ソフトウェア工学 111－6
（1996．9．11）

CASEツールによるデザインパターン適用支援

山本 純一　　松本 一教
株式会社 東芝 研究開発センター
システム・ソフトウェア生産技術研究所

本稿では，オブジェクト指向手法を支援するCASEツール上で，デザインパターンの適用を支援する方法を提案する．最初に，デザインパターン適用支援としてCASEツールに望まれる機能を検討し，それらの支援機能をCASEツール上で実現するためのアーキテクチャを提案する．このアーキテクチャでは，各デザインパターンをどのような形式で扱うかがキーとなる．そこで，既存のデザインパターン[1]を分析してデザインパターンの共通構造を構築し，この共通構造を基にデザインパターン記述形式を定義する．最後に，現在開発中のCASEツール上に実装予定の支援機能とその実現方法を具体的に説明する．

CASE Tool Support for Applying Design Patterns

Junichi Yamamoto　　Kazunori Matsumoto
Systems and Software Engineering Laboratory
Research and Development Center, Toshiba Corporation

To facilitate the widespread use of design patterns in software developments, their integration with existing CASE tools is required. This paper proposes a method to deal with design patterns for CASE tools. First, we discuss what facilities are required for CASE tools in order to help users to apply design patterns. Then, we analyze all design patterns described by Gamma et al.[1], and define a description format of design patterns(DFDP). Finally, we propose a CASE tool which manages design patterns represented in terms of DFDP. The CASE tool provides users with support for describing design specifications, understanding design specifications, verifying whether design patterns are applied correctly, and registering new design patterns in the CASE tool.

1　はじめに

デザインパターンとは，オブジェクト指向(OO)設計において繰り返し現れる重要な構造をパターン化し，再利用しやすい形式でまとめたものである．デザインパターンは近年，OO設計ノウハウの再利用方法として注目を集めている．既存のデザインパターンを集めたカタログ[1]も出版されている．

我々は，デザインパターンの適用をCASEツールで支援してほしいという要望が今後高まると予測している．このように考える理由は次の3点である．

1つ目の理由は，我々のデザインパターン適用経験からくるものである．我々は，カタログ[1]に載っているデザインパターンをあるソフトウェア設計に適用評価した．その際，デザインパターンの適用を支援するCASEツールがあれば，設計や実装のコスト低減，および，設計の質の向上に有効であろうとの感触を得た．

2つ目の理由は，OO手法とデザインパターンは，今後より密接に統合されるだろうとの予測による．例えば，D.G.Schmidt[4]はデザインパターン適用報告の中で，「デザインパターンは既存のOO手法を補完するものであり，デザインパターンとOO手法の統合は今後注目を集めるだろう」と述べている．また，Booch氏，Rumbaugh氏，Jacobson氏によって96年7月に公開された「統一モデリング言語 (Unified Modeling Language) Ver. 0.9」では，デザインパターンの適用を設計仕様書上で表現する方法が提案されている．このように，デザインパターンとOO手法がより密に統合されていけば，OO手法を支援するCASEツール(以下，OO-CASEと呼ぶ)にも，デザインパターン支援機能が要求されてくるだろう．

また，デザインパターン適用支援は，従来のCASEツールの設計支援機能を強化できる．これが3つ目

の理由である．例えばPancakeは，専門家の持つ設計ルールに基づく設計をユーザができるようなツールの必要性を唱えている[2]．デザインパターン適用支援機能を備えたOO-CASEは，このような要求にも合致する．

そこで我々は，デザインパターン適用を支援するOO-CASEツールの実現を目指した研究を行っている．本稿では，デザインパターン適用支援としてOO-CASEに望まれる機能を明確にし，それらの機能の実現方法を提案する．2節では，デザインパターン適用を支援するためにOO-CASEツールに望まれる機能を検討し，それらの機能をOO-CASE上で実現するためのアーキテクチャを提案する．3節では，デザインパターンをOO-CASEツールで扱うための記述形式を提案する．4節では，我々が開発中のOO-CASEツール上で実装予定の支援機能とその実現方法を説明する．

