

平成 22 年度 修士論文

プログラムの差分記述を
容易に行うための
レイヤー機構付き IDE の提案

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 09M37287

松本 久志

指導教員

千葉 滋 教授

平成 23 年 1 月 28 日

概要

ソフトウェア開発の場において、あるソフトウェアの開発を類似したソフトウェアに対する差分として行いたい場合が存在する。例としては、組み込みソフトウェアの開発やパッケージソフトウェアの複数エディションの作成などが挙げられる。このような場合差分記述を用いてソフトウェアの共通部分を流用することで、開発や保守にかかるコストの低減が期待できる。

プログラムの差分記述に利用可能な既存の支援としては言語拡張によるものがあり、その方法は差分の記述方法で2種類に大別できる。一つは元となるプログラムのソースコード中に差分を記述する方法、もう一つは差分をソースコードの外部に記述し挿入位置を指定する方法である。差分をソースコード中に記述する方法としては、C/C++で選択コンパイルを行うための`#ifdef/#endif`ディレクティブを用いる方法が挙げられる。しかしこのような記述方法には、コードの可読性の低下や、ある差分に関連するコードの抽出が困難であるといった問題点が存在する。また差分を外部ファイルに記述する言語拡張としては、アスペクト指向言語の一つであるAspectJが挙げられる。しかしこのような記述方法には、実行順序の曖昧性や、挿入位置の記述が困難であるといった問題点が存在する。

そこで本研究ではプログラムの差分記述を行うための支援として、レイヤー機構付きIDEを提案する。レイヤー機構付きIDEでは、プリプロセスディレクティブによって注釈されたコード断片を、IDE上でレイヤーとして管理する。レイヤーは、編集時の表示・非表示やコンパイル時の適用・非適用を個別に選択可能である。また選択したレイヤーの背景色を着色することで、レイヤーに属するコード断片を視覚的に抽出可能である。プログラムの差分を一つのレイヤーとしてまとめることで、差分に関するコードの抽出や一時的な非表示化が可能であり、プログラムの差分記述が容易になると考えられる。レイヤー機構付きIDEでの差分記述は、元となるプログラムのソースコード中に差分を記述する方法に分類される。しかし、IDEのエディター上にはプリプロセス処理を行った後のソースコードが表示されるため、可読性の低下は起こらない。

本研究ではレイヤー機構付きIDEの実装例であるLayerIDEの開発を、Eclipseプラグインとして行った。LayerIDEでは対象言語としてJavaを、

プリプロセスディレクティブとして`#ifdef/#endif`を選択した。LayerIDEを用いることにより、Eclipse上でJava言語を用いた差分記述をレイヤー機構を用いて行うことが可能である。また本研究では実際にLayerIDEを用いて複数のプログラム開発を差分記述により行い、レイヤー機構付きIDEによって既存の支援の問題点が解決することを確認した。

謝辞

本研究を進めるにあたり、研究の方向性や論文の構成などご指導くださった指導教員の千葉滋教授に感謝致します。

また、論文のスタイルファイルを作成して頂いた九州工業大学の光来健一准教授、共に研究に励んだ同研究室の皆様感謝致します。

目次

第 1 章	はじめに	9
第 2 章	従来手法の問題点と関連研究	11
2.1	元のファイル中に直接差分を記述するアプローチ	11
2.1.1	if 文を用いた差分記述	11
2.1.2	#ifdef/#endif ディレクティブを用いた差分記述	13
2.2	外部ファイルに差分を記述するアプローチ	14
2.2.1	AspectJ を用いた差分記述	15
2.3	関連研究	18
2.3.1	CIDE: Virtual Separation of Concerns	18
第 3 章	提案	20
3.1	レイヤー機構付き IDE	20
3.1.1	レイヤーとは	20
3.1.2	レイヤーに対する操作	23
3.1.3	IDE 内部の処理	25
3.2	既存の手法の問題点に対する考察	27
3.2.1	元のファイル中に直接差分を記述するアプローチの問題点に対する考察	27
3.2.2	外部ファイルに差分を記述するアプローチの問題点に対する考察	28
3.2.3	CIDE の問題点に対する考察	29
第 4 章	LayerIDE	30
4.1	概要	30
4.1.1	ビューの機能	31
4.1.2	エディターの機能	32
4.1.3	ビルダーの機能	32
4.2	3 章の設計との差異	33
第 5 章	実装	35
5.1	Eclipse 拡張のための基本知識	35
5.1.1	Eclipse のアーキテクチャ	35

5.1.2	マニフェスト・ファイル	36
5.1.3	マニフェスト・エディター	37
5.2	Eclipse の拡張方法	39
5.2.1	ソースコードを直接書き換える方法	39
5.2.2	似たプラグインの拡張ポイントを利用する方法	40
5.2.3	似たプラグインの API を利用して新たなプラグインを作成する方法	41
5.2.4	LayerIDE の実装方法	42
5.3	実装	43
5.3.1	ビューの実装	43
5.3.2	エディターの実装	46
5.3.3	ビルダーの実装	48
第 6 章	評価とまとめ	51
6.1	評価	51
6.2	まとめ	53

目 次

2.1	if文を用いた差分記述例	12
2.2	再コンパイルが必要な例	12
2.3	#ifdef/#endif ディレクティブを用いた差分記述例	13
2.4	AspectJを用いたプログラム例	16
2.5	GluonJを用いて記述した Debug アスペクト	17
2.6	意図した織り込みが行われない例	18
2.7	意図した織り込みが行われないアスペクト	18
2.8	CIDE	19
3.1	レイヤー分けを行ったソースコード	21
3.2	レイヤー分けを行ったソースコードのファイル中での表現	22
3.3	IDE 上での表示	22
3.4	レイヤーの選択	24
3.5	選択したレイヤーの着色	25
3.6	既存の IDE 内部の処理	26
3.7	レイヤー機構付き IDE 内部の処理	26
4.1	LayerIDE	30
4.2	LayerIDE のビュー	31
4.3	LayerIDE のエディター	32
4.4	JDT 内部の処理	33
4.5	LayerIDE 内部の処理	34
5.1	Eclipse のアーキテクチャ	36
5.2	plugin.xml の記述例	37
5.3	MANIFEST.MF の記述例	37
5.4	マニフェスト・エディター	38
5.5	ソースコードを直接書き換える方法	40
5.6	似たプラグインの拡張ポイントを利用する方法	41
5.7	似たプラグインの API を利用して新たなプラグインを作成 する方法	42
5.8	エディターのリストの取得処理の実装	44

5.9	ワークスペースのフルビルド	44
5.10	ツールバー及びコンテキストメニューへのアクションの追加	45
5.11	エディターのリストの取得処理の実装	45
5.12	レイヤーを折りたたむためのマーカー設置処理の実装	47
5.13	レイヤーの折りたたみ処理の実装	48
5.14	ソースファイルの変換処理の実装	50
6.1	シミュレーターの差分構成	52

表 目 次

2.1	AspectJ の主な用語	15
5.1	マニフェスト・エディター	38
5.2	view 拡張ポイントの category タグの主な属性	43
5.3	view 拡張ポイントの view タグの主な属性	43
5.4	レイヤー状態の保存・復元機構のためのメソッド	46
5.5	editors 拡張ポイントの editor タグの主な属性	47
5.6	ネーチャーの実装クラス	49
6.1	シミュレーターのクラス構成	52
6.2	シミュレーターのコード行数	52

第1章 はじめに

ソフトウェア開発の場において、類似した複数のソフトウェア開発が必要な場合が存在する。例としては、組み込みソフトウェアの開発やパッケージソフトウェアの複数エディションの作成などが挙げられる。このような場合、複数のソフトウェアをそれぞれ別々に開発を行うことは開発や保守にかかるコストが大きい。

類似した複数のソフトウェアの開発を行うための手法として、プログラムの差分記述が挙げられる。これは、あるソフトウェアの開発を、類似したソフトウェアに対する差分として行うものである。差分記述を用いてソフトウェアの共通部分を流用することで、開発コストの低減が可能である。またより身近な例としては、プログラムのテスト中にログコードの出力を行いたい、テスト終了後にはログ出力のコードを除去したい、といった場合にも差分記述は有効である。

このようなプログラムの差分記述は、主にプログラミング言語の拡張によって支援が行われており、そのアプローチは差分をどこに記述するかという点で大きく分けて2種類存在する。一つは元となるソフトウェアのソースコード中に直接記述するアプローチ、もう一つは差分をソースコードの外部に記述し挿入する位置を指定するアプローチである。

差分をソースコード中に直接記述するアプローチとしては、`#ifdef/#endif`ディレクティブなどが挙げられる。これは主にC/C++で使用されているプリプロセッサ命令のうちの一つであり、条件コンパイルに用いられる。ある差分に関連するコードを一つのシンボルに割り当てることで、`#ifdef/#endif`ディレクティブを用いてプログラムの差分記述を行うことが可能である。

しかし、差分をソースコード中に直接記述するアプローチは、一つのソースコード中に複数の差分に関するコードが混在するため可読性が低下するという問題点がある。さらに、ソースコード全体からある差分に関連するコードのみを抽出することも困難である。

また差分をソースコードの外部に記述するアプローチとしては、アスペクト指向言語が挙げられる。アスペクト指向言語の一つであるAspectJでは、オブジェクト指向言語ではうまくモジュール化できない横断関心事をアスペクトと呼ばれるモジュールに記述する。プログラムの差分は横断的関心事と見なすことができるため、AspectJを用いてプログラムの差分記

述を行うことが可能である。

しかし、差分をソースコードの外部に記述するアプローチは、差分の挿入位置の指定が困難であるという問題点がある。AspectJではメソッドの呼び出し時などをジョインポイント(差分の挿入位置)として指定可能であるが、ソースコード中の任意の行などをジョインポイントとして指定することはできず、差分記述の支援としては不十分である。また、AspectJはJavaの言語拡張によりアスペクト指向を実現しているため言語構造が複雑であり、コンパイラやIDEの実装も困難であるという問題点も存在する。

そこで本研究ではプログラムの差分記述を行うための支援として、レイヤー機構付きIDEを提案する。レイヤー機構付きIDEでは、プリプロセスディレクティブによって注釈されたコード断片を、IDE上でレイヤーとして管理する。レイヤーは、編集時の表示・非表示やコンパイル時の適用・非適用を個別に選択可能である。また選択したレイヤーの背景色を着色することで、レイヤーに属するコード断片を視覚的に抽出可能である。プログラムの差分を一つのレイヤーとしてまとめることで、差分に関するコードの抽出や一時的な非表示化が可能であり、プログラムの差分記述が容易になると考えられる。

本稿の残りは、次のような構成からなっている。第2章は既存の手法とその問題点について述べる。第3章では本研究の提案について、第4章では本研究の提案に基づいて実装を行ったLayerIDEについて、第5章では本研究の実装であるLayerIDEの実装方法について、第6章では本研究の評価とまとめについて述べる。

第2章 従来手法の問題点と関連研究

本章では、プログラムの差分記述を行うための従来の支援と、その問題点について記述する。最も一般的な支援は言語拡張によるものであり、そのアプローチは大きく次の2種類に分けられる。

- 元のファイル中に直接差分を記述するアプローチ
- 外部ファイルに差分を記述するアプローチ

本章ではこの2種類のアプローチについて2.1節と2.2節でそれぞれの例と問題点を挙げる。また、2.3節では関連研究のCIDE[7]について採りあげる。

2.1 元のファイル中に直接差分を記述するアプローチ

本節では、プログラムの差分記述を元のソースファイル中に直接行うアプローチについて扱う。ここでは例として、if文、及びC/C++のプリプロセッサ命令である`#ifdef/#endif`ディレクティブを用いた差分記述の方法と、その問題点を挙げる。

2.1.1 if文を用いた差分記述

差分記述を元のファイルに直接記述する方法のうち、最も原始的な手法はif文を用いたものである。図2.1はJava言語において、if文を用いて差分記述を行った例である。

図2.1では、デバッグ時のみ`run()`メソッドの呼び出しの前後でログメッセージを出力するという差分記述を、`final`変数とif文を用いて行っている。Java言語には`#ifdef/#endif`ディレクティブが存在しないため、JDT(Java Development Tools)のような大規模なプロジェクトでもこの手法が利用されている。

```
1 class Hoge {
2     static final boolean DEBUG = true;
3
4     static void main(String[] args) {
5         if(DEBUG) System.out.println("----start----");
6         new Hoge().run();
7         if(DEBUG) System.out.println("----end----");
8     }
9     :
10 }
```

