望ましい設計書への考察
－全体観と設計意図記述、そして伝達の重要性－

テクニカルコンピテンシー部
開発･品質技術チーム
アプリケーションスペシャリスト

白石　利夫

Toshio　Shiraishi
toshio-shiraishi@exa-corp.co.jp

設計書は、下流開発工程への仕様伝達のほか、テスト、保守への入力として重要な意味を持つ。最近、工期短縮とコスト削減の要求、およびお互いに多くの暗黙的共通知識を持つ固定メンバによる開発のためか、設計書に記述すべき内容が省略されている例を見る。この結果、保守の生産性･品質の低下や保守の属人化などの問題が生じていると考える。本稿では、この事象への対策検討の一歩として、設計書に本来記述すべき情報を標準・規格の観点から整理し、省略傾向にある情報、その情報が欠落した場合の問題・リスクを考察するとともに、対策の方向を示す。

1. はじめに

（社）日本情報システム・ユーザ協会の調査[1]によれば、IT投資予算の約60%が保守運用費で、そのうちの39%がソフトウェア保守費および外部委託費であり、また新規投資40%の36%が再構築の開発費である。再構築も適応保守のひとつだと考えると、実にIT投資の約40%が保守にかけられている。

保守のプロジェクトでは、保守対象システム全体に関する知識を、開発で作成したシステム文書として保守へと引継いでいることを前提にして、問題報告や修正要求の形で

要件を「差分」として提示される。また、再構築の場合も、現行システムがもっている機能についての要求は「現行機能保証」という言葉で済まされ、詳細が記述されないケースもある[2]。この意味で、現行システム全体に関する知識が得られるシステム文書は重要である。

一方、同じお客様や同じシステムの開発･保守を、ほぼ同じメンバで担当してきた場合、本来ならば設計書に記述もしくは参照しなければならない情報も、書かなくてもわかる暗黙的「共通知識」となり、かつ工期短縮・コスト削減の要求を受けて、設計書への記述もしくは修正を省略している例を見かける。確かに不要な作業の削減は、工期短縮・コスト削減の基本的手段であり、ある程度の設計書記述省略は一概に悪いこととはいえない。しかし、差分で与えられた要件で変更を繰り返す間に、文書が現状と合わなくなったり、失われたりしているという実態が現実に存在する。そして、最近の傾向として、コスト削減を目的としたオフショア利用に代表されるように、新しいメンバや新しい委託先がプロジェクトに参加することにより「共通知識」という前提が崩れるケースが増えてきている。

本稿では、このような状況を踏まえ、保守の生産性・品質の低下や保守の属人化などの問題への対策検討の一歩として、基本に立ち返って設計書に本来記載すべき情報を調査し、省略傾向にある情報とそれが欠落した場合の問題・リスクを考察し、対策の方向を示す。

2. 設計書に記述すべき情報

設計書は、設計者と、その後続プロセスの設計者、プログラマ、テスタ、保守者との間のコミュニケーション手段である。まず、現状の課題を考える土台として、設計プロセスおよび設計書に関する標準を調査し、後続のプロセスを担当する設計書利用者の立場から見て必要な設計書記述情報を考察する。

2.1. 設計プロセス

ISO/IEC 12207:2008[3]では、テクニカルプロセス群（設計・実装・テストの通常の開発のプロセス）の中で設計に関するプロセスを定義している。システムアーキテクチャ設計、ソフトウェアアーキテクチャ設計、ソフトウェア詳細設計の３プロセスに分けている。表１にそれぞれのプロセスのタスクと出力を示す。

ここで、ISO/IEC 12207:2008が今年の改定で整合性をとった相手であるISO/IEC 15288:2008[4]では、アーキテクチャを、「システムのコンポーネント、コンポーネント間および環境との関係、およびシステムの設計と進化を支配する原理で体現した、システムの基本的構造」と定義している。

「設計と進化を支配する原則」とは、構成要素に割り当てる要件の概略構想、全体に共通する問題に対する設計結果（関心の分離で切り出した技術的問題の解決策が多い。例えばエラー処理機構など）、品質要件を満たすために採用すべき設計手法や設計の評価基準であると考える。これらは通常、プロジェクトの設計標準や設計ガイドとして記述される。

それぞれの設計プロセスの目的、アクティビティー、出力は異なるが、表１において太字で示したように、設計とは、「想定した環境・方法で使用したときに想定した使用目的を達成でき、与えられた要件を満たして、かつ実現可能なもの」という問題に対し、「設計対象を構成する構成要素とそれへの要件、構成要素間および外部とのインタフェースとそれへの要件」として記述した解（実現策）を求めることであると言える。

それぞれの設計プロセスの、構成要素への分解・詳細化のイメージを図１に、また、問題と解との関係を解釈したものを表２に示す。設計が進むにつれ、構造が分解されていき、それに伴いインタフェースも詳細化・具体化される。

なお、ISO/IEC 12207:2008は、システムアーキテクチャ設計プロセスとソフトウェアアーキテクチャ設計プロセスをISO/IEC 15288:2008のアーキテクチャ設計プロセスのインスタンスとしている。しかし、ソフトウェア詳細設計プロセスにはこういった注記がなく、これだけ別のものと認識していると解釈される。

表１　ISO/IEC 12207:2008における設計プロセス

項No	プロセス名	目的	アクティビティ(★)／タスク(○)	出力
6.4.3	システム アーキテクチャ 設計	どのシステム要件をどの構成要素に割り振るかを決める。	★ アーキテクチャを確立 ○ ハードウェア、ソフトウェア、人による運用の3種類のアイテムを識別し、システム要件を割り当てる。 ★ アーキテクチャを評価	・システムアーキテクチャ ・システムの各アイテムに割り当てた 要件 ・各アイテムの、システム内部および外部インタフェース ・システム要件とアーキテクチャとの間の検証結果、各アイテムの要件の、顧客要件へのトレーサビリティ
7.1.3	ソフトウェア アーキテクチャ 設計	実装でき、要件に対して検証できる、ソフトウェアの設計を提供する。	★ ソフトウェアアーキテクチャを設計 ○ ソフトウェアアイテムへの要件を、トップレベルの構造を記述しソフトウェアコンポーネントを識別したアーキテクチャに変換する。ソフトウェアアイテム要件をソフトウェアコンポーネントに割り当てる。 ○ ソフトウェアアイテム外部、およびソフトウェアコンポーネント間のインタフェース をトップレベルで設計する。 ○ DBをトップレベル で設計する。 ○ 初期バージョンのユーザ向け文書を作成する。 ○ ソフトウェア統合のテスト要件とスケジュールを作成する。 ○ ソフトウェアアイテムのアーキテクチャ、インタフェース、DB設計を評価する。 ○ ステークホルダとレビューする。	・ソフトウェアアイテムのアーキテクチャ ・各ソフトウェアアイテムの内部および外部インタフェース ・DB基本設計結果 ・初期バージョンのユーザ向け文書 ・ソフトウェア統合のテスト要件とスケジュール ・ソフトウェアアイテムへの要件とソフトウェア設計との間の整合性とトレーサビリティ確認結果
7.1.4	ソフトウェア 詳細設計	実装でき、要件およびソフトウェアアーキテクチャに対して検証でき、かつコーディング・テスト可能なほど詳細化した、ソフトウェアの設計を提供する。	★ ソフトウェアを詳細設計 ○ 各ソフトウェアコンポーネントについて、詳細設計する。ソフトウェアコンポーネントをソフトウェアユニットを含む低レベルに詳細化する。ソフトウェア要件をソフトウェアコンポーネントからソフトウェアユニットに割り当てる。 ○ ソフトウェアアイテム外部、ソフトウェアコンポーネント間、ソフトウェアユニット間のインタフェースを、他の情報がなくてもコーディングできるレベルに 詳細設計する。 ○ 必要に応じ、ユーザ向け文書を改訂する。 ○ ソフトウェアユニットテストへの要件とスケジュールを文書化する。 ○ ソフトウェア統合のテスト要件とスケジュールを改訂する。 ○ ソフトウェア詳細設計とテスト要件を評価する。 ○ ステークホルダとレビューする。	・作成すべきソフトウェアユニットを記述した、ソフトウェアコンポーネントの詳細設計 ・各ソフトウェアユニットの外部インタフェース ・要件およびアーキテクチャ設計と詳細設計との間の整合性とトレーサビリティ確認結果

