要件定義・アーキテクチャ設計

目次

1 要件定義ーはじめに

1.1 はじめに

昨今、ソフトウェア業界を取り巻く環境は驚くべき勢いで変化を遂げている。パーソナルコンピューター、スマートフォン、そして IoT と呼ばれる電子機器など、我々は日々の生活において、息を吸うように当たり前にソフトウェアに触れるようになった。その結果、様々な企業がソフトウェア技術の分野に投資を行い、今までソフトウェア開発プロジェクトを経験した事がないような顧客からの依頼を受ける事も数多くなってきている。また、ハードウェアの進歩に伴ってソフトウェアの規模は膨れ上がり、要求は複雑化の一途を辿り、企業間の競争も激しい事から短い納期での開発を迫られる事も少なくない。

複雑かつ、規模の大きいソフトウェア開発プロジェクトにおいては、ある程度ソフトウェア開発プロジェクトを経験してきた顧客ですら、自身が何をソフトウェアに求めているのかを正確に把握しているケースは少ない、経験の少ない顧客であればなおさらである。しかし、我々は常に完成した成果物で評価される。例え、要求が曖昧な中、曖昧なまま作成された要求仕様書に挙げられた項目を滞りなく開発する事が出来たとしても、使ってみて満足がいかない成果物を顧客は使わなくなるし、それはイコール開発プロジェクトを引き受けたチームの信頼の低下に繋がる。

これが意味する事は、幾ら要件定義以降の工程を完璧にこなす事が出来ても、要件定義の段階で誤った方向に進んでしまえば、そのプロジェクトにおいて顧客が満足する成果物を作り上げる事が出来る可能性は大変低くなってしまう、そしてそれは往々にして起こり得る。

そこで、我々、ソフトウェアエンジニアは、要件定義を通して顧客との対話を行い、ただ顧客の発言を鵜呑みにするのではなく、その背景にある潜在的に求める物を理解し、それを抜け漏れなく、機能要求と非機能要求に振り分けて要求仕様書を作成する必要がある。このように、顧客が一人では表現しきれなかった要求を正しく引き出し、正しく後続の工程に引き継ぐ事でプロジェクトを成功に導き、最終的に真に満足度の高い成果物を作り上げる事が、顧客の信頼獲得に繋がる。

顧客との対話を通して行われる要件定義ではあるが、ソフトウェア工学では如何にして効率的に顧客の要求を引き出すのか、またそれらを正しく機能要求、非機能要求へと落とし込んでいくのか、についての知識体系がまとめられている。今回、我々は共通フレーム 2013 と言う、独立行政法人 情報処理推進機構 (IPA)が提唱する「標準」、における要件定義を軸に、より詳細に要件定義を扱う、要求工学や要求工学における知識体系である REBOK(アールイーボック)について触れる事でより詳細に掘り下げ、USDM を用い表現する方法について学ぶ。そして演習を通して具体的な要件定義の雰囲気をつかむべく対話形式で、実際に顧客から要求を抽出する様子を表す。

2 ソフトウェア工学の目的

2.1 ソフトウェア工学の定義

ソフトウェア工学は、さしあたって、社会的に広く普及している工業生産物としてのソフトウェアを対象とした学問であると考えられますが、以下、ソフトウェア工学の定義を確認します。

■IEEE による定義

IEEE[1]では、以下のように定義されています(IEEE Std 610.12-1990:強調は引用者)。

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.(2) The study of approaches as in (1).

(1) ソフトウェアの開発、運用、および保守に対する、系統的で規律のある定量化されたアプローチの適用、すなわち、ソフトウェアへの工学の適用
(2) (1)において示されたアプローチに関する研究

■プリーガーによる定義

ソフトウェア工学に関する著名な文献の著者であるプリーガーは、ソフトウェア工学を、高品質なソフトウェアを開発するための学問としていますが、ソフトウェアの開発を純粋な科学としては規定しておらず、同時に「芸術」でもあると考えています。

ソフトウェアを作成することは、科学であると同時に芸術でもある。コンピュータサイエンスの学生としては、この理由を理解することが重要である。コンピュータサイエンスの専門家やソフトウェア工学の研究者は、コンピュータのメカニズムを研究して、より生産的・効率的にする方法について理論づけを行なっている。しかし、彼らもコンピュータシステムを設計し、そのシステム上で作業するためのプログラムを書いている。このことは、多くの芸術性、精巧さ、技術力を包含する実践といえる。特定のシステム上で特別な作業を実行するには多くの方法がある。

1 IEEE は、「アイ・トリプル・イー」と呼称し、"The Institute of Electrical and Electronics Engineers "（全米電気電子技術者協会）の略称である。

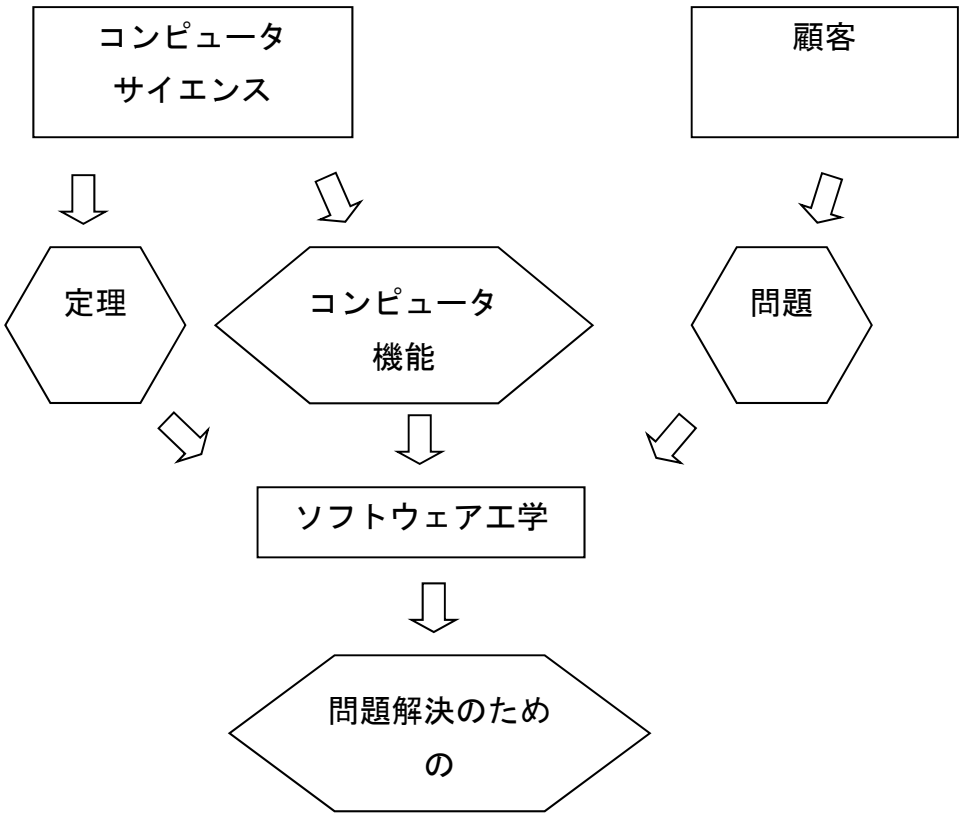
> しかし、そのいくつかは他よりも良いことは確かである。ある方法は、より効率的・実践的で、修正が容易で、使いやすく理解しやすいかもしれない。どんなハッカーでも何かを動作させるために適当なコードが書ける。しかし、頑丈で理解しやすく保守も容易で、かつ、できるだけ効率的で効果的な方法で仕事が遂行できるようなコードを生み出すことに労を惜しまないのがプロのソフトウェアエンジニアの姿である。したがって、ソフトウェア工学は高品質のソフトウェアを設計して、開発するために存在するといえる。
> (堀内泰輔 訳, 『ソフトウェア工学 理論と実践』, ピアソン・エデュケーション, 2006 pp.5-6: 強調は引用者)[2]

プリーガーは、さらに、〈コンピュータサイエンス〉(computer science:計算機科学)[3]と対比して、ソフトウェア工学の位置づけを、以下の図によって示しています。プログラム言語自体やコンパイラの開発、ハードウェアとしてのコンピュータ自体の開発、アルゴリズムの発見といった研究は、ソフトウェア工学ではなく、〈コンピュータサイエンス〉の役割とされています。ソフトウェアは、〈コンピュータサイエンス〉の研究成果やその応用としての工業生産物(ハードウェアなど)に基づいて開発されるという関係にあります。そして、ソフトウェア工学は、顧客の抱える実務上の問題をソフトウェアによって解決するさいに、そのための適切な手段や方法論を産み出すという役割を担っていると考えられています。

---

2 Pfleeger, Shari Lawrence & Atlee, Joanne, " Software Engineering: Theory and Practice", Pearson Education, 2005 (堀内泰輔 訳, 『ソフトウェア工学 理論と実践』, ピアソン・エデュケーション, 2006)

3 ウィキペディアでは、「情報と計算の理論的基礎、及びその計算機上への実装と応用に関する研究分野である」と定義されている (http://ja.wikipedia.org/wiki/%E8%A8%88%E7%AE%97%E6%A9%9F%E7%A7%91%E5%AD%A6 参照)。



このソフトウェア工学にとっての「顧客」という存在と、プリーガーがソフトウェア開発を「科学であると同時に芸術でもある」といっていたことの間に、どのような関連性があるのか興味深い問題です。

■河村一樹による定義

河村の定義は、ソフトウェア工学が扱う対象範囲を、漠然と「ソフトウェア開発、運用、および保守」とせずに、ソフトウェアの〈ライフサイクル〉(life cycle)の「全局面」とした点で明快です。反面、プリーガーがソフトウェア工学を、単純に「科学」として定義しなかった(できなかった？)問題意識は、河村の定義にはあまり見いだされません。

ソフトウェア工学(software engineering)という言葉は、他の工学分野(電子工学や電気工学など)と同義語になるように意図してつけられたといえる。工学という言葉には、理論的に体系化された定理や科学的な定義や法則などを、実践的な工業技術へ結びつけるという意味が含まれている。ソフトウェア開発管理における実務工程を、理論や方法論や技法によって

体系化し、工学的なアプローチにより実現しようという意向が見出される。
（河村 一樹，『改訂新版 ソフトウェア工学入門』，近代科学社，2008，p.35）

ソフトウェア工学は、ソフトウェアを生産し管理する工程（ライフサイクル）の全局面において、工学に基づく学問成果（基礎理論、方法論、概念）と実用的な技術（含む、技法）を、体系化した学術的および実践的な領域である。それとともに、ソフトウェア工学では、高品質のソフトウェアを効率よく生産し管理することを目標にする。つまり、ソフトウェア開発保守における生産性と信頼性をともに向上させることを目指すための学問かつ実務領域を、ソフトウェア工学と定義する。
（同前 pp.40-41：強調は引用者）

工学ということは、普遍的な科学的法則を適用するという立場をとることでもある。（途中省略）・・・ソフトウェア工学という名称には、このような工学的な立場を明らかにしたいという意図が表れている。これにより、経験や勘といった直観的で主観的な要素は極力排除して、より科学的に工学的なアプローチにもとづいた視点に立ちたいという目論見が見出される。
（同前 p.42：強調は引用者）

河村によるソフトウェア工学の定義を簡潔に表現するならば、以下のようになります。

ソフトウェア工学とは、ソフトウェアのライフサイクルの全工程（開発・運用・保守）に対して、体系化された科学的理論と科学的方法論を適用しようとする学問である。

河村の定義では、プリーガーとは異なり、科学的客観的法則に基づいた学問への志向性が強くなっています。

■『実践的ソフトウェア工学』の浅井治による定義

> このように、ソフトウェアを産業としてとらえた場合、生産性や品質をコントロールし、最適な開発方法を見いだすことが重要となる。ソフトウェアの難しさを払拭して、生産性と品質を制御することがまさしく、ソフトウェア工学の目的であり、ソフトウェア工学の最終的な目標は、「品質(quality)」、「コスト(cost)」、「納期(delivery)」の 3 つの最適なバランスを実現するための手法や方法論を学ぶことである。
> (浅井 治, 『実践的ソフトウェア工学』, 近代科学社, 2009, pp.11-12)

> ソフトウェア工学では、属人的な要素を取り除き、ベストプラクティス(best practice)を追求する必要があろう。ここで、ベストプラクティスとは、「お手本」のことであり、ソフトウェアの種別や機能、使われ方によって千差万別である。これらに、統一的なルールを当てはめることは、ある意味ナンセンスで、ソフトウェア工学の目的ではない。ソフトウェア工学では、このような個々のソフトウェアの事情に左右されないノウハウや、あるべき姿の集大成として位置づけ、読者は、これらの使えそうなところを適宜選択し、真似をしたり拡充したりして自身の環境に最適化した形で適用すればよい。このためのノウハウ集を体系化したものととらえ、ソフトウェア工学で学んだことを現場で活用することになる。「ソフトウェア工学」がカバーする範囲は広範に及ぶが、本書では、より実践的なトピックに絞り、解説してゆくことにする。
> (同前 p.12:強調は引用者)

浅井によれば、ソフトウェア工学は「ベストプラクティス」[4]を追求するものですが、それは「ソフトウェアの種別や機能、使われ方によって千差万別」であるため、「統一的なルール」とはならないものです。したがって、普遍的法則や普遍的判断のようなものではなく、経験的事実(理想的な成功事例)を集めたものでしかなく、「あるべき姿の集大成」、「ノウハウ集」です。そして、そのような「ノウハウ集」としてのソフトウェア工学のなかから、エンジニアが、「使えそうなところを適宜選択し、真似をしたり拡充したりして自身の環境に最適化した形で適用」するものです。この場合、ソフトウェア工学は「経験や勘」(河村)を排除することができないことになります。

---

4 「ベストプラクティス (英: best practice) は、ある結果を得るのに最も効率のよい技法、手法、プロセス、活動などのこと。最善慣行、最良慣行と訳されることもある。また、仕事を行うために最も効率のよい技法、手法などがあるという考え方をいう。すなわち、適切なプロセスを確立し、チェックと検証を行えば、問題の発生や予期しない複雑さを低減させて、望ましい結果が得られると考える。ベストプラクティスは、多くの人々によって反復され、最も効率的で最も効果的であることが時間をかけて証明されてきた。」
(http://ja.wikipedia.org/wiki/%E3%83%99%E3%82%B9%E3%83%88%E3%83%97%E3%83%A9%E3%82%AF%E3%83%86%E3%82%A3%E3%82%B9 参照)

■まとめ

ソフトウェア工学の定義は、研究者、技術者によっても、まちまちな印象です。一方では、経験的であり、他方では、学問的に規定されています。この両極の間で、その定義は揺らいでいるといえます。

2.2 SWEBOK ガイド

これまで見たように、ソフトウェア工学の定義については、様々な研究者やエンジニアの間で意見が分かれており、そのままでは共通見解に至ることができません。そこで、ソフトウェア工学の学問体系を規定し、かつ、標準として策定されている"SWEBOK"(「スウィーボック」もしくは「スウェボック」)について確認してみます(『実践的ソフトウェア工学』, pp.148-149 の「11.2 SWEBOK の目標」参照)。

■SWEBOK

"SWEBOK"(Software Engineering Body of Knowledge)とは、2004 年に、IEEE と ACM[5]により策定された、ソフトウェア工学に関する知識体系(Body of Knowledge)のことで、ソフトウェア工学の体系の全体(ソフトウェア工学の範囲、全構成領域)を定めたものです。SWEBOK ガイド(Guide to the Software Engineering Body of Knowledge)は、"SWEBOK"で定めた内容を示す文書です。

SWEBOK ガイドは、ソフトウェア工学の体系が網羅するべき知識の詳細(ソフトウェア開発・運用・保守に関する具体的な理論や方法論の詳細)を示すものではなく、ソフトウェア工学の体系を構成する全領域を示し、領域ごとの概要と基本用語を説明するだけにとどまるものです。さらなる詳細情報を求める場合は、領域ごとに提示されている推奨文献を参照することになっています。

SWEBOK ガイドの 2004 年版である"2004 SWEBOK Guide"は、http://www.computer.org/portal/web/swebok/html/contents (英語原文)および http://marunomaruno.web.fc2.com/swebok_application01.html?c=appendix(日本語)において参照することができます。また、日本語訳は、2004 年度版が出版されています(松本吉弘 訳,『ソフトウェアエンジニアリング基礎知識体系―SWEBOK 2004』. 2005, オーム社)。

以下、SWEBOK ガイド 2004 年度版の項目です。

第 1 章 序説
第 2 章 ソフトウェア要求

2.1 ソフトウェア要求の基礎
2.2 要求プロセス

5 ACM は、"Association for Computing Machinery "(米国計算機学会)の略称であり、米国の計算機科学分野の学会のひとつである。1947 年設立。IEEE とともに、IT 分野で最も影響力ある学会のひとつである。和名では「米国計算機学会」、「アメリカ計算機学会」とされる。

第 3 章 ソフトウェア設計

3.1 ソフトウェア設計の基礎
3.2 ソフトウェア設計における主要な問題
3.3 ソフトウェア構造とアーキテクチャ
3.4 ソフトウェア設計品質の分析と評価
3.5 ソフトウェア設計のための表記
3.6 ソフトウェア設計戦略および手法

第 4 章 ソフトウェア構築

4.1 ソフトウェア構築の基礎
4.2 ソフトウェア構築のマネジメント
4.3 実践上考慮すべきことがら

第 5 章 ソフトウェアテスティング

5.1 ソフトウェアテスティングの基礎
5.2 テストレベル
5.3 テスト技法
5.4 テストに関係した計量尺度
5.5 テストプロセス

第 6 章 ソフトウェア保守

6.1 ソフトウェア保守の基礎
6.2 ソフトウェア保守における主な課題
6.3 保守プロセス
6.4 保守のための技法

第 7 章 ソフトウェア構成管理

7.1 SCM プロセスのマネジメント
7.2 ソフトウェア構成識別

7.3 ソフトウェア構成コントロール
7.4 ソフトウェア構成実態説明
7.5 ソフトウェア構成監査
7.6 ソフトウェアリリース・マネジメントおよび引渡し

第 8 章 ソフトウェアエンジニアリング・マネジメント

8.1 始動および適用範囲の定義
8.2 ソフトウェアプロジェクト計画
8.3 ソフトウェアプロジェクトの計画実践
8.4 レビューおよび評価
8.5 終結
8.6 ソフトウェアエンジニアリング計量

第 9 章 ソフトウェアエンジニアリングプロセス

9.1 プロセスの実現および変更
9.2 プロセス定義
9.3 プロセス査定
9.4 プロセスおよびプロダクト計量

第 10 章 ソフトウェアエンジアリングのためのツールおよび手法

10.1 ソフトウェアエンジアリングツール
10.2 ソフトウェアエンジアリング手法

第 11 章 ソフトウェア品質

11.1 ソフトウェア品質の基礎
11.2 ソフトウェア品質マネジメントプロセス
11.3 実践上考慮すべきことがら

学問の体系が、規格団体が策定した定義に依存するということは、一般的な学問のあり方からすれ

ば異常なことです。哲学、文学、心理学、教育学、社会学、歴史学といった人文科学系諸学問も、数学、物理学、化学、生物学、医学といった自然科学系諸学問も、それ自身の学問体系は、その学問が自ら定めるのであって、規格団体が定めているのではありません。これら従来の学問は、それぞれが固有の学問領域を保持すると同時に、単なる経験的事実の「集大成」とは異なるものです。

3 ソフトウェアライフサイクル

3.1 ソフトウェア工学とソフトウェアライフサイクル

これまで扱ってきたように、ソフトウェア工学は、エンジニア、研究者によって、さまざまに定義されていますが、さしあたって、多くの消費者によって利用される工業生産物としてのソフトウェアを対象とした学問であると考えられます。以下、IEEE による代表的な定義です。

■IEEE による定義

IEEE[6]では、以下のように定義されています(IEEE Std 610.12-1990)。

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.(2) The study of approaches as in (1).

(1) ソフトウェアの開発、運用、および保守に対する、系統的で規律のある定量化されたアプローチの適用、すなわち、ソフトウェアへの工学の適用
(2) (1)において示されたアプローチに関する研究

ソフトウェアの開発、運用、保守とは、ソフトウェアの生成から廃棄に至るソフトウェアのライフサイクル(〈ソフトウェアライフサイクル〉:software life cycle)の全体を意味していると考えられます。つまり、IEEE の定義によれば、ソフトウェア工学は、ソフトウェアの生成から廃棄に至る全過程に対して、工学的な原理を確立し適用しようというものです。

6 IEEE は、「アイ・トリプル・イー」と呼称し、"The Institute of Electrical and Electronics Engineers "(全米電気電子技術者協会)の略称である。

3.2 ソフトウェアライフサイクル

それでは、ソフトウェア工学が自身の工学的な原理を適用しようとした場合、その適用先の〈ソフトウェアライフサイクル〉の全体は何かということが最初の問題となります。この問題の意味は、〈ソフトウェアサイクル〉の全体は、どのような時系列、作業区分、作業内容によって構成されているのかということです。

そこで、〈ソフトウェアライフサイクル〉に関して、一定の時系列モデル(概念モデル)を考案し、時系列の構造とその内部にある複数の作業区分と作業区分間の関係、さらに作業区分内部の作業内容を明確化することが必要となります。時系列の構造と作業区分、作業区分内の作業内容を学問的に(工学的に)規定できれば、ソフトウェア工学の目的である〈ソフトウェアライフサイクル〉への工学的原理の適用が可能になるからです。この問題に関連して、〈ソフトウェアライフサイクルモデル〉、〈ソフトウェアライフサイクルプロセス〉という二つの概念が SWEBOK ガイドで触れられています。

3.3 ソフトウェアライフサイクルモデルとソフトウェアライフサイクルプロセス

〈ソフトウェアライフサイクル〉がどのようなものであるかという問題に関連して、SWEBOK ガイド 2004[7]では、〈ソフトウェアライフサイクルモデル〉(Software Life Cycle Model)という概念と〈ソフトウェアライフサイクルプロセス〉(Software Life Cycle Process)という概念を規定しています。

以下、〈ソフトウェアライフサイクルモデル〉概念に関する規定です。

> ソフトウェアライフサイクルモデルは、開発の途上に生じる各局面(phases)に関する高レベルな定義(抽象度の高い定義:訳者注)を提供する。ソフトウェアライフサイクルモデルは、詳細な定義を提供することを狙っているのではなく、主要なアクティビティとアクティビティ間の相互依存関係を際立たせることを狙っている。ソフトウェアライフサイクルモデルの例として、ウォーターフォールモデル、使い捨てプロトタイピングモデル(throwaway prototyping model)、進化型開発(evolutionary development)、漸進/反復型引渡し(incremental/iterative delivery)、スパイラルモデル、再利用可能ソフトウェアモデル、さらに自動的なソフトウェア総合(automated software synthesis)が挙げられる。
> (CHAPTER 9 SOFTWARE ENGINEERING PROCESS 2.1.Software Life Cycle Models の拙訳:強調は訳者による)

7 http://www.computer.org/portal/web/swebok/html/ch9#Ref2.1 参照。日本語訳は、『ソフトウェアエンジニアリング基礎知識体系—SWEBOK2004』(松本 吉弘 訳, 2005, , オーム社) として出版されている。

以下は、〈ソフトウェアライフサイクルプロセス〉概念に関する規定です。

> ソフトウェアライフサイクルプロセスの定義は、ソフトウェアライフサイクルモデルの場合よりも、より詳細化する傾向をもっている。しかし、ソフトウェアライフサイクルプロセスは、各プロセスを時間軸上に配置しようとするものではない。このことの意味するところは、ソフトウェアライフサイクルプロセス(に含まれる各プロセス:訳者注)は、原理的には、いかなるソフトウェアライフサイクルモデルに対しても適合して配置されることが可能であるということである。この領域に関する主な参考文献は、"IEEE/EIAA 12207.0:Information Technology – Software Life Cycle Processes [IEEE 12207.0-96]"である。
> (CHAPTER 9 SOFTWARE ENGINEERING PROCESS 2.2.Software Life Cycle Processes の拙訳:強調は訳者による)[8]

SWEBOK ガイド 2004 の規定をまとめれば、〈ソフトウェアライフサイクルモデル〉とは、〈ソフトウェアライフサイクル〉から抽象化によって取り出された、ソフトウェアの製造から廃棄までの複数の局面(作業区分)と局面間の関係(作業区分間の関係)を含む時系列モデルを意味し、〈ソフトウェアライフサイクルプロセス〉は、それぞれの局面(作業区分)に当てはめられるべき、より詳細な作業区分、具体的な作業内容を含むということができます。簡単にいえば、〈ソフトウェアライフサイクルモデル〉は作業区分と作業区分間の関係を含む一定の時系列モデルを、〈ソフトウェアライフサイクルプロセス〉は、そこに(各作業区分に)当てはめられるべき、より詳細な作業区分、具体的な作業内容を意味します。

ところが、河村一樹は、〈ソフトウェアライフサイクルプロセス〉として、「ウォーターフォールモデル」(water-fall model)、「成長モデル」(evolution model)、「らせんモデル」(spiral model)を挙げています[9]。これらは、SWEBOK では、〈ソフトウェアライフサイクルモデル〉に該当するもので、両者では概念の意味が食い違っています。

また、プレスマン(Roger S.Pressman)は、「プロセスモデル」(Process Model)として、「ウォーターフォールモデル」、「インクリメンタルプロセスモデル」、「進化型プロセスモデル」などを挙げています[10]。

---

8 "IEEE 12207.0-96"とは、後述する"ISO/IEC 12207：1995"の米国版である。両者は同じ内容である。

9 河村一樹,『改訂新版 ソフトウェア工学入門』, 近代科学社, 2008, pp.53-60

10 Pressman, Roger S., "Software Engineering: A Practitioner's Approach 6th Revised edition", McGraw Hill Higher Education, 2004（西康晴・榊原彰・内藤裕史監訳 吉沢聡子・正木めぐみ・関口梢訳,『実践ソフトウェアエンジニアリング ソフトウェアプロフェッショナルのための基本知識』, 日科技連出版社, 2009）, 邦訳 pp.35-44

プリーガーも同様に、「プロセスモデル」として、「ウォーターフォールモデル」、「V モデル」、「プロトタイピングモデル」、「操作的仕様モデル」、「変形モデル」、「漸進的開発」(incremental development)、「反復的開発」(iterative development)、「スパイラルモデル」を挙げています[11]。

一方、国内の文献では、『ソフトウェアエンジニアリング講座1 ソフトウェア工学の基礎』が、「開発プロセス」として、「ウォーターフォール型開発プロセス」、「スパイラルモデル」、「反復型開発プロセス」、「Win-Win スパイラルモデル」、「アジャイル開発」を挙げています[12]。『実践的ソフトウェア工学』では、「開発プロセス」として、「ウォーターフォール型開発プロセス」、「スパイラルモデル」、「反復型開発プロセス」、「アジャイルプロセス」を挙げています[13]。『要求工学知識体系 第1版』では、「ソフトウェアプロセスモデル」として、「アジャイルプロセスモデル」、「インクリメンタルプロセスモデル」、「イテラティブプロセスモデルを挙げています[14]。SQuBOK ガイドでは、「プロセスモデル」として、「ウォーターフォールモデル」、「スパイラルモデル」、「インクリメンタルモデル」が挙げられています[15]。

Wikipedia("ソフトウェア開発工程")では、「開発工程モデル」として、「ウォーターフォール型」、「反復型」、「形式手法」を挙げています[16]。

---

11 Pfleeger, Shari Lawrence , Joanne, Atlee "Software Engineering: Theory and Practice", Pearson Education, 2005（堀内 泰輔 訳, 『ソフトウェア工学 理論と実践』, ピアソン・エデュケーション, 2006）, 邦訳 pp.49-65

12 IT トップガン育成プロジェクト, 『ソフトウェアエンジニアリング講座１』, 日経 BP 社, 2007, pp.23-40

13 浅井 治, 『実践的ソフトウェア工学』, 近代科学社, 2009, pp.48-57

14 JISA：一般社団法人情報サービス産業協会 REBOK 企画 WG, 『要求工学知識体系 第１版』, 近代科学社, 2011, pp.38-39

15 『SQuBOK ガイド第一版 β版』 p.71

16 http://ja.wikipedia.org/wiki/%E3%82%BD%E3%83%95%E3%83%88%E3%82%A6%E3%82%A7%E3%82%A2%E9%96%8B%E7%99%BA%E5%B7%A5%E7%A8%8B

それぞれの文献・資料の内容をまとめると次のようになります。

参照先	〈ソフトウェアライフサイクルモデル〉概念に該当する用語	〈ソフトウェアライフサイクルプロセス〉概念に該当する用語
河村一樹	ソフトウェアライフサイクルプロセス	なし（左の用語に含まれる？）
プレスマン	プロセスモデル	なし（左の用語に含まれる？）
プリーガー	プロセスモデル	なし（左の用語に含まれる？）
『ソフトウェアエンジニアリング講座１』	開発プロセス	なし（左の用語に含まれる？）
『実践的ソフトウェア工学』	開発プロセス	なし（左の用語に含まれる？）
『要求工学知識体系 第１版』	プロセスモデル	なし（左の用語に含まれる？）
SQuBOK ガイド	プロセスモデル	なし（左の用語に含まれる？）
Wikipedia ("ソフトウェア開発工程")	開発工程モデル	ソフトウェア開発工程（Software Development Process） ソフトウェアライフサイクル ソフトウェア開発プロセス ソフトウェアプロセス

以上のように、SWEBOK ガイドブック 2004 の提示する〈ソフトウェアライフサイクルモデル〉および〈ソフトウェアライフサイクルプロセス〉概念は、他の文献、資料では異なる用語（と説明）が用いられているだけでなく、そもそも二つの概念として分離されていない場合もあり、ソフトウェア工学の内部で混乱した状態にあるといえます。

3.4 ISO/IEC 12207：1995 における概念規定

それでは、SWEBOK が参考文献として指定している IEEE 12207.0-96 と同内容の ISO/IEC 12207:1995[17]に基づき、〈ソフトウェアライフサイクルモデル〉と〈ソフトウェアライフサイクルプロセス〉概念の意味を確認してみます。

ISO/IEC 12207:1995 は、〈ソフトウェアライフサイクルプロセス〉の標準化を定めた規格であり、1995 年に策定されたものです。その目的は、ソフトウェアのライフサイクルに関わる全作業に、一定の内容（ただし、具体的な内容でなく共通の枠組みとしての内容）を取り決め、ソフトウェア開発の顧客と生産者の間で発生している混乱、生産者内部における混乱を防止することにあります。

ISO/IEC 12207:1995 は、1996 年に日本工業規格（JIS）により採用され、翻訳および補足を行い JIS X 0160:1996 として規格化されています[18]。そこで、JIS X 0160:1996 の内容から、〈ソフトウェアライフサイクルモデル〉と〈ソフトウェアライフサイクルプロセス〉の関係を確認してみます。

序文では、JIS X 0160:1996 の目的が示されています。

> （ソフトウェアの開発・管理を巡る環境が複雑になってきているため：引用者注）ソフトウェアにかかわる人々がソフトウェアを作成し、管理するときにおいて"同じ言葉を話す"ことができるように、共通の枠組みをもつことが必要となる。この規格は、こうした共通の枠組みを提供する。
>
> この枠組みは、ソフトウェアの構想から廃棄に至るまでのソフトウェアライフサイクルを包含し、ソフトウェア製品およびサービスの取得から供給までのプロセスを含んでいる。さらに、この枠組みは、プロセスの管理および改善について規定する。

さらに、この規格の適用範囲について示されています。

> この規格は、明確に定義された用語を使い、ソフトウェアライフサイクルプロセスの共通枠組

17 ISO は、正式名称を International Organization for Standardization（国際標準化機構）といい、、各国の代表的標準化機関から成る国際標準化機関で、電気及び電子技術分野を除く全産業分野（鉱工業、農業、医薬品など）に関する国際規格の作成を行っています。IEC は、正式名称を International Electrotechnical Commission（国際電気標準会議）といい、各国の代表的標準化機関から成る国際標準化機関であり、電気及び電子技術分野の国際規格の作成を行っています。ISO と IEC は共同で規格の策定を行うことがあります。

18 さらに、情報処理振興事業協会(現在の IPA：情報処理推進機構)が中心となり、ソフトウェアライフサイクルにおける各プロセスを分類・形式化するガイドラインが、SLCP-JCF98（Software Life Cycle Process – Japan Common Frame 98）として策定されています。

みを規定しており、ソフトウェア産業界で引用することができる。この規格は、プロセス、アクティビティ及びタスクで構成され、ソフトウェアを含むシステム、単体ソフトウェア製品及びソフトウェアサービスを取得するとき、並びにソフトウェア製品の供給、開発、運用及び保守をするときに適用できる。(強調は引用者)

〈ソフトウェアライフサイクルモデル〉との関係は、「1.5 制限」の箇所に記述されています。

この規格は、ソフトウェアライフサイクルプロセスの構造を指定しており、プロセスの中に含まれるアクティビティ及びタスクの実現又は実行するための詳細な方法については指定しない。(・・・)

この規格は、特定のライフサイクルモデルやソフトウェア開発手法を指定しない[19]。この規格を適用する者は、そのソフトウェアプロジェクトに適したライフサイクルモデルを選択し、この規格中のプロセス、アクティビティ及びタスクをそのモデルに割り付ける責任を持つ。また、ソフトウェア開発手法を選択、適用して、そのソフトウェアプロジェクトに適合したアクティビティ及びタスクを実行する責任をもつ。(強調は引用者)

〈ソフトウェアライフサイクルモデル〉概念は、「3.11 ライフサイクルモデル(life cycle model)」において、次のように定義されています。

3.11 ライフサイクルモデル(life cycle model) ソフトウェア製品の開発、運用及び保守に伴う"プロセス"、"アクティビティ"および"タスク"の枠組み。この枠組みは、システムの要求事項の定義から、その利用終了までのシステムの生涯に及ぶ。

参考 "プロセス"、"アクティビティ"および"タスク"は、システム開発作業を階層化して表すものである。最上位の作業のくくりが"プロセス"、その"プロセス"の構成要素が"アクティビティ"、更に"アクティビティ"の構成要素が"タスク"である。(強調は引用者)

〈開発プロセス〉に関する説明(5.3.1.1)においても、〈ソフトウェアライフサイクルモデル〉との関係が規定されています。

契約で指定していない場合、開発者はプロジェクトの範囲、規模及び複雑さに適合したソフトウェアライフサイクルモデルを定義、又は選択する。さらに、この節で定義する開発プロセスの

19 「開発手法」とは、構造化手法、オブジェクト指向手法などのソフトウェア開発手法(技法)を意味すると考えられる。この概念は、河村一樹が「技法」(前掲書 p.45)、プリーガーが「パラダイム」(前掲書邦訳 pp.4-5)、UML の考案者のひとりであるイバー・ヤコブソンが「方法論」と呼んでいるものである。

アクティビティ及びタスクを選定し、定義又は選択したライフサイクルモデルに当てはめる。

備考　これらのアクティビティ及びタスクは、ライフサイクルモデルに当てはめたとき、数カ所に重複して現れていてもよいし、又は相互に影響し合う形であってもよい。また、繰り返したり循環的に実行してもよい。(強調は引用者)

〈プロセス〉(process)概念に関しては、次のように定義されています。

3.17 プロセス(process)　互いに関連をもった"アクティビティ"の集合で、入力を出力に変換するもの。
備考　用語"アクティビティ"には、資源を利用することも含まれる。

3.5 ここまでのまとめ

SWEBOK および ISO/IEC 12207:1995(JIS X 0160:1996)の記述をまとめると、〈ソフトウェアライフサイクルモデル〉、〈ソフトウェアライフサイクルプロセス〉概念は次のようにまとめることができます。

■ソフトウェアライフサイクルモデル

ソフトウェア製品のライフサイクル(〈ソフトウェアライフサイクル〉)には、複数の作業内容が含まれますが、それら作業内容が配置されるべき時系列モデル(一定の作業区分と作業区分間の関係を含む)を定義したものが〈ソフトウェアライフサイクルモデル〉です。ただし、詳細な作業区分、各作業区分に含まれる具体的な作業内容は規定していません。

以下、主要な〈ソフトウェアライフサイクルモデル〉です。

1) ウォーターフォールモデル(Waterfall Model)

システム開発の〈ソフトウェアライフサイクル〉に含まれる各フェイズ(phase:局面)を、先行するフェイズが完全に完了して承認を受けてから、はじめて後続するフェイズに進むという仕方で進行する最もオーソドックスな〈ソフトウェアライフサイクルモデル〉です。しかし、この場合、後続するフェイズに進んだ後に、先行するフェイズの作業に修正が発生してしまった場合、それ以降のフェイズの作業をやり直す必要があり、修正に多くの時間を消費する可能性があります。また、利用者は最後になってはじめて最終的な成果物に接するという大きな欠点があるため、環境やニーズの変化が激しい分野への適用は困難です。

ウォーターフォールモデルでは、一般的に、要件定義、外部設計(基本設計、概要設計)、内部設計(詳細設計)、実装、テスト、運用・保守といった作業区分が規定されています。以下の表は、おおまかな作業区分と一般的に割り当てられる作業内容です。この作業内容は、〈ソフトウェアライフサイクルモデル〉であるウォーターフォールモデルの中では決定できず、実際には、後述する〈ソフトウェアライフサイクルプロセス〉の内部で決定されます。

作業区分	一般的な作業内容

作業区分	一般的な作業内容
要件定義	システムに対して求められている品質の水準を確定する。つまり、必要な機能やパフォーマンスなどの条件を明確にし、要求仕様書を作成する。
外部設計	要求仕様書に基づき、システムのサブシステムへの分割、システムのユーザーインターフェイス設計(画面設計)、帳票設計、データベースの論理設計などを行う。
内部設計	外部設計の結果に基づき、サブシステムをさらに細かくモジュールに分割し、モジュールごとの機能やモジュール間の関係を設計する。データベースの物理設計を含みます。
実装	内部設計に基づき、プログラムの実装を行う。あわせて、データベースの作成を行う。
テスト	単体テストでは、モジュールごとに必要な機能を実装できているかを検証し、結合テストでは、複数のモジュールが正常に連携し、サブシステムが正常に機能できているかを検証する。さらに、システムテストでは、実際の運用環境を想定して、システムが要求仕様書どおりに動作するかどうかを検証する。
導入・運用	システムを運用環境に移行させ、利用する。

2) スパイラルモデル(Spiral Model)

開発の初期段階で、要求された機能を独立性の高いいくつかのサブシステムに分割し、サブシステムごとに、システム開発のライフサイクル(要件定義、外部設計、内部設計、実装、テスト)を実行します。ひとつのサブシステムのライフサイクルが完成すると、次のサブシステムに移行するというように開発してきます。したがって、最終的な完成段階に至る前に、各サブシステムの段階で動作を検証・修正することができます。

3) インクリメンタルモデル(Incremental Model)

最初にシステム全体の要件定義を行い、要求された機能を独立性の高いいくつかのサブシステムに分割し、それぞれのサブシステムを並行的段階的に開発し、リリースするモデルです。段階的な開発とは、システム開発のライフサイクル(外部設計、内部設計、実装、テスト)を複数回繰り返すことを意味します。したがって、最終的な完成段階に至る前に、それぞれのリリースの時点でシステムの動作を検証・修正することができます。

■ソフトウェアライフサイクルプロセス

〈ソフトウェアライフサイクルモデル〉は、おおまかな作業区分を含む時系列モデルを決定してはいますが、それに対して、そこに配置される詳細な作業区分・具体的な作業内容を決定するのが、〈ソフトウェアライフサイクルプロセス〉です。ISO/IEC 12207:1995(JIS X 0160:1996)では、〈プロセス〉は複数の関連する作業区分・作業内容の集合を意味し、その内部には、詳細な作業区分・具体的な作業内容となる〈アクティビティ〉が含まれ、さらに〈アクティビティ〉には〈タスク〉が含まれます。〈プロセス〉→〈アクティビティ〉→〈タスク〉の順に詳細化、具体化します。

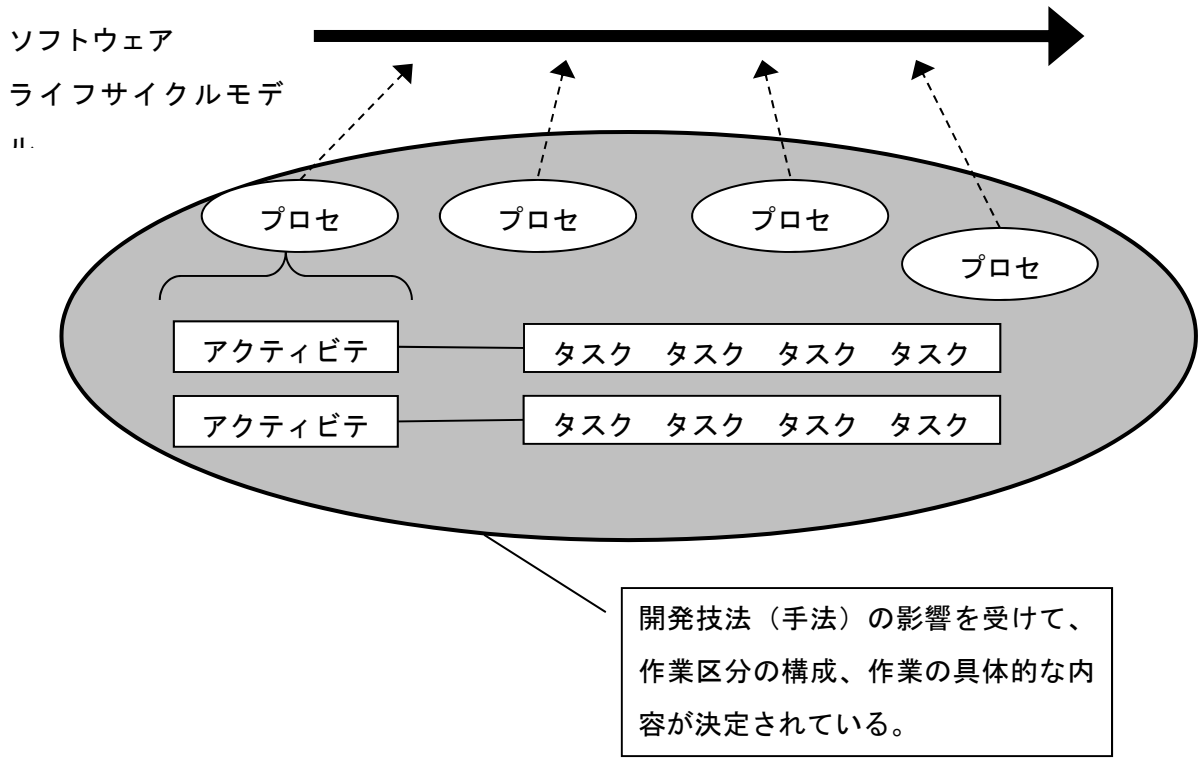
■開発技法(手法)

構造化技法やオブジェクト指向技法のような、詳細な作業区分・具体的な作業内容(〈プロセス〉・〈アクティビティ〉・〈タスク〉)を方向づける一定の考え方(概念モデル、論理モデル、実装モデルの統一的な思考方法)を意味します。

■具体的な開発・運用・保守方法論（実際の作業方法）

〈ソフトウェアライフサイクルモデル〉を選択し、そこに〈ソフトウェアライフサイクルプロセス〉の詳細な作業区分・具体的な作業内容（〈プロセス〉、〈アクティビティ〉、〈タスク〉）を当てはめたものが、実際の開発・運用・保守の方法論（実際の作業方法）になります。ただし、その〈プロセス〉、〈アクティビティ〉、〈タスク〉といった作業区分の構成、作業の具体的な内容は一定の開発技法に基づいて決定されています。

以上を踏まえて、実際の開発・運用・保守方法のイメージは次のようになります。



4 共通フレーム 2013

本章では、共通フレームの元となった ISO/IEC 12207(JIS X 0160)の目的や、共通フレームの必要性、そして設計思想など、共通フレームの概要について学ぶ。共通フレームはソフトウェア(システム)業界における特定の問題を解決する為に策定されたが、その設計の意図や、何を解決しようとしており、何について触れないのか、を理解する事で、よりその全体像を掴みやすくなる。次章以降に学ぶ、共通フレームにおける要件定義の概念や、要求工学、そしてその知識体系(REBOK)の理解を深める上でも、共通フレームの概要はしっかりと抑えておきたい。

4.1 ISO/IEC 12207：2008（JIS X 0160:2012）の目的

ISO/IEC 12207:2008(日本版は JIS X 0160:2012)が策定されたのは、ソフトウェアライフサイクルプロセスに関する共通認識の必要性が認識された為である。ソフトウェア開発は、ソフトウェアライフサイクルモデルにソフトウェアライフサイクルプロセスの詳細な作業区分・具体的な作業内容(一定の開発技法によって規定されたプロセス・アクティビティ・タスク)を当てはめて行われる。一定の認知度を得ている開発技法は、本質的な部分において共通項を持つ事が多いが、一方でプロセスにはどのような区分があり、そこにはどのような内容のアクティビティやタスクが含まれるのか、などという詳細に踏み込むと、それぞれが異なる思想を持ち、各種各様となっている。開発技法において違いがある以上、それを開発者が捉え実際に利用する段階になると、更にそれぞれの個人による解釈の違いが加わり多様性を増す。これらの認識の齟齬はじわじわとプロジェクトを蝕み最終的にはプロジェクトを頓挫に追い込むことすらある。例えば、開発の依頼者側と受託側の間で作業の認識が異なっていれば、お互いに見積もりや成果物の評価に食い違いが生じてしまうだろうし、受託側の内部で、どのプロセスにどのような作業(アクティビティ、タスク)が含まれるかの認識が共有されていなければ、共同作業に混乱が生じてしまう。

このような問題意識から、ソフトウェアライフサイクルプロセスに関して、世界共通の枠組みを考案し、開発の依頼側と受託側の間、あるいは依頼側、受託側それぞれの内部で、行われるべき作業の認識を一致させるための規格である、ISO/IEC 12207:2008(JIS X 0160:2012)が制定された。こうした規格化は、ソフトウェアライフサイクルプロセスの標準化と呼ばれ、ソフトウェア開発プロジェクトの規模増大、複雑化、そして多様化に伴い、徐々に浸透してきている。

4.2 ISO/IEC 12207 から共通フレームへの発展

国際規格である ISO/IEC 12207:1995 が発行された1年後、その完全翻訳版である JIS X 0160:1996 が発行された。更に JIS X 0160:1996 における規格を包括し、日本のソフトウェア産業界で必要とされるプロセスや作業項目を追加したものが共通フレームである。共通フレームは、、IPA の下に設置された「システム開発取引の共通フレーム検討委員会」により発行され、その後 ISO/IEC 12207 の改定に伴い、以下に示すような改定が行われてきている(現在の最新版は、「共通フレーム 2013」)。

■共通フレーム 2013 発行までの経緯

1989 年 ISO/IEC JTC1/SC7/WG7 で SLCP 規格検討開始
1994 年 SLCP 規格案を元に共通フレーム 94 発表
1995 年 ISO/IEC 12207:1995 発行
1996 年 ISO/IEC 12207:1995 を JIS 化し、JIS X 0160:1996 発行
1998 年 共通フレーム 98(SLCP-JCF98)発表
2002 年 ISO/IEC 15288:2002(システムライフサイクルに関する規格)発行、ISO/IEC 12207 追補 1 発行
2004 年 JIS X 0170:2004(ISO/IEC 15288 の翻訳)発行、ISO/IEC 12207 追補 2 発行
2005 年 情報処理推進機構ソフトウェア・エンジニアリング・センター(IPA/SEC)開発プロセス共有化部会『超上流』を発表
2007 年 JIS X 0160 追補 1(JIS X 0160:2007)発行
2007 年 10 月 JIS X 0160:2007 の原案に基づき共通フレーム 2007 発行
2008 年 3 月 ISO/IEC 12207:2008 に、「共通フレーム 2007」で取り決めた「契約の変更管理プロセス」が採用される
2009 年 10 月 JIS X 0160:2007 の正式版に基づき共通フレーム 2007 第 2 版発
2012 年 2 月 ISO/IEC 12207:2008 への改定に伴い、JIS X 0160:2012 発行
2013 年 JIS X 0160:2012 をベースとして共通フレーム 2013 発行

4.3 共通フレーム策定（ソフトウェアライフサイクルプロセスの標準化）の背景

共通フレームが策定された背景として、日本のソフトウェア（システム）開発に関わる産業界に以下のような問題が存在していたことが挙げられている（以下、『共通フレーム 2013』参照）。

■コミュニケーションチャネルの複雑さ

ソフトウェア（システム）開発に関わるステークホルダー（stakeholder：利害関係者）は、事業経営者、業務担当者、一次請け開発会社、二次請け開発会社、オフショア発注先、ヘルプデスク、コールセンター、個人ユーザー、企業ユーザーなど多岐に渡る。それぞれが、他のステークホルダーと連絡を取り合うようになると、コミュニケーションチャネルが複雑になり、作業連携に支障をきたす。

■用語とその意味内容の不統一

ソフトウェアライフサイクルに関して、たとえば、A 社では「システム分析」と呼ぶ工程を、B 社では「要件定義」、C 社では「システム計画」と呼んでいたり、A 社が「ユーザーインターフェイス設計」と呼ぶ工程を、B 社では「外部設計」、C 社では「システム概要設計」と呼んでいたりする事によって、お互い面と向かって打ち合わせをして、契約を結んでいたはずが、実は全く異なる認識で話していると言う事が起こり得る。

ここには、二つの問題が内在している。まず単純に用語が統一されていないということ。これによって、たとえば、ソフトウェア開発に関して上記 A 社と B 社から見積もりをとった場合、まったく用語が異なるため、見積額の明細を比較することが困難になる。さらに、対応するそれぞれの用語が決して同じ作業内容を意味しないということ。上記の例では、A 社の「システム分析」と B 社の「要件定義」では、対応するとはいうものの、実際にそこに含まれる作業内容が異なり、単純に見積もりの高低を比較したり、同一の作業内容を期待して契約を締結したりすることは困難となる。一定の工程を意味する用語を統一するだけでなく、それぞれの工程に含まれるべき一般的な作業内容を規定する必要がある。この問題は、ソフトウェア工学全般にわたる問題と不即不離の関係にあり、共通フレームが発行された今でも未だ解決されていない。

■役割分担の不明確

今日のソフトウェア開発（システム開発）では多くのステークホルダーが関与する。契約における

発注者（取得者）と受注者（供給者）という二者の間でさえ、企画、要件定義、開発、運用、保守のソフトウェアライフサイクルに関して、相互の役割が了解されていないことがある。とくに要件定義において、誰がどのようなレベルでソフトウェア（システム）要件定義を行うのかが不明確であり、それが原因で契約上のトラブルに至ることがある。詳細は後述するが、共通フレームでは、企画プロセスにおいて事業経営者が事業要件を、要件定義プロセスにおいて業務部門が業務要件を、開発プロセスにおいて情報システム部門がシステム要件を、それぞれ定義するものと規定している。

■ステークホルダ間での合意欠如・合意不十分の問題

プロジェクトを齟齬なく進めて行く為には、ソフトウェア（システム）開発の目的と背景、システム化の対象範囲、優先度に関して、複数のステークホルダー間で確実に合意に至っている必要がある。企画、要件定義、開発、運用、保守といったソフトウェアライフサイクルにおける作業についても同様である。しかし、現状では十分な合意に至らないまま、開発が着手されてしまうという問題が多発している。

■見積りの問題

一般的にシステム開発プロジェクトの発注者は、要件定義以前の段階で、プロジェクトの方向性がある程度固まった時点でプロジェクトの予算見積もりを行う。一方、その予算に合わせた受注企業の見積もりには様々な制限が設定されてしまう。予算に合わせた見積もり額のまま受注されるが、プロジェクトの進行に伴ってユーザーの要求が具体化されていくことにより、作業の追加が発生し、最終的には当初の受注額を超過してしまい契約上のトラブルに至ることがある。この問題を回避するためには、ソフトウェアファイルサイクルの進行段階に応じて、見積もりを多段階化することが指摘されている。

■仕様変更のルールの不備

ソフトウェア（システム）開発において仕様変更はもはや当たり前の現象となっている。その原因としては、ソフトウェア（システム）を巡る外部環境に生じた変化や、ステークホルダー間の契約時の合意内容が不明確であったことなど、多種多様なものが挙げられる。このような仕様変更に対して、必要に応じて契約の変更が必要な場合もあるが、仕様変更に対する処置が契約上明確化されな

い事が多く、プロジェクトの進行に問題をきたすことがある。

■情報システムの信頼性向上の問題

システムは、その利用環境や社会的影響度に応じて、要求される信頼性、堅牢性の水準が異なる。信頼性、堅牢性は、ソフトウェアの性質だけでなく、ハードウェア、設置環境、周辺設備、利用施設の選択など様々なプロジェクト上の決定から特徴づけられる為、プロジェクトの後半になって要求される信頼性、堅牢性の水準に齟齬があった事が判明すると、大きな手戻りの原因となりうる。一方、これらを規定する指標となるような規格が存在していない。

■業務全体を対象範囲として捉えられていない問題

企業の業務の中には、システム自体によって自動的に処理される業務、システムを利用する人間の業務、システムを利用せず人間だけで行われる業務の 3 種類がある。業務全体を見直す中で、それぞれの範囲を確定して行くこととなるが、システム化に注意を払うあまり、システムを利用する人間の業務、システムを利用せず人間だけで行われる業務の必要性やそれぞれに必要な準備が明確化されず、システムが完成しても、結局、業務全体が抱える問題が改善しなかったり、かえって業務環境を悪化させてしまう事すらある。したがって、システムを開発する際には、業務担当者が業務全体およびその改善の方向性を認識した上で、業務要件定義を行う必要がある。

他にも、「不明確な取引の問題」、「再利用の問題」、「ユーザービリティの問題」、「開発モデルの問題」、「プロジェクト任せという問題」といった問題が挙げられている（前掲書 pp. 5-13）。

4.4 共通フレームの目的

ソフトウェア産業界に対して、「共通の物差し」を提供することによって、国内におけるソフトウェア開発および取引を明確化し、さらにソフトウェア開発の国際取引における相互理解を容易にすることで、先に挙げた日本のソフトウェア業界に蔓延る問題を解決することが、共通フレームの目的である。

ここでいう「共通の物差し」とは、取引における契約者（取得者と供給者）の双方、あるいは、ソフトウェア（システム）開発に関与するすべての人々にとって、共通（かつ唯一）の参照となるもののこと。具体的には、取引の対象となるソフトウェア（システム）の企画、要件定義、開発、運用、保守における一般的な作業内容と取得者、供給者それぞれの役割と責任に関する規定。

「取引の明確化」とは、ソフトウェア（あるいはシステム）の受発注契約において、その取引内容が取得者、供給者の双方にとって、具体的かつ明確に規定されることを意味する。これにより、契約締結以降に契約時点で含まれなかった作業が発生しないこと、および、契約の第三者にとってもその取引内容が検証可能であることが実現できる。

4.5 共通フレームの対象範囲

共通フレームは、「業務分析、業務設計、及びソフトウェアを中心としたシステムの企画、要件定義、開発、運用、保守及びそれらにかかわる諸活動」をその対象とするものと規定されている。したがって、ハードウェアに関してはシステムの構成要素として、構成の検討、決定、導入、運用、保守に関するかぎりでこれを扱い、ハードウェア自体の開発には関与しないものとされている。

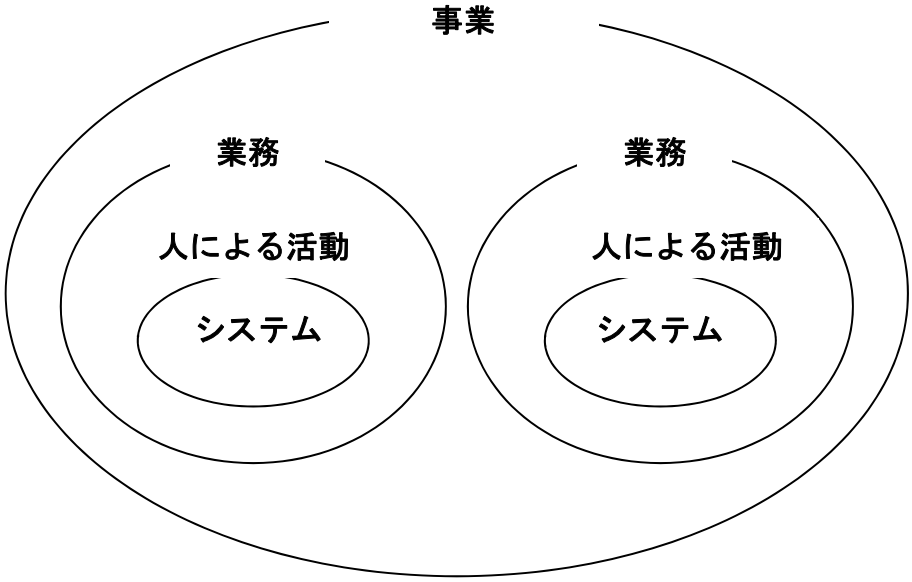
4.6 共通フレームの設計思想

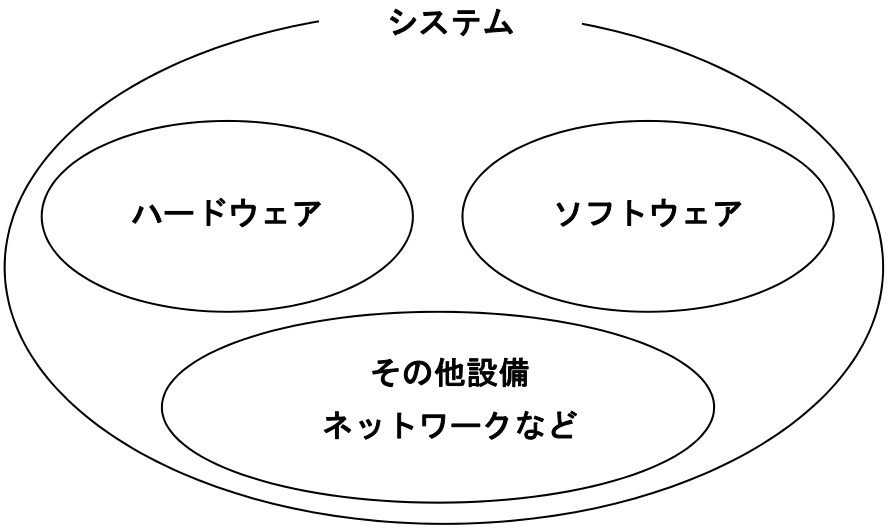
共通フレーム 2007 において提唱されている設計思想の概念として、システムの開発・利用に関わる組織構造を 4 つに分けて理解すると言う物がある。一方、共通フレーム 2013 では「5.1 共通フレーム作成における設計思想」と言う節において、共通フレームの骨格となる思想を 10 項目に渡り述べている。本節では共通フレーム 2007 で触れられていた 4 つの概念について詳説し、システムやソフトウェアと言った概念を理解した上で、共通フレーム 2013 において述べられる共通フレームの骨格となる思想 10 項目を紹介する。

■4つの階層の区別

共通フレームに含まれる「要件定義」、「システム」、「ソフトウェア」といった概念の意味を理解するために、共通フレームの設計思想を理解する必要がある。共通フレームの最も重要な設計思想の一つとして、システムの開発・利用に関わる組織構造を 4 つの階層に分けて考えるというものがある。この 4 つの階層は、「事業」、「業務」、「人による活動」、「システム」といった一連の概念に基づく。「事業」とは、企業が利益を取得するための手段として展開している一連の活動のことを意味し、「業務」とは、企業が事業を継続するために個々の実務担当者に担わせている活動のことを表す。

企業はその経営の具体的内容として一定の「事業」を展開している。例えば、トヨタ自動車株式会社の場合、自動車開発・生産事業、自動車レンタル事業、住宅事業、金融事業など多くの事業を手がけている。それぞれの「事業」の内部には、一定の具体的な作業から構成される様々な「業務」が含まれる。自動車開発・生産事業には、設計業務、生産業務、部品調達業務、生産管理業務、物流業務、販売業務などの業務が存在する。これらの「業務」には、今日では、人手による作業、つまり、「人による活動」によって達成される部分と、コンピュータの情報処理能力の総体である「システム」によって達成される部分の両方が含まれる。さらに、「システム」は、「ソフトウェア」、「ハードウェア」、ネットワークなどのその他設備から構成される。それぞれの関係性を次のページの図において示す。





これらの概念の階層構造から、ここでは、システムの開発・利用に関わる組織構造に 4 つの階層を見いだすことができる。最上位の階層として、「事業」の主体である「経営層」（事業主体である経営者・事業担当者が責任を担う層）、次に実際の「業務」を担う「業務層」（システムを直接利用する業務部門・業務担当者が責任を担う層）があり、さらにその下位に、「IT システム層」（IT システム部門・システム開発担当者が責任を担う層）、「ソフトウェア/ハードウェア層」（ソフトウェア、ハードウェアの構築担当者が責任を担う層）がある。

■4 つの階層それぞれの責任

この 4 つの階層の中で各層を担う者が、それぞれの層に応じた責任を負うものとされている。

・・・企画した内容（システム化の方向性、システム化計画）が運用に入ったのち、事業の観点から効果がでているかどうかを見るのは経営者や事業関係者である。同様に、業務運用の視点から要件定義（業務要件と機能要件・非機能要件のこと：引用者注）と運用テストの責任は業務部門にあることが分かる。システムレベルの設計とテストの責任を負うのはシステム開発担当者であり、ソフトウェアレベルの設計とテストはソフトウェア開発者がその責任を負う。
（『共通フレーム 2007 第 2 版』p. 22）

プロジェクトを起こした業務企画担当者は、プロジェクト責任者として、これらステークホルダの方針、意見、課題などについて、漏れなく綿密に把握し、できることとできないことを IT 担当者、ベンダとともに切り分け、業務要件として取りまとめていく責任を果たす必要があります。
（『経営者が参画する要求品質の確保 第 2 版』p. 9）

業務企画担当者は、構築しようとしているシステムの要件を明確に確定させ、承認する責任があります。また開発者に対しその要件を説明する説明責任があります。単なる説明、通り一遍の説明ではなく理解してもらう責任を負っています。
（『経営者が参画する要求品質の確保 第 2 版』p. 9）

ベンダは顧客企業の IT 担当者とともに、企業のビジネス戦略実現を担う存在です。共に顧客企業のシステム開発を担い、顧客企業に IT 担当者がいない場合は、担当者に代わって、その役割を引き受ける必要があります。加えて、顧客企業の IT 担当者が、特定の産業に特化した存在とは違い、ベンダはシステム開発のプロとして、その技術力、構築力を用いて、顧客企業の戦略実現を支える役割を強く担っているといえます。
（『経営者が参画する要求品質の確保 第 2 版』p. 15）

・・・事業を含めたおのおのの領域の仕様決定責任を見てみると、事業の領域は事業担当（経営者など）が、業務システムの領域は業務担当が、IT システムの領域は IT 担当（ベンダ企業も含む）が、それぞれ役割を担っているといえるでしょう。
（『経営者が参画する要求品質の確保 第 2 版』p. 25[20]）

20　独立行政法人情報処理推進機構ソフトウェア・エンジニアリング・センター, 『経営者が参画する要求品質の確保 第 2 版』, オーム社, 2005, p.9（この文献は、『共通フレーム 第 2 版』の中でソフトウェア開発における責任を論じている箇所で参照されている（『共通フレーム 第 2 版』p.22）。

■4 つの階層と要件定義の関係

『経営者が参画する要求品質の確保 第 2 版』では、4 つの階層との関係において、3 つの種類の要件定義の違いが説明されている。

名称	項目	内容
事業要件定義	ビジネスモデルの検討 等	新規事業/社外連携/組織改編/部門間業務分掌変更/現行業務踏襲、セキュリティなど
業務要件定義	業務モデルの検討 等	業務内容(手順、責任・権限など)、業務形態(ピークなど)、業務品質、性能目標、運用、移行要件、セキュリティなど
システム要件定義	システムモデルの検討 等	システム構成、業務アプリケーション(構造、DB・ファイル構造など)、運用、移行要件、セキュリティ、機密情報保護対策など

(『経営者が参画する要求品質の確保 第 2 版』p. 43)

4つの階層と要件定義の関係は、次のように言われている[21]。

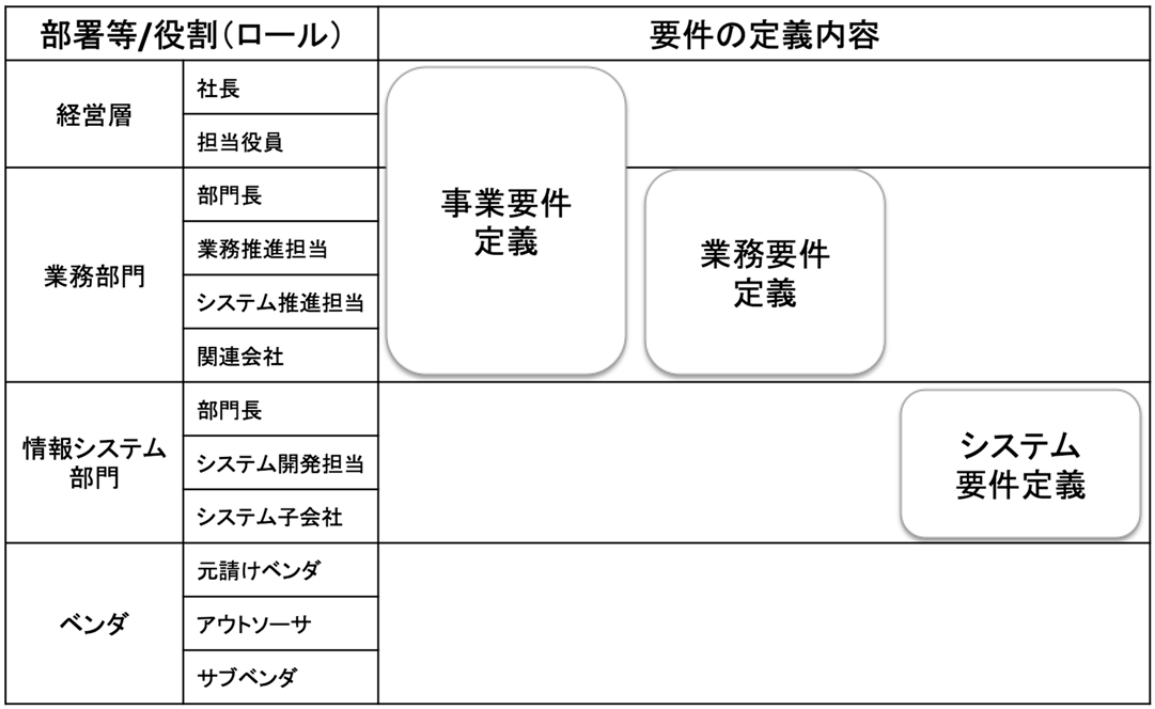
プロジェクトを起こした業務企画担当者(業務層の担当者のこと:引用者注)は、プロジェクト責任者として、これらステークホルダの方針、意見、課題などについて、漏れなく綿密に把握し、できることとできないことをIT担当者、ベンダとともに切り分け、業務要件として取りまとめていく責任を果たす必要があります。

21 上記一覧表が掲載されている箇所では、4 つの階層と要件定義に関して、事業要件定義を経営層と業務層 (業務部門) が、業務要件定義を業務層が、システム要件定義を IT システム層 (情報システム部門) が担うとされており、事業要件定義は経営層と業務層にまたがって担われるものとして解説されています (『経営者が参画する要求品質の確保 第 2 版』 pp.42-43)。

（『経営者が参画する要求品質の確保 第 2 版』p. 9 強調は引用者による）

・・・事業を含めたおのおのの領域の仕様決定責任（要件定義に関する責任のこと:引用者注）を見てみると、事業の領域は事業担当（経営者など）が、業務システムの領域は業務担当が、IT システムの領域は IT 担当（ベンダ企業も含む）が、それぞれ役割を担っているといえるでしょう。
（『経営者が参画する要求品質の確保 第 2 版』p. 25 強調は引用者による）

したがって、経営層を担う者が、「企画プロセス」の内部で定義される「事業要件」（経営・事業戦略、経営・事業目標、情報戦略などからなる条件）に対して責任を負い、業務層を担う者が、「要件定義プロセス」の内部で「業務要件」（業務において実現すべき機能）の定義に対して責任を負い[22]、IT システム層およびソフトウェア/ハードウェア層を担う者が、「開発プロセス」の内部で定義される「システム要件」（システムが満たすべき条件）、「ソフトウェア要件」（ソフトウェアを満たすべき条件）に対してそれぞれ責任を負うものと考えられる。部署/役割と要件定義の関係を以下の図に示す。
（『共通フレーム 2013』p. 343 より引用）



22 要件定義プロセスの段階で、システムおよびソフトウェアに対する機能要件・非機能要件も定義する。したがって、要件定義プロセスは、開発プロセスに含まれるシステム要件定義、ソフトウェア要件定義といったアクティビティをここで先取りしてもよいとされている（『共通フレーム 2007 第 2 版』pp.112-113）。

■4 つの階層についてのまとめ

これまでの内容を踏まえて、4 つの階層と、該当職位、要件定義、ソフトウェアライフサイクルプロセスの全体的な関係を以下の表にまとめる(『共通フレーム 2007 第 2 版』pp.22,291)[23]。

23 該当職位については、共通フレーム内であまり明確に規定されていない。とくに、いわゆる下流工程のアクティビティに関して顕著である。これは、階層と該当職位の間に経験的要因が介在する余地があること、および、共通フレームが「超上流」を重視していることに起因すると思われる。そのため、上記表における該当職位の対応付けは、一般的かつ経験的な業界認識に基づいたものである。

階層項目	階層名	主要該当職位	定義される要件	責任があると考えられるプロセス
事業	経営層	経営者・事業担当者	事業要件	企画プロセス （超上流）
業務	業務層	業務担当者 （部長、課長、係長など業務現場の担当者）	業務要件 機能要件 非機能要件	要件定義プロセス （超上流）
システム	IT システム層	情報システム部門のシステム開発担当者 ベンダー側アーキテクト、ベンダー側システムエンジニア	システム要件	開発プロセス （上流）
ソフトウェア・ハードウェア・その他設備（ネットワークなど）	ソフトウェア層 ハードウェア層 その他の設備層	ベンダー側プログラマー ベンダー側ネットワークエンジニアなど 現実的には、情報システム部門のシステム開発担当者、ベンダー側アーキテクト、ベンダー側システムエンジニアも関与する	ソフトウェア要件	開発プロセス （下流）

機能要求と非機能要求

システムおよびソフトウェアに対する「要求」(requirements:「要件」という訳語もある）は、二種類に大別することができる。ひとつは、「機能要求」(functional requirements) であり、他方は、「非機能要求」(nonfunctional requirements) である。これらふたつの概念も、研究者やエンジニアによって微妙に異なるため、ここでは、SWEBOK ガイド 2004 の説明を参照しておく。それによれば、機能要求はソフトウェアが実行する機能についての要求を意味し、非機能要求は「ソリューションを制限するように作用する要求」のことであり、「性能要求、保守要求、安全性要求、信頼性要求やそのほかの多くの種類のソフトウェア要求に分類される」("2004 SWEBOK Guide", CHAPTER 2 SOFTWARE REQUIREMENTS, 1.3. Functional and Nonfunctional Requirements：拙訳)。

