SEAMAIL

Newsletter from Software Engineers Association

Vol. 13, Number 12 September, 2003

目　次

ソフトウェア技術者協会

Software Engineers Asociation

ソフトウェア技術者協会 (SEA) は，ソフトウェアハウス，コンピュータメーカ，計算センタ，エンドユーザ，大学，研究所など，それぞれ異なった環境に置かれているソフトウェア技術者または研究者が，そうした社会組織の壁を越えて，各自の経験や技術を自由に交流しあうための「場」として，1985年 12月に設立されました．

その主な活動は，機関誌 SEAMAIL の発行，支部および研究分科会の運営，セミナー／ワークショップ／シンポジウムなどのイベントの開催，および内外の関係諸団体との交流です．発足当初約 200人にすぎなかった会員数もその後増加し，現在，北は北海道から南は沖縄まで，500 余名を越えるメンバーを擁するにいたりました．法人賛助会員も 24社を数えます．支部は，東京以外に，関西，横浜，名古屋，九州，広島，東北の各地区で設立されており，その他の地域でも設立準備をしています．分科会は，東京，関西，名古屋で，それぞれいくつかが活動しており，その他の支部でも，月例会やフォーラムが定期的に開催されています．

「現在のソフトウェア界における最大の課題は，技術移転の促進である」といわれています．これまでわが国には，そのための適切な社会的メカニズムが欠けていたように思われます．SEA は，そうした欠落を補うべく，これからますます活発な活動を展開して行きたいと考えています．いままで日本にはなかったこの新しいプロフェッショナル・ソサイエティの発展のために，ぜひとも，あなたのお力を貸してください．

ＳＥＡＭＡＩＬ Ｖｏｌ．１３，Ｎｏ．１２ ２００３年９月１５日発行 編集人 岸田孝一
発行人 ソフトウェア技術者協会（ＳＥＡ）
〒１６０-０００４ 東京都新宿区四谷３－１２ 丸正ビル５Ｆ
T: 03-3356-1077 F: 03-3356-1072 sea@sea.or.jp
印刷所 有限会社 錦正社 〒１３０-００１３ 東京都墨田区錦糸町４－３－１４
定価 ５００円 （禁無断転載）

編集部から

☆

前号の編集前記に：

次号もさらに何人かの幹事の方々からの寄稿が予定されているので，４月末までにはなんとか出せるのではと考えています．

一般会員の方々もふるって原稿をお寄せください．

と書いたのですが，依然として原稿が集まらず，どうしようかと考えているうちに，フォーラム，総会，ソフトウェア・シンポジウムとイベントに追われているうちに，とうとう夏になってしまいました．

☆☆

例年ですと，夏には分厚いシンポジウムの論文集を SEAMAIL の代わりにみなさんにお送りするのですが，今年の SS2004 の Proceedings は紙ではなく CD に焼付けです．そこで，この号は，過去５年間のソフトウェア・シンポジウムで最優秀論文賞を受賞した論文をまとめて掲載することで，なんとか船便状況を打開し，そこに SS2004 Proceedings CD を挟み込んでお送りすることにしました．

☆☆☆

次回のソフトウェア・シンポジウム2004 は，岡山理科大学の大西荘一先生に実行委員長としてお世話いただき，専修大学の飯田周作先生とシャープの鈴木郁子さん(お２人とも過去の SS の最優秀論文受賞者) にプログラム委員長をお願いして，来年６月中旬に岡山で開催することが決まっています．

☆☆☆☆

みなさん，ぜひこの号に載せられた論文を熟読・分析して，来年の最優秀論文を狙ってください．賞金は２０万円です．

☆☆☆☆☆

SEA年次総会報告

事務局

今年度の SEA 総会は，SEA 特別 Forum が行われた 2003年 6月 4日の夕刻， Forun の会場である東京国際フォーラム G510 会議室で開催された.

事務局から報告し承認された昨年度の収支および新年度の予算は次の通りである：

収支計算書 (2002年 4月 1日～2003年 3月 31日)

支出の部		収入の部	
人件費	17,910	新入会費	385,000
事務所費	2,806,650	更新会費	2,824,000
印刷費	2,289,935	賛助会費	2,800,000
通信費	2,745,500	雑収入	16,034
会議費	147,854		
支部支援費	134,820		
国際活動費	454,647		
消耗品費	84,611		
雑 費	29,190		
当期収支差額	－2,686,083		
合 計	6,025,034	合 計	6,025,034

予算案 (2003年 4月 1日～2004年 3月 31日)

支出の部		収入の部	
人件費	60,000	新入会費	440,000
事務所費	2,400,000	更新会費	3,200,000
印刷費	2,400,000	賛助会費	2,800,000
通信費	2,000,000	EVENT収入	1,000,000
会議費	180,000	雑収入	360,000
支部支援費	460,000		
国際活動費	180,000		
消耗品費	90,000		
雑 費	30,000		
当期収支差額	0		
合 計	7,800,000	合 計	7,800,000

なお，深瀬代表幹事の任期満了に伴う後任の代表幹事には，幹事会から後任として推薦した荒木啓二郎常任幹事(九州大学)が満場一致で承認された.

新年度の幹事会メンバーは，次ページの原案通り承認された.

2003年度 SEA幹事

代表幹事	荒木 啓二郎	九州大学
常任幹事	熊谷 章	東京エレクトロンソフトウェアテクノロジーズ
	高橋 光裕	電力中央研究所
	田中 一夫	日本フィッツ
	玉井 哲雄	東京大学
	中野 秀男	大阪市立大学
	深瀬 弘恭	インターネットイニシアティブ
幹事	石川 雅彦	SRA
	大場 充	広島市立大学
	落水 浩一郎	北陸先端科学技術大学院大学
	窪田 芳夫	東京電力
	小林 修	SRA
	小林 允	フリー
	桜井 麻里	ティップス
	酒匂 寛	Designers' Den
	塩谷 和範	SRA先端技術研究所
	篠崎 直二郎	NECソフト
	新谷 勝利	エイ・エフ・エス日本協会
	新森 昭宏	インテック W&G インフォマティクス
	杉田 義明	SRA
	中来田 秀樹	ネクスト ファウンデーション
	野中 哲	日本ジェムプラス
	野村 行憲	アイシーエス
	野呂 昌満	南山大学
	端山 毅	NTTデータ
	平尾 一浩	IIJ九州
	藤野 誠治	富士通総研
	松原 友夫	松原コンサルティング
	渡邉 雄一	アスキーソリューション
会計監事	橋本 勝	朝日監査法人
	吉村 成弘	公認会計士
事務局長	岸田 孝一	SRA

ソフトウェアトップダウン開発手法と適用例

－プロトタイプ可能な改良状態遷移表によるコード自動生成－

鈴木 郁子*、冨田 常雄*、乙井 克也*、上田 耕市**

シャープ株式会社
*IC 事業本部 設計技術開発センター
**生産技術開発推進本部　精密技術開発センター

要旨

ソフトウェアトップダウン開発とは、ソフトウェアの開発プロセスの中でも上流の設計工程を重視した開発手法であり、設計精度を高めて下流の開発工程を自動化しソフトウェアの品質の確保と生産性の向上を目指すものである。その１つのステップとしてＧＵＩを有するアプリケーションの外部仕様をプロトタイプ可能な状態遷移表で分析・設計し、この表からプログラムを自動生成する手法を考案し、実際の商品のソフトウェア開発に適用した。その結果、品質・生産性で効果を確認することができた。

1. はじめに

商品組込みのソフトウェアは、プロセッサの処理能力の向上やメモリの価格低下、デジタル化の影響を受けて大規模化、複雑化している。しかも、商品の開発期間は短くなる傾向にあり、ソフトウェアの開発スピードがボトルネックとなる事も少なくない。また、商品組込みのソフトウェアは、ユーザがソフトウェアの知識を持たない事が多く、パソコンのアプリケーションに比べると高い品質を要求される。そこで、LSI のトップダウン設計にヒントを得て、上流工程を重視し設計で品質を作り込み、下流工程を自動化するソフトウェアのトップダウン開発に取り組んでいる。

トップダウン開発のポイントは仕様の表現方法である。特に、液晶やタッチペンを備えた商品は、GUI（Graphical User Interface）を用いており、その仕様を表わすのは容易でない。これまでは、状態遷移図と自然言語による補足説明を組み合わせて表わす事が多かった。しかし、状態遷移図は、分かりやすいのだが細部まで記述し尽くすのは難しく、結局、文章による表現が必要になり、効率的ではなかった。また、一般に、図は、配置に自由度が高いため 2 つの図の同一性や差分を判定しにくく、作成や変更の時間も結構かかる。さらに、図からプログラムコードを自動生成するのも容易ではない。LSI の設計が、VHDL のような言語で開発されている事からも、図による表現はトップダウン開発に不適切と思われる。そこで、これに代わる方法を検討した。

まず、プロトタイプ手法[1]の導入を検討した。これは、システムの振る舞いが早期に確認できる、コード生成ができる、等の長所がある。しかし、組込みソフトウェアでは共通のプラットフォームがなく、コード生成は特定の CPU や組込み OS に依存したツールしかないのが現状である。また、この方法は要求を漏れなく備えているかの判定が難しい。画面レイアウトとシステム要件とに何ら関係がない事と、画面に現れないイベント（例えば、タイマーの発火や、通信の割込み、電池の消耗）が発生したときの仕様は、別の仕様書に記述せざるを得ないからである。

次に検討した形式的記述言語[2]は、これと対峙的であった。正確かつ簡潔に表現できるので、検証やコードの生成も可能であり、上流から下流までの情報の一貫性も保ちやすい。しかし、システムの振る舞い、特に、システムの実行時のイメージが分かりにくい。しかも、誰でもが理解できるものではないため、第 3 者による検査やユーザ部門による確認も難しいと思われる。

ところで、ユーザの操作が有限個のハードデバイスの組み合わせで実現されている商品では、状態遷移表による設計が行なわれており、効果を上げている。これは、表形式であるために自然言語より曖昧さがなく、形式的記述言語のような知識は要らないので、書く側、見る側のどちらにも理解しやすい。設計情報も構造を持つので、コードの自動生成も可能である。ただ、大規模な仕様を表現する

には、従来の方法では無理があるのと、システムの振る舞いを確認できないという、GUI 設計にとっては大きな欠点がある。そこで、表の構成を改良し、プロトタイプ機能とコード生成機能を開発し、実際の商品のアプリケーションの開発に適用したので報告する。

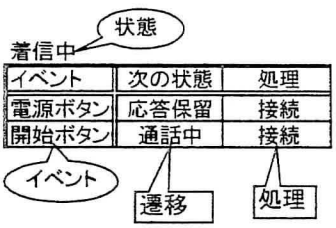
2. 方法

ソフトウェアトップダウン開発のキーとなる仕様の記述方法は、分かりやすく、記述自体は簡便であり、かつコードが自動生成できるに足る規則性をもったものが望ましい。我々は、上記の問題を踏まえて、状態遷移表による方法を選択し、大規模で複雑な GUI を有するシステムにも使えるように改良した。さらに、設計検査ツールやプロトタイプツールを開発し、設計の検証を支援するとともに、状態遷移表からソースコードを生成するツールを開発した。

2．1 状態遷移表の改良

一般的な状態遷移表では、図 2.1（a）のように、横軸にシステムが取り得る“状態”を縦軸には起こり得る“イベント”を記述し、その交差したセルに動作と次の状態を記述する。しかし、状態や事象の数が増えると表が巨大になり見にくく、理解や検証も難しくなる。また、ユーザの入力がハードデバイスによる場合にはイベントを洗い出すことが容易で有限個に収まるが、GUI ではイベントが現実的な数に収まらない可能性がある。つまり、各画面で使用される部品の種類・数・配置の組み合わせは無限に近いからである。そこで、本手法では、1)表を分割、2)内部状態を追加、3)GUI イベントを抽象化する事で、表の巨大化と複雑化を解決している。

(a)一般的な状態遷移表



(b)本手法の状態遷移表の基本的な構成

図2.1 状態遷移表の書き方

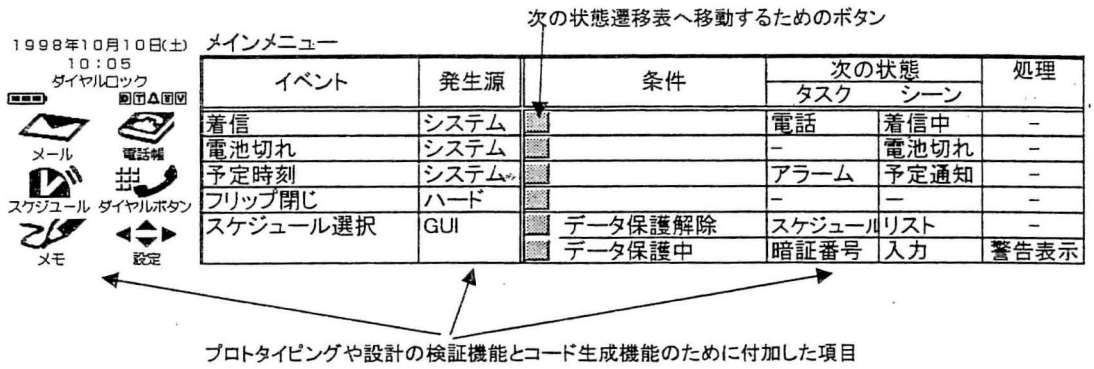
1) 表の分割：本手法では、図 2.1 の（b）のように１つの表は１つの状態でのイベントと動作、遷移先だけとした。この方式ならば、ある状態に固有のイベントは、その状態表にのみ記述するので、表が巨大化する事はない。一方、どのシーンでも共通なイベントは、新規の状態遷移表を作成する時に、デフォルトで入力しておくことで、記入漏れを防止できる。後述する適用例では、個々の操作画面（シーン）を１つの状態に割り当て、システム全体はシーンの集合として設計した。

2) 内部状態の追加：同じ画面で同じ操作をしても、その時のシステムのモードで動作や遷移が変わることがある。例えば、メニュー画面で、電話帳を選択した時、通常は電話帳のアプリケーションに移るが、システムが個人情報の保護モードならば暗証番号の入力が必要になる。システムのモード毎に１つの状態を割り当てることも可能だが、表の数が増えるのと、同じ画面が複数の表で定義されることになり整合性の保証が困難になる。そこで、図 2.2 のように内部状態を条件欄として追加し、これに応じて動作や遷移先を記述できるようにした。

3) GUI イベントの抽象化：GUI からのイベントを分類して抽象化した。例えば、リスト上でのクリックは“項目の選択”、ダブルクリックは“選択の決定”というイベントとして記述できるようにした。これを、デバイスレベルのイベントで表わすと“ペンダウン”“ペンアップ”のようになるが、抽象化する事により、ユーザ操作を明確にし、設計や設計検証が容易化する事を狙っている。

その他に、実装上の問題から、幾つかの項目を追加している。最近の組込みソフトウェアは、リアルタイム OS を有しマルチタスクで動くものが多くモジュール化して開発するのが一般的である。そこで、状態遷移先にタスク名（モジュール名）を追加している。これによりシーンの名前の競合範囲

を狭くした。また、後述するプロトタイプ機能や、ソースコード自動生成のためにイベントの発生源や画面イメージも追加している（図 2.2 参照）



イベント	発生源	条件	次の状態 タスク	次の状態 シーン	処理
着信	システム		電話	着信中	−
電池切れ	システム		−	電池切れ	−
予定時刻	システム		アラーム	予定通知	−
フリップ閉じ	ハード		−	−	−
スケジュール選択	GUI	データ保護解除	スケジュール	リスト	−
		データ保護中	暗証番号	入力	警告表示

図2.2 トップダウン設計手法で用いたシーンに基づく状態遷移表

2．2 設計支援機能の開発

一方、状態ごとに１つの表とするので、遷移先が分かりにくくなり、レビューが煩雑になる。そこで、項目の記入漏れや記入された遷移先に該当するタスクやシーンが無いセルを検査できる機能を作成した。状態遷移表は Excel で作成しているので、これらの機能はそのマクロを用いて作成した。

また、遷移先へ簡単に移行できるようにジャンプボタンを作成した。これを押すと、該当する行に記入されているシーンへ画面が切り替わるので、システムの動きが確認しやすい。これも Excel のマクロを用いて実装している。

2．3 プロトタイプ機能

上記の遷移先を確認するジャンプボタンを状態遷移表の画面イメージ上に複写することにより、簡易プロトタイプ機能を実装した。図 2.3 にその実行例を示す。

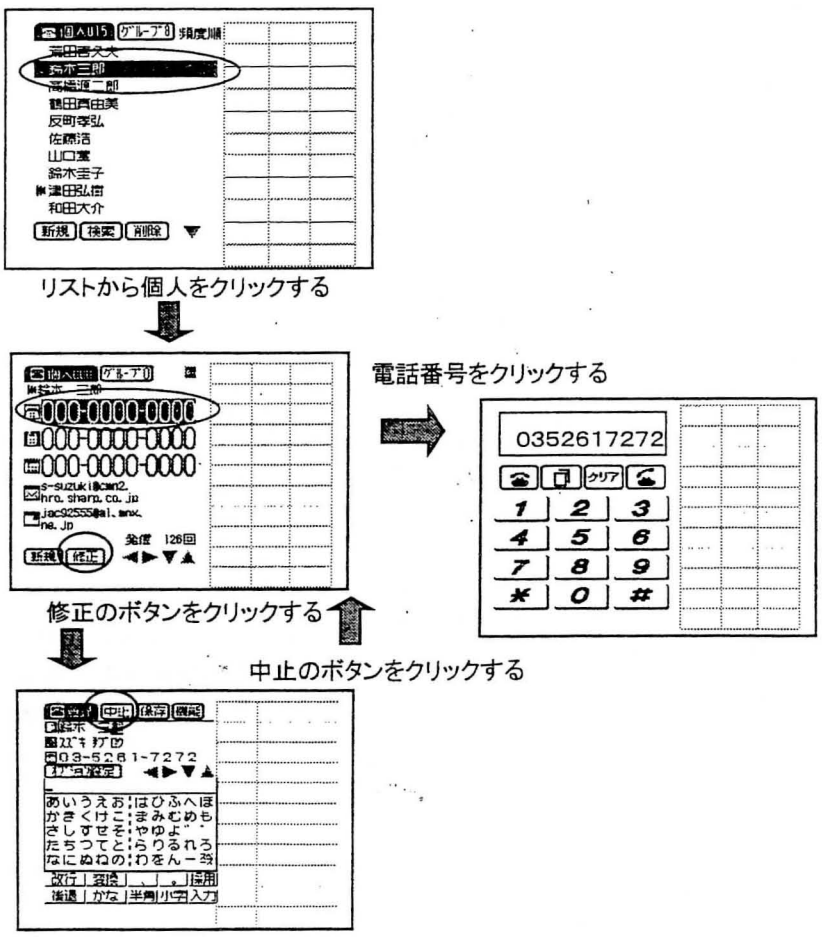


図2.3 本手法のプロトタイプ機能

2．4 コードの自動生成

トップダウン開発の目標の１つは、精度の高い設計情報を用いて後半工程の自動化を実現することである。状態遷移表には、イベントに対応する動作と遷移先が定義されているので、これを用いるとイベントドリブンな処理の部分のコードが自動生成できる。ただし、コードを生成するために、いくつかの必要な名前は状態遷移表に追加記入する必要がある。タスク名、

シーン名、GUI 部品名、処理関数名（いわゆるコールバック関数）、などである。また、状態遷移が実行の結果に依存する場合もあるので、このために実行に失敗した場合の遷移関数を指定できるようにした。なお、GUI の画像としての情報は、デザイン部門のレイアウト仕様書から取り込んでいる。

3. 本手法の適用例

本手法を、実際の商品アプリケーションの開発で採用した。

3．1 アプリケーションの概要

適用した商品は、携帯電話と携帯情報機器を合せたような小型の端末で、アプリケーションとして電話、メールの送受信、電話帳管理、スケジュール管理、手書きメモなどの機能を備えている。各機能は、それぞれが独立したタスクとして実装されており、リアルタイムＯＳμITRON 上で動作する。入力装置は、ボタン・ダイヤル・タブレット・マイクで、出力装置は液晶・スピーカーを備えている。

3．2 開発体制

アプリケーションの開発は、機能(メニュー)毎にチームを編成して並行して行われた。チーム構成は１～数名の担当者と、専任もしくは兼任の管理者からなる。各チームは、各機能の外部設計からデバッグまでを担当し、グラフィックライブラリやタイマーライブラリなどの共用部分の設計・開発・デバッグは別のチームが担当した。結合後のテストは、開発とは独立のチームが行なった。

3．3 本手法の採用

アプリケーション開発チームの約半数が本手法を採用した。本手法の採用・不採用の判断は、概要や実行例などの説明を行なった上で、各チームに任せた。不採用のチームの主な理由は、本手法に実績が無いことであり、モジュールの特性によるものではなかった。

なお、各シーンの画面生成部のプログラムもデザインデータから自動生成されるが、この方式は全チームが使用した。

3．4 本手法による開発手順

本手法を採用したチームは、以下の手順で設計・コーディング・デバッグを行なった。

ｉ）GUI 設計チーム（商品企画、デザイン部門）の仕様に基づき、各シーン毎に、イベントを列挙しそれぞれに対応する動作と次のシーンを定義する。この課程で問題があれば、GUI 設計チームにフィードバックする。

ii）プロトタイプ機能を用いて、GUI の動作を確認するとともに、イベントの漏れ、イベントに対応する動作や次シーンの定義漏れや定義ミスを、設計検査機能を用いてチェック、さらにレビューで設計精度を上げる。問題があれば、GUI 設計チームにフィードバックする。

iii）コード生成のための名前を与える。GUI 部品については、画面名、部品名を記入する。実行関数やエラー関数名も記入する。

iv）コード生成機能を用いて、イベント制御部のソースコードを自動生成する。この段階で問題があれば、iii）に戻り、内部設計を再検討する。イベントに対応した関数の処理をコーディングする。

v）コンパイルし、ドライバやミドルウェア、共通ライブラリとリンクし、EWS もしくは PC 上のエミュレータ上で実行し、デバッグを行なう。ただし、自動生成された部分に問題があれば、設計ミスであり、ｉ）から再実行する。つまり、自動生成されたコードを担当者が修正する事は禁止している。

vi）評価ボードもしくは実機上で動作を確認し、評価チームに引き渡す。

また、ii）以降の段階で、GUI 仕様の変更があった場合には、必ずｉ)、すなわち、設計に戻り、プログラムだけを変更する事は禁止した。

4. 結果

品質および生産性に関する指標としてモジュール毎に集計した結果を以下に記す。以下、A～D は本手法を採用したモジュール、E は、部分的に本手法を採用したモジュール、F～I は従来の方法で開発されたモジュールである。

4．1 規模

該商品は C 言語で開発した。図 4-1 は、モジュール別の C プログラムの行数を比較したものである。

本開発ではファンクションポイント[3]など質的な指標を採っていないが表 4-1 に、モジュール別画面数、GUI 部品数と難易度を示す。部品数は、構成している個々の部品数でなくグループ化されているものは1部品として数えている。例えばキーボードはキートップのボタンや機能ボタンを含めて部品化されているので、部品数は1と数えている。難易度は、内部条件の数とデータの更新頻度から3段階でランク付けし、設定した。

