システム開発概論

ver2.01

1　よりよいシステムを開発するために

よりよいシステムを開発するために、システムの信頼性を評価する 5 つの指標があります。

- 信頼性（**R**eliability）
- 可用性（**A**vailability）
- 保守性（**S**erviceability）
- 保全性（**I**ntegrity）
- 安全性（**S**ecurity）

この 5 つの指標は、英語の頭文字を並べて「RASIS」（レイシスあるいはラシスと呼ぶ）といいます。

信頼性（Reliability）

信頼性とは、不具合や障害による故障が起きにくく安定したシステムであることを示す度合いです。「故障せずに長時間稼働するシステムは信頼できる」ということです。

具体的な指標としては、システムが安定稼働し続ける平均時間である「平均故障間隔」があります。

可用性（Availability）

可用性とは、システムが継続して稼働できる度合いです。「正常稼働している割合が高いシステムは安定している」ということです。

具体的な指標としては、期待される稼働時間に対する実際の稼働時間の割合である「稼働率」があります。

「信頼性」と「可用性」は似て非なるものです。

- 信頼性：どれだけ壊れにくいか？
- 可用性：普通に使える状態をどれだけ維持しているか？

保守性（Serviceability）

保守性とは、障害の発生をどれだけ早く発見し、修復ができるかなど保守のしやすさを示す度合いです。「修復に要する時間が短いシステムは保守性が高い」ということです。

具体的な指標としては、システムを修復するのにかかる平均時間である「平均修理時間」があります。

保全性（Integrity）

保全性とは、障害や誤操作などによってデータの破壊や喪失、不整合などの起こりにくさの度合いです。「障害が誤操作などでもデータは壊れない」ということです。

安全性（Security）

安全性とは、不正利用に対してシステムが安全に保護されている度合いです。「セキュリティ対策をしっかりやっているシステムは安全」ということです。

実際のシステム開発では、よりよいシステムを開発するために、RASIS にも配慮します。

2 開発手法

2.1 前提

システム開発は次の２つに大きく分けられます。

社内開発：社内システムもしくは自社サービスのために社内で行う開発

受託開発：他企業のためのシステム開発

本テキスト内で**「顧客」という言葉を使っていますが、受託開発では発注元、社内開発ではプロダクトオーナーやビジネスサイドがその対象に当たると考えてください。**

明確な区分が必要な場合には、「受託開発発注元」「サービス利用ユーザ」などの言葉を利用します。

2.2 開発手法

開発のゴールは顧客が満足するプロダクトを完成させることですが、その過程では様々な方法が用いられます。その方法を開発手法や開発プロセスと呼びます。

開発手法には様々ありますが、代表的な２つの方法論について見ていきます。

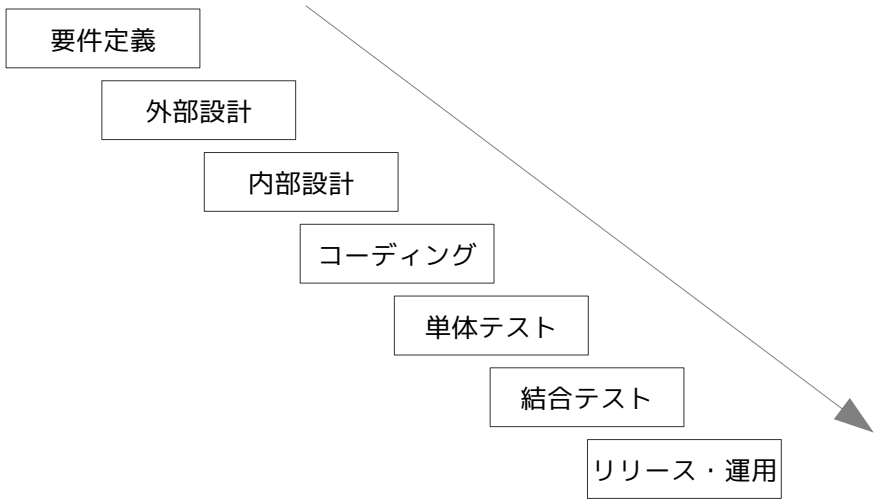
2.3 ウォーターフォール開発プロセス

システム開発の手法で、最も多く利用されているのが「ウォーターフォール開発プロセス」です。本研修でもメインとして、この開発プロセスを確認していきます。

ウォーターフォール開発プロセスとは

開発を複数の工程に分け各工程の終了時に成果物を作成することにより、各工程における品質の確保を図る開発手法です。

「水が流れ落ちる」様に工程が進むことから名付けられており、上流工程から下流工程まで流れる様に開発が行われます。



ウォーターフォール開発プロセスの各工程については後述します。

ウォーターフォール開発プロセスのメリット

メリットをまとめると以下の様になります。

計画の立てやすさ

上流工程から要求機能を詳細に落とし込む手順を踏んでゆくため、事前に今後必要となる事項を想定しながら開発できます。

進捗管理のしやすさ

全体を把握した上で工程別・タスク別に管理可能なことから、プロジェクト全体や、各開発要員の進捗管理を行いやすくなります。

成果物ベースでの開発

ドキュメントなど成果物がある状態で開発を行うため、どのような開発者でも仕様書を読めば開発する事ができます。

ウォーターフォール開発プロセスのデメリット

反対にデメリットは次のようなものです。

上流工程でしか要件定義できない

要件定義や基本設計フェーズでは、顧客との仕様検討を行いシステム要件を詰めますが、仕様

検討したのは「過去」の時点のものであるため、顧客やサービス利用ユーザの変化要求スピードが速い場合、仕様変更が発生する可能性が高くなります。

仕様変更時の影響

ウォーターフォール開発プロセスでは、前工程の成果物をベースに開発を行います。そのため、仕様変更や設計ミスなどによる成果物の変更が発生した場合、プログラムの修正はもとより、前行程の成果物も含めての修正が発生するケースがあります。

成果物管理の稼働負荷

成果物ありきで開発が進められるため、ドキュメントなどの成果物の作成・修正稼働に負荷がかかります。

ドキュメントによる品質・工程管理

ウォーターフォール開発プロセスにおける成果物（ドキュメント）管理について、もう少し詳しく解説します。

ウォーターフォール開発プロセスはドキュメントの作成量が多いことで有名です。各開発工程の最後には成果物として設計書やテスト結果報告書などさまざまなドキュメントを顧客に提示します。

提示したドキュメントを顧客が承認することによって、開発側は次の工程に進むことができます。ウォーターフォール開発プロセスは前工程の成果物ありきで進められるので、ドキュメントがないと先々の工程に進みにくい性質を持ちます。

実際には納期が短い場合、ドキュメントは後から提出して、できるだけ早く工程を進めて行くというケースもあります。

特に社内開発の場合には現実的にドキュメントは後回しという例もよく見られます。

大規模なシステムほど、ドキュメント量が膨大な量になる傾向があり、その作成や管理稼働に大きな負荷がかかります。また、工程は進んだものの、後になって要求仕様が変更された場合には多くのドキュメントを修正する必要があります。

これらの管理資料の作成だけでも、多くの稼働がかかってしまう場合があります。反面、計画の立てやすさや進捗管理のしやすさがあり、今でもウォーターフォル開発プロセスが主流の開発手法になっています。

代表的な管理資料としては WBS（Work Breakdown Structure）があります。開発プロジェクトを細かい作業項目に分割し、誰が、何を、いつまでに対応するか、進捗率はどうかなど、開発計画や実績を把握できる資料を作成してプロジェクト管理を行います。

特に受託開発で利用される理由

ウォーターフォール開発プロセスは、特に受託開発ではなくてはならない開発手法と言われています。

多くのエンジニアが慣れているとともに、「見積りが立てやすい」、「成果物と一緒に費用を請求しやすい」のがその理由です。

工程に分かれてドキュメントも制作するため、見積りが立てやすいのは明らかです。

また、大規模な開発になればなるほど開発期間は長期にわたりますので、納品してから請求となると、運転資金が調達できません。そこで採られる方法が「分割検収」という方法が取られます。

