Vol.2015-SE-187 No.27
2015/3/13

層の活性化処理のモジュール化を実現する文脈指向プログラミング手法の提案

園田直也† 荻原剛志†

概要：文脈指向プログラミング(Context-Oriented Programming : COP)とはプログラムの外部環境や内部状態のような文脈に依存して変化する振る舞いをモジュール化するためのプログラミング技術である．COPでは層の活性化/非活性化により実行するモジュールを切り替える仕組みを提供しているが，従来の手法では層の活性化/非活性化処理が横断的関心事として様々なモジュールに横断して存在していることが問題であった．本研究ではアスペクト指向の技術を利用することにより層の活性化/非活性化処理のモジュール化を実現する手法を提案する．

キーワード：文脈指向プログラミング, アスペクト指向プログラミング, モジュール化, 横断的関心事

Proposition of the Context-Oriented Programming Method to Implement Modularization of Layer Activation Routines

Naoya Sonoda† Takeshi Ogihara†

Abstract: Context-Oriented Programming(COP) is a programming method that modularize behavior depending on context, e.g., internal states or external environment of program. COP provides mechanism that can switch executed modules by activating/deactivating of layers. Existing methods have a problem that activation/deactivation routines should exist in various modules as crosscutting concerns. This paper proposes a new method to realize modularization of layer activation/deactivation routines by taking advantage of the aspect-oriented programming technology.

Keywords: Context-Oriented Programming, Aspect-Oriented Programming, Modularization, Crosscutting Concerns

1. はじめに

近年，システムの外部環境や内部状態などに応じて振る舞いが変化するシステムの設計手法に関心が集まっている．例えば，ATMシステムで現金を引き出す際，利用する時間帯によって手数料を取られることがある．この場合，ATMシステムは時刻という外部環境によって振る舞いが変化するシステムであるといえる.また，モバイル端末ではバッテリの状態に応じて消費電力を減らすモードに移行するものがあるが，この場合にもバッテリなどの内部状態に依存して振る舞いを変化する仕組みが導入されている．

文脈指向プログラミング(COP: Context-Oriented Programming)[1]とは，外部環境や内部状態に依存して振る舞いが変化するシステムの設計を支援するための技術である．COPでは外部環境や内部状態など，振る舞いを変化させる要因となるものを文脈と呼び，文脈によって変化する振る舞いを文脈依存の振る舞いという．文脈依存の振る舞いは層という概念を用いてモジュール化し，モジュール化した層を活性化/非活性化させることにより，文脈に応じて実行する層を切り替える仕組みを実現している．

現在，COPの技術を利用した様々な文脈指向言語が提案されている[2]．文脈指向言語には，言語処理系を拡張して文脈指向の仕組みを導入した言語と，言語処理系を拡張せずに文脈指向の仕組みを導入した言語が存在している．前者の言語の例として ContextJ[3]や EventCJ[4][5]などが，後者の例として Scala言語による文脈指向DSL[6]やContextFramework[7]などが挙げられる．

文脈指向の実現においては，層の活性化/非活性化が重要な役割を担っている．層の活性化/非活性化とは，文脈の状態が変化した時，その文脈の状態において実行される振る舞いをモジュール化した層を有効/無効にすることである．そのため，層の活性化/非活性化処理は文脈の状態に依存して実行される処理であるといえる．

言語処理系を拡張せずにCOPの技術を実現した従来の文脈指向言語では，文脈の状態を取得する必要のある様々なモジュールが，同時に層の活性化/非活性化処理も横断的に実行していた．そのため，文脈依存の振る舞いがどの活性化処理によって，どんなタイミングで切り替えられているのか把握することが容易ではなかった．

本研究ではアスペクト指向の技術を利用することにより，層の活性化処理のモジュール化を実現する手法を提案

† 京都産業大学 先端情報学研究科
Kyoto Sangyo University

する．提案手法を実現することで文脈の状態と層の活性化処理の依存関係を明確にすることができると考えた．本研究ではアスペクト指向の技術を利用するためにSpringにより提供されているアスペクト指向(AOP)のフレームワーク(Spring AOP)を利用する．Spring AOPはJava言語において提供されているフレームワークであるため，Java言語において文脈指向の機構を実装し，Spring AOPを利用して層の活性化処理のモジュール化を実現する．

