PART Ⅱ

設計事例

目次

1．はじめに

PARTⅡは、「設計」における適用事例を記載したものである。全体的な目的や共通事項は、「PART Ⅰ　概要」を参照願いたい。

ソフトウェア開発現場での認識として、要件定義プロセスも含めた設計工程が、後の工程での手戻りやソフトウェア品質に多大な影響を与えるといわれている。

この部分に工夫を加えることによって、ソフトウェアの信頼性向上につながるのではないか、さらに、そこに様々な取組みを行っている事例が多く存在するのではないかと考え、企業・団体に情報提供をお願いしたところ、数多くの事例を紹介していただいた。

「４．各事例報告」で、詳細を紹介する。

なお、掲載している適用事例は、様々な技術や手法を、様々な目的、分野に適用しているものであり、それぞれに適用の可否を判断する前提条件がある。したがって、すべての企業やプロジェクトへの適用を推奨するものではないことを了解のうえ、皆様の取組みに合った事例を参考にしていただきたい。

2. 事例一覧

PARTⅡで報告する設計事例の一覧を次に示す。

表 Ⅱ-2-1　収集事例一覧

	No.	標題	事例提供元
設計事例	A-1	アシュアランス技術を用いた鉄道信号の革新	東日本旅客鉄道(株)
	A-2	XDDP 導入による派生開発の品質改善とその効果	(株)日立産業制御ソリューションズ
	A-3	組込み系の利用品質における「HMI 品質メトリクス」開発と適用事例	(株)U'eyes Design
	A-4	要件定義段階における信頼性向上の取り組み事例紹介	ビッグローブ(株)
	A-5	要件定義の品質向上に向けた取り組み	富士通(株)
	A-6	ジェネレータツールを利用した高信頼開発、高速開発の実践	(株)市進ホールディングス
	A-7	設計工程における TERASOLUNA DS の適用	(株)エヌ・ティ・ティ・データ
	A-8	Grails/Groovy の適用推進	エヌ・ティ・ティ・ソフトウェア(株)
	A-9	個人依存開発から組織的開発への移行事例	三菱電機メカトロニクスソフトウエア(株)
	A-10	MBSE による双腕作業ロボット動作実行系のコンセプト設計	(独)産業技術総合研究所(AIST)
	A-11	仕様記述言語 VDM++ を用いたシステムの仕様の記述	フェリカネットワークス(株)
	A-12	車載 ECU 開発における上流工程での品質確保	東芝情報システム(株)
	A-13	独自開発したモデル駆動開発プロダクトラインエンジニアリングの実践	(株)デンソー

３．設計適用事例の状況

PART Ⅰ　概要　にも記載したが、以下のような観点で事例を収集した。

（１）Ｖ字モデルのプロセスにおける左辺の活動の成果が、右辺の検証の成果にポジティブに影響する。

（２）左辺の活動としては、要求の早期獲得のみならず、より厳密な獲得が以降の活動における手戻りを減少させる。

（３）結果として、左辺において、その中でもより上流、即ち要求獲得及び定義、更にそれをベースとした仕様記述への工数のシフトが起こる。

（４）要求獲得及び定義、更に仕様記述に関わる活動として、モデル化及びモデルの実行あるいはシミュレーションが行われる。

（５）結果として、上流における品質の作り込みが開発プロセス全体における手戻りを減少させると共に検証への工数を削減させ、生産性が向上する。

（６）更に、リリース時の品質も向上する。

この仮定、特に「設計」工程に関するものは、ほぼ妥当であったと考えられる。それは、後述の各事例で証言されている事項をピックアップすることにより、確認することができる。なお、ここでは全ての事例を取り上げておらず、効果等が明示的にわかるものを引用した。

【設計事例Ａ−２】XDDP 導入による派生開発の品質改善とその効果

品質評価において、設計段階で非適用に比べて１５４％の不良が発見され、顧客先での発見は８４％も減少している。

工数評価においては、設計段階への工数は４０％増えたが、総合テスト段階の工数は４０％減少しており、総じて若干の減少につながっている。

【設計事例Ａ−５】要件定義の品質向上に向けた取り組み

要件定義工程に対して、期間で２．３倍、工数で１．５倍かけたが、開発全体でみれば、期間は５％短縮、工数は４．５％短縮となっている。

【設計事例Ａ−６】ジェネレータツールを利用した高信頼開発、高速開発の実践

要求定義に５ヶ月、繰り返しのプロトタイプ作成に７ヶ月を配布したが、実装は４ヶ月であった。実装期間には検証も含む。

【設計事例Ａ－７】設計工程における TERASOLUNA DS の適用

結合試験時のバグ原因の４７％は設計書が原因。TERASOLUNA DS の設計工程への導入により抽出されたエラーは、大きな手戻りにつながるものが２５％、次工程で気づくと推定されるものが２２％であり、開発ライフサイクルの上流の段階でのエラー抽出が可能である。

【設計事例Ａ－９】個人依存開発から組織的開発への移行事例

技法適用前の２００５年では、分析/設計の工数割合は１０％以下で、作成とテストの工数割合が９０％をしめていた。技法適用の結果、上流で不具合検出率が９７％で、年々テストの工数割合が減少し、分析の工数割合が増加している。現在、分析の工数割合が約５３％、テストが約２０％である。

【設計事例Ａ-10】MBSE による双腕作業ロボット動作実行系のコンセプト設計

目標とした以下の項目はほぼ達成できた。

(a) SysML を用いて要求の関係を具体的に表現すること
(b) 分析した要求を満たすアーキテクチャ設計を行うこと
(c) 工程の成果物や参照された文書間のトレーサビリティを実現すること
(d) 要求を追加する場合に、効率よくシステムを拡張していけること

【設計事例Ａ-11】仕様記述言語 VDM++を用いたシステムの仕様の記述

形式手法記述手法の利用により達成したかった下記の目標を達成できた。

(a) 厳密な仕様の策定
　形式仕様記述言語による６７７ページの外部仕様書の策定時に、４４４件の欠陥を発見した。
(b) 開発プロセスの策定と導入
(c) 仕様の多方面からの分析・精査
(d) 仕様テスト
(e) コミュニケーションの促進

結果として、上流工程で厳密に仕様を記述、検証するのに従って、開発全体の中で、仕様策定工程が占める割合が大きくなる。そして、従来、下流の工程で問題が顕在化していた課題を、上流工程における厳密な仕様の記述と検証と共に解決した。

【設計事例A-12】車載 ECU 開発における上流工程での品質確保

これまでの要求分析は、特定のスペシャリストが、頭の中で展開し、設計に落としてきた工程であったため、体系的プロセスがなく、検討した経緯を示すドキュメントなどはほとんど残っていないのが現状である。本プロセスに従って、SysML を用いて要求分析を行ってみると、検討した過程/結果が非常にわかり易く、設計変更や差分開発時に本プロセスに従って要求分析を行った SysML 準拠のドキュメントがあれば、新しい要求を追加する際の影響分析などに非常に有効なドキュメントになることが分かった。

【設計事例A-13】独自開発したモデル駆動開発プロダクトラインエンジニアリングの実践

２００８年以降、延べ３０を超える製品を本手法で開発している。本手法の適用の前後でソフトウェアの規模やバリエーションの数自体が変化しているため、単純に計測することは難しいが、生産性の面で 2 割程度の効率化が認められた。また、コア資産を１００%再利用し、手動でのコーディングなしで多様なバリエーションを開発できるので、品質向上にもつながっている。

4. 各事例報告

（番号の付与体系について）

各事例の報告に先立ち、番号の付与体系について触れておく。

以降の事例報告は、全体的な章立てから考えると、PART Ⅱの第４章に属するものであり、例えば事例の１番目は、PART Ⅱ．４．１という章立てになる。しかしながら、それぞれが独立の読み物としても成立するものであること、及び、その中に記載の番号体系が深くなりすぎると読者の混乱を招くことから、以下では次のような番号付与体系としている。

（１） PART Ⅱの第４章、すなわち「設計事例」を表す記号として「Ａ」を用いる。上述の事例１番目の例では、「Ａ－１」という表記を行うことにする。なお、PART Ⅲの「検証事例」においては、「Ｂ」を用いて本篇の「設計事例」と区別している。

（２） ページ付与は、PART Ⅱで通し番号としており、Ⅱ－１ 、Ⅱ－２ 、・・・としている。

A－1

アシュアランス技術を用いた鉄道信号の革新[1]

1. 概要

本編では、列車システム開発においてアシュアランス技術を適用した、東日本旅客鉄道株式会社の事例を紹介する。

日本の社会は、少子高齢化、グローバル化、景気の低迷などの大きな環境の変化を受けており、社会の構造・組織などにも影響を与えている。これらの影響はシステムにも及んでおり、これまでのシステムには信頼性・安全性だけが求められていたものが、今や社会や利用者からの要求の変化に即座に対応し、その責任を果たし続けることが求められてきている。また、ネットワークの発達に伴い多くのシステムが有機的に接続されるようになっている。このようなシステムでは異種のニーズを持つだけでなく頻繁にニーズが変化しているなど、システムの外的変化及び内的変化の両方に対して対応が求められており、システムの安定稼動が今まで以上に求められていくことになる。システムの社会における責任はより一層重くなっていると言える。

社会基盤の一つである鉄道システムにおいても、異種のニーズへの対応と状況変化への適応が求められている。これら異種性と適応性に対応できる技術をアシュアランス技術と呼び、欠くことのできないものとなっている。

2. アシュアランス技術の歴史

アシュアランス技術研究については、社会情勢の変化する環境の中で、米国において研究が始められた。米国政府が中心となり、大統領令によって CIAO（Critical Infrastructure Assurance Office）といったアシュアランス研究機関が設立されたほか、DIAP（Defense-Wide Information Assurance Pro- gram）計画などのプロジェクトも進行している。こうした背景から、米国、日本及び欧州の研究者が共同で 1996 年に、アシュアランス技術に関する議論の場として、IEEE 主催の国際会議 HASE（High Assurance Systems Engineering Symposium）が設立された。HASE は 1996 年以降毎年開催されており、毎回 30 件以上の発表が行われている。

もともと、HASE は国防総省などが主体となっており、軍事技術関連が中心であることから、北米以外での開催は許可されていなかったが、電子情報通信学会でのアシュアランス研究会などにおける活発な活動が認められ、2002 年の大会 HASE2002 は日本の東京に

[1] 事例提供：東日本旅客鉄道株式会社　電気ネットワーク部　松本　雅行 氏

おいて開催されている。

3. アシュアランス技術とは

3.1. 定義

アシュアランス技術とは、システムの安定稼動を保証する技術で、異種性と適応性の 2 つの要素を考慮したシステムに適用される技術の総称であり、ここでは、この異種性と適応性の 2 つを合せ持つシステムをアシュアランスシステムと定義する[1][2]。

3.2. 異種性

システムは信頼性､安全性をはじめ多くのニーズを満足させるものとして構成されている。このニーズの要求度合いはシステムによって異なっており、これをニーズレベルと呼ぶ。システムの稼働の保証に求められるものとして、例えば、以下のようなニーズが挙げられる。

- 信頼性
- アベーラビリティ
- フェイルセイフ
- 安全性
- リアルタイム性

システムを設計する際には、これらのニーズごとにニーズレベルを満たす必要があるが、そのレベルはシステムごとにまちまちである。例えば、制御システムにおいては安全性とリアルタイム性が求められているのに対し、情報システムではアベーラビリティや信頼性がより求められる。もちろん、これらのレベルを、すべてのシステムのニーズよりも高く設定すれば、すべてのシステムのニーズを満たすことができるが、それは現実的ではない。異種のニーズレベルを持ったシステムの稼働を保証するため、異種のニーズレベルの共存を許容する能力のことを異種性と呼ぶ。

3.3. 適応性

システムの置かれている環境は、そのシステム自身も含めて、常に変化し続けている。この変化には、システムの成長、システムの初期設定、システムの移行、システム変更と改修、システムの異常などがある。システムが置かれている環境は、そのシステム自身も含めて、常に変化し続けている。従って、そのシステムが置かれる環境を事前に仮定し、システム構築時に設計に反映させるといったことは不可能である。

また、たとえば通勤電車での日中の時間帯、朝夕ラッシュ時と終電の時間帯では時間に対するお客様の制約や要望が異なるため、システムに求められる信頼性やリアルタイム性のレベルは時間ごとに変化する。

こういった、システムの状況によるニーズの変化や時間的なニーズの変化に対応するためには、常に柔軟に対応しうるシステムの構築が必要となる。このような状況変化への対応能

力を適応性と呼ぶ。

4. 鉄道における適用例

4.1. 列車輸送管理における適用

列車輸送管理においては、運行制御や設備の管理などの制御システムと、運行情報の伝達や旅客への案内といった情報システムとが共存している。信号機や転てつ機などのデバイスをコントロールする制御系のシステムは、扱う情報量は少ないがそのリアルタイム性、安全性に対する要求は非常に厳しい。一方で、乗客への情報サービスや運行計画の管理などを行う情報系では、制御系に比べて処理の速度は秒単位を求められるものではなく、フォールトトレランス性に対する要求も高くはないものの、データベースなどの大量のデータを処理する必要性があるため、高速かつ広域の情報処理能力が求められている。このように、輸送管理においては情報系と制御系という異種のニーズを持ったシステムの共存、異種性が求められている。

これに対応したシステムが列車の運転を管理するための運行管理システムと呼ばれるシステムが運転線区ごとに作られている。このシステムは、地理的にも広範囲で大規模であり、システム化も長期間を要すため、必然的に段階的な構築が必要となる。また、稼動したシステムは列車運転の性格から 24 時間連続運転システムとなり、システム拡帳時にも稼動システムの安定稼動を保証した無停止拡張が前提条件となり、かつ使用者の新しいニーズ（使用実績を踏まえての機能向上等）を吸収した変化･成長を前提とした構築が必要となっている。つまり、適応性が求められている。

同社はこれらの課題を克服するため､自律分散システム技術 をベースとしたアシュアランス技術を活用して､首都圏 ATOS（Autonomous Decentralized Transport Operation Control System）と呼ぶシステムを拡大・成長させつつ 20 線区、182 駅に展開してきた[3]。

①　システム線区展開・拡大の課題

ATOS をＪＲ東日本の東京圏の鉄道に導入し、線区展開・拡帳する場合、次のような技術的な課題を克服する必要がある。

a. 大駅では、複数線区の列車が運行されているが、線区のシステム化を一斉に実施することは困難なため、段階的な構築が必要となり、システム化は長期間となる。
b. 線区展開・拡大したシステムは 24 時間連続運転システムとなり、システム拡大において安定稼動を保証して無停止拡張が必要となる。
c. 各線区は相互乗入れのため独立システムにならず、システム構築中には、ネットワーク、コンピュータ上に稼動に必要なデータと未稼動設備を試験するためのテストデータを共存させる必要がある。
d .システム構築には長期間を要するため、使用者の新しいニーズ（折返し運転支援機能の

充実等の使用実績を踏まえての機能向上等）を吸収、システムを成長させる必要がある。

② 段階的構築の概要

アシュアランス技術は、高いレベルでの信頼度を保証しつつシステムを変化・成長させる技術であり、運行管理システムを大規模線区に展開・拡帳する手段として有効である。ソフトの適切な管理を実施するとともに、稼動中設備を制御するデータと試験のためのデータの混在を可能とすることにより、システム評価・品質の保証とシステムを段階的に無停止拡張することができた。

このデータの混在は、第一段階は駅装置コンピュータで、第二段階は中央ネットワークで、第三段階は中央装置コンピュータで、第四段階は運行管理ネットワークで実現した。

③ 段階的構築の手法

本システムは、自律分散システムであり、分散配置した駅装置が自律的に動作しながら、中央装置も１構成要素として全体統括を分担し、トータルで１つのシステムとして稼動できる。中央と駅、駅と駅相互間の通信では、個別診断などを除いてシステム間の直接やりとりをしないようにしている。各装置は、データの内容を表わす『機能コード』を付加したメッセージを宛先指定せずにネットワークへ送出するブロードキャスト伝送を行っている。そして受信側では各装置自身が機能コードによって必要な情報を取捨選択する方式としている。さらに、既に稼動している装置が送出するデータと試験中の装置が送出するデータとを区別する『フラグ』を設け、受信側の装置がアプリケーションレベルでフィルタリングすることが可能となるデータ構造としている[4]。

本システムでは、既に稼動している装置と試験中の装置が存在し、装置及びネットワークに稼動中設備が必要となるデータと試験中の装置が送出するデータとが混在するため、それぞれに支障しない基本的な仕組みが必要となる。このため各装置にはモードを定義し、このモード状態に応じた動作を行うようにアプリケーションレベルで規定することとしている。図 A-1-1 に示すように、各装置はネットワークに、前述したモードに対応したフラグを付加したデータを送出し、受信側の装置では機能コード及びフラグにより、自装置のモードと対比して、その装置自身が動作を決定する仕組みである。

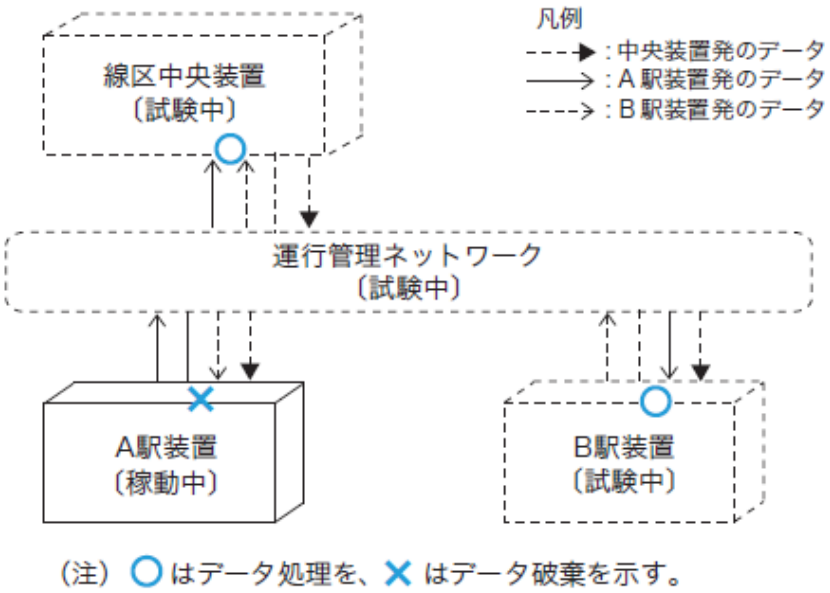

図 A-1-1　段階的構築の基本的な処理の概要

4.2. 列車制御システムにおける適用

高密度運転線区の列車制御システムにおいては、高輸送力、高安全性、高信頼性が求められる。当初の自動列車制御システム（以下旧 ATC と呼ぶ）は、固定した区間ごとに列車の存在を検知し、そこに列車を進入させるかどうかの判断により、各列車の速度を地上の集中制御装置で求め、それを各列車の車上制御装置に指示していた。これは、ブレーキ性能の一番悪い車両に合わせた区間長とするため、列車運転間隔が長く高密度輸送は難しかった。そこで、各列車が、自らの位置を認識し、地上からは停止する位置のみを列車に伝送する。この停止位置に対応したブレーキパターンに基づいて列車速度の制御を自律的に行う自律分散型列車制御システム（以下 D-ATC と呼ぶ）の開発が必要となった。さらに、新しい列車制御システムに取り替える際には、新旧両システムが共存する中、D-ATC の構築及び試験のためにその旧 ATC を止めることは出来ないため、システム全体の運行を妨げず、D-ATC の段階的投入と、旧 ATC を持つ列車とを共存させながらのオンライン稼動中のテストを保証するアシュアランスシステム構築技術を確立する必要がある。

一般にシステムは単体で動くことはまれで、他の異種システムとの情報の授受や、システムを取り巻く外部環境の変化の影響を受けながら稼働している。アシュアランス技術とは、このように複数のシステムがネットワーク等を介して互いに接続されたとき、それぞれの異なった目的や機能が互いに妨害されることなく、状況に応じて連携し共存できることを保証する技術であり、列車制御システムにもアシュアランス技術の適用が求められている。D-ATC における個々の課題について、どのようなアシュアランス技術を用いて解決したか、その方策を以下に述べる[5]。

① システムの拡張

ATC の更新においては、全線区を一度に更新するのではなく、幾つかの段階に分けた形で更新を行う。D-ATC の導入を予定している線区は、旧 ATC が設置されている区間である ため、本システムの導入に当たっては、システムの境界箇所において旧 ATC と D-ATC の新旧切換制御が必須となる。旧 ATC 区間から D-ATC 区間に列車が進む時は、有効な電文を受信することによって D-ATC 制御となり、逆の場合は切換区間の電文の「切換」情報を読み取り旧 ATC 制御に移行する（図 A-1-2）。

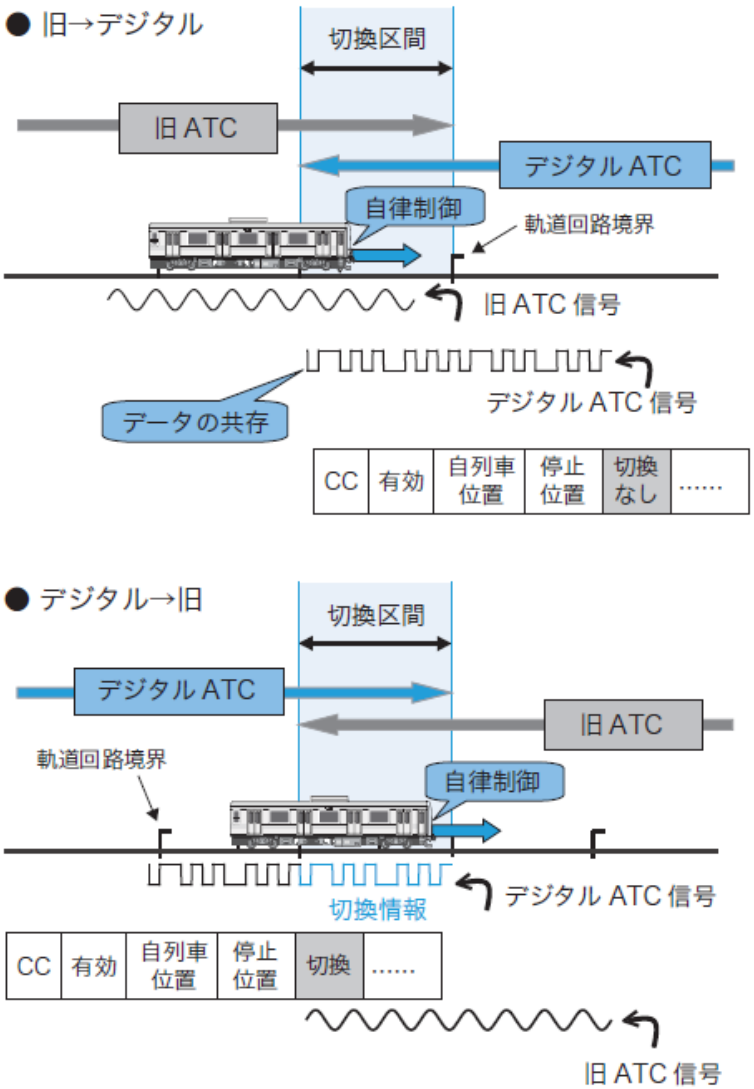

図 A-1-2　システム境界箇所における切換制御

② 異種システムとの共存

D-ATC における異種システムとは、同じ列車制御システムでもニーズレベルが異なる旧 ATC と自動列車停止装置（ATS）がある。これらのシステムが工事期間、試験期間また使用開始後も共存できる必要がある。

そこで、D-ATC では次のようなソフト、ハードの対策を施した。

a. D-ATC、旧 ATC 両方の信号が軌道回路に重畳できるように、旧 ATC で使用されていた 2.8～3.8kHz の周波数を避けて、D-ATC の周波数帯を 11.9～13.1kHz とした。また、軌道回路割の変更を前提にして、無絶縁軌道回路を採用し、現行の信号波は阻止し D-ATC 波のみ通過できるバイパスボンドを開発し挿入した。

b. D-ATC 電文には「有効/テスト」情報を持たせて、この情報が「テスト」の場合には車上装置は旧 ATC で制御できるようにした。

c. 車上装置は、D-ATC、旧 ATC 両方の信号波を受信し、その状況に応じてどちらのモードで制御するか自律性を持たせた（図 A-1-2）。

③ オンラインテスト

新旧システムを共存できるようにしたおかげで、旧 ATC を運転しながらテストを行うことも可能となった。昼間、旧 ATC の信号に重畳させて D-ATC のテストデータを流す。車上システムはテストか有効かのフラグを見て処理するので、テストデータの場合には旧 ATC の信号に従って運転する。つまり、地上のシステムのテストは昼間列車の運転時間に十分なテストを行うことが出来る。また、車上システムのテストも、列車運転は旧 ATC によって行って、D-ATC の電文による確認試験を行うことも可能としている（図 A-1-3）。

④ 短時間でのシステム切換

限られた時間で D-ATC の使用開始切換を行うには、最小限の操作で D-ATC の切換を行うシステムとしなければ ならない。

具体的には、D-ATC 電文の「有効/テスト」情報を利用して、この情報を変更するだけで車上装置がいずれのシステムで動作するかが選択できる。これにより、極めて短時間でシステムの切換ができるようにした（図 A-1-3）。

⑤ 列車運転の継続性

D-ATC が設置される区間と、旧 ATC または ATS 区間が設置されている区間をまたがって運行する列車は、その境界でシステムの切換が必要となる。通常は、停車後、 地上システムの切換と車上システムの乗務員による手動切換を行うという方法をとっている。そこで、切換区間の D-ATC 電文に「切換情報」を付加し、D-ATC から旧 ATC へはこの情報により自律的に切換を行なう。また、逆の場合は D-ATC 電文を受けることによって D-ATC に自動的に切り換わる。このような構成とすることにより運転の継続性を図ることができた。

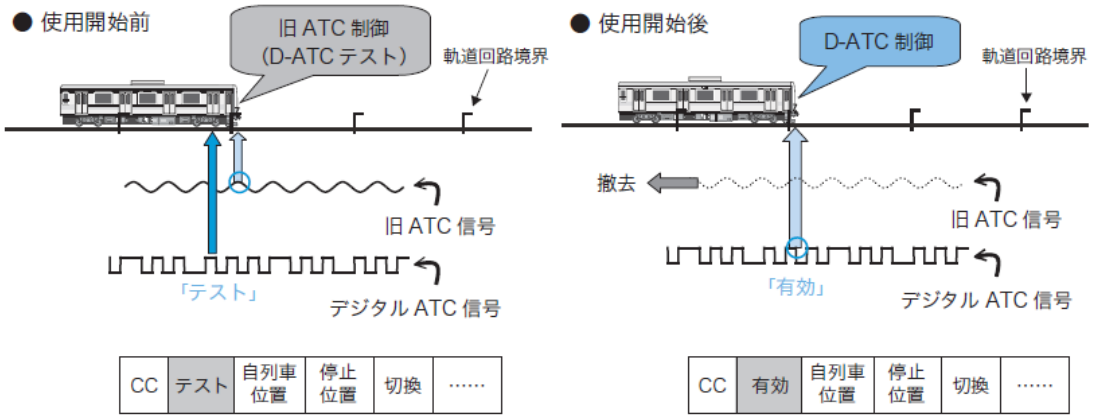

図 A-1-3　オンラインテストと使用開始切換

5. あとがき

アシュアランス技術の応用例として、鉄道システムについて紹介したが、これ以外にも宇宙応用、医療応用など情報サービスシステムなどの多くの社会インフラシステムに実用化が進んでいる。異種システム共存技術やオンラインテスト技術を活用することにより、システムの拡張やシステムを稼動させながらの保守管理ができるようになる。同社はこのアシュアランス技術を用いて鉄道システムの革新を今後ますます進めていきたいとしている。

参考文献

[1] I-Ling Yen、“Toward Integrated Methods for High-Assurance Systems”、IEEE Computer、1998

[2] 森 欣司、「アシュアランスシステムのニーズと技術動向」、電子情報 通信学会、アシュアランス研究会、2000

[3] 上条　恵司他、「アシュアランス技術（運行管理システムの大規模展開）」、日本鉄道電気技術協会、2001

[4] 森 欣司、「自律分散システム入門」、森北出版株式会社、2006

[5] 松本 雅行、「新しい列車制御システムの開発とアシュアランス技術」、日本信頼性学会、2000

(このページは空白です)

A－2

XDDP[1]導入による派生開発の品質改善とその効果[2]

1. 概要

本編では、組込みソフトの開発において、XDDP(eXtreme Derivertive Development Process)を適用した株式会社日立産業制御ソリューションズ[3](以下、同社とする)の事例を紹介する。同社の XDDP 導入経緯やその適用効果についても報告する。

同社では、2007 年頃から派生開発に特化した開発アプローチである XDDP(eXtreme Derivertive Development Process)の適用を進め、ソフトウェア派生開発プロセスの強化を図ってきた。

XDDP の特徴的な成果物である、「変更要求仕様書」、「変更要求トレーサビリティ マトリックス」、「変更設計書」の作成支援ツールも独自に開発して全社レベルで 適用促進を図っている。これまでに、400 件以上の案件で適用し、品質改善効果も顕著に表われてきている。

2. 取組みの目的

同社は、日立グループの中核企業の一つとして、創業以来培ってきた「情報と制御の融合技術」により、電力、交通、上下水、金融などの「社会インフラ」や、鉄鋼、自動車、電子デバイス製造などの「産業基盤」を支える監視制御、運行管理、生産管理、画像検査をはじめとする、さまざまな高信頼システムを提供してきた。

同社では、過去のソフトウェア資産を基にした改造や機能追加など、いわゆる派生開発が全体の約 8 割を占めている。

派生開発は新規開発に比べ、短納期・低コストの開発が要求され、他人が書いたソースコードを読まなければならない場面も多いため、「全体を理解」することが難しい。

このように全体を適切に理解できていない状態（部分理解の状態）では、思い込みや勘違いが起こりやすくなる「部分理解の罠」に陥りやすい。

同社では、この派生開発における品質の向上を目指して、ソフトウェアの派生開発のプロセスを強化するために、XDDP を導入してきた。

---

[1] XDDP(eXtreme Derivertive Development Process)は、株式会社システムクリエイツの清水吉男 氏が提唱する、派生開発に特化した開発アプローチである。

[2] 事例提供：株式会社日立産業制御ソリューションズ　櫻庭　恒一郎　氏

[3] 株式会社日立産業制御ソリューションズは 2014 年 4 月 1 日付けで、株式会社日立情報制御ソリューションズより社名変更した。

2007 年頃、同社では組込みソフトの分野で大規模な受託開発プロジェクトを遂行中で、そこで派生開発を繰り返していた。 このプロジェクトでは、トラブルがなかなか収束しない状況が続き、何とかこの状況を打破しなければならない状況に追い込まれていた。

そのようなときに、株式会社システムクリエイツの清水吉男氏が XDDP を解説していた書籍[1]に出会い、この方法が当時発生していた問題解決に有効なのではないかと考えた。

当時、派生開発の現場で発生していた問題を以下に列挙する。

(1)　新規開発については、厳格なプロセスが細かく定義されており、それに従って組織的にプロジェクトが進められていた。一方で、派生開発のプロセスも定義されてはいたものの、担当者の力量に依存するような曖昧な定義が見受けられ、過去に開発した資産があるための安心感も手伝ってどうしても管理が「緩く」なりがちだった。

(2)　当時の派生開発では、いくつかのドキュメント成果物を作成していたが、「どう直す」という視点で、検討した結果のみを記載していたことが多かったため、担当者の誤解や考慮漏れに起因する、変更要求と修正内容の不整合や修正箇所の漏れといった問題をレビューで摘出できなかった。

(3)　修正を任された担当者は、既存のソースコードを調べて修正箇所を探していくが、修正内容について深く検討しないままで修正に着手してしまうため、修正による思わぬ影響が出たりして混乱状態に陥っていた。

(4)　改造を繰り返していくうちに、類似処理をコピーしてその場しのぎの改造をしてしまうことが多くなっていた。そのほうが、他処理への影響を考えずに済むためである。しかしながらこれを繰り返した結果、いわゆるクローンコードが増殖して改造しづらい複雑な構造になってしまっていた。

(5)　顧客から「実績があるから大丈夫」と言われて既存のソースコードを貰いうけて改造する際に、全体を理解しないまま担当者の力量に依存した状態で手を加えた結果、思わぬところで（後のフェーズになって）改造漏れや改造誤りが発覚した。

当時は、上述したような様々な問題が輻輳し、プロジェクトが混乱していた。これらの問題には派生開発特有の「部分理解の罠」に起因するものが多く、「部分理解の罠」を克服する仕組みが必要であると考え、XDDP を導入して派生開発のプロセスを強化・改善することとした。

3. 取組みの対象

3.1. XDDP とは？

ここで、XDDP について、その概要を簡単に説明する。

XDDP は、株式会社システムクリエイツの清水吉男氏が考案した、派生開発に特化した開発アプローチである。「短納期」や「部分理解」といった派生開発特有の問題に合理的に対応

する方法として多くの現場で採用され、不具合の大幅な減少や工数の削減を実現している。

XDDP は、派生開発の要求に対して必要不可欠なプロセスと成果物の連鎖で構成されている。

変更する箇所の情報を「変更要求仕様書」、「変更要求トレーサビリティ マトリックス」、「変更設計書」の「3 点セット」の成果物に記述することで、各担当者が予定している変更内容や変更箇所、具体的な変更方法をプロジェクトメンバー間で共有することができ、派生開発フェーズの早い段階で誤解や考慮漏れに起因する不整合を摘出することが可能となる(図 A-2-1)。

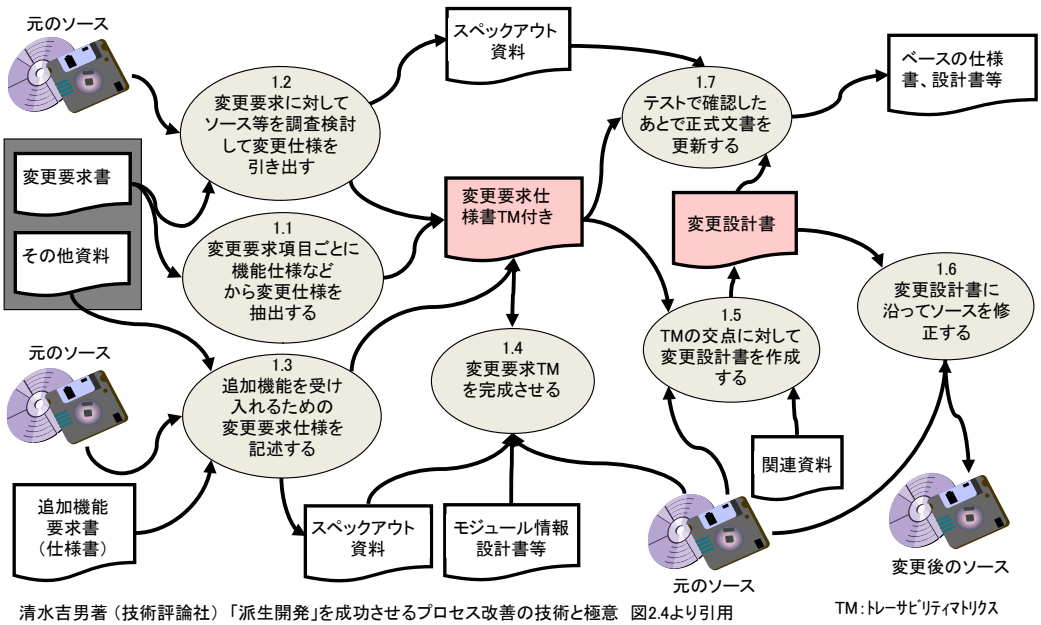

図 A-2-1　XDDP のプロセス概要

XDDP では、変更作業は原則として「差分」で進めていく。ソースコードの変更も事前に差分情報を揃えておいて一気に作業に着手し 1 回で変更作業が終了する。その結果、ソースコードの変更し直しがなくなり、必要最小限の工数でソースコードの変更が可能となる。

公式文書の更新は、テストで変更の正しさが確認されてから実施する。

3.2. XDDP 導入による改造プロセス強化

XDDP を導入する前の派生開発プロセスの概略を図 A-2-2 に示す。

顧客などからの変更要求に対して、修正対象を明確化する「修正計画書」や修正内容を記載する「改造仕様書」などのドキュメントを作成することになっていたが、どちらも結果しか書かれないことが多いため、修正対象が十分か、変更要求に対する誤解が無いか、といった観点でのレビューが十分機能していなかった。

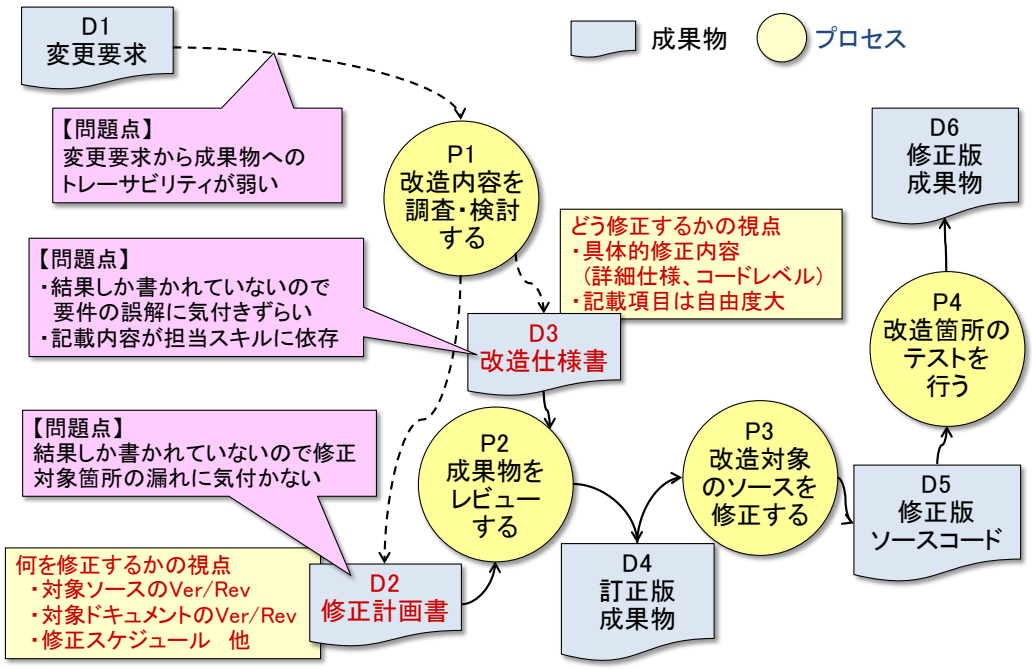

図 A-2-2　XDDP を導入する前の派生開発プロセス

これに対し、XDDP 導入後は、XDDP の成果物 3 点セットである「変更要求仕様書」、「変更要求トレーサビリティマトリクス」、「変更設計書」、の作成とそれらのレビュー実施により、派生開発の上流フェーズのプロセスが強化できた(図 A-2-3) 。

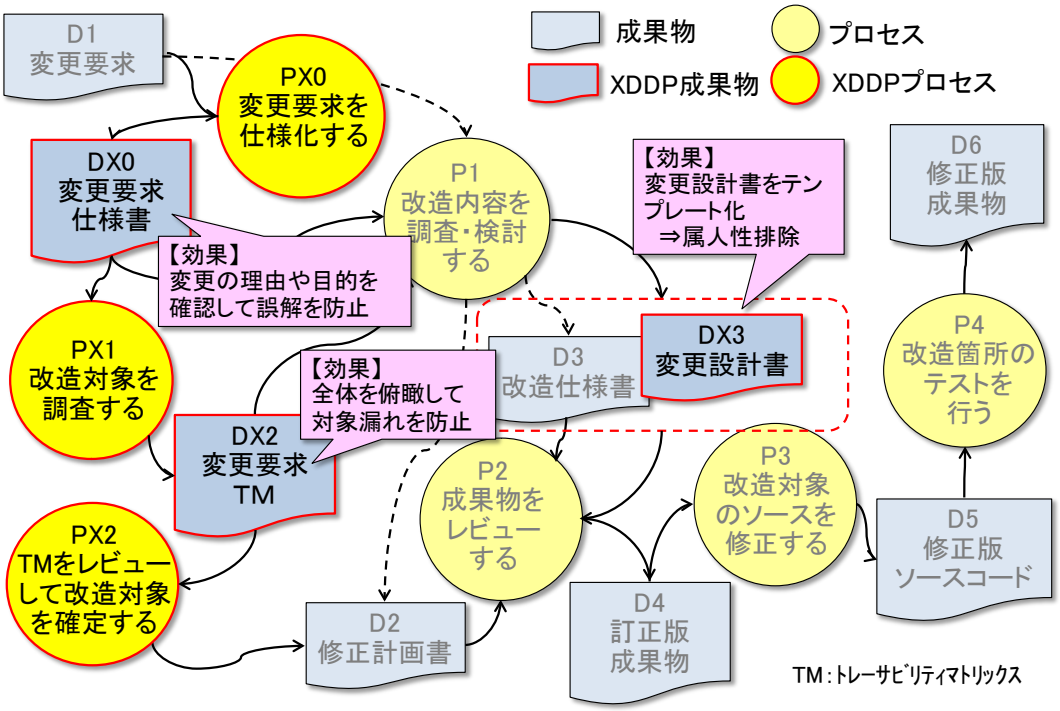

図 A-2-3　XDDP を導入した後の派生開発プロセス

顧客などからの変更要求を仕様まで詳細化し「変更要求仕様書」を作成することで、変更要求とそれに対する変更仕様の繋がり(トレーサビリティ）が確立できる。また、「変更要求仕様書」には変更の理由・目的を記述するため、要求を誤解したまま後フェーズに進んでしまうリスクが軽減できる。

さらに「変更要求トレーサビリティマトリクス」によってそれぞれの変更仕様に係わる対象(ドキュメント、ソースコード)が一覧で俯瞰できるため、変更対象の考え漏れに気付きやすくなる。

「変更設計書」は、従来の改造仕様書と同等の内容ではあるが、テンプレート化して記述すべき内容が整理されたことで属人性が排除できた。

3.3. XDDP 導入の期待効果

XDDP は、派生開発時の有効なプロセス改善手段になるであろうと感じていたが、実際に XDDP を導入するとどのような効果が得られるのか、試行を始める前に以下のような８つの仮説(期待効果)を考えてみた。その上で、実際に想定した仮説が正しいかどうかを試行プロジェクトによって確認することとした。

(1)　大きな品質向上を望めるのではないか

(2)　情報共有可能なレベルでの変更要求の明確化が実現できるのではないか

(3)　システム全体にわたり変更の影響範囲を確定できるのではないか

(4)　変更漏れを無くすことができるのではないか

(5)　上流工程強化により、後戻り作業を削減することができるのではないか
(6)　テストケースの作成、テスト作業の効率化を図れるのではないか
(7)　組織的なレビュー、情報の共有を徹底できるのではないか
(8)　原本開発者でなくても派生開発が可能になるのではないか

3.4. 試行プロジェクトによる評価

3.4.1. 品質の評価

XDDP の適用に際しては、「本当に品質改善できるのか？」「余計な作業が増えて工数が増大するのではないか？」といった懸念もあり、まずは試行してその効果を確認することとなった。 試行を実施した 2008 年当時、いくつかの類似プロジェクトが平行して進行しており、それらの中から XDDP を試行するプロジェクトをいくつか選定し、適用しなかったプロジェクトと品質面での比較を行って評価した。

開発の設計フェーズごとに不良発生率(件/KSLOC)を XDDP 適用・非適用のプロジェクトで比較した結果を図 A-2-4 に示す。

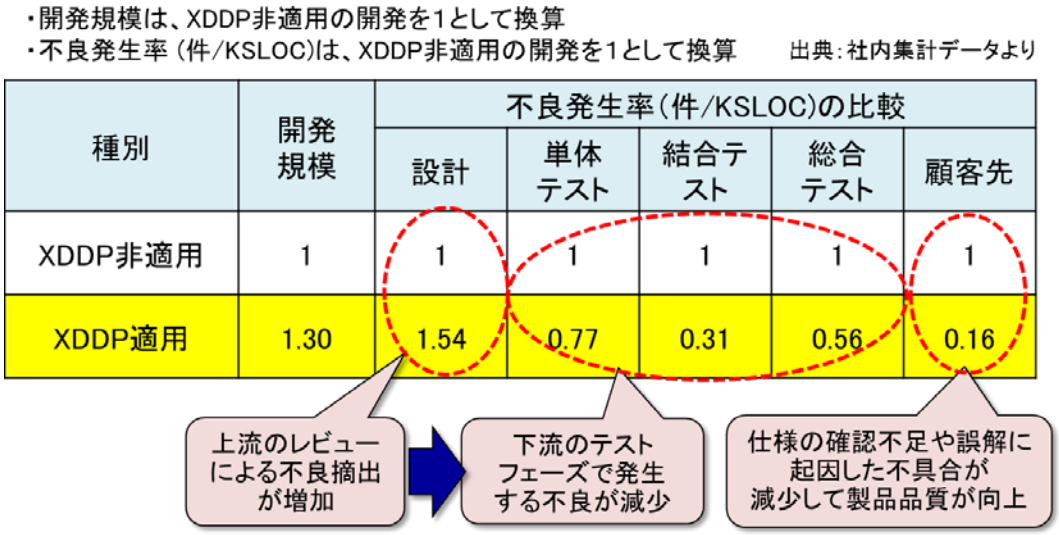

図 A-2-4　試行時の品質評価結果

設計フェーズでは、3 点セットの成果物レビューにより XDDP 適用プロジェクトの不良摘出率が多くなっているが、その後のテストフェーズでは XDDP 適用プロジェクトの不良摘出率が少なくなっていることがわかる。特に仕様の確認不足や誤解に起因した顧客先での不具合が大きく減少し、製品の品質が大きく向上したことがわかった。

3.4.2. 工数の評価

試行時の工数面での評価結果を図 A-2-5 に示す。

XDDP を適用したプロジェクトの計画時点の各設計フェーズの予測工数と終了後の実績工数を比較した。上流の設計フェーズでは、XDDP の 3 点セットの成果物の作成とそのレビューなどの作業が増えたことにより計画時に比べて 40%程度の工数増となった。しかしながら、品質向上による後戻り作業の減少などの理由により、以降の製作・テストフェーズでの工数が減少したため、全体工数は若干減少した結果が得られた。

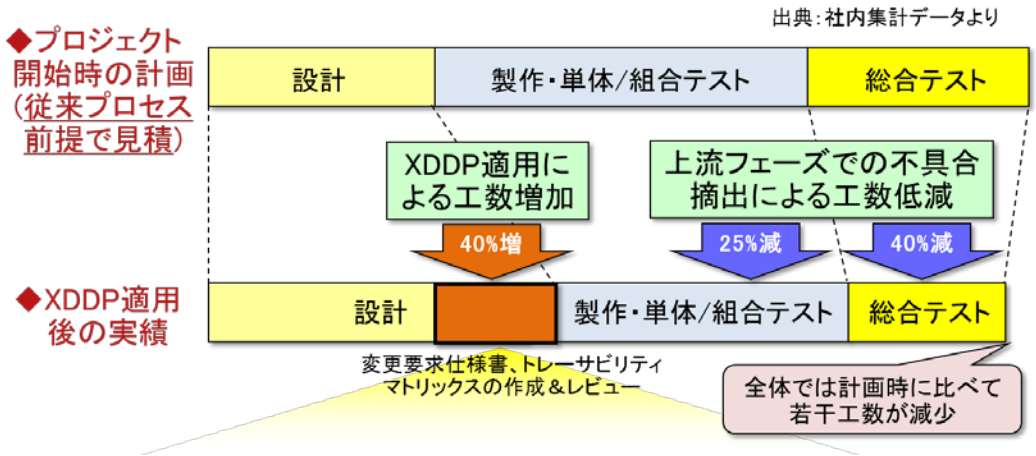

図 A-2-5　試行時の工数評価結果

試行した結果より、派生開発における品質面、工数面での XDDP の効果が確認できたため、実際の現場に広く導入していくことを決定した。

4. 取組みの実施、及び実施上の問題、対策、工夫

XDDP の試行は、組込みソフトの分野を対象にして実施したが、インフラシステム、産業システム、など分野を問わずに派生開発に適用できると考え、「派生開発時は必ず XDDP を適用する」こととした。同社幹部の理解も得ることができ、トップダウンで社内に迅速に徹底することができた。

ボトムアップで適用拡大するために、生産技術部門による説明会の実施や 3 点セット成果物のテンプレートの提供、適用ガイドの作成、などの支援を実施した。また、XDDP を適用した部門相互の情報交換の場として、社内フォーラムも開催するなど、XDDP の理解促進とその適用効果を社内に展開した。

一方、XDDP の試行時に下記に示す課題も見つかっていた。

(1)　XDDP の特徴である 3 点セットの成果物は、それぞれが相互に密接に連携している。開発の途中で変更が発生すると、その都度 3 点セットの成果物間のトレーサビリティを維持するための手間がかかっていることがわかった。

(2)　開発期間中は、3 点セットの成果物を複数のプロジェクトメンバーで共有することになるが、特に作成に時間がかかる「変更要求トレーサビリティマトリクス」では、

成果物の共有（競合管理）の仕掛けが必須であることがわかった。

これらの課題を放置したままでは適用拡大の障害になると考え、支援ツール「SagePro/eXM」[4] を独自に開発した。本ツールは、社内用ツールとして同社のイントラネット環境で自由に利用でき、2010 年にリリースを開始した後も継続して機能改善や機能追加を行ってきた。

5. 達成度の評価、取組みの結果

2008 年以降、同社内では分野を問わず XDDP の適用が拡大してきており、これまでに 400 件以上の派生開発に適用されている。

XDDP は、比較的簡易な開発アプローチであることから現場への浸透が早く、試行時の適用効果を定量的に示せたことで、現場リーダが自律的に導入を決断する事例も多かった。

全社展開後の品質面(品質保証部門摘出不良率：件/KSLOC)での評価結果を図 A-2-6 に示す。

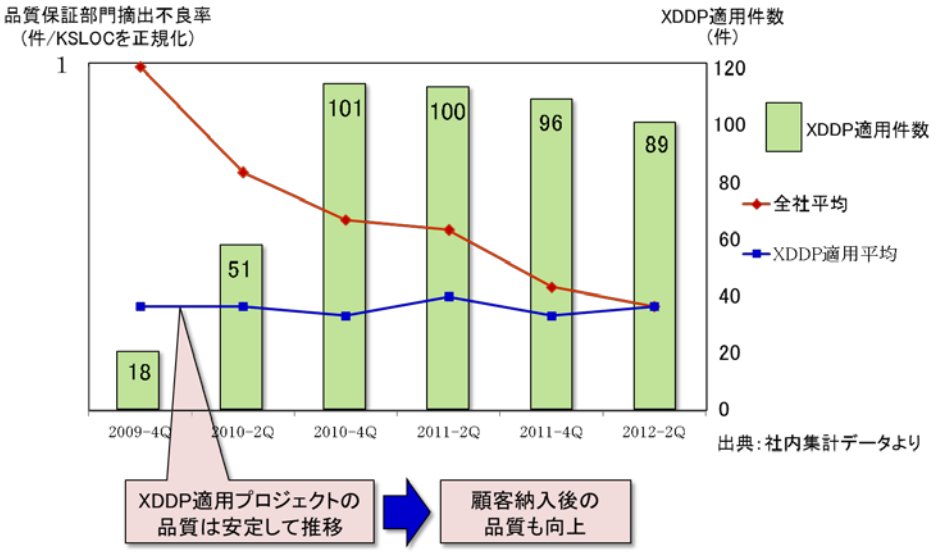

図 A-2-6　全社展開後の品質評価結果

XDDP を適用したプロジェクトの品質は安定しており、XDDP の適用数が増大するにしたがって全社平均の品質も向上してきている。もちろん XDDP の効果だけで全社平均の品質向上が実現できているわけではないが、多少の寄与ができているものと評価している。

全社展開後の開発効率についての評価結果を図 A-2-7 に示す。

ある分野における、XDDP を導入していないプロジェクトと導入したプロジェクトの開発効率(KSLOC/人月)を比較した。

[4] SagePro は、株式会社日立産業制御ソリューションズの登録商標である。

結果を分析してみると、統計的な有意差は見られず、XDDP を初めて適用した際でもそれまでと同等の開発効率を維持できている。このことから、XDDP の適用による開発効率への悪影響(余計な作業が増えて効率が低下する)は懸念に過ぎなかったと判断できる。

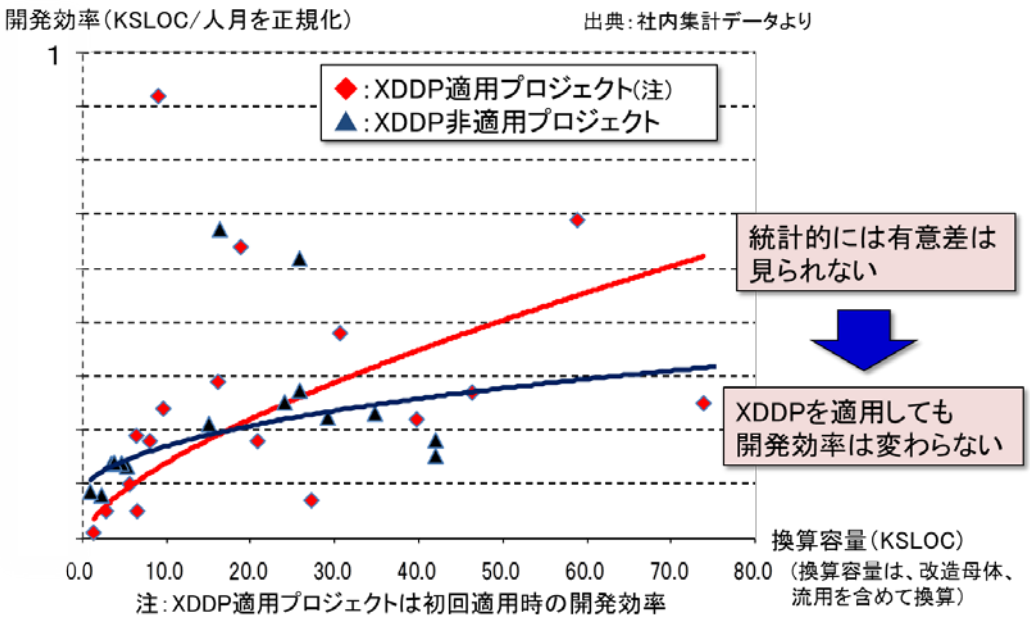

図 A-2-7　全社展開後の開発効率評価結果

XDDP は、派生開発 1 回ごとで完結するプロセスとなっているが、同社では永年にわたって同一の製品を何度も派生開発する事例が多い。このように、同一製品で XDDP を繰り返して適用した場合の開発効率について評価した結果を図 A-2-8 に、同一製品適用時の開発フェーズごとの効率評価結果を図 A-2-9 に示す。

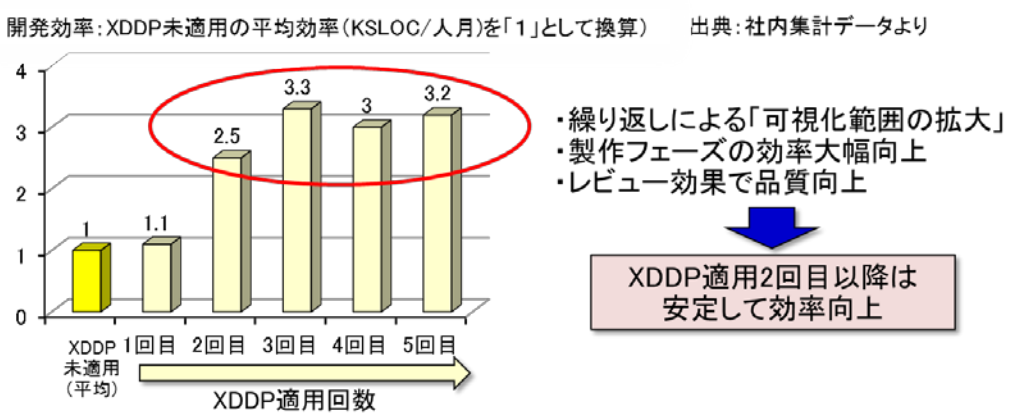

図 A-2-8　同一製品適用時の開発効率評価結果

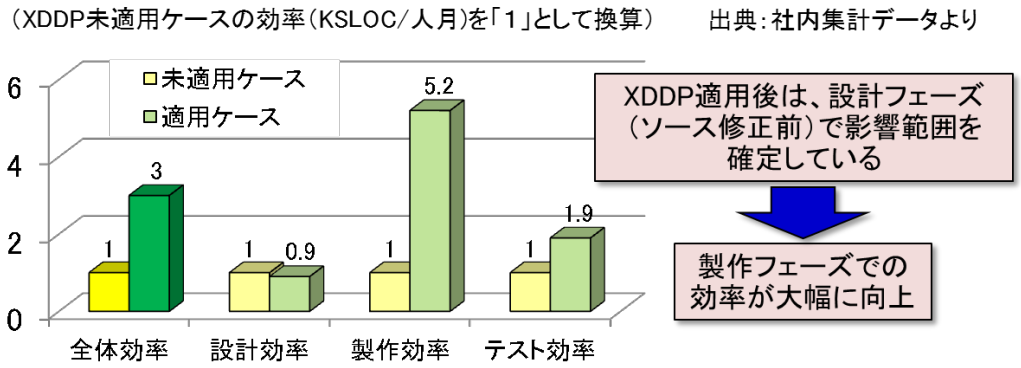

図 A-2-9　同一製品適用時の開発フェーズごとの効率評価結果

繰り返しによる 3 点セット成果物の蓄積によって「可視化範囲の拡大」を図ることができ、製作フェーズの効率向上や品質向上効果も相まって、2 回目以降の派生開発効率が大きく向上していることがわかる。

社内向けの XDDP 適用支援ツール「SagePro/eXM」は、リリース後もレビュー支援機能や変更履歴機能などを順次追加して利便性の向上を図ってきた。 前述したように、同一製品を永年にわたって繰り返し派生開発するケースが多いため、ツールに蓄積された過去の 3 点セット成果物を活用した「変更対象候補を抽出する機能」も新たに追加して、さらなる適用拡大を図っている。

XDDP を導入する前に 8 つの仮説を立てていたが、実際の適用後にこれらを評価した結果を図 A-2-10 に示す。 XDDP を適用した結果、それぞれの仮説に対して、期待した効果を上げていることが実証できたと考える。

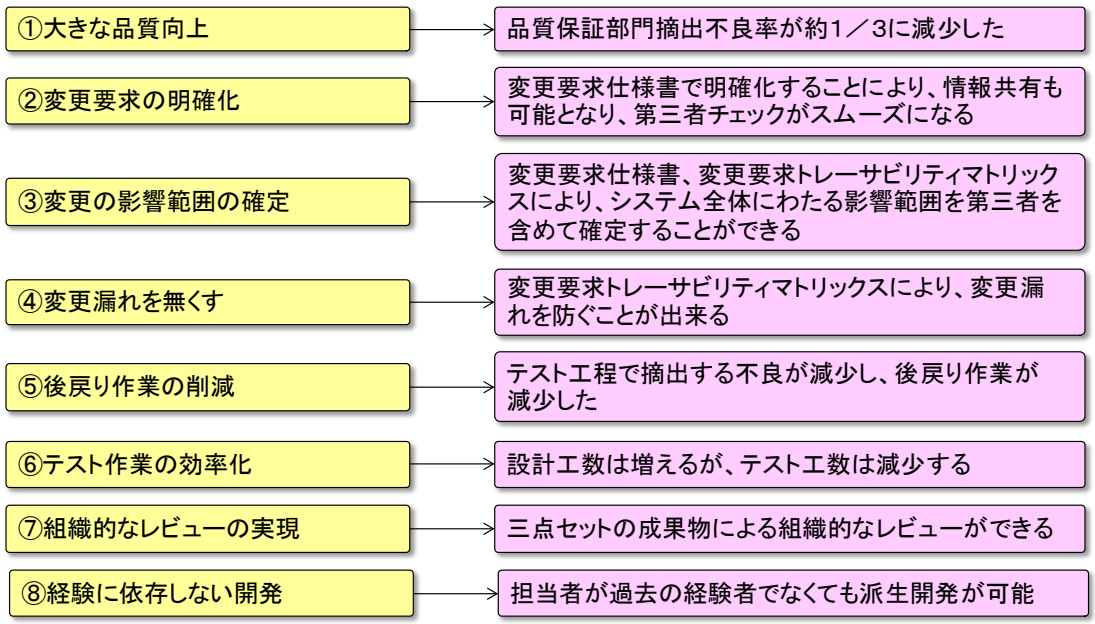

図 A-2-10　8 つの仮説(期待効果)の検証

6. 今後の取組みと考察

以下に今後の課題を示す。

（1）　適用件数が急激に増大した結果、「やらされ感」や「プロセス形骸化」が発生している懸念がある。適用現場でのヒアリングなどを通して、XDDP の理解を促進していく必要がある。

（2）　XDDP は、変更要求を正確に行うための非常に有効なアプローチである。
しかしながら、永年にわたる派生開発の繰り返しで歪んでしまったソフトウェアの構造(アーキテクチャ)を正せるわけではない。

将来の派生開発の予想状況などに応じて、リファクタリングの要否や、SPL(ソフトウェアプロダクトライン)への移行、などを検討する必要がある。

本事例では、同社における派生開発プロセス強化の目的で XDDP を導入した経緯とその効果について述べてきたが、これから導入を考えている方にとって多少なりとも参考になれば幸いである。

現時点では、派生開発に着目した開発アプローチは XDDP しか見当たらないが、まだ世の中に十分に普及しているとは言えない。このため、さらなる普及拡大に向けて適用実績や認知度を上げていく必要がある。

派生開発推進協議会は、2010 年に設立された団体で、XDDP 考案者の清水氏を代表として、数多くの有志や企業が参加して、勉強会やカンファレンス、フォーラム開催などの活発な活動を行っている。協議会の各種活動に参加することで、XDDP を新たに導入する場合や

導入後の様々な悩みを議論したり共有したりすることができ、より一層 XDDP の知見を深めることができる。協議会のホームページ（http://www.xddp.jp/）にも有効な情報が数多く掲載されているので是非参照されたい。

参考文献

[1] 清水吉男：「派生開発」を成功させるプロセス改善の技術と極意、技術評論社、2007
[2] 清水吉男：失敗しない派生開発、SoftwarePeople Vol.8　PP.2-69、技術評論社、2006
[3] 伯田誠：ソフトウェアの改造で悩んでいませんか？～派生開発の品質と効率の向上を目指して～、派生開発カンファレンス 2010、2010
[4] 櫻庭恒一郎：派生開発特有の問題を解決！～XDDP 導入経緯と成果～、第 25 回 JASA ET セミナー、2011
[5] 櫻庭恒一郎：XDDP 導入による派生開発プロセス改善とその効果、ソフトウェアプロセス改善カンファレンス 2012、2012