このようにモジュールにより画面数や部品数、難易度が大きく異なるのでモジュール間のサイズの格差も大きくなった。

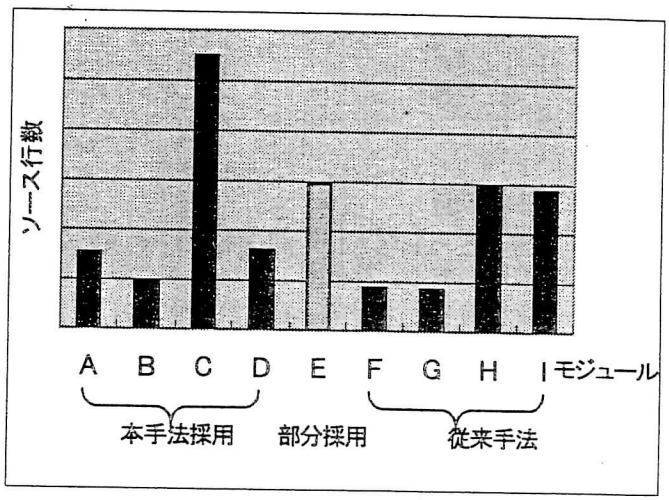


図4.1 モジュール別のプログラム規模

4．2 バグ数とバグ率

図 4-2 の棒グラフはモジュール毎に累積したバグ数を、折れ線は１０００行あたりのバグ数を表わす。ここでのバグは第３者により発見されたものであり、単体デバッグ中に開発者が発見したものは含まない。

表4.1 モジュール別の構成要素数と難易度

モジュール	シーン数	部品数	難易度
A	8	19	中
B	12	64	易
C	58	160	難
D	14	32	中
E	50	90	難
F	2	8	易
G	2	32	中
H	23	95	難
I	30	77	難

4．3 単純な生産性

図 4-3 は、各モジュールのソースコード数を、設計・コーディング・単体デバッグに要した人数と週数で除したものである。時間単位が週と粗いのは、当初、開発拠点が分散しており、しかも、統一的な作業報

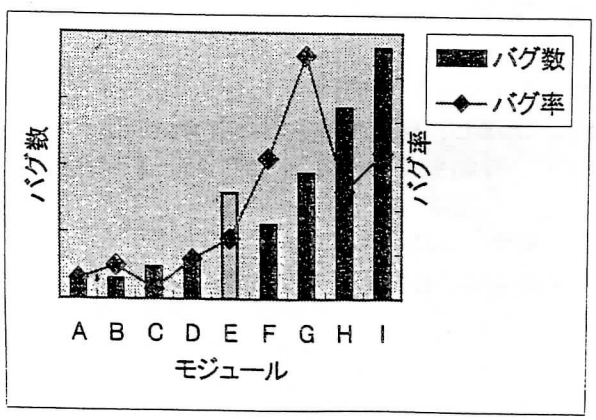


図4.2 モジュール別のバグ数とバグ率

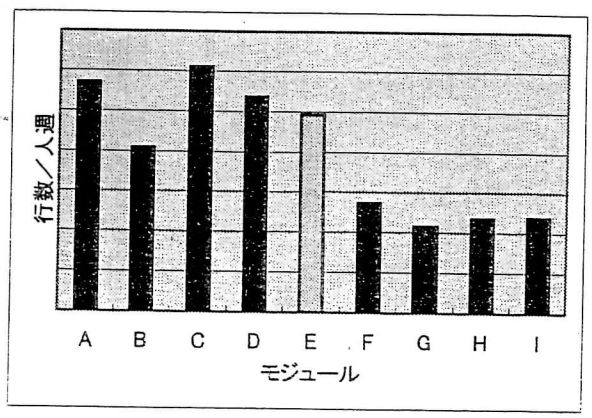


図4.3 モジュール別の生産性

告書が無かったためである。また、テスト部門へ引き渡してからのデバッグ時間は含まれていない。

4．4 その他

数値化していないが、トップダウン開発手法を採用したモジュールでは、以下が確認されている。

・デグレードが殆ど無かった。
・テスト工程で発見された不具合の原因の特定と対処の時間が短い。
・テストでバグが１件も発見されなかったサブモジュールがあった。
・担当者は本手法を半日程度の説明会と２時間程度の実習でほぼ習得した。
・ソースコードの書式が統一された。
・自動生成された部分のコードは、一度も手による修正はされなかった。

5. 考察

前述の結果から、トップダウン開発手法の効果を考察する。

5．1 品質に関する考察

バグ数及びバグ率ともに、本手法で開発したモジュールは、それ以外のモジュールとで有意な差が認められる。特にバグ率は、1000 行当り 1 件以下のモジュールもあり標準的なバグ率を大きく下回った。モジュールの規模が大きく、難易度が高いものでも、バグ率が低かった事は注目される。また、バグの修正が別のバグを生み出すというデグレードがほぼゼロであった事や、平均デバッグ時間が４時間以内と短かった事も考慮すると、本手法は品質の向上に有効であったと認められる。これは、以下によるものと思われる。

1）表形式で記述するので検討漏れやミスが少なく、検査機能もあるので、設計精度が高い。
2）プロトタイプで開発前に操作性を確認しているので設計変更が少ない。
3）ロジックが複雑になりバグが入り易い状態遷移の部分を、自動生成している
4）プロトタイプで期待値(システムの正しい振る舞い)を確認できるので、デバッグ時の評価ミスや修正ミスが少なくなる
5）設計書とプログラム間の整合性が保証される

5．2 生産性に関する考察

生産性は品質に比べると、本手法と従来手法の差が少ない。最大で、2.8 倍、最小で 1.5 倍の格差であった。これは、１つには、分母である時間の精度が粗いためである。また、時間を設計とコーディングとデバッグの３つの作業の和としているためと考えられる。どの作業にもっとも時間を要したかを比べると、トップダウン開発手法では設計作業であり、従来手法ではコーディングとデバッグの作業であった。また、モジュール B の担当者は、他のモジュールの管理者も兼ねていたが、その作業工数を考慮していない。さらに、テスト開始以降の工数はバグ数と相関するので、以上を考慮すると、トップダウン開発手法は、生産性においても効果があったと思われる。

5．3 その他

その他の効果として、品質や生産性のいずれにおいても従来手法ではモジュール間で格差が大きいが、本手法で開発したモジュール間ではばらつきが少なかった事が挙げられる。一般に、プログラミング力は、個人差が大きいとされているが、本手法では、プログラムの構造の大半が自動生成機能で決まってしまうために、モジュールの品質や生産性が平均化されたと推測される。

自動生成されたコードの効率や性能については絶対的な尺度での評価は行なっていない。最近のプロセッサの処理速度は速いので実行スピード面では十分な応答速度が得られたが、サイズは既存の同等のコードに比べると 2 割ほど多くなった。ただし、実用上の問題がなかったので、性能やサイズのチューニングのためにコード生成ツールを修正する必要はなかった。

その他に、本手法を使用した担当者からは、以下の意見があり、特に導入が容易である事が高い評価を受けた。

・簡潔でありながら網羅性に優れており設計精度が向上する。

・簡便で学習や訓練を必要としない手法であり容易に導入できる。
・設計情報の構造が単純であり独自の検証ルールも簡単に追加できる。
・社外の協力会社を使う場合にも、仕様の理解度が高まり、双方の誤解が少なくなる。

6. おわりに

我々は、ソフトウェアの品質と開発効率の向上を目的として、ソフトウェアトップダウン開発手法を提案しており、その１つとして、検証が容易でコード自動生成が可能な設計手法を考案した。また、この手法を、商品組込みのアプリケーションソフトウェアの開発に適用し、ソフトウェアの品質・生産性が向上する事を確認した。

しかし、本手法でも設計ミスがゼロではなく、改善の余地がある。特に、内部条件の漏れが多かった。これは、システムに共通なモードは予め分析し表に記載されているが、モジュールに固有のモードは担当者が要求分析と照らしあわせて記入するため漏れが多くなったと思われる。現状の設計検査機能は、状態遷移表のみを検査しており、要求仕様書との整合まではチェックできていないからであり、今後の課題である。

さらに、ソフト開発プロセス全体では、多くの課題を残している。特に、

１）従来のウォータフォール型のプロセスに比べると設計と開発の節目が曖昧になり、進捗状況が確認しにくい。

２）テストの工程との同期や情報共有が十分でない。

と言った問題があり、１）については表への記入数や記入率から完成度予測曲線を生成する機能を、２）については状態遷移表からテストデータを抽出し自動テストシステムでテストを行なう機能を、開発中である。

参考文献

[1]S. McConnell , "Rapid Development" Microsoft Press(1996) 日立インフォメーションアカデミー訳 ラピッドデベロップメント アスキー出版局(1998)

[2]B. Potter, J. Sinclair and D. Till , "An Introduction to Formal Specification and Z" Prentice-Hall(1991)田中他訳 ソフトウェアの仕様記述の先進技法－Z 言語 プレンティスホール(1993)

[3]C. Jones, "Applied Software Measurement Second Edition", McGraw-Hill(1996) 鶴保他訳 ソフトウェア開発の定量化手法第２版 共立出版(1998)

形式仕様を用いたシステムの非機能的性質記述の試み

－コンポーネントに基づくソフトウェア開発への応用－

飯田　周作
北陸先端科学技術大学院大学
情報科学研究科
TEL: (0761)51-1279
s_iida@jaist.ac.jp

二木　厚吉
北陸先端科学技術大学院大学
情報科学研究科
TEL: (0761)51-1255
kokichi@jaist.ac.jp

Abstract

コンポーネントに基づくソフトウェア構築では、コンポーネントを組み上げて作成したシステムのパフォーマンスの予測が難しい。パフォーマンスとは、システムの非機能的性質で、システムの応答速度や実行効率、メモリの使用効率等を指す。非機能的な性質は、機能的性質、つまりシステムの論理的で静的な性質が満たされた上で初めて議論可能である。本論文では、形式仕様を使ってコンポーネントに基づくソフトウェアの設計を行う方法を使い、その上で非機能的性質を形式的に記述することを試みる。形式仕様の記述には、代数仕様言語 CafeOBJ を使った。

1　はじめに

コンポーネントに基づくソフトウェア開発とは、独立性の高いソフトウェアの部品として用意されているコンポーネントを組上げて（用意されていないときには作成して）、システム全体を構成していく開発方法である。この方法には、再利用性の向上や、変更に対する柔軟性の向上等の利点が期待される。しかし、一方では、以下に挙げるような問題が考えられる。

1. 適切なコンポーネントを選択することが難しい。
2. 組上げたシステムのパフォーマンスの予測が難しい。

上記で挙げた両者の問題に共通する原因として、コンポーネントが通常バイナリ形式で提供され、内部の詳細な情報を得ることが難しいという点が挙げられる。このような問題に対して、例えば JavaBeans では、コンポーネントにイントロスペクションの機能を持たせてある程度の解決を図っている。しかし、このようなツールを使ったテストによる方法には限界があり、また２番目の問題に対しては解決法を与えない。本論文では形式仕様を用いて、これらの問題に対処する方法を考察する。１番目の問題に対しては、[7] 等に考察があり、ある程度有効性が確認されているものとして、本論文においては、特に２番目の問題を議論の対象とする。

システムの機能的性質とは、コンポーネントがどのように合成されているか、あるいは各々のコンポーネントがどのような振る舞いをするか等の論理的な側面を指す。システムの非機能的性質とは、システムの応答速度や実行効率、メモリの使用率等の、主にシステムのパフォーマンスに関する性質を指す。通常、コンポーネントの機能的性質を記述する際に一番重要となるのは、そのインターフェイスの正確な記述である。しかし、前述したように、このインターフェイスの情報だけで必要としているコンポーネントを選択することはできない。重要なのは、そのコンポーネントがどのような機能を持つか（あるいは、どのように振る舞うか）ということである。よって、コンポーネントの仕様は、この２つの側面を同時に記述できていなければならない。さらに、コンポーネントは、バイナリで提供され、利用者にとってはその実装に関する詳細な情報は不要であるため、これらの内部情報を記述することなく、システムのインターフェイスや機能を記述する方法が必要となる。非機能的な性質は、このような機能的性質とあわせて記述できることが望ましい。なぜなら、非機能的性質は機能的性質がただしく規定されている時にのみ議論が可能であり、これらを独立した記述にした場合、両者の間の整合性を保つことは、理論的にも、また、ドキュメント保守の観点からも困難である。

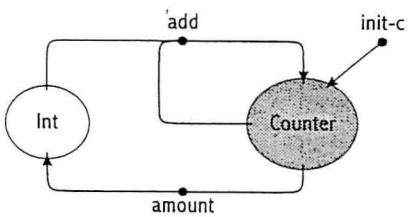
本論文では、コンポーネントの形式仕様を記述するために、代数仕様言語 CafeOBJ[3, 4] を用いた。CafeOBJ には、システムの機能（振る舞い）に着目した振舞仕様を

記述する能力がある。振舞仕様では、システムを外側から観測できる事柄にのみ基づいて記述し、内部の構造については記述しない。この振舞仕様は、上記のコンポーネントの仕様記述の要件を満たしており、コンポーネントの仕様記述に向いている。

2 コンポーネントの形式仕様

本節では、コンポーネントの仕様およびそれらを合成して、より大きなシステムの仕様を作成するための技術について解説する。コンポーネントの仕様は、CafeOBJ の振舞仕様として記述される。CafeOBJ は、実行可能な代数仕様言語で、与えられた仕様に関する性質を検証する際に、処理系を使ってその過程を補助することができる。

振舞仕様は、隠蔽代数 [5, 1] という論理に基づく仕様で、システムの動的な側面 (状態遷移) を外界からの観測できる事柄のみに基づいて記述できる。以下の図は、整数のカウンタを CafeOBJ で記述した例である (CafeOBJ の仕様は、通常テキストにより記述されるが、同等の情報を図により記述する方法も提供されている [2])。



I : Int C : Counter
amount(init-c) = 0 .
amount(add(I, C)) = I + amount(C) .

上記図では、塗りつぶされた楕円がシステムの状態空間を表し、楕円がデータの集合を表す。矢印は演算を表し、矢印の先が演算の値域を、元が定義域を表す。演算が複数の引数を持つときは、複数の定義域からの矢印になる。太い線による矢印は、システムの状態を遷移させたり、観測したりする演算を表す (このような演算を振舞演算と呼ぶ)。システムの状態を遷移させるような演算を操作演算 (action)、観測するための演算を観測演算 (observation) と呼ぶ。操作演算は、基本的に値域と定義域にシステムの状態をとるが、その他の引数がある場合には細い線を使う。上記の例の場合には、カウンタに値を足しこむための演算 add が操作演算に相当し、カウンタの現在の値を観測する演算 amount が観測演算に相当する。細い線による矢印は、この 2 種類の演算以外の演算を表すために使われる。定数は、引数を持たない演算として定義される。上記の例では、カウンタの状態の初期状態を定数 init-counter で定義している。

図で表現されるのは、システムのインターフェイスの部分を表している。代数仕様では、これを指標 (signature) と呼ぶ。システムの機能 (振る舞い、あるいは性質) を表すためには、等式を使う。上記図の 1 番目の等式は、"カウンタの初期状態を amount を使って観測すると 0 である"、という性質を表している。初期状態以外の状態は、2 番目の等式により再帰的に定義される。

上記の仕様を、CafeOBJ の処理系が処理できるような形で記述すると以下のようになる。本論文では、以下のようなテキストによる記法ではなく、図による記法で説明を行う。

```
mod* COUNTER {
  protecting(INT)

  *[ Counter ]*

  op init-counter : -> Counter
  bop add : Int Counter -> Counter
  bop amount : Counter -> Int

  var I : Int
  var C : Counter
  eq amount(init-counter) = 0 .
  eq amount(add(I, C)) = I + amount(C) .
}
```

2.1 コンポーネントの振る舞いに関する性質

振舞仕様が表現している対象は、その仕様を満たす全てのシステムである。よって、カウンタの例の場合、実装の違いによって以下のようなシステムが考えられる。

1. 実数の変数を使ってカウントアップした値を保持するような実装
2. リストやスタックを使って、add の履歴を保持するような実装
3. ...

カウンタにおいて、以下のような性質の検証を考えるとする。

$$(\forall c \in Counter)(\forall i1, i2 \in Int)$$
$$\mathrm{add}(i1, \mathrm{add}(i2, c)) = \mathrm{add}(i2, \mathrm{add}(i1, c))$$

この時、= の両辺はカウンタの状態である。この性質は、実装 1 では成り立つが実装 2 では成り立たない。なぜなら実装 2 では、内部のデータとして add の順番が保存されているので、左辺の場合と右辺の場合では保持しているデータが異なっているためである。しかし、カウンタをコンポーネントとして見た場合には、コンポーネント内部でどのようにデータを保持しているかは重要でない。公開されているインターフェイスを使っているかぎり、このような差異は意識できないからである。

振舞仕様では、上記のような性質を振る舞いに関する性質と呼び、振舞等価という通常の = とは異なる等価性を使って検証する[1]。2 つのシステムの状態 s と s' が振舞等価であるとは、以下のように定義される [5]。

定義 1　振舞等価性
s と s' をシステムの状態、c を観測テスト[2]としたとき、振舞等価関係 ~ を以下のように定義する。

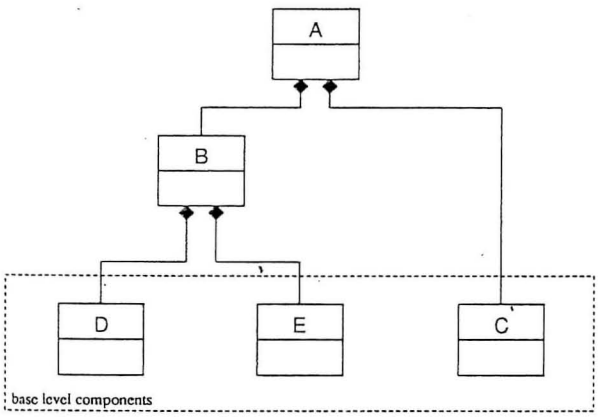
$$s \sim s' \ iff \ (\forall c)\ c(s) = c(s')$$

□

観測テストとは、操作演算を任意の順番に任意回適用した後で、観測演算を適用することをいう。つまり、2 つの状態が観測等価であるとは、それらの状態に同じ操作列を施した後で観測すると、常にその結果が等しいような状態ということである。

2.2 コンポーネントの合成

以下の図は、コンポーネントの合成を UML[10] を使って表したものである。四角はクラスを表し、塗りつぶされたダイアがついた線が合成を表す。前節で説明したコンポーネントの振舞仕様はクラスに相当する (UML と振舞仕様の関係については [11] を参照)。



合成の結果生成されたコンポーネントを、コンポジットと呼ぶ。例えば、コンポーネント B は、コンポーネント D とコンポーネント E を合成して作られたコンポジットである。コンポジットではないコンポーネントを、基本コンポーネント (base level component) と呼ぶ。コンポーネントの合成は、射影演算 (projection)[6, 8] を使って行う。射影演算は、コンポジットの状態をとりコンポーネントの状態を返す関数である。上記の図では、合成を表す線が射影演算に対応する。コンポジット c のコンポーネントを c_n(ただし、$n \in$ Nat) とすると、c_n に対する射影演算 π_n は、直感的には以下のように定義される (厳密な定義は [6, 8] を参照)。

- c の初期状態を、c_n の初期状態に対応させる。
- c の操作演算を、c_n の操作演算の列に対応させる。
- コンポジットにおける観測演算は、射影演算を使っていずれかのコンポーネントの観測演算に対応させるか、あるいは、幾つかのコンポーネントの観測演算に対応させた結果から導き出される値である。

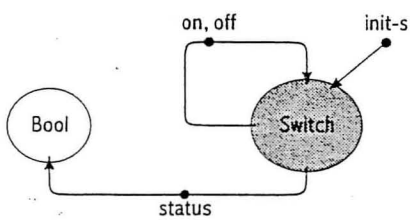
定義 2　操作演算の伝播
コンポジットにおける操作演算 m が、射影演算 π_n によってコンポーネント c_n の操作演算 m_n に対応付けられるとき、m は π_n によって m_n に伝播されるという。操作演算 m が C_n の恒等関数に対応付けられるときには、m は c_n には伝播されないとみなす。□

実際の仕様では恒等関数は省略され、単に m が無視されるように定義される。

定義 3　操作演算のグループ
コンポジットにおける操作演算は、それが射影演算によってどのコンポーネントに伝播されるかによってグループ化される。このグループを、操作演算のグループと呼ぶ。□

定義 4　操作演算のグループの重なり
もし、コンポジットにおけるある操作演算が、同時に異る操作演算のグループに属する場合には、操作演算のグループに重なりがあるという。□

ここで、例として、カウンタとスイッチのコンポーネントを組み合わせて、スイッチの状態に応じて整数の値をカウンタに加えたり、カウンタから引いたりできるようなシステムを作る。スイッチの仕様は、以下の通りである。

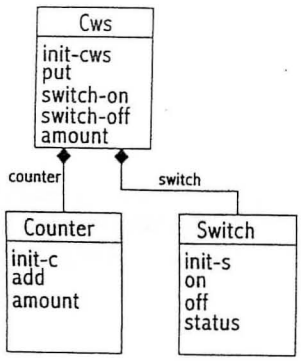


S : Switch
status(init-s) = false .
status(on(S)) = true .
status(off(S)) = false .

