# ソフトウェア設計方法論の開発と適用

**Development and Deployment of Software Design Methodology**

萱嶋　志門　　玉木　裕二
■KAYASHIMA Shimon　■TAMAKI Yuji

**近年，ソフトウェアの規模が大規模・複雑化する一方で，製品サイクルが短くなっている。この状況に対応できるほど開発人員は増えておらず，一人当たりの生産性をより高める必要がある。しかし，現状では，開発コストの増大や品質維持コストの超過が生じている。東芝は，この問題の本質的な原因として，ソフトウェアの一貫した設計方針がないことと考え，(1)設計方針をまとめソフトウェアアーキテクチャを構築する手法を形式化すること，及び(2)ソフトウェアアーキテクチャを形骸(けいがい)化させることなく維持・運用するためのノウハウを蓄積することに対して，技術開発を進めソフトウェアの開発現場への適用を行ってきた。**

In recent years, not only has there been a dramatic increase in the complexity and size of software but also a shortening of the product development cycle. The lack of a proportional increase in the size of software teams has resulted in the necessity to increase productivity. However, this has resulted in a huge increase in development and maintenance costs. Based on our experience, we have come to the conclusion that the essential cause of this problem is the lack of coherent software design policies.

Toshiba has been working in the following areas to solve this issue: (1) procedures to formalize techniques for deciding the design policy and the construction of software architecture based on it, and (2) accumulation of know-how on how to sustain architecture without destroying it.

## 1　まえがき

近年のソフトウェアは，ますます大規模・複雑化してきている。また，ソフトウェアの仕様変更，機能追加の頻度，バリエーションも多くなり，製品開発サイクルも短くなってきている。このような状況により，多くのソフトウェア開発組織では，市場のニーズに追従してソフトウェアをリリースし続けることが非常に困難になりつつある。

この原因として，ソフトウェア設計について次の二つの問題があることに注目した。

⑴　アドホックな開発や変更が行われる。

⑵　開発戦略が立てられない。

これらを踏まえ，東芝は，ソフトウェア設計力強化をミッションに，次の活動を行ってきた。

⑴　最適なソフトウェアアーキテクチャを構築する手法の形式化

⑵　ソフトウェアアーキテクチャを形骸化させることなく，維持・運用するためのノウハウの蓄積

当社はこれまでに，前記手法やノウハウをソフトウェア設計方法論としてまとめてきた。

ここでは，その技術開発と適用について紹介する。なお，ここで述べるソフトウェア設計方法論は，新規にソフトウェアを開発する状況を想定している。

## 2　ソフトウェア開発の現状

携帯電話やデジタルテレビ，カラーコピー機に代表される組込み機器は，周辺技術の発達によって急激に高機能・多機能化してきた。このため，これらの機器に組み込まれ，機器を制御する組込みソフトウェアも，大規模・複雑化してきている。組込みソフトウェアに限らず，あらゆるソフトウェアがこのような傾向にあると言っても過言でない。

また，多くの開発組織では，既存の製品に仕様変更や機能追加を繰り返し，シリーズ展開していくのが一般的である。こうしたシリーズ展開においても，ユーザーニーズの多様化による製品サイクルの縮小化，製品ファミリーの拡大などによって，短納期で開発する必要が生じている。

現状では，こうした状況に対応できるほど，製品開発をするための人員(リソース)が増えていない(**図1**)。したがって，製品に求められる品質を確保しつつ，一人当たりの生産性をいかに高められるかが課題であると言える。

しかし，開発コストが増大したり，品質の維持に予定以上のコストが費やされたりしている。その原因として，ソースコードに対してアドホックな変更が積み重ねられている点に着目した。アドホックであるとは，例えば，あるソースコードを記述する際，妥当な根拠がないまま，あるいは設計方針に照らし合わせることなく，コードを記述してしまう様を指す。アドホックな変更を繰り返すと，要件の作込みに抜けが生じ，