全体の見積り金額から、要件定義の成果物を納品したら○万円請求、基本設計から詳細設計までの成果物を納品したら○万円請求、というように、フェーズの終了毎に請求を行います。企業側は開発者の人件費を払う必要があり、資金繰りもしやすいこの手法が採られるケースが多いのです。

社内開発でもウォーターフォールプロセスは取られますが、後述するアジャイル開発プロセスが取られることも多くなってきています。

しかし、アジャイルで開発する場合でも、ウォーターフォールが基本となっていることには代わりがありませんし、作成するドキュメントの知識は必要です。

2.4 アジャイル開発プロセス

ウォーターフォールのデメリットに対応するために生まれた開発手法がアジャイル開発プロセスです。

アジャイル開発プロセスとは

アジャイル（agile）とは、和訳すると「機敏な、俊敏な」という意味で、その名のとおり機敏な対応を可能とする開発手法です。

開発中に仕様変更が発生した場合、変更内容を全システムに反映させないといけませんが、アジャイル開発プロセスでは以下の特徴により開発中の仕様変更にも柔軟に対応することができます。

(1)顧客とエンジニアの共同開発チームを編成

システム開発を行う際、顧客と共同の開発チームを編成します。

チーム内に顧客がいることにより、顧客ニーズに迅速に対応できる体制となり、変更が発生してもエンジニアに変更内容が伝わりやすくなります。

(2)開発範囲を分割

開発対象範囲全体から開発対象となる機能に優先順位をつけ、それらを 1〜4 週間単位に開発可能な単位に分割します。

例えば開発対象となる機能が機能 1〜機能 4 まで 4 種類あった場合、

優先度 A：機能 4（マスタ管理機能）
優先度 B：機能 3（顧客管理機能）
優先度 C：機能 1（案件管理機能）
優先度 D：機能 2（入出金管理機能）

といった様に優先順位をつけて、それぞれを 1〜4 週間など、定められた期間で開発するのが特徴です。

(3)開発業務

開発は優先順位に従って進められます。
上記の例では機能 4（マスタ管理機能）から開発を始め、定められた期間内に要件定義、設

計、コーディング、テスト、リリースまですべて行います。

ウォーターフォール開発プロセスでは最初に全機能の要件定義を行うのが特徴ですが、アジャイル開発プロセスでは定められた期間に要件定義からリリースまで実施します。

ただし、開発規模が小さい場合は、全機能をプログラムして、一気に要件定義 からリリースまで行うケースもあります。

その場合、リリースするプログラムはプロトタイプや初期バージョン的な扱いになり、顧客の全要求を満たしているとは言えない場合が多くあります。

(4)リリース後の検討

リリースしたプログラムを顧客と共に検証し次に行うべき対応を検討します。

この検証を行うことにより、顧客からのフィードバックを素早く受けることが可能になり、迅速な仕様変更にも対応できるのです。

アジャイル開発プロセスは(2)～(4)を優先順位で定められた順番に繰り返し開発することが特徴であり、これを「イテレーション（反復）」と言います。

イテレーションを実施することにより、顧客からのフィードバックが反映されやすく、柔軟な仕様変更にも対応できる開発を行う事ができるようになります。

多くの種類が存在するアジャイル開発プロセス

端的にアジャイル開発プロセスといっても、実際にはさまざまな開発手法があります。

代表的な手法としては、次のようなものが挙げられます。

- エクストリーム・プログラミング（Extreme Programming、 XP）
- スクラム（Scrum）
- フィーチャー駆動型開発（Feature Driven Development、 FDD）
- クリスタルファミリー（Crystal Family）
- リーンソフトウェア開発（Lean Software Development）
- 適応的ソフトウェア開発（Adaptive Software Development、 ASD）

それぞれで重視する内容や細かい点は異なりますが、基本的には以下のように 3 つ共通点があります。

1. イテレーションによる柔軟な変化に対応可能
2. 短期間での開発が可能
3. 開発範囲を分割し、小さな単位で開発を行う

そしてもうひとつ、アジャイル開発プロセスにおいて共通しているのはプロジェクトメンバー間のコミュニケーションを重視する点です。

制限や制約などにより開発を進めるのではなく、チームメンバー全員の相互作用を活性化させることにより、システム開発を高効率、高品質、低リスクで進めることが重要です。

前述のように、チームメンバーにはユーザも含まれています。ユーザからのフィードバックやミーティングなどといったコミュニケーションを通じて、ユーザが望むシステムを開発しやすいという特徴があるのです。

2.5 開発手法の選び方

開発手法を選ぶことが目的ではなく、開発をスムーズに行うこと、もっと先を見据えれば顧客のニーズを満たすことを目的に開発手法を選定します。

大切なのは、「顧客、案件、納期、チーム等を考慮し、最適な方法」を選択することです。

外部からの開発案件であればウォーターフォールが適している、内部で完成形を探りながらシステムを開発するのであればアジャイルがすぐれているなど、ケースバイケースで開発手法は選択していきます。

日本ではウォーターフォール開発プロセスがまだ主流ですが、それだけでは対応できない状況になりつつあります。顧客のスピード感が非常に速くなり、変化する経済情勢に柔軟に対応できるようなシステムの変更を行うことができるアジャイル開発プロセスは魅力的な開発手法になってきています。

2.6 ワーク：アジャイル開発について

アジャイル開発の手法「スクラム」「XP」「FDD」について調べてディスカッションをしてください。

指定されたチームは発表を行ってもらいます。（A チームはスクラム、B チームは XP、C チームは FDD について前に出て説明をしてください、など）

3 工数の見積り

プロジェクトを進めるのは多くの場合、営利組織です。そのため、プロジェクトを進める上で重要なのが、「かかる時間」と「人数」それに伴う「コスト」を事前に把握することです。

特に受託開発を行う際など、ビジネスには見積りが必要です。使うのは工数という考え方です。

3.1 工数

プロジェクトの規模やコストを見積る上で、一般的に用いられるのが工数という概念です。ある作業を行うのに必要とされる「人数 x 時間」のことを指します。

工数は、日単位であれば「人日」(MD = man day)、月単位であれば「人月」(MM = man month)という単位で表します。例えば、プロジェクトの完了に１人で２ヶ月かかるとすると、２人月となります。

例） ５人が３ヶ月間作業する場合

５人x３ヶ月 = １５人月

工数は通常、1 日を 8 時間、1 カ月を営業日となる 20 日間と仮定して算出することが多くあります。そのため、「1 人月」は「20 人日」と等しいということになります。

人月工数という言葉がよく使われますが、人月で計算した工数を意味します。また、「１人のエンジニアが１カ月働く場合にかかるコスト」、つまり「1 人月あたりの費用」のことを「人月単価」と呼びます。

また、少なくなってきていますが、プログラムのステップ数（行数）でプロジェクトの規模を測ることもあります。１キロステップ（KS）などと規模を見積る形になります。わかりやすい見積りの手法ですが、デメリットもあります。例えば、オブジェクト指向を積極的に利用しているプログラムや部品の再利用化を行っているプログラムでは見積りが少なくなってしまう現象が起きます。200〜1000 ステップ/1 人日で捉えられます。

3.2 開発工数の見積り

新規プロジェクトに必要な人月を正確に予測するのは、現実的ではありません。新規のプロダクトであること、メンバーの力量により誤差が出ることから不正確なものになります。しかし大幅にずれがあると、企業が損失を負ったり、反対に見積りが高くなり案件自体が受注できないことになります。

そのため、プロジェクトの開発工数を見積るためには、過去のシステム開発経験を土台にして「予測する」ことが大切になります。

例えば、「ユーザのログインシステムには何人月かかった経験があるため、今回も同じくらいの時間がかかるだろう」と予測して、見積りを立てていきます。企業に蓄積されたデータや経験により、見積りの精度は徐々に上がっていきます。

また、「プログラムを書く工数」だけに限らず、「設計書を作成する工数」や「テストを実施する工数」を見積る際にも同じように工数を計算することができます。

過去に経験がないプロジェクト、要するにものさしになるものがないプロジェクトでは見積りに大きな誤差が生じます。特に大きいプロジェクトになればなるほど誤差が大きくなります。そのため、経験蓄積の多い大手企業がプロジェクトマネージャーの役割を果たすことが多くなります。

