インタラクティブシステムのための共通アーキテクチャに基づくソースコードの自動生成に関する研究
ーデータ構造に着目してー

2013SE121 水田大貴　2013SE127 村瀬賢
指導教員：沢田篤史

1　はじめに

近年，スマートデバイスが多様化しており，それに伴ってインタラクティブシステムが多様化している．このようなインタラクティブシステムの開発環境と実行時環境も多岐にわたる．開発者は，既存の環境ごとで前提とされているフレームワークやプログラミング言語を，それぞれの環境ごとに調査することでコストを要する．また調査時間も要するため，開発時間の増大によって生産性の低下に関わる問題となる．その問題を解決するために，本研究室ではインタラクティブソフトウェアのための共通アーキテクチャが提案されている [3](以下，共通アーキテクチャと呼ぶ)．共通アーキテクチャは MVC アーキテクチャに基づき，アスペクト指向アーキテクチャとして設計されている．共通アーキテクチャにより，既存のフレームワークの理解を支援することが可能となる．共通アーキテクチャの構成要素として，イベント処理系，イベント変換系，ビューモデル変換系が定義されている．イベント変換系には外部イベントを内部イベントに変換する．イベント処理系はイベントに応じた処理を実行するもので，内部に状態遷移機械を保持している．ビューモデル変換系はアプリケーション特有の内部表現形式から外部表現形式へと変換させる．イベント変換系は入力を外部イベントとして内部イベントを出力する．ビューモデル変換系は，内部表現形式を入力として，外部表現形式を出力をする．

本研究では，これらのイベント処理系，イベント変換系，ビューモデル変換系をデータ構造変換系として扱う．開発するアプリケーションによってデータ構造変換系の入出力が異なる．また，開発する実行時環境も異なるので，アプリケーション開発を支援する目的として，アプリケーションのソースコードを自動生成することが挙げられる．

本研究の目的は共通アーキテクチャのコンポーネントのソースコードを自動生成することである．モデル駆動アーキテクチャ (以下，MDA と呼ぶ) [1] に基づいて共通アーキテクチャのソースコードを生成するコード生成系を実現する．プログラミング言語をプラットフォームとして，入力に対して，プラットフォーム情報を付加することでコードを生成する．コードの自動生成を可能にすることで，特定の開発環境や実行時環境に依存しないアプリケーションの開発を容易にできる．

コード生成系の設計は，入力となるデータ構造に着目する．MDA に基づくコード生成系は入力となるデータ構造に対する走査手続きとデータ変換処理をモジュール化したものとして定義できる．共通アーキテクチャの各コンポーネントをデータ構造変換系と捉え，入出力のデータ構造の種類を特定して分類する．それらのデータ構造をそれぞれ走査手続きについて定義する．各コンポーネントごとに変換規則を定義することで，共通アーキテクチャのいくつかのコンポーネントを生成できることを確認する．

結果として，生成系への入力となるデータ構造が同じであれば，同じ走査手続きを用いることが可能であることがわかった．いくつかの生成対象のコンポーネントに応じて変換規則を定義することにより，コンポーネントのソースコードを自動生成できることが確認できた．それにより，コード自動生成系の有用性を確認した．

2　背景技術

本研究と関連する研究や技術を以下に挙げる．

2.1　MDA

MDA とは，分析・設計においてモデルを中心としたソフトウェア開発の概念と標準規格からなる技術体系である．MDA は UML によって表されたソフトウェアの基本的な設計を，様々なプラットフォーム向けに詳細化された設計やソースコードに変換する．

MDA では，図 1 のようなアーキテクチャが提案されている [1]．以下に各構成要素の説明をする．

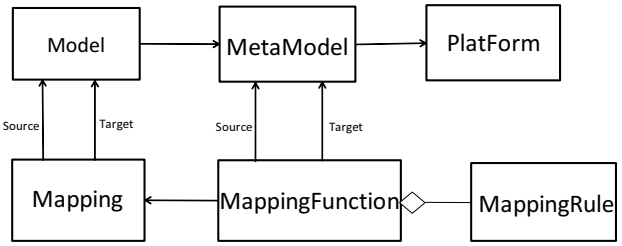


図 1　MDA で提案されているアーキテクチャ

- Model：入出力モデル
- MetaModel：入出力のメタモデル
- PlatForm：プラットフォーム
- Mapping：生成系
- MappingFunction：入力メタモデルに基づく走査手順
- MappingRule：入出力の対応関係