2 デザインパターン適用支援構想

2.1 OO-CASEに望まれる支援機能

はじめに述べたように，我々はデザインパターンをあるソフトウェア設計に適用評価した．この適用評価では，カタログ[1]に載っている23個のデザインパターンを適用候補とした．デザインパターンの適用は以下の手順で行った:(1) デザインパターンを理解し，適当なデザインパターンを選択する，(2) デザインパターンを設計に適用し，設計仕様書を記述 / 修正する，(3) デザインパターンが正しく適用されているかどうかを確認する，(4) 設計仕様書に基づいて実装する．我々はこの適用経験を通して，デザインパターン適用支援としてOO-CASEに次の機能が必要であると考えた．

S1: 問題解決に適したデザインパターンを選択する支援．

S2: デザインパターンを適用する際に，設計仕様書を記述 / 修正する支援．

S3: デザインパターンが正しく適用されているかどうかの検証．

S4: デザインパターンが適用された設計仕様書からのプログラム自動生成．

また，上記適用で直接必要性が感じられたわけではないが，以下2つの機能も重要と考えている．

S5: デザインパターンが適用されている設計仕様書を理解する支援．

S6: ユーザが自分で新たなデザインパターンを登録できる機構．

S5によりユーザは，どのデザインパターンが設計のどの箇所にどのように適用されているのか，を理解できる．この機能は特に保守工程で重要となる．S6は，新たなデザインパターンを登録するしくみを提供する．今後新たなデザインパターンが出現することを考慮すると，この機能は必須である．

2.2 ターゲットとする支援機能

我々の最終目標は2.1で挙げた全機能を備えたOO-CASEを開発することにあるが，本稿ではまず，S2, S3, S5およびS6の支援を目指す．

多くのOO手法では，ソフトウェア構造を設計 / 理解する上で，クラス図が中心的役割を果たす．カタログ[1]の各デザインパターンも，主にクラス図を用いてパターンの構造を説明している．そこで，S2とS5の支援としてはクラス図を対象とする．

S3の支援機能としては，構文的な検証と，設計者の意図にまで踏み込んだ検証の2種類が考えられる．前者は，デザインパターンが構文的に正しく適用されているかどうかを検証する．例えば，足りないクラスはないか，などをチェックする．一方，後者は，そのデザインパターンの適用により実際に設計が改善されるか，より適当なデザインパターンがあるか，などをチェックするものである．我々は今のところ，前者 (構文的な検証) のみを扱う．

本稿ではS1を検討対象から除外する．S1を支援するには，他の支援機能とは大きく異なったメカニズムが必要と考えられるためである．また，S4の支援も本稿では検討しない．S4の支援にはS2やS6が必須であるため，まずはS2とS6を実現するべきであろう．

2.3 デザインパターン適用支援のためのCASEアーキテクチャ

我々は現在，OO-CASEツールCOOAD (CASE tool for Object-Oriented Analysis and Design) を開発している．COOADは統一モデリング言語 (Booch氏，Rumbaugh氏，Jacobson氏がまとめつつある) によるソフトウェア開発を支援する．COOADはクラス図エディタなどの仕様書エディタを備えており，ユーザはこのエディタを通して分析 / 設計の仕様書を記述する．COOADは設計結果からC++などのプログラムのスケルトンを自動生成する．

現在我々は，COOADにデザインパターン適用支援機能を組み込むことを検討中である．COOADは，ユーザが記述した設計仕様書を図1に示したアーキテクチャで保持する．COOADは「図データ」，「デザインパターンデータ」，「対応テーブル」と呼ばれる3種類の保存データを扱う．設計仕様書中の図に関する情報は図データとして保存される．各デザインパターンはデザインパターンデータとしてあらかじめ保存されている．ユーザは設計仕様書を記述する際に，デザインパターンを適用できる．ユーザがあるデザインパターンを自分の設計のある箇所に適用すると，その適用箇所に一つの対応テーブルが用意される．対応テーブルには，ユーザの設計仕様書とデザインパターンとの対応関係が記録される．

