ソフトウェア工学とは？

内容

- 歴史・背景
- 定義・重要性
- 事例

歴史・背景

ソフトウェア工学誕生までの歴史

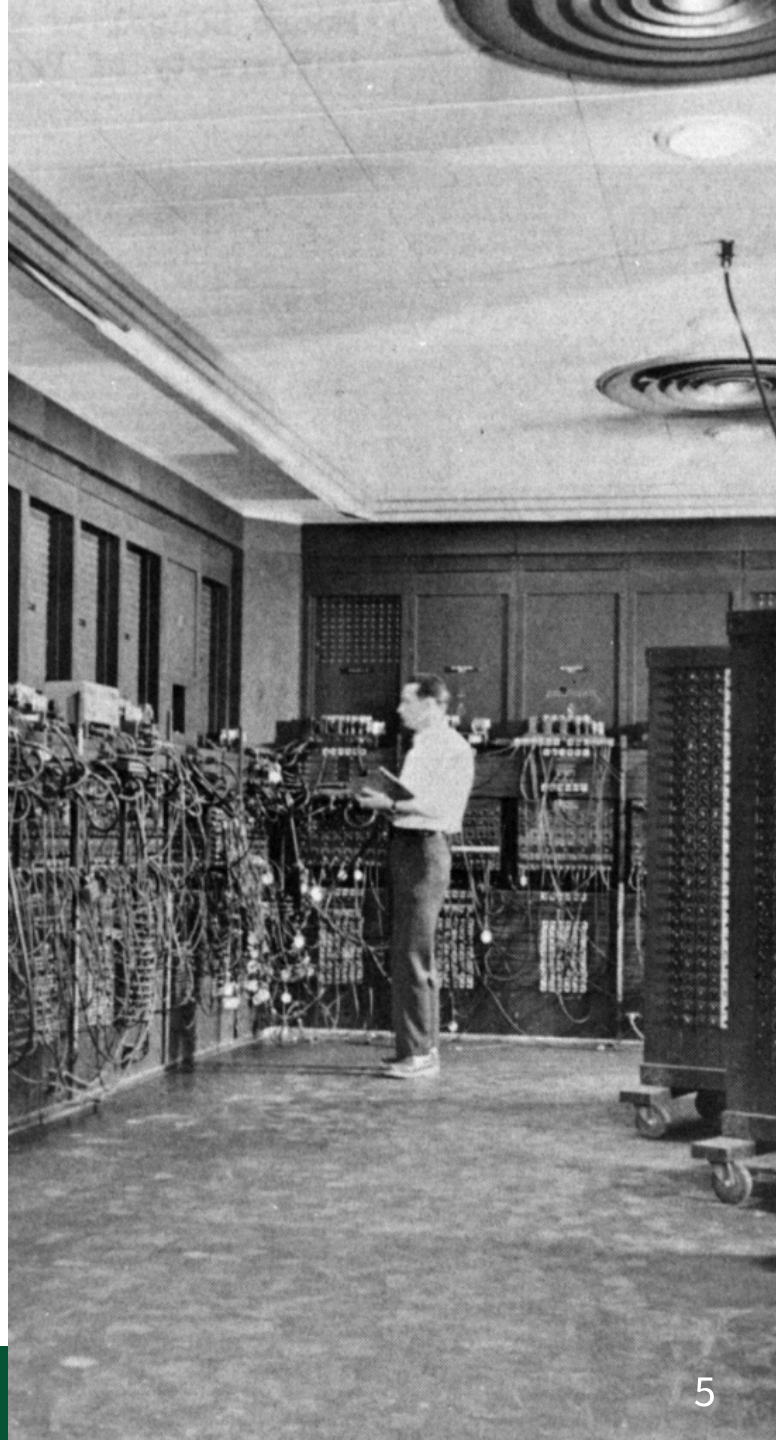
- 1942〜46 コンピュータの軍事利用
  （暗号解読，弾道計算，リレーによるプログラミング）
- 1949 プログラム内蔵方式（EDSAC）
- 1953〜54 基本ソフトウェア
- 1955 FORTRAN
- 1960 ALGOL, COBOL, LISP
- 1964 IBM System/360 (OS/360)
- 1968 NATO の国際会議「ソフトウェア工学」

Contd.

- **1942〜46 コンピュータの軍事利用**
  （暗号解読，弾道計算，リレーによるプログラミング）

特徴

- 手作業による配線やパンチカードで命令を入力
  → ハードウェアとプログラムが一体
  → 再利用・変更が困難、設計変更に数日〜数週間
- 特定用途に特化した専用マシン
  → 弾道表作成、暗号解読、水爆開発計算などに活用
- 巨大な装置と消費電力
  → ENIAC：30トン／17000本の真空管／常時空調が必要



Contd.

- **1949 プログラム内蔵方式（EDSAC）**

特徴

- ENIACまで：プログラムはハードウェア（配線）に書き込み→ 処理内容の変更に数日単位の再配線作業が必要
- プログラムもデータと同じく**メモリに格納**
  → **プログラム内蔵方式（Stored-Program Concept）**
- EDSAC（Electronic Delay Storage Automatic Calculator）
  - ケンブリッジ大学（モーリス・ウィルクス）により開発
  - 世界初の実用的プログラム内蔵方式（1949年5月）

Contd.