しかし、ソフトウェア詳細設計プロセスでは「設計と進化を支配する原則」（設計標準、設計基準）に従って設計を進める面が強いが、システムアーキテクチャ設計でも同様にエンタプライズアーキテクチャとして決めた標準などに従う面がある。他のアーキテクチャ設計では構造を決める面が強いが、ソフトウェア詳細設計プロセスでもソフトウェアユニットという構成要素に分解して内部構造を決めている。これらから、システムアーキテクチャ設計、ソフトウェアアーキテクチャ設計、ソフトウェア詳細設計の３つの設計プロセスは、それぞれ設計対象が異なり、また設計結果の詳細度が概略レベル（トップレベル）とコーディングできるまでの詳細レベル（プログラミング言語という構成要素に分解できる直前）と異なるが、設計プロセスとして同じものに抽象化できる。

全ての設計プロセスでは、解を求めるため、以下のことを行っている。

(1)設計対象に対する要件を分解・詳細化

機能要件などを分解・詳細化する。品質要件については、実現するための機能要件、各構成要素に割り当てるパフォーマンスなどの品質目標、または採用すべき設計手法と設計基準に詳細化する[5)]。

(2)設計対象の構成要素と各構成要素の役割構想を想定

提供サービスの自律性、循環しないコンポーネント間の依存関係、情報隠蔽、結合度や凝集度などといった設計基準を加味して、１構成要素に割り当てるべき要件のグループを想定する。そして、それを実現する役割を持った構成要素を想定する。

(3)構成要素および設計対象外部との協働作業（コラボレーション）を想定

設計対象が提供する機能やインタフェースなど、設計対象に対する要件を満たすために、想定した構成要素間および外部との間でどのようなやり取りをして処理を進めるかを想定する。市販製品や再利用製品を利用するときは、その機能やインタフェースを勘案して、協働作業の内容や、設計対象に対する要件の調整案を作る。

(4)各構成要素のインタフェースと各構成要素への、分解・詳細化した要件の割り当て

(5)設計結果の評価、最適な設計案の選択

次に示す評価を行い、場合によっては設計対象への

要件を調整する。

・各構成要素とインタフェースへの要件は、設計対象への要件を満たすために必要十分か　（つまり、設計対象への要件に対するトレーサビリティ）
・要件は整合性があるか、実現可能か
・先行のアーキテクチャ設計で決めた「設計と進化を支配する原則」を遵守しているか
・想定した環境、方法でこの設計対象を使って、使用目的を達成できるか

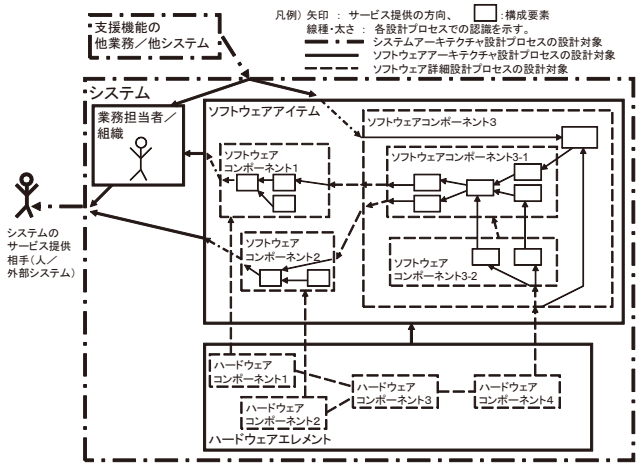


図１　設計の分解のイメージ

2.2. 設計書の標準

ソフトウェアの設計書記述内容に関するガイドとして、IEEE Std 1016-1998[6] がある。

このガイドが各設計対象（設計エンティティ）に対して記述すべきとしている設計書記述情報（属性）を表３に示す。また、推奨している設計ビューとそれが含む記述情報を、表４に示す。

IEEE Std 1016-1998は、設計エンティティのタイプ例としてプログラムやモジュールをあげている。また、推奨している設計ビュー「詳細記述」で「プログラマが実装の前に必要とする詳細度」を要求しているので、このガイドは詳細設計段階に至った設計書が対象であると考えられる。

しかし、設計エンティティ自体の説明ではシステム、サブシステム、データストアも含めていて、ソフトウェアアーキテクチャ設計などにも使用できそうである。2.1節で示した設計の３プロセスのそれぞれの設計対象を設計エンティティとみなして解釈した、設計対象粒度別の設計書記述情

表２　各設計プロセスの問題と解

			システムアーキテクチャ設計	ソフトウェアアーキテクチャ設計	ソフトウェア詳細設計
問題	設計対象		システム（人・ハードウェア含む）	ソフトウェアアイテム	ソフトウェアコンポーネント
	使用者		システムのサービス提供相手	システム	そのソフトウェアコンポーネントを含むソフトウェアアイテム
	使用目的		サービス提供を受け、実現したいこと	システム要件の実現	ソフトウェアアイテムに割り振られた要件の実現
	使用方法、使用環境		使用目的を達成するために、サービス提供相手が実施する作業と作業環境	システム内部の協働作業 システムの稼動環境	ソフトウェアアイテム内部の協働作業
	要件・制約		システム要件	ソフトウェアアイテムに割り振られた要件	ソフトウェアコンポーネントへの要件
解	構造および協働作業参加者	内部構成要素	ソフトウェアアイテム、担当の人・組織、ハードウェアエレメント	ソフトウェアコンポーネント	ソフトウェアユニット
		外部	システムのサービス提供相手 支援を提供する他業務／他システム	システムのサービス提供相手 支援を提供する他業務／他システム 担当の人・組織、ハードウェアエレメント（稼働環境のコンポーネント） 他のソフトウェアアイテム	ハードウェアエレメント（稼働環境のコンポーネント） 他のソフトウェアコンポーネント、 他のソフトウェアアイテム
		つながり・インタフェース	他業務・他システムなどとのインタフェース 内部構成要素（ソフトウェアアイテム、担当の人・組織、ハードウェアエレメント）間インタフェース	他業務・他システムなどとのインタフェース 担当の人・組織やハードウェアとのインタフェース 内部のソフトウェアコンポーネント間インタフェース	他システムとのインタフェースなど 担当の人・組織やハードウェアとのインタフェース 他のソフトウェアコンポーネントとのインタフェース 内部のソフトウェアユニット間インタフェース
	処理もしくは内部協働作業		システムへの各機能要件・品質要件を実現するための、インタフェースを介した、参加者間の協働作業	設計対象ソフトウェアアイテムに割り振られた要件を実現するための、インタフェースを介した、参加者間の協働作業	設計対象ソフトウェアコンポーネントに割り振られた要件を実現するための、インタフェースを介した、参加者間の協働作業
	割り振った要件		各ソフトウェアアイテム、担当の人・組織、ハードウェアエレメントに割り振った要件	ソフトウェアコンポーネントへの要件	ソフトウェアユニットへの要件（仕様）
具体度レベル			概略（トップレベル）	概略（トップレベル）	詳細（コーディング可能な具体さ）

表３　設計エンティティの属性（IEEE Std 1016-1998）

属性		内容	説明
識別子		エンティティの名前	同じ名前のエンティティがあってはならない。
タイプ		エンティティの種類	エンティティの性質を記述。たとえば、サブプログラム、モジュール、手続き、プロセス、データストア
外から見た記述	目的	エンティティの存在理由	このエンティティ作成の目的である、特定の機能要件やパフォーマンス要件を記述。 ソフトウェア要件仕様にない、このエンティティが満たすべき特別な要件も記述。
	機能	エンティティが何をするか	入力から出力を作る変換を記述。データエンティティの場合は、エンティティが蓄積／送付する情報種類。
	インタフェース	他エンティティがこのエンティティとインタラクトする方法。	メソッドとインタラクションを支配するルールを記述。方法には、エンティティの起動や割り込み機構、パラメータ・共有データ・メッセージ・内部データへの直接アクセスなどの機構を含む。ルールには、通信プロトコル、データフォーマット、受け入れ可能な値、各値の意味を含む。入力値の範囲、入出力の意味、タイプおよびフォーマット、エラーコードを記述。情報システムの場合は、入力、画面フォーマット、インタラクティブ言語。
	依存	他エンティティとの関係。	「use」または「require the presence of 」の関係を識別。タイミングや条件などを含む、インタラクションの性質を記述。インタラクションは、初期化、実行順序、データ共有、生成、デュプリケーション、使用、ストレージ、消滅などでもよい。
	リソース	このエンティティが使用する、設計の外部のエンティティ	このエンティティが機能を遂行するために必要な、設計対象外の全リソースを識別し記述。物理デバイス（プリンタ、ディスクパーティション、メモリバンク）、ソフトウェアサービス（算術ライブラリ、 OSサービス）、およびプロセッシングリソース（CPUサイクル、メモリアロケーション、バッファ）。デッドロックについても記述することが望ましい。
内部の記述	従属物	エンティティを構成する全エンティティの識別子	あるエンティティの「composed of 」関係を識別。設計エンティティに対する要件のトレースと、システム分解構造での親子関係を識別するのに使用。
	処理	機能を達成するために使用するルール	特定のタスクを遂行するためのアルゴリズムの記述。タイミング、イベントまたは処理の順番、プロセス起動の前提条件、イベントの優先順、処理レベル、処理ステップ、分岐条件、ループ条件を含むことが望ましい。また例外処理の記述では、オーバフローや妥当性チェックエラーの時にとるアクションを記述することが望ましい。
	データ	エンティティ中のデータ要素	内部データの、表現方法、初期値、使用内容、意味、フォーマット、および受け入れ可能な値を記述。ファイル構造、配列、スタック、キュー、メモリパーティションといった、データ使用も含む。内部データもしくはデータ構造について、このエンティティで使用するためのすべてを記述。データ要素の意味と使用内容に関しては、静的／動的の区別、トランザクションで共有するかどうか、制御パラメータか、値、ループカウンタ、ポインタもしくはリンクフィールドなのかといった記述も含む。プロセスにとって必要な妥当性チェックについても記述。