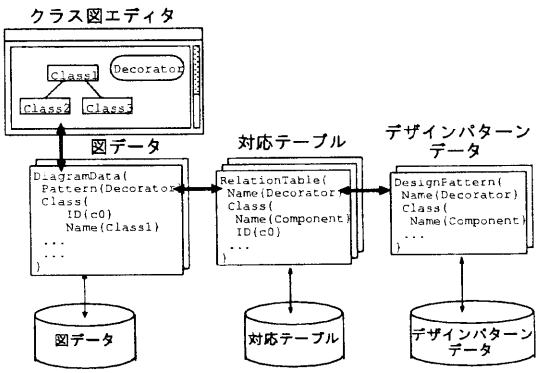

図 1: COOADのデータアーキテクチャ

3 デザインパターンの記述形式

2.2で挙げた支援機能を実現するためには，2.3の各保存データ(特にデザインパターンデータ)をどのような形式で扱うか，がキーとなる．本節では，これら3種類の保存データの形式を定義する．

2.2で述べたように，我々は機能S2とS5をクラス図を用いて支援する．また，機能S6のためには，任意のデザインパターンをCASEツール上で扱える必要がある．一方，カタログ[1]では，各デザインパターンの「構造」は主にクラス図を用いて表現されている．そこで我々は，カタログ[1]中の全デザインパターンを「構造」に注目してOO分析した．すなわち，各デザインパターンの「構造(クラス図で表現されている)」の一般構造をクラス図として表現した．この分析結果を3.1で説明する．

3.2では，3.1での分析結果を基にデザインパターン記述形式を定義する．この記述形式を本稿ではDFDP (Description Format of Design Patterns) と呼ぶ．

3.3では，3種類の保存データの形式を説明する．

3.1 デザインパターンの共通構造

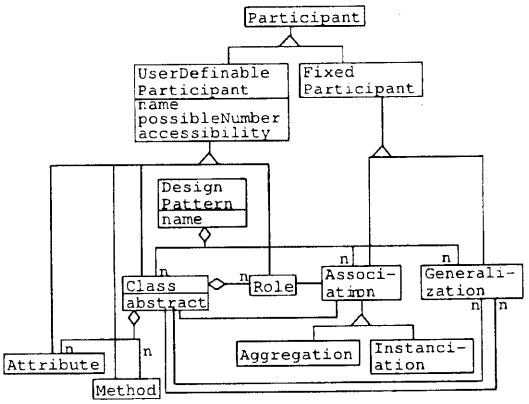

図 2: デザインパターンの一般構造を表すクラス図(DPClassDiagram)

3.1.1 DPClassDiagram

各デザインパターンの「構造(クラス図で表現されている)」を分析して得られたクラス図を図2に示す(OMT[3]のクラス図の表記法に基づく)．以下では，このクラス図をDPClassDiagramと呼ぶ．

DesignPatternクラスはデザインパターン自体を表す．**DesignPattern**のインスタンスは各デザインパターンである．**DesignPattern**は**Class**，**Generalization**，**Association**を集約する．

Classは各デザインパターンに登場するクラスである．**Class**は**Role**と**Attribute**と**Method**を集約する．**Association**は**Role**と**Class**に関連する．また，**Generalization**は**Class**に関連する．**Aggregation**と**Instantiation**は**Association**のサブクラスである．**Instantiation**とは，あるクラスが別クラスのインスタンスを生成することを表すクラス間関係である([1]のクラス図にて導入されている表記法)．

クラス図の最上部にある**Participant**クラスは，各デザインパターンを構成する要素を表す．本稿では，**Participant**のサブクラスのインスタンスを「(各デザインパターンの)構成要素」と呼ぶ．