3.3 工数を見積るときの注意

工数を見積る時には、メンバーの人数だけではなく、管理工数も見積る必要があります。

一般的に作業に関わる人間が増えれば増えるほど、要員を管理する工数は増える傾向にあります。コミュニケーションの齟齬や進捗管理に使う時間が増えるためです。

そのため、多くのメンバーが活動するプロジェクトでは、人数に応じて管理工数を上乗せする必要があります。通常、管理工数は開発工数の 1 割〜2 割程度とすることが多いです。

4 ウォーターフォールと成果物

ウォーターフォール開発プロセス（以下、ウォーターフォール）を使って今回の開発演習は進めます。

このモデルは、以下の工程(フェーズ)からなります。

- 基本計画: システムの目標や実現可能性の調査·分析（開発するかどうかを決める）
- メンバーの選定と役割の確定
- 要件定義: システムの利用者がシステムに求めることを定義
- 外部設計: システムの利用者の立場から見た設計
- 内部設計: システム内部(開発者の立場)から見た設計
- コーディング（プログラミング）: モジュールをプログラム言語で記述し単体テスト
- テスト: 動作検証
- リリース・運用·保守: システムを動かして業務を遂行

4.1 メンバーの選定と役割

プロジェクトを成功に導けるメンバーを選定します。優秀なエンジニアを集めればいい訳ではなく、プロジェクトの予算や納期、対応技術の範囲などをもとにプロジェクトチームを作っていきます。

プロジェクトのメンバーを決めると同時に、それぞれの役割を決めます。与えられた役割によって、自身の持ち場に責任を持ちます。

人数が少なくとも、必ず必要なのがプロジェクトマネージャー（PM）と呼ばれるプロジェクトの管理者です。自身がコーディングするのではなく、進捗管理や工数管理などプロジェクトが円滑に進むように管理する役割です。また、顧客とのやりとりやプロダクト変更に対する見積り作成など、プロジェクトチームの表に立って外部とコミュニケーションを取ります。

PM の役割は、チームメンバーが動きやすいような状態を作り出すことと言っていいでしょう。

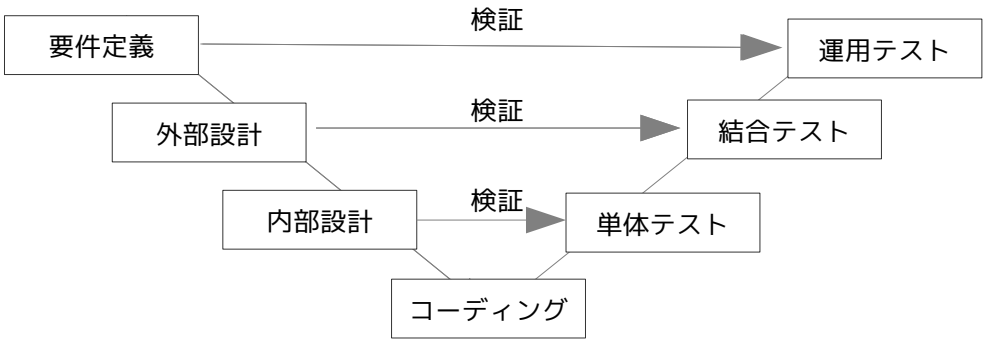
その他の役割は様々あり、開発規模や企業によって大きく異なります。

メンバーの役割例

略称	正式名	役割
PM	プロジェクトマネージャー	プロジェクトを取りまとめる
TL	テクニカルリーダー	技術面でチームのスキルアップを図る
DL	ドキュメントリーダー	日々、作成されるドキュメントや納品物を管理する

4.2 V字モデル

ウォーターフォールの「コーディング（プログラミング）」工程を折り返しとして、左側に設計、右側にテスト（動作検証）を配置した相関図が下記のV字モデルです。



ウォーターフォールではこの図を意識しながら開発・テストを行っていきます。

4.3 要件定義

成果物： 要件定義書

ウォーターフォールにおける最上流工程です。

開発するシステム全体の機能を顧客との打ち合わせを重ねて具体化し、開発するシステムの機能、目的、対象範囲を決定します。

システム開発を依頼するということは、「業務をもっと円滑にしたい」「ユーザ数を増やしたい」など、業務改善のために何らかの対策をしたいという目的があるということです。

その目的の認識がズレたまま開発が進むと下流工程に進むに従い、大きな認識違いが発生します。「どのようなシステムにしたいか」「システム化の対象範囲やシステム導入によって業務

がどう変わるか」、システムと業務の具体的な分析が必要となるフェーズです。

大きく分けて、機能要件と非機能要件があります。

機能要件

実装すべき機能に関する要件のことを機能要件と言います。

「何が必要なのか」、「何ができないと困るのか」など、そのシステムが必ず満たすべき要件のことを指します。「ユーザがログイン・ログアウトできる」「30 日間ログインがないユーザに通知を行う」などが機能要件です。

機能要件をどこまで用意した上で開発を行っていくかは、開発の規模やスケジュール、予算によって変わります。また、新規作成かリプレイスかによっても変わります。フォーマットも様々あり、プロジェクトごとに使用するものは変化します。

ポイントとなるのは、「外部設計をしていくのに、必要十分な情報が揃っているか」です。ウォーターフォールでは次のプロセスを行った際に、作業が無駄にならないだけの十分な情報があることが重要です。

非機能要件

主目的（機能要件）以外の要件は非機能要件と言います。

「応答の速度は」「セキュリティの対応程度は」「拡張性は」などが非機能要件と言えます。

この非機能要件については、発注者自身も意識していない場合や、暗黙の了解事項として期待している場合があります。そのため、トラブルが起こりやすい部分でもあります。開発者と発注者間で入念な話し合いをして、顧客がそのシステムに期待している、求めている品質を明らかにし、満たす必要があります。

下記のような項目を確認することが大切です。

- 可用性： いつでも使えるか、どれだけ安定感があるか
- 性能/拡張性： どれだけ快適に使えるか、利用者が増えても大丈夫か
- 運用/保守性： アフターサービスはきっちりとされているか
- 移行性： サーバーの引っ越しや、他サービスへの乗り換えは簡単にできるのか
- セキュリティ： ウイルス対策など、セキュリティ対策がしっかりされているか
- システム環境/エコロジー： そのモノを設置する環境は適切か？、環境保護に役立っているか

下記に IPA が発行している非機能要求グレードのチェックシートです。開発演習では時間の問題もあり、すべてを確認する必要はありませんが、参考のため一度ダウンロードして確認してください。

http://s.linuxacademy.ne.jp/java_master_non-functional

システム開発失敗の大半は要件定義の問題

要件定義は「一見簡単なようだが、実は難しい」というエンジニア間での共通認識があります「システム開発の失敗の原因のうち要件定義の割合」を答えるアンケート結果では、7 割近くの担当者が「５割以上が要件定義の問題だった」と回答しています。

なぜ要件定義が重要になるのか。要件が間違って伝わると、すべての工程がやり直しになり、実際にはリカバリーは困難になるからです。そのためしっかりと要件を固められるようにヒアリングを重ねるのですが、業務フローがすべてわかっているわけではないエンジニアが、顧客との認識をすべて一致させていくのは困難な作業です。

ポイントは「なぜそれを実施したいのか」を明確にしていくことです。ビジネス全般に言えますが、大切なのは常に目的意識です。目的が明確であれば、「当然そうするべきだろう」という前提を中心に話を進めることができます。もし、認識のずれがあっても小さい修正で済むことが多くなります。

