大規模受託開発へのZIPC適用とモデルベース開発

株式会社日立情報制御ソリューションズ　業務サポート本部
生産技術部　ソフトウェアエンジニアリンググループ　技師
渡辺　滋

1．はじめに

組込みソフトウェアの需要は急激に増大しつつあり、それに伴ってソフトウェア開発規模も増大してきています。一方、それに比べ、組込みソフトウェア技術者の数は圧倒的に不足しており、この状況が組込みソフトウェア技術者の負担増大や安易な外部委託となり、組込みソフトウェアに関するトラブル発生の増大を招き、品質低下につながる一因になっています。

経済産業省が情報処理推進機構（IPA：Information-technology Promotion Agency, Japan）に委託して行った2008年組込みソフトウェア実態調査によれば、組込みソフトウェアの開発規模は約3.5兆円、組込みソフトウェア技術者は約24.2万人となっています。そして調査時点で約8.8万人の組込みソフトウェア技術者が不足しています。

組込みソフトウェアの開発現場では、組込みソフトウェア技術者の人材不足を補うため、海外を含めた外部委託に頼らざるをえないのが実情です。

2．大規模受託開発について

現在、組込みシステム関連ビジネスは、当社にとって大きな柱となっており、組込みソフトウェアの大規模受託開発や自社製の組込みシステム製品開発で順調に業績を伸ばしてきています。一方、実際の組込みソフトウェアの開発現場では、ハードウェアのテスト不足による不具合発生の切り分けなど、組込みシステム特有の課題が数多く発生し、大変な苦労を重ねながら、これらを克服しています。

特に組込みソフトウェアの大規模受託開発（事例として、カーオーディオソフト開発があります）（図１、図２）においては、組込みシステム特有の問題以外に大規模ソフトウェア開発という、もう１つの大きな課題を抱えています。その中で品質を確保し、コスト、期間を厳守しながら、従来の小規模ソフトウェア開発から大規模ソフトウェア開発へと開発スタイルの転換が要求されています。今後は、それらに対応する手段として、新しい開発手法や開発プロセスの導入を検討し、当社に合った開発スタイルを築いていく必要があると認識しています。

以下に当社の現状の開発スタイルについて述べ、今後、長期的な視点で目指すべき開発スタイルについて提案させていただきます。

3．「ZIPC」適用の開発スタイル

CASE（ Computer Aided Software Engineering）ツール「ZIPC」を適用した開発とは、「拡張階層化状態遷移表（EHSTM：Extended Hierarchy State Transition Matrix）設計手法」を導入したソフトウェア開発です。一般に組込みシステムは、リアルタイム性を要求されるシステムであり、組込みソフトウェアはそのシステム要件を満足しなければなりません。

ご存知の通り「ZIPC」適用のメリット（効果）は、一言で言えば、リアルタイム性に必要な「いつ」「どこで」「何をする」かを考え、設計から実装（コード生成）、テストまでの正常系、異常系ケースのモレ・ヌケを防止できる点です。

そこで、１つの視点として重要視しているのが、「事象（event）」、「状態（state）」、「アクション（action）」、「遷移（transition）」に着目したソフトウェア設計です。「事象」とは外部からの刺激であり、「状態」は事象に応じたアクションを起動するタイミングを与えるものです。「アクション」は処理の最小単位であり、「遷移」は、ある状態から別な状態へ移ることです。「ZIPC」では、これらを「状態遷移表（STM：State Transition Matrix）」の構成要素として扱っています。