業務の遂行に直接利用されるシステムやソフトウェアの機能に関する要求、システムやソフトウェアが持つべき機能に関する要求が機能要求であり、ソフトウェア品質特性と関連して、システムやソフトウェアの品質に対する要求（品質要求）が非機能要求である。

【注意事項】

共通フレームでは、「共通フレーム上の役割と現実の組織/役割の対応が m:n（多対多）の関係である」と規定されており、各プロセスと現実の組織/役割が固定的に結び付けられているわけではないとされている（『共通フレーム 第 2 版』 p.23)。したがって、企画プロセスを担う「企画者」や、要件定義プロセスを担う「要件定義者」の役割（責任）を、経営層、業務層、IT システム層など、複数の層の担当者が果たしてもよいことになっている。この場合、経営層だから事業要件定義に、業務層だから業務要件定義に責任があるとはいえなくなる。

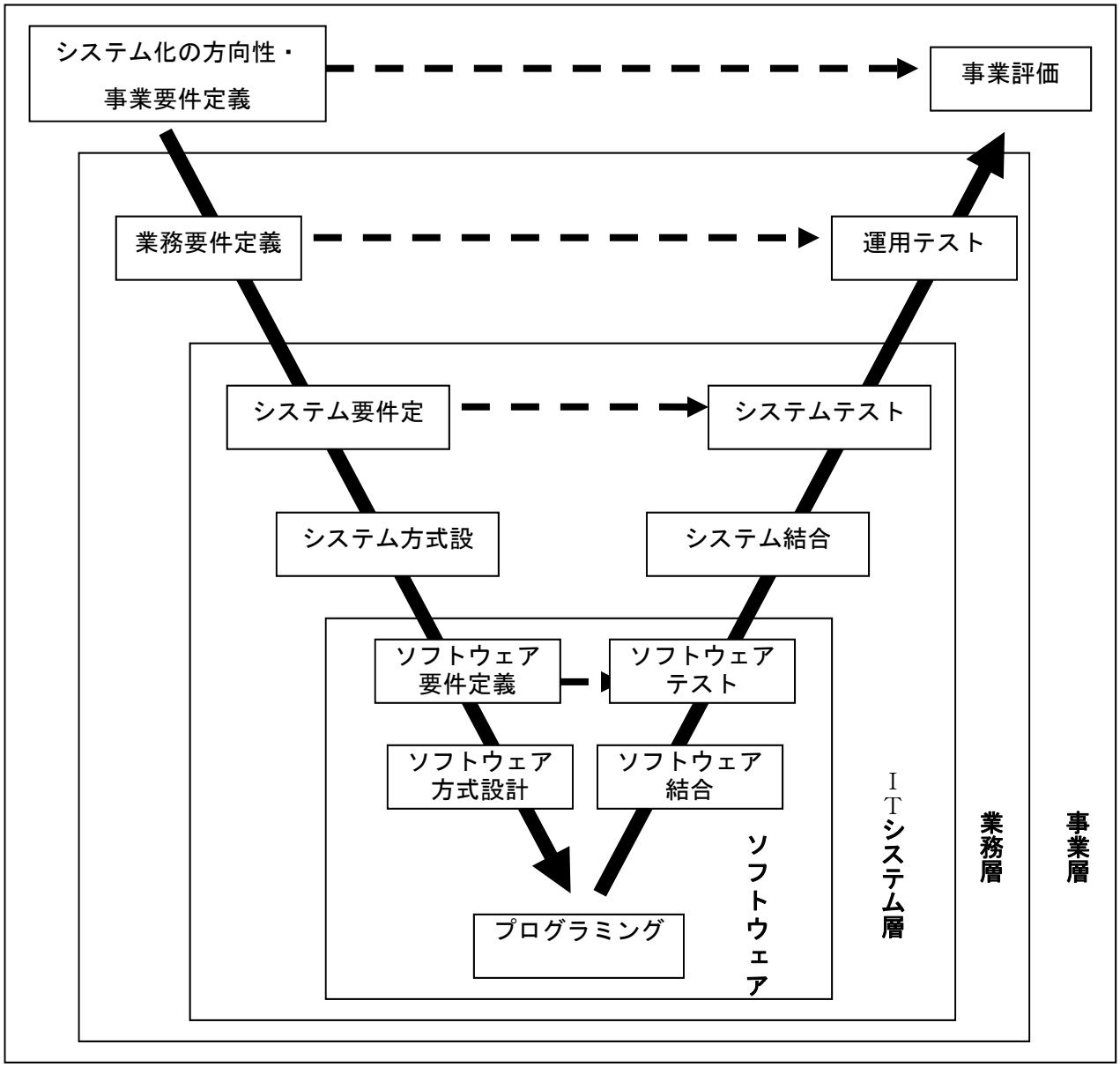
この点で、共通フレームにおける役割・責任に関する議論は、曖昧であるように思われる。いったん、階層ごとの役割（責任）を規定した後で、階層とは分離した「企画者」、「要件定義者」といった論理的な主体（役割を担う者）を定義し、そこに責任を帰着させている。さらに、『共通フレーム 第 2 版』 pp.290-291 では、業務層（「業

務部門」）が、事業要件定義にも関わるように「補足説明」されている。役割（責任）に関する議論に混乱が見られるように思われる。

したがって、前ページの表において、「責任があると考えられるプロセス」とあるのは、あくまで、『共通フレーム 第 2 版』p.22 の責任に関する規定から展開した結果である。

■4つの階層とアクティビティの関係

4つの階層の責任とソフトウェアファイルサイクル内のプロセス、アクティビティとの関係は、次のようなV字のモデルによって表現されています。



このV字のモデルでは、事業層、業務層、ITシステム層、ソフトウェア層の4層に関して、V字の左の辺の作業を担う主体が、そのテスト、検証、評価を担う主体でもあるべきだという考え方を表現している。たとえば、事業を担う経営者は、システム化の方向性や事業要件定義に対して責任があると同時に、完成したシステムを運用し、事業における最終的な効果を評価することに対して責任がある。同様に、業務を担う業務担当者は、業務の全体を見渡し、そこから業務要件を定義するこ

とに対して責任があると同時に、業務の中でシステムを実際に動作させる業務テストの実施に対して責任がある。

■ 共通フレーム 2013 における骨格となる 10 項目の思想

共通フレームは上述の通り ISO/IEC 12207(JI X 0160)規格作成時の基本原則に則り作成されている。ISO/IEC 12207 の中で述べられている 9 つの項目(「モジュール性の採用」、「責任の明確化」、「責任範囲の明確化」、「工程、時間からの独立性」、「開発モデル、技法、ツールからの独立性」、「ソフトウェアを中心としたシステム関連作業までを包含」、「システムライフサイクルプロセスとの整合性」、「文書の種類、書式を規定しない」、「テーラリング(修正)の採用」)に加え、共通フレーム 2007 から採用された「超上流の重視」と言う 10 項目が共通フレームの根幹を成している。以下にそれぞれを詳説する。

1. 超上流の重視

システムの品質を確保するためには、開発段階に入る前の超上流工程において、ユーザー内の役割分担(経営層、業務部門、情報システム部門)のもとに、業務要件ならびにユーザーがシステムに求める要件(機能要件、非機能要件)を明確に定義する責任がある。共通フレームでは、超上流工程として企画プロセス、要件定義プロセスそして契約の変更管理プロセスを追加した。超上流プロセスは、「システム化の方向性」、「システム化計画」、「要件定義」の三つの段階から構成されている。

2. モジュール性の採用

共通フレームの各プロセスは、必要に応じて組み合わせて実施が可能なようにモジュール化されて提唱されている。これらのプロセスは「強い密着」と「弱い結合」と言う 2 つの特徴を持つ形でモジュール化されている。

「強い密着」とは、1 つのプロセスに含まれるアクティビティ及びタスクの関係性が高いに強い関係で結ばれている事を表す。1 つのプロセスが特定の機能のみを果たすように設計される事で、それを構成するアクティビティ及びタスクの関連は「強い密着」関係となる。共通フレームで提唱されるプロセスはそれぞれのアクティビティとタスクが強い関係で結ばれている事が保証されている。

「弱い結合」とは、プロセスとプロセスの互いの依存度が低い事を表す。例えば、システム開発プロセスと運用・サービスプロセスはそれぞれに「弱い結合」で結ばれている。これは、運

用・サービスプロセスの実行中にソフトウェア開発プロセスが実行されない。もし、ソフトウェアの修正が必要になった場合は保守プロセスを呼び出す事となる。このように、共通フレームで提唱されるプロセスは、それぞれが互いに素の関係を取り、プロセス間のインタフェースが最小になるように設計される事が保証されている。

3. 責任の明確化

共通フレームでは、取得プロセスは「取得者」、規格プロセスは「企画者」のように、プロセスごとに作業の主体者（役割）を定義することで、その責任の所在を明確化する。これらの主体者の定義を議論で用いる事で、例えば発注者と受注者が協議を行う時にも、取得者にはどちらが何人の人を割当るかなどを明確にする事が可能となる。それによって、契約が済んだ後に、あるプロセスを誰が担当すべきかわからなくなり、工数の見積もりが正確でなくなってしまうなどの問題を回避する事が出来る。

4. 責任範囲の明確化

先の「4つの階層とアクティビティの関係」でも述べたように、共通フレームでは規格から要件定義、開発、運用テスト、運用、評価までの一連の流れを、設計とテストを対応させ V 字で表現し、そこに事業、業務、システム、ソフトウェアの4つの階層を割り当てる事で、それぞれの役割がどの階層を担当すべきかを明確にしている。更に詳細に責任範囲を明確化する為、共通フレームでの各プロセスにおけるタスクの説明は、上述の「責任の明確化」でも述べられた、それぞれのプロセスにおける主体者を中心に述べられており、「取得者は、～を行う」と言った形で明記されている。

5. 工程、時間からの独立性

共通フレームに含まれる「プロセス」、「アクティビティ」、「タスク」の記述は、特定の開発方法論の内部で決定された作業順序や一定の時間順序に従わない形で定義されている。共通フレームの文言を引用すれば、これらは、「最も一般的で自然な順序で配置されているだけである」とされている。したがって、共通フレームに含まれる「プロセス」間の関係は、作業順序を表現するものではなく、また「プロセス」概念は、それぞれのプロセスに関わる主体が担うべき作業内容をまとめたものとなる。

6. 開発モデル（ソフトウェアライフサイクルモデル）、技法、ツールからの独立性

実際のソフトウェア（システム）開発は、一定の開発モデル、技法、ツールを利用して進められます。しかし、共通フレームは、特定の開発モデル（ウォーターフォールモデル、インクリメンタルモデルなど）、特定の技法（構造化技法、オブジェクト指向技法）、特定のツール（各種

case ツールなど)に依存しない抽象的な規定です。したがって、実際のソフトウェア開発に対しては、共通フレームの「適用」が必要になります。

7. ソフトウェアを中心としたシステム関連作業までを包含

共通フレームでは、先の「4つの階層」における説明でも述べたように、「事業」、「業務」、「人による活動」、「システム」といった一連の概念に基づき、プロセスを定義している。一般的にソフトウェア工学における標準化の試みにおいては、ソフトウェアやその開発プロセスのみを扱う事が多いが、実際のシステム開発プロジェクトはソフトウェア及びその開発のみを考慮すれば良い訳ではない。ソフトウェアライフサイクルプロセスの国際規格を元に設計がなされてはいるものの、ソフトウェアを取り巻く、システム関連作業までを扱わなければ、真の意味で、実際のシステム開発プロジェクトを成功へと導く事は出来ないと言う思想の元共通フレームは設計されている。

8. システムライフサイクルプロセスとの整合性

「ソフトウェアを中心としたシステム関連作業までを包含」とも重なるが、共通フレームの元となる、ISO/IEC 12207 はソフトウェアライフサイクルプロセスの国際規格であり、基本的にはソフトウェアライフサイクルにおけるプロセスを扱う。しかし、ISO/IEC 12207 は基本的に、システムライフサイクルプロセスの国際規格である ISO/IEC 15288 と共に改訂されており、システムライフサイクルにおける各プロセスと整合性が保たれる形で設計がなされている。

9. 文書の種類、書式を規定しない

共通フレームでは、特定のドキュメントの種類や書式などの詳細については規定しない。これは、ドキュメントを規定することで、その文書化及びそれに連なる作業を縛ることになる為である。例えば「システム設計書」と言ってしまえば、既にその中で使われている項目や言葉によって作業内容が自ずと決まってしまう。

10. 修整(テーラリング:tailoring)の採用

共通フレームを「適用」する、つまり、一定の開発モデルに適用する際には、アクティビティやタスクを取捨選択したり、繰り返し実行したり、複数のものを一つにまとめても良いとされている。これらプロジェクトやプロジェクトで使用される開発モデルに合わせて行われる修整作業を「テーラリング」と呼ぶ。

■共通フレームの適用手順

共通フレーム自体は抽象的な規定であるため、これを実際のソフトウェア開発に適用するということは具体的な内容へと肉付けするということになります。この場合、次のような作業が含まれます。

1）プロセスの担当者を具体化する

企画プロセスは誰が実際に担当するのか、要件定義プロセスはといったように、それぞれの具体的な担当者（担当部署）を決定します。

2）ソフトウェアライフサイクルモデル、開発技法、ツールの選択

ウォーターフォールモデル、スパイラルモデル、インクリメンタルモデルなどのソフトウェアライフサイクルモデルを選択し、そこにプロセス、アクティビティ、タスクを割り当てる。また、アクティビティやタスクの実際の作業内容は、選択された開発技法（構造化技法、オブジェクト指向技法など）により決定される。

3）テーラリング

アクティビティやタスクは、選択されたソフトウェアライフサイクルモデルおよび開発技法に合わせて、取捨選択、反復、一括化といったテーラリング（修整）を行う。

以上の作業により、ソフトウェアライフサイクルモデルにしたがって、アクティビティ、タスクが配置され、実際の作業工程、作業順序が決定されます。

4.7 共通フレーム 2013 に含まれるプロセス・アクティビティ

「共通フレーム 2013」には、以下のソフトウェアライフサイクルプロセスが含まれます。

1. 主ライフサイクルプロセス

[契約関連のプロセス群」

1．1 取得プロセス

1．1．1 取得の準備

1．1．2 取得の通知

1．1．3 供給者の選定

1．1．4 契約の合意

1．1．5 合意の監視

1．1．6 取得者の受入れ

1．1．7 取得プロセスの終了

1．2 供給プロセス

1．2．1 供給機会の判別

1．2．2 供給者の提案依頼

1．2．3 契約の合意

1．2．4 契約の実行

1．2．5 製品・サービスの納入及び支援

1．2．6 供給プロセスの終了

1．3 契約の変更管理プロセス

1．3．1 プロセス開始の準備

1．3．2 契約の変更依頼

1．3．3 影響の調査及び分析

1．3．4 協議の実施と合意の形成

1．3．5 契約の修正

[システム開発関連のプロセス群]

2．1 企画プロセス

2．1．1 システム化構想の立案プロセス

2．1．1．2 システム化構想の立案

経営上のニーズ、課題の確認
事業環境、業務環境の調査分析
現行業務、システムの調査分析
情報技術動向の調査分析
対象となる業務の明確化
業務の新全体像の作成
対象の選定と投資目的の策定

2．1．1．3 システム化構想の承認
システム化構想の文書化と承認
システム化推進体制の確立

2．1．2 システム化計画の立案プロセス
2．1．2．1 プロセス開始の準備
2．1．2．2 システム化計画の立案
システム化計画の基本要件の確認
対象業務の内容の確認
対象業務のシステム課題の定義
対象システムの分析
適用情報技術の調査
業務モデルの作成
システム化機能の整理とシステム方式の策定
システム化に必要な付帯機能、付帯設備に対する基本方針の明確化
サービスレベルと品質に関する基本方針の明確化
プロジェクトの目標設定
実現可能性の検討
全体開発スケジュールの作成
システム選定方針の策定
費用とシステム投資効果の予測
プロジェクト推進体制の策定
経営事業戦略、情報戦略、システム化構想との検証
2．1．2．3 システム化計画の承認
システム化計画の策定と承認
プロジェクト計画の作成と承認

2．2 要件定義プロセス
2．2．1 プロセス開始の準備

要件定義作業の組み立て
必要なプロセスの組み込み
要件合意及び承認ルールの決定
要件定義環境の準備
要件定義プロセス実施計画の作成

2．2．2　利害関係者の識別

2．2．3　要件の識別

要件の抽出
制約条件の定義
利用者とシステム間の相互作用の識別
システムの使用が周辺に及ぼす影響への対処

2．2．4　要件の評価

導出要件分析

2．2．5　要件の合意

要件の問題解決
利害関係者へのフィードバック
要件の確立

2．2．6　要件の記録

要件の記録
要件の追跡可能性の維持

2．3　システム開発プロセス

2．3．1　システム開発プロセス開始の準備

開発作業の組み立て
必要なプロセスの組み込み
開発環境の準備
非納入品目の使用の容認

2．3．2　システム要件定義プロセス

2．3．2．1　システム要件の定義

2．3．2．2　システム要件の評価及びレビュー

2．3．3　システム方式設計プロセス

2．3．3．1　システム方式の確立

システム最上位レベルでの方式確立
利用者文書の作成

システム結合のためのテスト要件の定義

2．3．3．2　システム要件の評価及びレビュー

システム方式の評価

システム方式設計の共同レビューの実施

2．3．4　実装

2．3．5　システム結合プロセス

2．3．5．1　システム結合

2．3．5．2　テスト準備及びシステム結合の評価

2．3．6　システム適格性確認テストプロセス

2．3．6．1　システム適格性確認テスト

2．3．7　システム導入プロセス

2．3．7．1　システム導入

2．3．8　システム受け入れ支援プロセス

2．3．8．1　システム受け入れ支援

2．4　ソフトウェア実装プロセス

2．4．1　ソフトウェア実装プロセス開始の準備プロセス

2．4．2　ソフトウェア要件定義プロセス

2．4．3　ソフトウェア方式設計プロセス

2．4．3．1　ソフトウェア方式設計

ソフトウェア構造とコンポーネントの方式設計

各インタフェースの方式設計

データベースの最上位レベルの設計

利用者文書の作成

ソフトウェア統合のためのテスト要求事項の定義

ソフトウェア方式設計の共同レビューの実施

2．4．4　ソフトウェア詳細設計のプロセス

2．4．4．1　ソフトウェア詳細設計

ソフトウェアコンポーネントの詳細設計

ソフトウェアインタフェースの詳細設計

データベースの詳細設計

利用者文書の更新

ソフトウェアユニットのテスト要件の定義

ソフトウェア統合のためのテスト要件の更新

ソフトウェア詳細設計及びテスト要件の評価

ソフトウェア詳細設計の共同レビューの実施

2．4．5 ソフトウェア構築プロセス

2．4．5．1 ソフトウェア構築

ソフトウェアユニットとデータベースの作成及びテスト手順とテストデータの作成

ソフトウェアユニットとデータベースのテストの実施

利用者文書の更新

ソフトウェア結合テスト要件の更新

ソフトウェアコード及びテスト結果の評価

2．4．6 ソフトウェア結合プロセス

2．4．6．1 ソフトウェア結合プロセス

ソフトウェア結合計画の作成

ソフトウェア結合テストの実施

利用者文書の更新

ソフトウェア適格性確認テストの準備

ソフトウェア結合テストの評価

ソフトウェア結合の共同レビュー実施

2．4．7 ソフトウェア適格性確認テストプロセス

2．4．7．1 ソフトウェア適格性確認テスト

ソフトウェア適格性確認テストの実施

ソフトウェアの評価

ソフトウェア適格性確認テストの共同レビューの実施

利用者文書の更新

監査の支援

各納入ソフトウェア製品の準備

2．4．8 ソフトウェア導入プロセス

2．4．8．1 ソフトウェア導入（インストール）計画の作成

2．4．8．2 ソフトウェア導入の実施

2．4．9 ソフトウェア受け入れ支援プロセス

2．4．9．1 ソフトウェア受け入れ支援

取得者の受け入れレビューと受け入れテストの実施

ソフトウェア製品の納入

取得者への教育訓練及び支援

2．5 ハードウェア実装プロセス

2．6　保守プロセス

2．6．1　プロセス開始の準備

2．6．1．1　保守に必要な成果物の引き継ぎ

2．6．1．2　計画及び手続きの準備

2．6．1．3　問題管理手続きの確立

2．6．1．4　修復作業の管理

2．6．1．5　保守のための文書作成

2．6．2　問題把握及び修正の分析

2．6．2．1　問題報告又は修正依頼の分析

2．6．2．2　問題の再現又は検証

2．6．2．3　修正実施の選択肢の容易

2．6．2．4　文書化

2．6．2．5　修正案の承認

2．6．3　修正の実施

2．6．3．1　分析と修正部分の決定

2．6．3．2　修正の実施

2．6．4　保守レビュー及び受け入れ

2．6．4．1　修正システムのレビュー

2．6．4．2　完了の承認

2．6．4．3　保守のための文書の更新

2．6．5　運用テスト及び移行の支援

2．6．5．1　運用テストの実施支援

2．6．5．2　移行の実施支援

3．1　運用プロセス

3．1．1　運用の準備

3．1．2　運用テスト及びサービスの提供開始

3．1．3　業務及びシステムの移行

3．1．4　システム運用

3．1．5　利用者教育

3．1．6　業務運用と利用者支援

3．1．7　システム運用の評価

3．1．8　業務運用の評価

4.8 開発プロセスのソフトウェアライフサイクルモデルに対する適用

ここでは、共通フレームの開発プロセスを、最も一般的なソフトウェアライフサイクルモデルであるウォーターフォールモデルに対して適用する事例を紹介する(前掲書 pp. 275-276, 278)。

■開発プロセスの適用結果

一般的な工程の名称	共通フレームにおける開発フェーズの名称[24]	アクティビティ	一般的な担当職種	一般的な作業内容
分析/設計	システム・ソフトウェア要件定義	システム要件定義 システム方式設計[25] ソフトウェア要件定義	アーキテクト システムエンジニア	システム化する対象(問題領域:ドメイン)の特性や構造、顧客のニーズを把握します。さらに、ソフトウェア・ハードウェアの選定など、顧客のニーズを充足させるのに必要な、システム全体およびソフトウェアの基本的仕様と構造を決定します。
設計	基本設計 (外部設計)[26]	ソフトウェア方式設計	アーキテクト システムエンジニア	ソフトウェア全体の構造の決定および実際の操作画面の設計や印刷機能の設計、データベースの概念・論理設計など、ここでの成果物に基づいて、見積もりも行われます。

24 「開発フェーズ」という用語は、前掲書 p.276 および巻末の資料(図 5-1)に掲載されている用語であり、共通フレーム 2007 自体が規定している概念ではないという注意書きが付されている。ソフトウェアファイルサイクルモデルに含まれる段階を表す一般的名称として採用されていると思われる。

25 一般的には、「システム方式設計」は、分析から分離され、設計に含まれると考えるほうが自然に思われるが、『共通フレーム 2007 第 2 版』では、「システム・ソフトウェア用件定義」に分類されている。

26 前掲書の「基本設計」、「詳細設計」という概念は、一般的に、それぞれ、「外部設計」、「内部設計」という概念と同義である。

一般的な工程の名称	共通フレームにおける開発フェーズの名称24	アクティビティ	一般的な担当職種	一般的な作業内容
設計	詳細設計 （内部設計）	ソフトウェア詳細設計	システムエンジニア	ソフトウェアに含まれるモジュールの内部構造、データベースの物理設計などの、モジュールの実装に必要な細部にわたる設計を行います。
実装	実装・単体テスト	ソフトウェアコード作成およびテスト	プログラマー	設計に従い、プログラミングやデータベースへのデータのセットアップなどを行います。同時に単体テストを行います。
テスト	結合テスト	ソフトウェア結合 ソフトウェア適格性確認テスト システム結合 システム適格性確認テスト ソフトウェア導入	アーキテクト システムエンジニア プログラマー テスター	結合テストにより、プログラム中のバグの検出と改善を行います。
移行	移行・運用準備	ソフトウェア導入 ソフトウェア受け入れ支援	プログラマー	テストに合格した場合、システムを実際の業務へと導入します。

一般的な工程の名称	共通フレームにおける開発フェーズの名称24	アクティビティ	一般的な担当職種	一般的な作業内容
運用・保守	運用・保守	該当なし（運用プロセス、保守プロセスになるため）	プログラマー 運用・保守エンジニア	システムの維持や、利用中に発生した新たな要求の反映や不具合の除去を行います。ソフトウェアの保守は初期開発よりも長期に渡り、繁雑な作業となるのが一般的です。初期開発における設計・実装作業を行った者（開発会社）と保守作業を行う者（開発会社）が異なる場合、既存のソフトウェアの設計全体やが動作の仕組みを理解するだけでも多大な労力を必要とします。しかも、本来の設計に反するコードを追加しなければならないことも頻繁に生じるため、初期開発時のドキュメントと、実際のソフトウェアの設計・実装が食い違うといった事態を招くことが多くなります。

5 要件定義の意味

昨今、多くのソフトウェア開発プロジェクトにおいて、遅延や工数の超過など、プロジェクトが開始前の見積もり通りに終わらないケースが報告されている。そのような中、プロジェクトにおける失敗要因に関する調査を行うと必ず上位に挙がってくるのが、要件定義における失敗である。本章では、ソフトウェア開発プロジェクトにおける調査を元に、要件定義における問題を明確化、前章で学んだ共通フレームにおける要件定義の概念を理解する事で、要件定義のソフトウェア開発プロセスにおける立ち位置を捉え、そして最後に要求工学とその知識体系である REBOK を学ぶ事で、次章以降で詳説される「要件定義」実践の方法論である USDM を学ぶ為の基礎を身につける。

5.1 共通フレームにおける要件定義

前項では、要求定義での失敗が理由となりプロジェクトが工数・コスト・品質の観点から見積もり通りに終了が出来なかったと言う事実を紹介し、また要件定義が何故難しいのかと言った少し抽象的な概念から要件定義について触れた。本項では、前章にて学んだ共通フレームにおける、要件定義が実際にどのようなプロセスから成り立つかを学ぶ事で、要件定義のソフトウェア開発工程における位置付けを理解する。

共通フレームでは、ソフトウェアの開発にかかわる主要なプロセス、アクティビティとして、以下の事柄を含んでいた。

2．1 企画プロセス
- 2．1．1 システム化構想の立案プロセス
- 2．1．2 システム化計画の立案プロセス

2．2 要件定義プロセス
- 2．2．1 プロセス開始の準備
- 2．2．2 利害関係者の識別
- 2．2．3 要件の識別
- 2．2．4 要件の評価
- 2．2．5 要件の合意
- 2．2．6 要件の記録

2．3 システム開発プロセス
- 2．3．1 システム開発プロセス開始の準備
- 2．3．2 システム要件定義プロセス
- 2．3．3 システム方式設計プロセス
- 2．3．4 実装
- 2．3．5 システム結合プロセス
- 2．3．6 システム適格性確認テストプロセス
- 2．3．7 システム導入プロセス
- 2．3．8 システム受け入れ支援プロセス

2．4 ソフトウェア実装プロセス
- 2．4．1 ソフトウェア実装プロセス開始の準備プロセス
- 2．4．2 ソフトウェア要件定義プロセス
- 2．4．3 ソフトウェア方式設計プロセス

2．4．4 ソフトウェア詳細設計のプロセス
2．4．5 ソフトウェア構築プロセス
2．4．6 ソフトウェア結合プロセス
2．4．7 ソフトウェア適格性確認テストプロセス
2．4．8 ソフトウェア導入プロセス
2．4．9 ソフトウェア受け入れ支援プロセス

企画プロセスでは、事業要件が定義され、それに基づいてシステム化の構想・計画が立案される。それに続く要件定義プロセスでは、事業要件を満たすべき業務要件、さらに、業務要件を満たすために必要なシステム・ソフトウェアの機能要件・非機能要件が定義され、ステークホルダー間での合意・記録がとられる。システム開発プロセスでは、業務要件定義・機能要件定義・非機能要件定義を満たすようなシステム要件が定義され、システム要求仕様書が作成される。更に、その仕様書に基づきシステムの大枠の実装が行われる。ソフトウェア実装プロセスでは、同様にソフトウェア要求仕様書が作成され、実際にソフトウェアの構築や導入、受け入れ支援などが行われる。共通フレーム2007からの大きな変更点として、システム開発プロセス及びソフトウェア実装プロセスにおいて、要求仕様書を作成するところまでではなく、実装までをプロセスに取り入れた点があげられる。

事業要件定義、業務要件定義、システム要件定義の違いを以下の表に示す。

名称	項目	内容
事業要件定義	ビジネスモデルの検討 等	新規事業/社外連携/組織改編/部門間業務分掌変更/現行業務踏襲、セキュリティなど
業務要件定義	業務モデルの検討 等	業務内容(手順、責任・権限など)、業務形態(ピークなど)、業務品質、性能目標、運用、移行要件、セキュリティなど
システム要件定義	システムモデルの検討 等	システム構成、業務アプリケーション(構造、DB・ファイル構造など)、運用、移行要件、セキュリティ、機密情報保護対策など

(『経営者が参画する要求品質の確保 第2版』p.43)

事業要件定義、業務要件定義、システム要件定義とそれぞれターゲットとするスコープは異なるが、広義での要件定義とは顧客(取得者)の要求(requirement)から、仕様を確定する作業の事を言う。要件定義を正しく捉える為には、要求と仕様の関係を対比的に理解することが重要となる。

要求とは、現実世界の問題を解決するためにソフトウェアが装備しなければならない機能や性能である。それゆえ、要求に記述されるべきは、現実世界の問題やステークホルダのニーズでなければならない。すなわち、要求は問題を認識しているステークホルダの視点から定義したものである。要求が外部仕様と呼ばれる由縁である。しかし、そうした要求を、そのままの形で実際のソフトウェア開発に供することはできない。生の要求を、ソフトウェア開発者が理解でき、コンピュータによって処理可能な形に書き換える必要がある。こうして書き換えられたものを、要求仕様と呼ぶ。要求の多さに応じて要求仕様の数も増えることになる。定義された要求仕様は、1つの文書としてまとめられる。これを要求仕様書と呼ぶ。ソフトウェア開発の基盤となる要求仕様書には、要求のすべてが記述されていなければならず、その記述に曖昧性があってはならない。端的に言えば、要求仕様書とは、ソフトウェアの利用者が望んでいることを、開発者が正確に理解するための媒体と言える。

（『要求工学概論』, p. 7[27]:強調は引用者による）

要求と仕様の違いを表す具体例を以下に示す。いずれもあるオンラインショッピングシステムの開発に関わる表現であるものとする。

A) 利用者による商品購入を促進できるように、商品をひとつひとつ注文するのではなく、実際の買い物と同じように選択された複数の商品を一括して注文できるようにしたい。

B) ブラウザの画面上に、複数の商品情報を 60 件まで表示することができる。商品は、画面上でひとつずつ独立して選択/非選択の操作を行うことができ、また選択/非選択の状態を回数制限なしに変更することができる。ただし、商品の同時選択の最大数は10個までとする。10個を超えて商品を選択しようとした場合は、11 個目の商品は選択状態に変更できず、画面に「選択数の上限を超えるため選択不可能」という事態を伝える警告のメッセージを表示させる。選択された商品は、注文操作を行うことにより、合計金額の大小にかかわらず、一括して注文することができる。

ここで、A は、経営層（経営者など）や業務層（業務担当者など）の立場から、「要求」を表現したものである。経営層や業務層の担当者は、事業が成功する、もしくは、業務が改善されるという観点から、コンピュータやシステムに対するニーズや条件を「要求」として表現している。経営層や業務層の「要求」には、システムやソフトウェアの動作自体に関する条件や制約は含まれない。

一方、B は、IT システム層やソフトウェア層の立場から、「仕様」を表現したものである。IT システム層やソフトウェア層の担当者は、システムやソフトウェアが実際に動作する際の条件や制約を「仕様」として表現する。これは、「仕様」が、作成されるシステムやソフトウェアの動作自体を直接的に規定するものであることを意味する。IT システム層やソフトウェア層の担当者のアクティビティは、システムやソフトウェアが最終的に、この「仕様」を満たすことを目標に実行される。逆にいえば、IT システム層やソフトウェア層の担当者は、事業目標や業務目標の達成を直接的に意識するわけではない。

したがって、システム・ソフトウェア開発においては、顧客の要求を取り出し、それを仕様へと変換する作業が必ず必要となる[28]。この作業は、共通フレームでは、企画プロセスにおける事業要件定義、要件

27　妻木 俊彦, 白銀 純子, 『要求工学概論』, 近代科学社, 2009, p.7

28　しかし、共通フレーム 2007 では、上記「要求」と「仕様」が、「要求」もしくは「要件」というひとつの概念にまとめられているように思われる。「・・・rquirement は、事業目的や業務目的に照らして『本当に必要か』が十分検討されており、『どうなれば要求が満たされたことになるのか』を示す手段まではっきりしており、曖昧性のない形式で書かれたものでなければならない」と規定されている

定義プロセスにおける業務要件定義（機能要件・非機能要件定義を含む）から、開発プロセスのアクティビティであるシステム要件定義とソフトウェア要件定義までが該当すると考えられる。そして、実際にシステム仕様とソフトウェア仕様が決定されるのが、システム要件定義とソフトウェア要件定義のアクティビティとなる。

(『共通フレーム 2007 第 2 版』　p.34)。ここで、「手段」が「はっきりして」いるとは、「要求」が、システムの「運用/機能/設計上の特性を識別」し、「テスト可能で、測定可能で」あることを意味している（同前）。このような規定から、共通フレーム 2007 の「要求」概念は、「仕様」を含む概念であると考えられる。

5.2 要求工学と要求工学知識体系（REBOK）

要件定義は、その重要度から、ソフトウェア開発全般を学術的に扱うソフトウェア工学の一分野として確立された学術領域が存在する、それが要求工学である。要求工学とは、システム構築におけるユーザーの要求を、科学的に定義するための方法や考え方の総称で、ソフトウェア開発工程における最上流工程である要件定義のプロセスを工学的に定式化する技術である。さらに、要求工学の今日までの学問上の成果を総括し、知識体系化したものが、要求工学知識体系(REBOK:Requirements Engineering Body Of Knowledge:"アールイーボック")[29]と言う形で確立されている。本項では要求工学と、その知識体系である REBOK に関して学び、次章以降扱う「要件定義」実践の方法論の1つである USDM を学ぶ為の基礎を身につける。

システム・ソフトウェアに対する「要求」(requirement)という概念をどのように理解するべきか、どのように獲得するべきか、それはどのような成果物と帰結を生み出すべきか、この問題に関しては、様々な機関がそれぞれの定義を提唱しているが、細かな違いはあれ根本となる概念には共通項がある。さしあたり、IEEE Std 610.12-1990[Standard Glossary of Software Engineering Terminology]における「要求」概念の定義を確認してみよう。

> (1) A condition or capability needed by a user to solve a problem or achieve an objective.
> (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
> (3) A documented representation of a condition or capability as in (1) or (2).
>
> (1)問題を解決したり、目標を達成したりするために、ユーザーが必要とする条件または能力。
> (2)システムあるいはシステムのコンポーネントが、契約、標準、仕様、あるいは、その他の正式に課された文書を満たすために、合致すべき条件、もしくは、保持すべき能力。
> (3)(1)あるいは(2)に含まれるような条件、もしくは、能力の文書化された表現。
> (以上、拙訳)

上記定義から、システム・ソフトウェア開発における「要求」は、ユーザー(クライアント)、契約上の条

29 一般社団法人情報サービス産業協会 (JISA) REBOK 企画 WG,『要求工学知識体系 第１版』, 近代科学社, 2011

項、合意に基づく仕様などの強制力によって、システムやソフトウェアに対して課される条件として捉えられる事が分かる。一方、この定義では「要求」の成立過程、システムあるいはソフトウェアがそれを満たす過程について、具体的なことが分からない。このように、提唱期間によって満たす範囲が異なる要求や要件定義に関する定義を、包括的にまとめたものが要求工学(Requirements Engineering)であり、さらに、要求工学の今日までの学問上の成果を総括し、知識体系化したものが REBOK である。

REBOK の起源は、要求工学の実践を図るために、一般社団法人情報サービス産業協会(JISA)内に設置された REBOK 企画ワーキンググループの活動によって 2006 年に研究が始められ、2011 年に出版物として発表されたもので、REBOK では、

“「要求」には、スコープの違いにより、「ビジネス要求」、「プロダクト要求」、「システム要求」、「ソフトウェア要求」[30]といった区別があり、そのため、スコープに応じて、エンドユーザ、経営者、エンジニアなど、多様なステークホルダが関与するということ。”

を前提としており、この観点から、REBOK は、以下の5つの原則を基本方針として作成されている。

(1) ベンダ、ユーザが共通に利用できる知識体系である。

(2) 要求アナリストに加えて、エンドユーザ、経営者など、要求開発に関与するステークホルダが、必要に応じて習得すべき範囲と水準を整理した知識体系である。

(3) ビジネス要求/プロダクト要求、システム要求、ソフトウェア要求の 3 つのスコープに応じた知識体系である。

(4) エンタープライズシステム、組込みシステムの要求開発に共通する技術の知識体系とする。ただし、ドメイン固有の知識は個別に定義されるものとする。

30 共通フレーム 2007 との照合では、ビジネス要求/プロダクト要求は「事業要件」と「業務要件」、システム要求は「システム要件」、ソフトウェア要求は「ソフトウェア要件」にそれぞれ対応すると思われる。しかし、『要求工学知識体系 第１版』p.18 では、ビジネス要求は、なぜか、共通フレーム 2007 の「システム化構想」と「システム化計画」に対応するものとされている。共通フレーム 2007 では、これら「システム化構想」と「システム化計画」は、いずれも、企画プロセスに含まれるアクティビティの名称に含まれる概念だが、共通フレーム 2007 内では、とくに定義や説明がされておらず、その意味は明確になっていない。それにもかかわらず、『要求工学知識体系 第１版』の執筆者たちは、自分たちの概念「ビジネス要求」と、共通フレーム 2007 の「システム化構想」と「システム化計画」概念を等価であると判断している。『要求工学知識体系 第１版』と『共通フレーム 2007 第２版』をいくら照合しても、このような結論に達した理由は明確にならない。

(5)グローバルに広く利用できる形態とする。

5.3 REBOKの知識体系

REBOK の知識体系は、「REBOK 共通知識カテゴリ」(REBOK Core)と「REBOK 拡張知識カテゴリ」(REBOK Extension)の二つのカテゴリから構成される。

(1)REBOK 共通知識カテゴリ

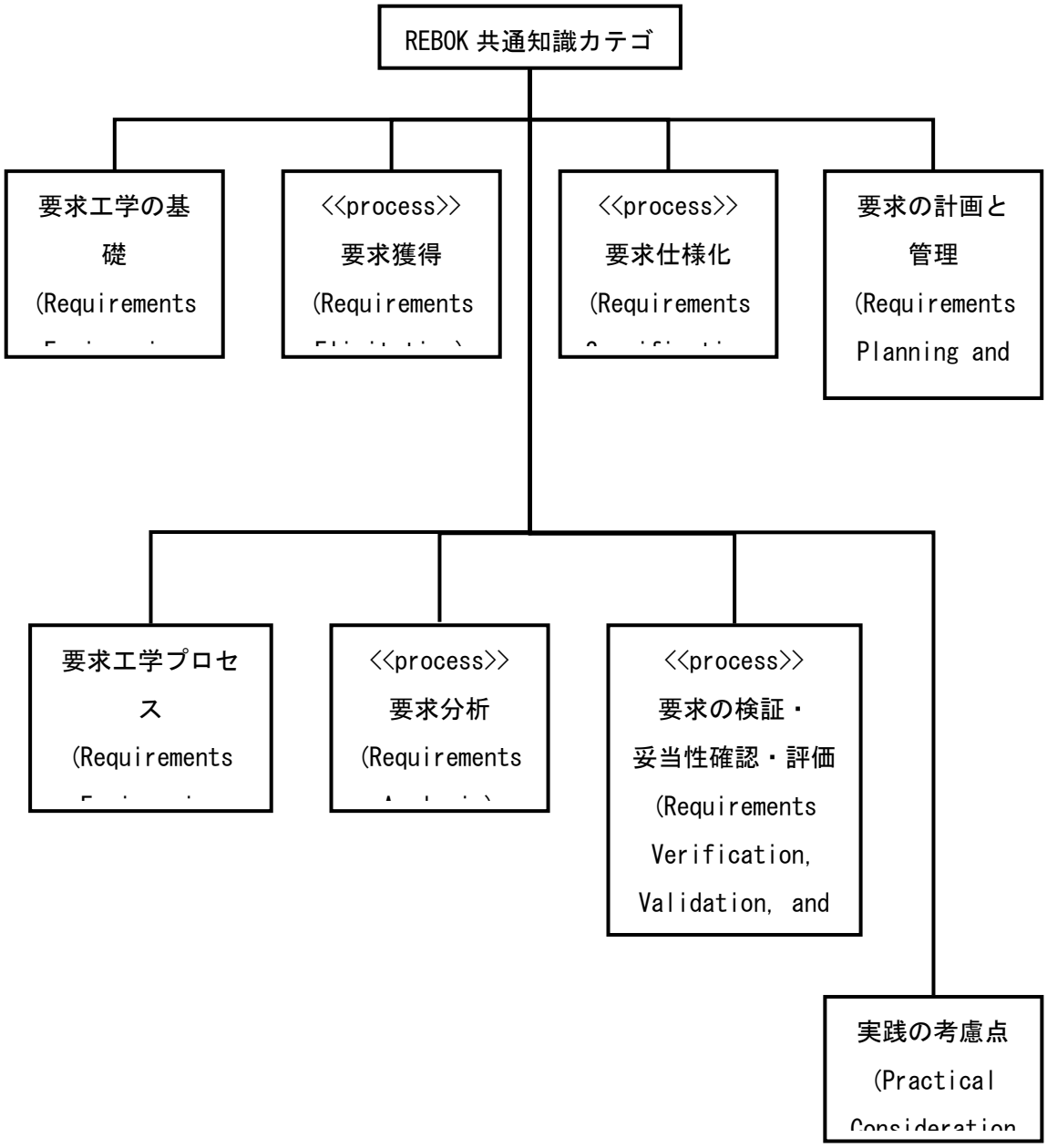
特定の「ドメイン」に依存せず、要求工学の知識として共通に利用される知識領域郡である。

※「ドメイン」とは、システム化の対象となる領域である「問題領域:problem domain」を意味し、単に「ドメイン」と呼ばれるようになったもの。

(2)REBOK 拡張知識カテゴリ

エンタープライズシステム、組み込みシステムなど、特定のドメインに対して要求工学を適用する際に必要となる知識領域群である。実際の適用にあたっては、ドメインに応じて、このカテゴリに含まれる知識を拡張する。

二つのカテゴリから、ここでは、重要度の高い REBOK 共通知識カテゴリについて確認する。REBOK 共通知識カテゴリは、8つの知識領域から構成される。



ここで、<<process>>の表記があるものは、「プロセス知識」(Process Knowledge)を意味し、何らかの活動を含む「プロセス」を規定する知識領域を意味する。他方、<<process>>の表記がないものは、「技術知識」(Technical Knowledge)を意味し、概念や技術に関する知識領域。

ただし、ここにおける「プロセス」概念は、共通フレームの「プロセス」概念よりも小さな単位(細粒度)の概念であり、内部に複数のアクティビティを含むものの、共通フレームのプロセスほどの規模にはならない。REBOK の知識体系は、共通フレームに対して、そのプロセスに含まれる一部のアクティビティを、いわば、クローズアップして記述したものであるといえる。以下、各知識領域の内容です(4つのプロセスに付された番号は、プロセスの進行順序)。

知識領域	内容
要求工学の基礎	要求とそのスコープや性質などの基礎的事項．機能/非機能要求も含む
要求工学プロセス	要求定義と管理のプロセスと主要なアクティビティなどに関する知識
要求獲得（１）	顧客を含むステークホルダーを明らかにし、会議やインタビューなどを通して要求を引出す技術に関する知識
要求分析（２）	要求項目を整理し、その間の関係づけ、優先順位づけなどを行い、実現すべき要求を明らかにして絞り込む技術に関する知識
要求仕様化（３）	分析された要求を規定の書式や表記法によって記述する技術に関する知識
要求の検証・妥当性の確認・評価（４）	要求間に矛盾がないことや、顧客にとって必要な要求項目を満たしていることの確認、あるいは、その達成の度合いを評価する技術などに関する知識
要求の計画と管理	要求管理を計画し、遂行や成果物を管理する技術に関する知識
実践の考慮点	要求工学を実践する上で知っておくべき知識やベストプラクティス

REBOK では、上記の4つのプロセス(要求を定義するためのプロセス)を、「要求工学プロセス」(Requirements Engineering Procss)と呼んでいる。

以下、それぞれの「要求工学プロセス」の定義について詳説する。

■要求獲得

REBOK では、要求の獲得(Requirements Elicitation[31])の趣旨について、次のようにまとめている。

> Requirements Elicitationという言葉には、ステークホルダも意識していない要求を発見し、構成し、定義するという意味がある。以前は、要求アナリストが、ステークホルダから要求を入手するという意味を持つ Requirements Acquisition が使われていた。しかし、要求獲得という活動には、抽出という意味を持つ Elicitation の方が実態を表しているとして、Acquisition に代わって使われるようになった。
>
> 要求とは、ステークホルダが抱えている課題を解決し、彼らが望んでいる状況を作るために実現すべき事柄を指す。したがって、要求アナリストがステークホルダから要求を獲得するためには、いくつかの準備が必要となる。第一に要求アナリストは現在の状況を理解しなければならない。第二に、現在把握されている課題を識別し、第三に、課題を解決することによって目指すべきゴールを発見しなければならない。発見されたゴールからは、ゴールを達成、あるいは維持するための手段を導出することが可能となる。その結果、将来に向けて望まれている状況を定義できるようになる。そして、要求は、現在の状況と、将来の状況との差分として定義される。
>
> (『要求工学知識体系 第1版』, p.42:網掛けは引用者による)

要求獲得プロセスには、以下の活動が含まれる(共通フレームの水準では「タスク」)。

活動	概要もしくは趣旨
1)ステークホルダの識別	獲得する要求に関係するステークホルダを漏れなく識別することが、要求の抜け漏れを防止するために必要。要求の獲得の過程で、ステークホルダは、要求アナリストがインタビューを行う対象者となる。

31 elicitation は、英語の名詞で、「(事実・情報などを) 引き出すこと」、「(事実・返事・笑い声などを) 誘い出すこと.」といった意味の単語です。また、後述する acquisition も英語の名詞で、「獲得」、「習得」という意味です。

2)現状システム[32]の理解	要求アナリストは、現状システムを観察し、ステークホルダがソフトウェア・システムによって、何を実現しようとしているのか、何を達成できれば成功となるのかについてシナリオを作成する。
3)現状システムのモデル化	上記2)によって作成したシナリオを、一定のモデル化技術によって、モデル化する。このモデルを「System-as-is モデル」という。シナリオは自然言語で作成されるため、モデル化によって共通理解が促進される。
4)課題の抽出と原因分析	「System-as-is モデル」を分析し、解決するべき課題を明らかにする。
5)課題解決に向けたゴールの抽出	上記課題を解決することによって、達成すべきゴールを明らかにする。
6)ゴールを達成する手段の抽出	現在の状況からゴールを達成した状況へ移行するために必要な手段を抽出する。ここで抽出された手段が、ステークホルダの意図と矛盾しない妥当な手段であることを、ユーザー側のシステム開発担当者が確認する。また、要求アナリストは、この手段によりゴールを達成した状況を、将来の状況を説明するシナリオとして定義する。
7)実現すべき将来システムのモデル化	将来の状況を説明するシナリオに基づき、実現される将来のシステムをモデル化する。これは、「System-to-be モデル」という。
8)要求の記述と詳細化	要求とは、ステークホルダが現在の状況からゴールを達成した状況へと移行するために、実現しなければ事柄のことである。つまり、要求は、現状システムのモデルと将来システムのモデルの差分として定義し、記述できる。

32　ここでいう「システム」概念は、共通フレーム 2007 の「システム」ではなく、「広義のシステム」であり、「コンピュータ内で稼働するシステムやコンピュータが組み込まれている製品だけでなく、それらの利用者、利用環境、ビジネス環境、市場を含めている」とされている（前掲書 p.45）。つまり、共通フレーム 2007 の事業、業務、IT システム、ソフトウェア・ハードウェアの 4 階層の全体に対応する。ただし、まだシステム（狭義のコンピュータによるシステムのこと）が導入されていない場合は、事業と業務の 2 階層だけを含むことになる。

■要求分析

要求分析とは、獲得または抽出された要求を、将来、仕様として完成させることができるような品質をもった状態となるように、洗練化して表現する作業。具体的には、要求を分類して、それぞれを概念モデルによって表現し、要求間の関係を明確にする。その上で、要求の全体がステークホルダーのゴールを達成できるように調整する。

要求分析プロセスには、以下の活動が含まれる。

活動	概要もしくは趣旨
1)要求の分類	ビジネス要求(プロダクト要求)/システム要求/ソフトウェア要求といった区分や、機能要求/非機能要求といった区分などに基づき、要求を分類する。その上で、要求間に重複、不整合、非一貫性がないかをチェックする。たとえば、ビジネス要求があるにもかかわらず、システム要求に反映されていない要求がないか、システム要求がありながらビジネス要求と対応していない要求がないことなどを確認する。
2)要求の構造化	要求を概念モデルによって表現し、自然言語によっては発見の困難な要求の漏れ、矛盾、曖昧さを検出する。
3)要求の割り当て	それぞれの要求をアーキテクチャに割り当てる。ビジネス要求はビジネスアーキテクチャに、プロダクト要求とシステム要求はシステムアーキテクチャに、ソフトウェア要求はソフトウェアアーキテクチャに割り当てる。システムアーキテクチャやソフトウェアアーキテクチャでは、その要素であるコンポーネントに割り当てる。この作業は、要求の実現可能性の見通しを確認する[33]。
4)要求の優先順位付け	ステークホルダ要求や開発コスト、要求同士の依存関係にしたがい、概念モデル化された要求、もしくは、システムアーキテクチャやソフトウェアアーキテクチャの要素であるコンポーネントに割り当てられた要求に対して、優先順位を設定する。

33 システムアーキテクチャやソフトウェアアーキテクチャに対する割り当て作業は、共通フレーム2007では、開発プロセスのシステム方式設計、ソフトウェア方式設計のアクティビティに該当する。

5)要求交渉	要求の範囲や優先順位の妥当性などについてステークホルダと合意を形成するための交渉を行う。とくに、ステークホルダごとに要求に対して異なる優先順位が提示され、場合によっては、要求が互いに対立することがあるため、適切なトレードオフを行うことにより、対立の解消を図る。この結果、ステークホルダとの間で仕様化の対象とする要求の範囲について合意を形成する。

■要求仕様化

これまでに獲得され、概念モデル化され、アーキテクチャへ割り当てられ、優先順位が設定された要求には、複数の水準がある。それは、ビジネス要求/プロダクト要求、システム要求、ソフトウェア要求の3つの水準であり、共通フレーム 2013 と比較した場合、それぞれ、事業要件と業務要件、システム要件、ソフトウェア要件に対応すると考えられる。要求仕様化の段階では、ビジネス要求とプロダクト要求を定義し、システム要求とソフトウェア要求を、それぞれ、システムの仕様、ソフトウェアの仕様へと変換し明確化する。その結果、成果物として、ビジネス要求定義書/プロダクト要求定義書)、システム要求仕様書、ソフトウェア要求仕様書を完成させる。

ビジネス要求仕様書/プロダクト要求仕様書は、すべてのステークホルダを読者とする文書である。それは、機能要求と非機能要求を含む情報システムに対する要求、および、情報システム以外に対する要求を定義したものであり、問題解決または目標達成のために、「ステークホルダが必要とする条件(Condition)、能力(Capability)、契約、企画、仕様、または規制などを含む。」(『要求工学知識体系 第1版』, p.117)

システム要求仕様書は、情報システムに関係するステークホルダを読者とする文書である。システムやシステムに含まれるコンポーネントの機能、条件、制約を明確化します。この文書により、導入する情報システムが技術面、運用面で実現可能かどうか、利用面で問題ないかどうかの確認を行う。

ソフトウェア要求仕様書は、ソフトウェアに関係するステークホルダを読者とする文書である。ソフトウェアの機能、条件、制約を明確化する。この文書により、導入するソフトウェアが技術面、運用面で実現可能かどうか、利用面で問題ないかどうかの確認を行う。

■要求仕様の検証・妥当性の確認・評価

要求仕様の検証・妥当性の確認・評価プロセスには、以下の活動が含まれる。

活動	概要もしくは趣旨
1)要求検証 (Requirements Verification)	ビジネス要求定義書、プロダクト要求定義書、システム要求仕様書、ソフトウェア要求仕様書の検証を行う(内部に矛盾を含んでいないかなどの検証)。
2)要求妥当性確認 (Requirements Validation)	要求仕様書が、ステークホルダの所期の要求を満たしているかどうか確認する。ビジネス要求定義書が顧客の要求を経営的な視点を含めて満たしていることを確認する。妥当性が確認されたビジネス要求定義書に基づき、システム要求仕様書が妥当であるか確認する。妥当性が確認されたビジネス要求定義書、システム要求仕様書に基づき、ソフトウェア要求仕様書が妥当であるか確認する。
3)要求評価 (Requirements Evaluation)	要求に対するリスクの評価を行う。
4)要求レビュー (Requirements Review)	ビジネス要求定義書、プロダクト要求定義書、システム要求仕様書、ソフトウェア要求仕様書のレビューを行う。
5)プロトタイピング (Prototyping)	ユーザーインターフェイスの使用感やシステムの複雑な挙動を確認するために、プロトタイプを作成して検証する。

6 要求仕様にかかわる諸問題

6.1 ソフトウェア開発の現場における仕様の問題とは

1. ソフトウェア開発の現場では、要求仕様書をいくら書いても仕様のモレが残ってしまって最後にバグとなって噴き出している。いったん「これでよいですか」と合意したはずなのに、実際に製品やシステムが動作するようになってから、「そこは違っている」「この仕様が考慮されていない」と言い出されて仕様の追加や変更が発生する。また、設計や実装作業のなかで設計者自身が仕様に記述モレがあることに気づき、問い合わせを発して仕様の不備を埋める作業が繰り返されたりすることが、大幅な作業の遅れの原因になっている。その結果、最近では要求仕様書を書くことに対して拒否反応すら見られるようになっている。このような状態では、要求仕様は不完全である。

2. そもそも要求仕様書に「完全」とか「不完全」という概念を持ち込むことに無理があり、このような流れのなかで、「ウォーターフォールモデル」も否定されるようになった。「ウォーターフォールモデル」が否定される最大の理由は、問題の発見が遅くなることにある。その結果、小さく刻んで問題を少しでも早く発見しようというので、「プロトタイピング」を取り入れたり、「インクリメンタルモデル」を採用したりする。この考え方そのものは間違ってはいない。しかし、適切な要求の仕様化技術を持たない状態では、これらの方法を採用したとしても、期待した効果が得られない可能性がある。

   仕様の把握が曖昧で、その記述も不適切な状態のまま、とりあえず基本的な機能を選んで動かそうとする。そして、動作させながら必要な仕様を作り込んだり、そのあとに付加的な機能を追加したりする場合、あとで明らかになった仕様が、すでに選択されたデータ構造やアーキテクチャでは対応できないことがある。そこで採用したアーキテクチャが、ある機能の仕様にマッチしないということは、まさに仕様の衝突(コンフリクト)が起きていることを意味する。つまり、この状態では「インクリメンタルモデル」を採用したとしても、あとのほうのインクリメントで仕様を作り込んでいくなかで、最初に採用したアーキテクチャが使えないことに気づく。このように、重要な問題の発見が遅れるからという理由だけで「ウォーターフォールモデル」を否定しても、問題の解決にならないことがある。この問題は、プロトタイピングでも発生する。

3. どのような要求分析モデリングを使おうと、発見された要求を適切に表現し、そこから必要な仕様が漏れないように抽出しなければならない。要求分析モデリングでせっかく「要求」をつかんでも、要求の表現が適切でなかったり、要求の範囲が見えなかったり、扱う範囲が大きいままであったりすれば仕様の抽出のモレが生ずる。必要な仕様が抽出できるかどうかは、「要求」をどのように表

現するかによって変わってくる。残念ながら、多くの人は要求を「文章で表現」することにそれほど大きな意義を見出しておらず、それどころか、要求を文章で表現することすら怠っている。

たとえ要求を表現したとしても、その表現方法によってイメージされる仕様は変わってくる。たとえば、

・計測データに異常があれば知らせる
・計測データを表示中に重要な変化があれば、画面に警告を出す

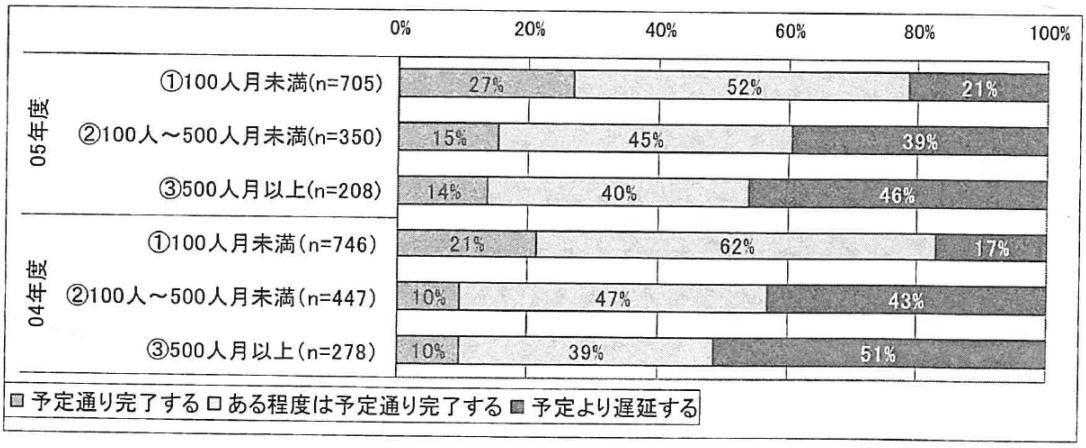
の2つの文章を比べてみると、同じことをイメージして書かれたものであっても、明らかに読み手がイメージする仕様は変わってくる。頭のなかでは同じことをイメージしていても、それをどのように文章するかで、「文章の伝える力」がまったく違ってくる。このことを理解していないと仕様のトラブルを防ぐことはできない。このように要件開発では、要求の気づき方だけでなく、要求や仕様をどのように表現するかということにも、注意を払う必要がある。

このように、要求を適切に表現することは重要で、要求が不完全な状態であると、システム開発おいてさまざまな問題が発生することになる。

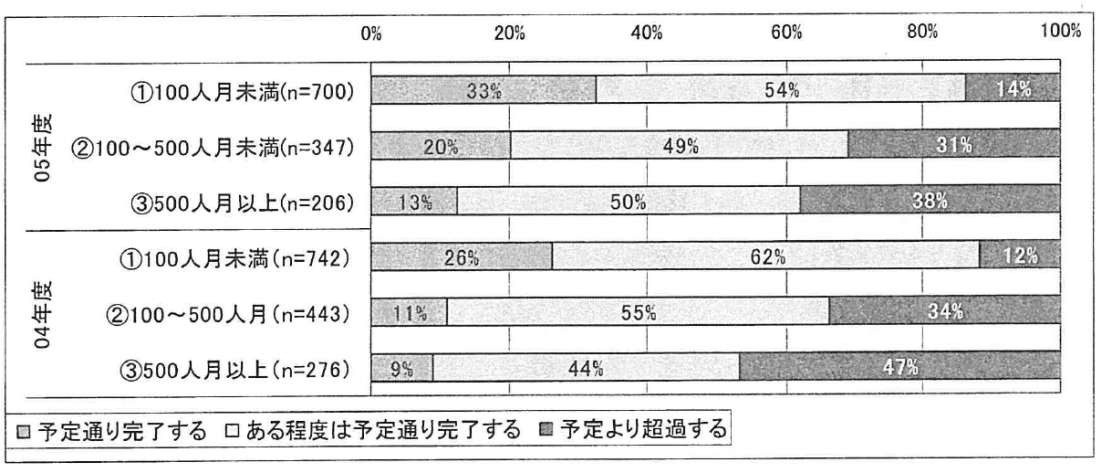
次に、要求仕様書に関わる問題について、具体的な調査結果からみていく。

6.2 要求仕様書に関わる問題と背景

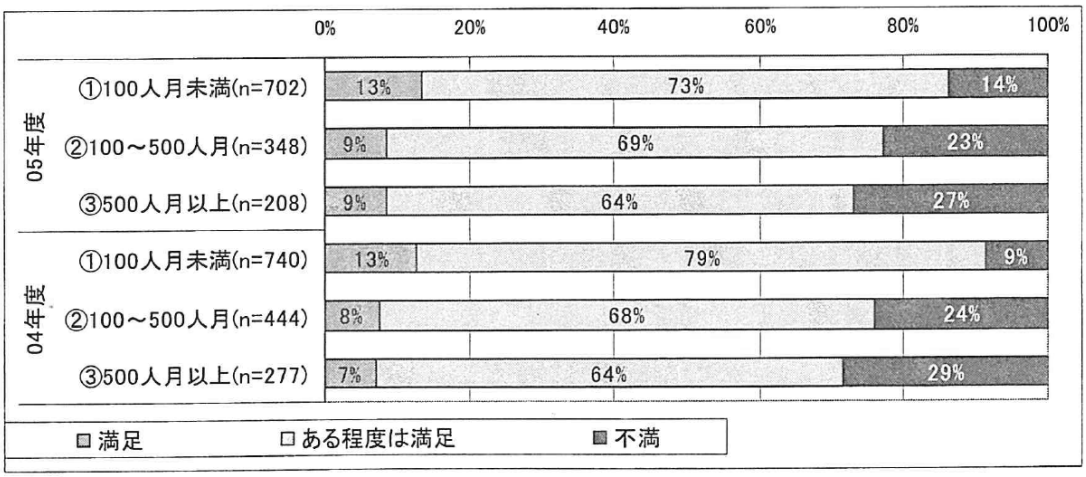
1. 日本情報システム・ユーザー協会(JUAS)によると、1994年より毎年ユーザー企業 IT 動向調査を行っており、2006年の調査報告で、システム開発に置ける工期、予算、及び品質についての満足度調査は以下のように結果を得ている。(図表1－1 システム開発における工期についての状況、図表 1-2 システム開発における予算の状況、図表 1-3 システム開発における品質の状況)



図表1－1　システム開発における工期についての状況



図表1－2　システム開発における予算の状況



図表1－3　システム開発における品質の状況

開発する情報システムの規模によって「予定通り完了する」、および「満足」の割合、あるいは逆に「予定より遅延する」、「予定より超過する」、および「不満足」の割合は異なっている。しかし、いずれの場合も、うまくいっている割合は少ない。

現状のソフトウェア危機の症状は、

・スケジュールの遅れ

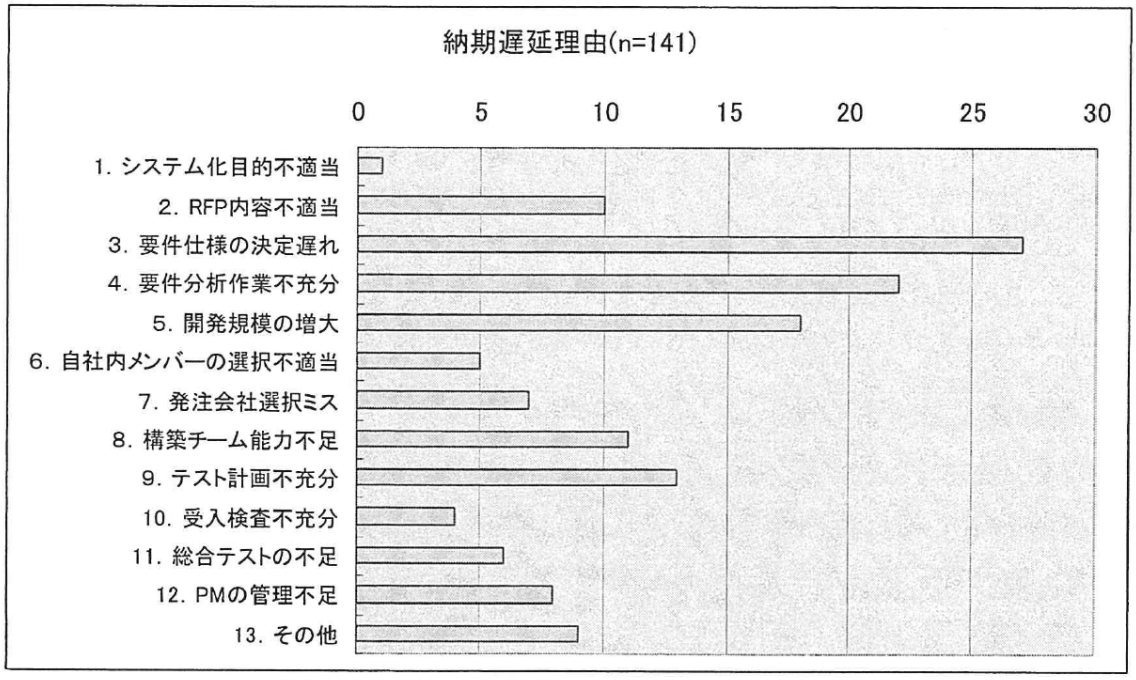
・開発予算の超過

・品質面での問題

の3つの中の1つ以上が顕在化し、場合によるとそれらが相まって、特に米国あたりではソフトウェア開発プロジェクトが当初予定していたプロダクトを完成できないまま解散させられるという形で現

れる。日本国内でもソフトウェア危機の症状の一部が顕在化していることが明確になっている。

2. これらのソフトウェア危機の現象が起きる理由は何か。JUSA が行っている「ソフトウェアメトリクス調査」は2005年から開始されたが、その2006年の調査報告に納期、つまりスケジュール遅れについてその理由を聞いた結果が掲載されている。(図表 1-4 納期遅延の理由)



図表1－4 納期遅延の理由

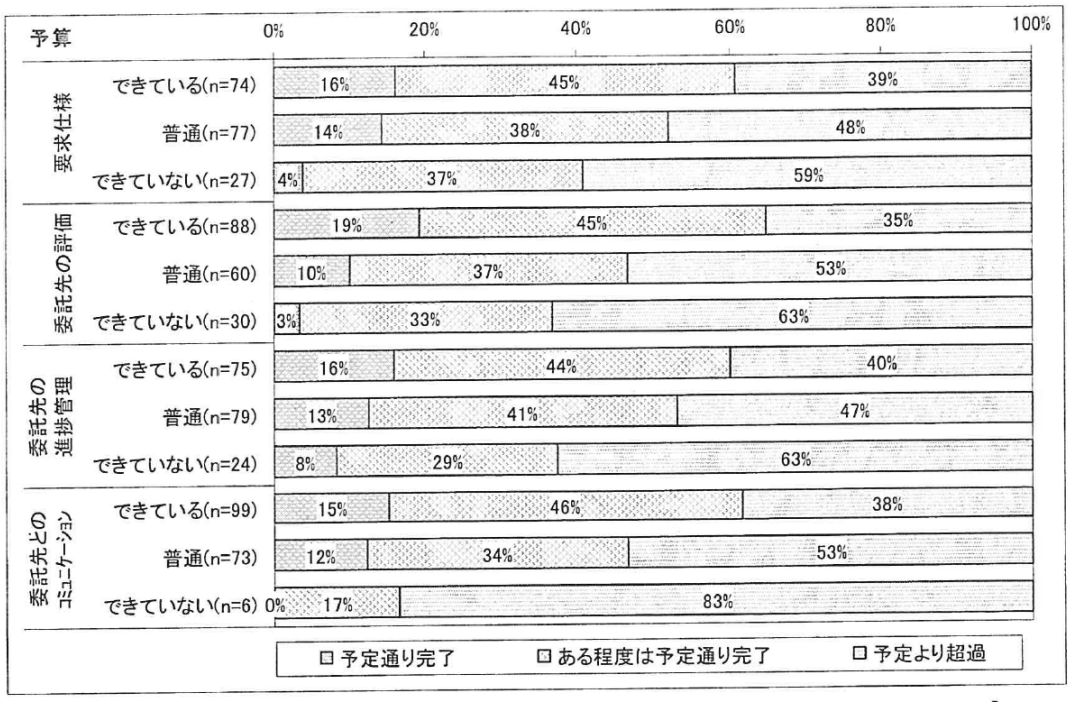
この図の中で、「2. RFP 内容不適当」と「3. 要件仕様の決定遅れ」、「4. 要件分析作業不十分」の3つが何らかの形で要求仕様書の作成に関わる項目である。グラフから明確なように、「3」と「4」は件数が多く、これに「2」を加えた3つの要件で59件、全体に占める割合では42%にもなっている。

つまり、スケジュール遅れの原因の少ない部分が要求仕様書の作成に関わりを持っているということを指摘しておきたい。言い換えると、要求仕様書を明確に作成することができると、スケジュール遅れのある部分は解消することができる。

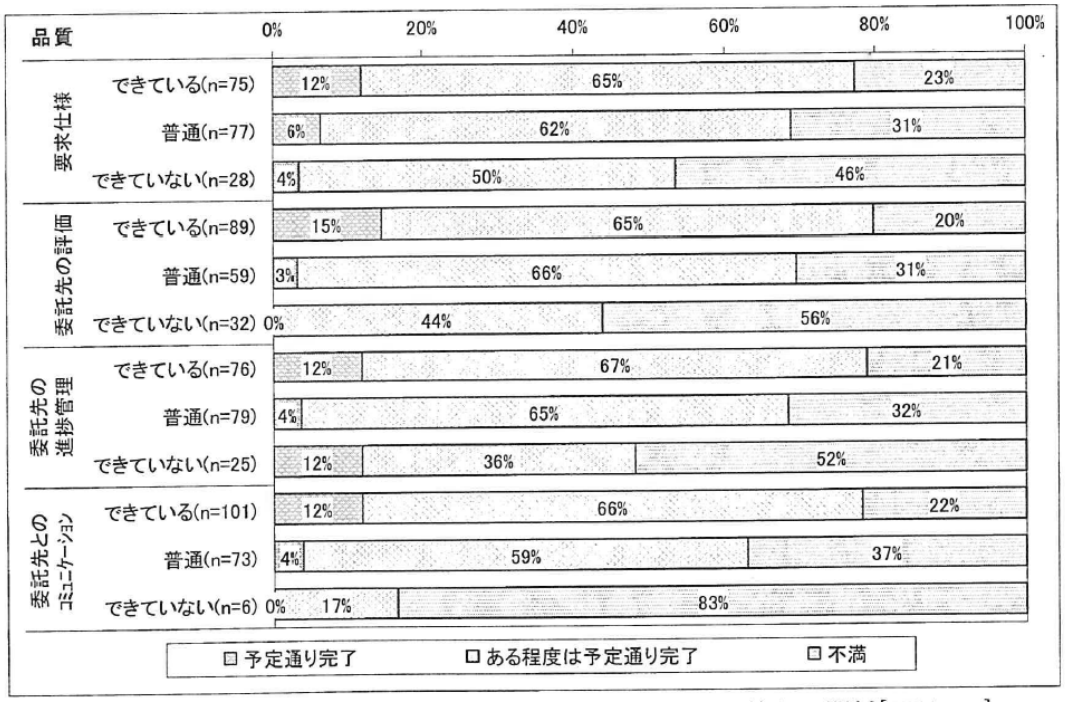
以上から、「要求仕様書をいかに書くか」ということが、いかに重要なテーマであるかということが分かる。

