ソフトウェア工学

01: ソフトウェア工学とは

理工学部 経営システム工学科
庄司　裕子

今回のテーマ

- ✓授業のガイダンス
- ✓ソフトウェア工学とは – ソフトウェア工学の目的と歴史的背景

授業の概要

- 授業の目的:
  - ソフトウェア工学の歴史的背景と目的を理解する
  - ソフトウェア工学の主要技術の基礎を習得する
- 扱うトピック:
  - ソフトウェア開発プロセス
  - 従来の分析／設計手法
  - 近年のUMLモデリングと関連技術

前提知識

- オブジェクト指向プログラミング言語
- データベース工学の基礎

到達目標

- ソフトウェア工学の意義を理解する
- ウォーターフォール型開発の欠点と反復型開発の重要性を理解する
- 従来の主要な分析／設計手法の基礎を習得する
- UMLモデリングおよび関連技術の基礎を習得する

成績評価方法・基準

- 期末試験の点数によって評価する
- 合格（単位取得）には6割以上の正答を要する

教材

- 教科書
  - 特に指定しない。必要に応じてプリントを配布する
- 参考書
  - 有沢誠:「岩波コンピュータサイエンス ソフトウェア工学」
  - 日経ソフトウェア(編):「ゼロから学ぶソフトウェア設計」
  - Martin Fowler:「UMLモデリングのエッセンス–標準オブジェクトモデリング言語入門 第2版」(邦訳)
  - その他・・・(適宜紹介)

今回のテーマ

- ✓授業のガイダンス
- ソフトウェア工学とは – ソフトウェア工学の目的と歴史的背景

ソフトウェア工学の背景

- コンピュータの歴史
  - 第1世代コンピュータ(1940年代～60年):リレー式コンピュータ、真空管式コンピュータ(ENIACなど)
  - 第2世代コンピュータ(1960年代前半):トランジスタ
  - 第3世代コンピュータ(1960年代後半):IC(集積回路)
  - 第3.5世代コンピュータ(1970年代):LSI(大規模集積回路)
  - 第4世代コンピュータ(1980年～現代):VLSI(超高密度集積回路)

ソフトウェア危機とソフトウェア工学

- 第3～3.5世代にかけて(1960年代末～1970年代)
  - ハードウェアの急速な進歩により、コンピュータの重点がハードウェアからソフトウェアに移る
  - ソフトウェア開発の需要が急増したが、ソフトウェア開発技術が未成熟でソフトウェア技術者の絶対不足が危惧された(「ソフトウェア危機」)
- ソフトウェア危機の打開のため、ソフトウェアの系統的な開発方法論/技術/ツールの整備の必要性が認識された
  - ⇨ ソフトウェア工学の誕生(1970年代初頭)

その後のソフトウェア工学を取り巻く状況

- ハードウェア技術の進歩
- 通信/ネットワーク環境の進歩
- プログラミング言語/環境/ツールの進歩
  - 特に重要なのは、オブジェクト指向技術
- システム開発方法論の進展
  - さまざまなOOAD → UMLへ統一

しかし・・・

- システム開発そのものが画期的に容易になったとは言えない。むしろ、その逆の傾向も。

ソフトウェア V.S. ハードウェア

- ハードウェアはMooreの法則どおり倍々ゲームで進歩してきたが、ソフトウェアはそうではない。何が違うのか。
- ハードウェア
  - 所定のデバイスを適切に制御し、命令セットを正確に実行できれば、それでよい。何を実行すべきか(仕様)が明確に決まっている。モデル(構成要素とそれらの相互作用)が明確。複雑さはLSIの高度集積化に起因(物理的な問題)。
- ソフトウェア
  - 何を実行すべきかは千差万別。モデルはさまざま。複雑さは解くべき問題とその記述に内在(論理的な問題)。

ソフトウェア工学上の有名な事件

- 西暦2000年問題
  - 記憶装置が高価であった時代の設計（ベストプラクティス）によって引き起こされた大混乱
    - 日付の情報を「yyyymmdd」ではなく「yymmdd」で表現
  - 今ならとんでもない設計ミス
  - 教訓
    - どこまで将来を見据えて設計できるか
    - 設計で何を優先させるべきか（コストか信頼性か）
  - 同じようなトラブルが将来起きないという保証はない

