平成 29 年度　文部科学省
「成長分野等における中核的専門人材養成等の戦略的推進」事業

地域 IT 産業振興のための IT アーキテクト育成事業

事業成果報告書

平成３０年２月

学校法人河原学園
河原電子ビジネス専門学校

本報告書は、文部科学省の生涯学習振興事業委託費による委託事業として、学校法人河原学園 河原電子ビジネス専門学校が実施した平成２９年度「専修学校による地域産業中核的人材養成事業」の成果をとりまとめたものです。

目次

1. 本事業の概要 5
1.1. 事業の趣旨と目的 5
1.2. 事業実施の背景 5
1.2.1. 地域産業の特性 5
1.2.2. IT 分野の成長可能性 6
1.2.3. 本事業を実施する意義 7
1.3. 事業の実施体制 7
1.3.1. 概要 7
1.3.2. 各委員会の役割 8
1.3.3. 委員会・作業部会の構成 9
1.3.4. 実施委員会の開催 10
1.3.5. 作業部会の開催 12
1.3.6. 評価委員会の開催 13
2. 事業活動の内容 14
2.1. 教育プログラムの方針 14
2.2. 成果物の概要 15
3. 実証講座の内容 16
3.1. 概要 16
3.1.1. 目的 16
3.1.2. 受講者 16
3.1.3. 実証講座の概要 17
3.1.4. 理解度確認テスト 18
3.1.5. アンケート 21
3.2. 実証講座の実施 23
3.2.1. 受講者 23
3.2.2. 実施内容 27
3.3. 実証講座の結果 33
3.3.1. 事前理解度確認テストの結果 33
3.3.2. 事後理解度確認テストの結果 34
3.3.3. 理解度確認テストの比較 36
3.3.4. 事前アンケートの結果 37
3.3.5. 事後アンケートの結果 38

3.3.6. 統計分析の結果 51
3.4. 総括 57
3.4.1. 考察 57
3.4.2. 結論 59
4. 事業成果物 60

1. 本事業の概要

1.1. 事業の趣旨と目的

近年、愛媛県は観光地として発展を遂げつつある反面、いまだに将来有望な地域産業を見出すことができないでいる。既存の産業構造の特性および立地・物流面での不利も影響していると考えられる。他方、県内情報産業は、この数年売上高を伸ばしているとともに、すでに 3 つの業界団体が組織され、一定の地域的基盤を形成している。ただし、県内情報産業は大都市圏の大手 IT 企業の二次請け、三次請けの受託が中心で、このままでは今後の大きな事業拡大を見込むことは難しいという課題を抱えている。

そこで、県内情報産業に、利益率の高い SI（System Integration）事業へと事業展開する契機を提供するため､SI 業務の中核をなす IT アーキテクトの育成プログラムを開発する。プログラム開発にあたっては、今年度は多様な理論・方法論を比較解説するテキスト、講座の再現性を高めた詳細なシラバス・コマシラバス、受講者の能力判定を行う理解度確認テストを開発する。全体で、IT アーキテクトに必要な広い視野と応用力を備えた人材育成プログラムの開発を目指す。

1.2. 事業実施の背景

1.2.1. 地域産業の特性

愛媛県は、道後温泉・松山城・しまなみ海道等の観光資源によって、観光地として全国的な知名度を獲得しつつあり、その結果、県外観光客の年間総消費額が 2014 年には 968 億円に達し､2005 年から約 16%も増加している（愛媛県「観光客数とその消費額」）。しかし、観光産業の振興とは裏腹に、県内人口は 2005 年 147 万人から 2016 年 138 万人へと約 6%も減少しており、典型的な“地方”現象に見舞われている。

事実､産業全体の視点でみると､愛媛県では 2005 年に約 3.4 兆だった売上高総計が 2013 年には 3.1 兆へと 7.5%も減少している一方で､企業数は 245 社から 246 社へと 1 社の増加にとどまっている（経済産業省「企業活動基本調査確報」※調査対象は従業者 50 人以上かつ資本金又は出資金 3,000 万円以上の会社のみ）。東京では同時期に、企業数が 7,430 社から 8,454 社へと約 14%も増加しているにもかかわらずである。

では、愛媛県には、どのような産業構造上の特性があるのだろうか。愛媛県で 2011 年に策定された「愛媛県産業振興指針（第二次改訂）」によれば、県内総生産上の比率でみると、サービス業（21.8%）、製造業（18.0%）の 2 つが基幹産業とされている。このうち製造業は、パルプ・非鉄金属・石油・石炭系等、基礎素材型産業が県内製造品出荷額の 55.7%を占めており、しかもそれらの多くが、（１）県外資本の大手企業によるものであり、（２）

製造品が県内で加工されず県外に中間素材として出荷されるという構造となっている。

（1）については、県外資本大手企業がグローバル戦略にともなって、中間素材の生産拠点を将来的に県外に移設する可能性があり、（2）については、加工製品の製造機会が県内製造業から奪われてしまっている。しかし、この構造は県外資本大手企業の経営判断に委ねられており、愛媛県の産業行政によって容易に解決できるものではないと思われる。

以上のことから、愛媛県において今後の成長分野は、観光産業、基礎素材型製造業以外に求める必要があり、かつ、（1）事業拡大が容易となるように設備投資等のコストが低い産業分野であること、（2）人材教育の利益還元率が高い産業分野であること、（3）愛媛県の立地・物流上の劣位に左右されない産業分野であることといった条件を満たす必要があると考えられる。

ここで再度、経済産業省「企業活動基本調査確報」の 2005 年と 2013 年のデータを比較すると、愛媛県では製造業の売上高が 1.71 兆円から 1.36 兆円へと約 20%も減少しているのに対して、飲食サービス業が 135 億から 202 億へと 50%の増加、情報通信業（2005 年段階では「情報サービス・情報制作業」と分類）が 288 億から 299 億へと 4%の増加となっている。他には小売業が 7,695 億から 8,138 億へと 6%程度の増加である。小売業にしても飲食サービス業にしても、売上を左右する商材、営業形態、立地条件は多種多様であり、ひとつの人材教育プログラムによって分野全体に大きな発展をもたらすことは難しいと思われる。

したがって、本事業では、愛媛県における新たな成長分野として IT 分野（とくにシステム開発分野）に焦点を絞りたい。IT 分野であれば、愛媛県内にすでに業界団体（愛媛県情報サービス産業協議会、愛媛県ＩＴ推進協会、愛媛ニアショア開発協議会）が組織され、産業の基盤が形成されているだけでなく、とくにシステム開発であれば、事業拡大に伴う生産設備コストを抑制できると同時に、立地・物流環境の影響を受けにくく、しかも、知的訓練による生産性の向上を期待しやすいからである。

1.2.2. IT 分野の成長可能性

しかし、IT 分野において企業の事業拡大の要因とは何だろうか。情報サービス産業協会の調査（「2014 年版 情報サービス産業基本統計調査」）では、IT 企業を 3 種（SI サービス型、ソフトウェア開発型、IT アウトソーシング/情報処理サービス型）に分類した上で、従業員 1 人あたりの売上高平均値、売上高成長率平均値等の比較を行っている。「SI サービス」とは、いわゆる超上流・上流（企画・コンサルティング・要求定義）から下流までを一括して提供するサービスであり、「ソフトウェア開発」は下流のみ（プログラム作成やソフトウェア保守）、「IT アウトソーシング/情報処理サービス」は、情報システムの管理運用やデータ処理サービスを意味している。この 3 種において、従業員 1 人あたりの売上高平均値は、SI サービス型が 2,861 万円、ソフトウェア開発型が 1,986 万円、IT アウトソーシング/情報

処理サービス型が 2,222 万円であり、売上高成長率平均値は、それぞれ 5.44%、2.59%、2.83%である。

ここから、IT 分野において利益や成長を促進する要因のひとつとして、いわゆる超上流・上流工程の技術を考えることができる。IPA の共通フレームの用語にしたがえば、業務要件定義、システム要件定義、ソフトウェア要件定義、システム方式設計、ソフトウェア方式設計といった技術である。このような技術を駆使する技術者は、近年では IT アーキテクトと呼ばれている。

他方、経済産業省「企業活動基本調査確報」（2013 年）によれば、情報通信業分野における従業員 1 人あたりの売上高は、東京都の 2,856 万円に対して愛媛県は 1,802 万円とされている。さらに、SI サービス型企業の平均従業員数は 1,148 人、ソフトウェア開発型が 504 人であるのに対して（「2014 年版 情報サービス産業基本統計調査」）、愛媛県内の IT 企業の平均従業員数は 166 人である（「企業活動基本調査確報」2013 年）。従業員・売上高の規模から、愛媛県内の IT 産業は SI サービス型が少ない、つまり、IT アーキテクト人材が少ないのではないかと考えられる。

1.2.3. 本事業を実施する意義

以上のことから、本事業では、愛媛県の IT 産業発展に寄与するべく、県内の IT エンジニア（主としてプログラマーが想定できる）を IT アーキテクトへと成長させる教育プログラムを開発するものとする。これにより、愛媛県内の IT 企業に、利益率の高いビジネス（SI サービス）へと事業展開するためのひとつの契機を提供したい。同時に、この教育プログラムは、類似地域 IT 産業への適用や標準的な IT アーキテクト育成プログラムの形成に役立つとともに、将来的に、近年議論されている「次世代高度 IT 人材」（IT サービスアーキテクト等の「異分野と IT の融合領域においてイノベーションを創出し、新たな製品やサービスを自ら生み出すことのできる人材」）の育成プログラム形成にも示唆を与えるはずである。

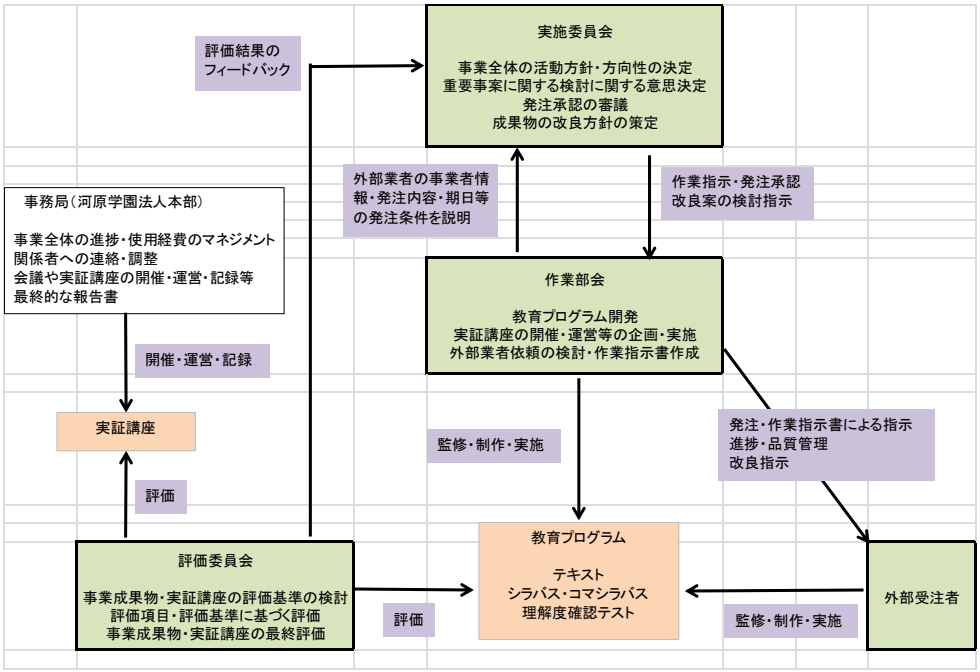
1.3. 事業の実施体制

1.3.1. 概要

本事業は、ふたつの委員会とひとつの下部組織から構成される。

まず、事業の推進主体として「実施委員会」を編成する。実施委員会の下部組織として、事業の実作業（外注作業の発注、作業指示や進捗・品質管理も含む）を機動的に担う「作業部会」を設置する。さらに、教育プログラムや実証講座の有効性・妥当性等の評価を担う「評価委員会」を設置する。

図 1（組織図）



1.3.2. 各委員会の役割

実施委員会は、事業全体の活動方針・方向性、実施内容等の重要事案に関する検討・意思決定、および、作業部会への作業指示を担う。

作業部会は、実施委員会の指示のもと、教育プログラム（テキスト・シラバス・コマシラバス・理解度確認テスト）開発、実証講座の開催・運営等の具体的内容の企画・実施を担う。また、各活動の過程において発生する作業を外部業者へ依頼する際の具体的内容の検討や発注に必要な作業指示書の作成も行う。これに基づき、実施委員会の審議を経て発注の承認を得る。作業部会は、発注後、作業完了に至るまでの進捗と品質の管理についても担当する。

ただし、外部業者の適切な候補選定が実施委員会開催時機に間に合わず、早急な発注を要する事案等に関して、実施委員会の審議を経ることが難しい場合には、実施委員会委員長に発注の承認を一任することがある可能性について、実施委員会において事前に同意を得ておくものとする。実際に実施委員会委員長に発注の承認を一任する事態が発生した場合には、実施委員会委員長に対して、作業部会が、外部業者の事業者情報、発注内容、期日等の発注条件を説明し、承認を得た上で発注を行うとともに、後日開催の実施委員会において事後承認を得るものとする。

評価委員会は、事業成果物および実証講座の評価項目・評価基準の検討およびそれに基づく評価を実施し、実施委員会にフィードバックを行う。このフィードバックに基づき、実施委員会は改良方針を策定し、作業部会に対して具体的な改良案の検討を指示する。さ

らに、評価委員会は、事業成果物および実証講座の最終的評価も担当する。

事務局（河原学園法人本部）は、事業全体の進捗および使用経費のマネジメントを行うとともに、関係者への連絡・調整、会議や実証講座の開催・運営・記録等の各種事務作業を担う。

1.3.3. 委員会・作業部会の構成

◆実施委員会の構成員（委員）

氏名		所属･職名	役割等	都道府県名
1	河原成紀	学校法人河原学園・理事長	統括・管理	愛媛県
2	鳥居高之	学校法人三橋学園船橋情報ビジネス専門学校校長	開発	千葉県
3	芦澤昌彦	学校法人河原学園・教務部長	開発・実証	愛媛県
4	神馬一博	学校法人河原学園河原電子ビジネス専門学校教頭	開発・実証	愛媛県
5	赤松民康	愛媛県情報サービス産業協議会	開発・実証	愛媛県
6	小林真也	愛媛県ＩＴ推進協会	開発・実証	愛媛県
7	大塚純孝	愛媛ニアショア開発協議会	開発・実証	愛媛県
8	中川隆	高知情報ビジネス＆フード専門学校	開発・実証	高知県
9	小澤慎太郎	中央情報経理専門学校高崎校	開発・実証	群馬県

◆評価委員会の構成員（委員）

氏名		所属･職名	役割等	都道府県名
1	梶原直樹	学校法人河原学園河原電子ビジネス専門学校教務部長	統括・管理	愛媛県
2	光永尚志	株式会社デーコム	評価・検証	東京都

◆実施委員会における下部組織（作業部会）の構成員（委員）

氏名		所属･職名	役割等	都道府県名
1	芦澤昌彦	学校法人河原学園・教務部長	統括・管理	愛媛県
2	井坂昭司	専門学校東京テクニカルカレッジ・情報処理科科長	開発	東京都
3	平澤一志	株式会社 ISID インターテクノロ	開発	東京都

		ジー		
4	神馬一博	学校法人河原学園河原電子ビジネス専門学校教頭	開発・実証	愛媛県

1.3.4. 実施委員会の開催

◆第１回実施委員会

日　時	2017 年８月 25 日（金）17：00～18：00
場　所	国際ホテル松山　　（愛媛県松山市一番町１－１３）
出席者	芦澤昌彦（学校法人河原学園・教務部長）、神馬一博（学校法人河原学園河原電子ビジネス専門学校教頭）、赤松民康（愛媛県情報サービス産業協議会）、中川隆(高知情報ビジネス&フード専門学校副校長)、小澤慎太郎(中央情報経理専門学校高崎校)、梶原直樹(学校法人河原学園河原電子ビジネス専門学校教務部長)、記録:江藤聡(学校法人河原学園河原電子ビジネス専門学校教員)
議　題	1.　委員長挨拶 2.　委員自己紹介 3.　事業計画の説明 (1)　平成 28 年度事業　概要説明 (2)　平成 29 年度事業　概要説明 4.　事業内容等についての意見交換 5.　今後のスケジュール 6.　交通費・謝金についての説明
配布資料	資料１　平成 29 年度「成長分野等における中核的専門人材養成等の戦略的推進」事業　事業計画書 資料２　平成 28 年度「成長分野等における中核的専門人材養成等の戦略的推進」事業趣旨・採択一覧 資料３　平成 29 年度「専修学校による地域産業中核的人材育成事業」　事業趣旨・概要 資料４　旅費等についての注意事項 資料５　平成 28 年度「地域 IT 産業振興のための IT アーキテクト育成事業」事業成果報告書

◆第２回実施委員会

日　時	2018 年 1 月 17 日（水）16：30～18：00
場　所	河原電子ビジネス専門学校　　（愛媛県松山市柳井町３丁目３－３１）
出席者	神馬一博（河原電子ビジネス専門学校 教頭)、鳥居高之（学校法人三橋学園船橋情報ビジネス専門学校校長)、中川隆(高知情報ビジネス&フード専門学校副校長)、小澤慎太郎(中央情報経理専門学校高崎校)、大塚純孝（愛媛ニアショア開発協議会）
議　題	1.　挨拶 2.　事業進捗状況報告と意見交換 3.　実証講座準備・運営・結果分析発注仕様書案の検討 4.　今後のスケジュール
配布資料	資料１　平成２８年度事業成果報告書 資料２　第２回実施委員会　進捗報告書 資料３　PBL テキストの構成 資料４　実証講座準備・運営・結果分析　発注仕様書案

◆第３回実施委員会

日　時	2018 年 2 月 13 日（火）16：30～18：00
場　所	河原電子ビジネス専門学校　　（愛媛県松山市柳井町３丁目３－３１）
出席者	神馬一博（河原電子ビジネス専門学校 教頭)、赤松民康（愛媛県情報サービス産業協議会)、小林真也（愛媛県ＩＴ推進協会）
議　題	1.　挨拶 2.　事業進捗報告 3.　実証講座検証・評価について 4.　実証講座アンケート、評価テストの結果について 5.　成果報告書印刷製本発注仕様書案の検討 6.　意見交換 7.　今後のスケジュール
配布資料	資料１　実証講座 PBL テキスト 資料２　（実証講座）事前アンケート 資料３　（実証講座）事後アンケート 資料４　（実証講座）理解度テスト１結果 資料５　（実証講座）理解度テスト２結果 資料６　成果報告書印刷製本発注仕様書案

1.3.5. 作業部会の開催

◆第 1 回作業部会

日　時	平成 29 年 10 月 18 日（水）16:00～18:00
場　所	東京テクニカルカレッジ会議室　（東京都中野区東中野 4-2-3）
出席者	芦澤昌彦（学校法人河原学園・教務部長）、井坂昭司（専門学校東京テクニカルカレッジ・情報処理科科長）、平澤一志（株式会社 ISID インターテクノロジー取締役）
議　題	1.　ご挨拶 2.　委員　自己紹介 3.　事業計画の説明 (1)平成 28 年度事業　概要説明 (2)平成 29 年度事業　概要説明 4.　事業内容（作業部会）についての意見交換 5.　今後のスケジュール 6.　交通費・謝金についての説明
配布資料	資料１　平成 29 年度「成長分野等における中核的専門人材養成等の戦略的推進」事業　事業計画書 資料２　平成 28 年度「成長分野等における中核的専門人材養成等の戦略的推進」事業趣旨・採択一覧 資料３　平成 29 年度「専修学校による地域産業中核的人材育成事業」　事業趣旨・概要 資料４　旅費等についての注意事項 資料５　2017 年度事業 テキスト修正案

◆第 2 回作業部会

日　時	平成 29 年 11 月 10 日（金）17:00～18:00
場　所	東京テクニカルカレッジ会議室　（東京都中野区東中野 4-2-3）
出席者	芦澤昌彦（学校法人河原学園・教務部長）、神馬一博（河原電子ビジネス専門学校・教頭）井坂昭司（専門学校東京テクニカルカレッジ・情報処理科科長）、平澤一志（株式会社 ISID インターテクノロジー取締役）
議　題	1.　挨拶 2.　事業進捗状況確認と意見交換 3.　要件定義、アーキテクチャ設計発注仕様案の再検討 4.　今後の作業内容 5.　今後のスケジュール

配布資料	資料１　システム開発要件案 資料２　要件定義、アーキテクチャ設計発注仕様案

◆第３回作業部会

日　時	2018 年 1 月 22 日（水）16：30～18：00
場　所	専門学校東京テクニカルカレッジ　　（東京都中野区東中野４丁目２−３）
出席者	神馬一博（学校法人河原学園河原電子ビジネス専門学校教頭）、井坂昭司（専門学校東京テクニカルカレッジ・情報処理科科長）、平澤一志（株式会社 ISID インターテクノロジー取締役）
議　題	1.　ご挨拶 2.　事業進捗状況報告と意見交換 3.　要件定義・アーキテクチャ設計テキストについて、評価委員会による評価を受けた修正作業の検討 4.　PBL テキストの内容確認 5.　実証講座準備・運営・結果分析発注仕様書案の検討 6.　今後のスケジュール
配布資料	資料１　第２回作業部会からの進捗報告書 資料２　要件定義・アーキテクチャ設計テキスト修正版 資料３　PBL テキスト現状版 資料４　実証講座準備・運営・結果分析　発注仕様書案

1.3.6. 評価委員会の開催

◆第１回評価委員会

日　時	2017 年 12 月 20 日（水）16：00～18：00
場　所	河原電子ビジネス専門学校　　（愛媛県松山市柳井町３丁目３－３１）
出席者	神馬一博（学校法人河原学園河原電子ビジネス専門学校教頭）、梶原直樹（河原電子ビジネス専門学校教務課長）、光永尚志（株式会社デーコム）、
議　題	1.　ご挨拶 2.　事業進捗状況確認 3.　要件定義・アーキテクチャ設計テキスト修正原稿の評価 4.　事業内容についての意見交換 5.　今後のスケジュール
配布資料	資料１　平成２９年度「専修学校による地域産業中核人材養成事業」事業計画書 資料２　成長分野等における中核的専門人材養成等の戦略的推進

	資料３　専修学校による地域産業中核的人材養成事業 資料４　平成２８年度事業成果報告書 資料５　要件定義・アーキテクチャ設計修正原稿 資料６　PBL テキストの構成

◆第 2 回評価委員会

日　時	2018 年２月９日（金）17：30～19：00
場　所	河原電子ビジネス専門学校　　（愛媛県松山市柳井町３丁目３－３１）
出席者	神馬一博（学校法人河原学園河原電子ビジネス専門学校教頭）、梶原直樹（河原電子ビジネス専門学校教務課長）、光永尚志（株式会社デーコム）、
議　題	1.　ご挨拶 2.　事業進捗報告 3.　実証講座検証・評価 4.　要件定義・アーキテクチャ設計テキスト、PBL テキストの評価 5.　意見交換 6.　今後のスケジュール
配布資料	資料１　実証講座 PBL テキスト 資料２　（実証講座）事前アンケート 資料３　（実証講座）事後アンケート 資料４　（実証講座）理解度テスト１結果 資料５　（実証講座）理解度テスト２結果 資料６　要件定義・アーキテクチャ設計テキスト修正版 資料７　PBL テキスト修正版

2. 事業活動の内容

2.1. 教育プログラムの方針

◆対象者の設定

本事業は、愛媛県内の IT 企業でシステム開発に携わり、かつ、超上流・上流工程（要件定義やアーキテクチャ設計）には直接携わった経験がないプログラマーを対象とする。ただし、超上流・上流工程の学習には、高度なプログラミング技術を前提とするため、複数の言語を習得している中堅以上のプログラマーで、将来、IT アーキテクトの業務を希望している者を対象とする。

◆知識領域の設定

IT アーキテクトの活動は、ビジネス領域とシステム領域を媒介するものであることから、「要件定義」、「アーキテクチャ設計」の 2 つの知識領域を扱う。いずれの知識領域にも、ソフトウェア工学上の著名な理論や経験的に有力だと評価されている方法論を導入し、IT アーキテクトに必要な幅広い視野の獲得を目指す。

◆PBL テキストの開発

PBL では、知識習得フェーズで獲得した理論・方法論の適用を訓練する。受講者を 2-3 グループに分割し、それぞれ同一の仮想開発プロジェクトによる演習を体験する。演習の実施については、知識習得フェーズで開発したテキストに準拠して演習用のテキストを新たに用いる。体験後には、結果の評価を受けるとともに、体験に伴って発見されたノウハウを演習用テキストに反映させる。

2.2. 成果物の概要

◆二種のテキスト

本事業では、愛媛県内の IT 企業の中堅プログラマーを対象に、IT アーキテクト育成の教育プログラムを開発する。28 年度には、要件定義とアーキテクチャ設計についてそれぞれ A４160 ページからなる教材の開発を行ったが、今年度の事業では、この成果を受けて、以下、二種のテキストの開発を行う。

（1）28 年度に開発した知識習得用テキストの拡充

現状では一般的な概念や抽象的な理論・方法論が中心となっており、具体的な開発事例への適用方法については説明が乏しい。そこで、具体的な開発事例に即して一般的な概念や抽象的な理論・方法論を適用する仕方についての記述を加える。

（2）PBL プログラムを記載した演習用テキスト

PBL プログラムの開発とともに、PBL プログラムのプロセスを遺漏なく記載した演習用テキストを開発する。この PBL プログラムは、要件定義からアーキテクチャ設計、さらには詳細設計に到るまでのプロセスを含み、それぞれの段階の成果物の完成度を確認しながら進行できるように設計する。

3. 実証講座の内容

3.1. 概要

3.1.1. 目的

本事業は、愛媛県内の情報産業振興に教育的側面から支援を行うものとして、IT アーキテクトの育成プログラムを開発するものであり、本プログラムは知識習得のフェーズと適用訓練のフェーズから構成される。

昨年度には、知識習得のフェーズを実行し、ソフトウェア工学上の著名な理論や経験的に有力だと評価されている方法論に裏打ちされたテキストの開発、シラバス・コマシラバスの開発、実証講座の実施と検証までを実施している。

本年度は、前年度開発されたテキストに対して事例への適用を含めた拡充、さらに、拡充されたテキストに準拠した仮想開発プロジェクト（PBL）による演習用テキストを開発した。

実証講座では、本プロジェクトで開発した仮想開発プロジェクトによる演習用テキストを用いた講座による、検証およびフィードバックを実施することを目的とする。

3.1.2. 受講者

（1）対象者

愛媛県内のプログラマー。ただし、超上流・上流工程（要件定義やアーキテクチャ設計）には直接携わった経験がないこと、Java や C#等、少なくとも一つのオブジェクト指向言語については業務レベルのコーディング能力をもったプログラマーであること、IT アーキテクト業務への関心を有している者を対象者としている。

（2）告知・募集

実証講座の告知・募集は、「愛媛県情報サービス産業協議会」「愛媛県ＩＴ推進協会」「愛媛ニアショア開発協議会」に協力を依頼し、「地域 IT 産業振興のための IT アーキテクト育成のための実証講座のご案内」を、会員企業に告知いただき、募集を募った。

また、河原電子ビジネス専門学校から、卒業生の就職先企業や求人企業に同案内を送付し、告知・募集を行った。

3.1.3. 実証講座の概要

実証講座は、以下の概要で実施した。

図表　1-1-1　実証講座の概要

開催日時	・1 日目　平成 30 年 2 月 7 日（水）10:00-17:00 ・2 日目　平成 30 年 2 月 8 日（木）10:00-17:00
対象	愛媛県内のプログラマーで、IT アーキテクト業務への関心を有している方
定員	20 名（先着順）
受講料	無料
会場	学校法人河原学園　河原電子ビジネス専門学校 〒790-0014　愛媛県松山市柳井町 3 丁目 3-31
講座内容	・RFP とは ・RFP と RFI ・RFP 作成の目的 ・RFP の作成方法（発行前アクティビティ） ・RFP の構造 ・提案を行う目的 ・シナリオの作成 ・ヒアリングの目的 ・ヒアリングにおける質問事項洗い出し ・ヒアリング実施 ・提案の作成 ・提案の発表 ・USDM とは ・USDM を用いた要求仕様書の作成 ・非機能要求の記述方法 ・品質特性駆動型設計とは ・アーキテクチャの設計 ・アーキテクチャの発表
形式	講義および演習：集合学習
担当講師	ワイズアイティ　代表　エンジニア　大政　雄也

3.1.4. 理解度確認テスト

（1）概要

本実証講座では、講座開始前の事前理解度確認テスト（事前テスト）と講座終了後の事後理解度確認テスト（事後テスト）で理解度確認テストを実施した。

（2）目的

理解度確認テストは、知識習得フェーズにおけるテキスト及び講座の効果を測定することを目的としている。

（3）実施方法

理解度確認テストは、Google フォームを使用して、事前テストと事後テストで同じ問題にて別に集計できるように実施した。

（4）効果測定

事前テストと事後テストにて同じ問題で出題し、正答数の差を計り効果を測定することとした。

あわせて、受講者の特性に応じた正答数などからの傾向等を測定することとした。

（5）設問内容

*必須

貴社名 *

氏名 *

前年度、本事業の講座を受講されましたか *

1 つだけマークしてください。

- はい
- いいえ

1. 次の文章のうち、間違っているものを選びなさい。(3.RFP)

1 つだけマークしてください。

- RFP とは、「取得者」が「供給者」 に「提案」を求める正式な依頼を表す。
- 取得者の要求のすべてのケースに RFP を用いることが望ましい。
- RFP を使うと、ある供給者を他の供給者より優遇したりせず、全員を同じルールと要求のもとで公平に競争させることができる。

類似しているが異なったソリューションが提案されるので、評価が容易になる。

2. 次の文章のうち、間違っているものを選びなさい。(3.RFP)

1 つだけマークしてください。

RFP が最終的に供給者に選ばれた理由について、それを明確に示すことは大変重要である。

RFP を供給者に発送する前に、RFP プロジェクトチームの主要メンバー以外の人々によって RFP をレビューすることが望ましい。

RFP プロジェクトチームでは、評価チームを置くことが推奨されている。船頭が多くなった場合に、最終的に異なるステークを持って評価に望んでしまうことがあり、評価基準策定が難しくなるためである。

システムを構築する費用は、供給者選定後に話合いで決める。

3. 次の文章のうち、間違っているものを選びなさい。(4.提案)

1 つだけマークしてください。

１次提案は、初回訪問から数回程度の打ち合わせの間に行う提案活動のことを表し、RFP をもらうことが目的である。

２次提案は、１次提案で承認されなかった箇所を修正して再提案するものである。

提案には単に顧客から求められた要求だけを満たしているのではなく、それ以上の付加価値が提示されていることが望ましい。

提案には、RFP で求められている項目に対して、すべて回答している（漏れなく）必要がある。

7. 次の文章のうち、間違っているものを選びなさい。(4.提案)

1 つだけマークしてください。

システム構成についても提案時点で詳細まで完成させておかなければならない。

「提案の趣旨」の構成要素としては、「提案の特徴」と「RFP に対する回答」の二つが挙げられる。

見積もりに関しては「システム一式」で総額だけ提示するのではなく、詳細な項目ごとの見積もりを提示すべきである。

提案対象領域を定義し、提案対象に関する認識のズレがないことを確認させ、食い違いが発生しないようにする。

8. 次の文章のうち、間違っているものを選びなさい。(4.提案)

1 つだけマークしてください。

ヒアリングの目的は、RFP には記載されていない、取得者の真に解決したい問題を発見する事である。

ヒアリングにおいては、まずは自社の技術の棚卸しを行い、RFP の内容と関連がありそうな部分と、それらを「提案」として盛り込むべきなのか否かを判断

する。

- [x] 独自性の高い提案が相手のニーズに沿うものであるか不明である場合、ヒアリング項目には盛り込むべきではない。
- [ ] 自チームの技術的な強みを棚卸した後、それを活かした提案をブレインストーミングする。ここでは、実現可能性や、クライアントのニーズに関しては考慮せずに、一旦思いつく限りのアイディアを出す。

9. 次の文章のうち、間違っているものを選びなさい。(5.要件定義)

1 つだけマークしてください。

- [ ] USDM 表記法では、まずシステムへの要求を明確にする。そして、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。
- [ ] USDM 表記法では、要求ごとにその要求がもたらされた理由をセットで記述する。
- [ ] USDM 表記法では、説明は非常に重要であり、要求と理由に加えて説明を付与することで内容理解が深まる。
- [x] USDM 表記法では、要求がもたらされた理由 1 つに対し 1 つずつ仕様を抽出していく。

10. 次の文章のうち、アーキテクチャ設計を行う事で期待できる効果として間違っているものを選びなさい。(6.アーキテクチャ設計)

1 つだけマークしてください。

- [ ] アプリケーション全体構造や、データ構造が明確化、そして言語化される事で、利害関係者の共通言語が出来るため効率的なコミュニケーションを行うことができる。
- [ ] 外部とのインタフェースが定義され、アプリケーションの担う責務の範囲が明らかとなる事で、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できる。
- [x] 実装をするコンポーネントのインタフェースや、役割などが明らかとなるため、実装の段階で必要な情報を全て洗い出すことが可能となり、実装時に生じるバグを完全に防ぐことができる。
- [ ] アプリケーションの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ことができる。

11. 次の文章のうち、アーキテクチャの成果物において期待される事柄として間違っているものを選びなさい。(6.アーキテクチャ設計)

1 つだけマークしてください。

- [ ] システム要件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。

- [ ] 全てのシステム構成要素が適切なサブ式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。
- [ ] システム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。
- [x] 実装を行う言語は何か。データベースは何に使うのかといった開発可能なレベルまで詳細化・具体化された仕様が完成しており、それぞれの利害関係者からの承認を得ていること。

12. アーキテクチャにおける「構造」と「ビュー」に関する文章として正しいものをすべて選びなさい。(6.アーキテクチャ設計)

当てはまるものをすべて選択してください。

- [x] 「構造」とはアーキテクチャをある視点から捉える切り口で、「ビュー」はそれを人が理解できる形で表現するもの。
- [ ] 「モジュール構造」は、ソフトウェアアーキテクチャの動作時の振る舞いを捉えるための構造。
- [x] 「割り当て構造」は、システム環境など非ソフトウェア構造との関連を捉えるための構造。
- [ ] 「コンポーネント&コネクター構造」は「分割構造」、「使用構造」、「レイヤー構造」などからなるシステムの静的な側面を捉えるための構造。

13. 次のうちソフトウェアアーキテクチャを表す「割り当て構造」が表すべき内容として間違っているものを選びなさい。(6.アーキテクチャ設計)

1 つだけマークしてください。

- [ ] 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。
- [ ] 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。
- [ ] 開発チームに割り当てるソフトウェアコンポーネントの実装と結果に対する責務は何か
- [x] 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。

3.1.5. アンケート

（1）概要

本実証講座では、講座開始前の事前アンケートと講座終了後の事後アンケートを実施した。

（2）目的

講座開始前のアンケートは、受講者の基礎情報及び業務経験や技能を確認するとともに講座への期待度などの情報収集を目的とした。

講座終了後のアンケートは、受講者の主観による講座内容の難易度・活用度・理解度・満足度・意見を集計することで、本プロジェクトで開発した仮想開発プロジェクト（PBL）による演習用テキストを用いた講座の効果を測定することを目的とした。

また、効果に影響を与えうる講師や講座環境について、確認のために質問を実施した。

3.2. 実証講座の実施

3.2.1. 受講者

受講者の情報は、講座開始前に実施した事前アンケートにおいて取得し、想定していた「愛媛県内のプログラマー」「Java や C#等、少なくとも一つのオブジェクト指向言語について業務レベルのコーディング能力をもったプログラマーであること」を概ね満たしていることが確認できた。

（1）受講者人数

応募定員に 20 名に対して、実証講座の受講者は 15 名であった。

（2）受講者の性別

受講者の性別は、男性 13 名、女性 2 名であった。

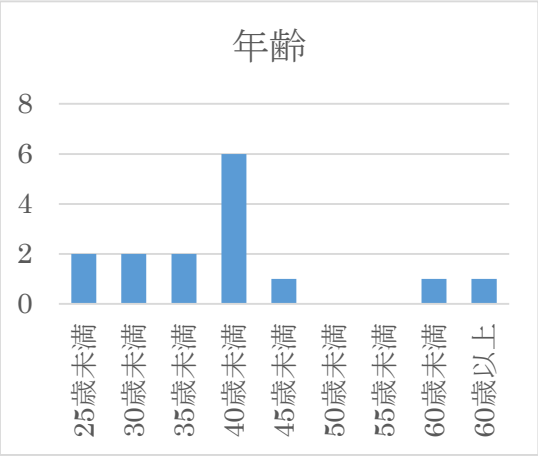
（3）受講者の年齢

受講者は､35 歳以上 40 歳未満の 6 名を中心に 21 歳から 62 歳までの広い年齢幅で集まったことが確認できる。

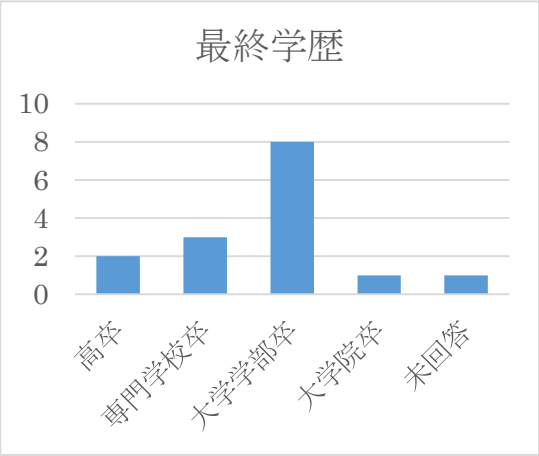
（4）受講者の最終学歴

受講者の最終学歴は、大学学部卒の 8 名を中心に、高卒、専門学校卒、大学院卒が集まった。

図表　1-2-1　受講者の年齢



図表　1-2-2　受講者の最終学歴



（５）所属している企業の勤務地

受講者の内 2 名を除き 13 名が愛媛県内の IT 企業にて勤務していることが確認できた。

（６）所属している企業の規模

受講者が所属している企業は、おおむね 50 人未満の企業規模であることが確認できた。

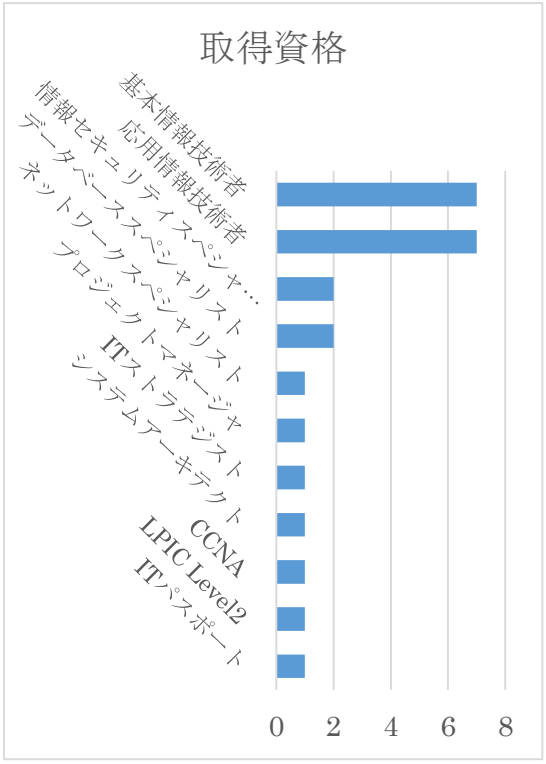
（７）受講者の取得資格

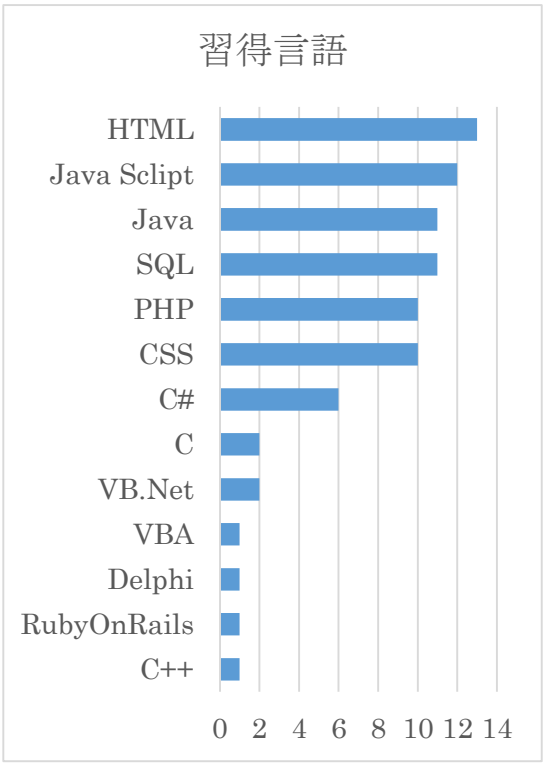
受講者は、2 名を除く 13 名が IT 分野の資格を取得しており、一定の知識を有していることが確認できる。

（８）受講者の習得言語

受講者は、未回答の 1 名を除き、14 名が複数のプログラミング言語を習得しており、Java や C#などのオブジェクト指向言語を概ね習得していることが確認できる。

図表　1-2-3　受講者の取得資格





図表　1-2-4　受講者の習得言語

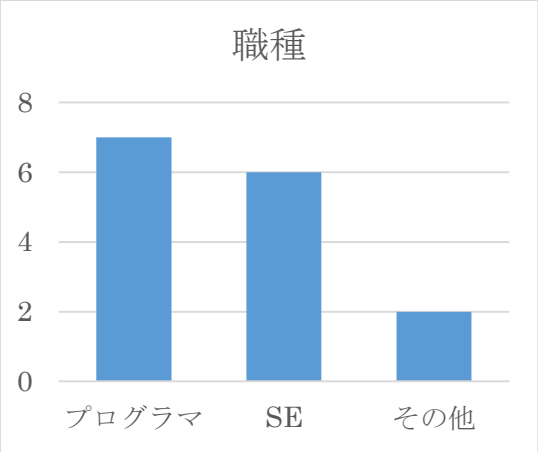
（９）受講者の現在の職種

受講者の現在の職種は、プログラマー7 名、SE6 名、その他 2 名であった。

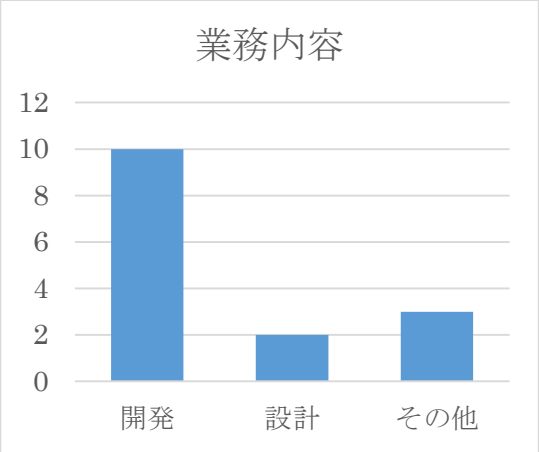
（１０）受講者の業務内容

受講者の内 10 名が開発を現在の業務としており、2 名が設計、その他が 3 名であった。

図表　1-2-5　受講者の職種



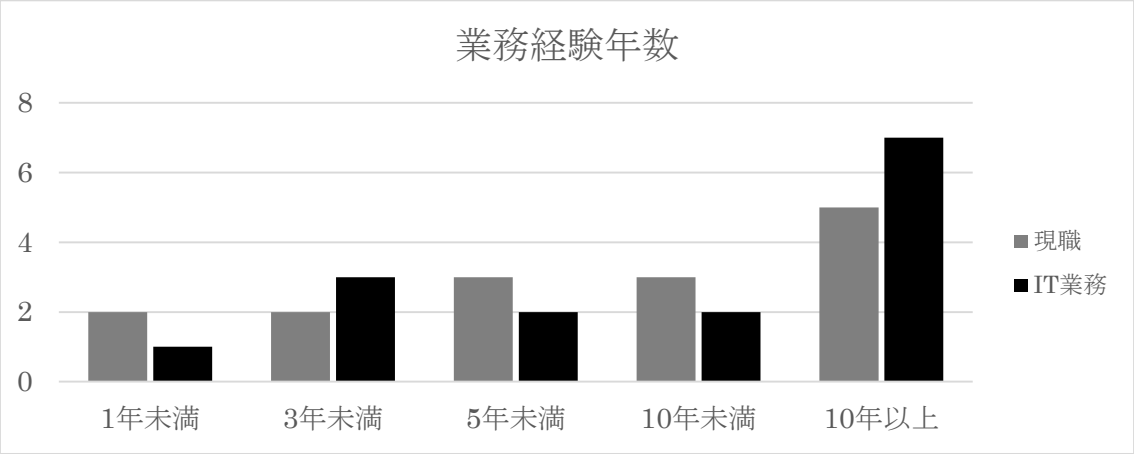
図表　1-2-6　受講者の業務内容



（１１）受講者の業務の経験年数

受講者の現職の経験年数および IT 業務の経験年数は、1 年未満、3 年未満、5 年未満、10 年未満、10 年以上で広く分散しており、10 年以上の経験年数がある中堅からベテランの受講者が集まったことが確認できる。

図表　1-2-7　受講者の業務経験年数

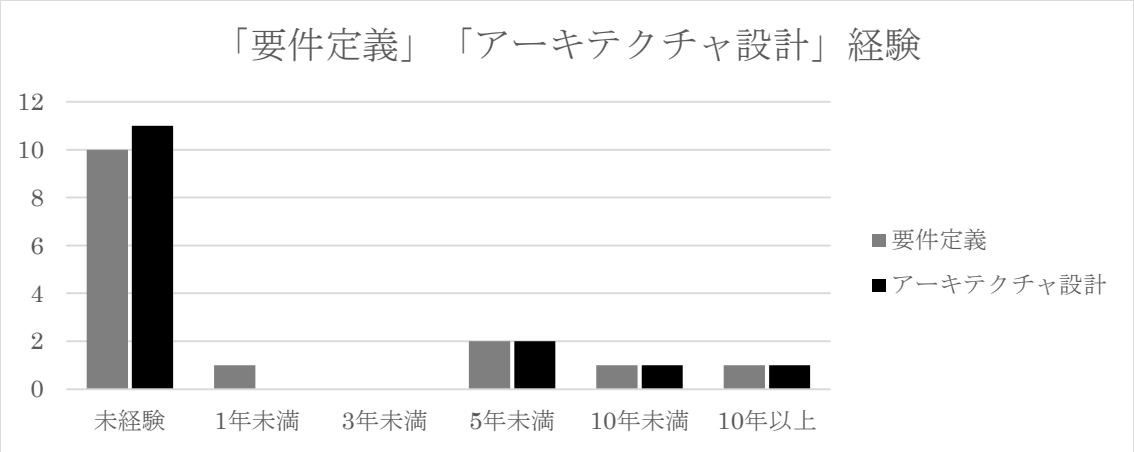


（１２）「要件定義」「アーキテクチャ設計」の経験

受講者の「要件定義」の経験は 10 名が未経験、「アーキテクチャ設計」の経験は 11 名が

未経験と回答しており、受講者の内 2/3 が今回の講座内容の業務に携わっていないことが確認できる。

図表　1-2-8　「要件定義」「アーキテクチャ設計」の経験



3.2.2. 実施内容

（1）講座内容

実証講座は、2 日間にわたり計 12 時間（90 分×4 コマ×2 日）、集合形式の講義および演習にて実施された。

図表　1-2-9　実証講座の内容

時間		内容
一日目	10:00-11:30	・事前アンケートおよび事前理解度確認テスト ・RFP と RFI ・RFP を作成する目的
	11:20-12:50	・RFP の作成方法（発行前アクティビティ） ・RFP の構造 ・「提案」を行う目的 ・大枠のシナリオを作成する
	12:50-13:50	昼食
	13:50-15:20	・ヒアリングを行う目的 ・ヒアリングにおける質問事項の洗い出し ・ヒアリングの実施 ・ヒアリングの振り返り
	15:30-17:00	・提案の作成 ・提案発表 ・提案発表の振り返り
二日目	10:00-11:30	・一日目の振り返り ・USDM とは ・USDM を用いた要求仕様書の作成
	11:20-12:50	・USDM を用いた要求仕様書の作成（続き）
	12:50-13:50	昼食
	13:50-15:20	・USDM を用いた要求仕様書のレビュー ・非機能要求の記述方法 ・非機能要求仕様の作成
	15:30-17:00	・アーキテクチャ設計における品質特性駆動形設計とは ・アーキテクチャを設計する ・事後アンケートおよび事後理解度確認テスト

（２）アンケートおよび理解度確認テストの実施

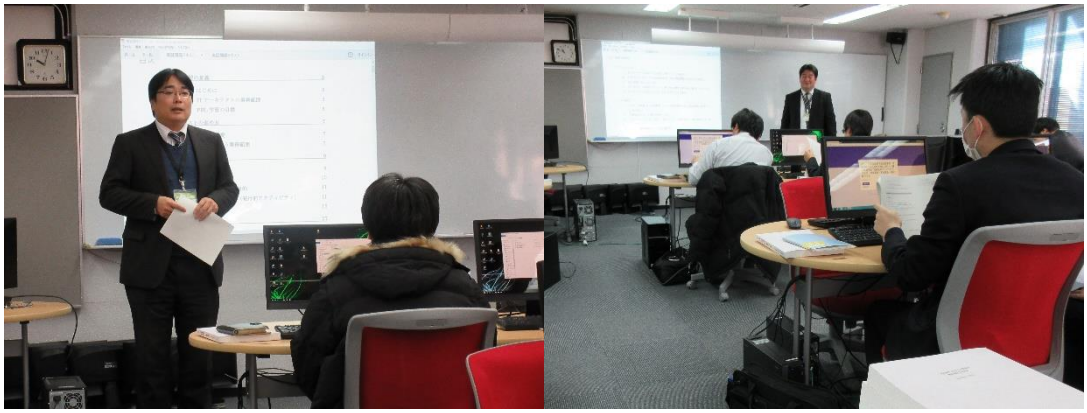
実証講座の開始時には事前アンケートと事前理解度確認テストを実施した。また、実証講座の終了時には事後アンケートと事後理解度確認テストを実施した。アンケートと理解度確認テストは、Google フォームによって実施した。

（３）講義と演習

実証講座は、講義と演習で構成され、知識習得のフェーズでは講義、適用訓練のフェーズでは演習が実施された。

演習では、仮想開発プロジェクトの「ヒアリング」「提案書の作成」「提案」「要求仕様書の作成」「アーキテクチャ設計」においてグループワークにて実施された。

図表　1-2-10　実証講座 1 日目　挨拶と授業風景



図表　1-2-11　実証講座 1 日目　グループワーク



図表　1-2-12　実証講座１日目　ヒアリング





図表　1-2-13　実証講座１日目　ヒアリングの振り返り



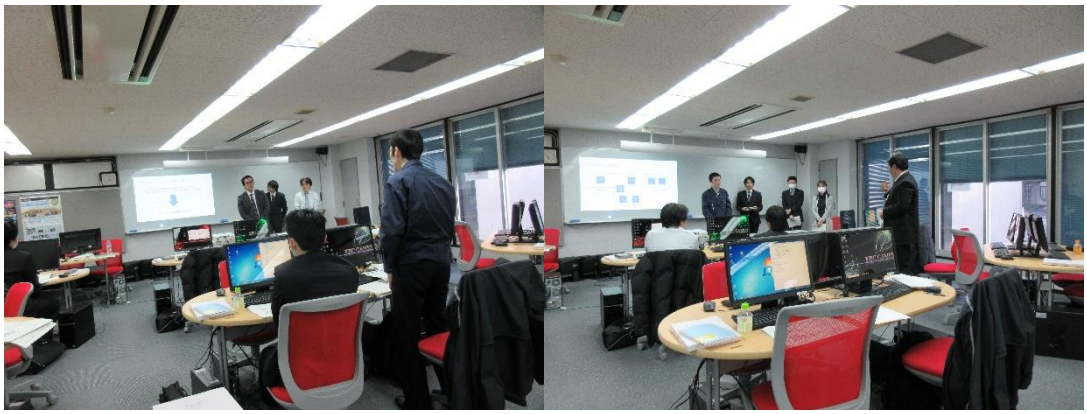
図表　1-2-14　実証講座 1 日目　提案書作成



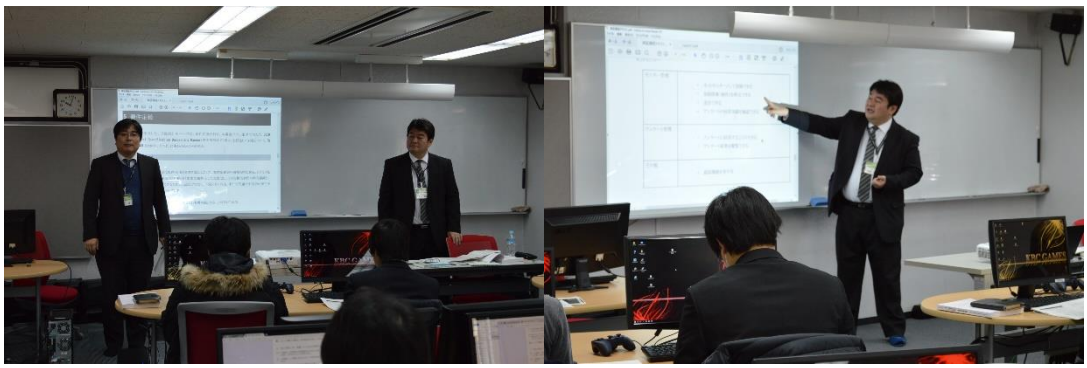
図表　1-2-15　実証講座 1 日目　提案



図表　1-2-16　実証講座 1 日目　提案質疑応答



図表　1-2-17　実証講座 2 日目　挨拶と授業風景



図表　1-2-18　実証講座 2 日目　USDM の記述



図表　1-2-19　実証講座 2 日目　USDM の記述（グループワーク）



図表　1-2-20　実証講座 2 日目　USDM 記述内容のレビュー



図表　1-2-21　実証講座 2 日目　アーキテクチャ設計



図表　1-2-22　実証講座 2 日目　終了挨拶



3.3. 実証講座の結果

3.3.1. 事前理解度確認テストの結果

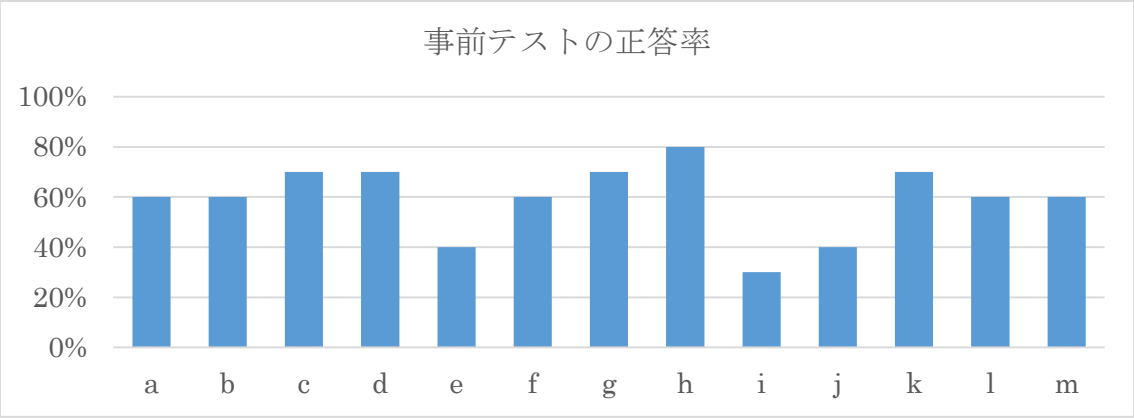
（1）事前理解度確認テスト

講座開始前に実施した事前理解度確認テスト（事前テスト）は、受講者 15 名中 13 名がが受験した。

（2）事前理解度確認テストの正答率

事前テストの平均正答率は 59.2%（平均正答数が 5.92 点）、最高得点者の正答率は 80%（正答数が 8 点）、最低得点者の正答率は 30%（正答数が 3 点）であった。

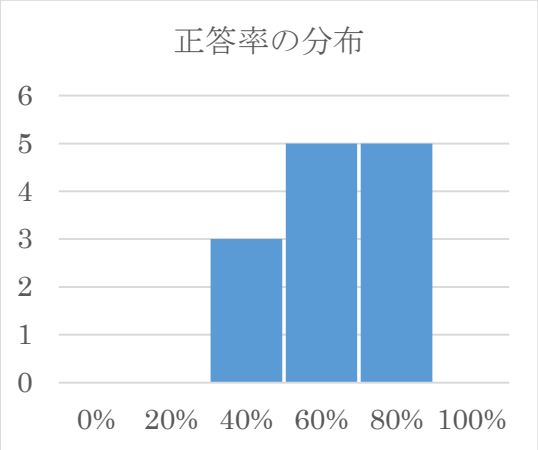
図表　1-3-1　事前テストの正答率



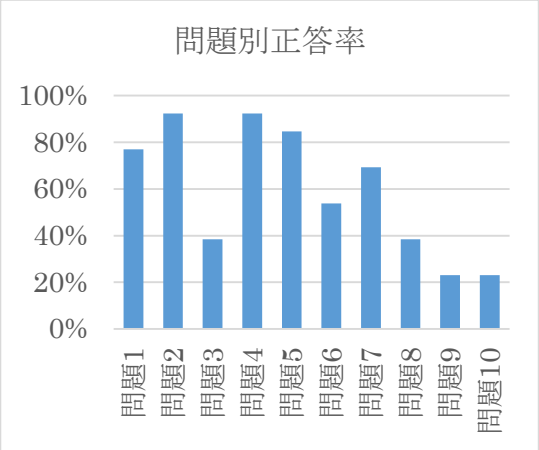
（3）事前理解度確認テストの正答率分布

事前テストの正答率分布は、80%未満と 60%未満をピーク（受験者 13 名中 5 名ずつ）に 40%未満から 80%未満の間で分布されていることが確認できる。

図表　1-3-2　事前テストの正答率分布



図表　1-3-3　事前テストの問題別正答率



（4）事前理解度確認テストの問題別正答率

事前テストの問題別正答率は、問題 2 と問題 4 で 92.3%（受験者 13 名中 12 名が正解）、問題 9 と問題 10 で 23.1%（受験者 13 名中 3 名が正解）と問題による正答率の差が大きくあることが確認できる。

3.3.2. 事後理解度確認テストの結果

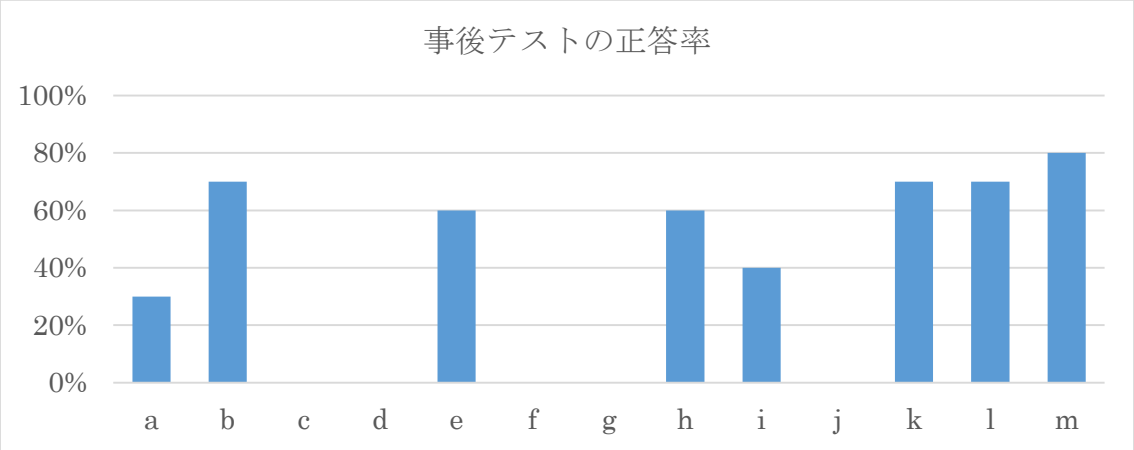
（1）事後理解度確認テスト

講座終了後に実施した事後理解度確認テスト（事後テスト）は、受講者 15 名中 8 名が受験した。

（2）事後理解度確認テストの正答率

事後テストの平均正答率は 60%（平均正答数が 6 点）、最高得点者の正答率は 80%（正答数が 8 点）、最低得点者の正答率が 30%（正答数が 3 点）であった。

図表　1-3-4　事後テストの正答率



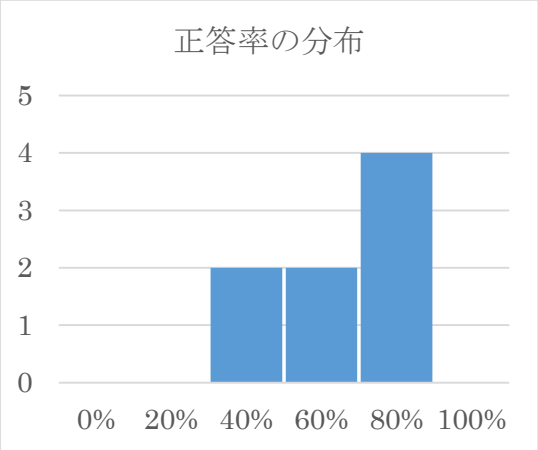
（2）事後理解度確認テストの正答率分布

事後テストの正答率分布は、80%未満をピーク（受験者 8 名中 4 名が正解）に 40%未満から 80%未満の間で分布されていることが確認できる。

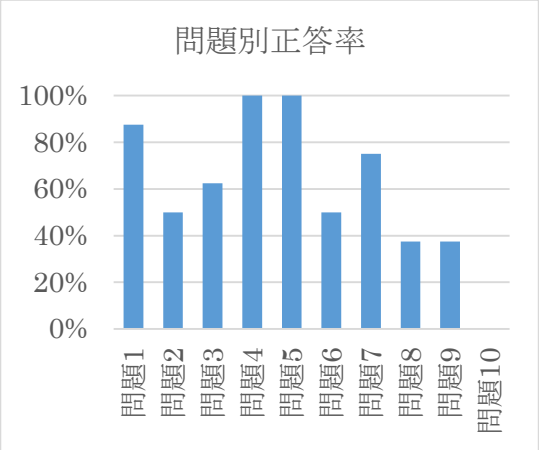
（3）事後理解度確認テストの問題別正答率

事後テストの問題別正答率は、問題 4 と問題 5 で 100%（受験者 8 名全員が正解）、問題 10 は 0.0%（受験者 8 名全員が不正解）と問題による正答率の差が大きくあることが確認できる。

図表　1-3-5　事後テストの正答率分布



図表　1-3-6　事後テストの問題別正答率



3.3.3. 理解度確認テストの比較

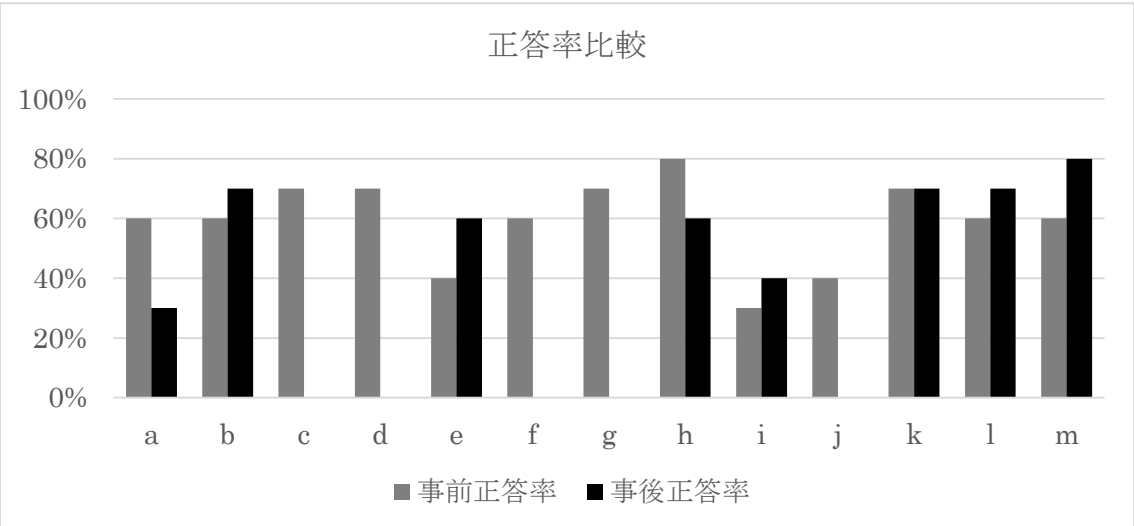
（1）理解度確認テストの受験状況

事前テストの受験者が 13 名（受講者 15 名中）であったのに対して、事後テストの受験者は 8 名であった。

（2）理解度確認テストの正答率比較

事前テストと事後テストの平均正答率は 0.8%の微増であることが確認できた。また、両テストの受験者 8 名の内 5 名の正答率に 10%から 20%の上昇が確認できる。

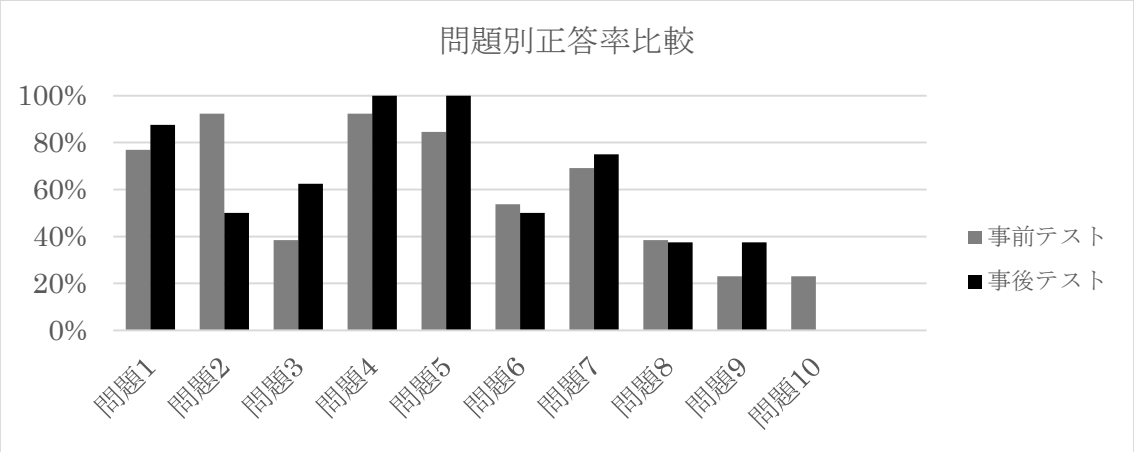
図表　1-3-7　理解度確認テストの正答率比較



（３）理解度確認テストの問題別正答率比較

問題別の正答率では、10 問中の 6 問で正答率が上昇していることが確認できた。一方で、10 問中 4 問で正答率が下がっており、その内の 1 問は正答率が 0%になっていることが確認できる。

図表　1-3-8　理解度確認テストの問題別正答率比較

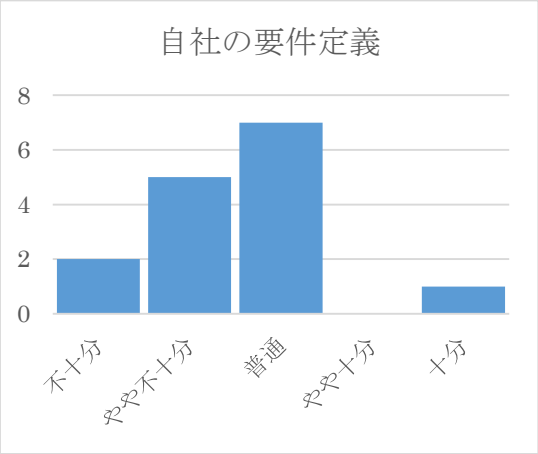


3.3.4. 事前アンケートの結果

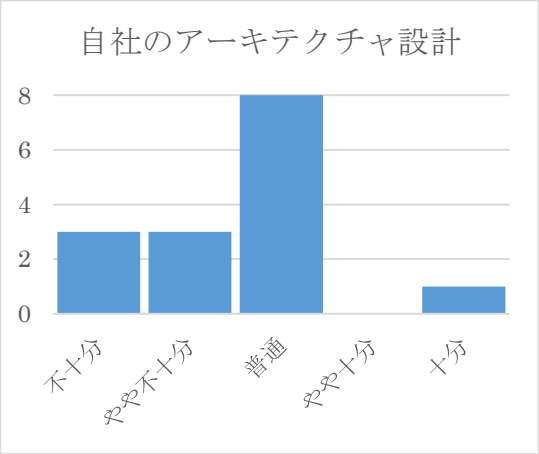
（１）自社の「要件定義」「アーキテクチャ設計」の妥当性・効率性

「自社の要件定義は妥当的・効率的に行われていると思いますか？」の設問に対して、受講者 15 名中 14 名が「普通」以下の評価であった。また、「自社のアーキテクチャ設計は妥当的・効率的に行われていると思いますか？」の設問に対しても受講者 15 名中 14 名が「普通」以下の評価であった。

図表　1-3-9　自社の要件定義



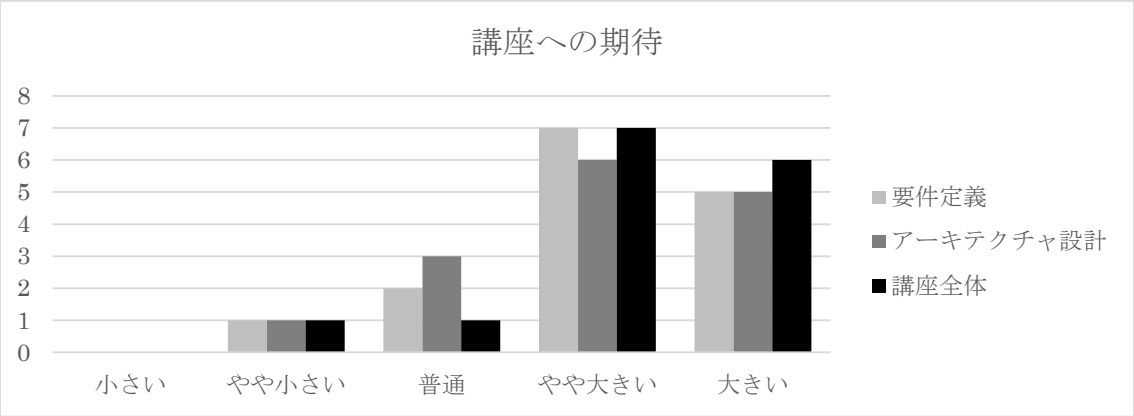
図表　1-3-10　自社のアーキテクチャ設計



（２）講座への期待

本講座への期待は、「要件定義」「アーキテクチャ設計」「講座全体」で 1 名を除く 14 名が「やや大きい」をピークにして「普通」以上に回答していることが確認できる。

図表　1-3-11　講座への期待



3.3.5. 事後アンケートの結果

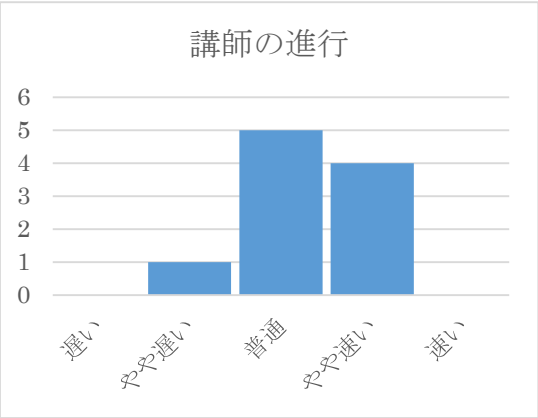
（１）事後アンケート

実証講座終了後に実施した事後アンケートは受講者 15 名の内 10 名の回答があった。

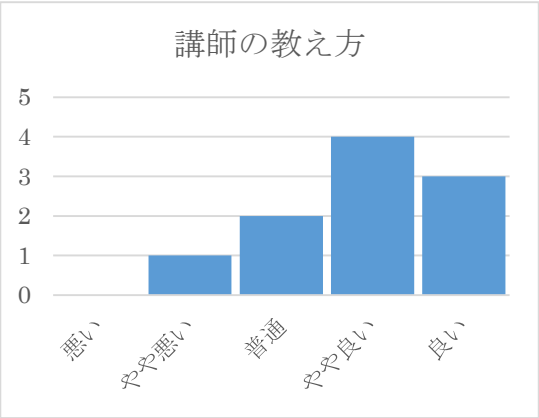
（２）講師について

講師の進行速度は、「普通」をピークに「やや遅い」から「やや速い」に回答している。また、講師の教え方は、1 名を除いて 9 名が「普通」以上に回答している。

図表　1-3-12　講師の進行速度



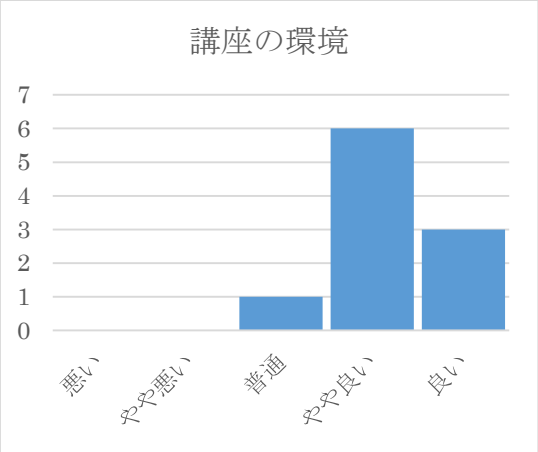
図表　1-3-13　講師の教え方



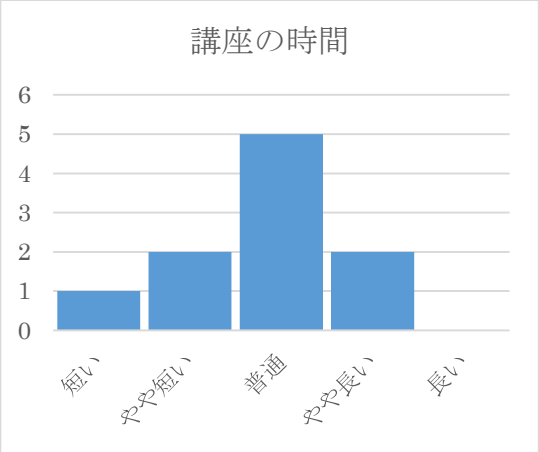
（３）講座について

講座の環境について、「やや良い」をピークに回答者全員（10 名）が「普通」以上に回答している。また、講座の時間については、1 名を除いて 9 名が「普通」をピークに「やや短い」「普通」「やや長い」に回答している。

図表　1-3-14　講座の環境



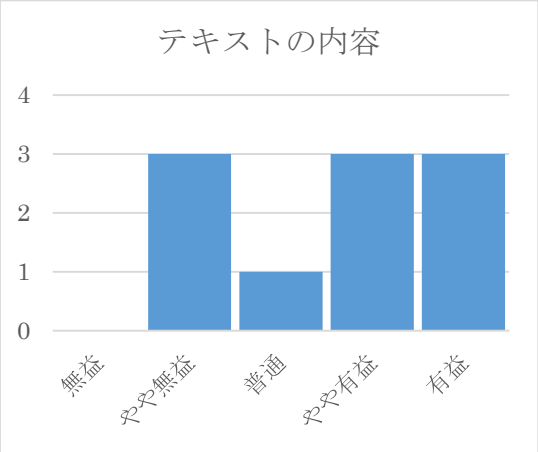
図表　1-3-15　講座の時間



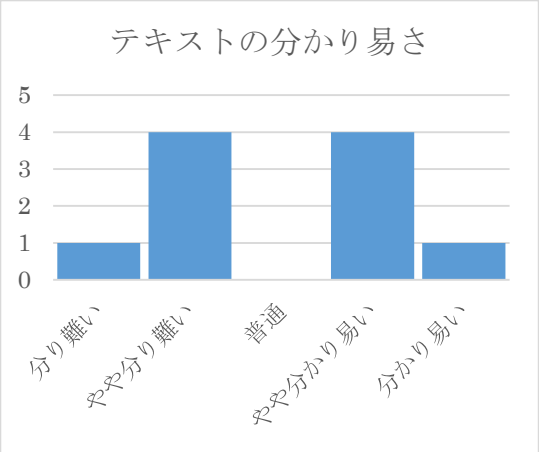
（４）テキストについて

テキストの内容は、3 名が「やや無益」と回答しているものの、7 名が「普通」から「有益」に回答している。また、テキストの分かり易さは、「分り難い」「やや分り難い」と「やや分かり易い」「分かり易い」に回答が 2 分されている。

図表　1-3-16　テキストの内容



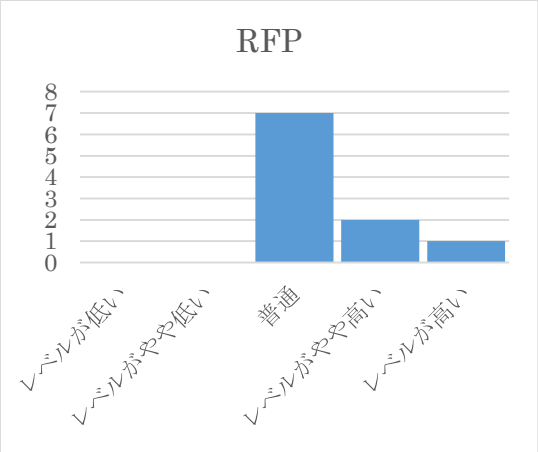
図表　1-3-17　テキストの分かり易さ



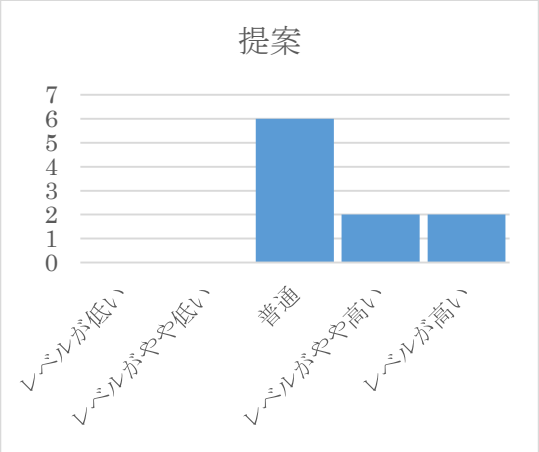
（５）講座の難易度

講座の難易度は、「RFP」「提案」「要件定義」「アーキテクチャ設計」の単元ごとに分布は異なるが、10 名の回答者全員が「普通」「レベルがやや高い」「レベルが高い」に回答している。

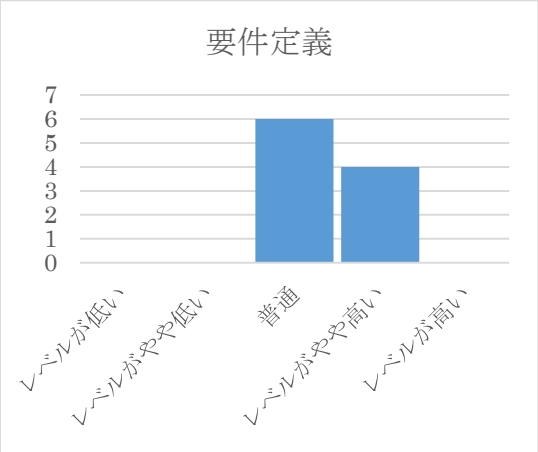
図表　1-3-18　RFP の難易度



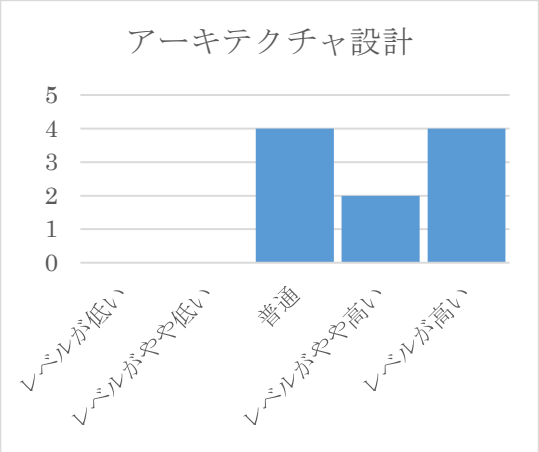
図表　1-3-19　提案の難易度



図表　1-3-20　要件定義の難易度



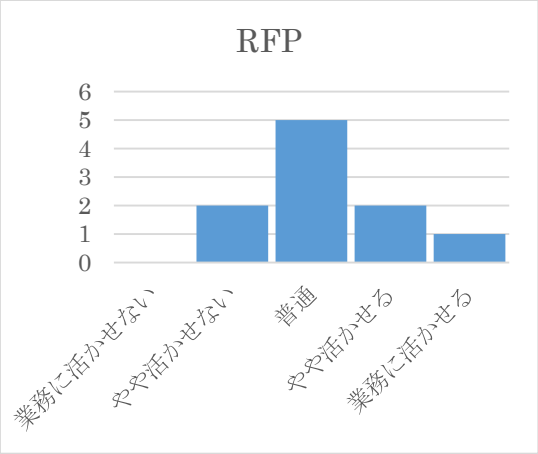
図表　1-3-21　ｱｰｷﾃｸﾁｬ設計の難易度



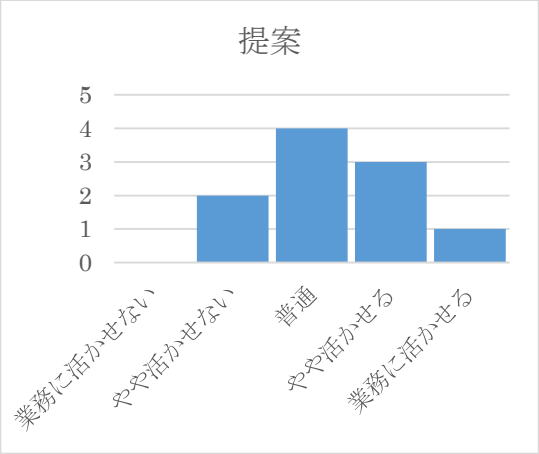
（６）講座の活用度

講座内容の単元ごとの活用度は、「RFP」と「提案」は「普通」をピークに「やや活かせない」から「業務に活かせる」までに回答している。「要件定義」は「やや活かせる」をピークに 10 名の回答者全員が「普通」以上に回答している。「アーキテクチャ設計」は「やや活かせない」をピークに回答されている。

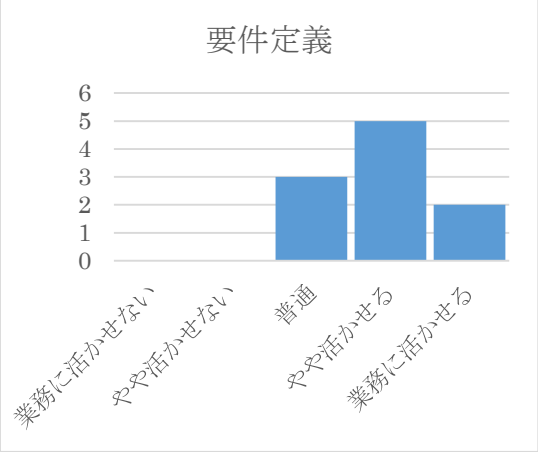
図表　1-3-22　RFP の活用度



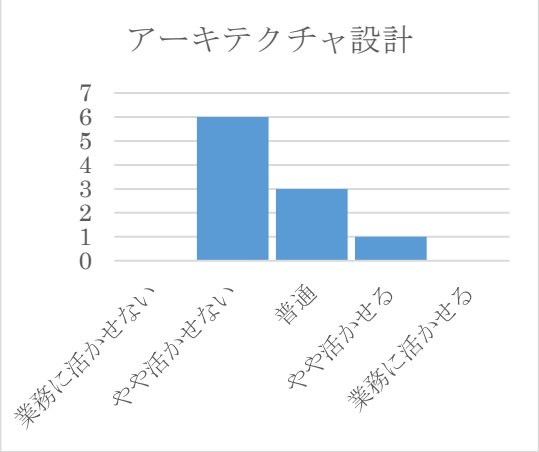
図表　1-3-23　提案の活用度



図表　1-3-24　要件定義の活用度



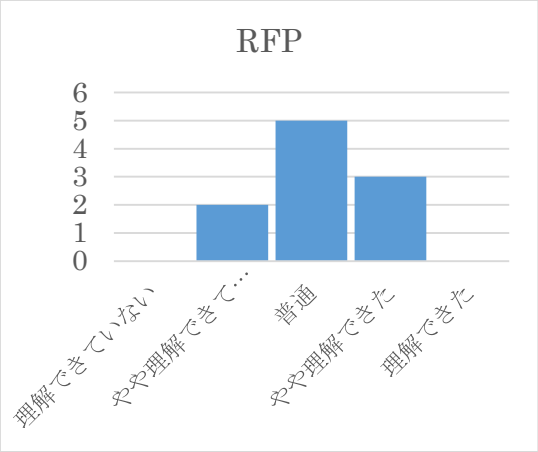
図表　1-3-25　アーキテクチャ設計の活用度



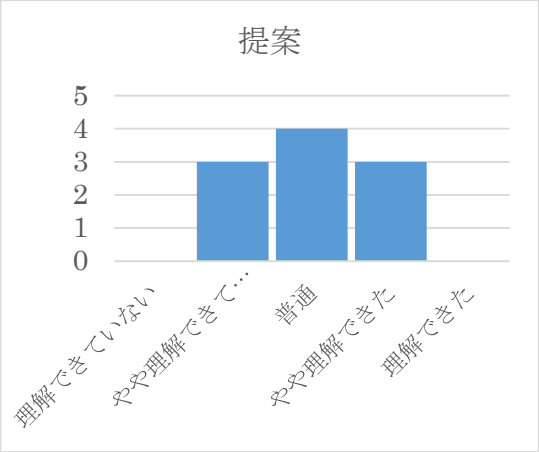
（７）講座の理解度

講座内容の単元ごとの理解度は、「RFP」と「提案」は「普通」をピークに「やや理解できていない」から「やや理解できた」に全員が回答している。「要件定義」は「やや理解できていない」と「やや理解できた」にピークがあり回答に広がりがみられる。「アーキテクチャ設計」は「やや理解できてない」をピークに全員が「普通」以下に回答している。

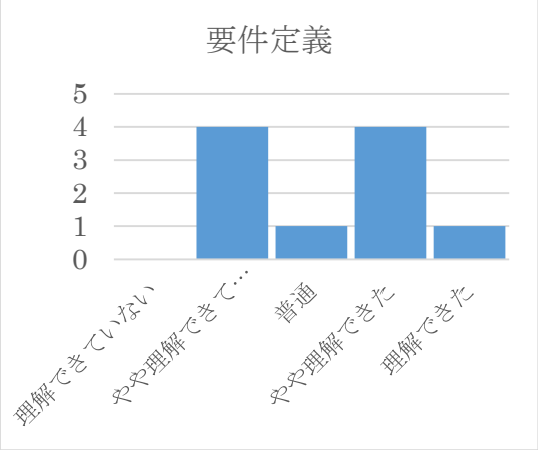
図表　1-3-26　RFP の理解度



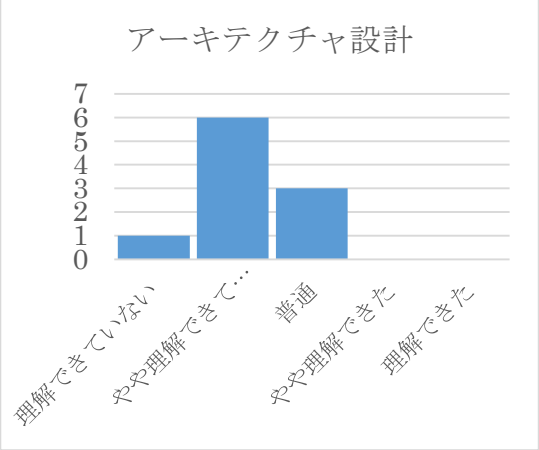
図表　1-3-27　提案の理解度



図表　1-3-28　要件定義の理解度



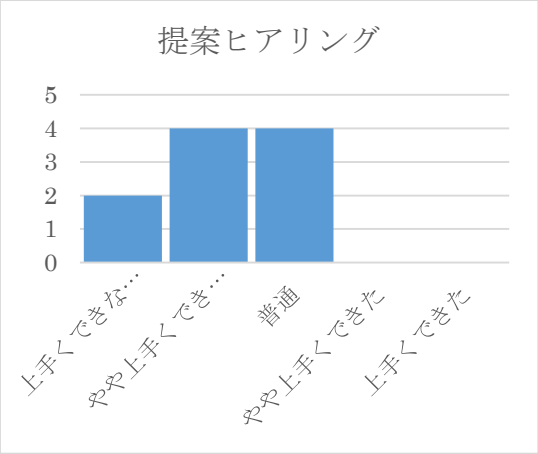
図表　1-3-29　アーキテクチャ設計の理解度



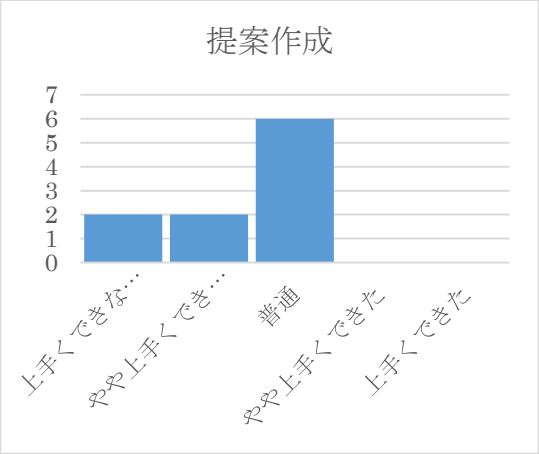
（８）演習の自己評価

実証講座内で実施された単元ごとの演習の自己評価は、「提案ヒアリング」「提案作成」「提案発表」「アーキテクチャ設計の課題」では回答者全員が「普通」以下に回答しており、「USDM の課題」では、「普通」をピークに「やや上手くできていない」から「上手くできた」に回答されている。

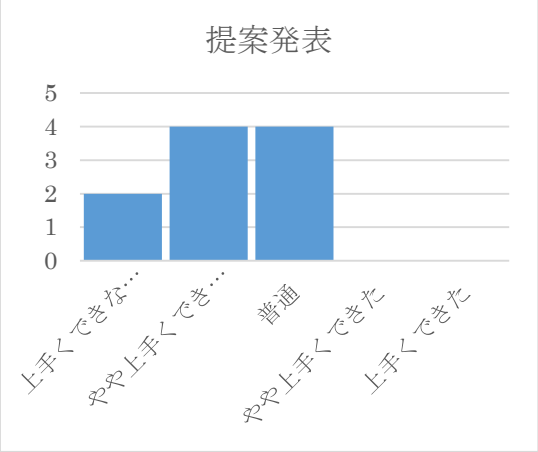
図表　1-3-30　提案ﾋｱﾘﾝｸﾞの自己評価



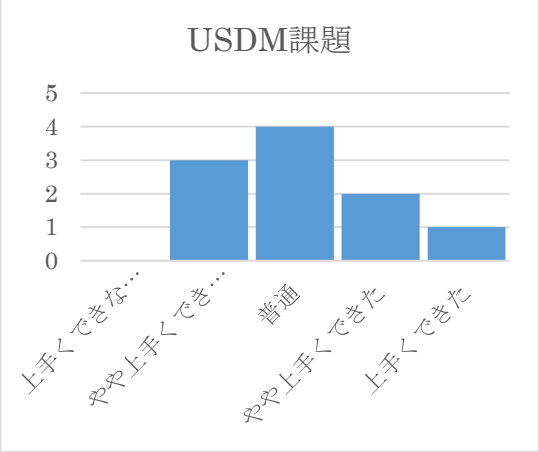
図表　1-3-31　提案作成の自己評価



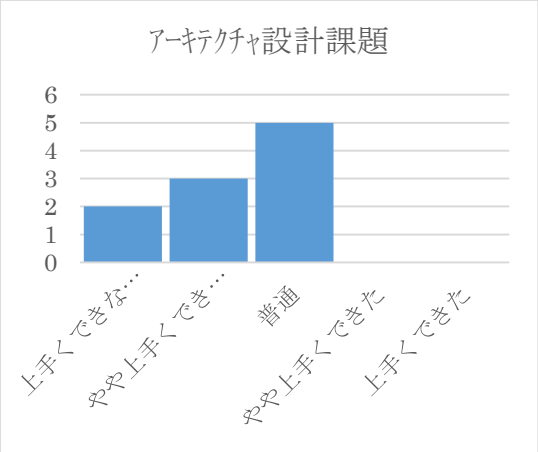
図表　1-3-32　提案発表の自己評価



図表　1-3-33　USDM 課題の自己評価



図表　1-3-34　ｱｰｷﾃｸﾁｬ設計課題の自己評価



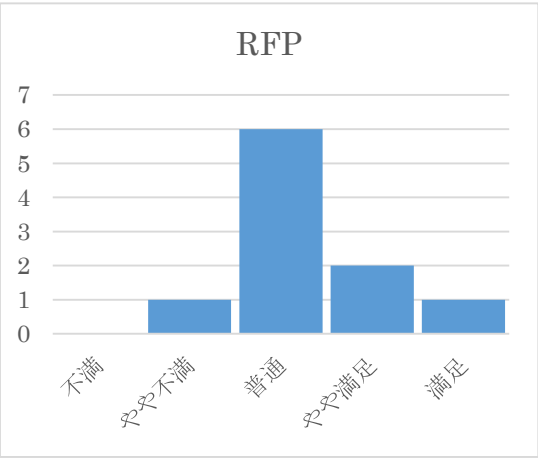
（１０）講座の満足度

実証講座の単元ごとの満足度は、「RFP」と「提案」では「普通」をピークに「やや不満」から「満足」に回答されている。「要件定義」では「やや満足」をピークに「やや不満」から「満足」に回答されている。「アーキテクチャ設計」では「やや不満」と「普通」に回答が集中している。

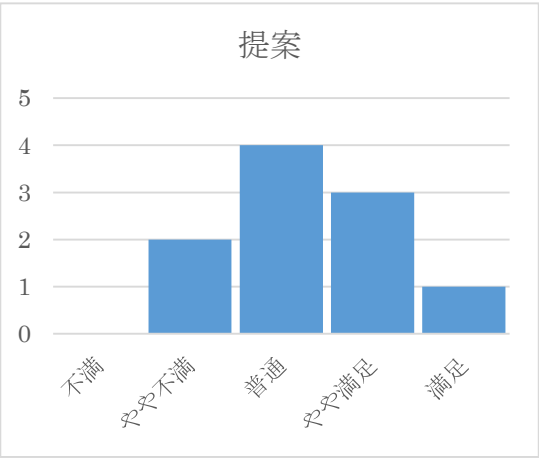
テキストの満足度は、「やや満足」をピークに「やや不満」から「満足」に回答されており、実証講座内の演習の満足度は「やや満足」をピークに「やや不満」から「満足」に回答されている。

実証講座の全体の満足度は、「やや満足」と「満足」に回答が集中しているものの、2 名からは「やや不満」と回答されている。

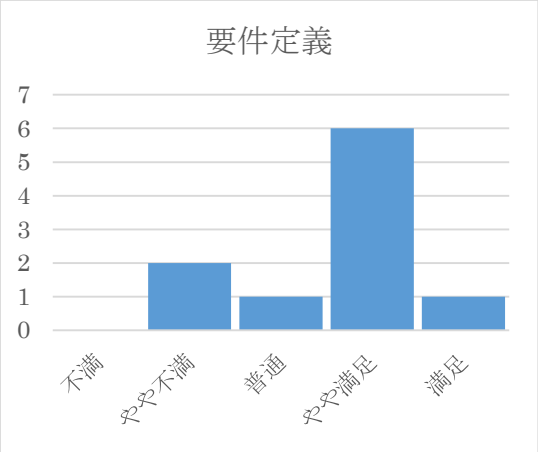
図表　1-3-35　RFP の満足度



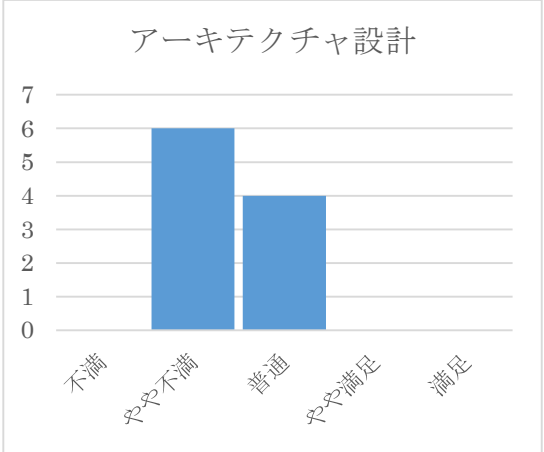
図表　1-3-36　提案の満足度



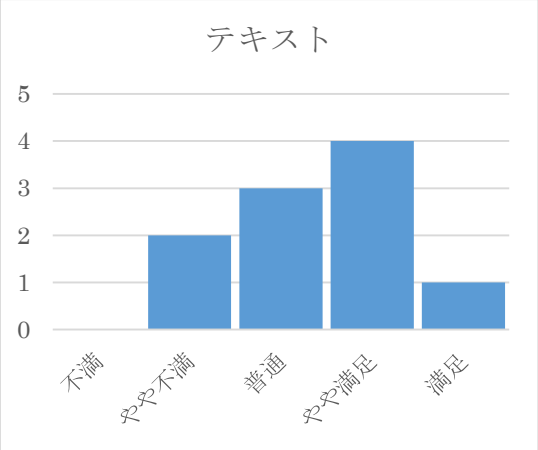
図表　1-3-37　要件定義の満足度



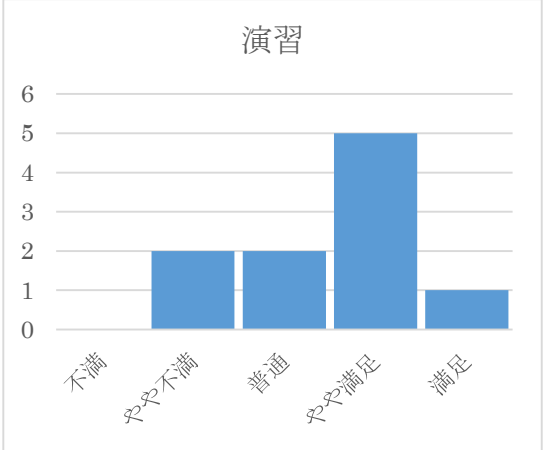
図表　1-3-38　ｱｰｷﾃｸﾁｬ設計の満足度



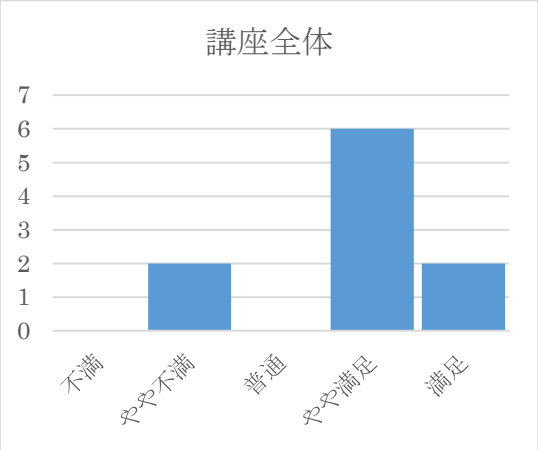
図表　1-3-39　テキストの満足度



図表　1-3-40　演習の満足度



図表　1-3-35　実証講座全体の満足度



（１１）講座に対する意見

■講座内容「RFP」に対する意見

・RFP をいただく機会が今後あれば使っていきたい
・RFP 自体をはじめて知りました。システム受託において、何に集中して、お客様からの要件を見ていくのかの判断基準をひとつ頂きました。
・理論と演習は、わかりました。実践に生かせるようにしていきたいと思います。RFP を読み解くのとヒアリングは実践で学んでいきます。
・RFP がどのようなものか理解できた。

■講座内容「提案」のヒアリングに対する意見

・事前準備と順序立てて聞いていくことが重要だと感じた
・ヒアリングする機会はないのでいい経験になったと思う。質問内容など考えたり難しかった
・お客様の前に行くことがほぼなく、困難さ肌で感じた。また、演習ではあったが、実際のビジネスの場の緊張感を演出されていて、貴重な体験となった。
・RFP の読み込み度とヒアリング力が足りないと痛感いたしました。
・ヒアリングする上で、どういった質問をすればよいかイメージが沸かなかった。

■講座内容「提案」の提案書作成に対する意見

・頭でイメージしていることを文書として形にすることの難しさを感じた
・提案のためにヒアリングが大事だと感じた。また、いろいろな IT のスキルが必要だと強く感じた。
・A 社のクライアントの益になることを想像していたが、お客様が何を求めているかに関する視点がまだまだ足りないことを実感した。
・提案内容に盛り込まないといかない点、RFP から外れないようしながら会社の強みをだすのが難しい。
・どういった提案をすればよいかイメージが沸かなかった。

■講座内容「提案」の提案発表に対する意見

・実際にはなかなか提案をする機会がないのでいい経験ができた
・ヒアリング、提案書作成の時点の視点がずれていたことの影響が強く出てしまった。
・提案の下準備が自分の中にうまくできておらず行き当たりばったりなってしまった。
・プレゼンの経験がなく、どういったことを行えばよいかわからなかった。
・本当にいい経験になったと思う。
・実際のビジネスの場で起こりうることがロールプレイできて非常に良かった。
・ひとつの答え（模範例）を提示していただけると、イメージがつきやすくもなると感じた。
・RFP 通りの提案だとほかの企業と同じになってしまうので会社の独自性が必要となることとプ

レゼンテーション、受け答えなどのコミュニケーションの総合的な力が必要。
・こういった上流工程での業務経験がないため何をすればよいかのイメージが沸かなかった。

■講座内容「要件定義」に対する意見
・USDM に関して今後少しずつ活用していきたい
・座学を実習してからの受講であればもう少し理解度が増していたのかなと思いました
・USDM は業務で使う機会がなさそうだが考え方など知れてよかったと思う。
・お客様視点の「要求」が何か、が印象に残った。作る側の視点ではなく、相手に伝えることに意識を持つことの大切さを学んだ。
・USDM を使用しての要件定義、仕様作成は、システム開発目線で作成してしまうので実際に使う人目線で考えたものにするということは勉強になりました。
・要求の定義において、理由の明記が重要なことを納得することができた。
・USDM とアーキテクチャについて理解を深められました。私が集中して聞けていなかったからかもしれませんが、USDM の「説明」について、講義を通してあまり重要でないイメージを抱いており、テスト結果をみるとそうではないようなので、テキストで省略されていないと目で分かりやすかったかと思いました。
・プログラムのコーディングを行っている自分にとって要件から仕様におとすということはイメージがわきやすく、取り組みやすかった。

