Vol.2011-SE-171 No.13
2011/3/14

# Javaプログラムの XML 表現を用いた Concurrency Design Pattern の検出

グエン ディン バー [†1]　大 森 隆 行[†1]　丸 山 勝 久[†1]

既存のプログラム内部に存在するデザインパターンを識別できれば，そのプログラムに対する理解をより深めることができる．このため，従来からパターンを自動検出する手法やツールがいくつも提案されている．しかしながら，それらのほとんどが GoF デザインパターンを検出対象としており，並行処理に関するデザインパターンの自動検出はいまだ残されたままである．本論文では，Java 言語で記述されたマルチスレッドプログラムから，Guarded Suspension，Balking，Future パターンを自動検出するアルゴリズムを提案する．検出アルゴリズムは，これらのパターンに含まれる構造的特徴を容易に検査可能とするために，Java プログラムの XML 表現を用いる．本論文では，さらに提案手法を実装したツールと簡単な適用実験の結果を示す．

# Detecting Concurrency Design Patterns by using XML Representations of Java Programs

Nguyen Dinh Ba ,[†1] Takayuki Omori [†1]
and Katsuhisa Maruyama[†1]

Recovering design patterns facilitates understanding of programs including them. Although many approaches and tools have been proposed, almost all of them focused on detecting GoF design patterns. Therefore, the detection of concurrency design patterns is still a remaining issue. This paper presents fully automated algorithms for detecting Guarded Suspension, Balking, and Future patterns in Java multi-threaded programs. These algorithms use XML representations of the programs to easily check their structural characteristics that are available for the pattern detection. The paper also shows a tool implementing these algorithms and brief experimental results with this tool.

†1 立命館大学情報理工学部
Department of Computer Science, Ritsumeikan University

## 1. はじめに

デザインパターンとは，ソフトウェア開発において頻繁に発生する問題，問題に対する解法，解法を適用した際の結果をまとめたものである[1),2)]．これは，ソースコードの再利用だけでなく，ソフトウェア設計における考え方，概念，指針，定石のような知見や経験を再利用することを目指している．このため，それぞれのデザインパターンの背景や考え方を習得することで，ソフトウェア設計における熟練開発者の経験や知識を再利用できるという観点での効果は大きい．また，デザインパターン習得した技術者同士であれば，その名前だけを使うことで意思の疎通が容易になる．このような背景により，近年のオブジェクト指向設計において，さまざまなデザインパターンが活用されている．

通常，デザインパターンは，解決したい問題や解決によって引き起こされる結果（利点や欠点）を十分に考慮した上で用いられる．すなわち，それぞれのデザインパターンは，それが適用される特定の設計文脈や設計者の意図を反映している．よって，プログラム内部に存在するデザインパターンを開発者や保守者が識別できれば，そのプログラムに対する理解をより深めることができる．プログラム理解が進むことで，プログラムの修正や改変が容易になり，ソフトウェア保守における作業の軽減が期待できる．

ここで，既存のプログラムに付属するドキュメントに，そのプログラムに適用されているデザインパターンが記述されている場合，それらの識別は容易である．しかしながら，ドキュメントには，適用されているデザインパターンが必ず記述されているとも，その記述が必ず適切であるとも限らない．このような状況において，大規模なプログラムにおけるデザインパターンの適用を，漏れなくかつ誤りなく人手で検出することは困難である．

このような理由から，既存のプログラムからパターンを自動的に検出する手法やツールがいくつも提案されている[3)–11)]．特に，文献 12) では，さまざまな手法やツールの分類が述べられている．従来の手法は，その検出精度や検出効率の観点より，プログラムの構造などに基づく静的な特徴を利用する手法，実際にプログラムを実行した際のメソッドの呼び出し関係など動的な特徴を利用する手法，あるいは，それら両方の特徴を利用する手法に大きく分けられる．また，プログラムの構文的な情報だけでなく，意味的な情報（たとえば，クラスの命名規則）を利用する手法も提案されている．これらの手法が，どれもプログラム理解を支援していることは明確であるが，残念ながら，それらのほとんどが Gamma らの GoF パターン[1)] を検出対象としている．

本論文では，Java 言語で記述されたマルチスレッドプログラムを対象とし，そのプログ



© 2011 Information Processing Society of Japan

ラムから Cuncurrency Design Pattern（並行デザインパターン）[13)–15)] を自動検出する手法を提案する．検出対象のデザインパターンは，Guarded Suspension，Balking，Future である．提案手法では，Java プログラムの XML 表現として，Jxplatform[16)] により生成される XSDML(eXtensible Software Document MarkupLanguage)[17)] を採用する．XSDML 文書は，もとのプログラムの構文要素だけでなく，そのプログラムに対する構文解析と意味解析により取得した情報を内包する．このため，Java プログラムにおける構造的特徴を検査するモジュールを，XML 処理器を用いて容易に作成可能である．さらに，本論文では，XML 表現を用いた検出アルゴリズムと，これらのアルゴリズムを Eclipse のプラグインとして実装したツールについて述べ，簡単な適用実験の結果を示す．