A－3

組込み系の利用品質における「HMI[1]品質メトリクス」開発と適用事例[2]

1. 概要

本編では、製品開発における「HMI 品質メトリクス」の適用について、株式会社 U'eyes Design（以下、同社とする）の事例を紹介する。

システムが大規模・複雑化し、機器やサービスの連携度が高まる今日、システムを構成するソフトウェアの高信頼化の重要度が増している。最近では、ユーザが直接ソフトウェアを操作する機会が多く、ソフトウェア開発の総合的な品質特性（ISO/IEC 9126）の使用性を強化する意味合いで、利用する人の視点から商品の価値をチェックする「利用品質」の向上が求められている。従来は、ユーザビリティの専門家による定性的な分析・診断や、人間工学的なガイドラインを参考に属人的ノウハウに頼った品質向上対策が主流であったが、本事例は、設計の目標値となり得る、客観的な定量指標の開発を目指し、開発現場への適用を試みたものである。こうした定量指標の一群は「HMI 品質メトリクス[1]」として提唱され、システム開発における利用品質とシステム品質とを繋ぐ役割となることが期待されている。

2. HMI ユーザビリティの重要性

ユーザビリティという概念が日本にもたらされてから四半世紀余り。この間、我が国においては、一部の工業デザイナーを起点とし、主にコンピュータ画面内のデザイン設計の考察ポイント[2]として、ユーザビリティは少しずつ産業界に浸透してきた。操作性に起因する事件、事故やパソコン、スマートフォンを代表とする IT 機器の市場拡大などの要因によって、近年ではユーザビリティの重要性が高まり[3]、多くの開発エンジニアに知られる用語となっている。

特に、自動車業界においては、より安全で、安心かつ快適に運転ができるようにコクピット内の運転操作、および付随するカーナビ・エアコン等の操作性向上が重視され、大型の液晶ディスプレイが車内に搭載される最近では、ユーザに訴求できるポイントとしてユーザビリティの重要性はさらに高まっている[4]。

---

[1] HMI（Human Machine Interface）主に操作パネルや表示装置などの機械（マシン）を主対象とし、人間を中心に据える視点を重視した研究領域（ユーザビリティハンドブック　共立出版　2007　p.8）

[2] 事例提供：株式会社 U'eyes Design
中嶋　智輝 氏、真行寺　由郎 氏、山中　佑也 氏、山田　知幸 氏、上田　涼裕 氏、鱗原　晴彦 氏

これまでのユーザビリティは、要素技術、研究開発的な取り組みを主とし、概念や定義、技法開発、プロジェクト開発への導入・実施事例を学ぶ啓発期にあった。最新の動向としては、商品競争力の一つとして、消費者の受容性を高める「ユーザエクスペリエンス（以下、UX と略す）」が注目される中、ユーザビリティは UX の基礎として認識されるようになっている。さらに、ユーザビリティは、デザイン的、研究的な取り組みの枠組みを超え、「利用品質の向上」という商品全体、システム全体の品質構成の要素として「品質の説明責任[5]」を果たす役割を期待されている。あらゆる商品の実開発になくてはならない基礎的な要件としての役割を担うべく、普及期を迎えたと言ってよい。

普及期に必要な取り組みは、

① より高い次元を目指す専門性の確立や目標設定

② より品質を安定させるための標準化活動や､市場を広げビジネスとして成長するための仕組み作り

である。①の代表例としては、特定非営利法人 人間中心設計推進機構による「人間中心設計専門家」の認定活動[6]や、IPA/SEC と一般社団法人コンピュータソフトウェア協会（CSAJ）の連携により第三者評価を意識した活動「PSQ 認証制度[7]」が始動した。

一方で、②の標準化活動やビジネス上の仕組み作りはこれからという状況であり、ユーザビリティの良し悪しに関する基準を明確化し、専門的なノウハウや知見を一般の開発担当者が扱えるツールとして整備することに加え、プロジェクトマネジメントの管理対象（工数確保、予算獲得）として採用する等の取り組みが必要である。

本事例では、同社が組込みシステム開発企業に対して支援してきた、ユーザビリティを改善し、利用品質を高めるノウハウを、定量的な判断指標へと変換し、HMI の品質を開発の上流工程から診断でき、かつ開発プロセス全体にフィードバックするための方法と、その運用事例を紹介する。

3. HMI 品質メトリクス開発概要

3.1. 開発背景

従来、組込みシステム開発でユーザビリティを向上させるには、V 字開発工程で統合テストを終えた動作可能なシステムでユーザビリティテストを実施し、改善を図ってきた。しかしながら、近年は、開発規模が肥大化・複雑化する一方で、開発期間は短縮されるために、ソフトウェアの不具合の発生頻度も高く、後工程でユーザビリティを改善することが不可能となっている（図 A-3-1）。また、迅速かつ適応的に開発する手法のアジャイル開発を採用すれば、動作可能なシステムを活用し、開発の前段階から開発者自身でユーザビリティ評価を実践し、効率的な改善が可能になる。しかし、大規模かつ複雑化したシステムにおいて、システム制約、ソフトウェア制約とユーザビリティを一括してマネジメントできる開発者は存在せず、上流工程で適正にユーザビリティ評価を実践することは極めて難しい。

ところで、開発の上流工程からHCD活動[3]を導入することでユーザビリティが向上することは、ISO 9241-210[4]、ISO 9241-11[5] やIPA/SEC提唱の超上流プロセス[8]等でも示されている。さらに、最新の規格情報としては、ソフトウェア品質モデルの体系としてISO/IEC 25000　SQuaRE[6]シリーズが紹介され、ユーザビリティはソフトウェア品質の一部として認知されるまでになっている。

本事例でも、ユーザビリティの向上は、品質特性の一つとして HMI 品質の向上と言い換えて記述する。

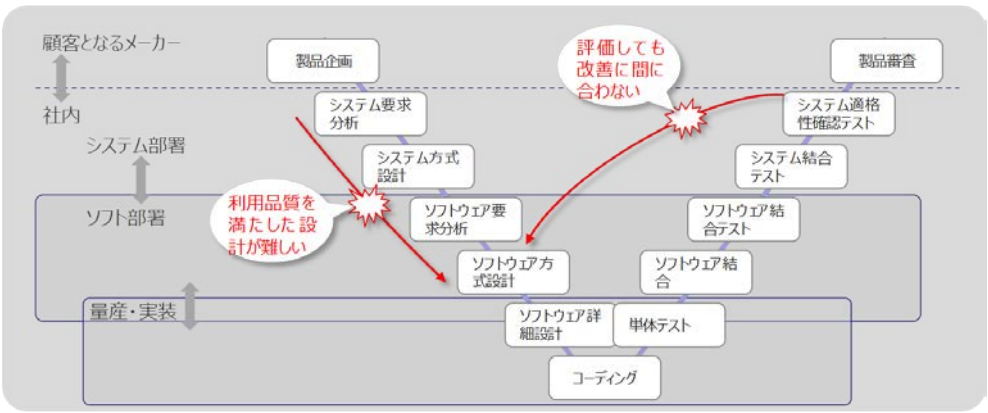

図 A-3-1　組込みシステム開発におけるユーザビリティ評価の難しさ

さて、前述したように昨今の開発が抱える背景的な課題を解決する手段の一つとして定量化指標に活路を求め、利用品質を構成する下記の 5 つのレイヤー[9]の中で、未着手の HMI 品質のためのメトリクス開発に着目することとした。

（1）　経営レイヤー（経営品質：KPI 等）
（2）　サービスレイヤー（サービス品質：SLA 等）
（3）　プロダクトレイヤー（HMI 品質：概念なし）
（4）　システムレイヤー（システム品質：既存）
（5）　ソフトウェアレイヤー（ソフトウェア品質：既存）

---

[3] ユーザビリティの高い人工物をいかにして作り出すか、という取り組み全般（戦略、企画、仕様、設計、品質保証、販売・サービス・メンテナンス、長期モニタリングに至る一連の開発活動）のこと。HCD は情報技術に関する人間工学として提唱（by Shackel,B.）された言葉。ユーザビリティを扱う領域を広げて捉える表現（ISO13407 などで採用）出典：ユーザビリティハンドブック　共立出版（2007）

[4] ISO 9241-210：Ergonomics of human-system interaction -- Part 210: Human-centred design for interactive systems（2010）

[5] ISO 9241-11：Ergonomic requirements for office work with visual display terminals. -- Part 11: Guidance on Usability (1998)

[6] Systems and software Quality Requirements and Evaluation

3.2. 開発目的

上流工程での仕様開発を対象に、ユーザビリティの知識を持たない設計者が、客観的にユーザビリティの良し悪しを判断できるような HMI メトリクス（以後、メトリクスと表記）を開発し、HMI 品質を確保することを目的とする。

3.3. メトリクスの対象範囲

メトリクスでユーザビリティの問題を計測する対象範囲は、ISO 9241-11 で定義されたユーザビリティの内で Small Usability と称される「指定された目標を達成するために用いられる際の有効さ、効率」を対象範囲とし、満足度は対象外とする。また、開発システムが車載機器であるため多重課題をもった「車載機器を利用する状況」を対象とし、ドライバーと車載機器とのインタラクションで発生する認知、判断、操作性を範囲（図 A-3-2）とした。なお、満足度は主観的要因の影響が大きく、客観的な定量指標を求めるメトリクスには時期尚早であると判断したため、現時点では対象外とした。

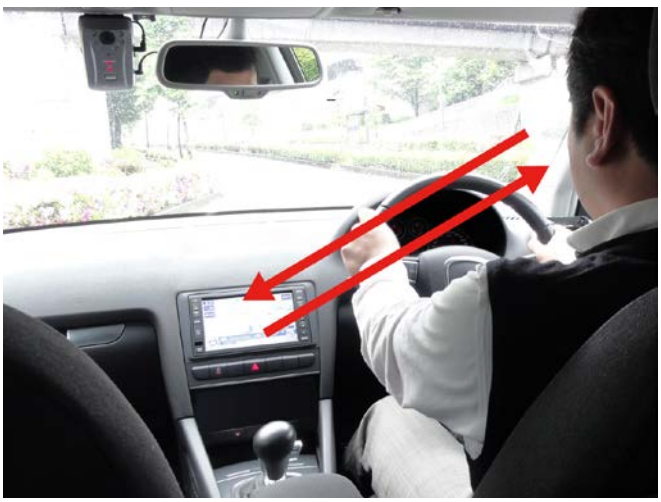

図 A-3-2　ドライバーと車載機器のインタラクション

3.4. 開発プロセス

HMI 品質メトリクス開発の全容は、図 A-3-3 の 4 つのステップで構成される。

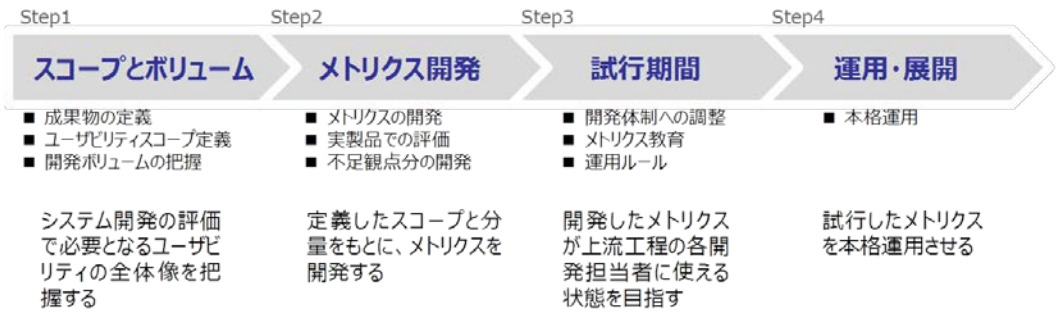

図 A-3-3　HMI 品質メトリクスの開発プロセス

Step1 は、システム開発の上流工程でユーザビリティ評価に必要となる項目の全体像を把握するため、ユーザビリティ評価項目のスコープ定義と必要分量のボリュームを把握する。スコープは開発するシステムによって決定し、ボリュームはシステム開発の分業体制に応じて必要量を把握する。

Step2 は、Step1 で定義したスコープと分量をもとに、メトリクスを開発する。また、開発したメトリクスは実機評価を通じて、ユーザビリティ専門家による評価結果とほぼ同等の効果を示すことを確認する。その際、専門家による指摘事項のうち、機能選定もしくはタスクシナリオの選定上 Step1 で定義したスコープから漏れたユーザビリティ観点がある場合、網羅性を高める意味で観点を補填するメトリクス開発を別途行う。

Step3 は、Step2 で開発したメトリクスを上流工程の各開発担当者が運用できることを目指す。その際、各開発担当者へのメトリクス導入教育、メトリクス自体の有効性向上、開発体制への導入マネジメント、運用ルール等を整備して、メトリクスの試行をし、運用上の問題点を抽出、改善を繰り返す。

Step4 は、Step3 で試行したメトリクスを本格運用させる。

4. HMI 品質メトリクスの開発

本活動における、メトリクスの開発方法は図 A-3-4 のように、「フェーズ 1：メトリクス候補の準備」「フェーズ 2：メトリクス化の判断」「フェーズ 3：メトリクスの作成」「フェーズ 4：メトリクスの妥当性評価」の 4 つのフェーズにて実施し、作成したメトリクスを評価しては改善を繰り返すスタイルを採用している。

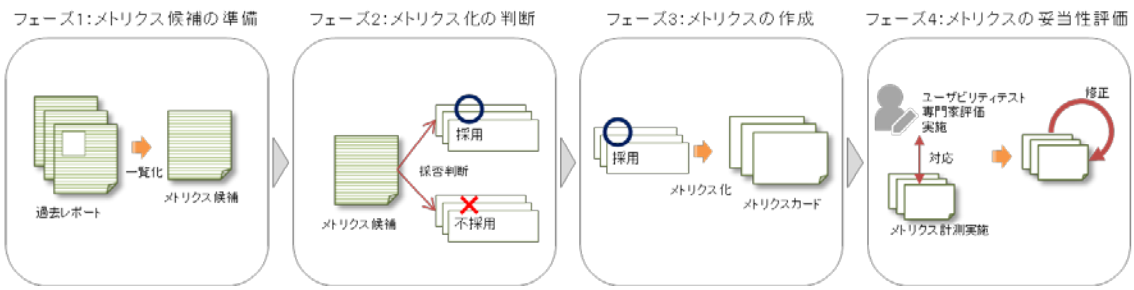

図 A-3-4　HMI 品質メトリクスの開発プロセス

4.1. フェーズ 1：メトリクス候補の準備

フェーズ 1 では、過去に実施したユーザビリティ評価業務から抽出され蓄積した膨大なユーザビリティ上の問題点を整理し、問題点に含まれるユーザビリティの観点を網羅的に抽出する。その観点をメトリクス候補と呼び、メトリクス候補の全体像を確認するため、ユーザビリティ領域の論文[10]、文献[11][12]内で既に公開されている主たるユーザビリティチェックリスト群と関連付ける。

4.2. フェーズ 2：メトリクス化の判断

フェーズ 1 で抽出したメトリクス候補を分析し、メトリクスとして採用するかを判断する。採用基準は、前述の「Small Usability の範囲内」で、「ユーザモデルや利用状況に影響を受けにくいユーザビリティ上の問題点」であることとする。また、メトリクス化（4.3 参照）を検討する上で「人間工学規格等、対応する知見や実験データ」など第三者への説明責任を果たせる根拠を明示できるようにすることが必要である。さらに「開発現場での利用価値があるか」といった当該開発部署の文化に沿った組織上の受容性もメトリクス化の判断基準として重要である。

4.3. フェーズ 3：メトリクスの作成

フェーズ 2 で判断した候補について、メトリクス化するため、次の 4 項目を検討している。各項目の業務上の呼称は当該組織に馴染む言葉で良いと考える。

- 品質特性：品質としての目的や効果の想定
- 属　　性：計測する対象の設定、明確化
- 尺　　度：計測値の判断基準、評価基準
- 測定方法：計測値の数え方、単位

これらを検討した結果を、メトリクスカードに記述する。なお、客観的な結果が出るような計測方法を検討し、測定方法には基準値として推奨とする目標値や、それ以下を不可とする限界値等を設定した。

4.3.1. メトリクスの構成

開発したメトリクスは 74 点である。これらの内訳が図 A-3-5 で、UI の要素ごとに画面遷移・操作フロー系が 4 点、画面情報量が 6 点、画面レイアウトが 13 点、機能の有無が 8 点、グラフィック・文言が 43 点である。

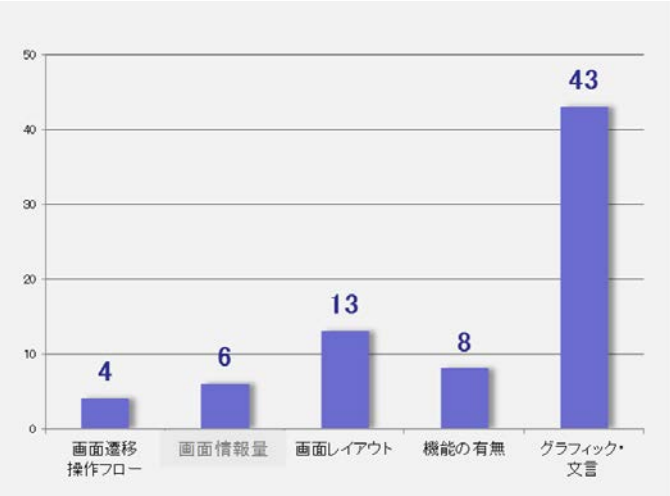

図 A-3-5　メトリクスの構成とその数

画面遷移・操作フロー系とは、画面の順序性や関連性がユーザにとって自然な流れになっているものか、必要十分な操作量であるかを計測するメトリクスである。例えば、画面の遷移量や遷移パターンを対象に計測をする。

画面情報量とは、画面内に表示する情報の量が適正量で、煩雑さや分かりにくさを感じることがないかを計測するメトリクスである。例えば、アイコンや文字量、選択項目数を対象に計測するものである。

画面レイアウトとは、画面内の提示情報を整理して並べ、作業順序に沿った配置としているかを確認するメトリクスである。例えば、数字の位置揃え、メニュー情報の配置順序を対象に確認するものである。

機能の有無とは、表示や操作をする上で最低限必要な機能が揃っているかを計測するメトリクスである。例えば、進捗状態を表す部品、戻るスイッチやキャンセルスイッチの設置を対象に計測をする。

グラフィック・文言とは、画面上の文字やアイコンなどのビジュアル要素が見やすく、読みやすいか、用語は適切かを確認するメトリクスである。例えば、押下と分かるスイッチ形状の適正、押下の可能性を認知できるカラーリング（可能時の色と不可時のトーンダウン色の見分け）の適正、利用時を想定した可読文字サイズ、ミスタッチを防止するスイッチの大きさ、動作を促す文言付きスイッチの仕様の適正、などを対象に計測をする。

4.3.2. メトリクスの計測事例

メトリクスによる計測事例として、操作ボタンの使いやすさに関する事例を紹介する。このメトリクスの考え方は、操作ボタンのレイアウトに対して、意味あるいは内容をもとにした配列と、物理的な配列との一致のし方を評価するものである。

図 A-3-6 には、【意味による配列】と【物理的な配列】を一致させたものと不一致なものを示す。この状況を測定するメトリクスとして、意味による配列のボタン間の寸法と、物理的な配列のボタン間の寸法の差を尺度にしている。意味による配列を明確にするために、ボタン間の間隔寸法をどの程度にすれば良いか測定することができる。

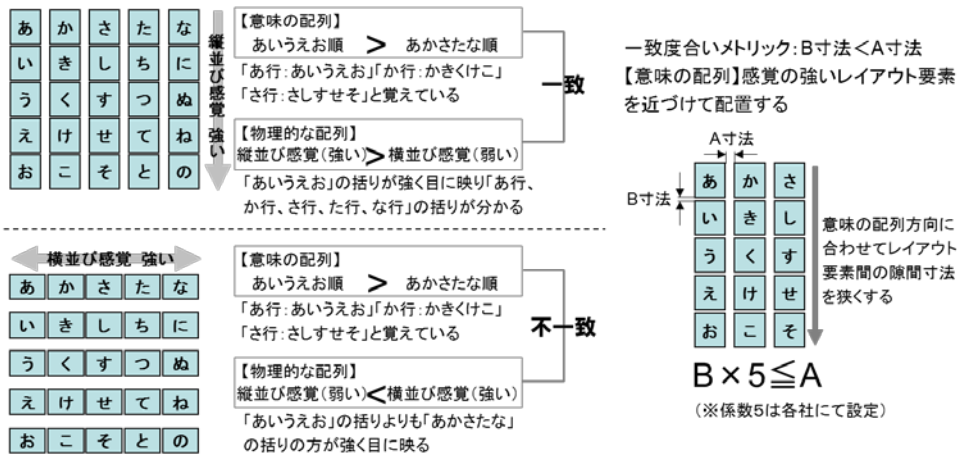

図 A-3-6　情報レイアウトにおける「意味の配列」と「物理的な配列」の一致の度合い

4.4. フェーズ 4：メトリクスの妥当性評価

開発したメトリクスが的確にユーザビリティの問題を抽出できるか、開発者がメトリクスを適切に扱えるか確認をするため、妥当性と運用性の効果検証を実施した。妥当性は、最新車載機器を対象にユーザビリティの問題抽出を行い、メトリクスによる抽出結果とユーザビリティ専門家による抽出結果との一致度を確認する。運用性は、仕様開発者を対象に仕様書での評価を実施し、適切にメトリクスを活用できることを確認する。

4.4.1. ユーザビリティ専門家による評価

最新の車載機器に搭載した機能のうち、ナビ、オーディオ、エアコンの標準的な機能のある 95 画面を対象に、メトリクスによる問題抽出とユーザビリティ専門家による問題抽出を実施した。また、メトリクスはこの時点で開発済み 68 点を使用し、メトリクスによる問題抽出はメトリクスの開発担当者が実施した。

その結果、専門家の抽出した問題と一致したメトリクスは 89%であった（図 A-3-7）。したがって、開発した大半のメトリクスが専門家と同等のユーザビリティの問題が的確に抽出できることを確認した。また残りの 11%は、不足するユーザビリティ観点として、新たにメトリクスを開発した。

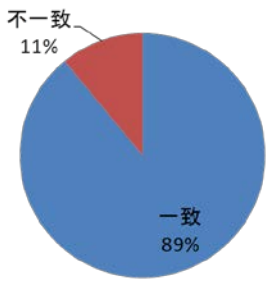

図 A-3-7　専門家とメトリクスの問題抽出の一致度

4.4.2. 仕様開発者による評価

8 名の仕様開発者および、開発中の仕様書 16 画面を対象に、メトリクスを用いたユーザビリティの問題抽出を行い、その運用性評価を実施した。評価対象として、仕様用に活用が見込める 14 点のメトリクスを選出した。

その結果が表 A-3-1 で、16 画面で 12 点のメトリクスが使用され、23 件のユーザビリティの問題を抽出した。よって、仕様開発者が 1 画面あたり 1.4 件のユーザビリティ問題を抽出できたことになる。このうち、仕様開発者がメトリクスを的確に使ってユーザビリティの問題を抽出できたメトリクスは 9 点であった。多くのメトリクスが期待通りの使い方で正確に問題を抽出しており、上流の設計工程での運用性を確認できた。残りの 3 点は、メトリクスの計測対象の不明確さ、計測方法の複雑さ（手順書を読み理解が必要）であったため、ユーザビリティ評価結果の的確性に欠けていた。なお、これらにはメトリクスの計測対象や計測方法に修正を加えた。また、2 点のメトリクスは、評価対象の画面に適用できる事例がなか

ったため、使用されなかった。

表 A-3-1　仕様開発者の運用性評価結果

評価画面数	16画面
仕様用メトリクス	14点
的確に使用可能	9点
使用に問題あり	3点
評価対象なし	2点
問題抽出件数	23件

5. HMI 品質メトリクスの運用

5.1. 製品開発プロセスにおける、従来の評価手法の課題

まず、そもそも、ユーザビリティを評価するということは、製品開発プロセスにおいてどのような意味を持つのであろうか。一つは、製品開発プロセス内での HMI 品質向上であり、製品に潜むユーザビリティ上の問題を、開発・設計する中で如何にゼロに近づけるか、どのように改善していくかという目的であり、これを目的にした評価を形成的評価と呼ぶ。

もう一方は、発注した製品を受け入れたり、製造した製品を出荷したりする際、要求を満たすに足る HMI 品質になっているかなどを確認（もしくは認証）する目的であり、こういったことを目的にした評価を総括的評価と呼ぶ。

製品開発プロセスでユーザビリティを評価する上で、この二つの目的が存在するということは、あまり広く認識されておらず、これらを混同することによって、期待する効果が得られなかったり、ユーザビリティ評価は実開発プロセスにあまり活用できないとの誤解を招いたりしている現状がある。

5.2. 形成的評価の留意点と主な手法の課題

形成的評価では、製品設計中に、どういった設計ポイントがユーザビリティ上の問題になっているのか、どのように改善すればユーザビリティ上の問題がなくなるのか、設計者自身が把握し改善することが目的である。

ウォーターフォールモデルや V 字モデルの開発を採用しているのであれば、次工程に移る前に、その工程で決めるべき設計情報にユーザビリティ上の問題点が潜んでいないかどうかを設計者自身で確認し、その工程内において改善できることが望ましい。

しかしながら、要件開発時点やシステム設計、ソフトウェア設計時点では、操作性を確認できるモックアップやプロトタイプを用意することが困難なため、ユーザビリティテストを

実施するには、少々工夫が必要である。

また､専門家評価を実施するためにはユーザビリティ専門家を複数人確保する必要があり、ユーザビリティ専門家が評価対象の設計を熟知していないと、各工程のドキュメントだけからユーザビリティ上の問題を発見することが困難となるため、ドキュメントを一旦エンドユーザとのインタラクションが分かるような情報に翻訳する必要が出てくる。

5.3. 総括的評価の留意点と主な手法の課題

総括的評価では、製品完成後（もしくは設計完了後）に、発注者が製品を受け入れたり、製造者が出荷したりする際、HMI 品質が確保されているか確認することが目的である。

こういった確認する目的があるため、発注時や設計前に、予め要件として一定の HMI 品質を定義しておくことが必要である。

しかしながら、予め要件として HMI 品質を定義しておくこと自体、製品設計プロセスの文化では稀であることや、HMI 品質自体が、定量化しにくく、客観的に判断できるようにするには、現状の評価手法に工夫が必要となってくる。

例えば、ユーザビリティテストを総括的評価として採用した場合、どういったタスクを、どの程度のユーザが達成でき、どの程度まで迷いや滞りを許容するのか、予め設定しておかないと、要件として確認できない。また、専門家評価を採用するのであれば、どのようなポイントでどの程度ユーザビリティ上の問題が発見されたか、予め設定し、チェックリスト等で評価する観点を共有しなければ、客観的な判断基準たりえない。

5.4. メトリクスの必要性

前述のように、主な評価手法には、ユーザビリティ専門家が介在しないとユーザビリティ上の問題を把握できなかったり、要件として定量的に判断できなかったりといった課題があった。また、システムが大規模・複雑化した今日、仕様書の段階でソフトウエア品質を確認するためには、一定期間内に効率よくチェックしなければならず、専門家以外の多数の設計者により判断できる指標が必要である。さらに、設計変更が生じた場合にも、メトリクスの導入により、その変更が HMI 品質に著しく影響を及ぼす範囲や程度を予測できれば、設計変更自体の是非や方針を判断することが可能になると期待される。

特に、本取り組みでは、ユーザビリティの専門的知識を有しない設計者自身が、ユーザビリティ専門家の知見に頼ることなく、ユーザビリティを定量的に判断できることを目指し、HMI 品質を測定できるようにするためメトリクスを開発した。対象のユーザビリティを定量的に把握し、その改善方法を得ることを目的に評価手法の開発を目指した結果として、メトリクスだけで改善できる部分を明確にすることができた。

このことは同時に、ユーザビリティ専門家の総合的な判断が必要となる部分が浮き彫りになるという副次的な成果も見えている。すなわち､健康診断においては専門家である医師が、あらゆる検査データを多角的に確認し、かつ、医師の経験や知見を基に、的確な分析・診断をくだしていく。HMI 品質に対しても、この検査表のデータと同様の信頼性あるメトリクス

が定着し、健康診断と同じくらい受診が必須となるような、社会的な仕組みが確立していくことが望ましいと考える。

6. HMI 品質メトリクスの運用環境

6.1. メトリクス管理ツール

上流の開発工程でユーザビリティを確保するために開発したツールは、メトリクスカード、ガイドライン、手順書、解説書のセット図 A-3-8 である。詳細は以下の通りである。

・ メトリクスカード

ユーザビリティの知識がない開発者がメトリクスを理解できるよう、品質特性、属性、尺度、測定方法を記したものである。品質特性には、ユーザビリティの観点と品質としての目的や効果を記した。属性は、画面遷移や画面上のどこをチェックするか、対象物を記した。尺度は、計測値の判断基準、評価基準を記した。測定方法には、計測により算出した結果の表現の仕方、数値、そして必要に応じて条件等を記した。

・ ガイドライン

メトリクスで抽出したユーザビリティの問題に対し、開発者が改善できるようにするため、ユーザビリティの考え方、改善方針を記したものである。ユーザビリティの考え方は、ユーザビリティの観点やそれに対する知見を記した。改善方針には、いくつかの解決パターンと改善事例を記した。

・ 手順書

メトリクスカードでの計測対象や計測方法が複雑である、または計測に厳密さが必要な場合に限り、詳細な計測方法の説明として手順書を用意した。

・ 解説書

メトリクスで抽出できるユーザビリティ上の問題点の重要性と、改善の必要性を説明するため、解説書を用意した。解説書には、メトリクスが有するユーザビリティ観点の説明、根拠やユーザビリティの原理原則となる資料、ユーザビリティの考え方を記した。

図 A-3-8　メトリクス管理ツール

6.2. HMI 品質チェックシート

開発製品の HMI 品質を管理するため、HMI 品質チェックシートを作成した（図 A-3-9）。

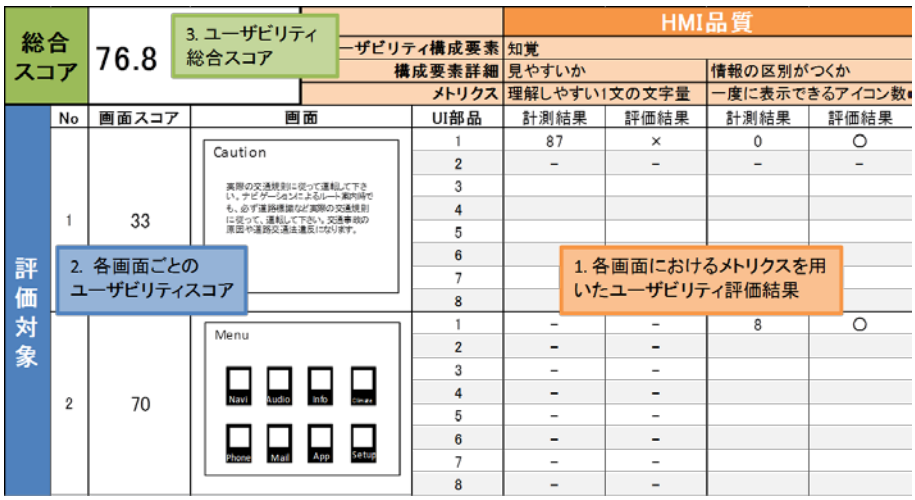

図 A-3-9　HMI 品質チェックシート

チェックシートの縦方向は、開発する各画面とその画面で使用する UI 部品、横方向は、HMI 品質を構成するメトリクスを並べ、メトリクスに該当する画面の UI 部品の評価結果を記入する。これは開発管理者、および責任者が HMI 品質を管理するため、次の 3 点が確認できる。

（1）各画面におけるメトリクスを用いたユーザビリティ評価結果
（2）画面ごとのユーザビリティのスコア
（3）ユーザビリティ総合スコア

（1）は、開発担当者がメトリクスを用いて各画面の UI 部品ごとに、ユーザビリティ評価した結果をチェックシートに記入し、ユーザビリティの良し悪しの詳細な一元管理ができる。

（2）は、（1）で記入した結果から、開発管理者が画面ごとのユーザビリティ結果のスコアを確認する。そのスコアと各画面の問題点をもとに、改善する画面の UI 部品を検討し、HMI 品質の調整をすることができる。

（3）は、（1）で記入した結果から、開発責任者が開発製品全体のユーザビリティの総合スコアを簡易的に確認する。これをもとに、メトリクスの構成ごとのスコアをレーダーチャート（図 A-3-10）にすることで、HMI 品質の度合いが分かり、HMI 品質の目標確認と開発全体にフィードバックをすることができる。

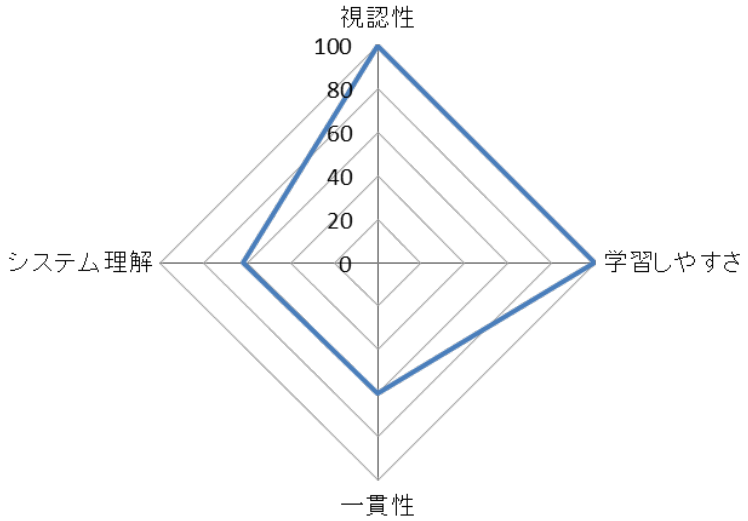

図 A-3-10　メトリクス構成ごとのスコア

6.3. HMI 品質メトリクス自動計測ツール

前述のように、現段階では、ユーザインタフェースの構成要素を手動で計測をするため、製品の設計者は、普段の製品設計に加えてメトリクスの計測作業を行う必要があるため業務負荷が高くなる。そのため、製品の設計者がその設計業務において、設計作業中に自動的にメトリクスを計測するツールが必要だと考えた。

そこで、同社は資本関係にある株式会社エイチアイと共同で、UI 設計ツール「exbens® UI Conductor（図 A-3-11）へメトリクス計測機能を実装する取り組みを行っている。実験的にメトリクスが計測できる 10 の測定機能を組み込み、現在、有用性を確認中である。

図 A-3-11　UI 設計ツール

http://www.hicorp.co.jp/ja/prux_uic.html より

7. 重点課題

7.1. メトリクスの網羅性

前述のように、メトリクスを開発した目的は、上流工程でユーザビリティの知識を持たない設計者が、客観的にユーザビリティの良し悪しを判断できるようにすることであった。しかし、目的を達成するために、メトリクスを開発し、効果検証を行ってきたが、開発済みのメトリクスでSmall Usabilityの範囲を網羅的に計測できているのか判断する必要がある。

そこで、世間一般で広く用いられており信頼性がある評価指標（sHEM[10]、70 デザイン項目[11]、SIDE[12]）と、同社の知見から、Small Usability の構成要素を網羅的に見渡せる、合計 274 項目のチェックリストを作成した。参照した評価指標は、車載機器に特化した内容ではなかったため、今回の目的に合わせてアレンジしている。

作成済みのメトリクスで Small Usability の範囲を網羅的に計測できているかを確認するため、作成したチェックリストに、開発済みのメトリクスを対応付けた。

結果、全 274 項目のチェックリストに対して、99 項目のメトリクスが対応付けられた。メトリクスが不足している項目については、今後、メトリクスおよびガイドラインを拡充していく予定である。

7.2. 開発者による Small Usability の理解促進

従来の開発では、ユーザビリティエンジニアではない各設計担当が、それぞれの裁量で任された曖昧な基準で仕様を作成していた。そのため、担当ごとに判断がバラつき、HMI 品質を確実に確保することができないという課題があった。それを改善するためには、メトリクスの運用以前に、Small Usability の全体像を体系的に把握し、Small Usability が十分に理解されるよう啓発活動の重要性を同社では再認識されている。

7.3. 実際の開発工程への導入

これまでは、仕様担当者が試験的にメトリクスを使用していたが、今後は、実際の開発工程へ導入し、実際のスピード感で実用に耐えられるようにカスタマイズや自動化を試みる必要がある。

8. 展望

メトリクスは、HMI 品質を示す値なので、継続的に精度の向上を目指す必要がある。時代と共に変わるユーザ像、技術革新による機能やサービスの変化、新たな操作デバイスの登場によって､測定すべき対象やメトリクスの許容値､採用して良い値のレンジには差が生じる。ユーザタイプやシナリオも含めたユースケースが異なる場合のメトリクス値の変化なども、マネジメントできるようになるのが理想である。

本事例で述べたように、HMI品質の向上に有効なメトリクスを考案するには、それなりの

経験とセンスが必要である。現状では、HCD-Net[7]認定 人間中心設計専門家など、ユーザビリティ評価やUI/UXデザイン開発に従事している人材が適任だが、システム開発の知識や経験も併せ持つ人材、もしくはチームの存在が求められると考える。

[7] 特定非営利活動法人　人間中心設計推進機構

参考文献

[1] Haruhiko Urokohara, Naotake Hirasawa：Managing HMI quality in embedded system development；HCI International 2013

[2] 野呂影勇ほか：図説エルゴノミクス　3. ソフトウエア編，pp.182-253

[3] 鱗原晴彦ほか：IPA Forum 2010～日本の元気を IT で～
ソフトウェア・エンジニアリング　プログラム　パネルディスカッション「第三者検証出来ること，出来ないこと～利用者を意識した品質の確保に向けて～」

[4] カープレイの搭載 http://monoist.atmarkit.co.jp/mn/articles/1403/03/news125.html

[5] 独立行政法人情報処理推進機構（IPA/SEC）ソフトウェア信頼性の見える化 （ソフトウェアグループ）http://www.ipa.go.jp/sec/software/index.html

[6] 人間中心設計（HCD）専門家 資格認定制度 http://www.hcdnet.org/certified/

[7] 一般社団法人コンピュータソフトウェア協会（CSAJ）:PSQ 認証制度;
http://www.csaj.jp/psq/

[8] 独立行政法人　情報処理推進機構　ソフトウエア・エンジニアリング・センター編：経営者が参画する要求品質の確保～超上流から攻める IT 化の勘どころ～, pp.83-110, 2006

[9] 平沢尚毅，鱗原晴彦：【HCD における品質／HCD の応用】
「人間中心設計における品質アプローチ」（１）－HCD とシステム品質との接点 （２）－HMI 品質への取り組み；特定非営利活動法人人間中心設計推進機構　研究発表会 2014

[10] 黒須正明，杉崎昌盛，松浦幸代：問題発見効率の高いユーザビリティ評価法 - 1. 構造化ヒューリス ティック評価法の提案 - ，ヒューマンインタフェースシンポジウム，pp.481-488，1997

[11] 山岡俊樹：ヒューマンデザインテクノロジー入門―新しい論理的なデザイン，製品開発方法，森北出版，2003

[12] 山岡俊樹，藤原義久，鈴木一重：構造化ユーザインタフェースの設計と評価―わかりやすい操作画面をつくるための 32 項目，共立出版，2000

A－4

要件定義段階における信頼性向上の取り組み事例紹介[1]

1. 概要

本編は、システム開発で手法・ツールを有効に活用したビッグローブ株式会社[2](以下、同社とする)の事例である。同社では、コンシューマ向けのサービスから法人顧客の受託システムまで様々なシステム開発を手がけている。サービスの品質向上やコスト削減のため従来からいろいろな手法やツールを試行錯誤しながら活用しており、その中で見られた問題点や効果的だった手法を紹介する。

2. はじめに

Web サービスやクラウドサービスの開発においては、低予算・短納期でのスモールスタートを前提としてプロジェクトを開始する場合が非常に多い。同社では、最短では数日～数週間のサイクルでアプリケーション開発を行うこともしばしばであり、常に数十本単位のプロジェクトが同時並行で動いている。このため、上流工程において性能や機能に関する要件を詳細に詰めるだけの十分な時間が確保できない場合が生じる。

また、顧客からの受託開発案件やカスタマイズの注文においては、顧客企業内でも関係部署間での調整がなかなか取れず、下流工程での急な要求変更による開発や品質検証への負担が無視できない状態になっていた。

アジャイル開発手法で要件を詰めていこうとした場合でも、業務契約上ベンダに完成義務が無くなり、最終的なコストが読みにくくなる不安が生じる[1]。また、納期ぎりぎりまで要件が固まらず十分な品質検証が行えないままリリースせざるを得ない状況も発生していた。

上記の問題を軽減するために同社で実践して効果的だった要件分析の効率化手法を紹介する。

3. 効率化活動の背景

様々なプロジェクトで生じていたトラブルを分析したところ、上流工程における原因の作り込み方に多くの共通性が見られた。特に、機能要件・性能要件・セキュリティ要件・運用要件の決め方では、プロジェクトの性質や規模に依らず類似の問題が見て取れた。機能や性

---

[1] 事例提供：ビッグローブ株式会社　ビジネスサービス事業部 ／BIGLOBE-CSIRT
橋田 幸浩 氏

[2] ビッグローブ株式会社は、2014 年 4 月 1 日付けで NEC ビッグローブ株式会社より社名変更した。

能などの要素に依らず共通性が高かった事象は、以下の事柄である。

① 要件毎の重要度や優先順位が適切に決められない
② 定量的な基準を定めたいが、どう定めればよいか分からない
③ 要件は定めたが、曖昧性が高いか非現実的な内容となってしまい、基準として形骸化してしまう

様々なモデリングやフレームワークの教科書に沿って要件定義方法やプロセスの標準化[2] [3]も図ったが、担当者毎のスキルやリテラシの違い、会社や部門毎の文化や開発手法・技術の違いもあり、完全に標準化することは事実上不可能だった。顧客企業でもリリースの直前に諸事情が急変し、機能変更を求められることもしばしばである。

受託開発であれ自主事業であれ、一般的に「インターネットサービスの強みは、思いついたらすぐにシステムを変更できることである」と思われていることを考慮すると、プロセスや手法を定型化して教科書通りのプロセスでのみ進めることを望むのは現実的ではない。しかし、実際には課金系や業務系のように従来型システムと何も変わらない複雑な要素も混在しているので、「間違えたら直せば良い」では済まされないのも現実である。

そこで、従来のモデリング手法を踏襲しつつも、文化や技術・スキル・手法の違いに左右されることがない要件定義方法を模索することにした。

4. 詳細分析

要件定義が円滑に進んでいない事例や、障害の原因が要件定義そのものにある事例を観察・分析したところ、表 A-4-1 の傾向が見られた。障害を起こしやすい案件に特に顕著だったのが、下記の事柄である。

(1) 機能要件の中身について

「データＡとデータＢを結合する」といったプログラム的な実装内容は細かく定めていたが、それぞれの処理が「エンドユーザにとってどのような結果をもたらすべき処理なのか」を開発者や運用者が正しく理解できていなかった。

(2) 重要度や優先順位について

オーソドックスな手法として「必須 or 任意」や「高中低」など単純化させた指標で整理していたが、すべての要求が「必須」かつ「最重要」となり、順位付けが形骸化していた。

(3) 性能要件の中身について

性能要件は最も技術的知識と経験則が求められる要素の一つで、適切な要件を設計できる者が不足し、出荷判定も形骸化していた。

(4) セキュリティ要件の中身について

担当者自身のスキルとリテラシによる解釈や対策の振れ幅が非常に大きく、要件自

体が形骸化し、チェックも形骸化していた。セキュリティ品質に関する「よく使われるが、実は最も困る要件」の例を表 A-4-2 に紹介する。

(5)　運用要件の中身について

手順書やマニュアルが「書いた本人にしか理解できない状態」で、文書化されていない行間の作業が多く埋もれていたが、適切に引き継がれていなかった。

表 A-4-1　多く見て取れたアクター毎の悩みと問題点の具体例

	アクター毎の悩み			結果として発生する問題	陥りがちな落とし穴	具体事例	左記の問題点	改善策
	顧客（発注者）視点	開発者視点	運用者視点					
機能要件	デザインのことは分かるがシステムやプログラムの実装についてはどう決めれば良いか分からない	・デザインなど表面的な議論に時間が費やされがちで、実装を決める時間が不足する ・実装に関する説明で工数を浪費する	納期とコストの圧縮が優先され、運用者の作業のしやすさは考慮の外に置かれがちになる	・納期や費用が不足する ・運用ミスを誘発しやすい ・運用時に余計な工数がかかる	すべての機能の優先度や重要度が同じになり取捨選択ができなくなる	全ての要件が必須・最重要となっているが、予算と納期に見合わない。	ステークホルダーの真意が見えなくなり、後から「そこまでの意図はなかった」と言われる	wish, should, must で要件の重みを決める
	優先順位や必要性の重み付けが判断しきれず、プロの意見を聞きたい	すべての要件が必須・最優先となり、何から着手すべきか分からない		発注自体が遅れて着手も遅れるが、納期の後ろ倒しは認めてもらえない。	実装要件は決めたが、実現したかった機能を満たせていないことに気づけない	システムＸとシステムＹを繋ぎ､データＺを受け渡す､という実装は決めていたが､「何のために」は誰も理解できていなかった。	・細かな実装を詰めようとするあまり、真の目的を見失う。 ・真の目的を見失っていても、「要件は満たしているのだから」と関係者が問題に気づけない	・ユースケースを起点とした要件定義を行う ・実装要件に対してもなぜ×３分析を行う
性能要件	システムに関する知識は乏しいのでどのように定めればよいか分からない	・現実的基準値を定めるための調整工数が浪費される ・要件に合わせた言語の選定やコード設計がしづらい	システムに関する知識は乏しいのでどのように定めればよいか分からない	・出来上がりが実用に耐えられない状態になる ・出荷判定が形骸化する	出来上がった機能の実測値をそのまま出荷判定の合格基準あるいは仕様としてしまう	要件＝標準サーバ x2 台で満たせる性能等になる	出来上がったサービスが実用に耐えられない	ユースケースを起点としてどれくらいの性能なら実用に耐えられるのかの下限を決める。
	性能は高ければ高いほど良いが、想定するコストや納期に対し妥当と言える基準が分からない	非現実的な要件が提示される	性能は高ければ高いほど良いが、想定するコストや納期に対し妥当と言える基準が分からない	・過剰品質となりコストの無駄が増える ・要件や出荷判定が形骸化し、問題発生時に責任の所在について揉め事になりやすい	非現実的な性能が要件として提示されてしまう	廉価なサーバ１台で 1000 トランザクション/sec 処理できること	出荷判定が形骸化する	wish, should, must で要件の重みを決める
セキュリティ要件	親会社や省庁・関連団体が定める基準は全て満たして欲しい	省庁や団体が定めるチェックリストは膨大すぎてコストや納期に見合わない	全作業のキーパンチを記録せよなど、現実味の乏しい要件が提示される	要件自体が形骸化し、チェックも形骸化する	過剰あるいは過少な要件になる	表 A-4-2 参照		
	具体的にどう定義したら良いか自分たちでは分からない	どのような実装とチェックをすれば良いか判断できない	―	・要件の過不足が生じる ・調整に時間を費やし工数・工期を圧迫する	曖昧で開発者のスキルやリテラシに大きく依存する定義になる			
運用要件	設計段階では細かなことまでは決められない （決めたくない）	設計段階では細かなことまでは決められない （決めたくない）	何をどうしたらよいか分からないまま障害対応を頼まれる	発注者や開発者にヒアリングしながら作業しなければならず、効率が下がる一方になる。	ドキュメント作成工数を削るあまり、マニュアル類が不足する	サーバのプロセスをリフレッシュするとは書いてあるが、誰が／どうやって／どのプロセスをといったことが書かれていない	都度、分かる人に確認しながら作業するため、効率が上がらない（分業の意味がなくなる）	開発と運用を組織的に分け、受入プロセスを分離する
	運用設計ノウハウが無く何をどう定めればよいか分からない	運用者向けの機能やマニュアルを作りこむ余力がないので人的にカバーして欲しい	限られたリソースで複雑かつ間違いやすい作業を押し付けられ、工数やコストを圧迫される	・運用のミスが起こりやすくなる ・考慮不足が生じ、ユーザにクレームされるまで誰も間違いに気づけない	マニュアルに書かれていない「行間の作業」がある	・データＡとデータＢをマージする、と手順には書いてあるが、具体的にどの項目をどうマージするのかは書いていない。 ・マニュアルの手直しを依頼しても工数不足等で対応してもらえない	・マニュアルが「書いた本人にしか分からない内容」になる ・手順の漏れや間違いに実務者が気づけない	受入プロセスの中で、実務者自身が自律的にマニュアルを改善できるよう、分担と権限を定める

表 A-4-2　セキュリティ要件の好ましくない例と改善例

分類	好ましくなかった定義例	実際に見られた問題点	定義の改善例	備考
曖昧性が強く、開発者のリテラシやスキルに依存しやすいパターン	OWASP TOP10[3]に対応していること	TOP10 の脆弱性にのみ対処され、それ以外の脆弱性が放置されそうになった	所定のセキュリティ診断ツールを用い、出荷時点における既知の脆弱性の問題が無いことを定量的に証明すること	著名なツールであれば OWASP TOP10 には確実に対応している。最新パターンで検証すれば「出荷時点における既知の脆弱性」には対応できている、と言える。
	セキュリティの脆弱性が無いこと	具体的な対処の内容は誰も理解しておらず、チェックが形骸化していた		
	安全なパスワードを利用すること	担当者によっては典型的な悪例パターン（123abc など）が用いられていたが、「自分はこれで十分複雑と思った」という状態だった	パスワードは大文字/小文字/数字/記号のうち３種以上の混合で８文字以上　かつ　アカウントと同じ文字列は用いないこと。	ツールで検証すれば脆弱なパスワードを用いているかどうかも検証可能な場合がある。
	無用なデバッグモードは使わないこと	トラブルが生じた場合、何が必要になるか分からないので、最も強いデバッグモードにした。必要なのでそれで良いと思った。	○○モジュールのデバッグモー ドは、本番環境ではレベル◎ 以上を使わないこと（must）	システムの適切な構成管理と変更管理が不可欠利用可能なポートやプロトコル、あるいは著名ツールのデバックモードはツールが検出してくれる場合が多い。
	不要なポート/プロトコルは閉じておくこと	何を使うか分からないので、 全てのポートとプロトコルを 解放した。必要なのでそれで良いと思った。	・利用するポートとプロトコルは具体的に明示すること(must) ・明示的に利用するポートとプロトコル以外は解放しないこと(must)	
過剰要件になり無駄なコストが生じやすいパターン	経産省・総務書・IPA が定める各種ガイドラインに準拠すること	チェックリストの確認だけで数人月の工数を消費することになり、現実味が無くなってしまった。	所定のセキュリティ診断ツールを用い、出荷時点における既知の脆弱性の問題が無いことを定量的に証明すること	指定するツールがどのガイドラインにどこまで準拠しているかはベンダやメーカが示してくれることが多い。ただし、内容は要求者も正しく理解しておく必要がある。
	PCI-DSS や FISC に対応していること	共用設備の全通信ログの開示など、パブリッククラウドでは対応不可能な要求が出された		
	セキュリティ診断ツールによって脆弱性が一切検知されないこと	・ツールからの通信を遮断してしまえば良い、と考えた ・過剰検知された項目や技術的に対処不可能な事柄まで対処を求められた		過剰検知や誤検知および対処方法を正しく判断・理解できる人材やサポートは必要
対処が限定され過少な対処となるパターン	XSS と SQL インジェクションに対応すること	左記以外の脆弱性が放置されそうになった		
	<script>alert(“警告”)</script>と入力しても反応しないようにすること	テキストマッチングで左記のパターンについてのみチェックされていた		
	入力フォームのあるページは脆弱性の対処を行うこと	hidden パラメータやクエリストリングスの改ざんや cookie の脆弱性などが放置されそうになった		
	個人情報を取り扱う場合は脆弱性の対処をすること	個人情報を扱わないページの脆弱性が放置されそうになった		
		全体を通しての改善ポイント	細かな定義を多数並べるよりも、「所定のセキュリティ診断ツールを用い、既知の脆弱性が残存していないことを定量的・技術的に証明すること」という定義を用いた方が、抜け漏れや属人性を排除しやすかった	

[3] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
OWASP（Open Web Application Security Project）で公開している、よく狙われる脆弱性のトップ 10。

5. 対策

以上の事柄を踏まえて、次のような対策を講じた。

5.1. 機能要件定義方法の工夫

■　上流工程でのなぜ×3 分析

機能要件定義には、プログラム的な実装内容を最初から指定することは原則排除し、エンドユーザ視点で何をするか（ユースケース視点）で定義を作成するようにした。同社でフォレンジックサーバ [4]を試験的に導入する際に実際に作成した例を表 A-4-3 に示す。この要件定義書はそのまま RFP として各ベンダへ提示させていただき、先方の担当者からも非常に好評だった。

左方の欄には「ユーザが何をしたいのか」を具体的に書き、右方に行くに連れて徐々に実装へブレイクダウンさせていく。一番左は、例えば「購入」「検索」「商品情報の管理」などといった「ユーザ行動の分類」を入れていく。要件区分は最後に整理すれば良く、初期段階では重複や矛盾等は気にせず埋めていけば良い。

この手法自体はモデリングに関する論文で多数紹介されていると思うので、本事例では書式や書き方の詳しい説明は省き、内容のチェック方法について紹介する。

もし、表の左方に「システム X で、データ A をバッチ Z へ引き渡す」といった実装論が書かれていた場合は、「エンドユーザの何を実現するためにこの処理が要るのか」の説明を設計担当者へ求める。もし設計担当者が説明できず「処理に必要と言われたので」といった回答が出てきた場合、真の要件を見失っているか見誤っていると判断し、要件の再確認を指示する。

WBS などのモデリング手法では元来「要件＝ユースケースで記述すべき」とされてはいるのだが、技術者のみで表を埋めているとどうしても「実装」を最初に書きたがる傾向が出てくるので、注意が必要である。もし左方の欄に正しくユースケースが記載されていれば、「何のために」を問う必要が無いので確認の時間もゼロ化できる。このように、上流工程においても「なぜ×3 分析」を行いやすくすることで、設計担当者が真の要件を正しく理解できているかを計ることが容易になった。

[4] ネットワークを流れるすべての通信データを記録・保存・解析するためのサーバ。

表 A-4-3　要件と対応方法のサンプル

No	要件区分1	要件区分2	要件区分3	要件	対応方法			判定
					対応区分	対応機能	説明	
025	機能要件	キャプチャ能力	暗号化通信対応	https や ssh 等、暗号化された通信もキャプチャできる	must	Yes		○
026			言語対応	キャプチャ対象は、日本語ほか、マルチ言語に対応している	must	Yes		○
027			再現力	サーバ側・エンドユーザ（ブラウザ/端末）側双方の動作（元の通信）が100%再現できる	must	Yes（条件付き）	製品はコネクション単位で再現するため、100%の再現はできませんが、HTTPコネクションをテキストベースで再現することが可能です。	○
028				実際のブラウザでどう見えたか？というレベルで（視覚的に）再現できる。	wish	Yes（条件付き）	同上	○
029				キャプチャしたデータを日時やプロトコル等の規定項目以外に、データに含まれる任意のキーワードで自由に検索できる	must	△	添付データ、暗号データについては、検索できません。また、検索にはパケットデータがシステム内部に必要となります。	△
030				プロトコルヘッダを含む全てのパケットをキャプチャできる	must	Yes		○
031				データ部（たとえばHTTP Response Bodyなど）のキャプチャ可能文字数（データサイズ）に制限が無い	should	Yes	ペイロードをブラウザで再現するのではなく、利用クライアントにペイロードを保存します。	○
032				キャプチャ対象毎に、データ部も全てキャプチャするのか、通信元/先と特定項目のみを取得するのか、といった指定ができる	wish	No		×
033				cookieの中身もキャプチャできる	wish	Yes		○
034				マルチエンコードに対応できる（EUC,UTF8,SJIS,UTF16）	wish	Yes	エンコードはブラウザが行うため、管理端末のブラウザの能力に依存します。	○
035				JIS2004拡張文字の入力（キャプチャ/検索）に対応できる	wish	Yes	文字コードをHEXで入力することで対応可能です。	○
036			統計分析	キャプチャしたデータの発生頻度やエラー率などを統計分析できる	wish	Yes（条件付き）	L2～4まで統計解析可能です。HTTPエラーコードは解析しません。また、エラーパケットも解析しません。	△
037				キャプチャした結果をリアルタイムに分析し、所定の条件に合致した場合はアラームが出せる（予め指定した所定のコマンドをcallできる）	wish	No		×
038				キャプチャ内容の傾向を自動的に学習し、特異点が発生したら自動でアラームが出る	wish	No	トラフィック量に関しては自動学習可能です。	×
039				保管データは暗号化される	wish	Yes	NIKSUN内部ではNIKSUNオリジナルフォーマットで保存。また、NIKSUNデータ部分はSFSファイルシステム部分に保存しており、	△
040			キャプチャ対応プロトコル	IPv6にも対応している	should	Yes		○
041				tcp, udp, http, smtp, snmp, ssh 等々、プロトコル等に制約がない	must	Yes		○
042				キャプチャ取得対象とするプロトコル、IPアドレス、ファイル名を案件毎に細かく指定できる	must	Yes	フィルタにてキャプチャデータを指定可能です。	○
043								
044		閲覧・解析能力	解析	https や ssh 等、暗号化された通信を復号した状態で解析できる	must	Yes（条件付き）	HTTPSはサーバの公開鍵を製品に登録することでデータ再現が可能です。	◎
045			検索（絞込み）	任意の文字列指定により検索が行えること	must	Yes（条件付き）	Web、メール等の添付データ内は検索できません。	△

5.2. 重要度と優先順位付けの工夫

■　感覚として分かりやすい指標の利用

重要度のランク付けは「高・中・低」や「1・2・3」のような数値的優先順位ではなく、表 A-4-4 のような「感覚的な言葉（must, should, wish）」を指標にした。

英語表記を用いた理由は、日本語で「望ましい」と書くと should と wish との区別が曖昧になりやすいからである。また、要件を「願望」と記すのはビジネス的に違和感がある場面もあったので、意味が正確かつ直感的に伝わりやすい表現として英語を選択した。結果として、数値や高中低に置き換えるよりも、言葉で直接書いた方が読み替えの手間が無い分、確認の効率も向上した。

同じ重要度の中で優先順位を付ける際は、表 A-4-5 のように、高中低といった分類ではなく必ず重複の無い数値順で表現する。重要度や優先順位は費用や難易度で決めるのではなく、まず「サービスとして必要か否か」で判断することで、取捨選択を効率的に行い「すべての優先順が同じで、どれを選択すべきか決められない」あるいは「重要度は高だが必須では無い」という曖昧な定義になる事態が避けやすくなる。

表 A-4-4　重み付け指標の定義

分類	意味	例
must	必須要件 予算や納期に影響を与えてでも これが満たされないとサービスと して使い物にならない事柄	・基本的なセキュリティ対策や 安全措置 ・ECサイトにおける商品表示や 決済機能など
should	推奨要件 予算や納期を勘案した上で 場合によっては妥協可能だが、 満たされないと制限やリスクが 生まれやすい事柄	・自己診断機能などの強固な フェイルセーフ機構など ・ECサイトにおける商品の詳細 情報検索機能や口コミ投稿機能
wish	願望 予算と納期の中で作れれば 嬉しいが、無くても良い事柄	・World Wide でのデータ分散管理 ・画面のカラーグラデーションや アニメーションなどの演出機能など

表 A-4-5　優先順位付けの例

できる限り避けるべき例

分類	優先度	要件（いずれもmust）
○○機能	高	カート機能
	高	同一品再購入機能
	高	履歴確認機能
	中	リコメンド機能
	中	口コミ機能
	中	比較機能
	低	いいね！ボタン
○○機能	…	…
	…	…
	…	…
…	…	…

できる限り目指すべき例

分類	優先度	要件（いずれもmust）
○○機能	1	カート機能
	2	同一品再購入機能
	3	履歴確認機能
	4	リコメンド機能
	5	口コミ機能
	6	比較機能
	7	いいね！ボタン
○○機能	…	…
	…	…
	…	…
…	…	…

5.3. 性能要件定義

■　モジュール単位のスループットではなく、サービスとしての体感性能で定義

性能要件も、機能要件と同様に、目標とする数値を wish, should, must の 3 種で重み付けをするようにした。また、モジュール単体毎の性能ではなく、エンドユーザ視点でのユースケースをベースに策定することにした。例えば、業務システムの Web サーバ内でバッチ処

理を動かす際は、単にバッチ処理のスループットを定めるのではなく、「データベースにｍ人のデータが入っている状態で、同時にｎ人のオペレータが、処理 X のバッチを同時に稼働させた場合に、Z 分以内に終わる（should）」といったように、利用条件・状況を踏まえて現実的かつ必要十分な条件を定められることが理想的である。

5.4. セキュリティ要件定義

■　診断ツールの特性の活用

セキュリティの脆弱性については、2005 年頃から既に AppScan や WebInspect といった診断ツールや、専門業者による診断サービスが広まっていた。セキュリティ診断ツールは、スキャンの設定さえ同じであれば、どんなスキルの持ち主でも同じ結果が得られる利点があるので、解釈や判断において属人性を排除することができる。そこで、「検証に用いるツールやサービスの特性」をそのまま要件定義として活用することにした。

具体的には下記のような一文を要件定義中に含めることにした。

・AppScanSTD もしくはそれに相当するツールを用い、出荷時点における既知の脆弱性が残存していないことを定量的・技術的に証明すること。

「AppScanSTD」の部分は、対象となるシステムやサービスの特性に合わせて適切な物に書き換えれば良い。例えば、プラットフォーム層のみが契約の対象であるならば Retina や Internet Scanner などを、アプリケーション層であれば WebInspect など、システムの特性や関係各社の文化に合ったツールを選択するのが良い。

オープンソース系のフリーウェアは検出能や結果の保証性、解説の分かりやすさ等に個々の特性が強いので、選択時に注意を要する。著名なツールとその検出能のベンチマーク情報は各所で公開されているので、ネットを検索していただきたい[5]。

対して、弊機構や関連省庁が定めているチェックリスト類は敢えて要件定義には用いず、開発者向けの教科書としての位置づけに留めることにした。

5.5. 運用要件定義の工夫

■　受入プロセスの整備

従来は、緊急での対応を優先すべく、特に決まったプロセスを定めず開発側が求めたことをできる限りそのまま受け入れることを優先させていた。しかし、それでは業務の内容や目的を理解できないままの運用となり、開発側の改善も進まない。そのため、業務受入のプロセスを一から再構築することにした。

