2023年度
ソフトウェア工学
ソフトウェアの保守と発展

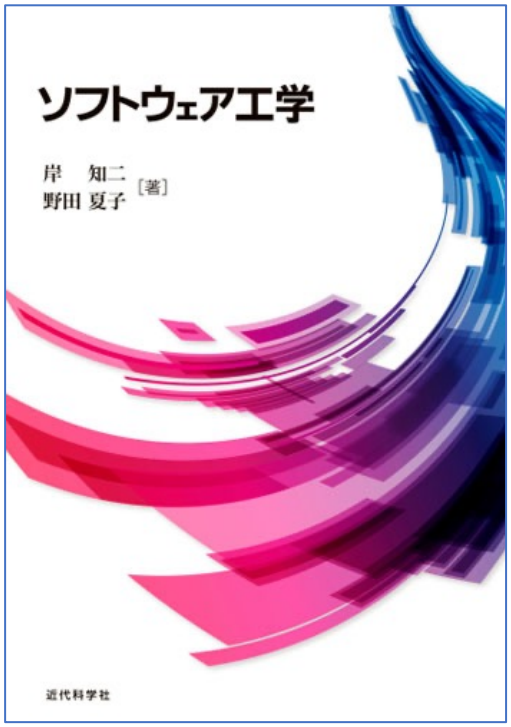
2024年 1月22日

渥美 紀寿 (情報環境機構)

京都大学

参考文献

- 岸 知二，野田 夏子著：ソフトウェア工学，近代科学者



保守性

- 障害発生や要求・環境の変化に応じたソフトウェアの改訂を行いやすいこと

- 解析性
  - 欠陥や故障の原因分析，修正箇所の分析が容易
- 可変性
  - 改良・改訂のための変更が容易
- 安定性
  - 改良・改訂によって予期せぬ影響を及ぼさない
- 試験容易性
  - 変更の妥当性が確認しやすい
- 適合性
  - 保守性が法規や規格を遵守している

ソフトウェア保守

- ソフトウェア開発ライフサイクルの最後
- リリース後の運用中の工程
- ソフトウェアの改良，最適化，不具合修正
  - 性能，効率，ユーザビリティの改善
  - 運用中に発見された不具合の修正
  - 新たな環境への適応，新機能の追加
- ソフトウェア開発ライフサイクルでかかるコストの 2/3 が保守に費される[1][2]

[1] Meilir Page-Jones, *Practical Guide to Structured Systems Design*, Yourdon Press, 1981.
[2] lAlain April, Alain Abran, *Software Maintenance Management:Evaluation and Continuous Improvement*, Wiley, 2008.

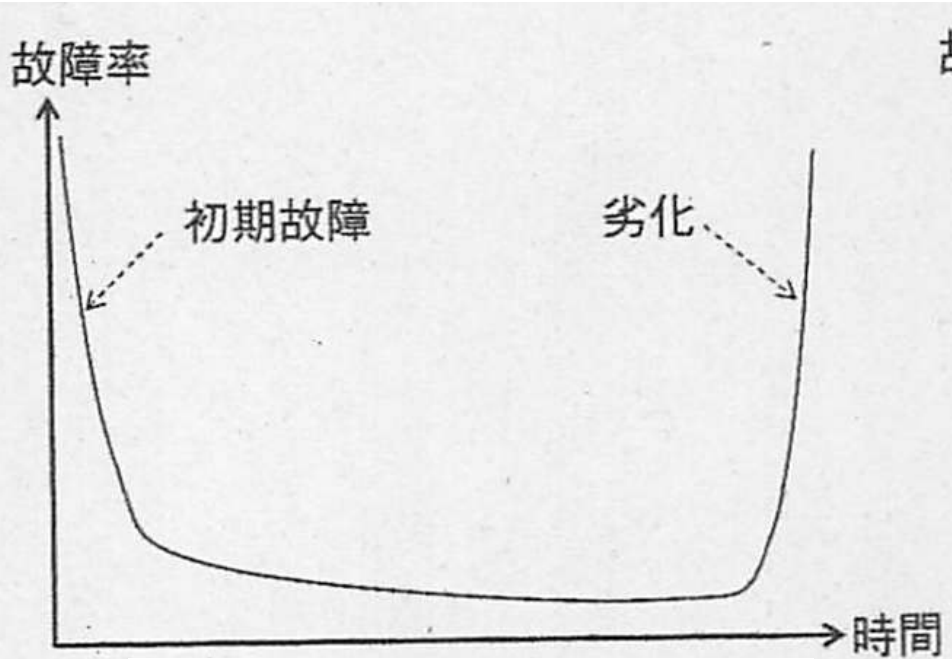
ソフトウェア保守の区分

- 修正保守
  - 運用後にみつかった欠陥や誤りの修正
- 適応保守
  - 運用後の環境の変化に対応するために
    ソフトウェアの変更
  - 環境の変化：ハードウェア，OS，ネットワーク，
    他のシステムとの連携
- 改善保守
  - 運用で必要性が明らかになった機能的・非機能的改善
- 予防保守
  - 潜在的な問題をみつけ，対応できるように変更

