**上流工程研修 資料**　　　　　　　　　２０１７／５／１１版

**［内容］**

[**【１】** ソフトウェア開発の基本知識 2](#_Toc481486949)

[1. 開発プロセスの種類と特徴、長所と欠点 2](#_Toc481486950)

[2. ウォーターフォール型開発プロセスの工程と成果物 5](#_Toc481486951)

[3. システム形態 8](#_Toc481486952)

[4. 開発環境 11](#_Toc481486953)

[**【２】** 上流工程の重要性、設計のポイント 16](#_Toc481486954)

[1. 利益とスケジュールが決まる 16](#_Toc481486955)

[2. 「できないこと」の明確化 17](#_Toc481486956)

[3. 「丸投げしないこと」「提案型」のＳＥであること 17](#_Toc481486957)

[4. お客様の宿題事項の回答がない 18](#_Toc481486958)

[5. テスト項目を設計時に作成すること 18](#_Toc481486959)

[**【３】** 要求分析 19](#_Toc481486960)

[1. 打合せの一般的な注意事項 19](#_Toc481486961)

[2. 業務知識について 19](#_Toc481486962)

[3. ヒアリングで聞き出すこと 20](#_Toc481486963)

[4. ヒアリングへの対応姿勢 20](#_Toc481486964)

[**【４】** 要件定義 21](#_Toc481486965)

[1. 要件定義とは 21](#_Toc481486966)

[2. お客様の要求事項を機能要件と非機能要件に分けること 21](#_Toc481486967)

[3. 仕様書の書式の標準化 21](#_Toc481486968)

[4. 細かい点まで注意して分析、設計する（当たり前と思って省略しない） 22](#_Toc481486969)

[5. 疑問点はできるだけ早く解決 22](#_Toc481486970)

[6. 要件定義のゴール 22](#_Toc481486971)

[7. 成果物 22](#_Toc481486972)

[**【５】** 基本設計（外部設計） 23](#_Toc481486973)

[1. ここでやること 23](#_Toc481486974)

[2. フレームワーク（実行環境） 23](#_Toc481486975)

[3. 入出力をまとめる 24](#_Toc481486976)

[4. 表示項目とレイアウト設計について 24](#_Toc481486977)

[5. 処理の設計にあたって 24](#_Toc481486978)

[6. 処理の機能分割の手順 25](#_Toc481486979)

[7. バグが混入したり設計不能となる場合 25](#_Toc481486980)

[8. 成果物 26](#_Toc481486981)

1. ソフトウェア開発の基本知識
2. 開発プロセスの種類と特徴、長所と欠点

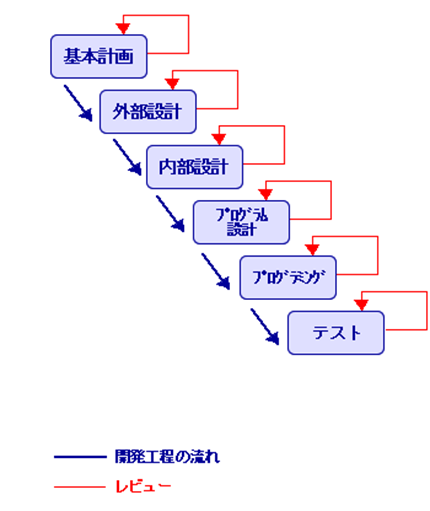
現在、実際に採用されている開発プロセスとしては

1. ウォーターフォール
2. プロトタイプ
3. スパイラル
4. RAD、インクリメンタル、インタラクティブ
5. アジャイル

が挙げられる。

この内②～⑤は主に小規模システム開発に使用されており、一般的なシステム開発で使用されているのは①のウォーターフォールである。

1. ウォーターフォール

開発の開始から完成まで一方通行で開発を進める方法。進捗状況が分かりやすい。やり直しに弱い

ざっ

くり

名前の通り滝のように上流の工程（企画、計画）から、下流に向かって設計、製造を経てのテストまで準備こなしていくプロセス

（左図で「基本計画」は「要求分析」「要件定義」を含む）

長所：

* 工程が一本道で進んでいくため、進捗が把握しやすい。（特に、大規模プロジェクトで有利）
* 工程完了時のレビューと品質評価の実施により、確実に品質向上が見込める。

短所：

* 仕様変更などによる工程後戻りに弱い（設計、製造のやり直し）。戻り先が上流であるほど、全体としての工数増加が甚大となり、また品質を落とす原因となる。
* 中間の工程（プログラム設計、製造、テスト）で必要工数が増えることから、必要工数が平準化せず（山形となる）、プロジェクトの人員編成が困難。
* 他の長所としては、プロジェクトの各工程の終了日が明確であることから、各工程終了時点での検収（成果物の納入とお客様からの入金）が行いやすい。  
  大規模プロジェクトの場合、システム運用開始（カットオーバー＝C/O）まで長期間となることが多く、それまで検収が行われないとなると、請負側としての資金繰りが困難になることから、中間段階での検収をいただけるよう、お客様と交渉していく必要がある。

1. プロトタイプ

試作品を作り、少しずつ仕上げて行く反復型開発方法。お客様のニーズに合わせ易いが、完成時期が見えにくい。

ざっ

くり

①のウォーターフォール開発を諸規模に繰り返し、試作品の作成を繰り返しながら少しずつ完成品の仕様に近づけていくプロセス。

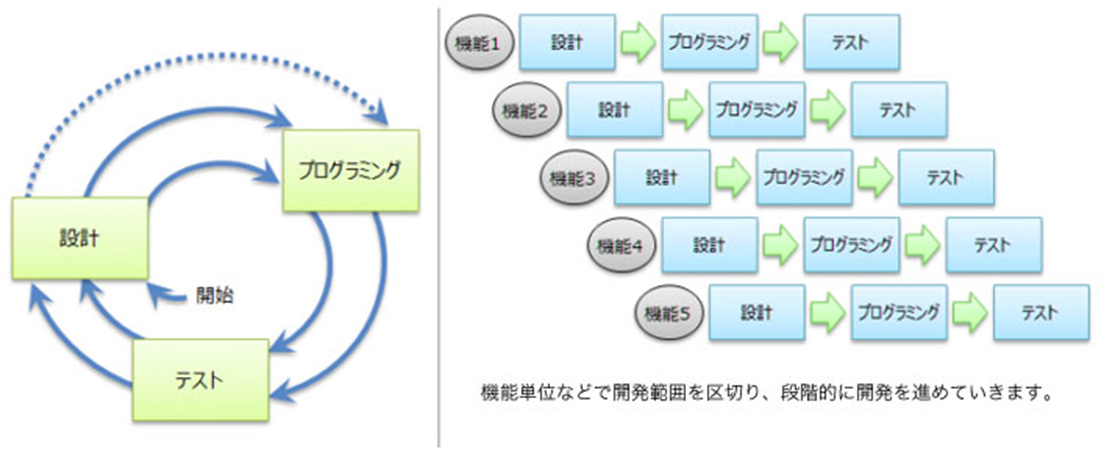
長所：

* 試作ごとのお客様との仕様の確認作業により、仕様についてのお客様との齟齬が生じにくい。
* 仕様変更が行われても、戻る部分が限定されていることから対応しやすい。  
  （お客様自身が、開発開始時点で要求を明確化できていない場合に有効）