[1]通常の=の定義は、[3] 等を参照。操作的意味では、"両辺を簡約して同じ項になるとき" と定義される。
[2]観測テストは、" 可視文脈" あるいは" 観測文脈" と呼ばれることもある

操作演算 on と off は、それぞれ同じ値域と定義域を持つので、1 つの矢印で代表させた。

このスイッチをカウンタと合成させると以下のようになる。




```
I : Int   CWS : Cws
amount(CWS) = amount(counter(CWS)) .
counter(init-cws) = init-c .
counter (put(I, CWS)) = add(I, counter(CWS))
      if  status(switch(CWS)) = true .
counter(put(I, CWS)) = add(-I, counter(CWS))
      if  status(switch(CWS)) = false .
counter(switch-on(CWS)) = counter(CWS) .
counter(switch-off(CWS)) = counter(CWS) .
switch(init-cws) = init-s .
switch(put(I, CWS)) = switch(CWS) .
switch(switch-on(CWS)) = on(switch(CWS)) .
switch(switch-off(CWS)) = off(switch(CWS)) .
```

2.2.1 射影演算の性質

振舞演算として定義された射影演算[3]によるコンポーネント合成には、以下のような性質がある [8]。

定理 1 h と h' をコンポジット O の状態空間、$\sim$ を O 上の振舞等価性、$CObj$ をコンポーネントの名前の集合、π_n をコンポーネント O_n に対する射影演算、$\sim_n$ を O_n 上の振舞等価性としたとき、π_n は以下の性質を持つ。

$$h \sim h' \; iff \; \pi_n(h) \sim_n \pi_n(h') \;\; for \; all \; n \in CObj$$

□

これは、直感的には、「コンポーネントの振舞等価性の論理積がコンポジットの振舞等価性になる」ということである。

[3]射影演算を振舞演算ではなく、通常の演算として定義する方法もある。これら両者には、それぞれ異なる性質があるが本論文では触れない。詳細は [9] を参照。

2.3 コンポーネントの合成のパターン

射影演算を使った合成は、そのコンポジットを構成するコンポーネントの連結形態に応じて様々なパターンに分類可能である。本説では、以下のようなパターンを例として考えるが、当然これ以外のパターンも存在する。

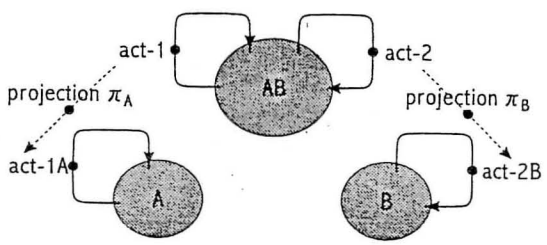
1. 並行連結
2. 同期並行連結
   (a) クライアント・サーバ
   (b) ブロートキャスト
3. 動的連結

2.3.1 並行連結

並行連結では、合成される各コンポーネントは互いに独立していて同期がなく、異るコンポーネントに対する操作は並行に実行可能である。

定義 5 並行連結
コンポジットにおける操作演算のグループに重なりがない場合、コンポーネントは並行連結されているという。
□

以下の図は並行連結のパターンを図示したものである。



射影演算 π_A は、コンポジット AB の操作演算 action1 をコンポーネント A の action1A に伝播させ、コンポーネント B には伝播させない (上記図では、恒等関数への対応は省略している)。

2.3.2 同期並行連結

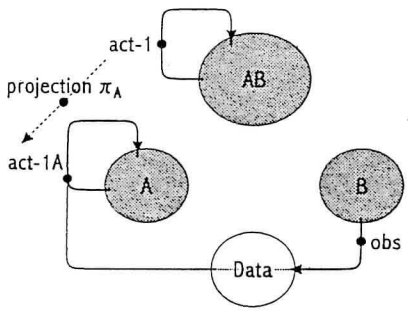
同期並行連結は、合成されるコンポーネント間に同期がある場合である。

定義 6 同期並行連結
以下の場合には、コンポーネント間に同期があり、コンポーネントは同期並行連結されているという。

- クライアント・サーバ型
  射影演算によって得られるコンポーネントの状態が他のコンポーネントの状態に依存している。
- ブロードキャスト型
  操作演算のグループに重なりがある。

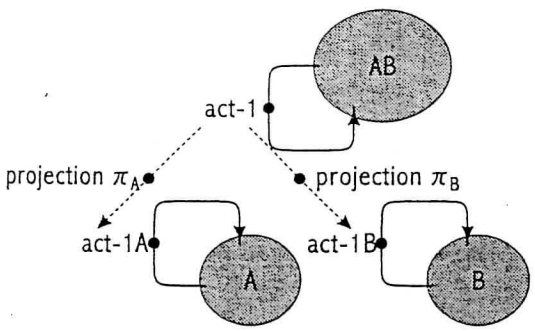
□

以下はクライアントサーバ型の図である。



コンポジットABにおける操作演算action1は、射影演算π_Aによって、コンポーネントにおける操作演算action1Aに伝播されるが、その際、action1Aのアリティにobservation1Bを使った観測結果を含む。

ブロードキャスト型は図示すると以下のようになる。



操作演算action1は、射影演算π_Aおよびπ_Bによって2つのコンポーネントにおける操作演算action1Aとaction1Bに伝播されている。

2.3.3 動的連結

動的連結は、動的コンポーネントを合成する際のパターンである。動的コンポーネントは、コンポーネントの識別子と共に初期化され、射影演算は、この識別子を使って各コンポーネントの状態を得る。各射影演算は、条件付き等式によって定義される。本論文では、この動的連結については扱わない。

3 非機能的性質の記述

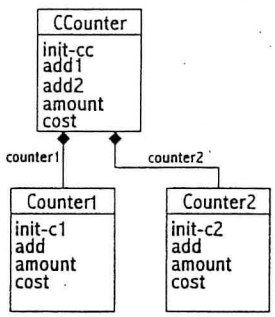
ここでは、非機能的性質を扱う試みとして、操作の実行時間を取り上げることにする。操作の実行時間を、ある操作(あるいは一連の操作列)が開始されてから終了するまでの時間とする。前提として、システムの仕様が前章までで導入したコンポーネント合成に基づいて記述されていること、基本コンポーネントにおける操作の実行時間が測定可能であることとする。基本コンポーネントにおける操作の実行時間は、観測演算として定義される。例えば、カウンタにおけるaddの操作に3ミリ秒かかるとすると、addの実行時間の総量を観測するための観測演算costは、カウンタの状態を引数としてとり自然数を返すような演算で、以下のように定義できる。

```
I : Int    C : Counter
cost(init-counter) = 0 .
cost(add(I, C)) = 3 + cost(C) .
```

コンポジットにおける操作の実行時間は、コンポーネント合成のパターンと密接な関係がある。以下では、並行連結および同期並行連結のクライアントサーバ型、ブロードキャスト型の各々の場合について説明する。

3.1 並行連結

並行連結では、異なるコンポーネントの操作演算に伝播される操作は、並行に実行できる。ここでは、例として2つのカウンタ(Counter1とCounter2)を並行連結で合成する場合を考える。ただし、Counter1とCounter2は、前節で説明したCounterと全く同じ仕様である。



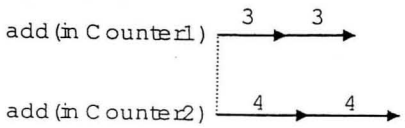

```
I : Int  C : CCounter
counter1(init-cc) = init-c1 .
counter1(add1(I, C)) = add(I, counter1(C)) .
counter1(add2(I, C)) = counter1(C) .
counter2(init-cc) = init-c2 .
counter2(add1(I, C)) = counter2(C) .
counter2(add2(I, C)) = add(I, counter2(C)) .
amount(C) = amount(counter1(C)) + amount(counter2(C)) .

cost(C) = cost(counter1(C))
      if cost(counter1(C)) > cost(counter2(C)) .
cost(C) = cost(counter2(C))
      if cost(counter1(C)) <= cost(counter2(C)) .
```

コンポジット CCounter は、add1 と add2 という 2 つの操作で Counter1 と Counter2 へ値を足し込む。観測演算 amount は、それぞれのコンポーネントにおける amount の結果を足したものである。

```
amount(C) = amount(counter1(C)) +
            amount(aounter2(C)) .
```

Counter1 の add の実行時間を 3 ミリ秒、Counter2 の add の実行時間を 4 ミリ秒としたとき、add1 と add2 を 2 回づつ任意の順番で実行したときの操作の実行時間は以下のように図示することができる。

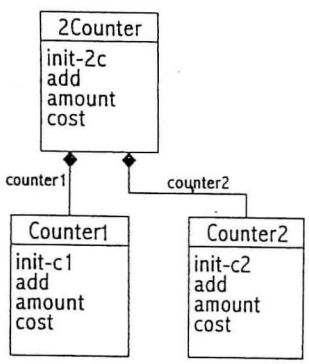


コンポジットにおける操作の実行時間 cost は、以下のように定義することができる。

```
cost(C) = cost(counter1(C))
    if cost(counter1(C)) > cost(counter2(C)) .
cost(C) = cost(counter2(C))
    if cost(counter1(C)) <= cost(counter2(C)) .
```

3.2 ブロードキャスト型

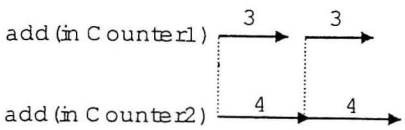
次に、同期が必要な場合について考える。カウンタを同期並行連結のブロードキャスト型で合成して 2Counter を作る場合を考える。




```
I : Int  C : 2Counter
counter1(init-2c) = init-c1 .
counter1(add(I, C)) = add(I, counter1(C)) .
counter2(init-2c) = init-c2 .
counter2(add(I, C)) = add(I, counter2(C)) .
amount(C) = amount(counter1(C)) + amount(counter2(C)) .

cost(C) = cost(counter2(C)) .
```

ブロードキャスト型の時には、コストが大きい操作によって同期がとられる。以下は、add を 2 回実行したときの様子である。



コンポジットにおける操作の実行時間 cost は、以下のように定義することができる。

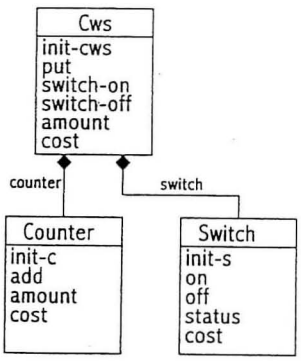
```
cost(C) = cost(counter2(C))
```

3.3 クライアントサーバ型

クライアントサーバ型の同期並行連結の例としては、スイッチ付きカウンタで考える。スイッチの on と off の操作にそれぞれ 1 ミリ秒かかるとする。

```
S : Switch
cost(init-switch) = 0 .
cost(on(S)) = 1 + cost(S) .
cost(off(S)) = 1 + cost(S) .
```

cost を加えたスイッチ付きカウンタ Cws の仕様は以下のようになる。

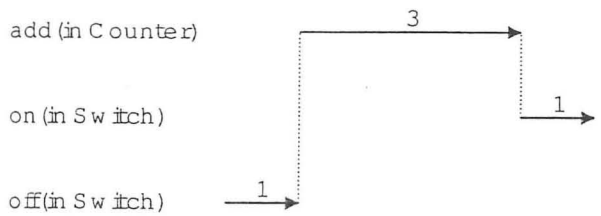



```
I : Int  CWS : Cws
amount(CWS) = amount(counter(CWS)) .
counter(init-cws) = init-c .
counter (put(I, CWS)) = add(I, counter(CWS))
      if  status(switch(CWS)) = true .
counter(put(I, CWS)) = add(-I, counter(CWS))
      if  status(switch(CWS)) = false .
counter(switch-on(CWS)) = counter(CWS) .
counter(switch-off(CWS)) = counter(CWS) .
switch(init-cws) = init-s .
switch(put(I, CWS)) = switch(CWS) .
switch(switch-on(CWS)) = on(switch(CWS)) .
switch(switch-off(CWS)) = off(switch(CWS)) .

cost(init-cws) = 0 .
cost(C) = cost(switch(C)) + cost(counter(C)) .
```

switch-on と switch-off は、共に同じコンポーネント (Switch) の操作 (on と off) に伝播されるので、これらは並行には実行できない。また、これらの操作と

put は、同期を必要とするのでこれも並行には実行できない。よって、Cws における操作の実行時間は、単純に各コンポーネントにおける操作の実行時間を足したものになる。スイッチを off にして、add し、on にした時の操作の実行時間は、以下のようになる。



コンポジットにおける操作の実行時間 cost は、以下のように定義することができる。

```
cost(init-cws) = 0
cost(C) = cost(switch(C)) + cost(counter(C))
```

4 議論

非機能的性質は、機能的性質が満たされてはじめて議論可能な性質である。特に、コンポーネントに基づくソフトウェア開発においては、システムの機能分解と各機能へのコンポーネントの配置が重要である。このため、形式仕様を用いて機能的性質を記述する方法を前提に、非機能的性質を記述することには意味があると考える。非形式的仕様を使って、本論文のアプローチと同様の方法をとることもできる。ただし、例えば実行時間が「計算できる」ということは、その仕様の実行時間に関係する部分に関しては形式的に記述されているということである。本論文では、仕様の全体を形式仕様言語を用いて記述したが、全体を非形式的仕様として（例えば UML 等を使って）記述して、部分的に形式的な記述を使うという方法も考えられる。ただし、このような方法を使う場合には、その記述法が機能分解やコンポーネントの配置に関しても、ある程度の能力を持っている必要がある。また、コンポーネントの変更や合成形態の変更に対して柔軟性を持っている必要もある。非機能的性質は、常に機能的性質が満たされた後に考慮すべき問題であるから、これらの要件が満たされなければ、効率の良い開発は難しい。この意味で、今回のように全体を形式的に記述する意義はあると考えている。ただし、形式仕様には記述性や可読性の面で、問題が生じる可能性もあるので、非形式仕様にどのくらいの形式性を導入すれば、どのような非機能的性質が記述可能であり、なおかつ機能的性質の記述も十分にできるかを研究することが重要であると思われる。

5 おわりに

本論文では、コンポーネントに基づくソフトウェア開発における非機能的側面を、形式仕様を用いて設計段階で把握する方法について考察した。コンポーネントに基づくソフトウェア開発の設計段階では、設計の機能的側面と非機能的側面を同時に扱うことが重要である。設計の機能的性質とは、コンポーネントの機能と結合の形態であり、非機能的性質は、実行の効率やメモリの使用効率等である。本論文では、基本コンポーネントの振舞仕様における観測演算として、非機能的側面を表すような演算を導入することで、このような２つの側面を同時に扱うことを試みた。非機能的側面としては、操作の実行時間に焦点をあて、コンポジットにおける操作の実行時間が、各コンポーネントにおける操作の実行時間からどのように計算できるかを考察した。その結果、コンポジットにおける計算は、コンポーネント合成のパターンによって整理可能であることが分かった。

今後の課題としては、まず操作の実行時間に関して、コンポーネント合成のパターン毎にもっと形式的に定式化を行う必要がある。しかし、実際のシステムでは、コンポーネント合成のパターンを幾つか組み合わせてコンポジットを作り、操作演算の数も多くなるので、今回考察したほど簡単にはコンポジットにおける操作の実行時間が定義できない。よって、今回考察した方法を現実的なものにするためには、定式化した計算方法をうまく組み合わせて問題を解くための方法が必要となる。その他の非機能的側面に関して、同様のアプローチが有効であるかどうかを確認することも重要な課題として挙げられる。

参考文献

[1] Razvan Diaconescu and Kokichi Futatsugi. Behavioural coherence in object-oriented algebraic specification. Technical Report IS-RR-98-0017F, Japan Advanced Institute of Science and Technology, 1998. To appear in J. Universal Computer Science, Springer.

[2] Razvan Diaconescu, Kokichi Futatsugi, and Shusaku Iida. Component-based algebraic specification and verification in CafeOBJ. In *proc. of the First World Congress on Formal Methods (FM'99)*, number 1709 in LNCS, pages 1644–1663, 1999.

[3] Răzvan Diaconescu and Kokichi Futatsugi. CafeOBJ *Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[4] Kokichi Futatsugi. An overview of Cafe specification environment — an algebraic approach for creating, verifying and maintaining formal specifications

over the net —. In *First IEEE International Conference on Formal Engineering Methods*. IEEE, 1997.

[5] Joseph Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97-538, UCSD Technical Report, April 1997. http://www-cse.ucsd.edu/users/goguen/pubs/index.html.

[6] Shusaku Iida. *A Formal Method for Distributed Systems based on Algebraic Specification*. PhD thesis, Japan Advanced Institute of Science and Technology, 1999.

[7] Shusaku Iida, Kokichi Futatsugi, and Răzvan Diaconescu. Component-based algebraic specifications. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 105–121. Kluwer Academic Publishers, 1999.

[8] Shusaku Iida, Michihiro Matsumoto, Răzvan Diaconescu, Kokichi Futatugi, and Dorel Lucanu. Concurrent object composition in CafeOBJ. Technical Report IS-RR-98-0009S, Japan Advanced Institute of Science and Technology, 1998.

[9] Michihiro Matsumoto and Kokichi Futatsugi. Object composition and refinement by using non-observable projection operators: A case study of the automated teller machine system. In Kokichi Futatsugi, Joseph Goguen, and José Meseguer, editors, *OBJ/CafeOBJ/Maude at Formal Methods '99 - Formal Specificatoin, Proof and Applications (workshop in FM'99)*. THETA, 1999.

[10] Rational Sortware et al. UML notation guide, September 1997. version 1.1.

[11] 飯田 周作 and 二木 厚吉. 代数モデルによる uml の意味論 - 形式仕様に基づくコンポーネント設計の視覚化を目指して -. In ソフトウェア工学の基礎 *V* 日本ソフトウェア科学会 *FOSE'98*, number 20 in レクチャーノート/ソフトウェア学. 近代科学社, 1998.

機器制御ソフトウェア開発のための設計手法

中井 晶也
キヤノン(株)創造環境推進センター、ソフトウェア技術開発部
nakai@ccc.canon.co.jp

1. 要旨

ステッパ開発において実用されいている，独自の機器制御ソフトウェア設計手法(以下 Bamboo 法と呼ぶ)を紹介する．適用分野としては「手順的機器制御」に特化し，そこでの問題点として第1に制御のモジュール分割がなされにくいことを上げる．そしてこの分野固有の問題を

制御仕様の階層的分割
制御の抽象モジュール化
分析に先立つモジュール分割

を中心として解決をはかっている．また，開発現場から発生したこの手法は，

徹底的に実用的であり，実行可能であること

を方針として，「日程厳守の中での最大限のメンテナンス性向上」という目的を達成しようとするものである．

同じ手法を，コンサルティングにより複写機に実用した例についても報告する．

2. はじめに

組込み系ソフトウェア開発においては，製品に占めるハードウェア開発のコストの大きさを考慮するとソフトウェア部門での日程遅延は許されないものである．しかし，一方では質の高い設計をすることも求められており，「日程厳守の中での最大限のメンテナンス性向上」という考え方が必要である．このような要求に応えるためには，然るべき設計手法を利用して効率的な設計を行う必要があるが，社内の実情としては多くの職場で混沌とした設計が行われている状況である．

本手法が実用された 1989 年当時の状況としては，機器制御を扱った設計手法は Ward[1], Gomaa[2]などわずかしか手に入らず，またそれも，われわれの要求を満たすものとはいえなかった．そのため，独自の設計法を開発する必要があった．

組込み系ソフトウェアの中でも，ステッパのような「手順的機器制御」を中心とする開発現場(以下単に開発現場)での作業とソースコードを見てみると，問題点の多くは「制御部分の妥当なモジュール分割がなされていない」ということが出発点となっており，この問題に対して直接的に応えられる設計手法が必要である．以下では，まず組込み系ソフトウェア開発の分類と，本手法の適用対象開発現場での課題を明らかにし，問題解決に向けて独自開発した Bamboo 法を紹介する．

3. 組込み系ソフトウェア開発の分類

3.1. 開発対象の分類

「組込み系」という言葉は，多くのものを含みすぎ，分野固有の問題点を探るためには分類が必要である．自社の製品は以下に示す「手順的機器制御」が多いが，身近にある機器を見ただけでもすぐにいくつかの分類ができる．[1]

「手順的機器制御」
多数のアクチエータを，可能な限りの並行動作と予備動作で駆動することにより処理速度性能を出す制御を行う．ステッパ，複写機，プリンタなど．

「自律的機器制御」
計測値や設定に基づいて，最適な状態・動作を保つ制御を行う．エアコン[2]など．

「組込み系データ処理」
入力データを高速で加工し，出力する．携帯電話など．

まだ分類は可能であろうと思うが，ここでは例として以上をあげておく．これらは，お互いに排他的なのではなく，1製品が複数の側面を持つのが普通である．また，これら制御の側面だけでなく，データ処理やユーザ I/F なども製品内に組込まれるが，ここで論じるのは制御の部分についてのみである．

---

[1] 網羅的な分類を目指したものではなく，組込み系という言葉の中に性格の異なるものが混在していることを示そうとしている．分類名は本論文独自のものである．

[2] エアコン，携帯電話の例は自社製品でなく，推測である．

3.2. 開発形態の分類

開発者のあり方で見れば,「自社開発」「外部委託」という分類ができる.

また, 製品のあり方で見れば, すでにある製品の系列製品を作る「シリーズ製品開発」, まったく新規の開発を行わなくてはならない「新規開発」の違いがある.

3.3. 実装制約による分類

メモリサイズと実行速度の制約から,「厳しい制約」「ゆるい制約」といった違いがある.

4. 適用対象分野の特徴

Bamboo 法の適用対象は以下を想定している.

- ・手順的機器制御
- ・シリーズ製品の自社開発
- ・メモリ・実行速度とも制約が厳しくない

それぞれについての特徴を述べる.

4.1. 手順的機器制御の特徴

スタートボタンを押された時点から連鎖的に発生する各アクチエータの動作手順の仕様は, 並列性を高めるために複雑になる. さらに, 予定的でないオペレーションやエラー対処の仕様が上乗せされる.

このような仕様は, 設計者の頭の中ではタイミングチャートのイメージで理解されるのであるが, 大きなシステムではかなり複雑になる. また, タイミングチャートは, ある実行条件下での例図でしかないため, 例外処理などまで正確に把握するためには状態遷移図などが必要になる. しかしこれは大きなシステムでは記述できないほどの大きさになる.

4.2. シリーズ製品の自社開発の特徴

シリーズ製品であるため, 製品仕様の概略がおおむね固まっており, 数世代の間で不変の仕様(仕様の骨格)というものが, ある程度わかっている.

また, 同じような機能のソフトウェアを, 同じチームまたは設計者が何回も開発することになるため, 設計者の製品理解が非常に深く, 過去製品の詳細仕様が頭に入っている. 開発途上においては, 未定の仕様を設計者自身が決められるという自由度もある.

4.3. メモリ・実行速度とも制約が厳しくないことの特徴

タスクの増加や, タスク間メッセージ授受が増加しても製品仕様を満たせなくなることがなく, 階層化やモジュール分割が可能である.

5. Bamboo 法の課題

5.1. 開発現場での具体的問題点

上述の「シリーズ製品の自社開発」,「メモリ・実行速度の制約が厳しくない」という条件は, よい設計をし, それを引き継ぐ上でプラスに働くはずのものである. しかし, 非効率的な開発状況として, 社内に以下のような例が見られる.

問題点1. 毎製品新規作り直しになったり, あるいは前製品のコードを流用して, かえって苦労する.

問題点2. デザインレヴューが行えず, 担当者任せになっている.

問題点3. ソース以外の設計資産を引き継ぐことができない.

問題点1は問題点3からの帰結である. また, 設計現場での作業内容を観察すると, 問題点2と3の出発点は, 妥当な設計単位(何に対して設計するのかの単位)を持っていないことにあると考える. つまり,

・妥当な設計単位がない
⇒明確な設計が行われない
⇒設計資料が作られない
⇒デザインレヴューを行えない

という流れである.

5.2. 問題原因の観察

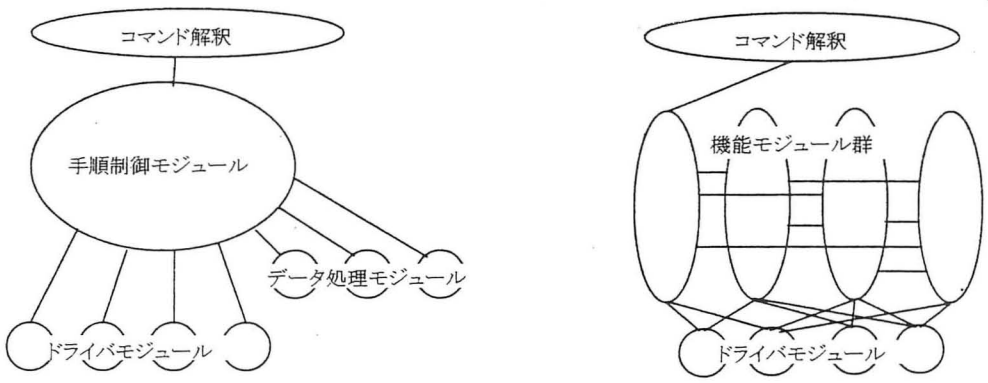
ではなぜ妥当な設計単位が作られないのだろうか. 実際に手法を適用したステッパと複写機の開発作業とソースコードを観察した結果から, 以下のことが言える.