設計エンティティに対する説明
「構造的にも機能的にも他のエンティティとは区別でき、分離して名前をつけ参照できる設計の要素（コンポーネント）。
ソフトウェアシステム要件の分解から得られる。他のエンティティへの最小限の影響で検討・実装・変更・テストできるような、
別々のコンポーネントにシステムを分けるのが目的。システム、サブシステム、データストア、モジュール、プログラム、プロセス。」

表４　推奨する設計ビューおよび記述情報との関係（IEEE Std 1016-1998）

設計ビュー		分解記述	依存関係記述	インタフェース記述	詳細記述
スコープ		システムの設計エンティティへの分解	エンティティおよびシステムリソースの間の関係	システムを構成する設計ティティを使用するために必要な情報全て	設計エンティティ内部の設計の詳細
表現方法例		階層分解構造図 自然言語	ストラクチャ・チャート DFD トランザクションダイアグラム	インタフェースファイル パラメータ表	フローチャート N-Sチャート PDL
使用例		★設計者・保守者－－主要な設計エンティティの識別（特定の機能実施の責任があるエンティティ）（要件の設計エンティティへのトレース） ★PM－－計画、監視と制御 ★構成管理者－－構成、トラッキング、変更管理 ★QA者－－要件トレーサビリティマトリクス構築	★要件や設計の変更のインパクト評価のための、システムがどう動くかの全体像提供 ★システム故障箇所やリソースボトルネックの分離 ★システム統合計画や統合テストの基となる情報	★設計者、プログラマ、顧客、テスタの間の契約。設計エンティティの詳細設計に進む前に必要な合意を提供。 ★テクニカルライタがユーザ用文書作成に利用（ヒューマンインタフェース）	★プログラマが実装の前に必要とする詳細情報。 ★ユニットテスト計画の基となる情報
含む情報	識別子	○	○	○	○
	タイプ	○	○		
	目的	○	○		
	機能	○		○	
	インタフェース			○	
	依存相手		○		
	リソース		○		
	従属物	○			
	処理				○
	データ				○

凡例）　○　：その行の情報をその列の設計ビューに記述する。

報を表５に示す。

さらに、設計書全体としてはアーキテクチャも記述する必要がある。「分解記述」や「依存関係記述」のビューを見ればわかるように、2.1節のアーキテクチャの定義のうち、「コンポーネントとコンポーネント間および環境との関係」は、これらの設計書記述情報で記述できている。残りの「設計と進化を支配する原則」の情報として、以下の３項目を追加すべきと考える。

(1)構成要素に割り当てる要件の概略構想

(2)全体に共通する問題に対する設計結果

(3)品質要件を満たすために採用すべき設計手法や設計の評価基準

2.3. 利用者の立場から見た設計書に必要な情報

後続の設計、テスト、保守の各プロセスで必要な情報は何か、2.2節で示した設計書記述情報のどれを利用するかを考察する。

2.3.1. 後続の設計プロセスでの利用

2.1節で示した、設計プロセスでの作業内容から考えると、後続の設計プロセスで必要な情報は下記のものとなる。

(1)設計対象に割り当てられた要件

機能要件、品質要件、インタフェース要件、使用すべき設計対象外部コンポーネントまたはリソースという制約

(2)先行するアーキテクチャ設計で設計され、設計ガイド・設計標準などで与えられる「設計と進化を支配する原則」

(3)設計評価のための下記の情報

・上位コンポーネントへの要件　（この設計対象の「使用目的」）

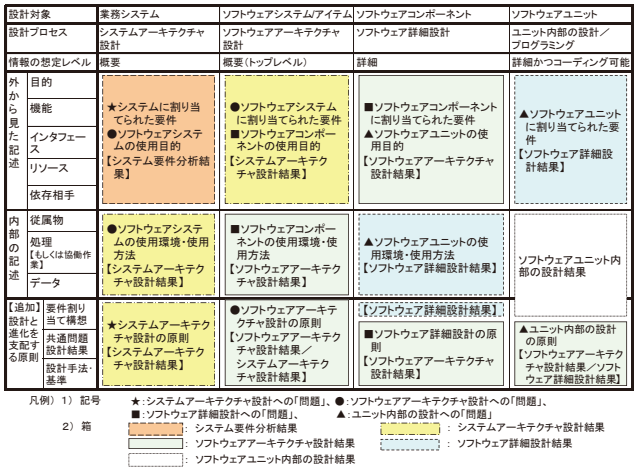
・設計対象に想定した使用環境、使用方法

それぞれの設計プロセスでの上記の情報および設計結果を表５にマッピングしたものを表６に示す。ある設計対象Ａの「割り当てられた要件」と想定した「使用環境、使用方法」は、１つ前の設計プロセスの設計対象のうち、Ａを使

表５　IEEE Std 1016-1998を解釈した設計対象粒度別の設計書記述情報

		業務システム	ソフトウェアシステム/アイテム	ソフトウェアコンポーネント	ソフトウェアユニット
情報の想定レベル		概要	概要(トップレベル)	詳細	詳細かつコーディング可能
外から見た記述	目的	業務の目的、業務の機能要件、品質要件	ユーザがソフトウェアシステムを使用する目的、機能要件、品質要件	ユーザおよび上位ソフトウェアアイテムが、このソフトウェアコンポーネントを使用する目的、このソフトウェアコンポーネントへの機能要件、品質要件	ユーザおよび上位コンポーネントが、このソフトウェアユニットを使用する目的、このソフトウェアユニットへの機能要件、品質要件
	機能	業務への入力を業務外への出力に変換する内容 (ビジネスルール)	ソフトウェアシステムが入力を出力に変換する内容 (ソフトウェアがになうビジネスルール)	ソフトウェアコンポーネントが入力を出力に変換する内容 (各ソフトウェアコンポーネントがになう、部分的ビジネスルール)	ソフトウェアユニットが入力を出力に変換する内容 (各ソフトウェアユニットがになう、部分的ビジネスルール)
	インタフェース	業務にとってのお客様と業務とのインタフェース	ユーザのソフトウェアシステム使用内容、他システムへの情報提供内容	ユーザインタフェース、他システムへの情報提供インタフェース、他コンポーネントへのインタフェース	稼働環境のAPI(ユーザや他システムへのインタフェースの実現手段)、他ユニットへのインタフェース
	リソース	この業務システムを支援する他の業務、他システム	サポートの人、他システム、稼働環境	サポートの人、他システム、稼働環境、再利用ソフトウェア	サポートの人、他システム、稼働環境、再利用ソフトウェア
	依存相手	リソースと同じもの	リソースと同じもの	リソースと同じもの、および他ソフトウェアコンポーネント	リソースと同じもの、および他ソフトウェアコンポーネント、他ソフトウェアユニット
内部の記述	従属物	人、ハードウェア、 ソフトウェアアイテム	ソフトウェアコンポーネント	ソフトウェア(サブ)コンポーネント、 ソフトウェアユニット	メソッド、処理ブロック
	処理 【もしくは協働作業】	業務への入力を出力に変換する計算手順 【人、他業務、他システム、およびハードウェア、ソフトウェアアイテムの間の協働作業】	ソフトウェアシステムが入力を出力に変換する計算手順 【人、他システム、およびソフトウェアコンポーネント(DBを含む)の間の協働作業】	ソフトウェアコンポーネントが入力を出力に変換する計算手順 【他システム、ソフトウェアコンポーネント(DBを含む)、および他ソフトウェアユニットの間の協働作業】	ソフトウェアユニットが入力を出力に変換する計算手順 【他ソフトウェアユニット、および自メソッドの間の協働作業】
	データ	業務で保管している情報	ソフトウェアシステム内の保管情報	ソフトウェアコンポーネント内の保有データ	ソフトウェアユニット内の保有データ
【追加】設計と進化を支配する原則	要件割り当て構想	人・ハードウェア・ソフトウェアの役割分担構想	コンポーネントへの要件割り当て方針	サブコンポーネントやユニットへの要件割り当て構想	
	共通問題設計結果		ユーザインタフェース設計方針、品質要件実現方針	技術的機構と、仮定されるソフトウェアサブコンポーネント／ソフトウェアユニット構造、 デザインパターンなど	
	設計手法・基準		採用すべき設計手法、目指すべき「良い設計」の基準		

