NAIST-IS-DT9861008

博士論文

導入/運用工数の制約を考慮したソフトウェア開発支援

阪井 誠

2000年 12月 26日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である。

阪井 誠

審査委員： 井上 克郎 教授
小山 正樹 教授
関 浩之 教授
松本 健一 助教授

導入/運用工数の制約を考慮したソフトウェア開発支援*

阪井 誠

内容梗概

ソフトウェアを効率よく開発する目的で，開発支援は行われるが，従来の開発支援技術は，導入と運用に多くの工数が必要であった．組織の規模や開発規模が小さい場合，開発支援の導入工数と運用工数の制約は大きく，適用が困難な場合がある．

本論文では，開発支援の導入工数と運用工数に制約を考慮した次の２つの支援方法を提案する．

(1) 小型計算機上での開発プロセスの改善法…ソフトウェア開発で広く用いられている作業報告書の記述から開発上の問題とその具体的な解決法を収集，蓄積し，形式化して開発プロセスモデルに組み込む．作業報告書からの解決法の収集には僅かな運用工数しか必要とならない．また，解決法を蓄積するデータベースのデータ構造は比較的単純で，導入工数も小さい．蓄積された解決法は，開発上の新たな問題に対する解決法の創造にも利用可能で，より現実的で具体的な解決法を発想する手助けとなる．提案方法を実際のソフトウェア開発プロジェクトに適用した結果，小型計算機上のソフトウェア開発でよく発生する６つの問題に対する具体的な解決法が収集でき，それらを組み込んだ新しいプロセスモデルを得ることができた．また，作業報告書に基づく解決法の収集には，わずかな人員と労力しか必要とならないことも確かめられた．

* 奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 博士論文, NAIST-IS-DT9861008, 2000年12月26日.

(2)　再利用が繰り返された組込みソフトウェアの改良法…外部変数によるデータの受け渡しを，引数と同様に上位関数とのインタフェースとして扱うことで，プログラム構造の評価を可能にする．提案方法は，(a) 広く知られている方法論である構造化設計が定める基準により，プログラム構造を評価できる，(b) 利用者の経験度を問わず短時間で習得できるように，設計ガイドとチェックリストが整備されている，ことで導入工数を少なくしている．提案方法に基づく支援ツールを開発現場で試行した結果，コードレビューでは指摘できなかったプログラム構造の問題点を，短時間の評価で指摘することができ，運用工数が少ないことも確認された．

キーワード

ソフトウェアプロセスの改善, ソフトウェアプロセスモデリング, リスクマネージメント, 組込みソフトウェア，メンテナンス，リエンジニアリング

New supporting methods of software development for allowing for limit on introduction and operation costs*

Makoto Sakai

Abstract

In order to improve productivity, we support software development. In a small organization or a small project, however, introduction and operation costs will be more limited. It is hard that we apply traditional supporting methods to such a project because those mehtods require large introduction and operation costs.

This thesis proposes new supporting framework and method of software development for allowing for limit on introduction and operation costs.

(1) A New Framework for Improving Software Development Process on Small Computer System: The key idea of the proposed framework is that the solutions to the software development problems can be extracted from status reports, which are used widely, and can be formalized and stored in a database. The proposed framework needs little operation cost because the solutions are extracted from status reports. The stored solutions can be used for not only process improvement, but also support for devising new solutions when new problems occur in the future. The proposed framework needs little introduction cost because that the solutions database has a relatively simple schema. This thesis also presents an application of the proposed framework to the practical

* Doctor's Thesis, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT9861008, December 26, 2000.

software project, where we can extract six solutions to the development problems and reconstruct the software process which has two solutions to prevent two typical development problems.

(2) A New Environment for Improving Legacy Software on Embedded Systems: The proposed method uses common coupling as an interface with a caller function, and supports the evaluation of a program structure. (a) The proposed method can evaluate a program structure using rules in the well known structured-analysis method. (b) The proposed method has a design guide and a checking-list to master the method in short time. The above characteristics reduce introduction costs. The tool based on the proposed method which we developed was introduced into a real project. The method enabled us to discover a problem of program structure which had been over looked at code review. Also it was verified that the tool needs little operation costs.

Keywords:

software process improvement, software process modeling, risk management, embedded software, legacy software, software maintenance

関連発表論文

1. 阪井 誠, 松本 健一, 鳥居 宏次, “ソフトウェアプロセスの動的な変更部分の分析によるプロジェクト運営実践技術のモデル化とプロセス改善”, 情報処理学会ソフトウェア工学研究報告, Vol.94, No.43, pp.41-48, 1994.
2. 阪井 誠, 松本 健一, 鳥居 宏次, “ダウンサイジング時代のプロセス改善モデル”, ソフトウェアシンポジウム’95論文集, pp.131-140, 1995.
3. Makoto Sakai, Ken-ichi Matsumoto, Koji Torii, “A new framework for improving software development process on small computer systems,” Proceedings of International Symposium on Software Engineering for the Next Generation, pp.151-160, Feb. 5-7 1996.
4. Makoto Sakai, Ken-ichi Matsumoto, Koji Torii, “A new framework for improving software development process on small computer systems,” International Journal of Software Engineering and Knowledge Engineering, Vol.7, No.2, pp.171-184, 1997.
5. 阪井 誠, 久保田 益史, 松本 健一, 鳥居 宏次, “組込みソフトウェア改造時の作業配分を容易にする改造工数の見積もり尺度の提案”, 信学技報, SS98-57, Vol.98, No.675, pp.39-46, 1998.
6. 阪井 誠, 久保田 益史, 沖田 昌也, 松本 健一, 鳥居 宏次, “組込みソフトウェアにおけるレガシーソフトウェアの改良”, ソフトウェア・シンポジウム’99論文集, pp.59-66, 1999.
7. 阪井 誠, 松本 健一, 井上克 郎, “Operation Probe: ユーザー操作の記録環境の提案”, ソフトウェア・シンポジウム2000論文集, pp.206-212, 2000.
8. 阪井 誠, 久保田 益史, 沖田 昌也, 松本 健一, 鳥居 宏次, “レガシーな組込みソフトウェアの改良支援ツール”, 電子情報通信学会論文誌,(掲載予定).

目次

1. はじめに **1**

2. 開発支援における導入工数と運用工数の制約 **5**

2.1 企業における利益 6
2.2 開発支援法を利用する際の制約 8

3. 小型計算機上での開発プロセスの改善法 **10**

3.1 小型計算機上のソフトウェア開発 11
3.1.1 特徴 11
3.1.2 自由度とリスク 12
3.1.3 問題の発生と解決 13
3.2 従来のプロセス改善法 16
3.3 提案するプロセス改善法 19
3.3.1 特徴 19
3.3.2 問題解決のプロセスモデル 20
3.3.3 データ構造 22
3.3.4 プロセス改善の手順 27
3.4 適用実験 31
3.4.1 適用実験 1 31
3.4.2 適用実験 2 35
3.5 まとめ 39

4. レガシーな組込みソフトウェアの改良法 **40**

4.1 関連研究 41
4.1.1 プログラムの評価法 41
4.1.2 レガシーソフトウェアの改良 43
4.2 提案する評価法 45
4.2.1 レガシーソフトウェアの調査 45
4.2.2 レガシーな組込みソフトウェアの評価法と改良法 47

4.3 改良支援ツール 50
4.3.1 ツールの設計と実装 50
4.3.2 ツールの導入 52
4.4 結果と考察 53
4.4.1 試行 53
4.4.2 導入結果 54
4.4.3 オープンソースを用いた内製の効果 55
4.4.4 考察 55
4.5 まとめ 57

5. おわりに **59**

謝辞 **61**

参考文献 **63**

図目次

1 問題発生時の再計画 15
2 天候のリスクと観光旅行のプロセス 21
3 データ構造 23
4 プロセスの記述 26
5 作業報告書の形式の例 28
6 提案する開発モデル 29
7 適用結果 34
8 解決法データベース 38
9 構造図を用いた評価 42
10 基本的なアイデア 47
11 評価と改良法 48
12 ツールのデータフロー 51
13 ツールのユーザ・インタフェース (GUI版) 57

表目次

1 CMMと提案する改善モデルの比較 20
2 作業の属性 24
3 解決法データベースのスキーム 24
4 収集された問題と解決法 32

1. はじめに

ソフトウェアを効率よく開発する目的で，開発支援は行われる．ソフトウェアの生産性を高めることがソフトウェア工学の課題の一つであり，多くの開発支援方法が提案されてきた．特に，大規模ソフトウェアの開発がソフトウェア工学の中心的な研究対象の一つであったことから [9]，組織的な管理や高機能なツールを用いた大規模な開発支援が多く行われてきた．具体的には，大規模ソフトウェアの開発において，プロダクトの品質や開発プロセスを管理する目的で専任の部門や担当者を置き，組織的な開発支援が行われてきた [18]．また，既存のソフトウェアを改良して保守性の高いソフトウェアを開発する場合も，専用の分析ツールやナレッジベース等を用いてプログラム全体の解析を行い，比較的大規模な再構築の支援が行われてきた [3].

しかし，組織の規模や開発規模が小さい場合や，開発競争が厳しい場合，開発支援の導入工数と運用工数への制約は大きくなり，従来の開発支援技術の適用が困難な場合がある．開発支援を長期に実施することで開発効率が向上する場合でも，単年度などの短期間の利益やプロジェクト単位での利益は考慮する必要がある．開発支援は導入や運用の制約を考慮して，より少ない工数でより大きな効果が得られなければならない．小型計算機上のソフトウェアやレガシーな組込みソフトウェアの開発支援も，導入/運用工数の制約が大きい.

低価格な小型計算機の普及は，ソフトウェア開発のリスクを増加させるとともに，開発支援に対する導入/運用工数の制約を大きくした．小型計算機上のソフトウェア開発では，開発環境や実行環境，実現方法，開発の対象となるドメインの新しい選択肢が増加している．実現方法の選択肢の広がりは，経験の少ない環境上での，経験の少ない実現方法を用いた，経験の少ないドメインの，ソフトウェアの開発をもたらした．経験の少ない新しい選択肢を使うことで生じる，ソフトウェア開発のリスクが増加している．また，小型計算機がソフトウェア開発の標準のハードウェアとなる事で，ライブラリやコンポーネントといったソフトウェアの開発効率を高める環境が整備された．このような開発効率の向上は，ソフトウェアプロジェクトの規模を縮小している.

