ソフトウェア工学　136－28
（２００２．３．８）

アーキテクチャ指向設計手法 ACE の紹介

西岡 竜大 †　野々垣 直浩 †　今井 健男 †
加藤 泰志 †　玉木 裕二 †　深谷 哲司 †
† 株式会社 東芝 研究開発センター システム技術ラボラトリー
E-mail: {tatsuhiro.nishioka, nobuhiro.nonogaki, tateo.imai, yasushi.kato, yuuji.tamaki, tetsuji.fukaya}@toshiba.co.jp

概要

仕様変更や機能追加が頻繁に繰り返されるソフトウェアは,「開発成果物の一貫性欠如」や「トレーサビリティーの低下」などの問題を起こしやすい．このため，変更を重ねるごとに変更量が増加していく傾向にある．我々は，これらの問題を解決するために，**アーキテクチャ指向設計手法 (ACE)** を導入する．**ACE** では，変更量を増加させるリスクを早期に分析し，リスクを回避するための対策を求める．そして，このリスクと対策を “ポリシー” としてソフトウェアアーキテクチャを中心に，ソフトウェアライフサイクル全体へ反映させる．ソフトウェア開発に **ACE** を適用することで，保守フェーズでの変更量を含むトータルコストの抑制が可能になる．本稿では **ACE** の概要と，ポリシーを反映したアーキテクチャ構築プロセスを説明し，**ACE** を適用した結果得られる効果について検証する．

Architecture Centric Engineering

Tatsuhiro Nishioka †, Nobuhiro Nonogaki †, Takeo Imai †,
Yasushi Kato †, Yuji Tamaki †, Tetsuji Fukaya †
†System Engineering Laboratory, Corporate Research and Development Center, Toshiba Corporation

Abstract

Many software systems is suffering from the problems such as ”inconsistency among the development artifacts” and ”less treaceability in the development processes” caused by frequent specification changes and functionality additions. The cyclic ad-hoc modifications applied to those systems, hence, likely to increase the maintenance cost. In this paper, we describe the concept of “Architecture Centric Engineering(ACE)” and its architecture design process as a solution for those problems. ACE allows the developers to alalyze the risks in the software lifecycle and consider the solutions. Developers will deploy such risks and solutions, what we call “policy”, into the development artifacts through software architecture. Applying ACE to the development process, we can reduce total development cost, especially in maintenance phase. We also evaluate the effects obtained.

1　はじめに

近年のソフトウェアは，ますます大規模，複雑化してきており，仕様変更，機能追加の頻度やバリエーションも多くなってきた．一方で，携帯電話等のシリーズ製品のソフトウェアは，その開発サイクルが短くなってきているため，生産性の向上や品質保証コストの削減が求められる．

ソフトウェアの生産性の向上・品質の確保のためには，如何に良い設計を行えるかがポイントの一つとなる．しかしながら，実際のソフトウェア開発現場では，納期などの制約からアドホックな変更が積み重ねられ，メンテナンスがやりにくくなっている場合が多い．

我々は，これまでに幾つかのソフトウェア開発に対して支援活動を行って来た．これらのソフトウェアは，シリーズ製品やインデント製品など，用途や将来の展開も様々であったが共通して，**「繰り返し不具合修正や機能追加を行って行くうちに，同じボリュームのソースコードに対する変更量が大きくなっていく」**という傾向がが見られた．特にシリーズ製品などでは，この傾向が顕著であった．ソフトウェアの静的構造解析や開発プロセスの分析，不具合情報の分析などにより，これらの開発成果物を調査したところ，ソフトウェアのライフサイクルの中で，機能追加や品質要求の変更と, それに伴う不具合修正が繰り返し行われていた．これらの変更や修正の可能性がリスクとして早期に分析され,「この部分が変更されてもよいように，このような作りにしておく」という考えが成果物に反映されていれば，最小限で済んだはずである．

つまり，早期にリスクを分析し，リスクを回避するための対策を設計根拠として開発成果物に反映していれば，変更量を抑制できた可能性が高いと考えられる．

我々は，変更量の抑制には，リスクとその対策を具体的な設計根拠として表現することが重要であると位置付け，これら設計根拠の集合を「ポリシー」と定義する．そして，ポリシーがどれくらい反映されているかという観点から開発成果物を再分析した結果，以下の 2 つの問題が重要であると結論付けた．

(1) 開発成果物の一貫性欠如

開発成果物の一貫性欠如とは，ソフトウェアライフサイクル内の全てのプロセスが，共通のポリシーに基づいて行われないため，成果物である設計ドキュメント及びソースコードが，各開発者の独自な考えに基づいたよせ集め

になっていることである．納期に追われてその場しのぎで行った変更の繰り返しも，一貫性の欠如を助長する要因となる．

一貫性の欠如した成果物は，冗長なソースコードに繋がる．これを変更し続けて行くことで不具合を散在させてしまい，結果として変更量の増大に繋がる．

(2) トレーサビリティの低い開発成果物

トレーサビリティの低い開発成果物には,「何を作るか」という情報は記述されていても,「なぜ，このような作りにしたか」というポリシーが明確に示されていない．

このため，仕様変更などが起こった場合，開発者は自分の考えや納期などの制約のみを考慮して成果物を変更してしまう．これを繰り返すことで，成果物の保守性が低下し，変更量は増大してしまう．

