修 士 論 文

リエンジニアリングによる レガシーシステムの ソフトウェアプロダクトライン化 に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

田中 憲吉

2009 年 9 月

修 士 論 文

リエンジニアリングによるレガシーシステムのソフトウェアプロダクトライン化に関する研究

指導教官　落水 浩一郎 教授

審査委員主査　落水 浩一郎 教授
審査委員　小川 瑞史 教授
審査委員　鈴木 正人 准教授
審査委員　青木 利晃 准教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

0710901 田中 憲吉

提出年月: 2009 年 8 月

概 要

近年，組み込みシステム分野のソフトウェア開発の発展が目覚しい．一方，ソフトウェアの大規模化と複雑化による，開発工数の増加や品質の劣化が問題となっている．これは，組み込みシステムの特徴である少量多品種の生産と，短いライフサイクルに起因する問題であると考えられている．このような少量多品種のソフトウェア開発を支援するソフトウェア開発方法論にソフトウェア・プロダクトラインがある．ソフトウェア・プロダクトラインは，その有用性が注目されているところであるが，共通資産であるコアアセットを準備するための初期投資が大きく，導入への敷居を高めている．本研究は，ソフトウェア・リエンジニアリングの手法により，レガシーシステムを再利用して，効率的にソフトウェア・プロダクトラインの資産を構築する手法を提案する．

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

近年，組み込みシステム分野のソフトウェア開発がめざましく発展してきている．身近な組み込みシステム製品として，携帯電話やゲーム機，カーナビなどの情報家電はもちろん，一般の電化製品や医療機器，自動車の制御装置など，様々な分野に活躍の場を広げている．昨今では，従来，情報化とは無縁と思われていた白物家電や設備機材のような製品までもが，競合製品との差別化を目指し，組み込みシステムを積極的に導入するようになってきた．こうした流れは，ユビキタス社会の到来を確信させるものであり，今後，組み込みシステムの益々の発展を予感させるに十分な根拠となっている．

一方，組み込みシステムが隆盛に向かう反面，ソフトウェアの大規模化と複雑化による，開発工数の増加や品質の劣化が問題となってきている．一般に，組み込みシステム製品は，競争と技術革新の速さから，製品の寿命が短い．携帯電話などは，わずか数ヶ月でモデルチェンジが行われることがある．その他の製品でも 1 年から 2 年，長くても 3 年程度でライフサイクルを終了するものが多い．また，現代の個別化した顧客のニーズに答えるべく，デザインやグレード，オプションの異なる複数の同系製品を，平行あるいは同時に提供する販売戦略も一般的になってきた．こうした事情から，組み込みシステム製品は，少量多品種の生産と，短いライフサイクルを特徴とするに至っている．これらの特徴は，組み込みシステム製品のソフトウェア開発に，次のような影響を与えている．

- 系列製品間の差異による，ソフトウェア構造の複雑化
- 類似商品による，クローンコードの発生と保守性の低下
- 新規商品の追加影響による，既存商品の品質劣化
- ライフサイクルの短期化による，開発期間の短期化

こうした問題は，開発現場に多くの負荷をもたらすのみならず，製品への信頼をも揺るがしかねないリスクとして認知されており，即急な打開が求められている．

少量多品種のソフトウェアを，効率的に開発することを支援するソフトウェア開発方法論に，ソフトウェア・プロダクトラインがある．ソフトウェア・プロダクトラインでは，特定ドメインに向けたソフトウェアの開発に必要となる機能を，あらかじめ共通資産として細分化して作成し，それらを組み合わせることで一連のソフトウェア製品群を生産する

手法を取る．組み込みシステムのような，少数の類似するソフトウェアを，短い期間で開発するに有効な手法として注目されている．ソフトウェア・プロダクトラインの問題点の一つとして，初期投資の大きさがあげられる．ソフトウェア・プロダクトラインでは，先にドメインの共通資産を開発し用意する必要がることから，そこに掛かる初期投資の費用が大きくなる．また，共通資産による製品の組み立てを前に，共通資産のスコープが外れていた場合のリスクについても考慮しなくてはならない．こうした問題が，ソフトウェア・プロダクトライン導入へのハードルを高めていると考えられる．

1.2 本研究の目的

本研究では，組み込みシステムの効率的な開発に有効とされる，ソフトウェア・プロダクトラインの基盤となる共有資産を，安全かつ効率的に構築するための手法を提案する．

類似するソフトウェアの少量かつ短期開発への要望は，組み込みシステムに限るものではなく，金融や法令など，一貫した知識体系を持つ大規模ドメイン向けのソフトウェア開発でも求められる課題でもある．本研究は，ドメインの種別を問わず，ソフトウェア・プロダクトラインの導入と構築の容易性を確保することについて，ソフトウェア工学の見地から考察し研究を行う．

1.3 アプローチ

ソフトウェアを新しく開発するにあたり，既存のレガシーシステムを参照し，活用するのが定石である．参照のレベルや利用の方法は様々であるが，ソフトウェア開発で必要となる情報を提供する一つの源泉として，レガシーシステムは重要な存在である．

既存のソフトウェアを再構築し，新たなソフトウェアを作り出すソフトウェア開発方法論に，ソフトウェア・リエンジニアリングがある．ソフトウェア・リエンジニアリングは，レガシーシステムのソースコードや仕様を，新しく構築するシステムで再利用することにより，効率的なソフトウェア開発を行うことを目的とする方法論である．レガシーシステムは，運用中に幾多の修正を受けた信頼性の高いシステムであることが多い．ソフトウェア・リエンジニアリングでは，レガシーシステムをソフトウェア開発における資産として，最大限の活用を行う．

本研究では，レガシーシステムを，ソフトウェア・リエンジニアリングの手法により，ソフトウェア・プロダクトラインの資産へと，再構築するアプローチを試し見る．

1.4 論文の構成

本論文の章の構成は以下である．

- 第 2 章 理論と研究計画
  ソフトウェア開発方法論であるソフトウェア・リエンジニアリングと，ソフトウェア・プロダクトラインについて，その理論的説明と研究計画を述べる．

- 第 3 章 ソフトウェア・リエンジニアリングによるソフトウェア・プロダクトラインの構築手法
  既存のレガシーシステムを，ソフトウェア・リエンジニアリングの手法により，ソフトウェア・プロダクトラインのコアアセットとして再構築する手法を提案する．

- 第 4 章 オーサリングシステムへの応用例
  本論文のアプローチに基づいて，既存のオーサリングシステムを題材に，リエンジニアリング手法によりプロダクトライン環境を構築する例を示す．

- 第 5 章 今後の課題
  上記の他に，レガシーシステムのリエンジニアリングによるプロダクトライン化で行うと良いと考える情報について述べる．

- 第 6 章 まとめ
  本研究についてまとめる．

第2章　理論と実施計画

2.1　ソフトウェア・プロダクトライン

ソフトウェア・プロダクトライン [1] は，特定のドメインを対象としたソフトウェアの開発方法論であり，コアアセットと呼ばれる共通のソフトウェア資産を組み合わせることで，新たなソフトウェア製品を作り出すアプローチを取る．ソフトウェア・プロダクトラインの目的は，ソフトウェア資産の集合からドメインに関連する一連のソフトウェア製品群を開発することで，類似する製品の生産性や品質を向上させビジネスでの競争優位を確立することにある．よって，ソフトウェア・プロダクトラインは，技術の新規性よりも，品質やコストに比重を置いた経営的な取り組みであると言うことができる．

プロダクトラインでは，ソフトウェア製品で必要となる個々の機能を資産化し，それらを組み合わせることで新たな製品の開発を行う．個々の資産とそこから生産する製品は，事前に定められたプロダクトラインのルール従い計画的に開発される必要がある．プロダクトラインの資産は，複数の製品で共通的に利用されることが前提であり，再利用が可能なものでなければならない．プロダクトラインにおける再利用とは，プロダクトラインの資産を単にライブラリとして利用するものではなく，計画の当初から再利用することを念頭に準備されるものである．また，既存のシステムを資産として再利用する場合も，単にコピーとして利用するのではなく，プロダクトラインでの資産として再利用を意図した計画をもって行う．

ソフトウェアの機能を資産として共有することで，製品間を横断する機能の再利用性を高めることが可能になる．だが，単純なコンポーネントベース開発とは異なり，ソフトウェア・プロダクトラインでは，計画の当初から特定ドメインに向けてのプロダクトライン資産としてコンポーネントを位置づけて開発を行う．プロダクトラインもコンポーネントをベースとする開発であるが，すべてのコンポーネントはプロダクトラインのアーキテクチャに従う必要がある．アーキテクチャは，製品ごとの差異による可変性を実現するための設計思想であり，アーキテクチャ自身もプロダクトラインの重要な資産のひとつとなる．プロダクトラインでは，アーキテクチャやコンポーネントなどの資産を活用することで，複数の製品を同時または平行して生産することになる．よって，プロダクトラインから生み出される製品は，全体で一つのものとして取り扱われる必要があり，個々の製品を超えた枠組みで捉える必要がある．それらの製品は，それぞれのリリースやバージョンも重要であるが，あくまで一つのプロダクトラインであり全体として管理されるべきものである．

2.1.1 ソフトウェア・プロダクトラインの用語

ソフトウェア・プロダクトラインに関する用語を説明する．

コアアセット ソフトウェア・プロダクトラインの基盤となる資産．代表的なコアアセットには，ソフトウェアを構成するためのアーキテクチャや機能を実装したコンポーネントがある．その他，各種の仕様書やスケジュール計画，予算や性能モデルなど，プロダクトラインの開発で再利用できるものはすべて含まれる．

開発 プロダクトラインのコアアセットを成果物として用意すること．または，コアアセットを組み合わせて製品を生産すること．プロダクトラインでは，自社あるいは外部機関でコアアセットを開発する他，既存の資産を利用する場合や市販品を購入する場合も開発として扱う．