このような小型計算機上のソフトウェア開発では，従来のプロセス改善法の適

用は困難であり，新しい支援法が必要である．ソフトウェア開発におけるリスクを低くする有効な方法の一つとして，Humphrey の CMM (Capability Maturity Model) をはじめとするソフトウェアプロセスの改善が考えられる [6][18][24]．しかし，CMM のような従来のプロセス改善法は，アプリケーションの開発対象となる経験豊富なドメインのソフトウェアを，経験豊富な環境上で，経験豊富な方法で開発することを前提としており，プロジェクト管理者がソフトウェア開発中に生じる新しい問題の解決法を創造することを支援していない．また,SEPG(Software Engineering Process Group) の設立が必要とされるなど，開発を支援するために多くの導入/運用工数を必要とする．このため，CMM は小型計算機上のソフトウェア開発には適用が困難であり，開発支援のための新しいプロセス改善法が必要である．

一方，改良が繰り返される組込みソフトウェアにおいても，導入/運用工数の制約は大きい．組込みソフトウェアは，高い信頼性が求められるほか，タイムリーに製品を市場に投入することが求められるので開発期間短縮の圧力が強く [35] [36]，マーケティングの観点から予算の制約も厳しいという特徴がある．そこで，新しい機能を付加したソフトウェアを，より信頼性を高く，より少ない工数で，より短期間に開発するために，既存のソフトウェアを改造して新規のソフトウェアを開発する場合が多い [28]．このような改造が繰り返され，保守性が徐々に悪くなったソフトウェアは，遺産的 (Legacy) ソフトウェアと呼ばれている．レガシーソフトウェアは改良が必要であるが，本来，少ない工数で短期間に開発する目的で遺産的になったことから，改良の支援法に対する導入/運用工数の制約は大きい．

また，組込みソフトウェアには外部変数によるデータの受け渡しが用いられている場合が多く，さらに保守性が悪くなっている．外部変数の参照を考慮した，新しい開発支援法が必要である．ソフトウェアが組み込まれる機器は，多くの動作モードや障害時の復旧対策が必要なので，通常の主記憶領域とはメモリ空間が異なる不揮発メモリが用いられる．これらは，実装の容易さや処理速度の観点から，外部変数として実装し，リンク時に不揮発メモリに割り当てられている場合が多い．また，OS とハードウェアの制約から，タスク間の高速なインタフェースとして外部変数が用いられる場合もある．このような外部変数は，機能が追加

されると動作モードや障害処理が増えるので，さらに増加する．外部変数を用いたデータの受け渡しは，旧来から悪いインタフェースであるといわれている[27]．しかし，やむを得ず外部変数を用いる場合に，プログラム構造のどのような点に注意すべきかが示されていなかったことが，保守性を悪くする一因になっている．このような，外部変数を用いたレガシーな組込みソフトウェアの開発支援には，導入/運用工数の制約を考慮した新しい改良法が必要である．

以下，2章ではソフトウェア開発の経済性に基づいた，開発支援における導入工数と運用工数の制約に関して述べる．長期的な視点では有効な開発支援法であっても，短期的な利益を考慮すると，開発支援の対象となる組織やプロジェクトの規模が小さい場合や，競争の厳しい市場のソフトウェア開発では，導入工数と運用工数の制約が大きくなる事を示す．

3章では，小型計算機上のソフトウェア開発におけるリスクの低減を目的とした，ソフトウェア開発プロセスの改善法を提案する[41][44][45]．提案する方法では，一般のソフトウェア開発で広く用いられている作業報告書の記述から開発上の問題とその具体的な解決法を収集し，形式化してデータベースに蓄積する．形式化された解決法をプロセスモデルに組み込むことにより，リスクに対して堅牢なプロセスモデルを得ることが出来る．蓄積された解決法は，開発上の新たな問題に対する解決法の創造に利用可能で，より現実的で具体的な解決法を発想する手助けとなる．提案する方法を実際のソフトウェア開発プロジェクトに適用した結果，小型計算機上のソフトウェア開発でよく発生する6つの問題に対する具体的な解決法が収集でき，それらを組み込んだ新しいプロセスモデルを得ることができた．また，提案する方法に基づくデータベースの構築と，作業報告書に基づく解決法の収集と蓄積の試行の結果，改善法の導入と運用にはわずかな人員と労力しか必要とならないことが確かめられた．

4章では，まず，レガシーな組込みソフトウェアを調査して保守性の悪い部分を分析し，評価法を提案する．次に，提案方法に基いて開発したソフトウェア改良支援ツールに関して述べる[42][43]．提案する評価法では，外部変数によるデータの受け渡しを引数と同じように扱うことができる．提案方法は，(a)広く知られている方法論である構造化設計が定める基準により[39]，プログラム構造を評

価できる，(b) 利用者の経験度を問わず短時間で習得できるように，設計ガイドとチェックリストが整備されている，ことで導入工数を少なくしている．提案方法に基づく支援ツールを開発現場で試行した結果，コードレビューでは指摘できなかったプログラム構造の問題点を，短時間の評価で指摘することができ，運用工数が少ないことも確認された．

最後に5章では，まとめと今後の課題について述べる．

2.　開発支援における導入工数と運用工数の制約

本章では，ソフトウェア開発の経済性に基づいた，開発支援における導入工数と運用工数の制約に関して述べる．長期的な視点では有効な開発支援法であっても，短期的な利益を考慮すると，開発支援の対象となる組織やプロジェクトの規模が小さい場合や，競争の厳しい市場のソフトウェア開発では，導入工数と運用工数の制約が大きくなる事を示す．まず，2.1 節では売上高と各種の費用により決定される，企業における利益について述べる．次に，2.2 節で開発法を利用する際に必要な費用と開発支援法を利用する際の制約について述べる．

2.1 企業における利益

企業の目的は利益を追求することにある．決算時に作成される損益計算書などの数字としては，利益は示す対象毎に売上総利益，営業利益，経常利益，税引き前当期利益，当期利益，当期未処分利益の6種類の指標で表されている[16].

式(1)の売上総利益は，売上高から原価(業種により売上原価あるいは製造原価と呼ばれる)を引いたもので，一般に粗利益と呼ばれている．

$$\text{売上総利益} = \text{売上高} - \text{原価} \tag{1}$$

原価は，ソフトウェア開発においては，プロジェクトごとの作業工数に作業単価を乗じて算出した費用などプロジェクトとの対応が可能な直接費と，開発部門共通で用いる開発支援ツールや電気代などのプロジェクトとの対応が不可能な間接費に分類される(ただし，開発支援ツールをプロジェクト単独で用いる場合は直接費である)[23].

次に営業利益は，式(1)で示される売上総利益から「販売及び一般管理費」を引いたものである．ここでの販売及び一般管理費には，研究開発費や設備投資が含まれる．

$$\text{営業利益} = \text{売上総利益} - \text{販売及び一般管理費} \tag{2}$$

ソフトウェア開発における専任の支援部隊の費用は開発に直接関わらないことから，全社的な組織である場合は販売及び一般管理費，開発部門内の組織である場合は間接費に含まれる．

この他には，営業利益に経営上の投資の配当や損失，預金の金利や借入金の金利を考慮した経常利益，経常利益に固定資産や投資有価証券の売却益および売却損などを考慮した税引き前当期利益，税引き前当期利益から税金を差し引いた当期利益(純利益とも呼ぶ)，当期利益に前期繰越利益を加えた当期未処分利益がある．

これらの利益を示す指標のうち，売上から原価と販売および一般管理費を引いた営業利益が，会社本来の業務による利益である．売上高は主に営業活動により決定されるので，開発部門と研究開発部門で営業利益を増加させるには，直接費，間接費販売及び一般管理費を低減させる必要がある．新たに開発支援法を導入す

る場合にも，開発支援法による効果だけでなく，開発支援法利用に必要な費用を考慮する必要がある．具体的には，生産性の向上による直接費の低減，支援方法の導入や実施による直接費の増加，支援ツールの取得による間接費 (あるいは直接費) の増加，専任の支援部隊による販売及び一般管理費 (あるいは間接費) の増加を考慮する必要がある．

2.2 開発支援法を利用する際の制約

ソフトウェア開発における新しい開発支援法の導入は，経済活動の一環として行われるので，開発支援法を利用することで利益が増加する必要がある．具体的には，以下の式で表されるような利益の増加が予想できなければならない．

$$\text{利益の増加} = \text{開発支援法による効果} - \text{開発支援法利用の費用} \tag{3}$$

式 (3) において，開発支援法による効果は生産性の向上により削減可能な作業工数に作業単価を乗じたものであり，開発支援法の費用は，以下の 3 項目の和である．

取得費用: ツールの開発あるいは購入に要する費用．

導入費用: ツールのマシンへのインストールや教育などに要する工数に作業単価を乗じたもの金額．

運用費用: 開発支援法利用に伴う作業やツールの保守作業などに要する工数に作業単価を乗じたもの金額．

式 (3) から，新しい開発支援法を利用するには，少なくとも，以下の制約を満たさなければならない．

$$\text{開発支援法による効果} - \text{取得費用} - \text{導入費用} - \text{運用費用} > 0 \tag{4}$$

式 (4) から，開発支援法による効果を大きくするか，他の項の値を低減することで，開発支援法の利用による利益を増加し，制約を満たすことが可能になる．

式 (3) および式 (4) は長期的な利益を示したものであるが，企業においては短期的な利益，すなわち，単年度 (あるいは半期，四半期) でも黒字が求められる．近年は，環境の変化が激しいことから，短期的な利益がより重視される傾向にある．具体的には，文献 [5] に示されるように，集中した支出を避けることで，ソフトウェア開発の経済的な価値を高める必要がある．短期的な利益を考慮した上で，新しい開発支援法を利用するには，少なくとも，以下の制約を満たさなければならない．

$$\text{期間内営業利益} + \text{期間内効果} - \text{取得費用} - \text{導入費用} - \text{期間内運用費用} > 0 \tag{5}$$

ここで，開発支援は期間内の原価を低減する方法であることから，期間内効果は期間内の原価よりも少ない．営業利益と原価の和は売上高より小さいので，少なくとも以下の制約を満たさなければならない．

$$売上高 - 取得費用 - 導入費用 - 期間内運用費用 > 0 \tag{6}$$

大学等で開発された無料の支援ツールを用いて取得費用を無くすことは可能であるが，支援の対象となる開発による売上高の少ない小規模な組織や，より多くの利益が求められる競争の厳しい市場のソフトウェア開発では，導入工数や運用工数に対する制約が厳しいことがわかる．

