要求から詳細設計までをシームレスに行うアジャイル開発手法

Agile development technique to perform seamlessly to detailed design from the requirement

藤原 啓一*
Keiichi Fujiwara

ソフトウェア設計とは、要求を、各開発フェーズごとに適した粒度で詳細化していく作業である。本稿では、あるアジャイル設計手法を例に、ソフトウェア要求分析～ソフトウェア詳細設計までシームレスに繋ぐ仕組みを示す。特に、方式設計を機能と非機能に分けて記述することで、ソフトウェアアーキテクチャの本質である非機能の実現方式を浮き彫りにすることを特徴としている。

Software design is work to make the software requirement detailed by a granularity suitable in each development phase. This paper shows the mechanism based on an agile design approach that connects software requirement with software detailed design seamlessly. In particular, it is characterized in that highlight the implementation method of a non-functional in the essence of software architecture by describing divided into functional and non-functional features.

1. まえがき

ソフトウェア製品を作る過程で作られる、仕様書・設計書、テストや障害記録、ソースコードなど、その関連を相互参照できる仕組みを、トレーサビリティ(traceability：追跡可能性）と呼ぶ。しかし、有効活用できるトレーサビリティの作り方が分からない、その効果を実感できない等の理由から、設計現場はそれを積極的に作成・維持したがらない。

一方で、ソフトウェア開発現場においては、個々人の生産性の差が5倍以上（時に10倍以上）となることが普通に起きている。経験の浅いソフトウェアエンジニアの困っている部分を紐解いてみると、作ろうとしているプログラムの上位設計書として何を見れば良いのか、どの部分を仕様として捉えれば良いのかが分からず苦労していることも多い。これは、システムの理解不足や設計手法の習得不足が原因で、解決としては、例えば、当該プログラムに関係する機能方式設計書の該当箇所と詳細設計書の対応（トレーサビリティ）を担当エンジニアに実施させ、有識者がそれをレビューするといったことが必要である。

本稿では、上記の設計手法に関する課題を解決できるよう、トレーサビリティの確立が設計作業の中に組み込まれたアジャイル設計手法を示す。本手法は、アジャイル設計手法の1つである、FDD（Feature Driven Developmen：ユーザ機能駆動開発）をベースにソフトウェア要求分析の方法論まで拡張したものである。また、特定分野でしか成功していないMDA（Model Driven Architecture：モデル駆動アーキテクチャ）の考え方を、一般的なソフトウェア開発に対して可能な限り取り入れた。それは、各フェーズ内の設計をモデリング作業と考え、フェーズ間の繋ぎをモデル間の連携と考えて、上流から下流まで設計をシームレスにつなげるというものである。

昨今、多くなっている派生開発では、ソフトウェアアーキテクチャ設計（後述の基盤方式設計）やクラス設計を経験することが無い為、その方法を知るエンジニアが育たず、いざ、新規開発となると問題が発生するという声を聞く。このような環境的な課題に対しても、本手法は具体的手順を示しているので、設計方法の疑似教育として有効利用することができると考える。

2. ソフトウェア設計手順の概要

ソフトウェア要求をプログラムに展開するソフトウェア設計手順を図2-1に示す。今回示すソフトウェア開発手法は、汎用系の表現として図2-1に示しているが、下記のとおり、プロセスの一部を修正することで、汎用系（業務系）、組込系のどちらにも使える開発手法となっている。

設計アプローチ	汎用系	組込系
業務フロー	適用。	外部イベント
データアクセス部	データベースへのアクセス	ハードウェアへのアクセス

*関西事業部 第三技術部

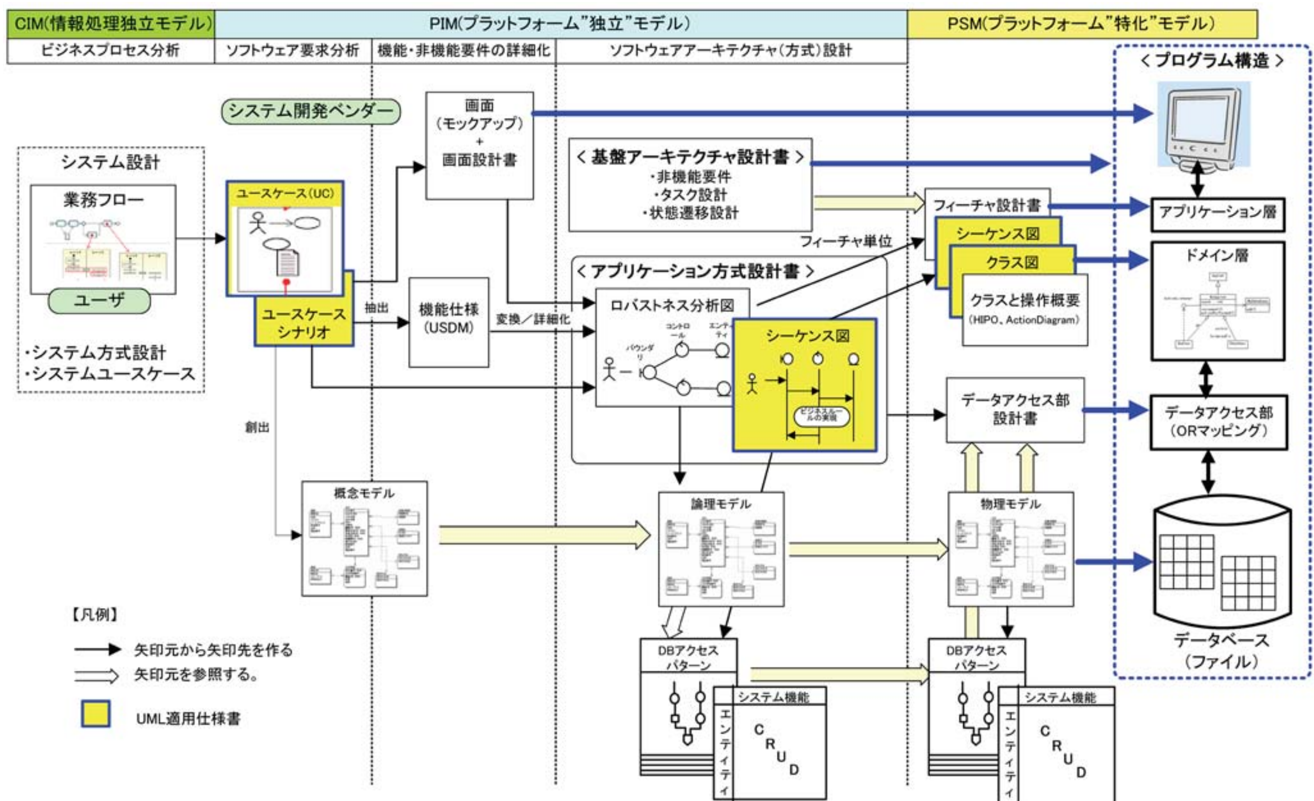


図2-1　ソフトウェア設計手法の流れ

本手法はモデリングを中心にシステムを開発していく考えを取り入れているので、MDAの分類も示した。CIM（Computation Independent Mode）は、ドメインに着目して開発するモデルで、コンピュータの処理方式は取り扱わない。PIM（Platform Independent Model）は、実装技術は特定せずにドメインの中で使用される処理方式に着目するモデルである。PSM（Platform Specific Model）は、実装技術を特定した上で詳細化するモデルである。