開発を行う「目的」や「ゴール」を大切にしましょう。

参考：要求定義とは

要求定義は要件定義の前に行われる「〜がしたい」という顧客の要望をドキュメントにまとめたものです。

要求定義と要件定義が同時に行われることもありますし、別に明確に分けて要求をまとめる場合もあります。

本研修内ではまとめて要件定義としていますが、下記のように認識しておくと良いでしょう。

・要求定義：「がしたい」という希望 / ビジネスサイドで何がしたいかまとめたもの

・要件定義：「が必要」という仕様書 / システムが何をしなければいけないかの記述

ユースケース

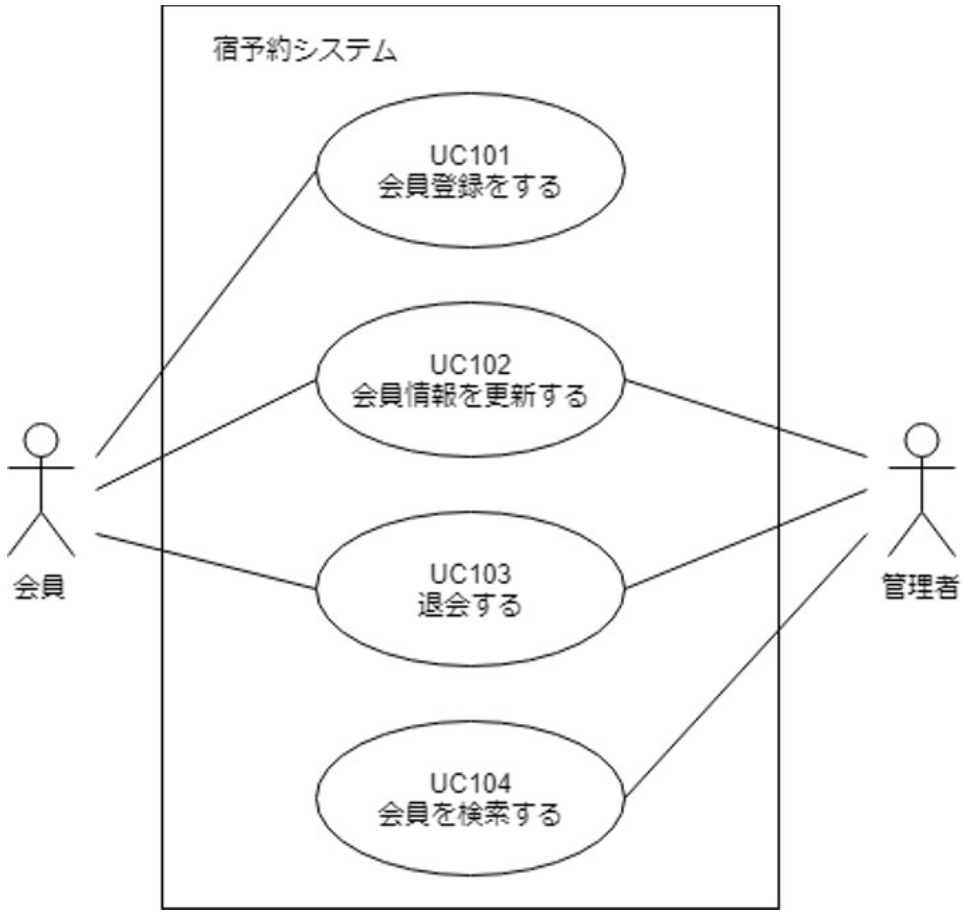
ユースケースとは利用者目線で「システムがどのように利用されるか」を表したものです。機能要件をまとめるのに利用されます。

ユースケースを事前に想定しておくことで「機能の漏れがなくなる」「画面が作成しやすく鳴る」「テストケースの洗い出しがしやすくなる」などの効果があります。

ユースケース図

利用者目線でのシステム利用例を表現した図です。要件（要求）を明確化できることに加え、ひと目でシステムで実現する機能の範囲を可視化することができます。

例



ユースケース記述

ユースケース記述はユースケース図で概要的に表されたユースケースを文章したものです。より明確に利用者の利用パターンが確認できるようになります。以下の内容をまとめます。

ユースケース名	X は Y をする（体言止めにしない）
概要	ユースケースの概要
アクター	利用ユーザ（システム外のものを指す。外部サービスなどもアクター）
事前条件	ユースケースが開始する前に満たされているべき条件
事後条件	実行された後に満たされているべき条件
基本フロー	正常に処理が進んだ場合。番号で順序を表す
代替フロー	途中でシステム内でエラーが発生した場合などに分岐する流れ
例外フロー	ユースケースの実行を断念しなければならないような手順。 ユーザから見えるシステムエラーで代替フローがない手順のこと。（例外フローを実行した場合には、事後条件が満たされない）

例

ユースケース ID	UC101
ユースケース名	会員登録をする
概要	新規会員としてシステムに登録する
アクター	会員
事前条件	会員メニューにアクセスできる
事後条件	会員登録される

基本フロー

1. メニューから「会員登録」を選択すると、このユースケースが開始される
2. システムは会員情報を入力する画面を表示する
3. アクターは登録する会員の名前、住所、電話番号、メールアドレス、生年月日を入力し、「確認画面へ」ボタンを押す
4. システムは登録情報の確認画面を表示する
5. アクターは「登録する」ボタンを押す
6. システムは会員を登録してランダムなパスワードを発行し、会員登録完了画面を表示する

代替フロー

なし

例外フロー

E-1： アクターが次の入力チェック条件を満たさずに「確認画面へ」ボタンを押した

- ✓ 「名前」は必須、50 文字以下
- ✓ 「住所」は必須、200 文字以下
- ✓ 「電話番号」は必須、20 文字以下
- ✓ 「メールアドレス」は、50 文字以下、メールアドレスとして正しいフォーマットであること、同じメールアドレスが登録されていないこと

1. システムは、入力が正しくない旨を伝えるメッセージとともに、会員情報入力画面を再表示する

備考

なし

4.4 外部設計

成果物： 外部設計書

外部設計とは要件定義で決まった内容をもとに画面や帳票などのユーザインタフェースを設計する工程です。利用者に提供する機能や操作感などを定義します。方法論の違いにより、外部設計に当たる工程を「基本設計」あるいは「概要設計」と呼ぶこともあります。

外部設計は要件定義と内部設計の中間に位置し、利用者が必要とする要件に基づいて、利用者から見てシステムがどのように振舞うべきかを決めていく作業です。操作画面や操作方法（ユーザインターフェース）の構成や、帳票類の書式だけではなくデータベースの構造などをこの時点で決めることが多くあります。

次のような図や表を作成します。

画面遷移図

画面名とその画面が遷移するときのイベントを記述します。要件定義書の「機能要件」の記述に基づいて、どの画面にどのような操作を行うと次の画面に遷移するかを顧客とすり合わせをしていきます。

画面遷移図ではなく、表形式などにすることもあります。画面の遷移が、開発側・顧客側で同一に認識されていることが重要です。

Web アプリケーションでは、リクエストのトリガーは何で、どの画面がレスポンスされてくるかをより明確にします。トリガーは、フォームのボタンなのか、リンクなのかも明確にします。

画面レイアウト

それぞれの画面のレイアウトを記述します。画面レイアウトは画面数分作ることになります。

Web アプリケーションでは、外部設計時に HTML を使って、実際に利用者にとってわかりやすい形で提示できるとより効果的です。外部設計の段階で認識の違いをすぐに修正することができるためです。

なお、デザインを乗せると修正に時間がかかるため、簡単な画面を作ることでまずは十分です。

テーブル設計書

システムに必要なデータベースを設計します。スキーマ設計などとも呼ばれます。

システムの内部操作や「画面レイアウト」で作られた画面の表示に必要な項目をデータベースのテーブルおよびカラムとして設計します。

例）テーブル設計書（研修日報テーブル）

番号	キー	カラム名	データ型	文字数	説明
1	PK	id	INTEGER		[ID] オートインクリメント 必須
2	FK	user_id	INTEGER		[ユーザ ID] 必須
3		post_date	DATE		[投稿日] 必須
4		comprehension	INTEGER	3000	[理解度] 必須 0〜100
5		content	TEXT		[日報内容] 必須
6		open	BOOLEAN		[公開設定] 必須
7		created_at	DATETIME		[作成日] 必須 自動生成
8		updated_at	DATETIME		[更新日] 必須 自動生成

PK：主キー（プライマリーキー） / FK：外部キー（フォーリンキー）

このとき、事前にテーブルに入れるためのデータについても検討し、実際にデータを作成することも行います。（都道府県の情報を先にマスターデータ[最初からないと困るようなデータ]として挿入しておくなど）

