オブジェクト指向開発論

2020年6月18日

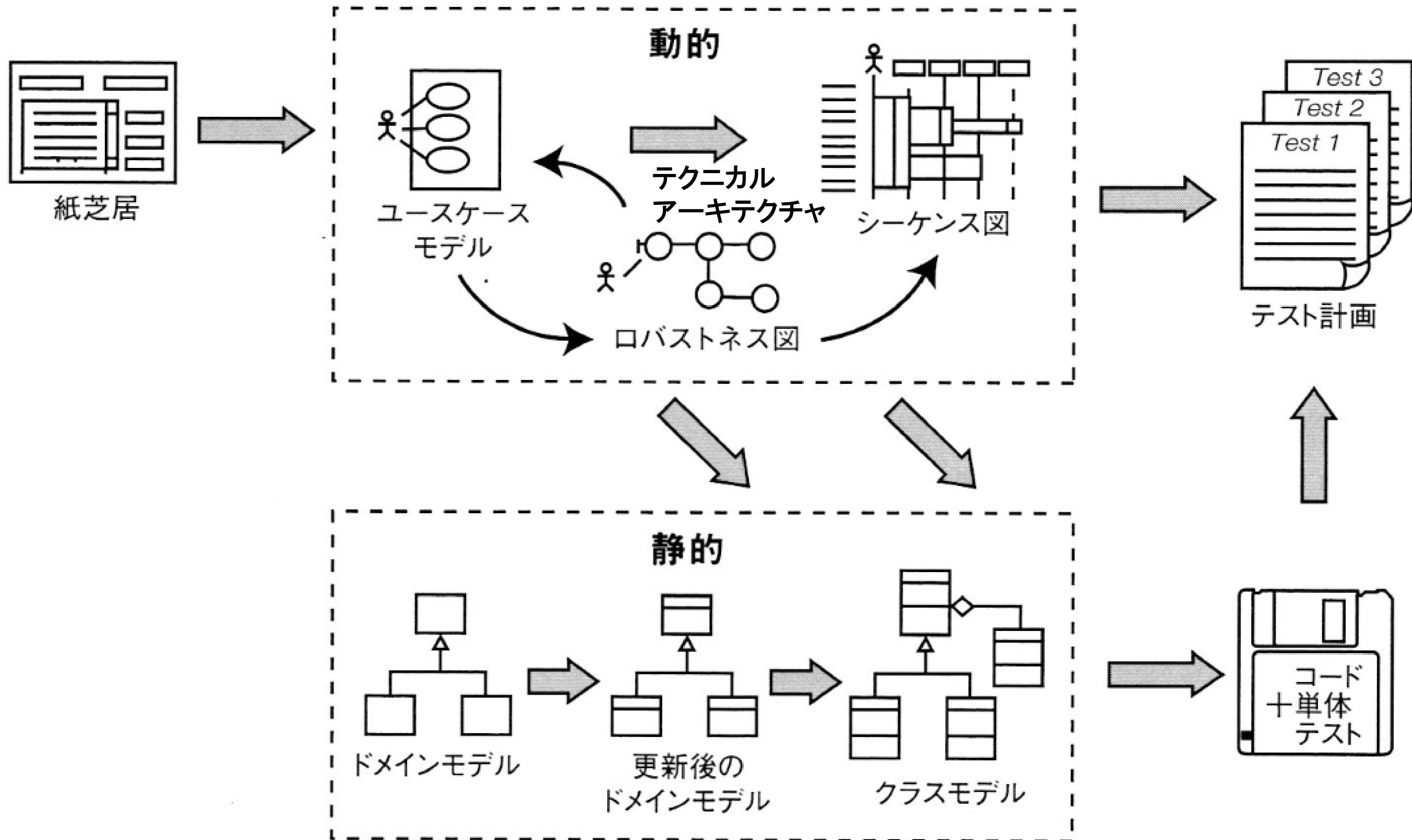
海谷 治彦

目次

- 詳細設計のレビュー
- アーキテクチャ決定について

- データフロー図
- アーキテクチャ選択の方法

ICONIXの全体手順



動機: 予備設計のレビュー

- 現時点で，ユースケース，ドメインモデル，ロバストネス図を描きました.
- これらに整合性があるかのチェックを行います.
- 整合性をとること自体が目的ではなく，
- 整合性をとりながら，それぞれに欠けてる情報，間違った情報を直していく，というのが目的です.

- その他，エンティティへの属性の付加，
- システムの全ての画面に名前を付けることが目的となります.

予備設計レビューガイドライン 1/2

1. ユースケース毎に，ユースケース記述とロバストネス図の刷り合わせをしましょう．蛍光ペンを使ったチェックが有効です．
   - ツールを使うなら，エクセルとastahでしょうか．
2. ロバストネス図中の全てのエンティティが，ドメインモデルにあるか確認し，無ければ追加してください．
3. エンティティと画面の間で，データの流れを確実に追跡できるようにしなさい．
   - ロバストネス図における B-C-E の関係
4. 代替，例外コースの漏れが無いか，チェックしてください．
   - 特に例外．
5. 各ユースケース記述が，アクターからの対話，システムからの対話を書いてあるかチェックしてください．
   - 特にシステムが主語のもの．

予備設計レビューガイドライン 2/2

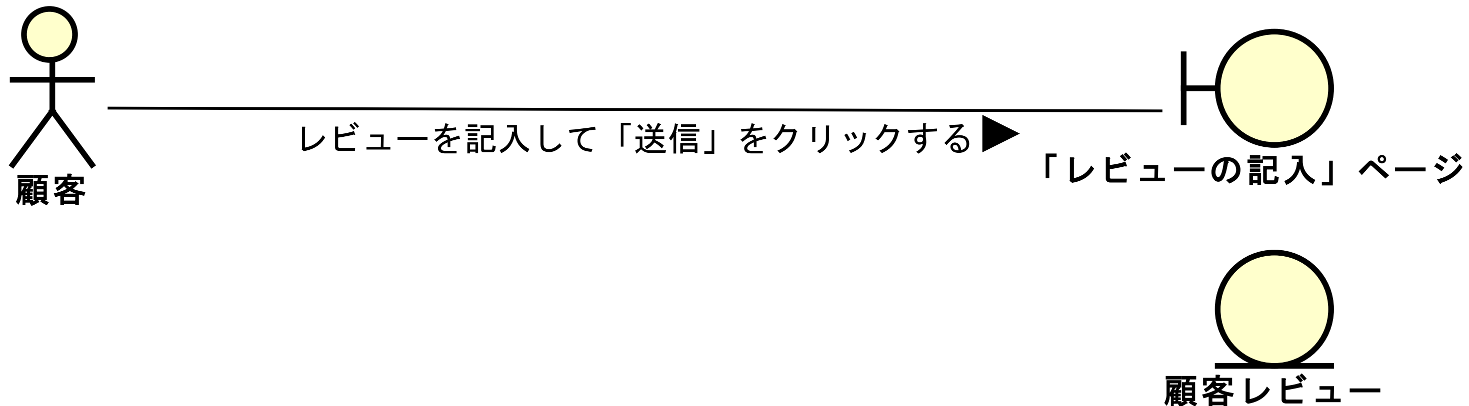
6. ロバストネス図の構文ルールが守られているかチェックしてください.
   - 名詞-名詞は不可、すなわち B-B B-E E-E は不可.
7. 技術者以外(顧客や営業等)と技術者の双方がレビューに参加できるようにしてください.
   - 双方が理解できるユースケース記述が丁度良い抽象化レベルのものになります.
8. ユースケースがドメインモデルとGUIの用語で記述されていることを確認しなさい.
9. ロバストネス図でシーケンス図(次回)上に表現するような詳細レベルを示さないでください.
10. より良い設計のために後述の「6つの手順」に従ってください.

6つの手順

1. ロバストネス図がユースケース記述に合致しているかを確認する.
2. ロバストネス分析の規則に従っているか確認する.
3. ロバストネス図がユースケースの論理的な流れに注視しているかを確認する.
4. 代替，例外コースがロバストネス図に示されているかを確認する.
5. 図がデザインパターン(後半の回で解説)に固執しないようにする.
6. 図が詳細設計に踏み込んでいないかを確認する.

レビューの例A 1/3

- 「顧客レビューを書く」のユースケース.
- 以下では レビュー:E が, 「レビュー記入」ページ:B と関連付いてない.
- 要は入れたレビューが入れ物であるEに入ってない.