2. 従来手法

2.1. 文脈指向プログラミング

文脈指向プログラミング(COP)により，文脈に依存する振る舞いを持つシステムを実現するためには２つの仕組みが必要となる．ここでは上で示したATMの例題を使ってそれぞれの仕組みについて述べる．

(1) 文脈依存の振る舞いをモジュール化する

COPでは文脈依存の振る舞いをモジュール化したものを層と呼ぶ．層はContextJ[3]のように言語処理系を拡張することで独自の構文を導入して表現する方法や，内部クラス[6]やメソッド[7]として表現する方法などが存在している．層の中に記述された文脈依存の振る舞いは部分メソッドと呼ばれ，層が有効な状態になっているときに呼び出される．層の外側には必ず部分メソッドと同じシグネチャを持つベースメソッドが宣言されており，有効になっている文脈がない場合に呼び出されるほか，部分メソッドの中から呼び出すことができる．ベースメソッドには文脈依存の振る舞いの中でどの文脈においても共通している振る舞いが定義されている．

図2.1はATMシステムを例題に，層によるモジュール化の仕組みを説明している．ここでは1つのベースメソッドと2つの層が定義されている．ベースメソッドでは現金を引き出す処理の中でどの文脈においても共通する振る舞いを定義している．「層:時間内」は文脈が時間内である場合における現金引き出し処理(手数料を取らない処理)を定義している．「層:時間内」ではベースメソッドと同じシグネチャを持つ部分メソッドが定義されており，この部分メソッドの中に文脈に依存する振る舞いを定義する．同様に「層:時間外」では文脈が時間外である場合の現金引き出し処理(手数料を取る処理)を定義している．

(2) 文脈依存の振る舞いの実行を切り替える

COPでは層の活性化/非活性化により，文脈依存の振る舞いの実行を切り替える．層の活性化とは文脈の状態を取得する処理が実行された際，現在の文脈に該当する層を有効な状態にすることであり，非活性化とは層を無効な状態にすることである．層の活性化方法は文脈指向言語によって異なっており，スタックのような層を管理する配列を用意し，そのスタックに層を追加することで活性化させる方法[6][7]や，現在の文脈の状態を記憶しておき，特定の制御フローに対してのみ層を活性化させる方法[3]が存在している．

文脈依存の振る舞いは層の活性状態に応じて動的に切り替える．活性化している層がなければ層の外側に宣言されているベースメソッドが呼ばれ，複数の層が活性化している場合は最も新しく活性化した層の部分メソッドが呼ばれる．また，複数の活性化している層を順番に実行していくことも可能であり，これをproceedという．

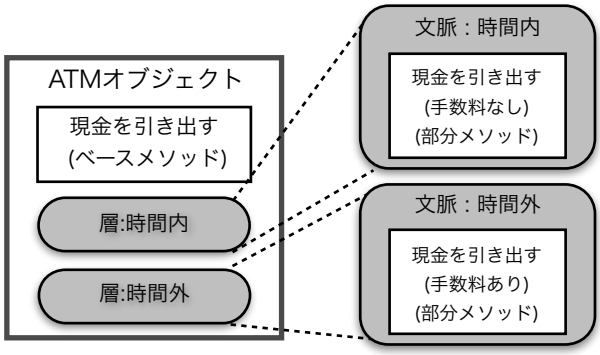

図2.1 層による文脈依存の振る舞いのモジュール化

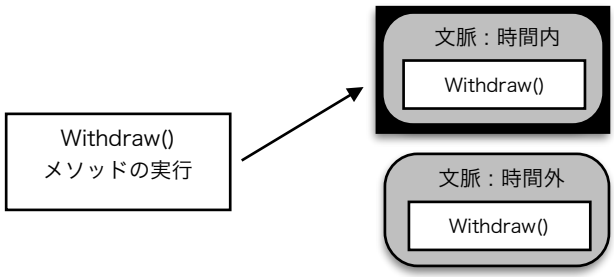

図2.2 「時間内」が活性化した場合の実行

図2.2は，「層:時間内」が活性化している状態において現金を引き出す処理(Withdraw()メソッド)がどの層を実行するかを示している．「層:時間内」が活性化している状態でWithdraw()メソッドが実行されると「層:時間内」に定義した部分メソッドが呼ばれ，「層:時間外」の部分メソッドは呼ばれない．逆に「層:時間外」が活性化している状態では「層:時間外」の部分メソッドが呼ばれる．