ドメイン プロダクトラインが対象とする特定の専門領域．あるいは，それに関連する専門的な機能の集合．特定ドメインのソフトウェア・プロダクトラインは，そのドメイン機能の一部を包括するソフトウェアシステムということができる．

ソフトウェアプロダクトラインプラクティス コアアセットや製品の開発，管理をする上での戦略的な知識体系．

フィーチャ ソフトウェアが持つ特徴．プロダクトラインの機能やメンバーを区別するための単位として使われる．フィーチャを単位に，製品間の共通性と可変性の解析を行うことをフィーチャモデリングという．

2.1.2 ソフトウェア・プロダクトラインの活動

ソフトウェア・プロダクトラインの基本的な活動は，コアアセット開発，プロダクト開発，組織管理の３つからなる．

コアアセット開発 プロダクトラインの共通資産となるコアアセットを開発する活動．コアアセットには，アーキテクチャやコンポーネントをはじめ，各種のドキュメントや計画など有形無形のものが含まれる．ドメイン・エンジニアリングとも呼ばれる．

プロダクト開発 コアアセットを組み合わせて，実際の製品を生産する活動．プロダクトラインでは，複数の製品を平行して生産することが可能であり，コアアセットは製品間で共通に利用される資産となる．アプリケーション・エンジニアリングとも呼ばれる．

管理 コアアセット開発とプロダクト開発を結び付ける活動，両者の整合性を確保するための管理プロセスであり，両者の間にフィードバックループを設けて関係付けを行う．

2.1.3 ソフトウェア・プロダクトラインのプラクティスエリア

ソフトウェア・プロダクトラインでは，上記したプロダクトラインを実現するための３つの活動の他に，プロダクトラインを成功させるためのガイドラインとして２９のプラクティスを３つのカテゴリーに分類し定義している．

1. ソフトウェア・エンジニアリングのプラクティスエリア

   このプラクティスエリアは，コアアセットとプロダクトの開発と進化に，適切な技術を適用するための必要なプラクティスを提供する．

   アーキテクチャ定義 コアアセットとなるコンポーネントの外部から見える特性や，コンポーネントを関連付けるシステムの構造について定義を行う．また，コンポーネントのインタフェースをについても定義を行う．

   アーキテクチャ評価 定義されたアーキテクチャについて，セキュリティや信頼性，使い易さや変更容易性といった品質面を評価する．

   コンポーネント開発 プロダクトラインのコアアセットとして必要となるコンポーネントを作成する．

   COTS の利用 プロダクトラインで必要となるコンポーネントについて，市販製品（COTS：Commercial off-the-shelf）の購入と有効性を検討する．

   既存資産の発掘 既存のシステムから，プロダクトラインで再利用可能な資産を調達する．再利用資産としては，プログラムや仕様の他，ルールや計画といった有形無形のあらゆる資産が対象になる．

   要求エンジニアリング ステークホルダーからシステムについての意見を収集し，分析を行いシステムに対する要求として定義付けを行う．

   ソフトウェアシステム統合 コアアセットのコンポーネントを統合し，プロダクトとして完成させる．コンポーネントの結合は，インタフェースを通じて行うことで統合後の動作を保障する．

   試験 コアアセットとプロダクトについて試験を行う．また，それらの相互作用についても試験を行う．

   関連ドメインの理解 プロダクトラインが対象とするドメインについて，調査と学習を行い広く見識を深める．

2. プロジェクト管理のプラクティスエリア

   このプラクティスエリアは，プロダクトラインのプロジェクトを管理するためのプラクティスを提供する．

   構成管理 プロダクトの完全性を保障するため，複数のプロダクトで利用されるコンポーネントの変更やバージョンを管理する．

データの収集/メトリクス/追跡 プロダクトラインの運営に関する意思決定に必要となるデータを収集し分析する．また，収集したデータは継続して追跡調査を行う．

内製/購入/発掘/外注 必要となるソフトウェアについて，自社で開発するか，市販製品を購入するか，既存システムから流用するか，外部に開発を委託するかの選択を行う．コストやスケジュール，開発メンバーのスキルなどを考慮し決定する．

プロセス定義 プロダクトラインで必要となる作業と，そこに携わる各人の役割と責任を明確にし，プロセスとして定義する．

スコープ定義 システムが対象とする範囲を明確にし，システムの範囲内と範囲外の境界を定義する．

プロジェクト計画策定 プロジェクトを遂行するための計画と内容，および実行に必要な見積を行う．

プロジェクトリスク管理 プロジェクトに関するリスクを管理する．リスク管理上の責任者を取り決めて，リスクの識別や評価を行いリスクに対応する．

ツールによる支援 プロダクトライン活動を支援するツールを活用する．一般的に，プロダクトライン全体をカバーするには，多様なツールを活用する必要がある．

3. 組織管理のプラクティス

このプラクティスアリアは，プロダクトラインの取り組み全体を調和させるためのプラクティスを提供する．

ビジネスケース作成 ビジネスケースは，経営上の意思決定を助言するための事例を文書化するものである．ビジネスケースには，意思決定によるコストや効果，リスクなどの定量化手法や理論が含まれる．

顧客インタフェース管理 顧客との関係を管理する．顧客の視点やビジネス取引の基本原則を考慮して関係を維持する必要がある．

調達戦略策定 プロダクトラインで，外部から調達する部品がある場合の方法と計画について策定を行う．

資金調達 プロダクトラインとプロダクトの開発に必要な資金を確保する．プロダクトラインでは，ベースとなるコアアセットの開発など，立ち上げのための費用を調達する必要がある．また，立ち上げ以降も，プロダクトラインを維持するための資金が必要になる．

プロダクトラインの着手と制度化 プロダクトラインを導入する場合，技術面だけでなく組織や体制面での変革も必要となる．プロダクトラインの着手と制度化では，プロダクトラインへの技術革新を目的に，現状と望ましい状態との違いを

明確にし評価を行う．そして，両者の乖離を分析し，両者に橋をかける戦略的なソリューションを実行する取り組みを行う．

市場分析 プロダクトラインが市場において成功するための外的な要因について調査と分析を行う．

プロダクトライン運営 組織がプロダクトラインを運営する上での指針となるプロセスと方針を策定する．プロセスには，プロダクトの出荷と管理，組織内部の協業，調達戦略の他，プロダクトの出荷方法や生産能力など様々なものがある．

組織計画策定 個々のプロジェクトを超えた組織レベルでの計画を策定する．計画には，プロダクトライン採用計画，資金調達計画，構成管理，ツールによる支援，トレーニング，組織の編成，リスク管理などが含まれる．

組織リスク管理 個々のプロジェクトをまたがった組織レベルでのリスク管理を行う．リスク管理の核となる原則として，開かれたコミュニケーションがある．個々のリスク管理を接続するための原則として，統合管理，チームワーク，連続プロセスの３つがある．リスクを定義するための原則として，先見性のある見方，全体的見方，結果に対するビジョンの共有の３つがある．

組織編成 プロダクトラインの取り組みに必要な組織のグループ編成を策定する．プロダクトラインでは，コアアセットとプロダクトの２つの開発機構が組織構成を決定付ける．

技術予測 プロダクトラインの枠の中で，長期の成功が続く場合，市場や技術の変化に気づかない体質に陥る可能性がある．そのような慢心を防止するため，技術予測を行い，動向を見極めていく必要がある．

トレーニング トレーニングは，ソフトウェア開発の核となる活動のひとつである．ソフトウェアに関する管理面と技術面の向上を目的に，必要なスキルと知識の提供を行う．

2.2 ソフトウェア・リエンジニアリング

ソフトウェア・リエンジニアリングは，既存のシステムを再構築し，新しいソフトウェアを実現する開発方法論である [2]． システム環境の変化やドメインの改革などに対応するため，局所的な修正による最適化ではなく，システム全体を抜本的に見直し造り直しを行う．ソフトウェア・リエンジニアリングでは，現存するソフトウェアを分析調査し，仕組みや目的，技術といった要素を仕様として抽出する．次に，それらを材料に新たなソフトウェアとして再構築を行う．

似た言葉にビジネスプロセス・リエンジニアリングがある．ビジネスプロセス・リエンジニアリングは，企業が収益を大幅に向上させるため，組織の体制やビジネスのやり方を根本から改革する取り組みである．一般的に，ビジネスプロセス・リエンジニアリングの実施には，情報システムのソフトウェア・リエンジニアリングを伴うことが多い．

2.2.1 ソフトウェア・リエンジニアリングの用語

ソフトウェア・リエンジニアリングに関する用語を説明する．

ソフトウェア保守 ソフトウェアの提供後に，欠陥の修正や性能の改善などを目的として変更を施すこと．

フォワードエンジニアリング 実装に依存しない抽象的かつ論理的な設計から，システムの実装を行い開発すること．要求定義から製品出荷に至る，一般的なソフトウェア開発プロセスを指す．

リバースエンジニアリング 既存のシステムを分析し，構成要素や関連，振る舞いを抽象化された仕様として抽出すること．

リエンジニアリング 既存のシステムを再構築し，新しいソフトウェアを実現すること．対象システムの仕様をリバースエンジニアリングで抽出し，次にフォワードエンジニアリングで開発を行う．

再構築 リビルディング．保守性の問題や機能追加による大幅な変更を原因として，システム全体を構築し直すこと．再構築の方法として，旧システムを破棄して一から作り直すスクラップアンドビルド方式と，旧システムを資産として再利用するリエンジニアリング方式がある．スクラップアンドビルド方式でも，上流工程では旧システムの画面や帳票，ドキュメントなどを再利用することがある．

再構造化 リストラクチャリング．構造化されていないソフトウェアの振る舞いを変えることなく，構造化された実装に変換すること．COBOL や FORTRAN などで非構造形式に構築されたシステムを，オブジェクト指向言語などで再構築する場合に必要となる．

