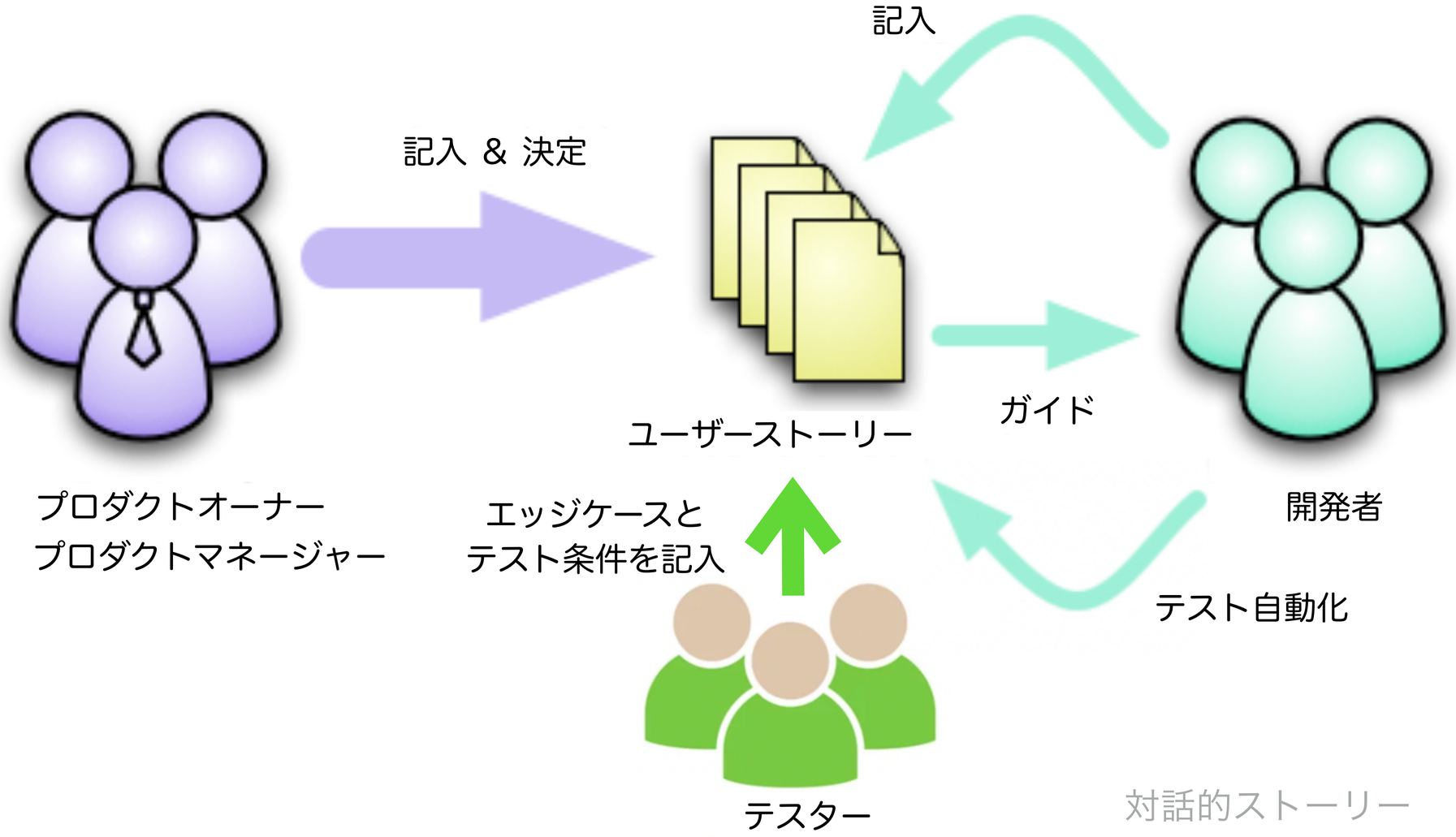
ストーリーを中心に置く



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

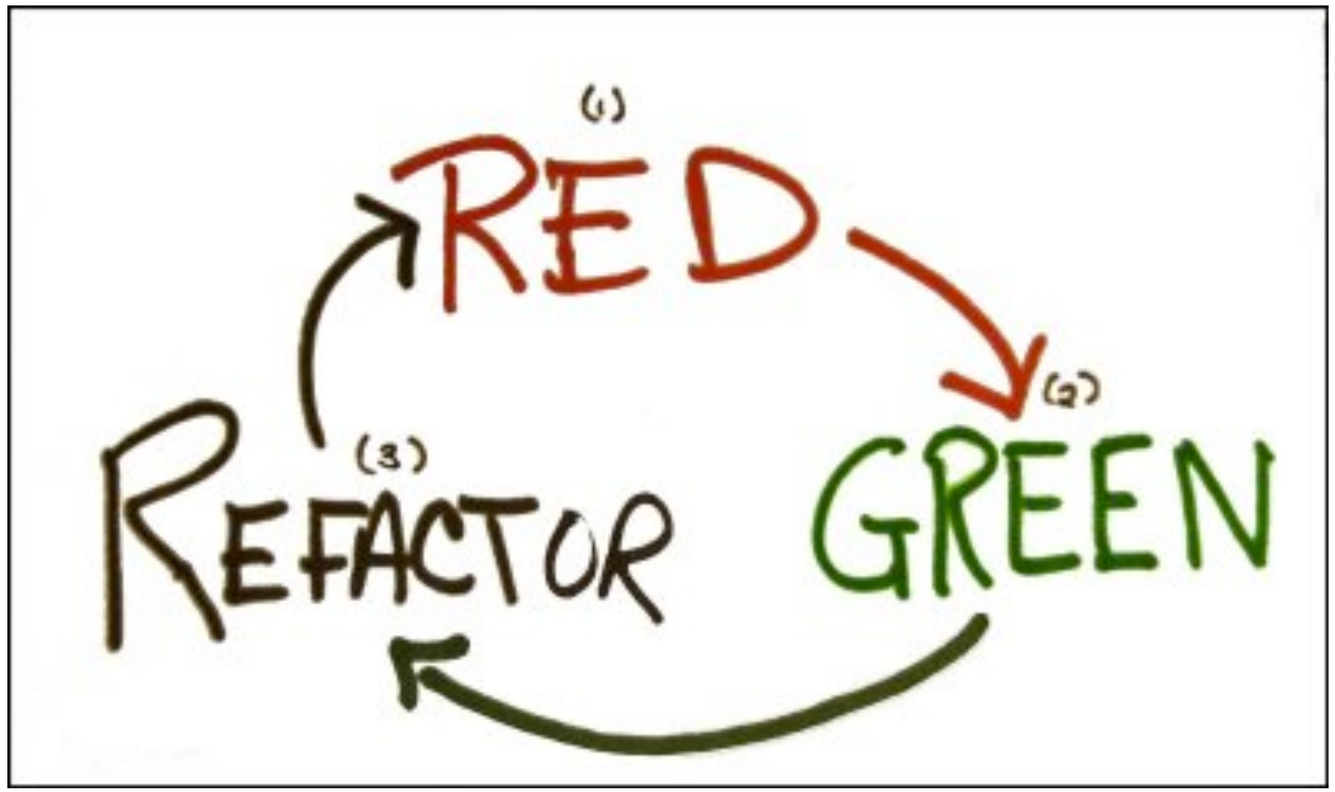
その後、モブプログラミングに引き継がれていく

『Clean Coder』 Bob C. Martin

TDD + リファクタリング

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

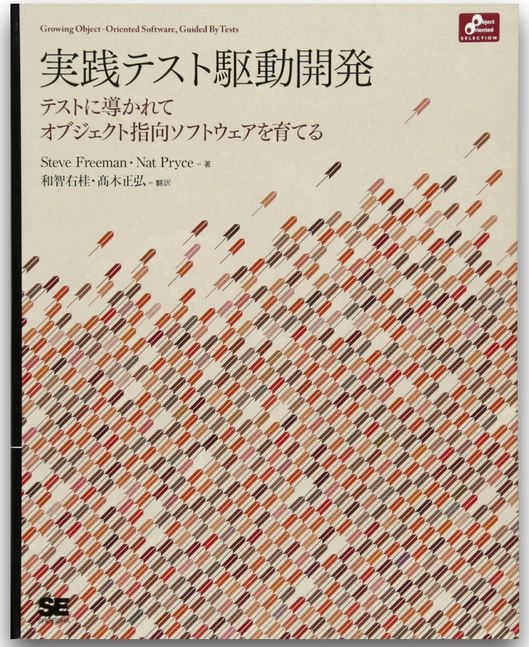
https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html

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』



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）
- 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』

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

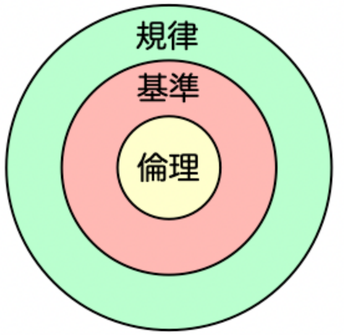
参考：野中郁次郎・竹内弘高『ワイズカンパニー』

日々の活動（習慣）が倫理を育む

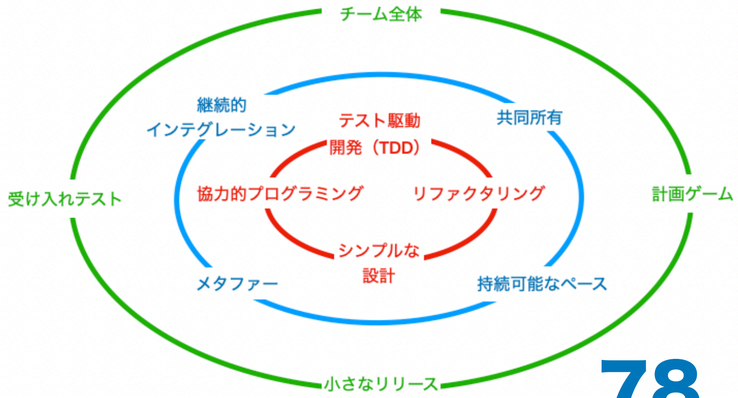
- "上手に家を建てることから人はすぐれた建築家になり、
  下手に建てることから劣悪な建築家になる"
- "要するに一言でいえば、同じような活動の反復から、
  人の性格の状態が生まれるのである"

