6A-5

リファクタリングパターンの適用・評価と フォワードエンジニアリングへの考察

橋本　憲幸　　位野木　万里

東芝ソリューション（株）

1. はじめに

ソフトウェアの保守性向上には，ソースコード(以下，コード)を理解しやすく，修正や拡張が容易な構造に保つ事が重要である．しかし，時間や予算の制約から，実際の開発現場では保守しやすいコードが生産されるとは限らない．そのため，出来上がったコードを，後から修正することが多いが，そのやり方は形式化されていない．

プログラムの外部から見た動作を変えずにコードの内部構造を改善する手法として，リファクタリングが提案されている[1]．近年，デザインパターン[2]をリファクタリングに適用する試みがされており，これはリファクタリングパターン[3]と呼ばれている．

本稿では，実際のコードにリファクタリングパターンを適用し，コードの保守性を向上させる手順を明らかにする．また，リファクタリングパターンの適用結果から，あらかじめ保守性の高いコードを作成するために注意すべき点を考察する．

2. リファクタリング適用手順

リファクタリングパターンを適用した保守性向上の手順を説明する．我々の考案した手順は次のようになる．

1)　コードの臭いの絞込み(3.1.節)
リファクタリングでは，問題のある箇所をコードの臭い(Code Smell)に比喩して定義している[1]　[3]．限られた時間で改善効果を出すめに，優先して取り組む問題を絞り込む．

2)　適用範囲の選択(3.2.節)
目標とするコードの臭い設定した後，それを発見するために適した方法を選択して，コードの臭いのする箇所を特定する．

3)　適用パターンの選択(3.3.節)
コードの臭いに有効なリファクタリングパターンの例[3]を元に，適用するパターンを選択する．

4)　パターンの適用(3.4.節)
選択したパターンを適用してコードを修正する．

3. 適用事例

リファクタリングパターンの適用事例を示す．リファクタリングの対象は，認証機能を持つ業務アプリケーションとした．対象コードのクラス数は 33 個，実行ステップ数は 13,840 行である．

3.1. コードの臭いの絞込み

我々の経験では，プログラムの保守は，修正作業よりもコードの理解にコストがかかる．特に開発者と修正者が異なる場合，このことは顕著となり，実際の開発現場でも問題となっている．

今回の適用事例も，開発者と修正者が異なるケースであった．そのため，可読性を低下させる次のようなコードの臭いを優先的に取り除くことにした．

- Long Method(長すぎるメソッド)
- Conditional Complexity(複雑な条件記述)

3.2. 適用範囲の絞込み

3.1.節で絞りこんだコードの臭いは，実行ステップ数や if 文の数から発見できる．今回は実行ステップ数が多い上位 3 つのメソッドを対象とした．以下，methodA(), methodB(), methodC()と呼ぶ．なお，ステップ数と条件分岐の数は，ほぼ比例していたため，対象としたメソッドは条件分岐の数が多いメソッドでもある．

3.3. 適用パターンの選択

3 つのメソッドに *Compose Method* を適用して，1 メソッドあたりの長さを短くする．*Composed Method* とは，分りにくいロジックを，意図の伝わりやすい，詳細レベルの揃った小さなメソッドの呼出しに置き換えることである．

ところで，methodC()は if-else 文で処理の振分けを行っており，1 つの条件ブロックが長いため，メソッド全体も長くなっている．また，実際の開発者にヒアリングした結果，将来，画面のメニュー項目に依存して，分岐の数が増減する可能性があることが分った．そこで，*Replace Conditional Logic with Strategy* を適用し，条件分岐を取り除く．*Replace Conditional Logic with Strategy* とは，アルゴリズムの選択を行っている条件分岐をサブクラスに置き換えることである．

3.4. パターンの適用

Replace Conditional Logic with Strategy の適用結果を図 1 に示す．

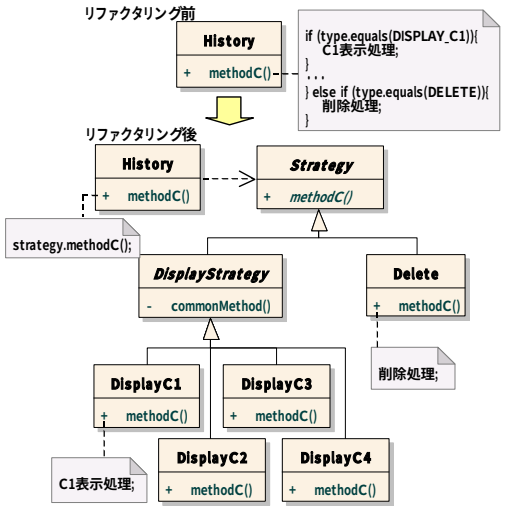

図 1：*Replace Conditional Logic with Strategy* の適用結果

リファクタリング前の methodC()は History に定義され，画面からの選択に応じて C1～C4 の 4 種類の履歴表示処理と削除処理を条件分岐で振分けていた．リファクタリング後のクラスについて説明する．Strategy のサブクラスは，処理の性質を表示と削除の観点から分類し，

Refactoring Pattern Adoption and Evaluation with Reference to Consideration for Forward Engineering
Noriyuki HASHIMOTO, Mari INOKI
Toshiba Solutions Corporation