本設計手法の各開発フェーズにおいて採用している設計アプローチは、以下のとおりである。

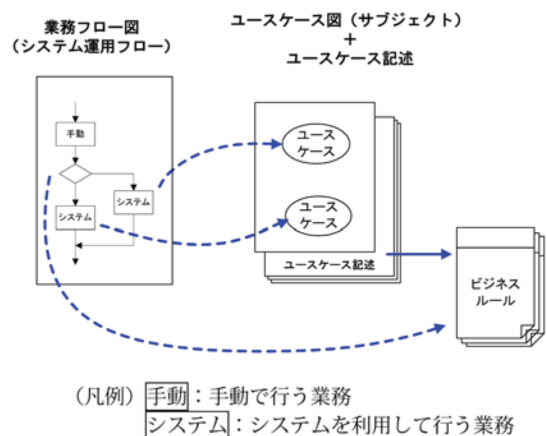
開発フェーズ	キーとなる設計アプローチ
要件定義	ユースケース、USDM
方式設計	ロバストネス分析、フィーチャー概念、 ドメイン駆動開発、OR マッピング アーキテクチャモデリング、品質特性

3. ソフトウェア要求分析

要求仕様は、①業務フロー、②ユースケース（とユースケース記述）、③USDM（Universal Specification Describing Manner）、④ビジネスルールの４つで構成される。

3.1　業務フローとユースケースの対応関係

業務フロー中の１業務（業務フロー内の□に相当）に対して１ユースケースを対応付ける。そして、１ユースケースごとにユースケース記述を作成する。ビジネスルールは、業務フローの判断や、組織で決められている規則等から抽出し、ユースケース記述と紐付ける。



また、業務フローのスイムレーンごとに定義してある「役割」が、ユースケースのアクターになると考える。組込系の場合、ユースケースは、主要なイベント対応で作成する。

図3-1　USDMの形式

3.2　ソフトウェア機能仕様

機能仕様書は、USDM（Universal Specification Describing Manner）を使って記述する。USDMの形式は図3-1に示す。USDMの特徴は、①要求と仕様を分離する様式となっていること、②要求と複数の仕様の対応が分かり易いこと、である。

3.3　ユースケースとUSDMの対応関係

ユースケースは、アクター（ユーザ）が、システムの中をブラックボックスとして、システムの外側から見えるシステム振る舞いを表現したものである。USDMは、システム内部の振る舞い（例：データ変換仕様等）を表現したものである。

ユースケースとUSDMの関係を図3-2に示す。

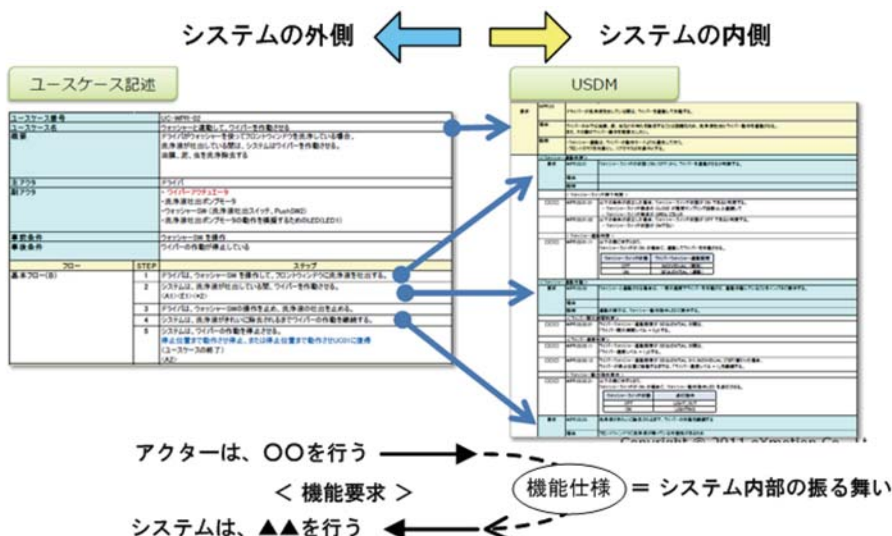


図3-2　ユースケースとUSDMの関係

ユースケースとUSDMを対応づけるルールは以下のとおりとする。

・ユースケース名（ユースケース全体）をUSDMの上位要求に対応づける。
・各ステップはUSDMの下位要求あるいはグループに展開する。
・各ステップの実現方法を仕様に展開する。

アクターの振る舞いステップの仕様としては、システムがそのイベントに反応して動作する条件、イベントのインタフェース仕様、イベントを受けた直後に行うシステムの振る舞い等を考える。アクターとシステムの振る舞いを別れた要求と考えて、仕様を考えるのが難しい場合は、それらをひとまとめの要求として仕様を考えるのが良い。

4. ソフトウェア方式設計手順

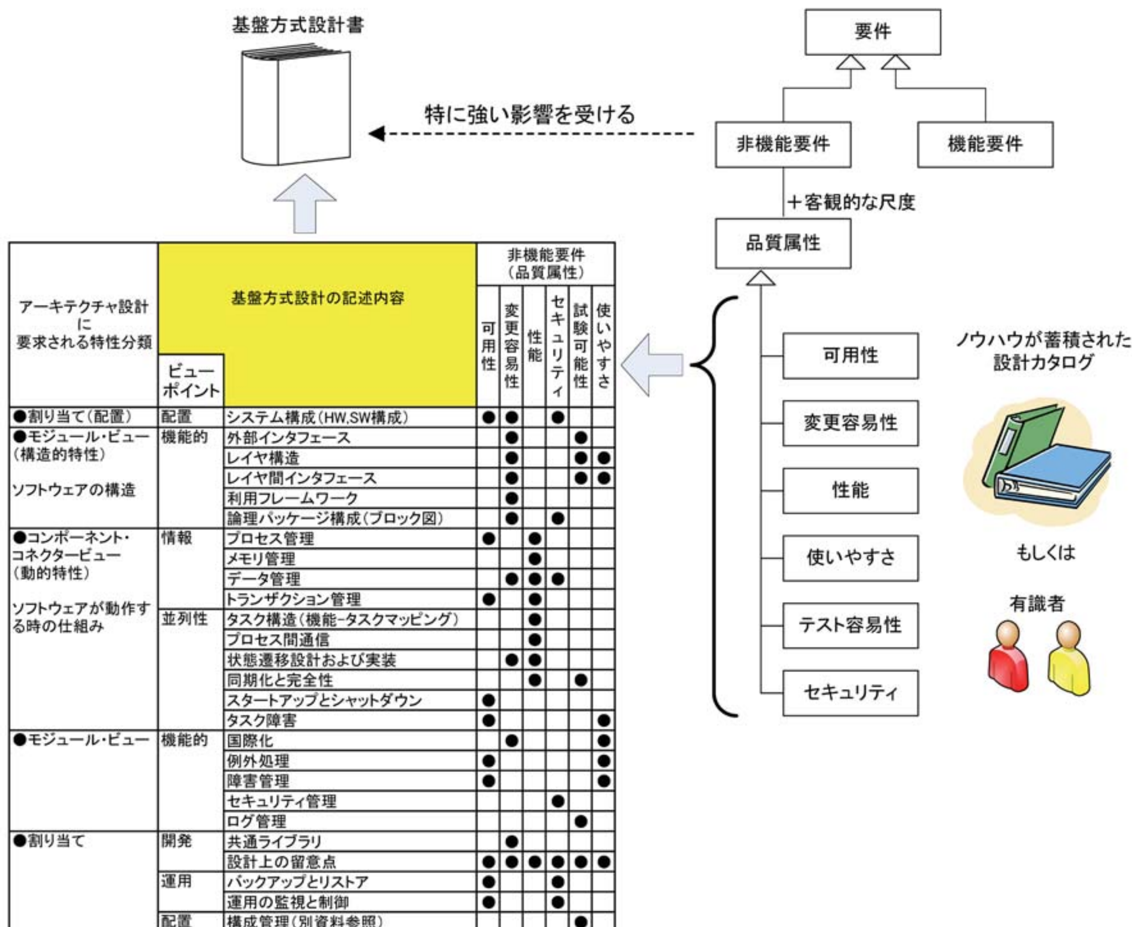
ソフトウェア方式設計は、大きく「基盤方式設計」と「機能方式設計」の２つに分けて行う。設計書も２つに分けるのが良い。