2.2 共通アーキテクチャ

本研究室では，スマートデバイス向けアプリケーションのための共通アーキテクチャが提案されている．共通アーキテクチャとは，任意の開発環境で用い，任意の実行時環境上で稼働可能なアプリケーションの作成支援の枠組みの基礎を与えることを目的としている [3]．共通アーキテクチャは，共通参照アーキテクチャと共通アプリケーションアーキテクチャとして定義している．

2.2.1 共通参照アーキテクチャ

共通参照アーキテクチャの設計は，MVC アーキテクチャとその派生である既存のアーキテクチャで分離を試みている横断的関心事をアスペクト指向アーキテクチャとして定義している．各次元の横断的関心事を指定し，指定した横断的関心事によって規定されるアスペクトを織り込むことで参照アーキテクチャを生成する．

次元 1

- MVC コンサーン
- UI コンサーン

次元 2

- 表示モデルコンサーン
- 表示ロジックコンサーン
- 階層化コンサーン
- 通信コンサーン

図 2 は例としての共通参照アーキテクチャである．以下に

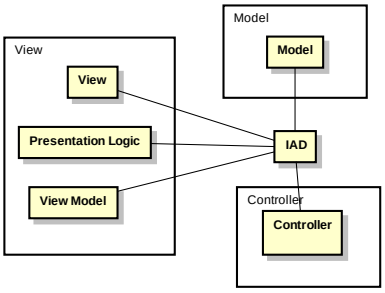


図 2 共通参照アーキテクチャ

コンポーネントの役割を示す．

Controller：ユーザの操作によるイベントを扱う

Model：アプリケーションのモデルを扱う

View：画面に表示される視覚的要素を扱う．

Presenation Logic：View を構築する

ViewModel：画面表示用に加工された Model

2.3 共通アプリケーションアーキテクチャ

共通アーキテクチャを詳細化したものが共通アプリケーションアーキテクチャである．共通アプリケーションアーキテクチャを図 3 に示す

以下にコンポーネントの役割を示す．

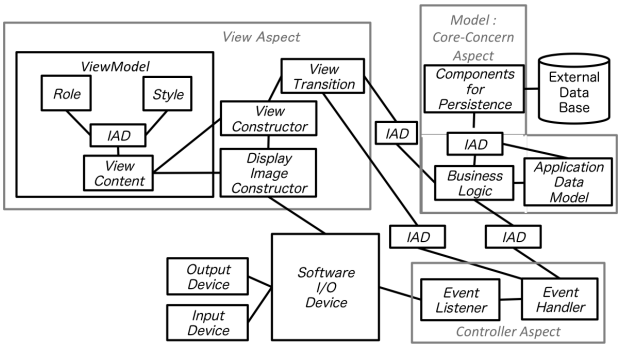


図 3 共通アプリケーションアーキテクチャ

—Controller

EventListener：イベントを内部形に変換し，アプリケーション使用者からイベントを受け取り EvenHandler に通知

EvenHandler：自身の状態と EvenListener からのイベントによって，Model または View にイベントを通知

—Model

Business Logic：Model や View 更新のロジックを状態遷移機会として抽出化したもの

Model：データベース

—View

ViewTransition：画面遷移を抽象化したもの

ViewContent：View のデータ構造を定義

ViewConstructor：Model を参照し，ViewContent のインスタンス化を行なう

DisplayImageContent：ViewConstructor で生成した View の各ノードの出力形式のデータ構造を定義

Style：外部表現（色，サイズ等）を保持

DisplayImageConstructor：ViewConstructor で生成した View に外部表現と出力方式を結合し View を構築

システムの開発において，開発者は Model，View，Controller に関して次の 6 項目を定義する必要がある．アプリケーションのモデル，アプリケーションの状態遷移画面レイアウト（内容，役割，見栄え），画面遷移イベント，コントローラーの状態遷移実装は，この 6 項目を入力として，特定の開発技術に依存した一連の開発ステップ群によって行なわれる．

3 データ構造変換系

データ構造変換系を，出力するデータ構造から独立するために，分類したデータ構造のモデルを中心とする開発を実現するために MDA に基づいて設計する．図 1 にデータ構造変換系を適用するとこのようになる．

Model：データ構造

MetaModel：データ構造の定義

Mapping：データ構造変換の処理

MappingFunction：データ構造の走査手順
MappingRule：データ構造の要素の変換規則
次にイベント処理系，イベント変換系，ビューモデル変換系の振舞いについてまとめて，それぞれについての入出力をデータ構造に着目して分類する．次に分類したデータ構造について走査手順を定義した．

