ソフトウェア工学 第13回
― ソフトウェア保守と再利用 ―

大連理工大学・立命館大学 国際情報ソフトウェア学部

大森 隆行

講義内容

- ソフトウェア保守
- ソフトウェア再利用

ソフトウェア保守

- ソフトウェア保守 (software maintenance)
  - ソフトウェアを維持・管理する作業
  - 目的
    - 常に正しく稼働させる
    - 顧客の要求に合わせて変更・修正する
  - 意義
    - ソフトウェアは経年劣化がない＝利用期間が長い
    - 導入後も変更が容易
      → 欠陥の修正や新しい要求への対応が求められる
- ソフトウェア進化 (software evolution)
  - M.M. Lehmanにより提唱
  - ソフトウェアの持続だけでなく、新しい要求を受け入れ、機能拡張、性能改善を目指す

システムの種類

- S-system (specifiable)
  - 機能を形式的に定義できる
  - 仕様から容易に導出可能
  - プログラムを変更しても入出力関係が不変
    or プログラムとして別物に変化
- P-system (problem-solving)
  - システムが解くべき問題は明確に定義できるが、
    実際には近似解を求める (e.g., チェスのAI)
  - 解が受け入れられる条件が変わると変更が求められる
- E-system (embedded)
  - 現実世界の変化に応じて進化しなければならない
  - 解くべき問題自体が変化

保守の対象

- 保守はソフトウェアの完成後に行われる
  → それまでに作成された全文書が保守の対象
  - 要求仕様書、基本設計書、詳細設計書、
    ソースコード 等
- 自分で作成した文書だけが対象とは限らない
  - 作成時点でわかりやすいものを作る必要がある
  - 既存の文書の理解支援も必要

保守の戦略

- 古いものを使い続ける vs 新しいものを作る
  - 新しいものを作った方が安い場合もある

- レガシーソフトウェア (legacy software)
  - 古いが価値があるソフトウェア
    - 保守費用がかかるが、すぐに破棄できない
  - 変更を繰り返すことでソフトウェアの構造が劣化することに注意
    - 初めの設計からの乖離

保守の種類

- 修正保守
  - 出荷後に発見された誤りや障害を修正
- 適応保守
  - 利用環境の変化に追従するための修正
- 完全化保守
  - 品質向上のための修正
    - e.g., 実行速度の高速化、
      保守性向上のためのモジュール再構成
- 予防保守
  - 故障を防ぐための修正
    - e.g., defensive programmingの導入

確認問題

- 空欄に当てはまる適切な語句を選択肢から選べ。
  - ソフトウェア保守とは、ソフトウェアを(　　)する作業である
    - 選択肢：　設計・開発　　維持・管理　　テスト・検証
  - 保守費用がかかるが、業務の都合等によりすぐに廃棄できないソフトウェアを特に(　　)ソフトウェアと呼ぶ。
    - 選択肢：　レジェンド　　ガベージ　　レガシー
- 以下の各説明に当てはまる保守作業の名称はどれか。語群から選んで答えよ。
  - 品質をさらに完全に近づけるための保守作業
  - 故障を防止するための保守作業
  - 誤りや障害を修正するための保守作業
  - 利用環境の変化に対応するための保守作業

語群：
修正保守、完全化保守、
予防保守、適用保守、適応保守

- 以下の各文は正しいか。○か×で答えよ。
  - ソフトウェア保守では、既存機能の性能を高める必要はあるが、新しい機能を導入する必要はない。
  - ソースコード以外の関連文書もソフトウェア保守の対象となりうる。
  - 自分が作成した文書だけがソフトウェア保守の対象とは限らない。
  - 一般的には、完成したソフトウェアに何度も変更を加えることにより、保守性は向上していくとされている。

保守の技法

- ソフトウェア理解
  - プログラム理解支援
- 構成管理
- 影響分析
- リバースエンジニアリング
- 回帰テスト

ソフトウェア理解

- 保守の際、現在のソフトウェアについて理解する必要
  - どのような要求により開発されたか？
  - どのような設計指針で開発されたか？
  - どのようなアーキテクチャを採用？
  - どのようなモジュール構成か？
  - どのようなプログラムで実現されたか？
  - どのようなテストが行われたか？
- 特に、現状のプログラムの理解が必須

