ソフトウェアライフサイクル

はじめに

- ソフトウェアライフサイクル
  - ソフトウェアが生まれてから死ぬまで
  - ソフトウェアが生まれるということ（誕生）は、ビジネス上、システム上の要求を具現化して企画し計画すること。要件定義書を作成する。
- 要件定義書
  - その後のソフトウェア開発に不可欠
  - 要件定義書をもとに、システムが開発され、実かどう、保守運用をへて、ソフトウェアは一生を終える

ソフトウェアのライフサイクル

- 開発計画
  - ソフトウェア開発では、はじめに、利用目的、利用期間、利用形態などをもとに、開発目的、開発期間、開発方法、開発工数や費用の見積もりなどの開発計画を策定する。
- 開発工程
  - 開発計画に基づいて開発体制が整えられると、図のような開発工程にしたがって、開発が進められる。
  - 開発工程はいくつかの作業工程に分けられる。
    - 要求分析、外部設計、内部設計、実装、テスト、保守・運用
  - 各作業工程は開発プロセスやフェーズと呼ばれ、生成物（プロダクト）と呼ばれる。
  - 開発計画から保守（廃棄）に至るまでの開発の過程をソフトウェアのライフサイクルと呼ぶ。

（a）要求分析

- 要求分析により、要求仕様書を作成する
- どのようなソフトウェアを作るべきかを明確にして、要求仕様を定める工程
- 顧客やユーザの要求の内容を詳細に分析して、過不足なく記述する
- 開発側の視点から実現性やコスト、開発期間の妥当性などを検討して仕様書にまとめる

ソフトウェアライフサイクル

- ソフトウェアライフサイクル（Software lifecycle）
  - ソフトウェアが立案され、開発されると、一定の運用期間をへて、やがて終焉を迎える
  - 該当システムのニーズから始まる

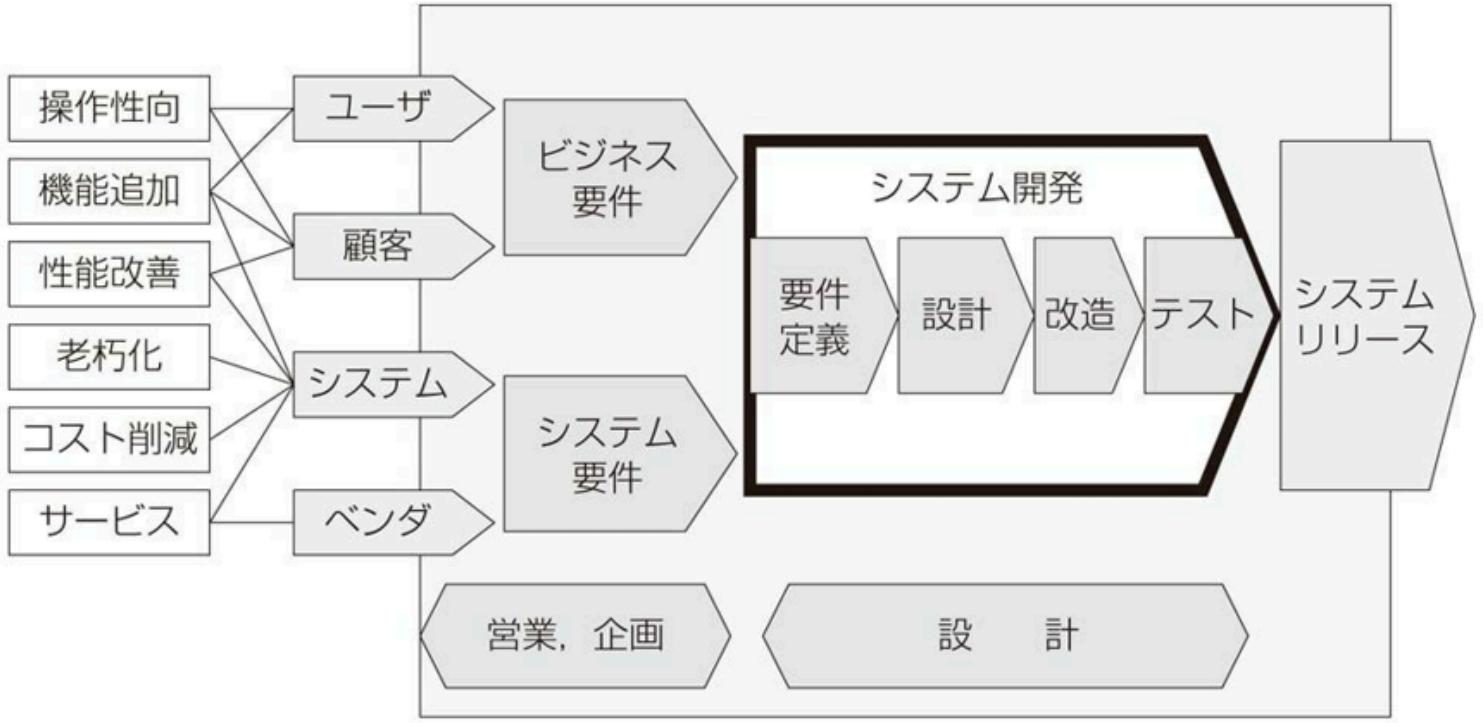


図 2.1　ソフトウェアライフサイクル

ソフトウェアライフサイクル

- ニーズをまとめる
  ビジネス要件
  システム要件
  ユーザ、顧客、システム、ベンダ
- まとめたものが、要件定義となる
- 要件定義は、そのプロジェクトの中で最も影響が大きい重要なもの