---

[5] セキュリティ診断ツールのベンチマーク情報例
https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools
https://www.ibm.com/developerworks/community/blogs/paulionescu/entry/benchmarking_web_application_security_scanners10?lang=ja
http://sectooladdict.blogspot.jp/2011/08/commercial-web-application-scanner.html

具体的には、システム運用体制を開発依りの体制(開発も兼任している体制)と運用専門の体制の二段階に分け、後者の受付フローを厳密化させた。

「まず運用を受け入れてから業務を整理していく」のではなく、「最低限の業務定義をしてもらい、受入をしながらブラッシュアップさせ、落着点に至ってから運用を開始する」というプロセスは、普通に考えればごく当たり前のことではある。しかし、実際の開発現場ではまず開発チームの負荷やコストを下げることが優先されがちで、運用設計の精査は後回しにされやすいのも現実である。

役割をあえて組織的に分離することで、「手順書やマニュアルを他人が読んで確認するプロセス」が必然的に発生するため、受入チェックの段階で業務手順の精査と曖昧性の排除が確実に行えるようになった。

5.6. チェック方法の工夫

■ チェックリストですべての項目を「はい」に敢えてさせない

前述した方法を用いて様々な要件を決めたとしても、各項目を一つ一つ読み合わせながら要件定義の妥当性を検証するのは非常に効率が悪い。そのため、確認要素を簡素化したチェックリストを用いるのが常道ではあるが、チェックリストの作り方にもまた十分な注意が必要であった。

チェックリストは、たいていの場合「すべての項目が Yes あるいは対象外にならないと不合格」という設計である。チェックのしやすさを考えれば当然の設計ではあるが、実はここには心理的な落とし穴があった。「すべてのチェック項目が Yes であれば合格」と言われると、担当者によっては「検証もできているし、合格になって当たり前なので、すべて Yes にチェックするか対象外にすれば良い」と考える。つまり、「チェックリストを埋めているだけ」で「内容はチェックしていない」ことが起こりうる。

これを防ぐためには、意図的に「不合格」あるいは「回答不能」になる項目(トラップ項目)を、リーダー（責任者）自身がチェックリストに仕込むことが効果的であった。意地が悪いと笑う読者もおられると思うが、この項目一つへの回答で、開発者が内容を正しく理解して確認しているかを簡単に判別できる。ただし、「相手を信用できないからトラップ項目を仕込む」のではなく、「お互いにチェック漏れや認識違いを防ぐために仕込む」という前提で用いることが重要である。トラップ項目は、リーダー自身がシステムの内容を正しく理解していることの証明にもなり、開発者の緊張感を維持することができる。また、トラップ項目を仕込むためにはチェックリストも「一つでも不合格があれば判定は不合格」とするのではなく、「不合格の項目があっても、適切にリスクが回避できていれば良い」と判断できるよう設計する必要がある。

「対象外」の選択についても、注意が必要である。安易に対象外を選択することを許してしまうと、連鎖的な影響範囲を見落とす可能性が高まる場合がある。レイヤードアーキテク

チャ[6]でアプリケーションを構築する場合、データベース(以下、DB と記す）を操作するための部品（M 層）とサービス(業務)を実行するための部品（C 層)とは独立しているのが常道である。例えば図 A-4-1 に示すような例では、開発者は「今回開発・改造したのは DB を操作する部品ではないので、関係ない」と考え、「DB への影響を考慮したか」という設問において対象外を選択してしまうケースがある。実際にはいずれの場合においても影響は DB に及ぶので、状況次第で DB の障害に繋がり得ることが予想できる。

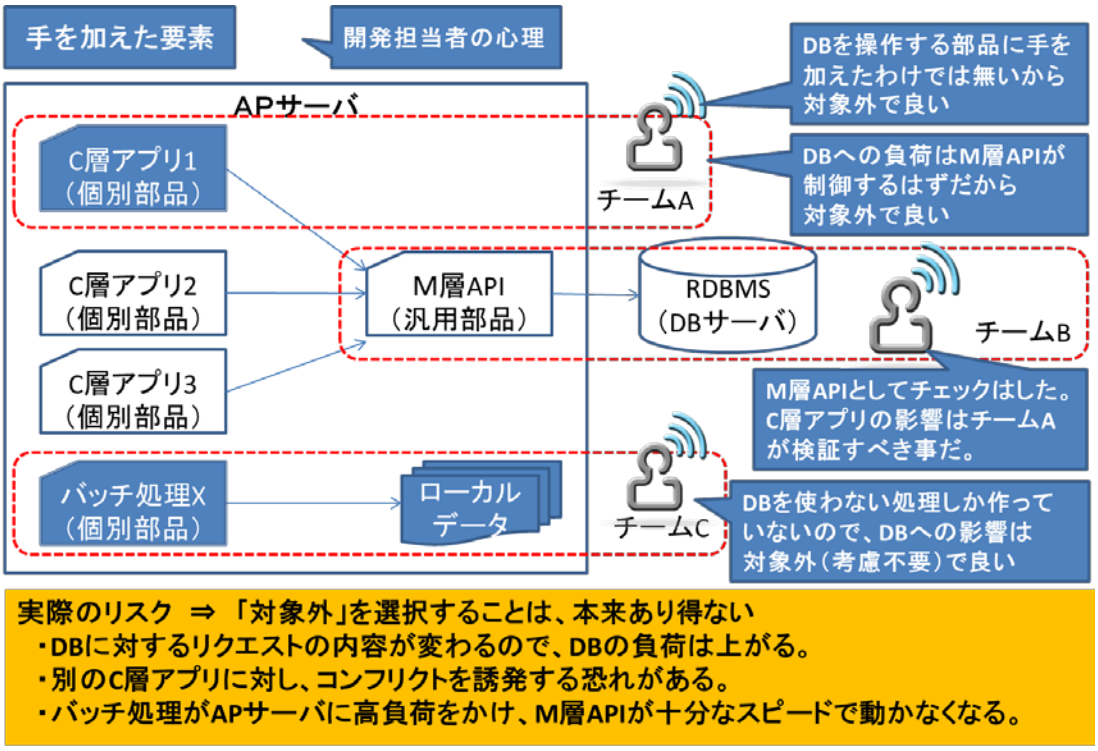

図 A-4-1　「対象外」選択のリスク例

こうした連鎖影響の見落としを予防するためには、チェックリストの各質問で安易に対象外を選べないよう工夫する必要がある。例えば、RDBMS への影響に関する質問において、下記のようにブレイクダウンさせ、選択肢に「対象外」を設けない方法が考えられる。

Q.データベースサーバへの影響を考慮・確認したか？　⇒　□はい □いいえ

- 直接 DB を操作する機能を作っていなかったとしても、AP サーバにかかる負荷など、連鎖的に及ぶ影響を考慮して確認すること。

---

[6] Layered Architecture/ 階層化アーキテクチャ
アプリケーションソフトウェアの流用性や再利用性、拡張性などを高めるために、ソフトウェアを階層構造で分離させる設計方法。

- RDBMS がシステム上に全く存在しない場合は、備考欄にその旨を記載し、影響を考慮すべき要素が無いことを確認した、という意味で「はい」を選択すること。

このように、チェックリストを作成・確認する際は、読み手の心理を先読みして設問を考え、安易に「対象外」を選択させず、また「対象外」が選択されていたとしても読み飛ばさず内容を吟味する必要がある。

チェックリストは「属人性を無くすため、考える必要を無くすように作る物」ではあるが、過剰にそれを進めてしまうと、人は「設問の意味を考える(理解する)こと」自体もやめてしまう。その結果、チェックは形骸化しリストの改善も行われなくなる。チェックリストは「熟練者でも忘れがちな要素を確実に確認するために作る物」であり、「従事者全員が継続的にチェックリストを充実させていく活動にこそ価値がある」と意識することが最も重要であった。

6. 効果

6.1. 機能要件定義法の効果

前述の分類法を用いると、機能要件は表 A-4-6 のように表現できる。直感的に理解しやすい表現を用いたことで、システムの実装やコーディングに関する知識が全く無い人であっても、「ユーザが何をするのか」をイメージすれば良いので、要件の伝達が簡単になった。また、開発担当者も「何を実現するためにその機能を作るのか、どういった諸元が必要なのか」を誤解すること無く、実装との摺り合わせをスムーズに行えるようになった。最短な例では、従来ならば数時間～数日かけて行っていたチェックが 30 分程度で終わり、何日もかけて関係者に説明・交渉していた要件決めを即日決着させることもできた。

表 A-4-6　機能／非機能要件定義の整理例

Item		must	should	wish	前提
セキュリティ	出荷検査時点における既知の脆弱性が無いことを定量的に証明すること	◎	－	－	AppScanを使うこと
購入処理	一度に複数商品を購入できる	5商品 x9点/商品まで	20商品 x15点/商品まで	256商品 x99点/商品まで	いずれの場合でも、一度に購入できる上限金額は選択したカードの限度額までとする(必須)
	カート内商品表示の際「これもお奨め」を表示する	－	◎	－	表示点数は5点までで良い
	カードはVISA,Master,DC,…のn種に対応する	◎	－	－	PCIDSS対応必須
	…				
コンテンツ管理	更新タイミングを指定した単位で制御できる	10分刻み	1分刻み	1秒刻み	
	ダウンロード用のパッケージデータ(tar or zip)も自動更新できる	－	◎	－	
	…				
…	…				

6.2. 性能要件定義法の効果

同様に、前述の分類法で性能要件を現すと、表 A-4-7 のようになる。「ユーザが対象とする機能を利用する際、感覚的に耐えられる限界の性能はどれくらいなのか」を記せば良いので、システムに関する知識が無くとも直感的に理解と定義ができる。また、実装を行う担当者も、最終的なサービスとしての性能イメージが明瞭になり、細かい部品毎の性能要件を連想しやすくなった。更に、全員が「真の合格ライン＝must」と「できる限り目指す目標点＝should ～ wish」とがどこにあるのか誤解なく共有できるようになり、出荷判定が形骸化することも無くなった。

表 A-4-7　性能要件定義の整理例

Item		単位	must	should	wish	前提
最大収容会員数		人	50万	100万	500万	DBは仮想Oracle x 1set
Web性能	トップページ	pv/sec	300	500	2000	仮想最廉価モデルx2 (act/act)
		sec/pv	5sec以下	3sec以下	1sec以下	
	商品詳細ページ	pv/sec	150	200	500	
		sec/pv	8sec以下	5sec以下	1sec以下	
	カート処理	trns/sec	1trns/sec	5trns/sec	10trns/sec	
		sec/trns	20sec/trns以下 （思考遅延除く）	15sec/trns以下 （思考遅延除く）	10sec/trns以下 （思考遅延除く）	
バッチ処理	購入記録のPOS連携処理	record/time	200rec/time	400rec/time	1000rec/time	仮想中級モデルx2 (act/stb)
		sec/time	120sec/time	60sec/time	15sec/time	
	コンテンツ更新処理	files/time	500	1000	2000	
		sec/time	600	300	90	
		Mbyte/time	100	200	400	
	…					

6.3. 検証ツールの特性活用の効果

セキュリティ要件は膨大なチェックリストを用いて対応の可否を確認するなど何日も工数を消費する事もあったが、「AppScan あるいはセキュリティ診断サービスを用いて安全性を検証・確認する」というだけで合意を得られるようになり、顧客への要件確認が AppScan の説明も含めて 1 時間程度で終わらせることが可能となった。著名なツールであれば各種団体やメーカが検査能についてベンチマークレポートを出してくれる事が多いので、検査手法の精度や信頼性の検証を自社で行う工数をほぼゼロ化できる利点もあった。

一部のパートナー会社からは「具体的な検査内容や実装要件と対処法が提示されないと、どうすれば良いか分からないので、この要件定義は困る」という声も寄せられたが、概ねは好評であった。好評な理由として、「要は in/out のサニタイズとヴァリデーションを適切に行っておけば良く、余計な事まで対処する必要も、あれこれチェックリストを埋める手間も無くなり、脆弱性の見落としも防げることになるので助かる」という意見が多かった。結果

として、脆弱性の検証を効率的に行えるだけでなく、この要件への反応と対応方法を観察することで、パートナー会社のスキル計測が容易に行えるようにもなった。

また、同社から納品したシステムにおいて、顧客が機能追加を行うために実際にリリースされるまで数ヶ月の空白期間があり、リリース直前に新たな脆弱性が見つかったことがある。従来であれば検査手法などについて事細かな報告や説明を要するところであったが、ツールのスキャン設定と、検証を行った時点での検出能の限界を顧客へ説明するだけで、「それならば仕方が無い」と即日で落着できた。

6.4. 運用要件定義の効果

受入プロセスを整えたことで、手順の曖昧性を排除しやすくなったことは間違い無く、運用部門の負担は減った。同社では BIGLOBE Business Service Desk という 24 時間 365 日の運用支援体制を敷いており、2011 年から 2013 年にかけてサービスや業務の数が 2 倍（約 25 サービス/50 業務が、約 50 サービス/100 業務）に増えた。従来から要員のオーバーワーク状態が問題化していたが、'12 年度から受入プロセスを大きく改善したことによって作業の定型化も進み、省力化施策も検討しやすくなった。また、「実装と手順書とで内容が異なっていて手順どおりに作業を進めることが出来ない」という問題も実質ゼロ化できた。結果として、単位作業あたりの消費工数も改善し、運用体制は発足当初のまま（二人一組の最小構成）を維持できている。更に、従来は四半期に一度は作業上のミスやトラブルが発生していたが、改善後は 1 年以上 0 件を継続できている。

ただし、開発側が手順書を作成・精査するための工数が減ったわけではなく、場合によってはむしろ増えたとも言える。従来ならば曖昧なままでも受け入れられていた業務が拒否される状態にもなったので、開発者や営業担当者の不満が募る場合もある。従って全体最適化とは言いがたい点も残っているが、運用品質の改善には大きく寄与したと言える。

6.5. チェック方法の工夫の効果

「トラップ項目だけをまずチェックする」ことで、読み合わせの手間やチェックの形骸化を減らすことができたケースはあったが、今のところは実験レベルに留まっている。トラップ項目を仕込むにはシステムやサービスの全容を理解し、かつトラップ質問を作成するノウハウも身につける必要があるため、参考テクニックとして紹介し始めたばかりの状態である。また、プロジェクト全体での開発期間や費用の圧縮、障害発生率等では複合的な要因が積み重なってしまうため、「紹介した要件定義法に直結した効果として計れる要素」が少なく、効果測定法はまだ模索を続ける必要がある。

6.6. 副次的効果

wish、should、must の 3 分類法は、会社の規定類やガイドラインなどを作成する際にも有効であった。従来の「～すべきである」あるいは「～しなければならない」といった表現では、それが「必達義務」なのか「理想とする努力目標」なのかの解釈が担当者によって変

わってしまう問題があったが、この方法によって区別が明確になった。

また、セキュリティ検証ツールを用いる効果として、テスト工程の省力化にも繋がった。ツールで精度良く検証するためには、作り込んだ機能がひととおり正常に動く必要がある。すなわち、セキュリティ検証の定義を作ることがシステムテスト工程における正常系テストと同義になる。また、検証の際には疑似攻撃として様々な異常値や異常操作を投入してくるので、「スキャンが停まること無く終わったということは、相応の異常系テストに耐えられた」と判断できる。このように、正常系試験と異常系試験とセキュリティ検証とを 1 step で、つまり従来の 3 分の 1 で済ますことができた。筆者のプロジェクトでは「セキュリティチェックと異常系テストの工数が 0.5 人月弱の工数は必要」という見積に対して、「AppScan を用いることで検証は十分」と見なし、実質 3 人日程度で済ませ、問題も生じなかった。

7. 適用と展開で苦労した点

適用と展開に当たっては、ルールの分かりやすさやツールの使いやすさといったテクニックやテクノロジーの問題ではなく、「アクター毎に対する気配りと心理戦」の様相が強かった。例えば、運用要件の定義法は、理屈で考えればごく当たり前のことではある。しかし、実際の現場では「そこまで運用手順を決めるのは面倒だ。そこまでしなければならないくらいなら、運用移管はしない方が楽だ」と今でも非難されることもしばしばである。開発会社によっては「ユーザの利用シーンではなく、実装で要件を書かれた方が開発の効率は良い。実装要件以上の事を知ろうとするのは時間と工数の無駄ではないのか」という声もあり、調整に苦慮することがあった。

更に、「一度 must で目標を決めてしまうとそれを変えられなくなってしまい、出荷判定に永遠に合格できなくなる」という不安の声も上がった。実際は予算・工期・諸条件を勘案して重要度も変えれば良いのだが、「一度 must と決めた以上は変えるべきではない」という声もあった。

ここで安易に妥協せず、根気強く交渉と啓発と改良を継続することが、最も苦労を要している。「ルールだから」で所定のやり方を押しつけるのではなく、従来のやり方のデメリット、新しいやり方のメリットを粘り強く何度でも同じ説明をし、手法自体も改良し続ける必要がある。一律の手法や基準で統一するのではなく、「トラブルさえ起こらなければ、やり方は柔軟で良い。トラブルを予防できると思ったら採用すれば良いし、もっと良い方法があるならばそれを使えば良い」という鷹揚さを持ちつつも、「トラブルさえ起こらなければ」という前提条件だけは崩さない意志が求められた。

8. 今後の課題

前述の「チェック方法の工夫」と「運用要件定義方法」については、効果が出ているプロジェクトもあるが、すべての案件に適用しているわけではなく、今後の横展開と担当者の教

育が課題である。その際は、手順書の書き方やシステムに関する知識などのテクニック面の向上ではなく、「他者に仕事を依頼する（あるいは引き継ぐ)際に必要な心構えと気配り」の啓発と向上が重要である。

8.1. どうしても目標値や基準値を決められない、という場合への対応

「今までに無かったものを作る」という前提でプロジェクトを起ち上げるのであれば、プロジェクトメンバーに十分な知見が無く、周囲の熟練者に尋ねても具体的な目標や基準を定められない場合もある。こうした場合、往々にして「やってみなければ分からないので、まずは作ってみよう」と、アジャイル開発手法で始めることになる。

その考え方は無論正しいのだが、適切にプロジェクトをコントロールしないと、何も決まらないままリリース日が迫り、実用に耐えられない物ができ、修正コストがかかり続けるリスクが付いて回る。

このリスクを回避するために、まずは仮説・仮定で良いので何か一つの指標を定めてから進める、ということにプロジェクトオーナーが注意を払う必要がある。例え仮説であっても、一つの定点が決まれば必然的に別の要素も決まってくる。そして、どこかで矛盾や破綻を来たしたならば、そもそもの仮説が間違っているとして最初の定点を見直せば良い。アジャイルとは「何も決めずに開発を進めること」や「開発した結果で条件を決めること」ではなく、「随時仮説を立てながら、適切な定点（目標）を次々と見出しながら進んでいくこと」である、という意識を醸成していく必要がある。

8.2. 上流工程をどれだけ工夫しても付いて回る課題

上流工程においてこれまで紹介した施策を活用していても、細かな要素に至るまでの徹底は難しい。仮に徹底できていたとしても、下流工程でのトラブルを皆無にできるわけではない。

セキュリティ検査や単純な機能検証における問題であれば修正は比較的容易なのだが、スループットなど性能面の検証においては表 A-4-8 のような「原因」と「結果（事象)」の取り違えが起こりやすい。

従来の物理システムが対象であれば、スケールアップもスケールアウトも安易には行えなかったため、真の原因追及に注力されることが多かった。しかし、仮想化システムやクラウドサービスではスケールアップもスケールアウトも容易に行えるため、ついつい真の原因の追及と対処がおざなりになりやすい。費用対効果で考えればそれで良い場合もあるが、人材育成の面からは問題と言える。

sar や vmstat[7] といったリソースモニターツールの描画だけでは、真の原因要素を見つけることは極めて困難であり、システム毎の特性もあるので、ツール化あるいはチェックリスト化すれば誰でも真の原因が判断できるような状態にはならない。真の原因要素を正しく

[7] CPU やメモリ、ディスクなどのシステム情報を確認・出力できる Linux のコマンド。

見出すためには、モニターツールで収拾した情報を単純に読み上げるのではなく、得た情報の意味を詳細に分析できるデータアナリストが必要である。データの分析には様々なシステムの経験と知識が不可欠であり、長期的視野で計画的に人材育成を行い続ける必要がある。

表 A-4-8　原因と結果の取り違え例

計測された事象	開発担当者がしがちな解釈	開発担当者が行いがちな対処	実際の原因例	真に必要な対処例
CPUが限界に達する	CPUのスペックが足りない	サーバ増強（CPU増強、台数増）を求める	無駄なループやSQL構文があり、CPUを無駄に消費している	コードのデバッグモジュールへのパッチ適用
メモリが限界に達する	メモリの量が足りない	サーバ増強（メモリ増設、台数造）を求める	メモリーリークを解消できていない	
ストレージの反応が遅い	ストレージのスペックが足りない	より早い(高価な)ストレージへのUpgradeやシステムの更新を求める	Fileのopen / close を無駄に繰り返している。ロック待ちを繰り返している。	コードのデバッグ ロック処理の改良
			nfsを用いて大量のファイルを一度に読み書きさせている。	ファイルの圧縮や統合 ストレージ領域の分散配置

9. おわりに

顧客に安心してご利用いただける安全なサービスを提供することは、企業としての必須条件である。しかし、サービスのライフサイクルが極めて短いスパンで回っているインターネットの世界では、いたずらに絶対的安全性の保証を追求しても、持続可能なビジネスとしての現実味を薄めてしまう。品質改善活動を品質管理部門が主導するとどうしても「信頼性・安全性」を求めがちであり、事業部門が主導すると「効率とスピード」を求めがちになる。しかし、顧客からすれば、信頼性・安全性と利便性は決してトレードオフではなく、同時に高めていくべき両輪の関係である。

そのためには、例えアジャイル開発手法を謳っていたとしても、「作ってから計る、作ってから決める」のではなく、「指標や目標を決めてから作り、結果を検証する」ことが必要である。これはウォーターフォール式を推奨するという意味ではなく、チームとして達成すべき目標を具体的に描き、メンバー全員でイメージを共有した上で開発を進める、という意味である。

今回紹介した手法も、すべての案件に対してルール的に適用させているわけではなく、あくまで「使った方が良いと判断する場合は使う」という扱いに留めている。サービスの品質を継続的に高めるためには、教科書的なやり方やチェックリスト類をそのままルールとして利用・模倣させるのではなく、当事者に「収益を出しながら工夫し続けるという覚悟を持たせること」が最も重要である。その意味で、品質向上に真に必要なのは、ルールやツールの

整備ではなく、顧客・開発者・運用者すべてと心理戦的な交渉を継続する意志と、対象者にとって分かりやすい説明を行う表現力である。

参考文献

[1] IPA/SEC 2013/08/05 アジャイル開発の「プラクティス・リファレンス」
http://www.ipa.go.jp/files/000035261.pdf
[2] NEC 技報 Vol. 57 No. 6 /2004
http://jpn.nec.com/techrep/journal/g04/n06/pdf/t040624.pdf
[3] NEC 事例紹介 2006/4　BIGLOBE における SOA 事例
http://www.nec.co.jp/library/jirei/bglo/index.html

A－5

要件定義の品質向上に向けた取り組み [1]

1. 概要

本編では、富士通株式会社が同社プロジェクにて、体系化した要件定義手法を適用した事例を紹介する。

今や社会インフラシステムに限らず、企業における情報システムは、社会や国民生活に欠かせないものになってきている。情報システムの信頼性や安全性の向上は、どのような情報システムにも求められるといっても過言ではない。

システム開発でのバグを分析してみると、要件定義に起因するものが多い。また、ユーザ企業も要件定義工程を改善しなければならないという思いがある。このような状況を捉え同社ではシステム開発における要件定義の品質向上のための取り組みをいくつか行っている。その中で、要件定義手法 Tri-shaping®（トライ・シェイピング®）で実施している品質向上策を紹介する。同社は、2011 年に要件定義手法 Tri-shaping®（トライ・シェイピング®）を発表[1]した。Tri-shaping は、要件定義の「抜け」「漏れ」「曖昧」だけでなく、その要求が企業や業務、利用者にどれだけ価値を提供できるかなどビジネス価値をも品質の対象としている。本事例では、要件定義での品質向上の取り組みを、３つの切り口で整理し対策を紹介していく。

2. システム開発における要件定義の品質

昨今のシステム開発では、要件定義起因の障害が多く、納期・コストの超過という問題が発生している。また、完成したシステムに対する満足度が低く、作成した機能の半分は不満足だったという事例や、せっかく IT 投資を行ったのに、「箱（IT システム）だけ最新のものに置き換わって、ビジネスや業務プロセスが変わらないなんてナンセンス」という声も挙がっている。後者は、納期どおりにバグ無く情報システムを構築できたとしても生じる問題である。

これらは、システム開発における上流工程である要件定義に大きな課題があることを示している。システム開発における要件定義の品質とは何であろうか。

ソフトウェアが正しく動作するかといった正確性に対する品質、システムが停止しない、性能が良いなど非機能に対する品質、コンシューマやシステム利用者が満足する、経営や業務に貢献するといった価値に対する品質まで求められるのが要件定義である。

---

[1] 事例提供：富士通株式会社　ＳＩ技術本部システム技術統括部　森田 功 氏、木村 剛 氏

要件定義の品質向上のため、同社は、要件定義手法 Tri-shaping をまとめた。Tri-shaping では様々な品質向上策を提言しているが、本編では、品質向上策として Tri-shaping の一部を紹介する。

3. 要件定義の品質の課題

3.1. 価値に対する品質の課題

昨今の企業における情報システムの位置づけは、従来のような手作業を機械化する、企業内の効率化のための道具から、経営や業務に貢献し、競争優位を築く戦略的な武器へと変わってきている。さらに、コンシューマやお客様に如何に満足を届けるかなど、社会システムへと広がりを見せている。このような状況の中、完成したシステムに対する満足度が、作成した機能のうち半分が不満足であるという結果に陥っている事例も少なくない。これらの問題は、納期どおりにバグ無く情報システムを構築できたとしても生じることである。システムとして実装する前段階の要求の時点で、要求が実現した時のビジネス的効果がどのくらいあるのか見極めていないからである。すなわち要求の「価値」に対する品質の問題である。

3.2. 正確性に対する品質向上への課題

正確性に対する品質とは、いわゆるバグと言われるものである。システム開発のバグを分析すると、要件定義書の「抜け」「漏れ」に起因するものが多い。

Tri-shaping では、「抜け」を必要な記載要素が全く書かれていないことと定義し、「漏れ」を記載要素に対し全てのパターンが書かれていないこと定義している。

「抜け」が発生する主な原因は、要件定義として「何を」「どの程度」「何のために」記載なければならないという事が不明確であることである。要件定義に時間がとれないことなどにより、必要なドキュメントや必要項目自体を定義しないことである。

「漏れ」が発生する主な原因は、１つの条件内で取りうる値のバリエーションが漏れる、また条件が複数組み合わさった時に組み合わせにおいて漏れが生じるなどである。

3.3. 非機能に対する品質の課題

システムへの要求は機能要件と非機能要件に分類して捉えることができる。機能要件は、業務を支援する機能で業務の一部であり、業務アプリケーションとして実現する。一方非機能要件は、システムが兼ね備えているべき条件であり、多くはシステム基盤で実現される。非機能は、システム基盤で実現されるが、ユーザ要求抜きに、システム基盤チームだけでは定義できない。しかし、非機能要件は、ユーザにとっては、実現イメージがわかず、機能要求より関心レベルが低くなり、設計工程に入ってからでも良いのでは、ベンダーに任せておけば良いのでは、という状況になりがちである。結果大きな手戻りが発生する。何を、どの程度決めればよいのかは非機能要求グレード等で示されてきたが、要件定義として業務要件や機能要件と連携しながら非機能要求の合意形成を行っていくプロセスが詳細には示されて

いない。

4. 要件定義の品質向上への対策

4.1. 要求の価値に対する品質向上策

要求の価値を見極め、価値ある要求を形成するために、Tri-shaping では図 A-5-1 に示すように、要求を構造的に整理する。

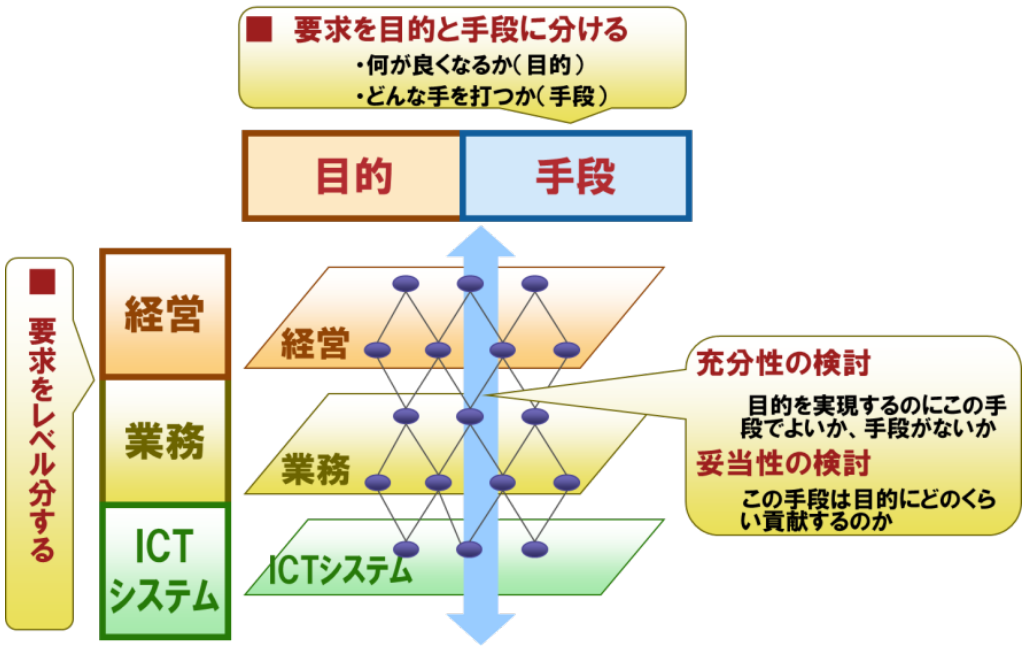

図 A-5-1　要求を構造的に整理する

仕掛けは簡単である。まず、要求を「○○を良くしたい」といった目的と「○○を実行したい」といった手段に分けることである。さらに、要求を経営レベル要求、業務レベルの要求、システムレベルの要求とレベル分けすることである。そして、それらの関連を明確にする。そうすることにより、ひとつは、この目的を実現するためには、この手段で良いのか、他の手段の方が良いのでは、他にも手段があるのかなど、いわゆる充分性の検討が行いやすくなる。もうひとつは、この手段は目的にどのくらい貢献するのか、この手段は今回の目的から外れるなど、いわゆる妥当性が検討しやすくなる。このような検討の中で経営や事業、業務などの上位の目的に基づいた、下位の目的の設定や手段の設定ができるとともに、どのくらいの効果があるかなどを明確にしながら価値の判断を行う。

4.2. 機能要件の正確性の品質向上策

「抜け」「漏れ」「曖昧」を低減するために、Tri-shaping では、以下の対策を提言してい

る。

4.2.1. 「抜け」の防止策

Tri-shaping では、バグを分析し、抜けやすい仕様を見つけ出し、その対策を提示している。バグの例として、マスタが存在しないことによる「Null Pointer Exception」の発生という、データに関する要求仕様の抜けがある。例えば、受注時に顧客マスタに存在しない顧客からも受注が来る、といったことの考慮不足によるバグの発生である。Tri-shaping では概念データモデルのカーディナリティの「0 あり」「制限あり」を精査し洗い出し、その時の仕様を明確にする手順とドキュメント（図 A-5-2）を用意している。昨今では、データ中心アプローチなどで手順やドキュメントが明確にされていたり、DBMS の機能として実装されていることだが、現実は漏れたり後工程で気づいたりしている。データのような共通の仕様は誰がどこまで定義しているか不透明であり、業務ごとに担当を振られ、機能の仕様を定義している人にとってはデータの仕様をどこまで定義するのか分からなく、結果データの仕様が漏れやすい。

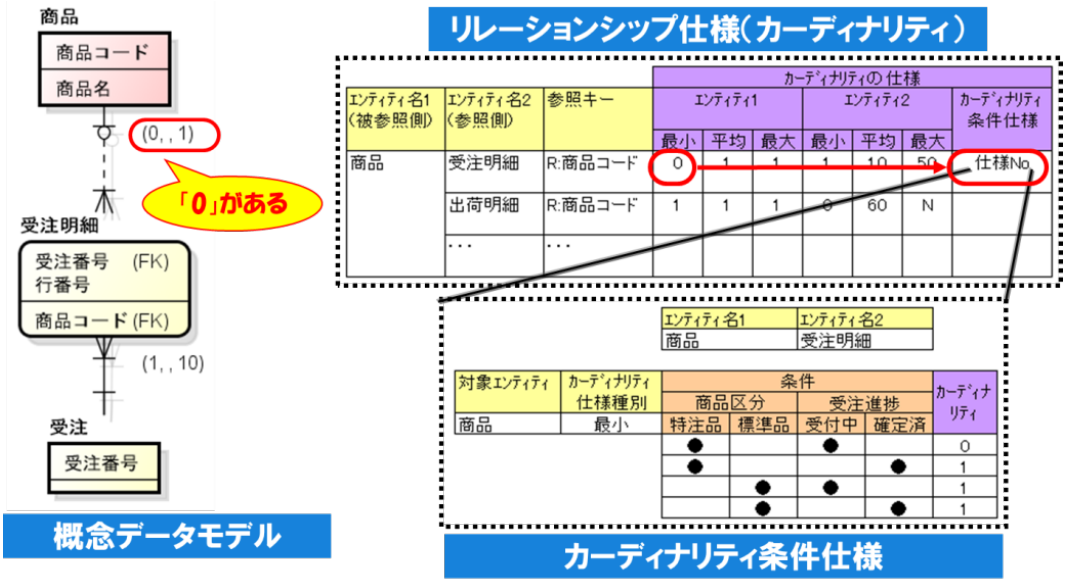

図 A-5-2　カーディナリティの仕様書

4.2.2. 「漏れ」の防止策

Tri-shaping では、仕様の漏れを低減するために、いくつかの対策を提示している。

課題で、１つの条件内で取りうる値のバリエーションが漏れる、また条件が複数組み合わさった時に組み合わせにおいて漏れが生じると述べた。その対策を紹介する。図 A-5-3 は業務ルールを文章ではなく、表で表現した例である。

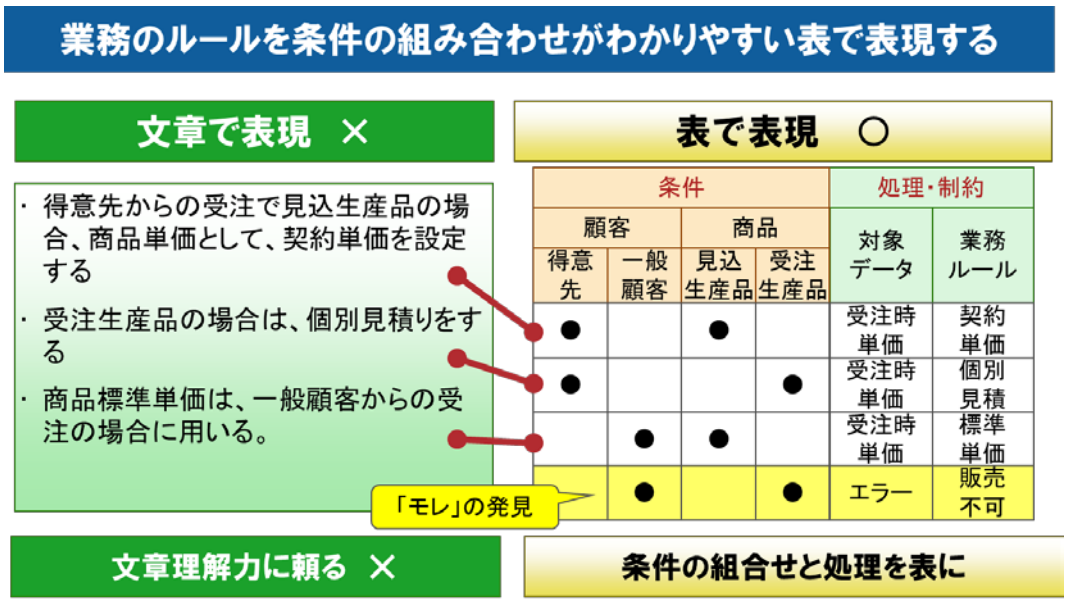

図 A-5-3　業務ルールを表で表現する

曖昧なく日本語文章を書くのは極めて困難である。検討過程を共有したり、文化を共有したりし、行間を読まないと正確に受け止められない。少しでも、曖昧なく表現するには条件部分を表で表現すると良い。表にすることにより、条件の取りうる値や組み合わせパターンの漏れがなくなり理解もしやすくなる。

「漏れ」を低減するためには、業務自体を整理し可視化しなければならない。業務そのものにバリエーションが沢山あり複雑になっている。漏れを無くすためには、少しでも業務上の複雑さを低減することが効果的である。業務上の複雑さは業務が標準化、整理できていないことも大きな要因である。例えば、似て非なる業務が存在している、同じ業務・機能なのに異なるシステムを利用している､顧客や商品のバリエーションが企業内でバラバラである、使われていないコードや区分が存在している、などである。Tri-shaping では詳細な仕様を作成する前に、業務上のバリエーションの整理、統合、削除を検討することから始める。

4.3. 非機能要件の品質向上策

非機能要件は、本来のビジネスの目的を満たすための業務要件や機能要件とは異なり、その業務や機能を実現するために情報システムが兼ね備えるべき品質要求と制約である。このため、要件定義工程は、ビジネス活動に適した情報システムの品質要求や制約を定義する、情報システムの設計に入る前の機会となる。それ故に、ビジネスの要求を捉えて非機能要件として取り込むための実践的なガイドラインが重要となる。具体的には、ユーザ企業が提供するサービスが情報システムに依存する度合いや、取り扱うデータの重要度など、ビジネス活動により情報システムに求められる条件をステークホルダーで共有し、段階的に詳細化し

ながら非機能要件を定義していくための合意形成プロセスである。

同社では、非機能要求グレード[2]を全社標準に取り込み、要件定義における非機能要求項目（何を）や要求レベル（どの程度）の可視化に活用している。Tri-shaping では、非機能要件定義の品質をさらに向上するべく、ビジネスの視点で妥当性を確保し、整合性の取れた定義内容に仕上げるための合意形成プロセスを提言している。

4.3.1. 非機能要件定義の問題点

(1) 非機能要件項目数の問題点

非機能要求グレードでは重要項目で 92 項目、メトリクス数が 236 もあり、従属関係やトレードオフの関係にあるこれらの項目の定義内容を短期間で手戻りなく決定することは容易なことではない。

(2) 非機能要件と業務要件の乖離（かいり）

非機能要件は、ビジネス目的を満たす業務要件や機能要件とは別に議論されることが多い。ビジネスを起点としたトップダウンの要求よりも、情報システムに携わるステークホルダーの課題・ニーズから抽出されるボトムアップの要求に偏りがちである。

また、要件定義工程においては、開発体制や規模により異なるが、業務要件・機能要件の定義は業務チーム、非機能要件の定義はシステム基盤チームなど、役割を分担して進める傾向がある。双方のコミュニケーション不足などにより定義内容に齟齬（そご）が起きることがある。これらが、非機能要件と業務要件が乖離する要因である。

(3) 合意形成の問題点

ユーザの業務に近い業務要件や機能要件の定義は業務部門が理解しやすく、技術的側面が強い非機能要件の定義は業務部門が理解しにくい。非機能要件の定義はしたものの、業務部門や情報システム部門、業務アプリケーション開発チームやシステム基盤構築チームなど、ステークホルダーが共通認識の下で合意しているとは言い難い状況が起こり得る。

4.3.2. 非機能要件定義のポイント

(1) 非機能要件項目の種類

システム化に当たり、業務要件はシステム要件に具体化され、機能要件と非機能要件に分類される。ソフトウェア品質特性の標準文書[3]には、品質特性として、機能性・信頼性・使用性・効率性・保守性・移植性の 6 つの種類がある。Tri-shaping では、ソフトウェア品質特性の標準文書をベースに非機能要求グレード、非機能要求仕様定義ガイドライン[4]の項目を非機能要件項目に取り込んでいる。非機能要件項目の種類は、可用性、性能・拡張性、運用・保守性、移行性、セキュリティ、システム環境・エコロジー、ソフトウェア保守性、ソフトウェア移植性、ユーザビリティの 9 つである。

非機能要件を満たすための仕掛け・仕組みを実現する領域はフレームワークなどのアプリケーション基盤やシステム基盤の領域であるが、異常時における仕掛中の取引の判定や再送処理などの非機能要件に基づく処理判断は、アプリケーションの領域で実現するもの

である。そのため、非機能要求グレードでは、システム基盤をスコープの対象としているが、Tri-shaping では、一部のアプリケーション領域も非機能要件のスコープの対象としている。

(2)　非機能要件要素の分解

先に述べた 9 つの非機能要件項目は、非機能要求グレードのメトリクス相当の要件要素に細分化される。それら要件要素は、品質属性、達成条件、達成手段の 3 つに要素分解している（図 A-5-4）。品質属性は、情報システムを構成するプロダクト、又はプロセスにおける品質特性であり、業務を実現する情報システムの品質目標に相当する。達成条件はある品質属性を達成する時の条件であり、ピーク時間帯における 1 時間あたりの平均値、などの条件が相当する。達成手段はある品質属性を達成するための方法であり、冗長化などの手段が相当する。

要件要素を上記の 3 つに要素を分解することで、業務を実現する上でのシステム化の品質目標を明確にし、①要件要素間の整合性の確保、②システム化のリスク認識、③概算見積りの精度向上、の効果を狙っている。

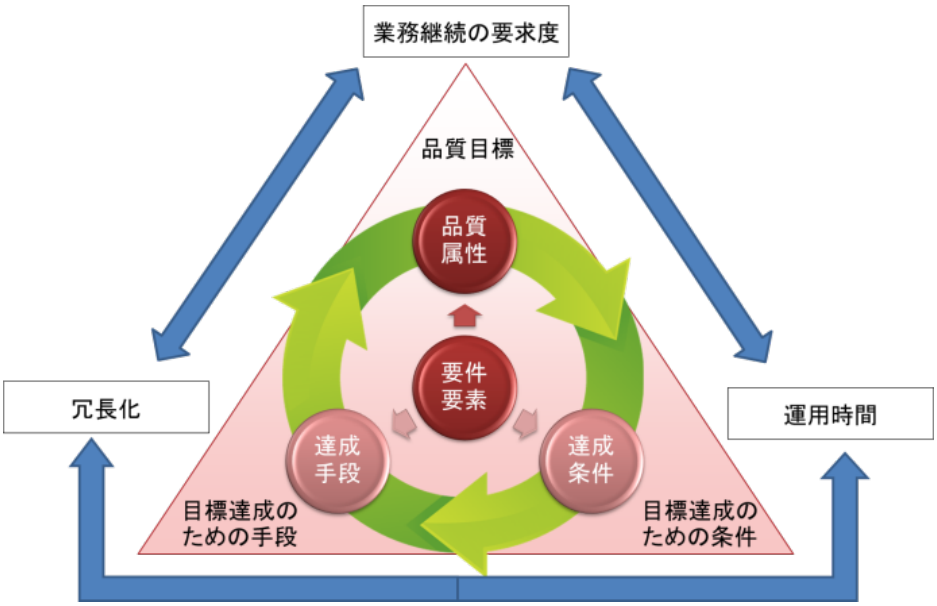

図 A-5-4　品質属性・達成条件・達成手段

(3)　非機能要件要素の領域分け

品質属性、達成条件、達成手段に分解された要件要素は、要求の対象ごとにそれぞれ 3 つの領域に分類することができる。3 つの領域とは、ビジネス活動を対象とするビジネス領域、情報システムが提供する機能・サービスを対象とするソリューション領域、情報システムの構成要素を対象とするテクニカル領域である。ビジネス領域は、ビジネス活動に伴う業務の機能や役割を観点に抽出される業務特性・業務継続・業務連携などの要求を分

類する領域である。ソリューション領域は、情報システムが提供する機能やサービスを観点に抽出される性能目標・システム復旧などの要求を分類する領域である。テクニカル領域は、アプリケーションやアプリケーション基盤・システム基盤などの情報システムの構成要素を観点に抽出する冗長化・資源拡張などの要求を分類する領域である。

非機能要件要素には依存関係があり、ビジネス要件から導き出される要件要素、その導き出された要件要素をもとに決定される要件要素がある。Tri-shaping では、非機能要件要素を 3 つの領域と 35 の非機能要求種別で分類し、それらの関係を表した非機能要求体系としてモデル化している（図 A-5-5）。そのモデルを利用して、ビジネス要件を起点に非機能要求種別の上位から下位の順に非機能要件を確定していくことで、ビジネス要件との妥当性を見極めながら手戻りを低減し非機能要件を定義することができる。

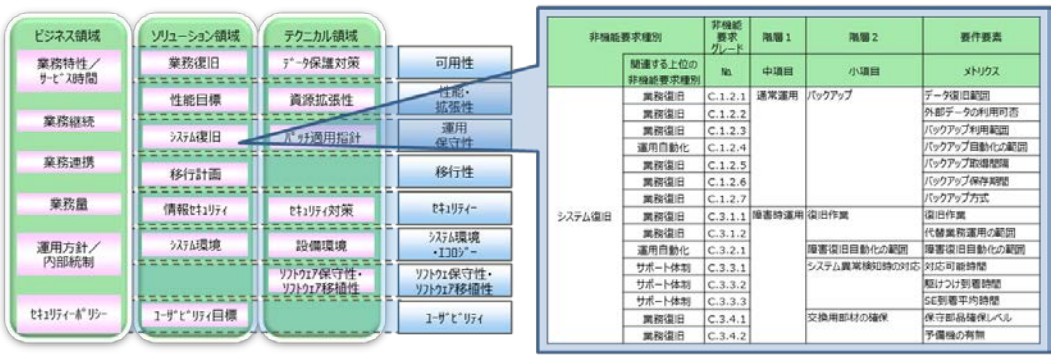

図 A-5-5　非機能要求体系の概念図

(4)　ビジネス要件と非機能要件の関係

a.　ビジネス要件との関係付け

非機能要件の優先度は、信頼性重視、性能重視など、ビジネス分野により異なる。業務継続やセキュリティポリシーなど、ビジネス活動に関わる非機能要件は、ビジネス視点のトップダウンの要求としてビジネス領域で整理する。信頼性や性能など、情報システムに関わる非機能要件は、ステークホルダーの課題・ニーズ視点のボトムアップの要求としてソリューション領域・テクニカル領域で整理する。非機能要求体系を用いてトップダウンの要求とボトムアップの要求を関係付けることで、ビジネス視点で合理的な、経営層に説得力のある非機能要件定義に仕上がる。非機能要件を整理・分析する過程において、トレードオフなどでステークホルダーのコンフリクトが発生した時には、要求の構造を用いてトップダウンの要求であるビジネス領域を基準に取捨選択や優劣をつけ判断する（図 A-5-6）。

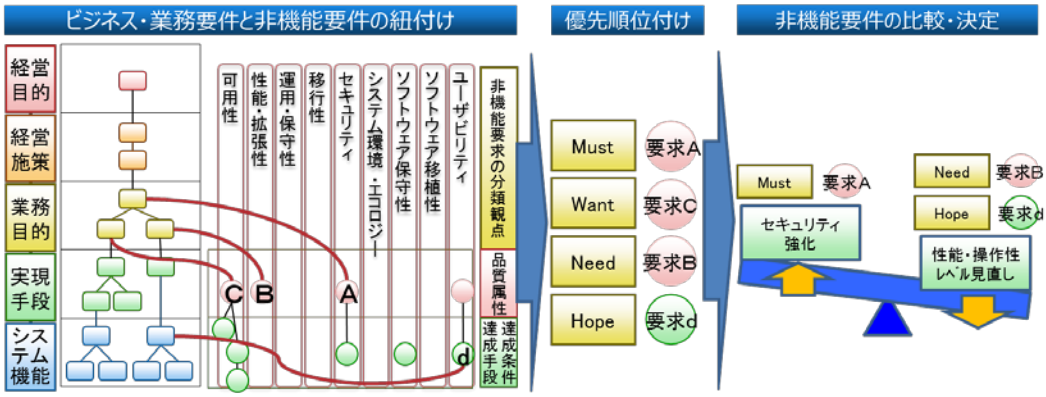

図 A-5-6　ビジネス・業務要件と非機能要件の紐付けと優先順位付け

b.　ステークホルダー要求の引き出し

ステークホルダーは、経営者、業務部門、情報システム部門（企画、設計、開発、運用、保守担当）、開発者（ベンダー）、利用者など様々である。また、ステークホルダーにより関心事や課題・ニーズは異なる。要件定義では、具体性に欠けることから、ステークホルダー要求の引き出しや確定に時間を要したり、手戻りが発生したりする可能性がある。そこで、情報システムで実現すべき非機能要件をステークホルダーから引き出すには、ステークホルダーの共通の理解を促すモデルが必要となる。ステークホルダーの IT リテラシーや経験を考慮し、システム仕様・論理設計・物理設計の整合が取れたモデル（以下、参照モデル）（図 A-5-7 ）を示すことで、ステークホルダーから暗黙の要求を引き出すとともに、数ある非機能要件を参照モデルの仕様で補完することができる。参照モデルは、非機能要求グレードを活用してシステム仕様（非機能要件）を可視化したものであり、実績のある現行システムや類似システム、あるいは各種ソリューションを指す。

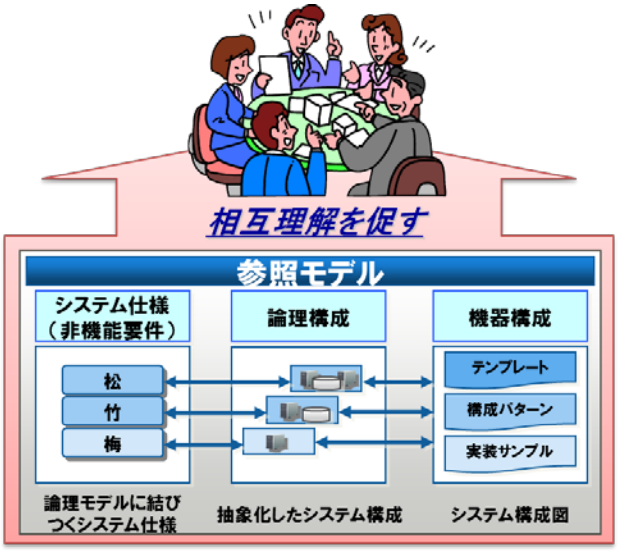

図 A-5-7　参照モデルの概要

c. 前提条件と制約

前提条件は設定された条件で、ステークホルダーの合意によって変更可能であるのに対して、制約は与えられた条件であり、変更することはできず、それを満足することが求められる。前提条件と制約は、ステークホルダーの権限や役割で視点が異なるため曖昧になりがちで要求や実現手段に影響が及ぶことから、可視化しステークホルダーで共有しておくことが重要である。要件定義における前提条件には、仮に決定する未確定な要素などがある。制約には、法令や商慣習などのビジネス制約と、利用するサービスやソリューション、製品などの技術制約がある。また、それ以外にもシステム環境、設備などの運用制約やコスト制約がある。システム化に必要なこれらの条件を洗い出すことで、システム化や業務・システム運用のベースラインを作り上げコントロールすることができる。なお、前提条件は、リスクとして管理するとともに要件の変更管理プロセスでコントロールする。

d. 優先順位

非機能要件を選択あるいはレベルを決定するための判断基準を明確にすることが重要となる。非機能要件には経営・業務の視点で必ず実現しなければならない要求から、できれば実現したい要求まで様々な度合いや優先順位があるため、それらをあらかじめ示しステークホルダーで共有することで優先する非機能要件を判断することができる。

e. トレードオフの関係

非機能要件には、トレードオフの関係がある。例えば、セキュリティを強化すれば運用における操作性が低下するなどの関係がある。トレードオフを解消するには、トップダウンの要求であるビジネス領域の要求に基づいて要求度合いや優先順位付けを行い、要求を選択するための判断基準を明確にする。

f. 実現手段の明確化

非機能要件の実現性や技術的根拠を示すには、ステークホルダー要求の引き出しで使用した参照モデルを用いて非機能要件を実現する手段を明らかにする。なお、具体的な実現方式は設計工程で具現化する。

g. 評価方法、確認手段の設定

非機能要件が満たされているか、品質保証における妥当性を確認するための測定項目や評価方法、評価タイミングを決める。定性的な内容の要件など、ステークホルダーによって認識が異なる可能性があるものは、要件定義段階であらかじめ評価項目を定めておき、ステークホルダーで合意する。また、評価実施のために必要なツールなどの確認手段を明らかにする。

h. 合意形成

Tri-shaping では、非機能要求グレードなど、要件要素の一覧をベースとした従来の方法に加え、上記に挙げた様々な分類観点を用いて、ビジネスの要求を起点に、

ステークホルダーと合意しながら要求を段階的に詳細化し要件として仕様化するプロセスを提唱している（図 A-5-8）。

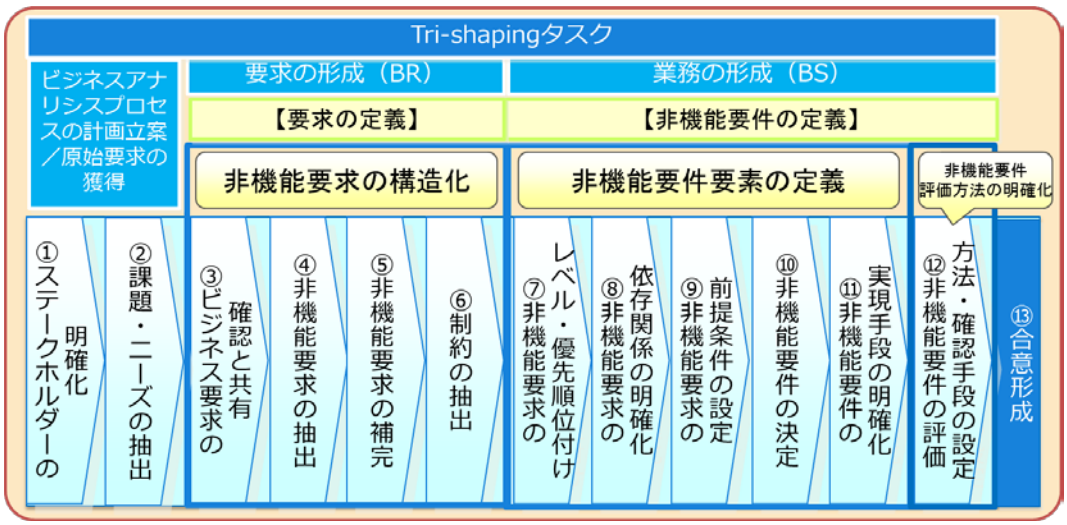

図 A-5-8　Tri-shaping 非機能要件定義のタスク

4.3.3. 非機能要件定義のプロセス全体概要

(1)　非機能要求の構造化

a.　ビジネス要求の確認と共有

経営ビジョン・経営方針、部門ビジョン・部門方針、システム化方針、スローガン、経営目的・経営施策、業務目的・業務施策などの情報から、数量・時間・場所などを観点に、ビジネス要求に含まれている業務特性・業務継続・業務連携・業務量・運用方針・セキュリティポリシーなど、トップダウンの非機能要求を確認する。確認した内容はステークホルダーで共有する。

b.　非機能要求の抽出

ステークホルダーの関心事から抽出された課題・ニーズをもとに、非機能要件項目である可用性、性能・拡張性、運用・保守性・移行性などを対象に、ボトムアップの非機能要求を抽出する。

c.　非機能要求の補完

現行システム仕様や類似システム仕様、あるいはソリューション仕様など、ステークホルダーが参照し共有できるモデルを具体的に例示し、明確に示されなかった暗黙の非機能要求を引き出すことで、ボトムアップの非機能要求を補完する。

d.　制約の抽出

法令や組織規定、提携先とのインタフェースなど、要求を実現する上でのビジネス観点の制約や、利用するサービスや適用する製品など、技術観点の制約を抽出し、システム化や業務・システム運用に制限を与える条件を明らかにする。

(2) 非機能要件の定義

a. 非機能要求のレベル・優先順位付け

制約による非機能要求の実現可否や、非機能要求間のトレードオフを整理し解消するために、非機能要求の要求度合いや優先順位付けを行う。

b. 非機能要求の依存関係の明確化

制約とともに非機能要求の依存関係を明らかにする。依存関係があるものについて、矛盾した要件がないか確認する。非機能要求間でトレードオフがある場合は、要求度合いや優先順位付けを元に、どちらの要求を優先して採用するかを判断する。

c. 非機能要求の前提条件の設定

システム化や業務・システム運用に必要な条件に漏れが無いか確認し、要求や制約として抽出されていない非機能要求を設定する。また、非機能要求を達成するときの条件があれば設定する。設定した前提条件は、リスクとして管理する。

d. 非機能要件の決定

これまで抽出された要求と制約、前提条件を整理し、非機能要件として定義する。

e. 非機能要件の実現手段の明確化

非機能要件の実現性や技術的根拠を示すために、非機能要件を実現する手段を明らかにする。その際は、システム構成要素ごとに、実現する手段の実装範囲を考慮して実現性を検討する。なお、具体的な実現方式は設計工程で具現化する。

f. 非機能要件の評価方法・確認手段の設定

非機能要件が満たされているか妥当性を確認するため、測定する項目や評価方法、評価タイミングを決める。定性的な内容の要件など、ステークホルダーによって認識が異なる可能性があるものは、要件定義段階であらかじめ評価項目を定めておき、ステークホルダーで合意する。また、評価実施のために必要なツールなどの確認手段を明らかにする。

5. 要件定義手法 Tri-shaping の概要

Tri-shaping は同社で従来行っていた手法に、BABOK[5]や REBOK[6]などの業界標準の要素を加味したもので要件定義として必要十分な内容になっている。そして最大の特長は、どんな成果物を作成するかだけではなく、どのように成果物を作成していくかという過程、すなわち、人の思考をサポートできるノウハウや考え方を可視化することに注力したことである。いわば要件定義の実践ガイドブックである。

5.1. 要件定義手法 Tri-shaping の体系

要件定義工程はビジネスとシステムの橋渡しを行う工程であり、経営や業務に貢献する情報システムの要求を明確にし、新しい業務を作り、システム開発へ仕様として渡さなければならない。これを踏まえて Tri-shaping では大きく以下の 3 段階のビジネスアナリシス手法

をベースに体系化している。

- 要求形成手法 shapingBR(shaping Business Requirements)
- 業務形成手法 shapingBP(shaping Business Process)
- 業務仕様形成手法 shapingBS(shaping Business Specifications)

このビジネスアナリシスの実行をマネージメントするためのビジネスアナリシスの計画や管理も含めて体系化している。図 A-5-9 に Tri-shaping の体系を示す。

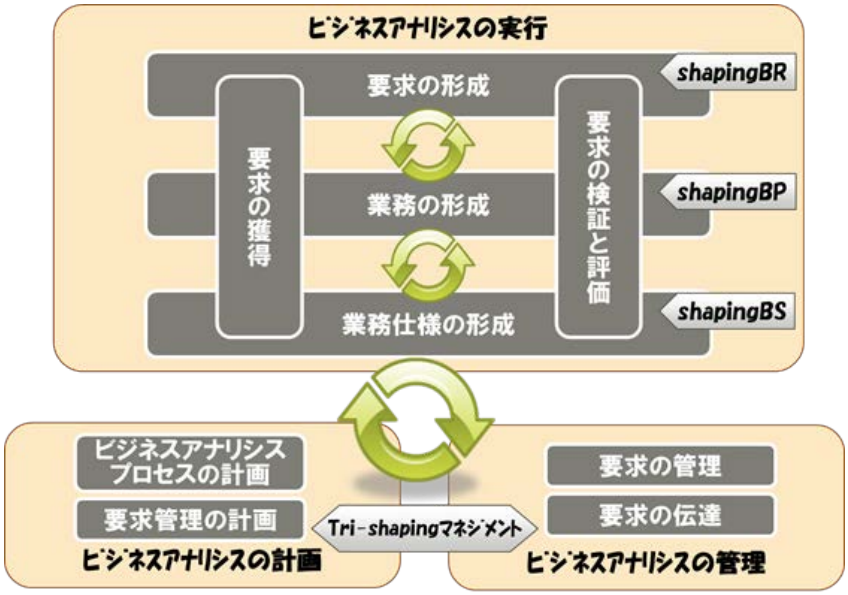

図 A-5-9　要件定義手法 Tri-shaping の体系

5.2. 要求形成手法 shapingBR

shapingBR は、要求の価値を見極め、経営や業務に貢献する要求をつくりあげていく部分である。ここでビジネス要求、機能要求を構造的に整理していくとともに、非機能要求の構造化も行い、これら全ての要求の関連を明確にしていく部分である。

5.3. 業務形成手法 shapingBP

shapingBP は、情報システムを利用する業務自体を検討していく部分である。情報システムは業務にとってはあくまでも手段である。shapingBR で明確になった要求をそのまま情報システムに落とし込むのではなく、まずは要求を実現する新しい業務を作り上げる。その際、業務をモデル化し、このモデルの上で新しい業務を検討し、決めていく。ここで「抜け」「漏れ」対策にあたる業務のバリエーションの整理、可視化や業務のシンプル化の検討も行う。

5.4. 業務仕様形成手法 shapingBS

shapingBS は、システム開発に渡すための仕様を作成する部分である。業務が立案できた

ら、その中の情報システムに関するシステム化要求を更に具体化する。業務アプリケーションとして実現する業務側のルールである業務仕様や非機能の要件を定義し、情報システム開発に渡す。「抜け」「漏れ」を低減するための表形式の仕様書などが用意されている。

6. 検証と考察

要件定義の品質を向上させようとすると、どうしても要件定義の負荷が大きくなる。

Tri-shaping を全面適用したあるプロジェクトのデータを図 A-5-10 に示す。

このデータは同社が参考指標として持っている工程工数比率、工程期間比率と実際のプロジェクトでの実績を比較したものである。

富士通の参考指標値との比較

	要件定義工程	開発全体 （要件定義～運用テスト）
期間	2.3倍	5%短縮
工数	1.5倍	4.5%低減

図 A-5-10　Tri-shaping 適用プロジェクトデータ

6.1. 要件定義の品質向上に対する考察

このデータは、要件定義の期間が 2.3 倍かかり、工数が 1.5 倍かかったことを示している。そして要件定義を含む開発全体では、期間が 5%短縮、工数が 4.5%低減したという結果が出ている。要件定義に時間、工数をかけても開発・テストなどが縮小できたということである。すなわち要件定義での品質向上が図られ開発工程での手戻り、バグ低減ができたと推察できる。