■講座内容「アーキテクチャ設計」の設計演習に対する意見
・非機能要件を仕様化することは難しいと感じた
・理解不足のため演習例の説明もいまいちピンとこなかった
・このアーキテクチャ設計の手法をどこで用いることで、よりお客様に対して、良い、有益なプロダクト（≒システム）を提供できるのかが、捉えられなかった。
・ワークフロー等に関わるところかと思いますが、実際の経験がなくイメージできなかった
・中大規模システムの開発経験はあるが、最初期に行うフェーズのため参加したことがない
・なかなか実業務での適用は難しいと思うが、基本構造を決める際に見直してみたい。
・USDM のように、あらかじめスプレッドシートを作成しての演習があればよかったと思います。
・興味がある内容だったが、実務として使用するケースがどこまであるかはわからなかった。(業務 AP とデータベース構築、インフラ構築が別担当となるケースが多いため、試験取りまとめをする部署などだと有効ではないかと感じた)
・座学を実習してからの受講であればもう少し理解度が増していたのかなと思いました
・あまり必要性を感じなかった
・今は、規模・対応実績等がなくイメージをつけることができませんが、今後、お客様の問題解決をしていくにあたり、必要な観点・設計視点となると思いました。その機会があった際に、思い起こして設計に取り組もうと思います。

・実務として利用してはいきたい
・できれば、中規模システムで活用可能、または中級者が設計する際のガイドラインとして使いやすい内容にしてほしい。
・例が事業成果報告書に明記されていたため、アーキテクチャ設計のことを少しは理解できたと思いますが、前日に出席していないこともあり、自分で設計を行うことはとても難しく感じました。
・アーキテクチャ設計を行うメリット？がいまいち掴めなかった。

■テキストに対する意見
・詳細に記載されていて、理解しやすくなっていた。
・今回は演習が中心の講座だと思うのでテキストについては特になし。
・自分には少し難しかった。
・講師の先生の説明がないと理解しづらいと思う。
・実習（PBL）とテキスト（文字説明ベース）が少し剥離していたように感じました。
・実習で持つイメージとつながる資料があると、理解・イメージの助けとなったのかと思いました。
・しっかりと読み込むことで、非常に有益なテキストかと思います。
・全体網羅されてていいテキストでした。
・時間がなく精読できていないが、USDM については業務に活かせると思う。
・USDM の「説明」が重要でないというイメージを抱いてしまうのが、少し気になりました。
・各設計の回答例等、記載されていたため理解しやすかった。

■講座内の演習に対する意見

・USDM についてあまり実施したことがなかったが、演習を通じて、業務に活用できる内容だった。
・ロールプレイ的な演習で実際の機会につなげられるような演習ができたと思います。
・テキストの専門用語が難しかったように感じた。講師の先生の説明で理解できるようになっていく感じだったので、数日たって見ても自分に理解できるか不安に感じる。
・講習会の中の演習は、実践的なものが多かったので、勉強になりいい経験になったと思う。
・説明、質疑応答の対応等々、大政講師の講義が非常に良かったです。
・実習等に関して、場・グループ創りが大変かと感じました。ですが、それも楽しかったです
・演習を繰り返すことでスキルが向上していくので続けていきたいです。
・ヒアリングの部分は、社内で顧客側、IT 企業側とわけてすることで社内でのスキル向上していくと思われます
・受講者同士での意見交換の時間が有意義だった。できれば、演習の成果物について講師による評価がほしい。
・アーキテクチャ設計の演習形式を、要件定義と同じくしてほしいです。
・グループ演習時、自分だけ IT に関する知識が低いため話についていけないことが多々あった。

（１２）評価委員のコメント

■講座内容「RFP」に対する意見

・例題的には特に問題はないと思うが、顧客の要求（ＲＦＰ）とシステム化の要件（提案）の関係性をもう少し丁寧な説明があればと思います。
・専門学校で使うためには、RFP 自体の講座が１，２コマ必要である。

■講座内容「提案」のヒアリングに対する意見

・システム化提案をする上の顧客ニーズや問題の確認とその分析、課題解決策の検討、ヒヤリング項目検討の不足が感じられ、以降の作業に影響をおよぼしたのではと思います。
・ヒアリングに入る前の時間が足りない感じであった。もう少し時間を取る必要があったことと、ここで、チェックシートを使ったヒアリング時の聞き漏れがないような活動を入れたかった。

■講座内容「提案」の提案書作成に対する意見
・PBL の実施視点では、チーム内の議論が不足していたのではと思います。
・提案書作成に対する時間が足りないようであった。もう少し時間を取りフィードバックなども含め、考えさせる時間が必要である。
・テキストが提案する深度(どのレベルまで)が明確になっていないため、開発者よりの提案をしている人が目立った。

■講座内容「提案」の提案発表に対する意見
・顧客の要望や要求に対する実現するためのシステム化の要件と課題が述べられず、全体的には、顧客要求（RFP)の焼き直しになり、提案として魅力を感じられなかった。
・発表前の時間が少ない。グループでもう少し揉み、ファシリテータとのやり取りも多くとれるようにすると良い。

■講座内容「提案」に対する意見
・顧客の要望や要求に対する実現するためのシステム化の要件と課題が述べられず、全体的には、顧客要求（RFP)の焼き直しになり、提案として魅力を感じられなかった。
・とにかく時間が足りない。それとテキストのサンプルが少なく、PBL にいくにしても受講生の基準となるものがない。

■講座内容「要件定義」に対する意見
・USDM のお作法が中心の講義と多少の演習となり、提案書と要件定義の一連性を踏まえたチーム議論（ＰＢＬ）にならず、なぜ要件定義が必要なのかが理解できなかったように思いました。
・基準となる流れのあるサンプルがないため、受講者も迷ったのではないかと思う。あとやはり授業時間が少なすぎる。

■講座内容「アーキテクチャ設計」の設計演習に対する意見
・知識ベースの講義、かつ部分的な項目の説明となり、時間的な制約もあるかと思うが。
・あとは読んでおいてといった単なる通常の授業感が感じられた。
・時間がなく、深いところまで踏み込めていない感じがした。

■講座内容「アーキテクチャ設計」に対する意見
・時間的、内容ボリューム等に今後の課題があり、本来の望む内容であったか改善が必要と感じました。
・受講者から質問もあったが、アーキテクチャと設計エンジニアはどう違うのかが同の程度理解されたか、なぜアーキテクチャ育成なのかなどを今一度整理する必要があると思います。
・決して、小さいシステムだからアーキテクト視点はいらないということではない。
・授業時間が少なく、アーキテクチャ設計は、タイトなスケジュールであった。

■テキストに対する意見
・この講座テキストでは何が重要なことで、補足的なことなどのポイントが見えづらく、受講者の受講後のゴールがはっきりしない資料になっていると思います。
・PBL に使う題材とは別に提案、要件定義、アーキテクチャ設計の流れに沿うサンプル（例題）が必要である。PBL を行うに当たっても事前に設計の流れがつかめた方がよいこととグループで作業をする上での基準（温度差を埋める）が必要である。

■講座内の演習に対する意見
・この実証講座は受講者が自主的に考え、チームで議論することでアーキテクチャとしての不足を感じ、これからの自己研磨へつながることを前提でＰＢＬを導入したのでは。今回の講座は通常の授業とあまり変わらないのではと思いました。
・演習についても時間が足りないことと、結果に対するフィードバックの時間も足りなかった。

3.3.6. 統計分析の結果

（1）理解度確認テストの分析

受講者ごとの事前テストと事後テストを分散分析すると以下の結果が得られ、寄与率（p-値）が有意水準 5%を満たしていないことが確認できる。

この結果は今年度の実証講座における知識習得フェーズの効果を否定するものではない。事前テストと事後テストの結果からは知識習得フェーズの効果を計ることができなかったことを表しており「十分な受講者人数が得られなかったこと」「理解度確認テストの問題数（点数差）が十分でなかったこと」「今年度の実証講座の内容が適応訓練フェーズの演習に重点が置かれたことによる結果」などが考えられる。

図表　1-5-1　分散分析表

変動要因	変動	自由度	分散	観測された分散比	P−値	F 境界値
行	25.75	7	3.678571	2.191489	0.16117	3.787044
列	0.25	1	0.25	0.148936	0.711015	5.591448
誤差	11.75	7	1.678571			
合計	37.75	15				

（2）アンケートの分析

実証講座終了時に取得した事後アンケートにおいて、分散分析等で寄与率(p-値)が有意水準 5%を満たした結果を以下にあげる。

■講座内容の活用度と講座全体の満足度

事後アンケートで取得した実証講座の単元ごとの「活用度」と「講座全体の満足度」において分散分析で寄与率（p-値）が有意水準 5%を満たし、「講座内容「提案」の活用度」と「講座全体の満足度」に強い相関が確認できた。

この結果から、「活用度の高い単元」と「講座全体の満足度」の関係が示され、本実証講座においては「講座内容「提案」の単元」が「講座全体の満足度」に強く関係していることが確認できた。

図表　1-5-2　分散分析表

変動要因	変動	自由度	分散	観測された分散比	P−値	F 境界値
行	9.22	9	1.024444	1.44741	0.205177	2.152607
列	12.52	4	3.13	4.422292	※0.005203	2.633532
誤差	25.48	36	0.707778			
合計	47.22	49				

図表　1-5-3　相関

	講座内容「RFP」の活用度	講座内容「提案」の活用度	講座内容「要件定義」USDM の活用度	講座内容「アーキテクチャ設計」の活用度
講座全体の満足度	0.280976	※※0.861858	0.116642	−0.45644

■講座内容の活用度とテキストの満足度

事後アンケートで取得した実証講座の単元ごとの「活用度」と「テキストの満足度」において分散分析で寄与率（p-値）が有意水準 5%を満たし、「講座内容「提案」の活用度」と「テキストの満足度」に相関が確認できた。

この結果から、「活用度の高い単元」と「テキストの満足度」の関係が示され、本実証講座においては「講座内容「提案」」の単元が「テキストの満足度」に関係していることが確認できた。

図表　1-5-4　分散分析表

変動要因	変動	自由度	分散	観測された分散比	P−値	F 境界値
行	8.02	9	0.891111	1.259027	0.292152	2.152607
列	10.12	4	2.53	3.574568	※0.01486	2.633532
誤差	25.48	36	0.707778			
合計	43.62	49				

図表　1-5-5　相関

	講座内容「RFP」の活用度	講座内容「提案」の活用度	講座内容「要件定義」USDM の活用度	講座内容「アーキテクチャ設計」の活用度
テキストの満足度	0.150188	※0.581914	0.062348	−0.16265

■講座内容の活用度と演習の満足度

事後アンケートで取得した実証講座の単元ごとの「活用度」と「演習の満足度」において分散分析で寄与率（p-値）が有意水準 5%を満たし、「講座内容「提案」の活用度」と「演習の満足度」に強い相関が確認できた。

この結果から、「活用度の高い単元」と「演習の満足度」の関係が示され、本実証講座においては「講座内容「提案」の単元」が「演習の満足度」に強く関係していることが確認できた。

図表　1-5-6　分散分析表

変動要因	変動	自由度	分散	観測された分散比	P−値	F 境界値
行	8.88	9	0.986667	1.436893	0.209355	2.152607
列	10.48	4	2.62	3.815534	※0.010977	2.633532
誤差	24.72	36	0.686667			
合計	44.08	49				

図表　1-5-7　相関

	講座内容「RFP」の活用度	講座内容「提案」の活用度	講座内容「要件定義」USDM の活用度	講座内容「アーキテクチャ設計」の活用度
演習の満足度	0.373254	※※0.903877	0.077475	−0.56592

■講座内容の理解度とテキストの満足度

事後アンケートで取得した実証講座の単元ごとの「理解度」と「テキストの満足度」において分散分析で寄与率（p-値）が有意水準 5%を満たし、「講座内容「提案」の理解度」と「テキストの満足度」に薄い相関が確認できた。

この結果から、「理解度の高い単元」と「テキストの満足度」の関係が示され、本実証講座においては「講座内容「提案」の単元」が「テキストの満足度」に薄く関係していることが確認できた。

図表　1-5-8　分散分析表

変動要因	変動	自由度	分散	観測された分散比	P−値	F 境界値
行	9.78	9	1.086667	1.582524	0.157793	2.152607
列	8.48	4	2.12	3.087379	0.027694	2.633532
誤差	24.72	36	0.686667			
合計	42.98	49				

図表　1-5-9　相関

	講座内容「RFP」の理解度	講座内容「提案」の理解度	講座内容「要件定義」の理解度	講座内容「アーキテクチャ設計」の理解度
テキストの満足度	−0.21822	※0.281718	0.121566	−0.32733

■講座内容の理解度と演習の満足度

事後アンケートで取得した実証講座の単元ごとの「理解度」と「演習の満足度」において分散分析で寄与率（p-値）が有意水準 5%を満たし、「講座内容「提案」の理解度」と「演習の満足度」に薄い相関が確認できた。

この結果から、「理解度の高い単元」と「演習の満足度」の関係が示され、本実証講座においては「講座内容「提案」の単元」が「演習の満足度」に薄く関係していることが確認できた。

図表　1-5-10　分散分析表

変動要因	変動	自由度	分散	観測された分散比	P−値	F 境界値
行	10	9	1.111111	1.626016	0.144829	2.152607
列	9.4	4	2.35	3.439024	※0.017646	2.633532
誤差	24.6	36	0.683333			
合計	44	49				

図表　1-5-11　相関

	講座内容「RFP」の理解度	講座内容「提案」の理解度	講座内容「要件定義」の理解度	講座内容「アーキテクチャ設計」の理解度
演習の満足度	0.232425	※0.280056	−0.10071	−0.36155

3.4. 総括

3.4.1. 考察

（1）受講者数確保についての課題

実証講座の受講者は、当初想定していた「愛媛県内のプログラマーである」「Java や C#等、少なくとも一つのオブジェクト指向言語について業務レベルのコーディング能力をもったプログラマーであること」をおおむね満たしていることは評価できる。

一方で、実証講座の全受講者人数は 15 名で、2 日間の全工程の受講者が 8 名にとどまったことで、本実証講座の評価に対して統計的に有意なデータを示すことができなかった。

（2）理解度確認テストによる評価

本実証講座で実施した事前テストと事後テストから、平均正答率が 0.8%微増したこと、両テスト受験者 8 名の内 5 名に 10%から 20%の正答率上昇がみられたこと、テスト問題の 10 問中 6 問で正答率が上昇したことから一定の成果があったとはいえるが、大きな学習効果が見出されたとはいいがたい。

（3）事前アンケートによる講座への期待

事前アンケートにおいて、実証講座に参加した受講者 15 名の内 13 名が、実証講座への期待が「大きい」「やや大きい」と回答しており、また、受講者の勤め先で行われている要件定義とアーキテクチャ設計が妥当的・効率的に実施されているかの問いと本実証講座での要件定義とアーキテクチャ設計への期待の大きさとの間に、それぞれ薄い逆相関（「-0.13」「-0.15」）がみられ、自社の上流工程に疑問を持っている受講者が本実証講座への期待をやや持っているとみられる。

（4）講師の評価

事後アンケートの回答者 10 名中 7 名が「講師の教え方」に対して「良い」「やや良い」と回答しており、「講師の進行速度」に対しては回答者 10 名中 5 名が「普通」で 4 名が「やや速い」と回答しており、講師の特性が実証講座の検証評価において悪い影響を与える要因にはなっていないと判断できる。

（5）テキストの評価

「テキストの内容」は、事後アンケートの回答者 10 名中 6 名が「有益」「やや有益」と回答していることから、一定の評価ができると思われる。

一方で、「テキストの分かり易さ」は、事後アンケートの回答者 10 名の内 5 名が「分り難い」「やや分り難い」に回答しており、改善の必要性があると思われる。

また、事後アンケートの「テキストに対する意見」には、「詳細に記載されていて、理解しやすくなっていた」や「各設計の回答例が記載されていたため理解しやすかった」等の評価されるコメントがある一方で、「自分には少し難しかった」や「講師の説明がないと理解しづらいと思う」とのコメントがあり、テキストや資料には改善の余地があると思われる。

（6）難易度

「講座の難易度」は、受講者 10 名の事後アンケートから「RFP」「提案」「要件定義」で 6 名以上が「普通」と評価しており、受講者にとってレベルが低すぎることも難易度が高すぎることもない実証講座になったと思われる。

「アーキテクチャ設計」においては、4 名が「レベルが高い」と評価しており、テキストや実証講座の内容について見直しが必要なのではないかと思われる。

（7）活用度

単元ごとの「活用度」において「要件定義」では、受講者 10 名の事後アンケートから 7 名が「業務に活かせる」「やや活かせる」との回答があり、講座内容「要件定義」に対する意見においても、「USDM に関して今後少しずつ活用していきたい」とのコメントもあり、コーディング業務の多い今回の実証講座の受講者たちにとって「要件定義」は特に活用度を高く感じる内容になったようである。

（8）理解度と自己評価

単元ごとの「理解度」で、「要件定義」は、受講者 10 名の事後アンケートから 5 名が「理解できた」「やや理解できた」に回答しており、他の単元に比較して高い理解度が得られたようである。

また、単元ごとの「自己評価」においても、「要件定義の USDM の課題に対する自己評価」においてのみ、「上手くできた」「やや上手くできた」に回答があり、「要件定義」は受講者にとって手ごたえを感じることができた単元になったようである。

事後アンケートで取得した単元ごとの「理解度」と「自己評価」の点数において、「要件定義の理解度」と「要件定義の USDM の課題に対する自己評価」のみで高い相関関係（相関係数 0.77）を示している。

（9）満足度

単元ごとの満足度では、「要件定義の満足度」に受講者 10 名の事後アンケートから「満足」「やや満足」に 7 名の回答があり、他の単元に比較して満足度が高いと言える。また、「講座全体の満足度」と「提案の活用度」に高い相関関係（相関係数 0.86）を示しており、合わせて「演習の満足度」と「提案の活用度」にも高い相関関係（相関係数 0.90）を示しており、「提案」の単元で実施した「演習」が実証講座の満足度に影響しているものを考えられる。

3.4.2. 結論

実証講座において、事前の理解度確認テストの結果が平均得点 5.75 点（10 点満点、以下同様）であるのに対して、事後の理解度確認テストの結果は 6.0 点であった。事前と事後の理解度確認テストの平均得点を比較すると大きな学習効果があったとはいいがたい（標本数が少ないため統計的有意差はなし）。実証講座は演習ベースで行われたのだが、理解度確認テストは形式上、知識を問うものとなっており、両者のマッチングに問題があった可能性がある。

また、事後アンケートで、「講座全体の満足度」は、「やや満足」が 60%、「満足」が 20%、「やや不満足」が 20%となっており（満足、やや満足、普通、やや不満、不満の 5 段階評価）、一定の満足要因とともに一定の不満足要因の存在を示唆するものになっている。

実際に、「講座全体の満足度」と講座内の「提案」パートの満足度の間に相関係数 0.86 の高い正の相関が見出され、同様に「テキストの満足度」との間にも、0.87 の高い正の相関が見出されている。今回の実証講座の満足度は、「提案」パートの内容と演習用テキストによって大きくえられたものだと考えられる。逆に、「講座全体の満足度」と「アーキテクチャ設計」パートの間では、相関係数が 0.58 とやや低くなっており、「講座全体の満足度」への寄与があまり高くない。このパートの構成改善が課題として考えられる。この点は、受講者のアンケート結果や評価委員のコメントに難易度の高さ、講座時間の不足が示唆されているところにも現れており、テキストによるいっそうの補助もしくは講座時間の拡大が必要ではないかと考えられる。

以上のように、本事業で開発した教育プログラムは、「提案」パートを中心に「要件定義」の領域については一定の完成度に達することができたが、「アーキテクチャ設計」の領域については、テキストの現状の量および質に対して、さらなる補助的記述の追加を行なうか、講座時間の見直しをはかることが必要である。

４. 事業成果物

本教材（『要件定義・アーキテクチャ設計』および『PBL 用テキスト』 ※次ページ以降に掲載）は、文部科学省の生涯学習振興事業委託費による委託事業として、学校法人河原学園 河原電子ビジネス専門学校が実施した平成２９年度「専修学校による地域産業中核的人材養成事業」の成果物です。

要件定義・アーキテクチャ設計

目次

1 要件定義－はじめに 9

1.1 はじめに 9

2 ソフトウェア工学の目的 11

2.1 ソフトウェア工学を学ぶ意義 ― 経験と学問の差異 11

2.2 ソフトウェア工学の定義 15

2.3 SWEBOK ガイド 21

2.4 ソフトウェア工学とソフトウェアライフサイクル 38

2.5 ここまでのまとめ 46

3 共通フレーム 2013 50

3.1 ISO/IEC 12207：2008（JIS X 0160:2012）の目的 50

3.2 ISO/IEC 12207 から共通フレームへの発展 51

3.3 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 52

3.4 共通フレームの目的 55

3.5 共通フレームの対象範囲 55

3.6 共通フレームの設計思想 56

3.7 共通フレーム 2013 に含まれるプロセス・アクティビティ 70

3.8 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 82

4 要件定義の意味 85

4.1 共通フレームにおける要件定義 86

4.2 要求工学と要求工学知識体系（REBOK） 91

4.3 REBOK の知識体系 93

5 要求仕様にかかわる諸問題 102

5.1 ソフトウェア開発の現場における仕様の問題とは 102

5.2 要求仕様書に関わる問題と背景 103

6 要求仕様書の意味 112

6.1 要求仕様書とは 112
6.2 「要求」とは何か 113
6.3 「理由」をつける理由 113
6.4 「仕様」とは何か 114
6.5 「要求」と「仕様」の関係 114
6.6 トレーサビリティへの対応 115
6.7 非機能要求の対応 115
6.8 「説明」の利用法 115
6.9 仕様漏れや仕様の衝突などの発見法 116
6.10 要求仕様書を記載するためのツール 116
6.11 要求仕様書は誰が書くか 117

7 要求仕様の定義方法 119

7.1 さまざまな要件定義技法について 119
7.2 USDM とは 119
7.3 要求仕様書の使いかた 120

8 非機能要求の定義方法 123

8.1 非機能要求とは何か 123
8.2 非機能要求の重要性 124
8.3 非機能要求の定義の仕方 124
8.4 品質 125
8.5 サービス・レベルに関する要件 125
8.6 サービス提供時間 126
8.7 保守・レスポンス・タイム 126
8.8 性能・バッチ処理の実行時間 127
8.9 性能・デリバリータイム 127
8.10 キャパシティに関する要件 128
8.11 データ量 128
8.12 ユーザー数 129
8.13 トランザクション数 129
8.14 アウトプット数 130

8.15 インプット数 130
8.16 安全性に関する要件 131
8.17 セキュリティ 131
8.18 停止許容時間 132
8.19 有事対策 133
8.20 コストに関する要件 133
8.21 保守に関する要求 134
8.22 非機能要求の適用について 134
8.23 開発リスク軽減の方法 136

9 USDM による要求仕様書作成 137

9.1 目的 137
9.2 要求には「理由」がある 137
9.3 演習①（要求から理由を探る） 139
9.4 要求を上手く表現するには 140
9.5 理由に要求が混在する 141
9.6 演習②（要求をうまく表現するには・要求から理由を探る） 141
9.7 要求の仕様化 142
9.8 必要なら説明をつける 143
9.9 用語集への展開 145
9.10 演習③（要求から仕様を抽出する－1） 146
9.11 演習④（要求から仕様を抽出する－2） 146
9.12 要求のカテゴリ化 147
9.13 品質要求 150
9.14 品質要求の仕様化 151
9.15 要求の階層化のテクニック 153
9.16 分割基準の共有 158
9.17 テストケース仕様とつなぐ 158
9.18 仕様のグループ化 159
9.19 シナリオ機能と単純機能の違い 160
9.20 ユースケースにおける問題点 162
9.21 状態遷移図から導出する 164
9.22 画面仕様のあり方 164
9.23 画面操作全体に対する要求 167
9.24 個別の画面設計の指針 168

9.25　画面遷移と画面仕様の連携　169

10 付録　172

10.1　在宅介護サービス支援システム　仕様書　（付１）　172
10.2　USDM 表記法フォーマット　176
10.3　引用・参考文献　177

11 アーキテクチャ設計ーはじめに　178

11.1　はじめに　178
11.2　本章の構成　180

12 アーキテクチャ設計の意味　181

12.1　アーキテクチャ設計とは　181
12.2　代表的なアーキテクチャ設計の定義　186
12.3　アーキテクチャの構造とビュー　191

13 アーキテクチャと品質特性　199

13.1　JIS X 0129-1 品質モデル　199
13.1　〈内部品質属性〉と〈保守性〉　211
13.2　ソースコード①　214
13.3　ソースコード②　216
13.4　ソースコード③　218
13.5　ソースコード④　220
13.6　ソースコード⑤　222
13.1　JIS X 25010:2013 システム及びソフトウェア品質モデル　224
13.1　CMU/SEI アーキテクチャ手法群におけるシステム品質特性　230
13.2　品質特性シナリオ　234

14 品質特性の実現手法　249

14.1　可用性の実現手法　249
14.2　変更容易性の実現手法　255
14.3　性能の実現手法　268
14.4　セキュリティ性の実現手法　274
14.5　テスト容易性の実現手法　276
14.6　使いやすさの実現手法　278

15 UML ダイアグラム　281

15.1 UML（Unified Modeling Language）と開発方法 281
15.2 UML のダイアグラムの種類 282
15.3 UML に対する ３つの考え方 284

16 概念モデルとしてのクラス図 285

16.1 伝統的論理学における概念論 285
16.2 様々なクラス論 286
16.3 UML の正式仕様書におけるクラスと集合の関係 291
16.4 概念モデル/論理モデル/実装モデルの関係 293

17 アクティビティ図 295

17.1 アクティビティ図の意味と用途 295
17.2 制御フローとオブジェクトフロー 295
17.3 分岐処理の表現 300
17.4 反復処理の表現 304
17.5 アクションの事前条件と事後条件 306
17.6 並行処理と同期処理の表現 307
17.7 データの遷移の表現 310
17.8 コレクション・配列の表現 313
17.9 データベースの表現 317
17.10 コレクションに対する反復処理の表現 318
17.11 イベント処理の表現 321
17.12 振る舞いの主体の表現 325
17.13 メソッドの引数と戻り値の表現 331
17.14 例外処理の表現 332
17.15 割り込み処理の表現 336
17.16 アクティビティのネストの表現 338
17.17 アクティビティ図に関する注意点 340

18 クラス図 345

18.1 クラス図の基本的な記述の仕方 345
18.2 属性・操作の表記サンプル 348
18.3 派生属性の表記サンプル 350
18.4 属性のプロパティ文字列 readOnly の設定サンプル 352
18.5 属性のプロパティ文字列 redefines の設定サンプル 354

18.6 属性のプロパティ文字列 ordered および unique の設定サンプル 358
18.7 属性のプロパティ文字列 union と subsets の設定サンプル 361
18.8 操作のプロパティ文字列 redefines の設定サンプル 362
18.9 操作のプロパティ文字列 ordered および unique の設定サンプル 364
18.10 オブジェクトコンポジションと関連 366
18.11 「関連端」による表現 370
18.12 集約 371
18.13 合成集約 374
18.14 汎化 377
18.15 抽象クラス 382
18.16 インターフェイス 384
18.17 実現 385
18.18 クライアントとサプライア 388
18.19 ロリポップを用いた表現 389
18.20 依存 390
18.21 限定子（qualifier） 394

19 コレクション・フレームワークと多重度 395

19.1 多重度 395

20 ユースケース図とユースケース記述 397

20.1 機能要求と非機能要求 397
20.2 ユースケース図の意味と用途 398
20.3 ユースケース図の記法 400
20.4 ユースケース図とユースケース記述 406
20.5 ユースケース記述のテンプレート例 406
20.6 ユースケース図とユースケース記述の例１ 408
20.7 ユースケース図とユースケース記述の例２ 411

21 シーケンス図 417

21.1 シーケンス図の意味と用途 417
21.2 シーケンス図の記法 418
21.3 メッセージの意味と種類 421

21.4 メッセージ名の記法 423
21.5 インスタンス化とインスタンス破棄の表現 425
21.6 オブジェクトの状態に関する制約の表現 426
21.7 複合フラグメント 427
21.8 インタラクションのネストの表現 432
21.9 RIA の表現例 436

22 ソフトウェアパターンの利用 437

22.1 ソフトウェア開発の現状とソフトウェアパターンの必要性 437
22.2 ソフトウェアパターンの分類 438
22.3 アーキテクチャパターン 440

23 アーキテクチャ設計の方法論 447

23.1 アーキテクチャドライバとは 447
23.2 代表的なアーキテクチャ設計手法 448

24 品質特性駆動型設計 450

24.1 アーキテクチャ設計の手順 450
24.2 アーキテクチャを文書化する 455
24.3 アーキテクチャを評価する 465

25 品質特性駆動設計によるアーキテクチャ設計 472

25.1 プロジェクトの背景 472
25.2 要件定義の成果 473
25.3 アーキテクチャを設計する 477

26 詳細設計 502

26.1 〈内部品質属性〉と〈保守性〉 502
26.2 オブジェクト指向設計の５原則 504
26.3 開放-閉鎖原則とは 504
26.4 開放－閉鎖原則に合致するための設計上の注意点 514
26.5 単一債務の原則の概要 514
26.6 違反した事例 515
26.7 違反事例における問題点 518
26.8 設計の修正 519
26.9 スーパークラスとサブクラス 523

26.10 リスコフ置換の原則の概要 523
26.11 リスコフ置換の原則の考え方 524
26.12 継承の妥当性の具体的基準となるもの 527
26.13 リスコフ置換の原則に基づいて修正した設計 528
26.14 その他の違反事例 530
26.15 アプリケーションの「上位レベルの方針」 533
26.16 依存関係逆転の原則 535
26.17 オブジェクト指向設計と上位レベルの方針 538

1 要件定義ーはじめに

1.1 はじめに

昨今、ソフトウェア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、スマートフォン、ウェアラブル、そしてモノのインターネットとも呼ばれる IoT(Internet of Things)など、我々の身の回りはソフトウェアを使用した製品で溢れており、ソフトウェアを全く使わずに一日を過ごす事は不可能と言っても良いほど、我々の生活に溶け込んできている。その結果、様々な企業が積極的にソフトウェア開発への投資を行うようになり、以前であればソフトウェア開発などとは縁遠い存在であったような業界の顧客から依頼を受ける事も多くなってきている。

また、ムーアの法則[1]などでも知られるようにハードウェアの進歩は日進月歩で、以前はハードウェアの制限により不可能と思われていた事が実現可能となる事で、そこに組み込まれるソフトウェアに求められる要求も大きな広がりを見せている。そのため、ソフトウェア開発の規模は膨れ上がり、要求は複雑化の一途を辿る。その一方で、企業間の競争においても良いソフトウェアを積んだ製品を如何に早く販売出来るかと言う所に重きが置かれるようになり、短い納期での開発を迫られる事も少なくない。

このような、規模が大きく、複雑なソフトウェアの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。

当然だが、そのような状況の中、顧客の要望を満たすようなソフトウェアを開発する事は困窮を極める。しかし、我々ソフトウェアエンジニア及びそのチームは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。幾ら、仕様書通り作りましたよと言い張ったとしても、顧客は落胆し、そして、最終的にそれは開発プロジェクトを引き受けたチームの責任となり、信頼の低下へと繋がる。

これが意味する所は、幾ら要件定義以降の工程を正確に執り行う事が出来ても、要件定義の段階で正しく顧客の要望を理解出来ずに進んでしまえば、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、と言う事である。

そのため、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「顧客が潜在的に求める物」を理解し、それを抜け漏れなく把握した上で、要求仕様書を作成する必要がある。顧客だけでは表現しきれなかった要求を正しく引き出した上で、後続の工程に引き継ぎ、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のである。

---

[1]インテルの創業者の一人であるゴードン・ムーアが、1965 年に自らの論文で唱えた「半導体の集積率は 18 か月で 2 倍になる」という半導体業界の経験則

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顧客の要求を引き出すのか、またそれらを要求仕様書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、そしてその知識体系である REBOK(アールイーボック)を学ぶ事で各プロセスにおける具体的な方法論を学ぶ。最後に、それらを通じて抽出された要求を、USDM を用い要件仕様書としてまとめる方法に触れる。

2 ソフトウェア工学の目的

2.1 ソフトウェア工学を学ぶ意義 ― 経験と学問の差異

■経験の累積性

ひとは、なんということもなく、多くのことを知っています。

ひとは、生誕以来、日常生活の多様な「経験」の中で、多くのことを直接見聞し、記憶に蓄積しています。たとえば、スマートフォンやインターネットについても、現在では多くのひとが、仕事や私生活の日常的な使用を通して、(個人差はあるとしても)多くのことを知っています。つまり、多くの日本国民が、スマートフォンやインターネットについて、なんということもなく知っており、日常的にそれなりに利用できているということです。

ところで、街中には多くの飲食店があります。ラーメン屋を例にとると、ラーメン屋さんは日々ラーメンを作る仕事をしていると思われます(厳密には、ラーメン屋で働くすべてのひとがラーメン作りに勤しんでいるわけではないでしょうが)。ラーメンを作ることがラーメン屋さんの日常生活の多くの部分を占めているはずです。したがって、ラーメン屋さんは、ラーメン屋としての経歴が長いひとほど、ラーメンについて多くのことを「経験」し、多くのことを知っているはずです。仮にそのような「経験」やそこから得られた「知識」を量的に捉えるとすれば、ラーメン屋さんの「経験」量、「知識」量は、ラーメン屋さんとしての経歴の長さに比例すると思われます。もちろん、こういった「経験」量、「知識」量の累積的な増大は、ラーメン屋さんに限った話ではなく、どのような職業についても、それどころか、どのような人間の生活にも当てはめて考えることができます。それは、どんな人間でも、どのような環境で成長した人間であっても、どんな内容の〈経験〉であっても、20年生きていれば、20年分の「経験」量と「知識」量があるといういみです。

このような「経験」の累積的増大は、職業分野においては、しばしば、職歴や経歴という仕方で、そのひとの人材としての価値を左右するものと見なされます。そこでは、「経験」の〈直接性〉と〈多様性〉が社会的に評価されるものと考えられます。

「経験」はまずもって〈直接〉的です。「経験」は自分の身に実際に起きた事実として、いつでも否定しようのない現実であり(仮に幻覚であったとしても)、書物のような他人からの伝聞とは異なって疑いを差し挟む余地がないものです。また、「経験」は多様です。人間にとって、そのときどきに経験する出来事は、つねにまったく同一ということはなく、「経験を積んだ」ことが、一般的にその人間の長所のようにいわれるのは、時間の経過とともに、そのときどきで異なるさまざまな現実に直面してきたことが評価されているからです。

したがって、「経験」を積むということは、いわば生(なま)の現実を多く「知る」ことと考えられていま

す。「習うより慣れよ」という慣用句がありますが、この言葉の根底には、「経験」の〈直接性〉と〈多様性〉、つまり、現実を直接的に、より多く「知る」ことへの信頼があります。

■学問の意味

では、そのように信頼される「経験」について、それがどのような性質をもつものなのか考えてみたいと思います。たとえば、古代ギリシャの哲学者であるアリストテレス[2]は、その著作『形而上学』[3]の中で、次のようにいっています。

> さて、このように、他の諸動物は、表象（ファンタシア）や記憶で生きているが、経験（エンペイリア）を具有するものはきわめてまれである。しかるに、人間という類の動物は、さらに技術や推理力で生きている。ところで、(3)経験が人間に生じるのは記憶からである。というのは、同じ事柄についての多くの記憶がやがて一つの経験たるの力をもたらすからである。ところで、経験は、学問（エピステーメー）や技術（テクネー）とほとんど同様のものであるかのようにも思われているが、しかし実は、(4)学問や技術は経験を介して人間にもたらされるのである。けだし、「経験が技術を作ったが、無経験は偶運を」とポロスの言っている通りである。さて、技術の生じるのは、経験の与える多くの心象から幾つかの同様の事柄について一つの普遍的な判断が作られたときにである。というのは、カリアスがこれこれの病気にかかった場合にはしかじかの処方がきいたし、ソクラテスの場合にもその他の多くの個々の場合にもそれぞれその通りであった、というような判断をすることは、経験のすることである。しかるに、同じ一つの型の体質を有する人々がこれこれの病気にかかった場合には ― たとえば粘液質のまたは胆汁質の人々が熱病にかかった場合には ― そうした体質の患者のすべてに対して常にしかじかの処方がきく、というような普遍的な判断をすることは、技術のすることである。
> （邦訳上巻 p. 22）

さしあたって学問と技術の区別は不問に付すとして、ここでは、それらと対比して「経験」のある種の制限が示されています。「経験」はたしかに、カリアスの場合、ソクラテスの場合というように、直接的で多様な事実を教えてくれます。しかし、学問（技術）とは異なり、〈普遍的な判断〉をもたらすことはできないということです。〈普遍的な判断〉とは、まだ「経験」（事実）がなかったとしても、たとえば、アリストテレスの場合にもプラトンの場合にも必ず当てはまるはずだといえるような判断のことです。「経験」に基づく判断はたんに、たまたまそのときそうだったという、多様ではあるが個別的な事実を追認する

2　アリストテレス（ギリシャ語で Ἀριστοτέλης：紀元前 384 年－紀元前 322 年）は、古代ギリシャの哲学者。英語では Aristotle と表記する。今日に伝わる主な著作は、『範疇論』、『命題論』、『分析論前書』、『分析論後書』、『トピカ』、『詭弁論駁論』、『自然学』、『霊魂論』 、『形而上学』、『ニコマコス倫理学』、『エウデモス倫理学』、『政治学』、『詩学』。アリストテレスによる論理学、自然学、形而上学といった成果は、その後の西欧の哲学、物理学といった諸学問の展開に大きな影響を与えた。

3　アリストテレス, 出 隆 訳, 『形而上学』 上下, 岩波書店, 1989

だけでしかなく、そのような「経験」からは、〈普遍的な判断〉を導き出すことはできません[4]。

> ・・・経験家よりも技術家〔理論家〕の方がいっそう多く知恵ある者だと我々は判断している、 ― このことは、「知恵（ソフィア）」なるものが、いずれの場合にも、「知ること」の方により多く関するものであることを意味するのであるが、 ― そのわけは、後者〔理論家〕は、物事の原因を知っているのに、前者はそうでないから、というにある。けだし、経験家の方は、物事のそうあるということ〔事実〕を知っておりはするが、それのなにゆえにそうであるかについては知っていない。しかるに他の方は、このなにゆえにを、すなわちそれの原因を、認知している。
> （同前 p. 23）

単なる「経験」から生じる判断は、物事がそのときは、たまたまそうだったという事実についての知識でしかありません。他方、〈普遍的な判断〉には、物事がどうしてそのようになるのか、その原因についての知識が含まれています。原因についての知識を含むからこそ、同一の原因が関与する帰結については〈普遍的な判断〉を下すことができます。つまり、まだ「経験」（事実）がなかったとしても、同一の原因からは同一の結果が必ず生じるはずだとして、一定の判断を下すことができます。そして、物事について、このような〈普遍的な判断〉を下せることが、現実について「いっそう多く知恵ある者」であるための理由だとされています。

アリストテレスはさらに、「経験」の〈直接性〉と〈多様性〉の起源であると思われる〈感覚〉にまでさかのぼり、そのいみについて論じています。

> さらに、我々は、いずれの感覚をも知恵であるとはみなさない。もちろん、たしかに感覚は、個々特殊の物事については、きわめて信頼に値する知識である。しかしこれは、なにごとについてもそれのなにゆえにそうであるかを語らない、たとえば、火のなにゆえに熱くあるかを説明しはしないで、ただその熱くあるということ〔事実〕を告げるのみである。
> （同前 p. 24）

〈感覚〉は、それ自体は否定することのできない直接的な事実です。たとえば、あるとき食べたラーメンの「塩辛い」という〈感覚〉は、それ自体否定しようがありません。〈感覚〉というのは、もっとも直接的であり、恣意や誤解が生じるスキがありません。しかし、〈感覚〉は、その〈感覚〉の原因を示すことはできません。「塩辛い」という〈感覚〉は、「塩辛い」というだけであって、「塩辛い」の原因を〈感覚〉として同時には含みません。だからこそ、アリストテレスは、学問の〈知る〉と、〈感覚〉や〈感覚〉から生じた〈経験〉の〈知る〉を区別せざるをえなかったのです。

---

[4] 統計的ないみで確率を考える場合でも、サイコロを二回ふったら二回とも一の目が出たという事実から、サイコロを二回振ると必ず一の目が出るはずだという結論を導き出すことにはなりません。統計学には統計的検定があり、観測された多様な事実を超えて、蓋然性を含む普遍的な判断を下そうとします。

・・・我々が或る物事を知っていると言いうるのは、我々がその物事の第一の原因を認識していると信じるときのことだからである・・・
（同前 p. 25）

アリストテレスが、『形而上学』の冒頭で、「存在としての存在の第一原因」（同前 p. 112）を探求する学問を規定するに先立ち論じたことは、学問の〈知る〉は、物事について、その原因を知る〈知識〉のことであり、事実をただそのまま追認するだけの「経験」の「知る」からは峻別されるということです。そして、現実について本当の意味で「知る」といえるのは学問の〈知る〉、つまり、原因を知る〈知識〉だということです。したがって、アリストテレスにしたがえば、「習うより慣れよ」では、原因を知る〈知識〉を欠くため、物事について〈普遍的な判断〉（経験していない事柄に関しても、必ず当てはまるはずだといえる判断）を下すことはできないということになります。むしろ、〈知る〉ことへの到達を放棄する姿勢だといえます。あるいは、「習うより慣れよ」は、そのような〈知識〉を必要としない、単純な身体動作の反復訓練のような場合にのみ当てはまる教訓だといえます。

■経験と学問の差異

アリストテレスにしたがえば、「経験」の「知る」と学問の〈知る〉の違いは、物事の原因についての〈知識〉の有無でした。「経験」は物事の事実をそのまま受容するだけであるのに対して、学問は、その物事の原因を明らかにするところに両者の違いがあります。別のいい方ををすれば、「経験」は、物事を、そのあるがままの物事として「知る」ことであるのに対して、学問は、あるがままの物事ではなく、物事を、原因との関係からとらえたものとして〈知る〉ことです。つまり、物事をそれがあるままそのままではなく、原因によって媒介されたものとして〈知る〉ことです。近代ドイツの哲学者ヘーゲル[5]は両方の〈知る〉ことの関係を次のように論じています。

したがって、ここには二つのものがある。すなわち、一つは直接的なもの、有的なものであり、もう一つは媒介されたもの、あるいは定立されたものである。このことは、われわれが或る対象を reflektieren（反省）する、あるいは、普通言われているように nachdenken（反省）する場合でも同じである。というのは、その場合われわれは対象を直接態においてでなく、媒介されたものとして知ろうとするからである。普通、人々は哲学の課題あるいは目的は、事物の本質を認識することにあると考えている。そして人々の理解するところによれば、このことはまさに、

5　ヘーゲル（Georg Wilhelm Friedrich Hegel、1770 年 8 月 27 日 - 1831 年 11 月 14 日）は、ドイツ哲学を代表する哲学者のひとり。主な著作は、『精神現象学』、『大論理学』、『エンチクロペディー』、『法哲学概要』。19 世紀以降、現代哲学に至るまで多大な影響を与えている。

> 事物はその直接態のままに放置さるべきではなく、他のものによって媒介あるいは基礎づけられたものとして示されなければならないことを意味するにすぎない。ここでは事物の直接的存在は、言わばその背後に本質がかくされている外皮あるいは幕と考えられているのである。さらに、あらゆる事物は一つの本質を持つと言われるならば、それは、事物の真の姿は直接現れているとおりのものではないことを意味する。
> (ヘーゲル, 『小論理学』, 邦訳下巻 pp.10-11)[6]

物事(現実)をそのままに受け取るだけの〈経験〉は、物事(現実)を〈直接態〉においてとらえているだけです。しかし、物事(現実)の〈直接態〉は、「その背後に本質がかくされている外皮あるいは幕」であって、したがって、物事(現実)はそのままでは「真の姿は直接現れているとおりのものではない」のです。物事(現実)に対して〈反省する〉ということは、つまり、物事(現実)を正しく〈知る〉ことは、物事(現実)をほかのものによって〈媒介されたもの〉としてとらえることです。アリストテレスのいい方を用いれば、物事の存在をその原因との関係において把握することです。しかし、物事(現実)は、そのままでは、〈直接態〉においては、「外皮あるいは幕」として、正しく〈知る〉ことを妨げるものでもあるのです。それが「経験」の本当のいみです。

2.2 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[7]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

6 ヘーゲル, 松村 一人 訳, 『小論理学』 上下, 岩波書店, 1978

7 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。

■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。

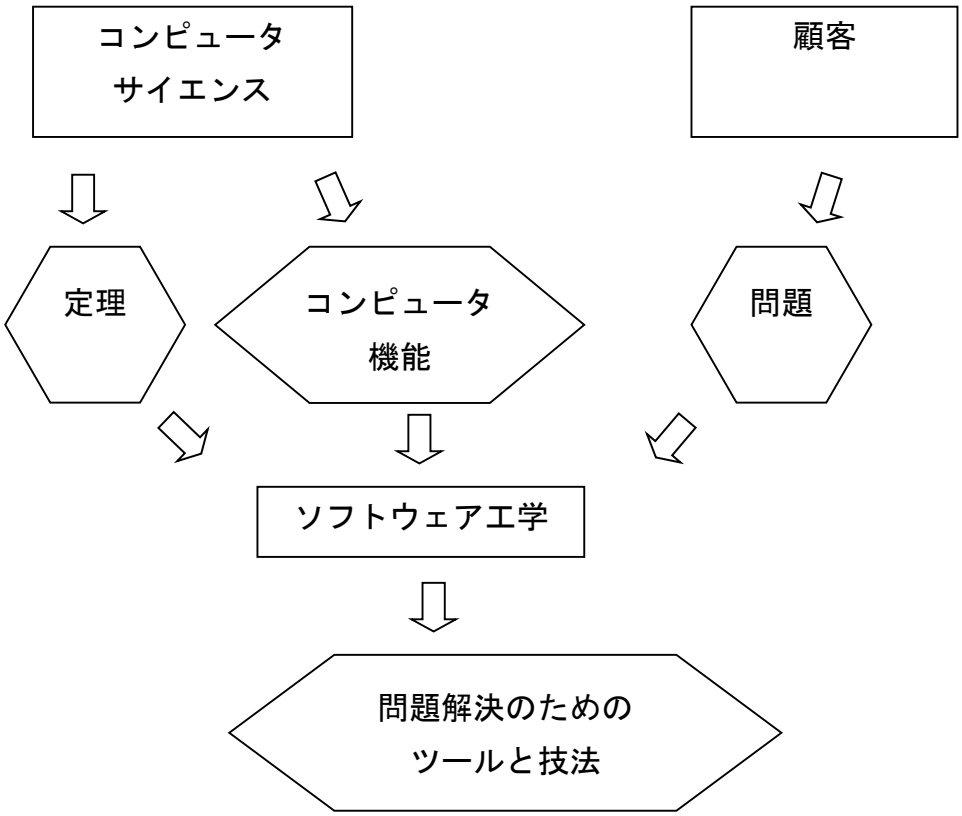
> ソフトウェアを作成することは、科学であると同時に芸術でもある。コンピュータサイエンスの学生としては、この理由を理解することが重要である。コンピュータサイエンスの専門家やソフトウェア工学の研究者は、コンピュータのメカニズムを研究して、より生産的・効率的にする方法について理論づけを行なっている。しかし、彼らもコンピュータシステムを設計し、そのシステム上で作業するためのプログラムを書いている。このことは、多くの芸術性、精巧さ、技術力を包含する実践といえる。特定のシステム上で特別な作業を実行するには多くの方法がある。しかし、そのいくつかは他よりも良いことは確かである。ある方法は、より効率的・実践的で、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)[8]

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)[9]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

8 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)

9 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)。

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。



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」[10]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。

---

10 「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。

2.3 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ない。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」)について確認する(『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[11]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)を定めたものである。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にあたる。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、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る。

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可能で、日本語訳は、2004 年度版が出版されている(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)。また、同ガイドは 2014 年に改訂版である SWEBOK V3.0 が公開され現在英語原文は https://www.computer.org/web/swebok/v3 にてダウンロードが可能となっており、同じく日本語訳も出版されている(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 V3.0』. 2014, オーム社)。

■ SWEBOK ガイド概要

以下に、SWEBOK V3.0 にて取り上げられている知識領域についてまとめる。各章にて述べられていることがらを、日本語訳版を参考に原文を拙訳し概要にまとめ、それぞれの章にて説明される大項目をリストアップ、更にそれぞれの項の中で説明される小項目を図示している。

11 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1章　ソフトウェア要求

「ソフトウェア要求」知識領域は、ソフトウェア要求の抽出(elicitation)、分析、仕様作成、および妥当性確認(validation)、さらにこれらと並ぶソフトウェアプロダクトの全ライフサイクルにわたる要求のマネジメント(management)を扱う。各項は以下の構成からなる。

1. ソフトウェア要求の基礎
2. 要求プロセス
3. 要求抽出
4. 要求分析
5. 要求の仕様化
6. 要求の妥当性確認
7. 実践上考慮すべきことがら
8. ソフトウェア要求ツール

また、それぞれは以下の図に示すような内容で構成される。

ソフトウェア要求の基礎□	要求プロセス□	要求抽出□	要求分析□	要求の仕様化□	要求の妥当性確認□	実践上考慮すべき事柄□	ソフトウェア要求ツール□
□	□	□	□	□	□	□	□
ソフトウェア要求の定義□	プロセスモデル□	要求の生成源□	要求のクラス分け□	システム定義文書□	要求のレビュー□	要求プロセスの反復性□	□
プロダクトおよびプロセスに 関する要求□	プロセスアクタ□	抽出手法□	概念モデルづくり□	システム要求の仕様化□	プロトタイピング□	変更管理□	□
機能要求と非機能要求□	プロセス支援およびマネジメント□	□	アーキテクチャ設計および 要求割りつけ□	ソフトウェア要求の仕様化□	モデルの妥当性確認□	要求属性□	□
創発的性質□	プロセス品質および改善□	□	要求折衝□	□	受け入れテスト□	要求追跡□	□
定量化可能な要求□	□	□	形式的分析□	□	□	要求の計量□	□
システム要求とソフトウェア要求□	□	□	□	□	□	□	□

第 2 章 ソフトウェア設計

設計は、「アーキテクチャ、コンポーネント、インタフェース、さらにシステムまたはコンポーネントの持つ他の特性を定義するプロセス」および「そのプロセスの成果物」である、と定義される。「ソフトウェア設計」知識領域では、「設計」という用語を包含するトピックすべてではなく、Tom DeMarco の用語にある D-設計(decomposition design)を主に論じる。D-設計の目的はソフトウェアをコンポーネントひとつひとつに写像することにある。加えて、ソフトウェアアーキテクチャという領域の重要性を考え、FP-設計(family pattern design)も取り上げる。
各項は以下のように構成され、それぞれは下図に示す内容を説明する。

1. ソフトウェア設計の基礎
2. ソフトウェア設計における主要な問題
3. ソフトウェア構造とアーキテクチャ
4. ユーザインタフェース設計
5. ソフトウェア設計品質の分析と評価
6. ソフトウェア設計のための表記
7. ソフトウェア設計戦略および手法
8. ソフトウェア設計ツール

ソフトウェア設計の基礎□	ソフトウェア設計における主要な問題□	ソフトウェア構造とアーキテクチャ□	ユーザインタフェース設計□	ソフトウェア設計品質の分析と評価□	ソフトウェア設計のための表記□	ソフトウェア設計戦略および手法□	ソフトウェア設計ツール□
□	□	□	□	□	□	□	□
一般設計概念□	並行処理□	アーキテクチャ構造と視点□	一般的なユーザインタフェース設計原則□	品質特性□	構造的記述(静的レビュー)□	一般的戦略□	□
ソフトウェア設計をとりまく前後関係□	イベントに対する制御と処理□	アーキテクチャスタイル□	ユーザインタフェース設計論点□	品質分析および評価手法□	ビヘイビア記述(動的レビュー)□	機能指向(構造化)設計□	□
ソフトウェア設計プロセス□	データの永続化□	デザインパターン	ユーザ対話様式設計□	計量尺度□	□	オブジェクト指向設計□	□
ソフトウェア設計原則□	コンポーネントのぶんさん□	アーキテクチャ設計における意思決定□	情報表示設計□	□	□	データ構造中心設計□	□
□	エラーおよび例外処理、フォールト・トレランス□	プログラムファミリおよびフレームワーク□	ユーザインタフェース設計プロセス□	□	□	他の方法□	□
□	対話と表示□	□	局地化と国際化□	□	□	□	□
□	セキュリティ□	□	暗喩と概念モデル□	□	□	□	□

第 3 章 ソフトウェア構築

ソフトウェア構築という語は、コーディング、検証、ユニットテスティング、統合テスティングおよびデバッギングの組み合わせによって、動作可能なソフトウェアを詳細に作り上げることを意味する。「ソフトウェア構築」知識領域は、他のすべての知識領域に関係しているが、とりあわけ「ソフトウェア設計」知識領域と「ソフトウェアテスティング」知識領域に強く関係している。本知識領域の各項は以下で構成され、それぞれは下図の内容を説明する。

1. ソフトウェア構築の基礎
2. ソフトウェア構築のマネジメント
3. 実践上考慮すべきことがら
4. 構築テクノロジ
5. ソフトウェア構築ツール

ソフトウェア構築の基礎□	ソフトウェア構築のマネジメント□	実践上考慮すべきことがら□	構築テクノロジ□		ソフトウェア構築ツール□
□	□	□	□	□	□
複雑さの最小化□	ソフトウェア構築ライフサイクルモデル□	構築設計□	API設計およびその使用□	文法準拠入力手法□	開発環境□
変更の予測□	構築計画□	構築言語□	オブジェクト指向実行時の論点□	並行プリミティブ□	GUIビルダ□
検証を考えた構築□	構築計量□	コーディング□	パラメータ化およびジェネリクス□	ミドルウェア□	ユニットテスティングツール□
再利用□	□	構築テスティング□	アサーション、コントラクト付き設計、防衛的プログラミング□	分散ソフトウェアのための構築法□	プロファイリング、性能分析およびスライシング・ツール□
構築のための標準□	□	再利用のための構築□	エラー処理、例外処理およびフォールト・トレランス□	非齋次システムの構築□	□
□	□	構築品質□	実行可能モデル□	性能及びチューニング□	□
□	□	統合□	状態準拠構築および表駆動構築手法□	プラットフォーム標準□	□
			実行時構成および国際化□	テスト・ファースト・プログラミング□	

第 4 章 ソフトウェアテスティング

ソフトウェアテスティングは、プログラムが、日常生まれる無限の実行ドメインから適切に選択された(selected)テストケースの有限(finite)集合のうえで行われる。期待される(expected)振る舞いを提示するための動的(dynamic)検証から構成される。「ソフトウェアテスティング」知識領域では、ソフトウェアテスティングの場における基礎的定義、用語、および主要な論点、そしてアクティビティの関係などから始まり、テストの分類やそれぞれのターゲット、目的そして、その組み合わせがテスト集合をどのように同定するかなどについて示す。各項は以下から構成され、それぞれは下図に示される内容を説明する。

1. ソフトウェアテスティングの基礎
2. テストレベル
3. テスト技法
4. テストに関係した計量尺度
5. テストプロセス
6. ソフトウェアテスティング・ツール

<table>
<tr><td>ソフトウェアテスティングの基礎□</td><td>テストレベル□</td><td colspan="2">テスト手法□</td><td>テストに関係した計量尺度□</td><td>テストプロセス□</td><td>ソフトウェアテスティング・ツール</td></tr>
<tr><td>□</td><td>□</td><td colspan="2">□</td><td>□</td><td>□</td><td>□</td></tr>
<tr><td>テスティングに関係する用語□</td><td>テスト・ターゲット□</td><td>ソフトウェアエンジニアの直感および経験に基づいた手法□</td><td>利用に基づいた手法□</td><td>テストされるプログラムの評価□</td><td>実践上考慮されるべきことがら□</td><td>テスティングツール・サポート□</td></tr>
<tr><td>主要な論点□</td><td>テスティングの目的□</td><td>仕様に基づいた手法□</td><td>モデル準拠テスティング手法□</td><td>実施されたテストの評価□</td><td>テストアクティビティ□</td><td>ツールのカテゴリ</td></tr>
<tr><td>テスティングがもつ他のアクティビティとの関係□</td><td rowspan="2">□</td><td>コードに基づいた手法□</td><td>アプリケーションの性質に基づいた手法□</td><td rowspan="2">□</td><td rowspan="2">□</td><td rowspan="2">□</td></tr>
<tr><td>□</td><td>□</td><td>フォールトに基づいた手法□</td><td>選択および組み合わせ手法□</td></tr>
</table>

第 5 章 ソフトウェア保守

ソフトウェア保守は、ソフトウェアに対する費用ー効果的サポートを提供するために必要なアクティビティ全体であると定義される。これらには、ソフトウェアプロダクトの引き渡し後に行われるソフトウェア是正、訓練、および相談機能の運用に加え、引き渡し後の運用、保守容易性、および引き継ぎ自体の後方支援決定に関する計画など、引き渡し前のアクティビティが含まれる。「ソフトウェア保守」知識領域は、ソフトウェアエンジニアリングにおける他のすべての側面との関係があるとされ、以下の各項に示されるような内容が記述されている。

1. ソフトウェア保守の基礎
2. ソフトウェア保守における主な課題
3. 保守プロセス
4. 保守者のための技法
5. ソフトウェア保守ツール



第 6 章 ソフトウェア構成管理

システムの構成とは、テクニカルに文書化されるか、またはプロダクトとして達成された姿で提示された、ハードウェア、またはソフトウェアの機能的および物理的特性を指す。そして、構成管理は、公式に次のように定義されている。

「構成要素の機能的および物理的特性を識別し、文書化すること、これらの特性に対する変更をコントロールすること、変更に対する処理および実現の実態を記録し、報告すること、および仕様化された要求に適合していることを検証すること、以上のことがらに対して、テクニカルおよび経営上の指示および観察を適用する実践規律である」(ISO/IEC/IEEE 24765:2010 Systems and Software Engineering - Vocabulary, ISO/IEC/IEEE, 2010)

構成管理の中でも特に、ソフトウェア構成管理(SCM)は、プロジェクトマネージメント、開発および保守アクティビティ、品質保証アクティビティ、ならびに顧客および最終プロダクトのユーザーに対して便益を提供することを目的としており、「ソフトウェア構成管理」知識領域においては、以下の項目が述べられている。

1. SCM プロセスのマネジメント
2. ソフトウェア構成の同定
3. ソフトウェア構成コントロール
4. ソフトウェア構成実態説明
5. ソフトウェア構成監査
6. ソフトウェアリリース・マネジメントおよび引渡し
7. ソフトウェア構成管理ツール

各項目は、次ページの図に示される項目で構成されている。

SCMプロセスのマネージメント□	ソフトウェア構成の同定□	ソフトウェア構成コントロール□	ソフトウェア構成実態説明□	ソフトウェア構成監査□	ソフトウェアリリース・マネージメントおよび引渡し□	ソフトウェア構成管理ツール□
□	□	□	□	□	□	□
SCMのための組織的背景□	コントロールされるべき構成要素の同定□	ソフトウェア変更の要請、評価、および承認□	ソフトウェア構成実態情報□	ソフトウェア機能的構成監査□	ソフトウェアビルディング□	□
SCMプロセスに対する制約とガイダンス□	ソフトウェア・ライブラリ□	ソフトウェア変更の実現□	ソフトウェア構成実態報告□	ソフトウェア物理的構成監査□	ソフトウェアリリースマネジメント□	□
SCMのための計画□	□	逸脱と放棄□	□	ソフトウェアベースラインのインプロセス監査□	□	□
SCM計画□	□	□	□	□	□	□
ソフトウェア構成管理の観察□	□	□	□	□	□	□

第 7 章 ソフトウェアエンジニアリング・マネジメント

ソフトウェアエンジニアリング・マネジメントは、ソフトウェアプロダクトおよびソフトウェアエンジニアリングサービスが効率的、かつ効果的に引き渡され、利害関係者の便益になっていることを確認するために行う、マネジメント・アクティビティ(すなわち、計画づくり、協調化、計量、モニタリング、コントロール、および報告)の適用である、と定義される。これらのアクティビティは、組織およびインフラストラクチャ・マネジメント、プロジェクトマネジメント、および計量にかんする計画的活動のマネジメントに分類される。「ソフトウェアエンジニアリング・マネジメント」知識領域では、特に後ろの2つのアクティビティが詳細に網羅されている。

1. 始動および適用範囲の定義
2. ソフトウェアプロジェクト計画
3. ソフトウェアプロジェクトの計画実践
4. レビューおよび評価
5. 終結
6. ソフトウェアエンジニアリング計量
7. ソフトウェアエンジニアリング・マネージメントツール

始動および適用範囲の定義	ソフトウェアプロジェクト計画	ソフトウェアプロジェクトの計画実施	レビューおよび評価	終結	ソフトウェアエンジニアリング計量	ソフトウェアエンジニアリング・マネジメントツール
要求の決定と折衝	プロセス計画	計画の実現	要求充足度の判定	終結の決定	計量約定の確立と持続	
実現可能性分析	納入成果物の決定	ソフトウェア調達および供給者契約のマネジメント	実行状況のレビューと評価	終結アクティビティ	計量プロセスの計画	
レビューおよび要求改訂のためのプロセス	工数、工程、およびコスト見積もり	計量プロセスの実現			計量プロセスの実施	
	資源割付	モニタリングプロセス			計量評価	
	リスクマネジメント	コントロールプロセス				
	品質管理	報告				
	計画マネジメント					

第 8 章 ソフトウェアエンジニアリングプロセス

エンジニアリングプロセスは、1つまたはそれ以上の入力から、必要な資源を消費しながら、出力への変換を達成するために必要な、互いに関連するアクティビティ集合から構成される。ソフトウェアエンジニアリングプロセスは、ソフトウェアエンジニアによって遂行される、開発、保守、および運用に関する作業アクティビティのことである。「ソフトウェアエンジニアリングプロセス」知識領域では以下の項目を扱う。

1. ソフトウェアプロセス定義
2. ソフトウェアライフサイクル
3. ソフトウェアプロセス査定および改善
4. プロセスおよびプロダクト計量
5. ソフトウェア・エンジニアリングプロセスツール

ソフトウェアプロセス定義□	ソフトウェアライフサイクル□	ソフトウェアプロセス査定および改善□	ソフトウェア計量□	ソフトウェアエンジニアリングプロセスツール□
□	□	□	□	□
ソフトウェアプロセス・マネジメント	ソフトウェアプロセスのカテゴリ□	ソフトウェアプロセス査定モデル□	ソフトウェアプロセスおよびプロダクト計量□	□
ソフトウェアプロセス・インフラストラクチャ□	ソフトウェアライフサイクルモデル□	ソフトウェアプロセス査定法□	計量結果の品質□	□
□	ソフトウェアプロセス適応□	ソフトウェアプロセス改善法□	ソフトウェア情報モデル□	□
□	実践上考慮すべきことがら□	連続的および段階的ソフトウェアプロセス格付け□	ソフトウェアプロセス計量手法□	□

第 9 章 ソフトウェアエンジアリングモデルおよび方法[12]

ソフトウェアエンジニアリングモデルおよび方法は、ソフトウェアエンジニアリング・アクティビティを体系化し、再現可能化し、かつ最終的により成功に結びつくようにすることを目標とした構造(モデル)と、その適用を表す。モデルの利用は、問題解決、表記、モデル構築および分析に必要な手続きに対する手法を提供し、方法は、目的対象であるソフトウェアおよび関連する作業成果物に関する、体系的な仕様化、設計、構築、テストおよび検証へ向けた手法を提供する。本知識領域は以下の項目からなる。

1. モデリング
2. モデルの型
3. モデルの分析
4. ソフトウェアエンジニアリング方法

モデリング□	モデルの型□	モデルの分析□	ソフトウェアエンジニアリング方法□
□	□	□	□
モデリング原則□	情報モデリング□	網羅性の分析□	発見的方法□
モデルの特性と表現□	振る舞いモデリング□	一貫性の分析□	形式的方法□
構文規則、意味規則、および実用体系□	構造モデリング□	正当性の分析□	プロトタイピング法□
前提条件、帰結条件、および不変条件□	□	追跡可能性□	アジャイル法□
□	□	相互作用の分析□	□

[12] 日本語訳本においては、方法と訳されているが、原文の methods と言う形のまま覚えた方が理解しやすいかもしれない。方法論などと訳されることもある。

第 10 章 ソフトウェア品質

ソフトウェア品質が言及する対象は、ソフトウェアプロダクトに対して期待する特徴であり、特定されたソフトウェアプロダクトが保有する特徴が及ぶ範囲であり、これら特徴を達成するために利用されるプロセス、ツールおよび手法である。「ソフトウェア品質」知識領域は、定義、実践規則、ツール、およびソフトウェア品質を定義するための手法、開発、保守、および実装にまたがる期間におけるソフトウェア品質状態を査定するための手法を包括した概要を提供する。

1. ソフトウェア品質の基礎
2. ソフトウェア品質管理プロセス
3. 実践上考慮すべきことがら
4. ソフトウェア品質ツール

ソフトウェア品質の基礎□	ソフトウェア品質管理プロセス□	実践上考慮すべきことがら□	ソフトウェア品質ツール□
□	□	□	□
ソフトウェアエンジニアリング文化および倫理□	ソフトウェア品質保証□	ソフトウェア品質要求□	□
品質の価値およびコスト□	検証と妥当性確認□	欠陥の特徴づけ□	□
モデルおよび品質特性□	レビューおよび監査□	ソフトウェア品質管理技法□	□
ソフトウェア品質改善□	□	ソフトウェア品質計量□	□
ソフトウェア安全性□	□	□	□

第 11 章 ソフトウェアエンジニアリング専門技術者実践規律

「ソフトウェアエンジニアリング専門技術者実践規律」知識領域は、ソフトウェアエンジニアが、専門職業人として、責任をもち、倫理に基づいた行動姿勢で、ソフトウェアエンジニアリングを実践するために見につけていなければならない、知識、スキル、および心構えを扱っている。「専門技術者実践規律」という用語は、なんらかの標準、または評価基準を達成するように、サービスを実施する方法に向けられたものと解釈出来る。ここで言う、標準および評価基準は、テクニカル、および非テクニカル両面を含んでおり、一般的に認められた知識体、倫理綱領、違反時処罰を伴った行動指針、認定、認証、免許の実施に必要な合意されたプロセスなどを含む。

1. 専門技術者意識
2. グループダイナミクスおよび心理学
3. コミュニケーションスキル

専門技術者意識□		**グループダイナミクスおよび心理学□**	**コミュニケーションスキル□**
□	□	□	□
認定、認証、および免許□	雇用契約□	チーム/グループにおけるワーキングダイナミクス□	書見、理解、および要約□
倫理綱領および専門技術者行動指針□	法律上の論点□	個人の認知力□	記述□
専門技術者・学/協会の性質と役割□	文書□	問題の複雑性に対する取り組み□	チームおよびグループコミュニケーション□
ソフトウェアエンジニアリング標準の性質と役割□	トレードオフ分析□	利害関係者との対話□	説明スキル□
ソフトウェアの経済的影響□	□	不確実性および曖昧性の扱い□	□
	□	多文化環境の扱い□	□

第 12 章 ソフトウェアエンジニアリング経済学

「ソフトウェアエンジニアリング経済学」は、ビジネス環境におけるソフトウェアエンジニアリングに関係する意思決定に必要な知識領域である。ソフトウェアプロダクト、ソフトウェアサービス、およびソフトウェアソリューションの成功は、良好なビジネスマネジメントに依存する。本知識領域では、ソフトウェアビジネスとソフトウェア開発・エンジニアリングとの関係性について概要を説明する。基本的事項、主要な用語の使い方、基礎概念、およびソフトウェアエンジニアリング経済学に関する共通の実践要綱を網羅し、ソフト絵羽えんジニアリングにおける意思決定が、どのようにビジネス上見通しを含めているか、また含むべきか、を提供する。

1. ソフトウェアエンジニアリング経済学の基礎
2. ライフサイクル経済学
3. リスクおよび不確実性
4. 経済学的分析後
5. 実践上考慮すべきことがら

ソフトウェアエンジニアリング経済学の基礎□		ライフサイクル経済学□		リスクおよび不確実性□	経済学的分析法□		実践上考慮すべきことがら□
□	□	□	□	□	□	□	□
財政□	減価償却□	プロダクト□	計画基準期間□	ゴール見積もりおよび計画□	利得志向・意思決定分裂□	コスト・効果性分析□	「グッド・イナフ」方針□
会計□	課税□	プロジェクト□	価値および価格設定□	見積もり手法□	最小許容利益率□	損益分岐点分析□	摩擦なき経済□
コントローリング□	通貨の時間価値□	プログラム□	コストおよびコスト算定□	不確実性の扱い□	投資利益率□	ビジネスケース□	エコシステム□
キャッシュフロー□	効率□	ポートフォリオ□	業績計量□	優先度づけ□	使用資本利益率□	多属性評価□	オフショアリングおよびアウトソーシング□
意思決定プロセス□	効果□	プロダクトライフサイクル□	アーンド・バリュー・マネジメント□	リスク条件下・意思決定□	コスト・便益分析□	最適化分析□	□
価値評価□	生産性□	プロジェクトライフサイクル□	終結決定□	不確実条件下・意思決定□	□	□	□
インフレーション□	□	提案□	交換および退役撤収決定□	□	□	□	□
	□	投資決定□	□	□	□	□	□

第 13 章 計算基礎

「計算基礎」知識領域の範囲は、開発およびソフトウェアが進化・実行される運用環境を包含する。ソフトウェアはコンピュータなしでは実行できないため、運用環境の中心にあるのは、コンピュータであり、それに付帯する様々なコンポーネントである。コンピュータおよびそれの基礎をなすハードウェアおよびソフトウェアの原理に関する知識は、ソフトウェアエンジニアリングを定着させるフレームワークとして役立つ。本知識領域においては、以下に示すような項目を扱う。

1. 問題解決手法
2. 抽象化
3. プログラミング基礎
4. プログラミング言語基礎
5. デバッギングツールおよび手法
6. データ構造と表現
7. アルゴリズムと複雑性
8. システムの基本概念
9. コンピュータ組織
10. コンパイラ基礎
11. オペレーティングシステム基礎
12. データベース基礎およびデータマネジメント
13. ネットワーク通信基礎
14. 並列および分散コンピューティング
15. ユーザ・ヒューマンファクタ基礎
16. 開発者ヒューマンファクタ基礎
17. セキュアソフトウェア開発および保守

第 14 章 数学基礎

「数学基礎」知識領域では、数を扱い、議論する一般的な算数とは全く異なり、論理と数論などソフトウェアが取り組まなければならない数学の本質に焦点を当てている。本知識領域は、考察しているシステムの背景を推論するために役立つテクニックを包含し、ソフトウェアエンジニアが数学的手段で行うべき推論および判定を表明し、これを先取りして進めることに役立つテクニックを定義し、論述している。以下は本知識領域にて説明される項目となる。

1. 集合，関係，関数
2. 論理基礎
3. 証明法
4. 計数基礎
5. グラフおよび木
6. 離散確率
7. 有限状態機械
8. 文法
9. 数値精度，正確度，誤差
10. 数論
11. 代数的構造

第 15 章 エンジニアリング基礎

IEEE は、エンジニアリングを「構造、機械、製品、システムまたはプロセスに対する、体系的、実践規律に基づいた、定量化可能な手法の適用である」と定義している。
「エンジニアリング基礎」の知識領域では、ソフトウェアエンジニアにとって有益な、いくつかの基本的エンジニアリングスキルおよびテクニックの概要を示す。ソフトウェアエンジニアリングに関する理論および実践慣行が成熟するに従って、ソフトウェアエンジニアリングは、すべてのエンジニアリング実践規律が共有する知識とスキルをベースにした、1つのエンジニアリング実践規律であることが明らかになってくる。本領域で扱うトピックスには、経験的方法および実験的テクニック(統計的分析、計量、エンジニアリング設計、モデリング、プロトタイピングおよびシミュレーション、標準、および根源原因分析)を包含している。

1. 経験的方法および実践的テクニク
2. 統計的分析
3. 計量
4. エンジニアリング設計
5. モデリング, シミュレーションおよびプロトタイピング
6. 標準
7. 根源原因分析

2.4 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るが、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る。以下、IEEE による代表的な定義。

■ IEEE によるソフトウェア工学の定義

IEEE[13]では、以下のように定義されている(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

■ ソフトウェアライフサイクル

上述の IEEE による定義に現れる、「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退役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る。つまり、ソフトウェア工学は「ソフトウェアの生成から退役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」であると言うことが出来る。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とは、どのようなものなのだろうか。

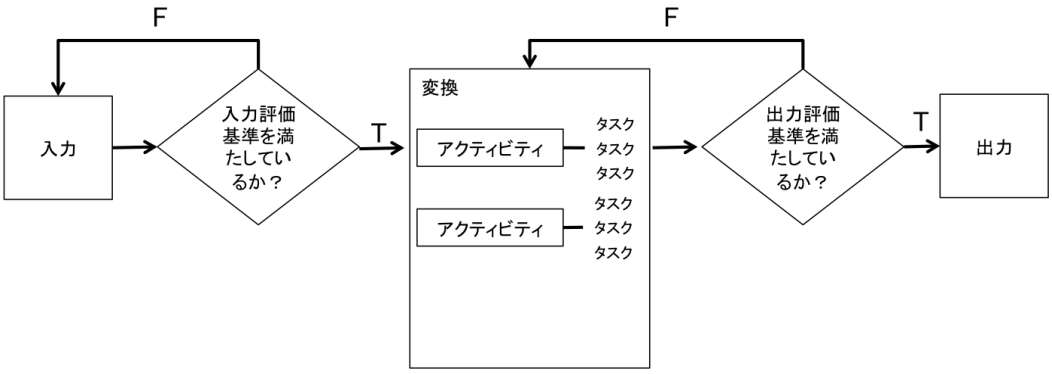
ソフトウェアライフサイクルの定義に関連して、SWEBOK ガイドでは、ソフトウェアプロセスとソフトウェアライフサイクルモデルという概念が触れられている。説明の都合上、ソフトウェアプロセス、ソフトウェアライフサイクル、ソフトウェアライフサイクルモデルの順に説明をする。以下、原文及び日本語訳版を参考に著者が拙訳を加えている。原文を参照される場合は、第8章:ソフトウェアエンジニアリングプロセス(松

13 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。

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 V3.0』. 2014, オーム社 p147-p166)を参照されたい。

■ ソフトウェアプロセスの定義

ソフトウェアプロセスは、作業成果物を入力し、これを出力作業成果物へ変換する、互いに関連するアクティビティおよびタスクの集合である。1つのソフトウェアプロセスに関する記述は、少なくとも、要求された入力、変換に必要な作業アクティビティ、および生成される出力を含む。下図は、ソフトウェアプロセスの要素を表したもの。図に示されるように、1つのソフトウェアプロセスは、入力評価基準、出力評価基準、および、作業アクティビティのタスクへの分割を含んでもよい。また、1つのソフトウェアプロセスは、サブプロセスを包含することができる。例えば、ソフトウェア要求妥当性確認は、要求がソフトウェア開発にとって適切な要素を提供しているかを確認するプロセスであるが、これは、ソフトウェア要求プロセスのサブプロセスの1つである。



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]"。

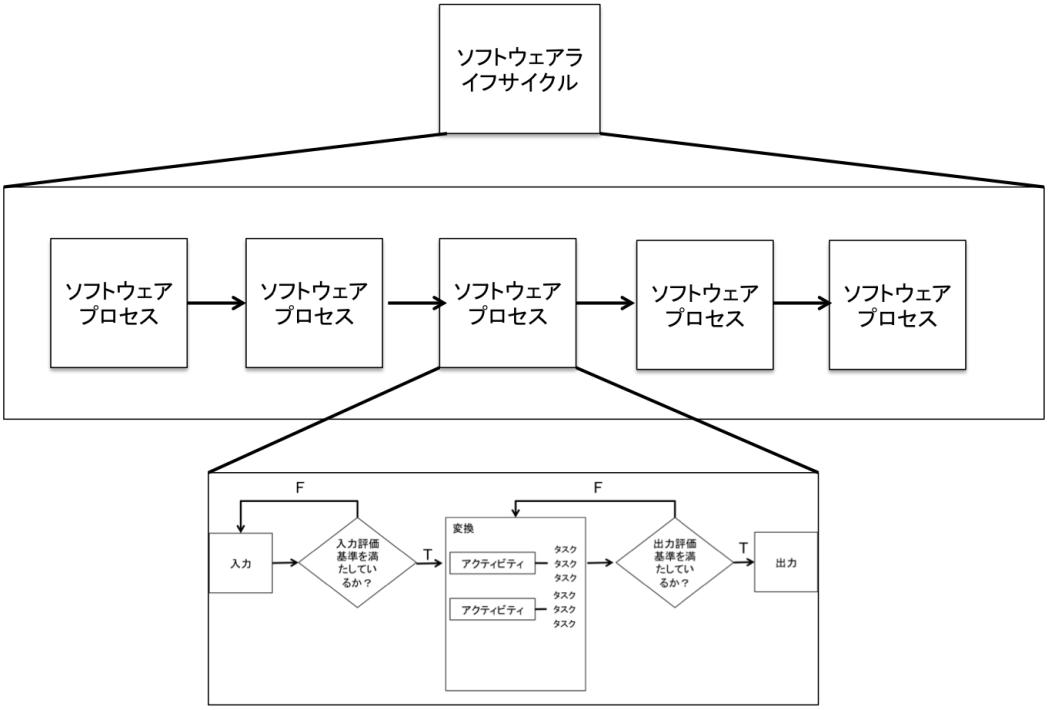
■ ソフトウェアライフサイクル

ソフトウェアライフサイクルとは、ソフトウェア開発において、一般的には、ソフトウェアの構想から廃棄に至るまでの一連のライフサイクルを表す。サイクルと言う名の通り、一度の構想から廃棄までではなく、それが連なり繰り返されることを想定していることが多い。

SWEBOK V3.0 では、

ソフトウェア開発ライフサイクル:ソフトウェア要求を引渡し可能なソフトウェアプロダクトに特定化し、変化するソフトウェアプロセスを包含したもの

ソフトウェアプロダクトライフサイクル:ソフトウェア開発ライフサイクルに加え、追加ソフトウェアプロセスを方眼紙、ソフトウェアプロダクトの実装、保守、サポート、進化、廃棄、などのプロセスを包含する。などが紹介されている。ソフトウェアプロダクトライフサイクルがそうであるように、あるソフトウェアライフサイクルが、他のソフトウェアライフサイクルを包含することも多々あり、基本的には個々に時間的順序を持たないソフトウェアプロセス及びソフトウェアライフサイクルそのものに、時間的、論理的関係を与えるものである。



■ 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モデルは、開発の途上に生じる各局面(phases)に関する抽象度の高い定義を提供する。ソフトウェアライフサイクルモデルは、主要なアクティビティとアクティビティ(ここで言うアクティビティは主にソフトウェアプロセスだが、まれに他のソフトウェアライフサイクルを包含することもある)間の時間的、論理的相互依存関係を与える。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。

SWEBOK ガイドの規定をまとめれば、ソフトウェアプロセスは、それぞれの局面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み、ソフトウェアライフサイクルモデルは、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する。言い換えれば、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することとなる。

このような規定があるものの、ソフトウェアプロセスやソフトウェアライフサイクルモデルの概念は混同して用いられることも多いため注意が必要である。また、SWEBOK 2004 では「ソフトウェアライフサイクルプロセス」と呼ばれていた概念が、SWEBOK V3.0 では「ソフトウェアプロセス」に変わるなど用語レベルでも曖昧性が存在する。以下、様々な研究者、機関（組織）の提唱する、ソフトウェア（ライフサイクル）プロセスやソフトウェアライフサイクルモデルの定義を紹介する。

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）を挙げている[14]。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う。

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」などを挙げている[15]。

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」を挙げている[16]。

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」を挙げている[17]。『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」を挙げている[18]。『要求工学知識体系 第1版』では、「ソフトウェアプロセスモ

14 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

15 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

16 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

17 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

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

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る[19]。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」が挙げられている[20]。

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[21]。

それぞれの文献・資料の内容を以下の表にまとめた。

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？）

19 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

20 『SQuBOK ガイド第一版 β版』 p.71

21 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

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

以上からわかるように、SWEBOK V3.0 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る。

■ ISO/IEC 12207:1995 における概念規定

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、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を目的とする。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る[22]。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プロセスの関係を確認する。

JIS X 0160:1996 の目的

> (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。

22 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。

JIS X 0160:1996 の適用範囲

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(「1.5 制限」)

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[23]。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の概念(「3.11 ライフサイクルモデル(life cycle model)」)

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)

23 「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。

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

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。

2.5 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るが、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ある。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ない。

以下に主要なソフトウェアライフサイクルモデルを示す。

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。しかし、このモデルでは、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る。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。

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る。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を示す。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プロセスの内部で決定される。

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)を実行する。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を進める。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る。

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。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する。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る。

■ 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が、おおまかな作業区分を含む時系列モデルを決定している一方、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。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る。
プロセス→アクティビティ→タスクの順に詳細化、具体化する。

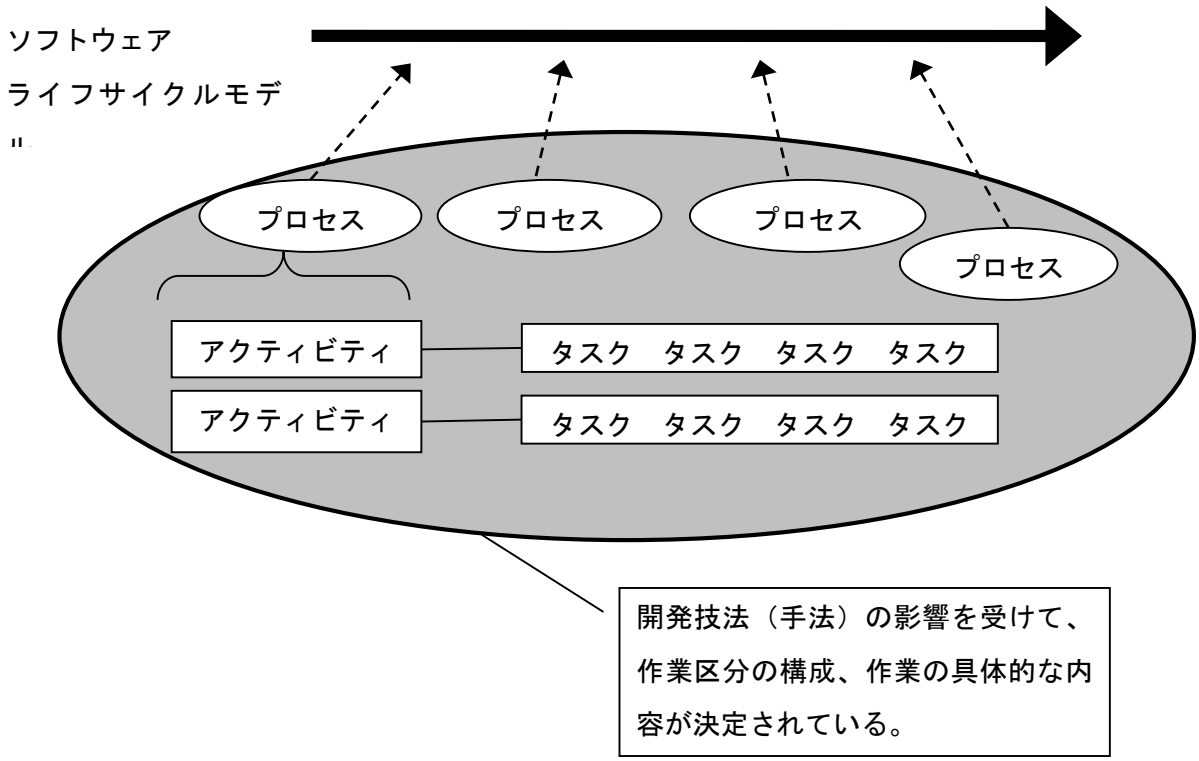
■ 開発技法(手法)

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)を意味する。

■ 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)になる。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る。

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る。


ソフトウェア
ライフサイクルモデ
"
プロセス
プロセス
プロセス
プロセス
アクティビティ
タスク　タスク　タスク　タスク
アクティビティ
タスク　タスク　タスク　タスク
開発技法（手法）の影響を受けて、作業区分の構成、作業の具体的な内容が決定されている。

3 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。

3.1 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。

3.2 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 情報サービス産業協会(JISA)、日本電子工業振興協会(現電子情報技術産業協会:JEITA)、日本情報システム・ユーザー協会(JUAS)、日本規格協会が、経済産業省産業構造審議会(以下産構審)と連携し、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
2006 年 産構審と連携し、モデル契約と共通フレームを改定
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

3.3 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)。

3.4 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。

3.5 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。

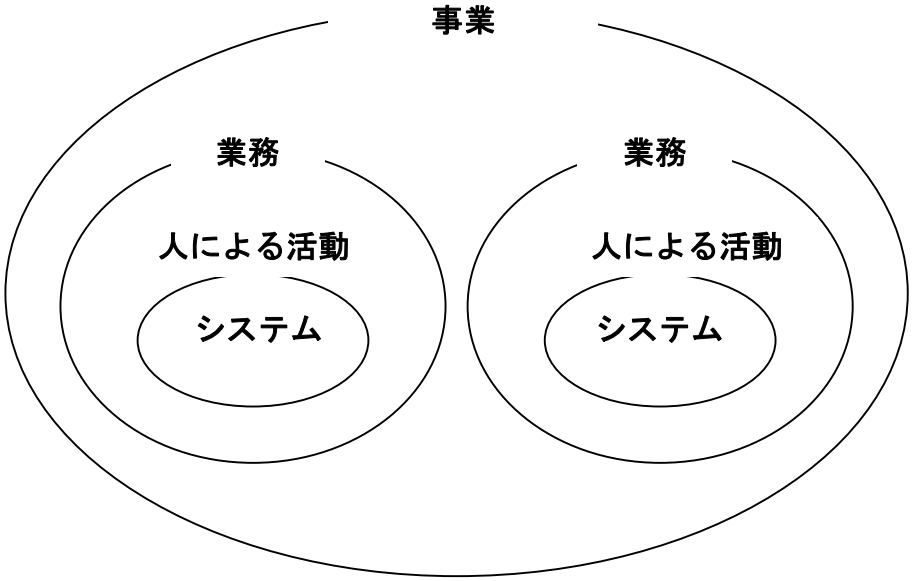
3.6 共通フレームの設計思想

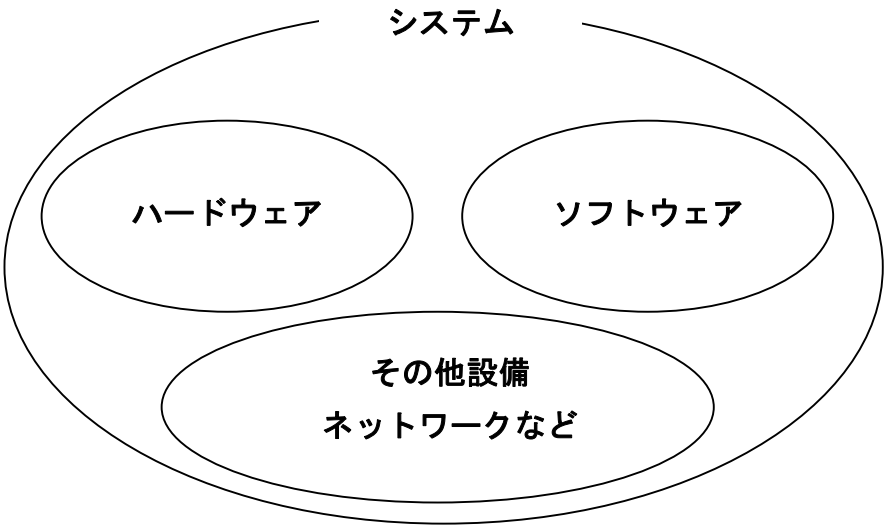
共通フレーム 2007 において提唱されている設計思想の概念として、システムの開発・利用に関わる組織構造を 4 つに分けて理解すると言う物がある。一方、共通フレーム 2013 では「5.1 共通フレーム作成における設計思想」と言う節において、共通フレームの骨格となる思想を 10 項目に渡り述べている。本節では共通フレーム 2007 で触れられていた 4 つの概念について詳説し、システムやソフトウェアと言った概念を理解した上で、共通フレーム 2013 において述べられる共通フレームの骨格となる思想 10 項目を紹介する。

■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[24]）

24 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

■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[25]。

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

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)。

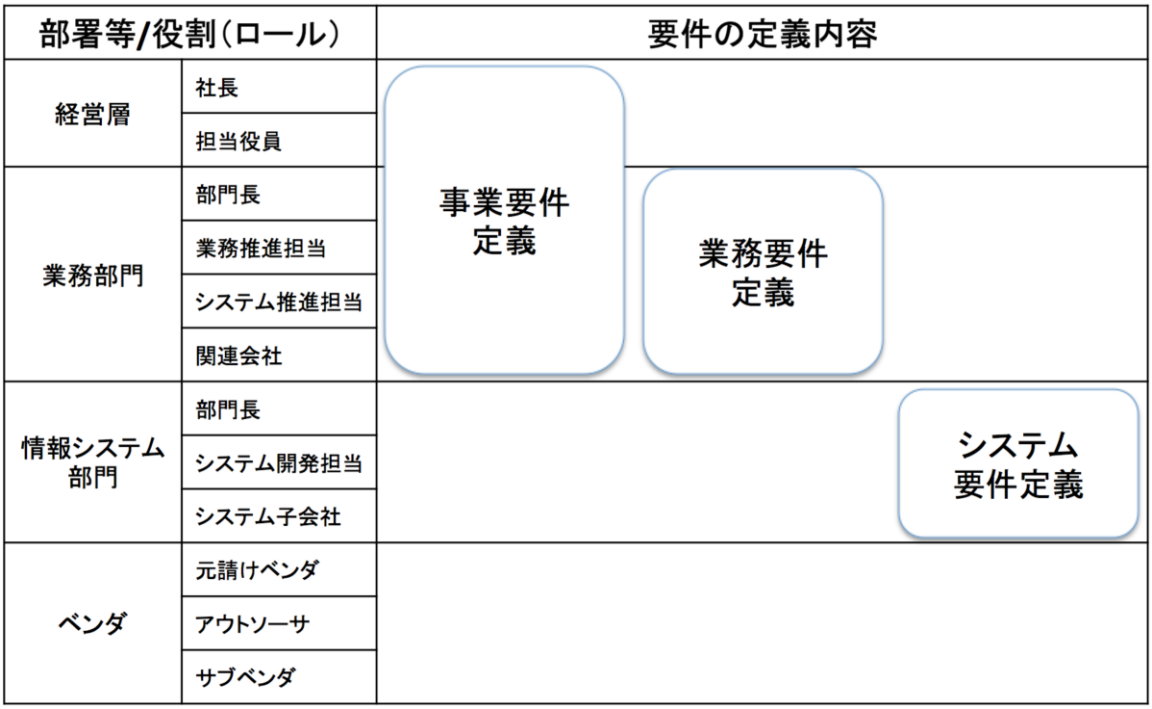
25 上記一覧表が掲載されている箇所では、4 つの階層と要件定義に関して、事業要件定義を経営層と業務層(業務部門)が、業務要件定義を業務層が、システム要件定義を IT システム層(情報システム部門)が担うとされており、事業要件定義は経営層と業務層にまたがって担われるものとして解説されています(『経営者が参画する要求品質の確保 第 2 版』 pp.42-43)。

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 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[26]、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)

26 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)[27]。

27 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。

例えば、遠隔地とのビデオ通話を実現するシステムを考えた時に、「前面カメラと背面カメラを切り替えられるようにしたい」、「通話相手を履歴から検索出来るようにしたい」、「グループ通話が出来るようにしたい」など、具体的にシステムとして実現出来るべき機能に注目した要求を機能要求と呼び、それらを機能に関する要求を違和感なく実現する為に「通信による遅延は 1 秒以内にする」、「予期せぬ人物が通話を傍受出来ないようにする」と言った品質に関する要求を非機能要求と呼ぶ。

【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

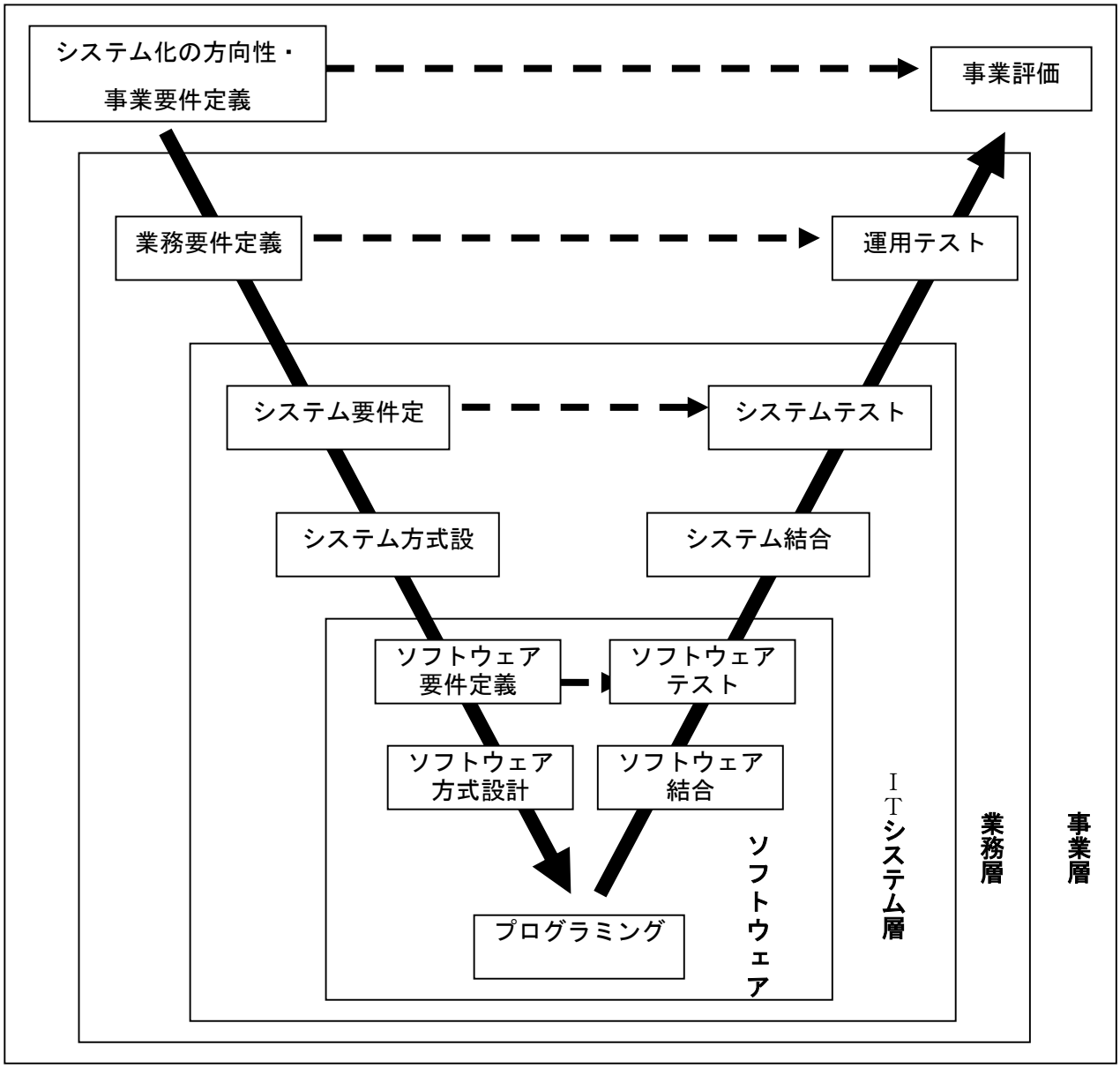
してもよいことになっている。この場合、経営層だから事業要件定義に、業務層だから業務要件定義に責任があるとはいえなくなる。

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。

3.7 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 合意プロセス

合意プロセスは、二つの組織の間の合意を確立するために必要なアクティビティを定義する。取得プロセスを呼び出すと、取得プロセスは、運用システムとして使用する製品の供給者、運用システムを支援するサービスの供給者又はプロジェクトで開発するシステム要素の供給者とともに業務を遂行する手段を提供する。供給プロセスを呼び出すと、供給プロセスは、取得者に納入する製品又はサービスを成果としてもたらすプロジェクトを遂行する手段を提供する。

1.1 取得プロセス

取得プロセスは、取得者が表明したニーズを満たす製品及び/又はサービスを得ることを目的とする。そのプロセスは、顧客のニーズの識別で始まり、取得者が必要とする製品及び/又はサービスの受入れで終わる。

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

供給プロセスは、合意した要件に合致した製品又はサービスを取得者に提供することを目的とする。

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 合意・契約の変更管理プロセス

合意・契約の変更管理プロセスの目的は、取得者と供給者の二者間で締結した契約内容に影響を与える変更要求が生じた場合、二者間で合意できる新たな契約内容を導くことにある。このプロセスは取得者又は供給者の変更要求の提示で始まり、変更要求の取下げ、全部又は一部了承など二者が合意する結論で終わる。

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

2 テクニカルプロセス

テクニカルプロセスは、組織及びプロジェクトの担当部門が技術的な決定及び行動の結果生じる利益を最適化し、リスクを軽減できるようにするアクティビティを定義する。これらのアクティビティは、製品及びサービスが適時性及び可用性、費用対効果、並びに取得組織及び供給組織に要求される機能性、信頼性、保守性、精算可能性、使用性及びその他の品質を可能にする。また、それらのアクティビティは、製品及びサービスが、健康、安全、セキュリティ及び環境上の要因を含めた、社会の期待又は社会の法制化された要求事項に適合することを可能にする。

2.1 企画プロセス

企画プロセス全体の目的は、経営・事業の目的、目標を達成するために必要なシステムに関係する要件の集合とシステム化の方針、及び、システムを実現するための実施計画を得ることである。システム化構想立案プロセスでは、経営課題を解決するための新たな業務とシステム構想を立案する。システム化計画の立案プロセスでは、システム化構想を具現化するための、システム化計画及びプロジェクト計画を具体化し、利害関係者の合意を得る。

2.1.1 システム化構想の立案プロセス

2.1.1.1 プロセス開始の準備

企画作業の組立て

必要なプロセスの組み込み

企画環境の準備

プロセス実施計画の作成

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

企画作業の組立て

必要なプロセスの組み込み

企画環境の準備

プロセス実施計画の作成

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

要件定義プロセスは、定義された環境において、利用者及び他の利害関係者が必要とするサービスを提供できるシステムに対する要件を定義することを目的とする。
このプロセスでは、システムのライフサイクルを通じて、システムに関わり合いをもつ利害関係者又は利害関係者の種類を識別子、そのニーズ及び要望を識別する。このプロセスは、これらを分析し、次のような要件定義の共通集合へ変換する。

- システムがその運用環境との間で行う意図された相互作用を表現する。
- システムがニーズを満たすことを確かめるために、妥当性確認する際、結果として得られた各運用サービスとの堆肥となる。
- 取得者側の利害関係者間での合意内容となる。

2.2.1 プロセス開始の準備

2.2.1.1 要件定義作業の組み立て

2.2.1.2 必要なプロセスの組み込み

2.2.1.3 要件合意及び承認ルールの決定

2.2.1.4 要件定義環境の準備

2.2.1.5　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

2.2.3.1　要件の抽出

2.2.3.2　制約条件の定義

2.2.3.3　利用者とシステム間の相互作用の識別

2.2.3.4　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

2.2.4.1 導出要件分析

2.2.5　要件の合意

2.2.5.1　要件の問題解決

2.2.5.2　利害関係者へのフィードバック

2.2.5.3　要件の確立

2.2.6　要件の記録

2.2.6.1 要件の記録

2.2.6.2 要件の追跡可能性の維持

2.3　システム開発プロセス

システム開発プロセスは、システム、ソフトウェア製品又はサービスを開発することを目的とする。このプロセスでは、要件を、顧客のニーズに合ったシステムへと変換する。このプロセスは、検証を通じて方式設計の要件を満足し、妥当性確認を通じて利害関係者の要件を満足するシステムを結果としてもたらすものである。

2.3.1　システム開発プロセス開始の準備

2.3.1.1　開発作業の組み立て

2.3.1.2　必要なプロセスの組み込み

2.3.1.3　開発環境の準備

2.3.1.4　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

ソフトウェア実装プロセスは、ソフトウェア製品又はソフトウェアサービスとして実現される指定のシステム要素を作り出すことを目的とする。

このプロセスでは、指定された動作、インタフェース及び実装上の制約条件を、ソフトウェア製品

又はソフトウェアサービスとして実現されたシステム要素、別名では“ソフトウェア品目”とされているシステム要素を作り出すことに変換する。このプロセスは、検証を通じて方式設計の要件を満足し、妥当性確認を通じて利害関係者の要件を満足するソフトウェア品目を結果としてもたらすものである。

2.4.1 ソフトウェア実装プロセス開始の準備プロセス

2.4.1.1 開発作業の組み立て

2.4.1.2 必要なプロセスの組込み

2.4.1.3 開発環境の整備

2.4.1.4 ソフトウェア実装プロセスの実施計画作成

2.4.1.5 非納入品目の使用の容認

2.4.2 ソフトウェア要件定義プロセス

2.4.2.1 ソフトウェア要件の確立

2.4.2.2 ソフトウェア要件の評価

ソフトウェア要件の共同レビューの実施

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

保守プロセスは、納入されたシステム及びソフトウェア製品に対して費用対効果が高い支援を提供することを目的とする。

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 運用・サービスプロセス

運用・サービスプロセスは、運用者が行うアクティビティ及びタスクを含む。このプロセスは、開発の終了したシステム、ソフトウェア製品、サービスなどの運用作業として、運用プロセス、廃棄プロセス、サービスマネジメントプロセスからなる。

3.1 運用プロセス

運用プロセスは、意図された環境でシステム及びソフトウェア製品を運用し、システム及びソフトウェア製品の顧客への支援を提供することを目的とする。(注: 本書の内容との関連が薄いため、大項目レベルでのプロセスのみを列挙する)

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
3.1.9 投資効果及び業務効果の評価

4 支援プロセス

支援プロセスでは、システム開発プロセス又はソフトウェア実装プロセスを実行するためのアクティビティの特定の絞り込んだ集合を提供する。支援するプロセスは、明確な目的をもつ不可欠な部分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を助け、プロジェクトの成功及び品質に寄与する。

4.1 文書化プロセス
4.2 品質保証プロセス

4.3　検証プロセス
4.4　妥当性確認プロセス
4.5　共同レビュープロセス
4.6　監査プロセス

5　プロジェクトプロセス

プロジェクトプロセスは次の二つの視点に分けられる。

a) プロジェクトマネジメントの視点
プロジェクト計画を確立して進展させ、計画に対する実績及び進捗のアセスメントを行い、プロジェクトの実行を完了時まで制御するために使用する。ライフサイクルのどの時点でも、プロジェクトの階層のどこからでも呼び出すことができる。
「プロジェクト計画プロセス」及び「プロジェクトアセスメント及び制御プロセス」からなる。

b) プロジェクトサポートの視点
特化した管理目標を実行するためのタスクの特定の絞り込んだ集合を提供し、組織全体から単一のライフサイクルプロセス及びそのタスクにわたる、どの取組みの管理においても全て存在する。
「意思決定管理プロセス」、「リスク管理プロセス」、「構成管理プロセス」、「ソフトウェア構成管理プロセス」、「情報管理プロセス」及び「測定プロセス」からなる。