テーブル設計書を作るタイミング

上記にはカラム名やデータ型なども含め詳細情報が含まれていますが、外部設計時には顧客が判断できるのに十分な情報だけ載っていれば構わないことがほとんどです。

そのため外部設計時にはテーブル設計書は簡易に作成し、内部設計時に上記のような詳細情報が含まれたテーブル設計書を作成する場合もあります。（後ほど、巻末のサンプルドキュメントを確認してください）

設計書のフォーマット

テーブル設計書に限らず設計書はどのエンジニアにとってもわかりやすく、作るものがブレないことが求められます。しかしながら、プロジェクトによって力を入れるべき点や強調し対戦が変わります。

そのため、「必要十分な情報が載せてあることを前提に、ある程度フォーマットは自由」となっています。

例えば、テーブル設計書は上記例のようにまとめることもできますし、巻末サンプルドキュメントのようにもまとめられます。

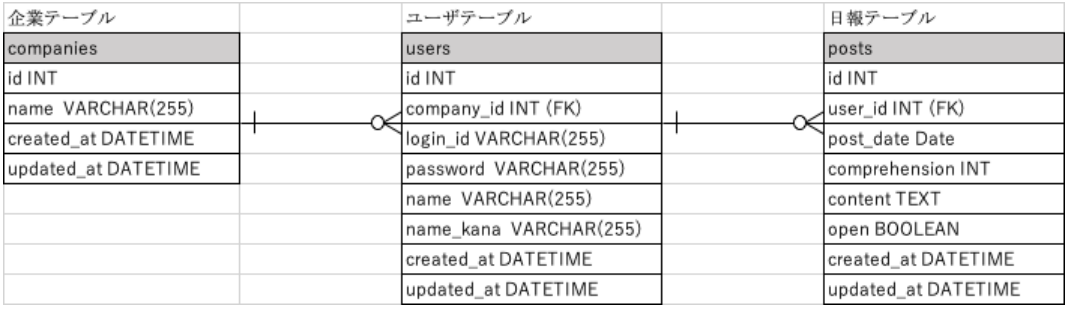
ER 図

データベースを情報のまとまり（エンティティ）と、情報の相互関係（）リレーションシップ）で表したものです。テーブル設計書を視覚的にわかりやすく表現するために作成します。

書き方には種類がありますが、IE（Information Engineering)記法について解説を行います。

下記 ER 図は、日報の管理システムに関するテーブル関係を示したものです。「企業テーブルとユーザテーブル」「ユーザテーブルと日報テーブル」はそれぞれ、「1 対多」の関係になります。

例）ER 図



関係性の記号

記号	意味
○ （しろまる）	0
| （交差する棒）	1
鳥足 （３本に広がる線）	多

これらの記号を組み合わせることで、関係性を表します。

例）1 対 0 以上

例）1 対 1 以上

ER 図はテーブル同士の関係性を可視化するのに有効であり、データベースの設計書だけではなく ER 図も書くことが一般的です。

4.5 内部設計

成果物： 内部設計書

外部設計を受けて、開発のためにシステム内部に特化した設計を行う工程です。全体の構成や行うべき処理の詳細など実装に必要な仕様を定義します。この工程を「機能設計」あるいは「詳細設計」と呼ぶこともあります。

外部設計で定められた機能や操作・表示方法などに基づいて、プログラムやシステムとしてそれをどう実現するかを具体的に定めていきます。そのため、顧客が通常認識をしなくていい部分を設計します。システム開発者に必要な情報について設計するということです。

画面レイアウト

外部設計でレイアウトした画面に対して、つぎの項目を定義します。

- フォームのアクション先の名前(URL)
- フォームの各部品の名前(name 属性)
- リンク先の URL

それぞれの URL については、必要に応じてクエリーの文字列も定義しておきます。また、フォームの文字数制限や必須・任意項目の確認なども、外部設計で実施していなければ、この時点で再度顧客とすり合わせを行います。

クラス設計書

オブジェクト指向型の言語では、クラスの設計がシステム全体に大きく関わってきます。そのため、クラスを事前に設計しておき、メンバー間で共通認識を図ります。

クラス設計では、必要なクラスを洗い出し、クラスのフィールドやメソッドについて記述します。設計する際のポイントには次のようなものがあります。

- データと、そのデータを使うロジックは、一つのクラスにまとめる
- 一つ一つのオブジェクトの役割は単純にする
- 複雑な処理は、オブジェクトを組み合わせて実現する

例）クラス設計書（研修日報モデル）

プロパティ メソッド	アクセス 修飾子	型	内容
■プロパティ			
id	private	int	ID
user_id	private	int	ユーザ ID
post_date	private	Datetime	投稿日
comprehension	private	int	理解度
content	private	String	日報内容
open	private	boolean	公開設定
・			
・			
・			
■メソッド			
setId(int id)	public	void	セッター：ID
setContent(String Content)	public	void	セッター：日報内容
・			
・			
getID()	public	void	ゲッター：ID
getContent()	public	void	ゲッター：日報内容
・			
・			
open?()	public	boolean	公開されているかどうかの確認

※ void は返り値がないことを指しています

クラス図

ソフトウェアの機能や構造を表す、よく使われる決まった図の書き方を UML と言います。クラス図は UML のひとつで、大規模な開発を行う際には書かれることが多いドキュメントです。

クラス設計を図に落とし込んだものです。クラスごとの関係性がわかりやすく、開発をスムーズに進められます。

クラスの構成はクラス名・属性（プロパティ）・操作（メソッド）の３つの要素です。

この構成に加えて、アクセス修飾子やクラス間の関係などを記述していきます。

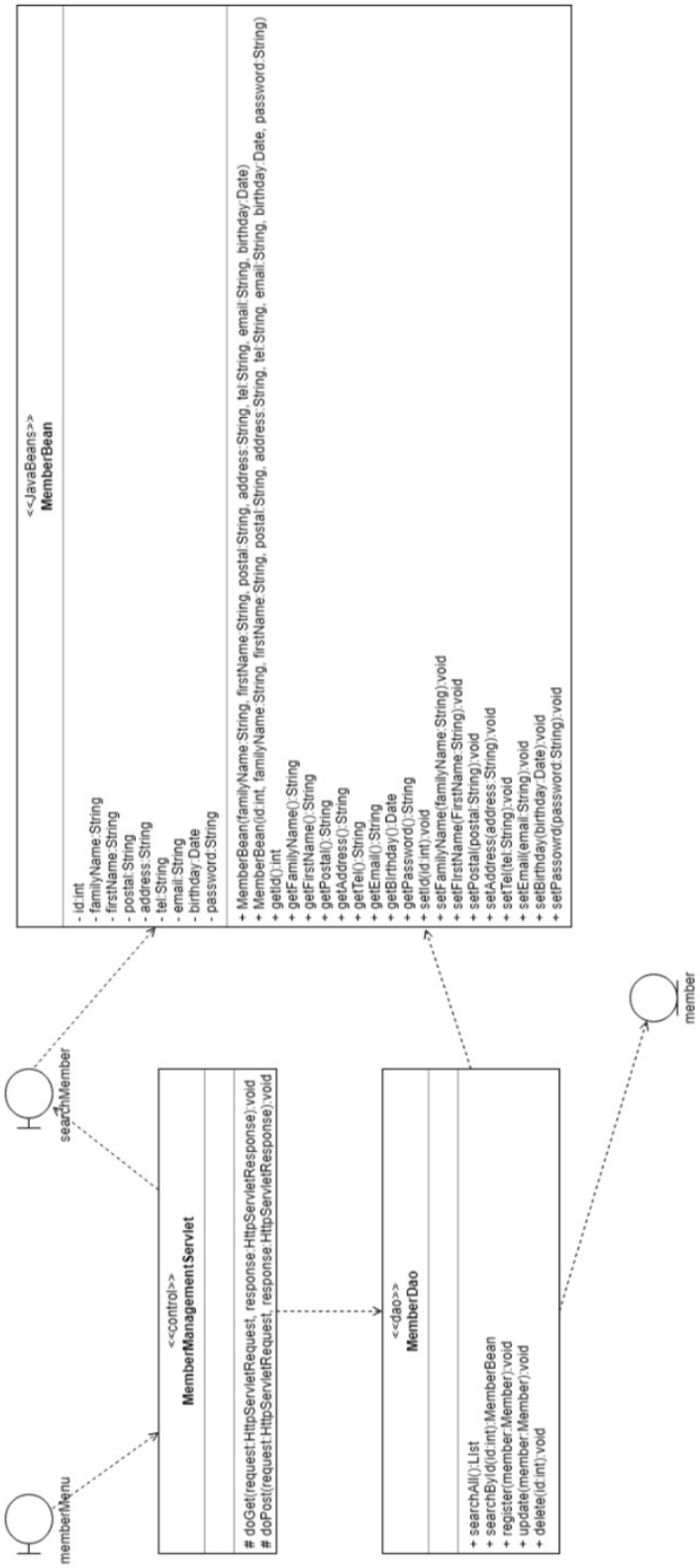
例えば、アクセス修飾子の記述は次の通りです。

アクセス修飾子	意味
+	public　：全てにおいて参照可能
-	private ：自クラスでのみ参照可能
#	protected ：自クラス及びサブクラスにおいて参照可能
~	package　：同パッケージ内で参照可能