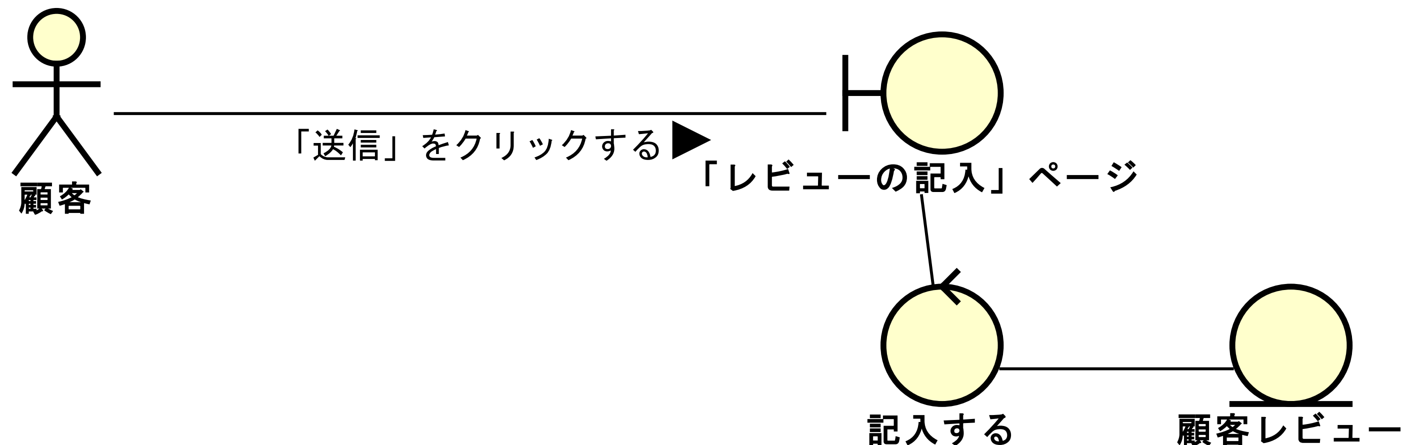


レビューの例A 2/3

- ユースケース記述を良く思い出すと，
  - 顧客評価を入力する
  - レビューを記入する

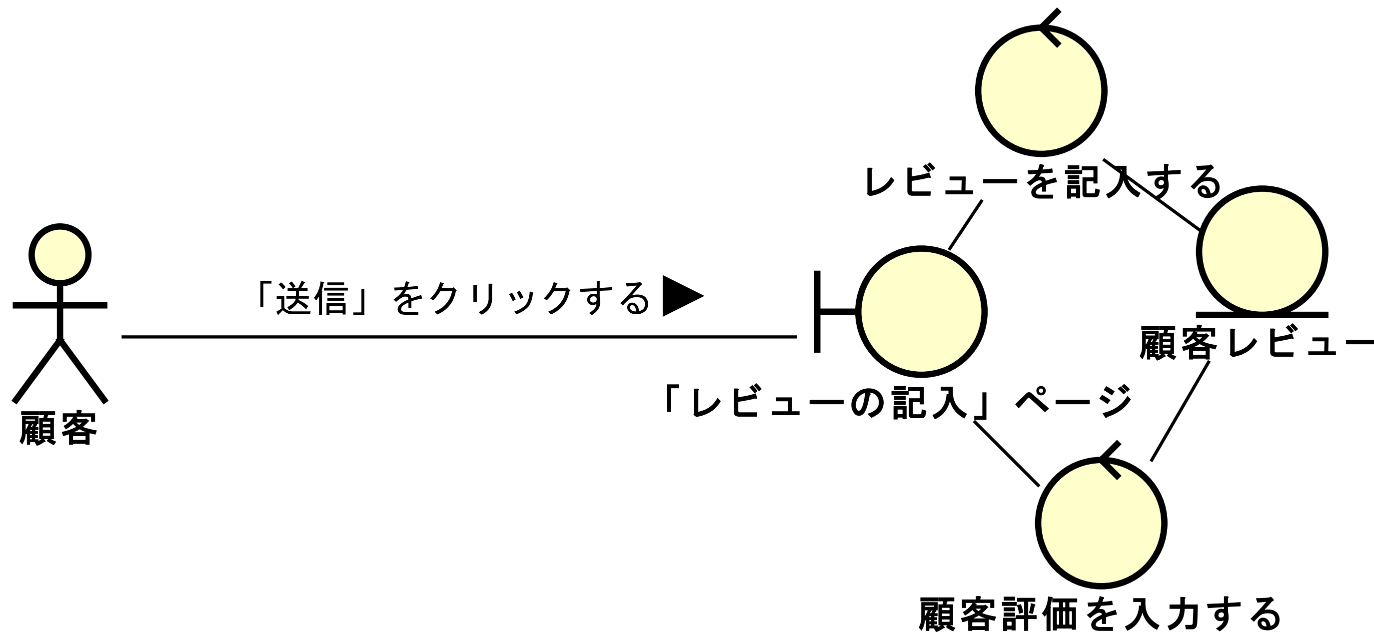
の二種類があったが，以下ではこれらを示せない．

- 加えて，アクター-バウンダリ間の名前も変更．



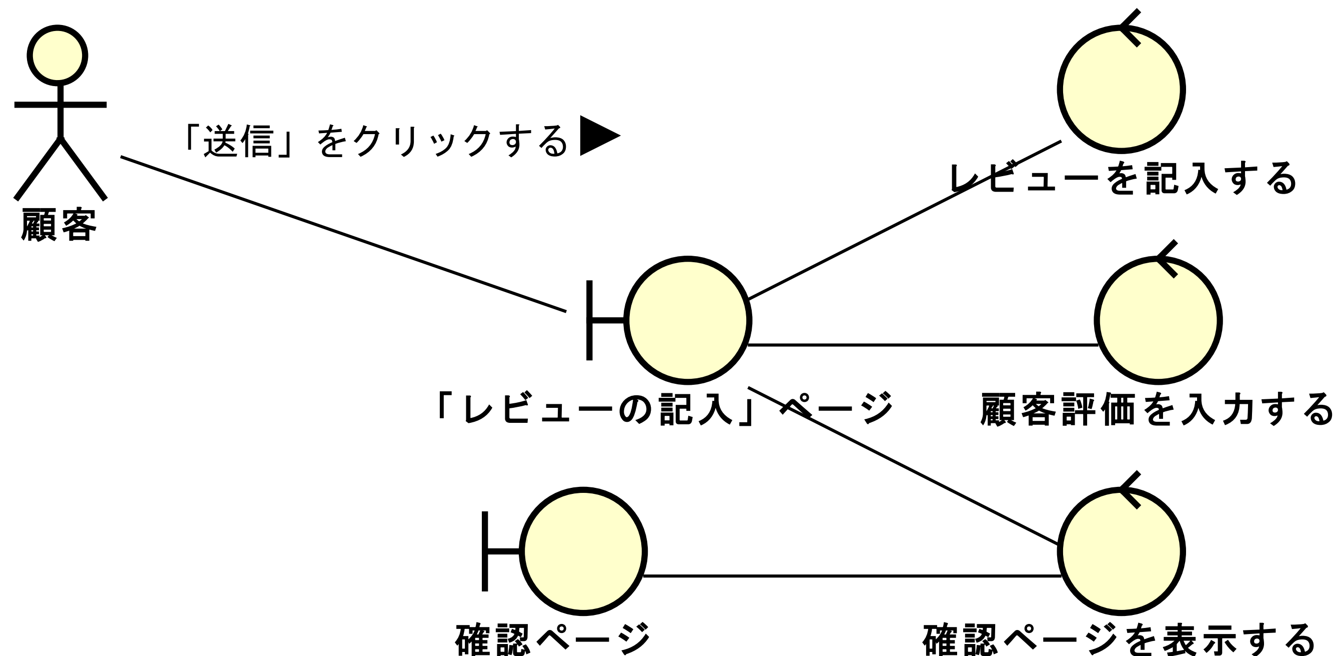
レビューの例A 3/3

- そこで，記入，入力それぞれに対応したコントローラを，それぞれに導入．



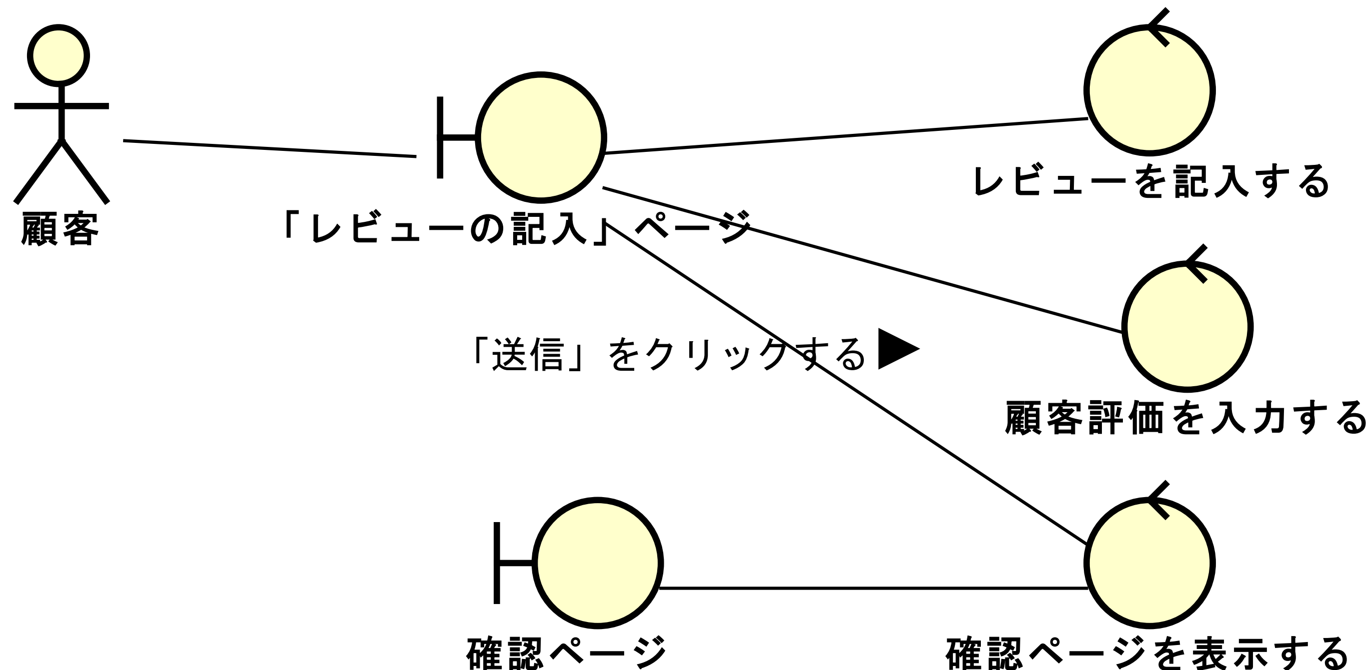
レビューの例B 1/2

- 「顧客レビューを書く」をさらにレビュー
- 以下では，クリックによって，3つのコントローラのどれが機能するかわからない．