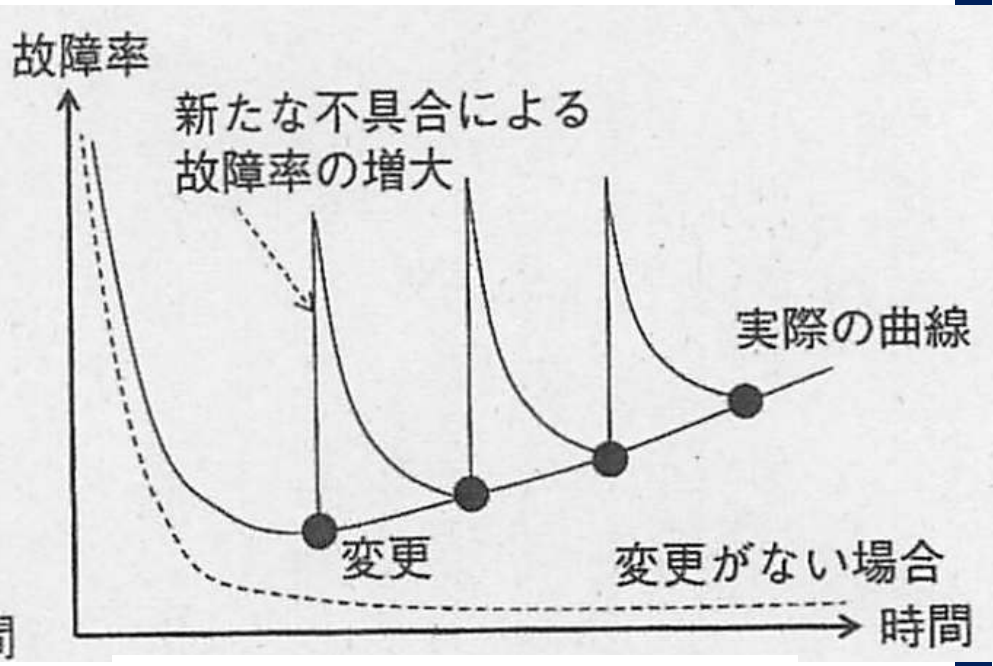
ソフトウェア保守の区分と性質

区分	保守のタイミング	修正か拡張か
修正保守	問題発生後対応	修正
適応保守	問題発生後対応	拡張
改善保守	前後両方あり	拡張
予防保守	先回り対応	両方あり

ハードウェアとソフトウェアの故障率曲線



ハードウェアの故障率曲線



ソフトウェアの故障率曲線

岸 知人, 野田 夏子著: “ソフトウェア工学”, 近代科学者 p.193 から引用

ソフトウェアの保守と発展のプロセス

- ソフトウェア開発ライフサイクルの中で最も長い工程
- 運用を通してユーザの要求を満足し，価値を保ち続けるようにする
- 廃棄や次期システムへの移行時期をあらかじめ決めることは難しい ⇒ 管理が難しい

ソフトウェア保守の体制

- 開発チームと保守チームを分ける
- 利点
  - 保守と開発の予算，資源等を明確に区分でき，計画と管理の室が高まる
  - 保守はユーザからの不定期な要求に対応するため，作業負担が一様でないため，兼任の場合，開発の進捗への影響も
  - 利用者の保守要求に対して迅速できめ細かい対応が可能
- 欠点
  - 開発時の知識を保守チームに受け渡すコストが高く，知識伝授がうまくいかない
  - 保守チームのメンバの士気や技術力の低下

保守作業のステップ

- 変更要求に対応するかどうかの検討
- 当初要求と変更要求との関係を把握
- 変更要求をどのように実装するか分析・設計
- 実装とテスト (回帰テスト)

ソフトウェアの保守期間

- ソフトウェアの寿命
  - 数日～数十年
  - 新たに作り直される場合でも機能的に同種のソフトウェアなら世代を跨いだ進化と見ることも
- 変更にかかる期間
  - 修正箇所を容易に特定でき，修正範囲が狭ければ短時間で終わる
  - 大規模な変更なら数年かかることも
- 変更の適用
  - 運用を停止することなく動的に更新
  - 運用を停止してからシステムの再構成，機能

ソフトウェアの発展

- ソフトウェアが新たな環境に適合できるように変更すること
- ソフトウェアは変化しつづける性質を持つ
  - この変化を発展と呼ぶ
    (進化と呼ぶこともある)
- ソフトウェアの保守と同じことを指すが,
  より前向きな印象を与える
- ソフトウェア：発展・進化する
- 人：ソフトウェアを保守する

