Implementation Patterns レジュメ (P26-P35)

2008/05/17 9:00-10:00

担当:齋藤 超

P26 抽象クラス

- Java において、抽象的なインターフェースと具象的な実装を区別する手段のひとつ。(もう 1 つはインターフェース)
- スーパークラスは、あらゆるサブクラスにランタイムで置換可能であるという意味で abstract である。⇔Java の abstract キーワードが付いているかということではない。
- インターフェースと抽象クラスとを比較して、それぞれいつ使うか？
  - インターフェースを使う場合
    - 単一のクラスが複数のインターフェースを同時にサポートする必要があるならインターフェース。
    - ただし、あらゆるインターフェースの変更によって、全ての実装クラスの変更が必要になる。
  - 抽象クラスを使う場合
    - 抽象クラスはデフォルトの実装を持てるから、上記のような苦労はない。追加メソッドが具象的であるなら(abstract でないなら?)、既存の実装を破壊することもない。
    - ただし、1 つのスーパークラスに対して忠実に定義されなければならない。同じクラスで複数の視点が必要ならば Java インターフェースを使うべき。
- abstract キーワードは、もしそのクラスを使いたいなら、サブクラスの実装をしなければならないことを読み手に伝える。
- もしクラス階層のルートを便利にインスタンス化可能にできる機会があるならそうすべき。
- 一度抽象化の道を選んだら、簡単に度を過ぎやすいし、まったく見返りのない抽象性を作ってしまう。ルートクラスをインスタンス化可能にしようと努力することは、役目を果たしそうにもない抽象性を抹消するのを後押ししてくれる。
- インターフェースとクラス階層は相互排他ではない。
  - インターフェース・・・機能へのアクセス手段の表現に使う。
  - スーパークラス・・・機能の実装方法のひとつを表現に使う。
- 変数(variable-引数?)はインターフェースと一緒に定義されなければならない。その場合、将来のメンテナが必要に応じて実装の取替えを自由に出来るように変数の型を定義してやらなければならない。

P27 バージョン付けされたインターフェース

- インターフェースを変更したいが、それができない場合。
  - 新しいオペレーションをインターフェースに追加したい場合
  - オペレーションを追加した新インターフェースを使うユーザと、それに気づかずに古いインターフェースを利用するユーザが同時にいる場合

  →ダウンキャストを使って新旧のインターフェースを切り替える。
- instanceofを使ってダウンキャストすることについて
  - 一般的には、特定のクラスを使うことをコードに記述してしまうので順応性が下がる。複数のインターフェース選択を作り始めたら、デザインを考え直すサインである。
- ただしこれはスマートな解決方法ではない。

- 「versioned interface」は「バージョン付けされたインターフェース」よりも「拡張バージョンインターフェース」という意味合いでは？
- マルチインターフェースでも対応できるのでは？
  - 例で言えばCommandインターフェースとは別にReversibleインターフェースを作る。

P28 バリューオブジェクト

- 2つの型のスタイル
  - Javaのプリミティブ型的。状態は不変。~~関数的スタイル。~~関数型言語的スタイル。
  - state。状態を持つ。~~手続き的スタイル。~~手続き型言語的スタイル。
- 図5.1の例・・・手続き的なインターフェースでは、~~暗黙のうちに意図しない変化を及ぼしやすい~~。オブジェクトの状態が手続きの順序に暗黙のうちに依存する。⇔関数型言語的スタイルならば、処理の呼び出し順序によって状態が変ることがない。

  →可能なら状態を持たない作りにしたほうがよい。
- 会計システムの例インスタンス生成時にのみ値を設定する。インスタンス生成後のステートの変化を気にしなくてよい。
- バリュースタイルのオブジェクトの操作は新しいオブジェクトを返す。オブジェクト自身の状態は変らない。
- システムにおいて状態を持つオブジェクトと不変なオブジェクトの境界を見極める必要がある。
- バリューオブジェクトが使われない理由
  - パフォーマンスの問題←しかしボトルネックとなるのは一部分である
  - このスタイルが馴染みがなく、状態を持つオブジェクトと不変オブジェクトの線引きが困難な場合

P31 特化

- 類似点と相違点の相互作用を伝えることで、コードを読みやすく変更しやすいものにする。
- プログラミングの根底にある巨大で普遍的な問題
  - 同一ロジック・異なるデータ→シンプル
  - 異なるロジック・同一データ→複雑
- ロジックとデータの線引きは実ははっきりしない。
- ロジックの分岐に使われるフラグデータ
- 計算に作用するヘルパーオブジェクトフィールド
- パターンはロジックの類似点と相違点を伝えるテクニックの集まりである。⇔データの類似点と相違点は、ロジックに比べれば複雑ではない。

P32 サブクラス

- 処理を共有するためにサブクラス化する場合の制限
  1. 一度だけ使える切り札である。機能していないサブクラス階層を再構築しようとした場合、まずクラスの混乱もつれを解きほぐさなければならない
  2. サブクラスの理解の前にスーパークラスの理解をしなければならない。スーパークラスが複雑であればあるほど制限は強くなる。
  3. スーパークラスの変更が危険である。その変更によってサブクラスにどんな影響があるかわからない。
  4. ~~これらの問題全てが、深い継承階層が原因で形成される。~~深い継承階層において上記 1～3 の問題が合成されて出現する。
- 平行階層特に致命的な継承の 1 つ。片方のクラス継承を変更したら、重複するもう片方の平行階層も変更しなければならない。
- 上記の注意点を念頭に置けば、サブクラス化はテーマとバリエーションでの計算を表現するためのパワフルなツールとなりうる。
- ロジックのファクタリングが便利なサブクラスへの到達のカギ。

- 図 5.3 と 5.4 に誤り。Product クラスのサブクラス名が間違っている。
  誤：InsuranceContract、PensionContract
  正：InsuranceProduct、PensionProduct

P34 実装役

- 多態的なメッセージは、オブジェクトから構築されたプログラムの選択を表現する基本的な方法。
- 処理のある部分から見た時、コードの意図と一致したことが起こる限りは、詳細は重要ではない。→「契約による設計」における契約を守る限りは、処理の詳細は重要ではない。
- 多態的なメッセージの美しさは、システムにバリエーションを持たせられることである。
- 意図の表現と実装を分離する。
- 将来新たなプログラマによって、考えもしなかったようなバリエーションが生まれうる。
- Java では手続き的な書き方をしてしまうとこの最高の資源(多態性)を簡単に失ってしまう。

P34 インナークラス

- 新規にファイルを作ってクラスを定義したくない場合。
- 多大なコストを払うことなくクラス作成の恩恵を受けられる。
- スーパークラスとなるのは、Object か、局所的にしか関心を持たない他のクラスの拡張を表現するのに便利なクラス。
- エンクロージングインスタンスのデータへアクセスできる。
- 実際には引数なしコンストラクタを持たない。リフレクションで引数なしコンストラクタを呼び出してインスタンスを作成しようとした場合に問題になる。
  →エンクロージングインスタンスと完全に切り離したいなら、static なインナークラスとして定義する。