2.2. アスペクト指向プログラミング

アスペクト指向は横断的関心事のモジュール化を実現する技術である．横断的関心事に関わる処理をそれぞれのモジュールから切り離して別ファイルに記述しておき，後から別ファイルとモジュールを結びつける(weaveという)ことを可能にする．

横断的関心事を記述する別ファイルはアスペクトといい，アスペクトにはジョインポイントとアドバイス，ポイントカットを設定する．ジョインポイントはアスペクトを挿入する位置を示し，メソッドの実行点やフィールドの値が変更されるタイミングなどを指定できる．アドバイスはジョインポイントに挿入する処理を設定する．ポイントカットはアスペクトを挿入するための条件を示しており，ポイントカットではアスペクトを挿入するジョインポイントとその条件を指定する．

2.2.1. Spring AOP

Spring AOPはSpringにより提供されているアスペクト指向のフレームワークである．多くのアスペクト指向の実装

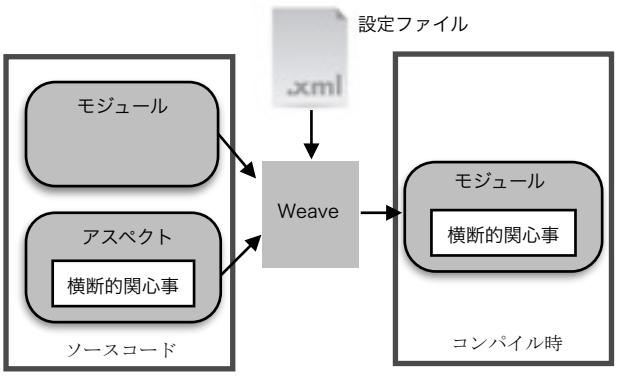

図2.3 Spring AOPによるアスペクト指向の仕組み

では，実行すべきアドバイスの内容やポイントカットをアスペクトに直接指定するが，Spring AOPではソースコードの外部にXML形式の設定ファイルを用意し，アスペクトの設定を行う（図2.3）．

ソースコードの記述では，通常はモジュールで利用される横断的関心事をモジュールから切り離し，アスペクトとして定義する．設定ファイルには，アスペクトを挿入するモジュール(ジョインポイント)と，挿入するアドバイスとその条件(ポイントカット)を設定する．このような構成とすることで，コンパイル時にモジュールにアスペクトの横断的関心事を挿入することが可能となる．

3. Java言語による文脈指向の実現方式

本節では，言語処理系を拡張せずにJava言語において文脈指向の機能を実現し，Springのアスペクト指向のフレームワークを利用して層の活性化処理のモジュール化を実現する手法を提案する．まず，Java言語における文脈指向の機能の実現方式について述べ，次にアスペクト指向を利用して層の活性化処理のモジュール化を実現する手法について述べる．

3.1. Java言語による文脈指向の実現方式

提案手法による文脈指向の機構をクラス図で示したものが図3.1である．それぞれのクラスについて具体的に説明していく．

3.1.1. 文脈依存の振る舞いをモジュール化する

図3.1において，JavaCopクラスが文脈依存の振る舞いを持つクラスである．文脈依存の振る舞いを持つクラスはLayerControllerクラスを継承しなければならない．

提案手法では文脈依存の振る舞いをモジュール化した層を内部クラスとして表現する．図3.1ではContextLayerAとContextLayerBが層に該当する．層となる内部クラスはContextインタフェースを実装しなければならない．Contextインタフェースではベースメソッドを宣言しておき，層となる内部クラスにはベースメソッドと同じシグネチャを持つ部分メソッドを宣言する．