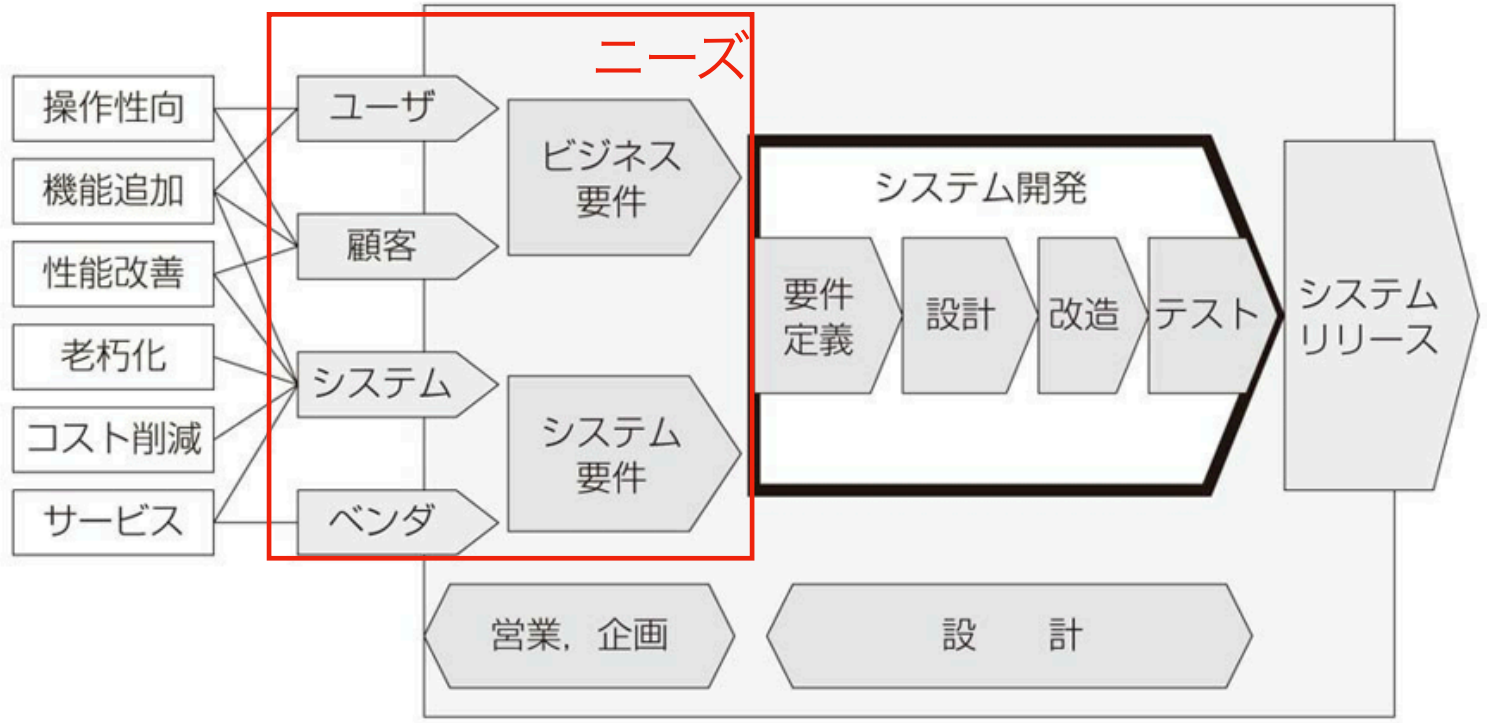


図 2.1　ソフトウェアライフサイクル

ソフトウェアライフサイクル

- ニーズをまとめる
  ビジネス要件
  システム要件
  ユーザ、顧客、システム、ベンダ
- まとめたものが、要件定義となる
- 要件定義は、そのプロジェクトの中で最も影響が大きい重要なもの

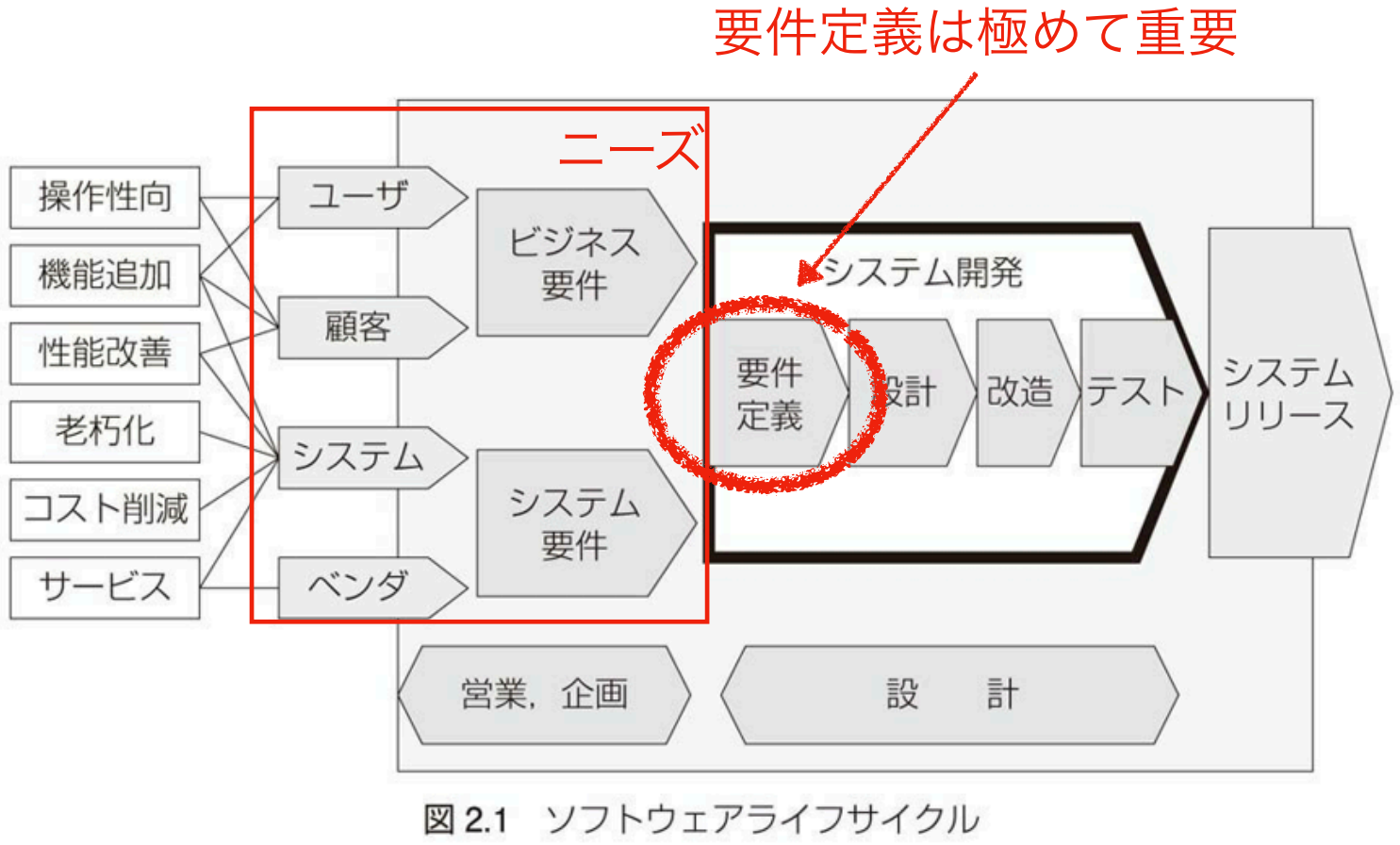


図 2.1　ソフトウェアライフサイクル

要件定義書に含むべき項目

〈要件定義書の記載項目〉

- ■ システムの目的
- ■ システムの概要
- ■ システムの機能
- ■ システム構成（システム構成図，ソフトウェアブロック図）
- ■ 目標性能
- ■ 他システムとのインタフェース仕様
- ■ 運用面の注意事項
- ■ 制限事項
- ■ 拡張性
- ■ 開発スケジュール
- ■ 開発体制
- ■ 納品物