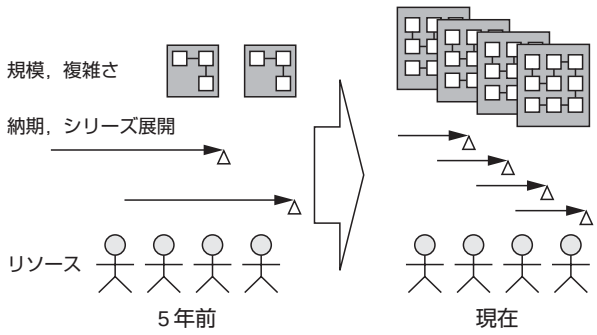


**図1．ソフトウェア開発の現状** ― 製品の高機能・多機能化からソフトウェアが大規模・複雑化し，ユーザーニーズの多様化から製品サイクルが縮小し，製品ファミリーが拡大している。それにもかかわらず開発人員（リソース）は変わっていない。

Actual state of software development

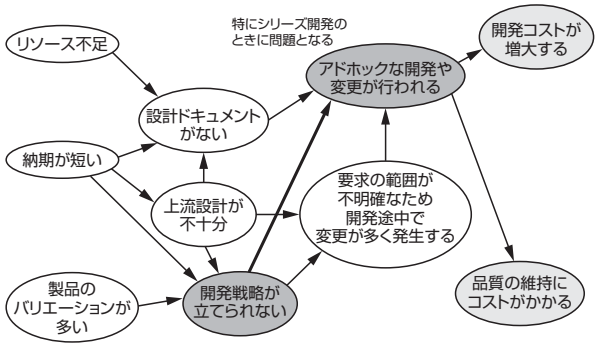


**図2．アドホックな開発・変更が行われる原因** ― アドホックな開発・変更が行われる原因として多くの要因を挙げられるが，主因は一貫した開発戦略がないことであると考えた。

Case of ad-hoc developments and changes

開発途中で変更が多く発生したり，大幅な後戻りが生じたりする。更に，変更するたびに変更箇所を増やし，上流設計段階で決定した構造やメカニズムから逸脱する事態となりかねない。

アドホックな開発や変更が行われてしまう原因として，十分な設計ドキュメントが残されていなかったり，開発途中で仕様変更が発生したりするなどが挙げられる。そのなかでもソフトウェアの一貫した設計方針（開発戦略）がない，あるいは浸透していないことが，その本質的な原因ではないかという結論に達した（**図2**）。

## 3　開発戦略とソフトウェアアーキテクチャ

ソフトウェア設計と言っても，ソフトウェア全体レベルのマクロな設計から，実装レベルの詳細な設計まで，いくつかの段階がある。各段階の設計作業は，いずれも高度な知的作業であり，随所でトレードオフの選択に迫られる場合が多い。元来，ソフトウェア設計は，なぜそうしたのかという根拠に基づいているべきである。根拠に基づいているとは，各段階で設計を行う前に，設計方針を明確化しており，トレードオフの選択もその方針に従っているということである。更に，各段階の設計方針は一貫しているべきである。一貫しているとは，任意の段階の設計方針は，一つ上位の段階で定めた設計方針をブレークダウンしたものになっている，ということである。

これらを満たすためには，まず，最上位の段階で設計方針を明確化し，これに基づいて全体レベルの構造とメカニズムを定めることが必要である。そして，モジュールごとに段階的詳細化をしていくのが望ましい。

最上位の段階での設計方針とは，機能要件及び非機能要件を満たすために，ソフトウェアをどのように構築すべきかを抽象的に表現したものであると考えている。例えば“ハードウェアが変更されてもよいように，ハードウェア依存部分を分離する”が挙げられる。通常，プロジェクトでは，いくつかの方針が挙げられ，場合によっては互いに矛盾する方針が挙げられる。最上位の設計段階における，こうした方針間の矛盾を解消し，あるいは優先度付けしたものを，開発戦略と定義した。また，ソフトウェアアーキテクチャを，この開発戦略を基に定めた，システムの全体レベルの構造とメカニズムであると定義した。

## 4　アーキテクチャ指向設計方法論

当社は，良質のソフトウェアを開発するには，上流での最適な設計が不可欠であり，良いソフトウェアは，良いアーキテクチャの基に構築されるという考えのもと，そのやり方を模索し，議論を重ね，次の内容を体系化する作業を行ってきた。