ソフトウェアの変更規模

- 1文や1文字の修正
  - break 文忘れなど
- 全体の数十%
  - 機能単位
  - 複数の機能に跨がる修正
- 作り直し
  - 多くのモジュールに依存した修正
  - アーキテクチャレベルの変更など

Belady & Lehman によるシステムの進化モデル

- 継続的変化の法則
  - 使用されるシステムは，変更を凍結し，作り直す方がコスト的に有利になるまで，継続的な変化を続ける
- エントロピー増大の法則
  - システムのエントロピー(構造の無秩序性)は，その増大を防いだり，減少させたりするための意図的な努力がなされない限り，時間とともに増大する
- 統計的に滑らかな進化の法則
  - システム属性の進化は，局所的にはランダムな過程に見える
  - 統計的にみれば，規則的な変動と円滑な長期傾向とが合成されたものとなる

[1] L. A. Belady and M. M. Lehman, "A model of large program development, IBM Systems Journal", Vol.15, No.3, pp.225-252, 1976.
[2] M. M. Lehman and L. A. Belady, "Program Evolution - Processes of Software Change", Academic Press, 1985.

ソフトウェア保守の戦略

必要となる意思決定

- 保守を続けるか，現在のシステムを廃棄して新たに作り直すかの判断
- 個々の保守作業をどのような基準で行うかの判断
  - 簡便型
    - 当面の保守作業の負荷を最小化
    - 長期的にはソフトウェアの構造劣化により保守費用が加速度的に増大する可能性
  - 構造保全型
    - ソフトウェア構造をできる限り整ったかたちに保つ
    - 長命のソフトウェアでは後の保守費用を軽減する可能性

ソフトウェア保守・発展の開発工数

- 新規開発
  - 開発初期の要求分析・設計で徐々に増大
  - 実装・テストでピーク
  - 運用に進むにつれ減少

- 保守・発展
  - 要求分析・設計段階で大きな工数
  - 実装コストは新規開発に比べ，低い

ソフトウェアの保守と発展の技術

- プログラム理解
  - 多くの場合，他人が書いたプログラムを修正
  - プログラムを読んで修正すべき箇所を検討
- 変更波及解析
  - 変更がどこにどのように変更を及ぼすかを明らかにする
  - 仕様書レベルでの波及解析で利益とコストの比較
  - プログラムレベルでの波及解析でテスト設計
- リエンジニアリング
  - アーキテクチャレベルでの設計を変更後，目的の変更を行う
  - レガシーシステムの移行
- リバースエンジニアリング
  - 設計回復
  - リファクタリング

ソフトウェアの理解

- ソフトウェアを変更する際，その変更がプログラムの誤りであれば，プログラムだけを修正すれば良いが，要求仕様や設計の変更であれば，それらも修正する必要がある
- 実際にはプログラムだけが変更され，要求仕様書や設計書の変更は後回しにされがちで，変更されないままソフトウェアが進化
- ソフトウェアの理解には一番信頼できるプログラムの理解が不可欠

プログラム理解

- 静的プログラム解析を用いた手法
  - プログラムのロジック理解
  - プログラムの静的構造理解
- 動的プログラム解析を用いた手法
  - プログラムの動作理解

プログラムの視覚化

- 図式表現などの視覚的表現に変換する技術
- プログラムの内容をわかりやすい表現で提示

例：

- 統合開発環境などのエディタによるシンタックスハイライト (予約語の色付け)
- 関数の呼び出し関係のグラフ表現
- 規模や複雑度などのメトリクスに応じた構成要素間の関係図示
- オブジェクト間の振舞いのアニメーション

静的プログラム解析

- プログラムの構造をプログラムを実行することなく解析
- 識別子の定義・参照関係の解析
- 定義されている関数やクラス，メソッドの数
- データフロー解析・制御フロー解析
- 行数やサイクロマチック数の測定
- プログラムスライシング
- 動的に決定されるものは不確定な情報

プログラムスライシング

- プログラムスライス
  - プログラム中の特定の箇所で行われる計算の値に影響を与える文の集合
- プログラムスライシング
  - プログラムスライスの抽出
- フォーワードスライシング
  - 注目箇所から影響を前方向に解析
- バックワードスライシング
  - 注目箇所から影響を後方向に解析

バックワードスライシングの例

```
a = input();
b = input();
x = f0(a);
y = f1(a);
z = f2(b);
s = g(x, y);
t = h(y, z);
output(s);
output(t);
```

```
a = input();
b = input();
x = f0(a);
y = f1(a);
z = f2(b);
s = g(x, y);
t = h(y, z);
output(s);
output(t);
```

プログラムのロジック理解

- 統合開発環境
  - Eclipse, Visual Studio, Xcode
  - 定義参照関係の参照支援
- コードリーディング支援ツール
  - global, SPIE
  - 定義参照関係等，意味解析して得られた結果を HTML で閲覧可能

