Implementation Patterns レジュメ (P36-P42)

2008/06/21 9:00-10:00

担当:齋藤 超

P36 インスタンス特化の振る舞い(Instance-Specific Behavior)

- インスタンスごとに異なる振る舞いを実行する。
- 通常よりもコストがかかる。
  - クラスの読み手は、そのクラスが何をするかを理解しにくくなる。
  - 処理の途中でロジック(を持つオブジェクト)が変ると、コードが理解しにくくなる。
    →オブジェクト生成時点でのみインスタンス特化の振る舞いを設定することで、コードを理解しやすくさせる。
- 異なる振る舞いを持つオブジェクトを持たせるなら、個々のオブジェクトがどのように振る舞うかを理解するために、動くサンプルの観察やデータフローの分析をしなければならない。

P36 条件分岐(Conditional)

- if/then、switch ステートメントは、インスタンス特化の振る舞いを表現するためのもっともシンプルな方法。
- データに従ってインスタンスごとに実行するロジックを切り替える。
- 条件分岐のメリットは、全てのロジックが 1 クラスに記述されているため、コードの読み手が処理のパスを見つけるのに複数のクラスを参照しなくて良いこと。(後述の委譲パターンとの対比)
- デメリットは、コードを変更する時に遠回りが必要なこと。
  - 条件分岐の増加はコードの信頼性を低下させる。
  - ~~1クラスに全てのロジックが記述されているため、ある箇所を修正した場合、依存関係のないパスのロジックであっても正当である見込みがなくなってしまう。~~
  - プログラム上の実行パスは、ロジックが正しいことの確実性をそれぞれ持っている。パス同士で確実性が互いに独立していると仮定するなら、プログラムはパスが増えるにつれて正しさを失う。実際は確実性は完全に独立しているわけではないが、多数のパスを持つプログラムは少数のパスしか持たないものよりも欠陥がありそうだと言えるだろう。(この段落では「プログラム中のある実行パスのロジックが正しいからと言って、他の実行パスのロジックも正しいとはかぎらない」と言いたい？)
- デメリットの GUI プログラムでの例
  1. editor#display()メソッド内で、複数の図形オブジェクトごとにケース分けする記述がある。
  2. editor#contains()メソッドが必要になった場合、display()と同様のケース分けを記述することになり、重複が発生する。
  3. さらにここに新しい図形オブジェクトの種類を追加しようとした場合、display()とcontains()とでケース分けの修正が重複する。
- 上記の問題はサブクラス化するか委譲を使い、条件分岐をサブクラスまたは委譲先オブジェクトへのメッセージ送信のかたちに置き換えることで解決できる。(どっちが良いかはコードによって異なる)
- 頻繁に変更のある条件分岐ロジックは、継承ツリー上のある枝の変更を単純化し、かつ他の枝への影響を最小限にするために、メッセージ送信として表現したほうが良い。
- 条件分岐の強みは、シンプルであることと局所的である(単一クラスに収まっている)ことである。しかし、条件分岐が非常に広範になると、この強みは障害となる。

P38 委譲(Delegation)