4.1　基盤方式設計

要件は、機能要件と非機能要件から構成される。非機能要件は、制約と品質を含む。そして、非機能要件の客観的な尺度として品質特性（品質属性とも言う）を導入する。品質特性は、いくつかの種類に分類できるが、ここでは、６種類（可用性、変更容易性、性能、使いやすさ、テスト容易性、セキュリティ）を使う。

基盤方式では、静的構造（モジュールビュー）、動的構造（コンポーネント・コネクタービュー）、配置的構造を表現する。静的構造とは、時間に関わらず変化しない構造であり、レイヤ関係、クラス、関数等の定義、実行時のタスクやリソースの構造である。動的構造とは、静的構造に現れる要素（クラスやタスク）の相互作用、つまり望ましいシステムの振る舞いの表現である。そして、基盤方式設計項目が１対１に非機能要件に対応することはなく、複数項目の組合せで、非機能要件を満足することになる。逆に、ソフトウェア構造（静的／動的）は、性能、信頼性、保守性など、ソフトウェアのさまざまな品質特性を決定づける要因となっている。

この様子を図4-1　基盤方式設計書の記述内容に示す。



アーキテクチャ設計に要求される特性分類	ビューポイント	基盤方式設計の記述内容	可用性	変更容易性	性能	セキュリティ	試験可能性	使いやすさ
●割り当て（配置）	配置	システム構成（HW,SW構成）	●	●		●		
●モジュール・ビュー（構造的特性） ソフトウェアの構造	機能的	外部インタフェース		●			●	
		レイヤ構造		●			●	●
		レイヤ間インタフェース		●			●	●
		利用フレームワーク		●				
		論理パッケージ構成（ブロック図）		●		●		
●コンポーネント・コネクタービュー（動的特性） ソフトウェアが動作する時の仕組み	情報	プロセス管理	●		●			
		メモリ管理			●			
		データ管理		●	●	●		
		トランザクション管理	●		●			
	並列性	タスク構造（機能-タスクマッピング）			●			
		プロセス間通信			●			
		状態遷移設計および実装		●	●			
		同期化と完全性			●		●	
		スタートアップとシャットダウン	●					
		タスク障害	●					●
●モジュール・ビュー	機能的	国際化		●				●
		例外処理	●					●
		障害管理	●					●
		セキュリティ管理				●		
		ログ管理					●	
●割り当て	開発	共通ライブラリ		●				
		設計上の留意点	●	●	●	●	●	●
	運用	バックアップとリストア	●			●		
		運用の監視と制御	●			●		
	配置	構成管理（別資料参照）					●	

図4-1　基盤方式設計書の記述内容

品質特性		実現手法	
品質特性	品質2次特性	実現手法	説明
■可用性 (Availability) システムがいかに停止することなく稼働するか	障害検出	Ping/Echo	Pingを発信して精査対象から時間内にエコーを受信
		ハートビート	一つのコンポーネントが、定期的にハートビートメッセージを出して、それを別のコンポーネントが聞き取る。ハートビートが停止した時に、発信元のコンポーネントに障害が起きたと見なす。
		異常発生	例外を捉える方法(発行と受け取り)の仕組み。
	修復準備および修復	多数決	冗長なモジュール群(同一プロセッサ上で複数の同じプロセスを実行)に同一入力を与えて、出力群から多数決などで決定、障害モジュールを検出する。
		アクティブ冗長性	バックアップも含め、冗長系をすべて並行実行させる。出力は、ただ一つのコンポーネントの応答だけを使う。そして、障害が発生した場合、別のコンポーネントを使う。
		パッシブ冗長性	1つのコンポーネント(メイン)がイベントに応答して、他のコンポーネント(予備)に状態の更新を通知する。障害が発生した場合、予備を利用してサービスを再開する。
		予備	様々なコンポーネントと交換できるように、予備の計算プラットフォームを準備する。

図4-2　設計カタログの例

基盤方式設計は、主として非機能要件（品質属性）を実現する具体的な仕組み（How）を表現するソフトウェア構造（タスク構造、ファイルの分割構造、重要な処理手順）を設計するものであり、次の２つの作業を行う。

(1) 非機能要件を満足する為に必要な設計上の方針、制約事項や設計上まもるべき原則、メカニズムの設計パターンを示す。

(2) 次に、それらを踏まえた上で、非機能を満足するソフトウェア構造はどうあるべきかを“具体的な仕組み”として表現する。

ここで重要なのは、次の３点である。

(1) 基盤方式は、ざっくりとした雰囲気（方針）を伝えるものではなく、開発、保守において具体的な判断の拠り所となるべき内容でなければならない点である。例えば、例外処理に考慮すべきコーディング規約まで含めた汎用的な準正常処理を記述しておき、詳細設計者すべてに考慮させる等。

(2) 非機能要求と基盤方式設計の記述内容のマッピング表は、必ず作成して設計書に入れる。これが、非機能要求をどのような設計で実現したかのトレーサビリティ・マトリクスとなる。マトリクスの中に書かれている黒丸（●）は、適用重要性によって、高／中／低と区別するのが望ましい。

(3) 具体的なメカニズムを考える際の検討資料（トレードオフ表等）は別資料として起こし、該当箇所から参照できるようにする（これが真のノウハウである）。

4.2　基盤方式設計を効率化する設計カタログ

事前に、品質特性項目毎に、図4-2に示す「設計カタログ」を準備しておくと、そこから仕組みのヒントを得ることが容易になるとともに、設計の知恵を流用することができるようになる。

4.3　機能方式設計

機能に対する要求は、典型的には入力からどのような出力が得られるかという、『入力から出力への変換』として定義される。機能方式設計は、この機能要求を満足する概念的な（論理的な）構造や振る舞いを明確にする作業である。ただし、入力に対してとにかく出力すれば良い、とういうことではなく内部状態に応じたリアクティブな振る舞いに関する記述は必要となる。

機能方式設計の目的は次のとおりである。

・ユースケースを実現するために必要なオブジェクトやクラスを識別する。
・必要なオブジェクトやクラス間の関係を識別する。
・属性や操作を識別して、それらをクラスに割り当てる。
・分析シーケンス図を使用して操作の振る舞いを特定する。

4.4　機能方式設計と基盤方式設計の関係

機能方式設計で表現される概念的な構造（本書では、後述のロバストネス分析図）は、より複雑性を排除するために、基盤方式設計で示すCRP（The Common Reuse Principle：共通再利用原則)、CCP（The

