第 35 年度（2019 年度）ソフトウェア品質管理研究会

特別講義　レポート

作 成 日：2020 年 1 月 10 日
書記氏名：西澤 賢一

日時	2020 年　1 月　10 日（金）　　10:00　　～　12:00
会場	一般財団法人日本科学技術連盟・東高円寺ビル 地下 1 階講堂
テーマ	ソフトウェア疲労をアーキテクチャ設計で改善する
講師名・所属	山田 大介 氏（ビースラッシュ株式会社 代表取締役）
司会者	岩井 慎一 氏（株式会社デンソー／本研究会 ソフトウェア品質保証の基礎 主査）
アジェンダ	１．ソフトウェア開発の状況 ２．ソフトウェア疲労 ３．アーキテクチャ設計と設計技法の違い ４．アーキテクチャ設計 ５．アーキテクトの役割とスキル ６．三階建て設計改善 ７．まとめ
アブストラクト	IoT 時代のソフトウェア開発は複雑さへの対応と素早い出荷の両立を求められています。開発現場では、動いているプログラムを使い続けることで、資産価値は低下し在庫化しつつあります。それらを解決するためにアーキテクチャ設計が有効です。アーキテクチャ設計では、複数のビューでモデル化を行い、設計意図を伝達します。今回は、アーキテクチャ設計の要点とそれを推進するアーキテクトの育成について紹介します。

◆ 講師紹介

山田 大介 氏
1990 年～ 構造化技法／オブジェクト指向分析設計
1998 年～ ソフトウェア部品化再利用／ドメインエンジニアリング
2003 年～ プロダクトライン開発／リファクタリング
2007 年～ アーキテクチャ設計／アーキテクト育成

1．ソフトウェア開発の状況

- デジタル化を起点として、組み込みソフトウェアの規模が増大。
- マネジメントとしては、プロセスやプロマネを強化する方向へ
  - 「エンジニアリングなきマネジメント強化」は悪循環を招くことも
- 近年、ネットワークに接続する製品の開発が増加 -> IoT では製品を取り巻く周辺の状況変化が発生している。
- ソフトウェアの資産価値が在庫化している？
  - 在庫: ソースコードは複雑で説明困難。信頼できるドキュメントは存在しない。保守コストが膨れていく
  - 属人資産: ソースコードはシンプル。設計ドキュメントがそろっていない。個人主導の開発
  - 半組織資産: ソースコードは複雑。設計ドキュメントはそろっている。マネジメント主導の開発
  - 組織資産: ソースコードがシンプル。設計ドキュメントは整備されている。予測可能な開発
  - 戦略資産: ソースコードと設計ドキュメントが統合。戦略的な資産活用ができる
- ソースコードの変更と改善のジレンマ
  - 機能追加・削除: 局所的な機能追加で当初の設計意図が埋没してしまう。
  - 障害対策: 局所的な修正の積み重ねでスパゲッティ化に陥る
  - 性能最適化: 設計構造を崩すことも。
  - 再利用資産化: 機種ごとのコンパイルスイッチはアドホックに追加

  → 「ソフトウェア疲労」が発生している → 抑制には設計改善による「可視化&洗練化」が必要

2. ソフトウェア疲労

ソフトウェア疲労【静的】

1. 一筆書き: 一つの巻子やファイルが長い。作った人にしかわからない
2. クローン: 同じコード断片が点在。修正漏れが発生する。grep 検索が最も便利な開発ツール
3. 神様データ: 全てを支配しているデータが存在する。修正の波及範囲が大きい。
4. 中央集権: 一つのファイルにたくさんの関数がある。いつも同じファイルを修正している
5. スパゲッティ: いろいろな関数を呼び出している。副作用が発生する。
6. 老舗温泉旅館: 階層を超えた呼び出し／命名規則がない。引継ぎができない
7. 一枚岩: #include しているファイルが多い。分割コンパイルができない

