平成23年度 修士論文

統合開発環境によって
表現された言語機構による
コードのモジュール化

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 10M37143

寺本 裕基

指導教員

千葉 滋 教授

平成24年1月27日

概要

プログラムのコードを再利用するのに最も簡単な方法はコピー＆ペーストである。この方法は便利であり直感的で使いやすいため、プログラミングに慣れていない初心者は使用してしまいがちである。しかし、コピー＆ペーストを用いて開発を進めていくと、特定の機能に関するコードが複数箇所に散在してしまい、コードの再利用性が低下してしまう。

そのため言語機構を用いてモジュール化し、コードを再利用するのが一般的である。しかし、言語機構を用いてモジュール化を行うと、その言語機構の文法や意味論を理解しなければならない。一度その言語機構の使い方を理解したとしても、今後新しいモジュール化技術が提案されると、そのたびにその技術に対応した新しい言語機構を理解し、覚えなければならない。また、新しい言語機構に対応したライブラリや開発環境なども用意しなければならない。開発者がこれらを導入するには非常に手間がかかり、負担となり得る。

そこで本研究では、言語機構だけに頼らずに、統合開発環境を活用してコード再利用のためのモジュール化を実現する方法について提案する。本研究では、直感的で使いやすいコピー＆ペーストを用いて開発を行うことを前提としている。本研究で提案するシステムを同期可能なコピー＆ペーストシステムと呼ぶ。

提案では、プログラマが再利用したいコードの領域をコピー＆ペーストによって複製し、その複製されたコードを本システムが同期させることによって擬似的なモジュール化を実現する。同期されているコードを開発者が編集すると、本システムにより同期されている別の領域のコードも自動的に編集される。さらに、同期させているコード領域のうち一部分のみ同期可能にすることで、関数などをパラメータ化させることができ、言語機構と同様なコード再利用が言語機構を用いずに実現可能となる。このように、統合開発環境を活用することにより、既存言語を拡張することなく言語機構によるモジュール化を表現することができる。このような統合開発環境を、Eclipse のプラグインとして実装した。

また、本研究では Java 言語に用意されていないモジュール化技法を、本システムを利用することによって言語拡張せずに導入できるか検証した。具体的には、アスペクト抽象やカリー化などの言語機構による既存の

モジュール化手法を取り上げ、それらを本システムによって表現できることを示した。言語拡張ではなく、直感的な操作で新しいモジュール化技法を取り入れることができるため、開発者の負担を減らすことができる。

謝辞

本研究を進めるにあたり、研究の方針などご指導を頂いた指導教員の千葉滋教授に心より感謝致します。武山文信氏には、本研究内容、システムの実装、論文の書き方などについて、様々な助言を頂きました。感謝致します。最後に、共に研究活動を行い、多くの助言を頂いた研究室の皆様に感謝致します。

目 次

第1章 はじめに **8**

第2章 関連研究 **10**
2.1 アスペクト指向プログラミング 10
2.1.1 オブザーバーパターン 10
2.1.2 AspectJ 12
2.1.3 AspectJ を利用したモジュール化 14
2.2 その他の関連研究 15
2.2.1 コードクローン 15
2.2.2 Fluid AOP 17

第3章 提案 **18**
3.1 コピー＆ペーストを利用したコードの再利用 18
3.2 同期可能なコピー＆ペーストシステム 19
3.2.1 ツールバー 19
3.2.2 Concerns ビュー 21
3.2.3 エディタ 23

第4章 実装 **27**
4.1 Eclipse プラグインの実装方法 27
4.1.1 Eclipse のアーキテクチャ 27
4.1.2 マニフェスト・ファイル 28
4.2 同期可能なコピー＆ペーストシステムの実装 29
4.2.1 エディタの実装 29
4.2.2 ツールバーの実装 32
4.2.3 Concerns ビューの実装 33
4.2.4 内部モデルの実装 36

第5章 評価 **38**
5.1 手続き抽象 38
5.2 パラメータ抽象 40
5.3 アスペクト 41
5.4 カリー化 43

第 6 章　まとめと今後の課題 49
6.1　まとめ 49
6.2　今後の課題 49

図 目 次

2.1 オブザーバーパターンの例 11
2.2 Shape クラス 12
2.3 アスペクトの例 13
2.4 アドバイスで局所変数を利用する場合 15

3.1 同期可能なコピー＆ペーストシステムの全体図 20
3.2 ツールバー 21
3.3 Concerns ビュー 22
3.4 同期領域の編集 24
3.5 パラメータの同期 26

4.1 plugin.xml の例 28
4.2 MANIFEST.MF の例 29
4.3 PDE 30
4.4 内部モデルのオブジェクト 37

5.1 手続き抽象の実現 39
5.2 Concerns ビュー 40
5.3 パラメータ 41
5.4 パラメータの実現 42
5.5 アスペクトの実現 42
5.6 複数パラメータを持つアスペクトの実現 44
5.7 Concerns ビューによるアスペクトの実現 45
5.8 Java にカリー化を導入する際の記述例 46
5.9 カリー化の実現 46
5.10 2 つ目以降のパラメータを固定するカリー化 47
5.11 Concerns ビューを活用したカリー化関数の定義 48

表 目 次

3.1 エディタ上でのコード表現 26

4.1 マニフェスト・エディタのタブページとその内容 31
4.2 org.eclipse.ui.editors 拡張ポイントで設定できる主な項目 . 32
4.3 org.eclipse.ui.editorActions 拡張ポイントで設定できる主な項目 32
4.4 org.eclipse.ui.views 拡張ポイントで設定できる主な項目 . . 34

第1章　はじめに

従来ではコードを再利用するために、言語機構を用いてモジュール化するのが一般的であった。言語機構を用いると強力なモジュール化ができるようになる反面、開発者はその言語機構の文法や意味論を理解しなければならないという問題がある。一度その言語機構の使い方を理解したとしても、今後新しいモジュール化技術が提案されるとその度に新しい言語機構を覚えなければならない。例えば、開発者が Java 言語を理解したとする。その後、アスペクト指向技術のような新しいモジュール化手法が提案され、それを使おうとすると、AspectJ のような Java 言語を拡張した新しい言語機構を理解しなければならない。また、新しい言語機構が提案されると、その言語に対応したライブラリ、フレームワーク、開発環境などを用意しなければならない。これらの導入には非常に手間がかかる。

さらに、言語機構の拡張には限界が存在する。例えば、AspectJ 言語では Java 言語より強力なモジュール化が可能となるが、AspectJ が提供しているアスペクトは、Java のクラスから横断的関心事と呼ばれる関心事を完全に分離する。そのため、Java のクラスを見ただけではプログラムの挙動を完全に把握することができない問題がある。そこで、完全に分離されていても Java のクラスにアスペクトの処理が織り込まれることを視覚的に表現できる AJDT のようなツールを利用する必要がある。このような言語機構の限界は、プログラミング言語が文字列に依存していることに起因する。文字列を使用している以上、1 次元な表現しかできないため、複数の視点からプログラムの構造を表現することが困難になる。

そこで本研究では、静的なテキストだけでなく動的なテキストも利用して、言語機構を表現する方法を提案する。動的なテキストを利用するために、統合開発環境（IDE）を活用する。本研究で提供するシステムでは、コードの領域を複製し、それを同期させることによって擬似的なモジュールを表現する。同期されているコードを開発者が編集すると、本システムによって同期されている別の領域のコードも自動的に編集される。さらに、同期領域のうち部分的に同期されない場所を作ることで、関数などのパラメータ化も表現できる。このように、統合開発環境上で動的なテキストを扱うことにより、既存言語を拡張することなく、新しい言語機構を導入することができる。

本論文は、以下のような構成になっている。第 2 章では、言語機構を利用したモジュール化について、アスペクト指向プログラミングを例に出しながら説明する。また、その他の関連研究について述べる。第 3 章では、本研究で提案する同期可能なコピー＆ペーストシステムの概要について説明する。第 4 章では、同期可能なコピー＆ペーストシステムの実装方法について述べる。第 5 章では、言語機構を利用したモジュール化と同期可能なコピー＆ペーストシステムを利用したモジュール化の比較を行う。最後に、第 6 章で本研究をまとめる。

第2章　関連研究

本章では、言語機構を利用したモジュール化とその問題点について述べる。言語機構を利用したモジュール化については、2.1 節でアスペクト指向プログラミングを例に出しながら説明する。また、2.2 節でその他の関連研究について述べる。

2.1　アスペクト指向プログラミング

アスペクト指向プログラミングはモジュール化技法の一つであり、横断的関心事を分離することができる。横断的関心事とは、ある処理の内容が複数のモジュール間に横断的にまたがり、単独のモジュールにすることができない関心事のことである。

横断的関心事が存在していると、プログラム中のコード断片が複数の関心事に関連する可能性があり、異なる関心事が絡みあうことになる。異なる関心事が絡みあっていると、コードを特定のグループごとに開発することが困難になり、効率的な保守ができない。また、特定の関心事に関するプログラムの動きを理解するためには、プログラマが複数の関心事を読む必要がある。

本節では、横断的関心事の例の 1 つであるオブザーバーパターン、アスペクト指向プログラミング言語である AspectJ[5, 8] 、そして AspectJ を利用したモジュール化について述べる。

2.1.1　オブザーバーパターン

オブジェクト指向プログラミングで使われるデザインパターンのひとつにオブザーバーパターンがある。オブザーバーパターンは、あるオブジェクトの状態が変更された時、別なオブジェクトで何らかの処理を実行するときのパターンである。このパターンを用いた例を図 2.1 に挙げる。

図 2.1 は図形エディタプログラムの一部である。図形エディタプログラムとは、四角形や円などの図形を描くことができるツールである。このコードでは、図形を表すオブジェクトは Shape クラス（図 2.2）を継承して作られ、それとは別に DisplayUpdate というディスプレイ再描画処理を

```
1  class DisplayUpdate {
2      static void update(Shape s) {
3          s.display.update(s);
4      }
5  }
6
7  class Rectangle extends Shape {
8      int width, height;
9      void setPos(int x, int y) {
10         xpos = x;
11         ypos = y;
12         DisplayUpdate.update(this);
13     }
14     void setWidth(int w) {
15         width = w;
16         DisplayUpdate.update(this);
17     }
18     void setHeight(int h) {
19         height = h;
20         DisplayUpdate.update(this);
21     }
22     int getWidth() {
23         return width;
24     }
25 }
```

図 2.1: オブザーバーパターンの例

定義するクラスがある。各図形オブジェクトでは、ユーザが図形の大きさを変更する、図形の位置を変更するなどの操作により、図形を表すデータに何らかの変化があった時、それぞれのオブジェクトが持つフィールド値がセットされる。また、図形オブジェクトのフィールド値が更新された時、それをディスプレイ上に反映する必要があるため、DisplayUpdate クラスで定義されたディスプレイ再描画処理を実行する。

このようなディスプレイ再描画処理は横断的関心事となる。ディスプレイを再描画するという関心事と、図形オブジェクトのフィールド値を変更する関心事が絡み合っている。このようなコードにおいて、例えば図形オブジェクトを表すクラスからディスプレイ再描画処理を取り除きたいとする。このとき、図形オブジェクトのフィールド値を変更するモジュールすべてを確認し、ディスプレイ再描画処理を取り除く修正をしなければならない。実際のコードでは図 2.1 に挙げたコードだけではなく、円を表す Circle や三角形を表す Triangle など他の図形オブジェクトや、色を表すフィールドなどが存在するため、 修正箇所が膨大な数になる。

```
1 class Shape {
2     Display display;
3     int xpos, ypos;
4 }
```