また，ContextインタフェースはLayerインタフェースを継承する必要がある（理由は後述する）．JavaCopクラスをコードに示したものが図3.2である．

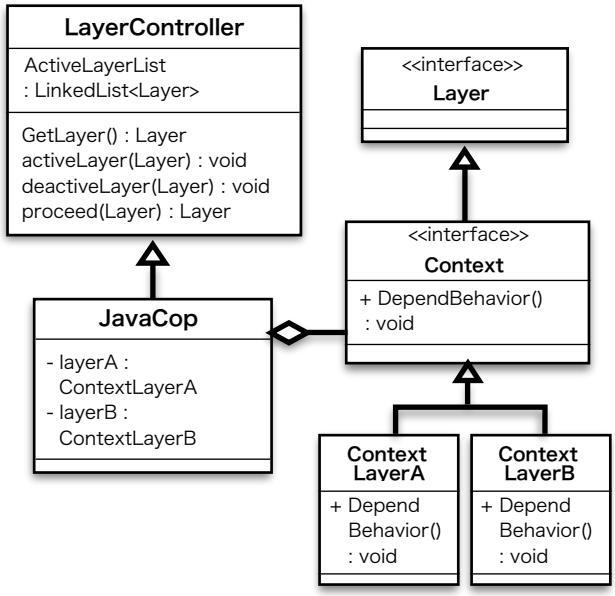

図3.1 提案手法による文脈指向の機構

```
1  public class JavaCop extends LayerController
2  {
3     private static ContextLayerA layerA
4
5     interface Context extends Layer {
6       public default void DependBehavior(){
7                     ......
8       }
9    }
10   public static class ContextLayerA
                      implements Context{
11      public void DependBehavior(){
12                    ......
13       }
14    }
15 }
```

図3.2 JavaCopクラスにおける層の表現

```
1  public class LayerController
2  {
3     static LinkedList<Layer> ActiveLayerList
             = new LinkedList<Layer>();
4
5     public void activeLayer(Layer layer){
6        ActiveLayerList.addFirst(layer);
7     }
8     public void deactiveLayer(Layer layer){
9        ActiveLayerList.remove
                -FirstOccurrence(layer);
10    }
11    ......
12 }
```

図3.3 層の活性化方式

3.1.2. 層の活性化方式

提案手法では，層をスタックに追加/削除することによって層の活性化/非活性化を実現している．スタックはLayerControllerクラスが持つ後入れ先出しの配列(ActiveLayerList)であり，LayerControllerクラスを継承したオブジェクトはこのスタックを利用できる．

層を活性化させるactiveLayerメソッド，層を非活性化させるdeactiveLayerメソッドをLayerControllerクラスに実装しており，LayerControllerクラスを継承したオブジ

ェクトはこれらのメソッドにより層の活性化/非活性化を行う．

コードで示すと図3.3のようになる．配列の型をLayer型とすることで，Layer型を継承したContextインタフェースを実装した層となる内部クラスをスタックに追加/削除できる．しかし，配列から取り出す層の型がLayer型となってしまうため実行時にはキャストが必要となる．

3.1.3. 実行時の層の選択

LayerControllerクラスにスタックの先頭を参照するgetLayer()メソッドを実装している．文脈依存の振る舞いの実行時にgetLayerメソッドを実行することで，スタックの先頭において活性化している層の部分メソッドを実行することができる．

3.1.4. proceedの実装

スタックの先頭の層だけではなく，それ以前に活性化された層に基づいて実行を行いたい場合もある．その場合，proceedという手続きによって，先頭の次以降の層を参照して実行を行うことができる．proceedメソッドは引数に現在活性化している層を与えると，次に活性化している層を返すメソッドである．

3.2. 層の活性化処理のモジュール化の実現方式

3.2.1. 文脈の状態取得と活性化処理の分離

前節ではJava言語を利用して文脈指向の機構を実現する手法について述べた．言語処理系を拡張せずに文脈指向の機構を実装した従来の手法や前節の提案手法のままでは，層の活性化処理が横断的関心事として様々なモジュールに横断して存在することが問題となる．

層の活性化処理では，文脈の状態に応じて活性化する層を選択しなければならない．そのため，層の活性化処理には文脈の状態に関する情報が必要となる．

さらに，層の活性化処理が実行されるタイミングは文脈の状態を取得する処理が実行されるときである．文脈の状態を取得する処理が実行され，文脈の状態が変化していた場合，その文脈に該当する層を活性化させる必要がある．文脈の状態を取得する処理はプログラム内の様々なモジュールにおいて横断的に実行される．層の活性化処理は文脈の状態を取得する処理に依存して実行されるため，層の活性化処理もまた様々なモジュールにおいて実行される．

提案手法による層の活性化処理の実行を示したものが図3.4である．この手法ではソースコードの状態では文脈の状態を取得する処理を持つモジュールから層の活性化処理を切り離し，アスペクトとして別ファイルにモジュール化する．Spring AOPのフレームワークを利用して，層の活性化処理と文脈の状態を取得する処理の依存関係を設定ファイルに記述しておくことで，文脈の状態を取得する処理が実行されるタイミングで層の活性化処理が実行されるようにコードを挿入することが可能となる．