図 2.1: if 文を用いた差分記述例

問題点

#ifdef/#endif ディレクティブと共通の問題点は 2.1.2 小節で述べるとして、ここでは if 文を用いた差分記述に特有の問題のみを挙げる。

まず、if 文を用いた差分記述の場合、通常の if 文との区別がつかないという問題点が挙げられる。2.1 の例では、5 行目及び 7 行目の if 文が、通常の if 文と差分記述のどちらの意味で用いられているかの判別がこの行だけでは不可能である。DEBUG という大文字変数名を用いていることで意図の読み取りは不可能ではないが、大文字変数名は定数全般に用いられるものであり、差分記述に限った記法ではない。そのため可読性を下げる結果になっていると考えられる。

また他にも、Java のような部分コンパイルが可能な言語では、final 変数に変更された際に、その変数を参照している他のクラスを明示的に再コンパイルする必要がある問題点がある。図 2.2 はそのような例である。

```
1 class Hoge {
2     static final boolean DEBUG = true;
3
4     static void main(String[] args) {
5         new Piyo().run();
6     }
7 }
8
9 class Piyo {
10    void run() {
11        if(Hoge.DEBUG) System.out.println("----DEBUG_MODE----");
12        else System.out.println("----RELEASE_MODE----");
13        :
14    }
15 }
```

図 2.2: 再コンパイルが必要な例

図 2.2 の 2 つのクラスをコンパイルして実行すると、コンソールに

—DEBUG MODE—

と正常に出力される。しかしその後 Hoge.DEBUG 変数を false に変更し、Hoge クラスのみを再コンパイルした場合、Piyo クラスにはその変更は反映されないため、実行結果は

—DEBUG MODE—

となる。IDE の一つである Eclipse[15] を用いてコンパイルを行った場合は Eclipse が自動的に Piyo クラスを再コンパイルするためこのようなことは起こらないが、javac コマンドを用いて Hoge クラスのコンパイルを行った場合は Piyo クラスの再コンパイルは行われず、バグの温床となることが予想される。

前述の JDIT のソースコード中では、クラスごとに DEBUG 変数を用意することでこの問題を避けている。しかしその場合、複数のクラスを一括してデバッグモードに切り替えるといった操作が不可能であるという問題点がある。

2.1.2 #ifdef/#endif ディレクティブを用いた差分記述

#ifdef/#endif ディレクティブは、主に C/C++ で用いられているプリプロセッサ命令のうちの2つである。プリプロセッサとは、プログラムをコンパイルする前に簡単な変換処理を行うためのプログラムのことで、「前処理系」とも呼ばれる。プリプロセッサに対する指示の記述をディレクティブと呼ぶ。#ifdef/#endif はそのうちの2つであり、#define ディレクティブと共に用いることで条件コンパイルが可能である。図 2.3 は、#ifdef/#endif ディレクティブを用いて差分記述を行った例である。

```
1 #include <stdio.h>
2 // #define DEBUG
3
4 int main(void) {
5
6 #ifdef DEBUG
7     printf("----DEBUG.MODE----");
8 #endif
9     :
10    return 0;
11 }
```

図 2.3: #ifdef/#endif ディレクティブを用いた差分記述例

図 2.3 では、6 行目と 8 行目の #ifdef/#endif ディレクティブで条件分岐を行っている。#ifdef/#endif で挟まれた行は、ソースコード中で #define

ディレクティブにより同名のシンボルが定義されていた場合のみコンパイル時に有効化される。図 2.3 の例では、コンパイル時にソースコード中で `#define DEBUG` が定義されている場合のみ、

```
—DEBUG MODE—
```

と出力される。

問題点

`#ifdef/#endif` ディレクティブによる差分記述の問題点の1つには、可読性の低下が挙げられる。C/C++で記述されたコードの中に、構造の異なるプリプロセッサのためのディレクティブが混入することにより、ソースコードの可読性の低下が起こると考えられる。図 2.3 ではインデントにより可読性の向上を図っているが、複数の `#ifdef/#endif` 領域が存在する場合や、`#ifdef/#endif` 領域が肥大化した場合、可読性の低下は免れない。

また、`#ifdef/#endif` ディレクティブによる差分記述では、差分の選択を変更するために、`#define` ディレクティブを用いる必要がある。そのためソースコードに微少なながら変更を加える必要があるという問題点が存在する。

さらに、1つの差分に関連するコードの抽出も困難である。1つの差分に関連するコードは、ある種のモジュールとして統一的に扱えることが望ましいが、1つのシンボルに関連するコードのみを抽出したり、全てを非表示にしたりといった機構は存在しない。

Visual C++[9] に付属するアウトライン機能を利用してソースコードを折りたたむことで、可読性の問題の一部は解決する。しかしその場合でも、同名のシンボルに関連するコードを統一的に扱うことはできず、同時に折りたたんだり元に戻したりといったことは不可能であり、差分記述のための支援としては不十分であると考えられる。さらに Visual C++ のアウトライン機能には、折りたたみ領域が正常に認識されなかったり、折りたたんだ際のメッセージに変更が反映されなかったりといったバグも存在する。また、Eclipse の C/C++ 開発環境である CDT(C/C++ Development Tooling)[13] では、このような折りたたみ機能は提供されていない。

2.2 外部ファイルに差分を記述するアプローチ

外部ファイルに差分を記述するアプローチとしては、アスペクト指向言語が挙げられる。アスペクト指向とは、従来のオブジェクト指向ではモ

ジュール間にまたがってしまう処理(横断的関心事)をアスペクトと呼ばれるモジュールにまとめることによりうまくモジュール化するための技術である。ここでは例として、アスペクト指向言語の一つである AspectJ[8, 12]を用いた差分記述と、その問題点について述べる。

2.2.1 AspectJ を用いた差分記述

AspectJ は、Java 言語にアスペクト指向プログラミングを行うための機構を追加した言語実装の一つである。表 2.1 は本小節を読むにあたって必要であると考えられる、AspectJ の主な用語である。

用語	意味
アスペクト	横断的関心事をまとめたモジュール単位。ポイントカットとアドバイスを組み合わせることで他のモジュールの動作を変更可能。
ポイントカット	ジョインポイントの集合。アスペクトを織り込みたいポイントを条件で指定する。
ジョインポイント	アスペクトによって処理を追加・変更するための基点。ソースコード上での位置ではなく、プログラム実行時のタイミングを意味する。代表的なものとして、メソッドやコンストラクタを呼び出した時や、フィールドの参照時や代入を行った時などがある。
アドバイス	ポイントカットに対し、追加・変更したい処理を記述したもの。
織り込み (weave)	ポイントカットで指定されたジョインポイントの集合でアドバイスが実行されるように結び付けること。

表 2.1: AspectJ の主な用語

プログラムの差分は、複数のモジュール間にまたがる処理であるため、アスペクトとみなすことができる。その観点に基づき AspectJ を用いて差分記述を行った例が図 2.4 である。

図 2.4 では、Debug アスペクトが差分にあたる。Debug アスペクトでは Hoge.run() メソッドを呼び出した時をポイントカットとして定義し、before() アドバイスと after() アドバイスにより Hoge.run() メソッドの呼び出しの前後にログメッセージを出力するように記述されている。Debug アスペクトを織り込んで Hoge クラスを実行すると、


```
1 public class Hoge {
2     public static void main(String[] args){
3         new Hoge.run();
4     }
5
6     private void run() {
7         System.out.println("run_method");
8     }
9 }
10
11 public aspect Debug {
12     pointcut print(): execution(void Hoge.run());
13
14     before(): print() {
15         System.out.println("[Before_run()_method]");
16     }
17
18     after(): print() {
19         System.out.println("[After_run()_method]");
20     }
21 }
```

図 2.4: AspectJ を用いたプログラム例

```
[Before run() method]
run method
[After run() method]
```

と出力される。

問題点

外部ファイルを用いた差分記述の問題点としては、任意の差分の記述が困難であるという点が挙げられる。AspectJ では、差分の記述はジョインポイントを介してのみしか行うことができない。そのためメソッド単位で変更を行うような差分の記述は可能だが、あるメソッドの何行目と何行目の間にこのコードを挿入したい、といった差分の記述は不可能である。また、同一のジョインポイントに対して複数のアドバイスの織り込みが行われた場合、適用順序が保障されないという問題点も存在する。

さらに、外部ファイルによって処理が織り込まれるクラス側からは、どのような処理が織り込まれるのか分からないという問題点も存在する。これはアスペクト指向言語の利点でもあるが、差分記述のアプローチとして見た場合、処理全体の流れが分からなくなってしまうため欠点であると考えられる。AspectJ では、AJDT[5, 11] のような IDE を利用することによりクラス側からどの位置にアドバイスが織り込まれるを知ることは可能

であるが、そのアドバイスによってどのような処理が織り込まれるかを知るためには実際にアスペクトの記述を参照する必要がある。

また AspectJ は、Java 言語を拡張することによりアスペクト指向プログラミングを行うための機構を実現している。そのため Java 言語に比べて習得が困難であるという問題点や、コンパイラや IDE の作成や拡張が困難であるという問題点がある。

他のアスペクト指向言語の実装例である GluonJ[3, 4] では、Java 言語に存在するアノテーションの機構を利用することで、言語拡張を行うことなくアスペクト指向の機構を実現している。図 2.4 の Debug アスペクトを GluonJ を用いて記述すると、図 2.5 のようになる。

GluonJ は言語拡張を伴わないため、コンパイラや IDE の拡張を行うことなく利用可能である。しかし、メソッドやポイントカットの指定は文字列として行われているため、IDE での編集は可能だがクラス名の検査などは行われない。また、Java のアノテーションや C# の属性にあたる機構の存在しない言語への応用は依然として困難である。

```
1 import javassist.gluonj.*;
2
3 @Glue public class Debug {
4     @Before("{_System.out.println('[Before_run()_method]');;-}")
5     @After("{_System.out.println('[After_run()_method]');;-}")
6     Pointcut print = Pcd.call("Hoge#run()");
7 }
```

図 2.5: GluonJ を用いて記述した Debug アスペクト

また、ジョインポイントはあくまでもプログラム実行時のタイミングであり、ソースコードとは異なる構造を持つ。そのため意図した織り込みが行われない場合がある。

図 2.6 は AspectJ で書かれたソースコード例である。ここで、7 行目の `getX()` メソッドを呼ぶ直前のフィールド `x` の値をログとして出力したいと考え、図 2.7 のようなアスペクトを記述したとする。

しかし、図 2.7 のアスペクトでは意図した織り込みが行われない。図 2.7 で織り込みを行っているのは 7 行目の直前ではなく、あくまで `getX()` メソッドの呼び出しの直前である。そのため、図 2.6 の 7 行目の `if` 文中で `b` の値が `true` だった場合、以後の評価は行われないため `getX()` メソッドが呼ばれず、従って 2.7 の `printX()` も実行されない。また、この例では図 2.6 の 7 行目の `if` 文の評価順序を入れ替え、

```
if(getX()==0 || b) {
```

```
1 public class Hoge {
2     private int x;
3
4     public static void main(String[] args){
5         boolean b = true;
6         :
7         if(b || getX()==0) {
8             System.out.println("b==trueまたはx==0");
9         }
10        :
11    }
12
13    private int getX() {
14        return x;
15    }
16 }
```

図 2.6: 意図した織り込みが行われない例

```
1 public aspect Log {
2     pointcut printX(Hoge h): target(h) && execution(void Hoge.getX());
3
4     before(Hoge h): printX(h) {
5         System.out.println("x=" + h.x);
6     }
7 }
```

図 2.7: 意図した織り込みが行われないアスペクト

とすると、実行結果が変化する。if文の評価順序を変えただけで実行結果が変化することは直感に反し、バグの温床にもなると考えられる。

2.3 関連研究

2.1節、2.2節では、プログラムの差分記述を行うための言語拡張によるサポートについて述べた。本節ではFOP(Feature-oriented programming[10]のための実装として、IDEによる差分記述のサポートを提供する研究について述べる。FOPの研究としてはCIDE(Colored IDE)[6, 7]やAHEAD[1, 2]が挙げられるが、本節では本研究と同様にEclipseプラグインとして実装が行われているCIDEについて採り上げる。

2.3.1 CIDE: Virtual Separation of Concerns

CIDE(Colored IDE)は、ソフトウェアプロダクトラインのためのIDEサポートの研究の一つである。CIDEはEclipseプラグインとして提供されており、ソフトウェアのフィーチャ(特徴)に関連するコードを、背景色

に色を付けることによって分類する。この色付けはソースコードの AST に対して行われる。図 2.8 は CIDE を用いてソフトウェア開発を行っているところである。