- インスタンスごとに異なるロジックを実行するためのもう 1 つの方法。(⇔条件分岐)
- 共有の処理は参照する側に、バリエーションは委譲先に持たせる。
- グラフィックエディタでの例。エディタ(参照する側・共有の処理)とツール(委譲先・バリエーション)。エディタの mouseDown()に応じて「矩形作成」や「図形オブジェクトの移動」を表現したい場合。
  - 条件分岐では、新しいツールを追加するたびにエディタの mouseUp()や mouseMove()の条件分岐も重複して修正しなければならない。
  - エディタ(オブジェクト)はその生涯でツールを切り替える必要があるから、サブクラス化しても解決しない。
    →そこで委譲を使う。エディタオブジェクトはツールオブジェクトに処理を委譲する。switch ステートメントの形式から各種ツールの実装に委譲する形式に変える。
- 委譲は既存の共有処理部分を壊さずに、新しいバリエーションを追加できる。
- 条件分岐と比較した場合のデメリット
  - それぞれの実装クラスにロジックが分散するため、コードリーディングの時には条件分岐の場合よりもより多くのナビゲーションが必要になる。
  - ある状況下での委譲側オブジェクトの振る舞いを理解するには、その時点でどの委譲先オブジェクトに委譲しているかを理解する必要がある。
- 委譲先オブジェクトはオブジェクトのフィールドに格納され、動的に評価される。
- Junit4 での例。テストクラスの記述が旧スタイルか新スタイルかを実行時に判別してテストオブジェクト生成処理を委譲している。
- Stream の例?
- 委譲を利用することにした場合、「"自身"へのメッセージ」と言った場合には 2 通りの解釈があり得る。
  - 委譲する側のオブジェクトへのメッセージ
  - 委譲先オブジェクトへのメッセージ
    - GraphicEditor と RectangleTool の例。委譲先である RectangleTool オブジェクトを生成する時に、委譲側の参照を渡す。不変のバックリファレンスを保持させる。
    - インスタンス生成時ではなく mouseDown()の引数として GraphicalEditor の参照を渡すようにすれば、複数の Editor オブジェクトから同一の Tool オブジェクトを利用できる。

P40 差し替え可能なセレクタ(Pluggable Selector)

- リフレクションを使って呼び出すメソッドをランタイムに切り替える。
- Junit の例
  - 最初は 1 サブクラスで 1 テストメソッドのみの実装となっていたため、テストを記述するのが概念的に非常に重かった。
  - メソッドの名前をもとに、テストメソッドをランタイムで収集・実行するようにした。
  - これによってクラス階層も 1 クラスに単純化された。
- このパターンが最初に広く知られたとき、多くの人が濫用する傾向があった。
- リフレクションを使っているため、システムのどこでメソッドが呼び出されるかがわからなくなってしまう。
- このパターンを使う場合はコストを考慮すべき。限定的な使い方をすれば、コストに見合った働きをするだろう。

P41 匿名内部クラス

- Java が提供するもう 1 つのインスタンス特化振る舞いの表現方法。
- 使いどころ
    - 特定の 1 箇所で、非常に局所的な目的のためにメソッドをオーバーライドしてクラスを作りたい場合
    - Runnable のような単純な API
    - 必要な処理の大部分スーパークラスが実装しており、匿名内部クラスの実装が単純になる場合。
- 匿名内部クラスの実装コードは可読性を損なわせる。コードの読み手を混乱させないように、コードは短くする必要がある。
- 匿名内部クラスの制限
    - 匿名内部クラスのコードを記述した場所でしか、そのインスタンスを設定できない。
    - インスタンスが一度作られたら変更できない(?)。
    - 直接テストすることが困難である。よって、複雑なロジックを含むべきではない。
    - 名前を持たないため、コードの意図を名前で表現できる機会を失う。

P41 ライブラリクラス

- ここでいうライブラリクラスの定義
    - どのクラスにもうまく適合しない処理を static メソッドで定義している。
    - インスタンス化不能。
- ライブラリクラスは拡張性がない。全てのロジックを static メソッド化してしまうことによって、オブジェクトを活用したプログラムの利点を失っている。
    - データを共有するプライベートな名前空間を使えば、ロジックをシンプルにできる。
    - 可能であるなら関連するオブジェクトのメソッドに変更すべきである。
- Static メソッドからインスタンスメソッドに順次変更していく例
    1. ライブラリクラスのインスタンスを作り、static メソッドからインスタンスメソッドに委譲するように作り直す。
    2. 複数のインスタンスメソッドで引数のリストが似通っている場合は、コンストラクタの引数として渡してインスタンスのフィールドに保持する。
    3. フィールドに保持するようにしたデータはインスタンスメソッドの引数から除去する。
    4. static メソッドの呼び出し元でインスタンス生成を行うようにし、static メソッドの呼び出しからインスタンスメソッドの呼び出しに書き換える。
    5. ライブラリクラスに定義されていた static メソッドを除去する。
- ライブラリクラスの static メソッドをインスタンスメソッドに変換しようと試みることによって、クライアントコードを読みやすくするためのクラスとメソッドの再命名のアイディアを与えてくれる。