要求・要件仕様を効率的、高品質に作成する方法
-経験的設計ルールを継続反映させながら使うモデルベース要件定義フレームワーク-

How to create user requirements and system requirement specifications efficiently and with high quality
- Model-based requirement definition framework to use while reflecting empirical design rules continuously -

藤原 啓一*
Keiichi Fujiwara

システム開発において、ユーザ要求をシステム実現仕様に変換するまでの作業工程（要求・要件定義）は、その後の設計効率を上げるために最も重要であるにもかかわらず、多くの記述モレ・不具合を内在させたままの要件定義書が多く存在する。これが生じる一つの原因として、要件仕様を表現するモデル間の整合ルールを定義しないまま、要求・要件モデル作成を行っていることが考えられる。モデル間の整合性ルールに従って「要求・要件・仕様」開発を行うことがトレーサビリティを確保することである。ここでは、そのようなトレーサビリティを自然と確保できるモデルベース要件定義手順を紹介する。我々は、本手法を適用した実プロジェクトのコスト低減・品質向上への有効性を確認しながら、必要に応じて整合性ルールを新たに定義し、改良整備を繰り返している。

In system development, although the work process (customer requirement／system requirement definition) up to converting the user request to the system realization specification is the most important to improve the design efficiency afterwards, there are many requirement definitions with descriptions defects inherent. As one of the reasons for this, it is considered that requirement／specification models are created without defining the consistency rules between models expressing requirement specifications. It is to ensure traceability by developing “requests／requirements／specifications” according to the consistency rule between models. Here, we introduce the model-based requirement definition procedure that can get such traceability naturally. We confirmed the effectiveness of the actual project applying this method on cost reduction and quality improvement, defined new consistency rules as necessary, and repeatedly improved and improved.

1. はじめに

システム／ソフトウェア要件仕様は、システムの利用者（ユーザと称する）の望む形をシステムの作り手に正確に伝える資料であり、その意図・意向までも正確に伝えることが必要である。ところが、ユーザ要望を思うがまま自然に記述したものや、曖昧な記述ルールに基づいた記述のため、複数通りの解釈ができてしまう資料を要件仕様としているケースが存在する。また、ユーザ要求とシステムから提供してほしいサービスに関する要求が混在して、誰の何に対する要求なのかが曖昧なものもある。システム開発全体のコストを抑えるには、要件仕様段階の曖昧性や仕様モレを、後段の設計や試験において取り去るよりも、要件仕様作成段階において埋め込まないことの方が有効であることはいくつかの研究で明らかになっている。そして、これを実現するには、誰もが、要件仕様作成段階で要求・要件モレや仕様誤り／モレを可能な限り減らせられるような手順を形式化していくことが必要であるが、設計段階ほどには、モデル作成の手順が明らかになっていない、又は適用の浸透がなされていないのが開発現場の現状である。

ここでは、「開発対象ドメインの正確な理解・表現を多視点モデル表現間の整合性により行う」という考えに基づいた、要求・要件の分析・定義方法を紹介する。

2. 要求分析とは何か

要求工学（Requirements Engineering）では、要件、要望、要求、仕様について統一的な見解はない。ここでは、それらを、「抽象度」（図１、表１）と「種類」（表２）の視点で分類記述する。そして、要望／要求／要件

＊関西事業部

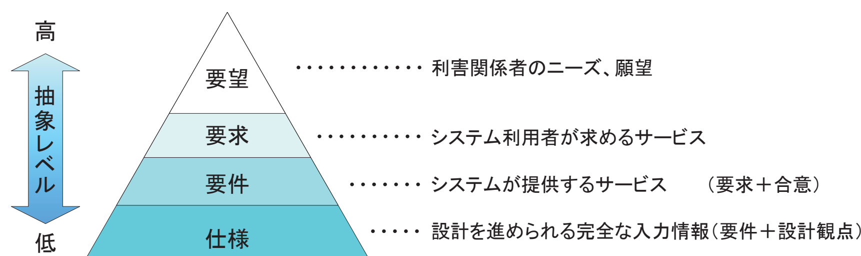


図１　要求の抽象度

表１　要望／要求／要件／仕様の違い

抽象度分類	説明	（例）
要望（Demands）	要望は、利害関係者（ユーザ、顧客）のシステムやサービスへの期待を、そのまま表現したものである。	もっと、■■を効率化させたい。
要求（User Requirements）	要求は、ユーザ、他システム等、自システム以外のシステム利用者が求めているものである（主語はシステム利用者）。要望の背後にある問題や課題を把握し、それを解決する方向、システムの利用環境から見たシステムに実現してほしい目標を表現したものである。将来、設計や実装の変更があっても要求を変更することが無いよう、解決策ではなく必要性記述のレベルに止める。	（顧客は）始動時の●●を目標値±2%以内に安定させたい。
要件（System Requirements）	要件は、要求を自システム（開発しようとしているシステム）視点から見たものであり、システムが提供するユーザにとって価値あるサービスや性能である（主語はシステム）。最終的に要求の中から本当に重要なものをまとめ、「何を実現する必要があるのか」を明らかにし、顧客と開発者で「合意した要求」である。	（システムは）始動時の●●を±2%以内に安定させるために、XXを±0.8%、YYを±1.2%に安定させる。
仕様（Specification）	仕様は、システムと利用環境の接点であり、システムが利用環境に対して、必要とされるサービスや性能を提供するための境界条件（機能の場合、入力と出力）である。仕様はソリューション（手段）を含めた言葉で記述される。また、要件を実現する設計や実装についての仮定や判断事項も記述する。	XXにおける、ZZ処理を、数式AAを使って計算し、入力αに対して出力β以下に抑える。

表２　SQuaREの非機能要件フレーム

特性項目	特性概要	副品質特性	要求内容
性能効率性	システムの実行時の性能や資源効率の度合	時間効率性	システムの応答時間、処理時間等の処理能力の度合
		資源利用性	実行時に使用する資源量や種類
		キャパシティ	要求を満足させるためのシステムのパラメータ最大許容量
互換性	他システムと機能や情報を共有、変換できる度合	共存性	他製品へ負の影響を与えず、共通の環境や資源を共有して機能を効率的に実行する度合
		相互運用性	２つ以上の製品やコンポーネント間で情報を交換、利用できる度合
使用性	効率的、効果的に利用できる度合	適切度認識性	要求（ニーズ）に適した利用かどうか認識できる度合
		習得性	システムの使い方の学習ができる度合
		運用・操作性	運用・管理・監視のしやすさの度合
		ユーザエラー防止性	ユーザが誤操作しないように保護する度合
		UIの快美性	UIが親しみやすく、満足感のある応答ができる度合
		アクセシビリティ	幅広い範囲の心身特性及び能力を持つ人々が利用できる度合
信頼性	必要時に実行することができる度合	成熟性	通常時に信頼性のニーズを満たす度合
		可用性	必要時に運用、接続できる度合
		障害許容性	ハードウェア、ソフトウェア障害に関わらず、意図したように運用操作できる度合
		回復性	障害時にデータやシステムを回復したり、希望状態に復元できる度合
セキュリティ	不正にアクセスがされることなく、情報やデータが保護される度合	機密保持性	認可された者のみがアクセスできるようデータを保証する度合
		インテグリティ	プログラムやデータへの変更において未許可なアクセスを防止する度合
		否認防止性	イベントやアクションの発生を証明する度合
		責任追跡性	エンティティの実行が唯一であることを証明する度合
		真正性	リソースや物事の身元が要求されたものであることを証明する度合
保守性	効率的、効果的に保守や修正ができる度合	モジュール性	変更による他コンポーネントへの影響が最小で済むよう、独立したコンポーネントで構成される度合
		再利用性	他のシステムや資産を利用できる度合
		解析性	変更部分や障害原因の特定のために診断したり、変更による影響を評価する際の効果・効率性の度合
		修正性	欠陥や品質の低下なく修正が効果・効率的にできる度合
		試験性	テスト基準を確立し、評価するために実行する際の効果・効率性の度合
移植性	効率的、効果的にハードウェアや実行環境に移植できる度合	順応性	別のハードウェアやソフトウェア、他の運用環境に効果・効率的に順応できる度合
		設置性	正しくインストール、又はアンインストールする際の効果・効率性の度合
		置換性	同一の目的、環境下で他のソフトウェア製品に置換（リプレース）できる度合

／仕様まで整合性を取りながらつなげることで、要望が仕様として十分かつ正確に記述されていることを保証する。

要求の種別としては、機能要求、非機能（機能の能力）要求、機能・非機能要求に影響を与える外部的な制約条件がある。ISO や JIS には、様々な品質モデルが定められており、非機能要件の洗い出しは、この品質モデルに基づいて行うのが良い。ここでは、ソフトウェア製品の品質要求及び評価として定められている、SQuaRE（System and software Quality Requirements and Evaluation：ISO/IEC 25000 シリーズ）を紹介する（非機能項目のみ）。

2.1　ユーザ要求をシステム要件に変換するとはどういうことか

電気ポットでお湯を沸かす例（参考文献(1)）を考えると、ユーザ要求（お湯を沸かしたい）を達成させるための提供手段を規定するのがシステム要件である。機能とは、システムやソフトウェアが行う仕事の総称である。サービスは機能群の集まりで、ユーザが直接分かる価値をシステムが提供するものである。したがって、“お湯を沸かす”ことは、システムが提供するサービスである。システムは直接的に、ユーザ要求を満足させるようなサービスを提供するが、組込みシステムの場合、ソフトウェアはシステムのハードウェアに働きかけて、間接的にシステムが提供するサービスを駆動・制御する。この違いを十分に理解した上で、システム要件には、ユーザ要求と対応させたシステムが提供するサービスのみを記述し、システム方式設計を経て作成するソフトウェアレベルの要件を記述しないようにする（図２）。

システムが行う働きとソフトウェアが行う働きの違いは表３のとおりであり、これがシステム要件とソフトウェア要件の表現の違いである。

3. 要求の仕様化プロセス

3.1　ユーザ要望・要求を仕様に変換するプロセス

要望・要求・要件は問題領域に属し、仕様はソリューション領域に属する。ユーザ要望を顧客向けに構造化表現することを「ユーザ要求分析」、要求を要件に変換し、顧客と開発者が合意する過程を「システム要件定義」、開発者向けに要件を仕様として表現することを「システム要件の仕様化」と呼ぶ。

表３　システム要件とソフトウェア要件（働き方、提供するサービス）の違い

働くモノ		働き（提供するサービス）
システム	物理的	熱する、冷ます、伸ばす、曲げる、切る、付ける、剝ぐ、支える etc
ソフトウェア	情報	記録する、変更する、複製する、取り出す、計算・比較する etc

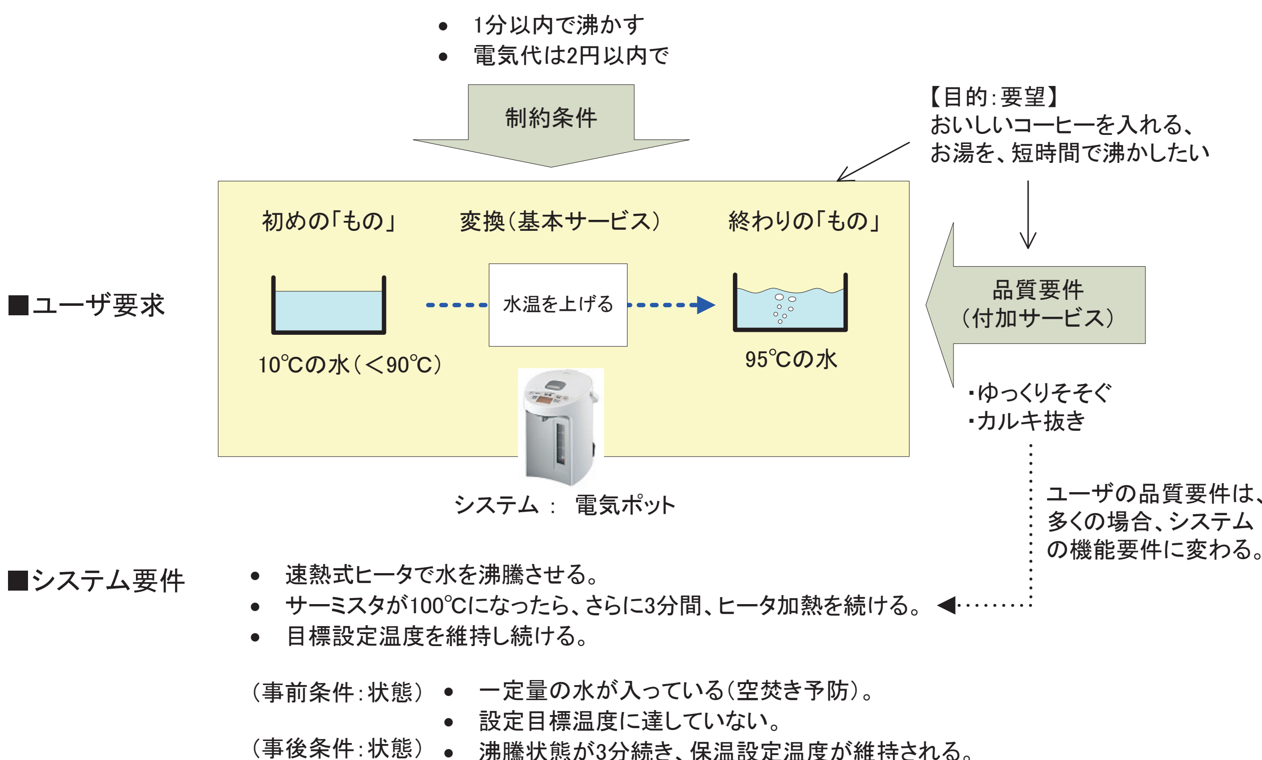


図２　ユーザ要求とシステム要件の違い

「要件定義」段階において、ユーザ要求が、問題領域の専門家の知識に基づいて、システムが提供するサービス（システム要件）と対応付けられる。ここで、ユーザはシステムを利用する人、顧客はシステム開発を依頼する人という意味で使い分けている。