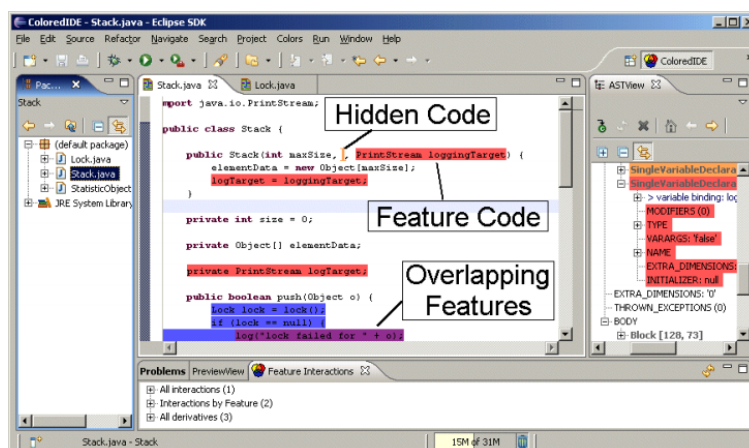


図 2.8: CIDE

プログラムの差分はフィーチャとみなすことができるため、CIDE を用いることでプログラムの差分記述を行うことが可能である。

CIDE では、フィーチャの管理のために次の機能を提供している。

- 一つのフィーチャに関連するコード間の移動 (navigation)
- 指定したフィーチャに関連したコードの非表示化 (selection)
- Jak または AspectJ 言語へのエクスポート (composition)

問題点

しかし、CIDE は AST に対してフィーチャの関連付けを行うため、完全な形の AST が必要である。そのため、全てのフィーチャの実装が実現されたソースコードに対してフィーチャごとに分類するといった開発には向いているが、ソースコードの編集では AST が不完全であるため、差分にあたるソースコードを新たに記述していくといった開発には不向きである。

また CIDE では、ソースコードの AST とフィーチャとの関連付けは、プログラマーには解読不能な形式で外部ファイルに記述される。そのため CIDE を用いて記述されたソフトウェアは、エクスポートを行わない限り CIDE 以外の開発環境で修正を行うことが困難である。

第3章 提案

本研究ではプログラムの差分記述を容易にするために、レイヤー機構付き IDE を提案する。本研究のアプローチは2章の分類では、元のファイル中に直接差分を記述するアプローチに含まれるが、プリプロセッサを通じた後のソースコードを IDE 上に表示することにより、可読性を下げることなくプログラムの差分記述を行うことが可能である。

本研究が提案するレイヤー機構付き IDE を用いることにより、編集時の差分の不可視化や差分の選択を行うための GUI の提供といった、差分記述のための支援が可能である。また、レイヤー機構を実現するための文法拡張はプリプロセッサの範囲で行うため、既存のコンパイラや IDE を比較的容易に拡張可能である。

本章ではまず 3.1 節で本研究の提案であるレイヤー機構付き IDE の概念について述べ、3.2 節では2章で述べた既存の手法の問題点に対する考察を行う。

3.1 レイヤー機構付き IDE

本節では、本研究で提案するレイヤー機構付き IDE の概念について述べる。3.1.1 小節ではレイヤーの概念について、3.1.2 小節では本研究で提案するレイヤーに対する操作について、3.1.3 小節では IDE 内部の処理と拡張方法についてそれぞれ述べる。

3.1.1 レイヤーとは

本研究で提案するレイヤーとは、差分に関連するソースコード断片を IDE 上で集めたものである。このソースコード断片とは、一つのファイル内からだけでなく、複数のファイル内の複数箇所全てを集めたものである。図 3.1 はその概念図である。

図 3.1 は Java で記述されたソースコードをベース部分、ログ出力に関するコード、同期に関するコードの3つのコード断片に分割したところである。このそれぞれのコード断片を、本研究ではレイヤーと呼ぶ。即ち図 3.1 では、ベースとなるレイヤー、ログ出力に関するレイヤー、同期に関

```

1 static void run() {
2     Hoge = new Hoge();
3     synchronized(lock) {
4         System.out.print("run");
5         h.run();
6     }
7 }

```

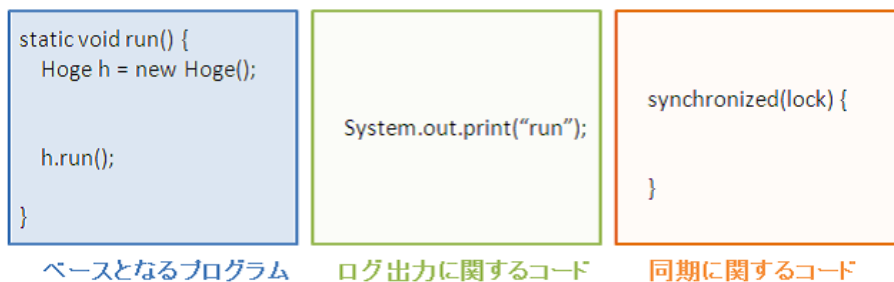


図 3.1: レイヤー分けを行ったソースコード

するレイヤーの3つのレイヤーが存在するということになる。このうちベースとなるレイヤーはプログラム中に必ず存在し、その他のレイヤーは編集時及びコンパイル時にどのレイヤーを有効にするかを IDE 上で自由に選択可能である。

レイヤーは画像処理ソフトにおいて、しばしば描画用の透明なシートと表現される。ユーザーは透明なレイヤーの上に描画を行い、複数のレイヤーを重ねることで1枚の画像を構成する。レイヤーを用いることにより、輪郭線を上書きせずに着色を行うことや、画像の一部のみを変更した差分の作成などが容易となる。

本提案のレイヤー機構付き IDE は、この概念を IDE に導入したものである。プログラマーはレイヤーを用いることにより、元となるソースコードを上書きすることなく差分の作成を行うことが可能となる。

各レイヤーの記述はプリプロセスディレクティブによって一つのファイル中に記述される。そのため、ファイルを直接テキストエディターで開いた場合には 2.1.2 小節同様に可読性の低下が起こる。しかし、IDE 上には図 3.1 のようにプリプロセス処理を行ったのちのソースコードが表示されるため、プリプロセスディレクティブによる注釈を表示する必要はなく、可読性は低下しない。

図 3.1 のソースコードを例として用いる。このソースコードは、ファイル中では例えば図 3.2 のように記述されている。

```

1 static void run() {
2     Hoge = new Hoge();
3     #ifdef SYNC
4         synchronized(lock) {
5             #endif
6             #ifdef LOG
7                 System.out.print("run");
8             #endif
9             h.run();
10            #ifdef SYNC
11            }
12            #endif
13        }

```

図 3.2: レイヤー分けを行ったソースコードのファイル中での表現

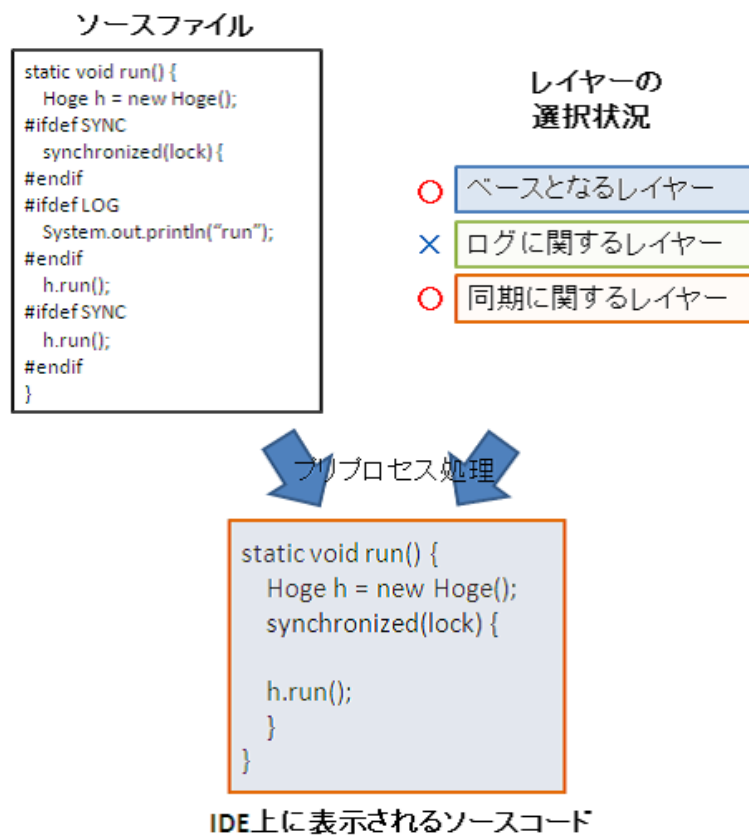


図 3.3: IDE 上での表示

図 3.2 は図 3.1 のソースコードのファイル中での表現の一例である。図 3.2 ではプリプロセスディレクティブとして 2 章で挙げた `#ifdef/#endif` ディレクティブを用いている。ここでは LOG シンボルがログ出力に関するレイヤーを表し、SYNC シンボルが同期に関するレイヤーを表している。図 3.2 では 2.1.2 小節同様に可読性の低下が起こっているが、IDE 上には図 3.3 のようにプリプロセス処理を行った後のソースコードが表示されるため、IDE での編集時には可読性は低下しない。

また、本研究に基づいて実装されたレイヤー機構付き IDE で作成されたソースコードを、IDE を用いずに一般的なテキストエディターで編集することも可能である。レイヤーに関する情報は全て一つのファイル中に記述されているため、可読性は低下するものの、図 3.2 のように記述されたファイルを直接テキストエディターで編集することも可能である。

図 3.2 ではプリプロセスディレクティブの例として `#ifdef/#endif` ディレクティブを用いたが、必ずしもこの構文を用いる必要はない。本提案のレイヤー機構付き IDE で用いられるプリプロセスディレクティブは、

- 差分記述を行う言語と別の構造を持っている
- レイヤー機構付き IDE を用いなくても編集可能

という条件を満たすものならば何を用いてもよい。`#ifdef/#endif` ディレクティブ以外の例としては、XML 規格などを利用することが考えられる。

3.1.2 レイヤーに対する操作

本提案ではレイヤー機構付き IDE がレイヤーに対して IDE 上で行える操作として次のものを考えている。

- レイヤーの作成・削除・統合
- レイヤーの選択表示・選択コンパイル
- 指定したレイヤーの着色

本小節ではそれぞれの項目について述べる。

レイヤーの作成・削除・統合

レイヤー機構付き IDE では、レイヤーを管理するために、IDE 上でレイヤーの作成・削除・統合を行うための GUI を提供する。このうち作成と

削除とは、IDE 上である差分に関連するコードを分類し、IDE 上でレイヤーとして扱うための操作である。レイヤーの作成機能により IDE 上でそのレイヤーにソースコードを記述することが可能となり、レイヤーを削除機能により IDE 上でそのレイヤーが扱う差分に関連するコードをファイル中から全て除去することが可能となる。またレイヤーの統合とは、複数のレイヤーに記述されたソースコード断片を一つのレイヤーにまとめることを表す。これは大きなプロジェクトにおいてレイヤー数が増大しすぎて管理が困難になることを避けるための機能である。

レイヤーの選択表示・コンパイル

レイヤー機構付き IDE では、編集時及びコンパイル時にどのレイヤーを適用するかを選択を IDE 上で行うための GUI を提供する。レイヤーの選択を外すことで、編集時にはそのレイヤーに記述されたコード断片を非表示にし、コンパイル時にはそのレイヤーに記述されたコード断片を除いてコンパイルを行うことが可能である。

図 3.4 は、図 3.4 で扱った 3 つのレイヤーについて、レイヤーの選択を行った例である。

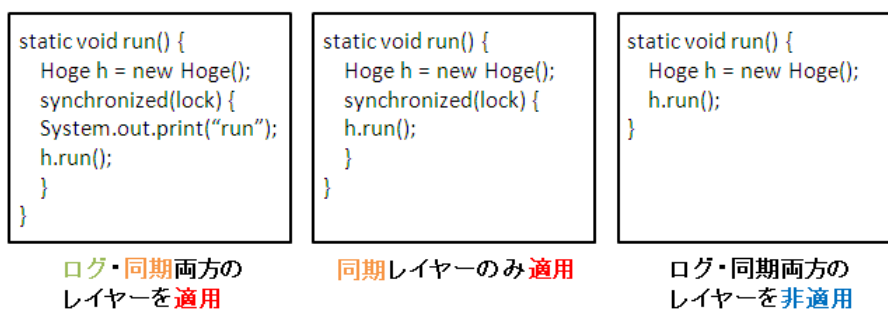


図 3.4: レイヤーの選択

ベース、ログ出力、同期の 3 つのレイヤーのうち、ベースとなるレイヤーは常に適用される。その他の 2 つのレイヤーについて、左図ではログ出力と同期の両方のレイヤーを適用し、中図では同期レイヤーのみを適用し、右図ではベースとなるレイヤーのみが適用されている。このようにレイヤー機構付き IDE では、編集時にはエディター上に選択されたレイヤーのコード断片のみが表示され、コンパイル時には選択されたレイヤーのコード断片を集めたソースコードを用いてコンパイルを行う。

