フレームワークとプラグインコンポーネントの インタフェースマッチングに関する研究

2002MT017 飯島 佳奈, 2002MT063 大江 公仁枝

指導教員　青山 幹雄

1. はじめに

開発においてフレームワークは利用性が高く, コンポーネントと組み合わせて利用されるがインタフェースの互換性の保持が必要である. しかし機能変更などのためのバージョンアップで各コンポーネントのインタフェースが変化するため, 互換性の維持が困難である. 本研究は統合開発環境として注目されている Eclipse のプラグインアーキテクチャを例に, コンポーネントのインタフェースマッチング問題を分析する. さらにマッチングを自動化するために, インタフェースの差分を視覚化するツールを開発し, 例題により有効性を確認する.

2. コンポーネントの組み合わせ

フレームワークはシステムの雛型である. フレームワークに追加するコンポーネントがプラグインである. プラグインは単体で動作せず, 必要に応じてフレームワークに組み込んだり, 外したりできる.

プラグインの結合方法は制約の強い順に, Java Applet, Lightweight Application Development , Pluggable Component, Eclipse に分類できる[1]. Applet はブラウザのみで実行され, 同じクラスの拡張に限定される. Lightweight Application Development は, 1 つのアプリケーションに対して複合的なプラグインを拡張でき, 違うインタフェースも持つことができる[2]. しかしプラグインの多重結合については言及していない. Pluggable Component は, 実行時にコンポーネントを交換するための基礎構造を提供する[3]. ブラウザ以外でも実装できるので Applet よりも制約が弱い. Eclipse は Java で実装された IDE (Integrated Development Environment)をフレームワークとツールに分離し, ツールをプラグインとして組み込むことで, 新しい機能を追加, 拡張できる. 実行中のプログラムに, プラグインを追加できない制約があるが, この中では最もプラグインの制約が弱い.

3. プラグインインタフェースの問題点

3.1. プラグインインタフェースの問題点

Eclipse はオープンソースで自由に入手できるフレームワークである. Eclipse に組み込み可能なプラグインは多く存在するが, 図 1 で示すように必ずしも Eclipse の版すべてに組み込み可能ではない. しかし開発者がプラグインによって, 版の異なる Eclipse を用意する必要があると効率が悪い. 従って複数の版の Eclipse とプラグインとの互換性を向上する必要がある.

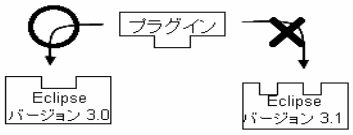

図 1. Eclipse におけるプラグインインタフェースの問題

3.2. Eclipse と EclipseUML の実験

表 1 は, 以上の問題を確認するために行った実験結果である. Eclipse 上で動く UML ツールの EclipseUML をプラグインの例として Eclipse にインストールし, Eclipse 上で作成したプログラムの UML 図が作成可能かを示した. インストール可能で UML 図も作成可能な場合は◎, インストールのみ可能な場合は○, 両者不可能な場合は×で示す.

表 1. Eclipse と EclipseUML の実験結果

Eclipse (リリース時期) \ EclipseUML (リリース時期)	2.0.0 (2004/10/26)	2.1.0 (2005/07/18)
2.1.3 (2004/03/10)	○	×
3.0.0 (2004/06/25)	◎	○
3.0.1 (2004/09/16)	◎	○
3.0.2 (2005/03/11)	◎	○
3.1.0 (2005/06/27)	○	◎
3.1.1 (2005/09/29)	○	◎

表 1 より, UML 図を作成する際に, Eclipse を改版すると EclipseUML も改版する必要があることが分かる. 従って両者の互換性の点から, プラットフォームが変わればプラグインも変える必要性があるという, 相対的な問題であることが確認できた.

版の変化による機能追加が原因で, インタフェースが変わる可能性がある. この変更を, できる限り抑える開発方法の一つがリファクタリング(Refactoring)である. リファクタリングは API を変化させないが(Non-breaking API), これで対応しきれないほどの機能追加は API を変化せざるを得ない(Breaking API)[4].