図 2.2: Shape クラス

2.1.2 AspectJ

AspectJ はアスペクト指向プログラミング言語の一つであり、Java 言語を拡張して作られている。AspectJ には、アスペクト、ポイントカット、アドバイスの 3 つの言語機構がある。

- アスペクト
  アスペクトは横断的関心事を実装した主要なモジュールである。アスペクトにはポイントカットとアドバイスを記述することができる。

- ポイントカット
  ポイントカットは、いつ横断的関心事の処理内容を実行するのかを指定する。ポイントカットで指定できるプログラム中の場所は、ジョインポイントの中から選択する。ジョインポイントとは、プログラム中の実行点を表す。ジョインポイントには、あるメソッドが呼ばれた時やフィールドに値が代入された時などのタイミングが含まれる。

- アドバイス
  アドバイスは、ポイントカットで指定された場所で実行される処理内容を記述する。

図 2.3 は AspectJ を用いてオブザーバーパターンを実装した例である。このコードは図 2.1 のコードと同じ動きをする。DisplayUpdate アスペクトには 1 つのポイントカットと 1 つのアドバイスが含まれる。change ポイントカットはジョインポイントを指定している。そのジョインポイントは、Shape クラスもしくはそのサブクラスで呼び出され、名前が set から始まるメソッドの実行時を表している。例えば、このジョインポイントには、Rectangle オブジェクト中での setWidth メソッド実行時などが含まれる。また、このポイントカットはパラメータ s を持っている。このパラメータは set から始まる名前のメソッドが実行されるオブジェクトを表す。

DisplayUpdate のアドバイス本体は、after 宣言されているため、change ポイントカットで指定しているジョインポイントの直後に暗黙的に実行される。ここには挙げていないが、AspectJ には after の他に、ポイントカットで指定されたジョインポイントの直前に実行する before 宣言など、

```
1  aspect DisplayUpdate {
2      pointcut change(Shape s):
3          execution(void Shape+.set*(..)) && this(s);
4
5      after(Shape s): change(s) {
6          s.display.update(s);
7      }
8  }
9
10 class Rectangle extends Shape {
11     int width, height;
12     void setPos(int x, int y) {
13         xpos = x;
14         ypos = y;
15     }
16     void setWidth(int w) {
17         width = w;
18     }
19     void setHeight(int h) {
20         height = h;
21     }
22     int getWidth() {
23         return width;
24     }
25 }
```

図 2.3: アスペクトの例

複数の実行点を指定できる。アドバイスで定義した処理内容をジョインポイントに統合することを、織り込むという。アドバイス本体が実行されている間、change ポイントカットのパラメータ s は set から始まる名前のメソッドが実行されるオブジェクトを表す。

オブジェクト指向なコードで記述された図 2.1 と、アスペクト指向なコードで記述された図 2.3 を比べると、オブジェクト指向なコードでは、DisplayUpdate クラスの update メソッドの呼び出しは Rectangle クラスのすべての set から始まる名前のメソッドの最後で呼び出されている。この呼び出しが記述されている場所は、図 2.3 の change ポイントカットで指定している場所と一致している。また、前述のような図形オブジェクトからディスプレイ再描画処理を取り除きたい場合、AspectJ ではプログラム中の全ソースコードの中から、DisplayUpdate アスペクトに関するコードを取り除くだけで良い。一方、オブジェクト指向なコードでは、すべての set から始まる名前のメソッドから DisplayUpdate.update メソッドの呼び出しを取り除かなければならない。

2.1.3　AspectJ を利用したモジュール化

AspectJ は横断的関心事を分離することにより、オブジェクト指向のモジュール化を改善した。コード再利用に関しては、AspectJ は、オブジェクト指向なコードである図 2.1 の図形を表す Rectangle クラスからディスプレイ再描画を表す DispalyUpdate クラスに関する余分なメソッド呼び出しを取り除くことができた。また、ディスプレイ再描画処理に関しては図 2.3 の DisplayUpdate アスペクトは図 2.1 の DisplayUpdate クラスと同様に、Rectangle クラスから再描画処理の詳細を隠すことができている。また、アスペクト指向なコードでは DisplayUpdate アスペクトの内容は、Rectangle クラスや他の Shape を継承したクラスを修正する必要がなく簡単に図形を表すクラスに追加、もしくは削除することが可能となった。

しかし、AspectJ のコードはオブジェクト指向なコードより劣っている点もある。図 2.3 のコードは、ディスプレイ再描画に関する処理を一つのアスペクトとしてまとめている。一方、Rectangle クラスのほうには、ディスプレイ再描画の処理は一切含まれていない。それに比べると、図 2.1 のコードは、ディスプレイ再描画処理を含み、DisplayUpdate.update メソッドの呼び出しがディスプレイ再描画関心事と四角形関心事の 2 つに関連することが分かる。AspectJ で開発すると、Rectangle クラスを見ただけではディスプレイ再描画されるかどうかが分からず、開発者はプログラムの挙動を把握しづらくなる。この欠点を補うため、AspectJ でプログラムを開発するために AJDT[7, 1] のようなツールを利用しなければならない。AJDT ではアスペクトが織り込まれる場所をソースコードエディタ上に表示させることができる。例えば、図 2.3 の Rectangle クラスの set から始まる名前のメソッドに印がつく。このように、Java 言語を拡張し、より良いモジュール化を行うことができる AspectJ でも他のツールに頼らなければ十分な開発環境を整えることができない。

また、AspectJ にはジョインポイント毎に異なる局所変数を必要とする場合などにコードの重複が増加してしまう問題がある。例えば、図 2.3 の update メソッド呼び出しにおいて、フィールドの値が変更された時のみ再描画処理を行いたいとする。この場合、図 2.4 のようにジョインポイント毎にアドバイスを記述しなければならない。このコードでは、2 行目から 9 行目までのアドバイスが setPos メソッド、11 行目から 18 行目までのアドバイスが setWidth メソッド、20 行目から 27 行目までのアドバイスが setHeight メソッドでの再描画処理を定義している。これらはすべて引数とフィールドの値を比べ、異なっていた場合のみ再描画処理を行なっている。図 2.4 に挙げたコードは一部のみであるが、実際は Shape クラス及びそれを継承したすべてのクラスの set から始まる名前のメソッドに対してコードを書かなければならない。そのため、似たようなコードが大量に

```
1  aspect DisplayUpdate {
2      void around(Rectangle rectangle, int x, int y):
3          execution(void Rectangle.setPos(int, int))
4          && this(rectangle) && args(x, y) {
5        if(rectangle.x != x || rectangle.y != y) {
6          proceed(rectangle, x, y);
7          rectangle.display.repaint(rectangle);
8        }
9      }
10
11     void around(Rectangle rectangle, int width):
12         execution(void Rectangle.setWidth(int))
13         && this(rectangle) && args(width) {
14       if(rectangle.width != width) {
15         proceed(rectangle, width);
16         rectangle.display.repaint(rectangle);
17       }
18     }
19
20     void around(Rectangle rectangle, int height):
21         execution(void Rectangle.setHeight(int))
22         && this(rectangle) && args(height) {
23       if(rectangle.height != height) {
24         proceed(rectangle, height);
25         rectangle.display.repaint(rectangle);
26       }
27     }
28     // ...
29 }
```

図 2.4: アドバイスで局所変数を利用する場合

出てしまう問題がある。

2.2　その他の関連研究

2.2.1　コードクローン

コードクローンとは、プログラムのソースコード中で似たようなコードのことを表す。コードクローンの編集を容易にするためのツールとして、Linked Editing[10] 、 Simultaneous editing[6] 、CReN[4] などが挙げられる。

Linked Editing

Linked Editing は重複するコードを同時に編集することが出来るツールである。このツールでは、複数のコードクローンをユーザが選択しリンクさせる。すると、Linked Editing はリンクさせたコードのうち、一致している部分とそうでない部分を区別する。ユーザがリンクさせたコードのうち一致している部分を編集すると、他のコードも同時に編集される。これにより、複数箇所に散在するコードクローンを一度に編集することが可能となる。

Simultaneous Editing

Simultaneous Editing では、レコードと呼ばれるユーザが編集する領域の集合を指定する。レコードの集合を定義した後、ユーザはそのうち一つのレコードを選択する。すると、Simultaneous Editing が他のレコードの集合からそのレコードと一致しているレコードを示す。ユーザがその共通のレコードを編集すると、すべてのレコードが同時に編集される。

CReN

CReN はコードクローンを検知し、そのコードクローン中で同期を行い、変数などのプログラム要素の名前変更を同時に行うことが出来る。これによりプログラムの保守性を向上させることができる。

共通の問題点

Linked Editing 、Simultaneous editing 、CReN の 3 つのコードクローン編集ツールを紹介した。どのツールも異なるコード領域の編集を同時に行うことができ、言語機構の限界をコード重複を同時に編集する面から補っている。しかし、どのツールも言語機構のみを用いた開発より保守性を向上させることができるが、同時に編集するコード領域の対応関係を複数持つとコードの管理が難しくなり、モジュール化という面では劣っている。コードクローンをモジュールとして扱うためには、コードに対して名前を付け、その名前で管理するなど、コード管理を容易にするための機能を用意する必要がある。

2.2.2 Fluid AOP

Fluid AOP[3] はアスペクト指向プログラミングのための統合開発環境である。このツールでは、AspectJ のような言語機構と、動的なテキストを利用した開発を行うことができる。Fluid AOP では、AspectJ のようなポイントカット記述をし、そのポイントカットで指定した場所を同時に編集することができる。また、ポイントカット記述で指定したすべてのコードの共通部分を表示させ、そうでない部分を隠すことができる。

Fluid AOP を用いることで、コードクローン編集ツールのように複数のコードを同時に編集することができ、同時にポイントカット記述によるコードの管理も実現している。しかし、このツールを使用するためには、AspectJ のようなポイントカット記述をしなければならない。そのため、アスペクト指向プログラミングの知識が必須となり、開発者が簡単に導入することが難しい。

第3章　提案

第2章で説明したように、新たなモジュール化手法を提案するときに、静的な構文拡張だけを利用するのは限界がある。この問題はプログラミング言語が1次元の文字列で表されるためである。そのため、AspectJ のように従来より良いモジュール化が提案されたとしても、AJDT のような言語以外のものを利用して開発者をサポートしなければならない。

そこで、本研究では、静的なテキストだけに依存するのではなく、動的なテキストを扱い、コード再利用のためのモジュール化を行う方法について提案する。動的なテキストは、静的なテキストと異なり、コードの編集中にその編集箇所と異なる場所が自動的に変更される。動的なテキストを扱うために、本研究では統合開発環境（IDE）を活用する。また、ここでいうコードとはクラスやメソッドなどのモジュール単位ではなく、式や文などの任意の文字列を表す。

以下、3.1 節で本提案の元となるコピー＆ペーストを用いたコードの再利用について述べ、節で本システムの概要について説明する。

3.1　コピー＆ペーストを利用したコードの再利用

コードを再利用するための方法の一つにコピー＆ペーストがある。たいていのエディタでは、プログラマは再利用したいコードを選択してクリップボードにコピーし、別の場所にペーストすることができる。この方法は便利であり直感的で使いやすいため、プログラミングに慣れていない初心者はついつい使用してしまいがちである。