6　組織のプロジェクトイネーブリングプロセス

組織のプロジェクトイネーブリングプロセスは、プロジェクトの開始、支援及び制御に酔って、製品又はサービスを取得及び供給するための組織の能力を管理する。このプロセスは、プロジェクトを支援するために必要な資源及び基盤を提供し、組織目標及び確立された合意を満足させることを確実にする。このプロセスは、組織の業務を管理できるようにする業務プロセスの包括的集合を意図してはいない。

6.1　ライフサイクルモデル管理プロセス
6.2　インフラストラクチャ管理プロセス
6.3　プロジェクトポートフォリオ管理プロセス
6.4　人的資源プロセス
6.5　品質管理プロセス
6.6　知識管理プロセス

6.7　ソフトウェア再利用プロセス

6.8　システム監査プロセス

7　プロセスビュー

プロセスビューは、特定の関心事に焦点を宛、その履行や達成に必要となる、目的や成果、“既定の”プロセス/アクティビティ/タスクを示すための表現法である。ISO/IEC 12207 や同 15288 の 2008 年版で新たに導入され、共通フレーム 2013 においても踏襲し取り入れている。

7.1　ユーザビリティプロセスビュー

8　テーラリングプロセス

テーラリング(修整)プロセスは、次のような特定の状況又は要因に対応するために、この共通フレームのプロセスを適応させることを目的とする。

a) 合意において、この共通フレームを用いる組織を取り巻く状況又は要因

b) この共通フレームを参照している合意を満たすために要求されるプロジェクトに影響する状況又は要因

c) 製品又はサービスを供給するために組織のニーズを反映する状況又は要因

8.1　テーラリング(修整)

8.1.1 影響の識別

8.1.2 重大な特性に関連するライフサイクルの考慮

8.1.3 当事者の意見聴取

8.1.4 テーラリング(修整)の決定

8.1.5成果、アクティビティ、タスクの削除

3.8 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[28]	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[29] 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)[30]	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。

28 「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。

29 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。

30 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28	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28	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。

4 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。

4.1 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[31]：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[32]。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

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

32　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。

4.2 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")[33]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

33 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」[34]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。

34 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。

4.3 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

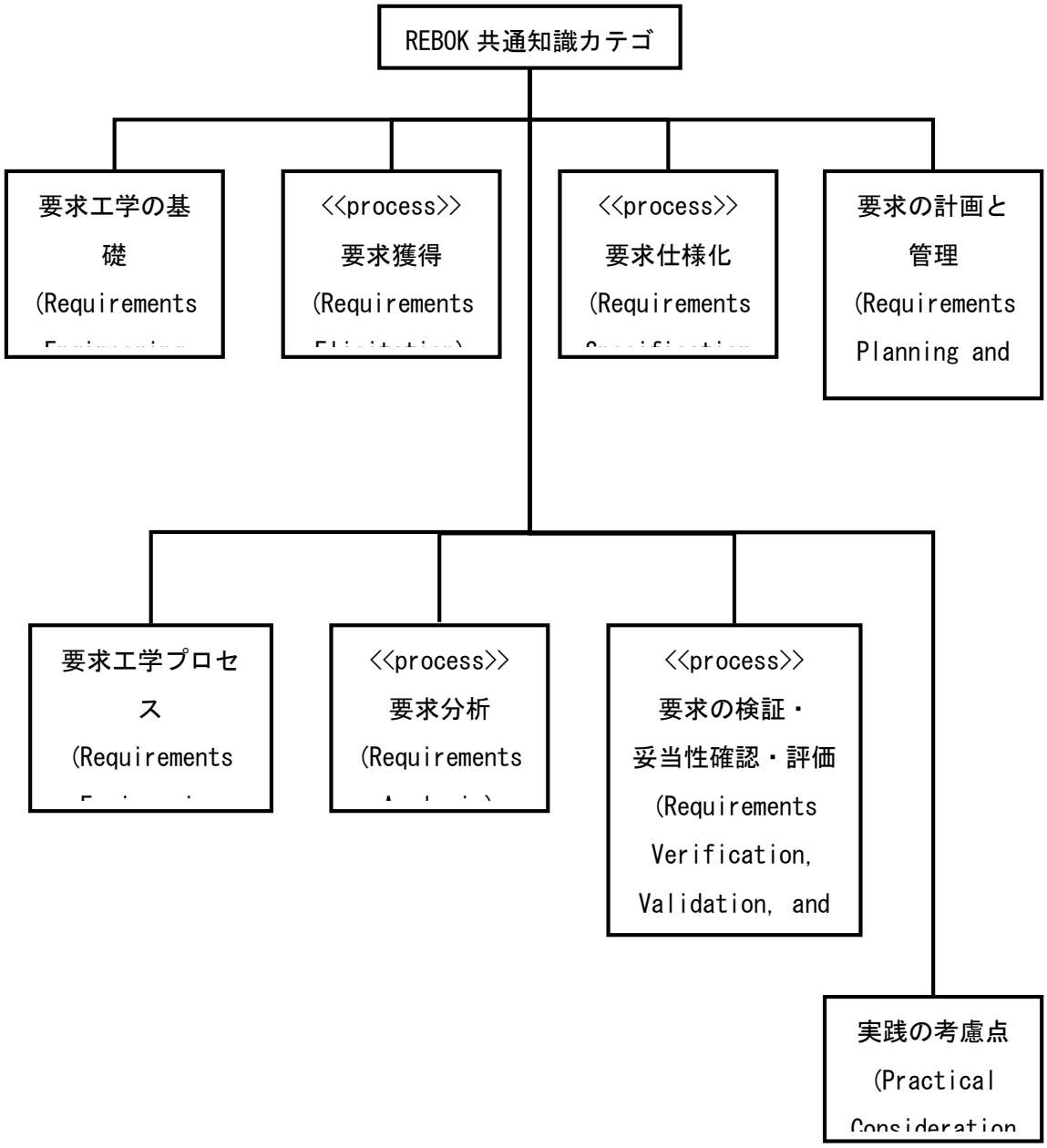
特定の「ドメイン」に依存せず、要求工学の知識として共通に利用される知識領域群である。

※「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[35])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。

35 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[36]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。

36　ここでいう「システム」概念は、共通フレーム 2013 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 2013 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[37]。
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。

37　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。

5 要求仕様にかかわる諸問題

5.1 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[38]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。

---

38 要求仕様書とは、顧客やステークホルダーからの要求が要件定義などを通して定まった際に、要件定義の成果物として作成されるドキュメントの事を表す。組織や開発体制などによって、求められる粒度や形式も異なるが、一般的に後続の開発プロセスにおいて必要な、どのような機能を作るべきで（機能要件）、それを実装する上で何を満たしているべきか（非機能要件）を網羅的に記述したものとなる。

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

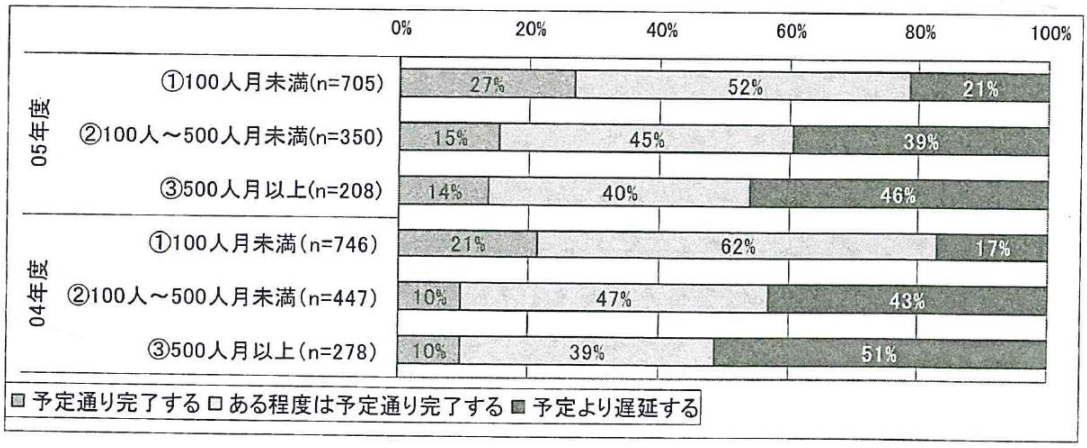
の2つの文章を比べてみると、同じことをイメージして書かれたものであっても、明らかに読み手がイメージする仕様は変わってくる。頭のなかでは同じことをイメージしていても、それをどのように文章するかで、「文章の伝える力」がまったく違ってくる。このことを理解していないと仕様のトラブルを防ぐことはできない。このように要件開発では、要求の気づき方だけでなく、要求や仕様をどのように表現するかということにも、注意を払う必要がある。

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。

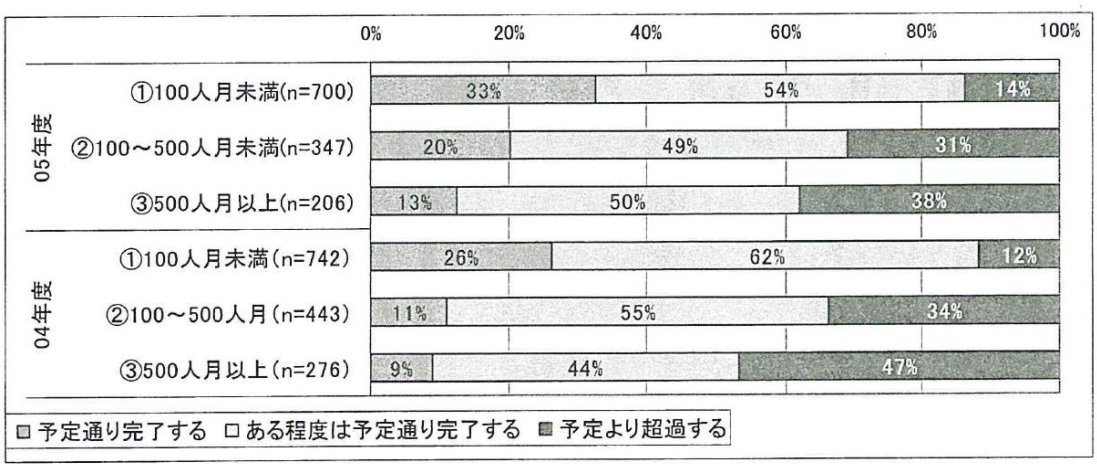
次に、要求仕様書に関わる問題について、具体的な調査結果からみていく。

5.2 要求仕様書に関わる問題と背景

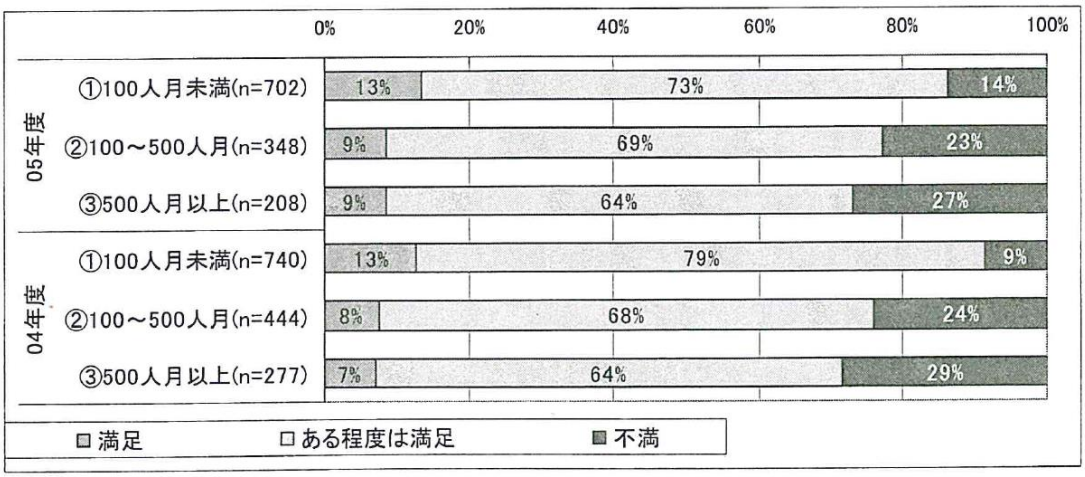
1. 日本情報システム・ユーザー協会(JUAS)によると、1994年より毎年ユーザー企業 IT 動向調査を行っており、2006年の調査報告で、システム開発に置ける工期、予算、及び品質についての満足度調査は以下のように結果を得ている。(図表1－1　システム開発における工期についての状況、図表 1-2　システム開発における予算の状況、図表 1-3　システム開発における品質の状況)



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

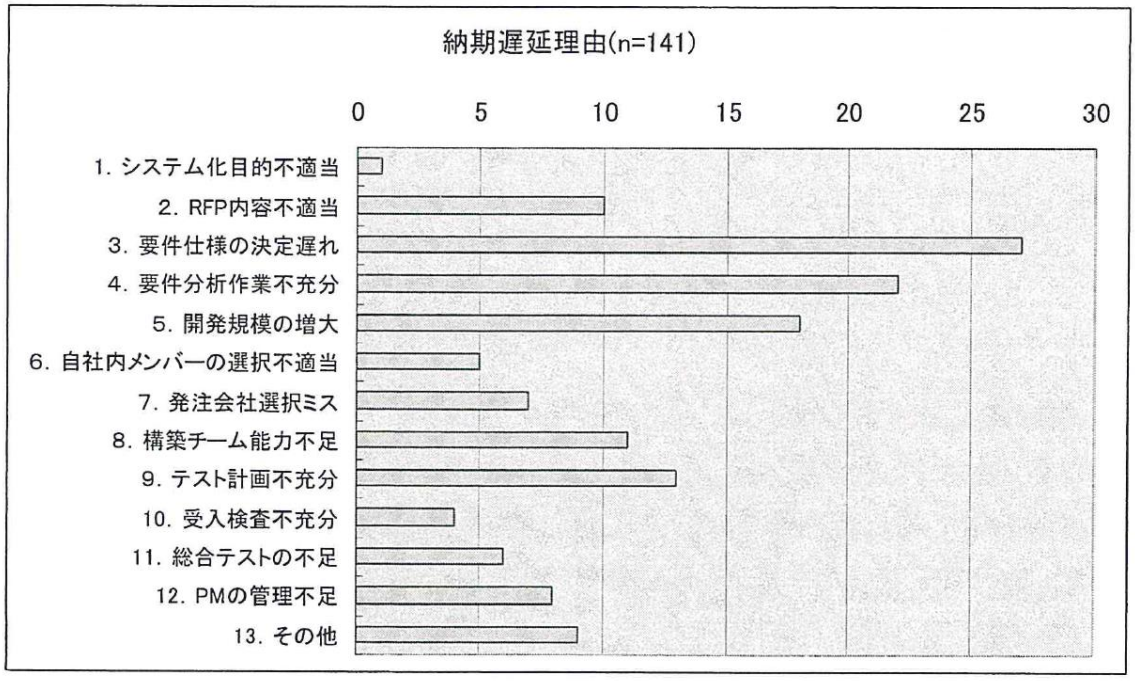
・開発予算の超過

・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

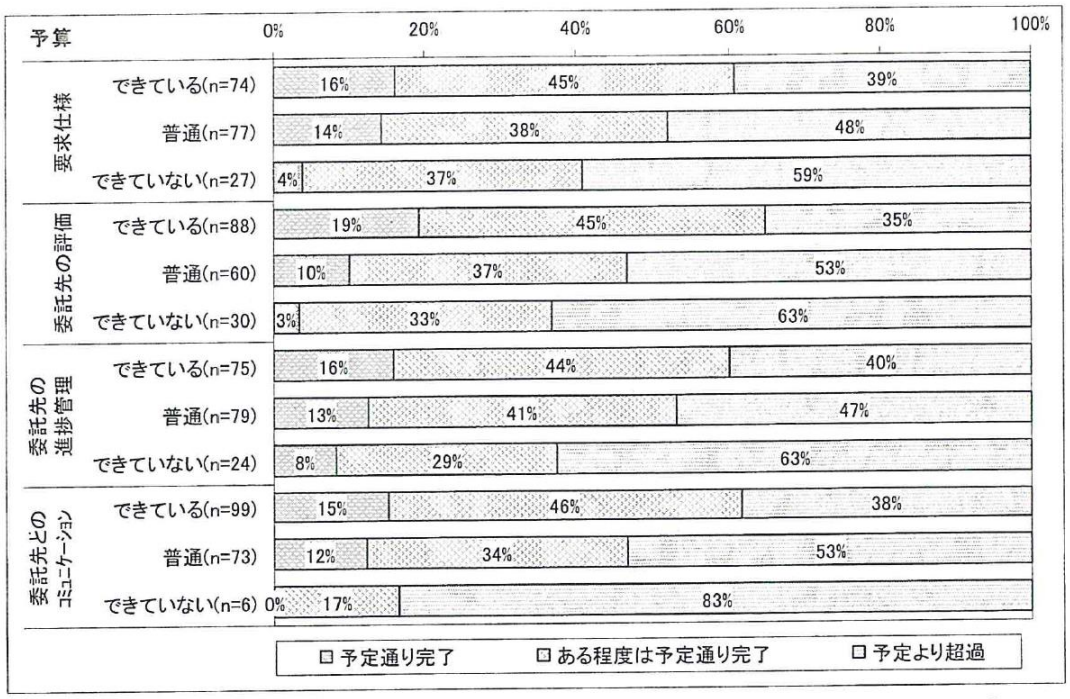
この図の中で、「2. RFP 内容不適当」と「3. 要件仕様の決定遅れ」、「4. 要件分析作業不十分」の3つが何らかの形で要求仕様書の作成に関わる項目である。グラフから明確なように、「3」と「4」は件数が多く、これに「2」を加えた3つの要件で59件、全体に占める割合では42%にもなっている。

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。

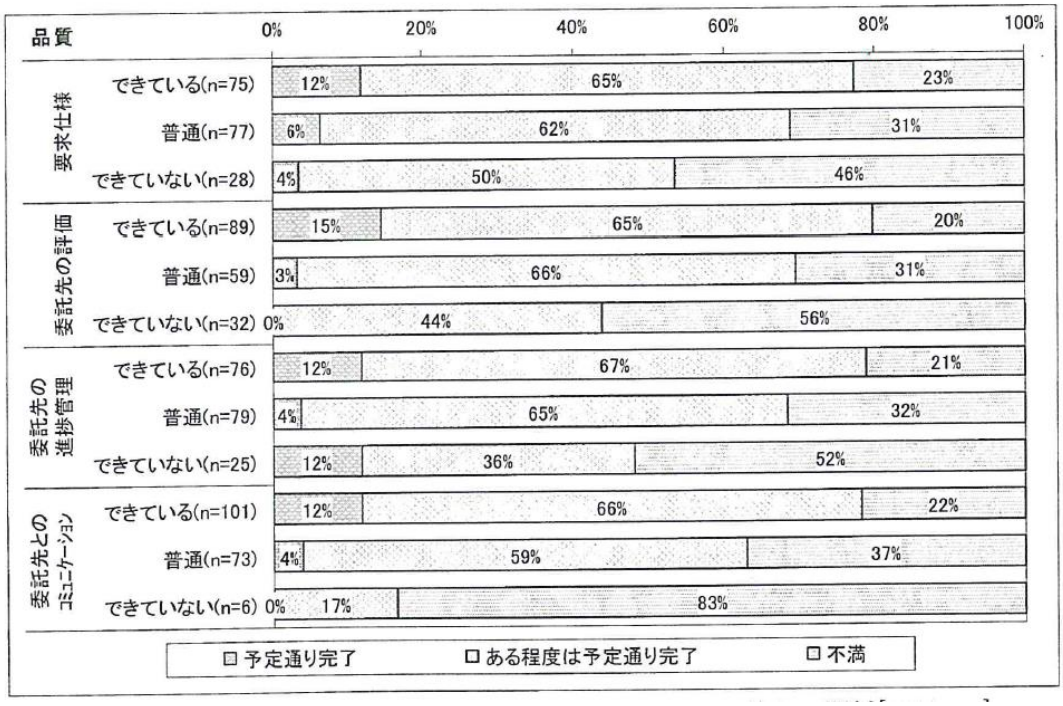
以上から、「要求仕様書をいかに書くか」ということが、いかに重要なテーマであるかということが分かる。

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)



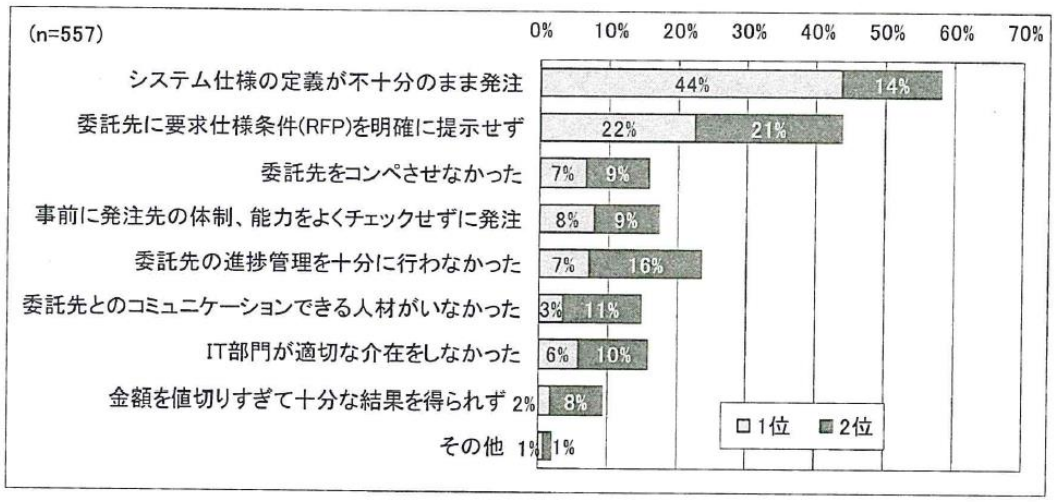
図表1－5　予算と要求仕様の提示を含むプロジェクト要件との関係



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

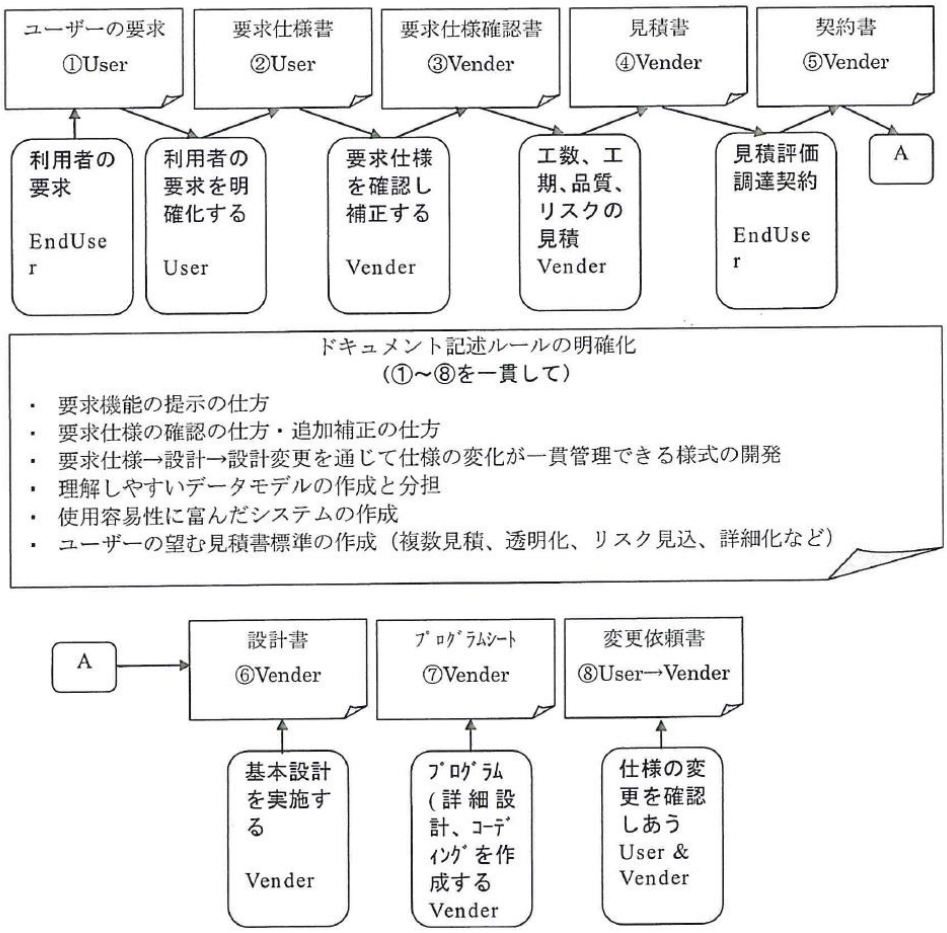
4. また、2006年度版の企業IT動向調査のアンケート調査によると、ソフトウェア開発を発注した後の「発注者としての反省点」を聞いた結果（図表 1-7　発注者としての反省点）において、要求仕様

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。

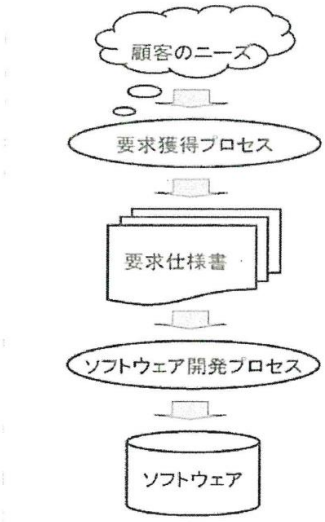
以上から、要求仕様書を的確に作成することは、非常に重要であることが分かる。

6 要求仕様書の意味

6.1 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。

6.2 「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。

6.3 「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。

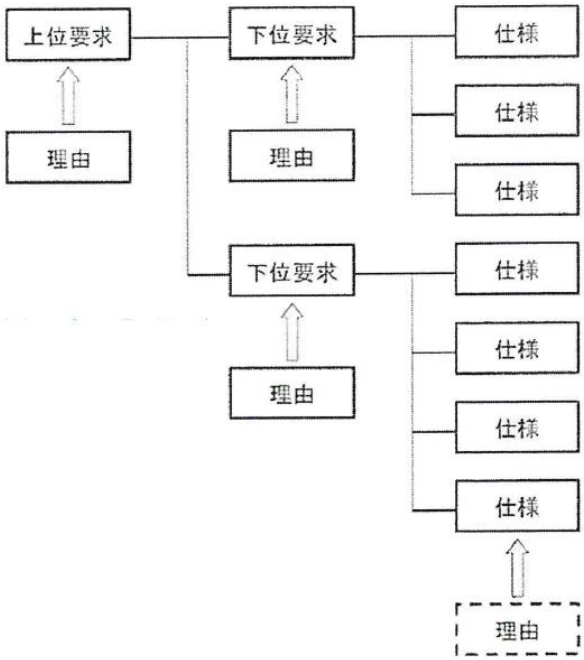
6.4 「仕様」とは何か

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。

6.5 「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。

6.6 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。

6.7 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。

6.8 「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。

6.9 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。

6.10 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

6.11 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。

7 要求仕様の定義方法

7.1 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。

7.2 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。

1. 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。

2. 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。

7.3 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

8 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)

8.1 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

8.2 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。

8.3 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。

8.4 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。

8.5 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

8.6 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）

8.7 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)。

8.8 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。

8.9 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。

8.10 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

8.11 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

8.12 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

8.13 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

8.14 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

8.15 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

8.16 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

8.17 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

8.18 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%

8.19 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。

8.20 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。

8.21 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。

8.22 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。

8.23 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。

9 USDM による要求仕様書作成

9.1 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。

9.2 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

9.3 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	

9.4 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。

9.5 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

9.6 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	

9.7 要求の仕様化

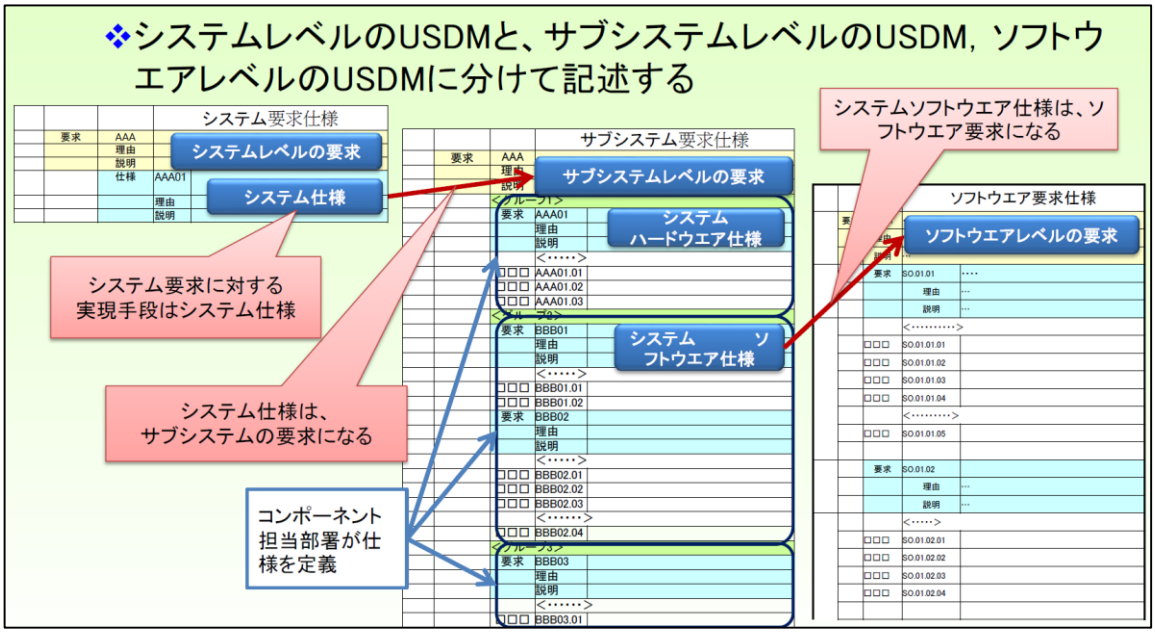
1. 要求の仕様化は、要求が適切に表現されていなければ仕様は正しく引き出すことができない。ソフトウェアとしてソースコードに変換されるのは仕様であり、正しく仕様化できなければモレが発生し後戻り作業が発生する。要求の仕様化を行うには、要求の中にある動詞の単位で仕様を設定する(目的語も仕様をもつことがある)。

＜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)

9.8 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＞

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	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。

9.9 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。

9.10 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

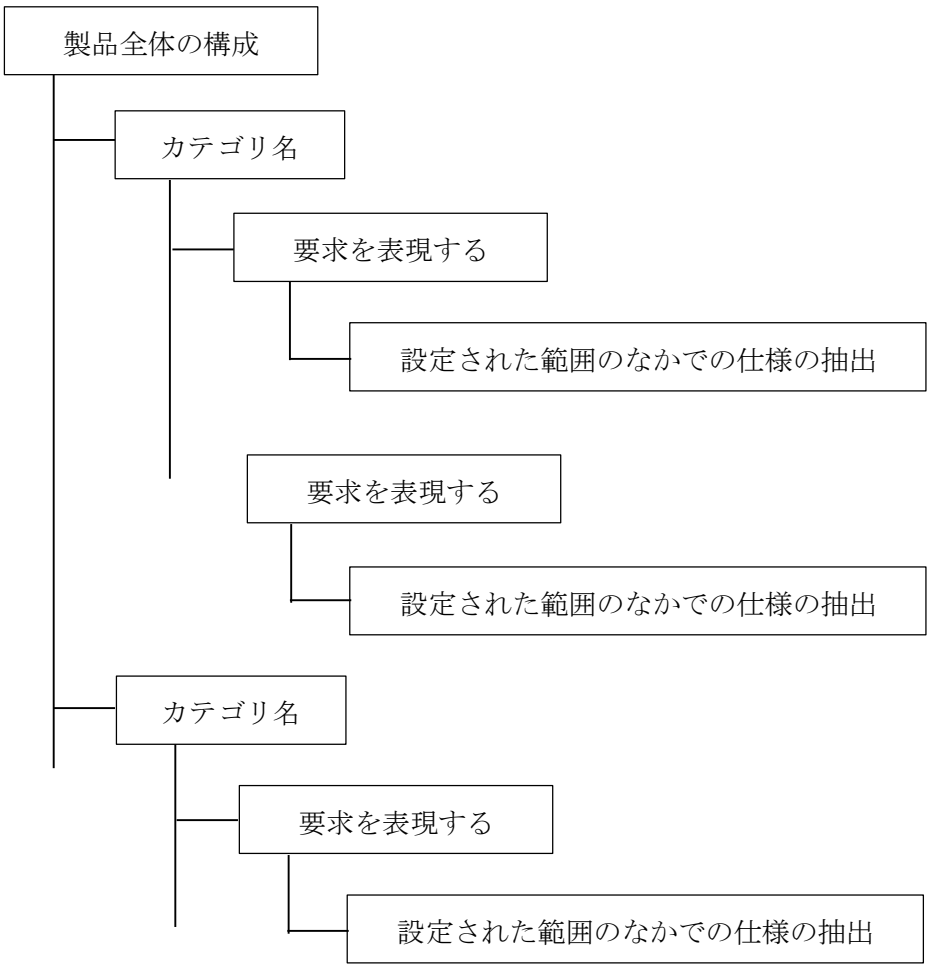
9.11 演習④（要求から仕様を抽出する－2）

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		

9.12 要求のカテゴリ化

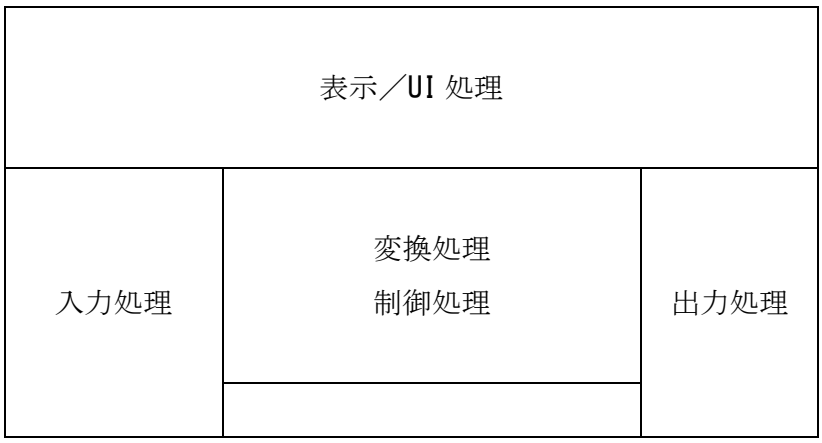
1. 要求仕様書は、それ自体が明確な思想（体系）の下に整理されていることが重要である。要求仕様書全体をどのように構成するかを検討し、要求を階層化して個々の要求の範囲を狭めて仕様を引き出していく。漏れにくい構造というのは漏れていることを発見しやすい構造でもあり、仕様などの追加に対しても、それがどこに収まるべきかがわかりやすい構造ということもできる。



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

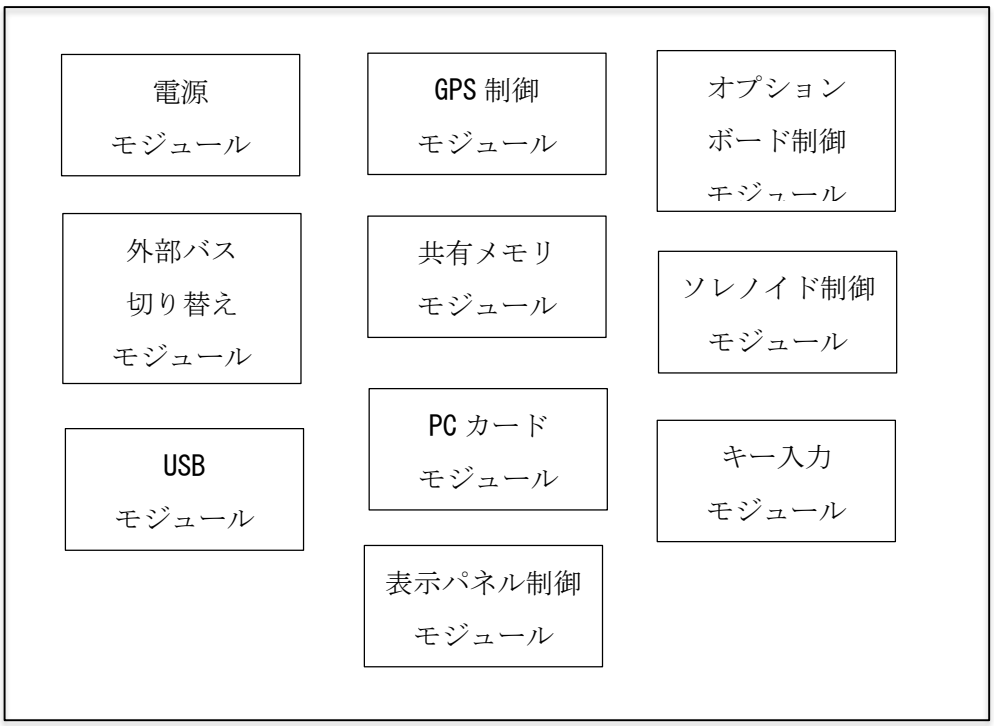
UI 処理、変換処理、保守処理では、次のようにリストアップする。

- 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＞



9.13 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					

9.14 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

9.15 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。

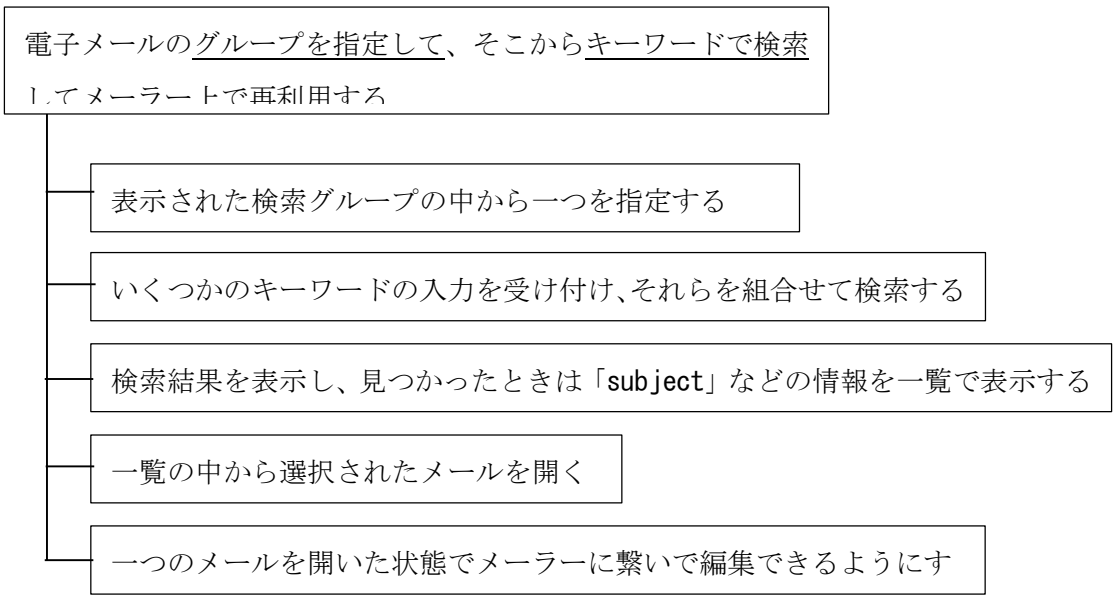
3. これは、構造化設計の「凝集度」の考えに似ており、いくつもの機能を含んだモジュールよりも一つの機能に絞り込む（機能的凝集）ことで、そこで実現すべき処理が明確になるためプログラムの

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)

9.16 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。

9.17 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。

9.18 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。

9.19 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。

9.20 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。

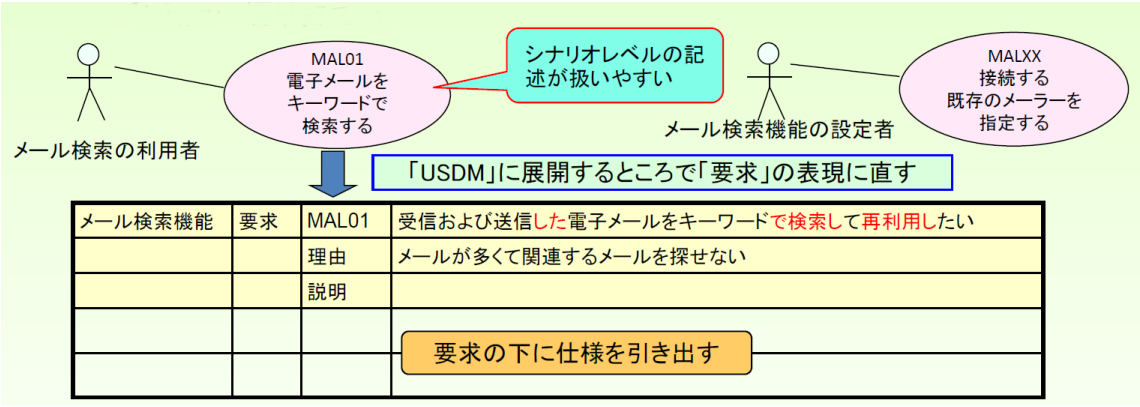
＜図表6－1　上位ユースケースに見立てた例(1)＞

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

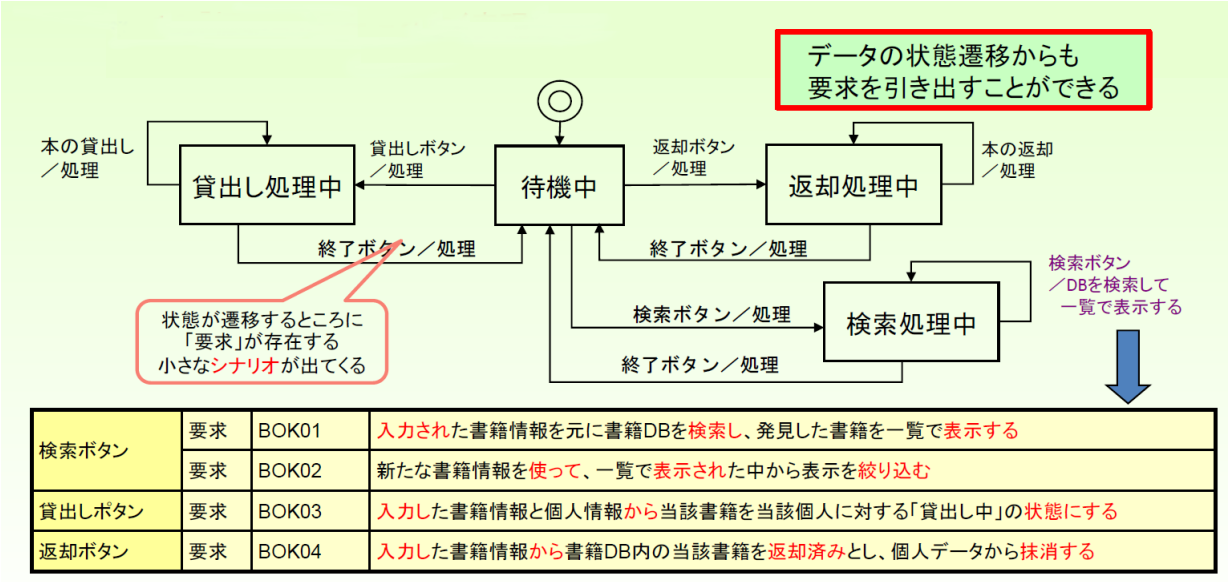
＜図表6－2　上位ユースケースに見立てた例(2)＞

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



9.21 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。

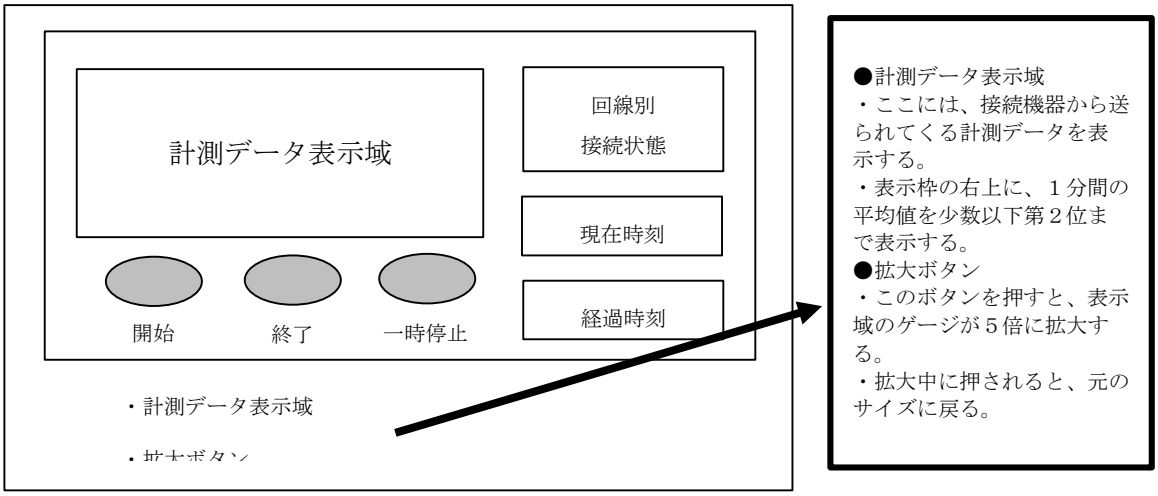


検索ボタン	要求	BOK01	入力された書籍情報を元に書籍DBを検索し、発見した書籍を一覧で表示する
	要求	BOK02	新たな書籍情報を使って、一覧で表示された中から表示を絞り込む
貸出しボタン	要求	BOK03	入力した書籍情報と個人情報から当該書籍を当該個人に対する「貸出し中」の状態にする
返却ボタン	要求	BOK04	入力した書籍情報から書籍DB内の当該書籍を返却済みとし、個人データから抹消する

9.22 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。

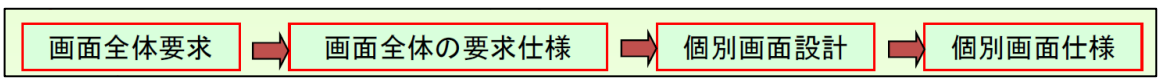
4. そのうえ試作に依存して仕様をまとめると、複数の画面にまたがる機能間で仕様上の矛盾(衝突)を引き起こしやすい。依頼者は一般にユーザーであり、表面的な操作や使い勝手しか見えないため要求できない。実際にその操作によって内部でどのような処理が行われるかを考えられる人でなければ、表示を一時停止しているときに計測データを、通常データと区別しておくことの必要性に気が付きません。

9.23 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)

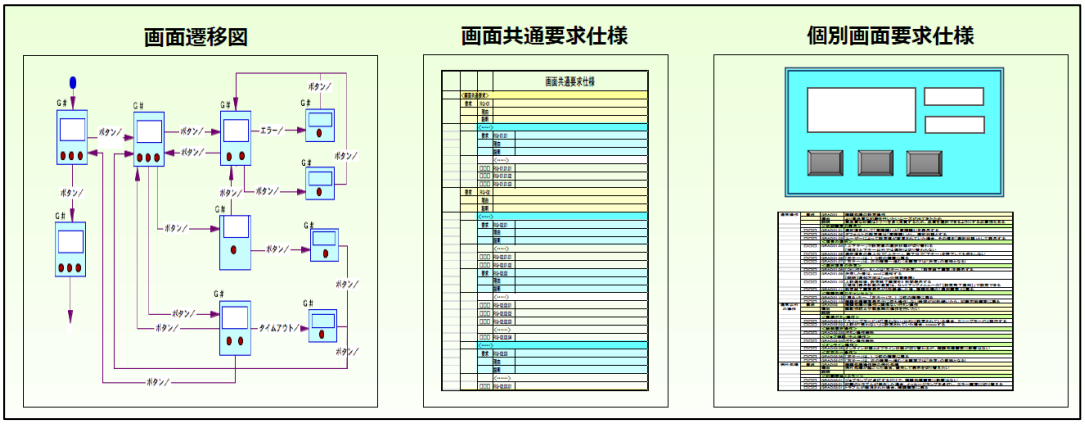


＜画面全体要求の事例＞

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



9.24 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

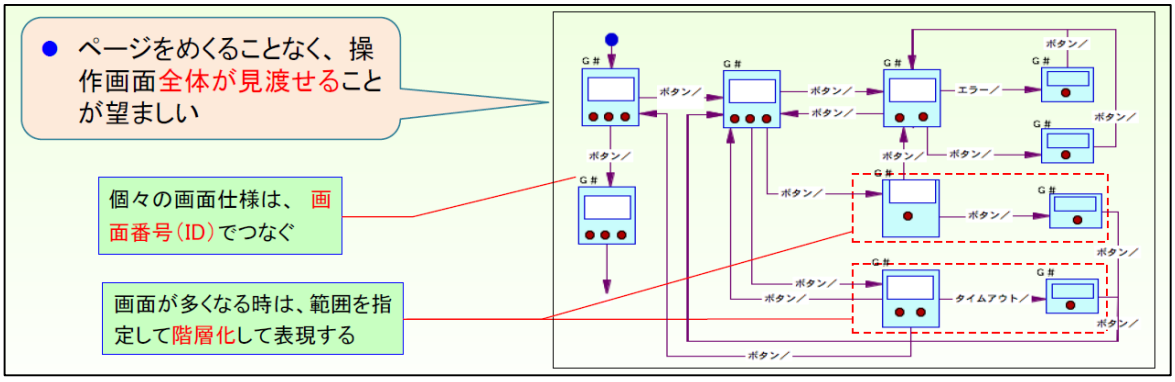
□	仕様	P04.2	ベース画面に直接戻ったときはそれまでの操作は無効になるので、入力操作が行われたときはその場で確認を求めること
要求	OP5	接続されている機器の計測データはすべてリアルタイムで表示し、異常が判断されるときはオペレータに適切な注意を換気すること	
	理由		
□	仕様	OP5.1	“異常”の判断には 3 段階程度の基準を設ける
□	仕様	OP5.2	異常を知らせているときは、画面の更新を停止させる
□	仕様	OP5.3	画面の更新を停止中も計測に影響を与えないこと
要求	OP7	エラー表示に対する操作は、簡単にヘルプで確認できること	
	理由		
□	仕様	OP7.1	「ヘ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

9.25 画面遷移と画面仕様の連携

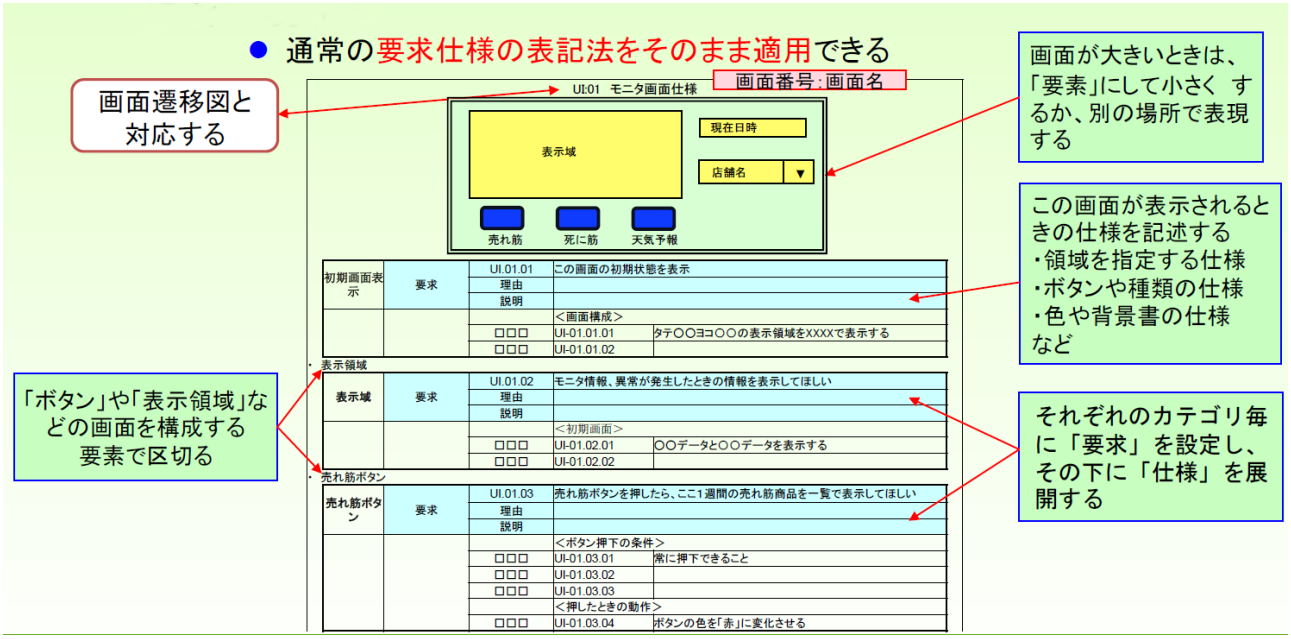
1. 読みやすい文書の一般的な構成は、最初に全体を示し、そのあとに個別のテーマに入る仕掛けが用意されていることである。画面仕様書についても同様で、画面全体の様子を一望できる方法を提供し、その下に個別の画面レイアウトを示し、さらに個別の画面の構成に対する仕様を展開するといった形が望ましい。

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

10付録

10.1 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。

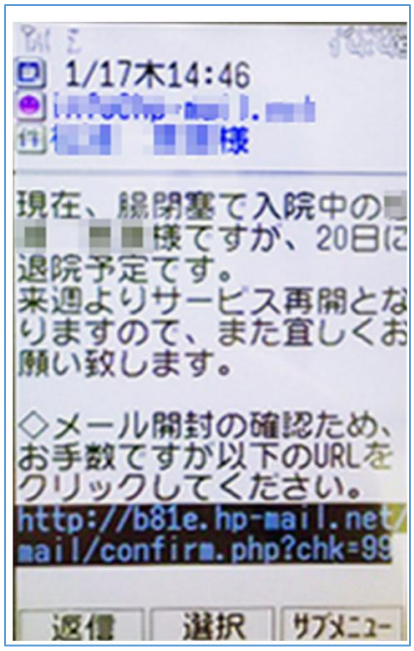
非機能要件

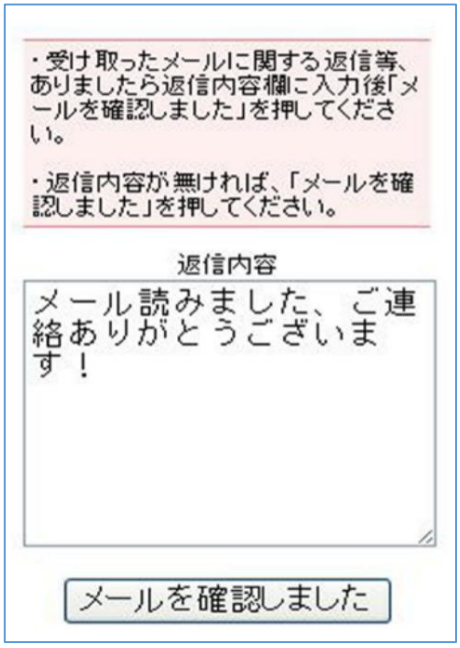
ユニークユーザー数	・介護施設の全職員：200 人(うち 100 人が非正規職員) ・携帯電話やスマートフォンを使いこなせない高齢職員数：50 人 ・アクティブユーザー数：200 人－50 人＝150 人がアプリを常用
被介護者数	・1,000 人 ・介護状況の報告者は、正規職員からおよそ 100 名が担当する。
システムピーク時間	・食事介助の後の報告となる 13 時から 13 時半、17 時半から 18 時の間に報告のピークが発生し、30 分間で 100 件程度。 ・報告を受けた結果の通知のピークが、報告の後の 30 分間で、報告 1 件に対して担当ヘルパー10 人に連絡するため、30 分間で 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。

10.2 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。		

10.3 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)

11アーキテクチャ設計ーはじめに

11.1 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。

11.2 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。

12アーキテクチャ設計の意味

12.1 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

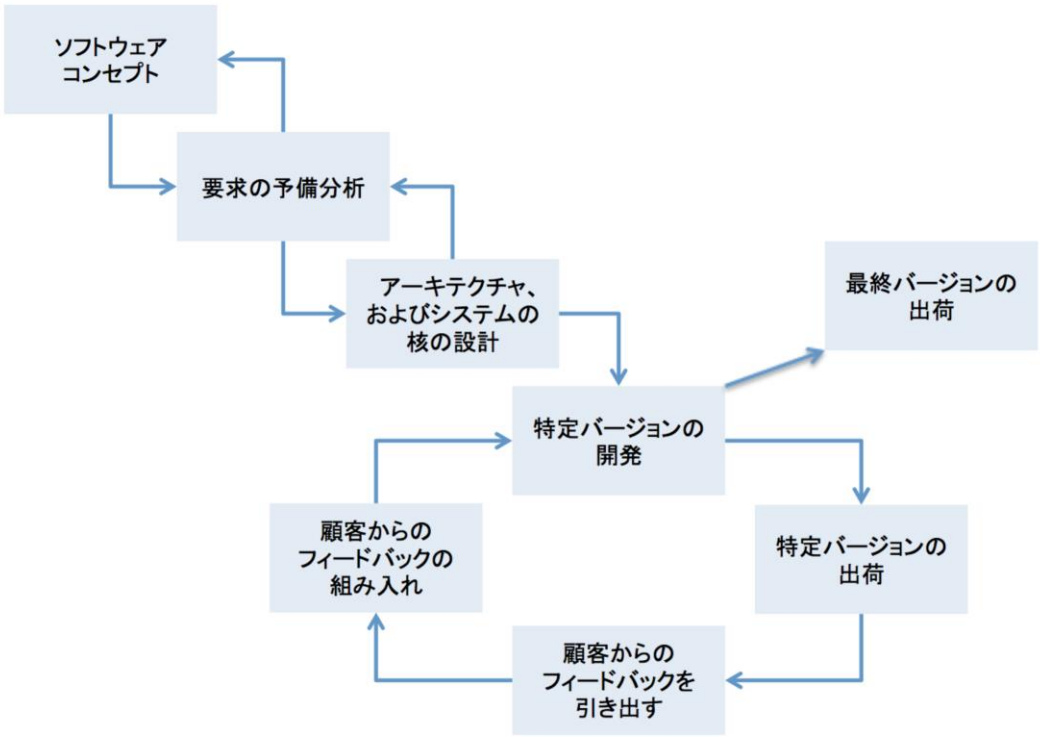
アーキテクチャ設計と聞いて何を思い浮かべるだろうか？ソフトウェア工学の慣例に習い、開発工程の例えとして頻出する建築に例えるなら、アーキテクチャ設計は、建主の要望を満たし、土地、予算、建築基準法など様々な制約条件を加味した上で、住宅の設計を検討し、間取りや、必要な材料を記した設計図として書き起こすまでの工程を言う。

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)

このように明確化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。

■ アーキテクチャ設計の必要性

前述の通り、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)要求をシステム化するのかを規定する作業である。

過去を振り返れば、アーキテクチャ設計が例え上手く行かなくても、開発プロジェクトが頓挫するという事はほとんどなかったように思える。開発の規模も小さく、複雑性も今ほどには高くなかったためである。しかし、昨今アーキテクチャ設計が注目を集める様子からも分かるよう、日に日にその重要性は高まっている。ビジネスにおけるソフトウェアの重要性の高まりや、ハードウェア技術の進歩と共に、ソフトウェアに求められることがらは大きくなり、規模、複雑性、何をとっても以前とは比べ物がならない程に、ソフトウェア開発プロジェクトが難易度を増しているためだろう。

ある特定の要件仕様書を満たすソフトウェアを開発する方法は星の数ほど存在する。どのようなプログラミング言語を選ぶか、フレームワークは何を使用するか、どのようなデザインパターンを使うか、など、考慮すべきことは山程あり、ソフトウェアエンジニアはプロジェクトの特徴や制約などからそれらを1つ1つ決定して行かなければならない。

これら考慮すべきことを決定し、どのようにソフトウェアを開発するのかを定めるのがアーキテクチャ設計である。そして、この時点の決定によって将来出来上がるシステムの品質特性もある程度定まってしまう。耐震設計をせずに建ててしまった建物を耐震補強するのに大変な手間とコストがかかること同様、誤ったアーキテクチャで作られたソフトウェアは、その改修に多くの手間とコストがかかる。また、設計書の要件が誤ったまま建てられる家の工事が上手くいかないよう、誤ったアーキテクチャ設計をしてしまえば、出来上がった成果物は低い品質のものとなるか、場合によっては開発が上手く行かないこともあるだろう。

IoT(Internet of Things)や人工知能など様々な技術が進展する中、更にソフトウェア開発の規模は増大し、複雑性は増し、そのビジネスでの重要度から納期は短くなるだろう。アーキテクチャ設計プロセスは、その成否が、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ことは疑いようもない。

12.2 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[39]、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。

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

39 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。

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　故障モード・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。

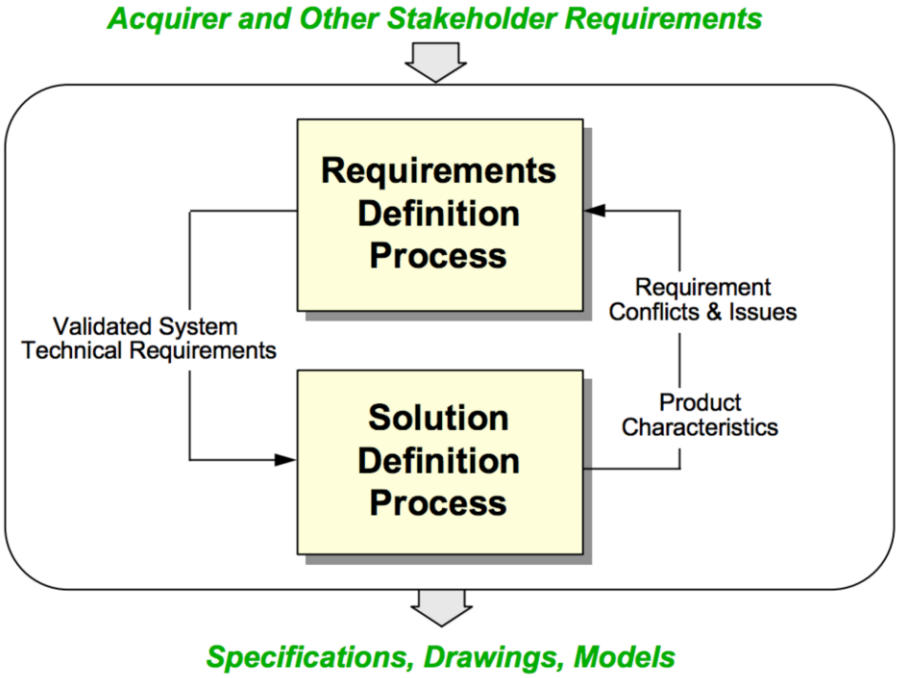
1　検証方法の選定
　1.1　検査、解析、デモンストレーション及びテスト要求の定義
　1.2　検証手順の定義
　1.3　検証環境の確立
2　検証評価の実施
　2.1　アーキテクチャの完全性の検証
　2.2　機能及び性能指標の検証
　2.3　制約を満たしていることの検証
3　差異とコンフリクトの識別
4　検証された設計アーキテクチャの確立
5　ライフサイクル成果物の検証された設計アーキテクチャの確立
6　検証されたシステムアーキテクチャの確立

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。



12.3 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[40]。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[41])

「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

40 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。
41 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。以下のビューに関しても同様。

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[42]をどのチームが開発するかなど、

[42] 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

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[43]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。

- 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。

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。

[43] 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。

- 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。

13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、そして、その更新版でもある ISO/IEC 25010(JIS X 25010:2013)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。

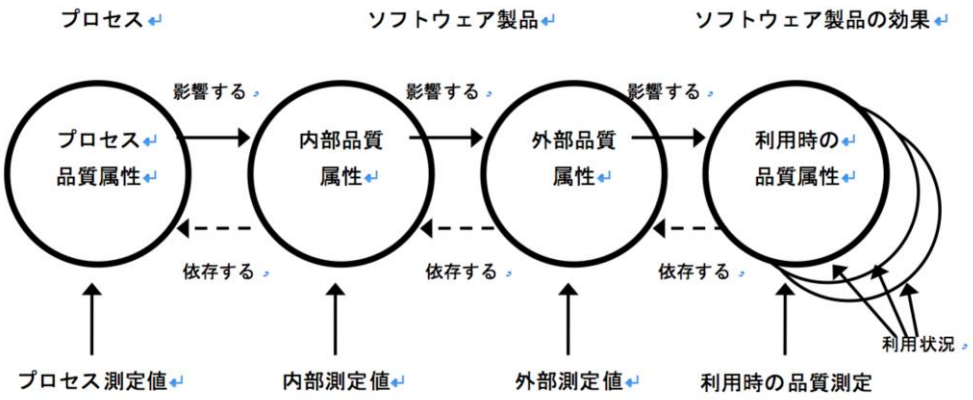
13.1 JIS X 0129-1 品質モデル

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)[44]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。

44 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

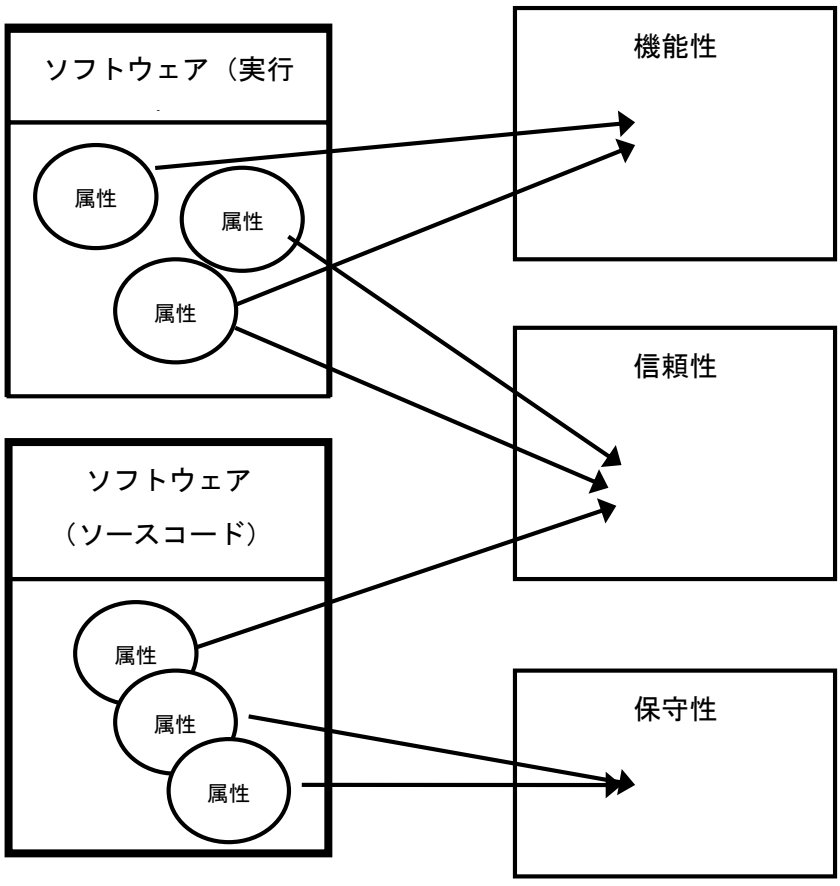
利用者に対する魅力を表す使用性などからなる。

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。

13.1〈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[45]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。)[46]

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[47])、プログラムの〈階層構造化〉(モジュールの階層構造化[48])、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

---

45 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。

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

47 「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。

48 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。

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。

> 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)[49]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[50]。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

---

49 〈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)。

50 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。

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。

13.2 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。

ソースコード

```
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。

13.3 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。

ソースコード

```
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。

13.4 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。

ソースコード

```
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。

13.5 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。

ソースコード

```
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。

13.6 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。

ソースコード

```
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
```

13.1 JIS X 25010:2013 システム及びソフトウェア品質モデル

JIS X 25010:2013 は 2011 年に第一版として発行された ISO/IEC 25010 を基に、技術的内容及び構成を変更することなく作成した日本工業規格である。上述の JIS X0129-1:2003 を改正し、同じソフトウェア品質特性に幾つかの追補をしたものである。

JIS X0129-1:2003 からの差分は以下に挙げる項目となる。(システム及びソフトウェア製品の品質要求及び評価(SQuaRE)システム及びソフトウェア品質モデルより引用)

- 品質モデルの適用範囲は,コンピュータシステム及びシステム視点からの利用時の品質を含むために拡張されている。
- 利用状況網羅性は，利用時の品質の特性として追加され，副特性として，利用状況完全性及び柔軟性がある。
- セキュリティは，**JIS X 0129-1** の機能性の副特性という扱いを改め，新たに品質特性として追加した。副特性として，機密性，インテグリティ，否認防止性，責任追跡性及び真正性がある。
- （相互運用性及び共存性を含め）互換性は，特性として追加されている。
- 次の副特性が追加されている。機能完全性，容量・能力，ユーザエラー防止，アクセシビリティ，可用性，モジュール性及び再利用性。
- 法律及び規制に対する適合性は，厳密な意味での品質の一部というよりはむしろ，システム要求事項全体の一部であるので，適合性の副特性は削除されている。
- 内部品質モデル及び外部品質モデルは，結合されて，製品品質モデルとなっている。
- 適切な場合に，ソフトウェア独自の定義を使用するよりはむしろ，一般的な定義が採用されている。
- 幾つかの特性及び副特性は，更に正確な名称が付いている。

この規格では主に、次のことが規定されている。

a) 利用時の品質モデル：このモデルは、特定の使用状況において製品が使用されるときに、対話の成果に関する５つの特性からなる。

b) 製品品質モデル：このモデルは、ソフトウェアの静的特徴及びコンピュータシステムの動的特徴に関係する、８つの特性からなる。

1　利用時の品質モデル

1.1　有効性

明示された目標を利用者が達成する上での正確さ及び完全さの度合い。

1.2　効率性

利用者が特定の目標を達成するための正確さ及び完全さに関連して、使用した資源の度合い。

1.3　満足性

製品又はシステムが明示された利用状況において使用されるとき、利用者ニーズが満足される度合い。

1.3.1 実用性

利用の結果及び利用の影響を含め、利用者が把握した目標の達成状況によって得られる利用者の満足の度合い

1.3.2 信用性

利用者又は外の利害関係者がもつ、製品又はシステムが意図したとおりに動作するという確信の度合い。

1.3.3 快感性

個人的なニーズを満たすことから利用者が感じる喜びの度合い。

1.3.4 快適性

利用者が(システム又はソフトウェアを利用する時の)快適さに満足する度合い。

1.4　リスク回避性

製品又はシステムが、経済状況、人間の生活又は環境に対する潜在的なリスクを緩和する度合い。

1.4.1 経済リスク緩和性

意図した利用状況において、財務状況、効率的運用操作、商業資産、評判又は外の資源に対する潜在的なリスクを、製品又はシステムが緩和する度合い。

1.4.2 健康・安全リスク緩和性

意図した利用状況において、製品又はシステムが人々に対する潜在的なリスクを緩和する度合い。

1.4.3 環境リスク緩和性

意図した利用状況において、環境に対する潜在的なリスクを製品又はシステムが軽減する度合い。

1.5　利用状況網羅性

明示された利用状況及び当初明確に識別されていた状況を超越した状況の両方の状況において，有効性，効率性，リスク回避性及び満足性を伴って製品又はシステムが使用できる度合い

1.5.1 利用状況完全性

明示された全ての利用状況において、有効性、効率性、リスク回避性及び満足性を伴って製品又はシステムが使用出来る度合い。

1.5.2 柔軟性

要求事項の中で初めに明示された状況を逸脱した状況において、有効性、効率性、リスク回避性及び満足性を伴って製品又はシステムが使用できる度合い。

2 製品品質モデル

2.1 機能適合性

明示された状況下で使用するとき、明示的ニーズ及び暗黙のニーズを満足させる機能を、製品又はシステムが提供する度合い。

2.1.1 機能完全性

機能の集合が明示された作業及び利用者の目的の全てを網羅する度合い。

2.1.2 機能正確性

正確さの必要な程度での正しい結果を、製品又はシステムが提供する度合い。

2.1.3 機能適切性

明示された作業及び目的の達成を、機能が促進する度合い。

2.2 性能効率性

明記された状態（条件）で使用する資源の量に関係する性能の度合い。

2.2.1 時間効率性

製品又はシステムの機能を実行する時、製品又はシステムの応答時間及び処理時間、並びにスループット速度が要求事項を満足する度合い。

2.2.2 資源効率性

製品又はシステムの機能を実行するとき、製品又はシステムで使用される資源の量及び種類が要求事項を満足する度合い。

2.2.3 容量満足性

製品又はシステムのパラメータの最大限度が要求事項を満足させる度合い。

2.3 互換性

同じハードウェア環境又はソフトウェア環境を共有する間、製品、システム又は構成要素が他の製品、システム又は構成要素のじょう方を交換することができる度合い、及び/又はその要求された機能を実行することができる度合い。

2.3.1 共存性

その他の製品に有害な影響を与えずに、他の製品と共通の環境及び資源を共有する間、製品が要求された機能を効率的に実行することができる度合い。

2.3.2 相互運用性

二つ以上のシステム、製品又は構成要素が情報を交換し、既に交換された情報を使用することができる度合い。

2.4 使用性

明示された利用状況において、有効性、効率性及び満足性をもって明示された目標を達成するために明示された利用者が製品又はシステムを利用することができる度合い。

2.4.1 適切度認識性

製品又はシステムが利用者のニーズに適切であるかどうかを利用者が認識できる度合い。

2.4.2 習得性

明示された利用状況において、有効性、効率性、リスク回避性及び満足性をもって製品又はシステムを使用するために明示された学習目標を達成するために、明示された利用者が製品又はシステムを利用できる度合い。

2.4.3 運用操作性

製品又はシステムが、それらを運用操作しやすく、制御しやすくする属性をもっている度合い。

2.4.4 ユーザエラー防止性

利用者が間違いを起すことをシステムが防止する度合い。

2.4.5 ユーザインタフェース快美性

ユーザインタフェースが、利用者にとって楽しく、満足の行く対話を可能にする度合い。

2.4.6 アクセシビリティ

製品又はシステムが、明示された利用状況において、明示された目標を達成するために、幅広い範囲の心身特性及び能力の人々によって使用できる度合い。

2.5 信頼性

明示された時間帯で、明示された条件下に、システム、製品又は構成要素が明示された

機能を実行する度合い。

2.5.1 成熟性

通常の運用操作の下で、システム、製品又は構成要素が信頼性に対するニーズに合致している度合い。

2.5.2 可用性

使用することを要求されたとき、システム、製品又は構成要素が運用操作可能及びアクセス可能な度合い。

2.5.3 障害許容性

ハードウェア又はソフトウェア障害にもかかわらず、システム、製品又は構成要素が意図したように運用操作できる度合い。

2.5.4 回復性

中断時又は故障時に、製品又はシステムが直接的に影響を受けたデータを回復し、システムを希望する状態に復元することができる度合い。

2.6 セキュリティ

人間又は他の製品もしくはシステムが、認められた権限の種類及び水準に応じたデータアクセスの度合いをもてるように、製品又はシステムが情報及びデータを保護する度合い。

2.6.1 機密性

製品又はシステムが、アクセスすることを認められたデータだけにアクセスすることができることを確実にする度合い。

2.6.2 インテグリティ

コンピュータプログラム又はデータに権限をもたないでアクセスすること又は修正することをシステム、製品又は構成要素が防止する度合い。

2.6.3 否認防止性

事象又は行為が後になって否認されることがないように、行為又は事象が引き起こされたことを証明することができる度合い。

2.6.4 責任追跡性

実体の行為がその実体に一意的に追跡可能である度合い。

2.6.5 真正性

ある主体又は資源の同一性が主張したとおりであることを証明できる度合い。

2.7 保守性

意図した保守者によって、製品又はシステムが修正することができる有効性及び効率性

の度合い。

2.7.1 モジュール性

一つの構成要素に対する変更が他の構成要素に与える影響が最小になるように、システム又はコンピュータプログラムが別々の構成要素から構成されている度合い。

2.7.2 再利用性

一つ以上のシステムに、又は他の資産づくりに、資産を使用することができる度合い。

2.7.3 解析性

製品もしくはシステムの一つ以上の部分への意図した変更が製品もしくはシステムに与える影響を総合評価すること、欠陥もしくは故障の原因を診断すること、又は修正しなければならない部分を識別することが可能であることについての有効性及び効率性の度合い。

2.7.4 修正性

欠陥の取り込みも既存の製品品質の低下もなく、有効的に、かつ、効率的に製品又はシステムを修正することができる度合い。

2.7.5 試験性

システム、製品又は構成要素について試験基準を確立することができ、その基準が満たされているかどうかを決定するために試験を実行することができる有効性及び効率性の度合い。

2.8 移植性

一つのハードウェア、ソフトウェア又は他の運用環境もしくは利用環境からその他の環境に、システム、製品又は構成要素を移すことができる有効性及び効率性の度合い。

2.8.1 適応性

異なる又は進化していくハードウェア、ソフトウェア又は他の運用環境もしくは利用環境に、製品又はシステムが適応できる有効性及び効率性の度合い。

2.8.2 設置性

明示された環境において、製品又はシステムをうまく設置及び/又は削除できる有効性及び効率性の度合い。

2.8.3 置換性

同じ環境において、製品が同じ目的の別の明示された製品と置き換えることができる度合い。

13.1 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[51]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

[51] 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。

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。

- 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数、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

13.2 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。
後の章で説明する品質特性駆動設計においては、品質特性一般シナリオをベースに、プロジェクトにおける具体的なケースに合わせ品質特性具象シナリオをする事で、品質要件を表す事となる。品質要件を表現する方法は様々存在するが、後半を読み進めるに辺り重要な要素となるため、品質特性シナリオの概念はしっかりと抑えたい。

■ シナリオとは

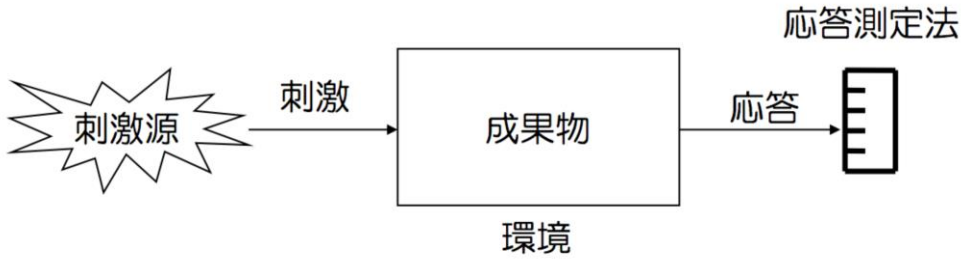
シナリオとは、要求を具体的なストーリーとして記述したもので、利害関係者の意図/興味の表現や、機能要件/品質要件の表現など、様々な用途で使われる。考慮されるべき対象を元に利用シナリオ（通常の利用方法を対象とするシナリオ）、成長シナリオ（将来起こりえる成長の予測を対象とするシナリオ）、調査シナリオ（問題の予測を対象とするシナリオ）などに分類される他、別の切り口として、システムに非依存でどんなシステムにも適用出来る一般シナリオと、検討中の特定システムに固有の具象シナリオに分類される。

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（○○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。

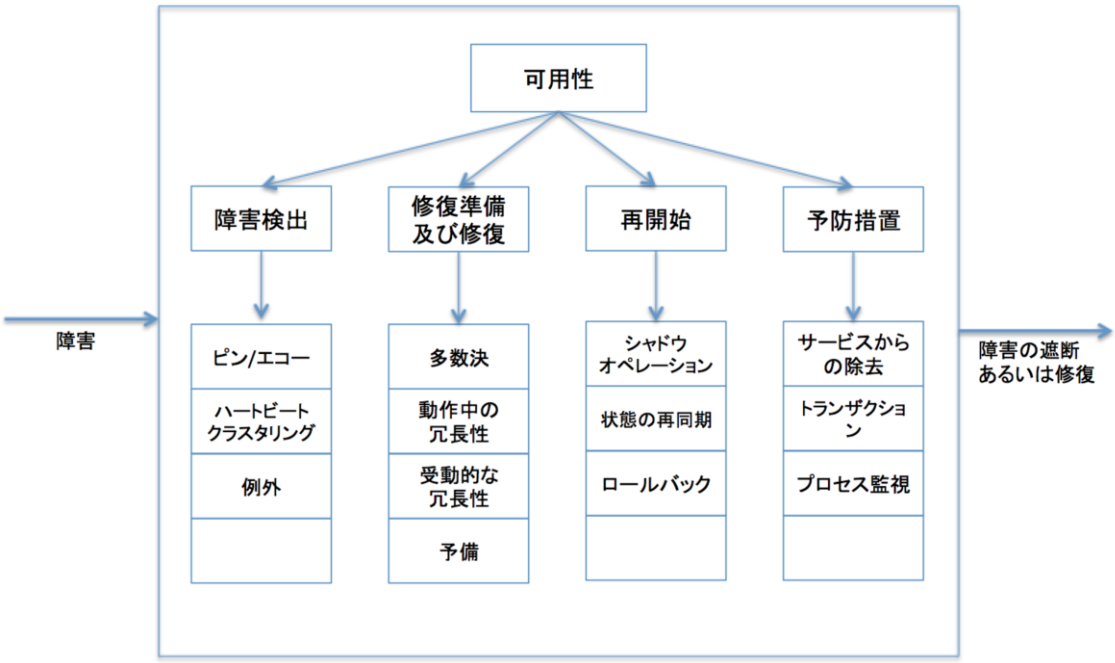
14品質特性の実現手法

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。

14.1 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。

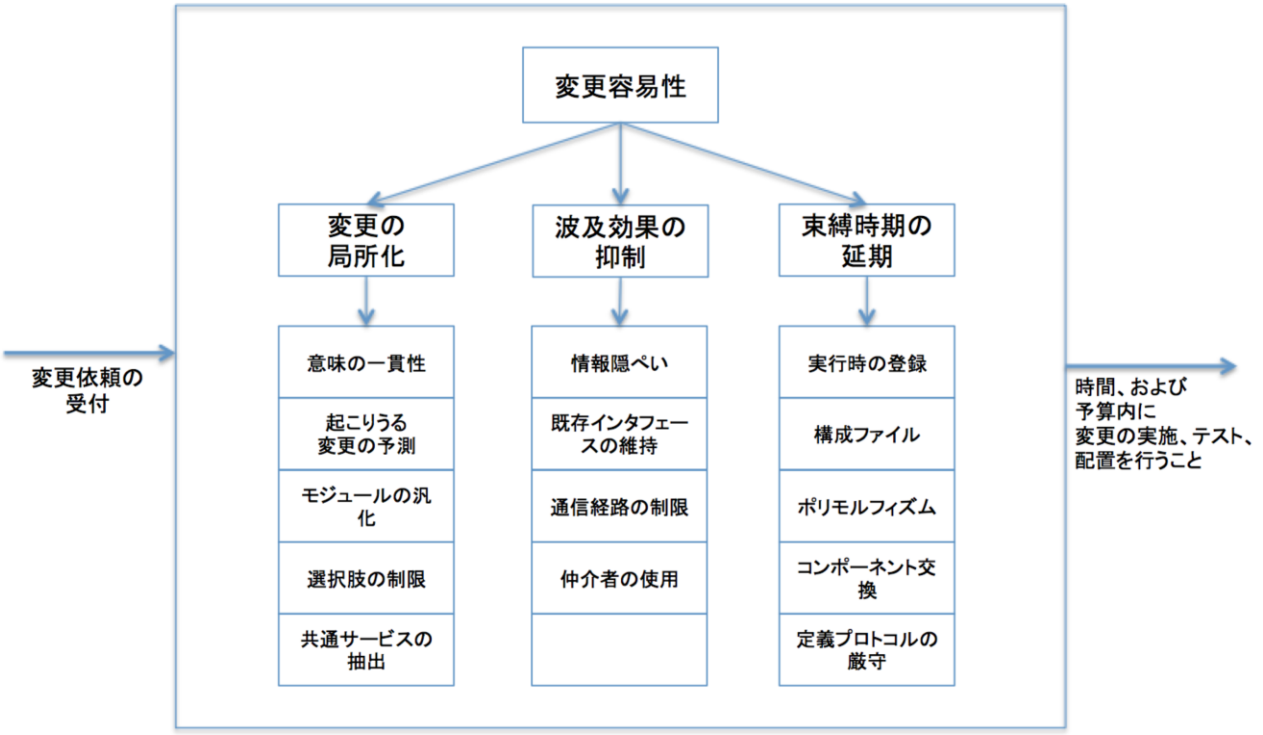
✧　プロセス監視

「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。

14.2 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[52]

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。

[52]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。

```
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;
}
```

- 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。

14.3 性能の実現手法

■ 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

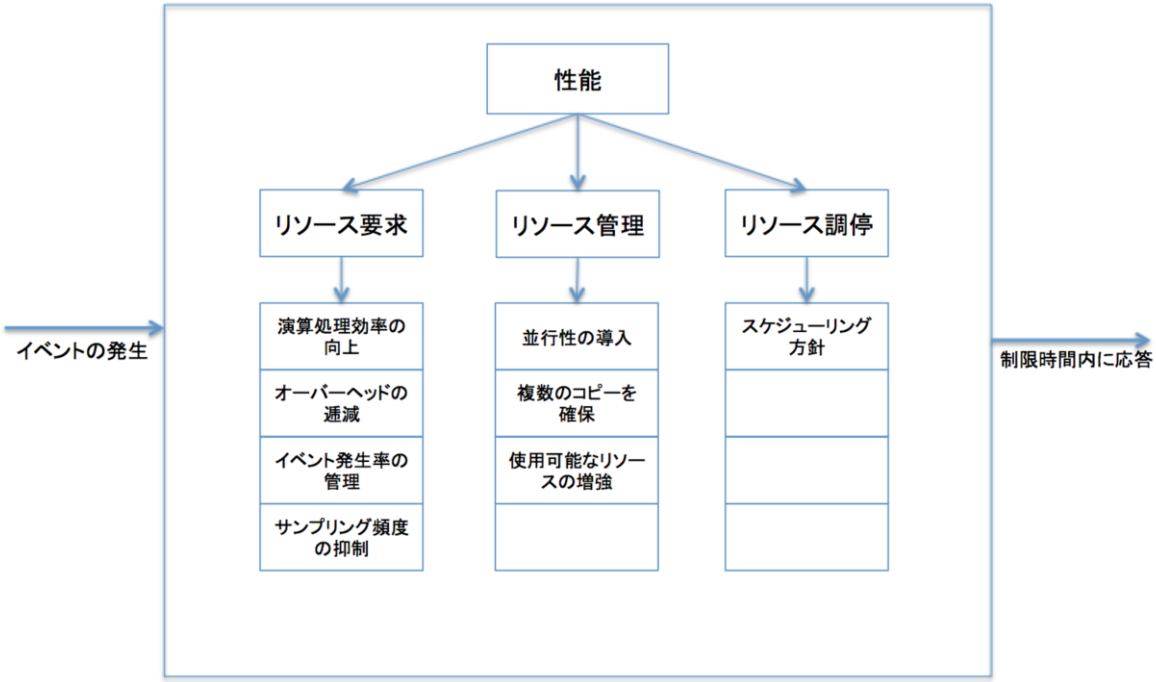
イベントが到着すると、システムはリソース（CPU、データ記憶装置、ネットワーク通信帯域、ま

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

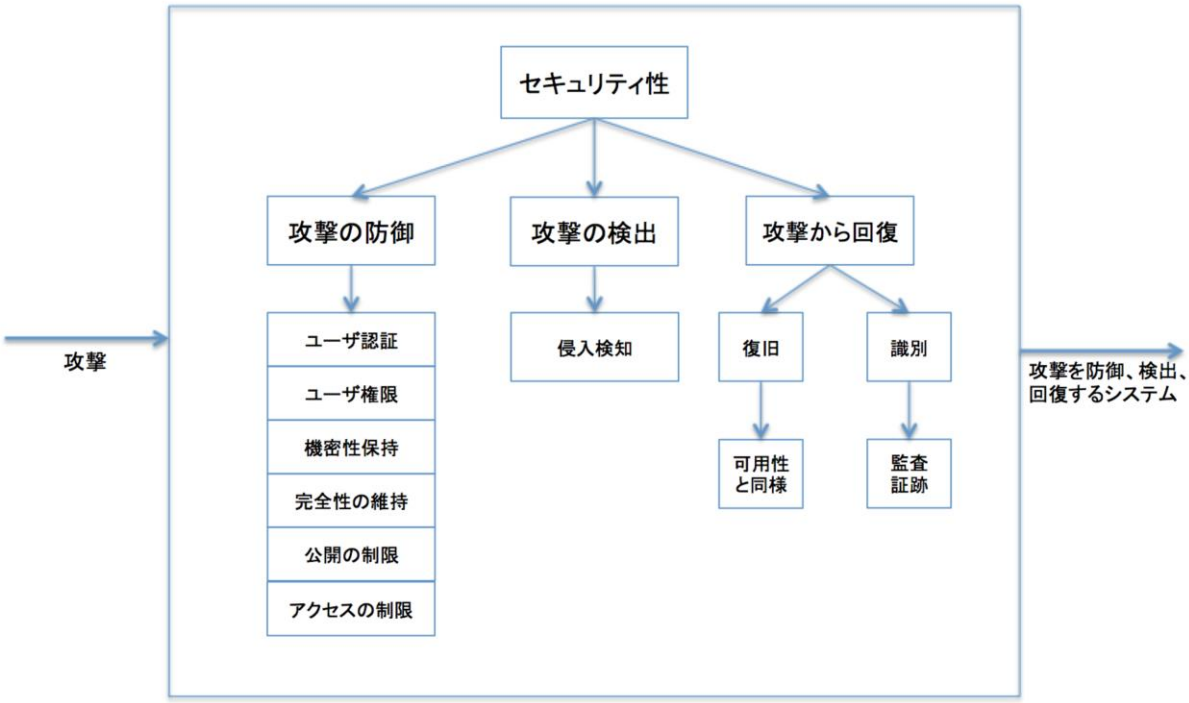
ラウンドロビンは、要求をある規則に基づいて順序付けして並べ、リソースをある決まった時間単位ずつ、全ての要求に、順番に割り当てて行くスケジューリング方針。優先順位が高く、処理にも時間がかかる要求が前にあると、多くのスケジューリング方針で、優先順位が低いが処理時間は短い要求が待たされ、イベントのキューが溜まって行ってしまう事が起こる。ラウンドロビンでは、優先度が低い要求にも、ある程度早期の段階で決まった時間単位のリソースが割り振られる為、キューが 1 つの大きな処理によって完全に詰まってしまう事を防ぐ事が出来る。一方で、本当に優先度が高く、応答時間のデッドラインも短い要求があった際などに、その要求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。

14.4 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。

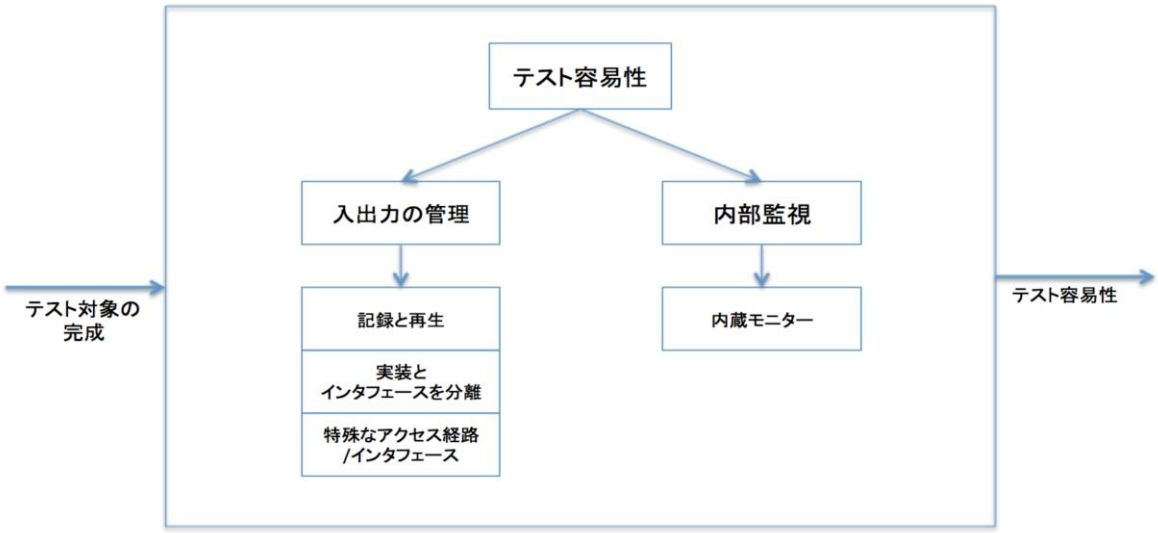
- 攻撃からの回復

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。

14.5 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

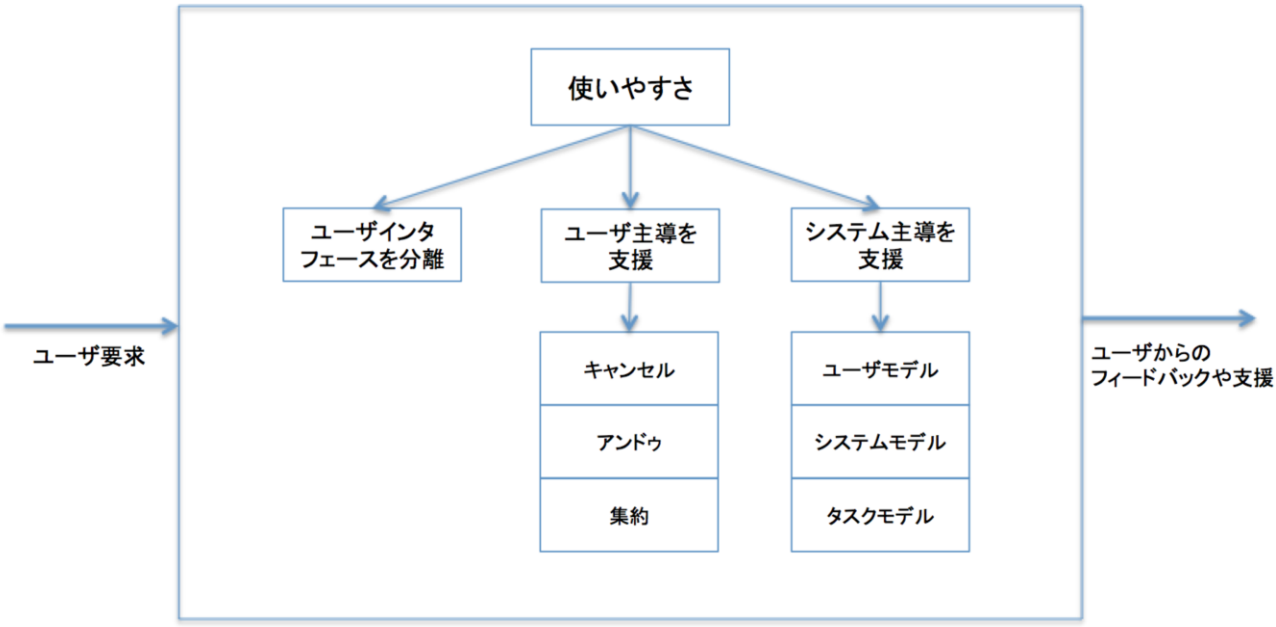
部で何が起きているかをしる為に必要な情報は、システムが通常稼働時に返すべき値と異なる場合が多く、内部の状態を知る事は、システム全体への入力と出力の比較のみでは発見出来なかった欠陥が発見される事や、その発見された欠陥を修復する事を助ける事に繋がる。

- 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。

14.6 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

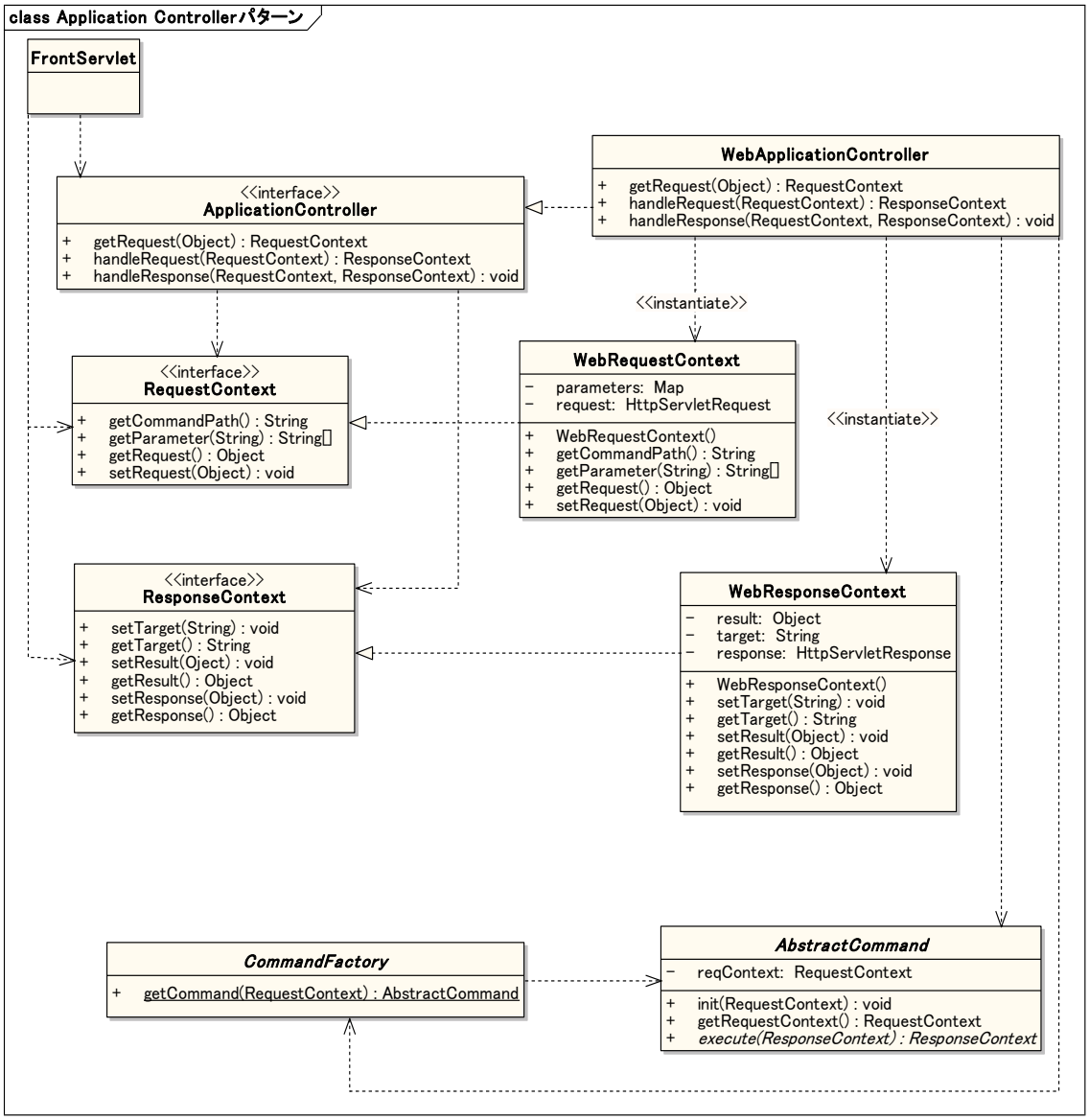
ユーザーの使用性を高める上で必要な機能の中で、ユーザーの某かの行動や、イベントの発生などに対し、システムが主導で某かの判断を行い、自動的にユーザー支援を行うような機能を表す。具体例としては、英文入力システムにおいて、小文字で文が始められた際に、自動的に文の最初の文字を大文字に変換する機能などが挙げられる。システム主導で実行される機能は、ユーザーや、ユーザーが着手しているタスク、またシステムの状態に関する情報（モデル）に基づき、その機能を実行するかどうかが決定される。「ユーザーモデル」に基づき実行される機能の例としては、ユーザーないしユーザーグループがシステムを使用する際の振る舞いなどに基づき、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。

15UML ダイアグラム

15.1 UML（Unified Modeling Language）と開発方法

一定のソフトウェアライフサイクルモデルに対して、ソフトウェアライフサイクルプロセスの"開発プロセス"(一定の開発技法に基づいたアクティビティとタスクを含む)を当てはめたものが実際の開発方法です。開発方法の過程に含まれる分析や設計といったアクティビティの内部で作成・参照される図に関して、その表記法を「ノテーション」(notation)といいます。以下では、UML のノテーションについて解説します。

■UML のサンプル



15.2 UML のダイアグラムの種類

UML(Unified Modeling Language)のダイアグラム(diagram)は、「構造図」(structure diagram)と「振る舞い図」(behavior diagram)に大別されます。

structure diagram（構造図）		
	class diagram	クラスの静的構造を描くための図です。 クラスおよびインターフェイスを作成することができます。
	object diagram	インスタンスを表現するための図です。クラス図とは独立しているが、記法はクラス図と同じです。
	package diagram	これもクラス図と同じ記法です。パッケージの構造を記すために用います。
	composite structure diagram	UML2.0 で追加された図です。 構造を持つ Classifier(クラスを含む)の内部構造を静的に表現し、この Classifier を構成している各部品の協調関係を描くことができます。
	component diagram	コンポーネントの外部に対する仕様と内部構造を描くために用います。
	deployment diagram	システム内で、各部品をどのように配置するかを意味する図です。
behavior diagram（振る舞い図）		
	interaction diagram（相互作用図）	
	sequence diagram	インスタンス間の相互作用を時間軸に沿って表現するための図です。 interaction 図の中で最も一般的な図です。 UML 2.0 では、記法が強化され、図のネストなどが可能になりました。
	communication diagram	インスタンス相互の関係を軸に、インスタンス間の相互作用を表現します。
	interaction overview diagram	UML2.0 で追加された図です。 システムの相互作用の流れを全体的に表現するための複合的な図です。
	timing diagram	UML2.0 で追加された図です。 システムの動作を、時間軸に沿って記述することができます。

	state machine diagram	UML2.0 で statechart 図を改称しました。 インスタンスを状態機械とみなして、その状態遷移を記述することができます。
	activity diagram	statechart 図から独立させた図です。
	use case diagram	システムのユースケースを描くための図です。システムの要求定義や分析に使用します。

「構造図」では、クラスの構造（フィールド、メソッド）を始めとするシステムの静的な構造（プログラムの状態）を表現するのに対し、「振る舞い図」では、システムの動作の仕方（実行時の挙動）を表現します。

15.3 UML に対する 3つの考え方

UMLをどのように利用するかについては、Martin Fowlerが彼の著作の中で、以下の三つの考え方を提示しています[53]。

■スケッチとしての UML(UML as sketch)

開発者間のコミュニケーションを目的として UML を利用するという考え方です。お互いに自分のソフトウェア設計のアイデアを伝達することが目的なので、UML の仕様に厳密に準拠することはあまり重視しません。また、UML によってプログラム設計の全体を必ず表現しなければならないということはありません。相手とのコミュニケーションに必要な部分だけ UML の図で表現します。特にアジャイルソフトウェア開発系の開発プロセスで用いられるUMLの使い方です。スケッチとしてのUMLの利用においては、会議中にホワイトボードなどを利用してUMLの図を描き、必要事項の伝達に利用されます。

■設計図としての UML(UML as blueprint)

設計者によって完璧な設計を作成・文書化し、それに従って、プログラマーが忠実にコーディングを行うという考え方です。システム全体に関する詳細かつ完全なUMLを描くために、UMLのモデリングツールを必ず必要とします。

■プログラム言語としての UML(UML as programming language)

UML をプログラム言語として利用するという考え方です。UML で表現された図から、変換ツールを利用することで実行可能なコードを生成します。「MDA」の考え方は、この立場からUMLを捉える一例です。

53 Fowler, Martin, 2004, UML Distilled Third Edition, Pearson Education（羽生田 栄一監訳, 2005, 『UML モデリングのエッセンス 第 3 版』, 翔泳社, , p.2-3）

16概念モデルとしてのクラス図

16.1 伝統的論理学における概念論

古代ギリシャの哲学者であるアリストテレス[54]に始まる伝統的論理学においては、概念(concept)は、思考 ― ここでは、判断(judgment)と推理(inference) ― の要素として考えられています。どのような思考(判断と推理)も、一定の概念に関して行われるため、概念が思考の最も基本的な要素であるということになります。たとえば、「人間は動物である」という判断において、「人間」は概念であり、この概念に関して、「人間は動物である」という判断が下されています。このように概念は、判断(そして推理も)が成立するための要素となります。

ここで、概念について最も重要なことは、それが一般的な表象(presentation:意識の対象)であるということです。「人間は動物である」という判断に含まれる「人間」という概念は、特定の個物としての「人間」を意味しません。「すべての人間」を意味しています。このように、概念は一般的な表象であり、これによって、事物を包括的に捉えることが可能になります。

ここでさらに、概念と、特定の個物に関する表象の違いを考えてみます。個物に関する表象は、たとえば、特定の人間に関する表象には、その人間にのみ備わる特殊的性質(一部の複数の人間にのみ共通であるという特殊的性質も考えられる)が含まれますが、他方で、そのような特殊性を超えたすべての人間に共通な一般的性質を見出すこともできます。そこで、特定の人間に関する表象から、特殊的性質を捨象する(取り除く)ことによって、すべての人間に共通な一般的性質のみを抽象する(取り出す)ことができます。このとき、捨象された特殊的性質を偶有的属性(accidental attribute)、抽象された一般的性質を本質的属性(essential attribute)といいます。このように、特定の人間についての表象から偶有的属性を取り除き、すべての人間に共通な一般的性質、つまり、本質的属性を取り出すことによって、「人間」という概念を得ることができます。概念には本質的属性のみが属します。

さらに概念には、本質的属性という内容の他に、さらに、その概念の本質的属性を備えた個物の範囲、つまり、概念の適用範囲を考えることもできます。前者の本質的属性を、概念の内包(intension)、後者の適用範囲を外延(extension)といいます。内包と外延には、内包が増加すれば外延が縮小し、内包が減少すれば外延は拡大するという関係があります。内包を減少させて外延を拡大することを概括(generalization)、内包を増加させて外延を縮小することを限定(determination)といいます。たと

54　アリストテレス（紀元前 384 年－紀元前 322 年）は、古代ギリシャの哲学者である。古代ギリシャ語で Ἀριστοτέλης、、英語で Aristotle と表記する。アリストテレスによる論理学、自然学、形而上学といった成果は、その後の哲学、物理学などの諸学問の展開に大きな影響を与えている。

えば、「職業」という概念には、「賃金をえるために恒常的に行う行為の集合」という本質的属性があるとした場合、これに加えてさらに「人に物事を教える」という本質的属性を追加することにより、「教職」という概念を得ることができます。このとき、「職業」概念から「教職」概念への移行には、内包の増加があり、逆に、「教職」概念から「職業」概念への移行には、内包の減少があります。しかし、概念の適用範囲、つまり、外延については、「職業」よりも「教職」のほうが縮小されています。このとき、内包が少なく外延が広い概念を上位概念（superordinate concept）、内包が多く外延が狭い概念を下位概念（subordinate concept）といいます。「職業」は「教職」の上位概念であり、「教職」は「職業」の下位概念です。

16.2 様々なクラス論

■羽生田栄一のクラス論[55]

羽生田のクラス論の特徴は、クラスを、同時に、概念、型、モジュールを意味する（もしくは表現する）としている点にあります。

> ・・・オブジェクト指向の基本原理は、ひとことでいってしまえば、「クラス＝概念＝型＝モジュール」という等式が成り立つことに尽きます。クラスが属性（インスタンス変数）と操作（メソッド）をカプセル化して定義する単位であり、かつモデル化対象に存在する概念を表現する際の単位であり、プログラミングの際に自由に導入できるユーザー定義可能なデータ型であり、それがオブジェクト指向でソフトウェアを作っていく際の基本的な管理対象としてのモジュールの単位になる、ということです。（脚注の URL 参照）

この基本原理に立脚して、羽生田は、オブジェクト指向のメリットについて次のように述べています。

> この等式が成り立つとしたおかげでオブジェクト指向では、複雑な問題領域の構造や振る舞いを自由にモデリングしプログラミングにつなげていくことができるようになりました。モデリング上必要とあればいつでも新しいクラスを導入して概念を定義したり、設計や実装上必要とあれば自由にクラスを使って新たなデータ構造をユーザー定義し導入できるわけです。そして、ここが重要なポイントですが、うまくクラスを定義すればそれが概念としてもデータ型としても取り扱うことができる、すなわちソフトウェア上の管理対象としてトレース可能になる、というのがオブジェクト指向の偉大な発明なのです。クラスは概念でもありデータ型でもあるということがどれだけソフトウェア

55 http://www.atmarkit.co.jp/farc/rensai/uml_b_l02/uml_b_l02a.html 参照

設計にとって柔軟性をもたらしているか考えてみてください。（同上）

概念の内包と外延とクラスとの関係については、次のように論じています。

ここでクラスを概念として理解する際の注意をひとこと。クラスは概念には対応しても集合には必ずしも対応しないので注意が必要です。通常、数学や論理学で使われる概念には内包的定義と外延的集合の両方が対応します。内包的定義とは、その概念の満たすべき条件を命題（真偽の定まる文）のセットとして表わしたものです。（同上）

つまり、文を用いてその概念の意味を定義するわけです。それに対し外延的集合とは、その概念で指示される具体的な実例すべてのことです。こちらは、意味は考えずにあくまでもそこに含まれる具体的なオブジェクトの集合によって実体的に概念を指定しようというものです。
実はオブジェクト指向では、クラスを用いて対応する概念の内包的定義を表わします。原理的には同様にその概念に対応する外延的集合を想定することができますが、クラスそのものはそこに定義してある属性のセットと操作のセットを満たすオブジェクトを生成しなさい、という宣言に過ぎません。したがって、クラス定義自体は対応する外延的集合と同一視することはできません。（同上）

では、オブジェクト指向で外延的集合を表わすにはどうするかというと、明示的にコレクションオブジェクトを用意し、そこにそのクラスのインスタンスの集合をプールすることになります。あくまでもコレクションオブジェクトがクラスとは別個に存在しなければならない点に注意してください。（同上）

■平澤章[56]のクラス論

平澤章のクラス論は、羽生田と共通する点があるとともに、大きな違いもふくんでいます。
まず、上流工程について、次のように述べています。

プログラミングの前段階で・・・コンピュータ化の対象となる現実世界の仕事の様子を整理する業務分析や、コンピュータに肩代わりさせる仕事の範囲を決める要求定義などの作業をきちんと行

56　後述する『オブジェクト指向でなぜつくるのか』（日経 BP 社）、『UML モデリングレッスン』（日経 BP 社）の著者

っておく必要があります。(『オブジェクト指向でなぜつくるのか』[57], p. 188)

では、オブジェクト指向は、このような上流工程においてはどのように役立つのでしょうか。平澤は次のように考えています。

> クラス、ポリモーフィズム、継承といった OOP の基本的な仕組みはプログラミングのために考えられたため、プログラムの設計には素直に適用することができました。しかし、いわゆる上流工程の仕事である業務分析や要求定義には必ずしもうまく当てはまりませんでした。これは第2章で議論したように、現実世界と OOP は似て非なるものだからです。(同上, p. 189)

> そして使えるところだけを応用した結果、上流工程においてオブジェクト指向は、2つの基本的な仕組みを提供することになりました。一つは集合論で、もう一つは役割分担の考え方です。(同上, p. 189)

> OOP では、クラスはサブルーチンと変数をまとめるものでした。そしてインスタンスは、実行時に確保されるメモリ領域でした。この「1 つのクラスから実行時にたくさんのインスタンスを作る仕組み」が、集合論の「集合」と「要素」によく似ていたため、上流工程でクラスとインスタンスは集合論に応用されることになりました。(同上, p. 189)

つまり、平澤は、クラスが、インスタンスとの関係から、その意味について変容を被り、いわば、副次的な意味を派生させたと考えています。

さらに、クラスに関する「役割分担の考え方」については、次のように論じています。

> OOP において「メッセージパッシング」は、インスタンスを指定してメソッド(すなわちクラスにまとめられたサブルーチン)を呼び出すことでしたが、上流工程では「ある特定の役割を持つもの同士が、決められた方法で連絡し合う様子」を表現する、一般的な役割分担のモデルに応用されました。(同上, p. 190)

以上のクラスに関する二つの捉え方が、上流工程に対するオブジェクト指向の適用を可能にしているということです。平澤は、これら二つの捉え方をまとめて、「汎用の整理術」と呼んでおり、この「汎用の整理術」のために、クラスは二義性をもち、そこから、クラスおよびオブジェクト指向理解についての混乱が生じたと考えています。

---

57 平澤 章, 2004, 『オブジェクト指向でなぜつくるのか』, 日経 BP 社

さて、オブジェクト指向の適用範囲が広がった一方で、異なる 2 つの概念に適用されたことによる混乱も生まれました。

何しろクラスという用語は、整理術としては「現実に存在するもの」を表現し、プログラミングにおいては「サブルーチンと変数をまとめる仕組み」を指すわけです。そして、これらはまったく異なるものです。(同上, p. 192)

オブジェクト指向には、抽象的な「汎用の整理術」と、地に足のついた「プログラミング技術」というまったく異なる 2 つの側面がある。(同上, p. 193)

大まかには、上流工程は「汎用の整理術」、設計以降の下流工程は「プログラミング技術」と考えれば大丈夫そうです。(同上, p. 194)

平澤は、「設計」を下流工程と考えており、設計段階では、オブジェクト指向(あるいはクラス)の二つの側面、「汎用の整理術」と「プログラミング技術」の両方が必要になるとしています。

設計段階では、オブジェクト指向の2つの顔である汎用の整理術とプログラミング技術の両方が必要になります。(同上, p. 259)

本書はここまで、オブジェクト指向にはプログラミング技術と汎用の整理術の2つの顔があり、それぞれを別物として分けて考えるべきと主張してきました。しかし設計段階では、汎用の整理術とプログラミング技術の両方が必要になります。そして、この2つを同時に考えることこそがオブジェクト指向設計の感覚と言えるでしょう。(同上, p. 272)

■児玉公信のクラス論

児玉もやはり、クラス、概念、集合の関係について論じています。

以下の概念論は、「共通性」を概念の内包、「集合」を概念の外延と置き換えれば、ほぼそのまま伝統的論理学における概念論と等しいものです。

人は生活をしていくなかで、食堂やレストランに行って食事するという経験を何度も繰り返し、食事をするためにはどのように振る舞うべきかを学習します。これは、経験を重ねるうちに、「料理」「注文」「請求」といったものごとについて類似性または法則性を見い出して、メニューを見て料

理を指定することを、一般に「注文」と呼び、それをおいそれとは変更できないことを学びます。このような一つひとつの経験を集めて、それから共通性を抽象化する知的操作を**帰納**といいます。ここで扱われる一つひとつの経験に属するものごとを**インスタンス**、そして抽象化されたものごとを**概念**と呼ぶことにします(図 1.3)。(『UML モデリング入門』[58], p.14, 原文中の脚注番号および参照指示された図 1.3 は省略)

概念とは、簡単に言えばこのように類似のインスタンスを集めて、それに命名したものと言えます。たとえば、「商品」という概念は、A 店の「牛丼」や B 店の「みそラーメン」といったインスタンスを要素とする集合の名前です。言い方を換えると、概念とは、ものごとを類似性によって集合に分類(classify)して命名したものです。分類とは、まさに「類」である集合に、個々のことがらを「分ける」ことなのです。(同上, p.15)

さらに、概念とクラスは、次のように関係づけられています。

モデリングを行うときに、概念を集合として扱うのはよいことですが、これを数式で記述するのは不便です。そこで、ベン図などの集合図を使って構造を記述する工夫がされたわけですが、概念の数が増えるとこれも煩雑になってしまいます。そこで、インスタンスを抽象化して、概念を属性種と操作を持った**型(タイプ)**として扱います。ここで操作とは、その型が他の型に向けて提供できるサービスのメニューです。この型を、オブジェクト指向ではクラスの原型と考えてもよいでしょう。データモデリングではテーブル(表)の原型と考えてもよいでしょう。(同上, p.18)

概念レベルでは、概念と概念の関係を UML のクラス図を使って、構造として記述します。クラス図の記法は、もともと、オブジェクト指向プログラミング言語の「クラス」を記述するための記法です。なので、この記法を使って概念(正確にいうと概念の「型」)を記述するのは、本来の使い方ではないと思われるかもしれません。でも、私たちは、第1章に書いたように、対象世界の概念構造を把握しておきたいのです。人の思考や情報の処理はすべてこうした概念を使って行われているからです。(同上, p.50)

さらに、汎化と集合の関係は次のように規定されています。

・・・いくつかの概念がその上位の概念に包含されると定義することを汎化(generalization)と呼びます。逆に、全体概念を下位の概念に分類することを特化(specialization)と呼びます。汎化と特化はつねに裏腹な関係にあるので、汎化という言葉のほうを優先して使うことにします。

58 児玉 公信, 2008, 『UML モデリング入門』, 日経 BP 社

（同上, p.75-76）

（注:汎化について）白抜き三角形の矢尻がついているほうを集合全体にあたる型とし、矢の柄についているほうを部分集合にあたる下位の型とします。（同上, p.76）

16.3 UML の正式仕様書におけるクラスと集合の関係

UML の正式仕様書の中で、クラスと集合の関係がどのように規定されているか確認します。ここでは、OMG Unified Modeling Language (OMG UML),Superstructure, V2.1.2 を参照します。

7.3.7 Class (from Kernel)の項では、「クラス」について、次のように規定されています。

■原文

A class describes a set of objects that share the same specifications of features, constraints, and semantics.
("OMG Unified Modeling Language (OMG UML),Superstructure, V2.1.2", p.49)

■訳

クラスは、特性、制約、意味からなる同一の仕様を共有するオブジェクトの集合(set)を表現します。

7.3.8 Classifier (from Kernel, Dependencies, PowerTypes)の項では、分類子(Classifier)について、次のように規定されています。

■原文

A classifier is a classification of instances, it describes a set of instances that have features in common.
("OMG Unified Modeling Language (OMG UML),Superstructure, V2.1.2", p.52)

■訳

分類子は、インスタンスの分類であり、特性を共有するインスタンスの集合を表現します。

7.3.3 Association(from Kernel)の項では、次のように述べられています。

■原文

Subsetting represents the familiar set-theoretic concept. It is applicable to the collections represented by association ends,not to the association itself. It means that the subsetting association end is a collection that is either equal to the collection that it is subsetting or a proper subset of that collection. (Proper subsetting implies that the superset is not empty and that the subset has fewer members.) Subsetting is a relationship in the domain of extensional semantics.

