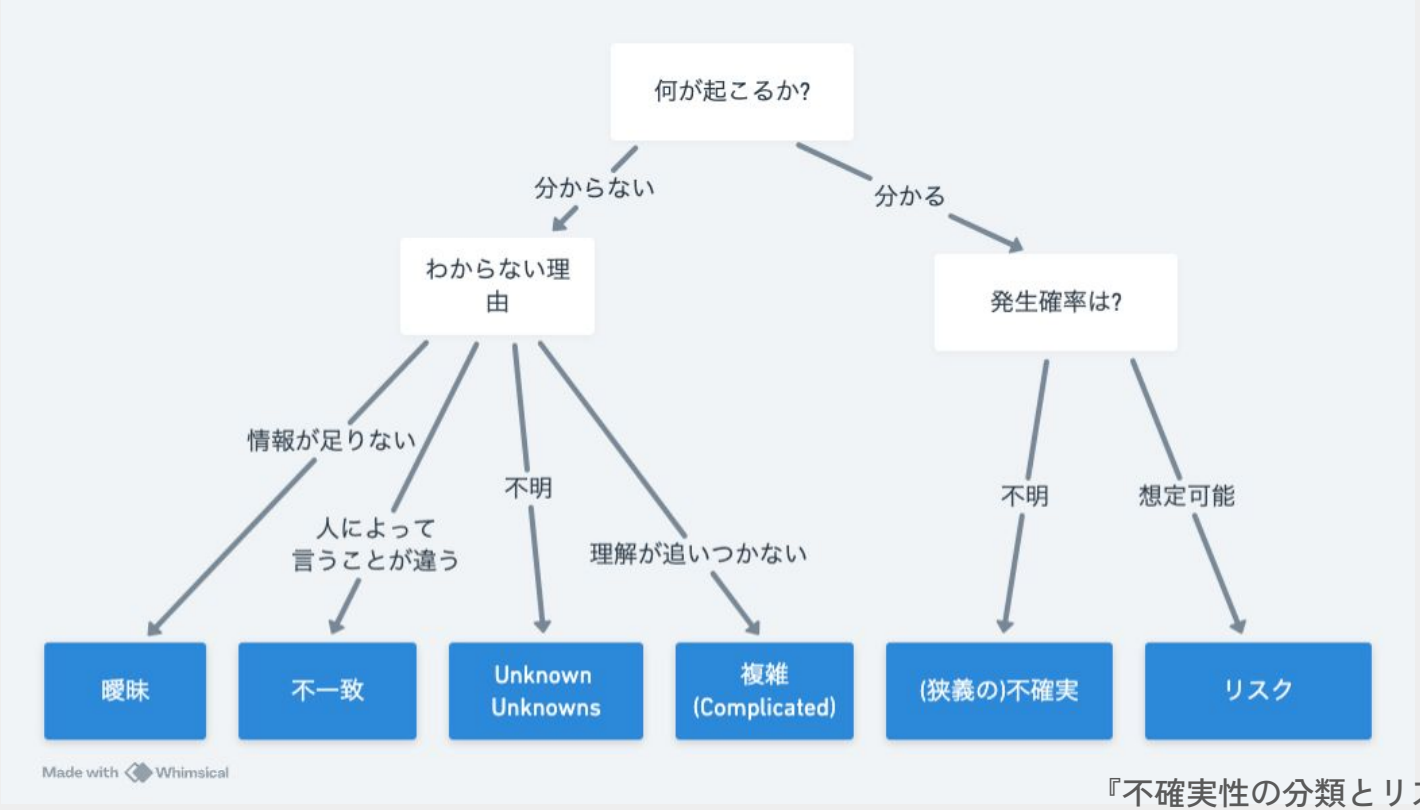
現代的システム開発概論 2024

株式会社ウルフチーフ
川島義隆

この講座で学んでほしいこと

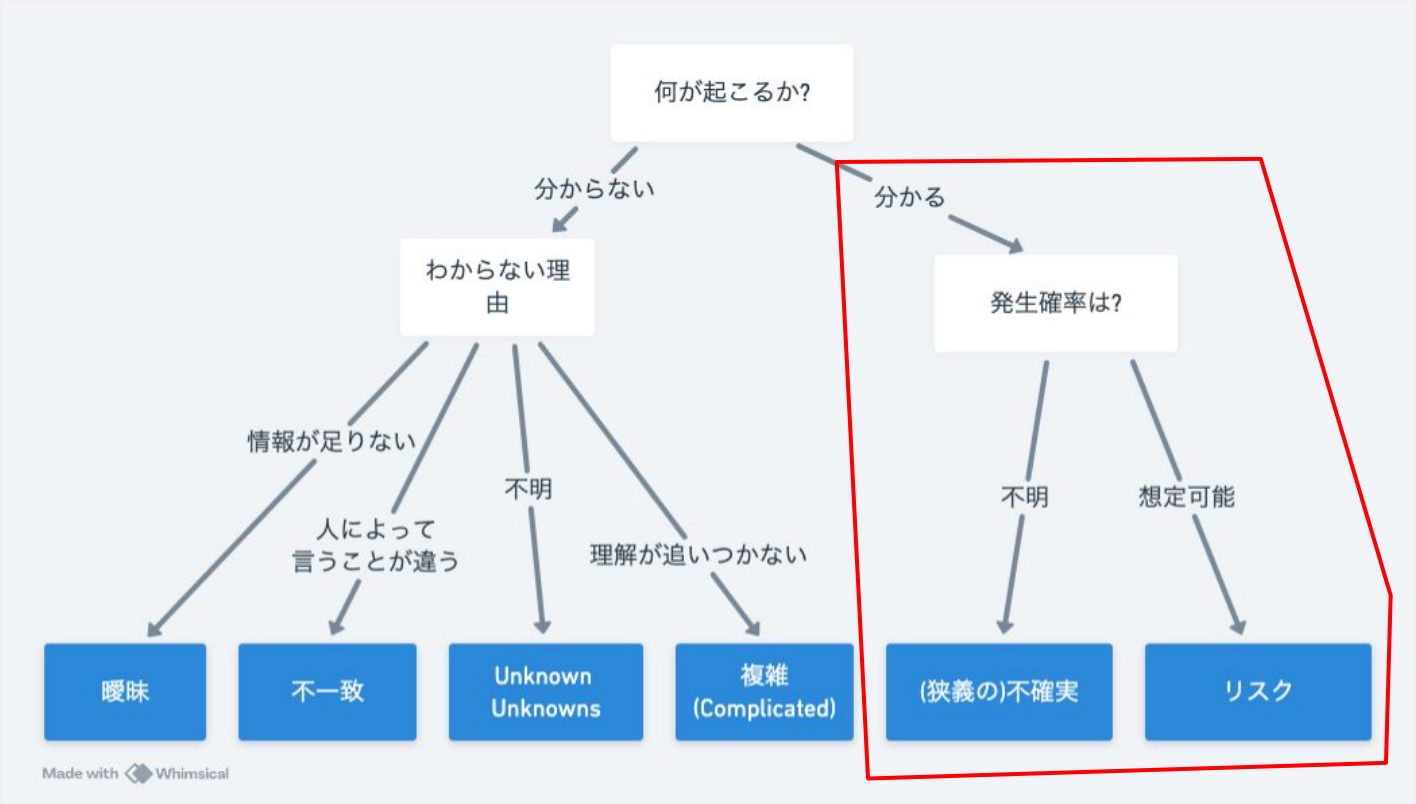
不確実性と複雑性の切り口から「ソフトウェアエンジニアリング」のNowを見れるようになること。

不確実性の分類



『不確実性の分類とリスク評価』

まずは右側の不確実性へのアプローチから



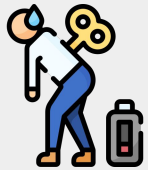
不確実性がプロジェクトに影響を与える

問題が起きることは予測できる、確率も過去の観測から導き出せる

はずなのに、それを見込んだ**計画**になっていない。

ある程度の不確実性は想定していたはずなのに、**バッファ**では

吸収しきれなかった。



What is ‘計画’

計画の「変数」を予測、コントロールの方針を立てる

- 開発手法
- 成果物
- 組織の要求事項
- 市場の状況
- 法律または規制による制限
- デリバリー
- 見積
- スケジュール
- 予算

すべてに不確実性が含まれる

PMBOK 7

計画

計画する行為により、不確実性(があること)を明らかにする。

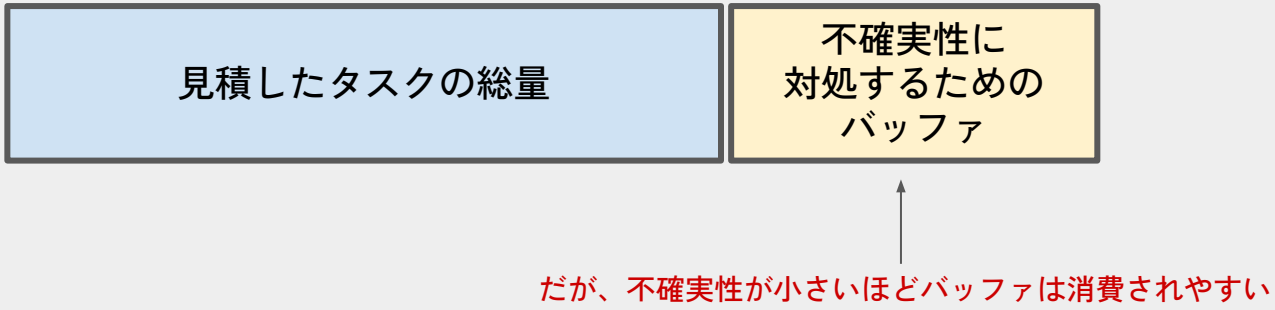
In preparing for battle I have always found that plans are useless,

but planning is indispensable.

Dwight D. Eisenhower

不確実性が少なければ...

不確実性の分だけバッファを用意して、計画を立てていればうまくいく



学生症候群

締切があると、そこから逆算し、取り掛からないとヤバいところまで来ないと課題に取り組まない。

https://ja.wikipedia.org/wiki/学生症候群

パーキンソンの法則

仕事は利用可能な時間を埋めるまで拡大する

https://ja.wikipedia.org/wiki/パーキンソンの法則

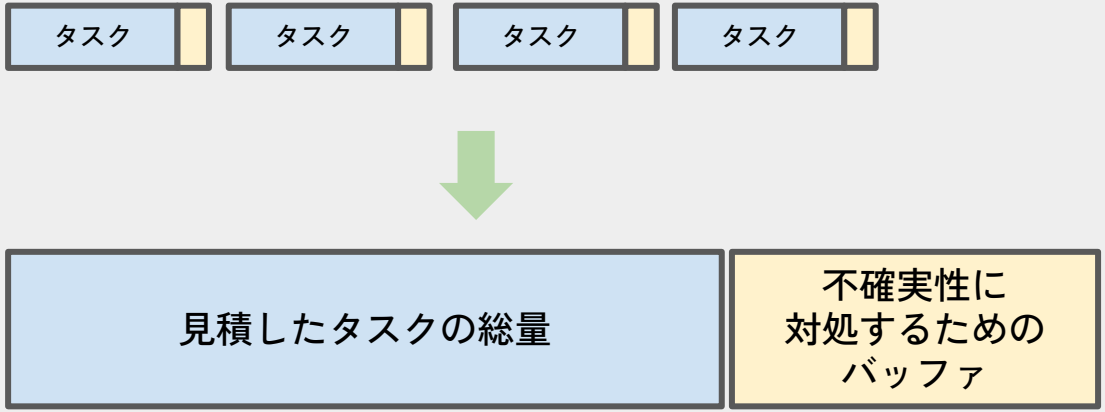
Planning Fallacy

将来のタスクを完了するのにどれくらいの時間が必要かを予測する際に楽観バイアスが働き、必要な時間を過小評価する現象

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

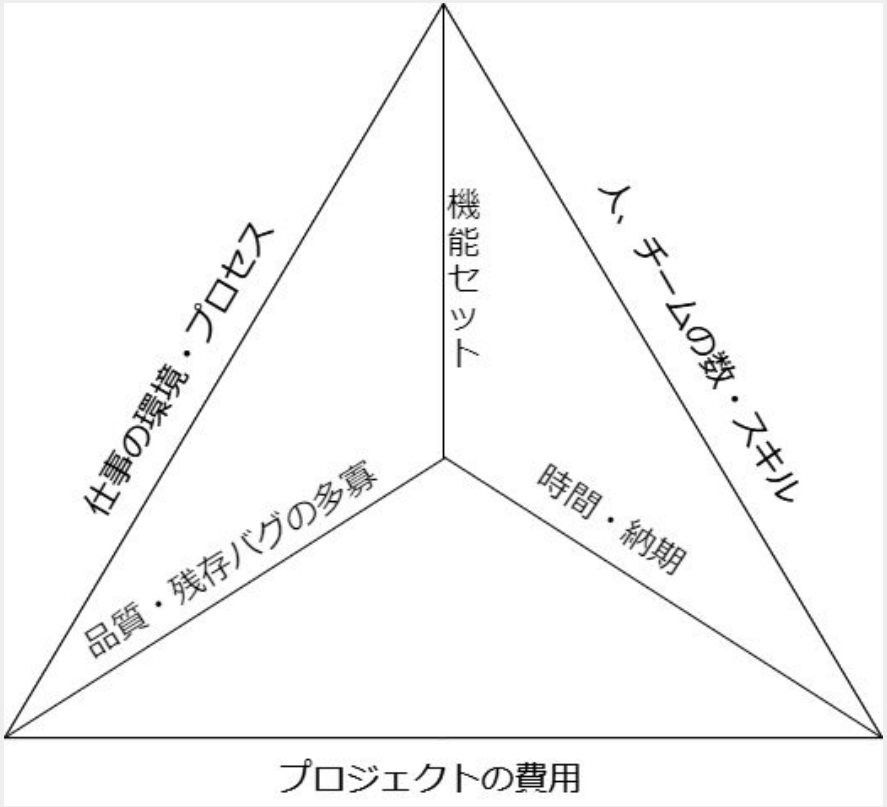
バッファは一箇所に集めないと食い尽くされる

プロジェクト全体で計画とバッファの管理することが重要



『クリティカルチェーン―なぜ、プロジェクトは予定どおりに進まないのか?』

プロジェクトピラミッド



変数は大まかに分類すると...

Q: 品質
C: 費用
D: デリバリー
S: 機能セット(スコープ)

バッファをどこに持たせどうコントロールするか?

https://scrapbox.io/kawasima/プロジェクトの変数コントロール

不確実性を扱う

複数あるプロジェクト変数を固定し、制御変数を1つに絞る。

制御変数にバッファを持たせる。

コスト可変型

成果物
デリバリー
コスト(可変)

スコープ可変型

成果物(可変)
デリバリー
コスト

あえて優先順位を付けて、優先順位の高いものを固定する

コスト可変型

納期に間に合わない

品質がリリース基準を満たさない

…

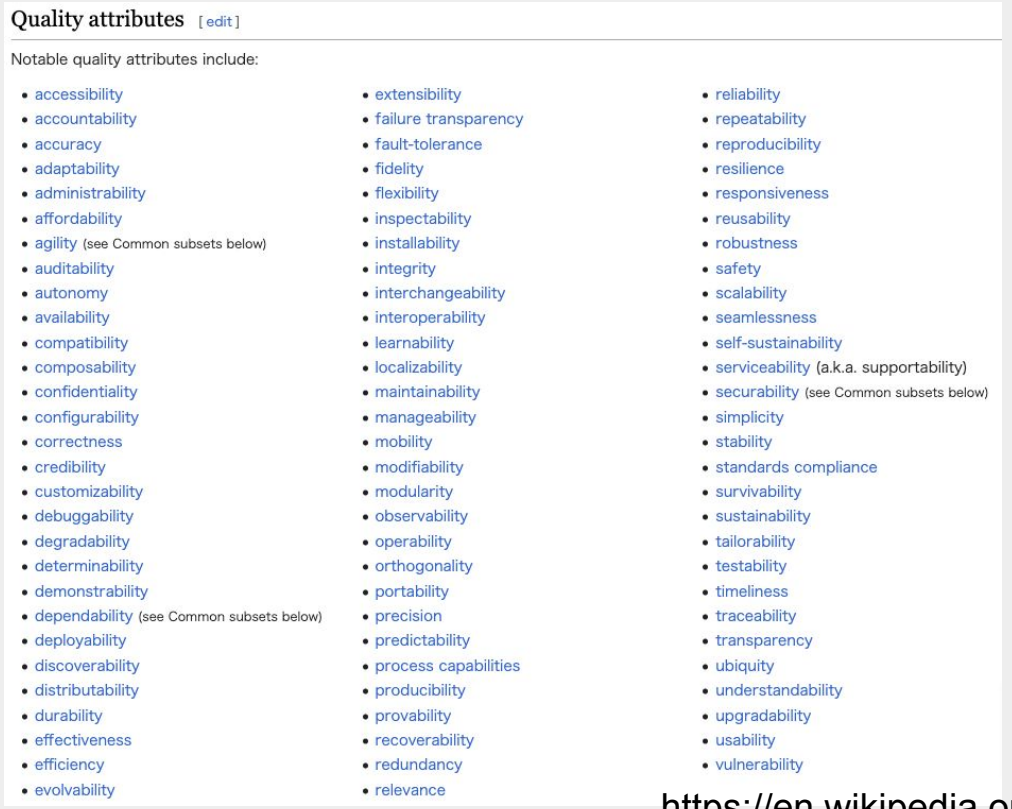
→ お金で解決する

→ コスト可変とはいえ、急に用意はできないので、不確実性の分だけ事前に用意しておく。

スコープ可変型

- 納期とコストを固定して、成果物の量でコントロールする。
- サービスとして体を成さない機能しかできない恐れがある
- ので、単一デリバリー型のプロジェクトではスコープ可変型は適さない。

品質は変数でないのか?