しかし、コピー＆ペーストを使用して開発を進めていくと、コードの再利用性が低下する原因になりうる。なぜならペーストされたコードの内容はコピー元のコード内容に依存するからである。もし、コードを修正する必要ができたときには、コピー＆ペーストされたコードを全てチェックして編集しなければならず、保守性が低下してしまう。

3.2　同期可能なコピー＆ペーストシステム

本研究では、コピー＆ペーストの直感的で使いやすい利点を残しつつ、保守性が低下してしまう欠点を補うために、同期可能なコピー＆ペーストシステムを提案する。同期可能なコピー＆ペーストができる環境を、統合開発環境である Eclipse のプラグインとして実装した。

本システムは簡単に使用することができる。ユーザがあるコードの再利用をしたいときには、通常のテキストをコピー＆ペーストするときと同様に、その再利用したいコードを本システムが提供する特別なコマンドを用いてコピーし、使用したい場所にペーストするだけで良い。この方法でコピー＆ペーストされたコード領域をユーザが編集すると、本システムが同期を行い、他方のコードも自動的に変更する。同期が行われると、複製された全てのコード領域が同じコードとなる。通常のコピー＆ペーストと異なり、複製したコードを書き換えたときに、もう一方を手動で探して手直しする必要はない。自動的に同期することにより、複数のモジュールに散在するコードが一箇所に書かれているかのように扱うことができ、擬似的なモジュール化が可能となる。

図 3.1 は本研究で提案するシステムの全体図である。本システムで使用するのは、ツールバー、Concerns ビュー、エディタの 3 つに分けられる。ツールバーには、本システムを利用するためのコマンドが置かれている。Concerns ビューにはモジュールのリストが表示されている。エディタでは、開発者がソースコードを書くことができる。

3.2.1　ツールバー

本システムのツールバーは Eclipse の上部に位置する（図 3.2）。ツールバーには 3 つのコマンドがある。1 つ目はコピー、2 つ目はペースト、3 つ目はパラメータ化である。以下で、それぞれのコマンドについて説明する。

コピー

コピーは、開発者にとって再利用したいコードが生じたときに使用する。たいていのエディタで使用することができるコピー＆ペーストと同様に、開発者は再利用したいコードを選択し、このコマンドを用いてコピーを行う。すると、本システムが選択されたコードの文字列と共に、そのソースコードファイルのパスと、オフセットを記録する。オフセットとは、ソースコードファイルの始めから何文字目でそのコードが使用されているのかを表す。

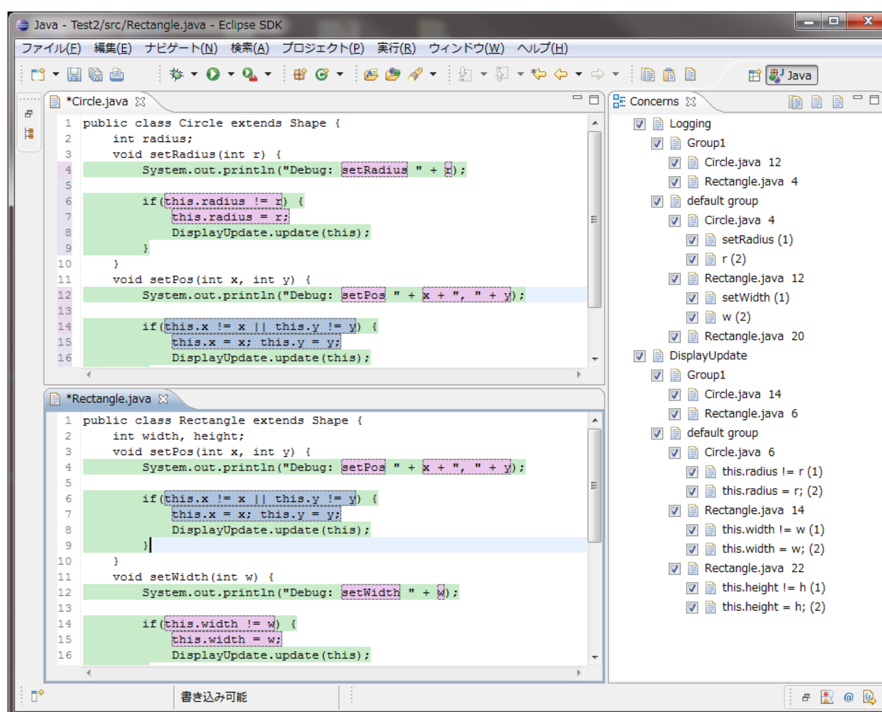

図 3.1: 同期可能なコピー&ペーストシステムの全体図

ペースト

ペーストは、開発者が再利用したいコードを貼りつけたいときに使用する。通常のコピー&ペーストと同様に、コピーしたコードを使用したい場所にカーソルを置き、本コマンドを使用する。すると、コピーしたコードが挿入されると同時に、本システムがペーストされたソースコードファイルのパスとオフセットを記録する。

パラメータ化

本システムを利用してコピー&ペーストしたコードの中に、パラメータとして使用したい場所がある場合、本コマンドを使用する。本システムを利用してコピー&ペーストしたコード領域の中からパラメータとして使用したいコードを選択し、本コマンドを使用する。コードを選択する方法は、コードをコピーするときと同様である。選択されたコードはパラメータとして扱われ、本システムがそのパラメータのオフセットを記録する。

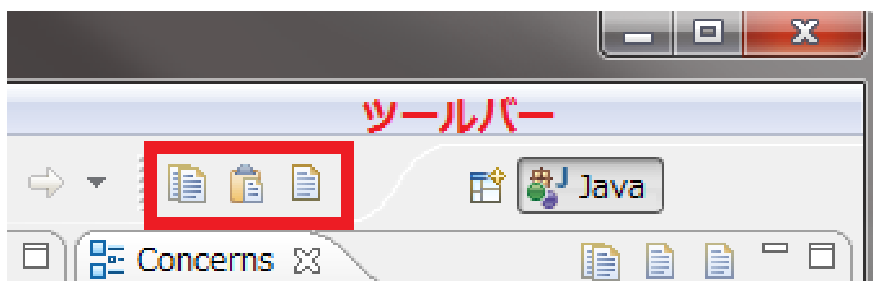

図 3.2: ツールバー

3.2.2　Concerns ビュー

Concerns ビューには本システムを用いてコピー&ペーストされたコードの一覧を表示する（図 3.3）。

Concerns ビューは、コードに対する名前、グループ、コードの使用場所、パラメータの 4 つの階層で表示する。

- コードに対する名前
  1 つ目はコードに対する名前である。コードに対する名前は、デフォルトではコピーを行った時のコード内容がそのまま表示される。そのコードをダブルクリックすることにより、Concerns ビューに表示する名前をユーザが定義することができる。この名前は、コピーしたオリジナルのコードと、そのコードからペーストされたコードに対してのものである。

- グループ
  2 つ目はグループを表す。コピー&ペーストによって複製されたコードをいくつかのグループに分けることができる。デフォルトでは default group にまとめられるが、後述するコマンドを使用することにより他のグループを作ることができる。

- コードの使用場所
  3 つ目はコードが使用されているファイル名と、そのコードがファイルの何行目で使用されているのかを示す数が表示されている。本システムが提供するコピー&ペーストによって複製された領域の数だけ、この階層に表示される。また、この階層において開発者が右クリックすることによりポップアップメニューが表示され、そのメニューからコード記録の削除を行うことができる。コード記録を削

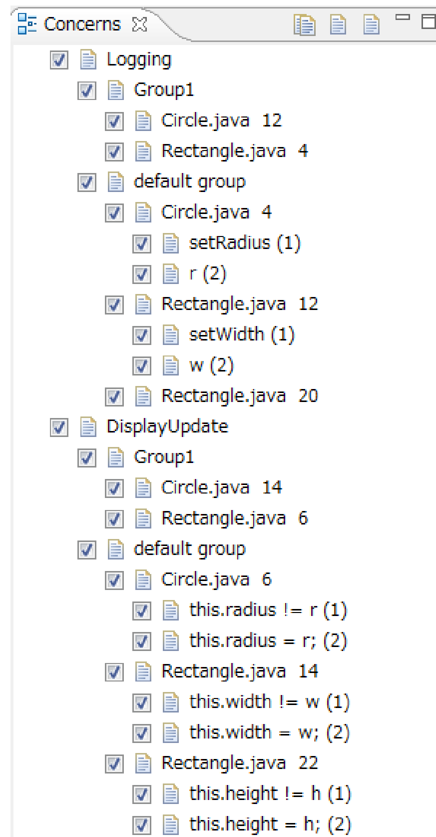

図 3.3: Concerns ビュー

除すると、本システムが記憶しているコードの位置情報などが失われるが、実際のコードは削除されない。

- パラメータ
  4 つ目はパラメータと、そのパラメータが何番目に使われているのかを表示している。デフォルトではパラメータの内容がそのまま表示されているが、パラメータをダブルクリックすることでそのパラメータに対して名前をつけることが可能である。

また、Concerns ビューには 3 つのコマンドが用意されている。1 つ目は再コピー、2 つ目はパラメータ同期、3 つ目はグループ化である。

再コピー

再コピーは以前コピーもしくはペーストされ、本システムが記録しているコードを、エディタ上で再び選択することなくコピーすることができるコマンドである。3 つ目の階層にあたる、コードが使用されているファイル名と使用されている行番号が表示されている部分を選択し、このコマンドを実行することで再びコピーすることができる。再コピー後は、本システムが提供するペーストコマンドを利用することでコードの複製をすることができる。

パラメータ同期

コピーもしくはペーストしたコードの中で同じパラメータが複数回出現する場合、それらをまとめることができる。同じ文字列のパラメータを Concerns ビュー上で複数個同時に選択し、本コマンドを実行することでまとめることができる。本コマンドを使用してパラメータをまとめると、Concerns ビュー上でそのパラメータは 1 度しか表示されない。例えば、radius というパラメータが 2 回使用される場合、Concerns ビュー上でその 2 つを選択し、本コマンドを実行することでビュー上の表示は、radius(1, 3) のようにまとめられる。この表示は、radius というパラメータがコード領域中の 1 つ目と 3 つ目のパラメータであることを示す。

グループ化

Concerns ビューの 2 階層目にあたるグループを作ることができるコマンドである。Concerns ビュー上で default group に属するコードリストの中から 2 つ以上選択し、本コマンドを実行することで新しいグループを作ることができる。

3.2.3　エディタ

エディタでは、開発者がソースコードを編集することができる。通常のテキストエディタとしての機能に本システムの機能が加わっている。本システムを利用してコピー＆ペーストされたコードであることは、エディタ上の背景色を変更することによって表される。背景色には、緑、ピンク、青の 3 色ある。