3. さらに、要求仕様の提示を適切に行うことで、予算を超過させないことにも品質の確保にも効果が

あるとの調査報告がある。(図表 1-5 予算と要求仕様の提示を含むプロジェクト要件との関係、1-6 品質と要求仕様の提示を含むプロジェクト要件との関係)



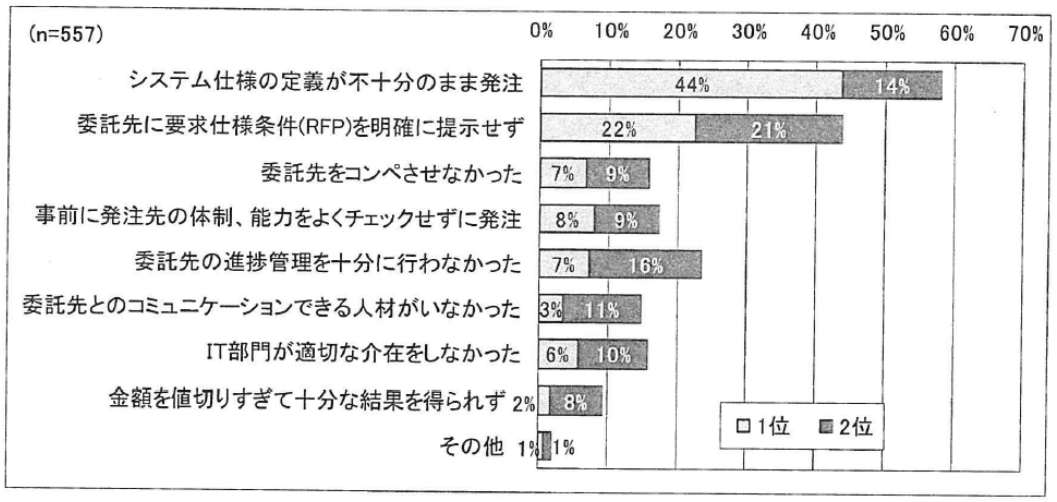
図表1－5　予算と要求仕様の提示を含むプロジェクト要件との関係



図表1－6　品質と要求仕様の提示を含むプロジェクト要件との関係

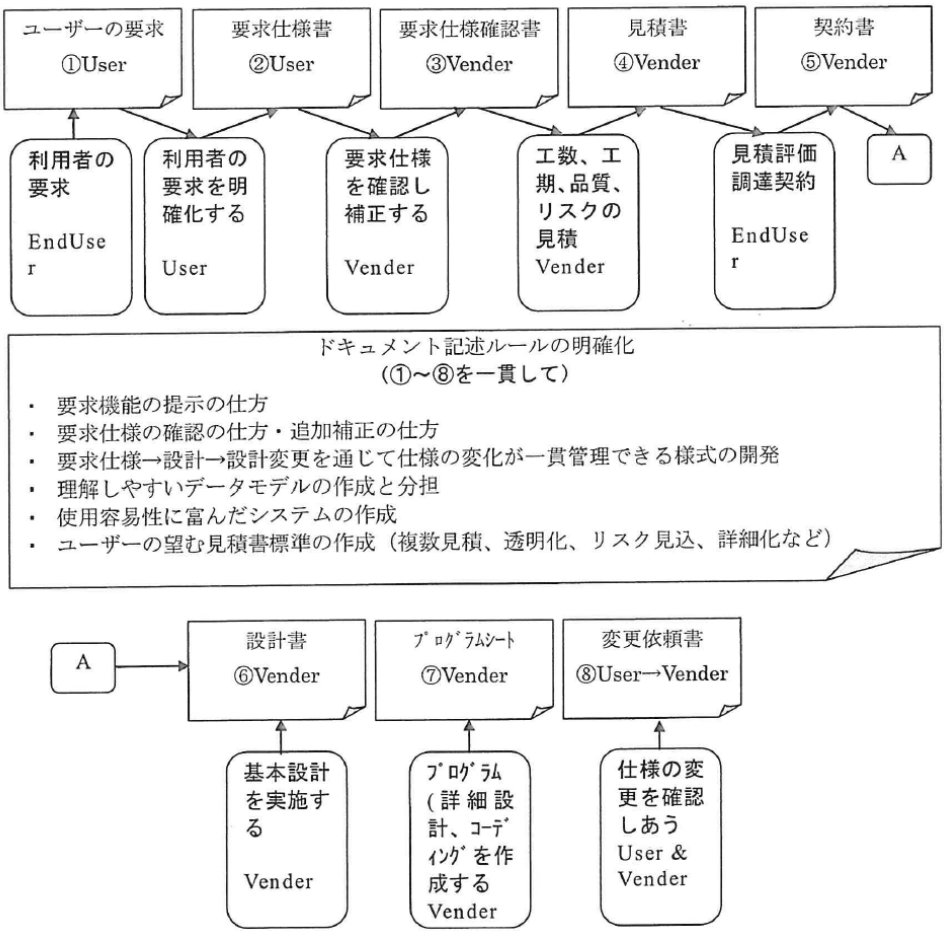
4. また、2006年度版の企業IT動向調査のアンケート調査によると、ソフトウェア開発を発注した後の「発注者としての反省点」を聞いた結果（図表 1-7　発注者としての反省点）において、要求仕様

書に関わる2つの要件(「システム仕様の定義が不十分なまま発注した」と「要件仕様条件(RFP)を明確に提示しなかった」)が、他を圧いた高い数値を獲得していることが分かる。



図表1－7　発注者としての反省点

5. 的確な要求仕様書を作成することで、ユーザーにとってどんなメリットがあるかについて考えたい。図表 1-8 は、情報システム開発の作業の流れを示したものである。この中で要求仕様書は「利用者の要求を明確にした」2 つ目の成果物として位置づけられている。さらに言うまでもないことであるがこの図から、構築される情報システムはこの要求仕様書に記述されたものを基に構築されるものであることが分かる。



図表1－8 情報システム開発作業の流れ

このことからだけでも適切な要求仕様書の重要性が分かるが、さらにユーザーが適切に要求仕様書を作成することで、次のようなメリットを受けることができる。

- 要求仕様書が適切に作成されていると、仕様の漏れが少ない。そのためベンダー側の作業である設計以降の工程で仕様確認のための余分な作業や作業の出戻りが減少し、開発の生産性向上が期待できる。
- 要求仕様書が適切に作成されていると、仕様から設計、プログラミングのトレースがより的確にできるようになる。そのため、仮に仕様変更が入っても、それに対応するための工数が少なくなる。
- 要求仕様書が適切に作成されていると、テストケースの洗い出しがより適切にできるよになり、テスト実施の効率が向上する。
- 設計作業とテストケース設計の作業を、並行して進めることができる。
- ベンダー側のリスクが減少する。それにともないベンダーの開発費用の見積がより適切なものとなる。

- 納期遅れのリスクが現象する。
- プロダクトの品質の向上が期待できる。

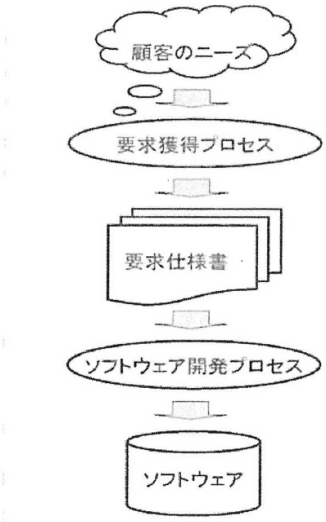
以上から、要求仕様書を的確に作成することは、非常に重要であることが分かる。

7 要求仕様書の意味

7.1 要求仕様書とは

1. 要求仕様書とは、「これから開発するソフトウェアの『作業のゴールでとしての要件』を明らかにするものであり、顧客や開発関係者間でこれから作るものについて合意するための文書である」。

ソフトウェア開発の課程では、多くの文書類が作成される。要求定義書はそのごく初期に作成されるものであり、情報システムへの要求(ニーズ)を初めて文書にしたもの、ということができる。(図表2-1 要求仕様書の位置づけ)



図表2－1 要求仕様書の位置づけ

「品質」についての国際的な公式定義は、ISO 9000 の規格(ISO 9000:2000)の中にあり、それによると品質とは、「本来備わっている特性の集まりが、要求事項を満たす程度」とある。これを「ユーザーの要求への適合度」と端的に捉えることができる。つまり、「ユーザーのニーズに合っている割合が高い製品ほど、品質が高い」ということである。

高品質なソフトウェアを作るためには、「要求獲得のプロセス」でユーザー(顧客)のニーズを的確に把握し、その結果を明確に要求仕様書に記述することが必要である。その上でその要求仕様書に記述された内容を、「ソフトウェア開発プロセス」を通してソフトウェアに仕上げていくことになる。このようにみると、高い品質のソフトウェアを作る上で、要求仕様書はたいへん重要な文書であるということになる。

なお、「ユーザーのニーズはビジネス全体から見て常に正しいのか」という、もう一段高い観点からの議論があるが、「ユーザーのニーズを我々の情報システム開発の出発点にする」という立場から、この議論については避けることとする。

7.2 「要求」とは何か

1. 要求とは「実現したいことのゴールである」。端的に「実現して欲しいこと」といっても良い。例えば、「お風呂が湧いたら、すぐにお風呂に誘導したい」は要求である。

2. 機能的要求は、「振る舞い」と表現することもできる。振る舞いとは一般的に「イベント」に始まり、「入力処理」、「変換処理」、「出力処理」などの一連の処理を終えて停止するまでの「動き」を意味する。つまり止まっている状態から何らかのきっかけを基に動き出し、ひと通りの動作を終えて再び停止するまでの範囲を持つ。これが「ゴール」、すなわち「実現してほしいこと」である。

3. この「要求」は、情報システムを開発する場合に「仕様」を導き出す、非常に重要なものである。この重要なものがヒアリングの場などで言葉として交わされることはあっても、情報システムの開発にともなって作成される文書類には、一般にどこにも表現されていない。しかし、USDM 表記法では、これを明確に記載する。

4. また要求は、「範囲」を見せるように表現する必要がある。範囲が広い場合は階層化して、範囲を狭める努力が必要である。この結果として要求は、上位要求と下位要求にわけられる。要求を分割する基準として、次の4つがある。

   - 時系列分割:時間軸に着目して、要求を分解する。
   - 構成分割:「機能」や「構成」で要求を分割する。
   - 状態分割:「状態」という概念で要求を階層化する。
   - 共通分割:共通する機能を取り出して独立させる。

   この要求の役割は、有効な仕様を引き出すことにある。各基準についての詳細は後述する。

7.3 「理由」をつける理由

1. 要求には、それが存在する理由がある。その理由を上位要求、下位要求を含むすべての要求について明示する。そうすることによって要求の背後にある理由を探ることができ、仕様を外さずに

捉えることができるようになる。お風呂の件に関わる上位要求の理由には、「燃料の消費を節約する」が該当する。

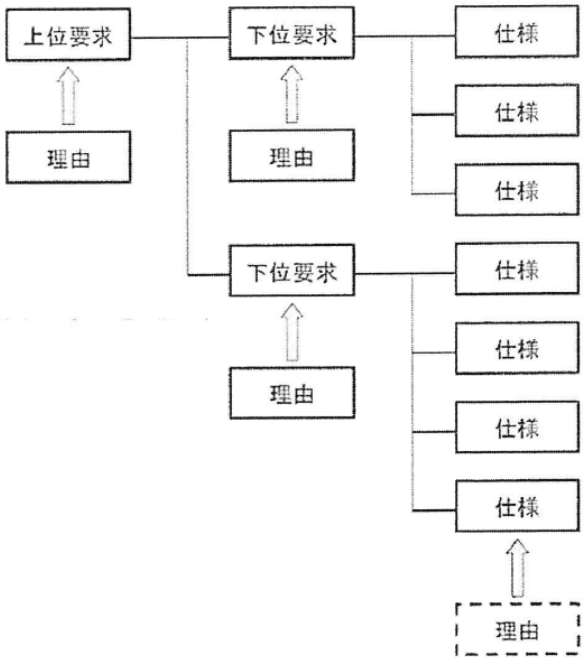
7.4 「仕様」とは何か

1. 仕様とは「要求（実現してほしいこと）を満たすための具体的な振る舞いの記述」である。したがって仕様は、必然的にいずれかの要求に属することになる。

2. さらに仕様は、最終的には何らかの形でプログラムのコードに変換されるものであり、「それが実現されていることを検証することができるもの」である。つまり仕様は、実行可能でなければならない。別の言い方をすれば、「仕様」はプログラムを読むことで把握することができる。しかし「要求」は、そのような方法で把握することができない。上記お風呂の件の要求に対する仕様の1つとして、「”沸く”状態の 1 分前に、『もうすぐお風呂がわきます』と知らせる」が該当する。

7.5 「要求」と「仕様」の関係

1. 「要求仕様書」とは、上記要求と仕様を別々に記述したものではなく、要求書と仕様書を合体したものでなければならない。つまり表現された要求の中に、仕様を収める形で記述することが必要である（図表 2-2　要求と仕様の関係）。



図表2－2 要求と仕様の関係

ここでは要求を、上位要求と下位要求にわけている。USDM 表記法では、要求は常にこのように 2 階層で表す。「ユースケース」が上位要求であれば、「シナリオ」が下位要求に対応する。「ユーザー要求」が上位要求、「機能要求」が下位要求と考えることもできる。

このような方法で仕様を明確化することで、的確に仕様を挙げ尽くすことが可能になる。

7.6 トレーサビリティへの対応

1. 仕様についてのトレーサビリティとは、どの仕様がどういう形の設計になり、さらにその設計に基いてプログラムのどの部分でその仕様が実現されているかを問題にしている。これを実現することにより、保守時の対応などで力を発揮することになる。

2. このトレーサビリティを実現するために、USDM 表記法では要求と仕様にそれぞれ「要求番号」と「仕様番号」を付けることにしている。この記番号が設計書やプログラムの中にまで持ち込まれ、それを使うことで仕様から設計、プログラムまで、一貫して追跡することが可能になる。

7.7 非機能要求の対応

1. 非機能要求とは、品質要求や保守性、移植性など、情報システムが持つべき機能以外のものへの要求である。これらの非機能要求にも、USDM 表記法では機能要求と同じ形で表現することができる。

2. つまり、この場合「要求」欄に記載される要求が「非機能要求」になり、それを実現するための「仕様」」が要求とペアになって記載されることになる。これは「機能要求」とそれを実現するための「仕様」を記載する場合と、何ら変わるところがない。この点が、USDM 表記法の1つの特徴である。

7.8 「説明」の利用法

1. 仕様の部分を適切に記述するためには、プログラムの設計についてのバックグラウンドが必要である。したがって仕様を記述している人が、それを記述している課程で、その仕様の良い実現方法に気づくこともある。

しかし、仕様を具体的にどのような形で実現するかを決める責任と権限は、後でその情報システムの設計を担当することになる人たちが持っている。仕様を決める段階でそれについて口を出すことは、越権行為となる。しかし、仕様を書いている人がその段階で気づいた実現方法は、あるいは本当に良い方法かもしれない。

2. このような場合に備えて USDM 表記法には、「説明」という項を設けて、そこには具体的な実現方法などを記載しても良いとしている。もちろんここに書くのものは、単なる付加情報でも良い。ここに書かれている情報を利用するかどうかは、後で設計を担当する人が決めればよい。しかし、これがあることで、この要求仕様書は一層の深みを増すことになる。

7.9 仕様漏れや仕様の衝突などの発見法

1. 仕様が的確に記載されていると、SE はそこから仕様漏れや仕様間の矛盾、仕様の衝突などを発見することができる。USDM 表記法を使用して要求仕様書を記述すれば、後は特別な方法を講じなくても、仕様の漏れや仕様の衝突は発見可能である。

7.10 要求仕様書を記載するためのツール

1. 要求仕様書はWordなどのワープロソフトを使って記述するのではなく、Excelを使って記述するのが良い。Excel を使うことで、要求(上位、下位に分けたもの)、理由、仕様、説明、要求と仕様に付けた記番号、それぞれについての内容などを的確に記述することができる。

2. さらにExcelには、ある場合に特定の業を折りたたんで表示しないようにする機能が当初から用意されており、その機能を有効に使うことで要求仕様書を要求書として見たり、仕様のグループ化を検討したりすることが可能になる。(図表 2-3 要求と仕様記述部分のテンプレート)

					備考欄
カテゴリ名（記号）	要求	（要求番号）	ここに要求を記述する。		
		理由	要求の背景や理由について記述する		
		説明	要求について必要に応じて説明する。仕様とは見なされない。セルを広げて図を貼り付けることも可能。		
		要求	（要求番号）	ここに階層化された要求を記述する。要求番号は一段下げられる。	
			理由	範囲を狭めた要求について、背景や理由を記述する。	
			説明	要求について必要に応じて説明する。仕様とは見なされない。	
		≪仕様分類名≫		主分割記号・・・全体を通して共通の分割基準として決める。	
			≪仕様分類名≫	補助分割名・・・主分割の中に異なるテーマの仕様が混在する場合、補助分割記号を使って純度の高い集合を作る。	
		□	（仕様番号）		
		□	（仕様番号）		
			≪仕様分類名≫		
		□	（仕様番号）		
		□	（仕様番号）		
		□	（仕様番号）		
		□	（仕様番号）		

図表2－3　要求と仕様記述部分のテンプレート

7.11 要求仕様書は誰が書くか

1. 要求部分は、ユーザーが書くのが妥当である。しかし、ユーザーの立場で仕様まで的確に書くことは難しい。では、誰が書くかということになるが、まず「SE が書くべき」とするもの、2 つめは「ユーザーと SE が共同で書くべき」とするもの、最後に「『要求エンジニア』と呼ばれる特別な訓練を受けた人が書くべき」とするものである。

2. SE だけが書くとする場合その SE は、ユーザーから作業中に十分な情報の提供を受けるか、事前にその領域について勉強しておくことが必要である。したがって実際に要求仕様書を記述するのは SE だけかもしれないが、実質は 2 つめの「ユーザーと SE の共同作業」の形をとっていることになる。

3. 要求エンジニアとは、ユーザー出身でも、SE 出身でも構わない。SE 出身の場合は適用業務の領域に深く、ユーザーに代わって要求を書けるレベルになることが必要である。逆にユーザー部門出身の場合は、仕様のレベルを的確に書くための特別な訓練を受けることが必要である。いずれにしろ「要求エンジニア」という言葉はまだ一般的ではなく、例えば経済産業省が定めて公表した「IT スキル標準」にも、この技術者は定義されていない。

4. もっとも推奨するのは、「ユーザーと SE が共同で書くべき」である。ユーザーが要求部分を書き、SE が仕様部分を書くというふうに、作業を分担させる方式である。この場合まずユーザーが要求部分

を完成させて、それを基にSEが仕様部分を追加していくという方法が可能である。複数のSEが分担して仕様部分を書いていくことも可能である。

5. ただし、仕様を書くためにはプログラムの設計などについてのバックグラウンドが必要であることと、「SE」の領域を考えると、一般に、情報システムの企画や分析といった上流からテストの実施／統括といった下流まで、あるいはソフトウェア・プロセスの改善や標準化の推進、要員教育など、たいへん幅広活躍することができる人材を意味する。ここで述べている「SE」とは、最低でもプログラム設計ができるスキルを持ち、ユーザー企業の情報システム部門所属の SE をイメージしている。ただし、ユーザー企業の情報システムの子会社所属の SE であってもよく、場合によれば、ソフトウェア会社所属の SE であっても構わない。つまり「要求を仕様に落とすことができる」人であれば、その所属する企業は関係ない。

8 知識共有の方法

8.1 知識共有はなぜ必要なのか

1. 要求仕様書に仕様もその背景にある要求も、さらに要求が出てきた理由も、すべて明確に書かれたとして、それで必要とする情報システムを、ユーザーが必要とする機能、性能、品質などを網羅して作ることはできない。

   ユーザーが必要とする機能、性能、品質などを情報システムの上に実現するためには、要求仕様書を作成した側、つまりユーザーと、情報システムを構築する側、つまり SE との間に、要求仕様書上に書かれた「言葉」と「暗黙のルール」について、共通の理解と認識がなければならない。

2. ところが不幸なことに、それぞれの業界には業界特有の言葉があり、その業界にいる人には大変便利な、使い勝手の良いものであるけれど、その業界の外側の人たちにはまったく馴染みがないという場合が多い。その業界の外側の人たちの中に情報システムを構築する立場の SE が含まれる場合、ここから難しい問題が発生することになる。

3. その言葉を SE が全く知らなくて、素直に質問してくれる場合はまだ良いが、SE が本当に理解してくれたということが分かるまで、ユーザーは SE にその言葉について説明する機会をもつことができる。しかし仮に、SE 自身がその言葉を理解していると考えていると、質問をしない。しかしその理解がユーザーが期待するものとは全く異なっていることがあると、事態は深刻である。

8.2 知識共有のレベル

1. 要求仕様書に書かれた内容の理解に食い違いを起こさないようにするために、ユーザーと SE との間で知識を共有していることが必要である。しかし単に、この両者が同じ知識を共有しているだけでは、まだ十分ではない。また、「両者がお互いに必要な知識を共有している」という事実を、さらに両者が共有していることも必要である。

2. 「『両者が必要な知識を共有している」という事実を、さらに共有している」場合、双方はこの問題について何ら気にする必要はなくなる。両者は、お互いが共に正しい認識と理解を持っているということを知っているわけであるから、非常に好ましい状態が存在していることになる。最終的に、すべての知識についてこの状態を目指したい。

3. 知識は共有されているが、その事実が共有されていない場合、両者に不安や疑心暗鬼が生じる可能性もある。結果的にはうまくいく可能性が高いが、無駄や摩擦が発生するかもしれない。あるいは、余計な神経を使わなければならないことがあるかもしれない。

4. 知識が共通されていない場合、そのこと自体がまず問題である。しかし知識が共有されていないということを少なくとも片方が知っている場合は、対応方法がある。その問題を解決しようとする力が働くきっかけを作り出せるからである。共有されていないことを両方とも気づいていない場合が、一番大きな問題を引き起こしてしまう。

8.3 知識共有のために必要なこと

1. ユーザーとSEが同じ知識を共有するために、IEEEの要求仕様書についての規格には、「1.3 専門用語、頭文字語、略語の定義」という部分があり、そこには要求仕様書の書き手、つまりユーザーの立場からみて、「読み手、つまり SE が知るべきと考える言葉」について記述することになっている。(図表 3-1 IEEEの規格のプロトタイプ)

標 題	内 容
1. はじめに	
1.1. 目的	この要求仕様書の目的、想定される読者
1.2. 範囲	開発するソフトウェアの種類。説明。このソフトウェア開発で達成しようとする目的、ゴール、得られる利益。「業務要求」もここに記載する。
1.3. 専門用語、頭文字語、略語の定義	
1.4. 参照	参照する資料・文献
1.5. 概要	この要求仕様書の構成
2. 全体の記述	
2.1. 製品についての考え方	システムを構成する他の部分との関係、インタフェース
2.2. 製品の機能	3で述べる機能の概要
2.3. ユーザの特性	想定されるユーザの経験や技術レベルなどで特記するべき事項
2.4. 制約	このソフトウェアの開発に伴う制約事項
2.5. 前提	このソフトウェア開発の前提条件
2.6. 先送りされる要求事項	
3. 要求と仕様	要求仕様書の本体部分。「USDM 表記法」で記述する。
付録	
索引	

図表3－1 要求仕様書のプロトタイプ

しかし、ここには何を書くべきなのかという問題が残っている。頭文字や略語ならまだ分かるが、「専門用語」が明確ではない。

2. 一般に我々は、「自分が何を知っているのか」、あるいは、「何を知らないのか」を知らない。まして自分ではない他の人が「何を知っているのか」、あるいは「何を知らないのか」は全く分からない。このため、ここに何を書けばよいのかは、一般に明らかではない。つまりここで専門用語として扱わなければならないの言葉には、明確な基準が必要となる。

3. 仮にユーザーが、「専門用語や頭文字語、略語の定義」を丁寧に書いたとして、それでも十分だろうか。要求仕様書を作成したユーザー側は、これで SE の知識は十分になると期待するかもしれない。しかし SE の知識がまだ十分でなく、しかも双方にその認識がなければ、問題を引き起こす可能性が高い。

4. ここで、あるテーマについて双方が知っている／知っていないという軸と、そのことを双方が認識している／認識していないという軸の二軸で整理すると、図表3－2のようになる。

	相手の状態を正しく認識している。	相手の状態を間違って認識している、または相手の状態を知らない。
両方が同じ内容を正しく理解している。	◎	b
両方とも理解していると思っているが、内容は一致していない。	a	c
片方は正しく理解しているが、片方は知らない。	a	d
両方とも知らない。	----	----

図表3－2　知識共有の状態

◎を付けたケース、つまり「両方が同じ内容を正しく理解している」場合で、さらに「相手の状態を正しく認識している」の場合は、もはや何もする必要がない。これが望ましい状態であり、最終的にすべてのケースでこの状態を目指すべきである。

また「両方とも知らない」というケースでは、議論する必要がない。この場合そのことが要求仕様書に書かれることがないからである。

したがってこれ以外の状態のものを、どうすれば◎を付けたケースにもっていくことができるかを考える必要がある。

「相手の状態を正しく認識している」、つまり a のケースでは、対応は容易である。お互いにじっくりと話をすることで、解決することが期待できる。

しかし相手の状態が不明の場合、つまり b、c、d の場合は、それほど簡単ではない。b の場合は、結果的に問題は起きないかもしれない。しかし、c、d の場合は簡単ではない。

ここで、a から d までの全てのケースについて何らかの方法で双方の理解／認識を確認する方法として、以下の 8 種類がある。

① 相手の言動を観察することで、相手の理解／認識を推測する（観察）
② 聞き取り調査をして、相手が理解／認識している内容を得る（聞き取り）
③ 理解／認識内容を網羅的に記述する用紙を用意し、それに記述してもらうことで相手が理解／認識している内容を得る（網羅）
④ 相手の言動から、その根底にある理解／認識を類推する（類推）
⑤ ブレイン・ストーミングなどの機会を用意して、相手の理解／認識を確認する（発想法）
⑥ 理解／内容などを表したモデルを作成し、それを通して理解／認識している内容を得る（表出による確認）
⑦ 相手の言動に表れる矛盾を通して、その根底にある理解／認識の不一致を発見する（矛盾の発見）
⑧ 実際に何かを作り、それを確認することで、相手が理解／認識している内容を確認する（作ってみる／試してみる）

これらの中のあるものはお互いに積極的であり、あるものは消極的である。あるものは相手の出方を単に待っているようなものもある。これらの中で、時間や費用、手間を考えなければ最も望ましい方法は、「作ってみる／試してみる」であろう。しかし、実際問題としてこの実現は困難である。

それに次ぐ望ましい方法は、「表出による確認」である。これはあるモデルを作って、それを通してその理解／認識している内容を表現してもらい、不一致を発見する方法である。ここで不一致を起こしていると思われるものには、次の二種類のものがあり得る。

・データ、及びそれを構成している項目
・データ（またはその項目）に対して行うべき処理、及びそれに関わる内容

データを明確にするモデルとして、概念データモデルがある。概念データモデルは実態関係図（E-R 図）、またはクラス図で記述されることが多い。またそれらを記述することは、一般にさほど難しい作業を要しない。

また、概念データモデルから新たに抽出された言葉を要求仕様書の「専門用語、頭文字語、略語の定義」に追記することによって、言葉のレベルでの理解／認識の一致が一層進むものと期待できる。

9 要求仕様の定義方法（USDM 表記法）

9.1 さまざまな要件定義技法について

1. 従来、ソフトウェアの品質などの問題は設計方法で解決しようとしてきたが、1990年代の初め頃に、要求仕様の問題をクリアしないと設計では対応しきれないということで認識が共有された。そして、世界中で要求仕様の在り方について研究が進められるようになり、「要件開発」の重要性について認識が広まった。近年では、「要件開発」の方法として、「I*」「CAOS」「E3VALLUE」「AFORA」などの要件開発技法があるが、これらは要求の抽出あるいは発見に力が置かれており、発見した要求をどのように表現するかということについてほとんど触れられていない。

2. 本書では、要求仕様書の書き方として、株式会社システムクリエイツの清水吉男が提唱する「USDM（Universal Specification Describing Manner）表記法」を採用する。前述のとおり、要求は複雑でかつ広範に渡るものであり、適切に要求を抽出できなければ、システム開発の下流まで引きずることになる。USDM を使用することで、要求の本質を明示し、テストや実装中に発見される仕様モレや仕様の衝突、仕様の変更を大幅に減少させることができる。要求の本質を発見するために適した表記法であり、その他の要件開発技法にはない特色を備えている。特に、仕様モレよりも、要求モレを防ぐことに重点を置いている点が、要求仕様の高品質化を実現している。

9.2 USDM とは

1. USDM は、要求仕様書に「要求」を記述して表現する。（要求を要求仕様書にまとめて、要件（仕様）を要件定義書に別々にまとめるドキュメント体系を採用している組織の方は、USDM の要求仕様書は要件定義書にあたる。あるいは、要求仕様書に要件（仕様）を書き加えることで、要件定義書の役割も持たせるというイメージの方がつかみやすいかもしれない）。USDM を使うと、「要求」と「仕様」を「要求仕様書」として文書化し、それを担当者間の共有認識として開発をすすめることができる。上流だけでなく、下流においても、余計な問題や不具合の発生を抑えることができる。

2. 具体的に、USDM を採用する理由として、以下のことが実現可能である。
   (1) USDM 表記法では、まずシステムへの要求を明確にする。要求を 2 段階で記述することにし、その下位の要求を引き出すための基準がある。これによって、要求を明確にすることがより容易になる。
   (2) 次にシステムの仕様を、該当するシステムの要求の下に記述していく。こうすることで、システムの有効な仕様を引き出すことができ、仕様を漏れなく、整合性が取れた形で抽出することができるようになる。またこれによって、仕様のレビューもより容易になる。

(3) システムへの要求に、なぜその要求をするのかについての「理由」を明記する。この「理由」を書くことによって、その要求を必要とする背景が明らかになり、仕様のレベルで別のより良い解を得ることも可能になる。

5. 仕様を明確にしている段階でもう一歩進んでそれを実現する具体的な方式まで頭に浮かんだら、それを「説明」として追記することで、要求仕様書に一層の深みを持たせることができる。

6. 次に、要求仕様書が果たすべき役割を考え、それを USDM 表記法に当てはめる方法を説明する。

9.3 要求仕様書の使いかた

1. 要求仕様書はその情報システムについて最初に作成される成果物であり、その情報システム構築に必要な全ての作業がこの成果物を出発点にしている。具体的には、要求仕様書を以下のように使うことになる。USDM 表記法の多様性がうかがわれる。

- 議論のたたき台として
- コミュニケーションの手段として
- 作業のための文書として
- アーキテクチャ設計の出発点として
- プロトタイプ作成の出発点として
- サイズ見積もりの入力として
- コスト見積もりの入力として
- 設計の入力として
- プロジェクト計画立案の入力として
- テストの入力として
- ユーザーマニュアルの作成の入力として
- 解決探しのヒントとして
- ユーザーが何を手に入れるかを示すため
- て慰安を評価するため
- 合意を得るため
- 知識を整理するため
- 理解を表現するため

2. 作成時に要求仕様書に求められる品質

このように多彩な使い方がされる要求仕様書であるから、それに求められる品質は必然的にシビアなものとなる。これを、作成時、検証時、更新時、及び参照時に分けて、それぞれ考えてみると、まず作成時においては、次のような要求を挙げることができる。

- 粒度の統一を図ることができること
- 客観的な記述方法を実現できること
- 先行文書との関係を表示できること
- 漏れ／抜け、矛盾、ダブリを発見しやすいこと。

粒度の統一を実現する原則として、「1つの文書には動詞を1つ」というものがある。この粒度の統一と客観的な記述は、簡単とはいわないにしても、現在の USDM 表記法で対応可能である。つまりこれらの要求へは、作成者が作成時にこれらの点に留意して記述する必要がある。

3. 検証時に要求仕様書に求められる品質

ここでの検証とは、レビューを意味している。レビュー時に求められる品質として、次のような要求を挙げることができる。

- 任意の事柄に関係する要求仕様を一覧にできること
- 要求の裏側にある「理由」を記述できること
- 検証済みかどうかの状態の管理が可能であること

USDM表記法では、任意の事柄に関係する要求仕様を一覧にでき、「理由」についても対処可能である。また、USDM表記法はExcelを使用して記述することが推奨されており、「状態の管理」についても必要に応じ列を追加し、その状態管理のために使うことができる。したがって、これらの品質は解決される。

4. 更新時に要求仕様書に求められる品質

更新とは要求仕様書に、変更などのために手を入れることを指す。要求仕様書は情報システムが開発されている間だけ存在する文書であって、保守などのためにこの文書がそのまま使われることはない。しかし、それでも要求仕様書は、変更されなければならない。

このような文書の変更に対応するためには、ソフトウェアの構成管理の仕組みを導入することが望ましいが、更新に備えて要求仕様書に求められる品質には次のようなものがある。

- 変更範囲を特定しやすいこと
- 文書を修正しやすいこと
- 版管理が可能であること

USDM 表記法は Excel を使って作成するため、感覚的に「箇条書き」に近い形で文書を作成することができる。このため、はじめの2つの要求には、比較的対応可能である。

版管理についても、USDM 表記法では、各要求に「要求番号」と呼ぶ固有の ID を付けることになっている。したがって、単純に考えれば、要求仕様書に記述された要求番号を該当するプログラム設計書にも、その機能を実現しているソースプログラムにも付けていくことで、要求の双方向へのトレースが簡単に実現できる。

図表●-●で、複数列ある「要求仕様」の列に、USDM 形式で要求や要求仕様が書かれる。その要求仕様の列のさらに右側に複数の列を用意し、それらを文書の列、プログラムの列とする。文書の列には、その仕様を扱った設計書などの「章、節」の番号を、プログラムの列には、クラス名や関数名を書き込む。この表を完成させることで、トレーサビリティを実現することが可能となる。このように、USDM 形式の表を拡張する方法がある。

5. 参照時に要求仕様書に求められる品質

要求仕様書は参照されることが多い。このため、要求仕様書には参照時に求められる品質が非常に重要である。参照時のための品質として、次のようなものを挙げることができる。

- 合意を明確にするための文書として、記述が曖昧でないこと
- 見積もりや計画への入力として、機能に優先順位が付けられていること
- アーキテクチャを含む設計への入力のために
  - スコープの記述ができること
  - 品質目標が明示できること
  - 設計書やソースプログラムなど他の文書との関係を示すことができること
  - 任意の事柄に関係する要求仕様を一覧にできること
- 要求仕様書の記述が詳細になりすぎないこと

10非機能要求の定義方法

非機能要求にも USDM を適用する。一般に非機能要求は明文化されていないことが多い。機能要求の実現ばかりに気をとらわれていると、非機能要求を実現するスキルを身につけることができない。そもそも非機能要求とは、どのようなものがあるかが経験によって培われることが多いのも課題である。ここでは、まず非機能要求とは何か、その種類について説明する。

- 情報処理推進機構(IPA)が公開している非機能要求グレードもあわせて参考にするとよい。
  (http://www.ipa.go.jp/sec/softwareengineering/reports/20100416.html)

- また、セキュリティについて、IPA が公開しているチェックリストを参考にするとよい。
  (https://www.ipa.go.jp/security/vuln/websecurity.html)

10.1 非機能要求とは何か

1. 要求仕様書に書かれるべき内容としては、次のものがある。
   (1) 機能
   (2) 外部のインタフェース(人間、ハードウェア、他のソフトウェアとの)
   (3) パフォーマンス(スピード、アベイラビリティ、レスポンス・タイム、復旧時間、など)
   (4) アトリビュート(移植性、正確さ、保守の容易性、安全性、など)
   (5) 設計上の制約

   「(1)機能」に関する要求を「機能要求」、それ以外の4つを「非機能要求」と呼ぶ。さらに「機能とは、入力を出力に変換すること」と単純に定義し、この定義に基づいて機能要求と非機能要求に区分する。非機能要求は範囲が広く、領域や意味が曖昧なことが多い。

   さらに、(1)～(5)の内容を詳細化、具体化し、以下の定義方法について考えたい。

   - 品質
   - サービス・レベルに関する要件
   - キャパシティに関する要件
   - 安全性に関する要件
   - コストに関する要件

- 保守に関する要件

10.2 非機能要求の重要性

1. 非機能要求を定義する目的とは、「システム強度」を確保するためである。「システム強度」とは、「変化／リスクに対する IT システムの強さ」を意味し、非機能要求を明確に定義することによって、この変化／リスクに対応できる情報システムを構築できるようになる。

2. 1990 年代に起きた IT 技術のオープン化によって、メインフレームの使用を前提とした「閉じられた」アーキテクチャから、インターネットを含む「開かれた」アーキテクチャに、情報システムの根底の部分が大きく変化した。このため新しいビジネスシステム毎にインフラが用意されるという形でインフラが乱立し、BtoB、BtoC などへの対応と共にオンライン処理の対象となるトランザクション数も、そのトランザクションを入力する端末（クライアント機）の数も、著しく増加するという変化が生じた。バッチ処理ですら一社の中だけではもはや完結せず、BtoB 接続やアウトソース活用などのためにオンラインと同様に分散化が進んでいる。

3. さらに日本経済のオープン化の結果、グローバル化、規制緩和が進み、バブル経済崩壊後の低成長の中での厳しいコスト面での制約の下、各企業は生き残りをかけて一刻を争う商品開発／サービス開発に取り組んできた。また、個人情報保護への対応、クラウドサービスやモバイルクライアントの台頭などが、企業の情報システムに求められるという状況に至っている。これらが「システム強度」の強化のための、情報システム化の根底にある社会と技術の両面での現時点の状況である。

10.3 非機能要求の定義の仕方

1. 目標の定義の仕方には、「ハード・ゴール」と「ソフト・ゴール」の2種類がある。「ハード・ゴール」とは定量的なもの、「ソフト・ゴール」は定量的ではないものを意味する。目標として、定量的なものが望ましいということはいうまでもない。目標が具体的な数値で示されることで、それを実現するための方法をより的確に考え、定めることができる。同時に、その目標が実現できたかどうかの判断もより容易になる。

2. しかし残念ながら、全ての領域の全ての項目で、定量的に目標を定めることができるわけではない。この場合には、「ソフト・ゴール」で対応せざるを得ないことになる。しかしハード・ゴールで対応できる項目には、極力数値の形で目標を明確にすることが望まれる。そのため、1つの領域の中で、項目によって「ハード・ゴール」と「ソフト・ゴール」の両方が混在することがある。

3. 機能要求は、ビジネス上の必要性から明確にされる。非機能要求も機能要求と同様、ビジネスや IT サービスの視点から明確にされることが必要である。したがってこの要求も、ユーザーやシステム・オーナと、システム開発者の間で合意する必要がある。しかし単にそれだけでなく、システム稼働開始後の安定稼働に責任を持つシステム運用部門とも合意しておくことが重要である。

4. 機能要求の性格から、それぞれのシステムでの共通点はさほど多くないかもしれない。しかし非機能要求では、各システムに共通する要素が多い。したがって各項目について基本的な要求を前もって定義しておき、それと異なる要求、あるいはその基本を上回る要求をする場合だけ明確に要求を記載するという方法が、この定義に当たっての現実的な対応と思われる。

5. 以上のことからこの非機能要求については、ユーザー部門やシステム・オーナの合意を得た上で IT 部門が記述することが妥当といえる。

10.4 品質

1. 品質要求はカットオーバ時に、1単位のソフトウェアの大きさ、つまり KLOC(Kilo Lines of Code: プログラム 1,000 ステップ)、または FP(Function Point)あたり、いくつかの欠陥が残っていても構わないとするかで表される。品質目標を持たないソフトウェア開発プロジェクトでは、持っているプロジェクトと比較すると欠陥率が2倍になったという調査結果もある。より品質の高いソフトウェアを望むなら、具体的にこの数値を決め、それをクリアすることをプロジェクトの目標とすることが望ましい。

10.5 サービス・レベルに関する要件

1. サービス・レベルに関する要求とは、コンピュータに対するユーザーの期待を表すものである。ユーザーは一般に、大量のデータの処理や複雑な処理を、時間をかけずに瞬時にこなすことを期待している。

2. しかしそのようなユーザーの期待に応えることは、単純にコスト面から考えても容易なことではなく、場合によれば不可能なことすらあり得る。この観点から事前にユーザーと十分に話し合いを行って、合意を取付けておくことが必要である。

3. さらにこのサービス・レベルは、システムの構成やアプリケーションの設計の仕方で大きく異なることがある。事前に合理的な要求を明確にしておき、カットオーバ後に「こんなはずではなかった」と

いうようなことがないようにしなければならない。

4. サービス・レベルに関する要求には、次のような項目が含まれる。
   - サービス提供時間
   - 性能
     - レスポンス・タイム
     - バッチ処理の実行時間
     - デリバリータイム

10.6 サービス提供時間

1. サービス提供時間とは、オンライン系サービスが稼働している時間帯を意味する。ユーザーの立場からすると一般に、サービス提供時間は長ければ長いほどよいことになる。しかしこの時間が長いことは運用コストの上昇を招き、単にそれだけでなく開発のワークロードやその結果として開発のコストにも影響する場合がある。それを考慮してユーザーやシステム・オーナは適正な時間を設定しなければならず、IT 部門もそれを事前によく説明しておかねばならない。

2. 場合によれば、ここで、「ベストエフォート」という表現が使われることがある。しかしこの表現は、極力避けるべきである。例えば「ベストエフォート」という表現が使われた場合、ユーザー側は「稀には障害が発生して止まるかもしれないが、自分には関係ないだろう」という期待を持つかもしれない。一方システム運用者は「安定稼働に全力を尽くすが保証はできない。サービスの提供を止めることもあり得る」と考えるとする。実際に運用が開始されてこの違いが明確になり、問題を起こすことがあり得る。

3. サービス提供時間設定の例として、次のような表現がある。
   - オンライン稼働時間　:8:00～20:00
   - 稼働曜日:　月曜日～日曜日（土日、祝祭日も稼働）
   - 稼働日数:　363日（年に2回の保守停止がある）

10.7 保守・レスポンス・タイム

1. レスポンス・タイムとは、オンラインでデータなどを入力してから、結果が出力され始めるまでの時間をいう。この時間はアプリケーションのロジックや画面の構成などによって大きく変わるのが普通である。したがって一律にこの時間を設定することは現実的ではなく、処理ごとに目標値を決めるのが妥当な方法といえる。

2. またこの数字は、サーバの構成、ネットワーク環境やユーザー側のシステムの構成などでも大きく変わるのが普通である。そのためこの数値を設定するに当たって、運用環境やユーザーの環境などについて前提条件をシステム・オーナと取り決めておくことが必要となる。

3. この定義の例として、次のような表現がある。
   - ログイン・メニューの遷移： 2秒
   - 個別照会： 2秒
   - 一覧表示(最大20件)： 5秒
   - 登録処理： 3秒
   - 複雑条件検索： 10秒
   - 回線の前提： ブロードバンド。実効速度は1Mbps 以上。
   - PC スペックの前提： Core i3、メモリーは4GB、Windows10 Proffesional。
   - サーバ等の前提： 本番環境。端末数も実稼働時と同じ(数千台以上のこともある)。

10.8 性能・バッチ処理の実行時間

1. バッチ処理の実行時間はシステムの安定性、継続性に大きく影響するので、エラー発生時のリカバリーも考慮して、予め開発側と運用側で取り決めておく必要がある。またシステムによっては、他のシステムとの連携についても検討する必要がある。この他のシステムが自社だけでなく外部のシステムのケースもあり得るので、注意が必要である。

2. この定義の例として、次のような表現がある。
   - バッチ稼働開始時間： 24:00
   - 完了限界時間： 28:00
   - 他のシステムとの依存要件： なし。

10.9 性能・デリバリータイム

1. デリバリータイムとは、ユーザーが入力情報を入力し終わってから、特定の担当者の手元に適切なアウトプットが届くまでに要する作業時間全体をいう。これについては定義が漏れやすいので、注意する必要がある。

2. デリバリータイムを短くするとコストがかさむことがある上、発送物の品質劣化や誤送などのリクスも

高まる可能性がある。これらの要素も勘案し、適切な時間／期間を決定しなければならない。

3. この定義の例として、次のような表現がある。
   - ○○表のデリバリータイム： データ入力翌々日15時までに、○○担当者の手元に届くこと。
   - 契約者向け○○はがき： 処理の翌日に発送できる状態にすること。

10.10 キャパシティに関する要件

1. 昨今、システムのリソース不足が原因になったシステム障害が目立つようになった。この要因でトラブルが発生した時、一般にリカバリーに時間がかかるという特徴があり、場合によれば利用制限をせざるを得ないことがあって、ビジネスの遂行に影響を与えることにもなりかねない。したがって、キャパシティ・プランニングの重要性を十分に認識しておかねばならない。

2. リソース不足を生む要因として、次のようなものを挙げることができる。
   - キャパシティ要件の予測の誤り
   - リソース設計の誤り
   - 運用開始後の監視の不備

3. キャパシティに関する要求には、次のような項目が含まれる
   - データ量
   - ユーザー数
   - トランザクション数
   - アウトプット数
   - インプット数

10.11 データ量

1. データ量とは、システム内に格納しなければならないデータの量を指す。これは、システム・オーナが、ビジネスの観点から事前に見積もっておく必要がある。システムの稼働開始時の量と共に、その後の推移も予測しておく必要があるが、どれくらい先まで見積もっておくかはケース・バイ・ケースで異なる。しかし少なくとも稼働開始後の直近のピーク時（年度末など）のデータ量と、1 年後の総データ量程度は算出しておくことが望ましい。

2. この定義の例として、次のような表現がある。
   - 契約件数： リソース時 1,000 千件、12 月末（初回ピーク）1,200 千件、1 年後 1,500 千件

10.12 ユーザー数

1. ユーザー数については、ユーザーの種類と、その種類毎の数について定義し、システム開発者、システム・オーナ、システム運用者の 3 者で合意を取付けておくことが必要である。

2. ユーザーの数はシステム上のデータベースの容量だけでなく、登録事務の体制やソフトウェアのライセンス費用などに対しても重要なファクターである。またユーザーの種類は稼働開始後のトラブル発生時での影響範囲を特定する上で役立つ。

3. この場合も稼働開始直後の数値と共に、その後の一定期間の推移を見る必要がある。

4. この定義の例として、次のような表現がある。
   - インターネット経由　稼働開始時:20 万ユーザー(不特定)、年内ピーク時:30 万ユーザー(不特定)、1 年後:40 万ユーザー(不特定)
   - 社内ユーザー:常時 3 万ユーザー
   - 外部(特定企業)接続:常時 500 ユーザー

10.13 トランザクション数

1. トランザクション数は、システムのキャパシティ・プランニングやリソース設計に最も大きな影響を与える重要な要件である。事前に予測し、定義することが難しいが、同時に予測を誤った場合最も大規模なトラブルに繋がりやすいファクターである。

2. また数字として何を示せばよいのかが、システムの構成や採用するアーキテクチャによって異なるという側面を持つ。例えば、次のような数字があり得る。
   - システムに入力されるトランザクション数
   - 単位時間あたりのページビュー(画面遷移)数
   - HTTP プロトコルのリクエスト数

3. また一般にトランザクションには集中する時期や時間帯があるため、年間、月間、および一日を通してのピーク時を明確にし、その時点でのトランザクション数を明示する必要があある。

4. この定義の例として、次のような表現がある。

- ピーク時の定義： 年間 20%が◯月に集中、月間の 10%が◯日に集中、1 日の 20%が◯時に集中
- アプリケーショントランザクション数(画面遷移数)： 300TPS
- HTTP リクエスト数： 1,000RPS

10.14 アウトプット数

1. アウトプット数とはシステムの処理を基に、印刷物やメールなどの形で最終的にアウトプットされるものの量を表す。システム内部の処理と比較して、印刷処理や発送処理は格段に時間がかかるという特徴がある。したがってこのアウトプット数は、業務設計に大きな影響を及ぼす。

2. この見積もりを誤ると想定した時間内に業務が終了せず、ビジネスに大きな損害を与えることがある。また発送処理を伴うアウトプット処理では、その件数が運用コストに直結することになる。

3. システムの設計や開発段階では、ソフトウェア技術者はデータを出力するところで検討を終えてしまうことが多い。しかし全てのアウトプットを正確、かつ適切に担当者の手元に届けるところまでを視野に入れて当たれば、システムの実現形態が変わることもある。

4. アウトプット数はこのように、たいへんに重要な定義項目である。この定義の例として、次のような表現がある。
   - ◯◯帳票のアウトプット数　印刷数:10,000 千枚／日、発送数:1,000 千通／日
   - ◯◯はがきのアウトプット数　印刷時:5,000 千枚／月、発送ス:5,000 千通／月

10.15 インプット数

1. 情報システムへの入力は、その比重を急速にオンラインに移している。しかし入力原票を基に専任者が手作業で入力する一括入力は、避けられないものと考えておかなければならない。このような場合一般に、入力原票の文字種、文字のサイズなどが、業務効率に大きな影響を与える。ユーザーに満足してもらえる情報システムを構築するためには、こうした点にも注意を払わなければならない。

2. この定義の例として、次のような表現がある。
   - ◯◯データのインプット数： 5,000 件／月

10.16 安全性に関する要件

1. ビジネスの継続は、企業が果たすべき社会的責任の中、最も重いものである。その企業のビジネスの遂行を支える情報システムの安全性の確保はこの観点から、極めて重要な要件であるといえる。

2. 安全性を高めるには、コストがかかる。したがってリスク評価を行いながら、適切な対応を撮る必要がある。

3. システムの安全性には、次の3つの要素がある。
   - 機密性(Confidentiality);情報にアクセスすることを認可されたものだけがアクセスできることを確実にすること。
   - 完全性(Integrity):情報及び処理方法の正確さ及び完全である状態を安全防護すること。
   - 可用性(Availavility):許可された利用者が、必要なときに情報にアクセスできることを確実にすること。

4. この 3 つの要素を、情報セキュリティ CIA と呼ぶ。これらは 1989 年に発行された OSI(オープン・システム・インターコネクション:開放型システム間相互接続)に関する ISO の規格で内容が定義されており、その実施は 1992 年に OECD(経済協力開発機構)が定めた情報セキュリティ・ガイドラインで勧告されている。

5. 各企業はこれらを基に対策を抽出し、優先順位を決めて、コストも考慮しながら具体的な対応を決定する必要がある。上記 3 要素から以下の観点で安全性要件を明確にし、対応することがある。
   - セキュリティ
   - 停止許容時間
   - 有事対策

10.17 セキュリティ

1. ビジネスの継続は企業にとってたいへん重要な要件であるが、セキュリティ対策の不備はこのビジネスの継続性を脅かす可能性がある。最近は、ウィルスやスパイウェア、クラッキングによる情報漏えいやデータ改ざんといった外部からの故意による犯罪だけでなく、開発者や運用者、あるいはユーザーの過失などによって結果的に情報漏えいが起きたり、あるいはシステム停止が起きるというケースが発生している。

2. こうした事態を防ぐには、企業全体のセキュリティ・ポリシーを定めて、各システムをそれに準拠させることが必要である。ただし、静的なページを表示させるだけの簡易的な Web サイトと多量の個人情報を入出力するオンライン・システムに全く同じセキュリティ対策を施すことは、適切なリスク・コントロールがなされているとは言い難い。セキュリティ・ポリシーを基本の指針とし、これに則ってシステム毎にリスク評価を行いながら適切なセキュリティ要件を決めなければならない。

3. セキュリティ要件の中には、「パスワードの桁数や文字種」、「パスワードの有効期限」、「ユーザー・アカウントのロック条件」などのようにユーザービリティとトレードオフになるものが多い。したがってこのようなセキュリティ要件を決めるに当たっては、システム・オーナの合意を取り付けておくことが不可欠である。機密の保持が高いレベルで要求される情報システムの場合、「ワンタイム・パスワード」を採用することが必要かも知れない。

4. これまでログを取得する目的は、性能評価や統計処理の資料取得などが主なものだった。しかしここに来てトラブル分析などのため操作証跡を残すために組み込まれるケースが増加してきている。さらに最近は存在証明（整合性がとれた確実なタイムスタンプの取得）、完全性証明（改ざん防止）などのためのログ取得があり、短期間で必要な情報を抽出することが求められている。このような立場・観点でログを取得するに当たっては、膨大なリソースを必要とし、システム設計や運用コストに大きな影響を及ぼすことになる。したがって要件定義の段階で、明確に定義しておくことが必要である。

5. セキュリティ要件の定義例として、以下のような表現がある。
   - パスワードの桁数： 7 桁以上
   - パスワードの有効期限： 3 ヶ月間
   - ユーザー・アカウントのロック条件： ログイン認証を 3 回連続して失敗した場合

10.18 停止許容時間

1. サービス期間中にシステムが停止した場合の、復旧までの停止許容時間をあらかじめユーザーやシステム・オーナと取り決めておくことが必要である。ユーザーの立場からすると、停まらないに越したことはない。しかし、高可用性を確保するためには、一般にたいへんなコストがかかる。経済合理性も考慮した、一定のレベルで合意を得ておくべきである。

2. 停止許容時間にあわせて、稼働率も明確な数字で合意しておくことが望ましい。停止許容時間の

定義例として、以下のような表現がある。

- オンライン停止許容時間: 年間 1 日(翌日までに復旧)
- システムの稼働率: 99.99%

10.19 有事対策

1. 企業にとって、「ビジネスの継続が重要」である。特に金融機関など公共性が高い企業には、今これが強く求められている。これまでは、大規模地震などを想定した対策が一般的だったが、ここに来てその想定範囲が広がり、テロやサイバーテロなどその対象にしたディザスタ・リカバリーの機能が求められている。

2. ディザスタ・リカバリーの機能とは平時には使用されることがない機能であって、そのために特別の情報システムを構築することになる。これには当然コストがかかるが、そのコストもリスク対策コストの一部と認識し、対応しなければならない。

3. つまり企業全体のリスク・コントロールの観点から有事対策についての方針を決め、この方針に則ってシステムのリスクを評価し、個々のシステム毎にリカバリ・プランを検討し、実現することが必要である。どこまでやるかによって投資する金額が大きく異なるのが、この有事対策についての特徴の 1 つということになる。

4. 有事対策の定義例として、以下のような表現がある。
   - バックアップセンターで、n 時間以内に業務を継続する。
   - バックアップデータを、週次で遠隔地保管する。

10.20 コストに関する要件

1. プロジェクト開始時には、開発コストやハードウェア/ソフトウェア導入のための一時コストに焦点が当たりやすい。しかしシステム稼働開始後の運用コストも、たいへん重要である。システム。オーナは「稼働開始後も、延々とコストがかかり続ける」ことを認識しなければならない。またそのために、要件定義時に運用コストについても明確な合意を取り付けることが重要である。

2. コストについての定義例として、以下のような表現がある。
   - 開発コスト 開発規模: 10k ステップ、300 人月、開発ツール: 500 千円×20 ライセンス、・・・。

- 運用コスト: 年間 30,000 千円
- その他: 稼働後 5 年目に、保守期限切れ対応で一時コストがかかる見込み。

10.21 保守に関する要求

1. 非機能要件の一部として、保守についての要件を明示するべきという考えがある。広い意味でこれも品質要求の 1 つといえるが、開発を委託する立場で、その情報システムの保守についてのあるべき姿を具体的に明示することが必要である。

2. 保守についての要求例として、以下のような表現がある。
   - 1 つのクラスに含まれるメソッドは、原則として 10 個以下とすること。
   - モジュールの複雑度は、原則 20 以下とすること。
   - モジュールの凝集度は、原則として”手順的凝集度”以下にはしないこと。
   - 処理と管理は、明確に分離すること。
   - 関数の呼び出しの深さは、原則として 5 以下に留めること。
   - タスク間でアクセスし合うグローバル・データは、原則として作らないこと。
   - 上記原則に違反する場合は、品質検討会議で了解を取ること。

10.22 非機能要求の適用について

1. 機能要求とそれに関連した仕様は、そのままアプリケーションの設計に結びつく。一方の非機能要求は再構成されて、アーキテクチャ設計に用いられることになる。

2. アーキテクチャ設計とは難し言葉だが、ここでは「アプリケーションに関わるソフトウェアを円滑に動かす仕組み」と定義しておきたい。例えば、以下のような事項は全て、アーキテクチャの取扱範囲となる。仮に、今対象としている情報システムの実現形態として、クライアント／サーバ方式(Web コンピューティング)を選ぶとする(この選択自身が、すでにアーキテクチャ設計の一部である)。その上で、
   - どんな種類のサーバを用意するか。
   - サーバとクライアント機に、どのコンピュータを使用するか。
   - サーバとクライアント機の OS に、何を使用するか。
   - 全体の機能を、どのように複数のサーバ機とクライアント機で分担するか。
   - その間のネットワークにどの方式の、どの帯域幅のものを使うか。

- データベースでのデータの格納方式に、何を使うか。
- データベースのバックアップ／リカバリーは、どのように行うか。
- どの部分に市販のパッケージを使い、どの部分を手作りするか。

・・・

3. アーキテクチャは、「基盤」と言い換えても良い。これらの上の問題に的確に答えを出すために、アーキテクチャを設計する人（アーキテクトと呼ばれる）は非機能要求などを基に、「基盤要件」と呼ぶ次の要件を明確にする。
   - 可用性
   - 性能／スケーラビリティ／キャパシティ
   - 拡張性
   - 管理性
   - セキュリティ
   - 前提・制約条件

4. 前提・制約条件には、さらに以下のような事項が含まれる。
   - 使用を前提とする市販の製品
   - 既存システムとの接続
   - アプリケーションの仕様
   - 運用要件
   - 保守要件
   - 機器等の設置環境
   - 法規制

   ・・・

5. 良いアーキテクチャはアプリケーションのソフトウェアにとって、良い稼働環境などを提供するものである。単にアプリケーションの稼働環境だけではなく、情報システムとしての拡張性、ユーザービリティ、セキュリティ、障害の起きにくさと復旧の容易さ、それらを統合した運用の容易さなど、非常に広範囲に影響を及ぼすことになる。逆説的な言い方になるが、あるいはその存在を意識させないアーキテクチャが、一番優れたアーキテクチャということになるのかもしれない。

6. このように非常に大切なアーキテクチャ設計のベースになるものであるから、非機能要求も機能要求と同様、業務の特質をふまえ、しっかりと定義しなければならない。

10.23 開発リスク軽減の方法

1. ソフトウェアの開発を外部に委託する場合、その開発にはリスクを伴う。開発を受託するソフトウェア会社の立場からすると、そのリスク分は開発受託の金額の一部として含まれており、リスクが高いほど受託金額が高くなるということになる。したがって発注者側と受託者側で共同してこのリスクを減少させることができれば、双方にとってメリットがあることになる。

11USDM を使用した要求仕様書の作成（演習）

11.1 目的

1. ユーザーが準備する要求仕様書から、ベンダーの作成する要求確認書、さらに設計書へと設計が進むに連れて仕様が固まってくる。要求仕様書および要求確認仕様書にかかれている情報をもとに下記のプロセス・フローに則り設計仕様の確認作業を進めることで仕様書の充実、設計内容の緻密性を図る。

11.2 要求には「理由」がある

1. 要求の表現が不適切であれば、あるいは、そこで表現されている要求が依頼者の本当の希望を表現できていなければ、その後の仕様の抽出で的を外し不足が発生する。そして、不足を補うために仕様変更が多発する。

2. 人の話を聞いたり本を読んだりして「わかったと思ったのは、その人がわかったと思った範囲だけ」の話であり、わかった範囲が話してや聞き手の認識と一致していないことには気づかない。この状況が、「コミュニケーションの不完全性」であり、この不完全性を調整する働きが「理由」である。

3. 要求にはそれが必要な理由や背景があり、要求と一緒に捉えることが重要である。依頼者は要求をうまく表現できないといわれるが、理由と組み合わせることで依頼者の思いがどこにあるのか見えるようになり、要求の表現が修正される。

    - 理由によって要求に対する関係者の認識のズレを調整できる
    - 要求全体に対する理由の他に、特定の動詞に理由が存在することもある
    - 必要かつ有効な仕様が引き出される
    - 設計上で工夫されることがある
    - システムの非機能要求が理由になることもある
    - 理由があることで要求の意義が明確になり、要求が洗練される
    - 思い込みや勘違いを防ぐことができる
    - 不純な要求を排除できる

4. 要求に対して適切な理由が付かない場合は、要求の表現が曖昧か、要求そのもの不純な動機で

出されている場合が多く、そのままでは仕様を歪めてしまうこともある。このため、要求には必ず「理由」をセットにする。また、要求に付随する「理由」は、必ずしも1つとは限らない。

たとえば、

「社員であれば、社内ネットワーク上で Web 操作によって図書を借りることができる」

という要求の理由としては、

- 「残業時間で担当者がいなくても貸出しの手続きができるようにしたい」
- 「電話での申し込みで起こっていた、担当者が聞き間違うことをなくしたい」
- 「貸出し申し込みが重なったときに誰が早かったのかわからなくなるのを防ぎたい」

といった理由が考えられる。

また、

「インクの残量を調べて、印刷できない可能性があれば印刷を依頼した人の PC に知らせてほしい」

と理由を表現し、相手にイメージを伝えたと思うかもしれないが、依頼者とのヒアリングの中でこの要求がどんな状況で出てきたのかを探った結果、この人は、以前に 200 ページの文書を印刷した際に、時間がかかるのでしばらくコーヒーを飲んで休憩したあとに戻ってきたら、半分ぐらいのところからインク切れによって文字がかすれて見えない、という経験をしていてそれがこの要求を発した理由だったことが判明したとすればどうだろうか。その結果、

- 印刷中に文字がかすれたくないから

というのが理由だと分かることもある。

以上をまとめると、一つの文章で表現することもできる。

- 印刷中に文字がかすれたくないから、インクの残量を調べて、印刷できない可能性があれば印刷を依頼した人の PC に知らせてほしい

USDM 表記法では、要求と理由をセットとし、次のように表記する。

要求	PRT07	インクの残量を調べて、印刷できない可能性があれば印刷依頼者の PC 上に知らせてほしい
	理由	印刷途中で印字がかすれたくないから

今回の場合、設計者によっては、「印刷途中で文字がかすれたくない」という理由を解消することは技術的に困難であると気づきます。要求によっては、こうして実現ができないものは削除されるが、この要求の場合は、印刷できない可能性があることを PC 上に表示するという要求そのものは、欲しい機能と思われる。このような場合は、「要求」や「理由」の置き換えを図る。

要求	PRT07（固有 ID）	印刷開始時に、インクの残量を調べて残量が少ないときは交換の必要性を画面に表示し、印刷中にインクがかすれる危険があると判断されたときは、印刷を停止させる。
	理由	印刷途中で印字がかすれたくないから

11.3 演習①（要求から理由を探る）

1. 演習「在宅介護サービス支援システム」(付1)の“現状の課題(まとめ)“より、要求を見つけ出し、その理由をセットで書き出しなさい。

要求		
	理由	

要求		
	理由	

要求		
	理由	

11.4 要求を上手く表現するには

1. 要求を上手く表現するには、要求の文章に含まれる動詞に着目し、目的語を上手く使って振る舞いを表現する。タイトルや簡単な動作(“名詞＋する”のような)だけでは、要求の役割を果たさない。

×	要求	DA07	計測データの表示
×	要求	DA07	計測データをリアルタイムに表示する

○	要求	ALM.03	計測データを受信し、平均値を算出しながらリアルタイムに表示したい。その際、異常値が検出されたときは警告を表示したい

要求(振る舞い)に含まれる動詞をすべて表現することを目指し、下線部分については仕様を展

開する。つまり、動詞を引き出してしまえば仕様は漏れない。

11.5 理由に要求が混在する

1. たとえば、次のように理由のところに要求が書かれている場合がある。理由に要求の必要性と、実現したいことが記述されている。これは、まさに要求であり見直しが必要となる。

＜理由に要求が混在する例＞

要求	SET5	設定画面を修正すること
	理由	以前のバージョンとの違いを明確にするため設定画面の大きさと、項目の配置を変更する

＜修正版＞

要求	SET5	設定画面の大きさと、項目の配置を変更する
	理由	以前のバージョンとの違いを明確にするため

11.6 演習②（要求をうまく表現するには・要求から理由を探る）

1. 演習（要求から理由を探る）を見直して、要求に含まれる動詞の表現の改良および、理由に要求が混在する場合は修正をしなさい。

要求		
	理由	

要求		

	理由	

要求		
	理由	

11.7 要求の仕様化

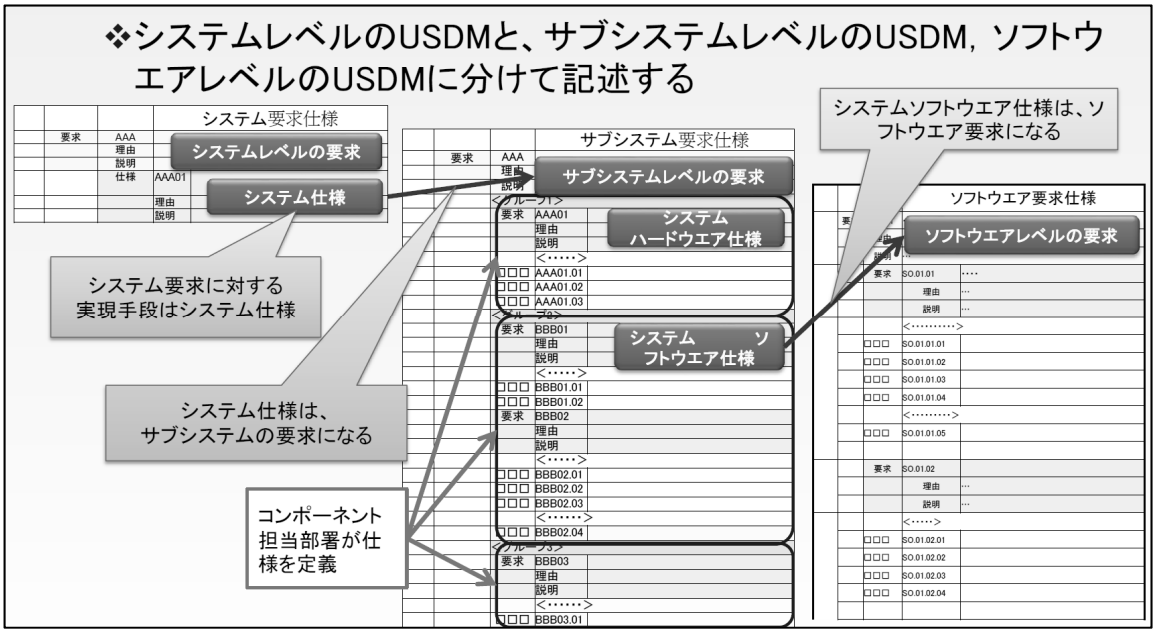
1. 要求の仕様化は、要求が適切に表現されていなければ仕様は正しく引き出すことができない。ソフトウェアとしてソースコードに変換されるのは仕様であり、正しく仕様化できなければモレが発生し後戻り作業が発生する。要求の仕様化を行うには、要求の中にある動詞の単位で仕様を設定する(目的語も仕様をもつことがある)。

＜USDM 表記法による要求と仕様の基本形＞

要求	CH-02	印刷開始時にインクの残量を調べて、最後まで印刷できない可能性があれば印刷依頼者の PC 上に知らせてほしい
	理由	印刷内容を知っている人に、最初に印刷できない可能性を知らせたい
	説明 ※説明欄については後述	印刷途中の監視機能は既に提供されており、これは印刷開始時に知らせるもの
□□□	CH-02-01	印刷開始の操作を完了した時点でプリンターと交信してインクの残量(全カートリッジ分)を知る。
□□□	CH-02-02	「注意」または「警告」のアナウンスを出したときは、「確認」の操作

		によって印刷処理を続ける

2. 要求仕様は「ビジネス要求」にはじまり、「システム要求仕様」「サブシステム仕様」などに階層化される。演習の過程では「ビジネス要求」の段階であり、ここでいう仕様とは、「ビジネス要求」の下の階層である「システム要求仕様」に含まれる。(下図:派生開発推進協議会カンファレンス 2015)



(派生開発推進協議会カンファレンス 2015 より)

11.8 必要なら説明をつける

1. 一般に要求仕様書のなかには、仕様の記述のなかで、その仕様の理解を助けるために用語の意味を説明したり、動きの事例を付けていたりするケースも少なくない。このような場合、注意して読まないとそれが仕様なのか説明なのか判断を間違える場合がある。

> ＜水量の表示＞
> 貯水タンクには、水の量を計るために 50 センチ間隔で 10 個のセンサーが付いており、水を検出するとセンサーが「ON」になり、センサーのデータから水の量を知ることができます。このときの様子が PC 画面上のタンクの図に反映されます。

これはごく普通の表現であるが、50 センチ間隔とか 10 個のセンサーが付いているというのはタンクを製造する人に対する仕様であり、ソフトウェアに対する仕様ではない。ソフトウェアエンジニア向けに特にセンサーの意味を知らせる必要がある場合、「説明」というキーワードを使ってセンサー

の個数や間隔を、図を付けて説明すると効果的である。次の例は、要求欄にデジジョンテーブルで分かりやすく説明を記載したものである。

＜デシジョンテーブルで表した例＞

<table>
<tr><td>要求</td><td>COM02.01</td><td>接続待機中、接続中、装置切り替え操作、接続不可通知、電源断ボタンの組み合わせと、そのときの処理(振る舞い)の関係を以下の表に示します。
<table>
<tr><th>接続
待機中</th><th>接続中</th><th>装置切り
替え操作</th><th>接続不可の通
知</th><th>電源断
のボタン</th><th>処理</th></tr>
<tr><td>○</td><td></td><td>○</td><td></td><td></td><td>操作無効の表示</td></tr>
<tr><td>○</td><td></td><td></td><td>○</td><td></td><td>接続失敗の表示</td></tr>
<tr><td>○</td><td></td><td></td><td></td><td>○</td><td>電源断のシーケンス</td></tr>
<tr><td>：</td><td>：</td><td>：</td><td>：</td><td>：</td><td>：</td></tr>
</table>
</td></tr>
</table>