本研究のインタフェースマッチング問題は，この時発生していると考えられる．今回の UML 図が作成できない問題も，Eclipse と EclipseUML プラグインのインタフェースの互換性がとれないためである．

4. Eclipse

問題を解決するにあたって，Eclipse がプラグインをどのような方法で組み込んでいるかを知る必要がある．

4.1. Eclipse のアーキテクチャ

図 2 のように，Eclipse のプラットフォームはいくつかのサブシステムから構成される．個々のサブシステムは 1 つ以上のプラグインで実装される．主なものを次に説明する．

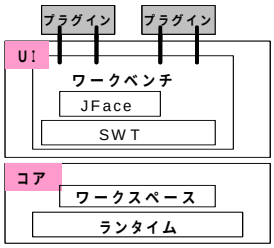

図 2. Eclipse Platform のアーキテクチャ

(1) ランタイム ： プラットフォームのコア部分で，プラグインを動的に検出し，登録･実行を行う．Eclipse プラットフォームの実行を管理するクラスである．

(2) ワークスペース ： プロジェクト，フォルダ，ファイルなどのリソースを管理するサブシステムである．リソース･プラグインにより，リソースを操作する API を提供する．

(3) ワークベンチ ： 他のプラグインが Eclipse のユーザインタフェース(UI)を拡張できるように，様々な拡張ポイントを提供する．拡張ポイントとは，フレームワークを拡張するために用意された”スロット”で，一意に識別可能な ID が割り当てられる．拡張ポイントには，対応する API が割り当てられ，これを使って機能拡張を行う．また開発したプラグインに拡張ポイントを宣言することで，他のプラグインに対してその機能を提供できる．

4.2. プラグインの結合方法

図4に示すように，個々のプラグインは，Eclipseをインストールした場所にある Plugins フォルダの下の，プラグインごとに作成される独自のフォルダにインストールされる．ここにはプラグインの構成リソース(マニフェストファイル，jar ファイル，及び他のリソース)がコピーされる．マニフェストファイル(plugin.xml)はプラグイン仕様を記述するメタデータである．そして，Eclipse ランタイムにプラグインをアクティブにするために必要な情報を提供する．図 3 に例として，HelloWorld のマニフェストファイルを示す．このファイルには，主に次の情報が記述されている．

(1) プラグインの名前などの情報

(2) プラグインが利用する他のプラグインへの依存関係(requires)

(3) プラグインを実装した Java ライブラリの指定(runtime)

(4) そのプラグインの拡張ポイント(extension-point)
※図 3 の HelloWorld.java の例にはついていない．

(5) 利用する拡張ポイント(extension)

```
<plugin id="org.eclipse.contribution.hello"
  name="Hello World"
  version="1.0.0"
  provider-name="">
   <requires>
    <import plugin="org.eclipse.ui"/>
   </requires>
   <runtime>
       <library name="hello.jar">
        <export name="*"/>
       </library>
   </runtime>
   <extension point="org.eclipse.ui.actionSets">
   </extension>
</plugin>
```

図 3. Hello World.java のマニフェストファイルの構造

図 4 に示すように，プラグイン･マニフェストファイルの構文解析されたプラグイン仕様は，プラグイン･レジストリ API を通してプログラムから利用でき，プラグイン･レジストリのリポジトリにキャッシュされる．インストールされたプラグインは Eclipse ランタイムによりアクティブ化される．アクティブ化は，ランタイム･クラスをメモリにロードし，インスタンス化し初期化することである．プラグイン･インスタンスはアクティブになると，プラグイン･レジストリを使ってそれ自身の拡張ポイントに関係のある拡張機能を発見し，Eclipse Platform を使ってアクセスされる[5,6].

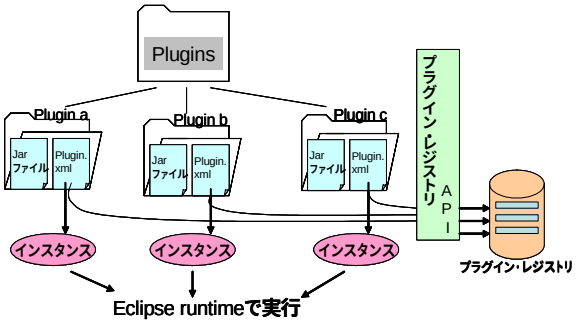

図 4. プラグインの結合方法

4.3. プラグインの仕掛け

Eclipse では，追加拡張の際に，基礎となるインタフェースの定義を変更することなく Eclipse のインタフェースを進化させるメカニズムが存在する．コア･ランタイムが提供するその拡張メカニズムには図 5 に示すデザインパターンが利用されている．

(1) IAdaptable：getAdapter()メソッドのみを持つインタフェースで，これによりオブジェクトが特定のインタフェースを持っているか動的に調べられる．

(2) IAdapterFactory：実装するクラスのコードを変えることなくその機能を拡張できる仕組みである．まず図5に示すように，特定の型に対するアダプタファクトリを実装する．また， IAdapter を使って必要なインタフェースのアダプタを生成する．オブジェクトの拡張ポイントに合うインタフェースを拡張するように，サブクラスのオブジェクトを生成する．図5の右下の IAdaptable は，要求を受け取るオブジェクトを引数に取る．IAdapterManager はアダプタ作成を管理する．この仕組みを使うことによってオブジェクトごとのアダプタが作成される[4].