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

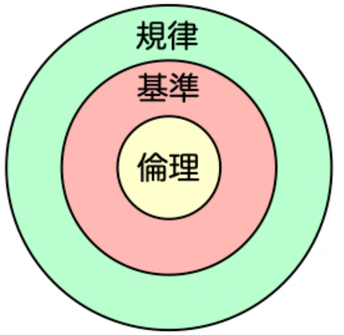
倫理、基準、規律



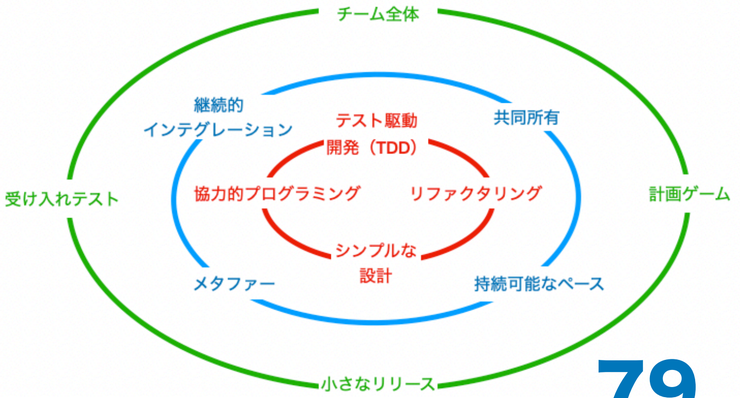
- 倫理
  - 3つのテーマ（無害、誠実、チームワーク）
- 基準
  - 品質、生産性、勇気（倫理の3つのテーマに対応）
- 規律
  - テスト駆動開発、リファクタリング、シンプルな設計、協力的プログラミング（ペア + モブ）、受け入れテスト



倫理、基準、規律



- 倫理
  - ・3つのテーマ（無害、誠実、チームワーク）
- 基準
  - ・品質、生産性、勇気（倫理の3つのテーマに対応）
- 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



4. シンプルな設計

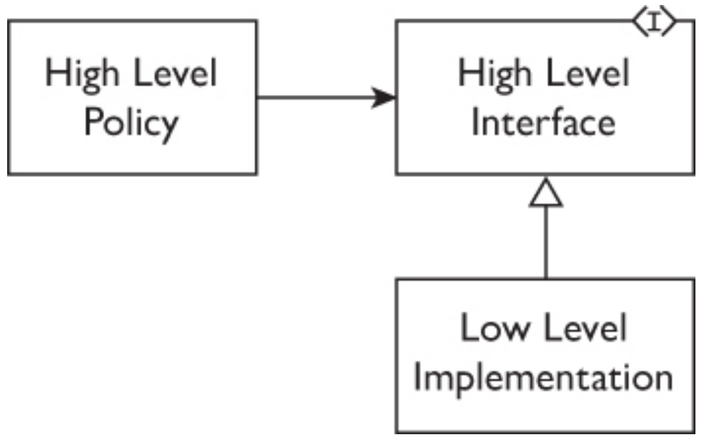
シンプルな設計 第6章

Bob C. Martin『Clean Craftsmanship』

シンプルとは？

シンプルとは？

- シンプル：Simplex ↔ Complex（複雑、複合）
  - 複雑に混ぜ合わさっている状態ではないこと
  - （ソフトウェアでは）方針と詳細が混ざっていないこと
- 解決策となるのは「抽象化」による分離（詳細から方針に依存する）

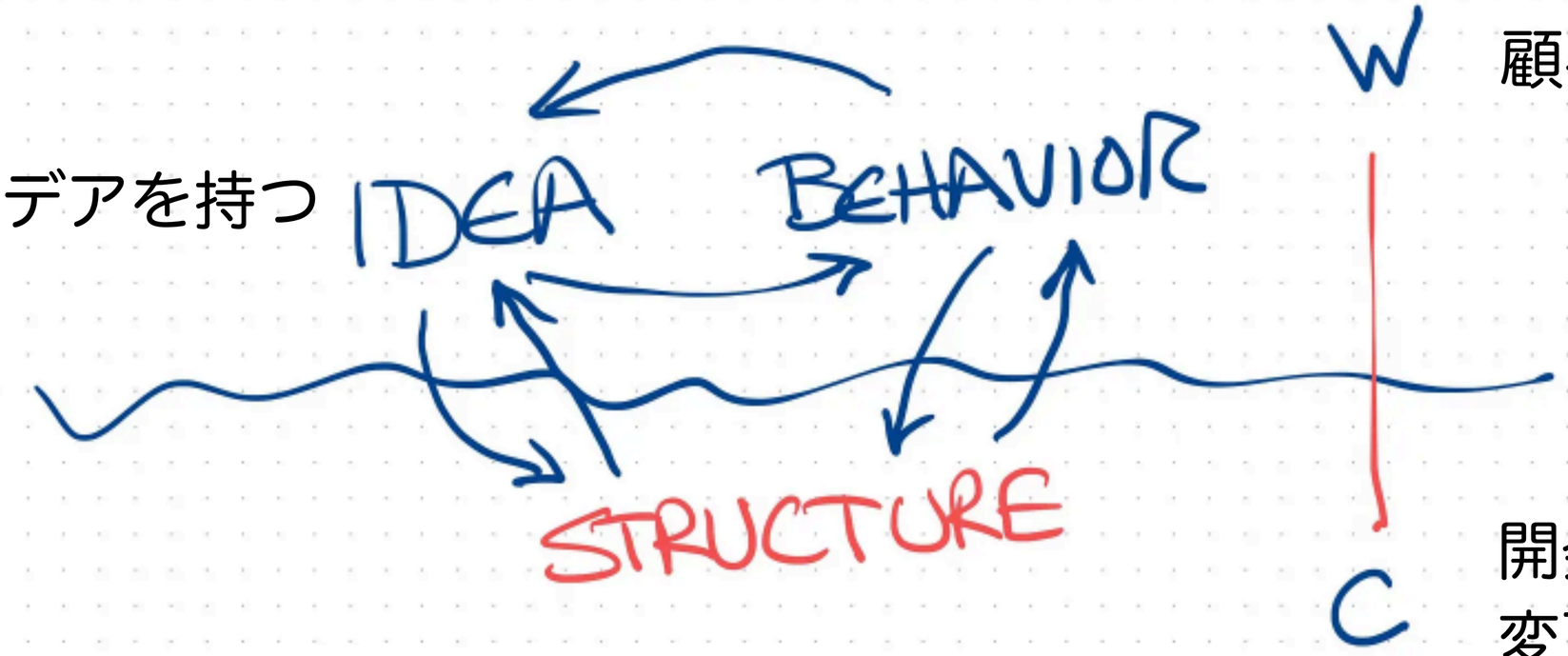


- "Simple Made Easy" - Rich Hickey (2011)
- Clean Code: A Handbook of Agile Software Craftsmanship, 2nd Edition

ソフトウェアの設計とは？

WとCの人間関係

顧客は振る舞いを期待/理解する



顧客がアイデアを持つ

顧客は待つ人

開発者は
変更する人

ソフトウェアの構造を理解/実装できるのは開発者だけ

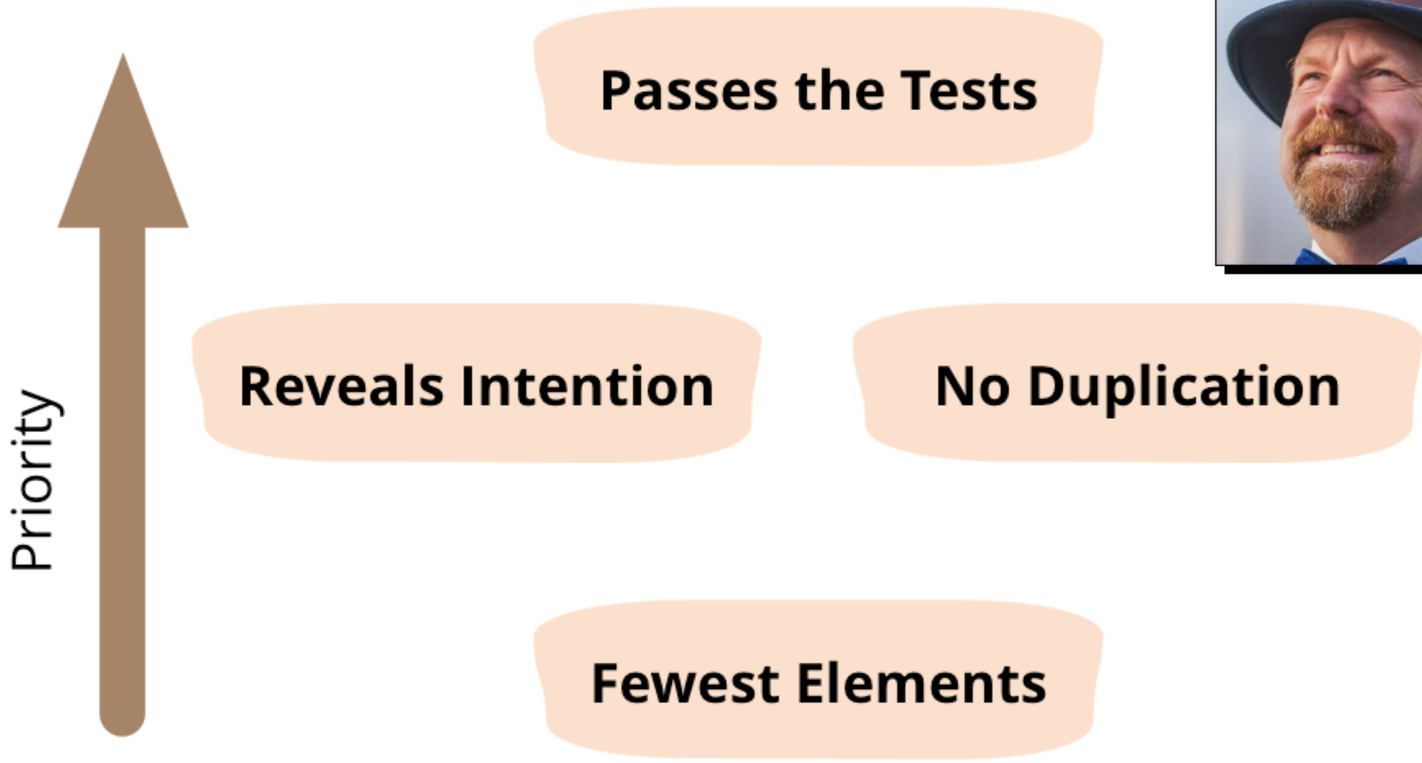
https://tidyfirst.substack.com/p/behavior-change-revenue-versus-structure

設計：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
- ただし「構造」の状態によって振る舞いの追加コストは変化する
  - 「いやあ、先月までなら簡単に追加できたんですけどね...💦」
    - これを避けたい！……が、「構造」を変更しても顧客は喜ばない
- 振る舞いと構造、両方やらないといけないのがつらいところ 😬
  - じゃあ、どうすればいいのか？

