テスト設計チュートリアル テスコン編 '21

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

- TDLC（テスト開発ライフサイクル）とは
- TRA（テスト要求分析）
- TAD（テストアーキテクチャ設計）
- TDD・TI（テスト詳細設計、テスト実装）
- テスト開発のTIPS
- 過去のテスト設計コンテストの応募作のポイント

テスト開発プロセスが必要である

- 大規模化・複雑化する一方で
  高品質・短納期も求められるソフトウェア開発のために、
  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が
  社会的な急務になってきている
  - 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
- しかしテスト計画やテスト戦略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とが混同されている
    - » 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る
  - 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沌として一体的に扱われている
    - » テスト(ケース)開発という用語はほとんど使われず、
      テスト計画、テスト仕様書作成、テスト実施準備といった用語が使われている
    - » テスト戦略という用語もイマイチよく分からない
- そこで「テスト開発プロセス」という概念が必要となる
  - テストケースを開発成果物と捉え、開発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
    - » 本来はテスト実施以降やテスト管理のプロセスも必要だが、今回は省略する

テスト設計とテスト開発は何が違うの？

- いきなりコーディングすることと、ソフトウェア開発との違いと同じである
  - コーディングは、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
    - »（従来の）テスト設計も、テスト（詳細）設計技法と文書フォーマットを分かっていればできる
    - »経験豊富でセンスのある人が規模の小さいプログラムを書くのであれば別に構わない
  - ソフトウェア開発は、要求を把握し設計原則を理解・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
    - »テスト開発も、テスト要求を把握しテスト設計原則を理解・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
    - »これまで皆の経験とセンスを結集して規模の大きなソフトウェアを開発することである
    - »経験とセンスを結集するために、様々なソフトウェア工学上の概念や技術、方法論や、モデリングスキル、抽象化／パターン化能力などが必要となる
  - 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
    - »キーワード駆動テストやモデルベーステスト、E2Eテストまで含めたフルCI/CDも目指せるようになる

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

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
  ソフトウェア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	＝	手動／自動化テストスクリプト（テスト手順）

旧来のテストプロセスでは粗すぎる

テスト項目の抽出

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

テスト観点とテストケースとテスト手順を区別する

- テスト観点とテストケースとテストスクリプトをきちんと区別する
  - テスト観点
    - » 何をテストするのか（テストケースの意図）のみを端的に記述したもの
      - 例）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などのみを特定したもの
      - 例）(1,1,1), (2,2,2), (3,3,3)…
    - » 通常は、網羅基準に沿って特定される値など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
  - テストスクリプト（テスト手順）
    - » 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
      - 例）1. PCを起動する 2. マイコンピュータからC:¥sample¥Myers.exeを起動する…
    - » 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一つのテストスクリプトにすることもある
- 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によって、
  テスト観点のみをじっくり検討することができるようになる

上流工程でテスト観点を構成要素としてモデリングを行う

- 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
    - » 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
  - 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、リモデリングする覚悟で行う
  - 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

- 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
  - 信号の電気的ふるまい
    - » 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

- 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

チュートリアルの流れ

- TDLC（テスト開発ライフサイクル）とは
- TRA（テスト要求分析）
- TAD（テストアーキテクチャ設計）
- TDD・TI（テスト詳細設計、テスト実装）
- テスト開発のTIPS
- 過去のテスト設計コンテストの応募作のポイント

テスト要求分析（TRA）

- テスト要求の源泉の準備
  - 入手できる限り（できるなら）、要求の源泉を準備する
- テスト要求の獲得と分割
  - 要求の源泉から要求を獲得する
  - テストケースを導くエンジニアリング的要求と
    テストケースを導かないマネジメント的要求に分ける
    - » マネジメント的要求は品質リスクに反映させる
- テスト要求モデルの構築と納得
  - エンジニアリング的要求を基に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リファインする

TRA - テスト要求の源泉の準備

- 入手できるなら、テスト要求の源泉を準備する
  - テスト要求の源泉の例）
    - » 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
  - 一つ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
    - » あくまで入手できないなら仕方が無い、という意味合いであり、 要求の源泉が無くても大丈夫、という意味ではないことに注意する
  - テスト要求は、システムやソフトウェアの要求のサブセットではなく、 むしろ広い情報が必要となる
    - » 例) テスト対象の途中経過に関する情報

TRA - テスト要求の獲得と分割

- テスト要求の源泉からテスト要求を獲得する
  - ヒアリングや観察など様々な手段がある
- 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
    - » システムの完成像への要求の例：
      - 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の例：
      - 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
        - ユーザがミスをしそうなところ
        - 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
  - マネジメント的テスト要求：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など）、
      ツール利用可否（ツールの種類とライセンス、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

TRA - テスト要求モデルの構築と納得

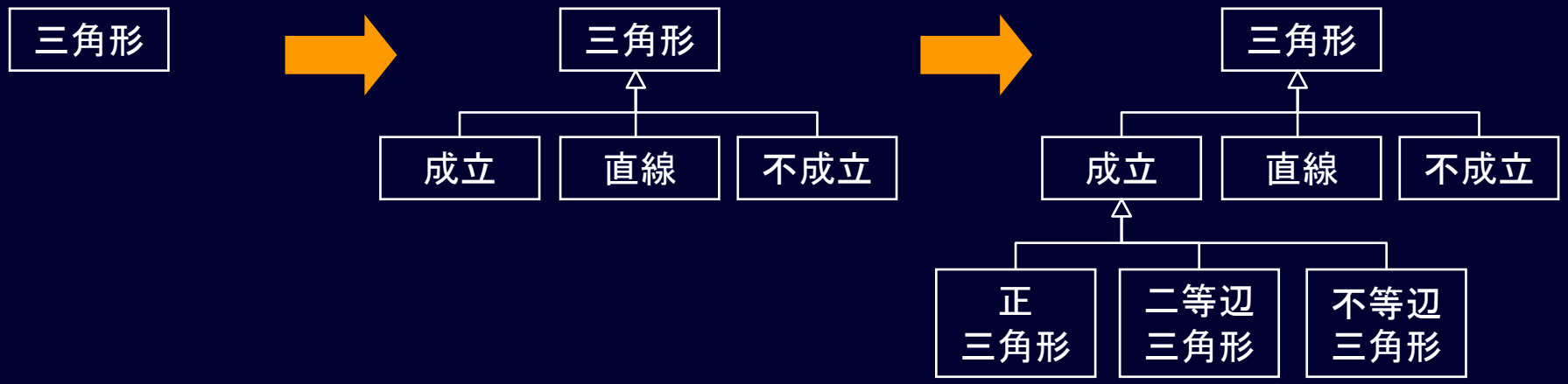
- テスト要求モデルの構築と納得
  - テスト要求モデルを構築し、
    自分たちとステークホルダーが納得するまでリファインする
  - エンジニアリング的テスト要求を基に、テスト観点を列挙していく
  - 階層構造やマトリクス、一覧表などでテスト要求モデルを記述する
  - エンジニアリング的要求を基に、テスト観点を詳細化したり関連を追加していく
  - モデルのリファインを行う
    - » MECE分析・親子関係分析による網羅化
    - » 整理
    - » 確定
  - 自分たちで十分に充実したと実感できたら、そのテスト観点図を囲んで、
    ステークホルダーが納得するまで一緒にリファインする
    - » 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きゃいけないんだ、
      という実感を持ってもらうことが重要である

TRA - テスト観点を挙げる時の注意点

- 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(もしくは存在せず)、(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を挙げたり、期待結果やその観測に関するテスト観点だけを挙げたり、
    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」だから期待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とする
  - 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である
    - » 見逃したバグを見つけう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？

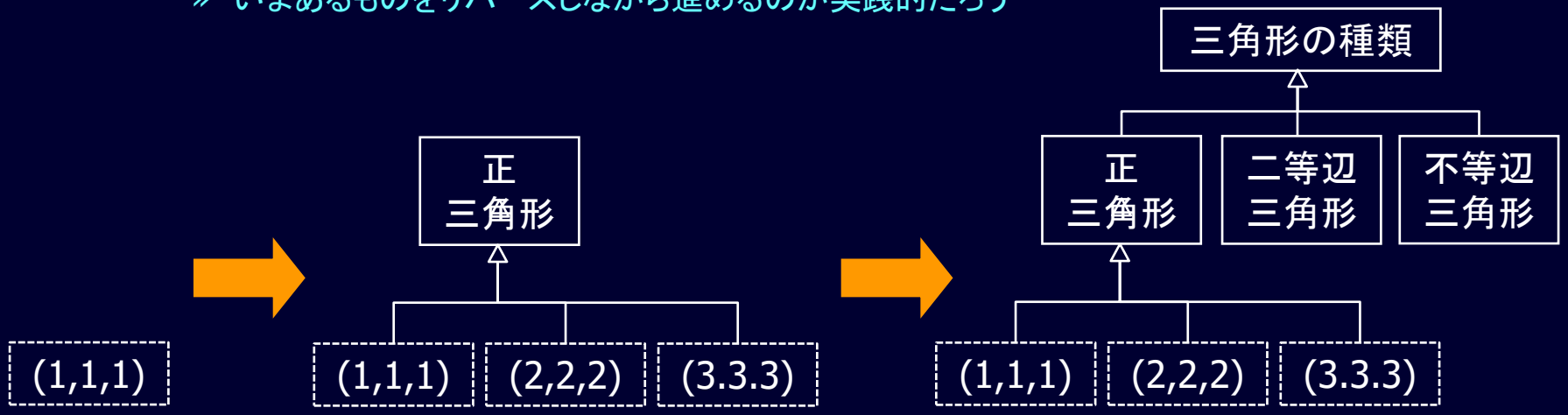
TRA - テスト観点モデリングの2つのアプローチ

- 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
    - » 三角形のテストには、三角形の構造に関する観点が必要だ
    - » 三角形の構造には、成立、直線、不成立の3つがあるな
    - » 成立する場合には、正三角形、二等辺三角形、不等辺三角形があるな
  - 実際には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する
    - » いまあるものをリバースしながら進めるのが実践的だろう