#ifdef/#endif ディレクティブを用いた差分記述では編集時に差分を非表示にすることは不可能であり、コンパイル時にもソースコード中で#define ディレクティブを用いて指定する必要があった。また AspectJ を用いた差分記述では、コンパイル時に引数で織り込むアスペクトを指定する必要があった。これらの選択機能を GUI として提供することで、より直感的な差分の選択が可能であると考えている。

指定したレイヤーの着色

レイヤー機構付き IDE では、指定したレイヤーに属するコード断片の背景色を変更する機能を提供する。これは差分に関連したコードの抽出を容易にするための機能である。背景色を変更することにより、編集時の差分に関連したコードの抽出を視覚的に行うことが可能である。

図 3.5 は、図 3.1 で扱った 3 つのレイヤーをそれぞれ指定して背景色の変更を行ったものである。

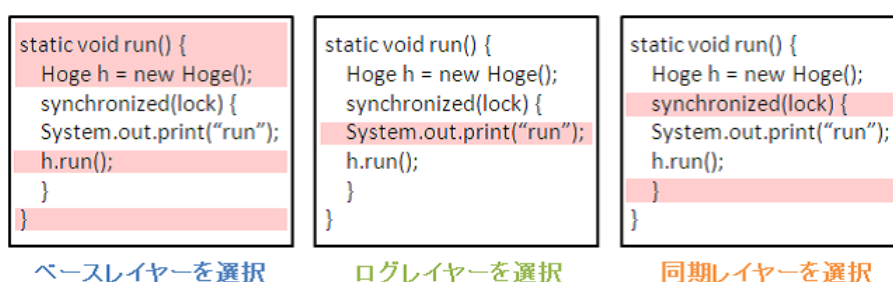


図 3.5: 選択したレイヤーの着色

3.1.3 IDE 内部の処理

図 3.6 は、既存のソースファイルを読み込んで表示及びコンパイルを行う際の IDE 内部の処理を図示したものである。既存の IDE では、ソースファイルから読み込んだソースコードを加工せずにそのままエディターやコンパイラに渡している。

一方、図 3.7 はレイヤー機構付き IDE 内部の処理を図示したものである。

レイヤー機構付き IDE では図 3.7 のように、ファイルから読み込んだソースコードとレイヤーの選択情報をプリプロセッサに入力し、ソースコードの変換を行う。その後変換したソースコードをエディターやコンパイラに渡し、表示及びコンパイルを行う。

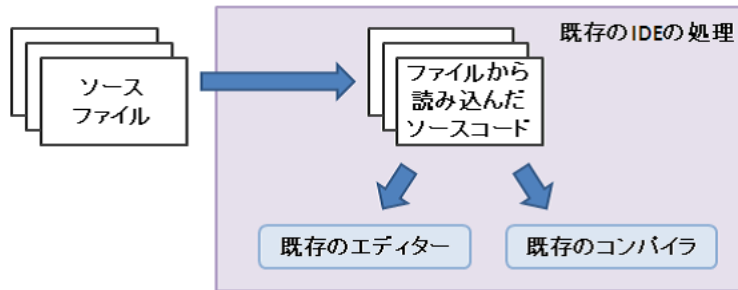


図 3.6: 既存の IDE 内部の処理

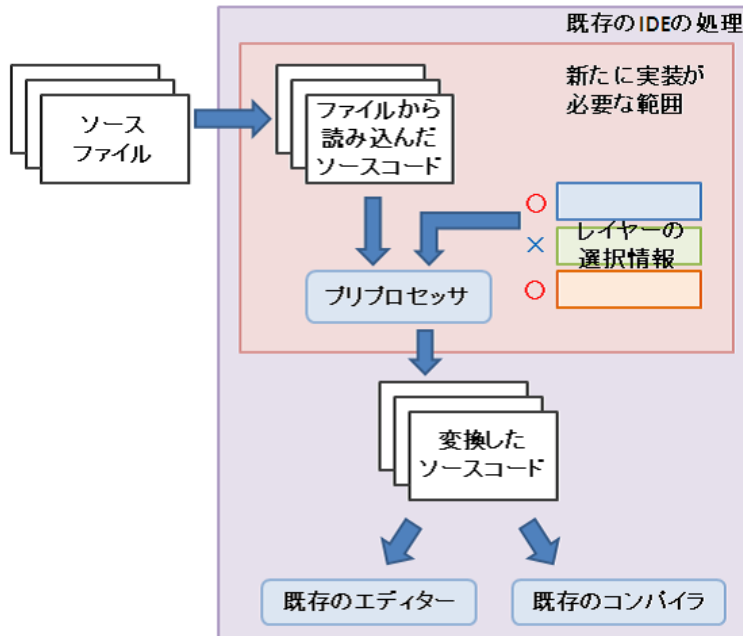


図 3.7: レイヤー機構付き IDE 内部の処理

既存の IDE を元にしてレイヤー機構付き IDE の実装を行う場合、図 3.7 のように、レイヤーの選択を行う機構とプリプロセス処理を行う機構を実装すればよい。この際ファイルの読み込みや、エディターやコンパイラにあたる部分は既存のものを流用することが可能であり、実装が比較的容易であると考えられる。

また、レイヤー機構付き IDE で扱うプリプロセスディレクティブは言語構造と独立であり、プリプロセッサもコンパイラと独立である。そのた

め、プリプロセッサ部分を他の言語の IDE へ移植することが比較的容易であると考えられる。さらに、プリプロセスディレクティブを複数の言語で統一することで、サーバーサイドとクライアントサイドで実装言語が異なるといった一つのプロジェクト内で複数の言語を扱っている場合においても、プロジェクト全体の差分記述が容易になるのではないかと考えている。

3.2 既存の手法の問題点に対する考察

本小節では2章で挙げた既存の手法のそれぞれの問題点が、本提案のレイヤー機構付き IDE を用いることでどのように改善されるかについて考察を行う。

3.2.1 元のファイル中に直接差分を記述するアプローチの問題点に対する考察

元のファイル中に直接差分を記述するアプローチとして2.1節では、if文によるものと`#ifdef/#endif`ディレクティブによるものを探り上げた。このうちif文を用いたアプローチの問題点については、本提案のレイヤー機構付き IDE が`#ifdef/#endif`ディレクティブを用いた差分記述の機能を包含しているため、問題とならない。

`#ifdef/#endif`ディレクティブによる差分記述の問題点としては次のようなものが存在した。

- プリプロセスディレクティブの混入による可読性の低下
- `#define`ディレクティブによるソース変更の必要性
- 1つの差分に関連するコードの抽出が困難

レイヤー機構付き IDE ではプリプロセス処理を行った後のソースコードを IDE 上に表示するため、可読性は低下しない。さらに、レイヤーを選択して非表示にすることで行っている編集に必要なないコード断片を除去することが可能である。

また、レイヤー機構付き IDE ではレイヤーの選択情報を IDE 上で保持するため、ソースコードに変更を加えることなく適用する差分を選択することが可能である。

差分に関連するコードの抽出が困難であるという問題については、改善のためにレイヤーの着色機能を用意した。編集時のレイヤーの背景色を変

更することにより、差分に関連したコードの抽出を視覚的に行うことが可能である。

3.2.2 外部ファイルに差分を記述するアプローチの問題点に対する考察

外部ファイルに差分を記述するアプローチとして2.2節では、アスペクト指向言語である AspectJ[8] や GluonJ[3] を採り上げた。アスペクト指向言語にによる差分記述の問題点としては次のようなものが存在した。

- ジョインポイントが限られるため任意の差分の記述が困難
- 複数のアスペクトの適用順序が保障されない
- 一つの処理のコードが複数のクラスやアスペクトに分散
- 言語構造が複雑なため習得が困難
- 言語拡張を伴うためコンパイラや IDE の作成・拡張が困難
- 言語構造の差異による意図しない織り込み

レイヤー機構付き IDE では、差分の記述はプリプロセスディレクティブにより行われるため、文字単位での任意の差分の記述が可能である。また、全てのレイヤーのソースコード断片は一つのファイル中にまとめられているため適用順序の曖昧性は存在せず、コードの分散も起こらない。

レイヤー機構付き IDE では、IDE 上ではプリプロセス処理を行ったのちのソースコードを編集するため、新しく構文を覚える必要はない。また IDE を利用せずに編集を行う場合でも、プリプロセスディレクティブとして `#ifdef/#endif` のような既に広く利用されている構文を用いることにより、新たな構文を覚えることなく編集可能である。

またレイヤー機構付き IDE の実装は、既存の IDE を拡張することにより行う。この際 IDE 内で動作するプリプロセッサの実装のみを行えば良いため、比較的容易に実装可能であると考えられる。

レイヤー機構付き IDE は言語構造が混在しないため、言語構造の差異による意図しない織り込みは起こらない。また複数のレイヤーの適用によって起こり得る意図しない実行順序については、レイヤーの適用状態を切り替えて視覚的にソースコードを確認することで回避が可能である。

3.2.3 CIDE の問題点に対する考察

CIDE では AST に対してフィーチャの関連付けを行うため完全な形の AST が必要であり、新たに差分を記述していく場合には不向きであるという問題点が存在した。レイヤー機構付き IDE では文字単位でのレイヤー分けが可能であるため、新たなレイヤーを作成してソースコードを記述していくことで、差分を新たに記述していくことが容易である。

また CIDE では関連付けを行うファイルが外部にあり、プログラマーには解読不能な形式で記述されているため、CIDE で開発したソフトウェアは CIDE 上でしか開発を行えないという問題点が存在した。レイヤー機構付き IDE では `#ifdef/#endif` ディレクティブのような、プログラマーが IDE を用いずに理解可能なプリプロセッサディレクティブを用いることにより、レイヤー機構付き IDE で開発したソフトウェアを IDE を用いずに編集することが可能である。

第4章 LayerIDE

本研究では3章の設計に基づいたレイヤー機構付き IDE、LayerIDE の実装を Eclipse プラグインとして行った。レイヤー機構を実現する対象言語は Java とし、プリプロセッサディレクティブとしては `#ifdef/#endif` を使用した。本章ではその概要について述べる。4.1 節では LayerIDE の概要について、4.2 節では3章の設計との差異について述べる。

4.1 概要

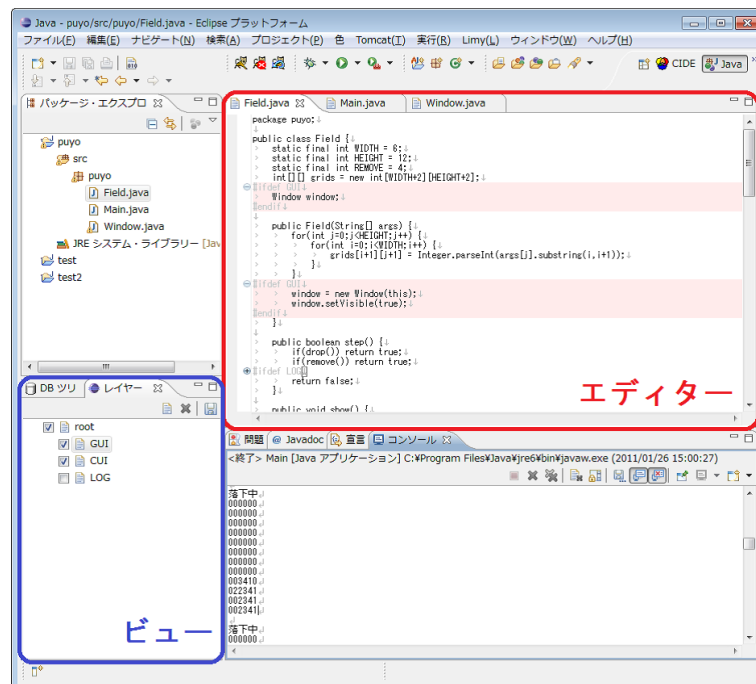


図 4.1: LayerIDE

本研究では LayerIDE の実装を、Eclipse の Java 開発環境である JDT (Java Development Tools)[14] を Eclipse プラグインにより拡張することで行った。JDT の GUI は、ソースコードの編集を行うためのエディターと、い

くつかのビュー、およびメニューバーなどからなる。LayerIDEは図4.1で表すエディターとビュー、及びコンパイルを行うためのビルダーをJDTに追加することによって、3章で提案したレイヤー機構付きIDEを実現した。