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



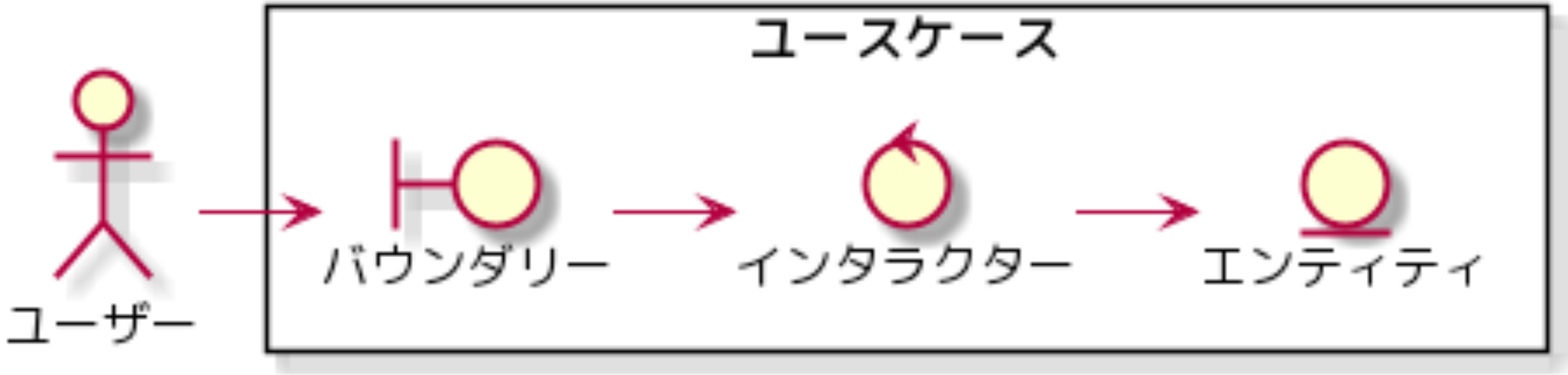
https://bliki-ja.github.io/BeckDesignRules/

1. テストをパスさせる

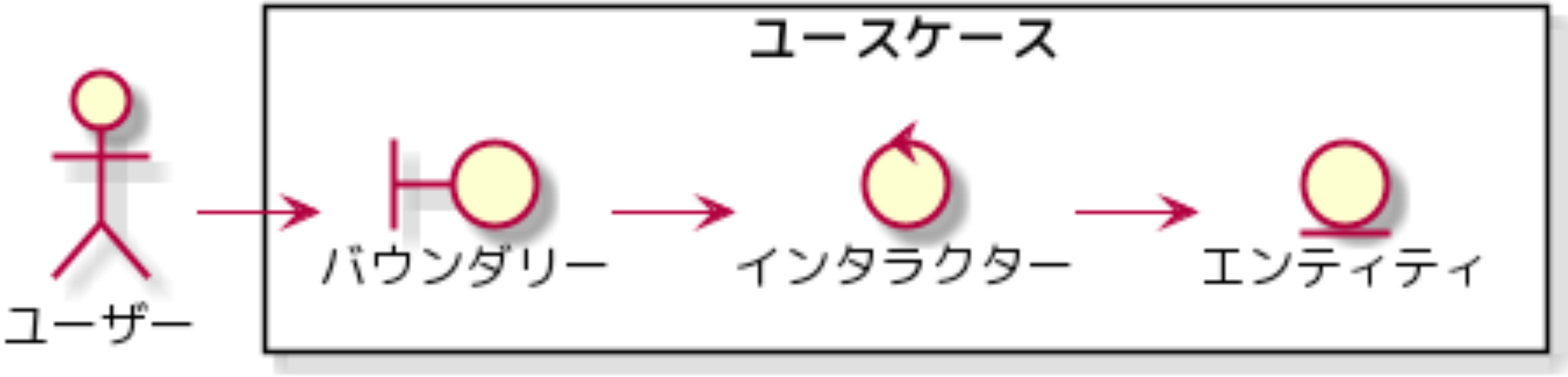
- テストのカバレッジを高めやすいのが良い設計という話
  - TDDとはまた違う（テストファーストや自動化を意味しない）
- 顧客にとって大事なのはユースケース（振る舞い）

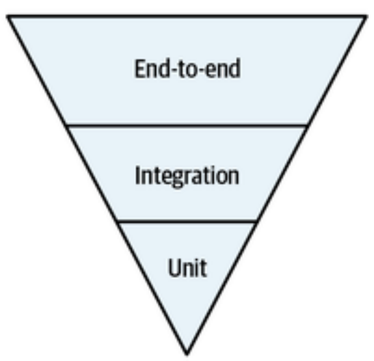
大事なのは振る舞い（ユースケース）

ユースケースを分割する（BCE）

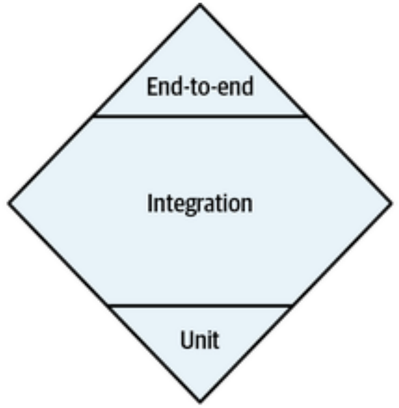


ビジネスロジックの場所とテスト戦略

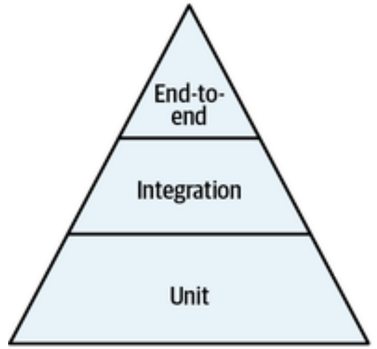




Reversed testing pyramid



Testing diamond



Testing pyramid

1. テストをパスさせる

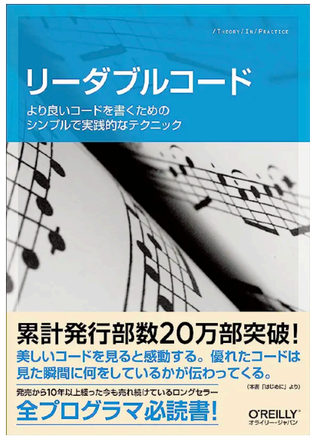
- テストのカバレッジを高めやすいのが良い設計という話
  - TDDとはまた違う（テストファーストや自動化を意味しない）
- 顧客にとって大事なのはユースケース（振る舞い）
- テストが必要な部分はユースケースの奥側にまとめる
  - ドメイン駆動開発をやりたい理由はこのあたり
  - そのほうがテストが楽になる → カバレッジを上げやすい

2. 意図を明らかにする

- ソフトウェアの設計は人間関係なので...
  - 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）



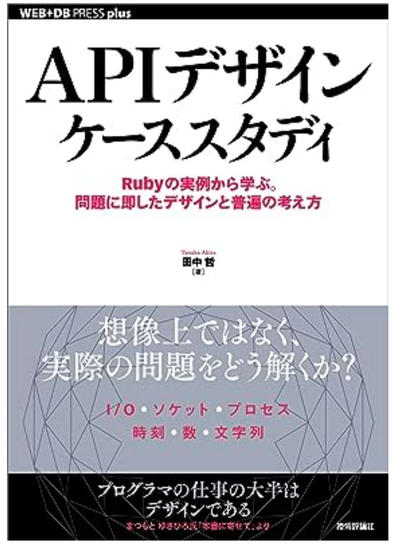
*同僚*のために意図を明らかにする

- 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

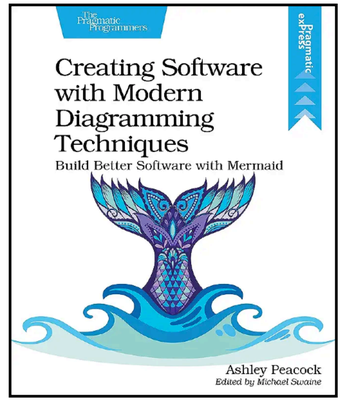
*同僚*のために意図を明らかにする

- 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をみんなで読む
  - UNIX文化に触れる



*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
  - 顧客にとって重要なのは構造ではなく振る舞い
  - 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>

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

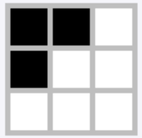
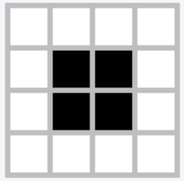
- DRY原則「すべての知識はシステム内において、単一、かつ明確な、そして信頼できる表現になっていなければならない」（『達人プログラマー』）
  - コードをコピペすんなという話ではない
  - DRY原則はコード以外にも適用される（p. 40）
    - ドキュメント、データ構造、APIのバインディング
      - 知識をそれぞれの表現に合わせてうまく落とし込む

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
  }
}
```

...ルールの記述をコードに反映する

```
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
  - 意図を明確に伝えたいなら、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
  - 意図を明確に伝えたいなら、重複や余計な要素があっても構わない
- そもそも「柔軟性を確保したい」は未来を予測しようとしているので危険
  - 未来は予測できるのか……？