さらに，各プロジェクト毎の採算も考慮する必要がある．開発支援法が特定のプロジェクトのみに有効であるなど，直接費で全ての費用をまかなう場合は，式(6)をプロジェクト内で満たす必要があり，制約はさらに厳しくなる．また，より一般的な開発支援法であり，開発部門共通で開発支援法を用いるなど，取得費用と導入費用を間接費や販売及び一般管理費から捻出した場合は，プロジェクトで開発支援法を実施するための制約は以下の式になる．

$$開発支援法による効果 - プロジェクト内運用費用 > 0 \tag{7}$$

式(7)において，プロジェクト内運用費用は，販売及び一般管理費(あるいは間接費)による開発支援の専任部隊の費用を，運用費用から引いたものである．開発支援法による効果はプロジェクトの総工数を超えることはできないので，プロジェクトの規模が小さい場合や，競争の厳しい市場のソフトウェア開発では，プロジェクト内の運用工数に対する制約は厳しくなると言える．

このように，短期的な利益を考慮すると，支援の対象となる開発による売上高の少ない小規模な組織や，より多くの利益が求められる競争の厳しい市場のソフトウェア開発では，ソフトウェア開発支援の導入工数や運用工数に対する制約は厳しくなる．また，プロジェクトの採算を考慮すると，プロジェクトの規模が小さい場合や，競争の厳しい市場のソフトウェア開発では，プロジェクト内の運用工数に対する制約はさらに厳しくなる．

3. 小型計算機上での開発プロセスの改善法

本章では, 小型計算機上のソフトウェア開発におけるリスクの低減を目的とした, ソフトウェア開発プロセスの改善法を提案する. 提案する方法では, 一般のソフトウェア開発で広く用いられている作業報告書の記述から開発上の問題とその具体的な解決法を収集し, 形式化してデータベースに蓄積する. 形式化された解決法をプロセスモデルに組み込むことにより, リスクに対して堅牢なプロセスモデルを得ることが出来る. 蓄積された解決法は, 開発上の新たな問題に対する解決法の創造に利用可能で, より現実的で具体的な解決法を発想する手助けとなる. 提案する方法を実際のソフトウェア開発プロジェクトに適用した結果, 小型計算機上のソフトウェア開発でよく発生する 6 つの問題に対する具体的な解決法が収集でき, それらを組み込んだ新しいプロセスモデルを得ることができた. また, 提案する方法に基づくデータベースの構築と, 作業報告書に基づく解決法の収集と蓄積の試行の結果, 改善法の導入と運用にはわずかな人員と労力しか必要とならないことが確かめられた.

以下, 3.1 節で小型計算機上のソフトウェア開発について述べ, 3.2 節で小型計算機上でのソフトウェア開発において CMM を適用する上での困難な点を述べる. 3.3 節では提案するプロセス改善法の特徴, データ構造, プロセス改善手順を述べる. 3.4 節では評価のために行った実際のプロジェクトデータへの適用結果を示す. 最後に 3.5 節で本論文の結論を述べる.

3.1 小型計算機上のソフトウェア開発

3.1.1 特徴

PC ユーザからの GUI(Graphical User Interface) 向上とレスポンス向上への要求，ビジネスの現場や経営部門からのタイムリーな情報提供への要求と業務拡張への柔軟な対応力への要求，そして PC/WS の価格性能比の向上といった社会的背景によって，情報システムの見直しを中心とした小型計算機を用いたシステム構築（ダウンサイジング）が広く行われている [1].

小型計算機を用いたシステム構築の特徴は，オープンシステムの考えを取り入れて構築されることと，分散処理による負荷分散である．オープンシステムとは，固有のベンダから独立しており，広く一般に利用されている標準規格に合致するように設計され，具体化された製品や技術で，可能な限りの接続性や互換性を確保しつつ構築されたシステムのことである [33]. 様々な標準規格に合致したハードウェア・ソフトウェアを組み合わせて小型計算機を用いたシステムは構築されている．オープンシステムによる分散処理環境では情報処理の負荷の増大（集中）に対し，分散によって負荷を拡散する方向で吸収しようとする．この発想はサーバ・クライアント・モデルの原点であるともいえるが [33]，エンドユーザ自身が計算機を自由に操って自分の欲しい答えをはじき出す，ツール指向のスタイルであるともいえる [34].

小型計算機を用いたシステムは，エンドユーザの要求に対応するために構築され，そのリスク（危険性）はソフトウェアハウスが受け持っている．オープンシステムは，情報を効率的に利用することを優先する（従来の）システムに対し，「情報を活用する」，とか「使いやすい」というところにメリット（美点）がある [33]. 小型計算機を用いたオープンシステムでは，GUI 構築ツールやネットワークアクセスツール，データベース・マネージメント・システム (DBMS) といったミドルウェアを用いてエンドユーザに使いやすくて安価なシステムが構築される．また，システムの設計に対しては，情報システム部門ではなくエンドユーザ自身が主導権／決定権を持つことが多い [1]. しかし，エンドユーザは技術が不足しているため，ハードウェアの構成を含めたシステムの設計を行うことはリスク（危険性）が高いためできない．そこで，ユーザの要求するシステムを，ソフトウェ

アハウスがハードウェアを含めて提供するシステムインテグレーションが広く行われ，ユーザのリスクをソフトウェアハウスが受け持つようになっている．

3.1.2 自由度とリスク

小型計算機上のソフトウェア開発は自由度を高めたが，それとともにリスクも高めた．それらは，環境，実現方法，ドメインの自由度と，各自由度に対応するリスクである．これらの自由度やリスクは従来から存在したが，小型計算機上でソフトウェア開発が行われることで顕著になった．

小型計算機上のソフトウェア開発では，ハードウェアの構成，オペレーティングシステム，DBMSやネットワークツールといったミドルウェアは，数多くの種類から複数のものが選択され構成されようになった．これを環境の自由度と呼ぶ．従来の汎用機のように1社のみからほとんどすべての環境が提供される場合は，この自由度が低く，システム構成は要求に応じて一意に決定できた．一方，小型計算機上のソフトウェア開発では，そのインタフェースが公開されたオープンシステムを用いることが多い．このため，ミドルウェアは多いだけでなく新製品が次々と開発され，同一の要求に対するシステム構成の選択肢も多く自由度が高い．この環境の自由度の高さはリスクも高くした．実行環境の構成を決定するためには考慮すべき項目が数多く存在し，最適な環境の構成を決定するために非常に多くの時間を要するが，システム構成の決定後であっても，ある特定の組み合わせで問題が生じた場合には構成が変更されることがある．また，新しい実行環境が構成されるとソフトウェア開発の環境を整備しなおす必要の生じる場合がある．従来はシステム構成の選択肢があまりないことから問題を運用で回避することが多かったため，環境のリスクはあまり高くなかった．

また，GUIや分散処理が可能になり，ユーザの要求を実現する方法の自由度も増した．GUIでは，従来のキャラクタベースのユーザ・インタフェースでは不可能だったマウスによるポインティング，ポップアップメニュー，操作手順等のGUIを用い，システムをより業務に適したものに作り上げることができるようになった．また，複数の計算機を用いた負荷分散や，サーバ・クライアント方式のような機能分散も，ユーザの要求に適した方法で行われるようになった．この実現方

法の自由度は要求を不安定にした．実現方法が多岐にわたるため，ユーザは実現したいアプリケーションの詳細なイメージを持つことが非常に難しくなっている．そのためユーザの要求は曖昧で不確かなものになっている．この不確かな要求を元に開発を行うと，ユーザの要求とかけ離れたものを開発してしまう可能性がある．これを実現方法のリスクと呼ぶ．

このような環境や実現方法に加えて，計算機が小さく安価であるので，これまで計算機が導入されなかった比較的規模の小さい業種や業務を開発対象とすることが可能になった．これをドメインの自由度と呼ぶ．小型計算機上のソフトウェア開発はこれまで開発者の知らなかった業種や業務が扱えるようになった．また，計算機システムの詳細を知らないユーザにも，計算機が使えるようになった．このドメインの自由度は知識の不足を生じさせた．開発者の経験の少ないドメインでは，知識の不足していることが多く，開発者が過去の経験で類推することによる誤解や仕様の抜けが生じやすい．また，開発者とエンドユーザの間では，お互いの知識や経験に大きな偏りがあるため，協調作業を行う上でのトラブルも生じやすい．

このように，小型計算機上のソフトウェア開発における自由度の高さは，開発上のリスクを高めた．また，技術の進歩により生じる問題も常に変化している．しかし，ダウンサイジングのプロジェクトは規模が小さいことが多いため，プロジェクト管理にかけることのできるコストも小さい．そこで，これらのリスクにより生じた問題は，プロジェクト管理者の経験により解決が図られている．

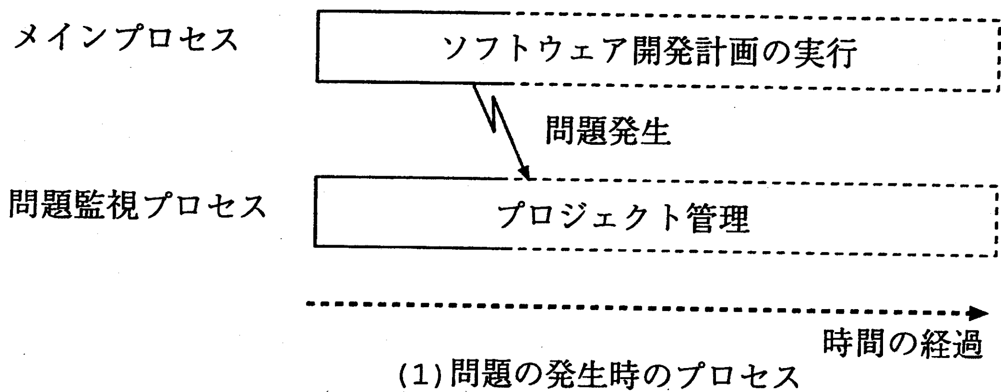
3.1.3 問題の発生と解決

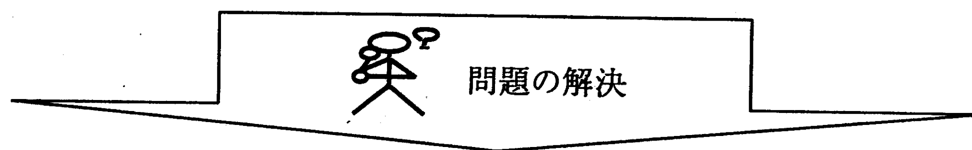
ソフトウェア開発中の問題を解決する場合のソフトウェア開発プロセスを図 1 に示す [46]．ソフトウェア開発のプロセスを問題の発生とその解決の観点でモデル化すると，ソフトウェア開発計画に従い開発を行う“メインプロセス”と，プロジェクト管理を行いながら問題の発生を監視する“問題監視プロセス”に分けられる．それぞれのプロセスは複数の作業の集まりであり，それらをまとめて四角で表している．また，点線の矢印は時間の経過を表している．