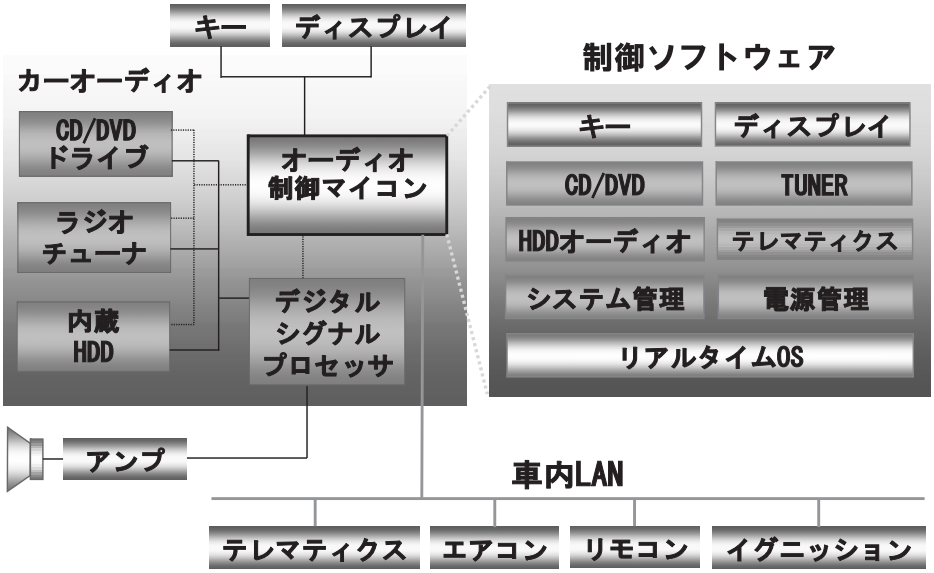


図1　カーオーディオシステムとソフトウェア構成

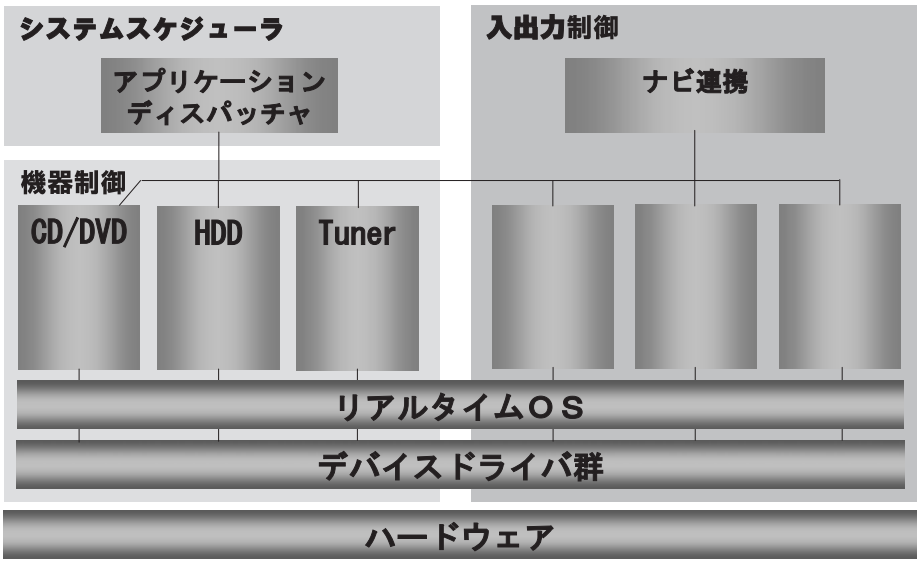


図2　カーオーディオシステムのソフトウェア構成詳細

当社では、「ZIPC」をより有効に適用すべく、独自に規定した設計ドキュメントを活用し、構造化手法を基本としたソフトウェア開発を進めています。この規定は、当社のこれまでの開発経験により導入している設計規準であり、状態遷移表の作成に不慣れな技術者のためや、「ZIPC」の「STM」の設計品質を向上させるために大変役立っています。また、「ZIPC」は状態遷移の視点なので、それ以外の設計の視点を補うという意味でも極めて有効と考えています。その設計規準の内容について、以下に簡単に紹介します（図3、図4）。