未来を予測すると悪い設計になる

- 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』

未来を予測すると悪い設計になる

- 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』

未来を予測すると悪い設計になる

- 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

シンプルな設計（変更が簡単な設計）
↔ 未来を予測した設計（変更が難しい設計）

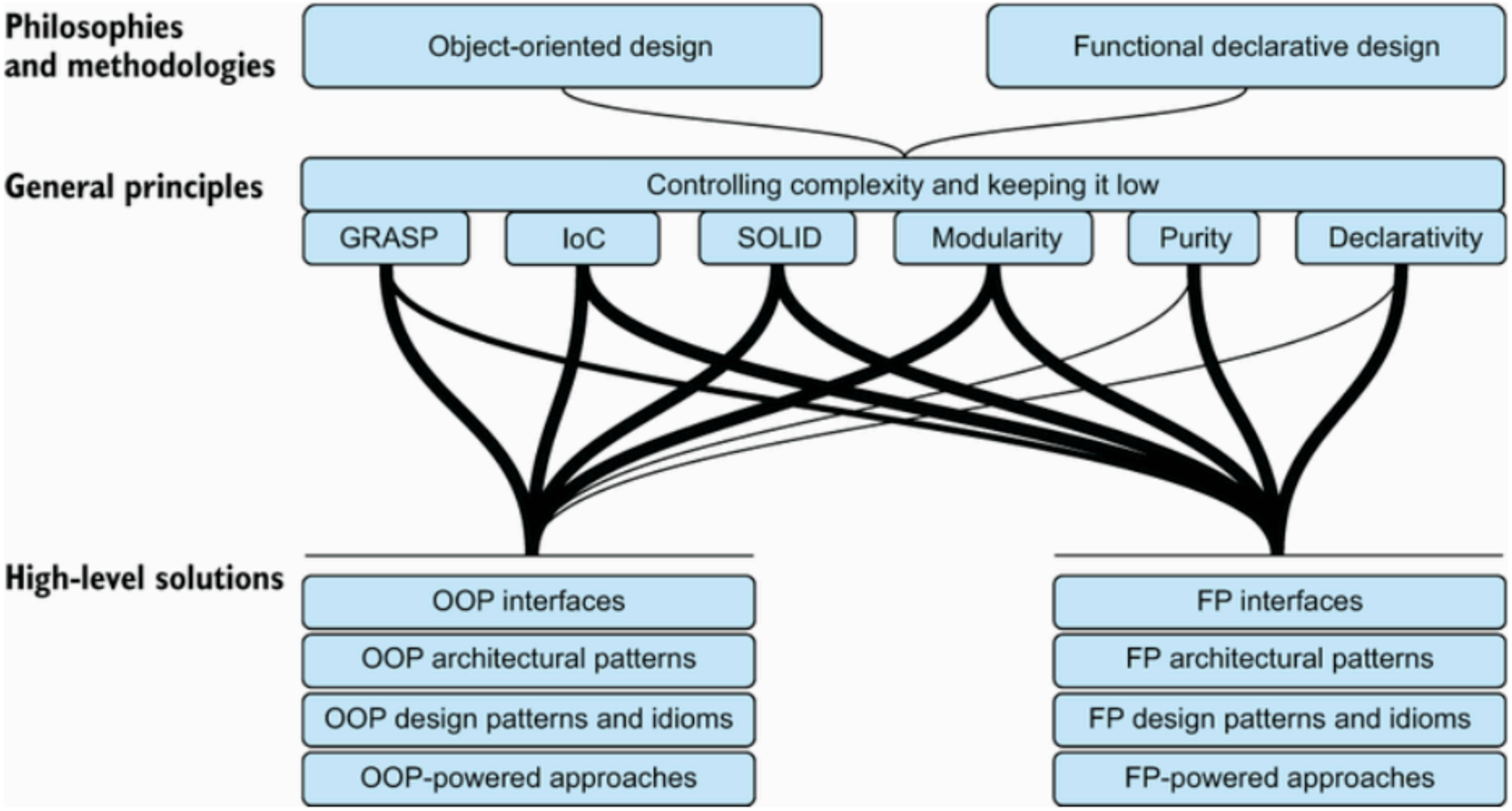
『Understanding the Four Rules of Simple Design』

5. SOLID原則

SOLID原則

- SOLID: アンクル・ボブが収集および考案した設計原則
  - ★SRP：単一責任の原則
  - ★OCP：オープン・クローズドの原則
  - LSP：リスコフの置換原則
  - ISP：インターフェイス分離の原則
  - ★DIP：依存関係逆転の原則
- オブジェクト指向の原則と思われていたが、実は普遍的な原則として使える

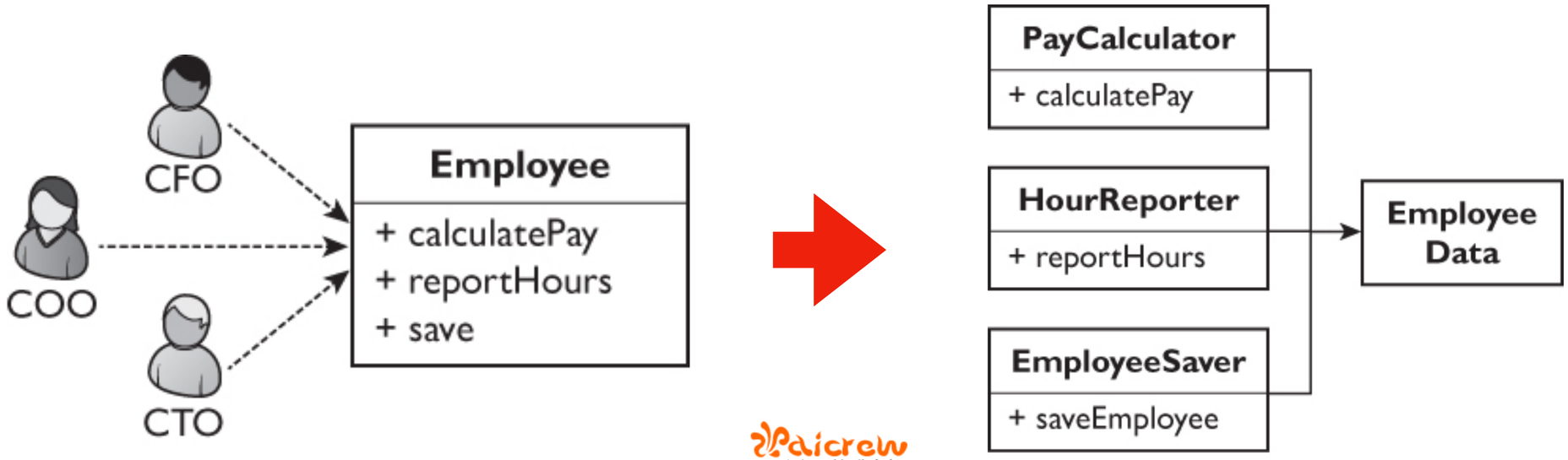
複雑性を制御するさまざまな手法



Source: Alexander Granin『Functional Design and Architecture』Manning

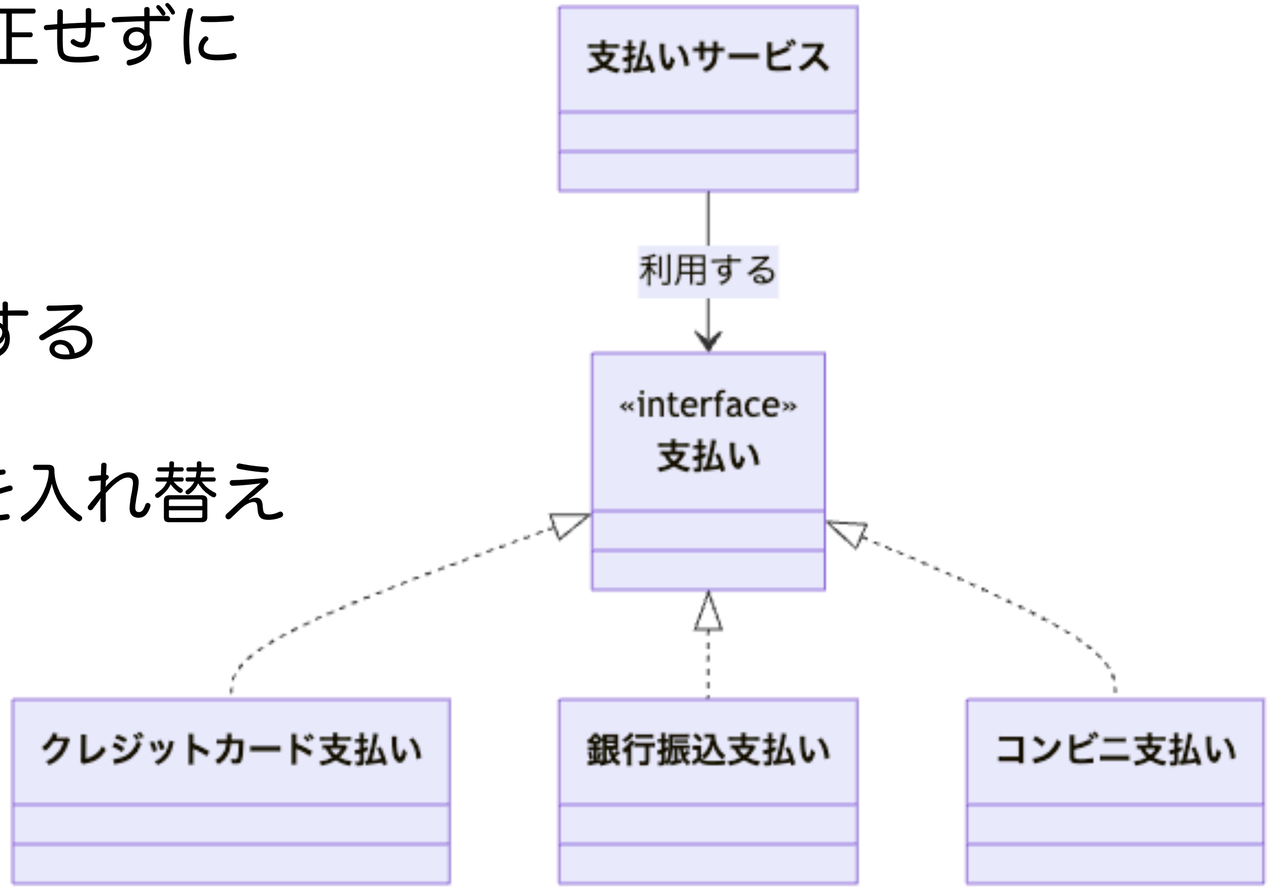
SRP: 単一責任の原則

- モジュールはひとつの責任を持つ……わけではない（名前が悪い）
- モジュールは「特定のアクターに責任を持つ」
  - 同じ理由で同じ時期に変更されるものはまとめておく
  - 違う理由で違う時期に変更されるものは分離しておく