短所：

* 全工程終了時期が不明確（各試作の段階で確定する仕様の明確化が難しい）
* お客様が試作品に必要以上の仕様の詳細化を要求することがあり、次の試作の際に破棄する設計が増えてしまって、コストがかかってしまう恐れがある。  
  （単なるウォーターフォールの後戻りと同じことになる）
* 試作の度に仕様が明確化されることによって、お客様からの要求仕様が肥大化する恐れがある。  
  最終的な機能仕様の大枠をはみ出さないように注意が必要。
* ウォーターフォールほどではないものの、必要工数の変動は小規模ながら発生する。

1. スパイラル



機能ごとに分けて平行して開発する方法。仕様変更が発生した時の影響範囲を小さくできるが、プロジェクトの管理が難しい。

ざっ

くり

独立した機能単位ごとに開発範囲を区切って、同時並行で開発を行う。

長所：

* プロトタイプの長所である、「お客様との仕様の齟齬が生じにくい」「仕様変更への対応が容易」であることは、このモデルでも当てはまる。
* すべての工程が同時に存在することから、必要工数の変動が少なく、人員構成が平準化する。

短所：

* 各機能開発プロセスが平行することから、各プロセスの進捗状況確認の作業量が増大する。
* 全工程終了時期がずれる可能性が高い
* 機能ごとの開発期間を平準化するための工数見積もりが難しく、ある機能の開発スケジュールが突出して乱れた場合、工数の平準化が乱れたり、機能間のインタフェースが密である場合、他の機能開発に影響が及んで、全体のプロジェクトスケジュールに乱れが生じる）

1. RAD、インクリメンタル、インタラクティブ

【RAD】効率的に開発できるツールを使いこなせる人によって、短時間で開発する方法。【インクリメンタル】機能を少しずつ追加していく反復型開発方法。【インタラクティブ】上記に述べた反復型開発で、１サイクルごとに次のサイクルでの開発内容を見直して行く方法。

ざっ

くり

RADは、高生産性開発ツールと、これを使いこなせる少数精鋭の開発者による、短期間反復型の開発方法である。主に小規模開発に向く。

インクリメンタルは、プロトタイプと似ているが、１サイクルの開発で作成するものは試作ではなく、開発開始時に、最終的な全体機能を分割しておいた１部分機能の完成形であることが異なる。

インタラクティブも反復型であるが、各開発サイクルでの到達目標は、各サイクルごとに開始前に設定し、サイクル終了時に評価を行ってから、次のサイクルの目標を定める。

1. アジャイル

反復型開発だが、お客様をチームに巻き込んで行う方法。具体的な開発方法は手法ごと異なっており、それぞれ細かい決まりごとがある。

ざっ

くり

反復型の開発ではあるが、単なるプロセスだけの話でなく、チーム編成や各工程の進め方などを含めた方法論を含んでいる。  
いろいろな手法（ＸＰ、スクラムなど）が存在するが、主に以下のような特徴を持つ。

* チームメンバーにはお客様も参加する
* 反復単位は非常に短い（２週間程度）
* 常にチームの合議でプロセスを進める（リーダー先導ではない）
* プロジェクトの目標は「当初予定仕様１００％カバー」に拘らず、システムとして「最大限の効果がある」ことを目標とする。

1. ウォーターフォール型開発プロセスの工程と成果物

特に多用されるウォーターフォール型のプロセスと工程について説明する。

プロトタイプなどの反復型のプロセスにおいても、本質的に以下に述べている工程は存在し、これを短いサイクルで繰り返しているものである。工程の意味や位置づけについては変わらない。

* 1. 要求分析

お客様の作って欲しいシステムの要望をヒアリングし、まとめる工程。（成果物：要求仕様書）

ざっ

くり

お客様からのシステム化の要求事項をヒアリングなどを通じて、分析する工程。

お客様の業務分析を行い、システム化を要求されている部分を確定する。

* + 成果物：要求仕様書
  1. 要件定義

要求分析の結果に基づき、開発するシステムの機能を決める工程。（成果物：機能一覧表、Ｉ／Ｏ関連図、業務フロー図、等）

ざっ

くり

何をつくるのか？ということを定義する。

要件とは、主に次の２つから成り立っている

1. 機能要件

業務を行う処理の定義。

何らかの入力データを処理し、処理を行い、出力を得ることシステムであり、その具体的な定義を行う

1. 非機能要件

業務そのものではなく、業務を行うに当たっての要求事項。

性能（レスポンスタイム、等）、セキュリティ、リソース制約（ハードウェアや基本ソフトウェアなどによる、使用可能なリソースの制約など）、バックアップなどの運用関係など。

* + 成果物：機能一覧表、Ｉ／Ｏ関連図、業務フロー図など
  1. 外部設計（基本設計、機能設計、概要設計）

システムの機能構成や、お客様から見える部分、操作していただく部分を設計し、お客様にシステムの全体イメージや使い方を理解していただく工程。（成果物：システム項目一覧、画面や帳票のレイアウト図、画面の遷移図など）

ざっ

くり

「要件定義」された、システムが「何をするか」について、これを「どうやってやるか」に変換する作業である。

要件定義と後に続く詳細設計との間で行わなければならない設計を含む。

システムとは「入力→処理→出力」であると考えられるので、

この工程では主に入出力（ユーザーインタフェース）の設計

（画面設計、帳票設計、データベースへの格納データ、ネットワークに流れるデータ、など）、

および、処理方式の詳細について設計を行う。

* + 成果物：システム項目一覧、画面レイアウト図、帳票レイアウト図、画面遷移図

　　　　イベント一覧、更新・処理仕様書など

* 1. 内部設計（詳細設計、構造設計）

システムを実現するために作成するソフトウェアの物理的な構成要素（画面、入出力データ、データベースなど）の具体的な設計を行う工程。（成果物：画面や帳票の設計図、テーブル設計図、インタフェース仕様書など）

ざっ

くり

外部設計で設計した入出力、処理内容をプログラム、フレームワークでどのように実現するかを設計する。

入出力については、使用する媒体（画面、帳票、ファイル、データベース等）とレイアウトの設計、処理については、プログラムの構造や処理を設計する。

例えば、画面、帳票については、具体的なレイアウトの設計、データベースについてはテーブルやリレーションの設計、処理については、プログラムのモジュール分割や実際の処理手順の設計を行う。

* + 成果物：画面設計図、帳票設計図、ファイルフォーマット定義書、テーブル設計書、ER図、プログラムインタフェース仕様書、フローチャートなど
  1. 製造（コーディング、プログラミング）