表６　設計対象粒度別の設計書記述情報と設計プロセスに必要な情報との対応



用すると設計したＢの設計結果であり、Ａに対する「使用目的」はＢがＡを使用して満たそうとしている要件になる。

2.3.2.　テストプロセスでの利用

ISO/IEC 12207:2008のテクニカルプロセス群の中で、テストが関係するプロセスには、ソフトウェア構築、ソフトウェア統合、ソフトウェア適格性確認テスト、システム統合、システム適格性確認テスト、およびソフトウェア受入サポートがある。表７に、それぞれの目的とテスト内容を示す。

なお、インタフェースの整合性確認が明記されているのはシステム統合プロセスだけであるが、CMMI®7）でインタフェースの変更管理と整合性確認を重視しているように、ソフトウェア統合でもインタフェースの整合性確認を行うべきである。

表７の中に太字で示した、統合戦略、テスト環境（運用環境もしくはそれと等価なプラットフォーム）、評価基準（要件に対するテスト､設計に対するテスト､インタフェースの整合性確認）、および回帰テスト戦略について、必要な設計書記述情報を表８に示す。

統合戦略は、どの部分からどの順番で統合しテストしていって、最終的な製品を作るかに関する戦略である。手戻りを少なくするため、クリティカルな要件から確認できるようにするほか、必要なスタブやドライバの量などを勘案する。そのために必要な情報は、各設計対象の機能要件・品質要件を示す「目的」、総合すると全体構造と必要なリソースを示す「従属物」、「リソース」、および「依存相手」である。

テスト環境は、テスト対象のソフトウェアが稼働環境での動作をシミュレートできることが必要である。その検討のために必要な情報は、稼働環境という「リソース」である。

要件に対するテストは、ブラックボックステストであり、インタフェースを介して入力に対する出力（機能、ビジネスルール）を確認し、機能要件・品質要件を満たすことを確認する。そのために必要な情報は、「目的」、「機能」、および「インタフェース」である。

設計に対するテストは、ホワイトボックステストであり、テスト対象の内部の動き（処理手順、協働作業の様子）や

表７　ISO/IEC 12207:2008におけるテストが関係するプロセス

節No	プロセス名	目的	テスト内容	その他、特記事項
7.1.5	ソフトウェア構築	ソフトウェア設計を適正に反映した、実行可能なソフトウェアユニットを作る。	・ソフトウェアユニット(およびDB)の、**要件**および**設計**に対するテスト	
7.1.6	ソフトウェア統合	ソフトウェアユニットおよびソフトウェアコンポーネントを結合して、ソフトウェア設計と整合性があり、**運用環境もしくはそれと等価なプラットフォームで** 機能要件および非機能要件を満たすことを示す、ソフトウェアアイテムを作る。	・ソフトウェアアイテムに割り当てた**要件**を満たすことを確認するよるテスト	・ソフトウェア設計と要件の優先順位に整合した**統合戦略** ・関連する要件・設計・コードを含めたソフトウェアユニットの変更に対する、**回帰テスト戦略**
7.1.7	ソフトウェア適格性確認テスト	統合したソフトウェア製品が、定義してある要件を満たすことを確認する。	・統合したソフトウェアがソフトウェア**要件**を満たすことを示すテスト	・ソフトウェアアイテムの変更に対する、**回帰テスト戦略** ・運用および保守の可能性を評価
6.4.5	システム統合	ソフトウェアアイテム、ハードウェアアイテム、人による運用、他システムといった、システムエレメントを統合し、システム設計およびシステム要件に記述した顧客の期待を満たす、完全なシステムを作る。	・システムエレメント間の**インタフェースを含め**、システムエレメントに割り当てたシステム**要件**に対するテスト	・システム要件の優先順位に従った**統合戦略** ・変更を行った場合の**回帰テスト戦略**
6.4.6	システム適格性確認テスト	各システム要件が実装され、引渡し準備が整っていることを確認する。	・システム**要件**に対するテスト	
6.4.8	ソフトウェア受入サポート	製品が要件にあっていることを獲得者が確信する作業の補助を行う。	・発注者が行う、受入レビューおよびテスト	

データの生成・参照・消滅の動きを確認する。これには、共通に採用すべき設計結果による動きも含まれる。そのために必要な情報は、「処理【もしくは協働作業】」、「データ」、「従属物」、「および共通問題への設計結果」である。

インタフェースの整合性確認では、依存側と依存相手側とで実装しているインタフェースの整合性を確認する。そのために必要な情報は、「依存相手」、「リソース」、「従属物」、それらが提供する「インタフェース」、および依存側でのインタフェース使用内容として「処理【もしくは協働作業】」である。

回帰テスト戦略は、修正で思わぬ影響が発生していないことを検証する回帰テストを行う範囲を、修正した部分と程度により決める戦略である。思わぬ影響発生を見逃すリスクの軽減利益とテストに必要なリソースコストとのトレードオフを計り、回帰テストを行う範囲を決める。影響がありえる範囲を把握するという意味で「依存相手」、「従属物」、「リソース」、およびそれらのインタフェース、そして影響の重大度を判断する意味で「目的」を必要情報とした。

2.3.3. 保守プロセスでの利用

ISO/IEC 12207:2008では、ソフトウェア保守プロセスを、引渡し後のシステムのサポートと位置づけ、表９に示す５つのアクティビティに分けている。

「プロセスを実装」アクティビティは保守活動を始める準備であり、実際の個別の保守活動は、問題報告または修正要求を受け取った時点から「問題および修正を分析」アクティビティとして始まり、「移行」アクティビティで終わる。ここで、「修正を実装」アクティビティの実際の修正作業はテクニカルプロセス群に従うとされている。

保守には、要件は変わらず欠陥に対応する是正保守や予防保守と、要件の変更・追加がある完全化保守や適応保守がある。保守では、全くの新規開発と比較して下記の特徴があり、それに伴う前提知識が必要になる。この特徴を前提知識とともに以下の(1)から(5)に示す。

(1)ニーズや要件が、差分で語られる。（問題報告、修正要求）

⇒　既存システムへのニーズや要件の全体像を知っている必要がある。

表８　設計対象粒度別の設計書記述情報とテストプロセスに必要な情報との対応

設計書記述情報		内容	統合戦略	テスト環境	評価基準			回帰テスト戦略
					要件	設計	インタフェース	
外から見た記述	目的	設計エンティティの存在理由。 このエンティティで満たす機能要件・品質要件	○		○			○
	機能	設計エンティティが何をするか。入力から欲しい出力を作る変換内容			○			
	インタフェース	他のエンティティがこの設計エンティティにインタラクションする方法、それを支配するルール			○		★（依存相手インタフェース）	★（依存相手インタフェース）
	リソース	設計エンティティが、機能を果たすために使用する、設計対象外のエンティティ	○	○（稼働環境）			○	○
	依存相手	設計エンティティの「uses」または「requires the presence of」の関係相手	○				○	○
内部の記述	従属物	設計エンティティの「composed of」関係相手	○			○	○	○
	処理 【もしくは協働作業】	入力を出力に変換するために使用するルール 特定のタスク遂行のためのアルゴリズム 【機能を実現するために行う、構成要素間および設計エンティティ外部との協働作業の動き】				○	○（インタフェース使用）	
	データ	設計エンティティ内部データの記述				○		
【追加】設計と進化を支配する原則	要件割り当て構想	構成要素に割り当てる要件の概略構想						
	共通問題への設計結果	共通に採用すべき設計結果				○		
	設計手法・基準	採用すべき設計手法、良い設計の基準						