また、「要求」が明確に表現されていないことが問題である。ソフトウェアに対する要求としては、

- 「各センサーの「ON／OFF」状態からタンクの水量を表現する」

あるいは、

- 「各センサーの「ON／OFF」状態からタンクの水量を判断して、PC 上のイラスト上に表現する」

が、該当すると思われるが、何がソフトウェアに対する要求で何が説明なのか不明である。しかも、PC 上に表示することが求められているのか、センサーで水の「位置」を知るだけでよいのかも不明である。

2. 仕様のなかでも、何か説明が必要であれば「説明」というキーワードを付けて記述するとよい。

＜要求と仕様に説明を付けた例＞

<table>
<tr><td rowspan="2">要求</td><td>CH-02</td><td>印刷開始時にインクの残量を調べて、最後まで印刷できない可能性があれば印刷依頼者の PC 上に知らせてほしい</td></tr>
<tr><td>理由</td><td>印刷内容を知っている人に、最初に印刷できない可能性を知ら</td></tr>
</table>

		せたい
	説明	印刷途中の監視機能は既に提供されており、これは印刷開始時に知らせるもの
	＜チェックのタイミング＞　※後述	
□□□	CH-02-01	印刷開始の操作を完了した時点でプリンターと交信してインクの残量(全カートリッジ分)を知る。
	理由	印刷ボリュームと比べて判断する必要があるから
	＜確認操作＞　※後述	
□□□	CH-02-02	「注意」または「警告」のアナウンスを出したときは、「確認」の操作によって印刷処理を続ける
	理由	インクの交換はこの間に行い、交換後に「確認」操作によって印刷を続ける

3. USDM では、仕様には”□(チェックボックス)”の記号を先頭に付けて、この項目を仕様として認識していることを識別できるようにしている。また、他の文書との照合に使用する。たとえば、仕様レビュー済みかどうか、仕様が設計やテスト仕様のなかで確実に扱われていることを確認したことを示す。現実には3つくらい必要である。

11.9 用語集への展開

1. 要求仕様書に書かれた「説明」は、開発が終わったあとは要求仕様書から切り離し用語集へ移すことができる。用語集へ移すことで事前の学習テキストとして活用でき、ソフトウェア開発時の作業を減らすことが可能である。

要求	MAL01-02	いくつかのキーワードを組合せて検索できる
	理由	可能性のあるキーワードで最初から絞り込んで探したい
	説明	キーワード検索の仕様は、用語集「キーワード検索」を参照のこと

2. 用語集は共通した知識となるだけでなく、要求仕様書の作成時間も短縮できる。また、周囲のメンバの知識レベルの合わせることができる。さらに、外部のソフトウェア企業に発注する場合も効果を発揮する。

3. 日本の組織ではこの種の「用語集」を目にすることはほとんどない。これは、個々の機能に対する十分な仕様書が作れないことが影響していると考えられている。適切な内容を持つ仕様書が出来

上がれば、それを切り出して用語集やテキストにすることで省力化につながる。

11.10 演習③（要求から仕様を抽出する－1）

1. 演習「在宅介護サービス支援システム」（付1）のシステム要件から、先ほどまでの演習で拾い出したビジネス要求に関連するシステム要求を抽出しなさい。※「要求」とは、一般に顧客が何を実現してほしいかを表す際に使用する言葉であり、「要件」とはシステムが何を実現するかを表現する場合に用いることとする。本書では「要求」という表現を統一して使用する。

ビジネス要求の番号	システム要求

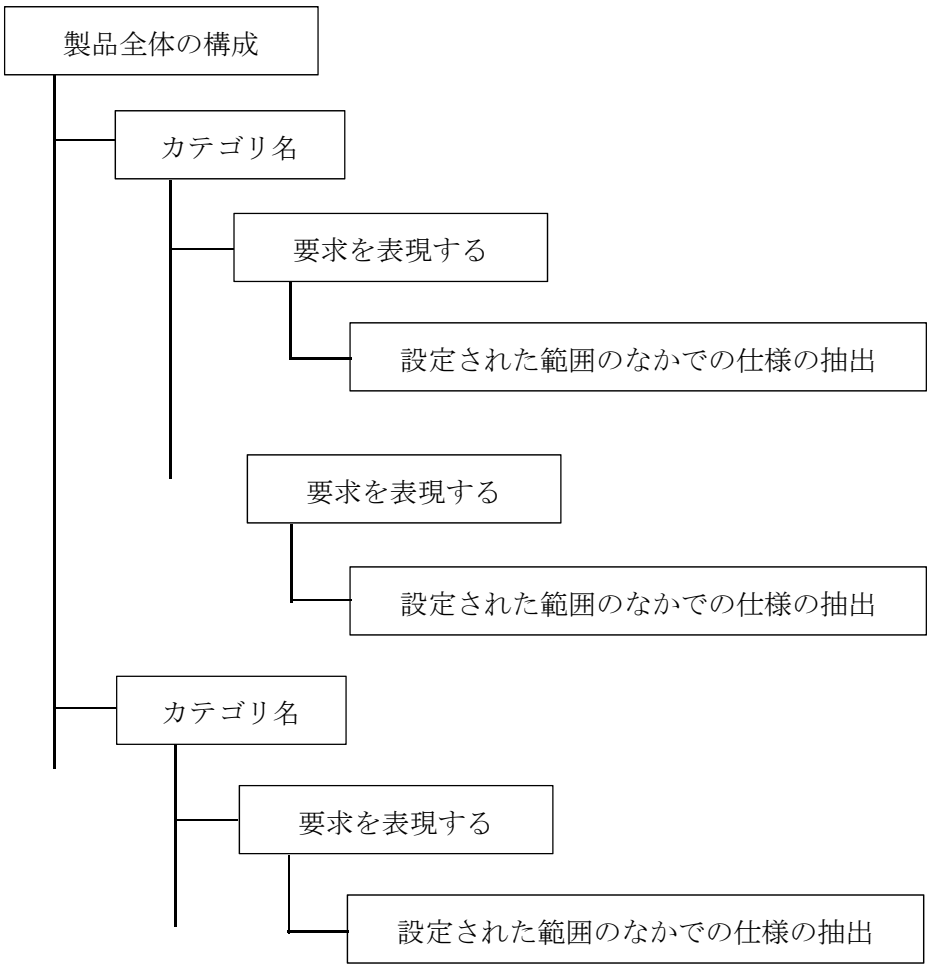
11.11 演習④（要求から仕様を抽出する－2）

1. 演習③（要求から仕様を抽出する－1）で抽出したシステム要求について、画面イメージ（案）を参考にして、USDM 要求仕様書にまとめなさい。※画面イメージに不足しているものがあれば、機能追加しても構わない）。

カテゴリ名	要求	要求番号 仕様番号	説明	備考
Req01	要求	SYS01	申し送り内容を本文に記して、ヘルパー宛へメールの一斉配信を行う	Bus01
		理由	1人の要介護者に対し複数のヘルパーが交代で介護担当する際、申し送りの抜けが発生することがある。また、複数の担当者がサービス記録表に記載する方式では、最新の情報を共有できない場合がある。	
		説明		
	□ □ □	SYS01.01		
	□ □ □	SYS01.02		
	□ □ □	SYS01.03		
	□ □ □	SYS01.04		
	□ □ □	SYS01.05		
	□ □ □	SYS01.06		
Req01	要求	SYS02	事務所スタッフが電話を受けてシステムに入力し、情報を蓄積する	Bus01、画面仕様なし
		理由	1人の要介護者に対し複数のヘルパーが交代で介護担当する際、申し送りの抜けが発生することがある。また、複数の担当者がサービス記録表に記載する方式では、最新の情報を共有できない場合がある。	
		説明		
	□ □ □	SYS02.01		
	□ □ □	SYS02.02		

11.12 要求のカテゴリ化

1. 要求仕様書は、それ自体が明確な思想（体系）の下に整理されていることが重要である。要求仕様書全体をどのように構成するかを検討し、要求を階層化して個々の要求の範囲を狭めて仕様を引き出していく。漏れにくい構造というのは漏れていることを発見しやすい構造でもあり、仕様などの追加に対しても、それがどこに収まるべきかがわかりやすい構造ということもできる。



① 機能で分割する例

単に機能を羅列するだけでは、担当者の注目度や意識の状態によって機能の並び方が変わる。要求や仕様は互いに絡んでいることが多く、並び方によってこの文書を見る人の連想脳に影響する。たとえば、構造化手法のアーキテクチャ分析で使用するテンプレートを参考にする方法がある。テンプレートを使用すると、要求の漏れを防ぐことができる。

＜アーキテクチャ分析のテンプレート図＞

	保守・診断処理	

このテンプレートは、リアルタイムで拡張された構造化分析手法のなかの、アーキテクチャ分析のなかで使用されるものであり、組込みシステム向けに考えられたもの。その他のシステムも概ねこのような分類でシステムを捉えることができると考えられる。

たとえば「入力処理」という分類に該当する機能をリストアップする。

- 写真データの入力
- 動画データの入力
- 音楽データの入力

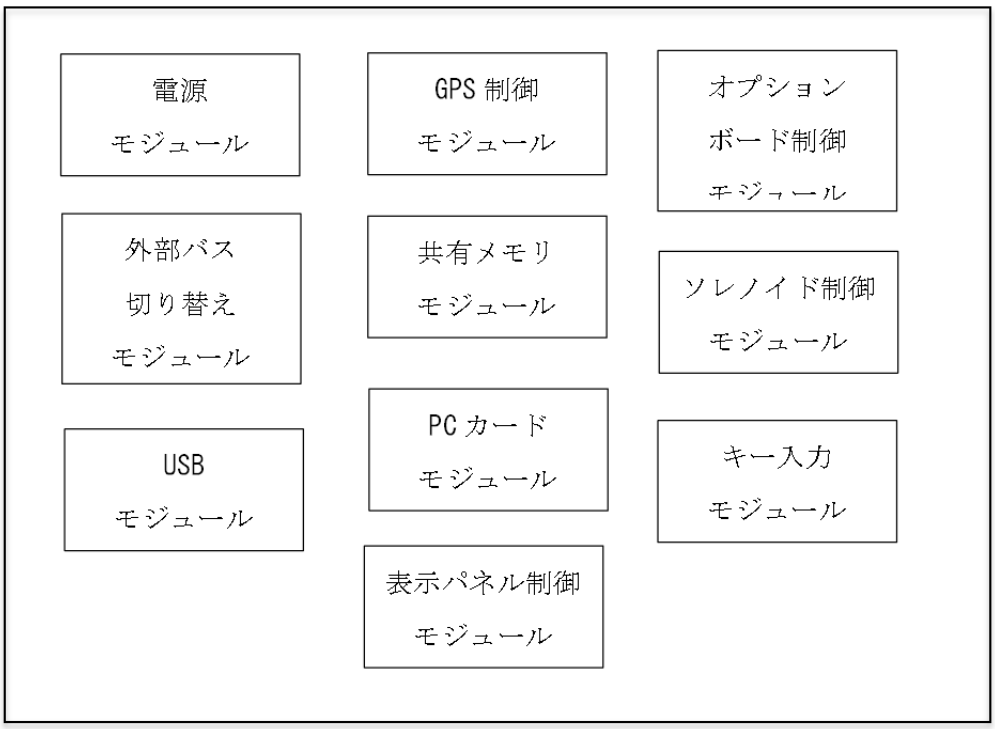
UI 処理、変換処理、保守処理では、次のようにリストアップする。

- 地図データの表示
- 地図データの更新
- 電話番号による検索

② 構成で分割する例

次の図は、回路設計者が設計の過程で残すブロック図であるが、これを要求仕様書のカテゴリ分割に活用することができる。コンテキスト図やカテゴリ分割のテンプレートなどは、顧客を巻き込んで共通の認識の上でこれからの要求の仕様化作業に入るのに大いに活用できる。できるだけ図を活用し、顧客側の担当者にも入りやすい状態を作ることが重要である。

＜メイン基盤回路図＞



11.13 品質要求

1. 多くの要求仕様書で抜け落ちるものが「品質要求」である。文書名が「機能仕様書」となっているケースでは、「性能」などの機能に関連した品質以外はほとんど扱われることがない。ソフトウェアを開発する際には品質要求を付けるという認識が稀薄なのが原因である。

2. 機能仕様書では、保守性などの「ソフトウェアの作り方に関する品質要求」や、操作性や対故障性、応答性などの「ソフトウェアの使い勝手に関係する品質要求」を記述する場がなく、このような品質要求は漏れてしまう。その結果、機能に関する要求と仕様だけが記述され、それが実現されてしまう。

3. すべての機能を実現したとしても、一つの品質要求が満たされなければソフトウェア(製品)としての価値が失われる。品質の未達成の問題は、個々の機能の完成度よりも深刻な事態を招く可能性が高い。それにも関わらず、品質に対する意識は低いのが現実である。

4. 「品質」という概念は抽象的であり、次の表のように、「保守性」「操作性」「応答性」「対故障性」「理解性」「完全性」「交換性」といった「品質特性」を使用する(5 章　非機能要求の定義方法も参照)。

＜品質特性の一覧　ISO-9000-3、品質特性・副品質特性＞

品質特性	機能性	信頼性	使用性	効率性	保守性	移植性
副品質特性	合目的性	成熟性	理解性	実行効率性	解析性	環境適応性
	正確性	障害許容性	習得性	資源効率性	変更作業性	移植作業性
	接続性	回復性	操作性		安全性	規格準拠性
	整合性				試験性	置換性
	セキュリティ					

11.14 品質要求の仕様化

1. 機能であっても品質であっても、それが要求として表現され適切に仕様化されないものは実現しない。品質要求がまったく書かれていない要求仕様書もある。機能仕様書と違い、品質仕様を実現するにはそれなりの設計技術も必要である。機能仕様の場合は、指定されたファイルからデータを読み出して指示されたようにデータをチェックし、計算ミスがないように注意してアルゴリズムを考えて実装すればよいが、品質仕様の場合はそこで求められる設計技術は異なる。

2. たとえば、パフォーマンスの品質要求として「○○時間内にホストに応答メッセージを返すこと」が求められているような場合、正しいメッセージが正しい返送先に送られたとしても、「○○時間内」に到達しなければ不合格である。これらの処理に関わるタスクのプライオリティや、どのタスクにこれらの機能を振り向けるか、あるいは高速に返送するためにどのようなデータ構造を設計するか、といった設計技術が求められる。なかでも「保守性」は、独自の設計技術が必要になる。

3. 品質要求を仕様化する場合、機能要求と違い工夫が必要である。要求を階層化するときは、下位の要求の持つ範囲の合計が上位の要求が求める範囲と一致する認識で階層化を行うが、品質要求の場合は、必ずしも範囲に収まるとは限らない。

4. たとえば、次の機能要求があった場合、

   - 「指定された一つの画像ファイルを受信し、それを画面に表示する」
   - 「その後は、“次へ”と“戻る”のボタン操作に応じて画面を切り替える」

   機能要求としては実現していたが、画像ファイルが大きいときはすべてを受信するまで何も表示されず、操作した人はいつ表示されるのか分からないといったパフォーマンスに関する品質要求を課していないことがある。

そこで、「ファイルの受信開始から 1 秒以内に最初の画像を表示する」という品質要求を加えると、次のような仕様になる。

＜パフォーマンスの仕様の例－１＞

要求	PAF01	ファイルの受信開始から 1 秒以内に最初の画像を表示する
	理由	すべてのデータを受信するまで表示しないわけにはいかない
	説明	データ全体の受信時間は平均で 20 秒ほどかかる
□	PAF0.01	先頭の画像データの受信が終えた時点で画像の表示を行い、2 枚目以降の画像データは、その間に受信する。
□	PAF0.02	受信中は「受信ランプ」を点滅させ、この間は「次へ」「戻る」の操作は止める
□	PAF0.03	受信終了後は、通常の操作ができる

＜パフォーマンスの仕様の例－２＞

要求	DSP01-02	【時間性効率】「表示」ボタンを押してから2秒以内に画面の表示が完了すること
	理由	ここで表示するデータの処理量が多く、表示が遅いとハングアップしたと勘違いする可能性があるため
□	DSP01-02-1	画像処理するデータ量が「2MB」以内のときは、通常通りに処理する
□	DSP01-02-2	画像処理するデータ量が「2MB」を超えるときは、処理中は「プログレスバー」を表示する

5. 保守性や移植性の品質要求も、仕様化しないと実現しない。ただし、品質を織り込む設計技術が必要である。通常のテストで確認できないが、仕様としての条件は同じで、実現していることの検証は「静的テスト」で行う。

要求	QUA.02	【変更性】ソフトウェアの変更に対する耐久性(崩れにくさ)を向上してほしい
	理由	この5年以上最低30回のバージョンアップをここから展開したいから
□	QUA.02-1	一つのクラスに含まれるメソッドは10個以下とし、超える場合は事

		前に了解をとる
□	QUA.02-2	モジュールの複雑度は17以下を原則とし、超える場合は事前に了解をとる
□	QUA.02-3	モジュールの凝集度も手順的凝集度以下になる場合は事前に検討する
□	QUA.02-4	処理と管理は明確に分離し、関数の呼び出しの深さが5を超えるときは事前に了解をとる
□	QUA.02-5	タスク間でアクセスし合うグローバル・データを作らない。アクセス時間等による制限から、グローバル・データとなることが避けられないときは、品質検討会議で承認されること 【説明】割込処理との間でのみ共有するケースはこの制限を受けない

11.15 要求の階層化のテクニック

1. スムースに仕様を引き出すには、要求が範囲をうまく表現しなければならない。また、範囲を表現したとしても、要求の扱う範囲が広すぎると仕様の抽出が拡散して漏れてしまう。

2. たとえば、一つの機能要求は、その機能を満たすのに必要な仕様が入る「箱」のようなもので、「箱」に書かれた「要求」の表現や範囲（大きさ）が適切であると、その箱に入るべき仕様を拾いやすくなる。

＜要求:メールをキーワード検索して中身を見たい＞

<table>
<tr><td>メールのグループを選択する</td><td>キーワードで検索する</td><td rowspan="2">選択されたメールの内容を表示する</td></tr>
<tr><td colspan="2">検索されたメールを表示する</td></tr>
</table>

単に「メールをキーワードで検索して中身を見たい」という要求に含まれる仕様を抽出するのではなく、さらに「メールのグループを選択する」「キーワードで検索する」「検索されたメールを表示する」「選択されたメールの内容を表示する」といった形で「要求」を小分けにすることで、仕様の抽出を容易にする。

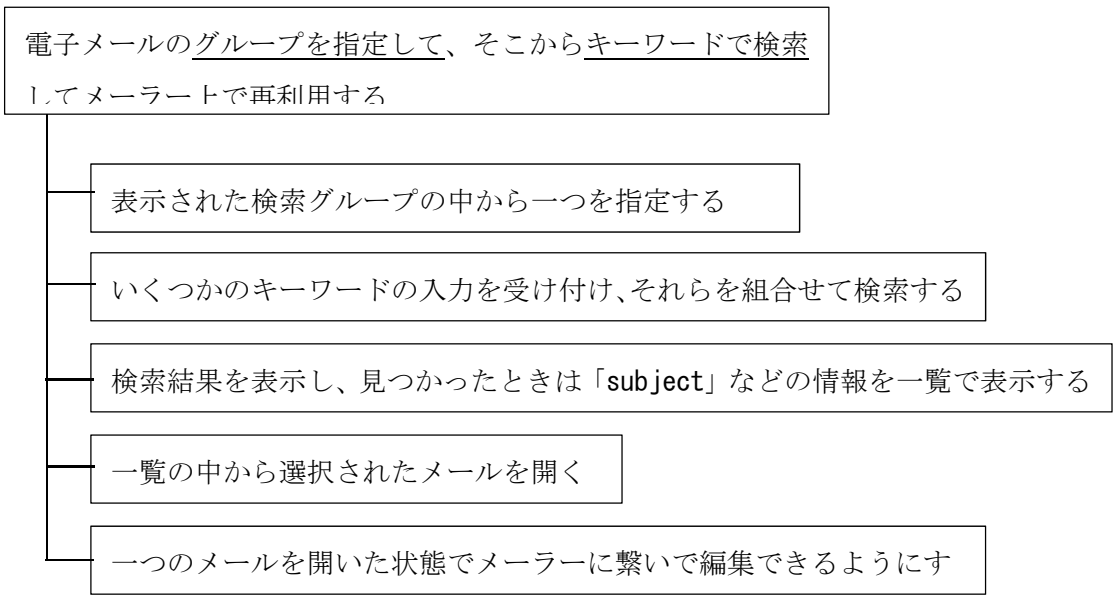
3. これは、構造化設計の「凝集度」の考えに似ており、いくつもの機能を含んだモジュールよりも一つの機能に絞り込む（機能的凝集）ことで、そこで実現すべき処理が明確になるためプログラムの

間違いが減っていくことになるが、要求と要求仕様の関係もこれに似ている。

4. 要求の表現に「範囲」を持たせ、さらに「理由」をつけることで要求のブレを抑えることができる。範囲を適切に設定するため、要求を分割して階層化する。

5. 要求を分割して階層化するポイントは、その要求に含まれるすべての「動詞」を表現することである。上位要求に表現された文章には、その要求の範囲に含まれる動詞をすべて表現しているとは限らない。

＜送受信した電子メールをキーワードで検索したい」を階層化した例＞



6. また、要求には両端の動作が表現されていることがあり、たとえば、「動画像のデータを受信して再生する」という要求は、「受信する」から「再生する」までの間にある動き(振る舞い)を全部含んでいる。

- 動画像データを受信する
- 受信したデータをファイルに保存する
- 保存ファイルを読み出す
- 読み出しながら再生する
- 受信しながらも再生できる
- 生成中にいろいろな操作ができる
    - 一時停止と再開

- 繰り返し再生の指定
- 再生中止
- 画像の調整

＜内包する動詞からの分割階層化した例＞

要求	VIS.1	動画像のデータを受信して一時保存しながら再生する	
	理由	動画像データの受信間隔のばらつきの調整のためと、途中で再生をさかのぼることに対応するために一時的に保存する必要がある。	
	説明	逆に、著作権の制作から再生が成功したデータは確実に削除することが求められる	
□□□	要求	VIS.1.1	接続されたコネクタから動画像データを受信する
		理由	複数のコネクタが存在するため
□□□	要求	VIS.1.2	受信したデータをファイルに保存する
		理由	数ブロック受信するまで一時保存する
□□□	要求	VIS.1.3	保存ファイルを読み出しながら再生する
		理由	動画像データの受信間隔を調整する
□□□	要求	VIS.1.4	再生中に一時停止の操作ができる
		理由	
		この他、いろいろな操作要求を扱う(省略)	
□□□	要求	VIS.1.10	動画データが終了した時、または強制終了したときは一時保存データを削除する
		理由	著作権に抵触するため

7. 範囲の大きい要求を分割して階層化するには、何らかの分割基準が必要になる。要求を分割・階層化する際の基準が曖昧であれば、下位の要求の設定(表現)が不安定になり、分割された要求の間に隙間が生じる。「分割・階層化」としては、構造化設計における「モジュール分割」があり、そこには4つの基準が定義されている。

- 設定－入力－変換－出力分割
- トランザクション分割
- データ構造分割
- 共通分割

という基準を(状況によって組み合わせて)使って並べる。これらの基準によって、大きな範囲を持つ要求を適当な範囲に狭めることができ、範囲が適用に狭められたことで、そのなかに含まれるべき仕様の抽出もスムースになる。

モジュール(関数)を分割階層化する際には、これらの分割基準を組み合わせることですべての状況に対応できる。実際にどのような基準で関数を分割したのかが分からなければ、気になる処理がどの関数で対応されているのか探す際に苦労する。

そこで、要求の分割・階層化にも、これと同じような基準を導入し、分割の際に注意しなければならないことは、隙間が所持無いことである。

- 時系列分割(時間軸分割)
  時間軸に着目して、要求を分割する。

要求:選択されたグループの送信と受信の電子メールの中からキーワードで検索して内容を見ることができる

階層要求1:メールボックスのグループから1つを選択できる
階層要求2:いくつかのキーワードを組み合わせて検索できる
階層要求3:検索結果を扱いやすく表示する
階層要求4:表示されたメールを選択し、内容を見ることができる

- 構成分割
  要求に含まれる「機能」や「構成」で分割する方法。時間的な順序性を持たない。コマンド別に要求を分割したり、メッセージの種類で分割する。

要求:PC からの印刷要求に対応する

階層要求1:Status コマンドでは印刷の状態を PC に知らせる
階層要求2:Print コマンドでは、そこまでに届けらている印刷データを印刷する
階層要求3:Frame コマンドでは、ここで指定されている印刷領域に合わせて以後の編集を行う

- 状態分割

「状態」は、ある視点で見ると「時系列」の要素を持っていることがあり、ある場面では「構成」の要素を持っていることがある。

たとえば、次の要求の場合、時系列的な観点からみると、セッションの確立から通信装置、セッションのクローズという分割が考えられる。セッション確立後には、端末の選定やデータ収集要求などのコマンドによる処理が考えられるので、構成分割が適用可能となる。

要求:通信装置で結ばれたホストと複数の端末の間を仲介してホストへのデータ収集を支援する

このようなケースは、画面遷移など状態が存在している場合に対応しやすい。

＜端末未確定状態＞
階層要求1:ホストからの端末問い合わせメッセージに対して、接続されている端末の情報を返す。
階層要求2:ホストから確認済みメッセージを受けて、選択された端末を接続中の状態にする
＜通信待ち状態＞
階層要求3:ホストから送信指示が送られてきたときは、端末に引き渡し、受信の準備に入る。
階層要求4:端末からホストに送信するデータが届いたときは、形式を統一してホストに送信する。
＜通信終了待ち状態＞
階層要求5:ホストからの接続断の指示が届いたときは、端末に通知し、受信終了を確認して採集データとしてホストに引き渡す。
階層要求6:端末からの終了を確認してホストとの接続を切る。

このほか、＜通信エラー状態＞や＜再接続待ち状態＞といった「状態」も考えられる。

- 共通分割

上位要求を分割階層化したとき、異なる上位要求の下位要求に同じ要求が出現する場合で、これを共通機能として切り出して別に扱う。参照先を指定しておく。

階層要求 k:カードまたは通帳から口座番号を読み取る。(詳細は「共通機能」の要求 CRD01 を参照)

11.16 分割基準の共有

1. 要求を階層化範囲を持たせ、さらに理由をつけるのは、関係者の間で要求仕様に対する認識を共有することにある。そうすることで、仕様の記述方法や記述箇所、要求の漏れを相互に確認することができる。また、せっかっく要求を分割しても分割の基準がわからない状態では、要求仕様のレビューははかどらない。

2. 要求仕様書について顧客を含めて関係者の間で合意するには、以下の事項が満たされていることが条件になる。

   - 文書の構成がわかりやすいこと
   - 要求が適切な範囲をもって表現されること
   - 要求につけられた理由が要求の目的より明確にしていること
   - 要求を分割・階層化した際の分割の基準を共有できること

3. また、要求の階層化は、要求の範囲を狭めるための重要なテクニックであるが、階層の深さは2層程度にとどめておくこと。深くなると煩雑になり、階層化のメリットが薄れる。一般に要求の階層にムラが生じるのは、要求の範囲にムラがあることが原因であるため注意をすること。

11.17 テストケース仕様とつなぐ

1. 設計者とは別に明確に評価部隊が確保されているときは、要求仕様書のベースライン設定を受けて、そこから評価部隊によってテストケース仕様書が作られることになる。基本的には、要求仕様の1つの項目が実現していることを確認するのに、いくつものテストケースが用意される。それだけに、要求仕様書のリリースが遅れるとテストケースの準備に支障をきたす。また、テスト項目であふれてしまわないために、関係する複数の要求仕様をまとめてテストする方法を考えることも重要である。

2. 個々の要求仕様の項目に対して、必要かつ有効なテストケースが用意されていることを検証する必要があり、そのために要求仕様の番号をテストケースの番号と交換し、テストケースに漏れがないことを確認する。双方で番号を交換できればよいが、それが実現しない場合は、次のようにテストケース仕様書側に、そこでテストされる要求仕様の項目番号を埋めることが考えられる。

＜要求仕様書＞

要求	FND07	検索結果を扱いやすく表示、そこから選択したい
	理由	目的のメールが一つとは限らないので、絞りこめるような操作をしたい
□□■	FND07-01	検索されたメールの件数を一覧の上に表示する
□□■	FND07-02	該当するメールが存在しないときは「該当なし」を表示する
□□□	FND07-03	検索されたメールの「Subject」を一覧で見せる

＜テスト仕様書＞

テスト番号	テスト内容	関連要求仕様番号
T-FND10.05		FND07-01 FND07-02
T-FND10.06		
T-FND10.07		FND0701

要求仕様の「チェックボックス」の 3 つ目が“■”となっていることで、テスト仕様が用意されていることを確認したことを示している。

11.18 仕様のグループ化

1. 要求を分割・階層化したとしても、要求によってはその中に含まれる仕様の数が多くなることがある。要求の粒度のバラツキから、仕様が 10 数個～数 10 個になるような要求も出てくる。1つの要求のなかにリストされる仕様が多くなると、必然的にいろいろな場面についての仕様が混ざり、焦点がぼやけてしまう。たとえば、

- 設計などのその機能の動作条件についての仕様
- データを計算する際の種類についての仕様
- 計算結果の表示方法についての仕様
- 計算結果の保存についての仕様

などの仕様がリストアップされ、それらが何の規律もなくま混在すると要求の範囲を狭めたメリットが薄れる。このような場合、仕様として触れていることの対象の違いなどからグループに分け、さらに集合を小さくし、できるだけ混じり気のない仕様のグループを作ることが重要である。

仕様が＜＞記号でグループ化されている様子を示した例

要求	FND07	検索結果と検索された電子メールのリストを表示し、そこから目的のメールを選択する
	理由	目的のメールが一つとは限らないので、内容を確認する必要がある。
	説明	
	＜検索件数の表示＞	
□	FND07-1	検索されたメールの件数を一覧の上に表示する
	＜検索メールの表示＞	
□	FND07-5	1 件以上発見されたときは、メールの「Subject」を一覧で見せる
□	FND07-6	検索されたメールに連続番号を付けて表示する
□	FND07-7	検索されたメールの件数が 10 件を超えるときはスクロールバーを見せる
	＜メールの中身の表示＞	
□	FND07-10	一覧から1つのメールを選んで、その内容を見ることができる
□	FND07-11	開示されたメールの中にある検索キーワードと一致している文字列を赤色で表示する

「グループ化」は、グループをあらわす記号(＜　＞)を使って適当なグループ名を挟む。抽出された仕様がいくつかの対象を扱っていると判断されるときは、対象をグループで表現し、抽出された仕様をそれぞれのグループの中にまとめる。グループの大きさは、「7±2」程度とする。

11.19 シナリオ機能と単純機能の違い

1. 機能要求を仕様化しているなかで、同じ仕様が何度も出てくることがある。機能要求には「単独の機能要求」と「シナリオ的機能要求」の2種類があり、要求が「シナリオ」の性質をもつ場合は、異なるシナリオ要求のなかに同じ「機能の仕様」が出現することがある。

2. たとえば、銀行の「ATM」では、「入金」、「出金」「送金」といった「機能」の要求を階層化(または仕様化)していくなかでこの状態に遭遇する。次の例は要求の階層化を使って「入金機能」の要求を階層化したものである。

＜シナリオ機能(入金機能)を階層化した例＞

要求	ATM.1	カードまたは通帳から確認した口座に対して、任意の現金の入金を受付け、結果をレシートまたは通帳に印刷する	
	理由		
	説明		
□□□	要求	ATM.1.1	カードまたは通帳を読み込んで口座番号を取得する
		理由	
□□□	要求	ATM.1.2	取得した口座番号が適正な口座であることを確認する
		理由	
□□□	要求	ATM.1.3	適正な口座番号が適正な口座であることを確認する
		理由	
□□□	要求	ATM.1.4	入金された金額を口座に加算する
		理由	
□□□	要求	ATM.1.5	処理結果を通帳またはレシートに印刷する 通帳に印刷するときは未記帳情報も一緒に記録する
		理由	

このとき、「ATM.1.1 カードまたは通帳から口座番号を読み取る」や「ATM.1.5 処理結果を通帳またはレシートに印刷する。通帳に印刷するときは未記帳情報も一緒に記帳する」という要求は、他の「出金機能」や「送金機能」の階層化のところでも出てくる。したがって、それぞれ階層化要求のところで仕様に展開した場合は、間違いなく仕様が重複する。かといって「入金機能」のところで仕様に展開して、そのほかの箇所では「入金機能の項を参照」とするのも保守性が悪くなる。

シナリオ要求の場合は一般に、このような「機能の重複出現」の問題がつきまとう。この場合は「共通分割」によって、

要求: KOZ02 カードまたは通帳から口座番号を読み取る

という形で「純粋な機能要求」として独立させ、そこで、

- カードまたは通帳を選択する方法
- カードから口座番号を読み取る方法や読む際の制限
- 通帳から口座番号を読み取る方法や、通帳であることに起因する読む際の制限

- 本人確認の方法
- カードや通帳からの読み取り時のエラー処理

について仕様化し、入金機能や出金機能などの要求からは「参照」する形とするのがよい。

11.20 ユースケースにおける問題点

1. オブジェクト指向によるソフトウェア開発のなかで、要求の表現に代わるものとして「ユースケース」が使われることが多いが、ユースケースは要求に対応させることができるが、そこでは、「シナリオ」的な要求を扱うことが少なくない。「シナリオ」の特徴はフローチャートになりやすいことである。そのためにシナリオで表現したままでは必要な仕様を見落とす可能性が高くなる。

2. ただ、UML では、シナリオはクラス間シーケンスと対応させることで、シナリオの基本的なシーケンスを検証する目的がある。最近ではシナリオに代わって「ユースケース記述」を使う方法があるが、この場合は確かに従来の記述方法よりも「仕様」を表現しやすくなると思われる。

3. ただし、ユースケースから「仕様」を引き出しやすくするために、ユースケースの粒度が小さくなり、いわば単一の動詞を扱う傾向があるため、その場合は必要な「動詞」がモレる可能性がある。

4. そこで、この問題を解決する方法として、ユースケースを「上位要求」と捉え、シナリオのレベルを「階層化した要求」として表現し、具体的な仕様は階層化した要求の下に展開する方法がある。

5. 図表6－1、6－2はユースケースとみなすことができる。このとき、「SEKI01.1」から「SEKI01,5」はシナリオに相当する。そして仕様は、階層化された要求(「SEKI01.1」～「SEKI01.5」)の下に展開する。Actor とユースケースを使うことで、要求の存在に気づきやすくしてくれる。「モデル」の効果である。その効果を活用して要求を抽出し、シナリオに展開しさらに仕様に展開する。

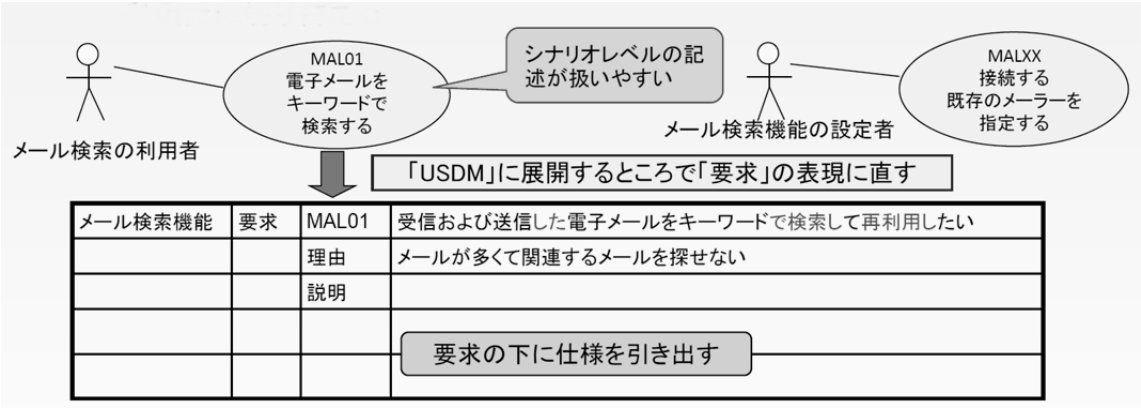
＜図表6－1　上位ユースケースに見立てた例(1)＞

要求	SEKI01	操作画面から指定された講演の座席を予約し、代金を引き落として予約番号を表示する
	理由	
要求	SEKI01.1	会員番号が入力されることで講演名が表示された開始画面を表示する
	理由	
要求	SEKI01.2	開始画面から講演名を1つ選択することで、座席の予約状況を知ることができる

	理由	
要求	SEKI01.3	講演の座席の予約図を見ながら空いている指定席の座席を予約できる
	理由	
要求	SEKI01.4	予約が成立したときは、料金は事前に登録されているクレジットの口座から引き落とす
	理由	
要求	SEKI01.5	予約番号とバーコードを含んだ予約結果を画面に表示し印刷を勧める
	理由	
	説明	表示された予約結果は予約者の方で印刷して会場に持参する

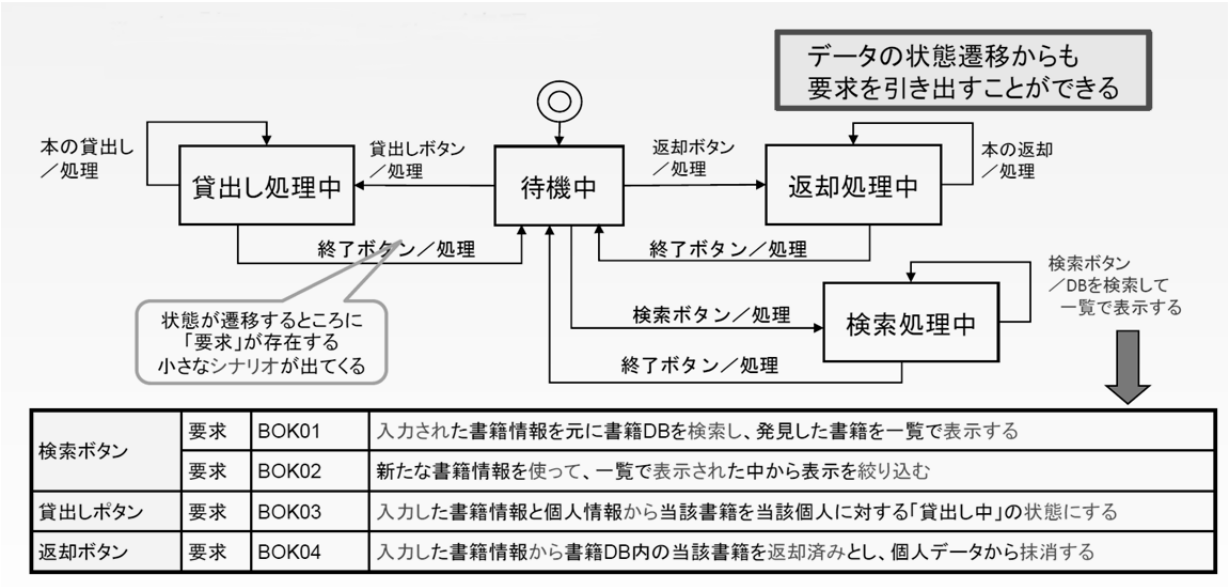
＜図表6－2　上位ユースケースに見立てた例(2)＞

要求	Lib10	新規登録に必要な情報を入力して書籍 DB を更新し、書籍に貼る管理用シールを印刷する	
	理由	管理に必要な情報を DB に登録する必要がある	
	要求	Lib10.1	新規登録画面の中で新規に登録する書籍の ISDN 番号または書籍名を入力し、既に登録されているかどうか書籍 DB を検索する
		理由	追加の購入に対して同じ情報を入力するムダを省くため
	要求	Lib10.2	新規登録画面の入力ボックスに書籍属性を入力し、登録する書籍冊数を入力する
		理由	書籍検索に耐えるための情報になる
	要求	Lib10.3	「登録」ボタン押下で、冊数分だけ書籍情報を作成して書籍 DB に登録し、管理用シールを冊数分印刷する。既存書籍の場合は冊数が更新される
		理由	これで「在庫」状態になる。管理シールはこのあと書籍に貼る



11.21 状態遷移図から導出する

1. 状態遷移図がある場合、遷移する際に「振る舞い」を持つので、「要求」を引き出しやすい。このとき、状態遷移のきっかけとなる“イベント”とそのイベントに付随する“振る舞い”があり、振る舞いを漏れなく表現することで要求を抽出できる。

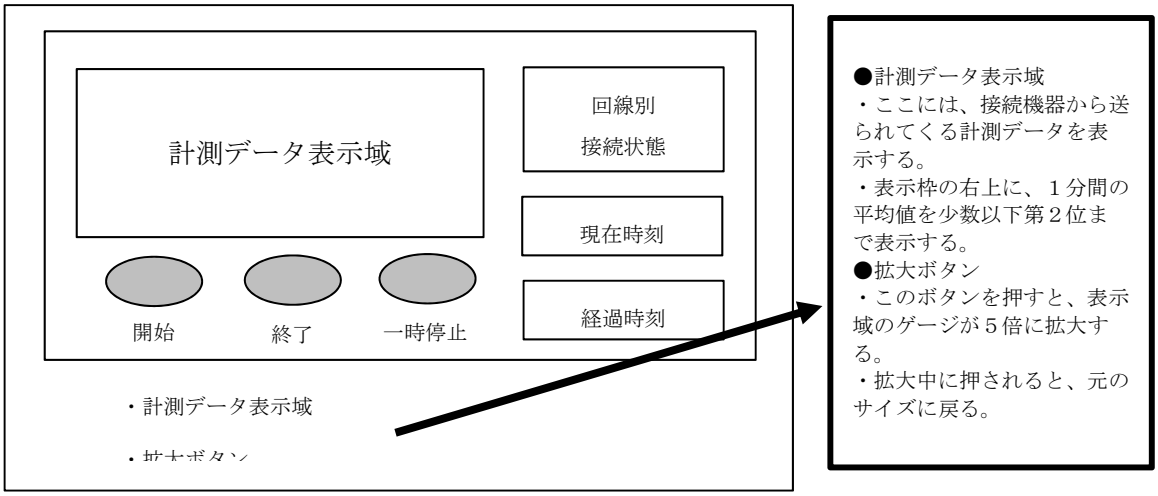


検索ボタン	要求	BOK01	入力された書籍情報を元に書籍DBを検索し、発見した書籍を一覧で表示する
	要求	BOK02	新たな書籍情報を使って、一覧で表示された中から表示を絞り込む
貸出しボタン	要求	BOK03	入力した書籍情報と個人情報から当該書籍を当該個人に対する「貸出し中」の状態にする
返却ボタン	要求	BOK04	入力した書籍情報から書籍DB内の当該書籍を返却済みとし、個人データから抹消する

11.22 画面仕様のあり方

1. 一般に操作画面や表示画面の仕様を表した「画面仕様書」は、画面としてレイアウト設計された図に対応させて、画面レイアウトの下に表示領域やボタンなどの仕様が書かれることが多い。しかしながら、表示領域やボタンなどに対する仕様は、以下のように項目の説明に過ぎないケースがほとんどである。これでは、仕様として必要な項目の多くがモレてしまう。

＜一般の画面仕様のイメージ＞



・計測データ表示領域 計測したデータを、項目ごとに1桁のスペースを入れて、レコード単位で表示する ・一時停止ボタン このボタンを押すと、表示を一時停止する

これでは、「計測データ表示領域」は計測で入手したデータを表示する場所、ということしか分からず、単純に入手したデータを表示するようなプログラムになる。

- 表示を開始する条件は何か？
- この画面を開いたときは、すでに計測データは収集されているのか？
- 何も表示するデータが届いていないときはどう表示するのか？
- 表示域一杯に表示したあとは、スクロール機能を持たせるのか？
- スクロールで後戻り操作されているときに表示データが届いたときはどうするのか？

2. このような場合でも、次のように、通常の要求と要求仕様の形で記述することで、最初からかなりのレベルまで仕様モレを防ぐことができる。

要求	DSA.1	計測したデータと区間平均値をリアルタイムに表示しながら異常を監視する
	理由	接続先の状況に素早く対応する必要があるため

	説明	
	＜初期表示＞	
□	DSA.1.1	計測を開始するまでは、画面の中央に“NONE”と表示する
	＜計測データの表示＞	
□	DSA.1.4	計測したデータは、1レコード1行に表示する
□	DSA.1.5	項目の協会に1桁のスペースを挿入する
□	DSA.1.6	30行を超えた時点で右側にスクロールバーを表示する
	＜平均値の計算＞	
□	DSA.1.10	計測データを20件取得した時点で、項目#3～7 の平均値を算出する
□	DSA.1.11	平均値データは、その後10分おきに20件のデータを加算して算出し直す
	＜警告表示＞	
□	DSA.1.15	項目#3～7 の項目で、それまでの平均値と 10%以上の差が生じたときは赤字で表示する
□	DSA.1.16	警告表示は、平均値が算出されてから実施され、その後継続される

画面上のボタン類もこれと同じように記述することができる。たとえば「一時停止ボタン」の場合、

要求： 計測されたデータの表示を一時停止し、画面の更新を止める 理由： 表示されたデータのなかで違和感を感じたとき、オペレータが詳しく見極めることができるように

この下にボタンを押せるタイミングや押した時の動き、表示再開の方法などの仕様が記述される。たとえば、次のように仕様が追加されるかもしれない。

＜再開時の表示方法＞ □ 表示停止中に収集されたデータは、青字で表示すること □ この色は、スクロール操作をされても消えないこと

計測データをまとめて保存する機能では、

□ 表示停止／再開の操作が行われたことを知る □ 収集したデータに表示停止中に収集したことを記録する

3. 一般に画面仕様書は、仕様としては稚拙なレベルのものが多い。これは「プロトタイピング」や GUI

構築ツールを使い、顧客に見せながら作りあげていけばよい、という考え方によるものかもしれない。画面かは外から見ることができ、このような構築の仕方によって簡単に作れるように思えるかもしれないが、目の前の動くものを見せられることで考える範囲もそこに誘導され、本当に希望することを考えられなくなることもある。

4. そのうえ試作に依存して仕様をまとめると、複数の画面にまたがる機能間で仕様上の矛盾(衝突)を引き起こしやすい。依頼者は一般にユーザーであり、表面的な操作や使い勝手しか見えないため要求できない。実際にその操作によって内部でどのような処理が行われるかを考えられる人でなければ、表示を一時停止しているときに計測データを、通常データと区別しておくことの必要性に気が付きません。

11.23 画面操作全体に対する要求

1. 画面仕様書で共通していえることは、最初の画面要求がほとんどまとめられていないことである。いきなり GUI 構築ツールを使い個々の「画面レイアウト設計」に入るため、画面操作全体に対する「要求」を確認するタイミングが失われている。あるいは、そのようなレベルでの「画面要求」の存在そのものに気づいていないことがある。

望ましい手順として、

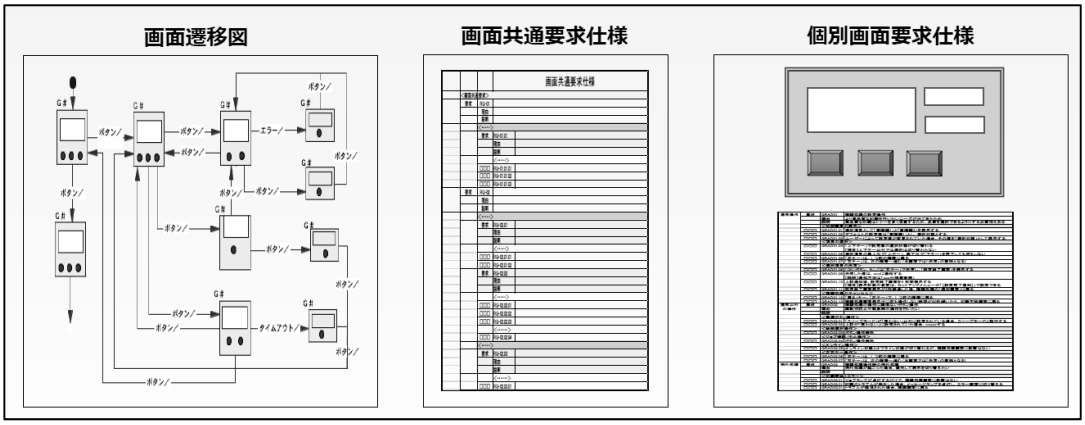
- 全体の画面要求
- 全体の画面仕様
- 個別画面設計(画面遷移図、個々の画面レイアウト)
- 個別画面仕様(画面ごとの表示上の制限、表示領域に関する仕様、個々の操作における仕様)

＜画面全体要求の事例＞

要求	#1	ファイルと接続に関する設定項目は、すべて画面上で行えること
要求	#2	セキュリティの設定ができること
要求	#3	セキュリティのレベルに応じて、見える画面、操作できる画面やボタンを設定できる
要求	#4	どの操作中でも、ベースの画面に直接戻って、そこから新しい画面操作が

		できること
要求	#5	接続されている機器の計測データはすべてリアルタイムで表示し、異常が判断されるときはオペレータに適切に注意を喚起すること
要求	#6	操作エラーを含めて、エラー表示は、エラーのグループ(種類)がわかるようにすること
要求	#7	エラー表示に対する操作は、簡単にヘルプで確認できること
要求	#8	見えにくい文字のサイズや色を使わないこと(品質要求)
要求	#9	マウスによる選択では、操作ミスを起こしにくくすること(品質要求)

2. 画面に関する仕様には、次のものがある
   - 画面遷移図:システム全体の画面構成を示す(画面全体を見渡せること)
   - 画面共通要求仕様:各画面に共通な個別画面の要求の方向性(思想)を示す
   - 個別画面要求仕様:個々の画面の構成要素(画面要素)ごとの要求と仕様を表す



11.24 個別の画面設計の指針

1. 操作画面全体に対する要求は、次のように仕様に展開することができる。これらの操作画面全体に対する要求および仕様は、要求によっては、操作画面に対する「統一思想」の意味をもつ。

＜画面要求を仕様化した例＞

要求	OP4	どの操作中でも、ベースの画面に直接戻って、そこから新しい画面操作ができること	
	理由		
□	仕様	OP4.1	操作画面が重なったときでも、ベース画面は隠れいないようにする

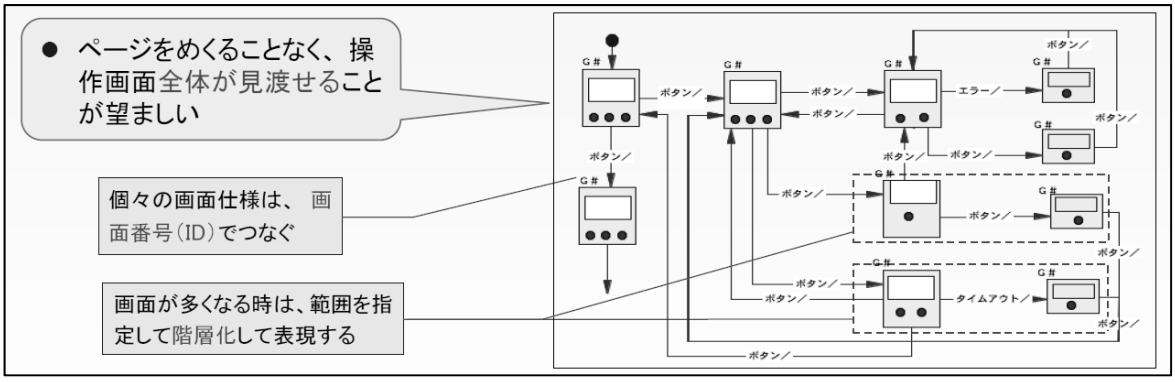
□	仕様	P04.2	ベース画面に直接戻ったときはそれまでの操作は無効になるので、入力操作が行われたときはその場で確認を求めること
要求	OP5	接続されている機器の計測データはすべてリアルタイムで表示し、異常が判断されるときはオペレータに適切な注意を換気すること	
	理由		
□	仕様	OP5.1	“異常”の判断には 3 段階程度の基準を設ける
□	仕様	OP5.2	異常を知らせているときは、画面の更新を停止させる
□	仕様	OP5.3	画面の更新を停止中も計測に影響を与えないこと
要求	OP7	エラー表示に対する操作は、簡単にヘルプで確認できること	
	理由		
□	仕様	OP7.1	「ヘルプ」を選択したときに、現在表示しているエラーコードに対する説明が出ること
□	仕様	OP7.2	エラーのヘルプでは、エラーの意味と、選択できる操作とその意味がわかること

11.25 画面遷移と画面仕様の連携

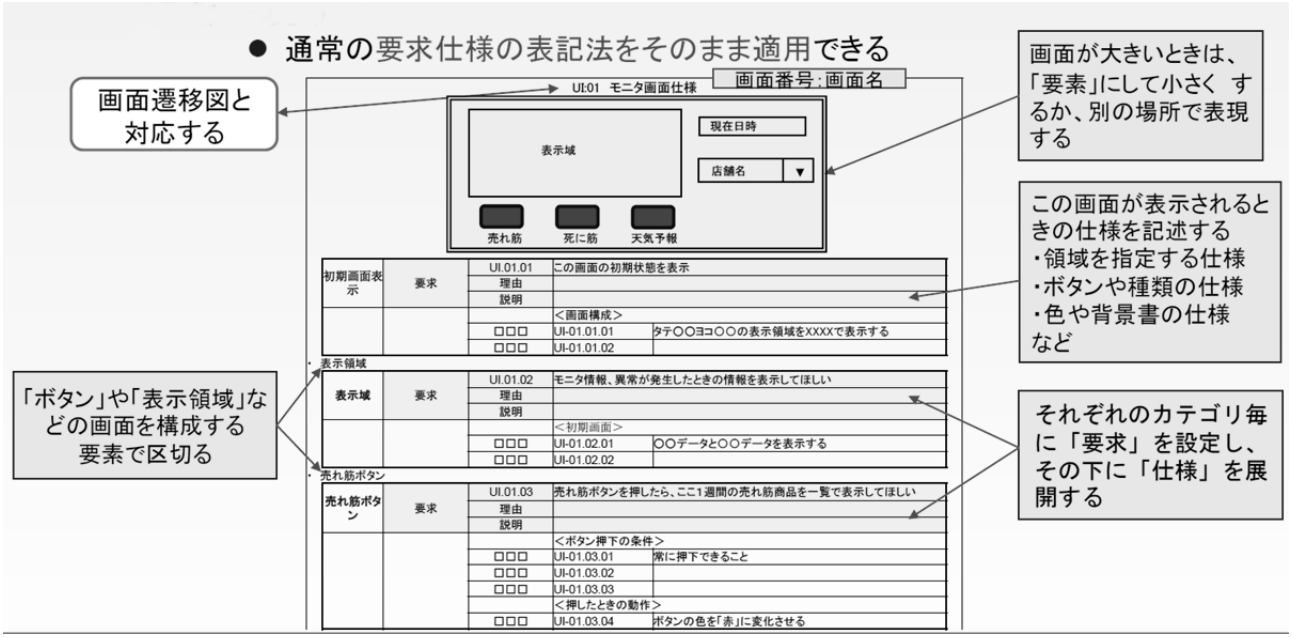
1. 読みやすい文書の一般的な構成は、最初に全体を示し、そのあとに個別のテーマに入る仕掛けが用意されていることである。画面仕様書についても同様で、画面全体の様子を一望できる方法を提供し、その下に個別の画面レイアウトを示し、さらに個別の画面の構成に対する仕様を展開するといった形が望ましい。

2. 遷移図は、それぞれの画面が持つ遷移の種類すべて表現することが重要である。代表的な操作だけを扱った遷移図は、遷移の様子を検証することができずほとんど役に立たない。そのような中途半端な遷移図を書くために投入した工数を、あとのプロセスで回収できない。

3. 画面遷移上の画面には、必ず「画面番号」を付け、個別の画面仕様書とつなぐことが必要になる。「画面の名称」の場合は、必ずしも一致しないことがある。番号も単純に連番にすると画面の追加によって崩れてしまうため、「画面グループ」を設定してグループ番号を使うなど、グループを表す情報をもたせることで、個々の画面を見ただけでどのような状況で使われる画面かがすぐ分かるメリットが出せる。



4. 画面仕様を上手く表現するには、個別画面の配置図のページに、画面の構成要素ごとに要求と仕様を表現する。



初期画面表示	要求	UI.01.01	この画面の初期状態を表示
	理由		
	説明		
	＜画面構成＞		
	□	UI.01.01.01	タテ○○ヨコ○○の表示領域を××で表示する
	□	UI.01.01.02	

・表示領域

表示域	要求	UI.01.02	モニタ情報。異常が発生したときの情報を表示してほしい
	理由		
	説明		

	＜初期画面＞		
	□	UI.01.02.01	○○データと○○データを表示する
	□	UI.01.02.02	

・売れ筋ボタン

売れ筋ボタン	要求	UI.01.03	売れ筋ボタンを押したら、ここ①週間の売れ筋商品を一覧で表示してほしい
	理由		
	説明		
	＜ボタン押下の条件＞		
	□	UI.01.03.01	常に押下できること
	□	UI.01.03.02	
	＜押したときの動作＞		
		UI.01.03.03	ボタンの色を「赤」に変化させる

12付録

12.1 在宅介護サービス支援システム　仕様書　（付１）

在宅介護サービス支援システム
企業規模　100 人～200 人

背景

訪問介護事業を運営する株式会社 AP 社は、介護現場で課題となっている日々の申し送り効率化の課題を抱えている。訪問介護を担当する介護ヘルパー（20～70 歳代までの幅広い年齢層が登録されている）は登録制の直行直帰型勤務が多く、事務所への出社は週に 1 度程度である。1 人の要介護者を複数のヘルパーが入れ替わり担当する介護の現場では、日々の申し送りに大変苦労している。

このたび、A 社の課題を解決するソリューション提案を求められ、当社の技術部エンジニアが現場担当者とヒアリングを行った。ヒアリング結果を以下にまとめる。

ヒアリング事項

- 訪問介護のサービス内容は、「生活介護」と「身体介護」の2種類がある。介護者の担当ケアマネージャーの訪問介護計画書に従って介護サービスを提供する。
- 「生活介護」は日常生活を送る上で必要不可欠な家事について、利用者本人がひとりではできない部分を支援する。例えば、掃除、選択、調理、買物、衣服の整理などにあたる。
- 「身体介護」は、日常生活において行う＜動作介護＞、＜身辺介護＞、＜生活介護＞を行う。例えば、起床・就寝介助、水分補給、歩行、更衣、入浴、洗面、排せつ介助、食事介助などあたる。
- 訪問介護にあたるのは、ホームヘルパーであり、常勤ヘルパー（正社員）およびパートタイマー・アルバイトが従事している。最も多い勤務形態は、直行直帰型のパートタイマー・アルバイトである。ホームヘルパーは、サービス提供責任者の介護計画に従って、自宅から利用者宅まで直行直帰で働く。勤務時間は8:00～18:00までの間（最大20:00まで対応可能）で、シフト制による。介護者の担当は2～3名がシフトの交代制によって担当しているが、全員が担当できない場合は、他の担当者が担当することがある。また、複数の担当者が1件の対応をすることがある。
- ヘルパーが現場に直行直帰するには、基本的には自転車やバイクを使用する。遠方になる場合は、事務所へ出社し、事務所から車で移動する（事務所の運転代行者による送迎）。
- 1回あたりの介護時間は1時間が最も多く、1日に平均して3件の介護を行う。介護者1人に対して、ヘルパーが介護にあたる週1回必ず事務所に出勤し、サービス記録票のチェックおよび申し送り事項報告と勤怠入力を行う。サービス記録表は、複数の担当者によって作成されるため、最新の情報を共有できな

い場合もある。介護にあたる前日に、事務所スタッフから担当ヘルパーに電話による申し送りをする。この申し送りでは、直近の事項のみが伝えられるため、過去の対応状況などを把握することができない。

- サービス提供責任者は、携帯電話を24時間オープンにしておく必要があり、緊急時には深夜においても電話対応に追われることがある。これについては、システム化する必要はない。
- 要介護者の体調や薬、食事の内容、量など内容は命に関わる事柄も含まれ、連絡の抜け漏れは許されない。それほど重要な業務にも関わらず、現在の運用は、その日の介護を終えた担当ヘルパーからの電話報告を事務所スタッフがノートにメモし、次の担当ヘルパーへ電話等で伝えるという方法を取っている。
- ヘルパーの業務内容において、行政監査の際にて提出すべきものは、介護日報と、申し送り事項の内容であり、介護者ごとに対して、これらの内容を提出する必要がある。

現状の課題（まとめ）

- 1人の要介護者に対し複数のヘルパーが交替で介護担当をすることが多い介護現場で、日々の申し送りを抜け漏れなく行うことに大変な労力を費やしている。
- 事務所スタッフが電話報告を受け、ノートにメモを取って次のヘルパーに申し送りをしているが、過去の履歴データを確認できず不便である。
- 介護記録は手書きで記録管理しており、数年に一度の行政監査の際には、スタッフが介護記録を整理し、ファイリングして提出している。

システム要件

事務所スタッフがヘルパーへ申し送りをする方法は、一斉送信用画面から、要介護者名、その介護に関わるヘルパー名（複数選択可）を選択し、申し送り内容を本文に記して、ヘルパー宛へメールの一斉配信を行う。メールを受信したヘルパーは、受信したメール本文から申し送り内容を確認後、同メールに記載されたリンク先をタッチして、ヘルパーメールシステムにログインし、「メールを確認しました」ボタンをタッチする。この作業でヘルパーメールサーバー上に既読情報が記録される。

ヘルパーから事務所スタッフへの申し送りは電話報告の運用を続ける。これは、ヘルパーからの報告をシステム化すると、携帯からの文字打ちが必要となり、操作に時間も要し、メールが苦手なヘルパーからの報告が抜け漏れするリスクがあるためである。

そのため、介護終了時のヘルパーからの申し送り内容は、事務所スタッフが電話を受けてシステムに入力し、情報を蓄積する。システムから次の担当ヘルパーへ一斉送信すれば申し送りが完了する。

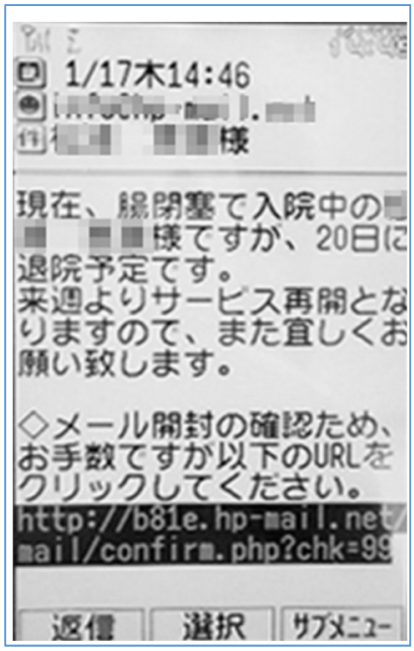
システム上では、ヘルパーの未読・既読一覧状況がすぐに把握できるほか、日々報告される申し送り情報を介護記録として管理できる。介護記録は、数年に一度の行政監査の際に使用する。過去の履歴を CSV 形式で書き出すことも可能である。

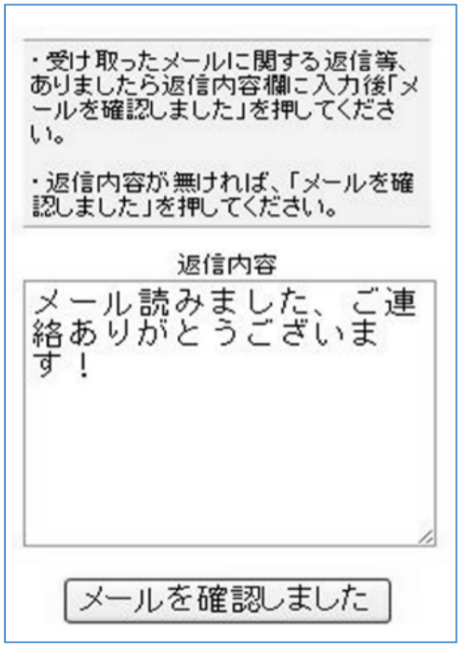
非機能要件

ユニークユーザー数	・介護施設の全職員:200 人(うち 100 人が非正規職員) ・携帯電話やスマートフォンを使いこなせない高齢職員数:50 人 ・アクティブユーザー数:200 人－50 人＝150 人がアプリを常用
被介護者数	・1,000 人 ・介護状況の報告者は、正規職員からおよそ 100 名が担当する。
システムピーク時間	・食事介助の後の報告となる 13 時から 13 時半、17 時半から 18 時の間に報告のピークが発生し、30 分間で 100 件程度。 ・報告を受けた結果の通知のピークが、報告の後の 30 分間で、報告 1 件に対して担当ヘルパー10 人に連絡するため、30 分間で 100 件×10 人＝1,000 件の通知が想定される。その 30 分後に返信のピークが同数程度発生するものとする。
サーバ仕様	・1 件当たりの通信データ量を想定して、トラフィック制限のあるサーバプランから、レンタルサーバ業者を選定する。可用性、信頼性についても同様に検討する。 ・耐用年数は 5 年間とする。

画面イメージ(案)

事務所スタッフが担当ヘルパーへ申し送りに使用する「一斉送信確認画面(PC)」。申し送りの顧客、報告したいヘルパーを選択し、連絡事項を記入して送信すれば、申し送りは完了する。





ヘルパー宛に配信される申し送り連絡メール（スマホメールもしくはケータイメールで確認）。ヘルパー連絡事項を確認したら、画面下方のリンク先をタッチするとメモ欄とメール確認ボタン画面に移行する。「メールを確認しました」ボタンを押せば既読連絡が完了する。

ヘルパーメール管理者画面。未読／既読の情報が一覧で確認できる。

12.2 USDM 表記法フォーマット

					優先度	備考欄
カテゴリ名 (記号)	要求	(要求番号)	ここに要求を記述する。			
		理由	要求の背景や理由について記述する			
		説明	要求について必要に応じて説明する。仕様とは見なされない。セルを広げて図を貼り付けることも可能。			
	要求	(要求番号)	ここに階層化された要求を記述する。要求番号は一段下げられる。			
		理由	範囲を狭めた要求について、背景や理由を記述する。			
		説明	要求について必要に応じて説明する。仕様とは見なされない。			
	<<仕様分類名>>		主分割記号・・・全体を通して共通の分割基準として決める。			
		<<仕様分類名>>	補助分割名・・・主分割の中に異なるテーマの仕様が混在する場合、補助分割記号を使って純度の高い集合を作る。			
		□ (仕様番号)	ここに仕様を記述する。			
		□ (仕様番号)				
		<<仕様分類名>>				
		□ (仕様番号)				
		□ (仕様番号)				
		□ (仕様番号)				
		□ (仕様番号)				
		理由	仕様について、必要に応じて背景や理由を記述する。			

12.3 引用・参考文献

参考図書:[入門＋実践]要求を仕様化する技術 表現する技術 清水吉男 技術評論社

参考図書:要求仕様定義ガイドライン JUAS・UVC 報告書 社団法人日本情報システム・ユーザー協会

参考図書:共通フレーム 2013 独立行政法人 情報処理推進機構(IPA)

13アーキテクチャ設計ーはじめに

13.1 はじめに

本稿の目的は、“ソフトウェア開発の成否を担うと言っても過言ではないアーキテクチャ設計について、理解を深め、実践の中で活かせるだけの知識を習得することで、読者の皆様のビジネス現場における市場価値をより高めること”である。

ソフトウェア開発の現場に身をおくと肌身を持って感じることではあるが、ソフトウェア開発の前線は驚くべき勢いで変化を続けている。注目を集めていた新技術が、たった数年のうちにほとんど使用されることがない廃れた技術と成り果てた例を数多く見てきた。また、技術の進歩と共に、ソフトウェア技術が適用される分野は拡大を続け、周りを見渡せば日常のあらゆるところでソフトウェアが利用されていることがわかる。このような時代ゆえ、エンジニアとして高い市場価値を保つためには、常に自己研鑽を続け、成長し続けることが求められる。少しの間でも学ぶことを止めてしまえば、ソフトウェア開発を取り巻く環境の変化について行くことが難しくなり、エンジニアとしての価値はあっという間に逓減していってしまうだろう。日々栄枯盛衰を続ける先端技術を追いかけ続けることは、さながら波乗りをするかのようで、新たな技術と言う波に、いかに上手く乗り続けられるか。そして、ただ乗るだけではなく、今後新たな潮流を巻き起こす大きくて良い波を常に見極め、その波に乗れるだけの基礎的な技術を身に着ける努力を続けていかなければならない。

こうした前置きは、現役エンジニア、そしてエンジニアを志す読者の皆様からすれば、耳にタコが出来る程繰り返し聞いてきた文言であろう。そして、この中で述べられる学ぶべき先端技術と言う言葉から連想されるのは、実装段階において使用されるプログラミング言語や、アプリケーション開発用フレームワークなど、下流工程における技術なのではないだろうか。確かに、下流工程の技術はその経験や知識を定量的に測りやすく、そのため注目もされやすい。近年の履歴書には、使用可能なプログラミング言語やフレームワークを書く欄が設けられているものもあり、その知識や経験によって面接に呼ばれるかどうかが決まってしまうことすらあるらしい。もし、現在の業務あるいは就職活動にてそれなりに良い成果を出すことが第一の目的なのであれば、そうした下流工程における技術の経験をそれらしく語れるようになることが近道かもしれない。しかし、本書の読者の皆様にはもう一歩進んで、実際の現場で重宝されるエンジニアとなることを目指して頂きたい。

では、どのようなエンジニアが現場において重宝されるのだろうか。筆者の乏しい経験に基づき論拠するのはいささか恐縮ではあるが、学生時代より複数の会社の立ち上げに携わり、また、外資系投資銀行と言う、世界中から優秀なエンジニアを集める会社での勤務を経験し、多くの市場価値が高い(≒必要とされる)エンジニアと仕事をともにしてきた中で、彼らに共通してみられた能力は、

“本質的に求められているものを理解し、ユーザーの期待以上のものを創り出す力”

だった。

新規開発案件においても、運用中のシステムにおける追加機能開発においても共通して言えることだが、ビジネスの現場では、毎日、驚くほどの多くの要求がユーザーから投げかけられる。ともすれば、与えられたプロジェクトを盲目的にこなしていくだけでも日々は過ぎていってしまう。しかし、それをこなすだけでは本質的に必要とされるエンジニアとなることはない。それどころか、近年、グローバル化による海外勢の進出や、デジタル・ネイティブ世代の台頭によりエンジニアのコモディティ化と言う言葉がささやかれるようになったが、与えられたプロジェクトをただただこなすエンジニアは、まさに取り替えのきく存在(=コモディティ)となってしまい兼ねない。コモディティ化の波を逃れ、現場において必要とされるエンジニアとなるためには、ただ言われたものを作るのではなく、数多くの要求の中から、ビジネスやシステムの現状を考慮した上で、与えられた要求の背景にある本質的に求められているものを理解し、自ら改善案を提案することで自分なりの価値を創出し、それを誤りなく実現する事が求められる。