- 項目はどれも重要
- **実現する機能と実現しない機能を明確にすること**
- これらの機能を盛り込んだシステムがスケジュール通り開発できるかの見極め

実現するかしないか

- 実現するかしないかは、実現性の裏付けが必要
  - **Feasibility study (FS)**
    - **実現性を事前に検証すること。**試作段階。やってみて初めてわかることもある
    - 試作段階のさらに前のデモンストレーションレベルの検証も重要とされる。**「概念検証」（POC: Proof Of Concept）と呼ばれる。**
- 不安材料がある場合は、機能を削る、スケジュールを見直すなど、の作業が必要となる。顧客も相手にするので大変な作業になる。
- 実現の見通しのないプロジェクトを始めることは「負け戦」。そしてデスマーチ（チーム全員が失敗の道を進むこと、成功の見通しのない作業を続けること）になる。
- 多くの企業では、これらの項目をテンプレートとして標準化し、テンプレートに基づいて要件定義を行うことが義務付けられ、漏れのない検討を行う「しくみ」を作っている

「要件定義書」の意義

- 「要件定義書」には３つの目的がある

① プロジェクト関係者のバイブル

- プロジェクトにおいて問題解決や判断の根拠にす

② 社内外への宣言と協力依頼

- 要件定義書をもとにプロジェクト憲章（プロジェクトの決まりごとを示す宣誓書や協約）を作り、社内外にプロジェクトを知らしめ協力を仰ぐ

③ RFP (Request for Proposal)

- 開発作業を外部に委託する場合に使う資料であり、システムの概要を説明しベンダからシステムの提案を要求するものである
- 多くの場合、RFPの元ネタとして要件定義書を使う。

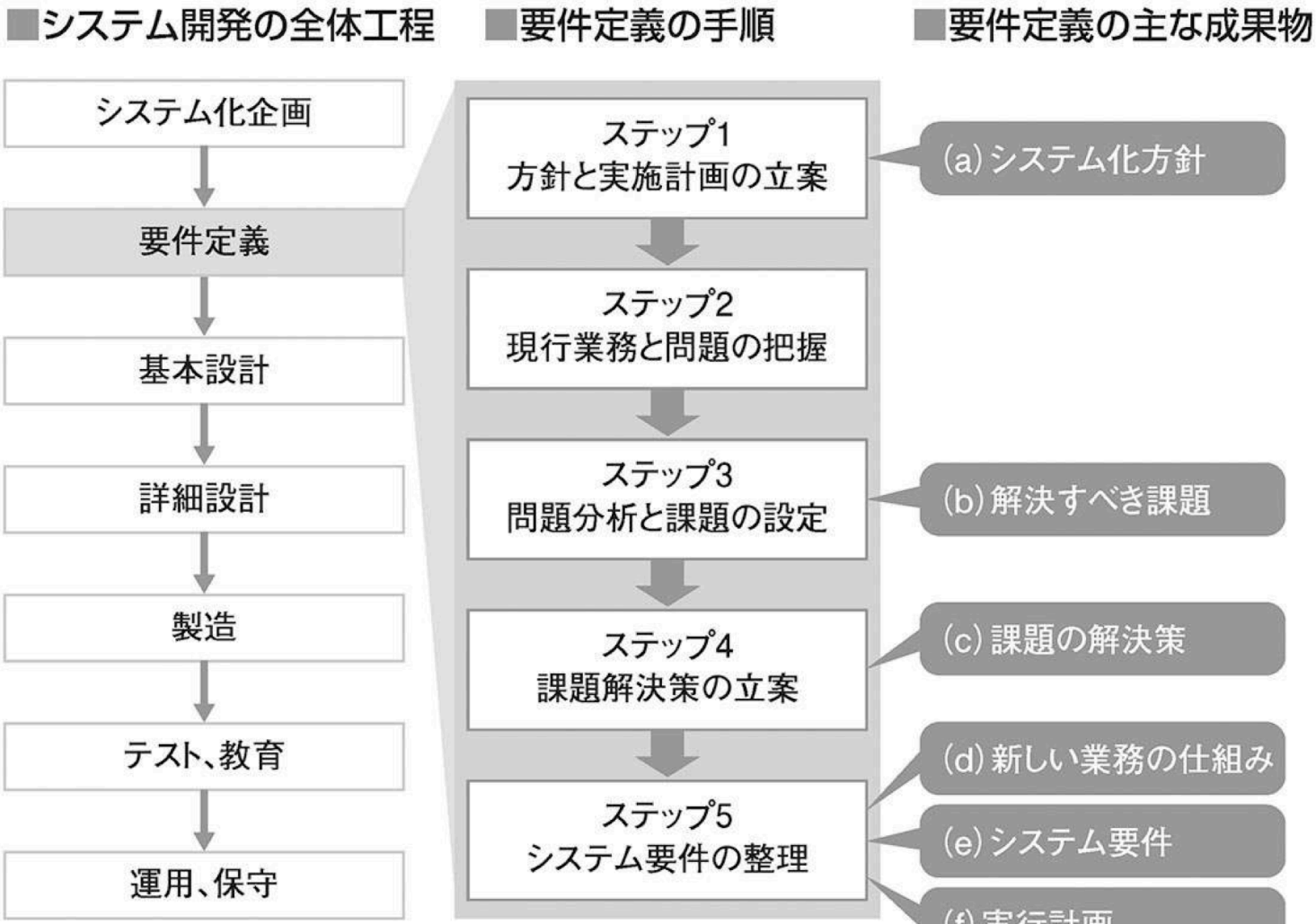


図2 システム開発における要件定義の全体手順

演習で身につく要件定義の実践テクニック. 日経BP社 (2017/11/17)より

設計

- 要件定義に基づいて設計を行うフェーズ
- 成果物は設計図であり、まだ物作りは発生しない
- 大まかな概念をより細かくブレークダウンして設計する
- 最終的にはコーディンができるレベルの仕様書に落とし込む
- どのようにブレークダウンするかがポイントで、以降のマネジメントプロセスにおいて最小の単位となる

