一般
投稿論文

既存ソフトウェアに対する実践的なモデル化方法およびデジタルテレビへの応用

浅田 幸則[†1]　大條 成人[†2]　松本 紀子[†1]　成川 沙希子[†1]

[†1] (株) 日立製作所　[†2] (株) 日立アドバンストデジタル

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，モデル化工数を抑えることが課題となる．これによって，限られた製品開発工数の中で，新規機能の開発に加え，既存ソースコードをモデル化することが可能となる．そこで本稿では，既存ソースコードを活用する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点に特徴がある．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．本稿では，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，製品開発工数内で既存ソースコードのモデル化を実現した結果も示す．この事例では，既存ソースコードをモデル化してモデルベース開発を行ったことにより新機能開発において約 16% の開発工数を削減するとともに，品質向上の効果を確認することができた．

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と言われている[1],[2],[3].

実際, 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かもしれない. しかし，すで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．したがって，実際の製品開発と並行してMBDを導入することは容易ではない．なぜならば，新機能の開発に加えて既存モジュールのモデル化を並行して実施しなければならないからである．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するためには，さまざまな課題を解決しなければならない．いったん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分かっていても，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のが現状である．このような問題を解決するために，既存資産に対してモデル化をするために，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の導入には障壁があるが，モデル化作業のコストが開発現場の大きな障壁の1つとなっている．そこで我々は，製品の開発工数内で，既存モジュール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

新規開発にMBDを導入する場合，機能要件等からモデルを設計し，実装するのが一般的な開発プロセスである．しかし，すでに継続的に開発されているソフトウェアにMBDを適用する場合，限られた工数内でモデル化作業を取り入れるかが鍵となる．本稿では，既存ソースコードを活用する方策を検討した．

MBDを導入するにあたり，考慮すべき工数には，以下のものがある．

(a) MBDを導入しない場合の工数
(b) 既存ソースコードのモデル化に要する工数
(c) MBD導入後の工数

2014 ©Information Processing Society of Japan

（d）MBD導入による効果で削減された工数

提案する方策では，**図1**に示すように，（b）の工数を減らし，相対的に（b）≦（d）になるようなモデル化の手法となる必要がある．

以下で，既存のソースコードを活用してモデル化する場合の課題を整理し，それらの課題を解決するためのアプローチを示す．

(1) ソースコード要因

- 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，プログラムのソースコードが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もある．
- 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によって，状態遷移が継ぎはぎになったり, 処理シーケンスが複雑になっていることも多い．
- 変更不可能なソースコードの混在
  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も少なくない．また，第三者のソースコードを利用してる場合は，ライセンスによっては変更することが許されないこともある．

(2) 製品開発プロセス要因

- 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い，規定の工数内に納めなければならない. 上記のような状況は, 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，既存ソースコードのモデル化工数も増大する原因にもなっている．

これらの課題を解決するためのアプローチを**図2**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，複雑な状態でモデル化しても，複雑なモデルが導出されることが予想される．そこで，モデル化の前処理として，モジュールを整理する等の簡易リファクタリングを行う（図2の（ア））必要がある．一方,「変更困難なソースコードの混在」に対しては，リファクタリングが不可能な場合が多い．そのため，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2の（イ））．また,「限られた製品開発の工数」に対しては, モデル化対象の規模が大きくなるに連れて増加する工数に対処する必要が生ずる．そのため，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2の（ウ））．これらの対処を限られた工数で行うために, ソースコードの性質により“モデルタイプ”を定め，タイプごとにモデルの対象範囲を制限し，必要最低限の工数でモデル化を実施する（図2の（エ））こととする．モデルタイプについては，後述する．