ドキュメント再構成 リドキュメンテーション．プログラムから，人間が理解しやすい形式のドキュメントを生成すること．生成されるドキュメントは，人間が理解しやすい形で表現することが求められる．

設計の復元 対象となるシステムの分析に，ドメイン知識や外部情報あるいは推論を加えて，設計上の意味を復元し明らかにすること．リバースエンジニアリングの一部として，システム分析の結果を意味付けするために行われる．

プログラム理解 対象のシステムに関する知識を獲得し，プログラムの修正，拡張，再利用，ドキュメンテーションといった活動が行える準備を行うこと．

2.2.2 ソフトウェア・リエンジニアリングのプロセス

ソフトウェア・リエンジニアリングのプロセスについて説明する．

1. 要求仕様

   再構築するシステムに課せられる要求を定義し文書化を行うプロセス．ソフトウェア・リエンジニアリングの場合，現行システムの問題点や，システムが新たにカバーするドメイン範囲，ビジネスプロセスのイノベーション対応など，幅広い分野から要求が提出される．要求仕様では，リエンジニアリング行うモチベーションと，新規システムに求められる要求を明確にし定義する．要求仕様は，次の工程を経て作成される．

   要求収集 システムのリエンジニアリングに係わるステークホルダーから，文書やヒアリングを通じて，新規システムへの要求を採集する．採集された要求は，ステークホルダーの立場に応じて多彩なものとなることが多い．

   要求分析 一般に，収集した要求は粒度，抽象度にムラがあり，一貫性が保障されていない．そうした要求を分析整理し，整合性と完全性を与え具体化を行う．

   要求定義 要求分析の結果を取り纏め，定義付ける．要求定義は，明確で簡潔，一貫性を備える必要がある．要求定義は，ステークホルダーとの要求の再確認を行うために利用されるとともに，設計以降の開発者へ要求を引き継ぐ目的で利用される．

   要求仕様記述 要求定義で明文化された要求に，ドメインやビジネス面での規則・制約を加え，設計以降のリエンジニアリング計画の骨子となる文書を作成する．要求仕様記述は，要求と設計の中間に位置する文書となる．

   要求管理 要求は，定義された後に修正または追加されることがある．これは，開発フェーズの進行によりステークホルダーの理解度が増していくことが原因である．よって，開発工程の全般を通して，要求の変更とその影響を管理調整していく必要がある．

   また，要求は，特性に応じて次の 3 種類に分類することができる．

   機能要求 ソフトウェアに求められる具体的な働き．

   非機能要求 ソフトウェアに求められる性能や制限，属性など機能以外の要求．機能外要求とも呼ばれる．

   制約条件 ソフトウェアの動作環境や開発言語など，ソフトウェアを制限する条件．

2. リバースエンジニアリング

再構築の対象となる既存のシステムを分析し，抽象的な仕様を探索するプロセス．レガシーシステムが，何を目的に，どのような構造になっているのかを調査し，新規システムで再利用可能な資産として分解する．リバースエンジニアリングの材料となる資産には，ソースコードをはじめ，要求仕様書，各種設計書，テスト資料などのドキュメント，レガシーシステムのデータベースに蓄積されたデータなど，多様な資産を活用できる．リバースエンジニアリングは，既存システムの再利用効率を上げることで，大幅なコストダウンも可能になることから重要なプロセスとなる．リバースエンジニアリングを行うにあたり，対象となる既存システムについての理解を獲得することが必要不可欠となる．次に，既存システムの解析手法を説明する．

テキスト解析 ソースコードをテキストと見做し，行数や文字数などからソフトウェアの大まかな規模を見積もる．

字句解析 ソースコードから，文字列を単語などの構文要素として識別する．識別した構文要素は，命令や変数などの字句として解析することで，プログラムの難易度を測る．

構文解析 ソースコードから，抽象構文木を作成する．抽象構文木を探索することにより，ノードが保持するプログラムの情報を取得する．

制御フロー解析 ソースコードから，プログラムの基本ブロックをノードとする制御フローグラフを作成する．制御フローグラフは，基本ブロック間の呼び出し関係を表すものであり，制御構造の複雑さを解析する．

データフロー解析 ソースコードでの，データの定義と参照の依存性を手掛かりに，値が参照されない変数や実行されないコードを特定する．

動的解析 プログラムを実行し，その振る舞いによって発生するモジュールの実行回数やメソッドの呼び出し状況を特定する．プログラムの実行時に，プロファイルと呼ばれるファイルに情報を出力し解析を行う．

プログラムスライシング プログラムから特定の視点により部位を切り出す分析方法．振る舞いによる切り出しや，特定の変数の参照をキーワードとした切り出しなど，解析者にとって有益な視点でスライシングを行う．

プログラム依存性解析 プログラムを構成する要素間の依存性を解析する．依存性の解析には，粒度により文レベル，プロシージャレベル，モジュールレベルの 3 種類のレベルがある．プログラム依存性解析により，プログラムの構造上の関係性を明らかにすることができる．

変数分類 ソースコードから，定義されている変数またはクラスを抽出し，分類別けや意味付けを行い整理する．変数分類を行うことにより，プログラムやドメイン構造の大局的な理解や，変数の用途などが得られる．

3. フォワードエンジニアリング

フォワードエンジニアリングは，リバースエンジニアリングで得られた再利用資産から，要求仕様を満たす実際のソフトウェアを構築するプロセスである．フォワードエンジニアリングの工程は，通常のソフトウェア開発と同じく設計，実装，検証から構成される．一般的なソフトウェア開発と異なる点は，いずれの工程もリバースエンジニアリングによって抽出された資産を再利用することにある．再利用素材には，抽象的なアーキテクチャから実装プログラムまで多様なレベルの素材が含まれる．フォワードエンジニアリングでは，再利用資産を有効活用することで，開発工数を大幅に削減できる可能性がある．また，再利用資産は，現行システムでの稼動実績があるため，高い信頼性を持つと期待できる．次に，フォワードエンジニアリングの各工程について説明を行う．

設計 設計工程は，現行システムから抽出した仕様をもとに，新システムに要求される仕様を満たすよう再設計を行う工程である．設計は，単に既存システムへの機能追加や変更といった局所的な最適ではなく，新規システムでの全体最適となる観点で行う．また，再利用する資産についても，設計に織り込んで計画を立てる．

実装 通常の実装と同じく，設計に従い実装を行うが，再利用資産として抽出された既存システムのプログラムやロジック，技術などを活用した実装を行う．実装工程では，レガシーシステムと新規システムの環境や言語の相違によって，実装工数が大きく変動する可能性がある．

検証 リエンジニアリングで新規にシステムを構築した場合，検証方法は二つの観点から行う必要がある．一つには，レガシーシステムをそのまま継承した機能について，レガシーシステムと同一の振る舞いであることを保障する必要がある．同一ではない場合は，レベルダウンまたはデグレードと呼ばれ，修正の対象となる．もう一つには，新規に追加または従来の機能を拡張・修正した場合の検証である．この場合，設計仕様に見合うテストケースを作成し，検証を行う必要がある．

4. 資産管理

リエンジニアリングで作成したシステムであっても，運用を開始した直後から保守の対象となる．保守プロセスを円滑に行うため，ソースコードはもちろんのこと，各種ドキュメントを資産として変更履歴などを十分に管理監督していく必要がある．

2.2.3 資産の再利用レベル

ソフトウェア・リエンジニアリングでは，レガシーシステムの資産を有効利用することが大きな課題となる．レガシーシステムの資産を利用するにあたり，再利用のレベルに応じて次の四つに分類し説明する．

システムレベルの再利用 レガシーシステムをそのまま再利用する方法．リエンジニアリング対象のシステム環境とレガシーシステムのシステム環境に互換性があり，レガシーシステム自体に変更を行う必要がない場合に有効である．レガシーシステム全体を再利用するため，リエンジニアリングのコストは低く抑えられる．単純コンバージョンまたはストレートコンバージョンとも呼ばれる．

プログラムレベルの再利用 レガシーシステムを，実装プログラムの単位で利用する方法．実装プログラムが，ダイナミックリンクライブラリなど実装レベルで再利用可能な形式である場合，新規システムから直接利用することが可能である．直接利用できない場合は，データファイルや中間プログラムなどを介して連携を行う．

コードレベルの再利用 レガシーシステムのプログラムをソースコードレベルで利用する方法．ソースコード内のロジックやデータ構造を，新規システムのソースコードに移植し再利用を行う．レガシーシステムと新規システムの開発言語が同じである場合，高い生産性で再利用が可能となる．

仕様レベルの再利用 レガシーシステムの仕様のみを利用する方法．新規システムでは，設計段階での利用となる．

2.2.4 再利用資産の分類

再利用資産は，もとは一つのシステムを構成する機能の集まりである．再利用を円滑に行うため，それぞれの資産を機能や用途に応じて分類し，利用方法を明確にする必要がある．利用方法には次の種類がある．

利用しない 新システムでは利用しない，または，利用される可能性がないもの．リエンジニアリングには取り込まずに廃棄する．

そのまま利用 レガシーシステムの資産を，そのまま利用するもの．修正を行わずに再利用できる資産が該当する．

改造して利用 レガシーシステムの資産を，改造して利用するもの．改造することにより利用が可能となる資産が該当する．

仕様のみ利用 レガシーシステムの資産から，論理仕様のみを抽出し利用するもの．

2.2.5 再利用資産の整理

再利用資産は，リエンジニアリングを行う上での素材として管理して行く必要がある．各素材を，利用目的や機能，特徴ごとに分類し，辞書化することで再利用の効率を高めることができる．

2.2.6 再利用資産の仕様抽出

再利用資産は，プログラムやドキュメントから仕様を抽出し，その目的を明確にすることで，再利用の容易性を高めることができる．

2.2.7 再利用資産の利用方法

実行プログラムやソースコードは，次のような方法で再利用行うことができる．

1. 実行プログラムの場合
    - 実行プログラム API の直接利用
    - データファイルを経由しての利用
    - リモートプロシジャコールを経由しての利用
    - ブリッジプログラムを経由しての利用
2. ソースコードの場合
    - モジュール単位ですべてを利用
    - ソースコードの一部を利用
    - ラッパークラスの利用
    - アダプタークラスの利用