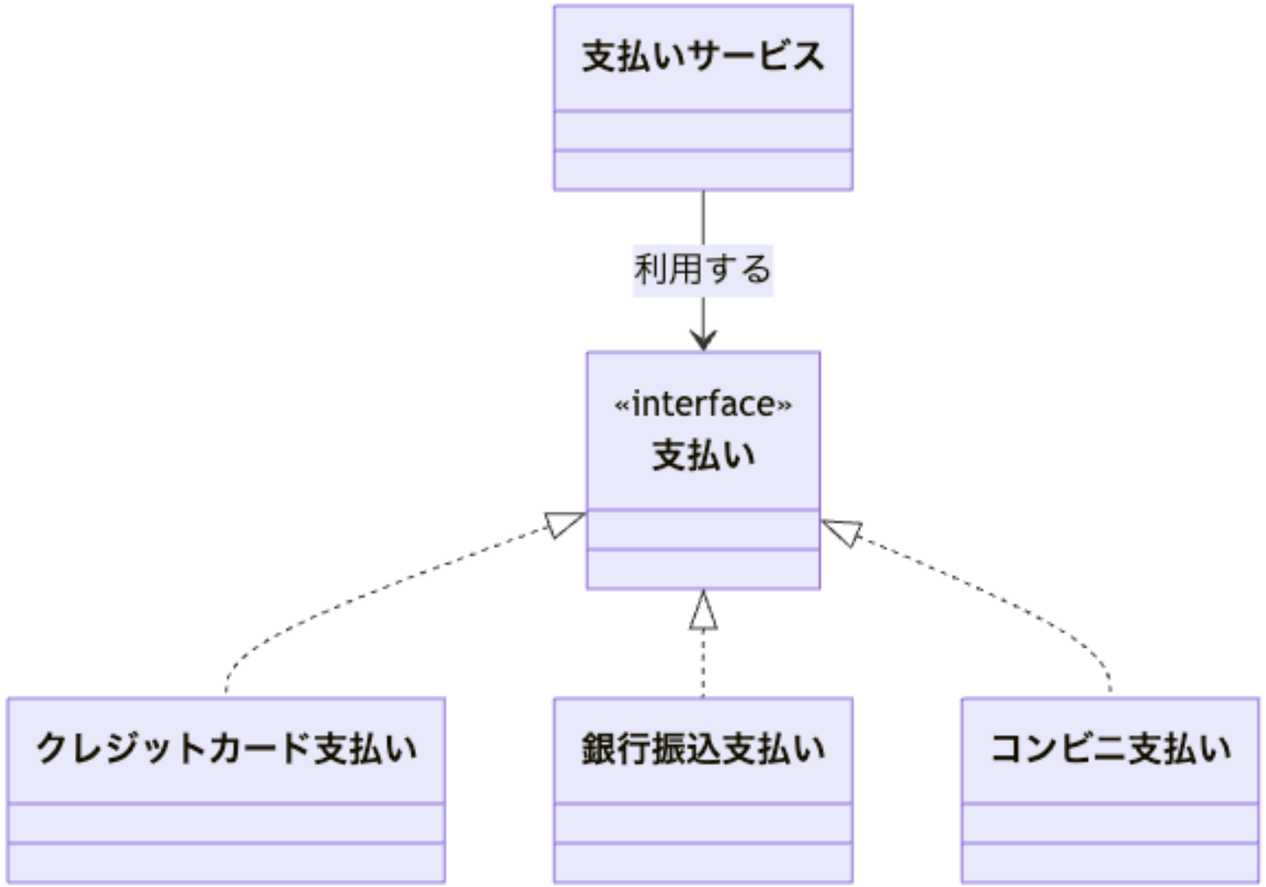
OCP: オープン・クローズドの原則

- 拡張にはオープンで、修正にはクローズである
  - 言い換えれば、既存のコードを修正せずに機能を追加したい
- 抽象化の仕組みを使って変更を制御する
  - OOP: ポリモーフィズムでクラスを入れ替え可能にする