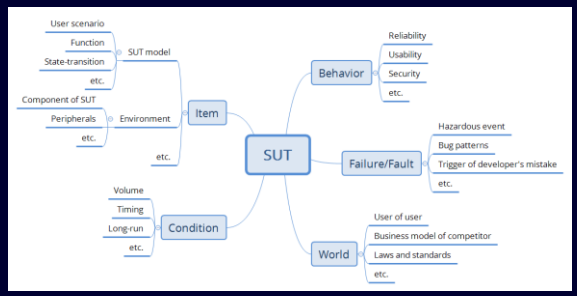
TRA - テスト観点モデリングの2つのアプローチ

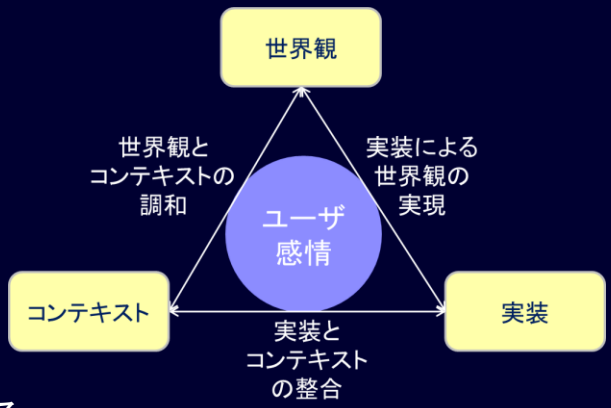
- ボトムアップ型
  - まず思いつくところからテストケースを具体的に書き、
    いくつか集まったところで抽象化し、テスト観点を導き出していく
    - » (1,1,1) なんてテスト、どうかな
    - » (2,2,2) とか、(3,3,3) もあるな
    - » これって要するに「正三角形」だな
    - » 他に似たようなのあるかな、う～ん、二等辺三角形とかかな
  - 実際にはトップダウンとボトムアップを循環させながらモデリングする
    - » いまあるものをリバースしながら進めるのが実践的だろう



TRA - 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いいか、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
  - 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モデル
      - Condition / Item / Behaviour / Fault / World
    - » 三銃士モデル
      - 世界観・コンテキスト・実装＋ユーザ感情
    - » システム構想モデル
      - そのシステムの構想がそのまま全体像になる（ラルフチャートなど）
- 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





TRA - テスト観点モデルのリファイン

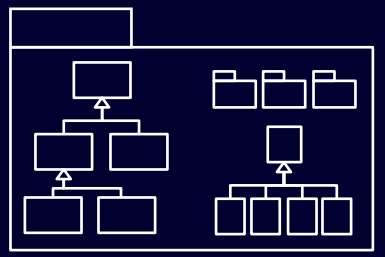
- 質の高いモデルにするために様々なリファインを行う
  - 網羅化：MECE分析
    - » 子観点がMECEに列挙されているかどうかをレビューし、不足している子観点を追加する
    - » MECEにできない場合、必要に応じて「その他」の子観点を追加し非MECEを明示する
    - » 子観点をMECEにできるよう、適切な抽象度の観点を親観点と子観点との間に追加する
  - 網羅化：親子関係分析
    - » MECE性を高めるために、テスト観点の親子関係を明示し子観点を分類する
  - 整理
    - » 読む人によって意味の異なるテスト観点を特定し、名前を変更する
    - » テスト観点や関連の移動、分割、統合、名前の変更、パターンの適用、観点と関連との変換、観点と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、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

チュートリアルの流れ

- TDLC（テスト開発ライフサイクル）とは
- TRA（テスト要求分析）
- TAD（テストアーキテクチャ設計）
- TDD・TI（テスト詳細設計、テスト実装）
- テスト開発のTIPS
- 過去のテスト設計コンテストの応募作のポイント

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
  - 1) 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
      - グループの例：テストレベルやテストタイプ、テストサイクル
      - テストレベルやテストタイプに何が含まれるかを、テスト観点を用いて設計する
      - テスト観点のグループを、VSTePではテストコンテナと呼ぶ

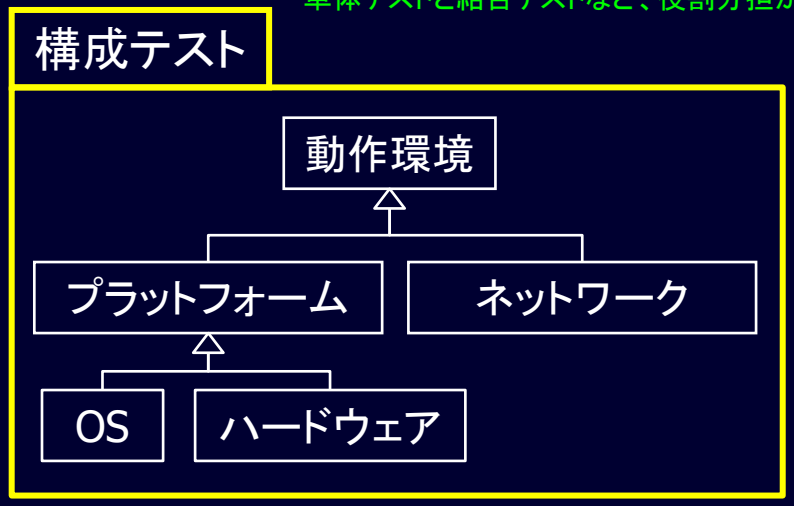


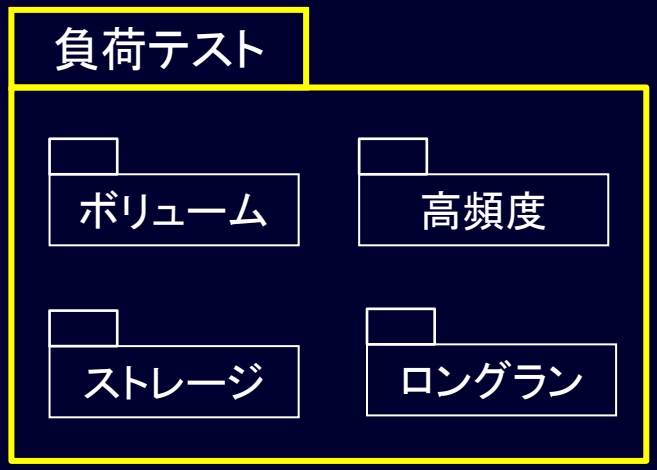
    - » グルーピングすると粒度が粗くなるので、
      テストスイートの全体像の把握や全体に関わる設計がしやすい
    - » 自社で定められているテストレベルやテストタイプの趣旨を理解し、きちんと全体像を設計できるようになる
    - » テストにも保守性など特有の「テストスイートの品質特性」や「テストコンテナの責務」があるが、
      どのような品質特性や責務を重視するかによって異なるテストアーキテクチャとなる
  - 2) テストフレームモデリング
    - » 複数のテスト観点を同じグループにまとめて
      テストケースの構造（スケルトン）を定義し、
      テスト詳細設計をしやすくする
      - テストケースのスケルトンを、VSTePではテストフレームと呼ぶ