- **1953〜54 基本ソフトウェア**

特徴

- EDSACのようなプログラム内蔵方式マシンが登場し，複数の処理をソフトで制御可能
- 利用者が直接マシン語を書くのは負担が大きい→ **作業を支援する基本ソフトウェア**
- 代表的な基本ソフトウェア
  - アセンブラ：ニーモニック（例：ADD, JMP）で機械語を記述
  - ローダ：複数の機械語プログラムを読み込み配置
  - リンクエディタ：複数の部品を一つにまとめる
  - モニタプログラム：実行順序やエラー処理を管理
- 「プログラムを書く人」と「機械を管理する仕組み」が分離

Contd.

- **1955 FORTRAN**

特徴

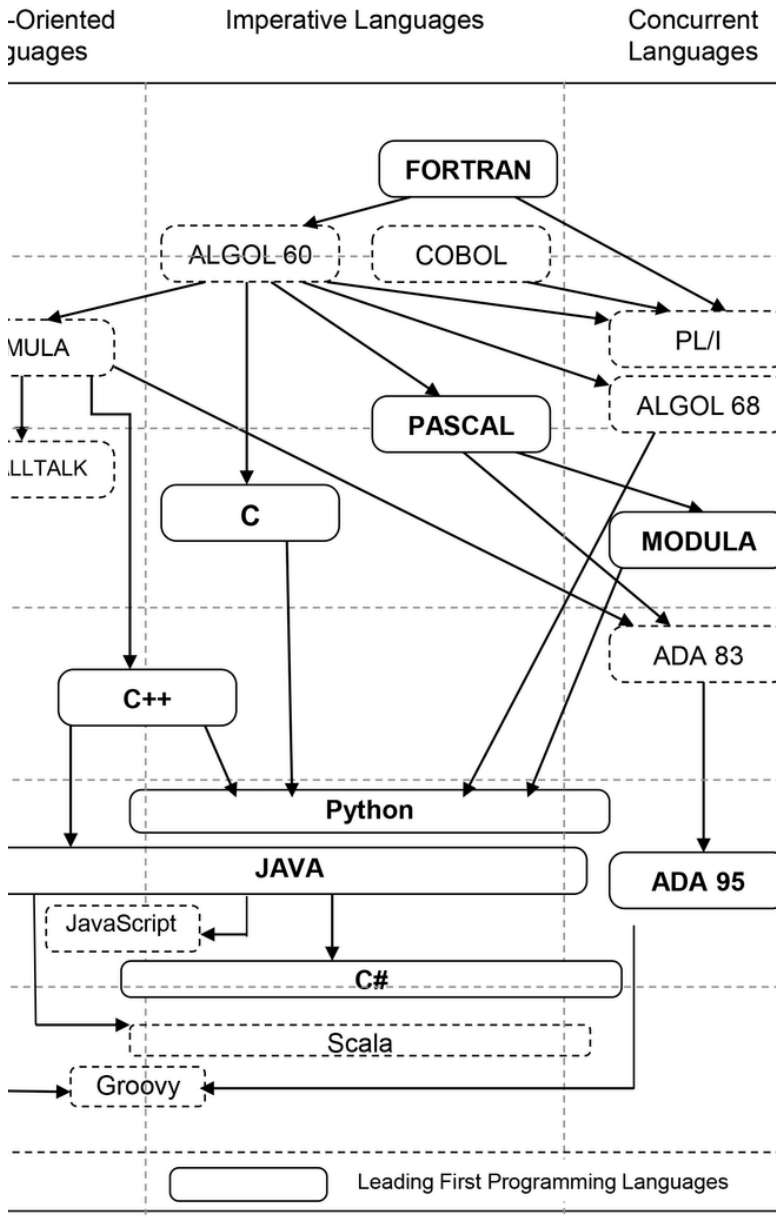
- 科学技術計算に使いたいが，機械語では効率が悪く人間に優しくない
- IBMのジョン・バッカスらが開発（1954年〜1957年リリース）
- 世界初の高水準プログラミング言語
- IF, DO, GOTOなどの制御構文を導入
- 数式を数学記法のまま書けるように
- コンパイラ技術も同時に進化（機械語への自動変換）

Contd.