設計現場での担当者割(それがそのまま設計単位になる)は, 機能仕様に基づいた分割で行う. 機能仕様はデータ処理の側面からはデータ処理モジュール[3]に対してきれいなマッピングが取れることが多い. しかし制御の側面からは, 一つの機能が影響を与えるアクチエータは多数にわたり, 実装面では方々に分散した形で現れることになる. したがって具体的実装対象というものが想定しにくくなる.

そこでとられたソフトウェア構造の実例が図1の2つのパターンである. 図1(a)は, モジュールとして切り出せるドライバとデータ処理は切り出した上で, 残った制御仕様は分割をせず, 1モジュールで組んでしまう方法である. 巨大な状態遷移テーブルの実装である. すべての制御の実装をここでするので, 制御の実装対象としては明確である. しかし前述したとおり, このモジュールの正確な仕様を状態遷移図などで記述し読み取ることは, サイズの面からいって現実的には不可能である.

図 1(b)の例は, 製品マニュアルに現れるような機能

[3] モジュールという言葉は, I/F に規定された, まとまりのあるいくつかの機能を, 他と独立性を持って提供する単位, という意味で使う.



(a)1モジュール実装　　(b)相互依存性のある分割実装

図1. よくない制御モジュールの実装例

名をもとに，機能毎にモジュール分割をし，それぞれに制御仕様を持たせるものである．これは制御の分割としては妥当性に欠けている．どの機能モジュールも，ほとんどすべてのドライバを操作することになり，ハードウェアの排他をしつつ最速動作をさせるためには，実行時期的に前後する機能モジュール間での同期制御が必要になる．そのためのメッセージが兄弟関係の機能モジュール間で多数飛び交うことになる．また，機能モジュール間でお互いを名指しでメッセージ交換をしているため，実際には独立性のあるモジュールとはなっておらず，制御的にはやはり1つの大きなかたまりである．さらに，機能間の関連を表すコードが各所に分散することになり，全体としてどう制御されるのかを見えづらくし，危険である．

このような例では，制御の観点から言えば，分割ができていない，あるいは，依存性の高い分割になってしまっているため，どちらも妥当な設計単位とはなり得ていないのである．

5.3. Bamboo 法の課題

以上の観察から，Bamboo 法は以下を解決できる手法でなくてはならない．

課題1．制御仕様を，独立性のある部分に分割できる方法を与えること．

さらに，これを実行するにあたり，一度全体の状態遷移図を展開し，それを分割する方法は取れない．なぜならそれは書くことができないほど大きいからである．そこで，以下も課題とする．

課題2．展開不能な大きさの状態遷移図を用いることなしに分割すること．

5.4. 既存手法

制御を扱った設計手法は，前述したとおり，構造化手法をベースにしたもの[1,2]があったが，これらはまずデータフローの解析をしている．

設計者の頭の中にあるものは，まずタイミングチャートのイメージなのである．それを出発点としない方法は，われわれ実設計者の感覚からずれている．また，経験的に，データ処理モジュールは制御部分に後から当てはめれば十分動くことをわれわれは知っており，制御より先にデータを検討する手法は，この分野では非効率的と考える．

6. Bamboo 法の制御分割

全体の開発手順の前に，Bamboo 法の中心技術である，制御の分割方法を説明する．説明の例題として図2のようなカラオケシステムの制御を考える．ハンド（上下，伸縮，回転の3軸）はレーザディスクを格納棚・予約台・プレイヤの間で移動させる．アーム(上下，回転の2軸)は，予約台とプレイヤとの間の移動を行う．プレイヤは再生機能を持つ．予約台とプレイヤ上では，ハンドとアームの干渉を避ける排他をしつつ，再生中に次ディスク予約をするなど，最速手順制御が必要である．

6.1. 制御仕様の階層的分割と制御の抽象化

まず制御仕様を独立部分に分割する方法を説明する．すすめかたの基本方針は，

・イベントの隠蔽ができるだけ多くできる分割をする．

・出入りするデータとイベントを抽象化し，抽象的モジュールとしてとらえなおす．

・下位の制御モジュール間の関係は，それらを結合するメタな制御仕様を上位において記述する．(同列の制御モジュール間での直接関連は持たず，階層的に組上げる)

である．具体的な分割過程の例を図3で示す．

図3(a)で，横方向の実線は各アクチエータの動作順序を表すタイミングチャート，太線は制御仕様を表す．制御仕様は，ひとつの状態遷移図をあらわすと考えればよい．制御仕様に入る矢印は発生イベントであり，出る矢印はイベントの結果発生する動作の指令である．このようなタイミングチャート図を，典型的な動作について書いてみる．

図3(b)の段階では，密接に動作が絡むアクチエータ同士をグループとみなし，グループ毎に制御仕様を書く．(ハンド制御仕様，アーム制御仕様，プレイヤ制御仕様の3つ．)このときグループ内にローカルなイベントと指令は他のグループから隠蔽する．

また，同列の抽象度を持つと思われるグループ間の関連を表すために，現実にはないメタな制御仕様を置く．すなわち，ハンド制御とアーム制御の関連を記述するために搬送制御仕様を置き，搬送制御仕様とプレイヤ制御仕様の関連を記述するために全体制御仕様を置く．

グループをまたがるイベントや動作は，制御仕様間のイベントとして記述する．

図3(c)の段階では，各制御仕様を抽象化する．まず，具体的な下位を図から消してしまう．すなわち，アクチエータの線とアクチエータに出入りしている矢印を消してしまう．残るのは，制御仕様間のイベント授受だけになる．これらのイベントは，実際には「ハンド縮駆動終了」などの具体的なイベントに対応するものではあるが，上位の概念で捉えなおして抽象化し，「回収完」のような意味を表すイベント名に付け直す．

以上の操作で，図4に示すような階層的な分割ができたことになる．

注意すべき点が3つある．1つは，下位の制御仕様の関連のみを記述したメタな制御仕様(搬送制御仕様と全体制御仕様)があることである．この制御仕様の中には，具体的なアクチエータへのアクセスは一つも記述されない．ここではイベントの中継と，下位の排他・同期のための制御しか行なわない．そのような，ある意味で無駄な物を置くことで独立性の高い分割ができている．実際この方法では，どの制御仕様も自分の直下の制御仕様しか知らない．兄弟関係の制御仕様同士の直接の関連がないだけでなく，親の制御仕様についても何も知らない．

2つ目は，制御仕様が，開始指令と終了イベント発生以外の，途中の指令やイベント発生も持っていることである．たとえば図3(c)のアーム制御の「設置」指令や，ハンド制御の「設置可」イベントである．それらを上位の制御仕様(この例では搬送制御仕様)で組合せることで制御の分割と最速手順制御が両立している．分割する際には，これら“途中”のイベントとして何が必要

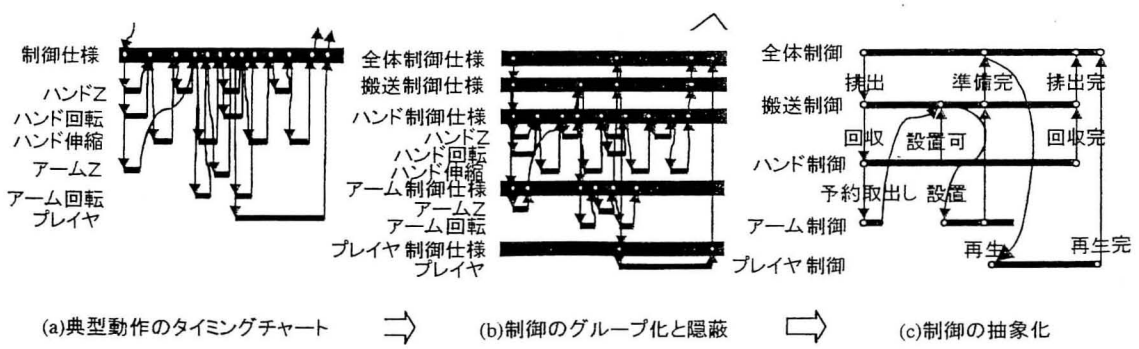


図 3. 制御の分割過程

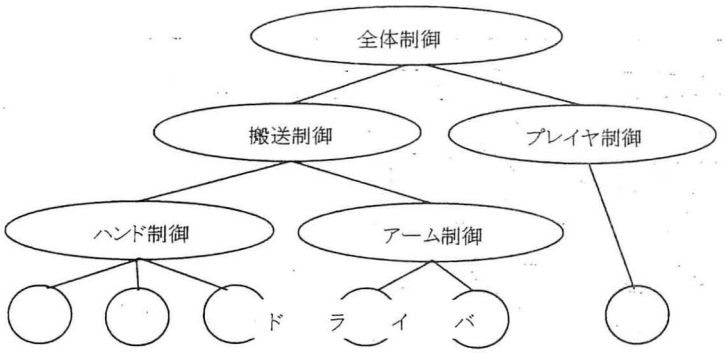


図 4. 制御の階層的分割結果

かということを発見することがポイントである.

3つ目は, イベントや指令を抽象化することで, 将来的なハードウェア変更に強くなることである. ハンドの構成が変化したとしても,「回収完」に相当する動作はあるはずで, 同じイベントとして返せば, それより上位の制御仕様は変える必要がなくなる.

6.2. モジュール化と抽象化

このように分割され, 抽象化された制御仕様は, そのまま抽象的な制御モジュールとして実装することができる. すなわち, 各制御仕様をタスクとして実装し, イベントが入る点は, タスクへイベント通知をするI/F関数として実装する. イベントが出る点は, コールバック関数からのイベント通知で実装できる.

ここまでで, 制御の分割とイベントの抽象化までができているが, さらに入出力のデータについても抽象化を図る. 上述の, タスクへイベントを通知する I/F 関数や, コールバック関数は, 指令値や実行結果を引数として持つことになるが, それらをできるだけ抽象化した形で定義する. たとえば, 指令値はパルス数などであるより,「ホームポジョション」「待機位置」などのような形にすることを目指す.

以上が, 制御を独立部分に分割し, 抽象的モジュールとする方法であり, 課題 1 に対する答えである.

7. Bamboo 法の開発プロセス

以上の制御分割と抽象モジュール化の方法は, Bamboo 法の基礎技術である. 実際の設計プロセスは以下の手順になる.

1. システム全体をモジュール分割し, モジュール関連図を作成する. モジュール単位で担当者割をする.
2. 各モジュール担当者は詳細動作を状態遷移図などで記述する. デザインレヴューで全体の動作可能性を検討する
3. 各モジュール担当者は, モジュールのI/Fを, ヘッダファイルの形で記述し公開する.

2, 3の進捗はモジュール毎に足並みがそろわないのが普通だが, 時期を一致させることにこだわらず, 進められるものを進めたほうがよい. 後戻りは必要があれば行う. 以下, 順を追って説明する.

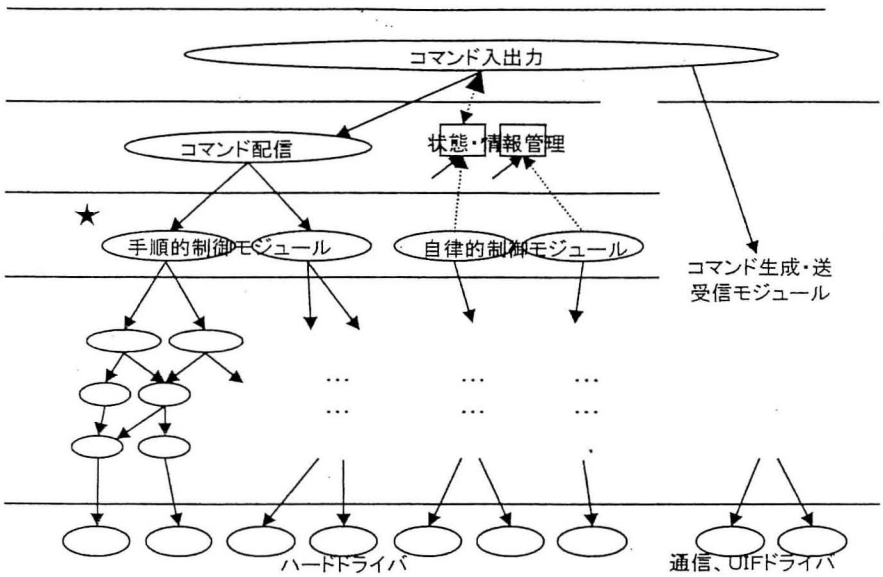
7.1. モジュール関連図の作成

モジュール関連図はシステム全体のモジュール階層構造を表す 1 枚の図である. 図 5 のアーキテクチャテンプレートを, 適用対象に合わせて具体化したものがそれである.

最下層のハードドライバはハードウェア構成で決まってしまうので, そのままそれを書けばよい. 上層のコマンド配信までと, 送受信, ユーザ I/F などは Bamboo 法によらず既存の手法で検討すればよい. Bamboo 法の中心課題は, 手順的制御をいかにドライバまで階層的に分割していくかである.

まず, 全体の制御のうち, 独立性の高い区分があれば, それにしたがって制御モジュールの最上位レイア(図 5 の★のレイア)を横方向に分割する. この分割は,「同時には実行されることがない制御」, または「排他・同期をまったくとらなくても問題ない制御」を分けることで行う. 前者のパターンの分割例は,「通常動作モード」と「サービスマンモード」のような分割, 後者のパターンの例は,「主動作制御」と「冷却ファン制御」のような分割である. しかし, これらの分割は,「巨大な全体制御」と「些細なその他の制御」程度にしかならないかもしれない. そうではあっても, 全体制御の部分を少しでも軽くする役には立っている.

次にそれらの制御を階層的に分割し, モジュール化していく. その分割方法は前項で述べたが, 実はモジュール関連図を作成するにあたり, 図 3 のような過程を踏んで分割することは, 実際にはあまりない. そのような手順をふみさえすれば, 大きな制御仕様も速度性能を落とすことなく分割できるということを設計者が理解してしまうと, 手順を踏まずに分割できるようになる.



このテンプレートは，おおよそのレイア・モジュール分けの指針を与えている．この図を具体化したものをモジュール関連図と呼んでいる．

ハードウェアからどれだけ抽象化されているかを上下方向の配置で表現する．矢印の向きはモジュール I/F 利用の方向を表す．

★のレイアの制御を，いかにドライバまで階層的分割するかが Bamboo 法の中心である．

図 5. アーキテクチャテンプレート(モジュール関連図)

課題 2 で挙げたように，展開不能なほど大きな図を展開してから分割することは実行不可能である．タイミングチャートは，状態遷移図よりはかなり情報を減らしたものにはなるが，それでも大きな図になってしまう．しかし Bamboo 法の分割方法が理解されると，部分的な動作のタイミングチャート程度を書けば分割はできてしまう[4]．そこには，「シリーズ製品の自社開発」という条件が良く働いている．分割に必要な，仕様の骨格というものが概ね確定しており，設計者がそれを熟知しているからである．

7.2. モジュール詳細図作成と検討

モジュール毎に担当者割をし(これは機能割とは違う)，各自がモジュール内の制御仕様を正確に記述する．状態遷移図を用いることが多いが，フローチャートで十分なモジュールもある．データ処理が中心のモジュールであれば DFD を使えば良い．すなわちモジュール詳細図は，状態遷移図，フローチャート，DFDなど，いずれでもよく，モジュールの特性に合わせて選ぶ．

詳細仕様に踏み込まずに分割されたモジュール関連図が，本当に正しかったかは，モジュール毎の詳細仕様を照合してはじめて検証される．このように，Bamboo 法では，分割をしてから分析を行うのである．

7.3. モジュール I/F の作成

モジュール I/F は，使用しているプログラミング言語のヘッダファイルとして書く．形式としてはソースコードではあるが，設計ドキュメントと捉える．モジュール毎に外部に公開するサービス(関数)，データ(引数など)，制御(コールバック，または送出されるメッセージ)の定義をする．定義したものについては，十分なコメントで利用方法が理解できるように記述する．

他の手法では，同様の内容を自然言語を使って表現する場合が多いが，技術者同士が読むものであることを考えれば，プログラミング言語を使ったほうが曖昧性がなく，かえって理解しやすい．また，そのままコードとして利用でき，設計から実装への明確な接続点となる．コードであるがゆえにメンテナンスもほぼ確実に行われる．

良く抽象化されたモジュールの I/F ヘッダは，次製品に無変更で流用できる可能性が高い．それに比べ，その実装コードはハードウェア変更に耐えられない場合が多い．オブジェクト指向言語を利用している場合，サブクラス作成による前製品ソースの無修正流用がありえるが，制御部分は差分コーディングでは対処しきれない場合が多い．(アクチエータが増減した場合など，差分追加では困難である．)I/F継承までを狙うか，C言語ならばヘッダの無修正流用を狙うのが良いと考える．

[4] 手法の評価の項で実例を述べる．

7.4. Bamboo 法の設計目標

以上が Bamboo 法の設計手順であるが，何をすれば設計できたことになるのか，という設計目標が具体的かつ単純でないと，設計行為のモチベーションはあがりにくい．Bamboo 法の設計目標は以下である．

1. ドキュメントとして以下を揃えること
   - モジュール関連図
   - モジュール詳細図(状態遷移図など)
   - I/F ヘッダ
2. それらを用いてデザインレヴューをすること

7.5. 状況に応じた設計プロセスの選択

上記の設計目標は，Bamboo 法での標準的な目標である．しかし，設計と実装に割り振れる工数に応じて，実施するプロセスを以下のように選択すればよい．

(I)標準プロセス

上述のとおりである．

(II)最低限度のプロセス

設計退避ラインとして提示する．ドキュメントとして「モジュール詳細図」も免除し，モジュール詳細は I/F ヘッダ公開後，設計者に委ねてしまう．後代への流用性よりもひとまず動くことを優先しており，途中で日程的に苦しくなった場合，このプロセスに切り替えれば，なし崩し的に設計ゼロになることを防ぐことができる．

(III)フルセットのプロセス

ドキュメントとして，機能仕様書とシーケンス図を加えている．しかし，機能仕様書の書き方についてはまだ研究不足であり，ここで紹介できるレベルではない．

8. 適用事例

8.1. ステッパでの例

Bamboo 法の発祥もとであるステッパ開発は，社内で最大規模の制御ソフトウェアを抱えている．コード量増大にともない，10 年前(1989 年)に開発スピードダウンと二次バグ頻発の状態になった．制御の分割と抽象モジュール化でこの状況に対処したのが Bamboo 法の原型である．

設計変更の対象は，ステッパ主シーケンスを担うCPU上のソフトウェアで，当時15万NCSSの規模で，そのうち60％程度が制御中心のソースファイルであった．抽象モジュール化の結果，6層，約100モジュール(末端のドライバモジュールを含む)の構成となった．装置の実行速度も，計測できる範囲内での低下は見られなかった．

データ処理を中心とした，流用性の高いモジュールはできるだけそのまま利用し，移行作業は 6 ヶ月という現実的な期間で終了した．以来，コード量，モジュール数は増加しつづけているが，基本的なアーキテクチャは現在までも引き継がれている．

8.2. 複写機機へのコンサルティング例

1999 年に，自社の主力商品である複写機開発に対し，コンサルタントという形で Bamboo 法を適用した．外部に適用するにあたり，手法としての体系化を進め，さらにコンサルティングの体験をまじえて，手法としてのまとめを行ったのが本論文である．手法適用は大きな障害なく進み，製品は実稼動している．

適用対象は5万 NCSS 程度の中規模開発の2製品であり，これを5層，50モジュール程度に分割した．2製品はハードウェア的にはかなり違いのあるものであったが，同時に設計を行い，モジュール分割とI／Fヘッダの共通度は非常に高かった．

デザインレヴューは状態遷移図中心で進められることが多く，モジュール関連図にさかのぼる修正は4～5件であった．

9. 手法の評価

9.1. 問題点を改善した

Bamboo 法の適用で，冒頭に挙げた問題点1～3を解決できたかという問いかけに対して，解決したわけではないがかなり改善した，と考えている．

ステッパでは，すでに 10 年間基本的考えが利用されており，製品シリーズ内でのアーキテクチャとモジュール I/F 引継ぎは順調に行われている．

複写機の設計では，ソース以外の設計資産としてドキュメントを残すことができた．その結果，それまでできなかったデザインレヴューがはじめて公開実施され，製品チームの枠を越えた検討がされた．具体的で有益な意見，質問が活発に出され，結果としてチームをまたがる共通認識が持たれるなど，効果は大きかった．

ただし，ドキュメントに関しては，実装段階に入ってからメンテナンスされなくなるということが，新たな問題点としてあがっている．

9.2. 開発現場に受け入れられた

手法はステッパと複写機という製品の違いを超えて，設計者に自然に受け入れられた．コンサルティング時の以下の事実がそれを物語っている．

1. 手法自体の説明にほとんど時間を割かずにコンサルティングができた．
2. 対象製品での制御分割の実例を示すと短時間で理解され，設計者自ら作業を進めはじめた．
3. 異論が出ずに全面採用された．(この部署では過去に Shlaer/Mellor 手法[3]を試みていたのだが，

結局全面採用とはなっていない.[5])

4. その後の製品でも, 同じ手法を用いて独力で設計している.

このように受け入れられやすい理由は, 設計目標が明確で, かつ, 設計者にとって実行可能で直感的に正しいと思えることだけを要求していることにあると考える.

9.3. 分析に先立つ分割は可能であり有効である

「シリーズ製品の自社開発」であるとはいえ, 分析に先立つ分割が本当に可能であるかというのは疑問であるかもしれない. 理論でその可能性を述べることはできないが, 結果からは可能であるといえる. コンサルティングの例では, 分割の手順を教えただけで, 1週間後にはほぼ最終形に近いモジュール関連図ができている. その間, コンサルタントとしては何の助言もしていない.

また, そのような手順が有効であることも確信している. 制御仕様は機能仕様に比べはるかに詳細で多数であり, 具体的事項に即して考えないと検証ができない. 多数の仕様を記述・議論できるような, 妥当なサイズの設計単位をまずは与えないことには分析結果の記述も検討もできないのである. 実際にデザインレヴューでは, 各モジュール毎に1枚の紙に収まる状態遷移図が用いられ, 活発な議論を生み出した.

9.4. 未解決の課題

少なくとも以下が未解決である.

- エレガントなハードエラー対処方法の研究
- 実装フェーズに入ってからの設計ドキュメントメンテナンス方法
- 機能仕様書の早期作成方法

エラー対処については, いくつかの解決パターンを提示し, 製品にあわせて選べるようにできればよいと考える. ドキュメントのメンテナンスは難しいが重要な問題である. 実装作業の効率を落とさない方法で解決しなくてはならない. 定式化したコードの書き方をしてもらい, コードからドキュメントへの逆変換をするなどの方法を模索中である. 機能仕様書は, ソフトウェア設計内に閉じた問題でなく, まだ解決の方向性がつかめていない.