Participant は **UserDefinableParticipant** と **FixedParticipant** に分類される．**UserDefinableParticipant** は，開発対象ソフトウェアに特化した名称を持ち得る構成要素を表す．**Class**, **Attribute**, **Method**, **Role** は **UserDefinableParticipant** のサブクラスである．一方, **Generalization** と **Association** は **FixedParticipant** のサブクラスである．これらの名称は開発対象に依存しない．**UserDefinableParticipant** は 3 つの属性 (**name**, **possibleNumber**, **accessibility**) を持つ．**name** は，各デザインパターンに登場する構成要素の名称を表す．**possibleNumber** は，あるデザインパターンの適用箇所において，その構成要素がいくつ存在し得るかを表す．**accessibility** は，デザインパターンの構成要素以外のクラスがその構成要素にアクセス可能かどうかを True/False で表す．

3.1.2 例

例として，Decorator パターン (カタログ [1] の中の 1 つ) を考える．Decorator パターンとは，オブジェクトに新たな機能を動的に追加する方法を提供するためのパターンである (このパターンの詳細については [1] を参照してほしい)．図 3に Decorator パターンの構造を表すクラス図を示す．

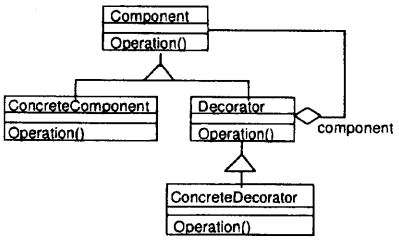

図 3: Decorator パターンの構造

Decorator パターンの構成要素は 4 つのクラス (Component, ConcreteComponent, Decorator, ConcreteDecorator), 一つの集約, 3 つの継承, 1 つの役割, 4 つのメソッドである．例えば，4 つのクラスは **Class** のインスタンスとなる．また，Component クラスと Decorator クラスの間の集約は **Aggregation** のインスタンスとなる．

Decorator パターンにおいては Decorator クラスは一つしか存在し得ないので，Decorator クラスの **possibleNumber** は 1 となる．Decorator パターンの外部のクラスからは，Decorator クラスのインスタンスに直接アクセスすることはないので，Decorator クラスの **accessibility** は False となる．

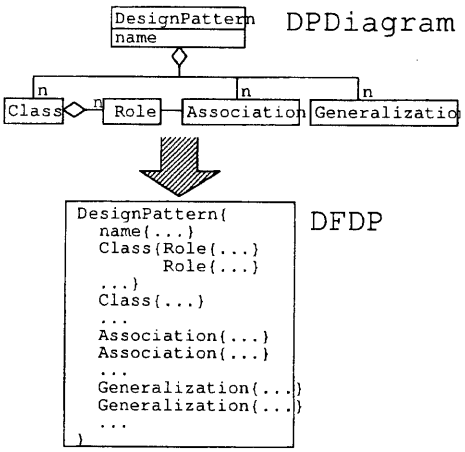

図 4: DPClassDiagram から DFDP の導出規則

3.2 DFDP の定義

DPClassDiagram を基に，デザインパターン記述形式 DFDP を BNF (Backus Naur Form) で定義する．

図 4に，DPClassDiagram から DFDP を得る規則を例示した．上部のクラス図では，**DesignPattern** は **name** という属性を持ち，**Class**, **Generalization**, **Association** を集約している．このクラス図から下部の言語形式を得る．

この規則に従って得られた DFDP の完全な定義を図 5に示す．ここでは < と > で囲まれた文字列はノンターミナルを表し，他の文字列はターミナルを表す．記号 '*' は，直前の文字列の複数回の繰り返しを意味する．