⑴　開発戦略を決定し，それを基に最適なソフトウェアアーキテクチャを構築する手法（ソフトウェアアーキテクチャ構築手法）

⑵　ソフトウェアアーキテクチャを形骸化させることなく，維持・運用するためのノウハウ（ソフトウェアアーキテクチャ維持・運用）

こうした，開発戦略という視点からのアーキテクチャ構築と，このアーキテクチャに基づいたソフトウェア開発スタイルを，アーキテクチャ指向設計方法論（ACE：Architecture Centric Engineering）としてまとめている。特に，設計に際して“何を作るか”という機能要件に加えて，“どのように作るべきか”という設計方針を重視し，設計工程の上流から下流まで，この設計方針を一貫させることをACEの柱とした。

現在までに，⑴については方法論として形式化し，⑵については事例ベースでノウハウを蓄積した。以下，それぞれについて述べる。

## 5　ソフトウェアアーキテクチャ構築手法

当社は，要件からソフトウェアアーキテクチャを構築するまでのフェーズを次の三つのステップに分け，それぞれについて手順やキーとなる考え方，成果物としての文書化の方法を形式化した（**図3**）。

⑴　機能要件・非機能要件の開発
⑵　開発戦略の決定
⑶　開発戦略に基づくソフトウェアアーキテクチャの構築

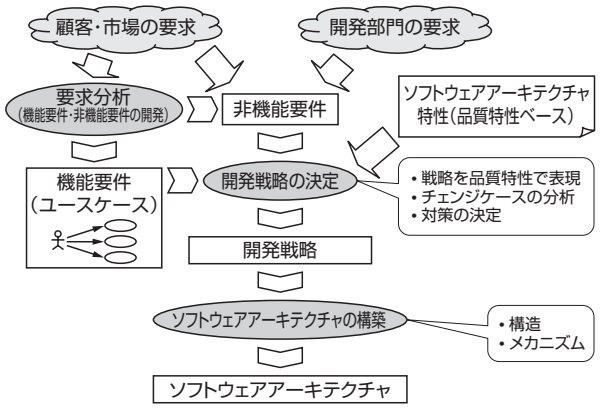


**図3．開発戦略に基づいたソフトウェアアーキテクチャ構築** ― 要件からソフトウェアアーキテクチャを構築するまでのフェーズを，要求分析，開発戦略決定，アーキテクチャ構築の3ステップに分け形式化した。
**Architecture design based on development strategy**

### 5.1　機能要件と非機能要件の開発

ソフトウェアアーキテクチャを構築する前に，“何を作るか（機能要件）”，“どのような品質を満たすべきか（非機能要件）”を明確にし，開発戦略としてまとめる必要がある。これらの要件は，顧客や開発部門が暗黙的に持っている場合が多いので，設計者が意図的に引き出す必要がある。

機能要件とは，これから開発するシステムはどのようなサービスを提供するのかを端的に表現し項目化したものである。顧客の持つ潜在的な要求を，ユースケース分析をベースに引き出し，顧客のほんとうに解決したいことを本質的なサービスとして項目化することを推奨している。これにより，顧客の要件のうち枝葉である変わりやすい部分だけに注力してしまうリスクを軽減できる。

非機能要件とは，移植性や保守性などの品質特性にかかわる要求を，端的に表現し項目化したものである。将来計画や製品展開，開発に割けるリソース制約など，開発部門としての要求も重要であるが，こうした要求は軽視されがちなので，これらを引き出しやすくするためのテンプレートや，各品質特性について一考することができる質問一覧などを整備した。

### 5.2　開発戦略の決定

このフェーズでは，機能要件と非機能要件から，開発するソフトウェアシステムの，最上位の設計方針を定める。この手順を次の4ステップに分割した。

⑴　チェンジケースとWishを洗い出す　　各機能要件について，非機能要件を基に将来の製品展開においてどのように変更や拡張がなされるのか（チェンジケース），開発者としてソフトウェアをどのように作り込んでおきたいのか（Wish）を，次の観点から洗い出す。
　ⓐ　どのような品質特性がかかわるのか
　ⓑ　どのようなプロダクトラインが存在するのか
　ⓒ　ドメインの知識から考慮すべきことはあるのか
　この段階では抜けなく洗い出すことを重視するため，前述の質問一覧を用いて，可能な限り列挙するのが望ましい。

