ASTER Association of Software Test Engineering

テスト設計コンテスト ’23

テスト設計チュートリアル
テスコン編

2023. 6. 14

NPO法人 ソフトウェアテスト技術振興協会（ASTER）

チュートリアルの流れ

1. テスト開発プロセス
2. TRA（テスト要求分析）
3. TAD（テストアーキテクチャ設計）
4. TDD・TI（テスト詳細設計、テスト実装）
5. テスト開発のTIPS
6. 過去の応募作のポイント

本資料はVSTeePを基に作成している。

1．テスト開発プロセス

求められるテスト開発

ソフトウェア　→　大規模化・複雑化している

ソフトウェア開発　→　**高品質**・短納期を求められる

↳ **顧客の要求・要件に合致すること**

【社会的な急務】

柔軟で高速かつ精度の高いソフトウェアテストを開発すること

10万件を超える様々な観点による質の高いテストケースを
いかに体系的にスピーディに開発するか、が課題である。

テスト計画とテスト戦略

テスト計画やテスト戦略の立案という工程は未成熟である。

テストケースという成果物の開発とテストのマネジメントが混同されている。
テスト計画書に**テストケース**も**スケジュール**も**人員アサイン**も全て記載されている。

さらに・・・
テスト（ケース）開発という用語はほとんど使われない。
テスト計画、テスト仕様書作成、テスト実施準備という用語は使われている。
テスト戦略という用語もイマイチよく分からない。

テストケースの開発という工程が見えにくくなっており、
そこで必要となる様々な工程が混ざり、一体的に扱われている。

テスト開発プロセス

テストケースを開発成果物と捉え、開発のプロセスを整備する必要がある。

大規模化・複雑化・高品質・短納期に対応するテストケースを開発する
ために、必要な作業を明らかにする。

→　**「テスト開発プロセス」という概念が必要となる。**

テスト設計とテスト開発の違い

いきなりコーディングをすることと、ソフトウェア開発との違いと同じである。

テスト設計

テスト（詳細）設計技法と文書フォーマットが分かっていればできる。

コーディングもアルゴリズムとプログラミング言語を分かっていればできる。

テスト開発

テスト要求を把握して、テスト設計の原則を理解・適用して、
段階的に詳細化しながら順序立てて進めていく必要がある。

ソフトウェア開発も同じ

→　様々なソフトウェア工学上の概念や技術、方法論、モデリングスキル
抽象化／パターン化能力なども必要となる。

テスト開発がきちんとできるようになると、自動化やAIを用いることもできるようになる。

つまりは・・・？

テストのことだけではなく
ソフトウェア開発に必要となる**思考力**や
モデリング力を身につけよう

テスト開発プロセスの基本的考え方

テストケースを開発成果物と捉え、
ソフトウェア開発プロセスとテスト開発プロセスを対応させよう。

ソフト要求分析　＝　テスト要求分析
ソフトアーキテクチャ設計　＝　テストアーキテクチャ設計
ソフト詳細設計　＝　テスト詳細設計
ソフト実装　＝　テスト実装

テストケースがどの工程の成果物かを考えるために、
各プロセスの成果物を対応させよう。

ソフト要求仕様（要求モデル）　＝　テスト要求仕様（要求モデル）
ソフトアーキテクチャモデル　＝　テストアーキテクチャモデル
ソフトモジュール設計　＝　テストケース　　.
プログラム　＝　テスト手順
（手動／自動化テストスクリプト）

旧来のテストプロセスでは粗すぎる

旧来

テスト項目の抽出

理想

テスト要求の
獲得と整理・
テスト要求モデリング

テストアーキテクチャ
モデリング

テスト詳細設計
モデリングと
テスト技法の適用による
テストケースの列挙

手動／自動化
テストスクリプト
（テスト手順）の記述

本資料はこの **VSTeP** を基に作成している。

テスト観点とテストケースとテスト手順

テスト観点とテストケースとテスト手順をきちんと区別する。

「Myersの三角形」を例にすると

テスト観点

何をテストするのかのみ端的に記述したもの

正三角形

テストケース

そのテスト観点でテストするのに必要な値のみを特定したもの

通常は、網羅基準に沿って特定される値のみから構成される。

テストケースは**基本的にシンプル**になる。

（1, 1, 1）
（2, 2, 2）
（3, 3, 3）
…

テスト手順（テストスクリプト）

そのテストケースを実行するために必要な全てが書かれたもの

手動でのテスト手順書の場合もあれば、自動テストスクリプトの場合もある。

複数のテストケースを集約して１つのテスト手順にすることもある。

1. PCを起動する
2. C:¥sample¥Myers.exe を起動する

これらを区別し、異なる文書に記述し、異なる開発工程に割り当てることで、テスト観点のみをじっくり検討することができる。

上流工程でのモデリング

上流工程でテスト観点のモデリングを行って、テストを開発すべきである。

- モデリングを行うと、テストで考慮すべき観点を一覧でき、俯瞰的かつビジュアルに整理できる
- テスト要求分析では、テスト要求モデルを作成する
- テストアーキテクチャ設計では、テスト詳細設計・実装・実行しやすいようにテスト観点をグルーピングする

（各項目の詳細は次ページ以降）

上流工程でのモデリング（モデリングの重要性）

モデリングを行うと、テストで考慮すべき観点を一覧でき、俯瞰的かつビジュアルに整理できる

- 10万件を超えるテストケースなど、テスト観点のモデル無しには理解できない
- モデルとして図示することで、大きな抜けや偏ったバランスに気づきやすくなる
- 複数のテストエンジニアでレビューしたり、意志を共有しやすくなる

上流工程でのモデリング（テスト要求モデル）

テスト要求分析では、テスト要求モデルを作成する

- テスト対象、ユーザの求める品質、使われる世界などをモデリングする
- 具体的なテスト条件が特定できるまで丁寧に段階的に詳細化する
- 仕様書や設計書を書き写すのではなく、モデリングし直す覚悟で行う

→ **最終的に「テスト観点を示す」ことにつながる。**

上流工程でのモデリング（テスト観点のグルーピング）

テストアーキテクチャ設計では、テスト詳細設計・実装・実行しやすいようにテスト観点をグルーピングする

- テスト観点をグルーピングして、テストタイプやテストレベル、テストサイクルを設計する
- テスト観点をグルーピングして、テストケースの構造（スケルトン）を設計する
- 見通しのよいテストアーキテクチャを構築する
- テストスイートの品質特性を考慮して適切なテストアーキテクチャを設計する
- よいテスト設計のためのテストデザインパターンを蓄積し活用する

テスト観点の例：組込みの場合（１）

機能：テスト項目のトリガ

- ソフトとしての機能
  - ＞　音楽を再生する
- 製品全体としての機能
  - ＞　走る

パラメータ

- 明示的パラメータ
  - ＞　入力された緯度と経度
- 暗黙的パラメータ
  - ＞　ヘッドの位置
- メタパラメータ
  - ＞　ファイルの大きさ
- ファイルの内容
  - ＞　ファイルの構成、内容
- 信号の電気的振舞い
  - ＞　チャタリング、なまり

プラットフォーム・構成

- チップの種類、ファミリ
- メモリやFSの種類、速度、信頼性
- OSやミドルウェア
- メディア
  - ＞　HDDかDVDか
- ネットワークと状態
  - ＞　種類
  - ＞　何といくつつながっているか
- 周辺機器とその状態

外部環境

- 比較的変化しない環境
  - ＞　場所、コースの素材
- 比較的変化しやすい環境
  - ＞　温度、湿度、光量、電源

テスト観点の例：組込みの場合（２）

状態
- ソフトウェアの内部状態
  - ＞　初期化処理中か安定動作中か
- ハードウェアの状態
  - ＞　ヘッドの位置、省電力モード

タイミング
- 機能同士のタイミング
- 機能とハードウェアのタイミング

性能
- 最も遅そうな条件は何か

信頼性
- 要求連続稼働時間

セキュリティ

GUI・操作性
- 操作パス、ショートカット
- 操作が禁止される状況は何か
- ユーザシナリオ、10モード
- 操作ミス、初心者操作、子供

出荷先
- 電源、電圧、気温、ユーザの使い方
- 言語、規格、法規

障害対応性
- 対応すべき障害の種類
  - ＞　水没
- 対応動作の種類

テスト観点はテストケースの「意図」を表している

Test Request Analysis

2．TRA（テスト要求分析）

テスト要求分析（TRA）

顧客やステークホルダーから**「何に対して、どのようなテストを行うように求められているか」**を得るため、主に以下の項目を確認しなければならない。

- 何をテストするのか？（何をテストしてほしいのか？）
- テストの範囲はどこからどこまでか？（他システムまで及ぶのか？）
- テストの規模はどれくらいか？（機能数や画面数など）
- テストレベルは何か？（コンポーネントテスト？統合テスト？システムテスト？）
- テストタイプやテスト観点は何か？
- 顧客やステークホルダが特に重視しているのは何か？
- テストの期間やコストはどれくらいか？

テスト要求の源泉

もし入手できるのなら、**テスト要求の源泉を準備する。**

テスト要求の源泉　＝　テスト要求分析に活かせる情報のこと

- 動いているテスト対象や動作の結果・状況
- 要求系文書、開発系文書、ユーザ系文書、マネジメント系文書など各種文書
- テストに関して従うべき社内・社外の標準や規格
- ヒアリングできるステークホルダー