また，Decorator パターンを DFDP 形式で表現した例を図 6に示す．

3.3 各種保存データ形式の定義

3 種類の保存データ (デザインパターンデータ，図データ，対応テーブル) の形式を説明する．

デザインパターンデータ デザインパターンデータは 3.2で定義した DFDP 形式とする．

図データ ユーザが設計時に記述するクラス図の例を図 8に示す．このクラス図に対応する図データ

```
<DesignPattern> ::=
     DesignPattern{
         PatternName{<name>}
         <class>*
         <generalization>*
         <association>*
         <aggregation>*
         <Instantiation>*     }
<class>::=
    Class{
         Name{<name>}
         PossibleNumber{<number>}
         Accessibility{<boolean>}
         Abstract{<boolean>}
         <attribute>*
         <method>*
         <role>* }
<attribute>::=
    Attribute{
         Name{<name>}
         PossibleNumber{<number>}
         Accessibility{<boolean>} }
<method>::=
    Method{
         Name{<name>}
         PossibleNumber{<number>}
         Accessibility{<boolean>} }
<role>::=
    Role{
         Name{<name>}
         PossibleNumber{<number>}
         Accessibility{<boolean>} }
<generalization>::=
    Generalization{
         Source{<className>}
         Target{<className>} }
<association>::=
    Association{
         Source{<className>,<cardinality>}
         Target{<roleName>,<cardinality>}}
<aggregation>::=
    Aggregation{
         Source{<className>,<cardinality>}
         Target{<roleName>,<cardinality>}}
<Instantiation>::=
    Instantiation{
         Source{<className>,<cardinality>}
         Target{<roleName>,<cardinality>}}
<name>::= ...string
<className>::= ...string
<roleName>::= ...string
<cardinality>::= ...integer
<number>::= ...integer
<boolean>::= ...True or False
```

図 5: DFDP の定義

```
DesignPattern{
    PatternName{Decorator}
    Class{
            Name{Component}
            PossibleNumber{1}
            Accessibility{True}
            Abstract{True}
            Method{
                    Name{ Operation }
                    Accessibility{True}
                    PossibleNumber{1} }
            Role{
                    Name{ component } } }
    Class{  Name{ConcreteComponent}
            (以下省略) }
    Class{  Name{Decorator}
            (以下省略) }
    Class{  Name{ConcreteDecorator}
                (以下省略) }
    Generalization{
            Source{ Component }
            Target{ ConcreteComponent } }
    Generalization{ (以下省略) }
    Generalization{ (以下省略) }
    Aggregation{ (以下省略) }
}
```

図 6: Decorator パターンの DFDP 表現

```
ClassDiagram{
    Class{ ID{c0}
           Name{ ClientClassA }}
    Class{ ID{c1}
           Name{ ClientClassB }}
    Class{ ID{c2}
           Name{ VisualComponent }
           Method{ ID{m(2,0)}
                   Name{ Draw() } } }
    Class{ ID{c3}
           Name{ TextView }
           (以下省略) }
    Class{ ID{c4}
           Name{ PSView }
           (以下省略) }
    Generalization{ ID{i0}
                    Source{c2}
                    Target{c3} }
    Generalization{ ID{i1} (以下省略) }
    Aggregation{ ID{ag0} (以下省略) }
    Aggregation{ ID{ag1} (以下省略) }
}
```