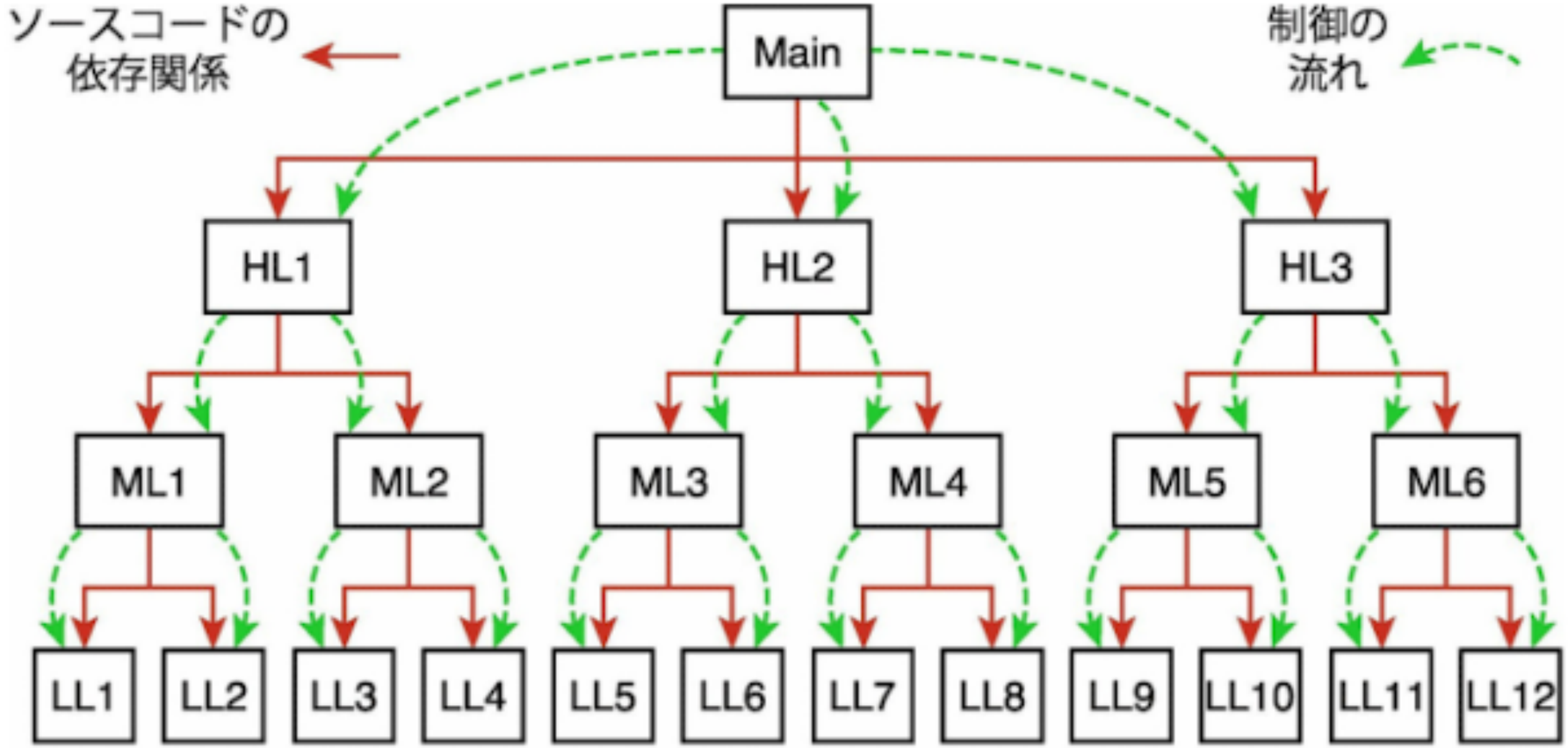


DIP: 依存関係逆転の原則

- 上位レベル（抽象）は下位レベル（具象）に依存しない
  - OOP: ポリモーフィズムを使用する



制御の流れとソースコードの依存関係



制御の流れ（緑）は仕方ないが、依存関係（赤）は好きに決めたい！

ポリモーフィズムによる依存関係の制御

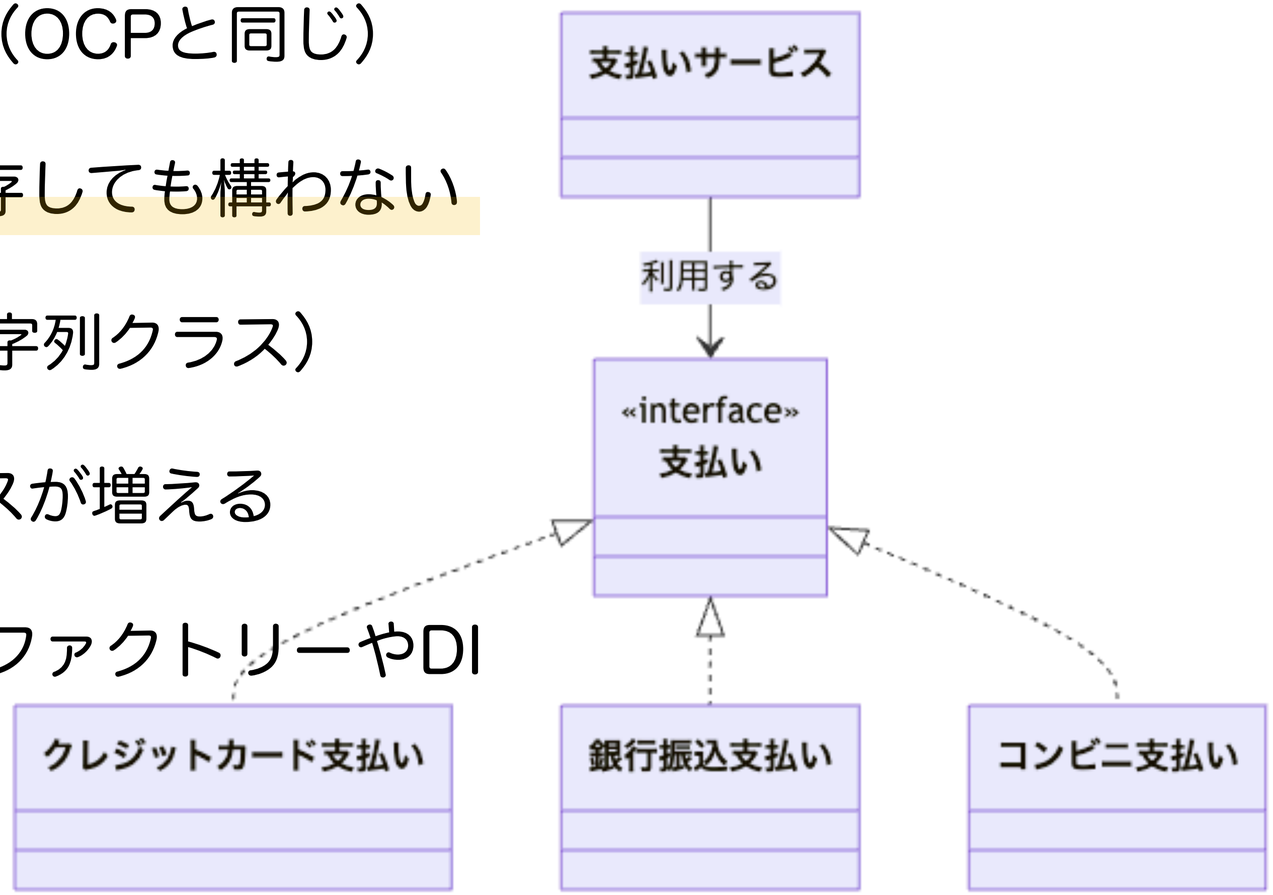
ポリモーフィズムを使えば、ソースコードの依存関係（赤）を逆転できる

そうすれば、

- 依存する側［詳細］を待たずに、依存される側［方針］を実装できる
- それぞれを独立して開発、テスト、デプロイできる

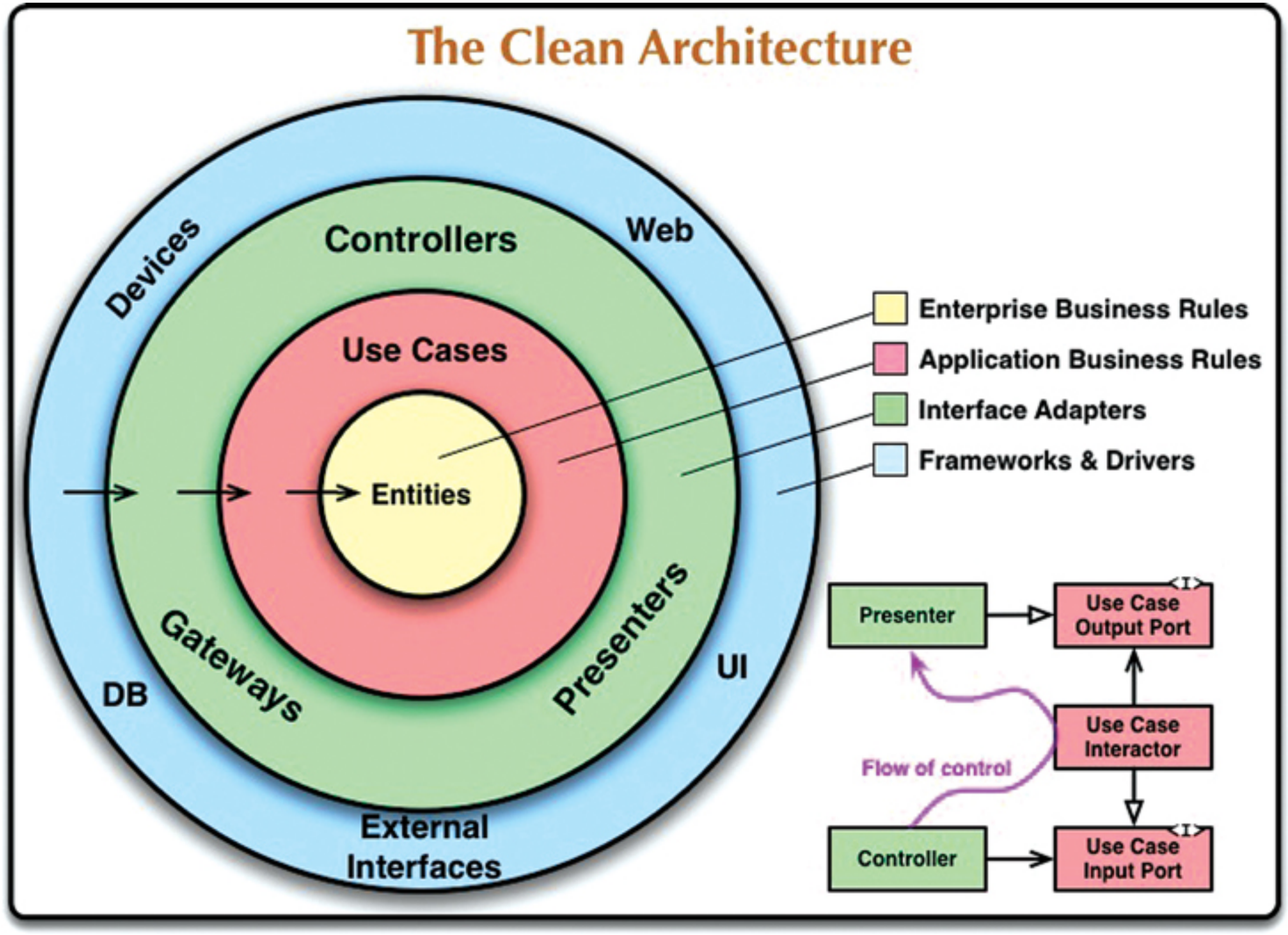
DIP: 依存関係逆転の原則

- 上位レベル（抽象）は下位レベル（具象）に依存しない
  - ・OOP: ポリモーフィズムを使用する（OCPと同じ）
- 例外として、安定しているものには依存しても構わない
  - ・たとえば、言語標準の機能（例：文字列クラス）
  - ・やりすぎると無駄なインターフェイスが増える
  - ・オブジェクトの生成も面倒になる（ファクトリーやDIの導入など）



6. クリーンアーキテクチャ

おなじみの同心円


The Clean Architecture
Devices
Controllers
Web
Use Cases
Entities
Gateways
Presenters
DB
UI
External
Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case
Output Port
Use Case
Interactor
Flow of control
Controller
Use Case
Input Port

叫ぶアーキテクチャ

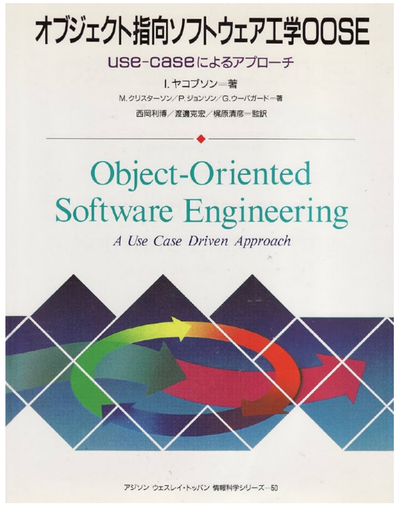
- システム構成（ファイル構成）を見れば、そのシステムの用途がわかる
  - ユースケースがモジュールとして存在すること
  - ドメインモデルにはドメインの用語が使用されていること

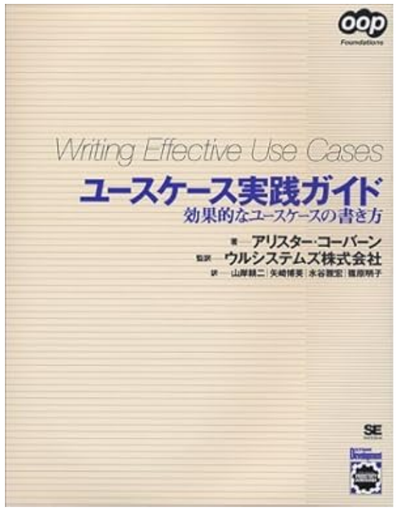
叫ぶアーキテクチャ

- システム構成（ファイル構成）を見れば、そのシステムの用途がわかる
  - ユースケースがモジュールとして存在すること
  - ドメインモデルにはドメインの用語が使用されていること
- アーキテクチャとはユースケースを実現するための構造である
  - そのために必要な人員は、少なければ少ないほど望ましい（簡単に変更できる）

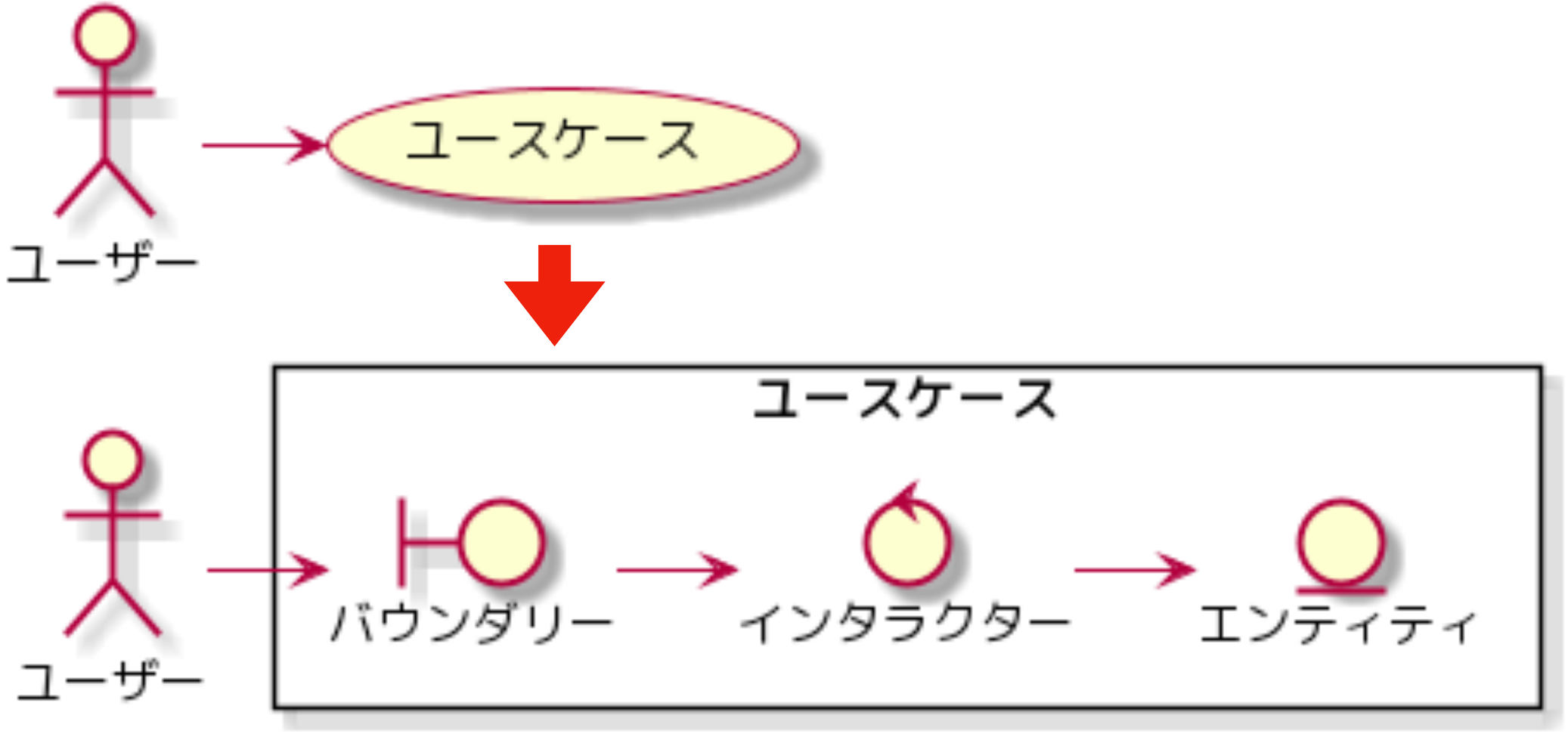
前提：ユースケースを中心に考える

- ユースケースとは、ユーザーが目標を達成するためにシステムを利用する方法のこと。データ構造やUIは規定しない。
  - Ivar Jacobsonが提唱（OOSE本）
  - Alistair Cockburnが発展および推進