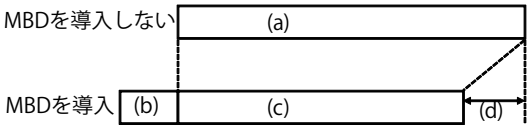


図1　MBD を導入する場合の開発工数

3. 既存ソースコードに対するモデル化プロセス

第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，その概要を**図3**にまとめた．

以下，詳細に各Stepを説明する．

【Step1】モデル化対象の選定

本手法では，既存ソースコードをモデル化することを前提としている．また，モデル化工数を正確に見積もることも困難である．そこで，モデル化工数を必要最小限度にするために，最初からシステム全体をモデル化するのではなく，部分的なモジュールをモデル化した後，工数を考慮しながら，段階的にモデル化の範囲を広げることとする．これによって，モデル化対象の規模が大きくなるに従いモデル化に費やす工数が増大し，予算を超過することを防ぐことが可能となる．加えて，システム全体のモデル化によるデグレーションのリスクを削減することも可能となる．もし，工数に余裕があり，システム全体をモデル化するのであれば,【Step1】はスキップし，各モジュールに対して【Step2】以降の各ステップを適

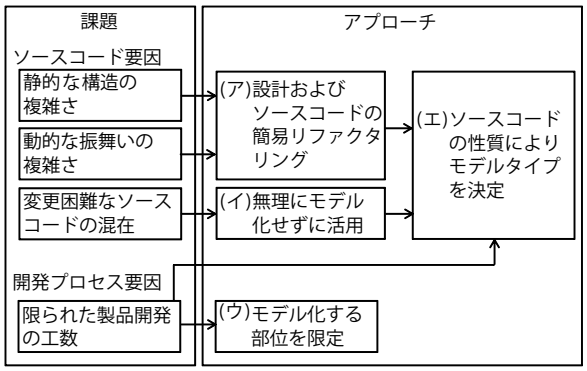


図2　既存ソースコードのモデル化活用における課題とアプローチ

用することも可能である．

モデル化対象を選定する基準は，プロジェクトの方針，工数，MBDの認知度等により変わるため一義的に決めることは難しい．【Step1】では，作業に先立って，MBDを導入する目的を明確化する必要がある．また，モデル化対象を選定するときは，設計書やソースコード等を参照する．さらに，技術的な観点だけでなく，必要に応じて，ビジネスの観点も考慮する必要がある．以下に，筆者らが実際に検討した例を示す．

例1）迅速MBD導入：MBDを低リスクで試験的に導入することが目的．比較的シンプルでモデル化が容易と予想されるモジュールを選択する．その後，徐々に複雑なモジュールにモデル化範囲を広げていった．

例2）実MBD効果獲得：再利用性の高いモジュールが複雑さを増大させていた．そこで，これらのモジュールをモデル化し，メンテナンスコストを下げることを目的とした．コードサイズが大きく複雑であったが，これを選択することで，MBD導入効果が大きくなることを期待した．

例3）MBD認知度向上：全面的にMBDを導入するために，MBDの認知度が低く，導入効果に対して懐疑的な人々に対して，MBDの効果をアピールすることが目的．例2）同様に複雑度が高いモジュールを選択し，MBDの有効性を示した．

【Step2】機能分析

【Step2】と【Step3】は，既存ソースコードをスムーズにモデル化するための前処理として，リファクタリングを位置付ける．したがって，すべてのソースコードに必要な作業ではない．モデル化対象となる部位は，モデル化を考慮した構造になっていないことも少なくない．そのため，モデル化の前に，モジュール内の機能を整理し，設計を見直さなければならない．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とができる．また，モジュールの中には，長年かけて機能追加や仕様変更を繰り返した結果，不必要になった処理や，重複している処理等が含まれていることもある．これらは，モジュールを，静的な構造と動的な振舞いという観点で整理することで，大きな工数を要することなく，見直すことが可能となる．