Specialization is, in contrast to subsetting, a relationship in the domain of intentional semantics, which is to say it characterized the criteria whereby membership in the collection is defined, not by the membership. One classifier may specialize another by adding or redefining features; a set cannot specialize another set. A naïve but popular and useful view has it that as the classifier becomes more specialized, the extent of the collection(s) of classified objects narrows. In the case of associations, subsetting ends, according to this view, correlates positively with specializing the association.This view falls down because it ignores the case of classifiers which, for whatever reason, denote the empty set. Adding new criteria for membership does not narrow the extent if the classifier already has a null denotation.
("OMG Unified Modeling Language (OMG UML),Superstructure, V2.1.2", p.40)

■訳

部分集合化は、よく知られた集合理論の概念を表現します。部分集合化は、関連端によって表現されたコレクションに対して適用され、関連自身には適用されません。このことは、部分集合化する関連端は、部分集合化されるコレクションに等しいか、そのコレクションの適正な部分集合であることを意味します(適正な部分集合化とは、包括的集合(superset)が空集合ではなく、部分集合が包括的集合よりも少ないメンバーをもつことを意味します)。部分集合化は、外延的意味の領域における関係です。

特化(Specialization)は、部分集合化とは対照的に、内包的意味の領域における関係です(訳注:部分集合化は外延的意味の領域における関係)。このことは、特化は、メンバーシップをではなく、コレクションにおけるメンバーシップを定義する基準を特色づけることだということを意味します。分類子は、特性を追加したり再定義したりすることによって、他の分類子を特化することができます。しかし他方、集合は他の集合を特化することができません。素朴ですがよく知られた便利な考え方によれば、分類子は特化されるにつれて、その分類子に分類されるオブジェクトのコレクションの範囲は狭まっていくとされています。この考え方を関連に適用した場合、部分集合化する関連端は(訳注:コレクションの範囲が狭まった関連端)、関連の特化に伴って生じることになります。(訳注:しかし、)この考え方は、空集合(空である理由は何であれ)を外延として示す分類子の存在を無視するため成立できません。分類子が空の外延をもっていた場合には、メンバーシップのための新しい基準を追加しても、その範囲を狭めることにはなりません(訳注:前文の不成立の理由)。

16.4 概念モデル/論理モデル/実装モデルの関係

前述のように、システム化の対象は、問題領域(ドメイン)であり、これは現実そのものであるということもできます。そのような現実そのものに対して一定の抽象化を加え、現実の中に見いだされた概念によって、問題領域(現実そのもの)を再構成するものが概念モデルです。一定の抽象化が必要なのは、現実を構成する個物が帯びている多様性(特殊的性質、つまり、偶有的属性)を捨象し、本質的属性を抽象することによって概念を構成する必要があるからです。

しかし、概念モデルは、それ自体では実装技術(実装モデル)と結びつきません。そこで、実装可能なモデルへと変換する必要があります。それが論理モデルです。概念モデルと論理モデルとの結びつきは、観念的な等価物であるという点にあると考えられます。実体としてはそれぞれまったく異なるものでありながら(論理モデルには操作、つまり、メソッドがある)、観念的な等価性が、両者の"等しさ"を保証します。逆にいえば、概念モデルと論理モデルの間の等価性は、形式化することが困難であるように

思われます。他方、両者は観念的に等価ではありますが、概念モデルとは異なり、論理モデルは実装可能です。別のいい方をすれば、論理モデルは、実装モデルによって表現可能です。

17アクティビティ図

17.1 アクティビティ図の意味と用途

アクティビティ図は、「ビジネスプロセス」、「ワークフロー」および「プログラムフロー」(プログラムの処理の流れ)を表現する際に用いることができます[59]。「ビジネスプロセス」、「ワークフロー」とは事業や業務におけるプロセスをモデル化したものであり、問題領域において発生するプロセスを意味します。これは、分析の段階で作成されるものです。一方、実装の段階では、実装者が作成すべきプログラムの処理の流れを正確に把握できるように、アクティビティ図を利用することができます。つまり、アクティビティ図は、分析および実装の段階において、両方の用途に対応することができます。

17.2 制御フローとオブジェクトフロー

■アクティビティ(Activity)

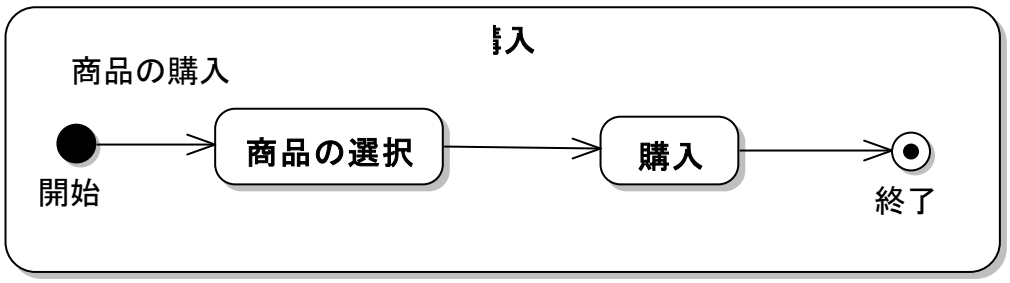
アクティビティは、手続き型処理もしくは何らかの主体による振る舞いを表現することができます。この場合、アクティビティ図によってクラスのメソッドの処理全体を表現します。一方、アクティビティは、企業全体もしくは企業間のビジネスプロセスや企業内部におけるワークフローを表現することもできます。アクティビティは、主に以下のようなプロセスを表現することができます。

・C 言語による算術関数のような関数の動作
・企業全体および内部における生産活動
・イベントに基づくコミュニケーション動作
・属性の読み書きのようなオブジェクトの操作

[59] Fowler, Martin, 2004, UML Distilled Third Edition, Pearson Education（羽生田 栄一監訳, 2005, 『UML モデリングのエッセンス 第 3 版』, 翔泳社)、および、(株) NTT データソフトウェア工学推進センタ, 2008, 『実例で学ぶソフトウェア開発』, オーム社 参照のこと

■アクション(Action)

アクションはアクティビティ内の単一のステップ(単一の振る舞い)を表現します。アクションはそのアクティビティ内ではそれ以上分解されない要素です。1つのアクティビティは個々の要素である複数のアクションによって、振る舞いの全体を表現します。



上記では、「開始ノード」(InitialNode)、「アクションノード」(ActionNode)が二つ、「アクティビティ終了ノード」(ActivityFinalNode)から、ひとつのアクティビティが構成されています。ノードとノードの間の接続線は、「アクティビティエッジ」(ActivityEdge:単に「エッジ」ともいう)といいます。これによって処理の流れが表現されます。(開始ノード、アクションノード、終了ノードなどはすべて、「ノード」(Node)です)。

■アクティビティノード(ActivityNode)

アクティビティエッジ(エッジ)によって結ばれた点を表現します。単に「ノード」(Node)ともいいます。ノードには、大別して、アクションノード、オブジェクトノード、制御ノードの三種類があり、制御ノードには、さらに、デシジョンノード、フォークノード、開始ノード、アクティビティ終了ノード、終了ノードといった種類があります。

■アクティビティエッジ(ActivityEdge)

ノード間を接続し、トークンの移動を表現します。

■トークン(token)

ノードからノードへ処理が流れていくことを、アクティビティ図では、「トークン」(token:しるし・代用貨幣)が流れていくと捉えます。トークンは、ノードからノードへと受渡されるオブジェクト、データを意味する場合と、制御上の位置を示す単純なマーカー(locus of control)[60]を意味する場合があります。

ノードは、入力エッジからすべてのトークンを取得し、トークンを内部に取り込みます。ノードが処理を完了した段階で、内部に取り込まれたトークンは出力エッジを通して外部に出ていきます。

60 OMG, “OMG Unified Modeling Language (OMG UML),Superstructure, V2.1.2”, p.318, 2007-11-02

■制御フロー(ControlFlow)

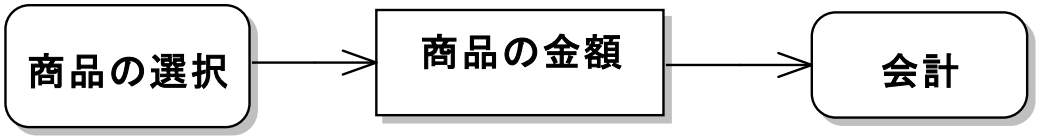
制御フローは「制御トークン」[61](制御上の位置を意味する単純なマーカーを意味する)のみを渡すアクティビティエッジです。起点ノードによって提供されるトークンは、そのまま終点ノードへすべて提供されます。

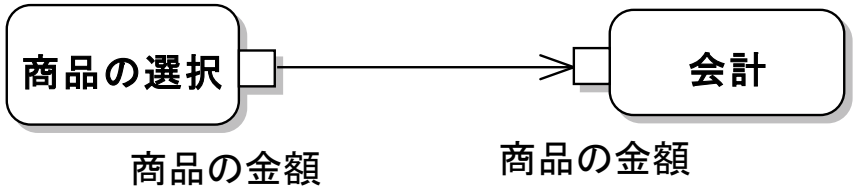


■オブジェクトフロー(ObjectFlow)

オブジェクトフローはアクティビティエッジであり、各ノードは通過するオブジェクトあるいはデータを利用することができます。

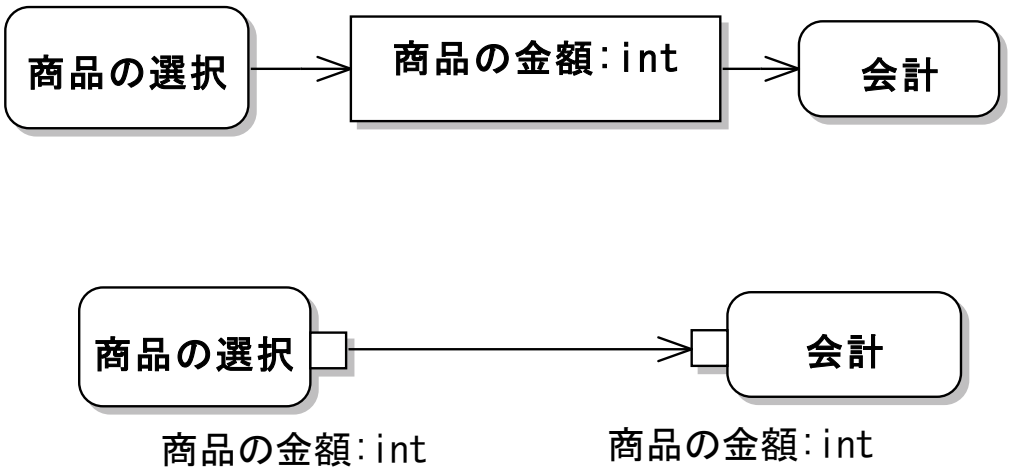
オブジェクトフローは、四角形によって表現される「オブジェクトノード」(ObjectNode)、もしくは、「入力ピン」(InputPin)と「出力ピン」(OutputPin)によって表現します。





次のように、「名前:データ型」の形式で表現することもできます。

61 "control tokens" (OMG Unified Modeling Language (OMG UML),Superstructure, V2.1.2, p.355)

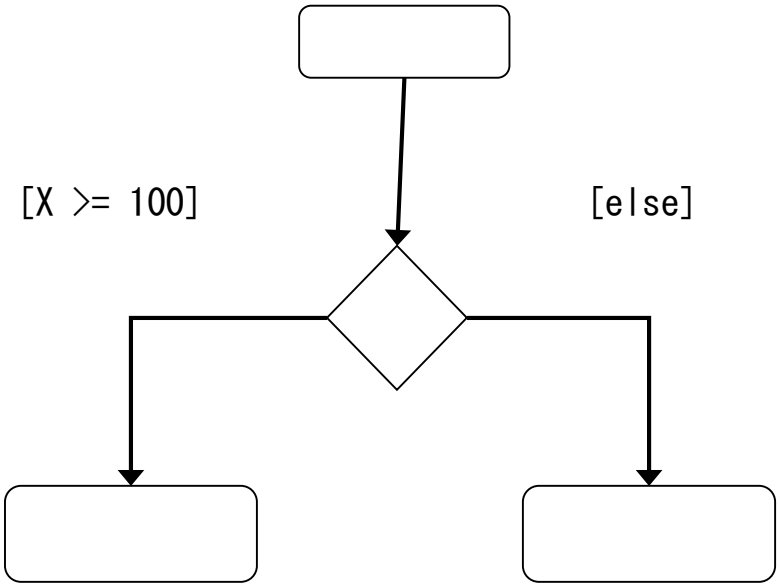

商品の選択
商品の金額:int
会計
商品の選択
会計
商品の金額:int
商品の金額:int

17.3 分岐処理の表現

切符を購入する際に、投入した金額が切符の金額に満たなかった場合は、お金の追加投入が必要になります。また、プログラム内部でも分岐処理(if や switch)は頻繁に用いられます。これらを表現する手段が、「デシジョンノード」(DecisionNode)と「マージノード」(MergeNode)です。

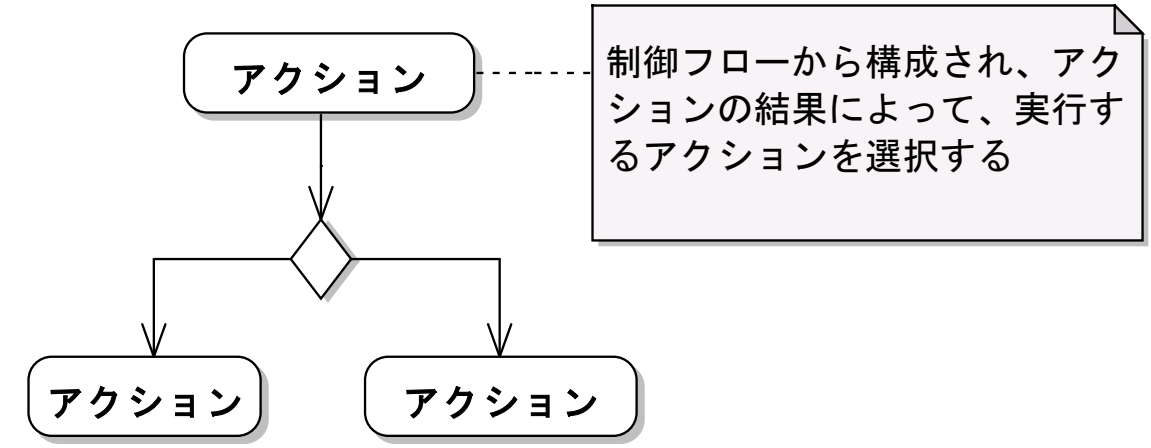
条件の分岐を表現するためのノードがデシジョンノードです。デシジョンノードは、入力エッジから入力されたトークンが、いずれか 1 つの出力エッジをたどります。

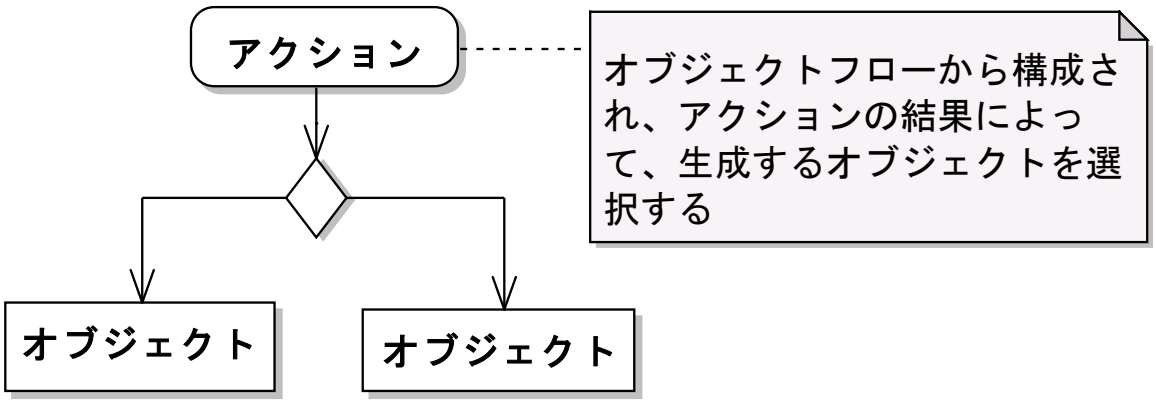
デシジョンノードは、次の図のように白抜きひし形によって表記します。分岐の条件は、「ガード」(guard)によって記述します。ガードによって表現される条件を「ガード条件」(guard condition)といいます。ガードは、「[ ]」によって表記します。

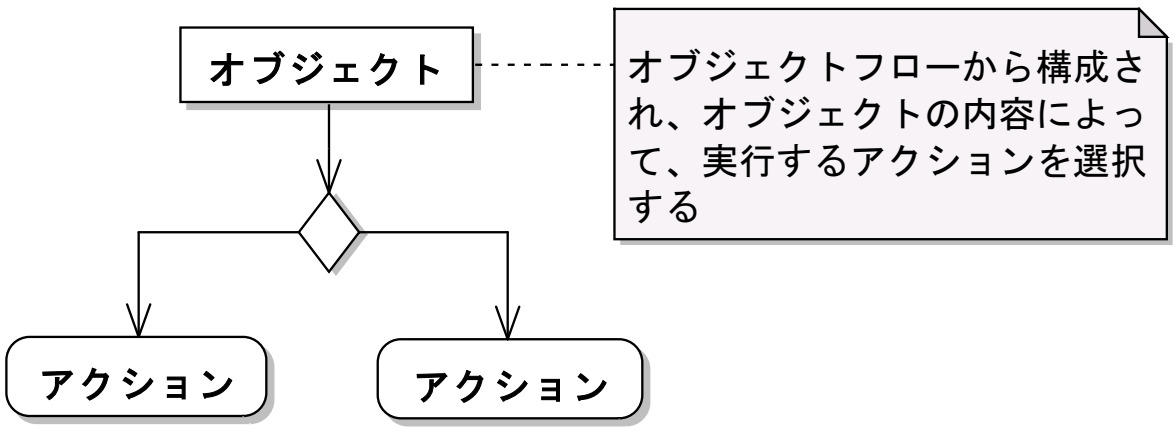


デシジョンノードは、単一の入力エッジと複数の出力エッジをもちます。複数の出力エッジの中で、ひとつだけ「[else]」というガード条件を記述することができます。

制御フロー/オブジェクトフローという観点からは、以下の3パターンがあり得ます。



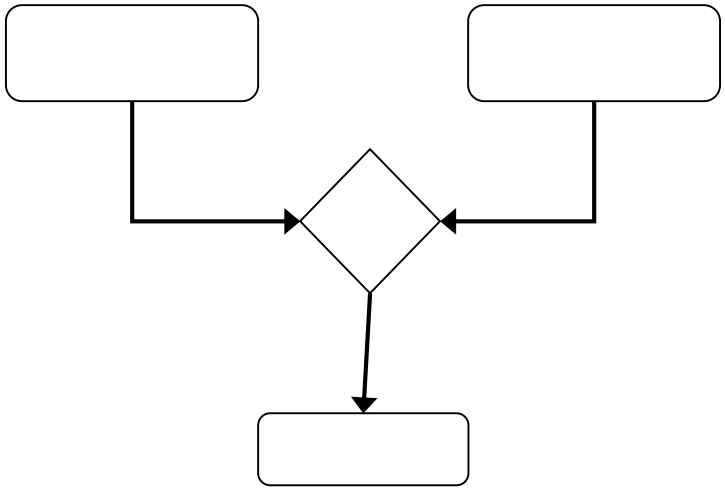




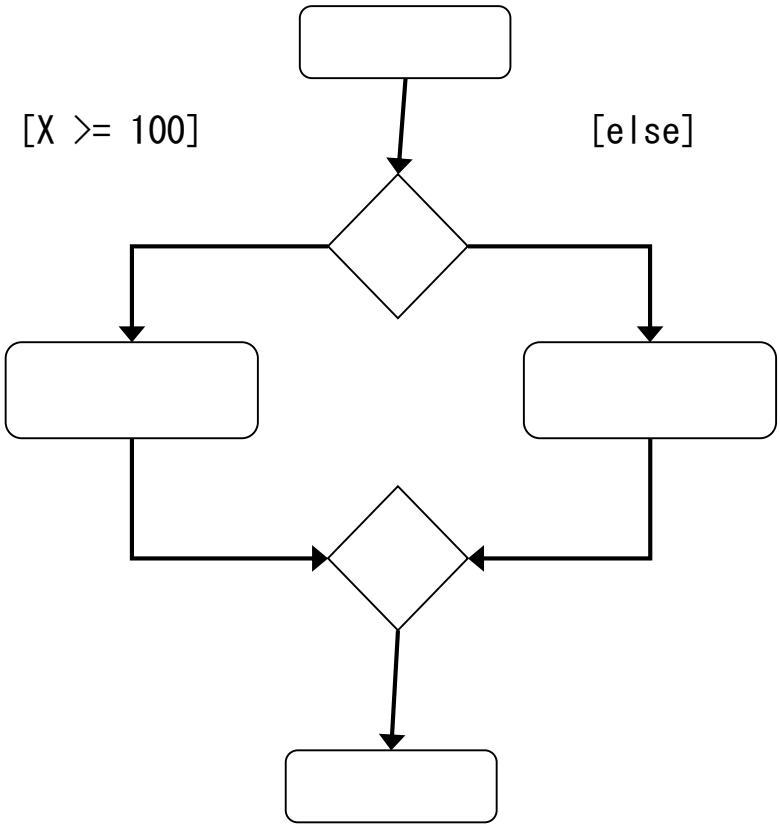
分岐の終了を表現するのは、マージノードです。マージノードでは、複数の入力エッジのどれかひとつから入力されたトークンが出力エッジに流れていきます。もし、複数の入力エッジから、それぞれトークンが入力された場合は、出力エッジに複数のトークンが渡され、それにより、マージノードからのアクテ

ィビティエッジの終点となるノードが複数回起動されます。

マージノードは、デシジョンノードと同様、次のように白抜きのひし形によって表記します。



分岐処理を含むプログラムの実装を表現する場合、一般的には、デシジョンノードとマージノードを組み合わせて使用します。



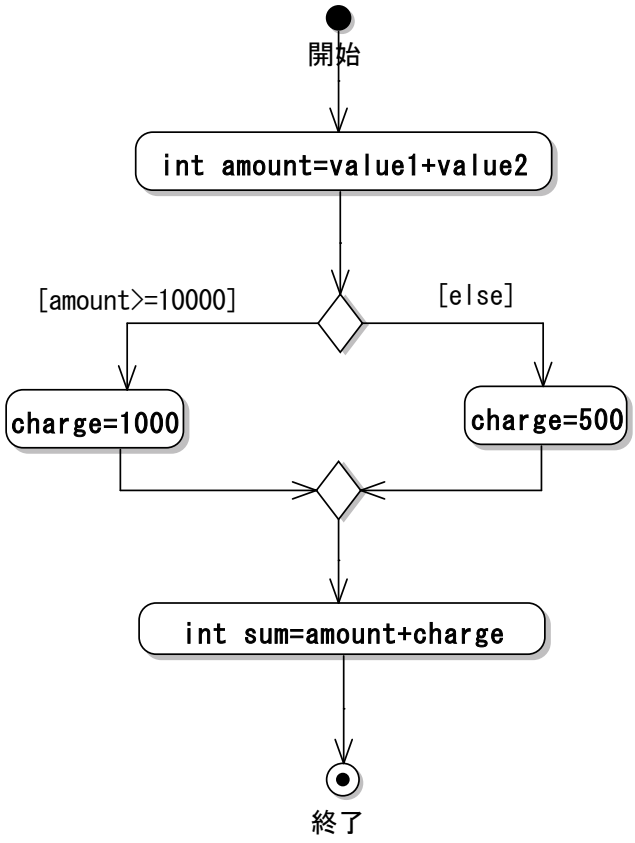
たとえば、次のようなソースコードが存在するとします。

```
int amount = value1 + value2;
int charge = 0;

if(amount >= 10000){
  charge = 1000;
}else{
  charge = 500;
}

int sum = amount + charge;
```

このソースコードのプログラムフローをアクティビティ図によって表現した場合、次のようになります。



17.4 反復処理の表現

デシジョンノードとマージノードによって、for や while のような反復処理を表現することができます。

たとえば、次のような Java のソースコードの場合、マージノード、デシジョンノードの順で結ぶことにより表現します。

```
int x = 50;

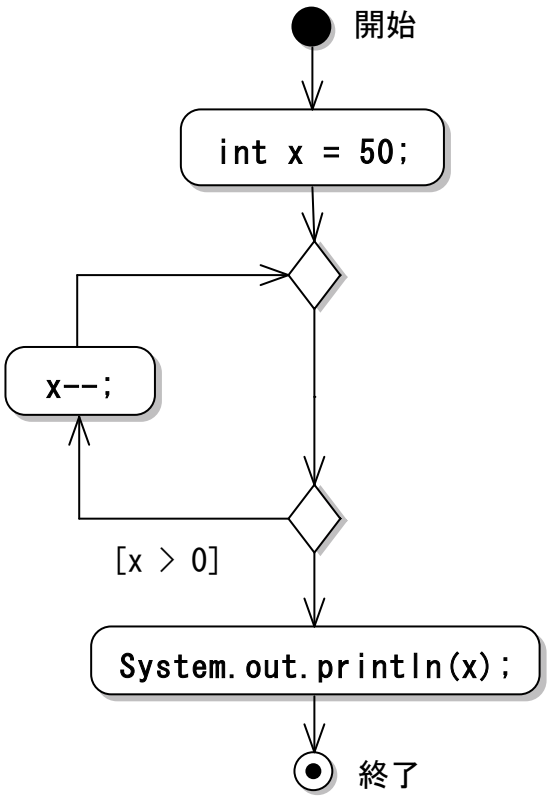
while(x > 0){

  x--;

}

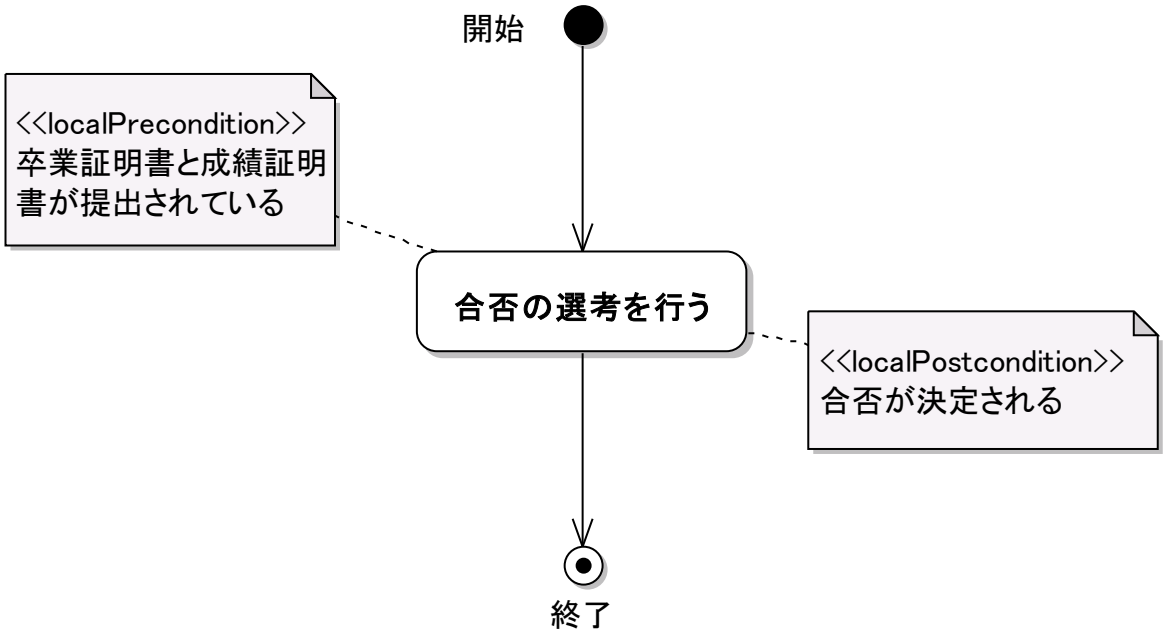
System.out.println(x);
```

このソースコードは以下のように表現できます。


開始
int x = 50;
x--;
[x > 0]
System.out.println(x);
終了

17.5 アクションの事前条件と事後条件

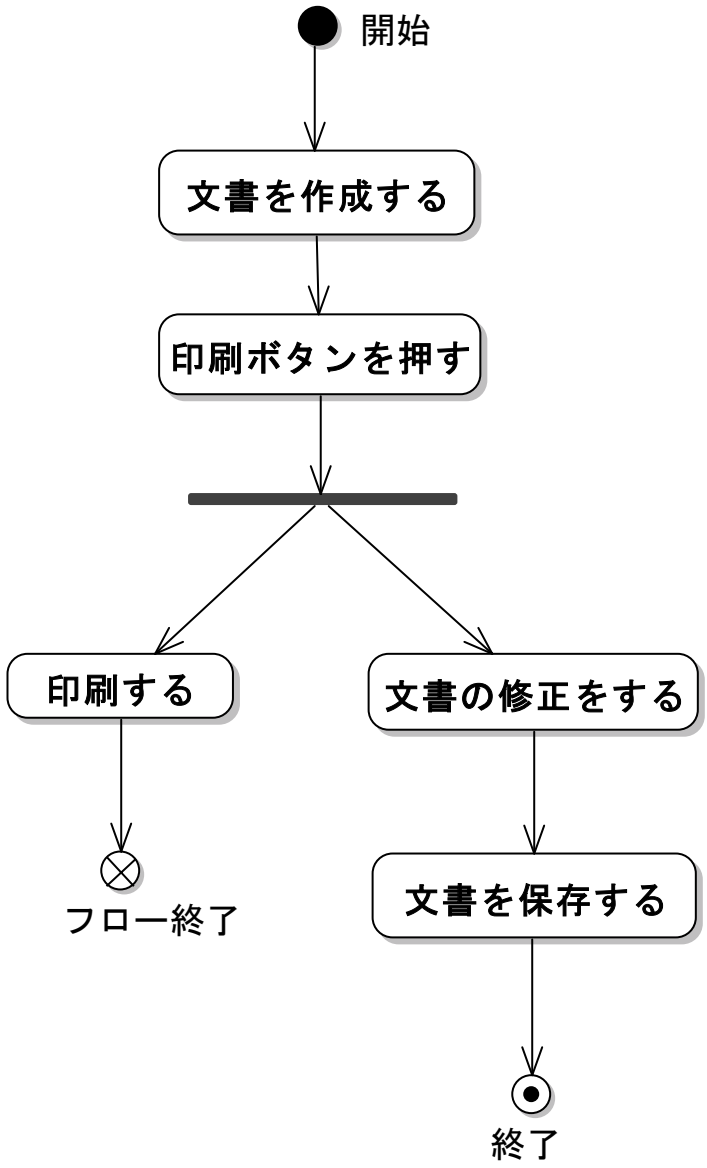
アクションの実行に関して、そのアクションを実行するための前提となる事後条件と、アクション実行後に維持されるべき事後条件を表現することができます。



17.6 並行処理と同期処理の表現

マルチスレッドプログラミングやビジネスプロセス・ワークフローにおける並行処理・同期処理を表現するのは、「フォークノード」(ForkNode)と「ジョインノード」(JoinNode)です。並行処理はフォークノードによって表現され、同期処理はジョインノードによって表現されます。フォークノードにおいては、トークンがコピーされて、すべての出力エッジに渡されます。

次はワープロソフトにおけるマルチスレッド処理をフォークノードによって表現したものです。

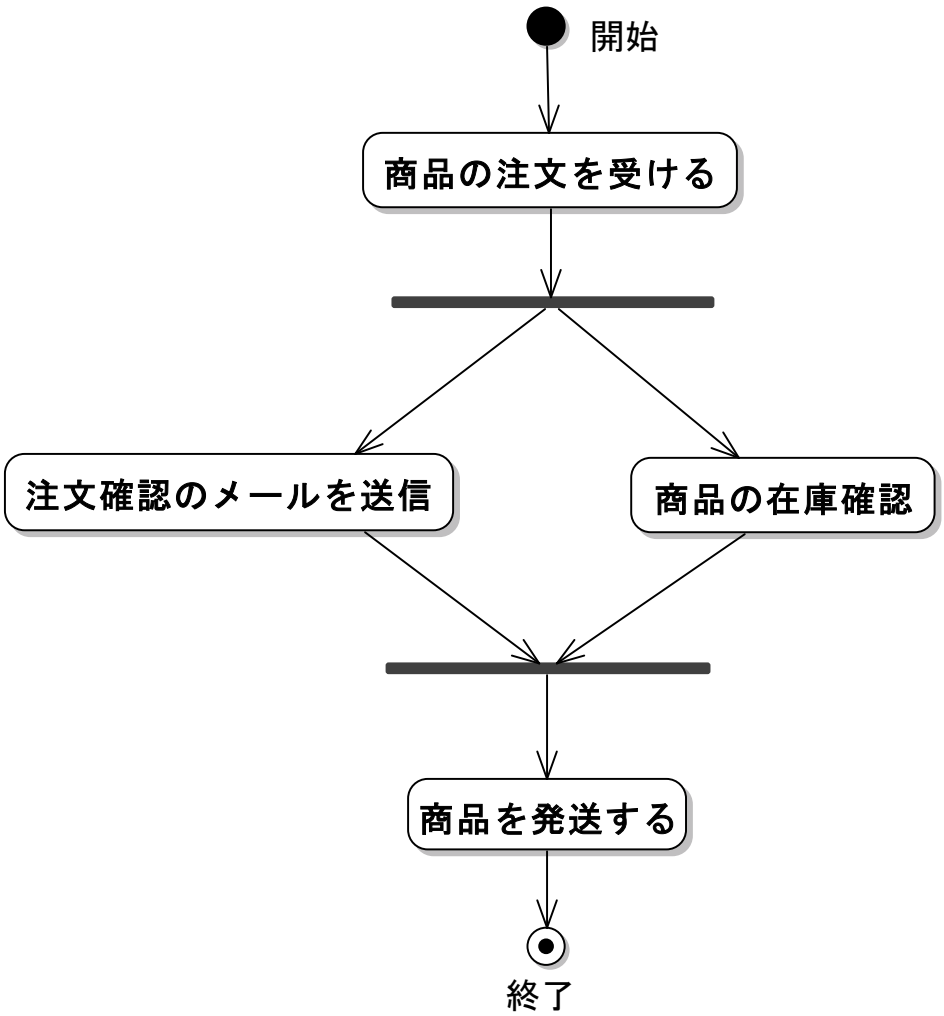


印刷が終了した場合は、「フロー終了ノード」(FlowFinalNode)によってそのスレッドの終了を表現し

ます。「終了ノード」はそのアクティビティ全体の終了を意味します。

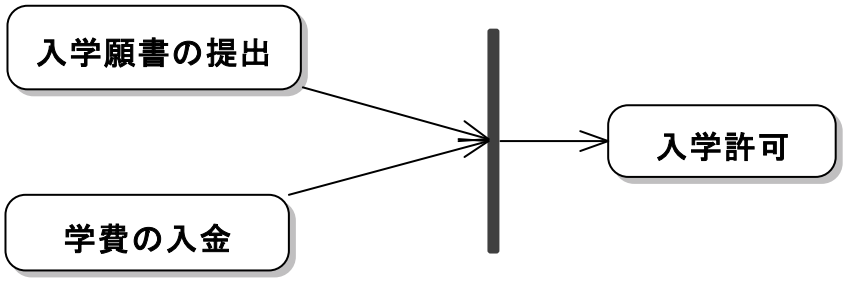
次の例では、並行処理がジョインノードによって同期される様子を表現します。

ジョインノードでは、すべての入力エッジからトークンが入力された時に処理が行われます。したがって、一部の入力エッジからのトークンが入力されただけでは、処理は先へ進まず、他の入力エッジからのトークンの入力を待機した状態になります。入力エッジに提供されるトークンがすべて制御トークンの場合、それらが 1 つの制御トークンにまとめられて出力エッジに提供されます。トークンがオブジェクトやデータの場合はその限りではありません。



この場合、メールの送信と商品の在庫確認の両方の処理が完了した段階で、商品を発送する処理が実行されます。

ジョインノードがトークンを出力する際の条件である「ジョイン仕様」（joinSpec）を指定することもできます。

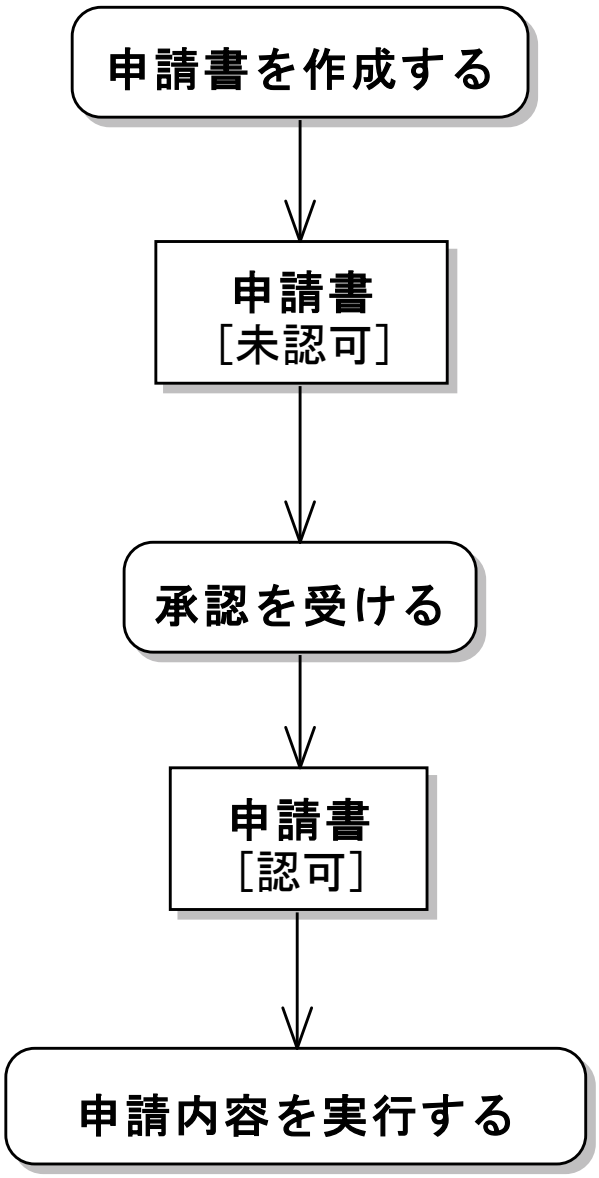


{joinSpec=提出日<=提出期限 and 入金日<=入金期限}

17.7 データの遷移の表現

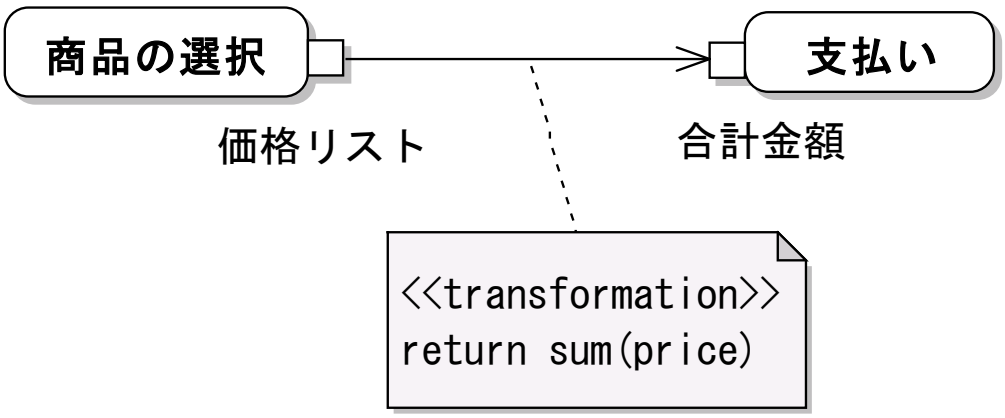
■データの状態遷移の表現

オブジェクトノードに含まれるトークンの状態の変化を表現することができます。大括弧の内部に状態を意味する記述を行います。コンマで区切ることにより、複数の状態を記述することもできます。

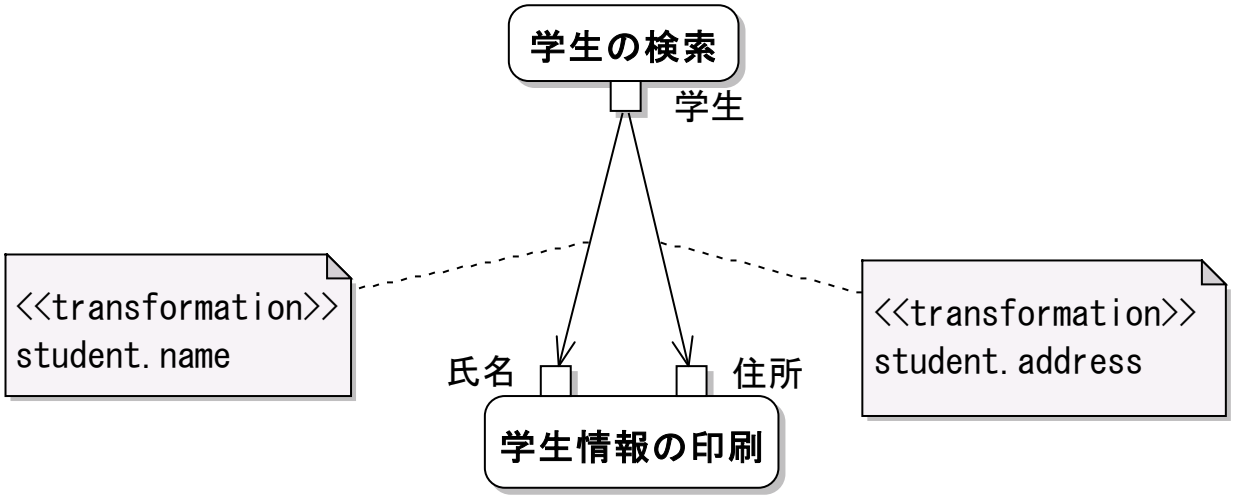


■データ変換の表現

オブジェクトフローにおいて起点ノードから終点ノードに渡されるトークン(データもしくはオブジェクト)の変換を表現することができます。変換とは、あるオブジェクトから他のオブジェクトを取得したり、オブジェクトから属性値を取得したり、単純にデータ値を変換したりすることを意味します。



上記の例では、商品選択の結果、価格リストデータが形成されますが、支払い処理が必要なのは、合計金額のデータです。そこで、オブジェクトフローにおいて変換の処理を行います。



上記の例では、ひとつのオブジェクトからプロパティを参照することにより、2 つのオブジェクトを

取得する仕方を表現しています。

この場合は、ピン（オブジェクトノード）から複数の出力エッジにトークンが提供されます。

17.8 コレクション・配列の表現

■データ集合の表現

オブジェクトノードは、内部に複数のトークン(データやオブジェクト)を保持することができます。さらに、オブジェクトノードによって保持された複数のトークンに対して、一定の順序を指定することができます。その場合、オブジェクトノードの下に中括弧で囲んで表記します。



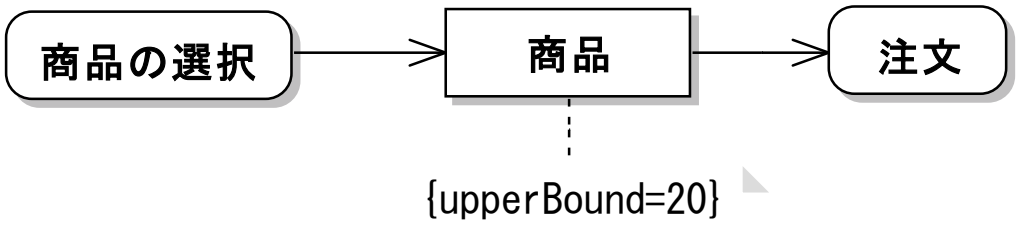
順序付けの指定には、「ordering=・・・」という書式を用います。指定可能な値は、以下の通りです。

unordered	並べ替えなし
ordered	並べ替えあり
LIFO	Last In First Out
FIFO	First In First Out

順序づけの仕組み自体を定義したい場合は、オブジェクトノードにおける選択と併用します。

■データ集合の要素数の上限値の表現

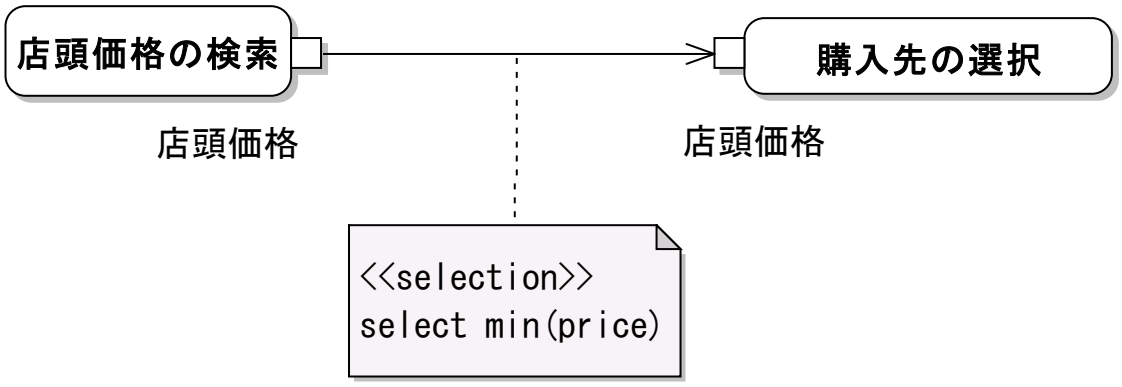
オブジェクトノードは、複数のトークン（データもしくはオブジェクト）を保持することができるため、その上限値の設定が可能です。



上記の図では、トークンの個数の上限値により、商品を 20 個までしか選択・注文できないことを意味します。

■データ選択の表現

オブジェクトフローの起点ノードから終点ノードに対して渡される複数のトークン（データもしくはオブジェクト）から、トークンを選択する処理を表現することができます。

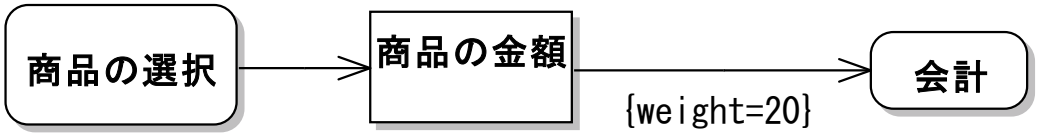


上記の例では、複数の店頭価格から最低価格を選択し、その最低価格を提示する店で商品を購入する処理を表現しています。

■データ数指定の表現

オブジェクトフローに対して、「weight」(重み)を設定することにより、エッジに出力されるトークンの最小数を指定することができます。

エッジに出力されるトークンは、一度にあるいは異なる時にエッジを通ることができます。「weight」属性は同時にエッジを通らなければならないトークンの最小数を示します。新しいトークンがエッジの起点に渡されるたびに評価される条件となります。



上記の図では、選択した商品の価格を、少なくとも20個以上合わせて会計処理に渡すということを意味します。

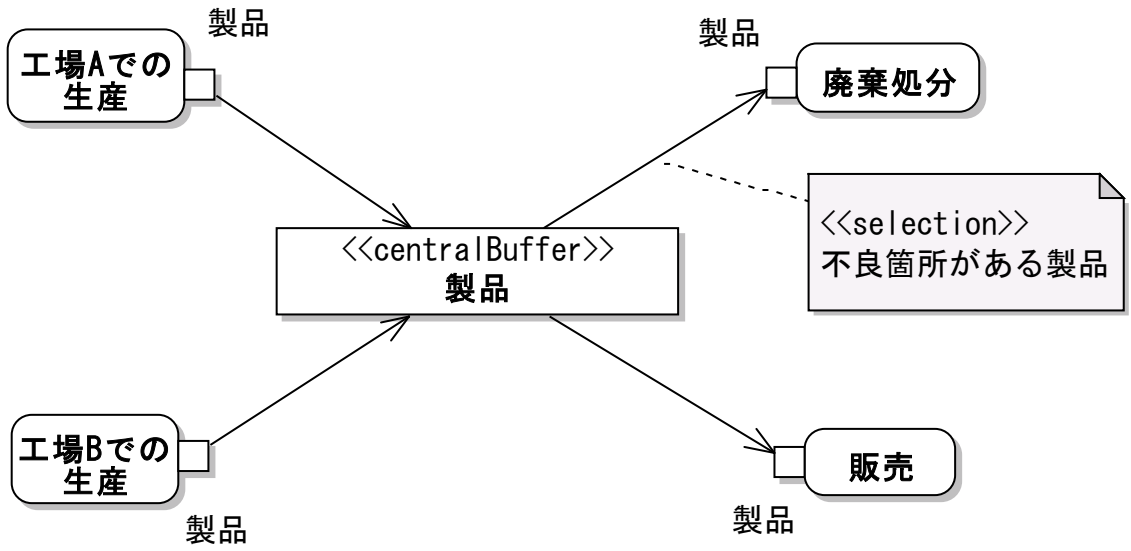
weight の指定には、上記の例のような定数の他、一定の整数として評価される変数や不定の全数を意味する「all」を指定することができます。

■中央バッファノード(CentralBufferNode)

「中央バッファノード」は、複数のオブジェクトノードからの入力と複数のオブジェクトノードへの出力を中継する特殊なオブジェクトノードです。中央バッファノードは、トークン(オブジェクト、データ)の入力と出力の中継を行うだけであり、そのために複数のトークンを一時的に蓄えます。その意味では、中央バッファノードは、トークンの集合体として捉えることができます。

中央バッファノードは、アクションノードと直接接続することはできない点に注意してください。

中央バッファノードは、実装上は、一定の入出力規則をもったコレクションクラスとして捉えることができます。一方、分析においては、概念モデルの一部として利用することもできます。

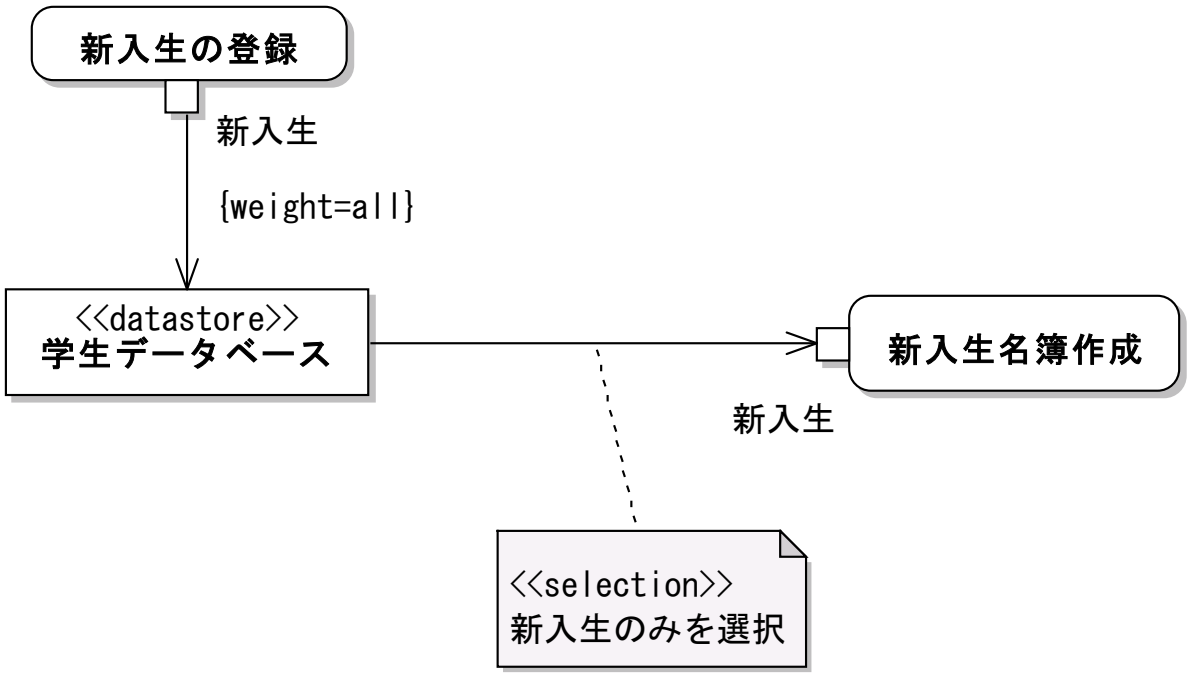


中央バッファノードは、オブジェクトノードであるにもかかわらず、複数の入力エッジと出力エッジを接続できる点に注意します。内部に蓄積されたトークンが、それぞれの出力エッジに対して渡されます。

17.9 データベースの表現

データベースのような永続化されたデータの集合を表現するのが「データストアノード」(DataStoreNode)です。データストアノードは、通常複数のトークンを既に保有しています。中央バッファノードとの違いは、内部のトークンが永続化されており、トークンを出力したとしても、それはコピーされて出力されるため、トークンが減少しないことです。

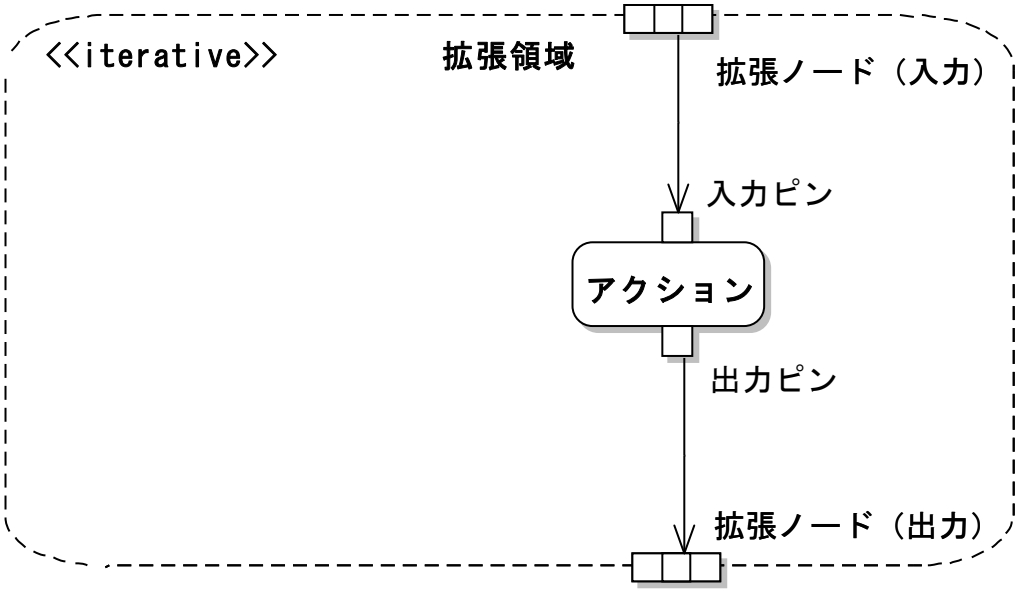
データストアノードに対する入力エッジは、データストアノードに対する新しいトークンの追加もしくは既存のトークンの置換を意味します。これは、リレーショナルデータベースに対する insert・update・delete の実行を表現することができます。データストアノードからの出力エッジは、データストアノードが保有するトークンのコピーの出力を意味します。これは、リレーショナルデータベースに対する select の実行を表現することができます。



17.10 コレクションに対する反復処理の表現

■拡張領域(ExpansionRegion)

配列やマップ、リスト、セットのようなデータのコレクションに対する処理を表現する場合は、「拡張領域」(ExpansionRegion)を使用します。



上記の破線で囲まれた全体が「拡張領域」(ExpansionRegion)です。長方形の内部が垂直に区分されているものが「拡張ノード」(ExpansionNode)です。拡張ノードは、それぞれ、入力されるコレクション、出力されるコレクションを指します。拡張領域の内部のアクションの入力ピンに対しては、入力側のコレクションからひとつの要素が入力され、逆にアクションの出力ピンからはアクションの結果得られた要素が出力側のコレクションに追加されます。

拡張領域の実行の「mode」として、<<iterative>>、<<parallel>>、<<stream>>が選択できます。

<<iterative>> ひとつひとつの要素に対する処理の表現
<<parallel>> 拡張領域内部の並行処理の表現
<<stream>> コレクションや配列ではなく、ストリーミングが入力され、ストリーミングを出力する処理を表現する

注意点は、拡張領域は、コレクションを受け取り、コレクションを返す処理を表現するということです。

■Java による実装サンプル

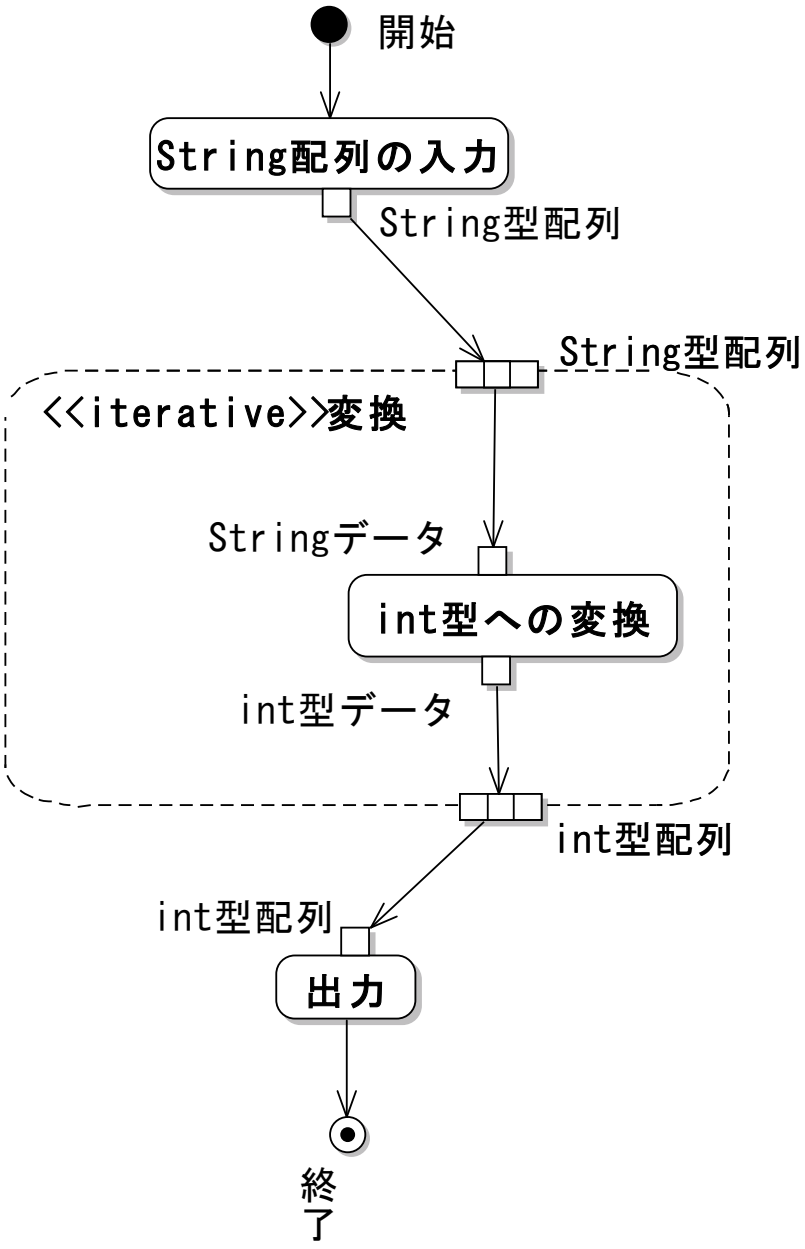
次のような Java のソースコードを拡張領域によって表現します。

```
public class Test {
  public static void main(String[] args) {
      int[] intArray = toIntArray(args);
      System.out.println(intArray);
  }
  // String型の配列をint型の配列に変換する
  private static int[] toIntArray(String[] source) {
      int[] newArray = new int[source.length];
      for (int i = 0; i < newArray.length; i++) {
          String element = source[i];
          int intValue = Integer.parseInt(element);
          newArray[i] = intValue;
      }
```

```
            return newArray;
        }
    }
```

■アクティビティ図による表現

以下のように、拡張領域の内部のアクションは、入力の拡張ノードのコレクションからひとつのトークンを取得し一定の処理を行い、出力の拡張ノードに対してひとつのトークンを返します。



17.11 イベント処理の表現

■シグナル送信アクション(SendSignalAction)とイベント受信アクション(AcceptEventAction)

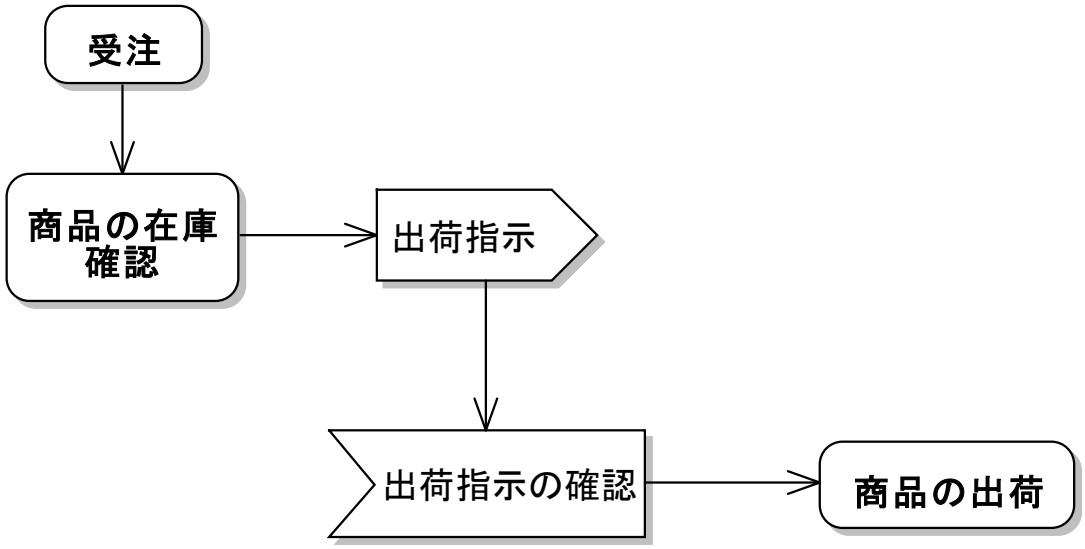
シグナル送信アクションは、非同期の呼び出し(イベントの送信)を表現します。イベント受信アクションは、イベントの受信を表現します。これらは、それぞれ二通りの表現の仕方があります。

出力エッジがないシグナル送信アクションは、ドメイン外部に対するイベントの送信を意味します。

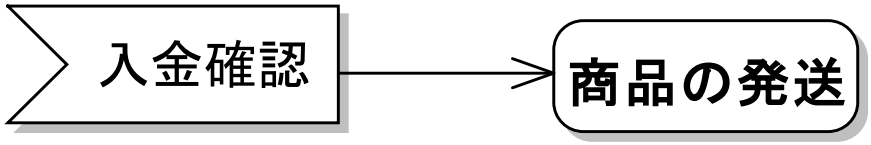
下記の例では、このアクティビティが表現するシステム外部、たとえば、商品倉庫などに対するイベントの送信を意味します。



次の図の場合は、ドメイン内部でイベントの受信も行うことを意味します。「出荷指示」のシグナル送信アクションからの出力により、「出荷指示の確認」イベント受信アクションが有効となり、イベントが受信された後に「商品の出荷」アクションが実行されます。この場合、シグナル送信アクションから出力されるまで、イベント受信アクションは有効になりません(起動していない)。



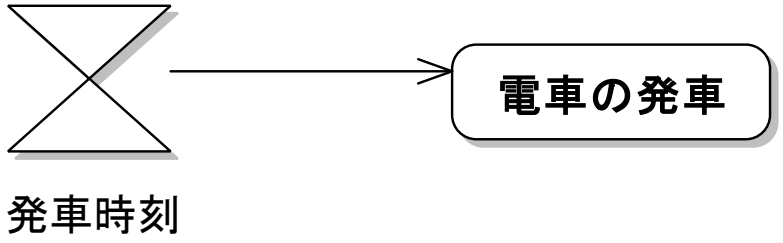
次の図では、入力エッジがないイベント受信アクションが使用されています。

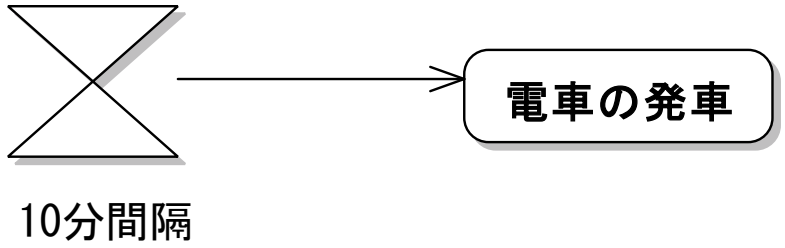


この場合、2 つの注意点があります。ひとつは、アクティビティ自体の起動と同時にイベント受信アクションが開始され、ドメイン外部からのイベントの受信を待機し続けることです。もうひとつは、イベントを受信して次のアクションにトークンを渡した後でもイベントの受信待機を継続することです。いちどイベントを受信して次のアクションを起動した後に、再度イベントが発生すれば、それを受信するということです。

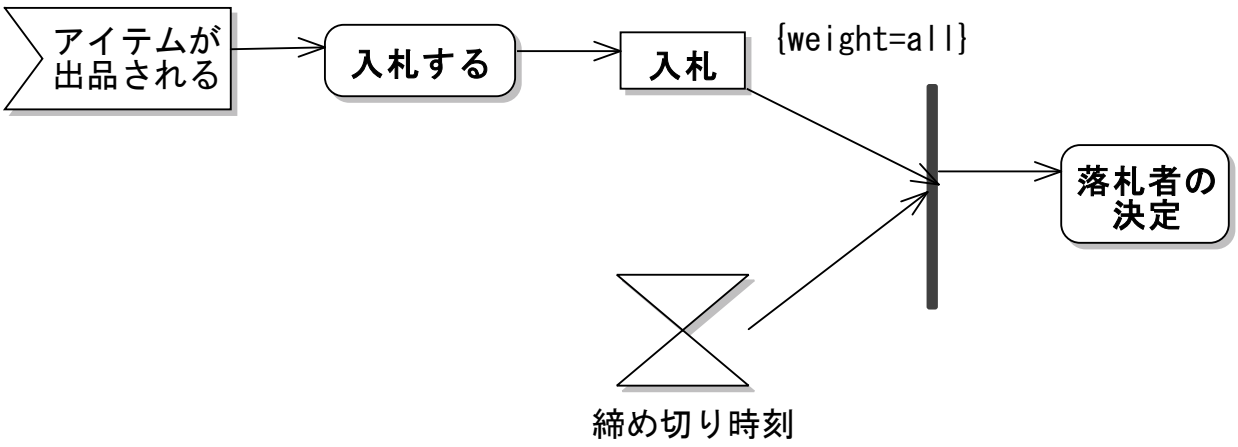
■タイマーイベント受信アクション(accept time event action)

イベント受信アクションのバリエーションとして、一定間隔ごとに発生するイベントや一定のタイミングや定刻に発生するイベントの受信を表現するのが、タイマーイベント受信アクション(accept time event action)です。

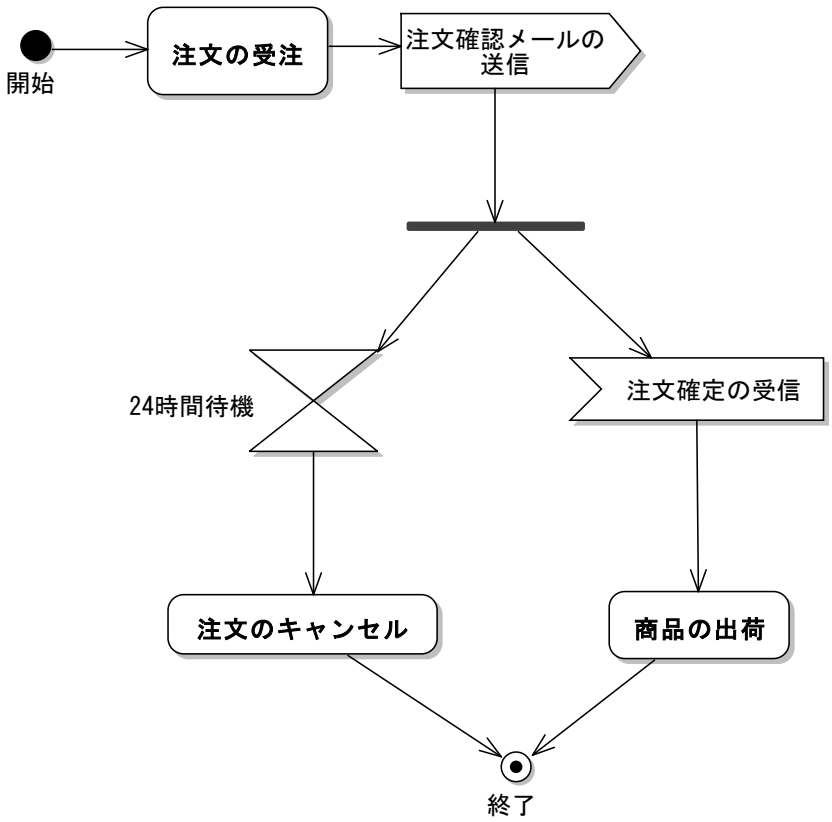




次の例は、オークションへの入札と落札者の決定という振る舞いをアクティビティによって表現したものです。出品者がアイテムを出品するという外部イベントの受信と、オークション締め切り時刻に到達するというイベントの受信を表現しています。



次の例では、注文のキャンセルか商品の出荷のどちらかが起動される状況を表現しています。



入力エッジの有無に関して、タイマーイベント受信アクションは、受信イベントアクションと同じ扱い方になります。

17.12 振る舞いの主体の表現

■アクティビティパーティションの意味と目的

アクティビティ図では、振る舞い間の接続や連続性を表現します。これまでの表記では、振る舞いの主体(クラス、オブジェクト、人物、役割、組織など)を表現することができませんでした。「アクティビティパーティション」(ActivityPartition)は、そのような振る舞いの主体を表現するための記法です。

アクティビティパーティションにおけるひとつの「区画」(partition)に含まれる振る舞いは、次のいずれかを表現します。

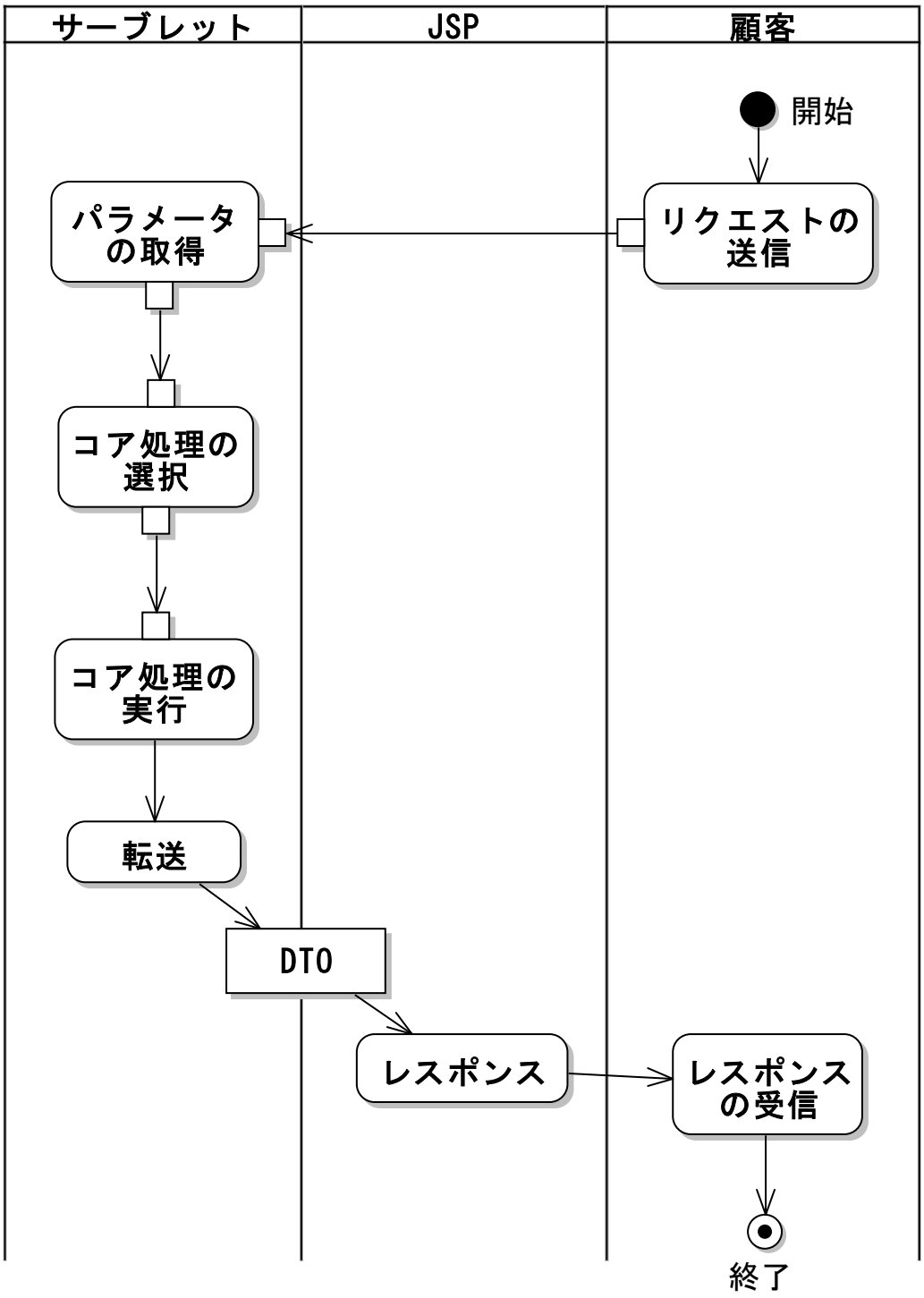
- 分類子の不特定のインスタンスの振る舞い
- 分類子の特定のインスタンスの振る舞い
- 分類子に含まれる部品の振る舞い
- 分類子の属性が特定の値を持った場合の振る舞い
- ドメイン外の要素の振る舞い

区画が階層化されている場合、複数の「下位区画」(subpartition)とそれを含む「上位区画」(superpartition)の関係は、主に次のいずれかの意味を持ちます。

- 集約(aggregation)もしくは合成集約(composition)における、全体のオブジェクトと部分のオブジェクトの関係
- 分類子の属性と特定の属性値の関係

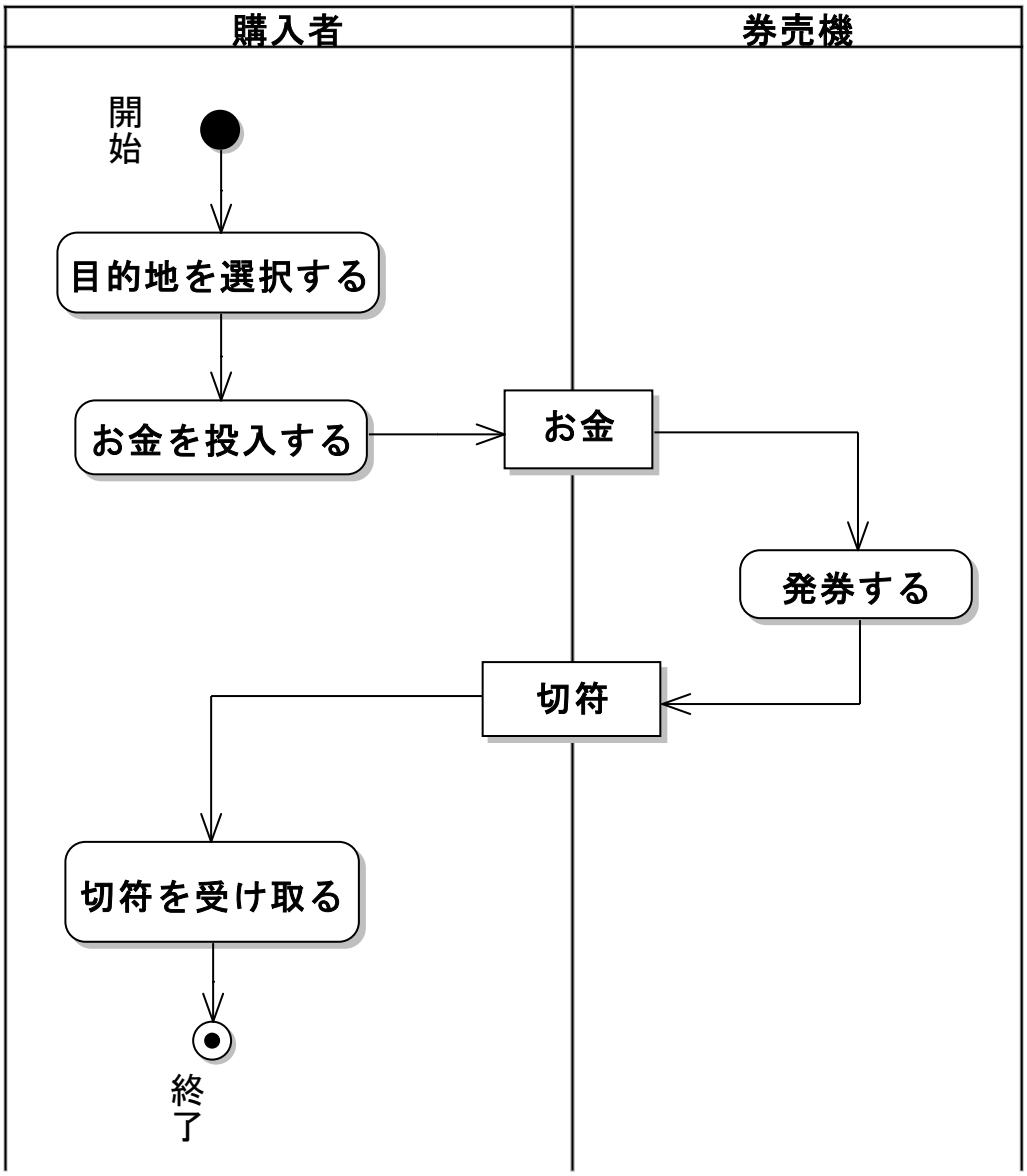
ドメイン外の主体の振る舞いは、区画に「<<external>>」というキーワードを付加します。

<<external>>



■アクティビティパーティションの使用例1

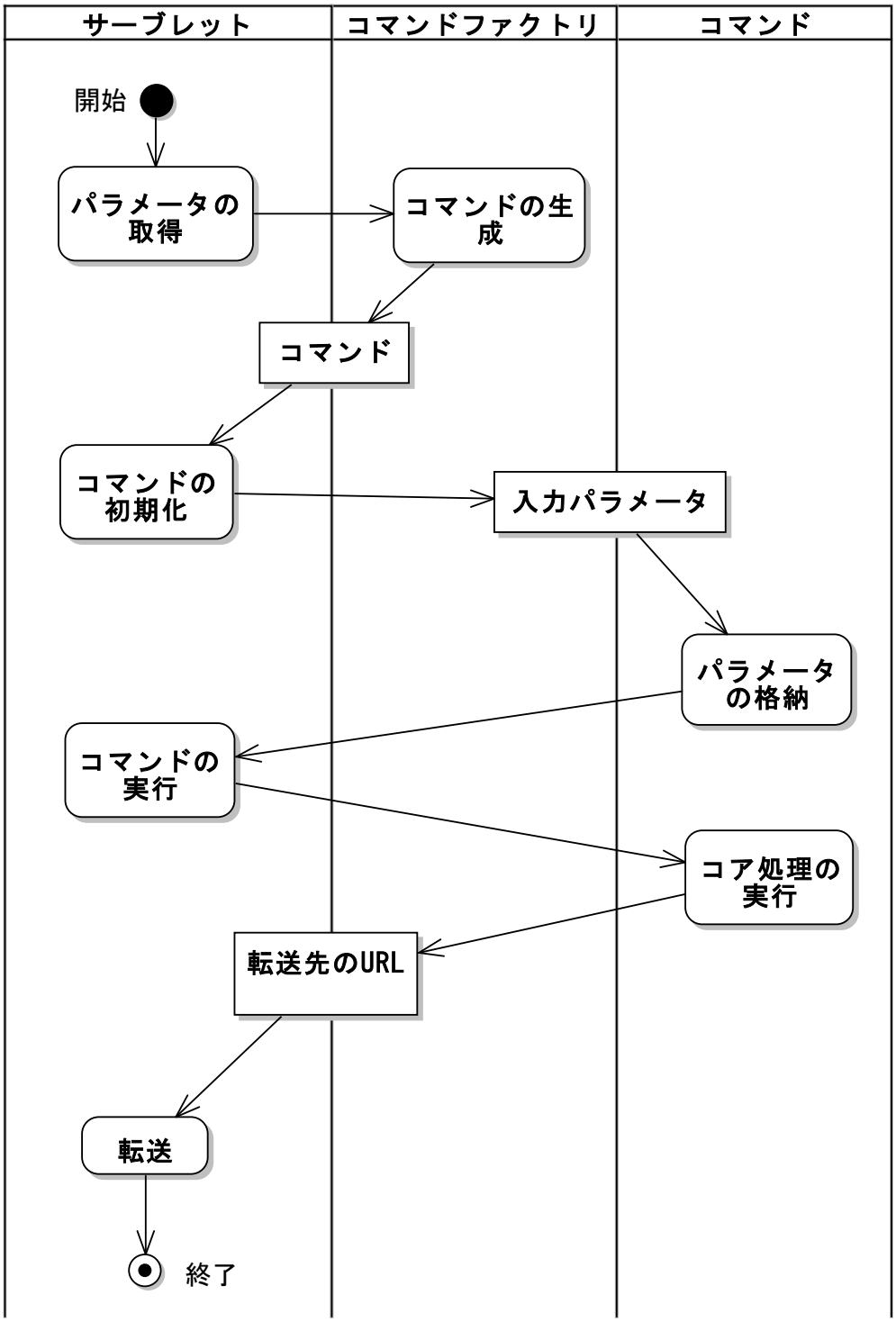
下記の例では、購入者と券売機の間の振る舞いの連鎖を表現しています。



このように区画をまたがったオブジェクトフローの場合、オブジェクトノードを区画の中間に配置します。

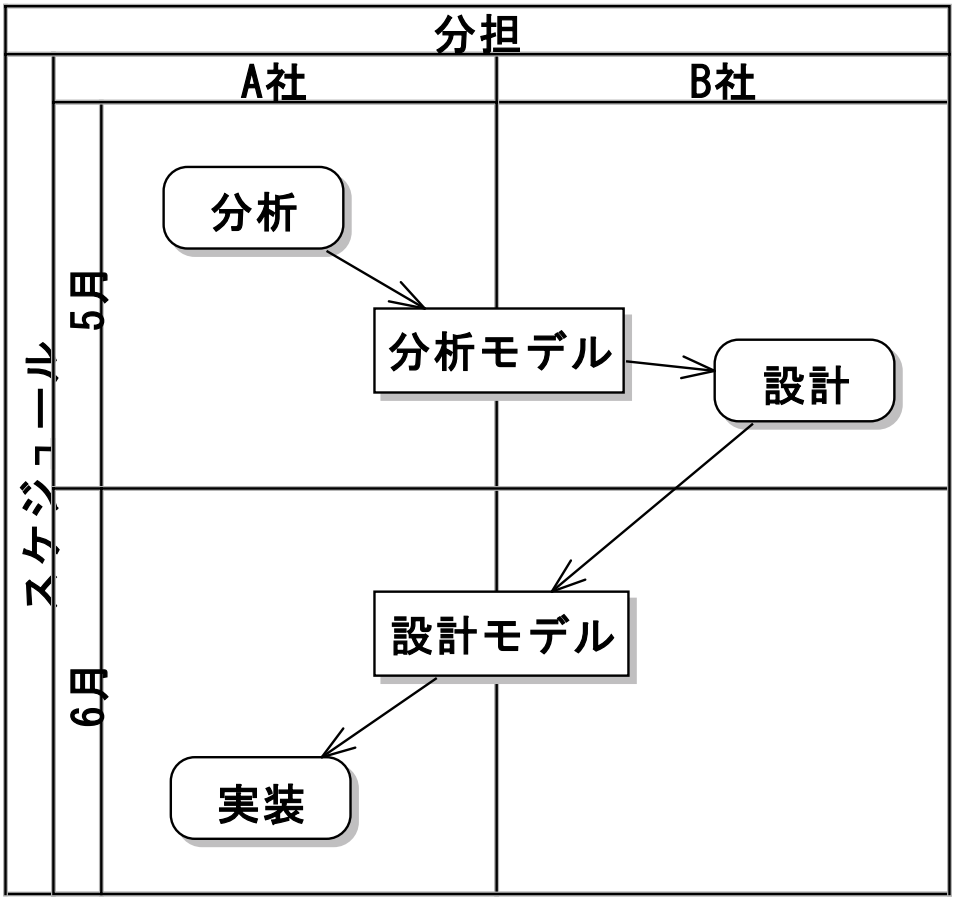
■アクティビティパーティションの使用例2

下記の例では、J2EE パターンの Front Controller を適用した場合の各要素の振る舞いをアクティビティパーティションによって表現しています。


サーブレット
コマンドファクトリ
コマンド
開始
パラメータの取得
コマンドの生成
コマンド
コマンドの初期化
入力パラメータ
パラメータの格納
コマンドの実行
コア処理の実行
転送先のURL
転送
終了

■アクティビティパーティションの多次元的構成

上位区画は、「次元」(dimension)としてとらえることもできます。縦と横にそれぞれ上位区画を作成した場合、二次元からなるアクティビティパーティションが構成できます。



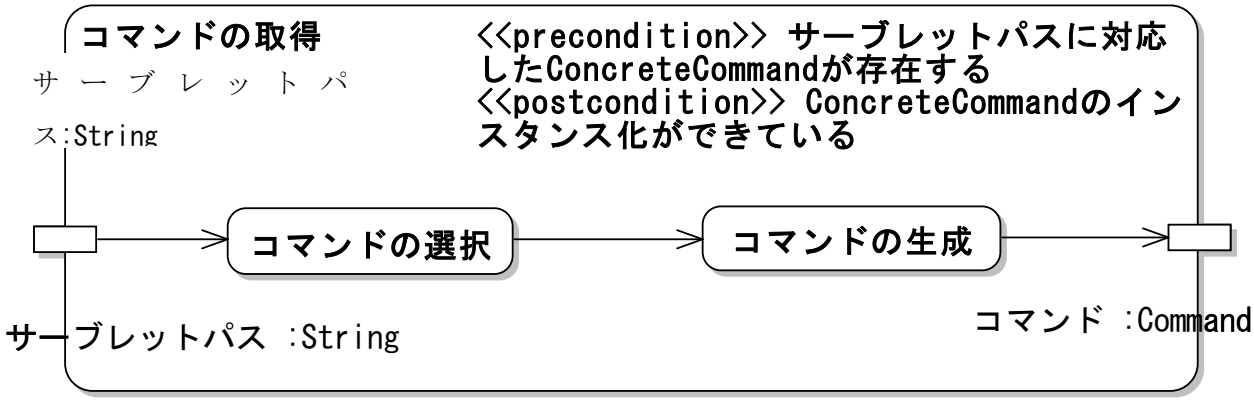
上記の例では、「分担」と「スケジュール」という上位区画が、それぞれ次元を成しています。

17.13 メソッドの引数と戻り値の表現

■アクティビティパラメータノード(ActivityParameterNode)

アクティビティには、引数や戻り値を設定することができます。これにより、アクティビティが引数や戻り値のあるメソッドを表現することができます。

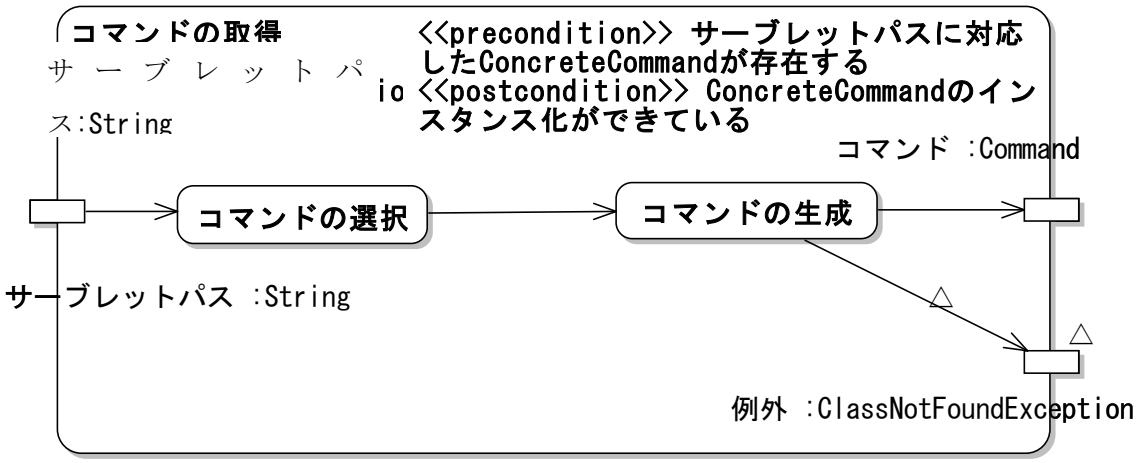
下記は、GoF の Command パターンを適用する際に利用する CommandFactory クラスの getCommand メソッドの振る舞いをアクティビティによって表現したものです。