「何をするか」を記述した仕様から、それを「どうしたら実現できるか」を表す設計に変換する過程を「アーキテクチャ（方式）設計」という。要件仕様に“どうする”という知識を付加する過程が「設計」である（図３）。

ユーザ要求を直接満足させるために、汎用計算機上で動作するソフトウェアを使ってユーザに情報提供する業務系システムで、かつ、開発ソフトウェアの規模が大きな場合（10 KL 以上）、複数に分割したサブシステムを統合したシステム全体の要件がシステム要件、個々のサブシステム要件がソフトウェア要件となる。規模の小さな業務系システムの場合は、システム要件・方式設計を省略し、ソフトウェア要件から始めても問題ない。

3.2　ユーザ要望をユーザ要求（概念的戦略や構想）に落とし込むには

ここでの目的は、業務フローやシステムを利用するシーン（利用シーン）を全て抽出することである。ユーザ要望を業務フロー／利用シーンのどちらで表現するかは、そのシステムを利用することで価値を得る利害関係者が自分のみか、自分以外にも存在するかで区別する（表４）。

業務フローも利用シーンも、ユースケースを導き出すための前提づくりである。そのためには、概念モデルを作成しておくことも重要となる。

全く新しいサービスの利用シーンを導き出すには、UX（ユーザエクスペリエンス）手法を使う。

3.3　要望を要求に変換する際の注意

要望は「こんなことを実現してほしい」という期待なので、これだけで、システム全体の構成や機能、性能、制約条件を把握することはできない。いくつかの注意点を理解した上で、ユーザ要求としての取込み選択と変換を行う。

表４　業務フロー、利用シーンの使い分け

表現方法	説明定義	記述単位の粒度
業務フロー	一般的に複数のユーザが関係する作業によって、当該者以外も価値を得る作業の流れ。	当該ユーザ以外の利害関係者が価値を得るまでの当該ユーザの一連の作業。
利用シーン	システム利用者のみが価値を得る作業。例）ツールやカーナビ等。	システム利用の当該ユーザが価値を得る一連の作業。

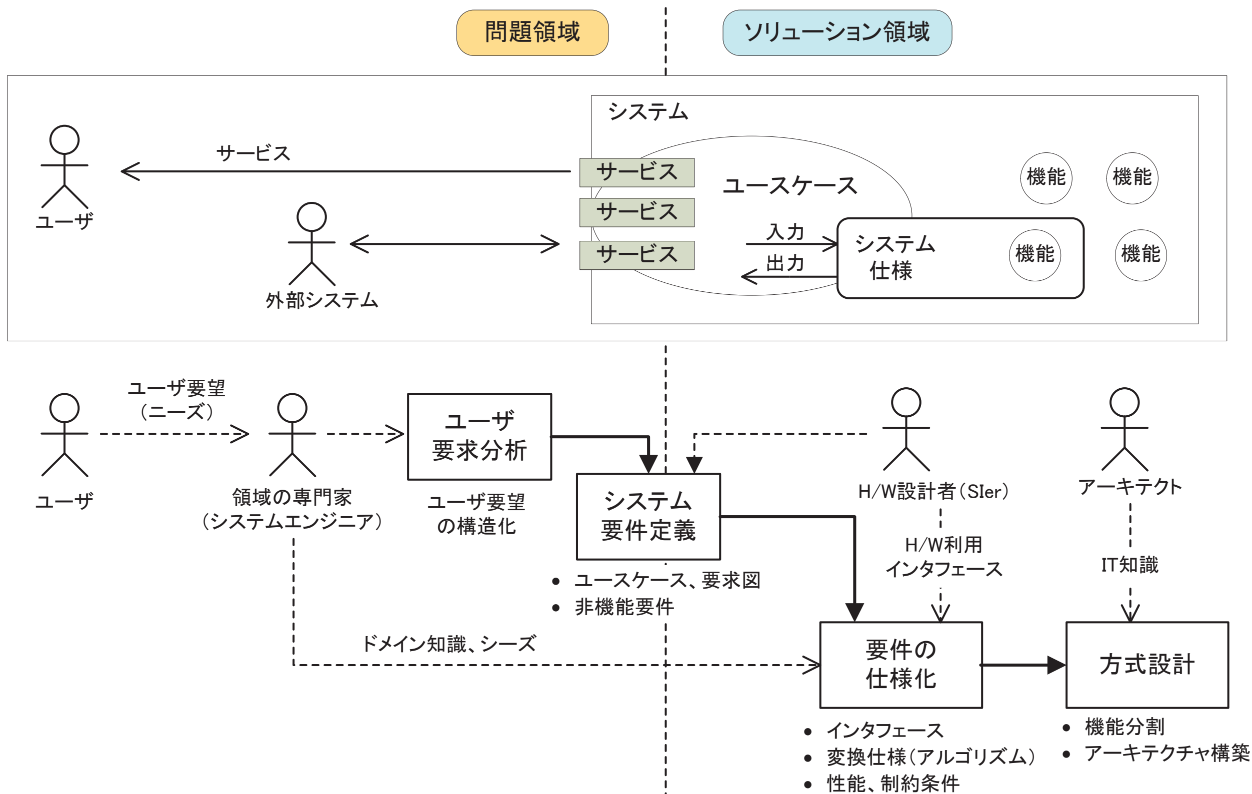


図３　要求獲得プロセスから方式設計への流れ

(1) 要望がユーザや顧客要望の全てではない（必要条件ではあるが、十分条件ではない）。
(2) 要望で表現されている用語の定義は、顧客が属する業界で使われるものであって、他の業界と同じ意味とは限らない。したがって、用語辞書が必要となる。
(3) 要望間には相互矛盾があったり、実現できないものが含まれている。要望と要求、要件仕様間はトレーサビリティを確保し、システム要件段階で、要望の矛盾の解消、実現することのみを残すようにする。実現しない要望は要求との間にトレーサビリティが無くなるので、「削除」したことを明記して、要望としては残しておく。
(4) 要望は、(a)やらなければならないこと、(b)やってはいけないこと、(c)やってもやらなくても良いこと、(d)決まっていないこと（T.B.D）が区別できるようにする。

3.4 ユーザ要求とシステム要件を対応付ける

ユーザ要求とシステム要件は、その表現において、それぞれの主語が、“ユーザ”と“システム”となり、内容表現も異なる（表５）。

ユーザ要求とシステム要件は、対応表を用いて関連付ける（表６）。

ユーザ要求とシステム要件の対応が複雑になる場合は、要求図を用いて、上位にユーザ要求、下位にシステム要件を階層的に表現して対応確認を支援する。

＜アンチパターン＞

(1) ユーザ要求とシステム要件の対応表を作らずに、ユーザ要求を漏らさず、システム要件仕様を作成することは難しい。

3.5 ユーザ要求を仕様に変える流れ

システム要件と仕様の違いを考える。システムを開発する目的は、それを利用することで何らかの作業を自動化したり、支援することである。したがって、システムを利用する主体（ヒトや他システム）は、システムの外部に存在する。利用主体とシステムの関係を考えると、システムを利用することで実現したいコトが要求であり、そのコトを実現するために、具体的にシステムを利用するインタフェース（入力・サービス・出力）がシステム仕様である。そして、要求は、利用環境側から見たシステムが実現すべき目標となる（図４）。

表５ ユーザ要求とシステム要件の表現手法

種別	ユーザ要求表現	システム要件表現
業務系	業務フロー	機能：ユースケース、ユースケースシナリオ＋機能表現 非機能：SQuaRE フレーム
組込み系	ユーザーズマニュアル	

表６ 機能におけるユーザ要求とシステム要件の対応表

ユーザ要求（ユーザストーリー）	システム要件
（私は）忙しい時にすぐにドリップコーヒーを飲みたいので、１分以内にお湯を沸かしたい。	（システムは）１分以内に水を 95℃以上のお湯に変える。
（私は）２杯以上コーヒーを作ることがあるので、沸いたお湯の温度は維持しておきたい。	（システムは）95℃以上の状態を自動的に保持する。

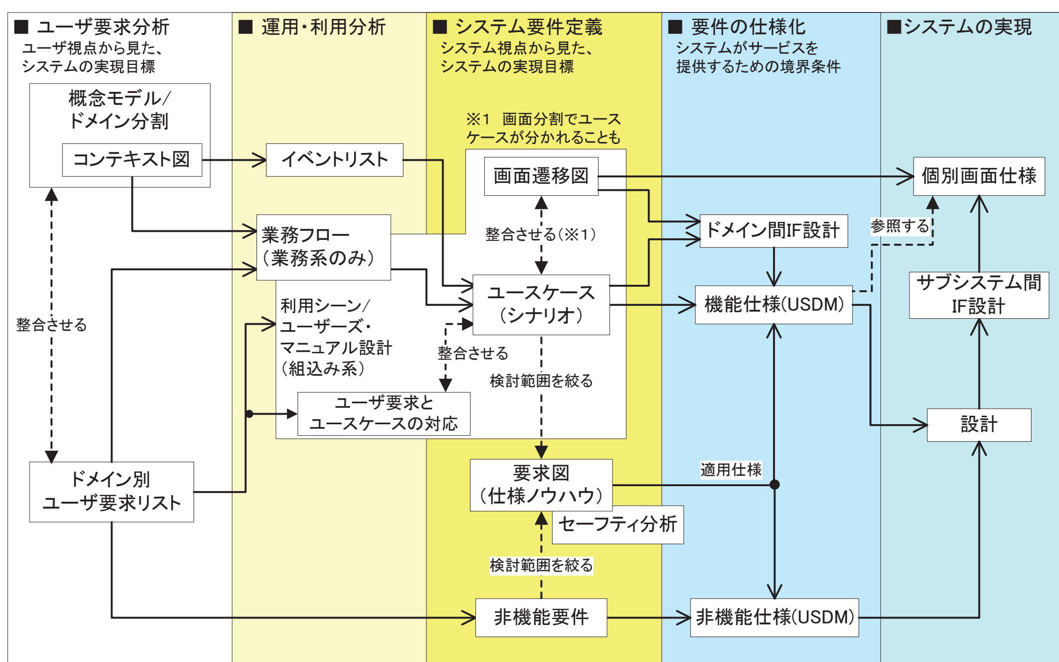


図４ ユーザ要求からシステム実現に至る流れ

3.6　顧客ニーズをいかに理解するか、仕様作成は誰が行う？

要求を仕様に変換するためには、「要求を正確に解釈」し、これから作る「システムが実現するサービスの振舞いと能力を正確に表現」できなければならない。この作業は、要求の背景となっているドメインの知識（その分野の専門知識）を持った人（システムエンジニアと称する）にしか行えない。

仕様を設計に変換する過程は、「What」を「How」に変換する過程であり、実現方式に関する知識を“足し込む”必要がある。この時の足し込み方は、何通りもやり方があるので、課題と矛盾を解決しながら根拠を残す手法で実施する（例：ATAM（Architecture Tradeoff Analysis Method：アーキテクチャトレードオフ分析方法））。

3.7　要件や仕様にHowを含める（従来言われてきたことと異なる点）

IEEEのSyRSガイドラインでは、特定の設計方法論等システム実装に関わる事項を含めないことを推奨しているが、システムエンジニア達が、ある設計方法論を開発制約にしたいのであれば、それを要件（非機能）に含めても構わない。事前検証で特定処理のアルゴリズムが具体的に決められているのであれば、それに沿った形で方式設計以降を考えてもらわなければならないので仕様とした方が良い。

古くは、要件仕様には、システムの実現方法（How）ではなく、システムの目的（What）のみを記述すべきと考えられていた。しかし、新しい技術を取り込む際には、PoC（Proof of Concept：概念検証）により検証を終えていることが通常であり、パターン化されている組織的ノウハウはシーズの適切適用が分かっているはずなので、これらアーキテクチャを前提にした手段としての情報は、システム又はソフトウェアによる解決策として要件仕様に記述した方が効率的である。ただし、機能分割を含め、仕様の実現方法は方式設計段階で考えるのが原則であり、仕様作成段階で、要求に勝手なHowを入れ込まないように、実現性の確からしさが確認できている内容（プロトタイピングで実装を確認、過去の資産ノウハウの流用）に限り、仕様にHowを含めて良いとすべきである。

＜アンチパターン＞

(1)　要件仕様に頑なにHowを記述しない方針を守ると、折角のノウハウが反映されないばかりか、解決策を誤らせてしまいかねない。

3.8　機能要件と非機能要件の関係

「ログイン」を機能とみるか、セキュリティを高める一つの手段として非機能とみるか、どちらで表現しても後工程に大きな影響はない。機能に入れ難いものは、全て非機能と考えて、非機能項目のどこかに記述すれば良い。これは、機能要件と非機能要件は、その記述においては、関係性は薄いものとなるが、方式設計段階で非機能要件はアーキテクチャとして実現され、機能要件は、アーキテクチャを背景としたアプリケーションの振舞いとして実現され、要件仕様段階よりは関係性が深くなるからである。