プログラムの静的構造理解

- クロスリファレンサ
  - cxref, lxr, global, SPIE
  - 識別子の定義位置・参照位置の一覧表示
- 関数の呼出し関係の視覚化
  - cflow + VCG
  - SXT
- メトリクスを用いたプログラムの視覚化
  - CodeForest
    - 構造・特性・依存を解析し,それらのメトリクス値からデータモデルを生成
    - データモデルを森林風景によるメタファーを利用し視覚化

動的プログラム解析

- プログラムを実行して解析
- テストデータを与えて実行し，実行中の情報をトレース情報として変数の値などを記録
- 静的解析よりも正確で詳細な情報を得られる
- 入力に依存した情報しか得られない

プログラムの動作理解

- 振舞いの可視化
  - プログラムの実行内容をシーケンス図等で可視化
- 統計的デバッギング
  - 成功した実行と，失敗した実行の差から，欠陥の原因となっているコードを自動的に特定
- 動的スライシング
  - 開発者が指定した変数の値に影響を与えたプログラム文を抽出
- Capture & Replay
  - あるオブジェクトの入出力をすべて記録し，オブジェクトの状態を再現

リバースエンジニアリング

- 要求・設計・プログラムはある時点では対応が取れている
- その時からのずれを追って対応関係の修復

リバースエンジニアリングの必要性

- プログラムから要求や設計を導き出す
- 要求仕様や設計は長い保守の間に
  プログラムとの差ができやすい
- あまり乖離してしまうと，プログラム理解が
  困難に
- 別の言語で書き換えるなど，仕様全体を
  把握する必要がある場合など

リバースエンジニアリングの問題点

- プログラムから設計情報の取得
  - プログラムには設計意図は書かれていない
  - コメントや識別子を基に設計モデルとの対応
- 設計モデルから要求仕様の対応
  - 要求仕様で使用される用語と設計モデルで使用される用語に統一がない
  - 対応が取れている設計モデルと要求仕様を基に辞書の作成

リストラクチャリング

- 変更が繰り返されると徐々に質が悪くなるので，それを改善するための技術
- 外部的な機能面での振舞いを変更することなく，その後の変更を行いやすい構造に変換
- 品質の劣化を防ぐ作業

リファクタリング

- プログラムのリストラクチャリング
- Fowler が提案したリファクタリング
  - メソッド抽出
    - プログラムの一部をメソッドとする
  - クラス抽出
    - 複数の役割を持ったクラスを役割ごとに分ける
  - 共通フィールドの上位クラスへの移動

  など
- 不吉な匂い：リファクタリングすべき状況
  - 重複したプログラム
  - 長過ぎるメソッド
  - 多過ぎる引数

  など

リエンジニアリング

- システムを新しい形態に再構築
- 必要な時
  - システムの稼動環境の大幅な変更
  - 従来のアーキテクチャで対応不能
- 再構築までのステップ
  1. リバースエンジニアリングで必要な抽象度の情報を取得
  2. フォーワードエンジニアリングにより新たなシステムを実装

ソフトウェアの再利用

- 他のソフトウェアの構成要素や構造を，ソフトウェア開発に利用
- 再利用資源
  - 再利用されるソフトウェアの構成要素や構造
- 再利用例
  - コピーアンドペースト
  - 標準ライブラリの利用
  - クラスライブラリの利用

ソフトウェアの再利用への期待

- 信頼性の向上
  - 新たに開発するよりも実績があり安定したソフトウェアを使う方が信頼性が高い
- リスクの軽減
  - 新規開発は不具合が入りやすいので，既存のものを使う方がリスクを軽減できる
- リソースの有効活用
  - 専門家の知識や労力の結果である既存資産を活用できる
- 標準への適合
  - 標準的な部品を使うことで標準への適合
- 開発効率の向上
  - すべてを開発することなく，再利用することで開発期間が短縮可能

ソフトウェアの再利用の問題点

- 保守コストの増大
  - 再利用資産のソースコードが利用できない場合，保守コストが増大
- ツール支援の欠如
  - 開発支援ツールでは新規開発の支援はするが，再利用資産の利用の支援が不十分
- NIH症候群
  - 自分で新たに作ることを好む開発者がいる
- 再利用資産の開発・保守
  - 再利用資産を広め，使わせ，維持することが高コスト
- 再利用資産の検索・理解・適用
  - 利用できる再利用資産を探し，それを理解し，開発中のソフトウェアに適用することは困難