Quality attributes [edit]

Notable quality attributes include:

- accessibility
- accountability
- accuracy
- adaptability
- administrability
- affordability
- agility (see Common subsets below)
- auditability
- autonomy
- availability
- compatibility
- composability
- confidentiality
- configurability
- correctness
- credibility
- customizability
- debuggability
- degradability
- determinability
- demonstrability
- dependability (see Common subsets below)
- deployability
- discoverability
- distributability
- durability
- effectiveness
- efficiency
- evolvability
- extensibility
- failure transparency
- fault-tolerance
- fidelity
- flexibility
- inspectability
- installability
- integrity
- interchangeability
- interoperability
- learnability
- localizability
- maintainability
- manageability
- mobility
- modifiability
- modularity
- observability
- operability
- orthogonality
- portability
- precision
- predictability
- process capabilities
- producibility
- provability
- recoverability
- redundancy
- relevance
- reliability
- repeatability
- reproducibility
- resilience
- responsiveness
- reusability
- robustness
- safety
- scalability
- seamlessness
- self-sustainability
- serviceability (a.k.a. supportability)
- securability (see Common subsets below)
- simplicity
- stability
- standards compliance
- survivability
- sustainability
- tailorability
- testability
- timeliness
- traceability
- transparency
- ubiquity
- understandability
- upgradability
- usability
- vulnerability

品質特性は山ほどあって、意図的にコントロールするのは難しい。

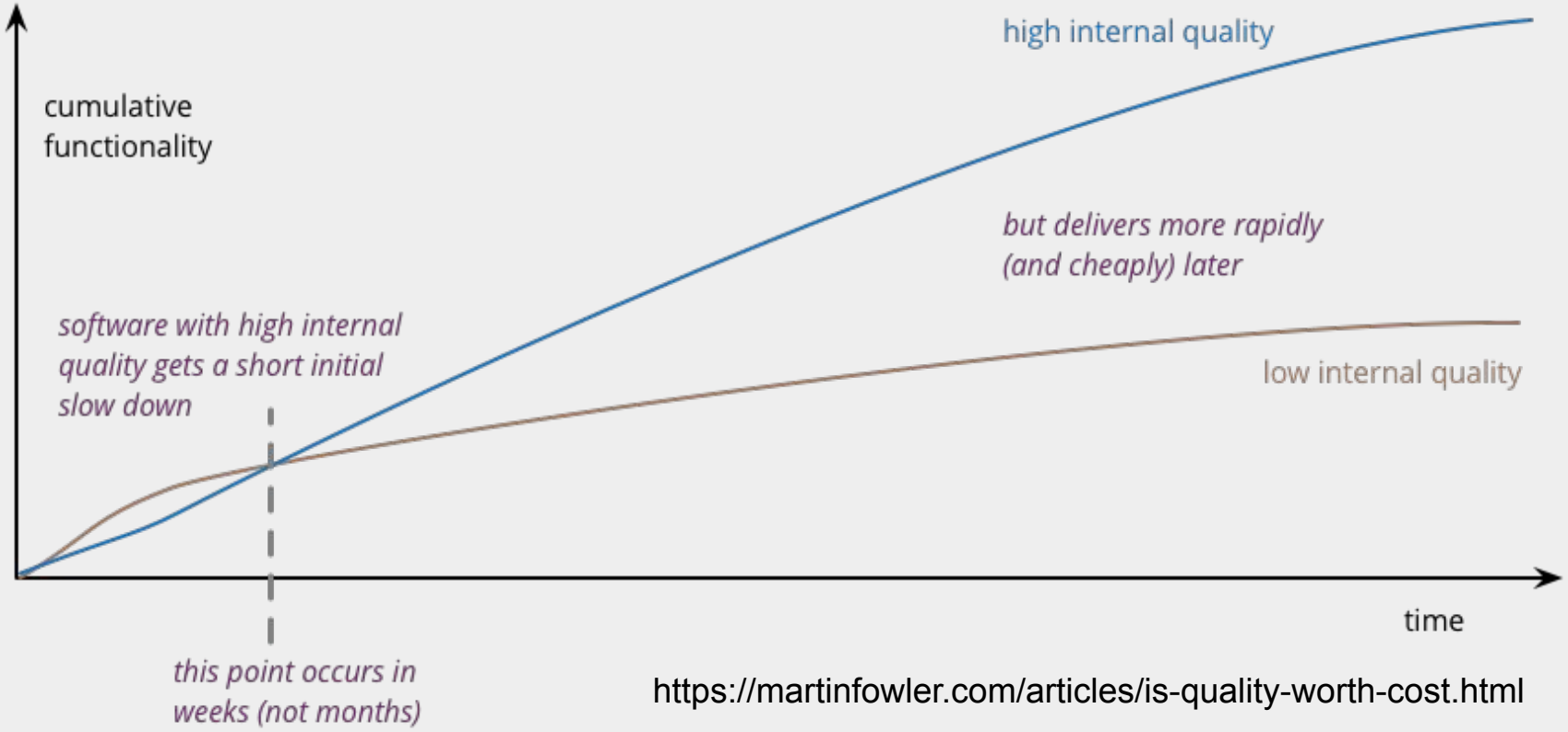
https://en.wikipedia.org/wiki/List_of_system_quality_attributes

品質は他の変数と正の相関がある。

成果物、デリバリー、コストを固定して、品質をコントロールすることは出来ない。

- 成果物の完成定義には通常、品質基準が含まれる。
- 品質特性のうち、機能性は成果物と強い正の相関がある。
- 品質特性のうち、可用性・性能やセキュリティは一定以上の閾値を越えなければ、サービス存続が出来ないノックアウトファクターになる。

(内部)品質とスピードはトレードオフではない



https://martinfowler.com/articles/is-quality-worth-cost.html

ここまでのまとめ

- プロジェクトの計画を立てる行為によって不確実性を洗い出す。
- 不確実性による計画の影響は、可変なプロジェクト変数によって吸収する。
- 可変にできるプロジェクト変数は現実的には以下2つ
  - コストを可変にする
  - スコープを可変にする
- 逆に言えば、固定するプロジェクト変数は、プロジェクトオーナーからすると「譲れないもの」である可能性が高い。

具体的にどうプロジェクト計画していけば良いの？

開発ライフサイクル

種類	ライフサイクルの例	長所および成功に必要な条件
逐次型 (予測型)	ウォーターフォール フェーズゲート	● コストのリスクを管理できる ● 既知かつ合意済みの要件 ● アーキテクチャをよく理解できている ● プロジェクトの過程で要件が変更されない
反復型	スパイラル 進化的プロトタイピング	● 技術的リスクを管理できる ● 要件が進化し続ける
漸進型	スケジュールに応じた設計、段階的納品	● スケジュールのリスクを管理できる
反復漸進型 (適応型)	アジャイル	● スケジュールと技術的の両方のリスクを管理できる ● 全てのメンバが同一サイトで作業を行える ● 統合チーム以外では円滑な進行が難しい
場当たり型	Code and Fix	

『Manage It! 3.2 ライフサイクルの概要』

逐次型

作るものが予測可能 (不確実性は含む)であることを前提とする

不確実性をコスト/時間に転化する

プロジェクト計画時点では、不確実性のボリュームを想定し、
Negativeな方に現実化された時の対処費用と対応時間を算出しバッファとして積んでおく

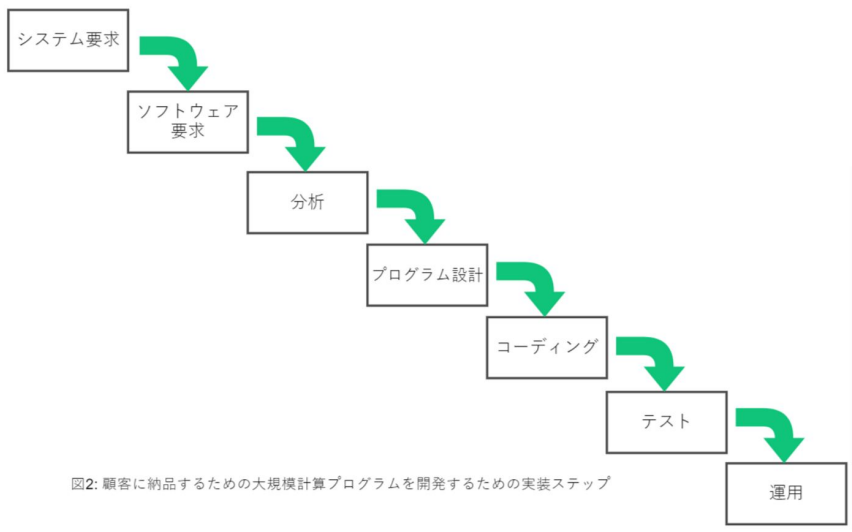


図2: 顧客に納品するための大規模計算プログラムを開発するための実装ステップ

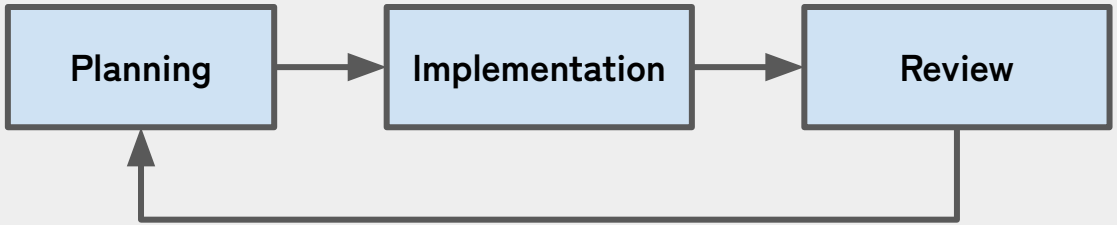
コスト可変型と相性が良い

適応型

最終的に作られるものを正確に予測できないことを前提とする。

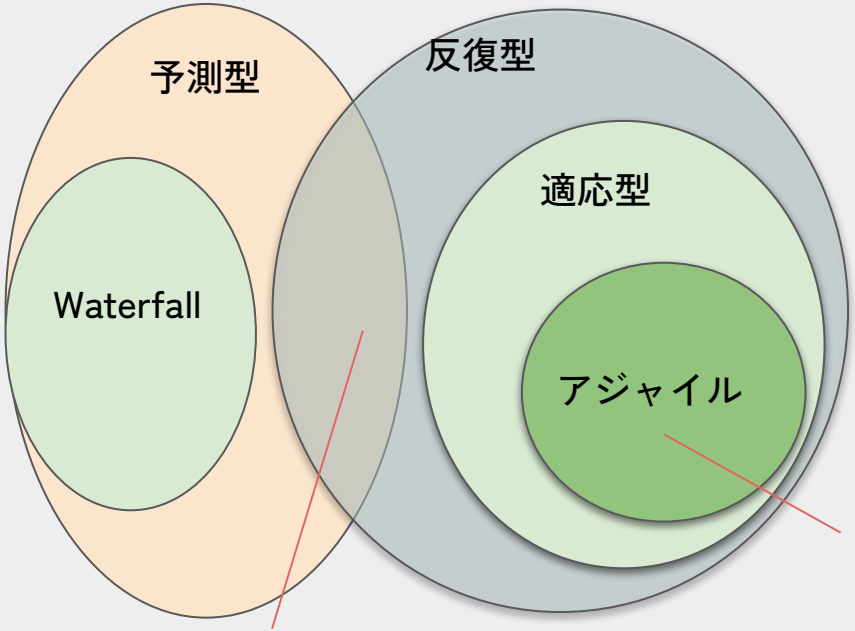
不確実性はスコープによって調整する。

1開発サイクルの規模が小さいほど不確実性も小さくなる



スコープ可変型と相性が良い

開発ライフサイクルの関係性

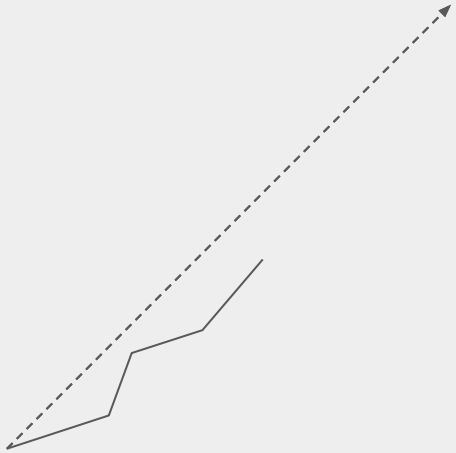


アジャイルプロセスは適応型でなければならない

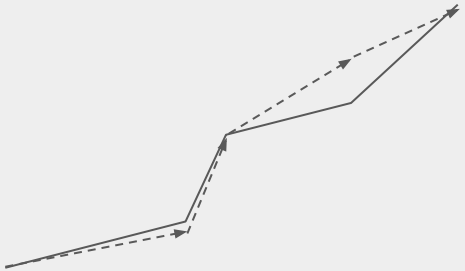
反復型でも機能固定でやる場合は、予測型計画

https://martinfowler.com/bliki/WaterfallProcess.html

予測型と適応型プロセスの違い(イメージ図)



目標が定まっている
不確実性はバッファでコントロールする
目標と実績のズレを検知し、是正しながら進む



目標が動き続けるので、
(大きな方針はある)
不確実性を目標を近くに置くことで減らす

それぞれの不確実性の扱い

- 予測型開発ライフサイクル + コスト可変型
  - 不確実性の見積もりが上振れすると破綻する
- 適応型開発ライフサイクル + スコープ可変型
  - 結果的にリリース基準に達しないものしか作れませんでした...では困るので、小さなMVPや短期間のイテレーション

なぜソフトウェア開発の世界で予測型が未だに存在するのか？

それだけスコープと納期を固定することに事業価値を見出しているということ

- 事業計画では投資判断のために、「スコープ」「納期」「コスト」を確定値として一度に、コミットする。
- スコープを約束するので、予測型にならざるを得ない。
- プロジェクト変数としてコントロールできるもののうち、バッファを見込めるのは「納期(時間)」と「コスト」
- ソフトウェア開発の世界は、労働集約型なので「時間」と「コスト」には強い相関がある。
- よって、よりコントロールしやすい「コスト」を可変変数として使う。

スコープをコミットするんじゃなくて、
ビジネスゴールをコミットすればいいのに...

不確実性の量をどうやって見積もり・計画するか?

リスクマネジメントはある程度
手法が確立されている

リスク

Risk	Issue
未来にフォーカス	現在・過去にフォーカス
プロジェクト期間中に、主要なリソースがいなくなるかもしれない。	チームメンバが離任する。最終日はxxなので、引き継ぎを計画する
チームメンバがプロジェクトの大事な期間に休暇をとるかもしれない。	チームメンバが休暇をとった時、他の誰も対処できないことがある。
予期せぬ要求の変更があるかもしれない	プロジェクトのスコープに追加するべき機能が見つかった
影響分析すると、プロジェクトのスケジュールを遅延させる問題が見つかるかもしれない	影響分析の結果、プロジェクトを一週間後ろ倒ししそうな新たな問題が2つ見つかった
マトリックス型の組織でプロジェクトを進めると、現場が混乱し生産性が低下するかもしれない	我々の組織はマトリックス型なので、PMの権限と説明責任について混乱を減らすために、それを明記した文書を作成する必要がある

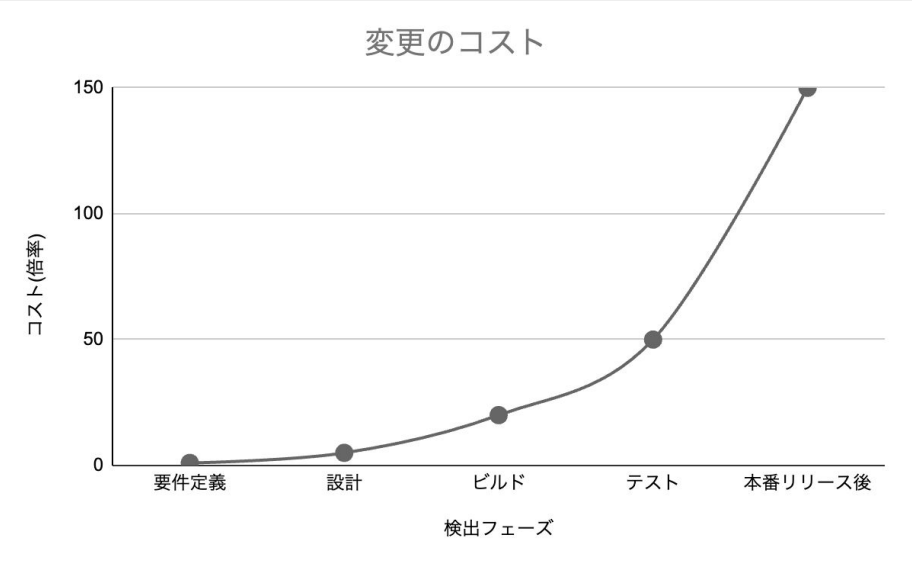
https://www.linkedin.com/pulse/issue-risk-what-difference-sanjeev-kulkarni/

IssueとRiskの役割の違い

- Issue
  - 顕在化した不確実性要素の内部シェア
  - タスク化してバックログに積む
- Risk
  - 対処予算の確保
  - エグゼクティブラインを動かす

変更コスト

ソフトウェアの要求仕様を作ってから、デリバリーまでの間に欠陥が見つかるのが遅ければ遅いほど、コストが高くつく。

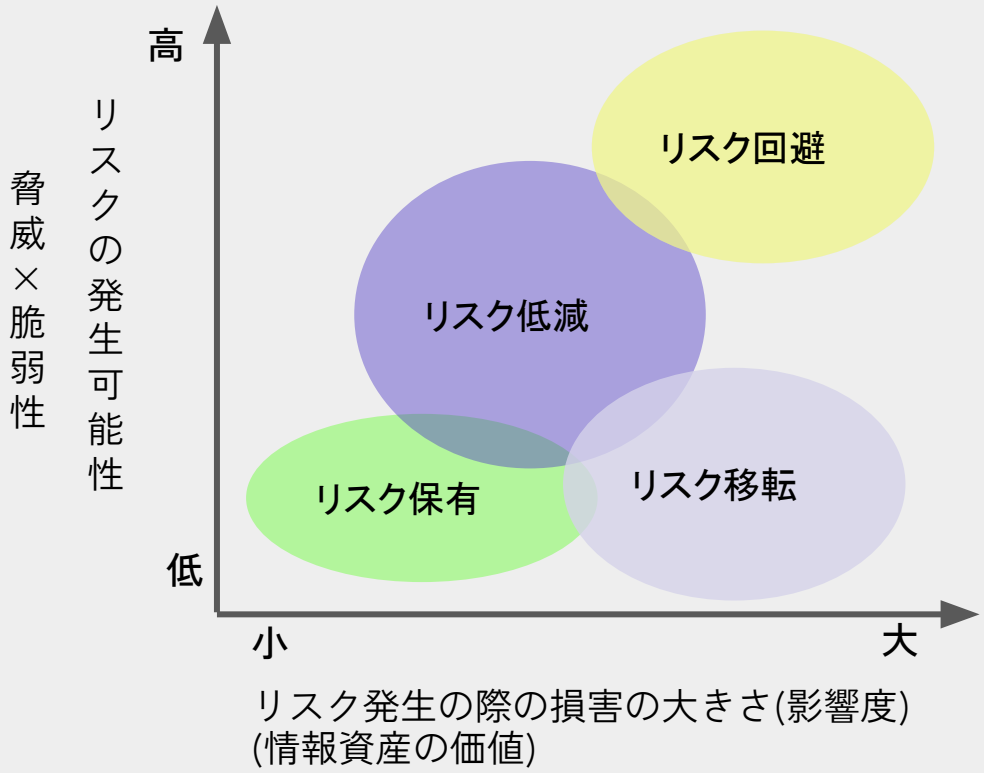


非線形なので、不確実性をコストに変換する予測型は、この面でも難しさがある。

いつ検出できるか、でコストが大きく変わるので見積もりが難しい。

『Boehmの変更コスト曲線』

リスク対応戦略


高
リスクの発生可能性
脅威×脆弱性
低
小
大
リスク回避
リスク低減
リスク保有
リスク移転
リスク発生の際の損害の大きさ(影響度)
(情報資産の価値)

発生確率をどうやって計算するのか?

