Vol.2019-SE-203 No.3
2019/12/12

アーキテクチャリファクタリングのための代数的アーキテクチャモデル

太田 陽一朗[1,a)]　新田 直也[1,b)]

概要：ソフトウェアアーキテクチャを構成する上で, データの転送を PUSH 型で行うか PULL 型で行うかは非常に重要な要素である. 本研究では, データの転送方式を PUSH 型と PULL 型の間で双方向に変更可能なアーキテクチャレベルのリファクタリングを提案し, その支援のために, 外部から観測可能なデータの振る舞いやデータ間の依存関係を形式的に記述できるアーキテクチャモデルを導入する. 本アーキテクチャモデルは代数的な記述によって, システムの外的な振る舞いがリファクタリングの前後で変わらないことを保証できるようにすると同時に, 設計者が選んだデータ転送方式に基づいて Java プログラムのプロトタイプを自動生成可能にすることも目指す.

1. はじめに

ソフトウェアアーキテクチャを構成する上で, データの転送を PUSH 型で行うか PULL 型で行うかは非常に重要な要素である. **PUSH 型のデータ転送**とは, あるソフトウェアコンポーネントから別のソフトウェアコンポーネントに制御が移るときに, 制御の移動と同じ向きにデータを転送する方法であり, **PULL 型のデータ転送**とは制御の移動と逆向きにデータを転送する方法である. 例えばコンポーネント間でのメソッド呼び出しを考えたとき, 引数を使ったデータ転送が PUSH 型に, 戻り値を使ったデータ転送が PULL 型に相当する. より高い抽象度では, Web アプリケーションにおいてサーバからクライアントへのデータ転送を PUSH 型で行うか PULL 型で行うか, MVC アーキテクチャにおいてモデル更新時の情報をビューに PUSH 型で転送するか PULL 型で転送するかなど, データ転送方式の選択はアーキテクチャ設計において大きな部分を占める.

そこで, 本研究ではデータの転送方式を PUSH 型と PULL 型の間で双方向に変更可能なアーキテクチャレベルのリファクタリングを提案し, そのようなリファクタリングを支援するための枠組みの構築を目指す. 一般にリファクタリングとはソフトウェアの外的な振る舞いを変えることなく, 内部構造を改善する枠組みのことであり [1], ここではデータの転送方式が, 改善すべき内部構造に相当する. データの転送方式の変更によって, データの使用量や実行効率などの非機能的要件の改善を図ることができる. 一方, データの転送方式を変更しても変わらない振る舞いに相当するのが, データ間の依存関係である. ただし, システムが持つデータには利用者から観測可能なものと観測不可能なものがある. したがって, リファクタリングの前後で変わらないことを保証しなければならないのは, 外部から観測可能なデータ間の依存関係や個々のデータの振る舞いということになる. そこで本稿では, システム全体のデータの振る舞いや依存関係を形式的に定義できるアーキテクチャモデル (以下本アーキテクチャモデルと略) を導入する. 多くの形式的なアーキテクチャモデル [2], [3], [4] がプロセスを中心として構成されているのに対し, 本アーキテクチャモデルはデータを中心として構成されている点が特徴である. 本アーキテクチャモデルには, さらに, 設計者がリファクタリング時にデータ転送方式を任意に変更できるよう, データ転送方式を示す情報を付加できるようにする. 一般にデータ転送方式の変更はソフトウェア全体の構造に大きな影響を与えるため, 与えられたアーキテクチャモデルと付加されたデータ転送方式の情報から, Java プログラムのプロトタイプを自動生成する手法についても検討する. 生成するプロトタイプとしては, テストケースを実行することによって機能的要件を満たしているか否かを検査できるようなレベルものを想定している.

2. 例題リファクタリング

本節では仮想的な POS レジシステムの Java プログラムを対象に, 外部から観測可能なデータの振る舞いやデータ間の依存関係を不変に保ったまま, データの転送形式のみ

[1] 甲南大学大学院 自然科学研究科
Graduate School of Natural Science, Konan University
[a)] m21924003@center.konan-u.ac.jp
[b)] n-nitta@konan-u.ac.jp

を PUSH 型から PULL 型に変更するリファクタリングを行う例を示す. このリファクタリングによって, プログラム全体の構造がどのように変化するかを見ていくと同時に, PUSH 型と PULL 型の間でデータ転送方式の変更が可能である場合とそうでない場合があることも見ていく. 本節で題材とする POS レジシステムでは, 簡単のため単独の顧客のデータのみを扱うものとする. このシステムでは顧客が購入を行うと購入額が決まり, 購入額に応じてポイントが計算される. また, 過去からのすべての購入履歴が購入額のリストとして記録され, 購入履歴から購入総額が求められる. なお, ポイントは購入額の 5%の小数点以下を切り捨てた値とする. ここで観測可能なデータを, 購入額, ポイント, 購入履歴, 購入総額とする. これらの間のデータの流れを図 1 に示す. まずはこのシステムを Java で実装することを考える. すべてのデータは, `CashRegister` クラスのインスタンスが直接的/間接的に保持しているものとする. 顧客が購入するたびに, `CashRegister#purchase()` が呼び出され, このとき引数に購入額が渡される. ポイント, 購入履歴, 購入総額はそれぞれ, `CashRegister#getPoints()`, `CashRegister#getPurchaseHistory()`, `CashRegister#getTotalayment()` の呼び出しを通じて, システム外部から常時観測されるものとする.