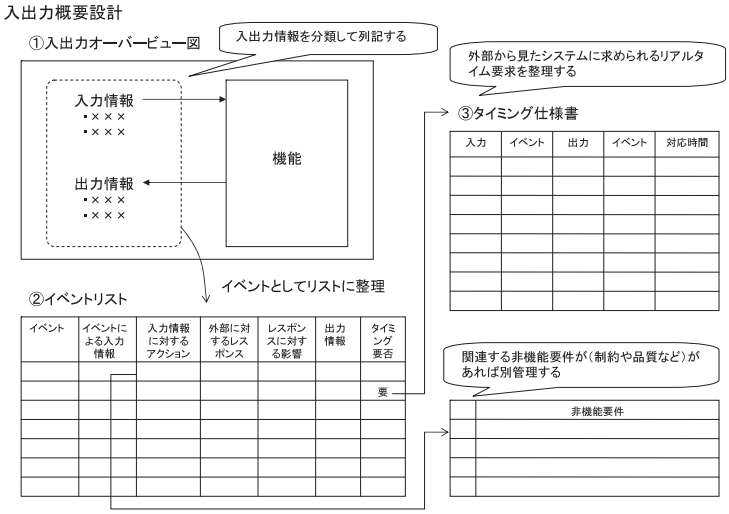


図 3　構造化手法設計規準（1）

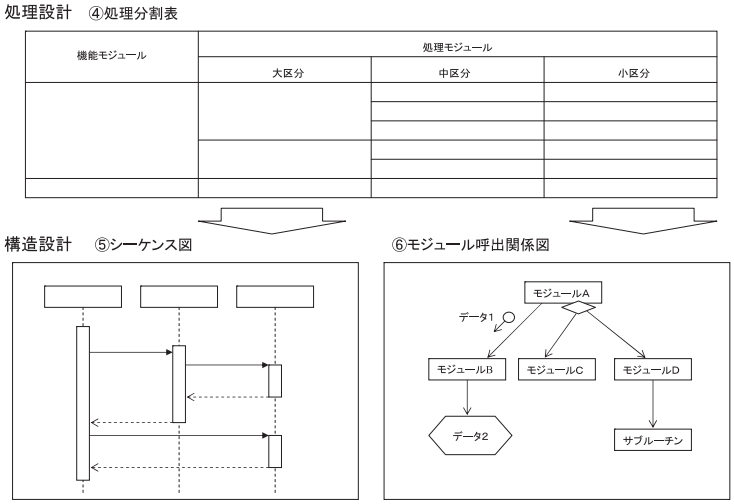


図 4　構造化手法設計規準（2）

まず概要設計フェーズにてシステムの外部要求を整理するため、「入出力オーバービュー図」を作成します。次にシステム全体のイベントを分析するため、「イベントリスト」を作成し、必要に応じて「タイミング仕様書」を作成していきます。

次の処理設計フェーズにて全体のソフトウェア・アーキテクチャを確定させるため、「処理分割表」を作成します。さらに構造設計フェーズにて「モジュール呼出関係図」、「シーケンス図」を作成し、それをもとにして「入出力オーバービュー図」、「イベントリスト」の入力毎にシステムの機能が実現できるかどうかの見通しを確認していきます。

そして実装設計フェーズにて「シーケンス図」をもとに「ZIPC」の「STM」を作成し、実装フェーズにて「ZIPC」でC言語プログラムのソースコードを生成していきます。当社では、このような手順を踏むことにより、「ZIPC」をより有効に活用し、ソフトウェアの品質を向上させています。

4．オブジェクト指向手法ベースの「ZIPC++」適用

今後の大規模開発においてソフトウェア資産の再利用を考える場合、１つの選択肢として「ZIPC」から「ZIPC++」への移行があります。では、どのようにして「ZIPC++」をソフトウェア開発に適用していけばよいのでしょうか。そのキーポイントは、以降で述べるモデルベース開発にあります。

従来の構造化手法ベースの「ZIPC」に対して、オブジェクト指向手法をベースとした「ZIPC++」を適用して開発することのメリット（効果）について考えてみます。そのためには、まずオブジェクト指向手法を導入する目的について、その本質を考え、理解する必要があります。では、その目的とは何でしょうか。一言で言えば、ソフトウェア資産の再利用性への対応です。大規模開発では、ソフトウェア資産の再利用が開発効率を大きく左右します。

オブジェクト指向手法について、よく聞くのは、現実世界のものを、そのままオブジェクト（クラス）として捉えてソフトウェアの構成要素にすることで、属人性を排除し、改造や機能追加などの仕様変更に強いソフトウェアを構築できるということです。

オブジェクト指向手法については、これまでのやり方ではできなかったことが、誰でも簡単にできそうなキャッチ・フレーズがたくさんあります。ところが、1980年代から始まった手法にも拘らず、未だにソフトウェア開発手法として、それほど爆発的に普及しているようには見えません。それは組込みソフトウェアの分野でも同様であり、開発手法としては構造化分析・設計や構造化プログラミング言語（C言語）が主流になっています。

確かにオブジェクト指向分析・設計は、組込みソフトウェアを開発する上で有効な手法ですが、誰でも簡単にできるものではありません。むしろ、構造化手法に比べ難易度が高く、抽象化やパターン化というハイレベルなスキルが要求され、そのスキルを身に付けるには、能力、知識、経験（即ち時間）が必要になります。

5．モデルベース開発

5．1　モデリングの視点と開発フェーズ

モデリングとは、視点を決めて対象を抽象化することにより、複雑さを解消し単純化することです。抽象化とは、ある視点における本質を見出し、それ以外のものを捨て去ることです。ここにモデリングの大きなリスクがあります。それは、一度モデリングを誤ると、それ以降の軌道修正ができないということです。言い方を変えると、一度捨て去ったものは、振出しに戻る以外、途中で元に戻すことができないということです。

そのため、モデルを考える上では、異なる複数の視点でモデリングしていくことにより、ソフトウェアの設計品質を作り込んでいくことが必要になります。

異なる複数の視点については、「4+1」ビューモデル [1)] と呼ばれるソフトウェア・アーキテクチャを設計する上でのガイドラインがあります。まず「4」つのビューとは、「論理ビュー」（機能構造の視点）、「プロセスビュー」（プロセス構造の視点）、「実装ビュー」（構成管理の視点）、「物理ビュー」（物理配置の視点）です。そして「+1」ビューは、「ユースケースビュー」（ユーザ要求の視点）です（図５）。