過去の統計に頼る

- タスクのハンドオフの遅れ
- 類似のプロジェクトで発生した仕様変更の量
- 類似のプロジェクトで発生した不具合検出の量

リスク低減の2つのベクトル

$$Availability = \frac{MTTF}{MTTF + MTTR}$$

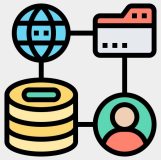
- 予防
  - 問題の発生確率を下げる予防策に投資する。
- レジリエンス
  - 問題が発生することを防止するよりも、問題が発生した時に元に戻すスピード、代替手段に投資する。

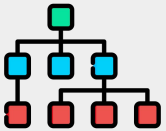
スコープを決める

主要な成果物と各成果物の受け入れ基準を特定する。

当然ここにも不確実性の考慮が必要

予測型





最終的なシステム像を予測し、必要なものを構造化する

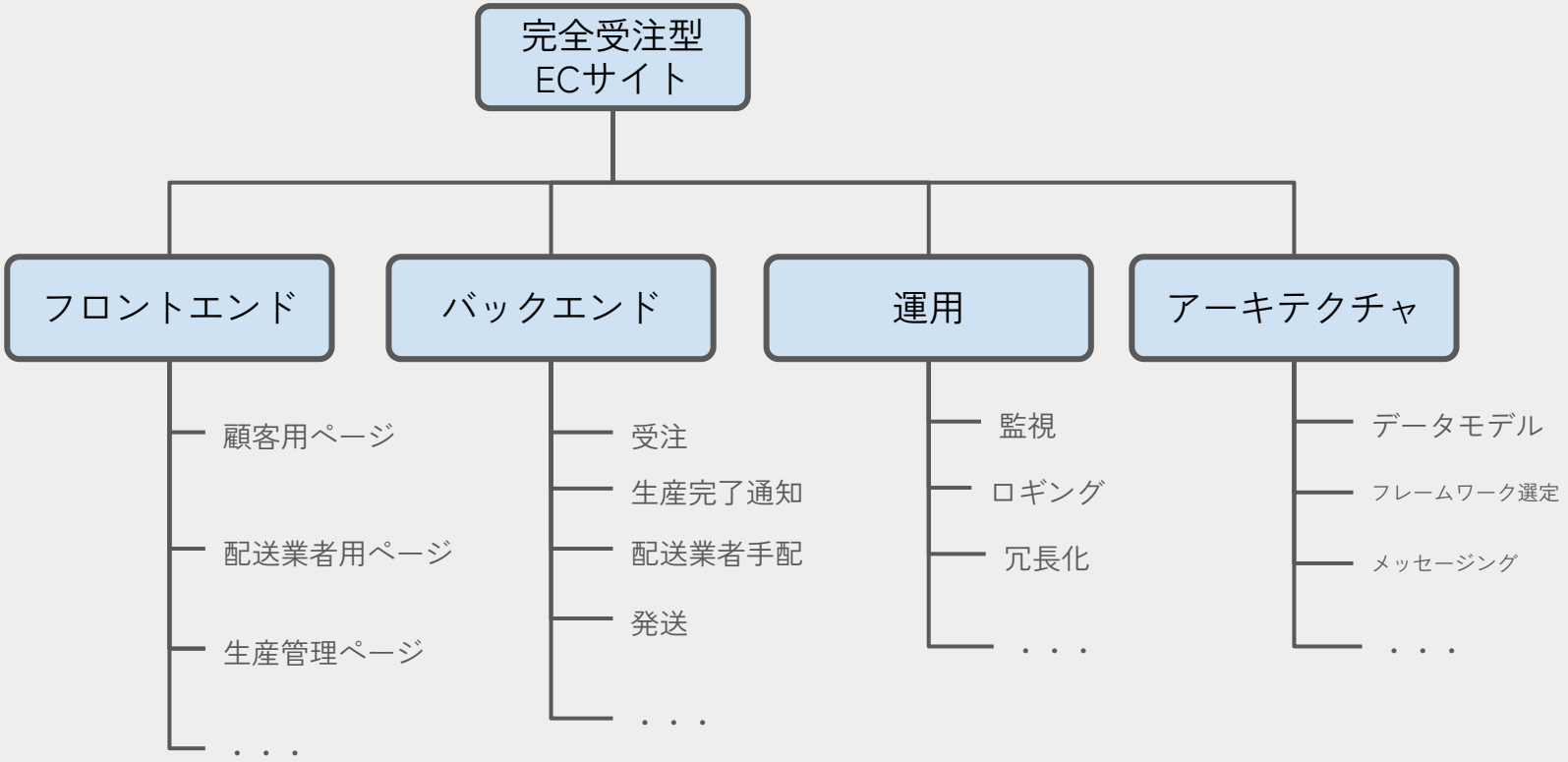
適応型

最終的なシステム像は、現時点の想定から変わる可能性が高いので、ハイレベルのビジネスゴールだけを決めておき、MVPから作っていく。

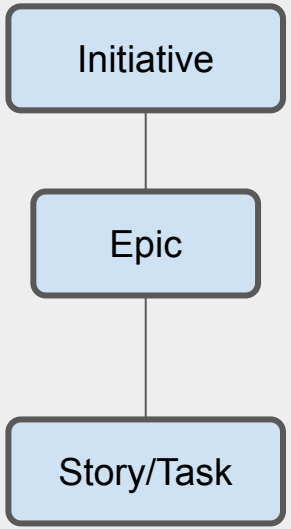
WBS

プロジェクトを進めるに当たって、必要なものを洗い出さないと計画を立てることができないので、階層構造として洗い出す。基本的には予測型計画の手法。

- 100%ルール (いわゆるMECEというヤツ)
- 基本的には成果物ベースでブレイクダウン (NOT タスクベース)
- 実行順序や実行時間はWBSには含めない (複雑さが増すため)


完全受注型
ECサイト
フロントエンド
バックエンド
運用
アーキテクチャ
顧客用ページ
配送業者用ページ
生産管理ページ
・・・
受注
生産完了通知
配送業者手配
発送
・・・
監視
ロギング
冗長化
・・・
データモデル
フレームワーク選定
メッセージング
・・・

適応型のスコープ決定



予測可能でないことが前提なので、

- WBSの100%ルールは適用しない
- 必要な範囲だけ詳細化(ブレイクダウン)する

https://www.atlassian.com/ja/agile/project-management/epics-stories-themes

User Story

ユーザーストーリーは、システムやソフトウェアのユーザや購入者にとって価値のある機能を説明するもの

- (理想)Storyは互いに独立している。Storyはどのような順序でも開発できるように記述する。
- ストーリーの詳細は、ユーザーと開発者の間で交渉される。
- ストーリーは、ユーザまたは顧客に対する価値が明確になるように書く
- ストーリーに注釈を付ける最も良い方法の1つは、ストーリーのテストケースを書くことである。
- ストーリーはテスト可能でなければならない。

『User Stories Applied』

いずれの場合もデリバリ基準が重要

- 予測型/単一デリバリの場合
  - 受け入れ基準/完了基準を定める
    - リリースの日程
    - 機能セット
    - 欠陥の少なさ
    - 品質特性
- 適応型/定期デリバリの場合
  - 各ストーリーのdoneの定義を定める

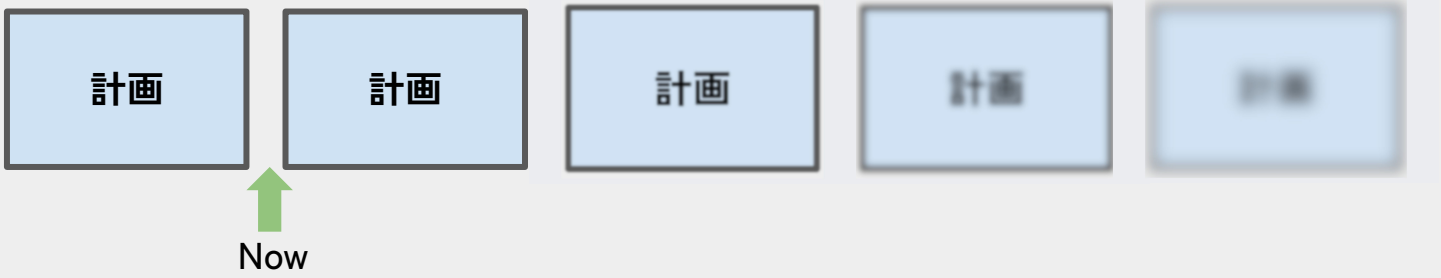
スコープが受ける不確実性の影響

- Doneドリフト
  - 適応型計画の元では、プロジェクトのDoneの定義が動き続ける。
  - イテレーション毎のストーリーはDoneをドリフトさせないようにコントロールしなければならない。
- スコープクリープ
  - 予測型計画の元では、対応するスケジュール、予算、リソースの必要性を調整することなく、追加のスコープや要件を受け入れてしまうことがある。
  - 受け入れる前に、必ず変更管理を行い、必ず時間バッファ、予算バッファへの影響を見積もり評価する。

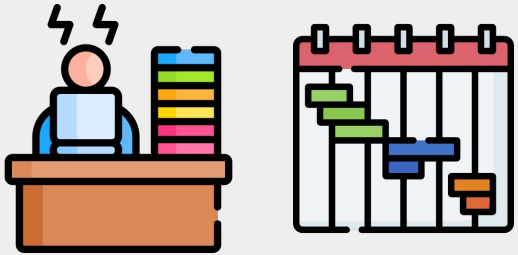
『PMBOK 7』

ローリングウェーブ計画法

- 遠い未来はざっくり計画しておいて、近い未来は詳細に計画する
- 必然的に、時間の経過にともなって計画の詳細化を繰り返す



予測型であっても、遠い未来のことを詳細までは計画してはならない。
大切なマネージャのリソースを、再見積・再スケジュールにばかり使ってしまうことになる。

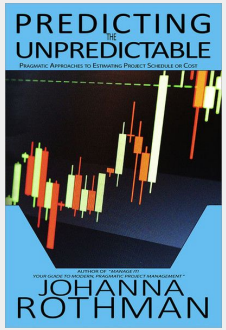


見積

プロジェクトのコスト、リソース、労力、期間を定量的に評価する

何のために?

- プロジェクトに関するサイズ/コスト/日付の「桁」がどれくらいかを把握する。
- もうすぐ終わるから、いつ終わるか知りたい。
- ある程度の期間、お金や人のチームを割り当てる必要がある。
- 誰が、誰を責任を求めるべきか知りたい。



https://www.amazon.co.jp/dp/1943487006

https://x.com/yosuke_furukawa/status/1605383031413698561

絶対見積と相対見積

- 絶対見積
  - 具体的な時間、コストを割り当てる
  - 誰がやるかによって変動性が少ないなら、これでも良いが...
- 相対見積
  - 基準タスクとのサイズの違いを相対的に割り当てる

SWAGとプランニングポーカー

経験に基づき見積ができない場合は、SWAG(科学的な野蛮な推測)を使う。

- 実際タスクを実行するチームメンバで見積もりをおこなう。
- 楽観的、可能性が最も高い、悲観的の3点見積を使うこともある。
- または、プランニングポーカーを使い、チームの合意する数値を導く。
  - プランニングポーカーにはメンバ間のストーリー、タスクに関する認識のずれを検出する目的もある。

プランニングポーカー https://en.wikipedia.org/wiki/Planning_poker

見積: DeterministicかProbabilisticか?

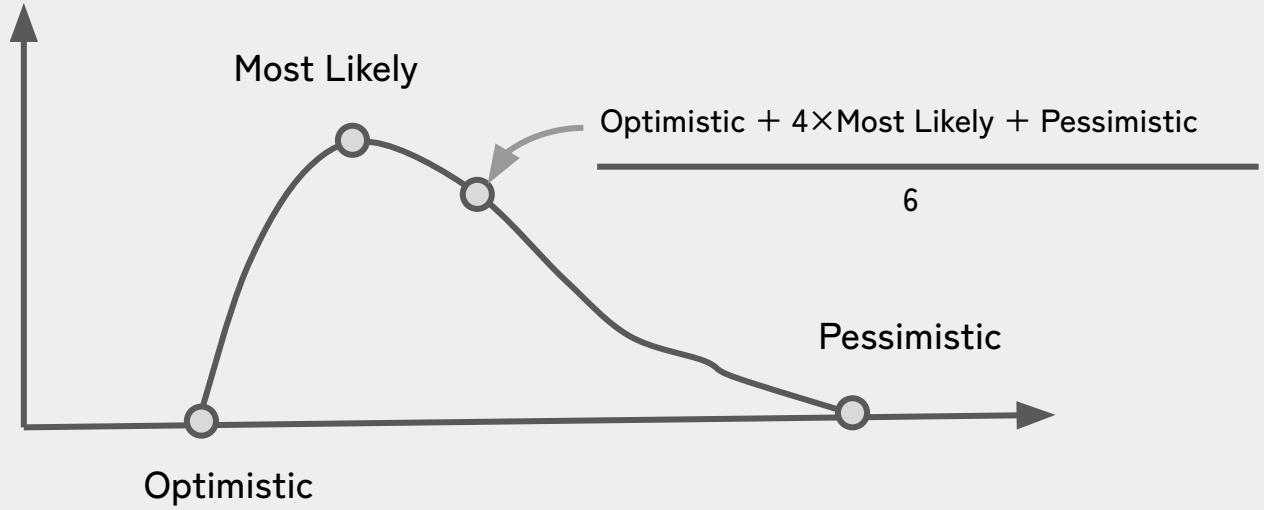
- Deterministic (決定論的)
  - コミットメントに使う
  - 不確実性分のバッファを含める
- Probabilistic (確率的)
  - 計画に使う

AccuracyとPrecisionの違い

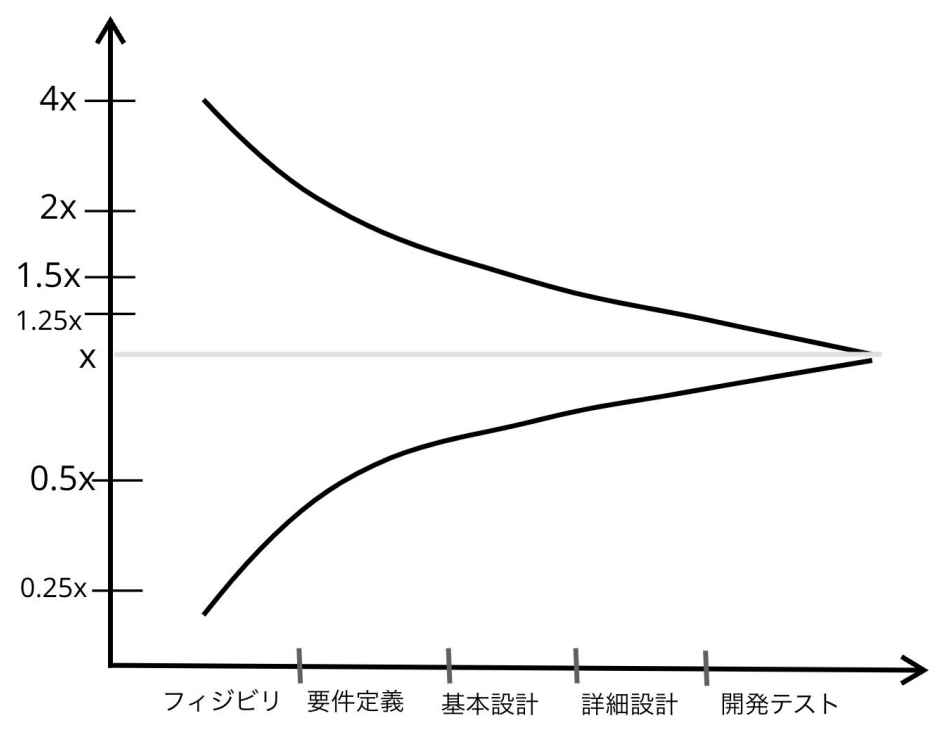
- Accuracyが高くないのにPrecisionを高めても無意味
- Accuracyを上げるためにできること
  - インチペブル
    - 見積もりたいタスク群に似たサイズの小さなタスクを1日〜2日で実際に実行してみる
  - Yesterday's Weather
    - 過去のタスク実績を記録しておいて、見積りたいタスクに類似の記録を探し、その実績値を見積とする

3点見積

- 楽観的見積: 最も良好な条件下での最小の見積 (Optimistic)
- 最も可能性の高い見積: 最もありえそうな見積 (Most Likely)
- 悲観的見積: 最も好ましくない条件下での見積 (Pessimistic)



不確実性のコーン



- 見積は「推測」であり不確実性を多分に含む
- プロジェクトの初期段階であればあるほど、不確実性は大きい

Reducing estimation uncertainty with continuous assessment: tracking the "cone of uncertainty"

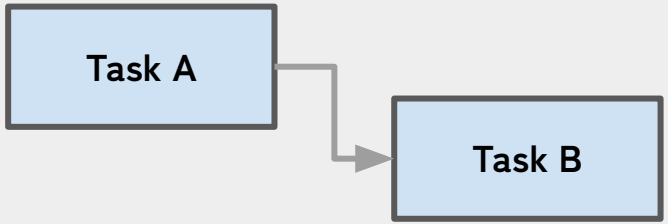
スケジューリング

予測型手法でのスケジューリング

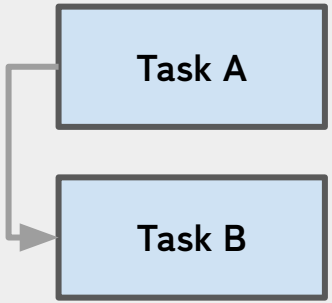
1. 具体的なタスクに分解する。(See. WBS)
2. 関連するタスクを順番に並べる。
3. タスクを完了するために必要な労力、期間、人、物理的リソースを見積もる。(See. 見積)
4. 利用可能性に基づいて、人員と資源を活動に割り当てる。
5. 合意されたスケジュールが達成されるまで、順序、見積もり、リソースを調整する。

タスク間の関係性

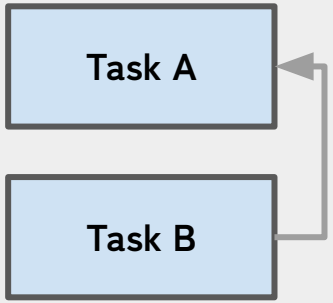
Finish-to-Start



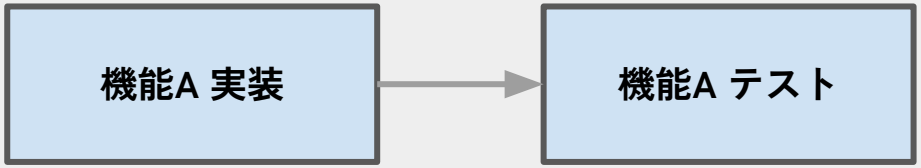
Start-to-Start

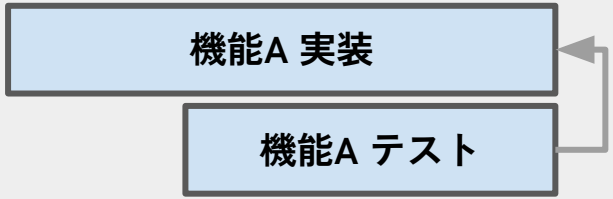


Finish-to-Finish



実はFinish-to-Startでないもの



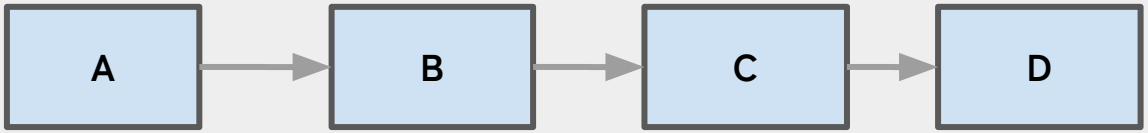


完了基準となるタスクはFinish-to-Finishでスケジューリングする。

見積が変わらない限り、スケジュール短縮されるものではないが、手戻りによるリスケを減らせる

演習問題: スケジューリングとタスクの分解

以下のスケジュールを短縮できるでしょうか?



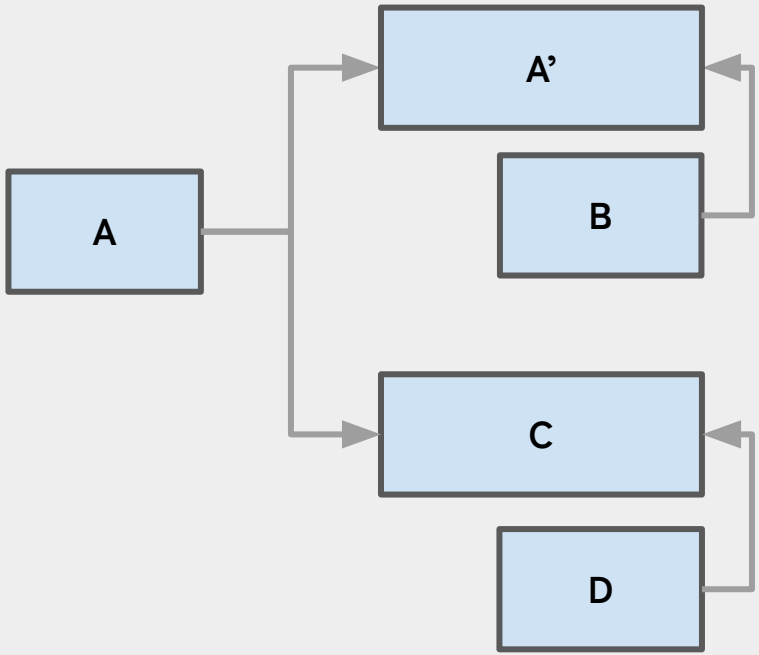
A: 認証モジュールを作る (見積: 3日)

B: 認証モジュールをテストする (見積: 2日)

C: 認証ライブラリが返すユーザオブジェクトを使って、検索するプログラムを作る (見積: 3日)

D: 検索機能をテストする (見積: 2日)

解答例



A: 認証モジュールインタフェースを作る (見積: 1日)

A’: 認証モジュールを作る(見積: 2日)

B: 認証モジュールをテストする (見積: 2日)

C: 認証ライブラリが返すユーザオブジェクトを使って、
検索するプログラムを作る (見積: 3日)