これらを実現する上でソフトウェア開発工程の上流から下流まで一気通貫知識を持つ事が重要である事は間違いないが、自分なりの価値を創出する際に他者との差を生み出す上で特に重要なのは上流工程の理解の深さである。本稿では上流工程の中でも特に、プロジェクトの方向性を定め、成否を担うと言っても過言ではないほど重要な要素を占めるアーキテクチャ設計について論ずる。本書の読者の皆様には、前部の要件定義と合わせ、ソフトウェア開発工程の最上流のにおける知識体系を身に着け、演習を通し擬似的にソフトウェアプロジェクトを体感することで、“本質的に求められているものを理解し、ユーザーの期待以上のものを創り出す力”を身につける第一歩を踏み出して頂きたい。

13.2 本章の構成

本章では 2 節以降、以下の構成にてアーキテクチャ設計の基礎を論ずる。

2 章「アーキテクチャ設計とは」では、アーキテクチャ設計とはどういったもので、ソフトウェア開発工程の全体像においてどのような位置づけであるか、などアーキテクチャ設計に関する概説を行う。また、多岐に渡るアーキテクチャ設計の定義の中から代表的なものに触れ、アーキテクチャ設計の実際の工程がどのように構成され、何を成果物としているかについて理解を深める。最後に、アーキテクチャ設計の成果物として出来上がる”アーキテクチャ”を適切に表現する上で欠かせない構造とビューに関する概説を行う。

3 章「アーキテクチャと品質特性」では、アーキテクチャ設計を考慮する際に欠かす事が出来ない品質特性について学ぶ。はじめに、ソフトウェア工学のアプローチの中でも特に品質特性に関して体系的にまとめられている、ISO/IEC 9126-1:2001(JIS X 0129-1:2003)が提唱するソフトウェア品質モデルを概説し品質特性に関する全体間を醸成する。次に、よりアーキテクチャ設計と結びつきの深い CMU/SEI アーキテクチャ手法群におけるシステム品質特性について触れ、また、品質特性を表現する際に有用なシナリオと言う概念を学ぶ。最後に、各品質特性においてプロジェクト横断的に使用する事が出来る一般シナリオを紹介、また品質要件を満たす上でアーキテクチャ設計の段階で考慮されるべき実現手法を、品質特性毎に体系的に紹介する。

4 章 「アーキテクチャを設計する」では、2 章、3 章で学んだ、アーキテクチャ設計を実施する上で理解する必要がある基本概念を用い、実際にプロジェクト要件に対してアーキテクチャを設計する上での方法論を学ぶ。現代のソフトウェア開発でも用いられる事の多い代表的なアーキテクチャ設計手法を幾つか紹介した後に、その中の一つで、3 章で学んだ品質特性を中心に、アーキテクチャ設計を行う品質駆動設計について各工程を詳説する。また、アーキテクチャ設計手法の成果として出来上がった”アーキテクチャ”を各ステークホルダーが理解しやすい形で文書化する方法論について述べ、最後に完成したアーキテクチャを評価する代表的な手法を紹介する。

5 章 「演習」では、これまで学んだ知識に基づき、前章の演習の成果物から、品質駆動設計を用いその成果物である UML を生成する事を目標とする。

14アーキテクチャ設計の意味

14.1 アーキテクチャ設計とは

アーキテクチャ設計は、ソフトウェア工学の中でもまだ若い分野であり、広く受けられている単一の定義は存在しない。アーキテクチャ設計と言う言葉も、提唱する機関によって実に様々な解釈があり、担う責務もそれぞれ異なる。本節ではまず、広義の概念としてアーキテクチャ設計を捉えた後に、代表的な機関が提唱するアーキテクチャ設計の定義を論じ、様々な角度からアーキテクチャ設計に触れる事でその理解を深める。その後に、アーキテクチャ設計において扱われるべき重要な要素である品質特性を概説し、それら品質特性を満たすためのアーキテクチャの設計手法について触れる。

■アーキテクチャ設計とは

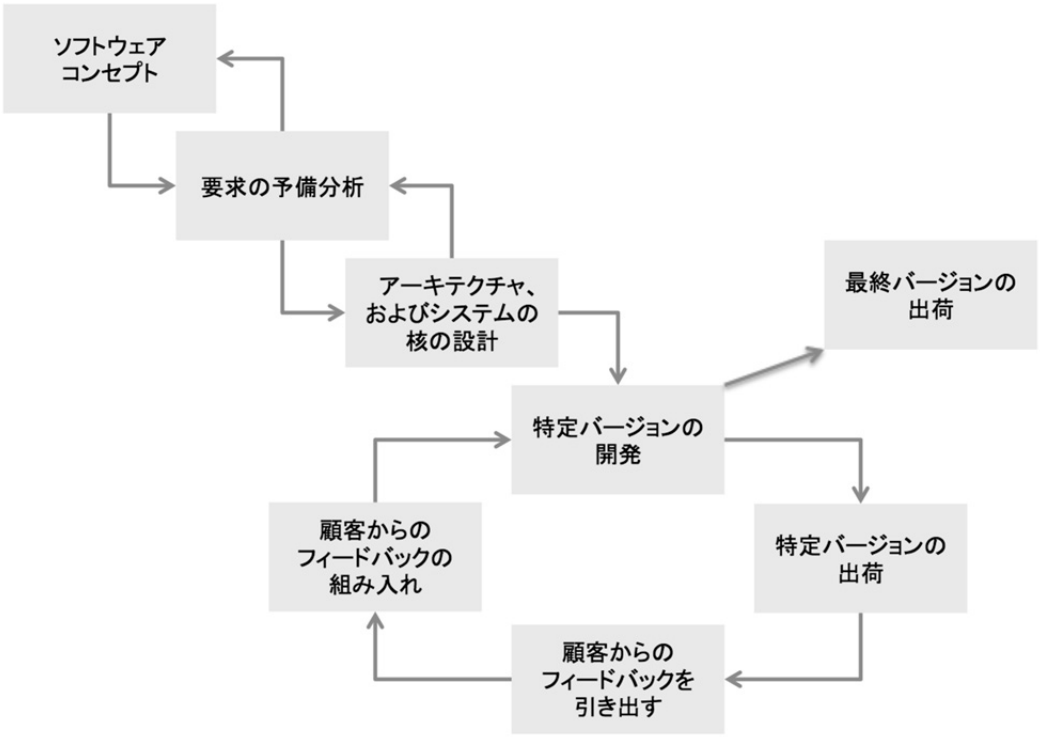
アーキテクチャ設計と聞いて何を思い浮かべるだろうか？ソフトウェア工学の慣例に習い、開発工程の例えとして頻出する建築に例えるなら、アーキテクチャ設計は、建主の要望を満たし、土地、予算、建築基準法など様々な制約条件を加味した上で、住宅の設計を検討し、間取りや、必要な材料を記した設計図として書き起こすまでの工程を言う。

ソフトウェア工学における定義では、要件定義の成果物である、機能要件、非機能要件などから成るプロジェクトのシステム要求仕様書を基に、プロジェクト上の制約や開発組織の状況などを考慮した上でシステムの動的/静的構成を設計し、システム構成要素の役割や振る舞いを明確化し、アーキテクチャ設計書としてそれらを書き起こすことを表す。要求や要求仕様書が何を(what)システム化するかを規定するのに対して、アーキテクチャ設計は、どのように(how)システム化するのかを規定する作業を意味する。成果物であるアーキテクチャ設計書を元に、以降の工程が進められて行くため、プロジェクトの成否を担うと言っても過言ではないほど重要な工程の一つである。

ソフトウェアのライフサイクルにおけるアーキテクチャの位置付けを理解するため、ライフサイクルモデルの全体像を表す図を以下に記す。ソフトウェア工学において、ソフトウェアのライフサイクルを表すモデルは複数存在するが、その中でも特にアーキテクチャを中心に据え置き、様々なソフトウェア開発のライフサイクルモデルの最善の組み合わせとも言われている、McConnel 提唱の Evolutional Delivery Life Cycle Model(発展的配布ライフサイクルモデルと訳される)を紹介する。

このモデルでは、ユーザーと顧客のフィードバックを得ながら、最終リリースの前に幾つかのリリースを繰り返す事を前提としており、「アーキテクチャ、およびシステムの核の設計」は、「要求の予備分析」が完了し、システム要求に関して何がしかの見解が得られ、アーキテクチャを方向づける要求である「アーキテクチャドライバ」が見つかった後に開始される。

アーキテクチャおよびシステムの核の設計が行われた後は、その成果物に基づき特定バージョンの開発が始まる。



アーキテクチャ設計は、ソフトウェア工学の中でもまだ若い分野であり、広く受け入れられている単一の定義が存在しない。アーキテクチャ設計と言う言葉も、提唱する機関によって実に様々な解釈があり、担う責務もそれぞれ異なる。一方で、一般的に普及した定義の多くはその主題においては一貫している。例えば、重要度の置き方において大小はあれ、多くのアーキテクチャ設計の定義では、以下の7つの事柄を明確にすることを目的としている。

1. ソフトウェアの全体構造
2. ソフトウェアで用いられるデータの構造
3. ソフトウェアと外部とのインタフェース
4. ソフトウェアの機能コンポーネントへの分割
5. 機能コンポーネント間の情報の連結や伝達
6. ソフトウェア全体構造に関わる性能属性

7. ソフトウェア全体構造に関わるセキュリティ属性

そして、これらの事柄を明確にすることで、以下にあげるような効果を期待する。

1. ソフトウェアの全体構造や、データ構造が明確化、そして言語化されることで、顧客、ユーザー、プロジェクト管理者、プログラマなど、ソフトウェア開発における利害関係者の共通言語となり、効率的なコミュニケーションを行うことが出来る。
2. 全体構造が明らかとなることで、早期の段階で必須項目の抜け漏れなどの問題を発見することが可能となり、実装段階になってからの手戻りなど大きな追加工数が必要となるような変更を避ける事が出来る。
3. 外部とのインタフェースを定義することで、ソフトウェアの担う責務の範囲が明らかとなり、過不足がない開発が可能となり、また外部システムとの接続を容易に行うことが出来る。
4. 機能コンポーネントの役割や構造、必要となる属性が明確化されることで、各機能を独立に開発することが可能となり、複数チームでの開発が容易になり、開発効率をあげることが出来る。
5. コンポーネント毎の必要機能や、性能属性、セキュリティ属性などが明確化され、曖昧さが排除されることで、開発者による解釈の違いにより発生する属人性を排除することが可能となる。

■システムアーキテクチャ設計とソフトウェアアーキテクチャ設計

アーキテクチャ設計と大きな概念で捉える場合もあるが、多くの場合、システムアーキテクチャ設計と呼ばれる求められるシステムの全体的な構成を扱うものと、ソフトウェアアーキテクチャ設計（アプリケーションアーキテクチャ、ソフトウェア方式設計など様々な呼称法が存在する）と呼ばれるソフトウェア部分の設計にフォーカスしたものは区別して扱われる。例えば前章でも触れた共通フレームでの分類が分かりやすいだろう。共通フレーム 2013 においては、システム方式設計（システムアーキテクチャ設計と同義）プロセスを、最上位の方式確立を目的としたものとし、システム要件のどれをシステム要素のどれに割り当てることが望ましいかを識別する。

システム方式設計を通し、次のような状態になる事を目指す。

1. システムの要素を識別子、定義された要件を満たすシステム方式設計が定義されている。
2. システムの機能及び非機能要件が取り扱われている。
3. 要件がシステムの要素に割り当てられている。
4. 各システム要素の内部及び外部のインタフェースが定義されている。
5. システム要件とシステム方式との間の検証が行われている。

6. システムの要素及びそれらのインタフェースに割り当てられた要件は、顧客の要件ベースラインへ追跡可能である。
7. システム要件とシステム方式設計との一貫性及び追跡可能性が維持されている。
8. システム要件、システム方式設計及びそれらの関係にベースラインが設けられていて、全ての影響される当事者に伝えられている。
9. 人的要因並びに人間工学知識及び人間工学手法がシステム設計に組み込まれている。
10. 人間中心設計の活動が識別され、実施されている。

(『共通フレーム 2013』, pp.142-143 より抜粋)

一方、ソフトウェア方式設計プロセスでは、要件を実装し、それに対して検証できるソフトウェアの設計を提供する事を目的とする。その構成タスクとして以下を定義し、これらを各ソフトウェア品目(明らかになっているならばソフトウェア構成品目)ごとに行う。

ソフトウェア方式設計を行う事で、以下のような状態となる事を目指す。

1. ソフトウェア要件を実装するソフトウェア品目を表現するソフトウェア方式設計が作成され、ベースラインが定められている。
2. 各ソフトウェア品目の内部及び外部インタフェースが定義されている。
3. ソフトウェア要件とソフトウェア方式設計との間の一貫性及び追跡可能性が確立されている。

(『共通フレーム 2013』, pp.156-157 から抜粋)

このように明確される定義がある一方で、一般的にソフトウェアアーキテクチャを考慮する際には多くの場合システムアーキテクチャ設計に含まれる要素の考慮が求められるため、我々ソフトウェア工学を学ぶものがアーキテクチャ設計と特に注釈なく述べた場合はソフトウェアアーキテクチャ設計のことを指す。一方、そこから足を広げシステムアーキテクチャ設計の範疇に至るまで議論が行われることも少なくないので注意が必要である(つまり、システムアーキテクチャ設計とソフトウェアアーキテクチャ設計に関しても分割の定義も曖昧であり明確な基準を持って語られる事が少ないと言う事)。

■成果物

アーキテクチャ設計の成果物としては、文章化されたアーキテクチャ設計書が求められ、システムアーキテクチャ設計、そしてソフトウェアアーキテクチャ含めた全ての工程終了後には以下のような状態となることが期待される。

1. クライアントからの要求を実現するための責務が、ハードウェア、ソフトウェア、手作業に振り分けられ、また、それを実現するためのハードウェア構成品目、ソフトウェア構成品目が明確となっていること。
2. 全てのシステム構成要素（ハードェア構成品目、ソフトウェア構成品目）が適切なサブシステムに分割されており、それぞれの役割や構造、必要となる性能・セキュリティ属性が明らかとなっていること。
3. 上記を満たすアーキテクチャの候補案が、評価・考察され、技術的・費用的なトレードオフおよびリスク分析がなされていること。
4. アーキテクチャの第一候補について、プロジェクトへの最適性が説明され、関係者の承認を得ていること。

14.2 代表的なアーキテクチャ設計の定義

アーキテクチャ設計は、上流工程の設計段階で、それぞれの工程分割が曖昧となりがちであることや、ソフトウェア工学の中でも若い分野であることもあり、統一的な定義や手法が明確となっていない。様々な組織がそれぞれ異なるアーキテクチャ設計の定義を提唱しており、それを実現する手法も多数存在する。本項では、世界標準とされる組織が提唱するアーキテクチャ設計の定義と、頻出するアーキテクチャ設計手法を概説し、アーキテクチャ設計における知識の全体感の醸成に役立てる。

■代表的なアーキテクチャ設計の定義

1. 共通フレーム 2013 （IPA)におけるシステム方式設計及びソフトウェア方式設計

   共通フレーム 2013 は、ソフトウェア、システム[34]、サービスの構想から開発、運用、保守、廃棄に至るまでのライフサイクルを通じて必要な作業項目、役割などを包括的に定めた共通の枠組みである。世界標準の一つである ISO の日本語版にあたる JIS X 0160 に日本独自のアレンジを加えて作成されている。共通フレーム 2013 においてアーキテクチャ設計(システム方式設計)とは、各システム要件を、業務効率、作業負荷、作業コスト、運用や保守の効率などの観点から検討し、ハードウェア、ソフトウェア、手作業に分類し、システムの構成を決定、システム要求事項と共に文章化することと定義される。

[34] システムとはソフトウェアやハードウェアそして手作業など全てを包括した何かを実現する為の仕組みを表す概念。ソフトウェアとシステムは明確に区別されるべき。

その構成タスクとして以下を定義する。

2.3.3 システム方式設計プロセス
2.3.3.1 システム方式の確立
2.3.3.1.1 システムの最上位レベルでの方式確立
a. 全てのシステム要件が品目に割り当てられている事を確実にする。
b. ハードウェア構成品目、ソフトウェア構成品目及び手作業を、これらの品目から識別する。
c. システム方式及び品目に割り当てたシステム要件を文書化する。
2.3.3.1.2 利用者用文書(暫定版)の作成
2.3.3.1.3 システム結合のためのテスト要件の定義
2.3.3.2 システム方式の評価レビュー
2.3.3.2.1 システム方式の評価
a. システム要件への追跡可能性
b. システム要件との一貫性
c. 使用する設計標準及び方法の適切さ
d. 割り当てられた要件を満たすソフトウェア品目の実現可能性
e. 運用及び保守の実現可能性
2.3.3.2.2 システム方式設計の共同レビューの実施

これらの方式が定まったのちには、システム要求事項への追跡可能性や、システム要求事項との一貫性、使用する設計標準や設計手法の適切性、ソフトウェア品目や運用・保守の実現可能性など様々な観点から、システム方式の評価を行うことを提唱している。

システム方式設計の後に以下の10項目が満たされていることを目標とする。

1. システムの要素を識別子、定義された要件を満たすシステム方式設計が定義されている。
2. システムの機能及び非機能要件が取り扱われている。
3. 要件がシステムの要素に割り当てられている。
4. 各システム要素の内部及び外部のインタフェースが定義されている。
5. システム要件とシステム方式との間の検証が行われている。
6. システムの要素及びそれらのインタフェースに割り当てられた要件は、顧客の要件ベースラインへ追跡可能である。
7. システム要件とシステム方式設計との一貫性及び追跡可能性が維持されている。
8. システム要件、システム方式設計及びそれらの関係にベースラインが設けられていて、全て

の影響される当事者に伝えられている。

9. 人的要因並びに人間工学知識及び人間工学手法がシステム設計に組み込まれている。
10. 人間中心設計の活動が識別され、実施されている。

システム方式設計が定まった後に、ソフトウェア方式設計プロセスが執り行われる。ソフトウェア方式設計プロセスでは、要件を実装し、それに対して検証できるソフトウェアの設計を提供する事を目的とし、以下のような工程が行われる。

2.4.3. ソフトウェア方式設計プロセス

2.4.3.1 ソフトウェア方式設計

2.4.3.1.1 ソフトウェア構造とコンポーネントの方式設計

a. ソフトウェア品目に対する全ての要件がソフトウェアコンポーネントに割り当てられる。

b. ソフトウェア品目に対する要件が、詳細設計を容易にするために細分化される事を確実にする。

c. ソフトウェア品目の方式を文書化する。

2.4.3.1.2 各インタフェースの方式設計

2.4.3.1.3 データベースの最上位レベルの設計

2.4.3.1.4 利用者用文書(暫定版)の作成

2.4.3.1.5 ソフトウェア結合のためのテスト要件の定義

2.4.3.1.6 ソフトウェア方式設計の評価

a. ソフトウェア品目の要件への追跡可能性

b. ソフトウェア品目の要件との外部一貫性

c. ソフトウェアコンポーネント間の内部一貫性

d. 使用した設計方法及び作業標準の適切さ

e. 詳細設計の実現可能性

f. 運用及び保守の実現可能性

2.4.3.1.7 ソフトウェア方式設計の共同レビューの実施

ソフトウェア方式設計を行う事で、以下のような状態となる事を目指す。

1. ソフトウェア要件を実装するソフトウェア品目を表現するソフトウェア方式設計が作成され、ベースラインが定められている。
2. 各ソフトウェア品目の内部及び外部インタフェースが定義されている。
3. ソフトウェア要件とソフトウェア方式設計との間の一貫性及び追跡可能性が確立されている。

2. IEEE1220 におけるアーキテクチャ設計

世界標準の一つである IEEE1220 では、ソフトウェア開発工程における設計段階を大きく分けて以下の6つのプロセスで構成されるものとしている。

① 要求分析
② 要求検証
③ 機能分析
④ 機能検証
⑤ 結合
⑥ 物理検証

この中で特に、アーキテクチャ設計は要求検証以下の4つのタスクで構成されているものとされ、それぞれに関して行われるべきタスクが明確にされている。

まず、要求検証の結果として与えられるシステム要件に基づき、機能分析プロセスが行われる。機能分析プロセスは以下の8つのタスクから構成される。

1 システム機能の分解
  1.1 サブ機能の定義
  1.2 機能インタフェースの定義
  1.3 性能要求の割付
2 機能の振る舞いの分析
3 サブ機能の状態・モードの定義
4 機能タイムラインの定義
5 データフローとコントロールフローの定義
6 機能の故障モードと影響の定義
7 危険モニタ機能の定義
8 機能アーキテクチャの確立

次に、機能分析プロセスにて明確化されたサブ機能の振る舞いや、性能要求に関する定義である、機能アーキテクチャに対し評価・検証を行う(機能検証プロセス)。機能検証プロセスは以下の4つのタスクから構成される。

1 検証手順の定義

2　検証評価の実施
　2.1　アーキテクチャの完全性
　2.2　機能及び性能指標の検証
　2.3　制約を満たしているかの検証
3　差異とコンフリクトの識別
4　検証された機能アーキテクチャの確立

機能検証プロセスにて、検証が行われ、要求分析で発見されたシステム要件との差異が最小化された、機能アーキテクチャを元に統合プロセスが行われる。統合プロセスを構成するタスクは以下となる。

1　機能グループ化と割付
2　物理的解決策の代替案の識別
3　安全性及び環境上の危険の評価
4　ライフサイクル品質ファクタの評価
5　技術要求の評価
6　物理的特性と性能特性の割付
7　物理的インタフェースの定義
8　標準化の機会の評価
9　汎用品利用の評価
10　Make/Buy 代替案の評価
11　モデル及びプロトタイプの開発
12　胡椒モード・影響及びクリティカリティ評価
13　テスト可能性の評価
14　進化の設計許容性評価
15　設計の終了
16　進化的開発の開始
17　図面、図表の作成
18　物理アーキテクチャの確立

最後に、統合によって確立される物理アーキテクチャの検証を行う物理検証プロセスが執り行われ、最終的なシステム構成図が生成される。物理検証プロセスは以下の8つのタスクから構成される。

1　検証方法の選定
　1.1　検査、解析、デモンストレーション及びテスト要求の定義

1.2　検証手順の定義

1.3　検証環境の確立

2　検証評価の実施

2.1　アーキテクチャの完全性の検証

2.2　機能及び性能指標の検証

2.3　制約を満たしていることの検証

3　差異とコンフリクトの識別

4　検証された設計アーキテクチャの確立

5　ライフサイクル成果物の検証された設計アーキテクチャの確立

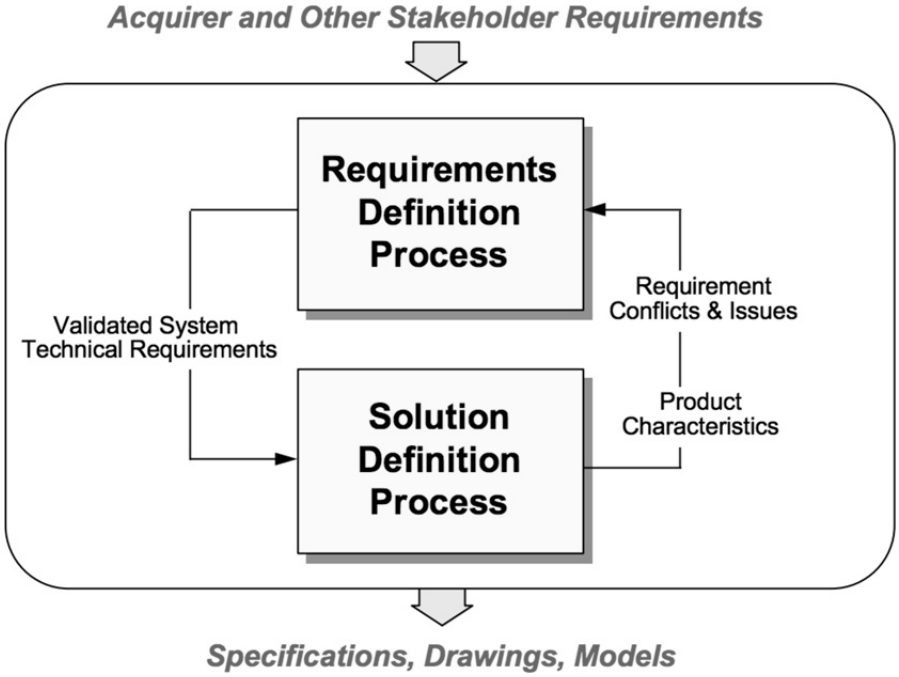
6　検証されたシステムアーキテクチャの確立

7　仕様と構成のベースラインの確立

8　製品構成図の作成

3.　ANSI/EIA632 におけるアーキテクチャ設計

ANSI/EIA632 において、アーキテクチャ設計に関する具体的な例示はないが、システム設計の中の要素である、解決策定義プロセスがそれに近いものであると考えられる。要件定義プロセスから最終的に目指すシステムの技術的要件を受けとり、それに対してプロダクトの特徴や、要求の実現における解決すべき問題点をフィードバックとして返す、と言うプロセスを繰り返すことで設計書を作成する。



14.3 アーキテクチャの構造とビュー

ここまで、広義な解釈における、アーキテクチャ設計の定義や意味、目的、成果物、そして代表的な機関の提唱するアーキテクチャ設計について議論を行ってきた。本節では、アーキテクチャ設計を行う際に考慮すべき構造やビューについて説明を行う。

読者の皆様は、ここまで説明を行ってきたアーキテクチャ設計の成果物として、一枚で全てを伝えることができる、ソフトウェア工学的に言うと、銀の弾丸(Silver Bullet)のような、アーキテクチャ設計書を想像していたのではないかと推察する。しかし、残念ながら現代のシステムは、一度にすべてを理解することができない程極めて複雑で、本書の執筆時点においてはそのような画期的な手法は存在しない。その代わりに先人達は、あるシステムにおける 1 つの構造に注意を限定しアーキテクチャを表す方法を多数生み出し、それらをシステムの特性に合わせ組み合わせることでシステムの全体像を表現することを可能とした。故に我々は 1 つのシステムについて議論する際にも、必ずこのアーキテクチャ表現はどのビューを用い、どの構造について議論しているのかを明確にしなければならない。まず、アーキテクチャ表現における構造とビューの定義について概説した後に、実際にアーキテクチャ設計の議論で多く用いられる構造について総括する。

■ アーキテクチャ設計における構造とビュー

システムアーキテクチャの全体像を理解する為には、システムの構成やデータ構造などの静的な側面から動作時の振る舞いなどの動的な側面まで、様々な切り口でシステムを捉える必要がある。構造とビューは、それらを捉える方法を提供する。以下にそれぞれの概念を示す。

- 構造

  アーキテクチャ設計における「構造」とは、システムアーキテクチャのある側面を捉える為のいわば切り口を表す。例えば、「モジュール構造」はシステムアーキテクチャを実装の単位となる「モジュール」と言う切り口で捉える事で、システムのモジュール分割がどのように為されているか、それぞれのモジュールはどのような役割を持つか、そしてモジュール同士の関係性はどのように成り立っているか、など、「システムの静的な構成」と言う側面を捉える。

- ビュー

  構造がシステムアーキテクチャの側面を捉える切り口であるならば、ビューは捉えられた側面を

人が理解できる形で表現するものである。顧客、ユーザー、プロジェクトマネージャー、開発者、テスト担当者、など、システムには様々なステークホルダーが存在する。それらのステークホルダーは、皆が異なる関心事を持ち、背景に関する理解も異なる。プロジェクトを円滑に進める為には、こうしたステークホルダーに対し、求められる情報を提供し、かつその情報の正確な理解を促す必要がある。同じ側面を表現する際も、受け取り手によって適切な表現方法は異なる為、ステークホルダーのニーズに応じて、適切な「ビュー」を用いて、「構造」を表現する事が重要となってくる。

ソフトウェアアーキテクチャの側面を捉える切り口である「構造」を大別すると、モジュール構造（システムの静的な構造）、コンポーネント&コネクター構造（動作時の振る舞い）、割当て構造（システム環境など非ソフトウェア構造との関連）の3つに分割する事が出来る。また、それぞれが更に、「ビュー」でそのまま表現出来るレベルに詳細な幾つかの構造から成り立つ。

1. モジュール構造

「モジュール構造」は、実装の単位となる「モジュール」を1つの要素としてシステムを捉える事で、「システムの静的な構成」を捉える為の切り口と言える。「システムの静的な構成」を捉えるためクラス図などを用いる事で表現する事が出来る[35]。モジュールには、機能的な責務範囲が割り当てられ、モジュール同士が相関する事で、1つのシステムが成り立つ。「モジュール構造」を表現する事は、次のような質問に答えることを可能とする。

（ア）各モジュールに割り当てられた主要な責務は何か。
（イ）モジュールが使用できる他のソフトウェア要素には何があるか。
（ウ）実際に使用する他のソフトウェアは何か。
（エ）汎化または特化（つまり継承）関係によって、他のモジュールと関係するのはどのモジュールか

この構造には、以下のものが含まれる。

- 分割（例:クラス図[36]）

35 アーキテクチャ設計のビューは、特定の記法で記述する事を決定づけられていない。アーキテクチャ記述言語（ADL）は一般的に線と図形で書かれることが多かったが、統一性のなさから近年 UML に代表されるような記法を用いる事も多くなった。
36 一般的な UML 記法で当該ビューを記述する際に有効であると考えられるものを記載している。必ずしもこの記法で記述しなければならないと言う形で限定しているわけではない事には注意が必要。以下

「分割構造」における基本単位は、プログラムの実装単位であるモジュールで、モジュール同士は互いに「～のサブモジュールである(is a submodule of)」の関係で関連付けられる。「分割構造」は、大きなモジュールが容易に理解できる程に十分小さくなるまで、どのように再帰的に分割されるかを示す。アーキテクトは、ソフトウェアの単位がしなければならないことを列挙して、後に続く(より詳細な)設計と最終的な実装のために、各項目をモジュールに割り当てる。モジュールは、成果物(すなわちインタフェース仕様書、コード、テスト計画など)と関連することが多い。「分割構造」は、起こりそうな変更がせいぜい少数の小さなモジュールの範囲で収まることを保証することで、システムの変更容易性の大部分を規定する。「分割構造」は、開発プロジェクトの編成の基礎として使用されることが多い。それには、文書の構造、統合とテストの計画が含まれる。この構造の単位には、組織固有の名前がつけられることが多い。

- 使用(例:クラス図)

「使用構造」の単位は、「分割構造」同様にモジュール、またはモジュールのインタフェース上の手続きやリソースである(以下。便宜上これらもモジュールと呼び説明を続ける)。この単位は、使用(uses)関係によって関連づけられる。あるモジュールが別のモジュールを使用する(使用関係にある)と言う事を定義付けると、「使用する側のモジュールが、使用される側のモジュールの正確なバージョンの存在を必要とする」となる。これらの関係はいわば依存関係でもあり、使用する側のモジュールは、使用される側のモジュールに依存関係を持つ。「使用構造」が明確になる事で、各モジュールの役割が明確になる為、拡張して機能性を追加する事が容易になる、また、そこから有益な機能性のサブセットを簡単に抽出可能となる事で再利用性が高まる。使用する側のモジュールからは、使用される側のモジュールの内部を知る必要がない事から移植性も高まる。

- レイヤー(例:クラス図)

「使用構造」ではモジュール同士の使用関係を捉えたが、関連する機能の一貫した集合である「レイヤー」を単位として、それらの使用関係からアーキテクチャを捉える構造が「レイヤー構造」である。レイヤーは複数の階層で構成されており、理想的にはn番目のレイヤーはn-1番目のレイヤーのサービスしか利用できない構成を取ることが望ましい。しかし、実際のシステムアーキテクチャにおいては、この制限を飛び越えてサービスを利用する必要が出

のビューに関しても同様。

てきてしまう為、絶対的な制約と言う訳ではない。レイヤーは、「使用構造」と同様に、上位のレイヤーから下位の実装の詳細を隠蔽する事が出来る為、移植性が高まる。「使用構造」よりも抽象度の高い形でシステムを捉えており、技術的な背景を持たないステークホルダーに対して、システムアーキテクチャの静的な成り立ちを説明する上でより効力を発揮する。

- クラスあるいは汎化（例：クラス図）

「クラスあるいは汎化構造」においても構成要素の基本単位はモジュールであるが、この構造を表現する際にモジュールを「クラス」と呼ぶ事も多い。クラス同士の関係は「～から継承する（inherites-from）」あるいは「～のインスタンスである（is an instance of）」である。この構造を表現する事で、類似の振る舞いや機能を持つクラス同士を見つけ継承関係を持たせる事が容易となる。単なる独立したモジュールの集まりよりも、人が理解しやすい形でシステムを構成する事が可能となる。また、親子関係にあるクラスのパラメータ化される違いについて論理的に考察することで、何が汎化可能であるかを見極める事が可能となり移植性が高まる。

2. コンポーネント＆コネクター構造

「コンポーネント＆コネクター構造」では、計算処理の基本単位となる「コンポーネント」と、コンポーネント間の通信手段である「コネクター」を基本要素としてシステムを捉える事で、「システム動作時の振る舞い」を捉える。「動作時の振る舞い」を捉えるためシーケンス図やアクティビティ図を用い表現することが出来る。具体的には、以下のような事柄を明確にする。

（ア）主要な実行コンポーネントは何か、そしてそれらはどのように相互作用するか。
（イ）主要な共有データは何か。
（ウ）システムのどの部分が複製されるのか。
（エ）データがシステムの中をどのように進行するのか。
（オ）システムのどの部分が並行して実行可能なのか。
（カ）実行とともに、システムの構造をどのように変更できるのか。

この構造には、以下のものが含まれる。

- プロセスあるいは通信プロセス（例：シーケンス図、アクティビティ図）

「プロセスあるいは通信プロセス構造」ではプロセスやスレッドを基本単位として、システムアーキテクチャを捉える。プロセスやスレッドは、それぞれが独立した計算処理を行っている中で、通信、同期化、あるいは排他的操作などによって互いに接続される。これらの関係は「付属物」と呼ばれ、具体的には「コンポーネント」と「コネクター」がどのように結びつくかを示す。これらプロセスやスレッドにおけるやり取りは、場合によってはオーバーヘッドになってしまう事もある一方で、処理を分散する事で性能が向上するケースもある為、システムの品質特性を議論する上では大変重要な構造となる。

- 並行性（例：アクティビティ図）

「並行性構造」は、アーキテクトが並行性を導入できる状況とリソース競合が発生する可能性のある箇所を判断できるようにする。計算処理の基本単位となる「コンポーネント」をベースにシステムアーキテクチャを捉え、ここでのコネクターは「論理スレッド」となる。論理スレッドは、演算処理の順序を示すもので、後の設計プロセスで独立した物理スレッドに割り当てることができる。並行性構造は、設計の早い段階で使用され、並行した実行に関連する問題を管理するのに必要な要求を特定する。

- 共有データまたはリポジトリ

この構造は、永続データを作成、記憶、アクセスするコンポーネントと、コネクターで構成される。システムが実際に1つ以上の共有データリポジトリを使って構築される場合、この構造はそれを説明するための良い構造の 1 つになる。この構造は、実行時の要素がデータをどのように生成して、消費するのかを示して、優れた性能とデータの完全性を保証するために使用できる。

- クライアントサーバ（例：シーケンス図、アクティビティ図）

システムが協力するクライアントとサーバーのグループとして構築される場合、「クライアント・サーバ構造」がシステムの全体構成を説明するための優れた構造となる。「クライアント・サーバ構造」におけるコンポーネントは、「クライアント」と「サーバ」である。そして、コネクターはコンポーネントがシステムの機能を実行するために「共有するプロトコル」と「メッセージ」である。この構造は、関心事の分離（変更容易性を支援する）、物理的な分散、および負荷均衡（実行時の性能を支援する）など、メッセージによる通信のオーバーヘッドを考慮した上で、何をサーバで実行し、何をクライアントで実行するのか、またそのやり取りはどのよう

になされるのかを議論する上で重要となる。

3. 割当て構造

システムはソフトウェアのみで成り立つ訳ではなく、ソフトウェアを配備し計算を行うハードウェアや、プロセスの途中で人手による作業が必要なケース、は必ず存在し、それらとの相関がシステムの性能や使いやすさなど様々な品質特性を決定づける。またシステム開発という側面から考えると、実装時の各要素をどのファイルに格納するか、モジュール[37]をどのチームが開発するかなど、様々な非ソフトウェア要素との相関が存在する。「割り当て構造」では、このような「非ソフトウェア要素」と「ソフトウェア要素（ソフトウェアを構成するモジュールやコンポーネント[38]と言ったソフトウェアの一部を便宜上ソフトウェア要素と呼ぶ）」の相関を表す。「割り当て構造」と言う視点でシステムアーキテクチャを見る事で、次のような質問に答える事が可能となる。

（ア）各ソフトウェア要素はどのプロセッサで実行されるのか。
（イ）開発、試験、システム構築の間に、各要素はどのファイルに格納されるのか。
（ウ）開発チームにソフトウェア要素の実装と結合に対する責務は何か？
この構造には、以下のものが含まれる。

- 配置（例：デプロイメント図）

「配置構造」は、ソフトウェアがハードウェア要素と通信要素にどのように割り当てられるかを示す。この構造の要素は、ソフトウェア、ハードウェア実体（プロセッサ）および通信経路である。この構造では、「～に割り当てられる（allocated to）」関係を使って、どの物理要素にソフトウェア要素を配置するのかを示す。割当が動的であれば、「～に移動する（migrates to）」関係が使われる。「配置構造」を表現するビューを作成しそれを基に議論することで、性能、データ完全性、可用性、セキュリティ性に関して論理的に考察できるようになる。「配置構造」はスタンドアロンなシステムにおいては、余り配置構造による差異が発生しない為、分散あるいは並列システムにおいて特に有用である。

37 モジュールとは機能単位、交換可能な構成部分という意味の英単語。システムへの接合部(インターフェース)が規格化・標準化されていて、容易に追加や削除ができ、ひとまとまりの機能を持った部品のこと。
38 コンポーネントとは何らかの機能を持った、プログラムの部品。プログラムだけでなく、ハードウェアや組織の一部を指して用いられることもある。プログラム以外の分野で使われる例としては、機械のオプションパーツなどがある。

- 実装

「実装構造」は、ソフトウェア要素（通常はモジュール）が、システムの開発、統合、あるいは構成管理環境のファイル構造にどのようにマッピングされるのかを示す。ファイル構造はシステム開発において大変重要な要素で、複数のチームで行われるような大規模開発においては、システム自体の静的構成が如何に整っていても、ファイル分割が適切に行われていないと必要以上に時間を取られるなど、プロジェクトの進捗に大きな影響を与える。「実装構造」を早期の段階から捉え、適切に表現し可視化する事は、開発活動と構築プロセスの管理に良い影響を与える。

- 作業割当

「作業割当構造」は、実装と統合に対する責務を適切な開発チームに割り当てる。誰が作業をするのかについての決定が、管理上の意味だけでなく、アーキテクチャ上の意味を持つということが明確になる。アーキテクトは、各チームに必要な専門知識を理解するようになる。また、大規模な複数の拠点に分散した開発プロジェクトでは、作業割当て構造は、機能的な共通性の単位を呼び起こし、それらを必要としている全員に実装させるのではなく、それらを単一のチームに割り当てるための手段になる。

（構造の分割について、『実践ソフトウェアアーキテクチャ』, pp.46 - 49 を参考に筆者が各構造に関する説明を追記）

これらの構造1つ1つが、システムに対する異なる視点を与え、特定の側面からアーキテクチャを捉える切り口となる。構造1つ1つを分析する事はそれだけでも有益であるが、システムの全体像を捉え、アーキテクチャを設計する上では、これらを独立に捉えるのではなく、それぞれの関係を十分に理解した上で、複合的に捉える事が必要になる。例えば、システムの性能を考える上では、静的な構成を捉える「モジュール構造」と、動作時の振る舞いを捉える「コンポーネント&コネクター構造」は、合わせて考慮されなければ、包括的な議論は出来ない。システムの並列性を議論するのであれば、「並行性構造」でソフトウェア要素間のやり取りは網羅出来るが、実際にどのような性能を持つハードウェアに配置されるかで、全く挙動が変わってきてしまうので、「配置構造」と合わせて議論がなされるべきである。このように複数のビューを複合的に捉える事で、よりシステムアーキテクチャについて詳細な議論が可能となる。一方、全ての構造を取り入れ、多くのビューを用い表現する事が善かと言えば、それはまた異なる。情報は多すぎてもそれを捉える人は混乱してしまうし、何より多くのビューを作成することは工数もかかる。次項で触れる、システムにおける品質特性や、前章で触れた機能要件・非機能要件と言った、システム

要件を捉えた上で、どの構造について議論をすべきで、どの構造をビューとして表現していくべきか、を考慮していくことが求められる。

15アーキテクチャと品質特性

アーキテクチャ設計を実施する上で、大変重要な要素としてシステム要件を正しく把握することがあげられる。第一部にて触れられた要件定義はまさしくそのシステム要件を、ステークホルダーからのヒアリングを通じて抽出する作業である。抽出されたシステム要件は、大きく分けると機能要件と非機能要件から構成され、非機能要件の中でもアーキテクチャと密接に関わりがあるのが、システムの満たすべき品質特性を列挙した品質要件である。

本項では、ソフトウェア工学におけるソフトウェアの品質を体系的に捉えられている ISO/IEC 9126-1:2001(JIS X 0129-1:2003)という規格が提唱するソフトウェア品質モデルを概説し、品質特性の概念について理解を深める。次に、品質特性の中でも特にアーキテクチャ設計(特に品質駆動アーキテクチャ設計)において考慮される事が多い CMU/SEI アーキテクチャ手法群におけるシステム品質特性とその品質特性シナリオについて議論した上で、最後にそれら品質特性シナリオを満たすための実現手法を詳説する。

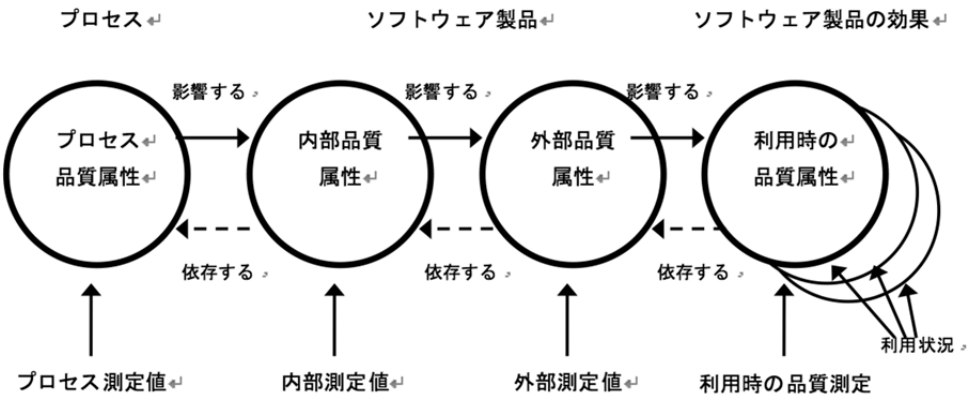
15.1 ソフトウェア品質モデル

ソフトウェア工学の分野において、品質に関する議論は古くから行われており、様々な組織や研究者が多種多様な概念を提唱しているが、近年はこれらに対する合意形成が議論の中心となってきている。ISO/IEC 9126-1:2001(JIS X 0129-1:2003)[39]もその中で産まれた規格の一つで、ソフトウェア品質モデルと言うモデルを定義している。ソフトウェア品質モデルでは、ソフトウェア品質を主に品質属性、品質特性、そして品質服特性という3つの概念からなる構造で表現したモデルであり、測定可能な形で各開発工程における考慮されるべき品質を定義したことで定評を得ている。

■ 品質属性とは

ISO/IEC 9126-1:2001(JIS X 0129-1:2003)では、ソフトウェア品質の局面ごとの推移を、下図のようにモデル化している。

[39] ISO/IEC 9126 は、ソフトウェア品質の評価に関する国際規格である。「品質モデル: quality model」、「外部測定法: external metrics」、「内部測定法: internal metrics」、「利用時品質測定法: quality in use metrics」の４つの部分から成る。特に品質モデルに関しては ISO/IEC9126-1 で規定している。
JIS X 0129 は特にソフトウェア製品の品質に関わる部分の日本語翻訳された規格である。



このモデルでは、ソフトウェアシステムには以下の4つの局面が存在するとしている

(ア) ソフトウェアが製造される以前に製造工程だけが存在する局面
(イ) ソフトウェアが仕様書、設計図、ソースコードとして存在する局面
(ウ) 利用環境を無視してソフトウェアが単独実行される局面
(エ) 利用者が特定の環境において実際にソフトウェアを利用する局面

各局面において、それぞれプロセス品質属性、内部品質属性、外部品質属性、利用時の品質属性と言う4つの測定可能な属性が用意されている。それぞれの属性は影響関係があり、図では左から右へと影響が伝播していく。

このうちプロセス品質属性は、ソフトウェアの品質属性よりも上位の概念である開発プロセスを評価する際の属性であるため本章では説明を割愛するが、他3つの属性は以下のように定義される。

A) 内部品質属性
仕様書、設計図、ソースコードなどを対象とすることにより測定することが可能な属性。変更・修正のしやすさを表す保守性や、ある環境から他の環境に移すことの容易さを表す移植性などを含む。

B) 外部品質属性
完成したソフトウェアをテスト実行し想定したとおりに動作するかどうかを検査することで測定することが出来ると考えられている。求められている機能を十分に満たすかどうかを表す機能性、指定された達成水準を維持出来るかどうかを表す信頼性、理解、習得の容易さなど

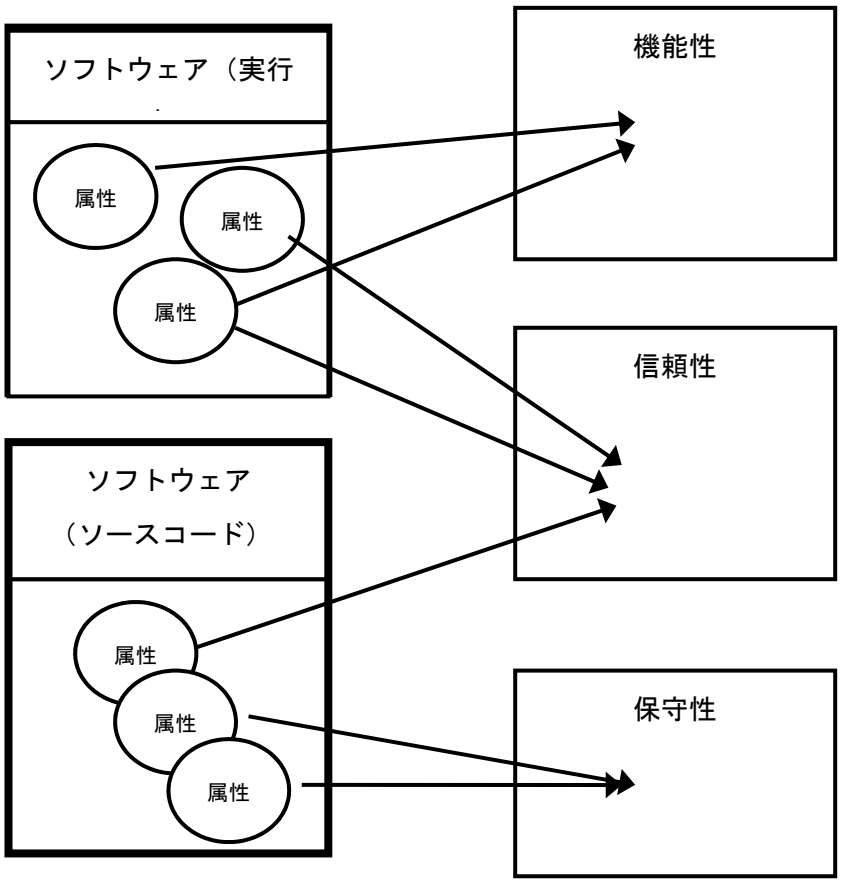
利用者に対する魅力を表す使用性などからなる。

C) 利用時の品質属性

想定される利用者の環境において、利用者の目的がどの程度達成できるかを検査することで測定することが出来ると考えられている。有効性、生産性、安全性、満足性などからなる。

■ 品質特性・品質複特性とは

品質特性は様々な品質属性を分類するためのグループである。以下の図のように、ひとつの品質特性には複数の品質属性が含まれる。たとえば、「読みにくい」、「変更しやすい」といった内部品質属性は「保守性」の品質特性に分類される。



図にも示したように、品質属性と品質特性の関係は、単に品質特性が複数の品質属性を含む含有の関係ではなく、一つの品質属性が複数の品質特性に分類される可能性があることには留意が必要である。

ISO/IEC 9126-1:2001(JIS X 0129-1:2003)のソフトウェア品質モデルでは、内部品質属性と外部品質属性は、品質特性と、さらに細分化された品質副特性にグループ化される一方で、利用時の品質特性は、品質副特性への分類が規定されておらず、プロセス品質特性に至っては品質特性への分類も

なされていない。

以下、ISO/IEC 9126-1:2001(JIS X 0129-1:2003)のソフトウェア品質モデルで規定される、それぞれの品質属性の品質特性及び品質副特性への分類を示すことで、品質特性とはどういったものが具体的な視点を持つ一助となることを期待する。

- 内部品質属性と外部品質属性

品質特性	品質副特性	主な内容
機能性 (functionality) ソフトウェアを指定された条件の下で利用するとき、明示的および暗示的必要性に合致する機能を提供するソフトウェア製品の能力。	合目的性(suitability)	指定された作業および利用者の具体的目標に対して適切な機能の集合を提供するソフトウェア製品の能力。
	正確性(accuracy)	必要とされる精度で、正しい結果もしくは正しい効果、または同意できる結果もしくは同意できる効果をもたらすソフトウェア製品の能力。
	相互運用性 (interoperability)	一つ以上の指定されたシステムと相互作用するソフトウェア製品の能力。
	機密性(security)	許可されていない人、またはシステムが情報またはデータを読んだり、修正したりすることができないように、および許可された人、またはシステムが情報またはデータへのアクセスを拒否されないように、情報またはデータを保護するソフトウェア製品の能力。

品質特性	品質副特性	主な内容
	機能性標準適合性 （compliance）	機能性に関連する規格、規約または法律上および類似の法規上の規則を遵守するソフトウェア製品の能力。
信頼性 （reliability） ソフトウェアを指定された条件の下で利用するとき、指定された達成水準を維持するソフトウェア製品の能力。	成熟性（maturity）	ソフトウェアに潜在する障害の結果として生じる故障を回避するソフトウェア製品の能力。
	障害許容性 （fault tolerance）	ソフトウェアの障害部分を実行した場合、または仕様化されたインターフェイス条件に違反が発生した場合に、指定された達成水準を維持するソフトウェア製品の能力のこと。指定された達成水準にはフェイルセーフの能力を含んでもよい。
	回復性（recoverability）	故障時に、指定された達成水準を再確立し、直接に影響を受けたデータを回復するソフトウェア製品の能力。
	信頼性標準適合性 （compliance）	信頼性に関連する規格、規約または規則を遵守するソフトウェア製品の能力。

品質特性	品質副特性	主な内容
使用性（usability） ソフトウェアを指定された条件の下で利用するとき、理解、習得、利用でき、利用者にとって魅力的であるソフトウェア製品の能力。	理解性（understandability）	ソフトウェアが特定の作業に特定の利用条件で適用できるかどうか、 およびどのように利用できるかを、利用者が理解できるソフトウェア製品の能力。
	習得性（learnability）	ソフトウェアの適用を利用者が習得できるソフトウェア製品の能力。
	運用性（operability）	利用者がソフトウェアの運用および運用管理を行なうことができるソフトウェア製品の能力のこと。合目的性、変更性、環境適応性、および設置性といった他の品質副特性のある側面は、運用性に影響を与える場合がある。
	注目性(attractiveness)	利用者にとって魅力的であるためのソフトウェア製品の能力のこと。色彩の利用やグラフィカルデザイン性のように、利用者にとってソフトウェア製品をもっと魅力的なものにするための属性に相当する。

品質特性	品質副特性	主な内容
	使用性標準適合性（compliance）	使用性に関連する規格、規約または規則を遵守するソフトウェア製品の能力。

品質特性	品質副特性	主な内容
効率性 （efficiency） 明示的な条件の下で、使用する資源の量に対比して適切な性能を提供するソフトウェア製品の能力。	時間効率性 （time behavior）	明示的な条件の下で、 ソフトウェアの機能を実行する際の適切な応答時間、処理時間、および処理能力を提供するソフトウェア製品の能力。
	資源効率性 （resource behavior）	明示的な条件の下で、ソフトウェアの機能を実行する際の、資源の量および資源の種類を適切に使用するソフトウェア製品の能力。
	効率性標準適合性 （compliance）	効率性に関連する規格、規約または規則を遵守するソフトウェア製品の能力。
保守性 （maintainability） 修正のしやすさに関するソフトウェア製品の能力。修正は、是正もしくは向上、または環境の変化、要求仕様の変更および機能仕様の変更にソフトウェアを適応さ	解析性（analyzability）	ソフトウェアにある欠陥の診断または故障原因の追及、およびソフトウェアの修正箇所の識別を行なうためのソフトウェア製品の能力。
	変更性（changeability）	指定された修正を行なうことができるソフトウェア製品の能力のこと。修正の実施にはコーディング、設計、および仕様書の変更を含む。

品質特性	品質副特性	主な内容
せることを含めてもよい。	安定性（stability）	ソフトウェアの修正による、予期せぬ影響を避けるソフトウェア製品の能力。
	試験性（testability）	修正したソフトウェアの妥当性確認ができるソフトウェア製品の能力。
	保守性標準適合性（compliance）	保守性に関連する規格、規約または規則を遵守するソフトウェア製品の能力。

品質特性	品質副特性	主な内容
移植性 （portability） ある環境から他の環境に移すためのソフトウェア製品の能力のこと。環境には組織、ハードウェアまたはソフトウェアの環境を含めてもよい。	環境適応性（adaptability）	ソフトウェアにあらかじめ用意された以外の付加的な作業または手段なしに、指定された異なる環境にソフトウェアを適応させるためのソフトウェア製品の能力のこと。環境適応性は、内部的容量（例えば、画面／表／トランザクションの規模など）の拡大縮小の調整能力を含む。
	設置性（installability）	指定された環境に設置するためのソフトウェア製品の能力。
	共存性(co-existence)	共通の資源を共有する共通の環境の中で、他の独立したソフトウェアと共存するためのソフトウェア製品の能力。
	置換性（replaceability）	同じ環境で、同じ目的のために、他の指定されたソフトウェア製品から置き換えて使用することができるソフトウェア製品の能力のこと。アップグレード時の新しい版の置換性も重要である。

品質特性	品質副特性	主な内容
	移植性標準適合性（compliance）	移植性に関連する規格、規約または規則を遵守するソフトウェア製品の能力。

- 利用時の品質属性

利用時の品質属性とは、特定の環境および特定の利用状況における、利用者の視点に基づくソフトウェアの品質属性を表す。これはソフトウェアシステム全体に対して認識される属性ではなく、特定の環境下での利用において、利用者がソフトウェアの利用によって目標をどの程度達成することができるかどうかを示す。同じソフトウェアであっても、環境や利用者が異なれば属性の測定結果は異なる可能性があるため、多様な角度から評価を行う事が求められる。

品質特性	主な内容
有効性	利用者が指定された利用の状況で、正確かつ完全に、指定された目標を達成できるソフトウェア製品の能力のこと。
生産性	利用者が指定された利用の状況で、達成すべき有効性に対応して、適切な量の資源を使うことができるソフトウェア製品の能力のこと。資源には、作業を完了するまでの時間、利用者の労力、材料または使用した費用を含めることができます。
安全性	利用者が指定された利用の状況で、人、事業、ソフトウェア、財産または環境への害に対して、容認できるリスクの水準を達成するためのソフトウェア製品の能力のこと。リスクは、一般に、機能性（セキュリティを含む）、信頼性、使用性または保守性の欠陥の結果です。
満足性	指定された利用の状況で、利用者を満足させるソフトウェア製品の能力のこと。満足性は、製品を対話的に利用したときの利用者の反応であり、製品の利用を含みます。

15.2〈内部品質属性〉と〈保守性〉

これまで扱ってきたように、ソースコードにおける〈品質属性〉は、〈内部品質属性〉です。そこで、ソースコードに関する〈内部品質属性〉の中でも、〈品質特性〉のひとつ〈保守性〉（変更の容易さ）に分類される〈属性〉を明確にしておくことは、仕様変更に耐えられる柔軟なソースコードを作成しようとした場合、重要な指標になります。つまり、ソースコードが、どのような〈内部品質属性〉を備えているときに、そのソ

ースコードは〈保守性〉に優れたソースコードになるのかという問題です。

〈保守性〉に優れたソースコードが備えているべき〈属性〉は、1968-1976年頃[40]に、〈構造化設計〉という設計技術が確立していく過程で考案されています。"COMPOSITE/STRUCTURED DESIGN"の著者であるGlenford J. Myersは、次のようにいっています。

> ・・・まず、うまく構造化されたプログラムがもっている属性とは何かとか、それらの属性はうまく構造化されていないプログラムのそれとどう違うのかといった問題について考えなければならない。よいプログラム構造とわるいプログラム構造とのはっきりした大きなちがいは、その**複雑さ**(complexity)にあると思われる。そこで、どんな種類のシステム(たとえば、コンピュータ・プログラム、人間の組織、生物のシステム、自動車)にでも共通に利用できる、複雑さを減少させるための3つの手段に焦点を当てて考えてみよう。これらの手段とは、次の3つである。
>
> 1. 識別可能な、わかりやすい境界をもった部分にシステムを区分けすること
> 2. システムを階層構造的に表現すること
> 3. システムの各部分間の独立性を最大にすること
>
> (Myers, Glenford J,『ソフトウェアの複合/構造化設計』, p.29: 邦訳中のコンマ、ピリオドは、引用者によってそれぞれ句点、読点に変更してしている。以下同様。)[41]

この3点は、さらに、プログラムの〈区分け〉(プログラムのモジュール化[42])、プログラムの〈階層構造化〉(モジュールの階層構造化[43])、モジュールの〈独立性〉として説明されています。このうち、プログラムの〈区分け〉と〈階層構造化〉は、人間のプログラムに対する理解を助けるためのものとされています(Myersはこの点のメリットだけを挙げています)。これは、ISO/IEC 9126-1:2001(JIS X 0129-1:2003)の〈品質特性〉のひとつである〈保守性〉の〈副特性〉、「解析性」(analyzability)の向上に役立つことになります。そして、最後のモジュールの〈独立性〉が、「最も重要な」ポイントとされています。

---

40 Edward Yourdon、Larry L. Constantine、Glenford J. Myersらが、〈構造化設計〉(Structured Design)、〈複合設計〉(Composite Design)といった設計技法の確立を目指して様々な論文や著作を発表していた時期である。

41 Myers, Glenford J., "COMPOSITE/STRUCTURED DESIGN", Van Nostrand Reinhold Company, 1978(國友義久・伊藤武夫,『ソフトウェアの複合/構造化設計』, 近代科学社, 1979)

42 「実行可能なプログラム命令の集合」のこと(同前 p.15)。Javaではメソッドもしくはクラスのこと。「モジュール化」とは、メソッドやクラスといった集合にプログラムを分割することを意味する。

43 Javaなどのオブジェクト指向言語によって開発されるプログラム(ソフトウェア)では、階層構造化は必ずしも、そのままは当てはまらない。

> よいプログラムの設計の目的は、プログラムを単に階層構造に区分けするのではなく、各モジュールが他のすべてのモジュールからできるだけ独立するように、プログラムを階層構造に区分けする方法を決定することである。
> (同前 p.32: 強調は引用者による)

モジュールの〈独立性〉が損なわれているシステムでは、「理解するのがむつかしい」、「保守がむつかしい」、「拡張がむつかしい」(同前 p.34)という問題が生じると考えられています。つまり、ISO/IEC 9126-1:2001(JIS X 0129-1:2003)の〈品質特性〉のひとつ〈保守性〉を低下させると考えられています。さらに、Myers は、モジュールの〈独立性〉を実現する方法について考察しています。

> モジュールの独立性を高くするためには、各モジュール内の関連性を最大にすることと、モジュール間の関連性を最小にすることの2つの面から考えていく必要がある。
> (同前 p.39)

この2つの面を、Myers は、〈モジュール強度〉(module strength)[44]と〈モジュール結合度〉(module coupling)という2つの指標(〈属性〉)として取り出します[45]。この2つの指標(〈属性〉)により、〈保守性〉の品質に関して優れたソースコードが、どのような〈属性〉をもつべきかが明らかになります。その〈属性〉とは、「モジュール強度が高いこと」(モジュールが内部要素の関連性を保ち、無関係な要素を含まず一定の機能だけを含むこと)と、「モジュール結合度が低いこと」(モジュールとモジュールは強く結びつかず、緩やかな関係を保っていること)の 2 点です。これら 2 つの〈属性〉が、ソースコードに見いだされたとき、そのソースコードは、〈保守性〉に優れたソースコードであると判断することができます。

では、これらの〈属性〉(「モジュール強度が高いこと」と「モジュール結合度が低いこと」)は、Java のソースコードにおいては、どのような実装として見いだすことができるでしょうか。

---

44 〈凝集度〉(cohesion)とほぼ同義の概念。〈凝集度〉は、Edward Yourdon と Larry L. Constantine の用語である(Yourdon, Edward & Constantine, Larry L., "STRUCTURED DESIGN:Fundamentals of a Discipline of Computer Program and Systems Design", Prentice-Hall, 1979, p.105 参照)。

45 ひとつ大きな問題がある。Myers は、モジュール内の関連性およびモジュール間の関連性が、何によって増大・減少するのか、その原因を明確にしていない。つまり、〈モジュール強度〉と〈モジュール結合度〉という〈属性〉の強弱を決定する原因を明確にしていない。ただ、〈モジュール強度〉の低いパターンから高いパターンに至る階級と、〈モジュール結合度〉の高いパターンから低いパターンに至る階級を 7 つ挙げているだけである。Myers によれば、〈モジュール強度〉が最高の状態は、「機能的強度」(funtional strength)の階級に達したときであり、この場合は「1 組の機能が、集約的な方法で、1 つの固有の機能としてまとめて記述できる」(邦訳 p.49)と説明されている。私(芦澤)は、〈モジュール強度〉、〈モジュール結合度〉という〈属性〉の強弱を決定する原因のヒントは、広義の「再利用性」だと考えている。

15.3 ソースコード①

次のソースコードの属性(〈モジュール強度〉)について考えてみます。とくに、Calculator クラス内の calc メソッドに注目します。

```
public class Test {

    public static void main(String args[]) {

        Calculator c = new Calculator();

        c.operation();

    }
}

class Calculator {

    public void operation() {

        // 一つ目の商品

        String name = "XPS 9100";

        int price = 150000;

        int taxedPrice = (int) (price + price * 0.05);

        System.out.println(name + "/税込価格/" + taxedPrice);

        // 二つ目の商品

        name = "Inspiron 780";

        price = 108000;
```

```
            taxedPrice = (int) (price + price * 0.05);

            System.out.println(name + "/税込価格/" + taxedPrice);

            // 三つ目の商品

            name = "Dimension R400";

            price = 123000;

            taxedPrice = (int) (price + price * 0.05);

            System.out.println(name + "/税込価格/" + taxedPrice);
        }
    }
```

実行結果

```
XPS 9100/税込価格/157500
Inspiron 780/税込価格/113400
Dimension R400/税込価格/129150
```

このソースコードで最初に目につくのは、消費税率という同じ意味の 0.05 という double 型のリテラル値です。このリテラル値は、同一の値が 3 箇所で使用されているにもかかわらず、それぞれが独立したリテラル値であるため、関連性が薄れています(バラバラに存在している)。これらは、意味としては消費税率という同一の意味をもつため意味上の関連性はありますが、ソースコード上では、独立したリテラル値であるため、要素間の関連性が低くなっています。つまり、「モジュール強度が低い」という〈属性〉をもったソースコードだといえます。

この場合のデメリットは、消費税率が変更になった場合、3 箇所のリテラル値をすべて修正する必要があり、〈品質特性〉の〈保守性〉が低いソースコードだといえます。

15.4 ソースコード②

変数を導入することにより、バラバラの三つのリテラル値が一つの変数へと統合され、メソッド内の要素の関連性が高くなっています。

```
public class Test {

    public static void main(String args[]) {

        Calculator c = new Calculator();

        c.operation();

    }
}

class Calculator {

    public void operation() {

        // 変数を導入する

        double taxRate = 0.05;

        // 一つ目の商品

        String name = "XPS 9100";

        int price = 150000;

        int taxedPrice = (int) (price + price * taxRate);

        System.out.println(name + "/税込価格/" + taxedPrice);

        // 二つ目の商品
```

```
            name = "Inspiron 780";

            price = 108000;

            taxedPrice = (int) (price + price * taxRate);

            System.out.println(name + "/税込価格/" + taxedPrice);

            // 三つ目の商品

            name = "Dimension R400";

            price = 123000;

            taxedPrice = (int) (price + price * taxRate);

            System.out.println(name + "/税込価格/" + taxedPrice);
        }
    }
```

実行結果

```
XPS 9100/税込価格/157500
Inspiron 780/税込価格/113400
Dimension R400/税込価格/129150
```

この修正により、〈モジュール強度〉はやや高くなったといえます。消費税率が変更になった場合、1箇所だけを修正することにより対応することができます。つまり、〈品質特性〉の〈保守性〉に関していえば、やや高くなったといえます。

しかし、まだ問題があります。税込価格を計算するという同じ意味の処理が、3 箇所に重複しており、同一の処理であるにもかかわらず、3 箇所にバラバラに存在しています。この場合の問題は、税込価格

の算出方法に何らかの修正が必要になった場合、3 箇所の計算式を修正する必要があるということです。つまり、〈保守性〉が低いソースコードです。さらに、同一の処理がバラバラに重複して存在していることだけでなく、この operation メソッドの内部に税込価格の算出という独立した機能が混入しているといういみでも、関連性が低い、つまり、〈モジュール強度〉が低い状態になっています。

15.5 ソースコード③

3 箇所は、独立したメソッド getTaxedPrice へと統合されることにより、〈モジュール強度〉が高くなっています。これにより、計算式の修正が生じても作業は 1 箇所に集中され、〈保守性〉が向上しています。

```
public class Test {

    public static void main(String args[]) {

        Calculator c = new Calculator();

        c.operation();

    }
}

class Calculator {

    public void operation() {

        // 変数を導入する

        double taxRate = 0.05;

        // 一つ目の商品

        String name = "XPS 9100";

        int price = 150000;

        int taxedPrice = getTaxedPrice(taxRate, price);

        System.out.println(name + "/税込価格/" + taxedPrice);

        // 二つ目の商品
```

```
        name = "Inspiron 780";

        price = 108000;

        taxedPrice = getTaxedPrice(taxRate, price);

        System.out.println(name + "/税込価格/" + taxedPrice);

        // 三つ目の商品

        name = "Dimension R400";

        price = 123000;

        taxedPrice = getTaxedPrice(taxRate, price);

        System.out.println(name + "/税込価格/" + taxedPrice);
    }

    private int getTaxedPrice(double taxRate, int price) {
        return (int) (price + price * taxRate);
    }
}
```

実行結果

```
XPS 9100/税込価格/157500
Inspiron 780/税込価格/113400
Dimension R400/税込価格/129150
```

これでもまだ問題があります。商品名、「税込価格」という文字列、税込価格の値の三者をスラッシュ(/)で区切るという形式で出力するという同じ意味の処理が、3箇所にバラバラに存在しています。〈モジ

ュール強度〉がまだ弱い状態です。このままでは、出力の形式が変更になったとき、3箇所変更する必要が生じます。つまり、〈保守性〉という品質に関して、まだ問題があるソースコードであるといえます。

15.6 ソースコード④

3 箇所は、独立したメソッド outputProduct へと統合されることにより、〈モジュール強度〉が高くなっています。これにより、出力書式に修正が生じても作業は 1 箇所に集中され、〈保守性〉が向上しています。

```
public class Test {

    public static void main(String args[]) {

        Calculator c = new Calculator();

        c.operation();

    }
}

class Calculator {

    public void operation() {

        // 変数を導入する

        double taxRate = 0.05;

        // 一つ目の商品

        String name = "XPS 9100";

        int price = 150000;

        int taxedPrice = getTaxedPrice(taxRate, price);

        outputProduct(name, taxedPrice);
```

```
        // 二つ目の商品
        name = "Inspiron 780";
        price = 108000;
        taxedPrice = getTaxedPrice(taxRate, price);
        outputProduct(name, taxedPrice);
        // 三つ目の商品
        name = "Dimension R400";
        price = 123000;
        taxedPrice = getTaxedPrice(taxRate, price);
        outputProduct(name, taxedPrice);
    }
    private void outputProduct(String name, int taxedPrice) {
        System.out.println(name + "/税込価格/" + taxedPrice);
    }
    private int getTaxedPrice(double taxRate, int price) {
        return (int) (price + price * taxRate);
    }
}
```

実行結果

```
XPS 9100/税込価格/157500
```

Inspiron 780/税込価格/113400
Dimension R400/税込価格/129150

この状態でもまだ問題があります。operation メソッドの内部で、税込価格を計算し商品名とともに出力するという同じ意味の処理を3回実行しています。これも、同じ意味の処理がバラバラに存在していると考えられます。また、このときはたまたま3つの商品の計算でよかったかもしれませんが、処理する商品の数が固定されているため、商品数に変更があった場合は、そのつど修正が必要になります。しかも、商品数が多い場合にはコーディング作業量が増加します。また、商品情報を出力する形式に変更が合った場合は、3箇所を修正する必要があります。

15.7 ソースコード⑤

3回実行されていた同じ意味の処理が、独立したメソッド doOperation へと統合されることにより、〈モジュール強度〉が高くなっています。これにより、税込価格を計算し、一定の形式で出力するという手順が一箇所にまとめられ、修正しやすくなっています。また、商品の数が増加しても容易に対応することが可能です。

```
public class Test {

    public static void main(String args[]) {

        Calculator c = new Calculator();

        c.operation();

    }
}

class Calculator {

    public void operation() {

        // 変数を導入する

        double taxRate = 0.05;

        // 一つ目の商品

        doOperation("XPS 9100", 150000, taxRate);

        // 二つ目の商品

        doOperation("Inspiron 780", 108000, taxRate);

        // 三つ目の商品
```

```
        doOperation("Dimension R400", 123000, taxRate);

    }
```

```
    private void doOperation(String name, int price, double taxRate) {

        int taxedPrice = getTaxedPrice(taxRate, price);

        outputProduct(name, taxedPrice);

    }

    private void outputProduct(String name, int taxedPrice) {
        System.out.println(name + "/税込価格/" + taxedPrice);
    }

    private int getTaxedPrice(double taxRate, int price) {
        return (int) (price + price * taxRate);
    }
}
```