＜アンチパターン＞

(1)　要件仕様作成段階において、記述しようとする内容が機能なのか非機能なのかを区分することに、相当の時間をかけることに価値はない。

4. 要求・要件・仕様作成の詳細

4.1　コンテキスト図

コンテキスト図は、対象システムが動作する環境・外部要素を明示し、分析／設計者が検討を行う範囲を宣言するための図であり、概念モデルのトップに位置する図である。

対象システムに直接影響を与える実体要素（エンティティ：現実世界に存在するヒト／モノ／コトを具体化したオブジェクト）のみをコンテキスト表現に含めるだけでは、システムサービスを起動するイベントがどのような目的、理由で発生するのかが分からないことがある。したがって、検討対象システムの周辺環境は、イベントの発生源となる外部エンティティを全てコンテキストとして表現する。その上で、以降の分析／設計作業において、設計者が着目し検討を行うべき要素と、そうでない要素の区別が分かるようにする。

コンテキスト図は、SysML仕様で明示的に定義された図ではないので、適切な形式の図を定義して記述する（図５）。

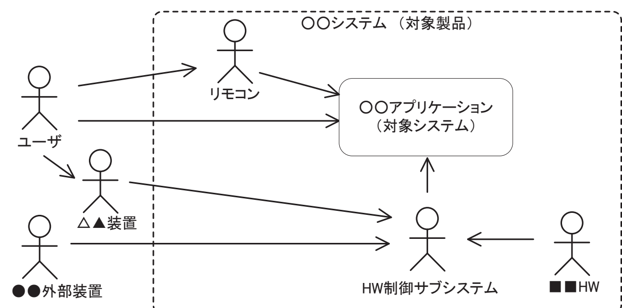


図５　コンテキスト図の例

(1)　システムの中で開発部位を明確にして名前をつける。
(2)　システムの目的を簡単に記述する。
(3)　主アクタ、副アクタを分かる範囲で区別する（表７）。
＜アンチパターン＞
(1)　コンテキスト図作成の目的を忘れ、なんとなく図を作成しない（その図は、後工程で役に立たない）。

4.2　概念モデル（モノ同士の関係を分析する）

概念とはモノに共通した特徴をある基準でまとめた定義である。概念モデルを作成する目的は、システム開発の最初に、これから作成するシステムの全体像を把握することである。

初期の概念モデルは、ユースケースに先だって、具体的なモノとして記述する。初期概念モデルの振舞いには、ユースケースレベルのサービス名称を記述する。

概念モデルは、「何を」にあたる部分と「どうするか」という部分に分ける。「何を」にあたる部分は、構造モデル（静的モデル、ドメインモデル）と呼ぶ。対して、「どうするか」に当たる部分は、振舞いモデル（動的モデル：アクション図、状態遷移、決定表等）と呼ぶ。概念モデルでは、ユーザに見える部分のみならず、その裏にある制御（仕組み）も表現するので、組込み系では、振舞いモデルの方が重要になることが多い。ここでは簡単な電気ポットの概念モデルを図６に示す。

システム要件仕様レベルの振舞いモデルは、主として、状態遷移図で記述する（後述、4.9 節参照）。

概念モデルの作成手順としては、まず、概念データモデルだけを作ってモノ同士の関係性を理解しつつ、ユースケース作成後に、振舞いモデルを“正確に”作成するのが良い。

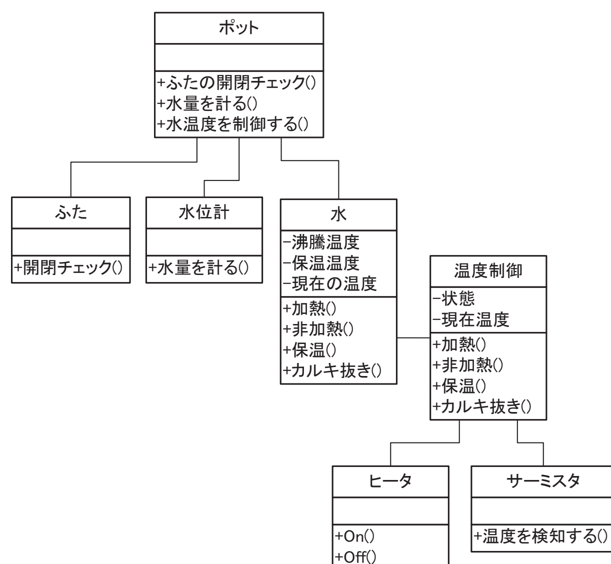


図６　電気ポットの概念構造モデル

表７　主アクタと副アクタの定義

主アクタ	システムと相互作用を行い、システムから何らかの価値を享受するオブジェクト。主としてシステムの利用者だが、接続する外部システムの場合もある。
副アクタ	主アクタからの動作や命令によりシステムを駆動するイベントを発生させるデバイス（ボタン、リモコン、シフトレバー等）。又は、自らのイベントでシステムを駆動するデバイス（タイマ、地震発生や渋滞情報通知の発生源等）。

＜アンチパターン＞
(1)　システムの仕組みや動作をよく理解せず、表面的な名詞句や実体の関係性をつないだだけのエンティティモデルを概念モデルとしてはならない（後工程で役に立たないばかりか、誤りを誘導する）。

4.3　用語辞書（用語集）

用語の混乱状態は、ユーザ要求を正しく理解する上でも、システムを開発する同士にとっても大きな障害である。全く違うと思っていた用語が同じことを意味していたり、逆に、同じ用語で違う内容であったりすると、設計の見直しが発生する。この対策として、システム開発書に用いる用語を集めた「用語辞書」を作成し、そこに定義されたものだけを使ってシステム開発を進める。用語辞書は、表８及び表９のような形式でまとめる。

用語間の関係を木構造で表現した時、上位にある枝別れした葉がドメイン分割のドメイン候補である。また、定義された用語だけを使って設計書が表現されているかどうかの確認に、チェックツールを使うことができる。

表８　用語辞書の内容

項目	説明	表現
基本用語	他の用語から派生するものや計算されるものではない基本的な用語。	名称／意味／属性／単位／取り得る範囲。
用語間の関係性	用語間の関係性を木構造で表現する。	カテゴリ（分類）、全体-部分関係（用語間の関係性は表９の記法を用いる）。

表９　用語間の関係性を表現する記法

記号	意味	説明
＝ // ～	～である	＝の左辺は定義される対象、右辺は定義の意味を示す。
a = x+y+z	～に加えて	a は、x と y と z を常に要素として持つ。
a = [x|y|z]	～１つを選択する	a は、排他的に x 又は y 又は z のいずれかである。
a = n{x}m	繰り返し	a は、x の n 回以上、m 回以下の繰り返しである。
a = @k+p+q	主キー	a は、k をキーとして p と q を要素に持つテーブルである。
( )	オプション	( ) 内はあっても無くても良い。

＜アンチパターン＞

(1) 早い段階から用語辞書を作って使用用語ぶれを制御するプロセスを行わず、各設計段階のレビューにおいて、不適切な使い方を制御しようとしてはならない（設計手順管理者不在）。

(2) 用語辞書の中に、アクションや変換、手順を表現してはならない（用語辞書に仕様は表現しない）。

4.4 ドメイン分割

システム全体の概念モデルの意味の変化点を基準に、ドメイン（domain：領域、空間）を定義する。ドメインとは、特徴的な規則と方針に従って振舞う概念エンティティによって形成される、自律した、現実的、仮想的、抽象的な領域である。

システム開発においてドメインという概念を使う目的は、「システムを構築する」という問題をドメインで分割し、以降、ドメインという区切りで並行開発していくことを可能とするためである。最終製品は、分割した問題領域を結合し、全体的に整合性のとれたシステムとして仕上げる。ここでのドメインモデルは、4.2 節の概念モデルのことである（図７）。

ドメインは、ある程度大きな規模の開発に適用するものなので、ここでは、電気ポットではなく、AV カーナビゲーションを例に、ドメイン構成（ドメインチャート）を示す。これは、方式設計段階で、例えば、レイヤ構造に変換され、さらに各レイヤは、サブシステムに分解される（図８）。

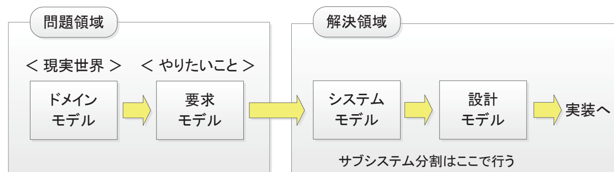


図７　モデル変換の概要図

ドメイン境界は最終製品の境界や設計途中で分かってくるサブシステムの境界とは必ずしも一致しないので、それらの境界と合致させることを狙って設計を進めることに余り意味は無い。最初のドメイン分割は、システム設計で最初に行う問題分割作業であり、以降の作業に対応するチームの能力をマッチさせるのに、システム内の意味の境界で分割した境界を一つの基準として使うのだと考えた方が良い。

4.5 イベントモデル、イベントリスト

仕様として知りたい最初のことは、「システムのサービスがどのスティミュラスで始まるのか」ということである。スティミュラスとは、発生したイベント（事象）によって発生する、システムに刺激を与えるための情報入力である。サービスは、システム外部・内部で発生する様々なイベントから届くスティミュラスにより開始される。サービスに関係するイベントとスティミュラスを明らかにすることがイベント分析であり、分析結果はイベントリストとして表現する（図９）。

イベントはシステムの外部・内部で区別する。外部イベントはシステム利用シーンに関わらず発生し、スティミュラスがシステムに入ってくるので、全てのシステム利用シーンにおいて、その影響を考慮する必要があり、外部イベントだけを、まとめた方が、検討網羅がしやすいからである。

外部イベントが、内部イベントに変わり「ユースケース」

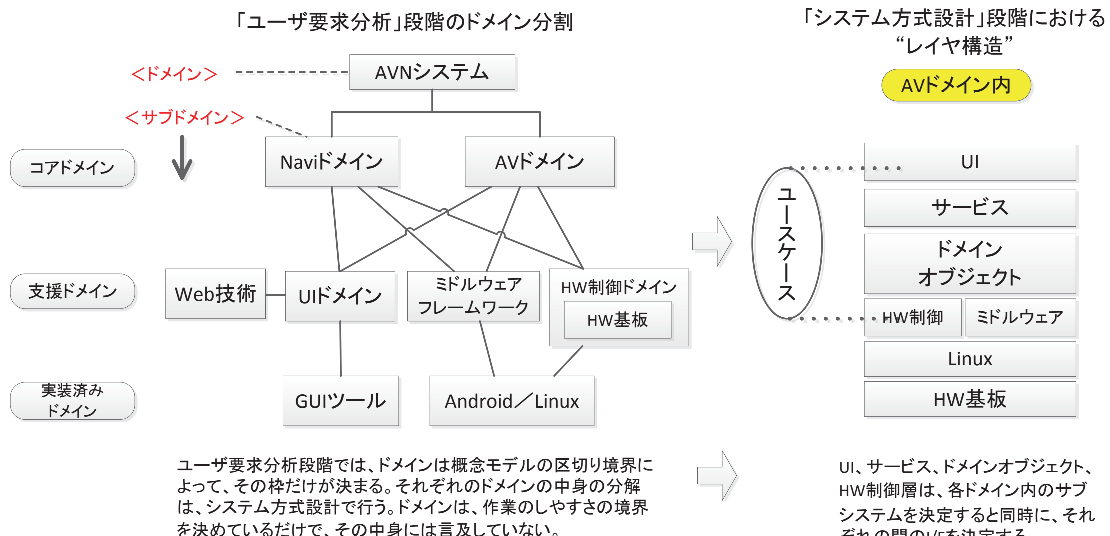


図８　AVN のドメイン構成と方式設計におけるレイヤ構造

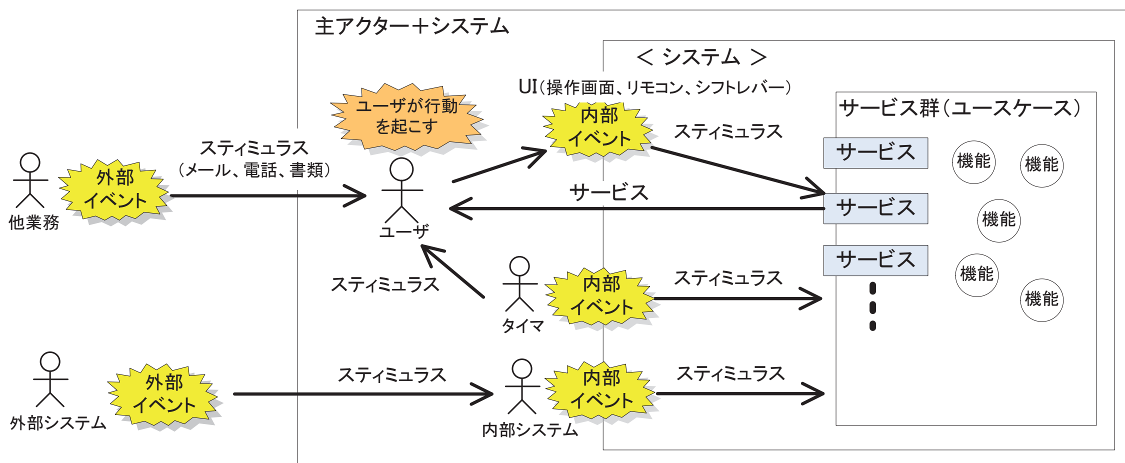


図９　外部イベント、内部イベント、サービスの関係

表 10　イベントリスト例

主アクタ（ユーザ）要求	主アクタへのトリガ		システムへのスティミュラス			イベント発生頻度			レスポンス
	イベント名（起動元）	スティミュラス	イベント名	主アクタ→副アクタ	副アクタ→システム	定周期	非定周期	発生頻度	
ドライバが、サンルーフをマニュアル作動で好きなところまで閉じたい。	（なし）			ドライバは[操作SW]のクローズSWを押し続ける	<閉方向マニュアル作動イベント>[操作SW]がシステムに対して、クローズSW信号＝ONを送る。		○	1回/秒	閉方向にモータを駆動開始
				ドライバは[操作SW]のクローズSWを離す	<作動停止イベント>[操作SW]がシステムに対して、クローズSW信号＝OFFを送る。				モータを駆動停止
お湯の残量を知りたい	（なし）		[センサ]水位変化	−	水位		○		水量
危険な温度状態を回避したい。	（なし）		[サーミスタ]温度異常	−	水温		○		水温、警告情報
支払いを行う	時間	各週金曜日の午後		支払い情報の入力［●●画面］	同左	○		30件/4hr	支払い予約完了
顧客クレーム対処を行う	顧客担当窓口からの依頼	依頼通知（システム）		対策情報［△△画面］	同左		○	20件/日	処理完了