図 7: 図データの例

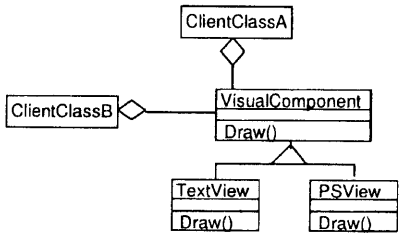

図 8: クラス図の例

```
<RelationTable>::=
     RelationTable{
        RelationTableID{<relationTableID>}
        PatternSymbolID{<elementID>}
        ID{<elementID>}
        PatternName{<PatternName>}
        <class>*
        <generalization>*
        <association>*
        <aggregation>*
        <Instantiation>*     }
<class>::=
    Class{  Name{<name>, <elementID>}
            <attribute>*
            <method>*
            <role>* }
<attribute>::=
    Attribute{ <name>, <elementID> }
<method>::=
    Method{ <name>, <elementID> }
<role>::=
    Role{ <name>, <elementID> }
<generalization>::=
    Generalization{
        ElementID{<userID>}
        Source{<className>, <userID>}
        Target{<className>, <userID>}
<association>::=
    Association{
        ElementID{<userID>}
        Source{<className>, <userID> }
        Target{<roleName>, <userID> }}
<aggregation>::=
    Aggregation{
        ElementID{<userID>}
        Source{<className>, <userID> }
        Target{<roleName>, <userID> }}
<Instantiation>::=
    Instantiation{
        ElementID{<userID>}
        Source{<className>, <userID> }
        Target{<roleName>, <userID> }}
<name>::= ...string
<className>::= ...string
<roleName>::= ...string
<elementID>::= ...string (ID)
<relationTableID>::= ...string (ID)
```

図 9: 対応テーブルの定義

を図 7に示す (本稿では説明を簡潔にするため, 位置などのデータを省略して簡略化している.)

対応テーブル 対応テーブルの形式を図 9に示す. この対応テーブルの定義は DFDP の定義 (図 5) と似ているが, 二つの相違点がある. 一つは, 対応テーブルには ElementID などIDに関する項目が含まれている点である. これらのIDには, ユーザが記述した図の設計要素のIDが記録されている. このIDを用いて, デザインパターン構成要素とクラス図の設計要素との対応付けがなされる. もう一つの相違点は, 対応テーブルには PossibleNumber, Accessibility, Abstract というキーワードが存在しない点である. これらの情報は各デザインパターンが持っているため, 対応テーブルで個別に持つ必要はない.

4 CASE ツールによるデザインパターン適用支援

2, 3節では, OO-CASE のアーキテクチャとデータ形式を説明した. 本節では, このアーキテクチャとデータ形式上で, どのような支援機能をどのように実現するのかを具体的に説明する.

4.1 デザインパターン適用支援

ユーザは, 図 10に示すデザインパターン適用ウィンドウ (単に適用ウィンドウとも呼ぶ) を通して, 自分の設計にデザインパターンを適用する. 適用ウィンドウには, デザインパターン一覧, 選択されているパターン名, 選択したパターンのクラス構造, および, 対応関係設定表が含まれる.

ユーザは次の手順で, 自分の設計にデザインパターンを適用する.

1. ユーザはデザインパターン一覧から 1 つのデザインパターンを選択する.
2. 選択されたデザインパターンの構造 (クラス図) がクラス構造欄に表示される. 対応関係設定表の構成要素欄には, そのデザインパターンの構成要素名が並ぶ. 存在可能数欄には, possible number の値が入る.
3. ユーザは設計要素欄に設計要素名を入力する.
4. COOAD は対応テーブルを生成する. また, ユーザが記述しているクラス図の図データを一部自動的に修正する.