本節ではLayerIDEの提供するビュー、エディター、ビルダーの機能について述べる。

4.1.1 ビューの機能

LayerIDEではレイヤーの管理を行うために、図4.2のようなビューを提供する。LayerIDEの提供するビューでは以下のような操作が可能である。

- レイヤーの作成・削除・統合
- レイヤーの適用化・非適用化
- 着色するレイヤーの選択
- 現在のレイヤー構成でビルド
- 前回終了時のレイヤー状態の復元

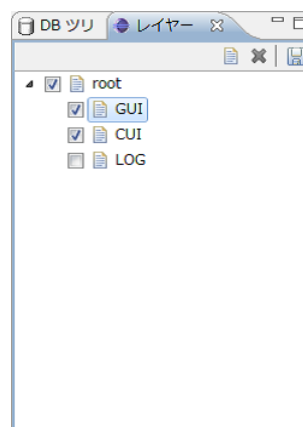
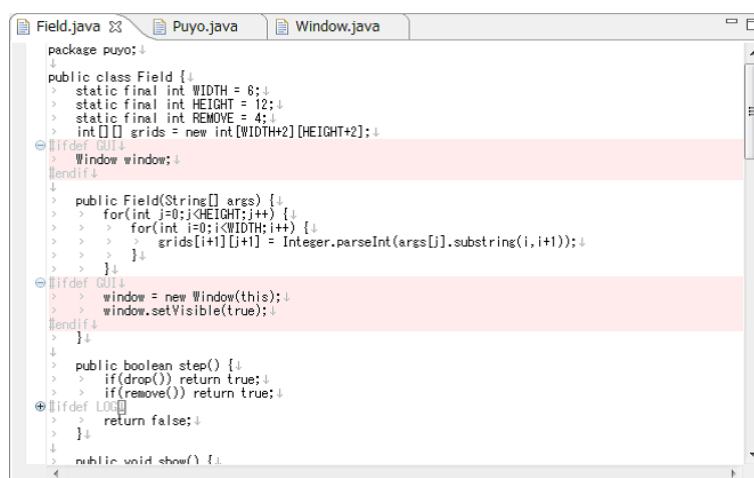


図4.2では中央にツリー状にroot、GUI、CUI、LOGの4つのレイヤーが表示されている。このうちrootはベースとなるレイヤーを表し、編集時及びコンパイル時には常に適用される。

レイヤーの左側のチェックボックスはレイヤーの適用状態を表す。図4.2ではroot、GUI、CUIの3つのレイヤーが適用状態であり、LOGのレイヤーは非適用状態である。

またレイヤー名をクリックすることで、エディター上で背景色を変更するレイヤーを選択することが可能である。図4.2ではGUIレイヤーに属するコード断片がエディター上で着色されている。

その他の操作は、右上のメニュー及びコンテキストメニューから行う。LayerIDEではこれらのビュー上の操作に従ってプリプロセス処理を行い、次小節で説明を行うエディター上に表示されるソースコードを生成する。



```
package puyo;
public class Field {
    static final int WIDTH = 6;
    static final int HEIGHT = 12;
    static final int REMOVE = 4;
    int[] grids = new int[WIDTH*2][HEIGHT*2];
#ifdef GUI
    Window window;
#endif
    public Field(String[] args) {
        for(int j=0;j<HEIGHT;j++) {
            for(int i=0;i<WIDTH;i++) {
                grids[i+1][j+1] = Integer.parseInt(args[j].substring(i,i+1));
            }
        }
    }
#ifdef GUI
    window = new Window(this);
    window.setVisible(true);
#endif
}
public boolean step() {
    if(drop()) return true;
    if(remove()) return true;
#ifdef LOG
    return false;
#endif
}
public void show() {
```

図 4.3: LayerIDE のエディター

4.1.2 エディターの機能

LayerIDE ではソースコードの編集を行うために、図 4.3 のようなテキストエディターを提供する。LayerIDE ではソースコードの非表示化を、Eclipse のフォールディング機能 (ソースの折りたたみ機能) を用いて表現した。これによりソースコードの表示・非表示の切り替えを感覚的に行うことが可能である。

また LayerIDE のエディターでは、`#ifdef/#endif` ディレクティブの記述されているコード行は灰色で表示する。プログラムに本質的に関係のないプリプロセスディレクティブ行を灰色で表示することにより、プリプロセスディレクティブの挿入により起こる可読性の低下の問題を緩和した。

図 4.3 では、図 4.2 のレイヤー状態に従ったソースコードが表示されている。GUI レイヤーに属するコード断片は背景色が着色され、LOG レイヤーに属するコード断片は折りたたまれ非表示になっている。その際 1 行目の `#ifdef` ディレクティブが残っているため、レイヤーが非表示状態であってもそこにレイヤーによるコードが存在することが確認可能である。

4.1.3 ビルダの機能

LayerIDE では `#ifdef/#endif` ディレクティブで注釈されたソースコードをコンパイルするためのビルダーを提供する。ビルダーは GUI を持たず、4.1.1 小節で述べたビュー上のレイヤー状態に従い、プリプロセスディレクティブを解釈し、コンパイルを行う。

ビルダーによるコンパイルはデフォルトでは、各ソースファイルの保存時、及びレイヤーの適用状態の変更時に自動的に行われる。またビュー上のボタンにより手動でコンパイルを行うことも可能である。

4.2 3章の設計との差異

本実装は Eclipse プラグインとして行ったことが原因で、3章の設計通りに実装を行うことができなかった部分が存在する。本節では3章の設計と LayerIDE の設計での差異について述べる。

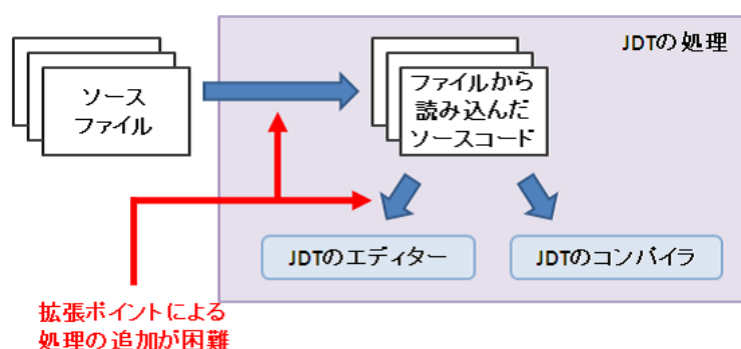


図 4.4: JDT 内部の処理

Eclipse プラグインを用いて Eclipse の拡張を行う場合、5.2 節で述べるように、特定の拡張ポイントに処理を追加することで拡張を行う。しかし拡張を行う元となった JDT の実装では図 4.4 で表すように、ファイルの読み込み際や読み込んだソースコードをエディターに表示する際に、拡張ポイントを用いてプリプロセス処理を挿入することは困難であった。そのため LayerIDE は、図 4.5 のような設計に基づいて実装を行った。

3章の設計では図 3.7 のように、ソースファイルを読み込む際にプリプロセス処理を行い、変換したソースコードをエディター及びコンパイラに渡すことを想定していた。しかし、JDT の実装ではソースファイルの読み込み時にプリプロセス処理を行い、なおかつ元のソースコードを保持しておくような実装をプラグインによって行うことは困難であった。これは JDT が編集ファイルの同期などのためにファイルシステムと独立した Java モデルを持っており、様々な箇所からそれを参照していたためである。そのため LayerIDE では、ソースファイルの読み込み後、エディターやコンパイラにソースコードを渡す前にプリプロセス処理を行うことにより実装を行った。

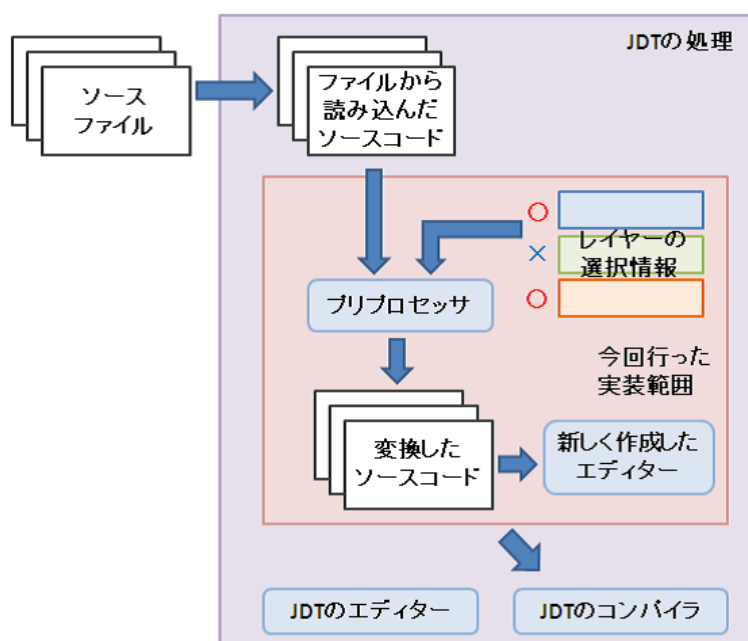


図 4.5: LayerIDE 内部の処理

また Eclipse 及び JDT の実装では、ソースファイルを読み込んでから JDТのエディターへ表示するまでの間に、プリプロセス処理を挿入することは困難であった。これは JDТのエディターが今回の拡張に適切な拡張ポイントを提供しておらず、またエディターが Singleton パターンのために static メソッドを多用する実装であったため、エディタークラスの継承による拡張も困難であったためである。そこで LayerIDE の実装では、JDТのテキストエディターを使用せず、プリプロセス処理を行った後に画面への表示を行う新たなエディターを作成することで、レイヤー機構のついたソースコードエディターを実現した。

これらの理由により LayerIDE の実装では、ソースの装飾やエラー報告といった JDТのエディターの機能をそのまま流用することが不可能であった。しかし、JDТのソースコードを直接変更する実装を行う場合はこのような問題はなく、JDТのエディターの機能をそのまま利用することが可能である。また JDТの実装に一部手を加えることが可能であるならば、プリプロセス処理を行うための拡張ポイントを追加することにより、Eclipse プラグインによるレイヤー機構付き IDE の実装も可能であると考えている。

第5章 実装

本章では、4章で述べた LayerIDE の実装方法について述べる。LayerIDE の実装は、Eclipse[15] プラグインとして行った。

5.1 節では Eclipse を拡張する上で必要な知識について、5.2 節では Eclipse を拡張するためのいくつかの方法について、5.3 節では LayerIDE の実装について述べる。

5.1 Eclipse 拡張のための基本知識

Eclipse は Java を用いて開発された、オープンソースのプラットフォームである。本節では Eclipse の拡張を行うために必要な知識を、本論文を読むのに必要と考えられる範囲で記載する。5.1.1 小節では Eclipse のアーキテクチャについて、5.1.2 小節では Eclipse プラグインの構成とその作成方法について記述する。

5.1.1 Eclipse のアーキテクチャ

Eclipse は Java 向けの IDE として有名であるが、Java の IDE としての機能は Eclipse SDK に標準で含まれる JDT(Java Development Tools)[14] によって提供されているものであり、本来的には様々な開発ツールのための統合プラットフォームを提供することを目的として開発されたものである。そのため、開発者が機能を追加可能なプラグイン・アーキテクチャを採用している。

図 5.1 は Eclipse のプラグイン・アーキテクチャを図示したものである。プラグインの管理を行う OSGi ランタイムがアーキテクチャの最下層に位置し、その他の構成要素は全てプラグインとして提供されている。

プラグイン同士の接続は、プラグインが提供する拡張ポイントを通して行われる。各プラグインは自分自身を別のプラグインから拡張可能にするために、拡張ポイントを提供することが可能である。JDT や CDT(C/C++ Development Tooling)[13] といった Eclipse 上で提供される各種の統合開発環境も、図 5.1 に表される Eclipse プラットフォームが提供する拡張ポイントにプラグインを接続することで実現されている。プラグイン同士の

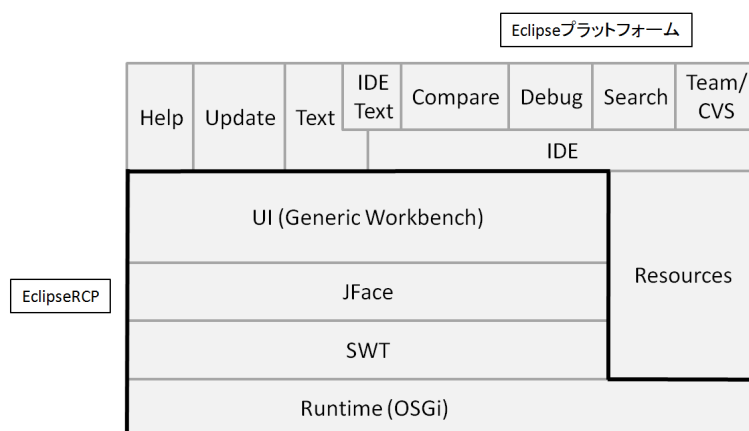


図 5.1: Eclipse のアーキテクチャ

接続は実行時に行われるため、拡張ポイントを提供するプラグインは実際にプラグインによってどのような拡張が行われるかを事前に知る必要はない。Eclipseはこのようなプラグイン・アーキテクチャにより、高い拡張性を実現している。