Java では言語仕様によりマルチスレッド機構が提供されており，比較的簡潔にマルチスレッドプログラムを記述することができる．さらに，システムの応答性の向上（利用者から見た応答時間の短縮）を目的として，複数のスレッドによる並行処理を実現した Java プログラムはますます増えている．このような状況において，既存のマルチスレッドプログラムから，Concurrency Design Pattern を自動検出することは，開発者や保守者のプログラム理解を促進するという点で重要である．特に，従来のデザインパターン検出手法では扱っていなかった Concurrency Design Pattern を検出対象とすることの意義は大きい．

## 2. Concurrency Design Pattern

ここでは，検出対象である 3 種類の Cuncurrency Design Pattern をそれぞれ説明する．

**(1) Guarded Suspension パターン**

このパターンは，インスタンスの状態が同期化メソッド（`synchronized` メソッド）の実行を許可するまで，同期化メソッドの実行を一時停止させておきたい場合に適用する．同期化メソッドとは，Java のモニタにより，複数のスレッドから同時に実行されることが禁止されているメソッドを指す．このパターンを用いることで，マルチスレッドプログラムにおいて，インスタンスが保持するデータの整合性（インスタンスの安全性）が保たれることを指す．Guarded Suspension パターンのクラス図と Java での実装コードの例を**図 1** に示す．

図 1 において，`GuardedObject` は同期化メソッドを持つクラスである．`execute()` は並行実行において同期化されたメソッドである．また，`state` は同期メソッドの実行を許可するかどうかを判定するフィールド変数であり，`GuardedObject` のインスタンスの状態に応じて変化する．同期メソッドの実行が許可されるときガード条件（`state` の値）は真となり，不許可のときガード条件は偽となる．あるスレッドが `execute()` を実行するとき，ガード

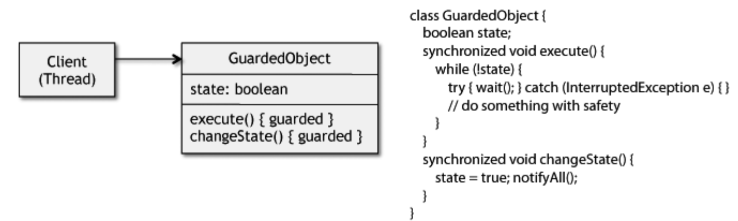


**図 1** Guarded Suspension パターンのクラス図とそのコード
Fig. 1 Class diagram for Guarded Suspension pattern and its code.

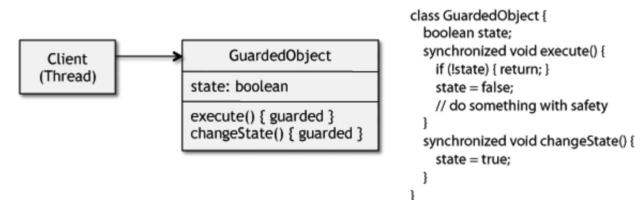


**図 2** Balking パターンのクラス図とそのコード
Fig. 2 Class diagram for Balking pattern and its code.

条件が真であれば，そのメソッドの実行は継続される．しかし，ガード条件が偽の場合，メソッドの実行はガード条件が真になるまで待たされる．`changeState()` は，インスタンスの状態を変化させるメソッドである．

**(2) Balking パターン**

このパターンは，同期化メソッドが呼び出されたにもかかわらず，インスタンスの状態がそのメソッドの実行を許可しない状況において，何もせずに実行を中断させたい場合に適用する．Guarded Suspension パターンがメソッドの実行を待たせることで，インスタンスの安全性を保証するのに対して，Balking パターンはメソッドの実行を中断することでインスタンスの安全性を保証する．Balking パターンのクラス図と Java での実装コードの例を**図 2** に示す．

図 2 を見るとわかるように，Balking パターンの構成要素は Guarded Suspension パターンと同じである．ただし，`execute()` メソッドの振る舞いが異なる．あるスレッドが `execute()` を実行するとき，ガード条件（`state` の値）が真であれば，そのメソッドの実行は継続される．ガード条件が偽の場合，メソッドの実行は中断され，何もせずに呼び出し元に戻る．`changeState()` は，インスタンスの状態を変化させるメソッドである．