ソフトウェアの物理的な構成要素の製造を行う工程。（成果物：プログラム本体、画面、帳票、データベースなど）

ざっ

くり

実際の入出力媒体やプログラムの製造を行う。

* + 成果物：プログラム本体、画面（フォーム）、帳票（EXCELファイル）、データベースなど
  1. 単体テスト

製造したソフトウェアの各プログラムのプログラム要素（各行、ブロック、変数など）が狙い通りの動作をしているかどうか確認する工程。（成果物：単体テスト仕様書、結果報告書）

ざっ

くり

製造したプログラムのテストを行う。

テストは、通常プログラムの作成者が行い、プログラムの処理の論理構造（ロジック）に誤りがないかどうかについてのテストを行う。  
このテストでは、プログラムが実現している機能の処理結果の正誤を確認するのではなく、あくまでプログラム言語で製造（コーディング）したプログラムそのもの（いくつかの行で構成される処理の単位ごと、およびモジュール単位ごと）が想定通りに動作しているかどうかの確認を行う。  
（機能の構成要素である実際のプログラムを確認してテストを行うことから、「ホワイトボックステスト」とも呼ばれる）

* + 成果物：単体テスト仕様書、単体テスト結果報告書
  1. 結合テスト

製造したソフトウェアが複数のプログラムで構成されている場合、プログラム間で取り決めたインタフェースが正しく実装されているか確認する工程。インタフェースが少ないときには別のテストの中で一緒に行う（成果物：結合テスト仕様書、結果報告書）

ざっ

くり

主に規模の大きなシステムにおいて、製造したプログラムが複数のモジュールに分かれており、特にモジュール間でのインタフェース（規約、プログラミング上の約束事）が多い場合に、その確認を行うテストである。  
モジュール構成が単純なシステムや、１名での開発など、特に確認の必要がない場合には、行う必要がないか、あるいは単体テストに含んで行われることが多い。

* + 成果物：結合テスト仕様書、結合テスト結果報告書
  1. 機能テスト（総合テスト）

製造したソフトウェアが、お客様の要求通り正しく機能するかどうか確認する工程。（成果物：機能テスト仕様書、結果報告書）

ざっ

くり

プログラムが実現しているシステムの各機能の動作確認を行うテストであり、実際にお客様が行う業務の流れに基づいて実際の操作を行うことで各機能の動作結果の確認を行う。

* + 成果物：機能テスト仕様書、機能テスト結果報告書
  1. システムテスト

製造したソフトウェアが、実際に使われる環境下で正しく動作するか確認する工程。（成果物：システムテスト仕様書、結果報告書）

ざっ

くり

システムの開発は、開発用の環境で行われるが、最終段階のテストとして、実際に業務で使われているシステム（本番系）を使用し、できるだけ業務で使用している環境に近い状態で行うテストである。テスト項目は、実行する環境に依存するものを除いて、通常は機能テストと同じである。  
ただし「本番系」とはいっても、実際には本番系の予備系（本番系の故障時に切り替えられるように待機しているシステム）で行われることが多い。

成果物：システムテスト仕様書、システムテスト結果報告書

* + 上記以外の成果物

開発においては、プロセスとは関係なく以下のようなドキュメントが作成される。

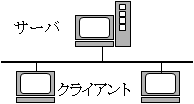
レビュー記録表（レビューでの指摘事項と、その解消の確認管理）  
Q&A票（お客様、あるいは他の担当者に行った質問と、その解消の確認管理）  
議事録  
など

1. システム形態
   1. クライアント・サーバシステム（C/S）

お客様の要求を受け取るコンピュータ（クライアント）と、要求を処理するコンピュータ（サーバ）に役割分担した複数のコンピュータで構成されたシステム。

ざっ

くり

処理を要求するコンピュータ（クライアント）と要求された処理を行うコンピュータ（サーバ）を分けた構成によるシステム。

クライアントはＵＩ（ユーザインタフェース）を担当し、サーバは通常はデータベースサーバであることが多い。アプリケーションとしての処理（ビジネスロジック）は、主にクライアントが担当することが多い。

長所：

* 負荷が複数のコンピュータに分散されているため、一台のコンピュータですべて処理するよりもコンピュータへの負担が小さい。
* クライアントのＵＩ開発ツールの充実、ハードウェアの価格低下（クライアントコンピュータの価格低下、サーバのクラウド化）により、安価、短期間に構築可能。

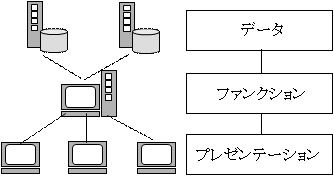
短所：

* 仕様変更などのシステム更新時に、ビジネスロジックを受け持っているクライアントの更新作業が発生するが、通常、多数存在するため、作業量が大きくなる。
* サーバに処理部分が置かれて「いない」ことにより、クライアントとサーバ間の通信が増大しシステム応答が遅延する。
  1. ３層システム

お客様からの要求の受け取り、結果表示するだけのコンピュータ（プレゼンテーション）と、要求内容を解析し処理するコンピュータ（ファンクション）と、そのためのデータを蓄積、提供するコンピュータ（データ）に役割分担した複数のコンピュータで構成されたシステム。

ざっ

くり



クライアント・サーバでの短所をカバーするために、クライアントをＵＩ（プレゼンテーション層）と要求を処理（ビジネスロジック）する部分（ファンクション層）に分けたもの。

主にＷｅｂアプリケーションで多用され、この場合は

第１層：プレゼンテーション　：　Ｗｅｂブラウザ

第２層：ファンクション　　　：　Ｗｅｂサーバ、アプリケーションサーバ

第３層：データ　　　　　　　：　データベースサーバ

となる。

長所：

* クライアントの機能をＵＩに特化することで、非常に軽くすることができる。  
  Ｗｅｂアプリケーションを３層システムで構築した場合、クライアントをブラウザにすることができ、クライアントのハードウェアの選択の自由度が上がる（クライアントのOSはブラウザが動けばよい）。
* クライアントとデータのサーバ間の直接の通信がなく、ネットワークの負荷を下げることができる。
* システム更新時のクライアントへの影響が少ない。（クライアントにビジネスロジックの処理部分がないため）

短所：

* システムの構造が複雑になり、開発にスキルが要求され、また開発工数も。
* クライアントの機能が少なくなった分、ＵＩの自由度が低い。  
  （ただし、JavaScript等の使用により、ある程度の表現力の向上は可能）
  1. ＭＶＣモデル

お客様からの　特にWebシステムの開発のために、機能をModel（実際の処理）、View（表示）、Control(要求の受け取り、ModelとViewに指示する)に分けて設計し構成されたシステム。