【Step2】では，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必要がある．

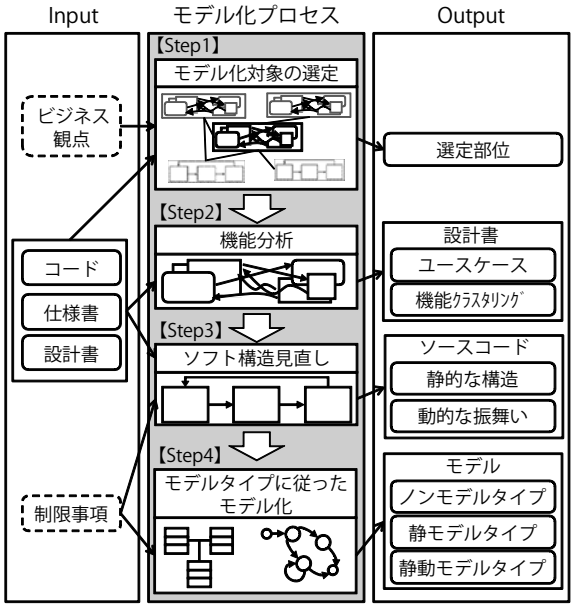


図3　既存のソースコードをモデル化するためのプロセス

- 仕様書，設計書が完璧だとは限らない
- モジュールの規模が大きく，かつ複雑な場合は，設計者の協力がなければ，分析に多くの工数を必要とする

【Step3】ソフト構造見直し

【Step2】の結果に基づいて，静的な構造と動的な振舞いを中心に，ソースコードをリファクタリングする．これらの作業を行うことで，次のモデル化作業が容易になる．ただし，リファクタリングそのものは，技術的な問題として解決できるが，多くの場合，考慮しなければならないのは，プロジェクトの制限事項である．この制限事項は，ビジネスの観点も入っており，プロジェクトごとに異なる．例を以下に挙げる．

例1）工数制限：【Step2】において，リファクタリングしなければならない項目が明確化されていても，すべての項目をリファクタリングするための工数が不足する場合があった．この場合，クラス構成に影響するような大枠のリファクタリングを優先し，詳細な処理ロジックのリファクタリングの優先度を下げることによって，全体のリファクタリング工数を削減した．

例2）インタフェースの変更制限：【Step2】においてモジュールを見直した結果，インタフェースを変更することが望ましかった．しかし，モデル化対象モジュールに依存する他のモジュールの数が多く，インタフェース変更による影響が大きくなることが予想された．そこで，暫定的に新インタフェースをラッピングし，他のモジュ

表1　3つのモデルタイプ

	(a) ノンモデルタイプ	(b) 静モデルタイプ	(c) 静動モデルタイプ
静的な構造	簡素	複雑	簡素または複雑
動的な振舞い	簡素	簡素	複雑

ールには従来のインタフェースを公開することで対応した．

例3）開発プロセス制限：筆者らの組織で行われているソフトウェア開発プロジェクトでは，製品リリースの前段階で，内部リリースを複数回実施することが多い．プロジェクトによって，内部リリースの目的は異なるが，少なくとも，内部リリース時には動作可能なものを提供しなければならない．この際，デグレーションするリスクに対処することも求められている．モデル化したモジュールに他のモジュールが依存している場合は，確実に動作するモジュールを複数回に分けてリリースすることがデグレーションリスクを下げる有効な手段であった．

【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，動的な振舞いのうち，以下のモデルはモデルの抽象度が高く，しかもコードの生成効率が高い．

- 静的な構造をモデル化したクラス図
- 状態遷移処理（動的振舞い）をモデル化したステートマシン図

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ととする．これは，モデル化工数を必要最低限に押さえ，最大のモデル化の効果を得るためである．以下，それぞれのモデルタイプを詳細に説明する．

(a) ノンモデルタイプ（**図4**）

ノンモデルタイプとは，モデル化せずに実装されたコードだけをそのまま取り扱うタイプである．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．これによって，既存のソースコードを最大限生かすことができるタイプである．【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舞いのどちらも簡素になり，モデル化の必要がない場合も該当する．このようなモジュールは，1つのクラスとして扱うこともできる．こうすることにより，他のタイプのモジュールとともに，UMLツール等で統一的に管理することが可能となる．