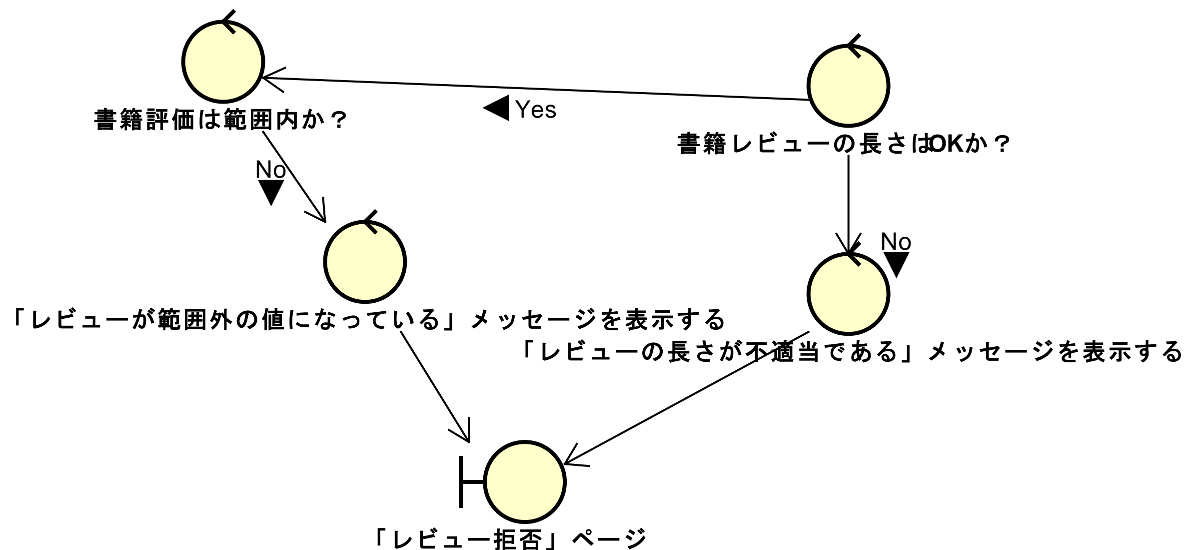
レビューの例B 2/2

- ラベルの位置を変えるだけで，図の曖昧さが排除された.



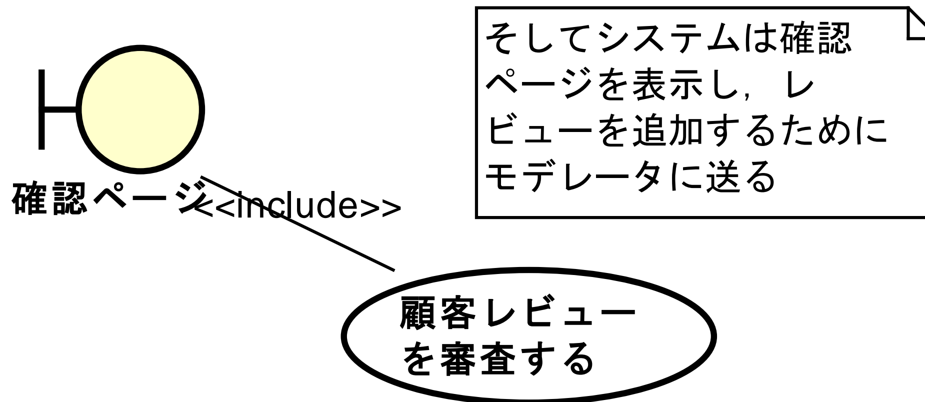
レビューの例C

- コントローラーの名前として，単に「表示する」，「登録する」とすると，何を表示するのかわからない．
- 幸運にも，astah では，「表示する」という名前のコントローラ(実体はクラス)は複数書けない．
- よって，それぞれ，内容に応じて，「・・・を表示する」等と詳しくかくのがよい．



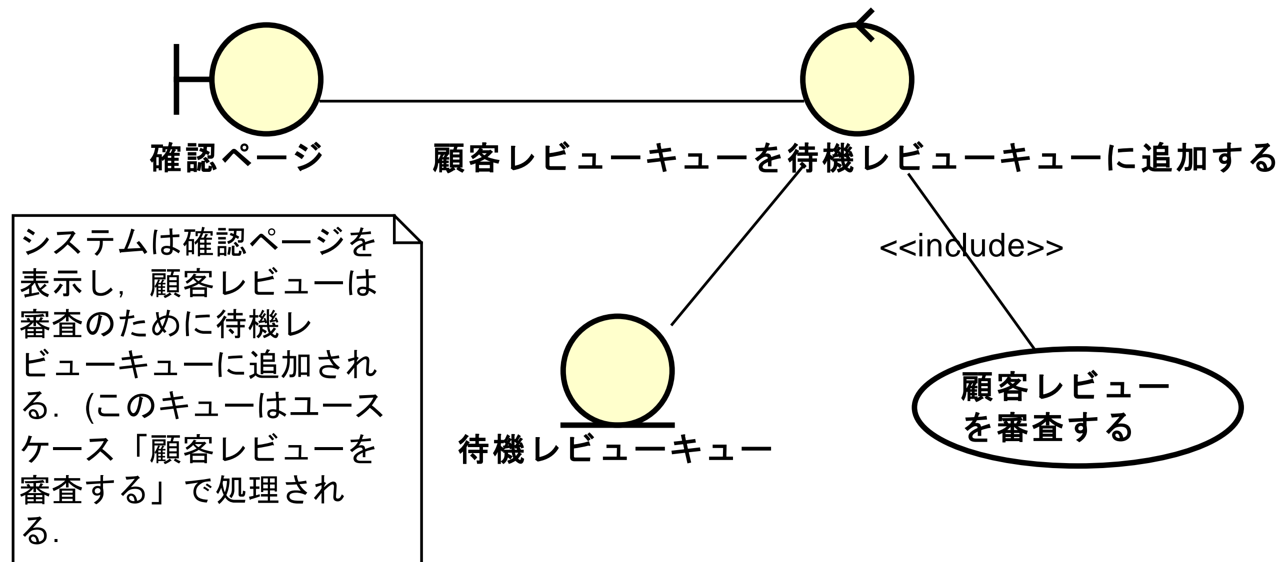
レビューの例D 1/2

- 「顧客レビュー」と「レビューを審査する」の関係.
- 当面，以下な感じだった.
- 二つのユースケース間で実際にレビューが，どのように受け渡されるかが未指定のままである.