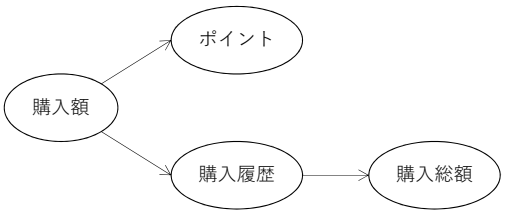

図 1 POS レジシステム内のデータの流れ

このシステムを PUSH 型優先で実装した例を図 2 に示す. ポイント, 購入履歴, 購入総額のデータはそれぞれ, `Points`, `PurchaseHistory`, `TotalPayment` クラスのインスタンス内部に保持される. `CashRegister#purchase()` の呼び出しで引数として渡された購入額のデータは, `Points#update()` および `PurchaseHistory#update()` の呼び出しによって, `Points`, `PurchaseHistory` クラスのインスタンスに PUSH 型で転送され, このデータを元にポイントと購入履歴のデータが更新される. 購入履歴のデータ更新時には, さらに `TotalPayment#update()` が呼び出され, 購入総額のデータも更新される.

今度はこのシステムを PULL 型優先で実装することを考えよう. 実装例は図 3 の通りである. PULL 型の場合, `CashRegister#purchase()` の呼び出しで引数として渡された購入額のデータは, 必要とされるまで他のオブジェクトには転送されず, そのまま `Payment` クラスのインスタンス内部に保持される. 一方, 購入額から算出することができるポイントのデータは, `Points` クラスのインスタンス内部には保持されず. `Points#getPoints()` が呼び出された時点で, `Payment` クラスのインスタンスから取得した購入額のデータを元に算出され, 戻り値として返される. ここで注意が必要なのは, すべてのデータを PULL 型で転送できるとは限らない点である. 例えば, 購入履歴のデータを更新する方法について考える. この場合, ポイントの算出の場合と違って, 最新の購入額のデータをどこかに保持しておいて, そのデータから必要に応じて購入履歴のデータを求めるということができない. 最新の購入履歴を求めるためには, 最新の購入額だけではなく直前の購入履歴のデータも必要となるためである. したがって, 図 3 の実装例においても, 購入履歴のデータは `PurchaseHistory` クラスのインスタンス内部に保持し, 購入額のデータを PUSH 型で転送することによってその内容を更新している.

POS レジシステムの PUSH 型優先の実装例と PULL 型優先の実装例を比較すると, 以下のようなことがわかる.

- **プログラム全体の構造に与える影響:** PUSH 型優先の実装例と PULL 型優先の実装例を比較してみるとわかるように, データの転送方式の変更がプログラム全体の構造に大きな影響を及ぼしていることがわかる. したがって, 既存のソースコードを対象に, データ転送方式を後から変更することは容易ではないことが予想される. そこで本稿では, リファクタリングをアーキテクチャモデル上でのみ行い, ソースコードについては, プロトタイプレベルのものをアーキテクチャモデルから自動生成することを考える.
- **データ保持の必要性とリソース:** 外部から観測可能なデータが, 実装レベルでもデータとして保持されているとは限らない. たとえばポイントのデータは, PUSH 型優先の実装例では `Points` クラスのインスタンス内部に保持されているが, PULL 型優先の実装例ではどこにも保持されていない. 一方いずれの実装例においても, 購入履歴のデータは保持しておく必要があった. そこで以降では, 外部から観測可能な‘見かけ上の’データを**リソース**と呼んで, 実装レベルにおけるデータ保持の有無と関係なく扱うことができるようにする. 各リソースが実装レベルでデータを保持する必要があるか否かについては, 関連するリソースとの間の依存関係の内容によって決まる. たとえば, 購入履歴のデータを実装レベルで保持しておく必要があるのは, 最新の購入履歴を求めるために, 直前の購入履歴と最新の購入額の両方のデータが必要となるためである. そこで, 次節ではデータ保持の必要性を判定するため, リソースの状態間の依存関係を代数的に定義できるようなアーキテクチャモデルを考える.
- **データ転送方式の PULL 型への変更可能性:** PUSH 型優先と PULL 型優先のいずれの実装例においても,

```
public class CashRegister {
   private Points points = new Points();
   private TotalPayment total = new TotalPayment();
   private PurchaseHistory history
                            = new PurchaseHistory(total);
   public void purchase(int pay) {
      points.update(pay);
      history.update(pay);
   }
   public int getPoints() {
      return points.getPoints();
   }
   public int getTotalPayment() {
      return total.getTotalPayment();
   }
   public ArrayList<Integer> getPurchaseHistory() {
      return history.getPurchaseHistory();
   }
}
public class Points {
   private int points = 0;
   public void update(int payment) {
      points = (int)((double) payment * 0.05);
   }
   public int getPoints() {
      return points;
   }
}
public class PurchaseHistory {
   private ArrayList<Integer> history
                               = new ArrayList<Integer>();
   private TotalPayment total = null;
   public PurchaseHistory(TotalPayment total) {
      this.total = total;
   }
   public void update(int payment) {
      history.add(payment);
      total.update(history);
   }
   public ArrayList<Integer> getPurchaseHistory() {
      return history;
   }
}
public class TotalPayment {
   private int totalPayment = 0;
   public void update(ArrayList<Integer> history) {
      totalPayment = 0;
      for (int payment: history)
         totalPayment += payment;
   }
   public int getTotalPayment() {
      return totalPayment;
   }
}
```