など

１つでも入手できないなら、**自分たちでステークホルダーになりきって**
ブレーンストーミングや仮想ヒアリングなどを行う。

テスト要求の源泉からテスト要求を獲得する

テスト要求の分割

テスト要求を、**テストケースを導くエンジニアリング的テスト要求**と、
テストケースを導かないマネジメント的テスト要求に分ける

エンジニアリング的テスト要求

システムの完成像とテスト対象の途中経過に関わる。

- システムの完成像への要求例
  - ＞ システム要求、ソフトウェア要求、機能要求、非機能要求、理想的な使い方、差別化要因、目的機能
- テスト対象の途中経過に関する情報例
  - ＞ 良さに関する知識：テスト対象のアーキテクチャや詳細設計、実装、自信があるところ
  - ＞ 悪さに関する知識：バグが多そうなところ

マネジメント的テスト要求

次ページで詳しく説明

品質リスクやテストアーキテクチャ設計などに反映させる。

- 工数、人数、スキル分布、作業場所、オフショアか否か、契約形態など
- 機材利用可否（シミュレータや試作機）、ツール利用可否（ツールの種類、ライセンス、保有スキル）など
- 目標残存バグ数、信頼度成長曲線など
- テストスイートの派生可能性や保守性など

前ページの「バグが多そうなところ」について

人的要因が主だが、以下のようなところに注意を向ける。

- ユーザがミスしそうなところ
- 構造上問題が起きそうなところ
- 前工程までの検証作業（レビューやテスト）が足りなかったり滞ったりしたところ
- 類似製品や母体系製品の過去バグ、顧客クレームから分析した知識
- スキルの足りないエンジニアが担当したところ
- 設計中に不安が感じられたところ
- 進捗が滞ったり、エンジニアが大きく入れ替わったりしたところ

テスト要求モデルの構築と納得

テスト要求モデルを構築し、**自分たちとステークホルダーが納得するまで**モデルを洗練する。

テスト要求モデルの構築

1. エンジニアリング的テスト要求を基に、テスト観点を列挙する
2. 階層構造やマトリクス、一覧表などでテスト要求モデルを記述する
3. テスト観点を詳細化したり、関連を追加したりする
4. モデルを洗練する

→ **最終的に「テスト観点図」や「テスト観点表」になる。**

自分たちが「十分に充実した」と実感できたら、そのモデルを囲んで、
ステークホルダーが納得するまで一緒に洗練し続ける。

→ ステークホルダーに不十分・不完全・不正確な把握が無いようにする。

→ **ステークホルダーに「本当はこれだけテストしなければならない」という実感を持ってもらうことが重要**である。

テスト観点を挙げるときの注意点（１）

- **仕様書に書いてある仕様や文、単語を書き写してもテスト観点は網羅できない**
  - 仕様書は**通常不完全**である（もしくは、テスト観点を挙げる時点で存在していない）
  - (特に致命的な）バグは**仕様書に書かれていないテスト観点でしか見つけられないことがある**

ここまで例に挙げたテスト観点は、皆さんの組織の仕様書に全て書いてありましたか？

- **テスト観点は多面的に挙げる**

  入力に関するテスト観点だけ挙げたり、機体結果やその観測に関するテスト観点だけ挙げたり、品質特性だけをテスト観点として採用すると、**漏れが発生する。**

「観点」だから機体結果やチェックポイントだけを挙げるのは
テスト観点を理解していない証左である。

テスト観点を挙げるときの注意点（２）

- **テスト観点を挙げただけで**バグが見つかる**ことがある**
  - **開発者が考慮していなかったテスト観点はバグの温床**であり、テストしなくてもバグだと分かることも多々ある
  - 期待結果に関するテスト観点を挙げたり詳細化したりすると、仕様の抜けが分かることがある

    テストケースにも期待結果を必ず書くのを忘れないこと。
    「正しく動作すること」は期待結果ではない！

- **「網羅した」風のテスト観点の文書や納品物を作ろうとせず、網羅しようと多面的に様々なことを考え尽くそう**
  - 「あーでもない、こーでもない」と**ワイワイ議論したアイデアのリポジトリがテスト観点図（テスト観点表）**である

    見逃したバグを見つけ得るテスト観点を気軽に追加して整理することも大事である。

  - 最初から網羅しようと気張ると大事なテスト観点を見逃す

    そもそも網羅することが必要なのか？

モデリングの２つのアプローチ

トップダウン型

おもむろにテスト観点を挙げ、詳細化していく

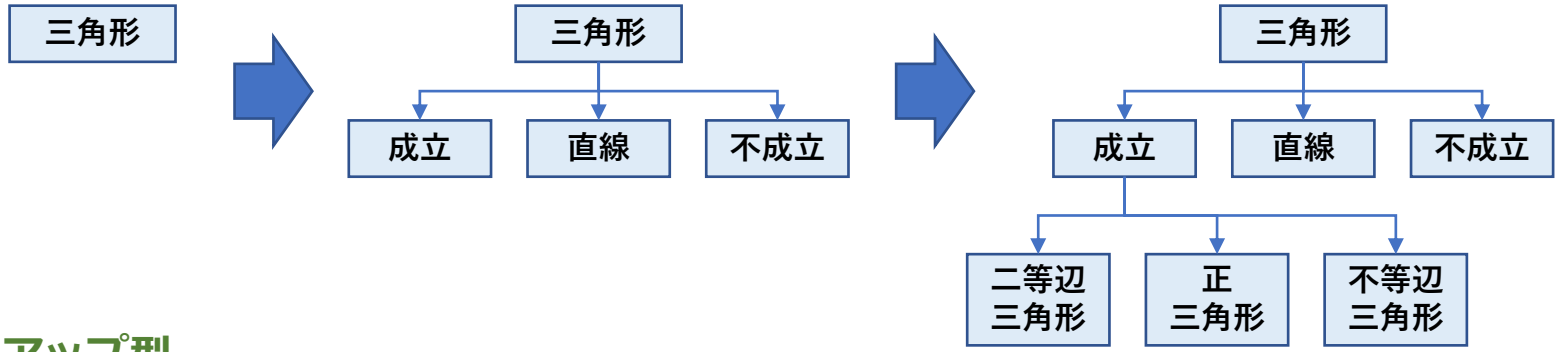
ボトムアップ型

思いつくところから具体的なテストケースを書き、いくつか集まったところで抽象化し、テスト観点を導き出していく。

モデリングの２つのアプローチ

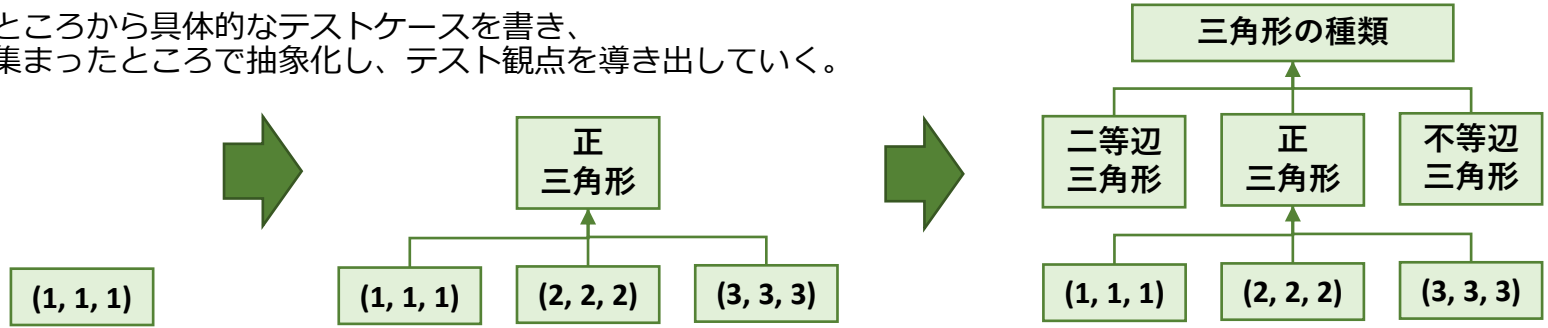
トップダウン型

おもむろにテスト観点を挙げ、詳細化していく



ボトムアップ型

思いつくところから具体的なテストケースを書き、
いくつか集まったところで抽象化し、テスト観点を導き出していく。

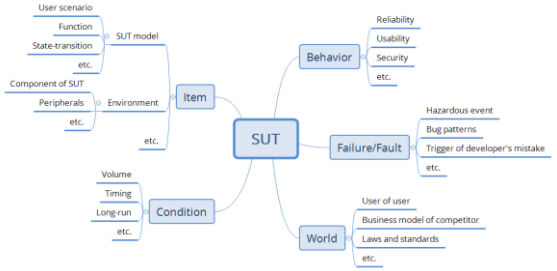


トップダウンとボトムアップを循環させながらモデリングしていく。

テスト要求モデルの全体像の種類

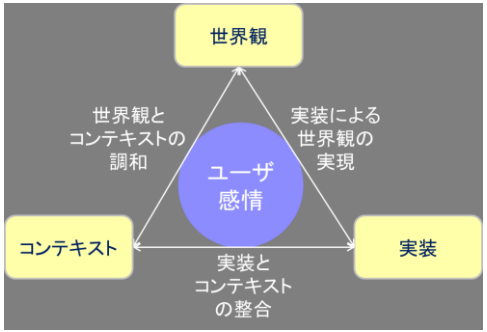
「テスト要求モデルの全体像をどう整理すればよいか」に**唯一絶対の解はない。**

- **一面的な全体像のモデルは考慮が足りないことが多い**
  - **仕様ベースモデル**
  - **機能ベースモデル**
  - **画面ベースモデル**
  - **正常系異常系モデル**