これらの問題を解決するために，我々は，トップダウンにポリシーを具体化し，ソフトウェアの開発成果物に反映する設計手法である**アーキテクチャ指向設計手法 (ACE)** を提唱する．**ACE** は以下のプロセスから構成される．

- ポリシーを具体化し，アーキテクチャを構築するプロセス
- アーキテクチャから実装までを行うプロセス

本稿では **ACE** の概要及び，手法の前段であるアーキテクチャ構築プロセスについて述べる．また，**ACE** によりこれら 2 つの問題が解決されたかを検証する．

2　ACE の概要

ここ数年，ソフトウェア開発において，アーキテクチャの重要性が提唱され [1, 2]，開発プロセスの中心的な役割を担ったり実際のソフトウェア開発に適用されてきた [3, 4, 5].

本稿では，アーキテクチャを，ソフトウェアシステムのマクロレベルの構造と動作原理を規定するものと定義する．アーキテクチャの役割は，開発メンバー全員がソフトウェアの全体的な捉え方や考え方を共有する基盤になることである．アーキテクチャで規定された構造や動作原理は，以降の開発プロセスにおいて成果物を導出する際の制約として働くので，開発プロセスに一貫性を持たせることができる．このため，**ACE** ではアーキテクチャを開発プロセスの中心として据えている．

これまでにも，機能要求に基づいてアーキテクチャを構築する手法や [6, 7]，非機能要求を用いたアーキテクチャの構築手法や分析手法が提唱されてきたが [8, 9, 10, 11]. トップダウンでポリシーを具体的に反映し，開発プロセスの一貫性を保つためのアーキテクチャ構築手法については言及されていない．

アーキテクチャを構築する作業は，非常に抽象的で複雑な知的作業を伴うため，実際の開発現場で適用することは納期などの制約から困難な場合がある．この困難さを克服するためには，ポリシーを抽出する具体的な方法と，ポリシーをアーキテクチャへ反映させる方法を確立する事ができれば，アーキテクチャを構築する作業が従来の方法に比べて比較的容易になるというのが我々の考えである．

そこで，以下の技術を **ACE** の中心的な技術に位置付けている．

- ポリシーを抽出する具体的方法
- アーキテクチャへ反映しやすいポリシーの表現方法

また，ポリシーの具体化と，アーキテクチャに基づいた開発方法により，開発成果物は以下の効果が得られるため，ソフトウェアライフサイクル全体における変更量の削減が期待できる．

- ポリシーによりソフトウェアライフサイクル全体で成果物の一貫性が保てる
- ポリシーを用いてアーキテクチャを構築する過程が示されるので，トレーサビリティが向上する

次章以降では，**ACE** の前段である，アーキテクチャ構築プロセスについて説明し，会議室予約システムのアーキテクチャ構築例を示す．

3　アーキテクチャ構築プロセス

アーキテクチャ構築プロセスを図 1 に示す．このプロセスの入力は VoC と VoM であり，出力はアーキテクチャである．

VoC(Voice of Customer) は，顧客の要求を記述したものである．また，VoM(Voice of Manager) は，経営者からプロジェクトマネージャに至るまでの管理者が提示した，製品の展望や開発環境，開発体制，利用する要素技術などである．

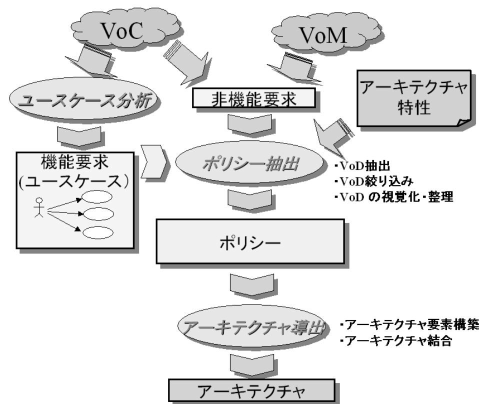

図 1: アーキテクチャ設計プロセス

アーキテクチャ構築プロセスは，以下に示すサブプロセスから構成される．

- ユースケース分析プロセス
- ポリシー抽出プロセス
- アーキテクチャ導出プロセス

ユースケース分析プロセスの役割は，VoC から「何を作るか」という機能要求を明確にする事である．

ポリシー抽出プロセスでは，以下の役割を担う．

- リスクを早期に分析し，対策を導く
- 重要なリスクに着目できるようにする
- ポリシーを，アーキテクチャに反映しやすい表現にする

アーキテクチャ導出プロセスは，ポリシーを，構造と動作原理で表現する作業である．このプロセスでは，VoD 毎にアーキテクチャの一部分 (アーキテクチャ要素) を構築し，それらを結合したものをソフトウェア全体のアーキテクチャとして表現する．

ACE では，アーキテクチャ構築プロセスを複数回行い，アーキテクチャを徐々に具体化する．以下に，**ACE** におけるアーキテクチャ構築プロセスの詳細について説明する．

3.1 ユースケース分析プロセス

ユースケース分析プロセスでは，VoC を入力として，ユースケース記述を出力する．VoC は機能要求や非機能要求を含んだ，あいまいな要求であるため，この VoC から機能要求のみを抽出する．