  * コンテナモデリングとフレームモデリングはどちらから始めても構わない
    - 現実的には反復的に進めることになるだろう

TAD - テストコンテナとは

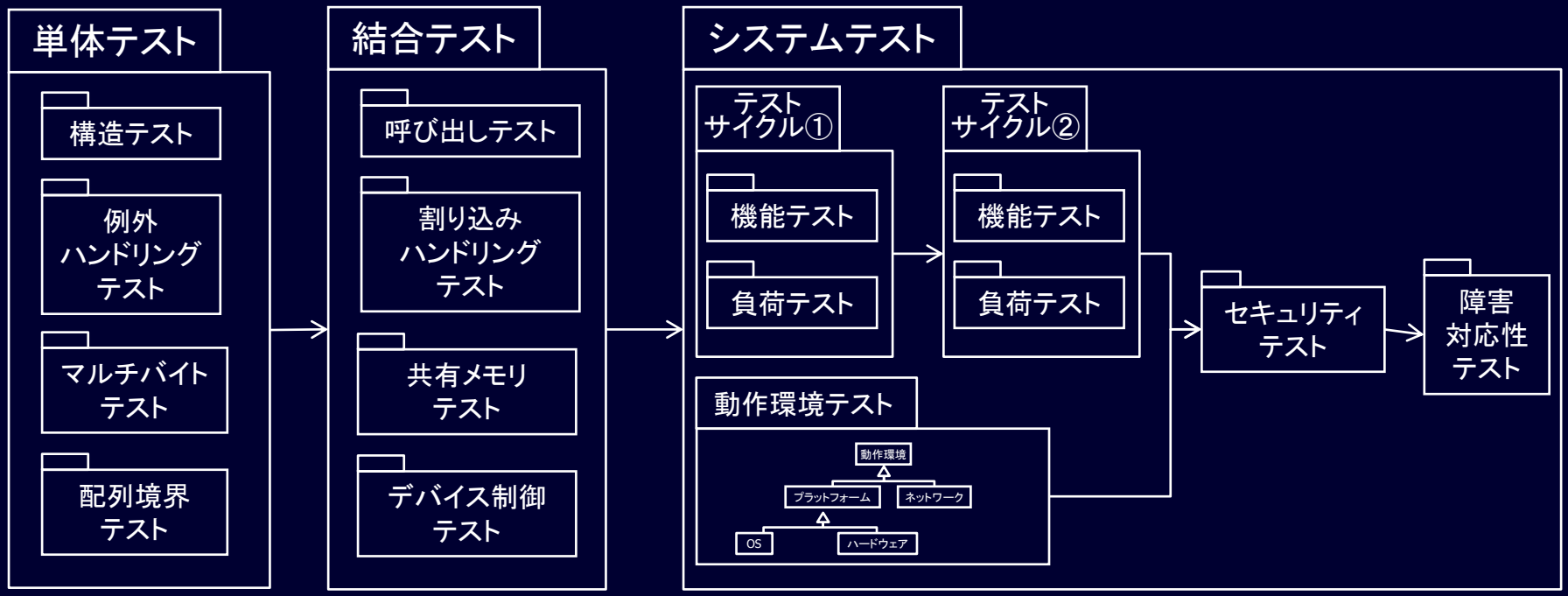
- 大規模なテスト設計では、
  複数のテスト観点をグルーピングして扱いたい時がある
  - 複数のテスト観点をグルーピングしたものをテストコンテナと呼ぶ
    - » テストタイプやテストレベル、テストサイクルなどを表現する
      - 機能性テストと機能テスト、機能テスト段階など、
        タイプなのかレベルなのかサイクルなのか分からないものの区別に時間を取られないで済む
  - テストコンテナは包含関係を持つことができる
    - » テストコンテナに含まれるテスト観点を比べると
      テストコンテナ間の役割分担を明確に把握できる
      - 負荷テストと性能テストなど、違いがよく分からないテストタイプも区別できる
      - 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も区別できる





TAD - テストコンテナモデル

- 大規模なテスト設計では、
  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
  - テストコンテナ（レベルやテスト、サイクル）を用いるとテストアーキテクチャを俯瞰できる
    - » 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
    - » 保守や派生のしやすいテストが可能になる

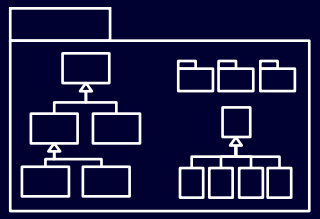


TAD - テストコンテナモデリング

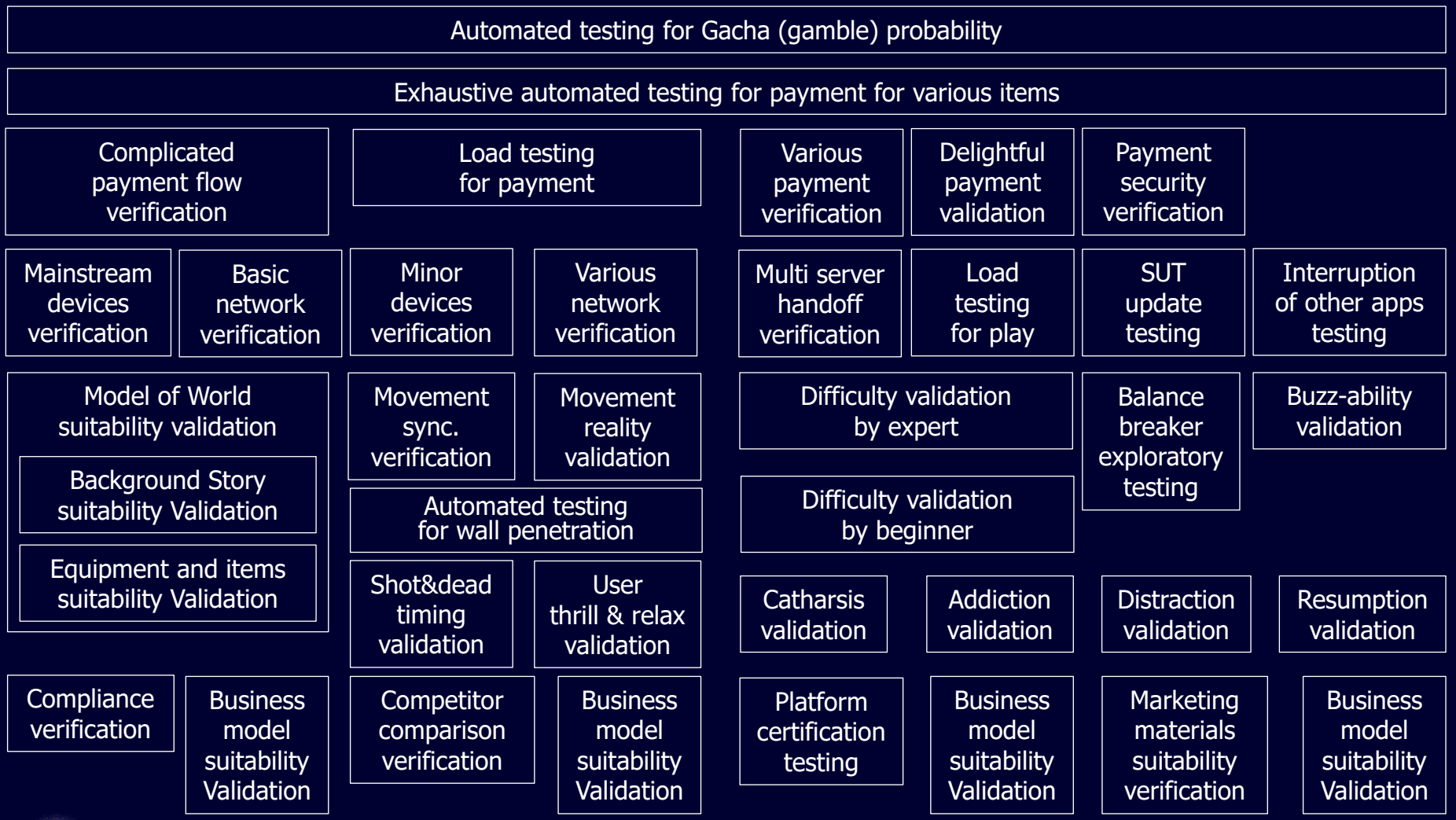
- テストコンテナを用いてテストアーキテクチャを表現するメリット
  - テスト観点よりも粒度の粗いテストコンテナを用いるので、俯瞰して把握しやすい
  - テストコンテナ間に含まれるテスト観点を比べると、役割分担を明確に把握できる
    - » 現場で経験的に用いているテストコンテナの妥当性を評価したり改善できるようになる
      - 負荷テストと性能テストなど、違いがよく分からないテストタイプを区別できる
      - 単体テストと結合テストなど、役割分担がよく分からないテストレベルを区別できる
  - 複数のテストタイプからなるテストレベル、サブレベルを持つテストレベルなどのように、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意味の違いに気付いて名前を変え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