CIBFWモデル

- 多面的な方が考慮されている可能性が高いが、
  お仕着せのものを流用したものは考慮が足りないことが多い
  - **品質特性モデル**
  - **一般テストタイプモデル**
  - **CIBFW（Condition／Item／Behavior／Fault／World）モデル**
  - **三銃士モデル**
    世界観／コンテキスト／実装 + ユーザ感情
  - **システム構成モデル**
    そのシステム構想がそのまま全体像になる（ラルフチャートなど）



三銃士モデル

全体像の構造によってモデリングの質がかなり左右される。
大事なのはどれを採用したかではなく、考慮が十分かどうかである。

テスト要求モデルの洗練

質の高いモデルにするために様々な洗練を行う。

- **網羅化：MECE分析（漏れやダブりがないか）**
  - 子観点がMECEに列挙されているかどうかをレビューし、不足している子観点を追加する
  - MECEにできない場合、必要に応じて「その他」の子観点を追加し、非MECEを明示する
  - 子観点をMECEにできるよう、適切な抽象度のテスト観点を親観点と子観点の間に追加する
- **網羅化：親子関係分析**

  MECE性を高めるために、テスト観点の親子関係を明示し、子観点を分類する。
- **整理**
  - 読む人によって意味の異なるテスト観点を特定し、名前を変更する
  - テスト観点や関連の移動、分割、統合、パターンの適用、テスト観点と関連・網羅基準との変換などを行う。
  - 本当にその関連が必要なのかどうかの精査を行う必要もある
- **剪定**

  ズームイン・アウト、テスト観点や関連の見直し、網羅基準や組み合わせ基準の緩和によって
  テスト項目数とリスクとのトレードオフを大まかに行う。
- **確定**

  子観点および関連が全て網羅的に列挙されているかどうかをレビューすることで、
  テスト要求モデル全体の網羅性を明示し、見逃しリスクを最小化する。

Test Architecture Design

3．TAD（テストアーキテクチャ設計）

テストアーキテクチャ設計（TAD）

テストアーキテクチャ設計とは、**テストスイートの全体像を把握しやすく**しつつ**後工程や派生製品、後継プロジェクトが作業しやすく**なるように、**テスト観点をグルーピングしてテスト要求モデルを整理する工程**である。

VSTePでは、テスト観点のグルーピングについて、
以下の２つのモデリングを推奨している。

- テストコンテナモデリング
- テストフレームモデリング

互いのモデリングを反復的に進めていくことを想定しており、
どちらから始めても構わない。

テストコンテナモデリング

テスト要求モデルを分割し、**同じまとまりとしてテストすべきテスト観点を同じグループにまとめること**で整理する。

複数のテスト観点をグルーピングしたものを**「テストコンテナ」**と呼ぶ。

テストコンテナモデリングのねらい

- グルーピングすることで粒度が粗くなり、テストスイートの全体像の把握や全体に関わる設計がしやすくなる
- 自社で定められているテストレベルやテストタイプの趣旨を理解し、全体像を設計できるようになる。

テストにも保守性など特有の「テストスイートの品質特性」や「テストコンテナの責務」があるが、どのような品質特性や責務を重視するかによって異なるテストアーキテクチャとなる。

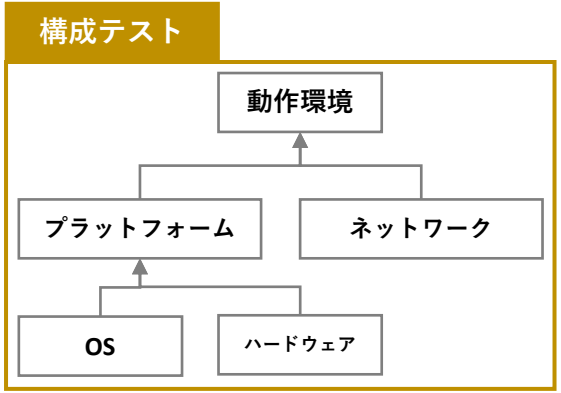
テストコンテナ

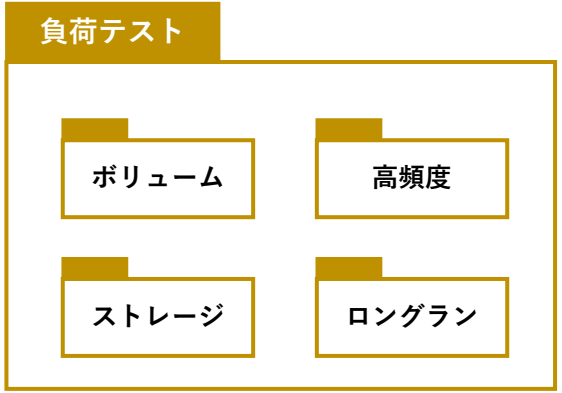
テストコンテナを使って、
テストタイプやテストレベル、テストサイクルなどを表現する。

それぞれの区別に
時間を取られないで済む

テストコンテナは**包含関係を持つことができる。**
テストコンテナに含まれるテスト観点を比べると、**テストコンテナ間の役割分担を明確に把握できる。**

違いがよく分からない
ものも区別できる

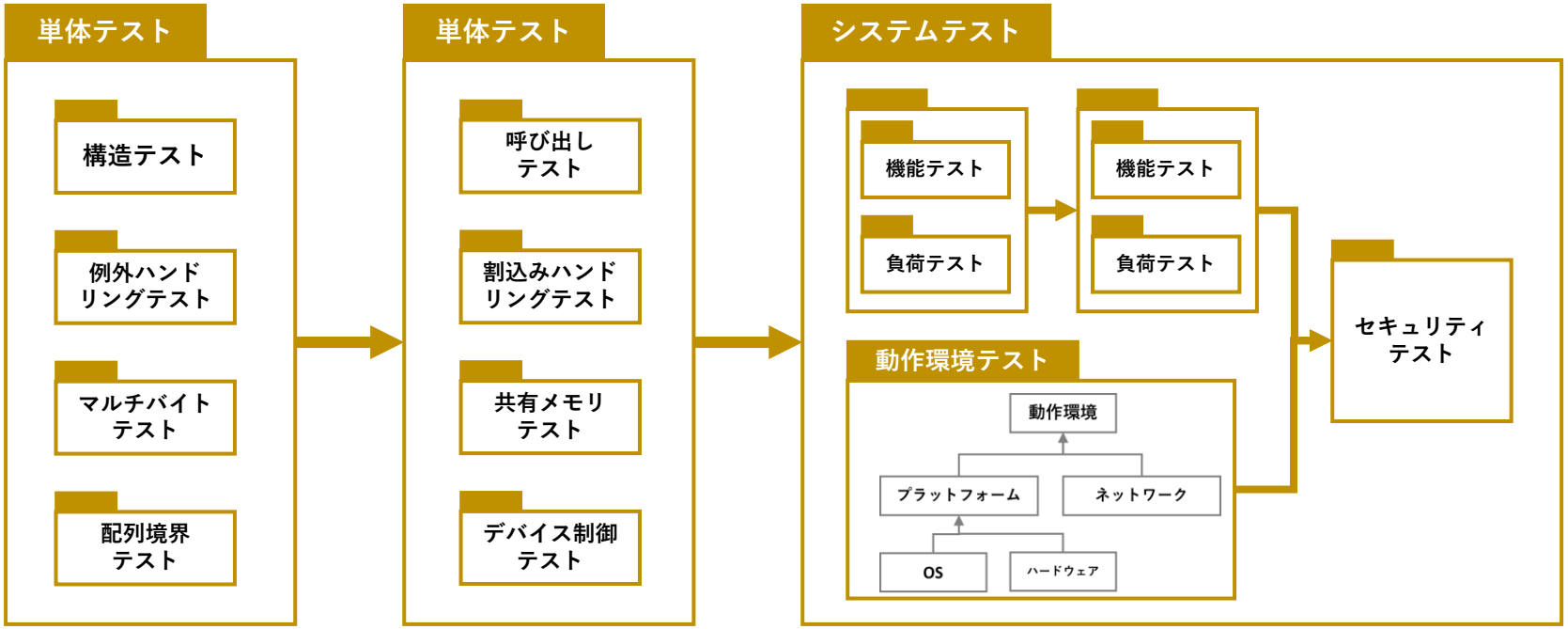




テストコンテナモデル

テストコンテナのモデルで表すと全体像を把握しやすくなる。

先に設計／実施しておくべきコンテナを後に回してしまう、といったトラブルが防げる



テストコンテナモデリングのメリット

- テスト観点よりも粒度の粗いテストコンテナを用いるので、**俯瞰して把握しやすい**

- テストコンテナ間に含まれるテスト観点を比べると、**役割分担を明確に把握できる**
  現場で経験的に用いているテストコンテナの妥当性を評価したり改善できるようになる。

  負荷テストと性能テストなど、違いがよく分からないテストタイプを区別できる。
  単体テストと結合テストなど、役割分担がよく分からないテストレベルを区別できる。

- 複数のテストタイプからなるテストレベル、サブレベルを持つテストレベルなどのように、
  他のテストコンテナを包含しながらまとめることもできる

- テストコンテナの順序や依存関係、設計時期（や実施時期）を適切に考慮することができる
  CIなどツールチェーンを構築したり、フロントローディングしたり、シフトライトをする際には必須となる。

- テストスイートの品質特性（保守性や自動化容易性など）やテストコンテナの責務を
  意識してテスト設計に反映できるので、**納得しやすいテストアーキテクチャになる**

