Vol.2010-EMB-18 No.8
2010/8/10

並行処理デザインパターンのアスペクト指向による記述

亀山信吾[†1]　松本倫子[†1]　吉田紀彦[†2]

デザインパターンは，関連するコードが横断的関心事であるため，ソフトウェア開発効率，および保守性，拡張性の向上に限界がある．この解決策として，デザインパターンをアスペクト指向で記述するという研究が行われているが，GoF デザインパターン以外については行われていない．そこで，本研究では，並行処理におけるソフトウェア開発効率，および，保守性，拡張性のさらなる向上を目的として，並行処理デザインパターンのアスペクト指向による記述を行った．また，実験により，記述した並行処理デザインパターンのアスペクトが正しく動作することを確認した．

Aspect-Oriented Implementation of Concurrent Processing Design Patterns

Shingo Kameyama,[†1] Noriko Matsumoto[†1] and Norihiko Yoshida[†2]

A design pattern has some limitations in improving software development efficiency, maintainability, and extensibility because codes of a design pattern are crosscutting concerns. To solve this problem, aspect-oriented implementation of so-called “GoF” design patterns (design patterns to promote component reuse) has been proposed. In this paper, we propose aspect-oriented implementation of design patterns for concurrent processing, and confirm that this implementation works correctly.

†1 埼玉大学大学院理工学研究科
Graduate School of Science and Engineering, Saitama University
†2 埼玉大学情報メディア基盤センター
Information Technology Center, Saitama University

1. はじめに

ソフトウェアを開発していると，設計における同様の問題に直面することがある．この場合，開発熟練者においては，以前のソフトウェア開発における知識，経験により，解決することができるが，開発初心者においては，試行錯誤により解決する必要がある．そのため，ソフトウェア開発効率および品質を低下させてしまう．

そこで，開発熟練者の蓄積された知識，経験を参照することができるようにまとめたものが，デザインパターンである．デザインパターンは，ソフトウェア開発効率を向上させる技術として普及し，以降，多くの研究者により様々なデザインパターンが提案されている．しかし，オブジェクト指向をはじめとする従来技術において，デザインパターンに関連するコードが複数のモジュールに散在しているため，ソフトウェア開発効率および保守性，拡張性の向上に限界がある．

これに対して，デザインパターンをアスペクト指向[1),2)] で記述するという研究が，最も広く普及している GoF デザインパターン[3)] について行われており[4)]，アスペクト指向実行可能 UML による記述の研究事例もある[5)]．アスペクト指向とは，複数のモジュールに散在してしまうコードを単一のモジュールとしてまとめる技術である．これにより，デザインパターンに関連するコードをモジュール化することができるため，デザインパターンの問題点が解決される．しかし，GoF デザインパターン以外のデザインパターンについては，その問題点が依然として残っており，その一つに並行処理デザインパターン[6)] がある．

そこで本研究では，並行処理におけるソフトウェア開発効率，および保守性，拡張性のさらなる向上を目的として，並行処理デザインパターンのアスペクト指向による記述を行う．また，GoF デザインパターンと異なる特性として，多くの並行処理デザインパターンでは，関連する他の並行処理デザインパターンが同時に使用される．そこで，そのようなデザインパターンにおいては，それぞれをアスペクト指向で記述し，それらを調整しながら組み合わせて実装する．この手法は，並行処理デザインパターンに限らず，新たにアスペクト指向で記述するデザインパターンにおいて，関連する他のデザインパターンが使用されている場合，適用することができる．このように，デザインパターンを実装した既存のアスペクトと組み合わせることにより，実装の効率化が期待できる．

以下，2 節では，デザインパターンとその利点，問題点，および本研究で扱う並行処理デザインパターンについて説明する．3 節では，関連技術であるアスペクト指向，およびアスペクト指向言語である AspectJ 言語の機能の一部について説明する．4 節では，関連研究と

© 2010 Information Processing Society of Japan

して GoF デザインパターンのアスペクト指向による記述について説明する．5 節では，本研究の例として Read-Write Lock パターンのアスペクト指向による記述について説明する．6 節では，他の並行処理デザインパターンのアスペクト指向による記述について説明する．7 節では，本研究の手法について考察を行う．最後に，8 節では，本研究のまとめを行う．