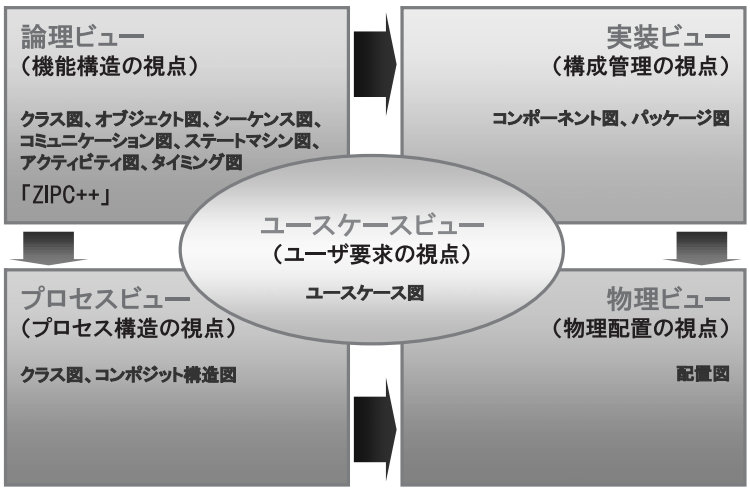


図５　「４＋１」ビューモデルとUML

ここで「ZIPC」と「ZIPC++」が持っている状態遷移表「STM」は、この中の「論理ビュー」に相当します。これらの複数の視点でモデリングし、それを設計ドキュメントに表現するためには、表現機能を豊富に持つモデリング言語が必要になります。

その条件を満たすモデリング言語の１つにUML（Unified Modeling Language）があります。UMLの具体的な表現機能としては、「論理ビュー」に対して、クラス図、オブジェクト図、シーケンス図、コミュニケーション図、ステートマシン図、アクテビティ図、タイミング図、「プロセスビュー」に対してクラス図、コンポジット構造図、「実装ビュー」に対して、コンポーネント図、パッケージ図、「物理ビュー」に対して、配置図、「ユースケースビュー」に対して、ユースケース図（アクテビティ図）があります。

モデルベース開発では、複数の視点に立ち、要求定義フェーズにおける「要求モデリング」、システム分析フェーズにおける「分析モデリング」、システム設計フェーズにおける「設計モデリング」、実装フェーズにおける「実装モデリング」を行っていきます。そして、「ZIPC++」と共にUMLを使用することで、開発の上流フェーズから下流フェーズまでモデルによって繋げることができます。

５．２　再利用性への対応

次にオブジェクト指向手法を導入する最大の目的であるソフトウェア資産の再利用について述べます。オブジェクト指向の概念で、それ以前には無かった考え方に「カプセル化」と「インタフェース」があります。「カプセル化」とは、簡単に言うと、「属性」と呼ぶ情報（データ）を「操作」と呼ぶ処理に閉じ込めてしまうということです。

目的は、プログラム処理間の結合度（coupling）を低くするためです。そして、類似処理を分散させずに１つのプログラム処理の中にまとめることで、プログラム処理内の凝集度（cohesion）を高めます。このような考え方でソフトウェアの部品を設計していきます。これがクラスの概念です。

更に、これらの部品を組合せることで、一つ一つの機能を実現させる単位を構成していきます。この構成の中で現在から将来にかけて共通的に使用できるものを抽出し、独立させながらカプセル化を行います。これがコンポーネントになります。コンポーネント間は、結合と分離ができるような疎結合の仕掛けを使用して設計していきます。これがUMLの「インタフェース」の概念です。UMLで規定している「インタフェース」の概念は、一般的に使われるインタフェースとは異なるものなので、注意が必要です。

この「カプセル化」したクラスの組合せを「インタフェース」で結合するという方式が最大のキーポイントであり、一般的には「コンポーネントベース開発」と呼ばれるソフトウェア開発の考え方です（図６）。そして、すべてのオブジェクト指向プログラミング言語は、「カプセル化」と「インタフェース」を実装する言語仕様を持っています。

「コンポーネントベース開発」により、コンポーネント単位のソフトウェアの再利用が可能になります。したがって、今後、ソフトウェア資産の再利用を実現させるためには、「ZIPC」から「ZIPC++」への移行を行い、UMLモデリングの中で１つの「論理ビュー」モデルとして併用しながら、モデルベース開発を実施していくことが必要であると認識しています。ただし、本格的なソフトウェア資産の再利用を実現するには、デザインパターンの活用や次に述べるソフトウェア・プロダクトライン（SPL：Software Product Line）[2] の導入が必要になります。

６．ソフトウェア・プロダクトラインの導入

組込み製品の生産性向上を考える上で、ソフトウェア資産の再利用がキーポイントになることは前述した通りです。再利用を更に効果的に行うための手段の１つとしてソフトウェア・プロダクトラインがあります。ソフトウェア・プロダクトラインは、複数の開発製品で共通利用するアーキテクチャやコンポーネントの基盤を構築し、その基盤を再利用した一連のプロダクト生産の中で個々の開発製品を扱うソフトウェアの開発方式です。