2.3 対応付け

ソフトウェア・プロダクトラインのプラクティスである既存資産の発掘と，ソフトウェア・リエンジニアリングを対応付ける．

2.3.1 既存資産の発掘とソフトウェア・リエンジニアリング

既存システムを再利用してコアアセットを抽出することを目的に，既存資産の発掘によって得られる資産と，ソフトウェア・リエンジニアリングによる再構築手法を対応付ける．

第3章　ソフトウェア・リエンジニアリングによるソフトウェア・プロダクトラインの構築手法

レガシーシステムをリエンジニアリングして，ソフトウェアプロダクトライン環境を構築する手法を提案する．

3.1　リエンジニアリング要求仕様

リエンジニアリング要求仕様は，レガシーシステムをリエンジニアリングすることにより，新しいソフトウェアを構築することについての要求を明らかにする工程である．主な要求として，レガシーシステムの問題点や不満点，再構築する新規システムへの要望，プロダクトラインで実現する製品系列についての要望などが考えられる．リエンジニアリング要求仕様を行うについて，ソフトウェア・プロダクトラインで定義されているプラクティスと関連付けを行い，解釈を与える．

要求エンジニアリング レガシーシステムの特性を踏まえ，リエンジニアリングにより実現する新システムに関する要求を明確に定義する．

本論文では，要求の分析手法としてゴール指向要求分析法を採用する．ゴール指向要求分析法は，要求の最終目標をゴールとして定め，ゴールを達成するために成すべきことをグラフとして詳細化していく手法である．レガシーシステムでの経験を踏まえ，新しく構築するソフトウェアに課される要求をゴールとし，その達成方法について分析を行うものとする．

3.2　レガシーシステムのリバースエンジニアリング

リバースエンジニアリングは，レガシーシステムを分析し，プロダクトラインの資源として再利用する準備を行う工程である．レガシーシステムから再利用する資源としては，実行プログラムやソースコード，ロジック，技術，仕様書など幅広い分野の成果物が対象となる．リバースエンジニアリングを行うについて，ソフトウェア・プロダクトラインで定義されているプラクティスと関連付けを行い，解釈を与える．

アーキテクチャ定義 レガシーシステムの外部から見える特性とコンポーネント間の関連を調査し，レガシーシステムのアーキテクチャを明らかにする．また，コンポーネントの振る舞いとインタフェースについても調査を行う．

アーキテクチャ評価 調査したアーキテクチャについて，プロダクトラインのコアアセットとして転用した場合の，共通性と可変性の実現性について評価を行う．

コンポーネント開発 レガシーシステムで利用されているコンポーネントとモジュールについて調査し，再利用の可能性を検討する．

COTS の利用 レガシーシステムで COTS が利用されている場合，プロダクトラインでの利用価値と代替手段について検討を行う．

既存資産の発掘 レガシーシステムの資産の内，プロダクトラインで再利用可能な資源を探索する．

ツールによる支援 リバースエンジニアリングを支援するツールを選定し活用する．

スコープ定義 ドメインに対して，レガシーシステムが対応する範囲を調査し，システムがカバーする領域を明らかにする．

関連ドメインの理解 リバースエンジニアリング作業を円滑に行えるよう，ドメインについての見識を深める．

本論文では，次に記述するプログラム理解と設計の復元の 2 段階の工程を経て，リバースエンジニアリングを行うことを提案する．

3.2.1 プログラム理解

プログラム理解は，レガシーシステムのプログラムや振る舞いから，システムに関する理解を獲得する工程である．プログラム理解については，リバースエンジニアリングツールを利用するのが効率的である．昨今では，多くの UML ツールがリバースエンジニアリングの機能をサポートしている．リバースエンジニアリングツールは，ソースコードを解析し，クラス図を生成するものが一般的である．製品によっては，MDA モデルを生成するツールも存在する．ツールの活用と合わせて，各種ドキュメントの参照や，レガシーシステム開発者へのヒアリングなどの活動を行うと効果的である．

ソースコード解析 − クラス図の抽出

クラス図は，リバースエンジニアリングツールを利用することで，容易に作成することができる．一般的な商用 UML ツールでは，複数のソースコードを，一括してクラス図へ変換する機能が備わっている．UML 図は直感性に優れており，このようなツールを活用することで，プログラム理解に必要となる工数を削減することが出来る．

プログラムスライシング − シーケンス図の作成

作成したクラス図から，シーケンス図を作成することでプログラムスライシングを行う．シーケンス図の作成に先立ち，レガシーシステムで実現している主要な操作をユースケースとして用意する． 次に，UML ツールを利用して，ユースケースからシーケンス図を作成することで，ユースケースを切り口としたプログラムスライシングのモデルを作成する．シーケンス図の作成には，先に作成したクラス図を利用できるため，効率的なプログラムスライシングが期待できる．

プログラム依存性解析 − コラボレーション図の作成

UML のコラボレーション図を利用して，プログラム依存性解析を行う．一般的な UML ツールは，シーケンス図からコラボレーション図への変換機能をサポートしている．シーケンス図から変換したコラボレーション図により，プログラム (クラス，モジュール) 間の依存性を確認することができる．

変数 (クラス) 分類 − クラスのパッケージ化

UML 図上で各クラスをパッケージ化して分類を行う．パッケージは，オブジェクト指向言語などで採用されているネームスペースに相当するものであり，クラスのグループ分けに利用できる．一般的な UML ツールでは，クラス図のパッケージ化がサポートされている．プログラムスライシングとプログラム依存性解析の結果をもとに，クラスの機能や依存性，特徴などによりパッケージ化することで分類を行う．

3.2.2 設計の復元

設計の復元は，プログラム理解で得られた成果物と知識から，レガシーシステムの仕様や意図を推測し，設計を復元する工程である．本研究では，フィーチャを単位とした仕様の復元を提案する．プログラム理解で得られた UML 図と知識をもとに，システムの仕様をフィーチャとして復元し定義を行う．フィーチャとフィーチャ同士の関係は，FODA[3] や PLUS[4] などで提案されている図式言語で記述する．

また，フィーチャ同士の関連性から，レガシーシステムで採用されているソフトウェア・アーキテクチャを抽出し復元する．復元したフィーチャとソフトウェア・アーキテクチャは，ソフトウェア・プロダクトラインで再利用される重要な資産となる．

3.3 プロダクトライン要求仕様

プロダクトライン要求仕様は，構築するソフトウェア・プロダクトライン環境についての要求を定義する工程である．主な要求として，プロダクトラインで生産可能な製品の種

類や機能，スコープの範囲などがあげられる．プロダクトラインリング要求仕様を行うについて，ソフトウェア・プロダクトラインで定義されているプラクティスと関連付けを行い，解釈を与える．

要求エンジニアリング レガシーシステムの分析結果を踏まえ，プロダクトラインで生産する製品群に関する要求を明確に定義する．

本論文では，フィーチャを単位としてプロダクトライン環境の要求分析を行うことを提案する．リバースエンジニアリングで作成したフィーチャグラフを再利用して，新しく求められる機能をフィーチャとして追加し，製品間の共通性と可変性についての情報を付与することで，プロダクトラインの構成を定義する．共通性は，すべての製品で必要となる必須の機能であり，製品間を横断する機能であることを示す性質である．可変性は，製品ごとに異なる機能であり，製品間の差異となる機能であることを示す性質である．FODA などのフィーチャ図式言語は，フィーチャに共通性と可変性の表現を行うことが可能であり利用価値が高い．

3.4 コアアセットへのフォワードエンジニアリング

フォワードエンジニアリングは，リバースエンジニアリングで得られた分析結果をもとに，レガシーシステムの資産をプロダクトラインのコアアセットとして再構築する工程である．ここでは，リバースエンジニアリングで得られた資産と成果物を有効に活用することで，コアアセットの効率的な構築を目標とする．フォワードエンジニアリングを行うことについて，ソフトウェア・プロダクトラインで定義されているプラクティスと関連付けを行い，解釈を与える．

アーキテクチャ定義 レガシーシステムのアーキテクチャを参考に，プロダクトラインで実現するプロダクトのアーキテクチャを策定する．アーキテクチャには，フィーチャの共通性と可変性を組み込み，製品系列に対応できる構造にする．

アーキテクチャ評価 プロダクトラインで採用するアーキテクチャが，プロダクト開発で想定する系列やオプションに対応可能かを評価する．

コンポーネント開発 個々のフィーチャを，プロダクトラインのコアアセットとしてコンポーネント化する．

COTS の利用 必要となるコンポーネントについて，COTS の導入を検討する．

ツールによる支援 プロダクトラインを支援する開発環境やソースコード管理ソフト，情報共有ソフトなどを選定する．

スコープ定義 ドメインの内，プロダクトラインがカバーする領域を明確にし定義する．

ソフトウェアシステム統合 開発したコアアセットコンポーネントが，一つの製品として統合され動作することを保障する．

内製/購入/発掘/外注 開発が必要となるコンポーネントについて，自社開発するか，外部委託するか，市販製品を購入するかを決定する．

プロセス定義 プロダクトライン環境の構築と運営に関するプロセスを定義し，メンバーの役割と責任を明確にする．

組織編成 プロダクトラインで必要となる組織を編成する．

本論文では，次に記述するアーキテクチャ設計とコンポーネント設計の 2 段階の工程を経て，フォワードエンジニアリングを行うことを提案する．アーキテクチャとコンポーネントは，プロダクトラインのコアアセットとなるものであり，両者を組み合わせて製品を生産することになる．