テストコンテナモデリングのポイント

- テストが小規模で単純な場合、テスト要求モデルの大まかな分類をそのままテストコンテナにしてしまってよい場合もある

- テスト要求モデルでは単一だったテスト観点を分割したり統合したりすることにより、**意味の違いに気付いて**名前を変えたりテスト観点を追加したりすることもある

  現実的には、テストコンテナモデリングとテスト観点モデリングは反復的に行うことになるだろう。

- マネジメント的要求から**テストスイートの品質特性やテストコンテナの責務を抽出し、それらを達成できるように**モデリングする

  例）保守性を高めたいというマネジメント的要求がある場合、
  変更要求があまり入らないテストコンテナと変更要求が頻繁に入るテストコンテナに分割する

- テスト（スイート）のアーキテクチャ、システム・サービス・製品・ソフトウェアのアーキテクチャ、開発環境やテスト実行環境（テストシステム）のアーキテクチャ、開発プロセスは**相互に影響する**

テストコンテナへのまとめかたの基本

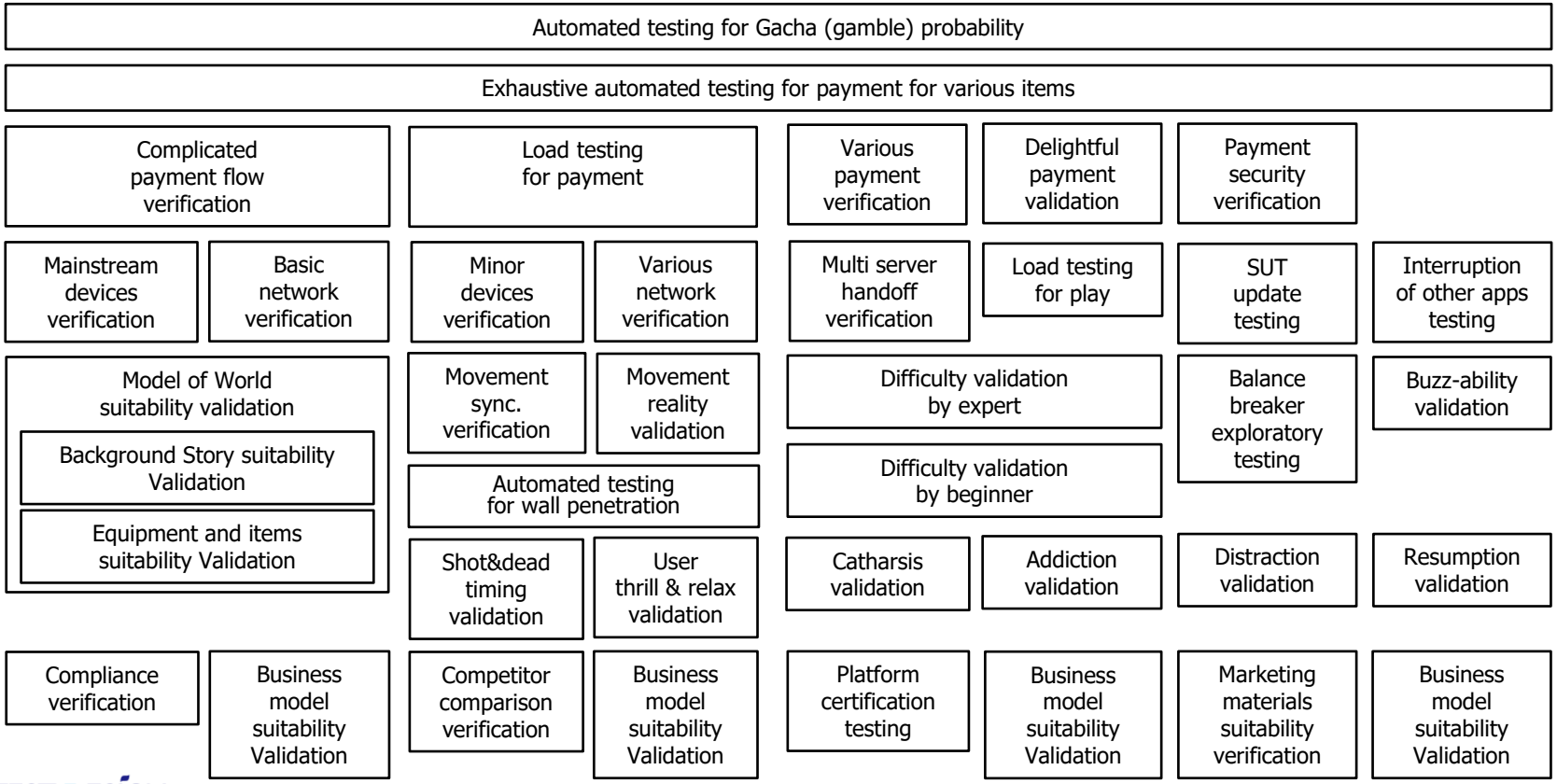
- **同じ時期にテストすべきテスト観点**を**テストレベル**としてまとめる
  テスト観点には、テスト観点同士、または欠陥特定の絞り込みのための順序依存関係や、
  プロジェクト制約としての時期依存関係がある。

- **同じ趣旨を持っているテスト観点**を**テストタイプ**としてまとめる
  例えば、負荷テストタイプは複数のテスト観点から成るが、「負荷をかける」という同じ趣旨を持つ。

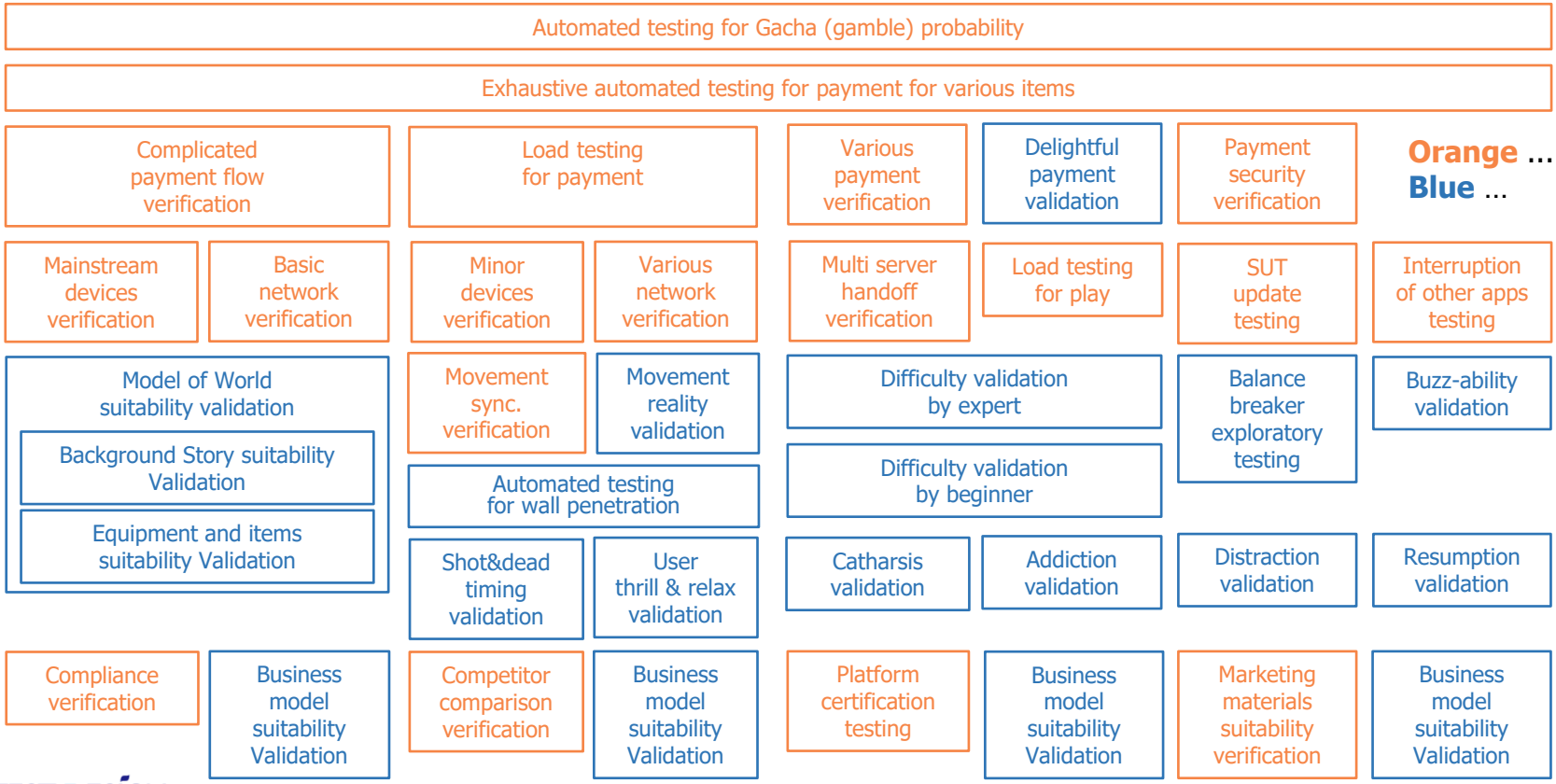
- **繰り返しのテストや回帰テストが必要なテスト観点、スプリントやイテレーションごとのテスト**を**テストサイクル**としてまとめる
  同じテスト観点を繰り返しているようでいて、実は意味が異なっていることがあるので注意する。

- **テストコンテナ間の関係がなるべく少なくなる（疎になる）ように**まとめる
  結合度を低く、凝集度を高く（同じテスト設計意図のテスト観点はまとめるように）設計する。

