Vol.2012-SE-177 No.2
2012/7/19

既存ソフトウェアに対するモデルベース開発導入プロセス

浅田幸則† 大條成人† 松本紀子† 田代沙希子†

継続的に開発されている製品にモデルベース開発を導入するには，限られた工数の中で既存ソースコードをモデル化することが課題となる．そこで，本稿では，既存ソースコードを最大限活用するためのモデル化プロセスを定義した．本モデル化プロセスは，モデル化の前に既存ソースコードをリファクタリングし，３つのモデルタイプを用いた選択的モデル化をすることが特徴である．このリファクタリングと選択的モデル化により既存ソースコードをモデルに活用することが可能となる．さらに提案したモデル化プロセスを実際にデジタルテレビ向けソフトウェアの開発へ適用した．その結果モデルベースを導入しない開発と比較し約 16%の開発工数削減の効果を確認した．

Model-based Development Process for Existing Source Codes

YUKINORI ASADA† SHIGETO OEDA†
TOSHIKO MATSUMOTO† SAKIKO TASHIRO†

In case of applying model-based development for continuously developed products, it is an issue to model an existing source code in a limited man-hour. Therefore, in this report, we defined a new modeling process that maximizes utilization of existing code base. This modeling process consists of two features. First one is re-factoring of existing codes before modeling. Second one is selective modeling from three model types before modeling. Utilization of existing code base becomes easy by these features. In order to confirm the effectiveness of the proposed method, we applied the modeling process to digital television software. We confirmed an effect of approximately 16% of development time was reduced in comparison with the development without applying model-based development.

1. はじめに

近年の組込みソフトウェアは，高機能化に伴い急速に複雑化・大規模化していることが指摘されて久しい．一方で開発期間の短縮と開発コストの削減が求められ，開発効率の向上が急務となっている．この状況に対し，ソフトウェアの構造や処理をモデルで表現して設計・検証を行うモデルベース開発(Model-based Development; MBD)が注目を集めている．MBD を導入することにより，モデルを設計，コーディング工程，テスト工程で一貫して活用することが可能となり，開発効率が向上する[1][2].

しかしながら，実際に製品開発と並行して MBD を導入することは容易ではない．新機能あるいは新製品をゼロから構築する場合, MBD の導入効果は比較的得られやすい．ところが，既に製品が継続的に開発されている場合における製品開発の多くは，既存コードの拡張である．つまり，新機能の開発に加えて既存機能のモデル化も並行して実施しなければならない．しかも製品開発工数は短縮傾向にあるため, 既存資産をモデルに変換する工数を確保できない．一旦 MBD を導入すれば，次機種以降で MBD の開発効率向上メリットを享受できるとわかっていても，初期導入コストの高さから MBD を導入することができない．このような，既存資産に対してモデル化をする取組みとして，アプリケーション層のみをモデル化した例[4]や，仕様設計にモデルを導入した例[5]がある．

モデル化作業以外にも，MBD 導入障壁はあるが，モデル化作業が大きな障壁の一つである．そこで我々は，製品の開発工数内で，既存機能のモデル化を実現するためのモデル化プロセスを提案し，デジタルテレビの製品開発へ実適用を行った．なお，本稿では，モデリング言語として UML(Unified Modeling Language)を想定する[6].

2. 課題とアプローチ

新規に開発をする場合，機能要件等からモデルを設計し，実装するのが一般的な MBD の開発プロセスである．しかしながら既に継続的に開発されている機能へ MBD を適用する場合，すべてを作り直す工数は認められないことも多い．この場合，いかに既存ソースコードを活用し，限られた工数の中でモデル化することができるかが鍵となる．そこで，既存ソースコードを活用しモデル化する場合の課題を整理し，その解決案としてのアプローチを示す．以下は既存ソースコードをモデル化し，活用する上での主要な課題である

- 静的な構造の複雑さ
  開発を繰り返すうちに, 暫定対策やその場しのぎのコーディングで，複雑な構造になっている．本来，分離すべき機能が混同している，あるいは，似たような機能が複数存在することがある．
- 動的な振舞いの複雑さ
  静的な構造と同様に, 暫定対策やその場しのぎのコーディングで, 状態遷移や処理シーケンスも複雑になっていることが多い. 継ぎはぎになった状態遷移や当初

† (株)日立製作所 横浜研究所
Hitachi Ltd. Yokohama Research Laboratory

の設計を無視した処理シーケンスも多く存在する.