プログラムライブラリ

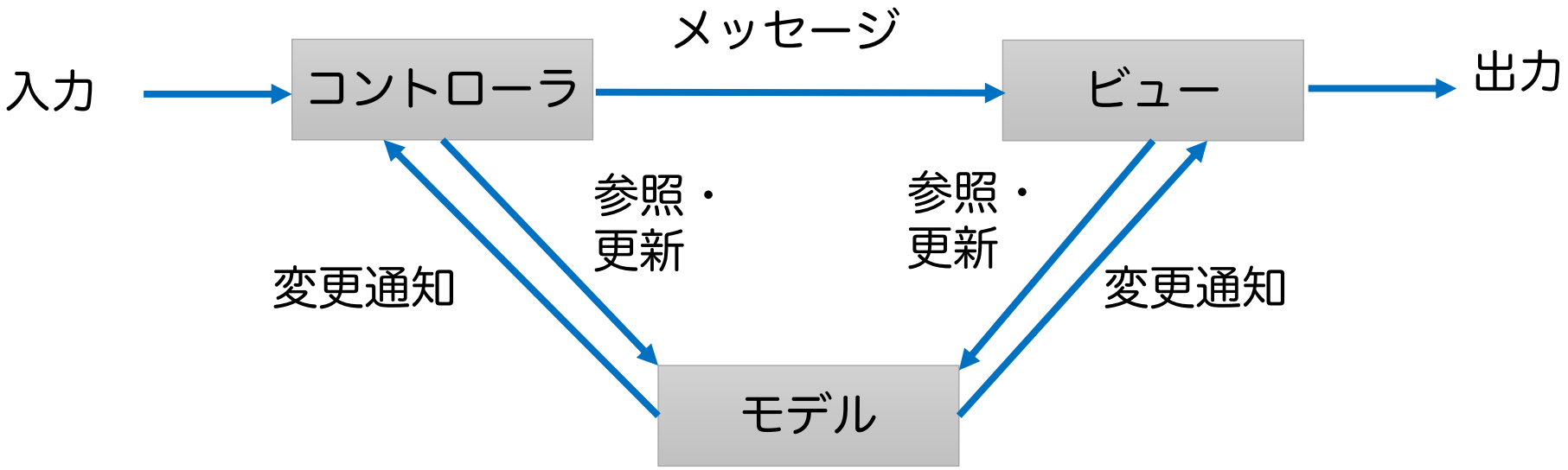
- 再利用されることを意図して用意されたプログラムの集合
- 入出力，数学計算，文字列計算など
- プログラムライブラリが異なると移植性が悪くなるため，標準的なものを利用することが望ましい
- C 言語では入出力，数学計算，文字列操作，日付計算などの関数群の仕様が規定

クラスライブラリ

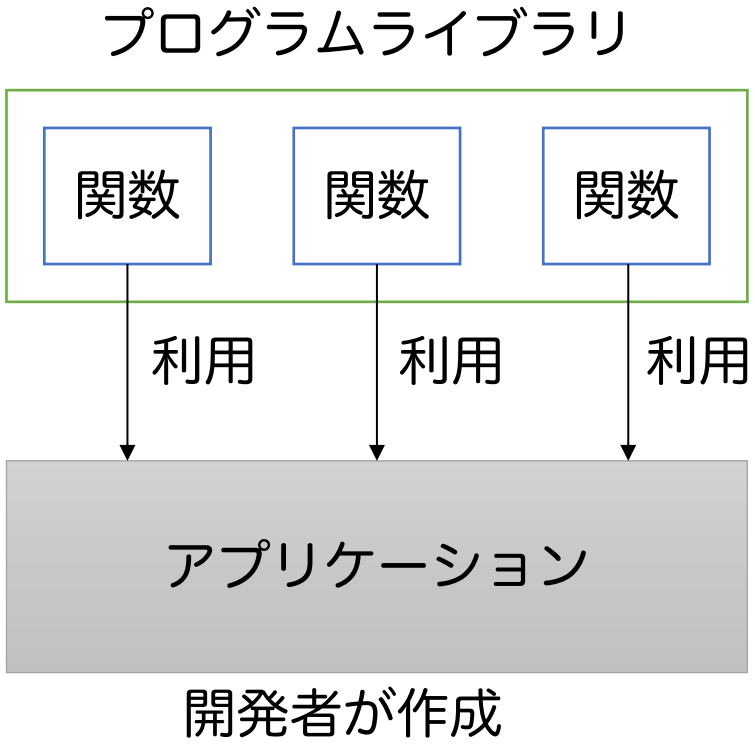
- オブジェクト指向言語向けのプログラムライブラリ
- 抽象クラスやインタフェースなどを利用し，適度に抽象化した再利用資産を提供可能
- C言語同様に C++ や Java など言語ごとに標準的なライブラリが提供されている