TAD - テストコンテナへのまとめかたの基本

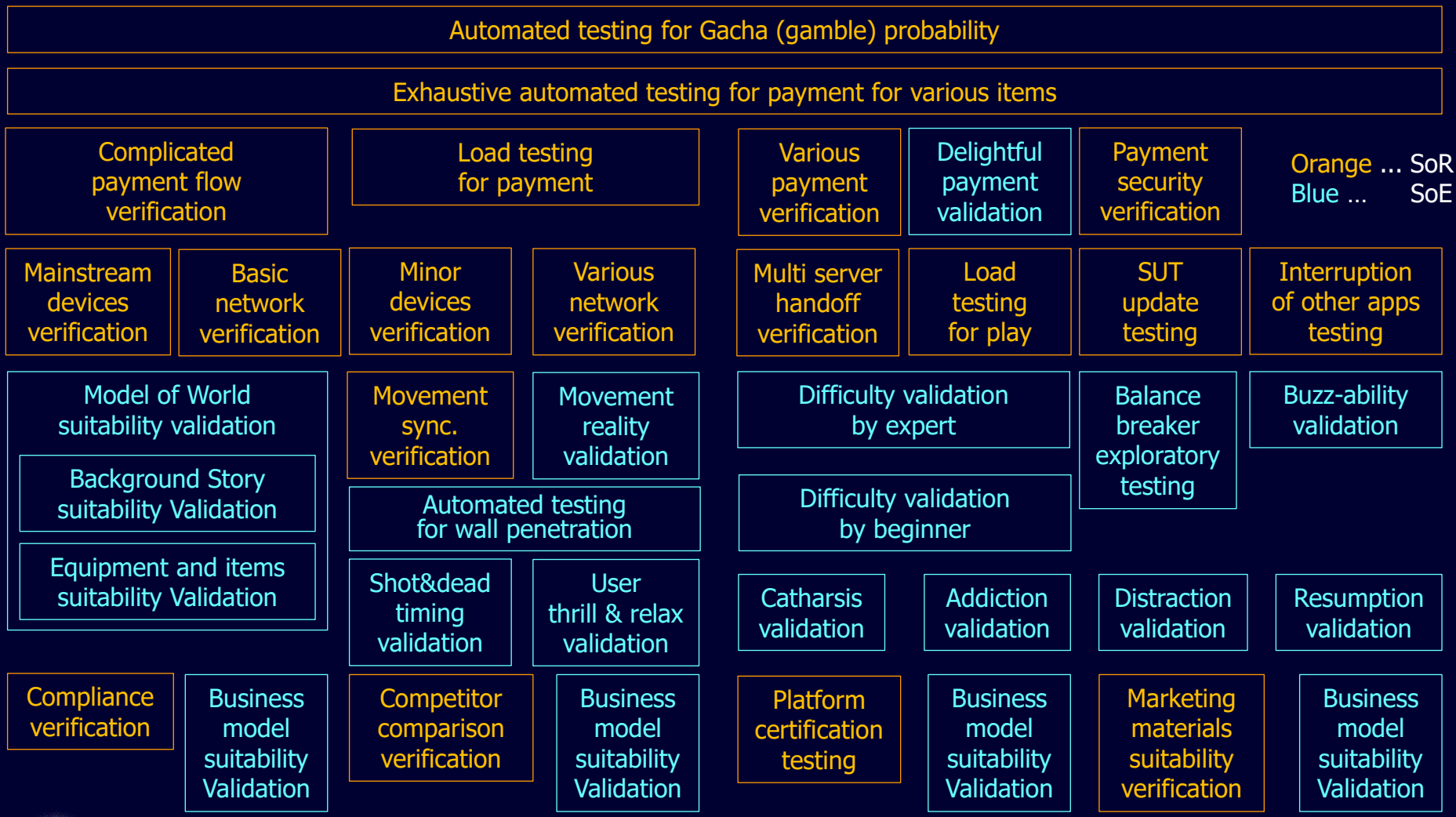
- 同じ時期にテストすべきテスト観点をテストレベルとしてまとめる
  - テスト観点には、テスト観点同士の順序依存関係や欠陥特定の絞り込みのための順序依存関係、プロジェクトの制約としての時期依存関係がある
    - » 例）機能テスト観点の実施情報を用いて負荷テストを設計したい
    - » 例）モジュール内設計のテストを実施した後にモジュール間設計のテストを実施しないと欠陥特定の工数がかかりすぎてしまう
    - » 例）特殊なテスト環境が揃うのが後半なので構成テストは最後に回したい
      - 開発環境やテスト環境、本番環境も考慮に入れる必要がある場合がある
    - » 例）性能に関わるバグのうちシステムアーキテクチャに関わるバグは早めに検出したい
    - » 例）リリース後にもテストしたい
- 同じ趣旨を持ってるテスト観点をテストタイプとしてまとめる
  - 例）負荷テストタイプは複数のテスト観点からなるが、負荷をかけるという同じ趣旨を持っている
- 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、実は意味が異なったりすることがあるので注意すること
- 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をまとめるように）設計する
    - » テスト設計意図（Test design intention）はテスト観点やテスト観点の趣旨とは異なることを理解する



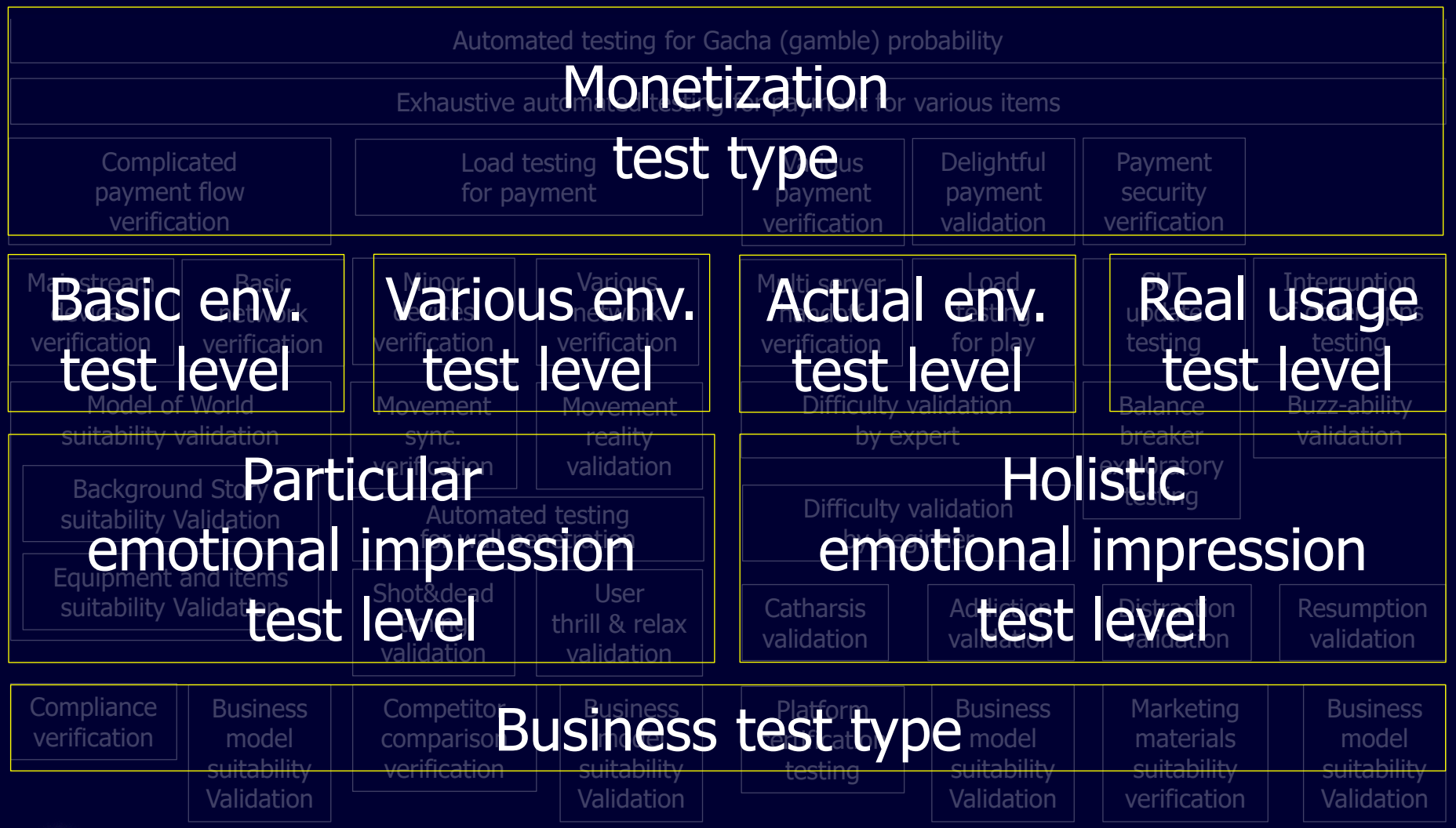
TAD - 自分なりのテストコンテナを考えてよい



TAD - SoEやSoRのテストコンテナが混在することもある

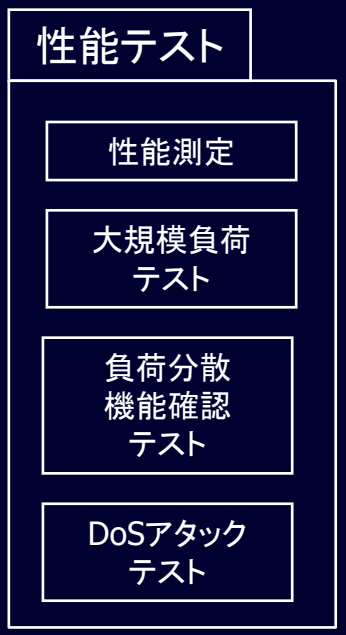


TAD - テストコンテナの粒度も自分なりでよい



TAD - 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
    - » 全く同じテストスイートでもコンテキストやフォーカス、制約条件などによって変わる
  - テストコンテナには適切に責務を表す名前をつける必要がある
- 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）
    - » Detection, Assurance, Traceability, Evidence-making, Learning, Exploratory, Trial-and-error, Rhythm-making, Requirement decomposition etc.
  - 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）
  - Execution 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）
    - » Design direction, Design positiveness etc.



