分散ソフトウェア用のアスペクト指向テスティング・フレームワーク

西澤無我
東京工業大学大学院 情報理工学研究科

1. 背景

今日、J2EE アプリケーションに代表される分散ソフトウェアの開発効率の向上は、ソフトウェア業界の重要な課題の 1 つである。このため、分散ソフトウェアの開発プロセスにテスティングを組み込む必要性が高まっている。テストを開発プロセスに組み込むことで、テスト対象ソフトウェアのソースコードは明瞭になり、リファクタリングが促進され、ソフトウェアの開発効率が向上する。特に最近では、XP(Extreme Programming)で単体テストを提唱しているため、多くの分散ソフトウェア開発者が単体テストをソフトウェアの開発プロセスに組み込むようになった。そして、JUnit を代表とする様々なテスティング・フレームワークやツールが開発されてきた。

しかし､これらのテスティング･フレームワークを利用しても分散ソフトウェア開発者が、white-box testing のためのプログラムを記述するのは手間のかかる作業である。その理由の 1 つは、テストをするかしないかによって、テスト対象ソフトウェア（プログラム）を修正・変更しなくてはならないからである。White-box test とは、プログラムの内部構造や振る舞いをチェックするためのテストであり、プログラム内の変数（メソッド呼出の引数やフィールド）の値をテストプログラム内でチェックする。この white-box test を実現するために、テスト作成者はテスト対象プログラムの変数の値をテストプログラムに通知するためのコードを、テスト対象プログラム内に挿入しなければならない。また、テストの終了時には、この通知するためのコードは不要となるため、テスト対照プログラムから削除しなくてはならない。テストをするかしないかによって、テスト対象プログラムを編集することは、分散ソフトウェアの開発効率を著しく低下させてしまう。
また、分散ソフトウェアのテスティングが困難な理由の 1 つとして、テスト対象ソフトウェアが複数のホスト上で動作するプログラムから構成されるケースが多いためである。このようなソフトウェアをテストする場合、テストプログラムもそれぞれのホスト上で動作するサブプログラム群として開発する必要が出てくる。1 つのテストプログラムが複数のサブプログラムから構成されていると、そのサブプログラム間でネットワーク・コミュニケーションが必要になり、テストのアルゴリズムの記述、理解が困難になる。

2．目的

本プロジェクトの目的は、テスト作成者が分散ソフトウェアのより簡潔な white-box test プログラムを効率良く記述・開発できるようにすることである。簡潔なテストプログラムを実現するためには、上述した問題を解決する必要がある。簡潔なテストプログラム

を実現するに

- テスト対象プログラムに手を加えることなく、テスト対象プログラムの変数の値を取得できること
- 複数ホスト上で動作するプログラムから構成される分散ソフトウェアのテストであっても、テストプログラムは分散を意識せず、テストのアルゴリズムにのみ注目して記述できること

が必要である。

3. 提案

上述した問題を解決するため、AOP 技術を利用した、分散ソフトウェア用のテスティング・フレームワークを提案する。本フレームワークの利用者は、テストプログラムをアスペクトとして記述することで、テスト対象プログラム内の変数を簡潔な記述で取得し、テストプログラム内で利用することができる。ここで使用する AOP 技術は、分散ソフトウェア用の AOP 技術である remote pointcut 機構である。これを使用することで、利用者は異なるホスト上で動作しているプログラム中の変数の値を、面倒なネットワーク処理を記述せずに、容易に取得することができる。これらの AOP 技術により、テスト作成者は分散環境を意識せずに white-box test プログラムを、単一のホスト上で動作するプログラムとして記述することができる。

4. 開発内容

本プロジェクトで開発するソフトウェアは、Java 用のアスペクトコンパイラとテストの実行時ライブラリの 2 つである。

本フレームワークでは、利用者にテストプログラムをアスペクトで記述させる。アスペクトはオブジェクト指向技術のクラスモジュールを拡張したモジュールであるため、独自の言語コンストラクトを使用して定義される場合がある。本フレームワークの利用者は、アスペクトを Java 用の汎用的な AOP 言語として知られる AspectJ を拡張した文法で記述する。そのため、利用者が記述したアスペクトを解釈するコンパイラが必要になる。本コンパイラは、利用者が記述したアスペクトを読み込み、Java の通常のクラスファイルを生成する。

本フレームワークのテストの実行時ライブラリは、テストの実行時に、JUnit の実行時ライブラリと同様、複数のテストプログラムを順次実行するためのメカニズムを提供する。また、この実行時ライブラリは、分散ソフトウェア内の変数値を自動でテストプログラム(アスペクト)に通知する仕組みも提供する。本プロジェクトでは、これらの機能を提供するテストの実行時ライブラリを JBoss web application server のサービスの 1 つとして開発する。以下では、これら 2 つのソフトウェアについて詳しく説明する。

4. 1. Java 用のアスペクトコンパイラ (Djcutter compiler)

本アスペクトコンパイラは、フルスクラッチから開発された。AspectJ 言語が提供するコンパイラを拡張する方法も考えられたが、テストの実行時ライブラリとの相性を考慮に入れてすべて手動で構築した。以下では、本コンパイラの front-end と back-end とが行う仕事について詳述する。