- **1960 ALGOL, COBOL, LISP**

特徴

- 高級言語の発展
  - ALGOL（Algorithmic Language）：アルゴリズム記述に特化、ブロック構造を導入
  - COBOL（Common Business Oriented Language）：ビジネス用途向け、データ処理に強い
  - LISP（LISt Processing）：人工知能研究用、リスト操作に特化した言語
- **利用目的に適した言語**の開発

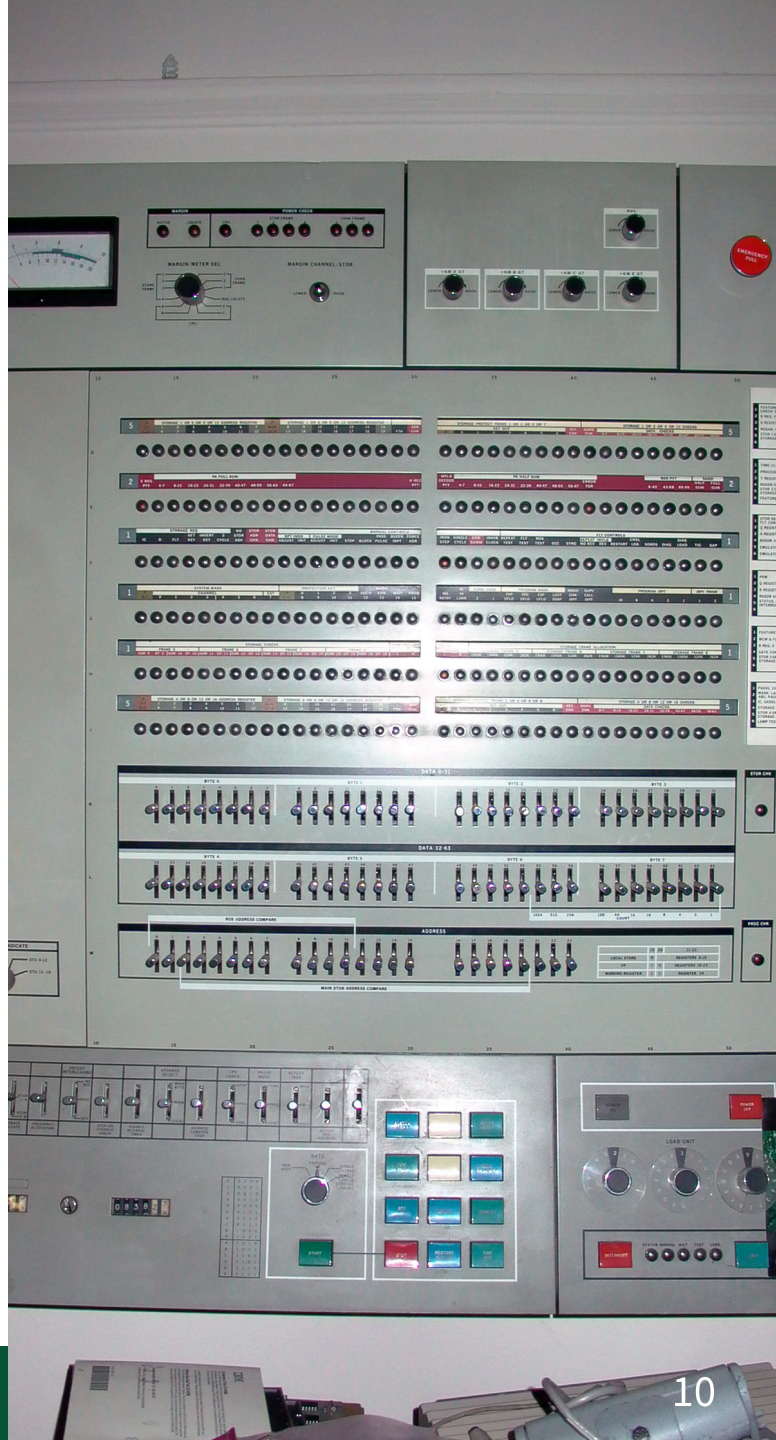


Contd.

- **1964 IBM System/360 (OS/360)**

特徴

- IBMが発表した汎用コンピュータ（System/360）→ 当時、業務・科学・制御計算が別々の専用機で実行されていた状況を統一
- ハードウェアの汎用化
  - 異なる機種間で命令セットレベルの互換性を実現
- ソフトウェアの汎用化
  - OS/360により1つのOSで複数用途をカバー
  - ジョブ管理、メモリ管理、I/O制御、プログラム実行制御などを統合



- IBM史上最大規模のソフトウェア開発プロジェクト
  - 約1,000人以上が関与、スケジュール・予算の大幅超過
  - 複雑な機能・多様な構成に対応する必要があり、開発難易度が非常に高かった

項目	内容
開発期間	1964年～1971年（約7年間）
開発規模	約1,000万行のコード
開発人数	最大1,000人以上の技術者が関与
遅延・超過	当初のスケジュールを大幅にオーバー、予算も倍増以上
技術的課題	並列作業の調整不足、仕様の変化、バグ修正の遅れ
社会的影響	プロジェクトマネジメントとソフトウェア設計の重要性が再認識される

→ **OS/360の開発は，後のソフトウェア工学の必要性と限界を可視化した事例**

Contd.