プログラム理解支援

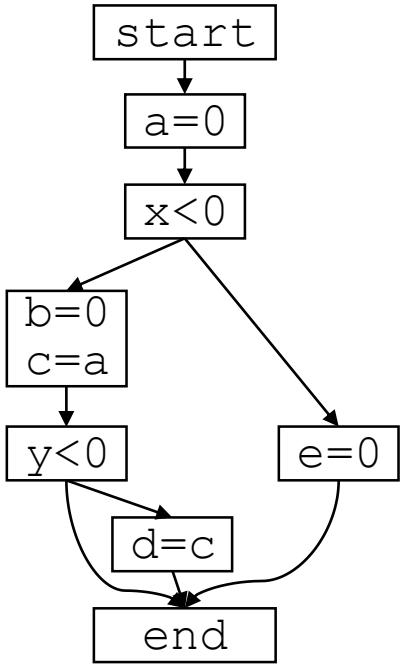
- プログラムの構成と対応する要求は
  各種設計仕様書・要求仕様書等で
  追跡可能であることが理想
  → 実際には文書間で矛盾が発生

- プログラム理解支援 が必要
  - 実際に稼働しているプログラムの理解を支援
  - 文書間に矛盾がある場合、信頼できるものは
    稼働しているプログラム

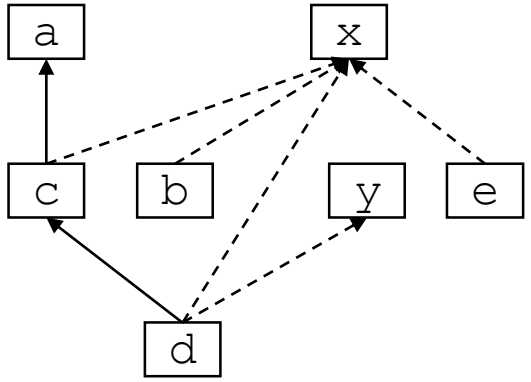
プログラム理解支援の技法　1/2

- 制御フローの視覚化、データの依存関係の視覚化

```
a=0
if(x<0){
 b=0
 c=a
 if(y<0){
  d=c
 }
}else{
 e=0
}
```



制御フロー



データ依存関係

プログラム理解支援の技法 2/2

- プログラムスライス (program slice)

```
 1: int func(int data[]) {
 2:    int sum = 0;
 3:    int prod = 1;
 4:    int i = 0;
 5:    while (i < data.length()) {
 6:       sum = sum + data[i];
 7:       prod = prod * data[i];
 8:       i = i + 1;
 9:    }
10:    print(sum);
11:    print(prod);
12: }
```

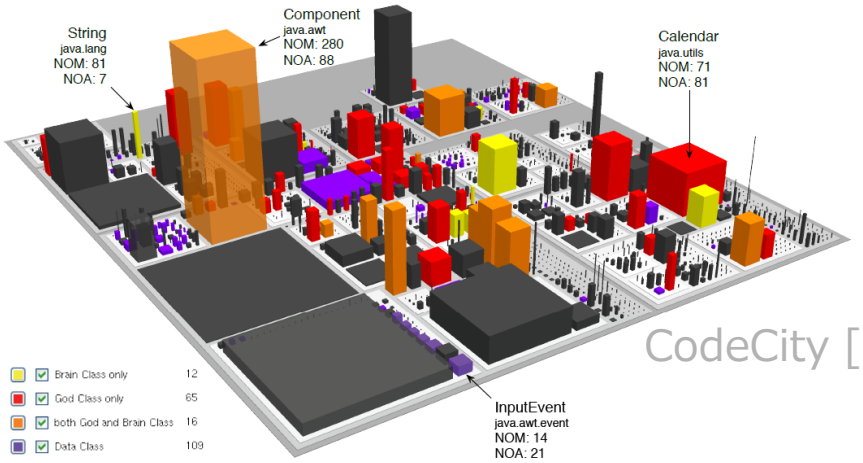
元のプログラム

```
 1: int func(int data[]) {
 2:    int sum = 0;
 3:
 4:    int i = 0;
 5:    while (i < data.length()) {
 6:       sum = sum + data[i];
 7:
 8:       i = i + 1;
 9:    }
10:    print(sum);
11:
12: }
```

