

Five recommendations of software evolvability

Václav Rajlich

Department of Computer Science,
Wayne State University,
Detroit, MI 48202, U.S.A.

目次

- 1.前置き
- 2.背景
- 3.目的
- 4.推奨事項
- 5.まとめ・所感

進化的ソフトウェア開発

- Evolutionary software development(ESD)
- **既存のソフトウェア**に、新たな関数やプロパティを追加する一連のプロセス
- ESDプロセスは、ソフトウェアが新たな関数やプロパティの追加を許可しているなど、進化可能な時のみ使われる

```
// C# のプロパティ構文を利用した際のフィールドアクセスの例
Person person = new Person();
// set アクセス
person.Name = "John";
// get アクセス
string userName = person.Name;
```

Wikipedia 「プロパティ_(プログラミング)」より引用

背景

- ソフトウェアのほとんどは、進化的ソフトウェア開発により開発されている¹
- 進化可能性が損なわれると、ソフトウェアをさらに進化させることはできず、プロジェクトは大きな問題に直面する
 - bad smellsやtechnical debt
- ソフトウェアの進化可能性は重要なソフトウェア品質である

1. Rajlich, V. Software evolution and maintenance. Proceedings of the on Future of Software Engineering. 2014:133-144.

背景

- ソフトウェアシステムの進化可能性は3つの要因の相互作用により成り立つ
 - system properties
 - human factors present in developer team
 - evolution demands
- しかし、進化可能性に関する研究のほとんどはソフトウェアのプロパティのみを扱っており、人的要因と進化の要求については間接的に触れているだけである

目的

- ソフトウェアの進化性を維持または改善する方法についての5つの推奨事項の提案をする
 1. ソフトウェア変更の定義されたプロセス
 2. コードの進化部分と安定化部分の区別
 3. 分析可能なコードセグメント
 4. 重要な概念のカプセル化
 5. ラッピングの回避

1. ソフトウェア変更の定義されたプロセス

ソフトウェアを変更する際に、
定義されたプロセスを用いることを推奨する

- ソフトウェア開発プロセスの成熟度が高いほど、製品の品質と生産性が向上する²²
→ソフトウェアの変更をする際にも同じことが言えるのでは？
- 開発者やタスクごとに異なる未定義の変更プロセスを使うと、生産性と品質を予測できないという問題がある

22. Harter DE, Krishnan MS, Slaughter SA. Effects of process maturity on quality, cycle time, and effort in software product development. Manage Sci. 2000;46(4):451-466.

PMSC

(Phased model of software change)

- ソフトウェア変更のプロセスモデル
- 要件の抽出, 分析, および優先順位付けを行った後, 以下の一部またはすべてを行う
 - コンセプトロケーション, 影響分析, アクチュアライズ, リファクタリング, プリファクタリング, 検証
- PMSCを使用した場合の開発者の生産性の向上の実証²⁷

27. Senin Y. Case study of phased model for software change in a multiple-programmer environment. M.S. thesis, Department of computer Science. Detroit, MI.: Wayne State University; 2014.