10. まとめ

以上のように Bamboo 法は, 今のところ適用職場で受け入れられている. また, デザインレヴュー実施, 設計ドキュメント作成という具体的成果もあげている. これらは実施されて当然といえる成果ではあるが, 現実には何年にもわたり行われなかったことが可能になったということで, 実際的な効果としては十分大きなことである.

このような効果が手法の良否に起因するのか, あるいはいずれの手法であれ, 単にコンサルティングを行った結果であるのか, 客観的に区別することはできないが, 「手順的機器制御」という具体的な分野を正面に捉えた手法はほかに知らない.

今後の方向としては, 前述の未解決の課題を解決する方向と, 適用例の拡大があり, どちらにも挑戦する. できれば他分野のソフトウェア設計においても独自手法を考えて生きたい.

Bamboo 法の適用を想定している分野はけして広くない. しかしこれが正しい姿であって, どのような分野でも当てはまる設計手法というものはない, あるいはあっても十分な効果を生み出せないだろうと考える.

分野を絞り, 固有の課題を研究し, それに最も適した手法を既存手法にとらわれずに各人各様で考えるべきではないだろうか. 整理統合するのは, それらが出そろってからで良い. 分野で言えば, すでにあげた自律的機器制御, 組込み系データ処理, 組込み系GUIなどいくらでもある. 適用条件も日程短縮優先, メンテナンス性優先, 高信頼性優先など違いがあると思う. それらのマトリクスで考えれば多くの手法が生まれるのではないだろうか. 利用者としては, 適用分野と指摘された課題が自分たちに適合しているかどうかで手法を選択したいのである.

謝辞

Bamboo 法を信じて実製品開発に全面適用してくださった, 当社映像時事務機開発の金子徳治室長, 黄松強主幹研究員に, 心から感謝いたします.

参考文献

[1] Paul T.Ward “ワード氏のリアルタイムSA(構造化分析)手法”, bit, 1988年5月号, 共立出版

[2] H.Gomaa “A SOFTWARE DESIGN METHOD FOR REAL-TIME SYSTEM”, Communications of the ACM, Sep 1984

[3] アンディ・カーマイケル編 “オブジェクト指向方法論の世界” (株)プレンティスホール出版,

[5] それで本当にできる, という見通しが立たなかったのが採用に踏み切れなかった理由であった.

プロセス成熟度向上に向けた生産管理データの活用事例

古賀順二，島中一俊
NTT コムウェア（株） システム本部 SE 部 開発技法グループ
koga.junji@nttcom.co.jp shimanaka.kazutoshi@nttcom.co.jp

あらまし

近年，良いプロセスから良いプロダクトが生まれるという考え方に立ち，プロセスの最適化に対するニーズが高まっている．これに伴い，プロセスアセスメントによるプロセス改善への期待も高まってきた．しかし，アセスメントには相当の経営資源と時間が必要で，プロジェクトに対する理解の獲得等も含め，障壁が高いのも事実である．

そこで，我々は生産管理データが，プロセスの状況を映している事に注目し，生産管理データによる効率的で即時性の高いプロセスアセスメントを実現した．さらに，この方法は，データ内容の精度に言及せずデータの有無のみを扱う．このため，通常問題になるデータの欠損さえも情報として活かすことができ，データの有功活用という利点もある．そして，アセスメント結果は CMM[1]と整合を図ることによって，成熟度レベルの考えを問題定義に取り入れ，効率的な改善提案への手順を確立した．本論文においては，当該手順とそれを用いた実施例を紹介すると共に，手順作成上のノウハウと今後の改善点について報告する．

1. はじめに

プロセス改善の開始にあたっては，一般的に当該組織のプロセスについて現状把握し問題定義を行う．そして，通常この現状把握はプロジェクトや管理組織のキーマンに対するインタビューによって進められる．このため，アセスメントはその対象組織の規模が大きければ大きいほど多くの経営資源が必要となる．これは，例えば，プロジェクトに理解を得るための説明会開催，開発作業への影響を最小限とするためのインタビュースケジュール作成等の社内調整が挙げられる．また，実際インタビューを実施し，その後の結果集計や問題定義にも時間と工数を要しフィードバックの即時性を失うと共に経費も嵩む傾向がある．こうしたアセスメントの問題に対し，我々は生産管理データが，プロセスの状況を映している事に注目し，プロジェクトに影響を与えない生産管理データによるセルフアセスメントを目指した．そして，以下の課題を設定した．

① 被アセスメント組織の開発作業への影響を限りなく無くす．

② アセスメント結果は定量評価し即時にフィードバックする．

また，把握した問題に対する改善提案過程において以下の課題を設定した．

③ 改善提案は経営資源とのトレードオフを考慮し経営戦略的視点で行う．

2. 改善の目的と目標

我々は，図1の様に生産管理データがプロセス活動の結果であり考古学的な意味において開発プロセスそのものである点に注目した．そして，生産管理データをアセスメント用として分析することによりインタビューに代わる効率的なプロセス把握の手段とする方針とした．

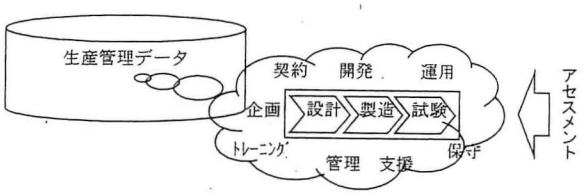


図1．生産管理データによる PM アセスメントの概念

そして，本アセスメントの目標を下記のとおり設定した．

① 組織の全体像を映し出す．
② 継続的なマネジメントのコミットメントを生み出す．
③ プロセス改善計画作成時にアセスメント結果を有益な情報源として利用する．

ところで，本アセスメント方式は本格的なインタビュー方式に比較すると多少表層的なものになる．そこで，効果に関しては同じく表層的なアセスメントを目的としているアンケート方式[2]を目標に据えた．

3. プロセスアセスメントの方法

以下に生産管理データを用いたプロセスアセスメント手順について記述する.

3.1. 生産管理データベース

NTT コムウェアでは, プロジェクトは全社的な標準の下で開発作業を進めており, 生産管理データによるプロジェクト管理を義務付けている. そして, プロジェクトは, プロジェクト管理支援ツールのサポートによって生産管理データを入力し管理業務へ活用する. また, 生産管理データは, プロジェクトが終了した後にもDB(データベース)化されて次期開発のための管理モデルや見積り根拠として活用される. このDB化される内容は, 開発計画, 成果物とマイルストーンのレビューや試験の計画と実績, プロジェクト完了報告, そしてフィールド品質等のデータが対象である. 図2に NTT コムウェアにおける生産管理データベースの概念図を示す.

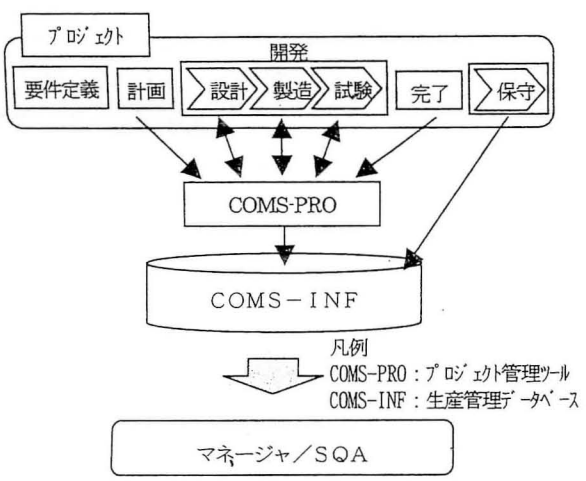


図2. 開発プロセスとCOMSシリーズの概要図

3.2. プロセスの調査方法(現状把握)

アセスメントにあたり, プロセスを静的観点／動的観点の二面からの調査を進める方針とした. 前者はプロセスの網羅度を扱う「プロセス定義」で, 後者はプロセスが実際どの様に実施されているかを扱う「プロセスの実施状況」の観点である.

(1) プロセス定義の調査方法

NTT コムウェアでは, 全社レベルの標準を元に各組織が作業標準を整備しており, この作業標準に定義したプロセスを管理モデルとしてプロジェクトの運営がなされている. そこで, プロセス定義の調査対象は対象組織の作業標準とした.

一方, 評価の基準となるプロセスモデルは, SLCP-JCF98[3]を評価基準として用いることにした. 我々がSLCP-JCF98を選んだ理由として以下の2点が挙げられる.

① プロセス定義が具体的である.
② プロセスの網羅度が高い.

特に, 我々は①のSLCP-JCF98がタスクによる具体的成果物を定義している点に注目した. これは, 成果物をタスクと見做し作業標準のタスク評価を成果物名によって頭出しするためである. 表1に SLCP-JCF98 のタスクと作業標準の対応つけのイメージを示す.

表1. タスクと作業標準の対応表イメージ

項番	SLCP－JCF98		プロセス定義	
	タスク	成果物	作業標準名	成果物
1.2.4.5	Pj管理計画の立案	Pj管理計画書	Pj管理要領	Pj計画書

(2) プロセス実施状況の調査方法

プロセスの動的な現状把握方法として SLCP との親和性が良いSPAの能力水準を参考にする方針とした. ただし, SPAの能力水準は5段階の評価となっており, これを開発管理データに適用して評価するのは困難で, 調査対象データ項目を増やさない条件では事実上無理であった. そこで, 我々は現実的な水準として表2に示す3つの水準を定義した.

表2. プロセス評価の水準

水準	意 味
定義	どの様にプロセスを定義しているのか
実施	定義したプロセスは実際に実施されたか
管理	実施したプロセスは監査／管理されたか

なお, 表2の水準中, 定義は(1)に説明したプロセス定義の調査結果を充てた.

そして, 実施と管理の調査についてもプロセス定義の調査と同様に成果物による評価を取り入れ以下の手順で作業を行った.

① SLCP-JCF98 に定義されるアクティビティー中の一部のタスクが, 別のタスクを管理する意味合いがある点に注目し, 管理を目的とするタスクとそれ以外を分類する.

② ①で管理が目的と分類したﾀｽｸに表2に示す「管理」の水準を，それ以外のﾀｽｸには「実施」の水準を割り当てる．

③ 一方，作業標準に定義されている成果物と生産管理DB定義を突合させることによってﾀｽｸ毎にﾃｰﾌﾞﾙ名とﾃﾞｰﾀ項目を割り当てる．そして，最終的に表3示すﾌﾟﾛｾｽ実施状況調査表を作成する．

④ ﾌﾟﾛｾｽ実施状況調査表を用いて生産管理DBのﾃﾞｰﾀを検索し，結果の有無によって該当ﾀｽｸの実施と管理状況を把握する．

表3. ﾌﾟﾛｾｽ実施状況調査表のｲﾒｰｼﾞ

SLCP-JCF98		実施		管理	
ﾀｽｸ	成果物	ﾃｰﾌﾞﾙ	ﾃﾞｰﾀ項目	ﾃｰﾌﾞﾙ	ﾃﾞｰﾀ項目
外部仕様の確立	外部仕様書	工程実績ﾄﾞｷｭﾒﾝﾄ	ﾄﾞｷｭﾒﾝﾄ枚数	－	－
外部仕様ﾚﾋﾞｭｰ	外部仕様書ﾚﾋﾞｭｰ結果	－	－	ｴﾗｰ分析	ｴﾗｰ現象

3.3. ﾌﾟﾛｾｽの評価

(1) ﾌﾟﾛｾｽ定義の評価基準

成果物の定義有無をｱｸﾃｨﾋﾞﾃｨｰの有無に置き換えて評価し，評価基準は有り無しの「0」，「1」判定とした．

(2) ﾌﾟﾛｾｽ実施と管理状況の評価

ﾌﾟﾛｼﾞｪｸﾄの評価は，生産管理DB中の該当ﾃﾞｰﾀを検索し有効ﾃﾞｰﾀの有無によって評価する．そして，組織としての評価は，検索の結果，有効ﾃﾞｰﾀが登録されているﾌﾟﾛｼﾞｪｸﾄ数を調査対象組織の全ﾌﾟﾛｼﾞｪｸﾄ数で割ったﾊﾟｰｾﾝﾃｰｼﾞで評価を行う．

(3) ﾌﾟﾛｾｽ調査結果のﾁｬｰﾄ化

調査結果は，効果的な状況把握の目的で図3に示すﾌﾟﾛｼﾞｪｸﾄ管理成熟度分析ﾁｬｰﾄ(以後PMMACと呼ぶ)にまとめる．

PMMACは，ﾌﾟﾛｼﾞｪｸﾄの実施状況をSLCP-JCF98のｱｸﾃｨﾋﾞﾃｨｰ毎に評価したﾄﾞｰﾅﾂｸﾞﾗﾌを3重に表示したものである．ｸﾞﾗﾌの各々は表2の水準に対応しており，内側から定義，実施，管理を示す．

また，ﾄﾞｰﾅﾂｸﾞﾗﾌ内の円ｸﾞﾗﾌは外側のﾄﾞｰﾅﾂｸﾞﾗﾌと対応しており，区切られた区画はSLCP-JCF98の定義によるﾌﾟﾛｾｽを示している．また，ﾄﾞｰﾅﾂｸﾞﾗﾌの区画内の数字はSLCP-JCF98のｱｸﾃｨﾋﾞﾃｨｰに付けられた項番に対応している．

そして，ﾄﾞｰﾅﾂｸﾞﾗﾌの各区画はｱｸﾃｨﾋﾞﾃｨｰ毎の調査結果の水準を，表4に示す色によって表示する．

なお，PMMACの評価単位はｱｸﾃｨﾋﾞﾃｨｰであるが，ｱｸﾃｨﾋﾞﾃｨｰは複数のﾀｽｸで構成される．そこで，代表ﾀｽｸを決定しｱｸﾃｨﾋﾞﾃｨｰの評価結果には(2)で示したﾀｽｸの評価結果をそのまま適用する．

表4. PMMACの読み方(表示色と水準)

表示色	水準	備考
赤色	0<実現度<25%	実現度が低い
黄色	25%≦実現度≦50%	実現度が不足
緑色	50%<実現度	実現度が高い

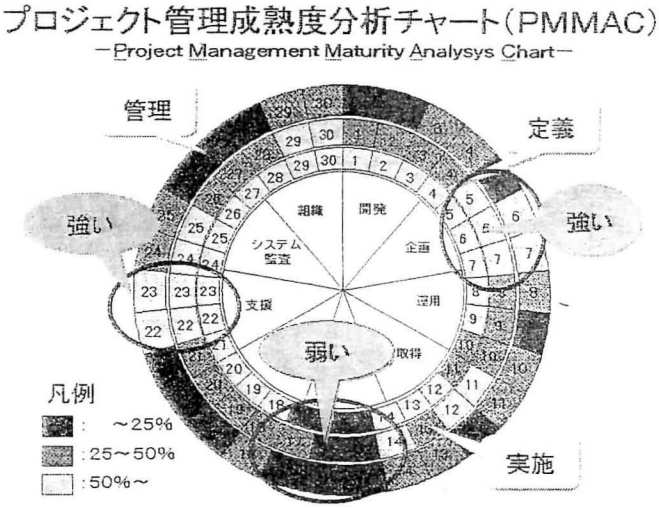


図3．PMMACのｲﾒｰｼﾞ

3.4. 問題定義と対策立案

次にPMMACによって映し出したﾌﾟﾛｼﾞｪｸﾄの現状から問題の定義を行う．

ところで，我々は調査対象組織が目標とするﾌﾟﾛｾｽ定義範囲と達成ﾚﾍﾞﾙを自由に選ぶ事を想定しﾌﾟﾛｾｽ網羅度の広さからSLCP-JCF98をｱｾｽﾒﾝﾄ用理想ﾌﾟﾛｾｽに採用した．そして，目標設定にはﾌﾟﾛｾｽと導入効果を組織が理解し戦略的に目標設定を行う必要がある．しかし，我々の取組みは，本調査がそうした根拠ﾃﾞｰﾀ獲得への着手であり，戦略的な目標設定が困難な状況にあった．

このため，我々はCMMが成熟度ﾚﾍﾞﾙに応じたｷｰﾌﾟﾛｾｽｴﾘｱ(KPA)を特定している点を利用し，戦略的な問題の絞込みを目指した．即ち，特定の成熟度ﾚﾍﾞﾙを達成目標に設定し，目標より高いﾚﾍﾞﾙでしか要求されないKPAの対策優先度を下げる事によって，問題の絞込みを図る方針とした．

CMM流の成熟度ﾚﾍﾞﾙ判定を行うためにSLCP-JCF98のﾌﾟﾛｾｽ/ｱｸﾃｨﾋﾞﾃｨｰの分類とCMMのKPAとをマッピングを行い，以下の手順で問題定義を行う．

① SLCP-JCF98 のｱｸﾃｨﾋﾞﾃｨｰ(ﾀｽｸ)とCMMのKPAで定義される活動との関連付けを行い SLCP-CMM(KPA)対応表を作成する．
② SLCP-CMM(KPA)対応表を，3．2で説明した調査結果に適用して，KPAの実施状況を把握し，対象組織のCMMにおける成熟度ﾚﾍﾞﾙを特定する．
③ 把握したﾌﾟﾛｾｽの成熟度ﾚﾍﾞﾙによって，該当成熟度ﾚﾍﾞﾙに要求されるKPAのうち弱いものの優先度を上げて重点的な対策を図る．

作業手順①で作成する対応表のｲﾒｰｼﾞを表5に示す．

表5. SLCP-CMM(KPA)対応表ｲﾒｰｼﾞ

CMM		SLCP－JCF98		作業標準	
KPA	活動	項番	ｱｸﾃｨﾋﾞﾃｨｰ	テーブル	データ項目
要件管理	活動1	1．2．1	開始	開発計画書	企画会議日
		1．4．2	ｼｽﾃﾑ要求分析		
		1．4．5	ｿﾌﾄｳｪｱ要求分析		

4. 調査結果の実例

4.1. PMMACによる分析結果

図4及び図5は社内のA，B2つの組織に対して実際に行った調査結果をPMMAC化したものである．なお，表示はSLCP-JCF98の主ﾗｲﾌｻｲｸﾙを対象とした．また，空欄は生産管理DBに該当ﾃｰﾌﾞﾙ定義が無かったものである．つまり，空欄は組織的なﾃﾞｰﾀ管理を行ってないことを示している．しかし，これは戦略的な管理の結果の場合が考えられ空欄＝悪ではない．ただし，当該組織の管理実態を映す鏡であり，次期管理ｼｽﾃﾑ開発のﾛｰﾄﾞﾏｯﾌﾟ作成への活用が期待される．

まず，A，B両組織とも[注]1.1 取得，1.2 供給ﾌﾟﾛｾｽの実現度が高く全社的に実施が徹底している事が判る．一方，1.5 保守，1.6 運用ﾌﾟﾛｾｽは，標準に定義はあるが実施，管理で両者とも空欄が多く全社的な状況把握が弱いといえる．

次に組織毎に個別に見ると，組織Aは「実施度」「管理度」の実現度は低いが組織的なﾌﾟﾛｾｽ定義の網羅度が素晴らしい．そして，1.4 開発ﾌﾟﾛｾｽについては，上流工程の実現度が高く下流工程が低い傾向が見える．一方，組織Bは，組織Aに比較して「実施度」「管理度」の実現度は高い．しかし，逆に組織的なﾀｽｸ定義は，弱い．そして，1.4 開発ﾌﾟﾛｾｽについては，上流よりも下流の実現度が高い傾向が見える．

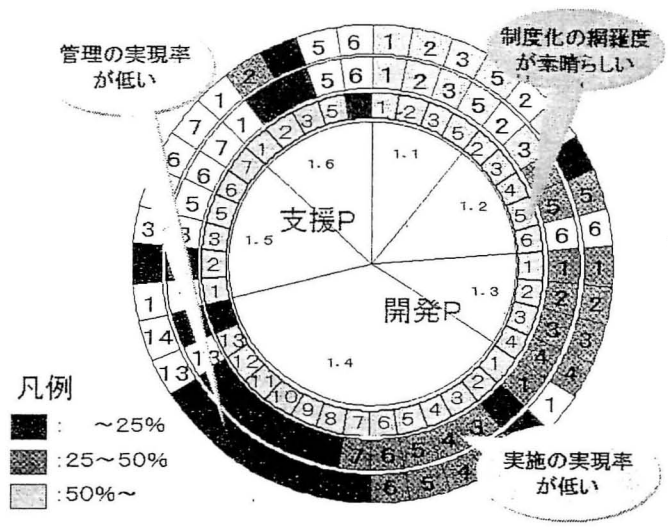


図4. 組織Aの調査結果のPMMAC

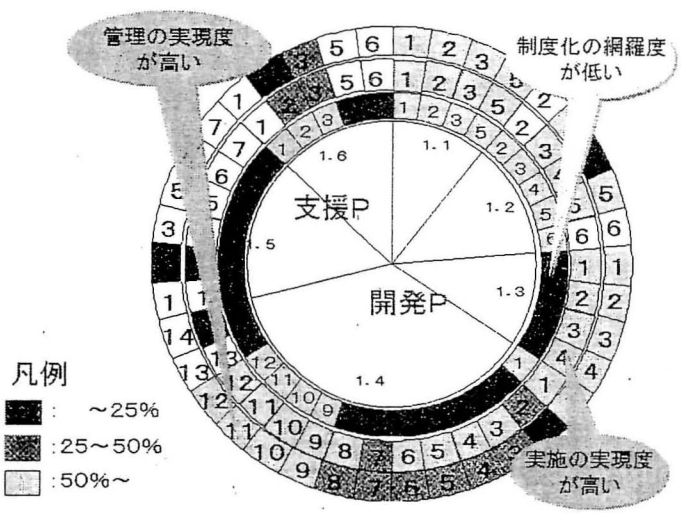


図5. 組織Bの調査結果のPMMAC

以上の様に各組織のﾌﾟﾛｾｽの状況は良く表されておりPMMACのﾌﾟﾛｾｽ把握の機能は十分であるといえる．

注：数字はSLCP-JCF98のﾌﾟﾛｾｽに付けられた項番を示し，PMMAC内部の円グラフの番号に一致する．

4.2. 成熟度レベル判定

3. 4に示した手順に従いCMMのKPAと生産管理データを関連付けた対応表を表6に示す. なお, 表6の作成にあたって, 生産管理データの限られた情報量に合わせるためにドラスティックな選択基準を必要とした.

表6. KPA(レベル2)と生産管理データとの対応表

KPA	テーブル	評価
RM	開発企画会議	有/無
SPP	開発計画書	有/無
STO	工程管理データ(試験項目数、DQ枚数)	式1
SSM	開発計画書∩外注規模∩工数	有/無
SQA	検査∩完了報告書∩SQA	有/無&式1
SCM	工程管理データ(エラー分析、バグ分析)	式1

注1: 工程実績報告数／工程稼動実績数・・・(式1)

前出の組織A, Bの調査データを表6の結果に適用して成熟度の判定を行った結果を表7に示す.

表7. 成熟度判定結果

組織	成熟度
A	レベル2を満足していない
B	ほぼレベル2を満足している

表7の結果から該当組織には成熟度レベルに応じたKPAに対する問題指摘が可能であり, 改善提案の絞込みが行える.