に作用する様を、表 10 を使って表現する（イベントリストと称する）。イベント発生源から要求を考えることで、イベントに端を発する要求モデルをイメージしやすくなり、要求及び仕様のモレを少なくすることができる。

組込みシステムは、タイミングや外部システムからの影響といった、システム動作を変化させる要因が多く存在する。こうした要因に対応するため、組込みシステム開発における分析では、より高いシナリオの網羅性が求められるため、イベントとユースケース（後述）の対応表が必要となる。

＜アンチパターン＞

(1)　イベントリストは要求と価値を中心に考え、インタフェースの実現手段等は記述しない（方式設計以降の後工程で検討する）。ただし、外部システムとのやりとりになるので、組込みシステムのように、既に、物理的な刺激が分かっている場合は、それが分かる記述にする。

4.6　システム要件はユースケース作成から始める

ユースケースは、システム視点で考えた「システムが提供するサービス」（要件）を表す。一つのユースケースの中で、連続して実行する一連の作業の流れがユースケースシナリオである。

ユースケースは、ユーザに提供する価値を中心に考えた機能要求を把握するのに、系統的で分かりやすい手段を提供する。ユースケースモデルにおけるユースケースシナリオは、ユーザが望むシステム利用方法全てを定義した完全な機能要件群と考えることができる。したがって、ユースケースを骨格として、システム要件を抽出していくことで網羅性を高めることができる。

4.7　ユースケースの見つけ方

システムは業務の中で使われるので、業務フローがあれば、その中のアクティビティがユースケース候補となる（アクティビティとシステム境界を表すユースケースが結びつくはずだから）。

“業務”という概念がないシステムにおいては、ユーザが価値あると感じる利用シーン（ユーザ視点から見たシステムの使い方（≠操作）表現）が、各ユースケースとなる。一方、ユーザができることを表現する「ユーザーズマニュアル」の目次は全サービスの骨格を表現したものなので、その目次を想定作成してみれば、自然とユーザに価値を提供するユースケースを見つけることができる（表11）。

＜アンチパターン＞

(1) ユースケースを、上位要求（例：ユーザ要求）のみから発見しようとし、そこに留まって発見作業に時間をかけてはならない（下位設計に進んで、そこからフィードバックを返した方が、ユースケースの発見や分割・統合が容易になることもある）。

4.8 ユースケースシナリオの代替フローの見つけ方

ユースケースシナリオの基本フローは、そのユースケースにおいて確率高く発生するユーザとシステムの相互作用の流れであり、代替フローは、基本フローよりも確率低く発生する準正常に相当する。代替フローは、利用シーンに影響を及ぼす、外部・内部イベントを全て列挙して抽出する。

＜アンチパターン＞

(1) 代替フローに、エラー処理レベルを記述してはならない。

4.9 システムの状態図（ステートマシン図）

オブジェクト指向では、状態図は１つのオブジェクトの状態に着目して振舞いを記述する。システムの状態図は、システム全体や、ユースケース間にまたがった共通のエンティティをオブジェクトとみなし、それに関係する外部イベント（とスティミュラス）に基づいて作成する。シーケンス図が複数要素間の相互作用、振舞いを表現するのに対して、状態図はある単独の要素に着目して振舞いを表現する図なので、その単独要素の境界を明確にすることが重要である（図10）。

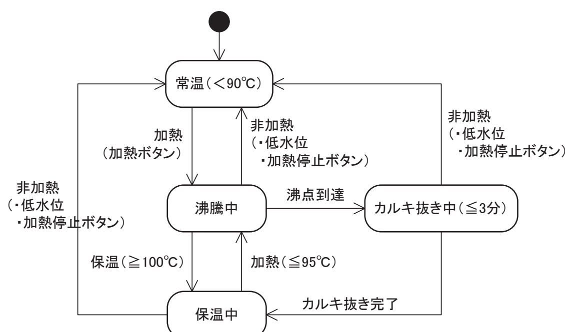


図10 水（エンティティ）の状態図

表11 ユーザーズマニュアル作成の題材表現

作成表現題材	内容
ユースケース	ユーザが受け取るサービス
UI（ユーザインタフェース）遷移	サービス利用に至るまでの画面遷移等
UI操作	サービス利用を目的としたユーザが行う操作

ただし、全ての概念モデル要素に対して、状態図を作成する必要はない。システム設計段階では、ユースケースシナリオ間の複雑な状態遷移を表現するのに使う等、他に適切な表現方法（例：決定表）がないか、その必要性を確認しながら利用するのが良い。

＜アンチパターン＞

(1) 構造化設計時代のとおり、状態とオブジェクトの対応が曖昧なまま状態遷移設計を行ってはならない。

(2) 真に状態定義が必要なものだけを選別し、イベントに対応するものを、全て状態としてはならない。
例）イベントに対応して起動されるサービスが常に同じであれば、そこに状態は定義されない。

4.10 要件の詳細化に要求図を使う

3.7節でも述べたが、既にノウハウ（シーズ：利用目的なしに発見された事実）があり、その組み合わせで要件を満足させられる場合は、それを事前に明らかにする方が効率的に仕様を導きだせる。また、性能を満足させるための方法がシステムの肝で、アルゴリズムをサブアルゴリズムの塊に分解し、それぞれのサブアルゴリズムが満たす性能を割り振りたい場合がある。このような場合、ユースケースシナリオでは表現し難いので、「(SysMLの）要求図」を使ってニーズとシーズの連携をツリーとして図示する（要件図と称する）。要件図の表現にはゴール指向形式を使い、目的（要件）／手段（仕様）の関係性をはっきりさせて作成する。さらに制御系の場合、ハードウェアをどう動かすのかが重要になってくるので、手段（仕様）とハードウェアの関係性も分かるようにする（図11）。

4.11 要件図作成のコツ

要件の階層化は、ユーザ目的視点と、システムサービス詳細化視点に分けて表現する。ユーザ目的視点表現は、ユーザが行うプロセスとシステム提供のサービスを対応させて表したものであり、第２階層にはユースケースを配置表現する。システムサービス詳細化視点表現は、トップ階層のユースケースを実現する手段を階層的に示したものであり、その一つ一つの手段には性能も含まれる。また、組込み系の場合、制御対象ハードウェア（HW）との関係も表現する。ユーザ目的視点表現は

システムサービス詳細化視点よりも上位に位置するものである（図12）。

要件図は、それだけで、ユーザ要求から仕様までの全てを表現しようとせず、(1)複雑になったユーザ要求の分解の流れを示すインデックスとして使ったり、(2)ユースケース実現に関係するシーズが複数存在する中から、

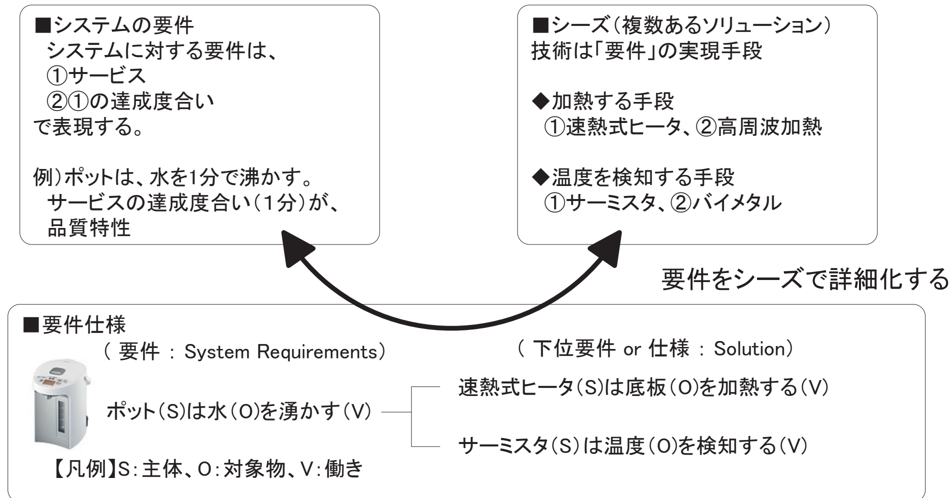
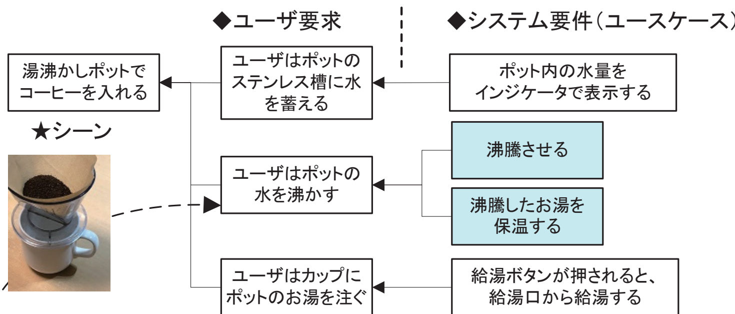


図11　シーズで要件の詳細化を行う



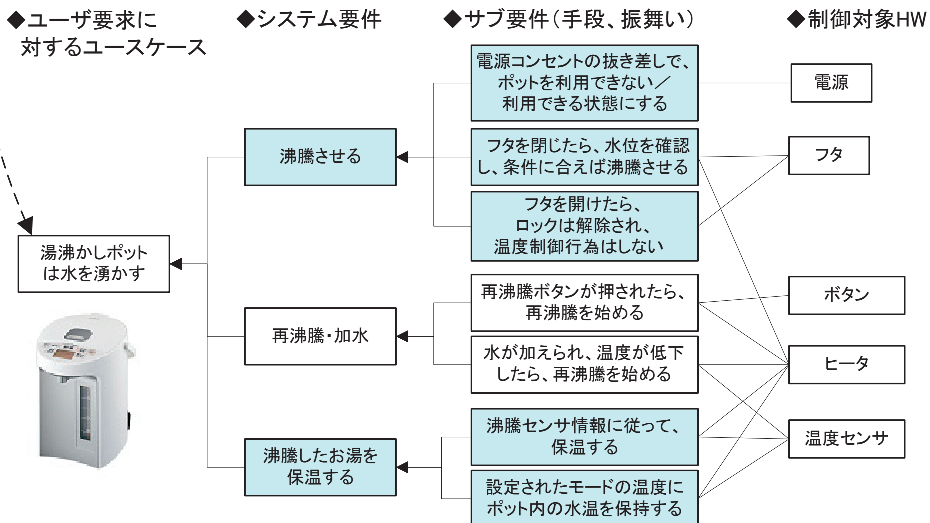


図12　要件・仕様の部分解を SysML の要求図で表す

何を選択したかを表現するノウハウ表現として使うのが良い。

ユースケースをキーにして、要件図と要件仕様（USDM）を照らし合わせることで、要件仕様の構成が分かりやすくなり、要件仕様の網羅の度合を判断することができるようになる。

＜アンチパターン＞

(1) 要件図だけで、全ての要求／要件／仕様を表現し、開発者への入力情報とすることはできない（開発する側からの視点だけでは情報不足）。

(2) 要件図作成にルールを設けず、思いつくまま記述してはならない（人依存、技術レビュー困難）。

4.12　要件仕様の本体（USDM 表現）

機能仕様、非機能仕様は、USDM（Universal Specification Describing Manner）を使って記述する。USDM の形式を図 13 に示す。USDM の特徴は、(1)要件と仕様を分離する様式となっていること、(2)要件と複数の仕様の対応が分かりやすくできること、である。ただし、(2)を実現するには、仕様をグルーピングする要件の表現ルールが必要である。これは、5.2 節に詳細を示す。

4.13　仕様検討過程の残し方

要件図を使用しても、要件や仕様を全て正確に記述できない。要件に対応する仕様のヌケ・モレがないことは、USDM 形式上の仕様と連携した仕様根拠資料の内容を見て確認する（図 14）。