以下では，前節の手法にSpring AOPのフレームワークを組み合わせる手法について述べる．

3.2.2. 設定ファイルによるインスタンスの生成

Spring AOPではAOPの機能を利用するクラスのインスタンスをXML形式の設定ファイルにおいて生成する必要がある．JavaCopクラスのインスタンス生成例を図3.5に示す．設定ファイルにおけるインスタンスの生成には<bean>タグを利用する．id属性でインスタンスのIDを指定し，class属性でインスタンスを生成するクラスを指定する．

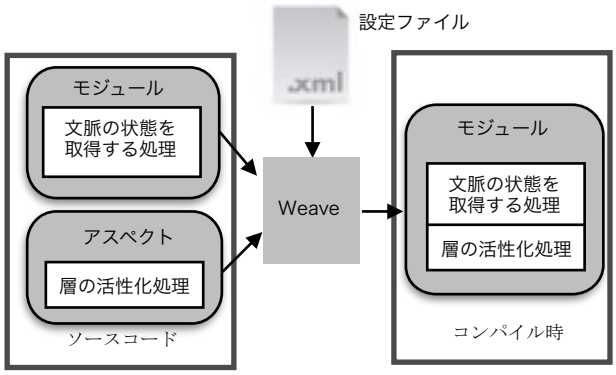

図3.4 提案手法による層の活性化

```
1  <bean id="contextlayerA"
         class="JavaCop$ContextLayerA">
2  </bean>
3  <bean id="javacopbean" class="JavaCop">
4        <property name="setcontextlayera"
           ref="contextlayerA"/>
5  </bean>
```

図3.5 JavaCopクラスのインスタンス生成

提案手法では内部クラスを層として利用する．設定ファイルにおいて内部クラスを持つインスタンスを生成するには，メンバとして内部クラスのインスタンスを設定する．図3.5の4行目の<property>タグに示すように，name属性で内部クラスのセッタメソッドを指定し，ref属性で内部クラスのインスタンスのIDを指定する．

Springにおいて内部クラスを利用するためには内部クラスのアクセッサメソッドを記述し，さらに内部クラスにはstatic修飾子を付けなければならない．

3.2.3. インスタンスの取得

設定ファイルからインスタンスを取得して利用するためにはSpringのフレームワークをインポートする必要がある．様々なモジュールにおいてフレームワークをインポートすることは好ましくないため，提案手法ではインスタンスを取得するUseSpringモジュールを実装した．

UseSpringモジュールではgetBeanメソッドを実装しており，設定ファイルで指定したインスタンスのIDを引数に与えることでインスタンスを取得することができる．

UseSpringモジュールによるインスタンスの取得例が図3.6である．インスタンスを取得する際にはキャストが必要となる．

3.2.4. 層の活性化処理のモジュール化

提案手法では層の活性化処理をモジュール化したクラスとしてAspectクラスを用意した．設定ファイルの記述に従ってAspectクラスのインスタンスを生成することでAspect

```
1  JavaCop JavaCopBean
2  JavaCopBean =
   (JavaCop) UseSpring.getBean(“javacopbean”);
```

図3.6 UseSpringモジュールによるインスタンスの取得

```
1  <aop:config proxy-target-class=“true”>
2    <aop:aspect id=“aspect” ref=“aspectbean”>
3      <aop:pointcut expression=“execution
         (*ContextState.getContext(..))
          id = “aspectlayeractivate”/>
4
5      <aop:after method=“AspectLayerActivate”
6        pointcut-ref=“aspectlayeractivate”/>
     </aop:aspect>
7  </aop:config>
```

図3.7 設定ファイルによるアスペクトの設定

クラスにアスペクト指向の機能を適用する．ここでは，Aspectクラスに層を活性化させるためにAspectLayerActivateメソッドを実装した．設定ファイルにおけるAspectクラスのインスタンスのIDをaspectbeanとした．

3.2.5. 設定ファイルによるアスペクトの設定

図3.7に設定ファイルにおいてアスペクトの設定を行う例を示す．図ではContextStateクラスの文脈の状態を取得するメソッドであるgetContext()メソッドが実行されると，層を活性化するためのAspectLayerActivate()メソッドを実行するように設定している．

アスペクトの設定は2行目の<aop:config>タグを利用する．デフォルトではアスペクト機能の利用はインタフェースにしか利用できないが，<aop:config>タグにおいてproxy-target-class属性の設定をtrueにすることでインタフェース以外のクラスでもアスペクト機能を利用できる．