実行結果

```
XPS 9100/税込価格/157500
Inspiron 780/税込価格/113400
Dimension R400/税込価格/129150
```

15.1 CMU/SEI アーキテクチャ手法群におけるシステム品質特性

本節ではこれまで、ソフトウェアの品質に関して、最も一般的な定義を示す ISO/IEC 9126-1:2001 (JIS X 0129-1:2003) によるソフトウェア品質モデルについて触れた。ソフトウェア品質モデルは、品質特性について包括的に説明がなされ、体系的に理解する上で大変有用なモデルであるが、他にも様々な品質特性が存在することは忘れてはいけない。ここからは、よりアーキテクチャ設計と強い関

連がある、CMU/SEI アーキテクチャ手法群[46]で述べられる、システム品質特性、そしてそれらの品質特性シナリオについて概説する。

アーキテクチャと品質特性の関連性について、「実践ソフトウェアアーキテクチャ」では、以下のような２つのことが述べられている。

- アーキテクチャは、システムで関心のある多くの品質の実現にとって非常に重要であり、多くの品質はアーキテクチャレベルで設計されなければならず、アーキテクチャレベルで評価できるものである。

- アーキテクチャだけでは品質を達成できない。アーキテクチャは、品質を達成するための基礎を提供するが、細部に注意を払われなければこの基礎は無益になる。

（『実践ソフトウェアアーキテクチャ』, pp.97）

これは、品質特性はアーキテクチャの段階から常に注意を払い設計が行われるべきで、ここをないがしろにしてしまうと、幾ら実装の段階で様々な工夫を施したとしても品質特性を満たすことが不可能となってしまうこと、一方、どんなに綿密な設計がなされていても実装においての配慮がなければ、品質特性を満たすことが難しくなること、を述べている。

■ CMU/SEI アーキテクチャ手法群におけるシステム品質特性

CMU/SEI アーキテクチャ手法群では、品質シナリオに基づくアーキテクチャ設計手法(Attribute Driven Design）や、品質シナリオに基づくアーキテクチャ評価手法(Senario-Based Architectural Analysis)、アーキテクチャ設計上の判断/選択/決定の結果を評価する手法(Architecture Trade-off Analysis Method)。品質特性について評価済みのアーキテクチャパターンのカタログ(Attribute-Based Architectural Style)などが論述されており、品質特性シナリオを用いてアーキテクチャを設計することを基盤にした概念が多く存在する。その、手法群において品質特性は以下の５つに分類される。

[46] Carnegie Mellon University/Software Engineering Institute（カーネギメロン大学/ソフトウェアエンジニアリング研究所）により提唱されているアーキテクチャ設計に関する手法の集合。Attribute Driven Design(ADD):品質特性駆動型設計手法、Scenario-Based Architecture Analysis(SAAM):品質シナリオに基づくアーキテクチャ評価手法、Architecture Trade-off Analysis Method(ATAM): アーキテクチャ設計上の判断/選択/決定の評価をする手法、Attribute-Based Architectural Style(ABAS): 品質特性について評価済みのアーキテクチャパターンのカタログ、などから成る。

- 可用性
  障害を遮断あるいは修復し、故障なく動作する能力
- 変更容易性
  要求や環境の変化時に必要な変更コスト・時間が小さい能力
- 性能
  時間的制約を満たす能力
- セキュリティ性
  外部からの攻撃に抵抗しデータ・情報を保護する能力
- テスト容易性
  拡張・修正したソフトウェアのテストによる妥当性確認を容易に出来る能力
- 使いやすさ
  利用者が必要な作業を容易に達成出来る能力

以下でそれぞれの特性をより詳細に取り上げる。

1. 可用性

可用性とは、障害を遮断あるいは修復し、故障なく動作する能力を表す。システムの故障とは、システムがもはや仕様と一致するサービスが供給できなくなった時に発生する。似て非なる概念として障害が存在するが、障害はシステムが予期せぬ動作を起こすことを表し、それが遮断（隠蔽）もしくは修復が出来ず、ユーザーによって観察可能となったものが故障と呼ばれる。可用性のより一般的な定義は以下となる。

$$\alpha = \frac{\textit{平均故障時間}}{\textit{平均故障時間} + \textit{平均修復時間}}$$

ここで、平均故障時間とは故障が発生するまでの平均時間を表し、平均修復時間とは故障が観察されてから、それが修復されシステムが平常稼働を開始するまでにかかる時間を表す。予期された休止期間は例えシステムが通常稼働していない状態でも、可用性の計算には含めない。可用性における共通の関心事は、システムの故障をどのように検出するか、どのくらい頻繁にシステムの故障が発生するか、故障が発生したらなにが起こるか、システムはどのくらい運転休止になるのか、いつ故障が発生すれば安全科、どうすれば故障を防げるか、そして故障が発生した時にどのような通知が必要になるかである。また、一度故障が発生すれば、それを修復するのに必要な時間も含まれる。

2. 変更容易性

変更容易性は、要求や環境の変化時に必要な変更コスト・時間が小さい能力を表す。より厳密に表すと、現状のシステムに某かの変更を行うことが必要となった際に、変更を実現するのにかかる時間とコストを測定し、それらが如何に小さいかが変更容易性を表す。システムの観点から考えると変更とは、システムが実行する機能、システムのプラットフォーム（ハードウェア、オペレーティング・システム、ミドルウェアなど）、システムが動作する環境（相互作用するシステム、外の世界との通信に使われるプロトコルなど））、システムが示す品質（性能、信頼性、そしてさらに将来の拡張性）、およびその能力（支援するユーザーの下図、同時に操作できる数など）などからなる、システムの側面、に追加、削除、あるいは修正を行うことを表す。共通の関心事としては、それら様々なシステムの側面において、何が変更できるのか、またいつ、誰が、行うのか、などが挙げられる。

3. 性能

性能は、時間的制約を満たす能力である。具体的には、イベント（割り込み、メッセージ、ユーザーからの要求、あるいは時間の経過）が発生した際に、システムが応答するのにどれだけ時間がかかるかを表す。性能の品質特性を特徴づける関心事としては、待ち時間（刺激の到達からシステムの応答までの時間）、処理のデッドライン（いつまでに処理が完了しなければならないか）、システムのスループット（単位時間辺りにシステムが処理出来るトランザクションの数）、応答の待ち時間のばらつき、などが挙げられる。これらは評価が定量的に行える要素であり、性能は容易に測定出来ると考えられがちだが、実際にはイベントの発生源と到達のパターンが相当数あるため、統一的な指標で測定を行うことは困難である。これらを、出来る限り統一的に測定するため、イベントの到達パターンを定期的、確率的の2つに特徴づけ、それらを変動または組み合わせることでモデル化を行い、可能な限りイベント発生源の違いを飲み込むことで複雑さを回避する方法が取り入れられることも多い。一方、実際にはイベントは離散的（すなわち定期的、確率的のいずれでも捉えられないパターン）に発生することもあり、必ずしも上述のモデル化で全てのイベント到達パターンを考慮出来るわけではないことには留意が必要である。

4. セキュリティ性

正当な利用者にサービスを提供しながら、不正利用などの攻撃に抵抗しデータ・情報を保護する能力を表す。ここで攻撃とは、セキュリティを突破し、データやサービスにアクセスすることを目的とするものや、データを改ざんしようとするもの、システムに障害・故障を起こし、サービスを停止させることを目的とするものなどが挙げられる。
このセキュリティ性は、システムが提供する阻害防止、機密性、完全性、保証、可用性、監査の6つの要素によって特徴づけられる。

- 阻害防止
  データやサービスへのアクセス、あるいは変更が、その関係者の誰からも妨害されることがないという特性。
- 機密性
  データまたはサービスが不正アクセスから保護されるという特性。
- 完全性
  データまたはサービスが、意図通りに届けられるという特性。
- 保証
  取引の当事者が、彼らがそうであると表明した人と一致するという特性。
- 可用性
  システムが正規の用途に利用可能であるという特性。
- 監査
  システムを復元するのに十分なレベルで活動を追跡するという特性。

5. テスト容易性

拡張・修正したソフトウェアのテストによる妥当性確認を容易に出来る能力を表す。厳密には、拡張・修正されたソフトウェアに少なくとも1つの欠陥があると仮定して、次のテストの実行でその欠陥が見つかる確率について言及する。この確率を計算することは非常に難しく、またテストは様々な開発者、テスト担当者、検証実験者、あるいはユーザーなど様々なステークホルダーによって行われ、更にソフトウェアライフサイクルの中の様々な工程で行われるため、厳密な意味でこれらを定量的に測定する方法はない。必要なカバレッジレベルを実施するのにどの程度時間がかかるかなどを扱う事が多い。

6. 使いやすさ

利用者が必要な作業を容易に達成出来る能力を表す。この特性は、システムが提供するユーザー支援の種類に関係があり、それは以下の領域に分類される。

- システム機能の学習
  ユーザーが未知のシステムに取り組む際に、学習作業をより簡単にするための支援。
- システムの効率的な利用
  ユーザーが効率的に操作出来るようにするための支援。
- エラーの影響の最小化
  ユーザーエラーの影響を最小化するための支援
- ユーザーニーズへの適合
  ユーザーあるいはシステム自身が、ユーザーの作業をより簡単に行えるようにするための支援
- 自信と満足の向上
  ユーザーが操作を正しく行っているという自信を与えるための支援

15.2 品質特性シナリオ

前項では、CMU/SEI アーキテクチャ手法群におけるシステム品質特性について詳説した。要件定義の中で発見される、非機能要件のうち、この品質特性に関する要求である品質要件を表現する1つの方法として、シナリオを用いる方法が挙げられる(品質特性シナリオ)。本項では、要件定義の成果物から、品質特性シナリオを生成出来るようになる事を目標に、シナリオとはどういったものかを概説した後に、先程挙げた6つの品質特性に関する品質特性の一般シナリオ(システムに非依存でどのようなシステムにも適用出来る一般的なシナリオ)を紹介する。

■ シナリオとは

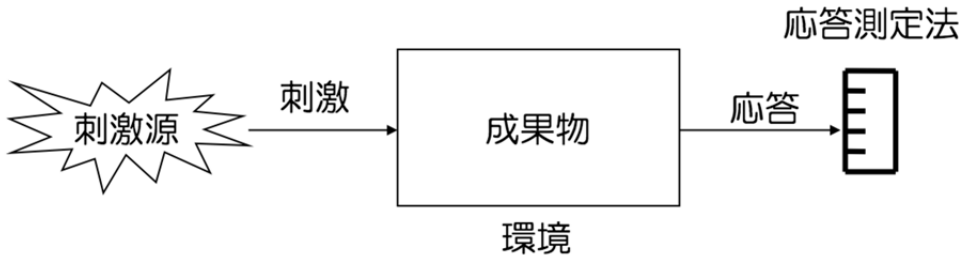
シナリオとは、要求を具体的なストーリーとして記述したもので、利害関係者の意図/興味の表現や、機能要件/品質要件の表現など、様々な用途で使われる。考慮されるべき対象を元に利用シナリオ(通常の利用方法を対象とするシナリオ)、成長シナリオ(将来起こりえる成長の予測を対象とするシナリオ)、調査シナリオ(問題の予測を対象とするシナリオ)などに分類される他、別の切り口として、システムに非依存でどんなシステムにも適用出来る一般シナリオと、検討中の特定システムに

固有の具象シナリオに分類される。

シナリオは構成要素としては以下の6要素を持つ。（例示は全て変更容易性に関する具体例）

- 刺激
  成果物に達した場合になにがしかの応答を必要とするイベントや状況
  例）システムへの変更要求（機能追加、削除、変更など）
- 刺激源
  刺激を生み出す人、システム、装置など
  例）システム変更要求を生み出すステークホルダー（ユーザー、プロジェクトマネージャー、顧客など）
- 環境
  刺激が発生、または刺激が成果物に到達する際の状況
  例）システムがライフサイクルのうちのどの工程にいるか（設計中、開発中、テスト中、運用中など）
- 成果物
  刺激を受ける対象、客体
  シナリオにおいて記述される、刺激や環境、応答などの要件・特性を満たすように作成される要素と言う意味で「成果物」と呼ばれる。
  例）変更されるシステムやシステムの機能、モジュールなど
- 応答
  刺激が成果物に到達した際に開始されるべき活動、または起こるべき状況
  例）変更を行った際にかかる工数・コスト・変更の影響の範囲などがどの程度で抑えられるべきか（〇〇と言う機能を削除する際に3人日以内で変更を行えるべき、××と言う機能を変更する際に影響を受けるモジュールは3つ以内であるべき、など）
- 応答測定法
  刺激が成果物に到達し、応答が行われた際、その応答をどのように観測・測定するかの規定
  例）変更を行う際の工数・コスト・変更の影響の範囲などを（可能ならば）定量的に測る為の方法（プロジェクトトラッキングツール、人手による測定、など）

それぞれの関係を以下の図に示す。



「刺激源」から「刺激」が発生し、それがシステムに到達した際に、「成果物」がある一定の「環境」下において、どのような「応答」をすべきか、そしてその「応答」はどのように測定されるべき(「応答測定法」)であるかを表すのが1つのシナリオとなる。
抽象的でややわかりづらい為、可用性の一般シナリオを先取りし、例として文書化してみる。可用性の一般シナリオにおいて、「刺激」はシステムの内部/外部で発生した障害となる。「刺激源」は障害を発生させるシステムの内部または外部の要素と言う事になる。「成果物」である、システムのプロセッサ、通信チャネル、プロセス、記憶装置、など障害を受け取る可能性のあるシステムの要素が、システムの状態という「環境」の元、障害の記憶を行う、管理者に通知をする、などと言う「応答」を行う。そして、その「応答」を測定するのが人間または他のシステム(「応答測定法」)。と言う所を規定するまでが1つの「シナリオ」となる。
誤解を恐れずにシンプルに要約すれば、「刺激」と呼ばれる事柄・イベントが起こった時・到達した時に、「成果物」と呼ばれるシステムにおけるある要素が、どのように「応答」すべきか・出来るべきかと言う、機能要件または非機能要件を、「刺激源」、「刺激」、「環境」、「成果物」、「応答」、「応答測定法」と言うそれぞれの要素を定義する事で明らかにする方法がシナリオである。

■ 品質特性の一般シナリオ

アーキテクチャ設計を検討する際、検討中のシステム固有となる「具象シナリオ」を生成する事が求められる。「具象シナリオ」は多くの場合、システムに非依存な「一般シナリオ」を元に、システム固有の要求や、システムのドメインに特有な言語などを使用して生成される。以下に、実際に具象シナリオを作成する際にテンプレートとして使用する事が出来る、CMU/SEI アーキテクチャ手法群におけるシステム品質特性６つに対する一般シナリオを紹介する。

- 可用性の一般シナリオ

可用性の一般シナリオでは、システムの障害/故障が発生した際に、システムとしてどのような反応をすべきかが関心事となる。応答測定では、先に挙げた可用性の定義に基づき、実際にシステムが稼

働しているべき時間のうち、障害によって、システムが縮退状態もしくは停止状態に陥る時間（修復時間）がどの程度を占めるかを測定する。そのため、刺激である障害が到達した際に、少しでも早く復旧するために必要なアクションが応答となる。以下の表に各シナリオの構成要素と、それぞれにおいて考慮され得る値を記す。

シナリオの構成要素	可能な値
刺激の発生源	システムの内部/システムの外部
刺激	システムの内部、または外部で発生した障害が刺激として捉えられる。発生する障害は以下のいずれかに分類される。 脱落：コンポーネントが、入力に対する応答に失敗する クラッシュ：コンポーネントが、繰り返して脱落障害を受ける タイミング：コンポーネントの応答が、早い、または遅い 応答：コンポーネントが誤った値を応答する
成果物	システムのプロセッサ、通信チャネル、プロセス、記憶装置、など
環境	システムの状態は必要な応答に影響を与える。通常状態であれば単純に通知を行うのみが望ましい応答であるような刺激も、縮退状態で受け取った場合にはシステムのシャットダウンをすることが好ましい場合もある。 よって、環境としては以下の2つの状態を考慮すべきである。 通常状態・縮退状態（機能の限定、フォールバックなど）

応答	システムは刺激到達時（一般的に障害発生時）に、少しでも早い復旧を目指すため、以下のうち1つ以上の応答を実行する。 ・ 記録する ・ ユーザーや他システムなどの適切な関係者へ通知する ・ 欠陥あるいは定義されているルールに従った障害を引き起こしたイベントの源を使用不可能にする ・ 間隔がシステムの重大な問題に依存する場合、あらかじめ定義されている間隔で使用不可能にする ・ 正常あるいは縮退状態で運用を継続する
応答測定	人間または他のシステムによって以下の項目が可用性の能力として測定可能。 ・ システムが使用可能でなければならない時間間隔 ・ 使用可能時間 ・ システムが縮退状態である時間間隔 ・ 修復時間

- 変更容易性の一般シナリオ

変更容易性の一般シナリオでは、ソフトウェアライフサイクルの様々な場面で起こりうる、システムのステークホルダーから要求されるシステムの変更に対して、どのように反応すべきかが関心事となる。具体的なシナリオの例としては、「設計時にシステム管理者から、管理機能を追加して欲しいと言われた際に、2人日以内に完成させる」、などが例となる。一方、このように定量的な形で常に全ての将来起こりうる変更に対して予測が出来るわけではないので、「システムのある部分に変更がある場合、各要素の独立性を高めて影響を受ける機能は変更箇所の子クラスに限定する」、などと定性的な形で指定する場合も多い。

シナリオの構成要素	可能な値

刺激の発生源	エンドユーザー、開発担当者、システム管理者などの、システムのステークホルダー
刺激	刺激の発生源である、システムのステークホルダーからから求められる変更の要求が刺激として捉えられる。変更には、一般的に以下のようなものが考えられる。 ・ 機能の追加 ・ 既存機能の修正 ・ 機能の削除 ・ 品質要件の変更（応答性の改善、可用性の増加、など）
成果物	成果物としては、変更される対象が特定される。システムにおけるあらゆる要素が対象となり得る。具体的には、システムユーザーインタフェース、バックエンドシステム、プラットフォーム、環境、また状況によっては相互運用する別のシステムなども考えられる。
環境	変更がいつ行われるかによって、応答は変わりうる。よって、環境としては、ソフトウェアライフサイクルにおける様々な場面が考慮される。より応答に影響を与えやすいソフトウェアライフサイクルの分割方法としては以下が挙げられる。 ・ 設計時 ・ コンパイル時 ・ ビルド時 ・ 初期化時 ・ 実行時
応答	応答は刺激である変更の要求に対し、以下のような応答が求められる。 ・ 変更すべきアーキテクチャ上の場所を特定する ・ 他の機能に影響を与えることなく変更を実施する ・ 変更をテストする ・ 変更を実施する

応答測定	測定は、一般的に定量的に測定可能な以下の尺度を用いて行われる。 ・ 変更によって影響を受ける要素の数 ・ 変更にかかる作業工数 ・ 変更にかかるコスト 一方、常に定量的な要素が予測可能であるわけではないので、他の機能あるいは品質特性に与える影響の範囲などを定性的に分析することもある。

- 性能の一般シナリオ

性能の一般シナリオでは、システム運用時にシステム内部もしくは外部から到達するイベントに対して、どの程度の素早さ、そして安定性でそれらを処理し、必要に応じてシステムの状態を変更するかが関心事となる。

シナリオの構成要素	可能な値
刺激の発生源	システムの内部/システムの外部

刺激	システムに対するイベントの到達を刺激と捉える。本質的にはシステムに対するイベントは様々な方法で到達するが、全てのイベントの到達方法は以下の3つのいずれかの方法で到達する ・ 周期的なイベントの到達 ・ 確率論的なイベントの到達 ・ 離散的なイベントの到達 イベントの種類に応じて、これらに具体的な到達方法（例えば1秒に30回のアクセスがある、など）に対して応答及び応答測定を定義する。一方、離散的なイベントの到達のみは、その到達方法が具体的には予期出来ず（外部のイベントによって引き起こされる突発的なアクセスなど）、どこまで離散的なイベントに対しシナリオを生成すべきかに関しては、可用性などとの兼ね合いも考慮の上システム毎に検討が必要。
成果物	イベントを受け取るシステム自体が成果物として捉えられる。
環境	刺激である、システム内部及び外部からのイベントに対する適切な応答は、システムの状態に応じて異なる。具体的には以下の3つの状態が考えられる。 ・ 通常状態 ・ 非常状態 ・ 過負荷状態
応答	応答としては、刺激として到達したイベントに対する処理を行うこと、また一定の条件が満たされた場合にシステムの状態を変えることが求められる（予期しないイベントが訪れた際に、一般状態から非常状態に移行する、など）

応答測定	応答測定では、刺激の到達時にイベントの処理及びシステムの状態変更を以下に素早くそして安定して行えるかを測定する。具体的には以下の5つ要素が挙げられる。 ・ 刺激の到達からイベントを処理するまでの待ち時間 ・ 刺激の到達からイベントを処理するまでのデッドライン ・ 刺激の到達からイベントを処理するまでの時間のばらつき ・ 特定の時間間隔の中で処理出来るイベント数 ・ イベントの検出失敗率 / データ損失率

- セキュリティ性の一般シナリオ

セキュリティ性において、刺激は一般的にシステムに対する攻撃と評されるが、この攻撃の定義は大変広範であり、実は通常のユーザーからのアクセスも、身元の識別が出来、正しく権限があることを確認されるまでは、刺激（攻撃）として捉え応答を行う。よって、セキュリティ性の一般シナリオを考慮する際に重要となるのは正規のユーザーからのアクセスを正しく通過させ、不正なユーザーからの攻撃に対し如何に対応するかとなる。

また、正規のユーザーの正しいアクセスであっても、それが極端に多い回数行われる場合には、それがシステムの可用性や性能など他の品質特性を低下させることに繋がる為、必ずしても正しいユーザーをただ通過させることが正しい応答であるとは限らない点には注意が必要。

シナリオの 構成要素	可能な値
刺激の発生源	主に人間または別のシステムから刺激が発生される。さらに、以下のようにそれら刺激の発生源を分類することが効果的であるとされる。 1. 識別の可/不可 ・ 身元を正しく識別されている ・ 身元を誤って識別されている ・ 知られていない身元 2. 権限の有無 ・ システム内部かつ権限が有る ・ システム内部だが権限は無い ・ システム外部だが権限が有る ・ システム外部の上権限も無い 3. アクセス先 ・ 広範なリソース ・ 制限されたリソース
刺激	セキュリティ性において、刺激は、システムのセキュリティを破ろうと試みる攻撃を表す。具体例としては以下のようなものが挙げられる。 ・ システムに関連する情報を変更しようとする試み ・ システムに関連する情報を削除しようとする試み ・ サービスにアクセスしようとする試み ・ システムを障害/故障に追い込もうとする試み
成果物	システムのサービス/システムに関連するデータ

環境	刺激であるシステムに対する攻撃は、システムが以下のどのような状態にあっても起こりうる。出来る限り多くの環境を想定してシナリオを生成することが求められるが、以下の3つの分類は攻撃の目的や方法、そして発生源を特定する上でも重要である。 ・ システムがオンラインであるか、オフラインであるか ・ システムがネットワークに接続されているか、されていないか ・ システムがファイアウォール内にあるか、公開されているか
応答	システムはユーザーを認証し、正規のユーザーに対してデータ/サービスへのアクセスを許可する一方で、不正なユーザーに対し、一般的に以下のような応答を取る。 ・ ユーザーの身元を隠す ・ データ/サービスへのアクセスを拒否する ・ データ/サービスへのアクセスを剥奪する ・ アクセスやデータの変更を記録する
応答測定	セキュリティ性は、刺激（攻撃）の種類が大変広範であり、応答測定を定量的に行うことは難しい。以下のような切り口で測定することが考えられる。 ・ 成功の確率があるセキュリティ対策を迂回する試みの範囲 ・ 攻撃を検出する可能性 ・ アクセスを拒絶した状態の閉鎖性 ・ 攻撃を受けた場合のサービスの回復にかかるコストや工数

- テスト容易性

テスト容易性における関心事は、テストによってシステムのうちどれだけの割合がカバーされたか、テストにおいて欠陥を発見出来る割合など、テスト自体の品質に関するものと、それらテストを作成または実行するのにかかる時間の2つとなる。ソフトウェア開発の各工程においてテスト実施が必要なタイミングに、どの程度の品質のテストを、どれ位の工数やコストをかけて行うかを明らかとする。

シナリオの構成要素	可能な値
刺激の発生源	テスト容易性において刺激の発生源は、刺激の特性から、曖昧となる事が多いが、例としては以下のものが挙げられる。 ・ 要求分析担当者 ・ アーキテクチャ設計者 ・ 単体プログラムの開発担当者 ・ ソフトウェアの統合担当者 ・ システム検証担当者 ・ システムユーザー
刺激	テスト容易性において刺激は、当該システムの開発がソフトウェア開発工程におけるテストが必要とされる段階に進むことを表す。プロジェクトにより様々なタイミングが考えられるが、基本的には以下の6つのタイミングが考えられる。 ・ 要求分析が完了したタイミング ・ アーキテクチャ設計が完了したタイミング ・ クラス設計が完了したタイミング ・ 単体プログラム（サブシステム）が完成したタイミング ・ サブシステムが統合されたタイミング ・ システムが出荷されるタイミング
成果物	テストされる要素を表す。具体的には環境に応じて以下の3つが考えられる。 ・ 設計 ・ コード ・ アプリケーション

環境	テスト容易性は、環境によって成果物も必要な応答も大きく異なる。一般的に以下の環境に分類して考えることが多い。 ・ 設計時 ・ 開発時 ・ コンパイル時 ・ 配置時
応答	テスト容易性における応答は、ソフトウェア開発におけるテストを行う意義とも捉えられる。テストを行う事により、以下の2項目を達成する。 ・ ソフトウェア内部の状態を制御可能とすること ・ テストの応答を観測可能とすること
応答測定	テストによって、システムのどれだけの範囲がカバーされたか、また欠陥などをどの程度発見出来るか、そしてテスト自体にかかるコストや、そのテスト開発を準備するのにかかる時間などを測定する。具体的には以下の5つの項目を抑えて置くと良い。 ・ 実行済みの実行命令の割合（テストカバレッジ） ・ 欠陥がある場合にその欠陥をテストが発見する割合 ・ テストの依存関係の大きさ ・ テスト環境の準備に必要な時間 ・ テスト実行時間

- 使いやすさ

使いやすさの品質特性は、一般的にエンドユーザーの行為によって生じるシステムに対する要求をどのように、またどのレベルで満たすかが関心事となる。刺激の発生源は常にエンドユーザーであり、成果物は常にシステムとなる。

シナリオの構成要素	可能な値
刺激の発生源	エンドユーザー
刺激	使いやすさにおける、刺激はエンドユーザーからのシステムに対する要求となる。多くの要求は、以下の5つの要求からの派生で考えられる。 ・ システム機能を知りたい（学びたい） ・ システムを効率的に利用したい ・ エラーの影響を最小限に抑えたい ・ システムにより適応したい ・ システムを快適に利用したい
成果物	システム
環境	システム上においてエンドユーザーが何らかのアクションを取る際に起こる。

応答	システムはそれぞれの刺激（エンドユーザーからの要求）に対して、以下の応答のうち1つ以上を示す。（各要求以下に記される小項目は具体的な例） ・ システム機能の学習を支援する – ヘルプシステムが状況に応じて応答する – ユーザーになじみやすいインタフェースである ・ システムの効率的な利用を支援する – データ/コマンドなどの集約 – 入力済のデータ/コマンドの再利用 ・ エラーの影響を最小限に抑える – ユーザーエラーの検出及び修正 – 元に戻すなどの作業取り消し機能 ・ システムに適応する – 国際化（多言語化・地域ローカライズ化）機能 – ユーザー毎のカスタマイズ機能 ・ 快適に感じる – ページ遷移や、ボタンアクションなどに軽快にレスポンスする – 現在のユーザー状態を可視化する
応答測定	使いやすさの応答測定は、応答も満たそうとするユーザー要求に応じて多岐に渡ることや、エンドユーザーの満足度、知識の増分など定量的には測れないものを評価しなければならないことが多いため具体的に指標を出すことは困難である。一方で、作業時間、エラー数、解決された問題数、成功した操作の割合など、一部定量的に測れるものもあり、それらを取り入れることでより具体的なシナリオを生成出来る。

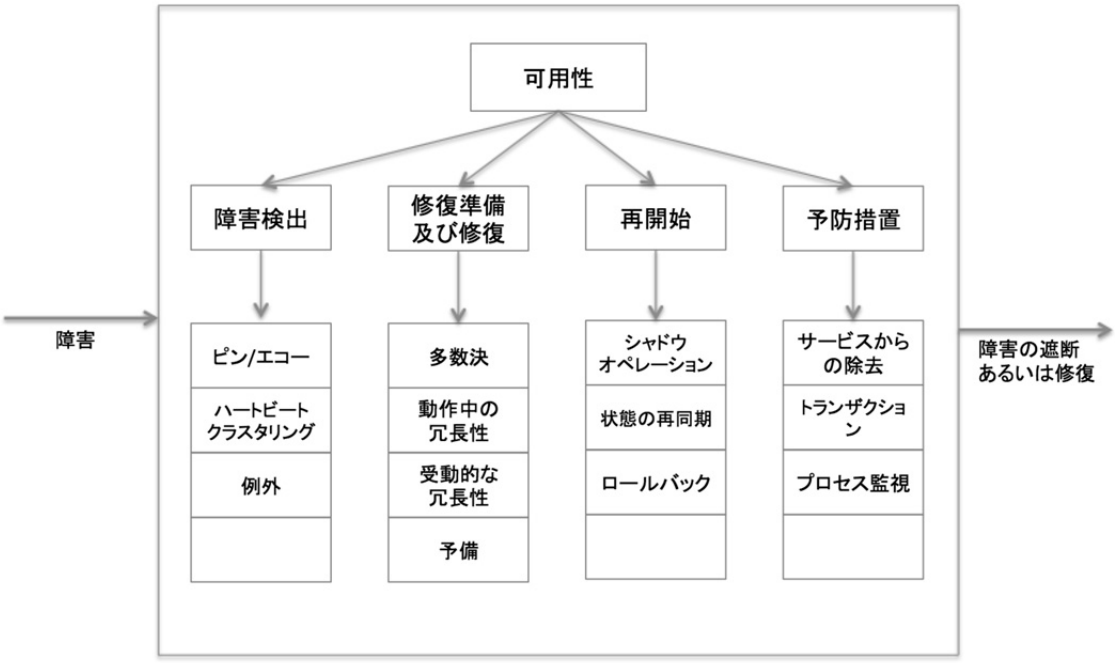
15.3 品質特性の実現手法

ここまで、CMU/SEI アーキテクチャ手法群におけるシステム品質特性について学び、またそれぞれの品質特性における一般シナリオに触れた。実際の開発プロジェクトにおいては、これら一般シナリオを基礎にプロジェクト特有の具象シナリオを生成し、具象シナリオを元にアーキテクチャ設計がなされていく。具象シナリオを実際にアーキテクチャ設計における決定へと結びつけるために必要なのが、品質特性を実際に達成するための実現手法である。

実現手法は、単独である品質特性を実現するものもあれば、複数を組み合わせることでより力を発揮するもの、ある品質特性にとっては良い影響を与える一方で、別の品質特性に悪影響を与えてしまうもの、など多種多様である。ここで全てを紹介することは叶わないが、先に紹介したそれぞれの品質特性それぞれに関連する実現手法を体系的に学び、理解を深めることで、実際のプロジェクトにおいて具象シナリオからアーキテクチャ上の決定を行う際の素地となる力を身に着けて頂きたい。

■　可用性の実現手法

先述の通り、可用性とは、障害を遮断あるいは修復し、故障なく動作する能力を表す。故障は障害が遮断あるいは修復出来ず、実際にシステムのユーザーに観測可能となった場合に発生する。よって、可用性を実現するためには、障害の発生を抑制すること、障害が遮断し故障となることを防ぐこと、障害の影響を抑制し修正すること、などの方法が挙げられる。以下の図に可用性の実現手法の全体像を示す。



- 障害検出

✧ ピン/エコー

相互に責任がある２つ以上のコンポーネント間の障害検出方法として利用される。
あるコンポーネントから検査対象のコンポーネントに対しピンを発行し、想定される時間内にエコーが返ってくることで、検査対象のコンポーネントの状態を確認する。また、ネットワークを介して接続を行っているコンポーネントを検査する場合、コンポーネント自身のみではなく、そのコンポーネントに至るまでの通信経路が、想定通りに機能していることを確認するためにも用いられる。
この障害検出方法は階層化を行うことが出来る。検出器と、１つまたは複数の検出対象のコンポーネントが１つの層を構成し、検出器は同じ層の検出対象コンポーネントにピンを打つことで、障害検出を行い、高い層の検出器は同じ層の検出対象コンポーネントと同時に、低い層の検出器に対してもピンを打つ。検出器は、自分と同じ層にあるコンポーネントで障害が検出された場合、自分よりも高い層の検出器からのピンにエコーを返さない（障害であることを伝える）ことで、すべてのプロセスにピンを打つよりも、使用する効率よくシステム全体の障害検出を行う事が出来る。

✧ ハートビート

ピン/エコーと同様に、相互に責任がある２つ以上のコンポーネント間の障害検出方法として利用される。検査対象のコンポーネントが、定期的にハートビートメッセージを発信し、それを別のコンポーネントが受け取ると言う構成を取る。通常稼働状態では、ハートビートを定期的に受信することを予期するが、設定された閾値以上の期間ハートビートメッセージを受信しない際に、発信元のコンポーネントに障害が起きたとみなすことで障害検出を行う。また、ハートビートはピン/エコーとは異なり、データを運ぶことも可能である。たとえば、定期的に発生することが想定されるアクセスに関するログをハートビートとして使用することで、生存確認と同時に処理されるべきデータを送付することが出来、通信の効率をあげる事が出来る。

✧ 例外

ピン/エコーやハートビートが複数のコンポーネントにおいて、お互い（もしくは一方的に）検査することで自分自身ではないコンポーネントの障害を検出していたが、例外は、１つのコンポーネント（もしくはプロセス）の中で障害を検出する。障害が発生した際に例外を発行させるように設計を行い、それを同一プロセス内に用意した例外ハンドラで処理させることで、障害に対し意味の変換を行い、処理出来る形式にする。例外ハンドラでは一般的に、システム運用者への通知を行ったり、システム利用者に障害が発生した旨を伝えたり、エラーログを書き出したり、障害が発生した際にその被害を最小限に留めるために必要なことが実施される。

➢ 障害修復

システムにおいて障害が発生した際に、迅速に修復を行う方法として、同様の動作をするコンポーネントを並列に配備することで、システムの冗長化を行うことがあげられる。１つのコンポーネントにおいて障害が発生しても、別のコンポーネントがその役割を代替することで、システムの故障時間を限りなく短くすることが出来る。冗長化の方法として「投票」、「動作中の冗長性」、「受動的な冗長性」、「予備」などがあげられる。以下にそれぞれの特徴を記す。

✧ 投票

システムを冗長化し、複数の同等な出力を返すことが想定されるプロセスを動作させる。実際の処理を行う際は、各プロセスに同等の入力を与えそれぞれの計算結果を、投票者と呼ばれるプロセスが受け取る。投票者は、決められたアルゴリズムに従い、

逸脱した振る舞い（障害）を検出し、それを落選させる。
投票のアルゴリズムには、「多数決」や「推奨コンポーネント」などが存在する。また、複数のプロセス全て全く同等のアルゴリズムを使う場合や、ほぼ同等の答えを返すことが想定される別のアルゴリズムを用いる場合が考えられる。前者の場合は、プロセスの動作するプロセッサの障害は検出出来るが、アルゴリズムの障害は検出が出来ない。一方、後者はアルゴリズムの障害も多くの場合検出可能であるが、アルゴリズムを複数開発及び維持することに工数がかかるため、人命が関わるような極端な例のみで使用されることが多い。投票者が単一障害点（その実、投票者も複数に分散することが出来る）になる一方で、複数のプロセスから同等の出力を受け取ることが想定されるため、そのうち幾つかのプロセスにおいて障害が発生したとしても、投票者が動き続ける限りはプロセスにおける故障発生時の停止時間がない。

✧ 動作中の冗長性

並列に応答する、冗長な複数のコンポーネント全てに対し、同等の入力を送り出力結果を受け取るが、そのうち１つのコンポーネントからの応答（通常最初の応答）のみを受け取り、残りを破棄する。複数のコンポーネントが並列に動いているため、例え１つのコンポーネントに障害が発生したとしても、すぐに別の動作中のコンポーネントに切り替えを行うことで、修復時間を数ミリ秒程度に抑えることが出来る。ゆえに、この実現手法は、クライアントサーバーシステムのサーバーサイドなど障害発生時も常に迅速なレスポンスと高い可用性が求められるシステムにおいて用いられる事が多い。一方、並列に並ぶ全てのコンポーネントに計算させるにも関わらず、使用するのはたった１つのコンポーネントからの応答のみで、データやりとりの効率はあまり高くない。

✧ 受動的な冗長性

冗長な複数のコンポーネントが配備される点では、「動作中の冗長性」と同様であるが、平常状態では、１つのコンポーネント（メインコンポーネント）のみが応答を行う。メインコンポーネントは、応答と同時に他のコンポーネント（バックアップコンポーネント）に状態の更新を通知することで正常に応答が出来ていることを伝える。応答を行わないバックアップコンポーネントは基本的には待機し、メインコンポーネントに障害が発生した時に備える。障害が発生した場合、システムは、バックアップコンポーネントが十分に新しいことを確認してから、サービスを再開する。
バックアップコンポーネントを立ち上げるかどうかの判断は、バックアップコンポーネント自体が判断する場合と、別のシステムがそのスイッチングを管理する場合がある。後者の場合は、システムがメインコンポーネントの異常を検知し、スイッチングを行うためシンプルな構成になるが、前

者の場合はバックアップコンポーネントがメインコンポーネントの障害を知る必要があり、「ピン/エコー」や「ハートビート」など他の実現手法を実施して検知する場合や、バックアップコンポーネントもメインコンポーネントと同等の入力を常に受け取り、入力があったにも関わらずメインコンポーネントから閾値以上の時間状態の更新通知が来ない場合に障害があったとみなす、などの方法がある。この実現手法では、バックアップコンポーネントが確実に処理を引き継ぐことに依存しているため、バックアップコンポーネントが十分に新しいことを保証する必要がある。バックアップコンポーネントがアップデートされているかどうかを保証する方法として、定期的にメインコンポーネントとバックアップコンポーネントの切り替えを強制的に行うことで、定期的に動作確認を行うことなどがあげられる。

✧　予備

予備のプラットフォームを用意し、システムにおける様々なコンポーネントと代替出来る構成を準備しておく。ある１つのコンポーネントに障害が検知された場合、予備のプラットフォーム上で適切なソフトウェア構成を立ち上げ初期化を行うことで、障害が起きたコンポーネントと交換し、システムの復旧を行うことができる。通常、この実現手法におけるシステムの停止時間は数分程度であると言われているが、これは「受動的な冗長性」で述べたのと同様で、予備に用意されているソフトウェアなどの構成が、十分に新しいことを保証する必要がある。
コンポーネントは常に初期稼動状態のまま動作している訳ではなく、様々なイベントに応じてシステム稼働中に状態が変化していることが多い。障害発生時にシームレスな復旧を行うためにも、コンポーネントの状態変化を、外部ファイルや、データベースなどに保持し、適切な状態で予備のコンポーネントを立ち上げることが求められる。
予備のプラットフォームは、平常状態では稼働しておらず、平常時のコストが下げられる一方で、「動作中の冗長性」、「受動的な冗長性」などと比べ、復旧までに手作業が求められ、時間がかかる。

上述のように冗長化されたシステムにおいて、一度障害が起きたコンポーネント（及びプラットフォーム）はそのまま停止し続けるのではなく、自動であれ手動であれ、修復され復旧させることで冗長性を高く保つことが求められる。その復旧を安全に行うための実現手法として、「シャドーオペレーション」、「状態の再同期」、「チェックポイント/ロールバック」などが挙げられる。

✧　シャドウオペレーション

修復後、暫くの期間「シャドウモード」と言う、実際のシステム環境には影響を与えない状態で動作をさせて、正常に機能しているコンポーネントと同様の動きをすることを確認してから復旧させる実現手法。複雑なシステムでは、静的な意味で修復が完了した事が確認されても、実際の動作環境と同様の入力を与えた際に、予期せぬ部分が障害の影響を受けており動かないということは多々起こる。シャドウオペレーションを導入することで、それらの問題をある程度防ぐことが出来る。

✧ 状態の再同期

複数の状態を持ちうるシステムにおいて、修復されたコンポーネントの状態を、現状稼働しているコンポーネントと同様の状態まで更新することで、復旧されたコンポーネントが現状稼働のコンポーネントと同様の動きをすることを保証する実現手法。
コンポーネントの取りうる状態はなるべくシンプルである事が望ましく、修復後実際に復旧するまでにかかる時間は、更新の大きさや、更新に必要なメッセージの数などに依存する。

✧ チェックポイント/ロールバック

システムに異常が発生した際、システムに一貫性がない状態が発生する事が多々ある。この状態でシステムが終了してしまっていると、上述の「状態の再同期」を行っても、適切にコンポーネントを復旧出来ない事がある。そうした事態が起こりうる、複雑な状態を持つシステムにおいては、定期的または特定のイベントへの応答として、一貫したシステムの状態を記録（チェックポイント）する。修復されたコンポーネントを一度強制的に最新のチェックポイントの状態で復旧させ、チェックポイント以降の後に発生したトランザクションログを利用して、システムを復元する実現手法。

➢ 障害防止

✧ サービスからの除去

冗長なコンポーネント構成を取るシステムにおいて、何らかの指標が閾値を超えたなど、障害が起こりそうなコンポーネントを除去することで、システム全体に影響を与えるような障害を未然に防ぐこと。これらの判断及びアクションを機械的に行う場合は、それらはアーキテクチャにおいて考慮されるべきである。具体的な例を述べると、「コンポーネント毎のメモリーの値をモニタリングしており、一定の閾値を超えたメモリリークが疑われるコンポーネントを障害が起こる前に、バックアップコンポーネントとスイッチして再起動を行う。」などが考えられる。

✧　トランザクション

一般的に、一度にロールバック可能な一連の手続き処理のまとまりをトランザクションと呼ぶ。可用性における実現手法と言う文脈では、このトランザクションを用い途中で失敗する事が許されないデータの作成/更新/削除が失敗した際に、その一連の処理全てをロールバックする為の設計の事を言う。データベースなどの、永続化データを扱う際に良く用いられる。具体例としては、預金管理を行うコンポーネント群（コンポーネントA，コンポーネントB）において、 1000円をコンポーネントAからコンポーネントBに移す事を考えると分かりやすい。
このストーリーをシンプルに捉えると

1. 1000円をコンポーネントAの預金データから減らす
2. コンポーネントBの預金データに1000円を加える

ことで実現される。一方、これらは途中で失敗出来ない変更で、コンポーネントAのデータベースから1000円を減らす事に成功した後に、コンポーネントBのデータベース更新で某かの障害が起きてしまうと、コンポーネントAは1000円を渡したはずだと考えているにも関わらず、コンポーネントBは受け取っていないと言う非常に危険な事態に陥ってしまう。それを防ぐために、先程の２つの処理をトランザクションとして扱い、もしコンポーネントBの預金データを更新する際に問題が起きた場合は、トランザクションをまとめてロールバックする事で最悪の事態を防ぐ事が出来る。

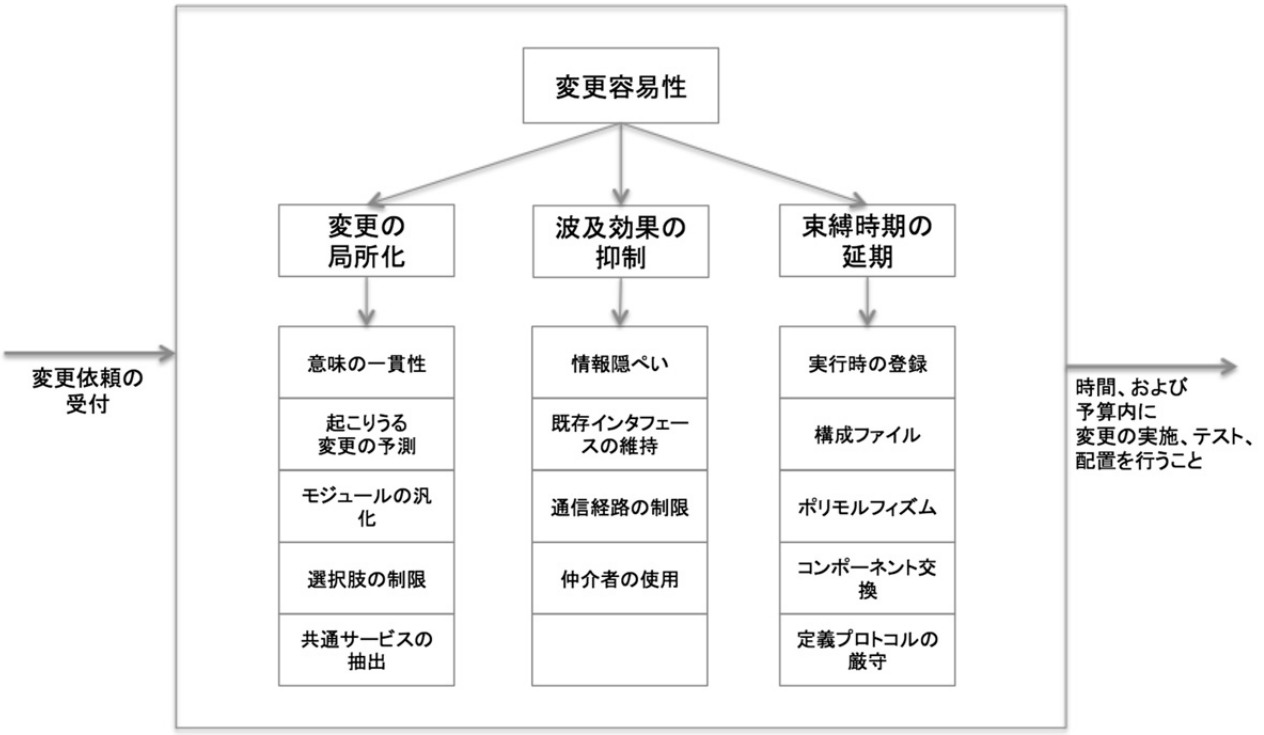
✧　プロセス監視

「サービスからの除去」に大変類似する実現手法、コンポーネント単位ではなくプロセス単位での障害防止を実現する。実働するプロセス（実働プロセス）と、それを監視するプロセス（監視プロセス）と言う構成を取り、実働プロセスにおいて障害が発生した際には、監視プロセスが自動的に機能していないプロセスを除去し、新たなプロセスのインスタンスを起動し、「状態の再同期」や「チェックポイント/ロールバック」などを用い初期化する事で、障害がシステム全体に波及する事を未然に防ぐ。

■　変更容易性の実現手法

変更容易性とは、要求や環境の変化時に必要な変更コスト・時間が小さい能力で

あり、変更容易性の実現手法の目的は、実装、テスト、及び配置の変更にかかる時間とコストを抑制することである。これらの実現には多くの実現手法が存在するが、それらを体系的に分類すると、変更の影響を直接的に受けるモジュールの数を減らす「変更の局所化」、変更によって間接的に影響を受けるモジュールの数を減らす「波及効果の抑制」、そして配置にかかる時間とコストを抑制する「束縛時期の延期」の３つに分類される。以下の図に変更容易性の実現手法の全体像を示す。



- 変更の局所化

  - 意味の一貫性

意味の一貫性を持った責務を1つのモジュールに割当ることで、モジュールの責務が他のモジュールとの間に過度な依存関係を持たず連携する事を保証する。一貫性がない形で責務がモジュールに割り当てられると、ある1つの変更要求に対して、どの部分が変更をなされるべきなのかがわかりづらくなる上に、モジュール間の依存関係が大きいため修正しなければならない範囲も増大する。一貫性を持った責務の割当を行うこと

で変更の範囲を局所化し、コストを逓減することが可能となる。
例）責務1→責務2→責務3→責務4と一連の流れで呼ばれるべき、意味の一貫性をもった責務群を実装する事を考えた時に、それらの責務を
「責務1と責務4をモジュール A に」、「責務2をモジュール B に」、「責務 3 をモジュール C に」実装すると、責務1に変更要求が入り提供するサービス内容（アウトプット）に変更があった場合、モジュール A だけではなく、責務 1 のアウトプットを使う責務 2 や場合によってはそれに続く責務 3 も変更しなければならず、モジュール B, モジュール C にも変更を行う必要が出てきてしまう。よって、このような意味の一貫性を持った責務群は可能な限り限られたモジュール（出来れば1つのモジュール）に割り当てる事で、変更がそのモジュール内で完結する為、変更容易性が向上する。

✧ 起こり得る変更の予測

想定される一連の変更を検討した上で、各変更を実現する際に、修正が必要になるモジュールの数が十分に少なくなる形で責務の割当を行う。「意味の一貫性」が、モジュールに割り当てられる責務の一貫性に関心を持ち、間接的に変更の範囲を局所化することを目指したのに対し、こちらは、起こりうる変更自体を予測し、その影響を直接的に最小化することに関心がある。予想出来る変更に対しては、「意味の一貫性」よりも効率的にコストを逓減出来る一方で、柔軟性が低く予測されなかった変更に対しては効力が少ない。「意味の一貫性」でシステム全体として、ある程度変更容易性を高めた上で、高確率で起こりうる変更に対しては「起こり得る変更の予測」で対応する、など共に用いられる事でより効果を発揮する。
例）同じく掲示板システムを考えると、「将来、投稿済みの内容を編集したい」と言う変更要求が起こる事を予測し、事前に投稿の為のモジュールを柔軟に変更出来るように設計をしておく。

✧ モジュールの汎化

モジュールを一般的な形で設計することで、より広範囲の機能を入力に基いて処理出来るようになる。モジュールが変更要求に対して十分に一般的であれば、モジュールに対する入力を変えることで、モジュール自体の修正を行わずに変更要求を実現出来る可能性が増える。
例）同じく掲示板システムを考えると、「観覧を行う」と言うモジュールをテキスト、画像、動画、など、投稿されたデータタイプに応じて機能を提供するように設計する事で、新

たに「スタンプと言うデータタイプを追加したい」と言う要求が発生した際に、表示部分の機能のみ変更すれば、「観覧を行う」と言うモジュール本体に変更を加えなくても、要求を実現できる。

✧ 選択肢の制限

可変な選択肢が多いほど、1つの変更を実現するために取れる手法が増え、変更にかかるコストを増大させる場合がある。システムの将来性などを予測した上で、可変箇所を予め制限することで、これらのコスト増大を防ぐことに繋がる。

✧ 共通サービスの抽出

システムを構成するモジュール群において、複数のモジュールが共通サービスを提供している場合に、その共通部分を別のモジュールとして切り出し、切り出されたモジュールを呼び出す形を取る。共通部分に対して変更要求が発生した際に、実現手法適用以前では共通部分を包含する全てのモジュールを変更する必要があったのに対し、適用後は切り出された1つのモジュールのみに変更を行えば良いので、変更のコストが逓減される。

➢ 波及効果の抑制

修正の波及効果とは、ある変更要求によって直接影響を受けないモジュールが、変更要求の影響を直接受けるモジュールとの依存関係により、間接的に変更の影響を受けることを言う。波及効果の抑制を図る実現手法を理解する上で、まずはモジュール間の依存関係としてどのようなものが存在するかを把握する事が必要。
ある1つのモジュールへの変更が、他のモジュールに波及効果を与える可能性があるモジュール間の依存関係としては、以下の8つの形態を考慮する必要がある。

1. シンタックス上の依存関係

(ア) データのシンタックス

B を正しくコンパイルする(あるいは、実行する)ためには、A が提供して B が使用するデータの型(または形式)が、B が想定しているデータの型(または形式)と一致していなければならない。

```
class A {
  public double getPrice() {
    return 0.06;
  }
}
class B {
  public static void main(String[] args) {
    A sample = new A();
    double price = sample.getPrice();
  }
}
```

このような状況で、A の getPrice()と言うメソッドが String 型に変わると、B はコンパイル(あるいは、実行)が出来なくなる。このような依存関係をデータのシンタックスにおける依存関係と呼ぶ。

(イ) サービスのシンタックス

B を正しくコンパイルして実行するためには、A が提供して B が呼び出すサービスのシグニチャが、B の想定と一致していなければならない。

```
class A {
  double price;
  public void setPrice(double price) {
    this.price = price;
  }
}
class B {
  public static void main(String[] args) {
    A sample = new A();
    sample.setPrice(0.06);
  }
}
```

このような状況で、A のサービス(setPrice)のシグニチャ(メソッドの名称や、引数の型など)が変更されると、B はコンパイル(あるいは、実行)が出来なくなる。このような依存関係をサービスのシンタックスにおける依存関係と呼ぶ。

2. 意味上の依存関係

(ア) データの意味

Bを正しく実行するためには、Aが提供してBが使用するデータの意味が、Bの想定と一致していなければならない。

```
class A{
  public static final double TAXED_RATE = 0.08;
}
class B{
  public static void main(String[] args) {
    double price = 1000;
    double taxedPrice = price * A.TAXED_RATE;
    System.out.println(“消費税込みの価格は”+taxedPrice+“です”);
  }
}
```

このような状況においてBでは、AのTAXED_RATEで消費税率がかえってくる事を想定しているが、もしAのTAXED_RATEが法人税率に変わってしまうと、Bを正しく実行する事が出来ない。

(イ) サービスの意味

Bを正しく実行するためには、Aが提供してBが使用するサービスの意味が、Bの想定と一致していなければならない。

```
class A{
  public static final double TAXED_RATE = 0.08;
  public double calcTaxedPrice(double price) {
    return price * TAXED_RATE;
  }
}
class B{
  public static void main(String[] args) {
    double price = 1000;
    A sample = new A();
```

```
        double taxedPrice = sample.calcTaxedPrice(price);
        System.out.println(“消費税込みの価格は”+taxedPrice+“です”);
      }
    }
```

このような状況においてBでは、Aのcalc TaxedPriceの結果として、消費税込みの価格がかえってくる事を想定しているが、もしAのcalcTaxedPriceが法人税を計算するサービスに変わってしまうと、Bを正しく実行する事が出来ない。

3. 順序的な依存関係

(ア) データの順序

Bを正しく実行するためには、BはAが発行するデータを決まった順序で受け取らなければならない。たとえば、データパケットのヘッダ情報は、本体の前に受信されなければならない。

(イ) 制御の順番

B を正しく実行するためには、A がある時間制約の中で事前に実現されていなければならない。たとえば、Bを実行する5ミリ秒前までに、Aの実行が完了してなければならない。

4. Aのインタフェースの識別情報への依存

Aは、複数のインタフェースを持つことがある。Bを正しくコンパイル、実行するためには、インタフェースの識別情報(名前、もしくは、ハンドル情報)が、B が想定しているものと一致していなければならない。

```
interface sampleInterface {
  public sample(String test);
}

class A implements sampleInterface {
  public sample(String test) {
    // implementation of sample;
  }
}
```

```
class B {
  public static void main(String[] args) {
    A a = new A();
    a.sample(“Test”);
  }
}
```

このような状態の時に sampleInterface の sample(String test)が変更になり、例えば sample2()となると、それに伴い class A も変更を求められる。その際に class B における a.sample(“Test”)部分がコンパイルが通らなくなる。

5. A の位置への依存

Bを正しく実行するためには、実行時のAの配置された位置がBの想定と一定していなければならない。例えば、同じシステム環境上にある A のサービスを呼び出す際に、Bが同一環境にある事を前提に相対パスで参照をしていた時にこの依存関係が生まれる。並列化などの目的で、別の環境に A が移ってしまった場合に B は A を呼び出すことが出来なくなってしまう。

6. A が提供するサービス/データの質

B を正しく実行するためには、A が提供するデータやサービスの質に関連する特性が、Bの想定と一致していなければならない。例えば、B は A が小数点 8 桁まで正確に計算を返してくれる事を想定して、機能が実装されているとして、A が提供するサービスの質を落とし小数点 4 桁までしか計算しなくなった場合に B の実行に問題が起きてしまう場合がある。

7. A の存在への依存

B を正しく実行するためには、A が存在していなければならない。たとえば、B がオブジェクト A のサービスを要求していて、かつ A が存在していなくて動的に生成することもできないならば、B は正しく実行出来ない。

8. A のリソースの振る舞いに依存

B を正しく実行するためには、A のリソースの振る舞いが、B の想定と一致しなければならない。これは、A のリソース用法(たとえば、A が B と同じメモリを利用する),またはリソースの所有権(たとえば、A が所有していると信じているリソースを B が予約している)のいずれかである。

(他のモジュールに波及効果を与える可能性があるモジュール間の依存関係8項目は『実践ソフトウェアアーキテクチャ』, pp. 142 - 144 より抜粋、説明を一部要約)

これら、依存関係の形態を理解した上で、特定の種類の波及効果を防ぐための実現手法について議論する。特定のモジュールのインタフェースに関する実現手法である「情報の隠蔽」、「既存インタフェースの維持」を議論した後に、依存関係の連鎖を断ち切る実現手法である「通信経路の制限」、「仲介者の使用」を取り上げる。

- 情報の隠蔽

各モジュールの構成要素の責務を更に小さい単位に分割し、パブリックにアクセス可能な要素と、それ自身からのみ呼び出す事が出来るプライベートな要素に分けて設計を行う。基本的に、他のモジュールとの依存関係がある部分はパブリックにアクセス可能な要素のみに制限されるため、プライベートな要素に起こる変更はモジュール内に閉じ込められ、波及効果を防ぐ事が出来る。また、モジュール間の依存関係がある部分を特定しやすくなると言う副次効果も期待できる。

```
class Dog {
  private String dogType;
  private int weight;
  private int height;
  public String getDogType() {
    return dogType;
  }
  private void setDogType() {
    if (weight > 10) {
      if (height > 50 ) {
        dogType = “Golden Retriever”;
```

```
    } else {
      …
    }
    …
  }
 }
}
```

一般的な、オブジェクト指向プログラミングにおけるカプセル化の概念と同様。例えば上記の例で dogType を設定するロジックに変更があったとしても、Dog クラスの外側から、getDogType を参照するモジュールには変更の影響はない。

- 既存インタフェースの維持

あるモジュールが、変更を行う必要があるモジュールのインタフェースの名前及びシグニチャに依存している場合、そのインタフェースを維持することで、波及効果を防ぐ事が出来る。ここで言うインタフェースとは、モジュールが別のモジュールとやりとりをする際のアクセスポイントの形式を表し、そのアクセスポイントからやり取りされる情報など、アクセスポイントの意味に当たる部分は含まれない。やり取りされる情報が変われば、多くの場合情報の受け取り手となるモジュールにも変更が必要となる。
既存インタフェースを維持する方法として、以下の3つのものがある。

1. インタフェースの追加
   あるモジュールが変更要求によって、今までと同様のサービスやデータの提供を維持する事が困難になった場合、多重インタフェースなどの技術を利用し、同様の役割をする新しいインタフェースを同じシグニチャで提供することで、変更されるモジュールのインタフェースに依存関係を持っていたモジュール群への波及効果を抑える事が出来る。

```
class A {
  public int calc(int a, int b) {
    return a * b;
  }
}
class B {
```

```
  public static void main(String[] args) {
    A sample = new A();
    int result = sample.calc(3, 1);
  }
}
```

上記のようにBがAのcalcと言うサービスを呼び出している際に、
calcを以下のように変更しなければならなくなった際

```
public int calc(int a, int b, int c) {
  return a * b + c;
}
```

以下のようなメソッドを作成する事でBにおける変更は呼び出しのメソッド名をcalcからorigCalcに変えるだけの変更となる。

```
public int origCalc(int a, int b) {
  return a * b;
}
```

2. アダプタの追加

あるモジュールが変更要求によって、今までと同様のサービスやデータの提供を維持する事が困難になった場合、アダプタと言う、変更後のモジュールのインタフェースと接続し、外部には変更前のモジュールと同様のインタフェースを提供する、ラッパーのようなモジュールを開発し、そのアダプタを通じて今までと同様のインタフェースを提供することで、依存関係を持っていたモジュール群への波及効果を抑える事が出来る。

```
class A {
  public int calc(int a, int b) {
    return a * b;
  }
}
class B {
  public static void main(String[] args) {
    A sample = new A();
    int result = sample.calc(3, 1);
  }
}
```

上記のようにBがAのcalcと言うサービスを呼び出している際に、

calcを以下のように変更しなければならなくなった際

```
public int calc(int a, int b, int c) {
  return a * b + c;
}
```

以下のようなアダプタを作成する事でBにおける変更は呼び出しのメソッド名をcalcからcalcAdapterに変えるだけの変更となる。

```
public int calcAdapter(int a, int b) {
  return calc(a, b, 0);
}
```

3. スタブ[47]

あるモジュールが変更要求によって削除される際、そのモジュールのインタフェースのみを持つ空のスタブを作成することで、削除されるモジュールのインタフェースのシグニチャのみに依存関係を持つモジュール群への波及効果を抑える事が出来る。

```
class A {
  public int calc(int a, int b) {
    return a * b;
  }
}
class B {
  public static void main(String[] args) {
    A sample = new A();
    int result = sample.calc(3, 1);
  }
}
```

上記のような状況において、某かの理由でAからcalcメソッドを削除しなければならなくなった際、スタブとして以下のような空のメソッドを用意する事で、波及効果を抑える。インタフェースの存在にのみ依存関係を持つ場合にのみ有効。

```
public int calc(int a, int b) {
  return 0;
}
```

[47]スタブ (stub) とは、コンピュータプログラムのモジュールをテストする際、そのモジュールが呼び出す下位モジュールの代わりに用いる代用品のこと。 下位モジュールが未完成でも代わりにスタブを用いることでテストが可能になる。

- 通信経路の制限

ある1つのモジュールに注目した時、そのモジュールが提供するデータを使用するモジュール（アウトプット先）の数と、そのモジュールが必要とするデータを提供するモジュール（インプット先）の数を可能な限り制限する事で、変更が起きた際の波及効果を抑える事が出来る。「意味の一貫性」などとも通ずるが、同じような役割を持つモジュールが複数絡み合う状態は、システムの複雑性が増し、変更容易性を逓減する事に繋がる。なるべく複雑な依存関係を避けるためにも、単純にやり取りをしなければ行けないインプット/アウトプット先のモジュール数減らす事が有効。

- 仲介者の使用

2つのモジュール間に「意味上の依存関係」がある場合、その間に仲介者を置くことで、依存関係に関連する活動を管理し、片側のモジュールに変更が起きた際に、反対側のモジュールにも変更の波及効果が及ぶ可能性を減らす事が出来る。仲介者の形式は、モジュール間の依存関係の種類に応じて変化する。例えばデータのシンタックスに依存関係を持つコンポーネントの場合、リポジトリを仲介者として配置する事で、データ提供側のコンポーネントに変更があっても、その変更をレポジトリが覆い隠す事が出来る。また、あるコンポーネントのネットワーク上の位置に、複数のコンポーネントが依存関係を持つ場合、ネームサーバーを仲介者として用いる事で、コンポーネントの位置が変更されたとしても、ネームサーバーにのみ現在の位置を伝える事で、依存関係を持つ複数のコンポーネントは、ネームサーバーからコンポーネントの新たな位置を知る事が出来、問題なく接続を行う事が出来る。

束縛時期の遅延

所謂、遅延バインディングまたは実行時バインディングと呼ばれる方法で、モジュールの柔軟性を高める実現手法。
ここで、バインディングとはオブジェクトがオブジェクト変数に代入されるときの処理を表し、その方法の違いにより事前バインディングと、遅延バインディングに分けられる。
事前バインディングとは、オブジェクトが事前（コンパイル時）にオブジェクト変数にバインドされる事。特定のオブジェクト型として宣言された変数に代入される場合に起こる。
遅延バインディングとは、オブジェクトが実行時にオブジェクト変数にバインドされる事。
この実現方法として、実行時の登録、構成ファイル、ポリモルフィズム、コンポーネント

交換、定義プロトコルの厳守などが存在する。

■ 性能の実現手法

性能とは、イベント（割り込み、メッセージ、ユーザーからの要求、あるいは時間の経過）が発生した際に、システムが応答するのにどれだけ時間がかかるかを表す。故に、性能の実現手法の目的は、システムに到達するイベントに対して何らかの時間制約の中で応答を生成する事となる。
イベントが到着後の応答時間を決定づける要素はリソース消費とブロック時間の二項目である。

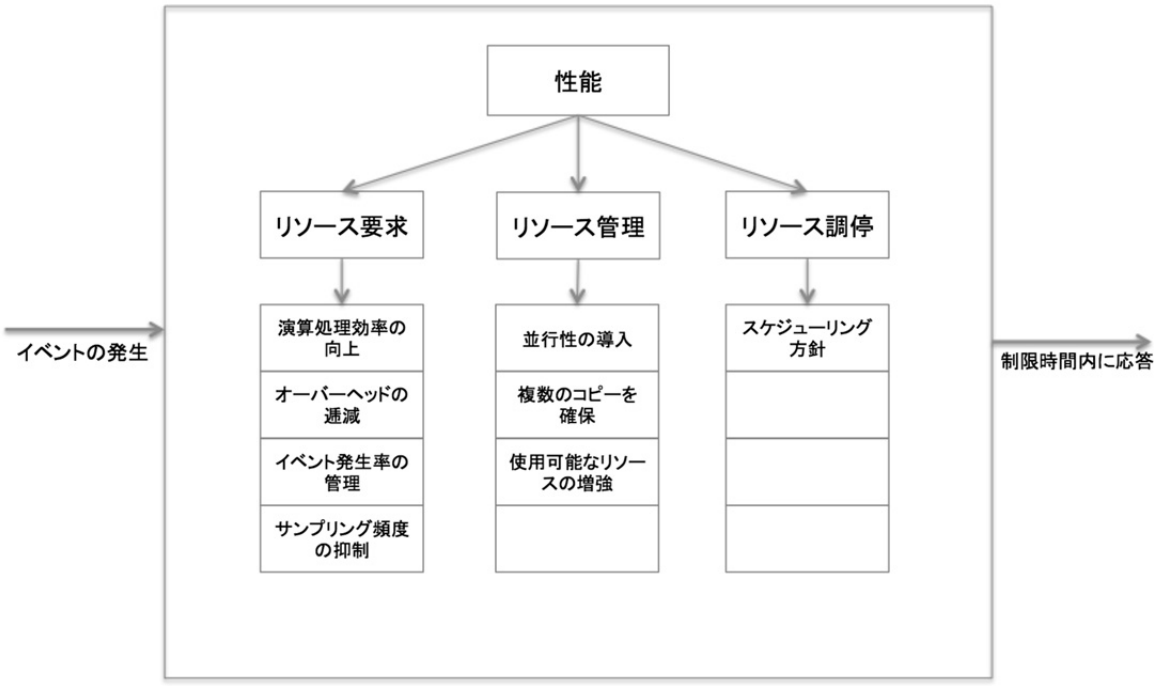
➢ リソース消費

イベントが到着すると、システムはリソース（CPU、データ記憶装置、ネットワーク通信帯域、またはメモリー）を使用してそのイベントを処理する。並列に並ぶリソースを仮想的に１つの処理の１フェーズと捉えると、イベントの処理は各フェーズにおいて逐次的に行われる。よって１つのフェーズでイベントの処理にかかる時間の合計が、イベントの処理全体の待ち時間となる。並列化可能な処理を並列化する事や、各フェーズでの処理時間を短くする為に、リソースの性能を挙げるなどをする事が、高速化に繋がる。

➢ ブロック時間

イベントは到着後、リソースの競合が原因で処理がブロックされる事や、並列で行われている依存関係の処理の結果を待つために明示的に処理をブロックする事がある。当然ブロックが多いほど処理全体にかかる時間が増えてしまうため、それぞれのリソースの可用性を高める事、他の処理への依存性を減らす事で、ブロック時間を短くする事がシステム全体の性能を高める。

応答時間を短縮するための実現手法には大きく分けて、リソース消費にかかる時間を短縮する方法と、ブロック時間を短くする方法がある。性能の実現手法の全体像を以下の図に示す。



- リソース要求

「リソース要求」にて列挙する4つの実現手法は、イベントを処理するのに必要なリソースを減らす事で、応答時間の短縮を図ろうとするものである。

- 演算処理効率の向上

システムがイベントを処理する際、必ず某かのアルゴリズムが適用される。イベントの処理において特に時間がかかっている部分のアルゴリズムを見直す事で、処理時間を短縮する事が出来る。このアルゴリズムの見直しによる演算処理の向上は、ブロック時間の短縮にも貢献する事がある。ボトルネックとなっているリソースを利用するアルゴリズムを、別のリソースを使用するアルゴリズムに変える事（例：DB ではなく、一部 cache を利用するなど）や、並列化可能な形にアルゴリズムを置き換える事でこれらが実現される。

- オーバーヘッドの逓減

様々なりソースへのアクセスが演算処理のオーバーヘッドとなる事がある。システムがイベントを処理する際に、不要なリソースへのアクセスをなくす事で処理効率が向上する事がある。例えば、サーバークライアントの方式を取っているシステムにおいて、クライアント側でも計算可能な内容を、都度サーバーにリクエストするような仕組みを取り入れている場合、そのリクエストをクライアントでの演算処理に置き換える事によって、ネットワーク通信にかかるオーバーヘッドを抑える事が出来る。一方、処理を分割した方が処理効率の向上に繋がる場合もあるので、常に少ないリソースで処理を行う事が良いわけではない事には注意が必要。

✧ イベント発生率の管理

「演算処理効率の向上」、「オーバーヘッド逓減」は、1つのイベントにかかる処理を減らす事で応答時間の短縮を図る実現手法であったが、イベントの発生率を管理する事でも応答時間の短縮を行う事が出来る。イベントの発生が集中することによるリソース競合が起こると応答時間が増大する事に繋がる。コンポーネント間のやりとりなど、管理可能なイベントはその発生タイミングを分散させる事で、リソース競合などを避ける事が出来、性能の向上に繋がる。

✧ サンプリング頻度の抑制

「イベント発生率の管理」と類似した概念で、環境の変化などを監視する際のサンプリング頻度を、少ない頻度で行える設計を組み、必要最低限のサンプリング頻度を考慮する事が性能の向上に繋がる。

➢ リソース管理

✧ 並行性の導入

異なるイベントを追加的なスレッドを作成するなど、並列に処理することでブロック時間を減少させる事が出来る。処理にかかる時間の大きさなどイベント毎の負荷をある程度推測した上で、スレッドの割当を行う事が並行性を最大限活用する為には重要である。

依存関係があるイベント同士を別のスレッドで処理する際などは、1つのスレッドの処理に遅れが出ると、両方のスレッドをブロックしてしまう事もあるため、システムの管理の複雑さが増してしまう事もある。