- ソースコードのブラックボックス化
  頻繁な担当者変更，設計ドキュメントの保守不備等，変更を加えることが困難なソースコードが混在している事も少なくない.
- 限られた製品開発の工数
  製品開発と並行してモデル化を進める場合，既存ソースコードをモデル化することに加え，新機能の追加，仕様変更への対応等も並行して行う必要がある．これに対して追加の工数が認められないことも多い.
- 既存機能の安定性
  上記のような理由から，一度にモデル化する対象となる既存ソースコードの規模が大きくなるに従い，限られた工数内で既存機能の安定性を確保することが難しくなる.

これらの課題を解決するためのアプローチを図 1 に示す． 「静的な構造の複雑さ」と「動的な振舞いの複雑さ」に対しては，機能を整理する等の簡易リファクタリングを行う(図 1 の①)が，「ソースコードのブラックボックス化」に対しては無理にモデル化せずそのまま既存ソースコードを活用する(図 1 の②)．また，「限られた製品開発の工数」と「既存機能の安定性」に対しては，システム全体をモデル化するのではなく，部分的にモデル化することを検討する(図 1 の③)．そして，モデル化をする場合も，ソースコードの性質によりモデルタイプを制限し，少ない工数で安定性を確保する(図 1 の④).

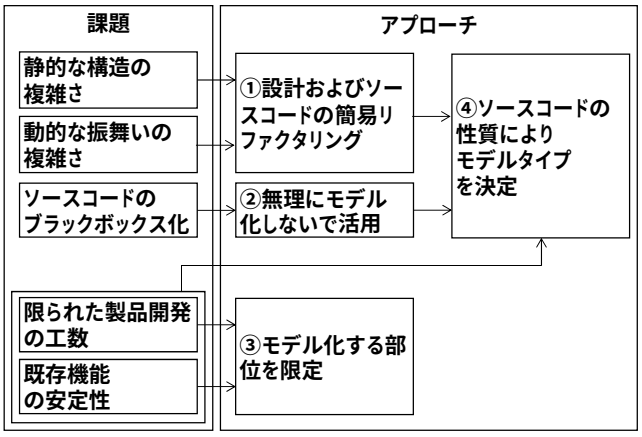


図 1 既存ソースコードのモデル化活用における課題とアプローチ

Figure 1 Issues and solution on utilization of existing code for modeling

3. 既存ソースコードに対するモデル化プロセス

2 章で示したアプローチを，既存ソースコードをモデル化するためのモデル化プロセスとして定義した．その概要を図 2 に示す.

まず，既存ソースコードをモデル化する場合，システム全体をモデル化するのではなく，部分的な機能をモデル化し段階的にモデル化の範囲を広げることも検討する(Step1). 次に既存ソースコードをできるだけ活用する．ソフトウェアの機能要件等から理想的にモデル設計を実施すると，既存ソースコードの再利用は困難であることが多い. つまり，ゼロから作り直すことになり，多くの場合工数が不足する. そこで，理想的なモデルを設計した後で，既存ソースコードを活用するのではなく，対象機能を分析(Step2)し，ある程度ソフトウェアの構造を見直した後(Step3)，ソースコードベースでモデル化を検討する(Step4).

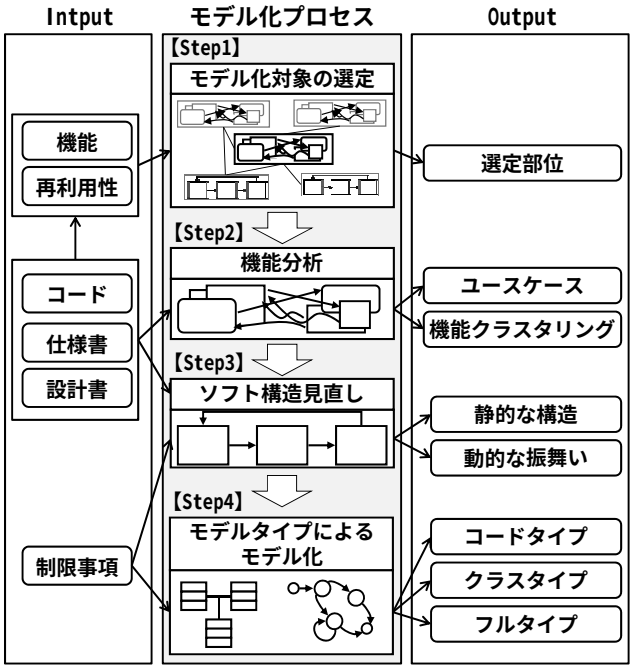


図 2 既存ソースコードをモデル化するためのプロセス

Figure 2 Modeling process for existing codes

以下，詳細に各 Step を説明する.