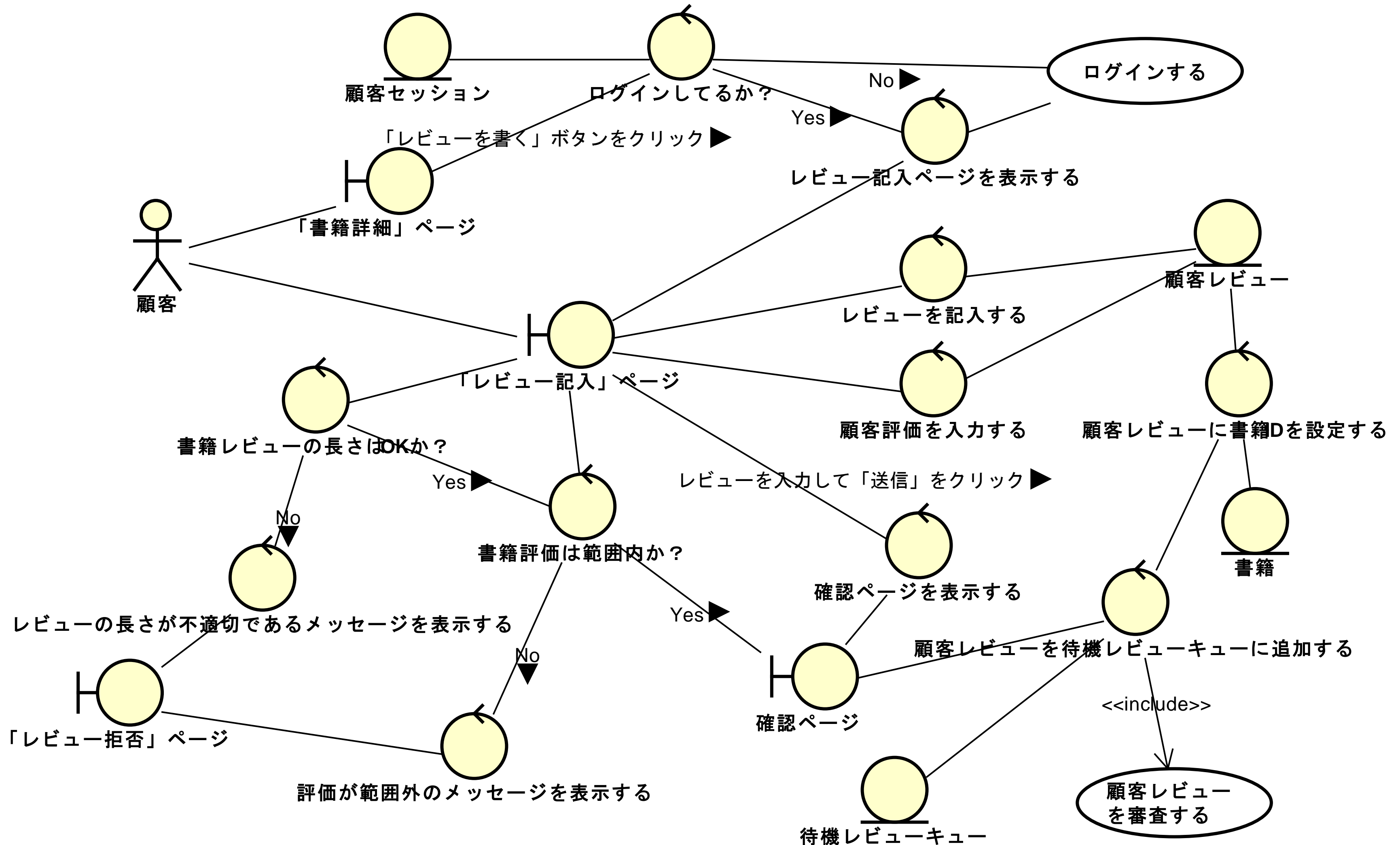


レビューの例D 2/2

- ドメインモデルではユースケース間で共有されるデータに相当するエンティティクラスを導入することで，前述の点を明確にする.
- この例では，審査待ちのレビューをキュー(待ち行列)に置くことにした.



顧客レビューを書く最終版



対応するユースケース記述は teams p176ucd.xlsx より参照のこと.

アーキテクチャ

アーキテクチャとは？

- Architecture もともとは建築用語
- コンピュータでは CPU やハードの構造のことも指す
- システム アーキテクチャ，ソフトウェア アーキテクチャとも呼ばれる.
- 以下を図式で書くことを指す場合が多い.
  - システム内のソフトウェア部品間の論理的な構造
  - システムが動くハードウェアと通信路上のソフトウェア部品の配置位置.

- レスポンスの速さ，処理能力，信頼性等のシステムの品質はアーキテクチャに依存する場合が多い.

ソフトウェアの部品

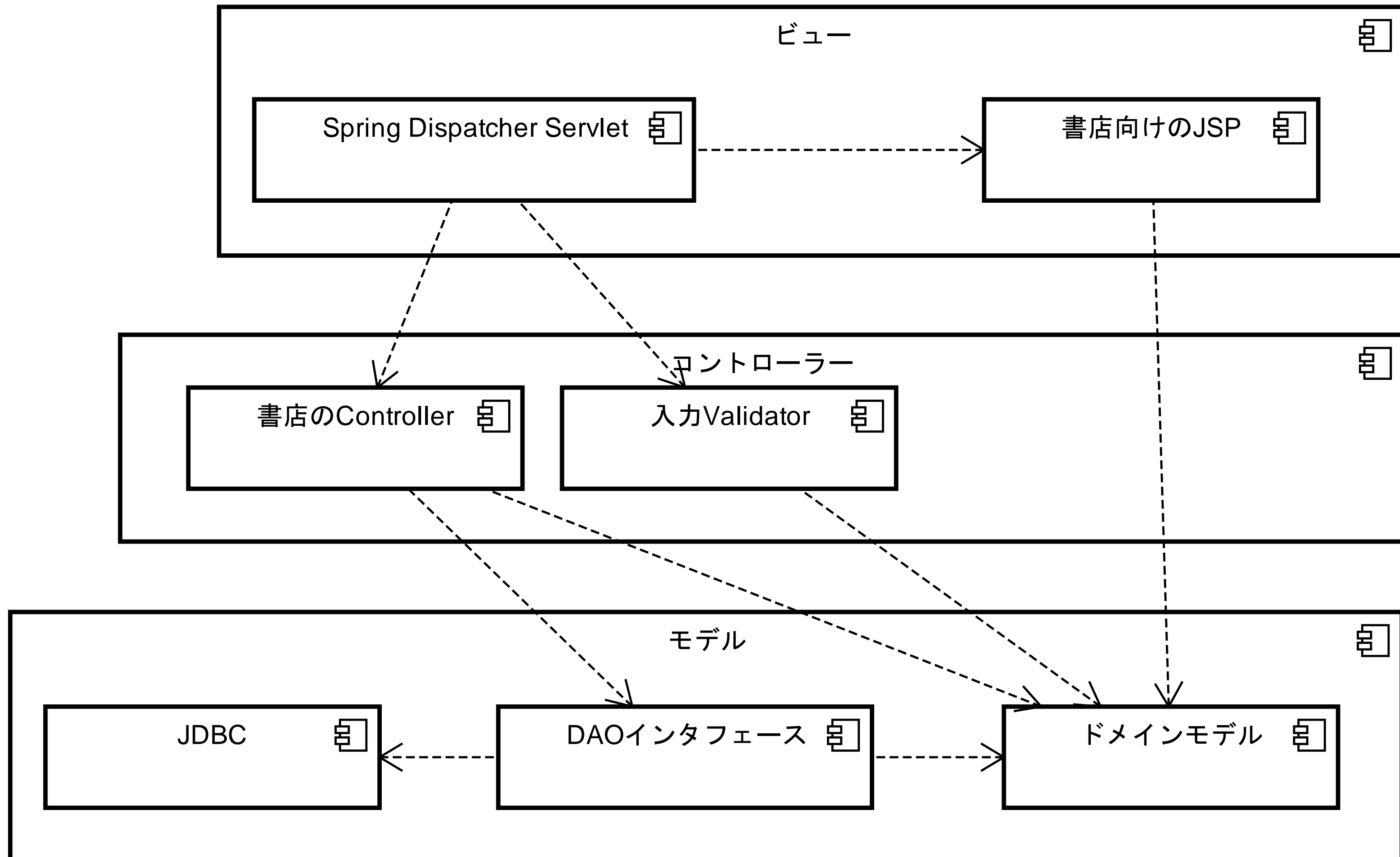
- 昨今のソフトウェアは複数のプログラムで構成される場合が多い.
  - 例 ウエブサーバー，データベースサーバー，ウエブブラウザ(UI担当)
- また，既存のライブラリやフレームワーク(後述)を使うことも多い.
  - 例 暗号関係部品(openssl等)，UIフレームワーク(Bootstrap等)，アプリケーションフレームワーク(cakePHPやPlay等)
- 上記の意味からプログラム，ライブラリ，フレームワーク等をソフトウェアの部品と呼ぶ.

アーキテクチャの例

- 以降にネット書店のアーキテクチャを示す.