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？

TAD - テストコンテナの責務:テスト設計意図

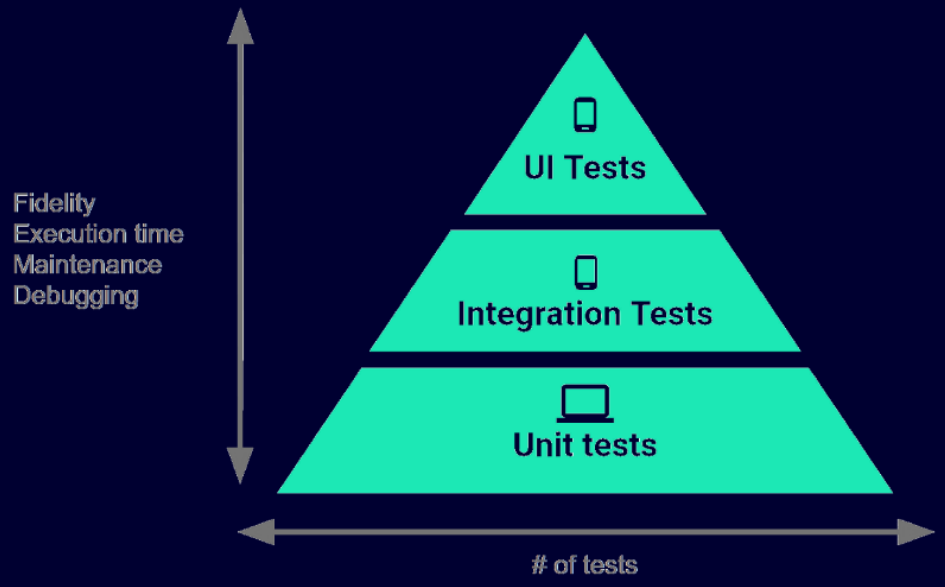
- テストには様々な設計意図があり、しばしば混在するため、
（サブ）コンテナとして同じ設計意図に揃えた方がよい
  - テスト設計意図は、そのテスト（コンテナ）の実行結果をどのように役立てたいか、である
    例）Assurance（網羅する/保証するのが目的）のテストと
    Detection（バグを見つけるのが目的）のテストを混ぜると、
    結果として網羅もバグ検出も中途半端に終わるし設計しにくい
  - テストの設計意図には他にも色々ある
    - » Traceability（トレーサビリティを確保する）, Evidence-making（エビデンスを取得する）,
      Learning（テスト対象を学習する）, Exploratory（テスト対象を探索する）,
      Trial-and-error（開発やデバッグ、チューニングの一部として試行錯誤する）,
      Rhythm-making（リズムをつくる）, Requirement decomposition（要求や仕様を詳細化する）
- 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るべきである
  - 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よって表現されるはずである
    例）AssuranceのコンテナにUSB-Cというテスト観点があったら
    様々な種類のUSB-Cを網羅して保証するというのがテストケースの意図になるだろうが、
    Detectionのコンテナにあったらバグを起こしそうなUSB-Cの実装・実機をつなげてみる
    というのがテストケースの意図になるだろう
  - テストコンテナの設計意図とテストケースの意図が食い違っている場合、
    何か理由が無い限り、それはきっとよくないテストアーキテクチャ設計であろう

TAD - テストコンテナの責務：設計種別類似性

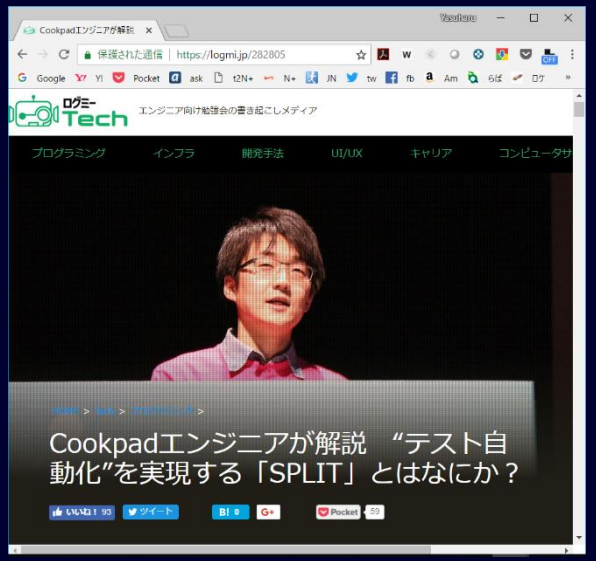
- 設計種別類似性は、設計者の頭の使い方の類似性である
  - テスト設計者はいくつかの異なる頭の使い方をするため、（サブ）コンテナとして揃えた方がよい
  - 設計種別は、テスト（コンテナ）の実行結果をどのように役立てたいか（テスト設計意図）、ではない
    - » もちろん強く関係するので、しっかり考えましょう
- Design direction（設計方向性）
  - （狭義の）テスト条件を挙げてから
    それによって引き起こされる期待結果を考えるテスト設計（順設計：Forward design）と、
    （望ましくない）期待結果やふるまいを挙げてから
    それを引き起こす（狭義の）テスト条件を考えるテスト設計（逆設計：Backward design）とは、
    設計時の頭の使い方が異なるので、混ぜない方がよい
    - » 一般に、逆設計の方が難易度が高い
- Design positiveness（設計肯定性）
  - 動くことを証明したいテスト設計（Positive test design：肯定的テスト設計）と
    動かないことを証明したいテスト設計（Negative test design：否定的テスト設計）とは、
    設計時の頭の使い方が異なるので、混ぜない方がよい
    - » 肯定的テスト設計では、まとめたりなぞったり対応させたりする頭の使い方が多い
      - テスト自動化には肯定的テスト設計が向いている
    - » 否定的テスト設計では、ツッコんだり踏み込んだり逆に考えたりする頭の使い方が多い
      - 探索的テストは否定的テスト設計が多いかもしれない
      - テストの再利用やナレッジベースなどを構築するためには、否定的テスト設計のパターン化や言語が必須となる
      - 「～が無いこと」という期待結果の（よくない）テストケースは、
        表現が否定形なだけでテスト設計種別は肯定的になることが多く上手くテスト設計できないため、
        スキルが低いテスト設計者には否定的テスト設計のパターンなどをなぞらせる
        （否定的テスト設計パターンを肯定的にテスト設計してもらう）という手もある

TAD - テストピラミッドやSPLITというアプローチもある

- SPLIT
  - Scope - テストの対象範囲をどこにするか / TRA
  - Phase - テストのどの段階（in dev. / in prod.）に焦点を当てるか / TAD
  - Level - どのレベルまでオートメーションを進化させるか / TSD&TAD
  - sIze - 対象の範囲をどう切り分けて実行するか / TAD
  - Type - どのような種類の目的でテストを行うか / TRA

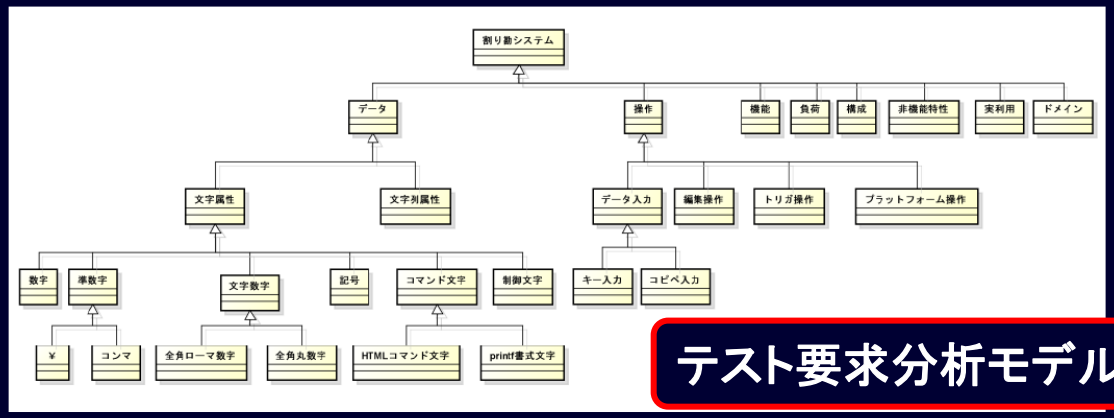


https://thinkit.co.jp/article/13346

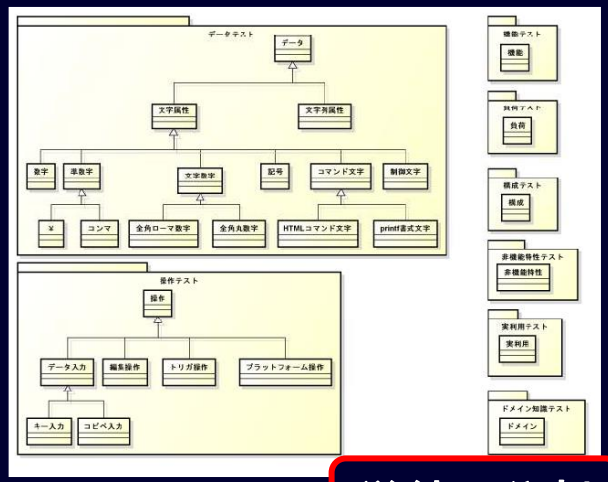


https://logmi.jp/282805

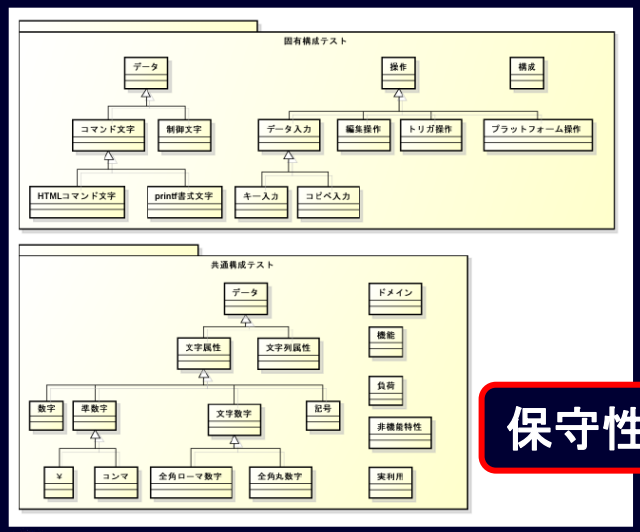
TAD - テストスイートに求められる品質特性によって異なるテストアーキテクチャの例