2. デザインパターン

デザインパターンとは，ソフトウェア開発における典型的な問題の解決策をパターンとしてカタログ化したものである．GoF（Gang of Four）と呼ばれる，Erich Gamma，Richard Helm，Ralph Johnson，John Vlissides の 4 人が，23 種類の再利用のためのデザインパターン[3)]（以下，GoF デザインパターン）を紹介したことにより，ソフトウェア開発効率を向上させる技術として普及した．以降，多くの研究者により，並行処理，Web アプリケーション，組込みシステムのためのデザインパターンなど，様々なデザインパターンが提案されている．

2.1 利点

デザインパターンは，以下のような利点により，ソフトウェア開発効率を向上させる．

- デザインパターンに関連するコード（以下，パターンコード）をそのまま，あるいは拡張して再利用することにより，プログラミング，テストにおけるコストを削減することができる．
- 開発熟練者の知識，経験が蓄積されているため，開発初心者の手引書として有用である．
- デザインパターン名を述べることにより，他の開発者へ簡潔，正確にソフトウェア設計を伝えることができる．
- プログラミング言語に依存する技法を記述したイディオムと異なり，設計を記述したものであるため，言語，アプリケーションに依存しない高い汎用性を持つ．

2.2 問題点

デザインパターンは，オブジェクト指向，手続き型などの従来技術において，パターンコードが複数のモジュールに散在していることにより，以下のような問題点がある．

- 構造などを理解し，サンプルコードからパターンコードを抽出して，プログラムに適用する必要がある．そのため，ソフトウェア開発効率の向上に限界がある．
- パターンコードを修正する場合，複数のモジュールを修正する必要がある．そのため，修正漏れによる不具合が発生することがあり，保守性，拡張性の向上に限界がある．

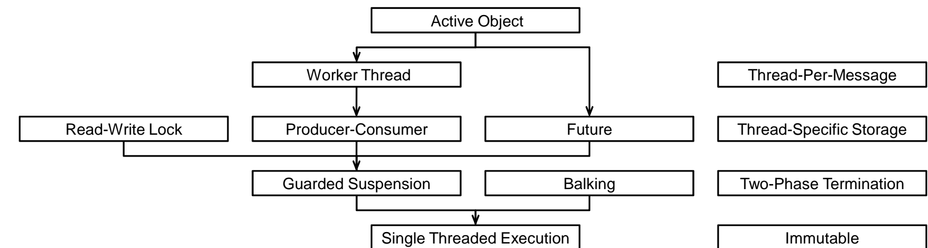

図 1 並行処理デザインパターン
Fig. 1 Concurrent Processing Design Patterns

2.3 並行処理デザインパターン

本節の冒頭で述べたように，GoF デザインパターン以外のデザインパターンとして，並行処理のためのデザインパターン[6)]（以下，並行処理デザインパターン）が提案されている．並行処理デザインパターンは，並行処理における以下のような問題に対して，典型的な解決策を提供するものである．

- 単一のスレッドによる処理にはない，レースコンディション，デッドロックなどの不具合が発生することがある．レースコンディションとは，複数のスレッドが共有資源に対して，排他制御を行わずに読み書きを行うことにより，安全性を失う現象である．デッドロックとは，複数のスレッドが複数のロックを取得し合い，相互にロックが解放されるまで待機することにより，生存性を失う現象である．
- 単一のスレッドによる処理と比べて，多くの資源が必要となることがあり，パフォーマンスに影響する．
- 現象の再現が難しいことがあり，テストが難しい．

また GoF デザインパターンと異なり，多くの並行処理デザインパターンでは，関連する他の並行処理デザインパターンが使用されている．すなわち，相互に包含関係が存在する．以下に並行処理デザインパターンを示す．また，**図 1** にその包含関係を示す．矢印は始点側による終点側の使用を表している．

Single Threaded Execution パターン スレッドセーフでないオブジェクトにおいて，一度に単一のスレッドのみが使用するように排他制御を行うことにより，安全性を守る．スレッドセーフとは，一度に複数のスレッドが使用しても，問題が発生しないことである．