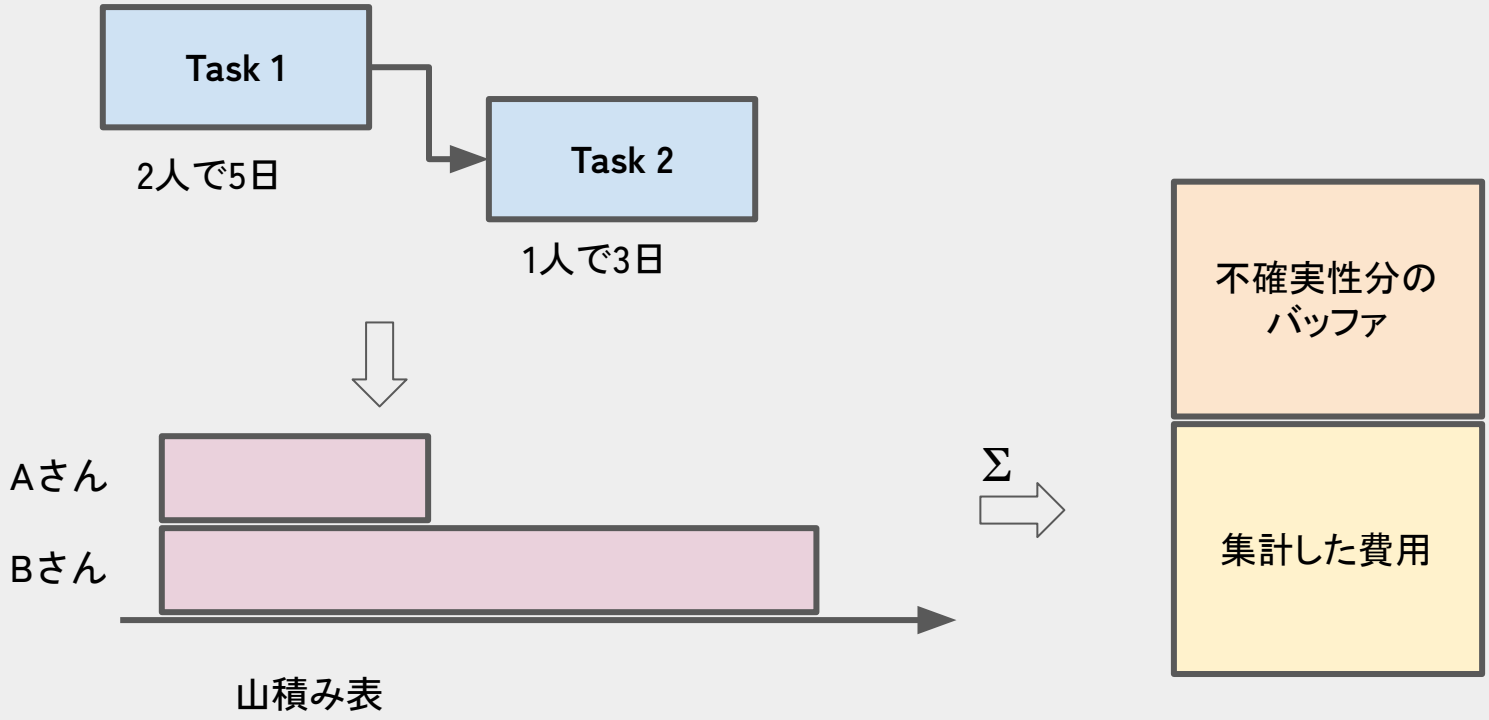
D: 検索機能をテストする (見積: 2日)

スケジュールの圧縮

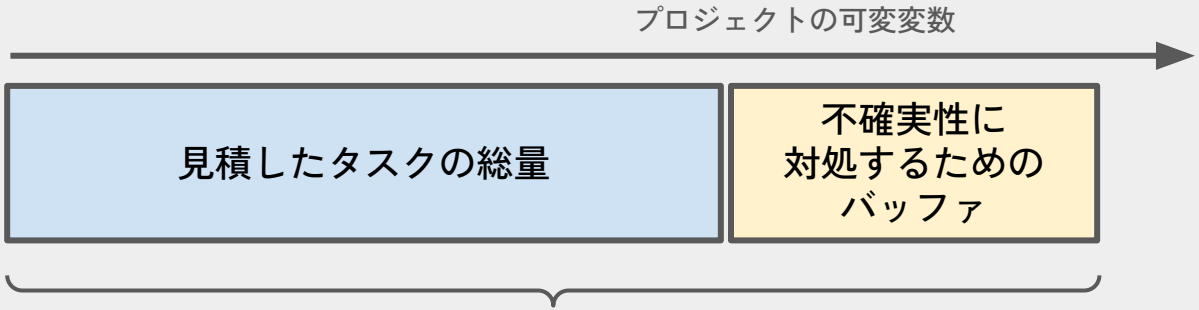
- Crashing
  - 重要なタスクにリソースを追加する
- Fast Tracking
  - クリティカルパス上の作業をオーバーラップさせる

どちらも危険な香りがプンプンするが、元々の計画が甘ければ
上手くいくこともある...

見積、スケジュールから予算への変換



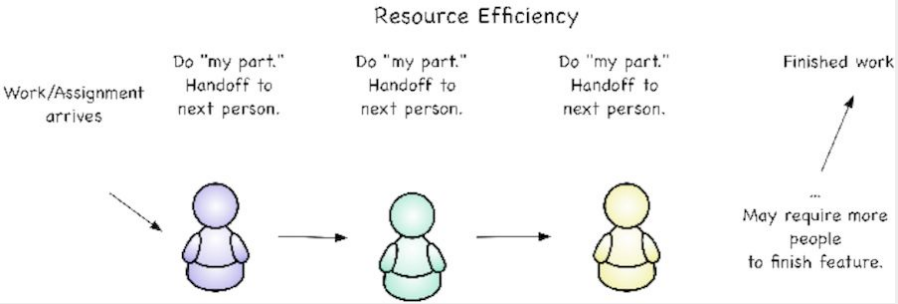
ここまでのまとめ

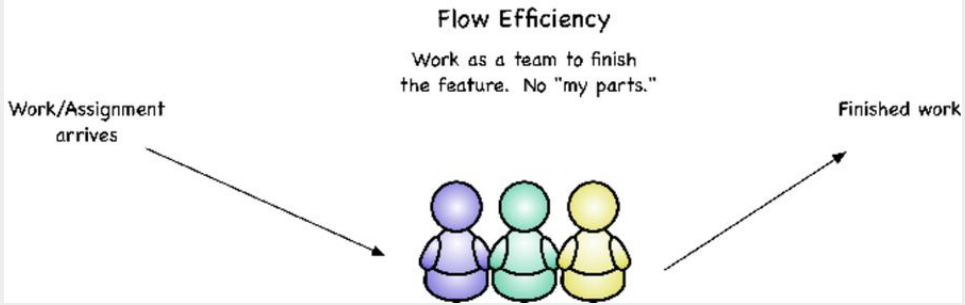


開発したものをどうデリバリしていくの?

リソース効率とフロー効率

- リソース効率
  - 各人の稼働率をあげることを目的にする
- フロー効率
  - タスクのリードタイムを短くすることを目的にする

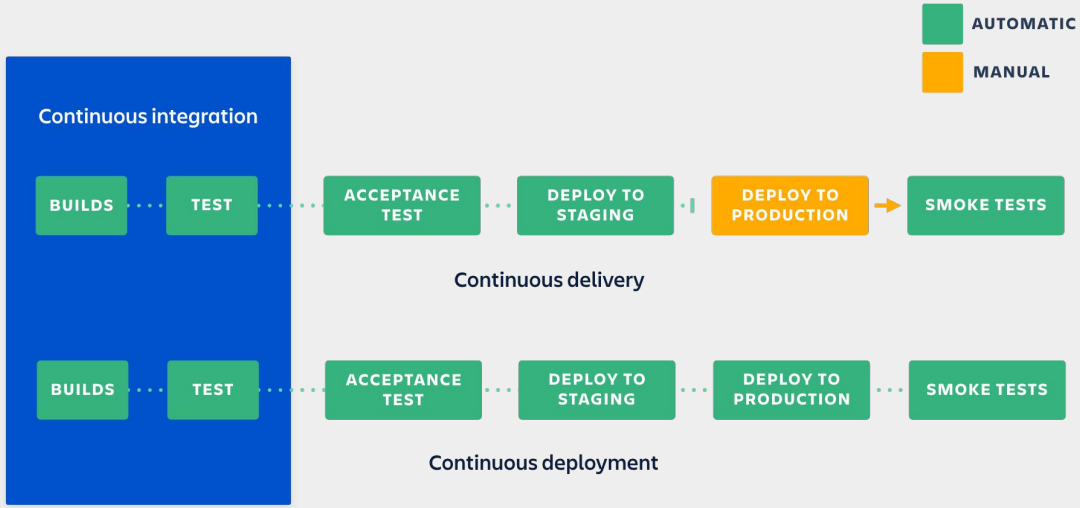




https://www.slideshare.net/i2key/xpjug

デリバリとデプロイ

- デリバリ
  - 本番リリース一歩手前まで持っていくこと(ビジネスサイドにデプロイ判断は委ねる)
- デプロイ
  - 本番リリースする



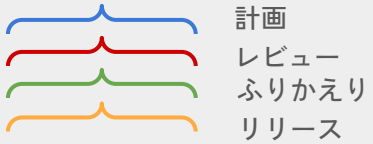
https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

デリバリーケイデンス

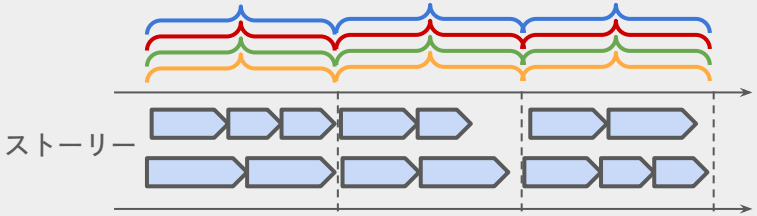
- 単一デリバリー: プロジェクトの終わりに1回だけデリバリを行う
- 複数デリバリー: プロジェクト期間中の異なる時期に複数回のデリバリを行う。
- 定期デリバリー: 固定されたデリバリースケジュールで行う

ケイデンスとリズム

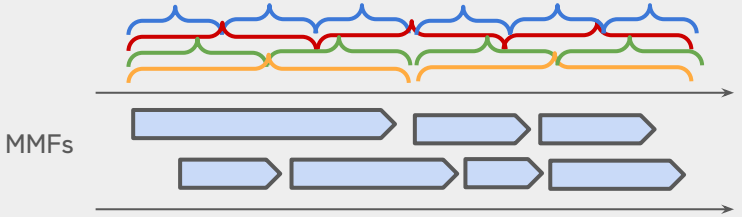
ケイデンスは周期性がより強調されたもの



タイムボックスベース

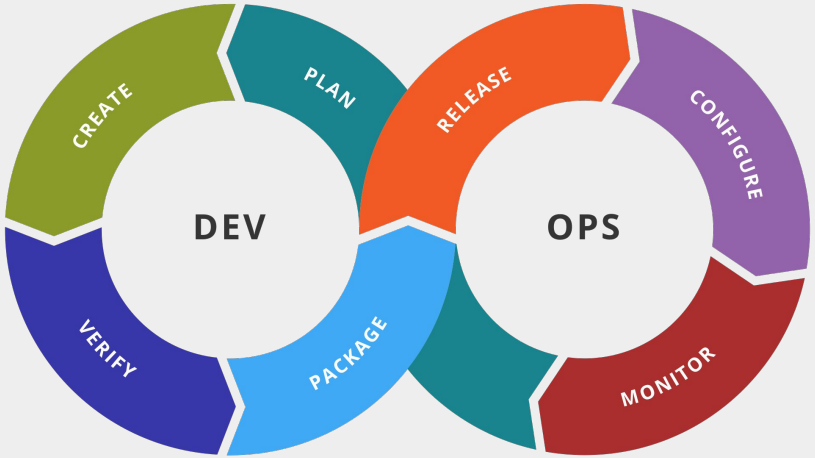


カンバン



https://availagility.co.uk/2009/07/21/what-is-cadence/

DevOps

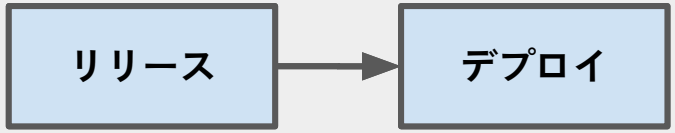


開発(Dev)と運用(Ops)が一連の開発ライフサイクルを持つこと。
(必ずしも一体組織である必要はない)

デリバリーケイデンスが単一でないことが暗黙の前提になっている。

https://ja.wikipedia.org/wiki/DevOps

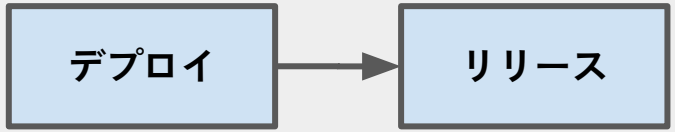
デプロイとリリースの逆転



機能完成
(実際の使用に耐えうると判定)

本番環境に配置し、ユーザが使えるようになる

デメリット:
開発者がデプロイ後の問題に備えて待機しておかなければならない。



本番環境に配置

実際の使用に耐えうると判定
ユーザが使えるようになる

メリット:
開発者が立ち会う必要がなく、ビジネスサイドの好きなタイミングで機能をユーザに開放できる。

デメリット:
デプロイした機能を一部のユーザや端末だけに開放する仕組み(フィーチャーフラグ)が必要で複雑化しやすい。
データベーススキーマの変更を伴うものに適用するのは工夫とノウハウが必要。

プロジェクト運営のOption

開発ライフサイクル	予測型	適応型
プロジェクト変数	コスト可変	スコープ可変
デリバリーケイデンス	単一デリバリ	定期デリバリ 複数デリバリ
最適化の優先軸	リソース効率	フロー効率

リソース効率に潜む問題

- コンテキストスイッチによって
- 1人1タスク以上持つことが多いので、必然的にチーム内のコラボレーションが発生しにくくなる
- タスク割当のための労力が大きい (マネジメントの工数が大きくなる)

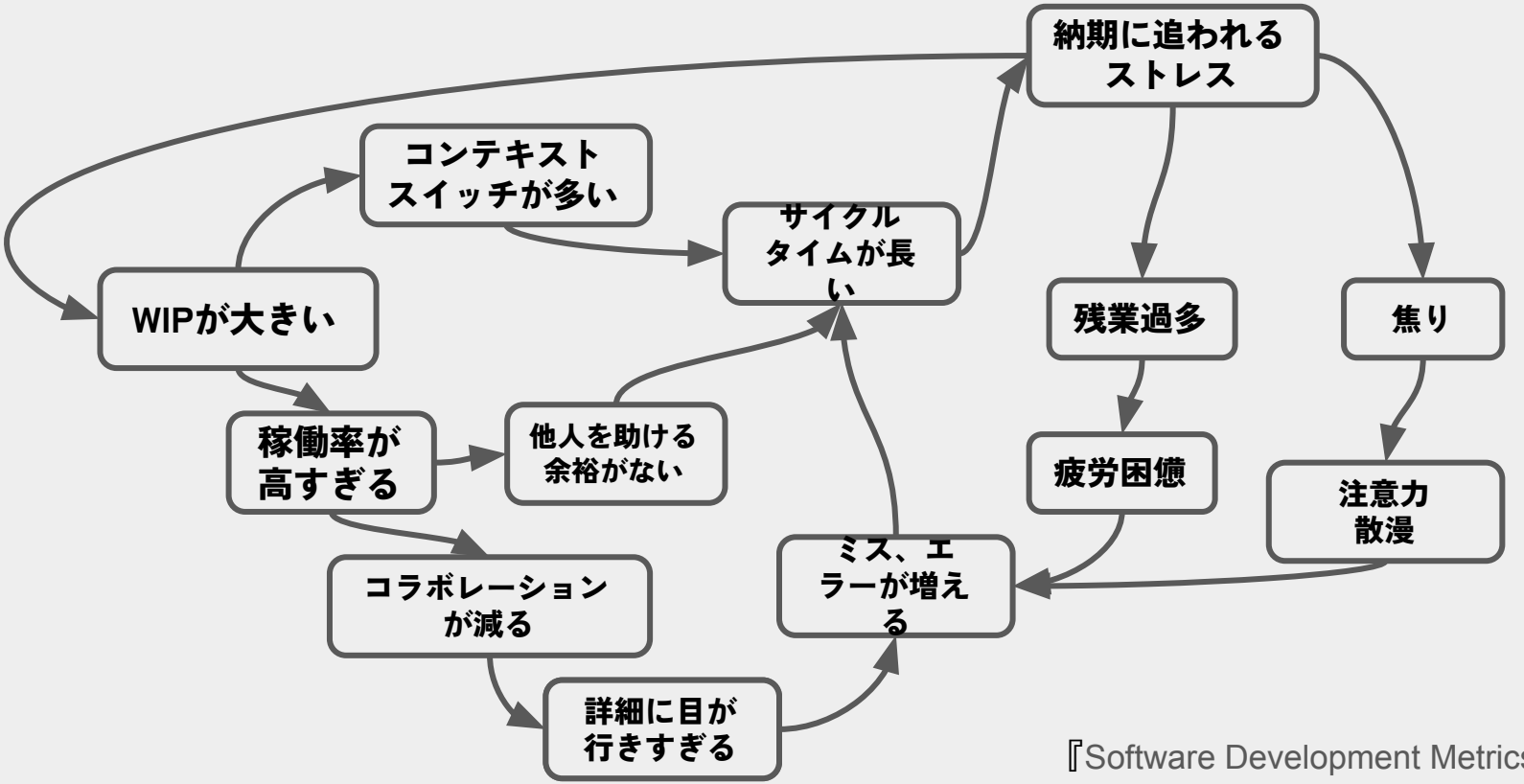
WIP(Work In Progress)

フロー効率においては、チームが一度にどれだけのタスクを扱うか? が重要。

→ WIPが多すぎると、フロー効率が下がる

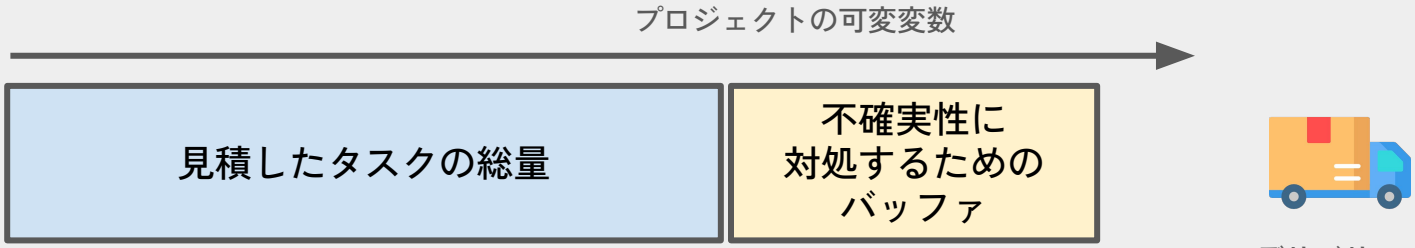
アジャイル(というかカンバン)におけるマネジメントのベースは、
WIPの上限を定めて、デリバリを安定させること

大きすぎるWIPの問題



『Software Development Metrics』

ここまでのまとめ

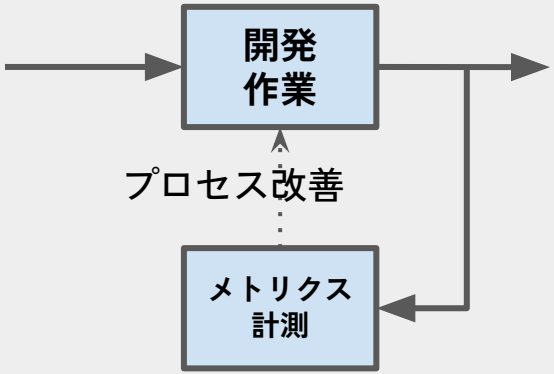


デリバリー

実際に不確実性をどうやってコントロールしていくの?