<aop:aspect>タグのref属性によりアスペクトとなるクラスのインスタンスのID(aspectbean)を指定している．

<aop:pointocut>タグではexpression以下の記述でジョインポイントとなるメソッドを指定し，IDを付ける．

<aop:after>タグでは，上記で指定したメソッドが実行された後にアスペクトに定義したAspectLayerActivateメソッドを実行することを指定している．

4. 既存手法と提案手法の比較実験

既存手法の有効性を示すために，例題を用いて既存手法と提案手法による実装実験を行った．例題は簡略化したATMシステムで，取引を利用する時刻によって手数料に関する振る舞いが変化するものとする．

実験では時刻という文脈を取得するTimerクラスと，層の活性化処理をモジュール化したLayerAspectクラスを実装した．

LayerAspectクラスの実装は既存手法と提案手法で大きな違いはない．既存手法によるコード例を図4.1に示す．Timerクラスから現在時刻を取得し，条件分岐を利用して活性化させる層を選択している．既存手法と提案手法の違いはTimerクラスのインスタンスの取得方法だけであり，提案手法ではUseSpringモジュールを利用して設定ファイルからインスタンスを取得している．

```
1  public class LayerAspect
2  {
3     public void TimeLayerActive(){
4        private Float currentTime;
5        Timer timerObj = new Timer();
6        currentTime = timerObj.getTime();
7        if(currentTime=BusinessTime){
8          activeLayer(“時間内”);
9        else{
10         activeLayer(“時間外”);
11       }
12    }
13 }
```

図4.1 層の活性化処理をモジュール化したクラス

```
1  public class Timer
2  {
3     private static Float currentTime;
4     public void getCurrentTime(){
5        TimeLayerActive();
6     }
7     public Float getTime(){
8        return currentTime;
9     }
10 }
```

図4.2 既存手法によるTimerモジュールの実装

```
1  public class Timer
2  {
3     private static Float currentTime;
4     public void getCurrentTime(){
5     }
6     public Float getTime(){
7        return currentTime;
8     }
9  }
```

図4.3 提案手法によるTimerモジュールの実装

```
1  <aop:config proxy-target-class=“true”>
2   <aop:aspect id=“aspect” ref=“layeraspect”>
3     <aop:pointcut expression=“execution
        (*Timer.getCurrentTime(..))
         id = “timeactivate”/>
4     <aop:after method=“TimeLayerActive”
   pointcut-ref=“timeactivate”/>
5   </aop:aspect>
6  </aop:config>
7  <bean id=“layeraspect” class=“LayerAspect”>
8  </bean>
```

図4.4 提案手法による設定ファイルの記述

Timerクラスの実装は，既存手法と提案手法で大きく異なる．

図4.2の既存手法のコードでは，時刻という文脈を取得するgetCurrentTime()メソッド内でLayerAspectクラスの層を活性化させるメソッドであるTimeLayerActivateメソッド()を実行しなければならない．

一方，図4.3の提案手法のコードではTimerクラスにおいて層を活性化させるメソッドを実行する必要がなく，図

4.4の設定ファイルにおいて文脈の状態を取得するメソッドと層を活性化させるメソッドの依存関係を設定することで層の活性化処理を実現することができる．

5. 提案手法の評価

5.1. 提案手法のメリット

提案手法により層の活性化処理のモジュール化を実現した．層の活性化処理がモジュール化されることにより，層の活性化処理に関わるコードの追加や変更が容易になる．

また，層の活性化処理を文脈の状態を取得するモジュールから切り離すことができ，文脈の状態を取得するモジュールの独立性を高めることができる．

さらに，設定ファイルにより層の活性化処理と文脈の状態を取得する処理の依存関係が明確になるため，文脈依存の振る舞いがどのように切り替えらえるのか把握することが容易になる．

5.2. 提案手法のデメリット

XML形式の設定ファイルを利用しているため，XMLファイルの記述が必要となる点と，その分だけプログラマの負担が増すことがデメリットとなる．

改善策として，DOMを利用してJavaプログラムからXMLファイルを生成するモジュールを作成することで，XMLファイルの記述をなくし，コードの記述量も抑えることができると考えている．