© 2010 Information Processing Society of Japan

Immutable パターン　オブジェクトの状態が変化しないようにして，排他制御を行わないことにより，スループットを向上させる．

Guarded Suspension パターン　事前条件が満たされていない場合，オブジェクトの状態が変化するまで待ち，オブジェクトの状態が変化した場合，それを通知することにより安全性を守る．事前条件とは，目的の処理の前に満たされていなければならない条件である．

Balking パターン　事前条件が満たされていない場合処理を中断することにより，安全性を失わずに，応答性を向上させる．

Producer-Consumer パターン　中継地点を介してオブジェクトの受け渡しを行うことにより，安全性を失わずに，スループットを向上させる．

Read-Write Lock パターン　読み込み処理同士は安全性に影響しないことを利用して，読み書きにおけるロックの取得，解放を分けて行うことにより，安全性を失わずに，スループットを向上させる．

Thread-Per-Message パターン　要求ごとに生成したスレッドに処理を任せることにより，応答性を向上させる．

Worker Thread パターン　要求をオブジェクトとして表し，あらかじめ生成したスレッドに処理を任せることにより，応答性を低下させずに，スループット，キャパシティを向上させる．

Future パターン　Thread-Per-Message パターン，および Worker Thread パターンにおいて，要求の処理の結果を後から取得できるようにすることにより，応答性を向上させる．

Two-Phase Termination パターン　あるスレッドを他のスレッドから終了させる場合，要求を介して終了することにより，安全性を守る．

Thread-Specific Storage パターン　スレッドセーフでないオブジェクトにおいて，スレッド固有の領域を用意することにより，使用側のコードを修正せずに，スレッド固有の処理を行えるようにする．

Active Object パターン　スレッドセーフでないオブジェクトにおいて，単一のスレッドに処理を任せ，その結果を後から取得できるようにすることにより，応答性を低下させずに，一度に複数のスレッドが実行できるようにする．

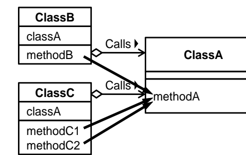

図 2　オブジェクト指向によるモジュール化
Fig. 2　Modularization by Object-Orientation

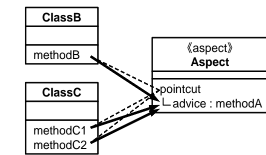

図 3　アスペクト指向によるモジュール化
Fig. 3　Modularization by Aspect-Orientation

3. アスペクト指向

ソフトウェア開発における保守性，拡張性を向上させる技術としてオブジェクト指向がある．オブジェクト指向では，ひとまとまりの処理（関心事）をオブジェクトとしてモジュール化することにより，保守性，拡張性を向上させる．しかし，これには限界があり，関心事のコードが複数のモジュールに散在してしまうことがある．このような関心事を横断的関心事と言い，ロギング，キャッシングなどが例として挙げられる．

アスペクト指向[1),2)] は，このようなオブジェクト指向の限界を補うモジュール化技術である．これにより，横断的関心事を単一のモジュールとしてまとめることができる．

3.1 概　要

オブジェクト指向では，あるモジュール（クラス）のメソッドは，一般に他のモジュールのメソッドから呼び出されて実行される．しかし，呼び出される側のモジュールを追加，除去する場合，呼び出す側のモジュール全てにおいても，関連するフィールド宣言，メソッド呼出しを同様に修正する必要がある．**図 2** にクラス図を示す．メソッド間の矢印は処理の流れを表している．

そこで，アスペクト指向では，呼び出される側のモジュールにおいて，呼び出される位置を指定し，メソッドと関連付ける．これにより，指定した位置に処理が到達した場合，関連付けたメソッドが実行されるように，モジュールが統合される．この位置の指定をポイントカット，関連付けるメソッドをアドバイスと言い，これらが記述されたモジュールをアスペクトと言う．これにより，呼び出される側のモジュールを追加，除去する場合，呼び出す側のモジュールにおいては，関連するフィールド宣言，メソッド呼出しが存在しないため，修正する必要がなくなる．**図 3** にクラス図を示す．本論文では，上段に名前，中段にフィールド，下段にポイントカット，アドバイス，メソッドを記述することにより，アスペクトを表し，点線はポイントカットによる指定を表す．また，次小節で述べる抽象，具象アスペクト