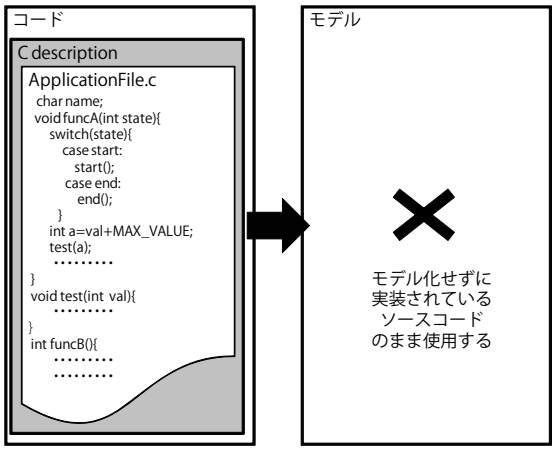


図4　ノンモデルタイプ

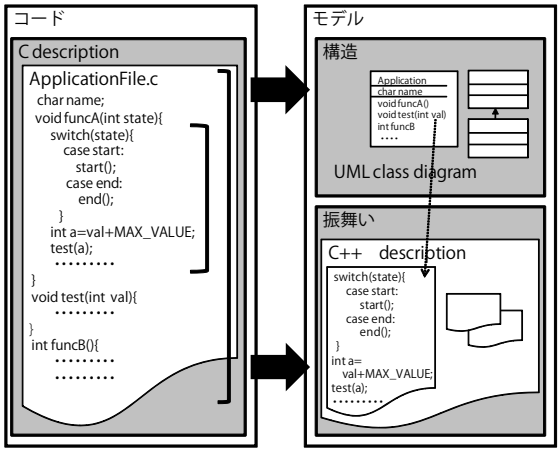


図5　静モデルタイプ

(b) 静モデルタイプ（**図5**）

静モデルタイプは，モジュール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，モジュール内部で相互に使用される内部関数／データ等と，モジュール外部から使用される外部関数／データなどがある．静モデルタイプのモジュールのモデル化では，静的な構造をクラス図で表現する．静的構造以外のメソッドなど，動的な振舞いに相当する部分は，モデル化せず実装をそのまま使用する．

(c) 静動モデルタイプ（**図6**）

静動モデルタイプは，静的な構造と動的な振舞いをモデル化するタイプである．動的な振舞いには，状態に依存した処理と，状態に依存しない処理の2種類の振舞いがある．このタイプのモジュールの静的な構造をモデル化するときには，クラス図を用いて表現し，動的な振舞いの状態に依存した処理はステートマシン図を用いて表現する．状態に依存しない処理などの動的な振舞いは，実装をそのまま使用する．このタイプを割り当てる必要があるモジュールとは，静的な構造

が簡素で，かつ，動的な振舞いが複雑であるものである．このようなモジュールは，動的な振舞いのみを分離してモデル化することが困難であることが多いためである．

したがって，静モデルタイプと静動モデルタイプのいずれを選択するかの判断は，状態遷移処理の比重によって決定する必要がある．状態遷移処理の比重が高い場合は，静動モデルタイプでモデル化を行わなければならない．しかし，状態遷移処理が少なく逐次処理の比重が高い場合は，静モデルタイプを割り当てればよい．これらの判断は設計者が仕様に基づいて行い，設計者がモデル化をすることが望ましい．しかし，設計者が不在である場合など，設計者がモデル化を行えない場合には，メトリクスや仕様書を用いることで，状態遷移処理の比重を見極めることは可能である．

4. デジタルテレビへの適用

本手法を検証するために，第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．図3で示した各Stepに従って，以下に説明する．