- 緑色の背景色
  緑色の背景色は、コードが同期関係にあることを示す。この領域を

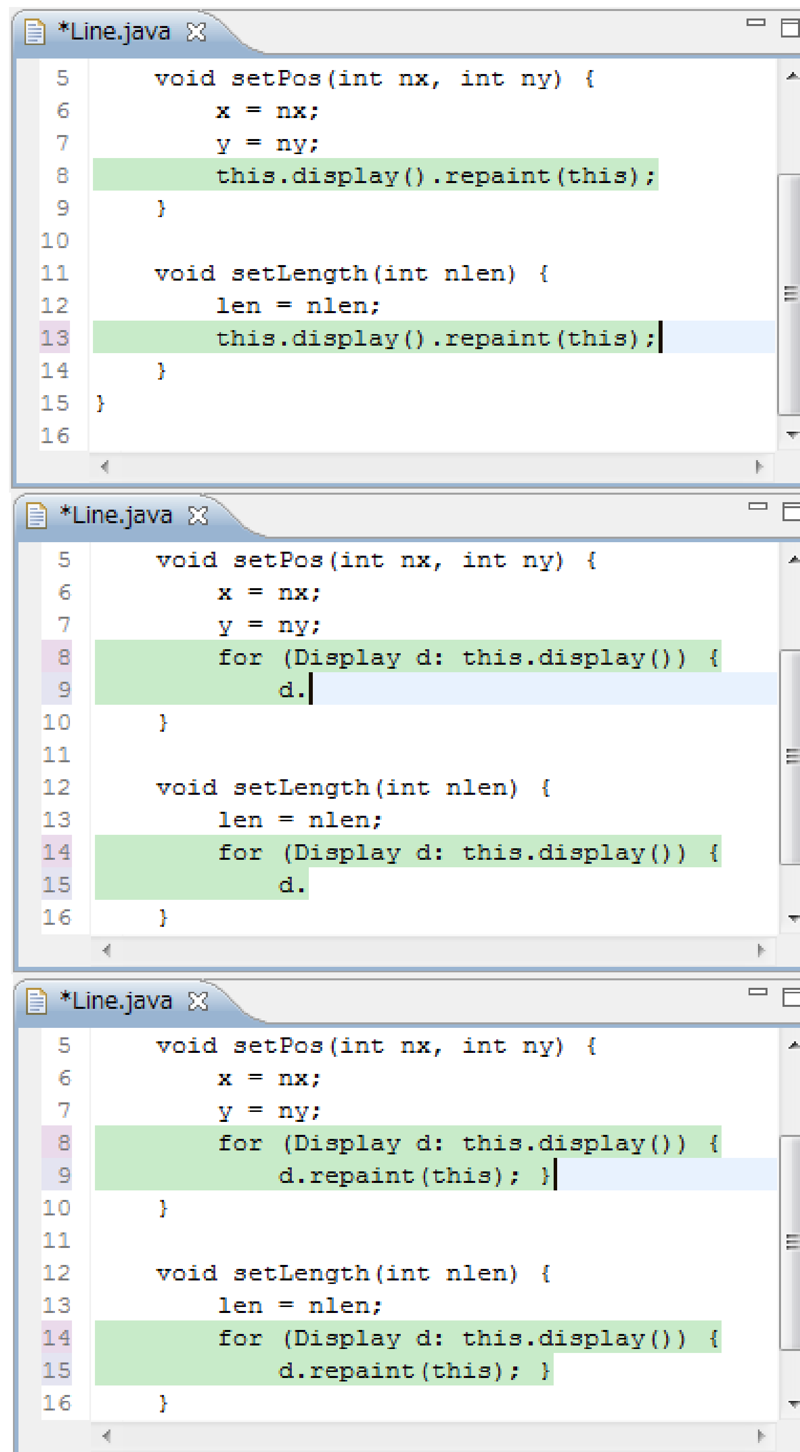

図 3.4: 同期領域の編集

開発者が編集すると、本システムが同期を行い、そのコードと関連付けられたほかのコード領域も自動的に編集される。これにより、コピー&ペーストによって複製されたコードを一度の修正ですべて編集することが可能となる。例えば、本システムを用いると、図 3.4 のように開発することができる。図 3.4 は、1 つ目のエディタがコピー&ペーストによってコードが複製され、同期されている状態である。2 つ目と 3 つ目のエディタでは、ユーザが setPos メソッド中のコードを書き換え、その変更が本システムによって setLength メソッド中のコードも同時に編集されている様子を表している。

- ピンク色の背景色
  ピンク色の背景色は、パラメータであることを示す。緑色の同期領域と異なり、本システムを利用して複製されたコードであっても、この領域の編集は他の領域に同期されない。すなわち、複製したコードの中で他の領域と異なる部分を作り出すことができる。

- 青色の背景色
  青色の背景色もパラメータを表す。ただし、パラメータが青色に変わるのは、その領域が Concerns ビューによってグループ化された領域（緑色もしくはピンク色の背景色の領域）であり、かつそのコード領域上にカーソルがある場合のみである。領域上にカーソルがない場合は、他の場合と同様にパラメータはピンク色である。ピンク色の領域と同様にパラメータを表すが、異なるのは、この領域を編集すると他のグループ化された領域のみそのコードが同期される点である。

パラメータの表現方法はもう 1 つある。それは図 3.5 のようにパラメータ領域に枠付けすることで表現される。図 3.5 では、緑色の領域が 2 つある。その中で、1 つ目の領域では 3 つの mhp と 2 つの p が、2 つ目の領域では 3 つの mhq と 2 つの q がそれぞれ枠付きのパラメータである。枠があるパラメータでは、Concerns ビューによって関連付けられた同じパラメータ同士が同期される。すなわち、1 つ目の領域のうちどこか 1 つの変数 mhp を例えば maxHeightP に変更したとすると、他の 2 つの mhp も同時に maxHeightP に修正される。この変更は下側のコード領域に伝わることはない。枠の色は対応するパラメータの数に応じて変化する。

以上のエディタ上でのコードの表現方法をまとめると、表 3.1 のようになる。

各領域の境目の編集は、そのカーソル位置がある直前の領域の編集とみなす。例えば、緑色の領域とピンク色の領域が順に並んでいて、その境界を編集したとすると、緑色の領域が編集された扱いとなる。また、背景

```
int mhp = g.getMaximumHeight();
g.setHeight(mhp < h ? mhp : h);
if (q == null) { return; }
int mhq = q.getMaximumHeight();
q.setHeight(mhq < h ? mhq : h);
```

図 3.5: パラメータの同期

表 3.1: エディタ上でのコード表現

領域の色	説明
緑	同期領域
ピンク（枠なし）	パラメータ
ピンク（枠あり）	パラメータ。ただし、同一領域内でパラメータが同期される。
青	パラメータ。ただし、グループ化された領域上にカーソルがある場合。他のグループ化された領域のパラメータを同期する。

色が変更されているすべての領域は、通常の編集で完全に削除することはできない。領域を削除したい場合は、Concerns ビューのポップアップメニューから同期情報を削除し、その後通常の編集で削除する必要がある。各領域の背景色を表示させないことも可能である。Concerns ビューに用意されているチェックボックスのチェックを外すことで背景色は表示されない。ただし、表示されないだけであり、前述のような同期処理は行われる。

第4章　実装

本章では、第 3 で提案した同期可能なコピー＆ペーストシステムの実装方法について述べる。本システムは Eclipse のプラグインとして実装した。4.1 節では、Eclipse プラグインの基本的な実装方法について、4.2 節では、本システムの実装方法について述べる。

4.1　Eclipse プラグインの実装方法

Eclipse は統合開発環境の一つであり、主に Java での開発に用いられる。本節では、Eclipse の拡張を行うために必要な基礎知識と、基本的な実装方法について述べる。

4.1.1　Eclipse のアーキテクチャ

Eclipse では、プラグイン・アーキテクチャを採用しており、様々な機能を自由に追加することができる。Eclipse はプラグインを管理するために OSGi を使用している。OSGi とは、Java のモジュールを動的に追加したり、その実行を管理することができる基盤システムである。Eclipse において OSGi はプラグイン・アーキテクチャの最下層に位置している。OSGi の上位にあるその他の構成要素はすべてプラグインとして提供されている。そのため、Java 用の統合開発環境 JDT[9] などもプラグインとして提供されている。Eclipse のプラグインは Java のモジュールとして開発されるため、OSGi を利用することで動的に拡張することが可能となる。

各プラグインは、それぞれ別のプラグインから拡張できるようにするため拡張ポイントと呼ばれるものを提供することができる。拡張ポイントとは、プラグイン同士を接続するために使われ、これを使うことによってプラグインに機能を追加することができる。拡張ポイントを使ったプラグイン同士の接続は、実行時に行われる。そのため、拡張ポイントを提供する側のプラグインではどのような拡張が行われるかを把握する必要がない。拡張ポイントには、メニューを追加するものやビューを追加するものなどが含まれる。

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?eclipse version="3.4"?>
3  <plugin>
4     <extension point="org.eclipse.ui.editorActions">
5           <editorContribution
6                 id="action.editorContribution1"
7                 targetID="JavaEditor">
8                 <action
9                       class="javaeditor.actions.CopyAction"
10                      icon="icons/copy_edit.gif"
11                      id="CopyAction"
12                      label="Copy"
13                      menubarPath="edit/additions"
14                      style="push"
15                      toolbarPath="additions"/>
16          </editorContribution>
17    </extension>
18 </plugin>
```

図 4.1: plugin.xml の例

4.1.2　マニフェスト・ファイル

マニフェストファイルには、MANIFEST.MF と plugin.xml の 2 つのファイルがある。MANIFEST.MF はプラグイン ID 、プラグインの名称、開発者名、他のプラグインとの依存関係などが記述されている。plugin.xml にはプラグインの拡張、拡張ポイントの定義などを XML を用いて記述されている。

図 4.1 と図 4.2 はそれぞれ plugin.xml、MANIFEST.MF ファイルの記述例である。これは、同期可能なコピー＆ペーストシステムで実際に記述したものである。plugin.xml の extension タグでは拡張ポイントを指定している。ここでは、記述を省略し、拡張ポイントを 1 つしか載せていないが実際は使用した拡張ポイントの数だけ extension 記述がされている。この記述をすることで、既存プラグインを拡張することが可能となる。MANIFEST.MF ファイルでは、プラグインのバージョンや他のプラグインとの依存関係などが記述されている。

Eclipse にはマニフェスト・ファイルを簡単に編集することができるプラグイン開発環境 PDE （Plugin Development Environment）も提供されている。PDE では図 4.3 のように、前述の 2 つのマニフェスト・ファイルをグラフィカルに編集することができるマニフェスト・エディタが提供されている。マニフェスト・エディタはいくつかのタブページを切り替えながら編集することができる。各タブページの内容を表 4.1 に示す。なお、図 4.3 では、Extensions タブページが表示されている。

```
 1 Manifest-Version: 1.0
 2 Bundle-ManifestVersion: 2
 3 Bundle-Name: MyEditorPlug-in
 4 Bundle-SymbolicName: CopyEditor;singleton:=true
 5 Bundle-Version: 2011.03.31
 6 Bundle-Activator: javaeditor.JavaEditorPlugin
 7 Require-Bundle: org.eclipse.ui,
 8  org.eclipse.core.runtime,
 9  org.eclipse.jface.text,
10  org.eclipse.ui.editors,
11  org.eclipse.ui.views;bundle-version="3.5.0",
12  org.eclipse.core.resources;bundle-version="3.7.100",
13  org.eclipse.ui.ide;bundle-version="3.7.0"
14 Bundle-ActivationPolicy: lazy
15 Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