この開発方式を実現させる基盤として、モデルベース開発が必要になります。以下にソフト

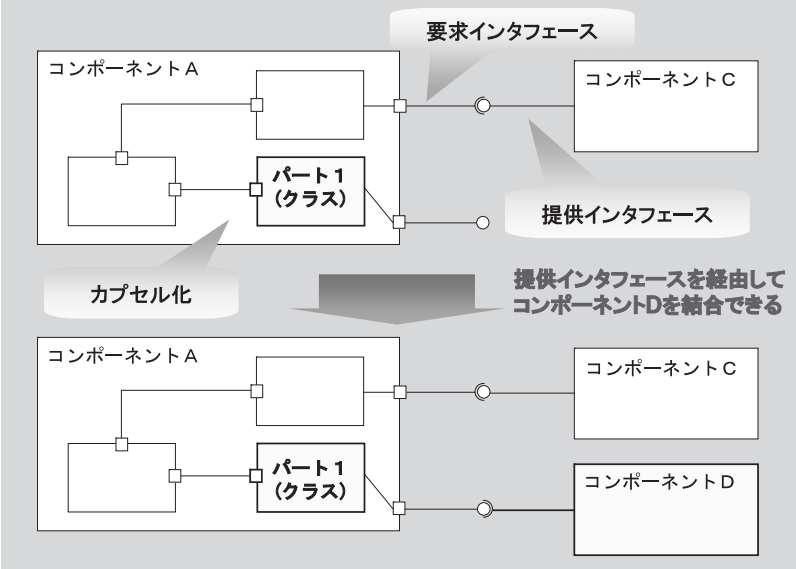


図6　カプセル化とインタフェース

ウェア・プロダクトラインの開発プロセスと、なぜモデルベース開発が必要なのかについて述べます。

6．1　ソフトウェア・プロダクトラインの開発プロセス

ソフトウェア・プロダクトラインの開発プロセスでは、以下のような作業を行います。まず開発プロセスを、大きく2つのプロセスに分けて開発を進めます。1つは「ドメイン・エンジニアリング」のプロセス、もう1つは「アプリケーション・エンジニアリング」のプロセスです（図7、図8）。

「ドメイン・エンジニアリング」のプロセスは、「プロダクトマネジメント」、「ドメイン要求開発」、「ドメイン設計」、「ドメイン製作」、「ドメインテスト」の5つのサブプロセスから成ります。「プロダクトマネジメント」では、プロダクト・ポートフォリオ（提供する製品タイプ群）の開発計画、即ち製品ロードマップを決めます。「ドメイン要求開発」では、製品ロードマップから提供製品に対するユーザの要求項目（フィーチャ）を定義し、その中で製品毎に「可変な要求項目（可変性）」と製品群に「共通な要求項目（共通性）」に分類してフィーチャ・モデル（または、バリアビリティ・モデルとも呼びます）を設計していきます。ここで、ドメインとは、ビジネス要求に起因した一連の概念や用語で特徴付けられるプロセスや知識の領域です。次の「ドメイン設計」では、フィーチャ・モデルから可変部分の機能と共通部分の機能を分けてコンポーネント化し、ソフトウェア・アーキテクチャ（これを「リファレンス・アーキテクチャ」と呼びます）を設計します。「ドメイン製作」では、「リファレンス・アーキテクチャ」をもとにコンポーネントモデルを実装します。「ドメインテスト」では、テスト戦略にもとづいたテストシナリオを作成し、共通部分の機能を中心としたテストを実施していきます。

「アプリケーション・エンジニアリング」のプロセスは、「アプリケーション要求開発」、「アプリケーション設計」、「アプリケーション製作」、「アプリケーションテスト」の4つのサブプロセスから成ります。「アプリケーション要求開発」では、「ドメイン・エンジニアリング」のプロセスで開発したフィーチャ・モデルに対して、アプリケーションの要求仕様（可変仕様と共通仕様から成る）との要求項目の差分分析を行い、その差分に対応できるかどうかを判断します。「アプリケーション製作」以降では、要求項目の差分分析の結果をもとに「ドメイン・エンジニアリング」のプロセスの成果物を再利用して、アプリケーションの設計、製作を行います。また「アプリケーションテスト」では、「ドメイン