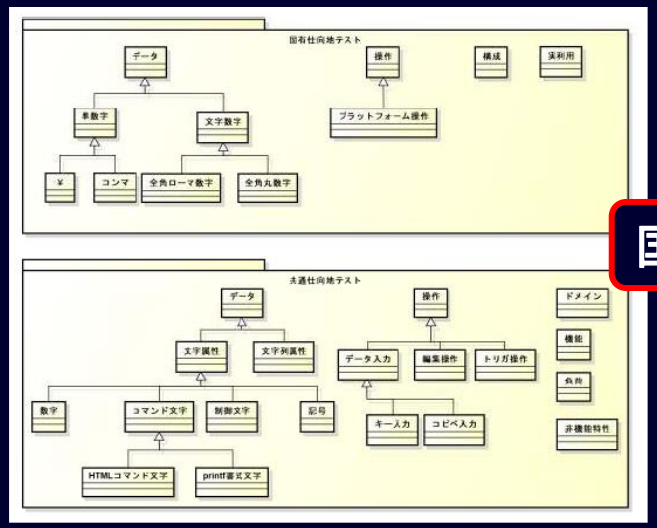
テスト要求分析モデル



単純に分割



保守性重視



国際化重視

TAD - 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
    - » 素人に触らせて品質が向上するほどQAは甘くない
      - 「素人がしそうな操作ミス」「素人はどう思うか」という明確なテスト観点に絞る場合は例外だが、
        その場合でも探索的テストやアドホックテストとは呼ばない
- 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
    - » チャーターを細かく書きすぎると創造性が減るし、粗すぎてもマネジメントできない
    - » チャーターを使わないフリースタイルというやり方もある
  - 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
    - » 60～120分程度と言われている
- 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ないと、「その他のテスト」コンテナに成り下がる

TAD - 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）
  - などなど
- 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
      - アジャイルだから探索的テスト、と短絡的に考える組織はおそらくアジャイル開発がちゃんとできていない
    - » 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

TAD - アジャイル開発におけるテストアーキテクチャ

- 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
      - どうしたらユーザが「満足」し「幸せ」になるのか、ビジネスが「成功」するのかを
        ユーザやチームとの対話によって深く理解し、満足や幸せ、成功が達成されているか・阻害されていないか、を
        実証・保証するテスト観点をテスト要求分析として検討することで、
        最初からテスト(QA)エンジニアもチームとして一緒にプロダクトをよくしていくという意識や行動が重要である
      - ただ開発のイテレーションに組み入れられてチケットに書いてあるようにテストすればいい、という姿勢ではいけない
  - 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
        テストがどんどん乱雑になっていき、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
        - * いつどのよう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行うべきかも判断しにくくなり、タイムボックスが管理できなくなる
      - 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、
        どんどん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
        - * 開発、チーム内テスト(QA)、SET、チーム外QAなどで議論しながら、どんどんテストを自動化していけるようなテストアーキテクチャを設計する

TAD - アジャイル開発におけるテストアーキテクチャ

- 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
      - 費用対効果の高い開発者テストをどういうコンテキストでどこまで行うと皆が幸せになるのか、
        それをどのように段々高度化していくのか、を開発エンジニアに納得してもらい自然にテストするようになるのが重要である
        - * 「開発者がテストを書かない」とただ嘆くのではなく、テストを書いた方が幸せになることを納得してもらえない自分たちに工夫の余地がある、と考える
      - 開発はどのスキルレベルにあって何ができて何ができないか、を冷静に見極めて、
        テスト(QA)が目配りすべきところをダイナミックに変化・成長させながら、開発と一緒にレベルアップできるよう考えて行動する
        - * あまりアジャイルではないと思ったら、テスト(QA)がチームのアジャイル化を牽引するくらいの意識でよい
      - サイロ化や対立、階級意識がもしあれば常に根絶し続ける
      - コンテキストによっては、TDDでのテストを品質の保証の役割に使わない方がよい
    - » 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どんどん双方向で共有しやすくできる
      - テスト要求分析やテストアーキテクチャ設計は、
        チームの内外に関わらず開発とテスト(QA)のコミュニケーションを増やし精度を高めスピードを上げるために行う
      - テスト(QA)エンジニアは、悪さの知識などテスト(QA)だからこそ気付けることが必ずある
      - テストでの抜け漏れを無くすために複雑で厳密なモデルにこだわりすぎる「モデルおたく」になってしまうのはよくない

TAD - イテレーション型テストアーキテクチャの設計

- 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
      - 隠された依存関係があるのかもしれないことを、アーキテクトや開発エンジニアとの対話から感じ取れると素晴らしい
  - 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
      - 運用中にテストした方がユーザの価値が高まったり品質が高まるテスト観点をいかに識別するかがキーである
      - ただ単にテストが間に合わないからシフトライトする、という意志決定をするのではなく、デプロイ後にバグが少ない見通しやバグが出ても大丈夫な仕組みが機能するかどうかを総合的に判断する必要がある
    - » 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

TAD - イテレーション型テストアーキテクチャの設計

- 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
      - 理想的にはテスト自動化を整備して、イテレーションごとにリリースが完結するべきである
      - 漸次的に自動化する場合、テスタビリティを高める開発のリファクタリングや、自動化容易性を高めるテストのリファクタリングを行う必要性が発生する場合もある
      - テストを（サブ）コンテナ化しておくと、（サブ）コンテナごとに自動化するかどうかの判断ができる
      - 直後/以降のイテレーションで（そのイテレーションで開発してないのに）テストすることを「イテレーションスリップ」と呼ぶ
      - テスト専門のイテレーションが並行/後追いで走り出すことを「イテレーションフォーク」と呼ぶ
    - » 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
      - 各イテレーションの最初のハッピーパステストコンテナをGUI変更確認とマッピングに用いる、など

TAD - イテレーション型テストアーキテクチャの設計

- 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
      - システム全体のテストに関するコンテナなどであっても、全部揃う前からできることを検討し設計・実施した方が品質リスクを少なくできる
- 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
    - » イテレーションごとに実装上の依存関係を測定し続けるという手も可能かもしれない
    - » 安易にイテレーションフォークをするのは避けるべきである

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
    - » テストが増えていくとテストケースを大規模ソフトウェアとして扱う必要が出てくる
  - 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
    - » 自動テストが増えていくとテストコードを大規模ソフトウェアとして扱う必要が出てくる
  - 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フルスタックで必要になってくる
  - だから常に勉強してスキルをつけて試して振り返って成長することで、楽しいエンジニアライフにしよう！ ☺

テスト設計コンテストの応募作のテストアーキテクチャ

- テスト設計コンテスト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（マトリックス）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

		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

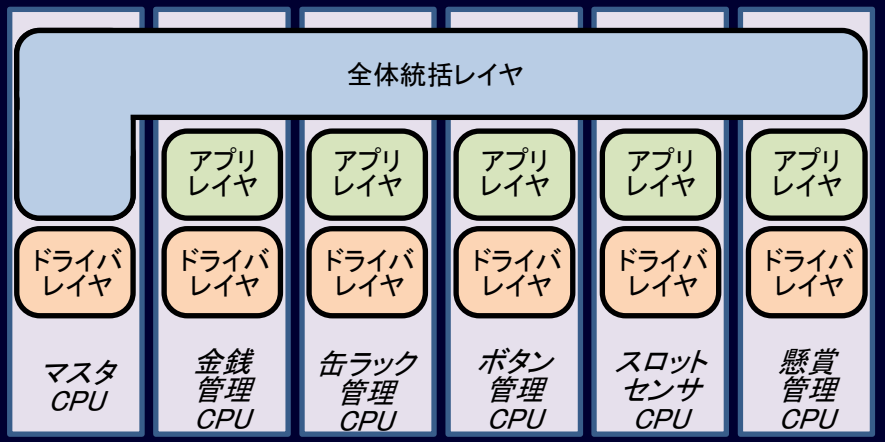
表（マトリックス）風の記法

テスト設計コンテストの応募作のテストアーキテクチャ

- テスト設計コンテスト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
  - レイヤー型アーキテクチャ
  - フィルター型アーキテクチャ
  - 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？
    - » 開発イテレーションに依存した安直なテストアーキテクチャが多かったのが残念である