- ソフトウェア部品間の依存関係
- 部品の配置図
- メッセージの流れ

ソフトウェア部品間の依存関係

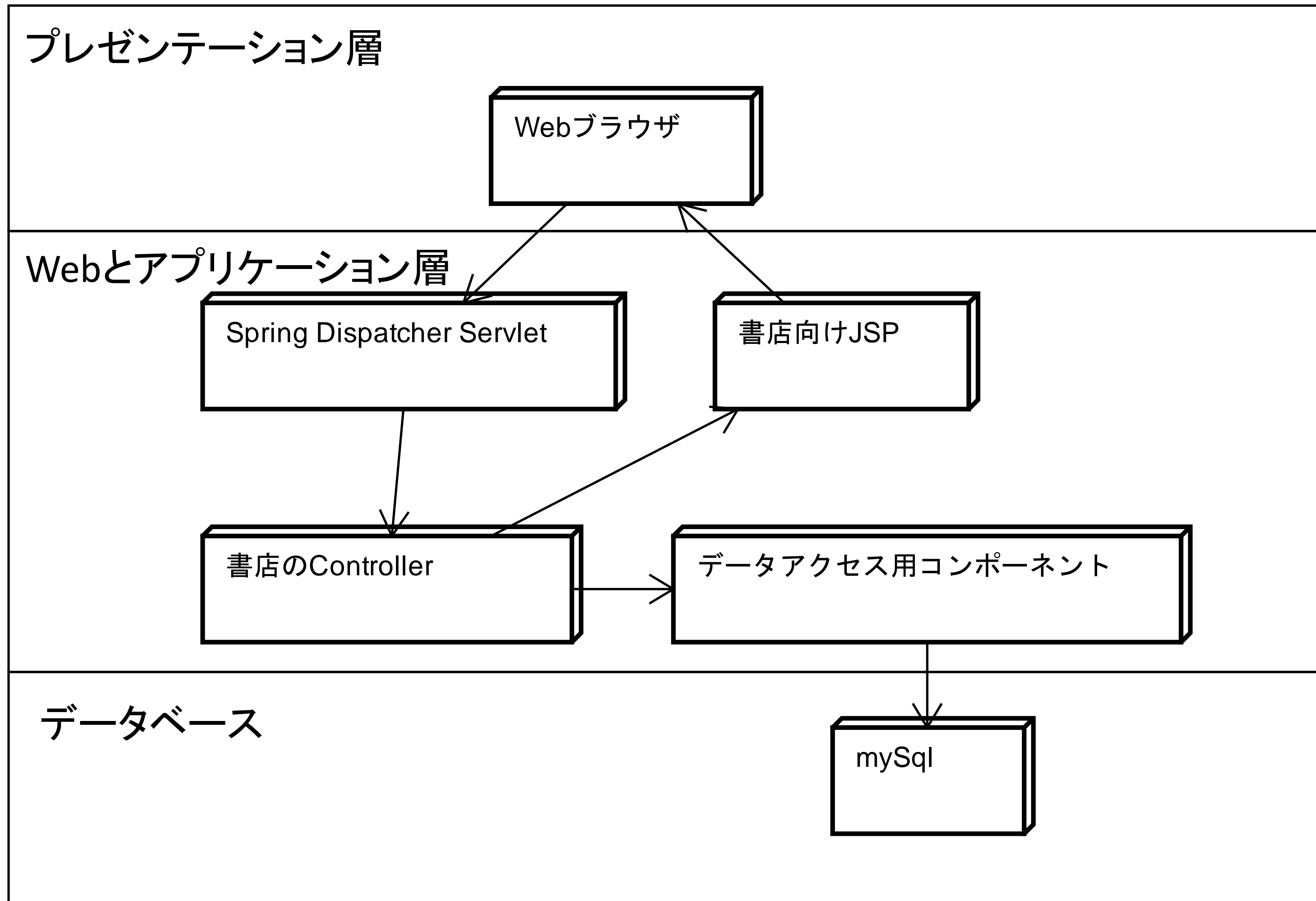


復習

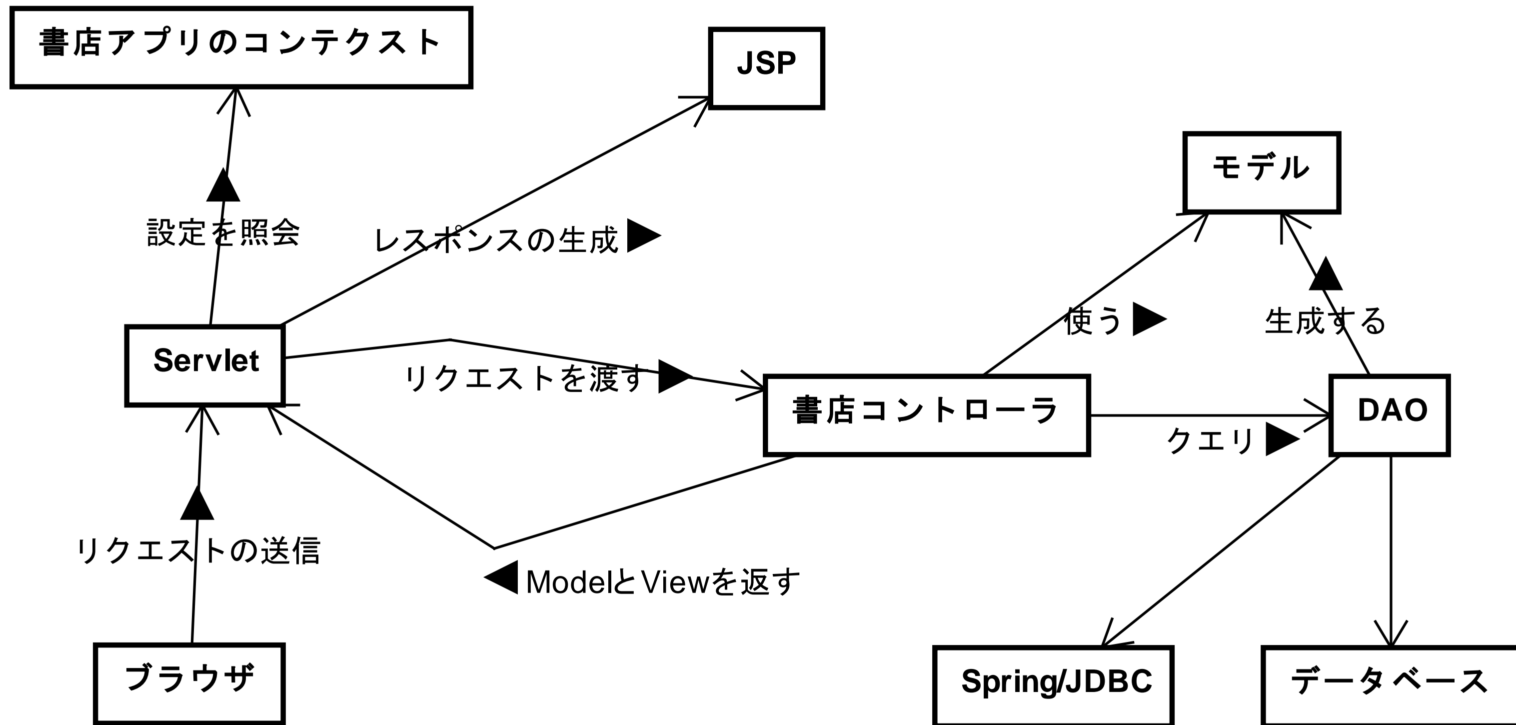
MVCについて

- Model-View-Controller の略.
  - オブジェクト指向プログラミングで習ったかもしれない.
- アプリケーションを作る際に上記の三つに分けて設計すると良いという指針.
- Model
  - アプリで扱う業務や活動のみを扱う部分.
  - ショッピングサイトの業務なら商品，注文，顧客等がコレに相当.
  - 基本，システムとは関係ない業務依存の部分.
  - 主に普通のクラスやJavaBeans等で実現される.
- View
  - システムとしてユーザーと相互作用する部分．入出力.
  - ウエブアプリならウエブページに相当し，主にJSPが担当.
- Controller
  - ModelとViewを関連付け，業務の進行を制御する部分.
  - 主にServletが担当.

部品のレイヤー別の関係



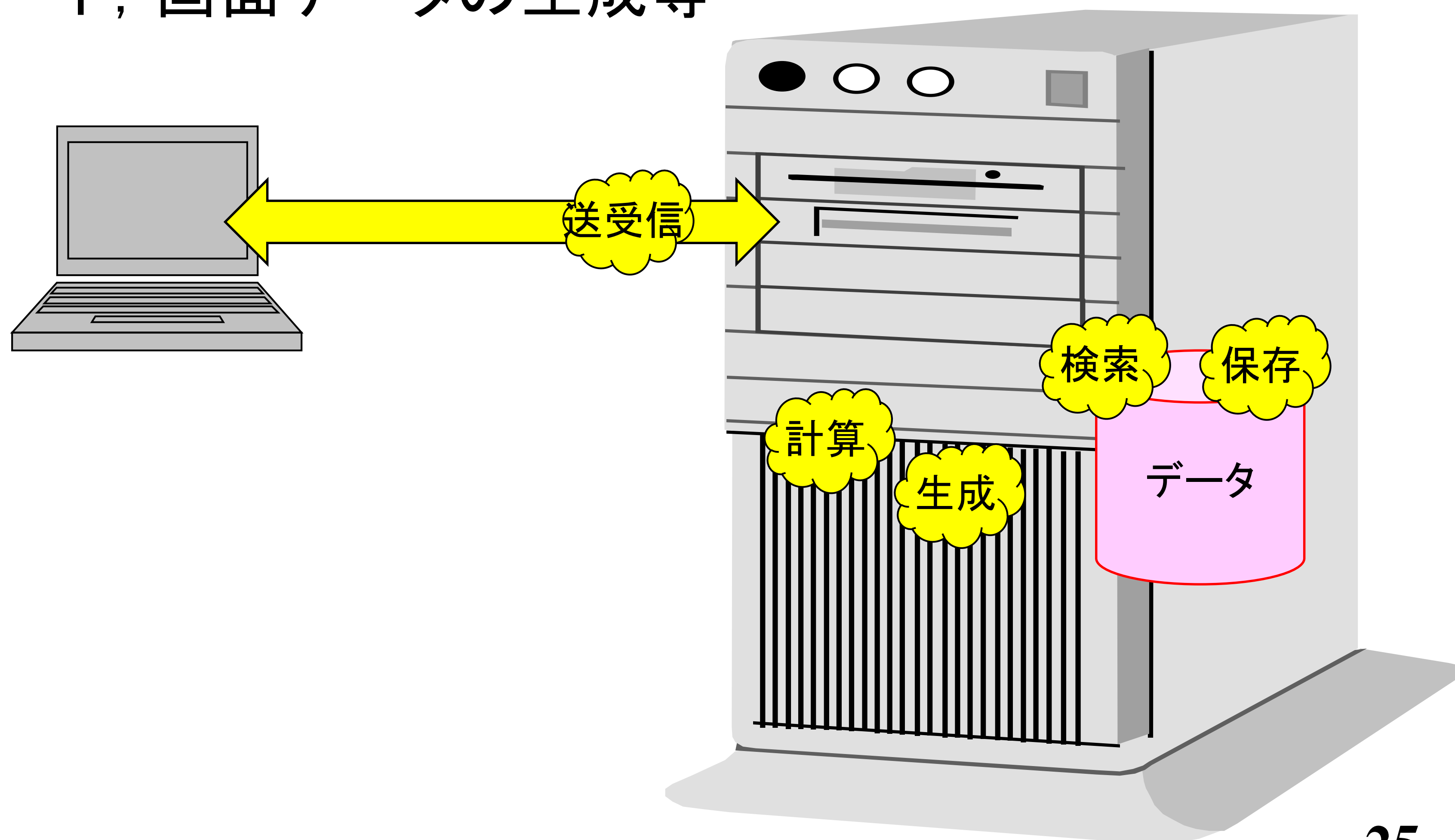
メッセージの関係



参考

サーバーサイドの主な役割

- 検索，計算，データ保存や共有，手順のナビゲート，画面データの生成等



参考

JSP and Servlet

- 共通点
  - どちらもサーバーで実行される.
  - すなわち，**クライアントに届いた時にはただのHTML**
    - コレがJavaScriptとの大きな違い
- JSP
  - HTMLにJavaっぽいものを埋め込む.
  - 雰囲気はJavaScriptに似てるが，上記のようにサーバー内で実行される.
    - phpはコレに考え方が似ている.
  - オリジナルのクラス等を作成する場合は，Servletとの連携が必要.
- Servlet
  - Javaそのもの.
  - mainメソッドは書かないのが普通.
    - そもそも スーパークラスがフレームワーク的にできている.
  - Javaのprint機能でHTMLの行を表示しないと，クライアント側で解釈不能になる.
  - どちらかといえば，JSPのサブルーチン的に使われるのが普通.