ACE のユースケース分析では，各サイクルの抽象度に応じたユースケースのみを記述する．このプロセスで重要な事は以下の通りである．

- 各ユースケースの抽象度を統一する
- アーキテクチャ構築プロセスを実装プロセスと切り分けるために，必要以上にブレークダウンしない

3.2 ポリシー抽出プロセス

ポリシー抽出プロセスは図 3 に示すように，VoM,VoC, ユースケース記述を入力として，ポリシーを出力する．ここで，ポリシー抽出プロセスで用いる用語を明らかにする．

[**アーキテクチャ特性**]　我々はユースケースからリスクを分析するために，**アーキテクチャ特性**を導入する．アーキテクチャ特性とは，図 2 に示すように，ソフトウェアの品質特性 [12] の内部特性を変更し，ソフトウェア開発プロセス上のリスクを洗い出しやすいように，35 の質問形式で表現したものである．アーキテクチャ特性は品質特性に当たる大項目，副特性に相当する中項目がある．また，内部項目に相当する小項目は，関連の深い中項目に分類している．中項目及び小項目は，ソフトウェアライフサイクル全体におけるリスクをできるだけ多く洗い出せるように削除／追加／変更している．このように本来ソフトウェアを評価するために利用している品質特性を，リスクを分析するための質問群として利用することで，最初からリスクを回避するための対策を考慮し，ポリシーとしてアーキテクチャに反映させることができるようになる．

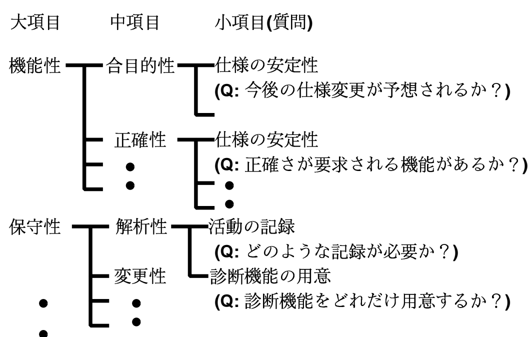

図 2: アーキテクチャ特性の質問 (抜粋)

[**VoD**]　**ACE** では，リスクを回避するために，アーキテクチャを構築する上での対策を求める．このリスクと対策の組を **VoD(Voice of Developer) と呼ぶ**．VoD は「あるリスクが起こってもよいように，この部分を独立にする」の様に表現される．

[**VoD Network**]　**ACE** では，VoD を整理して，ポリシーを抽出するために，VoD Network を導入する．VoD Network は，VoD 間の関係を明確に表現し，矛盾を解消することを目的とする．以下ではポリシー抽出プロセス内の,「VoD の抽出」,「VoD の絞り込み」,「VoD 間の関係の整理」という 3 つの手順について詳細に説明する．

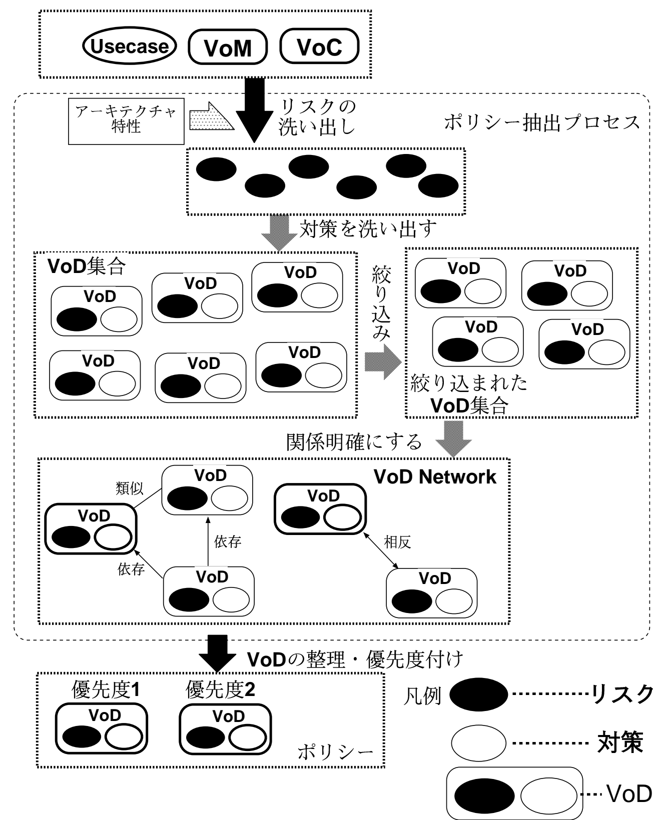

図 3: ポリシー抽出プロセス

3.2.1 VoD の抽出

VoD の抽出は，図 3 の，リスク洗い出しから絞り込みまでの手順であり，ユースケース記述を入力とし，リスクと対策の組である VoD の集合を出力する．VoD の抽出における目的は，機能要求からリスクを洗い出し，リスクを回避する対策を記述することである．以下に VoD の抽出手順を示す．

step1 リスクの洗い出し　ユースケース分析で導出された各ユースケースに対して，アーキテクチャ特性の各質問に答えることでできる限り多くのリスクを洗い出す．例えば，移植性に関するリスクの具体例は,「他のプラットフォームに移植するかもしれない」などである．

step2 対策の導出　得られた全てのリスクに対して，以下の項目を記述することで対策を導き出す．

- 可能性…どんな場合にリスクが発生するか
- 許容度…どこまで考慮すべきか
- 対策…アーキテクチャ設計上の対策

このようにして得られたリスクと対策を組とする VoD を作成する．リスクやリスクに対する各項目を記述するときには，VoC を達成目標とし，VoM を制約として扱う．