- **1968 NATO の国際会議「ソフトウェア工学」**

特徴

- NATO（北大西洋条約機構）が主催した国際会議
- 開催地：ドイツ・ガルミッシュ・パルテンキルヒェン
- 目的：ソフトウェア開発の現状と課題を議論し解決策を模索
- **「ソフトウェア工学（Software Engineering）」という言葉が初めて提唱された**
  - 体系的・構造的な開発手法の重要性が強調された
  - ソフトウェア開発における「プロセス」という概念
  - モジュール化/構造化、テスト、品質保証など

まとめ

- ソフトウェア工学の誕生は、1960年代のコンピュータ技術の進化と社会的なニーズの高まりによる
- **OS/360の開発**がソフトウェア工学の必要性を世に知らしめた
- ソフトウェア開発の難しさを可視化し、体系的なアプローチの必要性を認識

定義・重要性

ソフトウェア工学とは？

- ソフトウェア工学（Software Engineering）とは？
  - ソフトウェアを計画的に開発・運用するための体系的なアプローチ
  - 要件定義・設計・実装・テスト・保守までを一貫して扱う
  - 経験や勘ではなく、**再現可能で管理可能な方法**に基づく
  - 目標：**高品質**なソフトウェアを、効率的かつ安全に開発すること

ソフトウェア工学の定義（IEEE Std 610-1990）

> ソフトウェアの開発，運用，保守に対する，系統的で統制され定量化可能な方法．すなわちソフトウェアへの工学の適用．また，上記のような方法の研究．

- 補足
  - 「統制された方法」とは、仕様・設計・変更履歴などが記録されていて追跡可能であること
  - 「定量化可能」とは、バグ数や開発コストを測れること

なにが難しいのか？

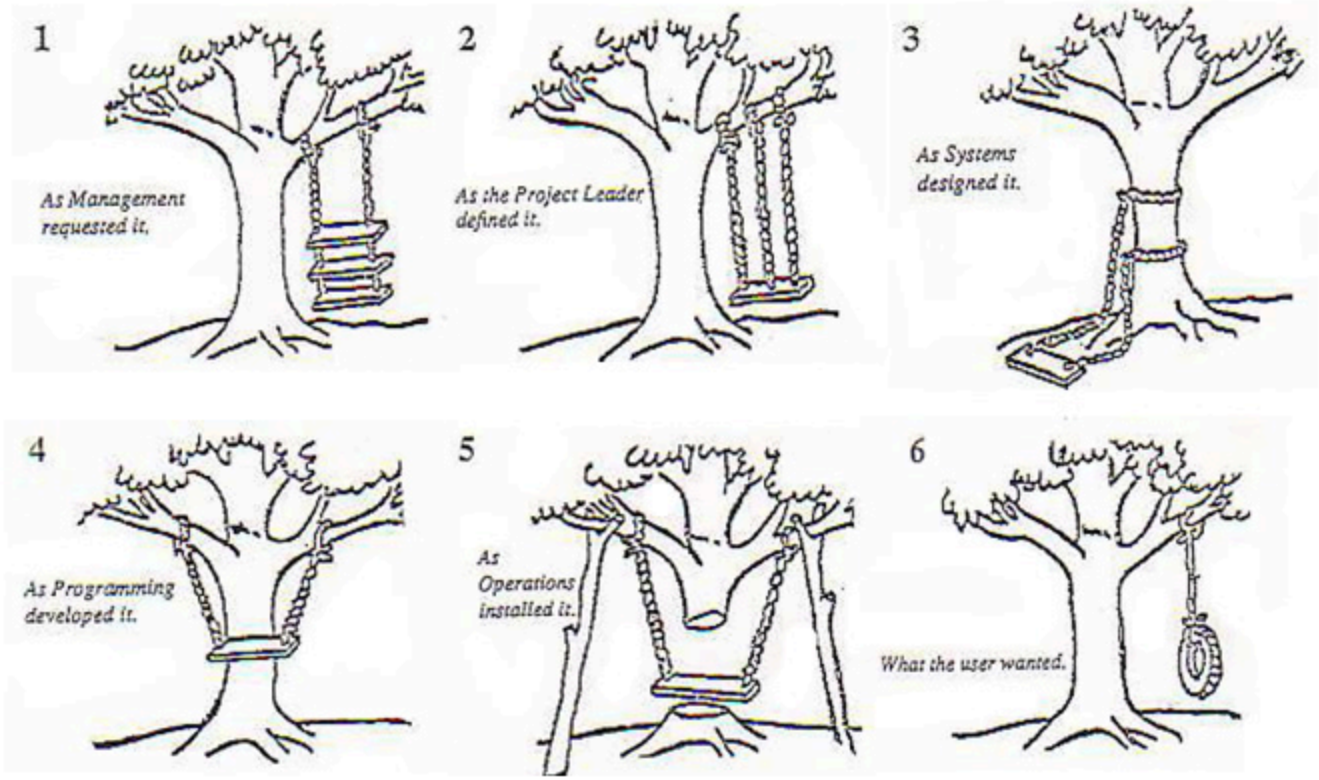
- Frederick P. Brooks の分析（IBM OS/360 の開発者）
- OS/360 開発プロジェクトの失敗を経て分析
  → 結論：規模の大きなソフトウェアを作るのは難しい
  - 目に見えないものは管理できない（完成度が見えない）
  - 変化させやすいものは管理が難しい（「これで最後」がない）
  - 絶対的な制約のないものは管理が難しい（物理制約がない）