（b）外部設計、論理設計

基本設計、**システム設計、論理設計、**機能設計、概要設計などとも呼ぶ

- システムの外部からみた仕様を設計する
  - 外部＝ソフトウェアの利用者やこのソフトウェアを利用する他のシステム
- どのような機能であれば定義された仕様を満たすかを設計する
  - 例えば、どのような入力に対してどのように動作し、どのような出力を出すかなど
- 機能の他にユーザーインターフェースや操作方法、他のシステムとのインターフェースや論理的なデータ構造なども設計する。
- 外部設計により、外部仕様書（外部設計書）が作成される。

論理設計（機能設計）

- 要件定義に基づいて論理設計を行う
- 論理設計は、ハードウェアスペックやプログラミング言語のような「具体的な実装イメージを考慮しない」ことがポイント
- 要件定義で掲げた要件を計算機システムで実現するための「仕組み」に置き換えていく作業

例：システムの大枠を考える

- 分析に使うデータはリレーショナルデータベースに保持する
- ネットワークを活用した分散システムにする
- 暗号化を行い、セキュアなシステムを実現する
- 本社システムではここまで、各支社では残りの処理を行う
- 顧客対応の処理はリアルタイムで行う
- 月次処理は月末の夜間バッチで行う

- 例のように、システムの大枠を考える作業で、アーキテクチャ設計とも呼ばれる
- システムの根幹に関わるので高度な設計レベルを要する
- 相当量のIT知識とシステムの構造や運用に関する経験を併せ持つエンジニアが必要とされる部分である
- システムアーキテクト、あるいは、システムアナリストと呼ばれる

- システムをゼロから作ることはあまりない
  - 既存のシステムへの機能追加、ネットワークシステムへのサーバの追加など
  - 既存のシステムを熟知している必要がある。
- TCO（Total Cost of Ownership）削減を意図した設計
  - TCO: コンピュータシステム構築の際にかかるハード・ソフトの導入費用から、運用後の維持費・管理費・人件費など全てを含む、システムの総所有コスト
- 論理設計では、モデル化の手法を用いることが多い
  - UMLを用いた場合、ユースケースなど図表化することで、「誰が」「何を」ということが整理され、集約されていく。さらに、クラス図、オブジェクト図、アクティビティ図へ展開する過程で、漏れやダブりに気づくこともある。オブジェクト指向開発のベースにもなる

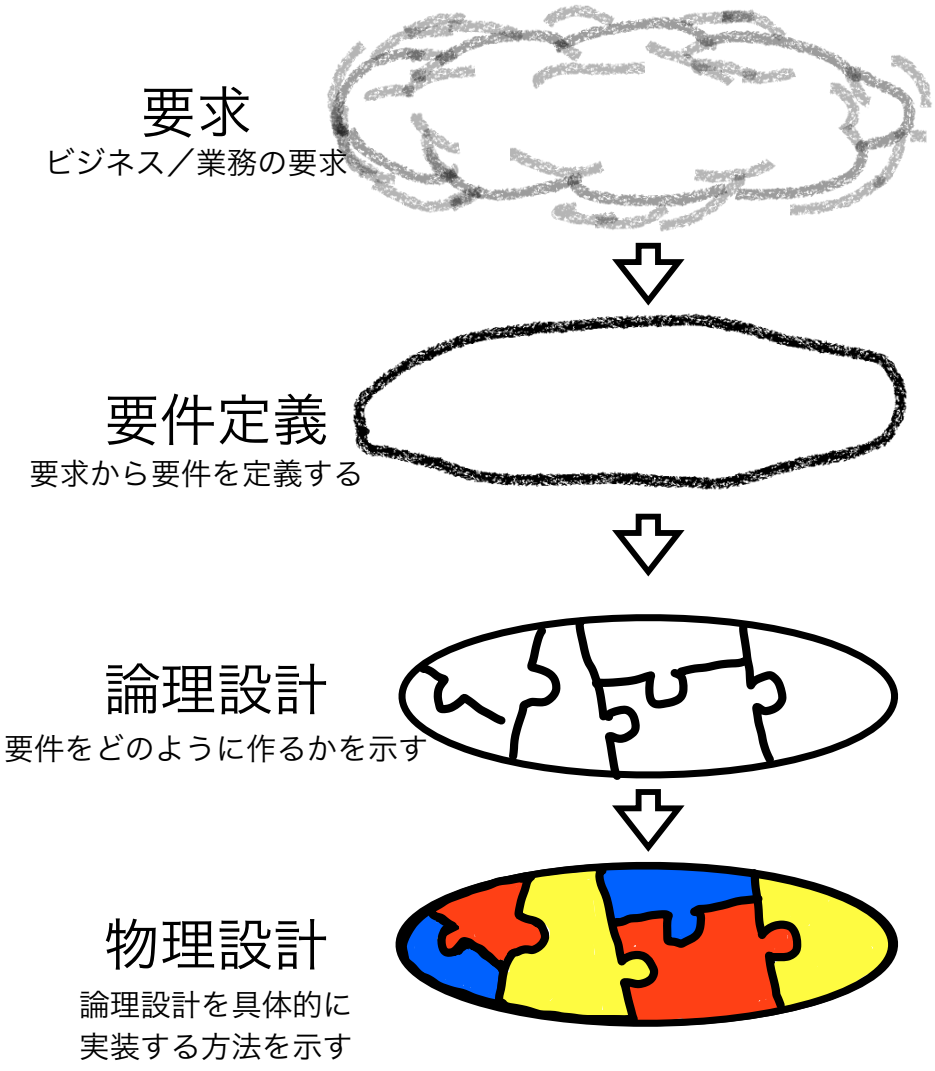
（c）内部設計、物理設計

詳細設計、**物理設計**、**プログラム設計**などとも呼ぶ

- 外部設計で構築された機能と、ソフトウェア（プログラム）的にどのように実現するかその内部方式を設計する
- すなわち、その機能によっていくつかの単位（モジュール）に分割し、個々のモジュールの仕様を作り上げると同時にモジュール間の関係を記述する。
- 内部設計により、内部仕様書が作成される。

物理設計

- 物理設計では、論理設計で考慮しなかった「実装」を意識した設計を行う。
- 論理設計で規定した内容を実際の計算機システムに落とし込み、実施方式を検討する段階である。
- 既存システムを強く意識する必要がある。時には改修する必要もある。
- 物理設計と論理設計は表裏一体。論理設計で意図したことをいかに実現するか、ということなので、論理設計と理論設計を共に意識しながら設計することが望ましい。