⑵　対策を検討する必要性を判断する　　先の⑴の作業では，実際には発生しえないチェンジケースや，実現するためのコストに見合わないWishが挙がる可能性がある。このフェーズでは，そのようなチェンジケースとWishを排除する。排除すべきかどうかの観点を次に示す。
　ⓐ　発生の頻度はどれくらいか
　ⓑ　対策を立てなかった場合，ソフトウェアのどの程度の構成要素に影響を与えるか
　ⓒ　対策を立てるための根拠は明確になっているか

⑶　対策を検討する　　チェンジケースとWishに対し，どう対応するか，その対策を検討する。対策は，この段階では“OS（Operating System）依存部分を分離する”や“デバッグのためにログを記録するメカニズムを導入する”という程度のものが挙がっていればよい。更に，各対策について，どの範囲まで対処すべきであるかも明確にする。例えば，“OS間の移植を容易にするために，OS依存部分を分離する”であれば，“移植対象のOSは，Linux（注1），Windows®（注2） 2000/XPに限る”などと対策の範囲を限定する。

⑷　同等の対策をまとめ，矛盾を解消し優先度をつける
　似たような対策は，チェンジケースやWishに共通の要素が含まれており，同じように扱える場合がある。そのような同等の対策は，個別に考えるのではなく，まとめて考えれば二度手間がなくなり，矛盾が生じるリスクも回避できる。
　そのような対策をまとめた後，与えられたリソース内でその対策を実現可能か判断する。リソース的に無理

（注1）　Linuxは，Linus Torvalds氏の米国及びその他の国における登録商標。
（注2）　Windowsは，米国Microsoft Corporationの米国及びその他の国における登録商標。

と判断された場合には，必要であれば，その対策の範囲を狭めることなどにより対応する。

次に全体を見渡し，各対策の間で矛盾がないことを確認し，最後に各対策に優先度をつける。この段階で対策間で矛盾が生じていたり，優先度があいまいであったりすると，以降の工程で優先度や対策を取り違えることにつながり，ソフトウェア内にむだや矛盾を生じさせる原因となる。

### 5.3 開発戦略に基づくソフトウェアアーキテクチャの構築

ソフトウェアアーキテクチャを構築する際には，様々な視点から考える必要がある。開発戦略はそのうちの一つと言える。開発戦略によってソフトウェアアーキテクチャの構築手順は異なってくるので，ここでは，汎用的な大枠の方法論だけを提示する。

開発戦略に基づきソフトウェアアーキテクチャを構築するステップを，次の二つに分けて説明する。

⑴ 構造をソフトウェアアーキテクチャに反映する

モジュール構成やクラス構成などレイヤリングやパーティショニングを行うことで開発戦略を満たしていく。この場合，チェンジケースとWishから影響を受けるソフトウェアの構成要素を見つけ，分離する。次に分離した要素間のインタフェースを定め，その分離が実現可能かどうか，他構成要素間との間に不整合が生じないかを確認する。

⑵ メカニズムをソフトウェアアーキテクチャに反映する

ほかのモジュールやクラスとの相互作用を伴う仕組みを構築することで開発戦略を満たしていく。構造と同様，チェンジケースとWishから影響を受けるソフトウェアの構成要素を見つけておく。次にメカニズムを導入するのに必要なソフトウェアの構成要素を見つける。そのメカニズムに必要な相互作用から構成要素が持つべき情報やふるまいを見つける。最後にそのメカニズムが，開発戦略に合致することを確認する。

どちらの場合でも，ある対策を実現するソフトウェアアーキテクチャを構築したら，そのソフトウェアアーキテクチャはほんとうに実現可能か，想定外の構成要素に影響を与えていないか，既存のソフトウェアアーキテクチャと矛盾していないかを確認することが重要である。

最後に，どのような構造にするのか，どのようなメカニズムを導入するのかを文書化する必要がある。文書化は，以降の工程における意思疎通に必要不可欠である。

## 6 ソフトウェアアーキテクチャ維持・運用

一般に，ソフトウェアアーキテクチャを構築した後は，複数のサブチームに分かれ，分担して個々のモジュールの基本設計と詳細設計を行うことになる。このフェーズ以降において，プロセス改善活動を適切に展開していくことの重要性に着目した。その理由として，組織やプロジェクトによって，文化や習慣，スキルが異なるため，開発プロセスも様々なことが挙げられる。言い換えれば，ソフトウェア開発プロジェクトに合わせて，開発戦略や設計方針を理解・浸透できるよう仕向けていくことが重要である。具体的には，プロジェクトが次のような状態になっている必要があることを，経験からつかんだ。