3.4.1 アーキテクチャ設計

アーキテクチャ設計は，プロダクトライン要求仕様で定義されたフィーチャグラフから，現実の製品を生産可能なプロダクトラインアーキテクチャを設計する工程である．プロダクトラインアーキテクチャは，リバースエンジニアリングで抽出したレガシーシステムのアーキテクチャを参考として設計を行う．レガシーシステムのアーキテクチャに，プロダクトライン要求仕様で定義された共通性と可変性を加味した拡張を行うことで，プロダクトラインアーキテクチャとして再構成する．

共通性と可変性の導入

プロダクトラインアーキテクチャは，プロダクトラインで生産を可能とする系列製品のスーパーセットとなるアーキテクチャである．よって，プロダクトラインアーキテクチャは，プロダクトラインで生産対象となる機能や特徴の異なる複数の製品を包括できるよう，可変性に対応できる設計である必要がある．可変性の実現については，Strategy パターンなどのデザインパターンの適用が有効である．

コンポーネントインタフェースの定義

フィーチャをコンポーネントとして，そのインタフェースを定義を行う．特に可変性のあるフィーチャは，製品ややオプションにより組み換えが行われるコンポーネントとなるため，インタフェースの設計は重要なものとなる．インタフェースの設計については，コンポーネントの交換容易性とともに，交換のタイミングも考慮する必要がある．主な交換のタイミングには，開発時，インストール時，実行時がある．コンポーネントインタフェースは，特に問題がない限りフィーチャの単位で定義を行う．

3.4.2 コンポーネント設計

コンポーネント設計は，コンポーネントインタフェースで定めたコンポーネントについて，振る舞いと機能を設計する工程である．ここでは，リバースエンジニアリングで得られた再利用資産の仕様や実装を，プロダクトラインコンポーネントへ転用し再利用することで，設計の効率化を目指す手法を取る．設計するコンポーネントは，個々の独立性と機能性はもちろんのこと，先に定めたプロダクトラインアーキテクチャに適合する必要があり，プロダクトラインアーキテクチャの制約の範囲で有効性が発揮されるものでなくてはならない．

3.4.3 プロダクトライン実装

プロダクトライン実装は，コンポーネント設計に基づき，コンポーネントの実装を行う工程である．ここで実装したコンポーネントは，プロダクトラインのコアアセットを構成する実行プログラムの一群であり，プロダクトラインアーキテクチャに従い，現実のソフトウェア製品の一部として出荷されることになる．実装コンポーネントは，アーキテクチャへの準拠はもちろん，将来製品でも再利用可能なことが求められる．プロダクトとしてソフトウェアシステム統合した場合の挙動も保証される必要がある．

第4章　オーサリングシステムへの応用例

既存のオーサリングシステムを例題に，リエンジニアリング手法により，レガシーシステムをプロダクトライン化する設計を行い，本論文の有効性を検証する．オーサリングシステムは，映像，音声，字幕などの素材から，商用の DVD タイトルに見られるような映像ディスクを作成するシステムである．プロユースのオーサリングシステムでは，映像構成の他，メニューの作成機能や不正コピーを防止するセキュリティ機能，PC 上での再生シミュレーション機能などが備わっている．

オーサリングシステムは，一般的なアプリケーションソフトの形態をもつソフトウェアであるが，ハードウェアである再生装置に準拠した実装が求められる．このことから，オーサリングシステムは，アプリケーションソフトと組み込みシステムの中庸にあるソフトウェアということができる．

4.1　リエンジニアリング要求仕様

レガシーシステムを再構築し新しいソフトウェアを構築するにあたり，そのモチベーションとなる要求を収集分析し，要求仕様として定義付けを行った．

4.1.1　要求収集

一般に，リエンジニアリング要求は，レガシーシステムに対する不満や不足，不具合といった問題から発生することが多い．要求収集を行った結果，今回，顧客が新しいシステムを必要とする主要な動機は，レガシーシステムの操作性に起因するものであった．その他，コストや拡張性など，新システムに付随する多彩な要求が提出された．

操作性の向上 レガシーシステムは，プロユースの製品であり，メディアで規定されている仕様がすべて実現可能なアプリケーションである．しかし，メディア規格の実現を視点とした設計がなされているため，ユーザビリティが極めて悪く，規格に精通した技術者しか利用することができない．レガシーシステムは，スーパーユーザー向けの製品として価値があるため存続させるが，一般のオーサーが利用できる優れた操作性のソフトウェアをラインナップに加えたい．

低コスト 新システムは，機能的にはレガシーシステムのサブセットの位置づけになる．よって，レガシーシステムを利用して，極力，コストを押さえ短期間で開発したい．

高品質 新システムは，レガシーシステムと同じくプロユースの業務用ソフトとして販売する．新システムについても，実績あるレガシーシステムと同等に高品質である必要がある．

拡張性 メディアは次世代記録媒体の一種であり，規格の拡張が予定されている．規格については，業界団体で日々研究と議論が行われており，複数の拡張計画が同時進行している．新システムは，将来のメディア規格の拡張に追従できるよう，拡張性に優れたものである必要がある．

機能のオプション化 新システムでは，機能をオプション化し，顧客の必要に応じた組み合わせで販売出来るようにしたい．

4.1.2 要求分析

収集された要求は，すべて非機能要求に分類される要求であった．レガシーシステムは，現状で考えられる必要な機能はすべて網羅しており，機能面での満足度は高い．新システムについては，レガシーシステムの機能を引き継ぐことが前提であるため，機能的な新しい要求が発生しなかった為である．要求の分析手法として，ゴール指向要求分析法の一つである i*を採用し，ゴール木を作成した（図 4.1）．

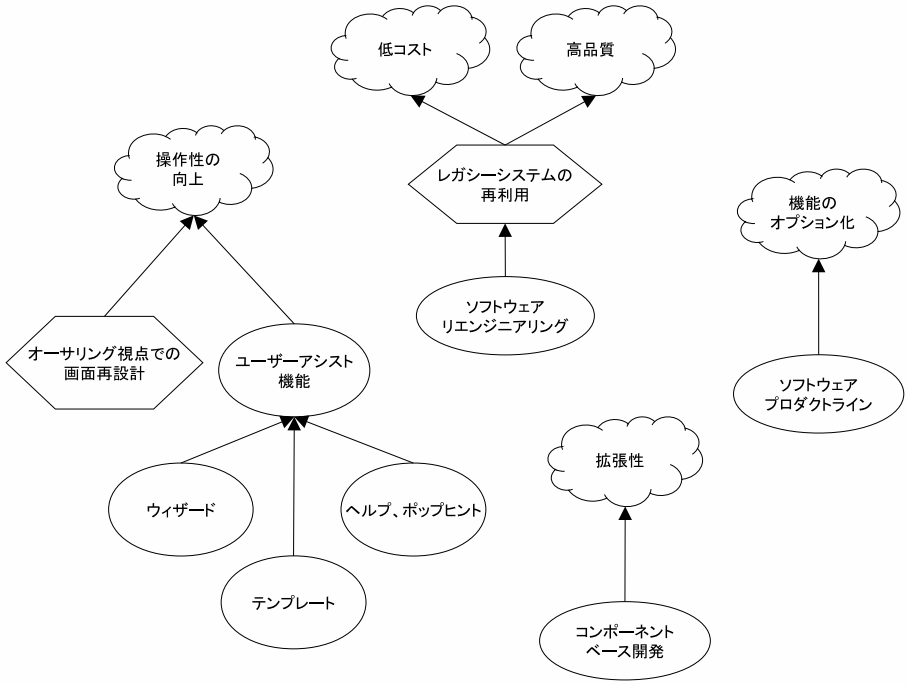

図 4.1: ゴール木

操作性の向上 新システムの画面を，オーサリングの視点で再設計することにより操作性を向上する．また，ユーザー補助機能を追加することでユーザーの負担を軽減する．

低コスト 既存のレガシーシステムのプログラムや仕様を再利用することで，開発コストを抑える．レガシーシステムの再利用については，ソフトウェア開発方法論であるソフトウェアリエンジニアリングの導入が有効である．

高品質 実績あるレガシーシステムのプログラムや仕様を再利用することで，一定の品質を確保する．レガシーシステムの再利用については，ソフトウェア開発方法論であるソフトウェアリエンジニアリングの導入が有効である．

拡張性 個々の機能をコンポーネントとして開発し独立性を高めることで，拡張性を確保する．

機能のオプション化 新システムの開発環境をソフトウェアプロダクトライン化し，個々の機能をコアアセットとすることで，機能のオプション化に対応する．

4.1.3 要求定義

上記の分析結果をもとに，以下の通り要求の定義付けを行った．

- ユーザビリィティー向上のため，新システムの画面はオーサリングの視点で再設計を行う．また，ユーザー補助機能を追加する．[1]
- コストと品質を両立させるため，新システムはレガシーシステムをリエンジニアリングすることにより開発する．
- 拡張性とオプション性を備えるため，新システムは機能をコンポーネント化し，ソフトウェアプロダクトライン環境を構築する．

以上のことから，レガシーシステムの機能をコンポーネントとしてコアアセット化し，ソフトウェアプロダクトライン環境を構築する手法が適切であるとの結論を得た（図 4.2）．

4.2 リバースエンジニアリング

レガシーシステムを再利用する上で，まずはレガシーシステムを解析し理解する必要がある．レガシーシステムは，.NET 環境の Windows アプリケーションとして構築されている．比較的，新しい環境で構築されているが，開発言語は.NET 初期のバージョンで記述されている．