ざっ

くり

Ｗｅｂアプリケーションの構造モデルで、機能を

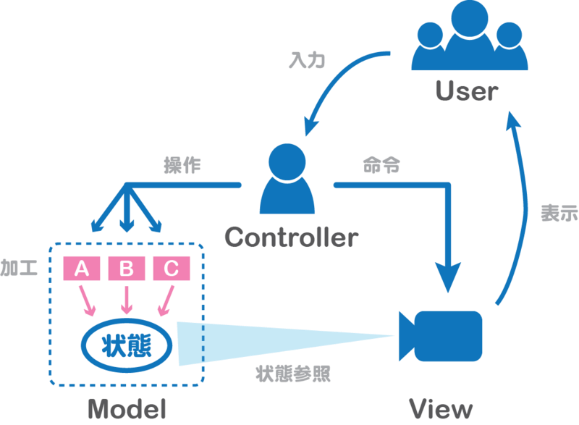
Ｍｏｄｅｌ

Ｖｉｅｗ

Ｃｏｎｔｒｏｌｌｅｒ

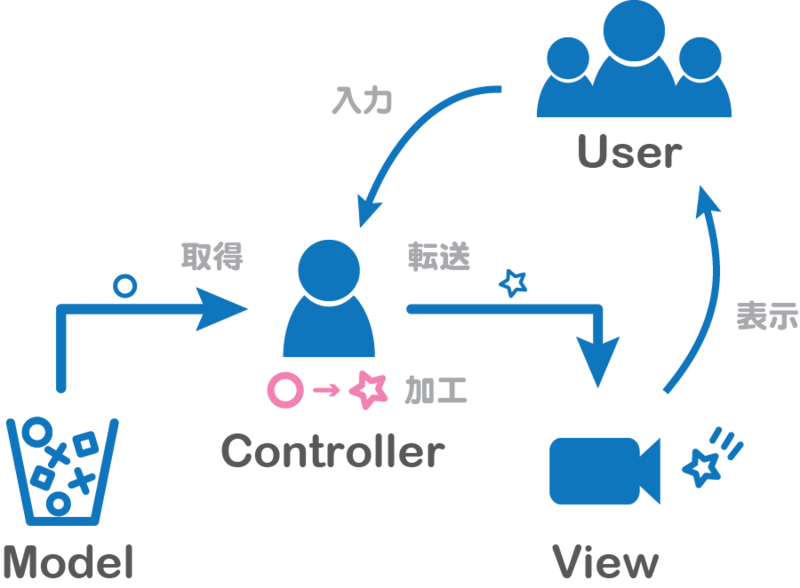
に役割分担させる。

それぞれの働きを図示すると下記のようになる



1. Userからの入力をControllerが受け取る
2. ControllerはModelにデータを加工（ビジネスロジック）させる（オブジェクト指向モデルでは加工メソッドの操作）
3. 加工によってModelの持つ "状態" が変わる
4. ControllerがViewに命令を送る
5. ViewはModelの "状態" を見て、その内容をディスプレイに表示する

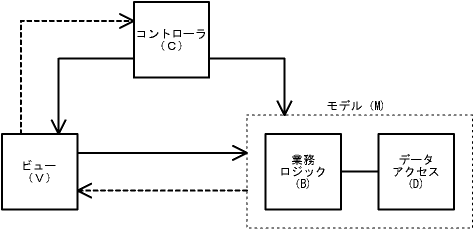
ＭＶＣについては、以下のように考えられることが多いが、これは誤りである。



左図ではビジネスロジックはControllerに置かれているが、これは誤りで、本来はModelに置くべきである。

Controllerの処理は、ModelやViewへの操作、指示だけであり、元々処理が大きくなるものではない。

また、ＭＶＣモデルを、先の３層システムに当てはめると、以下のようになる。



第１層のプレゼンテーションがController＋Viewとなり、

第２層のファンクションと第３層のデータがModelに含まれる。

1. 開発環境
   1. 主要なプログラミング言語
2. Ｃ言語

* 元々システム記述言語として作られ、ＵＮＩＸシステムの開発に使用された。
* 構造化記述の可能な言語であるが、一方でハードウェア寄りの（機械語的な）低級言語としての記述も可能。
* コンパイルすることで、完全な機械語に変換することができ、作成したアプリケーションを実行するための環境（「実行環境」「ランタイムライブラリ」などと呼ばれる）を準備、インストールする必要がないことが多い。
* 言語仕様は小さく、主な機能は関数ライブラリで実現させており、ハードウェア独自部分をライブラリに吸収させることで、多様なハードウェア環境でプログラミング環境（コンパイラ、ライブラリなど）が提供されている。
* 実行環境がほとんど必要ないことと、ハードウェアを直接制御できることから、現在も組込みシステム（使用可能メモリなどの制限が厳しい環境）で使われる主要な言語の一つである。
* プログラマーはアプリケーションとしての処理以外に、プログラムの動作上必要な操作（メモリ管理、バッファオーバーフローチェックなど）を、自ら処理に組み込まなければならないことが多いため、初心者は重大なバグを発生させる可能性がある一方、うまくコントロールすることで実行性能を上げることもできるため、プログラマーのスキルが高ければ非常に高速、効率的なアプリケーションの開発が可能。
* その後、Ｃ＋＋、Ｊａｖａ等に発展している。

1. Ｃ＋＋

* Ｃ言語にオブジェクト指向の仕組みを導入したもの。Ｃ言語の上位互換がある。
* Ｃ言語と同様にコンパイルにより完全な機械語に変換可能である。
* WindowsではVisual C++として提供されており、WindowsのOSそのものやMicrosoft Officeなどのアプリケーションソフトの開発に使用されている。
* システム構築のためのライブラリやフレームワーク（WindowsではMFCなどのライブラリや、.NET Framework）を使用することが可能になっており、そのため言語仕様が肥大化している。
* C言語と同じくメモリ管理などの操作を自らプログラミングする必要があり、言語仕様の大きさもあって、習得の難易度は高い。

1. Ｊａｖａ

* 上記のC、C++から発展したもので、様々な用途に使用することができるよう、汎用性に富んだ言語仕様を持つ言語。
* 作成したアプリケーションのプログラムソースコードは、機械語ではなくハードウェアに依存しない中間言語に変換されるため、実行環境をインストールすることで様々なハードウェアで動作可能である。
* Android向けアプリケーションの開発の中心となる言語でもある。
* オブジェクト指向的なプログラミングスタイルで記述されることが多く、そのための様々な用途別のフレームワーク（部品となる雛形やオブジェクトをまとめたもの）が提供されている。
* 標準的なフレームワークとしてJava EE（旧称：J2EE）が使いやすく、よく用いられる。また、Webアプリケーションソフト向けのフレームワークとしてはStrutsなどが有名である。
* 汎用的な機能をまとめて、再利用可能としたクラスとしてJava Beansがある。これはJavaBean仕様に従って作成されている。特にサーバーサイド向けのJava Beansとして、Enterprise JavaBeans（EJB）という仕様が定義されている。EJBを実行させるための動作環境は、EJBコンテナと呼ばれる。