Common Closure Principle：共通閉鎖原則)、凝集度と結合度等の設計方針に従って、サブシステムやパッケージにまとめ直す必要がある。その結果は、機能ブロック図等で表現する。そして、個々の機能は、基盤方式設計で採用した「プラットフォームが提供するメカニズム」(例えば、アーキテクチャパターン、割り込み機能、OSの提供するタスク、プログラミング言語の提供するクラスや関数など）と対応付けて、詳細設計（後述）において、実際に動作する実現構造とするのである。

変更容易な構造を導くことを例に説明する。変更容易性を実現するには、まず、開発されたソフトウェアが対象とする世界を理解し、どのような変更を容易にするのかを明確にする。

次に、ソフトウェアの論理的な構造の作り方の方針として、以下のことを、基盤方式設計で示す。

①ソフトウェア構造化の方針決定
・想定する変化に容易に対応する為の方針
・変更が想定される機能や情報を局所化するメカニズム
②設計上の技法の選択
・決定された構造を実現する為の技法の決定
・デザインパターンの適用方針

4.5 機能方式設計と基盤方式設計の進め方

機能方式設計と基盤方式設計は相互に関係しあって並行に進める。その設計作業の進め方と内容を図4-3に示す。

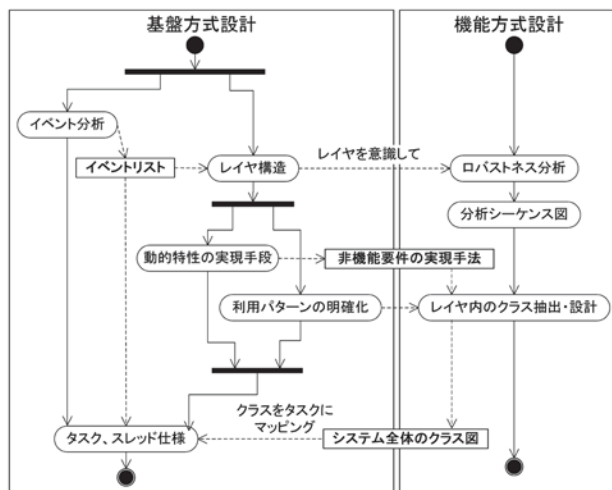


図4-3　機能方式設計と基盤方式設計の進め方

5. 基盤方式設計の内容例

非機能要求の実現方式を表現する基盤方式設計において、検討すべき内容の具体参考例として、レイヤ構造、イベント検知の仕組みの２つを示す。

5.1 ソフトウェアのレイヤ構造

現在、ソフトウェアを開発する場合、要件変更への設計・実装の変更の容易性、拡張性、等々の理由により、システム全体を論理的な階層（レイヤ構造）に分け考えるのが一般的である。機能と論理構造の違いに注意のこと。

各論理的階層の名称／分け方（階層数）は、人により様々な名称／分け方（階層数）で呼ばれているが、ここでは、情報系ソフトウェアのレイヤ構造例を示す。以降、この名称で説明する。

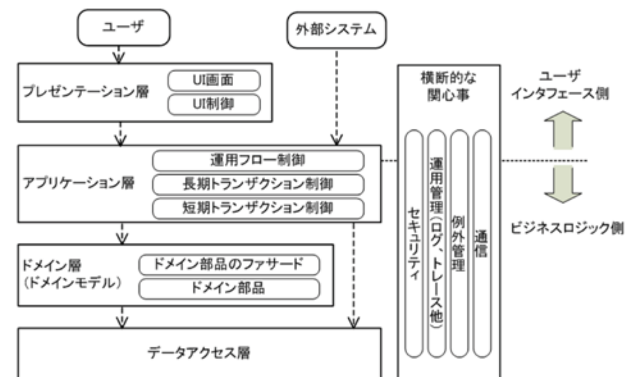


図5-1　階層アーキテクチャ

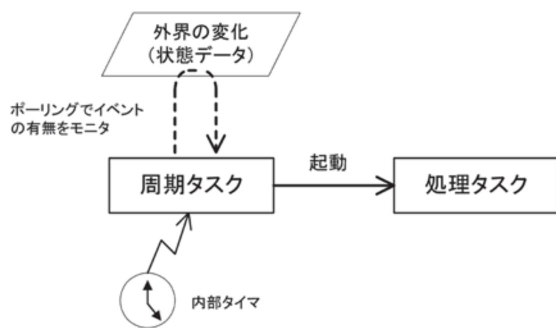
表5-1　論理的階層の役割説明

階層の名称	階層の役割
プレゼンテーション層	ユーザに情報を表示して、ユーザのコマンドを解釈する責務を負う。外部アクターは人間のユーザではなく、別のコンピュータシステムのこともある。
UI 制御	ユーザ入力値の検証、画面遷移の制御 等を行う。
アプリケーション層	ドメイン層のロジックをアプリケーション制御層以上から利用しやすい様に、まとまった 1 つのサービスとして定義する。 このレイヤには、ビジネスルールや知識を含まず、直下のレイヤに存在するドメインオブジェクトを活用してビジネスにとって意味があるものを実現する。トランザクション制御や、セキュリティ制御も含まれる。
ドメイン層	ビジネスの概念と、ビジネスが置かれた状況に関する情報、およびビジネスルールを表す責務を負う。ビジネスの状況を反映する状態はここで制御され使用される。
データアクセス層	ドメインを永続化する為の処理を実現する。典型的には、リレーショナルデータベースに対してアクセスする SQL コードを含む DAO パターンがこれに相当する。 拡張として、上位のレイヤを支える一般的な技術的機能を提供すると考えても良い。アプリケーション間のメッセージ通信、フレームワーク、ハードウェア制御等である。
横断的な関心事	横断的な関心事とは、複数のレイヤにまたがる共通機能である。通常、ここには、認証、キャッシュ、通信、例外処理、運用管理（ログ、トレース等）をサポートする機能が含まれる。これは、アプリケーション全体に影響を及ぼす。

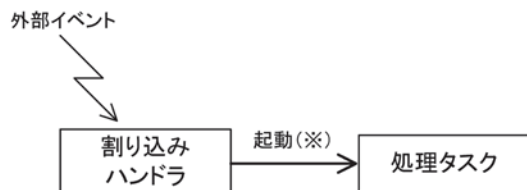
5.2　イベント検知の仕組み

機能と非機能の違いを表す例として、イベントが発生したら、何らかの処理を行うソフトウェアについて考える。機能的に見ると、どちらも、イベントの発生をトリガーとしてデータ処理を行うものであり、違いはない。非機能的には、イベント発生から処理を行うまでの応答性の違う２つの方式が考えられる。

(1) は周期的にイベントを監視し、イベントの発生を検知したら処理を行う構造である。(2) はイベントの発生で割り込みを起こし、割り込みハンドラやそれが起動するタスクで処理を行う構造である。



（1）時間駆動型：周期タスクで外界変化を監視



（※）
最小入力間隔 ＞ イベント処理のデッドライン：イベントフラグ
最小入力間隔 ≧ イベント処理のデッドライン：（キューイングされる）セマフォー
もしくはメールボックス