図 2　PUSH 型優先アーキテクチャによる実装

```
public class CashRegister {
   private PurchaseHistory history = new PurchaseHistory();
   private Payment payment = new Payment(history);
   private TotalPayment total = new TotalPayment(history);
   private Points points = new Points(payment);
   public void purchase(int pay) {
      payment.setPayment(pay);
   }
   public int getPoints() {
      return points.getPoints();
   }
   public int getTotalPayment() {
      return total.getTotalPayment();
   }
   public ArrayList<Integer> getPurchaseHistory() {
      return history.getPurchaseHistory();
   }
}
public class Payment {
   private int payment = 0;
   private PurchaseHistory history = null;
   public Payment(PurchaseHistory history) {
      this.history = history;
   }
   public int getPayment() {
      return payment;
   }
   public void setPayment(int payment) {
      this.payment = payment;
      history.update(payment);
   }
}
public class Points {
   private Payment payment = null;
   public Points(Payment payment) {
      this.payment = payment;
   }
   public int getPoints() {
      return (int)((double) payment.getPayment() * 0.05);
   }
}
public class PurchaseHistory {
   private ArrayList<Integer> history
                                = new ArrayList<Integer>();
   public void update(int payment) {
      history.add(payment);
   }
   public ArrayList<Integer> getPurchaseHistory() {
      return history;
   }
}
public class TotalPayment {
   private PurchaseHistory history = null;
   public TotalPayment(PurchaseHistory history) {
      this.history = history;
   }
   public int getTotalPayment() {
      int totalPayment = 0;
      for (int payment: history.getPurchaseHistory())
         totalPayment += payment;
      return totalPayment;
   }
}
```

図 3　PULL 型優先アーキテクチャによる実装

購入履歴のデータを求める際には PUSH 型のデータ転送を用いる必要があった．2 つのリソース間のデータ転送で PULL 型の転送方式が許されるか否かは, 転送先のリソースがデータを保持する必要があるか否かに依存する．転送先のリソースがデータを保持する必要があれば, そこで保持されているデータを常に最新に保つため, 転送元のデータが更新されるたびに PUSH 型のデータ転送を行う必要がある．

- **データの転送方式と非機能要件:** リソースはシステム外部から観測可能であるため, 機能的要件と関係する．一方, データ転送方式は非機能的要件と強く関係する．例えば, PUSH 型の転送方式の場合すべての転送先でデータを保持しておく必要があるため, 必要となる記憶領域が増大する傾向がある．更新する頻度より観測する頻度の方が高いリソースについては, PUSH 型で更新する方が無駄なデータ転送が発生せず, 逆に更新する頻度の方が高いリソースについては, PULL 型で更新する方が無駄なデータ転送が発生しない．

3. アーキテクチャモデルの概要

システムの外的な振る舞いがリファクタリングの前後で変わらないことを保証するため, 外部から観測可能なデータの振る舞いやデータ間の依存関係を代数的に記述できるアーキテクチャモデルを提案する．本節ではその概要のみを紹介し, 形式的な定義は次節で与える．前節で説明したように, 本研究では, 実装レベルにおいて保持されるデータと区別するため, 外部から観測可能な見かけ上のデータを**リソース**と呼ぶ．したがって, 本アーキテクチャモデルはリソースを中心として定義される．

アーキテクチャモデルの概要を, 前節で紹介した POS レジシステムの例を用いて説明する．POS レジシステムの例でリソースとして考えられるのは, 購入額, ポイント, 購入履歴, 購入総額である．本節ではこれらをそれぞれ, payment, points, history, toral で表す．本アーキテクチャモデルにおいて, 各リソースは**メッセージ**を受け取って状態遷移を行う状態遷移系としてモデル化する．文献 [5], [6] のアーキテクチャモデルでも, リソースをラベル付き状態遷移系 (LTS) としてモデル化しているが, 状態空間は有限である．一方本研究では, 無限の状態空間を持つことができるように状態遷移系を代数を用いて定義するものとする．複数のリソースの状態遷移を連動させる必要がある場合には, **チャンネル**を利用する．例えば POS レジシステムの例では, payment と points の状態遷移を連動させる必要がある．そのために, ここではチャンネル c_1 を用いる．チャンネルは任意個定義することができる．各チャンネルには複数のリソースが接続することができ, 状態遷移に応じて, チャンネルの入力側に接続したリソースから出力側に接続したリソースに向かってメッセージが送信される．このとき, 入力側リソースの状態遷移, メッセージの送信, 出力側のリソースの状態遷移はすべて同時に行われるものとする．システムと外部の環境とのインタラクションは入出力用のチャンネルを使って行われる．ただし, リソースの内部状態は入出力チャンネルを経由することなく常に外部から観測されているものとする．これらの仕組みによって, システムの外的な振る舞いを定義する．

具体的に POS レジシステムの例を用いて説明する．この例では, チャンネルとして $c_{\mathrm{I/O}}, c_1, c_2, c_3$ の 4 つを用い, $c_{\mathrm{I/O}}$ を入出力チャンネルとする．まず, チャンネルとリソースの対応を表 1 に示す．また, 遷移関数を図 4 に示す．各リソースの遷移関数名にはリソース名をそのまま用いる．関数名の右肩には, リソースが接続しているチャンネル名と入力側への接続か出力側への接続を示す I もしくは O を付記する．遷移関数の第 1 引数を遷移前の状態, 第 2 引数をメッセージとし, 関数値を遷移後の状態とする．一般に同一リソースの遷移関数が複数のチャンネルで重複して定義されるが, これらは同じ遷移の各チャンネル内での見え方の違いを表している．したがって, それらの間で第 1 引数の値および関数値は常に一致する．例えば図 4 の例では, 任意の遷移において $x = y = z$ が成り立つ．