6.2. 要件定義の期間と工数の関係の考察

また、このデータから、要件定義工程は、工数増よりも期間増の方が大きいということが伺える。要件定義工程は合意形成したり承認活動をしたりすることで期間が掛かるということが推察できる。要件定義の見積もりは、作業やアウトプットをベースにスケジュールや見積もりを行うが、その見積もりでは、期間内に要件定義は終わらないという事を意味している。本データは 1 プロジェクトのデータであり、今後たくさんのプロジェクトのデータを収集して分析する必要がある。

7. 今後の取り組み

Tri-shaping はスクラッチ開発の要件定義として同社で整備した手法である。今後は本手法の実適用のしやすさとしてパターン化を進めていくとのことである。保守・改造パターン、

パッケージ適用パターン、既存システム利用パターンなど特徴に合わせた要件定義の整理をしていく計画である。

また、要件定義の効果はなかなか測定できるものではない。同社では、本当に要件定義の効果で開発工程が低減したのか、要件定義に時間をかければどのくらい開発工程が短縮できるのか、など、事例を蓄積して効果を定量的に示す手段を確立していくとのことである。

参考文献

[1] 富士通:要件定義手法 Tri-shaping 2011-2014, 2011
,http://pr.fujitsu.com/jp/news/2011/02/9-2.html .
[2] IPA/SEC:非機能要求グレード 2010-2014, 2010
[3] JIS(日本工業規格):JISX0129-1 ソフトウェア製品の品質－第 1 部：品質モデル 2003, 2003
[4] JUAS(日本情報システム・ユーザ協会):非機能要求仕様定義ガイドライン 2008, 2008
[5] IIBA(International Institute of Business Analysis):A Guide to the Business Analysis Body of Knowledge (BABOK Guide) Version 2.0 2009, 2009
IIBA 日本支部:ビジネスアナリシス知識体系ガイド(BABOK®ガイド) Version 2.0 2009, 2009
[6] JISA(情報サービス産業協会):要求工学知識体系(REBOK) 第 1 版 2011, 2011

A－6

ジェネレータツールを利用した
高信頼開発、高速開発の実践 [1]

1. 概要

本編では、基幹システムの課題解決にジェネレーションツール「GeneXus」を適用した、株式会社市進ホールディングスの事例を紹介する。

同社は 2010 年にグループ企業がホールディングス化され、システムにも統合・一元化を図り、高品質なサービスを迅速に提供することが求められるようになった。「市進」ブランドをより生かした事業展開を進めるために、基幹システムの現状課題（他システムとの連携・柔軟なシステム・サービスの顧客満足・運用維持管理の費用対効果）を解決する為に、新たに先進的な技術・手法であるイテレーション（反復）開発ができ、かつプログラムとデータベースを自動生成するジェネレーションツール「GeneXus」を利用した。

今回は教育業界が初めてというベンダーが新しい開発方法でシステムを構築するということで懸念はあったが、結果的には、予定の期日通り稼働を開始し、移行後のサービス追加も予想以上に短期間・低コストで実現できている。今後、何らかの事情でインフラ環境を刷新しなければならない時に、このツールの特長の一つである「未来を守る非 IT 環境依存型システムの提供」を発揮できれば、顧客サービスの充実に集中し事業の拡大を実現できると確信している。

2. 取組みの目的

同社では、多くの企業と同じようにグループ内でお客様の管理にコンピュータシステムを利用している。

しかし、近年以下のように

- 複数のシステムが独立して稼働し連携していない
- 業務の運用にシステムが合致しない
- システムの変更に際し効果以上の費用がかかる
- お客様に対し満足なサービスを提供できない

という問題点を抱え、既存システムでは十分な費用対効果を維持できないため、基幹システムの更改を検討した。

---

[1] 事例提供：株式会社市進ホールディングス　情報管理部 IT 戦略推進室　今林　豊　氏

新しい基幹システムでは、複数の独立したシステムを統合し、

（1） 顧客満足度を高め、業務改革を推進するためには、最終的に、市進側で迅速に開発・改善できること（内製化）

（2） 本当に良いシステムの要件を予め決定することが難しいので、処理詳細は最後の段階まである程度変更が可能であること

（3） 業務の改革にシステムが追随すること

を目的とした。

3. GeneXus の特長

上記の目的を達成できるツールを選定するにあたって複数社のベンダーに RFP(提案依頼書)を提示し、同社の(1)から(3)目的を達成し得るツールとして GeneXus を以下の理由により選定した。

（1） 専門知識（プログラミング、データベース等）が不要なこと

（2） 反復開発、かつ詳細設計まで行かなくても業務の流れを作成し確認できること

（3） IT 環境に依存しないこと

選定した GeneXus（ジェネクサス）の特長は以下の通りである。

このツールは南米ウルグアイの A 社が開発したアプリケーション自動生成ツールである。最大の特長は、「ビジネス（顧客要件・業務仕様）とテクノロジー（実装方法）を分けてシステムを構築できる」ことである。つまり、システムを構築する人（社内情報システム担当者や社外のシステム開発担当者）がプログラム言語や物理 DB などの環境を意識せず、「顧客要件」や「業務・機能仕様」を設定し「実装方法」を指定することで業務・機能に応じたシステムを自動生成することができる。

また、更に以下のような特長がある。

① 優れたメンテナンス性

② ナレッジベースによる資産の一元管理

③ 開発メンバー間の資源共有による開発の効率化

④ データベースリバースエンジニアリングツールによる既存データモデルの利用

⑤ オブジェクトの履歴管理、ナレッジ（システム資産）のバージョン管理

⑥ セキュリティ機能

RFP に対するプレゼンテーション実施前には、既存のベンダーか教育業界に詳しいベンダーに発注する予定であった。しかし、これまでの開発方法とは大きく異なる提案内容が同社の抱えている課題を解決できる可能性が高いと判断した。

その開発方法とは、ジェネレータツール「GeneXus」を利用するというものでありその特長は、

・ プログラムを自動生成する

・ データベースを自動生成する

・　特定のＩＴ環境に依存しない

という従来のプログラムコーディング手法を行わない、進化した性能を備えていることである。

「GeneXus」を使えば、3 つの目的が同時に満たせるという印象を受けたが、そのジェネレータツールの信頼性を他社事例説明の確認のみならず、動作検証を実施して先行開発をすることでツールの評価を実施した。

いずれにおいてもジェネレータツールの信頼性を高める結果となったが、同社独自にそのジェネレータツールを採用し業務に適用しているユーザを調査し訪問した。大規模システムで実際の動作を確認し、同社の想定している開発規模でも十分に適応可能と判断できた。また、内製化により追加のシステムを開発している担当者の声を直接聞け、同社でシステムの内製が可能でコスト面でもかなりの効果があると判断できた。このようなツール選定の検証を持って「GeneXus」ジェネレータツールを活用し基幹システム更改を進めることとした。

4. GeneXus を如何に開発に適用したか

4.1. 取り組みへの考え方

ジェネレータツールの特長を最大限に活かすため、システムの開発にあたって同社システム部門と開発ベンダーとの間で以下の合意を形成しながら開発を進めた。

・　ジェネレータツールの標準機能のみで開発を進める
・　イテレーション(反復)開発手法で開発を進める
・　課題は原則 3 日で決める
・　安易な機能追加は認めない

この取り決めを開発の前に行うことにより、システム開発でありがちな機能追加による工数の増大や、決定の遅れによるスケジュールの遅延を防ぐことができ、予定通りサービスインを実現した。

開発中はベンダーが常駐できるプロジェクトルームを設置し、以下のような新規開発体制で実施した。

4.2. 開発組織

開発チーム（イテレーションチーム）は、システム構築を短期間に実装するために担当機能を分けてＡチーム、Ｂチームの 2 チーム編成したが、プロジェクト全体でお互いの齟齬や行き違いのないように双方のプロジェクトマネージャと同社の確認チームは 1 つのチーム（要求定義チーム）とし、すべての打ち合わせに参加する体制をとった。このチームを構成し打ち合わせをすることで、要求機能の漏れや勘違いを減らすことができた。（図 A-6-1）

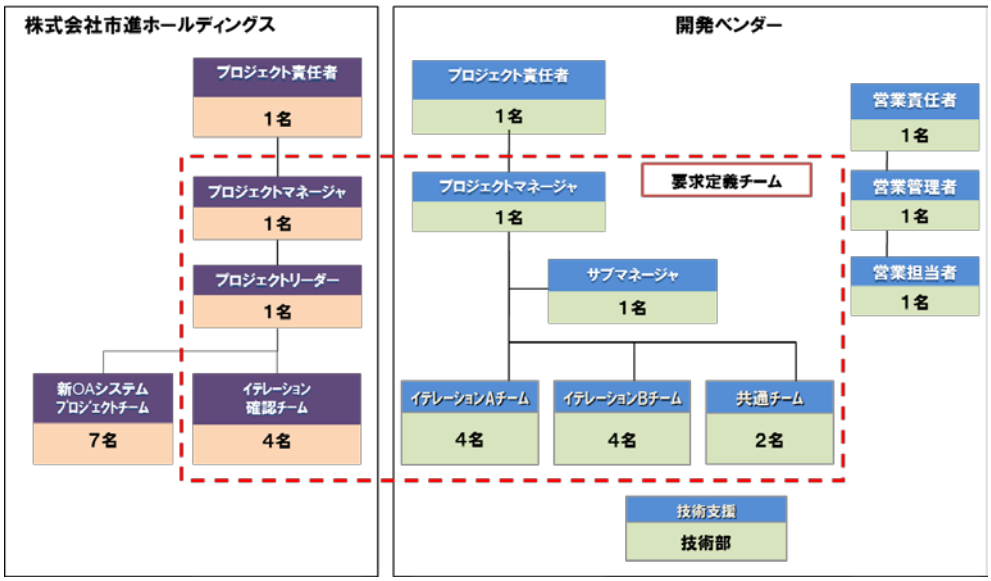

図 A-6-1　プロジェクト推進体制

4.3. 開発スケジュール

2010 年 11 月にシステム構築プロジェクトがスタートした。2011 年 3 月までは、業務フローなどを作成し、業務ルールを整理する要件定義フェーズに当て、4 月からアジャイル型開発が始まった。2012 年 5 月のゴールデンウィーク明けのサービスインを必須要件とした。今回の更改は、長年の細かい要望事項を吸収する必要もありシステム構築フェーズを細かいプロセスに分割し、短期間の開発サイクルを繰り返すアジャイル型開発を実践した。（図 A-6-2）

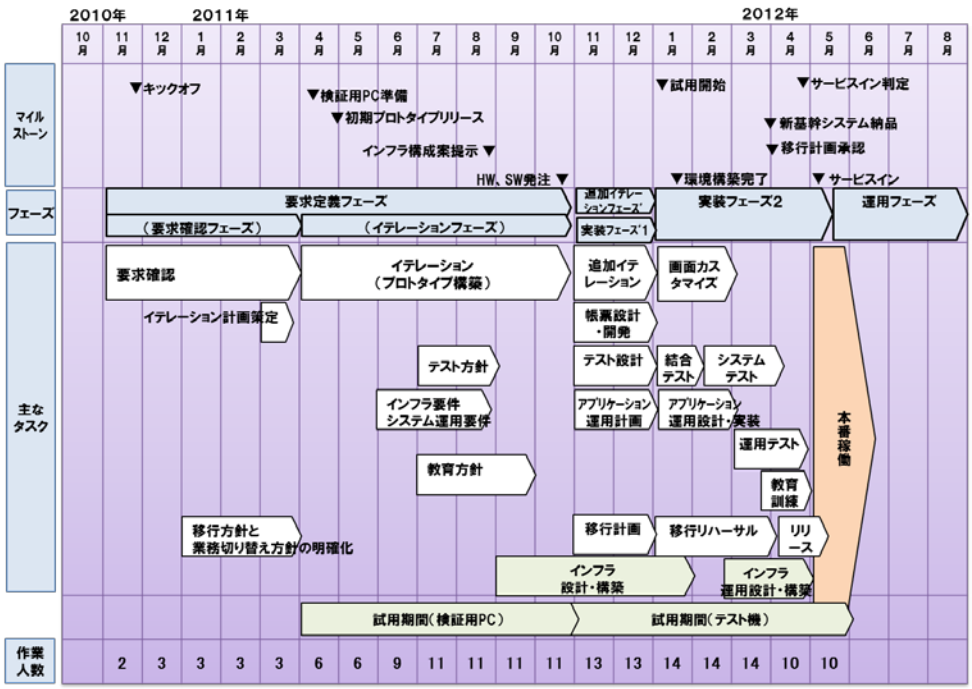

図 A-6-2　開発スケジュールと作業人数

4.4. 業務の整理と GeneXus の関係

要件を確定するため、開発実施前に要求定義チーム（図 A-6-1）を編成し、現行業務の整理を 5 ヶ月間にわたり実施した。

業務の整理を行うにあたり、

・　ステークホルダーの分析
・　業務機能構成図(DMM)
・　業務相互関連図

を作成し、同社とベンダー間で合意を得た。

ジェネレータツールの特長として、業務がすべて確定していなくても画面作成を行うことができる。そのためこの段階で詳細設計のようなレベルを意識せず、業務の流れを確認することに主眼をおいた。

業務の流れを確認する打ち合わせは主に同社システム部門とベンダー間で実施した。システム部門に現行業務の流れをある程度把握している人材がいたため、多くはシステム部門のみで対応し、不明点が出たときは各部署の担当者を打ち合わせに呼んで確認を行った。

業務の洗い出しは、以下のような方法で行った。

・　付箋を用いて業務を列挙する
・　列挙された付箋を大まかな業務別にグループ分けする
・　グループ分けされた付箋を業務の流れに合わせて分類する

その後､ジェネレータツールで画面生成を行いやすくするようにパワーポイントを利用し、

・　業務フロー
・　業務ルール

を作成した。

システムを利用しない部分についてもシステムに関連する部分があれば業務フローの中に組み込むようにした。この作業を行うことにより、作成すべき画面やバッチ処理の大半が明確になった。

この段階では、開発すべき業務の 7 割程度の把握に過ぎない。しかし、次のフェーズの打ち合わせで実際の画面を確認しながら進め、残りの部分を補完することができ、その状態で次のフェーズへと移行した。

一般的にシステム開発では､｢いかに業務に精通したユーザであってもシステム本番稼働時に必要なすべての要件を要件定義の段階で想定し決定することは難しい。動くシステムを見てはじめて不足している機能、不要な機能に気づく」と言われている。

同社の場合、業務の整理が 7 割程度でもジェネレータツールを利用した開発フェーズで実際に動く画面を見ながらシステム開発を行った結果、本番稼働時、全く機能しないシステムがなかった点は大変評価できる。その開発方法については次章で述べる。

4.5. GeneXus により可能になった 2 週間イテレーション開発

同社の以前までの開発ではウォーターフォール型の概要設計、詳細設計を経てからプログラム開発に着手していたが、今回は導入したジェネレータツールの特長を活かすため、イテレーション(反復)開発を実践した。

イテレーション開発期間はサービスイン日程を考慮し、28 週間に渡り実施することを事前に設定した。(図 A-6-3)

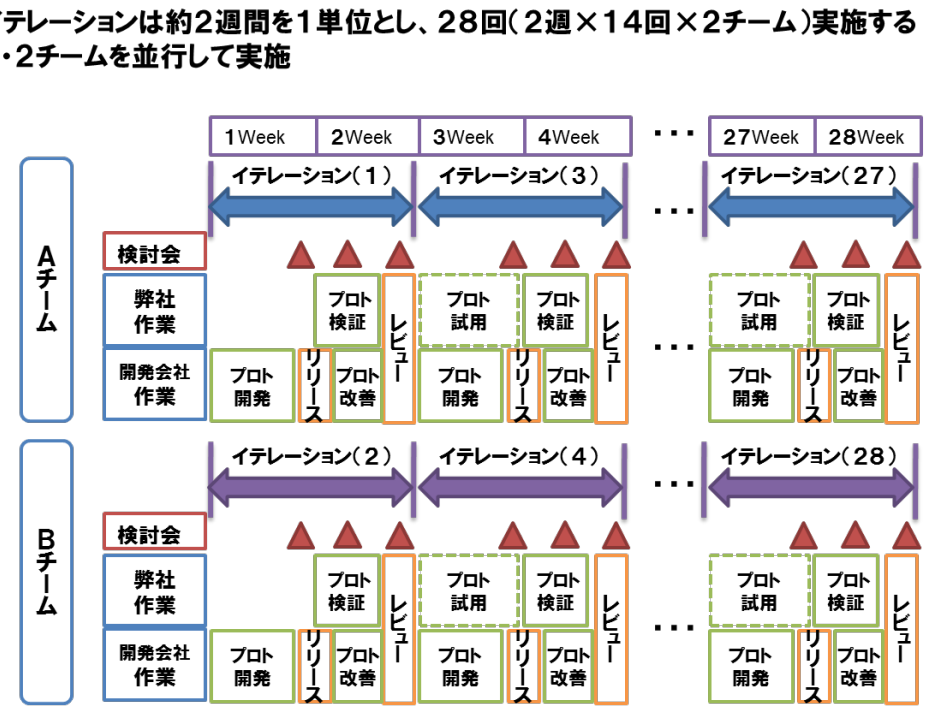

図 A-6-3　イテレーションの進め方

開発は 2 週間を 1 サイクルで打ち合わせを 3 回とし、開発期間を圧縮するため業務の異なるＡチームとＢチームが並行して開発を行う体制をとった。よって 2 週間の打ち合わせは最低 6 回行うこととなった。打ち合わせを 1 サイクル 3 回、開発会社のプロトタイプ開発後とプロトタイプ検証中とレビュー中に設定し、できるだけ業務理解のある同社 IT 担当と開発会社の担当とのコミュニケーションの頻度を高めた。それによって後戻りの可能性を少なくするように試みた。

打ち合わせでは、実際に動く画面と業務フロー、業務ルールを確認しながらそれぞれ同期がとれるよう修正しながら進めた。業務の内容が複雑な部分については、別途確認資料を作成し、参加メンバー全員のコンセンサスをとるようにした。

動く画面では、項目の過不足の確認と業務ルールに沿った入力手順、エラーチェックなどを確認した。イテレーションでは画面確認も同時に行うことができ、イテレーション終了時

点ではすでに単体テストがほぼ終了の状態となっていた。Ｖ字モデルでいう、ソフトウェア設計、プログラミング、ソフトウェアテストが同時に進んでいるイメージであった。

5. 運用環境の準備

5.1. ハードウェアの選定

主に以下の観点から選定作業を実施した。

- 価格性能比
- 保守性
- 同時アクセス数
- データ容量(数年分)

ハードウェアは数年で更新が必要であるので、データ容量については数年後までを見込んで考慮した。

また、昨今の自然災害や社会情勢を踏まえ、これまで事務所内に設置していた汎用機をサーバに変更し、ＢＣＰ(事業継続計画)を考慮しデータセンターへ設置する方針とした。データセンターの選定にあたっては、

- 事業所に近い
- 最新設備が整っている
- 費用が妥当である

の点を考慮し選定した。特にデータセンターの利用が初めてということもあり、緊急時に素早く対応できるよう事業所に近いという条件を重視した。

5.2. データ移行作業

データ移行については、以前のシステムベンダーに依頼する予定であったが、費用面で折り合いがつかず、ユーザシステム部門で移行作業をすることとなった。

移行は、旧システムのデータをテキストデータに変換し、VB(ビジュアルベーシック)を利用してレコードレイアウトを作成し直した。

また､システム間でバラバラだったコード体系を､変換テーブルを作成し同時に統一した。

今回のジェネレータツールはプログラム生成時に同時にデータベース構造を再編成する機能を持っている。そのため、仕様変更によりデータベース項目に変更がある場合、レコードレイアウトが変更になり、稼働直前まで VB を作成し直すという作業が発生した。

このデータ移行作業については、ユーザシステム部門のみで実施せず、ベンダーの協力を仰ぐべきであったと反省している。なぜならば、内部で作成した移行データの不具合によりデータの整合性がとれないことが原因で入力エラーが発生し、データのメンテナンスが必要になった状況が複数回発生してしまったからである。

5.3. エンドユーザ教育

エンドユーザへのシステム変更の説明は稼働 2 週間前に実施した。集合形式での説明を 2 時間程度実施した。

データの整備が稼働直前となってしまったため、一部のエンドユーザに対しては事前利用環境も提供できず、マニュアルと映像のみでの説明となってしまった。

今回、体制不備でエンドユーザの教育の遅れが稼働後の操作不案内の不満につながったことは否めない。早期にデータを整備して試用期間を十分とるべきであった。

5.4. システムの切り替え

旧システムから新システムへの切り替えは、5 月の連休を利用して実施した。

- 旧システムが複数存在し同期がとれない
- データの更新がデイリーで発生する
- 並行稼働はエンドユーザの負担が大きい
- 事前の切り替えテストはほぼ成功した

などの点を考慮し、連休明けのシステムの一発切り替えを敢行した。万一の場合は旧システムをそのまま稼働できる体制はとっていたが、切り戻しの必要な状態にまではなることはなく新システムの運用を予定通り進めることができた。

このようにシステムの切り替えをスムースにできたのは、以下の 2 点が起因していることも大きい。

① データ移行リハーサルを事前に 2 回実施し、データ不整合の問題点を事前に解消したこと

② ゴールデンウィークの 6 日間を全て移行期間に当てられ、検証時間を十分に確保することができたこと

6. 取組みの結果

とはいえ、システム稼働直後は仕様の食い違いやバグでシステムの修正が発生した。

今回は今までにない新しいツールを活用することになり、今までの取組み結果や予定とは大きく違う結果が出た。それを表 A-6-1 にまとめた。

ジェネレータツールは業務ルールを変更し、再ビルドすることでプログラムやデータベースを変更できる。その特性を利用して、仕様に合わない部分やバグの大部分を発生翌日には変更できた。

ジェネレータを利用しない開発では、データの関連や他のプログラムへの影響調査が必要なため、通常翌日の修正は大変厳しいものとなっている。その修正を簡単にできる点では期待通りの高生産性を実現した。

また、エンドユーザから標準画面での操作性について厳しい意見のあった部分についてはカスタマイズを実施した。しかし、ジェネレータツールでカスタマイズできる範囲内にとど

め、後々の仕様変更に素早く対応できるように配慮した。なぜならば、ソースコードに直接修正を加えると品質を担保できなくなるため必ずジェネレータツールを通して変更することを死守した。

これらの結果から今後の取組みが浮き彫りにされたため、更なるこのツールの特長を活かした保守運用を進めていくとのことである。

表 A-6-1　取組み予定と結果・今後に向けて

予定	結果	今後
仕様不整合やバグ対応は、設計確認からプログラム改造迄の相当の期間が発生すると予想（内容によって長短はあるが）	仕様に合わない部分やバグの大部分を業務ルールを変更し、再ビルドすることでプログラムやデータベースを発生翌日には変更できた	①内製化への取組み
エンドユーザからの要求（変更）依頼も上記のバグ対応と同様に相当の期間が発生すると予想（内容によって長短はあるが）	要求（変更）事項は、ツールのカスタマイズできる範囲内にとどめ、エンドユーザの要求を100%反映しない様にした。ツールの『優れたメンテナンス性』を守ることを優先した	②容易なシステム変更での取組み
新規システムの費用は、既存のシステムのプログラム開発分などは一部流用可能としてもインフラ分は新規費用発生と試算	通常開発規模をおよそ300人月相当と想定していたが、このツールを利用したことで当初試算の5割から6割程度で済んだ。 今後は維持管理費用は大幅に削減できる。	③未来を守る非IT環境依存型システムの構築

7. 今後の取組みと考察

7.1. 内製化への取り組み

ジェネレータツールのもう一つの特長として、技術習得の容易性が挙げられる。

今回の開発では、メンバーにジェネレータツールの作成技術を学ばせたところ、1 ヶ月程度の期間で習得が可能であった。なぜこのような短期間で習得できたかと言うと以下の３点に依るところが大きい。

① 日頃対応している業務に関する記述が中心で、短いステップ数で記載できること
② データベース定義を意識せずに記述できること
③ プログラム言語やプログラムのロジックに精通していない社員も業務や機能を理解しているだけでシステムを生成することができること

現在、この特長を活かして内製化を進めている。

内製化により当初の目的の一つでもある「業務の改革にシステムが追随する」ことが容易にできるようになりつつある。

7.2. 容易なシステム変更での取り組み

業務は常に変更を伴うものであり、システムは業務の変更に追随する必要がある。追随が遅れたりシステムの変更に多額の費用がかかる場合、売上げの機会損失につながったり、会社の利益を圧迫したりする可能性がある。

今回は､ジェネレータツールの特性を最大限に活かしてほぼ標準仕様で構築しているため、これまでのシステムに比べて以下の長所がある。

- システムの変更にかかる時間が劇的に短縮された
  具体例としてこれまででは 3 ヶ月程度かかる変更が 1 週間で完了する。
- システムの変更に躊躇がいらなくなった
  内製化を利用して変更するため、費用・時間の面で短縮され、必要な変更は必ず実施するようになった。それまでは現状のシステムで工夫して実施していた。

「日々進化するシステム」を目指して新たな開発に取り組んでいく予定である。

7.3. 未来を守る非ＩＴ環境依存型システムの構築

今後、何らかの事情により、インフラ環境を刷新してハードウェア環境が大きく変わったとしてもシステム上最適な環境を利用できる。それは、導入したジェネレータツールが非ＩＴ環境依存型システムを前提としているからである。非 IT 環境依存とは、多様で変化の激しい技術や将来の技術革新への対応を GeneXus ツールが吸収するので、メンバーは GeneXus 技術や使い方のみを習得していれば、将来にわたってシステム開発・保守を非 IT でも可能になる。

また、GeneXus は世間一般に利用されている標準的なプログラム、言語、データベースに対応しているため、新しい今までと違ったインフラ環境を導入してもプログラム生成先を自由に選択するだけで今まで同様にシステムが稼働できる、このように柔軟性に優れている。

また、ハードウェアの老朽化やＯＳのサポート終了などでシステムの載せ替えを余儀なくされたときに、以前のシステムと違い、サーバＯＳのバージョンや種類、データベースのバージョンを意識することなくスムースに業務アプリケーションを移行できることが推測される。

A－7

設計工程における TERASOLUNA DS の適用[1]

1. 概要

本編では、上流工程でのソフトウェア設計に関わる取組みとして、株式会社エヌ・ティ・ティ・データ（以下、同社とする）の事例を紹介する。

システム開発の上流工程である設計工程で、作るべきシステムの詳細な定義を漏れなく行うことは重要なことである。設計工程の品質向上のためには、下流工程に設計バグがすり抜けない様に設計工程の中でレビューを行うのが一般的であるが、昨今の開発現場ではそのレビューが適切に機能していないのが現状である。同社では、下流工程へのバグのすり抜けの原因は、「設計書量が多く、開発期間が短いために、設計書間整合性を確保することが人間のレビュー作業では不可能」と結論付けた。そして、それを解決するために設計支援システムを構築している。このシステムは、設計書を検索可能な情報として DB で管理し、対象の設計書の各設計項目単位で検索を行う機能（整合性チェック機能）とサービス開始後の保守維持のために利用される影響調査効率化機能で下流工程へのバグのすり抜けを防いでいる。

同社では、当該システムを 2012 年から累積で 50 を超える開発プロジェクトに導入し、現在、その効果分析を実施している。本事例では、過去に終了した本システムの未導入プロジェクトのデータを元にどの程度の工数削減が見込めるかを予測している。

以下の同社から提供された事例には、本システムで利用された技術やシステム機能を詳細に記載されているので参考にしてほしい。

2. はじめに

（株）NTT データ（以下、当社とする）では、現在、「生産技術革新」と謳いシステム開発、その中でも特にソフトウェア開発に関わるすべての工程での改革を進めている。

当社では、従来から開発の標準化や品質保証活動などを行ってきたが、昨今のシステムの大規模化と開発期間の短期化に対応するためにはこれらの活動だけでは不十分であり、工程の作業そのものを人手に依らない機械化、自動化することで抜本的に解決することを狙っている。

今回は、その中で、上流工程でのソフトウェア設計に関わる取り組みについて紹介したい。

---

[1] 事例提供：株式会社エヌ・ティ・ティ・データ　冨安 寛 氏

3. 取組の目的

3.1. 上流工程の課題

ウォーターフォール型のシステム開発における上流工程の重要性は過去から言われてきた。特にシステム開発の目的を明確にするための要件定義工程は、その失敗がシステム開発全体に影響することから様々な手法が開発されて利用されている。また、上流工程でのバグが下流工程にすり抜けてしまうと、その修復のための費用は上流工程で修復するのに比して何倍にもなることは古くから良く知られている[1]。本稿では、この問題に対する取組について説明する。

当社では、社内で標準化されている開発プロセスで、各工程におけるレビュー作業を規定している。このレビュー作業によって、上流工程における設計書のバグを下流工程にすり抜けさせないことを狙っている。恐らく多くのシステム開発現場においても、同様の作業が規定されて、実施されているであろう[2]。

しかし、昨今のシステム開発では、人海戦術のレビュー作業に期待するだけでは手に負えない状況が発生している。例えば、図 A-7-1 は当社において開発されたあるシステム開発プロジェクトの情報量を示したものである。このシステムは、いわゆる一般的な Web システムであり、主に Java 言語を使って製造された。開発期間 2 年、総工数 2 万人月を要して完成している。図は論理的なユーザの要求から実際にシステム上で稼働する物理的なソースコードや DDL に至るまでに情報量が増加していくことを模している。中に記したように、総設計書数は Microsoft Excel 形式で 4 万ファイル、総ページ数は 40 万頁にも及び、結果として製造されたソースコード類は 17 万ファイル、2M ステップとなった。

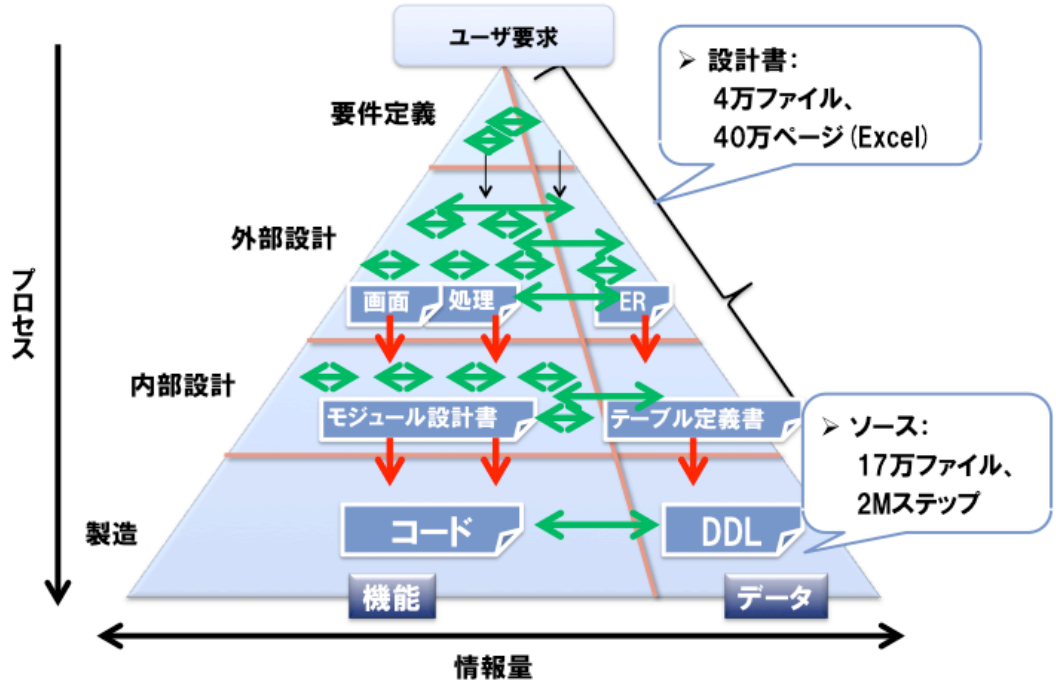

図 A-7-1　プロジェクトにおける情報量

図 A-7-1 中の左右の矢印は各工程において、設計書間で整合性をはかるべきことを意味している。上下の矢印は工程間を跨ぐ整合性、すなわちトレーサビリティを確保すべきことを意味している。どちらの整合性も人間の目視によるレビューにより確認することが開発プロセスにより求められているが、このように膨大な量の設計書やソースコードを人間の目視で厳密に確認するのは実効的に不可能であろう。しかし、仮にレビューが不十分で、図 A-7-1 中にあるような、画面、処理、ER（データモデル）間で不整合が発生していても、それぞれの製造を行うことは可能である。そのため短期間で、少しでもプロセスを先に進めたい圧力がかかるプロジェクトでは、不整合を孕んだままプロセスを進めてしまう。その結果、その不整合はさらにプロセスが進んだ結合試験工程で発現することになる。例えば「画面からボタンを押下しても処理が動かない」「想定していたデータベースのテーブルがない」などのバグとして初めて確認され、その修復には上流工程での修復の何倍もの費用が掛かることになる。

以上のことを簡潔にまとめると、昨今のシステム開発における下流工程へのバグのすり抜けは、上流工程での次の課題が原因となっていると言える。

- **設計書量が多く、開発期間が短いため、設計書間整合性を確保することが人間のレビュー作業では不可能である。**

3.2. 課題解決のための前提条件

前節で示した課題を解決する、もしくは発生させないには様々な解決策が考えられる。例

えば、すでに不整合が発覚しているプロジェクトではレビュー要員を増員して人海戦術で対応を試みることが一般的である。それ以外にも、すでに様々なツールや製品が提供されている形式手法や Model Driven Architecture を採用し、厳密な設計、開発を進めることも考えられるだろう。しかし、いずれの方法も現在のシステム開発を取り巻く環境の変化を考慮しておらず、部分的な解決しか実現できていない。もしくは解決に至るまでに多くの労力を必要とする。解決策を模索するにあたって、考慮すべき前提条件は以下がある。

(1) 多くのプロジェクトで多くの開発者が利用することを鑑み、修得にスキルや時間を要しないこと。
(2) 開発拠点が分散しており、開発拠点間の情報伝達や成果物の完成は一定期間毎に行われるため、常時の相互レビューが困難であること。
(3) 設計書の種類により進捗が異なり、結果完成タイミングが異なること。

それぞれについて簡単に解説する。

（1）については、次の通りである。当社ではソフトウェア開発を伴うシステム開発が年間 1,000 プロジェクトを越える。これらのプロジェクトには、当社の社員のみならず、協働してくれる国内外の多くのソフトウェア開発会社社員が関与する。これらのことを鑑みると、解決策は修得に時間を要するものや、特殊なスキルを必要とするものであってはならない。また、システムは開発するだけではなく、保守・維持するものであり、長期間に亘りこの解決策は利用されることが想定される。その間には要員の交代も起こりえるが、そのような状況にも耐えられることも考慮する必要がある。

（2）については、次の通りである。現在のシステム開発は 1 カ所で開発されるのではなく、複数拠点での分散開発が一般的である。特に昨今はオフショア開発が盛んに行われており、中国や東南アジア、インドでも開発の一部が行われるようになった。特に中国では日本語を解するメンバの調達も容易になっているため、上流の外部設計工程からの開発委託も増加している。このような分散拠点での開発においては、各拠点毎にある一定のまとまりを以て開発を委託することが一般的である。例えば、あるサブシステムの画面・処理、共通処理、画面に関する設計～単体試験、処理に関する設計～単体試験、データベース設計～製造など分割の方法は様々である。それぞれの開発拠点はネットワーク的に接続され、リアルタイムで情報連携できる拠点ばかりではなく、成果物はある一定の期間をおいて、集約することもある。そのため、その期間中は複数拠点の成果物間の整合性を確認することができず、潜在的に不整合が発生することになる。例えば、あるサブシステムと全サブシステムで利用する共通処理を別拠点で設計する場合、サブシステム開発側では共通処理の初期の設計書を基に共通処理を呼び出す設計を行うが、共通処理側で設計が変更されてもその情報は伝達されない。そのため、両者間に不整合が生じる。この不整合はその後両者の設計書が集約され、レビューを経た後にサブシステム開発側に対して修正依頼が掛けられることになる。このような例は開発期間中には挙げればきりがなく、解決策は一定期間の不整合を許容し、集約され

た時点で効率的に整合性の確認を行える方策が望ましい。

（3）については、次の通りである。システム開発における最終的な成功はユーザの業務を効率的に行えるシステムを作り上げることにある。そのため、開発期間を通じてユーザの関与と意志決定が欠かせないが、画面、処理、データベースのそれぞれで、ユーザの意志決定が簡単なものと難しいものがある。例えばデータモデルの設計は、開発対象のシステムだけを考えると容易な作業だが、企業内の他システムも全て考慮すると、多部門に跨がる多くの確認作業や意志決定が必要となり、容易には決定できないものになる。そのため、設計工程終了時も暫定版となり、最終決定は後工程でなされる場合も多い。この場合、他の画面や処理は暫定的に設計を進めることになり、その期間中は 3 者間の不整合が発生する可能性がある。これ以外にも、開発期間中に画面、処理、データベースのいずれかに仕様変更があり、仕様が確定するまでの間は他の設計に不整合が発生することになる。このような状態に対応できるよう、解決策は設計書の種類間の不整合発生を許容し、最終的な仕様確定を待たずに、作業を進められる方策であることが望ましい。

4. TERASOLUNA DS の開発

前章の課題を解決するために、前提条件を考慮して、当社では、TERASOLUNA DS（テラソルナ・ディーエス）と呼ぶ設計支援システムの開発を行った。本章ではその解説を行う。

4.1. 機能について

TERASOLUNA DS は開発プロジェクト毎に準備され、図 A-7-2 のような機能をプロジェクトに対して提供する。紙面の都合上、システム構成の詳細は省略するが、本システムは、設計書を設計書 DB に取り込む機能と、その DB に対して開発者が用途に応じて各種の検索を行う機能、整合性チェックと影響調査効率化の機能に大別される。以下に本システムの各機能の概要を解説する。

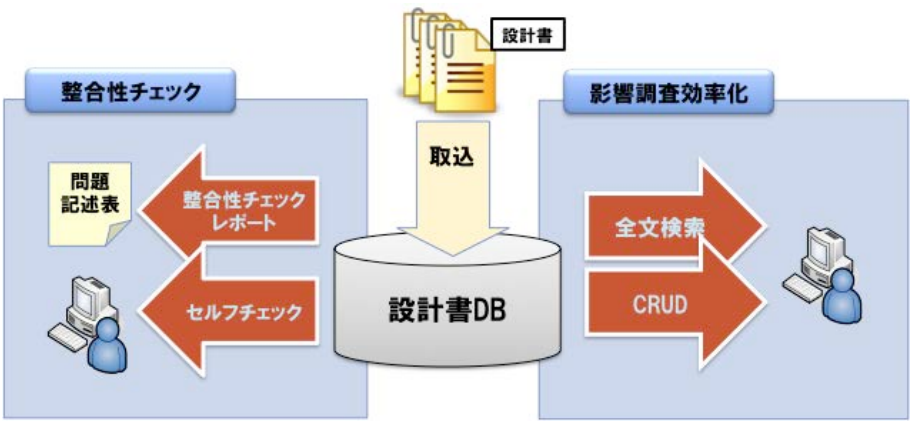

図 A-7-2　TERASOLUNA DS の機能

4.1.1. 設計書取込機能と設計書 DB

3.2 で昨今の開発が複数拠点にて行われることを説明したが、設計書や製造されたソースコードは最終的には 1 カ所に集約される。主に開発の中核となる拠点にファイルサーバや構成管理システムが設置されて正本管理されるのが一般的である。設計書取込機能はこの正本管理された設計書を定期的にバッチ処理にて設計書DBに取り込む。設計書DB内において、各設計書に書かれている設計項目は一つ一つが検索可能な情報として管理される。取り込み頻度は開発プロジェクトの状況を考慮して設定することができるが、日次での処理を推奨している。

4.1.2. 整合性チェック機能

本機能が本システムの中核機能である。

設計書 DB に蓄積された情報は、設計書の各設計項目単位で検索可能な状態にある。この情報に対して、整合性を確認するためのルールに基づいた検索を行う。ルールの一例を以下に示す。

例 1　入出力項目の整合性確認

データ授受を行っている機能間の「入力項目」と「出力項目」とを突合し、項目名と属性（型桁等）が一致していることをチェックする。処理設計書間、処理設計書-画面設計書間、処理設計書-テーブル設計書間、他で行う。

例 2　設計書が揃っていることを確認

機能一覧に記載された全ての機能に対して、対応する設計書が存在しているかをチェックする。

例 3　命名規約の確認

画面上に配置されたボタン名が、すべて命名規約どおりに記載されているかをチェックする。

上記ルールのうち、例 1 に関して、具体的な例を示す。図 A-7-3 は設計工程で作成する 3 種類の設計書間での整合性を確認する例である。

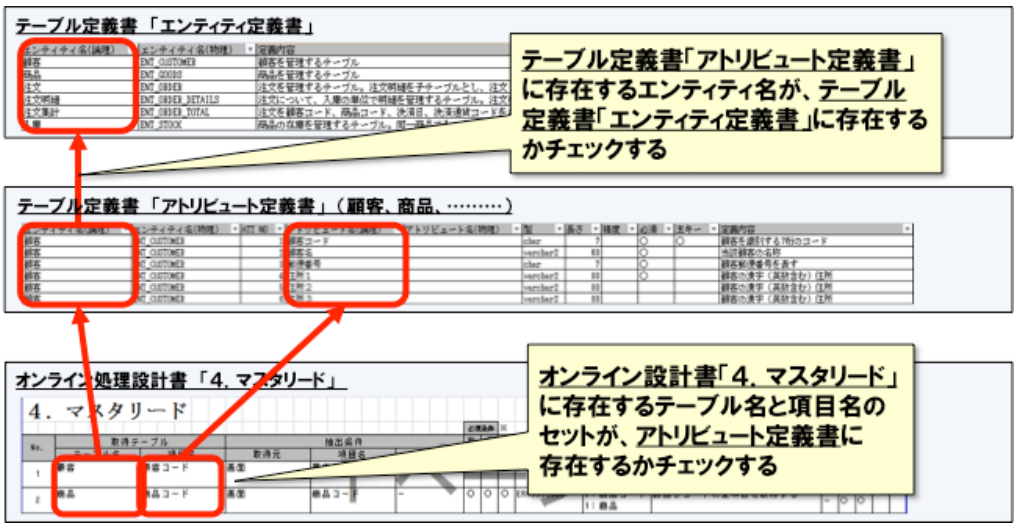

図 A-7-3　整合性チェックの例

テーブル定義書「エンティティ定義書」、テーブル定義書「アトリビュート定義書」はデータベースの内容を設計するために作成される。オンライン処理設計書「4. マスタリード」は処理内容を設計するために作成される。それぞれは独立に作成されるため、この例ではその内容がお互いに合致しているかどうかを確認している。

これらのルールは標準で約 337 種類提供されており、開発プロジェクトはそれらを利用することで一定の整合性チェックが可能になる。また、独自にルール作成も可能であり、その作成機能も提供されている。

この整合性チェック機能は、整合性チェックレポート作成機能として全ルールをバッチ処理によって実行し、不整合を問題記述表として出力するか、セルフチェック機能として開発者自らが作成している設計書が他の設計書との間で矛盾を有していないかを画面上で確認するかの 2 つの機能を利用することが可能となっている。図 A-7-4 は出力された問題記述表の一例である。不整合の内容を「メッセージ」に記載するようになっており、日本語として判りやすい文章を出力できるように工夫している。

不整合内容									
項番	チェックルール							チェック対象項目	
	レベル	ID	名前	メッセージ	ファイル名	シート名	Rev	項目名	内容
1	Warning	様式506-単01	必須入力(セル)	02-2_内部取引設計書_基本情報シート(A).xlsのシート「内部取引設計書_基本情報」の「章・節・項」が空です。	02-2_不整合1_内部取引設計書_基本情報シート(A).xls	2_基本情報	0	章・節・項	
2	Error	様式506-単07	リスト(パターン9)	02-2_不整合1_内部取引設計書_基本情報シート(A).xlsのシート「内部取引設計書_基本情報」の「応答要否」がリスト「○, ×, -」に含まれていません。	02-2_不整合1_内部取引設計書_基本情報シート(A).xls	2_基本情報	0	応答要否	*
3	Error	様式506-相03	必須入力(セル)	02-2_不整合1_内部取引設計書_基本情報シート(A).xlsのシート「内部取引設計書_基本情報」の「処理順序」が空です。	02-2_不整合1_内部取引設計書_基本情報シート(A).xls	2_基本情報	0	処理順序	
4	Error	様式506-相04	重複(複数項目)	02-2_不整合1_内部取引設計書_基本情報シート(A).xlsのシート「内部取引設計書_基本情報」のワークフローコード, 処理順序「BF000001, A-2」が重複しています。	02-2_不整合1_内部取引設計書_基本情報シート(A).xls	2_基本情報	0	ワークフローコード 処理順序	BF000001 A-2
5	Caution	様式506-相22	存在	02-2_不整合1_内部取引設計書_基本情報シート(A).xlsのシート「内部取引設計書_基本情報」の呼び出し先オンラインワーク名／ワークフローコード「BF000004」に対応する設計情報が、設計書様式「内部取引設計書_基本情報」の設計書に存在しませんでした。	02-2_不整合1_内部取引設計書_基本情報シート(A).xls	2_基本情報	0	呼び出し先オンラインワーク名／ワークフローコード	BF000004

図 A-7-4　問題記述表

4.1.3. 影響調査効率化機能

本機能はシステム開発時のみではなく、サービス開始後の保守維持のためにも利用されることを想定して提供されている。ただし、本稿の主旨とは異なるので、本機能については概要を説明するに留めたい。本機能は、仕様変更が発生した時に、変更に該当する設計書だけでなく、関連する設計書を漏れなく全て探し出すための機能である。設計情報は、設計書 DB 内に設計項目単位で細かく管理蓄積されているので、様々な条件での検索を行うことで、効率的に仕様変更によって影響を被る設計書を発見することが可能になる。

4.2. TERASOLUNA DS を実現する方法

本節では前述した機能を実現するために必要となる技術などについて説明する。

4.2.1. 設計書フォーマット標準化

TERASOLUNA DS の機能を実現するためには、まず設計書フォーマットが組織内で標準化されている必要がある。すなわち、システム開発で利用する各種設計書はその様式が定められており、また、組織内の各プロジェクトがその設計書を使用してシステム開発を実施することが前提である。標準化されておらず、プロジェクト毎に様式が異なっていると、4.1.1 で紹介した設計書取込機能を実現するにあたり、プロジェクト毎に機能の作り込みが発生し、コストパフォーマンスが大変悪くなる。

次に、標準化されている設計書は可能な限り形式化されている必要がある。すなわち、各設計情報の記述においては、可能な限り文章による自由な表現を廃しておくことが望ましい。例えば、表形式によって記述するなどである。

当社では、上記 2 つの条件に対して次のように対応している。まず、設計書フォーマットは、社内のシステム開発標準である TERASOLUNA 開発プロセスで定められているフォーマットが広く利用されている。そのため、TERASOLUNA DS でも、その利用を前提としている。当該フォーマットは、画面設計、処理、データモデルなどシステム開発に必要な全ての設計書毎に用意されている。次に、設計書フォーマット中の設計項目は一部の項目を除き表形式で記述することを定めている。表形式での表現が難しく、文章での記述が許されている設計項目については、残念ながら整合性チェックの対象とすることが難しいのが現状である。また、当該フォーマットは Microsoft Excel のファイルで当社社内に配布されている。Microsoft Excel で記録された設計書では、設計項目は各シートの決まったセルに記述されており、その後の解析が容易である。

なお、当社での利用において、各プロジェクトで設計書フォーマットの変更はほぼ間違いなく発生している。しかし、それは標準フォーマットの構成を大幅に改変するものではなく、設計項目の追加がほとんどである。Microsoft Excel のフォーマットでは表中の行や列の追加に相当し、4.1.1 で述べた設計書取込機能にカスタマイズ機能を具備する事で対応している。

4.2.2. DBMS

設計書情報を蓄積する設計書 DB には一般的な DBMS が利用可能である。ただし、前節でも述べたように、開発プロジェクトでは設計書フォーマットの改変が発生する。その際には併せて設計書 DB のデータスキーマも変更されることになる。開発プロジェクトによっては設計書フォーマットの改変をかなり頻繁に行う場合があり、DBMS のスキーマ変更と、その後の新スキーマに合わせたデータの書き換えが迅速に行えないと、その間設計書 DB の利用ができなくなる。よって、データスキーマの変更と、同時に蓄積されているデータの書き換えが柔軟かつ迅速に行える事が必要である。

当社では、TERASOLUNA DS の DBMS に XML DB を利用している。XML DB は広く利用されている RDBMS と異なり、データスキーマの変更処理とデータの書き換えを必要としない。すなわち、新しい設計項目が追加された場合、そのデータは XML DB へのロード時に即時解析され、検索用のインデックスが付与される。それまでに蓄積したデータの書き換えも発生しない。よって、上記の要件を満たしている。

4.2.3. 整合性チェックルール整備

設計書 DB に蓄積された設計書情報に対して、4.1.2 で述べたようなルールを検索処理として実行する。このルールをどれだけ整備できるかが品質向上の鍵になる。このルールを整備するには次の点を考慮する。

(1) 開発標準に含まれている設計書レビュー観点
(2) プロジェクトに蓄積されているレビューノウハウ
(3) 結合試験時にバグとなることが予想されるインタフェースに関する設計情報

（1)、(2）は、過去から組織内に蓄積されているノウハウである。これらをできる限り集めることが重要である。ただし、これらのノウハウは文章で記述されていることが多い。設計書を対象にしたルールとするためには、具体的な設計書フォーマットとそこに記載されている設計項目が定義できていないと難しく、4.2.1 の標準化が前提となる。当社では、TERASOLUNA DS の開発にあたり、TERASOLUNA 開発プロセスで具備している設計書レビュー観点を、標準化されている設計書フォーマットで具体的に検討してルールを整備した。標準ルール内において、(1）に関しては 199 のルールが最初から設定されている。(2）に関しては、各プロジェクトにおいて独自に設定することとしているため、標準ルール内では設定していない。

（3）は後述する TERASOLUNA DS の効果に直接つながる重要なルールの整備である。設計書レビュー時に CRUD（Create、Read、Update、Delete）などの各種インタフェースの整合性を確認しておくことが相当する。当社では、過去のシステム開発プロジェクトの結合試験時のバグ票を収集、分析して、ルールとしており、標準ルール内で 138 のルールが設

定されている。

4.3. TERASOLUNA DS の運用

運用については開発プロジェクト毎に状況が異なるために一概には決めきれないが、ここでは想定している標準的な運用について解説する。

設計書は設計書取込機能によって、一定の頻度で設計書 DB 内に蓄積される。この処理は日次で行われ、1 日の作業が終了した後の夜間バッチ処理で実施される。その後、続けて夜間バッチ処理にて整合性チェックを行い、問題記述表を翌日の開発作業開始前までに作成する。開発者はこの問題記述表を確認し、不整合を解消するように設計作業を行う。設計書完成後はセルフチェックを行い、最終的にファイルサーバもしくは構成管理システムに保存される。この一連の処理と設計作業を不整合が完全に解消するまで実施することで、設計工程は完了する。

ここで、3.2 で採り上げた前提条件（1）～（3）について検討してみる。（1）については、開発者が対峙するのは特別なインタフェースではなく、通常と変わらない Microsoft Excel の設計書フォーマットのため、特別なスキルや修得時間は発生しない。（2）と（3）のように、全ての設計書が一様に整備されない場合については、多少の考慮が必要である。例えば一部の設計書群がオフショアで作成されていた場合、これらの設計書が当システムに投入されるのは１ヵ月に１回ということもありえる。その１ヶ月間においては、他の設計書だけで整合性チェックが行われることになるが、不足している設計書群との関連がある全ての設計書において不整合が検出され､日々問題記述表に出力されることになる｡このような状態で、設計作業を進めることは勿論好ましいことではない。しかし、重要なのは「一部の設計書が不足、もしくは不正確であることによる不整合の箇所が漏れなく特定できていること」「一部の設計書が不足、もしくは不正確であっても他の設計作業の進捗に影響を与えないこと」であると考えている。よって、この場合は不足している設計書が取り込まれた後に整合性チェックを行い、十分に設計書を修正する時間を工程の中に織り込むことで対応すれば問題は発生しない。

4.4. 効果の目論見

TERASOLUNA DS を開発プロジェクトに導入した場合、得られる効果についてモデル化を行っている。これまで設計工程時の不整合の発見について説明してきたが、効果は後工程で得られることになる。図 A-7-5 はその効果を示したものである。

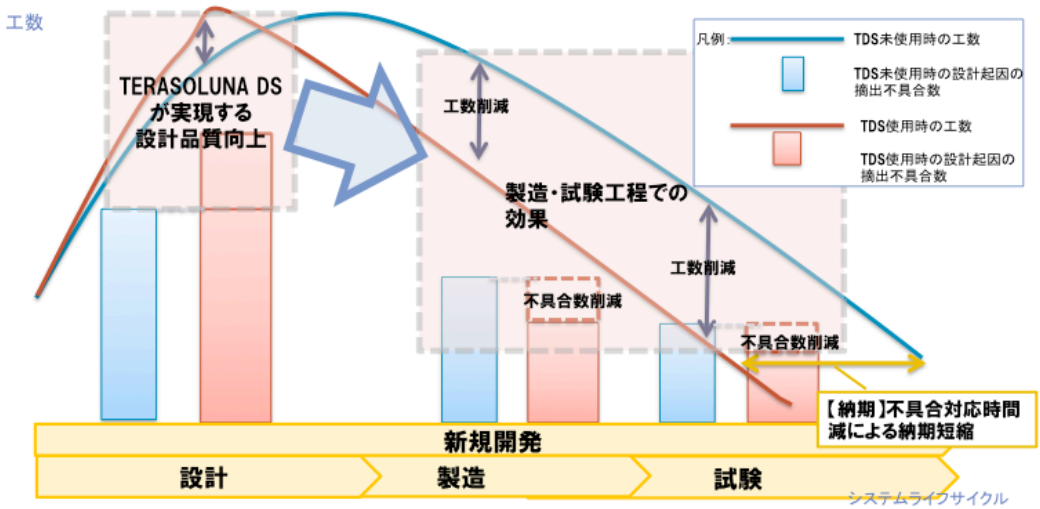

図 A-7-5　TERASOLUNA DS の効果

本システムを利用することで、利用しなかった場合に比して、設計工程時に発見される不具合数は増えると予想される。その結果、その修正のために工数も同様に増えることになる。しかし、その後の製造工程や試験工程にて、不具合数は削減される。後工程での修正工数は設計工程でのそれに比して大きくなり、それが削減されることになるので開発工数の削減と、さらには開発期間の短縮が見込まれると予想している。

5. TERASOLUNA DS の開発プロジェクトでの適用

TERASOLUNA DS は当社社内で 2012 年度からの累積ですでに 50 を越える開発プロジェクトで導入されている。これらのプロジェクトにおいて、4.4 で説明した目論見が得られるかどうかについて現在検証を進めている。長期的なプロジェクトが多いため、現時点では最終的な評価結果は得られていない。この結果については稿を改めて紹介したい。

そこで、ここでは、過去に終了した TERASOLUNA DS 未導入プロジェクトのデータを用いて、仮に TERASOLUNA DS を導入した場合にどの程度の工数削減が見込めるか、を紹介したい。

当該プロジェクトにおいて、試験時のバグ発生による不具合対応工数を精査したところ、単体試験、結合試験、システムテストそれぞれにおいて、各工程内において、手戻り工数の割合が、1%、37%、1%であることが判った。そこで、単体試験とシステムテスト実施時の不具合工数については無視できると仮定して、結合試験時の不具合工数を対象にさらに分析を行った。

図 A-7-6 は当該プロジェクトの結合試験時に発生したバグ原因を分類したものである。

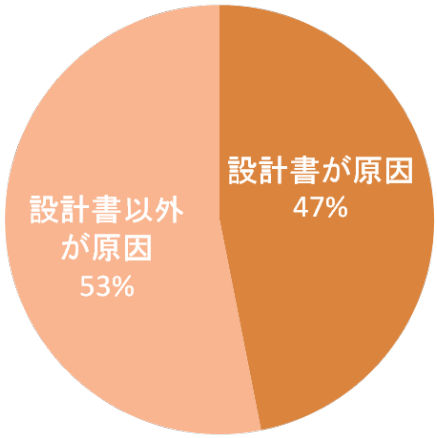

図 A-7-6　結合試験時のバグ原因

図中、「設計書以外が原因」の意味する所は、設計書は正しく記述されているが、仕様として正しくないものや、ソフトウェア製造時にミスがあったものなどである。つまり TERASOLUNA DS では抽出不可能なエラーである。「設計書が原因」は TERASOLUNA DS で抽出可能なエラーであり、設計書間で不整合が発生している場合である。それぞれ結合試験時のバグの 53%、47%を占めている。

仮にこの 47%のバグを TERASOLUNA DS を利用して設計工程で発見でき、結合試験工程にすり抜けないと仮定し、さらに 1 件あたりの不具合対応工数を同一と仮定すると、先の結合試験時の工数の 37%中 47%は削減することが可能である。すなわち、結合試験で従来必要とされる工数の約 17%は削減ができる。

今後、TERASOLUNA DS が導入されている実際の開発プロジェクトにおいて、結合試験時に「設計書が原因」のバグが発生しているかどうかを追跡し、上記の効果を確認していく予定である。

もう一つ別のデータも紹介する。前述したプロジェクトとは別のプロジェクトの設計工程時に TERASOLUNA DS を導入した。その結果、5,033 件のエラーが不整合として抽出された。そのエラーを一件ずつその後の工程にどのように影響を与えるかを確認し、分類した結果を図 A-7-7 に示す。

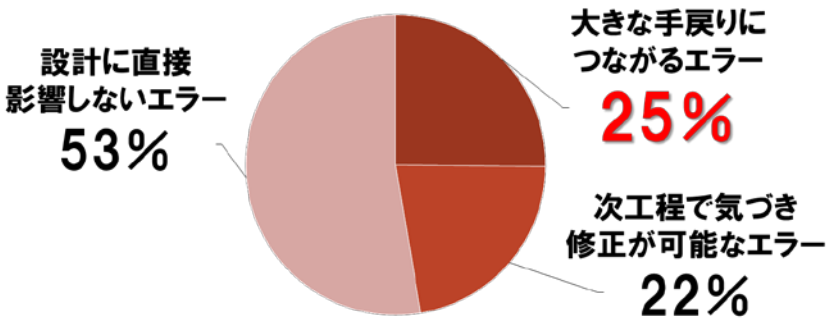

図 A-7-7　TERASOLUNA DS で抽出したエラー

図中の「設計に直接影響しないエラー」は文言の表記揺れによって不整合が発生するなど、後工程において大きな問題とならないものである。「次工程で気づき修正が可能なエラー」は例えば製造工程において、開発者が発見することが想定されるエラーである。この 2 種類のエラーに対し、「大きな手戻りにつながるエラー」は、結合試験以降にならないと発見が難しく、また設計工程時のレビューでも発見するのが難しいと思われるエラーである。この種のエラーを早期に発見できる事で、試験工程での工期、工数の削減効果が得られると考えられる。この TERASOLUNA DS 導入プロジェクトは、発見されたエラーを完全に修正した上で後工程を実施している。よって、今後結合試験終了時に発生したバグについて分析を行うことで TERASOLUNA DS の効果を確認できると考えている。

6. 発生している課題と今後の取組と考察

先述したようにすでに 50 以上の開発プロジェクトで TERASOLUNA DS は導入されているが、いくつかの課題が顕在化している。その課題と今後の取り組みについて簡潔に記す。

6.1. 大量抽出されるエラーへの対応

TERASOLUNA DS を導入している開発プロジェクトにおいて、抽出されるエラー数は、全てのプロジェクトで数千～数万にも及んだ。目視で人間が確認できないほど大量のエラーが抽出されることは当初想定しておらず、何らかの対策が必要である。

- 自動修正：図 A-7-7 にもあるように、エラーの半数以上は「設計に直接影響しないエラー」である。内容としては、表記揺れによる文言の単純な間違いが多い。これらの単純な間違いを自動修正機能により対応することを検討している。
- エラーの原因抽出：上記表記揺れも含めて、抽出されたエラーは、設計書中の 1 つのエラーに起因して、関連する設計書で複数抽出されてしまう。すなわち、1 カ所を修正するだけで複数のエラーを修正することが可能である。そこで、抽出されたエラーの中心となる原因を問題記述表と同時に提示する機能を検討している。
- エラーの深刻度設定：図 A-7-7 にもあるように、エラーは後工程への影響が大きいものと小さいものがある。最終的には全てのエラーを修正する事が前提ではあるが、後工程への影響が大きい深刻度の高いエラーを優先的に修正して、工程を進める事を可能にするため、抽出されたルールに準じて、問題記述表にエラーの深刻度を設定する機能を検討している。

6.2. 設計書取込機能の高度化

先述したように、TERASOLUNA DS は、現時点では TERASOLUNA 開発プロセスで規定された Microsoft Excel で書かれた設計書フォーマットに対応している。新規システム開発プロジェクトにおいては、当該設計書フォーマットの利用を前提に TERASOLUNA DS を

導入するため問題は発生しない。しかし、すでに開発を終えて保守維持に入ったプロジェクトで、当該設計書フォーマットとは大きく異なるフォーマットを利用しているプロジェクトや、ほとんどの海外子会社のプロジェクトのように Microsoft Word で書かれた設計書を利用している場合は TERASOLUNA DS の導入が難しい。

この問題は、各プロジェクトが利用する設計書が将来的に規定された標準設計書フォーマットに収斂していけば解決する問題である。しかしながら、早期に普及展開を推進して効果を広く得るためには、各種フォーマットに対応する必要があると考えており、設計書取込機能の高度化を予定している。

6.3. ルール作成機能の高度化

先述したように、標準で約 337 ルールを提供しているが、プロジェクトによっては、独自にルールを追加して、約 1500 ルールで整合性チェックを行っているプロジェクトもある。ところが、現在のルール作成機能は設計書 DB として利用されている XML DB の知識も要求することから、プロジェクトで追加作成することが容易ではない。この問題を解決するために、XML DB の知識を必要とせず、GUI を通じて簡易にルールを追加作成できるツールの提供を予定している。

以上、述べたように TERASOLUNA DS の改良は継続的に行う必要があり、未だ完成された状態とは言えない。また、4 で説明したように、実際の開発プロジェクトでの効果検証は今後行う予定であり、その結果によってはさらなる改良が必要になることを想定している。

今回は設計工程での取り組みを紹介したが、TERASOLUNA DS の機能は設計以外の工程や工程間を跨ぐ整合性チェックにも利用可能である。特に後者はいわゆるトレーサビリティの確保であり、今後はソフトウェア開発全工程で、工程内、工程間の利用についても検討を進める予定である。

参考文献

[1] Barry W. Boehm :「Software Engineering Economics」、Prentice Hall、1981

[2] IPA/SEC : SEC BOOKS「共通フレーム 2013」、IPA/SEC、2013

A－8

Grails/Groovy の適用推進 [1]