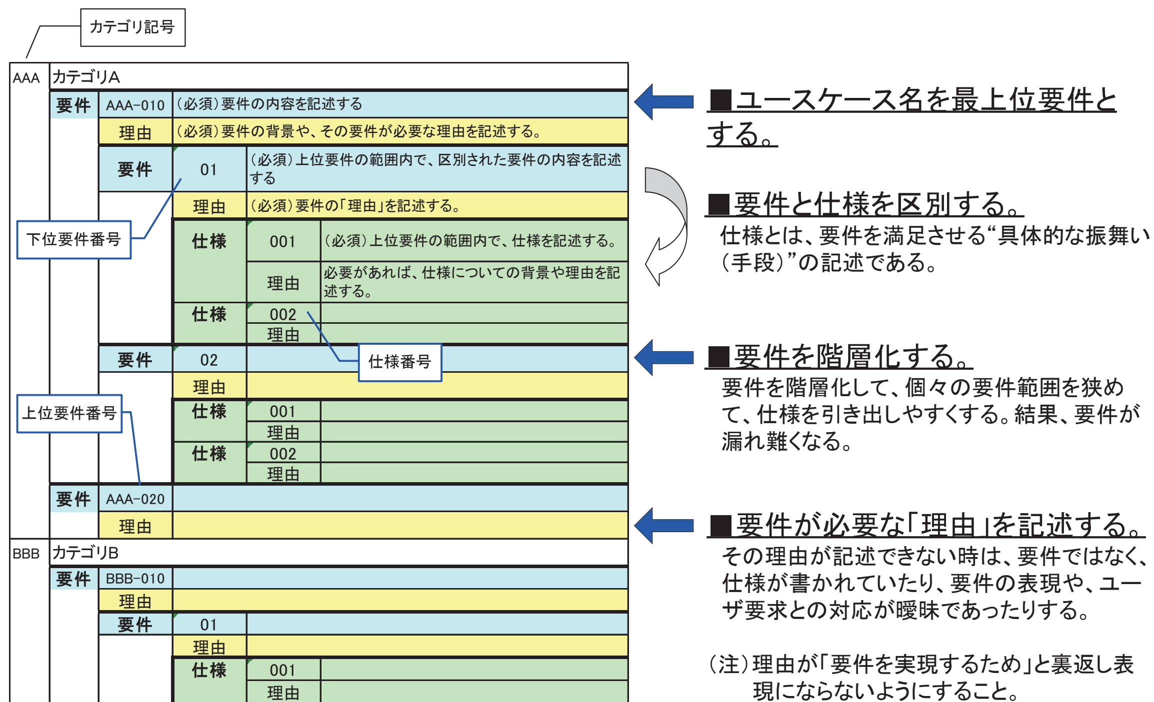


図 13　USDM の形式

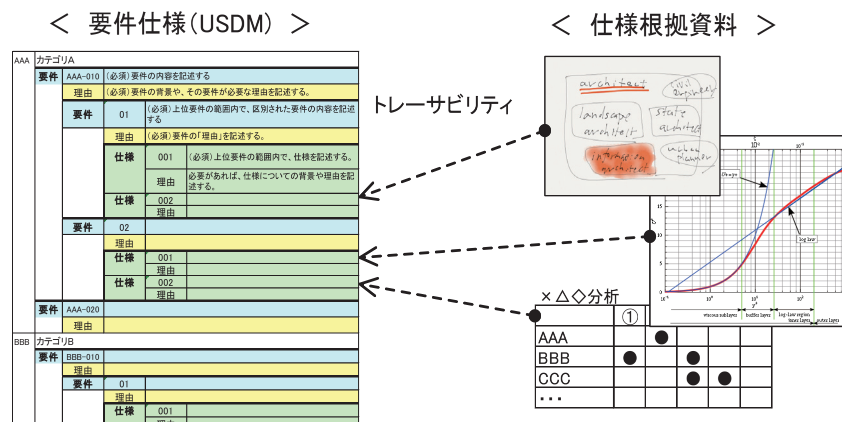


図 14　仕様と仕様根拠資料をつなぐ

仕様と仕様根拠資料（ノウハウ）をつなぐ（トレーサビリティを取る）ことにより、形式知（仕様）と暗黙知（仕様根拠資料）をつなぐ。仕様根拠資料は、なぜそのような仕様としたのか理由を説明する資料であり、メモ的なものもあれば、厳密なシミュレーション結果もある。

要件・仕様数と仕様根拠資料の対応付け度合（例：90％以上）を計測・管理すれば、ノウハウが机に埋もれることも少なくなる。

4.14 システムの異常系設計

想定外も含めて開発時に作り込まれてしまった欠陥（defect）があると、稼働中に障害（fault）が顕在化し、故障（failure）へと連鎖する。障害が発生した後に生じた機能達成能力の喪失状態が故障である。障害は瞬間的なイベント（事象）であり、故障は持続的な状態である。

障害が発生しても故障に至らず、運用を続けるための設計は、非機能要件の可用性レベル方針を決定することから始める。非機能要件仕様作成段階において、障害の発生を早期に認識し、その障害の原因を特定し回復させるための機能（フェールセーフ）を実装するように指定したり、運用オペレーションミス等を起こしにくくする対策（フールプルーフ）を指示する。

正常／非正常の状態を考える場合、当該ユースケースの事前条件／事後条件が成立する／しないという観点で障害・故障後の復旧処理を考える（表 12）。

具体的な欠陥・障害に対応する機能をどうするかは、事前条件／事後条件が成立しなかった状態、全てに対して検討する。

想定外を取り込む安全解析の検討は、次の手順で実施する。

(1) HAZOP でハザードを見つける（故障や障害が起きる可能性のイメージを膨らませる）。
(2) FTA でハザードに至る経路を対策する。
(3) FMEA で対策の正しさを確認する。

図 10「水（エンティティ）の状態図」における「状態遷移：沸騰中→保温」の「事象：温度 T ≧ 100℃」を例に、HAZOP 解析を表 13 に示す。

FTA／FMEA の詳細は、参考文献(2)(3)を参照のこと。

システムとしての対策設計方針は、次のとおりである。

(1) 障害が起きても、可能な限り故障に至らないような設計を行う（準正常設計）。
(2) 想定外も含めて、故障に至った場合、それを自動／手動で復旧できるように設計する（異常系、安全系設計）。
(3) FTA 解析は、システムの構成要素を明らかにした後に、正確に行えるようになるのであるが、要件仕様段階では、ユースケース粒度の制約条件（事前・事後条件、状態図等）から分かる内容で異常設計方針を決める。

表 12　正常系／非正常系（準正常、異常）の定義

区分		定義	事前条件／事後条件の成立具合
正常系		期待しているシステムの能力と振舞い	事前条件が成立し、サービス実施後に、事後条件が成立する。
非正常系	準正常	一時的に正常系から外れた状態になるが、いずれ正常系に移行する状態	正常系とは異なるフローであるが、この系の事前条件が成立し、準正常サービス実施後に、この系の事後条件が成立する。
	異常	システム単体では正常系へ移行する可能性がない状態	事前条件が成立しない。
			事前条件は成立するが、事後条件が成立しない。

表 13　「状態遷移：沸騰中→保温」の「事象：温度 T ≧ 100℃」の HAZOP 解析

ガイドワード		解釈	原因と状況	結果と対策
No	否定	事象が未発生	沸騰しても 100℃を超えない	加熱が続く。 低水位が作動すれば空だきは防げる。
As well as	質的変化	事象を誤検出	沸騰を誤検出	沸騰に至らないまま、95℃以下になれば加熱再開。
Part of		不完全な遷移	加熱を止められず、保温状態へ遷移	保温状態で加熱が続く。 低水位が作動すれば空だきは防げる。
Reverse	置換	事象が未伝達	沸騰しても事象が伝わらない	加熱が続く。 低水位が作動すれば空だきは防げる。
Other than		別事象を誤認	低水位を 100℃と誤認	保温状態に移る。その後、加熱状態に戻ってから、低水位が作動しない危険性がある。 対策として低水位を２回検出する。

＜アンチパターン＞

(1) 各サブシステム設計者や、ソフトウェア設計者が、思いつくまま、知っている範囲だけで障害対策を設計してはならない。システム“全体”で体系立てて異常系対策を考えること。
(2) 局所的エラー処理と異常系設計を混在させてはならない。

5. 要件仕様モデル間の関係性

システム設計のトレーサビリティは、それぞれの設計モデル間の関係性を辿り、レビューすることに価値があるもの同士をつなぐ。その関係性が遠く離れている連結、例えば、ユーザ要求とソースコードを直接つなぐことは、なぜ、それらがつながっているかをレビューできない連携であり、設計トレーサビリティとは言えない。

本設計において、モデル間整合にルール定義している関係性を表14に示す。モデル視点の要件仕様間の整合性を確認することで、要件仕様品質を確保することができる。

モデル間整合ルールの一部を以降に示す。

表 14　要求分析段階の様々なモデル表現の間の関係性

設計視点				設計視点間の相関関係														
				①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩	⑪	⑫	⑬		⑭
要件仕様作成		コンテキスト図	①		▲	▲	▲	▲	●	▲		▲						
		ドメイン分析	②			●	●	●		●								
		ユーザ要求	③						▲	▲								
		用語辞書	④					●	●	●	●	●	●	●	●	●		●
		概念モデル	⑤							▲			▲		●			
		イベントリスト	⑥							●	●	▲	●					
		ユースケース図	⑦								●	●	●		▲			
		ユースケースシナリオ	⑧										●	●	●			
		要求図	⑨										●	●	●			
		要件仕様（機能）	⑩											●	●	●		●
		要件仕様（非機能）	⑪												●	●		●
		状態遷移、決定表	⑫															
		仕様検討資料	⑬															
		…																
方式設計	論理構成	ロバストネス分析	⑭															
		…																

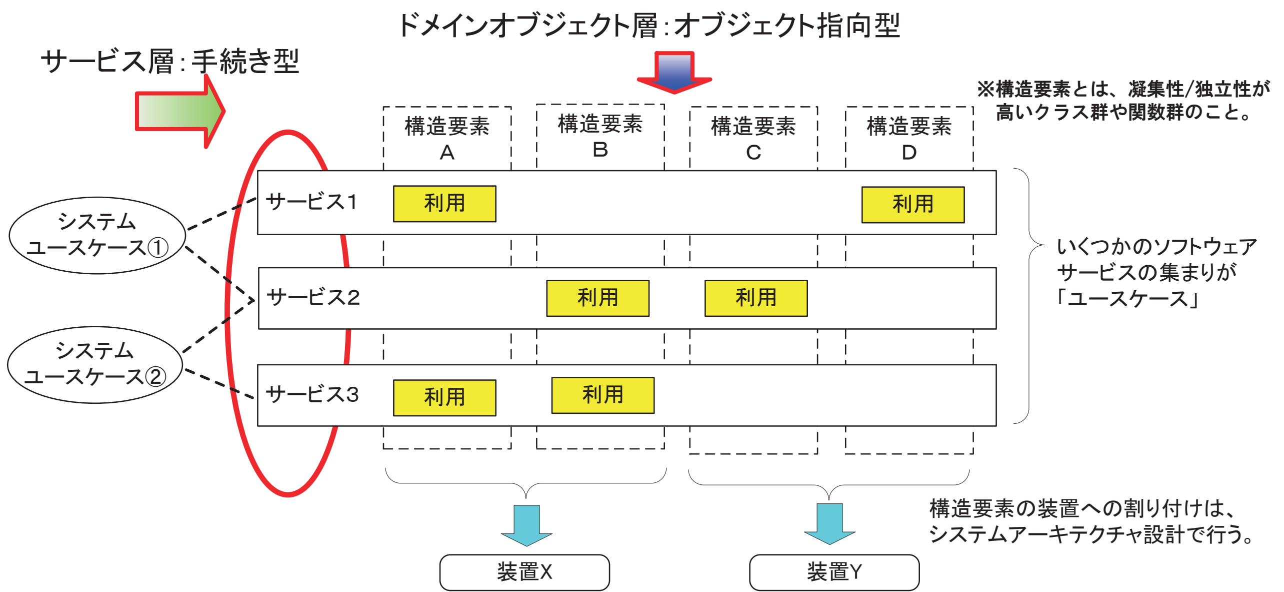


図 15　ユースケースと機能の関係

5.1　ユースケースとドメイン固有機能の関係

ユースケース（＝サービス）とドメイン固有機能はそれぞれ、次のように定義される。

サービス＝Σ（機能）：アクタにとって価値ある振舞い粒度までドメイン固有機能を集めたもの

ドメイン固有機能＝入力から出力への（変換）関数

ユースケースを構成するドメイン固有機能（構造要素）はマトリクスで表現できることから、それぞれは直交関係にあると考えることができる（図 15）。ユースケースから機能を導く手順は、参考文献(4)を参照のこと。

＜アンチパターン＞

(1)　ユースケースを機能ブロックと考えて記述してはならない。また、ユースケースだけからサブシステム分解を行ってはならない。

5.2　ユースケースとシステム要件の関係

ユースケースは、システムの中身をブラックボックスとしてシステムの外側から見える「システムが提供するサービス」（要件）を表したものである。一つのユースケースの中で、連続して実行する一連の作業の流れがユースケースシナリオであり、このユースケースシナリオはユーザが望むシステム利用方法全てを定義した完全な機能要件群である。ユースケースから機能要件を抽出するには、この形を利用する。