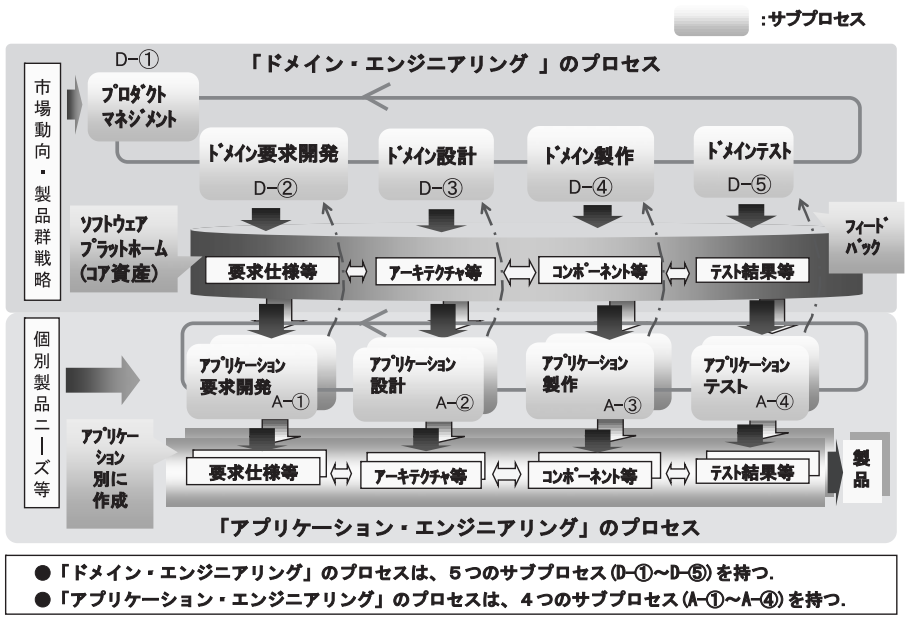


図７　ソフトウェア・プロダクトラインの開発プロセス

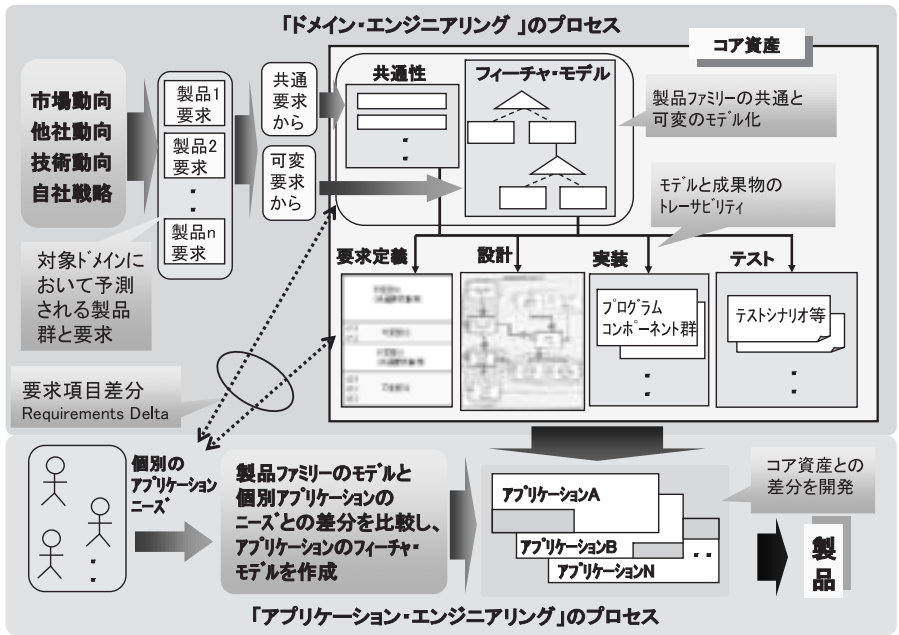


図８　ソフトウェア・プロダクトラインの概念図

テスト」の成果物を再利用してテストを実施していきます。

６．２　モデルベース開発との連携

ソフトウェア・プロダクトラインの開発プロセスの中で、モデルベース開発の手法を導入することが効果的です。「ドメイン・エンジニアリング」のプロセスにおける「ドメイン要求開発」、「ドメイン設計」、「ドメイン製作」のサブプロセスが対象となります。

「ドメイン要求開発」では、「ユースケースビュー」により「要求モデリング」とのトレーサビリティを確保します。具体的には、UMLのユースケース・モデル（ユースケース図）で、フィーチャ・モデルの「可変な要求項目（可変性）」をどう実現させるかを決定していき、可変部分の機能の一貫性を検証していきます。

「ドメイン設計」では、「論理ビュー」と「プロセスビュー」を使用した2種類のアプローチが考えられます（図9）。まず1つ目のアプローチは、トップダウン・アプローチです。フィーチャ・モデルからコンポーネント化の単位を決めて、その論理的構造を分析、設計し、UMLの分析、設計モデル（コンポジット構造図、クラス図、シーケンス図など）として表現していきます。

2つ目のアプローチは、ボトムアップ・アプローチです。可変部分の機能が反映されたユースケース・モデルのユースケース・パターン毎にUMLのクラス図によりクラスの静的構造を決め、さらに可変部分と共通部分についてコンポーネント化（コンポジット構造図で表現）し、UMLのシーケンス図、ステートマシン図などにより動的構造（振る舞い）の妥当性を検証していきます。