ユースケースを分割する（BCE）

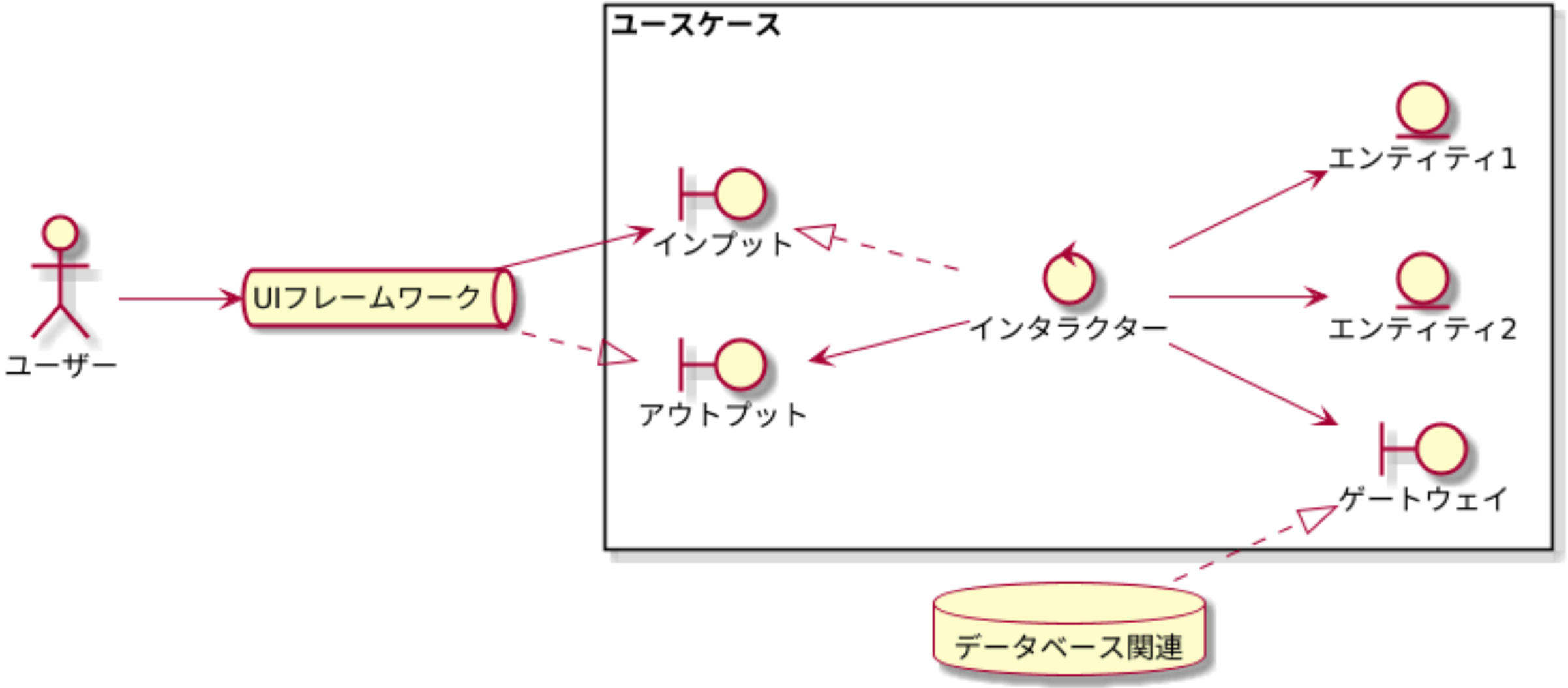


インターフェイス　アプリケーション　ドメインモデル

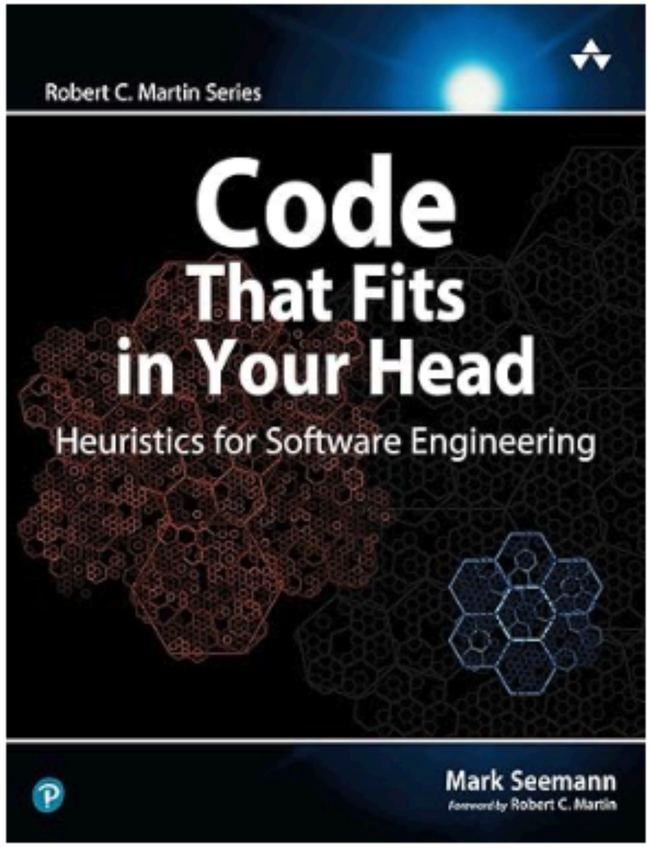
叫ぶアーキテクチャ

- システム構成（ファイル構成）を見れば、そのシステムの用途がわかる
  - ユースケースがモジュールとして存在すること
  - ドメインモデルにはドメインの用語が使用されていること
- アーキテクチャとはユースケースを実現するための構造である
  - そのために必要な人員は、少なければ少ないほど望ましい（簡単に変更できる）
- 変更のタイミングが異なるものは分離しておきたい
  - さらにはユースケース以外のことはあとで考えられるようにしたい

ユースケース以外はあとで考える



あとで考える → 認知的負荷の低下



まえがきをアンクル・ボブが書いている（彼のシリーズなので）

関心事の分離

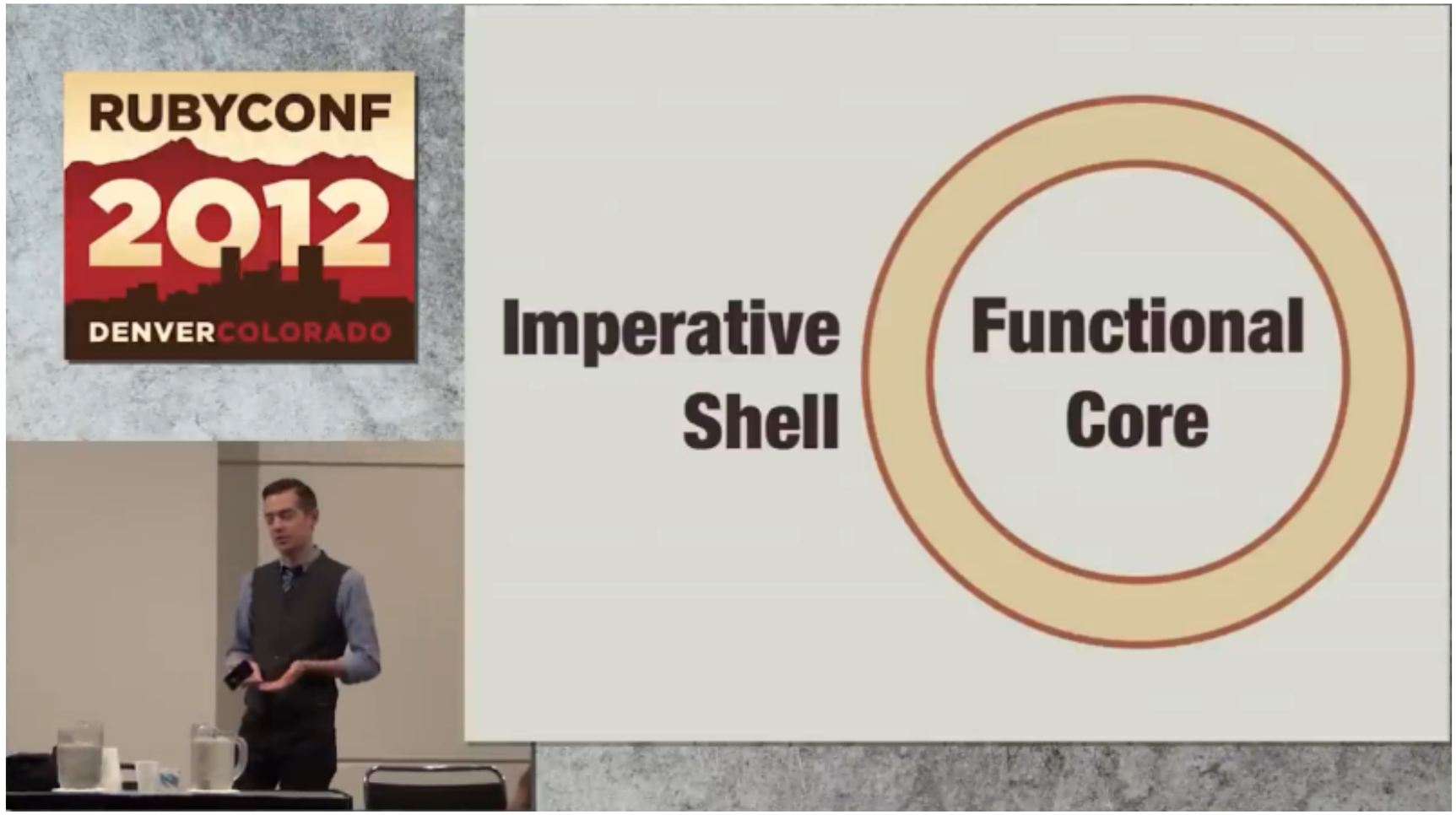
- 関係のない関心事を分離しましょう。ユーザーインターフェイスの変更に、データベースコードなどの変更が含まれるべきではありません。（OOPは分離は可能だが）重要なふるまいが隠れてしまうことになり、コードは脳に収まらなくなります。
- 私は、いわゆるオブジェクト指向のコードベースを捨ててHaskellに移行することを組織に期待しているわけではありません。関数型コア・命令型シェルへの移行を推奨しています。関数型コア・命令型シェルによって、コードベースで純粋でない操作を実装している箇所を簡単に独立させられます。

出典：『脳に収まるコードの書き方』（13章 関心事の分離）

関数型コア・命令型シェル

- 関数型と命令型を組み合わせた設計アプローチ
  - by Gary Bernhardt at Ruby Conf 2012
- 基本的な考え方：
  1. 関数型コア: システムで重要な部分（ビジネスルールなど）を表現
  2. 命令型シェル: 関数型を囲み、外部とのやりとりを処理