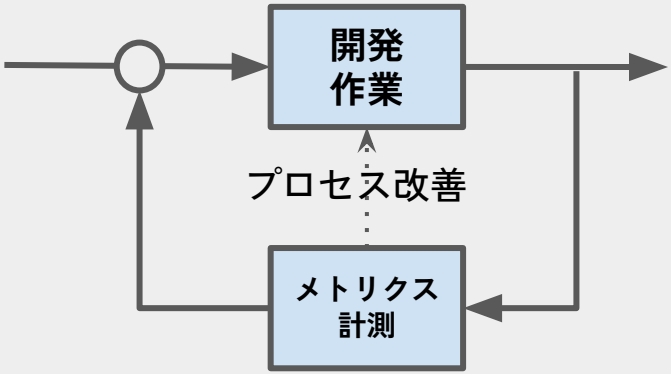
メトリクス活用の違い

予測型手法



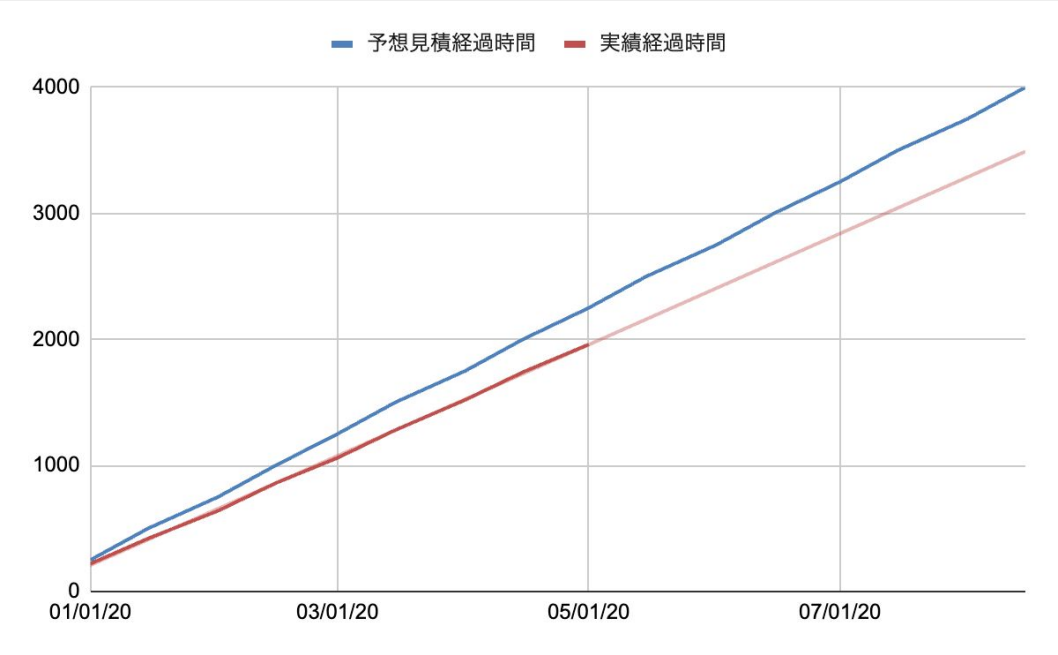
目標と実績のズレを計測しプロセス改善する

適応型手法



動く目標値に対して、計測したメトリクスを
元にイテレーション計画を調整する

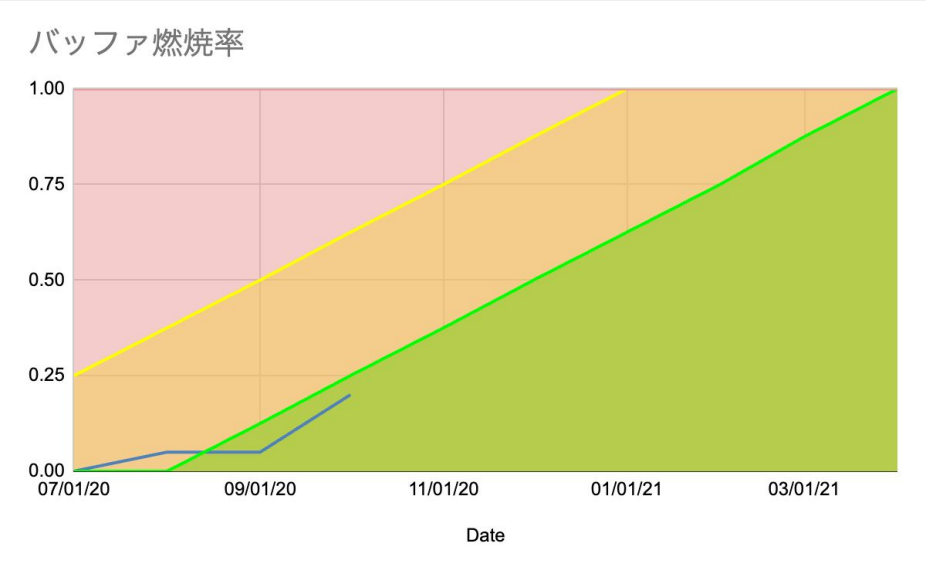
スコープ完了率



見積時間orストーリーポイントがどれだけ完了したかを時系列でトラッキングする。

トレンドラインを見れば、着地点を推測できるので、対策を打つ。

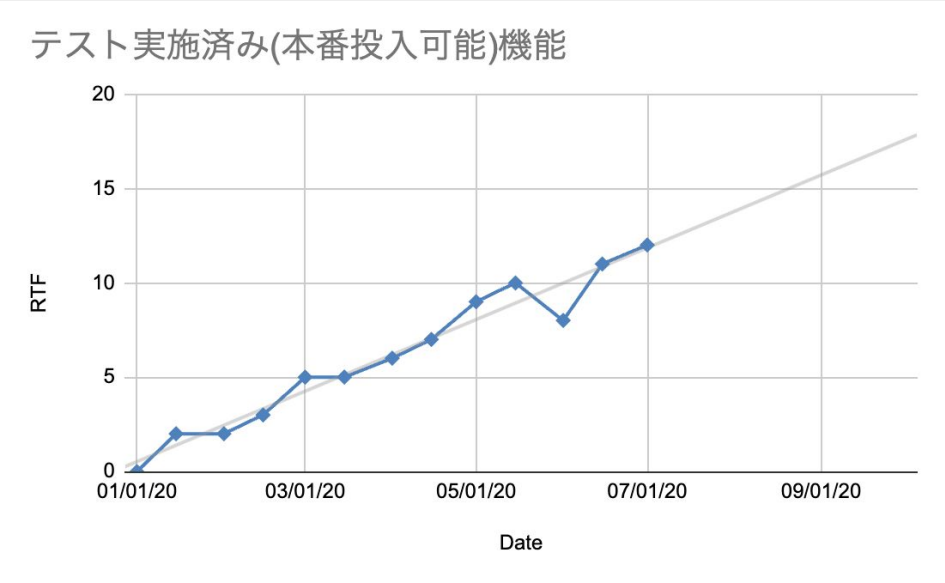
バッファ燃焼率



バッファを使い切らなければ、デリバリの遅れ、計画コストの超過は免れる。

『Software Development Metrics』

テスト実施済み(本番投入可能)機能



適応型プロセスにおいて、本番投入可能な機能をイテレーション毎に作れているかを検証する。

テストは回帰的に実行するので、あるフィーチャーの実装で、本番投入可能なものを壊してしまったことも検出する。

ベロシティ

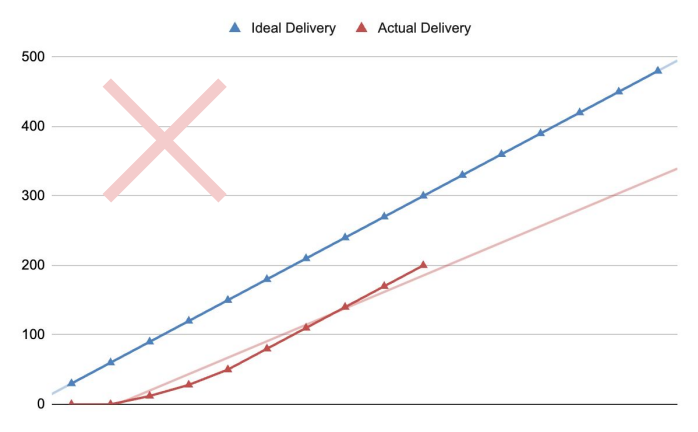
	A	B	C	D
1	イテレーション	ベロシティ	累計	スコープ
2	1	4	4	800
3	2	12	16	800
4	3	19	35	800
5	4	26	61	800
6	5	30	91	800
7	6	30	121	800
8	7	29	150	800
9	8	32	182	800
10	9	30	212	800
11	10	31	243	800
12	11	29	272	800
13	12	30	302	800
14	13	30	332	800
15	14			800
16	15			800
17	16			800
18	17			800
19	18			800
20	19			800
21	20			800

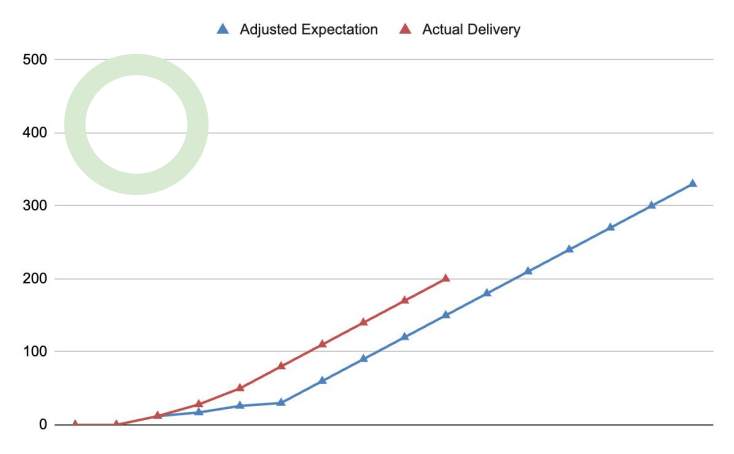
タイムボックスを使うプロジェクトにおいて、次のタイムボックスの出来高を推測するのに使う。

ベロシティを目標設定に使ったり、チーム間の生産性比較に使ったりしてはならない。

Ramp up

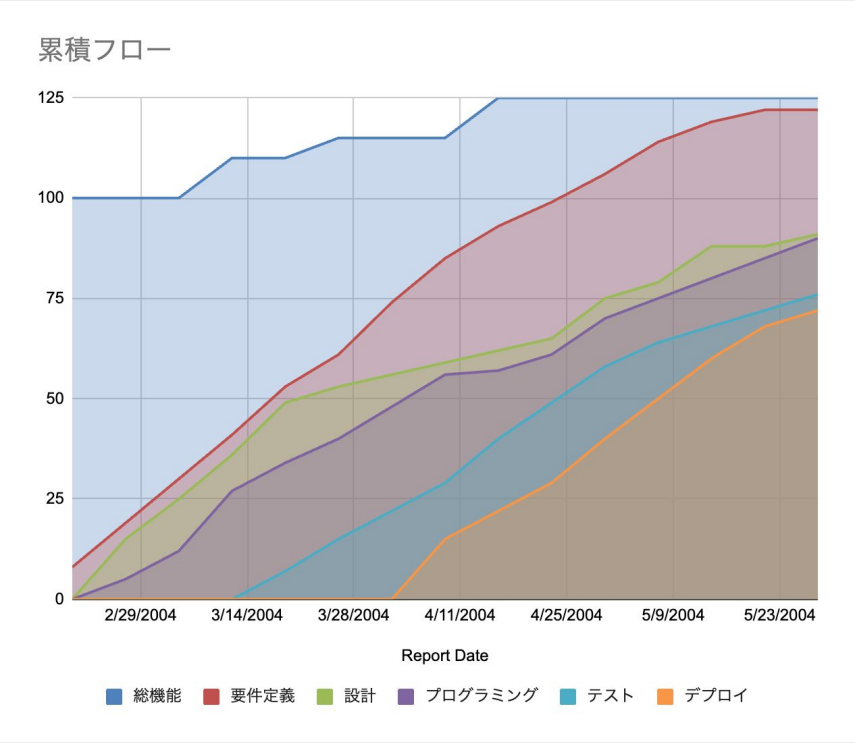
ベロシティは開発の初めの数イテレーションは最大値が出ない。
これを踏まえて計画する。





このRamp up期間は予測型開発においても同じ。
スケジューリングの際に考慮すべし。

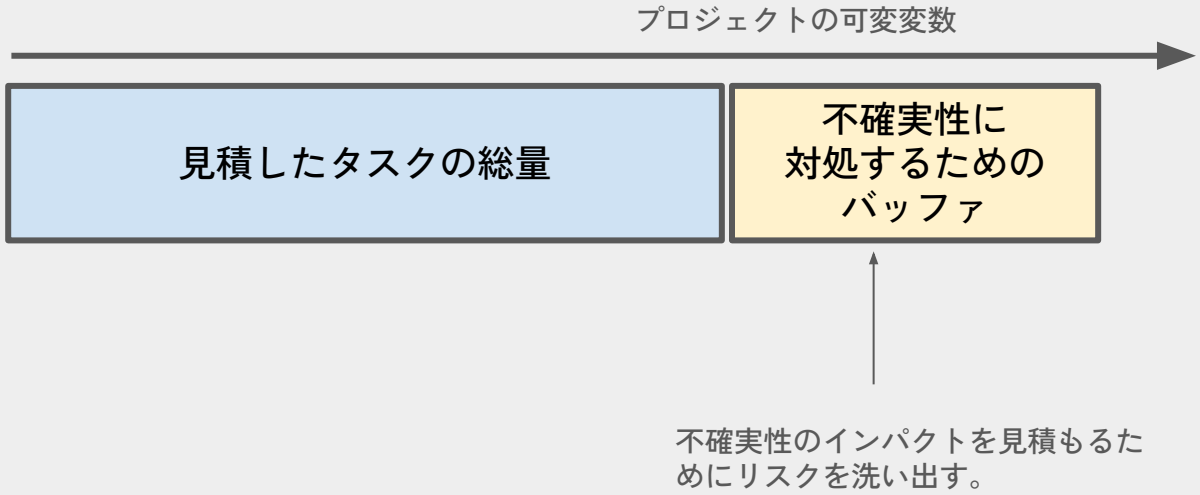
累積フロー



どの作業にどれくらい時間がかかっているか?

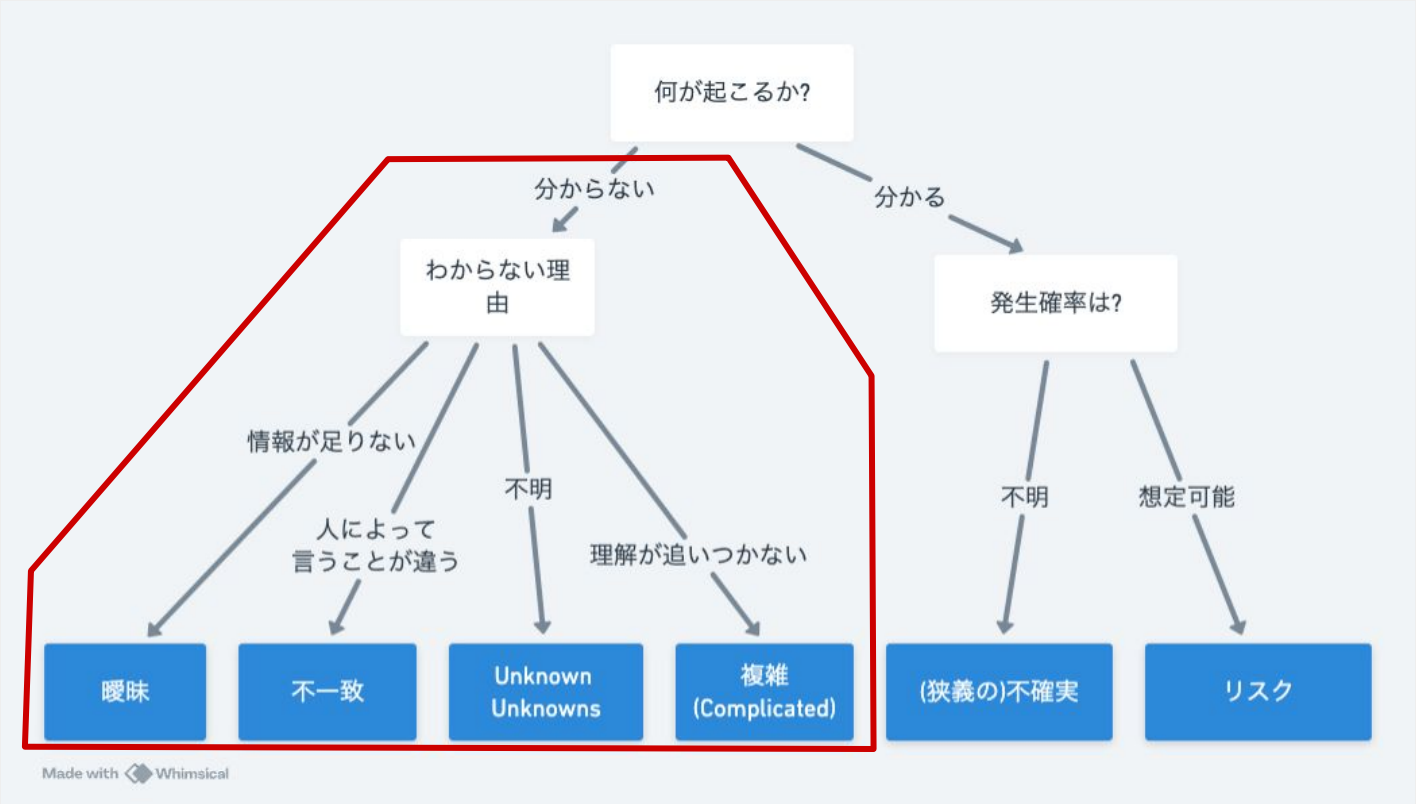
ボトルネックを探すために使う。

ここまでのまとめ

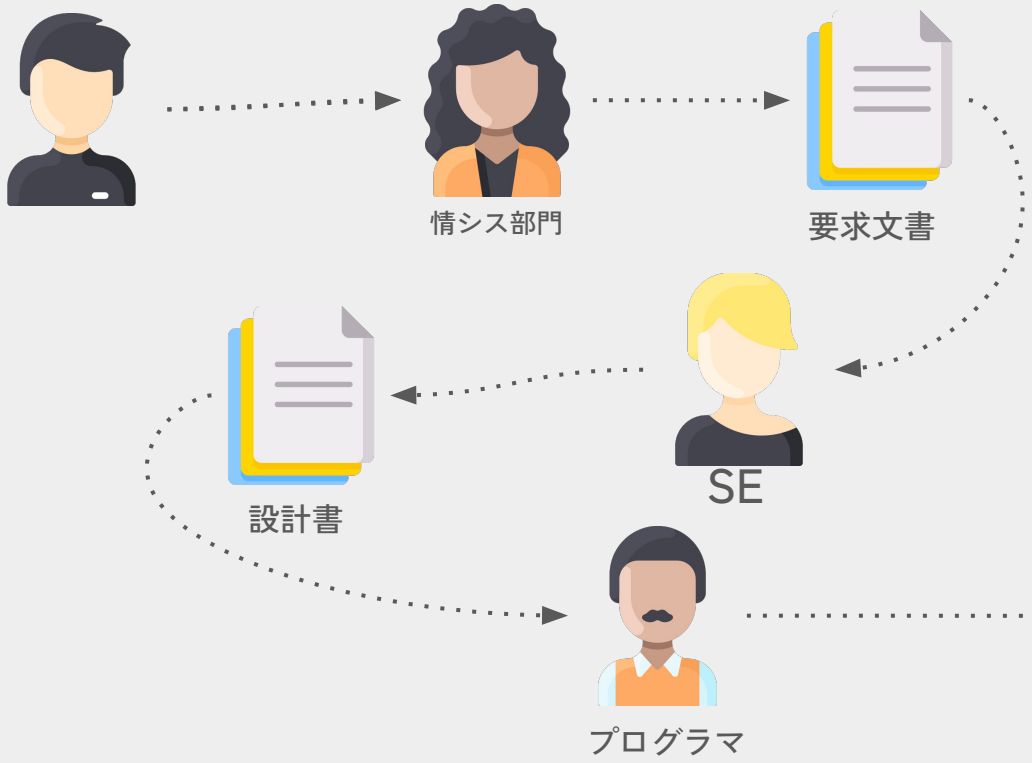


不確実性にはリスク(および狭義の不確実性)以外も含まれているのでは?