【Step1】モデル化対象の選定

システム全体を 1 度にモデル化するトップダウン方式で進める場合と，システムのモデル化を部分的に進めるボトムアップ方式で進める場合がある．トップダウン方式の場合には，システム全体を一度にモデル化するためのまとまった工数が必要となるが，モデル化に当たり個々の機能の抜本的な見直しが可能となる．一方，ボトムアップ方式

の場合には，部分的にモデル化を行っていくため大幅な機能の見直しは難しいが，部分的な修正となるため，まとまった工数の確保は必要とならない．部分的なモデル化にとどめることにより，システム全体でのモデル化の効果は小さくなるが，製品開発における遅延リスクを減らすことが可能となる．

ボトムアップ方式で進める場合には，モデル化する部位を選定する必要がある．再利用性の高い部位，コードサイズが大きく処理が複雑な機能を優先的に選定することでモデル化の効果を高めることができる．

【Step2】機能分析

モデル化対象となる部位は，大規模かつ複雑であることが多いので，まずは機能を整理し，設計を見直すことが重要である．ソースコード，仕様書，設計書等からユースケースや機能クラスタリングを再検討することで，複雑化していた設計を整理する．このとき，静的な構造と動的な振る舞いという観点で整理する．また，この検討の際，以下のような理由から，対象部位の設計者(あるいは設計を理解している者)を検討メンバに加えることが有効である．

- 仕様書，設計書が完璧だとは限らない
- 大規模かつ複雑なコードの分析には時間がかかる

【Step3】ソフト構造見直し

機能分析の結果を受け，静的な構造と動的な振る舞いをリファクタリングする．本来，Step3 はスキップすることは可能で，機能分析が完了すれば Step4 のモデルタイプの決定を行ってもよい．しかし，開発スタイルによっては，定期的にソースコードをリリースしなければならない．この場合，段階的にリファクタリングを進めることで，日程遵守と設計変更による動作不良リスクを削減することが可能となる．

また，Step2,3 におけるリファクタリングは，開発スケジュール(工数)や関連モジュールへの影響を考慮し，段階的に実施することも考慮する．

【Step4】モデルタイプに従ったモデル化

モデルの基本要素である静的な構造，動的な振る舞いのうち，構造をモデル化したクラス図と振る舞いのうち状態遷移処理をモデル化したステートマシン図については，コードと比べたモデルの抽象度が高く，コードの生成効率が高い．そこで，既存ソースコードをリファクタリングした後は，ソースコードの静的な構造と動的な振る舞いの複雑さに応じて表 1 に示す 3 パターンの中からモデルタイプを選択する．これは，なるべく工数をかけずに最大限モデル化の効果を得るためである．

表 1　3 つのモデルタイプ
Table 1　Three Model Types

	①コードタイプ	②クラスタイプ	③フルタイプ
静的な構造	簡素	複雑	複雑
動的な振舞い	簡素	簡素	複雑

クラスタイプとフルタイプの判断は，状態遷移処理の比重によって決定する．状態遷移処理の比重が高い場合は，フルタイプでモデル化を行う．状態遷移処理が少なく逐次処理の比重が高い場合は，クラスタイプでモデル化を行う．その判断は設計者が仕様から行うことが望ましい．しかし，設計者が不在など，設計者がモデル化を行うことができないケースも発生する．その際には，メトリクスや仕様書を用いて，状態遷移処理の比重の目途を立てることが可能である．以下，それぞれのモデルタイプを詳細に説明する．

①コードタイプ

コードタイプとは，モデル化せずにコードだけで実装するタイプである．簡素な構成のソースコードや修正が難しい他社 IP のソースコード等が該当する．これは既存ソースコードを最大限生かすタイプであり，Step3 までのリファクタリングによってソースコードの静的な構造と動的な振る舞いのどちらも簡素になった場合も該当する．ただし，該当する部分を一つのクラスとして扱ってもかまわない．そうすることにより UML ツール等で統一的に管理することが可能となる．