⑴ プロジェクト体制が明確で，それぞれの役割が機能している。
⑵ 進捗（しんちょく）が可視化されている。
⑶ リスクや課題が表面化している。
⑷ 定期的にピアレビューが実施されている。
⑸ エンジニアリングの環境やツールが整備されている。
⑹ 成果物のバージョンを取り違えずに，誰でも参照できるようになっている。

これらを確実なものとするためには，プロジェクト内で，ルールを定めたり，ドキュメントの記述ガイドやサポートツールを作成したり，あるいは，これらの普及・啓蒙（けいもう）を行ったりと，様々な改善活動が必要である。逆に，プロジェクト内で適切な改善活動が行われないと，たとえ適切なソフトウェアアーキテクチャがあったとしても，やがて形骸化し，混乱するプロジェクトになってしまうリスクが高い。

こうした改善活動の進め方についても模索し，次に示すサイクルで改善活動を行うのが効果的であるとの結論に達した（**図4**）。

⑴ 動機付け　プロジェクト内で発生している，あるいは発生する可能性のある問題を探すことなどによって改善点を見つけ，その改善理由を明確にする。

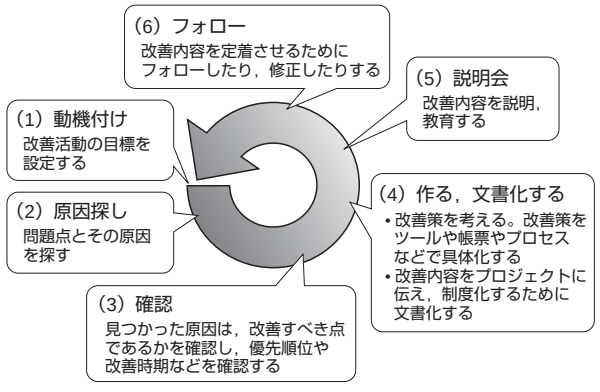


**図4．改善活動のサイクル** ― ソフトウェア開発プロジェクトにおいて，六つのステップで改善活動を進めることが効果的である。
Cycle of improvement activities

⑵　原因探し　　改善点として挙げられた項目に対して，なぜ改善を迫られる状況になってしまうのか，その原因を明らかにする。解決すべき課題の原因を洗い出せたら，それらを分類・整理して優先度をつける。

⑶　確認　　整理した原因に対して，どのような施策をとっていけばよいかを検討し，その改善計画を立案する。これを，プロジェクトマネージャーや主要メンバーに紹介し，優先順位や改善時期などについて合意を得る。

⑷　作る，文書化する　　計画に従って，改善施策を作り込む。具体的には，環境を整備したり，サポートツールを作ったり，文書のテンプレートを作成したりすることなどが挙げられる。また，メンバーへ説明するための補足資料を作成する。

⑸　説明会　　作成したツールや環境，テンプレートなどの説明用に文書化した資料を用いて教育を実施する。メンバーを集め集合教育形式で実施したり，説明用の文書の所在をメールで知らせたり，サブチームリーダーに説明をして各メンバーへの説明はサブチームリーダーに任せたりするなどの方法が考えられる。

⑹　フォロー　　実施状況を調査し，新たな課題がないか，改善されたかどうか，効果があったかどうかを確認する。メンバーにヒアリングして，理解できたか，意義を感じるか，面倒に感じる点はないかなどを確認することも重要である。

## 7　適用事例

### 7.1　ソフトウェアアーキテクチャ構築プロセス

ACEの設計方法論に基づいて，当社ソフトウェア開発部門に対する支援を実施している。支援先部門では，多くのソフトウェア開発者の設計力向上が急務となっており，設計方法論のトレーニングやガイドの作成が必要であった。

そこで，要求分析からアーキテクチャ構築まで，つまり最上位の設計方針をモデルに取り込むフェーズまでを扱う設計トレーニング及びアーキテクチャ設計ガイド，モデリングガイドを開発してきた。