© 2011 Information Processing Society of Japan

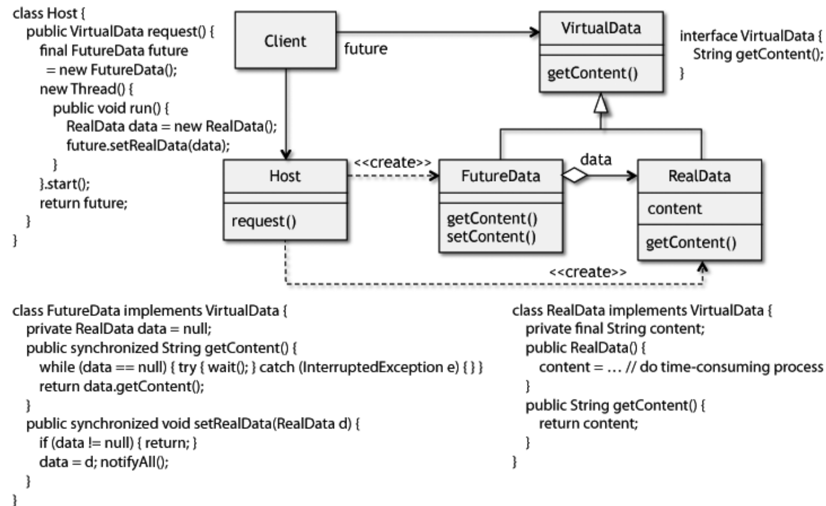


**図 3** Future パターンのクラス図とそのコード
Fig. 3 Class diagram for Future pattern and its code.

**(3) Future パターン**

このパターンは，実行結果を得るまでに時間がかかるメソッドを呼び出した際に，その処理の終了を待たずに呼び出し元に戻り，未来に実行結果を取得したい場合に利用する．つまり，Thread-Per-Message パターン[13] に，戻り値の取得を付与したパターンと捉えることができる．Thread-Per-Message パターンでは，メッセージ (メソッド呼び出し) ごとに新たなスレッドを生成し，生成したスレッドに処理を依頼する．これにより，非同期的なメソッド呼び出しを擬似的に実現でき，応答性が向上する．Future パターンのクラス図と Java による実装コードの例を**図 3** に示す．

図 3 において，`Client` クラスは `Host` クラスの `request()` メソッドを呼び出す (作業を依頼する)．作業の実行結果 (戻り値) として，`Client` は `FutureData` クラスのインスタンスを受け取る．`Host` は `request()` が呼ばれるごとに新しいスレッドを生成し，そのスレッドに `RealData` クラスのインスタンスの生成を依頼する．また，`Client` に `FutureData` のインスタンスを返す．`VirtualData` は，`FutureData` のインスタンスと `RealData` のインスタンスを同一視させるためのインタフェースであり，`Client` が受け取るこれらのインスタンスを区別することを不要にする．`RealData` は実際のデータを表す．通常，このクラスのインスタンスを生成するには長い時間がかかる．このため，`FutureData` のインスタンスを `RealData` のインスタンスの代わりに，引換券として `Client` に渡す．

実際に実行結果を取得するために `Client` から `FutureData` の `getContent()` メソッドが呼び出された場合，まず `RealData` のインスタンスが生成できているかどうかを調べる．すでに `RealData` のインスタンスが存在する場合には，`FutureData` の `getContent()` への呼び出しを `RealData` の `getContent()` へ転送する．`RealData` のインスタンスの生成が完了していない場合は，スレッドはインスタンスの生成を待つことになる．

このような観点から，Future パターンにおける `VirtualData`，`FutureData`，`RealData` の関係は，Proxy パターン[1),2)] と捉えることができる．`RealData` のインスタンスの生成には時間がかかるため，とりあえずその代理 (virtual proxy) として `FutureData` のインスタンスを生成する．実際に `RealData` のインスタンスが必要になった場合 (`getContent()` が呼び出された場合) に，`RealData` のインスタンスの生成が完了するのを待つ．

## 3. デザインパターンの検出手法

本手法では，デザインパターンに含まれる構造的特徴を検査することで，Guarded Supension パターンと Balking パターンを検出する．具体的には，同期メソッドとガード条件を，プログラム構造の特徴として検査する．さらに，検出したパターン間の関係を検査することで，Future パターンを検出する．

### 3.1 ソースコードの XML 表現

本手法では，ソースコード解析ツール Jxplatform[16)] を使って，ソースコードを XML 表現 (XSDML[17)] 文書) に変換する．デザインパターンの検出は，実際のソースコードの代わりに，これらの XML 表現に対して行われる．ここで，Jxplatform とは，Java ソースコードに対して構文解析と意味解析を適用することにより取得した情報を保存できるツールプラットフォームである．XSDML は，20 個の非終端要素と 7 個の終端要素でソースコードを修飾する．非終端要素は構文情報であり，子ノードとしてテキストノードを持たない．終端要素は字句情報であり，子ノードにソースコード断片を格納したテキストノードのみを持つ．