物理設計

- 携わるエンジニアは、広い見識のもと、採用しうる最適な技術を熟知し、最適なソリューションを見出す。
- 論理設計者が投手で、物理設計者が捕手。論理設計者と物理設計者の合意のもと「投げられた玉は必ず受け止める」という連携が必要。
- サイジング（sizing）
  - 運用するシステムやサービスの規模にあったリソース（サーバやネットワーク）を見積もること。あるいは用意しておくこと、このためにはサーバーにかかる負荷を見極める必要がある。

物理設計

- サイジング（sizing）
  - 運用するシステムやサービスの規模にあったリソース（サーバやネットワーク）を見積もること。あるいは用意しておくこと、このためにはサーバーにかかる負荷を見極める必要がある。
  - 設備投資をどの程度にするかという予測や裏付け
  - 性能に対する責任
  - システムの将来性を加味し、ビジネスの拡大を意識した、キャパシティを検討しなければならない。

物理設計

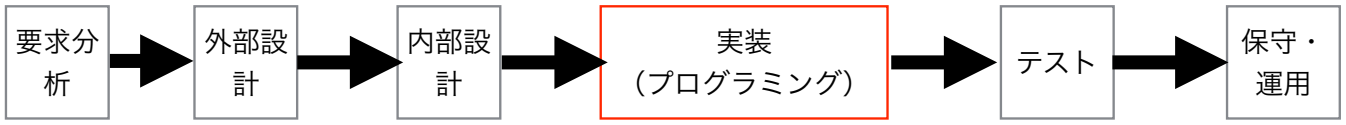
- サイジング（sizing）
  - 「最速、最強の設備を準備する」→安易
  - ライフサイクルが３年だとすると、過去の状況、同業他社の状況などを勘案し、少なくとも、数ヶ月から数年のスパンで機能拡充をしながら、ライフサイクルが満了するタイミングで、新システムへの移行ができるようなプランニングが重要。
  - 設備に関する減価償却費や投資回収など財務的な側面も重要。
- 物理設計（詳細設計）は様々な側面を検討しなければならない

物理設計

- モデリングについて
- 主役は配置図
- クラス図、オブジェクト図、アクティビティ図、シーケンス図をコーディングができる段階までブレークダウンする
- タイミング図、相互作用概念図、コンポジット図、状態マシン図などを用いて、構成要素の相互の関係と作用について詳細な設計を行う
- （重要）ユースケース図を適時参照し、ユーザの視点を忘れない！

（d）実装（プログラミング）・製作フェーズ

- フローチャートなどによりプログラミングの仕様を明確にし、プログラミング言語を用いてプログラムを記述する（コーディング）工程である。
- コーディングは（一般に）モジュールごとに行われる。
- プログラム仕様書、ソースコードが作成される。



制作フェーズ

- 制作フェーズは、設計書に基づいてシステムを仕上げるフェーズ。
- システム構築には、大きく分けて内製（自社開発）と調達（外部委託）がある。
- **内製（自社開発）**
  - 社内のメンバーで開発する形。プロジェクトを「推進できる」必要がある。
  - 頭数、要件や事例（ケース）の理解、技術的知識、素養、体制。
  - 部分的には外部委託をすることも可能
- **調達（外部委託）**
  - 開発を外部に委託し、出来上がったシステムまたは成果物を買い取る形態

調達（外部委託）

- 一般的な流れは図2.2
- 要件定義から**RFP**を抽出し、候補ベンダに提示する。ベンダから提示された提案書と見積りを吟味し業者を選定する。選定した業者と作業委託契約を締結する。

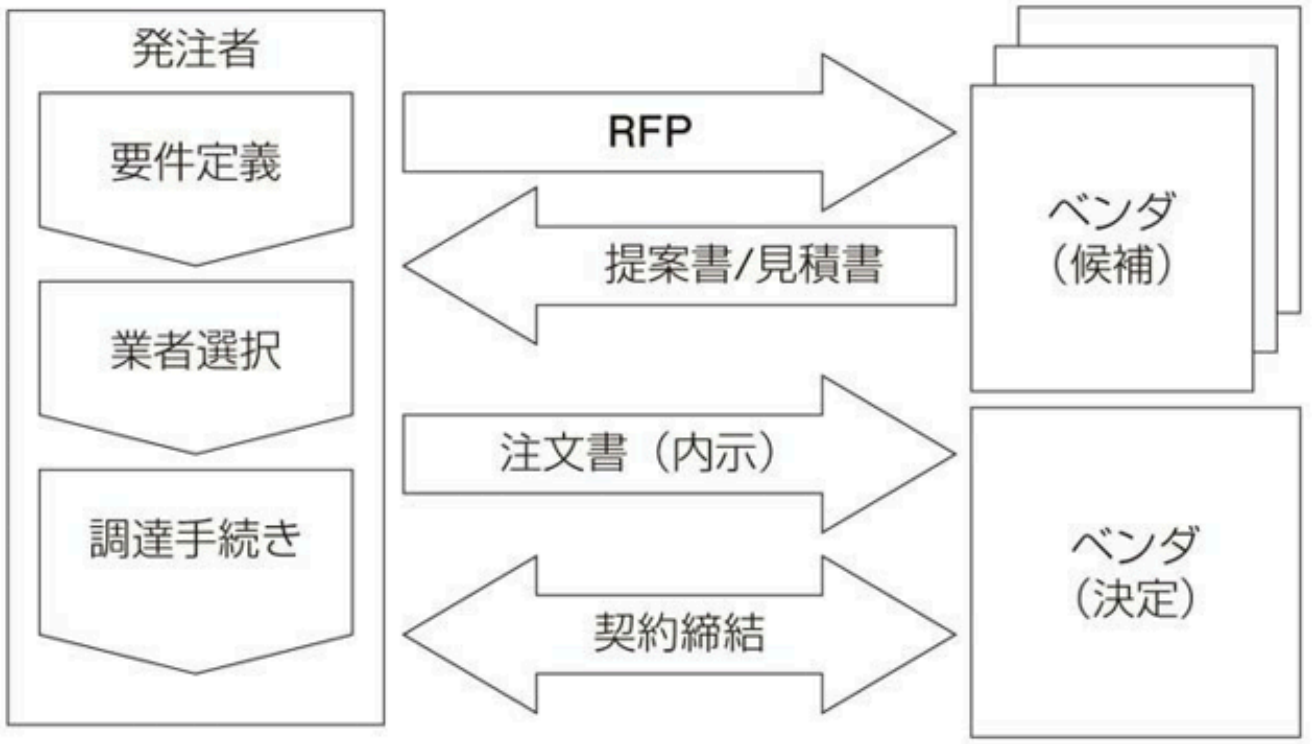


図 2.2　契約締結までのやりとり

調達（外部委託）

- 注意すべき点：**納品物の範囲と作業分担ついて合議しておく**
- 納品物の範囲について
  - 出来上がったシステムを生成するための開発環境とソースコード
  - ソースコードを受け取らない場合
    - その後の不具合を解消するために別途**作業委託契約**を結ぶ
  - ソースコードを受け取る場合
    - **自社内でシステムをメンテナンス**ができる体制化が必要
- 作業分担について
  - 明確に役割分担を決めて文書化しておく
- 守秘義務、著作権、特許の権利等も文書にしておく