図 1(1) は，メインプロセスの作業が四角の実線部分まで実行された時に問題

が発生し，問題監視プロセスに検知されたことを示している．ソフトウェア開発はなるべく問題が生じないように計画され，計画に従い実行される．しかし，小型計算機上のソフトウェア開発のように経験のない問題が生じやすく問題の種類の多い場合は，全ての問題を避けることはできない．開発中の問題の発生はプロジェクト管理者が行うプロジェクト管理の作業として監視され，問題の発生が検知される．

問題を検知したプロジェクト管理者は，問題を解決するために再計画する．図1(2) は，メインプロセスの作業が途中で再計画され，途中から変更された計画が実行されたことを示している．問題を解決するためにプロジェクト管理者は，過去の経験を生かして解決法を考え，再計画する．プロジェクト管理者が同じ問題を経験しているのであればその時の解決法に基づいた再計画を行って実行する．また，プロジェクト管理者の経験してない問題であれば，過去に経験した類似する問題の解決法を参考にして新たな解決法を創造し，それに基づいた再計画を行い実行する．以上に述べたのように，ソフトウェア開発中に生じる問題は再計画により解決されている．





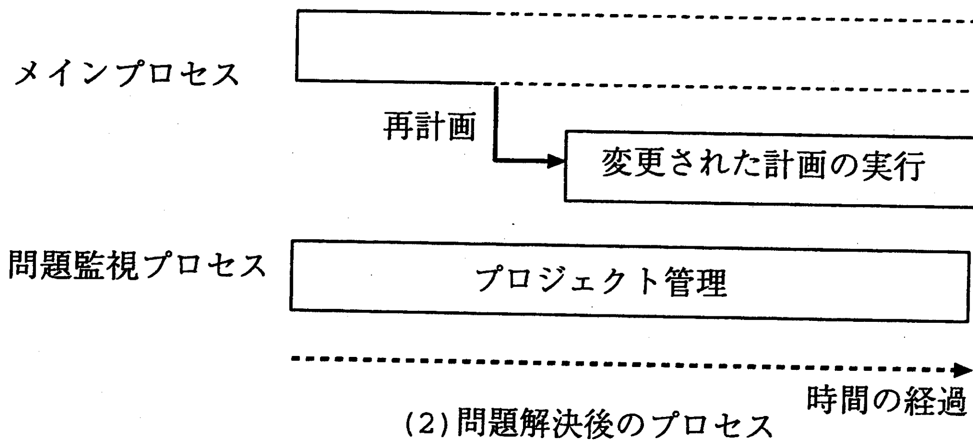


図 1 問題発生時の再計画

3.2 従来のプロセス改善法

従来のプロセス改善法には，Humphrey の CMM (Capability Maturity Model) がよく知られている [18]．CMM は高品質で一貫性があり予定どおりに開発できることを成熟度と呼び，開発組織を分類している [17]．ソフトウェア開発の基本作業は多くのソフトウェアプロジェクトで共通であり，ソフトウェアプロセスの改善法には，控え目なプロジェクト毎のカスタマイズ以外は必要でないと考えられている [19]．Humphrey は，ソフトウェアプロセスを適切に実行されれば要求される結果の得られる作業の集合と定義し，“ソフトウェアプロセスモデル” と呼んでいる [18]．

ソフトウェアプロセスモデルは多くのプロジェクトのプロセスの共通部分から作られ，ソフトウェア専門家に秩序だったやり方で選択可能な開発のガイドを示すものである．ソフトウェアプロセスモデルの定義があれば，開発担当者らは何をすべきか，協同開発者に何を期待すべきか，そして逆に何を提供することが期待されているかがさらに理解されるようになる．プロセスを改善する場合は，ソフトウェアプロセスモデルに対し少しずつ行われる．Humphrey はこのようなプロセス改善の方法をフットボールに例えてプロセスは少しずつ改善し続けなければいけないと述べている [19]．フットボールはシーズンオフ中にフォーメーションの考案と訓練が行われ，シーズン中はフォーメーションに若干の改良が行われるだけで新たなフォーメーションが作られることはない．すなわち，Humphrey はソフトウェアプロセスモデルにより一貫した作業の枠組を提供し，担当者が自分の仕事に専念できるようになると考え，プロジェクト毎にはプロセスの若干の調整しか許されていない．このため，ソフトウェアプロセスモデルの定義には，個別の要求に対応できる柔軟性と組織が要求する標準や一貫性の間にトレードオフが存在する [18]．

CMM の考え方に対し，電子ニュースのニュースグループである comp.software-eng において多くの議論がなされ，文献 [25] にも報告されている．批判の多くは民間市場相手のソフトウェアビジネス関係者 (例えばPC向けの各種ツールや環境プロダクトの開発者) 達から出されたものである．彼らの主張では，ソフトウェア開発に重要なことは，個人の能力の発展と，仕事の途中で新たなプロセスの発明や

リスクの認識や回避など柔軟な決定ができること，さらに付け加えると，ビジネスに精通していることである．CMM はこれらのうち，ビジネスに精通することを少し扱っているだけであると述べられている．このような反論に対し，CMM の考え方を養護する意見もある．また,柔軟な開発はすべての問題が新しくユニークと考え，同一の（あるいは類似の）問題を，別々の人間の手で何度も解決しなおすことで多くの創造的エネルギーが浪費されるという中立的意見も報告されている．

また，Weinberg は CMM の元となったクロスビーの分類を自らの分類に適用しながらも，信頼性に対する完全性の追求を批判している [49]．Weinberg はソフトウェア開発は製造作業のように要求が確かなものでなく，信頼性の問題よりも支配的となるような要求仕様の欠点のあること，すなわち信頼性が高くてもユーザの要求が満たされないことや，信頼性が低くてもユーザの要求が満たされることのあることを指摘し，“不当な完全性の要求は成熟というよりもむしろ幼児性の表れである” と述べている．

さらに，第 16 回 ICSE では，小規模な部門や小規模な会社への CMM の適用に関する調査が報告されている [8]．CMM では，ソフトウェアプロセスの維持と変更，通常業務の支援を行う目的で SEPG(Software Engineering Process Group) の設立が必要とされている．100 人未満の組織においても少なくとも一人の専任 SEPG 担当者と支援者が基本的に必要である．しかし,小規模な部門や小規模な会社ではプロジェクトも小規模であるため，2 で述べたように導入/運用工数の制約が大きい．このため,文献 [8] では，CMM の考え方の有効性を認識しながらも，CMM は導入/運用工数といったコストがかかるという理由で実施されない場合も多い．このように CMM の考え方は評価されながらも，適用分野によっては問題があると考えられる．

一方，前述の小型計算機上のソフトウェア開発における環境，実現方法，ドメインのリスクは，文献 [18] において要求の不安定性，とされている不安定な要求，誤解された要求，未知の要求と考えることができる．Humphrey はそれぞれの要求の不安定性に対し，不安定性の予想と分離，プロトタイプ，早期のプロトタイプあるいは増加的な開発を挙げ，機能とインターフェースの定義のモジュール化

設計とその変更制御の統制を挙げている．しかし，小型計算機上のソフトウェア開発のメリットはその自由度から得られる適応力である．開発プロセスを固定化し，新たな技術や要求を少しづつ受け入れることで安定した開発を行うことは，小型計算機上でソフトウェアを開発するメリットとのトレードオフになると考えられる．さらに，分離すべき不安定な部分やどの部分から増加的な開発を行うかによって，要求の不安定性は変化すると考えられるが，文献[18]では具体的な決定方法に関しては触れられていない．

以上のことから，小型計算機上のソフトウェア開発のプロセスを，CMMにより改善する上での困難な点を挙げると以下のようになる．

1. リスクにより生じる問題の解決は組織で検討する必要がある．小型計算機上のソフトウェア開発では，環境，実現方法，ドメインのリスクがある．これらのリスクに対しCMMでは不安定な部分の分離や増加的な開発を行うと述べているが，分離すべき不安定な部分や増加的な開発を開始する部分の具体的な判断の方法を示していない．そのため，問題の解決は開発者やプロセス改善グループに任されている．
2. 新たな問題への対応が柔軟でない．小型計算機の技術進歩はめざましいものがある．新たな環境や経験の少ないドメインで生じる未知の問題に対応するには，過去の経験を生かした柔軟な対応が行われる必要があるが，CMMではそれを支援していない．
3. プロセスの改善にコストがかかりすぎる．
   小型計算機上のソフトウェア開発ではプロジェクトの規模が比較的小さい．このため，導入/運用工数の大きいプロセス改善法は実施できない．小規模プロジェクトにおいても実施が容易な改善法が必要である．

このように小型計算機上のソフトウェア開発では，CMMの直接的な適用が難しいと考えられる．これは，CMMが巨大で高信頼性の要求されるソフトウェアのために作られたプロセス改善法であることに由来している．小型計算機上のソフトウェア開発に適したプロセス改善法が求められる．

3.3 提案するプロセス改善法

3.3.1 特徴

経験のない問題が多く発生し，リスクが高く，プロセス改善にコストを掛けることのできない小型計算機上のソフトウェア開発のプロセス改善法を提案する．

表1にCMMと提案するプロセス改善モデルの比較を示す．CMMでは，決められた通りに作業が行われることが重視され，ソフトウェアプロセスモデルを厳守しソフトウェア開発を行う．このため，ソフトウェア開発中の再計画の支援は，CMMでは考慮されていない．また，プロセス改善はSEPG (Software Engeneering Process Group) により行われ，開発中に生じた問題の解決法の収集と蓄積は定められておらず，SEPGが収集対象とするかどうかの判断による．また，多くの品質データが収集されるのでコストのかかるモデルである．

これに対し，提案するプロセス改善法は，以下の特徴がある．

1. ソフトウェア開発における問題を具体的に解決することを重視している．実際のプロジェクトで生じた問題が実際に解決された解決法を収集し，蓄積する．
2. ソフトウェアプロセスモデルを強制せず，計画の参考に用いてソフトウェア開発を行う．また，開発中の再計画を支援し，経験のない問題への適応を図っている．具体的には，CBR (Case Based Reasoning) の一つである "Case-based decision aiding" で提案されている方法により，新しい解決法を創造させる [26]．すなわち，人間の経験の蓄積と表示を計算機により支援し，問題を解決するために過去の（類似する）経験を用いる．
3. 開発中に生じた問題を解決した解決法は，作業報告書から管理者を中心とする開発者自身により抽出され，蓄えられ，再計画を行う際に用いられるためコストがあまりかからない．さらに，データベースに解決法が蓄積されるので，同一の解決法を何度も創造するために時間が費やされることを防いでいる．