© 2010 Information Processing Society of Japan

に分けて実装していても，一つにまとめて記述している．

3.2 AspectJ 言語

AspectJ 言語とは，オブジェクト指向言語である Java 言語を拡張したアスペクト指向言語である．本研究，および関連研究[4] では，オブジェクト指向による記述に Java 言語，アスペクト指向による記述に AspectJ 言語を使用している．そこで，本小節では，AspectJ 言語の機能の一部について説明する．

ポイントカット　ポイントカットで指定することができる位置は任意ではなく，メソッドの呼出し，実行，フィールドの参照，代入などに限られる．

アドバイス　アドバイスには，ポイントカットで指定した位置の処理に対して，その直前に実行される `before` アドバイス，その直後に実行される `after` アドバイス，それに代わって実行される `around` アドバイスがある．また，`around` アドバイスでは，元の処理を呼び出す `proceed` メソッドを使用することができる．

アスペクト　アスペクトはクラスと同様に，ポイントカット，アドバイスのみではなく，フィールド，メソッドを記述することができ，`abstract`，`extends` 句を使用することにより，抽象，具象アスペクトとして定義することができる．しかし，アスペクトのインスタンスはクラスと異なり，必要に応じて生成され，`new` 演算子を使用して生成することができない．通常は一つのみ生成されるが，`pertarget` 句を使用することにより，ポイントカットで指定した位置の処理に対して，呼び出される側のインスタンスごとに生成されるようになる．また，`privileged` 句を使用することにより，Java 言語のアクセス制御の制約を受けないようにすることができる．他にも，`declare` 句を使用することにより，アスペクトを適用する順番，コンパイルにおけるエラーなどを定義することができる．なお，アスペクトの型の直後に，`+` 演算子を記述することにより，下位アスペクトを含めることができる．

4. 関 連 研 究

前節で述べたアスペクト指向は，横断的関心事をアスペクトとしてモジュール化する技術である．これにより，横断的関心事であるパターンコードを単一のモジュールとしてまとめることができる．そこで，関連研究では，GoF デザインパターンのアスペクト指向による記述が行われている[4),5)]．

本節では，関連研究の例として，GoF デザインパターンの一つである Observer パターンのアスペクト指向による記述[4] を説明する．Observer パターンとは，あるオブジェクトの状態が変化した場合，それに連動して他のオブジェクトのメソッドを実行するというデザインパターンである．

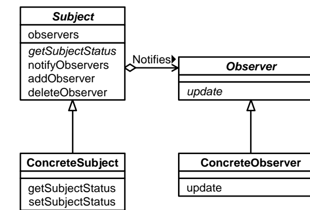

図 4　オブジェクト指向による Observer パターン
Fig. 4　Observer Pattern by Object-Orientation

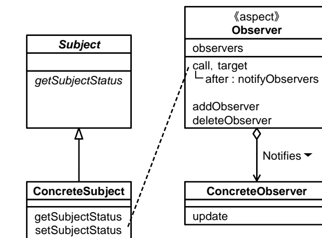

図 5　アスペクト指向による Observer パターン
Fig. 5　Observer Pattern by Aspect-Orientation

4.1 Observer パターンのアスペクト指向による記述

オブジェクト指向では，状態を変化させるメソッドの最後において `notifyObservers` メソッドを呼び出し，その中で連動させるメソッドを呼び出すことにより，Observer パターンを実装する．しかしこの場合，以下のパターンコードが複数のモジュールに散在している．**図 4** にクラス図を示す．

- `notifyObservers` メソッド，およびそのメソッド呼出し．
- `observers` フィールド，およびそのフィールドに対して，追加，除去する `addObserver`，`deleteObserver` メソッド．
- `Observer` クラス．

そこでアスペクト指向では，`notifyObservers` メソッドではなく，`after` アドバイスを実行することにより，Observer パターンを実装する．これにより，パターンコードを単一のモジュールとしてまとめることができる．**図 5** にクラス図を示す．

4.2 利　　点

パターンコードが単一のモジュールとしてまとまっていることにより，2.2 小節で述べた問題点に対応して，以下のような利点がある．

- 構造などを理解していなくても，デザインパターンを適用する位置をポイントカットで指定することにより，プログラムに適用することができる．そのため，ソフトウェア開発効率をさらに向上させることができる．
- パターンコードを修正する場合，複数のモジュールを修正する必要がない．そのため，保守性，拡張性をさらに向上させることができる．