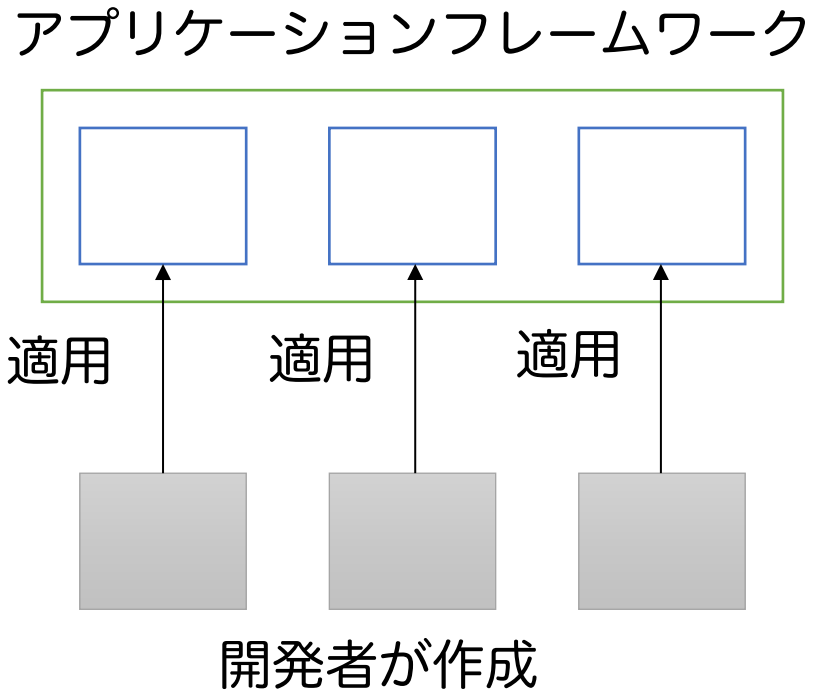
アプリケーションフレームワーク

- 具象クラス，抽象クラス，それらの間のインタフェースから構成されるシステムの設計
- 骨格となる構造があり，用意されたクラスを利用するなどしてアプリケーションを構築



プログラムライブラリとアプリケーションフレームワーク





COTS (Commercial-Off-The-Shelf)

- 市販の既製品ソフトウェア
- データベース，売上管理，顧客管理など
- カスタマイズして使用

コンポーネントベース開発

- コンポーネントを基本単位とする再利用開発
- クラスなどより大きい粒度のブラックボックス部品
- コンポーネントはネットワークにつながったマシン上に配置され，相互に通信できる
- コンポーネントモデルとして，コンポーネントの実現，文書化，配置などに関わる標準の定義
- CORBA，EJB，.NETモデルなど

サービス指向アーキテクチャ

- サービスを構成要素とした分散システムの開発方法
- サービス提供者はサービスを実現し，そのインタフェースを定義
- その定義はサービスレジストリに好評
- 利用者はサービスレジストリから必要なサービスを検索
- サービス記述言語：WSDL
- ビジネスプロセス記述言語：WS-BPEL
- サービスレジストリのための仕様：UDDI
- メッセージ交換のためのプロトコル：SOAP

RESTアーキテクチャ

- REST(REpresentational State Transfer)
  - Web サービスの設計モデル
- RESTful
  - アドレス可能性
    - URIによるリクエスト
  - ステートレス性
    - セッション等の状態管理はせず、やり取りされる情報はそれ自体で完結
  - キャッシュ可能
  - 統一インタフェース
    - リソースへの操作は CRUD のみ
  - 標準的なデータフォーマット
    - XML, JSON
  - 階層化システム
    - エンドポイントに直接アクセスするのではなく，中間サーバを経由

マイクロサービスアーキテクチャ

- ネットワークを通じてサービス間の連携
  - Web API を通じて連携
- 個々の小さな機能をサービスとして実現
  - 個々のサービスごとに開発可能⇒開発効率良
  - モノシリックなアーキテクチャと比べ，障害原因の突き止めが容易
  - 例：
    - 認証サービス，画像閲覧サービス，画像アップロードサービス