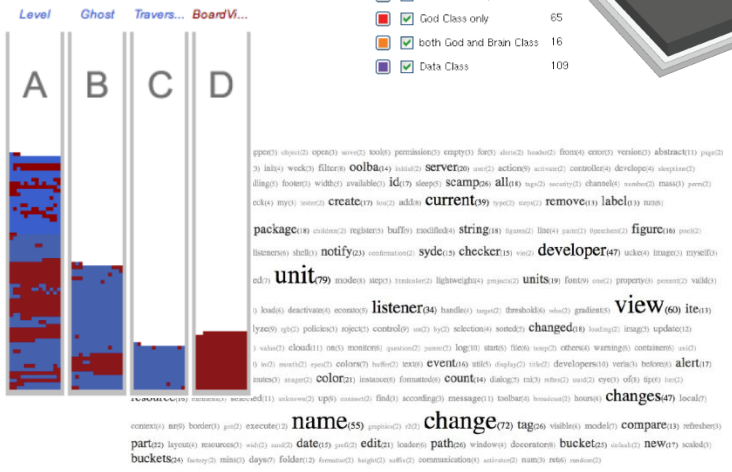
文10のsumに関する静的逆方向スライス

ソフトウェア視覚化 (visualization)

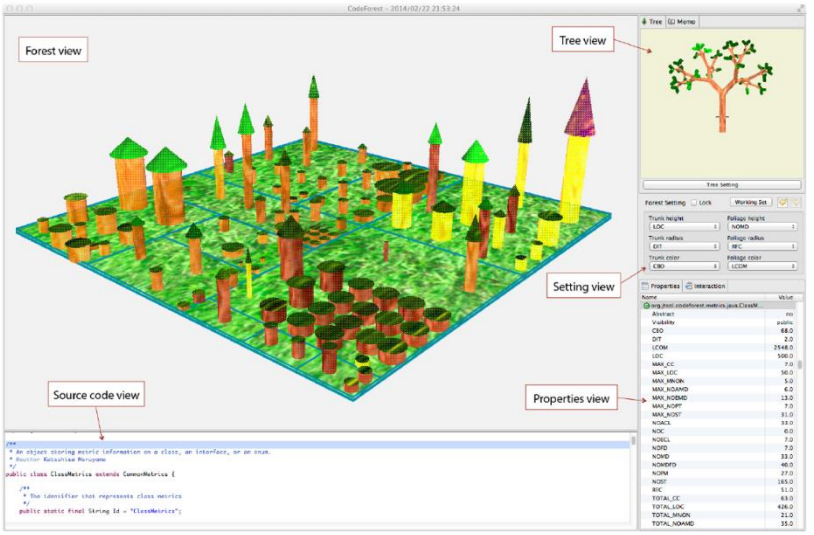
- ソースコード等を視覚的に表現



CodeCity [wettel-icse11]