凡例）○　:テスト対象の設計エンティティに関する必要情報　　★　:他の設計エンティティに関する必要情報

(2)組織（ユーザ業務）や接続する他システム、既存システムに与える影響などを概略検討し、修正規模、対応オプションなどを検討する。

⇒　概略レベルの検討に必要な程度に、ユーザ業務、接続する外部システムとインタフェース概要、および既存システムのアーキテクチャを知っている必要がある。

(3)既存の設計が存在し、要件の変更・追加から詳細な変更箇所を特定する。

⇒　要件の分解・詳細化の構造および構成物への要件割り当てを知っている必要がある。

(4)既存のアーキテクチャが存在し、構造を劣化させないようにこれを尊重して修正する。

⇒　既存のアーキテクチャを知っている必要がある。

(5)変更・追加された要件だけでなく変更されない要件も、思わぬ影響を受けていないか確認する必要がある。

⇒　既存システムの要件の全体像を知っている必要がある。そしてテストプロセスに関連して表８で示した回帰テスト戦略を作るための情報も必要である。

ISO/IEC 12207:2008では、保守作業の前提になっている上記の情報は、引渡し時にシステム文書として伝達されると想定されている。これらの情報がシステム文書に記載され、保守担当者に伝達されている必要がある。

特徴(1)～(5)に関して、ソフトウェアシステム（ソフトウェアアイテム）の保守という視点から、前提となっている情報を2.2節の表５にマッピングしたものを表10に示す。

3. 設計書記述省略の現状とリスク

２章で、設計書に記述されるべき情報と、後続の設計プロセス、テストプロセスおよび保守プロセスでの利用内容を考察した。本章では、記述省略や不十分な記述の例を示し、これが引き起こすリスクを、２章で整理した内容に照らし合わせ考察する。

設計書はコミュニケーションの道具であり、コミュニケーションは共通知識という土台の上に成り立っている。この土台となっている知識全ては書ききれないものであり、省略されることが通常である。

コミュニケーション相手と共通知識化した情報は、書か

表９　ISO/IEC 12207:2008におけるソフトウェア保守プロセス

目的	引渡したソフトウェア製品に対して、費用対効果のよいサポートを提供する。		
出力	・ 修正およびリリース戦略に従った移行を管理する保守戦略 ・ 既存システムの変更の、組織、運用、インタフェースへの影響 ・ 必要に応じて更新した、影響を受けるシステムとソフトウェア文書 ・ 修正した製品（要件を損なっていないことを示すテスト付き） ・ 顧客環境に移行した、改良した製品 ・ 修正に関する、影響を受ける人とのコミュニケーション		
アクティビティとタスク（○）	6.4.10.3.1	プロセスを実装	○ ソフトウェア保守プロセスを実施する計画と手続きを確定し、文書化し、実施する。 ○ ユーザから問題報告・修正要求を受領し、記録し、追跡し、ユーザにフィードバックする手続きを確立する。 ○ 既存システムへの修正を管理するための、構成管理プロセスを確立する。
	6.4.10.3.2	問題および修正を分析	○ 組織、既存システム、インタフェースがあるシステムに対する影響 と、保守タイプ（例えば、是正、予防、完全化、適応）、スコープ（修正規模、コスト、必要期間）、緊急度（例えば、パフォーマンス、安全性、セキュリティへの影響）について、問題報告・修正要求 を分析する。 ○ 問題を再現もしくは検証する。 ○ 分析に基づき、修正を実装するためのオプション を開発する。 ○ 問題／修正要求、分析結果、修正実装オプションを文書化する。 ○ 契約に基づき、選択したオプションの承認を得る。
	6.4.10.3.3	修正を実装	○ 分析を行って、修正すべき文書、ソフトウェアユニット、バージョンを決定 し、文書化する。 ○ テクニカルプロセス群を使用して、修正 を行う。【修正した部分だけでなく、修正しない部分もテスト。修正がない要件に影響がないことも確認 。評価基準、テスト結果は文書化。】
	6.4.10.3.4	保守レビュー／受入	○ 修正をオーソライズした組織と、修正したシステムが完全かどうかを決めるためレビューする。契約どおりに修正を完了したことの承認を得る。
	6.4.10.3.5	移行	○ ユーザを交えて移行計画を立て、文書化し、実施する。 ○ ユーザに、移行計画とアクティビティを通知する。 ○ 新環境と旧環境で並行稼動をしてもよい。この間に必要な教育訓練を実施する。 ○ 新環境に変わるインパクトを評価するため、運用後のレビューを実施する。 ○ 契約にデータ保護やデータ監査可能という要求があれば、旧環境で使用した、または旧環境に関連するデータを、アクセス可能な状態にする。

なくても問題として顕在化せずに作業は進む。問題となるのは、新しいメンバの参加や新しい委託先への外部委託など、共通知識という仮定が崩れるときである。

詳細設計以降を外部委託した場合に、思わぬ構造になっていたとか、あまりにも長く複雑なメソッドにコーディングされていたなどの問題事例を聞くことがある。これは記述省略の習慣が問題として、新しい委託先の場合に顕在化したことが一因ではないかと推測している。

3.1. 記述省略・不十分な記述

次のような情報を省略している例を見かける。省略する主な理由は、長年、同じシステムを同じメンバで保守してきた中で、文書化されない共通知識になっているためと思われる。

(1)ユーザの業務、開発・保守対象のシステムの使用環境と使用目的の省略

(1.1)ユーザ業務フロー、ソフトウェアシステムの使い方と使用目的

保守を長く継続しているプロジェクトや、内部設計以降を請負っていて画面設計結果を入力としているプロジェクトで、ユーザインタフェースがあるにもかかわらずこの情報を省略している場合がある。共通知識化しているか、または与えられた仕様どおりのシステムを構築することのみを意識しているためであると思われる。

(1.2)稼働環境もしくは指定された最終テスト環境、コンポーネントの配置

長く同じ稼働環境を使用しているプロジェクトで、「いつもの環境」として省略していることが多い。

(2)全体構造の参照文書を明示しないまま、変更部分のみ記述

(2.1)外部システムとの接続関係、インタフェース

保守を長く継続しているプロジェクトで、全体像

表10　IEEE Std 1016-1998設計書記述情報の設計エンティティ粒度別解釈と保守の前提情報との対応

		業務システム	ソフトウェアシステム/アイテム	ソフトウェアコンポーネント	ソフトウェアユニット
情報の想定レベル		概要	概要(トップレベル)	詳細	詳細かつコーディング可能
外から見た記述	目的	(1) (3) (5) 業務の目的、業務の機能要件、品質要件	(1)(2)(3) (5) ユーザがソフトウェアシステムを使用する目的、機能要件、品質要件	(3) (5) ユーザおよび上位ソフトウェアアイテムが、このソフトウェアコンポーネントを使用する目的、このソフトウェアコンポーネントへの機能要件、品質要件	(3) (5) ユーザおよび上位コンポーネントが、このソフトウェアユニットを使用する目的、このソフトウェアユニットへの機能要件、品質要件
	機能	(1) 業務への入力を業務外への出力に変換する内容(ビジネスルール)	(1)(2) (5) ソフトウェアシステムが入力を出力に変換する内容(ソフトウェアがになうビジネスルール)	(3) (5) ソフトウェアコンポーネントが入力を出力に変換する内容(各ソフトウェアコンポーネントがになう、部分的ビジネスルール)	(3) (5) ソフトウェアユニットが入力を出力に変換する内容(各ソフトウェアユニットがになう、部分的ビジネスルール)
	インタフェース	(1) 業務にとってのお客様と業務とのインタフェース	(1)(2) (5) ユーザのソフトウェアシステム使用内容、他システムへの情報提供内容	(3) (5) ユーザインタフェース、他システムへの情報提供インタフェース、他コンポーネントへのインタフェース	(3) (5) 稼働環境のAPI(ユーザや他システムへのインタフェースの実現手段)、他ユニットへのインタフェース
	リソース	(1) この業務システムを支援する他の業務、他システム	(1)(2) (5) サポートの人、他システム、稼働環境	(3) (5) サポートの人、他システム、稼働環境、再利用ソフトウェア	(3) (5) サポートの人、他システム、稼働環境、再利用ソフトウェア
	依存相手	(1) リソースと同じもの	(1)(2) (5) リソースと同じもの	(3) (5) リソースと同じもの、および他ソフトウェアコンポーネント	(3) (5) リソースと同じもの、および他ソフトウェアコンポーネント、他ソフトウェアユニット
内部の記述	従属物	(2) 人、ハードウェア、ソフトウェアアイテム	(2)(3) (5) ソフトウェアコンポーネント	(3) (5) ソフトウェア(サブ)コンポーネント、ソフトウェアユニット	(3) (5) メソッド、処理ブロック
	処理 【もしくは協働作業】	(1)(2) 業務への入力を出力に変換する計算手順 【人、他業務、他システム、およびハードウェア、ソフトウェアアイテムの間の協働作業】	(2) (5) ソフトウェアシステムが入力を出力に変換する計算手順 【人、他システム、およびソフトウェアコンポーネント(DBを含む)の間の協働作業】	(3) (5) ソフトウェアコンポーネントが入力を出力に変換する計算手順 【他システム、ソフトウェアコンポーネント(DBを含む)、および他ソフトウェアユニットの間の協働作業】	(3) (5) ソフトウェアユニットが入力を出力に変換する計算手順 【他ソフトウェアユニット、および自メソッドの間の協働作業】
	データ	(2) 業務で保管している情報	(2)(3) ソフトウェアシステム内の保管情報	(3) ソフトウェアコンポーネント内の保有データ	(3) ソフトウェアユニット内の保有データ
【追加】設計と進化を支配する原則	要件割り当て構想	(2) 人・ハードウェア・ソフトウェアの役割分担構想	(2)(3) コンポーネントへの要件割り当て方針	(3) サブコンポーネントやユニットへの要件割り当て構想	
	共通問題設計結果		(2) (4)(5) ユーザインタフェース設計方針、品質要件実現方針	(4)(5) 技術的機構と、仮定されるソフトウェアサブコンポーネント／ソフトウェアユニット構造。デザインパターンなど	
	設計手法・基準		(4) 採用すべき設計手法、目指すべき「良い設計」の基準		