図 4.2: MANIFEST.MF の例

4.2　同期可能なコピー＆ペーストシステムの実装

前節では基本的な Eclipse プラグインの実装方法について述べた。本節では、同期可能なコピー＆ペーストシステムの第 3 章で述べた基本機能であるエディタ、ツールバー、Concerns ビューの 3 つの機能について実装方法を述べる。

4.2.1　エディタの実装

Eclipse のテキストエディタは JFace Text フレームワークに基づいている。JFace Text は org.eclipse.jface.text パッケージで提供されている。ビューアーとして ITextViewer インタフェース、それを拡張した ISourceViewer インタフェース、それらを実装したクラスなどを提供している。これらのビューアーの実装は、SWT の StyledText ウィジェットをラップしている。Eclipse でテキストエディタを実装する場合、基本的には ITextViewer ではなく ISourceViewer を利用する。

Eclipse にエディタを追加する場合、org.eclipse.ui.editors という拡張ポイントを使用する。この拡張ポイントでは、表 4.2 のような項目を主に設定できる。

本システムでも、この org.eclipse.ui.editors 拡張ポイントを利用してエディタの実装を行った。本エディタは Eclipse からすでに用意されている TextEditor クラスを継承し、実装した。TextEditor クラスは、標準的なテキストエディタの機能を持つ。以下、本エディタに追加した機能の実装方法について述べる。

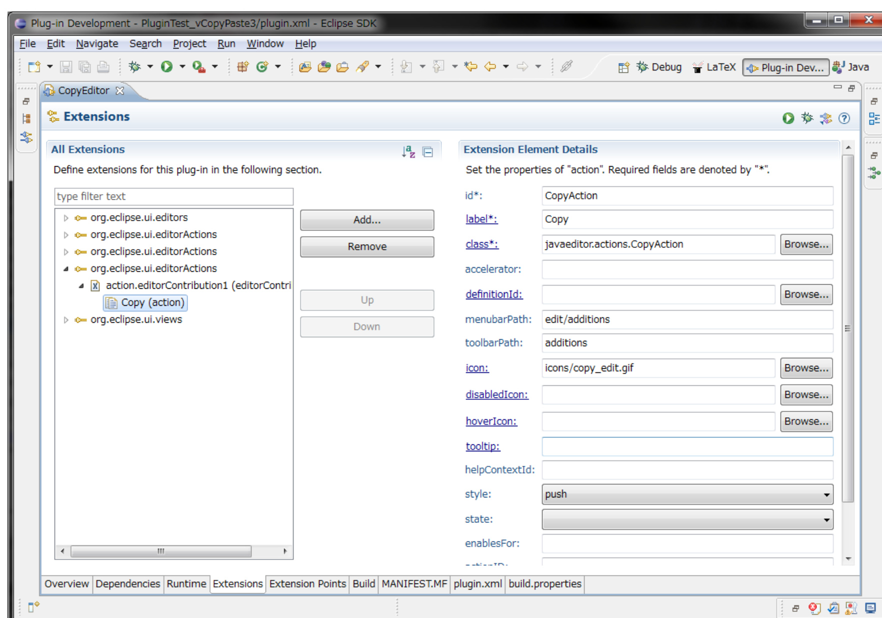

図 4.3: PDE

同期領域の背景色描画処理

同期領域の背景色描画処理は、TextEditor が持つ StyledText クラスに StyleRange というテキスト上の指定した範囲のフォント等を変更できるオブジェクトを追加することで実現した。StyleRange クラスには、その範囲の開始地点とその場所からの長さ、その範囲中の背景色や枠の形、色などを設定することができる。

背景色描画処理が実行された時、本システムが記録しているすべての同期領域情報を走査し、開いているエディタ毎に描画範囲を計算し、その StyleRange のリストを作成する。また、StyleRange を作成するときに、同期領域の状態に応じて背景色を緑色やピンク色に設定する。作成された StyleRange のリストを StyledText にセットし、アップデートすることで背景色が再描画される。

この背景色描画処理は、ソースコード編集中やコピー、ペーストなどが行われるたびに実行される。

同期処理

同期領域編集時に他の領域を同時に編集する処理は、TextEditor がもつ IDocument オブジェクトに IDocumentListener インタフェースを実装した

表 4.1: マニフェスト・エディタのタブページとその内容

ページ	説明
Overview	プラグイン全体の情報を設定できる。設定できる内容には、プラグインの ID、バージョン、名前などがある。また、開発中のプラグインを含めた Eclipse のランタイム・ワークベンチを起動できる。
Dependencies	プラグインの依存関係を指定できる。
Runtime	プラグインがエクスポートするパッケージ、クラスパスなどを設定できる。
Extensions	他のプラグインが提供している拡張ポイントを利用するときにその内容設定できる。
Extension Points	開発中のプラグインが他のプラグインに提供するための拡張ポイントを指定できる。
Build	ビルドするときにアーカイブに含めるファイルなどを設定できる。
MANIFEST.MF	MANIFEST.MF ファイルの中身を表示できる。
plugin.xml	plugin.xml ファイルの中身を表示できる。
build.properties	build.xml ファイルの中身を表示できる。

クラスを追加することで実現した。IDocumentListener インタフェースは、エディタが持つドキュメントが編集されるたびに実行される。ドキュメントが編集される直前に呼び出される documentAboutToBeChanged メソッドとドキュメントが編集された直後に呼び出される documentChanged メソッドが用意されている。この IDocumentListener でソースコードの編集を感知し、その編集された場所が同期領域の中なら他の領域情報も同時に更新し、そうでなければ記録している同期領域の場所を再計算する処理を追加している。本システムが内部で記録している他の同期領域の情報が更新されると、同時にソースコードファイルも編集する。また、このリスナーが動作した後は、再び同期領域の背景色描画処理を実行する。

特定条件下での背景色変更処理

本システムでグループ化された領域の表現方法として、その領域上にカーソルがある場合のみパラメータの背景色を変更する処理がある。この処理の実装には、MouseListener と KeyListener インタフェースを利用した。これらを実装したクラスを TextEditor に追加することで、それぞれマ

表 4.2: org.eclipse.ui.editors 拡張ポイントで設定できる主な項目

属性	説明
id	エディタの ID
name	エディタの名前
extensions	エディタに関連付けるファイルの拡張子
class	エディタを実装したクラス

表 4.3: org.eclipse.ui.editorActions 拡張ポイントで設定できる主な項目

属性	説明
id	アクションの ID
label	マウスカーソルを合わせたときに表示されるアクション名
class	アクションを実装したクラス

ウスクリックされたとき、キーボードから何らかの入力があったきの処理を追加することができる。本システムでは、これを利用し、マウスクリックされたとき、キーボードから入力があったとき共に、その入力があった時点でのエディタ上でのカーソル位置を取得し、その位置が同期領域の中なら、エディタの再描画を行うようにした。

4.2.2　ツールバーの実装

ツールバーは、org.eclipse.ui.editorActions 拡張ポイントを利用して実装した。この拡張ポイントはツールバーにアクションを追加するためのものである。この拡張ポイントでは、表 4.3 のような項目を主に設定できる。この表の他に、editorContribution タグの targetID という項目でこのアクションを表示させる対象であるエディタを指定することができる。

本システムでは、コピー、ペースト、パラメータ化の 3 つのコマンドがある。これら 1 つ 1 つが org.eclipse.ui.editorActions 拡張ポイントを利用している。この拡張ポイントの targetID 属性で本システムのエディタを指定している。この指定を行うと、本エディタを開いているときしかアクションを実行することができない。

以下、各コマンドの実装について述べる。

コピーコマンド

コピーコマンドは、実行された時、TextEditor から現在選択されているテキストとその位置を取得する。その後、そのファイル名やパスを取得し、一つのオブジェクトにまとめる。まとめられたオブジェクトを本システム内部で記憶し、ディスプレイ再描画処理を行う。また、そのオブジェクトをペースト対象のオブジェクトとして登録する。実行時に選択されている文字列がなかった場合は、そのまま処理を終了する。

ペーストコマンド

ペーストコマンドは、実行された時、ペースト対象のオブジェクトを取得する。その後、TextEditor から現在のカーソル位置やファイル名、パスを取得する。コピーの時と同様に、それらのデータをひとつのオブジェクトにまとめ、本システム内部で記憶する。TextEditor が持っている IDocument オブジェクトを修正し、カーソル位置があった場所にペーストするコードの文字列を挿入する。最後にディスプレイ再描画処理を実行し、本コマンドの処理を終了する。なお、ペースト位置がすでに他の同期領域の中だった場合、ペースト処理を実行せずに終了する。

パラメータ化コマンド

パラメータ化コマンドは、実行された時、TextEditor から現在のカーソル位置と選択されている文字列の長さを取得する。システム内部で記録しているすべての同期領域オブジェクトの中から、選択された文字列すべてが同期領域に含まれるようなオブジェクトを取得する。該当するオブジェクトが見つかれば、選択されている文字列部分のみをパラメータとして扱う処理を実行し、ディスプレイ再描画処理をした後終了する。また、選択されている文字列がなかった場合や、パラメータを作れるような同期領域オブジェクトが見つからなかった場合は、そのまま処理を終了する。

4.2.3　Concerns ビューの実装

Concerns ビューは、org.eclipse.ui.views 拡張ポイントを利用して実装した。この拡張ポイントはビューを追加するときに使われる。この拡張ポイントでは、表 4.4 のような項目を主に設定できる。

表 4.4: org.eclipse.ui.views 拡張ポイントで設定できる主な項目

属性	説明
id	ビューの ID
name	ビューの名前
class	ビューを実装したクラス

ビューアーの構築

ビューは JFace のビューアーを使用する。ビューアーは、ビューアーに表示させるデータモデルをコンテンツプロバイダーとラベルプロバイダーで管理し、それぞれのプロバイダーで取得した情報を表示させる。

- コンテンツプロバイダー
  コンテンツプロバイダーはビューアークラスからの通知を元に、モデルを提供する。JFace ではあらかじめ用意されているコンテンツプロバイダークラスがいくつかあり、テーブルビューアーには ArrayContentProvider、ツリービューアーには TreeNodeContentProvider などが提供されている。

- ラベルプロバイダー
  ラベルプロバイダーはビューザークラスから指定されたモデルに対応する表示内容を提供する。JFace であらかじめ用意されているラベルプロバイダーには、LabelProvider がある。

- モデル
  モデルには、任意の Java オブジェクトを使用することができる。

Eclipse にビューを追加する場合、org.eclipse.ui.views 拡張ポイントを利用する。Concerns ビューは、この拡張ポイントを利用し、ビューの構成には CheckboxTreeViewer クラスを使用した。このクラスは JFace で提供されており、チェックボックス付きのツリー構造を表すことができる。コンテンツプロバイダーを通じて、本システムが記憶している同期領域の情報を元にモデルを構築し、ビューアーに渡す。ビューアーはこのモデルを元にラベルプロバイダーを通じてビューに表示する内容を取得する。ラベルプロバイダーからは、コードに対する名前やコードの使用場所を表すファイル名、行数などの表示内容を取得できる。

ビューアーで使用されるコマンドの実装

ビューアーにツールバーを追加することができる。この拡張には拡張ポイントを使用しない。ビューアーから IActionBars を取得し、そのバーに IAction を実装したクラスを追加することで実装できる。

- 再コピーコマンド
  再コピーコマンドが実行された時、ビューアーから TreeSelection オブジェクトを取得する。TreeSelection では、ビューアーから現在選択されているモデルの情報を取得することができる。これを通して取得したモデルが同期領域の情報であった場合、再びペーストできるように本システムの内部状態を変更する。