以下に，提案する改善法の基本的な考え方を観光旅行の例で示し，次にデータ構造と具体的な手順を述べる．

表1 CMMと提案する改善モデルの比較

項目	CMM	提案モデル
ソフトウェアプロセスモデル	厳守する	参考にする
開発中の再計画の支援	考慮されていない	支援する
解決法の蓄積	定められていない	蓄積する
収集する対象	主に品質データ	作業報告書上の解決法
プロセスの改善担当	SEPG	管理者を中心とする開発者
プロセスの改善コスト	大きい	小さい

3.3.2 問題解決のプロセスモデル

3.1節に述べた自由度が高く，リスクの高い小型計算機上のソフトウェア開発のプロセスは，フリープランの観光旅行(以下フリーツアー)に例えることができる．パックの観光旅行は自由度が低く，観光者を十分満足させることができない．それに対し，フリーツアーは自由度が高く，観光者の満足を得ることができるが，反面，観光者のリスクは高くなる．

フリーツアーには様々なリスクがある．強盗や天災(地震，雷，嵐)，迷子，宗教上の問題など様々なトラブルが待ち受けている．そして，それぞれに10ドル紙幣やピストル，ヘルメットやレインコートと長靴，地図，教典などの解決法がある．経験のない旅先で生じうるすべての問題を予想しておくことはできないし，すべての解決法を持ち歩くのは不便であり，現実的でない．

旅先でのリスクを天候に関するものに限った例が図2である．図中の人がフリーツアーの観光者（以下観光者）であり，左が旅行開始時の初期状態であり，順に右の状態に変化し，右端の最終状態になった．観光者は，初期状態の小雨の時は本を解決法とし，次の状態では雨が強くなったので傘を購入し，さらに次の状態では水たまりができたので長靴を用意した．最終状態では風が強くなったのでレインコートを着込んだ．旅行者は必要に応じて解決法を用意したので，天候による被害をあまり受けずに観光することができた．この例で考えられるのは，このように様々な問題が生じうる場合，そのプロセスの順序をそのまま真似ることはあまり意味が無いことである．突然，大雨が降ってきたのであれば図2の順序に

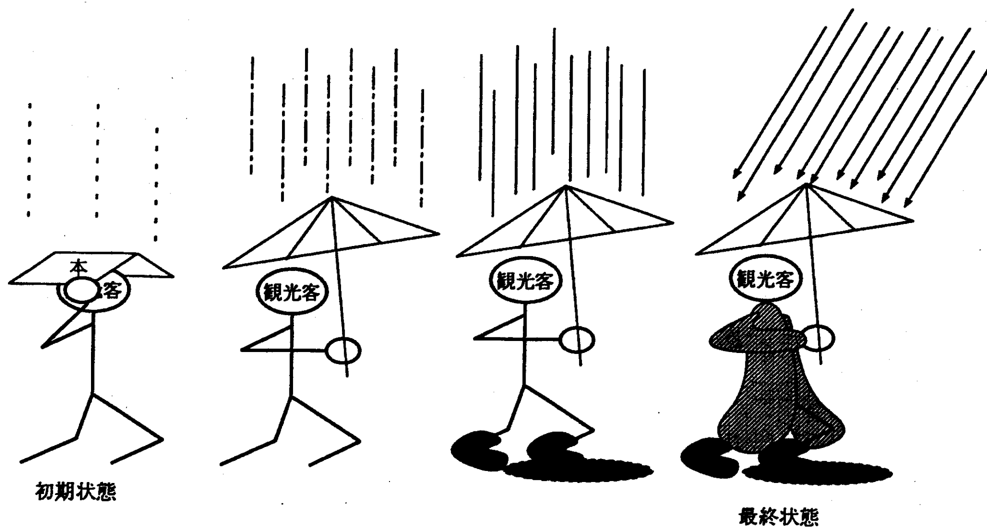


図 2 天候のリスクと観光旅行のプロセス

問題を解決するよりも，図 2 の例で用いられた解決法の一覧を参考にし，状況に応じた解決法 (例えば傘と長靴を用意する) を考えたほうが合理的である．小型計算機上のソフトウェア開発のようにリスクの多い場合も同様のことがいえる．

計画の変更を許した開発を行う場合，リスクにより生じる問題の被害を少なくするには従来の方法だけでなく，解決法の蓄積が不可欠である．フリーツアーの例において，リスクを減少させるスパイラルモデルのプロセスは，一定時間毎に天気予報を聞くことに相当する．プロトタイプは，取り敢えず少しだけ歩いてみることに相当する．ユーザの参加により実務知識を補うことのできる Participatory Design や Joint Application Design は [10]，その地域の知識 (雨季やスコールの情報) を現地の人に聞くことである．これらは，リスクにより生じる問題を早期に知るために有効な手段であるが，発生した問題を解決する具体的な解決法ではない．

このようなリスクにより生じる問題の解決法が収集できれば，更に分類も可能である．問題にはその原因となったリスクのタイプがあり，フリーツアーであれば，犯罪，天候などであり，ソフトウェア開発では，実行環境の変更や開発環境の不足といった環境のリスク，要求の不安定性から生じる要求の実現方法のリス

ク，開発対象のドメイン知識や計算機の知識の不足といったドメインのリスクに分類できる．

また，雨季やスコールのように，その問題の発生する時期が特定されるものもある．ソフトウェア開発においても時期が特定される問題も多い．時期が特定でき，頻繁に生じる問題であれば，基本的な手順(ソフトウェアプロセスモデル)にその解決法を組み込んでおくことができる．南の国ではスコールは夕方に頻繁にくる．このような場合には，夕方に行われる観光(プロセス)には，必ず傘を持って出かけるべきである．ソフトウェア開発においても，ソフトウェアプロセスモデルに組み込んでおくべき解決法は多い [46].

時期が特定できない，あるいは頻繁に生じない問題は，その解決法を検索可能な状態で蓄積されていれば，過去の経験はそれまで経験していない問題が生じた場合に役立つ．雨が降ってきた時に手元に新聞しかない場合，過去に本を使って小雨に濡れずにすんだ経験を思い出せれば，本の代わりに新聞を使うことを考えつく可能性は高い．過去の類似する経験を示すことができれば，新しく生じた問題に柔軟に対応できると考えられる．

3.3.3　データ構造

提案するプロセス改善方法で用いられるデータ構造を図3に示す．図3は，開発組織，プロセスモデル，解決法データベースから構成されている．

開発組織は，プロジェクトマネージャがソフトウェアプロセスモデルに基づき開発計画を立てる実際の開発組織あるいはプロジェクトを示す．開発上の問題が生じ，その解決法が計画に組み込まれていないとき，プロジェクトマネージャは解決法データベースを参照し，解決法を探す，あるいは，新たな解決法を創造する．

ソフトウェアプロセスモデルは適切に実行されれば要求される結果の得られる作業の集合である．ソフトウェアプロセスモデルに基づきソフトウェア開発は行われるが，ソフトウェアプロセスモデルは開発上のリスクを少なくするために開発組織により変更される．ソフトウェアプロセスモデルはメインプロセスと問題監視プロセスから構成される．メインプロセスはソフトウェア開発の作業の集合として定義され，いくつかの解決法が組み込まれている．メインプロセスの各作

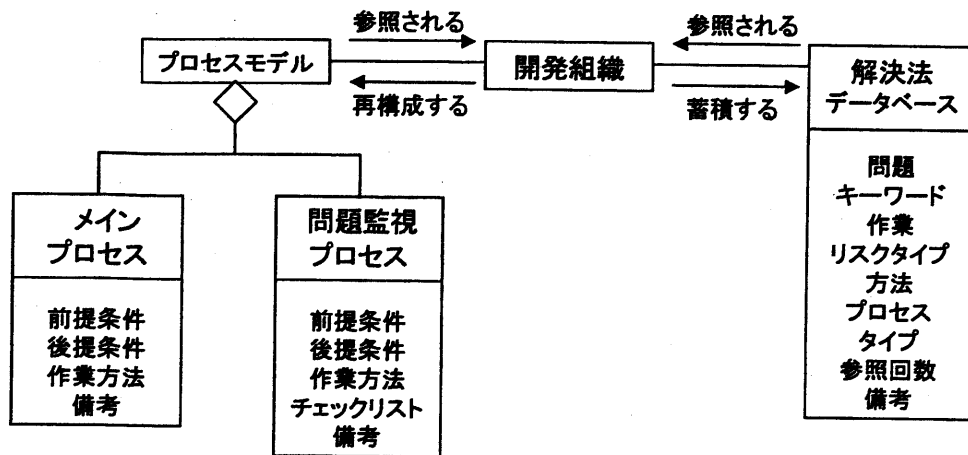


図 3 データ構造

業は以下の属性により定義される（表2）.

前提条件: 作業開始時に必要となるプロダクト.

後提条件: 作成されるプロダクトおよび品質要件.

作業方法: 作業の詳細.

備考: 特記事項.

問題監視プロセスはメインプロセスを監視し問題を検知するプロセスであり，メインプロセスの各属性のほか，解決法データベースの“問題”属性を一覧にしたチェックリストという属性を持つ.