1. 概要

本編では、企業システムのソフトウェア開発における社内標準フレームワーク適用事例として、NTT ソフトウェア株式会社（以下、同社とする）の例を紹介する。

同社では、主に企業システムを開発対象としたソフトウェア開発に関する種々の課題を解決するために、高機能なアプリケーションフレームワークである Grails[1]を全社標準フレームワークとして採用している。販売ソフトウェアの主力製品を含め、2012～2013 年度(2014 年 1 月現在まで)で計 42 件のソフトウェア開発プロジェクト(継続プロジェクトを含む)に適用している。

2. 取り組みの目的

本取り組みの目的は、インターネットサービスの開発で近年注目されている高生産性開発技術 Grails/Groovy[2]を企業向けのシステム開発に導入することで開発効率を高めることである。

開発期間短縮やコスト削減の要求が強くなるにつれ、従来から用いられてきた Java 言語のまま、あるいは文書化を過度に重視した従来の開発プロセスのままで開発生産性を向上することは徐々に困難になってきている。

片や、インターネットサービスの開発分野では開発効率の向上について長足の進歩が見られる。例えば、インターネットサービス開発で良く使用されるフレームワーク Ruby On Rails は、その分野では開発速度が死活的に重要であることを背景にして、迅速な開発を可能とする様々な機能と特長を備えている。また、Ruby on Rails はフルスタック・フレームワーク [2] としてアプリケーション全体の設計構造を提供するので、設計品質の底上げの利点も期待できる。しかしながら、Ruby On Rails を企業システム開発にそのまま適用することは困難な面もある。

Grails/Groovy は Netflix や LinkedIn, BSkyB, Vodafone, Walmart など多数の大手企業で

---

[1] 事例提供：エヌ・ティ・ティ・ソフトウェア株式会社　Grails 推進室　上原　潤二 氏

[2] フルスタック・フレームワークとは、画面遷移の制御から始まり、ビジネスロジックの定義、データベースアクセス処理など、ソフトウェアの実行の全レイヤについての包括的な枠組みを提供する単一のフレームワークのことである。

の採用実績のある [3]フレームワークおよび言語であり、Ruby On Rails と同様の利点や高い機能を持ちながら、従来の Java 開発手法と非常に互換性が高く、Java 開発を主体としてきた開発企業において低コストで多くのメリットを得ることができる。実質的には Grails/Groovy は Java 開発技術の延長であり、拡張であると言える。

この Grails/Groovy を企業システム開発に適用し、開発コスト削減と開発期間短縮を実現し、開発効率を高めることが本取り組みの目的である。

2.1. 背景

従来から Java を主力言語として開発を行なってきた同社のような企業において、Ruby on Rails に移行しようとすると以下のような課題が生じる。（表 A-8-1）Grails/Groovy ではこれらの問題を以下のように回避もしくは軽減した上で、Ruby on Rails と同様の利点を享受できる。

表 A-8-1　Java から Ruby on Rails へ移行する際の課題

観点	Ruby on Rails の採用で生じる問題	Grails/Groovy を採用した場合
習熟コスト	一般に Ruby と Java では文法・意味およびランタイムシステムの特性や構成が大きく異なっており、デバッグやトラブル対応能力を含めた習熟しなおしのためのコストが必要となる。	Java 技術の一部であると言えるほど、非常に親和性が高い体系であるため Java 技術者にとっては習熟コストが極めて低い。例えば Groovy の文法は Java のほぼ上位互換である。
性能や安定性	Ruby も向上してきているものの、性能や安定性、互換性ポリシーなどの面では Java VM プラットフォームと比べるとかなりの差がある。	Java VM 上で動作し、従来通りの安定性と性能を享受できる。
Java 資産	Java 既存資産、たとえばライブラリ、既存システムなどとの連携が難しく多くの場合にそれらを放棄しなければならない。また、広義の資産として、Java 高スキル者、Java 開発に習熟してきた開発協力会社との関係、ノウハウなどを 1 から構築する必要がある。	ツールやフレームワークを含め Java 資産との連動や共存が容易であり多くの場合そのまま使用し続けることができる。習熟・移行のためのコストが低く、人的資産を継続保持しやすい。

[3] Who's using Groovy in production? http://www.quora.com/Whos-using-Groovy-in-production

観点	Ruby on Rails の採用で生じる問題	Grails/Groovy を採用した場合
静的型チェック	Ruby ではコンパイル時の静的型チェックができないため大規模・長期開発では問題になる場合がある。	Groovy では任意に型指定を行うこともでき、Ruby に比べ型チェックを積極的に行うことで利点を得ることができる。

3. 取り組みの対象、適用技術、手法、評価・計測

3.1. 取り組みの対象

以下を開発するソフトウェア開発プロジェクトを適用対象とした。

- Java アプリケーション(Java VM 上で動作するシステム)
- Web アプリケーション(設定・管理機能として Web インターフェースを持っている場合を含む)

上記に該当するプロジェクト数は、例えば 2012 年度実績としては概算で年間 111 件ほどがあてはまっていた。Grails/Groovy の適用目的は以下の通り。

- 従来、Java で開発してきた種類のシステムを Grails/Groovy で開発し、開発効率の向上による利益を享受する。
- 従来、期間や費用制約により、PHP や Ruby を選択せざるを得なかったシステム開発に Grails/Groovy を適用することで、Java 技術の蓄積やノウハウを有効活用する。

3.2. 適用技術

OSS(オープンソースソフトウェア)[4]である Groovy および Grails を適用した。Grails と Groovy 概要を以下に示す。

（1）Groovy

Java プラットフォーム(Java VM)上で動作する言語であり、Java に似た文法と簡潔記述を特徴としている。ライブラリとして Java のクラスライブラリをそのまま、もしくはそれを拡張したものを使用する。

（2）Grails

Java EE で動作する Web アプリケーションフレームワーク。O/R マッピング [5]、DI

[4] Apache License, Version 2 に基づく OSS であり無料で利用することができる。コア開発者の多くが Pivotal 社(VMWare 社の Spring Source 部門と EMC 社からスピンアウトした企業)に雇用されている。

[5] O/R マッピング(オブジェクト関係マッピング、O/RM、ORM と表記する場合もある)は、オブジェクト指向における「オブジェクト」をリレーショナルデータベースのレコードや表スキーマに対応付けるための手法もしくはライブラリ。

コンテナ [6]、MVC フレームワーク [7]、ビルド機構 [8]、テスティングフレームワーク [9]なども含んだ包括的なフレームワーク(フルスタックフレームワーク)であり、Groovy 言語を用いてアプリケーションを記述する。実績のある Java フレームワークや技術群の基盤上に構築されている。Grails アプリケーションの構成を表 A-8-2 に示す [10]。

表 A-8-2　Grails アプリケーションの構成

レイヤ	主な Grails コンポーネント	基盤技術	
プレゼンテーション層	・ビュー(View) ・コントローラ(Controller) ・タグリブ(TagLib)		・Groovy 言語 ・Spring Framework ・Java EE ・Java Development Kit(JDK)
サービス層	サービス(Service)		
ドメイン層	ドメインクラス(Domain Class)		
データベース・O/R マッピング層	GORM (Grails O/R マッピング)	・Hibernate, JPA ・Jdbc(PortgreSQL, MySQL, Oracle, …) ・NoSQL(Mongo DB,…)	

3.3. 評価・計測

開発 PJ における Grails の採用数・採用率を評価軸の一つとして定めた。採用数・採用率の計測のため同社で使用しているプロジェクト実施報告の一項目として、 Grails/Groovy 採否と採用しなかった場合の理由記入欄を設け件数の計測と理由の分析、およびフィードバックを行なった。

また、Grails 採用による生産性向上効果はウォーターフォール型の開発プロジェクト全体で平均 23%の工数削減を期待している(後述)。

---

[6] DI(Dependency Injection)コンテナは、コンポーネントに関する初期設定や結び付け、生成や廃棄のライフサイクルなどの管理機能を提供するフレームワーク。Java 開発での利用は一般的。

[7] MVC(Model View, Controller)フレームワークは、Web アプリケーションの画面遷移を制御するためのフレームワーク。

[8] Grails では、Groovy ソースコードの更新を検出・再コンパイルし、アプリケーションを再起動せずに実行を継続する仕組みを含む。

[9] テスティングフレームワークは自動化されたユニットテストの作成と実行を支援するための仕組みであり JUnit が代表的。Grails では BDD(振舞駆動開発)を支援するフレームワーク Spock が使用できる。

[10] Grails のレイヤ構成は、マーチン・ファウラー氏によって提唱された「エンタープライズ アプリケーションアーキテクチャパターン」に影響を受けている(参考文献[3])。

4. 取り組みの実施、及び実施上の問題、対策・工夫

4.1. 経緯

Grails を標準フレームワークとして選定するまでの経緯は表 A-8-3 のとおり。2004 年から着目し、2012 年までに調査・検証・試行を行ない、方向性を定めて普及を行なった。

表 A-8-3　Grails を選定するまでの経緯

時期	項目	内容
2004～2007	技術調査	技術調査、書籍執筆、雑誌記事執筆、コミュニティ活動
2008～2011	試行評価・効果目標の設定	社内システム開発への試行適用を通じ、性能や機能に問題がないことの確認を行なった。また、適用効果目標(23%削減)を設定した。
2012～	適用準備・本格適用	Grails 推進室を設置、対象 PJ 条件の選定と全社標準フレームワークとしての普及促進を行なっている(継続中)

4.2. 本格適用前の準備

実際のプロジェクトへの本格適用に先立ち、以下の準備を 2008～2012 年度に行なった。

- 試行評価・効果目標の設定
- 推進室体制の構築
- 社内研修を整備
- リファレンスマニュアルの翻訳
- ドキュメントの整備公開
- 社内説明会の実施
- OSS ライセンスの整理
- 社外向けの普及活動

以下、それぞれについて説明する。

4.2.1. 試行評価・効果目標の設定

本格的な適用に向け、Grails を標準フレームワークとして活用した時の効果目標を適切に設定することで、社内展開の意識付けを高めることができる。そのため、以下表 A-8-4 に示す自社製品開発のプロジェクト、および社内利用システムの新規システム開発において Grails を適用し効果を確認した。

表 A-8-4　Grails の適用効果

項番	プロジェクト名	種別	Grails 開発メンバーのスキル	効果	Java とのコード量比較[11]	新規開発規模
1	顧客名簿管理システム	社内	高	詳細設計～結合テスト工程が、Java の約 1/2～1/5 の期間で開発	Java の約 1/10	50KL
2	NTT ソフトウェア(株)製品 ProgOffice ネットワーク電話帳	社外	中	詳細設計～製作(製造・単体テスト)工程が、Java の約半分の工数で開発(開発全体で工数 2 割減)	未測定	未測定
3	受託開発システム A	社外	高	詳細設計～結合テスト工程まで、Java の約 1/7～1/10 の期間で開発	Java の約 1/10	20KL

上記の結果およびヒアリング調査により、Grails 採用による生産性向上効果は、

- 製作(製造・単体テスト)工程コスト(工数)50%削減
- 詳細設計、結合テスト工程コストは 10%(工数)以上削減

と推定した。また、本推定に基づけば、工数比率[12]について以下表 A-8-5 のように重み付けすると、Grails 採用によりプロジェクト全体で 23%削減(1-0.7182/0.928)されると期待できる。

11 期間・開発工数の比較は、同社の見積りシステムによる予測に対しての比較。

12 工程名称および工数比率は「ソフトウェア開発データ白書の活用(IPA)」(http://www.ipa.go.jp/files/000004088.pdf)P5 より。

表 A-8-5　Grails 採用による工数削減効果

	基本設計	詳細設計	製作(製造・単体テスト)	結合テスト	結合テスト(ベンダ確認)	計
Java (工数比率中間値)	0.134	0.167	0.355	0.156	0.116	0.928
Grails(Java を 1 としたときの比率)	1	0.9	0.5	0.9	1	
Grails(工数比率)	0.134	0.1503	0.1775	0.1404	0.116	0.7182

上記は開発プロセスとしては従来通りのまま、言語とフレームワークを Java から Grails/Groovy に変更したことによって冗長な作業を除去したことによる効果である。Grails に即した開発の仕組みを構築することによる、さらなる開発効率向上の試みについては「6. 今後の取り組みと考察」にて後述する。

4.2.2. 推進室体制構築

Grails エキスパートを組織的に育成し、Grails の普及促進活動の中核となる組織「Grails 推進室」を 2012 年 1 月に発足した。2013 年 12 月現在 8 名が所属している。

まず、開発経験を推進室メンバーに積ませ、Grails で高品質かつ高セキュリティを必要とするソフトウェアを開発できることを実証するため、社内利用する高度な情報システム開発を実施した。本開発システムは全社員が利用するシステムであり、付随的成果として Grails の可能性と実績を社員に示すことができた。

また、技術獲得や Grails の機能・品質向上を目的として、OSS 開発、および Grails/Groovy へのバグ報告やパッチ送付なども行なっている。

4.2.3. 社内研修を整備

Grails 開発の人材を育成するために、表 A-8-6 に示すように研修メニューの整備を行なった。これらの研修メニューは、一部切り出して社外向けの研修としても活用している。

表 A-8-6　Grails 研修メニュー

研修名	内容
「Grails/Groovy 入門」	Java 開発、Web アプリ開発の経験者を対象とした、Grails 入門者向けの研修。
道場研修「Grails 応用」	Grails 入門受講修了者を対象とした Grails の研修。難解な部分の詳説、テスト技法など実践的なノウハウを学ぶ。

上記の研修に協力会社メンバー含め 2012 年度～2013 年度で、のべ約 200 人が参加し受講した。また、研修テキストはいずれも同社内で公開し、業務に関係する他社メンバーも常時利用可能としている。

これらの研修を受けるだけで使いこなせるようになるわけではないが、一般的な Java 経験者・Web アプリ開発経験者を前提すれば、上記の研修に加えて 1～2 週間の経験を積めば、レビューなどでの指導を前提として開発への主体的参加が十分可能となる。

4.2.4. リファレンスマニュアルの翻訳

開発時には、フレームワークに関するドキュメントが必須である。Grails では包括的な最新英文ドキュメントが常にリリースされるため、他のフレームワークと比べてもドキュメントが充実していると言えるが、日本語の情報は不足している。このため、英文の包括的リファレンスマニュアル(英文)を社内のボランティアを募り、手分けして翻訳・公開した [13]。

付随的な効果として、この活動を通じて、直接的に業務上の関係がないけれども、Grails/Groovy に対して技術的に興味を持っている有志との社内コネクションを作ることにも役立った。

4.2.5. ドキュメントの整備公開

開発時には、特定目的を達成したり、生じ易い問題を解決したりするための情報を集めたレシピ集や FAQ のような形でまとめられた情報も有用である。

このため開発者向けの実践的な技術的ノウハウを、研修テキストとは別に集積・整理し、FAQ(良く尋ねられた質問と回答)および「開発ガイド」として同社内に公開した。また、プロジェクトを統括するリーダー向けに、開発の進め方や、フレームワークの選定基準、プロジェクト管理上留意点、従来開発との差異についてまとめたドキュメントを作成し、Grails 採否の検討に役立てた。

4.2.6. 社内説明会の実施

新フレームワークを採用してもらう際には、プロジェクトリーダに対して社の方針を明確

[13] 翻訳結果は右記にて公開している。 http://grails.jp/doc/latest/guide/

に示したり、心理的な不安を払拭したりすることも重要になる。このため、社内標準フレームワークとして位置付けるに先立ち、Grails の技術的な説明会や、適用判断に関する各部署に対する説明会を実施しプロジェクト実施の際の疑問を払拭することに努めた。

社内説明会等で聞かれた質問や要望を以下に紹介する。

- Q:Grails の開発できる協力会社や要員の情報を集約して欲しい。
- A:相談に応じて、各部署間での調整を行なっている。
- Q:差分規模などを計算できる従来から使ってきたステップカウントツールを Grails/Groovy でも利用したい。
- A:該当ツールは OSS なので、Groovy に対応させるための独自のパッチを開発し、修正して社内に提供した。

4.2.7. OSS ライセンスの整理

一般に、OSS はライセンスの利用条件を明確にしておかないと利用方法によってはライセンス違反になる。適切な利用をするために、OSS ライセンスの利用条件調査は、Grails に限らず OSS 製品を業務の開発に使用する際には必須である。しかし、Grails は非常に多くの OSS を組み合せて構成されている。また、Grails では「Grails プラグイン」と呼ばれる、Grails 本体とは別に提供される多種のコンポーネントをアプリケーションに組み込んで使用することが良くあり、それらを通じて使用しているコンポーネントもアプリケーションの一部に組込まれる。これらすべてに対してプロジェクトごとにライセンス調査を実施すると、調査の工数が膨大なものになる。

このため、事前に Grails とそれが依存するコンポーネント群の OSS ライセンスを列挙し、代表的なライセンスについては対処法を事前に整理・提示しておくことで導入の敷居を下げるようにした。Grails プラグインについては個数が多く(2014 年 1 月現在で 868 件)、全てを調査することが困難であるため、主要なものについてのみ調査を行なった。

Grails を利用する際に生じるライセンスの問題(問題にならなかったものも含む)の例を以下に挙げる。

- Grails では O/R マッピングに Hibernate を使用する。Grails/Groovy 自体や、主要なコンポーネントのほとんどが Apache License Version 2 か MIT ライセンスであるが、Hibernate のライセンスは LGPL[14]である。LGPL の場合、リバースエンジニアリングに関する制約があるため、バイナリ提供の際には特に注意する必要がある。
- 「PDF プラグイン」、「Export プラグイン」は内部で iText を使用している。iText5 以降は AGPL(Affero GPL[15])を採用しており特に注意が必要なライセンスであるが、

[14] LGPL(GNU Lesser Public License, GNU 劣等一般公衆ライセンス)については下記を参照：
http://www.gnu.org/licenses/lgpl.html

[15] Affero GPL(GNU アフェロ一般公衆ライセンス)については、下記を参照:
http://www.gnu.org/licenses/agpl-3.0.html

調査したところこれらのプラグインが使用している iText のバージョンは 5 以前のものであり、AGPL は適用されないことが確認できた。ただし今後のバージョンアップなどでも同じものが同梱され続ける保証はないので、留意しておく必要はあるかもしれない。

4.2.8. 社外向けの普及活動

仮に社内開発のみに使用するにせよ、Grails/Groovy 自身の知名度や実績が低いと、顧客提案時に説明が難しいというような支障が生じ得る。そのために、JGGUG(日本 Grails/Groovy ユーザグループ)を中心として、コミュニティ活動および OSS 開発による社外への貢献活動、学生向けセミナー、社外向けホームページの技術情報の情報発信活動を行なっている。

さらに、Grails/Groovy に関する技術国内 Java コミュニティなど他のコミュニティ、カンファレンス、勉強会等で発表するなどを積極的に行なっている。また、Grails/Groovy に関する書籍執筆 [16]、雑誌記事執筆を行なった。

4.3. 本格適用

普及推進活動の結果として、2011 年度から 2013 年度第三四半期(2013 年 12 月)までの年度ごとの適用プロジェクト累積数の推移を図 A-8-1 に示す。

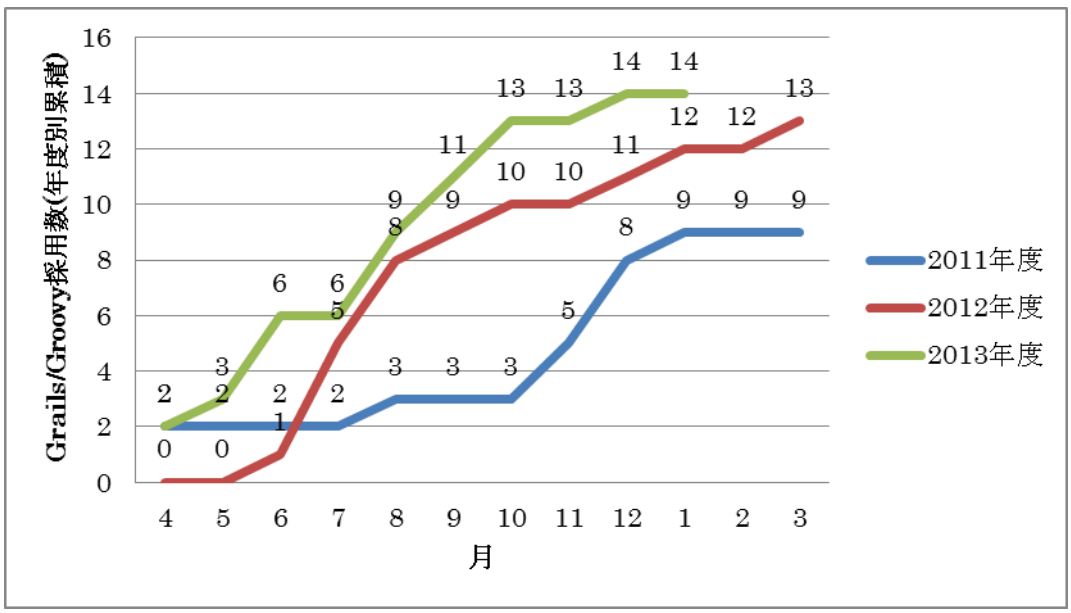

図 A-8-1　Grails/Groovy 採用数の推移

16 書籍に関しては、「Grails 徹底入門(翔泳社)」、「プログラミング Groovy(技術評論社)」を同社社員が執筆(共著)。

4.3.1. 不採用理由の分析

Grails を不採用としたプロジェクトの不採用理由は以下の通りである。

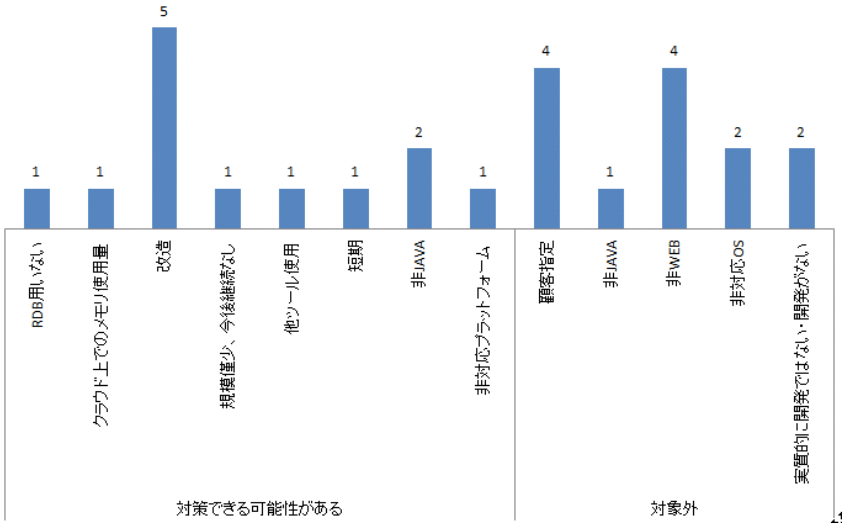

図 A-8-2　Grails を採用できなかった理由

不採用理由の収集を開始したのが 2013 年 10 月以降なので、件数がまだ少なく定量的な分析は難しく、本格的な分析とフィーバックはまだできていない。

図 A-8-2 で、理由が「改造」であるものは、開発内容が Grails 以外のフレームワーク(例えば Struts)を使用した既存システムに対する改造開発の場合である。なお、Grails 開発では一般に Java システムをとり込むことが容易にできる [17]ので、単に改造であることだけを理由に採用しないのは合理的ではない。例えば、既存 Java システムを Grails アプリケーションの一部に取り込んだ上で追加部分や新規画面のみを Grails の利点を活かして開発することは理論的には可能である。そのため改造の場合でも採否理由の妥当性は個々に検討する必要がある。

4.3.2. 実施上の問題と工夫

以下のような工夫を行なった。

(1)　多角的な相談チャネルを設けること

社内プロジェクト担当者から Grails 推進室への問い合わせ方法は、対面でのミーティングはもとより、メール、社内コミュニティサイト、IRC(インターネット・リレー・チャット)、イシュートラッキングなど多数のチャネルを準備し対応を行なった。こうすることで、正式ルートでは情報交換がし難い若手技術者とコネクションを作ることができた。

[17] 例えば Hibernate のエンティティはそのまま Grails のドメインクラスとして扱われるし、Spring Bean も直接利用できる。Grails のアプリの一部として Struts の Servlet を同居させて併存することもできる 。

社内コミュニティサイトは、Grails 推進室への問い合せを行なった開発プロジェクトのメンバーを中心に登録した(2014 年 1 月現在 77 名が参加)。これによって、Grails 開発に興味を持った個人が参加する、継続的なコミュニティを形成することができた。

(2) 高度な技術支援とノウハウ展開

技術支援の大半は、比較的基本的なものであった。しかし、Grails や Hibernate のソースコード解析が必要になるなどの難易度の高い問題は一定の比率で発生する。必要な問題はレベル回避策をプロジェクト側に提示すると共に、開発元に対するパッチ提供を行なった。なお、品質の問題は本体よりも Grails プラグインで発生することが多い。バージョンが追随していないなどの問題や、機能不足などの問題もあった。プラグインについては有用かつ利用実績を積んだものを「推奨プラグイン」として整理するようにした。支援においては、サンプルコードの提供やプロジェクト作業の一部を切り出して実施することも行った。また支援結果は、なるべく一般化して他プロジェクトも参照できるノウハウとして前述のコミュニティ経由で積極的に公開している。

(3) Grails/Groovy を採用した開発リーダーへのヒアリングとフィードバック

Grails を初期に採用した開発リーダーからは終了時に適宜ヒアリングを行い、課題を抽出した。初期の導入者は貴重な協力者であるため、そのアドバイスは傾聴すべきものが多々ある。例えば、OSS ライセンス整理に関する問題や、プロジェクトリーダ向けのガイドの作成はヒアリングを通じて得た意見に対するフィードバックとして行なった。

(4) 不採用理由の収集と分析

開発プロジェクト発足時に「不採用理由」の記入と報告を義務付けた。この目的は、収集した理由の分析を行い、必要なフィードバックを行うことである。結果として、Grails の採否検討や、採用にもし支障があるのであればその課題の克服のための、プロジェクト側の主体的な議論のきっかけともなっている。

5. 取り組みの実際と結果

適用プロジェクト 42 件の内容は、高信頼システム(365 日 24H 稼働を要件としたシステム)や人事情報を扱う高セキュリティシステムを含み、またウォーターフォール的なものとアジャイル開発を適用した場合なども含み多岐に渡る。これらの実適用例について、ヒアリング結果などから得られた実適用の様子を紹介する。

5.1. 採用の判断

プロジェクトリーダによる採否判断において、(1) 性能や (2) 品質 (3) セキュリティ面での質問を受けることは多かった。以下それぞれについて要点を示す。

(1)　性能面では例えばRuby on RailsやPHPでのシステムと比較すれば問題がないし、Grailsが提供するフロントエンド最適化 機能により、他のJavaやScalaベースのWebフレームワークより高速なレスポンスも期待できる。ただし、CPU時間で課金されるPaaS環境での実行が必要だったり、同時処理可能なユーザ数が多ければ多いほど良い大規模クラウドサービスの実装を実装したりする場合にはサーバサイドの性能が問題になり得る。その場合は「十分な性能があれば良い」のではなく「速ければ速いほど良い」からである。中長期的なランニングコストがビジネスの損益分岐を決定づけるようなクラウド上のシステム開発では、Grails/Groovy よりも Scala や Java を使用すべきケースもある。

(2)　品質についてはGrails/Groovy本体では問題はほとんど起きていないが、プラグインで問題が起きたケースがあった。データベース処理を透過的にマルチテナント管理するマルチテナントプラグインを採用した際、初期試用では発覚しなかったコーナーケース(めったに発生しない厄介なケース)で深刻なエラーが発生し、最終的には該当プラグインを使用しないようにアプリケーションを書き替える必要が生じた。対策として、事例を同社内に紹介するとともに、プラグインの品質ばらつきは本体よりも大きいため、信頼性や機能性について事前に調査評価を行い、推奨プラグインおよび要注意プラグインの情報を適宜提供している。

(3)　セキュリティについては、Struts のセキュリティ問題の多発、保守の停止の問題があったため、注目されやすく質問を受けることも多い。安全な Web アプリケーションを開発するためにGrailsでも注意が必要なことにはかわりがないが、

- 高機能・柔軟で、広く使われているSpring Securityにログイン・ユーザ管理機構をまかせ自分で認証処理を作る必要がないこと
- ORマッピングを前提することでSQLを隠蔽できること
- コード量が顕著に減少すること

などから、Grails 上ではセキュリティ向上を見通し良く実施できるとは言える 。このような特徴を踏まえ、StrutsからGrailsへの移行を促すため、Strutsで記述されたアプリケーションをGrailsに簡易に移行するためのノウハウを整理収集中である。

5.2. 参画したGrail利用者の経験レベル、育成方法等

新規プロジェクトのほとんどで Grails 経験者プロジェクトはいないか少ない状態で実施された。全体的に、開発担当者にJavaおよびWebアプリケーションの開発の経験があれば問題なくプロジェクトが立ち上っている。育成方法は前述の研修とOJTで、期間としては概ね2週間で立ち上り、2ヶ月で完全に使いこなすことができるようになっているようである。

OJTについて技術支援を行なっているが、実際の支援を通じて得たポイントは以下のとおりである。

- 質問に答える形での技術支援だけではなく、設計書レビューやコードレビューに識者

が入りコメントを出す形を含めた方が効果的な支援になりやすい。何をわかってないかがわかっていない場合や、勘違いや、判断できず従来の方法を元にした方法でやってしまう場合を修正できないためである。

- コード作成やトラブル対応については、時間を掛ければプロジェクト側で解決・回避することは多くの場合可能であり、それを通じて技術力が付く面もある。そのため、重要なのは、採用した次の段階の技術支援において、Grails の利点を引き出すことであり、どこをどうカスタマイズできるのか、既存の仕組みを利用することで、コーディングを不要にできることは何か、などの開発戦略を支援することであると考えている。事前に機能プロトタイプや雛形を準備し、それを参考にしてもらうことも有用である。

5.3. Java 開発との記述量比較

同機能のアプリケーションを Grails と Java で記述した場合のコード記述量は、ステップ数単位では比較できていないが、クラス数については、Java コードを Grails で置き換える事例において、Java では「User」というデータを表現するのに 10 個以上のクラス(例:UserListViewModel, UserListHelperBean, UserServiceController, UserServiceResult, UserListEvent, UserListEventListener, UserListEventResult, UserDTO, UserDAO, AbstractDAO..など)の組み合わせとして定義していたものを、Grails では単一のドメインクラス・ソースファイルで表現できた。一般に Grails ではデータ構造を表現するためのステップ数およびクラス数を顕著に減少させることによって、以下の問題が解決できる。

- データ構造の修正や改造が、画面の自動生成機能(Scaffold)の利用とも併せて、相対的に非常に容易になる。
- 多数の Java クラス群で構成される設計構造について、設計上の意図と必然性が保守を含むソフトウェアの長期的なライフサイクルの中で見失われ、抜本的な拡張や改善が実質不可能な大量コード群(レガシーコード)と化してしまうことがある。Grails を適用することでこの問題は抜本的に解決もしくは改善できる。

5.4. プロジェクトメンバーの所感

以下に開発に参加したプロジェクトメンバーからヒアリングした所感の主なものを示す。

- Spock という Groovy ベースのテストフレームワークを採用したが、単体試験の開発の時間短縮にとても役にたった。
- Java との親和性は非常に優れていることが確認でき、目論見通り問題は発生していない。
- Grails はリファクタリングに強い。結合試験中も検討もれがあったときに、修正にそれほど時間がかからなかった。今後の改造があるならメリットが活きてくる。

5.5. 今後の課題

以下にもヒアリングをベースにして収集した、開発側での懸念としての技術的な課題を示す。

- Grails 固有の問題ではなく、OSS 一般の問題ではあるが、多数の OSS コンポーネントを組合せて構築されたフレームワークにおいて、下位に使用しているコンポーネントでの脆弱性情報の共有の仕組み、及び見付かった場合の安全なアップデートや確認についての手法が必要。

6. 今後の取り組みと考察

まず、Grails/Groovy の導入の敷居を下げることが課題である。なぜならば、Grails に習熟すれば開発効率が上がるとしても、習熟していなければ開発効率は一時的には落ちてしまうためである。Java 技術者であれば導入時ギャップは相対的には低くはあるものの、このギャップを埋めることが非常に重要である。このために、アプリの基礎構造や画面遷移処理を対話的に生成できる Wizard 形式のツール開発を進めている。また、Grails 環境の標準開発環境を即座に利用できる、クラウド開発環境の整備も進めている。

さらに、Grails/Groovy を使用して得られる利点をより拡大していく。適用範囲が広く、拡張性の高い Web アプリケーションフレームワークを広く採用し、それに即した開発プロセスを組織的に確立することで、相乗的な効果を得ることができるはずである。このために以下のような取り組みを進めている。

- Grails を前提にして、タスクや検討・設計の成果物を明確に定義することで、タスクアサインの柔軟性を高め、初心者や低スキル者でも実施可能なタスクを切り出し、OJT による育成を効率化する。あるいはアウトソーシングを効率的にし、かつ制御しやすい形で実施できるようにする。
- 設計・検討情報のドキュメントを削減する。具体的には、可読性が高く、抽象度の高い Grails のコード群で設計・検討情報を表現しドキュメントを代替する。また、可能なものに関しては Grails コードからリバース生成を行なう。

以上より、全体としてより合理的で効率の高い開発スタイルを組織として確立し、開発者の努力をより高次のレベルの生産的な側面(例えばより的確なシテムのモデル化や、品質向上、継続的なシステム改善など)に注力していくことを目指す。

7. まとめ

内外の先駆的ソフトウェア企業が、インターネットサービスの開発において駆使しているような開発技術やツールの発展と比べるなら、現在の日本におけるソフトウェア受託開発業種の内情は合理的で効率の良い開発スタイルを導入しているとは必ずしも言い切れない状況である。諸般の事情はあるとは言え、座視すればそのギャップは拡大する一方となろう。

Grails/Groovy の取り組みは、2004 年から実施してきた技術調査を通じて、Java 開発のベースとして「習熟コスト」「性能・安定性」などの評価基準をクリアし、試行評価を実施し現在の同社に導入する時期に来ている。しかし、開発技術の進歩は日進月歩であり、Grails/Groovy の普及推進が長期的に不変・安定した施策であるわけではもちろんない。同社担当者によると、本事例は、同社が直面している課題を認識して、その上で今、辿ることが可能であると仮説できる開発技術力向上の道、現実と地続きの道として、Java 開発技術を Grails/Groovy を用いて拡大発展させる試みの紹介である。

参考文献

[1]山田 正樹, 山本 剛, 上原 潤二,他:Grails 徹底入門,,翔泳社,2008 年
[2]関谷 和愛,上原 潤二,須江 信洋,中野 靖治:プログラミング Groovy,,技術評論社,2011 年
[3]マーチン・ファウラー(著),長瀬 嘉秀 (監訳):エンタープライズ アプリケーションアーキテクチャパターン,,翔泳社,2005 年

A－9

個人依存開発から組織的開発への移行事例[1]

～要求モデル定義と開発プロセスの形式化による高生産性/高信頼性化～

1. 概要

本編では、個人依存開発の組織に対して技法を適用して組織的開発への移行について、三菱電機メカトロニクスソフトウエア株式会社の例を紹介する。

製品の成熟化に伴う付加価値増加やグローバル対応による製品バリエーション増加に伴い製品に搭載する機能量は増加する一方である。個人依存の開発では機能量の増加に対応しきれず生産性と品質を維持することが困難となる。個人依存の開発から組織的な開発への移行が急務となっている。

ソフトウェア開発で発生する課題は、人の作業によって作り込まれる。課題を作り込まなくするために組込みソフトウェア開発の現場で発生する課題を定義して、課題を抑制する分析・設計・実装・テストのフレーム上にソフトウェアを構築する手順を開発プロセスに定義して組織的な開発を進めることが大切である。ここでは、要求を正確に形式的に定義した分析モデルを設計・実装・テストの双方向に紐付することにより徹底した無駄取りと不具合の混入を極力抑え高生産性と高品質化を可能にした事例について説明する。

2. 解決する課題と目標

2.1. 課題

製品の機能量の増加に伴い以下に示す 7 つの課題が発生する。この 7 つの課題を抑制することで高信頼性/高生産性のソフトウェア開発を可能にする。

（1）　ニーズ対応スピード低下

組織が大規模化すると市場の要求がサービス部門、営業部門、設計部門を経由してソフトウェア開発部門に展開される。この過程で多くの意思決定の会議が開催されソフトウェア開発部門がソフトウェアの開発に着手するまで多くの時間を要する傾向がある。多くの時間を掛けて市場の要求に対応した製品をリリースしても既に市場要求が変動している可能性が大きい。また、機動力の高い企業や他社が先行して市場に投入することでビジネスが停滞する可能性もある。更に設計部門では設計の観点で製品を開発する傾向が強く市場

[1] 事例提供：三菱電機メカトロニクスソフトウエア株式会社　和歌山支所　岩橋　正実 氏

要求を見失いがちである。その結果、ソフトウェア開発部門に提示された要求には、市場の要求事項が読み取れない状況が進んでしまう。組織全体が上流を見ずに下流を見る傾向が進むことで市場ニーズの獲得力及び対応スピードの低下によりビジネスが停滞する課題が発生する。

（2） 個人依存の開発の限界

製品の成熟化に伴う付加価値増加やグローバル対応による製品バリエーション増加に伴い製品に搭載する機能量は増加する一方である。その結果、2 つの課題が懸念される。

限られた要員で規模の増加に対応する為に分析や設計の精度を落として実装を先行してしまう傾向がある。テストでこれをカバーしようとするが、分析や設計の精度が低いためにテストの実行精度も低下することになり品質を確保することが困難になる傾向がある。

もう一つの課題は、作業負荷増加とコスト制約に対応するためにオフショアを含めた外注企業に委託するケースが多くなる傾向にある。個人に依存したソフトウェア開発では、このような変化に対応しきれないばかりかモノ作り技術の外部依存が進むことになり品質を確保することが困難になる傾向がある。更にオフショアが進めば繊細な物づくりを得意とする日本のソフトウェア開発技術が海外へ流出して日本の弱体化も懸念される。

（3） 機能仕様の品質

製品仕様構築部門は、要求仕様を日本語で記載するのが一般的である。しかし、日本語記述には曖昧性の課題がある。仕様の記載粒度が荒いと多くの曖昧表現を含む場合がありソフトウェア開発部門が仕様を正確に読み取ることが困難となる。その結果、意図しない仕様を作り込みシステム障害を発生させる可能性がある。仕様が曖昧な状態では、分析・設計・テストの記載文書も曖昧になり品質を確保することが困難となる。この対応として要求モデルを形式手法言語で厳密に定義して、モデル検査を実施して高品質化を実現する形式手法がある。しかし、数学論理を用いてソフトウェア実装に近い言語で記述することになり要求提示部門とソフトウェア開発部門のギャップが大きくなり要求提示部門が厳密に定義した要求を正確に理解できない課題が発生する。

（4） 機能の組合せの増加

製品の単体機能の品質を確保することは比較的容易であるが、市場で様々な機能項目が様々なタイミングで並行動作する。その結果、想定していない機能項目の組合せが発生してシステム障害を発生するケースがある。

機能項目の増加により複数の機能項目が並行動作する組み合わせが増加する。多くの機能項目の並行動作により全ての機能項目の相互の関係の中からシステム障害に繋がる組み合わせを検出することが困難になる。その結果、要求分析で機能の組合せの仕様を正確に定義できないためテストで機能の組合せの問題を検出することが困難な状況になる。

（5） 例外の検討漏れ

製品の基本機能の品質を確保することは比較的容易であるが、市場で様々な例外事象が発生して全ての例外発生の状況を検討しきれずシステム障害を発生するケースがある。

例外は、組込みシステムの実行基盤と組込みシステムを構成する物理的なデバイスや通信等で接続されるサブシステムにより生じる。組込みシステムを構成する物理項目の増加により例外の数が増加する。機能項目の増加により機能項目の実行中に発生する例外の組合せが増加して全ての例外の発生時の動作仕様を正確に定義することが困難になる。その結果、要求分析にて例外の発生時の仕様を正確に定義できないためテストで例外発生時の問題を検出することも困難な状況になる。

（6）　時間制約の検討漏れ

分析時に時間制約を考慮せずにテストにおいて製品の動作から不具合を検出して時間制約の課題を開発の終盤に検出するケースがある。最悪の場合、特定のタイミングでしか発生しない時間制約の場合は、出荷後にシステム障害が発生する可能性もある。

時間制約に基づいたタスク設計ができていない場合も課題が発生する。時間制約に基づきタスク設計の指針が無いため多くのタスクを定義することによる資源の浪費やタスク間インタフェースの増加により複雑化する課題が発生する。

（7）　機能仕様の重複

同じ機能を実現する仕様定義が個人に依存して似て非なる仕様定義が増加する課題がある。更に製品のバリエーションの増加に伴い製品間で重複する仕様記述が増加する。機能量と製品シリーズの増加に伴い仕様の共通部分が整理できず仕様改定の漏れが発生する可能性がある。更に 1 つの機能仕様に複数の製品の動作の場合分けの記述を重ねることで製品シリーズの増加に伴い仕様の複雑化が進み仕様品質の悪化が進むことになる。

2.2. 目標

要求を厳密に定義することにより不具合を分析・設計・実装の工程で 90%以上検出を可能にすることとソフトウェア納品後のシステムテストに於いて検出する不具合を 0 件とすることを目標とする。

3. 適用対象と適用技法

3.1. 対象

対象とする領域は、機器を制御する組込みシステムで複数の機能が並行して動作するリアルタイム制御システムである。小規模な組込みシステム単体から、組込みシステム間がネットワークで接続され組込みシステム間で通信を実施しながら制御する組込みシステムを対象とする。

技法の適用工程は、全開発工程に適用した。分析を厳密に定義して、分析から設計、実装、テストへの双方向の変換パターンを用いることで分析の品質を他の工程に持ち込むプロセスを適用した。

3.2. 方法論

3.2.1. 概論紹介

本事例で紹介する技法は、自律オブジェクト指向（AOO : Autonomic architecture base Object-Oriented development technique）と呼ばれている。

AOO は、オブジェクト指向技術を基づき、要求を目的で機能項目に分解整理し、機能項目間の依存関係を無くし、自律した機能として表形式と形式化した日本語により厳密に要求モデルを定義した上で、分析・設計・実装・テストの双方向のトレーサビリティを確立する技法である。

AOO は、開発上で発生する課題をアーキテクチャ（文書/ソースコードのフレーム）とアーキテクチャに落とし込む開発プロセスにより組織的な課題抑制を可能にする。以下に AOO 技法のポイントを示す。

（1） 要求の発生源と目的定義
（2） 要求のカテゴリによる目的分解整理
（3） 表形式と日本語による要求モデル
（4） 日本語形式記法
（5） 製品内と製品間の共通部定義
（6） 類型化の推進
（7） 自律化による要素間の依存関係の排除
（8） 競合/例外/時間制約の解決
（9） 重複情報及び重複作業の徹底した無駄取り
（10） プロセス形式化と双方向のトレーサビリティによる定型化と自動化
（11） アジャイル/派生機種/ソフトウェアプロダクトライン開発の対応
（12） 要求モデルと標準プロセスに基づく見積り技法
（13） 要求モデルに基づく開発管理技法

3.2.2. 課題対応策

2.1 の課題の項番号に対応して課題対応策について述べる。

（1） 目的指向開発

市場の要求がサービス部門、営業部門、設計部門を経由してソフトウェア開発部門に展開される。多くの部門を経由する際、部門間は下位の組織へ向く傾向がある。特に設計部門では市場要求が読み取れない状況になる傾向がある。更にソフトウェア開発部門では、仕様の目的が読み取れない。このような状況を解決する為にソフトウェア開発部門が、要求を目的で機能項目に分解する際に、機能項目の目的を定義する。目的定義は、要求の発生源と要求の目的をヒアリングして目的を定義する。これにより設計部門は、営業部門やサービス部門へ問い合せることになり組織全体のフロントローディングが進むことになる。

目的が定義できることにより要求の価値が明確になり対応の優先レベルが判断できる。

更に目的を達成すための最適な仕様であるかを判断できるためソフトウェア開発部門から仕様改善の提案が進むことになる。目的指向開発は、機能項目リストの目的定義様式と開発プロセスの定義により組織的に展開が可能である。（図 A-9-1）

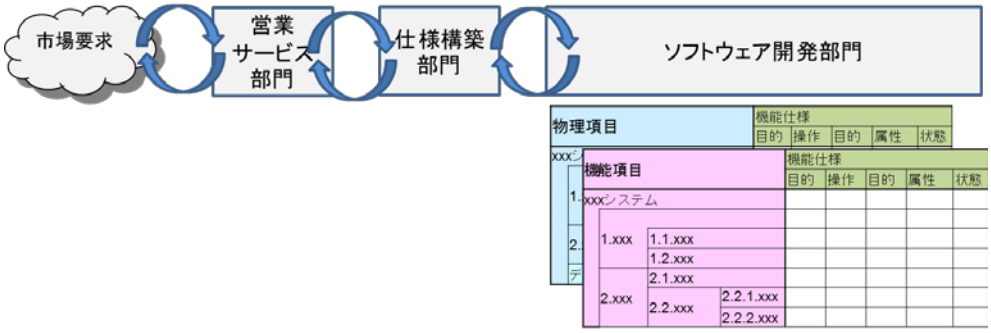

図 A-9-1　機能項目の目的定義

（2）　開発プロセス導入による個人依存開発からの脱却（図 A-9-2）

AOO による開発プロセス(AOO_PRS)の技法について述べる。AOO_PRS は、ソフトウェア開発で発生する課題を要求分析時に解決して、要求を設計/実装/テストに双方向にマッピングすることにより誤りの混入と重複した情報及び作業の除去により高品質と高生産性の両立を実現させた開発プロセスである。AOO_PRS は、ソフトウェア開発で発生する課題を抑制するための分析・設計・実装・テストのフレームを定義している。フレームとは文書様式のことで実装においては、フレームワークのことである。課題解決のフレームを前にすると特定の課題解決を実施する意識が働く、課題を解決するための作業をプロセス定義することで、組織的な課題解決を可能にする。

本開発プロセスは、アクティビティとタスクで構成される。タスクはサブタスクで階層表現され最下位のタスクは実行作業レベルの粒度で定義している。開発粒度（新規開発、機能追加、機能変更、先行開発）に基づき実行タスクを定義している。各組織の特性に応じて最適なタスクの選択及び追加するプロセスフレームを提供する。

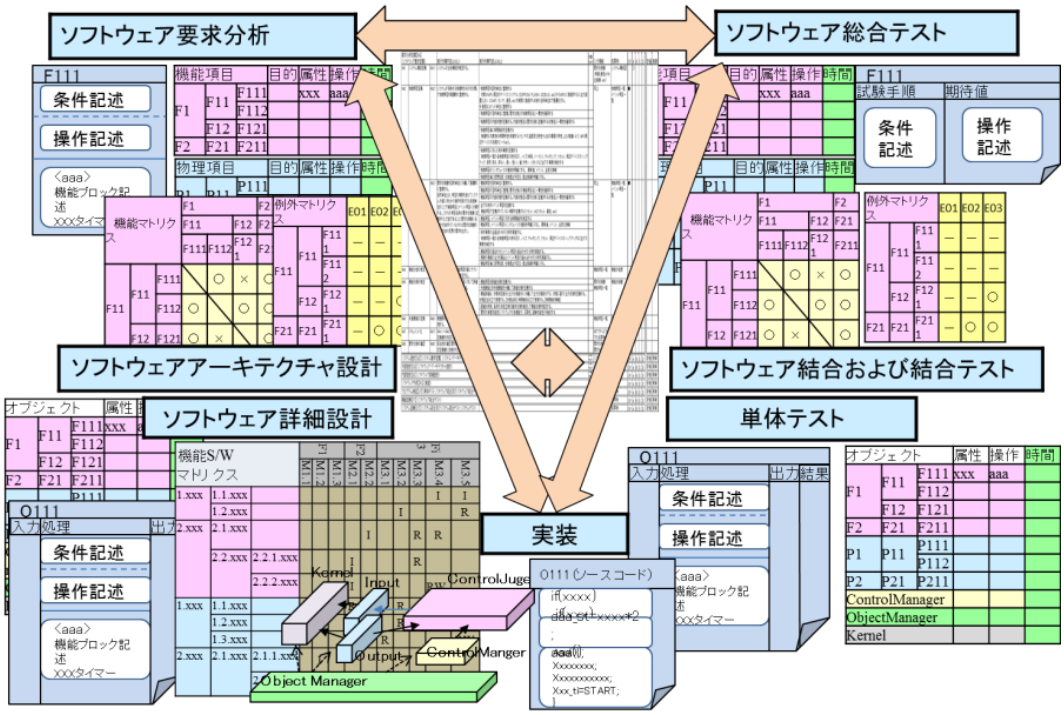

図 A-9-2　AOO による開発プロセス

AOO_PRS では、不具合の作り込みの真因を分析して、どのような作業を実施すれば、作り込みを抑制できるかを分析する。作業を安定して実施する為の文書フレーム（様式）と実装フレーム（フレームワーク）を定義して、フレームに要求を実装する作業手順を開発プロセスに定義することで組織的な開発へ移行することで高品質化を実現する。

対象組織の経営方針又は、事業戦略に対して現状のソフトウェア開発とのギャップを分析して、課題を定義する。定義した課題を未然に抑制する為のアーキテクチャとアーキテクチャを正確に使用してソフトウェアを開発するプロセスを構築する必要がある。経営方針や事業戦略上には、大きく分けて信頼性と生産性の課題解決を進めることになる。

最初に信頼性に関する課題の解決の方法について述べる。開発プロセス導入時に必要なことは、AOO_PRS を導入すれば成功するのではなく、対象部門の流出不具合を分析して AOO_PRS の導入効果の有無を判断する必要がある。対象の組織の不具合の発生源（不具合を作り込んだアクティビティ）と不具合の作り込み要因（真因）を定義する。同種の作り込み要因で分類して、上位階層のカテゴリに解決する課題を定義する。流出不具合は、経営上のリスクに繋がり、それを経営方針及び事業方針を基にその必要性を定義したうえでアーキテクチャとプロセスを定義することになる。

次に生産性に関する課題解決の方法について述べる。対象組織の機能追加変更の特性を分析する。同種の変更特性で分類して上位階層のカテゴリに解決する課題を定義する。生産性の課題は、規模の増加及びリードタイム短縮等の経営方針と満足させるためのアーキ

テクチャとプロセスを定義することになる。

以下に示すプロセス定義は、ESPR(組込みソフトウェア向け開発プロセスガイド)の文言を使用して、AOO_PRS と同じプロセスフレームの定義例である。(図 A-9-3) アクティビティタスクの文言は、課題の作り込みを未然に防止できるような実作業レベルのタスクが定義している。プロセスフレームは、開発粒度に応じた実行すべきタスク実行パターンが定義している。定義しているタスクは、プロジェクト毎にプロセス設計を実施する。実行タスクを計画して実行結果を管理する。

標準プロセス						プロセスパターン				計画管理	
アクティビティ		タスク		サブタスク		パターンA	パターンB	パターンC	パターンD	計画	実績
項	名称	項	名称	項番	名称						
1.	ソフトウェア要求分析	1.1	ソフトウェア要求仕様書の作成	1.1.1	制約条件の確認	○					
				1.1.2	ソフトウェア機能要求事項の明確化	○					
				1.1.3	ソフトウェア非機能要求事項の明確化	○					
				1.1.4	要求の優先順位付け	○					
				1.1.5	ソフトウェア要求仕様書の作成	○					
		1.2	ソフトウェア要求仕様の確認	1.2.1	ソフトウェア要求仕様書の内部確認	○					
2.	ソフトウェア・アーキテクチャ設計	2.1	ソフトウェア・アーキテクチャ設計書の作成	2.1.1	設計条件の確認	○					
				2.1.2	ソフトウェア構成の設計	○					
				2.1.3	ソフトウェア全体の振る舞いの設計	○					
				2.1.4	インターフェースの設計	○					
				2.1.5	性能/メモリ使用量の見積もり	○					
				2.1.6	ソフトウェア・アーキテクチャ設計書の作成	○					
		2.2	ソフトウェア・アーキテクチャ設計の確認	2.2.1	ソフトウェア・アーキテクチャ設計書の内部確認	○					
		2.3	ソフトウェア・アーキテクチャ設計の共同レビュー	2.3.1	ソフトウェア・アーキテクチャ設計書の共同レ	○					
3.	ソフトウェア詳細設計	3.1	機能ユニット詳細設計書の作成	3.1.1	プログラムユニット分割	○	○				
				3.1.2	プログラムユニット設計	○	○	○	○	○	
				3.1.3	インタフェースの詳細化	○	○				
				3.1.4	メモリ量の見積もり	○					
				3.1.5	ソフトウェア詳細設計書の作成	○					
		3.2	ソフトウェア詳細設計の確認	3.2.1	ソフトウェア詳細設計書の内部確認	○	○				
		3.3	ハードウェア仕様との整合性の確認	3.3.1	ハードウェア仕様との整合性の確認	○					
4.	実装および単体テスト	4.1	実装及び単体テストの準備	4.1.1	実装の準備					○	
				4.1.2	単体テストの準備	○	○				
		4.2	実装及び単体テストの実施	4.2.1	プログラムユニットの実装					○	
				4.2.2	単体テストの実施	○	○	○		○	
				4.2.3	単体テスト結果の確認	○	○				
		4.3	実装および単体テスト結果の確認	4.3.1	ソースコードの確認	○	○	○	○	○	
				4.3.2	単体テスト結果の内部確認	○	○	○		○	
5.	ソフトウェア結合テスト	5.1	ソフトウェア結合テストの準備	5.1.1	ソフトウェア結合の準備	○	○				
				5.1.2	ソフトウェア結合テストの準備	○	○				
		5.2	ソフトウェア結合テストの実施	5.2.1	ソフトウェア結合	○	○				
				5.2.2	ソフトウェア結合テストの実施	○	○				
				5.2.3	ソフトウェア結合テスト結果の確認	○	○				
		5.3	ソフトウェア結合テスト結果の確認	5.3.1	ソフトウェア結合テスト結果の内部確認	○	○				
6.	ソフトウェア総合テスト	6.1	ソフトウェア総合テストの準備	6.1.1	ソフトウェア総合テスト仕様書の作成					○	
				6.1.2	ソフトウェア総合テストの準備						
				6.1.3	ソフトウェア総合テスト仕様書の内部確認	○	○	○		○	
		6.2	ソフトウェア総合テストの実施	6.2.1	ソフトウェア総合テストの実施					○	
				6.2.2	ソフトウェア総合テスト結果の確認					○	
		6.3	ソフトウェア総合テスト結果の確認	6.3.1	ソフトウェア総合テスト結果の内部確認	○	○				
		6.4	ソフトウェア開発の完了確認	6.4.1	ソフトウェア開発の完了確認	○					

図 A-9-3　プロセスフレーム定義例

開発プロセスの適用方法について述べる。個人依存の開発となっている組織に開発プロセスを説明すると共感できるタスク、共感できないタスク、そして不足しているタスクが抽出できる。共感できないタスクは、タスク実行によりどのような課題を未然に防止できるかを明確にした上で、対象組織で同種の課題が発生していないかを分析してタスクの実行有無を確立する。次に不足しているタスクに対して、タスク実行の目的を明確にする。タスク実行で未然に防止する定義する。定義した課題が対象組織の課題であるかを判断した上で、課題を未然に防止するタスクを定義する。定義した課題＋タスクが対象組織に必要と判断した場合は、開発プロセスにタスクを追加する。不足しているタスクを定義することは、対象組織で個人の暗黙知となっている仕事のやり方を開発プロセスとして形式知化でき、組織全体の作業水準を上がることが可能になる。

（3）　機能仕様の品質

AOO による要求の形式化(AOO_DSL)の技法について述べる。(図 A-9-4) 仕様構築部門から受けた要求仕様をソフトウェア要求分析にて、要求モデルのフレームを機能項目リスト及び物理項目リストで形式化する。要求は、機能項目を同じ目的で分類してカテゴリを定義して階層的に機能項目を定義する。これによりカテゴリ内の機能項目に共通な部分仕様、名詞の定義が可能になる。各カテゴリ及び機能項目に対する分析/設計で必要な諸々の属性を定義可能な要求モデルのフレームを用いることで要求モデルの形式化を可能にする。

要求モデルに定義したカテゴリ及び機能項目に対応する機能仕様書の日本語表記の曖昧性の排除には形式記法を用いる。仕様記述は、条件記述と操作記述から構成される。これらの記述の意味を 1 ワードの記号を先頭行に付加することで形式化を進める。

形式記法を用いて機能仕様を定義すると機能仕様書で使用される名詞を見つけて名詞を定義する。更に機能仕様の文書の中で意味のある仕様のブロックを機能ブロックとして名前を付けて定義する。この定義を進めることにより機能ブロックを製品内で共通な機能ブロックや製品間で共通な機能ブロックの名称は、対象ドメインで汎用的に使用できる名称になり対象ドメインの要求を定義する言語を構築することが可能になる。これにより抽象度の高い記述で厳密な仕様定義が可能になり機能量の増大に対する仕様の複雑化の抑制と仕様構築スピードの向上が可能になる。

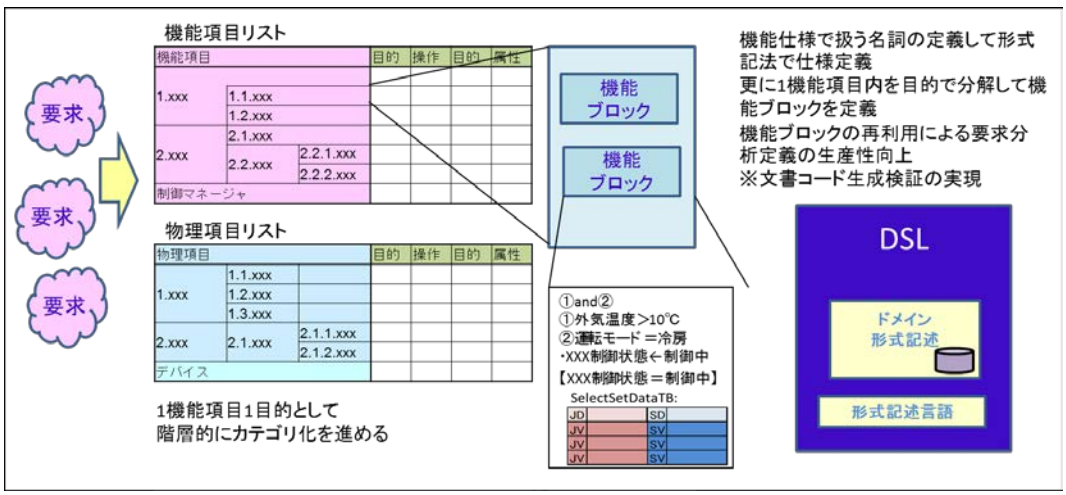

図 A-9-4　AOO による要求の形式化

（4）　機能の組合せの増加

製品に搭載する全ての機能項目を 1 機能項目 1 目的で機能項目リストに定義する。各機能項目に優先レベルを定義して、機能項目リストを横軸と縦軸でマトリクスを生成する。横軸の特定の機能項目に対して縦軸の他の機能項目実行時の競合仕様を定義することで想定可能な全ての機能項目の競合仕様の定義を可能にする。(図 A-9-5)

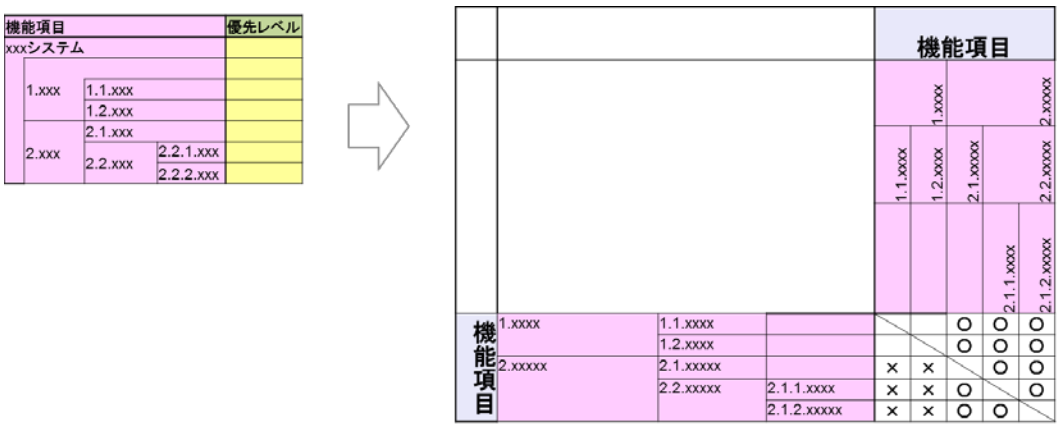

図 A-9-5　機能項目の競合仕様の定義

（5）　例外の検討漏れ

製品の基本機能の品質を確保することは比較的容易ですが、市場で様々な事象が発生して全ての例外発生の状況を検討しきれずシステム障害を発生するケースがある。

例外は、組込みシステムの実行基盤と組込みシステムを構成する物理的なデバイスや通信等で接続されるサブシステムにより生じる。対象製品の全ての物理項目を物理項目リストに定義して、各物理項目に対する発生しうる例外項目を定義する。定義した例外項目を例外項目リストに定義する。対象製品の全ての機能項目を定義している機能項目リストを横軸に縦軸に例外項目リストを定義することで想定可能な全ての例外事象に対応する機能仕様の定義を可能にする。　（図 A-9-6）

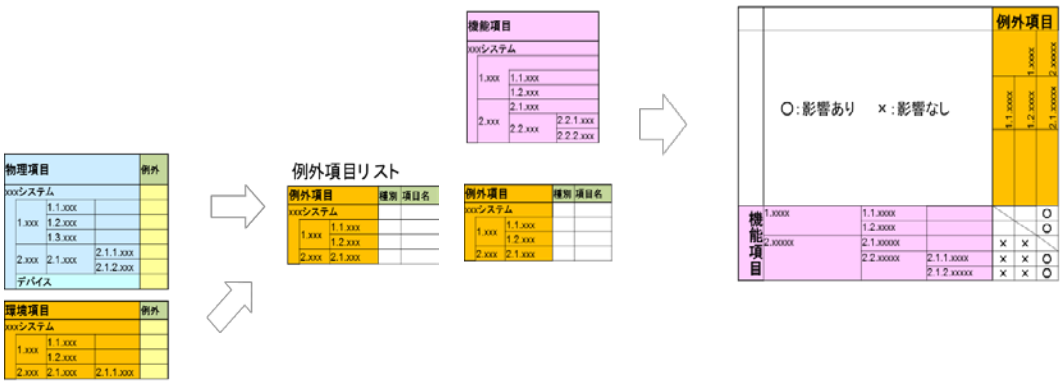

図 A-9-6　例外事象に対応する機能仕様の定義

（6）　時間制約の検討漏れ

時間制約は、組込みシステムを構成する物理デバイスや物理デバイスに接続されるサブシステムが制約を持っている。全ての制約を検出して CPU 処理性能内で確実に時間制約を守る必要がある。