なお，対策の記述は，構造の分割か，動作原理 (振舞いや仕組み) を示すようにする．

3.2.2 VoD の絞り込み

VoD の絞り込みでは，抽出された VoD 集合を入力とし，現サイクルで必要な VoD 集合を出力する．入力となる VoD 集合は 様々な抽象度のものが含まれているが，アーキテクチャ構築は繰り返し行われるため，現サイクルがフォーカスしている範囲の VoD に絞り込む必要がある．VoD の絞込み手順を以下に示す．

step1 VoD が現サイクルに必要かを判断する
トップレベルでの判断基準は以下の通り

- 全体的な構造の分割についての対策か
- 全体的な動作原理についての対策か

step2 不要とされた VoD は次サイクルで用いるためにマークする．

3.3 VoD 間の関係の整理

ここでは，絞り込まれた VoD を入力とし，ポリシーを得る．ここでの目的は，VoD Network を用いて VoD 間の関係を明確にすることと，VoD を整理することで，優先度付けされた VoD 集合, つまりポリシーを抽出することである．以下に各手順の詳細を説明する．

step1 VoD Network 構築 まずは，VoD Network を用いて VoD 間の関係を明確にする．VoD Network は VoD をノードとして,「類似」･「相反」等の関係を辺として表現したネットワークである．VoD Network を構築するためには，全ての VoD を以下の関係で結び付ける．

- 類似関係…実線
- 依存関係…単方向の矢印を持つ実践
- 相反関係…双方向矢印を持つ実線
- 互いに共存可能な関係…何も結ばない

尚，ここでいう依存関係とは,「VoD1 は，VoD2 の対策が行われることを前提としている」という意味であり，VoD1 から VoD2 への矢印で表現する．VoD1 がポリシーとして選択されると，暗黙のうちに VoD2 もポリシーとして選択されたことになる．しかし，この逆は成り立たない．

step2 VoD の整理･優先度付け この作業では，VoD の集合から，重要なものを選択してポリシーを出力する．このためには，VoD Network 上の VoD を整理してまとめればよいのだが，設計プロセスの一貫性を保つためには,「どの VoD がより重要か」を示す必要がある．つまり，VoD を優先度付けしなければならない．従って，この作業は以下の手順で行う．

1. **類似する VoD を 1 つにまとめる**
2. **相反する VoD から最も重要な 1 つに絞る**
3. **これらの VoD に優先度を付ける**

VoD の優先度付けや取捨選択の手法としては AHP[14] などの手法を利用すると良い．

3.4 アーキテクチャ導出プロセス

アーキテクチャ導出プロセスでは，ポリシーを入力とし，アーキテクチャを出力する．以下では，アーキテクチャ導出プロセスの各手順について述べる．

3.4.1 step1 アーキテクチャ要素構築

ポリシー内の各 VoD の対策を，アーキテクチャ要素という形で表現する．もし対策の内容を表現できるアーキテクチャパターンがあれば利用する．アーキテクチャ要素の記述は UML[13] のクラス図を用いた概念クラス図である．

3.4.2 step2 アーキテクチャの結合

ポリシーから構築されたアーキテクチャ要素を結合し，ソフトウェア全体のアーキテクチャを構築する．結合する際には，VoD の優先度順に，対応するアーキテクチャ要素を重ねていく．各要素の重ね合わせ時に考慮すべき項目を以下に示す．

todo1 役割の統一

- クラス名などの名前を統一する
- クラスの役割を統一する

todo2 抽象度の整合

- 抽象度が高いクラスは具体的なクラスと継承関係を結ぶ

todo3 振舞い (動作原理) の確認

- 各ユースケースを実行する際のシナリオを決定し，実行可能かを判断する．

ここで重要なのは，VoD の優先度順にアーキテクチャ要素を組み合わせて行く過程をドキュメントとして明記しておくことである．これによりポリシーのトレーサビリティが確保される．

4 アーキテクチャ構築例

本章では，**ACE** の アーキテクチャ構築プロセスを用いて，ある部署の会議室予約システムのトップレベルアーキテクチャを構築する例を説明する．プロセスの入力として，以下に示す VoC が与えられた．

- ユーザは端末にログインして会議室の予約・キャンセル・閲覧等を行う
- 日付，時間，会議室など，適切に入力して予約・キャンセル・閲覧する
- システムはユーザ情報を参照して予約，キャンセルを行う
- 画面は直感的に判りやすいものにする