システム全体は, 外部環境から入出力チャンネルへのメッセージの入力をトリガーとして動作する．例えば POS レジシステムのチャンネル $c_{\mathrm{I/O}}$ にメッセージ purchase(100) が入力されると, 最初に $c_{\mathrm{I/O}}$ の出力側の遷移関数 $\mathrm{payment}^{c_{\mathrm{I/O}},\mathrm{O}}$ が評価され, payment の状態が 100 になる．$\mathrm{payment}^{c_{\mathrm{I/O}},\mathrm{O}}$, $\mathrm{payment}^{c_1,\mathrm{I}}$, $\mathrm{payment}^{c_2,\mathrm{I}}$ はすべてリソース payment の遷移関数であるため, payment の遷移後の状態から $x = y = z = 100$ が求まる．チャンネル c_1 および c_2 の入力側リソースの遷移後の状態が決定するため, これにより c_1 および c_2 内にそれぞれメッセージ $\mathrm{update}_1(100)$, $\mathrm{update}_2(100)$ が送信され, 出力側リソース points, history の状態がそれぞれ 5, $\mathrm{cons}(100, \mathrm{nil})$ になる．ただし, リソース history の初期状態は nil とする．同様にして, チャンネル c_3 の入力側リソースの遷移後の状態 $u = \mathrm{cons}(100, \mathrm{nil})$ から出力側リソース total の遷移後の状態が決まり, $\mathrm{sum}(\mathrm{cons}(100, \mathrm{nil})) = 100$ となる．

本モデルでチャンネル毎に状態遷移関数をまとめる理由は, チャンネルに同一のメッセージ列が流れた後の入力側リソースと出力側リソースそれぞれの状態が持っている情報量を比較できるようにするためである．詳細は 5.2 節で述べるが, 出力側リソースの状態の方がより多くの情報を持つ場合, そのリソースは実装レベルにおいてデータを保持する必要がある．例えば, 図 4 のチャンネル c_2 にメッセージ列 $\langle \mathrm{update}_1(x_1), \mathrm{update}_1(x_2), \mathrm{update}_1(x_3) \rangle$ が流れた場合を考えると, payment の状態は x_3, history の状態は $\mathrm{cons}(x_3, \mathrm{cons}(x_2, \mathrm{cons}(x_1, \mathrm{nil})))$ となるため, 明らかに history の方がより多くの情報を保持していることがわか

る. したがって, history にはデータを保持する必要がある.

表 1 POS レジシステムにおけるチャンネルとリソースの対応

チャンネル	入力側リソース	出力側リソース
$c_{\mathrm{I/O}}$		payment
c_1	payment	points
c_2	payment	history
c_3	history	toral

$$\mathrm{payment}^{c_{\mathrm{I/O}},\mathrm{O}}(p_1, \mathrm{purchase}(x)) \quad = \quad x.$$

$$\begin{aligned} \mathrm{payment}^{c_1,\mathrm{I}}(p_1, \mathrm{update}_1(y)) &= y, \\ \mathrm{points}^{c_1,\mathrm{O}}(p_2, \mathrm{update}_1(y)) &= \mathrm{floor}(y \times 0.05). \end{aligned}$$

$$\begin{aligned} \mathrm{payment}^{c_2,\mathrm{I}}(p_1, \mathrm{update}_2(z)) &= z, \\ \mathrm{history}^{c_2,\mathrm{O}}(h, \mathrm{update}_2(z)) &= \mathrm{cons}(z, h). \end{aligned}$$

$$\begin{aligned} \mathrm{history}^{c_3,\mathrm{I}}(h, \mathrm{update}_3(u)) &= u, \\ \mathrm{total}^{c_3,\mathrm{O}}(h, \mathrm{update}_3(u)) &= \mathrm{sum}(u), \end{aligned}$$

where:

$$\mathrm{sum}(\mathrm{nil}) = 0,$$

$$\mathrm{sum}(\mathrm{cons}(v, w)) = v + \mathrm{sum}(w).$$

図 4 POS レジシステムの代数的アーキテクチャモデル

4. 観測可能なデータの振る舞いを表現する代数的アーキテクチャモデル

4.1 基本定義

リソースに求められている外的振る舞いを表す代数的アーキテクチャモデルを $\mathcal{R} = \langle R, C, \rho, T, \tau, \mu, \Sigma, \Delta, \Gamma, s_0 \rangle$ で定義する. ただし,

- R: リソースの有限集合,
- C: チャンネルの有限集合,
- ρ ($\rho : C \times \{\mathrm{I}, \mathrm{O}\} \to 2^R$): 各チャンネルの入力側および出力側に接続しているリソースの集合,
- T: ソートの有限集合,
- τ ($\tau : R \to T$): 各リソースが持つ状態のソート,
- μ ($\mu : C \to T$): 各チャンネルを流れるメッセージのソート,
- Σ: T 上のソート付きシグニチャ,
- $\Delta = \langle \delta_r^{c,d} \rangle_{c \in C, d \in \{\mathrm{I},\mathrm{O}\}, r \in \rho(c,d)}$ (ただし, $\delta_r^{c,d} \in \Sigma_{\tau(r)\,\mu(c),\,\tau(r)}$): 状態遷移関数のシンボルの有限集合,
- Γ: T 上の等式の有限集合,
- s_0: 初期状態 (ただし, $r \in R$ について $s_0(r) \in \Sigma_{\epsilon,\tau(r)}$),

とする. それぞれの詳細については以下で説明する.

4.2 リソースとチャンネル

リソースとはシステム外部から状態を常に観測できる見かけ上のデータである. システムで扱うリソース全体からなる集合を R で表す. 本稿のモデルでは R を有限集合とし, システムの実行中にリソースの生成, 消滅は発生しないものとする. なお, 各リソースは一般に無限の状態空間を持つことができる. チャンネルは複数のリソースの状態遷移を連動させるために用いられる. チャンネル全体からなる集合を C とし, 各チャンネルの入力側か出力側にリソースが接続することができるとする. チャンネル $c \in C$ の入力側に接続しているリソースの集合と出力側に接続しているリソースの集合を, それぞれ $\rho(c, \mathrm{I})$, $\rho(c, \mathrm{O})$ で表す. 入力側のリソースが状態遷移を行うと, チャンネル上にメッセージが送信され, それを受け取った出力側のリソースが状態遷移を行う. なお, C は空でない入出力チャンネルの集合 $C_{\mathrm{I/O}}$ を含むものとする. すべての入出力チャンネル $c_{\mathrm{I/O}} \in C_{\mathrm{I/O}}$ は入力側のリソースを持たない. すなわち, $\rho(c_{\mathrm{I/O}}, \mathrm{I}) = \emptyset$ が成り立つ.