（2）イベント駆動型：割込みをトリガーに処理を実施。

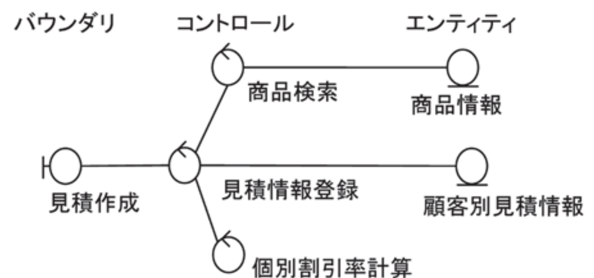
6. 機能方式設計手順の詳細

機能方式設計では、機能を構成する論理構造（コンポーネント）を明確にすることから始める。その基本は、意味的にひとまとまりの処理、ひとまとまりの情報をできるだけ同じコンポーネントにまとめるということである。これは、ロバストネス分析を利用して行う。

6.1　ロバストネス分析

6.1.1　ロバストネス分析図とは

ロバストネス分析図は、アプリケーションに要求される機能を、バウンダリ（画面、印刷）、コントロール（機能）、エンティティ（データ、ファイル）に分類し、それぞれの関係、振る舞いを表現するものである。



構成要素	定義	具体例	抽出方法
バウンダリ	バウンダリは、アプリケーションの外部とのインタフェースを表す。	・画面、帳票といったユーザインタフェース ・外部アプリケーションとの連携	ユースケース記述より、利用者と相互作用するクラスを識別する。
エンティティ	エンティティは、永続管理するデータを表す。	永続化したい「ドメインオブジェクト」 （注意）ドメインクラスと RDB テーブルは OR マッピングにより対応づけされるものであり、エンティティがデータベースのテーブルに相当すると考えるのは正しくない。	概念モデリングにおけるエンティティがクラス候補となる。 永続化手段構造（例：RDB）がどうであるか等は無視して、コントロールを保有するオブジェクトが何かを考える。
コントロール	コントロールは、アプリケーションの機能（システムが提供するサービス）を表す。	・バウンダリからのイベントの受付 ・エンティティの操作 ・他のコントロールの実行制御 ・バウンダリへの結果出力 ・アプリケーション層クラスの“主メソッド”となる。	ユースケース記述より、アクターが実行するアプリケーション操作を抽出する。

ロバストネス分析は、レイヤを意識せずに行い、後述するシーケンス図で、オブジェクトのレイヤ割り付けを行う。

6.1.2　ロバストネス分析図作成の粒度

ロバストネス分析図は、ユースケースに対応させて作成する。つまり、ユースケースひとつに対して、ロバストネス分析図を１つ作成する。

6.2　ユースケース／USDM仕様／ロバストネス分析図の関係

ロバストネス図はユースケースをオブジェクトの絵として表現したものである。つまり、ユースケース記述と機能仕様であるUSDMから、アプリケーションの機能コンポーネントの構成（関連）を決めるのがロバストネス図であると考えることができる。ただし、ロバストネ

ス図では、アプリケーションを「どのように」実行するかではなく、アプリケーションに「なに」をさせるかに焦点をあてている。

ユースケース（ユーザ要求）とUSDM（機能要求）とは、3.3章で示した関係で繋がっている。従って、ロバストネス分析のコントロールとUSDMの仕様のトレーサビリティを取ることで、コントロール仕様、すなわち、コントロールが「なに（what)」を行わなければならないかが明確になる。そして、この仕様に基づいて、ソフトウェア詳細設計では「どのように（How)」実行するかの実現方法を考える。この関係を図6-1に示す。

文章のみで記述されている機能方式設計書では、要求仕様に記述している内容の多くを再記述している場合も多く見られるが、この設計手法では、USDMに要求と仕様を集約し、参照関係（トレーサビリティ）を用いることで、二重に記述することを防ぐことができる。

また、ロバストネス分析図を作成している最中に、ユースケース/USDM仕様の曖昧さ、矛盾点に気がつくことが多々あるので、それらを、確実にフィードバックしてユースケース/USDMを修正することが重要である。

6.3　分析シーケンス図作成

⑴　作成の目的

分析シーケンス図は、ユースケースを実現するために必要なオブジェクト間のやり取りを時系列に表現したものであり、ロバストネス分析図に出てきたクラス及び、それを補完する主要なクラス間の関連、クラスの属性、操作を見つけ出すために作成する。

この分析シーケンス図は、実装を忠実に表すものとして、ロバストネス分析図で登場した、すべての論理的な静的クラスに加えて、ソフトウェアの実現に必要な新たなクラスも登場させて、それらの振る舞いを表現するものである。

⑵　具体的な作成方法

ロバストネス分析図とそれから分析シーケンス図の関係を図6-2図に示す。

アプリケーション機能の動的な振る舞いを分析する為に、ロバストネス図の構成要素である、バウンダリ(B)、コントロール（C)、エンティティ（E）をオブジェクトにしたシーケンス図（分析シーケンス図）を作成する。