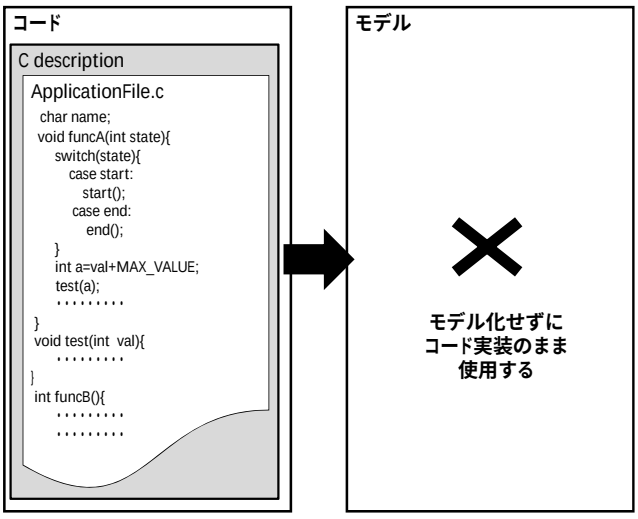


図 3 コードタイプ
Figure 3 “Code Type”

②クラスタイプ

クラスタイプは，システムの静的な構造のみをモデル化するタイプである．静的な構造には，機能内部で相互に使用される内部関数/データ等と，機能外部から使用される外

部関数/データなどがある.クラスタイプ機能のモデル化では，静的な構造をクラス図でモデル化する．静的構造以外の動的な振る舞い部分は，モデル化せず実装をそのまま使用する．

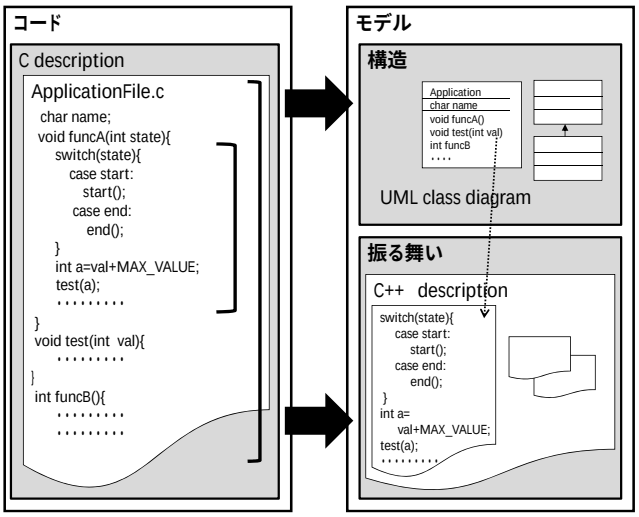


図 4　クラスタイプ
Figure 4 “Class Type”

③フルタイプ

フルタイプは，静的な構造と動的な振る舞いをモデル化するタイプである．動的な振る舞いには，状態に応じて実行する処理が変化する状態遷移処理と，状態遷移以外の処理を行う逐次処理の 2 種類の振る舞いがある．フルタイプ機能のモデル化では，静的な構造はクラス図を用いてモデル化し，動的な振る舞いの状態遷移処理はステートマシン図[8]を用いてモデル化する．状態遷移以外の逐次処理などの動的な振る舞いについては,実装をそのまま使用するか,あるいはアクティビティ図を使用する．