5.1.2 マニフェスト・ファイル

Eclipse プラグインを作成するには、プラグインの動作を記述する Java ソースコードの他に、plugin.xml 及び MANIFEST.MF という2つのマニフェスト・ファイルを記述する必要がある。plugin.xml はプラグイン・マニフェスト・ファイルと呼ばれ、プラグインの拡張および拡張ポイントの定義などを XML を用いて図 5.2 のように記述する。MANIFEST.MF は、Eclipse 3.0 からプラグインを管理するための導入された OSGi のためのマニフェスト・ファイルで、プラグインの ID や名称・作成者、他のプラグインとの依存関係などについて記述する。

図 5.2 は、本研究が提供する LayerIDE の実装のために、Eclipse に新たなビルダーを定義しているところである。extension 節の id 属性に拡張を識別するための ID を、name に拡張の名前を、point 属性に拡張を接続する拡張ポイントを定義している。このように extension 節を定義した後に、拡張ポイントごとの定義を記述する。ここでは builder 節の hasNature 属性により、このビルダーがネイチャーを持つことを宣言し、run 節の class 属性によりビルドを行うための LayerBuilder クラスを登録している。

また、図 5.3 は、LayerIDE プラグインの MANIFEST.MF ファイルの

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?eclipse version="3.4"?>
3 <plugin>
4     :
5     <extension
6         id="layerIDE.builder.LayerBuilder"
7         name="レイヤービルダー"
8         point="org.eclipse.core.resources.builders">
9         <builder
10            hasNature="true">
11            <run
12                class="layerIDE.builder.LayerBuilder">
13            </run>
14        </builder>
15        :
16 </plugin>
```

図 5.2: plugin.xml の記述例

```
1 Manifest-Version: 1.0
2 Bundle-ManifestVersion: 2
3 Bundle-Name: LayerIDE
4 Bundle-SymbolicName: LayerIDE; singleton:=true
5 Bundle-Version: 1.0.0.qualifier
6 Bundle-Activator: layerIDE.Activator
7 Require-Bundle: org.eclipse.ui,
8     org.eclipse.core.runtime,
9     org.eclipse.core.resources;bundle-version="3.6.0",
10    org.eclipse.ui.editors;bundle-version="3.6.0",
11    org.eclipse.jface.text;bundle-version="3.6.0",
12    org.eclipse.jdt.ui;bundle-version="3.6.0",
13    org.eclipse.jdt.core;bundle-version="3.6.0"
14 Bundle-ActivationPolicy: lazy
15 Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

図 5.3: MANIFEST.MF の記述例

中身である。MANIFEST.MF ファイルには、プラグインの ID やバージョン、Eclipse に読み込まれるタイミング、他のプラグインとの依存関係などが記述されている。

5.1.3 マニフェスト・エディター

5.1.2 小節で述べたマニフェスト・ファイルの記述を直接編集することは困難である。そのため Eclipse には Eclipse プラグインの開発環境として PDE(Plugin Development Environment)[16] が用意されている。LayerIDE の実装もこれを用いて行った。

PDE ではマニフェストの編集を、図 5.4 のようなマニフェスト・エディターを用いて行う。マニフェスト・エディターは、プラグイン開発者がマニ

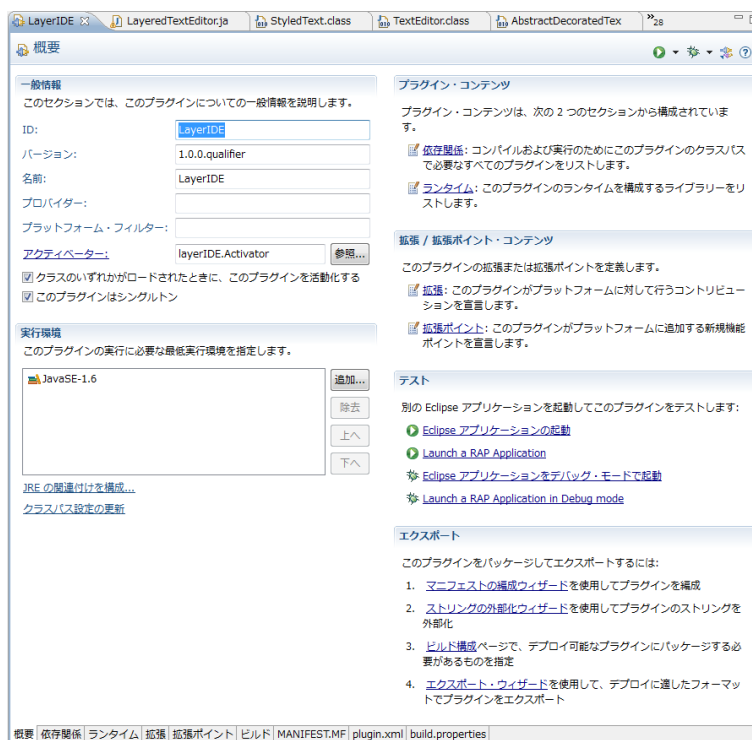


図 5.4: マニフェスト・エディター

ページ	説明
概要	プラグインの ID や名称などのプラグイン全体の設定、及び作成中のプラグインを読み込んだ Eclipse ワークベンチの起動や配布用アーカイブの作成などを行う。
依存関係	作成中のプラグインを実行するために必要なプラグインを指定する。
ランタイム	作成中プラグインが別のプラグインに公開するパッケージ、及びその可視性を設定する。
拡張	作成中のプラグインがどの拡張ポイントを用いて拡張を行うかについて設定する。
拡張ポイント	作成中のプラグインが他のプラグインに対して提供する拡張ポイントの定義を行う。
ビルド	ビルド時にアーカイブに含めるファイルやディレクトリを設定を行う。
MANIFEST.MF	MANIFEST.MF のソースを表示する。
plugin.xml	plugin.xml のソースを表示する。
build.properties	build.xml のソースを表示する。

表 5.1: マニフェスト・エディター

フェスト・ファイルを編集するための GUI を提供する。PDE 上で MANIFEST.MF ファイルを開くと、図 5.4 のようなマニフェスト・エディターが表示される。図 5.4 の下部に表示されている各タブを開くことで、表 5.1 のような編集を行うことが可能である。

5.2 Eclipse の拡張方法

Eclipse の拡張を行うための方法としては、次のようなものが考えられる。

- Eclipse を構成するプラグインのソースコードを直接書き換える
- 似た機能を持つプラグインが提供する拡張ポイントを用いて動作を変更する
- 似た機能を持つプラグインが提供する API を利用し、新たなプラグインとして記述する

LayerIDE の実装は JDT や JFace の API を利用して 3 つ目の方法で行った。本節ではそれぞれの拡張方法について利点と欠点を挙げ、LayerIDE の実装を何故その方法で行ったかについて述べる。

5.2.1 ソースコードを直接書き換える方法

これは、通常のプログラムを記述する際のように、機能を追加したい部分のソースコードを直接書き換えることにより Eclipse の動作を変更するという方法である。Eclipse や JDT はオープンソースであるため、このように元のソースコードを直接変更することで機能を拡張することも可能である。

この方法の利点としては、拡張ポイントなどの Eclipse プラグイン開発特有の知識がなくても比較的拡張が容易であり、また拡張を行う箇所が分かり易いということが挙げられる。他には、拡張を行いたいプラグインが拡張ポイントを公開していない場合でも動作の変更が可能のため、拡張ポイントを利用した拡張方法に比べてより自由度が高く、また局所的な動作の変更が可能であるという利点がある。

しかしながらこの方法は Eclipse のプラグイン・アーキテクチャに則っていないため、プラグインの作者が拡張を行う場合以外では基本的には使用するべきではない。この方法の大きな欠点としては、拡張元のプラグインがアップデートされた際にアップデートに合わせてソースコードを記述し直さなければならないことや、その際に自分が記述した箇所が分かりづらいこと、さらに、拡張元のプラグインのライセンスなどの問題が挙げら

れる。ただし、前述の拡張の容易さから、Eclipse の拡張を行う際に最初にこの方法で動作や実現可能性の確認を行い、その後新たにプラグインとして実装し直すといった方法は有用であると考えられる。

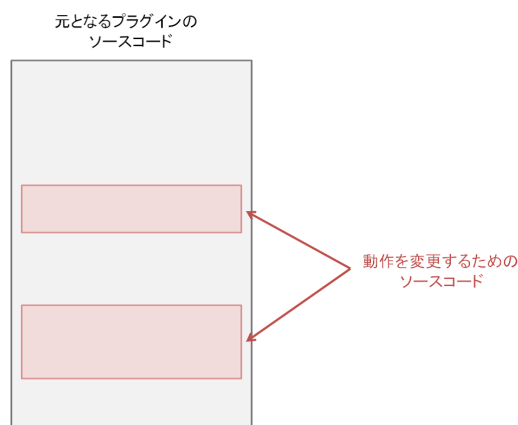


図 5.5: ソースコードを直接書き換える方法

5.2.2 似たプラグインの拡張ポイントを利用する方法

これは、Eclipse のプラグイン・アーキテクチャに則り、Eclipse を構成するプラグインが公開している拡張ポイントを利用し、作成したプラグインを接続する方法である。

この方法の利点は、元となるプラグインのソースコードを変更することなく、プラグインの動作を変更することが可能な点である。そのため元となるプラグインのバージョンが変わっても、提供されている拡張ポイントの仕様が変更されていなければ、作成したプラグインをそのまま利用可能である。

この方法の欠点は、変更したい部分に適した拡張ポイントを、元となるプラグインが提供しているとは限らない点である。プラグインによる拡張は提供された拡張ポイントに対してのみ行えるため、変更を行いたい場合はそこに拡張ポイントが元となるプラグイン側に用意されていなければならない。しかし、そのプラグインがどのように拡張される可能性があるかを、プラグインの作者が事前に把握することは困難である。元となるプラグインが変更可能であれば、行いたい変更に合わせて拡張ポイントを新しく作ることも可能であるが、そうでない場合は変更したい部分からは遠い拡張ポイントからクラスを辿って動作を変更することになり、ソースコードを直接書き換える方法に比べて実装コストが大幅に増加する。

また、小さなプラグインではそもそも拡張ポイントが用意されていない場合もある。そのような場合この方法を用いて元となるプラグインの動作を変更することは不可能であり、次に述べる3つ目の方法を用いることになる。

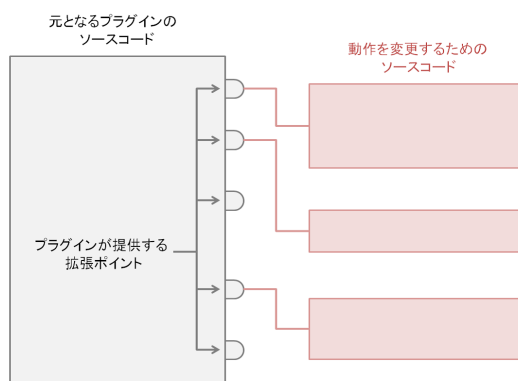


図 5.6: 似たプラグインの拡張ポイントを利用する方法

5.2.3 似たプラグインの API を利用して新たなプラグインを作成する方法

この方法は、元となるプラグイン自体の動作を変更するのではなく、元となるプラグインの実装を利用した新たなプラグインとしてエディタやビューなどといった部品を作成する方法である。多くのプラグインが提供している拡張ポイントは、そのプラグイン自体の動作を変更することを意図して用意されているが、Eclipse には新しいエディターやビューなどを作成することを意図した、`org.eclipse.ui.editors` や `org.eclipse.ui.view` といった拡張ポイントも用意されている。このような拡張ポイントを利用することで、既存のエディタやビューの動作を変更するのではなく、新しいエディタやビューを作成し、Eclipse 上で利用することが可能である。

この方法は、拡張ポイントにプラグインを接続するという意味で本質的には2つ目の方法と同じであるが、2つ目の方法が元となるプラグインと親子関係にあたるのに対し、この方法は元となるプラグインと兄弟関係にあたる。即ち、2つ目の方法が元となるプラグインの拡張ポイントに処理を織り込むのに対し、この方法では元となるプラグインの処理をラップする新たなプラグインを作成することになる。そのため元となるプラグインが、変更したい部分に適した拡張ポイントを提供していなかった場合でも、その実装を利用した新たなプラグインを作成することが可能である。

ただし、プラグインの処理全体を記述する必要があるため、実装コストは2つ目の方法に比べてさらに増加する。

2つ目の方法との大きな相違点として、元となるプラグインと自分が作成したプラグインのどちらが主体となるか、という点が挙げられる。2つ目の方法の場合は元となるプラグインの動作自体を変更することになるが、この方法の場合は元となるプラグインとは別に新しいプラグインで動作を定義することになる。そのため、プラグインのユーザーは新しく導入したプラグインを除去することなく、用途に合わせてプラグインを切り替える、といった利用方法も可能になる。