左上に、アクティビティの名前、パラメータの名前と型を記述します。

右上には、<<precondition>>と<<postcondition>>を記述します。<<precondition>>として、このアクティビティを起動する前に満たしておくべき条件を記述します。<<postcondition>>として、このアクティビティが終了した後に満たしておくべき条件を記述します。

次のように例外の発生を表現することもできます。例外オブジェクトを意味するアクティビティパラメータノードには、「△」のマークを付加します。また、例外オブジェクトへのアクティビティエッジにも、同様に「△」のマークを付加します。



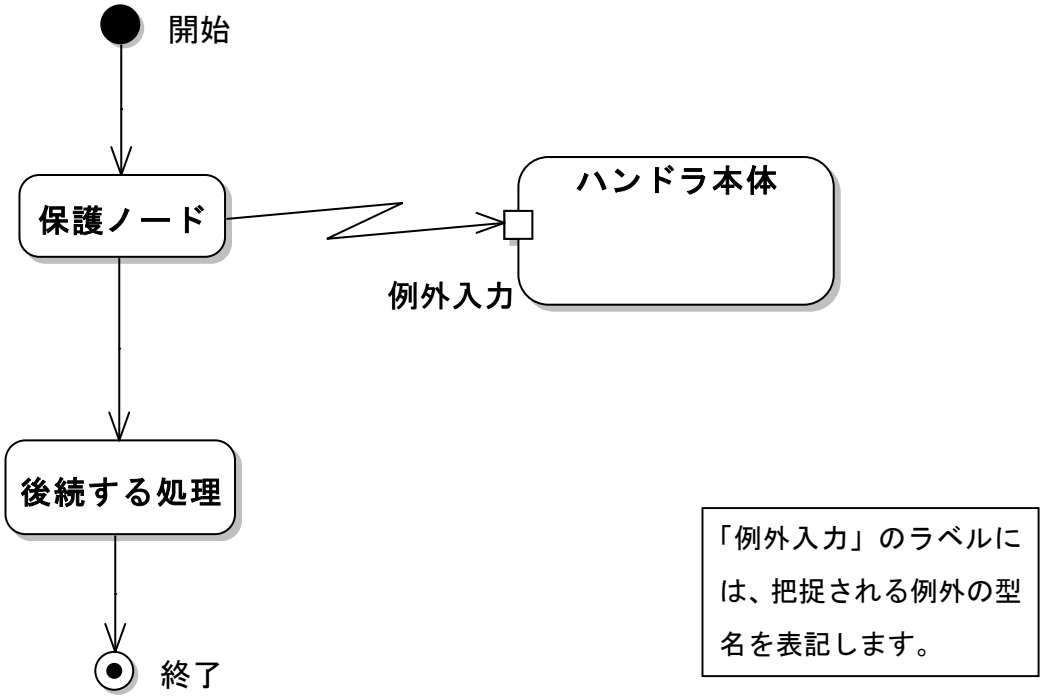
17.14 例外処理の表現

■例外ハンドラの意味と目的

Java などのプログラム言語は、構造化例外処理などの例外処理機構をもっています。一定の処理を実行中に例外が発生した場合、その例外は例外ハンドラによって把捉され回復処理やログ出力が実行されます。

例外の発生とそれに対するハンドラの存在は、「例外ハンドラ」(ExceptionHandler)によって表現することができます。

例外ハンドラの基本的な表記法は以下の通りです。



その実行中に発生した例外が把捉される対象となるアクションが「保護ノード」(protectedNode)です。把捉された例外を受け取り一定の処理を行うのが「ハンドラ本体」(handlerBody)です。「ハンドラ本体」には、オブジェクトノードである「例外入力」(exceptionInput)があり、例外が把捉された場合、「例外トークン」(exception token)が「例外入力」内部に置かれます。「ハンドラ本体」が処理を完了すると、「ハンドラ本体」の「結果トークン」(result token)が「保護ノード」自体の結果トークンとなり、「保護ノード」からの出力エッジに渡されます。

■例外ハンドラを表記する上での注意点

例外ハンドラは、通常のフローとは異なる意味と表記ルールをもつため、以下の点に注意します。

1）例外ハンドラは、稲妻型の線によって表記されるが、この線は入力エッジでも出力エッジでもない。

2）ハンドラ本体には、入力エッジ、出力エッジを作成しない。

3）ハンドラ本体内部の処理に後続する処理は、保護ノードに後続する処理である。

■Java による実装サンプル

```
public class Test {

  public static void main(String[] args) {

      String input = args[0];

      int intValue = 0;

      try {

          intValue = Integer.parseInt(input);

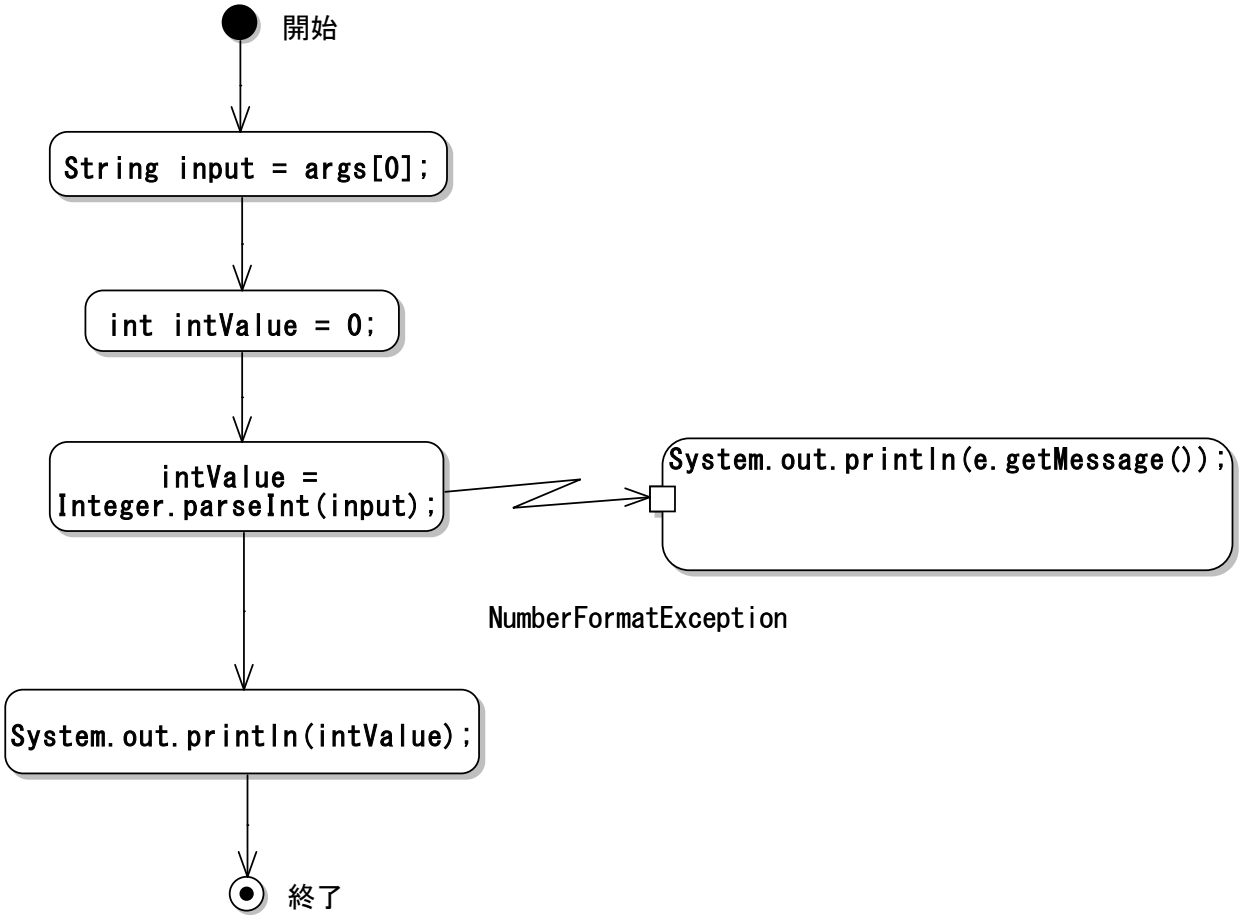
      } catch (NumberFormatException e) {

          System.out.println(e.getMessage());
      }

      System.out.println(intValue);
```

```
    }
}
```

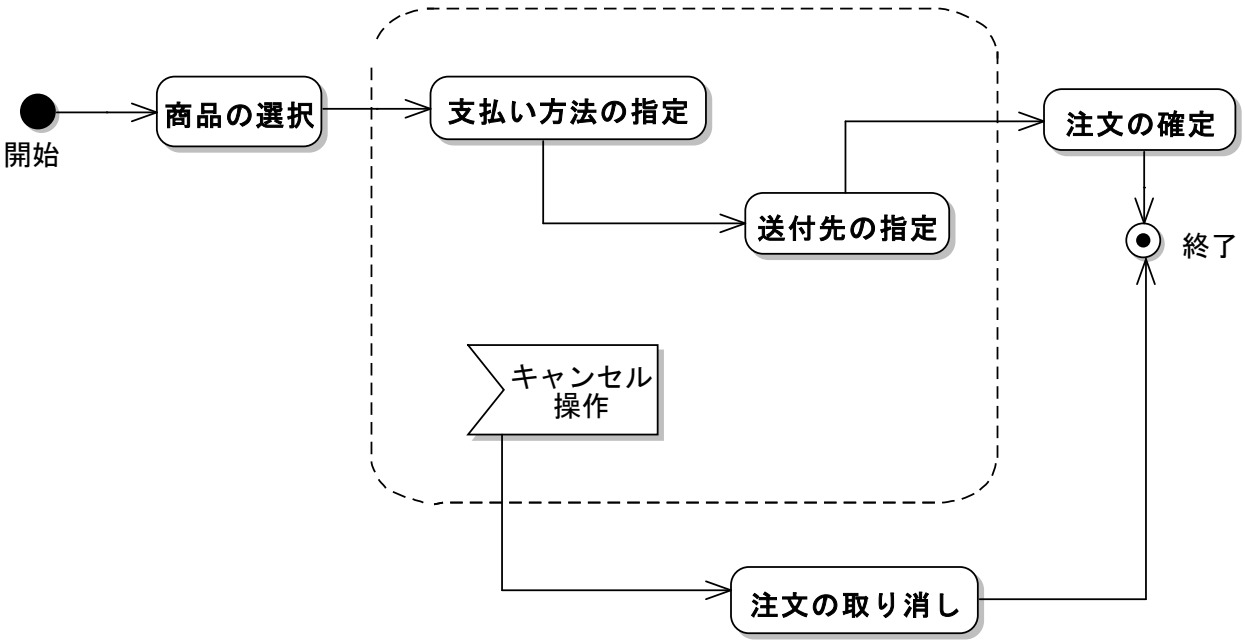
■アクティビティ図による表現



17.15 割り込み処理の表現

■割り込み可能アクティビティ領域(InterruptibleActivityRegion)

一定の処理を実行中に特定のイベントが発生した場合、処理は中断され、イベントのハンドラが実行されるようなケースがあります。そのようなケースを表現するのが「割り込み可能アクティビティ領域」です。これは例外ハンドラとは異なり、正常処理の過程の中で発生した、イベントの受信とハンドラによる処理を表現するものです。



上記の例は、商品選択後、支払い方法の指定、送付先の指定を終了すれば注文が確定するというアクティビティを意味します。しかし、支払い方法の指定、送付先の指定のアクションを行っている際にキャンセル操作が受信された場合は、注文の取り消しアクションを実行します。このように、何かを実行している間に何らかのイベントが発生した場合、その処理を中断して異なる処理を実行するという可能性を表現するのが、割り込み可能アクティビティ領域です。

割り込み可能アクティビティ領域は、破線で囲むことによって表記します。破線内に、割り込まれるアクションとイベントの受信によって割り込むアクションを包含します。受信イベントアクションからは「割り込み可能エッジ」(interruptingEdge)を出力し、破線の領域外のアクションに接続します。このアクションは、受信されたイベントに対するハンドラです。ハンドラの処理に進入した場合は、破

線内の割り込まれるアクションは中断されます。

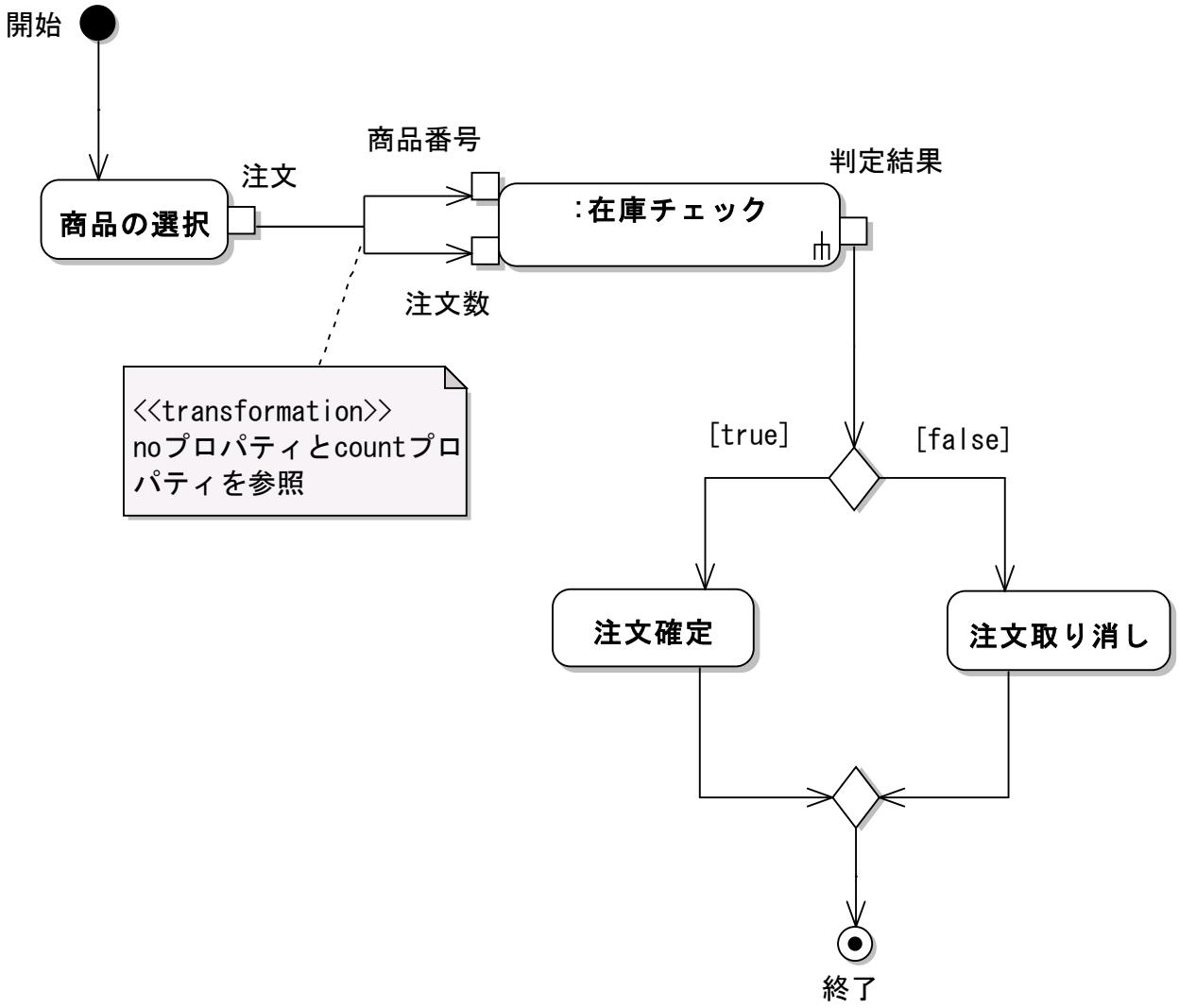
トークンが割り込みエッジを経由して割り込み可能領域を出て行く場合、領域内のすべてのトークンと振る舞いは停止します。

17.16 アクティビティのネストの表現

複雑なビジネスプロセス、ワークフロー、プログラムフローを表現したい場合、全体的な流れを表現する概略的なアクティビティ図と、それに含まれる個々のアクションを拡大して表現した詳細レベルのアクティビティ図の両方が作成できれば、プロセスやフローを正確に把握しやすくなります。そのような、いわばアクティビティのネストを表現するのが、「振る舞い呼び出しアクション」(CallBehaviorAction)です。

■振る舞い呼び出しアクション

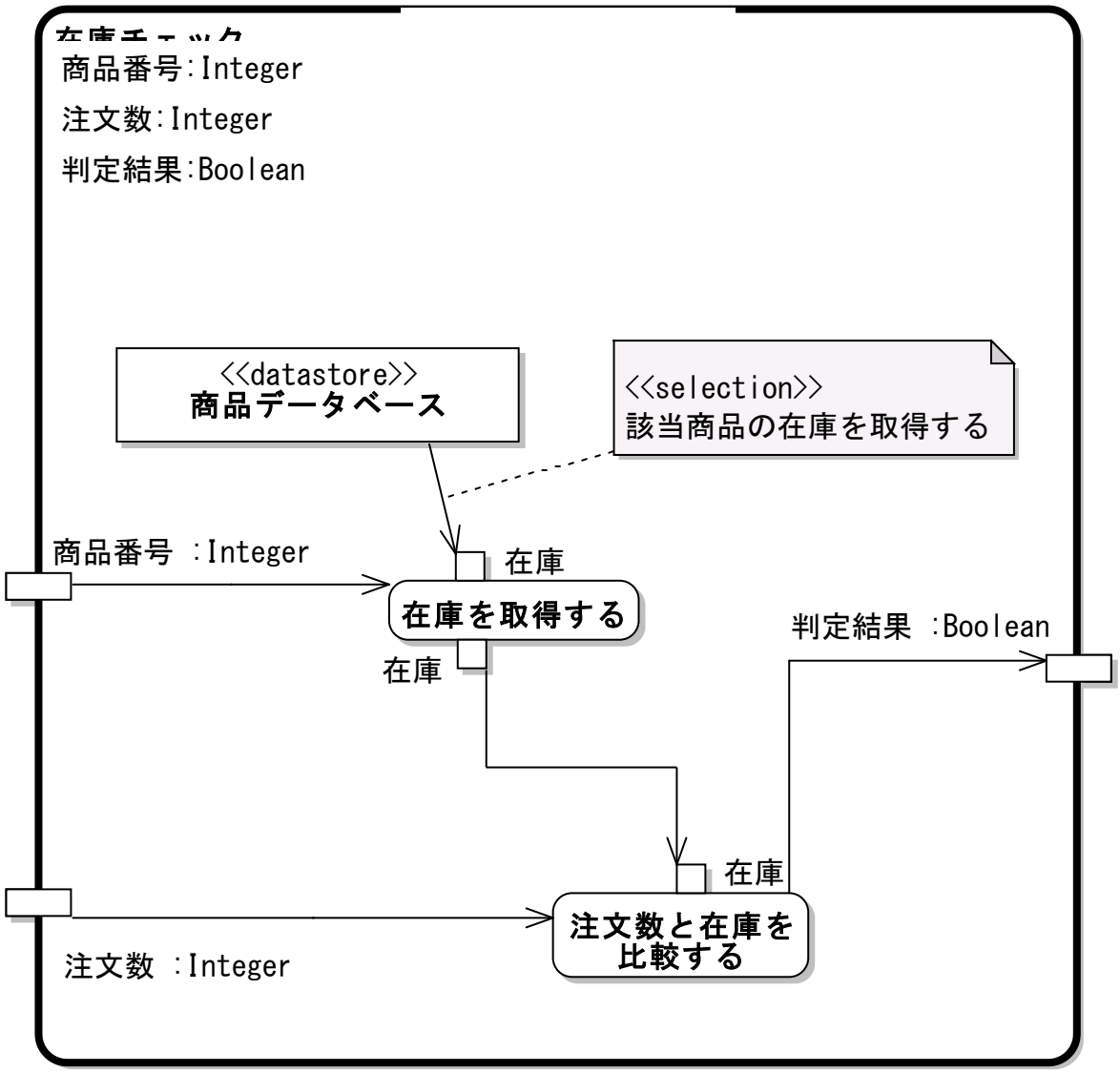
振る舞い呼び出しアクションには、内部に熊手のマークを表記します。



上記＜在庫チェック＞アクションには、次ページの呼び出されるアクションに合わせて、入力ピン

と出力ピンが付加されていますが、概略レベルのアクティビティ図においては、省略したほうが全体の見通しがよくなる場合もあります。

■呼び出されるアクティビティ



呼び出されるアクティビティは、境界線を太字で表記します。

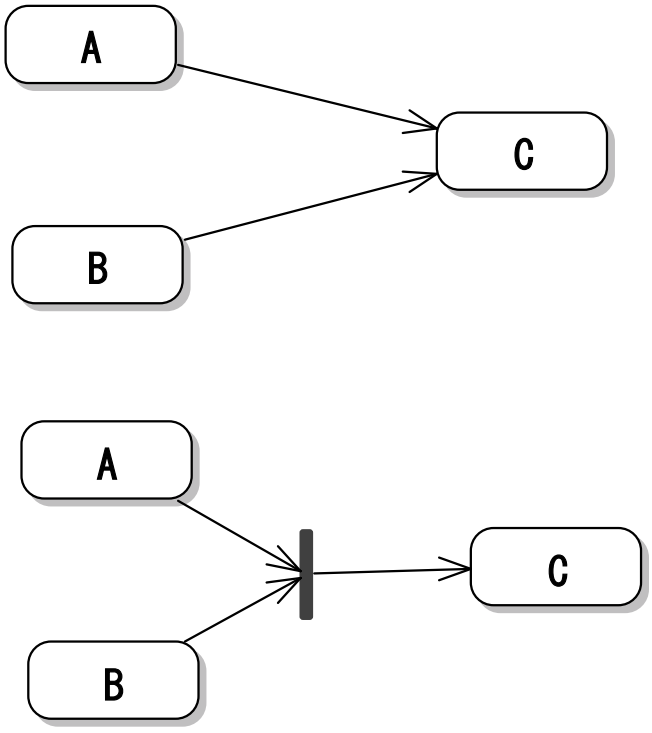
17.17 アクティビティ図に関する注意点

アクションの動作に関して、次のような注意点があります。

■暗黙的ジョイン

アクションノードへの入力エッジが複数存在する場合、暗黙的にジョインノードが存在するものと見なされます。つまり、すべての入力エッジからのトークンの入力があって初めてアクションノードが実行されます。入力エッジが制御フローの場合にも、オブジェクトフローの場合にも当てはまります。

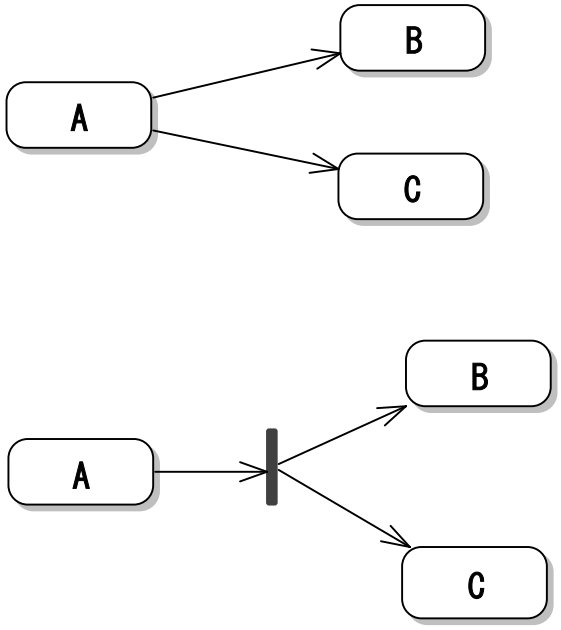
下記の図は同じ事態を意味します。



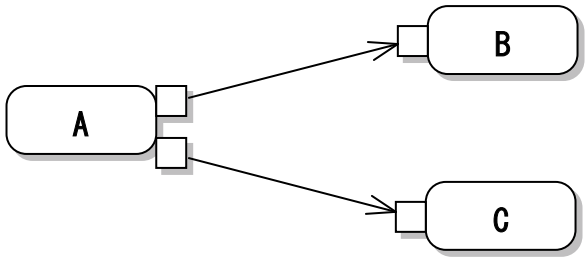
■暗黙的フォーク

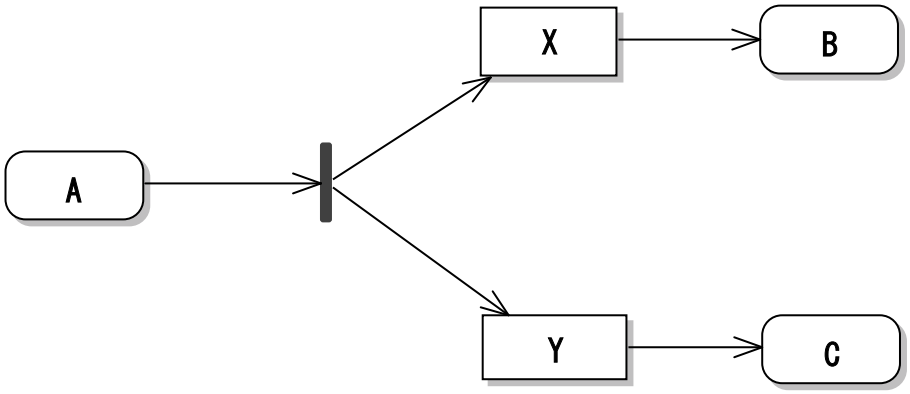
アクションノードからの出力エッジが複数存在する場合、暗黙的にフォークノードが存在するものと見なされます。つまり、すべての出力エッジにトークンがコピーされて渡されます。アクションノードからの出力エッジが制御フローの場合にも、オブジェクトフローの場合にも当てはまります。また、開始ノードに複数の出力エッジが存在する場合も同様の意味です。

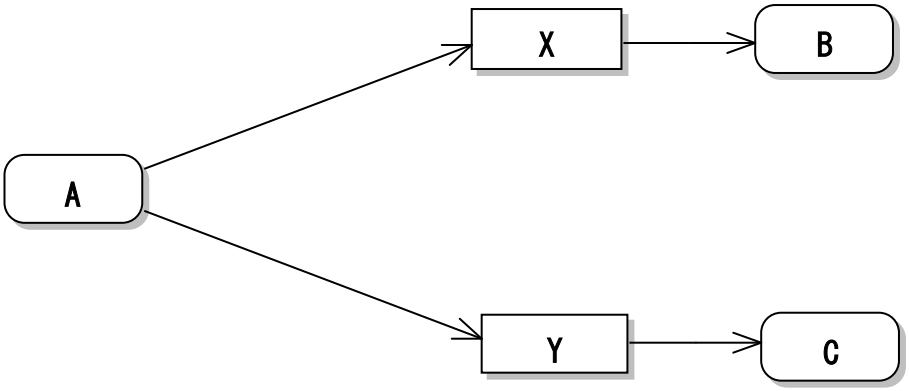
下記の図は同じ事態を意味しています。



オブジェクトフローの場合にも同じ原則が当てはまります。次のようなオブジェクトフローの出力エッジにおいても、暗黙的フォークが適用されます。ただし、この場合、それぞれの出力ピンが、異なるトークン（オブジェクト、データ）を送ることができます。



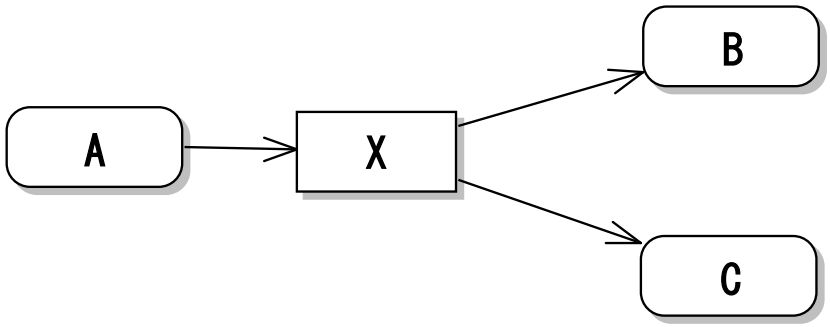


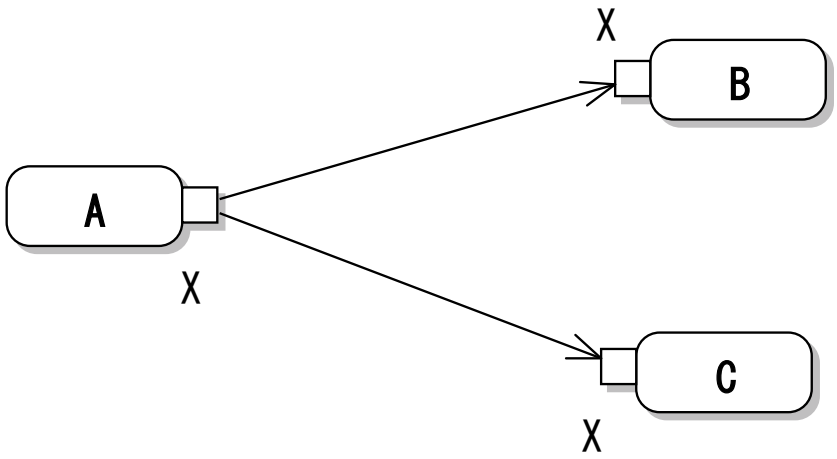


■オブジェクトフローの競合

オブジェクトノードから複数の出力エッジが存在している場合は、トークンはいずれかひとつの出力エッジに対して渡されます。

下記の図の場合、オブジェクトノード X からのトークンを取得できるのは、アクションノード B もしくは C のどちらかになります。





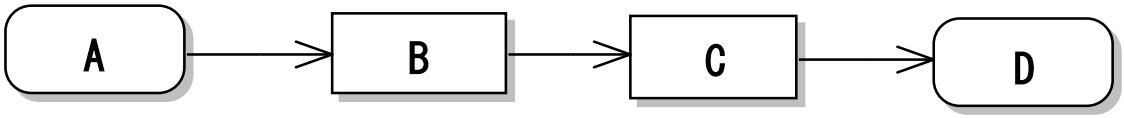
■オブジェクトノードとオブジェクトノードの接続

オブジェクトのノード同士の結合に関して、次のような制約があります。

1）オブジェクトフローで結合されたオブジェクトノード同士は、互換性のある型をもたなければない。特に、下流のオブジェクトノードは上流のオブジェクトノードと同じ型か、上位の型でなければならない。

2）オブジェクトフローで結合されたオブジェクトノード同士は、トークンの数に関して、同じ上限をもたなければならない。

以下のような場合、BとCは同じ型か、CはBの型の上位の型である必要があります。また、BとCは、トークン数の上限値が一致している必要があります。



18クラス図

18.1 クラス図の基本的な記述の仕方

■クラスの属性(attribute)と操作(operation)に関する記法

クラス図は、具象クラス、抽象クラス、インターフェイスを表現することができる図法です。ただし、概念モデル(分析モデル)のクラスも、論理モデル(設計モデル)のクラスも、この図法によって表現することができるため、ソフトウェア詳細設計のアクティビティだけでなく、システム要件定義、ソフトウェア要件定義、システム方式設計、ソフトウェア方式設計といったアクティビティでも利用されることがあります。

クラスは、3 つの区画(コンパートメント:compartment)によって内部が仕切られた長方形によって、表現され、それぞれの区画の内部に、クラス名、属性(attribute)、操作(operation)を定義します。

ひとつひとつの属性は、属性名の他、可視性(visibility)、型(property type)、多重度(multiplicity)、デフォルト値(default)、プロパティモディファイア(property modifier)を指定することができます。

ひとつひとつの操作には、操作名の他、可視性(visibility)、引数名(parameter-name)、引数の型(type-expression)、戻り値の型(return-type)、操作プロパティ(operation property)を指定することができます。

クラス名
可視性 属性名: 型 [多重度] = デフォルト値{プロパティモディファイア}
可視性 操作名(引数名:引数の型):戻り値の型{プロパティ文字列}

※属性の「プロパティ文字列」(property string)には、以下の設定が可能です。複数のプロパティ文字列をコンマで区切って指定することも可能です。

readOnly	読み取り専用の属性であることを意味する。
union	サブクラスの属性が参照する値の和集合であることを意味する。
subsets 属性名	スーパークラスの属性が参照する値の部分集合であることを意味する。
redefines 属性名	スーパークラスの属性の再定義であることを意味する。
ordered	属性によって参照される複数の値が一定の順序を持っていることを意味する。
unique	属性によって参照される複数の値に重複がないことを意味する。

※操作の「プロパティ文字列」には、以下の設定が可能です。複数のプロパティ文字列をコンマで区切って指定することも可能です。

redefines 操作名	スーパークラスの操作の再定義であることを意味する。
query	操作の実行によって、システムの状態は変化しないことを意味する。
ordered	戻り値として返される複数の値が一定の順序を持っていることを意味する。
unique	戻り値として返される複数の値に重複がないことを意味する。

■「可視性」(visibility)の記述の仕方

可視性	意味	対応する Java のアクセス修飾子
－	そのクラスの内部からのみアクセスを許す	private
＋	あらゆる範囲からのアクセスを許す	public

#	そのクラスとサブクラスのみからアクセスを許す	protected
～	そのクラスおよび同じパッケージからのみアクセスを許す	何も記述しない

■派生

派生は、プロパティの値が、なんらかの情報から算出できることを意味し、可視性のあとに、「/」(スラッシュ)記号を付加して表記します。たとえば、次のように表記します。

```
- / sum:int
```

上記のように属性を記述した場合、属性 sum は、その他の何らかの情報から(たとえば、他の属性の値などから)算出できることを意味します。このような属性は読み取り専用の属性であることが一般的です。

■クラスメソッド・クラス変数

クラスメソッドやクラス変数の場合は下線を引きます。

■抽象クラス

抽象クラスのクラス名と抽象メソッドはイタリックで記述します。

■インターフェイス

<<interface>>キーワードを、クラス名の上部に記述して表現します。
※インターフェイスの抽象メソッドはイタリックにしません。

18.2 属性・操作の表記サンプル

■ソースコード

```
class A {

    private int a;

    A(int a) {
        this.a = a;
    }

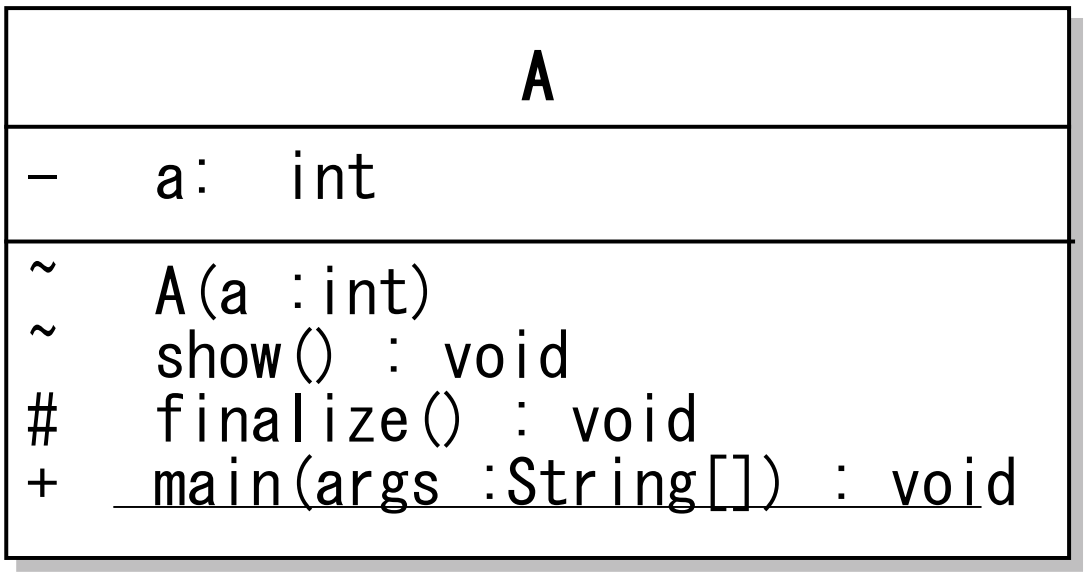
    void show() {
        System.out.println(this.a);
    }

    protected void finalize() throws Throwable {
        System.out.println("A");
        super.finalize();
    }

    public static void main(String[] args) {

        A a = new A(20);
        a.show();
        a = null;
        System.gc();
    }
}
```

■クラス図



18.3 派生属性の表記サンプル

■ソースコード

```
public class Calculator {

    private int x;
    private int y;
    private int result;

    private Calculator(String x, String y) {

        // Integrer.parseIntによってStringをint型に変換する

        this.x = Integer.parseInt(x);

        this.y = Integer.parseInt(y);

    }

    public static Calculator getInstance(String x, String y) {

        return new Calculator(x, y);

    }

    public void plus() {

        this.result = this.x + this.y;

    }

    public void minus() {
```

```
        this.result = this.x - this.y;

    }

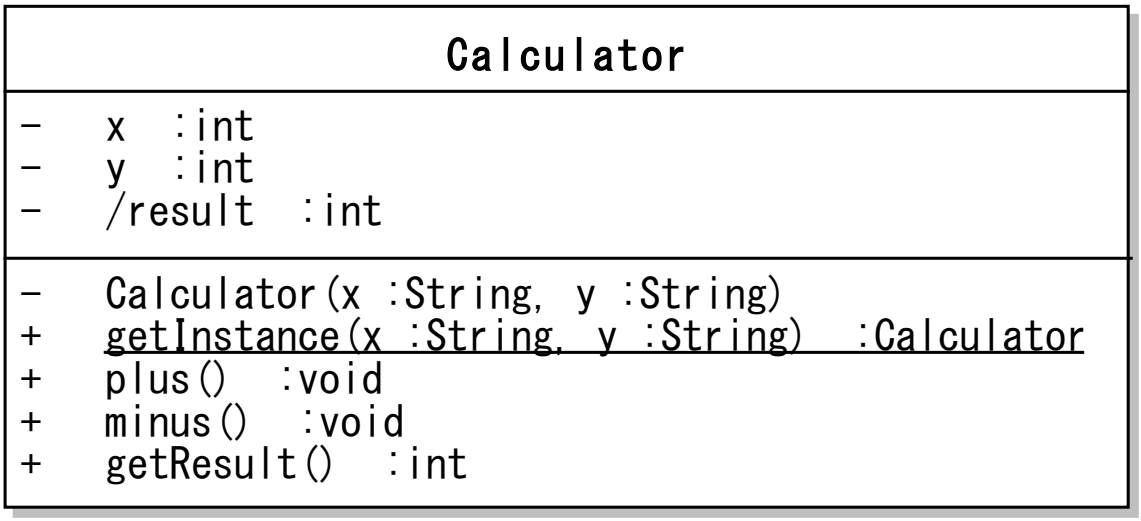
    public int getResult() {

        return this.result;

    }

}
```

■クラス図



18.4 属性のプロパティ文字列 readOnly の設定サンプル

■ソースコード

```
public class Book {
    public static final String ITEM_TYPE = "A001";

    private String title;
    private String author;
    private int price;

    private Book(String title, String author, int price) {

        this.title = title;
        this.author = author;
        this.price = price;
    }

    public static Book newInstance(String title, String author, int price)
{

        Book book = new Book(title, author, price);

        return book;

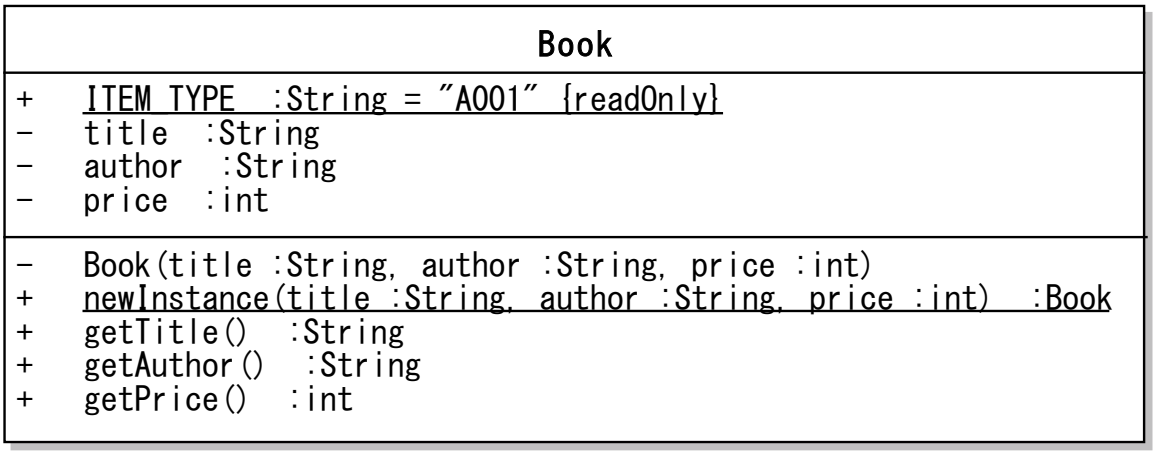
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }
```

```
    public int getPrice() {
        return price;
    }
}
```

■クラス図



18.5 属性のプロパティ文字列 redefines の設定サンプル

■redefines の意味

属性(関連端の場合も同様)の redefines とは、継承された要素の「再定義」を意味しますが、Java におけるフィールドの再定義である「隠蔽」とは意味が異なります。UML2.1 の規定では、再定義は次のような制約にしたがいます。[62]

1) 再定義するプロパティ(属性および関連端)が、再定義されるプロパティ(属性、関連端)の型に適合すること

2) 再定義するプロパティの多重度(指定されている場合)が再定義されるプロパティの多重度の範囲内におさまること

3) 再定義するプロパティのデフォルト値は、再定義されるプロパティのデフォルト値を特化(specialization:汎化の逆)したものでなければならない。

以上を踏まえた場合、「隠蔽」とはまったく異なります。

62 Object Management Group, 2006, "OMG Unified Modeling Language (OMG UML),Superstructure, V2.1", (西原 裕善, 2006, 『UML 2.0 仕様書 2.1 対応』, オーム社), 邦訳 p.325,327 参照

■ソースコード

```
abstract class SQLExecuter {

    public abstract void execute();

}

abstract class DBController {

    private SQLExecuter executer;

    DBController(SQLExecuter executer) {

        this.executer = executer;

    }

    protected abstract void connect();

    public void execute() {

        connect();

        executer.execute();

    }
}
```

```
class OracleController extends DBController {

    OracleController() {

        // スーパークラスのインスタンス変数
        // executerを再定義している

        super(new OracleExecuter());

    }

    protected void connect() {

        System.out.println("Oracleへの接続処理");

    }
}

class OracleExecuter extends SQLExecuter {

    public void execute() {

        System.out.println("Oracleに対する処理を実行する");

    }

}

class Test1 {

    public static void main(String[] args) {

        DBController c = new OracleController();
```

```
            c.execute();

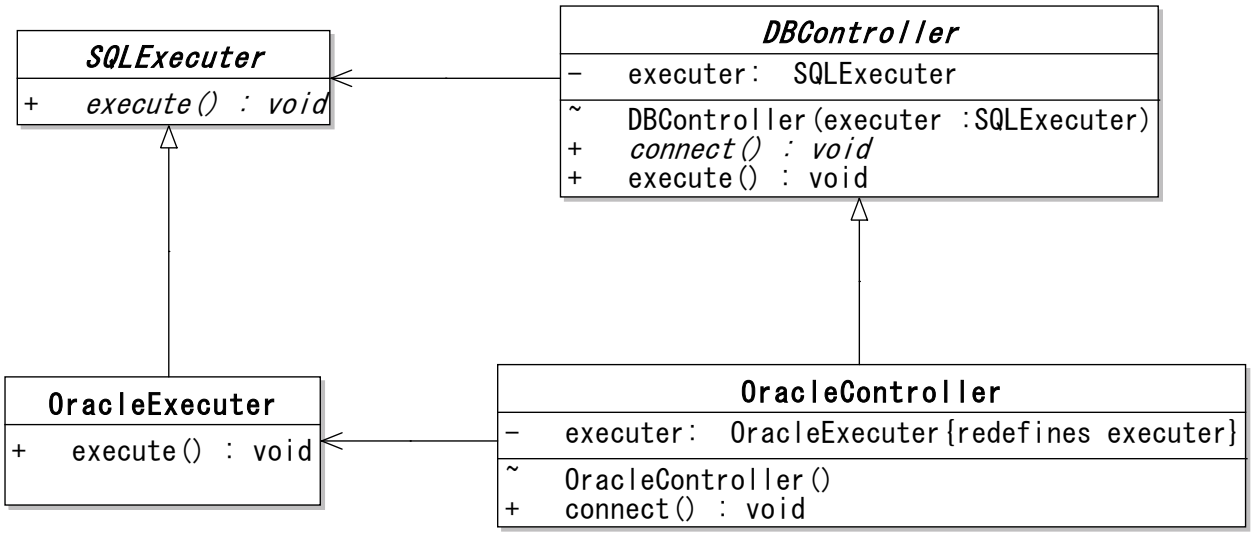
        }

    }
```

実行結果

Oracleへの接続処理
Oracleに対する処理を実行する

■クラス図



18.6 属性のプロパティ文字列 ordered および unique の設定サンプル

■ソースコード

```
import java.util.TreeMap;

class Food {

    private String category;
    private String name;
    private String price;

    public Food(String category, String name, String price) {

        this.category = category;
        this.name = name;
        this.price = price;

    }

    public String getCategory() {
        return category;
    }

    public String getName() {
        return name;
    }

    public String getPrice() {
        return price;
    }

}
```

```
class Fruit extends Food {

    private int price;

    public Fruit(String category, String name, String price) {

        super(category, name, "");

        this.price = Integer.parseInt(price);

    }
}

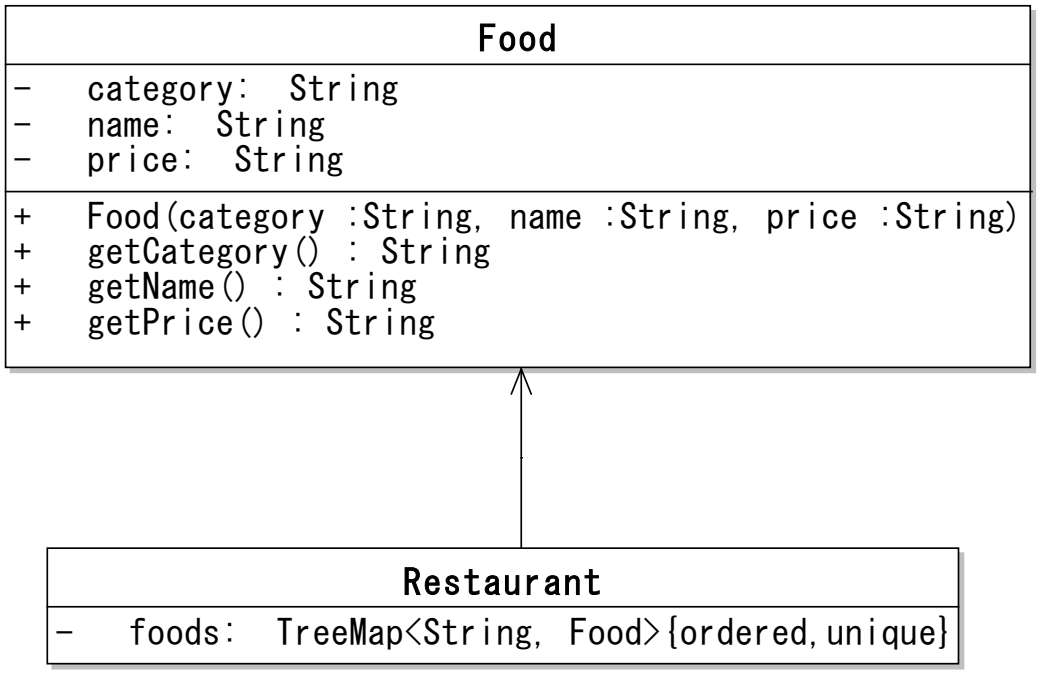
class Restaurant{

    private TreeMap<String,Food> foods = new TreeMap<String,Food>();

    // 以下省略

}
```

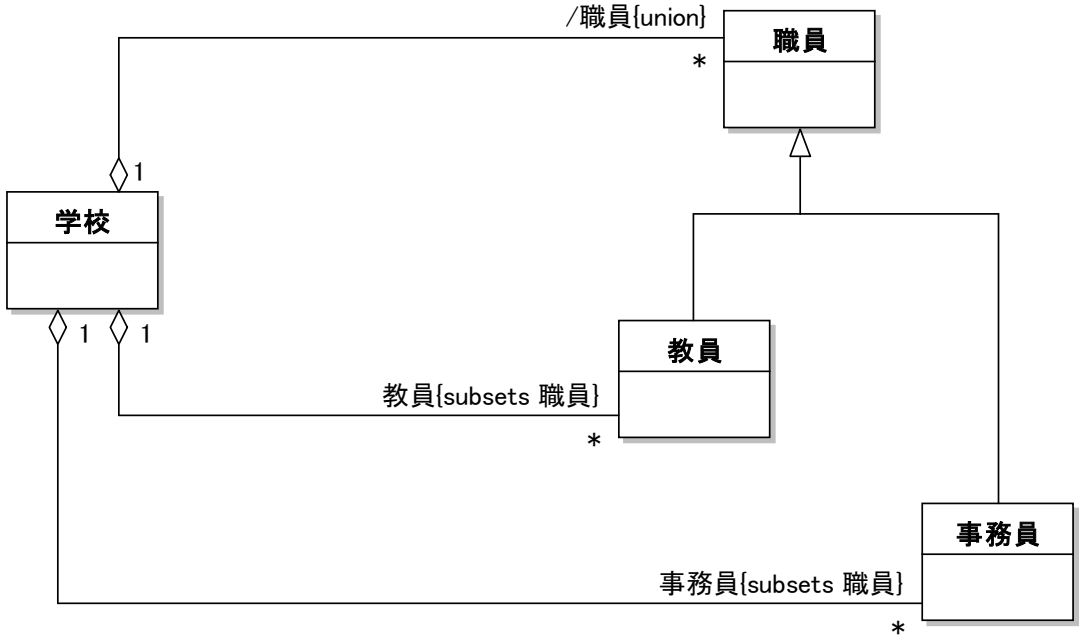
■クラス図


Food
- category: String
- name: String
- price: String
+ Food(category :String, name :String, price :String)
+ getCategory() : String
+ getName() : String
+ getPrice() : String
Restaurant
- foods: TreeMap<String, Food>{ordered,unique}

18.7 属性のプロパティ文字列 union と subsets の設定サンプル

union は、その属性の値が複数の部分集合からなる和集合であることを意味し、subsets は、全体集合を表現する他の属性に対して部分集合であることを意味します。これらは組み合わせて使用することができます。

下図では、学校は、教員と事務員をそれぞれ雇用する結果、学校が管理する職員としては、教員の集合と事務員の集合からなる和集合によって構成されることを意味します。



このように、ある関連が、複数の関連の集合を合成した値（インスタンス）によって構成されるような場合に、この表現方法を用います。

18.8 操作のプロパティ文字列 redefines の設定サンプル

■ソースコード

```
public class Food {

    private String category;
    private String name;
    private String price;

    public Food(String category, String name, String price) {

        this.category = category;
        this.name = name;
        this.price = price;

    }

    public String getCategory() {
        return category;
    }

    public String getName() {
        return name;
    }

    public String getPrice() {
        return price;
    }

}
```

```
class Fruit extends Food {

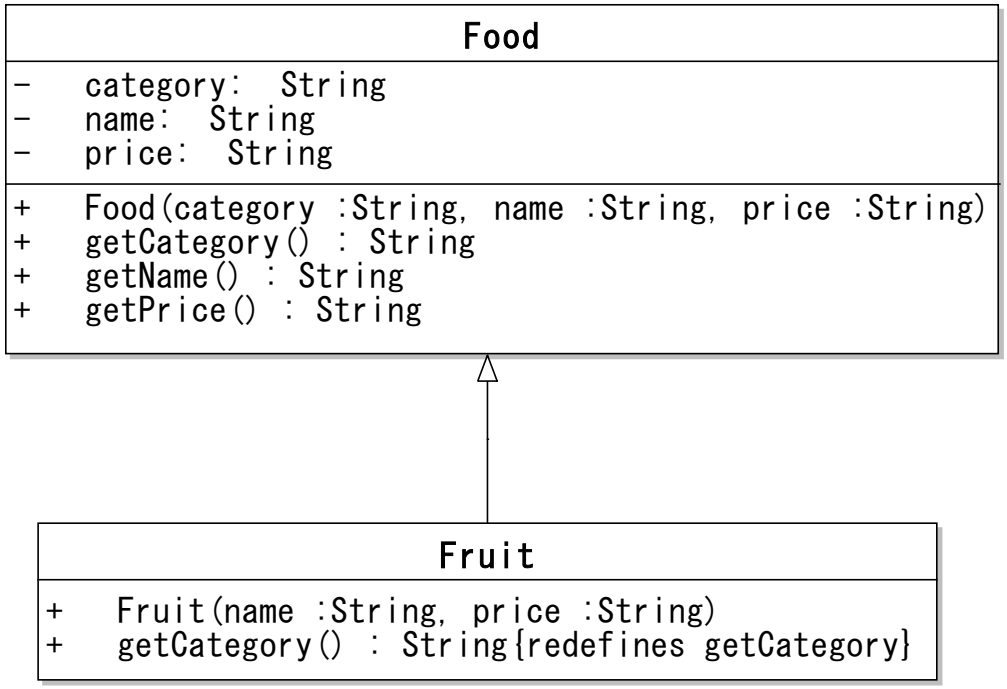
    public Fruit(String name, String price) {

        super("", name, price);

    }

    @Override
    public String getCategory() {
        return "";
    }
}
```

■クラス図



18.9 操作のプロパティ文字列 ordered および unique の設定サンプル

■ソースコード

```
import java.util.TreeMap;

class Food {

    private String category;
    private String name;
    private String price;

    public Food(String category, String name, String price) {

        this.category = category;
        this.name = name;
        this.price = price;

    }

    public String getCategory() {
        return category;
    }

    public String getName() {
        return name;
    }

    public String getPrice() {
        return price;
    }

}
```

```
class Fruit extends Food {

    private int price;

    public Fruit(String category, String name, String price) {

        super(category, name, "");

        this.price = Integer.parseInt(price);

    }
}

class Restaurant {

    private TreeMap<String, Food> foods = new TreeMap<String, Food>();

    public TreeMap<String, Food> getFoods(){

        return foods;

    }

    // 以下省略

}
```

18.10 オブジェクトコンポジションと関連

クライアントとサプライアの関係において、特に、クライアントがフィールドを通して、サプライアを利用している場合、その関係を、UML の用語では「関連」(association)といいます。次のふたつのクラスは「関連」の関係にあります。つまり、オブジェクト指向設計における「オブジェクトコンポジション」の設計を行ったクライアントとサプライアの間の関係は、UML では「関連」と呼ばれます。

```
class Product {

    // インスタンス変数

    private String no; // 商品番号
    private String name; // 商品名
    private int price; // 価格
    private int count; // 販売個数
    private int stock; // 在庫

    // コンストラクタで、商品番号、商品名、価格を初期化する

    public Product(String no, String name, int price) {

        this.no = no;
        this.name = name;
        this.price = price;

    }

    public String getName() {
        return this.name;
    }

    public int getStock() {
        return this.stock;
    }
}
```

```
class ProductViewer {

    private Product product;

    public void setProduct(Product p) {

        this.product = p;

    }

    public void showProduct() {

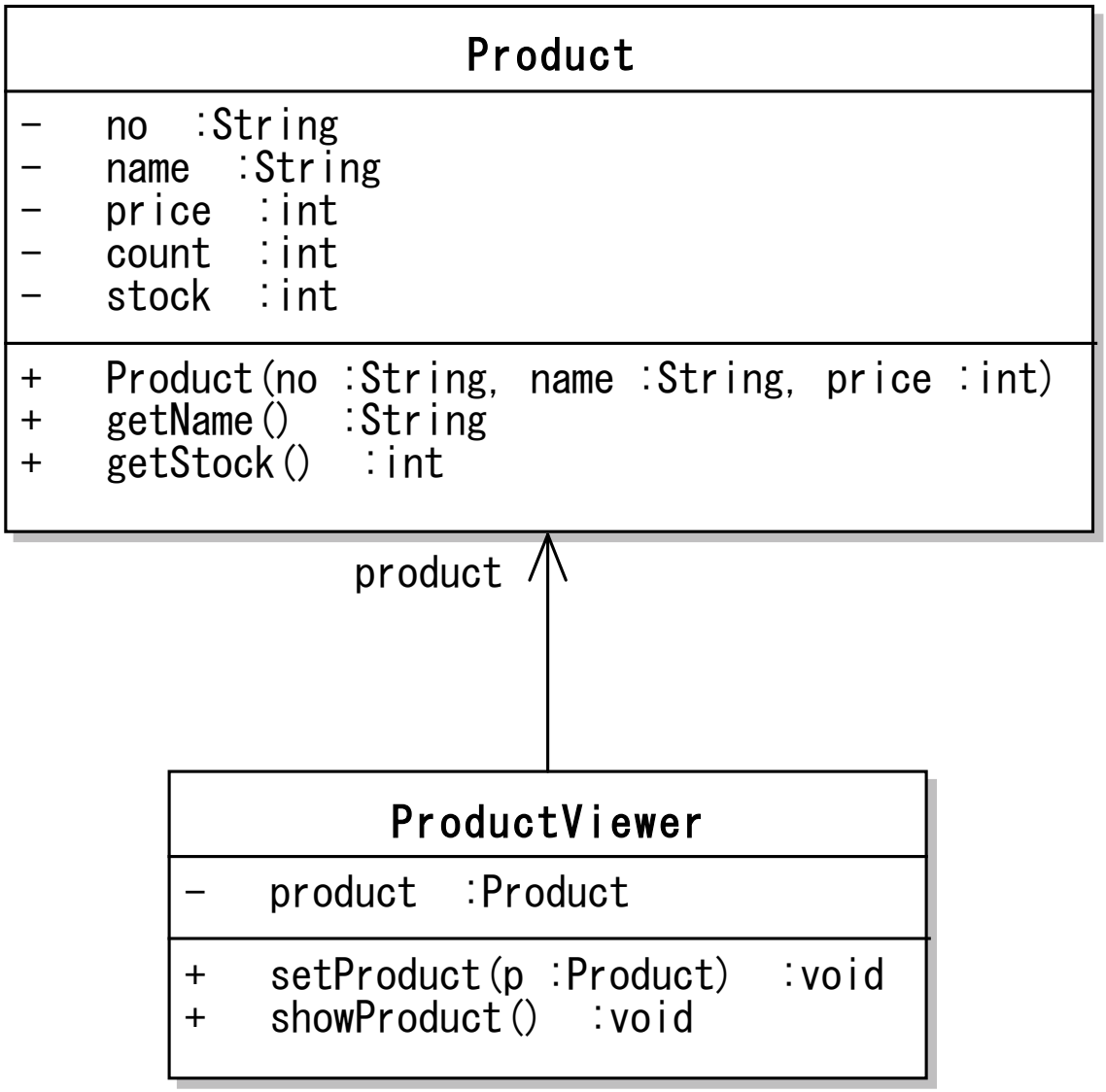
        // 商品名、在庫を出力する

        System.out.println(this.product.getName() + "の在庫＝"
                + this.product.getStock());

    }
}
```

上記の例では、ProductViewer クラスが Product クラスのクライアントです。ProductViewer クラスが Product クラスのインスタンス変数やインスタンスメソッドに対して何らかのアクセスを行います。この場合、ProductViewer クラスから Product クラスへの「誘導可能性」(navigability)があるといいます。「誘導可能性」も UML の用語です。

「関連」は、次のように、実線と誘導可能性の矢印によって表現されます。

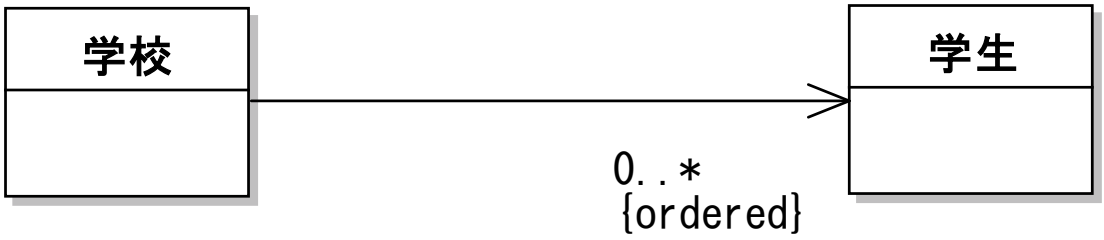


関連を示す実線の Product クラス側の端に記述されているものは、「関連端」(association end)もしくは「関連端名」(name of the association end)と呼ばれるもので、そのクラスの(相手に対する)役割を意味します。この場合の関連端「product」は、サプライア Product クラスの ProductViewer クラスに対する役割を意味します。逆に、ProductViewer クラス側に、Product に対する役割を「関連端」として記述することも可能です(両方に記述することも可能)。また、関連端は省略可能です。

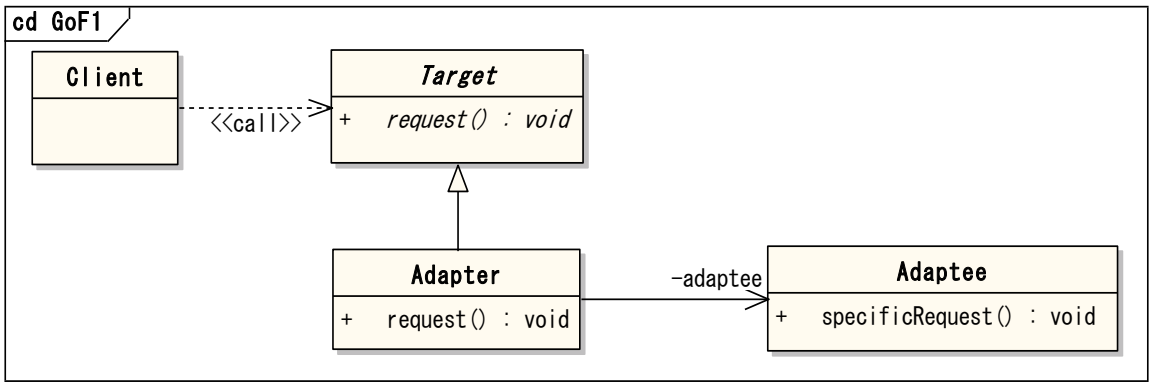
さらに、関連端には、以下のような「プロパティ文字列」(property string)を設定することができます。

プロパティ文字列	意味
subsets 関連端名もしくは属性名	該当する関連端もしくは属性の部分集合であることを意味する。
redefines 関連端名	該当する関連端の再定義であることを意味する。
union	複数の部分集合の和集合であることを意味する。
ordered	一定の基準で順序づけられた集合（要素に重複はない）であることを意味する。
bag	要素に重複をもったコレクションであることを意味する。
sequence もしくは seq	一定の基準で順序づけられた、要素に重複をもったコレクションであることを意味する。

たとえば、次のような場合、学校は 0 以上の学生のインスタンスに関連するが、学生のインスタンスは一定の仕方で順序づけられていることを意味します。

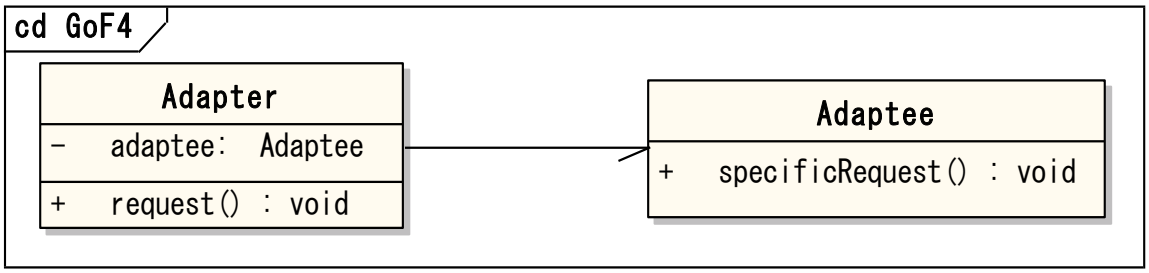


18.11「関連端」による表現



UML2.0 では、関連、集約、合成集約の接続線の端に「関連端」を設定することができます。これは、一般的には、インスタンス変数の名前になります。上記の例では、Adapter クラスの中に、Adaptee 型の adaptee というインスタンス変数があることを意味します。この場合、Adapter に属性の記述を省くことができます。一般的には、合成集約の場合、属性ではなく関連端で表すことが多いようです。

上記の関連端による表現は、以下のように記述することもできます。



18.12 集約

クライアントがフィールドによってサプライアを参照する関係が「関連」(association)ですが、とくに、クライアントが全体、サプライアが部分という関係が存在する場合、これを「集約」(aggregation)といいます。この全体-部分の関係は、ソースコードだけで決定されるものではなく、作成者の解釈に依存します。

```
class Student {

    // インスタンス変数
    private String no; // 学籍番号
    private String name; // 氏名
    private int point; // 得点

    // コンストラクタ
    // 学籍番号、氏名を初期化する

    public Student(String no, String name) {

        this.no = no;
        this.name = name;

    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPoint() {
        return point;
    }
```

```
    public void setPoint(int point) {
        this.point = point;
    }

    // インスタンスメソッド
    // 得点を加算する

    public void add(int point) {

        // 得点を加算する

        this.point += point;

    }
}

class School {

    private Student student;

    public void setStudent(Student s) {

        this.student = s;

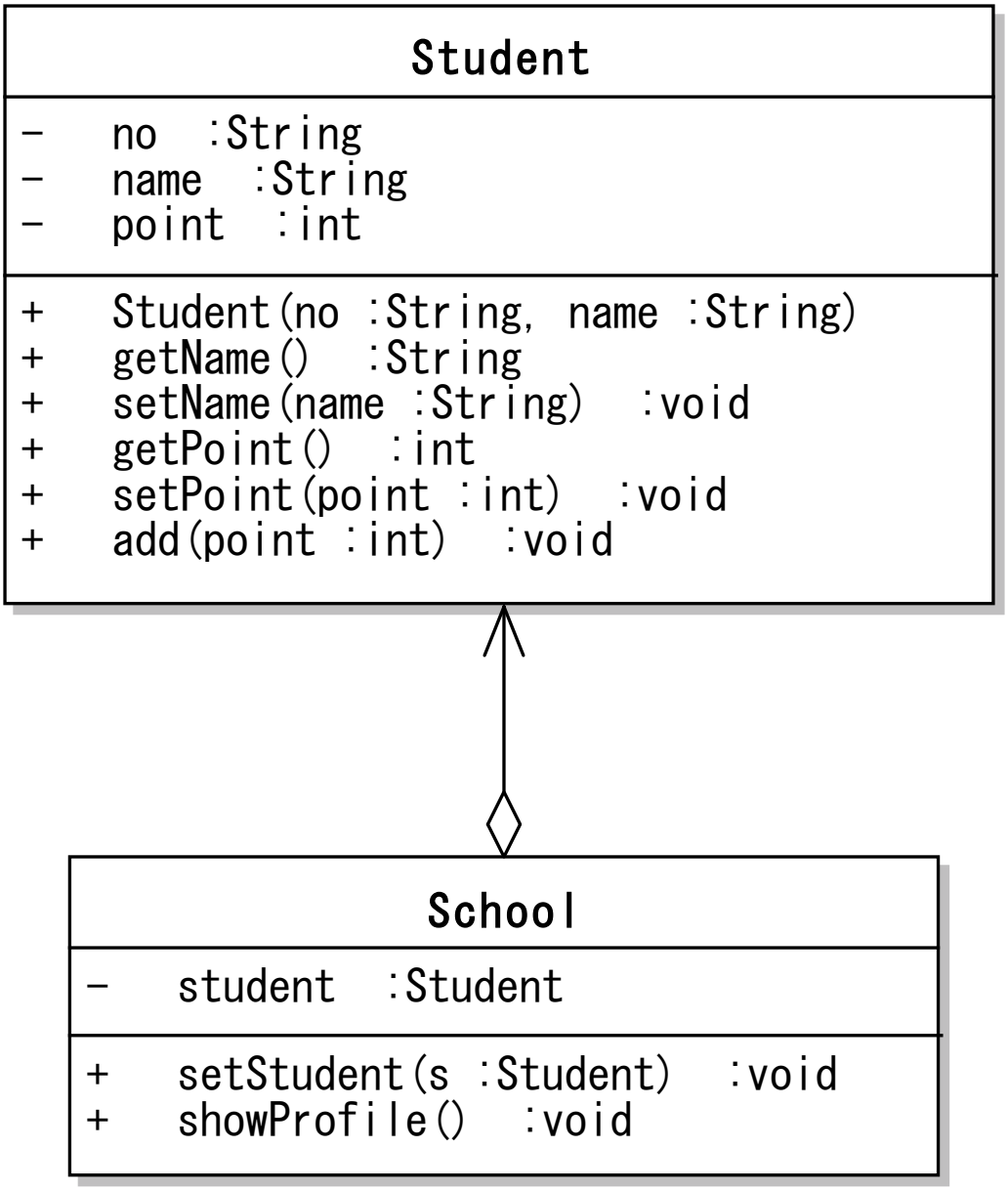
    }

    // インスタンスメソッド
    // 学生の氏名と得点を表示する

    public void showProfile() {

        System.out.println(this.student.getName() + "の得点="
                + this.student.getPoint());
```

```
    }
}
```



集約は、白抜きのひし形を、実線の全体側の端に付加して表現します。

18.13 合成集約

クライアントがフィールドによってサプライアと関係し、さらに、その間に全体-部分の関係がある場合は、「集約」ですが、さらに、クライアントのインスタンスの破棄が同時にサプライアのインスタンスの破棄を生起し、かつ、ひとつのサプライアのインスタンスが、ひとつのクライアントのインスタンスによってのみ利用される場合、「合成集約」(composition)といいます。ただし、サプライアのインスタンスを生成する順序や方法については特に規定されていません(UML 2.1)。また、クライアントである全体側のインスタンスを破棄する前に、サプライアである部分のインスタンスを全体から取り除く(クライアントの参照から外す)ことができます。その場合は、合成集約ではなくなるため、クライアントである全体側のインスタンスを破棄してもサプライアである部分側のインスタンスは保持されます。

```
class Computer {

    private Cpu cpu;

    public Computer() {

        // クライアントの内部で、サプライアをインスタンス化する

        this.cpu = new Cpu("Core 2 Duo E8500", 3.0);

    }

    public void operation() {

        this.cpu.work();

    }
}
```

上記のソースコードの特徴は、クライアントが、サプライアのインスタンスを自己のコンストラクタの内部で生成しています。クライアントに対して、外部からサプライアのインスタンスをセットするという仕方ではありません。つまり、サプライアのインスタンスは、このクライアントの内部でのみ利用されています。これにより、クライアントのインスタンスが破棄された場合、同時にサプライアのインスタンスも破棄されます。

```
class Cpu {

    private double clock; // クロック周波数
    private String name; // 製品名

    public Cpu(String n, double c) {

        this.name = n;
        this.clock = c;

    }

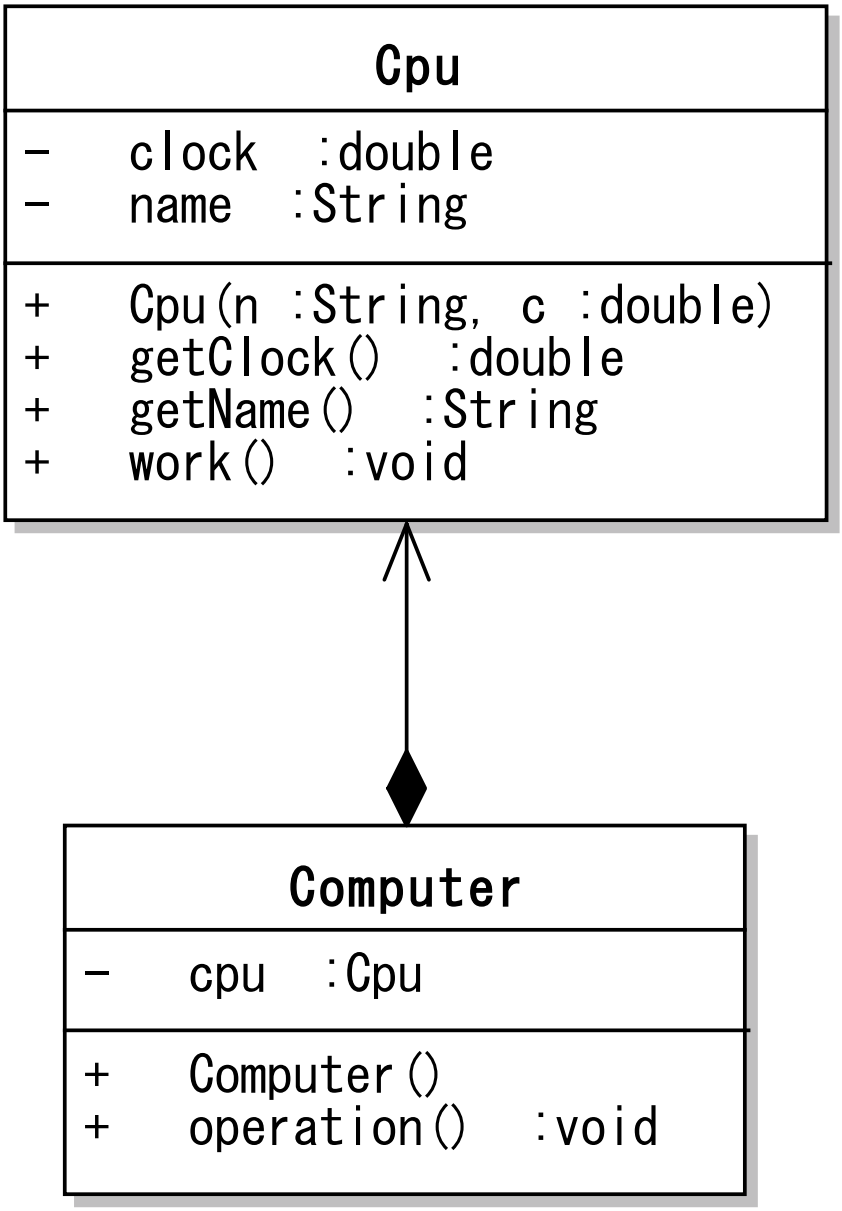
    public double getClock() {
        return clock;
    }

    public String getName() {
        return name;
    }

    public void work() {

        // 一定の処理を行う

    }
}
```



合成集約は、黒いひし形を、実線の全体側の端に付加して表現します。

18.14 汎化

Java における「継承」(inheritance)においては、サブクラスがスーパークラスを継承します。このとき、サブクラスがスーパークラスを継承によって利用しているため、サブクラスがクライアント、スーパークラスがサプライアになります。「継承」は UML の用語では「汎化」(generalization)といいます。ただし、継承とはスーパークラスとサブクラスの関係の捉え方が逆になり、スーパークラスがサブクラスを「汎化」すると捉えます。「汎化」とは、「一般化する」という意味で、上位概念が下位概念を汎化するという関係を意味します。継承は、サブクラスがスーパークラスを「継承する」という捉え方をします。

```
// スーパークラス
class Product {

    // インスタンス変数

    private String no; // 商品番号
    private String name; // 商品名
    private int price; // 価格
    private int count; // 販売個数

    public Product() {
    }

    public String getNo() {
        return no;
    }

    public void setNo(String no) {
        this.no = no;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    public int getCount() {
        return count;
    }

    public void setCount(int count) {
        this.count = count;
    }

    // 売上計算を行うインスタンスメソッド
    // 販売個数と価格の乗算結果から、
    // 売上金額を計算し、戻り値として返す

    public int calculateSales() {

        // 売上金額を計算する

        int sales = price * count;

        // 戻り値として返す

        return sales;
    }
```

```
    public void show() {

        System.out.println(this.no + "¥t" + this.name + "¥t"
            + calculateSales());
    }
}
```

```
// サブクラス
class Printer extends Product {

    private int printSpeed; // 1分あたりの印刷枚数

    public int getPrintSpeed() {
        return printSpeed;
    }

    public void setPrintSpeed(int printSpeed) {
        this.printSpeed = printSpeed;
    }

    public void showPrinter() {

        System.out.println(getNo() + "¥t" + getName() + "¥t"
            + calculateSales() + "¥t" + this.printSpeed);

    }
}
// サブクラス
class Computer extends Product {

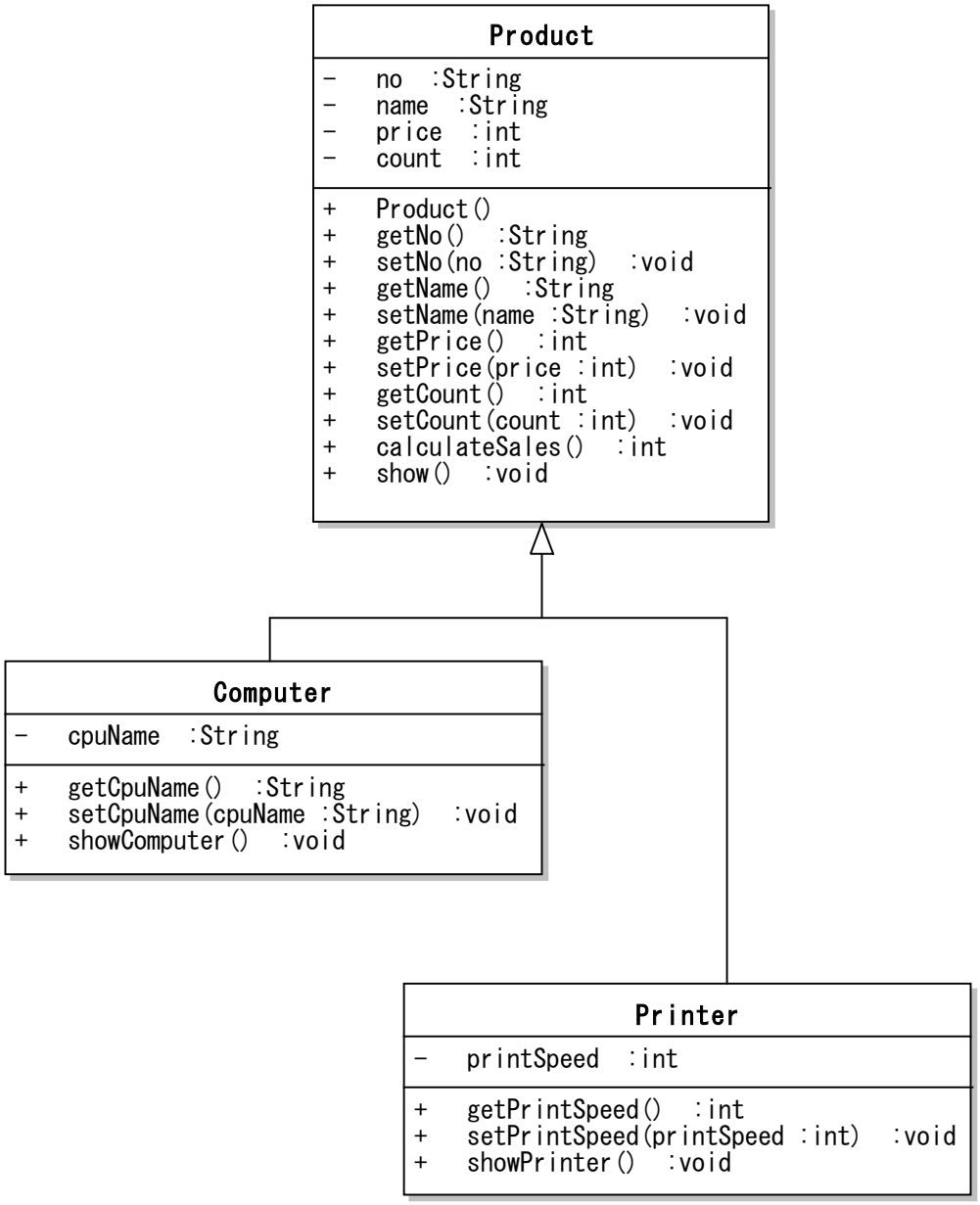
    private String cpuName; // 搭載するCPUの名称

    public String getCpuName() {
        return cpuName;
    }

    public void setCpuName(String cpuName) {
        this.cpuName = cpuName;
    }

    public void showComputer() {
```

```
            System.out.println(getNo() + "¥t" + getName() + "¥t"
                + calculateSales() + "¥t" + this.cpuName);
        }
    }
```



汎化は、スーパークラス(サプライア)とサブクラス(クライアント)を実線で結び、スーパークラス側の端に白抜きの三角形を付加して表現します。

18.15 抽象クラス

抽象クラスは、クラス名をイタリックで、抽象メソッドもイタリックで表現します。

```
//抽象クラス

abstract class Student {

    private String no;// 学籍番号
    private String name;// 氏名
    private int unit;// 取得単位数

    public Student(String no, String name) {

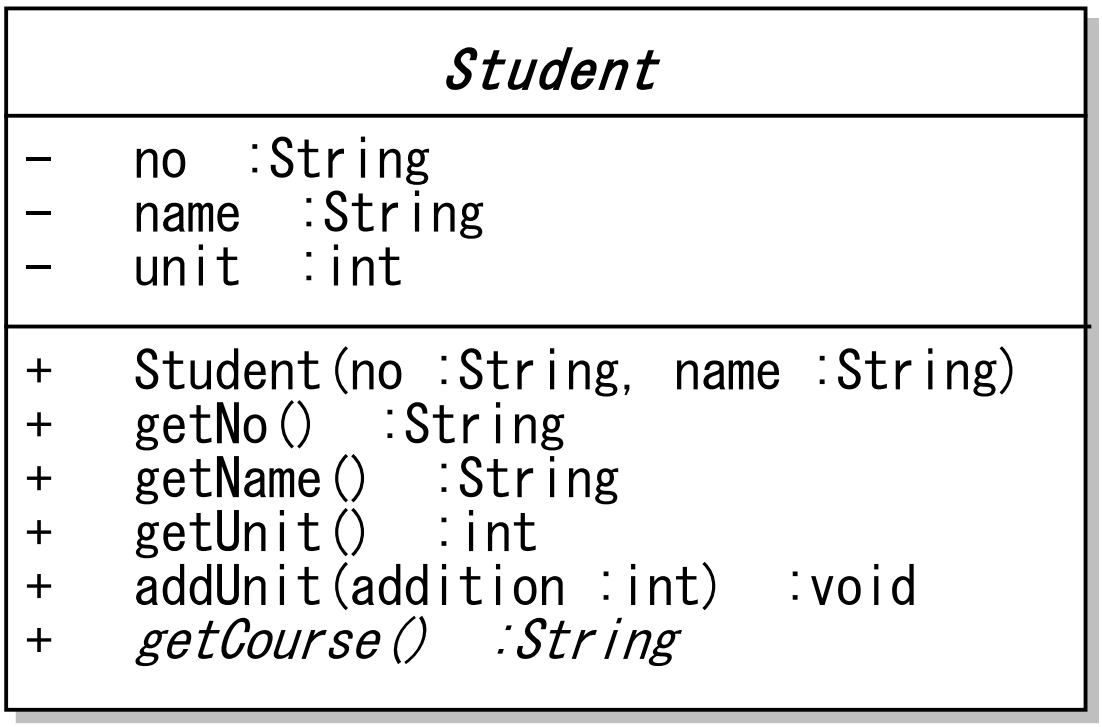
        this.no = no;
        this.name = name;
    }

    public String getNo() {
        return no;
    }
    public String getName() {
        return name;
    }
    public int getUnit() {
        return unit;
    }
    public void addUnit(int addition) {
        this.unit += addition;
    }

    // 抽象メソッド

    public abstract String getCourse();
```

}


Student
- no :String
- name :String
- unit :int
+ Student(no :String, name :String)
+ getNo() :String
+ getName() :String
+ getUnit() :int
+ addUnit(addition :int) :void
+ getCourse() :String

18.16 インターフェイス

インターフェイスは、インターフェイス名の上に、キーワード「<<interface>>」を記述します。また、抽象クラスと異なり、インターフェイス名や抽象メソッドをイタリックで表記しません。

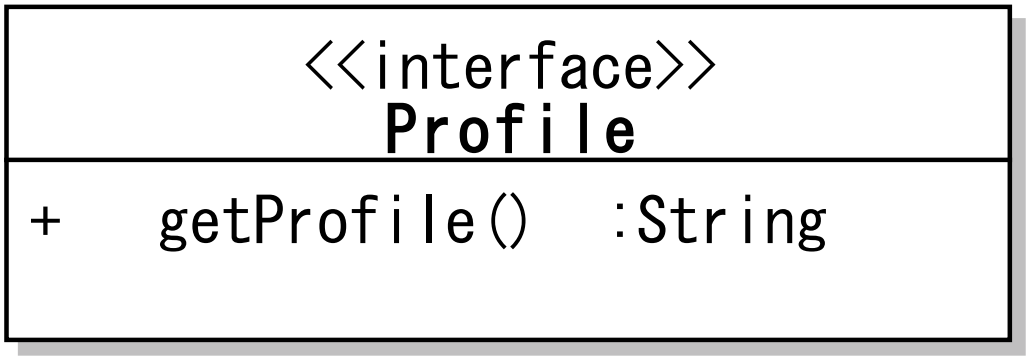
```
// インターフェイス

interface Profile {

    // 抽象メソッド

    String getProfile();

}
```



18.17 実現

Javaの用語では、インターフェイスの仕様を満たすクラスを作成することを「実装」(implimentation)と言いますが、UML では、「実現」(realization)といいます。それは、インターフェイスをメソッドに関する一定の仕様、規格と捉え、その仕様にしたがったメソッドを含むクラスを作成することを、仕様の「実現」と捉えるからです。

```
// インターフェイス（クライアント）

interface Profile {

    // 抽象メソッド

    String getProfile();

}

// サプライア（学生）（インターフェイスにとってはクライアント）

class Student implements Profile {

    private String no;// 学籍番号
    private String firstName;// 名
    private String lastName;// 姓
    private int unit;// 取得単位数

    public Student(String no, String firstName, String lastName) {

        this.no = no;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getNo() {
```

```
        return no;
    }

    public String getFirstName() {

        return firstName;
    }

    public String getLastName() {

        return lastName;
    }

    public int getUnit() {

        return unit;
    }

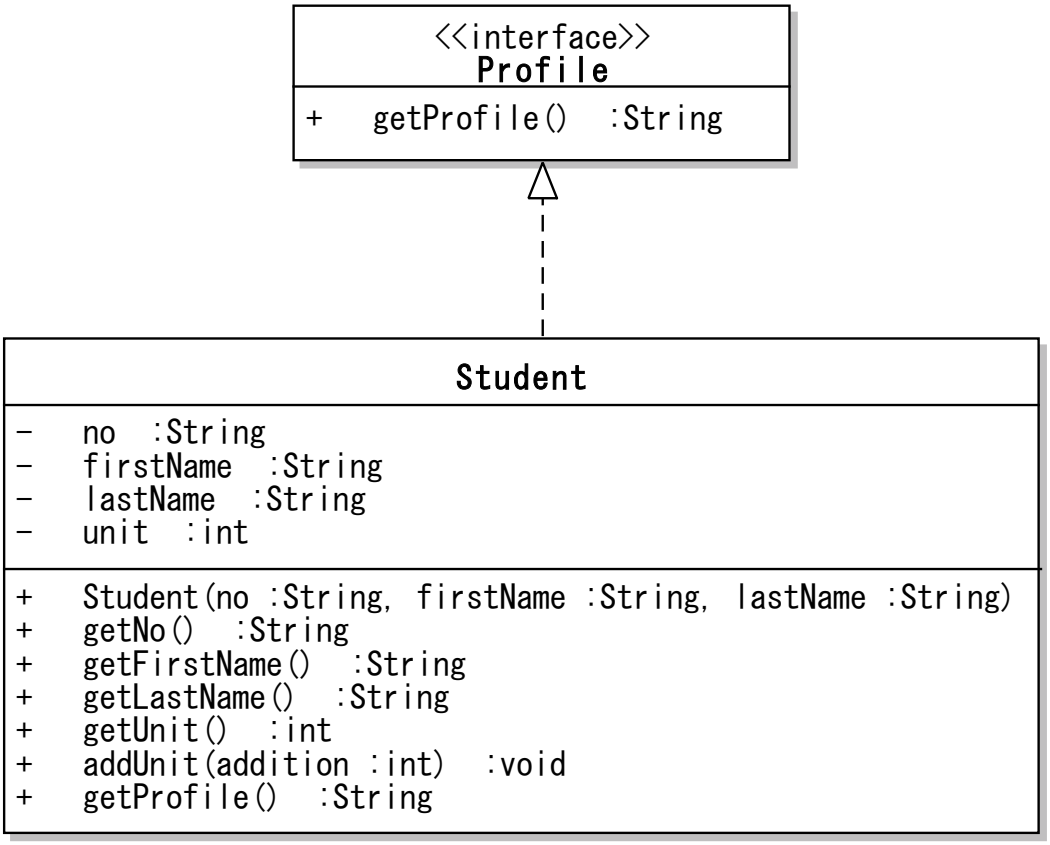
    public void addUnit(int addition){

        this.unit += addition;
    }

    public String getProfile(){

        return getNo() + "-" + getFirstName() + "-" + getLastName();
    }
}
```

「実現」は、白抜き三角形と点線で表現します。



18.18 クライアントとサプライア

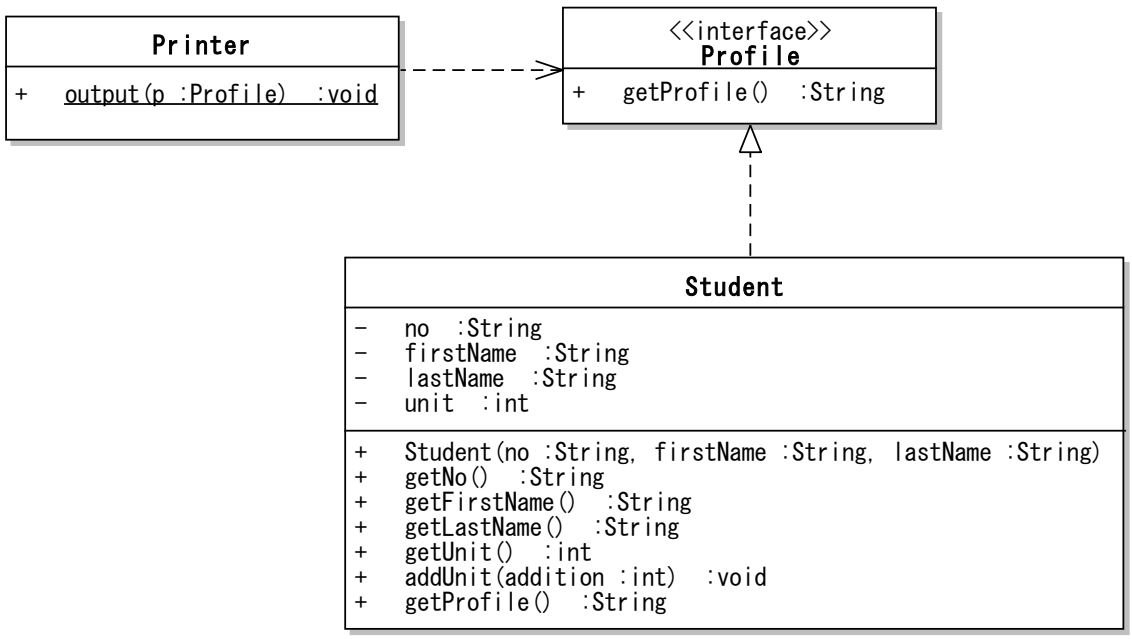
次のように、クライアントも同時に表現することもできます。

```
// クライアント

class Printer{

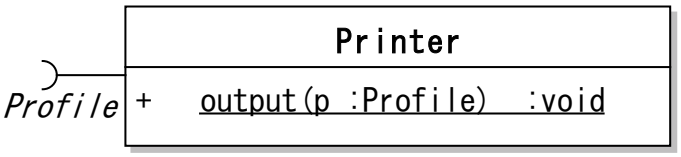
    public static void output(Profile p){
        System.out.println(p.getProfile());
    }

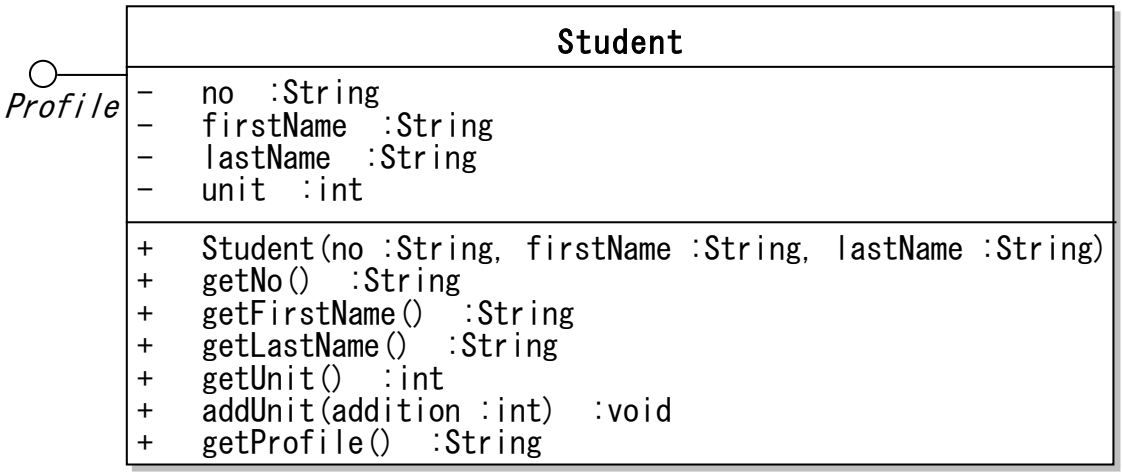
}
```



18.19 ロリポップを用いた表現

クライアントとサプライアの間に存在するインターフェイスを、「要求インターフェイス」(required interface)と「提供インターフェイス」(provided interface)によって表現することもできます。





18.20 依存

クライアントがサプライアを利用しているという関係を、UML の用語では、「依存」(dependency)といいます。クライアントがサプライアに「依存」します。次のふたつのクラスは「依存」の関係にあります。

```
class Product {

    // インスタンス変数

    private String no; // 商品番号
    private String name; // 商品名
    private int price; // 価格
    private int count; // 販売個数
    private int stock; // 在庫

    // コンストラクタで、商品番号、商品名、価格を初期化する

    public Product(String no, String name, int price) {

        this.no = no;
        this.name = name;
        this.price = price;

    }

    public String getName() {
        return this.name;
    }

    public int getStock() {
        return this.stock;
    }
}
```

```
class ProductViewer {

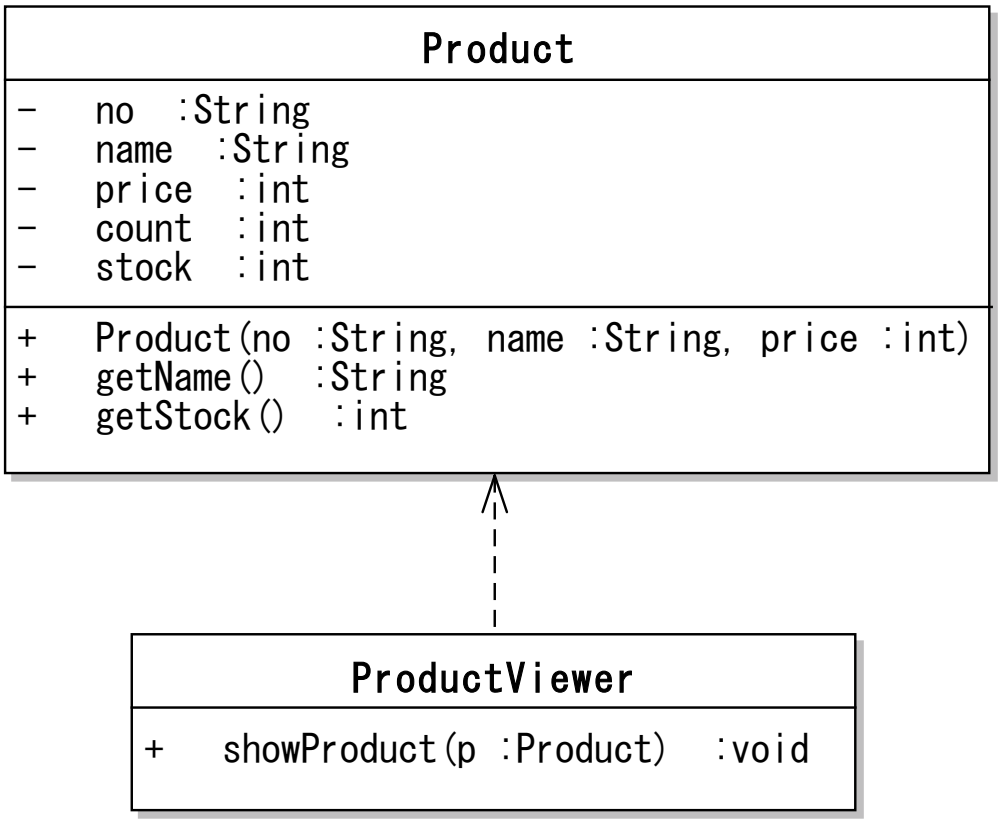
    public void showProduct(Product p) {

        // 商品名、在庫を出力する

        System.out.println(p.getName() + "の在庫＝" + p.getStock());

    }
}
```

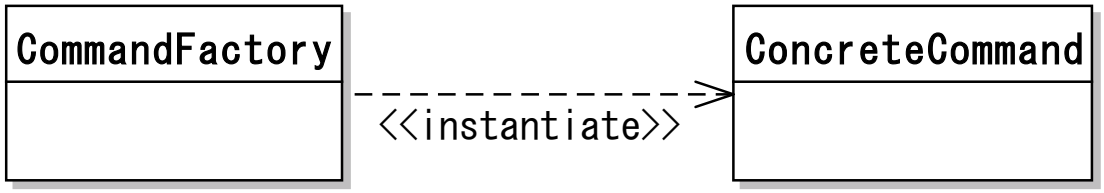
これは、クラス図では次のように、点線と矢印で表現します。クライアントからサプライアへ向かっての矢印になります。

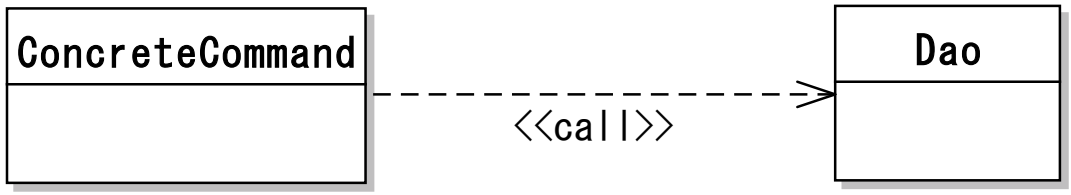


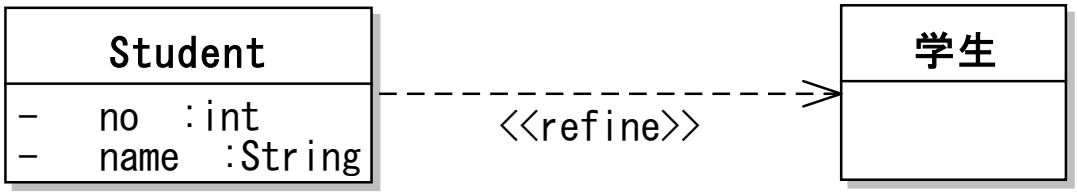
「依存」概念は、ある識別子を完成させるためには、他の識別子が必要であるという関係を表しており、その意味では、「関連」、「集約」、「合成集約」、「汎化」、「実現」を包括する概念です。しかし、「依存」

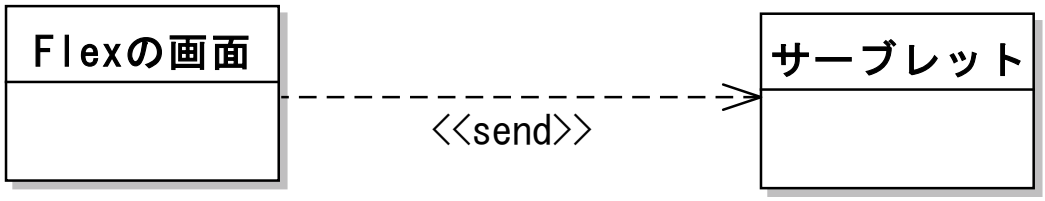
において、一定の条件に当てはまる場合、それは、それぞれ、「関連」、「集約」、「合成集約」、「汎化」、「実現」となり、いずれの条件にも当てはまらない場合、それを、単に「依存」とします。

依存においては、様々なステレオタイプを設定することができます。









それぞれのステレオタイプの意味は以下の通りです。

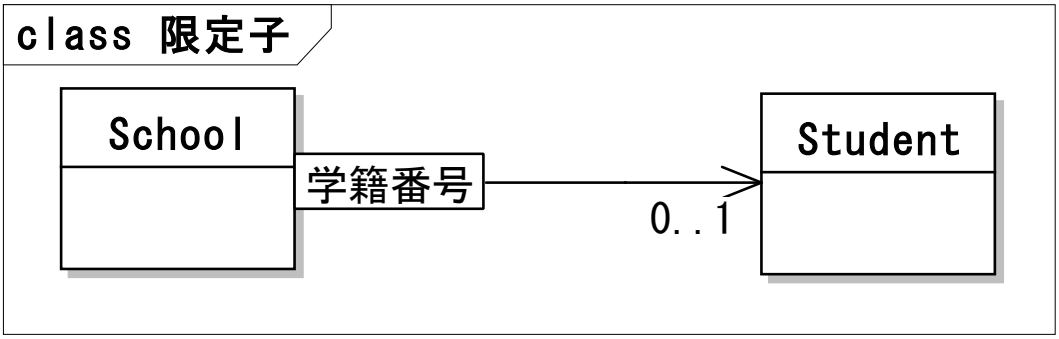
ステレオタイプ	意味
instantiate	操作の内部でサプライアをインスタンス化することを意味する。
call	クライアントの操作の内部でサプライアの操作を起動することを意味する。
refine	概念レベル（分析モデル）のクラスと論理モデル（設計モデル）のクラスのように、一方が他方を変換（洗練させた）した結果であることを意味する。
send	クライアントの操作の内部で、サプライアに対して非同期的な呼び出しを行うことを意味する。

18.21 限定子（qualifier）

「限定子」(qualifier)は、関連によって結びつけられたサプライアのインスタンスの集合から、クライアントが、特定のインスタンスもしくはインスタンスの集合の一部を選択することを宣言します。