1. Ｃ＃

* Microsoftが言語仕様の制定を行い発表した言語であり、主に.NET Frameworkを用いたアプリケーションの開発に使われる。
* 現在のWindowsのアプリケーション開発における主力言語である。
* 名前から、Ｃ＋＋を拡張した言語と思われることが多いが、実際の言語仕様としては基本的な文法以外はほぼＪａｖａの言語仕様を取り入れている（一部、Ｊａｖａでの問題点を解消している）。
* また、Ｃ、Ｃ＋＋にあったメモリ管理などのコーディングは基本的に不要となっており、プログラム開発の生産性も高くなっている。

1. Ｖｉｓｕａｌ Ｂａｓｉｃ

* Ｂａｓｉｃ言語から発展した言語で、Microsoftが手軽にWindowsアプリケーションができるように発表した言語である。
* 開発当初より、プログラムの開発環境（統合開発環境）と一緒に公開されており、最も早く公開されたＲＡＤツールの一つである。
* プログラミングスタイルとして、イベント駆動型（ボタンを押す、などのイベントに対して処理を定義する）の形を採用しており、フレームワーク的な開発（雛形があって、必要な処理を穴埋めしていくようなスタイル）に繋がるものである。
* Ｖｉｓｕａｌ　Ｂａｓｉｃ Ｖｅｒ６．０から２００２（Ｖｉｓｕａｌ Ｂａｓｉｃ．ＮＥＴ）にバージョンアップした際に、.NET Frameworkを使用するように大幅に言語仕様が見直されており、オブジェクト指向的なプログラミングの言語仕様も大幅に追加されている。

1. Ｖｉｓｕａｌ Ｂａｓｉｃ Ｆｏｒ Ａｐｐｌｉｃａｔｉｏｎ（ＶＢＡ）

* Ｏｆｆｉｃｅの各アプリケーションにおいて、処理を自動化するためのプログラミングが可能になった際に採用された言語である。
* ＡＣＣＥＳＳやＥＸＣＥＬの構成要素、機能はＶＢＡからはオブジェクトとして見えており、ＶＢＡからこれらを使うことでアプリケーションソフトを構築することができる。
* 言語仕様はＶｉｓｕａｌ Ｂａｓｉｃ Ｖｅｒ６．０のサブセットとなっており、統合開発環境としても同等のものが準備されている。
* 現在Ｖｉｓｕａｌ Ｂａｓｉｃの言語仕様が２００２年から.NET Framework対応になったにも関わらず、ＶＢＡの言語仕様はほとんど変化していない（.NET Framework対応は行われていない）ため、ＶＢＡと現在のＶｉｓｕａｌ Ｂａｓｉｃは大きく言語仕様が異なっており、ＶＢＡ習得者がＶｉｓｕａｌ Ｂａｓｉｃを習得するのは、新たにＪａｖａやＣ＃を学習するのと、ほとんど同じ程度の難易度、学習期間と考えてよい。

1. ＶＢＳｃｒｉｐｔ

* Microsoftの作成したスクリプト言語（簡易的なプログラミング言語）である。
* 当初はMicrosoft製のWebサーバ（ＩＩＳ）とブラウザ（Ineternet Explorer）を組み合わせた環境で、動きのあるWebページの作成の記述に用いられたが、その役割は、現在はJavaScriptに取って代わられている  
  （JavaScriptはＩＩＳ以外のWebサーバでも動作できる。VBScriptはＩＩＳのみ）。
* 現在は単体のスクリプトファイル（簡単な処理を行うためのプログラムを記述したテキストファイルで、それ自身で実行可能）の記述に用いられることが多い。
* 言語仕様はＶｉｓｕａｌ Ｂａｓｉｃ Ｖｅｒ６．０サブセットであるが、構文はほぼ同じであるものの、ＶＢＡよりもさらに言語仕様が削られている（組込み関数や手続きの大幅な削除、変数の型がVariantのみ、など）
* Microsoftは、.NET Framework対応のスクリプト言語としてPower Shellを開発しており、今後はこちらに移行していくと考えられているが、習得の平易さと過去の資産の蓄積により、VBScriptの需要はまだ高い。

1. ＪａｖａＳｃｒｉｐｔ

* スクリプト言語で、発表された当初はLiveScript と呼ばれていたが、Sun Microsystems社が、当時注目されていた言語の名前Javaの名前を用いて「JavaScript」と名付けたものであり、言語仕様としてはJavaとは全く異なるものである。
* 現在、ECMAScriptという名前で標準化が検討されているが、まだまとまっていない。
* 主にWebの記述言語であるHTMLとともに用いられ、HTMLでは不可能であった動的な表現を可能にしている。
* 記述を簡単するためのライブラリjQueryや、さらに機能を増強するためのフレームワークであるAngularJS、サーバ側での動作を記述するためのNode.jsなどが公開されており、より高度なアプリケーションの構築が可能である。
* ３層システムのプレゼンテーション層（ＵＩ設計）の担当エンジニア（いわゆる「フロントエンドエンジニア」）にとって、重要な言語である。

1. ＰＨＰ

* Ｗｅｂサーバ上で動くスクリプト言語であり、Ｗｅｂシステムの開発に用いられる。
* 文法が平易であり、その一方でオブジェクト指向のサポートや、豊富なフレームワークやライブラリが公開されており、本格的なＷｅｂアプリケーションが構築できることから、人気のある言語の一つである。
* 特によく使われるフレームワークであるＣａｋｅＰＨＰ、Ｌａｒａｖｅｌなどがあり、ともにＭＶＣモデルを前提としている。

1. Ｒｕｂｙ

* オブジェクト指向のスクリプト言語である。
* 言語仕様がシンプルで、少ない行数でプログラムを記述できる。
* オブジェクト指向は徹底しており、データ型はオブジェクト型のみである。
* ＭＶＣモデルのＷｅｂアプリケーションの構築用フレームワークであるRuby on Railsが用いられることが多い。
* 開発者が日本人であり、日本語のドキュメントが整備されていることから、日本人に人気がある。

1. Ｐｙｔｈｏｎ

* オブジェクト指向のスクリプト言語、汎用プログラミング言語である（両方の側面を持つ）。
* オブジェクト指向スクリプト言語であるRubyと比較されることが多いが、言語仕様としてはかなり差がある。
* 世界的に見れば、Rubyが日本で、特にRailsによりWebアプリケーションに用いられているのに対して、Pythonは公開されているライブラリが多く、また、Pythonの方は汎用のプログラム言語としての仕様も持っていることから、より広範囲の用途に使用されている（特に学術分野での数値計算が得意）。
* 現在の業務アプリケーション構築のトレンドとして、Webアプリケーションの需要が高いことと、日本人に人気が高いことから、日本ではRubyの方が需要が高い。
  1. フレームワークとライブラリ