調達（外部委託）：オフショア開発

- 安価な労働力を獲得できるインド、中国、東南アジアに発注すること
- 以下のことに（十分な）注意が必要
- 文化の違い
- 言語の違い
- 商習慣の違い
- 品質に対する意識の違い

- →ブリッジSEをおく
- →契約書類に明確に規定し、ドキュメントに明確に残す
- 書類については、「行間を読む」ということは一切ないので、書いてないことは範囲外とみなすのが普通。注意深いドキュメント化を怠ってはならない。

（e）テスト・検証

- プログラミングしたソフトウェアのテストを行う
- 単体テスト
  - まず各モジュールが単体で正しく動作するかどうかの単体テストを行う
- 総合テスト
  - すべてのモジュールが完成したらそれらを組み合わせたモジュール間の整合性を含めたテスト、さらに他のシステムとの関係などを含めて外部設計通りに動作するかどうかや性能を確認する総合テストを行う
- 運用テスト
  - 最終的には実際の運用環境下で要求仕様書通りに動作するかどうかの運用テストを行う
- テスト仕様書、テスト成績書が作成される

テストとデバッグ

- テストとデバッグに費やす工数は、**大きな割合**を占める
- 規模が大きくなるとテスト項目がふえ、それに比例してバグも増える
- １件のバグの対策で、原因究明、対策、確認の一連の追加作業が増えるため、バグ発生件数と作業量の関係は幾何級数的になる
- バグは少ない方がいい
- バグの摘出数と品質は統計的な分析の対象。ソフトウェア工学の真骨頂である。
- セオリーをベースに、それぞれの経験則で調整するアプローチが良い。

理想的なパターン

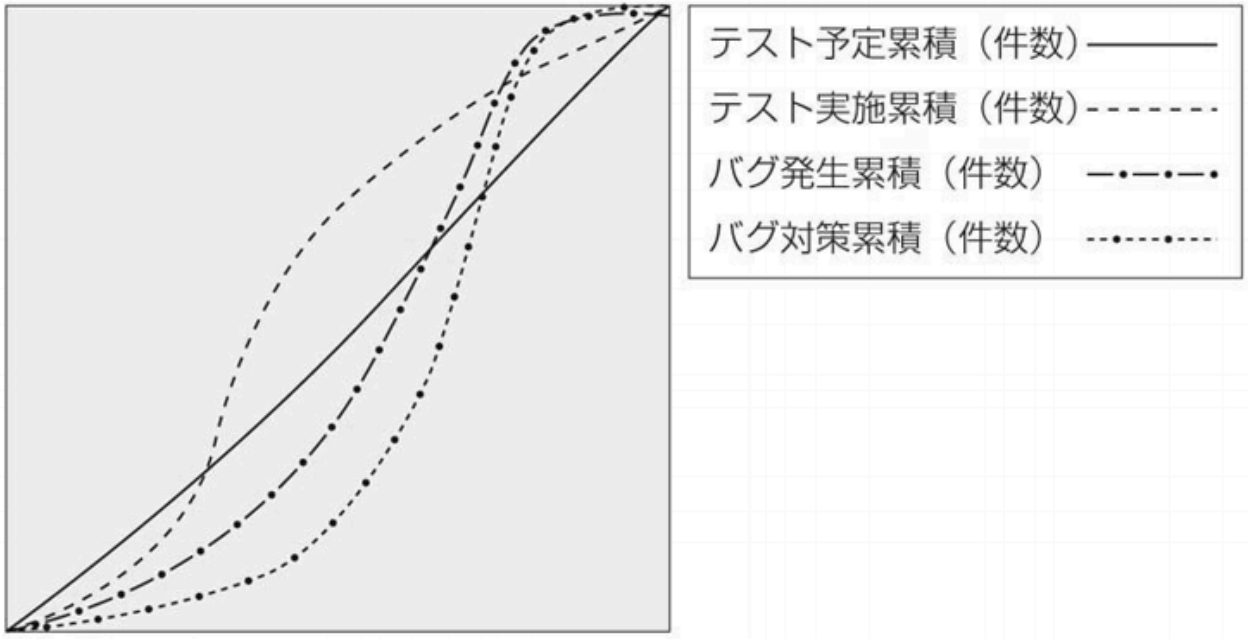


図 2.3　バグ累積曲線（1）

分析と評価

- バグの発生を追いかけるようにバグ対策が追従しており、テスト終盤でバグ発生率が低下し、やがて発生していない
- テスト消化は、テストの前半では環境整備など不慣れが原因で予定を下回っているが、テスト中盤で学習効果により件数をこなした
- 終盤でのバグの発生と対策の原因は、一定のバグが摘出され、対策されているので、一定の品質が確保できている