© 2010 Information Processing Society of Japan

Vol.2010-EMB-18 No.8
2010/8/10

また，適用側についてもパターンコードが入り込んでおらず，そのオブジェクト固有の関心事のみで構成されるようになるため，モジュール化が促進されており，再利用性，汎用性を向上させることができる．

5. Read-Write Lock パターンのアスペクト指向による記述

本研究の基本方針として，記述するデザインパターンについて，サンプルコードからパターンコードを抽出し，抽象，具象アスペクトに分けて実装する．抽象アスペクトには，アドバイスなどの適用するプログラムに依存しない情報を記述する．具象アスペクトには，ポイントカットなどの適用するプログラムに依存する情報を記述する．これにより，パターンコードをそのまま再利用する場合，具象アスペクトのみを再実装することにより，プログラムに適用することができる．そのため，アスペクトの再利用性，汎用性を向上させることができる．最後に，デザインパターンを実装した抽象，具象アスペクト（以下，パターンアスペクト）を使用して，動作確認の実験を行う．

本節では本研究の例として，Read-Write Lock パターンのアスペクト指向による記述を説明する．ロックの取得，解放は，典型的な横断的関心事であり，Read-Write Lock パターンでは，読み書きの前後に行われる．アスペクトでは，このようなある処理の前後に他の処理を行うという動作をうまく実装することができる．

5.1 オブジェクト指向による Read-Write Lock パターン

オブジェクト指向では，読み書きの直前において，排他制御下で事前条件のテスト，ロックの取得を行う．そして，読み書きの直後において，排他制御下で，ロックの解放，状態変化の通知を行うことにより，Read-Write Lock パターンを実装する．**図 6** にクラス図を示す．

読み書きにおけるロックの取得，解放は，読み書きを行っているスレッドの数を表すカウンタを増減することで行っており，事前条件のテストにはこれらのカウンタを使用している．この場合，読み込みにおける事前条件は，書き込みを行っているスレッドが存在しないことであり，書き込みにおける事前条件は，読み書きを行っているスレッドが存在しないことである．事前条件が満たされていない場合，状態（カウンタ）が変化するまで待機する．排他制御は，`synchronized` 句を使用し，Java 言語の機構であるインスタンスのロックを取得，解放することで行う．これは Java 言語の規則において，状態変化の待機，通知を行う `wait`，`notifyAll` メソッドを使用するためにも必要である．

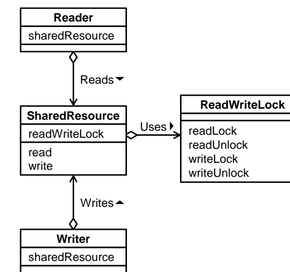

図 6 オブジェクト指向による Read-Write Lock パターン
Fig. 6 Read-Write Lock Pattern by Object-Orientation

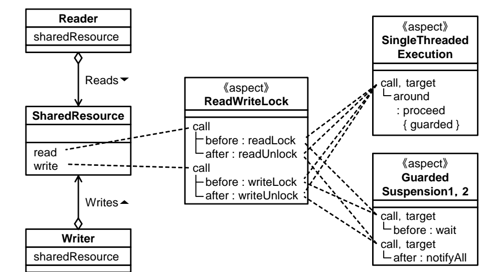

図 7 アスペクト指向による Read-Write Lock パターン
Fig. 7 Read-Write Lock Pattern by Aspect-Orientation

5.2 アスペクト指向による Read-Write Lock パターン

Read-Write Lock パターンでは，事前条件のテスト，状態変化の通知に Guarded Suspension パターンが使用されている．さらに Guarded Suspension パターンでは，排他制御に Single Threaded Execution パターンが使用されている．以上の包含関係は，図 1 にも示されている．

そこでアスペクト指向では，Single Threaded Execution および Guarded Suspension パターンアスペクトと組み合わせて Read-Write Lock パターンを実装する．本研究の追加方針として，組み合わせるパターンアスペクトは，それ自身のデザインパターンとしても使用することができるように実装する．**図 7** に Read-Write Lock パターンのクラス図を示す．また，**図 8**，**図 9** に Single Threaded Execution，および Guarded Suspension パターンのクラス図を示す．