プロセスモデルの記述には，文献[19]で用いられているUPM (Unconstrained Process Model) にアクションダイアグラム [29] の表記法と作業の定義を拡張したものが用いられる (図4). UPM は，プロセスの作業を縦軸に時間の経過を横軸にとり，作業の推移を横線で表現する. UPM では，実際のプロセスをそのまま記述でき，作業の報告に用いられている線表（スケジュール表）とほぼ同じであるためこれを基本とした. UPM のみでは，プロセス改善の単位である一連の作業と，問題の発生した時のみに行う作業を示せない. そこで，アクションダイ

表 2 作業の属性

属性	内容
前提条件	作業開始時に必要となるプロダクト
後提条件	作成されるプロダクトおよび品質要件
作業方法	作業の詳細
チェックリスト	解決法データベースの “問題” 属性のリスト
備考	特記事項

表 3 解決法データベースのスキーム

属性	内容	タイプ
問題	再計画の原因となった問題	文字列
キーワード	問題のキーワード	文字列
作業	問題の生じた作業	文字列
リスクタイプ	問題の要因となったリスク (環境 / 実現方法 / ドメイン / その他)	整数
方法	問題の解決法	文字列
プロセス	拡張 UPM で記述されている解決法の作業モジュール	図形オブジェクト
タイプ	解決法のタイプ (追加 / 置き換え / モニター)	整数
参照回数	リスク回避のために参照された回数	整数
備考	特記事項	文字列

アグラムのモジュールの記法である大括弧で一連の作業を表す作業モジュールを表し，条件分岐の記法である内部で分割された大括弧で条件による作業の分岐を表現している．図4(1)の作業モジュールでは，作業1，作業2，作業3の作業が少しずつ遅れて並列に実行される作業モジュールを示めしている．図4(2)の作業の分岐は条件が満たされる時は作業a，条件が満たされない時は作業bが行われることを示す．

解決法データベースには，開発中に生じた問題の解決法が形式化されて蓄積される．解決法データベースは，開発組織により開発中の問題の解決に用いられたり，新たな解決法が実行されると更新される．提案するプロセス改善方法で用いる“解決法データベース”には，以下の属性がある（表3）．

問題: 再計画の原因となった問題．

キーワード: 問題のキーワード．

作業: 問題の生じた作業．

リスクタイプ: 問題の要因となったリスク．

方法: 問題の解決法．

プロセス: 解決法が記述されている作業モジュール．

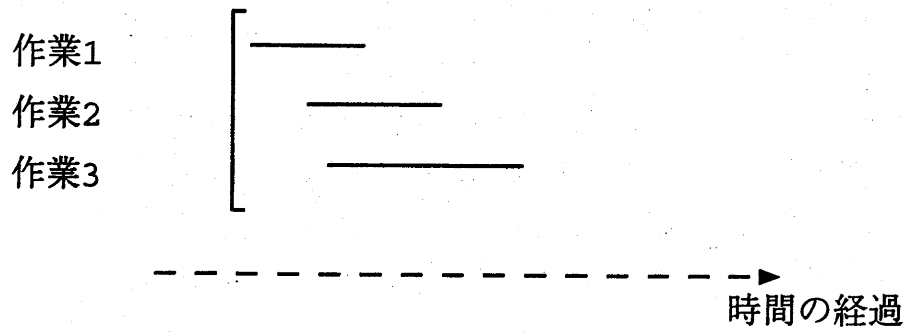
タイプ: 解決法のタイプ．

参照回数: リスク回避のために参照された回数．類似する解決法の一覧は，この属性でソートされる．

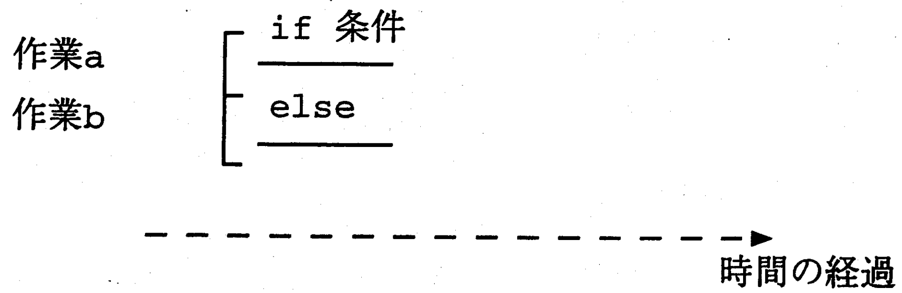
備考: 特記事項．

このうち，リスクタイプとタイプは内部コード（整数）化され，作業およびキーワードは独立した表で管理される．

作業報告書は図5の例に示すような形式で，一週間に一度記述され，スケジュール表とともに報告されることを前提としている．各項目には，以下の内容が記述される．



(1)作業モジュール



(2)作業の分岐

図 4 プロセスの記述

週間予定／実績:	前回報告後の実績と今後の予定.
今週提起の問題／ペンディング事項:	前回報告後に発生した問題と決定が保留になっている事.
前回提起の問題の対処:	前回報告した問題に対して行ったこととその状況
その他:	特記事項
別途資料:	添付する資料など

3.3.4 プロセス改善の手順

小型計算機上のソフトウェア開発におけるプロセス改善の手順を以下に述べる．このモデルでは，リスクにより生じた問題と解決法は，プロジェクト管理者が毎週報告する作業報告書から収集され，蓄積される．収集された問題が頻繁に発生する場合は，その解決法が標準プロセスに組み込まれる．開発中に問題が生じた時は，蓄積された解決法から類似する (すなわち，同一の分類や同一のキーワードの) 解決法の一覧を表示し，新しい解決法が創造されるか，あるいは過去の解決法がそのまま適用される．

図6で示す太い点線の矢印の順に開発が行われ，プロセスが改善される．プロセス記述のうち，置き換えや追加の細矢印の先にある太線と，解決法の中にある太線は，再計画された作業を示す．プロセス改善の手順を以下に述べる．

1. **ソフトウェアプロセスモデルに基づき開発する．**
   すでに得られているソフトウェアプロセスモデルを参考にして，実際のプロジェクトの制約に合った計画が立てられ，プロジェクトチームにより実行される．ソフトウェアプロセスモデルが無い場合は，ドキュメントの基準等を定めている作業基準書等をソフトウェアプロセスモデルの代わりに用いる．ソフトウェアプロセスモデルに準じた計画を立てるが，プロジェクト管理者の判断が作業報告書等に文書化されていれば，変更を加えてもよい．
2. **既知の問題は監視される．検出された問題は過去の解決法や，新たに創造された解決法により解決される．**

年　月　日

課長殿

プロジェクト進捗状況報告書

報告者氏名　　　　印

1.週間予定/実績

2.今週提起の問題/ペンディング事項

3.前回提起の問題の対処

4.その他

5.添付資料

*スケジュール表を添付のこと

図 5 作業報告書の形式の例

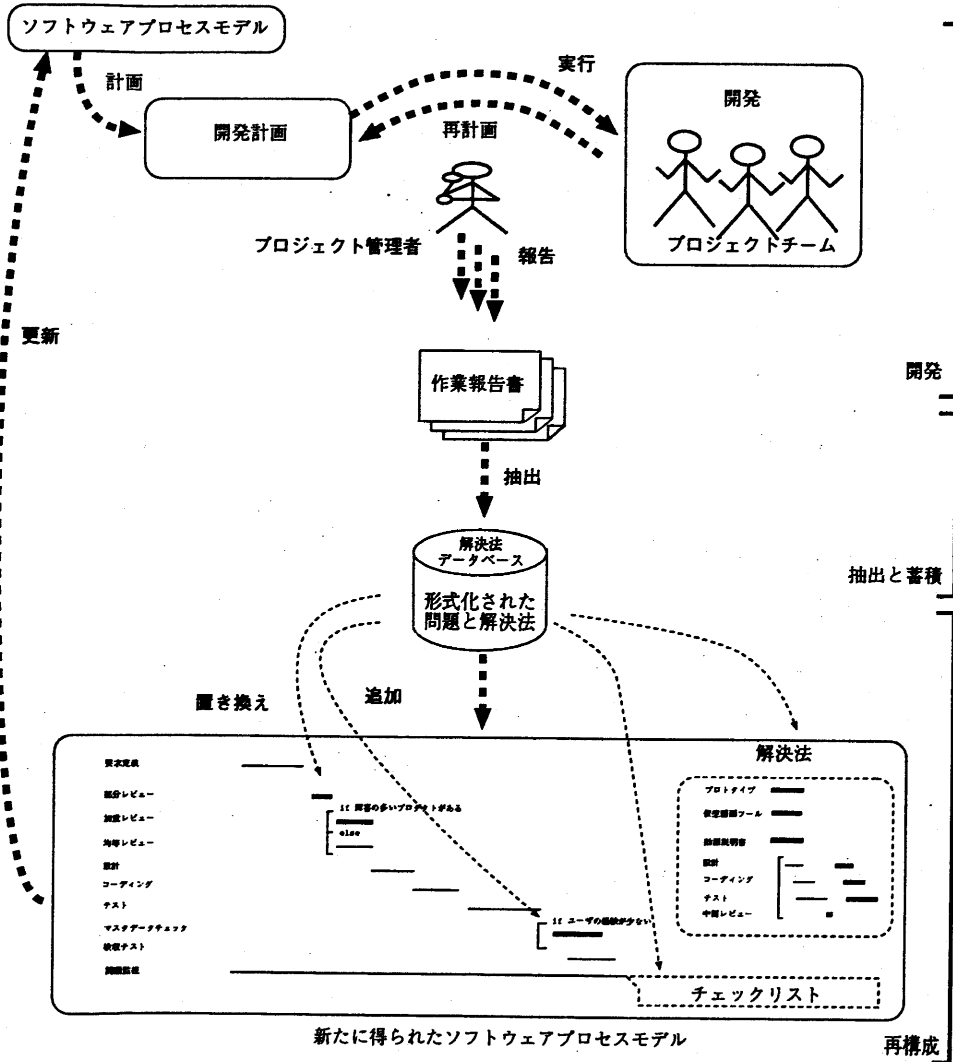


図 6 提案する開発モデル

既知の問題はメインプロセスに組み込まれた作業で検知されるか，チェックリストに基づいた問題監視プロセスにより検知される．開発中に問題が検知された場合は，メインプロセスに組み込まれた解決法で解決される．解決法が組み込まれていなかったり，不十分と判断された場合には解決法データベースに蓄積された解決法の追加や置換えが行われる．あるいは，過去の解決法データベースの参照によってプロジェクト管理者が新たに創造した解決法を用いて再計画され，問題の解決が図られる．

3. **再計画され，実施された解決法は，作業報告書に記述される．**
   作業報告書は工程の進み具合を報告するものである．工程の進捗にかかわる問題や，再計画の原因と内容等は必ず記述されるはずである．原因とは問題であり，再計画の内容は解決法であるので，記述された内容には問題とその解決法が含まれている．
4. **問題と解決法を作業報告書から抽出し，形式化し，蓄積する．**
   作業報告書から再計画の記述を探し，問題とその解決法を抽出する．リスクタイプ，解決法のタイプを分類し，形式化して解決法データベースに蓄える．
5. **ソフトウェアプロセスモデルを再構成する．**
   発生した問題が頻繁に発生し，かつ，問題の発生する時期が特定できるのであれば，ソフトウェアプロセスモデルに組み込む．問題の発生しやすい時期に作業の分岐として解決法を組み込み，問題が検知されれば解決法が実行されるようにする．問題が頻繁でなかったり，時期が特定できない場合には，問題監視プロセスのチェックリストに加える．

上記の手順を繰り返すことで，過去の経験を共有し，開発者の創造力を伸ばせ，問題が生じた場合も柔軟に解決ができるようになると考えられる．開発中に生じる問題による被害が少なくなると考えられるので，リスクの低減された開発が行えると考えられる．