### 3.2 検出アルゴリズム

ここでは，Java ソースコードの XML 表現である XSDML 文書を用いて，2 章で紹介したデザインパターンを検出するアルゴリズムを示す．

**(1) Guarded Suspension パターンの検出アルゴリズム**

図 1 に示すように，Guarded Supension パターンでは，`Object` クラスの `wait()` メソッ



© 2011 Information Processing Society of Japan

ドの呼び出しにより実行中のスレッドが待ち状態に移行する（その後，`Object` の `notify()` あるいは `notifyAll()` メソッドの呼び出しにより，ガード条件が再度テストされ，待ち状態から復帰する）．以上より，Guarded Suspension パターンの検出において，注目するプログラム構造の特徴は，`wait()` の呼び出しと，それを内包する `while` 文との構文上の関係となる．

[Guarded Supension パターン検出アルゴリズム]

入力: プロジェクト内部の Java ソースファイル群

出力: Guarded Supension パターンの Guarded メソッド

**Step 1.** 各 Java ソースファイルを XSDML 文書に変換する．

**Step 2.** `wait()` メソッドの呼び出しを含むソースファイルに対応する XSDML 文書を見つける．`wait()` メソッドの呼び出しを XSDML で表現すると，次のようになる．

```
<Expr bind="wait()" fqn="void" id="j1931137034"
  ref="java.lang.Object" sort="MethodCall"> ... </Expr>
```

属性 `sort="MethodCall"` を持つ`<Expr>`要素は，メソッド呼び出し式を指す．`<Expr>`要素の属性 `bind` は呼び出しメソッドのシグニチャ，属性 `ref` は呼び出し先メソッドの属するクラスを指す．よって，`wait()` メソッドの呼び出しを見つけるためには，属性 `sort="MethodCall"`，`bind="wait(*)"`，`ref="java.lang.Object"` を持つ`<Expr>`要素を探せばよい．`wait(*)` は，任意の数（実際には，0 ～ 2 個）の引数を指す．ここでは，`wait()` の呼び出し式に対応する`<Expr>`要素を Expr-wait と呼ぶことにする．

**Step 3.** Step 2 で特定した `wait()` メソッドの呼び出し式が，`while` 文の内部に存在するかどうかを検査する．`while` 文の XSDML 表現は，次のようになる．

```
<Stmt id="j1931137027" sort="WHILE"> ... </Stmt>
```

`while` 文は，属性 `sort="WHILE"` を持つ`<Stmt>`要素で表現される．よって，`while` 文が `wait()` メソッドの呼び出し式を内包するどうかを検査するには，XSDML 文書において，Step 2 で特定した Expr-wait が `while` 文に対応する`<Stmt>`要素の子孫であることを確認すればよい．ここでは，Expr-wait にから DOM 木における親要素を上方に辿ることで，Expr-wait を内包する最内（もっとも内側）の `while` 文に対応する`<Stmt>`要素を特定する．この要素を Stmt-while と呼ぶ．

**Step 4.** Step 3 で特定した `while` 文を持つメソッドが，同期化メソッドかどうかを検査する．同期化メソッドの場合，このメソッドを Guarded Suspension パターンの Guarded メソッドと判断し，検出結果として出力する．同期メソッド宣言部の XSDML 表現は，次のようになる．

```
<Method fqn="void" id="j1931137025" sig="execute()"
  synchro="yes" typefirst="j1615917060"> ... </Method>
```

Step 2 と同様に，もとのプログラムの XSDML 文書において，DOM 木における親要素を上方に辿ることで，Stmt-while を含む`<Method>`要素を特定することができる．この要素の属性 `synchro="yes"` の場合，これは同期メソッドである．

**(2) Balking パターンの検出アルゴリズム**

図 2 に示すように，Balking パターンでは，ガード条件の検査に `if` 文を使う．処理を中止するには，`return` 文で呼び出し先メソッドから戻るか，`throw` 文で例外を投げる，ガード条件の検査は同期メソッドの内部で行う．以上より，Balking パターンの検出において，注目するプログラム構造の特徴は，`return` 文あるいは `throw` 文と，それを内包する `if` 文との構文上の関係となる．

[Balking パターン検出アルゴリズム]

入力: プロジェクト内部の Java ソースファイル群

出力: Balking パターンの Guarded メソッド

**Step 1.** 各 Java ソースファイルを XSDML 文書に変換する．

**Step 2.** `return` 文と `throw` 文を含むソースファイルに対応する XSDML 文書を見つける．`return` 文を XSDML で表現すると，次のようになる．

```
<Stmt id="j1981591559" sort="RETURN"> ... </Stmt>
```

`return` 文は，属性 `sort="RETURN"` を持つ`<Stmt>`要素で表現される．よって，XSDML 文書において，この条件を満たす`<Stmt>`要素を特定すればよい．この要素を Stmt-return とおく．`return` 文でなく `throw` 文を特定したい場合は，属性 `sort="THROW"` を持つ`<Stmt>`要素を探せばよい．

**Step 3.** Step 2 で特定した `return` 文や `throw` 文が，`if` 文の内部に存在するかどうかを検査する．`if` 文の XSDML 表現は，次のようになる．

```
<Stmt id="j1981591555" sort="IFELSE"> ... </Stmt>
```