Read-Write Lock パターンアスペクトにおいて，ロックの取得，解放を行い，Single Threaded Execution および Guarded Suspension パターンアスペクトにおいて，排他制御および事前条件のテスト，状態変化の通知を行っている．この場合，Single Threaded Execution パターンアスペクトは，Guarded Suspension パターンアスペクトの前に適用される必要がある．これは，事前条件のテストと目的の処理，オブジェクトの状態変化とその通知が行われる間，オブジェクトの状態が変化しないことを保証するため，および前小節で述べたように `wait`，`notifyAll` メソッドを使用するためである．そこで，`declare` 句を使用することにより，適用される順番を定義している．定義する必要があるのは，具象アスペ

© 2010 Information Processing Society of Japan

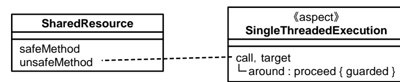

図 8 アスペクト指向による
Single Threaded Execution パターン
Fig. 8 Single Threaded Execution Pattern
by Aspect-Orientation

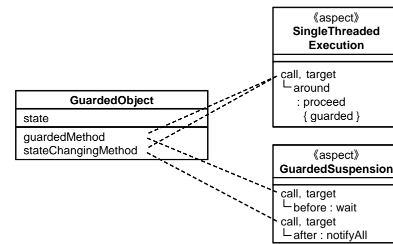

図 9 アスペクト指向による
Guarded Suspension パターン
Fig. 9 Guarded Suspension Pattern
by Aspect-Orientation

クトが適用される順番である．しかし，具象アスペクトにおける記述を削減するため，抽象アスペクトで定義して具象アスペクトの型を記述すると，具象アスペクトの型を追加，除去，変更する場合，抽象アスペクトを修正する必要があり，拡張性，汎用性を低下させてしまう．そのため，抽象アスペクトの型の直後に，+ 演算子を記述して具象アスペクトを含めることにより，適用される順番を定義している．

またオブジェクト指向では，`ReadWriteLock` クラスが，`SharedResource` インスタンス固有の情報（ロック，カウンタ）を保持している．そのため，`SharedResource` インスタンスごとに，`ReadWriteLock` クラスのインスタンスを生成している．しかし，アスペクト指向では，`ReadWriteLock` アスペクトが，`SharedResource` インスタンス固有の情報を保持しており，アスペクトのインスタンスは，3.2 小節で述べたように，通常は一つしか生成されない．そこで，`ReadWriteLock` アスペクトにおいて，`pertarget` 句を使用することにより，`SharedResource` インスタンスごとに，`ReadWriteLock` アスペクトのインスタンスを生成している．

なお，事前条件のテストを抽象メソッドとすることにより，読み書きにおける事前条件を `GuardedSuspension1`，2 アスペクトという 2 つの具象アスペクトに分けて実装している．そして，これらの具象アスペクトでは，`ReadWriteLock` アスペクトの `private` フィールド（カウンタ）の参照，代入を行うため，`privileged` 句を使用している．

5.3 実　　験

実装した Read-Write Lock パターンアスペクトを使用して，一度に複数のスレッドが読み書きを行うという動作確認の実験を行った．その結果，オブジェクト指向による Read-Write Lock パターンの動作と同様であることを確認した．これにより，実装した Read-Write Lock パターンアスペクトが正しく動作することを確認した．

6. 他の並行処理デザインパターンのアスペクト指向による記述

他の並行処理デザインパターンについても，アスペクト指向による記述を行った．また，5.2 小節と同様に，記述するデザインパターンで他の並行処理デザインパターンが使用されている場合，それらのパターンアスペクトと組み合わせて実装した．実験についても，5.3 小節と同様に，実装したそれぞれのパターンアスペクトが正しく動作することを確認した．

ただし，Thread-Specific Storage パターンについては，アスペクト指向でうまく記述することができなかった．これは，Thread-Specific Storage パターンが，一つのオブジェクトを，API が同一であるオブジェクトおよび処理が同一であるオブジェクトに分けて再実装するというデザインパターンであり，アスペクトの機能ではそのような大規模な修正が難しいためである．