4.3 ソートとシグニチャ

本モデルでは, リソースの状態遷移とチャンネル内のメッセージの流れを代数的に表現するため, **多ソート代数**を用いる. まず, リソースの状態, チャンネルを流れるメッセージは**ソート** (型) を持つものとする. ソートの全体集合を T で表す. リソース r の状態のソートを $\tau(r)$, チャンネル c を流れるメッセージのソートを $\mu(c)$ で表し, $\{\tau(r) \mid r \in R\} \cap \{\mu(c) \mid c \in C\} = \emptyset$ とする.

次に, $\Sigma = \langle \Sigma_{w,t} \rangle_{w \in T^*, t \in T}$ を T 上の**ソート付きシグニチャ**とする. ここで $\Sigma_{w,t}$ は, 引数のソートの列が w, 返す値のソートが t となるような演算シンボルの集合を表す. $\Sigma_{\epsilon,t}$ に含まれる演算シンボルを定数シンボルと呼ぶ. ただし, ϵ を空列とする. 例えば, $0 \in \Sigma_{\epsilon,\mathrm{int}}$, $\mathrm{true} \in \Sigma_{\epsilon,\mathrm{bool}}$, $\mathrm{false} \in \Sigma_{\epsilon,\mathrm{bool}}$ は定数シンボルである. 各チャンネル $c \in C$ に対して, 少なくとも 1 つの演算シンボル $m \in \Sigma_{w,\mu(c)} (w \in T^*)$ が存在し, これを c に定義された **API シンボル**と呼ぶ. 例えば, 図 4 の例では, $\mathrm{purchase} \in \Sigma_{\mathrm{int},\mu(c_{\mathrm{I/O}})}$ はチャンネル $c_{\mathrm{I/O}}$ に, $\mathrm{update}_1 \in \Sigma_{\mathrm{int},\mu(c_1)}$ はチャンネル c_1 に定義された API シンボルである. また各入出力チャンネル $c \in C_{\mathrm{I/O}}$ には, API シンボルとは別に, 操作をしないことを表す単位元シンボル $e_c \in \Sigma_{\epsilon,\mu(c)}$ が定義されているものとする.

Δ は状態遷移関数のシンボルの有限集合であり, 各リソースの各チャンネルへの接続毎にちょうど 1 つのシンボルを含む. ここで, リソース r がチャンネル c の d 側に接続している場合 (すなわち, $r \in \rho(c, d)$ が成り立つとき), 対応する状態遷移関数のシンボルを $\delta_r^{c,d}$ で表す. このとき, $\delta_r^{c,d}$ のソートは, $\delta_r^{c,d} \in \Sigma_{\tau(r)\,\mu(c),\,\tau(r)}$ となる. すなわち, そのリソースの遷移前の状態を第 1 引数, そのチャンネルを流れるメッセージを第 2 引数にとって, 遷移後の状態を返す関数を表している. 例えば図 4 の例では, $\tau(\mathrm{payment}) = \mathrm{int}$ とすると, $\delta_{\mathrm{payment}}^{c_1,\mathrm{I}} = \mathrm{payment}^{c_1,\mathrm{I}} \in \Sigma_{\mathrm{int}\,\mu(c_1),\,\mathrm{int}}$ となる.

なお, シグニチャ Σ に演算シンボルを加えて拡張した $\widehat{\Sigma}$ を考える. ただし $\widehat{\Sigma}$ は, 各チャンネル c の入力側の状

態遷移関数シンボル $\delta_r^{c,\mathrm{I}} \in \Delta$ に対して, 演算シンボル $\overline{\delta_r^{c,\mathrm{I}}} \in \widehat{\Sigma}_{\tau(r)\,\tau(r),\mu(c)}$ を含むものとする.

4.4 等式集合による状態遷移関数の定義

Γ は各状態遷移関数を定義するために用いられる等式の集合で, 適当な**ソート付き変数**の集合 $X = \langle X_t \rangle_{t \in T}$ および X 上の Σ **項**を用いて定義される. ソート t を持つ X 上の Σ 項の集合を $T_\Sigma(X)_t$ で表す. 特に, 変数を含まないソート t の Σ 項の集合を $T_{\Sigma,t}$, さらにソートを限定しないものを T_Σ と書く. $T_\Sigma(X)_t$ の厳密な定義は紙面の都合上割愛するが, 図 4 の例では, $\{x, y, z, p_1, p_2\} = X_{\mathrm{int}}$ となり, $\{x, p_1, \mathrm{payment}^{c_{\mathrm{I/O}},\mathrm{O}}(p_1, \mathrm{purchase}(x))\} \subset T_\Sigma(X)_{\mathrm{int}}$ となる. Γ に含まれるすべての等式 $l = r$ について, l および r は同じソートの Σ 項であるとする. また, すべての状態遷移関数シンボル $\delta_r^{c,d} \in \Delta$ について, Γ は,

$$\delta_r^{c,d}(t_s, t_m) = t'_s \tag{1}$$

の形の等式を少なくとも 1 つ含むものとする. ただし, $t_s, t'_s \in T_\Sigma(X)_{\tau(r)}$, $t_m \in T_\Sigma(X)_{\mu(c)}$ とし, 特に $d = \mathrm{I}$ のとき, $t_s, t'_s \in X_{\tau(r)}$ とする.