何が起きるか分からない領域もコントロールしなければならない



不一致

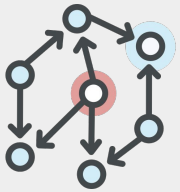


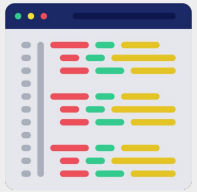
昔ながらの責務分担だと、伝言ゲームになりやすい。

不一致

共有メンタルモデル

全ステークホルダーおよびコードが共通のモデルを参照する。

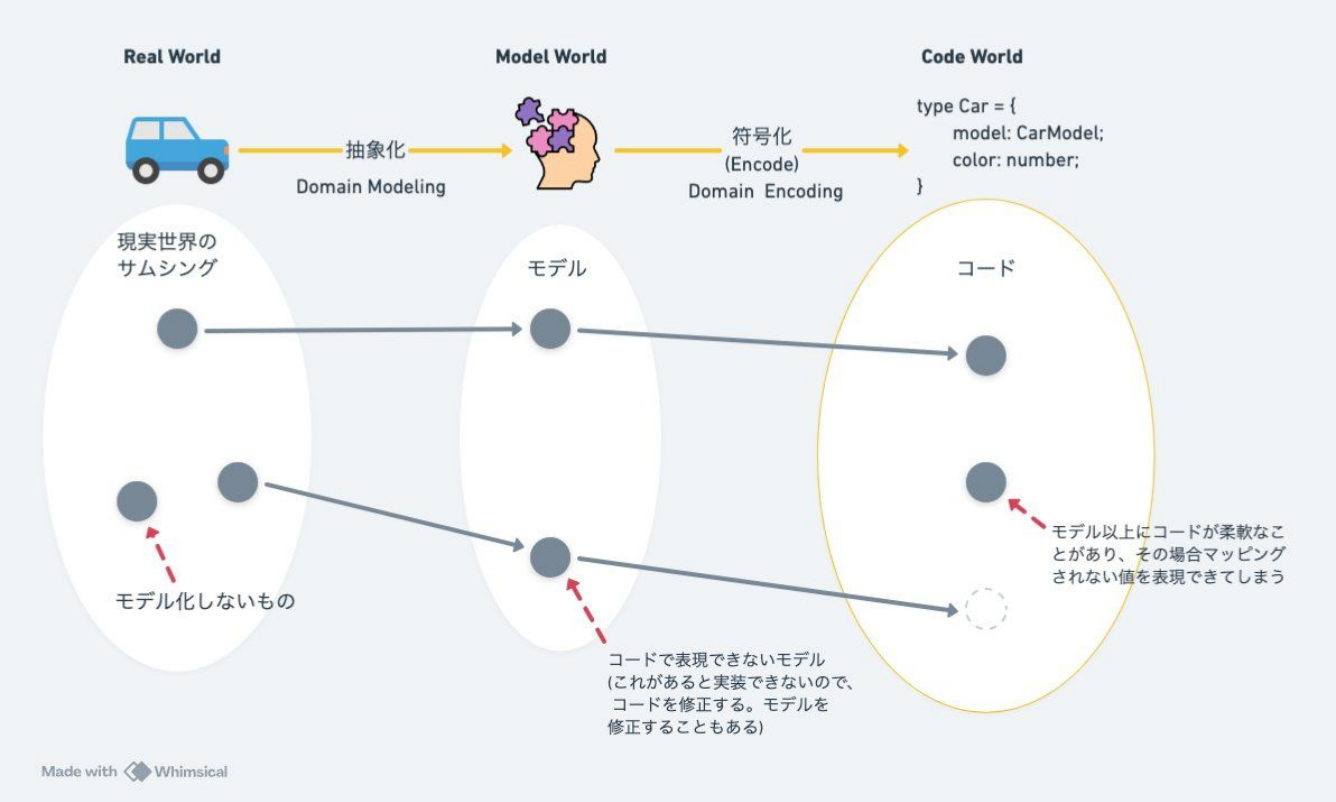




理想論では...?

不一致

共有メンタルモデルには実装の都合が入らないようにしたい



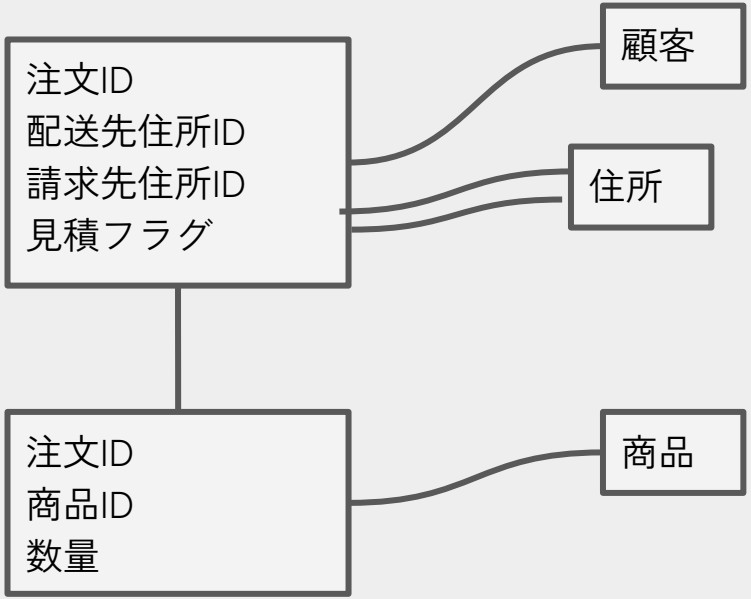
不一致

モデリングはコミュニケーション手段

https://unit8.net/maac

不一致

モデルをER図で表すと...



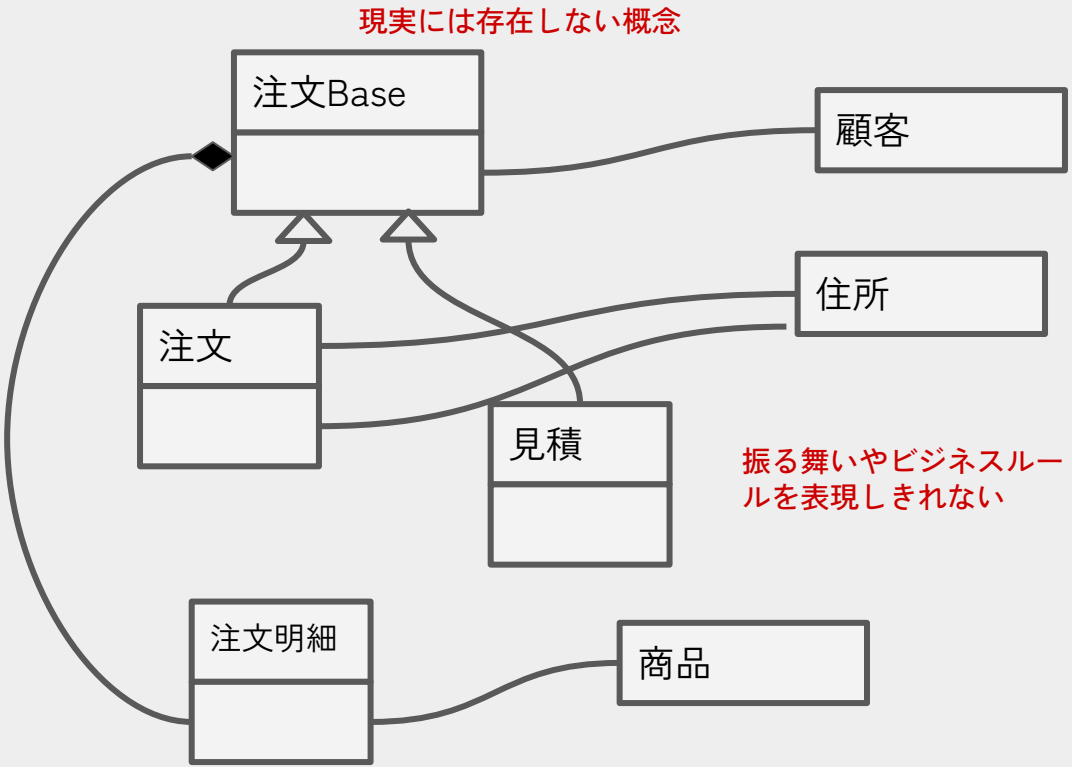
振る舞いが考慮されない

永続化しない概念は表されない

『Domain Modeling Made Functional』

不一致

モデルをクラス図で表すと...



『Domain Modeling Made Functional』

不一致

ドメイン記述ミニ言語

DSLで書くと良い(が、標準的なものが存在しない)

```
data 注文 =
  顧客情報
  AND 配送先住所
  AND 請求先住所
  AND 注文明細のリスト
  AND 金額総計

data 注文明細 =
  製品コード
  AND 注文数量
  AND 価格

data 顧客情報 = ??? // 現時点で詳細不明
data 請求先住所 = ??? // 現時点で詳細不明

data 注文数量 = 個数 OR キログラム
data 個数 = 整数 (1 ～ ?)
data キログラム = 数値 (? ～ ?)
```

事業構造の持つ複雑さと、それをどう解釈しソフトウェア上に表現するかに集中する。

https://scrapbox.io/kawasima/ドメイン記述ミニ言語

複雑さ & Unknown Unknowns

「分かりにくい」「分からない」が不測の事態を引き起こす。

「分からない」とは何か...?

複雑とは何か...?

分からないのレベル

「わからない」のレベル The Five Orders of Ignorance

- 0OI: 全部分かっている
  - 「答え」を持っている。あとは実装するだけで完成する。
- 1OI: 分からないことが分かっている
  - 答えを得るための「質問」を持っている。
- 2OI: 分からないことが分からない
  - 「質問」を持たない状態。決定的な答えを引き出すための「質問」ができない。
- 3OI: 分からないことが分からない状況を何とかする術を知らない
  - 2OI→1OI→0OIと進んでいくためのプロセスがない状態。
- 4OI: 無知にレベルがあることを知らない

Cynefin Framework

適応型

予測型でも良いが...

予測型

複雑さ(Complex & Complicated)

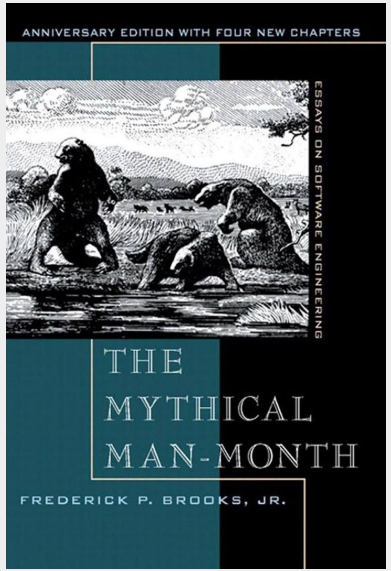
レベルが2つあるのに加えて、
その性質から次の2つに分類できる。

本質的

ソフトウェア開発に本来的に
備わっているもの

偶有的

存在するが必須でないもの



『人月の神話』Ch.16 銀の弾丸はない
https://www.amazon.co.jp/dp/0201835959

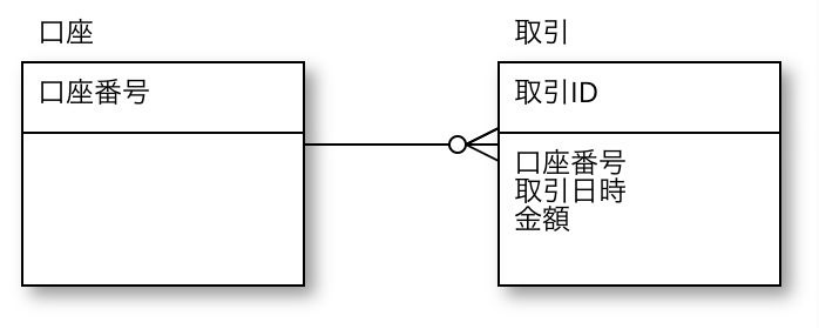
偶有的複雑さ

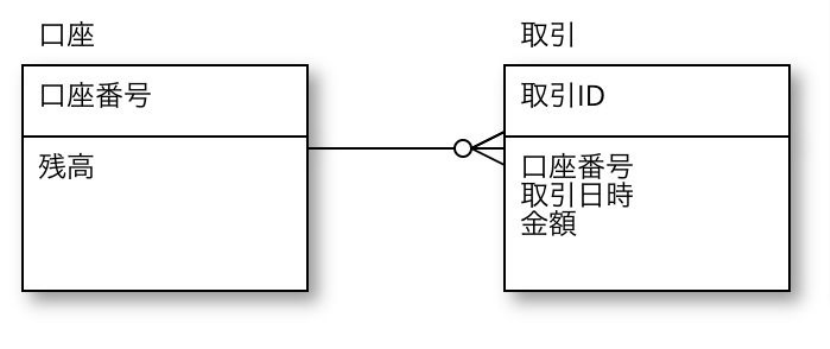
究極理想環境では無くなるもの

- どんな検索も一瞬で返すマシンや無限のメモリ空間
- 絶対に判断ミスをしない設計者

現実にはそういうものが存在しないので、トレードオフを考えつつ意思決定(偶有的複雑さを抱える)ことを決めていく。

偶有的複雑さの例①





取引を集計しさえすれば、現在の残高は求まるので、口座エンティティには「残高」は不要。

...なのだが、実際はその算出にそれなりの計算量が必要なので、口座に「残高」を持たせる。

そうした途端に付随して、

- 残高更新時の排他制御
- 取引の集計と残高のズレのチェック(リコンサイル)

などが発生し、さらなる複雑さを生む要因になる。

偶有的複雑さの例②

自分は②って書くのだが、もしかして少数派なのだろうか...🤔
どっちが好き？

```
const apartment = {
  metersToStation: 400,
  ageOfBuilding: 5,
  options: ["ShoeLocker", "AutoLock"],
}

// ①
const salesPoints1 = []
if (apartment.metersToStation < 1000) {
  salesPoints1.push("駅近")
}
if (apartment.ageOfBuilding < 5) {
  salesPoints1.push("築浅")
}
if (apartment.options.includes("AutoLock")) {
  salesPoints1.push("セキュリティ充実")
}

// ②
const tags2 = [
  apartment.metersToStation > 1000 && "駅近",
  apartment.ageOfBuilding < 5 && "築浅",
  apartment.options.includes("AutoLock") && "セキュリティ充実",
].filter(Boolean)
```

コントロールフロー

手続き型のプログラミングスタイルだとコード順に依存するので思わぬ問題に遭遇することがある。

https://twitter.com/t__keshi/status/1635267214008897537

偶有的複雑さの例③

純粋性＋完全性

理想論の悪魔
😈 「純粋かつ完全なるドメイン...美しい」
ただし、遅くて使い物にならないけどな

完全性＋性能

アンチドメインモデル貧血症の悪魔
😈 「業務ロジックを発見し、濃いドメインモデルができたで〜」
こんな簡単なものに、こんな複雑な仕掛けが必要なの?

純粋性＋性能

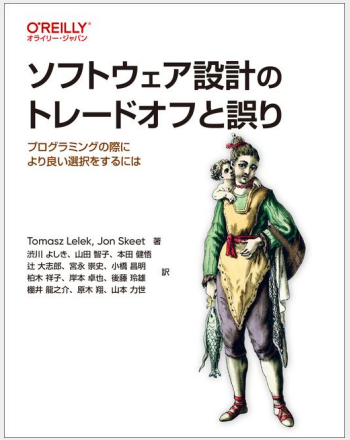
テスタビリティの悪魔
😈 「ドメイン層がピュアでテストしやすぅ〜」
ドメイン層スッカスカでテストが簡単に書けても、ユースケースからテスト通さないと、ほとんど品質保証の意味をなさない。

https://www.slideshare.net/kawasima/ss-255489610

偶有的複雑さは技術的トレードオフで現れる

- 評価軸には組織のケイパビリティも含める
- 代替案の評価基準は時代とともに移りゆく

これ以上の偶有的複雑さに関する話は、本講座の範囲を超えるので、こちらの本を参照ください。



https://www.oreilly.co.jp/books/9784814400317/

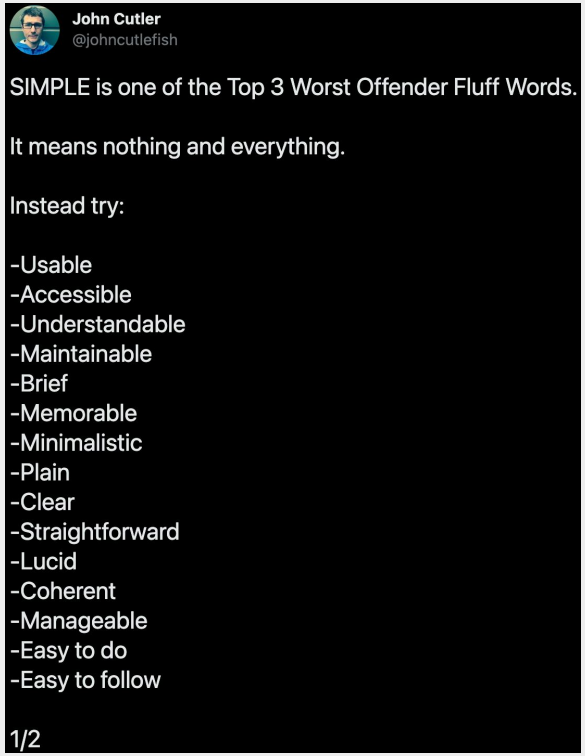
本質的複雑さ

業務要求が持つ複雑さそのもの。

ここに対する「銀の弾丸はない」とされている。

本質的複雑さとは何なのか? 対処のしようはないのか?

対義語から考える複雑



- 使いにくい
- 理解が難しい
- メンテしにくい
- 長ったらしい
- 覚えきれない
- 実行が難しい
- 管理できない
- 色々混ざっている
- たくさんの要素がある
- 一貫性がない
- 込み入ってる
- ゴテゴテしている
- 不透明

主観的なものとそうでないものが混ざっている

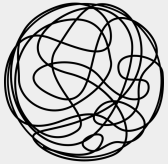
Simple Made Easy

SimpleとEasyは違う

主観的

自分にとって馴染みがある

- No → 難しい (Hard)
- Yes → 難しくはない (Easy)



対象

客観的

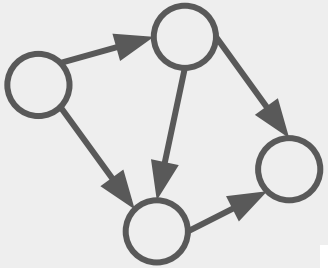
複数の概念や要素が組み合わさっている

- No → 単純 (Simple)
- Yes → 複雑 (Complex)

https://github.com/matthiasn/talk-transcripts/blob/master/Hickey_Rich/SimpleMadeEasy.md