また、矢印や記号を用いて継承などの関連性を表します。本研修の開発演習では、継承等の関係はそれほど用いることがないため、クラス設計書のみ記述しますが、現場ではクラス図が用いられることが多くなります。

※ Laravel や Rails などのフレームワークでのクラス設計に利用する場合、ある程度の形は決まっていることから、Model クラスのみをクラス図として書き起こすことで十分な場合がほとんどです。

例）クラス図　※Java 記述になっていますが他言語でも同様です



統合テスト仕様書

どのようなテストを行うかを事前に設計しておくのが統合（結合）テスト仕様書です。

Web アプリケーションでは、「リクエストからレスポンスまで」の一連の動作が重要な検証対象となります。そのため、「リクエストからレスポンスまで」の一連の動作を１ケースとします。したがって、統合テスト仕様書は、画面単位で作ると良いでしょう。

このテストケースを作ることで、仕様の確認や、仕様の漏れなどのチェックもできるため、プログラムを作る前にテストについて検討します。

4.6 コーディング・単体テスト

成果物： 単体テスト済のソースコード、クラスファイル（必要であればプログラム設計書）

内部設計書をもとにプログラミングします。この内部設計の要件および統合テスト仕様書に合うようにテストを行っていきます。

なお、チーム内のコーディングルールも設けておくことで、可読性が高まります。

プログラム設計書を作る場合もあります。プログラム設計の結果をまとめたドキュメントで、コーディングの基盤として作成します。プログラム設計書には、モジュールの処理内容や動作内容、入力データ、出力データ、データテーブルなどを記述します。

4.6.1 結合テスト

成果物： 結合テスト仕様書兼結果報告書

単体テストが済んだモジュール(クラス)をつなげて、結合テストを行います。このとき、統合テスト仕様書を作っているので、この仕様書に記述したテストケースをひとつずつテストをし結果が予想通りかどうかをチェックします。

結果が正しくない場合は、デバッグ作業を行い、モジュールを修正して、再度テストを行います。

このとき、修正に関係ないモジュールもテストしてきます。このテストを「再帰テスト(レグレッションテスト)」といいます。

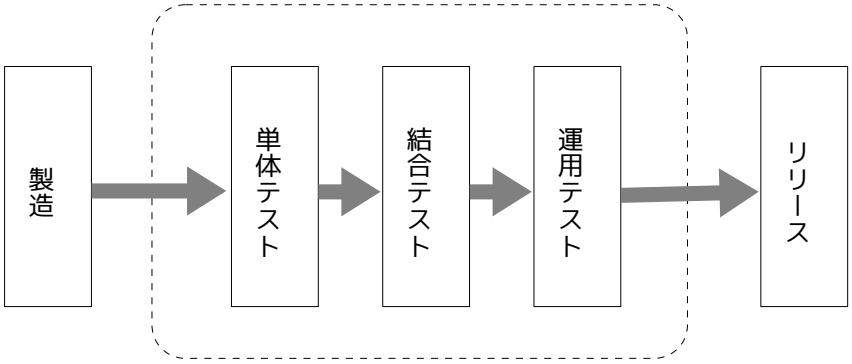
テストについて少し詳しく触れておきます。

4.6.2 テストとは

プログラムの欠陥（仕様誤りやコーディングミス）のことをバグといい、見つけたバグを取り除く作業をデバッグといいます。テストとはバグを検出する工程のことです。

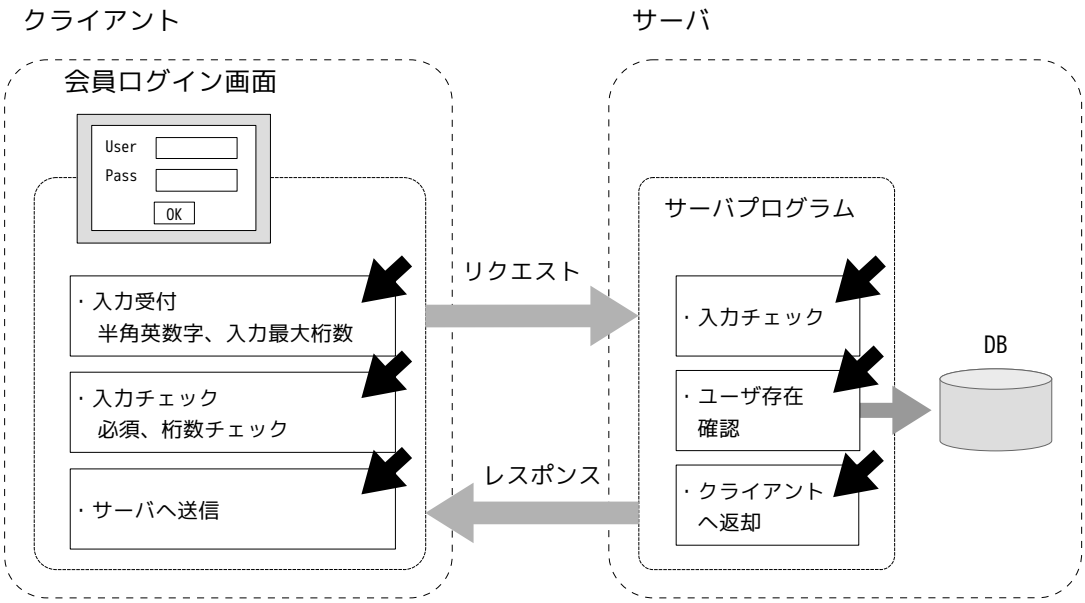
システム開発では、どの工程にもバグが混入する可能性があります。品質のよいソフトウェアとなるかどうかは、テストが正しく行えているかどうかにかかっているといえるでしょう。

ここでは、開発工程ごとのテストの種類とその解説をします。



テスト工程の流れ

以下の会員サイトへのログイン処理を例として、テスト工程を解説してきます。



会員サイトへのログイン処理の例

単体テスト

プログラムしたモジュール（部品）単位のテストを単体テストといいます。内部設計どおりにプログラムが動作するかを検証します。

上図の例であれば、黒矢印で示した箇所がモジュールに該当します。したがって、単体テストとしては以下のような検証を考える必要があります。

- クライアント処理
    - 入力受付（テキストボックスの IME モード、入力最大桁数）
    - 入力チェック（必須、桁数チェック）
    - サーバへ送信
- サーバ処理
    - 入力チェック（必須、桁数チェック）
    - ユーザ存在確認（DB 接続、SQL 問い合わせ）
    - クライアントへ返却（HTTP ステータス、次画面情報）

単体テストでは、ホワイトボックステストやブラックボックステストという手法を利用してテストケースを作成し検証します。各手法については後述します。

結合テスト

単体テストの工程が完了したら、結合テストに進みます。結合テストでは各モジュールを組み合わせせて、プログラムが外部設計どおりに動作するかを検証します。結合テストを行う単位はプログラム単位やサブシステム単位で行われることが多いです。