[1]画面のユーザビリィティーについては，本論文の趣旨と異なる問題であるため，これ以上の言及は行わない．

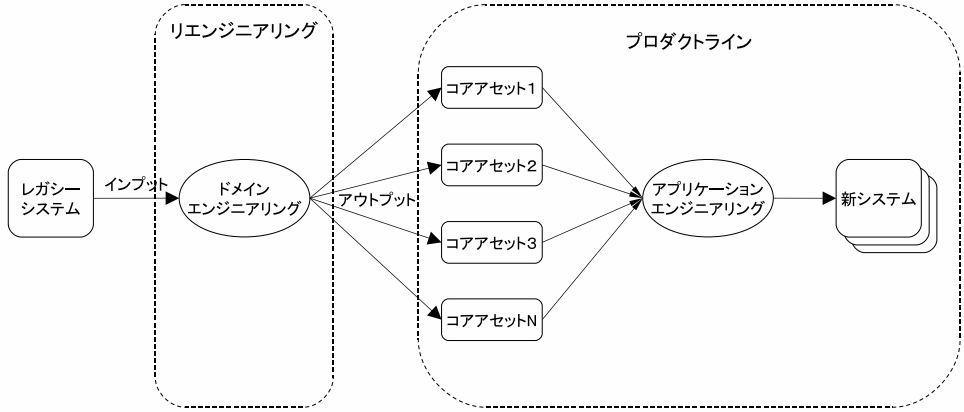

図 4.2: 新システムの開発方法

4.2.1 プログラム理解

プログラム理解は，レガシーシステムのプログラムに関する知識を獲得することを目的に行う．作業にあたって，ツールとして.NET 統合開発環境と市販の UML ソフトを利用して行うものとした．

ソースコード解析

ソースコードの解析は，UML ツールのリバースエンジニアリング機能を利用して行った．リバースエンジニアリング機能は，対象となるソースコードを解析し，クラス図を自動的に生成する機能である．また，クラスの種類や属性，操作などもクラス図の情報として出力する．ソースコードをクラス図として図式化することにより，システムの理解容易性を高めることができる．

今回のレガシーシステムでは，ツールを活用することで，約 500 のクラスをクラス図として簡便に生成することができた．ここで作成したクラス図は，これ以降のフェーズで再利用される重要な資産となる．

プログラムスライシング

プログラムスライシングは，UML のシーケンス図を作成することで行った．レガシーシステムの代表的な操作をユースケースとして作成し，その操作によるクラス間のシーケンスを調査することで，ユースケースを切り口としたプログラムスライシングのモデルを作成することができる．クラスとシーケンスの追跡には，統合開発環境のデバッグ機能を利用した．デバッガーのステップ実行や呼び出し履歴の機能を利用し，操作によるクラスの遷移やメッセージを追跡することで，ユースケースのシーケンスを確認することができる．シーケンス図のオブジェクトは，ソースコード解析で作成したクラス図を利用することができるため，効率的にシーケンス図を作成することができた（図 4.3）．

作成したシーケンス図から，レガシーシステムのシーケンスには次の特徴があることがわかった．

- 画面上の操作は，GUI コンポーネントからオーサリング操作クラスにメッセージとして通知される
- オーサリング操作クラスは，メッセージの種類に応じてライブラリクラスをコールし処理を依頼する
- ライブラリクラスは，処理に応じてエンティティのデータを更新する

この作業により，レガシーシステムのシーケンスは，GUI コンポーネントクラス，オーサリング操作クラス，ライブラリクラスの 3 つのクラス間で遷移することがわかった．

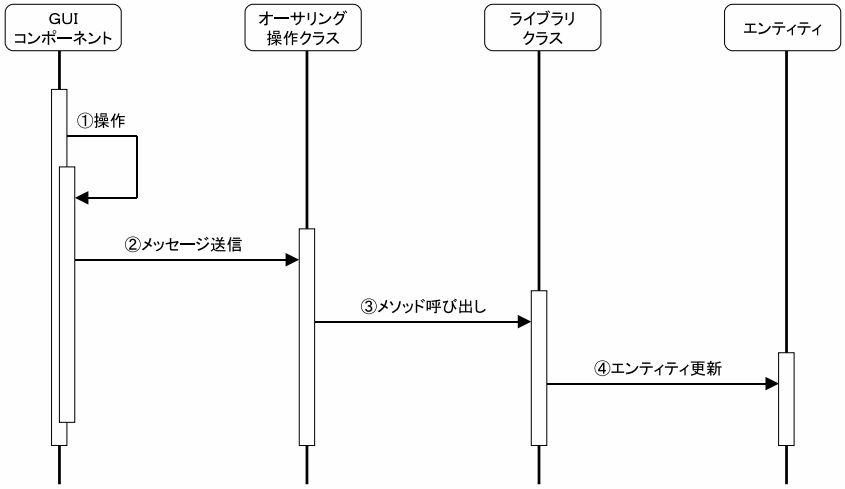
GUI
コンポーネント
オーサリング
操作クラス
ライブラリ
クラス
エンティティ
①操作
②メッセージ送信
③メソッド呼び出し
④エンティティ更新

図 4.3: シーケンス図

プログラム依存性解析

プログラム依存性解析は，UML ツールのコラボレーション図を利用することで行うことができる．コラボレーション図は，クラス間のメッセージや関連を相互作用として表現する UML 図である．コラボレーション図を利用することで，クラス間の依存関係を明らかにすることができる．UML ツールでは，シーケンス図を変換してコラボレーション図を作成することができる．今回のプログラム依存性解析では，プログラムスライシングで作成したシーケンス図を変換し作成したコラボレーション図を利用した（図 4.4）．

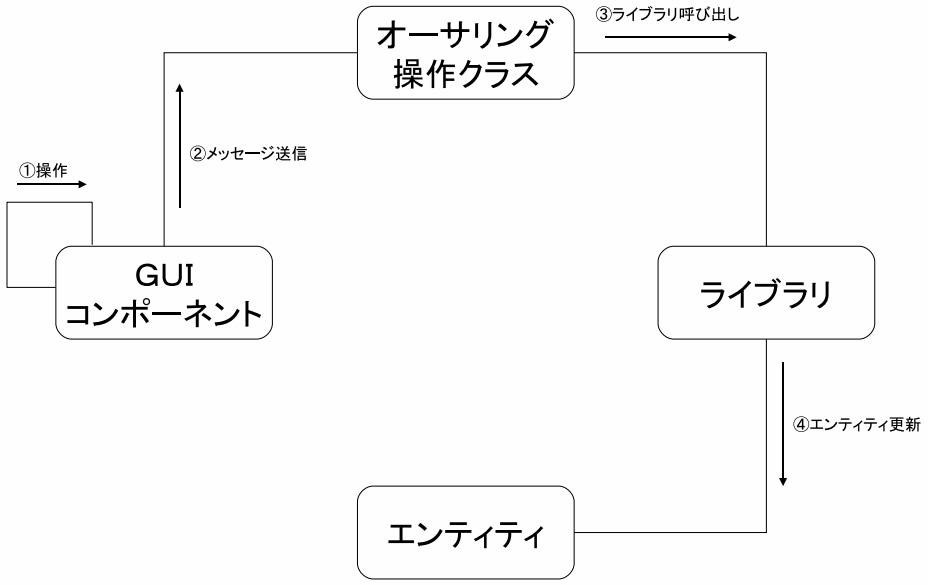

図 4.4: コラボレーション図

コラボレーション図を分析した結果，レガシーシステムの依存性には次の特徴があることがわかった．

- GUI コンポーネントクラスでの操作は，オーサリング操作クラスの操作メソッドに依存する．
- オーサリング操作クラスは，操作に応じた操作メソッドが実装されおり，操作メソッドはライブラリに依存する．
- ライブラリクラスは独立性が高く，新システムでの再利用性が高い．

分類	説明
オーサリング操作	オーサリングの操作とプロジェクト管理に関するクラス
共通処理	エラー処理やインターフェースなどの共通的クラス
データベース	保存処理に関するクラス
エンジン	メニューやタイトル遷移などのエンジンとなるクラス
環境設定	オーサリングシステムの環境設定に関するクラス
外部ツール	オーサリングシステムの外部ツールで利用されるクラス
ライブラリ	オーサリング全般で利用されるライブラリクラス
オブジェクト	システムで扱うオブジェクトの定義クラス
リソース	システムで利用する文字列や画像などのリソース
UI	画面上の UI クラス
XML	データ定義用の XML スキームと処理に関するクラス

表 4.1: クラス分類

クラス分類

分析対象のソースコードがオブジェクト指向言語である場合，ソースコード中のネームスペースをクラス分類に利用することができる．ネームスペースは，クラスを分類する論理ブロックである．ネームスペースは入れ子構造の階層化にすることができ，それぞれの階層に属するクラスを定義する．一般的に，クラスは役割や目的に応じてネームスペースにより階層構造化されいいるため，クラス分類の指標とすることができる．今回リバースエンジニアリングで利用した UML ツールは，ネームスペースをサポートしており，クラス図の作成時に，自動的にネームスペースによるパッケージとしてクラス図を分類する機能をサポートしていた．生成したクラス図は，ネームスペースでパッケージされた状態で生成さるため，効率的にクラス分類を行うことができた（表 4.1）．

4.2.2 設計の復元

設計の復元は，プログラム理解で得られた知識と成果物をもとに，レガシーシステムの仕様を抽象化し，設計の意図を復元することを目的に行う．

フィーチャ分析

レガシーシステムの仕様を抽象化するにあたり，レガシーシステムの機能や特徴をフィーチャとして分析しフィーチャグラフを作成した．フィーチャグラフのインプットとして，プログラム理解のソースコード解析で作成したクラス図とクラス分類の結果を主体に，実行プログラムの振る舞いや画面構成などを参考とした．フィーチャは，フォワードエンジ

ニアリング時の再利用資産として活用することを目的とするため，可変性は考慮せずにすべてのフィーチャを洗い出しFODAフィーチャグラフとして記述した（図4.5）.