建設プロジェクトと対比

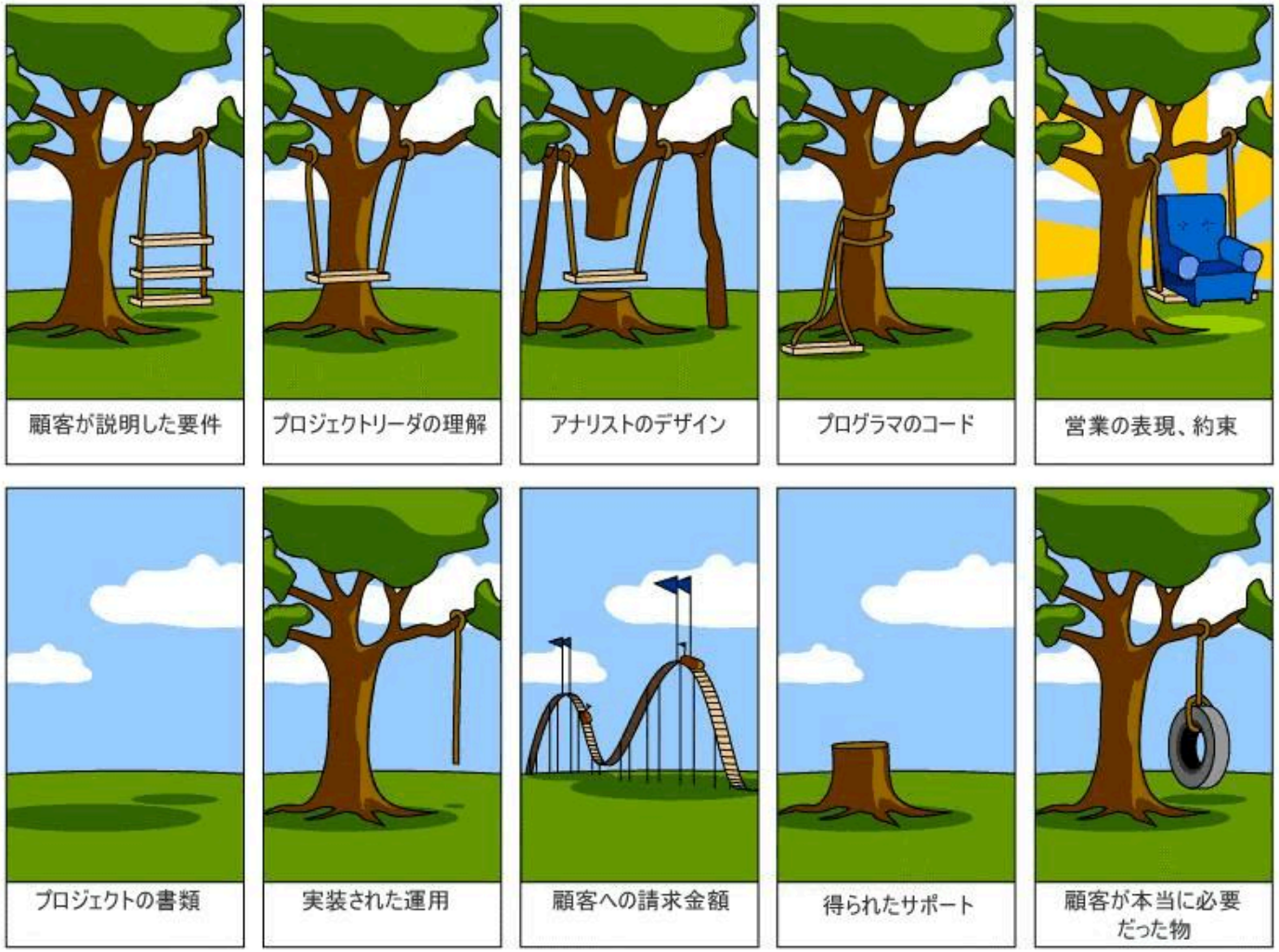
特徴	建築・土木工学	ソフトウェア工学
目に見えるか	基礎，骨組み，屋根などが目に見える	完成度が見えにくい（抽象的）
変更のしやすさ	着工後の変更は容易でない	仕様が揺れやすい（容易に変更できてしまう）
物理的制約	材料・重力・法律などの制約あり	制約が少なく設計自由度が高い
成熟度	数千年の歴史と標準化された手法	まだ歴史が浅く、標準が流動的
再利用性	モジュール化・設計パターンが進んでいる	再利用の体系化が進行中
検証のしやすさ	構造計算・試験施工などで事前に評価可能	実装後の動作確認が必要（テスト工程が必須）

ブランコの例

University of London Computer Center Newsletter, No.53, March 1973 (Pre-1970 cartoon; origin unknown)