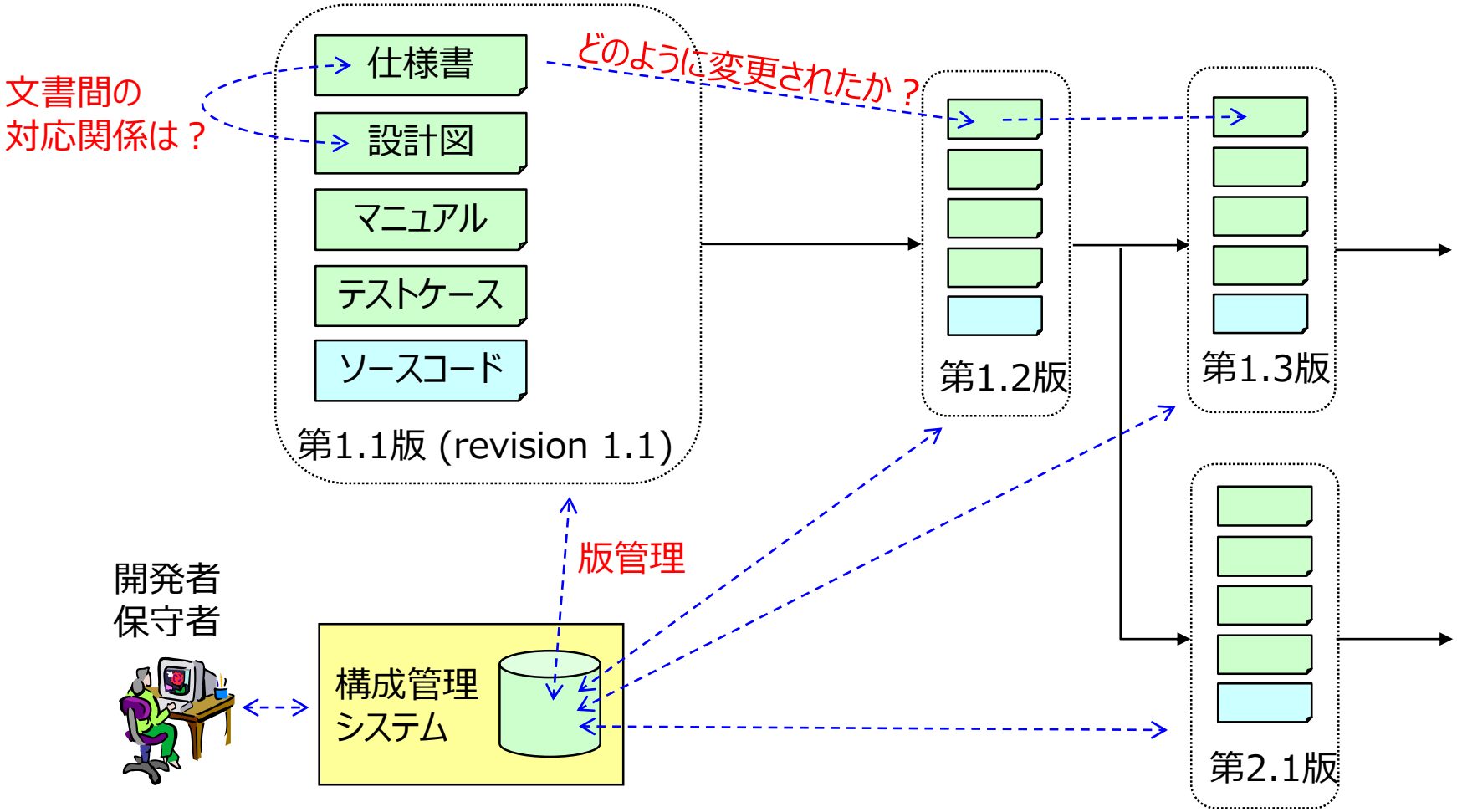


Syde [lanza-csmr10]



CodeForest [maruyama-icpc14]

構成管理 (configuration management)