Coupling

複雑さ(それによるメンテナンスコストの増大)はコンポーネントのインタラクションによって生じる

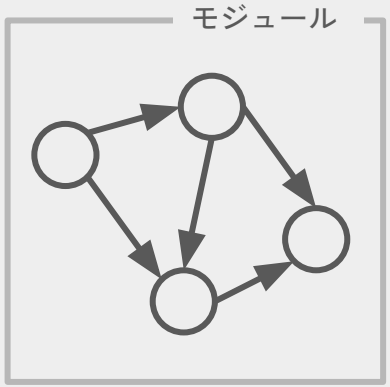


$$cost(software) \simeq cost(change) \simeq cost(big\ change) \simeq coupling$$

『Tidy First?』『Balancing Coupling』

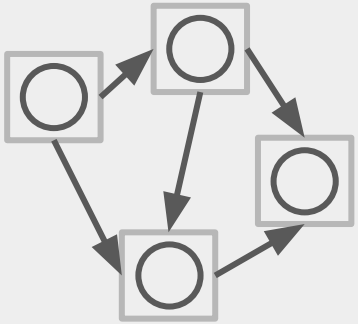
ローカルな複雑さとグローバルな複雑さ

ローカルな複雑さ



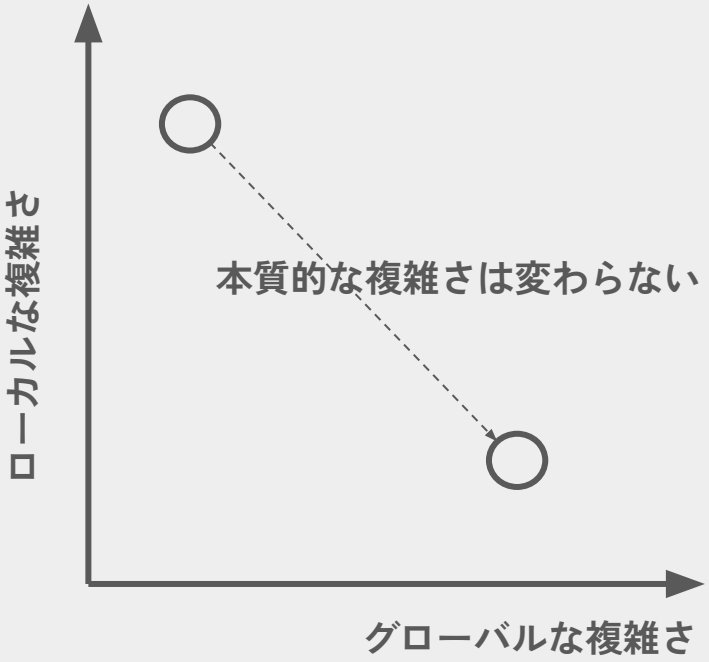
モジュール内に含まれるコンポーネントとそのインタラクションがもたらす複雑さ

グローバルな複雑さ



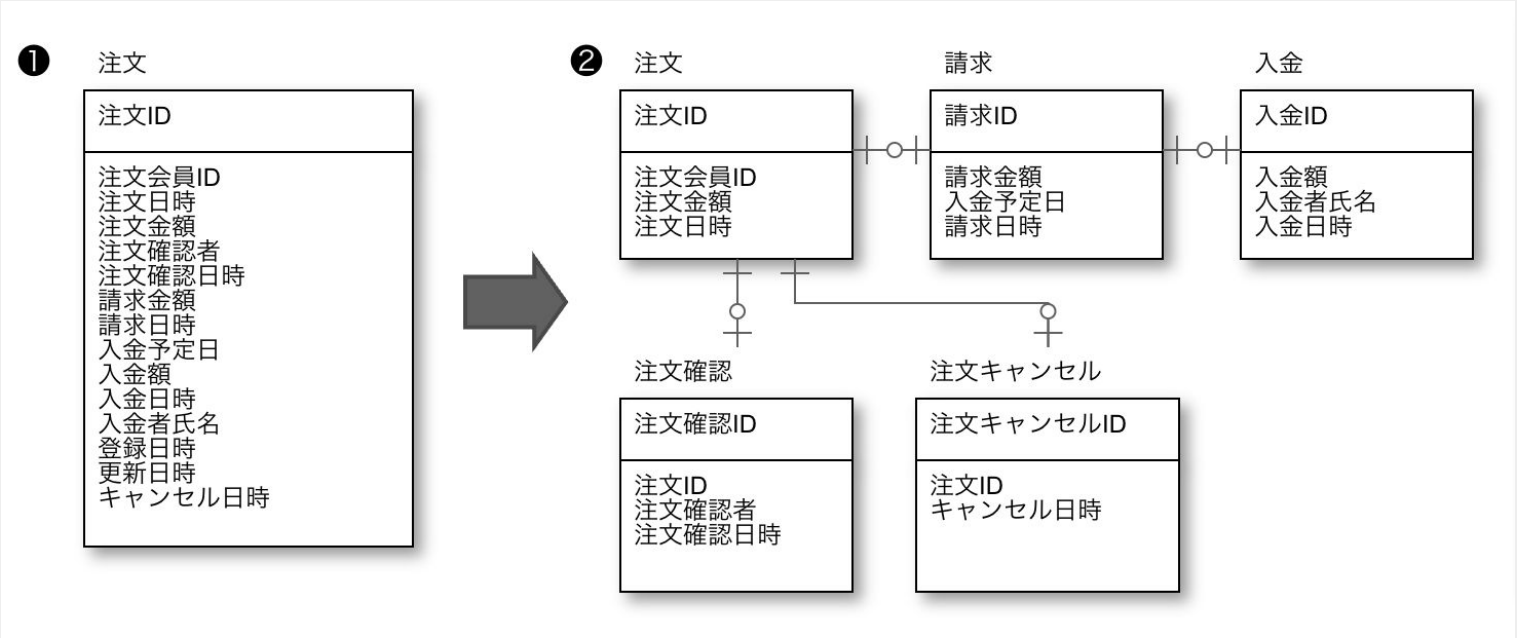
モジュールとそのインタラクションがもたらす複雑さ

本質的複雑さ保存則



ローカルな複雑さを持つコンポーネント、例えばGod Classを分割して、単一責務のクラスに再設計したとしても、ローカルな複雑さはグローバルな複雑さに転換されるだけで、本質的複雑さの総量は変わらない。

非対称な理解可能性



ローカルは複雑さは、それを説明する追加のドキュメントがないと、何が含まれているか分からない

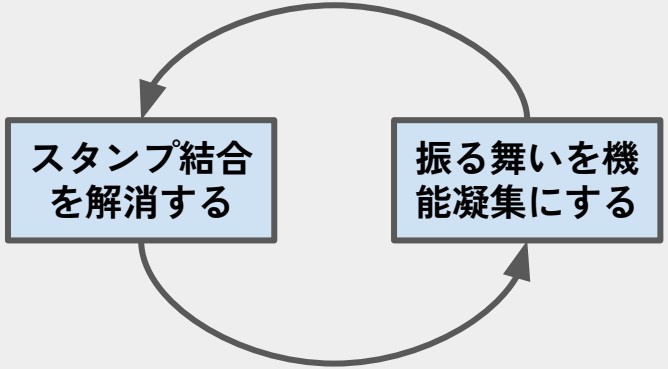
演習: ローカルな複雑さを見えるようにモデルを書いてみましょう

ある完全受注生産のECサイトでは、注文を受けた後、商品を手配し、手配が完了したら配送先住所と配送希望日を入力してもらい、その後配送処理を行います。このプロセスにおいて、注文の状態は「手配中」「配送処理中」「発送済み」の3つに分けられます。

配送処理を開始する際には、以下の点を確認する必要があります：

1. 配送先が「日本国内のみ」である商品が含まれている場合、その商品が国内の住所に配送されるかどうか。
2. 「土日配送不可」の商品が含まれている場合、指定された配送希望日が土日を避けているかどうか。

ローカルな複雑さの解消の仕方

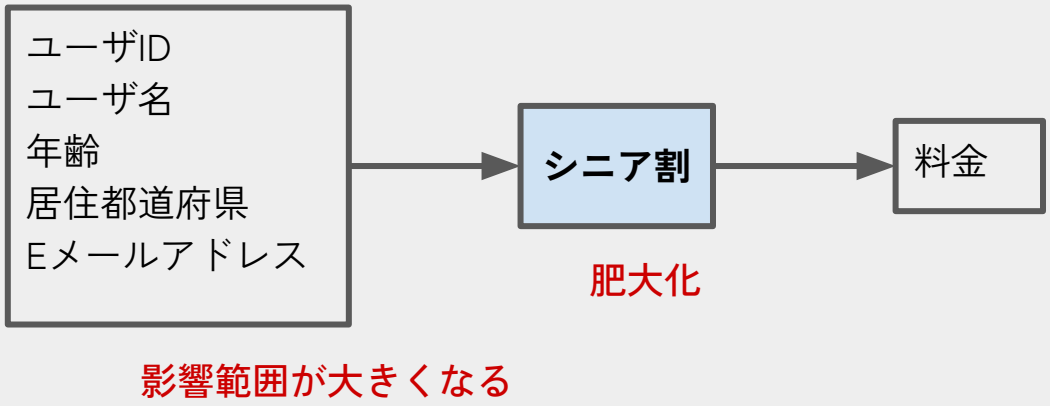


伝統の疎結合・高凝集

これをモデル世界で追求しておく。

スタンプ結合を解消する

ある振る舞いが、与えられたデータの一部しか使っていない

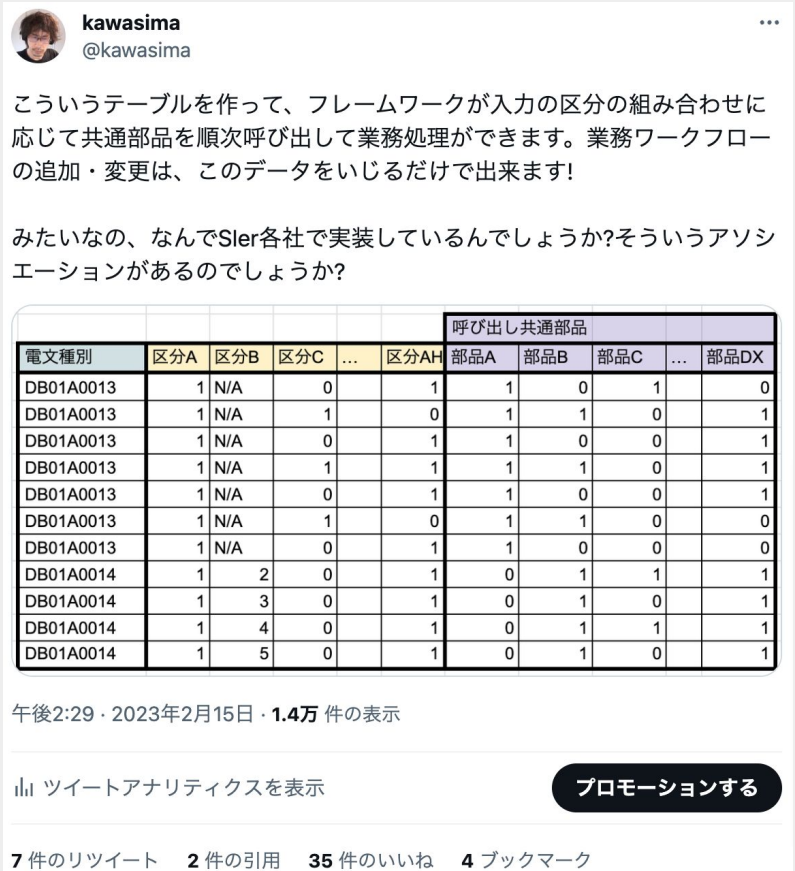


振る舞いの名前に着目し、振る舞いを分割する

振る舞い名前をHonest & CompleteにしてみてAndやOrで繋がなければならないなら、複数の責務が混在しているので分割する。

https://scrapbox.io/kawasima/命名のプロセス

本質的複雑さを明らかにしなかった末路①



kawasima
@kawasima

こういうテーブルを作って、フレームワークが入力の区分の組み合わせに応じて共通部品を順次呼び出して業務処理ができます。業務ワークフローの追加・変更は、このデータをいじるだけで出来ます!

みたいなの、なんでSIer各社で実装しているんでしょうか?そういうアソシエーションがあるのでしょうか?

						呼び出し共通部品				
電文種別	区分A	区分B	区分C	…	区分AH	部品A	部品B	部品C	…	部品DX
DB01A0013	1	N/A	0		1	1	0	1		0
DB01A0013	1	N/A	1		0	1	1	0		1
DB01A0013	1	N/A	0		1	1	0	0		1
DB01A0013	1	N/A	1		1	1	1	0		1
DB01A0013	1	N/A	0		1	1	0	0		1
DB01A0013	1	N/A	1		0	1	1	0		0
DB01A0013	1	N/A	0		1	1	0	0		0
DB01A0014	1	2	0		1	0	1	1		1
DB01A0014	1	3	0		1	0	1	0		1
DB01A0014	1	4	0		1	0	1	1		1
DB01A0014	1	5	0		1	0	1	0		1

午後2:29 · 2023年2月15日 · **1.4万** 件の表示

ツイートアナリティクスを表示　プロモーションする

7 件のリツイート　**2** 件の引用　**35** 件のいいね　**4** ブックマーク

属性値の組み合わせた結果だけが残っていて、組み合わせに関する制約や、属性間の依存関係などが抜け落ちている。

本質的複雑さを明らかにしなかった末路②

同じく属性値の組み合わせた結果だけが残る

ユーザプロフィール(取得している項目)			連絡手段
電話番号	メールアドレス	住所	
○	○	○	メール
○	○	×	メール
○	×	○	DM
○	×	×	電話
×	○	○	メールバージョン
×	○	×	メールバージョン
×	×	○	DM
×	×	×	なし

おそらく、要件自体は

メールアドレスが登録されていればEメールで。
メールアドレスは未登録だが、住所が登録されていればDMで。
メールアドレスも住所も未登録だが、電話番号が登録されていれば電話で。

http://www.slideshare.net/kawasima/ss-26968240

本質的複雑さを明らかにしなかった末路②

5	分割会計内訳数		N	2	「01」もしくは「02」もしくは「03」	出金方法の内訳数(「会計内訳」の件数)をセットします (最大3件まで) 1件のみ(項番6のみ使用)　「01」 2件(項番6・7使用)　「02」 3件(項番6・7・8使用)　「03」 <注意> ・円貨相当額送金の場合、「01」(1件)のみ指定可能です
6	会計内訳(1)	＃(1) 相場区分	N	2	「01」もしくは「02」もしくは「03」もしくは「04」	出金時の為替相場区分をセットします 01：SPOT･･･送金/出金通貨が異通貨で為替予約利用せず 02：CONT･･･送金/出金通貨が異通貨で為替予約利用 03：NOEX･･･送金/出金通貨が同一通貨で外貨(JPY以外)口座より出金 04：円建･･･送金/出金通貨ともに円貨(JPY) <注意> ・外貨口座出金による円建送金はお受付できません ・円貨相当額送金の場合、「01」のみ指定可能です
		＃(2) 処理内訳金額				円貨相当額送金以外の場合
		① 幣種	C	3	3桁の通貨コード	送金通貨コードをセットします(別表1参照) 例：米ドル「USD」、日本円「JPY」、人民元「CNY」
		② 小数点位置	N	1	1桁の数字	送金通貨に応じた小数点以下桁数をセットします(別表1参照) 例：USD「2」、JPY「0」、CNY「2」
		③ 金額	N	15	送金金額(含む小数点以下)	送金金額のうち、会計内訳(1)における内訳金額をセットします 出金方法が1つの場合 送金金額と同額 (会計内訳(2)[項番7]・会計内訳(3)[項番8]はすべてスペース) 出金方法が2・3つの場合 送金金額の一部 (会計内訳(1)～(3)の合計額＝送金金額) 【入力方法】右詰め／残り前「0」埋め 例：USD123,456.78の場合　「000000012345678」、 JPY123,456の場合　「000000000123456」、 CNY123,456.00の場合　「000000012345600」
		＃(3) 円貨対価額				円貨相当額送金(外貨建)の場合
		① ダミー	C	3	スペース	
		② ダミー	C	1	スペース	
		③ 金額	N	15	円貨対価金額	円貨相当額をセットします(○○○円相当) 【入力方法】右詰め／残り前「0」埋め 例：JPY1,000,000相当の場合「000000001000000」
		＃(4) 予約番号	C	16	スペースもしくは「為替予約番号」	(1)相場区分が「02」(CONT)の場合に為替予約番号(6桁)をセットします(それ以外はスペース) 【入力方法】左詰め／残りスペース
		＃(5) 引落口座				
						(1)相場区分の値に応じ、出金口座の通貨コードをセットします (別表1参照)

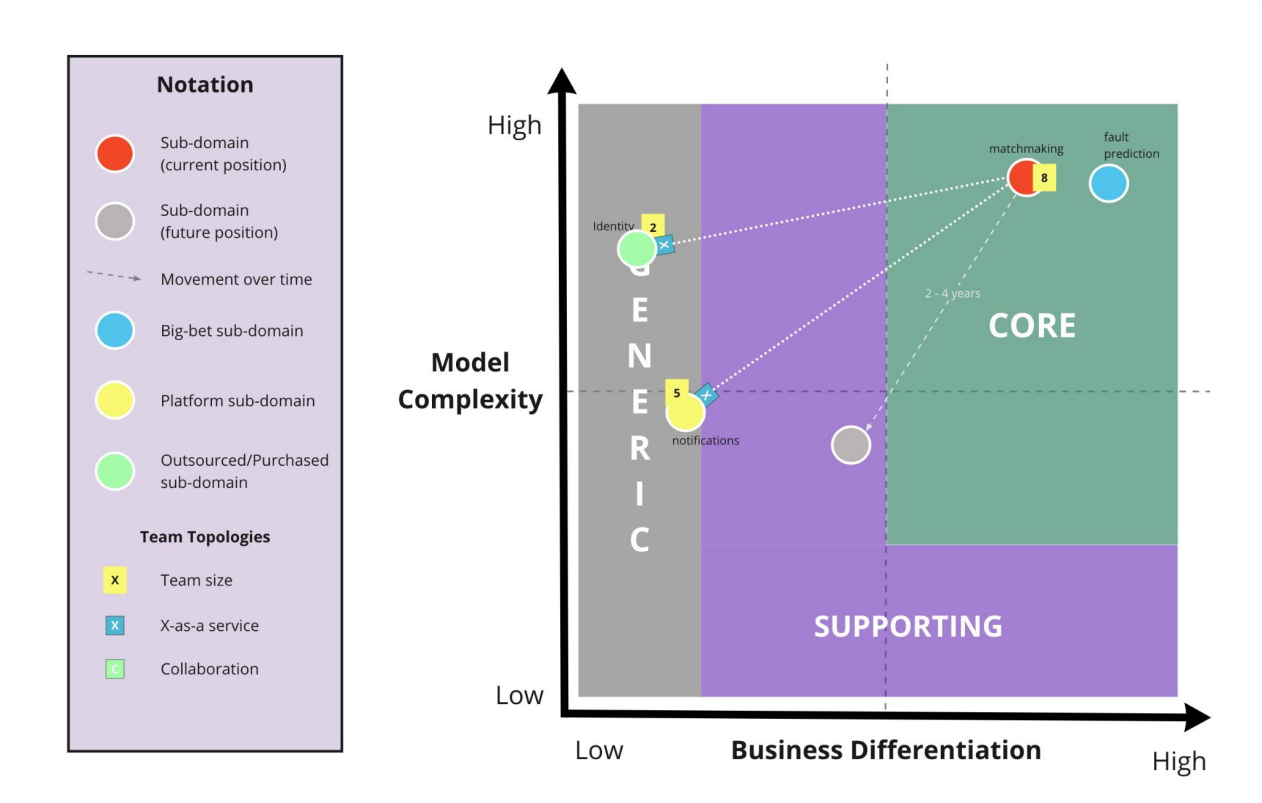
(2)か(3)のいずれかを指定

詳細を適切に隠せておらず、Domain WorldとCode Worldが混在している

業務の用語とシステム都合の用語が混じり合うと、さらに凶悪になる

https://www.smbc.co.jp/hojin/eb/computerbanking/resources/pdf/gaikokusoukinirai_zenginformat.pdf

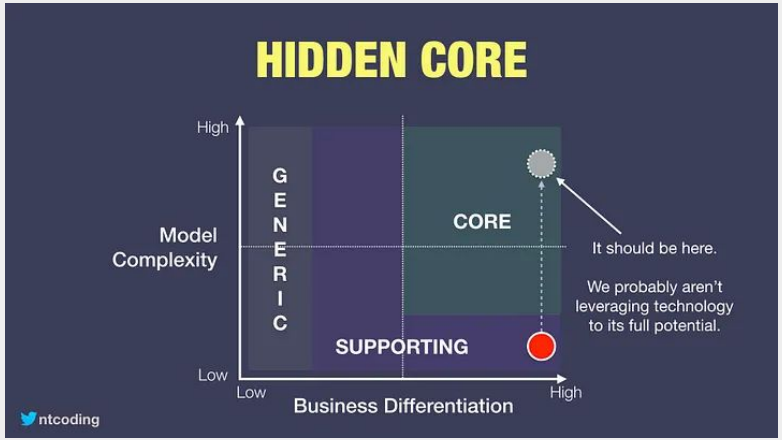
本質的複雑さが高いほど事業上重要な領域になりやすい

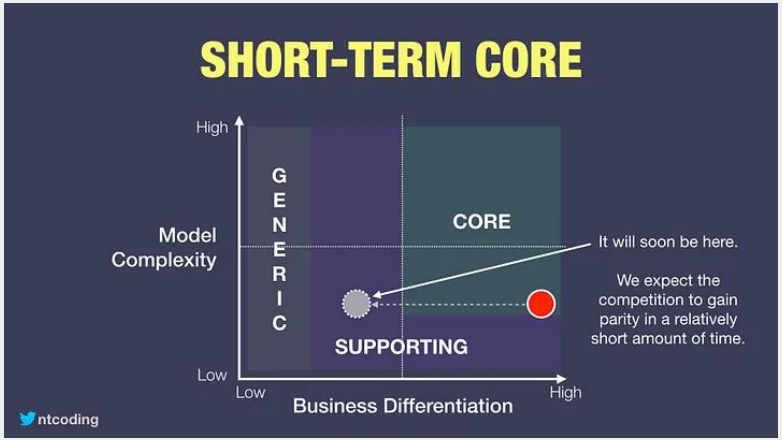


https://github.com/ddd-crew/core-domain-charts

事業上重要性が高いドメインほど変動性が高い

すなわち「複雑さ」を可視化しておかなければならない。





時間とともにサブドメインの立ち位置も変わるから
予測するのは無駄なのでは...

未来は予測できないのか(してはならないのか)?