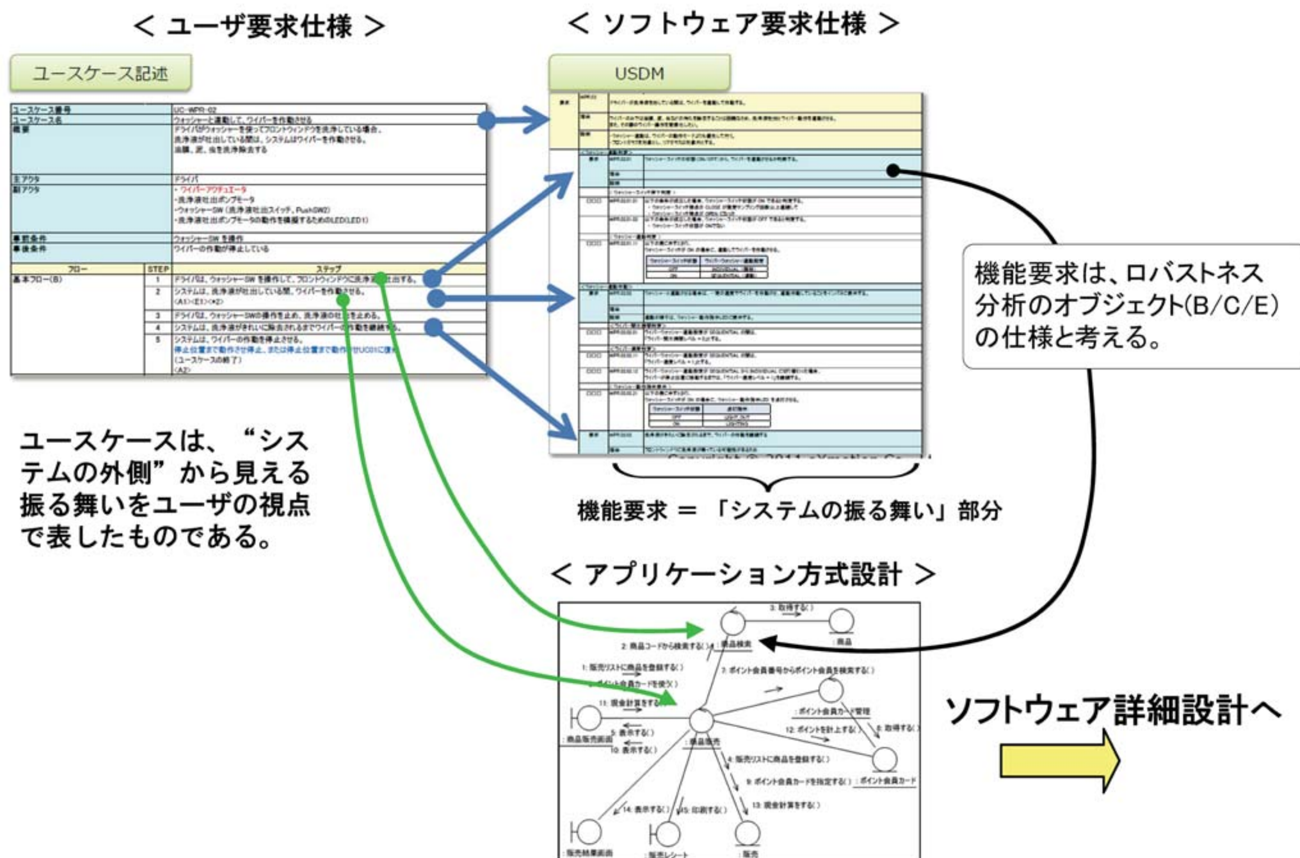


図6-1　ユースケース記述／USDM／ロバストネス分析図の関係

まず、エンティティが、どのレイヤ内に位置するのかを考える。そして、エンティティを、ドメイン層のクラスか、データアクセス層のクラスかに分類する。シーケンス図は、左から右に流れるので、バンダリ、コントロール、ドメイン層のエンティティ、データアクセス層のエンティティの順番に並べる。

コントロールとエンティティを結ぶ線は、それぞれのクラスのメソッドに相当するので、これをシーケンス図のメッセージで表現する。コントロールは、アプリケーション層のクラスの主メソッド（サービス）になると考える。

⑶　シーケンス図の作成単位

ロバストネス分析図ごとに、シーケンス図を作成する。ただし、エラー処理を記述したい場合、対応するシーケンス図は複数枚になっても良い。

ユースケースとロバストネス分析図は、１対１に対応しているので、結局のところ、ユースケースの独立性が、分析シーケンス図を過不足なく記述できるかどうかに影響を与えることになる。

6.4　レイヤ内の構成要素の設計

レイヤごとに割り付けられたオブジェクトの静的モデル作成を行う。ここでは、アプリケーション層とドメイン層に関して、設計の進め方を示す。

6.4.1　アプリケーション層の設計

アプリケーション層は、アプリケーション層以上の上位層にとって『価値のある小さな機能のかたまり』であるサービス（以降、フィーチャーと称する）がその構成要素となる。フィーチャーが、独立性の高いアプリケーション機能をプログラムとして実現する単位となる。つまり、フィーチャーは、サービスクラスの責務（≒そのクラスの主メソッド）を表している。フィーチャーは、ロバストネス分析図のコントロール（オブジェクト）の中から抽出する。

⑴　フィーチャー概念の定義

フィーチャーは、

〔目的語（オブジェクト）〕〔結果〕〔動詞（アクション）〕

ただし、〔結果〕は省略可能。

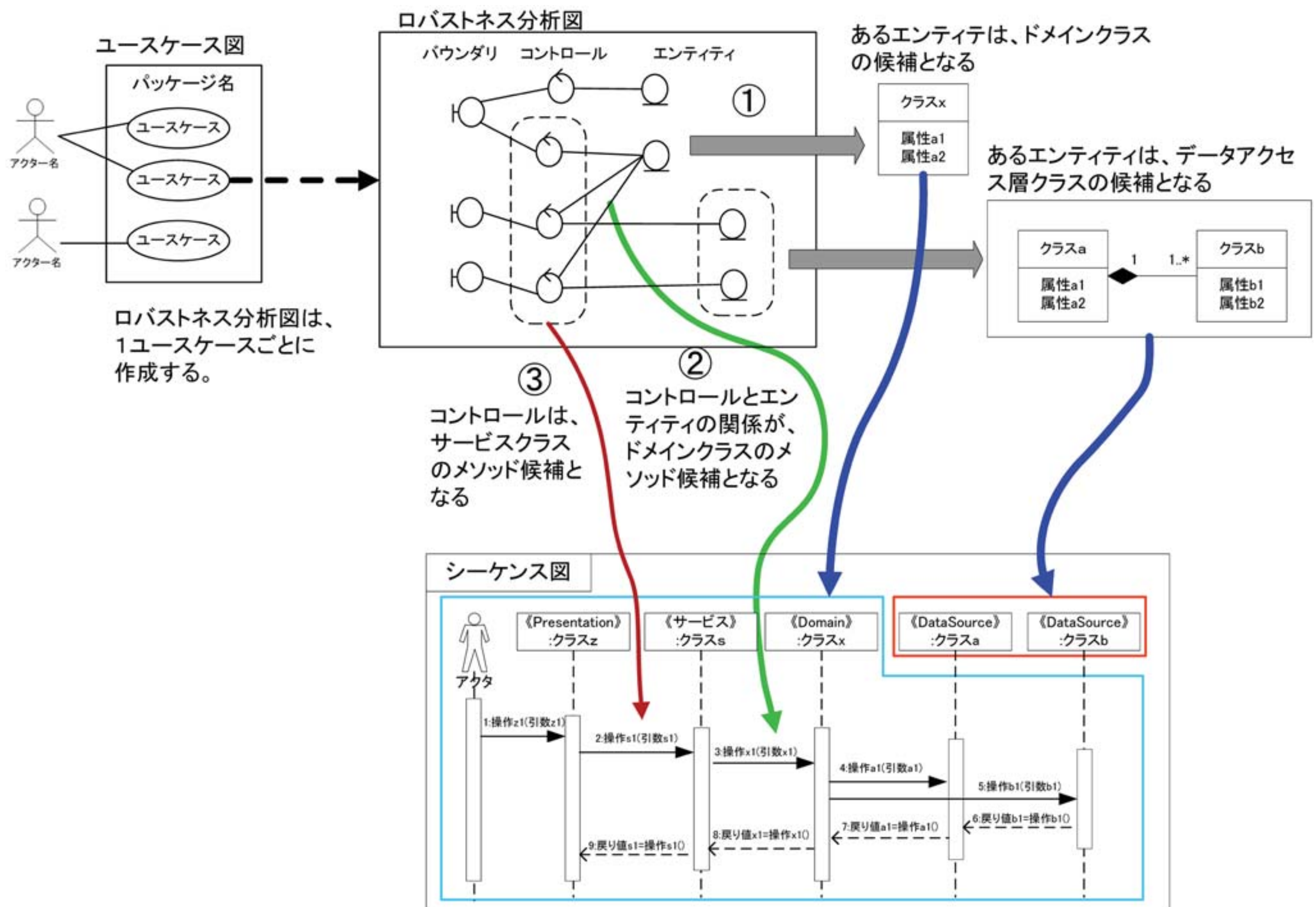


図6-2　ロバストネス分析図と分析シーケンス図の関係

という形式で表現する。
フィーチャーの例)
・販売員の成果を査定する。
・スケジュールされた自動車整備を実施する。
・カード所有者のクレジットカード取引を認証する。
・銀行預金口座の残高を取得する。

フィーチャーは、ドメインオブジェクトやドメイン層のサービスを使って、アプリケーション固有の振る舞い(アプリケーション毎に異なる処理)を実装したものである。時として、フィーチャーがデータアクセス層を直接利用する場合もある(例:データベース検索)。

⑵ 共通フィーチャーの明確化

1ユースケースごとに作成したロバストネス分析図間を跨ぐ共通なフィーチャーが存在する。これをこの段階で共通化しておくことが、詳細設計で同じものを作らせない重要な作業となる。