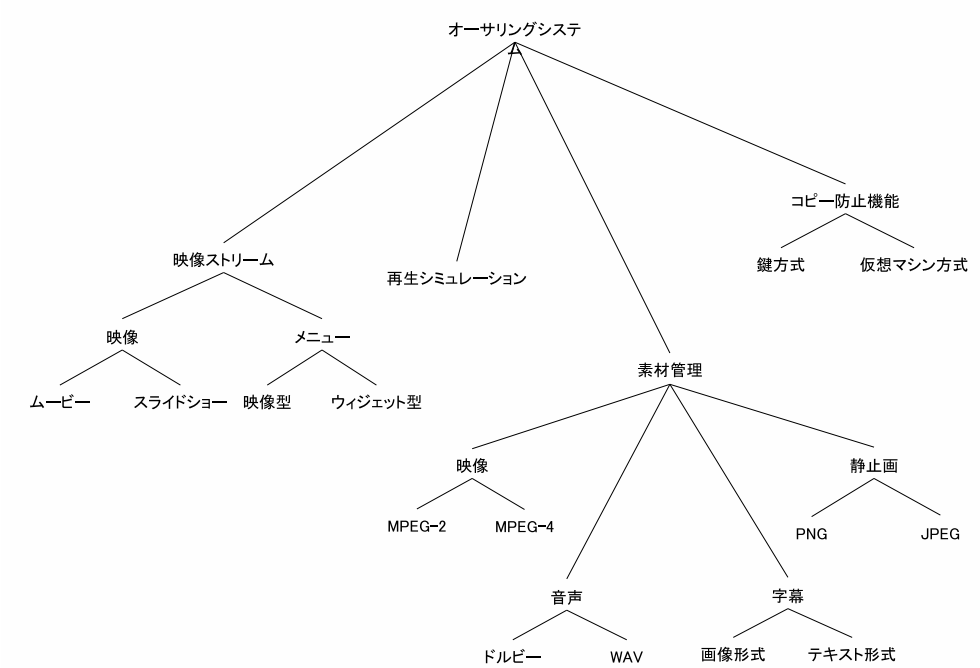

図 4.5: レガシーシステムのフィーチャグラフ

アーキテクチャ分析

リバースエンジニアリングで作成した各UML図と，フィーチャ分析で作成したフィーチャグラフをもとに，レガシーシステムのソフトウェア・アーキテクチャについて分析を行い取り纏めた．レガシーシステムは，品質面の管理は良好であったため，アーキテクチャについても比較的綺麗にレイヤー構造化されており，プロダクトラインでの再利用性が高いことがわかった（図4.6）.

- GUI層は，ベースとなるメインウィンドウと複数のGUIコンポーネントで構成されている.
- GUIコンポーネントは，操作に応じてビジネスロジック層のオーサリング操作クラスへメッセージを送信する.
- ビジネス層は，単一のオーサリング操作クラスから成り，GUIでの操作単位に対応する操作メソッドが用意されている.

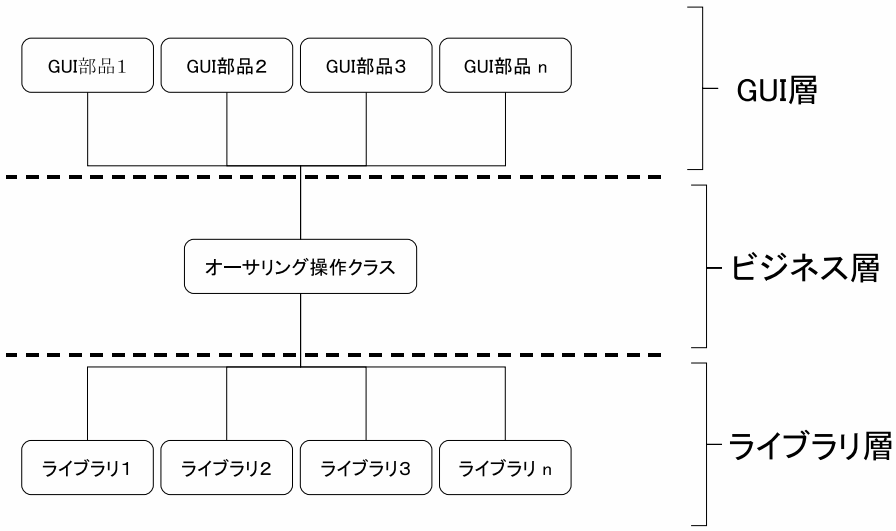

図 4.6: レガシーシステムのアーキテクチャ構造

- 操作メソッドは，GUI コンポーネントでの操作に最適化されたビジネスロジックが実装されており，処理に応じてライブラリ層の機能をコールする．
- ライブラリ層は，機能ごとに分類された複数の DLL からなる．各ライブラリは独立性が高く再利用性についても良好である．

アーキテクチャの分析結果から，レガシーシステムのアーキテクチャは優れており，プロダクトラインのコアアセットとして再利用可能であるとの結論を得た．

4.3 プロダクトライン要求仕様

プロダクトライン環境の構築にあたり，生産可能な製品と備えるべき能力について要求を収集し分析を行った．

4.3.1 要求収集

要求収集では，プロダクトラインから生産するファミリー製品の製品像と，新たに必要となる機能について，次の要求が提出された．

- ユーザー補助機能として，ウィザードとテンプレートを新たに追加する
- オーサリングが成立する最小限の構成を基本パッケージとする
- それ以外の機能は，すべてオプションとする

4.3.2 フィーチャモデリング

フィーチャモデリングは，リバースエンジニアリングで作成したレガシーシステムのフィーチャグラフに，要求収集で得られた要求を加味する形で行うものとした．レガシーシステムのフィーチャグラフに，新たに要求された機能をフィーチャとして追加し，既存のフィーチャと合わせて共通性と可変性の属性を与えることで分析と定義を行った．ここでは，新規フィーチャとしてユーザー補助機能を追加し，アプリケーションの最小構成となるフィーチャは共通性を付与し，オプションとなるフィーチャには可変性を付与することで，フィーチャグラフを作成した（図 4.7）．図式言語である FODA フィーチャグラフを再利用することで，要求の分析と定義を効率的に行うことができた．

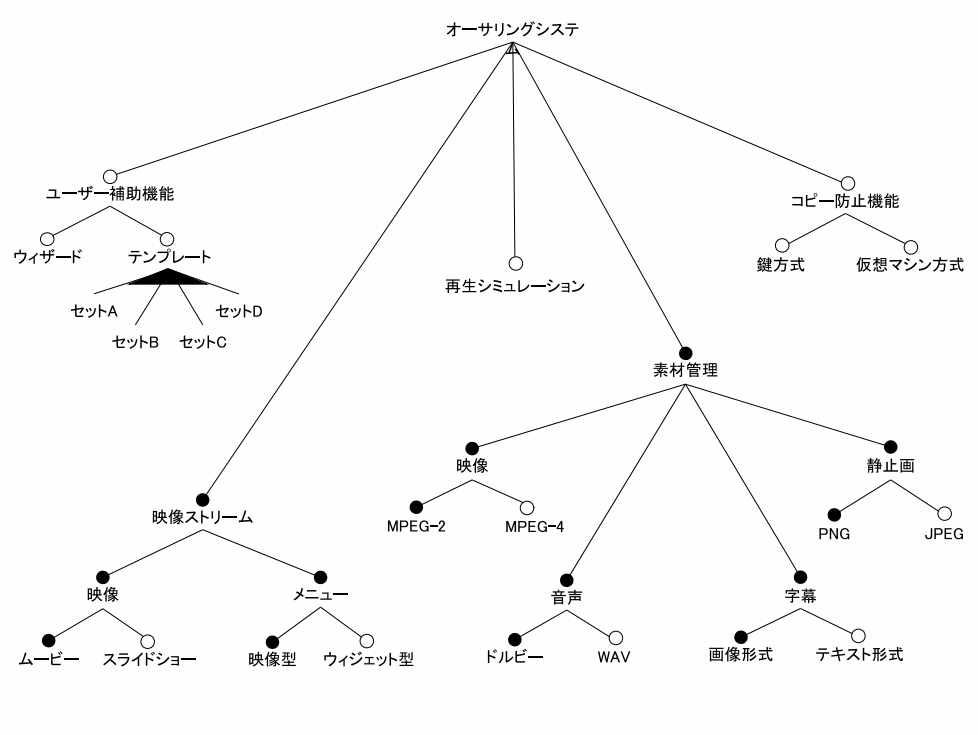

図 4.7: プロダクトラインのフィーチャグラフ

4.4　フォワードエンジニアリング

これまでのリバースエンジニアリングで得られた理解と成果物もとに，レガシーシステムをコアアセットとして再構築し，プロダクトライン環境を構築する設計を行う．プロダクトラインから導出する新しいシステムは，レガシーシステムと同じく.NET 環境でのWindows アプリケーションとして構築を行う．.NET については，最新のバージョンに対応するものとし，開発環境も合わせて最新のものを使用することとした．

4.4.1　アーキテクチャ設計

プロダクトラインアーキテクチャは，レガシーシステムのアーキテクチャを拡張し，プロダクトライン要求仕様で定義された可変性を導入することで対応するものとした．フィーチャの可変性は，デザインパターンの一つである Strategy パターンを適用することで実現した（図 4.8）．Strategy パターンは，複数の処理を戦略として用意し，状況に応じて戦略の変更と追加を柔軟に行うことができるデザインパターンである．Strategy パターンの特性を利用し，可変性のあるフィーチャを戦略としてクラス化し，プロダクトに応じてフィーチャを選択できる構造にした．

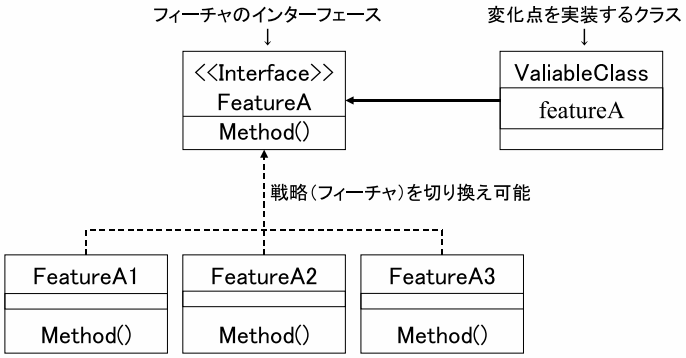

図 4.8: 可変性の実現方法

可変性は，アーキテクチャのビジネス層で実現するものとし，ライブラリ層は独立性が高いためレガシーシステムを踏襲する形態を取ることにした（図 4.9）．ビジネス層では，変化点となるクラスが多数設置されるため，GUI 層からのメッセージを受け取る窓口として Facade クラスを新設し，通信経路を集約した．