凡例)　番号:本文中の「保守の特徴」の番号

を表す文書を示さないで、変更部分だけやインタフェースのデータ構造の変更点だけを記述している場合がある。

(2.2)システムのトップレベルの構造と接続関係

特に、大規模システムの保守を長く担当しているプロジェクトの場合に、トップレベルの全体構造を表す文書を示さないで、変更部分のみの図を記述して済ます場合がある。

(2.3)機能分解構造と各構成要素への割り当て

小規模保守の場合で、実施する変更作業自体を要件と捉え、変更部分の既存プログラム構造と変更対象プログラムへの変更作業内容しか書かない場合がある。

(2.4)構成要素（パッケージ、クラス、プログラム、サブプログラム）間、および外部システムや再利用プログラムに対する「使用する」という情報

既存のプログラムライブラリからの再利用を明示していないことがある。また、プログラム・サブプログラム間の使用の関係は、手続き型言語を用いる場合は、プログラム構造図で表現しているが、オブジェクト指向言語の場合は書かれていないことがある。

(3)「設計と進化を支配する原則」の省略

(3.1)フレームワーク利用時などの構造のパターンと、各構成要素への機能割り当て構想

使用するフレームワークの知識として共通知識化されているものと思われる。場合によっては使用しているフレームワーク自体の情報も省略されていることがある。

(3.2)エラー処理などの共通機構の使用ルール

これもフレームワークと同様、同じ言語を使用する場合などでの共通知識になっているものと思われる。

(3.3)設計基準

このシステムではどの品質を重視すべきか、そのためにソフトウェアシステムではどういう対策を採るか、そして、どういう設計がよい設計なのかという設計基準（設計の品質基準）については書かれていない。

また、省略ではないが、下記の記述が不十分である例を見ることがある。

(4)詳細設計前の段階でのビジネスルール記述があいまい

ソフトウェアアーキテクチャ設計の段階でも、ルールという外から見た表現ではなく、内部の処理手順および協働作業手順で示していることが多い。IEEE Std 1016-1998では、「処理」と「データ」は「詳細記述」ビューで記述することになっており、プログラマが実装前に必要とする詳細度の情報であるので、ビジネスルールを処理手順で正確に記述できる。しかし、詳細設計前の段階では、手順で参照する個々の処理の内容が詳細ではないので、ビジネスルールがあいまいにしか記述できていないことが多い。

省略されている情報と不十分な記述の情報が、2.2節の表５で整理した設計対象粒度別の設計書記述情報のどの部分にあたるかを表11に網掛けで示す。これから、非常に広い範囲に影響を及ぼすことがわかる。

3.2. 記述省略のリスク

3.1節で記述省略などの現状を見てきたが、ここで、2.3節で述べた各プロセスでの情報の使用に照らし合わせて、省略によって知識がうまく伝わらない場合のリスクについて考察する。

設計プロセスやテストプロセスは保守プロセス中でも修正で使用されるので、保守プロセスの流れに沿って、保守プロセスの「問題および修正を分析」アクティビティ、「修正を実装」アクティビティ、設計プロセス、テストプロセスの順に検討する。

それぞれの記述省略から発生するリスクを、図２に示す。

3.2.1. 「問題および修正を分析」アクティビティでのリスク

ユーザの業務、開発・保守対象のシステムの使用環境と使用目的、あるいは外部システムとの接続の全体像が伝わらない場合には、既存システムの変更による業務や他システムへの影響を正しく把握できず、検討漏れが発生する恐れがある。

既存システムの全体構造が伝わらない場合は、既存システムへの要件変更による修正規模を正しく推定できない。また、稼働環境や再利用プログラムなどのリソースへの依存関係が伝わらない場合は、リソース側の変更による修正

表11　記述省略または不十分な記述が見られる情報

		業務システム	ソフトウェアシステム/アイテム	ソフトウェアコンポーネント	ソフトウェアユニット
情報の想定レベル		概要	概要（トップレベル）	詳細	詳細かつコーディング可能
外から見た記述	目的	【1.1 】 業務の目的、業務の機能要件、品質要件	【　2.3】 ユーザがソフトウェアシステムを使用する目的、機能要件、品質要件	【　2.3 】 ユーザおよび上位ソフトウェアアイテムが、このソフトウェアコンポーネントを使用する目的、このソフトウェアコンポーネントへの機能要件、品質要件	ユーザおよび上位コンポーネントが、このソフトウェアユニットを使用する目的、このソフトウェアユニットへの機能要件、品質要件
	機能	【1.1　　4】 業務への入力を業務外への出力に変換する内容（ビジネスルール）	【　　2.3　4 】 ソフトウェアシステムが入力を出力に変換する内容（ソフトウェアがになうビジネスルール）	【　　2.3　4 】 ソフトウェアコンポーネントが入力を出力に変換する内容（各ソフトウェアコンポーネントがになう、部分的ビジネスルール）	ソフトウェアユニットが入力を出力に変換する内容（各ソフトウェアユニットがになう、部分的ビジネスルール）
	インタフェース	【1.1 】 業務にとってのお客様と業務とのインタフェース	【　　2.1,2.2,2.4 】 ユーザのソフトウェアシステム使用内容、他システムへの情報提供内容	【　　2.4 】 ユーザインタフェース、他システムへの情報提供インタフェース、他コンポーネントへのインタフェース	稼働環境のAPI（ユーザや他システムへのインタフェースの実現手段）、他ユニットへのインタフェース
	リソース	【1.1 】 この業務システムを支援する他の業務、他システム	【1.2　2.1,2.4 】 サポートの人、他システム、稼働環境	【1.2　2.4 】 サポートの人、他システム、稼働環境、再利用ソフトウェア	【　　2.4 】 サポートの人、他システム、稼働環境、再利用ソフトウェア
	依存相手	【1.1 】 リソースと同じもの	【1.2　2.1,2.4 】 リソースと同じもの	【1.2　2.4 】 リソースと同じもの、および他ソフトウェアコンポーネント	【　　2.4 】 リソースと同じもの、および他ソフトウェアコンポーネント、他ソフトウェアユニット
内部の記述	従属物	【1.2 】 人、ハードウェア、ソフトウェアアイテム	【　　2.2,2.3 】 ソフトウェアコンポーネント	ソフトウェア(サブ)コンポーネント、ソフトウェアユニット	メソッド、処理ブロック
	処理 【もしくは協働作業】	【1.1 】 業務への入力を出力に変換する計算手順 【人、他業務、他システム、およびハードウェア、ソフトウェアアイテムの間の協働作業】	【1.1 】 ソフトウェアシステムが入力を出力に変換する計算手順 【人、他システム、およびソフトウェアコンポーネント（DBを含む）の間の協働作業】	ソフトウェアコンポーネントが入力を出力に変換する計算手順 【他システム、ソフトウェアコンポーネント（DBを含む）、および他ソフトウェアユニットの間の協働作業】	ソフトウェアユニットが入力を出力に変換する計算手順 【他ソフトウェアユニット、および自メソッドの間の協働作業】
	データ	【1.2 】 業務で保管している情報	ソフトウェアシステム内の保管情報	ソフトウェアコンポーネント内の保有データ	ソフトウェアユニット内の保有データ
【追加】設計と進化を支配する原則	要件割り当て構想	【1.1 】 人・ハードウェア・ソフトウェアの役割分担構想	【　　3.1 】 コンポーネントへの要件割り当て方針	【　　3.1 】 サブコンポーネントやユニットへの要件割り当て構想	
	共通問題設計結果		【　　3.3 】 ユーザインタフェース設計方針、品質要件実現方針	【　　3.2 】 技術的機構と、仮定されるソフトウェアサブコンポーネント／ソフトウェアユニット構造。デザインパターンなど	
	設計手法・基準		【　　3.3 】 採用すべき設計手法、目指すべき「良い設計」の基準		