4.1 [Step1] モデル化対象の選定

モデル化対象を選定するにあたり，以下の判断基準を導入した．

- DTVを構成するモジュールのうち規模が大きい
- 構造や処理フローも複雑である
- 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

これらの条件に該当するモジュールとして，状態管理部をモデル化対象として選定した．その根拠となったデータを**図7**に示す．この図は，DTVを構成するモジュールごとにオブジェクトサイズと，関数内の分岐の数に基づいて算出した循環的複雑度[7]を比較したものである．ただし，他社から提供を受けたソフトウェアは除いてある．図7に示すとおり，状態管理部は，関連タスク数が96であり他のモジュールよりも多かった．関連タスク数の大きさは，製品の仕様変更に伴う状態管理部の改変頻度が高かったことを意味しており，構造上の問題を表していると考えられる．さらに，オブジェクトサイズと循環的複雑度が大きいことから，このモジュールの複雑さが，改変工数を増大させる要因となっていることを読み取ることができる．以上の評価から，この状態管理部をモデル化することにより，改変工数を削減する効果が得られると予想した．

4.2 [Step2][Step3] 機能分析とソフト構造見直し

4.2.1 状態管理部の概要

状態管理部は，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8**に示すように，以下の順番で処理が進む．

(1)キーコード受信

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)システム状態の確認・変更

内部で管理しているシステム状態を確認し，必要に応じて番組再生中のHDDを停止させるなど，システムモジュールに対してシステム状態の変更を要求する．

(3)アプリの起動・終了

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．

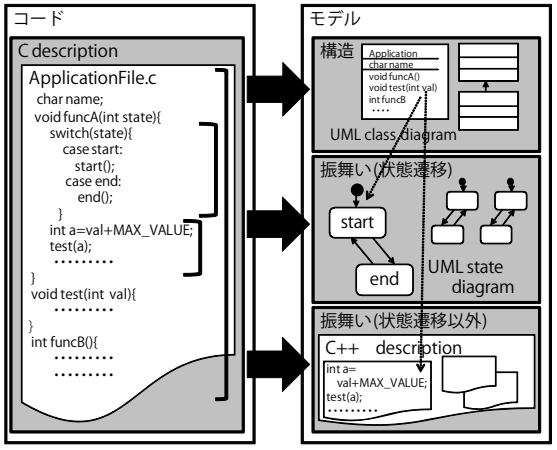
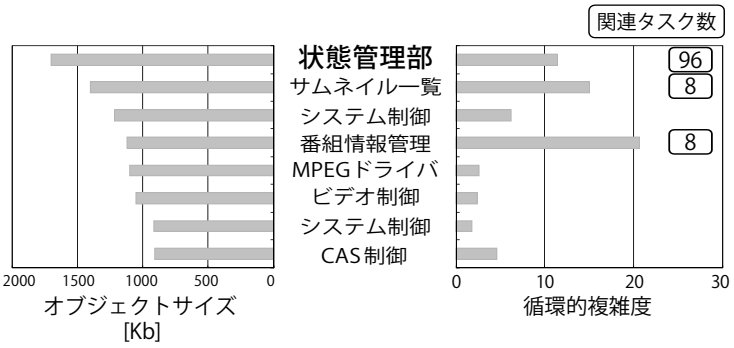