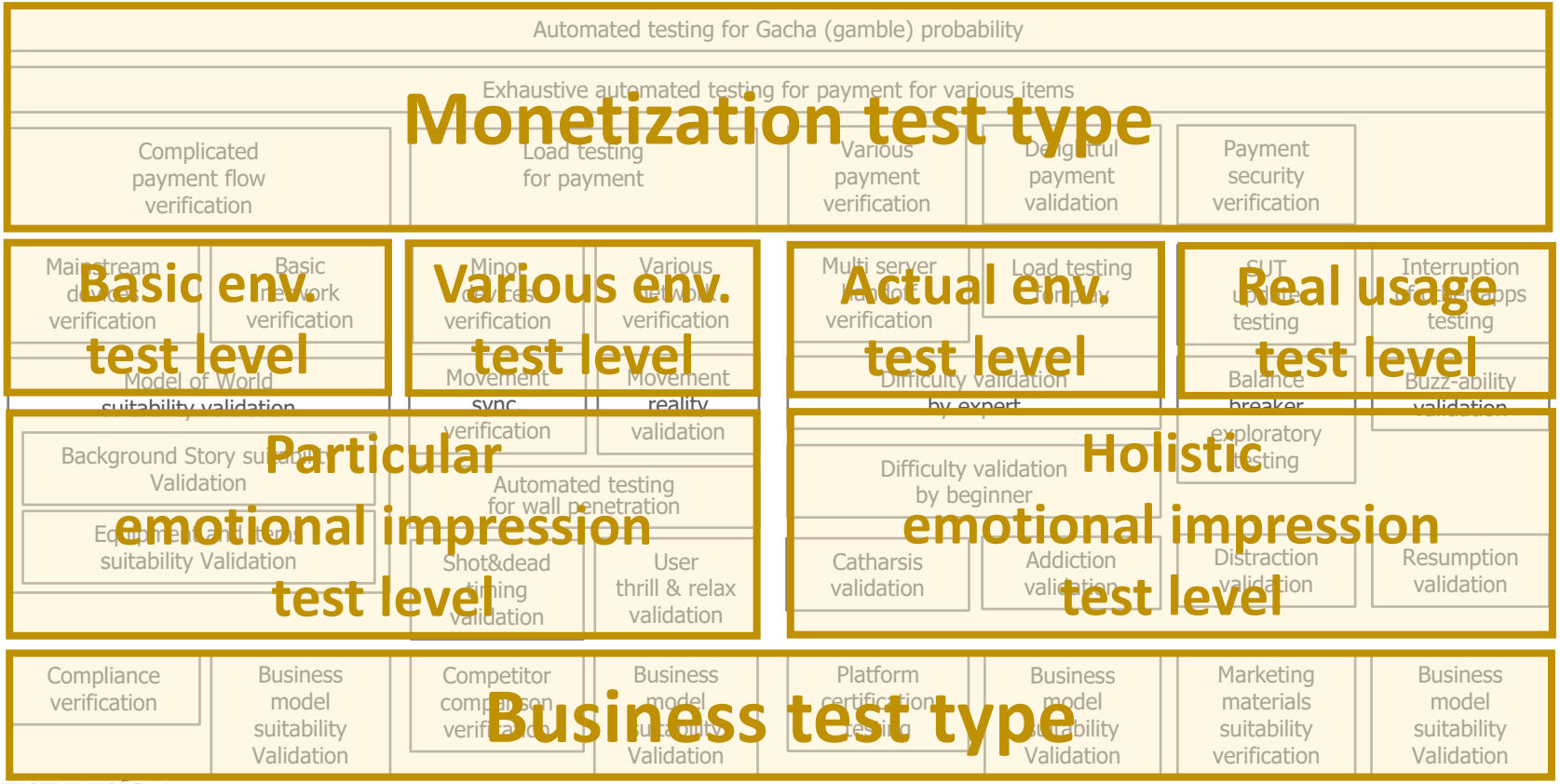
自分なりのテストコンテナを考えてよい



異なるテストコンテナが混在することもある



テストコンテナの粒度も自分なりでよい



テストアーキテクチャ設計における責務の分担

テストスイートの品質特性が満たされテストコンテナの**責務が適切に分担される**ように
テストアーキテクチャを設計する必要がある。

- アーキテクチャを検討するほど大規模で複雑なテストスイートでは、
  スイート自身の品質特性やコンテナの責務を考慮しなければならない
- テストコンテナには**責務を表す名前を適切につける**必要がある

テストスイートの品質特性やテストコンテナの責務には色々ある。

- Coupling（結合度）
- Cohesion（凝集度）
- Test design intention（テスト設計意図）
- Maintainability（保守性・派生容易性）
- Automatability（自動化容易性）
- Circumstance consistency（環境類似性）
- Development consistency（開発工程類似性）
- Describability（記述容易性）
- Execute velocity consistency（実行速度類似性）
- Stability / Change frequency（安定性／変更頻度類似性）
- Design type（設計種別類似性）

**適切に
「責務の分担」
がなされ、
適切な名前が
付いているだろうか？**

テストコンテナの責務

テスト設計意図

- テストには様々な設計意図があり、しばしば混在するため、（サブ）コンテナとして
  同じ設計意図に揃えた方がよい
  テスト設計意図は、そのテスト（コンテナ）の実行結果をどのように役立てたいか、である。

- テストコンテナの設計意図とテストケースの意図は**連動すべきである**
  テストケースの意図は、そのテストコンテナの設計意図を満たすように詳細化され、
  何らかのテスト観点に表現されるはずである。

設計種別類似性

- 設計種別類似性は、**設計者の頭の使い方の類似性**である
- 設計時の頭の使い方が異なる設計は**混ぜない方がよい**
  例：

テスト条件→引き起こされる期待結果を考えるテスト設計
vs
期待結果→引き起こすテスト条件を考えるテスト設計

動くことを証明したいテスト設計（肯定的テスト設計）
vs
動かないことを証明したいテスト設計（否定的テスト設計）

テストフレームモデリング

複数のテスト観点を同じグループにまとめて、テストケースの構造（スケルトン）を定義し、テスト詳細設計をしやすくする。

実際のテスト設計では、**複数のテスト観点をまとめてテストケースを設計したい**場合がある。
テストケースの構造をテスト観点の組み合わせで示すことで、複数のテスト観点によるテストケースを設計する。

「テストフレーム」
と呼ぶ

<<frame>>という
ステレオタイプの関連を用いる

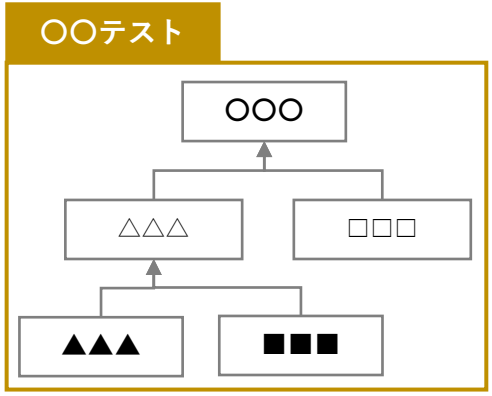
テスト条件＋テスト対象＋振る舞いをテストフレームとしている。
組み合わせテストを設計しようとしているわけではない点に注意する。

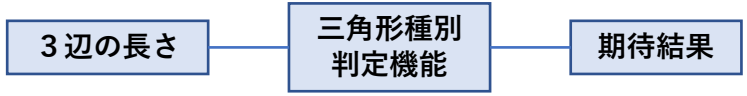
テストケースの構造の明示

「テストケースの構造」には単純なものから複雑なものまで色々あり得る。

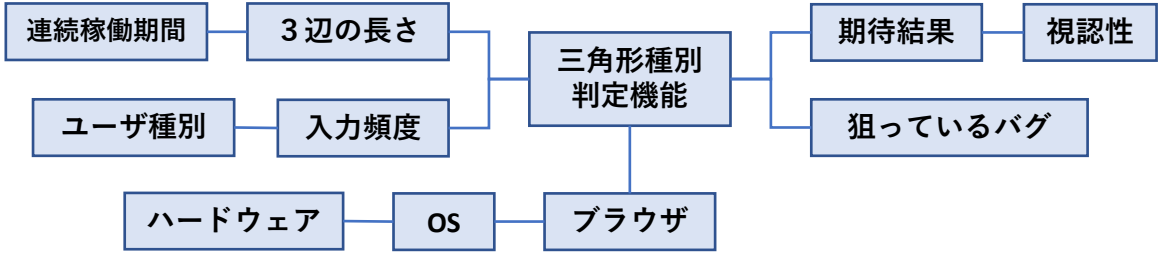
複雑なテストケースがバグを見つけやすいわけでもないことに注意する。

どのくらい複雑なテストフレームを組むかによって、テスト設計の良し悪しが変わる。
どういうトレードオフをどう考え、どういう意図でこの複雑さのテストフレームにしたのか、をきちんと説明できないといけない。





単純なテストフレーム例

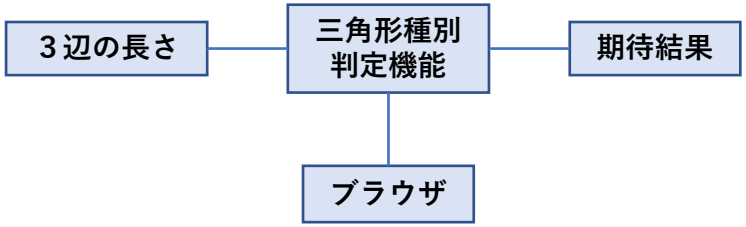


複雑なテストフレーム例

テストフレームはテストケースのスケルトン

テストフレームの**要素を見出しとして**表を作ると、テストケースになる。

- テストフレームをきちんと考えないと、「大項目―中項目―小項目」という見出しで整理した気になっていても、全く整理されていない、網羅しにくいテストケース表が出来上がる
- マインドマップやUMLツールなどを併用すると描きやすいが、全部一覧表やマトリクスでも構わない