Immutable パターン　Immutable パターンは，ある処理を行うことで実装するのではないため，アドバイスではうまく実装することができない．そこで `declare` 句を使用して，コンパイルにおけるエラーを定義し，デザインパターンを適用するオブジェクトのフィールドの代入が，そのオブジェクトのフィールド宣言，コンストラクタ以外で行われていないか検査することにより，Immutable パターンを実装している．

Balking パターン　アドバイスにおいて `return` 文を使用した場合，ポイントカットで指定した位置における処理ではなく，アドバイスが中断されるため，アスペクトでは他の処理を中断するという動作をうまく実装することができない．そこで，目的の処理に代わって `around` アドバイスを実行し，事前条件が満たされている場合は，`proceed` メソッドを使用して目的の処理を行う．満たされていない場合は，`return` 文を使用して `around` アドバイスを中断することにより，Balking パターンを実装している．

Producer-Consumer パターン　ポイントカットには，アドバイスの実行を指定するものもあるが，アドバイスには名前がないためうまく区別することができない．そこで，アドバイスにおいてオブジェクトの受け渡しを行うのではなく，その役割を果たすメソッドを呼び出す．そして，Guarded Suspension パターンアスペクトではメソッドの名前で区別して，事前条件のテスト，状態変化の通知を行っている．

Thread-Per-Message パターン　Thread-Per-Message パターンを適用しておらず，単純にメソッドを呼び出しているプログラムに後から適用する場合，オブジェクト指向で

ⓒ 2010 Information Processing Society of Japan

は，スレッドを生成して，要求の処理を任せるように修正する，あるいはその役割を果たすオブジェクトを用意して，使用側のフィールド宣言，メソッド呼出しを修正する必要がある．これに対して，アスペクト指向では，要求の処理に代わって `around` アドバイスを実行し，スレッドが `proceed` メソッドを使用して要求の処理を行うことにより，プログラムを修正する必要がない．

Worker Thread パターン　抽象アスペクトで要求の処理を行うメソッドを呼び出すようにすると，メソッドの名前を変更する場合，抽象アスペクトを修正する必要があり，拡張性，汎用性を低下させてしまう．そこで，抽象アスペクトにおいて，本体が空であるメソッドを持つ内部クラスを定義して，そのメソッドで要求の処理を行うようにオーバーライドした下位クラスを，要求を表すオブジェクトとして実装している．

Future パターン　実装したパターンアスペクトでは，適用側のプログラムを修正しないために，併用する Thread-Per-Message パターンアスペクト，および Worker Thread パターンアスペクトにおいて，要求の処理の戻り値の型と同一であるインスタンスを返すように拡張している．そして，マップを使用することにより，そのインスタンスと要求の処理の結果を関連付けている．

Two-Phase Termination パターン　Balking パターンの項で述べたように，アスペクトでは，他の処理を中断するという動作をうまく実装することができないため，`run` メソッドを中断してスレッドを終了させることができない．そこで，`stop` メソッドを使用することにより，スレッドを終了させている．`stop` メソッドは，他のスレッドから使用される場合，安全性が失われることがあるため，推奨されていない．しかし，実装したパターンアスペクトでは，終了するスレッド自身が安全な位置で使用するため，安全性が守られる．

Active Object パターン　スレッドセーフでないオブジェクトにメソッドを追加する場合，オブジェクト指向では，他のオブジェクトにもメソッドを追加して，そのメソッドに対応した要求を表すオブジェクトを追加する必要がある．これに対して，アスペクト指向では，要求の処理に代わって `around` アドバイスを実行し，スレッドが `proceed` メソッドを使用して要求の処理を行うことにより，組み合わせる Worker Thread パターンアスペクトの具象アスペクトを追加するのみでよい．

いずれも，オブジェクトの数，型の定義は抽象メソッドとしている．また，アドバイスの引数，戻り値の型として `Object` 型を指定しているため，任意のインスタンス，戻り値に対応できる．これらにより，パターンアスペクトの再利用性，汎用性を向上させている．

7. 考　　察

本研究では，記述するデザインパターンで他の並行処理デザインパターンが使用されている場合，それらのパターンアスペクトと組み合わせて実装した．これにより，図 7，図 9 にも示されているように，図 1 に示した並行処理デザインパターンの包含関係を，コードとして表すことができた．これは，パターンコードをアスペクトとしてまとめたことにより，デザインパターンをモジュール単位で扱うことができるようになったためである．