上図の例であれば、結合テストとして、以下のような検証を考える必要があります。

- 未入力の場合、サーバへリクエストがされずにエラーが表示されること
- 画面から入力されたユーザとパスワードの組み合わせが DB に存在しない場合、「ユーザとパスワードが一致しない」旨のエラーメッセージが画面に表示されること
- DB に存在するユーザとパスワードが入力された場合、正常にログイン処理が行われること

システムテスト

結合テストの中でも、システム全体を対象とした機能の確認や、実際の業務の使用に耐えうるかの検証をシステムテストといいます。

システムテストでは最終的に製品としてリリースする状態、またはそれに近しい状態で検証を行います。本番環境に近しい検証環境（ステージング環境）を用意して行うことが多いです。

上図の例であれば、システムテストとして、以下のような検証を考える必要があります。

- 機能テスト：システム全体を通して外部設計通りの利用ができることを確認する
- 回帰テスト（リグレッションテスト）：プログラムを修正・変更した箇所および影響する範囲が正しく動作するかを確認する
- デグレーションテスト：プログラムを修正・変更した箇所と関係のない箇所も検証することで、バグが混入していないかを確認する
- セキュリティテスト：外部からの悪意ある攻撃に対処できているか、脆弱性を検証する
- 性能テスト：処理能力（応答時間やスループット）が十分かを検証する
- 負荷テスト（ストレステスト）：高い負荷がかかった場合でも問題ないかを検証する

運用テスト

運用テストは、実際の業務と同じ条件下で動作検証を行います。要件定義通りにプログラムが動作するかを検証します。

テストの手順

テストは次のような手順で行います。

1．テストケースの設計

プログラムに入力するデータやメッセージを決めます。また、併せて出力結果の予想を立てます。

2．テストデータの作成

実際にテストするための入力データやメッセージを作成します。また、このときデータベースも作成します。

3．テストの実行

データベースを基に環境を作り、入力データを与え、プログラムの出力結果を得ます。

4．結果の検討

予想出力結果と実際の出力結果を比較します。結果が違う場合は，次の 2 つのケースが考えられます。

- バグがある
- テストケースの誤り

5．原因究明、修正

原因を究明し、デバッグ作業を行っていきます。

6．テスト済ケースの再テスト（回帰テスト）

修正によって、新しいバグが発生していないことを確認します。 リグレッションテストとも呼ばれます。

テストケースの設計

稼動後にバグが頻発するのは、テストケースがよくなかったこともその原因のひとつです。したがって、テストケースの設計はとても重要です。

テストケースの設計には、次の 2 つの方法があります。

- ホワイトボックステスト
- ブラックボックステスト

	ホワイトボックステスト	ブラックボックステスト
特徴	・ロジックから設計 ・経路や条件に依存	・仕様書から設計 ・入力データに依存
短所	・モジュールと仕様書が一致していることが保証できない ・データに依存するバグは発見できない ・機能の確認ができない	・すべての経路をテストできたと保証できない
手法	・論理網羅基準	・同値分割 ・限界値分析 ・エラー推測

ホワイトボックステスト

ホワイトボックステストは、プログラムの内部構造や内部論理に基づいてテストケースを選択していく方法です。 論理網羅基準という手法があり、次のような種類があります。ただし、コストやテスト期間のバランスを考えると、通常の適用業務プログラムの単体テストでは、判定条件網羅(C1 カバレッジ)を完了基準とする場合が多いです。

- 命令網羅(C0 カバレッジ)
- 判定条件網羅（分岐網羅）(C1 カバレッジ)

命令網羅(C0 カバレッジ)

プログラムのすべての命令は、少なくとも 1 回は実行されるように、テストケースを設定する基準です。

```
1     if (x > 1 && y == 0) { // (A)
2         z /= x;            // true のとき:(B), false のとき:(C)
3     }
4     if (x == 2 || z > 1) { // (D)
5         z++;               // true のとき:(E), false のとき:(F)
6     }
```

例えば、上記コードに対しては、以下が命令網羅のテストケースとなります。

経路	データ(x, y, z)	予想結果
A-B-D-E	(2, 0, 3)	z：2

判定条件網羅（分岐網羅）(C1 カバレッジ)

すべての判定条件を少なくても 1 回は真と偽の結果を持つように、テストケースを設定する基準です。

```
1     if (x > 1 && y == 0) { // (A)
2         z /= x;            // true のとき:(B), false のとき:(C)
3     }
4     if (x == 2 || z > 1) { // (D)
5         z++;               // true のとき:(E), false のとき:(F)
6     }
```

例えば、上記のコードに対しては、以下が判定条件網羅のテストケースとなります。

経路	データ(x, y, z)	予想結果
A-B-D-E	(2, 0, 3)	z：2
A-C-D-F	(1, 1, 0)	z：0

ソフトウェアの網羅率をカバレッジとも呼びます。プログラムコードからカバレッジを計測し目標とするカバレッジを達成するように単体テストを実施しようとした場合、テストケースの件数は多くなってきます。そのため、プログラムコードからカバレッジを計測・集計するツールも多く存在しています。

ブラックボックステスト

ホワイトボックステストを用いたケースを考えた後に、続いてプログラムの外部仕様に基づいてテストケースを選択するブラックボックステストを行います。ブラックボックステストは機能テストやシステムテストで利用される考え方です。

ブラックボックステストを実施する上で大事なのは、テストの観点を明確にすることです。以下にテスト観点一覧表の例を示します。

テスト観点一覧表の例[1]

大分類	中分類	テスト観点
機能	正常系	基本機能、表示、遷移（状態・画面）、ユーザインターフェース、設定（保持・変更・反映）、セキュリティ
	異常系	異常値入力・操作、異常データ、エラー検知・復旧、環境異常
	組合せ	同時操作、割込操作、排他処理、互換性
非機能	-	処理速度、連続動作、負荷、容量・リソース
ユーザ	-	業務シナリオ、ユーザビリティ、導入、保守
テスト	-	修正テスト、回帰テスト

代表的なテスト技法

ブラックボックステストを行うにあたって、入力値にどのようなデータを用いればよいか、また、テストケースをどのように作成すればよいかを考える必要があります。以下に具体的な技法を紹介します。

- 同値分割
- 限界値分析（境界値分析）
- エラー推測
- デシジョンテーブルテスト（組み合わせテスト）
- 状態遷移テスト

1　出典）石原一宏、田中英和『この一冊でよくわかるソフトウェアテストの教科書』を編集

同値分割

入力の値を同質のグループ（同値クラス）にわけ、それぞれのクラスの代表値をテストデータとする技法です。

同値クラスには以下のような条件で分割します。

- 同じ処理結果になる入力値のあつまり
- 同じ処理結果になる時間のあつまり
- 同じ入力値によって出力される結果のあつまり

同値クラスは、有効同値クラス（有効な入力）と無効同値クラス（誤った入力）に分けられます。

同値分割の長所としては、下記のことが挙げられます。

- 無作為にテストデータを選定する場合と比較すると、はるかに効果的なテストである
- 開発の早期から開始できる
- 利用者が作成しチェックできる

同値分割の短所としては、下記のことが挙げられます。

- 同値クラスごとにテストケースがひとつしかない
- 入力条件の組合せは考慮しない
- 通常は入力の同値クラスだけが重視される

同値分割の例

範囲の指定

入力/外部条件	有効同値クラス	無効同値クラス
数値として 0 から 100 までを指定	`0 <= 数値 <= 100`	`数値 < 0, 数値 > 100`

文字種類の指定

入力/外部条件	有効同値クラス	無効同値クラス
英字のみを指定	英字を指定	英字を指定しない

選択項目の指定

入力/外部条件	有効同値クラス	無効同値クラス
"MALE"または"FEMALE"を指定	"MALE", "FEMALE"	"MALE", "FEMALE"以外を指定

限界値分析（境界値分析）

仕様条件において、「入力値と出力値の境界となる値とその隣の値」に着目し、効果的にバグを検出する手法です。システムでは経験的に境界付近でバグが生じやすいことから、原因として、境界を表す条件の誤解やコーディングの時の誤りがバグを発生させる可能性を高くしています。