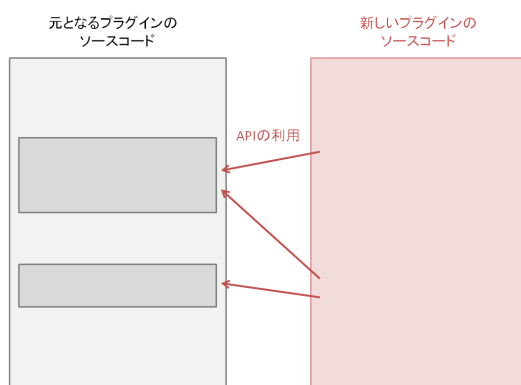


図 5.7: 似たプラグインの API を利用して新たなプラグインを作成する方法

5.2.4 LayerIDE の実装方法

LayerIDE の実装は本節で3つ目に述べた、新たなプラグインを作成する方法を用いて行った。これは LayerIDE の元となるプラグインである JDT に、今回の実装に適した拡張ポイントが用意されていなかったためである。LayerIDE の実装のうち、エディター及びビルダー部分は対象言語 (Java) の開発環境である JDT のエディターとビルダーの API をそれぞれ利用して行った。また、ビュー部分は JDT には該当する機能存在しなかったため、Eclipse の GUI を提供している JFace の API を利用して実装を行った。

5.3 実装

前節では一般的な Eclipse の拡張方法について述べた。本節では実際に行った、LayerIDE の実装方法について述べる。

LayerIDE の実装は、主にビュー、エディター、ビルダー部分から成る。本節ではそれぞれの部分の実装方法について記述する。

5.3.1 ビューの実装

LayerIDE ではレイヤーの管理を行うためのビューを Eclipse に追加した。Eclipse には新たなビューを追加するための拡張ポイントとして `org.eclipse.ui.views` 拡張ポイントが用意されており、LayerIDE の実装でもこれを利用した。この拡張ポイントの `category` タグ及び `view` タグに登録した主な属性は、表 5.2、5.3 の通りである。

属性	内容
<code>name</code>	このビューが属するカテゴリ名
<code>id</code>	カテゴリの ID

表 5.2: view 拡張ポイントの `category` タグの主な属性

属性	内容
<code>name</code>	このビューが属するカテゴリ名
<code>icon</code>	ビューに関連づけられたアイコン
<code>category</code>	ビューの属するカテゴリ
<code>id</code>	ビューの ID

表 5.3: view 拡張ポイントの `view` タグの主な属性

ビューの GUI には `CheckboxTreeViewer` を利用した。これは JFace で提供されている GUI 部品であり、チェックボックス付きのツリー構造を提供する。LayerIDE ではビューに `CheckboxTreeViewer` を配置し、このビューアーにチェックボックスの状態が変化した際に処理を行うための `CheckStateListener` と、ツリー上の選択されているノードが変化した際に処理を行うための `SelectionChangedListener` を登録することにより、レイヤーの適用状態の変更及び着色するレイヤーの選択を行うための機構を実現した。

LayerIDE の実装では、選択ノードが変化した際には背景色の再描画をエディターに要請する。またチェックボックスの状態が変化した際には次の処理を行う。

- レイヤーの適用状態の変更
- ワークスペースの再ビルドを LayerIDE のビルダーに要請
- ソース断片の表示・非表示の切り替えを LayerIDE のエディターに要請

エディターやビルダーはビューと互いに独立なため直接取得することは不可能である。そのためエディターの取得は PlatformUI クラスを用いて図 5.8 のように行った。

```
1 List<LayeredTextEditor> editorList = new LinkedList<LayeredTextEditor>();
2 IWorkbench workbench = PlatformUI.getWorkbench();
3 for(IWorkbenchWindow window : workbench.getWorkbenchWindows()) {
4     for(IWorkbenchPage page : window.getPages()) {
5         for(IEditorReference editorRef : page.getEditorReferences()) {
6             IEditorPart editor = editorRef[i].getEditor(true);
7             if(editor instanceof LayeredTextEditor) {
8                 editorList.add((LayeredTextEditor)editor);
9             }
10        }
11    }
12 }
```

図 5.8: エディターのリストの取得処理の実装

PlatformUI はワークベンチの情報を取得するための API である。ワークベンチとワークベンチウィンドウ、ワークベンチウィンドウとページ、ページとエディターリファレンスはそれぞれ 1:N の関係になっている。そのためそれぞれのワークベンチウィンドウに含まれるページからエディターリファレンスを取得し、instanceof を用いて LayeredTextEditor のリストを取得している。

また、ビルダーにワークスペースの再ビルドを要請するためには、ResourcesPlugin クラスを用いて図 fig:fullBuild のように記述すればよい。この例では引数として IncrementalProjectBuilder.FULL_BUILD を渡すことによりフルビルドを行っている。

```
1 ResourcePlugin.getWorkspace().build(IncrementalBuilder.FULL_BUILD, null);
```

図 5.9: ワークスペースのフルビルド

レイヤーの追加・削除・統合を行う機構は、それぞれの処理を行う Action を定義し、ビュー内のツールバー及びコンテキストメニューに追加することで実装した。ビュー上のツールバーへのアクションの追加、及びコンテキストメニューへのアクションの追加はそれぞれ、図 5.10 のように記述する

```
1 IMenuManager manager = getViewSite().getActionBars().getToolBarManager();
2 manager.add(makeLayerAction);

1 MenuManager manager = new MenuManager("#PopupMenu");
2 manager.setRemoveAllWhenShown(true);
3 manager.addMenuListener(new IMenuListener() {
4     public void menuAboutToShow(IMenuManager manager) {
5         manager.add(makeLayerAction);
6     }
7 });
8 Menu menu = manager.createContextMenu(viewer.getControl());
9 viewer.getControl().setMenu(menu);
10 getSite().registerContextMenu(manager, viewer);
```

図 5.10: ツールバー及びコンテキストメニューへのアクションの追加

ことで可能である。ツールバーへのアクションの追加は、ToolBarManager にアクションを追加することで行う。またコンテキストメニューへの追加では、他のビュー上やエディター上で右クリックを行った際と異なる動作を行う必要があるため、新たに MenuManager を定義してコンテキストメニューへ登録することで行う。

新規レイヤーを追加するための MakeLayerAction クラスは、図 5.11 のようにして実装した。

```
1 public class MakeLayerAction extends Action {
2     private CheckboxTreeViewer viewer;
3     private LayerNode layerTree;
4
5     public MakeLayerAction(CheckboxTreeViewer viewer, LayerNode layerTree) {
6         this.viewer = viewer;
7         this.layerTree = layerTree;
8         this.setText("新規レイヤーの作成");
9         this.setToolTipText("新規レイヤーの作成");
10        this.setImageDescriptor(PlatformUI.getWorkbench().getSharedImages().
11            getImageDescriptor(ISharedImages.IMG_OBJ_FILE));
12    }
13    public void run() {
14        WizardDialog dialog = new WizardDialog(null, new NewLayerWizard(viewer));
15        dialog.open();
16    }
17 }
```

図 5.11: エディターのリストの取得処理の実装

MakeLayerAction クラスのコンストラクタでは、ツールバー上でのアイコンやコンテキストメニュー上でのテキストなどについて設定を行っている。また、run メソッドにはアクションが選択された際の処理を記述する。図 5.11 では現在のレイヤー状態を引数として、新たなレイヤーを作

成するためのウィザードを開いている。

レイヤーの削除・統合についてもレイヤーの追加と同様に Action を定義し、ビュー内のツールバーやコンテキストメニューに登録することで実装を行った。レイヤーの削除時には、選択されたレイヤーに属するコード断片を削除する処理を、統合時には選択されたレイヤーのシンボルを統合する処理をそれぞれ記述した。

また LayerIDE では、Eclipse の再起動時に前回終了時のレイヤー状態を復元する機構を実装した。Eclipse ではこのような処理を行うために、Memento パターンを用いた、状態の保存・復元機構が用意されている。LayerIDE の実装では、ViewPart クラスの saveState メソッド及び init メソッドをオーバーライドし、createPartControl メソッドでロードした情報を適用することで、レイヤー状態の保存・復元機構を実現した。それぞれのメソッドに記述した処理は表 5.4 の通りである。

メソッド	内容
saveState(IMemento)	レイヤー状態をシリアライズし、メメントに追加する
init(IViewSite, IMemento)	メメントからレイヤー状態を取得し、デシリアライズする
createPartControl(Composite)	取得したレイヤー状態をビューに適用する

表 5.4: レイヤー状態の保存・復元機構のためのメソッド

5.3.2 エディターの実装

レイヤー機構付き IDE では編集時にレイヤーの着色及び表示・非表示の切り替えを行うために、エディターを実装する必要がある。LayerIDE の実装では 4.2 節で述べた理由により JDТ のエディターの拡張として実装することが困難であり、新たなエディターを作成することで実装を行った。

LayerIDE の実装では org.eclipse.ui.editors 拡張ポイントを用いることで、Eclipse に新たなエディターを追加した。この拡張ポイントの editor タグに登録した主な属性は、表 5.5 の通りである。

LayerIDE のエディターは TextEditor クラスを継承して実装を行った。このエディターではレイヤー機構を実現するため、標準的なテキストエディターの機能に加えて以下の機能を持つ。

- #ifdef/#endif ディレクティブに従ってマーカーを設置

属性	内容
class	エディターの実装クラス
extensions	このエディターで編集するファイルの拡張子
id	エディターのID
name	エディターの名前

表 5.5: editors 拡張ポイントの editor タグの主な属性

```

1 void updateFolding() {
2     try {
3         ProjectionViewer viewer = (ProjectionViewer)getSourceViewer();
4         if(viewer==null) return;
5         ProjectionAnnotationModel model = viewer.getProjectionAnnotationModel();
6         if(model==null) return;
7
8         model.removeAllAnnotations();
9
10        IDocument doc = getDocumentProvider().getDocument(getEditorInput());
11        String source = doc.get();
12
13        this.applyFolding(source, model);
14    } catch(Exception e) {
15        e.printStackTrace();
16    }
17 }

```

図 5.12: レイヤーを折りたたむためのマーカー設置処理の実装

- ビューの適用状態に従ってソース断片の折りたたみを行う
- ビューの選択状態に従ってソース断片の背景色を変更

マーカーの設置は、ソースファイルの保存時に図 5.12 の `updateFolding` メソッドを呼ぶことにより行った。`updateFolding` メソッドでは、ソースビューアーからアノテーションモデルを、ドキュメントプロバイダーからソースコードを取得し、`applyFolding` メソッドに引数として渡している。`applyFolding` メソッドではソースコードから `#ifdef/#endif` ディレクティブのオフセットを取得し、アノテーションモデルにマーカーの追加を行っている。

ソース断片の折りたたみは、レイヤーの選択状態の変更時に図 5.13 の `changeLayerState` メソッドを呼ぶことにより行った。`changeLayerState` メソッドでは、アノテーションモデルから図 5.12 で設置したマーカーを取得し、指定されたレイヤーのマーカーの折りたたみ状態のみを変更している。

背景色の変更は、ソースビューアーから取得したテキストウィジェットに `LineBackgroundListener` を登録することにより行った。これは、`JFace`


```
1 private void changeLayerState(String layerName, boolean apply) {
2     ProjectionViewer viewer = (ProjectionViewer)getSourceViewer();
3     if(viewer==null) return;
4     ProjectionAnnotationModel model = viewer.getProjectionAnnotationModel();
5     if(model==null) return;
6
7     for(Iterator i = model.getAnnotationIterator();i.hasNext();) {
8         Object o = i.next();
9         if(o instanceof LayerProjectionAnnotation) {
10            LayerProjectionAnnotation a = (LayerProjectionAnnotation)o;
11            if(a.getLayerName().equals(layerName)) {
12                if(apply == true) {
13                    model.expand(a);
14                } else {
15                    model.collapse(a);
16                }
17            }
18        }
19    }
20 }
```

図 5.13: レイヤーの折りたたみ処理の実装

の `StyledText` が持つ機能であり、ソースコードの行ごとに背景色を変更することができる。LayerIDE ではこのリスナーの `lineGetBackground` メソッドが呼ばれた際に、その行がビュー上で選択されたレイヤーに属していれば背景色を桃色に、そうでなければ白色にするという処理を記述することにより、レイヤーに属するコード断片の着色を実装した。

5.3.3 ビルダの実装

Eclipse でのコンパイル処理は、編集ファイルの保存時や手動ビルド時にビルダーによって行われる。ビルダーは、Eclipse 上で手動ビルドを行った際や、編集ファイルの保存時に行われる自動ビルド時の動作を決定する部分である。LayerIDE ではコンパイル時にプリプロセス処理を行うため、新たなビルダーを定義する必要がある。LayerIDE の実装においてビルダーが行うべき処理は、ファイルの保存時、及びレイヤーの適用状態の変更時に、ソースコード中のプリプロセス命令を解釈してコンパイルを行い、クラスファイルを生成することである。