4.3. 成熟度把握結果の考察

我々は, PMMACによる2つの組織のプロセス調査結果から次の事を確認した.

PMMACによる調査結果から, レベル2に達していないと評価された組織Aは, 開発プロセス定義を詳細に実施しているトップダウン的な文化で, レベル2と評価された組織Aは組織的な定義よりプロジェクトの自主性を重んじるボトムアップ的な文化と分析できる.

一般にプロセス定義が曖昧な組織の場合, 明確なプロセス定義の推進を改善提案としがちである. ところが, PMMACの調査結果からプロセス定義が曖昧なボトムアップ文化の方が実施レベルの実現度が高く, トップダウン文化の方が実施レベルの実現度が曖昧であることが分かる.

そして, 社内の別の調査からもトップダウン文化の問題プロジェクト発生率が高いという報告がある. つまり, PMMACの結果は, 安易なプロセス定義よりもプロセス活動の定着を優先すべきである事を示唆している.

このことはCMMの正当性を裏付けていると考えられる. つまり, 組織Aは, 成熟度レベルが1であるにも関わらず成熟度レベルをスキップしレベル3のKPAを実施していると考えられる. CMMは上位成熟度レベルのKPAを下位レベルにいる組織が実施するのは効率が悪いとしている. 前述のとおり, トップダウン組織では問題も多くCMMの指摘が符合する.

5. PMMAC分析の評価

ここで, PMMACによる成熟度把握の結果をその効果面でのターゲットにしたアンケート方式のプロセス把握方式と比較することによって評価する. ただし, アンケートのサンプル数の問題から組織的な取組みの状況判断が困難であったため, プロジェクトレベルのプラクティスで判定できる成熟度レベル2の把握結果で評価を行うこととした. 評価手順は, 以下のとおりである.

① 表6の対応表を個別のプロジェクトデータに適用し成熟度レベルを把握する.
② SEIがCMMのアセッサー用に用意したレベル判定用簡易アンケート(クェッショネ-アー V1.0)[2]を用いて①のプロジェクトに対して調査を行う.
③ アンケート結果と PMMAC による成熟度把握結果とを比較評価する.

前述の手順①に従いPMMACによる成熟度把握を行った結果を表8に示す. 調査対象プロジェクトはABCの3プロジェクトを選択した. 選択基準はフィールド品質の差で, ABCの順に良い, 平均的, 悪いの水準である.

表8. PMMAC 手法によるプロセス成熟度把握結果

プロジェクト	RM	SPP	STO	SSM	SQA	SCM
A	1.0	1.0	1.0	1.0	0.9	0.9
B	1.0	1.0	1.0	1.0	0.7	0.7
C	0.0	1.0	0.5	0.0	0.0	0.0

次に手順②のCMMの簡易アンケートによるアンケート調査を上記3プロジェクトに対して行う. なお, アセッサーは正式ではないが, アンケート結果を複数のプロジェクト関係者にレビューすることで客観性を高めた.

なお, アンケート結果は定量評価するため表9に示す基準で採点した.

表9. 採点基準

基準	点数
充分に励行している	2
部分的に励行している	1
励行していない	0

そして，結果をKPA毎に集計し式2を用いて百分率で表すことによってﾌﾟﾛｾｽを把握した，ｱﾝｹｰﾄ結果を表 10 に示す．

KPA 実施率(%)
=(回答の採点集計結果)/(全質問項目数×2 点)
・・・(式2)

表 10. ｱﾝｹｰﾄによるﾌﾟﾛｼﾞｪｸﾄ評価

ﾌﾟﾛｼﾞｪｸﾄ	RM	SPP	STO	SSM	SQA	SCM
A	63	93	93	100	88	69
B	63	93	93	100	81	44
C	29	36	7	38	19	13

表8と表 10 の結果を比較するとPMMACによる成熟度把握は大まかではあるが，特徴は捉えておりｱﾝｹｰﾄ方式に代替できると考えられる．

また，ｱﾝｹｰﾄ方式に掛かる工数を考慮するとその結果は十分に満足できるものである．

以上よりPMMACによるｱｾｽﾒﾝﾄは，当初設定した目的である①組織の全体像を映し出す②継続的なﾏﾈｼﾞﾒﾝﾄのｺﾐｯﾄﾒﾝﾄを生み出す③ﾌﾟﾛｾｽ改善計画作成時にｱｾｽﾒﾝﾄ結果を有益な情報減として利用するという3点はクリアしていると判断できる．

6. まとめ

6.1. ｾﾙﾌｱｾｽﾒﾝﾄ手法

今回行った作業手順を再度以下に示す．

① SLCP-JCF98 の生産物定義と作業標準の生産物定義を突合しﾌﾟﾛｾｽの静的側面を把握する．
② さらにその生産物定義と生産管理ﾃﾞｰﾀﾍﾞｰｽを突合させることによってﾌﾟﾛｾｽの動的な側面を把握する．
③ こうして把握したﾌﾟﾛｾｽの効果的な分析をｻﾎﾟｰﾄするﾁｬｰﾄ化を行う．
④ 生産管理ﾃﾞｰﾀとCMMのKPAを突合させることによってCMM流のﾌﾟﾛｾｽ成熟度を把握する．
⑤ 把握したﾌﾟﾛｾｽ成熟度を元に問題定義を行い改善提案する．

上記作業手順は，作業標準や生産管理ﾃﾞｰﾀが整備されている別組織においても適用が可能である．その意味で我々は，この手順によって，効率的なｾﾙﾌｱｾｽﾒﾝﾄの手法が提案できたと考える

6.2. 生産管理ﾃﾞｰﾀの有効活用

今回我々は，生産管理ﾃﾞｰﾀのﾃﾞｰﾀ内容には言及せずﾃﾞｰﾀの有無にのみ注目して分析を行った．この事により，これまではｻﾝﾌﾟﾙ外とした低精度のﾃﾞｰﾀや，ﾃﾞｰﾀの欠損さえも情報としての活用を可能としﾃﾞｰﾀの有効活用を実現した．

この手法は，完璧なｱｾｽﾒﾝﾄを目的としたものではないが，少ない経営資源によってｱｾｽﾒﾝﾄの基本機能は満足していることが証明できた．また，今回提案する手法はISO化を受けて今後拡がりが予想されるSPAによるﾌﾟﾛｾｽ改善で必要となるであろう戦略的ﾌﾟﾛｾｽ改善ﾓﾃﾞﾙ作成への基礎情報として期待される．

7. 今後の課題

今回の手順はｼｽﾃﾑ化が可能であり，さらなる効率化が期待できる．今後は，継続的なｱｾｽﾒﾝﾄによる効果の享受を目指し、手順の完全自動化を図りたい．

また，今回の取組みでは独自の改善ﾓﾃﾞﾙがない事からCMM を適用することで問題解決を図った．しかし，業界の競争は激化しておりさらに戦略的な改善ﾓﾃﾞﾙの提案が望まれている．そのためには定量的なﾌﾟﾛｾｽの分析が不可欠で，生産管理ﾃﾞｰﾀの継続的な分析とﾃﾞｰﾀそのものの精度向上が必須である．

参考文献

[1] Mark C. Paulk, Charles V. Weber, Suzanne M. Garcia, Mary Beth Chrissis, Marilyn Bush, 能力成熟度ﾓﾃﾞﾙのｷｰﾌﾟﾗｸﾃｨｽ 1.1 版 CMU/SEI-93-TR-25, ソフトウェア技術者協会

[2] 第15年度ｿﾌﾄｳｪｱ品質管理研究会 分科会報告書 財団法人日本科学技術連盟

[3] SLCP-JCF98 委員会編, 共通ﾌﾚｰﾑ 98—SLCP-JCF98—(1998 版)ｿﾌﾄｳｪｱを中心としたｼｽﾃﾑ開発および取引のための共通ﾌﾚｰﾑ国際規格適合, 通産資料調査会,1998.10.28

[4] Robert B. Grady 著 古山恒夫・富野壽監訳, ソフトウェアプロセス改善ｺｱｺﾝﾋﾟﾃﾝｽ獲得へのｽﾊﾟｲﾗﾙﾓﾃﾞﾙ 共立出版

汎用事務処理フレームワーク Gofo その概要と経験

酒匂 寛 *

平成 14 年 6 月 9 日

概 要

Gofo は事務処理システム構築のためのフレームワークである．フレームワークが提供するのは，上流のビジネスモデルと具体的なシステム実装の間のギャップを埋めるためのアーキテクチャと，そのアーキテクチャ内に統合されるべき構成要素の定義方法である．Gofo を用いることにより，仕様から設計への追跡性を高めることが可能となる．本論文では Gofo の概念と Gofo を用いたアプリケーション構築の手順を述べる．また実プロジェクトへの適用経験についても報告する．

1 はじめに - Gofo のようなものが必要とされる背景

Gofo は事務処理システム構築のためのフレームワークである．本論文では，Gofo の概念と，抽象クラスの意味を解説し，そのあと Gofo を用いた実際のアプリケーション構築の手順を説明する．

Gofo は業務の世界と，実装の世界を無理なく接続するための仕掛けとして考案された．通常のシステム構築手法では，業務仕様の世界と実装の世界は遠く隔たっている．仕様とプログラムの距離が遠いと何が問題になるだろうか．一番困ることは，業務レベルで発生した問題を実装に反映させようと考えた場合に，しばしば間違いが起きやすく，かつ既存の資産もうまく活かせない場合が見られるということである．特に設計者がプロジェクトを去ってしまうと，業務仕様の変更が実際のシステムのどの部分に反映されるべきかの手がかりを追うことが難しくなる．

システム設計，プログラム設計，そしてプログラミングという段階を踏んで実際に動くシステムは構築されるが，最終的に作成されたプログラムコードと業務の専門家が検討していた仕様との対応関係は，しばしばわかりにくいものになりがちである．

これは「業務仕様」を実際のプログラムで実現する「過程」が，設計者（デザイナ）の頭の中だけで行われていて，その結果（設計）だけが目に見えるものとして残されているためである．

仕様をまとめるのは業務の流れに精通した人達であり，設計実装を行うのはシステムに精通した人達である．仕様をまとめる人達は様々な仕様記述の手段を用いて（例えば簡単なメモから，UseCase や UML のクラス図，VDM++ などの形式仕様記述言語等々），実現したい「業務」を記述する．システムを実現する人達はこの実現したい業務をいかに計算機の世界で実現するかを考える．Gofo は業務とシステムの間のギャップを埋める中間的なモデル層を提供することにより，仕様からプログラムに移る段階の距離を縮める役割を果たす．

2 Gofo が提供する抽象化

分散事務処理システムで多く利用されるのはいわゆる３層（多層）モデルである．これはシステムをユーザーインターフェイス，業務論理，データベースの階層に分離し、分散化や業務論理の再利用に供しようという目的を果たすために考えられたものである (図 1)．

確かにこの抽象化の階層は、システム構築を単純にし、それぞれの階層の構築を容易にする役割を果たしてきたし、実際の事例も積み重ねられてきている。

しかしながら注意すべきは、この抽象化の階層は，あくまでもシステム構築のための抽象化であって，必ずしも業務を構築しようとするものにとって，分かりやすいものであるとは限らないということである．

*Designers' Den Corp., sakoh@ba2.so-net.ne.jp

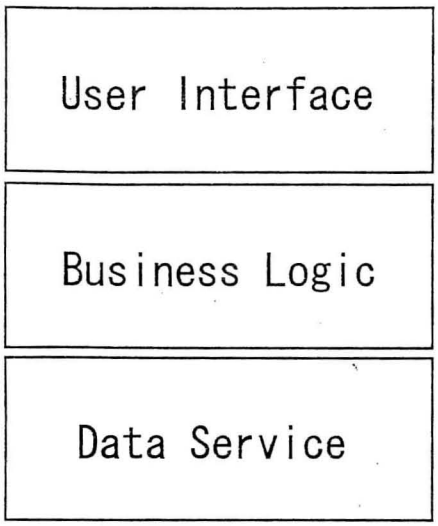


図 1: 典型的な 3 層モデル

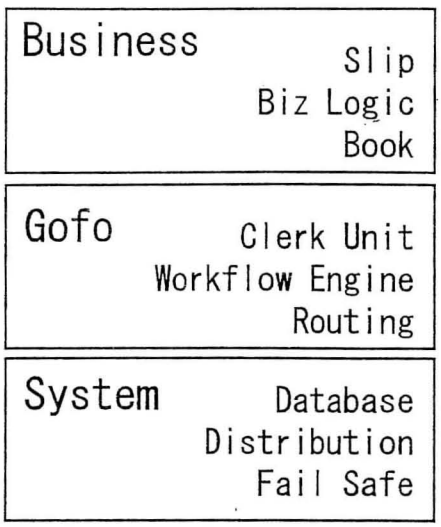


図 2: Gofo の抽象階層

業務要件を出す側にしてみれば，典型的な 3 層モデルの業務論理層が提供するインターフェイスの仕様書を眺めても，それが要求仕様書と一致しているか否かを検証することは難しい．いきおいその検証は設計・実装がかなり進んだ段階で行われる結合テスト以降に行われることになりがちである．

また開発者側の視点から見ても，妥当性が検証されない仕様を元に開発を進めることは大きなリスクが伴う．また要求仕様上に発生した変更が，確かにシステム仕様以降に反映されているのかどうかの判定が難しいことになる．

Gofo も抽象化の階層を提供するが，それは上記のような「システム構築」のためだけの抽象化ではない．システム構築のための抽象化とは直交した概念である．

Gofo が提供するのは，(1) 業務の世界にふさわしい抽象化コンポーネントを埋め込むための場所と，(2) そうして埋め込まれたコンポーネントを詳細な実装技術からは独立して組み合わせることを許すインターフェイスである．Gofo の利用によって，業務要件の世界での記述と，システム実装の世界での記述をより無理なくつなぐことが可能になる (図 2).

3 Gofo が想定する世界とそのモデル化

Gofo が想定するのは以下のような世界である．

1. 世界には伝票 (Slip)，事務員 (Clerk)，帳簿 (Book) が存在する
2. 事務員は到着する伝票を次々に処理し，結果を伝票として次々に送り出す
3. 事務員は伝票を処理する過程で，必要に応じて帳簿を参照したり，書き加えたり，訂正したりする
4. 各事務員は，到着する伝票に対してどのような仕事を行えば良いかの業務手順書 (BizLogic) を与えられていて，それにしたがって業務を行う
5. 世界にはこうした特定の仕事のやりかたを知っている複数の事務員が存在し，お互いに伝票をまわして，特定の目的を実現する

さて，ここである特定の Clerk に着目してみよう．一人の Clerk は複数種類の伝票を受け付けて，それに対する個別の処理を行うことが期待されている．ある Clerk に対して，処理すべき伝票と，それに対応する業務手順書を与え，情報の入出力のためのトレイを設定した一塊のオブジェクト (またはクラス) を事務員ユニット (Clerk Unit) と呼ぶ．

Clerk Unit は業務処理の基本を提供する．Clerk Unit 同士は，伝票の送付・受領という関係で結ばれているが，実際の伝票配送先は Clerk Unit の仕様とは独立に，Clerk Unit の外で提供される．このことにより，基本的な業務処理とワークフローを分離して定義することが可能となる．

4 Gofo を構成する抽象クラスとその関連

Gofo は複数の抽象クラスの集合体である．ここでは具体的なシステム毎に再定義されることを期待されている主要な抽象クラスを解説する．

4.1 Slip - 伝票

伝票を表すクラス．内部に必要なデータ項目を抱えている．属性を読み出したり設定したりすることができる．場合によっては伝票に対して簡単な問い合わせや計算（合計算出など）の仕事を依頼することができる．

実現方法は自由であるが，各 Clerk Unit の中ではその内部に合わせた形式になる．

4.2 Clerk - 事務員

事務員を表すクラス．伝票と手順書の対応関係を知っていて，到着する伝票に対して適切な業務手順書を適用する．

また事務員は業務手順書を適用する際の例外処理の方法を知っている．業務手順書自身は伝票の間違いを指摘することはできるが，その間違いが発見されたときにどうすべきかは一段「上」の事務員のレベルで判断する．

4.3 Book - 帳簿

帳簿を表すクラス．伝票の処理の結果生み出される，記録すべきものはここへ記入される．一般的な実装では RDBMS によって実現される筈であるが，それに縛られるわけではない．キーを与えてレコードを参照したり，挿入，更新を行う等のインターフェイスを持っているが，内部で，例えば join をしているか，等の詳細はこのレベルではわからない．

4.4 Tray - トレイ

事務員が使う入出力のためのトレイを表すクラス．事務員は自分への仕事の依頼を取り出すためのインレイと，仕事の結果を置くためのアウトトレイを持っている．ここへどのように伝票が置かれるか，あるいはここからどのように伝票が持ち去られるかについては，事務員は何も知らない．

4.5 BizLogic - 業務手順書

業務手順書を表すクラス．仕事の本質を表したもので，普通は伝票と一緒に設計される．ただし実際の伝票と業務論理の対応関係は個々の事務員ユニット内で定義される．

4.6 Carton - カルトン

事務員が業務手順書を適用している際に，必要となる様々な付加情報をつむための「お皿」を表すクラス．ある瞬間の Carton の上には，現在処理している伝票，現在の処理に必要な帳簿情報，一連の伝票処理を遂行するために必要な累積情報などが積み上げられている．

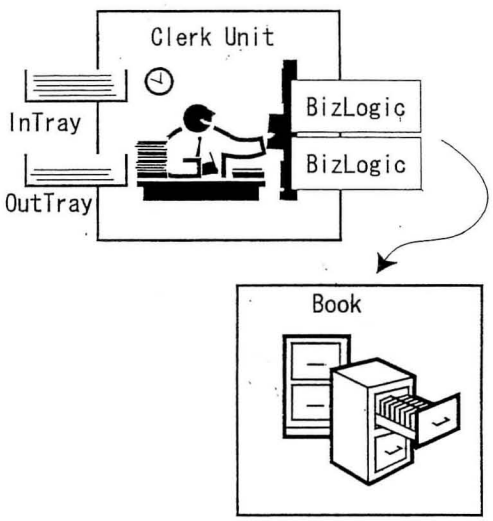


図 3: ClerkUnit の構成

図 3 に示したのが，概念的な Clerk Unit の構造である．Clerk Unit が使う Book （帳簿）は 特定

の Clerk Unit の占有物ではなく，複数の Clerk Unit から共有可能である．

5 アプリケーション設計手順

5.1 設計の前提

Gofo を使ったアプリケーション設計とは，ある仕事を果たす事務員ユニットを設計することである．すなわち事務員ユニットの構成要素を考えて，その構成部品を一つずつ用意していくことになる．

すなわち，業務の要求に従った仕様から，伝票と手順書（業務論理），カルトンそして帳簿（データベース）を抜き出して設計し，事務員ユニットに例外処理と共に統合する．他にも実際の情報配送手段や配送ルーティングに依存してトレイを用意する必要がある．

こうして設計・実装された事務員ユニットの間に伝票配送ルールを定義して実際に業務として仕事が流れるように組み上げられた時点でシステム構築が完了する．

前の方の節で，仕様をまとめる人達は様々な仕様記述の手段を用いて（例えば簡単なメモから，UseCase や UML のクラス図，VDM++ などの形式仕様記述言語等々），実現したい「業務」を記述する，と述べた．

このうち自然言語を使った仕様記述だけでは，設計へ移る段階での部品の品質を向上させることには限界がある．なぜなら仕様そのものが厳密に検証されていないため（目で追うウォークスルーだけでは不可能である），結局業務仕様の検証そのものがアプリケーション自身のテストに，しわ寄せのごとく押し付けられてしまうからだ．

このため Gofo の価値を高めるためには，少なくとも UML 等を使った構文レベルでのモデル化が必須なのであるが，本当に価値を引き出すには，意味レベルでも検証済みの仕様を与えるべきである．

後述するが現在の Gofo は，上流の仕様記述として VDM++ のような形式仕様記述言語と組み合わせてシステムの設計，実装に適用されている．

5.2 業務仕様からの部品の抽出

この節では前節で述べた事務員ユニットを構成するそれぞれの部品を業務仕様のどこから探すべきかを解説する

5.2.1 Slip - 伝票

仕様中の「業務論理」が取り込んで処理する情報を「入力伝票（InSlip）」として扱う．また他システムへ渡す情報は「出力伝票（OutSlip）」として扱う．特定のサブシステム内で用いられる伝票は，似通った形式を持つことが多く，また共通するメソッド（特定のデータフィールドを合算するとか，条件を指定して抽出するなど）を持たせやすいので，共通のスーパークラスを定義してそこに共通メソッドなどを定義しておくことができる．

5.2.2 Clerk - 事務員

事務員は現時点では事務員ユニットそのものである．事務員は「入力伝票」と「手順書」の対応関係を知っていて，到着する伝票に対して適切な業務手順書を適用する．よって,「業務仕様書」の中の伝票と業務論理の組み合わせの定義を探し，それを事務員クラスの中で対応つけるようにする．同じ入力伝票でも事務員が違えば異なる業務論理を適用するといったことができるが，本当にそうしたいかどうかは，業務設計自身の問題である．

また事務員クラスは，業務論理を適用した結果，例外処理をどのように行うべきかの方針も定義する（伝票を一枚スキップして残りの伝票処理を続行するか，残り全ての処理を中断するか，あるいはログを書き出すべきかなど）．

この定義は局所的な業務仕様書だけからでは読み取れない場合もあるので，全体の例外処理のポリシーと組み合わせて定義することが必要である．

いずれにしろ肝心なことは,「例外処理」と「業務論理」を分けて定義できるようにすることである．このことにより，運用と機能が分離されるため，構成要素の再利用が行いやすくなる．

5.2.3 Book - 帳簿

業務仕様に現れる情報モデルからこの部品の仕様を抽出する．情報モデルとしては ER モデルを仮定し，それを主にリレーショナルモデルへ写像する形で部品化を行う．

5.2.4 Tray - トレイ

業務仕様上にはこのクラスは現れない．このクラスは特定の情報配送メカニズムの世界 から，Gofo

の想定する世界へ伝票を出し入れするための仕掛けである．伝票の変換を行うことが主な仕事である．

また実際の処理効率を高めるために，入力伝票を使ったデータ先読み機構を，抽象化をあまり崩さずに入れるための場所がある．こちらの方は業務仕様を眺めて同じＤＢに対して繰り返しアクセスしている部分を取り込めないかを検討する．

5.2.5 BizLogic - 業務手順書

「業務仕様」の中核をなすクラス．業務論理はまず以下の４つのメソッドを再定義することを期待されている．

1. check
2. apply
3. export
4. summaryExport

BizLogic の check は，伝票をチェックし，必要なデータを編集し，あとは実際に帳簿を更新すればよいという段階までを行うメソッドである．ここで問題が起きた場合には，多くの場合その伝票の処理を中止し，残りの処理をどうすべきかを事務員に任せる．apply は check で編集された情報を使って帳簿（ＤＢ）を更新する．エラーが起きたときは，check の場合と同じく処理の続きの判断を事務員に任せる．業務仕様のうちにも，実際に様々な条件を検査したり事前条件を書いたりしている部分があるので，それを check に反映する．apply は事前に用意しておいた，意味のある業務（作業）の小単位を呼び出すだけのものとする．参照だけする業務論理の場合は check だけしか中身がない場合もあるだろう．

export と summaryExport は出力伝票の組み立てである．業務仕様で定義された「業務論理」が直接返す情報や，外部システムへの送信を指示されている情報が「出力伝票」として定義される．export では伝票を一枚処理するごとに出力される伝票を発行し，summaryExport は全ての伝票処理が終わったあとで，発行される合計伝票に責任を持つ．