3.1 各コンポーネントの振舞い

イベント処理系は EventHandler，BussinessLogic，ViewTransition，ViewConstructor が挙げられる．イベント処理系は内部に状態遷移機械を保持していて，状態ごとにアクションやアクション列を持つ．
イベント変換系は EventListner が挙げられる．イベント変換系は外部イベントと内部イベントの対応表を保持していて，振舞いとしては，入力を外部イベントとして，それに対応した内部イベント出力して EventHandler に通知する．
ビューモデル変換系は，DisplayImageConstructor が挙げられる．ビューモデル変換系は内部表現形式と外部表現形式の対応表を保持している．振舞いとしては，入力となる内部表現形式に対応した外部表現形式を出力する．

3.2 データ構造に着目した各コンポーネントの振舞い

前述の振舞いをデータ構造に着目して考察した．
イベント処理系の振舞いは内部イベントに対応するアクションを行う．EventHandler を例として，データ構造に着目すると，入力である内部イベントは木構造もしくはリスト構造として表すことができる．出力するデータとしてアクション列やアクションオブジェクトが考えられるので，データ構造としてはリスト構造もしくは木構造として表すことができる．また，内部に保持している状態遷移機械はデータ構造はグラフ構造によって表すことができる．よって，木構造もしくはリスト構造を入力として，木構造もしくはリスト構造を出力するデータ構造変換系と考えられる．
イベント変換系の振舞いは外部イベントに対応する内部イベントを出力する．EventListner を例として，データ構造に着目し，HTTP コマンドを外部イベントとした場合の振舞いの例を挙げる．外部イベントのデータ構造は木構造として表すことができ，内部イベントはイベントオブジェクトや Java のライブラリなどが考えられるのでデータ構造は木構造もしくはリスト構造として表すことができる．よって，木構造を入力として，木構造もしくはリスト構造を出力するデータ構造変換系と考えられる．
ビューモデル変換系の振舞いは入力となる内部表現形式を外部表現形式として出力する．DisplayImageConstructor を例として，データ構造に着目すると，入力となるビューモデルは木構造で表すことができ，出力となる外部表現形式も木構造で表すことができる．よって，木構造を入力として，木構造を出力するデータ構造変換系と考えられる．
以上からデータ構造は，木，リスト，グラフのいずれかであると考えられる．

3.3 データ構造の走査手順定義

前述で検討したとおり，コード生成系の入出力のデータ構造は

- 木
- グラフ
- リスト

の三つに分類される．ここでデータ構造の三つについて走査手順を定義した．
表 1 は分類したデータ構造と定義した走査手順のまとめた表である．
以下に，各データ構造とその走査手順を示す．

表 1

データ構造	走査手順
木	根から行きがけ順に走査
グラフ	根から行きがけ順に走査， 一度通ったノードは走査しない
リスト	根から順に走査

図 4 は木構造の走査手順である．木は根から走査を初め行きがけ順に走査をしていく．

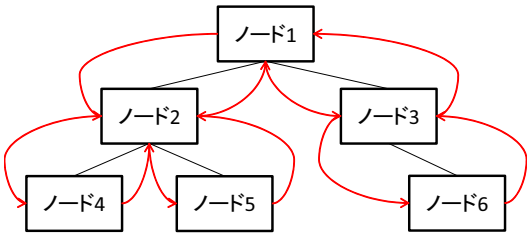


図 4 木構造の走査手順

図 5 はグラフ構造の走査手順である．グラフは根から走査していくが，一度走査したノードは走査しない．

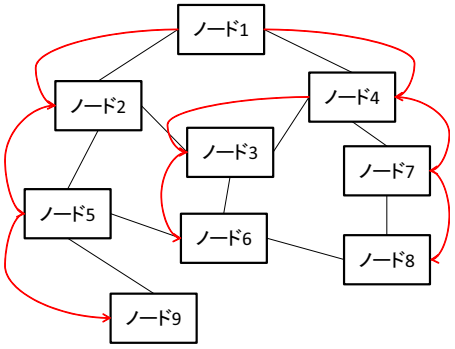


図 5 グラフ構造の走査手順

図 6 はリスト構造の走査手順である．リストは根から順に走査する．

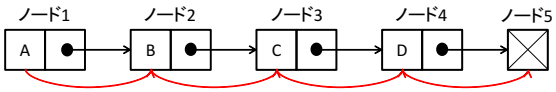


図 6　リスト構造の走査手順

4　考察

我々は共通アプリケーションアーキテクチャの各コンポーネントをデータ構造変換系として考えた．イベント処理系，イベント変換系，ビューモデル変換系の入出力のデータ構造を考察した．また，データ構造の種類を分類したことによって，共通アーキテクチャのソースコードを生成する生成系は，入出力が木，グラフ，リストとなるデータ構造変換系を生成することによって実現できることが考察できた．データ構造変換系の走査手続きは入力となっているデータ構造によって固定される．この走査手続きのソースコードを生成する場合には，そのデータ構造の種類を与えることによって生成できると考察した．