- ✧ 複数のコピーを確保

クライアントサーバーパターンのように、本来的には全ての演算処理を中央サーバーが引き受けるようなシステム構成において、クライアントに一部の処理やデータの保持を移譲する事により、中央サーバーにおけるリソースの競合を抑える事が出来る。ただし、複製されたデータは中央サーバーに存在するデータとの同期を保つ事が求められ、データ構成やシステム構成などで保証されなければならない。この同期を行う為の確認が逆に応答時間を遅らせる原因となってしまう事もあるため、「サンプリング頻度の抑制」でも述べたように、最低限必要な頻度のデータ同期は保ちながら、過剰に確認を行う事を控える事が性能向上に寄与する。

- ✧ 使用可能なリソースの増強

単純な解決策ではあるが、より速いプロセッサやメモリ、ネットワークを利用する事は待ち時間を減少させる可能性がある。必ずしも投資対効果が比例的に伸びていくわけではない為、求められるスペックと、かけられるコストのトレードオフを考慮した上での選択が求められる。

- ➢ リソース調停

- ✧ スケジューリング方針

リソースの競合が発生すると、リソースをスケジュールに入れる事が求められる。アーキテクチャ設計において、考えられるリソースの競合に対して、それぞれのリソースに合ったスケジューリング方針を選択する事は性能の向上に繋がる。スケジューリング方針には先入れ先出しのような単純な物から、応答までのデッドラインや、イベントの重要性などに基づくものまで様々な種類がある。以下に一般的なスケジューリング方針を挙げる。

1. 先入れ/先出し(FIFO)

リソースに対するすべての要求を等しく扱い、それらの要求に順番に応える。1 つの要求処理に時間がかかってしまうと、他の優先順位の高い要求があったとしてもその応答まで遅れてしまう。リソースに対する要求の優先順位が全て等しい場合に有効なスケジューリング方針。

2. 固定優先順位スケジューリング

リソースに対する要求の発生源に特定の優先順位を割り当てて、優先順位順にリソース割り当てを行う。高い優先順位の要求に対しては短い応答時間で対応出来るが、優先順位の低い要求は、もし自分よりも優先順位が高い要求が続いてしまうと、長い時間が経っても処理されず放置されてしまう可能性もある。よって、優先順位付けをどのように行うかで、システム全体の処理性能が決まってくる。優先順位付けには一般的に以下の3つの方法がある。

(ア) 意味の重要さ

イベントを生み出すドメインの特性に従って静的に優先順位が割り当てられる。メインフレームシステムなどで使用される事が多く、その際はタスクの起動時間に従い優先順位付けが行われている。

(イ) デッドラインモノトニック

より短いデッドラインのストリームにより他界優先順位を割り当てる。リアルタイム性のデッドラインを持った複数のイベントが考慮される場合に使用される。

(ウ) レートモノトニック

周期的なイベントに対する静的な優先順位割当方式で、より短い周期で発生するイベントに対して高い優先順位を割り当てる。オペレーティングシステムで用いられる事が多い。

3. 動的優先順位スケジューリング

「固定優先順位スケジューリング」が静的に優先順位を割り当てたのに対し、リソースの割当の時点における状況を考慮してスケジューリングを行う。一般的なスケジューリング方針としては以下の 2 つが挙げられる。

(ア) ラウンドロビン

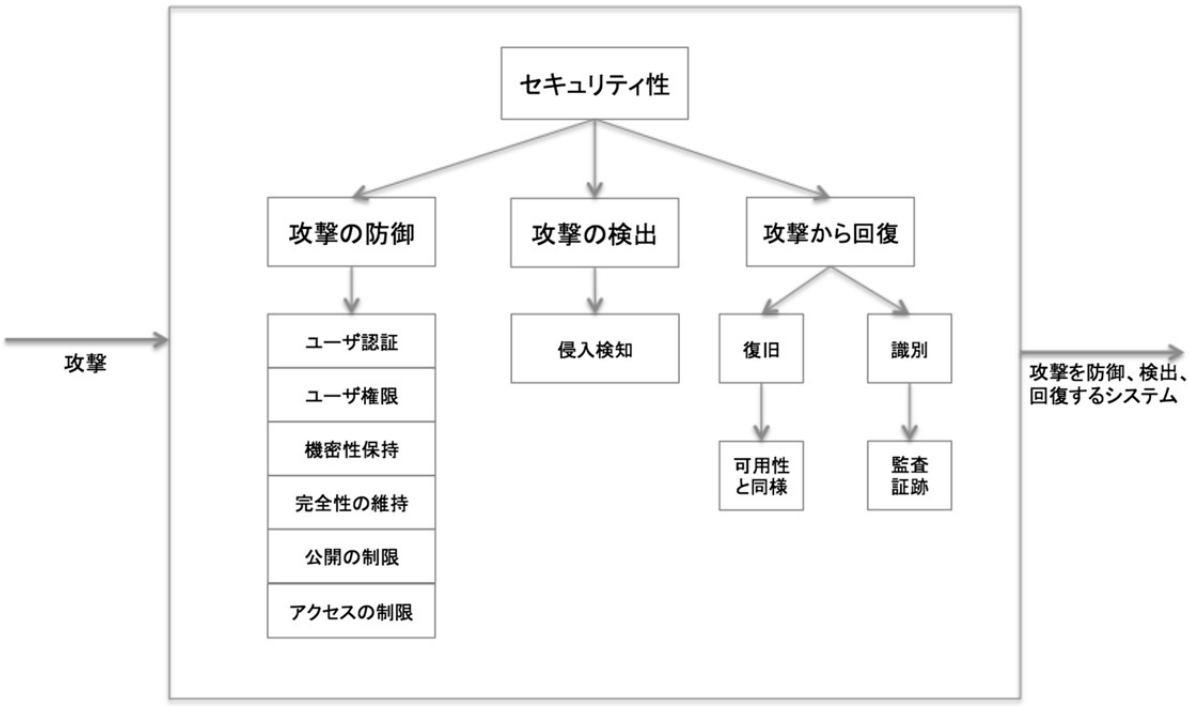
ラウンドロビンは、要求をある規則に基づいて順序付けして並べ、リソースをある決まった時間単位ずつ、全ての要求に、順番に割り当てて行くスケジューリング方針。優先順位が高く、処理にも時間がかかる要求が前にあると、多くのスケジューリング方針で、優先順位が低いが処理時間は短い要求が待たされ、イベントのキューが溜まって行ってしまう事が起こる。ラウンドロビンでは、優先度が低い要求にも、ある程度早期の段階で決まった時間単位のリソースが割り振られる為、キューが 1 つの大きな処理によって完全に詰まってしまう事を防ぐ事が出来る。一方で、本当に優先度が高く、応答時間のデッドラインも短い要求があった際などに、その要求を集中的に処理した場合と比較して、当然応答時間は長くなってしまう。

(イ) 最短デッドライン優先

リソース割り当てが可能になって時点で、保留中の要求の中で最もデッドラインが短いものを基準に、優先順位を割り当てる方式である。

■ セキュリティ性の実現手法

セキュリティ性とは、正当な利用者にサービスを提供しながら、不正利用などの攻撃に抵抗しデータ・情報を保護する能力を表す。セキュリティ性を達成する為の実現手法は、攻撃への抵抗に関するもの、攻撃の検出に関するもの、そして攻撃からの回復に関するものに分類が出来る。以下にセキュリティ性の実現手法の全体像を示す。



- 攻撃の防御

セキュリティ性について議論する際、攻撃とは、セキュリティを突破し、データやサービスにアクセスすることを目的とするものや、データを改ざんしようとするもの、システムに障害・故障を起こし、サービスを停止させることを目的とするものなどが挙げられる。これらからシステムを防御する為に、以下の実現手法を組み合わせて使用する事が求められる。

- ユーザーを認証する

認証とは、ユーザーやリモートコンピュータが、実際にそれを表明しているものと同じであることを保証する事である。認証の方法は技術の進歩や新たな攻撃手段の開発と共に変化を続けており、代表的なものとしてはパスワード認証、ワンタイムパスワード、デジタル証明書、生体認証などが挙げられる。それぞれ単独で使われる事も多いが、組み合わせて使用する事でよりセキュリティ性が高まる。

- ユーザーに権限を与える

認証されたユーザーのみにデータやサービスへのアクセスをする権限を与える事（アクセス

制御）で、予期せぬユーザー（攻撃者）がデータを改ざんしたり、盗用したりする事を防ぐ。アクセス制御は、ユーザー単位、ユーザー種別単位（ユーザーグループ、ユーザーロール、ホワイトリスト、ブラックリストなど）で管理する事が多い。

✧ データの機密性を保持する

データを不正なアクセスなどから保護する為、何らかの形式の暗号をデータそのものや、データをやりとりする通信回線に適用する。データへのアクセスは権限の付与によってある程度管理する事が出来るが、攻撃者によってそのセキュリティが破られてしまった際を考慮し、更にデータ自身を暗号化、簡単には中身を見ることが出来ない状態で保持する事でセキュリティ性が向上する。一方、通信経路はアクセス権限の付与が出来ず、更にデータも復号化された状態でやりとりされる事が多いため、仮想専用ネットワークの利用や、SSL を利用して通信経路自体を暗号化する事が求められる。

✧ 完全性を維持する

データの改ざんなどを防ぐため、チェックサムやハッシュ結果などの符号化した冗長的な情報をデータに持たせる事で、受け取り先での照合を可能とする方法。

✧ 公開を制限する

攻撃は通常システム上にある脆弱性をついて行われる。システムが大きければ大きいほど、全ての脆弱性を取り除く事は困難となる為、公開範囲を必要最低限に制限する事で、攻撃者が攻撃出来得る部分を最小化する事でセキュリティ性の向上を図る。

✧ アクセスを制御する

ファイアウォールのように、メッセージの発信源や送り先のポートに基づいてアクセスを制限する。一方で、ウェブサービスのように一般公開されるシステムにおいては、常に既知のアクセスポイントからのアクセスを期待する事は出来ない為、公開する部分と非公開の部分を明示的に違うネットワーク上に置くなどして管理を行う。

➢ 攻撃の検出

攻撃の検出は、ネットワークにアクセスされるトラフィックパターンを既知の攻撃パターンと比較して行われる。システムの異常の検知はトラフィックパターンを自分自身の過去の基準と比較して行われる。これらを行う為には、通信されるパケットを頻繁にフィルタにかける必要がある。これらの基準としては、プロトコル、TCP フラグ、ペイロード長、発信源や宛先アドレス、ポート番号などが使用される。

- 攻撃からの回復

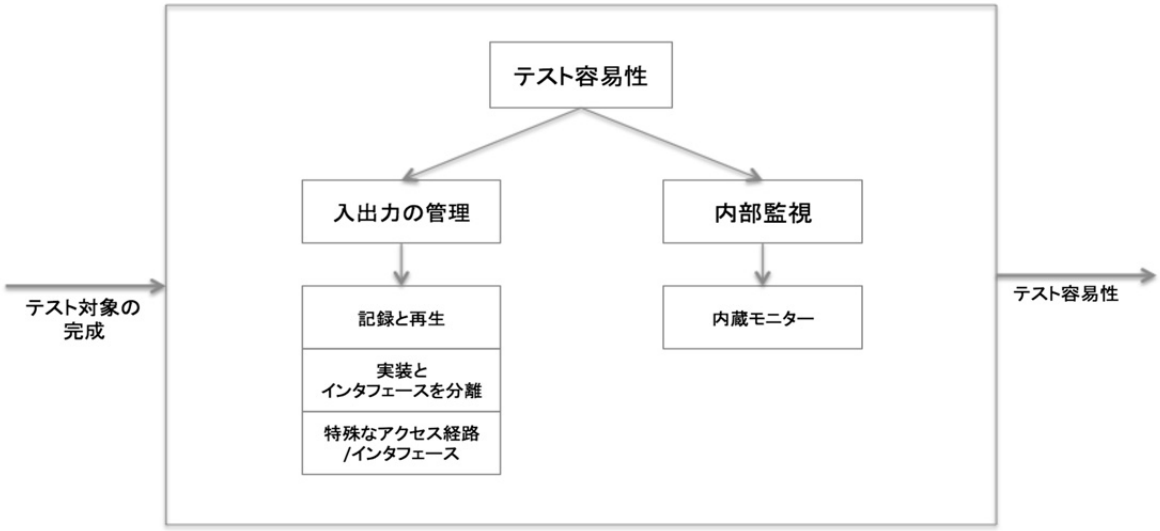
攻撃からの回復の実現手法には、システムやデータを正しい方法に戻すためのものと、攻撃者を識別する事に関するものの 2 つが存在する。前者は可用性の実現手法と重なる部分が多いが、先述の「攻撃の防御」で挙げられるような内容を考慮した上で、再度システムを回復させる必要がある事が違いとして挙げられる。

後者は、監査証跡と呼ばれる、システムのデータに適用される各トランザクションを識別情報と共に記録したデータ、を保持する事で実現される。監査証跡は攻撃者の行動を追跡し、妨害防止を支援し、システムの修復を支援する為にも利用される。

攻撃者は自身の身元を特定される事を嫌い、またシステムの回復を少しでも難しくするため、監査証跡自体を狙う事も多々ある。通常利用されるものでもないため、多少利用性が落ちたとしてもセキュリティ性が高い方法で保持する事が求められる。

■ テスト容易性の実現手法

テスト容易性とは、拡張・修正したソフトウェアのテストによる妥当性確認を容易に出来る能力を表す。テスト容易性の実現手法の目的は、ソフトウェア開発の追加分が完成した時に、より簡単にテスト出来るようにする事である。テストとは、テスト対象となるシステムに入力を与えて、その出力を捉え、それが期待した値と合致するかを確かめる事である。この入力を与え、出力を捉えるシステムをテストハーネスと呼び、その設計もテスト容易性の品質特性に影響を与えるが、今回はシステム自体の設計に注目した実現手法を紹介する。まずは、テスト容易性の実現手法の全体像を以下の図に示す。



- 入出力

  - 記録/再生

    通常利用時にインタフェースを通過する入出力の情報を捉え、それをテストハーネスの入力/出力として使用する事で、テスト用に改めてデータを用意しなくても効率よくテストが行えるようにする実現手法。

  - 実装とインタフェースを分離

    インタフェースと実装を分離することで、様々なテスト目的のために実装をスタブや、目的に沿ったコンポーネントに置き換える事を可能とする実現手法。ある 1 つのコンポーネントに欠陥が見つかった場合、そのコンポーネントの実装をスタブに置き換える事で、システムの他の部分が問題なく動作する事をテストする事が出来る。

  - 特殊なアクセス経路/インタフェース

    テストハーネスを通じてのみアクセスが出来る、テスト用の特別なアクセス経路やインタフェースを持つことで、コンポーネントの変数値や、その仕様を把握しやすくする実現手法。内部で何が起きているかをしる為に必要な情報は、システムが通常稼働時に返すべき値と異なる場合が多く、内部の状態を知る事は、システム全体への入力と出力の比較のみでは発見出来なかった欠陥が発見される事や、その発見された欠陥を修復する事を助ける事に繋

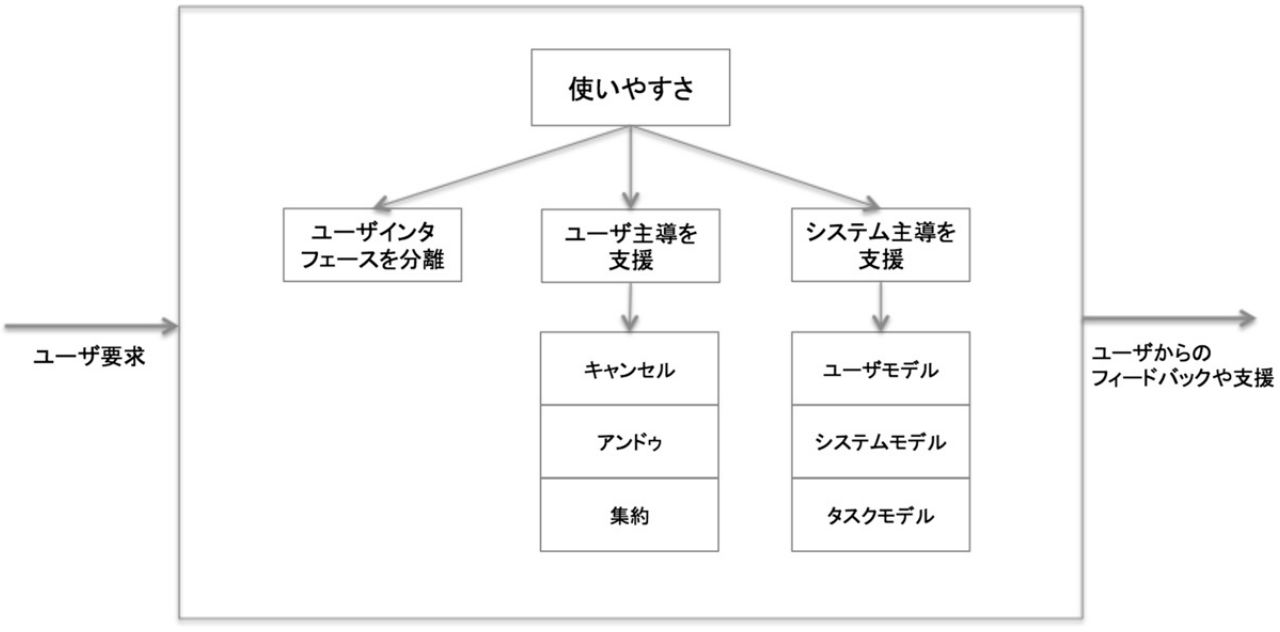
がる。

- 内部状態の監視

内蔵モニタなどのインタフェースを作る事で、コンポーネントの状態、性能負荷、容量、セキュリティ性などの情報へのアクセスを可能とする実現手法。「特殊なアクセス経路/インタフェース」でも述べたように、システムの内部状態を知る事は一般的にテスト容易性を高める事に貢献する。一方で、監視自体がシステムに影響を与えてしまう(特に性能特性など)事もあり、実際稼働時には監視の一部を落とし、必要な時のみ稼働するなどの対応が必要な事がある。その場合、監視の状態を変更したテストも行わなければならず、テストにかかる工数が増える事もある。

■ 使いやすさの実現手法

使いやすさとは、利用者が必要な作業を容易に達成出来る能力を表す。使いやすさを支援する実現手法は、設計時の実現手法と、実行時の実現手法の 2 つに分類することが出来る。使いやすさの実現手法の全体像を以下の図に示す。



- 設計時の実現手法

- ユーザインタフェースを分離

使いやすさの向上を考えた時、ユーザインタフェースは常にユーザーの要求から影響を受けて変更される可能性がある事を織り込んで設計がなされる必要がある。ユーザインタフェースに関する変更容易性を高める為、ユーザインタフェースコードを切り離しておく事で変更を局所化する事が、ひいては使いやすさの向上に繋がる。

- 実行時の実現手法

システム実行時の使いやすさは主にユーザインタフェースとして提供出来る機能に左右される。それら機能自体が実現手法となるが、これら実現手法はその実行主体によって以下の2つに分類される。

- ユーザー主導を支援

ユーザー主導の行動を支援する実現手法の頻出な例としては、「キャンセル」、「アンドゥ」、「コマンドの集約」などが挙げられる。システムの主たる機能要件として挙げられる事はないが、ユーザーの使用性を高める上で必要な機能の中でも特に、ユーザーが某かのアクションを明示的に取る事で実行される機能がここに分類される。
これらはあたかも機能性の一部であるように実装されるべきで、例えばユーザーが「キャンセル」のコマンドを発行した場合、システムはそれを受け取り、現在実行中でこの「キャンセル」コマンドが影響を与える処理をキャンセルし、その処理に使用されていたリソースを開放し、その結果をユーザーに通知しなければならない。

- システム主導を支援

ユーザーの使用性を高める上で必要な機能の中で、ユーザーの某かの行動や、イベントの発生などに対し、システムが主導で某かの判断を行い、自動的にユーザー支援を行うような機能を表す。具体例としては、英文入力システムにおいて、小文字で文が始められた際に、自動的に文の最初の文字を大文字に変換する機能などが挙げられる。システム主導で実行される機能は、ユーザーや、ユーザーが着手しているタスク、またシステムの状態に関する情報（モデル）に基づき、その機能を実行するかどうかが決定される。「ユーザーモデル」に基づき実行される機能の例としては、ユーザーないしユーザーグループがシステムを使用する際の振る舞いなどに基づき、クリックやスクロールなどのユーザーアクションに対する

応答時間を調整する事などが挙げられる。「システムモデル」に基づき実行される機能は、現在のシステムの使用されている状況と過去にシステムがイベントに対してどのように応答をしていたかの履歴を比較して行われる。現在行われている処理の予測される応答時間を計算し、それをユーザーコンソールに表示し、ユーザーの期待値をコントロールするなどが一例として挙げられる。「タスクモデル」に基づき実行される機能は、現在実行されているタスクからコンテクストから、ユーザーが行いたい事を判断して行われる。先に上げたが英文入力システムにおける、文の先頭を大文字にする機能などが挙げられる。

15.4 アーキテクチャパターンとは

アーキテクチャパターンとは、ソフトウェアパターンの中でも特にソフトウェア全体の構造に関するパターンの事で、典型的には前項で学んだ品質特性の実現手法を幾つか組み合わせ構築される。本稿では、まずはソフトウェアパターンの概要について触れ、次にアーキテクチャパターンを、具体例の1つであるLayer Patternを用いながら詳説する。

■ ソフトウェアパターンとは

アーキテクチャパターンの全容を理解する上で、その親要素とも言えるソフトウェアパターンを理解する事が重要である。ソフトウェアパターンはソフトウェア開発において必須の要素ではない。一方、開発規模および複雑性の増大、開発期間の短縮、品質に対する要求の増大、などソフトウェア開発を取り巻く環境の変化が起こり、複雑な機能要件、高い品質要件など難しい要件を持つクライアントからの要求に、短い工数で応える為に、個人的な経験知に依存するのではなく、一定の効果が認められているソフトウェアパターンを適宜利用することが求められるようになってきている。

ソフトウェアパターンは一般的に、以下のような構成で成り立っている。

前提（Context）	課題が生じる状況
課題（Problem）	前提を満たすときに、繰り返し生じる問題
解決策（Solution）	課題に対する解決策

ソフトウェアパターンは、多くのものが認知されているが、その規模（粒度）、抽象度（言語依存度）に応じていくつかに分類する事が可能で、その分類法は確立されていないが、おおむね

以下のような分類が可能。

ソフトウェアパターンの分類	意味
スタイル (style)	「コーディング規約」、「コーディングガイドライン」などとも呼ばれる。ソフトウェア開発を共同で行う際に、主として実装者（プログラマー等）間の相互理解のために便宜上定められるもので、必ずしも合理的な理由に立脚しているとは限らない。 クラスの名前の先頭は大文字で記述する、メソッドの先頭は小文字から始める、クラス変数（static 変数）はすべて大文字で記述する、などが例として挙げられる。
実装パターン (implementation pattern)	特定のプログラム言語において、特定の処理を行う際に頻繁に利用される実装方法をパターン化したもの。つまり、実装モデルに関するパターン。 たとえば、JDBC によるトランザクション処理を実装する際のパターンや、Java や C#などにおいて、メソッドやコンストラクタ、フィールドを目的に応じて適切に実装する際のパターンなどが含まれる。
デザインパターン (design pattern)	実装パターンがモジュール（クラス・インターフェイスなど）内部の実装に関するパターンであるのに対し、デザインパターンは、主にモジュール間の結合関係（設計モデル）に関するパターン。「GoF のデザインパターン」が代表例です。 実装パターンと比較して特定の言語への依存度は低くなり、ひとつのパターンを複数の言語によって実装することが可能です。

ソフトウェアパターンの分類	意味
アーキテクチャパターン (architecture pattern)	デザインパターンよりも大きな視点で捉えられるソフトウェア全体の構造（設計モデル）に関するパターン。デザインパターン以上に特定の言語への依存度はほぼ無く、アーキテクチャ設計の段階で、アーキテクチャを設計、評価する際に用いられる事が多い。 アーキテクチャパターンとして、MVC パターン（GUI アプリケーション設計などで利用されるパターン)、MVC Model2 パターン、Layers パターンなどが挙げられる。

■ アーキテクチャパターンとは

ソフトウェアパターンの中でも特にソフトウェア全体の構造に関するパターンの事で、典型的には前項で触れた品質特性の実現手法を組み合わせる事で構築される。抽象度が高く特定の言語への依存はほぼ無い。ソフトウェアパターン同様、アーキテクチャパターンも、パターンを利用せず、単に実現手法を組み合わせてアーキテクチャを設計する事が可能である。しかし、アーキテクチャパターンとして認識されている実現手法の組み合わせを用いる事で、パターンを認識している相手とのコミュニケーションが容易になる事や、パターンを適用する事のメリットとデメリットのトレードオフが既知である事、プロジェクト後の分析において変数が少なくなる事で次回以降のプロジェクトにフィードバックを活かしやすくなる、などの利点があり、近年ではその有効性が認められるようになっている。アーキテクチャパターンの代表的な例としては、MVCパターン、Layersパターンなどが挙げられる。以下にlayersパターンについて概説する（『ソフトウェアアーキテクチャ-ソフトウェア開発のためのパターン体系』（近代科学社）参照)。

■ Layersパターン

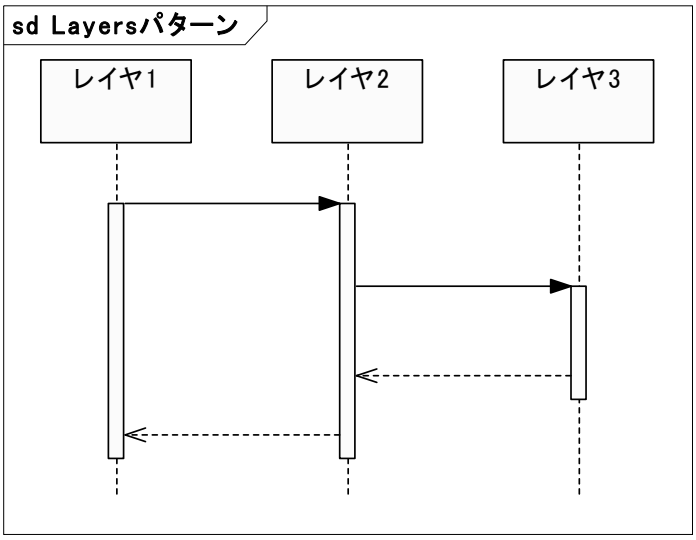
Layersパターンとは、ソフトウェア全体を、一定のまとまりをもった複数のレイヤ（layer：層）に分割し、システムをレイヤ間の協調として構成するアーキテクチャパターン。Layers パターンにおいては、各レイヤ間に次のような構造的な特徴がある。

1. 上位のレイヤは、下位のレイヤのサービスを利用する（下位のレイヤは、上位のレイヤにサービスを提供する）

2. 上位のレイヤから下位のレイヤのサービスを呼び出し、上位のレイヤはそのサービスの戻り値を取得する。

3. 上位のレイヤがアクセスできるのは、直下にある下位のレイヤである（それよりも下位のレイヤは、上位のレイヤに対して隠蔽される）。

4. 上位レイヤから下位のレイヤに対する一方向的な結合が存在する（ただし、前掲書p. 41では「最も多くの場合」と注釈されている）。



Layersパターンには、主に次のようなメリットがある。

1. レイヤの再利用

   各レイヤをそれぞれ別のプロジェクト（他のシステムやアプリケーション）などで再利用する事が出来る。

2. 依存性の局所化

   プログラムの動作条件が変更になっても、その影響を特定のレイヤに限定する事が出来、他のレイヤに影響を及ぼさない。

3. 変更の局所化

あるレイヤを別の実装へと容易に変更することが出来る。そして、その変更が他のレイヤに影響を与えない。上位のレイヤに存在するインタフェースに従い、下位のレイヤを他の実装に取り替えることが出来る。

Layersパターン自体は大変有用なパターンである一方で、利用に際してシステムをいくつのレイヤに分離すべきかと言う問題を解決しなければならない。１つの解決パターンとして提示されているのが、J2EEパターンなどで用いられるアプリケーション設計を利用した以下のパターンである。

レイヤ	役割
プレゼンテーションレイヤ	ユーザーからの入力の受付、ユーザー画面への出力を制御します（セッションの制御も含む)。

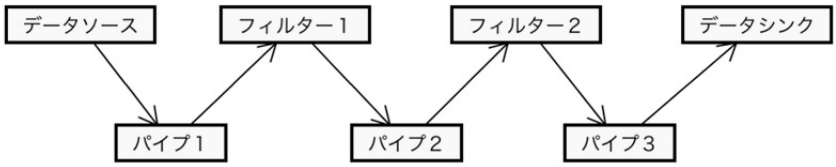
ビジネスロジックレイヤ	業務処理を実現するための処理を行い、データの妥当性のチェックや業務処理を実行する。 ●具体的には以下のような処理を行う必要がある。 ①プレゼンテーションレイヤから取得した入力データに関して、ビジネス上のルールと照らし合わせて妥当性をチェックする。 ②業務処理の実行（プログラムの本来の役割を実行） ③インテグレーションレイヤのクラスのインスタンスを利用して、データの取得やデータの記録などを行う。 ●ビジネスロジックレイヤの処理の例 商品の注文処理 商品の注文キャンセル処理
インテグレーション レイヤ	リソースレイヤへの入力やリソースレイヤからの出力を制御する。 ●具体的には、次のような処理を行う。 ①ビジネスロジックレイヤから取得したデータをリソースレイヤに書き込む。 ②リソースレイヤから取得したデータをビジネスロジックレイヤに返す。
リソースレイヤ	RDBMS、XML データベース、XML ファイル、CSV ファイルなどのデータを保存（永続化）しているレイヤ。RDBMS を利用するのが一般的。

■ その他のアーキテクチャパターン

ソフトウェアアーキテクチャ ― ソフトウェア開発のためのパターン体系（POSA：PATTERN-ORIENTED SOFTWARE ARCHITECTURE）で紹介されているアーキテクチャパターンを他にも幾つか紹介する。

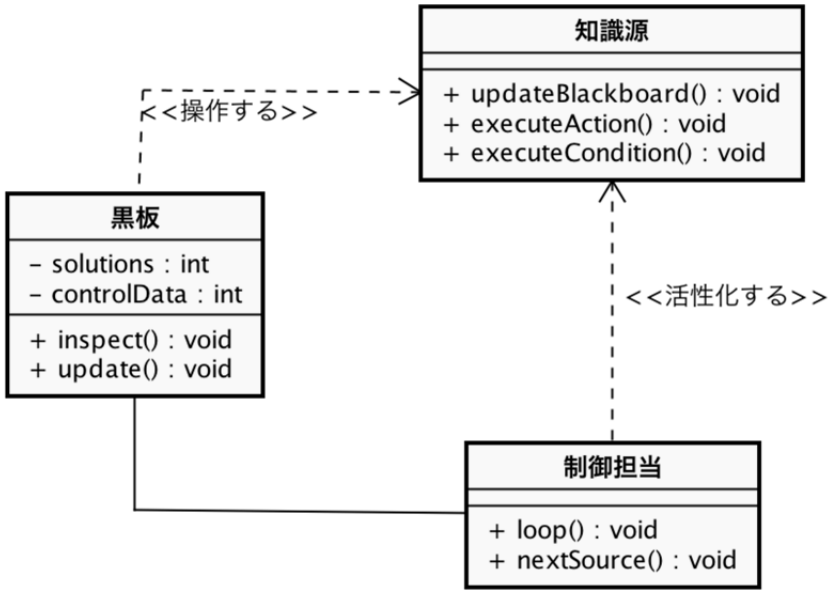
1. Pipes and Filtersパターン

データをストリームとして扱うシステムのための構造を提供する。個々のステップは1つのフィルタコンポーネントにカプセル化され、データはパイプコンポーネントを介して引き渡される。データソースからのデータがパイプコンポーネントを介して、データシンクと呼ばれる、フィルタを介して変更されたデータを使用するコンポーネントに引き渡される。



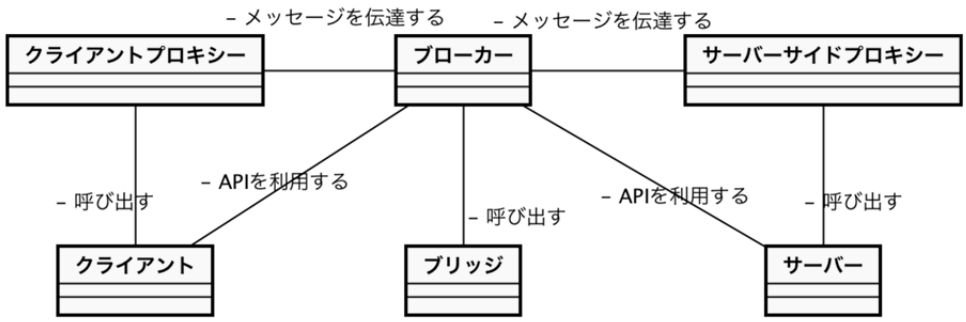
2. Blackboardパターン

複数のサブシステムが持つそれぞれの知見を組み合わせる事によって、部分的、もしくは、近似的に妥当な解を得るためのアーキテクチャパターン。答にいたる確実なアプローチが存在しないか、あるいは存在するとしても現実的ではないような、未だ成熟していないドメインで使用される。「黒板」と呼ばれる中核となるデータ書庫、「知識源」と言う問題に対して各側面からその解決に努める、独立したサブシステム、そして「制御担当」からなる。「知識源」は「黒板」からデータを読み出し、ある問題に対する特定の側面に対する解を導き出し、「黒板」に書き込みを行う。「知識源」は複数存在し、それぞれは直接やりとりしない変わりに、「黒板」を介しデータをやりとりする。「制御担当」が黒板上の変化をモニタし、次のアクションを決定、「知識源」を評価・活性化する事で適切にスケジューリングを行う。



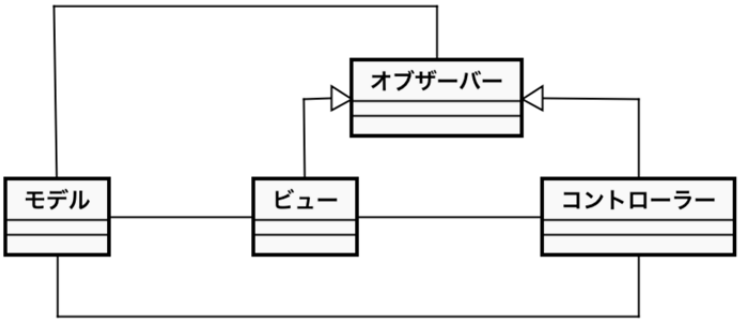
3. Brokerパターン

Brokerパターンは,分散ソフトウェアシステムを構築するために利用できるアーキテクチャパターンである。互いに依存性を持たないコンポーネント群がリモートサービスを起動することによって相互作用することで成り立つ。「ブローカー」はリクエストの転送レスポンスや例外の送信などの通信上の協調を行う責務を負う。「クライアント」はブローカーを介して「サーバー」にアクセスする事で、サーバーの位置や、変更に対しての依存関係を軽減する事が出来る。また、「ブローカー」同士は「ブリッジ」を介しやりとりが出来る。



4. MVCパターン

Model, View, Controllerの頭文字を取ってMVCパターンと呼ばれる。対話型のアプリケーションをそれら３つのコンポーネントに分類する。「モデル」は、アプリケーションの中核機能とデータを含むコンポーネント。「ビュー」は、情報をユーザに表示するコンポーネント。「コントローラ」は、ユーザーの入力を取扱、「モデル」と強調し必要な情報を集め、「ビュー」を利用して情報を返す。各モジュールが比較的はっきりと分かれ、プログラムの見通しがよくなるとともに、ユーザインタフェース（UI）部分を別のモジュールに取り替えることが容易となるのが利点である。Ruby on Railsなど主要なWeb Application Frameworkなどで取り入れられている事で有名。



（ソフトウェアアーキテクチャ ― ソフトウェア開発のためのパターン体系『近代科学社』参照）

16アーキテクチャ設計の方法論

ここまで、アーキテクチャ設計を実施する上で理解する必要がある基本概念を学んできた。第2章では、アーキテクチャ設計がどういったものであるかを概説した上で、多岐に渡るアーキテクチャ設計の定義の中から代表的なアーキテクチャに触れ、アーキテクチャ設計の実際の工程がどのように構成され、何を成果物としているかについて理解を深めた。また、成果物として出来上がったアーキテクチャを適切に表現する上で欠かせない構造とビューに関する概説を行った。第3章では、アーキテクチャ設計を考慮する際に欠かす事が出来ない品質特性についての理解を深め、シナリオの概念と、各品質特性における一般シナリオを紹介した上で、品質要件を満たす為の実現手法を、品質特性毎に体系的に概説した。本節では、これらの基本概念を用い、実際にプロジェクト要件に対してアーキテクチャを設計する上での方法論を学ぶ。まず、アーキテクチャ設計を始めるにあたり重要な要素となるアーキテクチャドライバについて概説し、現代のソフトウェア開発で用いられる事の多い代表的なアーキテクチャ設計手法を幾つか紹介した後に、代表的なアーキテクチャ設計手法の1つである、アーキテクチャ設計を行う品質駆動設計について各工程を詳説する。また、アーキテクチャ設計手法の成果として出来上がった「アーキテクチャ」を文書化する方法について述べ、最後に完成したアーキテクチャを評価する代表的な手法を紹介する。

16.1 アーキテクチャドライバとは

アーキテクチャドライバとは、アーキテクチャを「方向づける」機能要件[48]と品質要件の組み合わせである。多くの機関が提唱するソフトウェア開発工程において、「アーキテクチャ設計」は、「要件定義」を行う中でアーキテクチャドライバが見つかった時点で開始するべきとされている。

要件定義の段階では、クライアントから様々な要求が挙がってくる。それらは、プロジェクトにおいて重要な要素を占めるものから、その場の思いつきで挙げられたような些細なものまで、玉石混交の状態である。要件定義のフェーズでは、クライアントによって挙げられたこれら一連の要求を、取捨選択、また優先順位付けしながら、機能要件、非機能要件などから成るプロジェクトのシステム要求仕様書が作成される。一方で、一度の要件定義でクライアントの要求を抜け漏れなく理解し、プロジェクトの機能要件、

[48]機能要件とは、システムの機能やシステムがなすべきことなど、業務実現に直結する要件。非機能要件とは、システムを実現する上で必要な要件のうち機能要件ではないもの全般を表す。非機能要件の定義は曖昧で、提唱する機関によって、要件は機能要件、非機能要件の２つに分かれ非機能要件は品質要件や制約条件などを全て含むとするものや、要件は機能要件、非機能要件（品質要件）、制約条件の３つに分かれるとするものなど様々な定義が存在するが、本書では前者の非機能要件が品質要件と制約条件を含むものとして説明を行っていく。

非機能要件を全て抽出する事は不可能とも言える程に困難な事である(そもそも顧客自身も、自分達が何を求めているのか具体的に理解していない場合が多い)。要件定義を繰り返す事でも要求の抽出精度が上がる事は間違いないが、アーキテクチャを決定するに足る要件が定まった時点でアーキテクチャ設計を開始し、システムアーキテクチャの全容が具体的に見えはじめて来た段階で、再度要件定義を行う事でより精度の高い要求の抽出が可能となる。これは、実際にアーキテクチャ設計書を見ながら議論をする事で、クライアントが、より自分達が必要としているものを理解していく事や、今までの議論では見えていなかったトレードオフの関係、過不足のある要求などが見つかってくる為である。そして、このアーキテクチャ設計を始めるに足る、アーキテクチャを「方向づける」機能要件と品質要件の組み合わせがアーキテクチャドライバである。

16.2 代表的なアーキテクチャ設計手法

■ 品質特性駆動型設計

アーキテクチャを生成する為に、機能要件と、制約条件(非機能要件のうち品質要件ではない、システムの実装における制約を加える要件・要求。実装言語はJavaを使わなければならない、データベースはMySQLを使わなければならない、データセンターは豊洲におかなければならない、などが具体例となる)、そして一連の品質特性の具象シナリオ(品質要件)を入力として扱う。分析が行われ、アーキテクチャドライバが発見された時点で設計を開始する。与えられた品質特性の具象シナリオから、システムが満たさなければならない品質特性を把握し、それを元にシステム全体をモジュールに分割していく。モジュール分割を行うことで発見される機能性や品質特性が存在する為、ある程度再帰的なモジュール分割を行う事となるが、この時点で全ての要件を満たすような設計を完成させる事を目標とはしていない。再帰の各段階において、一連の品質シナリオを満足する実現手法とアーキテクチャパターン(実現手法の組み合わせ)を選択し、実現手法の実装に必要なサブモジュールを特定していく。実装可能なレベルに分割されたサブモジュールに機能性を割り当て、それを複数のビューを使用して表現する。最後にモジュール同士のインタフェースと制約条件を定義する事で設計を進める。

■ アーキテクチャ中心設計手法

アーキテクチャ要件を集め、それらを整理、分析し、アーキテクチャをデザインし、評価し、そして、それらをイテレーティブに洗練する手法。アーキテクチャドライバ及びアーキテクチャに

関する要件を、開発プロセスの初期段階で特定し、整理、分析することを、アーキテクトに対し強く促している。そのアーキテクチャドライバを基に最初のアーキテクチャデザインを導出し、そのアーキテクチャデザインを評価、必要があれば加えて実証事件などを行いながらイテレーティブにデザインを洗練していく事を念頭に置いた設計手法である。アーキテクチャドライバを中心にアーキテクチャの設計を開始する点は品質特性駆動型設計と類似するが、アーキテクチャドライバとしてハイレベルな機能性を用いる事としており、機能性も含めたシステムとして実現すべきコアを基に設計を開始するため、より品質特性に着目した上で設計を始める品質特性駆動型設計とは根本となる部分が異なる。また、イテレーティブにプロダクトを開発して行くのではなく、アーキテクチャ部分のみを整理、分析、評価を行い、アーキテクチャデザインを再帰的に洗練して行く部分も大きな特徴の 1 つ。
(『アーキテクチャ中心設計手法』(翔泳社)参照)

■ ニック・ロザンスキ、オウェン・ウッズによるアーキテクチャ設計手法

アーキテクチャを表現する基本要素としてビューを利用し、ビューを構築する為のパターンやテンプレート及び慣例などを集めたビューポイント(IEEE 標準 1471 に準拠)や、独自提唱しているアーキテクチャパースペクティブを利用する事で、アーキテクチャ設計に過去の経験や研究から提唱されている成功パターンを取り入れ、効率的に「要件を満たすアーキテクチャ」を設計する方法を提唱している。システムの特定の部分/機能に関する要件である機能要件と、品質特性などシステム/ビュー横断的に影響を与える非機能要件を明確に分け、機能要件に基づいて作成されるそれぞれのビューに対し、アーキテクチャパースペクティブと言う、アーキテクチャの側面から、品質要件を満たす事が可能であること保証する為の実現手法や指針を適用する事で設計を行っていく。機能要件にのみ準拠したアーキテクチャを、プロジェクトのステークホルダーに合わせビューで表現する点、各ビューにアーキテクチャパースペクティブを適用する事で品質要件を満たす事が保証される設計を作成していく点が特徴。
(『ソフトウェアシステム・アーキテクチャ構築の原理』(SB クリエイティブ)参照)

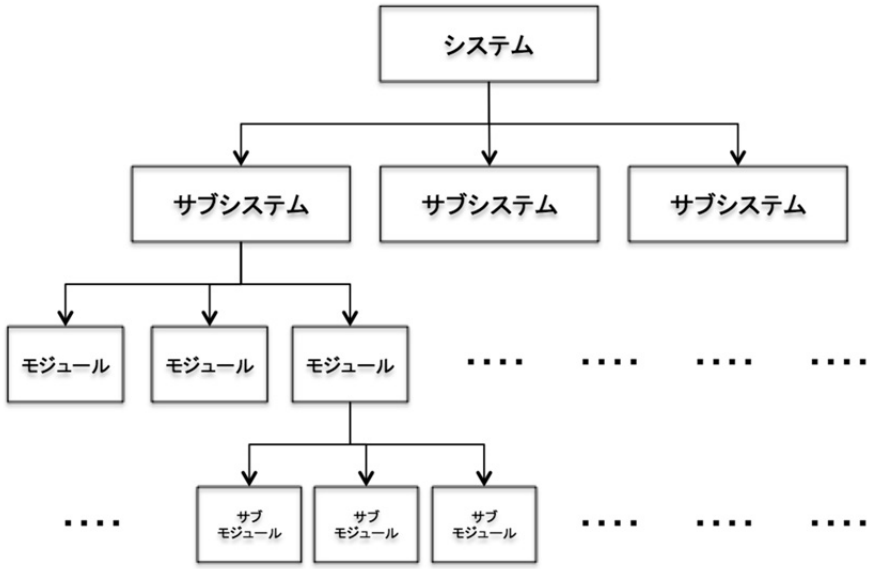
16.3 品質特性駆動型設計

前項で触れた設計手法の 1 つである、品質特性駆動型設計の各工程を詳説していく。品質特性駆動型設計は、入力として機能要件と非機能要件(制約条件、品質要件)を受け取る。アーキテクチャを「方向づける」機能要件と品質要件(一般的に制約条件は、アーキテクチャを設計上の制限をもたらす事はあるが、アーキテクチャを「方向づける」アーキテクチャドライバとはならない)の組み合わせであるアー

キテクチャドライバが特定可能なレベルまで深く要件が定義された時点で設計を開始する事が出来る。

1　分割するモジュールを選択する

品質特性駆動型設計ではモジュール分割を再帰的に行う事を前提においている。下記の図の示すように、システムはサブシステムに、サブシステムはモジュールに、モジュールはサブモジュールへと分割される（必要であればサブモジュールも更に分割される）。そのため、「モジュール」と言う言葉が重複してわかりづらいが、このコンテクストにおいては、以下の図におけるシステム、サブシステム、モジュール、およびサブモジュールは、全て分割対象となりうる「モジュール」である。



通常、最初のステップにおいて分割されるモジュールは多くの場合システム全体となる。この時点で、機能要件、非機能要件などの入力が明らかとなっている事は勿論、アーキテクチャを「方向づける」に足るアーキテクチャドライバを特定可能なレベルまで、それらが十分に精査されている事が求められる。また、品質特性駆動型設計の特徴として、品質要件が、システム/プロジェクト固有の具象シナリオとして表現されている必要がある。

2　モジュールを詳細化する

2.1　アーキテクチャドライバを選択する

一連の品質特性具象シナリオと機能要件から、アーキテクチャ、または検討中の特定のモジ

ュールを「方向づける」機能要件と品質要件の組み合わせである、アーキテクチャドライバを選択する。一般的にアーキテクチャドライバは、分割を行うモジュールに関して最も優先度の高い要件から見つかる。

ただし、優先度が高い要件が見つかったからと言って必ずしもアーキテクチャドライバとして選択が出来る訳ではない点には注意が必要。例えば、「高いセキュリティ性が必要である」と言う品質要件があった時に、これでアーキテクチャを方向づける事が難しい。実際にアーキテクチャの大きな分割構造を決定するに足る粒度の要件が抽出される必要がある。例えば「システムの一部機能を特定の IP アドレスからのアクセスのみに制限する」や「ユーザーログインを設け、ユーザーのタイプによって表示する情報を変更する」などは良いアーキテクチャドライバとなりうる。前者は、IP アドレスの制限がある部分とない部分に分けた上でそれぞれの大枠に割り当てて行けばよいし、後者ではユーザー毎にビューを変更する事を意図して MVC アーキテクチャパターンを使用して大枠を形作ると言う決断が可能となる。

選択するアーキテクチャドライバの数に決まりはないが、基本的には1つから多くても5つ程度に抑えるのが良いだろう。余り考慮すべき要件多すぎても結果どのアーキテクチャパターン(または実現手法の組み合わせ)を使用するのかを決める事が難しくなってしまう。多くのアーキテクチャドライバになり得る要件を見つけた際は、ある程度優先順位をつけて取捨選択を行う事が次以降のステップを効率的に進める事に寄与する。

2.2 アーキテクチャパターン(または実現手法の組み合わせ)を選択する

選択されたアーキテクチャドライバを実現するのに使用可能な実現手法に基づいて、既存のアーキテクチャパターン(実現手法の組み合わせ)を選択もしくは、必要な実現手法を組み合わせ、アーキテクチャパターンを生成する。

アーキテクチャドライバは、関心のあるモジュールが解決すべき最も重大な問題を特定する。全ての要件を等しく扱い、全ての問題を解決出来ればそれに越した事はないが、通常かかる工数やコストとのトレードオフなども考慮した上で、アーキテクチャドライバとなる最も重要な要件以外はその優先度や重要度に応じて据え置かれる事もある。

アーキテクチャパターンは通常 2 つの要因を考慮した上で選択が行われる。

A) アーキテクチャドライバそのものが必要とする機能要件と品質要件
B) 実現手法が他の品質特性に対して持つ副作用

ここで注意が必要なのは、既存のアーキテクチャパターンを使用する事が必ずしも有益であるケースばかりではないことである。アーキテクチャパターンは既知の問題を解決するために先

人が考え、洗練した実現手法の組み合わせであり、万能なものではない。無理にアーキテクチャパターンを取り入れた結果、より複雑性が増してしまうケースもあるため、どのアーキテクチャパターンを使用するかという事はもとより、アーキテクチャパターンを使用するか否かも重要な判断となる。

アーキテクチャパターン及び実現手法は、アーキテクチャドライバを満たすためのモジュールの大局的な分割構造(アーキテクチャの外観)を検討する上での1つのツールに過ぎない事を忘れてはいけない。

2.3 モジュールに機能性を割り当てる

アーキテクチャパターンを選択する事で、モジュールの大局的な分割構造(モジュールタイプ[49])が明らかとなった時点で、それぞれを、機能性が割り当てられる単位でサブモジュールに分割する、それぞれに機能要件を適用することで、機能性の分配を行う。この段階で、必要に応じてサブモジュールの追加・削除が行われる事となるが、親モジュールのすべての機能要件をサブモジュールの責務の範囲で表現出来るように行う必要がある。逆を言えばこの設計段階で見えていない機能要件は、開発段階において実装される事は決してないので、この機能性の分配は漏れなく行う事が重要となる。

抽象的な説明ではとてもわかりづらいので、具体的な例で考えてみる。性能と変更容易性がアーキテクチャドライバとなるシステムにおいて、性能を挙げるために並列化を行う事を考えると、システムが複雑化する事で変更容易性が低減すると言うジレンマが生じる事がある。その際のアーキテクチャパターンを考えると、「性能が重要な部分」と「性能が重要でない部分」を明示的に切り分ける構造を取る事で、「変更の局所化」を行い、並列化による複雑性の増大を、「性能が重要な部分」のみに閉じ込める事により、全体の変更容易性をある程度担保するということが考えられる。この際の「性能が重要な部分」、「性能が重要でない部分」と言う括りがモジュールタイプとなる。一方、このままではまだ機能性を割り当てる事が出来ないので、具体的な機能性のグループに分割して行く必要がある。例えば、チャット機能、通話機能、そして掲示板機能のあるチームコミュニケーションツールを考えると、リアルタイム性が求められ数秒の遅れも不便さに繋がってしまうような「性能が重要な部分」はリアルタイムチャット機能、通話機能で、「性能が重要でない部分」は掲示板機能、そして非機能要件ではある

49 アーキテクチャパターンにより分割されるモジュールの大局的な分割構造を表す。本項で扱う例では、アーキテクチャパターンによりモジュール群を「性能が重要な部分」、「性能が重要でない部分」と分ける事で、求められる品質特性を満たす設計を考えている。この、「性能が重要な部分」、「性能が重要でない部分」と言う大局的なシステムの分割構造がモジュールタイプとなる。分割方法は選択するアーキテクチャパターンや、重要となる品質特性によっても変化する。機能性とは独立して行われる分割となる事が多いため、その後機能性を持ったモジュールをそれぞれのモジュールタイプに割り当てる形を取る事になる。アーキテクチャの全体構成を決定づける重要な要素の一つである。

が、各モジュールの状況を確認するような診断機能などが挙げられる。これらそれぞれが機能性の割り当てが行われるモジュール（インスタンス）となり、これらに対し、機能要件に基づきサブモジュールの追加・削除などが行われる事で機能性が割り当てられる。チャット機能では例えば、「メッセージを送信する」、「メッセージが来た事を通知する」、「今までのメッセージ一覧を表示する」などの機能要件が考えられ、必要に応じてサブモジュールを作成し挙げられた機能要件を全て満たすように機能性割り当てを行う必要がある。

2.4 複数のビューでアーキテクチャを表現する

第一節で紹介したビューを用い、機能性割り当てまでで出来上がってきているアーキテクチャを表現し、アーキテクチャの視える化を図る。この段階では（特にプロジェクト初期においては）、完成されたアーキテクチャを網羅的に全て表現する事を目指すのではなく、プロジェクトのステークホルダーやアーキテクト自身がアーキテクチャを俯瞰出来るようにする事で、プロジェクトにおける共通の言語を定義する事や、一度可視化する事により早期にアーキテクチャ上の問題点を発見する事が目的となる。品質特性駆動型設計手法自体は、特定のビューに依存関係を持っておらず、プロジェクトの特性やステークホルダーの属性によって使用するビューを追加して行く方法を取ることが多いが、基本的には分割、並行性、および配置からなる、ビューの主要グループから 1 つずつ選択すれば良いとされる。

2.5 サブモジュールのインタフェースを定義する

モジュールのインタフェースとは、そのモジュールにおいて何が提供され、何を必要としているか、などの特性を示すものである。前のステップで作成された、分割、並行性、および配置のビューを分析し、文書化する事で、サブモジュール間において必要となる相互作用や、各サブモジュールの特性が明らかとなる。それらを作成したビューに加えて記録をする事で、それぞれのサブモジュールの役割がより明確となる。以下に、それぞれのビューの主要グループにおける記録すべきインタフェースの特性を以下に記す。

- 分割ビュー

  - 情報の提供者/利用者
  - 相互作用のパターン

- 並行性ビュー

- スレッド間の相互作用（モジュールが提供/使用するサービス）
- 活動中のコンポーネントに関する情報（所有し実行しているスレッドなど）
- コンポーネントが同期し、順次処理し、ブロックする呼び出し情報

➢ 配置ビュー

- ハードウェア要求
- タイミング要求
- 通信要求

このステップでは、アーキテクチャ構造に従ってビューを記述しただけでは捉えきれなかった、ビューの各要素間の関係性を補足的に記述することになる。前ステップと合わせて、各ステークホルダーとのコミュニケーションに必要な情報を文書化し整理する事となるが、この時点でどの程度詳細化してビューを記述するかは、プロジェクトの状況やモジュールの詳細化の状況に基づいて考慮されなければならない。ビューの利用用途（多くの場合は次の詳細化のサイクルを行う為の議論のたたき台となる）を考慮し、必要な情報を最低限載せる事となる。次章の「アーキテクチャを文書化する」にて学ぶ文書化の方法のうち必要に応じて幾つかを取り上げて利用するのが良いだろう。

2.6 機能要件と品質シナリオをサブモジュールの制約として検証し詳細化する

これまでのステップで、アーキテクチャドライバを満たすアーキテクチャパターンに基づき、トップダウンで分割されたモジュール群に対し、機能要件と、品質シナリオのうち、今回の分割において重要とされる要件を十分に満たしているかどうか、検証し、過不足があれば「サブモジュールの追加や削除を行う」、「更なるモジュール分割を行う」などして対処する。この検証、詳細化の結果、要件として挙げられているが、満たす事が出来ない要件が生じる事がある。重要な要件であった場合は、ここまでに行ったモジュール分割を再考する必要がある。もし、重要度が低くその要件を今回のモジュール分割・詳細化で扱わない事を決定するのであればその論理的根拠を記録する。

ここまでのステップで、分割を行う対象となったモジュールをサブモジュールに分割し、各サブモジュールに一連の機能要件、インタフェース、品質シナリオ、および制約などからなる責務を割り当てている。これらに基づき次のモジュール分割の反復を始める事となるが、ここで注意が必要なの

が一度の反復でどの程度モジュール分割を行うべきかと言う点である。基本的には、いずれもプロジェクト依存となり、明確な答えはないが、通常は次の反復に進む前に各チームを分割された一つ一つのモジュールに割り当てられる程度まで分割を行った段階までで留めておく事が多い。

16.4 アーキテクチャを文書化する

ここまで、代表的なアーキテクチャ設計手法を紹介し、品質特性駆動型設計に関して詳細なステップを学んできた。本項では、設計手法を用い作成されたアーキテクチャを、実際に各ステークホルダーが理解出来る形に文書化する方法について学ぶ。強固なアーキテクチャを作成しても、それを誰も理解出来ない、もしくはステークホルダーに誤解して捉えられてしまうものであっては、意味がなく、場合によってはプロジェクトの破綻の原因ともなりうる。アーキテクチャを文書化する際は、十分詳細に、曖昧さなく記述して、プロジェクトのステークホルダーが必要な情報を迅速に見つける事が出来るよう整理される必要がある。では、実際にアーキテクチャを文書化する方法論を以下に記す。

1　文書化されたアーキテクチャの使用用途を考える

最初のプロセスとして、あるアーキテクチャを誰に(who)、何を(what)、どのように(how)伝えるべきかについて考える必要がある。アーキテクチャを文書化する際に絶対に行っては行けない事の1つとして、1 つのビューで全てのステークホルダーに情報を提供しようとする事が挙げられる。ステークホルダー毎に関心やバックグラウンドの知識が異なる為、情報の粒度や、表現する側面は勿論の事、表現の仕方にも最新の注意を払う必要がある。例えば、エンジニアである実装者と、非エンジニアである顧客、どちらもシステム及びプロジェクトのステークホルダーではあるが、この 2 つの対象に全く同じアーキテクチャ文書を渡したら何が起こるだろうか。一般的に、非エンジニアが理解出来る粒度での情報は、実装を行うエンジニアにとっては粒度が粗すぎて、とても実装を行う事が出来ないし、エンジニアが使うテクニカルタームが頻出する文書は非エンジニアである顧客には理解し難いものとなるだろう。
このように、ステークホルダーによって関心事やバックグラウンドの知識は異なり、また、プロジェクトに関して既に持っている情報の種類やレベルも異なる。以下の表にて、一般的なステークホルダーとアーキテクチャへのニーズの関係性を記す。

ステークホルダー	(一般的な)文書化する目的

アーキテクトと顧客を代表する要求エンジニア	競合する要求間の対立を交渉し解決する事
アーキテクト、および構成部品の設計担当者	リソースの競合問題の解決や、性能や実行時のリソース消費の計画を確立すること
実装担当者	下流の開発活動で侵すことができない制約や、拡張可能な事由を示すこと
テスト担当者、および結合担当者	複数の部位を一緒に結合させたブラックボックステストにおいて、適切な振る舞いを定義すること
保守担当者	予想される変更によって影響を受ける部位を示すこと
当該システムと相互適用しなければならない他システムの設計担当者	当該システムとの相互運用を考慮し、当該システムが提供するもの、要求するもの、運用規約などを定義すること
品質特性の専門家	アーキテクチャ設計書が、各品質特性を評価する為の情報を含むモデルという形で提供されることで、シミュレーションやそのジェネレータ、モデル検証試験機、などの品質特性を分析するためのツールを適用し評価を行うことを可能とすること
管理者	識別済みの作業割り当てに対応する開発チームを編成すること プロジェクト資源を計画に割り当てること 様々なチームの進捗を追跡する事。
品質保証チーム	アーキテクチャ上の規定に即して、実装されていることを保証す遵守性検査の基礎を提示する事。

(『実践ソフトウェアアーキテクチャ』, pp.264, 「Clements03」から意訳の上、引用)

2 適切なビューを選択する

アーキテクチャを文書化する上で、どのビューを用いて表現するかは最も重要な要素の 1 つである。プロジェクトにおけるステークホルダーを知り、彼らが必要としている情報を把握する事で、どのビューをどのような粒度、深度で提供すべきかが変わってくる。また、システムにおける関心が高い品質特性によっても、ビューの選択に影響がある。例えば「レイヤービュー」からはシステムの移植性(変更容易性)が判断可能で、「配置ビュー」は、システムの性能や信頼性を判断する上で有効である。
これらを考慮し、プロジェクトに適したビューを選択する為には、以下の3つの手順を踏むと良い。

A) 候補となるビューのリストを作成する

最初に、プロジェクトにおけるステークホルダーを縦軸に列挙し、彼らが必要とすると考えられるビューを横軸に列挙した、下図のような対応表を作成する。この際、ビューの絞込などは行わず出来るだけ広範囲なものを挙げる事を意識する。分割ビューや使用ビューのように、すべてのシステムに適用出来るビューから、共有データやクライアントサーバーのように特定のシステムにおいてのみ適用出来るビューなど、全てを一度並列に並べてしまう事が重要。表が出来たら、それぞれのセルにステークホルダーがビューからどれだけの情報を必要とするか(なし、概要情報、ある程度詳細、非常に詳細)を記述する。
下図はステークホルダーと必要な情報の種類の対応表に関する例示、中小規模のスタンドアロンのシステム開発プロジェクトにおける例を示す。
保守担当者と開発担当者は、通常モジュールビューや割当てビューに関して非常に詳細な情報を必要とする事が多いが、今回はスタンドアロンかつ、規模も比較的小さいプロジェクトであるため、ハードウェアやプロセッサなどとソフトウェアの配置関係を表す配置ビューと、モジュールをどのファイルにおいて実装していくのかを表す実装ビューに関しては、この時点において詳細な情報がなくても良い為、ある程度詳細が記される。
プロジェクト管理者は開発の全体像を知る上で重要となる、分割、使用、レイヤービューに関して、開発担当者とシームレスにコミュニケーションを取る為ある程度詳細な情報を必要とするが、クラス構造など開発の詳細に関わる情報は必要としていない。一方、ハードウェアの構成などにも関連して、実費にも関わる配置ビューについては、顧客などチーム外部のステークホルダーとのコミュニケーションを取る為に、非常に詳細な情報を必要としている。
顧客は、開発の詳細に関わるモジュールビューは必要ない一方で、プロジェクトのコストや開発の全体像に関わる配置ビューは概要を理解しておく必要がある。

ステークホルダーと必要な情報の種類(ビュー)と深度の関係表(例)

ステークホルダー	モジュールビュー				割当てビュー	
	分割	使用	クラス	レイヤー	配置	実装
プロジェクト管理者	s	s		s	d	
保守担当者	d	d	d	d	s	s
顧客					o	
開発担当者	d	d	d	d	s	s

凡例: d=非常に詳細, s=ある程度詳細, o=概要情報

B) ビューを組み合わせる

前のステップでは、出来る限り広範囲のビューを列挙した為、現実的に考えると文書化するには多すぎる数のビューが列挙されている事が多々ある。十分に時間をかけて良いプロジェクトであれば、全てを記述しても良いかもしれないが、そのようなケースは稀である。そこで、概要しか必要としないビューや、ごく少数のステークホルダーにしか役に立たないビューを、より必要性の高いビューに統合する事で取り扱わなければ行けないビューの数を減らす事を考える。小中規模のプロジェクトでは、モジュール分割ビューに実装ビューを重ねる事や、モジュール分割ビューをレイヤービューと組み合わせる事がある。

C) 優先順位を付ける

前のステップが終了した段階で、ステークホルダーとのコミュニケーションに役立つ、一連のビューが選択されている。ここから実際にビューを文書化して行くわけだが、1 つのビューを文書化するにもある程度の時間や工数がかかる為、優先順位をつけて文書化を行って行く事がプロジェクトを円滑に進める鍵となる。優先順位付けの際は、プロジェクトにおいてより重要な品質特性をカバーするビューから作りはじめて行く事が基本となる。一方で、1 つのビューを完成しなければ他のビューに着手してはいけない訳ではないので、ある程度、幅優先で広く浅くビューを文書化していく事で、プロジェクトの全体像も俯瞰しやすく、またプロジェクト管理者や顧客などのステークホルダーから説明を求められた際に、必要な情報を提供しやすいなどのメリットがある。

3 ビューを文書化する

ビューを文書化する際の業界標準となるテンプレートは存在しないが、Bass, L が提唱する 7 項目の標準構成が、体系立ってビューを文書化する際に載せるべき最低限の項目を網羅しているので以下に紹介する。

- プライマリプレゼンテーション

  ビューに含まれる主要な要素とそれらの関係の概要を示す。システムについて第一に伝えたい情報を必ず含める。グラフィカルに図で示す場合や表で示す場合など様々な方法があるが、ビューを理解する上で必要な情報を簡潔に要約出来れば、どのような方法で表記しても構わない。多くのケースで UML（クラス図）などを用いて表記される。

- 要素カタログ

  プライマリプレゼンテーションで表現した要素と関係を詳細に述べる。図の中で使用されている要素が具体的には何を表すのか、そしてそれらの目的や役割に関しての説明が要素カタログにて示される。また、ビューに関する要素や関係の中でプライマリプレゼンテーションにおいて省略したものがあればここで説明する。

- コンテキスト図

  ビューで表現されるシステムが、その環境とどのように関係するかをビューの用語を使って示す。特に外部コンポーネントとの相互作用など、ビューにおいて示される事がない情報を表す事に使用される。
  各コンポーネントがどのように利用されるか、また外部システムとの連携がある場合はやり取りされる内容がどのようなものかを記述する事で、ビューによって表現されているシステムのある部分がシステム全体またはシステムを利用した業務フロー全体においてどのような役割を担うかを表現する。

- 可変性ガイド

  システムにおける各可変箇所の使用方法を示す。具体的には、選択可能なオプションとオプションの束縛時期（いつ可変性が失われるか）について述べる。

例えば、「あるシステムは、管理者画面とユーザー画面を持ち、同一の URL にアクセスした際、ユーザー属性によって表示される情報が可変となる。ユーザーログインをした時にユーザー属性が定まるため束縛される」などが良い具体例であろう。

- アーキテクチャの背景

設計がビューに反映されるようになった理由を説明する。設計の意図を説明する事で、妥当性の確認や、決定に至るまでの経緯などを追跡出来るようにする事が目的。一般的には以下の要素が含まれる。

  - 論理的根拠
    ビューに反映された決定の理由と、代替案が拒絶された理由を説明する。

  - 分析結果
    設計を正当化する、あるいは変更する場合に何を変更しなければならないかを説明する。

  - 設計に反映された前提

- 用語集

言葉そのままではあるが、ビューにおいて使われている(特に)プロジェクト特有の用語を説明する。ビューの読み手(ステークホルダー)の知識や、関心事に応じて、何を説明すべきかを考慮する必要がある。

- その他の情報

作者、構成管理データ、変更履歴などの管理情報が含まれる事が一般的。要求文書の特定の項目への参照を記録し、追跡可能性を確立することもある。

4 振る舞いを文書化する[50]

[50] 注) コンポーネント&コネクター構造を表すビューなどは動的な振る舞いを既に扱っている場合もある為、ビューの概念ビューの文書化の方法論の両者共 L. Bass が提唱している内容であるにも関わらず矛盾が生じるように感じる。以下、筆者の所感であるが、現状文書化を試みている部分を既に動的な振る舞いを扱うビューで表現しているのであれば、このステップは省略するなどして柔軟に対応すべきで

ビューはシステムに関する静的な構造上の情報を提示する。一方で、システムの特性について推論するには、振る舞いも合わせて記述される必要がある。たとえば、デッドロックを未然に発見し防ぐ為には、要素間の相互に行われるやり取りのシーケンスを追う必要がある。
振る舞いの記述では、要素間の相互作用の順序、並行性の機会、および相互作用の時間依存性を明らかにする情報を加える、1 つの要素、あるいは協調して動作する要素の集合に関する情報を扱う。UML で言えば、ステートチャート図とシーケンス図が振る舞い記述の例となる。

5 インタフェースを文書化する

インタフェースは、2 つの独立した要素が互いに相互作用や通信をする為の境界である。インタフェースの文書化は基本的に、名前をつけて識別する事、およびその構文情報と意味情報を記録することで行われる。実践ソフトウェアアーキテクチャで提案されている、インタフェース文書の標準構成が参考になる。

- インタフェースの識別

  要素が複数のインタフェースを保つ場合、それらを区別するために個々のインタフェースを識別する。通常、それらに名前をつける事を意味し、バージョン番号を付加する場合もある。

- 提供するリソース

  インタフェース文書の中心は、要素が提供するリソースである。リソースとは、構文、意味定義（使用した時に起こる事）、使用上の制約を与える事で定義される。それぞれについて以下に詳説する。

  - リソースの構文

    リソースのシグニチャの事を表す。シグニチャには、別のプログラムなど外部エンティティから、そのリソースを構文的に正しく呼び出す為に必要な情報が含まれる。具体的には、リソース名、引数の名前と論理的なデータ型などが含まれる。

あると考える。

- ✧ リソースの意味定義

  リソースを呼び出した場合の結果である以下の4つの要素のうちいずれかが記述される。

  1. リソースを呼び出しているアクターがアクセス出来るデータへの値の割付
  2. リソースを使用した結果として、発信されるイベント、または送信されるメッセージ
  3. リソースを使用した結果として、他のリソースがその後どのように振る舞うか
  4. 人が観測できる結果

- ✧ リソース使用上の制約

  リソースが他のリソースの存在を必要とする場合、またはある環境に依存する場合は、それらを文書化する。具体例としては、特定のメソッドが、他のメソッドが呼び出された後にのみ呼び出し可能である場合や、リソースを通じて同時に相互作用出来るユーザーの数に限界がある場合など。

- ➢ データ型の定義

  インタフェースリソースが、基本的なプログラミング言語で提供される以外のデータ型を扱う場合、そのデータ型の定義を記す必要がある。具体的には以下の 4 項目を記述する。

  1. データ型の変数や定数を宣言する方法
  2. リテラル値をデータ型に書く方法
  3. データ型のメンバーに対して、どのような操作や比較が実行されるか
  4. （データ型の値を他のデータ型に変換する方法）

- ➢ 例外の定義

  インタフェースのリソースによって発生する可能性がある例外の定義を記述する。特に規模の大きいシステムにおいては、同じ例外が複数のリソースにおいて発生する事が予期される為、各リソースの例外を個々にリストアップし、重複なく行う為、その定義についてはまとめて説明する方法が良い。

- インタフェースによって提供される可変性

  インタフェースは、何らかの方法で要素を設定出来るようになっている事がある。これらの設定パラメータと、それらがインタフェースの意味定義に与える影響を文書化する。具体的な例としては、観測可能なデータ構造の容量や、基本的なアルゴリズムの動作特性がある。

- インタフェースの品質特性

  要素の持つ品質特性を文書化する。実装の方法にもよるが、インタフェースにおいて宣言された要素の品質特性は、いわば要素のユーザーに対しての約束ともなり、要素の実装を制限する事にも繋がる。

- 要素の要求

  要素が依存関係を持つ（特に使用する）要素が提供するリソースの構文、意味定義、及び使用上の制約を記述する。設計時点でおいた一連の前提として文書化する事で、設計が進む前に前提をレビューし、その合目的性などを図る事が出来る。