現在、プログラム言語はそれ単独でアプリケーションを構築できることはほとんどなく、通常はアプリケーションの種類に応じたフレームワークやライブラリを利用することで初めて構築が可能となる。

従って、プログラム言語の学習と同じ、あるいはそれ以上にフレームワークやライブラリの習得が重要である。

フレームワークとライブラリの違いは、大雑把に言えば

* ライブラリ

プログラマーの作成プログラムが、必要とする機能に応じて、その機能を提供しているライブラリを呼び出して機能実現する。

* フレームワーク

フレームワーク側で、ある程度アプリケーションの雛形（例えばＭＶＣモデル）が作成されており、そこに処理したい内容のプログラムを埋め込んで行って、フレームワークに呼び出させて機能を実現する。

という違いがある。

プログラマーとして、ＩＴ業界の需要に答えていくためには、現在よく使われている（トレンドとなっている）フレームワークを常にウォッチし、習得し、構築経験を積むことが重要である。

* 1. コーディングツール（プログラム記述ツール）

プログラムはテキストで作られることから、それぞれの開発環境で使用可能なテキストエディタを使用することになる。

テキストエディタによっては、特定のプログラム言語のコーディングが便利になるような機能が搭載されているものもある（言語のキーワードの色が変わる、など）

ただし、統合開発環境（後述）では、通常プログラム記述用の画面（機能）を持っており、その機能を使ってコーディングを行う。

* 1. デザインツール

ＵＩのデザインを行うツールであるが、ＧＵＩでデザインできるものもあれば、画面の記述言語（WebであればHTML言語）で記述しなければならないこともある。

（その場合は、図形を書くことができるツールを使ってラフデザインを作成後、記述言語によるコーディングを行うことになる）

統合開発環境にはＧＵＩのデザインツールが内蔵されているものがある（Visual Studioなど）。

また、WebのＵＩに関しては専用のツールも存在する（ホームページビルダーなど）

* 1. デバッガ

プログラムのバグの調査に使用する。プログラムの動作の一時停止や、その時点での各変数の内容の表示の機能を持つ。

統合開発環境の他、一般的な開発環境（ＳＤＫと呼ばれるもの）には、コンパイラとともに装備されていることが多いが、一方で全く存在しないものも多い。

またデバッガ自身、コマンドベース（ＣＵＩ）のものもあればＧＵＩで操作できるものもある。

デバッガが存在しない場合には、変数の内容を画面に表示したり、実行を中止するプログラム文を一時的に挿入してデバッグを行っていく必要がある。

* 1. 統合開発環境

上記の①～⑤をすべて、あるいは一部備えているもの。

Ｖｉｓｕａｌ ＢａｓｉｃやＣ＃ではＶｉｓｕａｌ Ｓｔｕｄｉｏ、ＶＢＡではＶＢＥが使用される。

Ｊａｖａではいろいろなものが公開されているが、Ｅｃｌｉｐｓｅが使われることが多い。

1. 上流工程の重要性、設計のポイント
   * 1. 利益とスケジュールが決まる

この工程で初めて機能と開発フレームワークが定義され、工数の見積もりが可能となる。この見積もりにより、お客様の予算との関係から利益が、また工数見積もりと人員計画によりスケジュールが明確になる。

ただし、納期も予算も、この段階に至る前に決められていることが多く、見積もりの結果、プロジェクトが赤字、もしくは納期に間に合わない、という結果が予測されることがある。

この場合は、営業部門を通してお客様との交渉が必要になる。

納期については、動かせないことが多いので、機能について

* 機能を削る
* 追加費用を頂いた上で、納品を複数回に分割して、一部の機能を後日追加納品する

というような交渉が必要である。

* 政治的、営業的な理由などにより、見積もり結果を無視してプロジェクトが強行されたり、（「成功」するという結果ありきの）極端に楽観的な再見積もりを要求されることがあるが、そのような見積もりを受け入れれば、プロジェクトは「確実に」失敗するので、安易な受託は禁物である（失敗が判明した時点で責任を負わされ、大きな損害を被ることになる）。
  + 1. 「できないこと」の明確化

仕様書には、得てして機能が「できること」は書かれているが、「できないこと」が明記されていないことが多い。

お客様がシステムに実装されると考えている機能は

「機能仕様書で実装すると書かれた機能」＋「常識的に実装されているべき機能」

である。

この「常識的」の認識の違いにより、下流の工程になってからお客様より

「そのような機能が入っているのは（業務内容から）当然だ」

と指摘を受け、無償での機能追加を要求される、というトラブルになることが多い。

（上記のセリフは、非常によく聞く言葉である）

これを避けるためには、設計の際に以下の点について注意すべきである。

（当然、業務知識が要求される）

* 機能一覧の設計段階で、業務内容から推定される業務を考え得るだけ挙げておく。  
  その上で、ヒアリングでお客様から要求事項に挙がっていない項目があれば、実装の必要性の有無をお客様に確認する。
* 業務上必要となる可能性があっても、フレームワーク的に実装不可能である場合には、要件定義で「実装しない」ことを明確に記述する。
* 設計が進んだ段階でも、そのような未実装機能に気が付いた場合は、できるだけ速やかにお客様への確認と交渉を行う。
  + 1. 「丸投げしないこと」「提案型」のＳＥであること

設計の段階で、あまり詳細な仕様を決めてしまっても、後にお客様からその仕様の変更を求められて作り直しになる可能性がある。これは工数的に無駄である。

しかし、あまりに「常識的」な仕様の決定までお客様に丸投げすれば、

* 「常識的」な業務知識もない＝上流SEとして技術力がないとの評価を受ける
* お客様に余計な（設計を考えるという）負担をおかけすることになる。または、その負担を拒否されて、逆にこちらに仕様を丸投げされてしまい、結局詳細仕様を作成後に仕様の変更を要求され、工数の無駄が発生する、

というようなことが発生する。

仕様の最終決定までSE側で独断で決めてはならないが（必ずお客様の確認が必要）、業務知識上常識的な仕様は「たたき台」としてSE側から提案すべきである。

逆に、フレームワーク的に作りやすい形の仕様を提案することで、うまくこちらの考えている仕様にお客様を誘導することも可能になる。

なお、お客様のＩＴリテラシーの低さにより、

* 要求仕様が、システム化困難である、あるいはフレームワーク上無理がある
* 実際の使用時に、明らかに利便性の悪いものになる

と想定される場合がある。

（直接ご要望をお聞きした方ではなく、実際にシステムを使用されるお客様の現場での運用においてトラブルの発生が想定される場合も含む）

安直に、お客様のご要望を否定してはならないが、無条件に受け入れて、工程が進んだ後で仕様変更を求められるようなリスクが考えられるときには、別案を提案することも考慮する必要がある。  
ただし、このような場合には、お客様の要望のベースとなっている理由、考え方を理解した上で、注意深く対応する必要がある  
（安易なご要望の否定は、お客様との信頼関係の破綻につながるため）。