また VoM として，動作環境，開発環境，開発期間，要素技術，将来展望，移植性などについて制約が与えられた．これらの中には，

- 将来展望:他部門にも横展開したい
- 開発コスト: 2 人月
- 移植性:Unix/Windows で利用可能にする

などの，アーキテクチャに関連しそうな項目も含まれていた．以下では，これらの入力に対して行ったトップレベルアーキテクチャの構築過程を示す．

4.1 ユースケース分析

VoC から，アクタであるユーザに対して，システムが果たすべきサービスという観点でユースケースの候補を列挙すると，以下が得られた．

候補 1 ユーザを認証する

候補 2 予約を閲覧する

候補 3 会議室を予約する

候補 4 予約をキャンセルする

候補 5 日付，時間，会議室などを入力する

候補 6 ユーザ情報を参照する

これらの中でユーザと直接結ばれるトップレベルのユースケースは，候補 1-候補 4 の 4 つである．但し，候補 1 に関しては，他の 3 つのユースケースを実行するために必要なユースケースであるため，除外する．その結果として得られたユースケース UC1-3 を図 4 に示す．

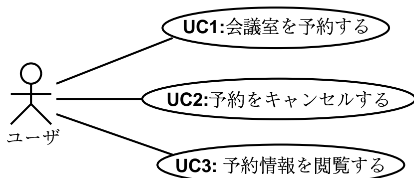

図 4: ユースケース図

4.2 ポリシー抽出

4.2.1 VoD 抽出

会議室予約システムの各ユースケースに対して，VoD 抽出を適用した．

step1: リスク洗い出し 各ユースケースに対して，アーキテクチャ特性の質問に答えることで，リスクを求める．UC1 に対して，図 5 に示すアーキテクチャ特性の質問に対する答えとして,「予約方法が変更されるかもしれない」と「予約画面の仕様が変更されるかもしれない」というリスクが得られた．同様に，各ユースケースに対して，全てのアーキテクチャ特性の質問に答えたところ，以下に示す個数のリスクが得られた．

- UC1 会議室を予約する … 11 個
- UC2 予約をキャンセルする … 11 個
- UC3 予約を閲覧する … 13 個

つまり，システム全体として考えられるリスク総数は 35 個である．

step2,3: 対策の導出，リスクと対策の組を VoD とする 各リスクに対して，アーキテクチャを構築する上での対策を求めたところ，37 個得られた．例えば，UC1 から得られたリスク例に対して，可能性，許容度，対策を洗い出したところ，以下が得られた．

- Risk 「予約方法が変更されるかもしれない」
  - 可能性…複数の日付を指定して予約したいかもしれない
  - 許容度…予約だけでなく，他のユースケースに対しても考慮する
  - 対策…予約方法が変更されてもいいように「予約依頼」という共通の枠組を作る
- Risk 「予約画面の仕様は変更されるかもしれない」
  - 可能性…他の部署に展開するときは，画面仕様が変更される．
  - 許容度…社内展開のみを考慮する
  - 対策…UI と予約情報処理を分離する

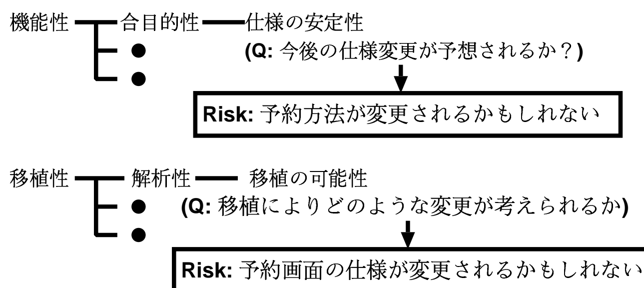

図 5: アーキテクチャ特性の質問と UC1 に対して得られたリスク

この例における VoD は,「予約方法が変更されてもよいように，予約依頼という共通の枠組を作る」「予約の画面仕様が変更されてもよいように，UI と予約情報部分を分離する」という表現になる．なお，リスクより対策の方が多いのは，リスクを回避するための対策が複数あったためである．

4.2.2 VoD 絞り込み

抽出された 37 個の VoD には「データの安全性を確保するために，DBMS のバックアップを定期的に行うようにする」などといった，トップレベルアーキテクチャには具体的すぎるものも含まれていた．これらの VoD から、トップレベルのアーキテクチャを導くために必要なもののみに絞り込んだ結果，以下に示す 8 つの VoD が得られた．なお，各 VoD は，括弧内のユースケースから導出されたものである．

VoD1: 予約の画面仕様が変更されてもいいように，UI と予約情報処理部分を分離する (UC1)

VoD2: プラットフォームが変わってもいいように，UI を変更可能にしておく (UC2))

VoD3: 様々な画面から予約できるように，UI と予約情報は独立にしておく (UC3)

VoD4: 予約方法が変更されてもいいように「予約依頼」という共通の枠組を作る (UC1)

VoD5: 予約取消方法が変更されてもいいように，予約を共通に処理できる仕組みを作る (UC2)

VoD6: 閲覧方法が変更されてもいいように,「閲覧依頼」という共通の枠組を作る．また依頼と予約情報は分離しておく (UC3)

VoD7: 複数の処理要求に対応できるように，予約情報の整合性を確保する仕組みを作る (UC1)

VoD8: 複数の処理要求に対応できるように，予約情報を多重化しておく (UC3)

4.2.3 VoD 間の関係の整理

step1: VoD Network 構築 会議室予約システムの VoD Network(図 6) では，VoD1-VoD3 及び VoD4-VoD6 がそれぞれ類似関係で結ばれ，2 つの VoD 集合となった．また，VoD7，VoD8 は相反する関係で結ばれた．