➔ 想定原因

- 1, 2: そもそも設計していない
- 3~5: 設計技法を使いこなしていない
- 6, 7: 全体を見ていない。アーキテクト不在

- 構造設計の基本
  - 一筆書きは設計ではない。フローチャート／コーリングシーケンスはモジュール化はできているが、レベル化ができておらず、これも設計ではない
  - 構造設計には箱（What の名称）、線（利用関係）、配置（水平垂直分割）が必要である。
- 設計品質の原則: 凝集度
  - 機能的、逐次的、通信的は OK。手順的、一時的、論理的、偶発的は NG である。

ソフトウェア疲労【動的】

1. 末広がり: 処理の起点からフラグ判断を積み重ねて動く。変更は、流れを追いかけないとできない。
2. 裏取引: 後からアドホックに追加されたフラグ変数が多い。改修時に思わぬ副作用が出てしまう
3. 状態迷路: 状態数やイベント数が 20 を超えている。状態やイベントを追加削除することが大変
4. 途中で待つ: ボタン操作が効かない。イベントの順番が変わると動かなくなる。
5. データ競合: セマフォでデッドロック。スレッドセーフの連続でパフォーマンス遅延
6. タスク過多: タスクの数が意味もなく多い。サイクリック実行型ではループの増殖
7. 凸凹 API: IF プロトコルが煩雑。IF の順序が分かると動かなくなる

➔ 想定原因

- 1, 2: そもそも設計していない

- 3~5: 設計技法を使いこなしていない
- 6, 7: 全体を見ていない。アーキテクト不在

- 制御仕様をそのままプログラミングすると「末広がり」になってしまう。コードレベルで改善しても、他の例にあてはめられない。構造化すれば設計図で説明できるようになる
- プログラミングスタイルの違い: アセンブラ的Cからモジュール的Cへと変わってきており、動解析ファーストではなく、静解析ファーストへと移ってきている

3. アーキテクチャ設計と設計技法の違い

- 多面化と詳細化
  アーキテクト: 多面的に図面化し設計意図を伝達する
  設計担当者: 設計を詳細化しプログラミングを行う
- アーキテクチャ設計とコンポーネント設計
  - アーキテクチャ設計は、構造と意図を明確にして人に伝える
    - 重要な部分から明確にしていく
    - 未記述部分（TBD）があってもよい
  - コンポーネント設計は、ソースコードに直結する設計
    - 設計書を最新に保つコツは、設計とコードを同期させること
- そもそも「設計」とは?
  1. **実装する前に**: 要求分析: 要求を正しく理解し、設計することで、手戻りをなくす
  2. **全体像を明らかにして**: 静的構造: 全体を構造的に俯瞰することで、適用範囲を明確にする
  3. **問題点を検討し**: 動的構造: 構造要素と要素間の問題を、あらかじめ検討する。わからないことを明確にする。
  4. **複数の関係者の認識を合わせる**: 文書化: 設計意図を伝達し、開発の方向性を合わせる
- 設計図とは
  - 静的構造図と動的構造図の両方が必要
  - 全体を俯瞰できる粒度で静的構造を表現する
- 「設計力」= 分離 + 構造化 + 図解。設計力があれば、いろいろな設計図を読み書きすることができる。そして、次第にいろいろな手法を使いこなすことができるようになる

4. アーキテクチャ設計

- 組込みシステムの5つのビュー: 静的構造を中心に、動的構造と実装構造を設計
  - 目論見: 製品の特徴や売り
  - 設計方針: 目論見を実現するための設計方針
  - 静的構造: 責務や機能の単位
  - 動的構造: 並行動作するタスク要素、タスクや割込みという実行要素単位
  - 実装構造: ファイルの構成単位、設計規約
- 動的ビューと静的ビュー
  - システムを複数の視点（ビュー）でとらえる。複数視点の図面を統合化することで方式設計する
- 各ビュー間の関係
  - 目論見を実現するための設計方針
  - 目論見、設計方針を具現化する静的構造
  - 静的構造を下敷きにして、動的構造／実装構造を記述

5. アーキテクトの役割とスキル

- アーキテクトの役割
  - 正しく判断し、プロジェクトを成功に導く
  - 技術支店のリーダーシップを発揮する人
- アーキテクトの基本スキル

- 複数のビューポイントで対象を図表化する
- 複数の図表を統合して一冊の空きてくちゃドキュメントにする
- 様々なステークホルダを巻き込んで、リーダシップを発揮する
  - ✧ プロマネは売上増大、アーキテクトは利益率向上に貢献する