USDMは、システム内部の振舞い（例：データ変換仕様等）を表現したものであり、ユースケースとUSDMは、以下の手順によって対応付ける（図16）。

(1) ユースケース名（ユースケース全体）をUSDMの上位要求に対応付ける。
(2) 各ステップをUSDMの第1階層要件に展開する（必要なら第2階層要件も考える）。
(3) 各ステップの実現方法を仕様として展開する。仕様記述には先の要件図表現の手段も利用する。

ステップごとの実現仕様としては、イベントのインタフェース仕様、ステップ実行直後に行うシステムの振舞い、判断、データ変換処理等を考える。

ユースケースシナリオの、それぞれのステップはアクタの要求と、それに対応するシステムが実現する要件の繰り返しを表現している。仕様は、それぞれのステップに対応する作業を実施するために、システムが行うことを記述する。要件をユースケースで表現するメリットは、要件に重なりはあってもヌケ・モレを防止する可能性が高くなることである。

5.3　ユースケースと状態遷移の関係

ユースケースの事前条件／事後条件は、ユースケースが駆動される前、駆動された後の「状態」を表すものであり、「（事後状態）－（事前状態）」で、ユースケース内のリソースの状態変化が分かる。したがって、ユースケース名は、その変化を抽象的に表現するものでなければならない。

また、ユースケースに影響を及ぼすイベントから発生する全てのスティミュラスが、ユースケースシナリオのどこかに表現されていなければならない。

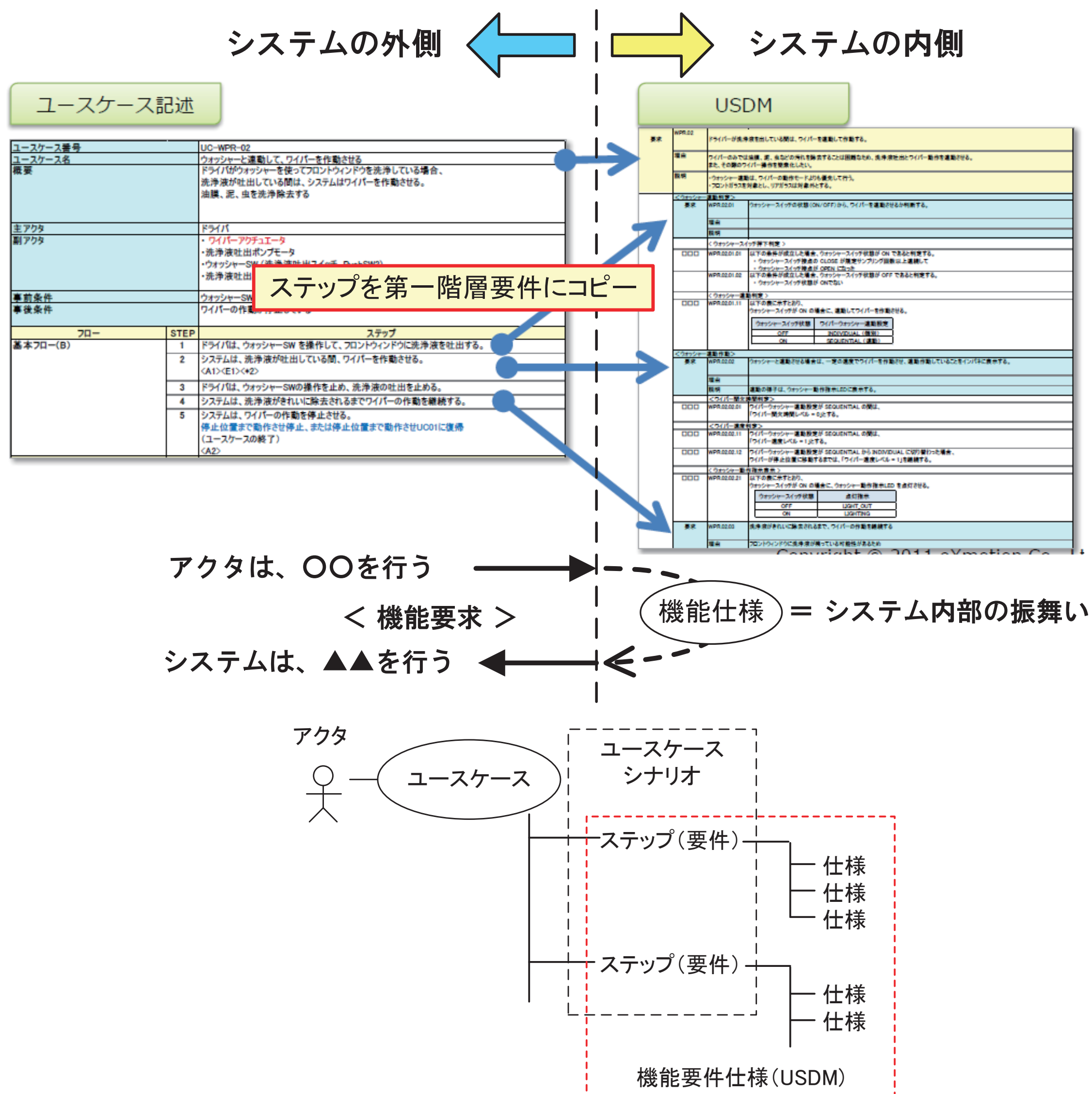


図16　ユースケースとUSDMの関係

5.4　状態遷移とシーケンス図の階層関係

シーケンス図はオブジェクト間の相互作用を表し、状態図は1つのオブジェクトの状態に着目して振舞いを表したものである。したがって、状態図の状態遷移を促すスティミュラスは、シーケンス図の相互作用のどこかに必ず現れていなければならない。この時、現れる相互作用は一つのシーケンス図に留まる保証はなく、対象オブジェクトを中心に関係する複数のシーケンス図との整合性を確認する必要がある（図17）。

6. システム要件をシステム方式設計以降につなぐ

6.1　ユースケースとフィーチャーの関係

フィーチャー（feature）という用語は一貫した定義がない。要件と同義に用いられることもあるが、ここでは、要件を実現するための、もう一段下の機能（ユーザ機能と称する）であると定義する。

要件仕様（USDM）をロバストネス分析した結果のコントロールオブジェクトが、ユーザ機能の候補となる。コントロールオブジェクトは、時に、制御クラスになることがあるので、システムが提供するサービス的内容を持つコントロールオブジェクトだけをユーザ機能とする。

要件仕様からロバストネス分析を行い、全ロバストネス分析図を見て論理サブシステムを定義する作業を図18に示す。

ロバストネス分析や論理サブシステム定義をしてみて初めて見つかる要件記述のモレや矛盾がある。したがって、要件仕様（USDM）とロバストネス分析作業等は密に作業を行い、それぞれ欠陥がなくなるまで、作成・修正サイクルを回す。

6.2　ソフトウェア要件仕様定義（ユースケースの階層関係）

ユースケースを階層化させながら設計詳細化を進めることもできるが、効率面からは、ユースケースは最初の要求・要件を明らかにする際のみに使い、それ以降の下位層はモジュールと考えて設計するのが良い。システム設計レベルのユースケース群から、方式設計を通じてサブシステムが定義される。ソフトウェア要件仕様作成段階では、ソフトウェアユースケースという概念を持ち出さずに、方式設計で定義されたサブシステムを境界として、そのサブシステム内のソフトウェア要件仕様記述にした方が、要件仕様をまとめやすい（図19）。

＜アンチパターン＞

(1)　システム要件仕様とソフトウェア要件仕様のつなげ方に規則なく、勝手なグルーピングでソフトウェア要件仕様を記述すると、ユーザ要求のモレ、矛盾する記述のダブり、誤りを見つけ難くしてしまう。