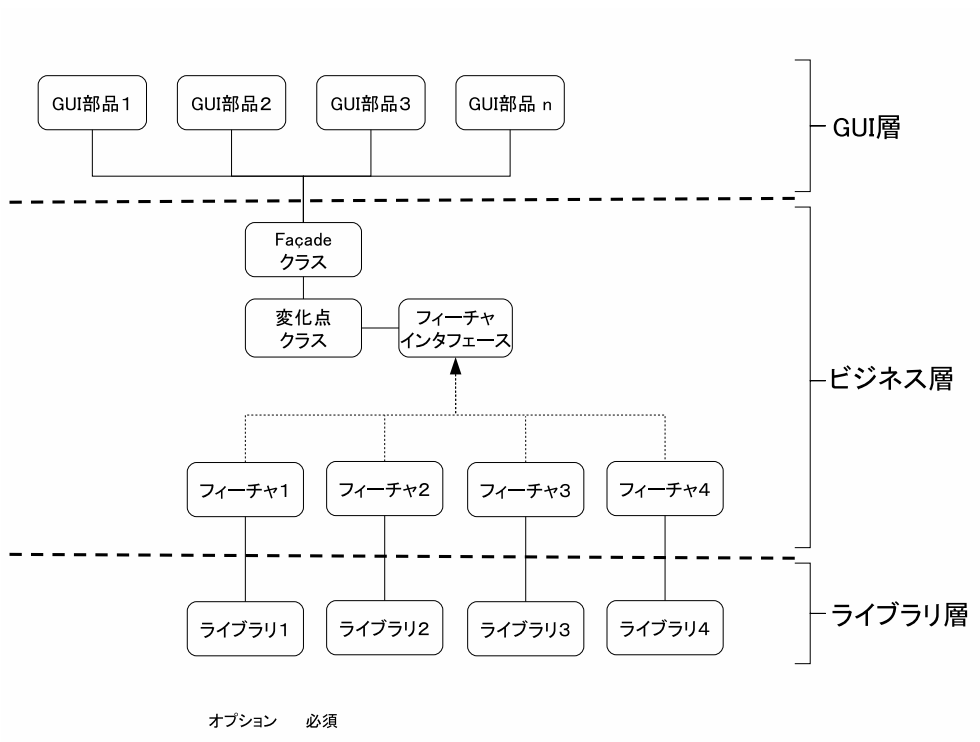

図 4.9: プロダクトラインのアーキテクチャ

4.4.2 コンポーネント設計

コンポーネントは，レガシーシステムのオーサリング操作クラスを再利用し，Strategy デザインパターンへリファクタリングすることで作成するものとした．レガシーシステムのオーサリング操作クラスは，GUI 層からの操作メッセージに対応するメソッドをすべて持つ巨大なクラスになっている．オーサリング操作クラスを Strategy パターン化するために，オーサリング操作クラスのメソッドをフィーチャ単位にクラス化し，グループごとにインターフェースを定めるリファクタリングを行うことでコンポーネントを導出した．また，フィーチャの変化点となるクラスを新設し可変性の実現に対応するものとした（図 4.10）．

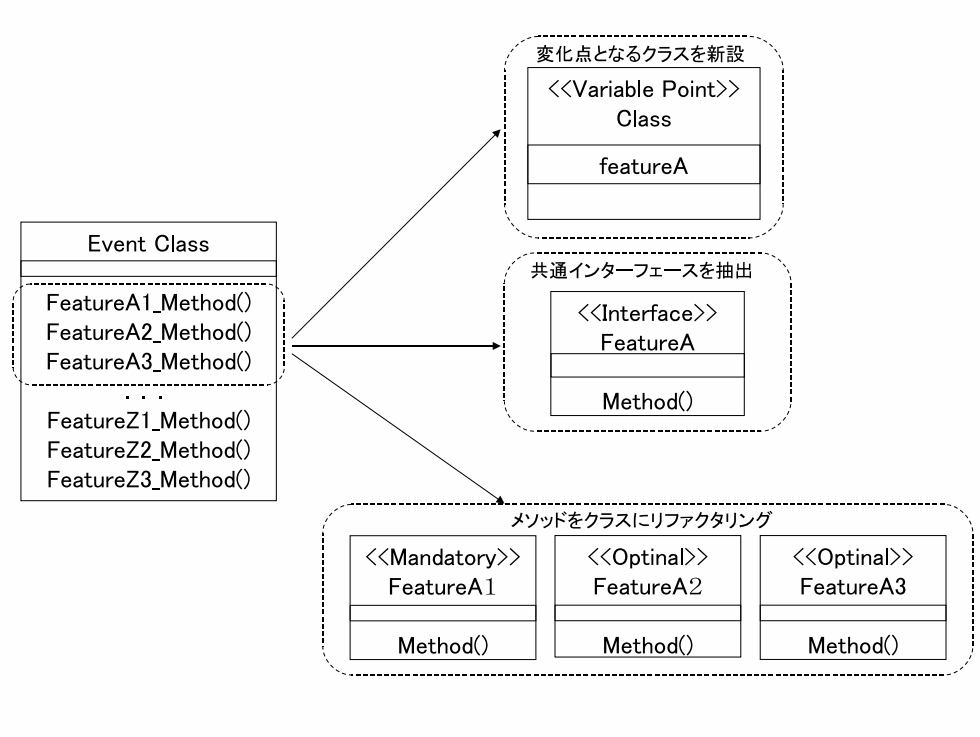

図 4.10: コンポーネントの導出

4.5 結果

本応用例では，レガシーシステムからアーキテクチャとフィーチャを抽出し，プロダクトラインで必要となる変更と拡張を行い，プロダクトラインのコアアセットとして再構築する設計を行った．レガシーシステムとプロダクトラインで実現する新規システムは，同じドメインを対象としているため，製品として必要となる機能の多くは重複したものである．レガシーシステムのアーキテクチャとビジネス層のコンポーネントを可変性に対応できる構造に拡張することで，プロダクトラインのコアアセットへ転用することができた．また，レガシーシステムのライブラリ群は，実行プログラムとして利用したことから，修正点を最小限に留めることができた．以上のことから，レガシーシステムをリエンジニアリング手法で再構築することにより，プロダクトライン環境を低コストで開発できることを確認した．

第5章　今後の課題

本研究の結果，次の課題が残された．

レガシーシステムとプロダクトラインの技術的乖離よる活用資産の変化 レガシーシステムと，プロダクトラインでの動作環境や言語などの乖離によって，再利用可能な資産の範囲が大きく変動することが考えられる．例えば，メインフレームからWEBシステムへリエンジニアリングする場合，再利用可能な資産の割合は低くなる可能性がある．環境や開発言語が近い場合は，比較的高い再利用率を予測できる．両者の技術的相違を見極め，計算し，再構築での再利用計画を立案する必要がある．

レガシーシステムとプロダクトラインの並存 プロダクトラインの稼動後にも，レガシーシステムが現役で運用される場合，両者の運用方法について対策を講じる必要がある．将来発生する変更要求が，両者に共通である場合と，一方にのみ必要である場合が想定され，保守を複雑にする可能性がある．対策として，レガシーシステムをプロダクトラインに組み込む，両者を完全に分離して扱う，などの方法が考えられるが，両者の技術的差異や将来の変更可能性を考慮して，構成管理を行う必要がある．

コスト見積もり プロダクトラインの構築に際して，レガシーシステムのリエンジニアリングで低減できるコストの見積もり方法を確立する必要がある．本研究は，レガシーシステムを再利用することで，プロダクトラインの構築コストを低減することを提案した．しかし，再利用の度合いや，採用するプロダクトラインのアーキテクチャによって，低減できるコストの幅が大きく変わる可能性がある．

第6章　まとめ

本研究は，レガシーシステムをリエンジニアリングすることで再利用し，ソフトウェア・プロダクトラインのコアアセットを構築することを課題とした．これにより，効率的かつ安全にソフトウェア・プロダクトライン環境を導入することを目指した．

レガシーシステムの分析には，リバースエンジニアリングの手法を用いた．また，その分析結果からコアアセットの導出に，フォワードエンジニアリングの手法を用いた．

リバースエンジニアリングによりレガシーシステムから仕様を抽出すること，及び，抽出した仕様をフォワードエンジニアリングしてコアアセットを作成することは，ソフトウェア・プロダクトライン環境の構築において有効であることを示した．

謝辞

最後に，本研究を行うにあたって，終始御指導頂きました落水 浩一郎教授に心より深く感謝申し上げます．本研究の副指導教員として，有意義な御指導，御助言を賜りました鈴木 正人准教授に心より深く感謝申し上げます．また，本研究の審査員として，本研究に関する貴重な御意見を賜りました小川 瑞史教授ならびに青木 利晃准教授に心より深く感謝申し上げます．みなさま，本当にありがとうございました．

参考文献

[1] Paul Clements, Linda Northrop, (前田 卓雄 訳), ユビキタスネットワーク時代のソフトウェアビジネス戦略と実践 ソフトウェア・プロダクトライン, 日刊工業新聞社, 2003.

[2] 上原 三八, Wei-Tek Tsai, 佐野 隆, 馬場 一弥, 保守とリエンジニアリング, 共立出版, 2000.

[3] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, 1990.

[4] Hassan Gomaa, Designing Software Product Line with UML, Pearson Education, 2005.