関数型コア・命令型シェル



https://www.youtube.com/watch?v=yTkzNHF6rMs

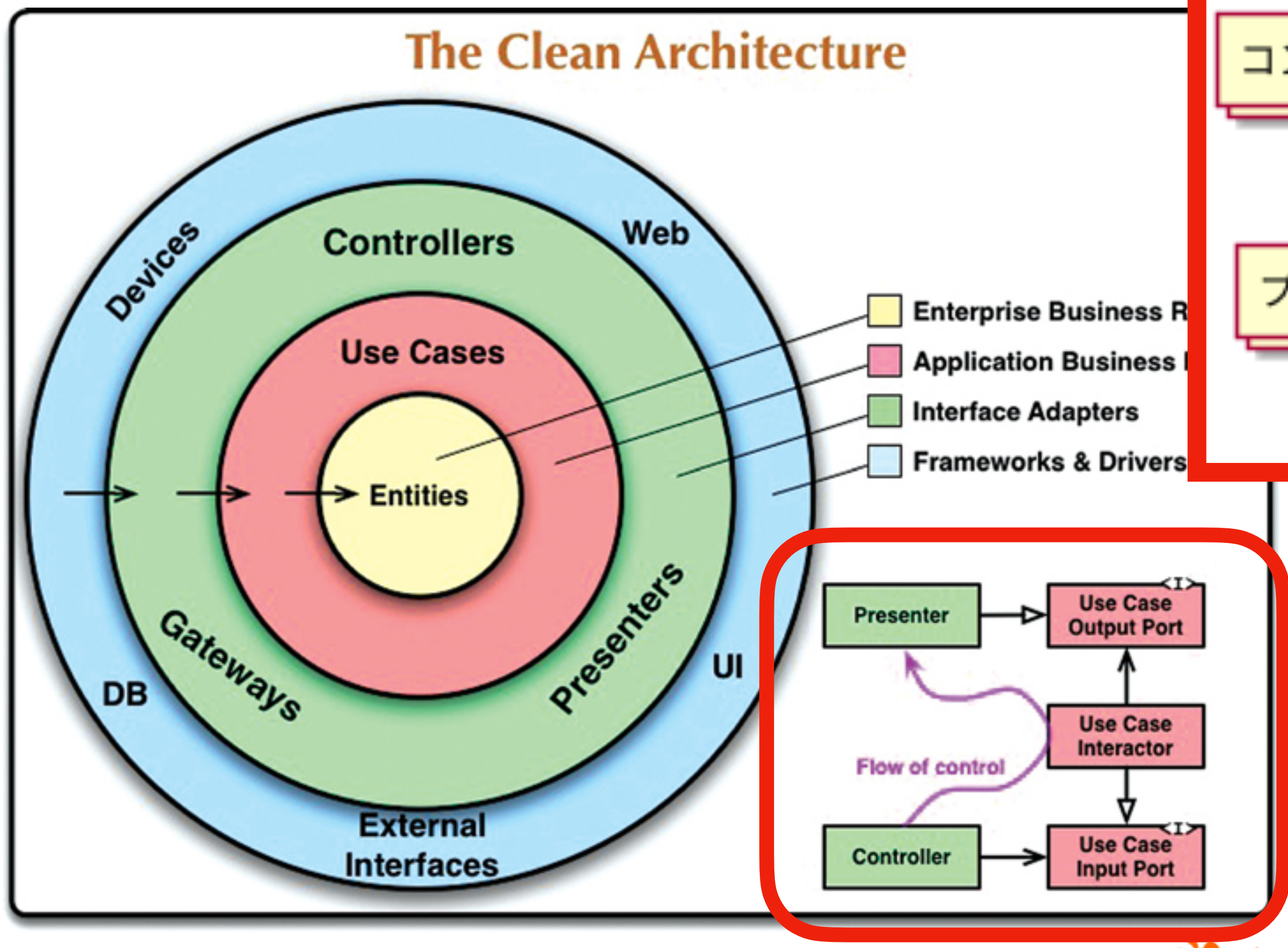
前に見た図に（なんとなく）似ている

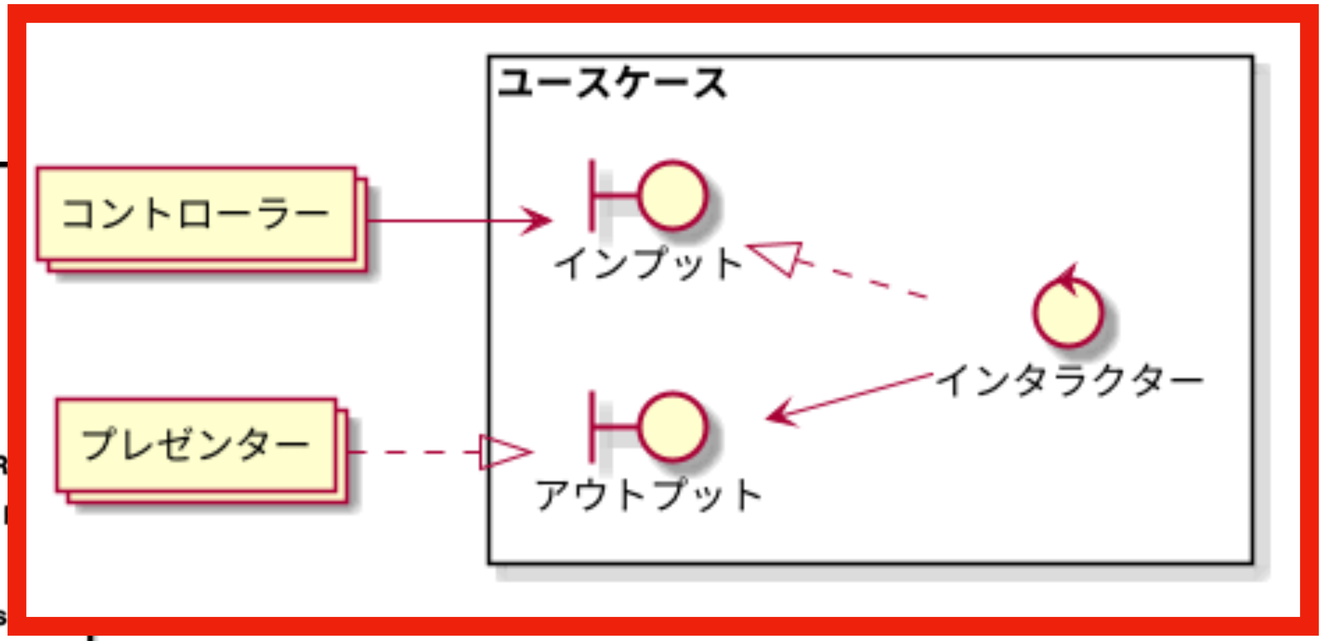
ポリモーフィズムを使えば、ソースコードの依存関係（赤）を逆転できる

そうすれば、

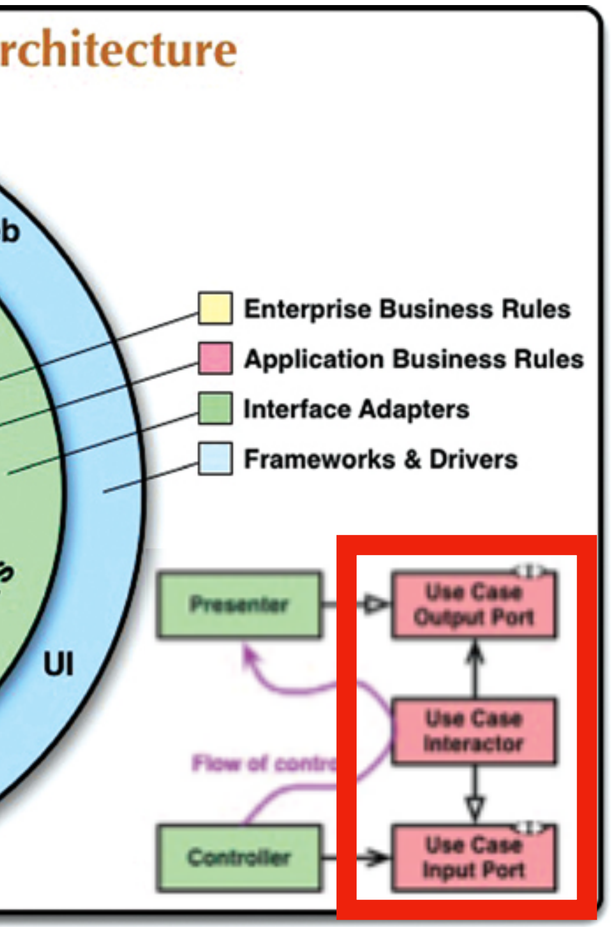
- ユースケース（ビジネスルール）以外のことをあとで決められる
- 関心事や変更のタイミングが異なるものが分離される → シンプルな設計

おなじみの同心円とは何だったのか





おなじみの同心円とは何だったのか



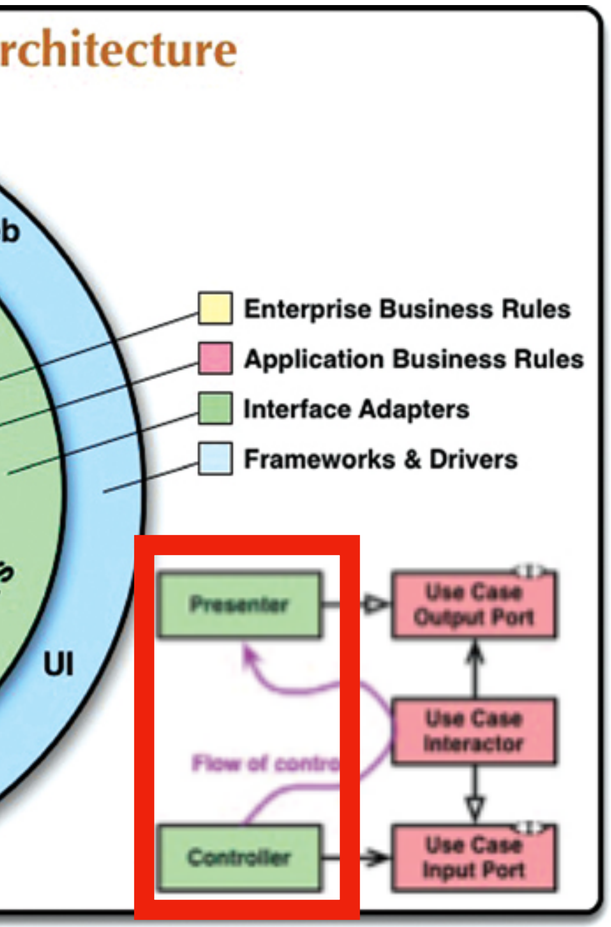

```ts
interface ForGreeting {
  execute(name: string): void
}

interface ForPresentingGreeting {
  present(message: string): void
}

class GreetUseCase implements ForGreeting {
  constructor(private readonly output: ForPresentingGreeting) {}

  execute(name: string): void {
    const msg = `Hello, ${name}!`
    this.output.present(msg)
  }
}
```

おなじみの同心円とは何だったのか




```
class GreetController {
  constructor(private readonly greet: ForGreeting) {}

  greetUser(name: string) {
    this.greet.execute(name)
  }
}

class GreetPresenter implements ForPresentingGreeting {
  present(message: string): void {
    console.log(message)
  }
}
```

```
const presenter = new GreetPresenter()
const usecase = new GreetUseCase(presenter)
const controller = new GreetController(usecase)

controller.greetUser("World")
```

全体のまとめ

- アンクル・ボブはいろんな経験をしてきた熟練のおじいちゃん
- ソフトウェア中心の世界ではソフトウェアが社会に害を与えてはならない
- ソフトウェア開発者は「倫理・基準・規律」を持たなければならない
- 現代の「規律」はXPの手法（テスト駆動開発やシンプルな設計など）
- ソフトウェアの設計は振る舞いと構造のバランスを取らなければならない
- アーキテクチャを見れば、そのシステムの用途がわからなければならない
- クリーンアーキテクチャ：ユースケース中心 + SOLID原則（依存関係の管理）

読んでみてください！


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


Clean Craftsmanship
クリーンクラフトマンシップ
Clean Craftsmanship
規律、基準、倫理
Robert C. Martin 著
角 征典 訳
より良く働き
生産性を高め
自分が書いたものに
誇りを持つ！
ASCII DWANGO