ソフトウェアパターン

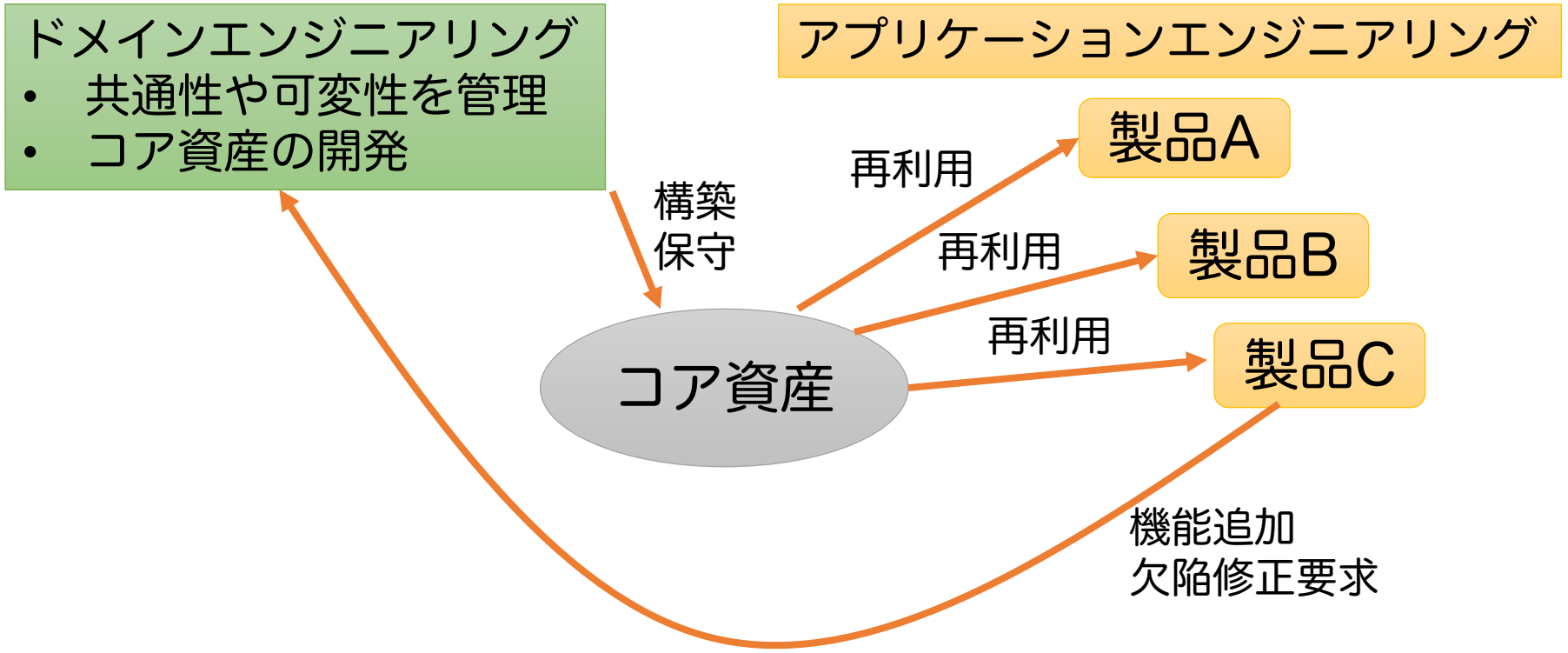
- よく使われるソフトウェアの構造
- 典型的な問題に対する解決策
- 分析パターン
- アーキテクチャパターン
- デザインパターン
- イディオム

アプリケーションジェネレータ

- 特定分野の問題を解くためのプログラムを作るコード生成機能
- 具体例
  - 文法を定義することで構文解析をするためのプログラムを生成
  - 画面を定義することでその画面によるユーザインタフェースを生成
  - ステートマシン図などでロジックを定義することでプログラムを生成

ソフトウェアプロダクトライン開発

- 製品ファミリを体系的な再利用によって効果的に開発するための再利用開発

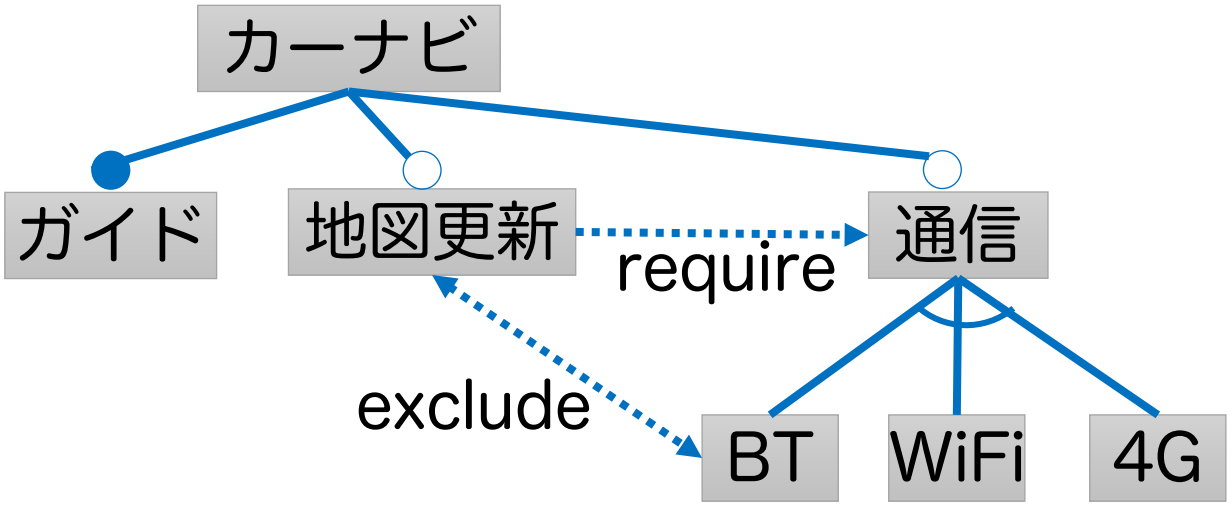


共通性と可変性

- 共通性
  - SPL 中のすべてのプロダクトに共通する機能的，非機能的な特徴
- 可変性
  - SPL 中のプロダクトによって変わりうる特徴
- 例
  - デジタルカメラ
    - 共通機能: 写真撮影，写真の削除
    - 非共通機能: WiFi 接続，写真編集