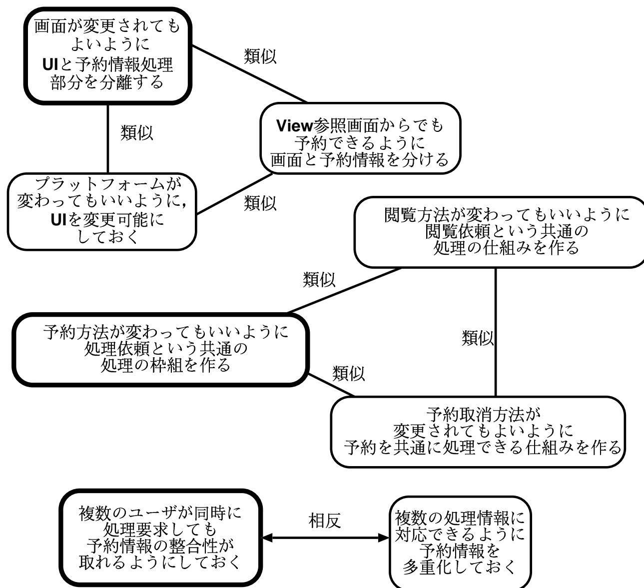

図 6: VoD Network

step2: VoD の整理，優先度付け

1. 類似する VoD を 1 つにまとめる
   VoD1-3 及び VoD4-6 の 2 つの集合は，それぞれ VoD1, VoD4 を基にして，1 つずつの VoD にまとめられた．VoD1-3 に対しては，全てのリスクは,「画面仕様が変更される」とまとめられた．対策部分は同じ意味のため,「UI を予約画面と分離する」とまとめた．これにより，3 つの VoD は「画面仕様が変更されてもいいように，UI と予約情報部分を分離する」となる．また，VoD4-VoD6 では，予約／キャンセル／閲覧方法が変更になるかもしれないというリスクは,「処理要求が増えるかもしれない」とまとめられた．また，対策は「依頼という共通の枠組を作り，依頼が予約情報を操作する」となった．
2. 相反する VoD から最も重要な 1 つに絞る
   3 つ以上の相反する VoD に対しては，AHP を用いるとよいが，ここでは 2 つだけなので，単純にどちらかを選択する．ここでは，VoM の開発コストが 2 人月であることから，開発コストの低い VoD7 を選択した．
3. VoD に優先度を付ける
   この段階で，VoD は 3 つに絞り込まれた．この 3 つの VoD に対して AHP を用いて，VoD4，VoD1，VoD7 という順で優先度付けされた．

この結果，会議室予約システムのポリシーは

PL1 (VoD4) 処理要求が増えてもいいように「依頼」という共通の処理の枠組を作り，依頼が予約情報を操作するようにする．

PL2 (VoD1) UI が変更されてもいいように，UI と予約情報処理部分を分離する

PL3 (VoD7) 複数の処理要求に対応できるように，予約情報の整合性を確保する仕組みを作る

となった．優先度もこの順序である．

4.3 アーキテクチャ導出プロセス

4.3.1 アーキテクチャ要素の構築

図 7 に，会議室予約システムにおけるアーキテクチャ要素 (構造部分のみ) を示す．**PL1** から概念クラスの候補を探すと,「要求」,「依頼」,「予約情報」が得られる．これがアーキテクチャ要素に含まれる概念クラスになる．これらの概念クラス間の関係を考えると,「要求」が「依頼」という共通の処理の枠組を通して，予約情報を操作する．となる．これを UML のクラス図を用いて記述すると 7a) になる．

これらの概念クラスの役割は，それぞれ,「ユーザの要求を受け，依頼する」,「依頼を受け，予約情報を変更する」,「予約に関する情報を保持する」というものである．要求と依頼との関係は,「要求」が「依頼」に「予約情報」の操作をお願いすることである．また，依頼と予約情報に関する関係は

ユーザが予約やキャンセル，参照という要求を行う際に，全ての要求を「依頼」という抽象的な概念で扱えるようにするという構造を持ったアーキテクチャ要素である．

このようにすることで,「予約依頼」,「キャンセル依頼」などと全ての依頼は，継承クラスとして表現できるようになるため，ユーザからの依頼の種類が増えても，新たな依頼のサブクラスとして追加するだけで良い．従って構造や動作原理を大きく変更する必要は無い．

また， 図 7b) は，**PL2** を満たすために，画面・予約情報という概念クラスに分離させていることを示す． 図 7c) は，**PL3** というポリシーから，予約情報に対して画面が複数存在するというアーキテクチャ要素を構築した．

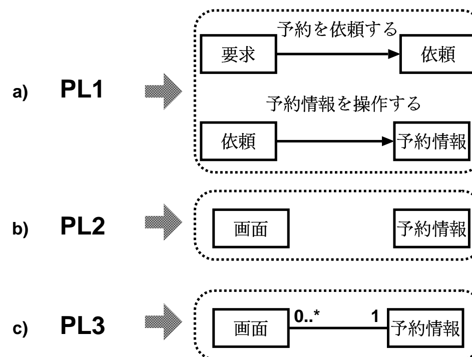

図 7: 会議室予約システムにおけるアーキテクチャ要素の構築

4.3.2 アーキテクチャ結合

会議室予約システムにおけるアーキテクチャ結合の過程を (図 8) に示す．

1. 図 7a) の要素を置く (図 8 step1)
2. 図 7b) の要素を重ねる (図 8 step2)
   - 役割の統一 … 画面と要求は同じであるので「画面」とする．画面の役割は以下の通り

– ユーザからの要求を受け付ける
– 予約情報に処理を依頼する
– 処理結果を受け取り表示する

3. 図 7c) のアーキテクチャ要素を重ねる

- 画面と予約情報の関係を加える (図 8 step3)

このようにして, トップレベルのアーキテクチャが構築された. 図 8 の step3 が会議室予約システムにおけるトップダウンアーキテクチャである.