参考

サンプル

```
<HTML>
<HEAD>
<TITLE>
JSP loop
</TITLE>
</HEAD>

<BODY>
<ul>
<%
int i;
for(i=0; i<10; i++){
%>
<li> number <%= i*3 %>
<%
}
%>
</ul>
</BODY>

</HTML>
```

```
// Simple Servlet
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Another extends HttpServlet {

  public void doGet(
      HttpServletRequest request,
      HttpServletResponse response)
    throws IOException, ServletException
  {
     response.setContentType("text/html");
     PrintWriter out = response.getWriter();
     out.println("<html>");
     out.println("<head>");
     out.println("<title>Another!</title>");
     out.println("</head>");
     out.println("<body>");
     out.println("<h1>Another!</h1>");
     out.println("</body>");
     out.println("</html>");
  }
}
```

フレームワークとは？

- 枠組み (これじゃ意味がわからん･･･)
- ライブラリの逆
- ライブラリは全体(main関数的なものを含む)は自身で作り，部分は他人の作ったものを流用する.
  - C言語の数値計算ライブラリ lm
  - Javaの各種API
- フレームワークでは全体は他人が作ったものを使い，部分を自分でカスタマイズする.
  - 多くのフレームワークでは業務の大まかな流れが想定されている.
  - サーバーサイドのアプリケーションフレームワーク. (spring や cakePHP等)

Springについて

- Javaベースのアプリケーションフレームワーク
- ServletとJSPを使う.
- 普通のJavaのクラス(POJO: Plain Old Java Object)をドメインモデルとして使いプログラミングできる.
- DAO (Data Access Object)も提供する.
  - データベースへのアクセスを抽象化，一般化しているクラスのこと.
  - データベースの詳細(SQL等)や種類の違い(MySQL or postgresql等)を吸収してくれる.

アーキテクチャ決定 Not TO DO!

1. ハードウェアやそのコストを考えずにアーキテクチャを決定する.
2. 「いままでこうやってきたから」ということで先祖伝来のアーキテクチャを使い続ける.
3. スケーラビリティを考慮しない.
4. セキュリティを考慮しない.
5. 市場や流行にふりまわされない.

続き

6. プロジェクトの要求に基づきアーキテクチャの目的を明確化できない.
7. 設計に入る前にアーキテクチャについて必要以上に長い時間を費やす.
8. システムをテストする方法についての考慮をするのを忘れる.
9. ユーザーが必要としていることを考慮せずに, アーキテクチャを定義しようとする.
10. アーキテクチャの構築を一切行わない.

次の話題

- データフロー図
- アーキテクチャ選択の方法例
  - アーキテクチャ記述を用いたセキュリティ分析の例

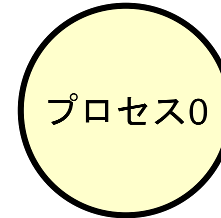
データフロー図

- 通称，DFD
  - Data Flow Diagram
- 処理と入出力データの関係のモデル.
- C言語等の手続き型言語との親和性が高い.

- UMLの一部ではない.
- 年配(50台以上)のSEなら，誰でも知っている.
- astah で記述できる.

記法

- プロセス
  - 処理を表す.
  - 関数とほぼ対応する概念.



- データフロー
  - データの入出力を示す線.
  - 関数の引数や返値に対応
- エンティティ
  - システム外部とのインタフェース

外部エンティティ0

- データストア
  - データを保持するデータベース的なもの.
  - 画としてはコンデンサ(電子部品)のアイコン.

データストア0

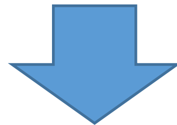
別途Webページにある例で説明

- 簡単な記法から本格的なモデルまで.

アーキテクチャ記述を
利用したセキュリティ分析例

そもそもアーキテクチャとは何か？

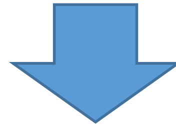
- 比較的，大きなソフトウェアは単体の実行プログラム(.exe)だけでは構成されない.
  - 複数のプログラムから構成される.
- 複数の実行系やライブラリ(部品)から構成され,
- それらが，異なるマシン上で動き,
- 必要に応じて通信等を行う.



- アーキテクチャとは，ソフトウェアが,
- どのような部品から構成され,
- それぞれの部品がどこに配置され,
- 部品間がどう通信するかを示したもの.

アーキテクチャの多様性

- 同じ仕様のソフトでも,
- 部品の分け方が異なる場合がある,
- 各部品の配置場所も異なる場合がある.



- アーキテクチャの決定はソフトの機能以外で決まってくる.
- 設計や実装の制約.
  - 既にあるマシンに配置しないといけない・・・
  - スマフォが流行ってるのでアプリにしないといけない
- 品質特性.

品質特性の分類例 FURPS+

例：FURPS+

機能要求・非機能要求	要素	内容
機能要求	Functionarity：機能性	システム機能
非機能要求	Usabirity：操作性・有用性	使いやすさ、表現の美しさ
	Reliability：信頼性	故障頻度、復旧のしやすさ
	Performance：性能	処理速度、資源（例：メモリ）使用量）
	Supportability：保守性・保全性	テスト容易性、拡張性、適応性、保守性
	＋plus：その他	

ソフトウェア品質特性標準 – ISO 9126

- **ソフトウェア品質特性**に関する標準
- http://www.cam.hi-ho.ne.jp/adamosute/software-quality/iso9126.htm より

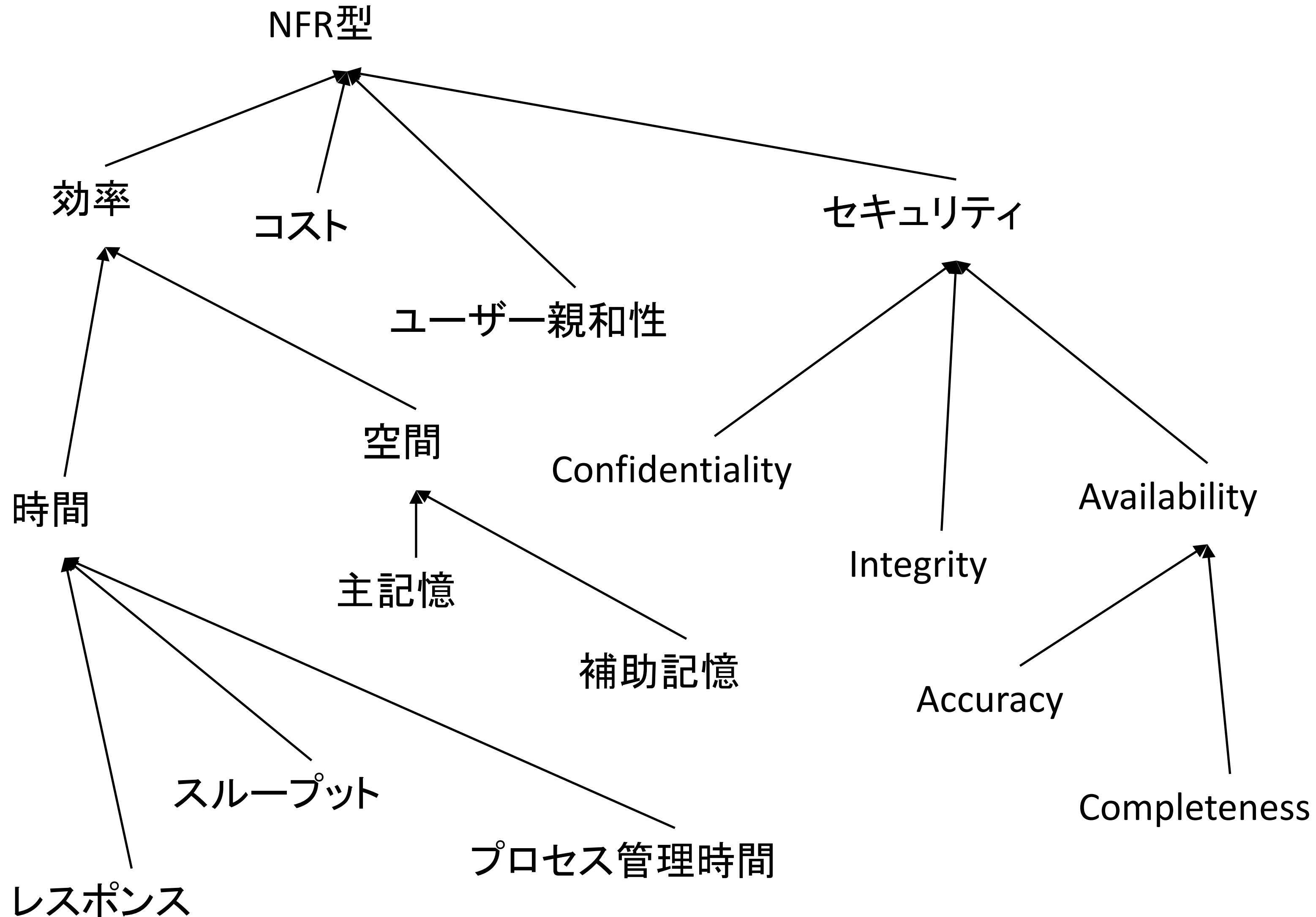
品質特性(quality charastristics)	品質副特性(quality subcharactristics)	主な内容
機能性(fuctionality)	合目的性(suitablity) 正確性(accuracy) 相互運用性(interoperability) 標準適合性(compliance) セキュリティ(security)	目的から求められる必要な機能の実装の度合い
信頼性(reliability)	成熟性(maturity) 障害許容性(fault tolerance) 回復性(recoverability)	機能が正常動作し続ける度合い
使用性	理解性 習得性 運用性	分かりやすさ、使いやすさの度合い
効率性	時間効率性 資源効率性	目的達成のために使用する資源の度合い
保守性	解析性 変更性 安定性 試験性	保守(改訂)作業に必要な努力の度合い
移植性	環境適用性 設置性 規格適合性 置換性	別環境へ移した際そのまま動作する度合い

NFR framework

- NFR=Non-Functional-Requirements=非機能要求
- 機能ではなく性能等に関するゴール(NFR)の雛形をあらかじめ与えて,
- NFRの扱いを見落とさないようにする支援とする.

- Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. **Non-functional Requirements in Software Engineering**. Kluwer Academic Publishers, 2000. ISBN 0-7923-8666-3, 439 pages

NFRの型



Security NFR

時節柄，注意しなければいけないゴール

- Confidentiality 許可されていないアクセスからデータを守ること．秘密にすること．
  - 一番，よく使われるセキュリティの意味．
- Integrity 不正改竄されてないこと．
  - Accuracy 正確である．
  - Completeness 完全である．
- Availability 不正なサービス割り込みを抑制すること．

アーキテクチャ決定のキー

- アーキテクチャ決定に際しては，設計・実装の制約や，品質要求を考慮しなければならない．
- 以降では，品質特性を考慮してアーキテクチャを選ぶための手法を紹介する．

背景

- 品質要求(セキュリティやパフォーマンス等)は，実現するシステムアーキテクチャに関係なくステークホルダが望むものである.
- しかし，現実には品質要求が達成できないことがわかって，初めて「それは必要だ」と気づくものである.
- 品質要求の達成の可否の多くは，アーキテクチャに依存している.
- アーキテクチャは思ったほど自由度は無く，例えば，既存のマシン配置やサーバー群を使わざるを得ないことも多い.

実例

- 昨今のdotcampusの挙動
- 本学のVDI (Virtual Desktop Infrastructure) の機能不全による企画倒れ.

- マイナンバーカード関係のサーバーのダウン
- オリンピック等の切符販売サイトのフリーズ

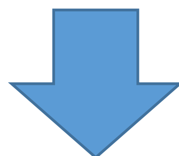
- わりと笑えない事例が満載.

目的

- セキュリティ要求獲得の支援法を考える.
  - 品質要求と一例として.
- 想定されるアーキテクチャ下で，与えられた機能要求を実現すると，どのような品質要求へのリスクが発生するかを見える化()する.
- このリスクの可視化を通して，品質要求達成の危機を認識してもらい，品質要求の獲得を行う.

動機付けの例

- 機能要求:「ウエブサイトを通して，買い物を連続して行い，同時決済したい．」
- アーキテクチャ1: クッキーベース
  - 認証情報，カート情報が毎回ネットを通る.
  - 通信路に関する強いガードが必要.
- アーキテクチャ2: セッションベース
  - 基本，二回目以降のページ遷移では，セッションIDしかネットを通らない.
  - セッション乗っ取りが起こらない程度のガードでOK.



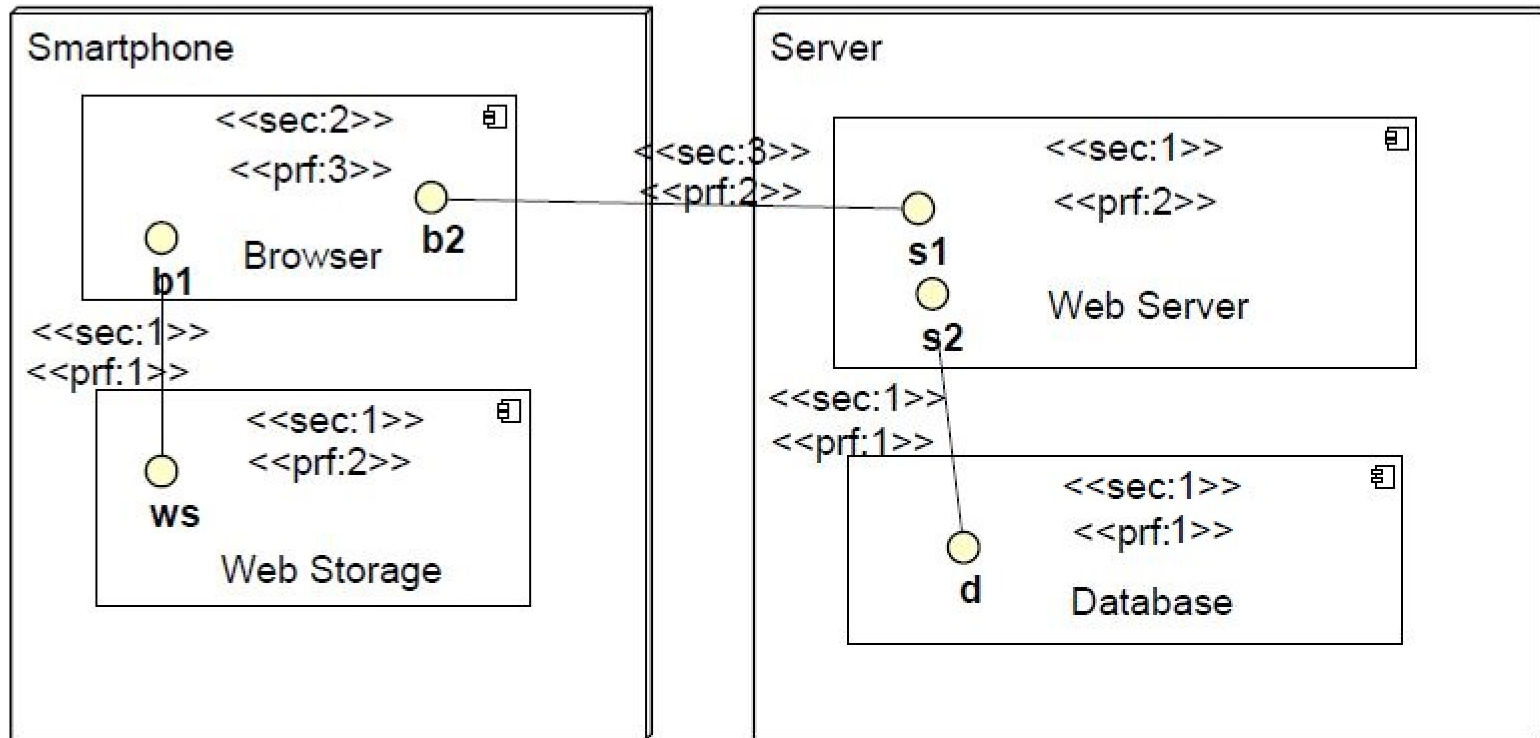
- ソフトウェアのアーキテクチャの違いによって保護対象のアセットもリスクも変わる.

手法 RiMALL

- Risk Model with Architecture and Likelihood Level
  - 大久保教授(IISEC)が命名
- データフロー図(DFD)と配置図(DD)を合体させたアーキテクチャ記述図の名称.
- この図で個別の具体的なシステムアーキテクチャと品質特性毎のリスクの高さを記述する.
- 与えられた機能要求を, RiMALLインスタンス上のデータフローにマッピングすることで,
  - 全体的なリスクの高さ
  - 品質特性の観点から対処しないといけない点

を明確にする.

RiMALLインスタンスの例



後にアーキテクチャC2と呼びます.

モデルの解説

- ○と線がデータフローです．データの通り道なので，正規のDFDと違い方向性はありません．
  - フローに名前をつければよかったのですが，現時点では端点に名前がついています．(ちょっと無駄)
- コンポーネントとデータフローに品質毎のリスクの高さに関する情報をステレオタイプとして付記してます．
  - 現時点では，sec=セキュリティ prf=パフォーマンス しか扱ってません．
  - 3: 高いリスク 2: 中くらい 1: 低いリスク
- このモデルは，機能要求に関係なく，事前に実装の専門家とともに準備しておきます．

手法全体の流れ

- 以下は準備されているものとする
  - 機能仕様書
  - RiMALL アーキテクチャモデル群

1. 機能仕様のRiMALL上へのマッピング
2. 各アーキテクチャでのリスクの計算
3. リスク値に基づくアーキテクチャの選択
4. 選択されたアーキテクチャおけるリスク原因に対する対処としての品質要求の獲得.

- 手法の結果
  - 選ばれたアーキテクチャ
  - 品質要求 (当該アーキテクチャで対処する事項)

マッピングとリスク計算

- 機能要求は，自然言語のユースケース記述(UCD)で提供されるものとしている.
- UCD中で異なるコンポーネント間を通るアセットを識別し，そのアセットのデータフローを文から分析者が見つける.
- 各データフローのパスをRiMALL上で識別する.
- パス上の likelihood level を掛け合わせる.
  - $\Pi\ ll$ とする.
- 当該アセットのpotential lossを分析者が決める.
  - *lo* とする，これも 重大:3 普通:2 低い:1
- データフローのリスク = $lo * \Pi\ ll$ とする.
  - 大きいほどマズい.
- 全機能の全データフローの和をあるアーキテクチャのある機能要求群に対するトータルリスクとする.