なお, Σ 項を用いて定義された等式の集合 Γ に $\widehat{\Sigma}$ 上の等式を加えて拡張した $\widehat{\Gamma}$ を考える. ただし $\widehat{\Gamma}$ は, 各チャンネル c の入力側状態遷移関数のシンボル $\delta_r^{c,\mathrm{I}} \in \Delta$ に対して,

$$\overline{\delta_r^{c,\mathrm{I}}}(x, \delta_r^{c,\mathrm{I}}(x, y)) = y \tag{2}$$

を含むものとする.

4.5 システム全体の状態遷移

システム全体の状態は, すべてのリソースの状態を合成したものとなる. リソース r の状態がソート $\tau(r)$ の定数シンボルで表すことができることから, 本モデルではシステム全体の状態を, 各リソースからその状態を表す定数シンボルへの写像として定義する. すなわち, s は, 各リソース $r \in R$ について $s(r) \in \Sigma_{\epsilon,\tau(r)}$ を満たす. これは初期状態 s_0 についても同様である.

システム全体は, 外部環境から入出力チャンネルへのイベント入力をトリガーとして動作する. 具体的には, アーキテクチャモデル $\mathcal{R}$ において, システム全体の状態が, 入出力チャンネル $c_{\mathrm{I/O}} \in C_{\mathrm{I/O}}$ に入力されたメッセージ m によって s から s' に遷移するとき,

$$s \overset{\langle m, c_{\mathrm{I/O}} \rangle}{\underset{\mathcal{R}}{\longrightarrow}} s'$$

と書く. このとき, チャンネルへの適当な**メッセージの割り当て** $\pi : C \to T_\Sigma$ が存在し, 以下の条件が成り立つ.

- すべての $c \in C$ について, $\pi(c) \in T_{\Sigma,\mu(c)}$,
- $\pi(c_{\mathrm{I/O}}) = m$,
- $c \in (C_{\mathrm{I/O}} \setminus \{c_{\mathrm{I/O}}\})$ ならば, $\pi(c) = e_c$,
- 各 $r \in R$ について, $c, c' \in C$, $d, d' \in \{\mathrm{I}, \mathrm{O}\}$ が $r \in \rho(c, d)$ および $r \in \rho(c', d')$ を満たすならば,

$$\delta_r^{c,d}(s(r), \pi(c)) = \delta_r^{c',d'}(s(r), \pi(c')) = s'(r). \tag{3}$$

これは, 入出力チャンネルのうち $c_{\mathrm{I/O}}$ しか使わないこと, および同一のリソースはどのチャンネルから見ても同じ遷移を行うことを意味している.

ここで, 遷移前の状態 s から, メッセージの割り当て π および遷移後の状態 s' を求める方法について考える. まず, $d = \mathrm{I}$ のとき, 式 (2), (3) より,

$$\overline{\delta_r^{c,\mathrm{I}}}(s(r), \delta_r^{c,\mathrm{I}}(s(r), \pi(c))) = \overline{\delta_r^{c,\mathrm{I}}}(s(r), s'(r)) = \pi(c) \tag{4}$$

が得られ, 式 (1), (2) より,

$$\overline{\delta_r^{c,\mathrm{I}}}(t_s, \delta_r^{c,\mathrm{I}}(t_s, t_m)) = \overline{\delta_r^{c,\mathrm{I}}}(t_s, t'_s) = t_m \tag{5}$$

が得られる. 式 (5) 中の t_s および t'_s はともに変数であるため, この式を一般性を失うことなく

$$\overline{\delta_r^{c,\mathrm{I}}}(x, y) = t_m \tag{6}$$

とおくと, 式 (4), (6) より,

$$\pi(c) = t_m[s(r)/x, s'(r)/y] \tag{7}$$

が得られる. ただし, $t_m[s(r)/x, s'(r)/y]$ は項 t_m 中に出現する変数 x を $s(r)$ に, y を $s'(r)$ に同時に置き換えて得られる項である. ここで $r \in \rho(c, \mathrm{I})$ が成り立っていることから, 式 (7) は, チャンネル c の入力側リソース r の遷移後の状態 $s'(r)$ から, c へのメッセージの割り当て $\pi(c)$ が求められることを示している. ただし, t_m が x, y 以外の変数を含んでいる可能性もあることに注意が必要である. そのような場合, x, y を置き換えても t_m に変数が残るため, 得られた項のみからでは $\pi(c)$ の値が一意に定まらない. 一方, チャンネル c の入力側リソースが r 以外にも存在している場合, それらのリソースの遷移後の状態からも $\pi(c)$ を示す項を独立に求めることができる. 本稿では $\pi(c)$ を示す項が複数求められた場合, それらの項を**単一化**することによって $\pi(c)$ の値を求めることとする. ここで, 単一化によって得られた項にも変数が残っている場合は, $\pi(c)$ の値は定まらないことになる. 逆に, $\pi(c)$ を示すすべての項と矛盾しないような $\pi(c)$ の値が存在しない場合は単一化に失敗し, システム全体の状態遷移も失敗する.