しかし，トレーニングやガイドだけでは，ACEを実際の開発に適用することは難しい。特にアーキテクチャ設計ガイドは，開発作業の際，手順や考え方を確認するために参照し続けるものであるが，次の問題点が出されていた。

⑴　開発者にとって十分詳細化されていない。

⑵　リソースなどとの兼ね合いで適用できない部分がある。

この支援では，開発者が無理なくガイドの手順を実施できるように，支援先部門に特化した形にガイドを改訂することから始めた。具体的には，ある開発プロジェクトにコンサルタントとして参画し，手法に従った様々なアドバイスを実施した。そのなかで，ガイドの疑問点やリソースとの兼ね合いで実施不可能な手順を挙げてもらい，その部分を改訂していった(**図5**)。これを通して開発者が初めてガイドを読んだときに生じる疑問を少しずつなくし，組織内に横展開しやすい内容に改めていくことができた。

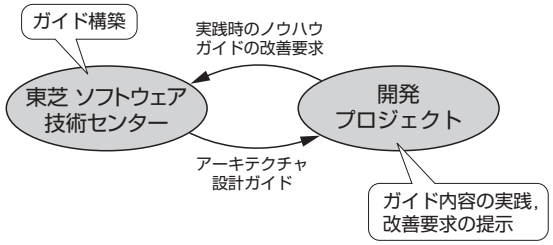


**図5．ガイドの開発形態** ― ガイドの改訂は，実際の開発プロジェクトへのガイド適用時に挙がった疑問点など改善要求のフィードバックを基に実施した。

Guide development team

ガイド改訂後は，週一回の頻度で定例会を開き，実際に作業をした際のガイド上の疑問点や問題点を挙げてもらった。必要に応じてガイドの内容を更にブレークダウンした手順やノウハウを提示した。このように，この支援の役割をガイドの提示，成果物のレビューにとどめ，開発者自身でガイドを基に設計作業をしてもらうことで，開発者の設計力向上及びガイド自体のわかりやすさの向上を図った。

今現在，ソフトウェアアーキテクチャを構築中であるが，開発者サイドから以下のような声が挙がっている。

⑴　これまで早期の段階で考慮していなかった非機能要求や経営，開発サイドの要求を，開発戦略として開発の早い段階から検討し，モデルに盛り込めるようになった。

⑵　ガイドの適用により，数人でソフトウェアアーキテクチャを構築できるようになった。

⑶　文書化しておくことで，各人がアーキテクチャの構造を維持して，詳細設計を展開できる土台ができた。

今後も現在の活動を通じて，ガイドの改訂を進めていく予定である。

### 7.2　ソフトウェアアーキテクチャ維持・運用ノウハウ

ここでは，当社が過去に行ったプロジェクトレベルの改善活動の事例を紹介する。このプロジェクトで開発する製品はリアルタイムOSを利用した専用端末で，利用者が端末を操作することで情報を蓄え，TCP/IP(Transmission Control Protocol/Internet Protocol)で接続された上位機器に情報を送信するための装置である。