凡例）【】内　：本文中の、記述情報省略例の番号

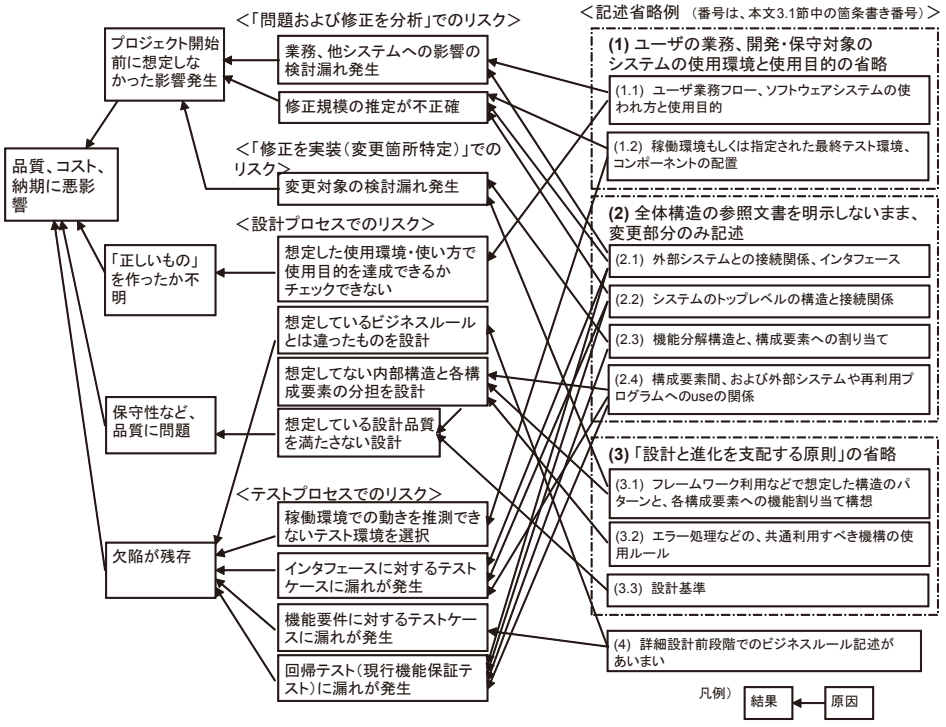


図２　記述省略から発生するリスク

規模を正しく推定できない。

3.2.2. 「修正を実装（変更箇所特定）」アクティビティでのリスク

機能分解構造と各構成要素への割り当ての情報が伝わらないと、要件変更に直接対応する変更箇所を特定しづらい。追加要件の場合は、構成要素にどのような種類の要件を割り当てるかの構想が伝わらないと、思わぬ構成要素に対する追加要件として設計される恐れがある。また、構成要素間の「使用する」という関係の全体像が伝わらないと、変更を波及させるべき箇所を確認しづらい。変更箇所の検討漏れが発生する恐れがある。

3.2.3. 設計プロセスでのリスク

ユーザの業務、開発・保守対象のシステムの使用環境と使用目的が伝わらないと、システム設計の評価の１つである「想定した環境・方法で使用して、ユーザが使用目的を満たすことができるか」のチェックができず、本当に望まれていたものとは違ったものを作る恐れがある。

内部処理の例示だけで、入力から出力に変換する正確なビジネスルールが伝わらないと、要件を誤解したものを作り、手戻りが発生する恐れがある。

使用すべき他の構成要素や外部リソースの指定が伝わらなければ、違ったものを利用するか自前で同様の機能を作る恐れがある。これは欠陥を呼び込むほか、コードの重複にもつながり、保守が複雑になる恐れがある。

フレームワーク利用などで想定した構造パターン、各構成要素への機能割り当て構想、および共通に採用すべき設計結果が伝わらないと、思わぬ内部構造と分担に設計され、設計の全体としての統一がなくなり、保守が複雑になる恐れがある。

また、重視すべき品質の指定や設計基準が伝わらないと、保守性など想定した品質が実現できない恐れがある。

3.2.4. テストプロセスでのリスク

開発･保守対象のシステムの使用環境が伝わらないと、テスト環境として必要な「稼働環境での動作をシミュレートできる」という条件を満たさない環境でテストを実施する恐れがある。

テストケースの設定に関しては、機能としての入力から出力に変換する正確なビジネスルールが伝わらないと機能要件に対するテストケースに漏れができる恐れがある。
外部システムとの接続関係とそのインタフェースや、システムのトップレベルの構造と接続関係の全体像が伝わらない場合は、インタフェース整合性確認でのテストケースに漏れが生じる恐れがある。

また、機能分解構造と構成要素への割り当て、システムのトップレベルの構造と接続関係、システムのトップレベルの構造と接続関係、および構成要素間の「使用する」という関係の全体像が伝わらない場合は、テストケースに漏れが生じる恐れがある。

4. 対策の方向

３章で、見かけた記述省略例とその情報が伝わらない場合のリスクを考察した。これらはリスクであり、担当者が固定し、プロジェクト内で共通知識となっている間は問題として顕在化しない。一方、プロジェクトを編成する上でも担当者自身としても属人化は避けたい。また、コスト削減要求から、オフショアなど新しい作業委託先を採用する方向にあり、問題が顕在化しない条件が崩れつつある。そのため､リスクが問題に変わる前に対策を始める必要がある。

対策の方向としては、共通知識として省略している情報の文書化と共有、および情報が伝わったことの確認と文書の追加の２方向が考えられる。以下、この２つの対処の方向性を説明する。図３に、３章で記述したリスクと、これから述べる対策の方向性との関係を示す。

4.1. 省略情報の文書化と共有

良いシステムを作るには、３章で記述した省略情報をメンバ間で共有しなければならない。開発・保守のために下記の文書を作成し共有することは必須である。

(1)ユーザの業務、システムの使用環境・使用目的
- ユーザ業務フロー、ユーザの使用目的、ユーザが使用する環境
- 稼働環境とコンポーネントの配置

(2)鳥瞰図的な概要を示す全体像
- 外部システムとの接続関係、インタフェース
- 機能分解構造、コンポーネントへの割り当て構想
- システムのトップレベルの構造と接続関係