仮に単一化が成功したとして, それによって他のリソースの遷移後の状態をどのようにして求めていけばよいかを見ていこう. いま, r が c の入力側に, r' が c の出力側に接続し, さらに r' が別のチャンネル c' の入力側に接続していると仮定する. このとき, $s'(r)$ から $\pi(c)$ が求まれば式 (3) を使って $s'(r')$ が, $s'(r')$ が求まれば式 (7) を使って $\pi(c')$ が, といったように, チャンネルへの入出力を経由するこ

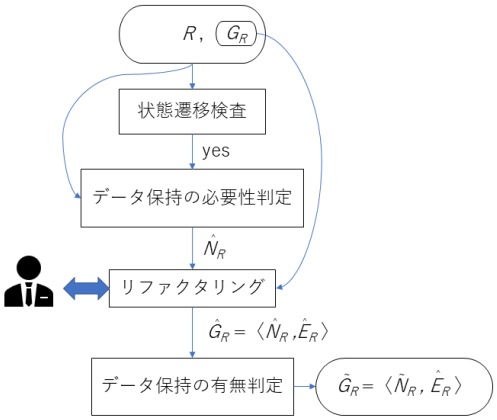

図 5　プロトタイプ生成の流れ

とによって, 入出力チャンネルの出力側リソースから, 連鎖的に各リソースの遷移後の状態を求められることがわかる. ただし, すべてのリソースの遷移後の状態を求めるには, 上記単一化以外にもいくつかの条件を $\mathcal{R}$ が満たしている必要がある. その内容については次節で議論する.

5. アーキテクチャモデルのリファクタリングとプロトタイプの導出

本節では, アーキテクチャモデル $\mathcal{R}$ から生成したデータフローグラフを用いて, Java プログラムのプロトタイプの生成に必要な情報の導出を行う. データフローグラフは $G_{\mathcal{R}} = \langle N_{\mathcal{R}}, E_{\mathcal{R}} \rangle$ で与えられる有向グラフで,

- $N_{\mathcal{R}} = R$,
- $E_{\mathcal{R}} = \{\langle r_1, r_2 \rangle \mid r_1, r_2 \in R$ かつ $\exists_{c \in C}.\{r_1 \in \rho(c, I), r_2 \in \rho(c, O)\}\}$,

を満たすものである. プロトタイプの生成には,
(1) アーキテクチャモデルの状態遷移検査,
(2) リソースのデータ保持の必要性判定,
(3) データ転送方式の選択によるリファクタリング,
(4) リソースのデータ保持の有無決定,
を順番に行う必要がある (図 5 参照). それぞれの手順について以下に説明する.

5.1 アーキテクチャモデルの状態遷移検査

前節で説明したように, 全てのリソースの遷移後の状態が一意に定まるためには, $\mathcal{R}$ がいくつかの条件を満たす必要がある. たとえば, リソース r の遷移後の状態 $s'(r)$ が唯一つの値を持つためには, いずれかの入出力チャンネルの出力側リソースの遷移後の状態から式 (3), (7) を使って連鎖的に遷移後の状態が決定し, それが r まで到達する必要がある. どの入出力チャンネルからも r に到達しない場合, $\mathcal{R}$ の遷移後の状態は不定になる. また, (3) 式において, $d = d' = O$ である場合, $\pi(c)$ と $\pi(c')$ から独立に $s'(r)$ の値が求められる可能性があり, これらの値が競合する可能性がある. このような問題が発生しないことを保証するため, 以下の 3 つの点を確認する.

- 全てのリソースがいずれかの入出力チャンネルの出力側リソースからグラフ $G_{\mathcal{R}}$ 上で到達可能であること,
- $\mathcal{R}$ において異なる複数のチャンネルの出力側に同一のリソースが接続していないこと,
- $G_{\mathcal{R}}$ 中に強連結成分が存在しないこと,

3 つ目は強連結成分が存在することによって前節で述べた単一化が失敗することを防ぐ. 強連結成分とは, 有向グラフにおいて, ノード間での双方向の行き来が可能な部分グラフを指す. 強連結成分は強連結成分分解を行うことで導くことができ, アルゴリズムとしてタージャンのアルゴリズムとコサラジュのアルゴリズムが良く知られている.

5.2 リソースのデータ保持の必要性判定

2 節で述べたように, リソースに求められている外的振る舞いを実現するために, 実装レベルでデータを保持する必要がある否かは, リソース毎に異なる. もし, あるリソースが実装レベルでデータを保持する必要があるなら, そのリソースの状態を最新に保つためのデータ転送は PUSH 型でなければならない. このように実装レベルでのデータ保持の必要性の有無は, 全体の設計や実装方法に大きな影響を与える. そこで, アーキテクチャモデル $\mathcal{R}$ に含まれる各リソースについて, データを保持する必要がある否かを判定する方法について考える.

リソース r におけるデータ保持の必要性は, 3 節で述べたように, 出力側に r が接続しているチャンネル c(すなわち, $r \in \rho(c, \mathrm{O})$ を満たすチャンネル c) の入力側に接続しているリソース群 $\rho(c, \mathrm{I})$ の状態遷移と, r 自身の状態遷移の間の比較によって決まる. すなわち, c を流れる同一のメッセージ列に対して, c の出力側のリソース r と, 入力側のリソース群のいずれの状態がより多くの情報を保持しているかによって決まる. もし, r の状態の方がより多くの情報を保持しているなら r にはデータの保持の必要があり, r の状態の方がより少ない情報を保持しているなら, r の状態が入力側リソース群の状態から常に求められるため, データ保持の必要はない. より形式的には, 任意の遷移において入力側リソース群の状態から出力側リソースの状態への写像が存在する場合かつそのときに限り, 出力側リソースにおいてデータ保持の必要がない. このように, 本アーキテクチャモデルを用いると, チャンネルに関わるリソースの状態遷移関数を代数的に評価するだけで, 状態が保持している情報量の比較とデータ保持の必要性の判定を行うことができる. $N_{\mathcal{R}}$ 中の各ノードにデータ保持の必要性の有無の情報を付加したものを $\hat{N}_{\mathcal{R}}$ とする.