3.4 適用実験

3.4.1 適用実験 1

提案するモデルを実際のプロジェクトに適用し，解決法の収集，蓄積，ソフトウェアプロセスモデルの再構築と，全ての解決法が作業報告書に記述されているかを確認した．具体的には，小型計算機上のエンドユーザ向けアプリケーション開発のプロジェクト管理者にインタビューした．インタビューで収集された問題と解決法を分類し，新たなソフトウェアプロセスモデルを得た．その結果，インタビューにより収集された問題と解決法のうち，作業報告書に記述されなかったのは1件であった．

対象としたプロジェクトは，エンドユーザ向けのアプリケーションを開発した．実行及び開発環境として Macintosh 上で 4thDIMENSION(DBMS) が用いられた．開発者1名，管理者1名の6ヶ月間のプロジェクトである．

開発中に生じた問題とそれに対する解決法をインタビューにより収集した．この結果を表4に示す．表には2行ごとに解決法が示され，上段の左から，解決法の番号，問題，問題が発生した時の作業，リスクタイプ，下段の左から，解決の方法，解決法のタイプ，作業報告書への記述の有無を示している．このプロジェクトでは，6つの問題と解決方法が収集された．このうち5つまでが，作業報告書に記述されていた．記述されなかった No.1 の解決法はユーザが仕様書を理解できないため画面イメージによるによるプロトタイプが再計画されたというものである．これは，工数や進捗に影響がなかったこと，比較的一般的な解決方法であることから，作業の進捗を報告する目的で記述される作業報告書に記述されなかったと考えられる．これは，文献 [18] で述べられているように，データがどのように役立つかをソフトウェア担当者に教えれば，彼らは結果に興味を持ち更に注意深く記述すると考えられる．

図7に適用結果を示す．図 7(a) が初期のソフトウェアプロセスモデルである．図 7(b) が開発終了時のプロセスであり，図 7(b) の作業のうち太い線の6つの作業は開発中に問題が発生したためにプロジェクト管理者により解決法が創造され，再計画された作業である．作業の左の番号は表4の解決法の番号を示す．複数の作業が並列に行われ，No.5 の作業は2回に分けて行われた．

表 4 収集された問題と解決法

No.	問題 解決法	作業 タイプ	リスクタイプ 報告
1	仕様書の理解が困難である	要求定義	実現方法
	画面イメージによるプロトタイプを実施する	追加	なし
2	2ページディスプレイが不足している	要求定義	環境
	仮想画面ツールインストールする	追加	あり
3	ユーザーズマニュアルが使いにくい	要求定義	ドメイン
	動画説明書を作成する	追加	あり
4	仕様書の品質が予測できない	レビュー	ドメイン
	部分レビューを行い，問題の多い部分に時間を加重配分する	置き換え	あり
5	デモバージョンが必要である	設計／コーディング／テスト	実現方法
	工程を2分割する	モニター	あり
6	ユーザのデータ入力誤り (一貫性なし) がある	検収テスト	ドメイン
	データチェックツールを作成する	追加	あり

図7(c)が6つの解決法から再構成されたソフトウェアプロセスモデルである．収集された解決方法のうち問題の発生する時期が特定でき，頻繁に問題の生じる可能性のある2つの解決法がソフトウェアプロセスモデルに組み込まれた．具体的には，No.4の部分レビューを行い問題の多い部分に重点的にレビューを行うという解決法と，No.6のユーザの入力したマスタデータの一貫性が低いためデータチェックツールを作成しチェックするという解決法が組み込まれた．組み込まれた解決法は作業の分岐として組み込まれ，問題の生じた時のみ解決法が実行される．点線で囲まれたNo. 1, 2, 3, 5の作業はソフトウェアプロセスモデルに組み込まれなかった解決法である．組み込まれなかった解決法は，組み込まれた解決法とともに解決法データベースに蓄積された．また，組み込まれなかった問題は問題監視プロセスのチェックリストに追加され，同じ問題が発生した場合に用いられる．

新たに得られたソフトウェアプロセスモデルに基づき開発すれば，開発中に生じた問題の経験を組織内で共有し，問題の解決が容易になる．提案するプロセス改善法により，ソフトウェア開発のリスクに対しより堅牢なソフトウェアプロセスモデルが得られたと考えられる．

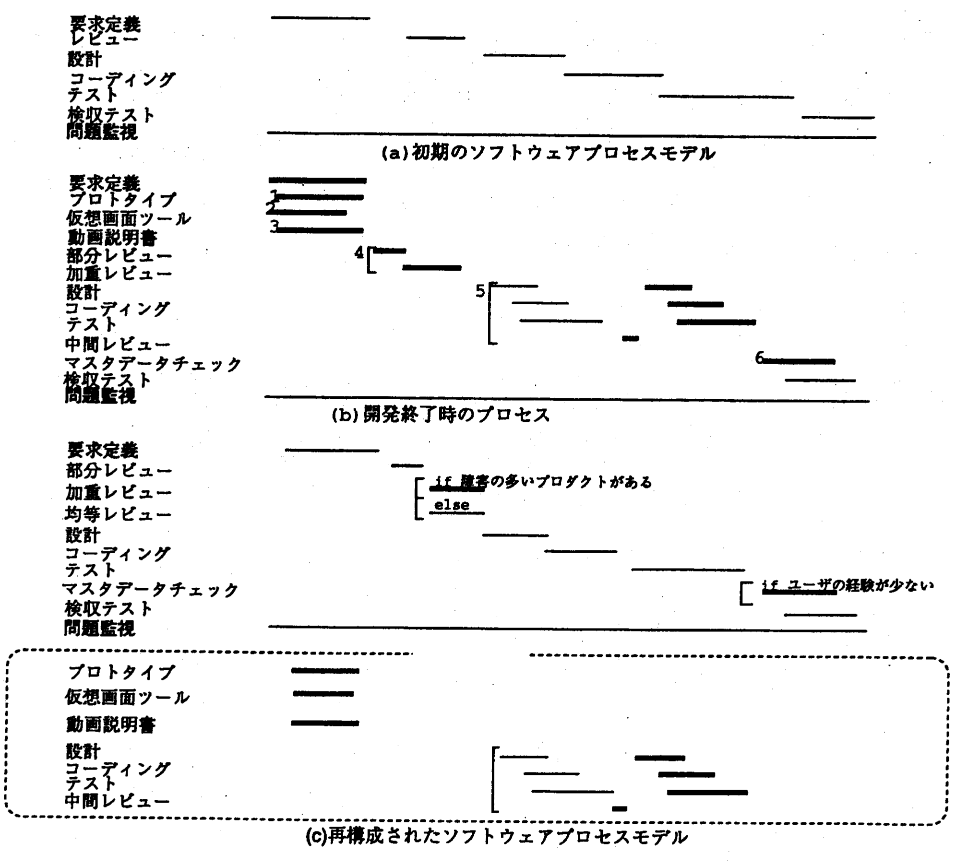


図 7 適用結果

3.4.2 適用実験 2

解決法データベースのプロトタイプを作成し，実際のプロジェクトで作成された作業報告書を用いて，解決法の抽出を実施した．本実験の結果，提案する改善法を用いた解決法データベースの開発/導入工数と，解決法の抽出に必要な運用工数が，比較的少ないことがわかった．

(1) 解決法データベースの開発

開発した解決法データベースを以下に説明する．解決法データベースは Microsoft Windows 3.1J 上のリレーショナル DBMS である Lotus APPROACH を用いて構築した．

仕様： 図 8 に解決法データベースの画面を示す．図 8 に示すフォーマットで入力，検索，印刷が行える．また，表形式の表示と印刷も可能である．

表 3 で示した属性は，使用した DBMS の機能を生かして実現した．問題，作業および方法は文字型である．キーワードは 3 つの属性に分けそれぞれ文字型，リスクタイプとタイプはデータ範囲の自動チェックをつけた数値型，プロセスは OLE(Object Linking and Embedding) オブジェクトの図形型，参照回数は数値型，備考は可変長の文字列であるメモ型になっている．

また，入力方式も使用した DBMS の特徴を生かして実現されている．データ範囲に制約のあるリスクタイプとタイプは，いずれか一つの入力しか許されないラジオボタンを用いて，間違った値の入力ができないようになっている．また，作業と 3 つのキーワードはフィールドボックス付きのドロップダウンリストになっている．これは入力枠（フィールドボックス）の右端をクリックすると既に入力されている内容をドロップダウンリストと呼ばれるメニューから探せ，もし入力したいデータがなければフィールドボックスに新しいデータを入力できる．このメニューは重複のない別の表であるので，入力を容易にするとともに同じ内容で異なる表現のデータが入力されることを少なくする．

プロセスの記述は OLE を 2 段階に用い，容易に参照できるようになっている．拡張 UPM で記述されたプロセスが表示されている四角の部分をダブルクリック

するとプロセス記述に用いたドローツールが自動的に起動され，さらに各作業の右端にある書類の形をしたアイコンをダブルクリックすると表2に示す内容を記述したテキストが表示される．

結果： 実験の結果，開発工数，導入工数は比較的少なかった．これは，提案するプロセス改善法で用いるデータ構造が単純であり，高機能なデータベースソフトの利用が可能になったからである．

支援ツールのプロトタイプの開発工数は，ツールのインストールと学習期間を含めて3人日であった．提案するプロセス改善法で用いるデータ構造が単純であること，高機能なデータベースを用いたこと，により比較的少ない工数で開発できた．

支援ツールの導入工数は，リレーショナルデータベースのインストールに約1時間と作成済みのデータベースのコピー (現状では数秒) が必要であるが，比較的少ない工数であると言える．

考察： 提案するプロセス改善法に基づいたツールは，利用される規模に合わせて拡張可能であり，開発/導入コストを常に少なくできる．

開発されたプロトタイプは，複数プロジェクトでの使用を考慮していない．複数プロジェクトで使用する場合は，さらに1人日ほど開発作業が生じるが，導入作業で必要であったデータベースのコピーが不要になり，ネットワークで共有するだけになる．

また，1万件以上のデータを蓄積する，より本格的なデータベースを構築することも可能である．その場合は，利用による効果も増加していると予想され，増加する開発/導入コストも組織全体では問題にならないと考えられる．

以上から，提案するプロセス改善法に基づくツールは，開発工数と導入工数は利用される規模に合わせて必要とされるので，文献 [5] に示されるように，集中した支出を避けることが可能である．

(2) 解決法の抽出

支援ツールの運用工数も，比較的少ない工数であった．

対象： 約3年間のプロジェクトの作業報告書から解決法のデータを収集した．収集作業は作業報告書からのデータの抽出と，データベースへのデータ入力に分けて一人で行った．