レイヤー型アーキテクチャ

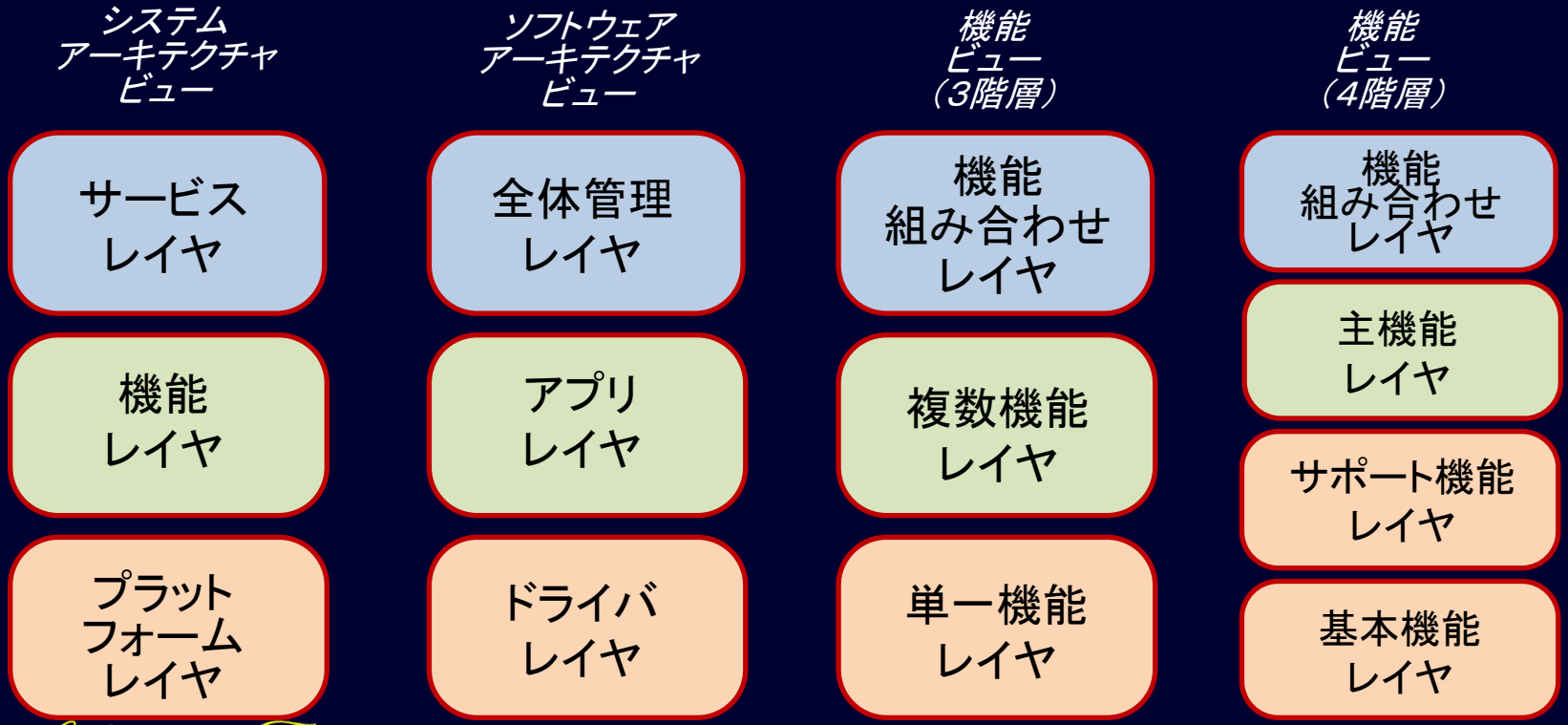
フィルター型アーキテクチャ

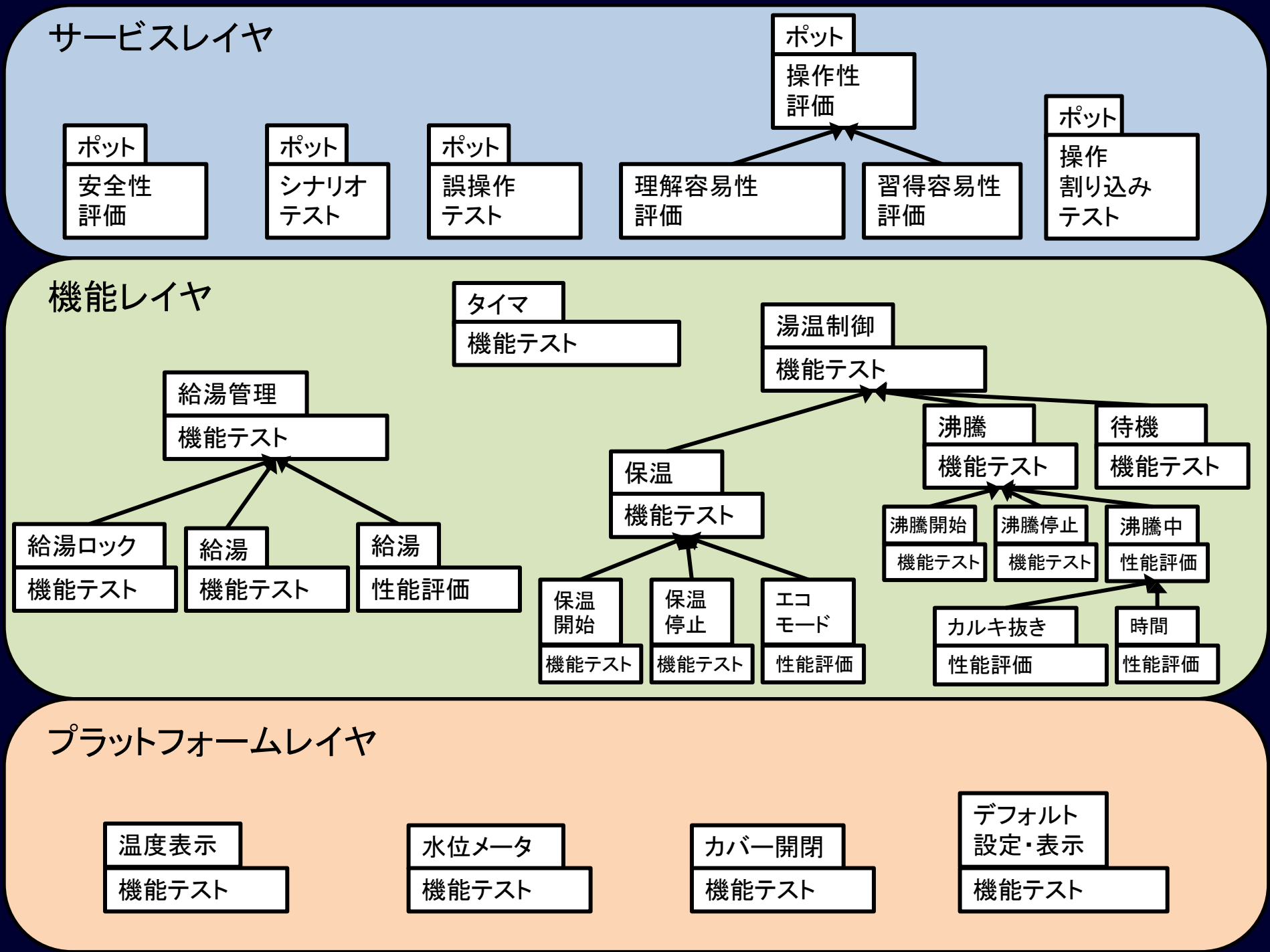


複合型アーキテクチャ

テスト設計コンテストの応募作のテストアーキテクチャ

- 一見似たようなレイヤー型アーキテクチャだが、実は異なっていた
  - ビューやレイヤ数も様々だった



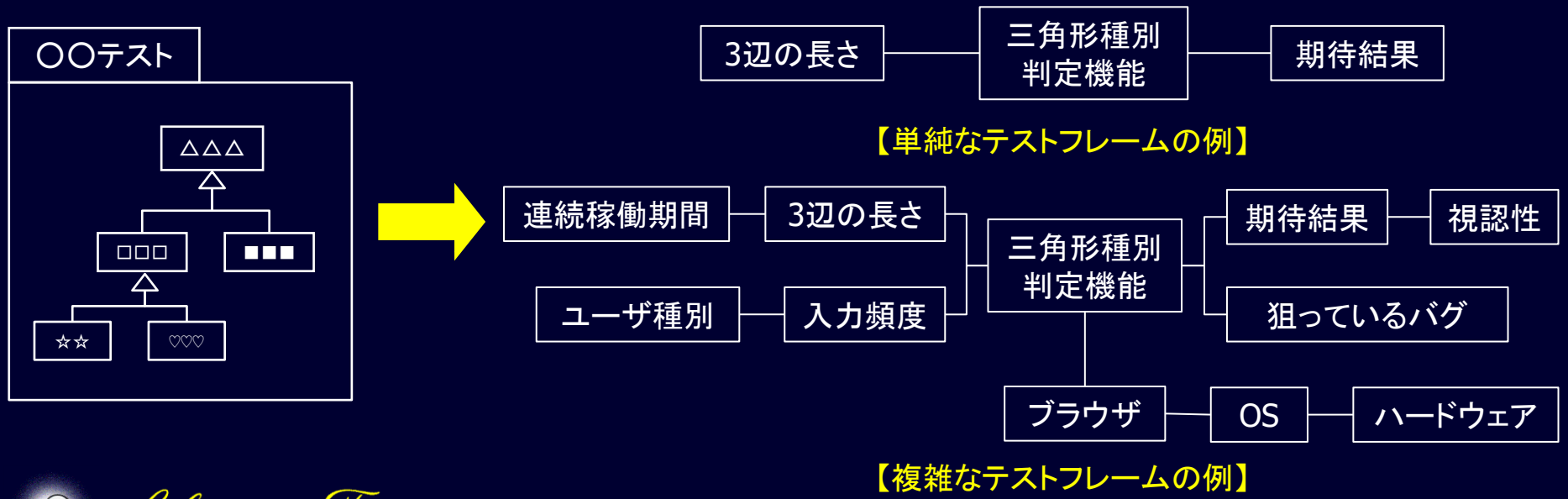

サービスレイヤ
ポット
安全性
評価
ポット
シナリオ
テスト
ポット
誤操作
テスト
ポット
操作性
評価
理解容易性
評価
習得容易性
評価
ポット
操作
割り込み
テスト
機能レイヤ
タイマ
機能テスト
給湯管理
機能テスト
給湯ロック
機能テスト
給湯
機能テスト
給湯
性能評価
湯温制御
機能テスト
保温
機能テスト
保温
開始
機能テスト
保温
停止
機能テスト
エコ
モード
性能評価
沸騰
機能テスト
待機
機能テスト
沸騰開始
機能テスト
沸騰停止
機能テスト
沸騰中
性能評価
カルキ抜き
性能評価
時間
性能評価
プラットフォームレイヤ
温度表示
機能テスト
水位メータ
機能テスト
カバー開閉
機能テスト
デフォルト
設定・表示
機能テスト

TAD - テストフレームモデリング

- 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
  - この条件のテストは、テスト対象のどの部分に行うんだろう？
  - この部分に対するテストでは、どの品質特性を確認するんだろう？
  - この品質特性をテストするには、どの操作を行えばいいんだろう？
- テストケースの構造（スケルトン）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
  - テストケースの構造を表すテスト観点の組み合わせを「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
    - シンプルなテストケースで十分であれば、テストフレームを組まなくてもよい
  - テストフレームの例：
    - 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