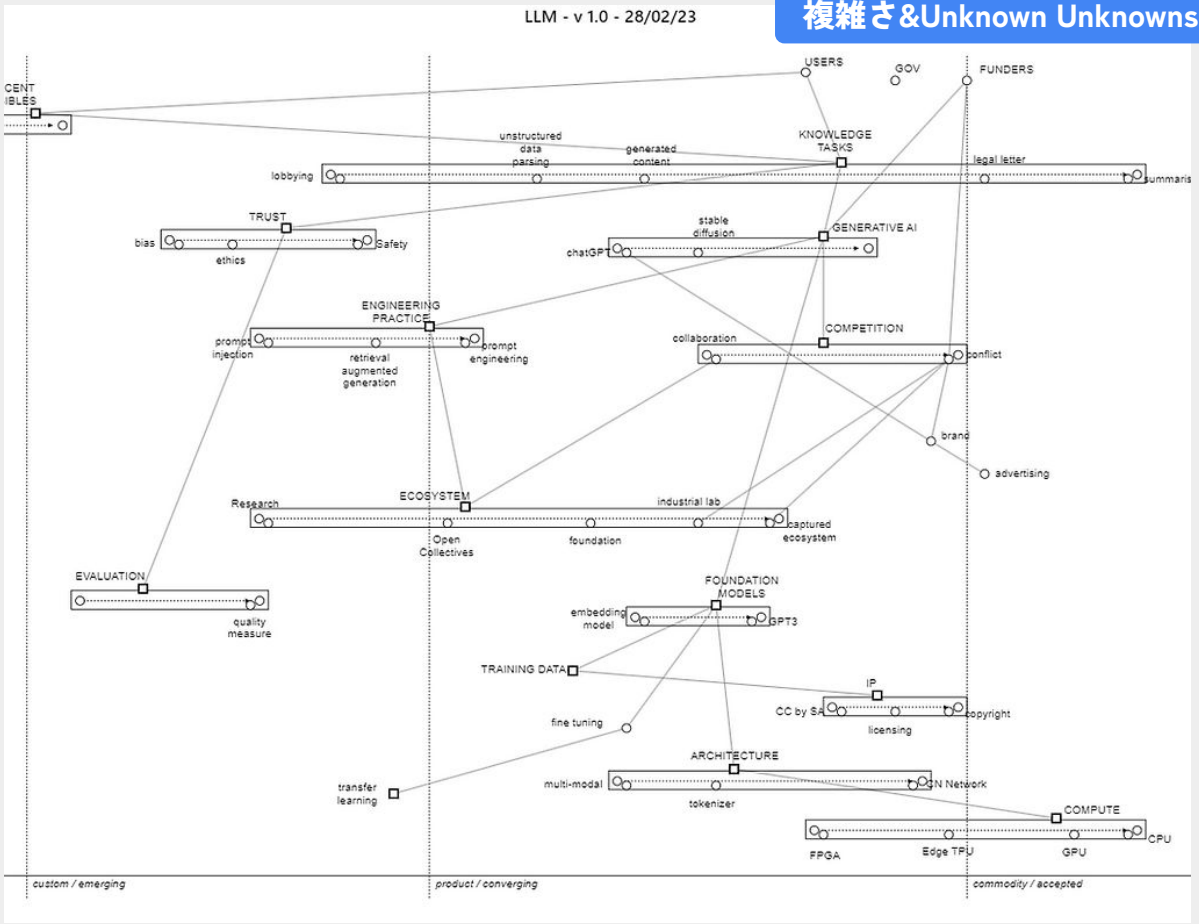
置かれている状況を理解せずに、「とりあえずやってみないと分からないから」はムダ撃ちになる。

置かれている状況(Landscape)を可視化し、それがどう移り変わるのか(Climate)を議論し、重点的に投資するコンポーネントを決める。

Simon Wardley曰く

「いつ起こるかは予測できないが、何が起きるかは予測できる」

Wardley Maps



https://www.wardleymaps.com/

Wardley Mapsからみる不確実性対処の失敗

- ProductやCommodityの領域に投資しすぎる
- ProductやCommodityに直に以降しそうな領域に投資する
- Genesisがないシステムを作る

曖昧さ

- 要求そのものに含まれる曖昧さ
  - 人によって解釈の分かれる語句の使い方
  - エッジケースの考慮もれ
- 問題設定の視座の低さに起因する曖昧さ
  - 本当に解決したい問題が別であることに後々気づき大きな代償を払うことになる。

曖昧さ

問題: 送料無料サービス

とある書籍のECサイトで、

5,000円以上の全ての注文で送料は無料にする

という旨の要求を聞いた。追加で確認しておかなくてはならないことは何だろうか?

『The Pragmatic Programmer』

曖昧さ

解答例

- 5,000円に税金は含まれますか?
- 5,000円に現在の送料が含まれますか?
- 5,000円に含むのは紙の本だけですか? 同じ注文の電子書籍も含まれますか?
- どのような配送手段がありますか? 優先順位は?
- 国際注文の場合はどうなりますか?
- 5,000円の制限は将来どれくらい変わりそうですか?

曖昧さ

問題: 音楽プレイヤー

ミュージックプレイヤーを作ります。

- プレイリストには複数の曲が含まれます。
- プレイリストを選択して再生します。
- 再生の方式には、通常再生の他に、ランダム再生とリピート再生があります。
- リピート再生は一回ボタンを押すと、全曲リピート、もう一度押すと1曲リピートに切り替わります。

仕様として確認しておくことは何でしょうか?

曖昧さ

演習: 自動精算機

要求を伝える側も受け取る側もなんとなくやりたいことは理解できるが、実装可能かわからない

現金の自動精算機のシステムを作ります。表示された金額を支払うため顧客は現金を投入し、投入し終えたら「投入完了ボタン」を押し、釣り銭を受け取ります。

- 紙幣および硬貨は一度に複数枚投入できます
- 投入ごとに精算機は投入額と請求額と比較し、投入額が大きければ「投入完了ボタン」を表示します

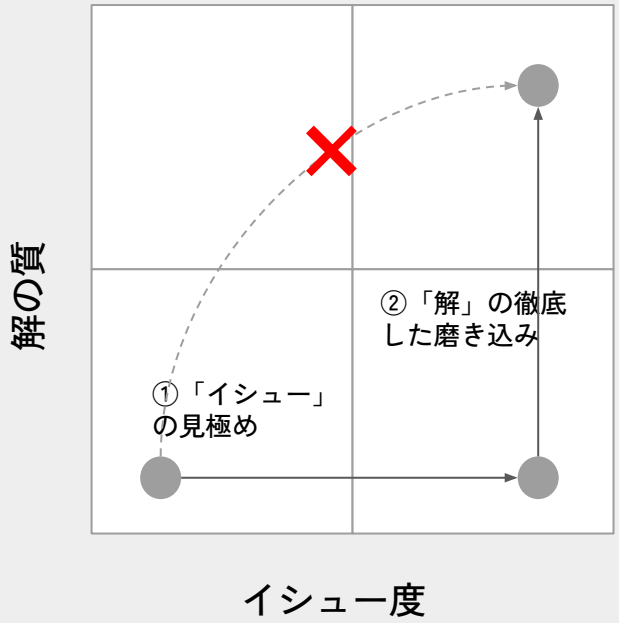
リリース後、投入完了ボタンを常に押すのは面倒なので、もう顧客が投入する必要はない、と判断できれば、ボタンを押さずとも、釣り銭を出すようにしたい、という要望が上がってきました。

ただ、小銭を減らすために、釣り銭をできるだけ大きな硬貨でもらえるように投入する人もいるので、単純に投入金額が請求金額を超えた時点で釣り銭を出す訳にはいきません。

仕様として明記してみましょう。

曖昧さ

本当の問題を見つける



https://www.amazon.co.jp/dp/4862760856

曖昧さ

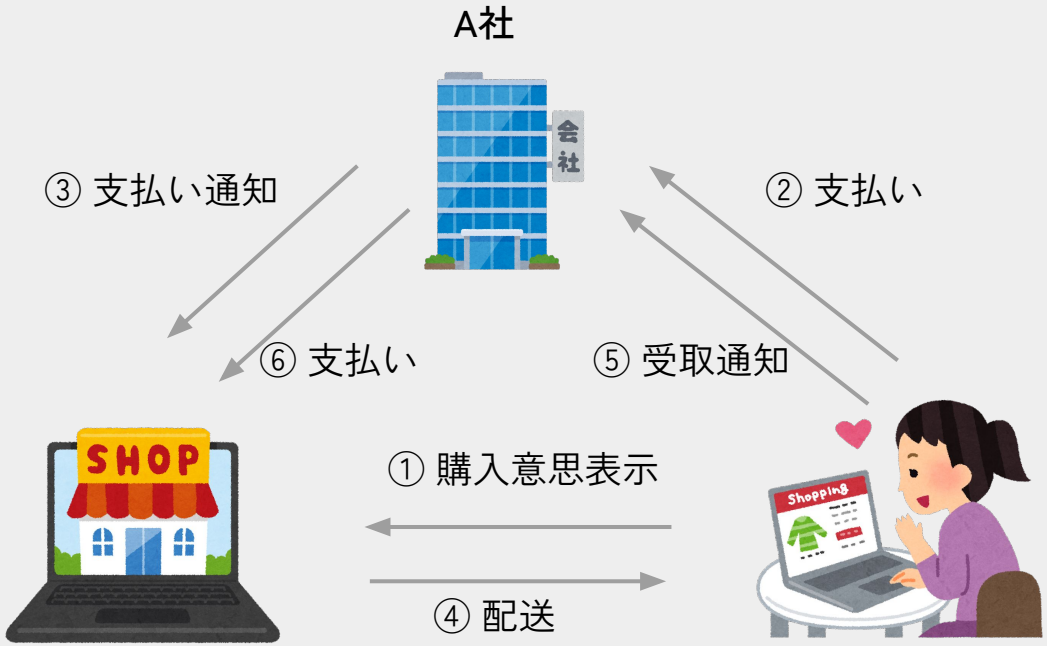
例題: 安心できないマーケットプレイス

マーケットプレイスを運用するA社は、出店している店舗の質低下に悩まされている。

発送したが届かない、違うものが届くなどが相次ぎクレーム対応におわれることになった。

この出店店舗の質低下に対して、どういう対策が考えられるだろうか…?

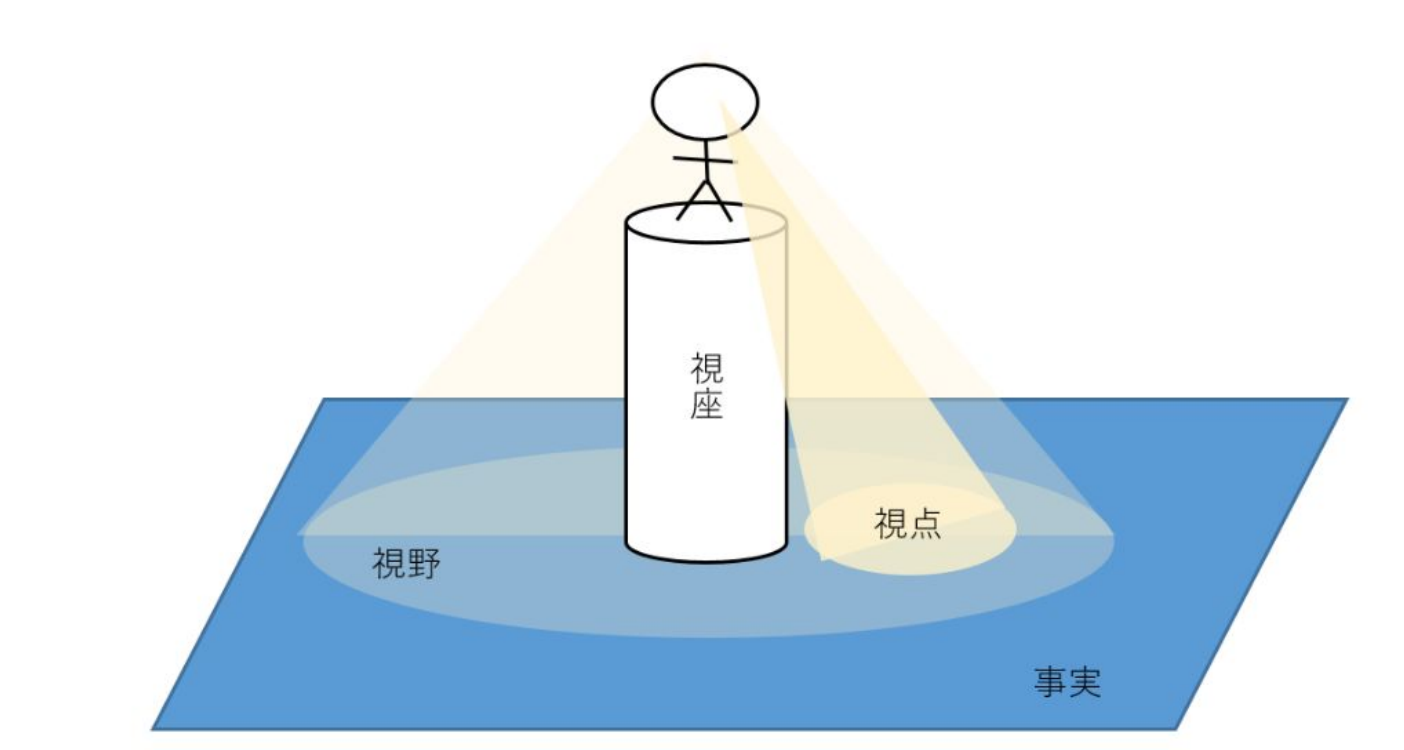
解答例: エスクロー



問題を「カスタマが安心して買える(カスタマ不利益をもたらさない)こと」と考える

曖昧さ

抽象化と視座の上げ下げ



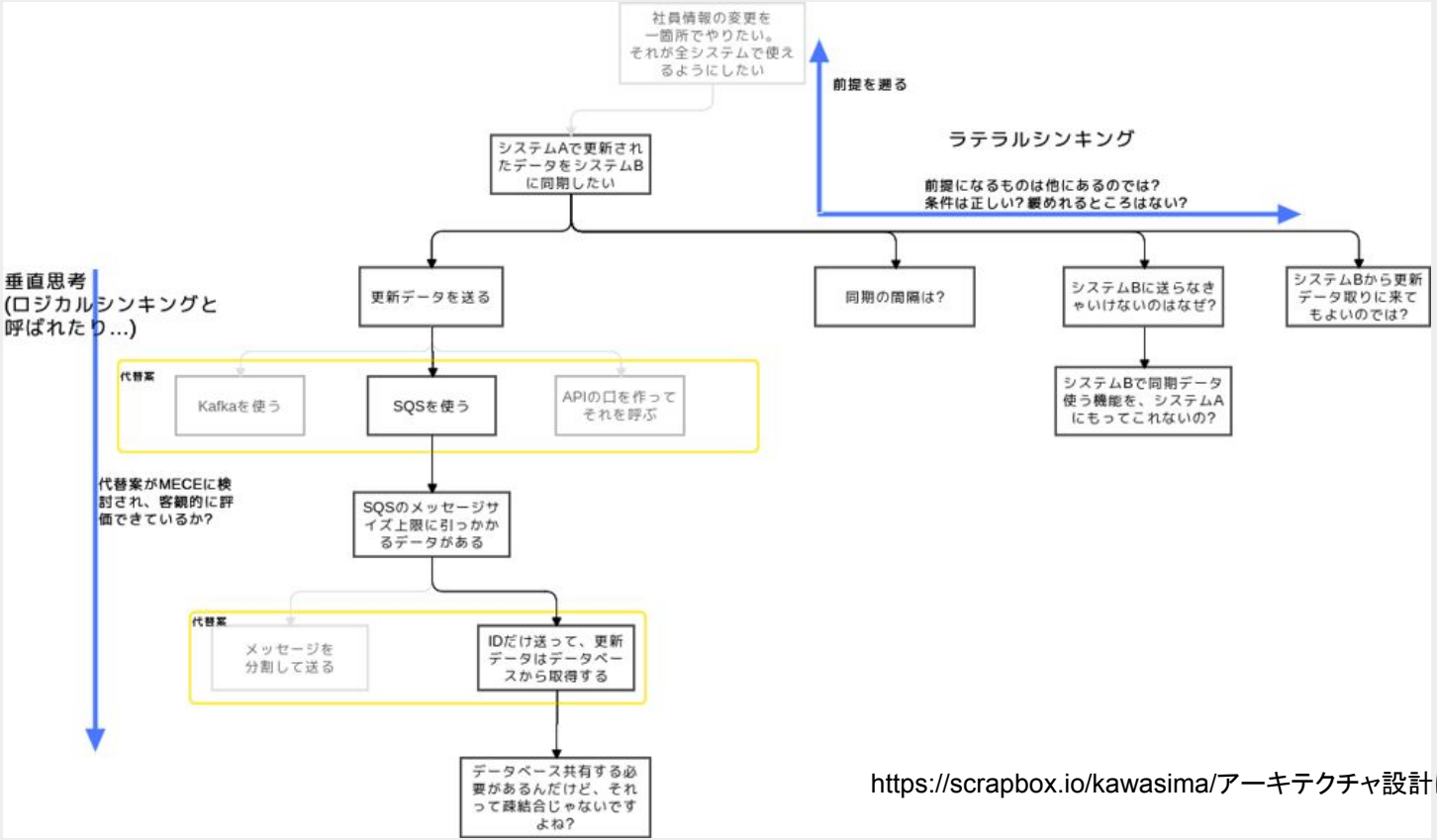
曖昧さ

垂直思考と水平思考

- 垂直思考
  - いわゆるロジカルシンキング
  - 代替案をMECEに検討し、思考を掘り下げる
  - デシジョンにおける代替案の検討
- 水平思考
  - 思考の幅をひろげ、本質を考える
  - 「抽象化」と「常識を疑うこと」が鍵
  - コンテキストの抽象化と多角的検討

曖昧さ

意思決定の構造



https://scrapbox.io/kawasima/アーキテクチャ設計における垂直思考と水平思考

曖昧さ

垂直思考: デシジョン

- 評価軸を定め、そのバリエーションにより取りうる代替案を洗い出す。
- 代替案それぞれの各評価軸について、Pros/Consを公正な目で記述する。

https://scrapbox.io/kawasima/アーキテクチャ設計における垂直思考と水平思考

曖昧さ

水平思考: コンテキスト

問題を深く理解する

- 明らかなファクトは何か？
- 条件は何か？各条件が直交しているか？
- 制約は何か？ 時間/お金/人など観点が足りているか？

曖昧さ

水平思考の肝: 抽象化

- コンテキストは「解決すべき問題」が記述されているか？手段が書かれていないか？
- 類似の問題をかき集め、共通点を見出す。
- 近い将来起こることが予想される、追加される機能や環境の変化を含めて考えると...

曖昧さ

水平思考の肝: 疑いの目

- 前提条件/制約を疑う、緩和する
  - 「〇〇さんが言っているから」を根拠としたものは覆りやすい。
  - この条件/制約を緩和できれば、代替案検討に大きなインパクトがあるのに...
  - 時間経過による制約の変化はあるか？

曖昧さ

視点・視野・視座と意思決定構造の関係性

	視点	視野	視座
影響する箇所	どこに着目するか 代替案の評価・選択	見えている範囲 代替案をどれだけ出せるか	見ているものの抽象度 コンテキストを適切に捉えているか?
必要な要素	センス (論理性)	経験	職位

曖昧さ

例題: サイネージ広告の掲載

A社はサイネージ広告プラットフォームを提供しています。

広告は1ヶ月契約で、時間枠を購入し、その分だけ広告原稿を制作して入稿します。

- 毎月、翌月の購入枠数を3営業日前までに、クライアントは入稿します。
  - 広告の事前審査を行うためのリードタイムです。
- 広告は当月のものと同じものを流用して入稿することができます。
- A社の担当者は、広告枠に穴がでないように、クライアントへ入稿を催促したり、月途中で契約落ちの穴埋めを売り込んで、代替原稿を入稿してもらったりします。

A社にとって業務がなるべく楽にまわせるよう、必要な仕組みを設計してみましょう。

曖昧さ

次月の広告枠購入

広告原稿作成

審査

枠数充足チェック

掲載

催促

Wrap up

- 不確実性や複雑さは無くすことはできない。
- 何が起こるか分かってないことによって発生する不確実性が実際はプロジェクトを苦しめる。
  - 不一致、複雑さは「シンプルな」モデルを書いてステークホルダー間で共有する。
  - 曖昧さは、分かったつもりがブラインドになる。
- 不確実性や複雑さとうまく付き合っていくスキルは潰しの効くスキル。