ブランコの例（別バージョン）



ソフトウェア危機 vs ソフトウェア工学

観点	ソフトウェア危機（1960年代）	ソフトウェア工学
開発の進め方	経験と勘に依存	プロセスに基づく計画と管理
設計手法	行き当たりばったり（コーディング先行）	要件→設計→実装→テストの順に構造化
品質管理	テストは最後に付け足し	テストを開発の中心に据える（例：TDD）
変更対応	文書がなく追跡困難	バージョン管理・変更管理を活用
チーム開発	個人依存、役割分担なし	分業と責任分離（設計・実装・テスト）
結果	バグ多発、納期遅延、予算超過	再利用・標準化・自動化で安定性向上

ソフトウェア工学とは「偶然の成功を再現可能な成功に変える技術体系」

ソフトウェア工学の重要性

- なぜ学ぶのか？
  - ソフトウェアは現代社会の基盤：スマホ、AI、自動運転、インフラ制御など
  - 一つのバグが社会全体に影響を与えることもある（例：バグによる事故・損失）
- 身近な問題：
  - アプリが突然落ちる、セキュリティ事故、更新で不具合発生など
- ソフトウェア工学の役割：
  - こうした問題を**未然に防ぐ**ための方法論を学ぶ

ソフトウェア工学の知識体系（Body of Knowledge）

- 知識体系（Body of Knowledge）とは：
  - ある専門分野において、「体系的に整理された重要な知識の全体像」
  - 単なる用語集やトピック集ではなく、標準的な教育・実務の基盤となるもの
- ソフトウェア工学では、IEEE が定めた SWEBOK（Software Engineering Body of Knowledge） が代表的
  - 世界中の大学教育、資格試験、実務ガイドラインのベースになっている
- BoKを学ぶ目的：
  - 「何を学べばよいか」の全体像を把握できる
  - 専門職としての一貫性と再現性を持った実践が可能になる

→ BoKは、単に“知っている”から“使いこなせる”技術者になるための学びの地図

ソフトウェア工学は広範な分野であり、IEEE SWEBOK v3.0 では以下の13の知識エリア（Knowledge Area）が定義されている：

1. 要件工学（Software Requirements）
2. ソフトウェア設計（Software Design）
3. ソフトウェア構築（Software Construction）
4. ソフトウェアテスト（Software Testing）
5. ソフトウェア保守（Software Maintenance）
6. 構成管理（Software Configuration Management）

7. ソフトウェア工学マネジメント（Software Engineering Management）
8. ソフトウェアプロセス（Software Engineering Process）
9. モデリングと手法（Software Engineering Models and Methods）
10. ソフトウェア品質（Software Quality）
11. 専門的実践（Software Engineering Professional Practice）
12. ソフトウェア経済学（Software Engineering Economics）
13. 基礎（Foundations）：計算理論・数学・工学の基礎知識

要求工学

- ソフトウェアが満たすべき要件を明確に定義する
- ユーザーのニーズを正確に把握し、文書化する
- 要件の優先順位付けとトレーサビリティを確保
- 例：ユーザーインタビュー、ユースケース、要求仕様書の作成

ソフトウェア設計

- ソフトウェアのアーキテクチャと構造を決定する
- モジュール化、インターフェース設計、データ構造の選定
- 設計パターンや原則（SOLIDなど）を活用

ソフトウェア構築

- 実際のコードを書くプロセス
- コーディング規約の遵守、コードレビューの実施
- バージョン管理システム（Gitなど）を活用してコードの履歴を管理

ソフトウェアテスト

- ソフトウェアの品質を保証するためのテスト手法
- ユニットテスト、統合テスト、システムテスト、受け入れテストなど
- テスト自動化の重要性（CI/CDパイプラインの活用）

ソフトウェア保守

- ソフトウェアの運用中に発生する問題の修正や機能追加
- バグ修正、機能改善、セキュリティパッチの適用
- 保守性を考慮した設計（コードの可読性、ドキュメント化）

構成管理

- ソフトウェアのバージョン管理と変更管理
- ソフトウェアの構成要素（コード、ドキュメント、設定ファイルなど）の管理
- 変更履歴の追跡、リリース管理、ビルド管理

AI時代の視点：価値の重心が移る

- 生成AI・AIコーディングエージェントの普及で、**「コードを書く」作業はAIが担う**ようになりつつある
- すると人間の価値は **書く力** から、開発工程全体を **使いこなす力** へ移る：
  - **要件定義・設計**：何を作るかを正確に定義する（あいまいだとAIは間違える）
  - **レビュー・テスト**：AIの出力を読み、検証し、欠陥を見抜く
  - **構成管理**：誰が（人かAIか）書いたコードも履歴として管理する
- ＝ 本講義で学ぶ各工程の知識は、**AIを正しく使うための土台**になる

> AIが使える人も使えない人もいる。まずは「何ができる道具か」を理解しよう

ソフトウェア工学マネジメント

- プロジェクトの計画、進捗管理、リスク管理
- チームの組織化、リソースの割り当て、コミュニケーションの促進
- プロジェクトマネジメント手法（アジャイル、ウォーターフォールなど）の適用

ソフトウェアプロセス

- ソフトウェア開発のプロセスモデル（アジャイル、ウォーターフォールなど）
- プロセスの改善と最適化
- プロセスの標準化と適用（CMMI、ISO/IEC 12207など）