テストフレーム例

テストケース表の例

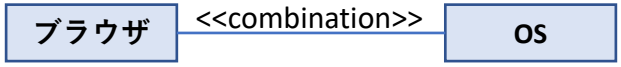
ID	3辺の長さ	ブラウザ	期待結果
1	(3, 4, 5)	Safari	不等辺三角形
2	(3, 3, 4)	Safari	二等辺三角形
3	(3, 3, 3)	Safari	正三角形
4	(3, 3, 6)	Safari	潰れて三角形にならない

組み合わせテストでもテストフレームに示そう

組み合わせてテストすべきテスト観点があったら、**テストフレームに明記する。**

- 明記しないと、組み合わせずに代表値や境界値だけのテストケースを作ってしまいがち
- 組み合わせが多い方が**良いわけでもバグを見つけやすいわけでもない**ことに注意する

どのくらい複雑な組み合わせのテストを設計するかによって、テスト設計の良し悪しが変わる。**どういうトレードオフをどう考え、どういう意図でこの複雑さの組み合わせにしたのか、**をきちんと説明できないといけない。



組み合わせのテストフレーム例

組み合わせのテストケース表の例

ID	ブラウザ	OS
1	Safari	iOS 9.3.2
2	Safari	iOS 8.4.1
3	Chrome	iOS 9.3.2
4	Chrome	iOS 8.4.1

探索的テスト

- **エキスパートによる非記述的なテストを探索的テストと呼ぶ**
  - テストエンジニアが**五感と経験（によって得られたパターン）と感受性を駆使する**ことで
    バグを出すことに集中しながら学習し、探索することで創造性を発揮するテストの方法である
  - 「ただ触るだけ」のモンキーテストやアドホックテストではない

- **チャーターとセッションでマネジメントを行う**
  - 大まかにどの辺をテストするか、どんなことをテストするか、を伝える
    「チャーター」を用いてマネジメントを行う
  - テストエンジニアが集中力を高めておける時間を１つの「セッション」という単位で捉え、
    マネジメントを行う

- **テストアーキテクチャ設計時に考慮しておく必要がある**
  - **チャーターをテスト観点として扱い、**いくつかのテストコンテナとしてまとめ、
    記述的テストと区別してテストアーキテクチャを設計するとよい
  - **記述的テストとの棲み分けをきちんと考慮して**おかないと、「その他のテスト」コンテナに成り下がる

探索的テストのポイント

- **探索的テストで何に着目するかはエキスパートに依存する**
  - バグの出そうな仕様やつくりに着目する（人がいる）
  - プロジェクトの状況やエンジニアの納得度に着目する（人がいる）
  - バグの出方に着目する（人がいる）
  - ユーザの使い方や、そう使うとは想像しない使い方に着目する（人がいる）
  - 出てほしくないバグや起こってほしくないハザード・アクシデントに着目する（人がいる）
  - ふるまいの一瞬の揺れなど怪しい動作に着目する（人がいる）

- **探索的テストは万能ではない**
  - **探索的テストと記述的テストは補完させるべきである**

    探索的テストだけでテストアーキテクチャを構成するのは極めて危険である。
    記述的テストを受注する第3者テスト会社でも50%近くを探索的テストで行う場合もある

  - **探索的テスト中にテストエンジニアがテスト対象について学習し、野生に還ったかどうか（感受性を鋭敏にしたかどうか）をマネジメントするとよい**
    - 網羅性や一貫性などを求めない方がよい

  - **探索的テスト後にテストエンジニアが仲間とそのセッションについておしゃべりするとよい**
    - 学習した結果や気付いた怪しい動作、言語化しにくい経験ベースのパターンを言語化するチャンス
    - 高い感受性で開発者と対話ができるように心理的安全を確保しておく必要がある

アジャイル開発におけるテストアーキテクチャ(1)

アジャイル開発でもTDLC（テスト開発ライフサイクル）やテストアーキテクチャに関する技術が必要になる。

- **ユーザの価値にフォーカスする**
  - ユーザの価値や、ストーリー、フィーチャーといったテストベースに関する非言語的理解が多いため、テスト要求分析を深く行わないとテスト設計に漏れが発生しやすくなる
- **変化・成長しやすいダイナミックなテストであることをキープする**

  計画やプロダクトの変化・成長にテストの変化・成長が追いつかねばならないため、
  テスト観点やテストコンテナのように**高い抽象度で可視化して分析・設計を行い、**
  保守性・派生容易性や自動化容易性を高めて**テストのダイナミックさを実現し続けること**がキーになる

  - 顧客との対話によってフィーチャーやユーザストーリーに留まらず、プロダクトの方向性や価値すらも変わっていくため、**抽象度の高い（モデルで可視化された）検討や意志決定がテスト（QA）エンジニアにも必要**になる
  - よいテストアーキテクチャがないと、フィーチャが高速かつ創発的に追加されていくのに釣られて**テストが乱雑になっていき、**開発の速度を落としてしまったりテストが追いつけなくなってしまう
  - よいテストアーキテクチャがないと、自動化できるテストと手動でしかできないテストを区別しにくくなるため、**手動テスト率が増えてしまい、**速度が落ちたり、開発に変化・成長したくない圧力をかけてしまう

アジャイル開発におけるテストアーキテクチャ(2)

アジャイル開発でもTDLC（テスト開発ライフサイクル）やテストアーキテクチャに関する技術が必要になる。

- **チーム全員でテスト（QA）を行う**
  - テストコンテナやテスト観点を可視化し整理し議論し納得感を共感することで、
    開発エンジニアが行うテストとテスト（QA）エンジニアが行うテストとの役割分担やその必要性・重要性を
    両者が理解・把握しきちんとダイナミックかつ有機的に連携できる
  - テスト（QA）エンジニアが最初から参画しレビューなども行うのであれば、
    テストアーキテクチャにはレビューなども含まれた「QAアーキテクチャ」になるべきである
- **チーム全員で考えてチーム全員で賢くなっていく**

  テスト観点やテストコンテナのように（複雑すぎない）モデルで可視化することで、
  テスト（QA）エンジニアが気にすべきことや（開発エンジニアが気付かず）テスト（QA）エンジニアが気付いたことを、プロダクトオーナーや開発エンジニアとの間で最初から双方向で共有しやすくできる。

イテレーション型テストアーキテクチャの設計(1)

イテレーションコンテナと非イテレーションコンテナの識別

- **イテレーションごとに分割統治できる独立フィーチャーと一部依存関係を持つ依存フィーチャー、基盤となるプラットフォーム、システム全体のテストに関するコンテナなどを識別する**

  通常はアーキテクトやPO、チームがQAが理解し設計しているはずなので、皆で対話しながら、テスト（QA）として品質を積み上げていけるかどうかをイメージしフィードバックする。

- **フィーチャーをまたいでテストしなくてはならないテスト観点を識別する**
  - UXなどふるまい的なテスト観点については、イテレーション間の一貫性などだけでなく、イテレーションが進むとどう変化するのかなどについても検討する
  - いくつか/全体のフィーチャーが揃わないとできないテスト観点については、フィーチャーごとのテストによって揃う前から品質を積み上げていけるように検討する
  - ソフトウェア設計上の依存関係については、アーキテクトや開発エンジニアと対話したり、過去のバージョンやそれまでのイテレーションのバグの出方、同種の製品で発生したバグの原因から推測する

- **運用中テストやシフトライトテストが必要かどうかを検討する**
  - 運用中に追加・変更されるフィーチャーやテスト観点の独立性を検討する
  - ある時点のデプロイより前に絶対にやらなくてはならないテスト観点、後に回せるテスト観点、運用中にテストすべきテスト観点などを識別する
  - 運用中テストやシフトライトテストで測定すべき項目を検討する

イテレーション型テストアーキテクチャの設計(2)

イテレーションコンテナの設計

- **各イテレーションのフィーチャーごとに必要なテスト観点やテストサブコンテナを設計する**

  どのフィーチャーを開発するかはイテレーションごとにダイナミックに決まることが多いので、イテレーションコンテナに含まれるだろうフィーチャー群（フィーチャープール）としてグルーピングしておくと、テストアーキテクチャ設計がしやすくなる。

- **イテレーションスリップやイテレーションフォークをどう扱うかについて検討しておく**
  - イテレーションが進んで回帰テスト量が増えるなど
    手動テスト工数やテスト自動化工数が増えてしまった際にどうするか、に正解はない
  - 最初から意図的にイテレーションスリップやイテレーションフォークを発生させる場合もある
  - なし崩し的に発生したイテレーションスリップやイテレーションフォークを回収するのは容易ではない
  - イテレーションプールごとに扱いが異なる場合もある
  - イテレーションスリップやイテレーションフォークが発生しないようテスト自動化のためのテストコンテナをわざわざ設計する場合もある

イテレーション型テストアーキテクチャの設計(3)

非イテレーション/準イテレーションコンテナの設計

プラットフォームやシステム全体に関するテストコンテナ、依存フィーチャーに関するテストコンテナ、フィーチャーまたぎのテストコンテナなどに含まれるサブテストコンテナ・テスト観点や、そのテストコンテナに関するテスト設計やテスト実行の頻度を設計する。