選択と対処

- 基本，トータルリスクが最も低いアークテクチャを選択する.
- 選択されたものでもリスクがゼロであることは無いので，リスクの点数を逆算して，対処すべきコンポーネントや通信路を識別する.
  - 点の大きいところを重点対処
  - 通信路やコンポーネントに対して，CIAのどれを考慮すべきか等を書く.
  - これが品質要求に気づくトリガーとなる.
    - 品質仕様ともいえる.
- 機能要求に品質要求を足して，要求仕様を完成に近づける.

適用事例

- 二つの機能を有する仕様に対して，4つの異なるアーキテクチャについて，リスク計算をしました．
- 結果は直感に合うものでした．

機能要求

- F1 イベントへの個人情報登録
  - 利用者はイベントを選択し, 個人情報等を入力する.
  - システムは承った旨を表示する.
- 下線をそれぞれアセット D1-1, D1-2 とする.

- F2 イベント詳細情報の通知
  - システムはイベントの詳細を登録されたメアドに送る.
- 下線をそれぞれアセット D2-2, D2-1 とする.

アーキテクチャ候補群

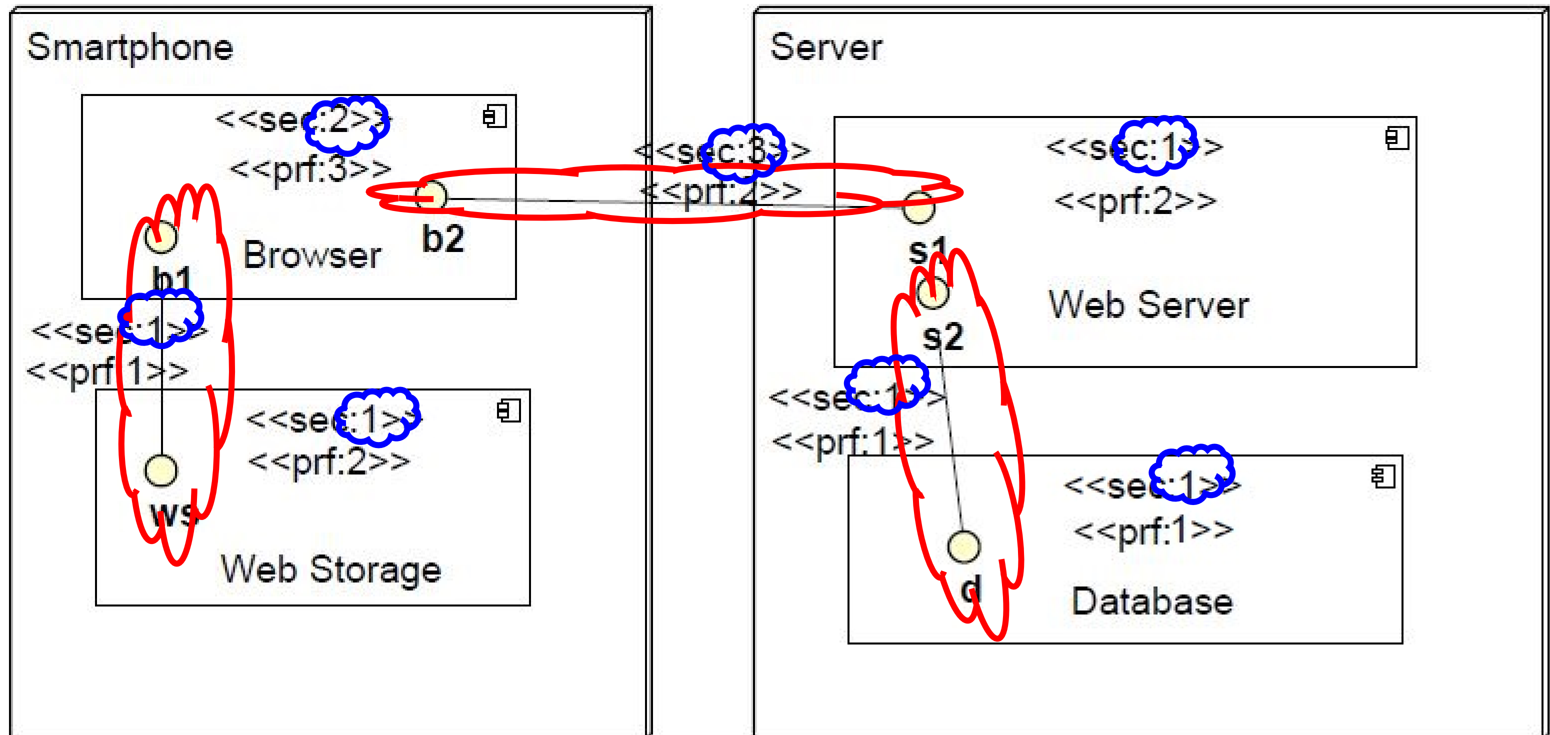
- C1 ブラウザ + FS
  - ウエブアプリだがデータはサーバーサイドのファイルシステム(FS)に直書き．古いCGI的なイメージ．
- C2 ブラウザ + DBMS
  - いまどきのウエブアプリ．
- C3 ネイティブアプリ + FS
  - ウエブでは無く，スマホのアプリのようなネイティブアプリを想定．サーバーサイドは古いCGIっぽい感じ．
- C4 ネイティブアプリ + DBMS
  - ウエブでは無く，スマホのアプリのようなネイティブアプリを想定．サーバーサイドはDBMS使用．

事例の手順

- C1からC4のRiMALLをセキュリティ専門家(実務経験あり)が書きました.
- セキュリティ専門家とソフトウェア専門家が共同で各RiMALLとアセットの流れの対応をつけました.
- 手法に基づき，セキュリティ，パフォーマンスのリスクスコアを計算しました.

C2にD1-1をマッピング

- In F1: 利用者はイベントを選択し，個人情報等を入力する．
- $\Pi\ ll = 1*1*2*3*1*1*1=6$
- $lo = 3$ (分析者が決めた)



セキュリティのスコア

Case	Func.	Data	Security Loss	Security Likelihood	Security Risk
C1	F1	D1-1	3	18	54
C1	F1	D1-2	1	6	6
C1	F2	D2-1	3	3	9
C1	F2	D2-2	1	6	6
				Total Security Risk:	75
C2	F1	D1-1	3	6	18
C2	F1	D1-2	1	6	6
C2	F2	D2-1	3	1	3
C2	F2	D2-2	1	6	6
				Total Security Risk:	33
C3	F1	D1-1	3	81	243
C3	F1	D1-2	1	9	9
C3	F2	D2-1	3	3	9
C3	F2	D2-2	1	9	9
				Total Security Risk:	270
C4	F1	D1-1	3	27	81
C4	F1	D1-2	1	9	9
C4	F2	D2-1	3	1	3
C4	F2	D2-2	1	9	9
				Total Security Risk:	102

前頁に
対応

コレは
アーキテクチャ
非依存

パフォーマンスのスコア

Case	Func.	Data	Performance Loss	Performance Likelihood	Perfomance Risk
C1	F1	D1-1	1	72	72
C1	F1	D1-2	1	12	12
C1	F2	D2-1	1	6	6
C1	F2	D2-2	2	12	24
				Total Performance Risk:	114
C2	F1	D1-1	1	24	24
C2	F1	D1-2	1	12	12
C2	F2	D2-1	1	2	2
C2	F2	D2-2	2	12	24
				Total Performance Risk:	62
C3	F1	D1-1	1	24	24
C3	F1	D1-2	1	4	4
C3	F2	D2-1	1	6	6
C3	F2	D2-2	2	4	8
				Total Performance Risk:	42
C4	F1	D1-1	1	8	8
C4	F1	D1-2	1	4	4
C4	F2	D2-1	1	2	2
C4	F2	D2-2	2	4	8
				Total Performance Risk:	22

考察

- いわゆるイマドキのウエブアプリ(C2)が一番セキュリティ的には良い.
  - コレは直感に合致する.
- パフォーマンス的にはスマホアプリ的なもの(C4)が良い.
  - コレも直感に合致する.
- 個人情報アセット(D1-1)が極めて大きくセキュリティに効いてくる.
- パフォーマンスも同様だが，バックエンドのDB vs FS の影響がセキュリティより大きい.

本手法のまとめ

- アーキテクチャ記述モデルに基づき，品質要求の気づきを与える手法を提案した.
- その評価を行い，良好な結果を得た.

- 適用事例を増やしたい.
- セキュリティとパフォーマンス以外も扱いたい.
- 機能仕様からアセットを見つけ出す支援.
- アセットとアーキテクチャの対応付けの支援.

アーキテクチャの何が重要か？

- アーキテクチャ決定（選択）によって，機能以外の部分がどう変わってくるかを，吟味する必要がある．
  - セキュリティ，パフォーマンス，ユーザビリティ等
- 実際に動かす場合，この機能以外の部分は，非常に大きく効いてくる．

- 残念ながら，ICONIXでは，あまりこの辺を組織的には扱っていない．
- よって，独自に他の手法を利用する必要がある．
  - 紹介したRiMALLではなくても，他の方法でもよい．

本日は以上