- システムズ・アーキテクト
  - システムズ・アーキテクトに求められること: アーキテクトが経営とテクノロジをつなぐ
    1. 異ドメインの接続: ヘテロジニアスな全体像をつかむ
    2. ビジネスへの補助線: コンセプトやバリューを考える
    3. 本質を見抜くスキル: 抽象化できる人材の育成
- IoT システム開発を成功に導く要素
  - ビジネス価値のプロト開発(PoC)から、素早くゴールに到達（アジャイル）し、運用までを考慮(DevOps)することで、ビジネスを成功に導く
    - ✧ これを支えるのが、システムズ・アーキテクト
      - 必用となるスキル: 仮説検証スキル、統合スキル、解析スキル、抽象化スキル、モジュール化スキル

6. 三階建て設計改善

- ソフトウェア疲労の再発防止
  - 開発プロセスに設計エンジニアリングを組み込む
- 三階建て設計改善
  - ソースコードを直接修正するのではなく、設計図上で原因を特定し、アーキテクチャの方針に従って、設計図を修正し、ソースコードを回収する。
- アーキテクチャ設計原則
  1. モジュール化: モジュール（変数/関数/ファイル/フォルダ）の識別性。What の名称であり、内部にデータを隠蔽している。
  2. 高凝集: モジュールが単一目的を持っている。まとまりのあるデータ群を持っている。
  3. 疎結合: グローバル結合していない。一覧できるスコープを超えない。
  4. レベル化: BOSS を作り、指示<->報告の上下関係を作る。上位がクライアント、下位がサーバの利用関係。
  5. 水平分割: 上位が論理、絵画物理。データの受け渡しは、求心/遠心で。
  6. 垂直分割: 基本系列と横断的関心及び異ドメインを分離する。横断的関心としては初期化やエラー処理。異ドメインは UI など
  7. 静動分離: 周期起動部（動的）と機能実現部（静的）に分ける。制御スレッドは走りきる。フィードバック処理の最小化
  8. 状態連動: 複数の状態が連動して動く。ひとつの状態遷移は状態数を 7 個以内にする
  9. 固定変動（資産化）: 共通部分のライブラリ化やフレームワーク化。変動部分の特定とアーキテクチャ上へのマッピング

7. まとめ

- エンジニアリングで全体を俯瞰: エンジニアリングすることで現状打破。図面化と設計の原則原理
- 在庫から脱却し、戦略資産へ: 資産価値を向上させる
- ソフトウェア疲労【静的】／【動的】を改善する
- アーキテクトが「丸投げ」をつなぐ
  - まずは「コードに近い設計」、最終的に「アーキテクチャで繋ぐ」
- ソフトウェアの資産化アプローチ
  - 既存資産を起点として戦略的な開発の実現へ
    - ✧ ボトムアップ: 既存コードを部品化して、洗練化・資産化していく
    - ✧ トップダウン: 設計意図を明確にして、ソースコードへ反映していく
- 悩むエンジニアから、考えるエンジニアへ: コード中心から設計中心へ

<質疑応答>
Q1: アーキテクチャの大切さはわかるが、マネジメント層にどう伝えるとよいか？例えば、ソフトウェア疲労を防ぐためにリファクタリングをするための工数をどう得るのがよいか

A1: 在庫という言葉を使うことでマネジメント層に伝えると理解できる人が多い。構造を図で見せるのもよい。中間マネジメントが阻止することも多いので注意が必要。

****

Q: 組み込みで今後、海外の OEM を相手にする場合、アセスメントでビジネスチャンスを逃すことが多い。ドイツのドキュメントの差は？

A: ドイツではリファレンスモデルが当たり前。トップダウン流のリファレンスモデルをまじめに作るしかない。ドイツはかなり厳密に作っている。

****

Q: リファクタリングはいつ挟むのがよいのか？

A: リファクタリング目的のリファクタリングはしない。修正、移植の目的のために前段階でリファクタリングするのがよい。
1. 修正のタイミングでリファクタリングする。リファクタリングをしてから、修正する。
2. 移植のタイミングでリファクタリングする。

以上