「ドメイン製作」では、「実装ビュー」によりフィーチャ・モデルを実現するコンポーネント構成を「実装モデリング」としてモデル化します。それぞれのアプローチの中で、「ZIPC++」は実装ソースコードを生成するための「論理ビュー」モデルの設計に適用できます。

7．今後の目指すべき開発スタイル

当社は、今後、上流から下流工程までオブジェクト指向手法をベースとしたモデル中心の開発スタイルを目指していきます。そして品質向上はもとより、開発効率向上を実現すべく、ソフトウェア資産の再利用をテーマにして開発を進めていきます。以下に、それを展開するための具体的な開発プロセスについて述べます。

これまで見てきたソフトウェア・プロダクトラインの開発方式を上流工程で取り入れるため、「ドメイン・エンジニアリング」と「アプリケーション・エンジニアリング」のプロセスを導入します。そしてフィーチャ・モデルと「ユースケースビュー」のユースケース・モデルを併用することで、システムの機能要件を、再利用のための可変部分と共通部分に分けていきます。そして、「論理ビュー」の構造モデル（クラス図、コンポジット構造図、シーケンス図、ステートマシン図、「ZIPC++」STM）を使用して、可変部分と共通部分のコンポーネント化を行うため、

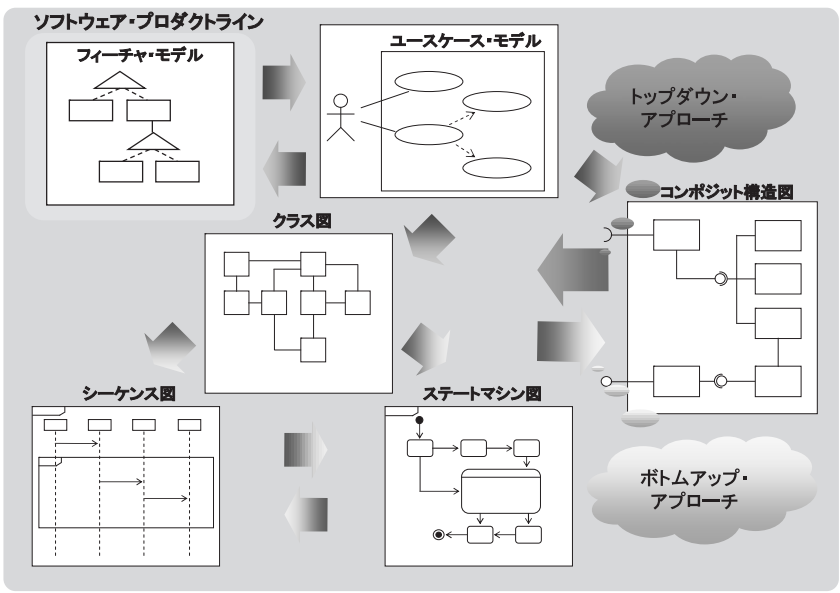


図9　ソフトウェア・プロダクトラインとモデルベース開発

分析モデリング、設計モデリングへと繋げていきます。「実装ビュー」の構造モデル（コンポーネント図、パッケージ図）を使用して、再利用のための可変部分と共通部分を明確にしたソフトウェアの全体構成をまとめます。このような開発プロセスを実現するためには、組込み技術者のモデリング技術力とモデルベースの開発支援ツールが不可欠であると認識しています。最後に、ソフトウェア・プロダクトラインから実機テストまで支援するツール連携について述べます。

8．モデルベース開発支援ツールの連携

開発効率向上のためには、ソフトウェア資産の再利用以外に、モデルベース開発を支援するツール群とツール間の連携機能が極めて重要になってきます。ソフトウェア・プロダクトラインを支援するツールとしては、新規に多品種開発支援ツール「ZIPC Feature」と「ZIPC SPLM」が登場しました。またUMLモデリングツールとしては、「Enterprise Architect」、「Rhapsody」「Rational Rose」、「Pattern Weaver」など、その他多数の開発支援ツールがあります。

「ZIPC Feature」とUMLモデリングツールとの連携や「ZIPC++」とUMLモデリングツールとの強力な連携機能に期待しています。また今後、当社の組込みソフト向けデバッグ・性能解析支援ツール「SagePro/eDEBUG」と「ZIPC++」との連携により、デバック効率向上を図っていきたいと考えています。

ここで「SagePro/eDEBUG」について簡単にご紹介します（図10、図11）。本ツールは、組込みソフト開発における最適なテスト環境を実現すべく、実機での組込みソフト全体の動きを「見える化」するソフトウェアです。主な特徴としては、「ZIPC」の状態遷移表を用いたデバッグが可能であること、組込みソフト全体の動きと個々のタスク動作との連携が可能であることが挙げられます。現在「ZIPC」との連携ツールとして販売を始めています。