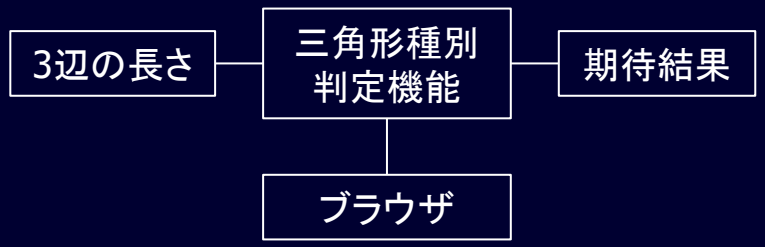
TAD - テストフレームモデルによるテストケースの構造の明示

- 「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うる
  - テストケースの構造をテスト観点で表したものを「テストフレーム」と呼ぶ
  - 複雑なテストケースが偉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必要がある
    - » 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ので、どういうトレードオフをどう考えてから、もしくはどういう意図でこの複雑さのテストフレームにしたか、はきちんと説明できないといけない



TAD - 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るけど、
    ちっとも整理されておらず網羅しにくいテストケース表ができあがる
  - 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)

【テストケース表の例】

TAD - 組み合わせテスト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必要がある
  - 明記しないと、組み合わせずに代表値や境界値だけで
    テストケースを作ってしまうかもしれない
  - 組み合わせが多い方が偉いわけでもバグを見つけやすいわけでもないことに
    注意する必要がある
    - » 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ので、
      どういうトレードオフをどう考えてからこの複雑さの組み合わせにしたか、
      は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

【組み合わせのテストケース表の例】

チュートリアルの流れ

- TDLC（テスト開発ライフサイクル）とは
- TRA（テスト要求分析）
- TAD（テストアーキテクチャ設計）
- TDD・TI（テスト詳細設計、テスト実装）
- テスト開発のTIPS
- 過去のテスト設計コンテストの応募作のポイント

テスト詳細設計（TDD）- 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
  - 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
    - » 十分詳細化されたテスト観点が「テストパラメータ」となる
  - 網羅基準（カバレッジ基準）を定める
  - 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
      - ・ テスト詳細設計を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のタイプには4つある
  - 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)
  - 間接テスト値：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、その後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あげるのは、質の高いテストとはいえない
  - テストフレームやテスト詳細設計モデルを特定すれば、モデルベーステストが可能になっていく

TDD - 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時に用いる
  - 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通り
  - 全網羅
    - » 例）0, 1, 2, 3, 4, 5, 6, 7 … 100 … 10000 … 65534, 65535
    - » 漏れは無いが、テスト値が膨大になるため現実的では無い
  - 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わすれないこと
    - » 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、
      よほど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
    - » テスト値は少数に収まるが、漏れが発生する

TDD - 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時に用いる
  - 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通り
  - 全網羅
    - » 例）Chrome, Firefox, Safari, Internet Explorer, Edge
    - » 漏れは無く、通常はテスト値もそれほど膨大にはならないが、
      組み合わせになったら無視できない程度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リリースされたブラウザ
    - » 要素の何らかの属性がある連続した範囲の場合に
      その属性の境界値を持つ要素を境界値要素とみなすことができなくもないが、
      よほど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
    - » テスト値は少数に収まるが、漏れが発生する

TDD - マトリクスタイプ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
- マトリクスタイプのテスト詳細設計モデルの網羅基準は以下の通り
  - 全網羅
    - 例）(XP, Chrome), (Vista, Chrome) … (Win8.1, Edge), (Win10, Edge)
    - 漏れは無いが、掛け算によりテスト値が膨大になるため現実的では無い
    - 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
    - N=2の時をPairwiseと呼ぶ
    - All-pairsなどN-wise系の網羅手法と、直交配列表を用いた網羅手法がある
    - Nまでの組み合わせの網羅で十分かどうかを検討しないと漏れが発生する
  - 代表値網羅
    - テスト値は少数に収まるが、漏れが発生する

TDD - 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
    - » 間接テスト値になることが多い
    - » 制御フローパステスト（プログラムのロジックのテスト）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通り
  - 全パス網羅
    - » 漏れは無いが、テスト値が膨大になるため現実的では無い
  - MC/DC（Modified Condition/Decision Coverage）
    - » 航空宇宙業界などで使われる
  - Nスイッチ網羅
    - » 連続するN個のリンクの組み合わせを網羅する方法 / N+1以上のスイッチのテスト漏れが発生する
  - リンク網羅（0スイッチ網羅、C1網羅）
    - » グラフのリンクを網羅するようにパスを生成する / 1以上のスイッチのテスト漏れが発生する
  - ノード網羅（C0網羅）
    - » グラフのノードを網羅するようにパスを生成する / リンクのテスト漏れが発生する
  - 代表パス網羅
    - » テスト値は少数に収まるが、漏れが発生する

テスト実装（TI）

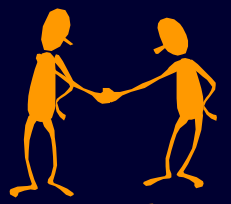
- テストケースが生成できたら、テスト実装を行う
  - テスト対象のシステムやソフトウェアの仕様、テストツールなどに合わせて
    テストケースをテストスクリプトに具体化していく
    - » 手動テストスクリプトの場合もあるし、自動化テストスクリプトの場合もある
  - テスト実装は、テスト対象のUI系の仕様や実行環境、ユーザのふるまいに関する
    ノウハウが必要である
- 集約
  - テスト実施を効率的に行うため、
    複数のテストケースを1つのテストスクリプトにまとめる作業を集約と呼ぶ
  - 同じ事前条件や同じテスト条件、同じテストトリガのテストケース、
    あるテ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
    - » テストトリガとは、テスト条件を実行結果に変えるためのきっかけとなるイベントである
- キーワード駆動テスト
  - 自動化テストスクリプトを“クリック”といった自然言語の「キーワード」で
    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

チュートリアルの流れ

- TDLC（テスト開発ライフサイクル）とは
- TRA（テスト要求分析）
- TAD（テストアーキテクチャ設計）
- TDD・TI（テスト詳細設計、テスト実装）
- テスト開発のTIPS
- 過去のテスト設計コンテストの応募作のポイント

テスト開発プロセスを現場に適用する際のポイント

- 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ィ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
    - » いきなり全てを天下り的に導入しない方がよい
  - まず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や悪さを実感してみるとよい
  - 常にテスト開発プロセスのよ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
    - » 皆でモデルを囲んで議論し、納得感を深めていく
  - 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

- 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
    - » トップダウンとボトムアップを循環させながら詳細度を整えていく
  - 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
    - » 例）テストタイプやテストレベルの定義を見直す
    - » 例）固定3レイヤー法は継続するが、大項目・中項目・小項目のレビューをきっちり行う
    - » 例）上流工程で経験的に（アドホックに）行っている工程を文書化する
    - » 例）アドホックな派生開発をテスト観点モデルで整理していく

チュートリアルの流れ

- TDLC（テスト開発ライフサイクル）とは
- TRA（テスト要求分析）
- TAD（テストアーキテクチャ設計）
- TDD・TI（テスト詳細設計、テスト実装）
- テスト開発のTIPS
- 過去のテスト設計コンテストの応募作のポイント

過去の応募作の審査で気になった点

- 設計根拠を明示して欲しい
  - 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欲しい
    - 「なんかよく分かん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
      - 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
    - 異なる設計根拠のテストアーキテクチャを単につなげただけではよくならない
  - 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欲しい
    - 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欲しい
- 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欲しい
  - 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欲しい

過去の応募作の審査で気になった点

- スコープとその根拠を明示して欲しい
  - 設計範囲とそのために考慮すべき範囲、考慮する必要が無い範囲と設計しない範囲を根拠とともに明示して欲しい
    - » 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欲しい
  - 巨大なマトリクス、ゴチャゴチャした線、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もない
- 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欲しい
  - 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欲しい
    - » オレオレテスト開発プロセスは、フェーズ間の結合度が高く凝集度が低い場合が多い
- 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欲しい
  - それぞれのプラクティスのキー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もっとよくなるように最新のテスト開発方法論を構築して欲しい
    - » とにかくアジャイルにしてみました、とにかく反復型にしてみました、とにかく自動化してみました、とにかく探索的テストをやります、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欲しい
  - 文章だけでなく図やモデルを使っても構わない
    - » 図やモデルは「モデリング言語」である
  - 他の文書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
    - » ○○の規格にあるからそれを採用しました、は思考停止である
    - » 「思考停止→理由なき引用→構造化されていない情報の羅列→実行→業務報告」から脱却する
- そして、常に全体像を俯瞰して捉えて欲しい
  - 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がある
    - » トレーサビリティさえ確保すればいい、というわけではない点に注意せよ
- 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
    - » 直感で判断などを行ったところには、「ここは直感で○○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、流用した時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、そもそもきちんと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進められていないはずである
    - » テスト部隊内部でのコミュニケーション、開発とテストとのコミュニケーション、
      テストとQA部門とのコミュニケーション、上級管理職とのコミュニケーション、
      自分たちのお客様や元請けとのコミュニケーションや協力会社とのコミュニケーションが
      「本質的に」取れていない可能性も高い
      - 通信量は多いが形骸化したコミュニケーションやプロセスになっている可能性が高い
  - 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
    - » テストはできてないけど開発ではきちんとできている、というのは幻想である
  - 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