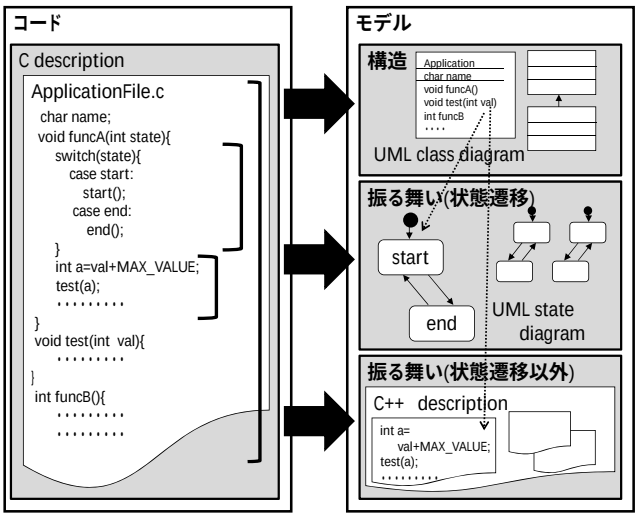


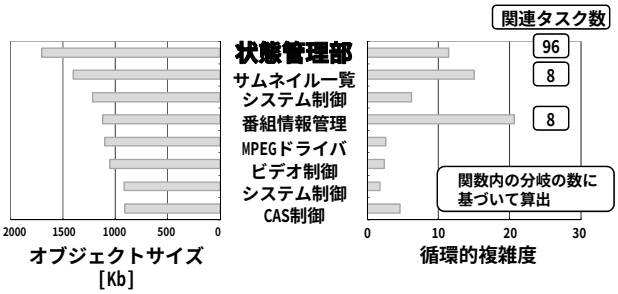
図 5　フルタイプ
Figure 5 “Full Type”

4. デジタルテレビへの適用

3 章において定義したモデル化プロセスを，デジタルテレビ(DTV)のソフトウェア開発に適用した．

4.1 モデル化対象の選定[Step1]

DTV を構成するモジュールのうち規模が大きく，構造や処理フローも複雑である上，今後の DTV の高機能化に伴って他タスクと比較して多くの改変が今後も継続的に必要となる状態管理部を選定した．図 6 に実測したオブジェクトサイズと複雑度を表す指標の一つである循環的複雑度の上位に位置するものを示す．ただし，他社から提供をうけたソフトウェアは除いている．図 6 に示すとおり，状態管理部は，規模と循環的複雑度が大きく，関連タスク数が 96 あり，モデル化の効果が大きいことが期待できる．循環的複雑度[7]は，関数内の分岐の数に基づいて算出した．



(a)オブジェクトサイズの比較　(b)循環的複雑度の比較
図 6　状態管理部の規模と複雑度
Figure 6 Code size and code complexity

4.2 機能分析とソフト構造見直し[Step2][Step3]

4.2.1 状態管理部の概要

状態管理部とは，ユーザ操作や予約機能などのトリガに基づいて，アプリケーションの起動・終了を制御するミドルウェア層のモジュールである．

たとえばユーザが録画番組一覧を表示させる操作を行った場合，図 7 に示すように，

(1) リモコンキーのコード(キーコード)を受信し，起動対象アプリケーションを決定する．
(2) 次に，内部で管理しているシステム状態を確認し，必要に応じてシステムモジュールに対して，番組再生中の HDD を停止させるなど，システム状態の変更を要求する．
(3) そして，最後にアプリケーションモジュールに対して起動，すなわち録画番組一覧の表示を要求する．

状態管理部は他にも番組表,メニューなど数多くのアプリケーションを制御しており，大規模かつ複雑なモジュールと化している．

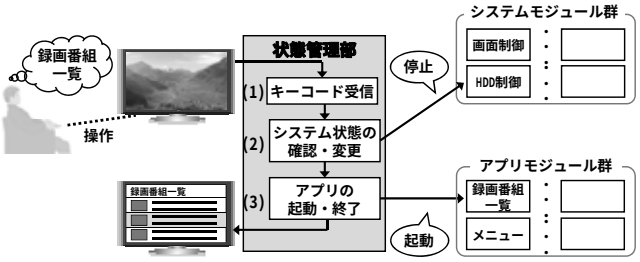


図 7　録画番組一覧の表示手順
Figure 7 Indication procedure of contents table

4.2.2 モジュール構成の最適化

状態管理部では，多様な機能の混在が複雑化の要因となっている．これらの整理と分類を行った結果，性質が異なる 6 種類の機能があることが明らかになった．

複雑度を軽減するため，キーコード制御部，アプリケーション制御部，システム状態制御部の 3 つのモジュールとして整理した．また，アプリケーション制御部とシステム状態制御部が，それぞれアプリケーションとシステムの状態を一元管理できる独立性の高いモジュールとなるように，6 種類の機能を振り分けた(図 8). 分割後の新しい構成では，3 つのモジュールによって従来の状態管理部と同等の機能を果たす.