図6 静動モデルタイプ



(a) オブジェクトサイズの比較　**(b) 循環的複雑度の比較**

図7 状態管理部の規模と複雑度

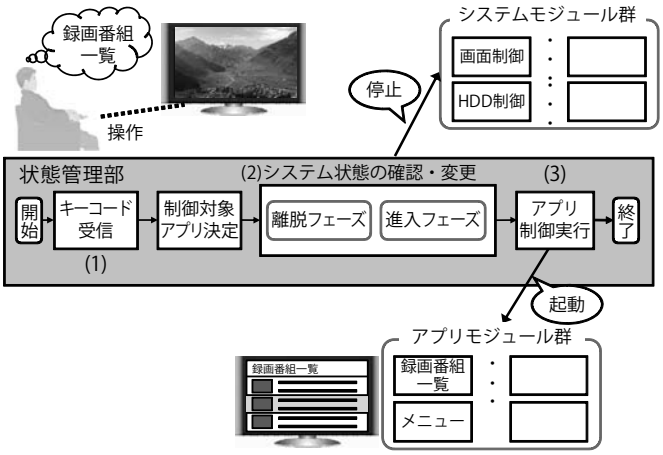


図 8　録画番組一覧の表示手順

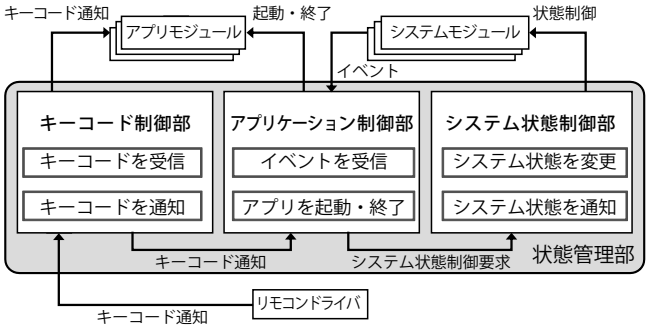


図 9　モジュール分割後の状態管理部

状態管理部は，これらの他にも番組表の表示，メニュー表示など，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なっていた．

4.2.2　「静的な構造」の機能分析とリファクタリング

状態管理部では，「静的な構造」の機能分析としてユースケースを再定義するとともに，各ユースケースで必要な処理を明確にした．その結果，性質が異なる6種類の機能が状態管理部に含まれていることが明らかとなった．

そこで，状態管理部の複雑度を軽減するため，状態管理部を，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つのモジュールに分割した．また，分散していた機能を6種類の機能に再定義した（**図9**）．以上により，状態管理部内の処理フローおよびアプリモジュールとシステムモジュールに対する制御フローが簡潔になった．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．

4.2.3　「動的な振舞い」の機能分析とリファクタリング

状態管理部の機能分析によって，**図10**に示す通り，システム状態遷移の制御フローが複雑化していることが判明した．

図10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，従来の制御フローでは，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，システムの状態を変更する制御フローも，アプリケーションに依存して多様なフローが存在していた．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，その数に応じて状態管理部は，繰り返し処理する必要があった．

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った．そこで，以下のリファクタリングを行った．

1）離脱フェーズと進入フェーズの区別を廃止した．
2）フェーズの区別を廃止した制御フロー中の処理を，3つのモジュールに分割した．
3）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．

以上により，複数のシステム状態を変更する場合でも，統一された制御フローで対応することが可能となった．再編された制御フローを**図11**に示す．

4.3　[Step4] モデル化タイプに従ったモデル化

「静的な構造」の見直しによって，状態管理部を大きく3つのモジュールに分割し，各モジュールを**図12**に示すモデルタイプとしてモデル化を行った．

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となるようにリファクタリングしたため，ノンモデルタイプで対処可能となった．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