QAチームが別にテストしたり、そのために新たなイテレーションを始めることもあれば、
バラしてイテレーションに組み込むこともあるので、チーム外QAやPO、SMなどと対話しながら検討する。

テストコンテナ間の設計

- イテレーションコンテナ内のテスト（サブ）コンテナ間の依存関係を識別し、組み合わせや順序関係を設計する

- バグの出方などからイテレーションコンテナ間に**「依存関係がある」**と感じられた場合は、すぐにアーキテクトや開発エンジニアと対話して一緒に取り扱いを検討する

フルスタックQAエンジニアを目指そう

これからは（特にアジャイル開発の場合）
テスト（QA）エンジニアであっても、開発の知識がフルスタックで必要になる

- イテレーション
- フィーチャーやユーザーストーリーなど
- リファタクタリング
- マイクロサービスアーキテクチャ（MSA）や疎結合、古くはカプセル化や結合度・凝集度
- 各種プラットフォームやアーキテクチャ
- クラウド・エッジ/サーバ・フロントエンドのバランス
- devOps / シフトライト（カオスエンジニアリング）
- CI/CDやテスト自動化の技術・プラットフォーム
- TDD（テスト駆動開発）を何らかの品質を「保証」するためのテストとみなすのか否か

など

もちろん、

テストの技術もTDLC（テスト開発ライフサイクル）の技術もQAの技術もドメインの知識も必要になってくる

4. TDD・TI（テスト詳細設計、テスト実装）

Test Detail Design　　Test Implementation

テスト観点からテスト値を網羅する

- **テスト値の網羅は以下の手順で行う。**
  1. テスト観点がどのタイプの「テストモデル」なのかを見極める
  2. 網羅基準（カバレッジ基準）を定める
  3. 定めた網羅基準でテストパラメータを網羅するようにテスト値を設計する

- **テスト詳細設計は機械的に行っていくのが基本である。**
  - 可能な限り自動化すると、Excelを埋める単純作業を撲滅できる
  - 機械的にできないところは、**網羅基準が曖昧だったり、テスト観点が隠れていたり**する

テスト詳細設計モデル

テスト詳細設計モデルのタイプには４つのタイプがある。

- **範囲タイプ**
- **一覧表タイプ**
- **マトリクスタイプ**
- **グラフタイプ**

テスト値には２種類あるので注意する必要がある

- **直接テスト値**：テスト値が直接テスト手順の一部（テストデータ）として実施できるもの
  例）３辺の長さ（3, 3, 3）
- **関節テスト値**：テスト値からさらにテストデータを導く必要があるもの
  例）制御パス

制御パスを網羅して、そのあとにそれぞれの制御パスを通す
テストデータを作成する。

テストパラメータやテスト詳細設計モデルを特定せずに
闇雲にテストケースを挙げるのは、質の高いテストとはいえない。

範囲タイプのテストモデルの網羅

範囲タイプのテスト詳細設計モデルは、

ある連続した範囲を意味するテスト観点を網羅するときに用いる。

例）
辺の長さ（0以上65535以下の整数値）
範囲のことを「同値クラス」と呼び、同値クラスを導出する技法を同値分割と呼ぶ。

範囲タイプのテスト詳細設計モデルの網羅基準は以下のとおりである。

- **全網羅**
  例）0, 1, 2, 3, 4, 5, 6, … 100, … 10000, … 65534, 65535
  漏れはないが、テスト値が膨大になるため現実的ではない。
- **境界値網羅**
  例）0, 65535, -1, 65536
  - **境界値分析や境界値テスト、ドメインテストなど独立した技法として説明されることが多い**
  - **有効境界値だけでなく無効境界値も忘れないこと**
  - **範囲が開いている（上限や下限が定まっていない）時は自分で定める必要があるが、よく検討しないと漏れが発生しやすくなる**
- **代表値網羅**
  例）3
  テスト値は少なく収まるが、漏れが発生する。

一覧表タイプのテストモデルの網羅

一覧表タイプのテスト詳細設計モデルは、

複数の要素からなる集合を意味するテスト観点を網羅するときに用いる。

例）
ブラウザ

一覧表タイプのテスト詳細設計モデルの網羅基準は以下のとおりである。

- **全網羅**
  例）Chrome, Firefox, Safari, Internet Explorer, Edge
  漏れはなく、通常はテスト値もそれほど膨大にならないが、組み合わせになったら無視できない程に
  テスト値が多くなってしまう。
- **境界値網羅**
  例）一番昔にリリースされたブラウザと一番最近にリリースされたブラウザ
  よく検討しないと漏れが発生しやすくなる
- **代表値網羅**
  例）Chrome
  テスト値は少くなく収まるが、漏れが発生する。

マトリクスのテストモデルの網羅

マトリクスタイプのテスト詳細設計モデルは、

2つ（以上）のテスト観点の組み合わせを網羅するときに用いる。

例）
OSとブラウザの組み合わせ
組み合わせたいテスト観点の数をここでは段数と呼ぶ。

一覧表タイプのテスト詳細設計モデルの網羅基準は以下のとおりである。

- **全網羅**
  例）Chrome, Firefox, Safari, Internet Explorer, Edge
  漏れはないが、掛け算によりテスト値が膨大になるため現実的ではない。
  禁則（無効な組み合わせ）に注意する。
- **N-wise網羅**
  N+1以上の段数の組み合わせの網羅を意図的に無視することによって、現実的な数で
  Nまでの組み合わせを全網羅する方法である。N＝2のときをPairwiseと呼ぶ。
  Nまでの組み合わせの網羅で十分かどうか検討しないと漏れが発生する。
- **代表値網羅**
  テスト値は少くなく収まるが、漏れが発生する。

グラフタイプのテストモデルの網羅

グラフタイプのテスト詳細設計モデルは、

丸と線で図が描けるようなテスト観点を網羅するときに用いる。

例）
プログラムのロジック、状態遷移図、シーケンス図、ユーザシナリオ、地下鉄の路線図
折れ線グラフや棒グラフのグラフではない。
丸と線で描ける図を（フロー）グラフ、丸をノード、線をリンク、丸と線による経路をパスと呼ぶ。
制御フローパステストの場合は、if文の複合判定条件の真理値表の網羅と組み合わせることがある。（C2網羅）

グラフタイプのテスト詳細設計モデルの網羅基準は以下のとおりである。

- **全パス網羅**
  漏れはないが、テスト値が膨大になるため現実的ではない。
- **MC/DC（Modified Condition / Decision Coverage）**
- **Nスイッチ網羅**
- **リンク網羅（0スイッチ網羅，C1網羅）**
- **ノード網羅（C0網羅）**

  } **テスト漏れが発生する。**

- **代表パス網羅**
  テスト値は少なく収まるが、漏れが発生する。

テスト実装（TI）

テストケースが生成できたら、**テスト実装**を行う。

- テスト対象のシステムやソフトウェアの仕様、テストツールなどに合わせて**テストケースをテストスクリプトに具体化**していく
- テスト実装は、テスト対象のUI系の仕様や実行環境、ユーザのふるまいに関する**ノウハウ**が必要である

集約

- テスト実施を効率的に行うため、**複数のテストケースを１つのテストスクリプトにまとめる作業**を**「集約」**と呼ぶ
- 同じ事前条件や同じテスト条件、同じテストトリガのテストケース、あるテストケースの実行結果が**他のテストケースの事前条件やテスト条件になっている**場合は集約しやすい

キーワード駆動テスト

- “クリック”といった**自然言語の「キーワード」で自動化テストスクリプトを記述することにより、保守性を高めたり自動生成をしやすくしたりする技術**である
- **テスト観点とキーワードを対応させる**と、キーワード駆動テストを実現しやすくなる

5. テスト開発のTIPS

テスト開発プロセスを現場に適用する際のポイント(1)

- **テストエンジニアのマインドを、「表を埋める単純作業」から**クリエイティブなモデリング**に変えていく**
  - 単純作業だと思ってしまっている限り、**モチベーションもスキルも向上しない**
  - **単純作業の部分は自動化する**ように技術を高めていく

- **できるところから小さく始めて小さく回し、効果を実感しながら進めることが重要**
  - まずは腕の良いエンジニアに、テストスイートのごく一部から適用してもらうとよい
  - 初めはあまり網羅性を意識せず、テスト要求モデルをつくり、**自分たちのテストの良さ悪さを実感**してみる
  - 常にテスト開発プロセスの良さを**エンジニアが実感しながら**進めるようにする
  - 自分たちに必要だができていないことを明らかにして、さらなる導入を進める

- **モデルをコミュニケーションの道具として使う**
  - 同じプロジェクトの異なるテストエンジニア同士で、役割の違うエンジニアと、そして顧客との**コミュニケーションの道具**としてテスト観点を用いるとよい
  - 網羅しているかどうか、質の高いモデルかどうかなどを**皆で「納得」することがとても重要**である

テスト開発プロセスを現場に適用する際のポイント(2)

- **テスト要求分析・テストアーキテクチャ設計でのモデリングスキルを高める**
  - 観点の意味をブレないようにする
  - 詳細化の種類を明示し整理する
  - テスト観点の抽象度をきちんと意識し、詳細度が丁寧に落とし込まれるようにする
  - 単に適当に組み合わせるのではなく、関連が発生する要因を推測する
  - テストのデザインパターンを抽出し蓄積する
  - テストアーキテクチャスタイルを蓄積する
  - 自分たちが気にしている・気にすべきテスト品質特性を明らかにする

- **テスト開発プロセスの導入と合わせて**
  リバースエンジニアリングで現状のテストを改善できるとよい
  - テストプロセスそのものの改善や、テストをきっかけとした開発プロセスの改善の道具として
    テスト開発プロセスを活用できるとよい