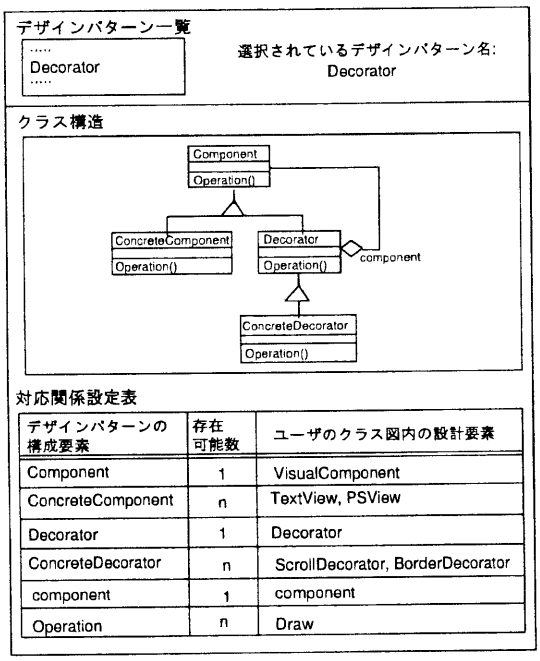

図 10: デザインパターン適用ウィンドウ

例として，ユーザが図 8のクラス図に Decorator パターンを適用する場合を考える．図 10は，ユーザが Decorator パターンを選択し，関係設定テーブルを埋めた直後の適用ウィンドウである．設定が完了すると，COOAD はユーザのクラス図 (図 11参照) および図データを修正 (修正後の図データは省略) し，また，対応テーブルを生成する (図 12参照)．修正されたクラス図にはデザインパターンアイコンが含まれている．このアイコンは図データに記録される．対応テーブルの PatternSymbolID には，このアイコンの ID が記録される．

さらに，ユーザが自分の設計を修正したとき (クラス図にクラスを追加するなど) には，COOAD はその操作に合わせて対応テーブルを自動修正する．

例として，図 11に，BitmapView クラスを VisualComponent のサブクラスとして追加した場合を考える．この場合，BitmapView クラスは，Decorator パターン構成要素の ConcreteComponent クラスまたは Decorator クラスのどちらかに対応するはずである．ここで，Decorator クラス と ConcreteComponent クラスの存在可能数はそれぞれ 1 と n(任意個数) である．従って，BitmapView クラスは ConcreteComponent と対応付く．COOAD は BitmapView クラスを ConcreteComponent として対応テーブルに登録する．

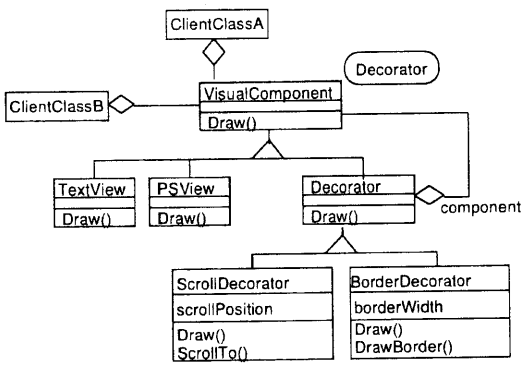

図 11: Decorator パターンが適用されたクラス図

4.2 設計仕様書の理解支援

図 11のクラス図は Decorator とラベル付けられたパターンアイコンを含む．このアイコンは，その辺りに Decorator パターンが適用されていることを示唆している．

我々は現在，クラス図上でより詳細に適用状況を把握する方法を検討中である．例えば，「ユーザがこのアイコンをクリックすると，このパターン適用に関係する設計要素が強調表示される」などの方法が考えられる．Booch 氏，Rumbaugh 氏らの統一モデリング言語では，「利用関係」を使ってデザインパターンの適用状況を表現する方法が提案されている．

4.3 デザインパターン適用の仕方の検証支援

ユーザが自分の設計にデザインパターンを適用する際，誤った方法で適用する可能性がある．COOAD の検証支援機能は，以下のような誤った適用を検出する．この機能により，誤った適用を防ぐことができる．

E1: アクセスしてはいけない設計要素へのアクセス

E2: 存在していなければならない設計要素の不足

E3: 存在してはいけない設計要素の存在

E4: デザインパターンの構成要素との対応が曖昧な設計要素の存在

```
RelationTable{
    RelationTableID{rt0}
    PatternSymbolID{p0}
    PatternName{Decorator}
    Class{ Name{ Component, c2 }
           Method{ Operation(), m(2,0) }
           Role{ component, r(2,0) } }
    Class{ Name{ ConcreteComponent, c3 }
           Method{ Operation(), m(3,0) } }
    Class{ Name{ ConcreteComponent, c4 }
           Method{ Operation(), m(4,0) } }
    Class{ Name{ Decorator, c5 }
           Method{ Draw(), m(5,0) } }
    Class{ Name{ ScrollDecorator, c6 }
           Method{ Draw(), m(6,0) } }
    Class{ Name{ BorderDecorator, c7 }
           Method{ Draw(),m(7,0) } }
    Generalization{ ElementID{i0}
         Source{Component, c2}
         Target{ConcreteComponent, c3} }
    Generalization{ ElementID{i1} (以下省略)}
    Generalization{ ElementID{i2} (以下省略)}
    Generalization{ ElementID{i3} (以下省略)}
    Generalization{ ElementID{i4} (以下省略)}
    Aggregation{ ElementID{ag2} (以下省略)}
}
```