5.3 データ転送方式の選択によるリファクタリング

2 節で述べたように, PUSH 型優先の実装例と PULL 型

優先の実装例ではプログラム全体の構造が大きく異なる. プログラム全体の構造は各リソースの最終的なデータ保持の有無によって変わるが, データ保持の有無は転送方式がPUSH 型か, PULL 型かによって決定される. そこで, $G_{\mathcal{R}}$ の各辺のデータ転送方式を PULL 型にするか PUSH 型にするかを設計者に問い合わせ, 設計者が選んだデータ転送方式を各辺に属性として付与することを考える. データ転送方式が PUSH 型でも PULL 型でもリソースの外的振る舞いは変わらないため, この属性の付与が本アーキテクチャモデルにおけるリファクタリングに相当する. ただし, 2 節で述べたように, 全てのデータ転送方式を PULL 型にも PUSH 型にも変更できるわけではない. データ転送方式の選択によって決まる最終的なデータ保持の有無が, 5.2 節で求めたデータ保持の必要性と矛盾する可能性があるためである. そこで, データ保持の必要性があるリソースに向かう有向辺は設計者に問い合わせることなく PUSH 型の属性を付与し, それ以外の有向辺に対して PULL 型と PUSH 型のいずれの属性を付与するかを設計者に問い合わせる. 属性を付与した有向辺の集合 $\hat{E}_{\mathcal{R}}$ を持つ $\hat{G}_{\mathcal{R}} = \langle \hat{N}_{\mathcal{R}}, \hat{E}_{\mathcal{R}} \rangle$ を **PUSH/PULL 属性付き依存グラフ**とする.

5.4 リソースのデータ保持の有無決定

PUSH/PULL 属性付き依存グラフ $\hat{G}_{\mathcal{R}}$ から各リソースのデータ保持の有無を決定する. まず, **PULL 属性部分グラフ**を PUSH/PULL 属性付き依存グラフ $\hat{G}_{\mathcal{R}}$ から求める. PULL 属性部分グラフとは, $\hat{G}_{\mathcal{R}}$ 中の PULL 属性を持つ有向辺のみからなる部分グラフである. データを保持するリソースは, PULL 属性部分グラフの入次数 0 のノードと, 5.2 節で求めたデータ保持の必要性があるノードである. データ保持の有無の情報を付与したノードの集合を $\tilde{N}_{\mathcal{R}}$ とし, $\tilde{G}_{\mathcal{R}} = \langle \tilde{N}_{\mathcal{R}}, \hat{E}_{\mathcal{R}} \rangle$ を元にプロトタイプを生成する.

6. 事例研究

2 節で紹介した POS レジシステムに, ソースコードの導出に必要なプロセスを手動で適用してみる. 図 4 のアーキテクチャモデルからデータフローグラフを作成すると, 全てのノードが連結しており, 同一のリソースが複数のチャンネルの出力先になっておらず, サイクルも存在しないため状態遷移先は一意に定まる. 前節で示したリソースのデータ保持の必要性判定を行うと, データ保持の必要なリソースは history のみとなる. ここからデータ転送方式の選択可能性判定を行うと, 表 2 のようになる. 表 2 に基づいてリファクタリングを行い, PUSH 型優先と PULL 型優先で属性を指定すると, リソースのデータ保持の必要性は表 3 のように判定される. これは図 2 と図 3 で示したソースコードの設計と一致する.

表 2 POS レジシステムのリソース間のデータ転送方式

入力側リソース	出力側リソース	選択可能な転送方式
payment	points	PULL/PUSH
payment	history	PUSH
history	total	PULL/PUSH

表 3 POS レジシステムのリソースのデータ保持の必要性

リソース	データ保持	PUSH 優先	PULL 優先
payment	不要	無	有
points	不要	有	無
history	要	有	有
total	不要	有	無

7. 考察

前節で示した事例から, 本研究で提案するアーキテクチャモデルは今回の事例においてはリファクタリング及びソースコードの自動生成において, 有効であると考えられる. 今後の課題としては, 5 節で示した条件を自動で判定, ソースコード化するツールの開発, 更に複雑な事例へのアーキテクチャモデルの適用による評価が挙げられる.

8. おわりに

外部から観測可能なデータの振る舞いやデータ間の依存関係を変えることなく, システム内部のデータ転送方式のみを変えるようなアーキテクチャレベルのリファクタリングを提案し, そのようなリファクタリングを支援するための枠組みとして, 代数的アーキテクチャモデルを導入した. また簡単な事例を対象に, 本アーキテクチャモデルのリファクタリングが Java のプログラムのプロトタイプを生成する能力があることを示した. 今後, プロトタイプの生成方法を詳細化してツールとして開発すると同時に, より複雑な事例への適用によって有効性を評価することを検討している.

参考文献

[1] M. Fowler: *Refactoring: improving the design of exsiting code.* Addison Wesley (1999).
[2] R. Allen and D. Garlan: A formal basis for architectural connection, *ACM Trans. Softw. Eng. and Method.*, vol. 6, no. 3, pp. 213–249 (1997).
[3] J. Magee, J. Kramer, and D. Giannakopoulou: Behaviour analysis of software architectures, in *Proc. of Working IEEE/Int'l Federation for Information Processing Conf. Software Architecture* (1999).
[4] P. Pelliccione, P. Inverardi and H. Muccini: Charmy: A framework for designing and verifying architectural specifications, *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 325–346 (2009).
[5] U. Klein and K. S. Namjoshi: Formalization and automated verication of RESTful behavior, in *Proc. of the 23rd international conference on Computer aided verification, CAV' 11*, pp. 541–556 (2011).
[6] N, Nitta: A formal approach for guiding architecture design with data constraints, in *Proc. of IEEE/ACIS ICIS* (2016).