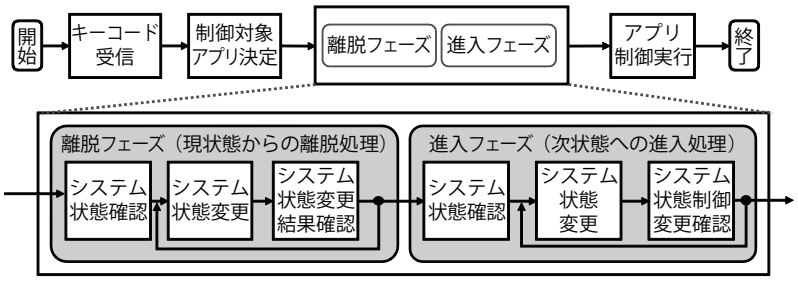


図 10　従来のアプリケーション制御フロー

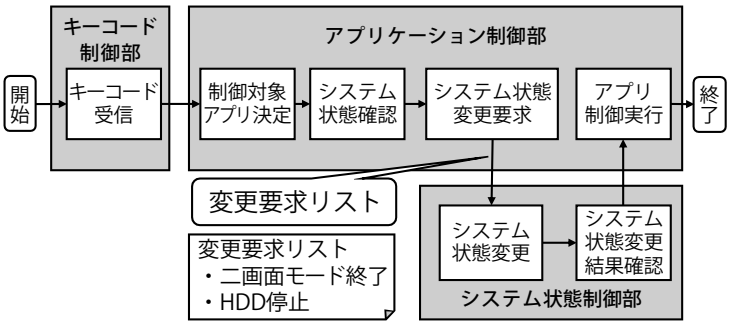


図 11　再編後のアプリケーション制御フロー

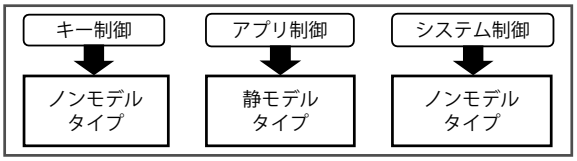


図 12　状態管理部のモデルタイプ

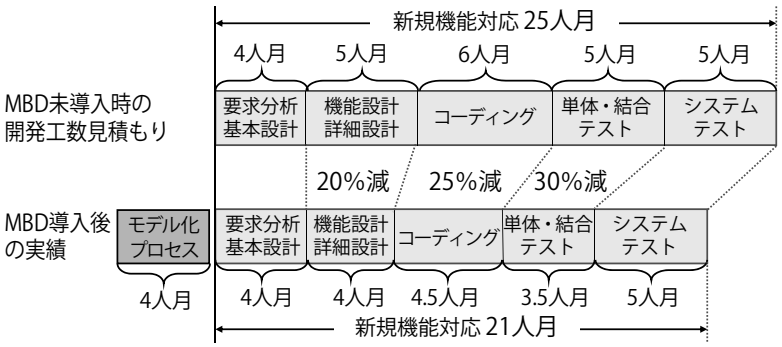


図 13　MBD 導入による工数削減効果

保持するだけのモジュールとなったためノンモデルタイプとなり，モデル化作業を省略する決断をすることができた．アプリケーション制御部においては，「動的な振舞い」のリファクタリングの結果，状態遷移処理が少なく逐次処理が90%以上を占めていたため静モデルタイプとした．以上の判断により，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を導入しない場合の工数は，25人月と見積もられていた．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13**の上段は見積もり工数，下段は実績工数を示している．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に導入した．この作業には，4人月を要した．また，本製品の新規機能に対応するために21人月の実績工数が必要であった．以上のように，モデル化プロセスを導入しても，当初の製品開発工数内で開発を完了させることができた．新規機能の対応に要する工数は25人月から21人月へ削減できたことから，約16%工数削減を実現できたことになる．今回新規に追加した4人月のモデル化プロセスは，次機種以降必要がなくなることから，次機種以降の開発では，今回行ったモデル化による工数削減効果の恩恵を受けることが可能となるであろう．以上の結果から，リファクタリングによって，既存資産に対して効率的なMBDの適用が可能となったといえる．

図13に示す単体・結合テストの工数が30%減少していたことから，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に寄与し，制御フローの統一は主に設計ミスの減少に寄与したと考えられる．この他にも新機能の追加や修正作業等のコーディングの作業量が減少するなど，MBD導入により，各工程で工数削減の効果を観測することができた．