図 12: 対応テーブルの例

例として，ユーザが図 11のクラス図を図 13に変更した場合を考える．

この変更の結果，ClientClassB が TextView と PSView を直接集約することになった．しかし，ClientClassB は TextView と PSView に直接アクセスすべきではない (TextView と PSView は Decorator パターンの ConcreteComponent に対応しており，また，ConcreteComponent の Accessibility は False であるため)．そのため，この変更は E1 エラーを引き起こす．COOAD は対応テーブルおよび Decorator パターンの DPDF 情報を基に，このエラーを検出できる．

また，図 13のクラス図では VisualComponent と Decorator の間の集約が削除されている．しかし，Decorator と Component の間には集約が存在しなければならず，この削除は E2 エラーとなる．COOAD は Decorator パターンの DFDP を基に (Decorator と Component の間には集約が存在しなければならないことが表現されている)，このエラーも検出できる．

この例では現れていないが，E3 と E4 のエラーも同様に COOAD で検出可能である．

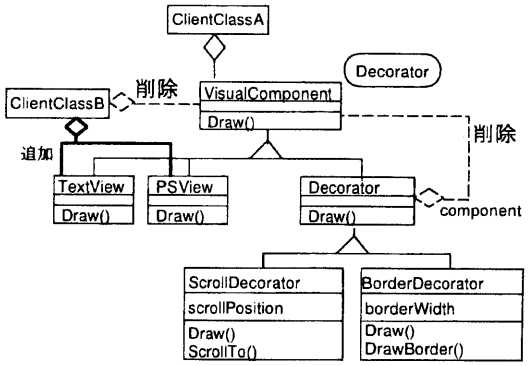

図 13: エラーを含むクラス図

5 おわりに

本稿では，CASE ツールでデザインパターンの適用を支援する方法を述べた．最初に，デザインパターン適用支援として CASE ツールに望まれる機能を検討し，それらの支援機能を CASE ツール上で実現するためのアーキテクチャを提案した．このアーキテクチャでは，各デザインパターンを保存する「デザインパターンデータ」と，デザインパターンの適用の仕方を記録する「対応テーブル」が中心的役割を果たし，これらのデータ形式をどう定義するかがキーとなる．そこで，既存のデザインパターンを OO 分析してデザインパターンの共通構造をクラス図として表現し，この共通構造を基にデザインパターン記述形式 DFDP を定義した．最後に，現在開発中の CASE ツール COOAD で検討している支援機能とその実現方法を説明した．

今後は，COOAD へのデザインパターン支援機能の実装を完了し，適用評価および機能改良 / 拡張を行っていく予定である．

参考文献

[1] Gamma, E., Helm, R., Johnson, R., Vlissides, J.(本位田真一, 吉田和樹監訳), オブジェクト指向における再利用のためのデザインパターン. ソフトバンク, 1995.

[2] Pancake, C. M. The Promise and the Cost of Object Technology: A Five-Year Forecast. *Communication of the ACM*, Vol. 38, No. 10, pp. 33–49, 1995.

[3] Rumbaugh, J. et al. *Object-Oriented Modeling and Design.* Prentice Hall, Inc, 1991.

[4] Schmid, H. A. Using Design Patterns to Develop Reusable Object-Oriented Communication Software. *Communication of the ACM*, Vol. 38, No. 10, pp. 65–74, 1995.