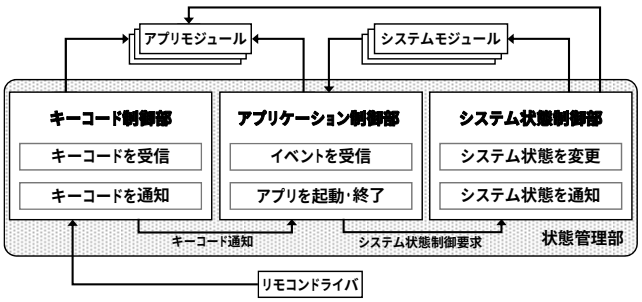


図 8　モジュール分割後の状態管理部
Figure 8 State controller after refactoring

4.2.3 制御フローの統一

従来のアプリケーション制御フロー(図 9)では，まず，キーコードを受信して，制御対象となるアプリケーションを決定する．次に，現在のシステム状態の確認，システム状態の変更，変更結果の確認を行う．最後にアプリケーション制御を実行する.

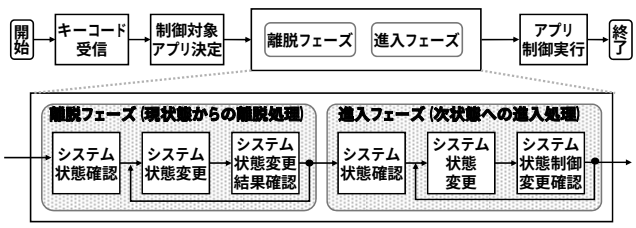


図 9 従来のアプリケーション制御フロー
Figure 9　Control flow for applications before refactoring

図 9 中の離脱フェーズと進入フェーズは，システムの状態遷移をする際の前処理として現在の状態から離脱する処理，後処理として次の状態へ進入する処理を目的としたものである．本来は，全アプリケーションの制御において両フェーズを処理すべきである．しかしアプリケーションに応じていずれか一方のフェーズを選択する実装となっていた．また，システム状態を変更するフローもアプリケーションに依存していた．変更が必要となるシステム状態の数はアプリケーションによって異なるが，その数に応じて繰り返し処理することが必要であった.

制御フローを統一するためにはアプリケーションに依存した処理をなくす必要があり，まず，離脱フェーズと進入フェーズの区別を廃止した．次に，フェーズの区別を廃止したフロー中の各処理を，分割した 3 つのモジュールで分担した．ここで，アプリケーション制御部からシステム状態制御部に対してシステム状態の変更を要求する際には，変更要求リストを送信するようにした．これにより，複数のシステム状態を変更する場合でも，統一されたフローで処理することが可能となった(図 10).

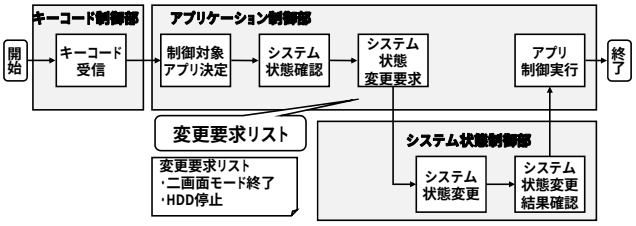


図 10 再編後のアプリケーション制御フロー
Figure 10 Control flow for applications after refactoring

4.3 モデル化タイプに従ったモデル化[Step4]

機能分析とソフト構造の見直しによって，状態管理部は大きく 3 つのモジュールに整理した．また，動的な振る舞いも見直したため，図 11 のようなクラスタイプに決定した.

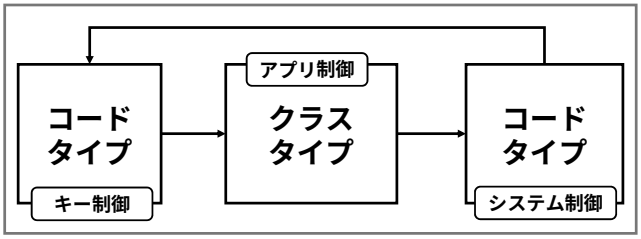


図 11　状態管理部のモデルタイプ
Figure 11　model types for state controller

キー制御部は，キーコードを受信し単純に振り分けるモジュールにリファクタリングしたため，コードタイプとした. 同様に, システム制御部もリファクタリングした結果，

システムの状態を保持するモジュールとなったためコードタイプとした．アプリケーション制御部においては，状態遷移処理が少なく逐次処理の比重が高いのでクラスタイプとした．以上のように，最低限の機能分析とソフト構造の見直しを行うことにより，既存ソースコードを活用しながら，モデル化することができた．