今、ソフトウェア工学が目指すもの

- ソフトウェア/システムの以下の側面を向上させること
  - プロダクト　⇨ 品質
  - プロセス　⇨ 生産性
- 要は、「良いシステムを少ない工数で」開発できる方法論と技術の開発がテーマ

プロダクトの品質向上

- 開発するソフトウェア/システムの品質を向上させる
  - 提供するサービス/機能:
    - 性能（レスポンス、リソース消費などの観点から見た）
    - 信頼性（耐障害性、可用性、ロバスト性など）
    - 安全性（セキュリティ）
    - 管理性
    - 保守性
  - 変化への適応能力:
    - スケーラビリティ
    - 拡張性

品質要求の定義と検証環境

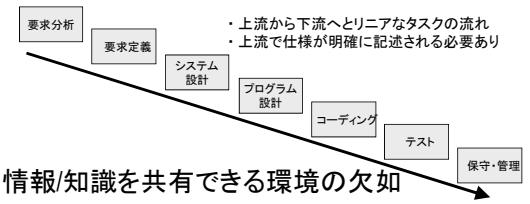
- 要求に品質条件を明記する
- 運用環境を再現するテスト環境を構築し、運用さながらのテストを行えるようにする。
  - 特に、昨今のエンタープライズクラスシステムの場合など
  - 完成したシステムを最後にまとめてテストするのではなく、要求からテストケースを生成するなどして、プロジェクトの初期から、実行可能なプロトタイプでテストを繰り返す

プロセスの生産性向上

- ソフトウェア/システムの開発プロセスにおける無駄を省く
- 起こり得る主な無駄:
  - クライアントの「見かけの」要求（requirements）に振り回されて、いつまでも「真の」要求を仕様（specifications）としてまとめることができない
  - プロジェクト終盤で根本的な欠陥が見つかり、大幅なやり直しを強いられる
  - 既存のアーティファクト（artifact）をわざわざ一から作り直す（inventing the wheel）

無駄の原因

- トップダウン方式の問題解決方法（ウォーターフォール モデル）は理解しやすいが、ほとんどの場合、現実的でない



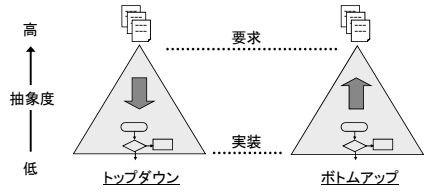
- 情報/知識を共有できる環境の欠如

無駄を省く決め手は？

- クライアントの「真の」要求を洞察できない
  - ⇨ 実行可能なプロトタイプを手早く作成（ラピッド プロトタイピング）して、クライアントとデベロッパの相互理解を深め、真のクライアント要求を引き出しやすくする
- プロジェクト終盤で根本的な欠陥が見つかり、大幅なやり直しを強いられる
  - ⇨ プロジェクト リスクの早期解消（要求とのミスマッチを小さいレベルで潰していく）
- 既存のアーティファクト（artifact）をわざわざ一から作り直す
  - ⇨ 既存資産の有効活用（再利用）

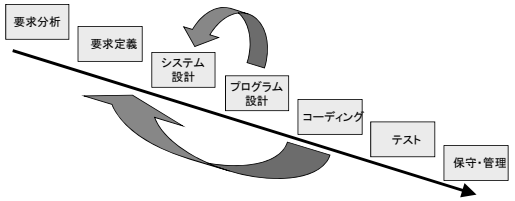
人間向きの望ましいアプローチ

- トップダウン方式とボトムアップ方式の融合
- 反復によって、不完全/あいまいなものから完全/明確なものへと近づける



反復型開発モデル

- 扱う問題の複雑さ（complexity）を段階的に緩和
  - ⇨ 多段の「分析→設計→実装→テスト」サイクルを繰り返し（フラクタル的）、それぞれのレベルでリスクを解消、次のレベルへ進む