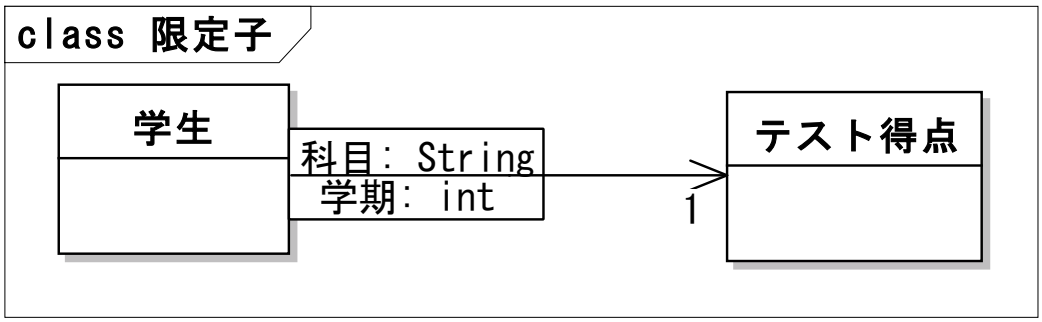
この場合、注意点は、限定子によってサプライアの多重度が制限されるということです。

たとえば、学校が文字列からなる学籍番号によって学生を一意に特定できる場合、次のように表現されます。



この場合、学校は学籍番号によって、ひとりの学生を特定できるか、該当する学生がいないかのどちらかであることを意味します。

次のように、限定子に複数の限定子属性を記述することもできます。この場合は、学生は、科目名と学期番号によって、テストの得点をひとつ特定することができることを意味しています。



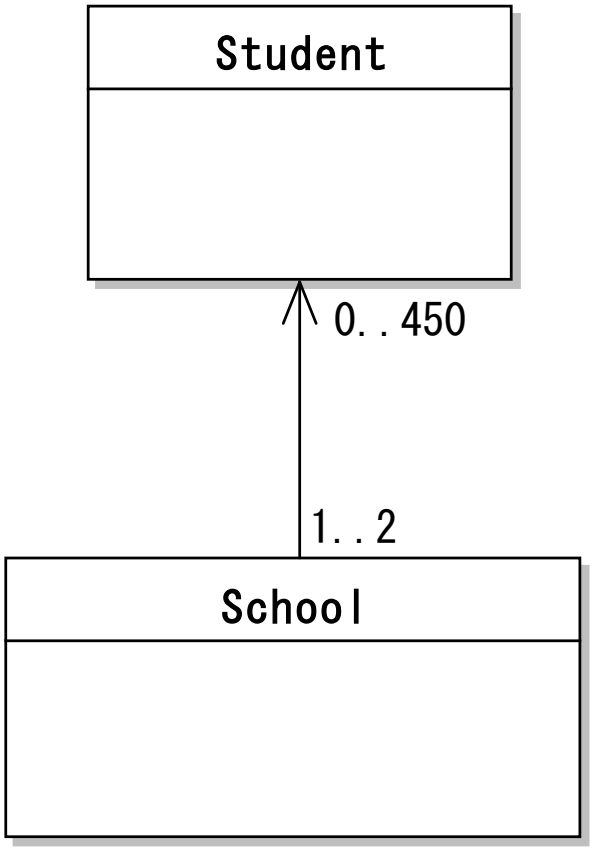
19コレクション・フレームワークと多重度

19.1 多重度

「多重度」(multiplicity)とは、クライアントのインスタンスとサプライアのインスタンスの間の数量的な関係を意味します。

多重度は、関連、集約、合成集約の接続線の両端(クライアント側とサプライア側)に付加することができます。

たとえば、次のように表記します。



上記の Student クラスと School クラスの例では、次のような意味になります。

Student インスタンスから見た School インスタンスの数は、「1..2」であり、Student インスタンスは所

属する School インスタンスが 1 もしくは 2 であることを意味します。

School インスタンスから見た Student インスタンスの数は、「0..450」であり、ひとつの School インスタンスには、ゼロ以上 450 以下の Student インスタンスが所属することができることを意味します。

以下、多重度の表記例です。

4	インスタンスの数が 4 つであることを意味する。
4..10	インスタンスの数が 4 以上 10 以下であることを意味する。
*	インスタンスの数が 0 以上であることを意味する。
4..*	インスタンスの数が 4 以上であること(上限無し)を意味する。
1,4,6	インスタンスの数が 1、4、6 のいずれかであることを意味する。
1,4..6	インスタンスの数が 1 もしくは 4 以上 6 以下であることを意味する。

20ユースケース図とユースケース記述

20.1 機能要求と非機能要求

システム開発において開発されるシステムは、「機能要求(functional requirement)」と「非機能要求(nonfunctional requirement)」の2つの種類の要求を満たすことが求められます。

「機能要求」とは、ユーザーが一定の目的を達成するために、ソフトウェアに対して要求する機能上の条件のことです。

「非機能要求」とは、ソフトウェアの性能や設計、セキュリティといった、機能とは異なるソフトウェアの品質に関する要求のことです。これは、必ずしもユーザーの要求する事柄とは限りません。ソフトウェアの品質特性の中でも特に保守性に関する要求は、ユーザー自ら構想することは困難です。しかし、保守性に優れた(＝優れた設計の)ソフトウェアは、開発期間の短縮、ユーザーのニーズに応じた柔軟な機能変更が可能といった仕方で、ユーザーの利益に還元されます。

つまり、ソフトウェアは、ただユーザーの目的(機能要求)を達成することだけではなく、一定の性能や高い保守性など(非機能要求)を満たすことも求められます。

機能要求と非機能要求の両者に関して、ユースケース図やユースケース記述は、ユーザーの機能要求を定義するための手段になります。

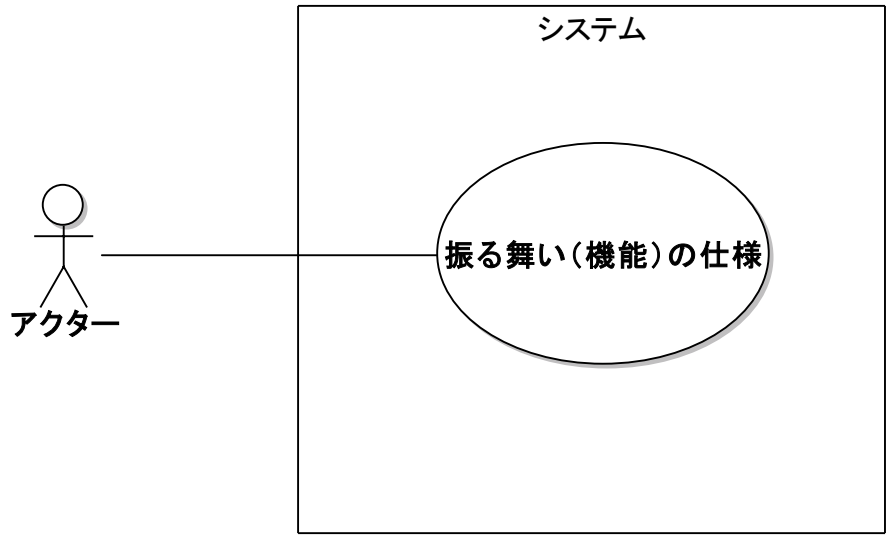
20.2 ユースケース図の意味と用途

「ユースケース図」(use case diagram)は、分析の段階(「システム要件定義」、「ソフトウェア要件定義」)において使用されるダイアグラムです。

ユースケース図に含まれる「ユースケース」(UseCase)は、システムが外部に提供する振る舞い(機能)についての仕様を表現します。「仕様」(specification)とは、ここでは、振る舞いの「実装」(implementation)に対立する概念です。「実装」は、振る舞いを実現する具体的な手段のことですが、「仕様」は、外部から見た場合のシステムの振る舞いに関する抽象的な規定を意味します。

一方、ここでの「システム」とは、ユースケースが規定する振る舞いの「主体」(subject)です。「アクター」(Actor)は、振る舞いの「主体」の外部に存在し、この「主体」と相互作用するユーザーや他のシステムの一定の「役割」(role)を表現します。

ここでいう「役割」とは、システムが外部に要求する振る舞い(行動)を包括する特定の「型」のことです。振る舞い(行動)に関する特定のパターンということもできます。



ここで、注意すべき点は、ユースケースは、主体の内部構造に言及することなく、主体が外部に対して、どのような振る舞い(機能)を提供するかを抽象的に規定するだけであるということです。つまり、ユースケースが振る舞いの仕様を表現するということは、振る舞いの実装については無関心であるということです。

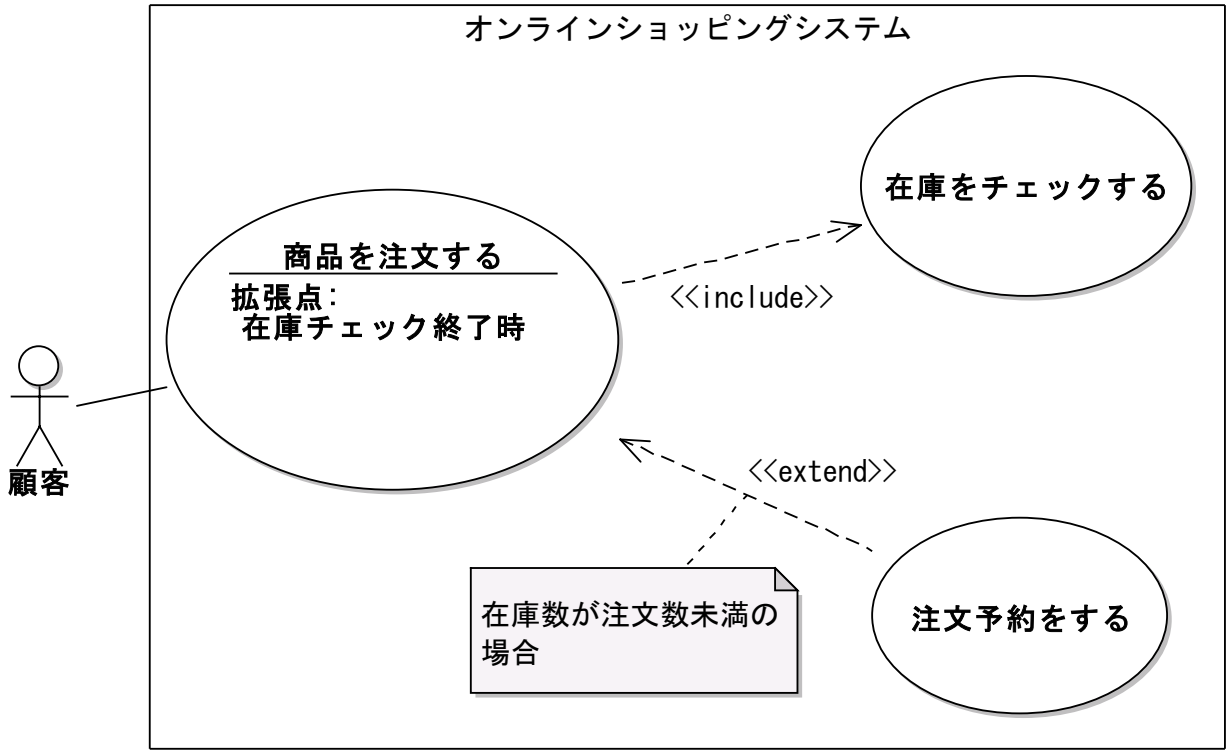
ユースケース図は、以下の 2 つの事柄を表現します。

1) システムが外部(ユーザーや外部の他のシステム)に提供する振る舞い(機能)の仕様

2) システムと相互作用する外部の実体(ユーザーや外部の他のシステム)が、そのために果たすべき役割

ユースケース図は、エンジニアだけでなく、開発されるシステムの関係者全体(経営層、業務層、IT システム層、ソフトウェア層のすべての利害関係者)の間で同意を得る必要があります。

以下の例は、オンラインショッピングシステムに関して、ユースケース図によって要求モデリングを行った例です。



20.3 ユースケース図の記法

■主体(subject)もしくはシステム境界(system boundary)

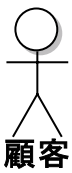
ユースケースが規定する振る舞いの仕様を実現する主体(システム、コンポーネント、サブシステム、クラスなど)を意味します。四角い枠によって表記します。<<subsystem>>などのステレオタイプを設定することも可能です。

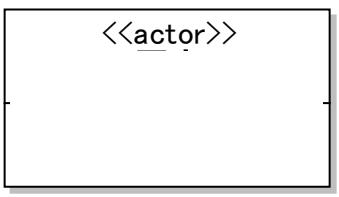


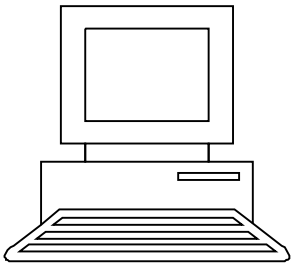
■アクター(Actor)

主体と相互作用する主体の外部に存在する実体の役割を表現します。一般的にはシステムを利用するユーザーや外部システムの役割を表現するために用いられます。

スティックマン、<<actor>>というステレオタイプを記述した四角形、パソコンの形のアイコンで表記します。





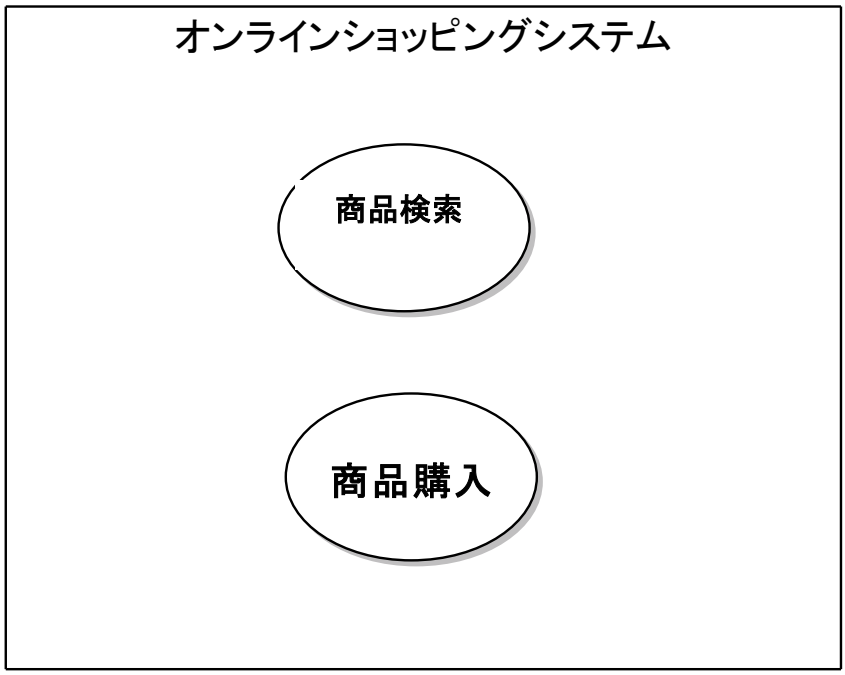


クレジットカード会社の
システム

■ユースケース(UseCase)

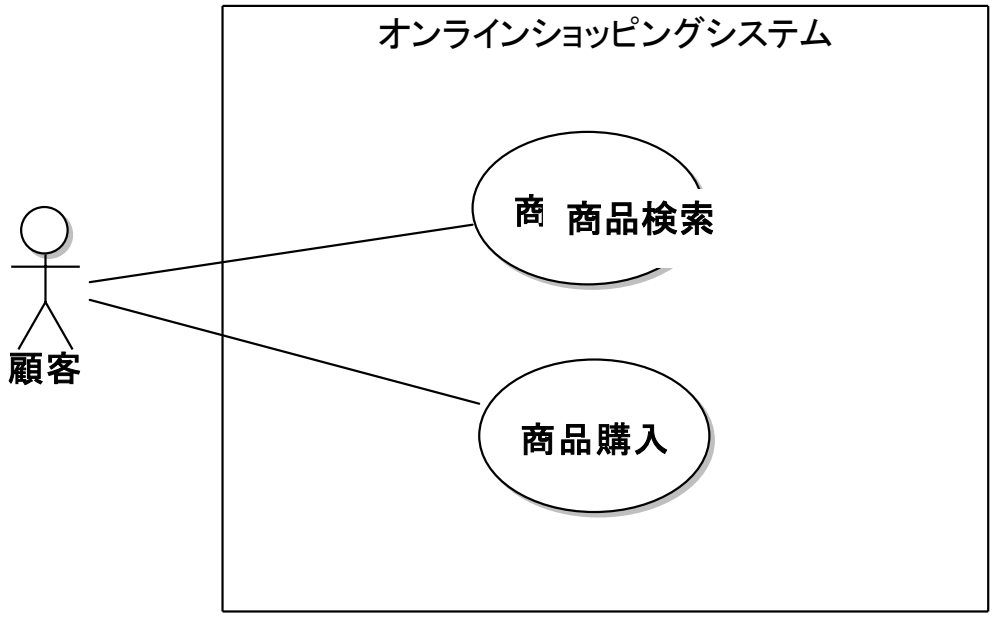
主体がシステムの外部に提供する振る舞いについての仕様を定義します。

主体(システム境界)を表現する四角の内部に、楕円によって表記し、さらに楕円の内部にユースケース名を記述します。ユースケース名については特にルールは定められていませんが、一般的には、「・・・を・・・する」という形式が良いとされています。



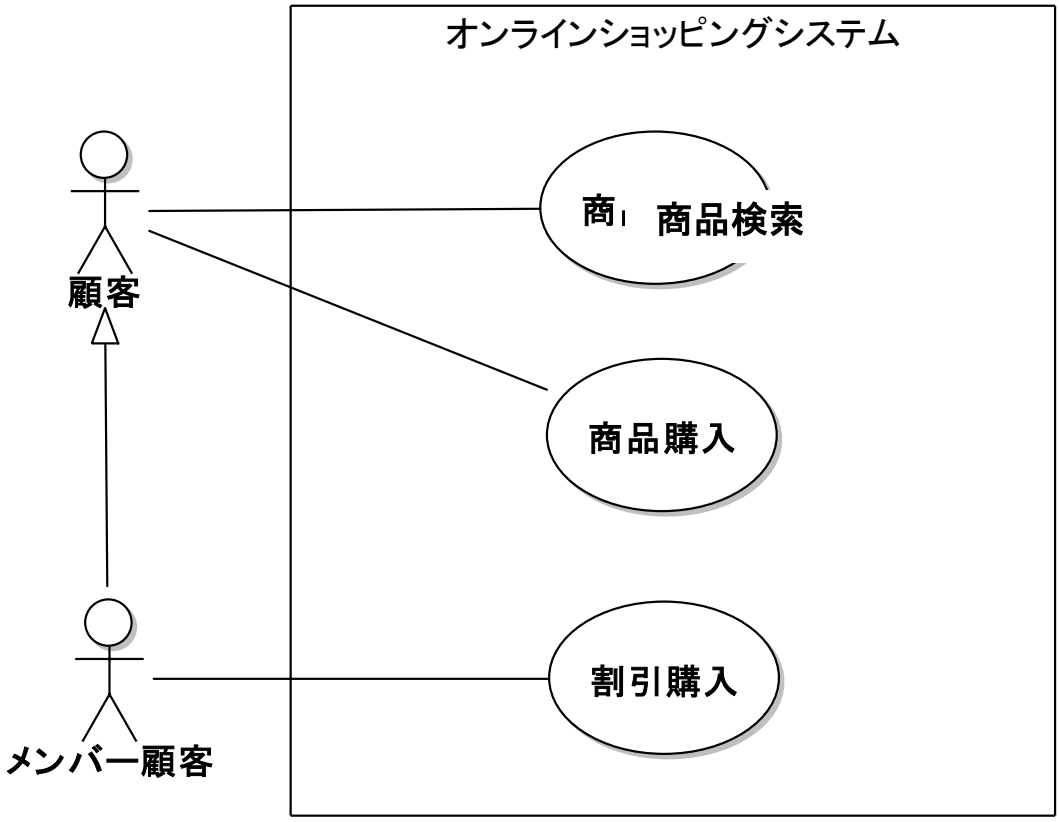
■関連(Association)

アクターとユースケースの関係を意味します。一般的には、利用の関係です。直線によって表記します。



■汎化(Generalization)

ユースケース同士やアクター同士での汎化を意味します。

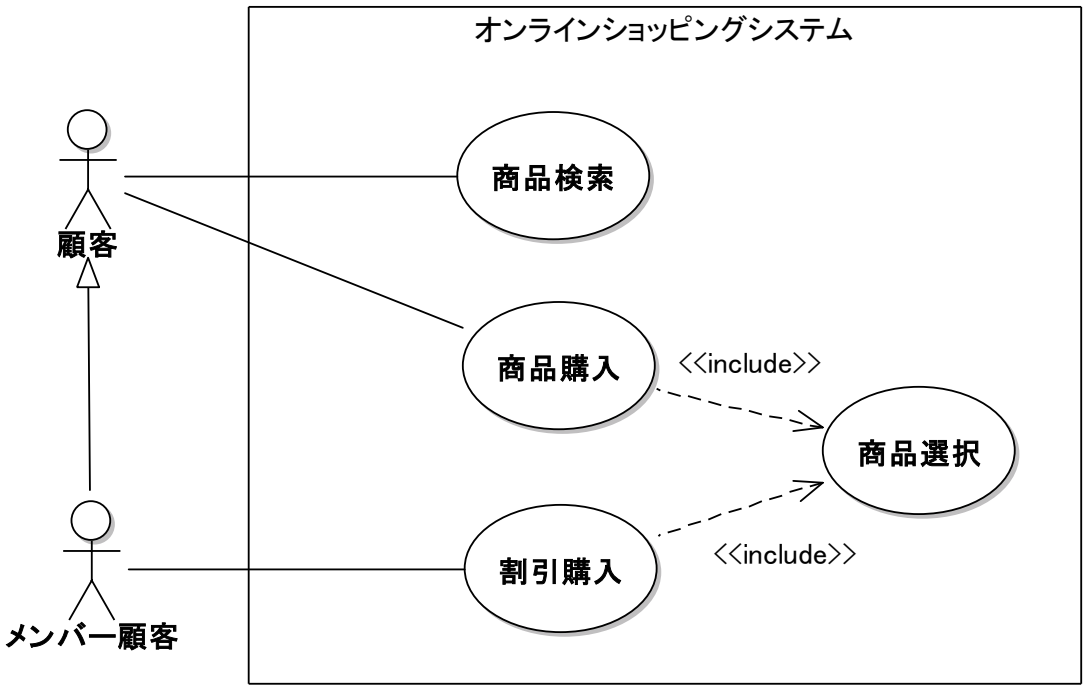


■包含(Include)

あるユースケースが他のユースケースの振る舞いを必ず含むという関係を表現します。

複数のユースケースに共通な振る舞いが含まれる場合、その振る舞いは、再利用可能な独立したユースケースとして抽出することができます。このような再利用される共通な振る舞いを表現する際に用います。

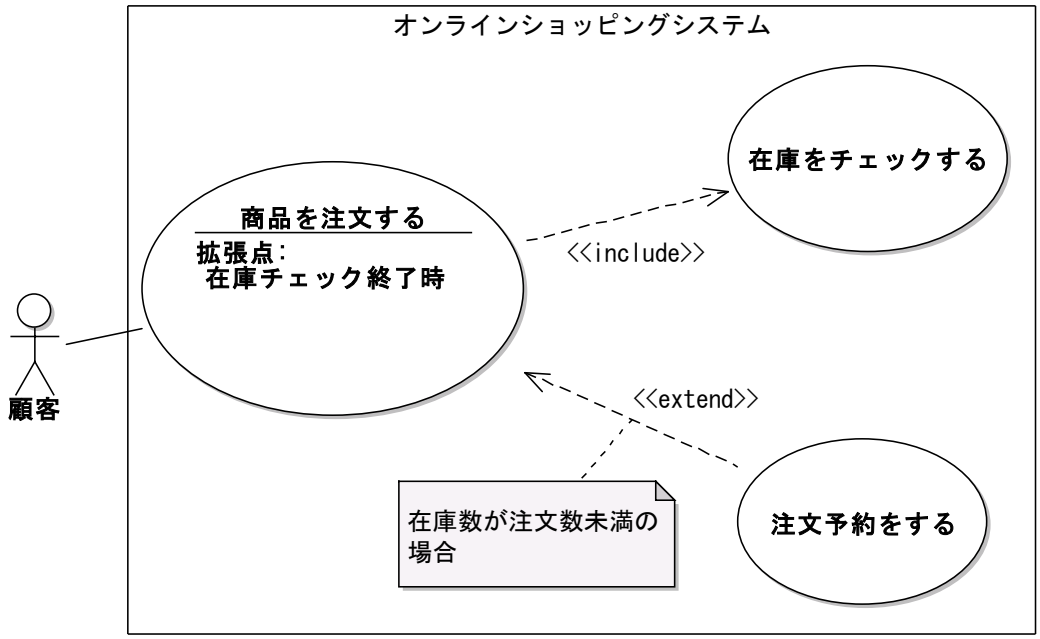
破線の矢印と<<include>>というステレオタイプによって表記します。このとき注意点は、包含するユースケースから包含されるユースケースに線を引くことです。



■拡張(Extend)と拡張点(ExtensionPoint)

「拡張」は、すでに存在するユースケースに、一定の条件下で、別のユースケースの振る舞いが挿入されることを意味します。この場合、挿入されるユースケースは、それ自体では、必ずしも意味を持つユースケースとは限りません。また、拡張において、拡張されるユースケースの内部に、拡張が挿入される箇所や時点を「拡張点」によって記述し、拡張の条件をノートによって記述します。

「拡張」は、破線の矢印と<<extend>>というステレオタイプによって表記し、「拡張点」は、ユースケース内部を直線で仕切り、その下部に記述します。このとき注意点は、拡張するユースケースから拡張されるユースケースに線を引くことです。



20.4 ユースケース図とユースケース記述

ユースケース図は、システムが外部に提供する振る舞いの仕様（抽象的な規定）を列挙するには適していますが、その振る舞いの内容を定義するのには適していません。ユースケース図のこの欠点を「ユースケース記述」（「シナリオ」、「ユースケーステキスト」、「イベントフロー」などとも呼ばれる）というドキュメントによって補う必要があります。これは、通常、自然言語によって記述されます。ユースケース記述は UML のダイアグラムではありませんが、ユースケース図に対応して作成する必要があります。対応の仕方は一対一、つまり、ひとつのユースケースに対して、ひとつのユースケース記述を作成します。

20.5 ユースケース記述のテンプレート例

ユースケース記述のテンプレートは、UML のダイアグラムのように規格化されているわけではありません。ユースケース記述については、様々な考え方があり、著名なエンジニアがそれぞれのテンプレートを提案しています。次のものは、比較的一般的な形式のテンプレートです[63]。

項目	記述内容
ユースケース名	ユースケース図で定義したユースケースの名前を記述する。
アクター名	ユースケースに関与するアクターの名前を記述する。
概要	ユースケースで処理される内容を要約します。
事前条件	ユースケースの振る舞いを開始する前に、アクターまたはシステムが事前に満たしておかなければならない条件を記述する。
事後条件	ユースケースの振る舞いを終了した後で、システムが満たしておかなければならない条件を記述する。

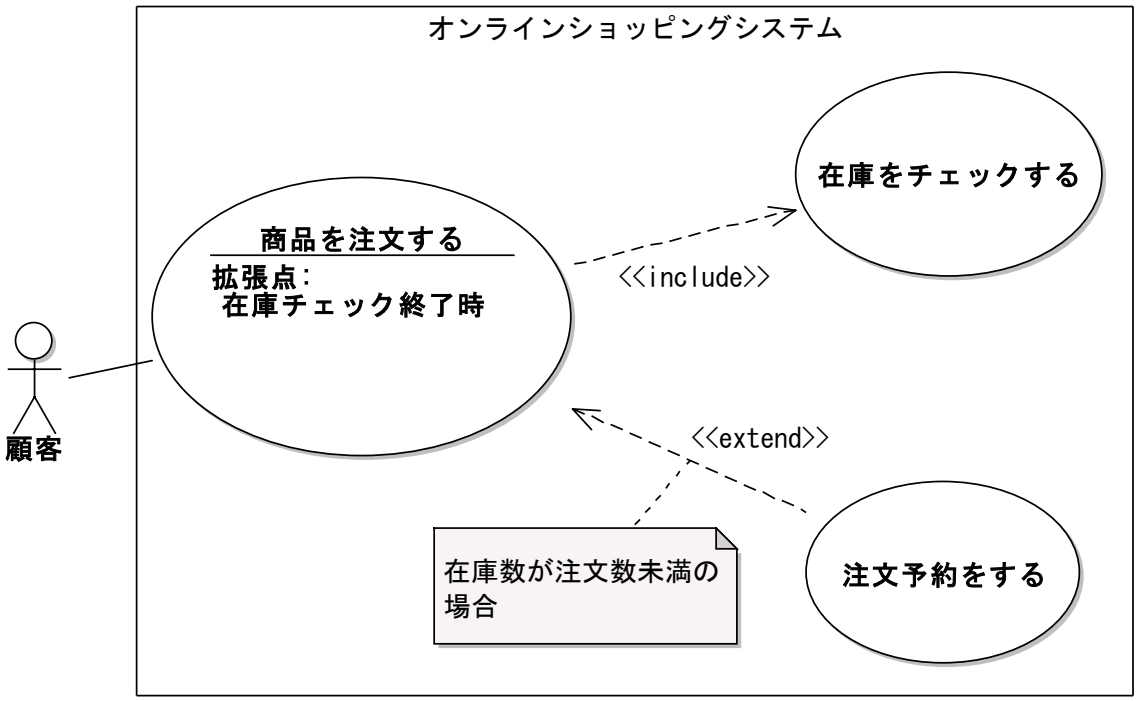
63 （株）NTT データソフトウェア工学推進センタ,『実例で学ぶソフトウェア開発』, オーム社, 2008, p.45（同書掲載のテンプレートを参考に、本書のテンプレートでは、「アクター名」という項目を追加するなどいくつかの修正を加えている。）

基本フロー	ユースケースの振る舞いが正常に進行した場合の、その振る舞いに含まれるひとつひとつの処理内容を記述する。処理内容には実行順序を示す番号を付加する。最大でも 7 項目前後に抑制した方が良いとされる。 このとき、アクターが主体の処理内容とシステムが主体の処理内容を交互に記述する。
代替フロー	例外の発生などにより基本フローを逸脱した場合の振る舞いに含まれる、ひとつひとつの処理内容を記述する。対応する基本フローの振る舞いの番号にアルファベットの枝記号を添えた番号を付加する。
備考	基本フローおよび代替フローに含まれる処理内容に関する細かい条件や注意点などを記述する。

20.6 ユースケース図とユースケース記述の例１

次のユースケース図のユースケース「商品を注文する」に関するユースケース記述について考えてみます。

■ユースケース図



■ユースケース記述

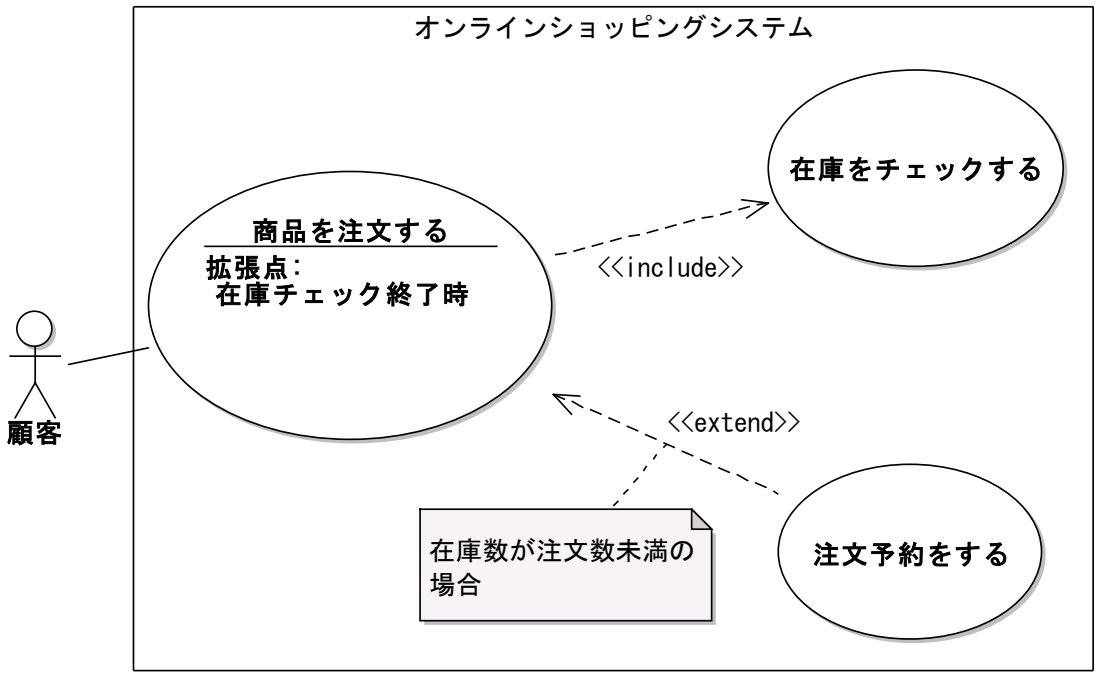
項目	記述内容
ユースケース名	商品を注文する
アクター名	顧客
概要	顧客からの商品の注文を受け付け、注文情報を登録し、注文番号を発行する。
事前条件	なし
事後条件	顧客からの注文情報がシステムに登録され、顧客に対して注文番号が発行される。
基本フロー	1.顧客が注文の要求を行うことにより、本ユースケースを開始する。 2.システムは、顧客に注文数の指示を促す。 3.顧客は注文数をシステムに指示する。 4.システムは、ユースケース「在庫をチェックする」を起動する。 5.システムは、顧客に対し、氏名と連絡先の指示を促す。 6.顧客は、システムに対し、氏名と連絡先を指示する。 7.システムは、注文情報として、顧客の氏名、連絡先、商品番号、注文数を登録する。 8.システムは、顧客に注文番号を提示する。 9.顧客は注文番号を確認して、本ユースケースを終了する。

代替フロー	4a. 注文数未入力の場合 (1) 基本フロー２. に戻る。 4b. 在庫が不足している場合 (1) 顧客にエラーの旨を提示する。 (2) ユースケース「注文予約をする」を起動する。 (3) 本ユースケースを終了する。

20.7 ユースケース図とユースケース記述の例２

先程の例と同じユースケース図のユースケース「商品を注文する」に関して、インターフェイス上の操作方法に言及する仕方で、ユースケース記述を作成してみます。

■ユースケース図



■ユーザーインターフェイス（抽象的な機能だけを備えた画面）

【商品詳細画面】

Wamazon

商品画像

タイトル情報

著者情報

出版社情報

注文開始

【注文数入力画面】

Wamazon

タイトル	価格	注文数
・・・・・・・・・・	￥・,・・・	個

次へ

【注文者情報入力画面】

Wamazon

氏名	
連絡先	

完了

【注文完了画面】

Wamazon

注文番号	・・・・・・・・・・・・・・・・・・・・

ここに、注文番号を確認させるためのメッセージを表示

【注文予約開始画面】

Wamazon

商品の在庫が不足しているため、注文予約を行う必要がある旨のメッセージを表示

注文予約開始　キャンセル

■ユースケース記述

項目	記述内容
ユースケース名	商品を注文する
アクター名	顧客
概要	顧客からの商品の注文を受け付け、注文情報を登録し、注文番号を発行する。
事前条件	なし
事後条件	顧客からの注文情報がシステムに登録され、顧客に対して注文番号が発行される。

基本フロー	1.顧客が＜商品詳細画面＞上で、＜注文開始＞ボタンを押すことにより、本ユースケースを開始する。 2.システムは、顧客に＜注文数入力画面＞を表示する。 3.顧客は＜注文数入力画面＞上で、＜注文数＞を入力し、＜次へ＞ボタンを押す。 4.システムは、ユースケース「在庫をチェックする」を起動する。 5.システムは、顧客に対し、＜注文者情報入力画面＞を表示する。 6.顧客は、＜注文者情報入力画面＞上で、＜氏名＞と＜連絡先＞を入力し、＜完了＞ボタンを押す。 7.システムは、注文情報として、顧客の氏名、連絡先、商品番号、注文数を登録する。 8.システムは、＜注文完了画面＞を表示し、顧客に対して＜注文番号＞と＜注文番号を確認させるためのメッセージ＞を表示する。 9.顧客は注文番号を確認して、本ユースケースを終了する。
代替フロー	4a. 注文数未入力の場合 (1)基本フロー２.に戻る。 4b. 在庫が不足している場合 (1)システムが＜注文予約開始画面＞を表示する。 (2a)顧客が＜注文予約開始＞ボタンをクリックし、ユースケース「注文予約をする」を起動する。 (2b)顧客が＜キャンセル＞ボタンを押して、本ユースケースを終了する。

21シーケンス図

21.1 シーケンス図の意味と用途

オブジェクト指向言語によって作成されたプログラムでは、クラスやインスタンスといった要素が他の要素を利用しながら、アプリケーションの機能全体を実現します。他の要素を利用するということの実装は、主に相手の振る舞い(メソッド)を呼び出すことです。したがって、アプリケーションの機能をどのように実装するかを示したい場合、そこにどのような振る舞いの主体が参加し、どのように他の主体の振る舞いを呼び出すのかを示すことは、ひとつの有力な手段になります。シーケンス図は、そのような振る舞いの主体(クラス、インスタンスなど)同士の、振る舞いの呼び出しを、「メッセージ」(Message)の送受信として時系列的に表現したダイアグラムです。

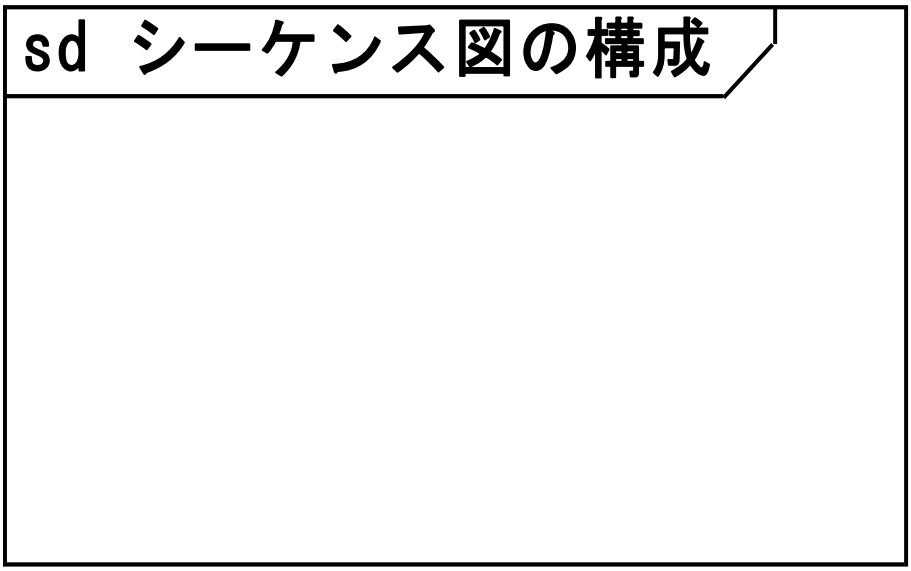
ICONIX プロセスでは、シーケンス図は、「詳細設計」段階において作成し、ひとつのユースケースに対応して、ひとつのシーケンス図を作成します。これにより、ひとつのユースケースによって表現されたシステムの振る舞いが、オブジェクト間のメッセージの送受信として表現されます。さらに、シーケンス図の作成とその検証により、クラス図に含められるべき操作を確定することができます。ICONIX プロセスにおいては、属性だけでなく操作も含めたクラス図の完成が、シーケンス図作成の主要な目的のひとつとなっています。この場合、シーケンス図は、設計における設計モデル(論理モデル)を表現します。

一方で、シーケンス図は、分析において分析モデル(概念モデル)の表現に利用することも可能です。その場合は、シーケンス図は、アクティビティ図と同様にワークフローやビジネスプロセスあるいはシステムの概念的なフローを表現します。そのような表現においては、振る舞いの主体は、ユーザー、企業、概念モデルとしてのクラスやインスタンスとなります。

21.2 シーケンス図の記法

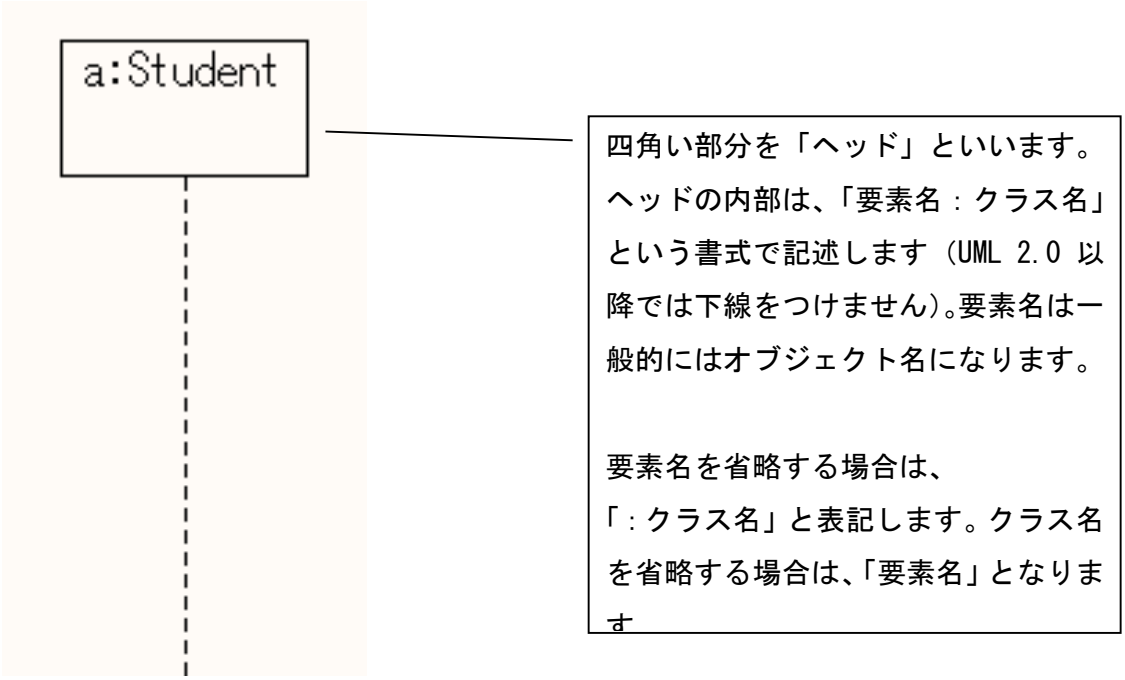
■相互作用(インタラクション:Interaction)

四角いフレームの、左上に表題をつけて作成します。要素と要素がメッセージをやりとりしながら実現する処理の単位。

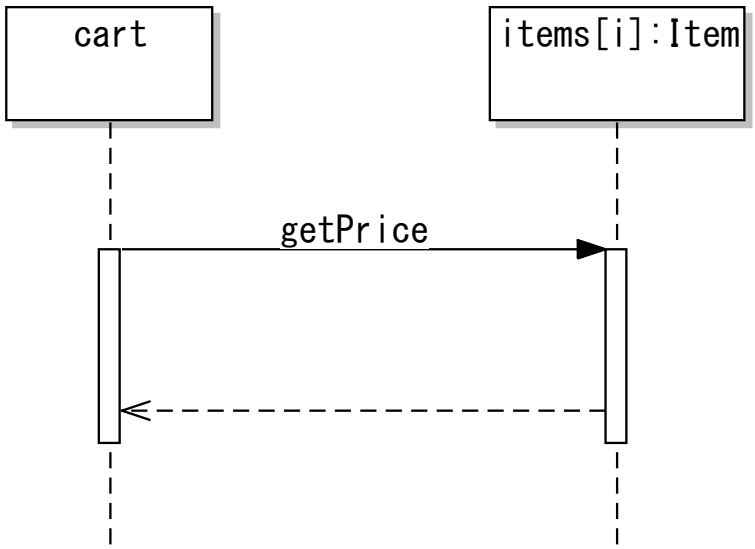


■生存線(LifeLine:ライフライン)

「ヘッド」(head)と垂直な点線から構成されます。インタラクションに参加している要素(クラス、インスタンスなど)つまり振る舞いの主体を意味します。



ライフラインが、配列やコレクションの要素を意味する場合は、次のように表現します。

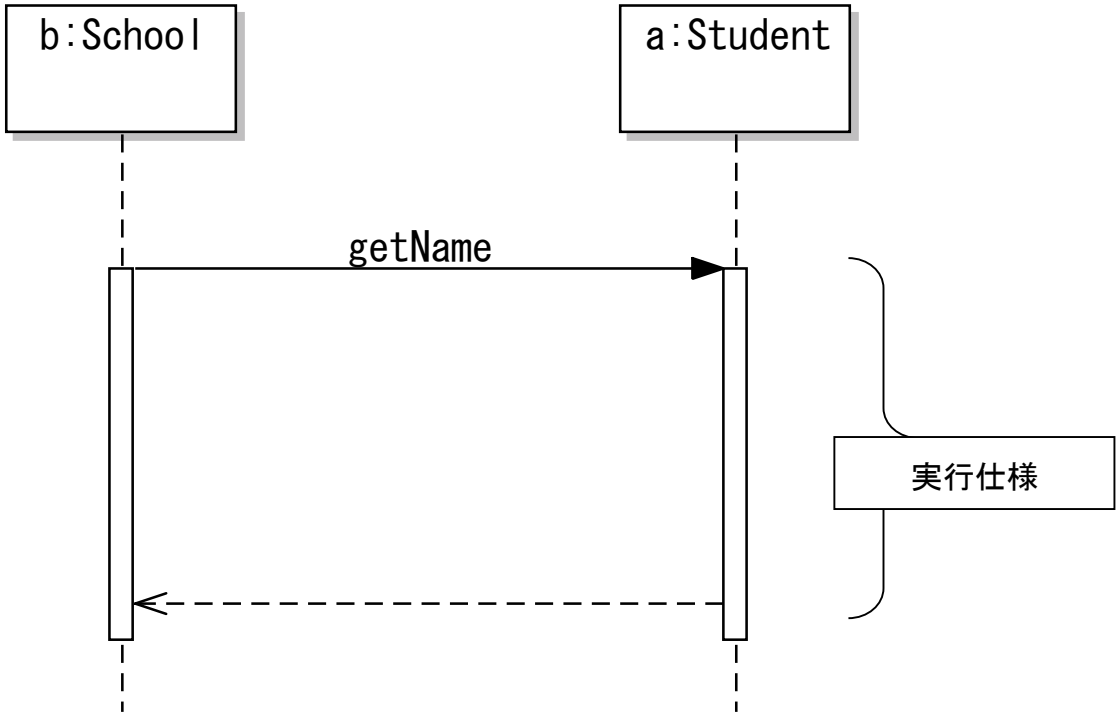


この場合の書式は、「配列もしくはコレクションとなる要素名[式]:クラス名」です。

■メッセージ(Message)と実行仕様(ExecutionSpecification)

「メッセージ」(Message)は、ライフラインとライフラインの間のコミュニケーション(シグナルの送受信、操作の呼び出し、インスタンスの作成など)を意味します。

「実行仕様」は、ライフライン上で、操作が実行されている期間を表します。



21.3 メッセージの意味と種類

■メッセージの意味

シーケンス図における「メッセージ」(Message)は、主として操作の起動(同期処理)、シグナルの送受信(非同期処理)を表現します。

■メッセージの種類

メッセージは主に次の 3 種類があります。

1)同期メッセージ(synchCall、Synchronous Message)

操作(一般的にはメソッド)の呼び出しを表現します。実線と黒い三角形によって表記します。



2)非同期メッセージ(asynchCall、Asynchronous Message:異なるスレッドの処理を起動する場合)

非同期呼び出しを表現します。実線と黒い三角形によって表記します。非同期メッセージは、戻り値を取得することができません。



3)戻りメッセージ(reply message)

同期メッセージの呼び出し側に対する戻り値の返却を表現します。点線と矢印によって表記します。



これ以外に、インスタンスの生成を行う場合のメッセージの表記や、メッセージの発生源がシステムのスコープ外の場合やメッセージが受信されなかった場合の表記があります。

特に、「ファウンドメッセージ」と「ロストメッセージ」は、それぞれ同期メッセージおよび非同期メッセージと組み合わせて使用します。

4)オブジェクト生成メッセージ(Object creation Message)

コンストラクタの呼び出しなどインスタンスの生成を表現します。点線と矢印によって表記します。



5)ファウンドメッセージ(Found Message)

メッセージの発生源がシーケンス図のスコープ外であることを表現します。黒丸を起点とするメッセージによって表記します。



6)ロストメッセージ(Lost Message)

メッセージの送信は確認されているが、メッセージの受信がないことを表現します。これは、メッセージが相手に届かなかった場合を表現します。黒丸を終点とするメッセージによって表記します。



21.4 メッセージ名の記法

メッセージを示す線の近くには、メッセージ名を表記します。

■記法

1)同期メッセージ

メッセージ名（パラメータ名＝値,パラメータ名＝値・・・）

メッセージ名（値,値・・・）

メッセージ名（パラメータ名,パラメータ名・・・）　　※値が不明

メッセージ名（ー,ー・・・）　※引数の値が未定義（「undefined」）

2)非同期メッセージ

シグナル名（属性＝出力パラメータ名：値,属性＝出力パラメータ名：値・・・）

シグナル名（属性＝出力パラメータ名,属性＝出力パラメータ名・・・）

シグナル名（出力パラメータ名,出力パラメータ名・・・）※値が不明

シグナル名（ー,ー・・・）　※引数の値が未定義（「undefined」）

3)戻りメッセージ

戻り値を格納する変数名=メッセージ名（変数名＝値,変数名＝値・・・）：戻り値

戻り値を格納する変数名=メッセージ名（値,値・・・）：戻り値

戻り値を格納する変数名=メッセージ名（変数名,変数名・・・）：戻り値
※変数名のみを表記する場合は値が不明であることを意味する

戻り値を格納する変数名=メッセージ名（ー,ー・・・）:戻り値
※引数の値が未定義（「undefined」）

■補足

UML2.1 正式仕様書の Figure 14.11 では、ハイフン(-)は、戻りメッセージにおいて値を省略して表現したい場合に用いられるように見えます。

メッセージの記法に関しては、UML2.1 正式仕様書の「14.3.20 Message (from BasicInteractions)」において、次のように表現されています(一部単純化して表記)。

attribute = signal-or-operation-name
(parameter-name=argument-value) : return-value

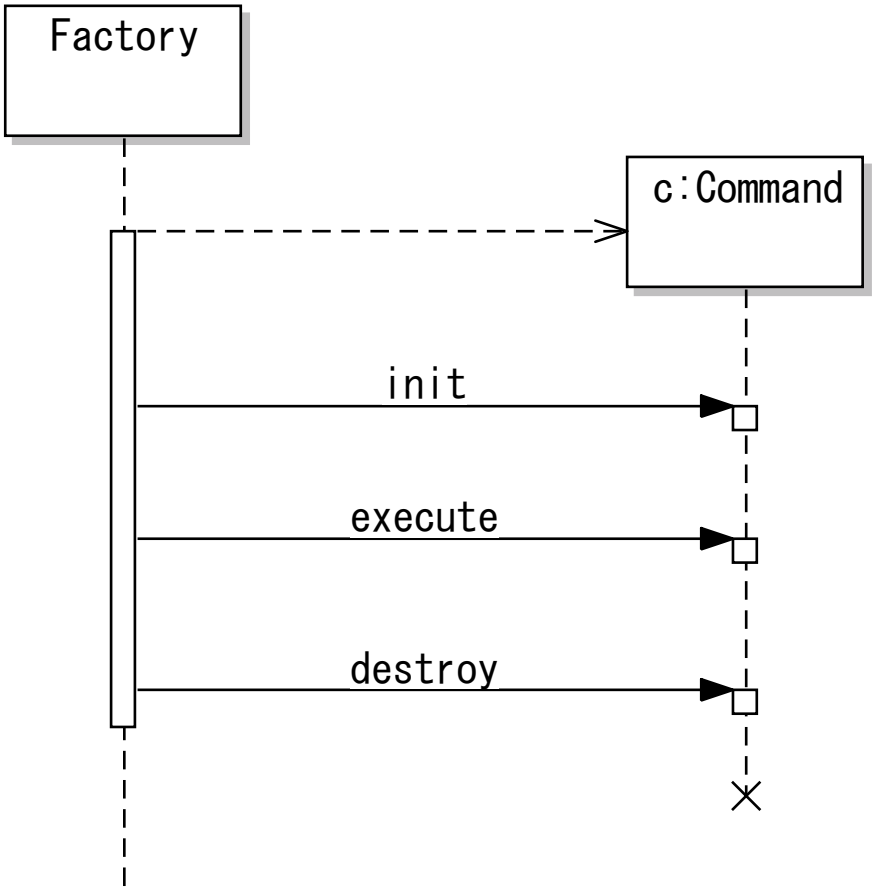
戻り値を格納する「変数」は、「attribute」(属性)と表現されており、メッセージの送信側のライフラインの"属性"の意味であると思われますが、一般的には、送信側のライフラインの振る舞いの内部においてアクセス可能な"変数"と解釈することができます。

■記述例

```
mymessage(14, - , 3.14, "hello")
v=mymsg(16, x):96
v=mymsg( - , - ):96
mymsg(x=16)
```

21.5 インスタンス化とインスタンス破棄の表現

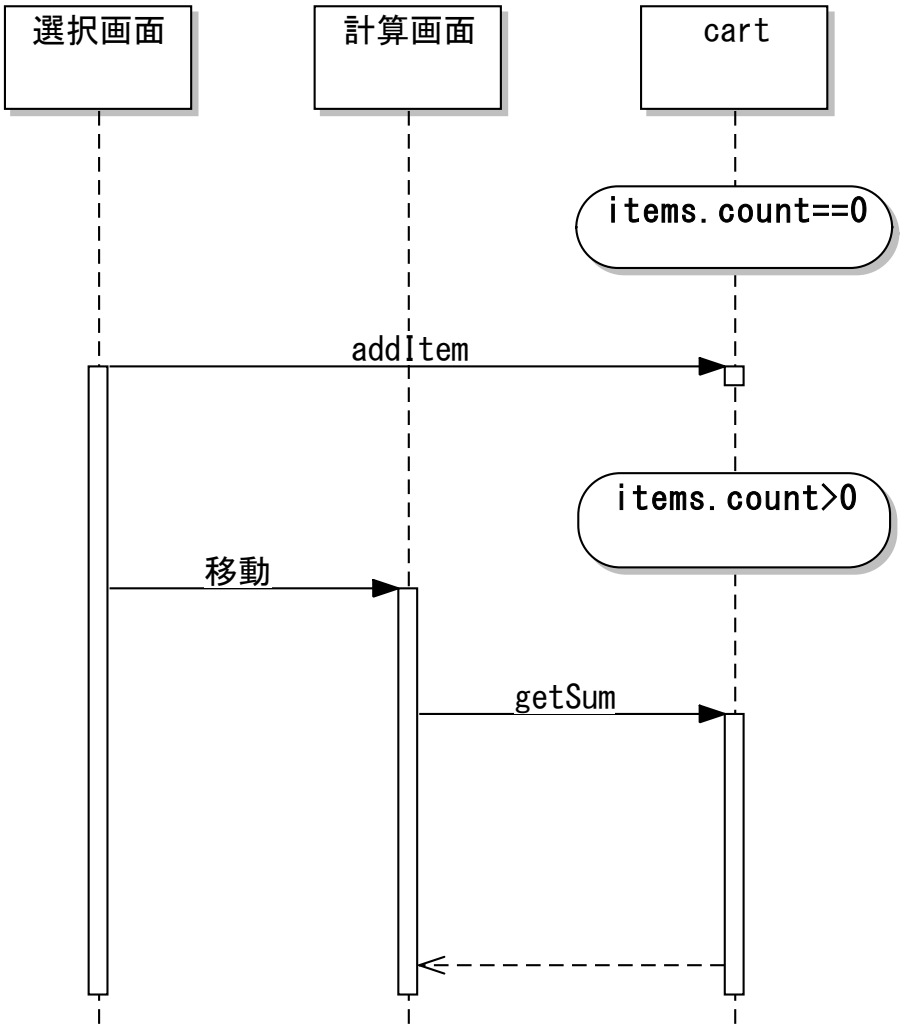
コンストラクタの呼び出しや factory メソッドによるインスタンスの生成は、ライフラインのヘッドを終点とするオブジェクト生成メッセージ（点線と矢印によるメッセージ）によって表記します。インスタンスの破棄は、ラインラインの下端に×印を記述します。



21.6 オブジェクトの状態に関する制約の表現

ライフラインが表現するオブジェクト内部の属性・変数の値やオブジェクト内部・外部の状態に関して、一定の制約が存在することを表現することができます。これは、「状態不変式」(StateInvariant)によって表現します。

次の例では、cart 内部で参照されるコレクション items のカウントプロパティは初期状態では 0 なければならず、集計を行うメソッド getSum を呼び出す前には 0 を超えていなければならないことを意味します。



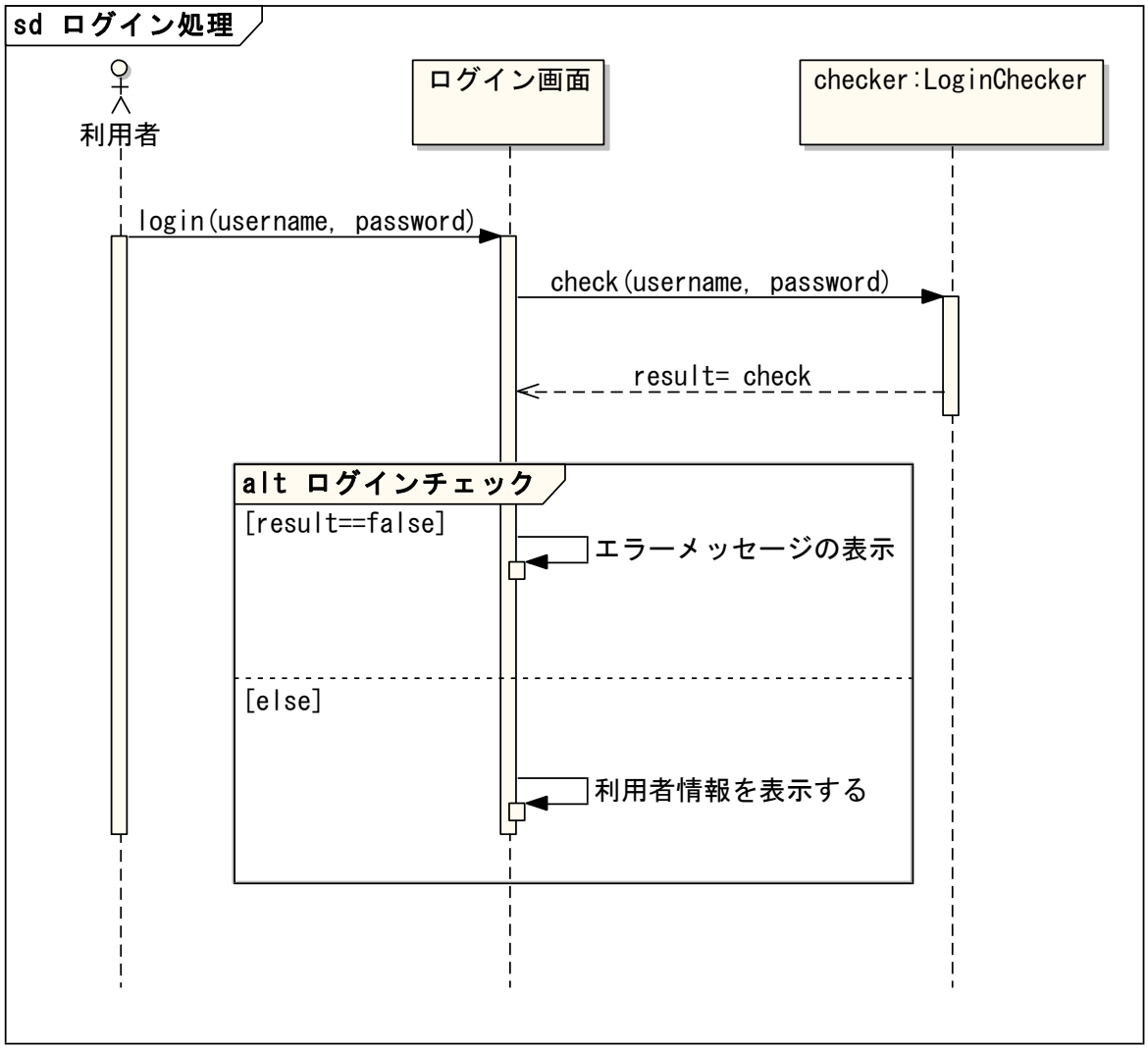
「状態不変式」は、楕円の中に式を記述します。

21.7 複合フラグメント

「複合フラグメント」(CombinedFragment)は、分岐処理、反復処理などの複雑なメッセージの送受信を表現することができます。

■分岐処理の表現1

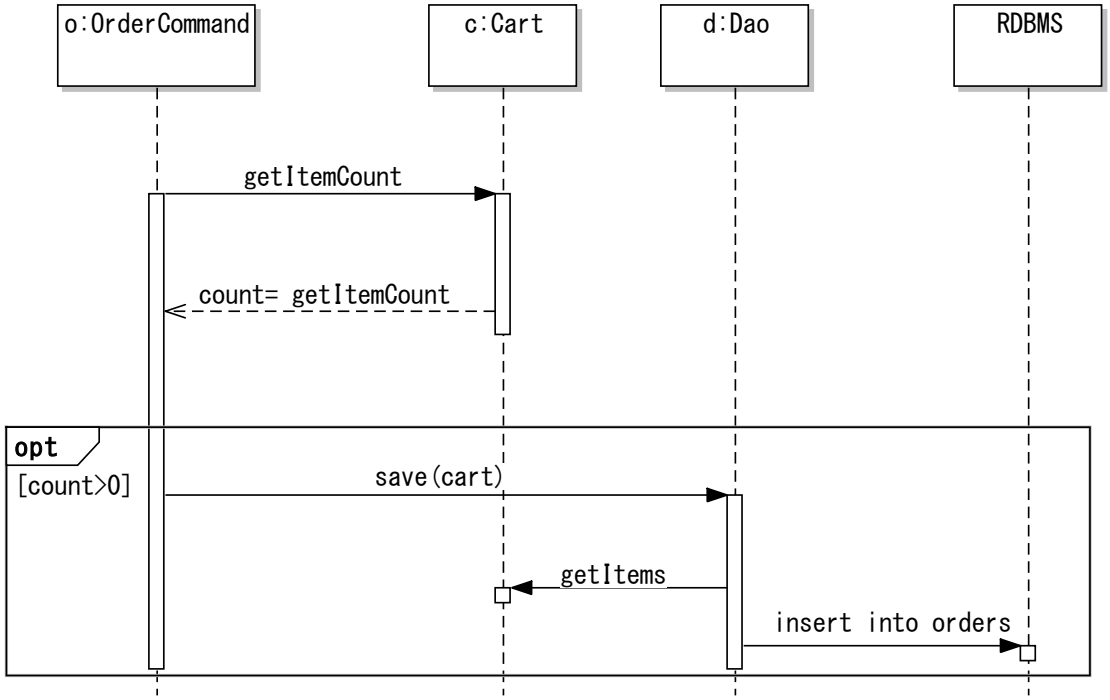
Java の if 文や switch 文による分岐処理の表現に対応します。



アクターの形状をしたライフラインは、UML2.1 の正式仕様書には規定されていませんが、一般的によく用いられます。

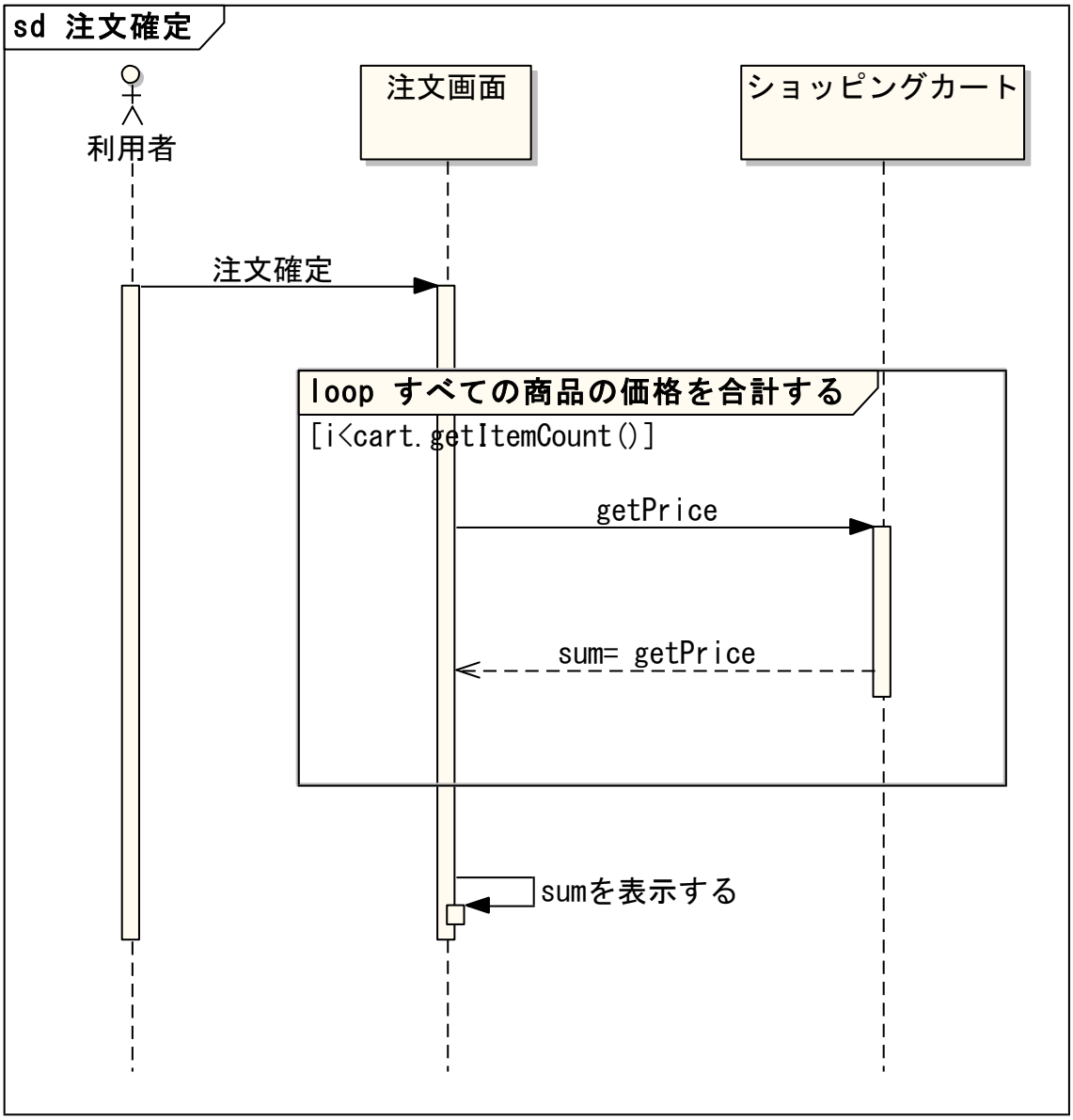
■分岐処理の表現2

「alt」(alternative)とは異なり、分岐がひとつしかない場合の記法です。



この場合、変数 count に格納された getItemCount メソッドの戻り値が 0 を超えていた場合にのみ、save メソッド以下の処理を実行します。

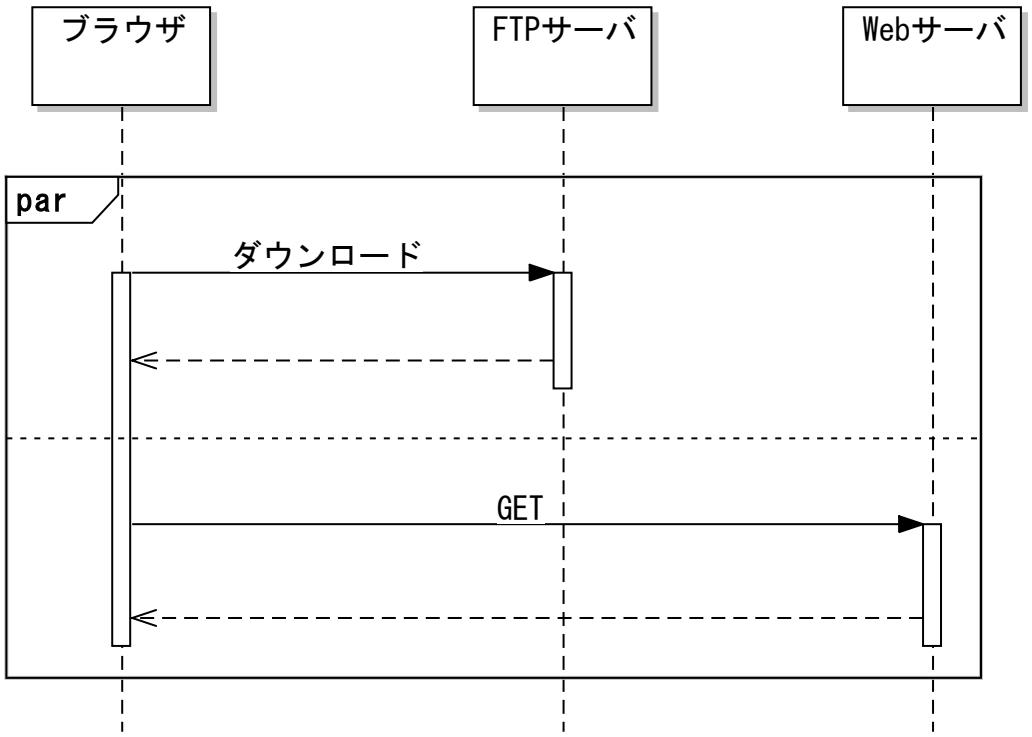
■反復処理の表現


sd 注文確定
利用者
注文画面
ショッピングカート
注文確定
loop すべての商品の価格を合計する
[i<cart.getItemCount()]
getPrice
sum= getPrice
sumを表示する

■並列処理

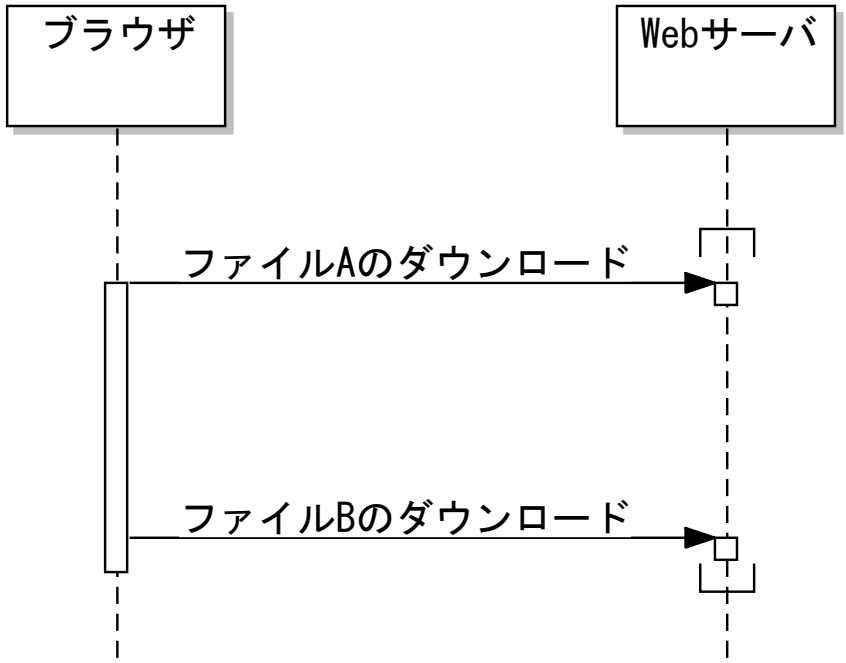
マルチスレッドによる並列処理を表現することができます。

下図の例では、ブラウザから、FTP サーバにアクセスしファイルをダウンロードする処理と、Web サーバに GET メソッドによってアクセスし HTML を閲覧する処理を、並列的に実行する様子を表現しています。



並列処理となる複数のメッセージが、ひとつのライフラインに向かう場合は、次のように、「コリジョンエリア」(coregion area)として表現することもできます。

下図の例では、ブラウザから、同じWebサーバに対して異なるファイルのダウンロードを並列的に実行する様子を表現しています。



21.8 インタラクションのネストの表現

「相互作用使用」(InteractionUse)を用いることにより、「相互作用」(Interaction:インタラクション)を他の「相互作用」の内部で参照したり、複数の「相互作用」の内部で共通の「相互作用」を参照したりすることが可能です。

たとえば、次のような Java のソースを考えます。

■ソースコード

```
public class Test {

    public static void main(String[] args) {

        int x = 20, y = 40;

        Calculator calc = new Calculator();

        calc.doCalc(x, y);

        int resultValue = calc.getResultValue();

        System.out.println(resultValue);

    }
}
```

```
class Calculator {

    private Result result;

    public void doCalc(int a, int b) {

        int c = a + b;

        Result result = new Result(c);

        this.result = result;
    }

    public int getResultValue() {

        int resultValue = this.result.getValue().intValue();

        return resultValue;
    }
}

class Result {

    private Integer value;

    public Result(int value) {

        this.value = new Integer(value);

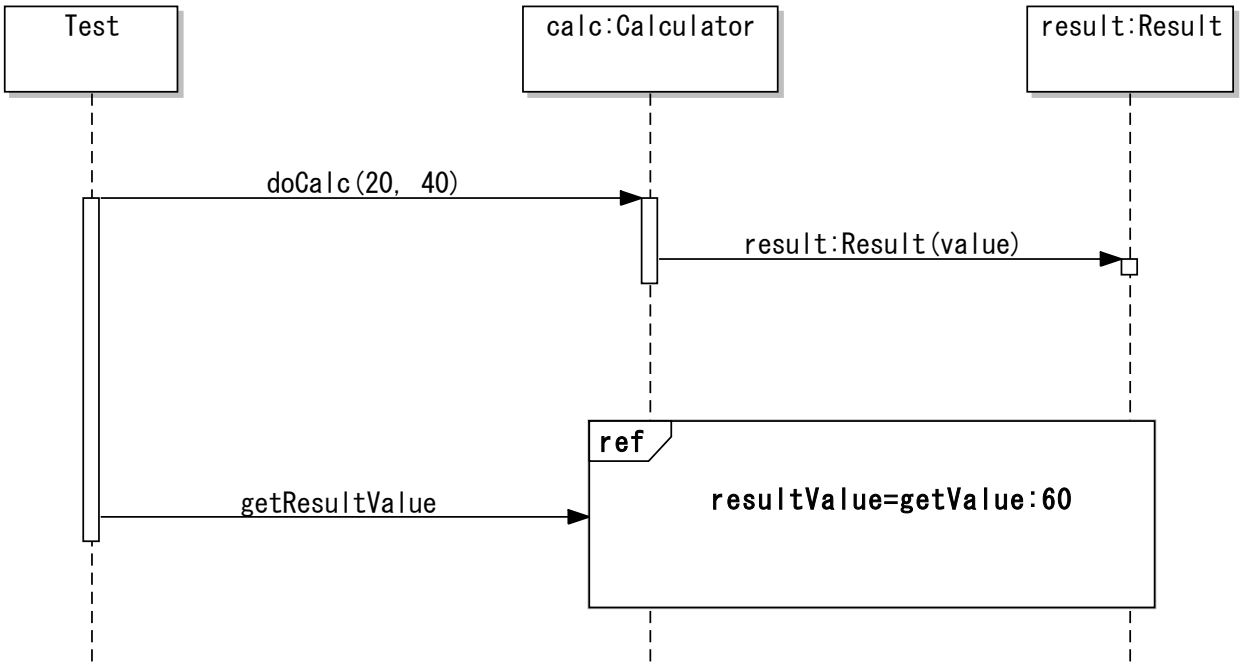
    }

    public Integer getValue() {

        return this.value;
```

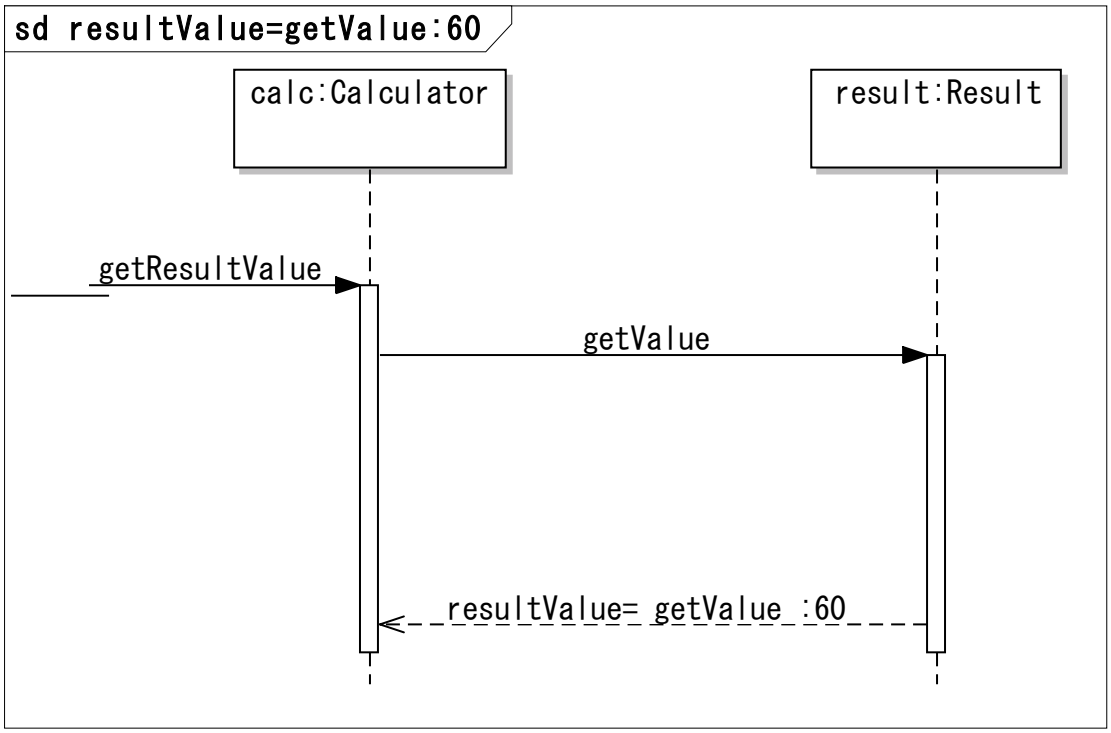
```
        }
    }
```

■相互作用使用を行う側のシーケンス図



※相互作用使用に入力されるメッセージの受信側の端点を「ゲート」(Gate)といいます。

■参照される相互作用



※相互作用自体に入力されるメッセージの送信側の端点を「ゲート」(Gate)といいます。
相互作用使用の名前は、以下の書式に従います(一部簡略化)。[　]内は省略可能です。

[属性名 =] 相互作用名[（引数,・・・）][: 戻り値]

※属性名は、相互作用に含まれるライフラインの属性を指定する

相互作用使用を表記する上での注意点は、以下の通りです。

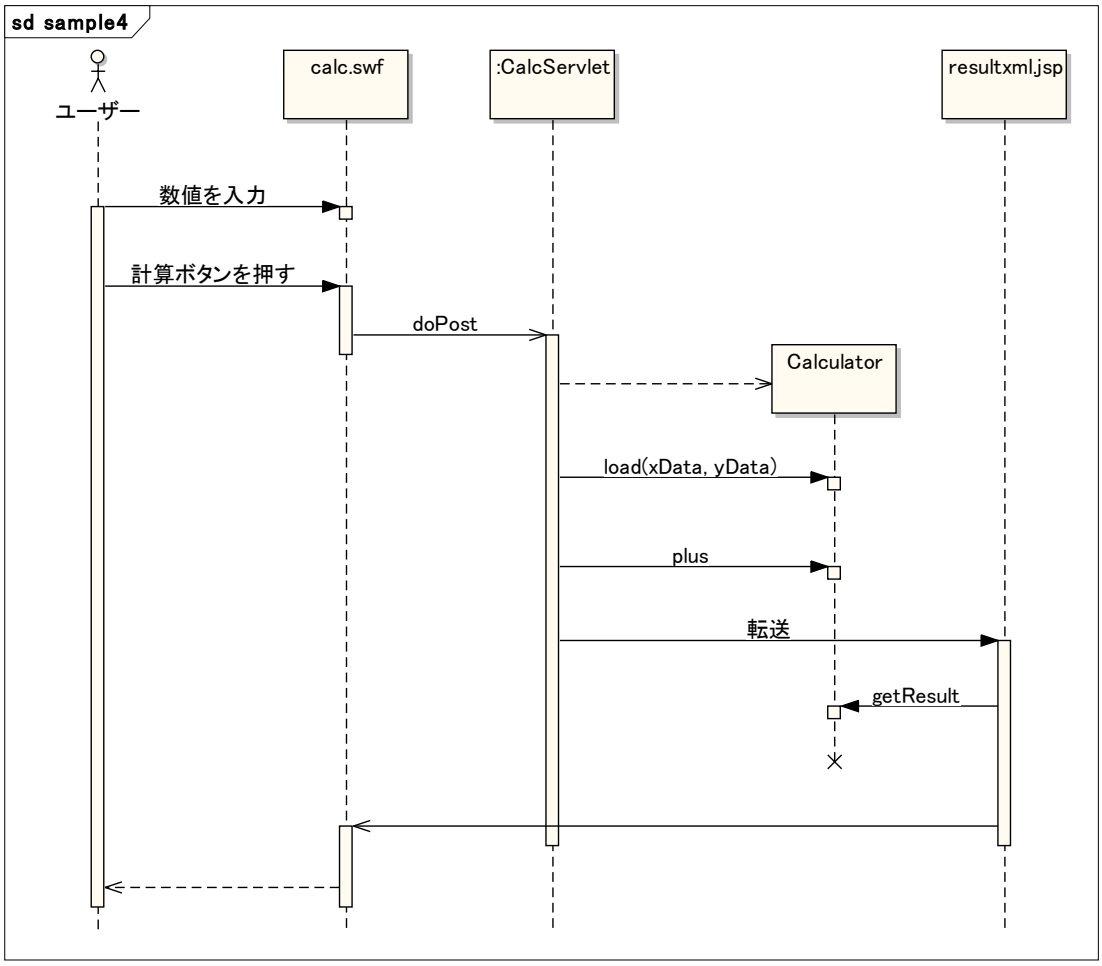
1）相互作用使用の名称と参照される相互作用の名称は合致していなければならない。

2）相互作用使用に含まれるライフラインと参照される相互作用に含まれるライフラインは一致していなければならない。

3）相互作用使用のゲートに接続するメッセージと、参照される相互作用のゲートから接続するメッセージは合致していなければならない。

4）相互作用使用の引数と、参照される相互作用の引数は合致していなければならない。

21.9 RIA の表現例


sd sample4
ユーザー
calc.swf
:CalcServlet
resultxml.jsp
数値を入力
計算ボタンを押す
doPost
Calculator
load(xData, yData)
plus
転送
getResult

22ソフトウェアパターンの利用

22.1 ソフトウェア開発の現状とソフトウェアパターンの必要性

ソフトウェアパターンとは、元来、それなしでもソフトウェア開発が可能だったものであり、ソフトウェア開発にとって絶対的に不可欠な要素ではありません。しかし、情報化社会の進展と軌を一にして、ソフトウェアパターンは急速に普及しつつあります。それは、情報化社会の進展が、以下のような変化を生み出してきたからです。

1）開発規模および複雑性の増大

情報社会化の進展に伴って、システム化の対象が、大規模な業務、複合的な業務にまで広がりつつあり、それによって、システム自体の規模が増大化しつつあります。また、そのようなシステムは、複数のサーバや多くのモジュールから構成されるため、システム全体が複雑化しつつあります。

2）開発期間の短縮

ビジネスの情報化が進展することにより、システム開発がビジネス上の競争関係を反映し、より短期間での製品開発を迫られるようになりつつあります。

3）品質に対する要求の増大

情報システムが社会の様々な領域に浸透している情報社会では、システム・ソフトウェア品質の低下が、社会全体に大きな影響を及ぼす可能性があります。したがって、システム・ソフトウェア品質に対する要求と期待はますます増大しつつあります。

システム開発をめぐる、このような矛盾した状況の中で、個人的な経験知に依存するだけでは、上記の矛盾を克服することは困難です。しかし、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により、システム・ソフトウェアの品質を高く保ちながら開発期間の短縮を図るという矛盾した要求を満たすことが可能になります。

22.2 ソフトウェアパターンの分類

ソフトウェアパターンは、現在、多くのものが認知されていますが、その分類法は確立していません。しかし、その規模(粒度)、抽象度(言語依存度)に応じて、一定の種類に分類すること自体は可能です。ソフトウェアパターンを扱う文献においては、おおむね、以下のように分類されるようです。[64]。

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」などとも呼ばれます。ソフトウェア開発を共同で行う際に、プログラマー間の相互理解のために便宜上定められるものです。必ずしも合理的な理由に立脚しているとは限りません。 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、クラス変数はすべて大文字で記述する、などのようなものがスタイルです。

64 以下参照。
パターンワーキンググループ著・羽生田栄一監修, 『ソフトウェアパターン入門』, ソフト・リサーチ・センター, 2005

Frank Buschmann & Regine Meunier & Hans Rohnert & Peter Sommerlad & Michael Sta, "Pattern-Oriented Software Architecture, A System of Patterns", Wiley, 1996（金澤 典子・桜井 麻里・千葉 寛之・水野 貴之・関 富登志, 『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, 2000）

Gamma, Erich & Johnson, Ralph & Helm, Richard & Vlissides, John, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley Longman, 1995（本位田 真一・吉田 和樹監訳, 『オブジェクト指向における再利用のためのデザインパターン』, ソフトバンク パブリッシング, 1999）

Martin Fowler, "Analysis Patterns: Reusable Object Models", Addison-Wesley Professional, 1996（堀内 一・友野 晶夫・児玉 公信・大脇 文雄, 『アナリシスパターン—再利用可能なオブジェクトモデル』, ピアソンエデュケーション, 2002）

ソフトウェアパターンの分類	意味
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です。 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、フィールドを実装する際のパターンなどが含まれます。
イディオム (idiom)	一般的に実装パターンと同じ意味で用いられます。
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（論理モデル）に関するパターンです。「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ます。そのため、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。「GoF」のデザインパターンが代表例です。
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た、ソフトウェア全体の構造（論理モデル）に関するパターンです。デザインパターン以上に特定の言語への依存度は低くなります。したがって、デザインパターンと同様に、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。 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ます。

ソフトウェアパターンの分類	意味
J2EE パターン	J2EE による Web アプリケーション開発専用のパターンです。実際の内容は、デザインパターンやアーキテクチャパターンの集まりです。
アナリシスパターン (analysis pattern)	概念的に把握されたビジネスプロセス（概念モデル）に関するパターンです。Martin Fowler が考案したものが有名です。

22.3 アーキテクチャパターン

■ 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。

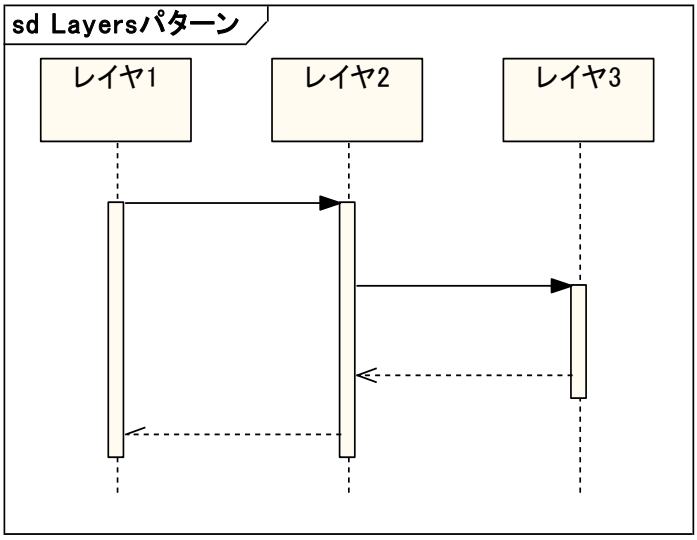
1. 上位のレイヤは、下位のレイヤのサービスを利用する（下位のレイヤは、上位のレイヤに

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)。

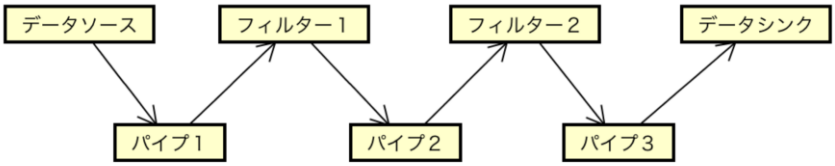
ビジネスロジックレイヤ	業務処理を実現するための処理を行い、データの妥当性のチェックや業務処理を実行する。 ●具体的には以下のような処理を行う必要がある。 ①プレゼンテーションレイヤから取得した入力データに関して、ビジネス上のルールと照らし合わせて妥当性をチェックする。 ②業務処理の実行（プログラムの本来の役割を実行） ③インテグレーションレイヤのクラスのインスタンスを利用して、データの取得やデータの記録などを行う。 ●ビジネスロジックレイヤの処理の例 商品の注文処理 商品の注文キャンセル処理
インテグレーション レイヤ	リソースレイヤへの入力やリソースレイヤからの出力を制御する。 ●具体的には、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。

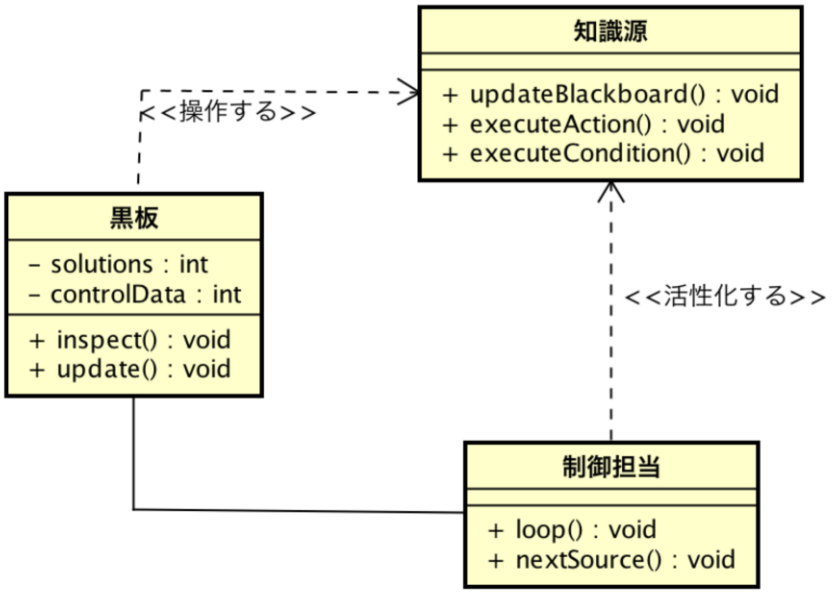
1. Pipes and Filtersパターン

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。



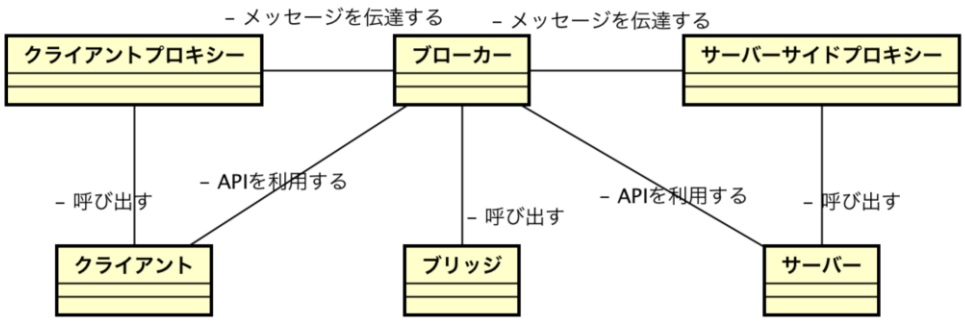
2. Blackboardパターン

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。



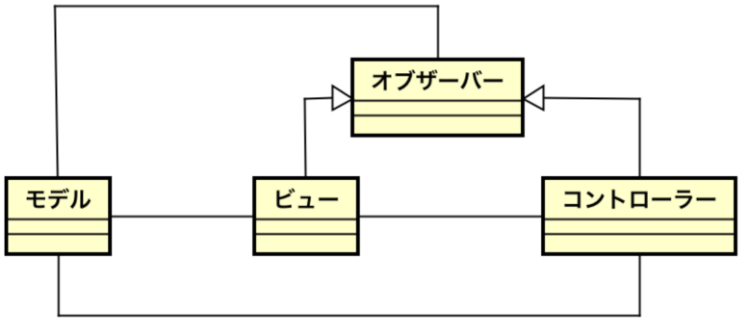
3. Brokerパターン

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）

23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。

23.1 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[65]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、

[65]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。

23.2 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)

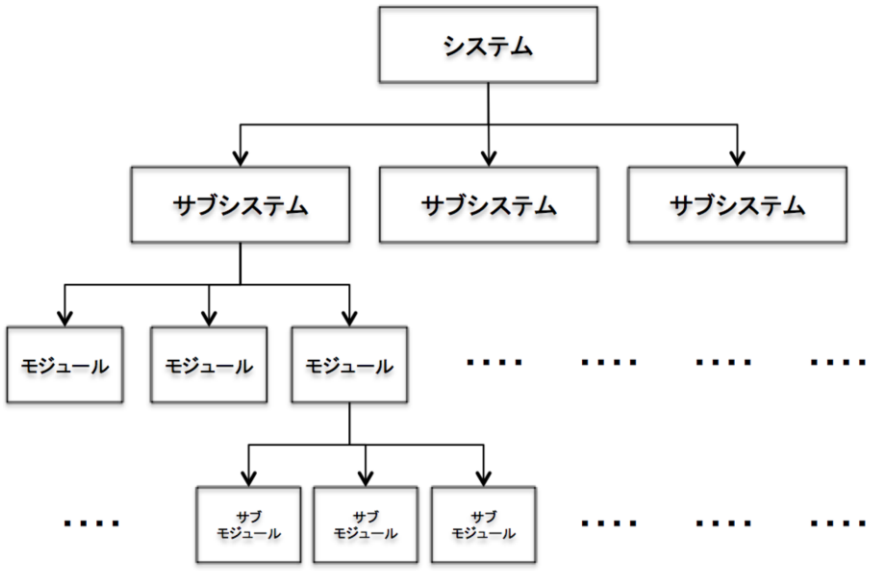
24品質特性駆動型設計

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。

24.1 アーキテクチャ設計の手順

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[66])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

66 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。

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る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。

24.2 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[67]

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

67 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あると考える。

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。

✧ 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

24.3 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。

25品質特性駆動設計によるアーキテクチャ設計

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。

25.1 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。

25.2 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。	

25.3 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

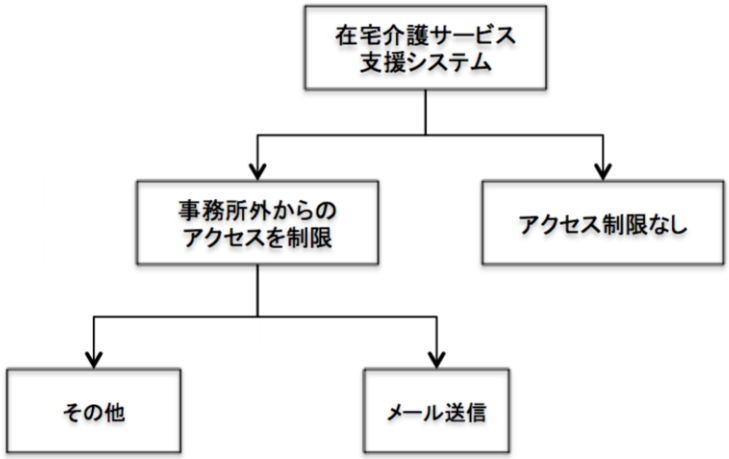
れる事は明らかである。よって、それらのブロック時間が大きい処理を別のスレッドで行う事でブロック時間を可能な限り短くする事が求められると考えた。

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[68]	

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

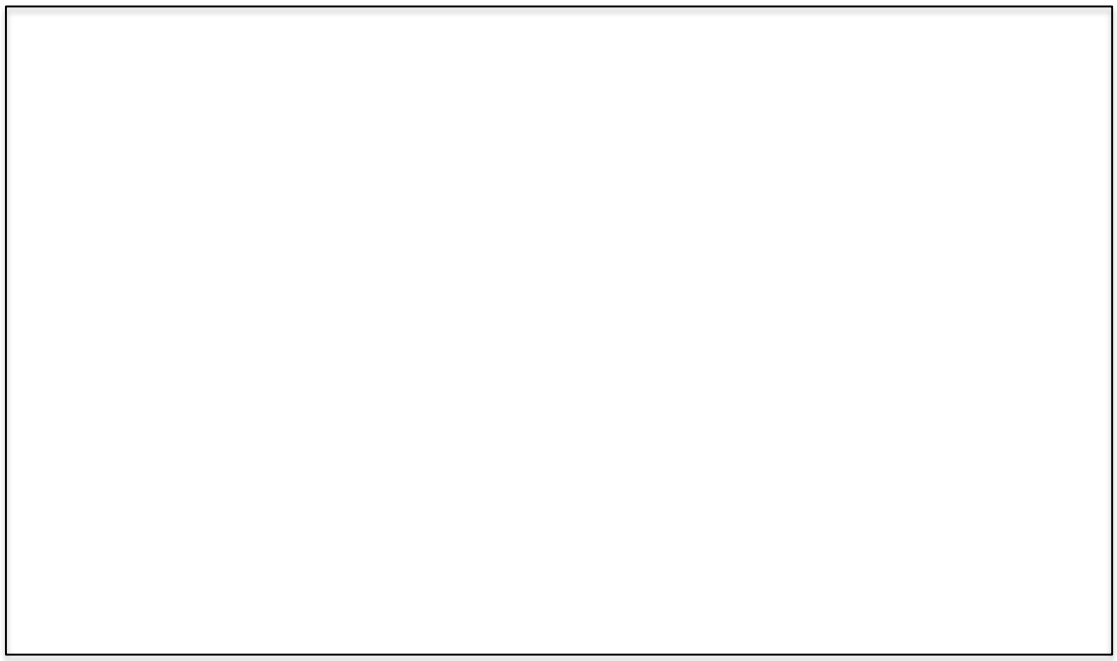
[68]機能性を割り当てる前のモジュールタイプによるシステムの大局的な分割構造を図化したもの。UMLのような特定の記法はなく、全体構造を捉える事が出来て、表現する事が出来ればどのような方法を用いても構わない。

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

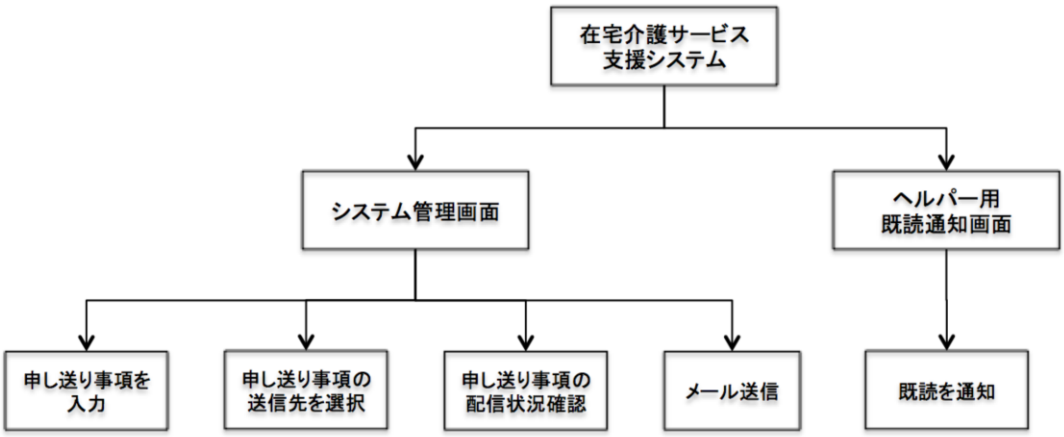


システムの全体構造を記述出来ただろうか？今回は演習という事もあり、シンプルな構成になっており、簡単だという感想を抱くと同時に、これをやる意味があるのだろうかと考える読者の皆様も多かったのではないかと考えられる。一方、実際のシステム・ソフトウェア開発プロジェクトにおいては、求められる規模も比べ物にならない程大きく、必要な機能要件が割り当てられず抜け落ちてしまう事や、全体構造が分からないまま議論されたために必須の機能が、機能要件として抽出されないまま設計が進んでしまう事も多々ある。このように一度表現をして言語化する事で、各ステークホルダーとのコミュニケーションにおける共通言語が出来る上に、全体構造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	

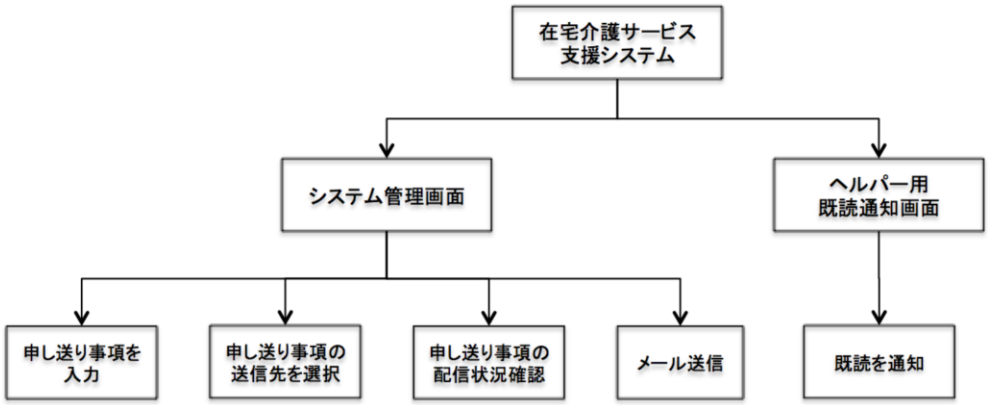
凡例： d=非常に詳細， s=ある程度詳細， o=概要情報

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:

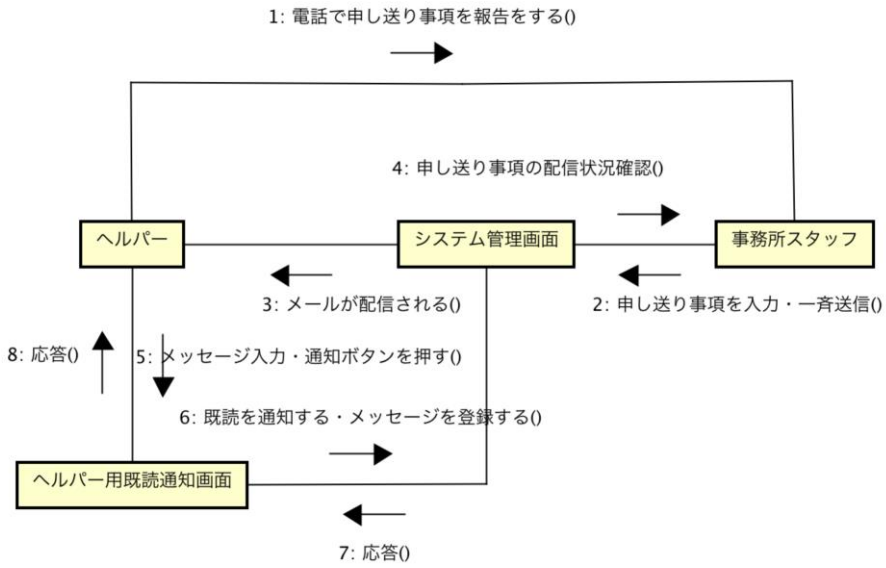
コンテキスト図	
可変性ガイド	
アーキテクチャの背景	
用語集	申し送り事項とは: 申し送り事項の送信要件とは: その他:
その他の情報	