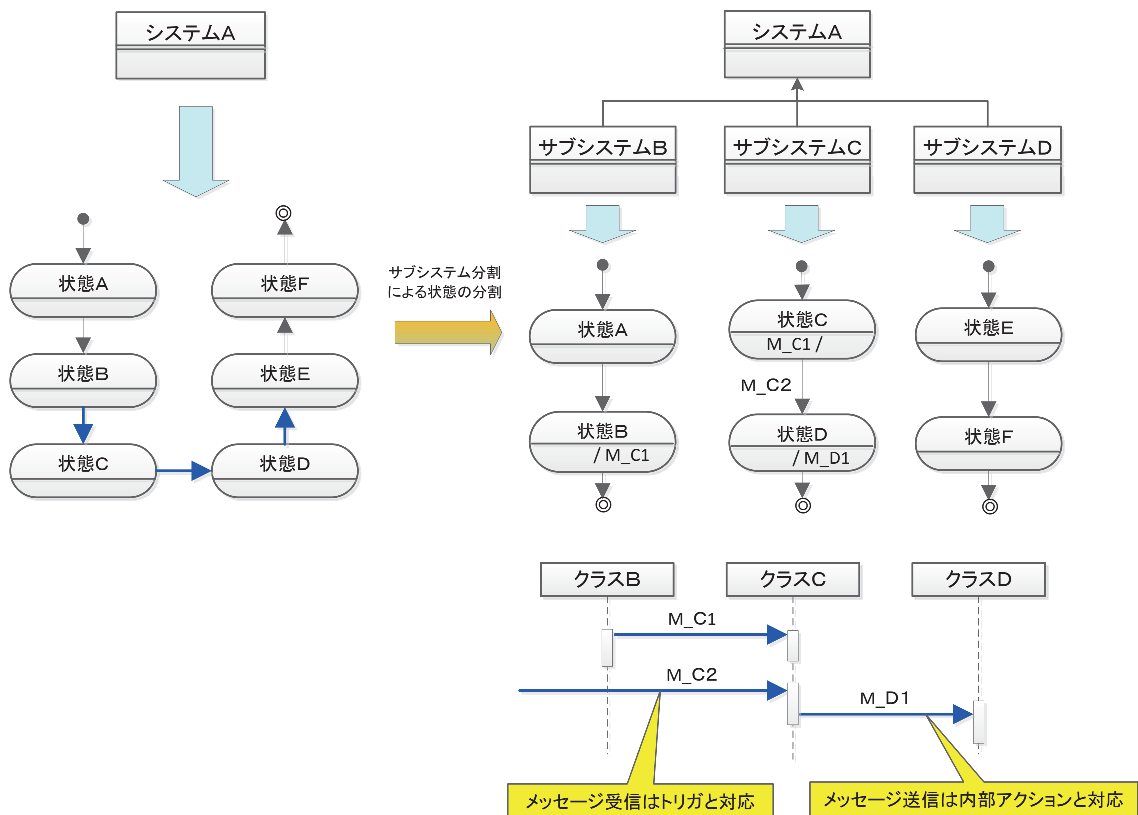


図17　状態遷移とシーケンス図の関係

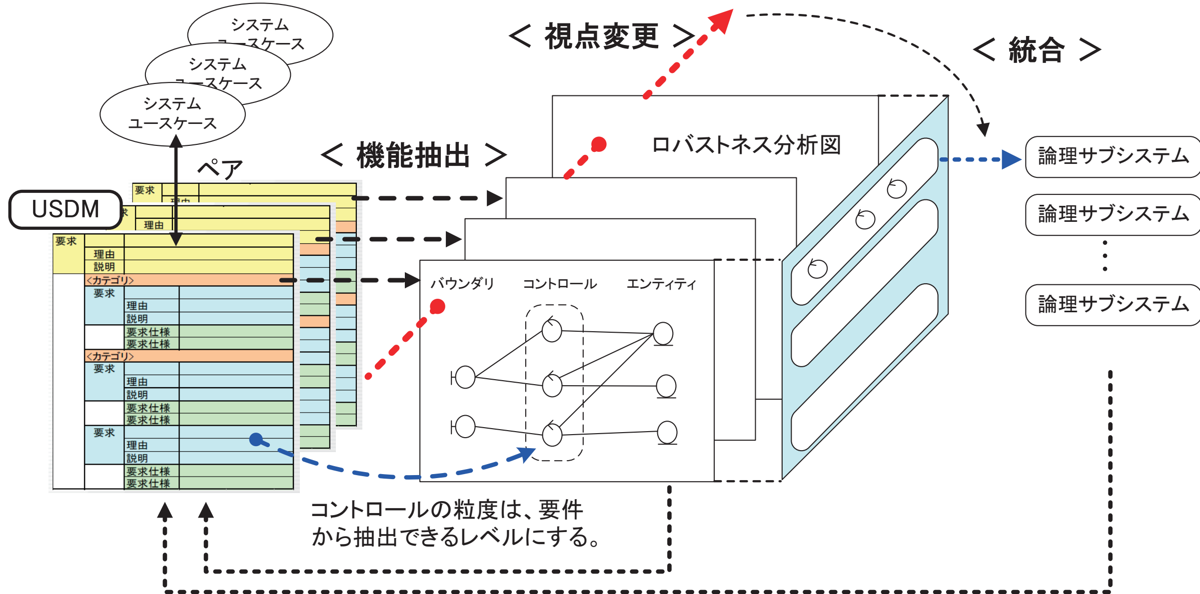


図 18　要件仕様から論理サブシステム定義までの作業手順

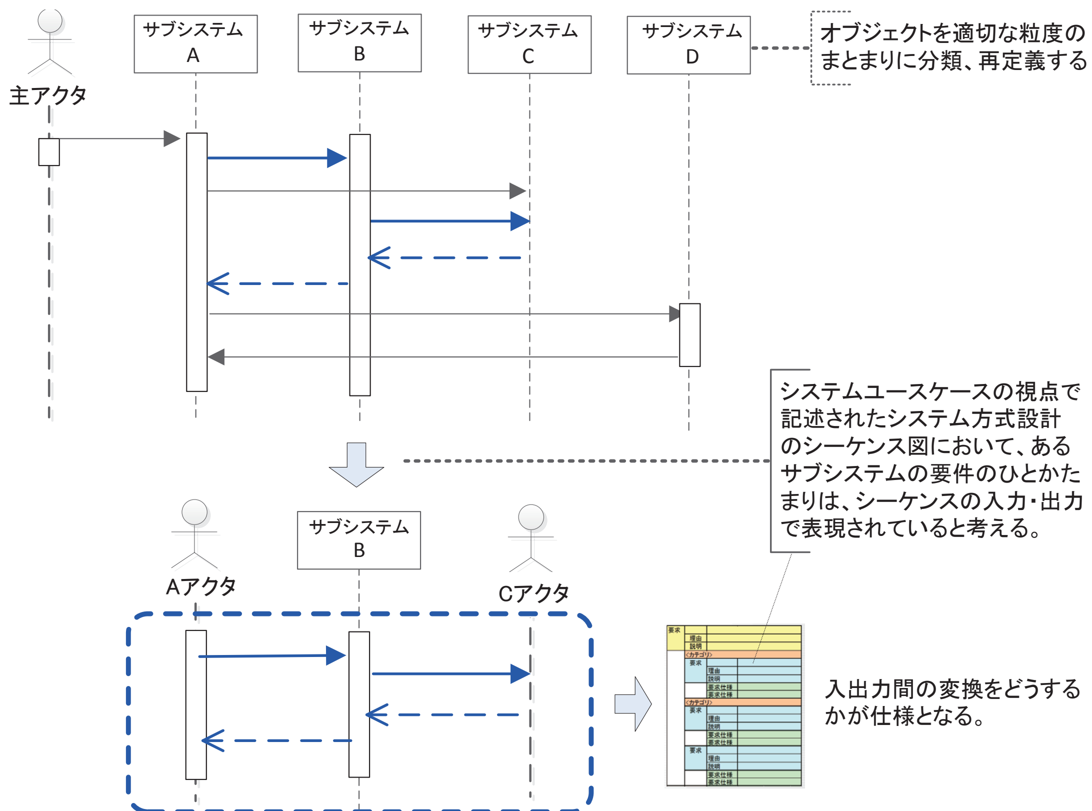


図 19　ソフトウェア要件仕様の記述のグルーピング粒度

7. システム開発プロセス

7.1 システム開発プロセス例

設計モデルの表現方法一つ一つは、対象ドメイン（業務系／組込み系）に関わらず利用できる。ただし、組込み系ではハードウェア設計と効率的に整合性を取るための“すりあわせ手順”が必要であったり、サービス内容が概念的に全く新しい場合、サービス構想段階と製品開発のつなぎ方の検討が必要であったりと、開発対象構成品や開発内容の新規度合によって開発プロセスが異なる。ここでは、ハードウェアとソフトウェアの複合製品に関する構想段階からの開発プロセス例を示す（図 20）。

「事業・サービス・製品構想」から、初期の概念モデル、コンテキスト図、ドメイン図が作られる。

これを受けて、製品開発は設計を進める。ハードウェア、ソフトウェア開発をコンカレントに進めても、その結合時に手戻りが発生しないよう、システム要件は、ハードウェア、ソフトウェアの区別なく記述表現し、設計・製造以降、分かれた段階では、ドメイン間の設計結合を適宜、実施する。

7.2 UI 設計プロセス例

UI（ユーザインタフェース）は、システムが提供するサービスの動きを、ユーザが見る唯一のモノなので、具体的な形にして機能的な内容よりも早く見せることが多い。ただ、開発効率視点から見ると、UI はユーザ意向の変更が最も入りやすいドメインであり、変更を受入れ、かつ、その変更影響が機能に影響を及ぼさないようにするには、アプリケーション・ドメイン部分とは独立性高く開発を行う必要がある。そのための、開発プロセス例を示す（図 21）。これは、UI に先行してサービス部位のアプリケーション機能を明らかにし、UI はユーザ要求とアプリケーション・ドメイン部分をつなぐ役割として、その変更を UI ドメイン部で最大限吸収するように進める開発プロセスである。

UI 仕様の設計は、非機能要件である UI ポリシーを基に、ユースケース分析から分かるサービスと相互に整合性を取りながら進める。

(1) まず、ユースケース分析を行い、ユースケースシナリオを作成する（5.2 節参照）。
(2) 並行して、「UI ポリシー」を基に UI ナビゲーション設計を行い、「UI - アプリドメイン間 IF」と称する、ユースケースとユースケースに関係する主画面の対応表を作成する。
(3) システム要件仕様定義から作成される「機能要件仕様書」を基に、論理設計（ロバストネス分析）を行う。
(4) 「ロバストネス分析図」、及び「UI - アプリドメイン間 IF」と「ユーザ操作イベント」から作られる「UI - アプリドメイン間ブリッジ仕様」を使って、論理サブシステム定義を行い、「UI - アプリサブシステム間 IF 仕様」を作る。
(5) 「UI - アプリサブシステム間 IF 仕様」を基に、UI アクション応答設計を行う。その結果、「個別画面設計書」が作成される。
(6) UI アクション応答設計の際に生じた、画面の分割、遷移の変更は画面ナビゲーションに反映するシステム要件仕様定義へのインプットとする。

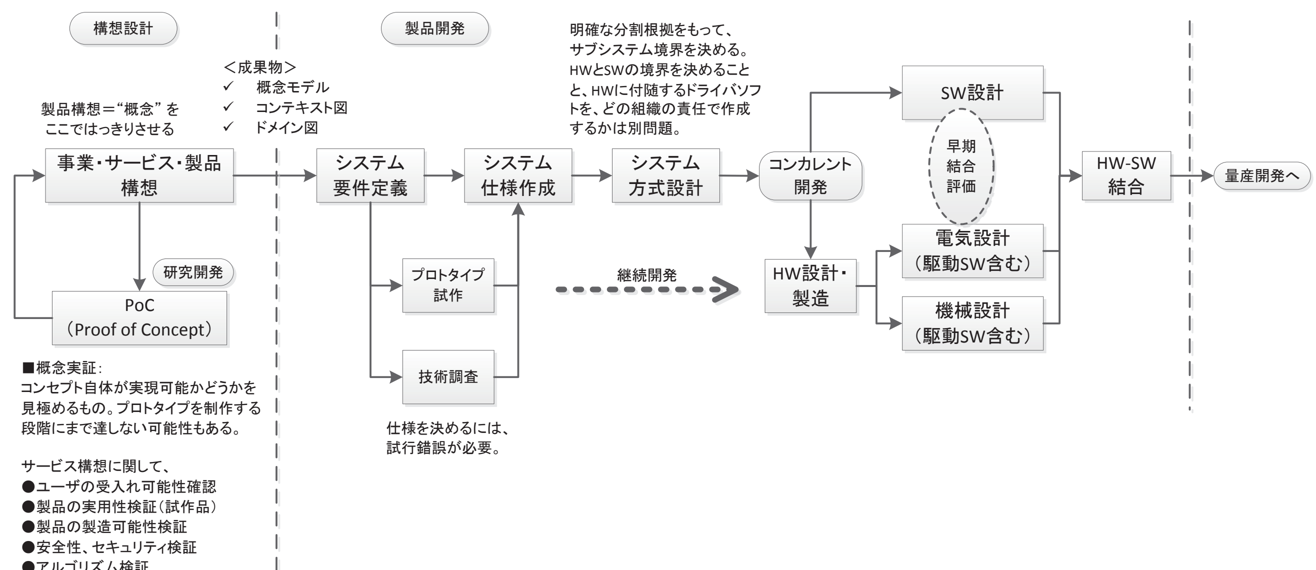


図 20 ハードウェア、ソフトウェア複合製品の新規開発プロセス

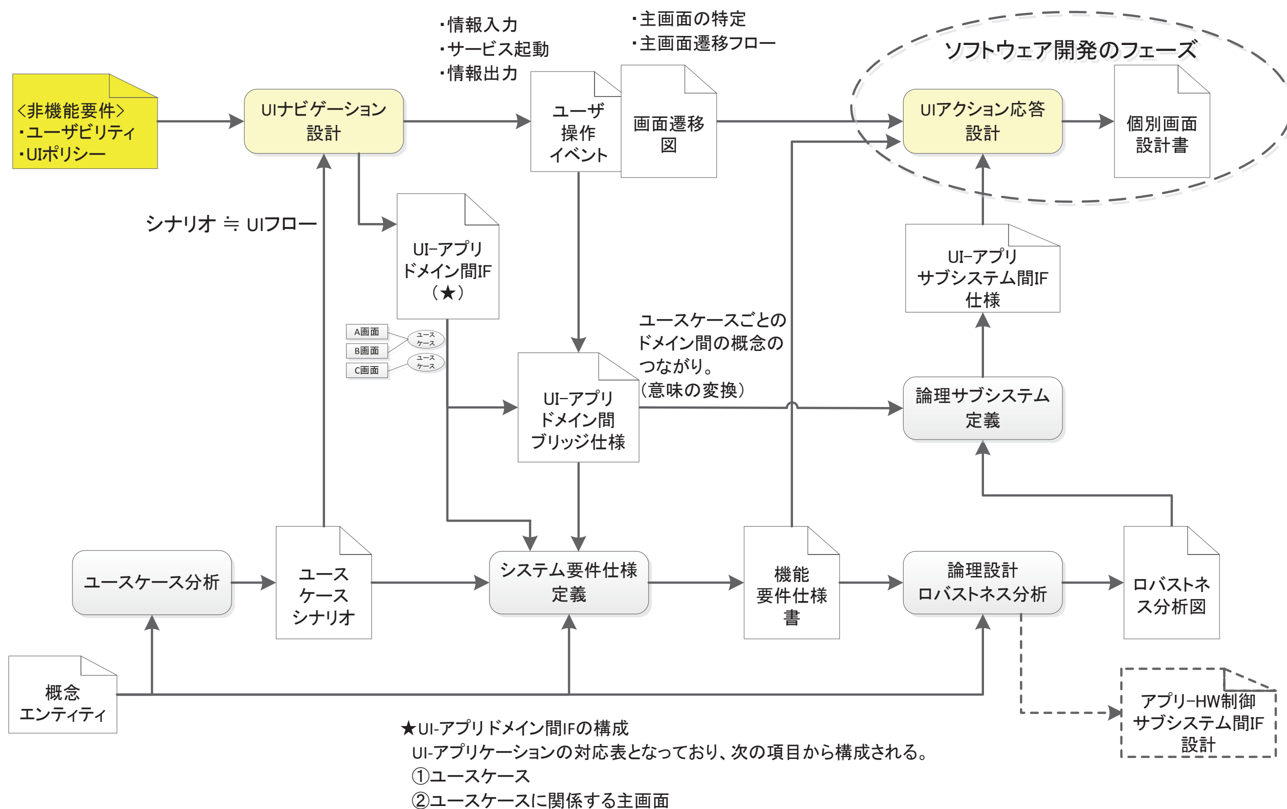


図 21　UI 設計プロセスの位置付け

個別画面仕様における“ユーザ操作・表示系部品をどの画面に配置するか”は、「機能要件仕様書」から分かる「ユーザへの表示項目（参照系）」と「ユーザからの入力項目（更新系）」の UI とユースケース間の相互作用のタイミングを考慮することで決まる。

8. 大規模システムの要求分析テクニック

8.1　開発しやすくするために、ドメインやサブシステム単位に分解する

大規模なシステム開発の定石は、いくつかの小さな塊に分解して考えることである。この時、ドメインとサブシステムの概念を使って分解する。ドメインは、システム内部を「意味（専門知識）の境界」基準で、複数の問題領域に分割する。意味の分割なのでドメイン同士が機能的に重複することはないが、実装時のコンポーネントの境界と異なることもある。ドメイン定義で得られたドメインのうち、開発単位として大き過ぎるものについて、それらを開発しやすい単位（サブシステム）に分割する。分割のポリシーはデータ構造や機能ごと等、様々な基準に従って、サブシステム同士がなるべく疎になるように分割する（図 22）。

システム分解の最初は、業務や専門知識的に独立した塊をドメイン（さらには、サブドメイン）で分解する。

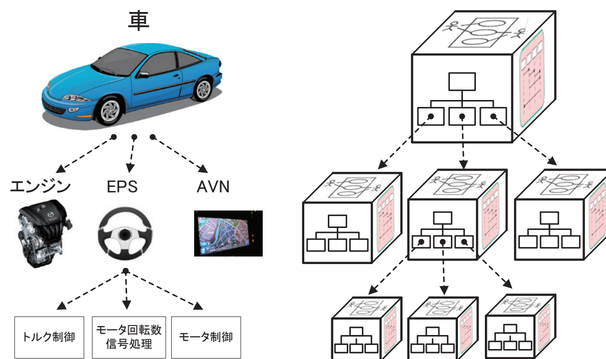


図 22　システム分解イメージ図

次に、一つのドメインに関して、それを大きな境界としてサブシステムを抽出する。サブシステムは、ユースケースから直接的には導き出せないので、設計手順を踏んで抽出する。

開発しやすい単位は、組織の方針によって変わる。最初にシステム全体の規模を想定し、それを何段階で設計するかを想定したら、各開発段階でのサブシステム分解粒度を定める。そして、各段階でのサブシステム分解粒度を目指して、ここで示したモデルベース設計手順に従って、様々な設計角度から（複数の設計図を作り出すように）設計を行う。一つの設計段階が終了したら、