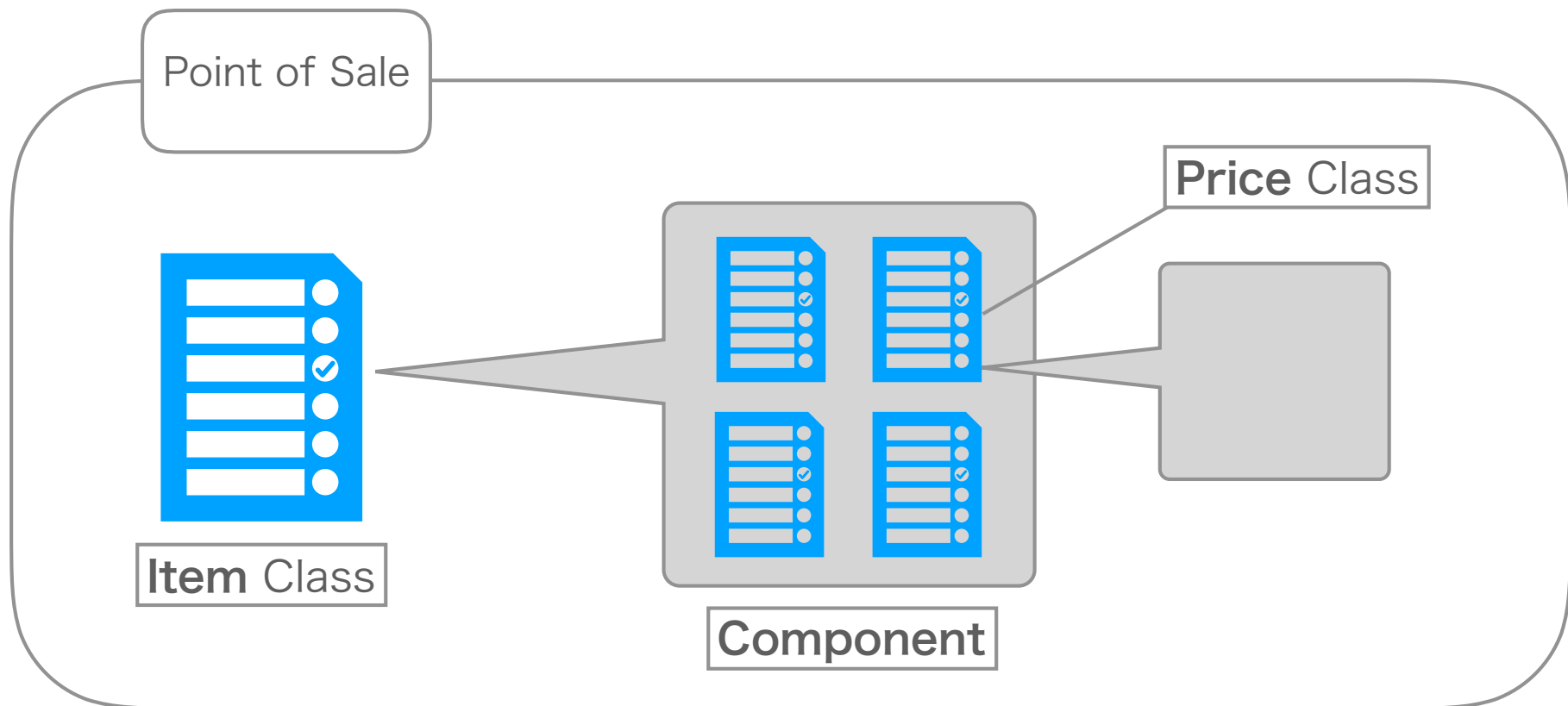
2. コードの進化部分と安定化部分の区別

- 進化可能部分
 - 将来的に進化性の修正を受ける
 - 安定化部分
 - 将来的にほとんど進化しない（ライブラリなど）
 - 進化可能でない
- 開発者個人で進化可能部分・安定化部分の判断をすることは危険

進化可能部分・安定化部分の明確な区別と、
ステークホルダーがどこに関係しているかを扱う
プロセスを持つことを推奨する

3. 分析可能なコードセグメント

- ソフトウェアの分析可能性は進化可能性の一部と言われている^{3,4}
- 分析可能性 → コードセグメント（コードの責任範囲）の明確化



3. Breivold, HP, Crnkovic I, Eriksson PJ. Analyzing software evolvability. In Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE Inter- national. 2008:327-330.

4. Organización Internacional de Normalización, ISO/IEC 25010:2011 Systems and software engineering—systems and software quality requirements and evaluation (SQuaRE)—system and software quality models. 2011: ISO.

3. 分析可能なコードセグメント

- 開発者は分析ツールでセグメントの範囲を決定する
- 分析できないソフトウェア構造が存在する³¹
 - C++のポインター演算, Javaのリフレクションなど
- これらの構造を進化可能な部分で無計画に使用するのは危険

進化可能なコードの分析可能性の保存が必要

31. ivshits B, Sridharan M, Smaragdakis Y, et al. In defense of soundness: a manifesto. Comm ACM. 2015;58(2):44-46.