モデリングと手法

- ソフトウェアの抽象化とモデリング手法
- UML（統一モデリング言語）やER図などの使用
- モデル駆動開発（MDD）やドメイン駆動設計（DDD）の活用

ソフトウェア品質

- ソフトウェアの品質特性（機能性、信頼性、使用性、効率性、保守性など）
- 品質保証の手法（レビュー、テスト、静的解析など）
- 品質管理の指標（バグ密度、コードカバレッジなど）

専門的実践

- ソフトウェアエンジニアとしての倫理と専門性
- 継続的な学習と専門知識の更新
- ソフトウェアエンジニアリングの職業倫理（ACM/IEEE Code of Ethicsなど）

ソフトウェア経済学

- ソフトウェア開発のコストと利益の分析
- 投資対効果（ROI）の評価
- ソフトウェアのライフサイクルコスト（LCC）の理解

基礎（Foundations）

- ソフトウェア工学の基礎理論（計算理論、離散数学、アルゴリズムなど）
- ソフトウェア工学の数学的基礎（形式手法、論理、集合論など）

ソリューション（今のところ）

- 目に見えないものは管理できない
  → 「動作」が見えるようにする（継続的インテグレーション; Continuous Integration）
- 変化させやすいものは管理が難しい
  → 変化を許容する（リファクタリング; Refactoring）
- 絶対的な制約のないものは管理が難しい
  → 最低限の制約を与える（テスト駆動; Test-driven development）

まとめ

- ソフトウェア工学は、ソフトウェア開発の体系的なアプローチ
- ソフトウェア開発の難しさを理解し、適切な手法を学ぶことが重要
- ソフトウェア工学の知識体系（BoK）を学ぶことで、専門職としての一貫性と再現性を持った実践が可能になる

事例

ソフトウェアの失敗事例

2011年 みずほ銀行 システム障害

- 背景：東日本大震災（2011年3月11日）の義援金取引が急増
- 原因：
  - 義援金振込の処理が集中し、バッチ処理が遅延
  - メインシステムとバッチ処理スケジューラ間の連携ミス
  - オペレータのマニュアル操作失敗も重なり、障害が連鎖
- 結果：
  - 3月15日〜22日にかけて5日連続の大規模トラブル
  - 約80万件の振込に影響、ATMやオンラインバンキングも停止
- 教訓：
  - 例外的な状況に備えた設計・負荷対策の欠如
  - 人とシステムの連携設計（ヒューマンインタフェース）の重要性
  - マルチベンダー開発による責任分界の不明確さ

2020年 東京証券取引所 売買全面停止

- 背景：国内最大の証券取引所である東証の全銘柄売買が停止
- 原因：
  - 共有ディスク装置に障害 → 冗長化システムが自動切替に失敗
  - 切替失敗後も手動切替の手順が正しく実行されず
- 結果：
  - 全銘柄の終日取引停止（史上初）、経済への深刻な影響
  - 金融庁から厳重注意、東証社長辞任へ
- 教訓：
  - フェイルオーバー機構の設計とテストの重要性
  - マニュアル運用手順の明確化と訓練
  - 金融インフラに求められる可用性と透明性

トヨタの電子制御問題と大量リコール（2010年前後）

- 背景：2009〜2010年にかけて、米国を中心に「急加速問題」の苦情が多数寄せられた
  → 2009年には死亡事故も発生し、米国議会での公聴会へ発展
- 原因（複合的）：
  - フロアマットによるペダルの引っかかり，アクセルペダルの物理的摩耗
  - 一部はソフトウェア制御系の誤動作も疑われた（が証明は不十分）
- 対応と結果：
  - 世界中で900万台以上のリコール
  - ソフトウェアに対する規制・標準化の動きが活発に（例：ISO 26262）
- 教訓：
  - ハードとソフトが密接に連携する組込みシステムにおけるソフトウェア設計の責任
  - 異常系（フェールセーフ、フォールトトレラント）設計の重要性
  - 不具合の特定・再現が困難な場合における検証プロセスの整備の必要性

なぜ失敗するのか？

原因の種類	例	工学的対策
要件定義の不備	ユーザーの期待とずれた機能	ユースケース記述、ステークホルダー分析
設計ミス	データ型や並行処理の誤り	設計レビュー、モジュール分割
テスト不足	境界条件・例外処理の漏れ	単体・結合・システムテスト、自動化
運用ミス	デプロイ設定ミス、オペレータの誤操作	手順書、監視・ログ設計、教育訓練
変更管理不足	意図しないバグの混入	バージョン管理、CI/CD、コードレビュー