近い将来において、上流工程の要求定義から下流工程のテストまで、開発支援ツールの連携が実現できれば、大幅な開発効率の向上が期待できると確信しています。

9．おわりに

以上述べてきた通り、現在、組込み分野では、体系的な開発手法や標準的な開発プロセスの確立が必要になってきています。日本の組込み技術の発展は、これからの組込みシステム開発技術の進化と、組込みソフト技術者の技術力アッ

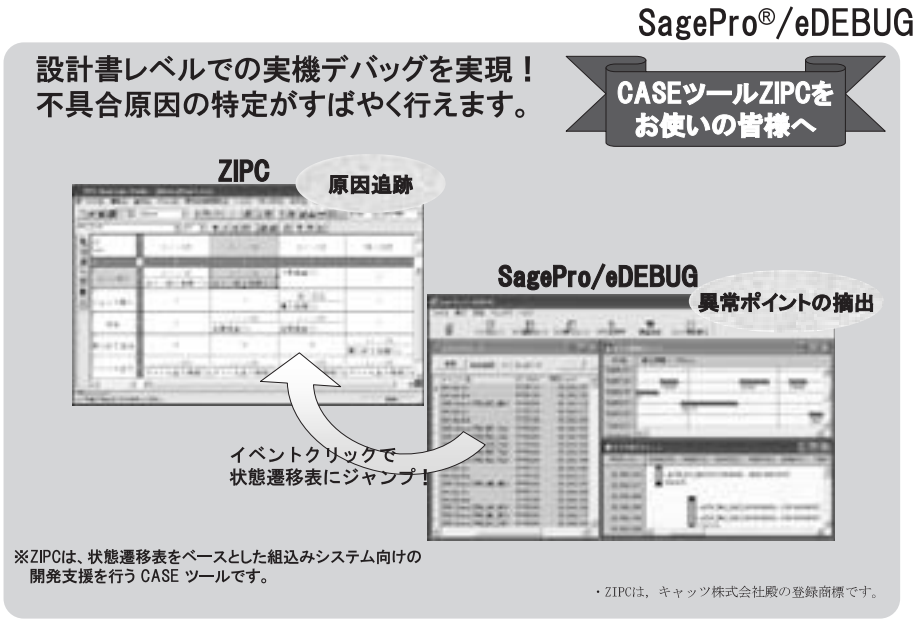


図10　ZIPCとのツール間連携

プに懸かっています。

当社も日本の組込み技術の発展や組込みソフト技術者の育成に少しでも役立ちたいと考えております。

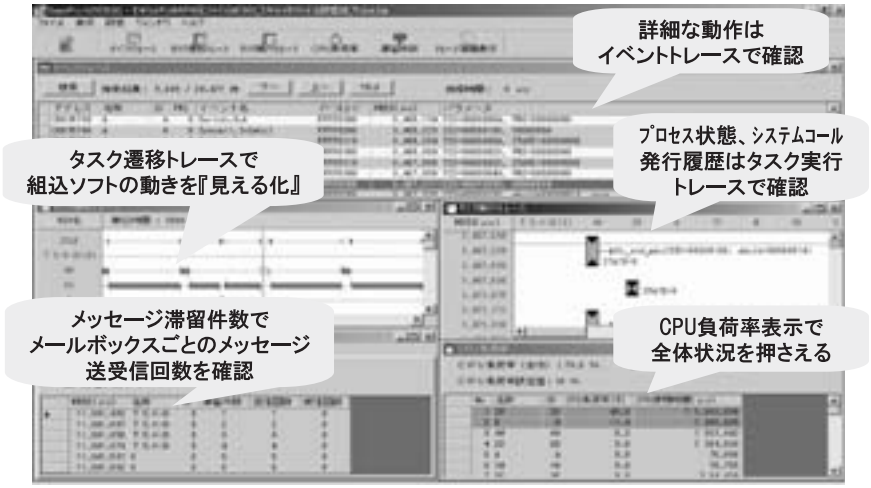


図11　デバッグ・性能解析支援ツールの機能概要

［参考文献］

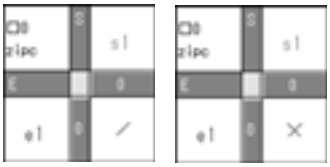
1）Philippe Kruchten:
「Architectural Blueprints - The “4+1” View Model of Software Architecture」
(IEEE Software,vol.12,no6,1995,pp.42-50)

2）Klaus Pohl, Gunter Bockle, Frank van der Linden:
「Software Product Line Engineering」
(Springer-Verlag)

ご存知ですか？ EHSTM/ZIPC 豆知識

STM の表現で、無視と不可の違いは何でしょうか？

無視とは、マトリクスの組み合わせ上ありえるが、あえて処理を行わない場合に使用します。不可とは、マトリクスの組み合わせ上、絶対にありえない場合に使用します。



動作結果はどちらも同じですが、設計上の無視と不可を明示する意味で使い分けます。
また、無視と不可を分別し、エントリーした際に特定の処理を行うことも出来ます。