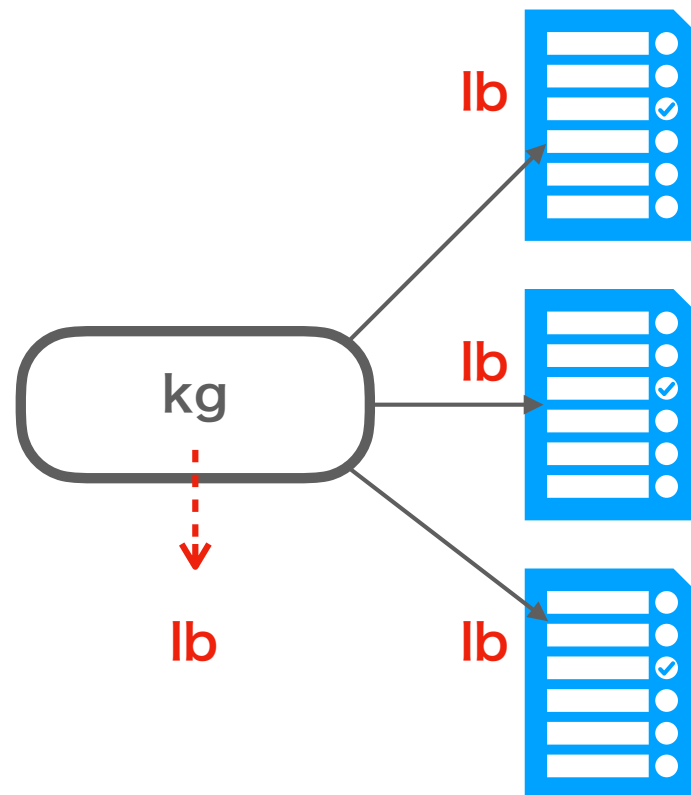
4. 重要なオブジェクトのカプセル化

- オブジェクトが、適切な名前やコードでカプセル化されていると、進化は容易である
 - 発見、理解および修正が容易
- しかし、開発者は重要なオブジェクトのカプセル化をしない時がある^{33,34}
 - コード内のクラスが多すぎないようにする時
 - プログラムの全体的な構造にほとんど寄与しないと思われる些細なクラスであると判断した時

33. Yu Z, Rajlich V. Hidden dependencies in program comprehension and change propagation. In: International Workshop on Program Comprehension. 2001: 293-299.

34. Eaddy M, Zimmermann T, Sherwood KD, et al. Do crosscutting concerns cause defects? IEEE transactions on Software Engineering. 2008;34(4):497-515.

4. 重要なオブジェクトのカプセル化

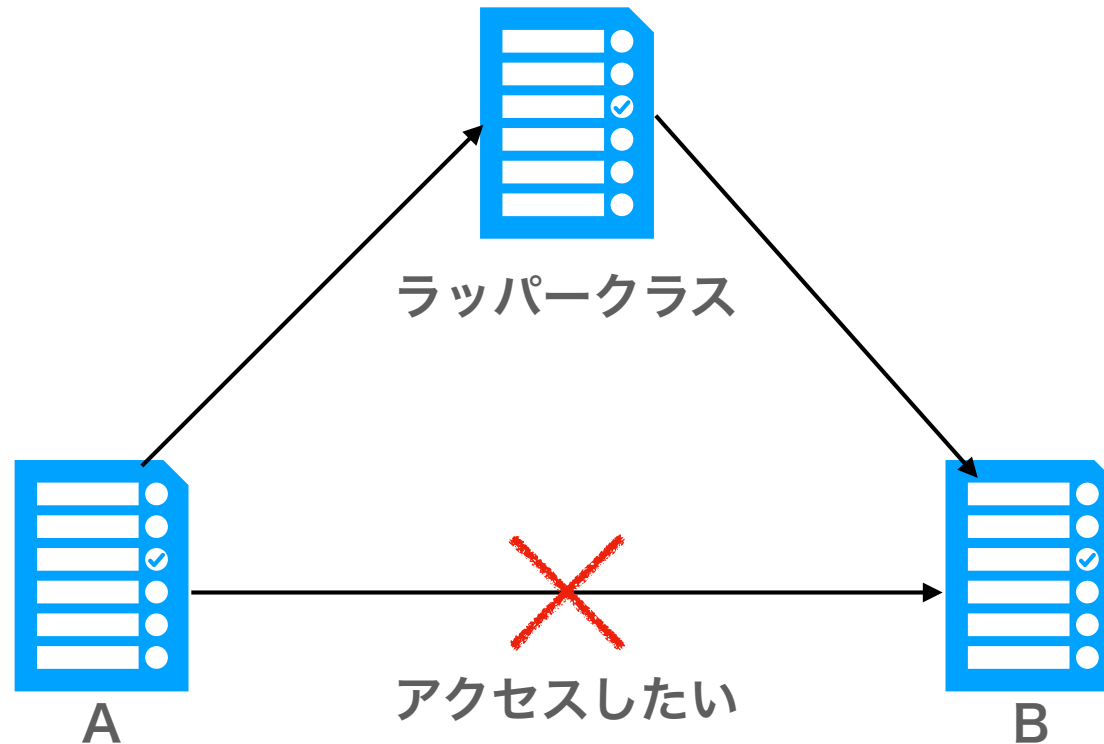


- 大規模な変更
- 異なる変更プロセス, ツールが必要となる

重要なオブジェクト
のカプセル化

5. ラッピングの回避

ラッパークラスを使うと、依存関係がある2つのモジュール間の不一致を迅速に修正できる



5. ラッピングの回避

- ただし、ラッパークラスを使うと
 - Aクラスの実装を非局在化する
 - 結果のコードは、元のオブジェクトではなくラッパークラスのコードを反映する

➡進化性を著しく下げる

安定化部分のコードでのラッパーの使用 ○

進化可能部分のコードでのラッパーの使用 ✕

まとめと所感

- まとめ

- 進化的ソフトウェア開発プロセスの5つ推奨事項を提案した

- 所感

- technical debtの研究に活かそうだと思った