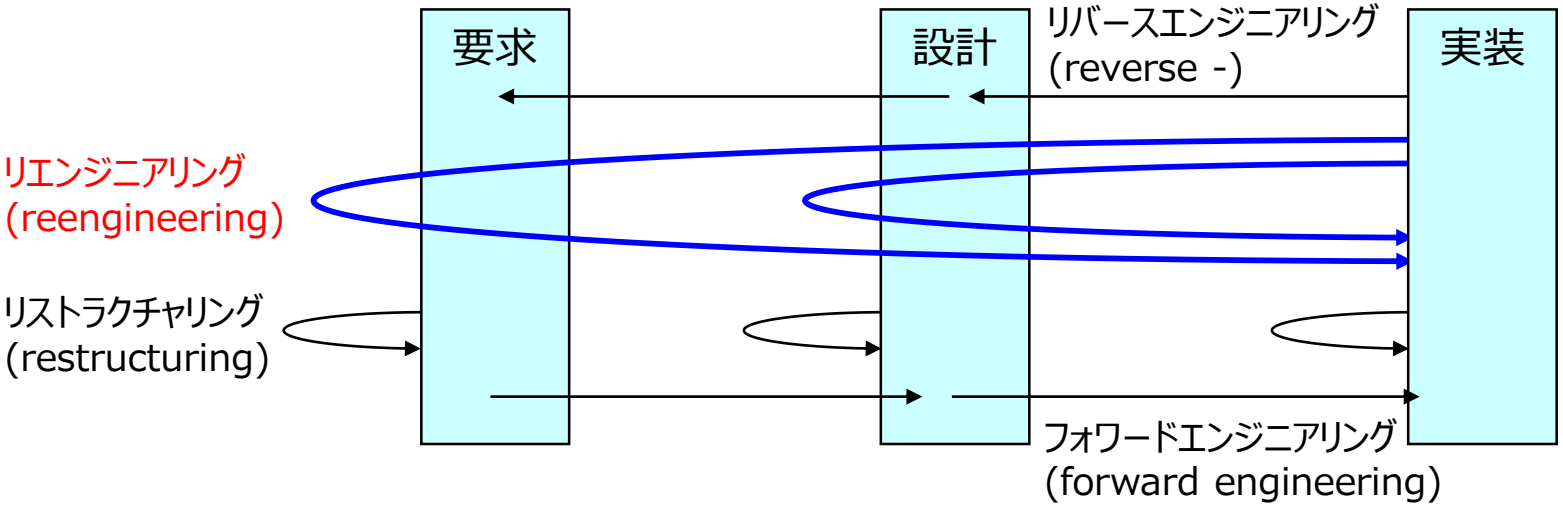


影響分析

- 影響分析
  - ソフトウェアの変更の影響を受ける箇所を分析すること
  - 波及効果解析(ripple effect analysis)とも呼ぶ
- 影響分析の必要性
  - ソフトウェアの構成要素は互いに密接に関連
    (e.g., 変数・関数・クラス等の参照関係、ポインタによる参照)
    → 一つの変更が他の箇所に影響を及ぼす
    → 予期しない影響が発生することもある
    → 障害が発生
- 影響分析の技法
  - プログラム理解支援を利用
    (e.g., 変更箇所に関する制御やデータの依存関係)

リバースエンジニアリング

- リバースエンジニアリング
  - 後工程の成果物から前工程の成果物を作成
- フォワードエンジニアリング
  - 通常の開発
- リエンジニアリング
  - リバースエンジニアリング ＋ フォワードエンジニアリング
- リストラクチャリング
  - 特定の種類の成果物を改善 (ソフトウェアの振る舞いは不変)



- ✓ 成果物間の矛盾の解消
- ✓ ソフトウェアの再構成による設計・実装改善

リファクタリング (refactoring)

- リファクタリング
  - 既存のソフトウェアの外部的な振る舞いを変えることなく、実装を改善する作業
  - コードの理解性や保守性を向上させる
  - リストラクチャリングの一種
  - コード変換の目的や内容が「カタログ(catalog)」としてまとめられている
    - Rename: 既存の構成要素の名前を変更する
    - Extract Class: 既存クラスの一部を新しいクラスとして抽出
    - Move: クラスの構成要素を他のクラスに移動する
    - Pull Up: クラスの構成要素を親クラスへ移動する
    - Push Down: クラスの構成要素を子クラスへ移動する
    - 他多数

回帰テスト (regression test)

- 変更前のソフトウェアで動作していた機能が、変更後も正しく動作するか検査する作業
  - テストケースの内容は元のテストケースと同じで良い
  - 誤り混入時の対応を容易にするため、少し変更するたびに行うのが良い
  - 何度もテストを行う手間がかかる
    → テストの自動化が有用