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．さらに，本開発ではMBD用のツール☆1を活用した．このツールは，UML図を入力することにより自動的にソースコードを生成することができる．このツールが自動生成したソースコードには，ツールが管理目的で利用するソースコードも含まれており，これがソースコード量を増加させた要因となっていた．次機種以降，リファクタリングの範囲を拡げ，モデルを洗練化させることによりソースコード量は削減可能であると考えている．

5.3　複雑度

MBD導入前の状態管理部と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のモジュールでも導入前の状態管理部より低くなっている．このことから，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

☆1　IBM Rational(R) Rhapsody(R)

表 2　ソースコードのステップ数

（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

表 3　循環的複雑度の平均値

（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

らかとなった．

5.4　品質

定量的な品質として，コードの規模に対する不具合件数（不具合数/STEP数）を測定した．システム全体の平均と比較し約60%に不具合数を抑えることができた．これは，既存ソースコードをリファクタリングし，モデル化した効果と言える．さらなる品質改善を目指すために不具合内容を分析した結果，MBD導入が原因とされる不具合も多く見られた．MBD導入を起因とする主な不具合を下記に説明する．

- 潜在不具合

  モデル化をするにあたり，アプリケーションの制御フローをリファクタリングした．これは動的な振舞いを変更したことになる．その結果，制御のタイミングも変更されることとなった．この制御タイミングの変更により，引き起こされた不具合が多くあった．これはモデル化前は偶然，正しいタイミングで動作していたことを意味し，今回のモデル化により潜在不具合を露呈させることができた．

- 移植ミス

  本稿では，モデルタイプを設定し，既存資産の再利用を最大限図る方針を択った．ただし，MBD導入にあたり，前述したMBDツールが生成したクラススケルトンにロジックの既存ソースコードを移植する等，設計変更を伴わない移植作業も多く実施しなければならなかった．その際，移植作業者の勘違いにより，意図しない移植が行われ，その結果発生した不具合があった．

- 仕様変更対応

  本稿で紹介したMBDを適用した状態管理部は，DTVの状態管理を集中的に管理するモジュールであった．そのため，アプリケーションの仕様変更等が発生すると，状態管理部も変更しなければならなかった．アプリケーションと，より疎結合になるように改善し，アプリケーションの仕様変更の影響を状態管理部が受けないようにすることは今後の課題である．

6. おわりに

製品開発工数を遵守しながらMBDを導入する目的で，既存ソースコードを有効的に活用するモデル化プロセスを定義した．既存ソフトウェアのリファクタリングをモデル化プロセスの一部とするかは議論の余地があるが，我々はモデル化のための前処理として位置付けた．また，デジタルテレビの状態管理部の開発へ実適用した結果，工数削減および品質向上の効果を，定量的に確認することができた．

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., Arata, B., J. and 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, pp.60-62 (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，Vol.52, No.12, pp.3184-3191 (Dec. 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).

浅田 幸則（非会員）yukinori.asada.xa@hitachi.com
1997 年（株）日立製作所入社．携帯電話や TV 等の組込みソフト開発に従事．2009 年より生産性向上研究に従事, 現在に至る．

大條 成人（非会員）sh-ooeda@hitachi-ad.co.jp
1985 年（株）日立製作所入社．2001 年より組込みソフトの生産性向上研究に従事．2012 年より（株）日立アドバンストデジタルに勤務．

松本 紀子（非会員）toshiko.matsumoto.ad@hitachi.com
2002 年（株）日立製作所入社．携帯電話や TV 等の組込みソフト開発に従事．2008 年より生産性向上研究に従事, 現在に至る．

成川 沙希子（非会員）sakiko.narikawa.bo@hitachi.com
2008 年（株）日立製作所入社．TV やプロジェクタ等の組込みソフト開発に従事．2009 年より生産性向上，ユーザ・インタフェースの研究に従事，現在に至る．

投稿受付：2013 年 5 月 30 日
採録決定：2014 年 2 月 21 日
編集担当：中谷多哉子（筑波大学）