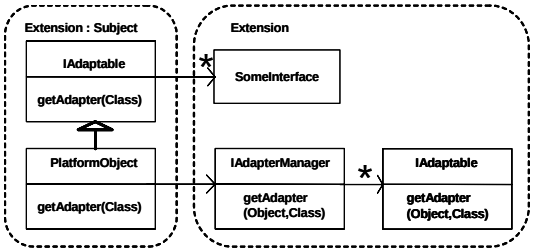

図 5. IAdaptable と IAdapterFactory

5. インタフェースマッチング視覚化の提案

5.1. インタフェースの差分の視覚化

4 章で述べたように，Eclipse は IAdapterFactory デザインパターンによってインタフェースのマッチングを行っている．しかし版の違いによって対応できない問題が生じている．この原因は，機能追加により，以前の Eclipse に依存した情報を持ったプラグインが，別の版ではその情報を参照できないために起こる問題ではないかと考える．

この問題を解決するために機能追加におけるインタフェースの差分を視覚化する．またその情報をデータベースに格納し，そこから参照することでどの版でも対応できるようにする．最終的にはそれをユーザが処理することなく，コンピュータが自動的に処理を行う環境を目指す．

本研究では自動ツールに向けての第一歩として，インタフェースの差分を視覚的に表示できるツールを作成した．開発環境は Eclipse3.0.1 と Eclipse3.1.1，実行環境は Eclipse3.0 系と Eclipse3.1 系である．入力はボタンで行い，出力は Eclipse 上で行う．プログラムサイズは合計で，723 行，22912byte である．

ツールの流れを図 6 に示す．このツールは，必要なマニフェストファイルの名前を指定すると，参照しているクラスで利用されているメソッドを抽出できる．さらに 2 つの版の間で利用されているメソッドの差分を視覚的に表示できる．

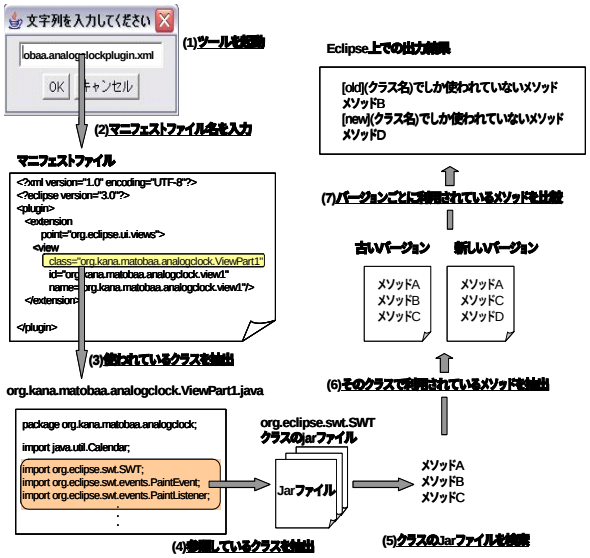

図 6. 作成したツールの流れ

例として Eclipse2.1.3 と Eclipse3.1.1 のインタフェースの差分を検証するため，2 つの版の実行環境を用意した．またツールの検証のため，Eclipse 上で現在時刻をアナログの時計で表示する Analogclock プラグインを実装した．出力結果を図 7 に示す．

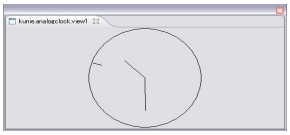

図 7. Analogclock プラグイン

プラグインのマニフェストファイルのパスを図 6 の(2)に示すように入力すると，プラグインが参照しているクラスが利用しているメソッドをファイルに出力する．比較のためファイル名は，図 8 に示すように，旧版は[old]+クラス名，新版は[new]+クラス名とする．

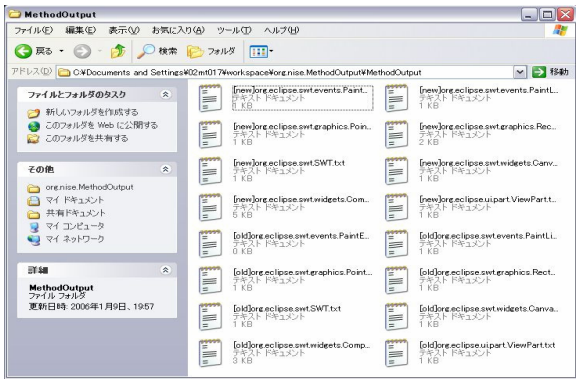

図 8.出力されたファイル

同じクラス名の旧版と新版のファイルを比較し，差分を

Eclipse 上に出力する. その際, 図9 に示すように差分のあるクラスだけを出力するようにした.