問題の複雑さ（complexity）とリスク

- 現在入手可能な情報だけで顧客の（真の）要求をすべて記述ことは非現実的
- 仕様化されている要求が100%でなければ、開発中のシステムには常にリスクがつきまとう。
  - 完成してから顧客に「イメージと違う」の一言を言われないようにするにはどうするか
  - 一応完成したが性能条件をクリアできないというような事態を避けるにはどうするか

複雑さの取り扱い

- 「銀の弾丸」は存在しない（No silver bullets）
  - 何にでもそれ1つで解決できるような方法論を人間は欲しがるが、それは少なくとも現時点では存在しないし、おそらく将来も出てこないであろう。
  - ⇨ 利用可能な方法論を組み合わせる複合戦略が必要
- 完全さの追求からの脱却
  - 最初に問題を完全に記述してからそれを解決するというアプローチは、少なくとも大規模システムの場合は本質的に不可能
  - ⇨ 実行可能なプロトタイプの作成をマイルストーンにした反復型開発で、リスクを早期に解消しながら要求とのミスマッチをゼロにするアプローチが最も有望

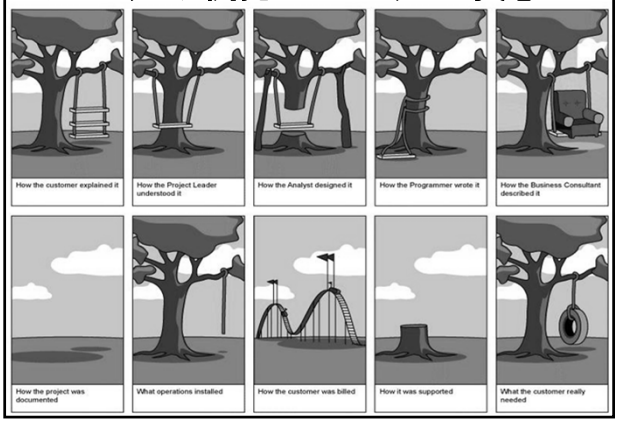
技術の進歩は複雑さを軽減するか？

- 技術が進歩すれば、それでできることが増えるので顧客の要件のレベルが上がり（あるいは新たなニーズが生まれ）、結局、解決すべき問題の複雑さが増大する場合がある
  - ⇨ 技術が進歩すれば開発すべきシステムの複雑さが増すという皮肉
- （例） セキュリティ
  - ⇨ ネットワーク（特にインターネット）がこれほど普及しなければ、セキュリティ問題がこれほど深刻化することはおそらくなかったであろう。

ソフトウェア工学のヒューマン ファクタ

- システム開発は純粋な技術論では済まず、さまざまな関係者の利害が複雑に絡み合うことが多い
  - ⇨ソーシャル スキルが物を言う。
- 特に、コミュニケーションが難しく、また重要

ソフトウェア開発プロジェクトの実態



理想的なプロジェクト管理環境

- 要求管理ツール（要求と成果物のリンク）
- 変更管理ツール（成果物の一貫性を保つ）
- 構成管理ツール（種々のバージョンを管理）
- 統合開発環境（IDE）
- テスト ツール
- コミュニケーション支援ツール
  - メーリング リスト
  - ディスカッション グループ

統合開発環境（IDE）の主な機能

- UML ビジュアル モデリング
- フォワード/リバース エンジニアリング
- 入力支援機能付きプログラミング エディタ
- デバッガ
- ソースコード検査＆メトリクス測定
- リファクタリング
- 単体テスト
- ドキュメンテーション

まとめ：ソフトウェア開発の問題空間

- ソフトウェア開発の問題空間は閉じていない
- 常に新たな問題が生まれ、複雑さが増す。
- δ＝（問題の新たな複雑さ）－（新たな技術）
  - おそらく、本質的にδ ≧ 0
  - また、そうでなければ、ITビジネスが成立しない
- ソフトウェア工学とは、δを 0 に近づける取り組みと言える