あるパターン

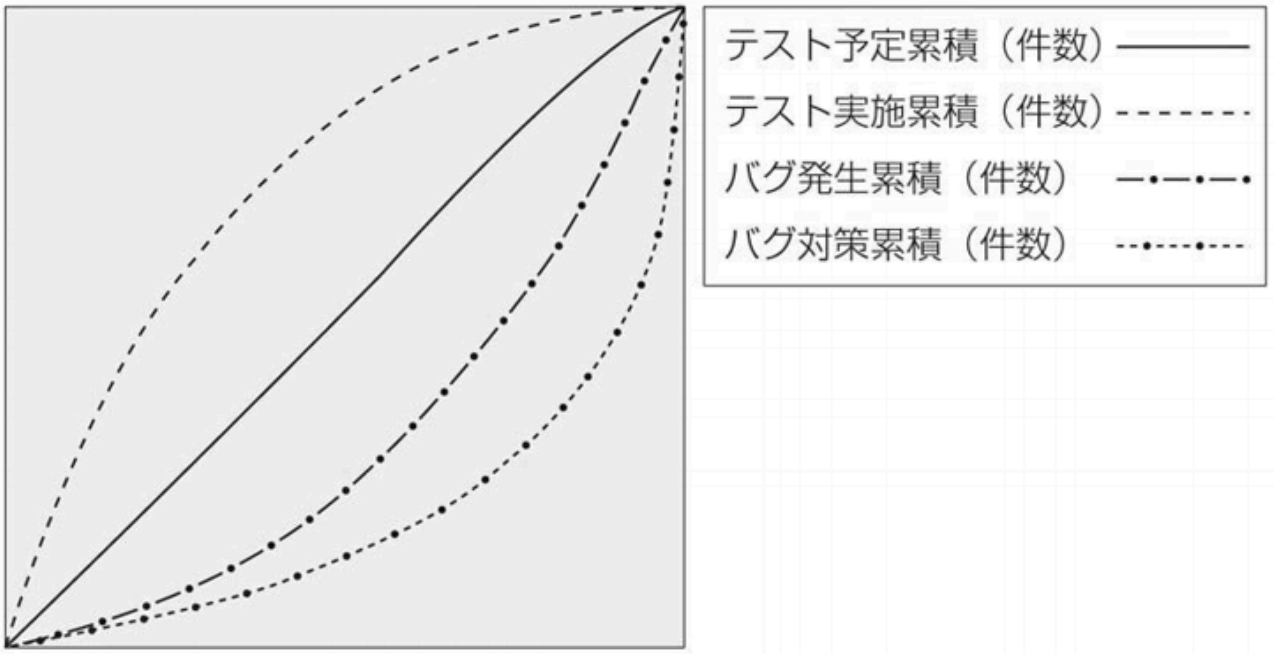


図 2.4　バグ累積曲線（2）

あるパターン

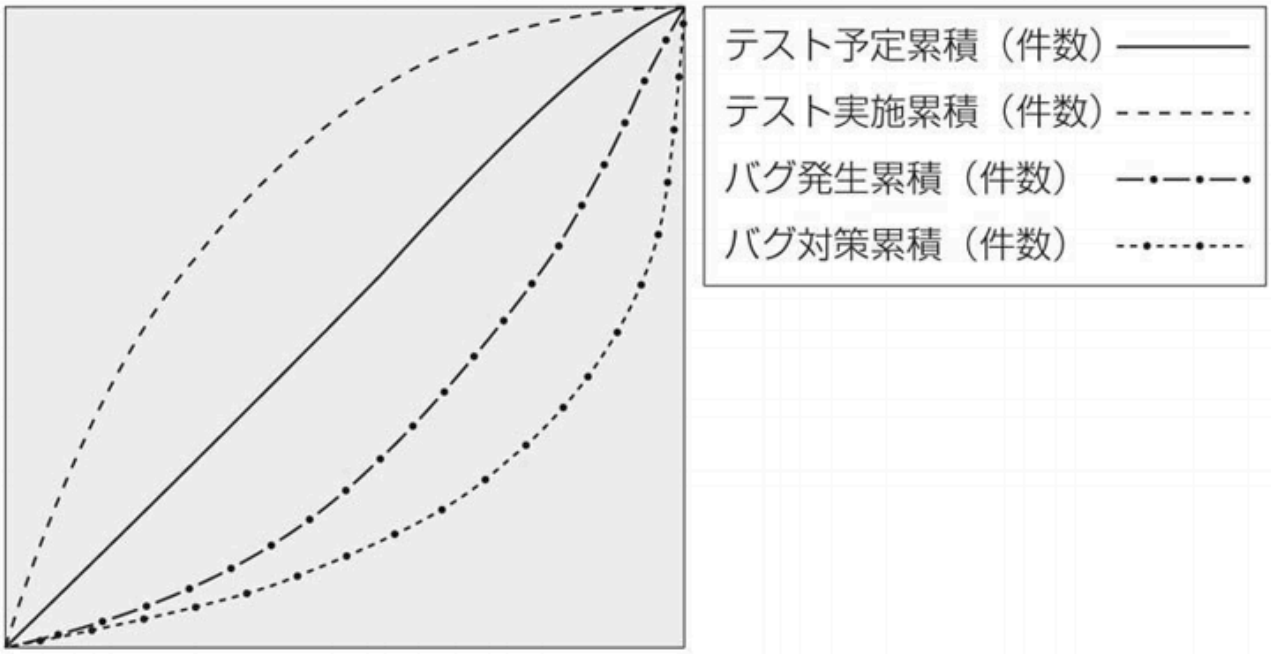


図 2.4　バグ累積曲線（2）

分析と評価

- テストの実施累積の立ち上がりはよく、前倒しで進んでいる
- バグは、テストの進捗に伴い発生しているが、後半に集中している傾向

→テストの難易度をチェックすべきである。テストの難易度が低かったため、テスト実施が前倒しに進み、その間に発生したバグの件数が少なかった、ということが考えられる。

- バグ発生パターンからは、
  - 潜在バグが摘出されてないことが推察できる、もしくは、
  - テストが特定の機能に偏っていたとも考えられる

→実施したテストの内容、範囲、偏りについて見直しを行い、適切な範囲、難易度のテイストによる再テストを行うべきである

プロジェクトマネージャの手腕

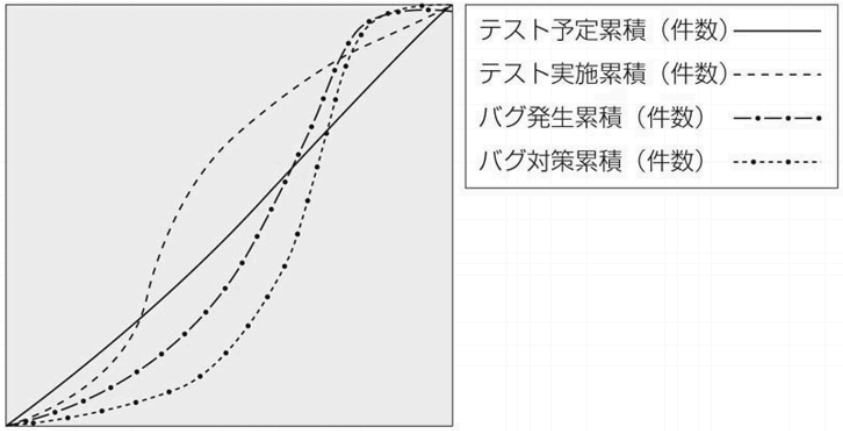


図 2.3　バグ累積曲線（1）

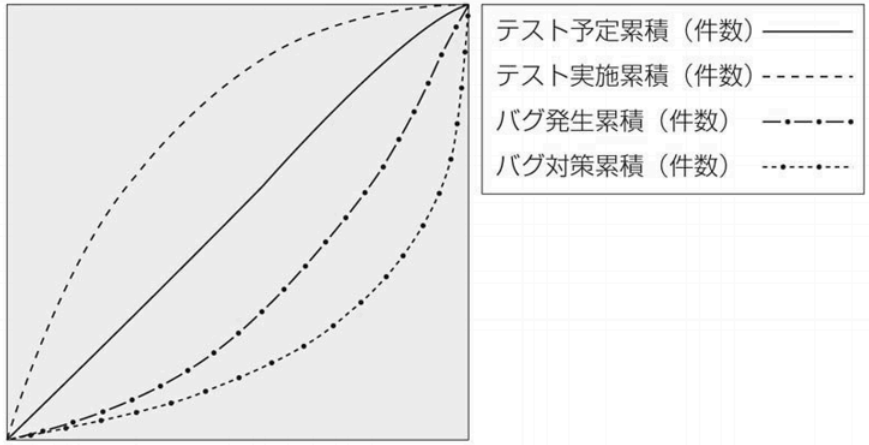


図 2.4　バグ累積曲線（2）

- このようなパターンから何を読み取り、どのように対処するかは、プロジェクトマネージャの手腕となる
- 統計データが活用する

（f）運用・保守

- ソフトウェアを運用し、維持管理する工程である。
- 運用時に不具合が生じた場合は修正し、また必要に応じて細かい改良や拡張が行われる。
- 運用に先立ち、運用マニュアル、操作説明書、保守マニュアルが作成される

- システムにはライフサイクルの期間では、システムを保守、運用する必要がある
  - 保守、運用の設計をきちんとしないと、そのコストが膨大になることもある