Eclipse に新たなビルダーを追加するには、そのビルダーを扱うためのネーチャーを定義する必要がある。ネーチャーは、Eclipse 上のプロジェクトとビルダーを関連付けるためのものである。Eclipse には新たなネーチャーを定義するための拡張ポイントとして `org.eclipse.core.resources.natures` が用意されており、LayerIDE でもこれを利用した。この拡張ポイントの `builder` タグの `id` 属性にビルダーの実装クラスを、`runtime` タグ内の `run`

メソッド	内容
configure()	プロジェクトにビルダーをインストール
deconfigure()	プロジェクトからビルダーをアンインストール
setProject(IProject)	ネーチャーにプロジェクトを関連付ける
getProject()	ネーチャーに関連付けたプロジェクトを取得

表 5.6: ネーチャーの実装クラス

タグの class 属性にネーチャーの実装クラスをそれぞれ登録することで、プロジェクトとビルダーを関連付けるためのネーチャーを定義できる。

ネーチャーの実装は IProjectNature インターフェースを実装することにより行った。IProjectNature インターフェースでは4つの抽象メソッドが宣言されている。それぞれのメソッドに記述した内容は表 5.6 の通りである。

また新たなビルダーの定義は、org.eclipse.core.resources.builders 拡張ポイントを用いて行った。拡張ポイントに登録した内容は 5.1.2 小節及び図 5.2 で例として採り上げたのでここでは割愛する。

ビルダーの実装は一般的には抽象クラスである IncrementalProjectBuilder を継承することで行う。LayerIDE の実装では JDT の JavaBuilder クラスを継承することで実装を行った。LayerIDE のビルダーは以下の処理を順に行うことで、プリプロセスディレクティブで注釈されたソースファイルからクラスファイルを生成する。

1. Java プロジェクトとレイヤー情報を取得
2. ソースファイルをプリプロセッサを通して変換
3. JDT のビルダーを用いてクラスファイルを生成
4. 変換したソースファイルをもとに戻す

Java プロジェクトはビルダークラスから getProject メソッドにより取得し、レイヤー情報を保持するビューの取得は図 5.8 同様に PlatformUI クラスを用いて行った。

ソースファイルの変換は IResourceVisitor を用いて図 5.14 のように行った。Eclipse にはリソースのビジターとして IResourceVisitor インターフェースが用意されている。LayerIDE ではレイヤー状態の変更時にフルビルドを行うためこのインターフェースを用いて、全てのリソースに対してそれがレイヤー機構付き Java ファイルであればプリプロセッサにかける、という処理を記述した。

```
1 protected void preprocess(final IProgressMonitor monitor) throws CoreException {
2     final HashMap<String,TreeObject> layerMap = new HashMap<String,TreeObject>();
3
4     class PreprocessVisitor implements IResourceVisitor {
5         public boolean visit(IResource resource) {
6             internalPreprocess(resource, layerMap, monitor);
7             return true;
8         }
9     }
10
11     LayerNode root = view.getLayerTree();
12     getLayerList(layerMap, root);
13     try {
14         getProject().accept(new PreprocessVisitor());
15     } catch(CoreException e) {
16         e.printStackTrace();
17     }
18 }
19
20 private void getLayerList(Map<String,LayerNode> map, LayerNode parent) {
21     for(LayerNode node : parent.getChildren()) {
22         map.put(node.getName(), node);
23         getLayerList(map, node);
24     }
25 }
```

図 5.14: ソースファイルの変換処理の実装

preprocess メソッドでは IResourceVisitor を継承した PreprocessVisitor により、リソースの訪問時に internalPreprocess メソッドを呼んでいる。この際引数としてリソースの他に、getLayerList メソッドを用いて取得したレイヤーのリストを渡している。実際のプリプロセス処理は internalPreprocess メソッドで行われるが、本研究の主題とは離れるためここでは割愛する。LayerIDE の実装ではプリプロセス処理を行う際に元のソースコードを保持しておき、JDT のコンパイラによるクラスファイルの生成が終了した後に、ソースファイルの復元を行っている。

第6章 評価とまとめ

本章では本研究の評価とまとめを行う。

本研究ではレイヤー機構付き IDE の有用性を確かめるために、差分記述が有用な状況を想定し、実際に LayerIDE を用いて差分記述によるプログラムの作成を行った。6.1 節ではそのプログラムの説明と本研究の評価について記述し、6.2 節では本研究のまとめを行う。

6.1 評価

本研究ではレイヤー機構付き IDE の有用性を確かめるために、複数の類似したプログラムの作成を LayerIDE を用いて行った。ここでは次のような類似した3つのプログラム作成が必要な状況を想定している。

- コンソール上でシミュレートを行うプログラム
- ウィンドウ上でシミュレートを行うプログラム
- その両方を行うプログラム

本研究ではウィンドウ上及びコンソール上でシミュレート可能なサンプルプログラムとして、ぶよぶよの連鎖シミュレーターを作成した。作成したシミュレーターはプログラムの引数としてぶよぶよのフィールド情報を受け取り、連鎖をウィンドウ及びコンソール上でシミュレートする。このシミュレーターは表 6.1 のクラス構成からなる。

想定されている状況では、ウィンドウ及びコンソールに関連する部分がそれぞれ独立に付け外し可能な構成が望ましい。そのため、連鎖のシミュレートを行う部分をコアとし、ウィンドウ・コンソールに関連する部分は差分としてそれぞれ GUI・CUI レイヤーに記述した。

またフィールド状態の変化をログとして出力するために、LOG レイヤーを作成した。デバッグ時には LOG レイヤーを適用し、デバッグ終了時には LOG レイヤーを非適用または削除することを想定している。ログ出力に関する処理を独立したレイヤーに記述することでプログラムのデバッグが容易になり、また完成時にはログ出力のための余計なコードが残ることを防ぐことが可能である。

クラス	内容
Puyo	プログラムを開始するメインクラス 引数からフィールドを作成し、ループ内でステップの進行及びフィールドの描画を行う
Field	フィールドクラス step() メソッドでフィールドの状態を一つ進め、 show() メソッドでフィールドを描画する
Window	フィールドを描画するためのウィンドウクラス GUI 差分でのみ使用

表 6.1: シミュレーターのクラス構成

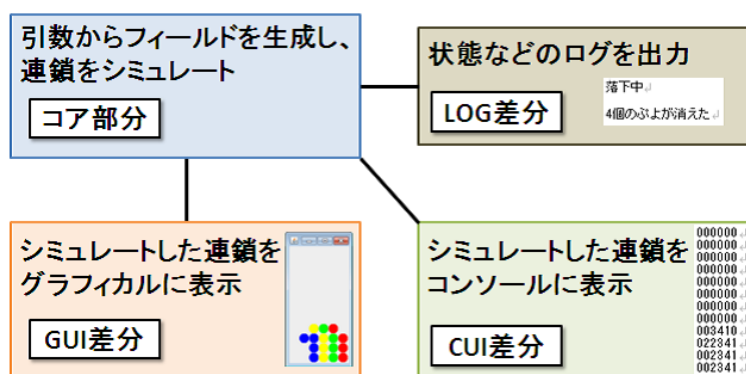


図 6.1: シミュレーターの差分構成

	Puyo	Field	Window	合計
root	12 行	95 行		107 行
GUI 差分		10 行 (3 箇所)	43 行 (1 箇所)	53 行
CUI 差分		9 行 (1 箇所)		9 行
LOG 差分		15 行 (5 箇所)		15 行
合計	12 行	129 行	43 行	184 行

表 6.2: シミュレーターのコード行数

図 6.1 は作成したシミュレーターのレイヤー構成を、表 6.2 はそれぞれのレイヤーに属するコード行数をクラスごとにまとめたものである。空白の部分はそのレイヤーに属するコード行が存在しないことを表す。

図 6.1 のようにレイヤー分けを行ったことにより目的の 3 つのプログラムはそれぞれ、

- CUI レイヤーのみを適用
- GUI レイヤーのみを適用
- CUI・GUI レイヤーを適用

とすることで、それぞれのプログラムを別々に作成することなく、コア部分に対する差分として記述することが可能であった。

シミュレーターの実装では消えるぶよの個数をログとして出力するため、for 文の中に個数をカウントするための処理を挿入する必要があった。このような差分の記述は、AspectJ や GluonJ といったアスペクト指向言語を用いた差分記述では、適当なジョインポイントが存在しないため困難である。しかし、レイヤー機構付き IDE ではプリプロセスディレクティブを用いて元のソースファイル中に直接差分を記述するため、容易に記述可能であった。

また、ログ出力のようなソースコードの各所に分散しがちな差分の記述を行う場合、一般的なプリプロセスディレクティブによる差分記述では可読性の低下が起こる。しかし LayerIDE では IDE 上でレイヤーを非適用とすることにより、LOG レイヤーのコード断片のみを非表示とすることが可能なため、可読性は低下しない。また、プログラムと本質的に関係のないプリプロセスディレクティブを薄い色で表示することにより、プリプロセスディレクティブ行の挿入により処理の流れが分かりづらくなるという問題の緩和に成功した。

LayerIDE を用いて差分記述を行った場合、プログラムの実行時間に対するオーバーヘッドは存在しない。これは、非適用のレイヤーに記述されたコード断片はプリプロセッサにより除去され、生成されたクラスファイルには含まれないためである。

6.2 まとめ

本研究ではプログラムの差分記述を容易に行うためのレイヤー機構付き IDE を提案した。レイヤー機構付き IDE では差分に関連するコード断片を IDE 上でレイヤーとしてまとめて管理することにより、可読性を下げることなく任意の差分の記述が可能である。レイヤー機構付き IDE では、それぞれの差分に関連したコード断片を、レイヤーごとに編集時の表示・非表示の切り替えやコンパイル時の適用・非適用の選択が可能である。ま

た選択したレイヤーに記述されたコード断片の背景色を変更することにより、ある差分に関連するコード断片を視覚的に抽出することが可能である。レイヤー機構付き IDE を用いることで、類似した複数のソフトウェア開発のコストの低減が可能となった。

また、本研究の提案するレイヤー機構付き IDE の実装例である LayerIDE の開発を、Eclipse プラグインとして行った。LayerIDE では対象言語として Java を、プリプロセッサディレクティブとして `#ifdef/#endif` を選択した。LayerIDE を用いることにより、Eclipse 上で Java 言語を用いた差分記述をレイヤー機構を用いて行うことが可能となった。本研究では複数の差分からなるプログラム例としてぷよぷよの連鎖シミュレーターを LayerIDE を用いて作成し、レイヤー機構付き IDE の有用性を確かめた。

参考文献

- [1] Batory, D.: AHEAD Tool Suite, <http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/>.
- [2] Batory, D.: Feature-oriented programming and the AHEAD tool suite, *Proceedings of the 26th international Conference on Software Engineering*, IEEE Computer Society, pp. 702–703 (2004).
- [3] Chiba, S. and Ishikawa, R.: Aspect-oriented programming beyond dependency injection, *ECOOP 2005-Object-Oriented Programming*, pp. 121–143 (2005).
- [4] Chiba, S., Nishizawa, M. and Kumahara, N.: GluonJ home page.
- [5] Clement, A., Colyer, A. and Kersten, M.: Aspect-Oriented Programming with AJDT, *ECOOP Workshop on Analysis of Aspect-Oriented Software* (2003).
- [6] Kastner, C., Apel, S. and Kuhlemann, M.: CIDE: Virtual Separation of Concerns, <http://www.fosd.de/cide/>.
- [7] Kastner, C., Apel, S. and Kuhlemann, M.: Granularity in software product lines, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008* (Schafer, W., Dwyer, M. B. and Gruhn, V.(eds.)), ACM (2008).
- [8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.: An overview of AspectJ, *ECOOP 2001 Object-Oriented Programming*, pp. 327–354 (2001).
- [9] Microsoft: Visual Studio ホームページ, <http://www.microsoft.com/japan/msdn/vstudio/>.
- [10] Prehofer, C.: Feature-oriented programming: A fresh look at objects, *ECOOP'97 Object-Oriented Programming*, pp. 419–443 (1997).

- [11] the Eclipse Foundation: AspectJ Development Tools (AJDT), <http://www.eclipse.org/ajdt/>.
- [12] the Eclipse Foundation: The AspectJ Project, <http://www.eclipse.org/aspectj/>.
- [13] the Eclipse Foundation: Eclipse CDT, <http://www.eclipse.org/cdt/>.
- [14] the Eclipse Foundation: Eclipse Java development tools (JDT), <http://www.eclipse.org/jdt/>.
- [15] the Eclipse Foundation: Eclipse.org home, <http://www.eclipse.org/>.
- [16] the Eclipse Foundation: PDE, <http://www.eclipse.org/pde/>.