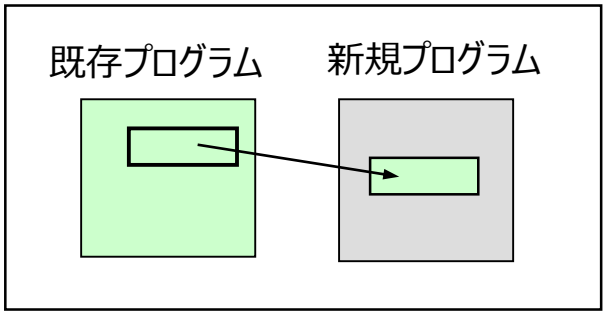
講義内容

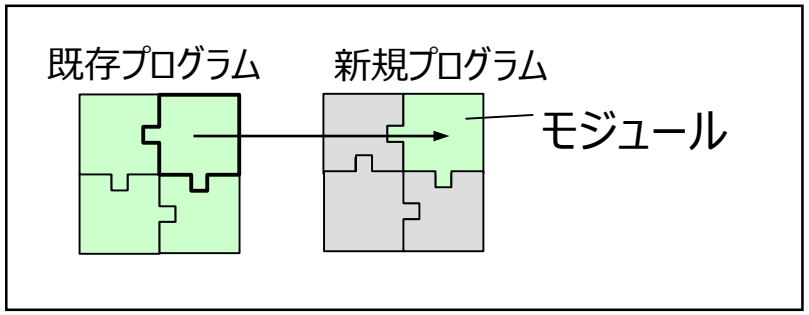
- ソフトウェア保守
- ソフトウェア再利用

ソフトウェア再利用

- 既存のソフトウェア成果物を新しい開発で利用すること
  - 利点：作業効率の向上、安全性・信頼性の向上
  - 注意点：再利用元の品質、不適切な再利用
  - 例
    - 設計書やソースコードの一部をそのままもう一度使う
      → ライブラリ化、コンポーネント化(次頁参照)することもある
    - 開発で得られた知見を再利用
      → ソフトウェアパターン：ソフトウェア開発において何度も現れる問題とその解決策を体系的にまとめたもの
      - e.g., デザインパターン、コーディングパターン

コンポーネント

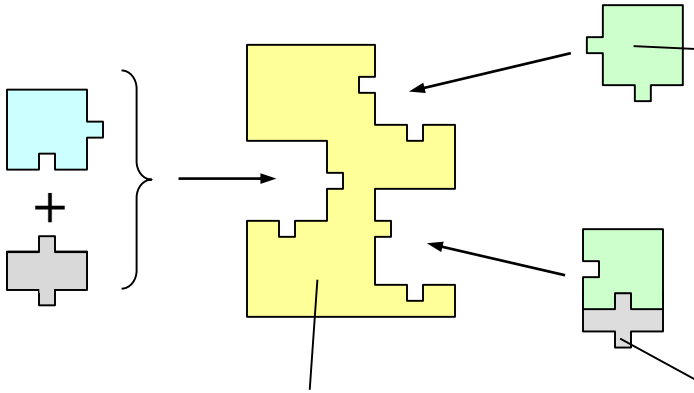




コードの再利用

モジュール [コード + インタフェース] の再利用

コンポーネント [(コード) + インタフェース + 設計] の再利用



ソフトウェアアーキテクチャ
(software architecture)

コンポーネント(component)
- 標準化された特定のアーキテクチャの上で動作
- 実装が隠蔽され、インタフェースのみが公開
- 他のコンポーネントと置換可能
- 複数のバイナリファイルで構成

インタフェース(interface)
- 提供サービスあるいは要求サービスを明確に定義

確認問題

- 以下の各文は、ソフトウェア保守のための技法を説明したものである。それぞれどの技法を説明したものか、語群から選んで答えよ。
  - 変更後もソフトウェアの機能が正しく動作するか検査する。
  - ソフトウェアに変更を加えた時に影響を受ける箇所を分析する。
  - ソースコードから設計や要求を遡って生成する。
  - 既存のソフトウェアの外部的な振る舞いを変えることなく、実装を改善する。
  - テキスト形式の文書を適切なメタファを用いて視覚的に表現する。
  - 様々なソフトウェア成果物を一貫して管理する。

    リファクタリング、リバースエンジニアリング、フォワードエンジニアリング、回帰テスト、影響分析、構成管理、ソフトウェア視覚化
- 以下の各文の空欄に当てはまる適切な語を選択肢から選べ。
  - ソフトウェア再利用の一形態として、開発において頻出する問題とその解決策を体系的にまとめたソフトウェア(　　)がある。
    - 選択肢：　パターン　　プロセス　　メトリクス
  - コードやインタフェースのみでなく設計まで含めて再利用するとされるソフトウェア再利用の単位として、(　　)がある。
    - 選択肢：　オブジェクト　　メソッド　　コンポーネント

参考文献

- 「ソフトウェア工学」
  高橋直久、丸山勝久 著、森北出版、2010
- 「ソフトウェア工学 理論と実践」
  シャリ・ローレンス・プリーガー 著、堀内泰輔 訳、
  ピアソン・エデュケーション、2001
- Martin Fowler
  https://refactoring.com/catalog/