また，提案手法におけるCOPの機能は十分ではない．例えば，提案手法において文脈依存の振る舞いが実行されるとスタックの先頭の層が参照されるが，スタックの先頭の層が文脈依存の振る舞いを持っていない場合，その実行はエラーとなってしまう．現状では，スタックに活性化されている層をプログラマが明確に把握しなければならない．

5.3. 先行研究との比較

言語処理系を拡張することなくCOPの機構を実現した研究としてScala言語を利用した文脈指向DSL[6]が挙げられる．文脈指向DSLは実時間システムの設計を対象にした文脈指向言語であり，実時間を文脈と捉えることで実時間システムの設計におけるCOPの有効性を示している．本研究における文脈指向の機能の実現には文脈指向DSLの実現手法を大いに参考にした．しかし，文脈指向DSLでは層の活性化処理のモジュール化が実現できていないことが問題となっていた．

Java言語を利用して言語処理系を拡張することなく，文脈指向の機構を実現した言語としてJavaCtx[8]が提案されている．しかし，JavaCtxはAspectJのコンパイラを利用しており，実行環境は制限される．さらに，その機能の利用も利用しやすいものであるとは言い難い．

層の活性化処理のモジュール化を実現した文脈指向言語としてはEventCJ[4][5]が挙げられる．EventCJはAspectJのコンパイラを利用することにより，アスペクト指向の機能を導入して層の活性化処理のモジュール化を実現している．EventCJは様々な面で優れた点も多いが，層の活性化処理のモジュール化の仕組みを言語処理系を拡張することで実現しているため，実行環境が制限されることが問題となる．本研究の提案手法は言語処理系を拡張していないため，多くの応用分野で利用可能であると考えられる．

5.4. 今後の課題

今後は，Java言語による文脈指向の機能を高めることが求められる．提案手法のデメリットとして述べたように，提案手法におけるCOPの機構は既存の文脈指向言語と比べて十分に機能を実装できているとはいえない．

COPの機能の実現には実行時にメソッドをディスパッチすることが必要となり，そのためには実行時にメソッドやクラスの情報を取得する仕組みを導入する必要がある．このために，Javaのリフレクション機能を利用できるのではないかと考えている．

6. おわりに

従来の言語処理系を拡張していない文脈指向言語では層の活性化処理が横断的関心事して様々なモジュールに横断して存在していることが問題となっていた．

本研究ではSpringが提供するアスペクト指向のフレームワークを利用することで層の活性化処理のモジュール化を実現する文脈指向プログラミング手法について提案した．

COPを利用して開発されたシステムは多くない．これはCOPの技術を実現した言語の多くが言語処理系を拡張することで実装されていることが理由のひとつとして考えられる．多くのプログラマが容易にCOPという技術に触れる機会を与えるためには言語処理系を拡張することなく，COPの機構を実現することが求められるであろう．

参考文献

1) R. Hirschfeld, P. Constanza, and O. Nierstrasz : “Context-Oriented Programming,” Journal of Object Technology, vol.7, no.3, pp. 125-151, (2008).
2) 紙名哲也 : “文脈指向プログラミングの要素技術と展望”, コンピュータソフトウェア, vol.31, no.1, pp.3-13, (2014).
3) M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara : “ContextJ: Contexted Oriented Programming with Java”, Journal of the Japan Society for Software Science and Technology, vol.28, no.1, pp. 272-292, (2011).
4) 青谷知幸, 紙名哲生, 増原英彦 : “オブジェクトごとの層遷移を宣言的に記述できる文脈指向言語EventCJ”, コンピュータソフトウェア, vol.30, no.3, pp.3_130–3_147, (2013).
5) T. Kamina, T. Aotani, and H. Masuhara : “EventCj: A Context-Oriented Programming Language with Declarative Event-based Context Transition”, Proc. 10th Aspect-Oriented Software Development Conference (AOSD.11), pp. 253-264, (2011).
6) 中村遼太郎, 渡部卓雄 : “実時間システム向けの文脈指向DSL”, 情報処理学会, vol.2013-SE-179, no.31, (2013).
7) 鈴木将哉, 渡部卓雄 : “Objective-Cによる文脈指向プログラミングの実現手法”, 電子情報通信学会ソフトウェアサイエンス, SS2012-32, pp133~138, (2012).
8) G. Salvaneschi, C. Ghezzi, and M. Pradella, “JavaCtx: Seamless toolchain integration for context-oriented programming,” Proc. 3rd Int’l Workshop on Context-Oriented Programming, no.4, (2011).