- 論理的根拠と設計上の課題

  アーキテクチャについての全体的な論理的根拠と同様に、インタフェース設計についての理由を記録する。記載する項目としては、設計の背景にある動機、制約と妥協案、検討され棄却されただいたい設計、および将来のインタフェースの変更方法に対する洞察などが挙げられる。

- 使用ガイド

  「提供するリソース」「要素の要求」では、ある要素が提供及び必要とするリソースを文書化している。一方で、要素間で発生する個々の相互作用同士がどのように関係するかを推論し、プロトコルを記述する必要がある。プロトコルでは、相互作用の完全な振る舞いや、繰り返し現れると予想される使用パターンを表現する。

6　ビューをまたぐ文書の作成

ここまで、それぞれのビュー、システムの振る舞い、そして要素間のインタフェースを文書化した。最後に、複数のビューまたは文書パッケージ全体に適用される情報を文書に記録し、ビュー文書の補完する方法について議論する。
基本的に以下の3つの事柄に触れる事で、ビュー文書の補完は完遂される。

- 「どのように(How)」文書がレイアウトされ体系化されていて、アーキテクチャの利害関係者が自分の必要とする情報を効率的に探せるようになっているか。文書一式に含まれるビューの一式を紹介し、文書の読み手がより俯瞰的に文書を理解出来るようにする、ビューカタログとビューの標準構成(ビューの文書化にあたってどのような構成で文書化がされているのかを明記する)であるビューテンプレートによって成り立つ。

- 「何(What)」がアーキテクチャであるか。ここでは、ビュー単体では表現しきれないシステムの概要を記録する。システムの概要としては、複数のビューが互いにどのように関係するか、要素のリストとそれらが現れる場所、及びアーキテクチャ全体に適用される用語集など。

- 「なぜ(Why)」アーキテクチャがこの形式となっているのか。ビューやインタフェースの文書化でも触れたが、背景や、環境など、この設計が施された理由を明記する事で、より設計の意図を把握しやすくなるのと同時に、どのような議論がなされ、アーキテクチャが変様を遂げて今の形になったのかを追跡可能となる。また、何らかの方法でアーキテクチャを方向づける外部的な制約や、アーキテクチャの決定に関する根拠があればそれも合わせて記録する

16.5 アーキテクチャを評価する

4 章ではここまで、アーキテクチャの設計手法、そして、その成果物である「アーキテクチャ」をステークホルダーが理解出来る形に落とし込む文書化の方法について議論してきた。本章最後となる 4 節では、成果物として導き出された「アーキテクチャ」を評価し、分析する事で、より洗練して行く方法について学ぶ。まずは、アーキテクチャを評価する事の利点や必要条件などを確認した上で、代表的なアーキテクチャの評価手法である、ATAM の実施方法を学ぶ。

1. アーキテクチャを評価する利点

一般的なシステム開発において、手戻りは起こるのであれば早い方が良いとされている。開発工程が進む程、システムの複雑性は増し、変更にかかる工数は飛躍的に上昇していく。アーキテク

チャを評価する事は、大きなコストのかかる開発の工程に入る前に問題を発見する事が出来る 1 つのチェックポイントとなる。コスト的な観点のみではなく、アーキテクチャを評価する事による利点は複数存在し、レビューの準備を強制する事で、アーキテクチャを分かりやすく文書化するモチベーションに繋がる事、設計の論理的根拠を再確認する事で設計を見直す事、アーキテクチャが要件をどれ位満たしているかを議論する事で、要件自体について議論し見直す事、などのメリットが考えられる。

2. アーキテクチャを評価する際の必要条件

アーキテクチャの評価は、次項以降に述べるATAMやCBAMに代表される評価手法を用いて行われる。それぞれの手法は異なる切り口でアーキテクチャを評価し、設計を洗練する事に寄与するが、成功する評価プロセスには共通して、以下のような評価の為の準備が行われている。

- プロジェクトの規模がそもそも評価を行うべきかを判断する
  プロジェクトの規模によっては、そもそも費用対効果を考えると評価を行うべきではないプロジェクトが存在する。一般的に、アーキテクチャの評価は中規模以上のプロジェクトにおいて有用とされる。規模が増大する程システムの複雑性が増し、手戻りなどが開発工程の後に起きた際に計上される工数が余りにも大きい為である。逆に小規模のプロジェクトでは、評価にかかるコストが、評価を行う事の利点よりも明確に大きくなってしまう事がある。評価を行う前に、再度評価の利点と、それにかかる工数を見直し、評価を行うべきなのかどうかを判断する必要がある。

- アーキテクチャを評価する為に必要な指標を定義する
  評価を行う際には、プロジェクトにおいて求められる目標や要件と言う、設計されたアーキテクチャを適切に評価する為の評価基準を設定する必要がある。その際、少数の高い優先度の目標を定める事で、評価を行うスコープを明確化する。全体を曖昧なスコープに基づき評価すると、品質特性やその実現手法には様々なトレードオフが考えられる為、評価自体を適切に行う事が出来ず、重要な目標や要件が見過ごされてしまう事に繋がる。

- 適切な人員を確保する
  アーキテクチャの評価は、ないがしろにされがちな工程で、曖昧なまま始めてしまえば、適切な人員の工数が割かれないままに進んでしまうケースが多々存在する。評価を行うべきであると判断されたのであれば、アーキテクトを始め、各主要なコンポーネントの設計者、要求のステークホルダー、そして独立した評価チーム、などの人員の工数を先に確保する事が重要

である。次に紹介する ATAM で、最初のフェーズにおいて各人員間の協力関係を築くと言うフェーズを明示的に設けてある。

3. ATAM

ATAM: Architecture Tradeoff Analysis Method とは、アーキテクチャがどのように特定の品質目標を達成しているのかを明らかにすると同時に、品質目標同士がどのように相互作用し、トレードオフされるのかについて評価を行う。アーキテクチャの評価は、重要な要素である一方で、必ずしも必要な工程ではない為、割り当てられる時間も少なく、その限られた時間の中で適切な評価を行う事が主要な問題となる。ATAM は、目標を達成するためのアーキテクチャの中心的な部分に評価を当てる事でその問題を解決するよう工夫されている。ATAM による評価は、独立した評価チーム、顧客やプロジェクトマネージャーからなるプロジェクトの意思決定者、そして開発チームや、ユーザー、などアーキテクチャを利用し、アーキテクチャの決定事項に左右される、アーキテクチャの利害関係者を巻き込み行われる。
評価の結果として得られる出力は、評価を行う過程で得られる

1. アーキテクチャの簡潔なプレゼンテーション
2. ビジネス目標の明確な表現
3. シナリオの集まりで表現された品質要件
4. アーキテクチャ上の決定と品質要件の対応関係

そして、評価・分析によって得られる

1. 特定された一連のセンシティビティポイント(品質要件を満たすために必要なアーキテクチャ上の決定)とトレードオフポイント(品質要件に対しネガティブな影響を与えるアーキテクチャ上の決定)
2. 一連のリスク(品質要件の観点から見て望ましくない結果を引き起こすかもしれないアーキテクチャ上の決定)とノーリスク(分析に基づいて、安全であると見なされたアーキテクチャ上の決定)
3. 分析の結果発見されたリスクから、アーキテクチャ、アーキテクチャプロセス、そしてチームの組織的な弱点を特定するような一連のリスク主題

などが考えられ、実際には、これら出力を使って最終的な報告書が作成される。

ATAM に基づく評価の活動を以下に紹介する。

- フェーズ0 「協力関係と準備」

評価チームのリーダーシップとプロジェクトの主要な意思決定者が非公式に集まり、実行の詳細を計画する。評価チームが適切な人員を配備する事が出来るように、プロジェクトの要点を伝え、ミーティングの頻度や評価実務の方法などについても議論し合意を形成する。「アーキテクチャを評価する際の必要条件」でも述べたように、ATAM では、適切な人員の工数を確保する事と、お互いの協力関係を築き上げる事を最初のフェーズとして行う。

- フェーズ1

全体で9つある評価ステップのうち、ステップ1からステップ6がフェーズ1で行われる。フェーズ1では主に評価チームがプロジェクトの意思決定者とのミーティングを通して、情報収集と分析を行う。以下に詳細なステップを示す。

- ステップ1 「ATAM の提示」

評価チームのリーダーシップが、プロジェクトの代表者の集まりに対して ATAM を提示する。ATAM がどのようなプロセスで成り立っているかを伝え、質問に答え、残りの活動の背景と期待を設定する。

- ステップ2 「ビジネスドライバーの提示」

評価に関わる全員(評価チーム、プロジェクトの代表者)に、システムの背景と、開発の動機となった主なビジネスドライバーを周知する。これは、プロジェクトの意思決定者がビジネスの視点からシステムの概要を提示する。その際、以下の5つの点に触れると良い。

1. システムの最も重要な機能
2. 技術的制約、管理上の制約、コストの制約、政治の制約など、プロジェクトにまつわるあらゆる制約
3. プロジェクトに関連するビジネス目標と背景
4. 主要な利害関係者
5. アーキテクチャドライバ

- ステップ3 「アーキテクチャの提示」

アーキテクチャチームが、既に設計され、文書化されているアーキテクチャの割合、振る舞いと品質要件の性質、そして利用可能な時間、などを考慮した上で、適切な詳細レベルでアーキテクチャ説明用のプレゼンテーションを作る。プレゼンテーションでは、アーキテクチャ上の重要な情報をコンテキスト図（システムの技術的な制約や、そのシステムが相互作用する他のシステムを記す）や各種ビュー（分割ビュー、コンポーネント＆コネクタビュー、配置ビューなど）を交えて説明、またアーキテクチャ上のアプローチであるパターンや採用した実現手法を説明する。このステップでは、評価チームが評価を進める上で必要なアーキテクチャに関する情報を共有する事が目的となる。

✧ ステップ4 「アーキテクチャ手法を特定する」

ステップ4は大変短いステップで、ステップ3で行われたプレゼンテーション後の簡単なディスカッションの中で行われる。基本的にはプレゼンテーションにおいて説明がなされているはずではあるが、評価チームは、プレゼンテーションにて述べられた、アーキテクトがシステムを設計する際に用いたパターンと手法を理解した上で、更に明示的に示されていないパターンなどが使用されているようであればそれを特定する。既知のアーキテクチャパターンは、特定の品質特性への作用が周知されている為、分析にかかる工数を減らす事が出来る。

✧ ステップ5 「品質特性のユーティリティツリーを生成する」

このステップでは、ユーティリティツリーを用いて品質特性目標を明らかにする。ユーティリティツリーとは、その名の通りツリー構造を取りノードである葉とノード同士を繋ぐ節である枝で構成される。ツリー構造の一番上に位置するルートノードとして、システムの総合的な優秀性を表現した「ユーティリティ」を置くことから、ユーティリティツリーと呼ばれる。ルートノードの次のレベルは、品質特性で形成される。なぜならシステムの総合的な優秀性は、品質特性によって構成される為である。更に次のレベルでは、特定の品質特性が詳細化される。例えば品質特性の1つである「性能」は、「データ待ち時間」、「トランザクションスループット」に分解され、「データ待ち時間」は「顧客データベースの書き込み遅延を 20ms 以下にする」と「20フレーム/秒の映像をリアルタイムに配信する」などより具体的な目標に詳細化される。これら、ユーティリティツリーの葉となるノードは、3章で学んだシナリオを用いて表現される。評価時は、シナリオの6つ全ての構成要素を用いる必要はなく、そのうちの3つ（刺激、環境、応答）にて簡略化される。このようとツリー構造を深くして行くと、簡単に手作業で全てを評価する時間が足りない

だけの、シナリオが生成される。よって、優先順位付けを行う事が重要。1つ1つのシナリオに、分析したい評価基準によって高、中、低などと評価を行い、優先順位付けをする。

✧ ステップ6 「アーキテクチャ手法を分析する」

前のステップで高くランク付けられたシナリオを一度に精査し、アーキテクチャによって、それぞれのシナリオがどのようにサポートされるかアーキテクトによる説明が行われる。評価チームは、関連するアーキテクチャ上の決定を記録して、それらのリスク、ノーリスク、センシティビティポイント、トレードオフポイントを特定し列挙し、アーキテクチャ上の決定と、満たすべき品質特性の間の関係性を確立する。

ここまでが、フェーズ1の各ステップとなる。評価チームは、アーキテクチャ全体のもっとも重要な側面についての明確な図面、主要な設計方針の論理的根拠、そしてユーティリティツリーによって得られた、品質特性目標からなるシナリオをリスク、ノーリスク、センシティビティポイント、およびトレードオフポイントに分類したリストを持っている事になる。通常、フェーズ1とフェーズ2の間は1週間から2週間程度の中断を挟み、フェーズ1の時間内では分析されなかった(後回しにされた)シナリオ達が分析される。

➢ フェーズ2

プロジェクトの意思決定者が再開の準備をし、利害関係者が集められるとフェーズ2の開始と成る。フェーズ1がプロジェクトの意思決定者、評価チームのみで行われたのに対し、フェーズ2はアーキテクチャの利害関係者も巻き込み行われる。ここで、利害関係者とはアーキテクチャを利用し、その決定如何によって求められるタスクが変わるなど、行動が左右されるグループ(開発者、テスト担当者、結合担当者、保守担当者、性能エンジニア、ユーザー、検討中のシステムと相互作用するシステムの開発者、など)を表す。最初にフェーズ1で行われたステップ1を今回新たに加わった利害関係者に向けて行い、ATAM の手法と、利害関係者に求められる役割について理解させる。次に、評価チームのリーダーシップによってステップ2からステップ6の結果の要点を述べ、現在導き出されているリスク、ノーリスク、センシティビティポイント、およびトレードオフポイントのリスクを共有し、フェーズ2の残りのステップに進む準備が完了する。

✧ ステップ7 「ブレーンストーミングとシナリオの優先順位付け」

ユーティリティツリーが、主にアーキテクトが品質特性のアーキテクチャドライバをどのように理解し、扱っているかを理解するために使われたのに対し、このブレーンストーミングでは、より大きな利害関係者による意向を探ることである。このステップでは、評価チームが利害関係者に個々の役割に関して経営上重要な意味を持つシナリオをブレーンストーミングするように求める。ある程度シナリオが集められた後に、優先順位付けと、ユーティリティツリーによって得られたシナリオのリストとの比較が行われる。優先順位付けには特に工夫が求められ、数多くの利害関係者それぞれに優先順位について議論をさせると多くの時間を消費してしまう為、通常全体のシナリオ数の30%程度の投票数を割り当て投票させるなどする。

✧ ステップ8 「アーキテクチャ手法を分析する」

ステップ7でシナリオが集められ、優先順位がつけられた後は、評価チームがアーキテクトを誘導することで、それぞれのシナリオとアーキテクチャ上の決定に関する関係（特に、事前に説明がなされているアーキテクチャ手法や実現手法をベースに話を進めると良い）を明らかにしていく。そして、ステップ6と同様にリスク、ノンリスク、センシティビティポイント、トレードオフポイントを列挙していく。

✧ ステップ9 「結果の提示」

評価チームのリーダーシップにより、ATAM の各ステップと、この手法で集められた情報を要約したプレゼンテーションが行われる。通常、プレゼンテーションでは以下に挙げるような出力を提示する。

- 文書化されたアーキテクチャ
- ブレーンストーミングからの一連のシナリオとそれらの優先順位付け
- ユーティリティツリー
- 発見されたリスク
- 記録されたノンリスク
- 見つかったセンシティビティポイントとトレードオフポイント

これらの出力はただ文書化されるのみではなく、基本的な関心事やシステムの欠陥などに基づきリスクをリスク主題にグルーピングする。例えばある古くなった文書に関するリスクのグループを、「文書が不十分とかんがえられる」というリスク主題にまとめる。グルーピングによって求められた一連のリスク主題を、ビジネスドライバーと関連付けし、

それらのリスクが管理可能なレベルまで持ち上げる。

- フェーズ3

フェーズ3は、評価チームが最終報告を作成し配布する事から開始される。しかし、このフェーズの本質は、チームの自己分析と改善にある。事後検討で、上手く言った事、上手くいかなかった事を議論し、次のプロジェクトに活かせる改善点を探す。定量的に行う為、評価チーム、プロジェクトの意思決定者、そしてアーキテクチャの利害関係者、の3つのグループでそれぞれがどの程度の労力を費やしたかをまとめておき、後に評価を行ったプロジェクトの依頼人と接触するなどして長期的な効果を測定、費用対効果を計算する。

17演習

本章では、要件定義の部でも扱った「在宅介護サービス支援システム」を題材として、実際に品質特性駆動型設計を用いてアーキテクチャを設計し、前章までに学んだ知識を体系立てて理解する事を目指す。要件定義の部における演習で抽出した機能要件、非機能要件、そして品質特性を入力として、成果として文書化されたアーキテクチャ設計書を生成する事を目標とする。

17.1 プロジェクトの背景

要件定義の部との重複とはなるが、まずはプロジェクトの背景を再確認するところから始める。本プロジェクトは、訪問介護事業を運営する株式会社 A 社からの発注を受け、「在宅介護サービス支援システム」を開発する事が目標である。A 社は従業員 100 人以上 200 人以下の中小企業で、以下のような問題を抱えシステム開発プロジェクトを依頼してきた。

A 社の現状

訪問介護事業を運営する株式会社 A 社は、介護現場で課題となっている日々の申し送り効率化の課題を抱えている。訪問介護を担当する介護ヘルパー（20～70 歳代までの幅広い年齢層が登録されている）は登録制の直行直帰型勤務が多く、事務所への出社は週に 1 度程度である。1 人の要介護者を複数のヘルパーが入れ替わり担当する介護の現場では、日々の申し送りに大変苦労している。
要介護者の体調や薬、食事の内容、量など内容は命に関わる事柄も含まれ、連絡の抜け漏れは許されない。それほど重要な業務にも関わらず、現在の運用は、その日の介護を終えた担当ヘルパーからの電話報告を事務所スタッフがノートにメモし、次の担当ヘルパーへ電話で伝えるという方法を取っている。

A 社の課題

- 1人の要介護者に対し複数のヘルパーが交替で介護担当をすることが多い介護現場で、日々の申し送りを抜け漏れなく行うことに大変な労力を費やしている。
- 事務所スタッフが電話報告を受け、ノートにメモを取って次のヘルパーに申し送りをしているが、過去の履歴データを確認できず不便である。
- 介護記録は手書きで記録管理しており、数年に一度の行政監査の際には、スタッフが介護記録を整

理し、ファイリングして提出している。

システム要件

事務所スタッフがヘルパーへ申し送りをする方法は、一斉送信用画面から、要介護者名、その介護に関わるヘルパー名（複数選択可）を選択し、申し送り内容を本文に記して、ヘルパー宛へメールの一斉配信を行う。メールを受信したヘルパーは、受信したメール本文から申し送り内容を確認後、同メールに記載されたリンク先をタッチして、ヘルパーメールシステムにログインし、「メールを確認しました」ボタンをタッチする。この作業でヘルパーメールサーバー上に既読情報が記録される。

ヘルパーから事務所スタッフへの申し送りは電話報告の運用を続ける。これは、ヘルパーからの報告をシステム化すると、携帯からの文字打ちが必要となり、操作に時間も要し、メールが苦手なヘルパーからの報告が抜け漏れするリスクがあるためである。

そのため、介護終了時のヘルパーからの申し送り内容は、事務所スタッフが電話を受けてシステムに入力し、情報を蓄積する。システムから次の担当ヘルパーへ一斉送信すれば申し送りが完了する。

システム上では、ヘルパーの未読・既読一覧状況がすぐに把握できるほか、日々報告される申し送り情報を介護記録として管理できる。介護記録は、数年に一度の行政監査の際に使用する。過去の履歴をCSV 形式で書き出すことも可能である。

17.2 要件定義の成果

要件定義の部の成果として、以下のようなUSDMで記述された機能要件と、リスト化された非機能要件が挙がってきている。

USDM で記述された機能要件

要求	要求番号 仕様番号	説明
要求	SYSM01	一斉送信用画面から、要介護者名、その介護に関わるヘルパー名（複数選択可）を選択し、申し送り内容を本文に記して、ヘルパー宛へのメールの一斉配信を行う。

	理由	介護ヘルパーは登録制の直行直帰勤務が多く、日々の申し送りに苦労しているから、これを解消するため。
	説明	
<Req-01-01>		
□ □ □	Req-01-01-01	（サインインユーザによって申し送りができるヘルパーに制限があるか？）ヘルパー名（複数選択可）を選択する。 なお、選択するヘルパー名は、要介護者名を選択すると、その介護に関わるヘルパー名の一覧が選択できるようにする。
	理由	申し送りをしたいヘルパーを選びたい。
□ □ □	Req-01-01-02	メールで一斉配信を行う。
	理由	1回の操作で全員に申し送りを送信したいから。
□ □ □	Req-01-01-03	送信できなかったヘルパーがいれば、画面にエラーメッセージを表示する。
	理由	ヘルパーがメールアドレスを変更し、システムでそれを変更しなかった場合、申し送りの抜けができてしまうから。
□ □ □	Req-01-01-04	メールの一斉配信数は、申し送り事項1件に対して10名程度に配信する。
	理由	
要求	SYSM02	システムから一斉送信された申し送り事項（メール）をヘルパーが受信したら、同メールに記載されたリンクをタッチすると、ブラウザに返信内容画面が表示され、「メールを確認しました」ボタンをタッチすると、ヘルパーメールサーバ状に既読情報が記録される。返信内容画面にメッセージを入力して「メールを確認しました」ボタンをタッチすれば、既読情報とともに、メッセージがシステムに記録される。
	理由	ヘルパーが申し送り事項を確認することができることと、事務所スタッフがヘルパーが申し送り事項を読んだこと、メッセージがあればそれを確認するため。
	説明	
<Req-01-02>		
□ □ □	Req-01-02-01	ヘルパーは、携帯のメール本文のリンクをタッチすると、リンクアドレスがブラウザで表示される。
	理由	システムに接続する方法を統一して、ヘルパーによって操作ミスをなくすため。
□ □ □	Req-01-02-02	「メールを確認しました」ボタンをタッチすると、システムに既読情報と返信内容が記録される。
	理由	単純操作で、ヘルパーが申し送りの確認を怠らないようにするため。

□ □ □	Req-01-02-03	ヘルパーメールシステムに既読情報と返信内容が記録される。 なお、記録情報はこれらの他に、 ・回答日時 ・ヘルパー名 ・対応介護者名 ・返信内容 が記録される。
	理由	事務所スタッフが申し送り事項の既読状況を確認するため。
□ □ □	Req-01-02-04	ヘルパーが申し送り事項を送信した際に、正しくシステムに送信されると、正しく送信されたことを確認するための確認画面を表示する。
	理由	送信時になんらかのトラブルがあり、正しく送信されたかどうかを確認するため。
□ □ □	Req-01-02-04	ヘルパーが申し送り事項を送信した際に、正しくシステムに送信されなかった場合は、再度、確認メッセージを送るように伝える画面を表示する。
	理由	送信時になんらかのトラブルがあり、正しく送信されたかどうかを確認するため。これはヘルパーが再度、申し送り事項の確認をさせるために設ける。
要求	SYSM03	事務所スタッフが申し送り事項の配信状況を確認する。
	理由	どのような申し送りがなされたかを確認するため。また、ヘルパーの既読状況を確認し、未読で急ぎの場合は申し送りを再送する。
	説明	
<Req-01-03>		
□ □ □	Req-01-03-01	事務所スタッフはシステムで、要介護者名を選択すると、その介護者を担当するヘルパーへ宛てた申し送り事項が一覧で表示される（最も最新の申し送り事項については、申し送り事項の詳細内容が表示される） なお、詳細内容には、 ・送信時刻 ・所属事務所 ・件名 ・本文 ・送信数 ・既読数 が表示される
	理由	事務所スタッフは、最新の申し送り事項について確認する作業がもっとも優先度が高いため。

□ □ □	Req-01-03-02	事務所スタッフはシステムで、要介護者名を選択すると、その介護者を担当するヘルパーへ宛てた申し送り事項が一覧で表示される（2回前の申し送り事項は、Req-01-03-01の詳細内容の下に、既読管理というタイトルでて表示される。 なお、表示される内容は、 ・送信グループ名 ・スタッフ名 ・状態（既読／未読／送信内容確認） ・既読にするボタン（未読の場合に表示） が表示される
	理由	要介護者に対する申し送り事項の一元管理を行い、ヘルパーの対応状況を確認するため。
□ □ □	Req-01-03-03	Req01-03-02 にある、申し送り事項の一覧数は過去10件を表示し、それ以上過去のものは、25件ずつ次の画面に遷移して表示する。
	理由	一画面での見え方、および画面表示のレスポンスを考慮する必要があるため。

抽出された非機能要件

ユニークユーザー数	・介護施設の全職員：200 人(うち 100 人が非正規職員) ・携帯電話やスマートフォンを使いこなせない高齢職員数：50 人 ・アクティブユーザー数：200 人－50 人＝150 人がアプリを常用
被介護者数	・1,000 人 ・介護状況の報告者は、正規職員からおよそ 100 名が担当する。
システムピーク時間	・食事介助の後の報告となる 13 時から 13 時半、17 時半から 18 時の間に報告のピークが発生し、30 分間で 100 件程度。 ・報告を受けた結果の通知のピークが、報告の後の 30 分間で、報告 1 件に対して担当ヘルパー10 人に連絡するため、30 分間で 100 件×10 人＝1,000 件の通知が想定される。その 30 分後に返信のピークが同数程度発生するものとする。
セキュリティ性	・申し送りデータ（申し送り事項）を暗号化する ・ヘルパーが受信した申し送り事項確認画面において、リンクアドレスはハッシュ化されており、アドレスは一時的（24時間有効）な利用とすること。 ※24時間以上を経過したアドレスは無効となり、事務所スタッフから再送される必要がある。 （理由）リンクアドレスの流出を防ぎ、第三者による不正回答がなされな

	いようにしたい。 ・システムに登録されているデータのうち、パスワードは暗号化されていること。 （理由）不正アクセスからの情報流出を防ぎたい。 ・システムへの想定外のアクセス（HTTPアクセスによる参照）記録をとっておく。 （理由）不正アクセスの痕跡を発見し、対応したいから。 ・システム管理画面には、事務所内からのアクセスに限定する。 （理由）社外からの不正アクセスを制限するため。	7
サーバ仕様	・1 件当たりの通信データ量を想定して、トラフック制限のあるサーバプランから、レンタルサーバ業者を選定する。可用性、信頼性についても同様に検討する。 ・耐用年数は 5 年間とする。	

17.3 アーキテクチャを設計する

上述の機能要件と非機能要件は、アーキテクチャドライバを抽出するのに十分な要件が抽出されていると考えられる為、アーキテクチャ設計を開始出来る。4章で学んだ品質特性駆動型設計の各ステップに基づきアーキテクチャの設計を行っていく。

以降は質問と解答形式になるが、是非各問いに対して自分なりに答えを導き出した上で、先を読み進めて行って欲しい。アーキテクチャ設計には１つの答えがあるわけではないため、筆者の示す解答は様々なパターンが考え得る中での１つの例であり、自ら問いに対して熟考をする事が重要である。また、より多くを学び得るために、もし周りに同様にアーキテクチャ設計を学ぶ者がいれば、自らの思考のプロセスを開示し、お互いにフィードバックや議論を行うと良い。多くの視点や考え方を知る事がとても大きな学びとなる。

1） 分割するモジュールを選択する

品質特性駆動型設計では「分割するモジュールを選択する」、「モジュールを詳細化する」と言うプロセスを繰り返し行う事で、モジュール分割を詳細化して行く事を前提としている。今回は要

件定義が終了して直後の第一回目のサイクルとなるので、システム全体が分割を行うモジュールという事になる。

2) モジュールを詳細化する

2.1 アーキテクチャドライバを選択する

上述の要件定義において抽出された機能要件と非機能要件のリストからアーキテクチャドライバを選択する。演習という概念上、先に挙げられた USDM によって表現されている機能要件(SYSM01, SYSM02, SYSM03)は全てアーキテクチャドライバとなるべき重要なもののみが抽出されているため、今回は非機能要件から選択を行う。非機能要件として挙げられている項目の中からアーキテクチャドライバとなり得る重要な品質要件を選択し、それを第三章で学んだ品質特性一般シナリオを参考にしながら、品質特性具象シナリオを作成し、以下の表を埋めてみよう(アーキテクチャドライバとなりそうな品質要件が複数ある場合は複数挑戦してみてほしい)。
注)一般的に品質特性駆動型設計を行う際には、要件定義の段階で品質特性具象シナリオとしてが捉えられる。今回は要件定義の部においてメインで USDM を扱った事から、要件が USDM 形式で記述されているため、非機能要件から特に品質要件が抜き出されずに同等に扱われている。今回は、品質特性具象シナリオを生成する練習も兼ねて、非機能要件の一般的なリストの中から重要な品質要件を見つけ出し、品質特性具象シナリオとして記述するプロセスをここで行う。

シナリオの構成要素	可能な値
刺激の発生源	
刺激	

成果物	
環境	
応答	
応答測定	

読者の皆様はどのような非機能要件をアーキテクチャドライバとして取り上げただろうか？筆者は、アーキテクチャに重大な影響を与え、かつ、大変重要な非機能要件として「システム管理画面には、事務所内からのアクセスに限定する」を取り上げる。理由としては、システム管理画面からは全てのヘルパーからの申し送り事項の入力がなされ、そして一覧が確認できる。第三者からの攻撃を想定した場合、申し送り事項のデータの改ざんは要介護者の命に関わる可能性もある重大なリスクをはらみ、更に申し送り事項に含まれる医療情報に準ずる機密性の高い個人情報の流出は企業としての信頼を大きく失う事に繋がる。また、一部分の機能へのアクセスを制限する事はアーキテクチャの構造にも大きな影響を与える為、プロジェクトに与える影響度が大変

大きな要件であると想定される。セキュリティ性の品質要件である事から、セキュリティ性の品質特性一般シナリオを元に以下のような形で品質特性具象シナリオが生成された。

シナリオの構成要素	可能な値
刺激の発生源	1. 識別の可/不可 ・ 身元を正しく識別されている ・ 身元を誤って識別されている ・ 知られていない身元 今回のケースでは、識別が出来たとしてもアクセスを制限する。例えば社員が社内以外のネットワークを利用してアクセスを試みると、第三者に情報を傍受されてしまう可能性が0とは言い切れない為、それも防ぐ。身元が改ざんされていたり、知られていないものなどはもってのほか。 2. 権限の有無 ・ 社内ネットワーク外部だが権限が有る ・ 社内ネットワーク外部の上権限も無い 社内ネットワーク外部からのアクセス 3. アクセス元 ・ 事務所内ネットワーク以外
刺激	・「システム管理画面」にアクセスしようとする試み
成果物	システムのサービス/システムに関連するデータ
環境	・ システムがオンラインで利用可能な状態 ・ システムがネットワークに接続されている状態 ・ システムがファイアウォール内にある状態

応答	システムは刺激発生源が社内ネットワーク外部である刺激を受けた場合データ/サービスへのアクセスを拒否する
応答測定	アクセスログを定期的に解析し、その中に外部からのアクセスに対して間違いなく拒否が行われている事を確認

また、「ピーク時には 30 分間で 100 件程度の報告」と言う品質要件もシステムの要件、そしてアーキテクチャの設計、双方に大きな影響を与える可能性がある。約1分に3回の報告が考えられ、一時的にはその2倍程度の負荷がかかる事も考えられる。報告を受けた後に、事務所スタッフが報告内容をシステムに入力し、ヘルパーを選択、一斉送信を行う事を考えると、一定期間におけるメールの送信数が過大になる事が想定される。メール送信は I/O を伴うリソースをブロックしがちなプロセスなので、システム管理画面と同一プロセスで行われると、性能に重大な悪影響を及ぼす可能性がある。そこで、ピーク時の平均値の約2倍の10秒に1件以上の申し送り内容の一斉送信を行っても、システム画面に遅延が出ない事という、要件定義であがった品質要件をより具体化した、性能の具象シナリオを作成した。

シナリオの構成要素	可能な値
刺激の発生源	ヘルパーからの報告（電話）
刺激	ヘルパーからの電話による申し送り内容の報告を受け、事務所スタッフがシステム管理画面から要介護者の申し送り文書を入力、選択されたヘルパー（要介護者を介護する可能性のある）に一斉送信をする。 ピーク時には 30 分間で 100 件程度の報告。
成果物	システム管理画面
環境	通常状態（ここでは便宜上通常状態における性能のシナリオを記述する、非常状態や過負荷状態も必要に応じて考慮すると尚良い）

応答	10秒に1件以上の一斉送信(ピーク時で約20秒に1件)を行ってもシステム管理画面の機能に遅延が出ない事(1ページビューに5秒以上かからないこと)。
応答測定	システム管理画面のレスポンスタイムを計測

2.2 アーキテクチャパターンを選択する

アーキテクチャドライバとして上で品質特性具象シナリオを作成した、2つの品質要件を選択したと仮定した上で演習を進めていく。

1.「システム管理画面には、事務所内からのアクセスに限定する」

2.「ピーク時には 30 分間で 100 件程度の報告」

注意点として、非機能要件として要件定義で発見された一方で、重要度が比較的に低くアーキテクチャドライバとして選択されなかった要件は、アーキテクチャ設計上満たすべき制約として考慮される必要がある場合がある。
「システムに登録されているデータのうち、パスワードは暗号化されていること」、「申し送りデータを暗号化する」、「リンクアドレスはハッシュ化されており、アドレスは一時的な利用とする」などは特に設計に影響を与える可能性のある制約ではないかと考えられるため、モジュールの大局的な分割構造などを考慮する際には意識をする必要があるかもしれない。

さて、これらからアーキテクチャパターン及び実現手法を選択していく。アーキテクチャドライバ及び、アーキテクチャ設計上の制約として考慮した上で、まずは適用されるべき実現手法を特定してみよう、次にその実現手法が他の機能性や品質特性に対して持つ副作用を考えてみよう、そして最後にそれらを踏まえた上で、モジュールの大局的な分割構造(モジュールタイプ)はどのようになるだろうか？

考慮すべき内容	

選択するアーキテクチャパターン・実現手法	
アーキテクチャパターン・実現手法が他の機能性や品質特性に与える影響	
モジュールの大局的な分割構造	

モジュールの大局的な分割構造まで設計出来ただろうか。

筆者は、品質要件である「システム管理画面には、事務所内からのアクセスに限定する」、「ピーク時には 30 分間で 100 件程度の報告」の2つに注目し、実現手法としてセキュリティ性の実現手法である「アクセスを制御する」と性能の実現手法である「並行性の導入」を選択した。

「アクセスを制御する」は明らかで、「システム管理画面には、事務所内からのアクセスに限定する」を満たすためには必ず実施する必要がある実現手法となる。

「並行性の導入」は「ピーク時には 30 分間で 100 件程度の報告」という非機能要件を満たす為のものであるが、これは少し理由を考える必要がある。まず、ピーク時に 100 件程度の報告を受けると言う事は、それに伴い一斉送信のメールが送られる事になる。メールが送信された直後からシステムには当然ヘルパーからの既読情報とメッセージが送られて来る事になる。メールの送信は一般的にブロック時間の長い処理と考えられ、暗号化された申し送りの内容を復号化、ハッシュ化されたリンクを生成して、メールが送信されるまでのプロセスを考えた時に相応のリソースが消費さ

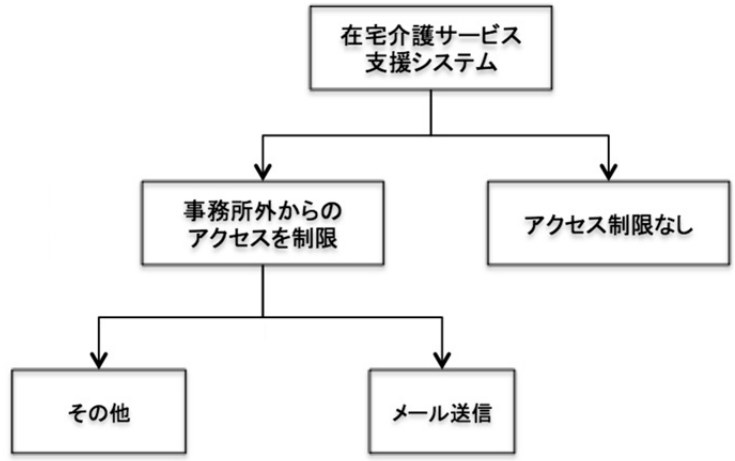
れる事は明らかである。よって、それらのブロック時間が大きい処理を別のスレッドで行う事でブロック時間を可能な限り短くする事が求められると考えた。

これら実現手法が他の機能性や品質特性に与える影響においては、「アクセスを制御する」の与える影響を特に考慮する必要がある。これは、システム全体に実施すると全ての機能が事務所内以外からアクセス出来なくなってしまう。これは機能要件、特に「ヘルパーが「メールを確認しました」ボタンをタッチすれば、既読情報とともに、メッセージがシステムに記録される」を考慮した時に重大な問題となる。

そこで筆者は、モジュールの大局的な分割構造として「事務所外からのアクセスを制限」と「アクセス制限なし」と言う切り口で「在宅介護サービス支援システム」を分割する事とした。更に「事務所外からのアクセスを制限」に関しては、「メール送信機能」と「その他」の部分を明確に分割する事で「並行性の導入」をアーキテクチャ上で実現する構造を取った。

以下、

考慮すべき内容	
選択するアーキテクチャパターン・実現手法	「アクセスを制御する」、「並行性の導入」
アーキテクチャパターン・実現手法が他の機能性や品質特性に与える影響	「アクセスを制御する」 システム全体に実施すると全ての機能が事務所内以外からアクセス出来なくなってしまう。「「メールを確認しました」ボタンをタッチすると、システムに既読情報と返信内容が記録される。」と言う機能要件に大きな影響を与える事となる。 「並行性の導入」 並行性が導入され一部機能が別スレッドに移されると、相互のスレッド間のやり取りが発生し、システムの変更容易性、テスト容易性などが低下する事が考えられる。

モジュールの大局的な分割構造[51]	

2.3 モジュールに機能性を割り当てる

次は、1つ前のステップで作成した機能性を割り当てる前のモジュールタイプによるシステムの大局的な分割構造を元に、更に機能性が割り当てられる単位であるサブモジュールに分割し、機能性の分配を行ってみよう。この分割において抜け落ちてしまった機能は後の精査のタイミングで拾い上げられる事がない限り基本的に実装される事はない為、必要な機能要件が全て取り入れられている事を確認する必要がある。また、一度モジュールに機能性を割り当て、図として表現した段階で、システムの構成に必須な機能要件が要件定義の成果物から抜け落ちてしまっていないかをよく吟味する事で、手戻りが少ない形で上流工程を進めて行くことが出来る。

まずは、「事務所外からのアクセスを制限」の部分と「アクセス制限なし」の部分それぞれに、以下の3つの機能要件を割り当てる所から始める。

1.「一斉送信用画面から、要介護者名、その介護に関わるヘルパー名（複数選択可）を選択し、申し送り内容を本文に記して、ヘルパー宛へのメールの一斉配信を行う。」

2.「ヘルパーが「メールを確認しました」ボタンをタッチすれば、既読情報とともに、メッセージが

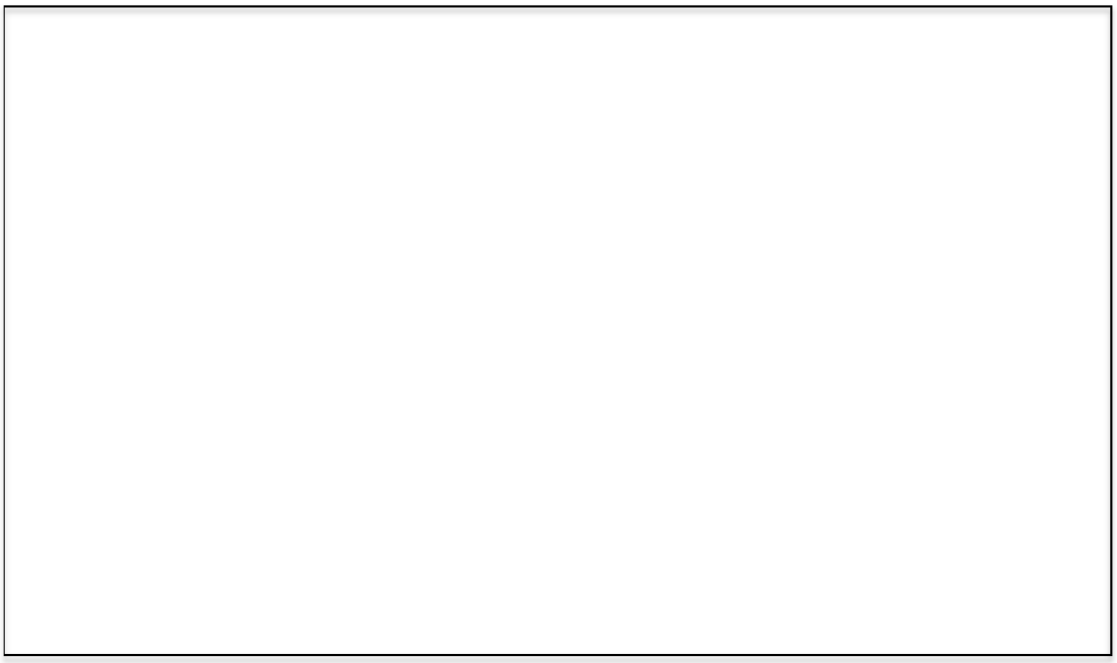
[51]機能性を割り当てる前のモジュールタイプによるシステムの大局的な分割構造を図化したもの。UMLのような特定の記法はなく、全体構造を捉える事が出来て、表現する事が出来ればどのような方法を用いても構わない。

システムに記録される。」

3.「事務所スタッフが申し送り事項の配信状況を確認する」

以下の枠内に、機能性を割り当てたシステムの全体構造を記述してみよう。

在宅介護サービス支援システム全体構造

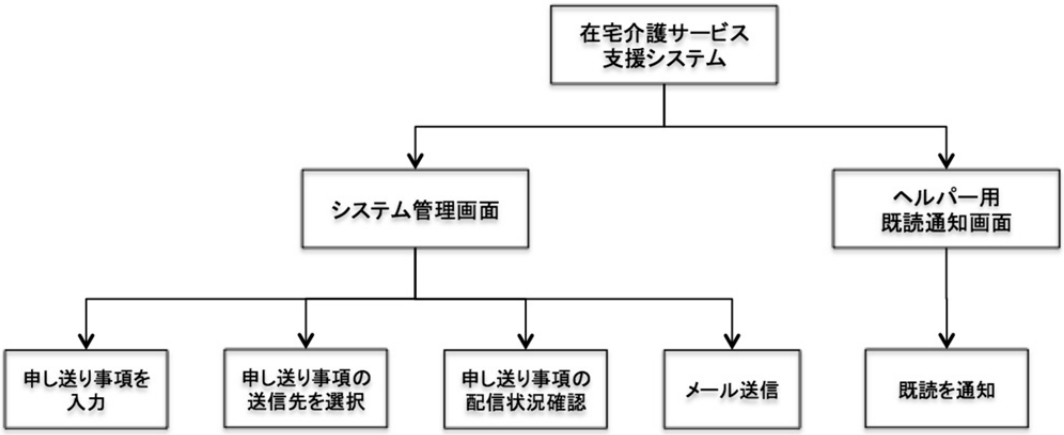


システムの全体構造を記述出来ただろうか？今回は演習という事もあり、シンプルな構成になっており、簡単だという感想を抱くと同時に、これをやる意味があるのだろうかと考える読者の皆様も多かったのではないかと考えられる。一方、実際のシステム・ソフトウェア開発プロジェクトにおいては、求められる規模も比べ物にならない程大きく、必要な機能要件が割り当てられず抜け落ちてしまう事や、全体構造が分からないまま議論されたために必須の機能が、機能要件として抽出されないまま設計が進んでしまう事も多々ある。このように一度表現をして言語化する事で、各ステークホルダーとのコミュニケーションにおける共通言語が出来る上に、全体構造への理解を深めた状態で要件定義を見直す事も可能となる為、大変重要な工程となっている。

さて、筆者の記述したシステムの全体構造は以下のようになる。「事務所外からのアクセスを制限」と「アクセス制限なし」をそれぞれ「システム管理画面」、「ヘルパー用既読通知画面」とし、それらのサブモジュールにそれぞれ機能性を割り当てた。

機能同士が可能な限り疎結合となるよう、「一斉送信用画面から、要介護者名、その介護に関わるヘルパー名（複数選択可）を選択し、申し送り内容を本文に記して、ヘルパー宛へのメールの

一斉配信を行う。」と言う機能要件を「申し送り事項を入力」、「申し送り事項の送信先を選択」、「メール送信」と言う3つの機能に分割している。
この部分で例えば、メールは一度送信されてしまうと編集が出来ないのだけど、送信先を選択する画面で、申し送り事項も入力するのは危険ではないかと言う疑問が生まれ、申し送り事項は送信前に一度登録、確認をしてからメール送信の画面に進めた方が良いのではないかなどという議論が生じたりする。こうした議論は、システムとして実現したい事を考慮していた段階では具体性に欠けて気付けない事も多く、このように機能性として分割され、具体化されていく中で見えて来ることも多い。



2.4 複数のビューでアーキテクチャを表現する

モジュールへの機能性割り当てが出来た所で、現状のアーキテクチャをビューとして表現して行く。今回の演習では、この段階から「アーキテクチャを文書化する」で紹介したプロセスを使って表現を行っていくが、どの段階からアーキテクチャを文書化するのかは、実際にはプロジェクトにおいてアーキテクチャ設計にかけられる工数や、システムの複雑さなどに依存する。現状のような一度目のモジュール分割で、抽象度及びモジュール分割の粒度が高い状態からアーキテクチャを文書化して行くことは当然より多くの情報が視える化されるため、より詳細な議論が可能となる一方で、工数が多くかかってしまう。この段階では、簡易なビューを幾つか作成するまでとし、実際に詳細なモジュール分割が完了して初めて文書化を行う場合も多々あるため、そこはプロジェクトの特性などによって臨機応変に対応する必要がある。以下、早速「アーキテクチャを文書化する」に

おける各ステップに従って、複数のビューを用いアーキテクチャを表現していく。

2.4.1 文書化されたアーキテクチャの使用用途を考える

アーキテクチャを表現し文書化する最初のステップとなるのが、アーキテクチャの使用用途を考える事である。4.4 項にて触れたステークホルダーの一覧(「アーキテクトと顧客を代表する要求エンジニア」、「アーキテクト、および構成部品の設計担当者」、「実装担当者」、「テスト担当者、および結合担当者」、「保守担当者」、「当該システムと相互運用しなければならない他システムの設計担当者」、「品質特性の専門家」、「管理者」、「プロダクトライン管理者」、「品質保証チーム」)から、現状のアーキテクチャは誰と何を議論・明確にする為に使用されるべきかを考えてみよう。4.4 項に記した、「一般的なステークホルダーとアーキテクチャへのニーズの関係性」の表から各ステークホルダーとそれぞれの文書化する目的を参考にしながら、考えてみて欲しい。

ステークホルダー	文書化する目的

現状のアーキテクチャをどのステークホルダーに向けて、何を議論・明確化する為に文書化して行くかは明らかとなっただろうか。

筆者は、現状の段階で考慮すべきステークホルダーは「アーキテクトと顧客を代表する要求エンジニア」、「アーキテクト、および構成部品の設計担当者」ではないかと考えられる。まだ表現する対象となるモジュール分割された全体構造が、実際に実装、テスト、保守運用などを行う「実装担当者」、「テスト担当者、および結合担当者」、「保守担当者」や、それらの管理、監視を行う「管理者」、「プロダクトライン管理者」、「品質保証チーム」と議論するには抽象度が高すぎる状態である事が考えられる。また、「当該システムと相互運用しなければならない他システムの設計担当者」は初期の段階から議論を行うべきであるが、本システムの特性からは相互運用される外部のシステムは存在しない。消去法のような形で見つけ出されたステークホルダーだが、以下にそれぞれの文書化する目的を記す。

ステークホルダー	文書化する目的
アーキテクトと顧客を代表する要求エンジニア	システムの全体像を捉え、必要な要求に抜け漏れがない事を確認する事 競合する要求があればそれを早期に発見し再議論（要件定義）する事で解消する事
アーキテクト、および構成部品の設計担当者	システムの全体構造から、必要な品質要件を満たす事が出来る事を再確認する事 システム全体のフローを把握する事でリソース消費の予測を立てボトルネックとなる部分を予め解消する事 更なるモジュール分割を行うために必要な情報を可視化する事

2.4.2 適切なビューを選択する

先程検討したアーキテクチャの使用用途を元に適切なビューの選択を行っていく。各ステークホルダーとそれぞれが各ビューにおいて必要とする情報の詳細度を表す以下の表を利用して、今回の文書化において必要なビューを特定していく。
まずは凡例に従い、第二章で紹介されている各ビューの特徴を元に、以下の表に各ビューにおいて必要な情報の粒度を埋めてみて欲しい。この時、何故そのビューにおいてステークホルダーがその詳細度の情報を必要とするか、一つ一つ検討し理由付けをする事が重要となる。

ステークホルダー	モジュールビュー				コンポーネント&コネクタビュー			割当てビュー	
	分割	使用	クラス	レイヤー	並行性	共有データ	**クライアントサーバー**	配置	実装
要求エンジニア									
設計担当者									

凡例： d=非常に詳細， s=ある程度詳細， o=概要情報

筆者は以下のように表を埋めた。
分割ビューは、全体の構成を把握する上で大変重要なビューとなる為、「要求エンジニア」、「設計担当者」ともに詳細な情報を必要とする。
使用ビューは、各モジュール同士の使用関係を表す為、「要求エンジニア」、「設計担当者」ともに

情報を必要とするが、今回のプロジェクトにおいては複雑な使用関係は現状想定しづらい為、ある程度詳細な情報で十分とした。
クラスビューは、「設計担当者」としてはあれば今後設計を詳細化する際に役に立つ為概要として知っている必要があるのではないかと考えた。
レイヤービューは、まだレイヤー構成を取るほどに複雑化されていない為今回は必要ないものとした。

並行性ビューは、並行性を導入している為「設計担当者」はある程度詳細な情報を必要とするが、一方単純にメール送信機能を切り分けているだけなので実は簡易な構成で、この段階では必要性が低いと捉えた。
共有データビューは、1つの永続データ(DB)に対してシステム管理画面と既読通知画面の2つがアクセスをするが、クライアントサーバービューにて扱える為、重要度を下げた。
クライアントサーバービューは、今回2つのクライアント(システム管理画面と既読通知画面)に加え、サーバーも「事務所外からのアクセスを制限する」と「アクセス制限なし」に分かれる事から、「要求エンジニア」、「設計担当者」とも詳細な情報を必要とする。
配置ビューは、「要求エンジニア」としては知る事で要求抽出の手助けになる可能性がある一方で、必ずしも詳細まで全てを知っている必要がないので、ある程度詳細な情報があれば良い。
「設計担当者」は、性能を担保するために必要最低限なプロセッサーの能力などを把握する必要があり、詳細な情報を必要とする。
実装ビューは、現状の段階では詳細な情報まで踏み込めるだけのモジュール分割が行われていない為必要がない。

ステークホルダー	モジュールビュー				コンポーネント&コネクタビュー			割当てビュー	
	分割	使用	クラス	レイヤー	並行性	共有データ	**クライアントサーバー**	配置	実装
要求エンジニア	d	s					d	s	
設計担当者	d	s	o		s	o	d	d	

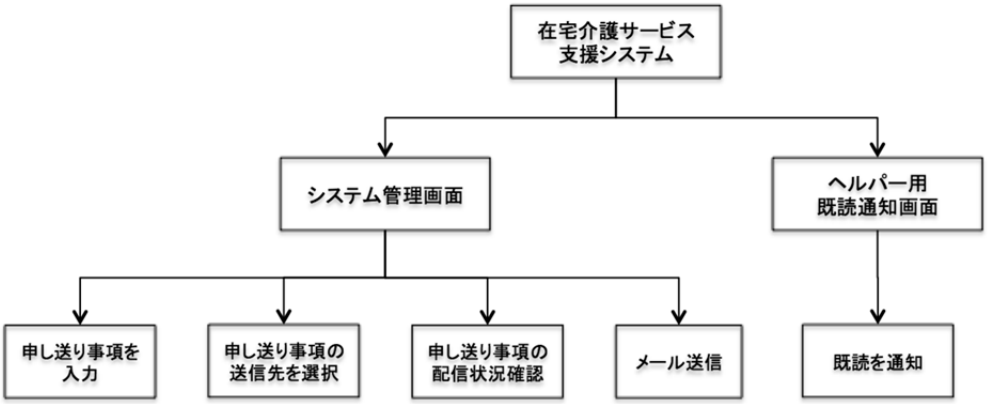
凡例： d=非常に詳細， s=ある程度詳細， o=概要情報

表が埋まった段階で、概要しか必要としないビューや、ごく少数のステークホルダーにしか役に立たないビューを、より必要性の高いビューに統合する事で取り扱わなければ行けないビューの数を減らす事を考える。今回は、便宜上「要求エンジニア」、「設計担当者」ともに非常に詳細な情報を必要としている「分割ビュー」について文書化を進めていく。実際のシステム・ソフトウェア開発プロジェクトにおいてどのビューを(そしていくつのビューを)文書化するかは、プロジェクトの複雑

さや、アーキテクチャ設計にどの程度の時間をかけて良いのか、などと言った要素で決定をしていく必要がある。

2.4.3 ビューを文書化する

ここからは、「分割ビュー」の文書化を行っていく。「アーキテクチャを文書化する」で学んだ Bass, L が提唱する 7 項目の標準構成に従い文書化を行っていく。先程の「2.3 モジュールに機能性を割り当てる」における成果物は「分割ビュー」としてそのまま利用出来るため、以下に記す図を今回文書化して行くことを考える。



「4.4 アーキテクチャを文書化する」のステップ3を参考に、次のページの表を埋めてみよう。幾つかの要素については既に記述すべき項目が列挙されているので、本章の最初に紹介した、要件定義の成果物である機能要件と非機能要件を再度確認しながら、必要な情報を埋めていって頂きたい。

文書化項目	内容
プライマリプレゼンテーション	
要素カタログ	システム管理画面とは: 申し送り事項を入力とは: 申し送り事項の送信先を選択とは: 申し送り事項の配信状況確認とは: メール送信とは: ヘルパー用既読通知画面とは: 既読を通知とは: システム管理画面とヘルパー用既読通知画面の関係:

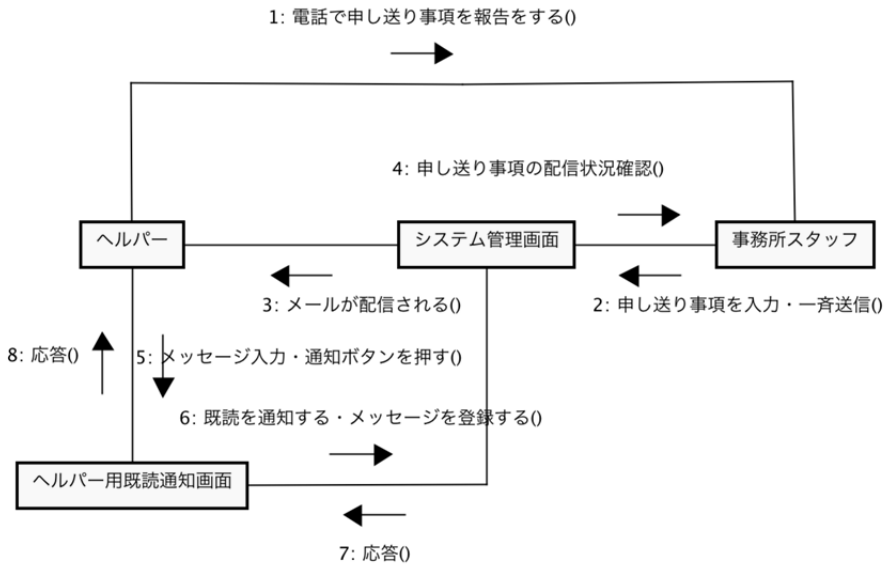
コンテキスト図	
可変性ガイド	
アーキテクチャの背景	
用語集	申し送り事項とは: 申し送り事項の送信要件とは: その他:
その他の情報	

上手く文書化する事は出来ただろうか？実際にはこれらの文書化を複数のビューに対し行い、必要であれば文書化しているビューの振る舞いやインタフェースの定義を行うが、本演習ではあまりに詳細に踏み込みすぎてしまう為割愛する。
次のページに筆者が行った文書化の例を記載するので参考にして欲しい。

文書化項目	内容
プライマリプレゼンテーション[52]	「在宅介護サービス支援システム」の大きなモジュール分割構造を示す「分割ビュー」 大きく分けると以下2つのサブシステムに分割されている。 ・ 事務所スタッフが利用し、オフィス内からのみアクセスが出来る、「システム管理画面」 ・ ヘルパーが利用し、どこからでもアクセスが出来る、「ヘルパー用既読通知画面」 「システム管理画面」では事務所スタッフが 「申し送り事項の入力」、「申し送り事項の送付先の選択」、「メール送信」、「申し送り事項の配信状況の確認」を行う。

[52] プライマリプレゼンテーションに関しては、UML などを用いて記述するパターンもあり、どのように書くかの定めはない為、今回はビューが大まかな構造を表している為、文字でキーポイントを伝える形を取った。

要素カタログ	システム管理画面とは: 事務所スタッフが利用する画面、オフィス内からのみアクセス出来る。「申し送り事項を入力」、「申し送り事項の送信先を選択」、「申し送り事項の配信状況確認」、「メール送信」などが機能性として割り当てられている。 ・申し送り事項を入力とは: 事務所スタッフが、ヘルパーから電話を受けた際に受け付ける申し送り事項を入力出来る機能。 ・申し送り事項の送信先を選択とは: 「申し送り事項を入力」で入力した申し送り事項をメールで送信する際の宛先を選ぶ機能。 ・申し送り事項の配信状況確認とは: 「申し送り事項を入力」で入力された申し送り事項の配信状況(未配信、配信済、既読、など)を確認出来る機能。 ・メール送信とは: 「申し送り事項の送信先を選択」をした結果実際にメールを送信する機能。 ヘルパー用既読通知画面とは: ヘルパーが利用する画面、「既読を通知」する事で、「申し送り事項の配信状況確認」と言う部分での、申し送りのステータスを変更する機能を持つ。 既読を通知とは: ヘルパーが申し送り内容を受け取った時点で、既読である旨を事務所スタッフに伝える(より具体的には申し送り事項の配信状況を更新する)機能。 システム管理画面とヘルパー用既読通知画面の関係: 「システム管理画面」上に表示されるべきデータ(申し送り事項の配信状況確認)を「ヘルパー用既読通知画面」が書き換える。

コンテキスト図	
可変性ガイド	可変箇所はこの段階では顕在化していない為、省略。

アーキテクチャの背景	論理的根拠: アーキテクチャドライバに「システム管理画面には、事務所内からのアクセスに限定する」、「ピーク時には 30 分間で 100 件程度の報告」と言う品質要件（上の説明では他の品質要件ではない要求と合わせて非機能要件と記載）を選択。アクセスの限定の有無でまず、システム管理画面と、ヘルパー用既読通知画面を明確に分割、その後システム管理画面に関して、メール送信の件数がピーク時に大きくなりブロック時間が長くなる事を想定し、メール送信部分を後に別スレッドに割り当てられるように、その他の部分と明確に分割。最も大枠の分割構造が完成した後は基本的に大枠の機能性を割り当てて行く事でサブモジュールへ分割が行われて現状のビューが出来上がった。 分析結果: 1. 事務所内からのアクセスに制限をするためには IP アドレスでのアクセス制限を行う事もしくは Intranet 内にシステムをおく事が考えられる。いずれにしても、アクセス制限がない部分とは物理的に切り離されることで、この要件は担保される。 2. ピーク時には 1 分間に平均 3 件程度の報告が上がる。最ピーク時をピーク時平均の2倍程度と考えると、1 分間に 6 件、つまり 10 秒に 1 件の報告がある。一件あたりの申し送りが数分程度で行われるとすると、実際には事務所スタッフが入力する事も考えられる時間も含めると、5 分弱で 1 人が 1 報告をあげる形になる。スタッフの人数が 30 名未満であればスタッフの人数分、15 名以上であれば最大で 30 名が並列で稼働する事になる。1件の報告で 10 名程度に一斉送信を行う事を考えると、割とシビアな数のメール送信用 I/O が発生し性能に重大な影響を与える可能性が高い。 設計に反映された前提:更なるモジュール分割を見据えた大枠を表現
用語集	申し送りとは: 業務の継続に必要な情報を後任者に伝える事 申し送り事項の送信要件とは: ・ 要介護者の個人情報が含まれる為セキュリティが重要 ・ 要介護者を担当する可能性があるヘルパー全員に送信が必要 ・ 要介護者の命に関わる場合もある為、次に担当するヘルパーは必ず申し送り内容を把握していなければならない（配信状況確認の必要性） その他:特になし

その他の情報	2017/02/xx　Ver1.0　Initial Draft 作成者：xxxx

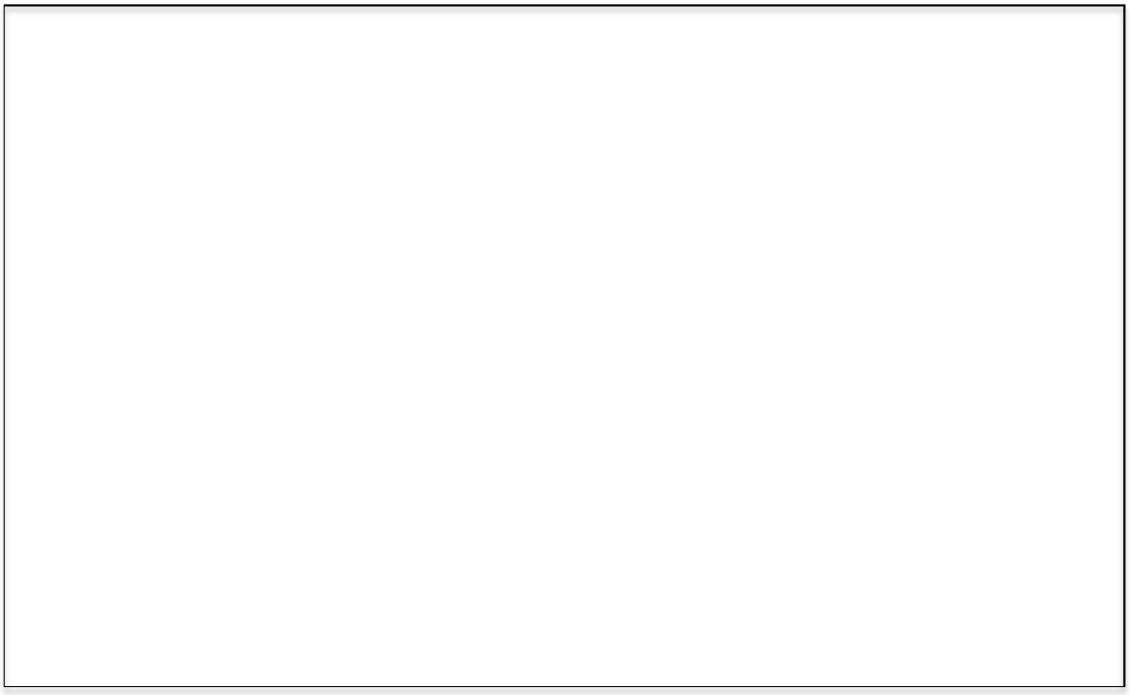
ここまで、「品質特性駆動型設計」のモジュール分割の各ステップと、その最後のステップ[53]を「アーキテクチャを文書化する」と組み合わせる事で、それぞれの実施方法を学んだ。
実際には、「品質特性駆動型設計」においては、次のモジュール分割のサイクルに入り、再度「1. 分割するモジュールを選択する」へ戻り、次のソフトウェア開発工程に進むのに十分な形でモジュール分割が行われるまで再帰的にモジュール分割のサイクルを繰り返すと言う事を行う。

最後に、開発工程に入るのに辺り重要な、クライアントサーバービューとクラスビュー（クラス図）がどのような形になるか、最終形態を考える事を持って演習を終了とする。可能であれば、先程の続きから（もしくは自分で最初から）「品質特性駆動型設計」のモジュール分割を再帰的に行い、詳細設計まで踏み込んでみて欲しい。

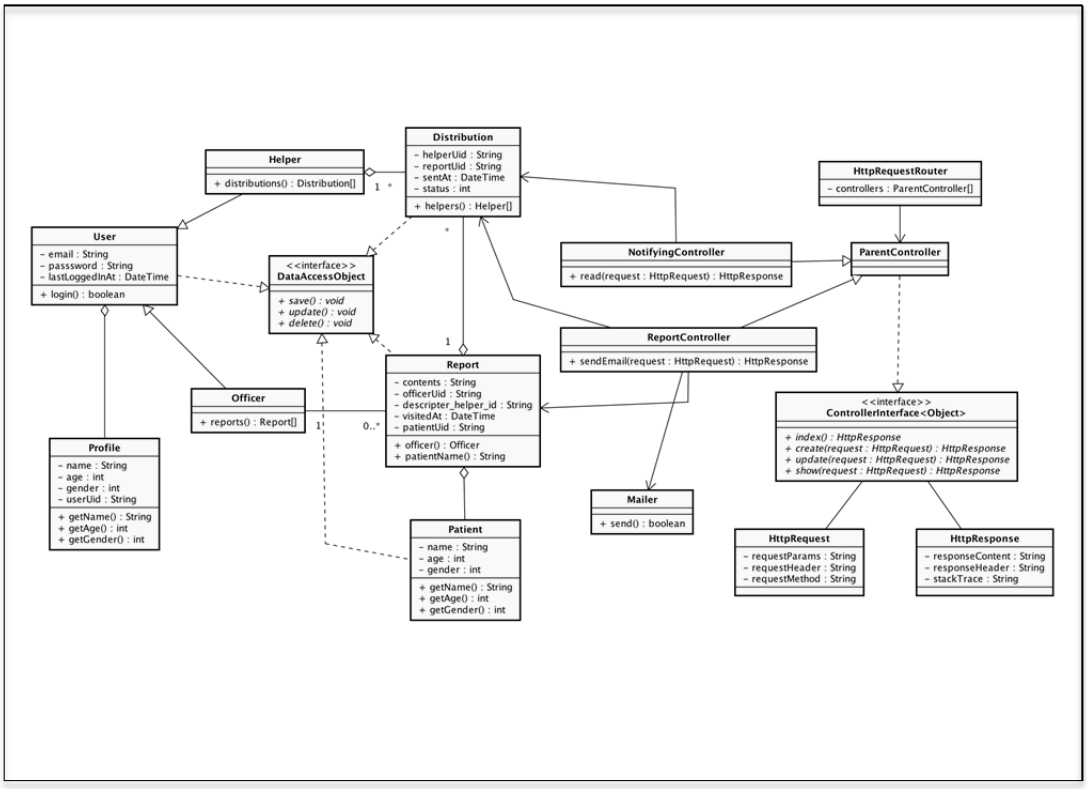
53 注：実際には「サブモジュールのインタフェースを定義する」、「機能要件と品質シナリオをサブモジュールの制約として検証し詳細化する」があるが、今回は割愛。詳細の方法については「品質特性駆動型設計」の「ステップ 2.5: サブモジュールのインタフェースを定義する」や「ステップ 2.6：機能要件と品質シナリオをサブモジュールの制約として検証し詳細化する」、「アーキテクチャを文書化する」の「ステップ５：インタフェースを文書化する」などを参照。

クラスビュー

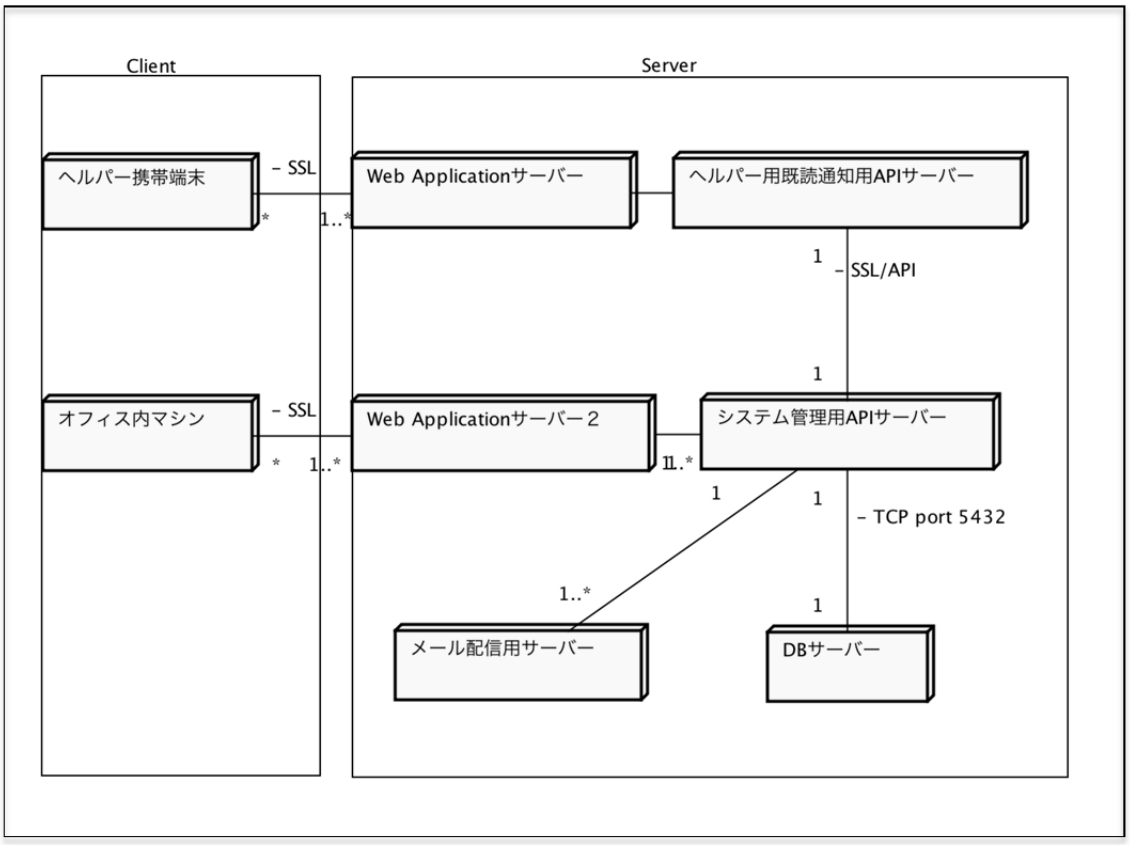
クライアントサーバービュー



クラスビュー



クライアントサーバービュー


Client
Server
ヘルパー携帯端末
SSL
Web Applicationサーバー
ヘルパー用既読通知用APIサーバー
*
1..*
1
SSL/API
1
オフィス内マシン
SSL
Web Applicationサーバー２
システム管理用APIサーバー
*
1..*
1..*
1
1
TCP port 5432
1..*
1
メール配信用サーバー
DBサーバー