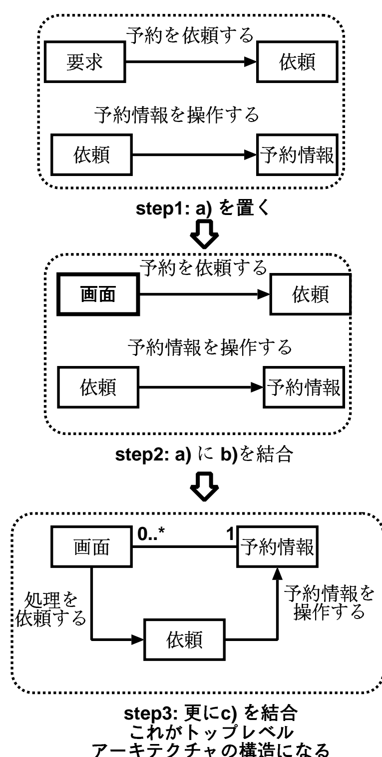

図 8: アーキテクチャ要素結合過程

5 考察

ACE に対する考察として，問題解決の検証及び，今後の方向性について述べる．

5.1 問題解決の検証

成果物の一貫性確保 VoD Network で，VoD 間の矛盾の排除及び類似，依存関係の整理を行うため，無矛盾なポリシーを導出できる．無矛盾なポリシーに基づいて構築されたアーキテクチャ要素を優先度順に結合することで導出されたアーキテクチャもまた，機能要求を満たす構造と動作原理の無矛盾性が保証される．

また，アーキテクチャは，アーキテクチャ構築プロセスの繰り返しによりブレークダウンされる．このとき，アーキテクチャの概念クラス単位でブレークダウンされるため，前サイクルのアーキテクチャを壊すことはできない．従って アーキテクチャ構築プロセスによって導出されたアーキテクチャには，一貫性が保たれる．

トレーサビリティの向上 ユースケース記述からアーキテクチャへの導出過程を表すドキュメントを以下に示す.

- **ユースケースからリスクへの過程**… アーキテクチャ特性の質問と答え
- **VoD からポリシー抽出への過程**… VoD Network を用いた VoD 間の関係
- **リスクからアーキテクチャ構築への過程**… ポリシー

アーキテクチャ特性の質問と答えは，どのユースケースからリスクが抽出されたかということと，どの質問に対するリスクかを示す．また，VoD Network は，どのリスクが重視され，選択されたかを示す．また，ポリシーはリスクを回避するために，アーキテクチャ要素をどのように構築すればよいかを示しているため，アーキテクチャ構築の過程を示すことになる．これらのドキュメントを用いれば，ユースケースからアーキテクチャまでの導出過程をトレース可能であり，アーキテクチャから，それを導き出したユースケースへと遡ることもできる．従来の開発手法では，機能要求からアーキテクチャを導出した根拠を示すドキュメントはコメントなどだけであり，トレースすることは考慮されていない．従って，**ACE** は，アーキテクチャ構築において，トレーサビリティを向上させている．

5.2 研究の優位性及び方向性

ここでは，**ACE** の優位性及び今後の研究課題を，関連研究と比較することで述べる．**ACE** の新規性は，アーキテクチャ特性と，アーキテクチャに反映しやすいように考慮されたポリシーである．

アーキテクチャ特性は，リスクを早期に分析し，アーキテクチャ構築上の対策を表現するために考案されたのだが，従来のソフトウェア設計手法において，ソフトウェア品質特性をリスク分析のために使用した例はあっても，アーキテクチャ構築にリスクを回避する対策を反映させるという観点で具体的に使用された例はない．早期にリスクを分析し，ある程度の変更を吸収できるアーキテクチャを構築することで，結果として変更量を抑制することができる．

ACE では，アーキテクチャを考慮する際に VoD 単位でアーキテクチャ要素を構築し，それらを結合するアプローチを取っている．このアプローチだと，問題を細分化できるため，全体を一度に解決しようとする従来の設計手法よりも解を導きやすくできるし，トレーサビリティの向上も容易になる．

また，従来設計では，ポリシーに相当する事が列挙されていることはあっても，それらの無矛盾性は保証されていない場合が多い．**ACE** では，VoD Network を用いて早期にポリシー内の矛盾を排除しており，アーキテクチャ構築上での後戻りを無くすことで，開発成果物の一貫性確保に大きく貢献している．