- 保守、運用の設計は、システムの運用上、重要な要素であり、システムの設計の段階で十分に検討し、吟味すべきである。
  - 運用担当者は顧客との接点。「生の声」は顧客満足度の向上につながる。
- Service Level Agreement (SLA)
  - サービス水準合意。サービスの提供者とその利用者の間で結ばれるサービス水準に関する合意である。AWSなどで一般的(https://aws.amazon.com/jp/legal/service-level-agreements/)
  - 保守、運用業務の「見える化」の一環として、各種の数値情報を定義し数値化、制御する。
  - 保守、運用費用の適正化でもあり、TCOの削減に持つながる

保守性

- 例：AWS S3
  - 99.9%がService Level Agreement
  - 95%以下になったら、１００%返金

Monthly Uptime Percentage	Service Credit Percentage
Less than 99.9% but greater than or equal to 99.0%	10%
Less than 99.0% but greater than or equal to 95.0%	25%
Less than 95.0%	100%

バスタブ曲線

- 運用・保守では、稼働後のログにより、品質データを把握できる
- ログのプロアクティブな活用が重要
  - プログラムが特定の条件で異常終了していたらそこに問題があると思う
  - 一定時間以上稼働しているハードウェアは定期的に交換
- 一般的な故障率の推移をグラフ化したものをバスタブ曲線と呼ぶ
  - 導入直後は初期異常が発生し、安定稼働し、時を経て「摩耗故障」によってシステムは終焉を迎える

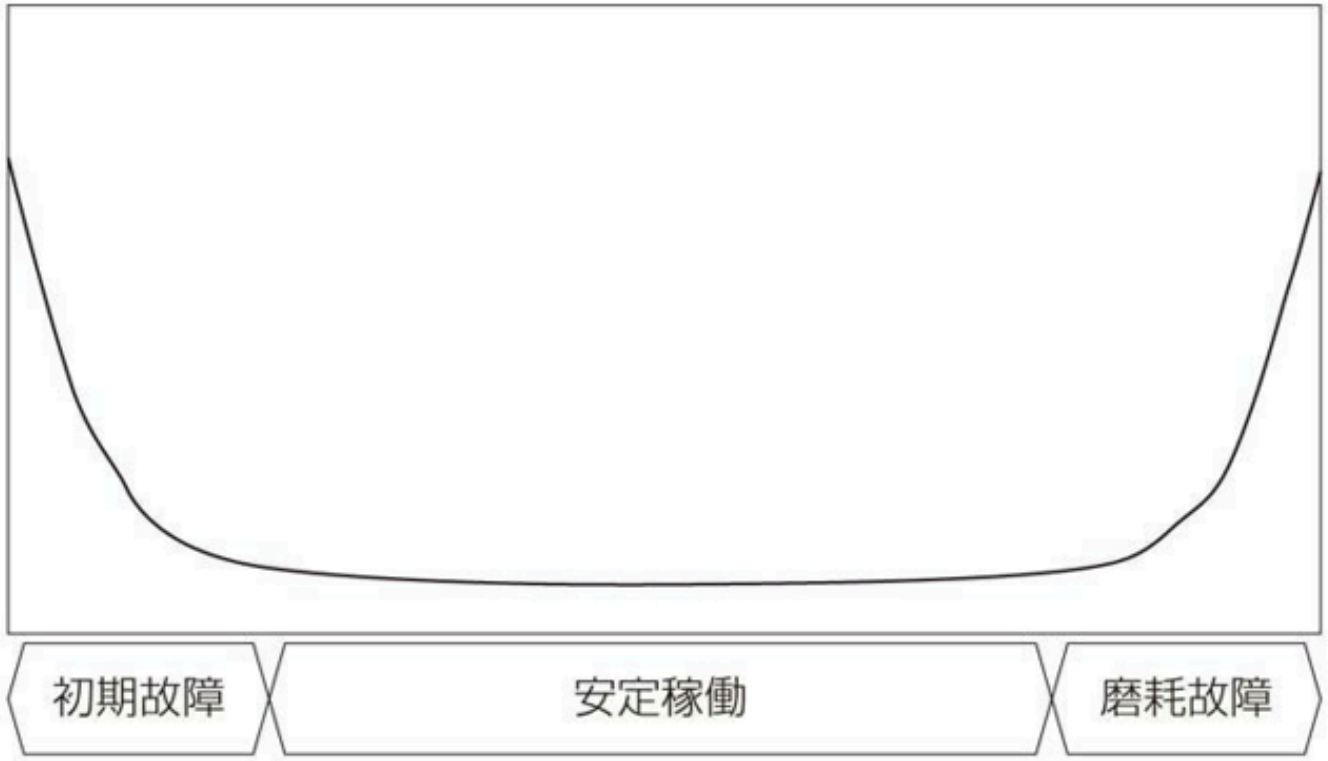


図 2.5　バスタブ曲線