製品の物理的な構成は、物理項目リストに定義されている。物理項目リストの各物理項目に対して時間制約を定義する。更に製品に搭載する機能は機能項目リストに定義してい

る。各機能項目に対する時間制約を定義する。(図 A-9-7) 物理項目リスト及び機能項目リストがソフトウェアに変換される。変換された各ソフトウェアは、変換元の物理項目リスト及び機能項目リストの時間制約が継承される。各ソフトウェアに対して時間制約に応じた起動イベントを正確に与えることで時間制約の検討漏れを防止する。

時間制約をタスク設計の基準とする。同じ時間制約のオブジェクトを同一タスクとする。タスク設計の指針は、時間制約を極力共通化してタスク数の増加を抑制することで資源の抑制と複雑化の抑制を実施する。

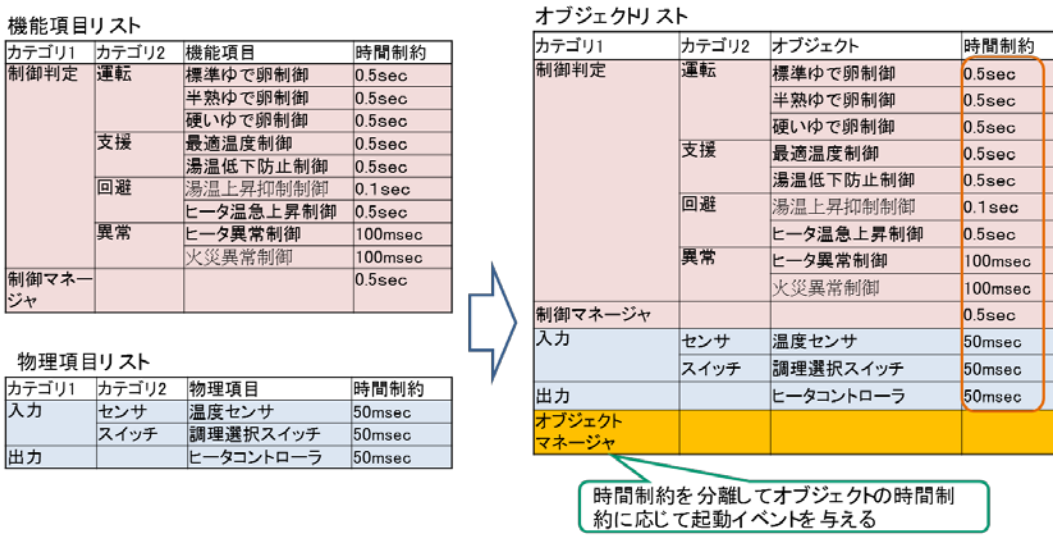

図 A-9-7　時間制約の検討

(7)　機能仕様の重複

AOO によるソフトウェアプロダクトライン(AOO_SPL)の技法について述べる。(図 A-9-8) 製品のバリエーションは、機能項目リスト及び物理項目リストの各項目の有無や各項目の機能仕様内の部分仕様（機能ブロック）が変動する。この部分仕様を機能ブロックとしてバリアントを定義する。変動する仕様の可変ポイントをバリエーションポイントとして定義する。バリエーションポイントは、<<バリエーションポイント名称>>で定義する。バリアントは、機能ブロックと同じで<機能ブロック名称>で定義する。

バリエーションにはイニシャル時にポインターで解決する動的とコード生成時にマクロで解決する静的の 2 つのタイプを定義することでバリエーションを解決する。

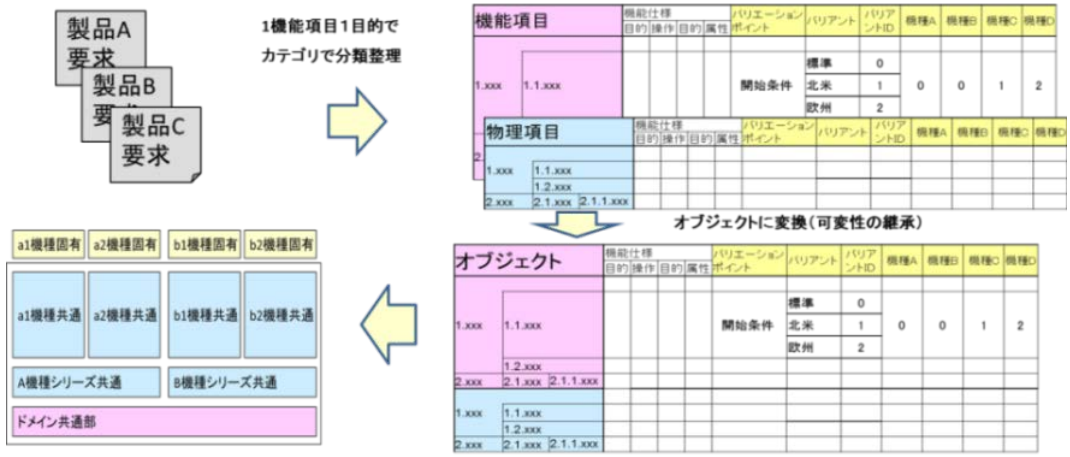

図 A-9-8　AOO によるソフトウェアプロダクトライン技法

3.3. 評価方法

3.3.1. 工数割合

要求分析で厳密に定義した文書を設計・テストへ変換することにより上流で工程の工数割合が増加して、設計及びテストの工数割合が減少傾向になることを評価する。

3.3.2. 不具合検出率

テスト実行以外の上流工程で不具合を未然に防止できているか評価する。今回の技法では、要求分析を厳密に定義した文書を設計・テストへ変換する技法を用い結果、分析での不具合の検出率が増加傾向になることを評価する。

3.3.3. 複雑化の抑制

要求分析で機能仕様書内の製品内で重複する部分仕様を機能ブロックとして分離、更に製品間で重複する部分仕様を機能ブロックで分離する。これにより機能仕様の記述量が減少傾向になることを評価する。

3.3.4. 生産性

複雑化を抑制して徹底した無駄取りによる生産性の改善傾向を評価する。

3.3.5. 品質

不具合の作り込みを抑制するフレームとタスクの実行により不具合を作り込まなくなる。不具合の残存率によりこれを評価する。

4. 適用計画と適用内容

4.1. 適用組織と計画

対象組織では、納期の制約の中、分析及び設計に十分な設計時間を確保することが困難となっている組織である。分析及び設計で定義した文書が曖昧で精度が無いためテスト実行時の基準文書の定義も不十分な状況にあった。対象組織では、ソフトウェアの不具合は、テスト工程でのみの検出が殆どで設計部門へ納品後の受け入れテストでも不具合の検出は少なからず発生していた。対象の組織の規模は、約 100 名規模で 2005 年から対象組織の流出不具合の真因を分析して対応策 2006 年に確立して 2007～2008 年で技法を全組織に適用した。

4.2. 適用内容

AOO 開発プロセスに基づき組込みソフトウェア開発で想定可能な課題を抑制する為の開発プロセスと様式を定義した。更にサブタスクに 1999 年～2004 年までのソフトウェア不具合の作り込みの真因を分析して実行すべきタスクを追加することにより対象組織で作り込まれる不具合要因を一掃した。

5. 適用効果分析

AOO を 2007 年から適用して現在は、全てのソフトウェア開発に適用して多くの成果を上げている。技法適用前の 2005 年では、分析/設計の工数割合は、10%以下で作成とテストに工数割合が 90%をしめていた。技法適用の結果、上流で不具合検出率が 97%で年々テストの工数割合が減少して分析の工数割合が増加している。殆どの不具合が分析で検出され試験では、殆ど不具合が検出できない状況となり高品質/高生産性を実現することができた。

技法の適用評価として 2009 年～2012 年のデータで評価する。この結果を 3.3 の評価方法に基づき技法の適用効果を分析する。(図 A-9-9)

5.1. 工数割合

年々テスト工数が削減され分析工数が増加して、分析の工数割合が約 53%、テストが約 20%で安定している。

5.2. 不具合検出率

分析での不具合検出割合が約 97%で安定。テストでの不具合検出が減少して、単体/結合/総合テストでの不具合は、1 プロジェクトに付き平均 10 件以内で収束している。

5.3. 複雑化の抑制

要求分析で機能仕様書内の機能ブロック化により機能仕様量を分析すると約 15%の削減が進んでいる。これにより機能仕様の記述量が減少して機能量の増加による複雑化を軽減できたと評価できる。

5.4. 生産性

技法の適用による複数プロジェクトで生産性の改善を評価した結果、130%以上の生産性向上が確認できた。分析で厳密に定義した文書を設計、実装、テストへ双方向に紐付けることにより徹底したムダ取りの結果、生産性が向上したと評価できる。

5.5. 品質

客先納品後不具合の発生は、抑制され不具合が殆ど流出していない状況が継続している。

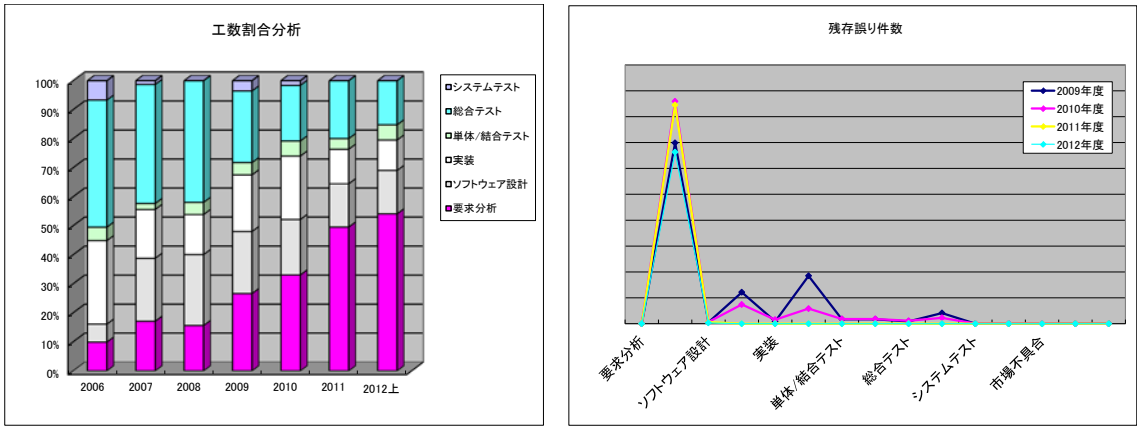

図 A-9-9　AOO 導入効果

参考文献

[1] 岩橋 正実:TECHI Vol.12　リアルタイムシステム実現のための自律オブジェクト指向,CQ 出版,(2002)

[2] 岩橋正実，満田成紀，鰺坂恒夫，中島毅：オブジェクトの自律化と競合解決に基づく組込みオブジェクト指向開発手法の提案，情報処理学会組込みシステムシンポジウム 2009 論文集，pp.81-86(2009)

A－10

MBSE[1]による双腕作業ロボット動作実行系のコンセプト設計[2]

1. 概要

本編では、多品種少量生産のロボット開発（短い製品サイクルの開発）に SysML[3]モデルを適用した、独立行政法人 産業技総合研究所（以下、同研究所とする）の事例を紹介する。

現在の産業用ロボットの多くは大量生産に最適化したラインを構築するためのもので、人とロボットが分離して運用される。ところがこのようなロボットは技術や用途が飽和してきているのが現状で、今後は多品種少量生産へのロボットの導入が期待されている。ただし、これまでは自動化で得られる恩恵よりもロボットの導入コストが大きく、導入が困難であるという問題があった。

これに対して、近年、多品種少量生産へのロボット導入の有力なアプローチとして、双腕など、人に近い構造を持ったロボットが注目されている。このようなロボットは人に近い構造を持つために、作業環境を少し変更するだけで、人間の作業者の代わりとなって作業を代替できる可能性がある。

このような人に近い構造を持つ多品種少量生産のためのロボットは、短い製品サイクルに対応することが求められる一方、導入を容易にして人と空間を共有して作業することが求められる。そのためには要求仕様の頻繁な変更に対応しつつ、センサ等を用いた人との安全な共存、すなわち機能安全が求めるウォーターフォール型の V 字開発プロセスを実現するという矛盾した要求に応えなければならない。

本取組みでは、MBSE の手法により、このような双腕作業ロボットシステムの設計プロセスを改善し、頻繁な変更と機能安全という矛盾した要求に応えることをもくろんだ。具体的には図 A-10-1 のような双腕作業ロボットのコンセプト設計の部分で、SysML モデルを用いて要求の分析およびアーキテクチャの設計を行った事例を紹介する。

---

[1] MBSE: Model-Based Systems Engineering

[2] 事例提供：独立行政法人 産業技総合研究所　知能システム研究部門　中坊 嘉宏 氏、花井 亮 氏

[3] SysML: Systems Modeling Language

図 A-10-1　対象システムのコンセプト

図 A-10-2 に対象とするロボットシステムの構造を示す。ロボットはシナリオで定義された手順にしたがって組立て等の作業を行う。その際対象物をカメラで認識し、その認識結果にしたがって動作を計画する。計画された動作軌道は、その下に示す動作実行系に送られる。動作実行系では与えられた軌道を補間し、目標位置を早い周期で下位のサーボドライバに送信する。以上の対象物の認識、動作計画はロボットの台座に収納された外部 PC で実行され、動作実行系についてはロボットの上半身に格納された CPU ボード上で実行される。

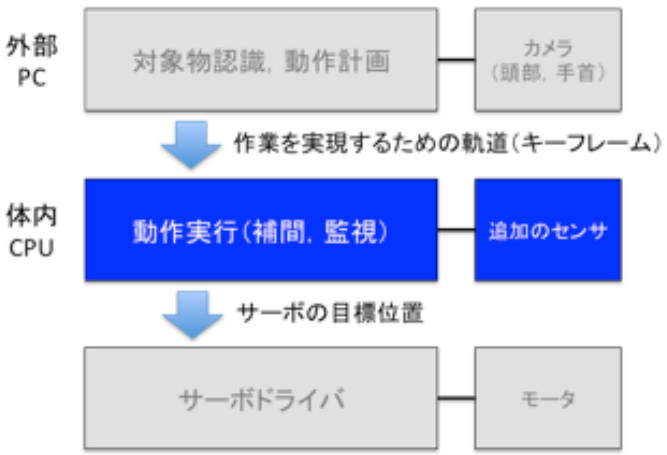

図 A-10-2　システム構造

2. 取組みの目的

2.1. 解決すべき課題

従来の産業用ロボットを導入する際には、適用する分野の経験が充分あり、また開発したシステムを長く使用することから、当初から全体の計画をたててひとつひとつ開発を進めるウォーターフォール型の開発プロセスが主として適用されてきた。

一方、本取り組みで対象とする、人に近い構造を持ち、人の作業の代替を目指したロボットシステムでは、作業目的の多様性ゆえに、実装を重視した、試行錯誤的な開発が進められているのが現状である。その結果、システムに対する要求や、要求される機能を実現するアーキテクチャについては、充分な議論が行われず、場当たり的に機能コンポーネントの設計と実装が行われるという問題があった。このような問題を解決するには、今後、多くの作業での利用を想定した、比較的汎用性の高いコンポーネント群を作成してそれを活かしつつ、顧客の個別の要求に対応するという形でシステムを実現しなくてはならない。また、そのシステム開発は、機能安全を満たしつつ、変化する要求にも対応できるものでなければならない。

2.2. 取組み前の状況

一般のロボットシステムの開発においては、多くのハードウェア、ソフトウェアコンポーネントの統合によるシステム開発の重要性が近年認識され、実装コンポーネントの組合せによるシステム開発が有効と考えられている。

その際、個々の開発者は、各自が担当した部分について、どのコンポーネントが何の目的で使われているか、それぞれがどのように相互作用しているかなどのシステム全体の設計が頭の中にあると考えられるが、それらの意図が開発者間で正しく共有されているかについては疑問である。またこれを補うものとしてコンポーネントに関するドキュメント、システムのデプロイメント情報や実行状態を監視・可視化するツールなどがあるが、ドキュメントは記述者の裁量の余地が大きすぎ、またデプロイメント情報や実行状態の監視・可視化ツールは、設計というよりも、実装やテストフェーズのツールであるという問題がある。また表記法についても、ツール独自のもので、SysML のように標準化されているものではない。

このような状況から、目的のシステムの本来満たすべき要求を明確にすることができず、結果として目的に対して必要以上に多くの機能を詰め込んだシステムが作られることも少なくない。どの要求に対してどのコンポーネントが使われているか、また、そのコンポーネントの選択理由が何であるかなどの議論を明示的に行うことは、あまりなされていない。

2.3. 関係する過去の取組みの内容、成果、取組み結果としての課題

今回の開発事例以前の取組みとして、同研究所が行った MBSE に関する類似の取組み事例を 2 件示す。それぞれに成果と課題があり、それらを取り込んで今回の開発を行った。

(1) 複数の移動ロボットシステムの SysML を用いたモデリングと、SysML モデルを用いたシステムアーキテクチャの比較

内容：

自律移動機能を備えた AGV[4]（無人搬送車)、車いす、シニアカー、駐車場管理システムといった複数のロボットシステムを対象に SysML によるモデリングを行って、各システムのアーキテクチャの比較、コンポーネント共通化の可能性について検討を行った。

具体的には、それぞれのシステムについて、要求図、コンテキスト図、外部コンポーネント図、ユースケース図、アクティビティ図、内部コンポーネント図を作成した。

成果：

SysML モデルを用いることで、移動ロボットシステムの設計を高い抽象度で理解し、システムの違いや、再利用性を向上させる上での問題点などの議論が行い易くなった。具体的にはセンサやアクチュエータ、CPU などのデバイスの共通点と違っている点、それぞれのコンテキストによるシステムの利用方法の前提の違い、自律移動という大きなレベルでの要求の共通性などが見える化された。

実際、移動ロボットの自律移動のためのシステムの構成方法についてはかなり確立されてきており、上位の要求仕様や、最下位のデバイスのレベルでは共通性が高かったが、その間のシステムの実装レベル、ソフトウェアのレベルでは、やりとりするデータの書式の違いや、自己位置推定のアルゴリズム部とセンサからの入力データの処理部とが十分分離した設計になっていないなどの点で、システム同士の構成と実装方法の違いが大きく、共通化は容易ではないことが確認された。

課題：

分析を目的としてリファクタリングによるモデル化を行ったため、取組みの結果は現状の実装や設計には反映されず、次回以降に開発する際の課題や教訓が明らかになったのみであった。

(2) 機能安全設計による高信頼台車の開発

内容：

機能安全の国際標準規格 IEC 61508(SIL2(安全度水準レベル 2))に準拠可能な安全性を備えた生活支援ロボットの中核ユニットの開発を想定し、制御系の冗長化による単一故障耐性を持つ高信頼車両（電動車いす）の試作を行った。開発は SysML を用いたオブジェクト指向のシステムエンジニアリング方法論(OOSEM[5])を用い、機能安全に従った設計文書記述を通して実施した。ドキュメント群はトレーサビリティの確保を重視し、ソフトウェアにはロボット用ミドルウェアプラットフォームである RT

[4] AGV: Automated Guided Vehicle

[5] OOSEM: Object-Oriented Systems Engineering Method

ミドルウェアの高信頼版で、機能安全 SIL2 認証済みとなる RTMSafety[6](セック社)を活用してソフトウェアのコンポーネント化を進めるなど、安全なロボットの開発を効率よく行うための試みを進めた。

成果：

ロボット車両の冗長系設計を、高信頼というコンセプトから実装までの整合性をとりながら実機に落とし込むことができた。また検証についても、V 字モデルに従った設計と検証とを対にして妥当性確認と試験を行うことができた。RTMSafety を利用した部分のソフトウェア安全性が確保できた。

課題：

トレースツールが未完成であったため、モデリングツールとトレースツールとの連携が不十分で、トレース作業の手間が増大し、またミスが入り込む余地が残った。おなじくトレースの手間がかかるため、仕様の変更が簡単には行えなくなった。

2.4. 本取組みの目標

以上の反省を踏まえ、本取組みでは、上記課題解決の第一歩として、これまで十分注意が払われてこなかったシステム開発と設計時の要求分析に重点を置き、またその変更を容易にする方法の開発を試みた。

具体的な目標は、

（1）　SysML を用いて要求や機能との関係を具体的に表現すること

（2）　分析した要求を満たすアーキテクチャ設計を行うこと

（3）　これらの工程の成果物や参照された文書間のトレーサビリティを実現すること

（4）　上記工程は、要求が追加された際でも効率よくシステムの拡張ができることとした。

2.5. 目標設定のために実施した課題解決のための仮説設定

MBSE の手法を適用することで、双腕作業ロボットの動作実行系の構成、振舞い、要求とシステム要素との対応が見通しの良い形で整理される。また、機能追加を想定したアジャイル開発が、短い商品サイクルに対応する必要があるロボットシステムでは有効である。以上の 2 点を仮説として、これを検証する取組みを行った。

[6] 機能安全の国際規格 IEC 61508 認証取得 済み RT ミドルウェア

3. 取組みの対象、適用技術・手法、評価、計測

3.1. 対象製品・プロジェクト、適用工程

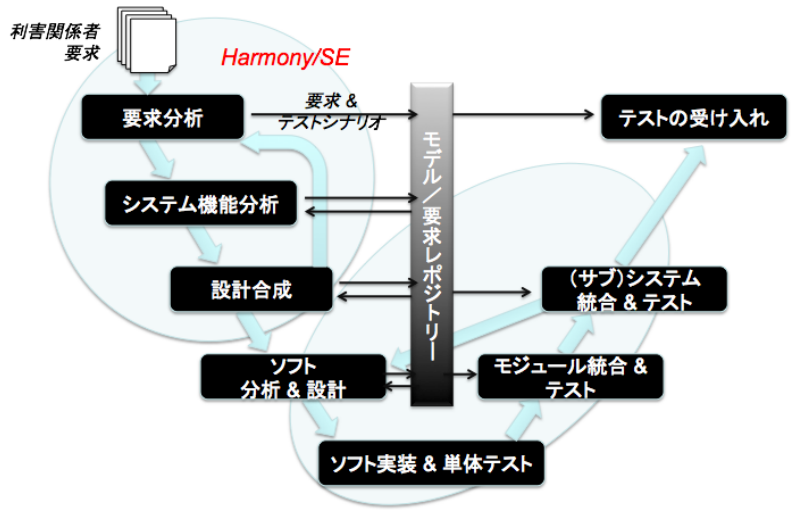

図 A-10-3　V 字モデルと Harmony/SE

設計対象は、人に近い構造、形状をもつ双腕作業ロボットの動作実行系である。ロボットのハードウェア大部分、ソフトウェアの一部は既存のものをそのまま使うことを想定している。これに、ハードウェアではセンサを追加、ソフトウェアにおいては作業目的にしたがった動作を実行する部分を再設計した後、安全機能、使用性を高める機能を追加することで拡張されたシステムを設計する。

安全機能は、ロボットの作業空間を距離画像センサで監視し、人が作業空間内にいる場合にはロボットの動作を停止する機能である。使用性を高める機能は、認識や操作の失敗により作業を継続できない状態（チョコ停）に陥った場合に、人がロボットに直接触れて手先の位置を変える、あるいは操作対象物の位置を変えることにより、短時間でシステムを復帰させる機能である。

適用工程は、図 A-10-3 に示す V 字モデルの左上部分である。システムに対する要求を洗い出すところから始まり、その要求を分析した後、システムの機能を分析し、その要求を満たすためのアーキテクチャの設計を行った。

3.2. 方法論（どんな技術・手法を用いて実施したか）

本プロジェクトでは、要求が追加される場合にシステムを拡張していくことを想定した技術という理由で、MBSE プロセスの 1 つである Harmony/SE プロセスを用いて進めた。Harmony/SE プロセスは、IBM 社のベストプラクティスを集約、体系化したものであり、図 A-10-3 に示すように要求分析、システム機能分析、設計合成のフェーズを反復的に行う

システムズエンジニアリングプロセスである。反復はユースケースベースで行われる。

要求分析の手順は次の通りである。

(1)　要求の書き出し

プロジェクトメンバによるブレインストーミングの要領で、要求を自由に書き出した。要求は、ひとつの木構造でうまく表現することが難しい。また分類しながら要求を書き出すことを考えると思考が制限されやすい。人によって分類の仕方が異なるという問題もある。したがって、要求書き出しの段階では要求間の関係は気にせず、単純に表の形に列挙した要求を出力とした。

書き出された要求は、「pick&place 作業を行う」、「人が近づいたら動作を止める」といったステークホルダ要求に該当するもの、「動作を止める」方法として「センサで人の接近を検出する」といったシステムの特定の機能に関するもの、あるいは「そのセンサには距離画像センサを用いる」といった実装レベルのものが混じっていた。

技術者が考えた場合、実装寄りの要求が多く書き出される傾向がある。また、分析のスコープから外れる要求も多数含まれていた。その理由として、ロボットのハードウェア、ソフトウェアとして既存のものがあり、そのイメージが強かったことが挙げられる。関連して、この段階の分析ではスコープが明確でなく、メンバ間で認識のずれがあったことも影響している。

(2)　モデリングツールへの要求の取り込みとユースケースの識別

複数人によって書き出された要求を集め、整理するときの手がかりとなるタグをつけることをエクセルの表の上で行った。タグは 3 種類である。1 つ目は要求が機能要求か非機能要求であるかを示すタグ。2 つ目は要求の粒度に関するもので、システム全体に対する要求か、システムの特定の機能に関する要求か、何らかの実装方法に関連する要求かを示すタグ。3 つ目は ISO/IEC9126-1:2001 に定義されたソフトェアの品質カテゴリ（機能性、信頼性、使用性、効率性、保守性、移植性）を示すタグである。

その後、エクセルファイルをモデリングツールにインポートした。エクセルファイルはモデリングツールと同期して管理される。以後はモデリングツール上での作業である。

まず、インポートした要求は上記の品質カテゴリによってパッケージとして分類、管理した。

そして、システムに対する機能要求をもとにユースケースおよびアクターを識別し、ユースケース図で表現した。ユースケースについてユースケースシナリオをエクセルファイルで記述した。このとき、オペレータの作業手順についてはロボットのユーザマニュアルを参考にした。

(3)　要求の分析

ここでは要求の関係を整理し、構造化する。これは次の手順で行った。

注目するユースケースに関係する要求を集めて要求図を作成する。このとき、粒度タグの情報を利用し、要求間の関係を作成する。基本的にはステークホルダ要求を特定機能の要求が、また特定機能の要求を実装に関する要求が、それぞれ refine ないし derive する関係になる。ここで、重複する要求の削除、および曖昧な要求の詳細化を行った。メンバ間での認識の齟齬等により矛盾する要求が存在する場合は議論を行い、どちらを採用するかを決定した。

(4)　システム機能分析

システムの機能要求をシステム機能（操作）の一貫した記述に変換する。具体的には、注目するシステムレベルのユースケースを検証可能なモデルに変換する。これには Harmony プロセスにおいて、3 通りのワークフローが定義されている。その中で、本プロジェクトでは、状態マシン図により状態ベースの振舞いを定義し、その状態マシンの実行パスとしてユースケースシナリオであるシーケンス図を作成するワークフローで分析を行った。

ここでは、状態マシン図の実行を通して、振舞いの妥当性を視覚的に確認するとともに、振舞いを定義する中で確認された新規要求または派生要求は(2)におけるパッケージ構造に追加する。また、機能要求とそれを実現している状態遷移図の要素の間にトレースをとる。

(5)　設計合成

一連の所定の機能要求および品質要求を満たすアーキテクチャの設計を行う。

トップダウンにアーキテクチャ分解により構造を定義、ブロックに割り当てられた操作を各パーツに割り当てていく。構造は特定の（レガシー）アーキテクチャに基づいて決める場合、複数のデザインコンセプトに対しトレードオフ分析をして決める場合がある。

本取組みでは、次の二点をデザインコンセプトとして構造を決めた。

- ソフトウェアの実装にロボットのミドルウェアの 1 つである RTMSafety を利用する
- ロボットの状態を 1 つの構造体に詰めてコンポーネント間で循環させるレガシーアーキテクチャの採用

システムの分解は、まずブロック定義図によって表現した後、分解された各要素間の関連を定義するために内部ブロック図を用いてシステムを表現する。そして、分解された各システムブロックに対し、ポートとインタフェースの定義、状態マシン図による振舞いの定義を行う。

(6)　トレーサビリティの実現

2 種類のツールを用いてトレーサビリティの構築を行った。

1 つはモデリングツールを用いて構築した、設計の理解や、分析の手助けとするためのモデル要素間のトレーサビリティである。これはモデリング中に随時更新していった。

もう 1 つはシステム要求仕様書、設計仕様書、各種参考資料、根拠資料等の間での文書レベルでのトレーサビリティである。これは IEC61508 規格を想定して行った。この文書レベルでのトレーサビリティには、TERAS (Tool Environment for Reliable and Accountable Software) を利用した。

(3)から(5)はユースケース単位で分析、設計を進めた。具体的には最初にシステムの主要機能である「作業目的に沿った動作実行」というユースケースに着目し、(3)から(5)を行った。その後、同様に「距離画像センサを用いたセンシングによる安全確保」のユースケース、チョコ停 [7]状態から人がロボットに直接触ってロボットの作業を継続させる」ユースケースについて、(3)から(5)を行い、システムを機能拡張していった。

3.3. 解決のため採用したツール

Enterprise architect、 MagicDraw、 Rational Rhapsody の 3 つのモデリングツール、トレーサビリティツール TERAS を検討した。

最初、モデリングツールに関してはメンバ間で使い慣れているツールが異なるので、複数のツールを使い連携することを検討した。しかし、複数設定やフォーマットによりエクスポート、インポートを試みたが上手くいかず、データ移行の困難さが明らかとなった。特に、モデル図のレイアウトの崩れは大きかった。したがって、モデリングツールは 1 つを選択することとした。

Enterprize architect は仕様の独自性が強い、MagicDraw はモデルの文法チェッカー、Java なので Linux や Mac でも使えるのが便利などの意見が出た。本取組みでは、モデリングを主に担当するメンバのなじみのツールであった点、ステートマシン図のシミュレーション機能を試すという点から、Rational Rhapsody を採用した。

トレーサビリティツールとしては、以前の取組みで利用していた TERAS を採用した。Reqtify などの類似コンセプトのツールや、Rhapsody との連携に優れた Doors も候補として挙げられた。TERAS の採用には、低コストのツールチェーン実現の可能性を探るという意図もあった。

4. 取組みの実施、及び実施上の問題、対策・工夫

4.1. 計画、準備

この事例におけるプロジェクトは、メンバが 6 人で期間が 3 ヶ月弱であった。メンバの大

[7] 軽微な原因による一時的な作業の停止

半は、ロボット研究者である。ただ、過去の取組みを通して、ある程度 SysML を用いたモデリングや設計プロセスに関する知識は持っている。

また、MBSE、特に Harmony/SE に精通したエンジニアを外部からが一人招き、参加してもらった。

ツールを用いてプロジェクト管理を行う案も出たが、ツールの導入、プロジェクトメンバがその使い方に慣れるまで時間を要することからその案は見送られた。

既存の双腕作業ロボットをベースに、要求やシステム設計を考えるために必要な資料を集めた。特に重要なものは以下の 2 点である。

- ロボットのマニュアル

  ロボットの構造等の設計情報、操作デバイス、ステータス表示 LED など運用に関係する情報が記載されている。ロボットのユースケースや要求を検討するために必要な資料として用意した。

- PHA[8]安全分析レポート

  過去に行った類似ロボットに対する安全分析の資料である。米軍規格 MIL-STD-882 を用いて分析を行ったものである。本プロジェクトにおいて PHA を行うための参考資料として用意した。

4.2. 実施（具体的取組み内容、活動）

MBSE に精通する 1 人のメンバが進捗を管理し、主導する形でプロジェクトを進めた。その管理者がスケジュールを立て、進捗に応じて他のメンバに作業の割り振りを行い、2 週間毎に開かれる全員が集まるミーティングにおいて要求や設計モデル等のレビューおよびスケジュールの確認を行った。

別の 2 名は、設計対象の動作実行系以外のソフトウェアや、双腕作業ロボットのハードウェア自体に詳しく、要求の書き出しやシステム設計の多くを分担した。

残りのメンバはロボットにも詳しい安全の専門家とシステム開発の専門家であり、要求の書き出しと上記ミーティングにおいて、設計レビューやツール選定、安全に関する議論に参加した。

用意した資料や、作成した SysML モデル図などの成果物は Dropbox で常に共有できる状態とした。Dropbox が備える機能以上のバージョン管理は行っていない。モデリング途中のモデル図が大量にできたため、整理の仕方を考える必要がある。また、モデルの編集は 1 人の手元で集中して行ったため問題にならなかったが、複数人でモデリングを行うときには、このやり方では難しいと思われる。

4.3. 結果の分析・まとめ

今回の取組みでは、既存のロボットやソフトウェアなどが大きな部分を占め、設計対象で

[8] PHA: Preliminary Hazard Analysis

はないが、設計対象が依存する部分が多かった。具体的には、ロボットのハードウェア（機械系と電装系）はそのまま使用し。ソフトウェアについても作業を実現するための画像処理や動作計画、下位の位置制御を行うサーボドライバなどは対象ではなかった。設計対象は、作業実現のための軌道が与えられたときに、実時間で監視しながらその軌道を実行する部分と、追加されるいくつかのセンサハードウェアであることが、後でわかった。図 A-10-2 はプロジェクト終了後に整理したものである。

それらの注目する部分以外をどの範囲まで、またどの程度詳しくモデリングすべきかの判断が難しかったが、要求やシステムの振舞いを分析していく中で、具体的になっていった。

上記のように、この取組みを通してシステム構造が見通しよく整理された。

結果的に、対象としたものが作業内容に依存しない安全機能や、使用性向上機能であったため、ロボットの構造や作業手順などタスクの実行に関わる部分が SysML モデルには現れていない。この部分をモデリング対象とすると、ハードとソフトを含むシステム全体を高い抽象度で表現、分析する MBSE の有効性の議論として面白いのではないかと考えている。

人材育成、意識改革等という観点では、頭の中にあることでも、モデルという形で表現し、モデル図を見ながら議論を行うことで、仕様が曖昧な部分や、他のメンバと意識共有できていない部分に気づくことができたというプロジェクトメンバからの反応であった。やはり、自分達が考えている事例で具体的に示されると説得力がある。

5. 達成度の評価、取組みの結果

5.1. 対目標

- 目標 1（SysML を用いて要求の関係を具体的に表現すること）について

  要求をいくつかの観点で整理することで、要求間の関係を理解しやすい形で表現することができた。分析の過程で実際にいくつかの矛盾する要求の解決や、曖昧な要求の明確化を行うことができた。しかし、一貫性や検証可能性など要求の品質について十分な分析をするには至っていない。

  また、設計対象のスコープが曖昧であったため、今回の分析対象と関係ない要求が多数書き出された。4 割程度の要求はスコープ外であった。前提条件となっているロボットそのものの機能や性能に関するもの、設計対象としない故障耐性に関するもの、ロボットの作業を設計するための要求などが含まれていた。逆に、必要な要求で不足しているものも多かった。これらについては、振舞いを分析する中で追加されていった。また、大部分が機能要求で、非機能要求は少なかった。

- 目標 2（分析した要求を満たすアーキテクチャ設計を行うこと）について

  システムの構造と一部のコンポーネントの振舞いを表現するところまで行った。また、モデリングが軌道に乗るまで時間がかかった。試行錯誤する中で多数のモデル図を描いたが、成果物として必要な大部分のモデルは 3 ヶ月の期間の内、最後の 1 ヶ月

程度で作成したものである。今後このモデルをベースに別のシステムを作っていくことを考えると、モデルなら何でも良いのでなく、上手く再利用できるものを作ることが重要と感じた。

モデルを導入することでスコープが定まって、今回対象としていた動作実行にかかわる層と、作業実行のための認識や動作計画に関する層、下位のサーボモータを制御する層といったシステム構造が見えてきた。

・ 目標 3（これらの工程の成果物や参照された文書間のトレーサビリティを実現すること）

要求と設計要素間のトレーサビリティを実現することで、コンポーネントやコンポーネントの特定の振舞いがどの要求から来ているかが見える形に整理できた。

文書間トレーサビリティ構築のために、モデリングツールからモデル図を pdf 形式でエクスポートし、TERAS の pdf プラグインを利用してインポートした。インポート時には正規表現で REQ などの単純なキーワードで始まる要素をトレーサビリティの起点としてインポートするだけである。したがって、モデル上で作成したトレース構造と同じものであっても改めて TERAS 上で作成する必要があった。

・ 目標 4（上記工程は、要求を追加される場合に、効率よくシステムを拡張していけること）

Harmony/SE のモデリングプロセスにしたがって、システムを拡張していくことはできた。

ただし、設計においては 1 つのアーキテクチャを経験的に選択している。機能追加時の影響の大きさについて他のアーキテクチャとの比較を行っていく必要がある。

6. 今後の取組みと考察

本取組みでは双腕作業ロボットの動作実行系（図 A-10-2）にモデリングスコープを設定し、その動作の実行中に働く安全機能などを追加することを目標とした。

この階層の上には、具体的な作業目的を達成するための機能がある。また、このようなロボットの使い方として、作業目的に応じてハンドやセンサ、ツール等のハードウェア的な拡張が行われる。更には、制御方法や作業手順も変化する。今後は、これらをモデリングし、要求と紐づけることで、要求を満たす各種拡張システムを効率的に設計する方法論について取組む予定である。

A－11

仕様記述言語 VDM++ を用いたシステムの仕様の記述 [1]

1. 概要

本編では、IC チップファームウェアの開発にあたり、形式仕様記述手法を導入したフェリカネットワーク株式会社（以下、同社とする）の事例を紹介する。

同社では、おサイフケータイに搭載するモバイル FeliCa IC チップファームウェアの開発を行っており、形式仕様記述手法を導入し、開発の成果を上げるのと同時に、手法適用の効果を確認している。ここでは、開発プロジェクトにおける形式仕様記述手法適用の成果を、仕様の開発効率、仕様を活用したテストや仕様自身のテストの結果、仕様に関わるコミュニケーションの分析、導入障壁をはじめとする問題点や課題、今後の展望等を交えて紹介する。

2. 課題と目標

本事例の開発対象は、モバイル FeliCa IC チップのファームウェアである。

モバイル FeliCa [11] は、ソニー株式会社が提唱する FeliCa 技術方式に基づく非接触 IC カード技術を発展させ、携帯電話にモバイル FeliCa IC チップを搭載することにより実現した、携帯電話を通してネットワーク上の世界と現実の世界とをつなぐ、システムとマルチアプリケーションプラットフォームの総称である。システムは、モバイル FeliCa IC チップを搭載する携帯電話と、携帯電話通信網経由で接続する FeliCa サーバ、一般の利用者が携帯電話を FeliCa カードとして店舗や駅構内などでかざすリーダ・ライタから成る。モバイル FeliCa システムの概要を図 A-11-1 に示す。

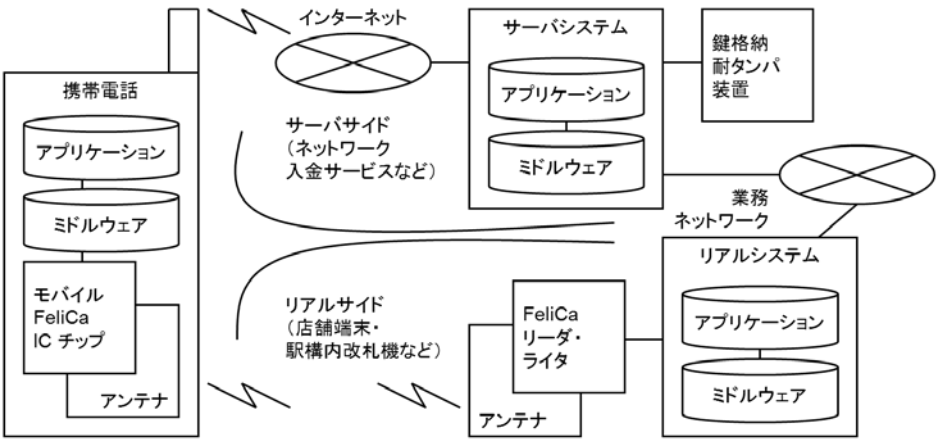

図 A-11-1　モバイル FeliCa システム

[1] 事例提供：フェリカネットワークス株式会社 開発 2 部 2 課　栗田 太郎 氏、中津川 泰正 氏

FeliCa のカードや携帯電話に組み込まれる IC チップは、悪意のある攻撃者による情報の読み出しや改竄、通信路の盗聴を想定し、様々な攻撃への対策を施した耐タンパセキュリティチップである。そして、本事例における開発対象である、IC チップに組み込まれるソフトウェアは、ファイルシステムやコマンドインタプリタ、無線や有線の通信インタフェースを司るハードウェア機能ブロックの制御ドライバ等から構成するファームウェアである。

FeliCa は、国内において、公共交通機関の乗車券や電子マネー等に広く応用されており、社会基盤の構成要素としてその一翼を担っている。モバイル FeliCa IC チップは、様々な事情や制約から、数多くの携帯電話に取り外しができないよう直付けされている。更に、モバイル FeliCa IC チップファームウェアは、製造コストや運用面の都合により、プログラムを書き換えることができない IC チップの ROM 領域に格納している。

以上の背景を踏まえて、システムやソフトウェアの開発に当たっては、求められる要求や定めた正常系の仕様を満たした設計や実装及びテストを行うのみならず、セキュリティ上のリスクを可能な限り低減するための設計や、準正常系や異常系を含めた完全な仕様の記述と設計、実装、テストを網羅的に行うことで、商用品質を確保する必要がある。

本事例におけるプロジェクトにおいては、また、一般に産業界での業務としてのシステム開発においては、開発対象や体制が大規模化、複雑化していく中で、複数人で効率良く品質の高いソフトウェアを開発し、プロジェクトやチームで責任を果たしていくために、組織立って、様々な工学の成果や工夫を組み合わせていく必要がある。

このような状況の中で、ソフトウェアの開発現場では、曖昧な仕様に起因するトラブルが多く、課題となっている。先行する工程での問題は、後続の工程に先送りされ、より大きな問題へと発展していく。後続の工程での欠陥の修正コストは、先行する工程での修正コストよりも大きくなり、予定通りの開発や、システムの信頼性や安全性の確保は不確実なものとなる。

システムの開発により何を作るのかを表すべき仕様の厳密な記述は、開発の上流工程における成果物の品質確保のために不可欠である。「何を」作ろうとしているのか、「何を」作ったのかを、仕様として厳密に記述し、これを維持することにより、開発と運用・サポートを正確に、精密に行うことができる。これにより、設計、実装、テスト、運用、サポート等の仕様策定の後続の工程、プロジェクトマネジメント、コミュニケーションを建設的に収束させることが可能となる。

自然言語で記述した曖昧で非論理的な、時には情報が欠落した、検査やレビューが容易ではない文書に基づいて開発を行うことは困難である。また、プログラミング言語は、プログラムの記述のためにある。プログラムは、コンピュータが具体的な処理を実行するための詳細な情報を、処理を実行するための様々な事情を考慮して記述するコードとデータであり、様々な役割を持つチームのメンバが、プログラムとして書かれた論理的な情報を起点として開発を行うことは難しい。仕様の論理から記述のための記法への割り付けが明確な仕様記述

言語を用いることにより、仕様を文書として素直に表現したり、記述した仕様を一意に解釈したりすることができるのである。

様々な抽象度の仕様を厳密に記述し、これを検証するための手段として、数多くの仕様記述言語や、記法、ツール等が提案されており、これらを活用した開発手法の一つとして、モデル規範型の形式手法［1、7］がある。手法や言語、ツールの活用により、チーム全体で議論の対象としたり､参照したりすることができるシステムのアーキテクチャやデータの構造、処理等を、開発の拠り所となる仕様として記述することが可能となる。

本事例では、新たにモバイル FeliCa IC チップ上のファームウェアを開発するに当たり、以上の課題を踏まえ、形式的に仕様を記述する手法を導入することにした。導入の当初に、形式仕様記述手法の利用により達成したい目標は以下のとおりである。

(1)　厳密な仕様の策定

厳密な仕様を策定する。形式仕様記述手法導入の背景に照らし合わせ、プロジェクトの予定通りの進捗と、メンバ間の、そして利害関係者とのコミュニケーションを円滑にするための仕様を策定する。

(2)　開発プロセスの策定と導入

仕様策定、設計・実装、テストを行う開発全体プロセスを検討する。特に仕様の分析・検討・記述・テストのためのプロセスの策定と導入を行う。自然言語や UML (Unified Modeling Language) によって記述した概要仕様、形式仕様、仕様に基づく設計を多方面から分析及び精査できるようにするとともに、形式仕様と実装をもれなくテストできるプロセスを策定する。

(3)　仕様の多方面からの分析・精査

検討したプロセスに則り、概要から段階的に詳細化した形式仕様を多方面から分析・精査・単体テストを行うことで仕様の記述レベルを高め、問題を後工程に先送りしないようにする。

(4)　仕様のテスト

開発環境上で動作するプログラムや IC チップとともに動作する形式仕様のテストが行えるようなテスト環境を構築し、これを用いるテストを行う。形式的に記述した仕様をもテストできるテスト環境を構築することで、形式仕様と同様の挙動の IC チップの開発、テストを行うことができるようになる。これにより、仕様に基づいたテストが行えるようになるとともに、仕様と実装の差異の厳密な確認が可能となる。

(5)　コミュニケーションの促進

形式仕様記述手法をプロジェクト内でのコミュニケーションツールとして位置付ける。これにより、実装・テストを含めた開発プロセス全体を俯瞰しながら、仕様策定についてプロジェクト全体で意識・考察できるようになる。メンバがこれらの目的を意識しながら開発を推進することで、一般的なソフトウェア開発の現場における課題に対処できるようにする。

3. 形式手法と形式仕様記述手法の導入

3.1. プロジェクトの概要と開発環境

本事例では、前節で述べたモバイル FeliCa IC チップファームウェア及び、FeliCa IC チップが組み込まれた携帯電話を運用するための周辺ツールの開発を行った。

プロジェクトに所属するメンバは 56 名、平均年齢は 30.8 歳、形式仕様記述手法に関する知識や経験のあるメンバはいなかった。

モバイル FeliCa IC チップファームウェアの実装に用いる言語は、C 言語、C++ 言語、アセンブラ言語である。テスト環境や周辺ツールなどの開発を含めると、C# 言語、Ruby 言語等が加わる。

また、開発ターゲットであるスマートカード用セキュリティ IC とその開発環境をそれぞれ複数準備し、各 IC 用のファームウェアは、複数種類の CPU 上で同一の動作をするよう、設計、実装している。これにより、国内のほぼ全ての携帯電話に搭載される IC チップとして必要な製造と供給における様々なリスクを軽減できる。

3.2. 形式的な仕様の記述と検証

形式手法とは、数理論理学に基づく科学的な裏付けを持つ言語を用いて設計対象の性質や機能を表現することにより、ある側面の仕様を厳密に記述し、開発工程で利用する手段の総称である。形式的な仕様の記述を行うことで曖昧さを取り除くと同時に、仕様書の機械処理が可能となり、システムの信頼性や安全性の確保等に向けた様々な可能性が開ける。

ここで、形式手法の適用レベルの定義を引用する [3]。本事例で紹介する適用はレベル 0 である。

レベル 0: Formal specification。 形式的に仕様を記述する。数理論理学を基礎とする記法を用いて厳密に仕様を記述する。この記述を元にプログラムを開発する。

レベル 1: Formal development and verification。 形式的に開発と検証を行う。段階的詳細化により仕様からプログラムを作成したり、仕様やプログラムの性質を証明したりする。

レベル 2: Machine checked proofs。 機械支援による証明を行う。定理証明器や証明支援器を用いて、仕様やプログラムの性質を証明する。

近年、システムの機能安全やセキュリティに関する国際標準規格や、各国の定めるガイドラインにおいても、形式手法の利用が推奨されている。例えば、機能安全の規格として制定された ISO/IEC 61508 においては、電気・電子関連の国際安全の基準として、安全度（SIL: Safety Integrity Level）が 4 段階に区分されており、すべての段階において準形式的な手法の利用を、SIL 2 以上では形式的な手法の利用を推奨している。

更に、例えば、ISO/IEC 15408 情報技術セキュリティ評価のためのコモンクライテリア

(CC: Common Criteria) セキュリティ評価認証制度においては、情報技術を用いた製品やシステムが備えるべきセキュリティ機能に関する機能要件と、設計から製品化に至る過程でセキュリティ機能が実現されていることを確認する保証要件を網羅した要件が規定されており、これらの要件に基づき、ソフトウェア開発や開発拠点等に関係する文書とソースコードが第三者によりレビュー、評価される。評価保証レベル（EAL: Evaluation Assurance Level）は、実装の確からしさの評価方法について、7 段階に区分されており、EAL 5 以上では準形式的な、EAL 7 では形式的な設計と検証が求められる。

3.3. モデル規範型の形式手法

現在、ソフトウェアの開発現場で実用化されている主な形式手法には、モデル規範型とモデル検査の二つのアプローチがある。

モデル規範型では、集合論や命題論理、述語論理等を基礎として、情報の構造や、ある状態から他の状態への変化をモデル化しながら、不変条件、事前条件、事後条件を記述する。これにより、専用言語の利用によるコンパクトな仕様の記述、仕様の検査による曖昧さの除去、仕様の実行、テスト、回帰テスト、定理証明等が可能となる [4]。

モデル規範型の一つである VDM (Vienna Development Method) は、1970 年代の前半に IBM のウィーン研究所で、プログラミング言語 PL/I (Programming Language One) の仕様記述や、コンパイラの検証のために開発された手法が基になっており、1996 年に、汎用的な仕様記述言語として、VDM-SL (VDM Specification Language) [5] が ISO の標準となった。

更に、VDM++ 言語 [6] は、VDM-SL に対して、主にオブジェクト指向の拡張を行った言語であり、これにより、様々な抽象度で、モデリングから仕様記述までを大規模、広範囲に行うことができる。

一方のモデル検査は、検証対象を有限状態遷移機械として表現したモデルと、モデルの性質を記述した論理式に矛盾がないことを網羅的に検査する手法である。従来ハードウェアの設計や通信プロトコルの検証等に活用されていたが、実用的なツールの開発とコンピュータの処理性能の向上により、ソフトウェアの開発現場での適用が可能となった [10]。

3.4. 形式仕様記述手法の導入に向けた検討

形式仕様記述手法の導入に先立ち、国内の専門家と議論を重ねた結果、形式仕様記述言語 VDM++ [6] と、これを用いたモデル分析・仕様策定・仕様単体テストを支援する統合開発ツール VDMTools [2] を導入することとした。VDM++ 言語は、国際標準規格 ISO において標準化されている形式仕様記述言語 VDM-SL [5] に対して、主にオブジェクト指向に関して拡張を行った汎用形式仕様記述言語であり、モデルの記述から仕様の動作まで広範囲をカバーする。VDMTools は、VDM-SL、VDM++ 言語により記述した仕様のモデル分析・仕様策定・仕様実行・仕様単体テストを支援する統合開発ツールであり、

(1) 仕様の構文検査
(2) 仕様の型検査
(3) 証明課題の生成
(4) 実行可能仕様の逐次実行とデバッグ支援
(5) 実行可能仕様のコードカバレッジ計測
(6) 実行可能仕様から C++ 言語や Java 言語への変換
(7) Java 言語から VDM++ 言語への変換
(8) 各種 CASE ツールとの連動
(9) 仕様の清書

の機能を有している。

本事例においては、上記機能のうち、(1)、(2)、(4)、(5)、(8)、(9) を利用した。中でも、開発当初から特に有用であると考えていたのが、(1)、(2)、(4)、(5) の機能である。これらの機能により、自然言語で記述した仕様では困難な、仕様に対する欠陥検出のための確認やテスト、デバッグ、テスト仕様の網羅性（カバレッジ）の確認を行うことができるようになる。

一方、開発の当初、その機能に期待はしていたものの、最終的には本事例への適用を見合わせたのが(6)の実行可能仕様からプログラムを自動生成する機能である。その理由は、生成されるコードが複雑でデバッグが困難であると考えたからと、生成されるコードを、プログラムの大きさやメモリの利用量の問題により、セキュア IC チップの組み込み開発環境に適用することができなかったからである。

3.5. 文書やプログラムの構成

図 A-11-2 に、本事例の題材としたプロジェクトの、全開発工程における、主な成果物を示す。図では、同社が本事例において開発した成果物を実線で、開発に当たり利用した既存の開発環境やツールを破線で表す。

図 A-11-2　主な成果物の構成

複数（n 種類）あるハードウェアの命令セットと開発環境は全て異なるが、IC チップシステムとしては、一つの外部仕様を定義し、複数種類の IC チップシステムが、同様の挙動を示すよう互換性を確保する必要がある。本事例では、外部仕様を VDM++ 言語を用いて形式的に記述、テストし、これを開発における基本的な文書と位置づけた。

形式仕様は、大きく以下から構成する。

- モバイル FeliCa IC チップファームウェアの基本的なデータ構造を表すファイルシステム仕様
- 非接触 IC カード FeliCa の無線通信と、モバイル FeliCa IC チップと携帯電話を接続する有線通信の、2 つのインタフェースとその制御の仕様
- FeliCa ファームウェアのセキュア通信プロトコルを実現する、各種コマンド仕様を記述、実行、テストするためのフレームワークフレームワーク上に記述した FeliCa コマンドの仕様
- FeliCa ファイルシステムとコマンドの仕様と一体となるセキュリティ仕様

3.6. 開発のプロセス

本事例に当たり、過去の開発プロジェクトの反省を踏まえ、また、形式仕様記述手法や各種開発、テスト手法をプロジェクトの状況に最適化して導入するために、仕様策定、設計・実装、テストを行うためのプロセスを新たに検討した。本事例における、開発プロセスの全体像と、仕様開発プロセスの概要を図 A-11-3 に示す。

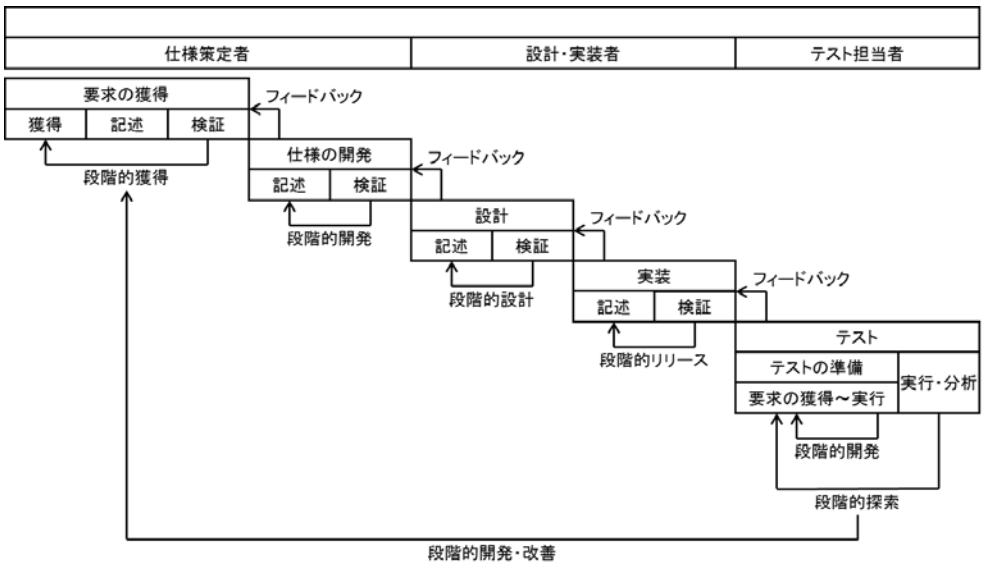

図 A-11-3　段階的開発のモデル

また、本事例では、開発を大きく仕様策定者と設計・実装者、テスト担当者に分けて遂行した。チームの構成図を図 A-11-4 に示す。ここで、設計・実装者が開発するファームウェア(F/W) は、複数あるハードウェアに組み込むために n 種類開発し、テスト担当者は、n 種類のファームウェアに加えて、形式仕様と自らが開発するテストスクリプトの確認を行うため、n + 2 種類の動作する仕様やプログラムをテストすることになる。

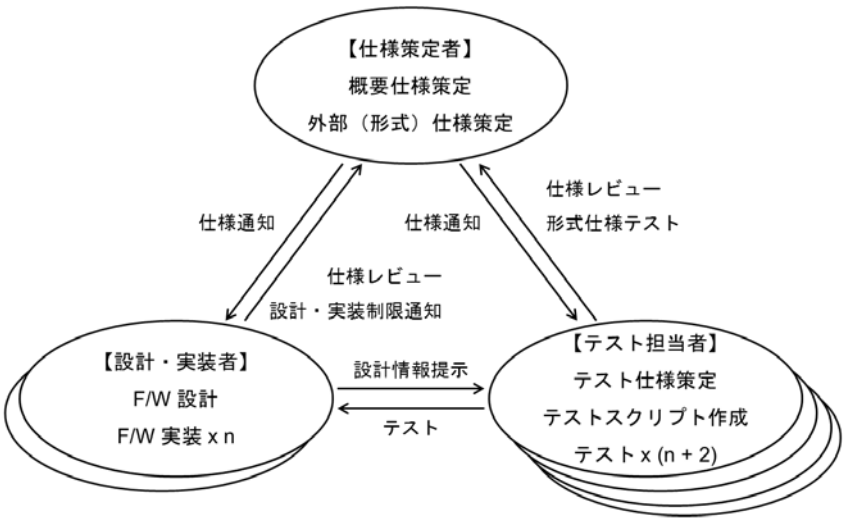

図 A-11-4　開発全体プロセスの概要とチームの構成

開発は、

（1）　仕様策定者による、要件の獲得

（2）　仕様策定者による、仕様の開発とテスト

（3）　設計・実装者による、プログラムの設計

（4）　設計・実装者による、プログラムの実装とテスト

（5）　仕様策定者や設計・実装者とは異なる、第三者であるテスト担当者によるテストの準備

（6）　テスト担当者による、仕様と、複数のプログラムのそれぞれに対するテストの実行と、テストの結果の分析

の実行を 1 イテレーションとし、これを複数回繰り返すことで、段階的に開発を行うプロセスを用いた。仕様策定、設計・実装、テストのフローを図 A-11-5 に示す。具体的には、仕様策定者は、プロジェクト外の利害関係者から聞き取りをしたり、プロジェクト内のメンバと議論しながら、要求を獲得するとともに、仕様の策定と、仕様記述言語による仕様の記述、動作する形式仕様の単体テストを行い、仕様が、「思い通り」に、正しく動作することを確認してから設計・実装者とテスト担当者に引き渡す。

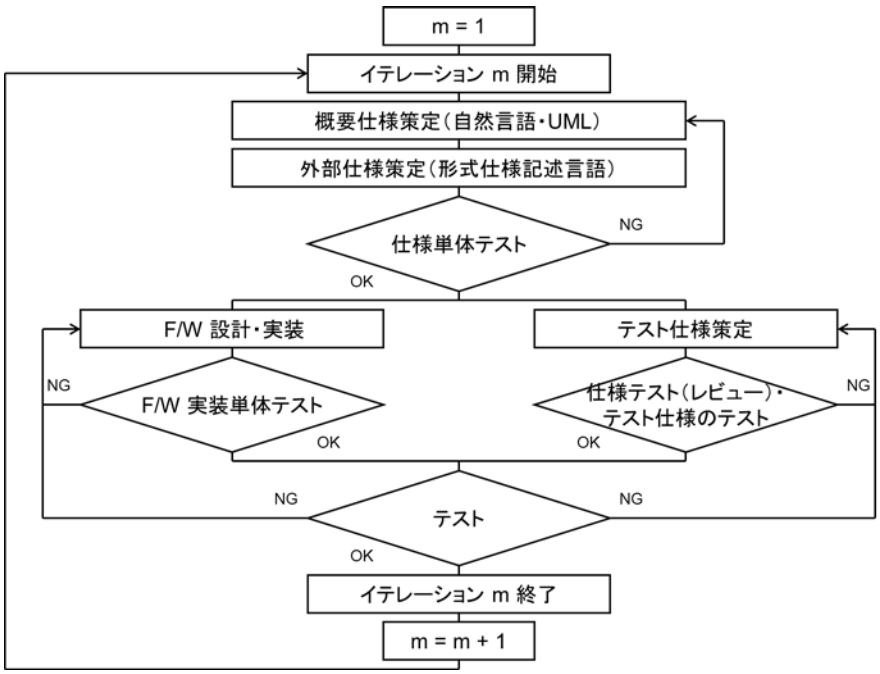

図 A-11-5　開発のイテレーション

仕様は、自然言語で記述した文書にこれを補足するための状態遷移図、シーケンス図など UML の記法に則った各種図を加えたものを概要とし、これをもとに形式仕様を記述する。このとき、性能や信頼性などに関する非機能要件に関する仕様は、形式仕様とは別に自然言語を用いて概要仕様とともに記述する。また、仕様策定と同時に、形式仕様の単体テストを

行うためのクラスを開発し、動作する仕様が単独でホワイトボックステストができるようにする。概要仕様は、開発プロジェクト外の企画者、ユーザとのコミュニケーションの結果（ニーズ）と、開発プロジェクト内からの自発的な要件（シーズ）をもとに策定する。その後、状態遷移図、シーケンス図により概要を補足しながら仕様の段階的な詳細化とレビューを行い、全体の不整合や記述の誤りに関する検討を行う。

そして、概要仕様とこれを補足する各種図を入力として、基本的なコマンド、セキュリティ仕様の策定展開を考慮しながら、FeliCa のファイルシステムをモデル化した、データ構造仕様と仕様を策定・実行するためのフレームワーク [8]、[9] の設計・実装を行い、いくつかのコマンド・セキュリティ仕様を詳細化しながら、ファイルシステムモデルとフレームワークを確定させ、形式仕様の記述を本格的に展開する。

設計・実装者は、決定した仕様をもとに設計と実装、ソースコードの静的解析及びホワイトボックステストが中心の単体テストを行い、テストの結果、実装が意図通りであることを確認してからテスト担当者に引き渡す。実装は、複数の開発環境上で共通の設計・実装と、開発環境ごとに異なるファームウェアの設計・実装に分けて行われ、これにより複数種類の IC チップが製造される。設計・実装者と同時に仕様を通知されたテスト担当者は、仕様を元にテスト仕様（項目）を列挙し、新たなテスト観点を追加したテスト仕様の策定（または更新）とテスト環境の構築（または更新）を行う。