`if` 文は，属性 `sort="IFELSE"` を持つ`<Stmt>`要素で表現されるので，XSDML 文書



© 2011 Information Processing Society of Japan

において，Step 1 で特定した Stmt-return から DOM 木における親要素を上方に辿ることで，Stmt-return を内包する最内（もっとも内側）の if 文に対応する<Stmt>要素を特定する．この要素を Stmt-if と呼ぶ．

**Step 4.** Step 3 で特定した if 文（Stmt-if に対応）を持つメソッドが，同期メソッドかどうかを検査する．同期メソッドの場合，このメソッドを Balking パターンの Guarded メソッドと判断し，検出結果として出力する．Guarded メソッドの特定方法は，Guarded Suspension パターンの検出アルゴリズムにおける Step 4 と同じである．

**(3) Future パターンの検出アルゴリズム**

Future パターンは別のパターンを内包するため，このパターンを検出する際には，事前に検出したパターン間の関係を検査する．図 3 を用いて，Future パターンに内包されるデザインパターンを説明する．

- Proxy パターン: VirtualData クラスと，その子孫クラスである FutureData クラスと RealData クラスで構成されている．
- Guarded Suspension パターン: FutureData を介した RealData へのアクセスは，RealData のインスタンスの存在（FutureData が RealData のインスタンスへの参照を保持しているかどうか）によりガードされる．
- Balking パターン: FutureData に対して，RealData のインスタンスが，複数回設定されることを防ぐ．

以上より，Future パターンを検出するために，Proxy パターン，Guarded Suspension パターン，Balking パターンの存在を検査する．同時に，Future パターンでは，時間のかかる処理のために新規にスレッドを生成し，そのスレッドに処理を委譲する．そこで，スレッドの生成に関する特徴も検査する．

[Future パターン検出アルゴリズム]

入力: プロジェクト内部の Java ソースファイル群

出力: Future パターンにおいて依頼を受け取るクラスとメソッド

**Step 1.** Proxy パターンを検出する．検出した Proxy パターンにおいて，代理の役割を持つクラスを FutureData，実際のデータを保持する役割を持つクラスを RealData とおく．また，処理の委譲メソッドを getContent とおく．

**Step 2.** Step 1 で検出した Proxy パターンに関連するソースコードに対して，Guarded Suspension パターンを検出する．このパターンが検出されたとき，その Guarded メソッドが，Proxy パターンの委譲メソッド getContent() と一致するどうかを検査する．

**Step 3.** Step 1 で検出した Proxy パターンに関連するソースコードに対して，Balking パターンを検出する．このパターンが検出されたとき，その Guarded メソッドが，Proxy パターンの RealData のインスタンスへの参照を設定しているかどうかを検査する．ここでは，このメソッドを setRealData() とおく．

**Step 4.** Step 1 で検出した Proxy パターンに処理を依頼するクラスを特定する．ここでは，このクラスを Host とおく．Host においてスレッドを生成しているメソッドを，Future パターンにおいて要求を受け取るメソッドと判断し，検出したクラスとメソッドを出力する．Step 4 の詳細は後で述べる．

Step 4 における Host は，以下の 3 つの処理を実行する．

(1) FutureData のインスタンス (引換券) と RealData のインスタンスを生成する．同時に，FutureData のインスタンスに RealData のインスタンスへの参照を設定する．
(2) 新規にスレッドを生成して，そのスレッドを起動する．
(3) FutureData のインスタンスへの参照を返す．

Java で新規にスレッドを生成する際には，(1) Thread クラス（あるいはその子孫クラス）のインスタンスを直接生成する方法と，(2) Runnable インタフェースの実装クラスのインスタンスを生成し，それを Thread クラスのコンストラクタに渡す方法がある．図 3 の例では，Host の request() メソッドにおいて，Thread の子クラス（匿名クラス）のインスタンスを直接生成することで新規にスレッドを生成し，直後にそのスレッドを起動している．

以上をふまえて，Future パターンの検出アルゴリズムの Step 4 において，Host を特定するアルゴリズムの詳細を示す．

[Future パターン検出における Host の特定アルゴリズム（**Step 4** の詳細）]

入力: プロジェクト内部の Java ソースファイル群
　Proxy パターンと Balking パターンの検出結果

出力: Future パターンにおいて依頼を投げるクラス Host

**Step 4a.** 各 Java ソースファイルを XSDML 文書に変換する．

**Step 4b.** FutureData のインスタンスを生成する式を特定する．インスタンスを生成する式の XML 表現は，次のようになる．

```
<Expr bind="FutureData()" id="j0192765957" sort="InstanceCreation" ... > ...
  <Expr bind="FutureData()" fqn="future.FutureData" id="j0192765958"
    ref="future.FutureData" sort="CtorCall"> ... </Expr></Expr>
```