6.4.2 ドメイン層の設計

アプリケーションが実現しようとする問題領域(例:通信、カーナビ、○○制御等)の構造と“できるだけ”親和性をもつようにした設計モデルを、ドメインモデルと呼ぶ。ドメイン層の静的モデルは、最大限流用性を図る為にドメインモデルで作る。

⑴ ドメインオブジェクト作成手順

ロバストネス図のエンティティの関係を整理し、その関係をドメイン層のクラス(ドメインオブジェクト)図として表現する。ドメインオブジェクトは、ドメインで共通的に利用されるものという観点で抽出する。また、エンティティがすべてドメイン層のクラスになるわけでは無く、データアクセス層のクラスになるものもある。

⑵ アプリケーション層とドメイン層の関係

アプリケーション層にあるフィーチャーは、ドメイン層にある複数のドメインオブジェクト(の処理)を使って、その役割を果たすことになる。この関係は、図6-3のとおりである。

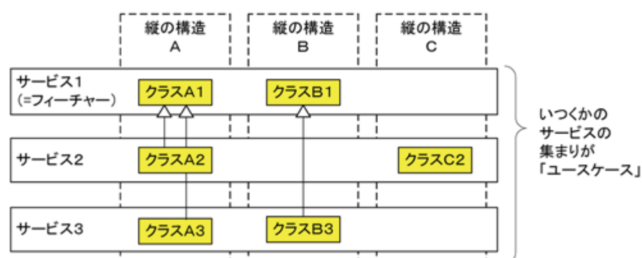


図6-3 アプリケーション層とドメイン層の関係

ロバストネス分析図は、ユースケースをオブジェクトの関連として表現したものであり、そこに登場するコントロールオブジェクトの殆どがフィーチャー(=サービス)に相当するので、フィーチャーの集まりがユースケースの機能的実現方式を表現したものと考えられる。

⑶ ドメイン層のサービスクラス

単一ドメインオブジェクトの振る舞いではなく、複数のドメインオブジェクトが協同して、1つの意味ある振る舞いを表す場合、それを、“ドメイン・サービス”として定義する。

ドメインオブジェクトに含むことができる振る舞いは積極的に含め、どのドメインオブジェクトにも所属させられないものをドメイン・サービスとする。そして、オブジェクト間の動作を含めて、ドメイン駆動開発のアグリゲート(Aggregate)に集約するという考え方でドメインサービスクラス設計を行う(図6-4)。

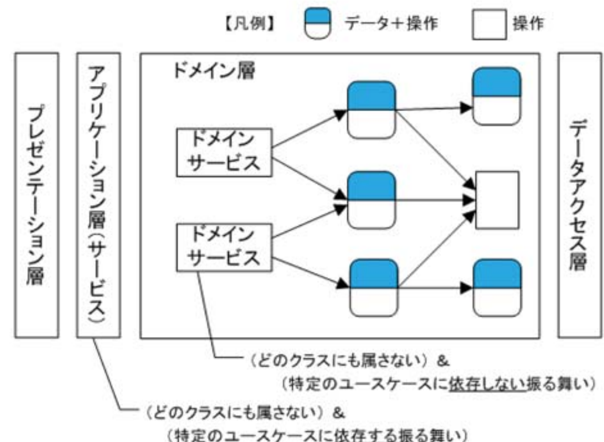


図6-4 アプリケーション層とドメイン層のサービスの違い

アプリケーション層とドメイン層のサービスの違いは対象とする相手である。アプリケーション層のサービス(=フィーチャー)は、ユーザにとって意味のある(利用価値がある)ものであり、ドメイン層のサービスは、アプリケーション層にとって意味のあるものである。

7. タスク設計(基盤方式設計)

リアルタイム性(非機能要件の性能属性)を満たす為には、アーキテクチャの動的な構造に視点を向ける。具体的には、CPU(マルチCPU、マルチコア)上で動作する塊であるタスク(プロセス、スレッド)構成とタスク間通信を検討する作業である。

タスク設計は、実時間を満たす視点からタスクという器を抽出し、機能の視点で抽出したオブジェクトをどの器に入れるかという分類を考えていく作業であり、次の手順で実行する。

①イベント分析
②タスク抽出とオブジェクトのタスクマッピング
③タスク優先度の決定
④タスク間インタフェースの決定

機能方式設計で表現される「論理的な振る舞い」に、「OSが提供するメカニズムを対応付けて性能要求を満足する」ことを表現した結果が、基盤方式設計のタスク設計となる。従って、安易に、機能＝タスクにしてはならない。

この作業はPIM領域であるが、④タスク間インタフェースの決定においては、メッセージキューやTCP/IPで行う等の方式を決め、その物理的メッセージ形式まで決定する。詳細は割愛する（参考（12）(13)）。

8. 機能方式設計から詳細設計への展開

詳細設計は、方式設計を入力として、個々のクラスのプログラム実装方法を検討する作業である。
詳細設計では、主に下記の成果物を作成する。
①実装クラス図：プログラムとして実現するためのクラス図
②実装シーケンス図：プログラムとして実現するためのオブジェクトの振る舞い
③状態遷移図：プログラムとして実現する上で重要な状態遷移
④アルゴリズム設計書：②のシーケンス図では表現できない処理手順。

この時、重要なのは次の２つである。
(1) 設計対象クラスの仕様範囲は、対象となるソフトウェア要求仕様（USDM）とトレーサビリティを確立することで明らかになる（図9-1参照）
(2) 詳細設計が基盤方式設計で規定している内容と矛盾ないことを必ずレビューで確認する。

9. トレーサビリティ確立方法

9.1 機能方式設計段階のトレーサビリティ確立方法

USDMとロバストネス分析図および分析シーケンス図を入力情報として、図9-1に示すトレーサビリティ管理表を作成する。このトレーサビリティ管理表は、要求仕様（USDM）とフィーチャー及び、フィーチャーを構成するクラスとの対応関係を示すものである。このトレーサビリティ管理表をフィーチャーリストと呼ぶ。

9.2 トレーサビリティ管理表の利用方法

トレーサビリティ管理表は、双方向トレーサビリティを確立するために用いる。このトレーサビリティ管理表は、設計フェーズでのレビューや仕様・設計変更時に威

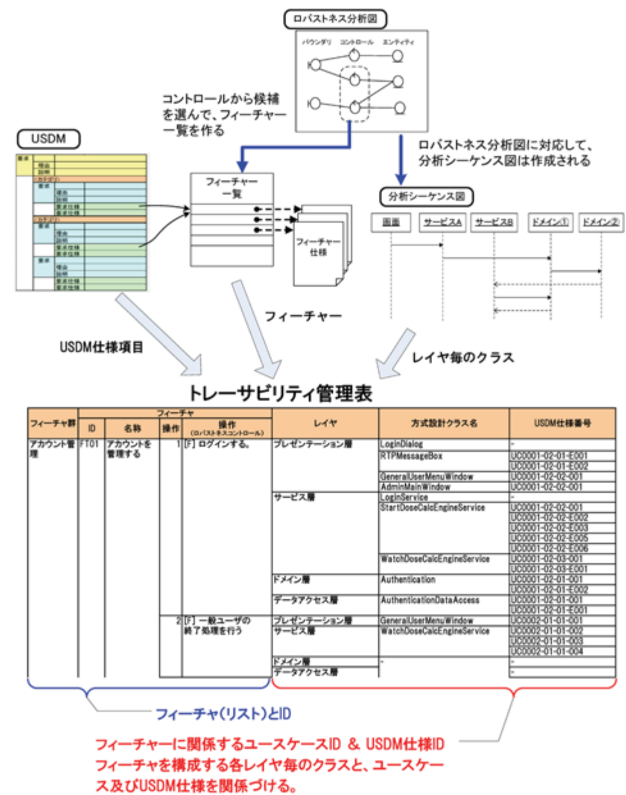


図9-1　フィーチャーリストとトレーサビリティ管理表

力を発揮する。ここでは、その活用がより有効性を発揮することを示すために、トレーサビリティ管理表は、何らかの形でデータベース化され、検索やフィルタリング機能を実現できていると仮定する。