5. 適用結果

5.1 開発工数

状態管理部の開発において MBD を導入しない場合は，25 人月と見積もった．この見積もりは，社内の指標に従い長年の担当者が算出したものである．図 12 の上段は見積もり工数，下段は実際の工数実績を示している．従来の工程は要求分析から始まるが，今回はその前段でモデル化プロセスを新規導入し，4 人月を要した．また，本製品の新規機能対応として 21 人月の実績となった．その結果，製品開発工数内で開発することができた．新規機能の対応に要する工数は 25 人月から 21 人月へ，すなわち約 16%削減できた．今回新規に追加した 4 人月のモデル化プロセスは，次機種以降必要なくなることから，次機種以降の開発ではこの工数削減効果の恩恵を受けることが可能となった．

なお，既存ソースコードを利用せず，ゼロから理想的にモデル化した場合の見積もりは，コーディングだけでも 20 人月を超えると予想していた．

モジュール構成の最適化と制御フローの統一を実現したことで状態管理部の複雑度が軽減され，静的モデルを用いた既存資産への MBD 導入が容易になった．

モジュール構成の最適化は主に不具合解決のスピード向上，制御フローの統一は主に設計ミスの減少に寄与した．この他にもコーディングの作業量が減少するなど，MBD 導入により，各過程で工数削減の効果が得られた．

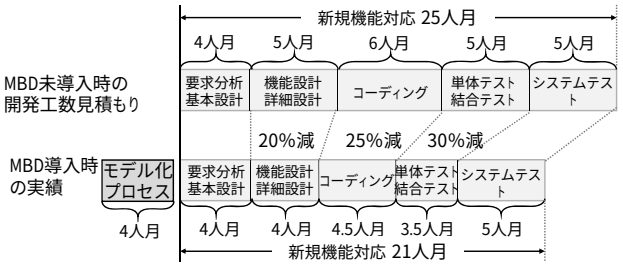


図 12　MBD 導入による工数削減効果

Figure 12　Efficiency of workload reduction

5.2 ソースコード規模

開発前の状態管理部と MBD 導入後の状態管理部のソースコード規模を表 2 に示す．MBD の導入の結果ソースコードのステップ数は増える結果となった．これは，新規機能の追加，他タスクに対するモデル化の影響を最小限にするための API 追加，モジュールを整理した際の API 追加等の影響が考えられる．次機種以降，さらなるリファクタリングおよびモデルの洗練化をすることによりソースコード量は削減可能である．

表 2　ソースコードのステップ数

Table 2　Step count of Codes

（従来の）状態管理部	27.3 Kstep
アプリケーション制御部	21.9 Kstep
システム制御部	10.6 Kstep
キー制御部	9.2 Kstep

5.3 複雑度

開発前の状態管理部と MBD 導入後の状態管理部の循環的複雑度を表 3 に示す. 表 3 に示すとおり循環的複雑度の平均値がいずれも従来の状態管理部より低くなっており，各モジュールにおける処理が簡潔になっていることが数値上でも明らかになった．

表 3　循環的複雑度の平均値

Table 3 Average of code complexity

（従来の）状態管理部	11.5
アプリケーション制御部	9.7
システム制御部	5.9
キー制御部	8.9

6. 関連研究

ソフトウェアの構造や処理をモデルで表現して設計・検証を行うモデルベース開発(Model-based Development; MBD)あるいはモデル駆動開発(Model-driven Development)を導入する取組みが行われている[1][2][9][10]．本取組みもこれらの事例の一つである. MBD を導入することにより，モデルを設計，コーディング工程，テスト工程で一貫して活用することが可能となり，開発効率が向上する．

MBD の導入に際してオブジェクト指向でソフトウェアを構築する方法や, その効果などが論じられている. また，モデルを効率的に設計するために, UML の有効な活用方法の研究もおこなわれてきている[6][10]．本取組みにおいても，UML を活用しモデル化を行った．UML は，コード自動生成などのツールが充実しているというメリットがある[13][14]．UML ツールを用いモデル設計を行い，ソースコードを自動生成できるメリットは大きい．また，モデル設計の典型的なパターンを定義し，モデル設計の効率化を図る取り組みも行われてきた[8]．そして，モデルを用いたソフトウェア開発のプロセスの統一化やガイドライン化などの取り組みも行われ[11][12], 実際の製品開発への適用も進