インスタンスの生成を行う式は，属性 sort="InstanceCreation"を持つ<Expr>要


© 2011 Information Processing Society of Japan

素で表現される．この要素は，子要素にコンストラクタ呼び出しを伴う．よって，XSDML 文書において，属性 sort="CtorCall"を持つ<Expr>要素を探し，生成するクラスの名前（完全限定名）を取得すればよい．ここでは，入力となる FutureData の名前（future.FutureData）を用いて，属性 ref="future.FutureData"を持つ<Expr>要素を特定する．この要素を Expr-new とおく．

**Step 4c.** Step 4b で特定した Expt-new を含むメソッドの戻り値を検査する．return 文を検出する方法は，Balking パターンの検出アルゴリズムにおける Step 2 と同じである．ただし，Future パターンでは，FutureData のインスタンスを戻り値として返すため，戻り値の型を検査する必要がある．return 文の戻り値に対応する式の XSDML 表現は，次のようになる．

```
<Stmt id="j0192765967" sort="RETURN"> ...
  <Expr fqn="future.FutureData" id="j0192765968" read="yes" sort="VarRef"> ...
  </Expr> ... </Stmt>
```

この XSDML 表現を見ると，戻り値に対応する<Expr>要素の fqn 属性の値を検査すればよいことがわかる．

以上より，まず XSDML 文書において，DOM 木における親要素を上方に辿ることで，Expr-new を含むメソッドに対応する<Method>要素を探す．次に，この<Method>要素の子孫として存在する，return 文を表す<Stmt>要素を探し，その戻り値に対応する<Expr>要素を特定する．最後に，その<Expr>要素の fqn 属性の値が，Step 1 で特定した Proxy パターンの FutureData の名前と一致するかどうかを検査する．

**Step 4d.** Guarded Supension パターン検出アルゴリズムの Step 2 と同様に，start() メソッドの呼び出しを探し，その存在を検査する．start() の呼び出し式の XSDML 表現を次に示す．

```
<Expr bind="start()" fqn="void" id="j0192765966"
  ref="java.lang.Thread" sort="MethodCall"> ... </Expr>
```

属性 sort="MethodCall"と bind="start()"を持つ Expr 要素を探せばよい．属性 ref に関しては，Thread となるとは限らないため，ここでは検査しない．

**Step 4e.** Step 4d において特定した start() メソッドの呼び出しから，start() メソッドが属するインスタンス（メソッド呼び出し obj.start() のインスタンス obj に相当）に対応する<Expr>要素を特定する．これは，start の呼び出し式に対応する<Expr>要素の接頭子 (prefix) として，DOM 木上で検索することが可能である．ここでは，検索により見つかった<Expr>要素を Expr-obj とおく．

Exp-obj 要素がインスタンス変数への参照あるいはメソッド呼び出しの場合，その<Expr>要素の属性 fqn の値を見ることで，インスタンス obj の型が分かる．Exp-obj 要素がローカル変数への参照の場合，参照に対応する<Expr>要素の defid からその変数の宣言に対応する<Local>要素を探す．その後，その要素の型を表す<Type>要素を探し出す．<Type>要素の属性 fqn の値が obj の型となる．このようにして，start の属するインスタンスの型を取得する．ここでは，これを Type-start とおく．

start の呼び出しに明示的なインスタンスが指定されていない場合は，匿名の内部クラスを利用している可能性がある．この場合の Expr-obj の XSDML 表現を次に示す．

```
<Expr id="j0192765980" sort="InstanceCreation" typefirst="j0192765982"> ...
  <Class anonymous="yes" fqn="Host$1" id="j0192765981">
    <Type fqn="java.lang.Thread" id="j0192765982" sort="Object">... </Type>
  ... </Class> ... </Expr>
```

<Expr>要素の属性 sort="InstanceCreation"のとき，その要素の子要素の<Class>要素を見つける．<Class>要素の属性 anonymous="yes"は，そのクラスが匿名クラスであることを指している．この場合，<Class>要素の子要素の<Type>要素の属性 fqn を見ることで，start() の属するインスタンスの型を知ることができる．

最終的に，start の属するインスタンスの型 Type-start が Thread あるいはその子孫クラスと一致する場合，start() の呼び出しを含むメソッドが，Future パターンにおける要求を受け取るメソッド（図 3 の request()）の候補となる．さらに，このメソッドを含むクラスが，出力である Host の候補となる．

**Step 4f.** Step 4e で特定したインスタンスの型 Type-start に対応するクラスで，スレッドでの実行処理を記述した run() メソッドを探す．その後，run() の内部で RealData のインスタンスを生成しているかどうかを検査する．まず，XSDML 文書において，Type-start に対応する<Class>要素を探し，その子孫において run() に対応する<Method>要素を探す．さらに，Step 4b と同様に，この<Method¿要素の子孫に RealData のインスタンスの生成を行う式が存在するどうかを検査する．