仕様・設計変更時に、関係設計書をどのように取り出すのか、その手順を説明する。ただし、既に確立された仕様項目内の変更か、仕様追加かによって、関係設計書の取り出し方は異なる。
(1) 既に確立された仕様項目内の仕様変更
①USDMを使って変更仕様を特定する。
②<トレーサビリティ管理表利用>特定した変更仕様項目に関係するフィーチャーを取り出す。
(2) 新規の仕様追加
①USDMを使って追加仕様に関係するであろう既存仕様を特定する。
②（これ以降は、既に確立された仕様項目内の変更に同じ）

既に確立された仕様項目内の仕様変更よりも、新規の仕様追加の方が多い為、システムを変更しようとする人は、『システムの理解』が重要なファクターとなり、ここに多くの時間を要する。

それでも、既存仕様・設計書間にトレーサビリティが確保されていれば、変更箇所を特定する作業は楽になる。

10. どの部分がアジャイル開発なのか？

アジャイル開発とは、「顧客価値を持続的に提供するために、繰り返しながら、コミュニケーション重視で、チームを改善しつつソフトウェア開発に取り組むやり方」である（設計せずにコーディングを始める開発というのは誤解）。開発プロセスとして見た場合、一番重要な部分から、動くソフトウェアを細かく作って完成させていく「短期の繰り返し型」であることが一つの特徴である。

本手法の中では、独立性の高い、ユーザから見た小さな機能であるフィーチャーが繰り返しの単位となる。
フィーチャーは、設計とプログラム開発を２週間程度（もしくは更に短い期間）で開発が可能な単位であり、独立性が高いので複数人で並行開発、かつ、反復（イテレーション）開発ができる。

11. 短納期開発が可能

本手法では、レイヤ毎に並行開発が行える。十分なリソースを準備すれば、どれだけ開発する規模が大きくとも、各レイヤは１～２ヶ月で詳細設計～単体テストを完了させることができる。
また、ドメイン層がかなりの割合で流用可能なまでに成熟していれば、プレゼンテーション層と、アプリケーション層のフィーチャーの追加・修正でプログラムを完成させられる。

12. ソフトウェア設計品質検証方法

設計プロセスよりも一段詳細な設計手法を決めて、そのトレーサビリティをどのように確保するのかの説明を行った。これで分かることは、設計そのものがスパゲッティになっている、つまり、仕様項目や設計項目間の結合度が高い場合、いくらトレーサビリティを確立したとしても、下位設計から上位を見れば多くの設計条件があり、上位設計の絡まり具合を下流に持ち越したままになってしまうということである。

そこで、設計品質の良し悪しを計る指標として、「成果物項目間の多重度」を用いることを考える。成果物項目間の多重度とは、表12-1のように、ある成果物の項目が、別の成果物の項目といくつの関係を持っているかを示す指標値のことである。

良い設計が行われた際、成果物項目間の多重度が取るべき大凡の範囲が予め分かっていれば、その範囲内にあれば良い設計、範囲外であれば悪い（どこかおかしい）設計、と判断できるようになる。

表12-1　多重度管理表

	USDM仕様項目（数）			フィーチャー（数）			クラス（数）			プログラム規模（KL）		
	最小	最大	単位	最小	最大	単位	最小	最大	単位	最小	最大	単位
ユースケース（UC）	5.0	15.0	仕様/UC	3.0	6.0	FT/UC	8.0	20.0	クラス/UC	1.0	1.8	KL/UC
USDM仕様項目				3.0	5.6	仕様/FT	3.8	5.9	仕様/クラス	14.0	22.0	仕様/KL
フィーチャー（FT）							1.0	3.0	クラス/FT	88.0	178.0	Line/FT
クラス										10.0	18.0	クラス/KL
メソッド										15.0	25.0	Line/メソッド

13. むすび

本設計手法は、いくつかの実プロジェクトに適用して効果が見えてきているものである。

設計手法を統一する意義は、エンジニアの目を設計品質と生産性向上（流用）に集中させられること、ドメイン知識を有すれば、エンジニアの流動性を高められることにある。特に、レビューにおいて、忙しい有識者がチェックするポイントを統一できる効果は大きい。これは、構造化プログラミングが果たした成果と同じことが、設計プロセスレベルで発揮できていることになる。

基本的な手法は変えず、多重度のような改善概念を付加していくことで、さらに、ソフトウェア設計における生産性・品質向上が図れると考える。

謝辞

本手法を多くの実プロジェクトに適用した際に発生した課題に対して、現場で使えるよう工夫・改善を継続してくれている関西事業部第三技術部の各位に感謝する。

参考文献

（１）ソフトウェアアーキテクチャ　岸知二、野田夏子、深澤良彰、共立出版（2005/6）
（２）実践ソフトウェアアーキテクチャ　Len Bass/ Paul Clements/ Rick Kazman 前田卓雄他訳、日刊工業新聞社（2005）
（３）システムアーキテクチャ構築の原理 ITアーキテクトが持つべき3つの思考（IT Architects’ Archive ソフトウェア開発の実践）ニック・ロザンスキ、イオイン・ウッズ、榊原　彰、牧野祐子、翔泳社（2008/12/3）
（４）リアルタイムUMLワークショップ　ブルース・ダグラス　鈴木尚志訳、翔泳社（2009/12）
（５）オブジェクト指向設計 滝澤克泰、ソフトバンクパブリッシング（2005/1）
（６）【改訂第２版】要求を仕様化する　技術表現する技術　清水吉男、技術評論社（2010/6）
（７）「派生開発」を成功させるプロセス改善の技術と極

意　清水吉男、技術評論社（2007/11）
（8）Java/Webでできる大規模オープンシステム開発入門　林浩一他、丸善出版（2012/10）
（9）ユースケース駆動開発実践ガイド　ダグ・ローゼンバーグ、マットステファン、佐藤竜一他訳,翔泳社（2007/10）
(10) 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　マーチン・ファウラー　長瀬嘉秀監訳,翔泳社（2005/4）
(11) エリック・エヴァンスのドメイン駆動設計エリック・エヴァンス　今関剛監訳、翔泳社（2011/4）
(12) リアルタイム組込OS　Qing Li/Caroline Yao翔泳社（2005/11）
(13) 組み込みソフトウェアの設計＆検証　藤倉俊幸、CQ出版（2006/9）

執筆者紹介

藤原　啓一
1985年入社。関西事業所に配属。入社～1992年、防衛のソフトウェア開発に従事以降 カーナビ、気象レーダ、ニューラルネット、医用画像、衛星通信、業務系システム等のソフトウェア開発に従事。
2014年（現在）通信に関するソフトウェア開発部長。