6. 過去の応募作のポイント

応募作のテストアーキテクチャ(1)

過去の応募作を分析すると、テストアーキテクチャの**構成要素は様々**だった。

- 多くのチームが以下のような**テスト観点**を抽出していた
  - **テスト条件、ふるまい（期待結果）、見つけたいバグ**
  - **テスト対象（の要素）、Test item**

- 多くのチームが２つの書き方で**構造**を記述していた
  - **モデルっぽい書き方**
  - **表（マトリクス）っぽい書き方**

テスト対象（の要素）

テストすべき
テスト条件／ふるまい／バグ

モデル風の記法

表（マトリクス）風の記法

		テストすべき条件／ふるまい／バグ		
		C1	C2	C3
テスト対象（の要素）	P1	P1をC1でテスト	P1をC2でテスト	P1をC3でテスト
	P2	P2をC1でテスト	P2をC2でテスト	P2をC3でテスト
	P3	P3をC1でテスト	P3をC2でテスト	P3をC3でテスト

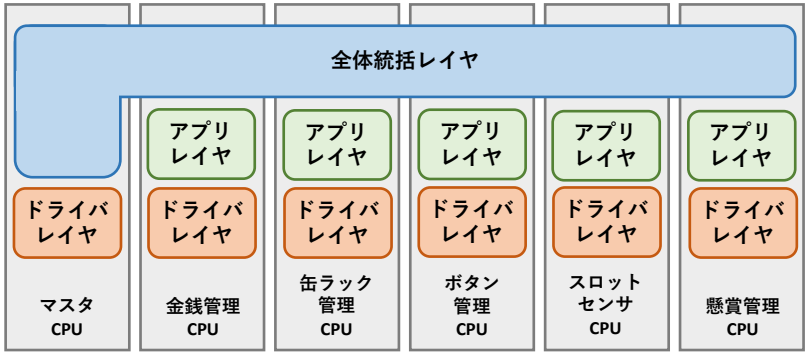
応募作のテストアーキテクチャ(2)

過去の応募作を分析すると、**様々なテストアーキテクチャ**が構築されていた。

- （伝統的）テストタイプ型アーキテクチャ
- レイヤ型アーキテクチャ
- フィルタ型アーキテクチャ
- 複合型アーキテクチャ
- アジャイル開発に対応したテストアーキテクチャ？

レイヤ型アーキテクチャ

フィルタ型アーキテクチャ

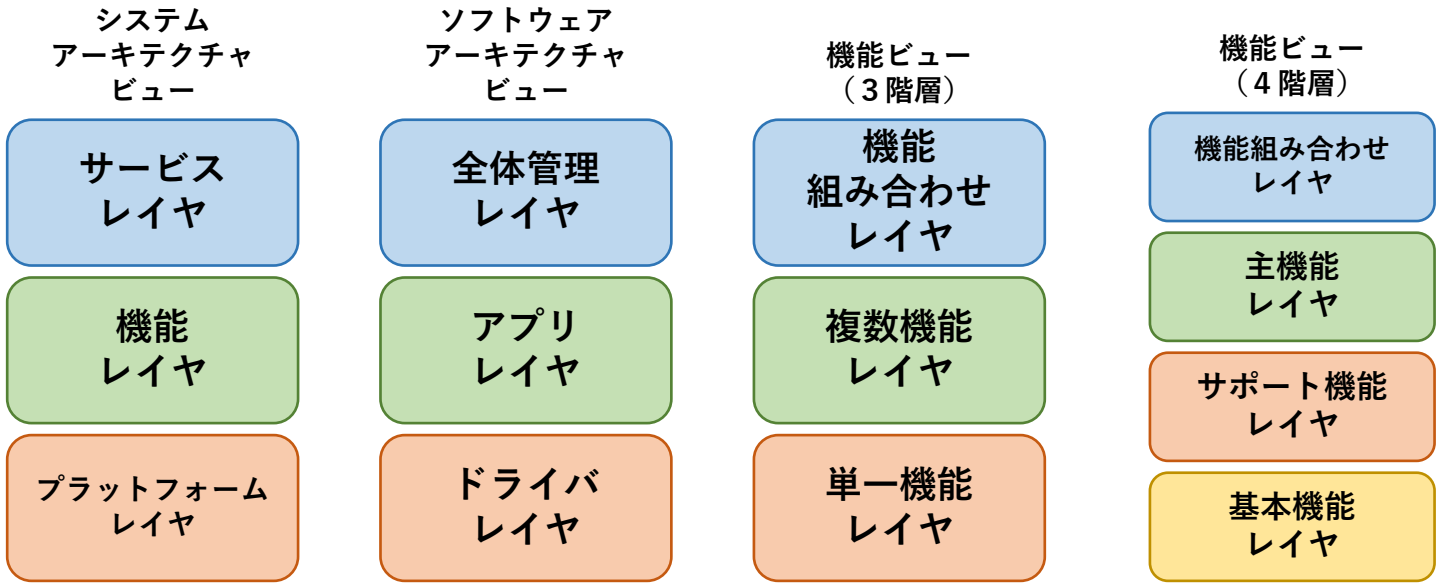


複合型アーキテクチャ

応募作のテストアーキテクチャ(3)

一見似たようなレイヤ型アーキテクチャであっても、**実は異なっていた。**

ビューやレイヤ数も様々だった。



過去の応募作の審査で気になった点(1)

- **設計根拠を明示してほしい**
  - 特にテストアーキテクチャ設計の設計根拠を論理的に記述してほしい
    「なんかよく分からないけどこんな風に配置してみました」は設計したことにならない。
    「このような文脈だからこのような品質特性を達成する必要があるので、
    これこれこうしたコンテナの責務に分割されなくてはならない」、のように書けるとよい。

    機能重要度・リスクベースドや実行順序だけを考えたテストアーキテクチャはもう平凡である。
    最初に機能で分割することが果たしてよいテストアーキテクチャなのか、を熟考すべき。
  - トレードオフの種類と判断根拠や、対案を却下した理由を記述してほしい
    多面的な／複数のトレードオフが必要になるはずである。
    異なる判断に基づいた複数の設計案が示されていると素晴らしい。
  - トレーサビリティやカバレッジは、読めば確保できていることが分かるようにしてほしい

- **テスト観点を多面的に考慮し、それらの関係と根拠を明示してほしい**
  テスト対象は一般に、テスト対象ごとに固有なものも含めて多面的なテスト観点が必要になる。
  また、テスト対象ごとにテスト観点間の関係も異なる。したがって、多面的にテスト観点を考慮し、
  それらの関係をきちんとモデリングし、その根拠を明示してほしい。

過去の応募作の審査で気になった点(2)

- **スコープとその根拠を明示してほしい**
  設計範囲とそのために考慮すべき範囲、考慮する必要がない範囲と設計しない範囲を根拠とともに明示してほしい。**仕様書にあるから考慮しました、設計したところがスコープです、ではない・・・**

- **これで本当に俯瞰しやすいのかどうかをきちんと検討してほしい**
  巨大なマトリクス、ごちゃごちゃした線、責務が重複した箱・・・
  だからといって単にシンプルにすればいいわけではない。

- **自分たちのテスト開発プロセスの結合度と凝集度に目を向けてほしい**
  TRA、TAD、TDD、TIの各フェーズを自分たちなりに定義し直したり、新たに定義したりするのは問題ないが、なぜそのようにフェーズを決めたのかをきちんと説明してほしい。

- **最新のプラクティスを取り入れるのであればきちんと取り入れてほしい**
  それぞれのプラクティスになるコンセプトや強み、実際に直面する弱みなどをきちんと熟考して、それぞれのプラクティスがテストによってより良くなるように、最新のテスト開発方法論を構築してほしい。
  とにかくアジャイルにしてみた、とにかく自動化してみた、とにかく探索的テストやります、ではそれぞれのプラクティスに対応したことにならない。

考えろ、考えろ、そして考えろ

- **まずは、一つ一つきちんと、**
  自分たちが扱っているものや、行っていることの意味を熟考して、
  それらを言語化して論理的に順序立てて説明してほしい
  - 文章だけでなく図やモデルを使っても構わない。
  - 他の文章や規格から引用する際は、それらの意味や妥当性、引用してよい理由をきちんと熟考して説明する。

- **そして、常に全体像を俯瞰して捉えてほしい**
  - 全体像が分からないと、そこかしこで思い込みや矛盾、部分最適が発生する。
  - 全体像から、詳細なテストケースやテスト手順までを段階的に詳細化する必要

- **ただし、言語化や論理を重視するあまり、直感を捨ててはならない**
  - 直感も大事な技術力である。

  直観で判断などを行ったところは、「ここは直感で〇〇した」と書いておけばよい。
  後でリスクヘッジをしたり、振り返りや改善をしたり、流用したときに説明可能になることを期待する。

テストはエンジニアリングである

テスト設計コンテストは、業務での「熟考度」や「俯瞰度」を反映している。

- 言語化して論理的に説明できなかったり、俯瞰できないテスト設計は
  レビュー不能であるし、そもそも**テスト設計できていない見込みが高いし、**
  何より自分たちが納得してテストを勧められていないはずである

- テスト設計コンテストOpenクラスは、
  自社の最先端の技術開発のオープンイノベーションの場である

- テストはソフトウェア開発の（重要な）一部であり、**テストはエンジニアリング**である

テストのことだけではなく、
ソフトウェア開発に必要となる
思考力やモデリング力を身につけよう

ご清聴ありがとうございました