- パラメータ同期
  再コピーコマンドと同様に、実行された時ビューアーから TreeSelection オブジェクトを取得する。TreeSelection オブジェクトから取得できる現在選択されているモデルが 2 つ以上ありかつパラメータの情報である場合、本システム内部において選択されたパラメータを関連付けている。

- グループ化
  グループ化コマンドも他のコマンドと同様に、実行された時ビューアーから TreeSelection オブジェクトを取得する。TreeSelection オブジェクトから取得できる現在選択されているモデルが以下の条件をすべて満たしている場合、本システム内部において同期領域のグループを作成する。
  - モデルの数が 2 つ以上である。
  - すべてのモデルが同期領域の情報である。
  - すべての同期領域が他のグループに属していない。

上記 3 つのコマンドとは別に、本システムが記録している同期領域の情報を削除するコマンドもある。このコマンドはユーザが右クリックしたときに現れるポップアップメニューとして実装されている。ポップアップメニューは、MenuManager クラスを使用して実装される。MenuManager クラスに、IMenuListener インタフェースを実装したクラスを追加し、そのクラス中でメニューとして表示するアクションを登録する。登録するアクションの実装方法は、ビューアーのツールバーで使用するアクションと同じである。同期領域の削除コマンドが実行された時、TreeSelection オ

ブジェクトを取得し、そのオブジェクトから取得できるモデルが同期領域の情報であった場合、本システムが記録している該当するデータを削除する。

チェック状態の変更

前述の通り、本システムはチェックボックス付きのツリー構造を表現できるビューアーを提供している。ここでは、チェックボックスのチェック状態を変更した時の処理について述べる。

チェック状態の変更は、ビューアーに ICheckStateListener を実装したクラスを追加することで感知できる。本システムでは、あるモデルに対してチェックがオンの状態であればエディタの背景色を描画し、オフの状態であれば描画しない仕様である。ユーザの操作によりチェックが切り替われば、モデルの描画状態を変更し、エディタを再描画する。

4.2.4 内部モデルの実装

ここまでで、ユーザインタフェースに関連する部分の実装方法について述べた。ここでは、同期領域の情報を管理するための内部モデルの実装方法について述べる。

同期情報は、主に 4 つのクラスを使って管理する。すべての同期領域の情報は、SyncroRelation クラスにまとめられる。この SyncroRelation は SyncroMemberGroup オブジェクトのリストをもっている。SyncroMemberGroup クラスは、コピー＆ペーストによって複製された領域の対応関係を表す。例えば、ログ出力文をコピー＆ペーストによって複製すると、SyncroMemberGroup のオブジェクトが一つ作られ、その中に領域の情報が登録される。各領域の情報は SyncroMember が持つ。また、SyncroMember はいくつかの PartSyncroMember に分割して情報を管理している。PartSyncroMember クラスは、パラメータ化などによって分割された領域の数だけオブジェクトが作られる。例えば、System.out.println(“rectangle”); というコードがコピー＆ペーストによって複製され、そのうち rectangle の部分だけをパラメータとして設定したとすると、System.out.println(“、rectangle、”); という 3 つの PartSyncroMember オブジェクトが作られ、この 3 つを SyncroMember クラスで管理する。この PartSyncroMember クラスには、コードの内容や使用される場所、同期状態などの情報を格納できる。これらの対応関係を図 4.4 に示す。

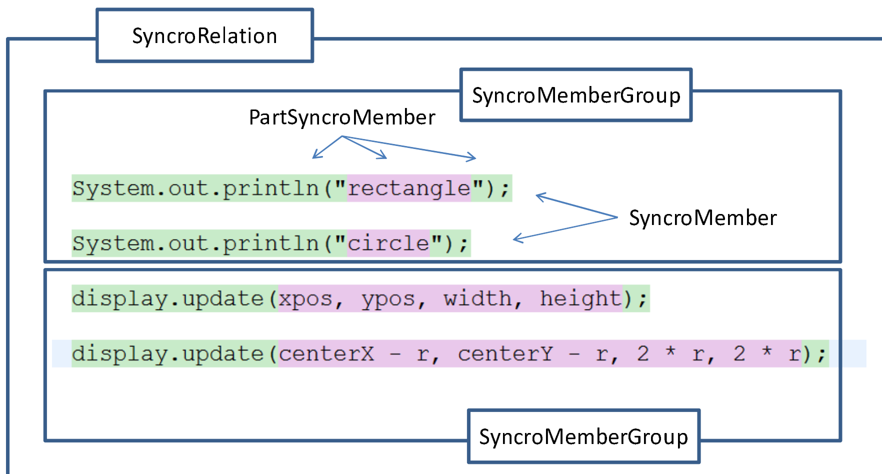

図 4.4: 内部モデルのオブジェクト

第5章　評価

同期可能なコピー＆ペーストは、プログラミング言語において動的なテキストを扱うシステムであった。また、動的なテキストを導入することにより、本システムを用いて様々な言語機構を表現できるようになった。本章では、従来の言語機構を利用したモジュール化方法を、第3章で提案した同期可能なコピー＆ペーストシステムを用いてどの程度表現できるのか検証する。評価対象とする従来の言語機構を利用したモジュール化方法は、手続き抽象、パラメータ抽象、アスペクト、カリー化である。

5.1　手続き抽象

開発者がプログラミングしていると、ある処理に関するコードを何度も使用したい場合がある。そのような処理を処理内容ごとにまとめたものを手続きと呼ぶ。手続きの他に、メソッドや関数などの呼び方がある。手続きとしてまとめておくと、例えばメソッド呼び出しを記述するだけでその処理内容を再利用することができる。

Java 言語で手続き抽象をしたコードの例としては、図 2.1 のコードが挙げられる。図 2.1 の Rectangle クラスにおいて、各メソッド中に出現する DisplayUpdate.update メソッドの呼び出しは、ディスプレイ再描画に関する処理をまとめた手続きである。

本システムを利用してこのコードを表現すると、図 5.1 のようになる。

これは、setPos メソッド中に定義した this.display().repaint(this); というディスプレイ再描画処理を他の setWidth メソッドや setHeight メソッドに複製したものである。図 2.1 のコードでは、ディスプレイ再描画処理は DisplayUpdate クラスにまとめられていたが、図 5.1 では各メソッド中にインライン展開されている。

双方のコードにおいて、ディスプレイ再描画処理 this.display().repaint(this); を以下のコードのように修正したいとする。

```
for (Display d: this.display()) {
    d.repaint(this);
}
```

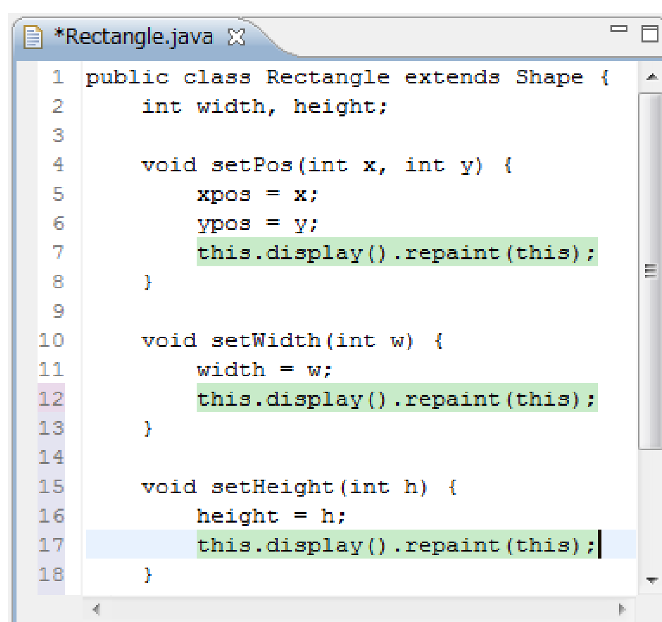

図 5.1: 手続き抽象の実現

このとき、オブジェクト指向で記述されたコードでは、再描画処理の定義を変更したい場合は DisplayUpdate.update メソッドの中だけを修正するだけで済んだ。一方、本システムにおいては、複製されたコードのうちどこか 1 箇所、例えば setWidth メソッド中の記述を修正すると、他の setPos や setHeight メソッド中のディスプレイ再描画処理も同時に編集される。よって、修正箇所数という点では本システムは言語機構と同等である。

また、一度コピーしたコードは図 5.2 のように Concerns ビューにリストアップされる。開発者は、このビューからコピーしたコードに対して名前付けを行うことができる。この図では DisplayUpdate という名がつけられている。この名前は、手続きの名前に相当する。言語機構を利用した場合、メソッド名としてコードに対して名前をつける。コードをユーザが定義した名前で管理できるという点では、どちらの場合も同じである。

Concerns ビューには過去にコピーしたコードも登録されている。開発者がこれらのコードを再利用したい場合は、Concerns ビューからコードを選択した後、コピーコマンドを実行し、再利用したい場所にペーストするだけでよい。これは、言語機構でいうと、メソッドの呼び出し記述をするのと同等である。

このように、本システムは言語機構と同じような手続き抽象が可能だが、劣っている点もある。複製されるコードは様々な場所においてインラ

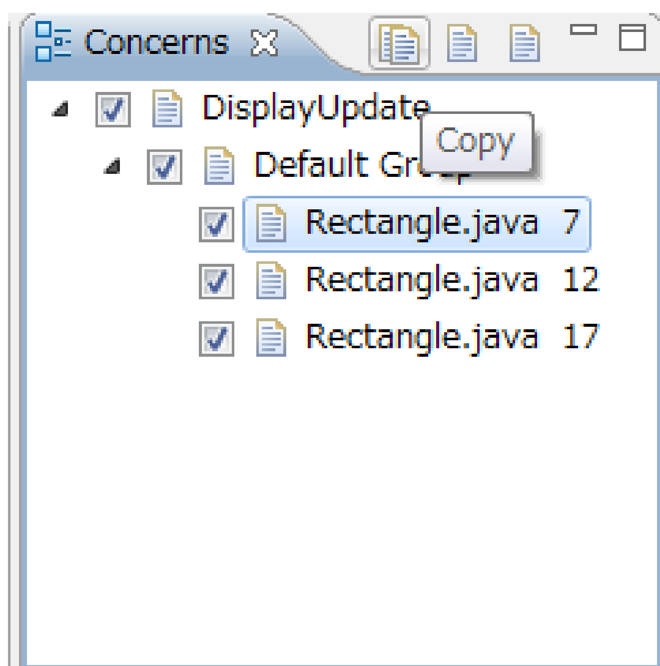

図 5.2: Concerns ビュー

イン展開されている。そのため、コード全体の行数が増加してしまう問題がある。また、例えば、Rectangle クラスの setPos メソッドのように図形オブジェクトのフィールド値を変更する処理に関するコードを開発者が読むとき、その処理と直接関わりのないディスプレイ再描画処理を読まなければならない。

さらに、言語機構での手続き抽象とは異なり、手続きを再帰的に利用することができない。言語機構では、メソッド中で自身の呼び出しを記述することで、再帰処理を行うことができるが、本システムでは、ある同期領域内に別の同期領域を作ることができないため実現不可能である。