ここで、一般的な開発プロセスであれば、構築したテスト環境を用いて実装のテストを行うが、本事例においては動作する形式仕様が存在するため、まず形式仕様のテストを行う。テストは形式仕様、開発環境上の実装（プログラム）、IC チップ内の ROM 領域に格納したプログラムに対して、同一のテストを行うことができるようにするため、テスト環境は、テスト共通フレームワークと、テスト項目と 1 対 1 に対応したテスト用プログラム、テスト環境を仕様、開発環境、IC チップそれぞれに対して接続するための複数のテストドライバから構成する。テストの結果から、形式仕様通りのテスト環境であることを確認するとともに、動作する形式仕様のカバレッジを計測して、仕様の全範囲を網羅するテスト仕様となっていることを確認する。これにより、形式仕様を入力したテスト環境の構築と、開発環境上のプログラムや IC チップのテスト環境の構築、テスト仕様の妥当性検証、テスト仕様策定段階における仕様の動作確認、レビューが行える。

そして最後に、設計・実装者が開発したプログラムのテストを行う。このテストにおいて、形式仕様と複数の実装が同じテスト環境から見て等価であることを確認することができる。

以上で説明したテストの構造の全体像を図 A-11-6 に示す。ここでも n は、ハードウェアや最終的な開発の対象である IC チップの種類を示す。

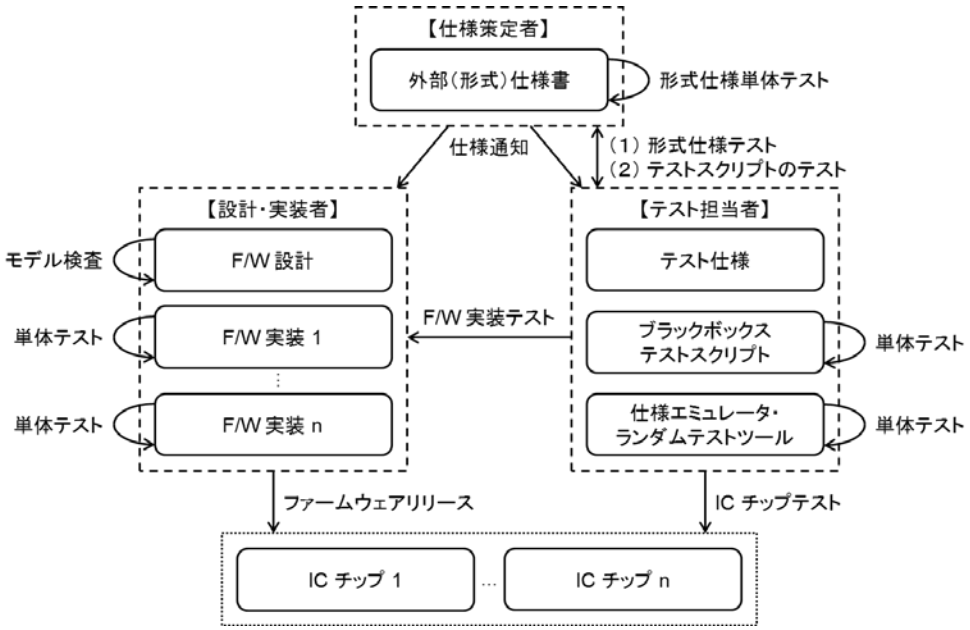

図 A-11-6　テストの構造

本事例においては、上記を 1 回のイテレーションとして、1～ 2 週間のサイクルで開発を行った。

ここで、本開発プロセス(図 A-11-3 参照)を見直すと、以下の形式手法導入の利点が明らかをなった。

- 要求獲得工程において整理した要件に基づいて、形式的仕様の記述と検証を行うと同時に、厳密な仕様の記述により、仕様の明確化や要件の実現可能性の検討及び開発ドメインに対する理解が進み、要求獲得工程における開発の具体化や検討精度の向上も可能となる。これにより、要件と仕様の双方の開発を、両者の相補的な関係を保ちながら、段階的に進めていくことができる。
- 厳密な形式仕様に基づいて、「何を」開発しようとしているのか、「どう」開発しようとしているのかを厳密に切り分けながら、そして、プログラムを実装するための都合や制約条件と向き合いながら、設計に集中することができる。同時に、仕様を実現するための設計が困難である場合は、明確になりつつある仕様の具体的などの部分をどのように変更すればシステムを実現するための設計が可能になるのかを明確にすることができる。これにより、仕様と設計の双方の記述や検証を連動しながら、しかし双方の記述を完全に分離しながら、段階的に開発を進めていくことができる。
- 一方、厳密な仕様が存在しないプロジェクトにおいては、曖昧な仕様に基づいて設計を行うため、曖昧な設計しか行うことができない、仕様と設計の境界が曖昧になり「何を」作ろうとしているのかと「どのように」作ろうとしているのかが区別できない、実現が困難な設計のフィードバック先や仕様の修正方法が不明確になる等の課題が発生する。

- 厳密な形式仕様と仕様に基づいた設計に従ってプログラムを実装する際には、「何が」仕様として必ず満たさなければならない条件であるのかと、「どのように」実装するのかを表した設計の都合の双方を完全に把握しながら、開発を進めていくことができる。同時に、プログラムの実装において実現が困難な、あるいは改善が可能な仕様または設計を明確にして、仕様開発工程や設計工程にフィードバックをかけることができる。
- 一方、厳密な仕様が存在しないプロジェクトにおいては、曖昧な仕様と設計に基づいてプログラムの実装を行うため、「何を」「どのように」作ろうとしているのかを自信を持って実装することができない、実現が困難な仕様や設計のフィードバック先や修正方法が不明確になる等の課題が発生する。
- 厳密な形式仕様を拠り所として、仕様に基づいて設計・実装されたプログラムをテストする際には、必ず達成すべき仕様と、プログラムの実行環境の事情や制約を考慮しながら設計・実装したプログラムとを完全に切り分けて確認を行うことができる。このため、要求や仕様を満たしたプログラムであることを、仕様策定者や設計・実装者ではない第三者がテストにより確認することができる。
- 一方、厳密な仕様が存在しないプロジェクトにおいては、曖昧な仕様に基づいてプログラムのテストを行うため、「何が」テストすべき確認項目なのかが明確にならず、テスト担当者は自信を持って検証を行うことができない。また、テスト後のデバッグにおいても、「何を」基準として「どのように」修正すれば良いのかが不明確で修正の都度仕様や設計を検討する必要性が生じる可能性がある。

更に、仕様開発プロセスの概要には記していないが、システムを開発しこれをリリースした後のサポート・運用・派生開発においては、以下を本開発プロセスの導入効果として挙げることができる。

- 密な仕様の記述により、何を開発したのかを明確にして的確なサポートと運用を行うことができる。
- 一方、厳密な仕様が存在しないプロジェクトにおいては、何を開発することができたのかが不明瞭であるため、サポートと運用を自信を持って行うことができない。
- 外部からの問い合わせに対して、問い合わせの内容が仕様に関わることなのか、設計・実装に関わることなのかの完全な分離を行うことができる。
- 一方、厳密な仕様が存在しないプロジェクトにおいては、例えば問い合わせの内容が、仕様どおりであるのか、仕様どおりではない考慮不足や欠陥であるのか、を明確にすることができない可能性がある。
- 欠陥の修正や派生開発において、仕様変更と仕様拡張と設計変更と設計拡張を完全に分離して議論することができる。

・　　一方、厳密な仕様が存在しないプロジェクトにおいては、そもそも「何を」作ったのかが曖昧な上、仕様と設計の境界が曖昧であるため、欠陥の修正や派生開発による影響を特定することができず、開発対象の位置づけが曖昧になる可能性がある。

開発や運用・サポート・派生開発にまつわる全ての活動の起点を仕様とし、そして「何を」作るのかを記述・維持していく原典を形式仕様として厳密に記述検証することで、形式仕様記述のみならず開発全体の活動の精度を高めながら、段階的に開発を洗練させていくことができる。

4. 形式仕様記述手法導入の直接的な効果

4.1. 形式仕様記述手法の適用結果の概要と考察

本事例における開発プロジェクトは当初からの予定どおりに進捗した。

形式仕様記述手法は開発の現場に浸透し、コミュニケーションツールとして有効活用できた。そして、仕様の各要素の記述と実装、テストの正確性、仕様や最終成果物であるプログラムの品質が、それぞれに対する要件の優先順位や複雑度・難易度によらず一定となった。従来の開発においては、相対的に重要ではない機能の仕様の記述がおざなりになったり、利用頻度の低い機能の品質が低かったりした。

仕様の策定においては､具体的には､形式仕様自身のテストである形式仕様単体テストと、形式仕様を補足するための自然言語によるコメントを含め、約 10 万ステップの形式仕様を策定し、これを活用した設計・実装、テストを行った。仕様に関連する主な成果物は以下のとおりである。

・　自然言語による 383 ページのプロトコルマニュアル（社内他部署向け・社外顧客向けマニュアル）
・　形式仕様記述言語による 677 ページの外部仕様書

仕様は、86 のコマンド・レスポンス仕様と、ファイルシステム・セキュリティ仕様から構成している。また、この仕様をもとに、一種類の IC チップ上のファームウェアとして、コメントを含め約 11 万ステップの C/C++ 言語のコードを実装した｡仕様策定時にデバッグした形式仕様の欠陥の件数は表 A-11-1 のとおりとなった。

表 A-11-1　仕様策定時にデバッグした欠陥の件数

欠陥が摘出できた仕様策定における工程	欠陥の件数
仕様の記述	162
仕様の実行と単体テスト	116
形式仕様のレビュー	93
設計・実装者やテスト担当者とのやりとり	69
合計	440

そして、形式仕様を活用して実装・テストを行った結果、多くの「仕様不明確」に苦しめられることなく、手法導入の目的のひとつである仕様の明確化の効果を確認することができた。プロジェクト全体の欠陥原因の中で、仕様関連の割合は表 A-11-2 のとおりである。

表 A-11-2　ファームウェアの実装における欠陥原因の割合

欠陥の原因	割合
仕様記述もれ	0.2%
仕様記述誤り	0%
仕様不明確	1.8%
仕様見落とし	5.6%
仕様理解不足	10.7%
仕様確認不足	0%
仕様変更通知不徹底	0.2%
その他仕様関連外	81.5%

形式仕様記述手法は、開発の初期段階における誤りの発見に有効である。構文や型の検査を行うことができる上、動作する仕様を策定すれば、仕様の単体テスト、テスト環境との接続テストを行うことができるようになる。そして、ケアレスミスを排除した上で、仕様の品質向上と後工程であるテストの品質向上を目指すことができるようになる。

仕様策定段階においては、自然言語よりも簡単に、統一したフォーマット、フォームで仕様を書くことができ、設計・実装とテストの段階においては、形式仕様をコミュニケーションの道具として、仕様の本質について、仕様とは何か、コミュニケーションとは何か、ということについて議論することができるようになる。表 A-11-2 からもわかるように、仕様は仕様策定者の意図通りに考え、書けたと言える。一方で「仕様見落とし」と「仕様理解不足」が合計 16.3% あり、課題となった。これは、動作するプログラムとしての形式仕様のコーディングに気を取られ、「読ませる仕様」の記述に十分取り組むことができていなかったためである。具体的には、形式仕様記述手法導入に関する位置づけの説明、メンバへの導入ガイド及びフォロー、関連ドキュメントの充実などが不足していた。仕様が書けるようになった開発者の次の目標である。

4.2. 仕様の策定に関する結果

以下、仕様の策定に関する結果を述べる。

形式仕様の全体量と総工数から計算した、外部仕様策定のための形式仕様記述言語によるモバイル FeliCa 仕様記述フレームワーク完成以降の、1 人の仕様策定者が 1 ヶ月（約 160 時間）に記述した仕様の行数は平均約 1,900 行であった。一般に、専任の担当者であっても開発（仕様策定）業務は記述だけではないので、一概に論じることはできないが、この数値は、本事例における設計・実装やテストの工程での開発効率と同等である。形式仕様記述言語を用いた仕様の記述に特段の障壁はないと言える。

仕様記述フレームワーク完成後、本格的に形式仕様を策定し、外部仕様書バージョン 1.00 が完成するまでの 8 ヶ月間の開発効率を表 A-11-3 に示す。

表 A-11-3　仕様開発の生産性

経過	開発効率
1 ヶ月目	2,600 行/人月
2 ヶ月目	3,200 行/人月
3 ヶ月目	1,400 行/人月
4 ヶ月目	850 行/人月
5 ヶ月目	1,800 行/人月
6 ヶ月目	2,000 行/人月
7 ヶ月目	2,500 行/人月
8 ヶ月目	2,600 行/人月

各期間における仕様策定のトピックスは以下のとおりである。

1 ヶ月目: FeliCa 仕様記述フレームワークが完成し、外部仕様の策定を開始した。仕様策定環境に慣れない新規メンバのサポートも必要であったが、順調にスタートした。

2 ヶ月目: 更にメンバの追加も行い、またフレームワークや基本的な仕様であるファイルシステムの修正も行ったが、仕様策定は順調に進捗した。

3・4 ヶ月目: 難しい仕様の策定にも取り組んだ。このため、コーディングの効率だけを見ると生産性が落ちている。

5 ヶ月目: コメント抽出フィルターの作成、UML によるシーケンス図の作成なども行った。

6 ヶ月目: 自動テスト実行環境の構築、テスト実行時間の見積もりなども行った。

7 ヶ月目: 導入のための資料作成なども行った。

8 ヶ月目: バージョン 1.00 をリリースした。

形式仕様単体テストのラインカバレッジ率は 82% である。仕様策定工程に限らず、単体テストの終了条件をどのレベルに設定するのかは難しい問題であるが、本事例においては、仕様策定者が自ら実装した形式仕様について「一通りテストする」レベルとした。後工程からのフィードバックの分析については後述するが、この結果、仕様策定者の単体テストは後

工程に数多くの欠陥を先送りしてはいなかった。特に、不変条件・事前条件・事後条件においてロジックの分岐があるものは、カバレッジ分析からテストケースの網羅率を高めることができた。テストの結果、例えば事後条件の不正なパスを発見することができており、品質確保につながっている。一般に仕様のこのような不整合をレビューで発見することは困難である。動作する形式仕様を策定した効果であると言える。

一方で、開発環境として VDMTools を用いる場合、言語処理系がインタプリタであることを理解した上で、コードの書き方とカバレッジの測定結果を分析しなければならない点に注意する必要がある。ラインカバレッジ率はインタプリタがトレースしたテキストの行数の割合であり、実行した文の割合ではないからである。外部仕様書バージョン 1.00 が完成後、仕様の追加や変更を 182 回行った。これには後述する設計・実装者、テスト担当者、ユーザとのコミュニケーションによって顕在化した、あるいはプロジェクト外からもたらされた仕様変更依頼や欠陥候補の指摘、仕様変更提案が含まれている。その内訳は表 A-11-4 のとおりである。

表 A-11-4　仕様追加と変更の分類と数

変更元	変更数
仕様策定者自身による変更依頼	84 件 (46%)
設計・実装者からの変更依頼・指摘・提案	25 件 (14%)
テスト担当者からの変更依頼・指摘・提案	23 件 (13%)
その他	9 件 (4%)
プロジェクト外からの変更依頼・提案	41 件 (23%)
合計	182 件

ここで、「設計・実装者からの変更依頼・指摘・提案」には、設計・実装者が外部仕様を満たすことができないことが判明したことによる仕様変更依頼、「プロジェクト外からの変更依頼・提案」には、ハードウェアの制約や欠陥により開発が困難になった仕様変更依頼が含まれる。また、「その他」には、セキュリティ上問題のあることが判明して変更した仕様が含まれる。

4.3. 仕様の策定に関する考察

一般に、要件や仕様の変更をどの工程で行うべきか、あるいはどの工程・段階まで許容できるのかが問題となるが、本事例においては、仕様策定や形式仕様の単体テストの工程では仕様策定における品質確保を完結できずに、仕様変更が次工程以降に先送りされてしまったケースもあった。

仕様の策定においては、設計・実装者やテスト担当者、プロジェクト外メンバのレビューが不可欠であることは間違いがない。当然ながら、プロジェクトの開始当初からメンバや利

害関係者の総力を結集して仕様策定やテストに当たれば早期から仕様確定の確度は高くなる。

一方で、変更数が多いのは外部仕様を詳細に記述したからであるという捉え方もできる。外部仕様が詳細に記述されていれば、仕様について詳細に議論し、変更することができる。また、従来のプロジェクトにおいては、十分な検証が行われていない部分もあったが、本事例においては、形式仕様と一部のテスト仕様も検証可能となり、仕様もメンテナンス対象となったばかりではなく、仕様に関するコミュニケーションのフィードバック先が明確になったとも言える。

また、仕様変更によるプロジェクト全体の手戻り工数を算出することは難しい。本事例では、1～2 週間程度の短い期間で仕様策定・実装・テストのイテレーションを繰り返したが、この中で、欠陥対応と仕様変更に費やした時間の計測が困難であった。これは、開発者の全ての作業を記録しておき、開発後に最終的に確定する欠陥と、仕様変更に対応した時間を集計することができなかったためである。

更に、形式仕様記述手法は仕様を「決める」助けにはならない。多くの根本的・本質的な誤りは要求を仕様に落とし込む過程で生じる。このことは、特にプロジェクトの初期フェーズにおいて、プロジェクトオーナーやメンバと合意しておいた方がよい。何故なら、形式仕様記述手法を導入すると、仕様策定期間が従来よりも長くなる（可能性がある）が、厳密な仕様の策定に時間を掛けて取り組んでいることと、要件が定まっていないことを混同してしまうと、形式仕様記述手法の導入がプロジェクト全体のボトルネックになっているのではないかといった、誤った分析がなされてしまう可能性があるからである。

また、仕様の記述やテストの効率に関しては、様々な目的や用途に応じた実装用の言語があるため一概に論じることはできないが、形式仕様記述では、文字数や行数当たりの情報量を多くする、すなわち記述密度を高くすることもできるため、開発効率を高めることもできる。しかし、記述密度の高い仕様書が、プロジェクトメンバ全員にとって分かりやすい仕様であるとは限らない。メンバの平均的な熟練度に合わせて、適切なレベルの記法やテクニックを用いた仕様を記述すべきであろう。

4.4. テストに関する結果と考察

本節では、動作する外部仕様である形式仕様を、開発環境上のプログラムや IC チップのテストも行うテスト環境を用いてテストを行った結果とその考察を述べる。テスト環境は、複数種類の IC チップファームウェア開発環境上のプログラムや複数種類の IC チップ、形式仕様に接続できるようになっており、IC チップファームウェアと形式仕様が同様の動作をすることが確認できる。テストはファームウェアや仕様のプロジェクト内部リリースに合わせて。1～2 週間程度に一度行う。テストの結果は、仕様策定・実装・テストのそれぞれの開発イテレーションにフィードバックされる。これにより、仕様と実装の一致を確認し続けることができる。テストの観点からは、策定したテスト仕様とこれを具体化したテスト環境、テスト用プログラムを接続することで、テスト仕様の正確性と仕様に対するテストの網羅率

を確認することができる。更にこれに加えて、後述する「ランダムテスト」とそのための仕様エミュレータも開発した。

最終的なテスト環境を用いた形式仕様テストのラインカバレッジ率は 100% である。形式仕様の記述上、ロジック上実行することができないコードが、あるいは実行することが難しいコードが存在する。テストを行わなくても問題がないかどうかは、テストの度に目視で確認している。この目視による検査を加えての 100% である。実行されていない行は、ツールのドキュメント清書機能が赤く強調して示してくれる。

形式仕様のテストの結果、仕様の欠陥を 6 件検出することができた。本来であれば、テスト仕様の策定及びテスト環境の構築プロセスの入力が仕様であり、仕様どおりに実装がテストされる工程において、仕様の欠陥が検出されることは道理に合わないが、仕様策定段階におけるレビューや単体テストでは検出できなかった欠陥をテスト担当者が発見している。この 6 件の欠陥が、仕様策定工程が次工程に先送りした仕様の欠陥であるということができる。この欠陥も、上記の単体テスト同様、仕様策定段階でのレビューで見つけることが難しい類の欠陥であった。こちらも動く形式仕様を策定した効果であり、構文や型のチェックに加えて、動作する仕様をテストすることで、より精密に検証できる効果は大きい。更に、テスト項目の明らかな欠落をテストすることもできる。テスト環境を用いた形式仕様テストのラインカバレッジ率が 100% ではない場合は、テスト項目が不足している可能性がある。一方で、ソフトウェア開発プロジェクトにおける総合的なテストは、外部仕様のみを入力とするものではないため、またテスト環境を用いた形式仕様テストのラインカバレッジ率が 100% であっても、組み合わせテストの欠落は検出できないため、形式仕様テストのラインカバレッジのみをクライテリアとしてテストを行ってはならない点に注意が必要である。

テストの工程においては、テスト用プログラムやテスト環境の間違いを指摘できる。特に、テスト仕様（テスト項目）は正しく記述されているが、テスト用プログラムが誤っている場合や、テスト仕様とテスト用プログラムの双方が誤っている場合の検出を簡便に行うことができる。一方で、テスト仕様書が間違っているがテスト用プログラムは正しいケースは検出できない。これは、概要仕様・マニュアルが間違っているが形式仕様は正しいケース、仕様・設計・実装・テストの全てが間違ってしまうレアケースとあわせて、今後の検討課題である。

また、実装のテストのひとつの観点として、「ランダムテスト」を行った。ランダムテストとは、仕様を完全にエミュレートする「仕様エミュレータ」を内蔵する「ランダムテストツール」が実装に対してランダムにコマンドを送信し、実装から返却される戻り値が仕様通りであることを確認する耐久試験である。ランダムテストツールや仕様エミュレータの開発には、テスト項目ひとつひとつからテスト用プログラムを作成するテストよりも、更に詳細な仕様に基づく期待値が必要となる。不変条件・事前条件・事後条件を明確に定義した形式仕様があるからこそ可能となるテストである。

そして、これは特に組み込みの開発とテストの全般に言えることであるが、実機と開発環境、形式仕様の差異によって、テスト結果が異なるテスト項目の取り扱いに注意が必要であ

る。こちらも今後の検討課題である。

5. 形式仕様記述手法導入の間接的な効果仕様書にまつわるコミュニケーションに関する結果

本節では、形式仕様をもとにしたコミュニケーションの代表例として、仕様に関する質問についてまとめた結果を示す。仕様に関する質問と回答は、仕様策定者、設計・実装者、テスト担当者、その他ユーザ（社内他部署・社外顧客）の質問と質問に対する回答を管理できる Web アプリケーション上のツールを通して行った。ここで、仕様策定者、設計・実装者、テスト担当者は、形式仕様記述言語による外部仕様書と自然言語による概要仕様書の双方を参照している。一方、「その他ユーザ」は、主に自然言語による概要仕様書のみを参照している。

仕様に関する質問は大きく「理解」、「意図」、「誤り」の各カテゴリに分類した。詳細は以下のとおりである。

(1-1) 理解・記載済み: 仕様書に記載されているが理解できなかった仕様、導入教育が不十分であった仕様、参照ドキュメントが明示されていれば理解できた仕様に関する質問

(1-2) 理解・背景: 仕様策定の背景が書いていないまたは分かりづらいが、知っておきたい仕様、背景を理解した上で実装、テストを行いたい仕様に関する質問

(1-3) 理解・記載なし: 仕様が既に確定しているが、仕様書に記載が無いために理解できなかった仕様、図による補足説明、正確なインデックス（目次、索引、文書の関連を表す情報）等があれば、正しく理解できたであろう仕様に関する質問

(2-1) 意図・実装確認依頼: 詳細設計、テスト方式設計が仕様どおりであることを確認すべき事項、仕様策定、実装、テスト全体に関係するアーキテクチャに関する質問

(2-2) 意図・実装要望: 開発環境や詳細設計の制約、課題を形式仕様に反映すべき仕様に関する質問

(2-3) 意図・精査依頼: 仕様に間違いはないが、記述に統一性がないと考えられる仕様、仕様の細目間に不整合があると考えられる仕様に関する質問

(2-4) 意図・明確化依頼: 不明確、未確定な仕様、仕様は明確だが記述が不明確なので明文化すべき仕様、用語が統一されていないため不明確な仕様に関する質問

(3-1) 誤り・差異（テスト）: 形式仕様と、過去の開発成果物も含めたテスト仕様やテスト環境に差異がある仕様に関する質問

(3-2) 誤り・差異（仕様）: 形式仕様と、自然言語による概要仕様、マニュアルに差異のある仕様、自然言語による概要仕様、マニュアルの記述に誤解が生じる可能性がある仕様に関する質問

(3-3) 誤り・純然たる誤りの指摘: ここで、「誤り・差異（実装）」、すなわち実装と仕様の

差異を分類する項目があってもよいが、これは「意図」に分類した。同様に、用語変更案、要件に関する質問も「意図」に分類している。また、質問を受ける側である仕様策定者がカテゴリの分類を行った。

仕様に関する質問や指摘、要望などの件数及び上記カテゴリの件数と割合を集計したのが表 A-11-5 である。

表 A-11-5　仕様書群に対する質問の件数と割合

	(a)	(b)	(c)	(d)	(e)	(f)
(1-1)	3 (4%)	19 (23%)	8 (17%)	11 (9%)	27 (21%)	0 (0%)
(1-2)	2 (3%)	6 (7%)	9 (197%)	11 (9%)	15 (12%)	0 (0%)
(1-3)	3 (4%)	5 (6%)	1 (2%)	4 (3%)	6 (5%)	0 (0%)
(2-1)	2 (3%)	5 (6%)	5 (10%)	7 (6%)	10 (8%)	9 (36%)
(2-2)	6 (8%)	7 (9%)	3 (6%)	9 (7%)	10 (8%)	0 (0%)
(2-3)	5 (7%)	7 (9%)	9 (19%)	14 (12%)	16 (12%)	15 (60%)
(2-4)	28 (38%)	3 (4%)	0 (0%)	28 (23%)	3 (2%)	1 (4%)
(3-1)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
(3-2)	0 (0%)	0 (0%)	11 (23%)	11 (9%)	11 (8%)	0 (0%)
(3-3)	24 (33%)	30 (37%)	2 (4%)	26 (21%)	32 (25%)	0 (0%)
合計	73 (100%)	82 (100%)	48 (100%)	121 (100%)	130 (100%)	25 (100%)

コメントの対象は、

(a) 自然言語で書かれた概要仕様、マニュアルに対する質問

(b) 形式仕様のみに対する質問

(c) 概要仕様、マニュアルと形式仕様の双方に対する質問

(d) 概要仕様、マニュアルに関する質問の合計

(e) 形式仕様に関する質問の合計

(f) 仕様というよりは更に上流の要件に関する質問

に分類した。

更に、上記カテゴリに分類できないコメントが 31 件あった。

5.2. 仕様書にまつわるコミュニケーションに関する考察

表 A-11-5 に示した結果から、自然言語による概要仕様・マニュアルと、形式仕様記述言語による仕様のそれぞれについて、理解と明確化依頼を除く内容に関する質問の割合は似通

っていると言える。自然言語による概要仕様・マニュアルは、形式仕様記述言語による仕様と比較して明確化依頼が多い。一方で、形式仕様記述言語による仕様は、自然言語による概要仕様・マニュアルと比較して、理解の質問の割合が多いと言える。自然言語による記述は意味が一意に定まらず曖昧になることがあり、厳密性に欠ける。形式仕様は仕様を理解するためのコメントや導入教育が必要であるが、これが不足していたと言える。形式仕様記述言語では仕様の背景を説明することは難しい。形式仕様のコメントに仕様の背景や策定経緯、代替案などを自然言語を用いて記述することが望ましい。

一方、自然言語の概要仕様、マニュアルに対して、仕様理解をするための質問の割合が少ない。これは、現在の概要仕様、マニュアルに仕様策定の意図が多く記述されていないため、読者が「わかったつもり」になってしまうからではないかと推察される。その後の欠陥発生の原因となったコミュニケーション不足もあった。対して、形式仕様は中途半端な理解ができないので、仕様の理解に関する質問が多いとも言える。更に、仕様の背景までも理解しようとする開発者は、物事を突き詰めて考えようとするため、形式記述仕様まで読み込んでから質問をするとも考えられる。

「(3-2) 誤り・差異（仕様)」に関する質問は、仕様策定者間のレビューやコミュニケーションの不足と捉えることができるかもしれないが、冗長性を持たせることで発見することができた誤りであると捉えることもできる。また、形式仕様から自然言語による概要仕様、マニュアルを作成する方法とテストに関する課題もある。更に、実装確認依頼、実装要望、精査依頼などの質問があることから、実装工程に入らなければ課題を顕在化できなかった仕様があるとも言える。これは、別の捉え方をすれば、上流工程での検討や、開発環境、実装方式に関する検討や配慮が足りなかったとも言える。更に、実装確認依頼及び実装要望に関してどこまで概要仕様やマニュアル、形式仕様に反映させるのか、外部仕様に関わる設計情報をどの仕様書に反映させるのかといった課題もある。

5.3. プロジェクトへの形式仕様記述手法導入に関する考察

本事例における適用のレベル（仕様を記述してテストするレベル 0）であれば、形式仕様記述手法導入や仕様を策定するためのフレームワークの検討には専門家の助言が必要であるが、形式仕様記述には、例えば高度な数学的能力などは必要としない。形式仕様記述者に求められる能力は、通常のプログラマに求められる抽象化能力を上回るものではない。特にVDM++ 言語であれば、C++ 言語、C# 言語、Java 言語などとオブジェクト指向技術の知識、開発経験があれば導入できる。

もちろん、形式仕様記述手法だけを礼賛しているわけではなく、可能であれば自然言語で仕様を記述した方がよいと考えているが、現時点での一般的なソフトウェア開発プロジェクトにおいては、自然言語を用いて仕様を記述することができるメンバを確保、教育するよりは、プログラマを確保、教育する方が容易である。プログラマの教育プログラムは存在するが、自然言語による仕様策定教育プログラムは少ない。CMM や UML などのプロセスや記

法に関する教育プログラムはあるが、実装に必要な全ての仕様を記述することは困難な上、仕様をテストすることもできない。プロジェクトメンバへインタビューしても、皆異口同音に、先輩からプログラムの書き方は教えてもらっているが、仕様の書き方は教えてもらうどころか、先輩達も悩んでいるようだと言う。

自然言語を用いて仕様を記述するのだとしても、集合や論理を意識しながらロジックを組み立てていくことができるのであれば､形式仕様の導入と同じ効果が得られる可能性がある。そういったことができるエンジニアにとっては、いわゆる形式仕様記述は必要ないのかもしれない。このことは、形式仕様記述言語による仕様記述に熟達すると、自然言語による仕様記述のスキルが向上すると言い換えることもできる。

また、仕様がプログラムに近いことによる効果も大きい。プログラム開発において蓄積された技術が応用できるからである。バイナリファイル形式の仕様書では困難な変更履歴を把握できるようにするための構成管理、各種フィルタプログラムやバッチプログラム、開発支援ツールの活用、VDM++ 言語であればオブジェクト指向分析設計技術の活用、ユニットテストなどである。更に、仕様策定者、設計・実装者に与える心理的な効果もあるだろう。精確に調査はできていないが、ソフトウェア開発エンジニアは、自然言語で書かれた仕様書をメンテナンスせずに放置しておくことに躊躇はないが、動かない、あるいは欠陥のあるプログラムをそのままにしておくことには心理的な抵抗があるのではないだろうか。また、ソフトウェア開発エンジニアは、自然言語の文章を書くよりもプログラムコードを書く方が好きなケースも多い。これは、プログラムに限らず、プログラムに近い形式仕様にも言えることで、プロジェクトの別なメンバに指摘されて気がつくこともある。

そして、これも形式仕様記述に限った話ではないが、仕様を書くのと読むのとは全く異なる。本事例においては、初めての形式仕様記述手法導入ということもあり、動く仕様を策定する、記述する、コーディングすることに注力してしまい、設計・実装者やテスト担当者、ユーザに「読ませる」仕様になっていなかった。皆に読まれる仕様とする以上、実装用のプログラムよりも、規約に則った読みやすい、読者が誤読することのない、サービス精神旺盛な、かゆいところに手が届く、しかしテクニックに堕しない仕様を書くことを志向していくことが必要になる。

5.4. プロジェクトへの形式仕様記述手法導入の障壁

言語仕様の理解という観点においては、ツールベンダが開催する講習会への参加の効果もあり、導入への障害はなかった。また、途中から参加したメンバは講習会へ参加していないが、社内研修や自習によりスムーズに導入できているため、特に大きな問題にはなっていない。

講習会へ参加していない、社会人 1 年目のメンバであっても、オブジェクト指向などの基礎的なソフトウェア工学について理解していれば、短期間で言語仕様と仕様記述方法を理解して仕様を記述することができるようになった。VDM++ 言語の場合、C++ 言語、C# 言語、

Java 言語の仕様やこれを利用した実装に慣れ親しんでいれば、半月から 1ヶ月程度のトレーニングにより容易に仕様をコーディングできるようになる。ただし、新規メンバの導入には、手本となるようなテンプレートやフレームワーク、チーム内での積極的なコミュニケーションが不可欠である。

仕様を読む側の設計・実装者、テスト担当者も事情は同じである。習得が早い実装者の中には、1 日で仕様を読むことができるようになったメンバもいた。一方で、例えば C 言語の知識と実装経験しかない組み込み系開発者が苦労したケースもあった。習得に最も時間を必要とした例では、3 ヶ月経ってもチームメンバの助力なくしては仕様を読むことができなかった。

一方で、現在普及している一般的なプログラミング言語よりも導入の障壁は高いと言えるだろう。言語仕様、ライブラリ、テンプレート、ツール、ノウハウの集約や整理が必要である。

また、本事例の導入段階においては、仕様策定統合ツールの使い勝手とその性能、品質、パフォーマンスが大きな導入障壁になっていたが、ツールベンダの手厚いサポートにより、プロジェクトの進捗を妨げる大きな問題は回避できた。更に、トップマネジメントや他部署、社内他プロジェクト、顧客の理解も必須である。一般にプロジェクト外の利害関係者が形式仕様を熟読することはないだろうが、本事例の例では、プロジェクト全体のソフトウェア開発に対する「取り組み姿勢」が肯定的に評価され、プロジェクトを推進していく上で大きな効果を発揮した。

6. 結論

形式仕様記述手法は、開発の上流工程における誤りの発見に有効である。仕様は、仕様記述言語やツールを活用することにより、構文や型を検査しながら、厳密に、統一したスタイル、フォームで記述することができる。更に、動作する陽仕様の記述により、プログラムと同様に、デバッグのためのテストや、変更や修正による品質低下がないことを確認するための回帰テストを行うことができる。

そして、厳密な仕様は、開発対象であるプログラムの徹底的なテストを可能とする。テストケースの列挙や、テスト用プログラムの開発においては、動作する仕様を記述することにより、テスト用プログラムから動作する仕様を実行することが可能となり、これにより、仕様の再確認と、仕様に対するテストケースの網羅性確認、テストケースに基づいて作成したテスト用プログラムが仕様通りなのかどうかの確認を行うことができる。

また、本事例における形式仕様記述やテストの効率は、従来のプログラム開発の効率と変わらなかった。そして、プロジェクトマネジメントを含む、開発工程全体のコストも、一般的な開発コストの見積もり手法の算出結果よりも小さかった。一方で、従来手法と比較して、上流工程で厳密に仕様を記述、検証するのに従って、開発全体の中で、仕様策定工程が占める割合が大きくなる。そして、従来、下流の工程で問題が顕在化していた課題を、上流工程

における厳密な仕様の記述と検証とともに解決しなければならなくなるため、仕様の確定や変更に関する議論や管理が重要となる。

形式的に仕様を記述することにより、記述した仕様の属人性をなくし、統一したフォーマットの仕様書群を、チームで記述、検証することができるようになる。この結果、仕様に関する議論を、「問題対私たち」といった形式で、チーム全体で行うことができるようになる。そして、形式的な記述とテストをした仕様を開発の拠り所にすることで、非形式的な文書の品質も向上する。これは、仕様の記述とテストにより、記述や検討、考察の不足が開発の初期段階で明らかとなり、問題を解決するために議論を重ねる中で、様々な抽象度で問題に対する理解が深まるとともに、開発の目的や条件、制約を明確にしながら、構造や関係性の見通しが良く、分かり易い文書構造の検討や、記述を行うことができるようになるためである。また、仕様に関するコミュニケーションのフィードバック先や、修正の方法、回帰テストの方法が明確になることから、仕様の維持を継続する動機が生まれるためである。更に、仕様が明確になってくると、プロジェクト内外の利害関係者が、論点を明確にして互いが暗黙知として認識している「つもり」のすり合わせをしながら、そしてこれを形式仕様として記述しながら、議論や、認識、理解、経験の共有を進めることができるようになる。仕様の明確化と、記述の改善、コミュニケーションの活性化が、開発の上流工程における成果物の品質確保のみならず、開発工程全般において効率良く品質を確保することに寄与するのである。一方で、仕様は形式的な記述ばかりではなく、仕様策定の背景を含めた仕様に対する理解を促す記述を書き添えることが重要である。また、設計・実装者の仕様に対する合意形成、読みやすい仕様の記述、仕様と設計との適切な境界の設定、設計と実装に仕様策定者が意図しない影響を及ぼさない仕様の記述範囲の検討と記述の方法を検討していくことが重要である。

また、形式手法はソフトウェア開発において、単独で用いる手法ではなく、他の手法や工学の成果、工夫を組み合わせて応用していくものであり、形式仕様記述手法は、仕様策定の工程に限らず、開発の全工程に対して効果がある手法であるため、開発の構造全体について議論していくことが重要である。

参考文献

[1] 荒木啓二郎. フォーマルメソッドの過去・現在・未来 －適用の実践に向けて－. 情報処理, Vol. 49, No. 5, pp. 493-498, 2008.

[2] 佐原伸, 荒木啓二郎. オブジェクト指向形式仕様記述言語 VDM++ 支援ツール VDMTools. コンピュータソフトウェア, Vol. 24, No. 2, pp. 14-20, 2007.

[3] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods. IEEE Computer, Vol. 28, No. 4, pp. 56-63, 1995.

[4] 荒木啓二郎, 張漢明. プログラム仕様記述論. オーム社, 2002.

[5] John Fitzgerald and Peter G. Larsen. Modelling Systems: Practical Tools and Techniques in Software Development. Cambridge University Press, Second edition, 2009.

[6] John Fitzgerald, Peter G. Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. Validated Designs for Object-oriented Systems. Springer, 2005.

[7] 中島震. ソフトウェア工学の道具としての形式手法 －彷徨える形式手法－. ソフトウェアエンジニアリング最前線 2007, pp. 27-48, 2007.

[8] 中津川泰正, 栗田太郎, 荒木啓二郎. FeliCa IC チップ開発における仕様記述フレームワークの構築. ソフトウェア工学の基礎 XVI, pp. 13-24. 日本ソフトウェア科学会 FOSE, 近代科学社, 2009.

[9] 中津川泰正, 栗田太郎, 荒木啓二郎. 実行可能性と可読性を考慮した形式仕様記述スタイル. コンピュータソフトウェア, Vol. 27, No. 2, pp. 130-135, 2010.

[10] 中津川泰正, 栗田太郎, 米田篤生, 谷川正和, 守屋繁. 携帯電話組み込み用 "モバイル FeliCa IC チップ" 開発におけるモデル検証手法の導入と課題. 組込みシステムシンポジウム 2006 論文集, pp. 58-62, 2006.

[11] 杉山寛和, 栗田太郎. 携帯電話と FeliCa を融合したモバイル FeliCa 技術. 情報処理, Vol. 48, No. 6, pp. 561-566, 2007.

A－12

車載 ECU 開発における上流工程での品質確保[1]

1. 概要

本編では、車載 ECU 開発においてモデルベース開発を適用し上流品質の向上に取り組んだ、東芝情報システム株式会社（以下、同社とする）の事例を紹介する。

先進運転支援システム、HV 車、EV 車、燃料電池車など、車載 ECU の役割は増大し、ますます複雑化している。ECU の開発現場では品質確保が絶対条件の使命であるが、設計時に予測しなかった問題が検証時に見つかることも多く、後戻りのコスト／期間の問題、市場への不具合流出のリスクなど、多くの課題を抱えている。

この課題を克服するために、上流品質の向上が叫ばれている。上流品質を向上するための手法の一つとして、シミュレーション技術を活用したモデルベース開発がある。モデルベース開発を適用することにより、上流でシミュレーションを行うことで、効率よく設計の妥当性を検証することが可能となる。また、上流で作成したシミュレーションモデルを活用し、自動コード生成を行うことができるため、開発効率も上げることができる。さらに、検証工程においても、上流で作成したシミュレーションモデルを活用した検証ができるため、品質向上と生産性の向上の両立が可能となる開発手法として注目されている。

同社では、2000 年代前半よりモデルベース開発に取り組んできている。現在は、モデルベースによる受託開発をはじめ、HILS（リアルタイムシミュレータ）やモータ／モータ制御モデルライブラリ、バッテリ／バッテリ制御モデルライブラリなど、自社製品の開発を行っている。また、モデルベース開発の導入や、モデルベース開発を効率よく活用するためのコンサルティングなども行っている。

本事例では、モデルベース開発を活用することによる上流品質の向上について、同社における車載 ECU 開発への適用の事例として、それぞれ特徴の違う 3 つのドメインを紹介する。

1 つ目は状態遷移を中心としたシステムで、比較的簡単なシステムであるが今後他システムとの連携などで複雑化する可能性が大きいため、モデルベース導入の効果が期待されるドメインである。ボディー系やエコランなどがこれに該当する。

■ ボディー系

パワーウィンド、パワーシート、ワイパー、ヘッドライトなど主にドライバが操作する装置を指します。

---

[1] 事例提供：東芝情報システム株式会社 エンベデッドシステム事業部 三島 隆司 氏

■　エコラン

アイドリングストップなど燃費向上などを目的とした装置

2 つ目の例は既に Simulink によるモデルベース開発が導入されているドメインに対し、さらに上流品質を向上させる施策を行っている事例である。

3 つ目は先進運転支援システム（ADAS）を例に、まったく新しい領域に対する上流品質の向上の取り組みを紹介する。

2. モデルベース開発とは

モデルベース開発は、上流（設計工程）で作成したモデルを基にシミュレーションによる検証を行いながら開発を進めていく手法であり、設計工程でシミュレーションを行うことによって設計品質の向上効果が期待できる。設計の不具合による後戻りを低減することによって生産性向上効果も得られる開発手法である。（図 A-12-1）

プログラム作成工程では自動コード生成による大幅な生産性の向上、ヒューマンエラー低減効果が得られる。

また、シミュレーションによる検証を行うことで、設計工程でもタイミングチャートなどを出力することができ、実際に動作した結果の見える化が実現できる。また、カバレッジ確認ツールの活用により、シミュレーションによって検証されたテストカバレッジも確認可能である。検証工程においては、シミュレータの活用や設計工程で作成した検証データの再利用による効率化が実現でき、モデルの再利用が進めば、開発期間／コストの大幅な削減が期待できる。

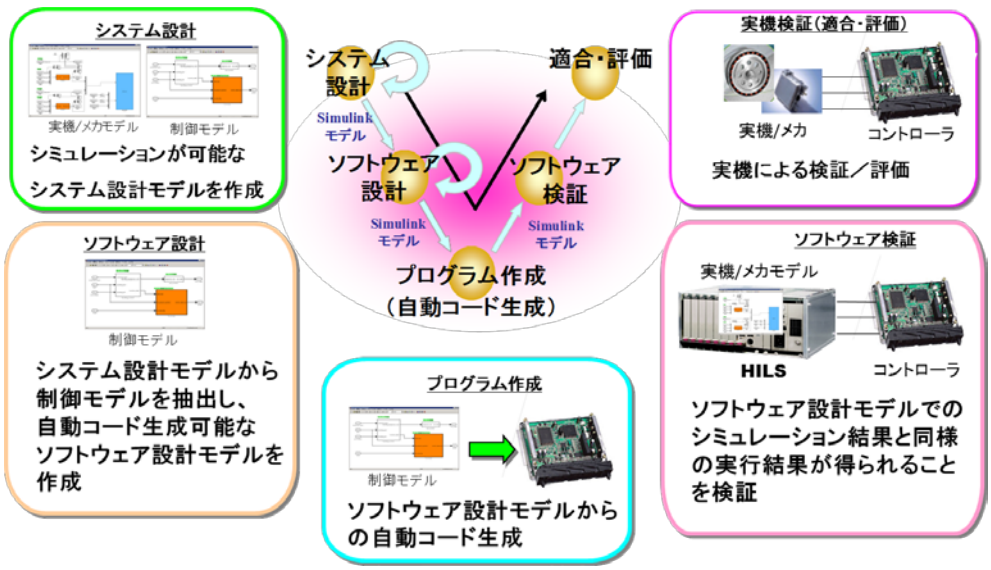

図 A-12-1　モデルベース開発全体像

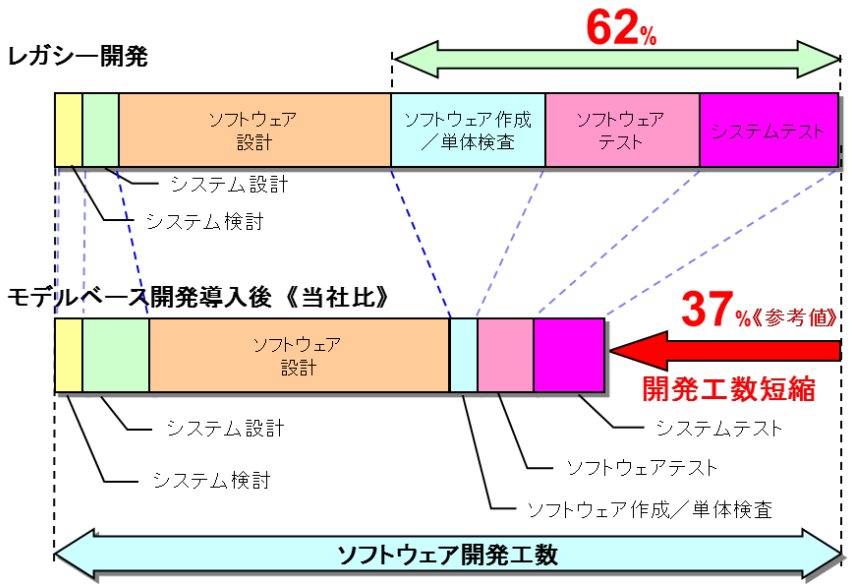

図 A-12-2　モデルベース開発の効果

モデルベース開発は、上流工程の品質向上を実現することで、下流工程の開発工数の短縮を実現することができる。　　※数値は同社のデータ（ 図 A-12-2）

3. モデルベース開発導入による設計品質の向上

これまでの設計品質の確保は、レビューによる手法が一般的である。チェックリストによるチェック、レビュー記録を残すなどの方法で、設計検証の結果をできるだけ見える化し、設計品質の向上を行ってきた。しかし、この方法では、第三者によるチェックも難しく、設計の妥当性を検証することも限界があった。（図 A-12-3）

モデルベース開発を導入することにより、シミュレーション結果で具体的に検証結果を示すことができるため、設計品質の見える化を実現し、第三者によるチェックや設計の妥当性を確認することが可能となる。

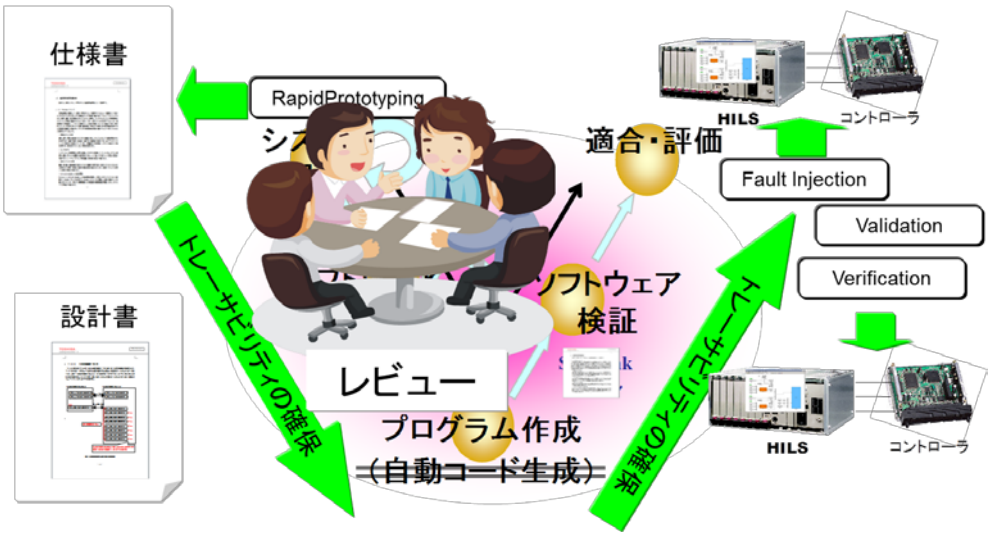

図 A-12-3　レガシー開発における設計工程の検証（レビュー）

同社では受託開発が中心であり、受託したソフトウェアの設計においては、その妥当性を検証することは非常に重要なことである。設計工程で、顧客の要求をその意図と違う解釈をしてしまうと、それ以降どの工程でも、それを発見することができず、完成したソフトウェアを納品／検収する際に大きなトラブルとなってしまう。そこで、図 A-12-4 のようなプロセスを構築し、要求からテストシナリオ（モデル検証用データ）作成し、シミュレーションを行い、シミュレーション結果をチャートとしてグラフィカルに表示することで、検証結果の見える化を実現した。これによって、顧客と設計段階で、その妥当性を確認できるようになり、飛躍的に設計品質の向上を行うことができ、ソフトウェア全体の品質を向上することができた。なによりも、顧客との信頼関係の構築に大きく貢献した。

本手法をさらに一歩進めて、要求を元にしたアサーションモデルを作成することにより、モデルを使って自動検証する手法を導入した。

アサーションモデルとは、モデル検証データと連動し、入力に対しての出力が期待値通りになっているかをチェックするモデルであり、これを検証に用いることで、シミュレーション中に自動的に異常を検出することができるため、モデル検証用データを整備することにより、自動検証が可能となる手法である。本手法を導入することにより、シミュレーション結果の確認が非常に効率的になるため、多くの検証パターンで確実に検証することが可能となり、検証カバレッジを簡単にあげられる効果をもたらした。また、本仕組みは、設計変更による再検証、差分開発の際にも大きな効果を発揮する。ターゲットとなるモデルと、アサーションモデルの変更が必要だが、設計変更や差分開発では、ベースとなる要求は大きく変わらないため、アサーションモデルは設計変更による他の機能への影響がないことを確認する効果もあり、確実にメンテをすれば大きなメリットを得られる仕組みである。

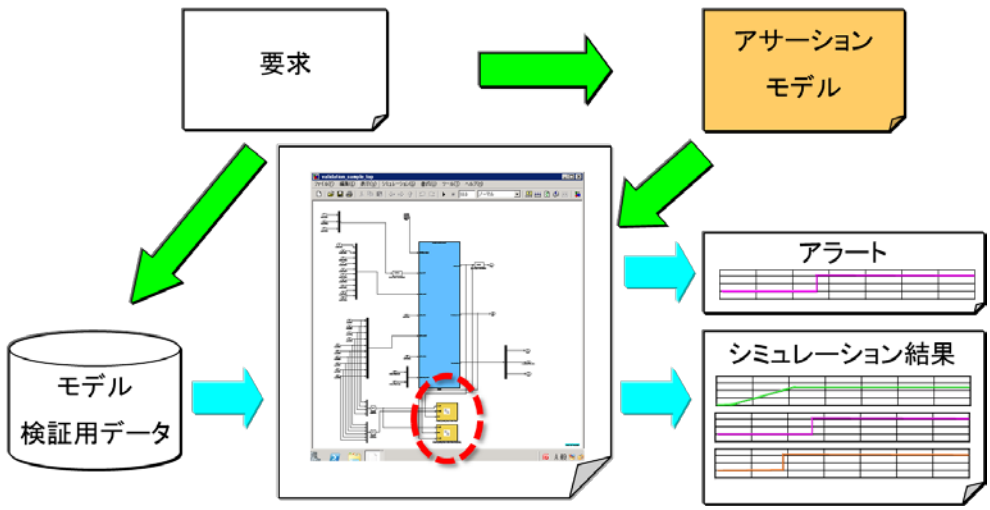

図 A-12-4　アサーションモデルを活用した検証手法

4. 要求分析の重要性

モデルベース開発を導入することによる上流品質の向上についてここまで述べてきたが、設計工程以降に関しては本手法の導入で効果をあげられることがわかった。さらに上流品質を確保するためには、要求分析が重要であり、特に差分開発が多い車載システムにおいては、要求と設計のトレーサビリティが重要である。

差分開発の場合、新しい要求をシステムで実現しようとする際に、新しい機能が思わぬところで、既存の機能とコンフリクトを起こし、その機能を損なってしまうことがある。これをできるだけ上流で解決すためには、シミュレーションが有効である。システムが複雑になってきているため、シミュレーションを活用したとしても、発生しうるすべての事象の組合せを網羅することは難しい。そのため、変更したことによる影響範囲を分析し、変更の影響を受ける特定な事象に注力しシミュレーションを行う必要がある。トレーサビリティを確実にとることで、新しい要求による影響範囲を確実に特定することができるようになるため、差分開発を行う場合は、要求のトレーサビリティは重要である。

5. 要求分析手法の確立

同社では、MATLAB/Simulink によるモデルベース開発を行っているが、要求分析工程ではこれまで実践してきたプロセス＋MATLAB/Simulink だけでは十分な分析ができなかった。そこで、図 A-12-5 のように ISO1220 のシステムズエンジニアリングプロセスに基づいたプロセスを実践するために、SysML を用いたモデルベース開発による要求分析プロセスを検討している。

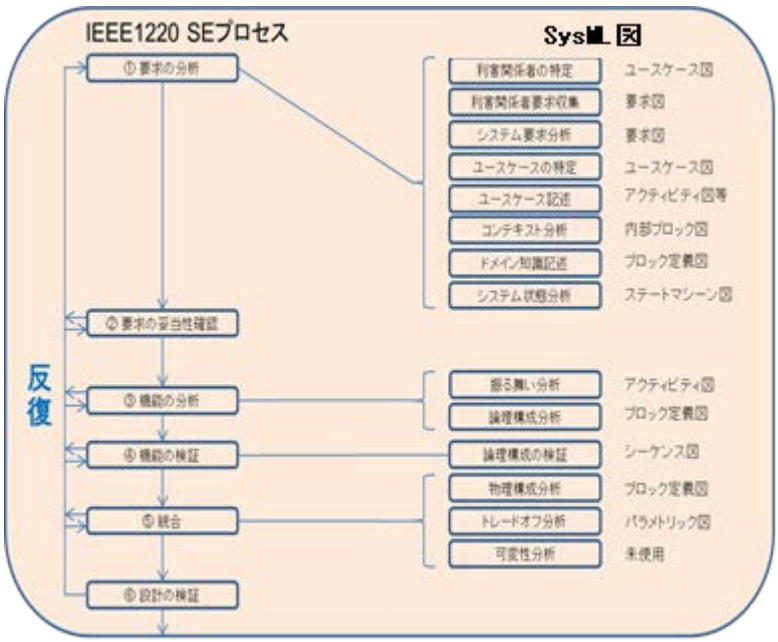

図 A-12-5　SysML を用いたシステムズエンジニアリングプロセス

図 A-12-5 の ISO1220SE プロセスの設計検証まで SysML を用いて実践した。これまで要求分析は、特定のスペシャリストが、頭の中で展開し、設計に落としてきた工程であったため、体系立てたプロセスがなく、検討した経緯を示すドキュメントなどはほとんど残っていないのが現状である。本プロセスに従って、SySML を用いて要求分析を行ってみると、検討した過程/結果が非常にわかり易く、設計変更や差分開発時に本プロセスに従って要求分析を行った SysML 準拠のドキュメントがあれば、新しい要求を追加する際の影響分析などに非常に有効なドキュメントなることが分かった。

課題としてはメンテナンス性と、設計工程で活用する MATLAB／Simulink とどのようにつないでいくべきかである。

メンテナンス性に関しては、今後ツール整備によって向上すると考えられるが、同社はその取り組みを始めたばかりであるため、現段階では有効な対策を見いだせていない。

Simulink との連携については現段階では図 A-12-6 のように、振る舞い分析以降の工程で Simulink を活用することで、SysML を活用した要求分析工程から Simulink を活用した設計工程へシームレスにつなぐアイデアがあるが、今後実践を重ねて、効果的なつなぎ方を見つけていきたい。

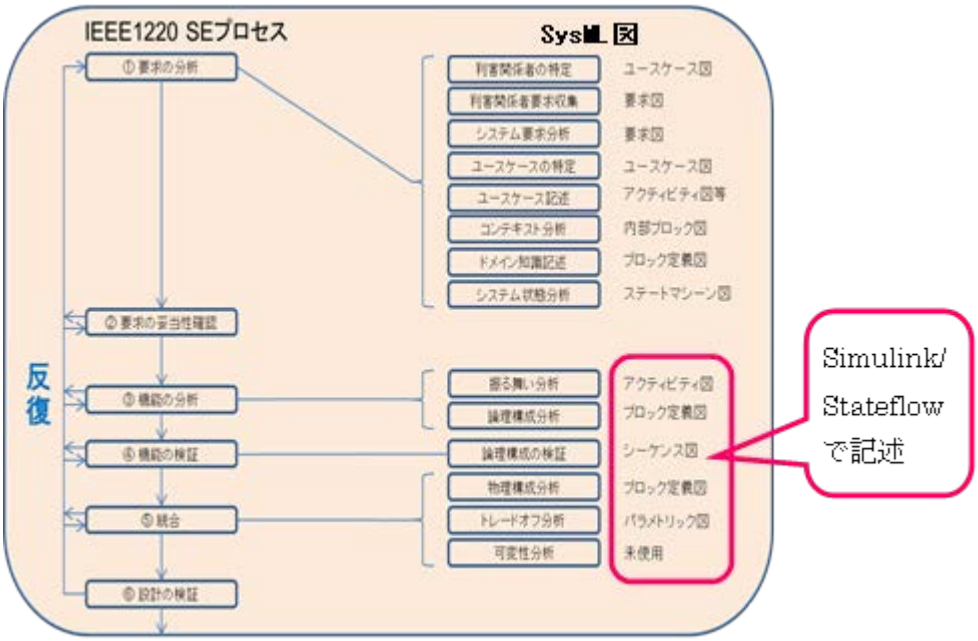

図 A-12-6　SysML と Simulink の連携検討

6. 適用事例

6.1. 状態遷移を中心とした制御システム

6.1.1. システムの特徴と背景

本システムは自動車の電子制御システムとしては、比較的簡単なシステムで、状態遷移モデルを中心として設計すれば、非常にわかり易いシステムである。ソフトウェアの大きさも小さく、長い年月をかけて作り上げた資産が多く存在するということもないため、比較的リファクタリングし易いシステムである。

レガシーなスタイルで開発され、多くの車種に展開されているため、一つ一つの開発期間が短いことが特徴である。またシステム設計者はレガシーコード（C ソース）を用いて、プロトタイプを作成し､検証を行いながらECUの制御仕様にまとめていく作業を行っている。こうしてまとまった仕様はサプライヤに渡され､ECU のソフトウェア開発フェーズに移行するプロセスになっている。システム設計部門では期間もリソースも限られているため、開発フェーズに入ってから、サプライヤとの Q&A によって、仕様をブラッシュアップしなければならないケースも多く発生し、開発の後戻りなどを誘発する要因となっている。

これらの問題を解決し、システム設計者の手間を削減する目的で、モデルベース開発を導入する計画を立てた。

6.1.2. 上流工程における課題

システム自体は確立されたシステムであるため、現在では新しい要求を盛り込むための差分開発がほとんどで、システム設計者はスペシャリストであり、仕様書を見なくても何をどうするかが見通せるため、システム設計を進めるにあたっては、要求分析、仕様検討をするためのドキュメントを用意する必要性がなかった。システム設計工程で作成された制御仕様書は、ソフトウェアの設計にかかわる細部に及ぶ事項まで記載されたものになっていた。しかし、仕様書を作る過程で検討した内容や、目的、理由が記載されていないケースがあり、ソフトウェアの設計フェーズにおいて、その部分がたびたび問題になっていた。

上流工程では、要求を分析することが重要なプロセスであるが、これまではスペシャリストの頭の中で行われていたため、ドキュメント化できていなかった。ドキュメントがないために、それ以降のプロセスを別のメンバが担当する場合、あるいは、仕様がサプライヤに渡った際に、仕様の背景／理由を仕様書から読み取ることができず、Q&A の量が多くなってしまっていた。最悪のケースとしてシステム設計者の意図と違った制御が組み込まれてしまったりすることがあった。

要求を分析してドキュメントにすることの重要性は理解できるが、何をどうすれば良いのか、現状のシステム設計者がそこまでできるのか？ということが課題であった。

6.1.3. 対策

5 で記載した、IEEE1220SE プロセスをベースとした SysML を活用したプロセスを提案した。しかし、現状は頭の中でできてしまっている工程に対して、いちいちチャートを書いて分析することが煩わしく、その手間が、システム設計者にとって大きなメリットとして直接的に返ってくる姿を十分描くことができなかったため、簡単に受け入れられるものにはならなかった。

上記の反省から、システム設計者にとってのメリットを検討することとなった。上流の品質向上による Q&A の削減が当初の目論見であったが、Q&A の削減では効果が間接的あり、システム設計者のモチベーションを持ち上げるには十分とは言えない材料であった。そこでさらに大きなメリットが出ることを模策することとなった。いろいろ検討した結果行き着いたのが検証である。仕様書は、その仕様書をインプットとしてシステムを検証できることも重要であり、上流で作成したドキュメントがそのままシステム検証に利用できるのであれば、全体の品質向上につながり、システム設計者がサプライヤから ECU を受け入れるときの品質基準にもつながるため、この仕組みを確立するためのドキュメントと位置付けて、要求分析の結果をドキュメント化することとした。

システムの検証に目線を置いた際に一番重要なのはユースケースである。検証は機能を中心に考えることが多いが、機能だけに着目すると、十分にユースケースを考慮した検証項目が導き出せない場合があり、漏れ抜けの温床になる可能性がある。仕様はユースケースを基に導き出されたものであるため、ユースケースを中心に要求分析の結果をまとめるようにすることで、検証に利用しやすく漏れ抜けの少ないドキュメントを作成することができると考えた。

要求を分析する際に用いるチャートは必要に応じて使い、中間ドキュメントして位置付けることで、必須項目からははずし、チャートを書く煩わしさのイメージを軽減した。ユースケースは Excel で記述し、そのユースケースに紐づけて仕様を展開する形式で上流ドキュメントを作成するプロセスをガイドラインにまとめることとした。こうすることによりなんとかシステム設計者に受け入れてもらうことができ、運用を開始することができた。