次に，run() の内部で，FutureData のインスタンスに RealData のインスタンスへの参照を設定しているかどうかを検査する．参照を設定するメソッドは，Step 3 で特定した Balking パターンの setRealData() である．よって，Guarded Supension パターン検出アルゴリズムの Step 2 と同様に setRealData() への呼び出しを探し，その存在を検査する．このメソッド呼び出し式が存在する場合，Step 4e で特定したクラスを


© 2011 Information Processing Society of Japan

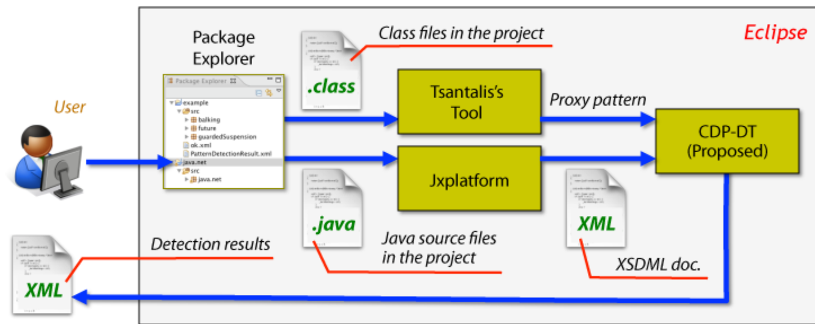


図 4　検出システムの全体構成
Fig. 4　Overall architecture of the proposed system.

Future パターンの Host と判断する．

## 4. 検出システム

3 章で述べた検出手法を実装した Eclipse のプラグインを開発した．ここでは，このプラグインを含む検出システムの全体構成と，このシステムを用いて行った簡単な適用実験の結果を示す．

### 4.1 システム構成

検出システムの全体構成を**図 4** に示す．図中の CDT-DT が実装した検出ツール (Concurrency Design Pattern Detection Tool) である．このツールのプログラムは，コメントと空行を除いて 580 行である．

利用者（開発者や保守者）は Eclipse のパッケージエクスプローラにおいて，プロジェクトを選択する．Tsantalis のツール[8] は，選択されたプロジェクト内部に存在する Java ソースコードのクラスファイル（.class ファイル）を読み込み，GoF パターンの検出結果（XML ファイル）を出力する．このツールでは，プログラムをグラフで表現し，グラフの類似性を判定するアルゴリズムによりパターンを検出する．本システムでは，Proxy パターンの検出結果のみを用い，それが CDP-DT に渡される．同時に，プロジェクト内部に存在する Java のソースファイルは，Jxplatform[16] により XSDML 文書（XML 表現）に変換され，CDP-DT に渡される．パターンの検出結果は，利用者が指定した XML 文書に書き込まれる．

図 1，図 2，図 3 に示す Java のソースコードに対してパターンの検出を適用した結果を**図 5** に示す．検出された 3 種類のデザインパターンは`<Pattern>`要素において列挙される．`name` 属性がパターンの種類を表している．それぞれのパターンのインスタンスは，`<Instance>`要素で表現される．`<Role>`要素は，各パターンにおける役割の名称 (`name`)，それを実現しているクラス (`class`) とメソッド (`method`) を属性に持つ．

```
- <System>
  - <Pattern name="GuardedSuspension">
    - <Instance>
        <Role class="guardedSuspension.GuardedObject" method="execute()" name="GuardedObject"/>
      </Instance>
    - <Instance>
        <Role class="future.FutureData" method="getContent()" name="GuardedObject"/>
      </Instance>
    </Pattern>
  - <Pattern name="Balking">
    - <Instance>
        <Role class="balking.GuardedObject" method="execute()" name="GuardedObject"/>
      </Instance>
    - <Instance>
        <Role class="future.FutureData" method="setRealData(future.RealData)" name="GuardedObject"/>
      </Instance>
    </Pattern>
  - <Pattern name="Future">
    - <Instance>
        <Role class="future.FutureData" method="getContent()" name="FutureData"/>
        <Role class="future.RealData" method="getContent()" name="RealData"/>
        <Role class="future.Host" method="request()" name="Host"/>
      </Instance>
    </Pattern>
  </System>
```

図 5　検出結果の例
Fig. 5　Example of Detection results.