5.2.6 Carton - カルトン

これも業務仕様には直接記述されないが，業務論理が使う外部データの蓄積を行う場所として利用する．設計上バランスが一番要求されるクラスである．

6 Gofo の適用

6.1 適用対象システム

Gofo は 日本フィッツ株式会社の証券バックエンドシステム - Trade One の一部サブシステムに適用されている．

なお適用に際しては Gofo をいきなり設計段階から用いたのではなく，上流の業務仕様の部分をまず形式仕様記述言語 VDM++ で記述し，そこから Gofo の各部品の仕様を前述の手順で抽出し，実装を C++ で行った．もちろん Gofo フレームワークを構成する抽象クラスもすべて C++ で準備されている．

Gofo そのものの設計と，VDM++ による仕様記述パターン開発が並行に行われたため，機械的な変換だけでは不十分な部分が残ったが，仕様記述から，Gofo としての実現方法への写像に関して，いくつかのパターンは見出されたため，開発の後段になるに従い設計の生産性そのものは上昇した．

現在はこのパターンをより洗練していこうとしている最中である．具体的には Gofo の部品仕様を導出するための（主に VDM++ による）業務仕様の記述要素とその形式の整理が進められている．

6.2 適用結果と考察

Gofo を適用したことによるメリットとしては，アーキテクチャとそこに適用する部品の形式が統一化されるため，プログラムを構成する際に個人ごとのばらつきが減ったことが挙げられる．

また同一構造と形式で記述されることにより，設計との対応関係，ひいては仕様との対応関係も追跡しやすくなるため，開発や保守効率に有利に働いたことが大きい．ただし，今回の適用では VDM++ 表記との対応関係は完全にパターン化されていないため，その分追跡性は少々失われている．

伝票や業務論理といった抽象化を行い，それに沿った部品化を行ったため，構築部品のパターン化が容易となり，データ検証やデータ抽出，データ更新，例外処理などの場所も明示的に示すことができるようになった．

Gofo の C++ 実装を用いた際の問題として発生したのは主に（１）C++ という言語の特性に由来するもの，（２）フレームワークを用いた開発への不慣れ．の２点から由来したものが多かった．

（１）の例としては，確保したオブジェクトの解放タイミングに係るものである．これは自動メモリ管理パッケージなどを購入したりすればあるていど解

消されるものと思われるが，それでも不慣れなプログラマは容易にメモリ関連の問題を引き起こすコードを書いてしまう．同様にちょっとした不注意が，用意に不正メモリアクセスなどを引き起こすため，細心の注意が必要となる．

また（２）としては，明示的な手続き呼び出しを行うのではなく，フレームワークが要求する機能部品を埋め込んでいくという開発スタイルが，通常のプログラマには最初のうちは馴染みにくいものであったようである．この問題は仕様書と対応する実装例が増えるに従って段々と解消した．

7 Gofo のこれから

7.1 トータルフレームワークの必要性と GSD

今回の報告ではシステム構築を直接支援するフレームワーク Gofo について解説したが，実際のプロジェクトでの使用を考えた場合単にフレームワークだけではなく，それを活用するためのより包括的な方法論が必要とされる筈である．

このより包括的な方法論は GSD (Gofo System Development) と命名されていて，現在 Gofo と形式仕様記述手法を組み合わせて試行されている最中である

GSD は（Gofo が対象とするものと同様な）分散かつ開放型の事務処理システム（基幹系，情報系）を構築するための方法論である．GSD そのものはまだ統合の端緒についたばかりで，流動的な部分も多いが，その内容を以下に概観する．

7.2 GSD の構成モデル

GSD の目標の一つは，上流から下流へのモデル要素のシームレスな適用である．すなわち上流で定義されたモデル要素は，次のステップで構築されるレイヤーの中に埋め込まれ，そのまま構築要素として利用可能になることが原則である．このことにより，上流ドキュメントは単なる読まれるだけの文書ではなくなり，積極的に保守される対象となる．GSD は以下の３つのモデルで構成されている．

- GBM - Gofo Business Model: 業務を記述するモデル．システムとしての実現を考慮せず，情報の出し入れと適用される業務論理に着目してモデル化を行う
- GSM - Gofo System Model: モデル化された業務を情報システムとして構築するためのモデル．この段階でも実装手段からは独立している．
- GFW - Gofo Framework: GFW を実装するための特異的なフレームワーク．実際のプログラム言語で記述されている．

7.3 GBM

GBM は業務のコアを切り出すためのモデルである．実際の実現手段のイメージから書き起こしてもよいが，その内容は入出力，適用される論理，そして永続的な情報という形式でまとめられる．GSD はこうした定義を，要求を出す側の直感に合わせるために，用語を工夫し，以下の三者を記述することを目的としている．

- 伝票 (Slip)
- 業務論理 (BizLogic)
- 帳簿 (Book)

伝票 (Slip) は外界との入出力情報をあらわすものであり，実世界では伝票，電文，画面，帳票，その他の入出力等に対応するものである．基本としては階層的なデータモデルとして表現される．

業務論理 (BizLogic) は伝票を処理し，後述の帳簿 (Book) 上に永続的な変化を加える．伝票を表す抽象構造と帳簿の上に現れる変化を厳密な式で表現する．

帳簿 (Book) は，業務を遂行するために必要とされる永続的なデータである．通常のリレーショナルデータモデルを表現したものと解釈することができる．

7.4 GSM

GSM は GBM で定義された業務のコアを実際のシステムとして展開するためのモデルである．GBM では業務を構成する基本の情報操作を定義しているが，実際のシステム構築に際しては，業務を遂行する人間，権限，場所，処理能力などを勘案して設計を進めなければならない．GSM はこうした要素を直接的な構築手段（プログラミング言語）などから分離して定義することを可能とする．このことにより，要求の変化に対する対応，実装技術の進化への追従が容易になるだけでなく，共通業務処理の再利用が容易となる．

GSM の構成要素は GBM を取り込んだ形で定義され，以下のようなものとなる．

- GBM
- 事務員 (Clerk)
- 配送路 (Routing)

事務員 (Clerk) は処理エンジンを抽象している．その役割は GBM で定義された「業務論理」を手順書として入力伝票を処理し，必要なら結果を帳簿に書き込んで連携のための出力伝票を書き出すことである．事務員に業務論理を対応付けたものを「事務員ユニット」と呼ぶ．ここで再利用の階層を考えると，最小単位は GBM で定義された伝票，業務論理，帳簿だが，ここで定義された事務員ユニットも，システムレベルでの解を再利用するための単位と考えることができる．

配送路（Routing）は上記で定義された事務員ユニットをどのように連携させるかを指定するものである．複数の具体的な事務員ユニットを連携させたものを，上位の事務員ユニットとして構成することで，更に大きな部品として利用することができるようになる．

7.5 GFM

GFM は GSM を具体的に実装するレイヤである．現在ある実プロジェクト用に開発した C++ を用いたものと，プロトタイプ開発用の Ruby を用いたものが用意されている．

なお蛇足であるが，実際に C++ と Ruby を用いた開発を並行させてみると，ほとんどの場合 Ruby による実装で性能的に遜色ないことが観察された．これは普通の事務処理のオーバーヘッドの大部分がデータベースのアクセスや通信などの部分に存在するためである．一方生産性そのものの比較では Ruby の方が C++ に比べ１０～２０倍高いことが観察された．

8 謝辞

Gofo の実プロジェクトへの適用の機会を提供していただいた日本フィッツ株式会社の関係各位に感謝致します．

参考文献

[1] 加藤貞行, 失敗のないシステム開発入門, 日経BP社 2000

[2] Jackson, M., システム開発 JSD 法, 共立出版株式会社 1989

[3] Cockburn, A., Agile Software Development, Addison-Wesley 2002

[4] Jackson, M., ソフトウェア博物誌, トッパン 1997

[5] Jackson, M., Problem Frames: Analysing and structuring software development problems, Addison-Wesley 2001

[6] Gamma, E. 他, Design Patterns, Addison-Wesley 1995

[7] Fitzgerald, J., Larsen, P.,Modeling Systems: Practical Tools and Techniques in Software Development, Cambridge University Press 1998

プロジェクト運営におけるエラー・プローン分析の活用に関する研究

日本アイ・ビー・エム株式会社ソフトウェア開発研究所
原 正雄 HARAMASA@jp.ibm.com
阿部 仁美 HITOMIA@jp.ibm.com
岡崎 毅久 TOKAZAKI@jp.ibm.com

要旨

エラー・プローン領域とは(本稿では，エラー・プローンと呼ぶ)とは，プログラム中で欠陥の多発するコンポーネントを指す．一旦システムに混入したエラー・プローンは，広範囲にわたりシステム動作に悪影響を与えるため，莫大な保守コストを発生させることが指摘されている．一定期間内に要求品質を達成するためには，できる限り早期にエラー・プローンを修復することが重要である．開発期間がますます短縮される中，エラー・プローンへの対処の遅れはプロジェクトの破綻を招きかねず，早期対応の重要性はますます高まっている．限られた時間と人員の中でエラー・プローンに対処するためには，エラー・プローンを高精度で特定する分析手法と，エラー・プローンに対して柔軟に対応するプロジェクト管理が必要不可欠である．

本稿では，エラー・プローン分析に重点を置いて運営されたプロジェクトを紹介するとともに，そのプロジェクトで実施したエラー・プローンの分析の内容および，その有効性を述べる．

1. はじめに

人間が過ちを犯す場合，ランダムに間違うのではなく，ある傾向をもって間違えると言われている．開発作業のほとんどを人手に依存するソフトウェアでは，この影響を強く反映し，欠陥は一様に分布するのではなく，特定のコンポーネントに偏在するとことが知られている[1]．この欠陥の集中するコンポーネントは，エラー・プローン・コンポーネント(以後，エラー・プローンと記述する)と呼ばれ，一般に，非常に複雑で不安定な構造をもつため，システム動作に悪影響を与え，莫大な保守費用を発生させることが指摘されている．一旦システムにエラー・プローンが混入すると，その影響範囲は多岐にわたるため，エラー・プローンの修復は非常に困難となる．限られた時間の中で要求品質を達成するためには，できる限り早期にエラー・プローンを特定し，これを修復することが重要である．

しかしながら，実際の開発プロジェクトでは，計画されたテストの消化に主眼が置かれ，品質の監視が軽視されがちで，また，仮にエラー・プローンの兆候がみられても，実際の対策まで踏み切ることができない場合が多い．そのため，テストの後段で欠陥が噴出して初めてエラー・プローンに対処するケースが数多く見受けられる．エラー・プローンへの対処の遅れは工程の後戻りを余儀なくされ，時間，予算の面でプロジェクトの破綻を招きかねない．したがって，テスト工程ではエラー・プローンの除去に常に注力すべきである．

エラー・プローンの特定は，テスト工程における動作検証を通じて，ソフトウェア全体の挙動を把握し，その中で特異な傾向がみられる箇所を指摘することによる．したがって，早期にエラー・プローンを特定するためには，まず，エラー・プローンにみられる傾向を的確に捉えるための分析手法が前提となる．それに加え，現場レベルでの日常的な品質の監視とともに，プロジェクト運営の視点から，ソフトウェア全体の挙動を早期に把握できるようなテスト計画を立てることも重要である．

過去，我々はエラー・プローンを特定するための様々な分析手法(エラー・プローン分析)を提案してきた[2]．今回，以下を目的とする中間時点を設け，エラー・プローン分析をプロジェクト運営に積極的に活用した．

- 製品全体をカバーするデータを早期に収集する
- 多角的にデータを分析しエラー・プローンを特定し，これを集中的に修復する

本稿では，プロジェクトの概要，およびスケジュールを述べた後に，プロジェクトの中で実施した分析について説明し，それらがエラー・プローンの特定に有用であることを述べる．

なお，本稿では，個々のコンポーネントを組み合わせ，仕様の実現度合いを検証するテストを FVT (Function Verification Test)と呼び，これは，一般に呼ばれる機能テスト，統合テストに相当する．また，実際に顧客環境を想定し，要求の実現度合いを検証するテストを SVT (System Verification Test)と呼び，これは，一般に呼ばれるシステム・テストに相当する．

2. 事例紹介

2.1. 製品概要

今回紹介する事例は，過去に開発されたPC上のアプリケーション開発ソフトをオープンな統合開発環境である eclipse[3]上に統合するためのプロジェクトである．この製品は表 1に示すようなコンポーネントにより構成されている．製品に対して，前バージョンの機能性を削ぐことなく操作性の向上が求められる一方，開発期間は，前バージョンと比較し半分に短縮され，さらにテスト期間は 1/3 にまで圧縮を迫られていた．

表 1:適用製品のコンポーネント構成

コンポーネント	開発形態	機能
a	再利用	データ変換
b	新規	他のコンポーネントを統括
c	新規	グラフィック系エディター
d	新規	データベースと連携
e	新規	テキスト系エディター
f	新規	データベースと連携

2.2. スケジュール

通常，テスト期間からテスト量が割り出されることが多いが，そのようなテストの割り出し方では，テストの作業量が平均化され，テストの途中で，特定のコンポーネントに重点を移すことが難しい．

今回のプロジェクトでは，図 1のスケジュール略図に示すように，エラー・プローンに柔軟に対応するため，事前に用意するテストケースは，機能テスト期間の半分程度で実行可能な量にとどめ，それらの実行後，品質の検証を通じて再度テストケースを追加した(重点テスト)．各フェーズでは次に述べる事項を目的として，全 6 週間のテスト工程を分割した．

FVT1 (2 週間)

- 事前に用意したテストケースを実行する．

事前に準備するテストケースの分量は，中間分析までに実行が完了する程度とする．ただし，FVT1 までに実行することができなかったものは FVT2 に持ち越す．

中間分析

- FVT1 のテスト結果を多角的に分析し，エラー・プローンを特定する．
- 特定されたエラー・プローンに対し，特殊ケースや，エラー・ケースを中心にテストケースを追加する

FVT2 (2 週間)

- 中間分析で追加したテストケースを重点テストとして実施する．

エラー・プローンは，上流工程における設計上の問題を抱えることが多く，発見された欠陥に対してその場限りの修正を加えても，副作用を誘発しやすい[4][5]．この重点的テストは，細部に潜む欠陥を一度に表面化することで設計上の問題を顕在化し，それらの修正を通じてより堅固設計へと修復することをねらっている．

SVT (2 週間)

- エラー・プローンが除去されたことを確認
- 製品の最終品質レベルを確認

実用環境のもと，実際に顧客で使用されていたデータを用い，製品全体を満遍なくテストし，最終的な製品品質を評価する．

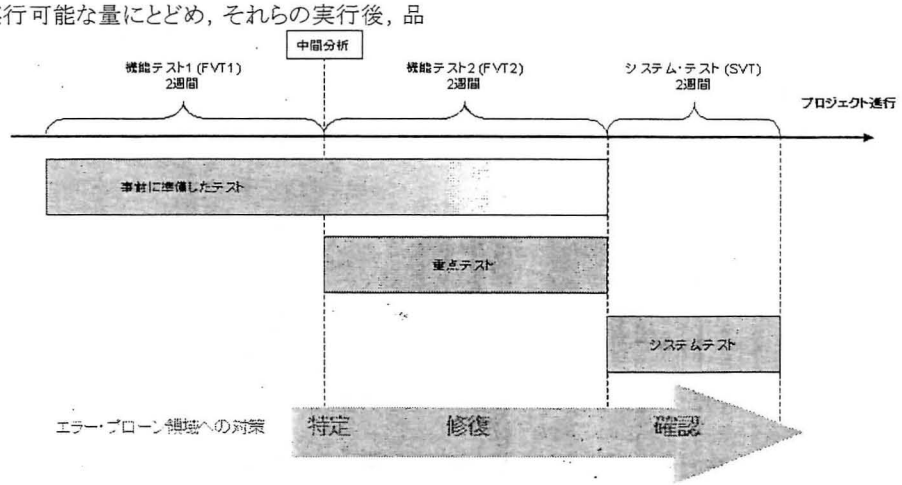


図 1:テスト工程における活動とエラー・プローンに対する対策との関係

2.3. 中間分析

2.3.1. 定量的な評価

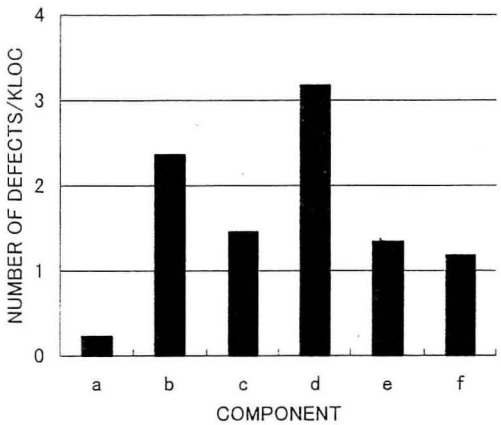


図 2:CP1 における欠陥密度の状況

図 2は，中間分析の時点における欠陥密度を示したものである．図から明らかなように，

- コンポーネント b, d の欠陥密度が相対的に大きい

ことが読み取れる．しかしながら，最も大きな値がみられたコンポーネント d においても，絶対値そのものは，過去の経験値と比較して特に警戒すべき値ではなく，この定量的な指標である欠陥密度のみからコンポーネント b, d をエラー・プローンと断定するのは難しい状況にあった．そこで，欠陥の特徴，欠陥の修正内容といった，より定性的な側面から分析を施し，コンポーネント毎の品質を評価した．

2.3.2. 定性的な分析による品質評価

一般に，定性的な分析方法は，内部領域，外部領域に区別される．前者はホワイトボックス的にソースコードの修正状況を観察するのに対し，後者はブラックボックスに的に機能面，欠陥の発見条件などで欠陥を分類するものである．ここでは，実施した様々な分析の中で特にエラー・プローン特定につながった内部領域，外部領域の分析結果について述べる．

内部領域(コード絶対変化量)

内部領域の分析としてコード絶対変化量を測定した．コードの絶対変化量とは，コードの更新行数，追加行数，削除行数の総和を意味する．コードサイズを観察する典型的な方法として，単にコード行数の変化量を観察するものがある．しかし，この方法では，50 行削除して，50 行追加して，50 行更新したような場合，見かけ上，行数には何も変化がないように見え，コードの修正状況を把握する上で正確性に欠ける．

それに対し，コードの絶対変化量を観察する方法では，コードの更新行数，追加行数，削除行数を別々に計測しているため，実際にコードに加わった修正量を正しく把握することができる．

この分析では，絶対変化量に占める更新行数の割合が大きなコンポーネント，つまり更新が発散傾向にあるコンポーネントはまずエラー・プローンであることが疑われる[2]．したがって，コードの絶対変化量に占める更新行数の割合（以下では，更新量の割合と述べる）を計測することで，対象コンポーネントの安定性を評価することができる．

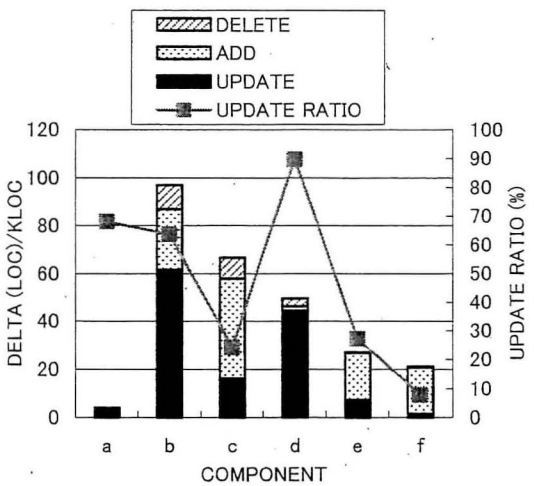


図 3:コードの絶対変化量，および，コード絶対変化量に占める更新行数の割合

図 3は，サイズ当たりの絶対変化量をビルド毎に集計し，足し合わせた累積的なコード絶対変化量と，その絶対変化量に占める更新の割合を示したものである．図 3から特徴的な傾向がみられたコンポーネントについて以下のように評価することができる．

- コンポーネント a

更新量の割合は大きいものの，コードの絶対変化量は非常に小さいため，品質には問題ない．

- コンポーネント b

コードの絶対変化量が最も大きく，更新量の割合も比

較的大きいことから，非常に安定性に乏しく，エラー・プローンである可能性が高い．

- コンポーネント c

コードの絶対変化量はコンポーネント b に次いで大きいものの，更新量の割合はさほど大きくない．したがって，現段階では安定性を危惧するには至らない．

- コンポーネント d

コード絶対変化量は多くないが，更新量の割合が非常に大きく，エラー・プローンであることが疑われる．

外部領域（ODC 分析）

一方，外部領域からの定性的分析として ODC の修正タイプを評価した．弊社では，外部領域からの分析として ODC (Orthogonal Defect Classification) 分析が積極的に用いられている[6]．ODC 分析とは，個々の欠陥を分類するための手法を指し．欠陥の発見時，修正時の状況や内容などに 7 通りの分類方法が定義されている．それぞれの分類方法により欠陥を分類し，得られた分布傾向からテスト効率や品質を診断することができる．Orthogonal とは「排他的な」という意味であり，ODC 分析の特徴は，それぞれの分類方法が互いに独立して評価できる点にある．

ODC 属性の一つである修正タイプ属性（以後，修正タイプ）は，「何を修正したか」により欠陥を分類するもので，分類結果からコンポーネントの安定性を評価することができる．実際には，修正タイプに対して複数の項目が定義されているが，ここでは簡略化のため，それら項目を比較的単純なミスに起因する欠陥である EASY と，より複雑な修正を必要とする欠陥である COMPLICATE の二つに分類して示している．EASY，COMPLICATE それぞれは，次に挙げるような項目を含む．

EASY
- 値のチェック
- 値の初期化，代入部分

COMPLICATE
- アルゴリズムの修正
- 同期処理
- 構造，インターフェースの変更

EASY が大部分を占める場合，安定性に問題がみられる場合が多い[7]．

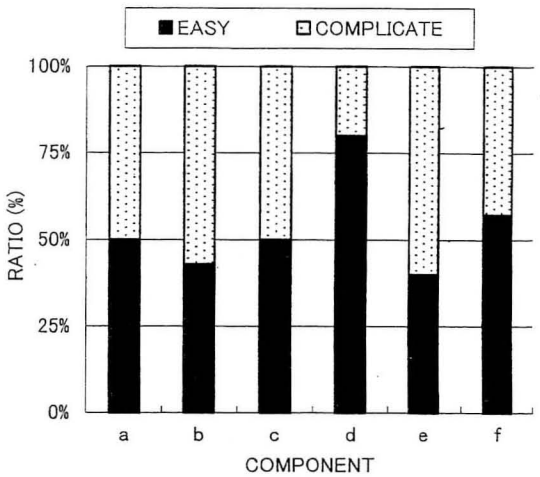


図 4：ODC 分析（修正タイプ）による分析結果

図 4は，修正タイプによりコンポーネント毎の欠陥を分類したものである．この分析により，

- コンポーネント d で，他のコンポーネントと比べ，EASY の割合が大きい

ことが明らかとなった．

図 2でみられたように，コンポーネント d は，最も欠陥が集中したコンポーネントであったが，その一方，図 3に示すコードの絶対変化量は，それ程大きな値はみられていなかった．つまり，コンポーネント d は，少量のコード変更により修正可能な欠陥を多く含んでいたことを意味し，そのような欠陥の多くは，比較的単純な誤りに起因するものが多いと考えられ，図 4の修正タイプでみられた傾向と一致する．これにより，外部領域の修正タイプに加え，内部領域からもコンポーネント d の安定性に問題があることが予想され，コンポーネント d が，エラー・プローンである可能性が高いと言える．

2.3.3. テストケース追加のための分析

表 2：中間分析結果

	a	b	c	d	e	f
欠陥密度		△		△		
定性分析（内部領域）	△	×	△	×		
定性分析（外部領域）				×		

表中の「×」は，エラー・プローンの兆候がみられたことを表し，「△」はエラー・プローンにみられる何らかの特徴が観察されたことを表す．

表 2は，以上述べた中間分析の結果をまとめたもの

である. 実際には, 以上に紹介した以外の分析も実施しているが, 表 2に挙げる分析で最も明確な傾向がみられたためこれらを重視し, コンポーネント b, d をエラー・プローンと位置付けた. この中間分析の結果に加え, コンポーネント b, d それぞれに対して, 以下のような評価を加えテストケースを追加した.