* + 1. お客様の宿題事項の回答がない

宿題事項（アクションアイテム＝AI）について、SE側の宿題事項を期限までに対応するのは常識であるが、  
お客様の宿題事項について、合意した期限までに回答が無い場合がある。

このことに対してSE側で何もアクションを採らなければ、お客様は

「SEはこちらから回答しなくても、問題点について自己解決できた」

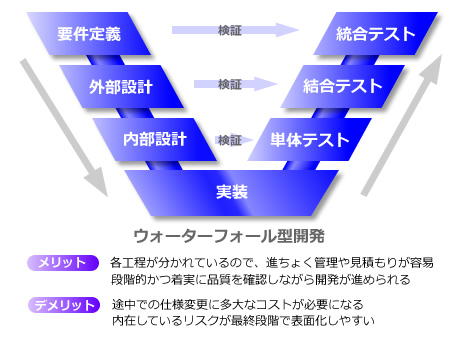
と認識する。責任がお客様側にあるとは絶対に考えない。

もし回答がないことで仕様の確定が遅れたとしても、それはSE側に責任があるとされる。

従って、期限内に回答が無い場合には、回答を得られるまで催促を続ける必要がある。

* + 1. テスト項目を設計時に作成すること

ウォーターフォールのモデルでは、設計とテスト項目は以下の関係にある。

[](https://www.google.co.jp/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwjjrJmg9IfTAhWBUbwKHXQfAtEQjRwIBw&url=https://japan.zdnet.com/article/35003090/&psig=AFQjCNHUYgBqyPoQp2nBYXDHfLcXDcp8UA&ust=149129585811057)

例えば、外部設計の際には業務の流れを想定して設計を行うので、その「業務の流れ」はそのまま結合（機能）テストのシナリオとして活用できる。

従って、設計段階で（正式版のテスト仕様書ではなくても）テスト項目をリストアップしておくことは、テスト工程での効率化につながる。

1. 要求分析
2. 打合せの一般的な注意事項

ヒアリングに限らず、検討（まとめ、摺り合わせ）会議、レビュー会議などの打ち合わせを実施する際には、以下の点に注意する。

* 目的、ゴールの明確化
* 出席者全員の都合の調整
* レジュメの作成、時間配分の設定
* レビューであれば、資料の事前配布
* 合意事項の確認
* 速やかな議事録作成
* 会議で未決となった事項のアクションアイテムとしての管理

1. 業務知識について

事前に、お客様の業務知識の理解しておくことが必要。

（お客様のおっしゃることを理解できるようにするために）

一般的な業界知識だけでなく、お客様独自の社内ルール等については、別途事前ヒアリング（お客様によるレクチャー）の開催に協力いただいて、システムに関係する業務の流れは把握しておくこと。

1. ヒアリングで聞き出すこと

お客様からの要求事項の聞き出しに関しては、５W1Hの観点から行う。

●いつ

　どこで

●だれが

●なにを

　なぜ

●どのように

特に●を押さえておく必要がある。

1. ●いつ

タイミングのこと。業務のアクティビティ（具体的な作業）について

「時間（時刻、回数）」、「順番（作業間の）」、「条件（やる、やらないの条件）」

を確認する。

1. どこで

システムの構造にかかわるような特殊環境下（例えば、全国数万店以上の店舗から同時使用する、など）でもなければ、通常は聞きだす必要ないことが多い。

1. ●だれが

システムを使用する担当者。担当者の所属組織の構造、あるいは権限などにより、システムの機能に考慮が必要になることがある。

1. ●なにを

入出力のこと。入力するもの（お客様の打ち込みデータ、他システムの出力など）、出力するもの（画面表示、帳票印刷、DBへのレコード追加、更新など）、

1. なぜ

特に、お客様の経営上の「戦略的な要件（人員削減、ペーパーレスなど）」があって、そのために特にシステムに対する要求事項があれば聞き出しておく。

1. ●どのように

手順内容。「なにを」に書かれたものをどう処理して、何が出来上がるのか、を明らかにする。

1. ヒアリングへの対応姿勢
2. 聞き手に回ること

お客様発言中に口を挟まない。疑問点はメモして、最後にまとめて質問。ただし、継続が難しい場合には中断を求める。

1. 実装方式を考えない

この時点では、どうやって実現するか（ソフトウェア的な技術的要素）を考えない（これは、後の工程で考えること）。

ただし、開発フレームワークや開発環境についての指定があり、お客様の要求がその環境で実現不能と（リーダーが）判断される場合は、その時点でリーダーが意見することは必要。

* その場合、「できない」で終わらせず、指定されたフレームワークでできる範囲での妥協点を探ること。

1. 要件定義
2. 要件定義とは

×「要求を明らかにする作業」→それは当たり前（単なる要求確認作業）

○「要求の解決策を考える作業」

すなわち、解決するための「手段」を明確化すること。

（手段＝入出力媒体、サブシステムへの分割、アクティビティの定義など）

* プロジェクトを予定通り完遂させるために大変重要な工程である。  
  この工程には以下のようなリスクがある。
  + - 仕様の掘り下げの甘さ

下流工程の設計で確定しなければならず、遅れが蓄積する。

* + - ＳＥの業務知識の欠如

その業界にとって、当たり前の常識が、ＳＥに認識されておらず、結果として仕様漏れ、誤解＝認識のズレが生じる。

* + - ＳＥのフレームワーク等の技術知識不足

実装不能、あるいは非効率な仕様を設計してしまう

これらの原因により、仕様修正（変更）が生じて、基本設計段階まで工程後戻りが発生すると、基本設計自身のやり直しが発生し大幅な納期遅れや費用増大をもたらす。  
失敗プロジェクト（納期遅れ、赤字）の主な原因はこの工程の設計不良にある場合が多い。

1. お客様の要求事項を機能要件と非機能要件に分けること

それぞれの要件については、前述の通り。

非機能要件については、フレームワークの限界や、リソース制約などにより制限を受けることがあり（性能など）、この点も明確化しておく。

1. 仕様書の書式の標準化

IEEE 830や、ＳＩ会社独自で書式が標準化されている。これに従うことで、設計漏れを、ある程度防止することができる。

1. 細かい点まで注意して分析、設計する（当たり前と思って省略しない）

お客様との認識の齟齬をできるだけ排除するため、業務の作業内容を細かく分析するとともに、自分自身の「常識」に捕らわれて設計書の記述を省略たり一般的な表現（曖昧な表現）を使わないように心がける。

1. 疑問点はできるだけ早く解決

お客様の「言い忘れ」も含めて、気づいたときにはＱ＆Ａ票のできるだけ早い発行する。その際には、回答期限を明記すること

* ＳＥの想定が崩れて、やり直しが発生する可能性があり早急に解消する必要がある。

1. 要件定義のゴール

要求の解決策が具体的に明確になっており（＝文書化されている）、それぞれの解決策について具体的に設計を開始できる状態。

* この段階で初めて費用見積もり、スケジュールが明確化できるのだが、実際には、事前に費用、納期とも決められていることがある。  
  その場合、現在の見積もりとの差が吸収不能なほど乖離した状態のまま工程を進めれば、プロジェクトは１００％失敗するため、  
  この時点での開発機能の簡略化、実装機能の一部後回し、スケジュール見直しなどの調整が必要である。