表 1　適用実験の結果
Table 1　Experimental results.

| Project | #Files | #G.Suspension | #Balking | #Future |
|---|---|---|---|---|
| Sample | 6 | 2 (2) | 2 (2) | 1 (1) |
| java.net | 54 | 1 (1) | 99 (84) | 0 (0) |

### 4.2 適用実験

提案した検出手法の実現可能性を示すため，さらに有効性に関する課題を洗い出すために，実装ツール CDP-DT を用いて，既存の Java プログラムに対して適用実験を行った 検出対象プログラムは，図 1，図 2，図 3 に示すサンプルの Java コードと JDK1.4 の java.net パッケージである．**表 1** に実験結果を示す．

表 1 において，#Files は対象プログラムに含まれる Java ソースファイルの数，



© 2011 Information Processing Society of Japan

#G.Suspension，#Balking，#Future はそれぞれ検出したパターンの数である．括弧内の数は，それぞれのパターンに対して人手で検出した結果である．検出時間は，Sample の場合に 8 秒程度，java.net の場合に 1 分程度であった．

検出した個々のデザインパターンに対して，人手で検出した結果と比較したところ，GuardedSuspension パターンと Future パターンに関しては，正確に検出できていることが確認できた．ただし，これらのデザインパターンは，実験対象プログラムに少数しか存在していなかった（java.net パッケージには Future パターンは存在しなかった）ため，ツールの検出結果が妥当であるとは現時点で断定できない．

Balking パターンに関しては，Sample において正しく検出できていることが確認できた．java.net において検出された 99 個の内訳は，return 文で実行を中断する場合が 34 個，throw 文で実行を中断する場合が 65 個であった．人手により検出した 84 個のデザインパターンとシステムにより検出された 99 個のデザインパターンと比較したところ，人手で検出した 84 個はすべてシステムにより検出されていることが確認できた．つまり，検出漏れはなかった．残りの 15 個（15 = 99 - 84）は検出間違い（余分な検出）であった．

検出の精度に関してはさらなる改良が必要ではあるが，検出システムにより短時間に自動的にデザインパターンが検出できることの意義は大きいといえる．

## 5. おわりに

本論文では，Java ソースコードの XML 表現を用いて，並行プログラムのデザインパターンを自動的に検出する手法とそのシステムを提案した．システムの利用者（開発者や保守者）は，デザインパターンを検出したい Java プロジェクトを指定するだけで，既存のマルチスレッドプログラムに存在する 3 種類の ConcurrencyDesign Patterns (Guarded Suspension, Balking, Future) を知ることができる．これにより，そのプログラムを理解する手間が大幅に削減されることが期待できる．

4.2 に示したように，現在の検出アルゴリズムでは検出誤りが発生する．今後の課題として，検出漏れを増やさずに誤りを減らすようにアルゴリズムを改良することがあげられる．また，現在のシステムでは，検出可能なデザインパターンは 3 種類しかない．文献 13)–15) で紹介されている他のデザインパターンに関しても構造的特徴を抽出することで，検出可能なデザインパターンの種類を増やすことを考えている．さらに，静的解析だけでなく，動的解析の利用も検討している．

## 参考文献

1) E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley (1995)
2) F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, “Pattern-Oriented Software Architecture, Volume 1: A System of Patterns”, Wiley (1996)
3) J. Seemann, and J.W. Gudenberg, “Pattern-Based Design Recovery of Java Software”, FSE ’98, pp.10–16 (1998)
4) J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, “Towards Pattern-Based Design Recovery”, Proc. ICSE ’02, pp338–348 (2002)
5) D. Heuzeroth, T. Holl, G. Högström, and W. Löwe, “Automatic Design Pattern Detection”, Proc. IWPC ’03, pp.94–103 (2003)
6) J.M. Smith and D. Stotts, “SPQR: Flexible Automated Design Pattern Extraction From Source Code” ASE ’03, pp.215–224 (2003)
7) 堅田淳也, 小林隆志, 佐伯元司, “静的解析と動的解析を用いたデザインパターン検出手法”, 信学技報 SS 105(24), pp.19–24 (2005)
8) N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T. Halkidis, “Design Pattern Detection Using Similarity Scoring”, IEEE TSE, Vol.32, No.11, pp.896–909 (2006)
9) N. Shi and R. Olsson, “Reverse Engineering of Design Patterns from Java Source Code”, ASE ’06, pp.123–134 (2006)
10) J. Dong, D.S. Lad, and Y. Zhao, “DP-Miner: Design Pattern Discovery Using Matrix”, ECBS ’07, pp.371-380 (2007)
11) 鷲崎弘宜, 深谷和宏, 久保淳人, 深澤良彰, “パターン適用前のソースコードを用いたデザインパターン検出”, コンピュータソフトウェア, Vol.27, No.2, pp.136–141 (2009)
12) J. Dong, Y. Zhao, and T. Peng, “A Review of Design Pattern Mining Techniques”, IJSEKE, Vol.19, No.6, pp.823–855 (2009)
13) D. Lea, “Concurrent Programming in Java: Design Principles and Pattern”, 2nd Ed, Prentice Hall (1999)
14) M. Grand, “Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML”, Wiley (1998)
15) Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects”, Wiley (2000)
16) Jxplatform, http://www.jtool.org
17) K. Maruyama and S. Yamamoto, “A CASE Tool Platform Using an XML Representation of Java Source Code”, SCAM ’04, pp.158-167 (2004)


© 2011 Information Processing Society of Japan