5.2　パラメータ抽象

手続きの中にはパラメータを含むものも存在する。パラメータとは、ある手続きを実行するときにその手続に渡す値のことであり、引数とも呼ばれる。例えば、図形エディタプログラムにおいて、ディスプレイ再描画処理をするときに再描画する範囲を実行時に渡すとする。すると、図 5.3 のように表すことができる。この再描画処理には 4 つのパラメータが与えられており、1 つ目が x 座標、2 つ目が y 座標、3 つ目が x 座標からの距離、

```
1  class Rectangle extends Shape {
2      int width, height;
3      void setWidth(int w) {
4          width = w;
5          display.update(xpos, ypos, width, height);
6      }
7  }
8  class Circle extends Shape {
9      int radius;
10     void setRadius(int r) {
11         radius = r;
12         display.update(xpos − r, ypos − r, 2 ∗ r, 2 ∗ r);
13     }
14 }
```

図 5.3: パラメータ

4 つ目が y 座標からの距離を表す。このように同じ手続きでも実行箇所ごとに異なる引数を渡すことで、異なる計算結果が得られる。

本システムでも、この引数を作り出すことができる。開発者は前節に記述したように手続きを表現し、その後引数として使用したい場所を選択しパラメータ化のコマンドを実行すれば良い。本システムで実現した例を図 5.4 に示す。パラメータ部分には異なる値を設定し、その他の部分は同期できるようになっている。

5.3　アスペクト

2.1 節で述べたアスペクトも本システムを利用して表現することができる。本節では、アスペクト指向で記述された図 2.3 のコードを本システムを用いて表現する。

図 2.1 のコードは図 2.3 と同じ挙動をするプログラムであった。本システムで図 2.1 からアスペクトを表現するためには、図 5.5 のようにディスプレイ再描画処理である DisplayUpdate.update(this); のコードをコピー＆ペーストにより複製し、同期させれば良い。この変更により、ディスプレイ再描画処理の詳細は DisplayUpdate クラスに隠れたままコードの再利用性が向上している。

図 2.1 のコードでは、ディスプレイ再描画処理を図形オブジェクトから切り離したい場合、すべてのメソッド呼び出しを手動で削除しなければならなかった。しかし、本システムでは、すべてのディスプレイ再描画処理が同期されているため、どこか一箇所の呼び出し記述を削除するとすべて取り除かれる。また、本システムではエディタ上の削除は空白スペースが

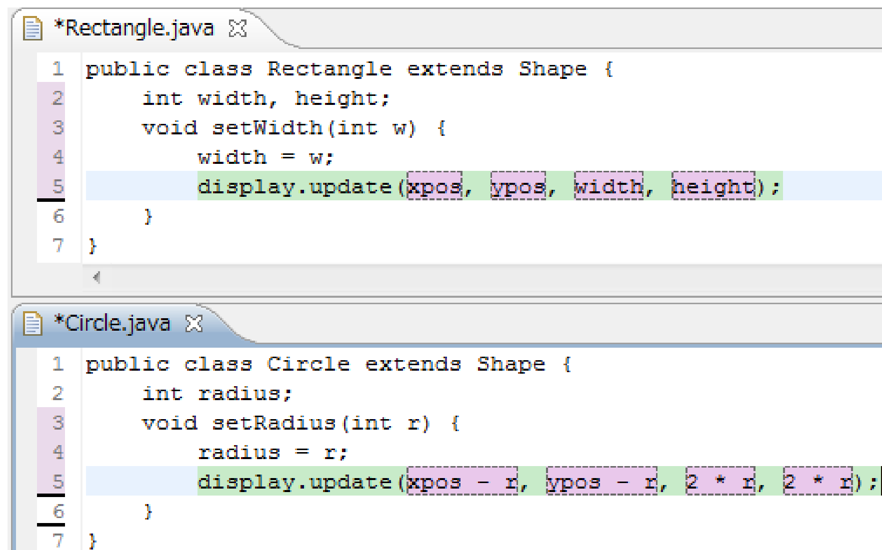

図 5.4: パラメータの実現

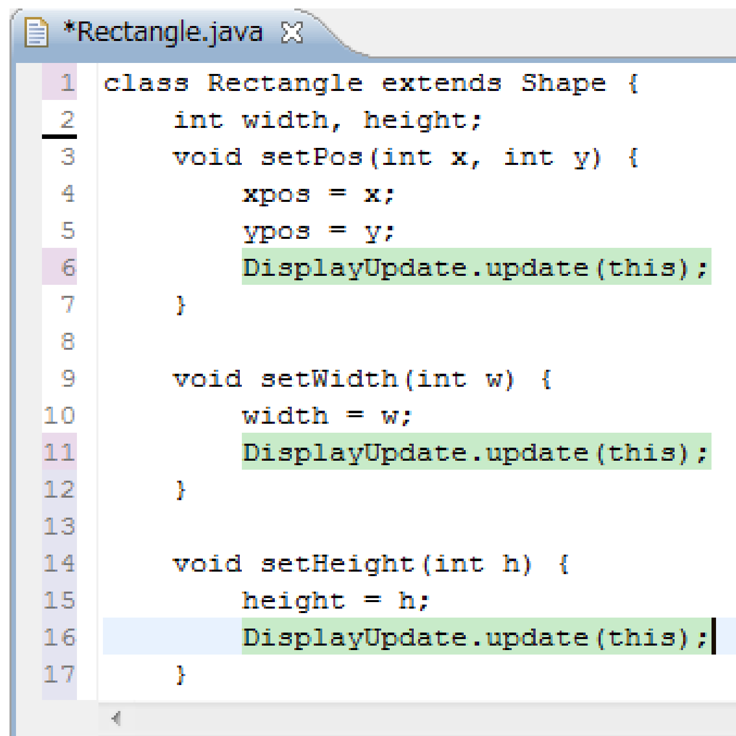

図 5.5: アスペクトの実現

1つだけ残り、その同期関係は記録されたままである。そのため、残っている空白スペースを利用すると、再びディスプレイ再描画処理をすべての箇所に簡単に挿入することができる。

本システムには、AspectJ よりも優れている点がある。AspectJ では、図形オブジェクト側からディスプレイ再描画処理に関するコードが完全に取り除かれており、AJDT のようなツールを使用しなければプログラムの動きを把握することが難しかった。一方、本システムではアスペクト指向プログラミングの利点を残しつつ、図形オブジェクトを見ただけで挙動の把握ができる。

第2章で述べたように、AspectJ にはディスプレイ再描画処理を実行する場所によって異なるパラメータを設定したい場合、ジョインポイント毎にアドバイスを記述しなければならず、似たコード数が増加する原因となっていた（図2.4)。一方、本システムを利用すると、似たコードを同期させることができるので、保守しやすくこのアスペクトを表現することができる。

本システムで図2.4のコードを実現した例を図5.6に示す。図2.4のアドバイスでそれぞれ異なるコードが記述されていた、if 文の判定式と代入文がパラメータ化されている。その他のコードは同期されているため、コードの再利用がしやすく、保守性が向上する。

また、図2.3のコードでは、ポイントカット記述を見ることによりディスプレイ再描画処理がプログラム中のどの場所で使われるのかを把握することが出来た。さらに、各ジョインポイントでどのようなパラメータでアドバイスが実行されるのかも把握することができた。一方、本システムでは図5.7のように、複製されたコードがすべて Concerns ビューに登録されている。そのため、アスペクトのポイントカット記述と同様にディスプレイ再描画処理がプログラム中のどこで使われるか把握することができる。また、パラメータも Concerns ビューにリストアップされている。アスペクトと同様に、各使用場所でどのようなパラメータでコードが実行されるのか把握することができる。

5.4　カリー化

カリー化とは、引数を複数持つような関数において、引数を1つだけ渡すと、残りの引数を持つ関数を返すことであり、Scala[2] などのプログラミング言語に含まれる技術である。

例えば Scala 言語では、 `def sum(x: Int)(y: Int) = x + y` のようにカリー化された関数を定義することができる。この関数は、x、y という2つの Int 型の引数を持ち、その2つの引数を足し算した結果を返す。

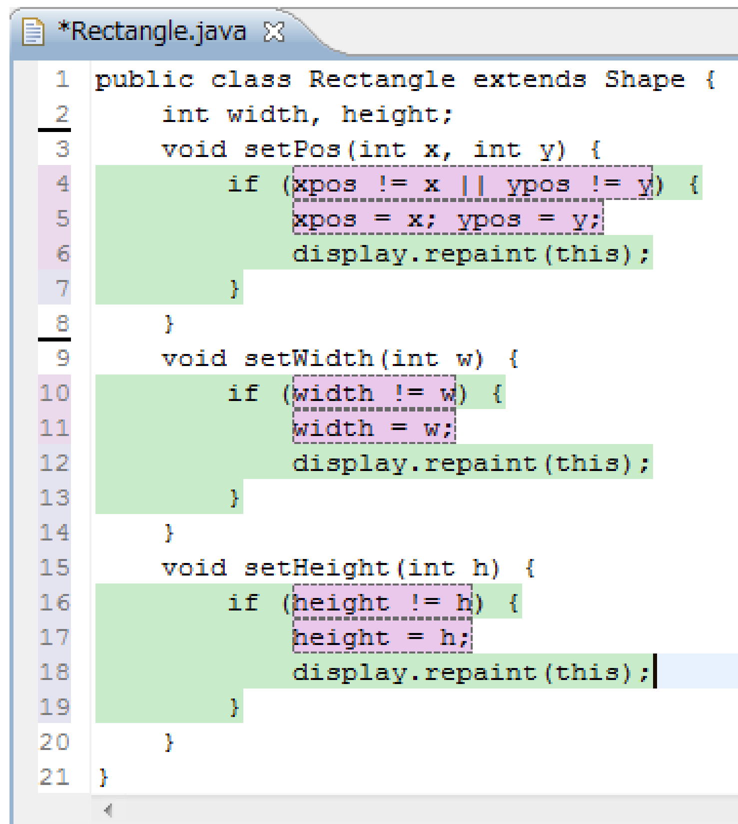

図 5.6: 複数パラメータを持つアスペクトの実現

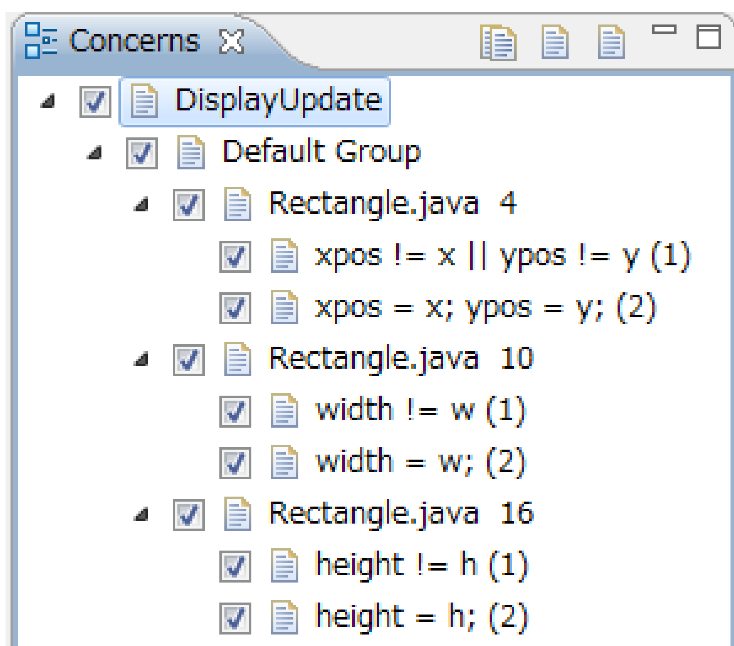

図 5.7: Concerns ビューによるアスペクトの実現

この関数を例えば sum(1)(2) のように呼び出すと、3 という結果が返される。ここまでは、普通の関数と変わらないが、カリー化された関数は呼び出すときに引数を 1 つだけ与え、sum(1) のように呼び出すことができる。このように呼び出すと、宣言されていたうち最初の引数だけが渡され、 `sum(y: Int) = 1 + y` という関数を返す。