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[69]	「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。

69 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

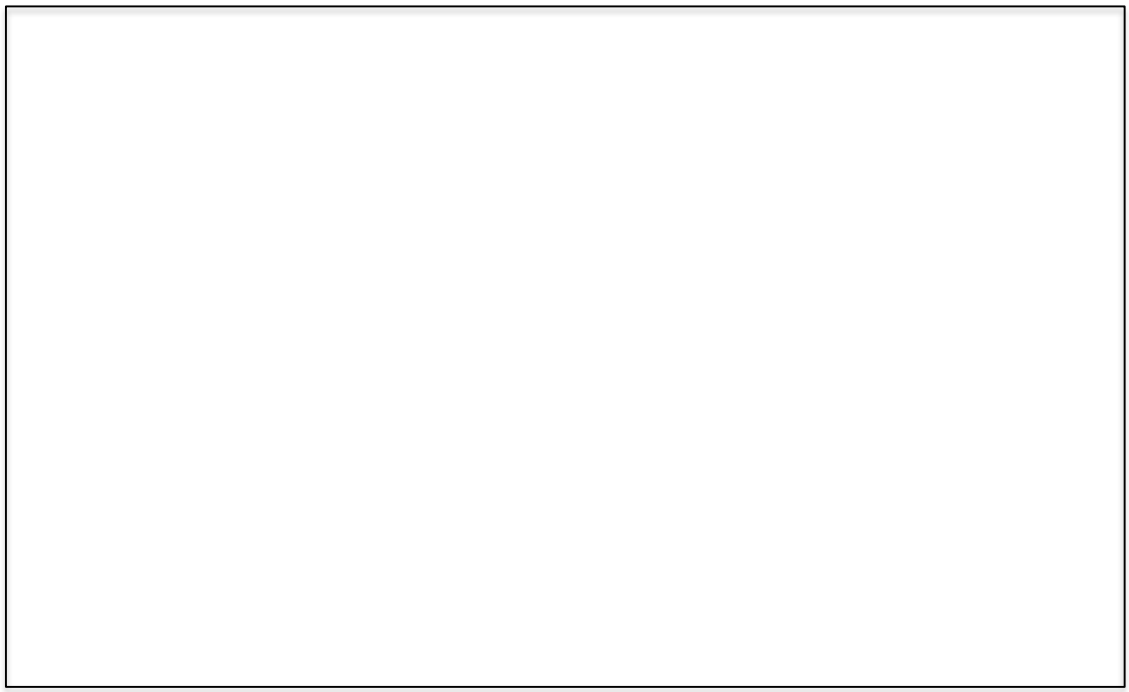
その他の情報	2017/02/xx　Ver1.0　Initial Draft　作成者：xxxx

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[70]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。

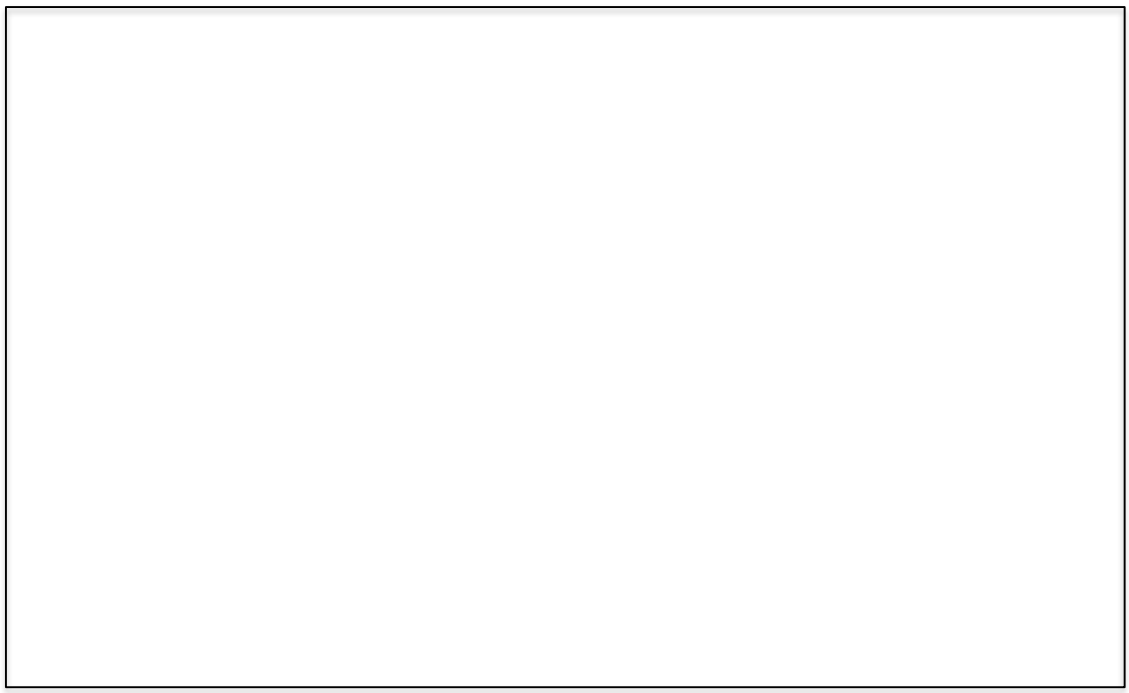
最後に、開発工程に入るのに辺り重要な、クライアントサーバービューとクラスビュー（クラス図）がどのような形になるか、最終形態を考える事を持って演習を終了とする。可能であれば、先程の続きから（もしくは自分で最初から）「品質特性駆動型設計」のモジュール分割を再帰的に行い、詳細設計まで踏み込んでみて欲しい。

70 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

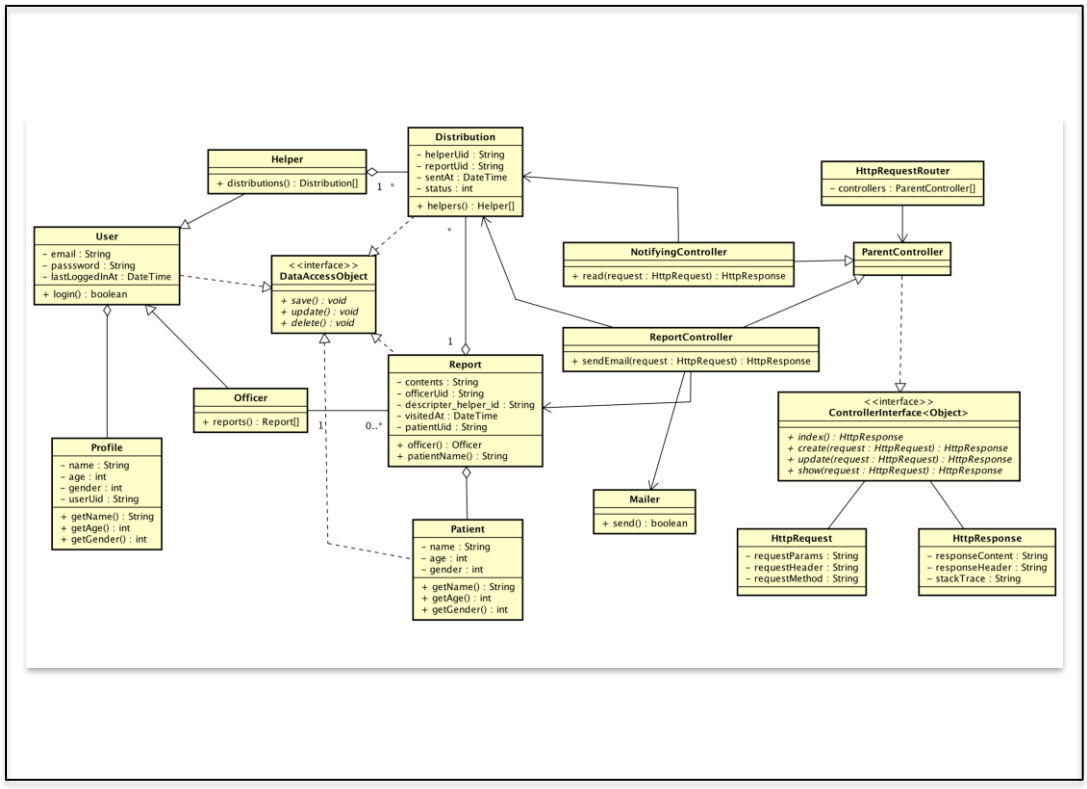


クライアントサーバービュー

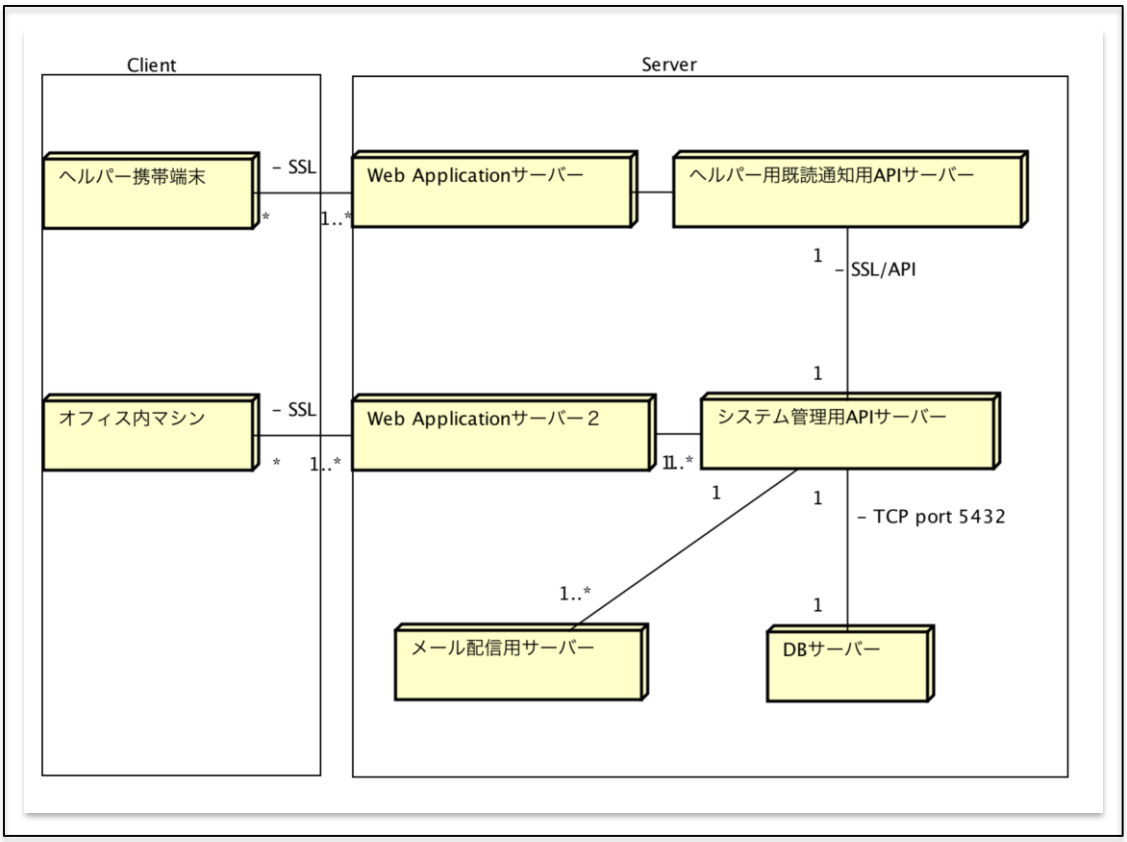


アーキテクチャ設計

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
SSL/API
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
TCP port 5432
メール配信用サーバー
DBサーバー

26詳細設計

26.1〈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[71]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。)[72]

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[73])、プログラムの〈階層構造

---

71 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。

72 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)

73 「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

化〉(モジュールの階層構造化[74])、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。

> 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)[75]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[76]。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〉

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。

74 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。

75 〈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 ）。

76 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。

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。

26.2 オブジェクト指向設計の5原則

「オブジェクト指向設計の原則」とは、米国の Objectmentor 社[77]の Robert C. Martin が、さまざまなソフトウェア技術研究者によるオブジェクト指向研究の成果を総括し、以下の5つの原則にまとめたもののことです。

単一債務の原則（SRP：Single Responsibility Principle）
開放-閉鎖原則（OCP：Open-Closed Principle）
リスコフ置換の原則（LSP：Liskov Substitution Principle）
依存関係逆転の原則（DIP：Dependency Inversion Principle）
インターフェイス分離の原則（ISP：Interface Segregation Principle）

このなかでも、彼が「オブジェクト指向設計のゴール」と呼んでいる最も重要な原則が、「開放-閉鎖原則」です。

26.3 開放-閉鎖原則とは

「開放-閉鎖原則」は、もともとは、オブジェクト指向研究の第一人者である Bertrand Meyer(バートランド メイヤー)の研究成果[78]に由来しますが、この原則を解釈し、他の 4 原則と並べてオブジェクト指向設計の 5 原則としてまとめたのは、Robert C. Martin です。彼は、以下のように「開放-閉鎖原則」を定義しています。

> ソフトウェアの構成要素（クラス、モジュール、関数など）は拡張に対して開いて（オープン：Open）いて、修正に対しては閉じて（クローズド：Closed）いなければならない。
> (『アジャイルソフトウェア開発の奥義』, p.128) [79]

---

77 Objectmentor 社のサイトの URL は http://www.objectmentor.com です。

78 Meyer, Bertrand, 1988, "Object-Oriented Software Construction", Prentice Hall（二木 厚吉 監訳, 酒匂 寛・酒匂 順子 訳, 2005, 『オブジェクト指向入門』, アスキー出版局）

79 Martin, Robert C., " Agile Software Development Principles, Patterns and Practices", Prentice Hall, 2003（瀬谷 啓介訳, 『アジャイルソフトウェア開発の奥義』, ソフトバンク クリエイティブ, 2004）

これは、一見矛盾するような次のふたつの条件を同時に満たすことを意味します。

1) ソフトウェアの仕様や要件定義に変更が生じた場合、ソフトウェアの振る舞いを変更して対応することができる。

2) ソフトウェアの既存のソースコードの変更や再コンパイルを行う必要はない。

通常、ソフトウェアの処理を変更するためには、ソースコードの変更および再コンパイルが必要です。したがって、上記の1)、2)を同時に満たすことは、一見すると矛盾であることになります。

しかし、この原則は、次のような仕方で実現することが可能です。

ソフトウェアの仕様や要件定義に生じた変更に対して、クライアントを始め既存のソースコードを変更することなく、新しいサプライアを追加することによってソフトウェアの振る舞いを変化させ対応する。

●開放-閉鎖原則に違反するソースコード

```
class OracleDao {

    private String dbName; // データベース名

    public String getDbName() {
        return dbName;
    }

    public void setDbName(String name) {
        dbName = name;
    }

    public void connect() {

        // Oracleへの接続処理
```

```
            System.out.println("Oracleに接続");

        }

        public void execute() {

            // Oracleに対するSQLの実行

            System.out.println("Oracleに対してSQLの実行");

        }
    }

    class MySQLDao {

        private String dbName; // データベース名

        public String getDbName() {
            return dbName;
        }

        public void setDbName(String name) {
            dbName = name;
        }

        public void connect() {

            // MySQLへの接続処理

            System.out.println("MySQLに接続");

        }
```

```
        public void execute() {

            // MySQLに対するSQLの実行

            System.out.println("MySQLに対してSQLの実行");

        }
    }

    class Executer {

        private OracleDao dao;

        public void setDao(OracleDao dao) {

            this.dao = dao;

        }

        public Executer(OracleDao db) {

            this.dao = db;

        }

        public void doDao() {

            this.dao.connect();
            this.dao.execute();

        }
    }
```

```
class Test {

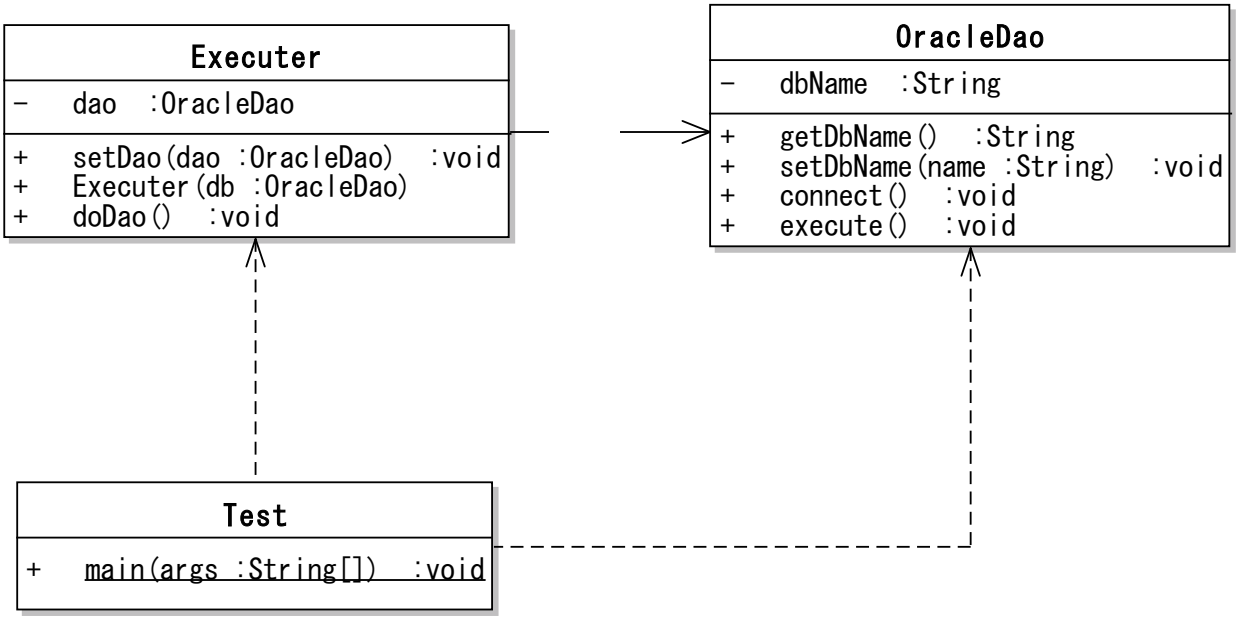
    public static void main(String args[]) {

        Executer e1 = new Executer(new OracleDao());

        e1.doDao();

    }
}
```

●クラス図


Executer
- dao :OracleDao
+ setDao(dao :OracleDao) :void
+ Executer(db :OracleDao)
+ doDao() :void
OracleDao
- dbName :String
+ getDbName() :String
+ setDbName(name :String) :void
+ connect() :void
+ execute() :void
Test
+ main(args :String[]) :void

●開放-閉鎖原則に合致するソースコード

```
abstract class Dao {

    private String dbName; // データベース名

    public String getDbName() {
        return dbName;
    }

    public void setDbName(String name) {
        dbName = name;
    }

    public abstract void connect(); // 抽象メソッド

    public abstract void execute();// 抽象メソッド
}

class OracleDao extends Dao {

    public void connect() {

        // Oracleへの接続処理

        System.out.println("Oracleに接続");

    }

    public void execute() {

        // Oracleに対するSQLの実行

        System.out.println("Oracleに対してSQLの実行");
```

```
    }
}

class MySQLDao extends Dao {

    public void connect() {

        // MySQLへの接続処理

        System.out.println("MySQLに接続");

    }

    public void execute() {

        // MySQLに対するSQLの実行

        System.out.println("MySQLに対してSQLの実行");

    }
}

class Executer {

    private Dao dao;

    public void setDao(Dao dao) {

        this.dao = dao;

    }
```

```
    public Executer(Dao db) {

        this.dao = db;

    }
    public void doDao() {

        this.dao.connect();
        this.dao.execute();

    }
}

class Test {

    public static void main(String args[]) {

        Executer e1 = new Executer(new OracleDao()); // 変更部分

        e1.doDao();

    }
}
```

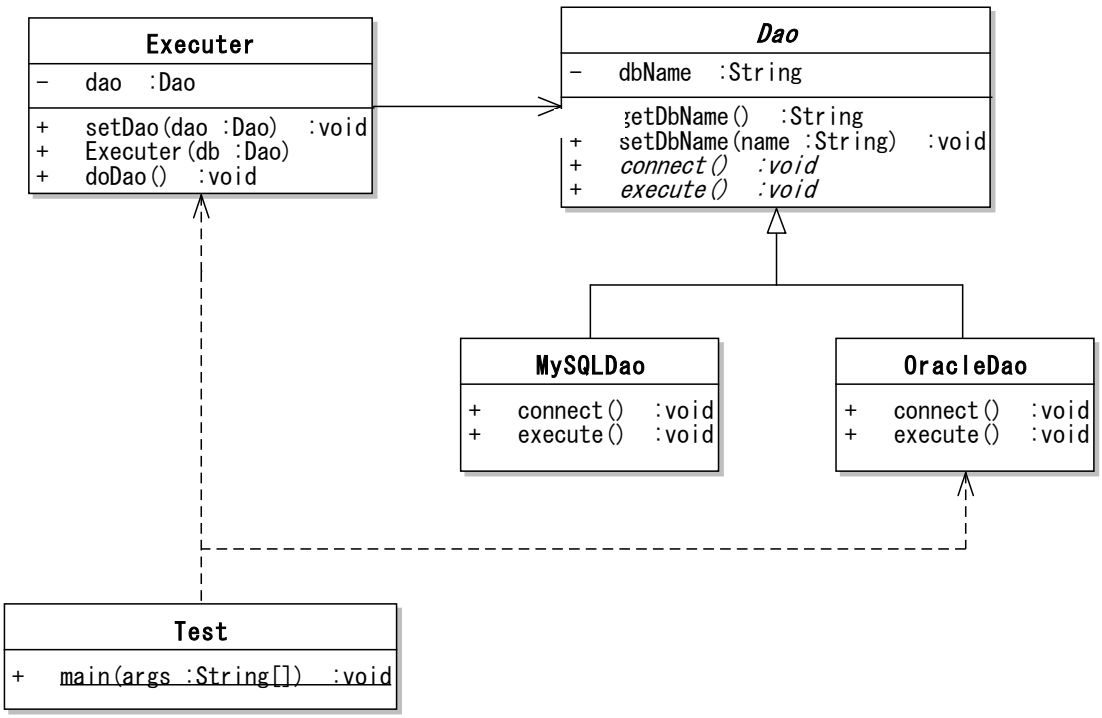
実行結果

```
Oracleに接続
Oracleに対してSQLの実行
```

上記変更部分を「Executer e1=new Executer(new MySQLDao());」と変更した場合は、以下のような結果になります。

```
MySQLに接続
MySQLに対してSQLの実行
```

●クラス図



●開放－閉鎖原則への合致

この場合、クライアント（および抽象クラス、既存のサプライア）は、「閉じて」います（修正しない）。しかし、新しいサプライアを追加することで、ソフトウェア（クライアント＋サプライア）の動作を変化させることができます（「開いている」）。つまり、拡張に対しては「開いて」いるが、修正に対しては「閉じて」います。

●開放－閉鎖原則と結合度

開放－閉鎖原則を適用することにより、クライアントとサプライアの間の結合度を下げることができます。開放－閉鎖原則は、凝集度ではなく結合度にかかわる原則であることが分かります。

●注意点

このサンプルコードでは、サプライアを取り替える場合、クライアント Test は、変更を行わなければならないため、開放-閉鎖原則を完全に満たしているわけではありません。

26.4 開放－閉鎖原則に合致するための設計上の注意点

開放－閉鎖原則に合致するためには、次のような設計上のポイントがあります。

1）クライアントは、具象クラスとなるサプライアの型を直接参照せずに、複数のサプライアに共通なスーパークラス（抽象クラス）やインターフェイスの型を参照する。

2）具象クラスからなるサプライアは、クライアントに対して実装を隠蔽しなければならない。

2）にもかかわらず、クライアントに対して具象サプライアを提供するコードは、サプライアの取り替えの際に修正が必要になるという問題が生じる。

26.5 単一債務の原則の概要

単一債務の原則は、以下のような原則です。

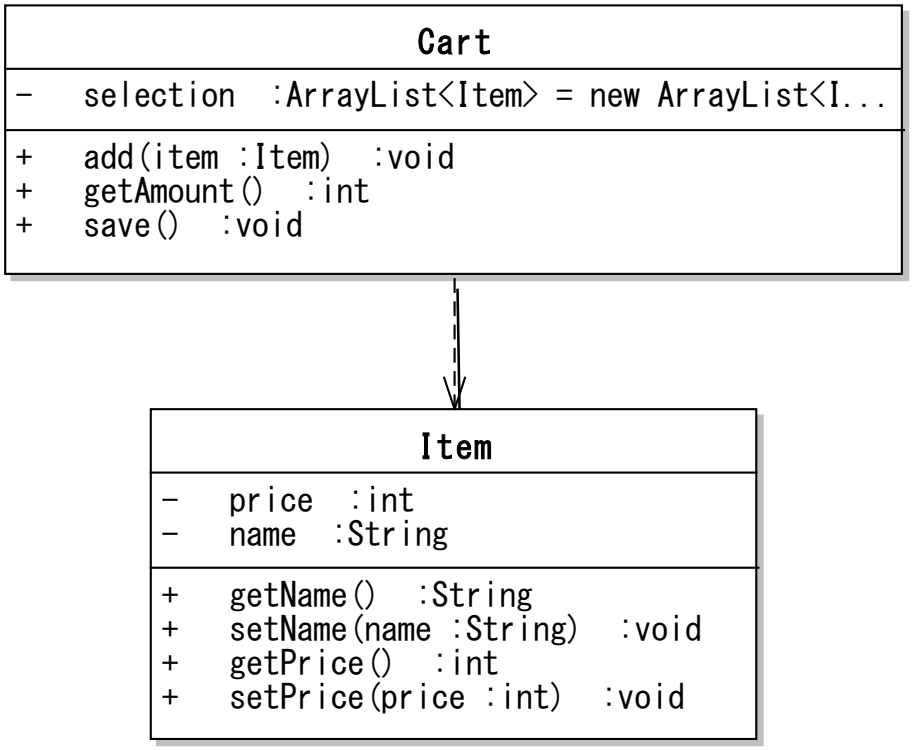
クラスを変更する理由はひとつでなければならない＝クラスの役割はひとつでなければならない。

ただし、クラスが複数の役割を持ち、その複数の役割が常に結びついている、関連性が高い（＝複数の役割が同じ理由でいっしょに変更される）ものであるならば、そのクラスに複数の役割を持たせても良い。

26.6 違反した事例

次のように多くの役割を持ったクラスを設計しないこと。ひとつのサプライアに多くの機能を持たせないこと。以下は違反例です。

●クラス図



●ソース

```
import java.util.ArrayList;

class Cart {

    private ArrayList<Item> selection = new ArrayList<Item>();

    // 商品をカートに追加する

    public void add(Item item) {
        selection.add(item);
    }

    // 合計金額を返す

    public int getAmount() {

        int amount = 0;// 合計金額を格納する

        for (Item item : this.selection) {
            amount += item.getPrice();// 価格を取得して加算
        }

        return amount;
    }

    public void save() {

        // cartの内容を、JDBCによってデータベースに
        // 書き込む
    }

}
```

```
class Item {

    private int price;

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }
}
```

クラスに含まれるフィールドやメソッドが、同じ役割に含まれるかどうかは、文法的に決定することはできません。同じ役割に含まれるかどうかは、次のような基準で判断することができます。

フィールドやメソッドが、同じ理由によって、いっしょに変更されることが多い。

この例では、商品を意味するクラスを、Itemクラスから Product クラスに変更するといったことを行った場合は、add メソッドも、getAmount メソッドも同時に変更する必要がある。一緒に変更されることが多いメンバーは、同じ役割に含まれると考えられる。

26.7 違反事例における問題点

Cart クラスには、ショッピングカートとしての役割(選択された商品(Item)を保持し、合計金額を計算するという役割)と、永続化(persistence)の役割(選択された商品の情報をデータベースに書き込むという役割)が存在します。

ショッピングカートとしての役割は、「オンラインショッピング」という問題領域における業務上のルール(商品はいくつまで選択できるか、そのユーザーはいくらまで購入することができるかなど)の変更に伴って変化する可能性があります。他方、データベースに書き込みを行う役割は、使用するデータベースの変更や、保存先がデータベースからXML 文書に変更されるといったことにより、変化する可能性があります。

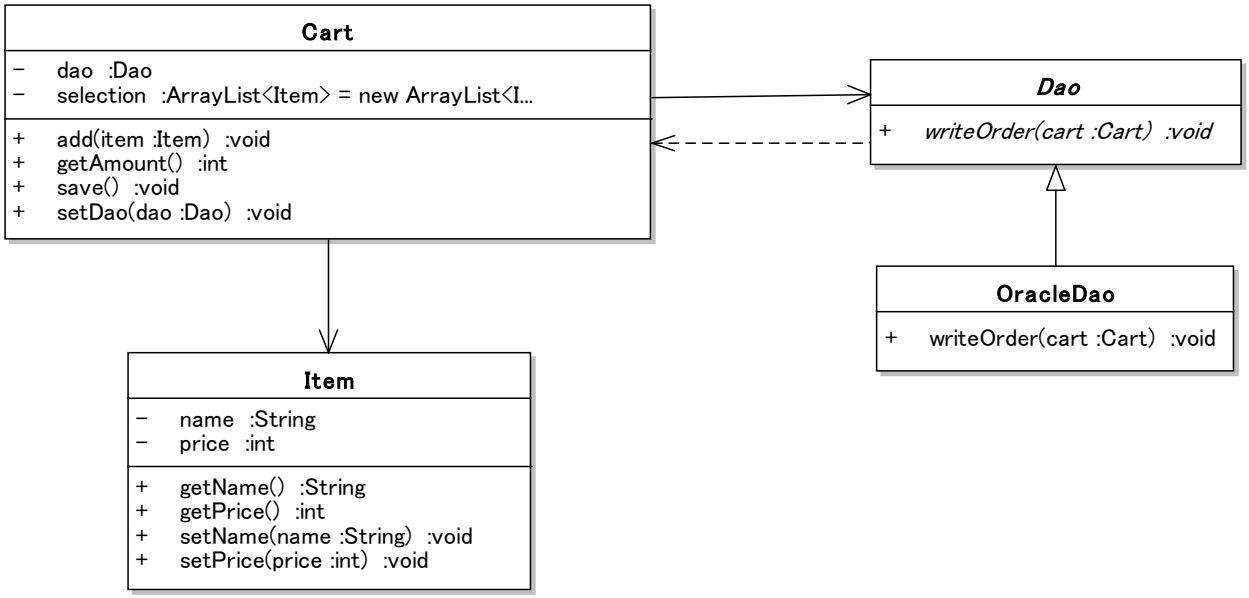
この場合、以下のような問題が生じます。

このクラスには、変更の理由＝役割が2つ存在しており、このような場合、一方の役割の変化に伴ってソースが変更された場合、他方の役割が正常に動作しなくなる可能性があります。一方の役割の変化に伴うソースの変更の後で、他方の役割が以前と同様に正常に動作するかどうかチェックする必要が生じます。

26.8　設計の修正

次のように、Oracle のようなデータベースにアクセスする処理を、他のクラスに分離します。

●クラス図



●ソース

```
import java.util.ArrayList;

class Cart {

    private Dao dao;

    public void setDao(Dao dao) {

        this.dao = dao;
    }
```

```
    private ArrayList<Item> selection = new ArrayList<Item>();

    // 商品をカートに追加する

    public void add(Item item) {

        selection.add(item);
    }

    // 合計金額を返す

    public int getAmount() {

        int amount = 0;// 合計金額を格納する

        for (Item item : this.selection) {
            amount += item.getPrice();// 価格を取得して加算
        }

        return amount;
    }

    public void save() {

        dao.writeOrder(this);

    }
}
```

```
abstract class Dao {

    public abstract void writeOrder(Cart cart);
}

class OracleDao extends Dao {

    public void writeOrder(Cart cart) {

        // Oracleにデータを保存する処理

    }

}

class Item {

    private int price;

    private String name;

    public String getName() {

        return name;
    }

    public void setName(String name) {

        this.name = name;
    }

    public int getPrice() {

        return price;
```

```
    }

    public void setPrice(int price) {

        this.price = price;
    }
}
```

SRP は、クラスに関して、凝集度（cohesion）を高めるという原則である。凝集度が高められたメソッドは、ひとつの機能だけをもつように、SRP が適用されたクラスは、ひとつの役割だけをもつ。これに違反した場合は、次のような問題が発生する可能性がある。

① 方の役割の変化に伴うソースコードの変更が、他方の役割を構成するソースコードを変更してしまい正常に動作しなくなる可能性がある（正常動作の再確認が必要になる）。

②関連性のないふたつの役割がクラスに含まれる場合、同様の役割が別のクラスにも含まれる可能性がある（ソースコードの重複が発生する可能性）。これは、メソッドに関する凝集度が低い場合と同様の問題である。

26.9 スーパークラスとサブクラス

継承において、スーパークラスとサブクラスの間には、次のような関係がありました。

1）スーパークラス（共通要素の実装）　　サブクラス（差分の実装）

2）スーパークラス（上位概念）　　サブクラス（下位概念）

さらに、その結果として、サブクラスのインスタンスをスーパークラス型に置換し、サブクラスの型をクライアントに対して隠蔽できるという効果が得られました。

3）スーパークラス型　←置換可能　　サブクラスのインスタンス

リスコフ置換の原則は、上記1）、2）、3）のうち、とくに3）に関連する原則です。

26.10 リスコフ置換の原則の概要

リスコフ置換の原則[80]は、以下のように定義されています。

> 派生型はその基本型と置換可能でなければならない。
> （『アジャイルソフトウェア開発の奥義』, p. 144）

これは、次のことを意味しています。

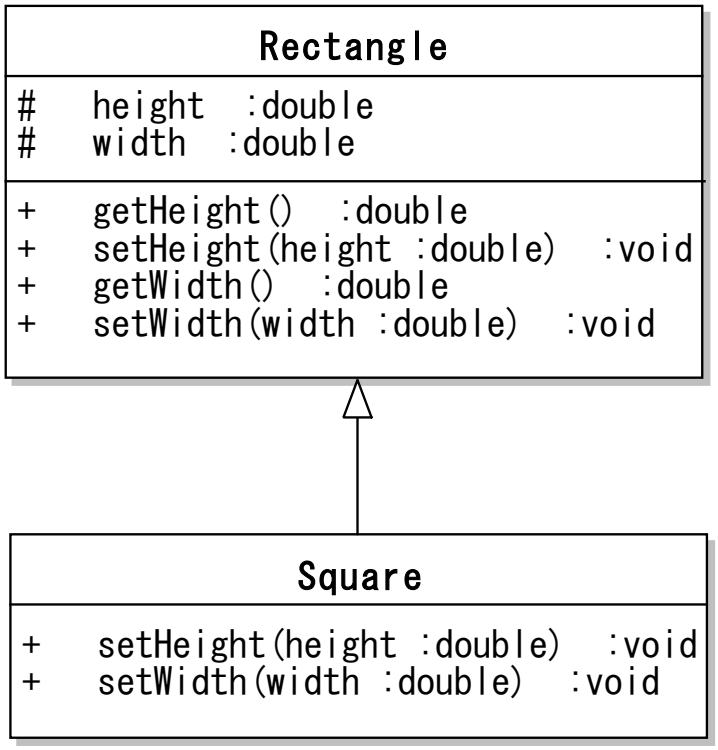
> クライアントがスーパークラスを利用するような関係が存在する場合、クライアントに、どのような具象サブクラスのサプライアを渡しても、正常に動作するのでなければならない。あるいは、クライアントが、サブクラスである複数のサプライアの違いを無視して利用できるように、各サブクラスを作成しなければならない。

リスコフ置換の原則に違反した場合、開放-閉鎖原則にも違反します。開放-閉鎖原則に合致するためには、リスコフ置換の原則に合致しておく必要があります。

80　「リスコフ」とは、マサチューセッツ工科大学教授の Barbara Liskov のことです。リスコフ置換の原則は、リスコフ女史が 1988 年に発表した論文（"Data Abstraction and Hierarchy", SIGPLAN Notices, 1988）の成果に準拠するため、彼女の名前を原則名に採用しています。彼女を紹介するマサチューセッツ工科大学のホームページの URL は http://www.pmg.csail.mit.edu/~liskov/です。

26.11 リスコフ置換の原則の考え方

●クラス図



この場合、Rectangle クラスは長方形を、Square クラスは、その具象サブクラスで、正方形を表します。

●ソース

```
class Rectangle {

    protected double height;
    protected double width;

    public double getHeight() {

        return height;

    }
```

```
    public void setHeight(double height) {

        this.height = height;

    }

    public double getWidth() {

        return width;

    }

    public void setWidth(double width) {

        this.width = width;

    }
}

class Square extends Rectangle {

    public void setHeight(double height) {

        this.height = height;
        this.width = height;
    }

    public void setWidth(double width) {

        this.height = width;
        this.width = width;
    }
}
```

この例では、正方形は高さと幅が同じ長さの長方形である(正方形は長方形の一種である)という意味で、一見、正当な継承のように見えるが、次のようなクライアントが存在していた場合、問題が生じます。

```
class Client {
    public double getArea(Rectangle r, double height, double width) {

        // 高さと幅を設定する

        r.setHeight(height);
        r.setWidth(width);

        // 面積を求めて返す

        return r.getHeight() * r.getWidth();
    }
}
```

このクライアントは、サプライアが長方形(Rectangle)の具象サブクラスであることを想定して、高さと幅から面積を求め戻り値として返すというメソッドを持っています。このクライアントの実装では、Square クラスのインスタンスをサプライアとした場合、第二引数の高さが無視され、幅の2乗として面積が計算されてしまう。また、この場合、第二引数と第三引数の順序を入れ替えると面積が変わってしまいます。これは、クライアントの作成者にとって想定外のことです。通常の面積ならば、このメソッドの第二引数と第三引数の値を入れ替えても面積は同じですが、Square のインスタンスの場合は事情が異なります。ということは、第一引数に Square のインスタンスが渡されてきた場合は、第二引数と第三引数の値が等しいかどうか、クライアントの内部でチェックする必要があるということになります。

しかし、クライアントは、サプライアがどのような具象サブクラスのインスタンスであるのか、その型および実装を知っているべきではありません。クライアントに対して、具象サブラクスの型や実装は隠蔽されているべきだからです。問題は、スーパークラスである長方形(Rectangle)(およびそのサブクラス)に対して一定の想定を前提としてクライアントを作成した場合、不適切な継承およびサブクラスの実装は、クライアントを作成した開発者にとって想定外の現象を引き起こしうるということです。

つまり、スーパークラスからの継承が適切であるかは、長方形-正方形という概念的な「IS-A」関係(上

位概念-下位概念の関係)からだけではなく、スーパークラスを利用するクライアントに対して、そのサブクラスのインスタンスを渡しても動作に問題が生じないか、という観点から判定する必要があります。それが、リスコフ置換の原則の意味です。つまり、ある継承の妥当性(validity)は、クライアントとの関係によって決定されるということです。したがって、リスコフ置換の原則は、単なる設計の原則にとどまらず、継承において、サブクラスの妥当な実装方法に関する原則でもあります。

26.12 継承の妥当性の具体的基準となるもの

それでは、ある継承が、クライアントにとって妥当かどうかを判定するには、どのような考え方に基づいて、クライアントおよびスーパークラス-サブクラスを作成すればよいのかという問題が残ります。その答えは、「契約による設計(Design By Contract:DBC)」です。これは、クライアントとサプライアの間には、一定の契約(相互に束縛するルール)があり、そのような契約に基づいて作成されなければならないという考え方であり、この考え方を最初に提唱したのは、Bertrand Meyer です。

> ルーチンを再定義する場合、事前条件はより弱い事前条件のみと、事後条件はより強い事後条件のみと置き換えてもよい。
>
> (二木 厚吉監訳, 酒匂 寛・酒匂 順子 訳, 『オブジェクト指向入門』, p.345)[81]

事前条件と事後条件については、OCJ-P の「アサーション」の項目で学習していますが、実は、あのアサーションは、Bertrand Meyer の「契約による設計」の考え方を Java において実現するために加えられた機構です。

●事前条件

事前条件とは、クライアントがサプライアのメソッドを呼び出すための条件を意味します(一般的には引数に関する条件)。

●事後条件

事後条件とは、サプライアのメソッドが処理を完了したときにクライアントに対して保証しなければならない条件を意味します。

リスコフ置換の原則を守るためには、スーパークラスを継承する際に、サブクラスは、「契約による設計」の考え方に基づいてオーバーライド(実装)を行わなければならない。つまり、次のようにオーバーライドする必要があります。

---

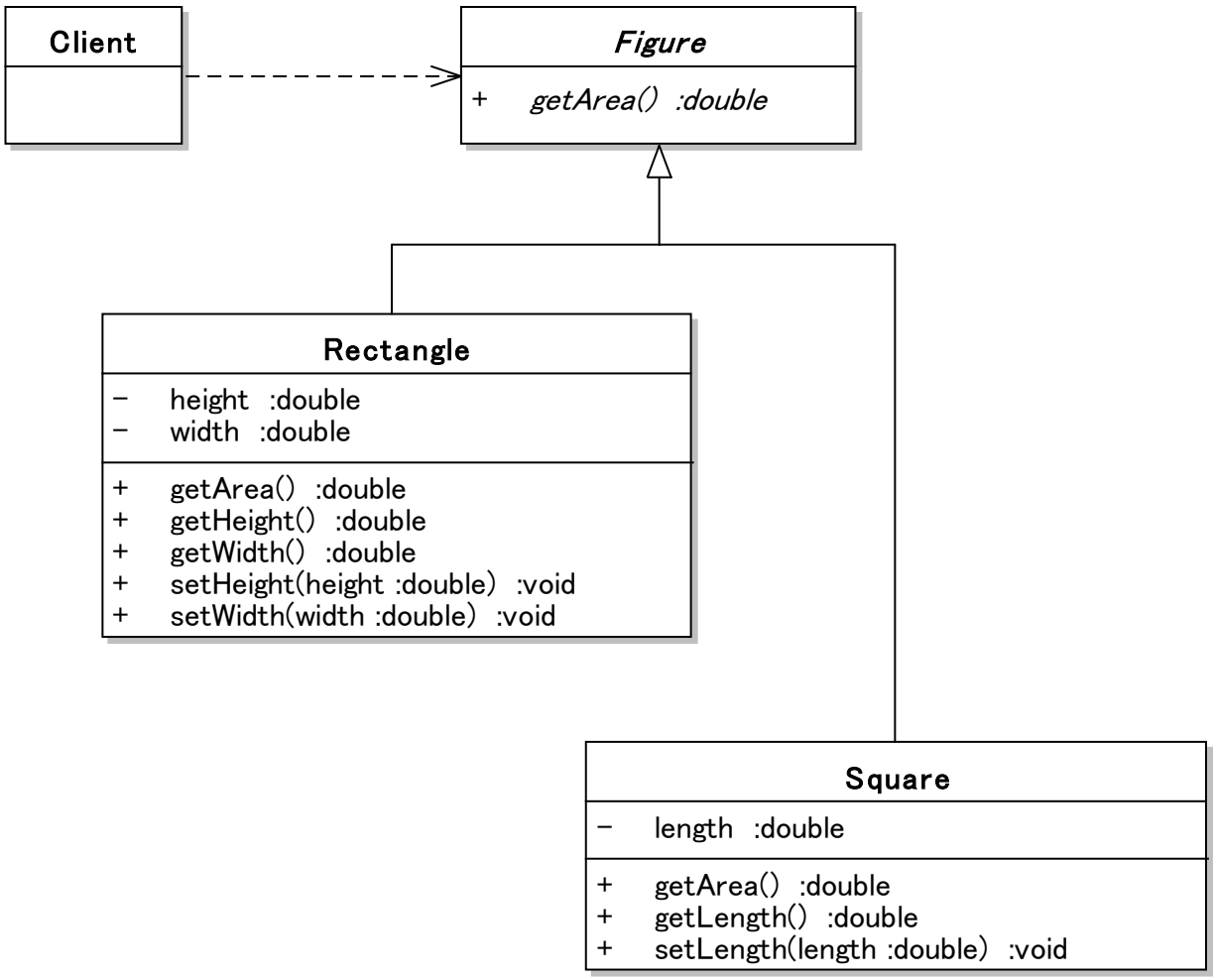
81 Meyer, Bertrand, 1988, "Object-Oriented Software Construction", Prentice Hall (二木 厚吉監訳, 酒匂 寛・酒匂 順子 訳, 2005, 『オブジェクト指向入門』, アスキー出版局)

サブクラスは、オーバーライドに際して、スーパークラスがクライアントに対して保証する事後条件と同じか、より強い事後条件を満たすようにオーバーライドしなければならない。

先ほどの Rectangle の例では、クライアントである Client クラスに対して保証しなければならない setWidth メソッドの事後条件が、インスタンス変数 width を変更しても、height は変化しないとなっていました。setHeight メソッドの事後条件も同様です。しかし、Square クラスの setWidth メソッドや setHeight メソッドはこの事後条件を満たさないために、リスコフ置換の原則に違反したのです。

26.13 リスコフ置換の原則に基づいて修正した設計

●クラス図



Rectangle と Square を全く別の継承階層に分離するか、上記のように、抽象クラスを導入し、RectangleとSquareを継承させます。この場合は、上位概念-下位概念というだけでなく、getAreaメソッドのオーバーライドに関して、その事後条件を維持するように(同じ条件かより厳しい条件によって)オーバーライドを行っています。

●クライアントのイメージ

```
class Client {

    public double getArea(Figure f) {

        // 面積を返す

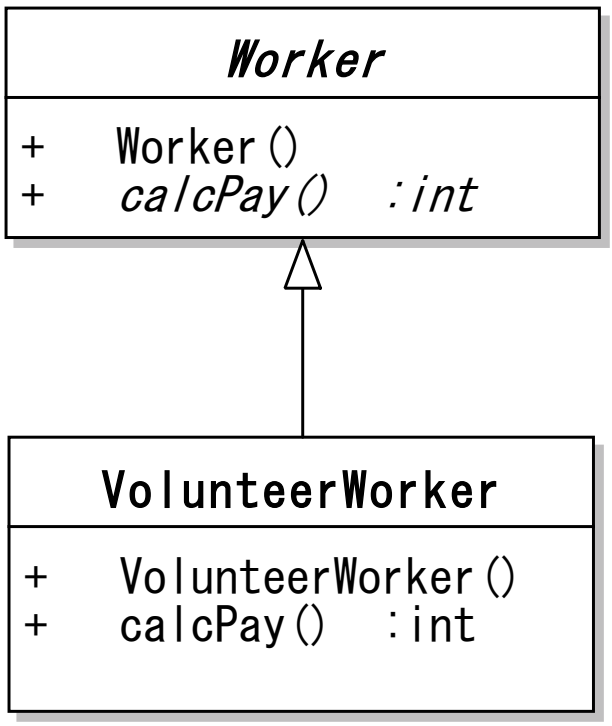
        return f.getArea();

    }
}
```

26.14 その他の違反事例

次のような場合もリスコフ置換の原則に違反する。

●クラス図



●ソース

```
abstract class Worker {

    public Worker() {

    }

    public abstract int calcPay();

}
```

```
class VolunteerWorker extends Worker {

    public VolunteerWorker() {

    }

    public int calcPay() {

        return 0;
    }
}
```

VolunteerWorker の calcPay メソッドが存在し、0 という戻り値を返すということは、VolunteerWorker にも給与を払う必要があることを意味します。しかし、実際には無意味です。実際には、サブクラスである VolunteerWorker における calcPay メソッドは、そもそも必要ないメソッドであり、「退化」したメソッドといいます。サブクラスにおいて「退化」しているメソッドは、リスコフ置換の原則に違反する設計の特徴の 1 つです(『アジャイルソフトウェア開発の奥義』, p.161)。

先ほどの事後条件という考え方を当てはめると、クライアントに対して Worker の calcPay メソッドが保証する事後条件は、0 よりも大きな整数値をクライアントに返すことと考えられます。それが、クライアント側の想定です。その場合、0 を返すというのは、事後条件に反することになります。したがって、リスコフ置換の原則に違反します。

また、例外に関しても、抽象クラスのメソッドが投げない例外を、具象サブクラスのメソッドで送出する場合、抽象クラスの事後条件に違反します。つまり、次のように記述するのも、リスコフ置換の原則に違反します。

```
abstract class Worker {

    public Worker() {

    }
```

```
    public abstract int calcPay();

}

class VolunteerWorker extends Worker {

    public VolunteerWorker() {

    }

    public int calcPay() {

        // 使用する意味がないため、
        // 呼び出すと例外を送出する

        throw new RuntimeException();
    }
}
```

クライアントに対して Worker の calcPay メソッドが保証する事後条件に、呼び出された後で例外を送出するというものが含まれないからです。

26.15 アプリケーションの「上位レベルの方針」

以下のプログラムが意味することを考えてみます。

```
class Client {

    private String data;

    public Client(String data) {
        this.data = data;
    }

    public void execute(Writer w) {

        w.open();
        w.write(data);
        w.close();

    }
}

interface Writer {

    void open();

    void write(String data);

    void close();
}
```

この2つの部品によって、このアプリケーションの内部で、文字列を「書く」という方向性が決定されています。どこに(ファイルか、XML 文書の中か、リレーショナルデータベースのテーブルの中か)書くかは、決定されていないが、文字列データを「書く」ということだけは決定されています。つまり、「書く」ことの詳細は不明だが、「書く」という「方針」だけは決定されている、ということができます。

このとき、クライアント内で記述されているwriteメソッドは、具体的なwriteメソッドを意味していないことが分かります。クライアント内で呼び出されている write メソッドはインターフェイス内の抽象メソッドを意味しています。

ここで多態性は、抽象メソッドの呼び出しだけを方針として事前に決定しておいた上で、実際に動作する際に、何らかの具象サプライア(この場合はWriter インターフェイスの実装クラスのインスタンス)が渡され、その具象メソッドが呼び出されるというメカニズムとして働きます。これにより、クライアントは、色々な具象サプライアに対応することができます。

つまり、クライアントとインターフェイス/抽象クラスによって、そのアプリケーションの「方針」(policy)が事前に決定されます。さらに、具象サプライアがクライアントに渡されることにより、その「方針」が、一定の具体的な手段により実行されます。上記の例では、まず、「文字列データを書く(記録する)」という方針が事前に決定され、その上で、実行時に具象サプライアを利用することにより、文字列が具体的な手段(ファイルやXML 文書、データベースのテーブル内など)を通して、書かれ(記録され)ます。

クライアントとインターフェイス/抽象クラスによって、アプリケーションの方針が事前に決定され、具象クラスのサプライアが、その具体的な手段を提供します。

Robert C. Martinは、クライアントとインターフェイス/抽象クラスによって構成され、事前に決定されるアプリケーションの方針を「上位レベルの方針(high level policy)」、具体的な手段となる具象サプライアを「下位のモジュール」、「実装の詳細」と呼んでいます(『アジャイルソフトウェア開発の奥義』, p168)。

26.16 依存関係逆転の原則

依存関係逆転の原則は、次のように規定されています。

a. 上位のモジュールは、下位のモジュールに依存してはならない。どちらのモジュールも「抽象」に依存するべきである。

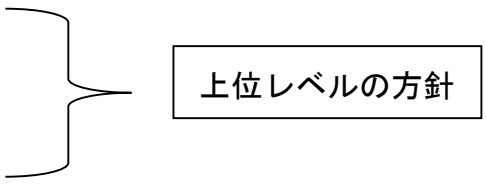
b. 「抽象」は実装の詳細に依存してはならない。実装の詳細が「抽象」に依存すべきである。
(『アジャイルソフトウェア開発の奥義』, p.163)

これは、次のような意味である。

プログラムの中で、「上位レベルの方針(high level policy)」に、具象クラス(サプライア)が従うのでなければならない、上位レベルの方針が具象クラス(サプライア)に従ってはいけない。

ここで、「・・・に従う」というのは、「・・・に後続する」、「・・・に合わせる」という意味合いを持っている。

「上位レベルの方針」(抽象)は事前に決定され、具体的な手段である様々な具象サプライアを利用するとしても、変化することがないソフトウェアの方針であり、ソフトウェアの枠組みとして機能する。正しいオブジェクト指向設計においては、具象クラスに先行して、「上位レベルの方針」が決定される。また、抽象クラス/インターフェイスは、クライアントの要求により決定される。



3) 具象サプライア

抽象クラスやインターフェイスに対して、決定力をもつのはクライアントである。クライアントの要求に従って、抽象クラスやインターフェイスが決定される。

なぜ、「逆転」というかというと、従来の C 言語では、サプライアに応じてクライアントを変更しなければならないため、「上位レベルの方針」が「下位のモジュール」、「実装の詳細」に依存する。それと比較した場合、依存関係逆転の原則が提示するルールは、依存の方向性が逆転しているからである。また、C 言語では、サプライアを再利用するように設計するのに対し、オブジェクト指向設計では、クライアントを再利用するように設計するという違いがある。

以下は、違反例です。

```
class XMLReader {

    public String read() {
        // 内部省略
    }

}

class XMLWriter {

    public void write() {
        // 内部省略
    }

}

class Executer {

    public void doRead(XMLReader r) {
        String data = r.read();
    }

    public void doWrite(XMLWriter w) {
        w.write();
    }
}
```

具象クラスのサプライア XMLReader を別のサプライアへ変更したり、XMLReader の read()メソッドの引数を変更したりした場合、それに合わせて、クライアント（上位レベルの方針）を変更しなければならない。

以下は、依存関係逆転の原則に合致した例です。

```
class Executer {

    public void doRead(Readable r) {
        String data = r.read();
    }

    public void doWrite(Writable w) {
        w.write();
    }
}

interface Readable {
    String read();
}

interface Writable {
    void write();
}

class XMLReader implements Readable {
    public String read() {
        // 内部省略
    }
}
```

```
class XMLWriter implements Writable {
    public void write() {
        // 内部省略
    }
}
```

サプライアは、インターフェイスに従わなければならないため、サプライアがアプリケーションの上位レベルの方針に依存します。

26.17 オブジェクト指向設計と上位レベルの方針

ソフトウェア開発において、オブジェクト指向設計を適切に進めるためには、ソフトウェアの構造の内部に、「上位レベルの方針」を発見する必要がある。Robert C. Martin は、この「上位レベルの方針」を「内在する抽象(Underlying Abstraction)」(『アジャイルソフトウェア開発の奥義』, p.168)と言い換えています。ソフトウェアの構造全体の中で、この「内在する抽象」を発見することこそ、オブジェクト指向設計であるということができます。

PBL 用テキスト

目次

1 PBL 学習の意義 4

1.1 はじめに 4
1.2 IT アーキテクトの業務範囲 4
1.3 PBL 学習の目標 6

2 本テキストの進め方 8

2.1 PBL の前提 8
2.2 本講義で扱う業務範囲 8

3 案件・RFP 10

3.1 案件および RFP 演習の進め方 10
3.2 RFP とは 10
3.3 RFP と RFI 11
3.4 RFP を作成する目的 12
3.5 RFP の作成方法（発行前アクティビティ） 12
3.6 RFP の構造 15
3.7 課題１ 17
3.8 課題２ 21

4 ヒアリング演習 28

4.1 ヒアリング演習の進め方 28
4.2 『提案』を行う目的 28
4.3 大枠のシナリオを作成する 29
4.4 ヒアリングを行う目的 34
4.5 ヒアリングにおける質問事項洗い出し 35
4.6 課題３ 36
4.7 課題４ 37
4.8 課題５ 37
4.9 課題６ 38
4.10 課題７ 39

4.11 ヒアリング演習 39
4.12 提案の作成 40
4.13 提案の発表（受講者への対応） 44

5 USDM 演習 47

5.1 USDM 演習の進め方 47
5.2 USDM とは 47
5.3 USDM を用いた要求仕様書の作成 48
5.4 課題８ 49
5.5 課題９ 52
5.6 非機能要求の記述方法 55
5.7 課題１０ 63
5.8 USDM の発表（受講者への対応） 64

6 アーキテクチャ設計演習 65

6.1 アーキテクチャ設計演習の進め方 65
6.2 品質特性駆動形設計とは 65
6.3 アーキテクチャを設計する 66
6.4 課題１１ 67
6.5 課題１２ 69
6.6 課題１３ 70
6.7 課題１４ 72
6.8 アーキテクチャの発表（受講者への対応） 74

7 要件変更・外部要因変化への対応演習 77

7.1 要件変更・外部要因変化への対応演習の進め方 77
7.2 課題１５ 77
7.3 要件変更・外部要因変化への対応発表（受講者への対応） 78

8 詳細設計演習 79

8.1 詳細設計演習の進め方 79
8.2 オブジェクト指向設計の５原則 79
8.3 開放-閉鎖原則とは 79
8.4 開放－閉鎖原則に合致するための設計上の注意点 88
8.5 課題１６ 88
8.6 詳細設計発表（受講者への対応） 88

9 付録１ オンラインアンケートシステム設計開発仕様 90

9.1 用語の定義 90
9.2 業務の概要 90
9.3 必須機能要件 91
9.4 非機能要件 93

10 付録 2 RFP 94

10.1 はじめに 94
10.2 提案依頼の趣旨 94
10.3 業務要求 96
10.4 技術要求 98
10.5 運用要求 100

11 各フェーズの解答例と解説 101

11.1 課題１に対する解答事例と解説 101
11.2 課題２に対する解答事例と解説 103
11.3 課題３に対する解答事例と解説 105
11.4 課題４に対する回答事例と解説 106
11.5 課題５に対する解答事例と解説 106
11.6 課題６に対する解答事例と解説 107
11.7 課題７に対する解答事例と解説 108
11.8 課題８に対する解答事例と解説 109
11.9 課題９に対する解答事例と解説 111
11.10 課題１０に対する解答事例と解説 113
11.11 課題１１に対する解答事例と解説 115
11.12 課題１２に対する解答事例と解説 116
11.13 課題１３に対する解答事例と解説 118
11.14 課題１４に対する解答事例と解説 119
11.1 課題１５に対する解答事例と解説 122
11.2 課題１６に対する解答事例と解説 123

1 PBL 学習の意義

1.1 はじめに

昨今、ソフトウェア業界を取り巻く環境は大きく変化している。一部のユーザーが趣味で使用する程度だったコンピュータは、いまや業務で当たり前のように使用されており、街を見渡しても多くの機器がインターネットを通じて接続されている。それらを制御するソフトウェアは最早社会インフラと呼んでも過言ではないだろう。当然、ソフトウェアの使用頻度があがるという事は、ビジネスのチャンスも増え、多くの企業が参入し、凌ぎを削るような状況となっている。日々、新たな要件が生まれるこの環境において、全く同一のソフトウェア開発プロジェクトと言うものは存在せず、ソフトウェア開発に従事するエンジニアは、常に新しい要件に触れ、それらに日々挑戦して行くことが求められる。座学でソフトウェア工学を学ぶ意義は、新たに現れる課題を解決にする基礎的な知識をつける上で大変大きい一方で、それだけではなく、座学にて学んだ知識をどのように応用し、活用して行くのかと言う部分における知識・経験が重要である事は自明であろう。

このような背景から、より実践的な IT 人材育成のために、多数の高等教育機関において、実際のプロジェクトを課題として与え、受講者がその実現方法や問題解決を主体的に考えてプロジェクトを推進していく、Project Based Learning(以下、PBL)が導入され、効果を上げ始めている。本章では、その PBL を用いて、これまで学んできた「要件定義」と「アーキテクチャ設計」の知識を活かす実践的な方法について理解を深める事で、IT アーキテクトに求められるとされる、「ビジネスの要件に基づくビジネスとシステムの構造(アーキテクチャ)と最適な開発手法の策定」を可能とする能力を身につける事にある。

1.2 IT アーキテクトの業務範囲

上述のように、本講座における目的は、一般的に IT アーキテクトに求められるとされる、「ビジネスの要件に基づくビジネスとシステムの構造(アーキテクチャ)と最適な開発手法の策定」を可能とする能力を身につける事にある。そこで本項ではまず、IT アーキテクトの定義とその業務の範囲に関して簡単に触れていく。

「IT アーキテクト」の定義と言った時に、情報処理推進機構(以下、IPA)が提唱している「IT スキル標準 v3」における「IT アーキテクト」が参考になる。IT スキル標準 v3 では、IT アーキテクトは専門分野毎に以下の三種類に分類されるとされる。

- アプリケーションアーキテクチャ
- インテグレーションアーキテクチャ
- インフラストラクチャーアーキテクチャ

同じく IT スキル標準 v3 では、IT アーキテクトの業務の範囲として以下を定義している。

ビジネス及びIT上の課題を分析し、ソリューションを構成する情報システム化要件として再構成する。ハードウェア、ソフトウェア関連技術(アプリケーション関連技術、メソドロジ)を活用し、顧客のビジネス戦略を実現するために情報システム全体の品質(整合性、一貫性等)を保ったITアーキテクチャを設計する。設計したアーキテクチャが課題に対するソリューションを構成することを確認するとともに、後続の開発、導入が可能であることを確認する。また、ソリューションを構成するために情報システムが満たすべき基準を明らかにする。さらに実現性に対する技術リスクについて事前に影響を評価する。IT投資の局面においては、戦略的情報化企画(課題整理と分析(ビジネス及び IT)、ソリューション設計(構造とパターン))を主な活動領域として以下を実施する。

- 戦略的情報化企画 ・ソリューションの枠組み策定 ・ソリューションアーキテクチャの設計 当該職種は、以下の専門分野に区分される。

- アプリケーションアーキテクチャ
  ビジネス及びＩＴ上の課題を分析し、機能要件として再構成する。機能 属性、仕様を明らかにし、アプリケーションアーキテクチャ（アプリケー ションコンポネント構造、論理データ構造等）を設計する。設計したア ーキテクチャがビジネス及びＩＴ上の課題に対するソリューションを構 成することを確認するとともに、後続の開発、導入が可能であることを 確認する。

- インテグレーションアーキテクチャ
  全体最適の観点から異種あるいは複数の情報システム間の統合及 び連携要求を分析し、統合及び連携要件として再構成する。統合及び 連携仕様を明らかにし、インテグレーションアーキテクチャ(フレーム ワーク構造およびインタオペラビリティ)を設計する。設計したアーキ テクチャが統合及び連携要求を満たすことを確認するとともに、後続の開発、導入が可能であることを確認する。

- インフラストラクチャアーキテクチャ
  ビジネス及びＩＴ上の課題を分析し、システム基盤要件として再構成す る。システム属性、仕様を明らかにし、インフラストラクチャアーキテク チャ(システムマネジメント、セキュリティ、ネットワーク、プラットフォー ム等）を設計する。設計したアーキテクチャがビジネス及びＩＴ上の課 題に対するソリューションを構成することを確認するとともに、後続の 開発、導入が可能であることを確認する。
  (情報処理推進機構 『IT スキル標準 v3』より引用)[1]

[1] 情報処理推進機構『IT スキル標準 v3』https://www.ipa.go.jp/files/000010109.pdf

1.3 PBL 学習の目標

本講義の主題となるPBL(Project Based Learning)についてその概要と、学習の目標に関して触れていく。PBL の定義に関しては、先導的 IT スペシャリスト育成推進プログラムの提唱する、PBL(Project Based Learning)型授業実施におけるノウハウ集[2]の第1章が参考となる。以下引用。

昨今、特に情報系の分野における実践的な教育手法として、PBL に対する注目度が高まっている。PBL とは、Project Based Learning の略称であり、日本語では「プロジェクト型学習」や「問題解決型授業」などと訳されることが多い。PBL は、通常の授業や教育手法と以下のような点が異なっているとされ、この違いが今まさに、注目を集める要因となっている。

1. 課題の解決を目的とする（アウトプット・総合力志向）
2. チームの力によって課題を解決する
3. 受講者の自主性・自立性を重んじる

   => 従来の教育手法では育成が難しかった能力を効率的に育成することが可能

情報系分野における PBL は、上に示したような特徴を踏まえつつ、システム・ソフトウェア開発に関連する形で行われることが多い。例えば、チームで一つの開発テーマに取り組み、何らかのシステムやソフトウェアを完成させるような教育形態が一つの典型例である。また、システムやソフトウェアを開発するプロセスの一環として、顧客との交渉などを体験するケースもある。産業界での実務に近いシステム・ソフトウェア開発を疑似体験し、将来の活躍の基礎となる総合的な能力・スキルを習得できるという点で、情報系分野における PBL には高い期待が寄せられている。 PBL は欧米発祥の教育手法であるが、日本でも、医歯薬学の分野など、実習・演習が重視される分野においては以前から有効な教育手法として知られてきた。この PBL が、特にここ数年、情報系の分野の新しい教育手法として、大きな注目を集め始めた背景には、情報系分野における教育に対して、現在以上の“実践性”が必要だと考える産学双方の強い問題意識の台頭がある。IT（情報技術）が多くの産業の基盤となり、わが国の技術力や国 際競争力の根幹となっている現状において、IT に関する高い技術力や活用力を有する人材 が、これまで以上に強く求められるようになっている。こうした中、産学双方では、高度 な人材の育成に向けて、産業界に入る以前の段階、例えば高等教育の段階から、産業界で の実務に資するような実践的な教育を実施することが重要であるとの声が強まりを見せている。そして、このような流れの中で、近年、大学等の高等教育機関において、産学連携による実践的な教育が積極的に展開されるようになり、その中で、実践的な教育を実現す るための効果的な教育手法として PBL が注目を集めているのである。PBL の実施は、今や情報系の分野における実践的な教育には

[2] IT スペシャリスト育成推進プログラム『PBL(Project Based Learning)型授業実施におけるノウハウ集』http://grace-center.jp/wp-content/uploads/2012/05/pblknowhow20110726.pdf

不可欠とまで言われるようになっている。

（IT スペシャリスト育成推進プログラム『PBL(Project Based Learning)型授業実施におけるノウハウ集』p.1 より引用）

定義においても述べられているが、PBL における学習の目標は、一般的に座学などインプット型の教育で得られた知識を、どのような形でアウトプットにして行くかと言う「実践力」を身につける事にある。「実践力」と一口に言った時に、様々な解釈はあるが定義にもあげられている、問題解決能力、チームで物事に取り組む為のチームワーク、そして積極性などのソフトスキルは勿論の事、知識を応用して利用する事で、座学において学んだ内容のより深い理解が得られることも期待される。

2 本テキストの進め方

2.1 PBL の前提

・お客様（発注側）は、教師または第三者、IT ベンダ（受注・開発）は受講者とし、より実践的な体制とする。
・基本は、学生主体で進めるが、学生への支援（ファシリティ）は、状況に応じ、教師や実務経験者が講義や相談を行う。
・IT ベンダーより情報システム化の情報を求める RFI（Request For Information: 情報提供依頼）から実施することは期間、スキル等からも現実的ではないので本講座の対象外とし、RFP(Request For Proposal 提案依頼書)の作成は情報システム開発経験を有する経験がないと正確に記述することは難しいことから、RFP は予め（教員）で作成し、IT ベンダー（学生）に提示するところより、PBL を開始する。
・システム提案書は、提案書テンプレートを準備し作成させる。
・チーム作業とするが、事前にチーム体制や役割、スケジュール等は別紙を参照。
・受講対象の前提は、開発（実装）知識・経験がある、USDM 手法の知識があること。

2.2 本講義で扱う業務範囲

本講義は、ある条件に当てはまるユーザーに対し、街角アンケート、会場アンケート、座談会、などを実施し、クライアント企業の製品開発などに役立てる、「市場調査」業務の一部をオンラインアンケートにて実施することで、低価格かつスピーディーに行い、今後多様化していく市場調査のニーズを捉えることを目的としたシステムである『付録１：オンラインアンケートシステム　設計開発仕様』の RFP を提供されたところからはじまる。提供される RFP を元に、当該システムを開発する上で想定される要件を洗い出し、提案、要件定義、アーキテクチャ設計までを行うことを目指す。
提供される RFP の構造については、『Bud Porter-Roth RFP 入門 はじめての提案依頼書 日経 BP ソフトプレス 2008 年』の各章「１．事務要求」、「２．技術要求」、「３．管理要求」、「４．供給者の参加資格と参照先」、「５．供給者」、「６．価格」、「７．契約書とライセンス契約」、「８．付録」を参照した上で作成されている。本テキスト付録２において抜粋・要約しているので必要に応じて参照し理解を深めて欲しい。

本講義では、以下のプロセスに従いプロジェクトを進めて行くことを期待している。

1. ヒアリング内容の検討

提供された RFP を元に、なぜオンラインアンケートシステムの構築が必要となったのか、その原点を探り、より良い提案を提供するために必要事項を洗い出すヒアリング内容を検討する。

2. ヒアリング実施

作成したヒアリングシナリオを元に、実際に講師（「取得者」企業担当者）に対しヒアリングを実施する。ヒアリング時間は１チーム最大１５分とし、事前に質問項目を提出した上で順番に行う。

3. 提案の作成

ヒアリングで聞き出した情報と RFP にある内容を精査し、チームとしての『提案』を作成する。『提案』は基本的には取得者企業に提出、及び説明発表を行う事が求められる。今回は、PBL の中でより各チームの

4. 提案内容の発表

作成された提案を、「取得者」企業担当者に対し行い、フィードバックを貰う。提案発表の時間は発表２０分質疑１０分の２５分間。実際にはこの結果で「供給者」企業として採択されるかどうかが決まるが、今回は受けたフィードバックを元に提案の改善を行い次のステップに進む。

5. 品質駆動型アーキテクチャ設計を用いたアーキテクチャ設計

提案を元に、アーキテクチャ設計を行う。品質シナリオの作成、ビューの作成、そしてアーキテクチャ設計書を作成するところまでを一連の流れとする。座学で学んだ品質駆動形アーキテクチャ設計を応用し、明快なアーキテクチャ設計書を作成することが期待される。

3 案件・RFP

3.1 案件および RFP 演習の進め方

本章では、講義（PBL）で扱うソフトウェア開発プロジェクトである、オンラインアンケートシステムに関する RFP（Request For Proposal）を読み理解するための知識を学んでいく。付録２として掲載されている次章で学ぶ 『提案』 につなげる事が出来るレベルで、RFP を理解する事が出来る様になることが目標となる。

実際の現場において、RFP を受領した後に最初に行うべきは、RFP の内容を読み込み、供給者として求められている条件を満たすことが出来るか、契約上留意すべき点はないか、そして、取得者が何を目的にして、どのようなシステムを作りたいかを把握することである。これら、プロジェクト受託の可・不可を判断するのに加えて、プロジェクトの規模に対して予算感に違和感はないか、RFP の内容から測られる取得者のリテラシーレベルはどうか、取得者企業との関係性としてこのプロジェクトの重要度はどの程度のものか、など、実際に自社として RFP を受諾した上で 「提案」 を行うべきなのかどうかについても検討していく。

本章の前半部分では、RFP とは何か、また、何のために必要であるかについて RFP 発行者側の目線から学ぶことで、必須項目や考え方を学ぶ。 3.5 節においては、一部穴埋め形式の演習を通して RFP の中に記述されている内容を考える事で、本講義での主たる題材となる「オンラインアンケートシステム」において求められている内容を読み解く。

3.2 RFP とは

RFP(Request For Proposal)とは、「取得者」が「供給者」[3]に「提案」を求める正式な依頼を表す。この RFP に対する返答である「提案」が、後に供給者との間に結ばれる契約の礎となることが一般的である。RFP は多くの場合複数の「供給者」に提供され、提案を競わせるために使われることが多い。

RFP を利用することで、「取得者」はより多くの異なる「供給者」のソリューションを評価して、自分たちのニーズと予算に合うものを選ぶことが可能で、「供給者」は契約などの打ち合わせに入る前に「取得者」が求めていることをある程度把握して、自社の得意とする領域などを上手く絡めた形での「提案」を作り込むことが出来る。また、RFP が適切に作成され、提供されている場合、「取得者」と「供給者」との間で対話を深めるためのたたき台としても使用することができる。同一のルールと要求、スケジュール、そして共通の言語を用いて議論が出来るため、「取得者」、「供給者」共に、RFP 作成時点では曖昧だった

[3] 「取得者」 とは、RFP を作成し、「供給者」 に対して装置、ハードウェア、ソフトウェア、サービスなど、何某かの製品の製造を依頼し、購入する意図のある企業や組織を表す。「供給者」 は RFP に基づき、自社で展開可能なサービスを提案し、実際に製品を製造、「取得者」 に納入する事で収益を得る企業を表す。

要求をより明らかにすることが可能となる。

ただし、この RFP も必ずしも全てのケースで必要とされる訳ではない。Bud Porter-Roth 著、渡部洋子監訳の「RFP 入門」では、RFP を利用するに足るケースとして、以下の条件が満たされることを挙げている。

- ニーズに合うソリューションが複数ある
- 同じソリューションを提供できる供給者が複数いる
- 取得者は、供給者のソリューションの中から「最良のもの」を決定したいと望んでいる
- プロジェクトで使う製品を明確に指定することができない
- 供給者にさまざまなスキル、専門知識、技術的能力を要求するプロジェクトである
- 供給者が製品とサービスを組み合わせて請負契約を結ぶ必要がある
- 最低価格が契約を決める決定基準ではない
- 最終的な価格は供給者と交渉して決める

（日系 BP ソフトプレス『RFP 入門』p.2 より引用）

3.3 RFP と RFI

RFP において取得者は

1. すべての技術的な問題を明確に理解していること
2. それら技術をどう実装し管理していくかを明確化していること
3. 供給者が受け入れられるための条件

の3つを示す必要がある。これらを示すために取得者は、様々な情報を収集し、調査した上で、自身のニーズと照らし合わせながら RFP を記述していくこととなる。一方、流通している情報には限りがあり、それらで全ての必要事項を賄えない時、取得者は RFI(Request For Information)を発行する。RFI は取得者が、供給者から何が得られるのかを判断する方法の1つで、望む要求が合理的であるかどうか、そして適切な技術を利用することができるかどうかを判断する方法の1つでもある。一般的に、RFI を発行すべき時と言うのは以下のような条件を満たす時と言われている。

1. プロジェクトで使用する技術が業界や企業にとって新しいものであるとき
2. 様々なソリューションの可能性を調査したいとき

3.4　RFP を作成する目的

RFP は、供給者に「提案」を求める正式な依頼である。先述の通り、ここでの「提案」は最終的に契約の一部となることが多く、RFP の作成はプロジェクトの成否を担う要素となる可能性を強く秘めたプロセスであることは明白である。RFP を作成する目的は、大きく言えば、プロジェクト初期のプロジェクト定義フェーズと、実装フェーズの間にある重大なギャップを埋めることである。構造化された RFP を利用することで、作成済みのプロジェクト要求を供給者が使用でき、理解でき、応えられる形式に変えることが出来る。

また、プロジェクトの実装方法や、ステップ、そして成功の基準を説明することも可能となる。RFP を使う利点に関して、「RFP 入門」では以下の5点を挙げている。

1. RFP を使うと、RFP チームはプロジェクトの課題や問題点を、通常の場合よりもずっと詳細に調べる必要がある。

2. RFP を使うと、供給者は RFP の要求に対応するだけでなく、それを超えるような競争力のあるソリューションを作らざるを得ず、したがって所定の価格で付加価値をつけてくれる。

3. RFP を使うと、ある供給者を他の供給者より優遇したりせず、全員を同じルールと要求のもとで公平に競争させることができる。

4. 供給者が同じルールと要求のもとで作業するので、提案されたソリューション間の違いを理解するのがより簡単になる。

5. 類似しているが異なったソリューションが提案されるので、評価が容易になる。
（日系 BP ソフトプレス『RFP 入門』p.12-13 より引用）

3.5　RFP の作成方法（発行前アクティビティ）

■ RFP プロジェクトの立ち上げ

RFP の作成をはじめる起点となるのは常に技術やサービスを買いたいと言う取得者の要求となる。これは、企業内の様々な部門で上る可能性があり、その理由も様々である。例えば、仕事の効率化を図りたい社員が、新たな技術を用いたサービスが業務改善につながることに気づき要求をあげる場合もあれば、自社のサービスと連携することでシナジーを生む可能性があるサービスを購入する提案を行う場合もあるし、技術者が社内セキュリティ向上のためにアンチウイルスサービスの導入を検討する場合もあるだろう。

これらの購買要求を大きく分けると、部門や業務プロセスの効率を挙げたり、改善したりするために製品を購入しようとする企業内の「ユーザー」が控えている要求(ある特定の部やチームに対して効果がある購買)と、IT 部門やリスク部門などが提案し全社に対して何某かの効用が得られる購買要求がある。RFP の作成に際しては、その起点となる購買要求がどのようなものであるのかを鑑みた上で、以下の3部門の中から均等にスキルバランスを取り、RFP プロジェクトチームを組織することが重要である。

1. 業務部門(ユーザー部門)
   ユーザーは、その仕事が請求査定処理か顧客サービスかまたは人事部のような支援サービスかに関わりなく、業務として仕事をします。ユーザーは自分がいまやっていることは何かを知っており、同様に、したいとは思っていても現状できないことは何かということも知っています。つまり、システムの操作要求や機能要求を作成することができる人々です。しかしユーザーは通常、たとえばネットワークはどうやってサーバーにつながるのか、どうやって受け入れテストの準備をして実行するのか、どうやってデータはファイアウィールお通過するか、といったシステムの基礎技術に強いわけではありません。そういう技術的な要求は、IT 部門に任せます。

2. IT 部門
   IT 部員は通常、購入する製品が既存の技術インフラと技術的に適合するか、そのときの要求は何か、製品サポートには何が必要か、製品導入時にユーザーをサポートするためにはなにが必要か、などに関する知識を持ちます。しかし IT 部員は通常、ある部門が業務でどういう働きをしているか、またはその業務要求をどう記述すべきかはわかりません。

3. 調達部門(購買部門)
   調達部員は、提案の種類(ハードウェアかソフトウェアかサービスか)に応じてどんな契約を結べばよいかを理解しています。また、供給者の身元を確認し、D&B を入手して財務の安定度を確かめ、供給者との契約をレビューして交渉する支援をしてくれます。RFP プロセスの初期に調達部員を巻き込むことは極めて重要です。そうすることで、調達部員がプロジェクトを良く知るようになり、契約に関する専門知識を貸してくれるようになります。調達部員は通常、基礎技術もプロジェクト組織の要求もあまりよく知りませんので、それらの事項の決定は RFP チームに任せることになります。

(日系 BP ソフトプレス『RFP 入門』p.13-14 より引用)

■ RFP 作成

RFP プロジェクトチームが決定し、プロジェクトの立ち上げが行われた後は、実際に RFP の作成を行っていく。本ステップでは、実際の RFP 作成作業に加えて、次の 10 個の項目を考慮する。

1. RFP プロジェクトの要因と組織
2. RFP プロジェクトスケジュール
3. 技術と供給者に関する教育
4. 予算の見積もりと作成
5. 投資回収率(ROI)分析
6. RFP 作成
7. 提案の評価
8. 決定と契約
9. RFP 発行後のアクティビティ
10. 新システムのプロジェクトスタッフと組織

（日系 BP ソフトプレス『RFP 入門』 p.14 より引用）

これらは、RFP プロジェクトチームでミーティングを執り行い議論し作成されるべき内容で、企業文化や、購買する内容に応じて、これらの計画やミーティングを管理、推進するプロジェクトリーダーを設定すべきである。

■ RFP の評価基準策定

供給者は RFP に対して「提案」を返してくる。その「提案」を評価する基準も事前に策定しておくことで、実際に複数の供給者からの「提案」を受けた時に、客観的な視点でそれらを比較し、どの供給者にプロジェクトを依頼するか策定することが出来る。多くの供給者は、RFP に対して「提案」を返す際に、実はすでに内諾を受けている供給者がいるのではないか、などと言う疑いを持っていることが多い。そこで、なぜ最終的に選ばれた供給者が選ばれたのか、それを明確に示してあげることは、今後の供給者との関係を保つ上で大変重要である。

これを行う上では、評価基準をいずれかのタイミングで供給者に伝える必要がある。提案前に伝える場合と、一部だけ提案前に伝える場合、提案後に伝える場合でそれぞれ気をつけなければならないことがある。提案前に伝える場合は、当然供給者はその評価基準に沿った項目に力をいれて提案を行うこととなる。しかし、それはつまり、取得者が重要と思う所(評価基準として定めたところ)以外から、新たな可能性を見出すことを困難にしてしまうことに繋がる。次に、一部だけ提案前に伝える場合であるが、こちらも基本的には提案前に伝える場合と同様。どうしても評価基準に即する部分に注目をしてしまうことは避けられない。ただ、一部分の評価基準が明確になっていないことで、その他の部分にも力をおいた提案を受けることが出来る可能性は高まる。一方で、提案後に公開される評価基準が、あまりにも提案前に公開された評価基準と整合が取れていない場合、同じく供給者からの信頼を失うことに繋がる。最後に、提案後に共有する場合だが、こちらはあまりにも1つの部分の評価基準が高いなど、せっかく努力をしてまんべんなく「提案」を行った努力を踏みにじるような評価基準が提示されると、同様に信頼の低下につながる。

このように、取得者の RFP プロジェクトチームは、評価基準の内容、提示方法も含めて、供給者との関係性を正しく保ちながら、RFP に対して良い「提案」を受けられる形を考慮することが求められる。それらを実現するために、多くの RFP プロジェクトチームでは、評価チームを置くことが推奨されている。船頭が多くなった場合に、最終的に異なるステークを持って評価に望んでしまうことがあり、評価基準策定が難しくなるためである。よって、最終的なステークホルダーとなるメンバーを数名 RFP プロジェクトチームより選抜し、そのメンバーで評価基準を定めていくことが一般的であると言われている。

一般的なソフトウェアプロジェクトにおいては、以下のようなことを評価基準とすることが多い。

1. 技術要求
2. 管理要求
3. 価格
4. 照会先
5. 参加資格/類似プロジェクトの経験
6. Web サイト/口頭のプレゼンテーション
7. 製品テストまたはデモンストレーション
8. RFP への回答全体
9. 供給者チームとの共同作業の可能性

（日系 BP ソフトプレス『RFP 入門』 p.15-16 より引用）

■ 作成された RFP のレビュー

RFP を供給者に発送する前に、RFP プロジェクトチームの主要メンバー以外の人々によって RFP をレビューすることが推奨されている。多くの場合、RFP プロジェクトチームのメンバーは、自分たちの求める機能要求など目に見える所に注釈しがちであるが、それを違った視点からレビューすることで、RFP でカバーしきれていないところや、逆にあまりに詳細に渡って記述されてしまっている部分などを洗い出し、適切な RFP が発送されることを保証するために役に立つ。

3.6 RFP の構造

RFP の対象は、ソフトウェアプログラム、コンピュータシステム、事務サービス、技術サービス、法律上のサービス、機械装置、医薬品、などさまざまな種類の製品に渡る。種類が異なる場合は基本的に、異なる RFP が求められるが、以下の項目は基本的に共通して用いられることが多い。

1. 事務要求の章

要求のエグゼクティブ・サマリー（経営陣向けの要約）と同様に、課題の概要または要約を記述す

る。そして、RFP に関する事務的な情報を記述する。

2. 技術要求の章
技術的な要求を記述し、供給者が問題を理解してしっかりした提案を書けるだけの情報を提供する。

3. 管理要求の章
プロジェクトを実行し管理するための条件を記述する。

4. 供給者の参加資格と照会先の章
供給者の参加資格について説明し、自社の資格と照会先を記載するよう供給者に依頼する。

5. 供給者の章
RFP では求められていなくても、供給者が関連すると思う情報をいれるように勧める。

6. 価格の章
供給者が提案の中でどうやって価格情報を伝えればいいか説明する。

7. 契約書とライセンス契約の章
購入契約、機密保持契約、その他法律関係の文書をいれる。

8. 付録
ネットワーク図や技術要求の検討結果、プロジェクト計画の概要など、量の多い関連情報をいれる。

（日系 BP ソフトプレス『RFP 入門』p.17-18 より引用）

3.5.1 概要、事務要求の章

「事務要求」の章では、供給者に対して自社の企業概要を説明し、RFP を通じて解決したい課題を論じる。ここで、課題について供給者が正しく理解出来るように、業務上の問題、課題に潜む技術的な問題、などを明確にして述べる必要がある。加えて、必要な事務的な要求、そして、「提案」を受け入れるに至るまでに、供給者が従わなければならない事項を記述する。具体的には以下の項目が挙げられている。

1. 提案の提出期限と提出先

2. 提案前 RFP 説明会を行う条件と日時

3. 調達に関連する日程

4. 提案作成要領

5. 提案の評価方法

6. RFP 窓口担当者名とメールアドレス

7. その他、供給者が回答するために必要な情報

3.7 課題１

付録2にある、「オンラインアンケートシステムのRFP」を概要と事業要求の章を熟読し、以下の空欄を埋める事。全ての項目が存在するとは限らないため、存在しないと思われる項目に関しては「該当なし」と記述すること。

1. 提案の提出期限と提出先

2. 提案前 RFP 説明会を行う条件と日時

3. 調達に関連する日程

4. 提案作成要領

5. 提案の評価方法

6. RFP 窓口担当者名とメールアドレス

7. その他、供給者が回答するために必要な情報

3.5.2技術要求の章

「技術要求」の章では、RFP の作成を行う理由となった問題や課題の要約が述べられている必要がある。問題の記述に合わせて供給者が「提案」の中で対応しなければならない要求を記載していく。

1. プロジェクトの目的と目標
2. 重要成功要因
3. 現行システムの機能仕様
4. 計画中のシステムの機能仕様
5. 性能要求
6. ハードウェア要求
7. ソフトウェア要求
8. 通信要求

3.8 課題２

付録2にある、「オンラインアンケートシステムの RFP」の技術要求の章を熟読し、以下の空欄を埋める事。全ての項目が存在するとは限らないため、存在しないと思われる項目に関しては「該当なし」と記述すること。

1. プロジェクトの目的と目標

2. 重要成功要因

3. 現行システムの機能仕様

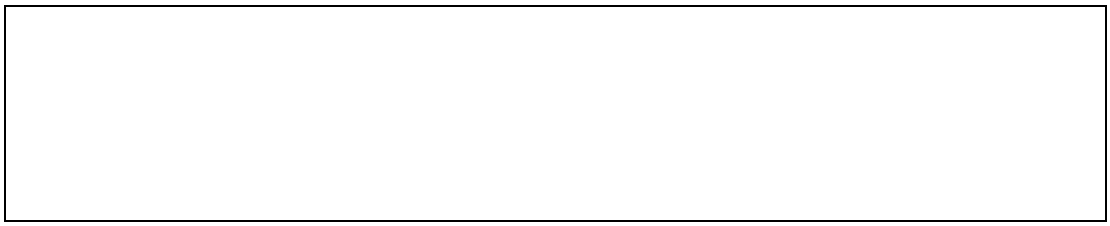
4. 計画中のシステムの機能仕様

5. 性能要求

6. ハードウェア要求

7. ソフトウェア要求

8. 通信要求



3.5.3 管理要求の章

実装、導入、教育・訓練、保守、およびプロジェクトのその他の側面を網羅したプロジェクト計画を作成するために供給者が必要とする情報を提供する。この章が不完全な場合、供給者がプロジェクト要求を漏らしてしまうことが起こりうる程重要な章となっている。「管理要求」の章では一般的に以下の項目が満たされているべき。

1. プロジェクト実行要求
2. 要因配置要求
3. 作業場所確保責任
4. 出荷と導入のスケジュールと計画
5. システム受け入れテスト要求
6. システム保守要求
7. システム教育要求
8. 文書化要求

3.5.4 供給者の参加資格と照会先の章

「供給者の参加資格と照会先」の章では、供給者に対して自社の情報や財務状態、作成中の提案の照会先として顧客の名前を出すように要求する。この章は供給者が正しくプロジェクトを履行することが出来るかどうかを確かめる上で重要な項目を抑えている。

1. 供給者の会社の簡単な沿革
2. 供給者の導入・保守提案と導入・保守能力
3. 供給者と各メーカーとの関係、およびその関係の継続期間
4. 供給者が契約を実行するために必要な技術力、技術スタッフ、財源を所有している証拠

5. 現在導入済みのシステム一覧

6. 類似の構成と業務をもち、照会に応えてくれる顧客名（連絡先や電話番号もつける）

3.5.5 供給者の章

「供給者」の章は、求められていないが供給者が必要だと思う情報を提供するための場所を、RFP 中に確保するためのもの。RFP や提案に潜む問題について議論をしたり、RFP 範囲外の追加製品機能を説明したり、RFP で抜けていると思われる要求に関してコメントするなど、取得者が想像もしないような何かから、より良い問題解決を図る提案を受け入れることが出来るように用意する章となる。

3.5.6 価格の章

「価格」の章では、供給者が価格提案を作成する際に使用する詳細な書式を明確化する。複数の供給者からの価格提案を同じ目線で比較するために大変重要な章となっている。書式を明確化する際に特に意識しなければならないのは、「1度限りの費用」と「繰り返しかかる費用」の対比である。同じ項目でも、契約形態によって「1度限りの費用」になる場合と「繰り返しかかる費用」となる場合があり、特に「繰り返しかかる費用」に関しては。期待する稼働年数などに応じて比較検討が異なってくるため、これらを明確に分別出来る形での書式が求められる。『RFP 入門』では、サンプルとして、提案を受けるシステムを次のような要素に分割することが検討出来ると述べている。

1. ハードウェア

2. システムソフトウェア

3. アプリケーションソフトウェア開発

4. 導入

5. 保守

6. 教育・訓練

7. 文書作成

8. プロジェクト管理

9. 独自のハードウェアまたはソフトウェアの統合

10. ライセンス料金

3.5.7 契約書とライセンス契約の章

「契約書とライセンス契約」の章では、契約や協定にどう対応すればよいかについて、供給者の基本的なガイダンスとなるものを提供する。「価格」の章の一部となる事もある。契約書を供給者に渡すと、供給者は RFP の要求とともに契約書の調査を始めることが出来る。これを RFP 内に記述することで、そもそも契約に進むことが出来ない供給者をこの時点で弾くことができるため、取得者、供給者ともに無駄な時間を使わずに済むこととなる。
契約書の中には以下のような項目が含まれる。

1. 購買同意書

2. 保守契約

3. 保証期間

4. ソフトウェア・ライセンス契約

5. 契約保証

6. 支払い保証

7. 機密保持契約

3.5.8 付録

RFP 本体にいれるには長すぎる詳細情報などをこの章に記述する。付録に添付されることが多い資料の例としては

1. ワークフロー図と検討資料

2. 統計情報が入ったスプレッドシート

3. 通信ネットワーク図と計画

4. 現行機器の一覧

5. 企業内で使っている標準

6. 日付の入った仮プロジェクト計画

（日系 BP ソフトプレス『RFP 入門』 p.18-25 より引用）

4 ヒアリング演習

4.1 ヒアリング演習の進め方

本章の目標は、前章で扱った RFP を理解した上で抜け漏れがなく、魅力的な『提案』を作成すること、そして、それをプロジェクトの案件を受注するに足る形で、取得者企業に対して適切に説明を行うことである。良い『提案』を行うためには、幾つかのステップが存在する。

1. RFP を抜け漏れなく理解すること

2. 提案の大枠となるシナリオを作成すること

3. 差別化を行うに足る情報をヒアリング（質疑応答）を通して得ること

4. 魅力的な提案資料を作ること

5. 相手の理解度に応じて適切な提案発表を行うこと

このうち、ステップ1については既に前章にて完了しているものとする。本章では、ステップ2からステップ5までを PBL 形式で、チームで課題を解きながら学んでいく。一方、具体的な方法論に入る前に再度、『提案』とは何で、どのようなことを目的として行われているかについて再確認する。

4.2 『提案』を行う目的

本章で扱う所の「提案（2時提案）」を行う目的はたった1つで、自社の技術力そして提案力を持って取得者企業のニーズを満たし、開発プロジェクトの「受注」を勝ち取ることである。

通常、多くの現場において「提案」には1次提案と2次提案が存在する。

1次提案は、初回訪問から数回程度の打ち合わせの間に行う提案活動のことを表し、主に以下の3点を目的とする。

- 提案活動を行う意志があることをユーザーに表明する。つまり RFP をもらうこと

- ユーザーへ製品情報などを提供すること

- ユーザーがどの程度本気で案件を検討しているか、推し測ること

（日系 SYSTEMS 永井昭弘著 RFP&提案書 完全マニュアル p97 より引用）

1次提案はよって、基本的には汎用的な提案書（宛先を変えれば他社でも使用可能なもの）や、商品紹介のパンフレット・カタログなどを利用する事が多い。

翻って、本節でも対象として扱う2次提案は、RFP など正式な「提案」の依頼が来た際に作成するもので、取得者企業の熱量も異なれば、期待値も異なる。よって、汎用的なものではなく、取得者企業にカスタマイズされた形の提案が求められる。特に、RFP を発行する規模の企業の案件は一件で数千万円から数億円になる事もあるため、RFP を受け取った供給者企業は皆、ライバル企業に負けないような提案を心がけて作成してくる。よって、RFP の内容をカバーするだけではなく、自社独自の技術や工夫などを織り込んだ「提案」を作成、また提案発表も高い質で行うことが求められる。

	目的	時期	提案のレベル
1次提案	RFP をもらう	初回訪問から RFP 発行まで	・ 汎用的な内容 ・ カタログやパンフレット
2次提案	受注する	RFP 発行から ベンダー選定まで	・ ユーザーごとにカスタマイズした内容 ・ オリジナルな提案資料

（日系 SYSTEMS 永井昭弘著 RFP&提案書 完全マニュアル p97 より引用）

4.3 大枠のシナリオを作成する

前節で述べたように、『提案』を行う最大の目的は、プロジェクトの受注を勝ち取ることである。よって、取得者企業に選ばれるような『提案』を作成し、その発表を行う必要がある。そのために、作成された『提案』は、以下の4つのポイントを抑える必要があると言われている。

1. RFP に盛り込まれた要求をライバルより多く満たしている。特に優先順位の高い要求を満たすことは必須である。

2. 単に顧客から求められた要求だけを満たしているのではなく、それ以上の付加価値が提示されている。

3. 魅力的な価格が提示されている

4.　上記がすべてユーザーに理解できるようわかりやすく記述されている

（日系 SYSTEMS 永井昭弘著 RFP&提案書 完全マニュアル p99 より引用）

これらを満たすためには、『提案』の骨子となる大枠のシナリオを正しく制定する必要がある。基本的に、『提案』には定型的なフォーマットは存在せず、自身が最も説明しやすい形で行うのが一般的ではあるが、ここでは先程より引用している、『提案』作成時のバイブルとも言える「RFP&提案書 完全マニュアル」にて紹介されている項目を引用して、提案書の骨格づくりの手引とする。

『提案』の骨子を作る上で、必ず意識するのは、MECE と言う問題解決のフレームワークで提示されている考え方である。

MECE (Mutually Exclusive and Collectively Exhaustive)は、直訳すると 「相互に排他的」かつ「完全な全体集合」となり、これはつまり 「漏れなく、だぶりなく」と言う形で表現される。例えば、人間という集合を例にとると、「年齢」による分類は、ある人が同時に 20 歳と 21 歳になることは不可能（重複なし）であり、全ての人が X 歳という集団に属する（漏れなし）ため、MECE となる。一方、「職業」による分類は、兼業を行う人間（重複）もいれば無職の人間（漏れ、ただし職業として「無職」という項目を設けるとすれば、漏れはなくなる）もいるため MECE とならない。

『提案』に置き換えて考えると、RFP で求められている項目に対して、すべて回答している（漏れなく）必要がある。もし RFP のニーズが満たせない部分があるのであれば、それは「満たせない」と記述すべき、もしくはその代替案などを述べるなど、「抜け漏れ」があったと判断されないような配慮が求められる。「だぶり（重複）」の例としてよくあがるのは、ボリュームを少しでも増やそうと、コピー＆ペーストを繰り返して、同じ記述が繰り返される『提案』である。例えば、「松・竹・梅」と複数の選択肢がある場合に、取得者企業が知りたいのはそれぞれの違いであって、共通部分が繰り返されることを望んでいない。それどころか、共通部分に埋もれて重要な違いの部分が見逃されてしまうことさえ起こる。

これらは、『提案』 のコンテンツのことを述べているようであり、実は取得者企業（クライアント企業）へと向き合う際の基本的なことを述べているにすぎない。例えば、提案発表を聞いたときに、「あっ、この部分がカバーされていない」と感じれば、その発表を聞いていた人は、この会社は駄目な会社かもしれないと言う意識を持つようになってしまうし、「この資料同じことを繰り返してばかりで実がないな」と思われてしまえば、その他の部分にどれだけ素晴らしいことが書かれていても、読み飛ばされてしまうかもしれない。

『提案』 の中身に、どれだけ付加価値を加えて、差別化を行っていても、それを発表する企業自体の信頼が低くなってしまえば、その付加価値も失われてしまう。当然それはプロジェクトの受注と言う目標に対してネガティブに働く。更に言えば、取得者企業（クライアント企業）との関係は、今回の提案発表一度きりのものではないため、信頼を失うことのリスクは、受注が出来ないことよりも遥かに大きい。実務の世界においてはクライアント企業との関係は一度きりの発表の間、プロジェクトが受注出来るか出来ないかが決まる間だけで終わるものではなく、今後もずっと続いていくもので、その中でしっかりとした信頼を勝ち得て行くことがいかに重要であるか、と言うことは肝に銘じる必要がある。

脇道にそれてしまったが、「RFP&提案書 完全マニュアル」では MECE に『提案』を行うための、提案書の構成要素と言うものを提示している。(以下、要約して掲載する)

枕詞 ＋ 8つのテーマ ＋ 特記事項 ＋ 別紙

1. 枕詞

表紙、挨拶文、目次の３つを、ここでは枕詞と呼ぶ。この枕詞の部分は提案内容に一切関係ない。しかし、ビジネス・マナーとしては必要なパーツとなる。

2. ８つのテーマ

（ア）提案の趣旨

提案書で最も重要なページと言っても過言ではないのが「提案の趣旨」である。構成要素としては、「提案の特徴」と「RFP に対する回答」の二つが挙げられる。

「提案の特徴」では、自社の提案のどこがユニークで優れているのかをアピールする。ただ、ユニークな点を述べるだけではなく、RFP と対応した形で記述される事が望ましい。例えば RFP で述べられている内容に対して、敢えて逆の提案をすることでユニークさを出すことがある。飛び道具的ではあるが、他社との差別化と言う意味では大変有効な手法である。一方で、突然何の説明もなく逆の提案がなされていた場合、取得者企業はこの企業は RFP の内容をしっかりと読んでいないと言う判断をしてしまう。RFP の内容との対応を考え、必要であれば説明を補足する形でアピールを行う事が求められる。
「RFP に対する回答」では、基本的には RFP で求められている内容のどの部分が満たされていて、どの部分が満たされていないのかを記述することとなる。この部分を

簡潔に並べ、その理由（特に満たされていない部分がある場合）を述べる事で、提案発表を聞く側としても、全体像を把握した上で話を聞くことが出来る。

「RFP に対する回答」を簡潔に述べる方法として、別紙に対応表を作り、その要旨を「提案の趣旨」で述べることがある。

(イ) 提案対象領域

「提案の趣旨」が自社の提案の特徴をアピールし、RFP の要求を前向きに対処していることを示す、いわば “攻め”のページであるとすれば、この提案対象領域を定義し確認することは“守り”である。RFP が出されている以上、提案対象に関する認識が大きく異ることは考えにくいが、提案書できちんと定義を行うことで食い違いの発生を抑える。

(ウ) システム構成

システム概念図、ネットワーク構成図、ハードウェア構成図、ソフトウェア構成図など、システム構成を表現する図を掲載する。提案時点では、詳細まで煮詰まっていない部分もあるかと思うが、現時点での構想を展開する事で、取得者企業側にある程度完成形のイメージを持たせる手助けとなる。

(エ) プロジェクトの内容

提案しているシステム案件でどのようなことをするのかを説明するパートである。情報システム部門であれば既知の作業内容であろうが、経営者やエンドユーザーも観覧する可能性の高い提案書にはきちんと書いておくべきである。

(オ) 運用体制

提案するシステムの運用フェーズまでを検討し、記述する。システムはユーザーが正しく運用してこそ、投資の効果が期待できる。ユーザーが適切に運用できるようなシステムを開発し、なおかつその運用管理についても支援サービスを提供するような提案を作成する必要がある。

(カ) プロジェクト体制

プロジェクトをどのような体制で推進していくのかを記述する。提案段階で担当者のアサインまで計画出来ているベンダーと、契約が取れてから対応すれば良いと考えているベンダーでは、明らかに差がある。計画力、管理力もさることながら、その案件にかける想いがユーザーに透けて見える部分である。自社側のみではなく、ユーザー側の体制も含めて提案が出来ると、なお良い。

(キ) スケジュール

スケジュールの提案は、基本的にはマスター・スケジュールの提示をもって行う。マスタースケジュールに記入すべき内容は下記３点となる。

① システム開発に関するスケジュール

1. 要件定義
2. 基本設計
3. プログラム作成
4. テスト
5. 移行
6. 並行稼動
7. カットオーバー

② 導入スケジュール

上記の開発スケジュール以外に実際にシステムを稼働させるために必要な導入計画を記入する。例えば、ハードウェアの搬入スケジュール、エンドユーザーや管理者に対する研修のスケジュール、システム導入に伴う各種工事のスケジュールなどを指す。

③ 進捗会議などのスケジュール

定例の進捗会議の開催予定や、マイルストーンとなるような重要な判定会議などの大枠のスケジュールを記入する。

（ク）見積もり

見積もりに関しては「システム一式」で総額だけ提示するのではなく、詳細な項目ごとの見積もりを提示すべきである。

3. 特記事項

提案活動および提案書に関する事務連絡事項や各種条件を記述するパートである。この特記事項のパートも提案内容そのものにはあまり大きく影響せず、また重要度もさほど高くないが、ここをきちんと記述しておくことで、顧客とのちょっとした思い違いによる事務トラブルなどを削減することが出来る。

4. 別紙

前述の通り、提案書はなるべく少ない枚数で簡潔（だぶりなく）に要点がわかりやすくかかれているべきである。しかし、意思決定するのは経営者であっても実際に提案書を読み込み、中身を評価するのは情報システム部門のメンバーであることが多い。彼らに対してアピールすべき部分、伝えておくべきことは、提案の本筋からはあえて切り離して別紙として載せることが望ましい。別紙の候補となる資料は以下のようなものが挙げられる。

- RFP との対照表
- ハードウェアの詳細なスペックや技術的なアピール・ポイント
- ソフトウェアの詳細なスペックおよびパッケージ利用などの場合の機能説明
- ネットワークの詳細な構成
- フローチャートなどの情報システム部門のメンバーだけが理解出来る資料
- 会社概要、実績紹介

次節より、実際に提案を作成するために必要な作業をチームで行っていくこととなる。本節で学んだ『提案』に掲載すべき事柄を意識した上で作業にあたって欲しい。

4.4 ヒアリングを行う目的

前節で述べた、取得者企業に選ばれる『提案』を作成する4つのポイントのうち2つめの、「単に顧客から求められた要求だけを満たしているのではなく、それ以上の付加価値が提示されている」でも挙

げられているように、『提案』において、プロジェクトを受注出来る可能性をあげるためには、他社との差別化を図り、自社の得意とする技術領域や、自社ならではの創意工夫を盛り込んだ内容を作り込むことが求められる。

これらを盛り込むためには、単純に RFP に記載された内容だけで判断をすることが難しいことも多く、より深く取得者のニーズを理解し、その問題解決を測るためにも、取得者との間でRFPをたたき台にした議論を行うことが望ましいとされる。取得者次第ではあるが、通常 RFP 発行後に、『RFP 説明会』と言われる、RFPプロジェクトの担当者から送付されたRFPに関する説明がなされる場や、RFPプロジェクト責任者へのヒアリングの機会などが設けられる事が多く、この機会を活かしRFPには記載されていない、取得者の真に解決したい問題を発見する事が、RFP 説明会の参加や、RFP 担当者（責任者）へのヒアリングを行う最大の目的となる。

4.5 ヒアリングにおける質問事項洗い出し

ヒアリングにおいては、まずは自社の技術の棚卸しを行い、今回のRFPの内容と関連がありそうな部分と、それらを「提案」として盛り込むべきなのか否かを判断するために、どのような項目の確認が必要であるかを洗い出す。本講義では、ヒアリングの時間を各チーム15分間儲けている。他チームよりも少しでも良い『提案』を行うためにしっかりと準備を行って欲しい。ヒアリングの内容を詰めて行く方法に規定の方法は存在しないが、自由に話し合いを行うだけではまとまらない可能性があるため、以下のステップに従い行って欲しい。

1. RFP の内容への質問事項の洗い出し

   前章で行った RFP の内容の精査の結果を利用して、チーム内でお互い説明し合える部分は共有する。その上で、理解出来なかった部分、より深く知りたい部分を洗い出す。前述の通りいかに独自性のある提案を行ったとしても、基本的な RFP の内容を理解出来ていなければ、その『提案』の価値は極めて低くなってしまう。曖昧な理解になってしまっている部分があれば、そこをしっかりと詰められるように準備をする事が重要である。

2. 自チームの技術的特徴の棚卸し

   自己紹介も兼ねて、自チームの構成メンバーがどのようなバックグラウンドで、どの分野に強みを持っているのかを棚卸しする。ステップ1と合わせて、彼我の状況を正しく知ることで、独自性があり、付加価値を提供出来る『提案』を行える可能性が高くなる。

3. 『独自性の高い提案』のブレインストーミング・絞り込み

自チームの技術的な強みを棚卸し出来たところで、それを活かした提案をブレインストーミングする。この時点では、実現可能性や、クライアントのニーズに関しては考慮せずに、一旦思いつく限りのアイディアを出す。技術ベースのアイディアでも良いし、この時点での着想はどんなところでも構わない。
ある程度（例えば全体で 20 個以上など目標を決めると良い）揃った時点で、今度は実現可能性やクライアントのニーズを考えて絞込を行う。今回はヒアリング時間が 15 分と限られているので、その中で現実的に聞き込みが出来そうな数に絞込を行うべきである。

4. 『独自性の高い提案』の可能性を探る質問を用意

独自性が高い提案も相手のニーズに沿ったものでなければ意味がない。そこで、前のステップで絞り込まれた『独自性の高い提案』が、相手のニーズに沿うものであるかについてを問う質問を用意する。特に技術発信で出てきた提案については相手がその技術に対して理解できない場合も考慮した上で、それを使うことの良い点、悪い点を明確に述べた上で、相手が判断出来る形で質問をする事が望ましい。

以下、課題形式にてステップバイステップで進めていく。それぞれの議論の中でのアウトプットを空欄の中に埋めて行って欲しい。この議論や、ヒアリング結果をそのまま『提案』の作成に活かして行くこととなる。

4.6 課題３

前章で洗い出した RFP の構成要素を突き合わせて、チームでの理解を共有する。その上で、曖昧になってしまっている部分、理解できなかった部分、より深く知りたい部分を洗い出し、以下の枠に質問形式で記述する。

RFP の内容への質問事項の洗い出し

4.7 課題４

自己紹介も兼ねて、自チームの技術的特徴（強みなど）を棚卸しする。現在仕事で取り組んでいること、学生時代に研究で行っていたこと、趣味で取り組んでいること、など些細なことから技術的な特異点が見つかる可能性がある。

自チームの技術的特徴の棚卸し

4.8 課題５

自チームの技術的な強みを棚卸し出来たところで、それを活かした提案をブレインストーミングする。この時点では、実現可能性や、クライアントのニーズに関しては考慮せずに、一旦思いつく限りのアイディアを出す。技術ベースのアイディアでも良いし、この時点での着想はどんなところでも構わない。

『独自性の高い提案』のブレインストーミング

4.9 課題6

実現可能性やクライアントのニーズを考えて絞込を行う。今回はヒアリング時間が 15 分と限られているので、その中で現実的に聞き込みが出来そうな数に絞込を行うべきである。

『独自性の高い提案』の絞り込み

4.10 課題 7

絞り込まれた『独自性の高い提案』が、相手のニーズに沿うものであるかについてを問う質問を用意する。特に技術発信で出てきた提案については相手がその技術に対して理解できない場合も考慮した上で、それを使うことの良い点、悪い点を明確に述べた上で、相手が判断出来る形で質問を用意する。

『独自性の高い提案』の可能性を探る質問を用意

4.11 ヒアリング演習

前章で検討したヒアリング内容を元に各チーム15分間のヒアリングを行う。本講座では、講師が RFP 担当者役を務める。それぞれのチームは講師の指示に従い、順番にヒアリングを開始すること。ヒアリング後は、基本的にヒアリングで質問した内容と、手元にある RFP を元に「提案」の作成を行うこととなる。

追加での質問を行うことは実際の現場でも難しいことが多い為、基本的には講座内でも行えないようにする。そのため、抜け、漏れがないようにヒアリングを行うよう心がけること。また、RFP 内に明確に記述されている内容をただ繰り返し聞くようなことは、取得者企業からの信頼低下につながるので避けること。このヒアリングの段階からすでに取得者企業の選別が始まっていることを理解した上で望むこと。

ヒアリング中は適度にメモを取るなど、取得者企業の担当者が話した内容を網羅的に記憶することが重要である。せっかく、良い質問が出来たとしても、それへの回答が提案に活かされていなければ、質問をしなかった企業よりもより悪い評価がついてしまうことすらあり得る。人間関係全般にも言えることだ

が、相手の話をしっかりと聞いた上で、些細なことでもそれをしっかりと記憶し、何某かのリアクションを行うことが、相手からの信頼を得ることに繋がる。

講師は以下の評価項目に沿ってチームのヒアリングを評価しフィードバックを行う。よって、受講生はそれぞれの評価項目を意識して発表を行うこと

1. RFP の内容を深く理解した上で質問がなされているか
2. 質問の中で明確に RFP の中で記述されていることを聞いていないか
3. RFP に記述がない内容を聞き出すような質問が出来ているか
4. コミュニケーションのキャッチボールが正しく行えているか
5. 自信を持って質問が出来ているか

4.12 提案の作成

前述の提案書の構成要素(枕詞 ＋ 8つのテーマ ＋ 特記事項 ＋ 別紙)に沿って、RFP から読み取れたこと、ヒアリングにおいて取得者企業より引き出した内容を元に『提案』を作成していく。

実際の現場では、提案発表は、プレゼンテーションを通して説明をする方法や、会議の中で説明をする方法が存在し、取得者企業から特に指定がない場合は、自分達にあったスタイルでの発表を行うこととなる。会議室の準備などがあるため、どのように発表するか、発表の際に必要な機材は何か、などは事前に取得者企業に伝えることが出来ると良い。

本講座では、各チームの成果を共有し、質疑応答には他チームも参加出来る形を取るため、プレゼンテーションスタイルでの提案発表を期待する。各チーム、20分の説明発表時間に加えて、10分間の質疑応答の合計30分の中での発表を行う。

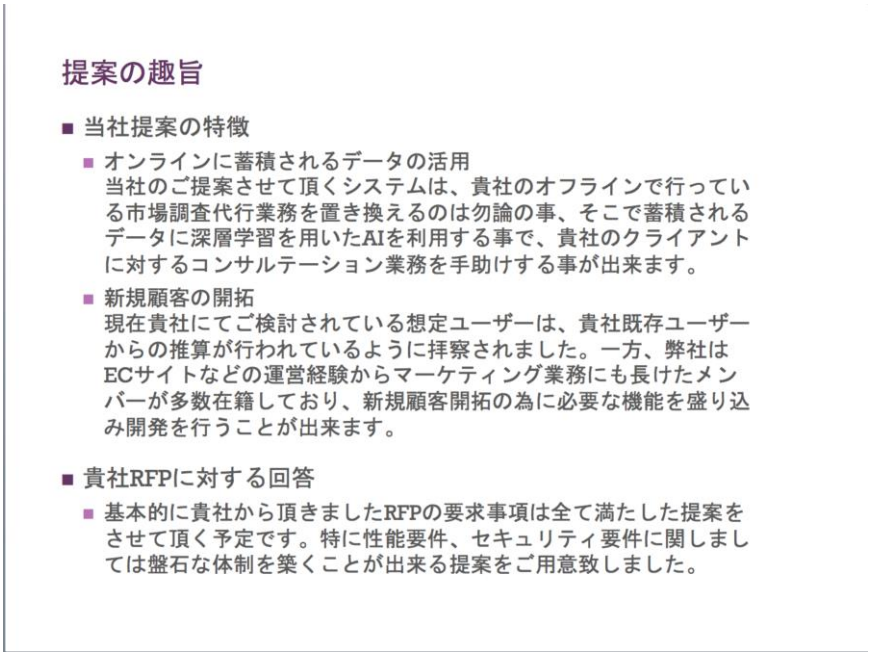
枕詞、特記事項、別紙については、実際の『提案』においては必須のものとなるが、講義としては必ずしも重要な部分ではないので、今回は8つのテーマのうち以下を作成する。(勿論、余力があればそれ以外の部分を作成しても構わない。)

これまでの説明を踏まえた上で、プレゼンテーション用の資料(ppt 形式)として作成を行う事。以下にサンプルとして提案資料の例を掲載する。

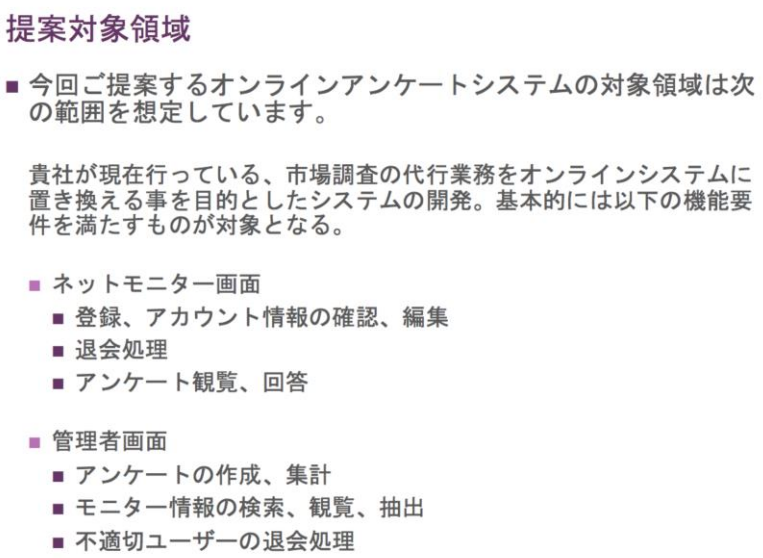
（ア）提案の趣旨

① 当社提案の特徴

② 貴社 RFP に対する回答



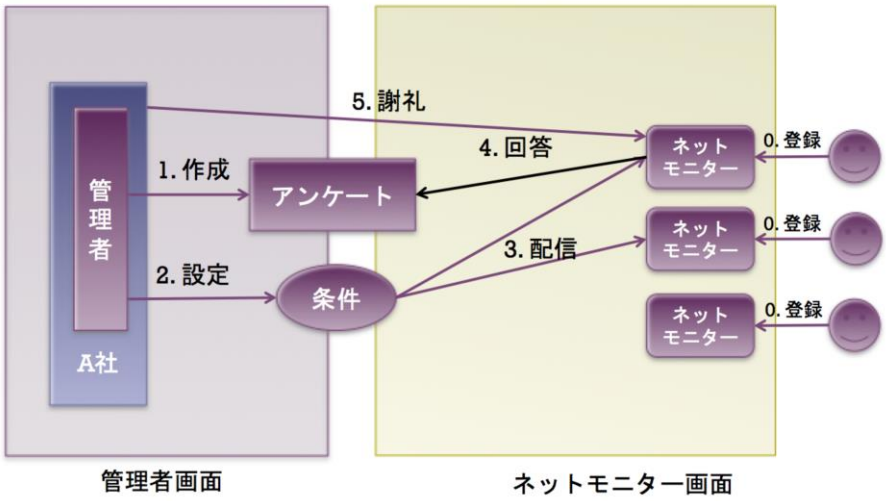
（イ）提案対象領域



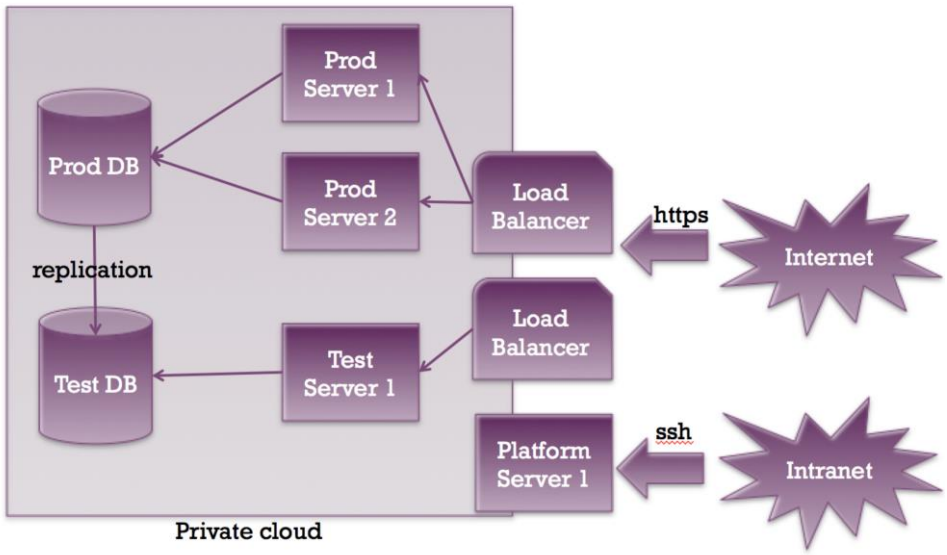
（ウ）システム構成

システム概念図、ネットワーク構成図、ハードウェア構成図、ソフトウェア構成図、etc.