1. 成果物
2. 機能一覧

システムを分割した各サブシステムのアクティビティ（登録、検索、印刷、など）をすべて列挙する。  
単純な書式の表であるが、漏れがあれば以下の仕様書がすべて誤りとなるので、要求分析に基づいてすべて慎重に漏れなく抽出することが重要である。

1. I/O関連図

各機能での入出力媒体（データ、ユーザ入力、帳票など）を明確化する。

1. 業務フロー図

システムにおける、業務に対応する機能の流れ（処理順序）を定義し、そこに上記のＩ／Ｏ関連図で明確化した入出力媒体を組み合わせて、（利用者から見える）システム全体の構造と業務の流れを明確化する。

1. 基本設計（外部設計）
2. ここでやること

要件定義の工程までで、これから作るシステムで

「何をやるか（＝どういう機能を持つか）」

を定義したので、基本設計の工程では、それを「どうやって実現するか」を設計する。

すなわち

「何をするか？⇒どうやってするか？」の変換

である。

情報システムとは

入力⇒処理⇒出力

であるので、「どうやって」の設計とは

入出力の具体化＝入出力の具体的な媒体（画面、帳票、データベース）のデザイン

処理内容の具体化＝モジュール構成と処理内容の定義

である。

1. フレームワーク（実行環境）

要件定義で定義した「何を」に雛形はないが（対象業務によって様々）、

基本設計の「どうやって」は、その実現手段（実装方法）として、

構築手段のフレームワーク

実行環境

が雛形となる。

それによって、前述の「入出力媒体の方法」が自動的に決まってしまう場合が多い。

* 例えば、ACCESSの場合だと、

画面（ＵＩ＝ユーザインタフェース）⇒フォームとコントロール

帳票⇒レポートやEXCELファイル出力

データ⇒ACCESSのテーブルや外部データベース、データファイル

など。

また、処理方法についても、使用するフレームワークによってある程度プログラムの作り方や、モジュールの構造が決まってしまう。

* 例えば、ACCESSの場合だと、

イベントプロシージャ、標準モジュール、クラス、

EXCELやFileSystemObjectなどのオブジェクト

* 基本設計の段階で行うのは、

これらの手段を使ってシステムが「実現可能であるかどうか」の確認

までである。

具体的なこれらの手段の使用方法の設計については、主に下流工程での作業となる。

1. 入出力をまとめる

入出力媒体について、共通化できるものをまとめていき、システム規模を抑えていく。

* 例えば、

可能なら、機能の起動を１つのボタンで実現できるようにする（そうした方が利用者の利便性が上がる場合）

データベースのテーブルを重複した情報がないようにまとめる（正規化を行う。その後、リレーションシップを考慮の上、ＥＲ図にまとめる）。

1. 表示項目とレイアウト設計について

ＵＩはお客様に直接見える部分であるので、

最も要求、仕様変更の出やすい部分であり、あまり最初から完成形に近いものを作成せずに、

1. レイアウトの概略

フォーム、コントロールの種類、大体の位置（類似項目を近接して配置）

1. 表示する内容の項目一覧

フォーム、リスト、コンボボックスなどのメニューごとに分けて

を分けた資料として提示し、そこで具体的な配置の確認を行う。

（その後、改めて確定版にて最終的なレイアウト確認を行う）

* これは、あまり詳細なレイアウトを作っても、表示データやコントロールの変更などにより、簡単に作り直しとなってしまうため。  
  ただし、ある程度お客様にイメージが伝わるように表現を工夫する。  
  （表示項目をグループ分けし、グループの配置を示すなど）
* お客様に丸投げしないこと。  
  概要がなければ、お客様もイメージが湧かない⇒お客様から丸投げ返される⇒詳細レイアウト設計後にクレーム発生、というトラブルとなる。  
  また、表示項目を決められないことで、上流スキル（業務知識）を疑われる。

1. 処理の設計にあたって

下流工程では

「データの変換＝入力をどう加工して出力を得るか」

の具体的手段が明確（実現可能）でなければ、以降の設計ができない。

* + 具体的な言語、ないしフレームワークの一つ一つの機能＝基本操作、およびその集合に対応させることができること

すなわち、要件定義において、

「動詞」で表現されている部分（～を「どうする」の部分）

が、既存のフレームワークやプログラミング手法に落とし込めなければ、他のフレームワークやライブラリの流用、処理のさらなる細分化などを実施する。

1. 処理の機能分割の手順
2. 骨格になるフレームワークを決める

ACCESSだと、中核となる各フォーム、アクティビティに直結するボタン類など

1. フレームワークをカスタマイズする

利用者の使い勝手、コントロールの限界などを考慮

1. 入出力のフレームワークを決める

ACCESSで不足する場合、EXCELやFileSystemObjectのオブジェクトの使用、ADOやDAOを使用したクエリー発行などを採用する

1. 処理の共通部分を決める

アクティビティ＝新規登録、更新操作などで共通となる処理を切り出す

1. 一つのアクティビティに対して、3の入出力フレームワークをどのタイミングで使い、4の共通部分をどのように使うかを決めて、アクティビティのフレームワーク、処理構造を決める。
2. バグが混入したり設計不能となる場合

この段階でのこのようなトラブルは以下のようなことが主な原因である。

1. アクティビティが基本操作に落ちない
2. 要求仕様書の誤記
3. 要求仕様書の解釈の誤り
4. 機能分割の失敗、処理の欠落や間違い
5. 共通部分の切り出しの誤り（共通にならないアクティビティを誤ってまとめた）
6. 詳細設計者とのコミュニケーション上のミス

上記を避けるには、上記の十分なレビューとともに、

* ドキュメントから意味の曖昧な「単語」を排除する
* お客様の業界用語（特に「動詞」）は表現不能な場合を除いて避ける

ことが重要である。

下流工程担当者が誤解しないように、なるべく、平易な言葉（誰が聞いてもイメージできる言葉）の選択が重要である。

1. 成果物

* システム項目一覧

入出力するデータ項目の一覧

* 画面レイアウト図

前述のように、概略⇒詳細と２段構えで設計する

* 画面遷移図

画面の状態が変化や別画面に推移する流れを示したもの

* 帳票レイアウト図

画面レイアウト図に同じ。ただしお客様から書式指定がある場合を除く

* イベント一覧

イベントプロシージャへの機能割り当て

* 更新・処理仕様書

システム項目とデータベースの関係図。機能間でまたがって使用するデータベースへのアクセスの確認の際に重要となる設計書

以上