まず、本コンパイラの front-end の役割は、利用者が書いたアスペクト記述を読み込み、字句解析、構文解析をおこない、アスペクトを表現した抽象構文木を出力することである。字句解析器は、Java 言語の字句解析器が提供する機能と同じ機能を実装した。また、アスペクトは、フィールド、メソッドの他にポイントカット、アドバイスと呼ばれるアスペクトメンバを定義することができる。これらの抽象構文木を作成するために、LL(k) を利用した構文解析器を開発した。

次に、本コンパイラの back-end の役割は、front-end が出力したアスペクトの構文木を読み込み、アスペクトを表現する Java の通常のクラスファイルを生成することである。本コンパイラの back-end は、クラスファイルを bytecode 変換器を利用して直接作成する。例えば、back-end は、アスペクトを表現する抽象構文木を受け取ると、Java のクラスを表現するクラスファイルを出力する。あるいは、アスペクトのメソッドやフィールドを表現する構文木を受け取ると、back-end はクラスのメソッドやフィールドを表現する bytecode を出力する。アドバイスを表現する構文木を受け取ると、本構文解析器はメソッド名の無いメソッドを表現する bytecode を生成する。ただ、ポイントカットに関しては、既存の Java のクラスメンバにマッピングすることができないので、ポイントカットの構文木を受け取ると、本構文解析器はその構文木の内容をクラスファイルの属性に格納する。クラスファイルの属性には様々な役割をもつものが存在するが、開発者が自由に属性を定義することが可能である。これらの back-end の処理は visitor method により実装される。Bytecode 変換器には、Javassist を利用した。

4. 2. JBoss 用のテストの実行時ライブラリ (DjcManager)

テストの実行時 (分散ソフトウェアの実行時) までに、本実行時ライブラリは様々な bytecode 変換を行う。例えば、テスト対象プログラムの中の変数値をテストプログラムに通知するためには、テスト対象プログラムに手を加えなければならない。本実行時ライブラリは、テスト対象プログラムが JBoss にデプロイされた際に、テスト対象プログラムの bytecode に手を加えている。こうすることで、テスト作成者には、テスト対象プログラム内の変数値が自動的にテストプログラムへ通知されているように見えるのである。以下では、本実行時ライブラリが行う bytecode 変換について詳述していく。また、今回利用した JBoss のバージョンは 4.0.0RC1 である。

[ロード時 (デプロイ時) の bytecode 変換を実現]

クラスのロード時 (デプロイ時) に bytecode 変換を行うためには、JBoss のクラスローダを拡張する必要がある。JBoss クラスローダ (以下 UCL と略す) は、あるクラスをロードする場合、そのクラスのクラスファイルのバイト列をローカルディスク上から検索する。そして、そのバイト列が見つかった場合には、そのバイト列を読み込み、そのクラスを表現する Class クラスのオブジェクトを生成する。そのため、ロード時の bytecode 変換を簡単に実現するには、ロードしようとしているクラスのクラスファイルのバイト列がローカルディスク上から検索されたときに、そのバイト列を編集し、編集後のバイト列を利用して Class クラスのオブジェクトを生成すれば良いのである。

[ロード時にテスト対象プログラムの変数値を取得 (Load-time Weaving)]

DjcManager はロード時に、テストプログラムが必要とする変数値を取得できるよう、テスト対象プログラムを bytecode 変換、編集する。DjcManager は取得する変数値の詳細をテストプログラム (アスペクト) のポイントカット宣言からチェックする。もしロードしているクラスファイル内に取得するべき変数が見つかれば、DjcManager は、その値をテストプログラムに通知するコードをクラスファイルに挿入する。もしクラスファイル内に変数が見つからなければ、クラスファイルを編集せずに、UCL にバイト配列を渡す。

[ネットワーク越しからテストプログラムのクラスファイルを取得]

DjcManager がテストプログラムに変数値を通知するコードを挿入するためには、テストプログラムのクラスファイルを取得している必要がある。分散ソフトウェアのテストの場合、テストプログラムとテスト対象プログラムとは異なるホストに配置されている可能性が高い。そのため、DjcManager にはテストプログラムのクラスファイルをネットワーク越しから取ってくるメカニズムが実装されている。具体的には、ロードするクラスのクラスファイルの検索をネットワーク越しのディスクからも行えるようにすればよい。このネットワーク通信には、Stateful SessionBean を利用する。

[テスト対象プログラムの Un-Load]

テスト終了時には、編集されたテスト対象プログラムを修正しなくてはならない。本実行時ライブラリ、DjcManager は、利用者からは見えないところでテスト対象プログラムを編集していたので、編集されたテスト対象プログラムの修正も利用者には気づかれずに行うべきである。このメカニズムを実現するために、JBoss の hot deploy 機能を利用する。Hot deploy とは、一度ロードされたクラスを、JBoss を再起動せずに再ロードするメカニズムである。DjcManager は、テストが終了時に、この hot deploy 機能を強制的に呼出し、編集されたテスト対象プログラムを再ロードする。その再ロードの際には、DjcManager がテスト対象プログラムを変換することはない。

5. 既存の技術や製品との比較

本テスティング・フレームワークの特徴は、分散ソフトウェア中の変数の値を、分散ソフトウェアに手を加えずに取得し、テストプログラム内で利用できる点である。これにより、既存のテスティング・フレームワークを利用した white-box test プログラムよりも、簡潔な記述でテストプログラムを実装することができる。