また，図 8，図 9 に示したように，組み合わせるパターンアスペクトは，それ自身のデザインパターンとしても使用することができるように実装した．それ自身のデザインパターンとして使用することはできても，他のパターンアスペクトとうまく組み合わせられない場合，相互のパターンアスペクトにおいて，動作を変更せずに，組み合わせることができるように調整（チューニング）する必要がある．例えば，最初の実装では，Single Threaded Execution パターンアスペクトは，自身のアスペクトインスタンスのロックを取得，解放することで排他制御を行っていた．この場合，Single Threaded Execution パターンとして使用することはできる．しかし，Guarded Suspension パターンアスペクトは，自身のアスペクトインスタンスの `wait`，`notifyAll` メソッドを使用していたため，そのアスペクトインスタンスのロックを取得，解放する必要があった．そこで，Single Threaded Execution パターンアスペクトを，適用側のインスタンスのロックを取得，解放することで排他制御を行うように調整した．これにより，Guarded Suspension パターンとしても，Single Threaded Execution パターンとしても使用することができるようになった．しかし，複数の事前条件がある Read-Write Lock パターンアスペクトなどと組み合わせる場合，Guarded Suspension パターンアスペクトを，1 つの具象アスペクトとして実装すると，複数の事前条件をうまく記述できず，2 つの具象アスペクトに分けて実装すると，他方のアスペクトインスタンスに状態変化の通知を行うことができないため，うまく組み合わせることができなかった．そこで，Guarded Suspension パターンアスペクトを，適用側のインスタンスの `wait`，`notifyAll` メソッドを使用するように調整した．そして，Single Threaded Execution，Guarded Suspension パターンアスペクトを同一のインスタンスに適用するように変更した．これにより，それぞれのデザインパターンとして使用することができるようになった．また，Single Threaded Execution，Guarded Suspension パターンアスペクトにおいて，適用側のインスタンスごとに，アスペクトのインスタンスを生成する必要がなくなった．

以上が本研究における新規性であるが，この手法は，並行処理デザインパターンに限ら

ⓒ 2010 Information Processing Society of Japan

ず，包含関係が存在するデザインパターンのアスペクト指向による記述においても，適用することができる．これにより，記述するデザインパターンで使用されている他のデザインパターンに対して，既にパターンアスペクトが存在する場合，それらと組み合わせて実装し，プログラミング，テストにおけるコストを削減することができる．そのため，並行処理を基礎とする，2 節の冒頭で述べた Web アプリケーション，組込みシステムのためのデザインパターンへの，本研究で実装したパターンアスペクトの応用が期待できる．

8. おわりに

本研究では，並行処理におけるソフトウェア開発効率，および保守性，拡張性のさらなる向上を目的として，並行処理デザインパターンのアスペクト指向による記述を行った．また，記述するデザインパターンで他の並行処理デザインパターンが使用されている場合，それらのパターンアスペクトと組み合わせて実装した．

最後に，今後の課題を以下に示す．

- アスペクト指向による記述，チューニングの方法論
- 並行処理デザインパターンアスペクトの分類
- デザインパターンの実装の洗練，検討
- アスペクト指向実行可能 UML による記述

参考文献

1) 長瀬嘉秀，天野まさひろ，鷲崎弘宣，立堀道昭：AspectJ によるアスペクト指向プログラミング入門，ソフトバンク パブリッシング (2004).
2) 千葉　滋：アスペクト指向入門，技術評論社 (2005).
3) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns*, Addison-Wesley (1994).
4) Hannemann, J. and Kiczales, G.: Design Pattern Implementation in Java and AspectJ, *Proceedings of 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.161–173 (2002).
5) Kosuge, S., Teruya, A., Iwata, E., Sugai, M., Matsumoto, N. and Yoshida, N.: Design Pattern Specifications in Aspect-Oriented Executable UML, *Proceedings of International Conference on Applied Computing 2009*, Vol.2, pp.139–144 (2009).
6) 結城　浩：増補改訂版 Java 言語で学ぶデザインパターン入門マルチスレッド編，ソフトバンク クリエイティブ (2006).

© 2010 Information Processing Society of Japan