んできた．しかしながら，既存のソフトウェア資産に対して MBD を導入するのは容易ではない. それは, 開発工数, スケジュール，予算などの厳しい製品開発の制限の中，大規模化・複雑化した既存ソースコードのモデル化と並行して新規機能の開発を行わなければならないからである．こういった既存のソフトウェアに MBD を導入する取組みも行われており[3][4], 小規模の開発に対してアプリケーション層全体をモデル化するという検証プロジェクトを行い，高い効果をあげている．本取組みは，既存のソフトウェアに対して MBD を導入するという点で共通しているが，大規模な商品開発に導入していること，さらに，ミドルウェアの機能をモデル化した点が異なっている. 本取組みでは, MBD の効果を最大限アピールするため，敢えて難易度の高いミドルウェア層で，しかも製品の中核をなす機能をモデル化した．これにより，今後他の部位にモデル化の範囲を広げるための心理的効果も得られたと考えている．

既存ソースコードに対して開発効率向上を目的にソースコードのリファクタリングをする取組みもおこなわれている[17]．既存ソースコードをリファクタリングする点は共通しているが, 本取組みで提案したモデル化プロセスは, モデル化の前にリファクタリングするところが異なっている．

また，近年では，製品系列を横断した既存ソースコードの再利用技術としてソフトウェア・プロダクトライン工学(SPLE：Software Product Line Engineering)[15]が注目されており, 製品に適用する取組みも行われている[16]. SPLE は, 製品系列を横断する共通的なアーキテクチャを明確にし，それに基づいた再利用可能なソフトウェア資産を体系化するものである．SPLE は，派生開発を繰り返している製品には特に有効と考えられ, DTV においても効果があることが予想される．今後，提案手法への適用可能性を検討していきたい．

7. おわりに

製品開発工数を遵守しながら MBD を導入し開発効率を向上させる目的で，既存ソースコードを有効活用するモデル化プロセスを定義した．また，DTV の状態管理部の開発へ実適用し効果を確認した．今回は，一部の機能への適用であったが，今後システム全体へ適用できるかが次のチャレンジであり，さらには，DTV 以外への製品に適用していく．

参考文献

1) Frankel,S. : MDA モデル駆動アーキテクチャ，日本アイ・ビー・エム TEC-J MDA 分科会，エスアイビー・アクセス(2003).
2) LANGLOIS,B., BARATA, J., EXERTIER, D.: Improving MDD Productivity with Software Factories, International Workshop on Software Factories, San Diego, California, USA (2005).
3) IBM：モデル駆動型開発手法によりソフトウェア開発の効率化と品質の向上を実現，IBM PROVISION, No. 57, pp. 18-24 (2008).
4) 高橋知信，南光孝彦： 組込み分野へのモデル駆動型開発手法の適用, Panasonic Technical Journal Vol.56,pp60-62(2010).
5) 川上真澄，小川秀人：デジタル家電ドメインに特化したモデルベース開発環境, 情報処理学会論文誌(2011 年 12 月), Vol.52 No.12, pp3184-3191(2011).
6) OMG: UML2.4.1, available from <http://www.omg.org/spec/UML/2.4.1 >, (2011).
7) McCabe, T.J.: A Complexity Measure, IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 308-320(1976).
8) Erich, H., Richard, J., Ralph, V., et al.: Design Patterns: Elements of Reusable Object-Oriented Software (1995).
9) Cook, S. and Kent, S.: Generative Techniques in the context of Model Driven Architecture, The Tool Factory, OOPSLA(2003).
10) ヨシュ ヴァルメル 他：UML/MDA のためのオブジェクト制約言語 OCL 第 2 版，エスアイビーアクセス(2004).
11) Philippe K.：ラショナル統一プロセス入門，ピアソン・エデュケーション(1999).
12) 和田洋，安竹由起夫：MDA（Model Driven Architecture）と現実の開発プロセス，UNISYS TECHNOLOGY REVIEW 第 81 号, (2004).
13) 照井康真：Eclipse3+UML2.0 による実践ソフトウェア開発, 秀和システム(2005),
14) Bruce, D.:リアルタイム UML ワークショップ, 翔泳社(2009).
15) Clements, P and Northrop, L.M.: Software Product Lines: Practices and Projects, Addison-Wesley(2001).
16) Steger, M., Tischer, C., Boss, B., et al.: Introducing PLA at BOSCH gasoline systems: Experiences and practices. Proceedings of the Software Product Line Conference, SPLC'04, pages 34–50,(2004).
17) Fowler, M.: Refactoring Improving The Design of Existing Code, Addison-Wesley(1999).