ACE を製品のアーキテクチャ構築に適用した際に得られた意見の中には，アーキテクチャ特性の質問により，あいまいな機能要求の具体化や新規抽出にも役立つというものが含まれていた．機能要求が明らかになることで，開発成果物の一貫性確保やトレーサビリティ向上の他にも，後のプロセスで発生する設計の後戻りを防ぐ可能性を示すと考えられる．

今後の方向性としては，ポリシーからアーキテクチャパターンへ変換する具体的な手法，既存の設計に対する評

表 1: **ACE** 関連研究との比較

項目	**ACE**	ABAS[8]	NFR[9, 11]	AOA[10]
非機能要求の利用方法	ポリシー抽出	アーキテクチャパターンの特徴付け	アーキテクチャパターンと開発目標の関連付け	共通部分と差異の分析
アーキテクチャ要素導出手法	ポリシーの具体化･反映	パターンを選択	パターンを選択	固定部分と変動部分から構築
トレーサビリティ	ポリシー，VoD Network など	パターンの特徴で示す	パターンと目標との関連で示す	分析過程で示す
プロダクトラインへの適用	○	記載なし	記載なし	◎
アーキテクチャの評価	×	記載なし	記載なし	◎

価手段，またプロダクトラインへの積極的な適用手段などを導入することを考えている．

アーキテクチャパターンへの変換に関しては，非機能要求を用いてアーキテクチャパターンを特徴付けるアプローチ [8] や，非機能要求の観点からソフトウェアシステムの目標を表現し，アーキテクチャパターン選択の基準を与えるアプローチ [9, 11] などが興味深い．

既存の設計に対する評価手段は，非機能要求からアーキテクチャを評価する Bosch らのアプローチ [4] が参考になる．また，プロダクトラインへの積極的な適用手段としては，AOA[10] などのアプローチも有用である．

また，現在の手法に対しても，リスクに対する対策が一意に求まるわけではないため，対策を導き出す基準や候補リストなどを用意したり，過去の事例などを参照できるような仕組みを導入していきたい．

6 まとめ

ソフトウェア開発現場から得られた共通の傾向,「変更量の増大」を設計成果物における一貫性欠如とトレーサビリティーの低さが問題であると位置付け，ポリシーを設計プロセスに反映させる手法として，アーキテクチャ指向設計手法 **ACE** の概念と，その前段であるアーキテクチャ構築プロセスを説明した．

ACE を用いることで, 設計者のポリシーをアーキテクチャに反映できるだけではなく,どのようにポリシーをアーキテクチャへ反映したかというトレーサビリティを示すことができる. これら事を可能にするために，技術要素としてアーキテクチャ特性や VoD Network を導入した．

現在，アーキテクチャから実装までのプロセスへの適用／評価に関しては現在進行中であり，今後設計成果物の一貫性などを検証して行く．更に，2 つの問題点を解決することで，開発成果物の変更量にどれくらい影響を与えるかの検証も行っていく必要がある．

この他にも，**ACE** の今後の研究課題として，以下の項目が挙げられる．

- **ACE** 手法のツール化で設計コスト削減
- アーキテクチャパターンの積極適利用を促進
- 既存の設計成果物に対するアーキテクチャ評価・改善手法の導入

また，これらの課題に取り組む過程で，**ACE** をより多くの開発プロセスへ適用し，アーキテクチャ特性へドメイン特化の質問を充足させたり，リスクから対策を導出するコストの省力化などを行いたい．

参考文献

[1] M.Shaw, D.Garlan, “Software Architecture”, Prentice Hall, 1996

[2] Software Engineering Standards Comittee of the IEEE Computer Society, “IEEE Recommended Practice for Architectural Description of Software-Intensive Systems”, IEEE Standard, 2000.

[3] L.Bass,P.Clements and Rick Kazman, “Software Architecture in Practice”, Addison Wesley, 1998

[4] J.Bosch, “Design & Use of Software Architectures”, Addison Wesley, 2000

[5] M.Jazayeri, A.Ran and F.Linden, “Software Architecture for Product Families”, Addison Wesley, 2000

[6] I. Jacobson, G.Booch and J. Rumbough, “The Unified Software Development Process”, Addison Wesley, 1999

[7] D.F. D Souza and A.C.Wills, “Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach”, Addison Wesley, 1999

[8] M.Klein, R.Kazman, “Attribute-Based Architectual Styles”, CMU/SEI-99-TR-022, 1999

[9] D.Gross and E.Yu., “From Non-Functional Requirements to Design through Pattern”, Proceedings of the 6th International Workshop on Requirements Engineering: Foundations for Software Quality, 2000

[10] 岸 知二,, “ソフトウェアアーキテクチャのためのアスペクト指向分析”, 情報処理学会ソフトウェア工学研究会, 131-8, 2001

[11] N. Subramanian and L. Chung, “Software Architecture Adaptability: An NFR Approach” Proc., Int. Workshop on Principles of Software Evolution (IWPSE‘01), IEEE Computer Society Press, 2001

[12] A. Abran, et al, “Software engineering-Product quality - Part 3:Internal metrics”, ISO draft for Review, 2000.

[13] G. ブーチ, オージス総研オブジェクト技術ソリューション事業部 訳, “UML ユーザーズガイド”, ピアソン・エデュケーション, 1999.

[14] Saaty, T.L., “The Analytic Hierarchy Process”, McCraw-Hill, 1980