フィーチャモデル

- フィーチャ
  - システムの重要な側面，品質，特徴，機能
- フィーチャモデル
  - ソフトウェアプロダクトラインの持つフィーチャを階層的に記述するモデル



ソフトウェアプロダクトライン開発の課題

- アーキテクチャ不整合
  - 再利用資産開発者が想定しているアーキテクチャとそれを利用する側のアーキテクチャの違い
- 利用効果と汎用性のトレードオフ
  - 特定の分野向けの部品は利用効果が高いが，利用者は少ない
  - 汎用部品は利用者は多いが，利用効果が低い
- 再利用のエコシステム
  - 再利用資産提供者と利用者，その間でやりとりされる再利用資産，コスト，対価から構成される関係

モデル駆動開発

- ソフトウェアモデルとモデル変換技術を活用した開発
- ソフトウェアモデル
  - UMLモデル (ステートマシン図，クラス図)
  - ドメイン特化モデル
- ソフトウェアモデル上でシミュレーション
- ソフトウェアモデルからコード生成

モデル駆動開発の利点

- 設計レベルのモデルを作成
  - クラス図，シーケンス図，ステートマシン図等
- 設計段階で検証が可能
  - 機械可読なモデルを用いてシミュレーション
- モデルからソースコードの生成
  - 実装パターンがある程度あるので，そのパターンに従って変換
  - 様々な言語，バージョンに対応可能
- 実装と設計のずれが生じにくい

まとめ

- ソフトウェアの保守・発展
  - 修正保守，適応保守，改善保守，予防保守
  - 新たな環境への適合，進化
- ソフトウェアの保守・発展のための技術
  - プログラム理解，変更波及解析，リエンジニアリング，リバースエンジニアリング
- ソフトウェアの再構築
  - リバースエンジニアリング，リストラクチャリング，リファクタリング，リエンジニアリング
- ソフトウェアの再利用
  - ライブラリ，フレームワーク，コンポーネント，サービス，マイクロサービス

総まとめ

- ソフトウェアモジュール
  - 適切なモジュールとは?
    - 機能 (モジュール内)，独立性 (モジュール間)，大きさ，再利用性，保守性
- ソフトウェアテスト
  - 網羅性の基準 (制御構造に基づく基準，状態遷移モデルに基づく基準)
  - データの選定基準 (同値分割，ペア構成...)
- 形式手法
  - 形式手法とは? 利点や欠点は?
  - Hoare 論理による証明付きプログラム
- ソフトウェアメトリクス
  - 凝集度，結合度，McCabe のサイクロマチック数，...
- ソフトウェアの保守と発展
  - 静的・動的プログラム解析，再利用

試験

- 2024年1月29日
- 15:00-16:20 (試験時間 80分)
- 試験教室：総合研究7号館 情報1
- 持ち込み：
  - 教科書，講義資料，ノート可
  - PC，タブレット，スマホ等端末や通信機器は不可

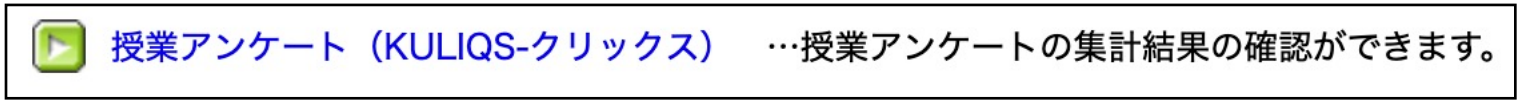
評価方法

- レポート
  - 配点 15
    (伊藤担当分 5, 星野担当分 5, 渥美担当分 5)
  - 伊藤担当分 1回, 星野担当分 1回,
    渥美担当分 1回

- 試験
  - 配点 85
    (伊藤担当分 35, 星野担当分 30, 渥美担当分 35) * 0.85
  - 対面での試験
  - 80分

授業アンケート

- 回答期間
  - 1/5(金) – 2/6(火)
- KULASISから
  - ホームの下記リンクをクリック



- 学生ポータルから
  - アンケートシステムのリンクをクリック
- 授業アンケート (KULIQS-クリックス)
  - https://enq.gakusei.kyoto-u.ac.jp/user