ソフトウェア工学は、こうした問題を未然に防ぐ“体系的な対策”の集合体

失敗事例

とある大学で教員の活動をデータベース化するシステムを導入することになった．

- 学部
  - 教養部，文学部，法学部，経済学部，理学部，医学部，歯学部，薬学部，工学部，農学部，情報学部
- 教員数：1,700名

事前の教員会議

教員A：
「教員の活動だけど、違う学部の教員同士でも活動状況をうまく比較できるようしたいよね」

教員B：
「そうだよね。違う学部で教員の活動状況がわかると色々便利そうだし、学部間の連携も進むかも」

教員A：
「どうせ電子データで保存しておくんだから、使うかどうかは別として、入れられるデータは全部入れておけばいいんじゃない？」

教員B, C, D：
「それはいいアイデアだね！データがあれば、後から分析もできるし、将来の参考にもなるよね」

システム設計ミーティング

教員A：
「教員の活動をデータベース化するシステムを作ることになったけど、各学部でそれぞれこんな感じをデータを入れたいと思うんだ」

学部	データ項目
教養部	学生数、授業数、研究費、社会貢献活動
文学部	学生数、授業数、研究費、社会貢献活動、論文数
医学部	学生数、授業数、研究費、社会貢献活動、論文数、臨床実習時間

etc...

システム設計者A：
「各学部の要件を聞いてデータベースの設計を考えたのですが、将来的にいろんなデータが入りそうですよね？それなら学部に関係なく共通のデータ項目にしておいたらどうでしょう？」

システム設計者B
「医学部の臨床実習時間とか、文学部の論文数とか、全部同じテーブルに入れちゃう感じで。設計の工数も減るのでコストも抑えられますし」

教員A：
「なるほど、確かにそうすればデータベースの設計もシンプルになりますね。じゃあ、共通のテーブルに入れる形で進めましょう」

システム設計者A：
「入力方法はどうします？Webから入力できるようにします？」

教員A：
「Webって便利そうにだけど、操作覚えるのが大変そうですね。一年に一回だけの入力だし、年配の教員は使い慣れていない人も多いから・・操作の問い合わせが増えそうです」

システム設計者B：
「じゃあ、Excelで入力してもらってファイルをアップロードする形にしましょうか？そうすれば、各学部の教員も慣れたツールで入力できますし」

教員A：
「それはいいアイデアですね！Excelなら使い慣れている人も多いですし、アップロードだけなら操作も簡単そうです」

教員B：
「でも、Excelで入力するとなると、各学部のデータ項目が違うから、どの項目を入力すればいいか分からなくなりませんか？」

システム設計者A：
「それなら、これまでの情報に追加する形にしたらどうですか？既に入力している内容をいったん全部 Excel ファイルでエクスポートして、それに追記すると入力しやすくないですか？」

システム設計者B：
「それなら、Excelをアップロードするときにデータの差分じゃなくて、全部置き換えるように作ればどうですか？ロジック簡単だからバグも減りますよ」

教員A：
「なるほど、それは良さそうですね。じゃあ、その方向で進めましょう」

システム開発後のテスト

システム設計者A：
「システムの開発が完了しました。テストを行いますので、データを入力してみてください」

教員A：
「わかりました。まずは自分の学部のデータを入力してみますね。動きました」

本番運用後

システム管理者：
「アップロード集中でサーバがダウンしました…！」

教員A：
「なんで！？ 試験のときはうまくいったのに！」

開発者A,B：
「ファイルサイズが10MB超 ×1,700人 → 負荷が想定外…負荷試験、足りなかったかも」

教員A：
「仕方ない、じゃあ学部毎に入力できる期間を決めよう」

Question: どこに問題があった？

- 誰がどんな問題を起こしているのか？
- 技術的な問題と、コミュニケーションの問題は？
- みなさんならどこで軌道修正して失敗を回避しますか？

まとめ

- ソフトウェア開発の失敗は、**単なる技術的ミスだけではない**
  - 要件の曖昧さ、コミュニケーションの欠如、目的の見失いなどが根本原因に
- 「使えそうだから全部入れる」「慣れてるからExcelで」など、**判断基準が利用者視点になっていない**ことが失敗を招く
- テストや設計が形式的でなく、**実運用を想定していたか？**
- 誰がどのように使うか（**ユースケース**）を中心に設計することが不可欠

成功への視点

観点	重要な問い
目的	このシステムは何のため？ 誰が得をする？
ユーザー	誰が使う？ 使いやすさは？ 教育は必要？
要件	本当に必要な機能はどれ？ なくてもよいものは？
設計	将来的な変更・保守を考慮しているか？
テスト	実環境で問題が起きないか？ 負荷に耐えられるか？

システム設計は、「技術」だけでなく「人」「目的」「運用」を設計すること

全体のまとめ

- ソフトウェア工学とは、**再現可能で体系的な開発手法**により、高品質なソフトウェアを効率的に構築・維持するための学問領域
- その誕生は、1960年代の技術的進展と社会的失敗事例（例：OS/360）を背景に生まれた
- ソフトウェアの特徴（見えない・変化しやすい・制約が少ない）が、建築などの他分野と比べて開発を難しくしている
- 成功には、「**要求→設計→実装→テスト→保守**」の一貫したプロセスと、チームでの協調、品質の重視が不可欠
- ソフトウェア工学の知識体系（BoK）には、**要求工学・設計・構築・テスト・保守・プロジェクト管理**など多くの領域が含まれる
- 事例を通して、**技術だけでなく、意思決定、運用設計、コミュニケーションの重要性**も学んだ