Java 言語には カリー化技術がないが、これを含めるためには、例えば図 5.8 のような記述ができるように新しい言語機構を導入する必要がある。一方、本システムを使うと Java 言語を拡張することなく擬似的なカリー化を扱うことが出来る。本システムで実現した例を図 5.9 に示す。ここでは、Java 言語の通常のメソッド宣言を利用する。そのメソッドの呼び出し記述を同期させることにより、擬似的なカリー化が可能である。図 5.9 の 7、9、11 行目で呼び出されているコードは、それぞれ sum メソッドの引数 y にあたる部分がパラメータ化されている。x にあたる引数は同期されているため、sum(1, /* parameter */); というメソッド呼び出しは、引数 x が 1 に固定された関数であるとみなすことができる。(/* parameter */の部分には任意の数値が入る。) また、図 5.9 の 13 行目はパラメータが 2 つ設定されている。これは、Scala のカリー化された関数で言う、sum(2)(2) という呼び出しに相当する。このように、本システムを用いると Java 言語を拡張することなく、他の言語機構の利点を表現することが可能である。

```
1 int sum(int x)(int y) {
2     return x + y;
3 }
```

図 5.8: Java にカリー化を導入する際の記述例

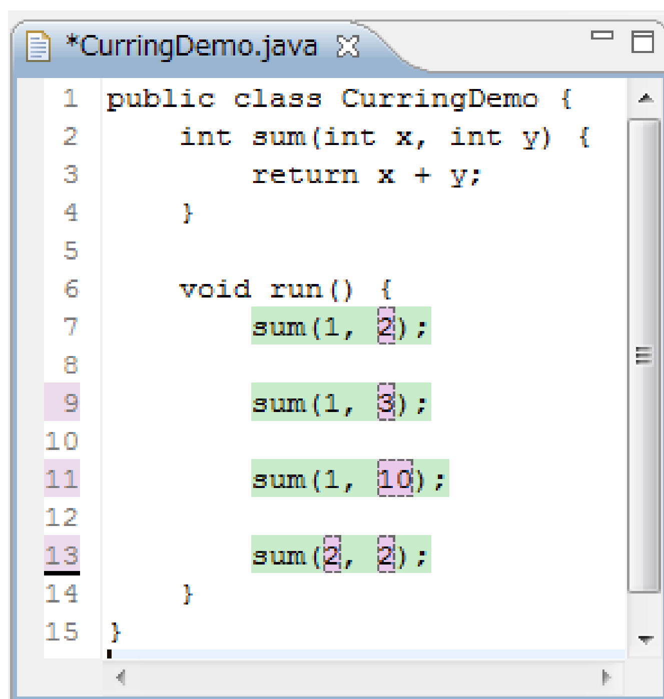

図 5.9: カリー化の実現

また、本システムではカリー化にはできないことも実現可能である。先ほどの Scala のカリー化において、sum(2) と呼び出すと、最初の引数が 2 に固定された関数が返された。しかしながら、カリー化では 2 つ目以降の引数を固定することができない。すなわち、 `sum(x: Int) = x + 2` のような関数を返すようにはできない。一方、本システムではこれが実現可能である。図 5.9 の例とは逆に、引数 x にあたる部分をパラメータ化すれば良い（図 5.10）。このように設定することにより、引数 y を固定したまま、引数 x の値を設定することができる。

Scala では、val onePlus = sum(1)_のように記述することで、カリー化された関数から新たな関数を定義することができる。この関数 onePlus は、カリー化された関数 sum の引数 x を 1 に固定した関数のことを表す。すなわち、onePlus(2) と記述することで、 1 + 2 が実行され、3 という結

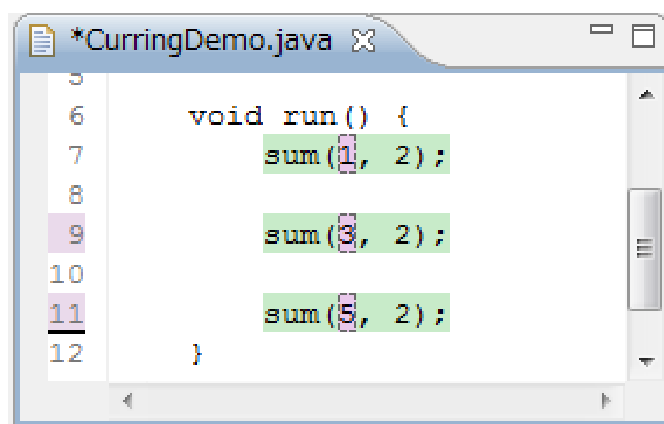

図 5.10: 2 つ目以降のパラメータを固定するカリー化

果が返される。一方、本システムでは、Concerns ビューを活用することによりこれを実現できる。図 5.11 はカリー化関数を定義した Concerns ビューである。ここには、onePlus と twoPlus という 2 つの関数が定義されている。関数 onePlus は引数 x を 1 に固定したもの、関数 twoPlus は引数 x を 2 に固定したものである。このように定義しておくことで、Concerns ビューから関数をコピーし、開発者が使用したい場所にペーストできるため、再利用が簡単である。例えば、関数 onePlus を使いたければ、ここからコピーして使用したい場所にペーストする。その後、パラメータの部分だけその状況にあわせて変更すれば良い。

本システムで実現できるカリー化には欠点もある。図 5.11 のように、カリー化された関数の呼び出しに対して名前付を行うと、sum(1, 2) のような元のコードの把握ができない。そのため、Concerns ビューに表示されているパラメータの情報が、引数 x のものか、引数 y のものか判断できない。これは、本システムで表現したカリー化関数の呼び出しにおいて、固定する方の引数をパラメータとして扱っていないことに起因する。コード上ではパラメータを表していても、本システムがパラメータと認識していないと Concerns ビューには表示されない。この問題を解決するためには、どちらの引数もパラメータとして設定した上で、片方のパラメータは他の領域と同期できるようにする必要がある。本システムで提供するグループ化の機能を用いると、パラメータが他の領域と同期できるが、この方法だと 2 つの引数が両方とも他の領域と同期されてしまう。そのため、片方のパラメータだけを固定することができない。

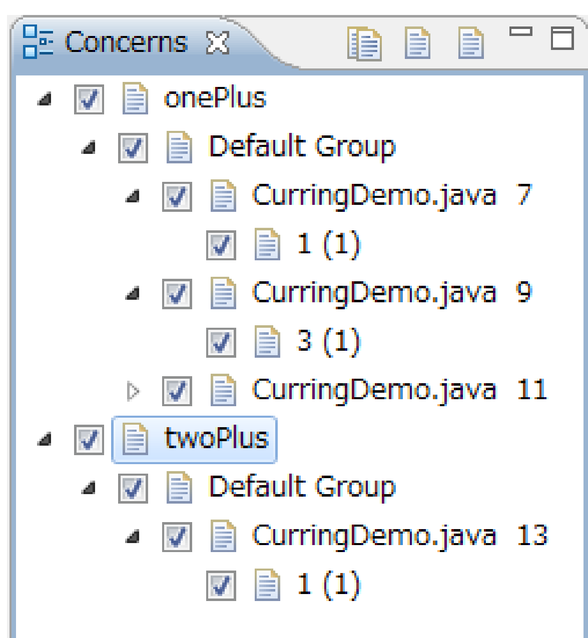

図 5.11: Concerns ビューを活用したカリー化関数の定義

第6章　まとめと今後の課題

6.1　まとめ

本研究では、言語機構を利用したモジュール化の限界について述べ、その解決策として動的なテキストを扱うことができる同期可能なコピー＆ペーストシステムについて提案した。同期可能なコピー＆ペーストシステムでは、コード領域を複製し同期させることによって擬似的なモジュールを表現できる。さらに、同期領域のうち部分的に同期しない指定ができることによるパラメータ化の表現や、Concerns ビューによるアスペクトの表現など、既存の言語機構によるモジュール化の利点を、言語拡張することなく本システムでも生み出せることを示した。本システムでは、直感的な操作方法のみ把握できれば良いので、言語を拡張したときのように難しい文法などを覚える必要がなく、より簡単に扱うことができる。

6.2　今後の課題

本システムでは、コピー＆ペーストによってコードが複製されるため、プログラム全体のコード行数が膨大になってしまうという問題がある。そのため本システム上で、複製されたコード領域を出来る限り省略して表示する機能が必要であると考えられる。このような機能を追加することにより、開発者は統合開発環境上でプログラムの動きを把握しやすくなり、保守が用意になる。例えば、複製されたコードをエディタ上に表示させず、かわりに Concerns ビューで登録したコードに対する名前を表示させるなどの機能が考えられる。

また、第 5 章で紹介した本システムによる言語機構の表現は極一部である。本システムを用いた言語機構の表現について、第 5 章で紹介したもの以外についても比較、検証する必要がある。さらに、本稿では Java 言語を中心に述べたが、本システムはすべてのプログラミング言語に対応できるよう、同期対象であるコードは任意の文字列としている。そのため、プログラムコードだけではなく、Javadoc のようなドキュメントにも適応することができる。このような、本稿で紹介した以外の応用例についても調査をする必要がある。

参考文献

[1] Clement, A., Colyer, A. and Kersten, M.: Aspect-Oriented Programming with AJDT, *Analysis of Aspect-Oriented Software (ECOOP 2003)* (Hannemann, J., Chitchyan, R. and Rashid, A.(eds.)) (2003).

[2] École Polytechnique Fédérale de Lausanne (EPFL): The Scala Programming Language, `http://www.scala-lang.org/`.

[3] Hon, T. and Kiczales, G.: Fluid AOP join point models, *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, New York, NY, USA, ACM, pp. 712–713 (2006).

[4] Hou, D., Jacob, F. and Jablonski, P.: Exploring the design space of proactive tool support for copy-and-paste programming, *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '09, New York, NY, USA, ACM, pp. 188–202 (2009).

[5] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, London, UK, UK, Springer-Verlag, pp. 327–353 (2001).

[6] Miller, R. C. and Myers, B. A.: Interactive Simultaneous Editing of Multiple Text Regions, *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, Berkeley, CA, USA, USENIX Association, pp. 161–174 (2001).

[7] The Eclipse Foundation: AspectJ Development Tools (AJDT), `http://www.eclipse.org/ajdt/`.

[8] The Eclipse Foundation: The AspectJ Project, `http://www.eclipse.org/aspectj/`.

[9] The Eclipse Foundation: Eclipse Java development tools (JDT), `http://eclipse.org/jdt/`.

[10] Toomim, M., Begel, A. and Graham, S.: Managing Duplicated Code with Linked Editing, *Proc. IEEE Symp. Visual Languages: Human Centric Computing*, IEEE Press, pp. 173–180 (2004).