**図6**に示すプロジェクト体制の中で支援担当(改善担当)として，プロジェクトの上流工程では前述のソフトウェアアーキテクチャ構築プロセスに基づいてコンサルティングを実施し，中流工程以降は定められたソフトウェアアーキテクチャ

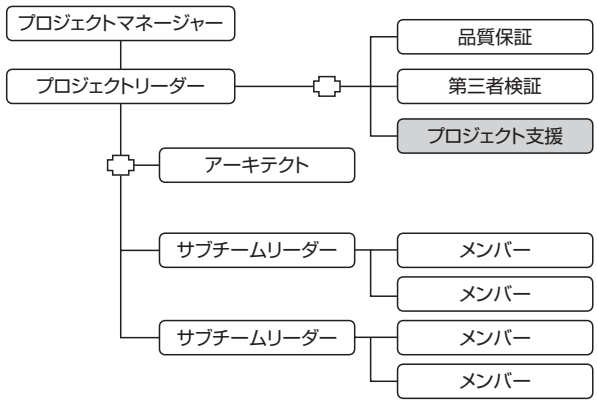


**図 6．支援体制** ― 事例の改善活動を行ったプロジェクトの体制であり，図中の“プロジェクト支援”を当社のソフトウェア技術センターのメンバーが担当した。

**Project team members**

を正しく維持，運用できるよう，様々な改善活動を行った。以下，基本設計書の標準化についての改善事例を紹介する。

⑴ 動機付け　各モジュールの基本設計にさしかかる段階で，数人のメンバーにヒアリングしたところ，基本設計書の書き方に悩むメンバーが少なくなく，基本設計書の書き方や質にバラツキが生じるリスクが高いことが判明した。そこで，同じ考え方に基づいて基本設計を行い，基本設計書の書き方がそろっていることをあるべき姿とすることとした。

⑵ 原因探し　プロジェクトマネージャーやメンバーへのヒアリングを行い，更に，作業が先行しているメンバーと共同で基本設計書を作成する作業を行った。この作業を通して，“基本設計書の書き方”において，具体的には，次のようなことに困っていることが判明した。

ⓐ どこまで詳細化すればよいかわからない

ⓑ どのように表現すればよいかわからない

⑶ 確認　前述の問題点を解決するために，作業が先行しているメンバーに，基本設計書の見本を作ってもらい，そこから基本設計書テンプレートと記述ガイドを作成する案を，プロジェクトマネージャーに提案した。また，見本の作成にあたっては，支援担当もいっしょに検討作業をさせてもらうことで合意を得た。また，基本設計書のレビューを行うときのポイントを整理，文書化することで合意した。

⑷ 作る，文書化する　検討の結果，基本設計書には状態図を中心に記述するのが妥当であるという結論に至った。目次構成や各章での記述内容，記述の仕方についても検討しながら作業し，これを基に基本設計書テンプレートを起こし，基本設計書の記述ガイドとして文書化した。

記述ガイドで，各章に何をどこまで書くかの基準を定義し，レビューの実施基準，レビュー時のポイントも文書化した。更に，基本設計工程の概要を説明するプレゼンテーション資料と，前記のテンプレート・ガイドの読み方，保管場所を説明するプレゼンテーション資料も作成した。

⑸ 説明会　全員教育形式で，作成したプレゼンテーション資料を用いて説明会を実施した。また，説明会での資料は，プロジェクトの共通サーバに置き，いつでも見られることをメンバー全員に通知した。

⑹ フォロー　説明会後，基本設計書のレビューに出席したり，サーバに登録された文書ファイルを閲覧したりして，各メンバーが作成する基本設計書を調査した。この結果，状態図の書き方そのものに対して，理解にばらつきがあることがわかり，新たな課題として，更に改善を行った。

上記以外にも，進捗の可視化，リスク管理の実施，要件管理の実施，不具合管理の実施，試験計画の明確化，ピアレビューの常態化，ソースコードの標準化，デバッグ環境の標準化など，多岐にわたる改善活動を行った。この結果，定められたソフトウェアアーキテクチャが正しく維持，運用され，各モジュールの基本設計書やソースコードのすべてが，プロジェクトで定めた基準に従っていることを確認した。プロジェクトの立ち上げ段階に定めた品質目標も達成することができた。

## 8　あとがき

当社が提唱するソフトウェア設計方法論の基本的な考え方と，これに基づく適用事例について述べた。今後は，方法論を更に洗練させていくとともに，関連する技術の整備，適用実績の拡大をしていきたい。

## 文　献

⑴ 西岡竜大，ほか．“アーキテクチャ指向設計手法 ACE の紹介”．情報処理学会 第136回ソフトウェア工学研究会予稿集．東京，2002-03，p.211－218.

⑵ 玉木裕二，ほか．“組み込みシステム開発における上流設計とプロセス改善”．UML Forum／Tokyo 2005 カンファレンス予稿集．2005-04，p.215－242.

萱嶋　志門　KAYASHIMA Shimon

ソフトウェア技術センター 技術開発担当。ソフトウェアアーキテクチャ設計方法論の研究・開発に従事。情報処理学会会員。

Software Engineering Center

玉木　裕二　TAMAKI Yuji

ソフトウェア技術センター 技術開発担当参事。ソフトウェアの上流設計の技術開発とソフトウェア開発プロジェクトの支援業務に従事。

Software Engineering Center