システム概念図



ネットワーク構成図



(エ) プロジェクトの内容

プロジェクトの内容

		貴社	当社
要件定義	システムの対象となる業務の要件、領域を定義します	○	○
アーキテクチャ設計	要件定義の内容に基づき、システムの全体像を設計します	△	○
システム設計	システムの詳細設計を行って実装を行う技術的方法を決定します	×	○
システム開発	システム設計にもとづいて、プログラムの作成を行います	×	○
システム・テスト	プログラムが設計通りに稼働するかテストします	△	○
受け入れ・テスト	プログラムが要求通りに稼働するか、使用に足るかテストします	○	×
教育研修	システムの使用方法や業務の流れについての研修を実施します	○	○
本番稼働	オンラインアンケートシステム稼働開始となります	○	△

(オ) 運用体制

運用体制

- 運用にかかる費用削減の為の試み
  - 貴社からのご要望：
    システム運用にかかる人的コストを出来る限り削減したい。
  - ご提案１：
    - 本番サーバーは２台の冗長構成を取り一台は常にホットスタンバイ

      コールドスタンバイで、有事にそれらを起動するためのチームの配置や、定期テストなどのコストを削減するため。
  - ご提案２：
    - 弊社運用サポートサービスのご利用

      ロードバランサーにピンエコー方式で異常検知の仕組みを導入、
      コールドスタンバイにする代わりに、弊社の運用サポートチームが
      異常時には24/365日対応します。

4.13 提案の発表（受講者への対応）

前章で検討した『提案』を各チーム30分間（発表20分間＋質疑応答10分程度を想定）のプレゼンテーション形式で発表を行う。本講座では、講師が RFP 担当者役を務め、それぞれのチームは講師の指示に従い、順番に提案発表を開始すること。なお、質疑応答では、他チームへの質問や、フィードバックを積極的に行い、お互いの成長の機会と出来れば良い。

また、講師や、他の受講生から受けた質問やフィードバックを元に、チームで KPT（Keep Problem Try）の振り返りを行って提案発表を修了とする。

KPT とは、プロジェクトの振り返りをするときに最もよく使われる基本的な手法。今回のプロジェクトを通してのアクティビティを Keep（続けた方が良いこと）、Problem（問題だったこと）、Try（次はやりたいこと）に分類して振り返る手法。

PBL

メモ欄

PBL

Keep(続けたいこと)

Problem(問題)

Try(今後変えていきたいこと)

5 USDM 演習

5.1 USDM 演習の進め方

前章で作成した、『提案』をベースに、要件定義を行う。本演習では、まず、USDM（Universal Specification Describing Manner）の概略および表記法を学習し、USDM による「要求」表現の方法を確認する。その後に、演習事例に USDM を適用し、要求仕様の記述を行うという手順で進める。

USDM は、現在、一定度の普及がみられるユースケース図、ユースケース記述の補完手段としても利用可能であるとの指摘もあり、また、JUAS（一般社団法人日本情報システムユーザー協会）で採用されている点からも一定の有効性が認められているものと考えられる。

5.2 USDM とは

USDM 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と言う形を取る。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ため、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と期待されている。

USDM を採用するメリットとして、以下のことが実現可能となることが挙げられる

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。

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。

5.3 USDM を用いた要求仕様書の作成

要求には必ず理由が存在する。
「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」と言う要求があったとする。

この時点では様々な理由が考えられるが、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する。
すると、この要求が作られた理由は以下のようになる。

印刷中に文字がかすれたくないから

USDM 表記法の最初のステップでは、要求と理由をセットとし、次のように表記する。

要求	PRT07 (要求分類識別ための番号)	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

5.4 課題8

前章で作成した提案を元に、取得者企業の挙げた機能要求を理由とセットで書き出しなさい。全てを書き出すのは時間制約の中難しいため、優先度が高いものから順に5つを書き出すこと（先んじて終わった場合は5つ以上探しても良い）

要求		
	理由	

要求		
	理由	

PBL

要求		
	理由	

要求		
	理由	

要求		
	理由	

次に、要求の仕様化を行う。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)。先程のインクの例では以下のようになる。

＜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
□□□	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

更に、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	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	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

5.5 課題9

課題8で作成した要求と理由のセットを更に仕様化する。

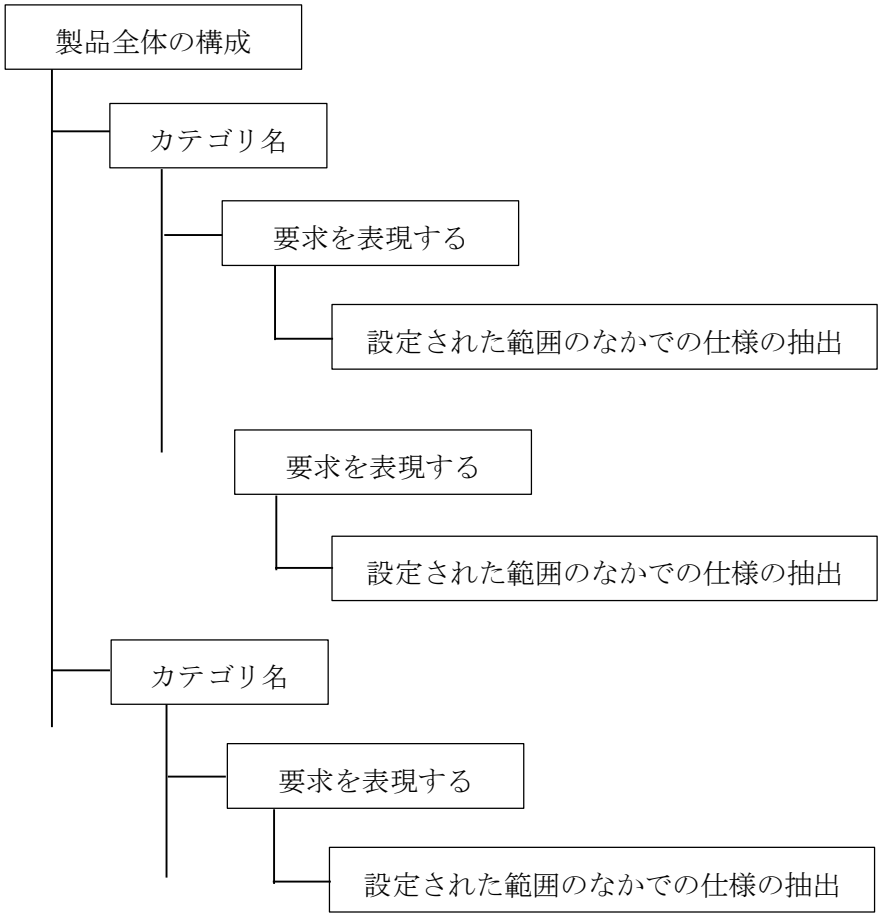
要求		
	理由	
	説明	
□□□		
□□□		

要求		
	理由	
	説明	
□		
□		

PBL

要求		
	理由	
	説明	
□		
□		

最後に、要求のカテゴリ化を行う。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。



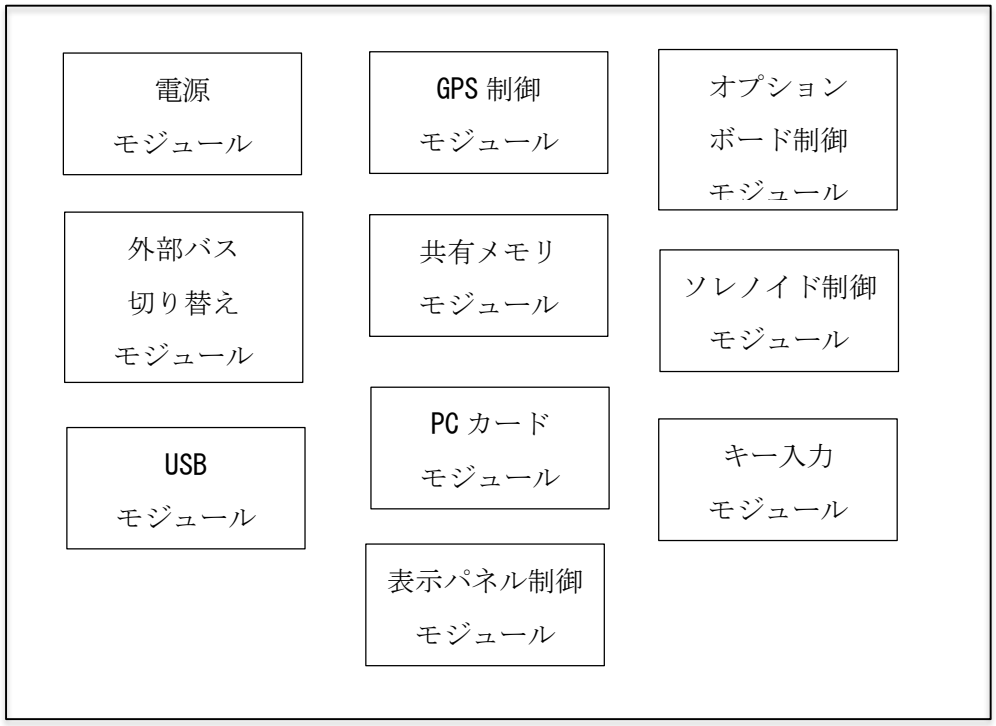
1. 機能で分割する例

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。

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。



要求の種類が多い際は上記のような方法を用いてカテゴリ化することで全体の把握が容易になる。他にも、要求の階層化などを行うことで適切な粒度に要求を分割する事で、より仕様に落とし込みやすくしたり、開発の際のタスク分割が容易になることもある。本講義では、時間の関係上要求の数を絞っていることもあり、カテゴリ化、階層化に関しての作業は省略する。

5.6 非機能要求の記述方法

機能要求については、取得者企業の要望を単純に明瞭化して記述する事になるが、非機能要求は上述の通り総じて曖昧で、記述が難しい事が多い。一方で、本講座で実践しようとしているアーキテクチャ設計手法である、品質特性駆動型設計においては非機能要求が重要な要素を占める事になる。そこで、以下に非機能要求の記述方法を、座学講義の教科書より抜粋する。

非機能要求において達成スベき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。

1. 品質

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。

2. サービスレベルに関する要件

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。

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。

(ア) サービス提供時間
(イ) 性能
    ① レスポンス・タイム
    ② バッチ処理の実行時間
    ③ デリバリータイム

3. サービス提供時間

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。

この定義の例として、次のような表現がある。

(ア) オンライン稼働時間 :8:00～20:00
(イ) 稼働曜日: 月曜日～日曜日(土日、祝祭日も稼働)
(ウ) 稼働日数: 363日(年に2回の保守停止がある)

4. 保守・レスポンス・タイム

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。

この定義の例として、次のような表現がある。

（ア）ログイン・メニューの遷移： 2秒
（イ）個別照会： 2秒
（ウ）一覧表示（最大20件）： 5秒
（エ）登録処理： 3秒
（オ）複雑条件検索： 10秒
（カ）回線の前提： ブロードバンド。実効速度は1Mbps 以上。
（キ）PC スペックの前提： Core i3、メモリーは4GB、Windows10 Proffesional。
（ク）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）。

5. 性能・バッチ処理の実行時間

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。

（ア）バッチ稼働開始時間： 24:00
（イ）完了限界時間： 28:00
（ウ）他のシステムとの依存要件： なし。

6. 性能・デリバリータイム

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。

この定義の例として、次のような表現がある。

（ア）〇〇表のデリバリータイム： データ入力翌々日15時までに、〇〇担当者の手元に届くこと。
（イ）契約者向け〇〇はがき： 処理の翌日に発送できる状態にすること。

7. キャパシティに関する要件

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。

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。

（ア）キャパシティ要件の予測の誤り
（イ）リソース設計の誤り
（ウ）運用開始後の監視の不備

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

（ア）データ量
（イ）ユーザー数
（ウ）トランザクション数
（エ）アウトプット数
（オ）インプット数

8. データ量

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。

この定義の例として、次のような表現がある。

（ア）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

9. ユーザー数

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。

この定義の例として、次のような表現がある。

（ア）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）
（イ）社内ユーザー:常時 3 万ユーザー
（ウ）外部（特定企業）接続:常時 500 ユーザー

10. トランザクション数

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。

この定義の例として、次のような表現がある。

（ア）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
（イ）アプリケーショントランザクション数（画面遷移数）:　300TPS
（ウ）HTTP リクエスト数:　1,000RPS

11. アウトプット数

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

この定義の例として、次のような表現がある。

（ア）◯◯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
（イ）◯◯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

12. インプット数

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。

この定義の例として、次のような表現がある。

(ア) ○○データのインプット数: 5,000 件／月

13. 安全性に関する要件

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。

システムの安全性には、次の3つの要素がある。

(ア) 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。
(イ) 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。
(ウ) 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。

14. セキュリティ

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。

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。

(ア) パスワードの桁数: 7 桁以上
(イ) パスワードの有効期限: 3 ヶ月間
(ウ) 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

15. 停止許容時間

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。

定義例として、以下のような表現がある。

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

16. 有事対策

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。

有事対策の定義例として、以下のような表現がある。

（ア）バックアップセンターで、n 時間以内に業務を継続する。
（イ）バックアップデータを、週次で遠隔地保管する。

17. コストに関する要件

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。

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。

（ア）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、・・・。
（イ）運用コスト： 年間 30,000 千円

（ウ）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。

18. 保守に関する要求

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。

保守についての要求例として、以下のような表現がある。

（ア）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。
（イ）モジュールの複雑度は、原則 20 以下とすること。
（ウ）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。
（エ）処理と管理は、明確に分離すること。
（オ）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。
（カ）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。
（キ）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。

5.7 課題１０

上記の非機能要求の記述方法を参考に、オンラインアンケートシステムにおいて述べられている非機能要求を仕様まで記述する。

要求		
	理由	
	説明	
□□□		
□□□		

要求		
	理由	
	説明	
□		
□		

要求		
	理由	
	説明	
□		
□		

5.8 USDM の発表（受講者への対応）

以上で、優先順位が高い順に3つずつの機能要求と、非機能要求を USDM 形式で記述が出来た。最後に、テストケース仕様とのつなぎこみを行う。本講義中では、課題としては行わないが、実際の業務の中ではテストケースを抜け漏れなく作成し、人為的なミスを限りなく少なくして行くことが求められるため、大変重要なプロセスである。

＜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

6 アーキテクチャ設計演習

6.1 アーキテクチャ設計演習の進め方

前章で作成した、USDM を元に、品質駆動型設計でアーキテクチャ設計を行う。基本的な流れは、座学の講座で行った演習と同様である。
今回のオンラインアンケートシステムにおいては、ネットモニター画面と管理者画面の二つが並行して作成される形となる。アーキテクチャ設計においてそれら二つを同時に扱う事は難易度をかなり高めてしまう事に繋がる。本質的には、それら二つの関連性を表すシステム全体に対するアーキテクチャ設計を行い、その中でそれぞれを分割して行く事が望ましいが、今回は管理者画面に注釈したアーキテクチャ設計を行う方針で進め、余力があれば、ネットモニター画面のアーキテクチャ設計も行うという形で進めるのが良いのではないかと考える。一方、勿論、双方を同時に扱うアーキテクチャ設計にチャレンジする事も歓迎する。

以下に復習も兼ねて、品質駆動形設計におけるアーキテクチャ設計のプロセスを要約する。

6.2 品質特性駆動形設計とは

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。

6.3 アーキテクチャを設計する

1 品質特性駆動型設計の開始

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。

前章で作成した USDM はアーキテクチャドライバを特定するのに十分な形で要件が定まっているため、それを元に以下のステップを進めることとなる。

2 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。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)。
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。

3 モジュールを詳細化する

3.1 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。

6.4 課題１１

前章の課題の成果物として作成した USDM は、優先度が高いものを基本的に残しているため、全てがアーキテクチャドライバとなり得る重要な要件となっている。その中で品質要件となる日機能要求を抜き出し、品質特性具象シナリオとして記述する。

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	

3.2　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。

- ✧　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
- ✧　実現手法が他の品質特性に対して持つ副作用

6.5 課題１２

アーキテクチャドライバを実現する為に必要な実現手法の組み合わせを検討する。また、その実現手法が他に与える可能性がある影響を検討し、その上で適切なモジュールの大局的な分割構造を記述する。

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	

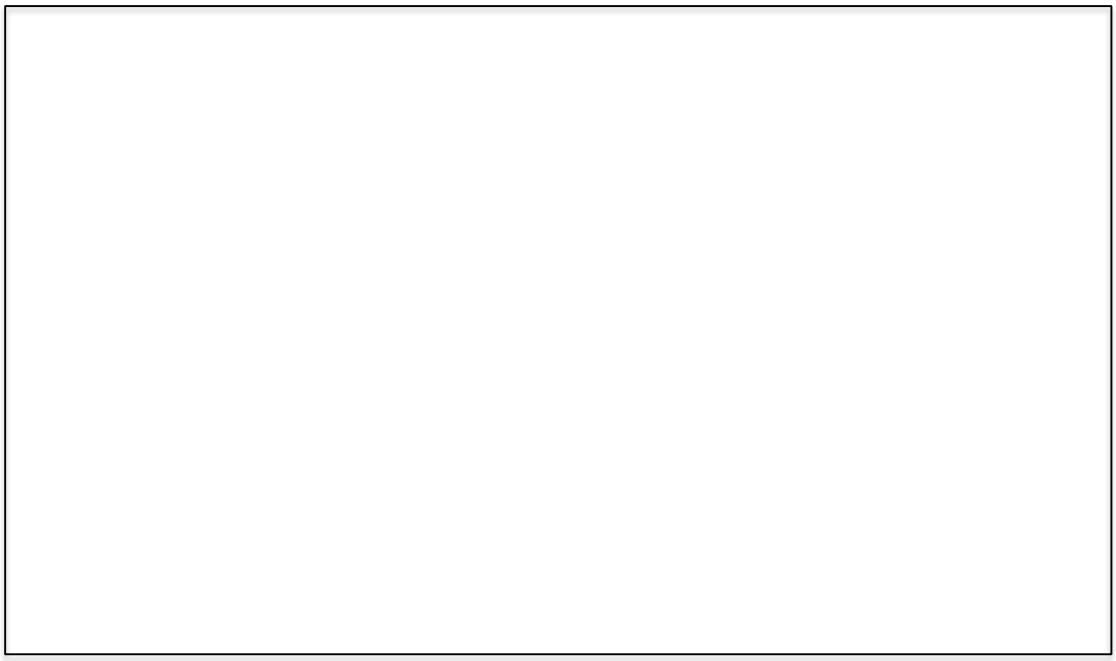
3.3　モジュールに機能性を割り当てる

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[4]）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。

6.6 課題１３

先程記述したモジュールの大局的な分割構造に対し、機能要件を適用することで、機能性の分配を行う。



3.4 複数のビューでアーキテクチャを表現する

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。

3.5 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る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。

6.7 課題１４

ここまでに作成した分割ビューを文書化して、アーキテクチャ設計の成果物を作成する。以下の表を埋めて、アーキテクチャ設計書を完成させる。（実際には複数ビューに関して作成をしてそれらを取りまとめる必要があるが、今回は分割ビューのみに関するものを扱う）

文書化項目	内容
プライマリプレゼンテーション	

PBL

要素カタログ	
コンテキスト図	
可変性ガイド	
アーキテクチャの背景	
その他の情報	

6.8 アーキテクチャの発表（受講者への対応）

前節までに作成した『要件定義』、『アーキテクチャ設計』の成果物を各チーム15分間（発表10分間＋質疑応答5分程度を想定）で発表を行う。なぜそのように考えて、どうしてそのような成果物が出来たのかという思考のプロセスを明らかにし、共有し合う事で、様々な考え方が身につく。他のチームがどのようなプロセスで取り組みを行ったのかについて、お互いコメントや、質問を積極的に行い、お互いの成長の機会と出来る事が望ましい。
それぞれのチームは講師の指示に従い、順番に提案発表を開始すること。

また、講師や、他の受講生から受けた質問やフィードバックを元に、チームで KPT (Keep Problem Try)の振り返りを行って本講義を終了とする。

PBL

メモ欄

PBL

Keep(続けたいこと)

Problem(問題)

Try(今後変えていきたいこと)

7 要件変更・外部要因変化への対応演習

7.1 要件変更・外部要因変化への対応演習の進め方

ここまで、品質駆動型アーキテクチャ設計の手法を用い、モジュール分割ビューの作成と、その文書化を行ってきた。通常の開発プロセスにおいては、このアーキテクチャ設計の成果物を元に、詳細設計、開発、とプロジェクトを進めて行く事になる。一方、現実の世界においては、要件変更や外部要因変化など、現在作っていたアーキテクチャを覆すような出来事が日々起こり得る。

例えば、「競合企業が新たなプロダクトを作ってきた為、それに対抗するための機能を作成して欲しい」という依頼や、「新たな技術が発表された事で、実現出来る事の幅が広がりそれを取り入れたい」という要望など、多くの要件変更は突発的に起こる。また、「全容が見えてきたら、ここはもっとこうしたい」などという、設計がある程度見えてきた事による要件変更も多々起こり得る。

アーキテクトとしては、これらの変更をある程度予測した上で、例えば変更が多く起こり得るかもしれない領域においては、「変更容易性」をアーキテクチャドライバに持ってきて、実現手法の中にも「変更容易性」を高めるようなものを持ってくる事で対応する。

一方、不足の事態で起こった要件変更については、その規模に応じてアーキテクチャ設計のレベルから変更する必要がある事もあるし、状態によっては詳細設計のレベルの変更で抑える事が出来ることもある。いずれにしても、要件変更はいずれかの設計に変更を与え、それがそれ以降の全てのプロセスに影響を与える。

要件変更はその規模の大小によらず、発生が遅れれば遅れる程その影響が大きくなる傾向にある。アーキテクチャ設計を行う目的の1つは、取得者企業の担当者が全容を理解する事で早期に後に起こりうる要件変更を洗い出す事にもある、と言われている。

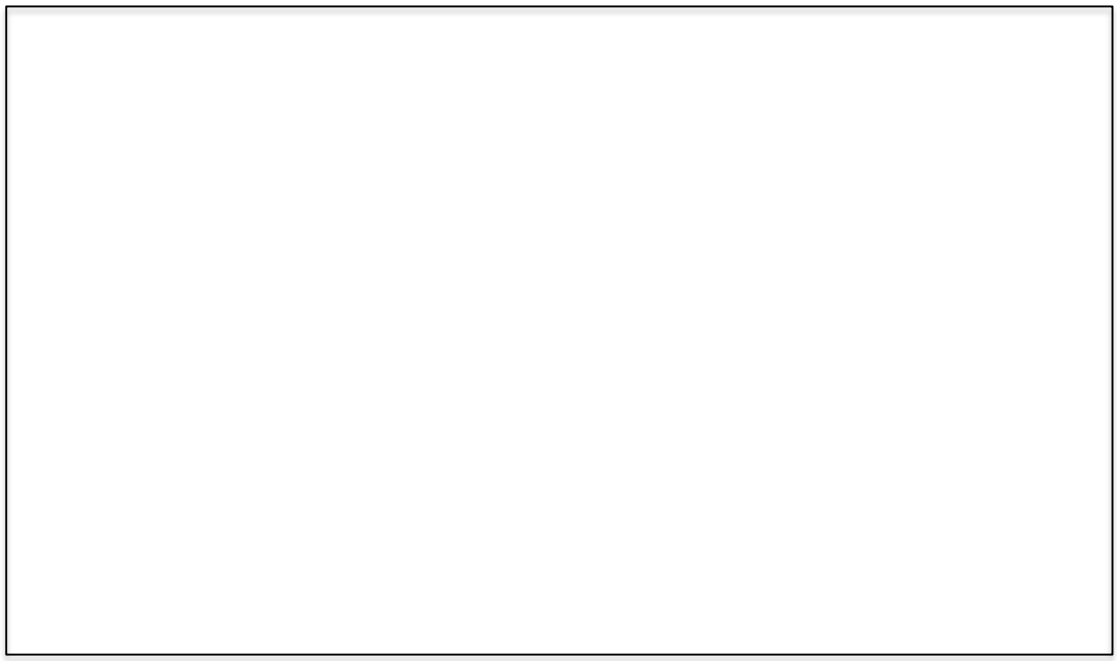
7.2 課題１５

課題14までに品質駆動形アーキテクチャ設計において作成した、モジュール分割ビューを、以下で説明される変更にあわせて再設計する。

『競合他社が同様のオンラインアンケートシステムを作成している事が判明した。競合との差別化のため、オンラインアンケートのみではなく、座談会や、会場アンケートと言ったオフラインの市場調査案件も一括で管理出来る仕組みに変更したい』

具体的には以下の機能が変更要求としてあげられている。（全て管理画面の変更）

1. アンケート以外の案件を作成出来るようにしたい。
2. 管理者権限ユーザーはリクルーターを追加出来るようにしたい。
3. 一般権限ユーザーは調査参加者をリクルーター毎に管理出来るようにしたい。



7.3 要件変更・外部要因変化への対応発表（受講者への対応）

課題15で再作成したモジュール分割ビューについて、元々作成したものとの差分を明確にしながら、どのような経緯でその変更が行われたのかについて議論を行う。各チーム 5 分間で発表し、質疑は 10 分間で行う。

8 詳細設計演習

8.1 詳細設計演習の進め方

詳細設計の目的は、機能性の品質を満たしたうえで、保守性の品質の点でも一定の品質を満たすソースコードを生み出せるように、UML 等を利用することによってモジュール間に適切な関係を設定することである。

本章では、オブジェクト指向設計にのっとった詳細設計の理論を踏まえたうえで、演習テーマに詳細設計を加えるという手順で学習を進める。演習テーマについては、そのすべてに関して詳細設計を行なうことはボリューム過多となるため、一定の部分をピックアップし、その箇所について UML のクラス図で詳細設計を行なうものとする。

8.2 オブジェクト指向設計の５原則

「オブジェクト指向設計の原則」とは、米国の Objectmentor 社 の Robert C. Martin が、さまざまなソフトウェア技術研究者によるオブジェクト指向研究の成果を総括し、以下の5つの原則にまとめたもののことです。

単一債務の原則(SRP:Single Responsibility Principle)
開放-閉鎖原則(OCP:Open-Closed Principle)
リスコフ置換の原則(LSP:Liskov Substitution Principle)
依存関係逆転の原則(DIP:Dependency Inversion Principle)
インターフェイス分離の原則(ISP:Interface Segregation Principle)

このなかでも、彼が「オブジェクト指向設計のゴール」と呼んでいる最も重要な原則が、「開放-閉鎖原則」です。

以下、とくに開放-閉鎖原則について確認しておく。

8.3 開放-閉鎖原則とは

「開放-閉鎖原則」は、もともとは、オブジェクト指向研究の第一人者である Bertrand Meyer(バートランド メイヤー)の研究成果[5]に由来しますが、この原則を解釈し、他の 4 原則と並べてオブジェクト指向設計の 5 原則としてまとめたのは、Robert C. Martin です。彼は、以下のように「開放-閉鎖原則」を定義しています。

5 Meyer, Bertrand, 1988, "Object-Oriented Software Construction", Prentice Hall（二木 厚吉 監訳, 酒匂 寛・酒匂 順子 訳, 2005, 『オブジェクト指向入門』, アスキー出版局）

> ソフトウェアの構成要素（クラス、モジュール、関数など）は拡張に対して開いて（オープン：Open）いて、修正に対しては閉じて（クローズド：Closed）いなければならない。
> （『アジャイルソフトウェア開発の奥義』, p.128）[6]

これは、一見矛盾するような次のふたつの条件を同時に満たすことを意味します。

1）ソフトウェアの仕様や要件定義に変更が生じた場合、ソフトウェアの振る舞いを変更して対応することができる。

2）ソフトウェアの既存のソースコードの変更や再コンパイルを行う必要はない。

通常、ソフトウェアの処理を変更するためには、ソースコードの変更および再コンパイルが必要です。したがって、上記の1)、2)を同時に満たすことは、一見すると矛盾であることになります。

しかし、この原則は、次のような仕方で実現することが可能です。

> ソフトウェアの仕様や要件定義に生じた変更に対して、クライアントを始め既存のソースコードを変更することなく、新しいサプライアを追加することによってソフトウェアの振る舞いを変化させ対応する。

●開放-閉鎖原則に違反するソースコード

```
class OracleDao {

    private String dbName; // データベース名

    public String getDbName() {
        return dbName;
    }

    public void setDbName(String name) {
        dbName = name;
    }
```

[6] Martin, Robert C., "Agile Software Development Principles, Patterns and Practices", Prentice Hall, 2003（瀬谷 啓介訳, 『アジャイルソフトウェア開発の奥義』, ソフトバンク クリエイティブ, 2004）

```
    public void connect() {

        // Oracleへの接続処理

        System.out.println("Oracleに接続");

    }

    public void execute() {

        // Oracleに対するSQLの実行

        System.out.println("Oracleに対してSQLの実行");

    }
}

class MySQLDao {

    private String dbName; // データベース名

    public String getDbName() {
        return dbName;
    }

    public void setDbName(String name) {
        dbName = name;
    }

    public void connect() {

        // MySQLへの接続処理

        System.out.println("MySQLに接続");

    }
```

```
    public void execute() {

        // MySQLに対するSQLの実行

        System.out.println("MySQLに対してSQLの実行");

    }
}

class Executer {

    private OracleDao dao;

    public void setDao(OracleDao dao) {

        this.dao = dao;

    }

    public Executer(OracleDao db) {

        this.dao = db;

    }

    public void doDao() {

        this.dao.connect();
        this.dao.execute();

    }
}
```

PBL

```
class Test {

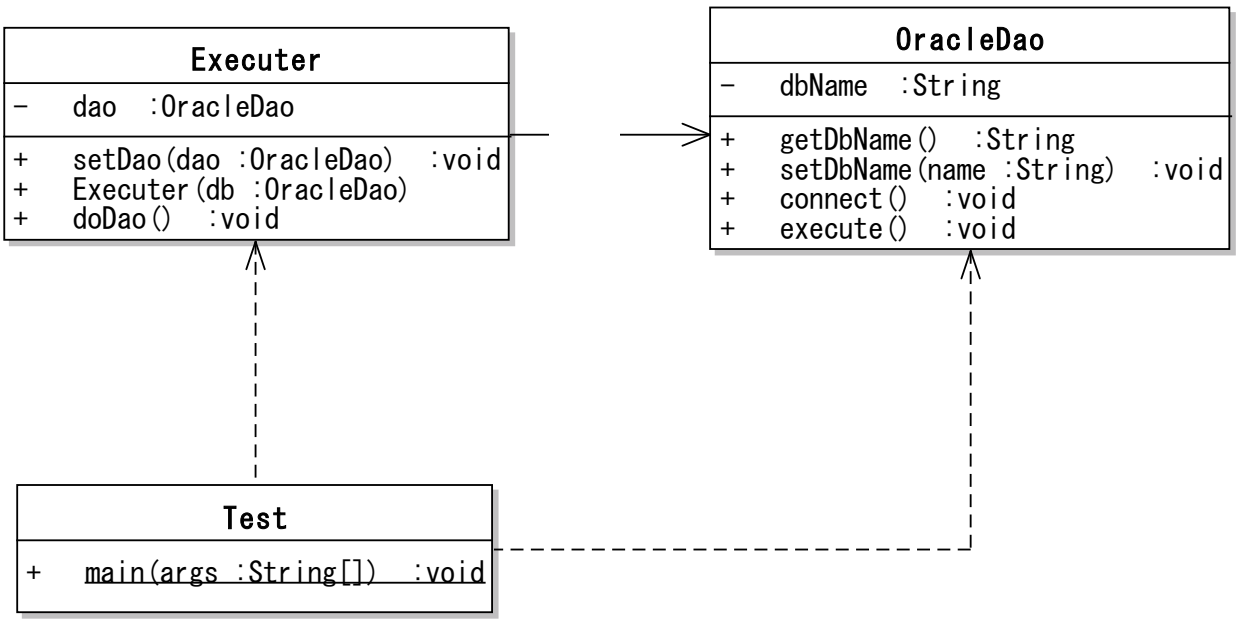
    public static void main(String args[]) {

        Executer e1 = new Executer(new OracleDao());

        e1.doDao();

    }
}
```

●クラス図



●開放-閉鎖原則に合致するソースコード

```
abstract class Dao {

    private String dbName; // データベース名

    public String getDbName() {
        return dbName;
    }

    public void setDbName(String name) {
        dbName = name;
    }

    public abstract void connect(); // 抽象メソッド

    public abstract void execute();// 抽象メソッド
}

class OracleDao extends Dao {

    public void connect() {

        // Oracleへの接続処理

        System.out.println("Oracleに接続");

    }

    public void execute() {

        // Oracleに対するSQLの実行

        System.out.println("Oracleに対してSQLの実行");

    }
```

```
}

class MySQLDao extends Dao {

    public void connect() {

        // MySQLへの接続処理

        System.out.println("MySQLに接続");

    }

    public void execute() {

        // MySQLに対するSQLの実行

        System.out.println("MySQLに対してSQLの実行");

    }
}

class Executer {

    private Dao dao;

    public void setDao(Dao dao) {

        this.dao = dao;

    }
```

```
    public Executer(Dao db) {

        this.dao = db;

    }
    public void doDao() {

        this.dao.connect();
        this.dao.execute();

    }
}

class Test {

    public static void main(String args[]) {

        Executer e1 = new Executer(new OracleDao()); // 変更部分

        e1.doDao();

    }
}
```

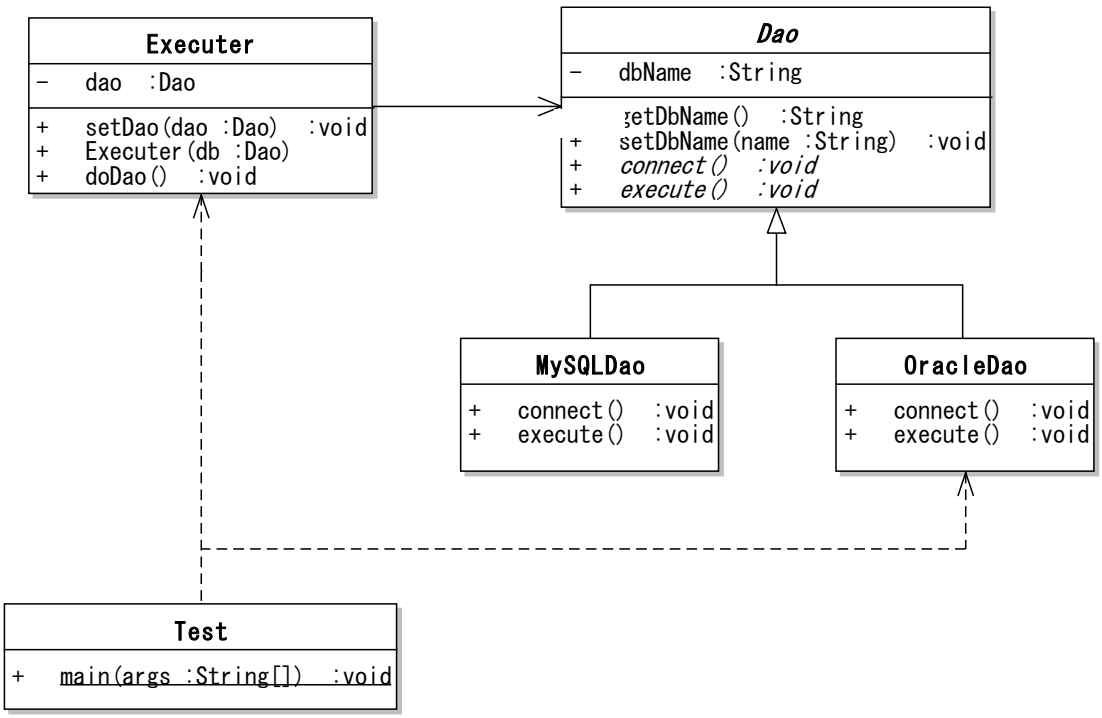
実行結果

```
Oracleに接続
Oracleに対してSQLの実行
```

上記変更部分を「Executer e1=new Executer(new MySQLDao());」と変更した場合は、以下のような結果になります。

```
MySQLに接続
MySQLに対してSQLの実行
```

●クラス図



●開放－閉鎖原則への合致

この場合、クライアント（および抽象クラス、既存のサプライア）は、「閉じて」います（修正しない）。しかし、新しいサプライアを追加することで、ソフトウェア（クライアント＋サプライア）の動作を変化させることができます（「開いている」）。つまり、拡張に対しては「開いて」いるが、修正に対しては「閉じて」います。

●開放－閉鎖原則と結合度

開放－閉鎖原則を適用することにより、クライアントとサプライアの間の結合度を下げることができます。開放－閉鎖原則は、凝集度ではなく結合度にかかわる原則であることが分かります。

●注意点

このサンプルコードでは、サプライアを取り替える場合、クライアント Test は、変更を行わなければならないため、開放-閉鎖原則を完全に満たしているわけではありません。

8.4 開放－閉鎖原則に合致するための設計上の注意点

開放－閉鎖原則に合致するためには、次のような設計上のポイントがあります。

1）クライアントは、具象クラスとなるサプライアの型を直接参照せずに、複数のサプライアに共通なスーパークラス（抽象クラス）やインターフェイスの型を参照する。

2）具象クラスからなるサプライアは、クライアントに対して実装を隠蔽しなければならない。

2）にもかかわらず、クライアントに対して具象サプライアを提供するコードは、サプライアの取り替えの際に修正が必要になるという問題が生じる。

8.5 課題１６

前章までに作成したアーキテクチャ設計の成果物を元に詳細設計を行っていく。まずは、上記で学んだ原則を考慮した上で、分割ビューで作成したモジュールのうち1つを選んでクラス図を作成する。余力があれば、システム全体をクラス図に起こしても良い。

8.6 詳細設計発表（受講者への対応）

課題において、クラス図を作成したモジュールがアーキテクチャ設計で作成した分割ビューの中でどのような役割を果たすか、そして、その役割を果たす為に必要なクラス図における各クラスの果たす役割、それぞれの関係性などについて説明をする。

9 付録 1 オンラインアンケートシステム設計開発仕様

9.1 用語の定義

ネットモニター（モニター）	オンラインアンケートシステムに登録したユーザーを「ネットモニター」と呼ぶ。「オンラインアンケート」などへの回答募集があった際に、「属性」が当てはまる「モニター」は、メールなどにおいて通知を受け、当該「オンラインアンケート」に回答する事で謝礼を得る権利を獲得する。
属性	「ネットモニター」が登録する自身に関する情報。 性（漢字）、名（漢字）、性（かな）、名（かな）、性別、生年月日、郵便番号、都道府県、市区町村、番地、ビル名、国籍、既婚未婚、子供の有無、電話番号、職業、メールアドレス、パスワード
オンラインアンケート	「ネットモニター」を対象にオンラインアンケートシステム上で回答を行う事が出来るアンケート。 「ネットモニター」1名につき1回限り回答可能で、回答後その内容に悪質性がない場合に限り、謝礼が登録された住所に送付される。 質問への回答方法は、単一回答・複数回答・記述回答が存在する。設問数は最大でも30問程度、アンケート毎に対象となる「ネットモニター」の属性を指定する事が可能で、対象にあてはまる「ネットモニター」へは通知が送られそれではじめて回答権が得られる。

9.2 業務の概要

「オンラインアンケートシステム」は、日本国内に在住する「ネットモニター」に対して、「オンラインアンケート」を配信し、オンライン上でそれらに回答をする事を可能とするシステムで、街頭調査や、会場アンケート（会場に被験者を集めてアンケートを取るタイプの調査）などと比較して、手軽かつ、低コストにアンケートによる市場調査を行うことが出来るような仕組みを提供するためのシステムである。
「オンラインアンケート」配信側としては、アンケートに回答して欲しい「ネットモニター」の「属性」を指定した上で、配信を行うことが出来るため、より正確に調査を行いたい分類のユーザーに対する調査を行うことが出来る。
「ネットモニター」は、自身の情報（属性）を登録しておくだけで、「オンラインアンケート」が届き、「オンラインアンケートシステム」上で回答することで、謝礼を受け取ることが出来る。

今回のプロジェクトは、「オンラインアンケート」配信を行いたいと考えている、A 社からの依頼ではじまっている。A 社は現在まで、市場調査を行いたいクライアント企業の依頼で、街頭調査や、会場アン

ケートなどを、オフラインベースで開催していた企業で、現状は「リクルーター」と言われる、「モニター」を探してくる業務委託契約を結んだ人々に対して、案件の情報を流し、条件にあう「モニター」を紹介された場合、「リクルーター」に紹介費用を支払い、当該の「モニター」をアンケートに参加させると言う形のビジネスを展開している。今回の「オンラインアンケート」が完成することで、以下のような効用を得られることを期待している。

1. 「リクルーター」を介さずにA社自身が「モニター」を抱えることが出来る
2. 「リクルーター」と言う属人性を挟まずに条件に合う「モニター」を募集できる
3. 「リクルーター」に支払う紹介料を浮かすことができる
4. 「リクルーター」とのコミュニケーションにかかるリソースを減らすことができる
5. 会場アンケートと比したときに、会場費用を払わなくて済む
6. 会場アンケートと比したときに、現場のスタッフなどの費用を払わなくて済む
7. 会場アンケートなどと比較したときに、早期にアンケート結果の回収ができる

一方、A 社は街頭アンケートや、会場アンケートの他に、「座談会」と言う、一定の条件の人々を一同に介し、ある商品やサービスについて議論をして貰う会の開催も行っている。アンケートなどに比べて、より本音に近い形の意見や、自然な会話の中で生まれる意見が商品開発に活きることがあり、アンケートなどに比して、案件数も単価も高いメインのビジネスである。「座談会」に関しては、オンラインでの開催が基本的に難しいため、継続してオフラインでの会場確保、そして参加者(「モニター」)の集客を行う必要がある。この部分に関しては、「オンラインアンケート」で集客した「ネットモニター」と参加者層が被らない可能性が高いことが示唆されており、「リクルーター」との関係性や、会場確保の優位性などを考えた時に、全てのアンケートをオンラインにしていくかどうかは、ビジネス上の決断となりうる。よって、当面は「オンラインアンケート」と街頭アンケート、会場アンケートは並列して行っていくこととなり、その管理や集客も一部「オンラインアンケートシステム」で行えればと言うのがA社からの要望として挙がっている。

9.3 必須機能要件

■オンラインアンケート配信者

モニター管理	・ ネットモニターの登録情報の管理(登録、修正、一時利用停止、削除)ができる ・ 不適切なモニターに対し強制的に退会処理ができる ・ 一定の条件に該当するモニター(アンケート回答回数、属性)による検索、抽出ができる。

アンケート管理	・ アンケートの作成、修正、削除ができる ・ アンケート回答期限を設定できる ・ アンケート回答上限数を設定できる ・ アンケート回答可能なネットモニターの属性を設定できる ・ アンケートの集計管理ができる ・ アンケートの条件に合うネットモニターに通知を行うことができる
その他	・ 認証機能を有する ・ IP アドレスのホワイトリストを用いたアクセス制限をかけることができる

■ネットモニター

モニター管理	・ ネットモニターとして登録できる ・ 登録情報（属性）を修正できる ・ 退会できる ・ アンケートの回答実績を確認できる
アンケート管理	・ アンケートに回答することができる ・ アンケート結果を観覧できる
その他	・ 認証機能を有する

9.4 非機能要件

■性能要件

オンライン応答時間	・ 同時接続数 500 台の状態で、各処理毎の平均応答時間3秒以内

■信頼性要件

サービス時間	・ ネットモニター 24時間365日。ただし、年末年始(12 月 29 日-1月 3 日)および保守などの計画停止を除く ・ オンラインアンケート配信者 午前6時ー午後 11 時
可用性	・ ネットモニター 99.5% ・ オンラインアンケート配信者 99.0%

10付録 2 RFP

10.1 はじめに

株式会社〇〇〇システム　御中

拝啓　貴社ますますご盛栄のこととお喜び申し上げます。

さて、このたび当社におきまして「オンラインアンケートシステム」の構築を検討いたしております。つきましては貴社からのご提案を依頼したく本提案依頼書を発行致します。
ご多忙の折誠に恐縮では御座いますが、何卒宜しくお願い申し上げます。

10.2 提案依頼の趣旨

弊社は、広く言う所での市場調査の代行、開催支援、及びコンサルテーションを主な業務とする企業で御座います。平たく言えば、クライアントとなる企業様（製造業、製菓、飲食、小売などの大手企業様との取引実績が御座います）が新商品や新サービスの開発を行う際の、街頭アンケート、会場アンケート、座談会、ホームユースと言った、市場調査業務を、クライアント様のニーズに沿ってお手伝いをする事でいくばくかの報酬を頂く形の業務です。

具体的には、以下のような事を主業務として行っております。

1. 対象となるユーザー（モニターと呼んでおります）を、年齢、性別、住所、お仕事、年収、趣向など様々な切り口の条件によって絞り込み、クライアント企業様の探す顧客像のモニターを斡旋する業務

2. クライアント企業様が行いたい調査に合わせて、会場の予約・設営、当日の運営スタッフの確保・教育など、調査実施の代行・協力する業務

3. 調査を行いたい内容のヒアリングから、実際にアンケートや座談会内容の計画をコンサルテーションする業務

昨今、人口動態の変動、購買動向の変化、ユーザーのニーズの多様化などの外部的な要因でのユーザーヒアリングのニーズ増加に加え、弊社クライアント企業様側でも、より統計的なアプローチでのマーケティングを行いたいと言う要望が増えており、幸いにも市場調査実施件数が前年比でも 15%以上延びている状況で御座います。

そのようなニーズの変化を受け、より多くのクライアント様のニーズを満たすことが出来るようにするために、現状全て対面で行われている市場調査業務の一部をオンライン上で行うことが出来るシステムを構築することで、より低価格かつシームレスに調査を行うことが出来ないかと検討しております。

一方、ご想像に難くはないかと存じますが、弊社業務は、クライアント企業様の新規製品などの秘匿情報や、モニターとして参加される個人の皆様の、いわゆる個人情報にあたる情報を扱う業務になります。弊社には内製のバッチシステムを作成、簡易なSQLの利用、また社内ネットワークの管理を行うITチームは存在するものの、個人情報を適切に扱った上で、外部向けにシステムを公開出来るだけの開発部隊は存在いたしません。

そこで、貴社には大変お手数では御座いますが、機能要望などを踏まえた上で、更に可用性、セキュリティなど幾つかの性能要件を満たす形での、システム開発提案を頂きたく存じます。ご提案は複数社に依頼しており、頂きましたご提案を弊社プロジェクトチームにより総合的に判断の上、案件発注する企業様を決定させて頂きたく存じます。基本的には、弊社提案依頼内容の要件が満たされている事は前提として、弊社の解決したい問題をより適切に解決出来るシステム提案を頂いた企業様に発注をさせていただこうと考えております。

以下、一般的なRFPの形式に沿って記述させて頂いておりますが、個別にご質問等御座いましたら、下記事務局宛にご一報下さいませ。その他事務事項は以下に記載させて頂きましたのでご確認下さい。

- 提案書作成に際しまして、秘密保持契約を締結させて頂きたく。ヒアリングの際に付録に御座います秘密保持契約書に署名捺印の上お持ち下さい。
- 提案書作成にかかわる費用はすべて貴社負担でお願い致します。
- RFPに関する説明会は全社向けには事前には執り行いません、個社毎にQ&A及びヒアリングの機会を設けさせて頂きますので、ヒアリング内容がある程度固まったタイミングでご連絡を頂ければと存じます。
- ご提案書は、電子メールの添付にて平成〇〇年2月7日17時までに下記の事務局にご提出下さい。
- 提出いただいた提案書に関して、プレゼンテーションによるご説明の機会を頂きたく存じます。また、プレゼンテーションはプロジェクトマネージャ候補の方が執り行って頂くようお願いいたします。
- 事務局

  株式会社 Aリサーチ 市場調査部 オンラインアンケートシステム導入事務局

  TEL 03-1234-1234 FAX 03-1234-1235

  担当:経営企画部 山田花子 hanako_yamada@aaaa.co.jp

  情報システム部 田中太郎 taro_tanaka@aaaa.co.jp

以上となります。それではご提案の程お待ちしております。引き続き宜しくお願い致します。

10.3 業務要求

（1）業務基本要求

（ア）ネットモニター

① ネットモニターが自身で本人情報の登録を行うことが出来る
② ネットモニターがアンケート参加履歴など自身の情報を確認することが出来る
③ 退会処理を行うことが出来る
④ ネットモニターが条件にアンケートが発行された際にそれを知る（通知）ことが出来る
⑤ ネットモニターが自身でアンケートに回答することが出来る

（イ）管理者

① モニターの情報（本人入力情報、アンケート参加履歴など）を検索・観覧・抽出することが出来る
② 不適切なモニターの退会処理が出来る
③ オンラインアンケートの作成が出来る（アンケート内容の設定、参加可能モニターの条件など）
④ アンケート結果の集計をすることが出来る

（2）業務機能要求

基本的には管理者ページと、ネットモニター用ページの二つの作成を検討する。それぞれの機能要求は以下の通り。

（ア）管理者画面

① ネットモニター管理

1. ネットモニターの登録情報の管理（登録、修正、一時利用停止、削除）ができる
2. 不適切なネットモニターに対し強制的に退会処理ができる
3. 一定の条件に該当するネットモニター（アンケート回答回数、属性）による検索、観覧、抽出ができる。
4. ネットモニターのオンラインアンケート参加履歴が確認できる

② アンケート管理

1. アンケートの作成、修正、削除ができる
2. アンケート回答期限を設定できる
3. アンケート回答上限数を設定できる
4. アンケート回答可能なネットモニターの属性を設定できる
5. アンケートの集計管理ができる
6. アンケートの条件に合うネットモニターに通知を行うことができる

③ その他

1. 認証機能を有する
2. IPアドレスのホワイトリストを用いたアクセス制限をかけることができる

（イ）ネットモニター画面

① ネットモニター画面

1. ネットモニターとして登録できる
2. 登録情報（属性）を修正できる
3. 退会できる
4. アンケートの回答実績を確認できる

② アンケート画面

1. アンケートに回答することができる
2. アンケート結果を観覧できる

③ その他

1. 認証機能を有する

10.4 技術要求

1. IT インフラ要求

（ア）管理画面、モニター画面ともスタンドアロンのシステムやモバイルのネイティブアプリではなく、ユーザー・インタフェースには Web ブラウザを使いアプリのダウンロードなどのユーザー側の導入コストを下げると同時に、開発費用も一元化してまずは安価に進められるよう取り計らいたい

（イ）管理画面を使用する弊社マシンは基本的には 2016 年販売の xxxx で OS は最新版の yyyy を利用しており、標準ブラウザは zzzz を利用している事をご想定下さい。ユーザー側は出来る限り汎用性の高い形でお作り頂き、もし対応ブラウザに制限があるようであれば、必ずユーザーにわかる形で明示して頂きたい

（ウ）管理画面におけるユーザー抽出の機能では csv 形式での抽出を基本としたい

（エ）オンプレミスサーバーでの管理は障害発生時に効力を発揮するとあるが、弊社に対応人員を常に割くだけの余力がないため、後の運用要求にも記載するが基本的にはクラウドサーバーでの管理が良いと考えている。この部分に関しては検討の余地があるため、ネットワーク構成に関しては、特記事項があれば提案書に含めて頂きたい

（オ）本番環境とは物理的にも論理的にも切り離された、テスト環境を用意して頂きたい。テスト環境には、本番環境のデータのうち、個人情報にあたる部分をマスクした形のデータが入っている事が望ましい。（実際の環境に近い形でテストが行えるため。

2. システム能力要求

（ア）現在の弊社の一意なアクティブ利用者数は2万人前後。うち30％程度が、10代から40代でオンラインアンケートにも積極的に参加して頂けることを想定している（ユーザー数6000名程度）。ただ、今後の拡大を考えれば当然 10000 アクティブユーザーは想定の範囲内とさせて頂きたく、複数のアンケートが同時に配信されたとして、瞬間の同時接続数はその5％程度の 500 台接続で、ユーザーが不快に感じない範囲で平均応答時間が3秒以内となるようにしたい

3. 障害対策要求

サーバーの障害対応は重視する。ハードウェア障害だけでなく、停電やコンピュータ・ウイルス対応なども十分に考慮したい。

（ア）ネットモニター画面：

市場調査においては、クライアント企業様に調査結果を返さなければ行けない期日があり、これに遅れてしまう事は致命的。障害発生などが長引く事で、アンケート回答数が十分に得られないと言う事があった場合、当社ビジネスを大きく毀損する可能性がある。よって、ネットモニター画面は24時間365日稼働を基本（ただし、年末年始（12 月 29 日-1月 3 日）および保守などの計画停止を除く）とし、可用性も 99．5%以上を担保したい。

（イ）アンケート管理画面：

アンケート管理画面に関しては、当面は弊社社員による入力・管理が行われる事になる。ビジネスモデル的に、クライアント様からの急な要望などがあったとしても、基本的にはアンケート掲載までは数日の猶予がある。午前6時ー午後 11 時の一般的な業務時間を必須稼働時間とし（勿論、それ以外の時間も、サーバーを止める必要はない）、可用性は当該時間帯において99．0%以上担保が望ましい。

4. セキュリティ要求

弊社は、ネットモニター、クライアント企業様、共に大変気密性の高い情報を扱う事となる。セキュリティ要件は大変重要となる。現状、弊社にセキュリティのスペシャリストが存在しないため、一般的な要件を挙げるが、より機密度が高まるような方法があればご提案頂きたい。

（ア）ネットモニター画面：

利便性との兼ね合いにはなるが、パスワードの要件（12文字以上、記号、大文字、数字を含む組み合わせ）を設定し、web である事から総当たり攻撃に対する対策として、captcha などのロボット対策および、複数回パスワード入力失敗で、アカウントロックなどの基本的な対策は設けたい。また、二段階認証（SMS 利用）も要検討としたい。

（イ）アンケート管理画面：

弊社 IP アドレスからのアクセスのみを受け入れるようにした上で、パスワードの要件（12文字以上、記号、大文字、数字を含む組み合わせ）を設定し、web である事から総当たり攻撃に対する対策として、captcha などのロボット対策および、複数回パスワード入力失敗で、アカウントロックなどの基本的な対策は設けたい。
また、社員の中でもアンケート登録のみが出来る人、ユーザー情報を参照・検索・抽出出来る人、社員アカウントの管理ができる人、と三段階でユーザー権限を分けたい。
IP アドレスだけで不十分な可能性がある場合は、クライアント証明書を発行するなど、PC レベルでの制限（クライアント証明書のリモート破棄も含める）がかけられるとなお良い。ここは、ご提案時にご説明頂ければと考えている。

10.5 運用要求

1. 技術要求でも述べたように、システム管理専用の要因確保が困難なため、システム保守の負荷が軽微な形が望ましい。よってクラウド型で、遠隔、外注運用もあり得る形が良いのではないかと考えている。一方で、オンプレミスサーバーでも低コストでの運用が可能である方法があればご提案頂きたい。

2. システム受け渡し時に、従業員に対するトレーニングや研修などもご提案頂きたい。

3. 将来的に外注でのシステム保守・運用を考えていること、必要事項はドキュメントとして残して頂きたい。

4. 将来の機能拡張を引き受け可能である場合は契約の形式などご提案時にご教示頂きたい。

11各フェーズの解答例と解説

11.1 課題１に対する解答事例と解説

付録2:オンラインアンケートシステムのRFPの中から関連する情報を抜き出している。必ずしもこの形が正解とは限らないので、正解例として捉えて欲しい。重要なことは、RFPの中から必要な要件を抜き出して、それを整理すること。

1. 提案の提出期限と提出先

提出期限: 平成○○年2月7日17時
提出先:
株式会社 Aリサーチ 市場調査部 オンラインアンケートシステム導入事務局
TEL 03-1234-1234 FAX 03-1234-1235
担当:経営企画部御中 山田花子様 hanako_yamada@aaaa.co.jp
情報システム部御中 田中太郎様 taro_tanaka@aaaa.co.jp

2. 提案前RFP説明会を行う条件と日時

提案前RFP説明会は行われない。代わりに、個社毎のQ&A及びヒアリングが行われる。

3. 提案作成要領

提案書作成にかかわる費用はすべて供給者負担
事前に秘密保持契約の締結が必要
提案に関してプレゼンテーション形式による説明が求められている

4. 提案の評価方法

機能要望などを踏まえた上で、更に可用性、セキュリティなど幾つかの性能要件を満たす形での、システム開発が求められる。
提案及び提案発表を、取得者企業プロジェクトチームに行い、その結果で判断される。
提案依頼内容の要件が満たされている事は前提として、取得者企業の解決したい問題をより適切に解決出来るシステムが評価される

5. RFP 窓口担当者名とメールアドレス

株式会社 Aリサーチ 市場調査部 オンラインアンケートシステム導入事務局
TEL 03-1234-1234 FAX 03-1234-1235
担当:経営企画部御中 山田花子様 hanako_yamada@aaaa.co.jp
情報システム部御中 田中太郎様 taro_tanaka@aaaa.co.jp

6. その他、供給者が回答するために必要な情報

個別の質問に関しては事務局へご連絡
取得者企業は、市場調査の代行、開催支援、及びコンサルテーションを主な業務としている
市場調査業務の一部をオンライン上で行うことが出来るシステムを構築したい
ネットモニター画面と管理者画面の二つの大きな塊の作成が必要
扱う情報を鑑みると、セキュリティ対策は重要な要件の一つとなる

11.2 課題２に対する解答事例と解説

付録2:オンラインアンケートシステムのRFPの中から関連する情報を抜き出している。必ずしもこの形が正解とは限らないので、正解例として捉えて欲しい。重要なことは、RFPの中から必要な要件を抜き出して、それを整理すること。

9. プロジェクトの目的と目標

近年、市場調査を行うニーズが高まり、実施案件が年々増えている。そのような環境の中、より多くのクライアント企業のニーズを満たすため、市場調査業務の一部をオンライン上で行う事が出来るシステムを構築することが目標

10. 重要成功要因

ネットモニター画面（市場調査協力者が使用する画面）、管理者画面（取得者企業社員が業務遂行に使用する画面）の二つを作成し、それらを、セキュアな形で提供する事が求められている。低価格かつシームレスに市場調査業務を遂行出来る事が出来るようなシステム提供が重要成功要因となる。

11. 現行システムの機能仕様

現在は、全てオフラインで管理されているためシステムは存在しない。

12. 計画中のシステムの機能仕様

ネットモニター画面と管理者画面の二つが存在する。

ネットモニター画面:

・ネットモニターが自身で本人情報の登録を行うことが出来る

・ネットモニターがアンケート参加履歴など自身の情報を確認することが出来る

・退会処理を行うことが出来る

・ネットモニターが条件にアンケートが発行された際にそれを知る(通知)ことが出来る

・ネットモニターが自身でアンケートに回答することが出来る

・パスワードの要件(12文字以上、記号、大文字、数字を含む組み合わせ)を設定し、web である事から総当たり攻撃に対する対策として、captcha などのロボット対策および、複数回パスワード入力失敗で、アカウントロックなどの基本的な対策は設けたい。また、二段階認証(SMS 利用)も要検討としたい。

管理者画面:

・モニターの情報(本人入力情報、アンケート参加履歴など)を検索・観覧・抽出することが出来る

・不適切なネットモニターに対し強制的に退会処理ができる

・オンラインアンケートの作成が出来る(アンケート内容の設定、参加可能モニターの条件など)

・アンケート結果の集計をすることが出来る

・弊社 IP アドレスからのアクセスのみを受け入れるようにした上で、パスワードの要件(12文字以上、記号、大文字、数字を含む組み合わせ)を設定し、web である事から総当たり攻撃に対する対策として、captcha などのロボット対策および、複数回パスワード入力失敗で、アカウントロックなどの基本的な対策は設けたい。

・社員の中でもアンケート登録のみが出来る人、ユーザー情報を参照・検索・抽出出来る人、社員アカウントの管理ができる人、と三段階でユーザー権限を分けたい。

13. 性能要求

・同時接続数 500 台で平均応答時間が3秒以内
・ネットモニター画面は24時間365日稼働を基本とし可用性は99．5%以上
・午前6時ー午後 11 時の一般的な業務時間を必須稼働時間とし可用性は当該時間帯において99．0%以上

14. ハードウェア要求

・オンプレミスサーバーではなくクラウドサーバーを使用する事
・本番環境から完全に切り離された(別サーバーで)テスト環境を用意する事

15. ソフトウェア要求

・ ユーザー・インタフェースには Web ブラウザを使いアプリのダウンロードなどのユーザー側の導入コストを下げる
・ 2016 年販売の xxxx で OS は最新版の yyyy を利用しており、標準ブラウザは zzzz を想定

11.3 課題３に対する解答事例と解説

RFP の内容を読んだ後に、内容の中でより詰めた方が良いこと、また要件として不完全で曖昧になっていることを洗い出す。解答例では、よりセキュリティ要件を固めるために曖昧さを排除する質問、既存のビジネスとシステムを絡める事が出来るのか出来ないのかを問う質問、解決しようとしている問題をよりクリアにする質問、今後のプロジェクトの進め方に関する曖昧さを排除する質問などを挙げている。

RFP の内容への質問事項の洗い出し

Ex)

- セキュリティ要件に関して、セキュリティはユーザビリティとある程度逆相関関係にある、どの程度までユーザーに不便さを受け入れさせる事が出来るか？
- 既存のビジネスと今回新たに起こそうとしているオンラインアンケートシステムにおけるビジネス、どのように共存させていくつもりか？
- 既存のビジネスからのコスト削減と言う事が銘打たれているが、既存のビジネスにおけるコスト形態をより詳しくご教示下さい。
- RFP 上に記載がある質問に関しては適宜とあるが、受注後も同様に適宜質問する形態が望ましいか、それとも定期的にミーティングを持つことは可能か？

11.4 課題 4 に対する回答事例と解説

これは各チーム異なる内容となるはずなので、解答もないのだが、サンプルとして何例か挙げている。チームとしての強みを把握するために行うアクション。

自チームの技術的特徴の棚卸し

Ex)

- 金融機関でセキュリティ業務を行っている
- 月間 100 万 PV を超える EC サイトの運営を行っている
- 深層学習によるデータサイエンスを行っている
- ブロックチェーンネットワークの作成を行っている
- etc.

11.5 課題 5 に対する解答事例と解説

自チームの技術的な特徴を活かした内容（例えばブロックチェーンの経験があるものがいるのであれば、これを提案にいれる事で、新規性や独自性は高まる、勿論相手あってのものなので、それがウケるかどうかは判断が必要）を記述する。この時点では数は多ければ多いほどよく、実現性が低くてもとにか

く色々と出してみる事が重要となってくる。
技術的な強みだけではなく、ビジネス的に独創性があること、RFP では述べられていないが、出来れば取得者企業が喜びそうな事をアイディアとして出してみる。

『独自性の高い提案』のブレインストーミング

- ブロックチェーンを使ったネットモニターの参加記録管理
  （セキュリティ向上）
- LINE や Facebook と言った SNS と連携した友人紹介機能
  （現在は基本的に既存顧客のうち一部がオンラインアンケートシステムに流れる事を想定しているが、新規顧客開拓を検討してはどうか？）
- 取り溜まるデータに深層学習を使用し、アンケート内容のリコメンデーションを行う
  （企業向けコンサルテーションビジネスに活用）

11.6 課題 6 に対する解答事例と解説

課題5でブレインストーミングした中から、実際に実現可能性などを考えて幾つかに絞り込んで見る。実際に『提案』において受け入れられるかどうかは、独自性、実現可能性、期待される効果、そして取得者企業のニーズなどによって決まる。可能であれば技術的な部分だけでなく、先方のビジネスの広がりも考えて絞りこめるとなお良い。

『独自性の高い提案』の絞り込み

- 取り溜まるデータに深層学習を使用し、アンケート内容のリコメンデーションを行う
- LINE や Facebook と言った SNS と連携した友人紹介機能

11.7 課題 7 に対する解答事例と解説

課題6で絞り込んだ提案を実際に質問として落とし込む。自社の技術的なアピールだけではなく、供給者企業に取って価値がある形に落とし込む事が重要。

『独自性の高い提案』の可能性を探る質問を用意

・ オンラインアンケートを行う事の最大の利点は御社の仰る通り、コスト削減にあるかと思います。ただ、同時に定量データが取り溜まる仕組みが出来ることを有効活用する事はご検討可能でしょうか？弊社では、深層学習を用いた AI による、データ分析、予測、介入と言う所を強みとしています。御社クライアント様向けに、より効率的に市場調査を行う事が出来るアンケート内容などをリコメンデーションする仕組みを作れるのではないかと検討していますが、御社のビジネスにとってメリットはございますか？

・ 現在、御社では既存の登録モニター様を、オンラインアンケートシステムに登録させる事を考えているように拝察されます。一方、オンラインアンケートシステムは、現状のオフラインとくらべて、簡単に友人に紹介出来たり、気軽に登録が出来るなどのメリットがあるように感じられます。例えば、このシステムを使って新規顧客獲得を行うような施策を打つことにメリットは感じられますか？

11.8 課題８に対する解答事例と解説

ネットモニター画面(MON)と管理者画面(ADM)について3つずつ優先度の高そうな機能要求を記述してみた。実際にはまだかなり多くの機能要求が存在する。余力がある受講生はそれらについても是非記述して見て欲しい。

要求	MON-01	ネットモニターが本人情報の登録を行うことができる。
	理由	ネットモニターは自身の情報を入力することで、オンラインアンケートの条件に合致して、参加する事が出来るオンラインアンケート数が増える。また、報償送付先情報を作成・更新させることによって、送付ミスを防ぎたい。

要求	MON-02	ネットモニターアカウントにログイン出来る
	理由	ネットモニターアカウントに紐づく個人情報は、それを観覧する権利がある人(ここでは、ネットモニター登録者本人)のみとなるべきで、その認証を行いたい。

要求	MON-03	ネットモニターは自身の条件にあったオンラインアンケートに回答出来る
	理由	ネットモニターは、オンラインアンケートに回答をする事で、報奨が与えられる。こうすることでアンケート回答数を確保したい。

要求	ADM-01	管理者用アカウントにログイン出来る
	理由	管理者用アカウントは、ネットモニターの個人情報の観覧や、アンケートの参加条件やコンテンツの観覧・変更が行えるため、、それぞれのアカウントにおいて認証を行いたい。

要求	ADM-02	アンケートの編集が出来る
	理由	新しいアンケートの作成から、アンケート回答参加のための条件設定、アンケート内容の入力（回答方式の選択）、回答期限や謝礼などのその他静的情報の入力を行うことで、ネットモニターにオンラインアンケートの配信を行いたい。

要求	ADM-03	アンケート結果の集計が出来る
	理由	回答済みのオンラインアンケートの結果を簡単に集計してクライアント企業に提出したい。集計結果に応じて分析を行いたい。

11.9 課題９に対する解答事例と解説

上述の機能要求を更に詳細に渡り分解し、機能要件とした。〇〇をしたいと言う要求から、それを実現するのに必要な機能に分割をしていく。開発を進めて行く上で適切な粒度に機能要件が分割されているとより、アーキテクチャ設計以降のステップが進めやすくなる。正常フローのみでなく、異常フローについてもカバー出来ているとなお良い。

要求	MON-01	ネットモニターが自身で本人情報の登録を行うことができる。
	理由	ネットモニターは自身の情報を入力することで、オンラインアンケートの条件に合致して、参加する事が出来るオンラインアンケート数が増える。また、報償送付先情報を作成・更新させることによって、送付ミスを防ぎたい。
	説明	生年月日、性別、居住地域、電話番号(連絡用)、職種 /業界、年収、既婚・未婚、などが基本的なオンラインアンケート参加者の絞込条件になっているため、それらを入力させる。その他にも画面上あまり多すぎる印象とならない形が取れるのであれば、子有り・無し、喫煙の有無・頻度、飲酒の頻度、所有車種、などがあり得る。
□□□	MON-01-01	システムは、ネットモニター本人の登録時 Email とパスワードがログイン画面から入力されたら、それらを照合して正しければログインさせる。
□□□	MON-01-02	システムはネットモニター会員管理画面を表示させる。
□□□	MON-01-03	ネットモニター管理画面から編集画面に飛び、本人情報の登録が出来る

□□□	MON-01-04	編集後に、会員管理画面に戻り、登録した情報を確認出来る

要求	ADM-01	管理者用アカウントにログイン出来る
	理由	管理者用アカウントは、ネットモニターの個人情報の観覧や、アンケートの参加条件やコンテンツの観覧・変更が行えるため、、それぞれのアカウントにおいて認証を行いたい。
	説明	アドレスは社内のイントラネットからのアクセスに限り、管理者によって設定されたクライアント証明書を持つマシン以外のアクセスを受け付けない。Emailとパスワードでログイン可能で、アカウント毎に三種類の権限(アンケート編集のみ可能、アンケートの編集、ネットモニターの情報観覧のみ可能、全ての機能要件として実装される事が実施可能)のうちどれかを持つ。
□	ADM-01-01	システムは、社内の認められたPCから、発行されたEmailとパスワードがログイン画面から入力されたら、それらを照合して正しければログインさせる。
□	ADM-01-02	システムは、社外からのアクセス、許可されないアクセス、Emailまたはパスワードのミスマッチ、全ての場合において、ログインに失敗しましたと言う文言を表示する。

11.10 課題１０に対する解答事例と解説

非機能要求に関しても、洗い出しを行った上で、要件として切り出しを行う。

要求	SEC-01	本番環境はプライベートクラウド上に配置され、外界のネットワークから直接アクセスさせたくない
	理由	ネットモニターの個人情報、クライアント企業の新製品に関するアンケートなどの機密情報を扱うため、必要十分以上のセキュリティ対策が練られている必要があるから。 また、それら機密情報の全てを持つ、本番サーバーに関しては最も厳重なセキュリティが施されている必要があるため。
	説明	本番環境上には通常データベースへのアクセス情報であったり、その他連携サービス等のアカウント情報などの機密情報が多く保持されている。これらの漏洩はサービスに致命的な損害を与える事は明白であるため、外界からの直接アクセスを禁ずる事で、多くの攻撃に対して備える事が出来る。

要求	SPC-01	同時接続数は500台を想定し、平均レスポンス時間は3秒以内としたい
	理由	既存の取得者企業のアクティブ利用者数は2万人前後。うち30%程度が、10代から40代でオンラインアンケートにも積極的に参加することを想定している(ユーザー数6000名程度)。市況を鑑みて、今後の拡大も考えれば 10000 アクティブユーザーは想定の範囲内とおく。 アンケートが配信されたタイミングでのアクセスの集中を考えると、瞬間の同時接続数はその5%程度の 500 台接続は最低限賄える事が望ましく、ユーザーが不快に感じない範囲で平均応答

		時間が3秒以内となるようにしたい
	説明	レスポンスタイムに換算されるのは、ネットワーク遅延、サーバー処理、DB クエリ、描画処理である。この中で、最も懸念されるのは DB クエリで、案件数が増えてきた際に、3秒以内のレスポンス時間を確保するためには、データ構造及びインデックスなどのクエリチューニングが必須となる。
□	SPC-01-01	DB クエリにおいて計算量のオーダーが nlog(n)以下となるように、インデックスとデータ構造の調整を行う
□	SPC-01-02	データ量が増えてきた場合に備えてページング処理を行う。

要求	MON-AVA-01	24 時間 365 日稼働で可用性は 99.5%(年末年始:12 月 29 日-1 月 3 日および保守などの計画停止を除く)
	理由	市場調査においては、クライアント企業に調査結果を返さなければ行けない期日があり、これに遅れてしまう事は致命的。障害発生などが長引く事で、アンケート回答数が十分に得られないと言う事があった場合、取得者企業のビジネスを大きく毀損する可能性があるため。
	説明	可用性 99.5%はコールドスタンバイでも対応可能である、一方で取得者企業の希望で、運用の人的コストを出来るだけ下げたいという要望があるため、コールドスタンバイのマシンを異常事態発生時に手動で起動することや、コールドスタンバイマシンの定期テストは難しいように感じる。そこで、ロー

		ドバランサーを使用して2台のマシンが常時稼働している形式を取る。
□	MON-AVA-01-01	ロードバランサーが、ピンエコーなどで問題を検知し問題発生時にアラートが飛ぶ形を取る。1台稼働の状態を限りなく短くする事で、99.5%の可用性を達成する。
□	MON-AVA-01-02	保守などの計画停止を Email 及び画面上でユーザーに知らせる機能

11.11 課題１１に対する解答事例と解説

今回の案件において最も重要な品質特性はセキュリティ性であると考えられた。よって、以下のような「管理者画面」と言う最も気密性の高い情報へのアクセス権がある

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 本件では、アカウントの認証（Email/Password）の他にクライアント証明書などを利用した PC 毎の識別を行う。 3. アクセス元 ・ 事務所内ネットワーク以外
刺激	「管理者画面」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身元を正しく識別されていない、もしくは、社内ネットワーク外からのアクセスの場合、サービス/システムに関連するデータへのアクセスを拒否する。 システムは識別された身元の権限を確認し、権限がないデータ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や権限の足りない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

11.12 課題１２に対する解答事例と解説

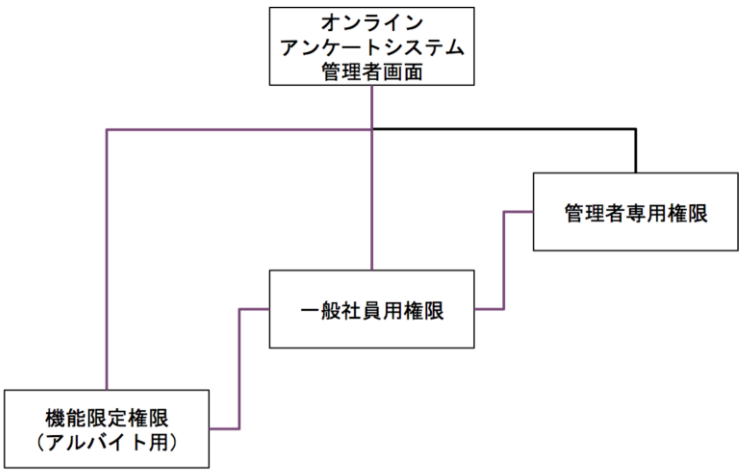
アーキテクチャドライバとして以下の二つを想定した上で、アーキテクチャパターン・実現手法の組み合わせを考える。

1.「管理者画面には、社内からのアクセス、かつ、識別されたマシン、ユーザーからのアクセスのみを許す」

2.「最大瞬間接続数は 500 を想定し、平均レスポンスタイムは3秒以下に抑える事」

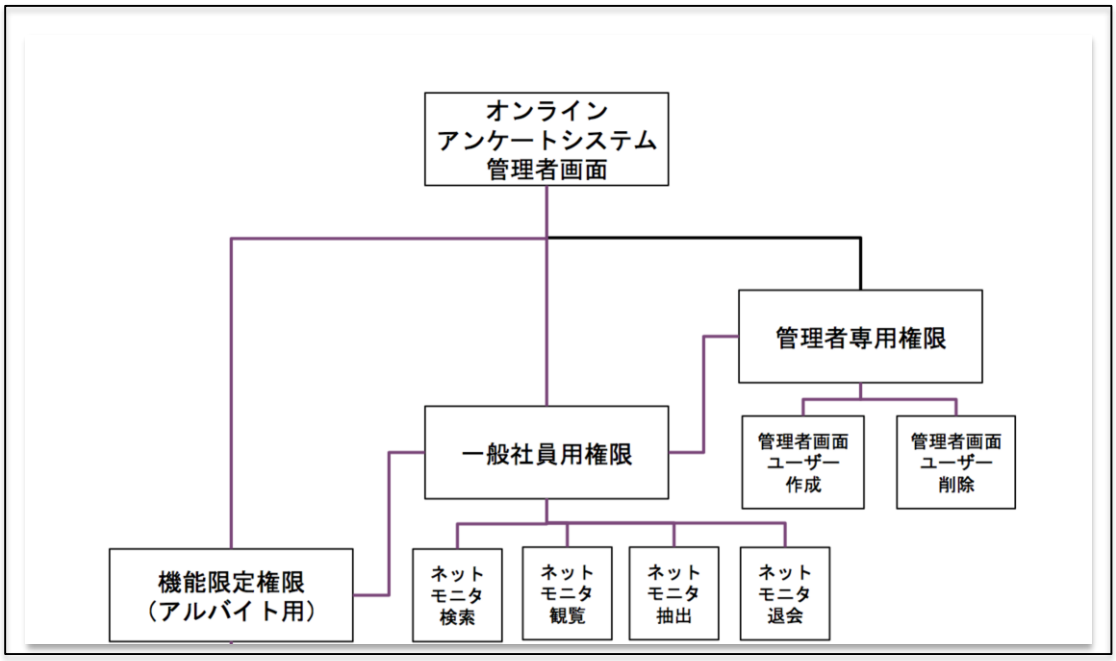
注意点として、非機能要件として要件定義で発見された一方で、重要度が比較的に低くアーキテクチャドライバとして選択されなかった要件は、アーキテクチャ設計上満たすべき制約として考慮される必要がある場合がある。

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	「ユーザーを認証する」、「アクセスを制限する」

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	「ユーザーを認証する」 ユーザー毎に権限関係が異なる場合、機能の塊をそれに沿って分けていく事が望ましい。ある大きな塊の中で一部の画面はアクセスが出来て、一部はアクセスが出来ないと言う状態はユーザーに取って使いやすさの品質要件に大きな影響が出る。よって大局的な分割構造も、権限単位で分けていくと良いのではないかと推察される。 「アクセスを制限する」 基本的に管理者画面は全てにおいて、承認されたマシンから、かつ、承認された IP からのアクセスのみを承認するため、他の機能性は品質特性に与える影響は軽微であると考えられる。
モジュールの大局的な分割構造	

11.13 課題１３に対する解答事例と解説

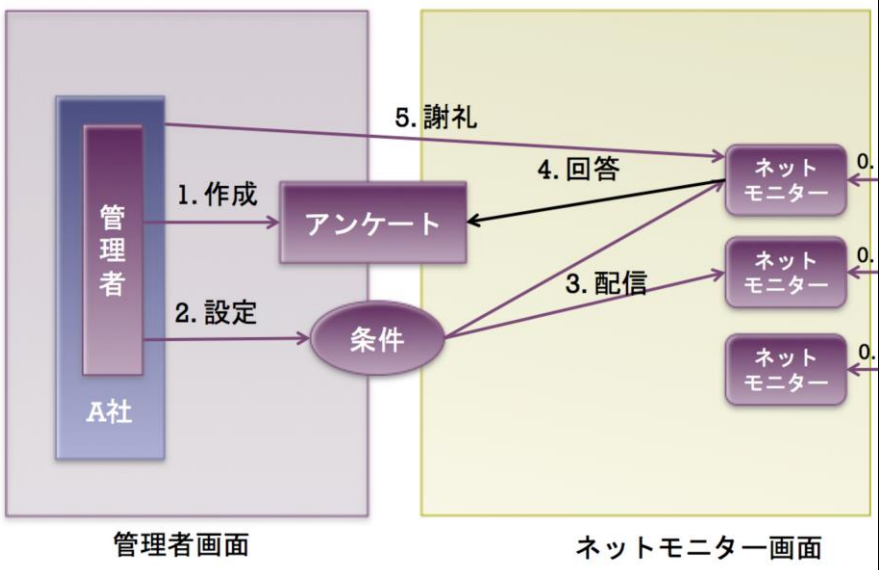
課題12で記述したモジュールの大局的な分割構造に対し、機能要件を適用することで、機能性の分配を行う。



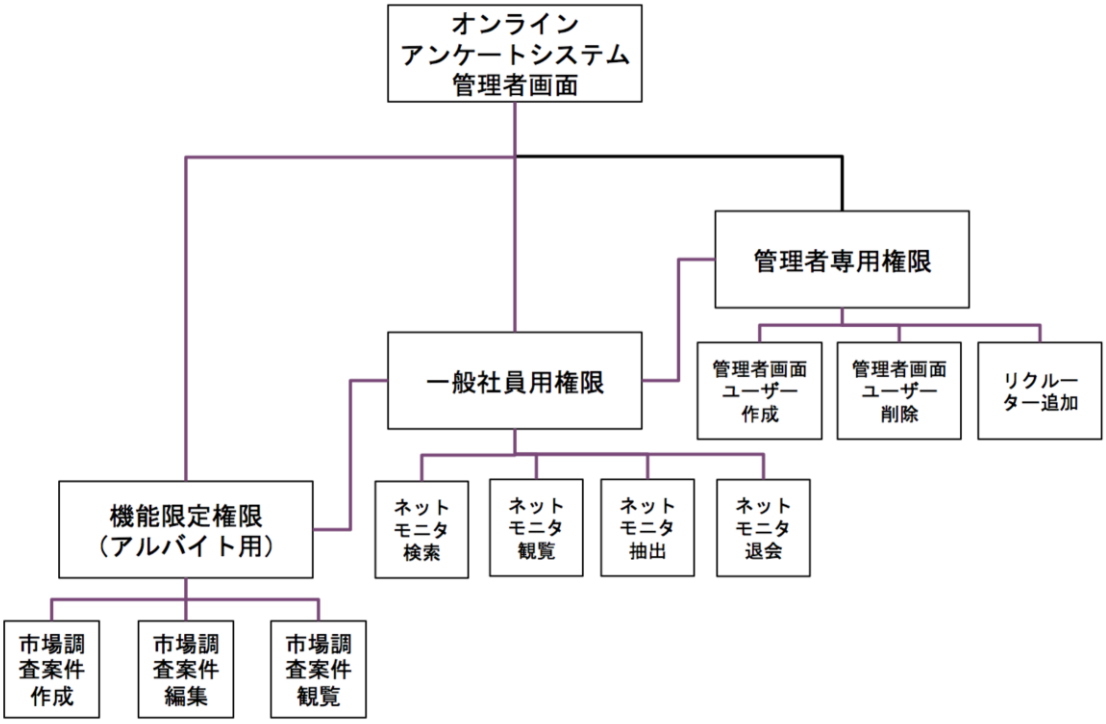
11.14 課題１４に対する解答事例と解説

文書化項目	内容
プライマリプレゼンテーション	市場調査代行業務の一部をオンラインで置き換える「オンラインアンケートシステム」における管理者画面側の大きなモジュール分割構造を示す「分割ビュー」を示す。 ユーザー権限毎に、大きく３つのサブシステムに分類されている。 1. 機能限定権限 2. 一般社員権限 3. 管理者権限 「オンラインアンケートシステム」全体としては、ネットモニター画面と管理者画面から構成される。それぞれの関係は要素カタログで簡易に示す。 「管理者画面」では、大きく分けて 1. アンケート管理（作成、編集、削除） 2. ネットモニター管理（検索、観覧、抽出、退会） 3. 管理者画面のユーザー管理 が行われる。

要素カタログ	市場調査とは: ネットモニターとは: オンラインアンケートシステムに登録したユーザーを「ネットモニター」と呼ぶ。 「オンラインアンケート」などへの回答募集があった際に、「属性」が当てはまる「ネットモニター」は、メールなどにおいて通知を受け、当該「オンラインアンケート」に回答する事で謝礼を得る権利を獲得する。 オンラインアンケートシステムとは: 「ネットモニター」を対象にオンラインアンケートシステム上で回答を行う事が出来るアンケート。 「ネットモニター」1名につき1回限り回答可能で、回答後その内容に悪質性がない場合に限り、謝礼が登録された住所に送付される。 質問への回答方法は、単一回答・複数回答・記述回答が存在する。設問数は最大でも30問程度、アンケート毎に対象となる「ネットモニター」の属性を指定する事が可能で、対象にあてはまる「ネットモニター」へは通知が送られそれではじめて回答権が得られる。 ネットモニター画面とは: 「ネットモニター」が使用する画面で、自身のアカウントへログインの上、以下の事が実現できる。 1. ネットモニター登録 2. 登録情報(属性)を修正できる 3. 退会できる 4. アンケートに回答することができる 5. アンケートの回答実績を確認できる 管理者画面とは: アンケート配信者(管理者)が使用する画面、自身のアカウントへログインの上、権限に応じて以下の事が実現できる。 1. アンケートの作成、修正、削除ができる 2. アンケートの集計管理ができる 3. ネットモニターの登録情報の管理ができる 4. 不適切なモニターに対し強制的に退会処理ができる 5. 一定の条件に該当するモニター(アンケート回答回数、属性)による検索、抽出ができる 6. 管理者画面ユーザーの管理が出来る

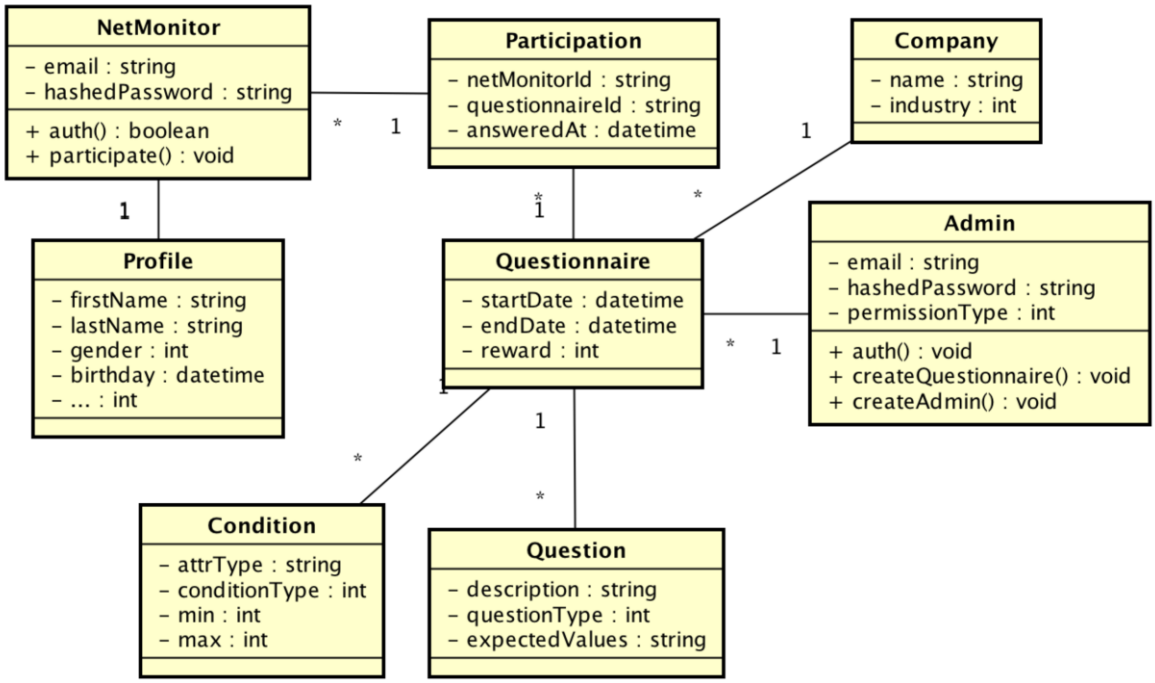
コンテキスト図	
可変性ガイド	将来における機能拡張として、オンラインアンケートだけでなく、モニター情報の管理(参加案件)などを検討出来る形に作成を行う。
アーキテクチャの背景	アーキテクチャドライバとしては、「ユーザーを認証する」、「アクセスを制限する」を選択している。これらは、セキュリティの品質要件を満たす為に必要で、RFP で求められていたセキュリティ要件にも合致する。「アクセスを制限する」では、A 社オフィス内 IP アドレスからのアクセスに限定し、更にクライアント証明書がインストールされたマシンからのみアクセス出来る形を取る。それに加えて「ユーザーを認証する」では、単純な Email/Password の認証のみではなく、権限を３つに分けて、それぞれが適切なデータへ適切なアクセス権を持つことで、よりシームレスな形で、情報漏えいなどのセキュリティ向上に寄与している。今回の分割ビューでは、主に後者の「ユーザーを認証する」をベースとした分割が取られており、それぞれの権限の階層構造が作られている。それぞれの権限毎に機能追加などが行う事が出来、可変性に関してもある程度担保が出来る形がとられている。
その他の情報	2018/02/08　Ver1.0　Initial Draft　作成者:xxxx

11.1 課題１５に対する解答事例と解説



解答例として作成しているモジュール分割ビューにおいては、権限毎に機能を割り振っているが、今回の変更で影響が考慮されるのは、2. の『リクルーター追加』のみ。他は、それぞれのサブモジュールにおける細かなパラメーター設定などで飲み込めるような変更となっていた。

11.2 課題１６に対する解答事例と解説



オンラインアンケートシステムの全容を簡単なクラス図で示した。今回はいわゆる MVC パターンで言う所のモデルに相当する部分の関係性を表したものを例示する。一方、実際にはこの他に、ウェブサービスである事から、コントローラーや、ビューについてもクラス図を検討する必要がある。