DsiplayStrategy と Delete の 2 つを作成した．表示処理は C1～C4 の 4 種類があるので，DsiplayStrategy のサブクラスとして，DisplayC1～DisplayC4 の 4 つのサブクラスを作成し，図 1のように継承構造を 2 階層とした．また，DisplayC1～DisplayC4 間で共通処理があったため，DsiplayStrategy に引き上げ commonMethod()として定義した．

Compose Method の適用で工夫した点を示す．メソッドを抽出するにあたって，ローカル変数の整理を行った．特に，中間データを取得するコードと，それを使うコードが離れているため，一見するとメソッドにまとめることが難しい箇所がいくつかあった．このような箇所は，副作用がないことを確かめた後，データが必要となる直前に取得するようにコードを移動した．

4. 評価

我々はリファクタリングによる改善効果を，次の方法で評価した．

1) メソッド単位の実行ステップ数と分岐の数(4.1.節)
2) クラス単位の実行ステップ数とメソッド数(4.2.節)
3) コードレビュー(4.3.節)

1)で局所的な定量評価を，2)で大局的な定量評価を，3)で定性的な評価を行う．

1)はリファクタリング対象である 3 つのメソッドのみを測定した．2)は図 1の *Replace Conditional Logic with Strategy* の適用前後のクラスを測定した．3) はリファクタリング後のソースコードと UML のクラス図を元に，実際の開発者を交えて評価した．

4.1. メソッド単位の実行ステップ数と分岐の数

リファクタリング前後の測定結果を表 1に示す．実行ステップ数は，3 つのメソッド全て減らすことができた．

また，methodC()にあった 7 つの条件分岐のうち，5 つはサブクラスへの置き換えで減らすことができた．残りの 2 つはエラー処理であるため残したが，最終的に別メソッドに抽出したため，表 1での条件分岐の数は 0 となった．なお，methodA()と methodB()の条件分岐の数も減っているが，これは分岐そのものが減ったわけではなく，別のメソッドに移動したためである．

表 1：リファクタリング対象の測定結果

メソッド名	実行ステップ数		条件分岐の数	
	前	後	前	後
MethodA	428	118	34	9
MethodB	84	8	8	0
MethodC	82	9	7	0

4.2. クラス単位の実行ステップ数とメソッド数

図 1のリファクタリング前後の測定結果を表 2に示す．リファクタリング後の実行ステップ数は約 2 倍増加したが，1 メソッドあたりの平均実行ステップ数は約 1/10 に減った．また，クラス数，メソッド数は増加した．

表 2：*Replace Conditional Logic with Strategy* の測定結果

	実行ステップ数	クラス数	メソッド数	実行ステップ数/1 メソッド
前	125	1	1	125.0
後	215	8	18	11.9

4.3.コードレビュー

実際の開発者にコードレビューを行い，次の意見を得た．

- クラス数は増えたため，管理が難しくなった可能性がある．
- 1 つのメソッドに混在していた複数の処理が，クラスやメソッドとして見えるようになり，全体としては可読性が向上した．

5. 考察

5.1. リファクタリングパターンの考察

メニュー項目の増減に応じて分岐の数が変化するような場合，*Strategy* パターンを適用して処理をサブクラスにカプセル化すると，画面のメニュー項目が増減してもサブクラスを交換するだけで対応可能になり，拡張性が高まる．今回は性質の異なる処理が条件分岐に混在していたため，継承構造を 2 階層とした．図 1のように，クラス図上にコードの特性を表現することができたことは，コードの理解に有用であった．

5.2. フォワードエンジニアリングへの適用可能性の考察

適用および評価の結果を設計手法へフィードバックし，あらかじめコードの臭いを取り除いておくために注意すべき点を考察する．

まず，*Composed Method* から得られた考察を示す．処理が複雑でメソッドが長くなる場合，以下の処理のまとまりを意識してメソッドを抽出すると，元のメソッドを短くできると考えられる．

1) 入力チェック
2) 処理に必要なデータの取得
3) 処理本体
4) 後処理(データ保存，遷移先決定など)

ただし，2)と 3)は完全に分けて考えず，特定の処理でしか使わないデータは処理本体の直前で取得するようにし，変数の有効範囲を狭くするとメソッドを抽出しやすい．

次に，*Replace Conditional Logic with Strategy* から得られた考察として，設計段階から変化の起きる箇所を洗い出しておくと望ましいことが言える．ただし，初めから変化の起こる箇所を確実に予測できるとは限らないため，コーディング段階で条件分岐の数が基準値より多くなったら，*Strategy* パターンの適用を考慮すると効率的である．

6. まとめ

リファクタリングパターンを適用したコードの保守性を向上させる手順を明らかにした．適用結果から設計手法へのフィードバックの可能性を考察し，あらかじめ保守性の高いコード作成する手法を整理した．

今後の課題としては，コードの特性を浮かび上がらせる方法は，その特性に応じて様々な方法があり，さらに多くの実施例を通じてノウハウを溜める必要がある．また，あらかじめ保守性の高いコード作成する手法について，妥当性を検証する必要がある．

参考文献

[1] Martin Fowler, Refactoring Improving the Design of Existing Code, Addison-Wesley, 1999.
[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Design patterns: Elements of reusable object orientated software, Addison-Wesley Professional, 1995.
[3] Joshua Kerievsky, Refactoring to Patterns, Addison-Wesley, 2004.