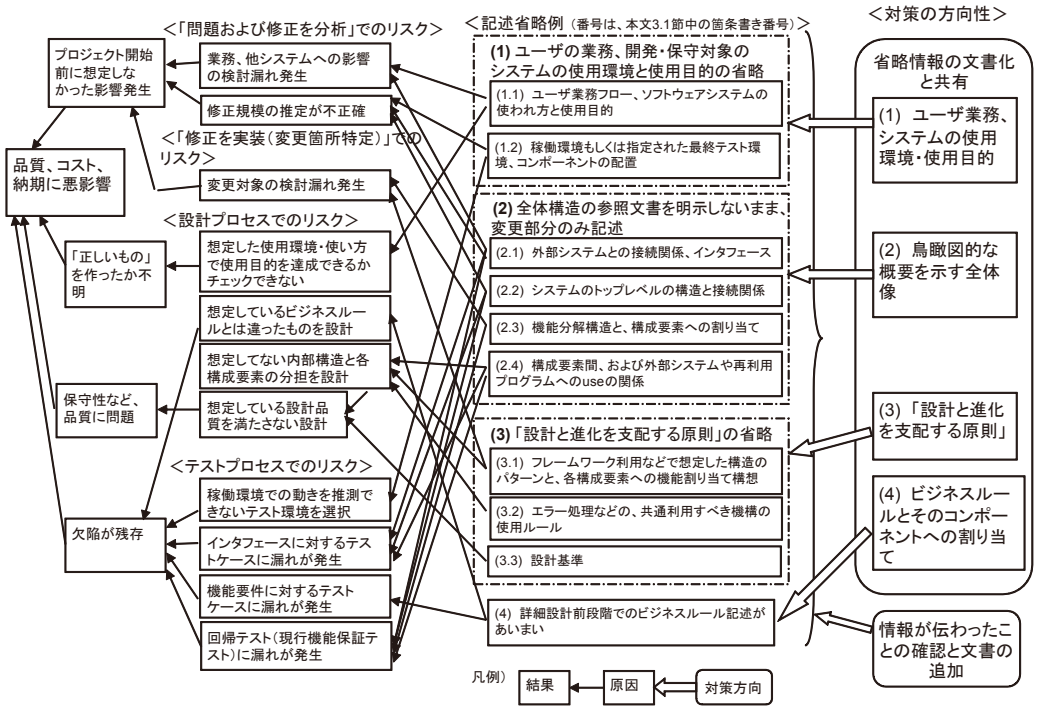


図３　記述情報の省略から発生するリスクと対策の方向性との関係

(3)「設計と進化を支配する原則」

- フレームワーク利用などで想定した構造のパターンと、各構成要素への機能割り当て構想
- エラー処理などの、共通利用すべき機構の使用ルール
- 重要視する品質要件と、それを実現するための設計基準

(4) ビジネスルールとコンポーネントへの割り当て

特に、プログラミング以降ではなく詳細設計以降を他の人に委託する場合は、アーキテクチャの思想――「システムの設計と進化を支配する原則」、およびコンポーネントに割り当てた機能――入力から出力への変換ルール（ビジネスルール）を正確に伝達する必要がある。これらがうまく伝わらないと手戻りが発生し、また複雑な構造や整合性のない構造になった保守性の悪いシステムになってしまう。

なお、機能として入力から出力への変換ルールを正確に記述するのは難しい。数学やフォーマルメソッドが候補にあがるが、記述内容を一般に理解される状況にはまだ達していないと考えている。そのため、ビジネスルールの表記方法のひとつに使われている[8] デシジョンテーブルの利用がよいと筆者は考えている。

さらに、保守のために残すべき設計書記載情報を考えてみる。実際のソースコードに表現されている詳細情報は、ソースコードから機械的にリバースエンジニアリングできる。例えば、クラスやモジュール間のインタフェースや接続情報、そして最新情報に保たれているか不安もあるが、コメントとして書いたメソッドやブロックおよびパラメータや内部データの意味などである。ツール化すれば、ソースコード作成後は最新情報を得ることは容易である。従って、本当に残すべき設計書は、ソースコードに書かれていない情報、人しかまとめられない全体を俯瞰する概要情報を記述するものだと考える。

この意味で、保守の場合にも、上記の(1)～(4)の概要文書を整えることを提案する。この情報はシステムの中身を理解するために必要な情報である。開発時に最新情報ではなくなっている場合は、保守引継ぎ時に「対象システム理解」タスクを追加し、更新すべきだと考える。

しかし、失われた文書を作り直すにも、作った文書の情報を最新に保つのにも、時間、労力、コストがかかる。そのため、以下の点に注意が必要である。

- 概要の情報は、「概略ではあるが正確なもの」[9]、「全体の開発を駆動するのに十分なくらいのもの」[10] であること。細部の変更では修正しなくても良いような抽象度で、かつ全体理解に必要な情報があることが必要である。これにより、最新状態を保つための更新を少なくできる可能性があ

る。

・システムの重要度、情報が完全に失われる可能性、保守作業発生の可能性などを考慮し、コストとのバランスをとって優先順位を付ける。全てのシステムについて失われた文書を作り直す必要はないかもしれない。また場合によっては、大規模保守にかかる直前に整備・更新するので良いかもしれない。

・必ずしもプロジェクト毎に作成する必要があるものばかりではない。稼働環境やアーキテクチャ関連の文書など、情報の種類によっては、お客様、開発組織、システムごとの単位で共通の文書として作ればよいものもある。

4.2. 情報が伝わったことの確認と文書の追加

設計書はコミュニケーションの手段であるので、読者との共通知識を想定し、それに合わせて理解に必要な情報を記述する。しかし、想定が間違う場合もあるので、相手の理解内容の確認と、場合によっては説明追加が必要である。

他の人に作業を委託する場合、手戻りを防ぐためには、できるだけ早期に相手の理解内容の確認を行うことが重要である。CMMI®では、供給者合意管理プロセスエリアの中で、委託先の作業成果物評価と委託先プロセス監視のプラクティスがあるが、これを早くから実践すべきである。作業成果物のチェックの視点として、委託したものに割り当てた要件に誤解はないか、漏れなく満たされているか、実現可能か、アーキテクチャのルールは遵守されているかが重要である。

最初のロットでチェックするプラクティスは既に行われているが、それよりも早く、最初に作成した納品前の作業成果物をレビューしてチェックすることが考えられる。そして、レビューで見つけた設計などの欠陥について原因分析して、必要な情報を追加する必要がある。このためには、納品前の作業用ファイルを共有できる環境があるとよい。

また、Q＆Aの内容を見て、理解度をチェックすることも考えられる。これは、相手の要件理解プロセスの監視に当たる。どのような質問が来ているかで、どこまで分析し理解しているかを見る。質問がなければ逆に、どう理解したかを質問してみることも必要である。また、人間性を疑われるリスクと欠陥が残るリスクはあるが、故意に間違いを埋め込んで指摘の具合を見るという「フォルトの人為的な種まき」をやってみることも考えられる。

そして、この確認により誤解を発見したときは説明を追加することになるが、それが共通知識だと思っていたものの補填の場合、その説明内容を資料にまとめさせて、相手の理解の確認を再度行うと同時に、文書化されていない共通知識の文書化を行うことも考えられる。

5. 終わりに

本稿では、設計書に記述すべき項目と、記述省略の現状とリスクを考察し、リスクを解消するための対策の方向を示した。

１章で述べたように、開発案件といっても、ビジネス環境も含めた環境変化への適応保守と捉えることができるものが多く、保守案件の比率は高い。一方で、保守文書の不備に起因する問題も数多く見受けられる。この意味で、保守に引き継がれ、維持していく文書のあり方を明確にすることは重要であり、その観点から次の知見を得た。

・保守には、保守引継ぎ時に保守対象システムのシステム文書を受領し、システムを理解することが必要である。そしてシステム文書中に、全体像を鳥瞰図的に把握できる文書がなければ、保守対象システムの理解が難しい。全体像を鳥瞰図的に把握できる文書は重要である。

・次工程への作業委託という観点から見ると、アーキテクチャの思想――「システムの設計と進化を支配する原則」や、正確な機能（入力から出力に変換するルール）を記述し、伝達することが重要である。

・コミュニケーション手段を確立し、相手が理解したことの確認と必要情報の追加が必要である。

今後、この考察で得た知見を、さらに発展させて、開発に用いる設計書標準化や作業標準化に盛り込みたいと考えている。

参考文献

1) 「企業IT動向調査2007概要」,（社）日本情報システム・ユーザ協会, 2007
http://www.juas.or.jp/project/survey/it07/

2) 「要求仕様定義ガイドライン」, 経済産業省情報処理振興課、（社）日本情報システム・ユーザ協会, 2007

3) ISO/IEC 12207:2008, “Systems and Software engineering

－Software life cycle processes", ISO(2008)

4) ISO/IEC 15288:2008, "Systems and software engineering －Systems life cycle processes", ISO(2008)

5) Karl E. Wiegers, "Software Requirements : Practical Techniques for Gathering and Managing Requirements Through out the Product Development Cycle", Microsoft Press, 2003
渡部洋子 監訳,「ソフトウェア要求　顧客が望むシステムとは」,日経BPソフトプレス,2003

6) IEEE Std 1016-1998, "IEEE Recommended Practice for Software Design Descriptions", IEEE(1998)

7) CMMI Product Team, "CMMI® for Development Version 1.2", CMU/SEI, 2006
JASPIC CMMI v1.2翻訳研究会 訳,「開発のためのCMMI® .1.2版」, 2007

8) Ronald G. Ross, "Principles of the Business Rule Approach", Addison-Wesley, 2003

9) Desmond Francis D'Souza, Alan Cameron Wills, "Objects, Components and Frameworks with UML － The CATALYSIS℠ Approach", Addison-Wesley, 1999

10) Scott W. Ambler, "Agile Modeling : Effective Practices for Extreme Programming and the Unified Process", John Wiley & Sons,Inc., 2002