そのサブシステムへの入力・出力を要求とみなして、次の段階のサブシステム分解を行う。これを、ソフトウェアとハードウェア単独の要件仕様が作成できるまで繰り返す（図 23）。

8.2　要件仕様作成～論理サブシステム決定までをアジャイルに回す

論理サブシステムを決定検討する際の方が見つけやすい要件仕様の不具合（欠陥・モレ）や要件仕様のグルーピングがあることは、多くの開発現場で経験している(6.1 節)。論理サブシステム決定は、方式設計の一部としているが、「要件仕様から論理サブシステム決定まで」を「要求の仕様化プロセス」であると定義して開発を進めても特に問題はない。むしろ、品質保証の観点からは、開発プロダクト間の関係性が強いものを一つの開発フェーズとして管理を行った方が、不具合発生の複雑な要因分析を行わなくて良い分、メトリクス分析による品質管理は実施しやすくなる。その場合、今まで利用していた設計プロダクトに関する品質基準を使って、新たな設計フェーズとしての設計プロダクト品質基準を修正確立しておけば良い。

8.3　設計書の適切記述分量（メトリクス）で品質を計る

記述の多すぎる設計書は必ず重なりがあり、その重なり間で矛盾が発生しやすくなる。記述の少なすぎる設計書は、モレ、欠落を早い段階で検出する可能性を低くする。最終成果物品質を制御するには、各設計段階で適切な量の記述を行うことが最低限必要である。この分量は、ソフトウェアの場合、例えば、１KL を構成するクラス数やシーケンス数、それらを正確に記述するためのソフトウェア要件仕様数、そして、ソフトウェア要件仕様を正確に記述するためのシステム要件仕様数、及びプロダクト量から逆算してメトリクス記述を決める。

8.4　要件仕様のプロダクトライン化

ユースケース、要件図、USDM を使った要件仕様、それぞれのレベルに適した粒度、記法を使って共通化を行う。ユースケースは包含関係を使い、要件図では、共通要件や共通仕様に印をつける。USDM は、表の列方向に、グループを列挙し、仕様との関係をマトリクスで表現する。

ただし、いずれも、明らかに独立性のある塊を共通化すること。ある部分が共通だという理由だけで、共通化して外出しすると、制御の流れが分断されてしまい、読みにくい要件仕様ができあがるので注意が必要である。

9. 設計手法改善により生産性向上させるには

9.1　設計ルールの組織的整備

５章の「要件仕様モデル間の関係性」は設計時に守るべきルールであり、設計内容の整合性を形式的にチェックする項目である。これらを多くルール化して組織的に運用することが生産性向上策のひとつである。複数のルールをツール化すれば、設計の自動化にもつながっていく。

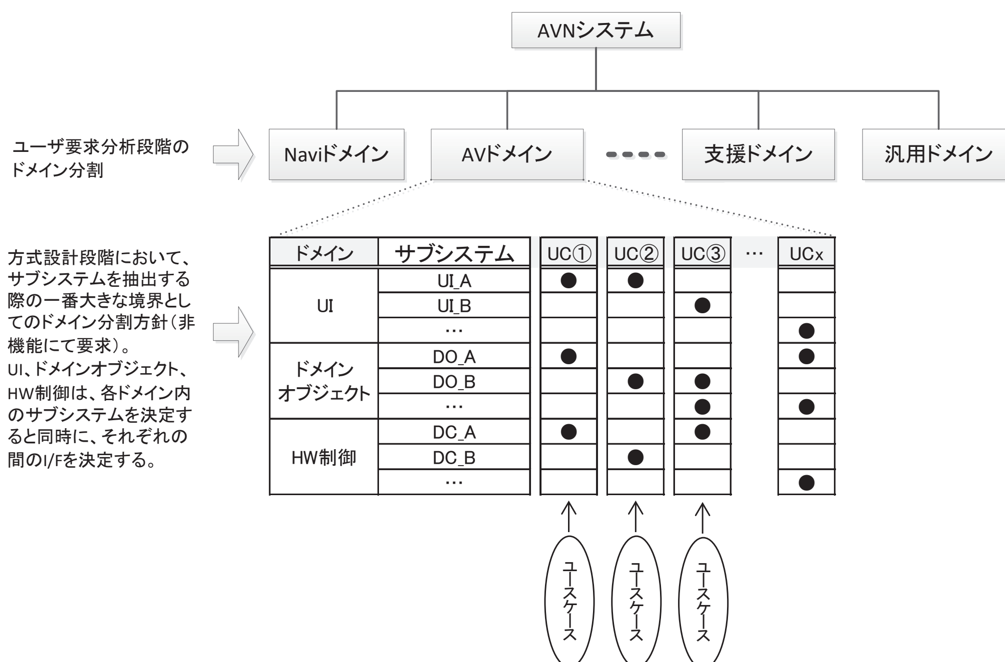


図 23　AVN システムのドメイン・サブシステム分解

9.2　複数視点の要件仕様を作成してもコストが下がる理由

要件仕様定義の変更コストを1とすると、他の開発作業コスト増大率は、設計で3-6倍、コーディングで10倍、テストで15-40倍、受入テストで30-70倍となる[5]。

ここでは、全体開発工数に占める要件仕様定義作業の割合を20％、要件仕様定義で5％の誤りを発生させているとする。要件仕様定義誤りのテスト段階における修正コストを30倍と仮定すると、この誤りを要件仕様の段階で落とすように開発プロセスを変更できれば、現在の全体コストの3割を削減できることになる。要件仕様定義作業／全体工数（20％）×誤り（5％）× 30倍 = 30％

9.3　要件仕様とテスト仕様の関係

要件仕様は適格性確認試験仕様の項目と1対1に対応する。したがって、試験の網羅性を上げる試験仕様作成を行うには、一つの仕様に対して、入力データの範囲の組み合わせを適切に設計することだけに集中すれば良いことになる。

9.4　どうやって、コスト低減を阻む障壁を取り除くか

新しい設計手法を導入する初期段階は、どうしてもそれを学習し身につける時間を要するため、コスト増となる。良い指導の下、多くの現場設計者に手法が浸透していけば、必ずコスト削減と品質改善の両方が実現できるが、その期間の長さによっては、コスト責任を負う管理者側は我慢ができなくなる。したがって、多くの改善で言われていることだが、コスト低減を阻む最大の障壁は身近な管理者であり、改善の初期段階から管理者を巻き込み、開発者と同じ気持ちで改善に当たれるようにすることが大変重要である（図24）。

9.5　熟練者の設計手順（潜在的手順ノウハウ）を形式化する

設計過程において熟練した有識者と経験を経ていない若年者とでどのような違いがあるかを表15に示す。熟練者の設計手順を細部まで形式化し、それを真似ることで、コスト削減・品質向上を図ることができる。

＜アンチパターン＞

(1)　設計能力を考慮せず、熟練者の手順で設計初心者に設計を行わせてはならない（設計プロセス依存症は必ず病気を発症する）。

(2)　要求・要件仕様定義の全てを方式設計に先だって決めることを目標にして設計順序を遵守してはならない（ウォータフォールを遵守することは最終目標ではない）。

10. むすび

本稿では、UML、SysMLといったモデリング言語を使って設計を行ったとしても、モデルを補完する資料を作成しなかったり、モデル間整合を行わなければ、モレ・矛盾を含んだ要件仕様が作成されやすいことを示した。しかし、モデル間整合設計ルール数が少な過ぎると、手順に従っただけの要件仕様ができあがり、モレ・矛盾を含むだけでなく、レビューがおろそかになり、却って不具合を増やす結果になることも経験的に分かっている。これらを改善するには、多くの設計ルール定義を行うとともに、定型手順や検証を、可能な限りツール利用という形で設計プロセスの制約とするのが良い。今後、この方向を進め、要件仕様の品質を一定に抑えつつ、作業効率向上が図れるプロセス作りを進めていく。

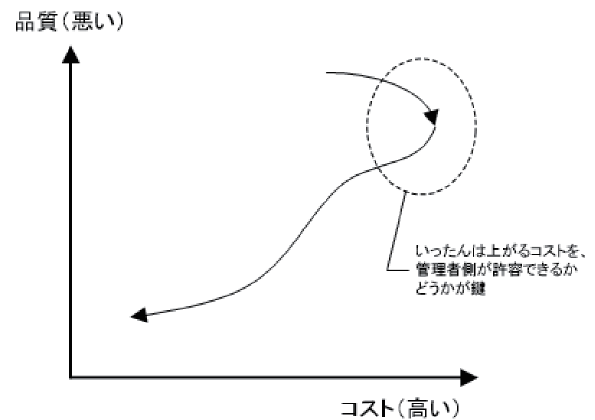


図24　新設計手法導入時のコスト・品質改善の進捗カーブ

表15　熟練者と若年者の設計手順の違い例

設計ポイント	熟練者	若年者	対策案
抽象化	本質的な問題を事前に抽出し、適切に問題を制限する条件をとらえることの重要性を実感的に把握している。	設計作業を形態の細かな調整等によって様々な問題を解決していくととらえている。	自由な手順で作成することを許す。ただし、生産性は測定する。
	（類似例）クラスから処理（シーケンスへ）	処理を考えながらクラスを作る。	
設計カタログの利用	形式知（カタログ）には重きを置かず、自分の経験を重んじる。	分析、つまり、既存技術システム分析（例）を重要視する。	若年者が熟練者の経験を要件図にする。
発想行為	余り重視していない。自分の専門分野だけで設計解を探す傾向にある。したがって、既に持っている解（シーズ）で解決してしまう。	かなり重要視している。新しく、技術根拠に基づいた設計解を探せる可能性がある。	ニーズ／シーズ対応の要件図を作成する。
スタート	シーズから仕事を始める。	ニーズから仕事を始める。	

11. 謝辞

本内容を実践の場に適用していただいた、様々な分野の社内外の方々からのフィードバックがあってこそ、最初は個人の経験則に留まっていた設計ルールを、多分野共通の設計ルールに変えることができています。変化する設計ルールを根気強く適用し続け、改善へのヒント、気付きを頂いた多くの方々に感謝の意を表します。

参考文献

(1) 組込みソフトウェア管理者・技術者育成研究会（SESSAME）：話題沸騰ポット（GOMA-1015 型）要求仕様書 第７版（2005）

(2) 益田 昭彦，ほか：新 FTA 技法，日科技連出版社（2013）

(3) 益田 昭彦，ほか：新 FMEA 技法，日科技連出版社（2012）

(4) 藤原 啓一：大規模組込システムの要求分析、システム方式設計、そして、ソフトウェア設計までをつなぐモデルベース設計手法，MSS 技報，**27**（2017）
http://www.mss.co.jp/technology/report/pdf/27_03.pdf

(5) Boehm，B. W.：Software Engineering Economics，Prentice Hall（1981）

(6) 藤原 啓一：要求から詳細設計までをシームレスに行うアジャイル開発手法，MSS 技報，**24**（2014）
http://www.mss.co.jp/technology/report/pdf/24-04.pdf

(7) 山本 修一郎：要求を可視化するための要求定義・要求仕様書の作り方，ソフト・リサーチ・センター（2006）

(8) 神崎 善司：顧客の要求を確実に仕様にできる要件定義マニュアル，秀和システム（2008）

(9) Beatty，J.，ほか：ソフトウェア要求のためのビジュアルモデル，日経 BP 社（2013）

(10) レフィングウェル，D.，ほか：ソフトウェア要求管理 新世代の統一アプローチ，ピアソン・エデュケーション（2002）

(11) 清水 吉男：［改訂第２版］［入門＋実践］要求を仕様化する技術・表現する技術，技術評論社（2010）

(12) Japan MBD Automotive Advisory Board（JMAAB）：要求開発ガイドライン Version1.0

(13) コーバーン，A.：ユースケース実践ガイド，翔泳社（2001）

(14) ワイルキエンス，T.：SysML ／ UML によるシステムエンジニアリング入門，エスアイビー・アクセス（2012）

(15) 鈴木 茂，山本 義高：実践 SysML その場で使えるシステムモデリング，秀和システム（2013）

(16) 独立行政法人情報処理推進機構：製品開発における SysML 適用の取り組み（先進的な設計・検証技術の適用事例報告書 2015 年度版）（2015）

(17) 児玉 公信：UML モデリング入門，日経 BP 社（2008）

(18) 浅海 智晴：上流工程 UML モデリング，日経 BP 社（2008）

(19) SESSAME WG 2：組込みソフトウェア開発のためのオブジェクト指向モデリング，翔泳社（2006）

(20) 渡辺 博之，ほか：組み込み UML eUML によるオブジェクト指向組み込みシステム開発，翔泳社（2002）

(21) 独立行政法人情報処理推進機構：障害未然防止のための設計知識の整理手法ガイドブック（2017）

(22) レブソン，N. G.：セーフウェア，翔泳社（2009）

(23) 橋本 正明：システムの非正常系の要求分析，情報処理，**49**，No.4，380 ～ 385（2008）

執筆者紹介

藤原 啓一

1985 年入社。防衛のソフトウェア開発に従事以降 カーナビ、気象レーダ、ニューラルネット、医用画像、衛星通信、業務系システム、通信システム、車載制御システム等のシステム設計、ソフトウェア開発に従事。2015 年から副事業部長、兼技術部長。