コンポーネントb:

比較的大きなコンポーネントであることから, より詳細な内部, 外部領域の分析を実施し, エラーの潜む領域を機能レベルまで絞り込んだ.

内部領域の分析にあたる図 5は, 機能エリア毎にソースコード修正回数を分類したものであり, また, 外部領域の分析にあたる図 6は, 機能エリア毎に欠陥数を分類したものである. 分析の結果,「新規作成機能」および「ソース生成機能」に欠陥が集中していることが判明し, これらの機能を中心にテストケースを追加した.

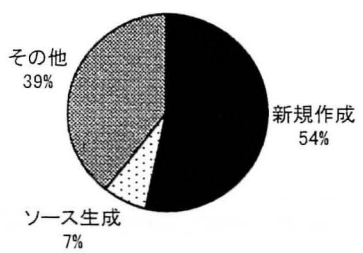


図 5:コンポーネント b における内部領域からの分析結果(ソースに多くの修正が加わった機能エリア)

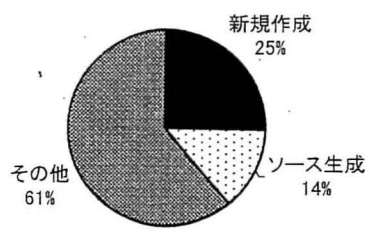


図 6:コンポーネント b における外部領域からの分析結果(欠陥が発見された機能エリア)

コンポーネントd:

中間分析から, 比較的単純な誤りに起因する欠陥を多く含んでいたことが判明している. この原因として, 開発者のスキルや, ユニット・テスト不足, スケジュール・プレッシャーといったことが考えられるが, いずれの要因も, コンポーネント全体の品質に影響を及ぼすことから, 特定機能ではなく, コンポーネント全体を網羅するように, テストケースを追加した.

2.4. テスト終了後における品質確認

2.4.1. 定量的な総合評価

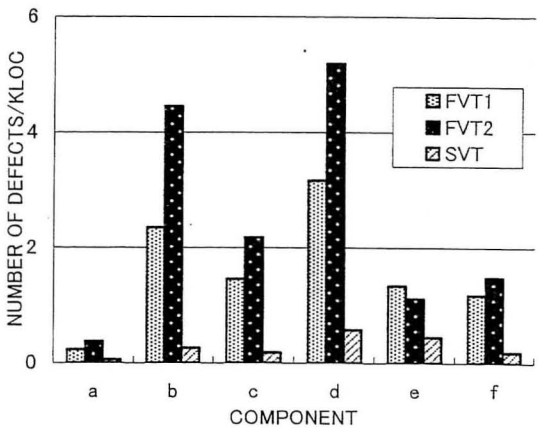


図 7:各テストフェーズにおける欠陥密度

図 7は, 各テストフェーズにおける欠陥密度をコンポーネント毎に示したものである.

図で, テスト工程全体を通じた欠陥密度(FVT1, FVT2, SVT の欠陥密度の和)に注目すると, コンポーネント b, d の値は, 他のコンポーネントを大きく上回り, かつ, その絶対値は過去の経験値と比べて非常に危険なレベルに達していた. したがって, 最終的な欠陥数から, コンポーネント b, d が中間分析で指摘したようにエラー・プローンであったことが確認される.

ついで, エラー・プローンであったコンポーネントb, d のフェーズ毎の欠陥密度の分布をみる. FVT2 ではエラー・プローンの欠陥密度は, 突出してみられたがSVTでは収束し, 他のコンポーネントと同等のレベルに落ち着いていることから, コンポーネント b, d が FVT2 の対策を通じて修復されたことが確認できる. また, 他のコンポーネントについても, SVT で欠陥の集中がみられなかったことより, エラー・プローンである可能性は低いと判断できる.

なおFVT2では, コンポーネントb, dに対して集中的にテストを実施したため, 工数増大が欠陥数増大につながった可能性もあるが, SVT では実用環境のもと, 実際に顧客で使用されていたデータを用い, 製品全体に対して満遍なくテストを実施している. したがって, FVT2 における欠陥数は, 工数増大よりも, コンポーネ

ント b, d が含有していた欠陥数そのものを反映した結果だと言える.

以上は定量的な評価であるが, 定性的な側面に注目することにより, エラー・プローンとその他のコンポーネントの差異がより明確になる.

2.4.2. 定性的な総合評価

定性的な側面を評価するため ODC 属性の一つであるトリガー属性(以後, トリガー)を採用した.

品質が未熟なコンポーネントは, 当然, 単純な操作下で欠陥に遭遇する可能性が高い. トリガーは, 欠陥がどのような操作で発見されたかを分類し, 品質を評価するものである. トリガーには, 下記のような属性が定義されているが, ここでは単純化のため, 比較的単純な操作で発見された欠陥を COVERAGE, それ以外を OTHER の二通りに分類している. 品質が未熟なほど, COVERAGE の割合が大きくなる[7].

ODC トリガー属性

COVERAGE:

- 正常ケース, 簡単な条件

OTHER:

- NULL, 境界値など特定の入力値
- 特定の順序
- 特定の組み合わせ
- 大量データ
- ソフトウェア/ハードウェア構成
- …

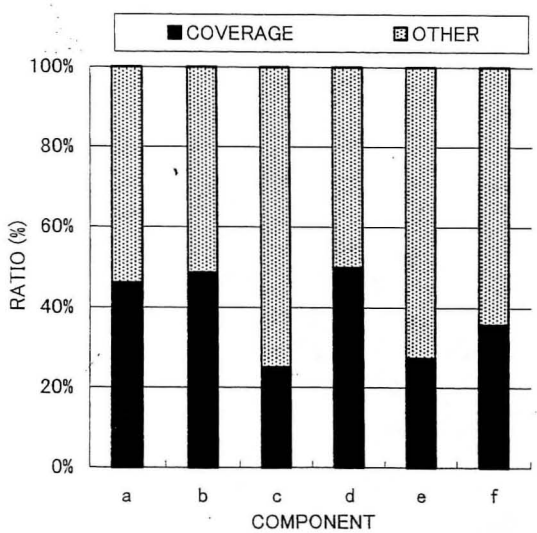


図 8:ODC 分析(トリガー)による分析結果

図 8は, テストを通じて発見された全ての欠陥のトリガーを, コンポーネント毎に比較したものである. 図から, エラー・プローンであった b, d が COVERAGE に分類される欠陥を多く含んでいることが分かり, このトリガーという定性的な側面からも, コンポーネント b, d の品質が未熟であったと言える.

3. 中間分析における定性的評価の重要性について

今回のプロジェクトでは, 無事スケジュール通りにテストを完了し, 目標品質を達成することができたが, それに大きく寄与した中間分析と, 重点テストの役割を評価するため, 時系列で定量的, 定性的な側面を観察する.

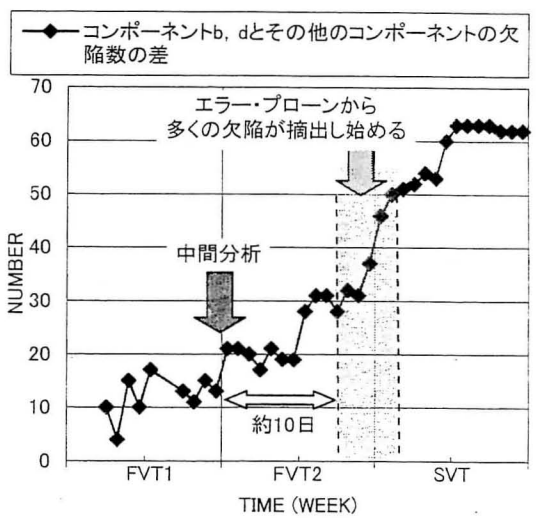


図 9:コンポーネント b, d の欠陥数とその他のコンポーネントの欠陥数の差の推移

図 9は, エラー・プローンであったコンポーネント b, d と, その他のコンポーネントにおける欠陥数の時間推移を比較したものである. 図では, 比較が容易なようにコンポーネント b, d から発見された欠陥数と, その他のコンポーネントで発見された欠陥数の差を示している.

図から, エラー・プローンとその他のコンポーネントとの欠陥数の差はテスト開始直後から徐々に開きつつあることが分かるが, 特に FVT2 の中間あたりから, その差が発散傾向になっていることが観察される.

中間分析の実施時期は, 発散傾向に移る約 10 日前にあたることから, 定量的な欠陥数に注目した場合と比べ, 今回の中間分析により, 早期にエラー・プローンを特定できたことが分かる. これから, エラー・プローン

の存在が，その欠陥数により確認できるタイミングは，中間分析で観察したような定性的な症状が現れるタイミングより遅れて表れると言え，品質分析における定性的な分析手法の重要性が確認される．

一方，定性的な側面について時系列的に観察したものが図 10である．図は，前述の ODC 属性のトリガーにおける COVERAGE の割合の推移をエラー・プローンであったコンポーネント b, d と，それ以外のコンポーネントとで比較したものである．この COVERAGE が収束してゆく過程をみることにより，エラー・プローンが修復されてゆく過程を観察することができる．なお，SVT の期間については，発見された欠陥数が非常に少なく，分析結果に統計的な誤差を生じていることが懸念されたため，評価に際して参考適度にとどめた．

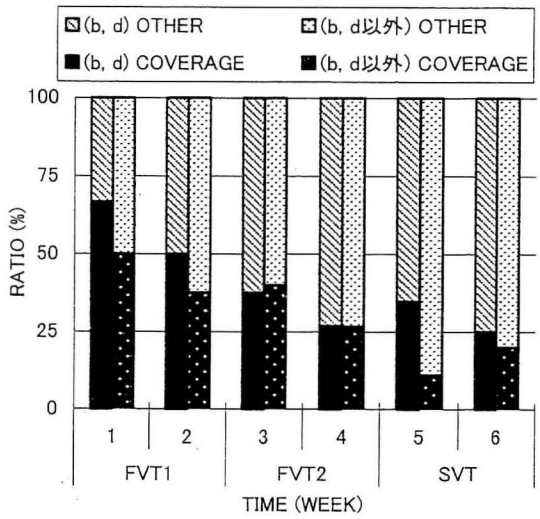


図 10:コンポーネントb, dと，それ以外のコンポーネントにおける ODC(トリガー)の推移

テスト工程全体の傾向をみたとき，いずれのコンポーネントにおいても，テストの進捗とともに，COVERAGE の割合が減少し，最終的に約 20〜30%前後に収束し，エラー・プローンをはじめ，他のコンポーネントも SVT までに安定な状態に至っている様子が確認される．

次いで，各フェーズの特徴に注目したとき，まずテスト開始直後の第 1, 2 週目の FVT1 において，エラー・プローンであったコンポーネント b, d とその他のコンポーネントの COVERAGE の割合の差が最も大きくみられることが読み取れる．FVT1 は，事前に用意したテストケースを実行した期間にあたり，コンポーネントの隔たりなくテストを実施している．したがって，テスト初期段階において，この ODC のトリガーのような定性的な側面に注目することにより，欠陥の量的な集中箇所としてエラー・プローンが顕在化する前に，正常なコンポーネントと，エラー・プローンとが区別可能であると言える．換言すれば，今回の中間分析のようにテスト初期のデータからエラー・プローンを特定する場合，エラー・プローンの特定精度の向上には，定性的な側面からの分析が不可欠であると言える．

また，その後のコンポーネントb, dのCOVERAGEの割合をみたときFVT2では他のコンポーネントと同程度になっている．この期間は，コンポーネント b, d に対して特殊ケースやエラー・ケースに焦点を置き重点的にテストしていた時期にあたることから，重点テストの工程増加が OTHER の増加をもたしたことが分かる．この重点テストを FVT2 で実施しなかった場合，コンポーネントb, dの OTHER に分類される欠陥が，SVT へ漏れ出し，SVT でのテスト品質，ひいては製品品質の低下をもたらした可能性が高く，重点テストが，エラー・プローンを安定な状態に導くことに有効であったと言える．

4. 考察

プロジェクトでは，以前からコード絶対変化量をビルド毎に自動的に記録する仕組みを構築している．また，ODC の各要素は，問題報告システム上で，必須入力項目としているため，問題発見時，もしくは問題修正時に必ず登録される．したがって，挙がった問題から分析材料を集計すること自体はさほど困難な作業ではなく．エラー・プローン特定に際しては，いかに有用なデータを収集し，かつ得られたデータをいかに分析するかが重要となる[8].

データの収集に関しては，今回の中間分析のように，プロジェクト運営の視点から区切りを設け，テスト工程の初期にソフトウェア全体の傾向をつかむための計画を立てることが有効である．テストの進行途中では，コンポーネント毎にテストの進捗にばらつきが生じるため，コンポーネント毎の相対的な比較は難しいが，中間分析を設け，テスト期間を細分化することで，限られた期間の中である程度まとまったデータを収集することができる．

また，データの分析に関しては，プロジェクトのメンバーとの対話が重要である．弊社では，過去の経験を蓄積し，データの傾向とその傾向を与えた要因を結びつける試みがなされている[6]. しかしながら，実際にみられるデータの傾向から一意にその要因が求まることは

少なく，データの解釈に際して，多角的なデータを収集するとともに，その中から実情に即したデータを抽出する作業が必要となる．特にプロジェクトの外部の立場から品質を評価する場合，プロジェクト内のメンバーと十分に対話し，それにより，具体的な根拠に基づきデータを評価することが重要である．今回紹介した手法は，十分なコミュニケーションの上に立ち初めて効果を発揮されるものである．

5. まとめ

テスト工程の早い段階で，品質を検証する中間分析を設け，ここで様々な角度から分析を実施した．この分析を通じてエラー・プローンを特定し，その後，重点的にテストを実施することにより，特定されたエラー・プローンを修復することができた．また，最終的な品質評価を通じて，中間分析における分析が的確にエラー・プローンを捉えていたことを確認するとともに，実施した分析がエラー・プローン早期発見に有用であることが確認された．

参考文献

[1] Carpers Jones，ソフトウェア品質のガイドライン，共立出版 (1999)

[2] 岡崎毅久，効率的な品質向上のためのエラー・プローン分析手法に関する研究，ソフトウェアシンポジウム (June, 1999)

[3] Eclipse Project (http://www.eclipse.org/)

[4] 岡崎毅久，ソフトウェア品質特性に基づいたシステム・テスト設計，(社)情報処理学会 第110回ソフトウェア工学研究会ソフトウェアシンポジウム (July, 1996)

[5] 岡崎毅久，他:，ソフトウェアのテスト・ケース設計に関する一考察，ソフトウェア技術者協会(SEA) ソフトウェアシンポジウム'95 (June, 1995)

[6] Center for Software Engineering Details of ODC v5.11 (http:// www.research.ibm.com/softeng)

[7] M. Butcher, H. Munro, T. Kratschmer, Improving software testing via ODC: Three case studies, Software Testing and Verification Vol. 41, No. 1, 2001

[8] 岡崎毅久，インプロセス・データ分析に基づいたプロジェクト管理－ 連携的なデータ分析による進捗管理と品質管理の実践，(財) 日本科学技術連盟 第 16 回ソフトウェア生産における品質管理シンポジウム (September, 1996)

第13回 テクニカル マネジメント ワークショップ

設 計 論

ー参加者募集ー

主催： ソフトウェア技術者協会

「ソフトウェア産業における真のマネジメントのあり方を，その本来の立脚基盤である技術を軸にして，根本的に問い直す」ために企画されたこのワークショップも,早いもので今回で１３回目を迎えることになりました．前回は，静岡県の熊(くんま)で熱い議論を展開しましたが，今年は，山形県の鳥海温泉に場所を移し，テクニカル・マネジメントの本質について，さらなる探求を行いたいと考えます．

情報システムの開発においては，CMMIに代表されるプロセス改善,プロジェクト管理のノウハウ集であるPMBOK，ソフトウェア工学のノウハウ集であるSWEBOK，XPに代表されるアジャイル等，いくつかの技術的トレンドがありますが，やはり、設計は未だに重要であり，完全な設計手法はないと想います．

このような状況下にあって，われわれソフトウェア技術者はいまどのように行動すべきか,基調講演に,奈良先端大学院大学の蔵川先生をお迎えし,情報システム開発での設計論について,蔵川先生の考察をお伺いしたあと,全員で多角的な討論を行いたいと考えています．

多くの方々の申込みをお待ちします．なお，討論の性格上，ポジションペーパや配布資料ならびに討論内容は非公開とさせていただきます．

開 催 要 領

期 日： 2003年 10月 2日(木) 午後から4日 (土) 正午まで (2泊3日)

場 所： 山形県飽海郡遊佐町・鳥海温泉「遊楽里」

定 員： 15名

参加資格： ソフトウェアの開発・管理に従事しておられる，30歳以上の管理者または経営者

費 用： SEA会員 **30,000円**， 一般 **40,000円** (現地集合・現地解散とし，期間中の宿泊費と資料代を含みます)

実行委員長： 田中 一夫 (JFits)

基調講演： 蔵川 圭氏 (奈良先端科学技術大学院大学)

運営方法： オープニング，基調講演のあと，参加者全員からそれぞれ20分程度の発表をしていただき，あとはフリー・ディスカッションを行います．

申込方法： 参加申込票に必要事項を記入の上，**9月22日(月)**までに， E-mail または FAx で，下記 申込み先 へお送り下さい．折り返し詳細な参加案内をお送りします．ただし，受付けは先着順とし，定員(15名)になり次第締め切らせていただきますので，あらかじめ御了承ください．

*** 申込み先： ***

〒130-6591 墨田区錦糸3-2-1 アルカイースト 日本フィッツ(株) 証券システム第二事業部 田中 一夫

Tel: 03-3623-4035, Fax: 03-3623-8779, E-mail: tanaka@jfits.co.jp

********** 第13回 テクニカル マネジメント ワークショップ 参加申込 **********

氏名（ふりがな）：

種別：☐ 正会員 ☐ 賛助会員 ☐ 一般 年齢： 才 性別：☐ 男 ☐ 女

会社名，部門/役職：

住所：（〒 － ）

TEL'& FAX:

E-MAIL:

ポジション ステートメント(ワークショップで討論したい問題，それについての意見，etc)：

********** Deadline: 2003/9/22, mail to: **tanaka@jfits.co.jp** **********

SEA & FSIJ 合同フォーラム

オープンソースをめぐる最近の動向

参 加 者 募 集

主 催

ソフトウェア技術者協会(SEA) & フリーソフトウェアイニシアティブ(FSIJ)

このたび SEA では，オープンソースに関する新しい分科会 SIG-OSS を発足させることになりました．E-Japan との関連で Linux がようやく世の中の大きな関心を集めるようになり，オープンソースやフリーソフトウェアについての議論があちこちで行われています．-OSS は，今後 FSIJ (Free Software Initiative Japan)と連携をとりながら，技術者の立場から，この問題にアプローチして行こうと考えています．

今回のフォーラムは SIG-OSS のキックオフ・イベントとして，FSIJ の月例会とジョイントする形でのパネル討論です．最近新聞紙上などで話題になった SCO 問題をはじめとして，オープンソースにまつわるさまざまなことがらを多角的にとりあげて議論したいと考えています．多くの方々の参加をお待ちしています．

なお，SIG-OSS では，みなさんの情報交換のための Mailing List を開設しました．この ML に参加希望の方は：

sea-sigoss-ctl@media.osaka-cu.ac.jp

宛に，本文の最初に

subscribe あなたの名前　(例： subscribe Hoshino Senichi)

と書いて mail をお送りください．確認のメールが来るので，それに正しく返信をすれば自動登録されます．

********** 開催要領 **********

1. 日 時： 2003年 9月 19日 (金) 13:30 ～ 17:00

2. 場 所： 全国情報サービス産業厚生年金基金会館 7 階会議室 (東京都中央区築地4-1-14)
http://www.mapion.co.jp/c/f?el=139/46/17.173&scl=20000&pnf=1&uc=1&grp=all&nl=35/39/50.384&size=500,500
地下鉄 日比谷線 都営浅草線「東銀座」出口A6 または 大江戸線「築地市場」出口A3より徒歩3分

3. プログラム (予定)：

13:00～13:30	受付
13:30～15:00	パネリストによるプレゼンテーション
	鈴木 裕信 (FSIJ)
	中野 秀男 (大阪市立大学)
	林 香 (SRA)
	葉 雲文 (コロラド大学)
15:00～15:30	Break Time
15:30～17:00	質疑およびフリーディスカッション
	司会: 杉田 義明 (SRA)

4. 参加費： SEA および FSIJ 会員 1,000円，一般 2,000円

5. 定 員： 50 名 (先着順にて締切ります)．

6. 申込み方法： 下の申込票に必要事項を御記入の上，SEA 事務局まで E-Mail でお申込みください．なお，参加費は当日会場受付にてお支払いください(領収書を差し上げます)．申込受付後のキャンセルは原則としてお断りします．

申込み宛先： ソフトウェア技術者協会(SEA)　E-Mail: sea@sea.or.jp

.....

SEA & FSIJ 合同 Forum (September 2003) 参加申込

氏名(ふりがな)：
所属：
住所：(〒　　)
Tel:　　Fax:
E-Mail:
種別(該当欄にチェック): ☐ SEA 会員(No.　　)　☐ FSIJ会員(No.　　)☐ 一般

..... mail to; sea@sea.or.jp .....

ソフトウェア技術者協会
〒160-0004　東京都 新宿区 四谷 3 - 12 丸正ビル 5F
Tel: 03 - 3356 - 1077　Fax: 03 - 3356 - 1072
E-mail: sea@sea.or.jp
URL: http://www.iijnet.or.jp/sea