限界値分析の例

範囲の指定

入力/外部条件	テストする値
数値として 0 から 100 までを指定	0 の境界値 ： -1, 0, 1 100 の境界値 ： 99, 100, 101

エラー推測

仕様書や設計ドキュメント、ソースコードや開発者へのヒアリングによって、特異値をピックアップし、それをテストする値として設定します。特殊な処理を行っている変数に着目し、ピンポイントでテストすることによって発生しやすいバグの検出を目的としています。

デシジョンテーブルテスト（組み合わせテスト）

デシジョンテーブルとは、複数の条件の組み合わせに着目し、組み合わせの条件に漏れがないかを確認するための表です。デシジョンテーブルテストとは、デシジョンテーブルを作成することで、仕様から考えうる論理的な条件の組み合わせを洗い出し、その網羅性を高めるテスト手法です。条件の組み合わせによって、その結果が変化する場合に有効です。

例えば、以下のような遊園地の料金体系を考えます。

- 2000 円：「60 才以上、かつ、シルバーデー以外」、または、「12 才以上で 60 才未満」
- 1000 円：「60 才以上、かつ、シルバーデー」、または、「3 才以上で 12 歳未満」
- 0 円：「3 才未満」

上記の条件から入力と結果の論理的な組み合わせを考えると以下のようなデシジョンテーブルを作成することができます。

条件	パターン	1	2	3	4	5	6	7	8
入力	60 才以上	Y	N	N	N	Y	N	N	N
	12 才以上、60 歳未満	N	Y	N	N	N	Y	N	N
	3 才以上、12 才未満	N	N	Y	N	N	N	Y	N
	3 才未満	N	N	N	Y	N	N	N	Y
	シルバーデー	Y	Y	Y	Y	N	N	N	N
結果	2000 円		X			X	X		
	1000 円	X		X				X	
	0 円				X				X

上記のとおり、すべての論理的な組み合わせを考えると 8 パターンのケースが考えられます。

しかし、デシジョンテーブルには「複雑な仕様を整理する」という目的もあります。したがって一度論理的な全パターンを洗い出した後、以下のようにしてデシジョンテーブルを整理することで見やすい表を作成することができます。

- 矛盾している条件を削除
- 表を簡略化
- 表を分割

例えば、上記の例の場合は「シルバーデーであるかどうか」は「60 才以上であるか 60 才未満であるか」の 4 パターン確認できれば、検証の組み合わせとしては十分です。

したがって、以下のようにデシジョンテーブルを整理することができるでしょう。

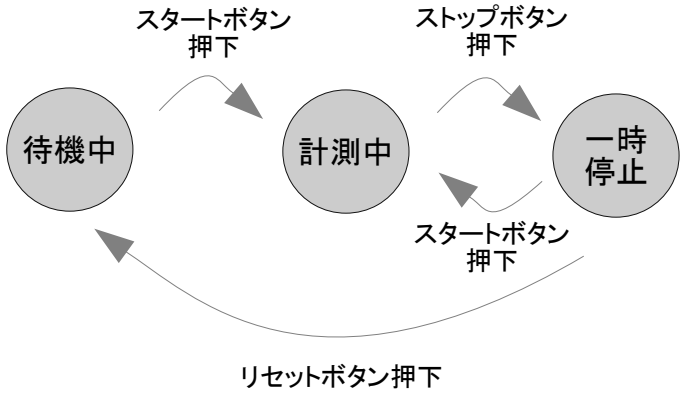
条件	パターン	1	2	3	4	5
入力	60 才以上	Y	Y	N	N	N
	12 才以上、60 歳未満	N	N	Y	N	N
	3 才以上、12 才未満	N	N	N	Y	N
	3 才未満	N	N	N	N	Y
	シルバーデー	Y	N	Y	N	N
結果	2000 円		X	X		
	1000 円	X			X	
	0 円					X

今回の例では、年齢の下限値や上限値が設定されていません。つまり「-5 才」や「1000 才」といった入力も可能です。しかし、仕様として「1000 才」という入力が可能なのは妥当とはいえないでしょう。テスト担当者は、設計者に対し、このような仕様の不備をあげることも重要な役割となります。

状態遷移テスト

「状態遷移図」や「状態遷移表」を用いたテスト技法を状態遷移テストといいます。状態遷移テストでは、状態遷移図から状態遷移表を作成し、「有効な状態遷移が正しく行われること」および、「無効な状態遷移が行われないこと」をテストします。

以下に例として、簡易的なストップウォッチの状態遷移図を示します。



状態遷移図（ストップウォッチ）

上記の状態遷移図を元に、状態（図の○で囲われている箇所）をすべてテストするのが、状態の網羅テストです。状態遷移図から以下の状態遷移表を作成します。状態遷移表のセルにはイベントによって遷移する先の状態を記述します。

状態遷移表（ストップウォッチ）

状態/イベント	スタートボタン押下	ストップボタン押下	リセットボタン押下
待機中	計測中	-	-
計測中	-	一時停止中	-
一時停止	計測中	-	停止中

状態遷移表におけるセル一つ一つをすべて（-としている箇所も含む）テストケースとして起こすことで、状態遷移の網羅テストを行うことが可能です。

代表的なテスト手法を利用したテストケースの作成例

以下のような機能に対して、テストケースを作成する場合の例を記します。

入力値として「年」「月」「日」「性別」があります。性別の項目には”MALE”か”FEMALE”の入力を受け付けます。年月日と性別の入力に誤りがなかった場合は「正しい年月日・性別」と出力し、誤りがあった場合は「正しくない年月日・性別」と出力する機能に対してテストを行う場合を考えます。

テストデータは以下の手順で作成します。

1. 条件に複数の組み合わせが存在する場合デシジョンテーブルを作成し、論理的な条件の組み合わせを洗い出す。
2. 同値分割されたそれぞれの同値クラスで、境界となる値を抽出する（限界値分析）。
3. 入力条件で「値を指定しない」 は無効同値クラスとする。ただし「値を指定しない」場合でデフォルト値があれば、有効同値クラスとする。
4. 有効同値クラスを組み合わせたテストデータを書き、すべての有効同値クラスがテストデータでカバーされるようにテストケースを作成する。
5. 無効同値クラスのひとつをカバーするテストデータを書き、すべての無効同値クラスがテストデータでカバーされるようにテストケースを作成する。

まずは、ディシジョンテーブルを作ります。

条件	パターン	1	2	3	4	5	6	7
入力	年は4桁	Y	Y	Y	Y	N	Y	Y
	1 <= 月 <= 12	Y	Y	Y	Y	Y	N	Y
	1 <= 日 <= 31	Y	Y	Y	Y	Y	Y	N
	“MALE”	Y	N	N	N	Y	Y	Y
	“FEMALE”	N	Y	N	N	N	N	N
	無指定	N	N	Y	N	N	N	N
	上記以外	N	N	N	Y	N	N	N
結果	正しい年月日・性別	X	X	X				
	正しくない年月日・性別				X	X	X	X

前記条件より、同値分割と限界値分析を使ってテストデータを考えます。

有効同値クラス	無効同値クラス	限界値(有効)	限界値(無効)
年は 4 桁	4 桁以外、無指定**(2)**	4 桁**(1)**	3 桁**(3)**，5 桁**(4)**
1 <= 月 <= 12	月 < 1，月 > 12，無指定**(7)**	1**(5)**，12**(6)**	0**(8)**，13**(9)**
1 <= 日 <= 31	日 < 1，日 > 31，無指定**(12)**	1**(10)**，31**(11)**	0**(13)**，32**(14)**
"MALE"**(15)** "FEMALE"**(16)** | 無指定**(17)**	左記以外**(18)**	-	-

上記をテストケースにすると、以下のようになります。ケース 1 から 3 は正常ケース、4 から 13 はエラーケースです。

番号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
値	1960	-	200	20003	1	12	-	0	13	1	31	-	0	32	MALE	FEMALE	-	MAN
正常ケース																		
1	X				X					X					X			
2	X					X					X					X		
3	X				X					X							X	
エラーケース																		
4		X			X					X					X			
5			X		X					X					X			
6				X	X					X					X			
7	X						X			X					X			
8	X							X		X					X			
9	X								X	X					X			
10	X				X							X			X			
11	X				X								X		X			
12	X				X									X	X			
13	X				X					X								X