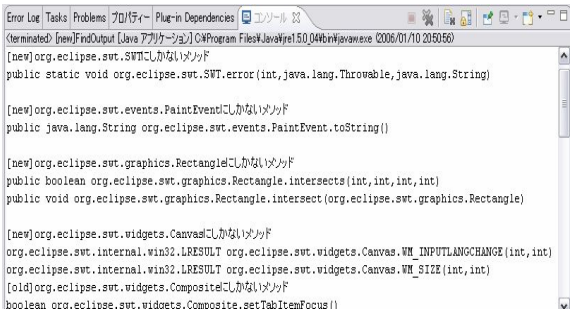

図 9.出力結果

Analogclock の例では 8 個のクラス中 6 個のクラスに違いが見られ, メソッドの差分は 29 個であった. さらに他の例として, HelloWorld を表示するプログラムで実験した. その結果, 5 個のクラス中 3 個のクラスに違いが見られ, メソッドの差分は 6 個であった. 2 つのプラグインの解析に必要な実行時間を表 2 に示す.

表 2.解析に必要な実行時間

プラグイン＼回数	1回目	2回目	3回目	4回目	5回目	2~5の平均
HelloWorldプラグイン	2.73秒	1.45秒	1.55秒	1.37秒	1.48秒	1.46秒
Analogclockプラグイン	2.86秒	1.67秒	1.59秒	1.64秒	1.61秒	1.62秒

表 2 に示すように, 1 回目は 2 回目以降より時間がかかる. これはプラグインをロードする時間が余分にかかるためだと考えられる. よって平均時間は 2~5 回の平均時間を示した. 実験結果より, このツールを使用すると, マニフェストファイルのパスを入力するだけでメソッドの差分が数秒で視覚化できることが分かる.

本研究では, このツールによって抽出されたメソッドの差分をデータベースに格納し, プラグイン利用時に参照可能にすることで, インタフェースのマッチングをとることができると考えている. また, この差分の定義に XMI の利用を検討している.

5.2. XMI によるインタフェースの定義

XMI(XML Metadata Interchange)は, UML のメタモデルを交換する標準仕様である. MOF(Meta Object Facility)は, メタモデルを記述する文法の仕様である. 構文仕様は XML 形式で書く. オブジェクト指向のインタフェース交換に適用できると考えている.

5.3. 一般のコンポーネントへ適用

本研究では, Eclipse で利用されている仕組みを一般のコンポーネントに適用することで, インタフェースマッチングの問題を解決できると考える. 図 10 に示すように全てのコンポーネントに XML 記述によるマニフェストファイルと, アダプタを作る機能を持たせる. 組み合わせる際, 相手が同一記述方法のインタフェース情報を持っているので参照できる. またインタフェースが合わない場合に, 自らアダプタを作成することで既存のインタフェースを変えることなく, 新しいインタフェースとマッチングできる. 従って, コンポーネント間でインタフェース情報を参照してマッチングが可能となる.

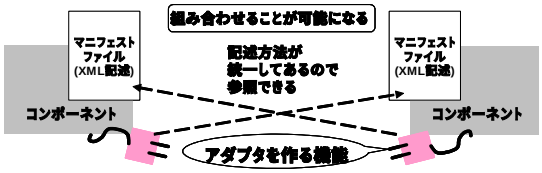

図 10. 一般化したコンポーネントの組み合わせの提案

6. 今後の課題

解決策の第一歩として, 版ごとのインタフェースの差分を視覚化するツールを開発した. 今後はこの差分をデータベースに格納し, 版に依存した情報を参照可能にする必要がある. さらに外部開発のプラグインにも対応させなければならない. また, アダプタ作成機能の一般化に向けて, コンポーネントにマニフェストファイルとアダプタ作成機能を付加する必要がある.

7. まとめ

本研究では Eclipse のプラグインアーキテクチャを例に, コンポーネントのインタフェースマッチング問題を分析した. その問題を Eclipse と EclipseUML の実験を通して考察し, マッチングの自動化を提案した. その第一歩としてインタフェースの差分を視覚化するツールを作成し, 具体例で効果を示した. さらにコンポーネントを組み合わせるために, メタデータとアダプタによる解決案を提示した.

8. 参考文献

[1] R. Chatley, et al., Painless Plugins, department of computing, Imperial college.

[2] J. Mayer, et al., Lightweight Plug-in-Based Application Development, 2002.

[3] M. Volter, PluggableComponent - A Pattern for Interactive System Configuration, In EuroPLoP'99, 1999.

[4] D. Dig and R. Johnson, How Frameworks Learn. The Application Developer's Perspective.

[5] 田坂澄生, Eclipse プラグイン開発, JavaWORLD, 2004 3 月, pp. 36-77.

[6] E. Gamma, et al., Eclipse プラグイン開発, ソフトバンクパブリッシング, 2004.