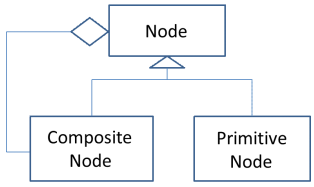
例として，木構造を入力とする EventListner の走査手続きを考える．木構造の走査手順を標準化したプログラムを図 7 に示す．初めに，Node に適応させる MappingRule を Node と MappingRule の対応関係を持つ MappingRuleMap から取得し，Node に適応させる．次に子要素の型が CompositeNode かどうかを審査し，CompositeNode であれば子要素を取得し，繰り返し子要素も走査する．最後に Node に MappingRule を適用させる．次に，データ構造の種類を定義したものを UML でモデル化したものと各要素を変換規則にそって出力するソースコードの関係を図 8 に示す．走査手順と変換規則の流れを図 9 に示す．現状では，データ構造変換系は走査手順は定義できるが変換規則を手書きで書かなくてはならない．今後の課題として，現在 MappingRule が手書きで書かれているので自動生成可能な箇所を特定して，MappingRule のフレームワーク化していくことが挙げられる．

```
apply(Node node){
    MappingRule r =MappingRuleMap.getRule(node);
/ノードに応じたマッピングルールを適用させる
    r.apply(node);   /ノードに応じたデータ変換処理
    for (Node n:node.getTrans()){
        if(!TraverHistory.isContains(n)){/ノードが通ったかどうかを確認
            TraverseHistory.add(n); /ノードが通ったことを登録
            this.apply(n);
        }
    }
```

図 7　木構造の MappingFunction

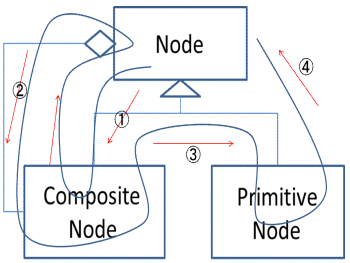
5　おわりに

スマートデバイスが多様化に伴って，開発環境や実行時環境が多岐にわたるようになり，生産性が低下する．この問題に対して，本研究室ではインタラクティブソフトウェアのための共通アーキテクチャが提案されている．本研究では共通アプリケーションアーキテクチャの各コンポーネントを調査し，各コンポーネントのソースコードを生成する生成系は入出力が木，グラフ，リストであるデータ構造変換系を生成することによって実現できると考察した．また，各データ構造について走査手順を定義して，アプリケーションごとに変換規則を定義するとコード生成系が生成できることを考察した．今後の課題としては，実際に生成できることを確認し，また現在 MappingRule が手書きで書かれているので自動生成可能な箇所を特定して，MappingRule をフレームワーク化していくことがが挙げられる．



要素	MappingRule	Javaのコード
Node	NodeMappingRule	MappingRule r= MappingRuleMap.get(node)
CompositeNode	CompositeMappingRule	If(node.instanceofCompositeNode){ for(Node n:node.child) ….
PrimitiveNode	PrimitiveMappingRule	r.apply(node)

図 8　木構造の定義と出力されるソースコードの対応表



	MappingRule	ソースコード
①	NodeMappingRule	Nodeに適応させるMappingRuleを取得して，Nodeに適応，またNodeの型を調べるプログラム
②	CompositeMappingRule	Nodeの子要素を取得する，また，再起呼び出しのプログラム
③	CompositeMappingRule	}
④	NodeMappingRule	NodeにMappingRuleを適応させるプログラム

図 9　木構造の定義を走査する流れ

参考文献

[1] S. J. Mellor, S. Kendall, U. Axel, and W. Dirk, MDA destilled : principles of model-driven architecture. Addison - WesleyProfessional, 2004.

[2] Sokolova, K., Lemercier, M., and Garcia, L. :Towards High Quality Mobile Applications: Android Passive MVC Architecture. International Journal On Advances in Software, Vol. 7, No. 2, pp. 123-138, 2014.

[3] 江坂篤侍，野呂昌満，沢田篤史，” インタラクティブソフトウェアの共通アーキテクチャの提案”．情報処理学会研究報告，ソフトウェア工学報告，vol.2015-SE-187, pp.1-8，2015-03-05.