6.1.4. 考察

本手法を導入することで、これまで、スペシャリストの頭の中で展開していたプロセスをドキュメント化することのメリットは感じていただけるようになったと思われる。

しかし、検討した軌跡をドキュメント化する作業は、慣れた人からすると煩わしいと感じることが多いと思われる。この点で、かける手間と得られるメリットのバランスが十分ではないと感じられる。今後はツールの導入やツールによるシミュレーション機能の拡充が必要である。

6.2. パワートレーン系システム

6.2.1. システムの特徴と背景

全く新規のシステムであり、これまでは試作段階でシステム設計者が Simulink を用いて試行錯誤しながら開発をしたモデルをベースにして量産の開発に移行したものである。本システムはパワートレーンの一部であり、量産に移行するに当たり、機能安全にも対応することも必須である。

本システムは、はじめからモデルベースで開発することが前提となっているため、ソフトウェアは Simulink モデルから自動生成されたものを利用することが前提となっている。

モデルの構造は、上位のレイヤでシステム構成を俯瞰でき、下位のレイヤではソフトウェアの機能構成が俯瞰できるような構造になっており、システム設計者の視点で見やすいモデルとなっている。本モデルは動く仕様書として位置づけられていて、シミュレーションができることに加え、必要なポイントには適宜コメントが入れられ、仕様書としても機能できるような仕組みとなっている。

本システムのポリシーとして、設計書はすべて Simulink のモデルに集約することとし、開発段階で作成した中間ドキュメントは Simulink モデルに反映した後に破棄されることが前提となっている。これは、メンテナンス性を意識し、多くのドキュメントが存在するとメンテナンスされず、形骸化してしまう恐れがあるため、動く仕様書である Simulink に集約することがベストであるという思想からである。

6.2.2. 上流工程における課題

本システムは試作段階でシステムに詳しいスペシャリストが試行錯誤しながら作り上げたモデルが設計のベースなっているため、Simulink モデル以外の上位のドキュメントはほとんど整備されていない状況であった。先にも述べたように、仕様を検討する段階で中間ドキュメントは作成しているが、それをメンテナンスすることはないため、その仕様を Simulink に反映した後は、ドキュメントとしては保存されていない状況である。Simulink モデルには目的など、コメントとしては記載されているため、検討した際の情報が失われているわけではなく、モデルの中に保存されている状態である。

量産に向けて、品質向上を目指すため、上流で仕様の妥当性を検証する仕組みが確立できていないことが課題であった。仕様が Simulink のモデルの中に埋まってしまっているため、機能間をまたがって実現する仕様については、ドキュメントして記載されたものがなく、検証が漏れてしまう危険性があった。また、開発フェーズが進むにつれ、コンフリクトする要求が増え、それを調停する機能が複雑化しているため、調停する機能がそれぞれの要求を満たしているのかどうかを検証することが困難になってきている。そのために上流の仕様書をどう整備すればよいかという課題があった。

6.2.3. 対策

機能安全（ISO26262）の観点からも要求分析、要求仕様といった上流のドキュメントは、仕様の妥当性を検証するために Simulink の仕様書とは別に独立した仕様書を作成する必要があることがわかった。ただし、その仕様書はできるだけ手間をかけずに作成できることと、メンテナンス性が良くなければならない。なぜなら、手間をかけなければならない資料は、やがてメンテナンスされなくなり、メンテナンスされない資料は形骸化し役に立たないものとなってしまうからである。

個々の機能の仕様については Simulink モデルの中にわかり易く書かれているため、ここについては特に改善する必要もなくそのまま継続すべき内容である。しかし、複数の機能あるいは、システム全体として実現しなければならない仕様／目的については、Simulink の中には記載しきれていない状況であることが分かった。

一方、要求を分析し、Simulink のモデルに反映するまでのプロセスについては既に確立されていて、KJ 法によるブレーンストーミングを行い、現存のタイミングチャートを見ながら、どこに、どういう機能を追加するかをシステム目線で検討するプロセスがあった。このプロセスのアウトプットをそのまま上流工程のアウトプットにできれば実現できると考えた。

要求／目的については要求一覧にし、Simulink の仕様書に紐づけることで、要求と仕様のトレーサビリティを確保でき、これまでの仕組みもそのまま活用できる。ユースケースについては、シミュレーションシナリオとタイミングチャートで表現することとした。
Simulink でのモデルベース開発の良いところであるシミュレーション機能を活用することにより、ユースケースの保存／再現性を確保する仕組みとし、同時に検証可能な仕様を作成することができた。

次に、本仕組みを完成させるため、これまでなかった要求一覧を作成することになった。最初は現状の Simulink モデルからのリバースを考えたが、仕様から要求を抽出することは非常に難しく、この方法では真の要求にはたどり着かず、できたとしても後々利用できるものになるかどうか確信が持てなかった。そこで、要求抽出からやり直すことにした。

しかし、ここでまた課題が発生した。何を基に要求を抽出すればよいのか、白紙から要求を抽出することの難しさに直面した。そこで、もう少し目線を上にあげてみた。制御を必要とするということは、何かを実現したいということが主目的ではあるが、何かを実現するために障害となる事項を予測し、そうならないように制御することと考えた方が、要求事項が出やすいのではないかと考えた。そこで、FTA（Fault Tree Analysis）を行い、その結果を要求として反映することを検討し、部分的に実践してみた。するとほとんどの要求事項が FTA によって見えてくることが分かった。この方法を使って、要求一覧を作成することにした。要求一覧を作成し、Simulink のモデルとの紐付けを行う方針とした。

※FTA （Fault Tree Analysis）とは（図 A-12-7）

システムに起こり得る望ましくない事象（特定の故障・事故）を想定し、その発生要因を上位のレベルから順次下位に論理展開して、最下位の問題事象の発生頻度から最初に想定した特定故障・事故の発生確率を算出し、同時に故障・事故の因果関係を明らかにする手法のこと。

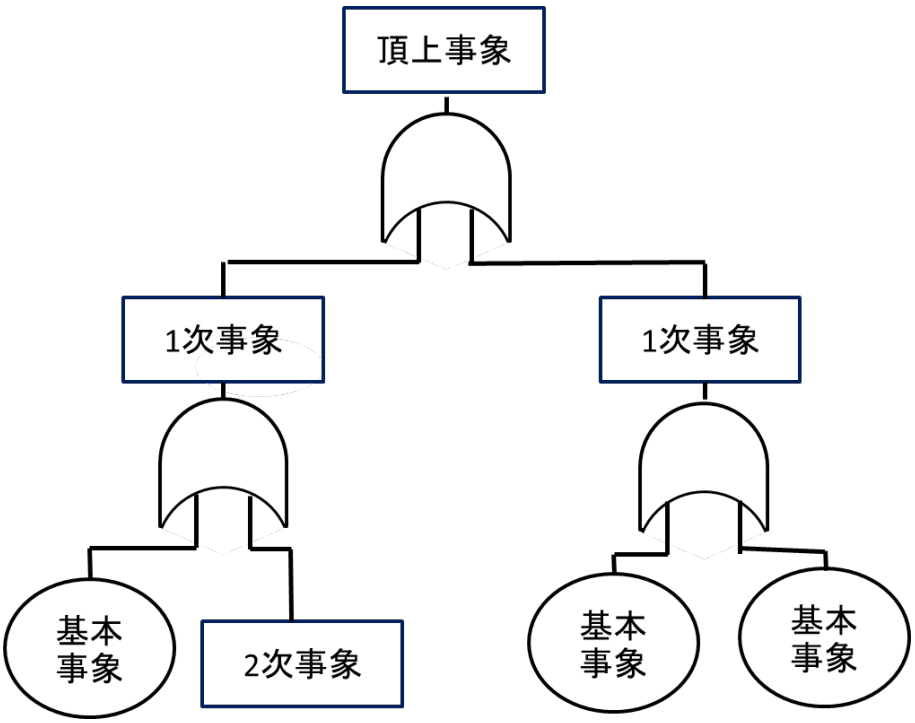

図 A-12-7　フォールトツリー

※出典：FTA（fault tree analysis）　@IT 情報マネジメント用語事典
http://www.itmedia.co.jp/im/articles/0906/17/news098.html

6.2.4. 結果／考察

現在のところ、まだ作成過程であり、成果としてはまだ出てきていないが、要求と仕様を紐づける段階で、要求のコンフリクトや、複数の要求が一つの機能に集約されている部分、要求間の依存関係など、Simulink モデルを見たり、シミュレーションを行うだけでは見えてこないことが見え始めている。

今後は本仕組みが品質向上に大きく役立つと予想され、量産開発に移行する中、本仕組みの構築を継続して行っている。

6.3. 先進運転支援システム

6.3.1. システムの特徴と背景

先進運転支援システム（ADAS）は自動車のドメインの中では新しく様々な課題を抱えな

がらも、自動車メーカ間の競争において、主戦場とも言うべきシステムであり、製品化を急がなければならい分野である。

先進運転支援システムには、プリクラッシュセーフティ（PCS)、アダプティブクルーズコントロール（ACC)、レーンキープ（LKA)、死角モニタリング、歩行者検知、ドライバーモニタリングなど様々なシステムがあるが、自動車の走る、曲がる、止まるといった基本的な機能をつかさどる部分ではなく、付加価値の部分でありながら、安全に対して非常に大きな役割を果たすシステムである。

自動車の歴史のほとんどは走る、曲がる、止まる、の基本機能についての試行錯誤であり、基本機能に関するノウハウや経験は十分に蓄えられている。しかし、ADAS についての歴史は非常に浅く経験やノウハウの蓄積がほとんどない分野である。

自動車の基本機能については、走り込むことで様々なユースケースが自然に現れるため、いろいろな道路で、多くの時間、長い距離を走ることで、多くの検証ができていた。しかし、ADAS については自車の状態だけでなく、外界の状態も大きく関係するため、単独で走り込むだけでは、ほとんどのユースケースは出現しない。ゆえに、これまでの検証方法とは全く違った検証が必要となることになる。

さらに、経験値もそれほど蓄積されていないため、想定できる範囲が限られている上に、センサなどの機能上の制限や制約がある。センサをたくさんつけることで、限界を広げたいが、コストの制約もあるため、どこかでトレードオフをしなければならない。また、複数のシステムが同時に作動した際に、機能コンフリクトを起こす可能性もあるため、システム間の背反事象を理解したうえで調停を行う制御も必要となる。このように様々な制約条件がある中で、上流の設計を行う必要があり、システムの妥当性やシステム設計の妥当性を検証することが非常に難しいシステムとなっている。

本事例では、特定の ADAS システムではなく、ADAS システム全体として、上流の設計、設計検証をどうするかという課題について試行錯誤した事例である。

6.3.2. 上流工程における課題

前述したように、課題はたくさんあるが、本件では ADAS システムにおける要求をどのように定義するか、また、その要求に対してシステムの振る舞いが妥当かどうかをどのように検証すべきかという点について絞って課題を抽出し、その対策を検討することとした。

たとえば、プリクラッシュセーフティシステムに対する要求を定義する場合、基本的な要求は、“衝突の危険性がある場合、事前に危険を回避する制御を行う。衝突する可能性が高い場合は、衝突によるダメージを最小限に食い止める。“という要求があるとすると、まず要求事項の中に前提条件がある。この前提条件はどんなシーン（ユースケース）を想定すればよいのか、どういうケースが衝突することを回避するのか、衝突が前提の制御をするのはどういうケースか、その判断の違いで、制御の方向性が大きく違ってくる。たとえば、衝突する前提であればドライバへのダメージを軽減するためエアバッグを開くという制御が考えられ

るが、衝突を回避できた場合、エアバッグを開く必要はなく、逆に衝突していないにもかかわらず、エアバッグが開くことはドライバにとって非常に不便な状況であり、場合によってはクレームにつながる可能性もある。

このような状況を避けるためには、制御が働くシーンを特定し、その閾値をどこに設定すべきかを上流工程で分析（要求分析）することが非常に重要である。製品開発の上流工程ですべてのシーンを把握し、それに合わせてすべての制御を検討し決定できれば、安全で便利なシステムを早く市場に投入できる。しかし、現状ではデータの蓄積が十分とは言えず、すべてのシーンでの検証ができていると言い切れるものはないと思われる。さらに、自動車が単独で走行することだけでは、現れないシーンがほとんどであり、様々な事象が組み合わされた時にどうなるかを検討すると、その組み合わせは天文学的数字になってしまうため、すべてのケースを予測することすら難しい。

もう一つの課題としてセンサの精度／機能についてである。技術の進歩で性能の良いセンサが安価に作られるようになってきているが、それでも、製品化となるとコストを考慮しなければならない。また、価格だけではなく、そのシステムの用途／要求に合わせて、最適なセンサを選定することも重要である。さらに１つのセンサの情報を複数のシステムで共有することも検討されている。このようにセンサを選定するということもシステムにとって重要な課題である。システムにとって最適なセンサを選定するためにも、ユースケースをしっかり分析し、センサに対する要求を明確にすることが重要である。

これまでは、まず、入手できるものを使って試作を行い、試作したシステムで実験を行い、それを次の試作にフィードバックするといった試行錯誤の末に新しいシステムを生み出してきた。実験と試行錯誤と実績の上に良いシステムが生まれてきたといえる。しかし、システムが複雑化し、実験だけではすべてのユースケースの数%にも満たないケースしか試すことができないため、“実績をフィードバックしたので本システムは妥当です“とは言えない状況になってきている。したがって、システム開発の上流でターゲットのシステムが働くユースケースをどれだけ多く検討できるかが課題となっている。

6.3.3. 対策

上流工程では要求分析が重要であり、その中でもユースケースをどうやって抽出するか、そのユースケースに対してどのような制御を行うことが適切かを検討する必要がある。まず、ユースケースの抽出についてだが、これまでのように過去の実績の積み上げが豊富にあるわけではなく、ブレーンストーミングを行い様々なケースを抽出したとしても、十分とは言えないという課題がある。この課題を解決するためにシミュレーションを導入することを考えた。ADAS では車両の挙動と、その車両に関与する外界の環境が重要であるため、それらをシミュレーションできる車両運動シミュレータを活用し、シミュレーションのパラメータを変更しながらユースケースを抽出することとなった。

ユースケースが抽出できれば、その一つ一つのケースにおいて最適な制御を検討すること

ができるため、これを仕様として積み上げることで、これまでの実績データに代わって、仕様としての積み上げが出来上がると考えた。

シミュレータを導入したからと言って、何も考えずにアットランダムにシミュレーションを繰り返しても、意味のあるユースケースにはならないため、想定シーンをある程度限定し、その中で重要なパラメータは何か、そのパラメータはどの範囲で振れるのかなどを検討し、その範囲でシミュレーションを行うことで新たなユースケースの発見を促すような仕組みを作成し、ユースケースの抽出を行うこととした。

シミュレータはユースケースを特定したあと制御の検討を行う際にも有効であり、そのシーンで最適な制御を検討する場合は、シミュレータによるアニメーションを参照しながら検討することが可能となる。また、検討した制御をすぐにシミュレータで確認することも可能となるため、制御検討した後にシミュレーションし、さらに制御をブラッシュアップするということも可能となる。

また、シミュレータはセンサの特性をどうするかということを検討するツールにもなる。センサの特性を変更することで、同じシーンでも最適な制御の方法が変わる場合もあり、それらを検討するうえでもシミュレータは重要なツールである。

こうして検討した結果を仕様として残す必要があるが、アニメーションの一部をキャプチャし、ドキュメントとして活用する機能や、シミュレーションシナリオをユースケースとして保存することで、再現可能となるため、そのシナリオとドキュメントを連携させることにより、強力な上流工程のツールとなるはずである。

6.3.4. 結果／考察

本事例では、ますます複雑化していくシステムに対して、どのようなアプローチで上流設計を進めていくかを試行錯誤している段階であり、これ以外にも様々な手法があると考えられるが、確実に言えるのは、これまでの上流工程での手法では設計が難しくなってきている。その対策の一つとして、シミュレーションを導入やモデルベースの導入は必要不可欠になると考えられる。

上流工程に積極的にシミュレーション技術を取り入れることで、製品の品質を向上することが可能になることと、さらに上流で設計品質を向上されることにより、新しい技術やシステムの開発期間を確実に縮めることができるため、他社に先駆けて市場に投入することができるなど、その効果は大きいと思われる。

7. まとめ

ここでは上流工程の品質向上の取り組みを適用した 3 つの車載システムの事例を紹介したが、車載システムでは、機能安全の対応、ADAS のような新しく複雑なシステムの登場で、これまでのシステム設計スペシャリストの経験と勘に依存したプロセスでは対応が難しくなってきている。特に上流設計の品質を確保するためには、そのプロセス、エビデンス、トレ

ーサビリティの確保なども非常に重要となっている。

本事例で紹介したようにドメインによってその手法やプロセスに違いはあるが、いずれも開発プロセスの上流で設計品質を向上させるためのプロセス、手法を定義し、これまでのスペシャリストに依存した開発手法を何とかして、ドキュメントやシミュレーションの形で可視化することで、スペシャリスト依存から脱却しようと試行錯誤している。

上流の品質を上げることは、実はそれを担う人にとっては決して楽ではない。可視化することが増えればそれなりに手間も時間もかかる。これまではスペシャリストが頭の中でやってきたために、それほど時間をかけずにできていたことが、本取り組みによって手間を増やしてしまうことになりかねない。

上流品質の向上を実現するためには、体制の変更、場合によっては組織を含めて対策しなければならないと考えられる。今回紹介した手法や取り組みは、スペシャリストがこれまで頭の中でやってきたことを可視化することで､経験を積んだスペシャリスト以外のメンバが、上流工程に参画できる間口を広げることになるための仕組みとして捉えられる。結果として、スペシャリストの負荷を上げてしまうだけの取り組みにならないように注意が必要である。

(このページは空白です)

A－13

独自開発したモデル駆動開発と プロダクトラインエンジニアリングの実践 [1]

1. 概要

本編では、上流工程でのソフトウェア設計に関わる取組みとして、株式会社デンソー（以下、同社とする）の事例を紹介する。

大規模化、複雑化、多様化する車載電子制御ユニット（ECU）の組込みソフトウェア開発において、従来型の、既存ソースコードベースの派生開発では、品質を確保しつつ作業効率を維持・向上することは難しくなっていた。

そこで、モデルを基にしたソフトウェア・アーキテクチャを自動生成する「モデル駆動開発（MDD : Model Driven Development)」を適用し、コア資産(Component)開発の効率向上、製品開発におけるコード自動生成(コーディングなし)を実現した。また組織をコア資産開発と製品開発に分け、それぞれに適した技術者の育成にも努めた。

更にソフトウェア・プロダクトライン・エンジニアリング（SPLE : Software Product Line Engineering）の実効性を高めるため、製品の機能要件を整理するフィーチャー・モデルを定義し、製品開発においてはコア資産を Component 単位ではなくフィーチャー単位で認識することにより、コア資産の再利用方法を適切に統制できるようにする。

これまでに 30 以上の ECU の開発にこの手法を適用して高い再利用性を実現できることを確認している。

2. 取組みの目的

2.1. 解決すべき課題

自動車に搭載される ECU のソフトウェア規模は、年々指数関数的に大きくなり、開発形態も大きく様変わりしている。現在では 1 台の自動車に数十個の ECU が搭載されており、単一の ECU だけでもプログラム・コードは数十万行になる。また自動車には毎年のように新しい機能が追加され、機能数が増加し続けている。ECU のバリエーションも増加する。また、グローバル化が進み、世界各国のニーズや法規制に適合させようとすると、バリエーションはさらに増える。

---

[1] 事例提供：株式会社デンソー 情報通信事業部　IVI 改革室　加藤 滋郎 氏

こうしたソフトウェアの規模およびバリエーションの増大により、ソフトウェアの開発量は大幅に増えている。かつては数名のチームで開発していたが、現在では数十名のプロジェクトチームで開発するようになり、ソフトウェアの開発者の数はこの 10 年で 5 倍以上に増えている。

このような状況下で、従来のいわゆる派生開発、すなわち既存製品のソース・コードをベースに変更を加えていくタイプの再利用では、変更が局所化されずに広範囲に及ぶため、その影響がどこまでなのかを把握するのが困難になる。機能の追加や変更に対してソース・コードのさまざまな箇所を変更しているようでは品質確保も難しくなる。

製品(自動車)の開発スケジュールに合わせて、複雑化、大規模化に対応しつつ、品質(信頼性)と開発効率（生産性）を維持・向上させることが難しくなってきた。

体系的な再利用を可能にするために、より大きな粒度で再利用できるようにし、全体を把握できるような開発スタイルに変える必要があった。

2.2. 目標

MDD および SPLE の導入、実践により、複雑化、大規模化しバリエーションも多い EUC ソフトウェアをモデルベースで設計し、ソース・コードを自動生成する開発・保守体制を確立することである。これによって、品質(信頼性)と開発効率（生産性）を維持・向上させることができるようになる。

2.3. 取組み前の状況と関係する過去の取組み

10 年程前には一枚岩の構造のソフトウェアも珍しくなかった。モジュール分割されていても、モジュール間の依存関係に注意が払われることはほとんどなく、一つの機能の追加、変更に対してソース・コードの複数箇所を変更しなければならないような再利用性の低いソフトウェア構造だった。ちょうどバリエーションも増え始めた頃であり、ベースとなるプログラムを少しずつ変更しながら同じようなソフトウェアを何度も開発していた。

2001 年ごろからまずオブジェクト指向の導入に取り組み、製品のソフトウェア・アーキテクチャの改善に着手した。しかし、オブジェクト指向の考え方をマスターし、複数のアーキテクチャ構造を適切に設計し、正確に記述できるソフトウェア技術者になるのには 5 年ぐらいの時間がかかった。特にオブジェクト指向は、原理が簡単ですぐ理解したつもりになるが実際に使いこなすことが難しい。

2.4. 目標設定のために実施した課題解決のための仮説

ECU の組込みソフトウェア開発は、バリエーションが多く、大規模なソフトウェアを高い品質で効率的に開発するためにはソフトウェアの構造、すなわちソフトウェア・アーキテクチャを重視する必要がある。例えば、建築において一軒家を立てるのと高層ビルを建てるのでは、構造や使用する技術が異なるのと同じである。大規模な建築になるほど建築工学が重要になるように、ソフトウェアも規模が大きくなるほど、技術者にはより高度な知識、技術

が求められ、ソフトウェア工学の重要性が増す。

そこで、再利用性を高め、新しい価値のある部分の開発に専念できるようになれば、開発効率も向上し、技術者のモチベーションも高まる。そして、顧客からの要求により多く応えられるようになる。そのためにはソフトウェア工学に立脚した開発体制・環境の導入が有効と考えた。

3. 取組みの対象、適用技術・手法、評価・計測方法

3.1. 取組み対象と適用工程

取組み対象は、自動車メーカー各社の多種多様な車種に搭載される機能やデバイスの性能などを制御する車載電子制御ユニット(ECU)の組込ソフトウェア開発である。

適用工程は、アーキテクチャ分析から設計、開発、テスト、保守・運用とソフトウェアライフサイクルのほぼ全般である。また、この製品は、新規から変更開発のみならず、派生開発も伴うものである。

3.2. 方法論

この ECU の組込みソフトウェア開発に、モデルを基にした開発方法と体系的再利用のためのアプローチの両方を組み合わせ、品質(信頼性)と開発効率（生産性）を維持・向上させ、品質を効率的に保証するテスト工程の仕組みをも以下のような技術・手法を適用した。

(1) モデル駆動開発：MDD（Model Driven Development）

再利用性に重点を置いているため、MATLAB/Simulink などの上流のシミュレーションを中心に迅速に開発する方法ではなく、MDD という C 言語などのソース・コードではなく、より抽象度の高いモデルを用いてコンピュータが理解できるフォーマルな形式で記述されたアーキテクチャ定義からソース・コードを自動生成する。（図 A-13-1）これにより、手動のコーディングによるミスを防ぐことができる。モデルは形式化され厳密にソース・コードとの一貫性が保たれるので、一度作成したモデルが保守されずに形骸化することもない。また、モデルをフォーマルに UML（Unified Modeling Language）メタモデルなどで記述すると、その内容や整合性をツールでチェックすることもできる。

図 A-13-1　独自開発した MDD

こうしたMDDのための仕組みは、OMG（Object Management Group：オブジェクト指向技術の標準化を普及すめるために1989年に設立された国際的な非営利団体)において以下のような技術仕様が標準化されて、利用している。

① MOF（Meta-Object Facility）：クラスを用いて構造を定義する。これによりメタレベルにおける概念（モデル要素）とその構造を定義する。

② QVT（Queries/Views/Transformations）：MOFを使って定義されたメタモデル間のモデル変換のための仕組みである。

③ XMI（XML Metadata Interchange）：モデルをXMLによりテキスト形式で記述するための技術仕様で、MOF で定義したメタモデルに則ったモデルを記述できる。

④ OCL（Object Constraints Language）：モデルやメタモデルについて、図表の形式では表現できない制約やクエリを表現することができる正確なテキスト言語である。

また、取組みを始めた頃にはOMGの標準に即した市販のMDDツールは存在しなかったが、2003年にオープンソースの統合開発環境プラットフォーム「Eclipse」プロジェクトにおいてEMF（Eclipse Modeling Framework）が公開された。独自開発したMDDは、このEMFを活用している。EMFは、MOF、QVT、XMIなどOMG の標準仕様に一通り対応している。EMF には、「JET（Java emitter templates)」というソース・コード自動生成するための仕組みがある。JETはコード生成ルールの記述にJavaを採用しているが、生成するソース・コードの種類はJavaでなくても良く、テンプレートの記述さえ変えればC言語などのソース・コード、さらにはあらゆるテキスト記述の自動生成に利用できる。

(2) ソフトウェア・プロダクトライン・エンジニアリング：SPLE（Software Product Line Engineering）

MDDだけでは不十分な点は、体系的な再利用のためのバリエーション管理ができないことである。これを支援するのが、SPLEである。SPLEは多くのバリエーションを効率よく開発するためのアプローチのことで、さまざまなソフトウェア製品群(SPL)に共通する部分の開発と個別の製品開発とを、組織的にも技術的にも明確に分けて開発する点に特徴がある。

4. .取組みの実施、及び実施上の問題、対策・工夫

4.1. 具体的取組み内容、活動

オブジェクト指向の導入に一定のメドが付いた 2004 年ごろから、再利用性および開発効率のさらなる向上を狙って、SPLE および MDD の実践に着手した。2006 年にはコンポーネント・ベースのソフトウェア開発環境を構築し、共通のコア資産からさまざまなバリエーションの製品(ソフトウェア)を体系的かつコーディング作業なしで生成する SPLE のアプローチを確立した。2011 年にはバリエーションを体系的に管理し、品質と開発効率の双方を向上できる手法へと発展させることができた。

4.1.1. 独自開発した MDD と SPLE の導入

4.1.1.1. コンポーネント・ベース開発の全体像

コンポーネント・ベースの MDD 技術を使った開発の全体像を図 A-13-2 に示す。一般に MDD では、図 A-13-2 のようにモデルを「M0 層」～「M3 層」の四つの階層に分類して定義する。

M0 層：システムを構成するオブジェクト（モデルをインスタンス化したもの）の定義

M1 層：モデルの定義

M2 層：モデルの言語使用であるメタモデルの定義

M3 層：メタモデルを記述する基盤の MOF

M0 層に近づくほど、実際のアプリケーションに、M3 層に近づくほどモデリングのための汎用的な技術基盤で構成されている。

今回開発した MDD のモデルのメタ階層は、SPLE での各活動に対応している。各層の概要は以下の通りである。

M0 層（製品開発）：

製品モデル開発は製品開発者が担当し、構成モデルからは、製品固有の C 言語ソース・コード（プロパティ・ファイル）およびビルドに必要な makefile を自動生成できる。ソース・コードの生成は、前述した JET の仕組みを利用している。JET に必要な「JET テンプレート」は、製品開発者ではなくコア資産開発者が Component ごとに事前に用意しておく。この仕組みによってコーディングなしで製品開発できるようになった。

M1 層（コア資産開発）:

コア資産のモデル開発はソフトウェア工学の専門家であるコア資産開発者が行い、Component モデルは、M2 層の EMF から自動生成される「Component モデル・エディタ」で開発する。M1 層にはこれ以外にも、UML で表記した設計モデル（ユースケース・クラス図など)、ライブラリに相当する C 言語のソース・コード（クラスの実装コード)、M0 層のプロパティ・ファイル生成のための JET

テンプレートなどが含まれる。

M2 層（メタモデル）：

メタモデルの定義は、開発環境を提供する役割も担うアーキテクトが担当する。定義されるものは、Component メタモデルと UML メタモデル、C 言語メタモデルである。

M3 層：

OMG が策定した MOF をそのまま利用するため、この層で何かを独自に定義することはない。

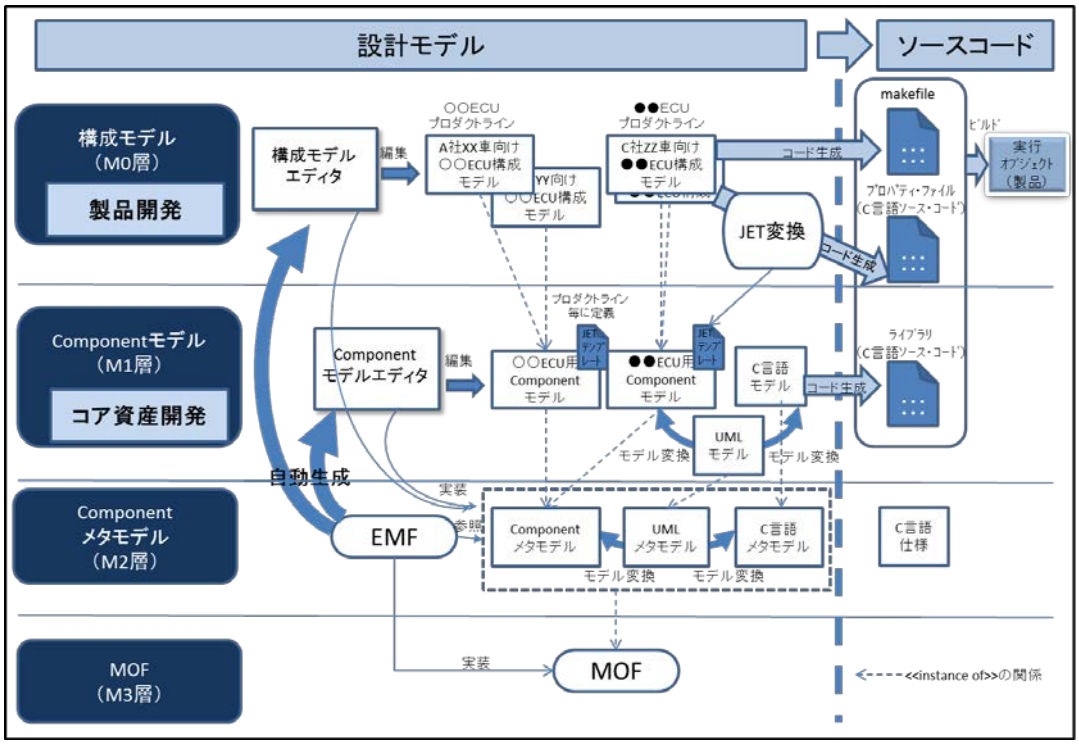

図 A-13-2　コンポーネント・ベース開発の全体像

この MDD では、Component メタモデルが中心になっている。（図 A-13-3）製品開発では構成モデル・エディタで構成モデルを定義し、コア資産開発では Component モデル・エディタで Component モデルを定義する。M3 層を除いた各階層の特徴は以下の通りである。

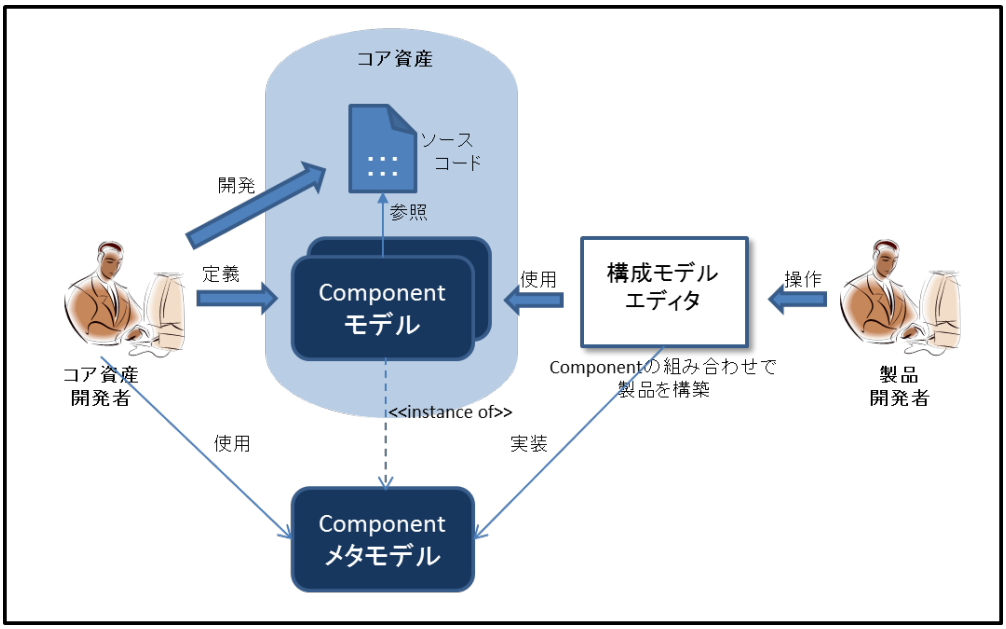

図 A-13-3　メタモデル中心の開発

(1)　製品開発（M0 層）の特徴

①　構成モデルを用いて製品を開発可能

②　作成した構成モデルとベースとなったモデルとの比較可能

③　構成モデルの設定の正しさをモデルの検証である程度確認可能

(2)　コア資産開発（M1 層）の特徴

①　JET テンプレートを Component ごとに用意することでメモリなどの資源効率の向上および性能の確保など、組込み分野ならではの実装の最適化を実施

②　OCL で制約を記載でき、Component の使用方法に一定の制限をかけることができ、構成モデル（M0 層）における Component の誤った使われ方を防止可能

③　SPLE において、構成管理をしやすくするために、Component モデルをパッケージ単位に分割し、パッケージ単位でバージョン管理を実施

(3)　メタモデル（M2 層）の特徴

①　独自開発した MDD 技術の中心（基盤）

②　資源制約の強い組込みソフトウェア開発でも使用できるように、C 言語での実装を強く意識したメタモデル

③　M3 層の MOF だけでなく OCL を併用することでメタモデルを抽象的かつシンプルにでき、実装も簡素化でき、一貫性を維持

④　大小さまざまな粒度での再利用を可能とするように、Component 同士を階層化した入れ子構造を実現

4.1.1.2. SPLE 視点から見た独自開発技術体系の特徴

(1) モデル変換を活用してコア資産開発を効率化

SPLE において再利用される「コア資産」は、「Component モデル」(M1 層)で構築し維持される。「コア資産」を含む MDD は、独自の「Component メタモデル」(M2 層)を基にして、「Component モデル」(M1 層)から「構成モデル」(M0 層)へとメタ階層の垂直方向にモデルを再利用していくものである。

更に、この MDD 技術では水平方向、すなわちコア資産開発における M1 層の中でのモデルの再利用もある。水平方向でのモデルの再利用とは、異なる種類のモデル間での変換に相当する。具体的には、UML モデルを Component モデルに変換したり、UML モデルから C 言語のソース・コードを自動生成したりする。これらのモデル変換には UML プロファイルおよび OMG の「QVT (Query/View/Transformation)」の仕組みを利用している。モデル変換を利用しているのは、コア資産開発における煩雑な作業を効率化するためである。

この MDD 技術では非オブジェクト指向で設計された既存資産も Component として扱えるように、Component モデル、ライブラリとなる C 言語ソース・コード、および製品固有部分のコード生成に用いる「JET テンプレート」といったコア資産開発活動の成果物について一定の柔軟性を持たせている。

モデル変換は、こうした冗長で煩雑な作業を自動化するのが目的である。オブジェクト指向で実装する場合、コア資産開発者は UML モデルを記述し、それ以外の Component モデル、ライブラリとなるソース・コード、JET テンプレートは UML モデルから自動生成できるようにした。

これにより、コア資産開発者は非オブジェクト指向対応のための冗長な実装や、煩雑な実装ルールに悩まされることなく、UML のモデリングに専念できるようになる。人手のミスによる不整合がなくなり、一貫性を保つために神経をとがらせる必要がなくなる。

(2) SPLE を成功させるためのコア資産と製品バージョン管理

(ア) Component 化だけでは不十分

バリエーションが多いという事業特性を踏まえ、オブジェクト指向に基づく独自の「コンポーネント(Component)メタモデル」を構築し、これを利用することにより一定の成果を上げることができたが、実践を通じて課題も明らかになった。

Component の組み合わせによる製品開発は、ソース・コード中心の開発と比べて抽象度は向上しているものの、一つの製品ソフトウェアの構成に数千の Component Instance があり、各 Component に 10 個程度の設定項目があるため、それらの組み合わせには事実上、無限のパターンがある。

また、同じバリエーションを実現する場合に、製品開発者によって異なる設定(構成モデル)が可能であるが、一つのバリエーションに対して複数の構成を許してしまうと、それぞれの構成についてテストを実施しなければならない。製品間でコア資産の再利用方法を統制して重複したテストが発生しないように管理しなければ、全体での効率化は難しい。

Component の組み合わせパターンを統制するために再利用手順書を整備しても、機能を追加、変更していく過程でモデルとの一貫性を失いがちであり、その維持には多大な労力を必要とする。

MDD の導入によって再利用性は向上したが、これらの問題を差し引くと、発効率を大幅に向上することはできなかった。

(イ) フィーチャーの導入

上記の問題を解消するために SPL(Software Product Line)のフィーチャー(feature)という概念を導入した。フィーチャーとは製品の特徴のことで、製品のバリエーション間の違いを特徴付けるものである。

Component はコア資産開発の視点から定義されるが、フィーチャーは製品開発の視点で要件を整理したものであり、製品開発者には Component よりも理解しやすく、扱いやすい。バリエーション管理において、フィーチャーが中心的な役割を果たす。

個々のフィーチャーと、その実現に必要な Component とその設定方法を「マッピング」すれば、フィーチャーを単位として製品構成を定義できるようになる。つまり、フィーチャーを選択すれば製品構成が決まり、そこからソース・コードを生成して製品を開発できるようになる。これにより、製品開発者が Component の知識を要求されたり、それらの使用方法が人によって変わってしまったりといった事態を避けられる。

また、フィーチャーとコンポーネント聞のマッピンクがあることで、製品のバリエーション管理が容易になる。バリエーションを Component の設定やソース・コードの差異で管理しようとすると複雑であるし、要件へのトレーサビリティが必要になる。フィーチャーから製品構成が決まれば、要件レベルで、バリエーションを管理できるようになる。

(ウ) FORM を参考に SPLE を実践

フィーチャーを基にした SPLE のアプローチは、「FORM (feature oriented reuse method)」と呼ばれる。製品企画を基にフィーチャー・モデルを定義して、そのフィーチャーに対応するようにコア資産を開発する。そして、製品開発ではそのフィーチャーを選択することによって適用するコア資産を決める。SPLE を実践するに当たって、この FORM を特に参考にしている。

(エ) フィーチャーもモデル化

フィーチャーは SPL のスコープを決める。つまり、フィーチャーの組み合わせで表現可能なものが、可能な製品バリエーションの範囲となる。フィーチャーは SPL の可変性(variability)を定義するものであり、その可変性を実現するようにコア資産を開発していく。

フィーチャーは、「フィーチャー・モデル」としてモデル化され、一般にツリー構造の図で表現される。今回 SPL では現在約 300 個のフィーチャーがあり、可能な組み合わせは 2^{300} 通りあるが、それでも数千の Component を扱う場合と比べれば管理可能なレベルになっている。

(オ))製品計画と密接なつながり

フィーチャーは、その製品系列における製品企画・製品展開計画と密接なつながりがあり、計画している個々の製品をフィーチャー選択モデルで表現できる。(図 A-13-4) フィーチャーを通して製品計画を整理することで、いつ、どのようなフィーチャーが必要になるか、製品群で何を共通化すればいいのかなど、コア資産の開発計画を立てられる。

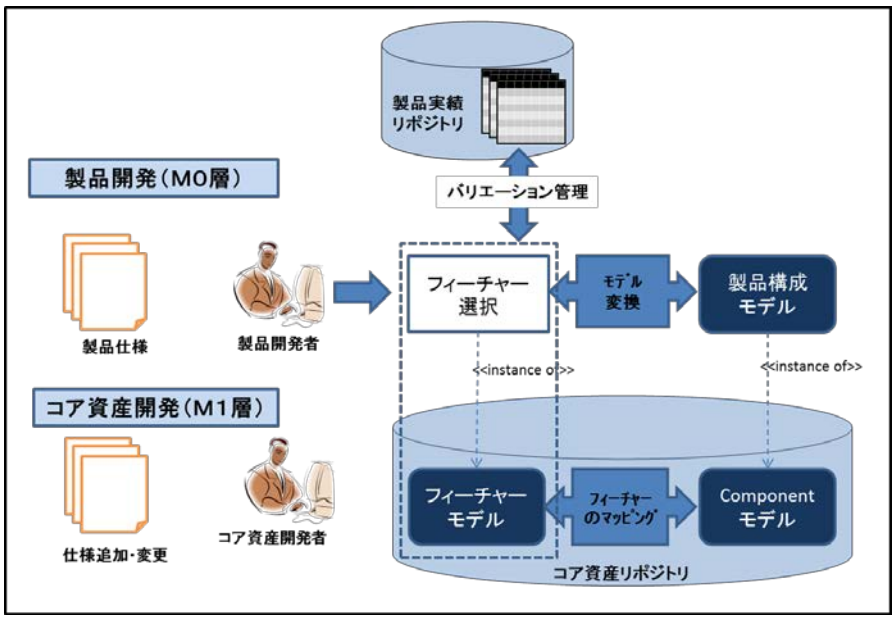

図 A-13-4　フィーチャー・モデルから構成モデルを自動生成

4.1.2. SPLE に適した組織・開発体制

SPLE では、組織もコア資産開発と製品開発とで明確に分ける。コア資産の開発は、ソフトウェア工学の知識および分析能力を備えた専門の技術者が当たる。一方、製品開発は、その特定のシステムの製品開発に従事した経験のある技術者が担当する。製品開発者には、個別の製品の特徴を把握し、それに必要なテスト設計ができるだけのシステ

ム知識が求められる。一般には製品開発者の方が人数は多くなる。例えば、組織ではコア資産開発者と製品開発者の人数比率は、およそ 1 対 5 である。しかし、この比率はコア資産が安定して再利用効率が上がるにつれて変化し、製品開発者の割合が減っていくはずである。

4.2. 導入初期のトピックス

SPL 開発の初期の頃はコア資産の品質がまだ安定しないことが多いため、製品開発で、見つかったバグをコア資産開発へフィードバックするといったやり取りが頻繁に発生する。このため、SPL 開発の初期段階ではコア資産開発と製品開発を同一のチームとして運営する、あるいは強く連携できる体制が効果的である。

コア資産の開発が進み、品質が安定してくると、SPLE の理想通り製品開発を分離して、多くのバリエーションを並行して開発できるようになる。

4.3. 人材育成、意識改革等

(1) 従来の手続き中心の設計との考え方のギャップ解消

オブジェクト指向導入のハードルは、従来の手続き中心の設計との考え方の違いである。オブジェクト指向に特有の抽象化や設計原則、デザイン・パターンなども考え方も切り替えられなければ理解しにくいものである。これらの考え方の違いは、オブジェクト指向を学習する人の不快感となり、学習の大きな障害になる。（図 A-13-5）そして、これらは実際にやって自ら体験しないことには乗り越えられないものである。この障害を乗り越えられなければ、オブジェクト指向を少し勉強しただけで導入をあきらめることになる。

しかし、一度この障壁を乗り越えられるとパラダイム・シフトが起き、考え方が大きく変わる。一度、パラダイム・シフトができると、従来の手続き型の設計に戻りたいとは思わなくなる。それは、オブジェクト指向の方が物事を考えやすく、システムの設計が簡単になるからである。

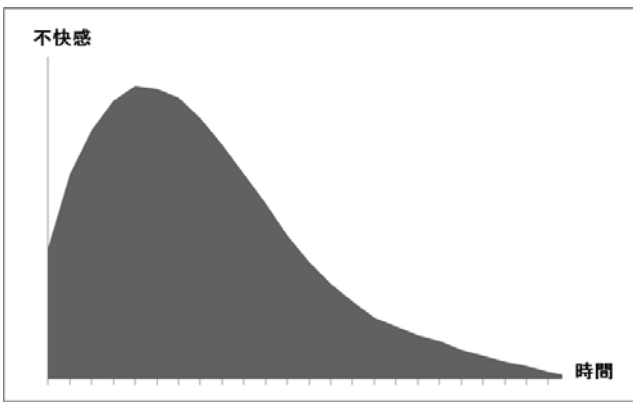

図 A-13-5　オブジェクト指向の学習曲線

※サイモン・ルイス着「サクサク smalltalk」より引用

(2) チャレンジこそがリスク低減に

オブジェクト指向に始まり、UML からのコード自動生成、MDD、SPLE と 10 年以上にわたって新しい技術や手法を継続的に導入してきた。10 年前には成熟していなかった技術や手法もある。段階的かつ継続的に各技術を導入してきたからこそ達成できたと感じる。

技術者の育成にも時間がかかる。MDD によってコア資産を構築するのにも何年もかかる。10 年前から取り組んでいなければ、現在の大規模なソフトウェア開発に適用することはできなかった。ソフトウェア開発が破綻しかけ、改革の必要性に迫られてから新しい技術に取り組み始めても手遅れになる。

新しい手法を導入する際は、これまでのやり方を変えることになる。このため、リスクを心配して反対する人は必ず出てくる。それが早い時期であれば、なおさらである。

2、3 年先のことだけを考えると、新しい技術の導入はあまり必要性を感じないものである。しかし、社会は常に変化している。10 年後、20 年後のことを考えれば、いつまでも同じやり方では対応できなくなる。さまざまな新しい手法にチャレンジしておくことは、長期的に見ればリスクを低減することにつながる。

長期的な視点で新技術や新手法に取り組もうとするのは、いつの時代も少数派である。従って、少数派の意見には常に耳を傾け、早い段階から取り組み始めなければならない。一斉に従来のやり方を変えるのはリスクが高いので、最初は小規模に導入するのがよいだろう。

(3) 最初はボトムアップ

革新的な取り組みは、技術者個人の自発的な取り組みから始まることも多い。ある程度成功して周囲の理解が得られると、最初に取り組んだ人たちが推進リーダーとなり、新しい技術を組織的に広げていく。効果があると判断されるだけの実績と、推進者の情熱が必要である。

最初はボトムアップで始まり、その人たちが推進リーダーとなってトップを動かし、次にトップダウンの指示によって組織に展開する。そして新しい技術が広まると、その中から次の技術の推進リーダーが出てくる。こうしたサイクルが回り始めると、常に新しい技術に挑戦する風土ができ、継続的な改善ができるようになる。

(4) 推進リーダーの自由と責任

新しい技術を導入しようとする推進リーダーには、さまざまなスキルおよび資質が必要である。コミュニケーション能力、プレゼンテーション能力、指導・育成力などのパーソナル・スキルは当然必要だが、欠かすことのできない資質は、自由であること、そして責任感(義務感)であると考える。

自由であるとは、過去のやり方や価値観に捉われたり、周囲の意見に流されたりせずに、新しいことに挑戦できることである。自分の知識や技術に自信がなければ周囲に流されてしまうので、自由であるためには人一倍の努力と勉強が必要だ。責任感(義務感)は自由と相反するように思われるかもしれないが、そうではない。責任感の強い技術者は自分で責任を取ろうとするために自由な意志を持てる。自分で責任が取れない技術者は、自由を棄てて前例や慣例に従い、失敗しても自己弁護できるようにするものである。

技術の推進リーダーはその責任感の下、常に自分の考えや技術が本当に正しいのか客観的理性を持って検証し、実現可能なレベルにまで、持っていく。決して直感や感情に頼った無責任な提案をしてはいけない。これらを実践するのは難しいが、推進リーダーは常に心がけておきたいことである。

(5) 技術者のモチベーションも向上

推進リーダーの下、新しい技術を導入し展開していくと、それに携わった技術者のモチベーションは向上する。一般に多くの技術者は常日ごろからスキルアップしたいと思っているものである。

MDD を導入し、コア資産を開発する際は、このように感じてくれると思われる技術者を選抜してチームを形成した。業務を通じて新しい技術を習得できるということで、すべてのメンバーが熱心に取り組んだ。理解を深めるため、業務外での勉強会も週 2 回のペースで実施した。この勉強会はその後、業務が忙しい際でも休むことなく、約 2 年間継続した。各人のモチベーションが高かったからこそ継続できたといえる。

オブジェクト指向や MDD を導入することで再利用性が向上し、開発も効率化したが、技術者が育ったことが一番の成果だったように思う。「教育は 100 年の計」と言われるように、技術者の育成は企業の長期的な存続を左右する。教育の効果は見えにくいため、つい目先の業務を優先してしまいがちだが、ある日突然、必要性に迫られて最新の技術を取り入れようとしても、それを使いこなせる技術者がいなければ導入できない。継続的に新しい技術や手法を導入し、実際の業務を通じて技術者を育成していれば、時代の進化に自然と対応できるだろう。

4.4. SPLE を実践する上での課題

4.4.1. コア資産の品質の効率的な保証方法

SPLE を実践する上での大きな課題は、どうやってコア資産の品質を効率的に保証するかである。特に自動車分野では品質を重視するため、開発コストの大部分をテストやレビューが占める。MDD 技術およびフィーチャーを基にした SPLE を導入して

も、コア資産の品質に自信を持てなければ製品開発で過剰にテストを実施することになる。この点が、品質が重視される組込み分野においてSPLEが、なかなか浸透しない要因の一つだろう。コア資産の品質を保証してテスト工程を効率化しなければ、SPLEを挿入したとしても、開発効率の大幅な向上は望めない。

そこで、SPLEにおいて過去の開発したすべての製品からコア資産の利用実績を把握して、その後の製品開発におけるテスト範囲を合理的に絞り込むアプローチを提案している。品質を重視するからといって闇雲にテストをするのではなく、その製品において注意すべき部分を具体的に特定し、テストする方法である。

SPLEで多様なバリエーションを体系的に開発していれば、比較対象を1製品に限定する必要はない。過去に開発したすべての製品と差分を特定できれば、集中してテストすべき差分範囲は1対1の比較よりも減らせるはずである。(図A-13-6)

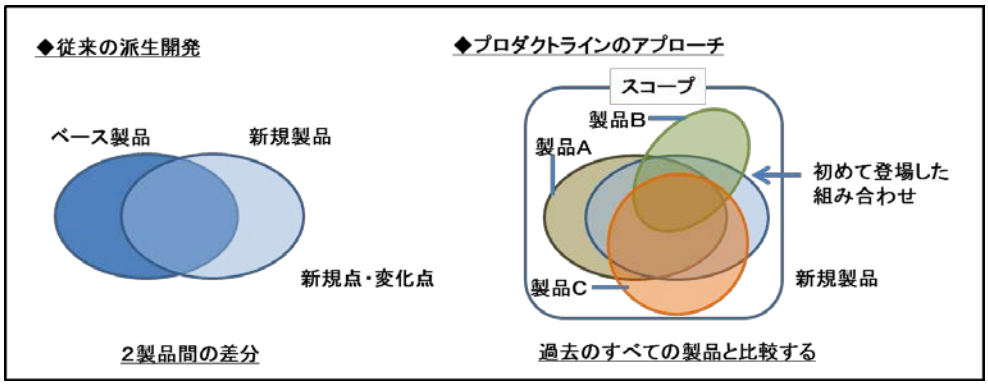

図 A-13-6　過去のすべての製品との差分に着目

5. 達成度の評価、取組みの結果

これまでに述べてきた手法を開発現場で実践し、2006年から本格的にコア資産開発を開始した。

2008年以降、延べ30を超える製品を本手法で開発している。本手法の適用の前後でソフトウェアの規模やバリエーションの数自体が変化しているため、単純に計測することは難しいが、生産性の面で2割程度の効率化が認められた。

また、コア資産を100% 再利用し、手動でのコーディングなしで多様なバリエーションを開発できるので、品質向上にもつながっている。

5.1. バリエーション管理の効果

ここで、具体的な効果をバリエーション管理のアプローチによるテスト効率化の度合いを、従来型の派生開発と比較してみる。テストしなければならないのは、各製品で新規に出現する組み合わせであるから、この新規点の累積数に着目した。

図A-13-7に、派生開発で1対1の比較で新規点を追跡した場合と、過去のすべての製品

と比較して新規点を累積した場合を比較したグラフを示す。派生開発のグラフでは、過去の製品の中で最も似ている製品と 1 対 1 で比較している。この場合でも製品数が増えるほど類似製品が存在する確率は高まるが、傾きはあまり変わらずに累積数がほぼ線形に増えていく。

一方、今回のバリエーション管理のアプローチでは、過去のすべての製品との差分で新規点を特定するため、その累積数の傾きは製品数が増えるにつれて緩やかになっていく。こうした新規点の数はテスト工数にほぼ比例し、製品開発における工数の半分はテストが占めているので、図 A-13-7 のグラフは、ほぼそのまま開発コストの累積の差と見なせる。バリエーション管理によって、製品開発を経れば経るほど、従来型の派生開発と比べて開発工数の削減効果が大きくなることが期待できる。最後の 12 製品目の新規点数は、派生開発では 400 以上あるのに対して、過去のすべての製品と比較した場合は 32 しかない。SPLE によって開発効率が 10 倍になるというのも、決して大げさなことではない。

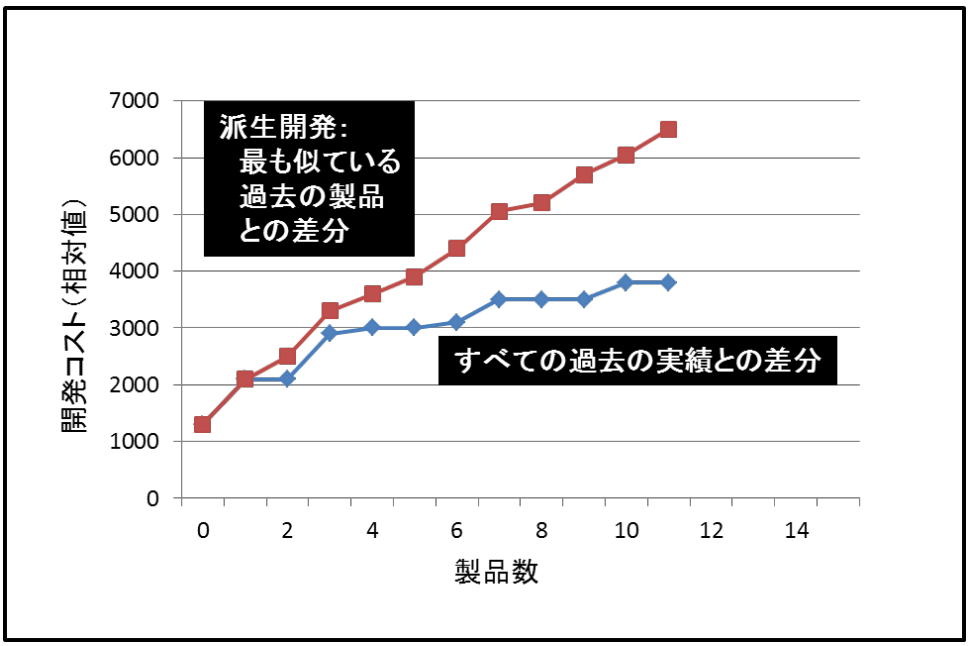

図 A-13-7　派生開発とは開発コストで大きな差

5.2. バリエーション管理の効果を最大限にする 4 つの前提

バリエーション管理は、どんな場合にもうまく効果を発揮するわけではなく、適用限界がある。うまくいく前提は主に四つある。

(1) その製品ドメインに安定したスコープであること

フィーチャーが段階的に追加されたり、拡張されたりする製品群でなければならない。多くのフィーチャーが一度に変更になるような進化が激しいドメインでは、フィーチャーの組み合わせも新規点ばかりとなり、折角の過去の製品実績もあまり生かせなくなる。同様に製品個別の特別仕様があまりないことも必要である。

(2) プロダクトラインの進化に対して安定したアーキテクチャとコア資産があるこ

と

これはソフトウェア設計上のテーマである。機能の追加や変更に対し、変更箇所が局所化されているアーキテクチャでなくてはならない。機能の追加に対して広い範囲を修正しなければならないようなアーキテクチャでは、過去の製品実績が無効になり、比較する意味が無くなってしまう。

(3) 同じフィーチャーからは同じ製品（構成モデル）が生成されること

仮に、フィーチャーは同一なのに用いる Component などが製品ごとにバラバラだと、過去の実績が役立たなくなる。MDD 技術によってフィーチャーと Component のマッピングを厳密に定義し、ツールで自動的に生成できるようにしているので、バリエーション管理の恩恵を受けることができる。

(4) 三つ以上のフィーチャー同士の組み合わせが重要にならないこと

このアプローチは、組み合わせに起因するソフトウェア・バグの多くが二つの因子同士の組み合わせで発生するとのソフトウェア工学上の経験則を前提としている。ただし、三つ以上のフィーチャーの組み合わせが重要となる場合もあるので、前述のように、三つ以上のフィーチャーの組み合わせについてもツールで対応している。

5.3. SPLE を成功させるためのポイント

最後に、実績経験を基に SPLE の取組みを失敗させる要因と成功のポイントについて考察する。SPLE の失敗要因は主に三つある。以下にその解説をする。

(1) 要件及び変更管理が不十分

ソフトウェア開発における基本的なプロジェクト管理の問題である。変更管理や構成管理といったソフトウェア開発の基本的な管理は、SPLE ではさらに重要になる。これまでは、こうした管理が徹底できていない組織では SPLE の実践は難しい。

SPLE における要求管理、変更管理では、複数の製品開発プロジェクトで顧客から並行にやってくる数多くの要求を管理して、ソフトウェアに正しく反映されているかを管理しなければならない。また、製品間で矛盾する要求、よく似ているけれども詳細が異なる要求などを共通化して再利用性を高める調整をしなければならない。そのためには、顧客からの具体的な要求を分析して、普遍的で安定した要求を抽出する必要がある。このような努力を怠ると、共通のフィーチャーが少なくなり、SPL 全体での効率化を妨げてしまう。製品個別でパッチを充てるように対応するのは論外である。そのためには、コア資産開発を統括し、コア資産の変更に対して最終的な責任を負い、コア資産のアーキテクチャを維持および管理する役割を担う「アーキテクト」が要求・変更管理に絶対に必要である。

顧客からの要求がコア資産の変更を必要とするような場合であっても、製品開

発プロジェクトで個別に要望を受けて回答するのではなく、かなわずアーキテクトを通して回答しなければ SPL を維持することはできない。プロジェクト・マネージャーだけでなく、コア資産の責任者であるアーキテクトが管理の中心になる必要がある。

(2) 再利用性や拡張性の低いアーキテクチャ

再利用性や拡張性の高いアーキテクチャの実現は、技術的な問題であり、同時に難しいテーマである。再利用性、拡張性が高くなければ、顧客からの変更要求に対して影響範囲が広くなり SPLE の効果がなくなってしまう。それには、ドメインの適切な関心事を見つけて分離すること、インタフェースと実装を分離して Component 化し、抜き差しを容易にする、凝集度を高くして、結合を弱くするといった、ソフトウェア設計の原則を徹底することが重用だ。インタフェースを分離すると、「Adapter パターン」や「Decorator パターン」といったオブジェクト指向のデザイン・パターンを利用して、新規の仕様に対して既存部分を変更せずに対応できるようになる。

抽象的にインタフェースを分離しても、アーキテクチャ上の多くの要素が具象部分に依存してしまうと変更の影響を受けやすくなる。抽象部分は普遍的で安定するが、具象部分は変更されていくからだ。具象に依存しないようにするには、依存関係を逆転させる手法が有効である。「Template Method パターン」や「Strategy パターン」 といったデザイン・パターンを活用する。アーキテクトには、適切な関心事を見つけ、それらを適切に分離する能力が必要だが、これらのテクニックを知っていると、より多くの関心事に気付けるようになる。

(3) 目先の工数の最小化を目的に個別最適に走ること

個別最適に走るという現象は、ソフトウェア開発現場では恒常的に起きている。組み込み分野で一般的な派生開発では、ソフトウェアの変更量を最小限に抑えることが美徳とされてきた。変更量が少ないほうが工数も減るし、不具合を織り込む可能性が低くなると考えられているからだ。しかし、この考え方は、個別にパッチを当てて対応することを良しとする風潮になりやすい。前述したように、これを繰り返すとアーキテクチャが崩れ、バリエーション管理のための過去の実績などが利用できなくなってしまう。また保守性が悪くなり、かえって不具合を生みやすくなる。SPLE を成功させるには、多少、開発工数が余計に掛かっても、再利用可能な本質的な変更をし、後々の開発が楽になるような対応をするべきである。

MDD と SPLE の取り組みを紹介してきたが、これらの技術はそれぞれ単独では効果が得られない、ということを最後に強調しておきたい。MDD によって再利用性を高めても SPLE によるバリエーション管理によって再利用を徹底しなければ効果がな

い。バリエーション管理も MDD によってコア資産がモデル化され、一貫したツールの支援がなければ実現できない。また、オブジェクト指向によって優れたアーキテクチャを構築することも重要だ。

これらの事は一朝一夕にできることではない。長い時間をかけて取り組みながら、組織的に改善を繰り返して達成できることである。そして、何より重要なことは、考え方の変革である。SPLE の成功のための原理は再利用である。再利用を根気よく続けていけば必ず成功するので、再利用性が悪くても諦めずに改善を続け、工夫していかなければならない。

バリエーション管理では、コア資産だけでなく、製品開発の実績まで再利用する。これまで製品開発プロジェクトは製品が完成すれば終了で、そこでの労力はそのプロジェクトのためだけのものであったが、SPLE でバリエーションを管理すれば、その労力は後に続くすべての製品で再利用されることになる。個別最適から全体最適へ考え方を変革し、すべての開発者が全体の効率化に貢献できるという意識を持てるようにすることが SPLE 成功のポイントである。

5. おわりに

多くの設計技術の適用事例を紹介いただいた。色々な工夫によって、開発工程全体の工数削減や生産物の品質の向上、あるいは開発者間のコミュニケーション向上に至るまで、一定の効果を得られたことが示されている。世の中に認知された手法をそのまま適用するだけでは、開発現場での効果にどれほど寄与できるかは疑問であり、今回紹介いただいたような適用時の実践的な細かな一工夫を行うことによって、初めて役立つものになるということを知り得た。開発現場での実践的な工夫は、まだまだ存在すると予想し、それらを引き続き収集していきたいと考えている。

本報告書の内容は、提供いただいた事例に基づいています。