結果： 43件の解決法のデータを，比較的少ない工数で収集できた．データの抽出には約55分，データベースへの入力は約1時間30分で終了した．

本実験により，適用実験1で述べたプロジェクトの6件の解決法を含め，重複を除いた解決法の全体として39件が蓄積できた．

考察： 約3年間のプロジェクトにおいて，3時間弱の運用工数は比較的少ないと言える．プロジェクトの数が増加した場合も，運用工数は比例増加すると考えられるので，文献[5]に示されるように，集中した支出を避けることが可能である．

本実験の結果，提案する改善法を用いた解決法データベースはデータ構造が単純であり，開発/導入工数は比較的少ないことがわかった．また，データベースに蓄積する解決法は，通常業務で記述される作業報告書から，容易に抽出されるので，SEPGを必要とするCMMよりも運用工数が少ないといえる．さらに，提案する方法は徐々に規模を大きくできるので，集中した支出も避けることが可能である．このことから，提案するプロセス改善法は，導入/運用工数の制約が大きい小型計算機上でのソフトウェア開発のプロセスを改善できると考えられる．

解決法データベース

問題

仕様書の品質が予測できない

No. 4

○ 現境
○ 実現方法
◉ ドメイン
○ その他

作業

レビュー

キーワード1	キーワード2	キーワード3
仕様書	品質	予測できない

方法

部分レビューを行い，問題の多い部分に時間を加重配分した

○ 追加
◉ 置き換え
○ モニター

参照回数 1

部分レビュー

加重レビュー

備考

図 8 解決法データベース

3.5 まとめ

本研究では，小型計算機上のソフトウェア開発におけるリスクの低減を目的とした，ソフトウェア開発プロセスの改善法を提案した．提案する方法では，一般のソフトウェア開発で広く用いられている作業報告書の記述から開発上の問題とその具体的な解決法を収集し，形式化してデータベースに蓄積する．形式化された解決法をプロセスモデルに組み込むことにより，リスクに対して堅牢なプロセスモデルを得ることが出来る．蓄積された解決法は，開発上の新たな問題に対する解決法の創造に利用可能で，より現実的で具体的な解決法を発想する手助けとなる．提案する方法を実際のソフトウェア開発プロジェクトに適用した結果，小型計算機上のソフトウェア開発でよく発生する6つの問題に対する具体的な解決法が収集でき，それらを組み込んだ新しいプロセスモデルを得ることができた．新しいソフトウェアプロセスモデルには2つの解決法が組み込まれ，残りの4つの解決法は，将来生じる問題の解決のために解決法データベースに蓄積された．提案する方法に基づくデータベースの構築と，作業報告書に基づく解決法の収集と蓄積の試行の結果，改善法の導入と運用にはわずかな人員と労力しか必要とならないことが確かめられた．

本研究は小型計算機上のソフトウェア開発プロセスをCMMにより改善する際の問題点に基づいて議論したが，CMMの有効性を否定するものではない．提案するプロセス改善方法は，ソフトウェア開発上の問題を解決する具体的な解決法を収集し，開発中に生じる問題の影響が少なくなるようにプロセスを改善するモデルである．開発プロジェクトと組織の規模が大きければ，提案するプロセス改善法により小型計算機上のソフトウェアに特有の問題の解決が容易になり，CMMを用いたプロセス改善が容易になると考えられる．

提案したプロセス改善法をより効率良く計算機により支援すれば，プロセス改善の運用工数はさらに小さくなると考えられる．今回のプロセス改善のための解決法の収集は手作業で行った．将来的には，作業報告書を電子化し，タグ付けによる効率的な解決法の収集をする予定である．また，さらに解決法を収集し蓄積するとともに，効率的に検索できるようなキーワードの体系化を予定している．

4. レガシーな組込みソフトウェアの改良法

本章では，改造の繰り返しにより保守性が低下した組込みソフトウェアを対象として，プログラムのモジュール構造を評価する方法を提案すると共に，提案方法に基づき開発したソフトウェア改良支援ツールについて述べる．提案する評価法は，構造化分析法におけるモジュール間結合度とモジュール凝結度の概念を応用し，組込みソフトウェアで多用される外部変数によるデータの受け渡しを含めたモジュール構造の改良を支援するものである．提案方法は，(a) 広く知られている方法論である構造化設計が定める基準により，プログラム構造を評価できる，(b) 利用者の経験度を問わず短時間で習得できるように，設計ガイドとチェックリストが整備されている，ことで導入工数を少なくしている．提案方法に基づく支援ツールを開発現場で試行した結果，コードレビューでは指摘できなかったプログラム構造の問題点を，短時間の評価で指摘することができ，運用工数が少ないことも確認された．

以下，4.1 節で関連研究，4.2 節で提案する評価法，4.3 節で開発したツール，4.4 節で結果と考察，4.5 節でまとめを述べる．

4.1 関連研究

4.1.1 プログラムの評価法

レガシーソフトウェアには，柔軟に保守できるプログラムであるかどうかがわかるような評価法が必要である．保守には障害を少なくする修理保守，性能・機能・保守性などを改良する完全化保守，環境の変化にあわせる適応保守がある [31]．修理保守を対象とする場合は，文献 [48] に示されるように，欠陥数と相関の高い評価法が必要である．完全化保守や適応保守には，プログラムの変更に対し，柔軟に対応できるような評価法が必要である．レガシープログラムを生じさせるのは，完全化保守であるので，プログラムの新規作成時に用いられるような評価法が有効であると考えられる．

構造化設計では，理解しやすく，変更が容易なプログラムを作成するために，モジュール間のインタフェースを示す「結合度」，同一モジュール内の機能的連結の強さを示す「凝結度」，保守性を高めるための「追加の設計ガイド」で設計を評価する [39]．構造化分析/構造化設計はオブジェクト指向の基本になっているものであり [2]，これらの考え方は現在でも非常に有効な考え方であると考えられる．

結合度は2つのモジュール間の関係度合いを示しており，狭く，直接的で，局所的で，明確で，柔軟な連結にすることを目指している．正常な結合として，単純なデータをパラメータとする「データ結合」，複合データを渡す「同一データ結合」，他のモジュールの内部ロジックを制御する「制御結合」の3つがある．良くない結合として，外部変数によるデータの受け渡しを行う「共通結合」，他のモジュールの内部を参照する「内容結合」などがある．なお，「データ結合」であってもパラメータが20を超えるような難解なモジュールは凝結度が疑わしいとされている．また，「共通結合」に似ていても，データベースを用いる「データベース結合」は，不揮発性であり，方法論に基づき，一度の処理では必要なデータのみ操作する点で「共通結合」とは異なるとされている．

「凝結度」はモジュールの強さを表し，ただ一つの問題に関連した仕事をこなす「機能的」，あるモジュールの出力が次のモジュールの入力に次々につながる「逐次的」，同一のデータを扱う「通信的」，順に実行される「手順的」，同一の

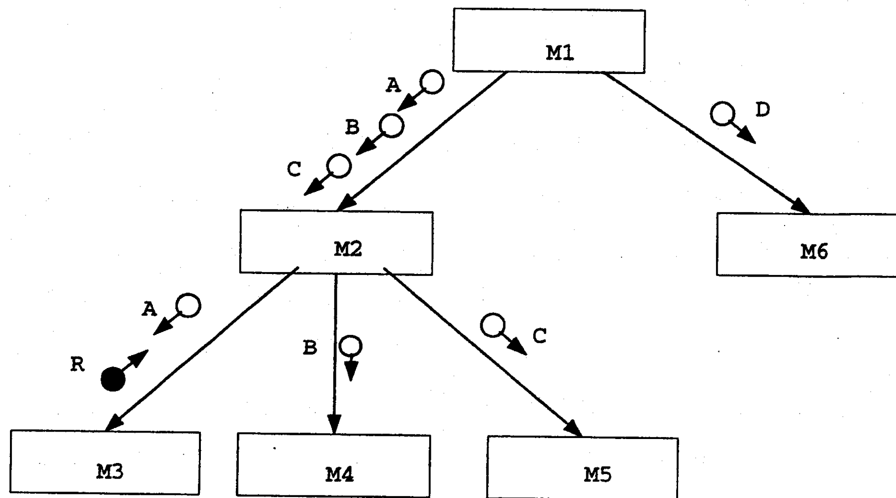


図９ 構造図を用いた評価

タイミングで実行される「一時的」，同一のカテゴリ処理の「論理的」，関係のない処理の「偶発的」に分けられ，「機能的」が最も良く，「逐次的」と「通信的」までが保守しやすく，他は保守が困難なモジュールとされている．

「追加の設計ガイド」は，「結合度」と「凝結度」だけでは不十分な点を補足し，保守性を高める指針である．具体的には，トップダウン設計，機能を重複させない，小さく，汎用的に，上位の管理モジュールと下位の作業モジュールを分けるといった，評価の基準としてモジュールの機能分解の基準を示している．

図９は従来の方法論で用いられる構造図である．構造図では上位と下位の呼出関係を矢印で，渡されるデータを丸のついた矢印で表す．図１は比較的保守性の良い構造を示しているが，保守性の悪い構造の場合，それはデータの流れに現れる．仮にモジュール M1 が最上位の関数でなく，変数 A,B,C,D が全て引数で渡されたなら，M1 のインタフェースはやや複雑なものになる．変数 A,B,C を M1 が操作しないなら，M2 以下の処理とそれに関係するモジュールだけで操作するデータであり，それらを構造体へ統合し，インタフェースを単純化すべきかを検討する必要がある．次に，モジュール M3 は M4 と M5 に共通の前処理であるが，独立した共通モジュールにならずに M4 と M5 に M3 の処理が組込まれていた場

合，M4とM5のインタフェースに変数Aが共通に現れ，インタフェースはより複雑になり，共通モジュールを抽出するきっかけになる．

オブジェクト指向においても上記の評価法は用いられており，現在でも有効な評価法であると考えられる.「結合度」と「凝結度」によるモジュール化は，オブジェクト指向におけるパッケージングの際に用いられている [7]. また,「追加の設計ガイド」もトップダウン設計を除いて再利用のための作法として用いられている [40]. これらの評価法は一般的なものであり，より良いモジュール化が可能になると考えられる．

4.1.2 レガシーソフトウェアの改良

過去のソフトウェアを改良して保守性の高いソフトウェアを作る方法は，プログラム全体を再構築するリエンジニアリングとして研究されてきた．従来の研究には大きく分けると2つの方法があり，トップダウンにプログラム全体を作り直す方法と，レガシーソフトウェアから再利用可能な部分をコンポーネントととして取り出して，それをもとにボトムアップにシステムを再構築する方法である [3]. いずれの場合も専用の分析ツールやナレッジベース等を用いて，ドメインエキスパートと開発者が協力しながらプログラム全体の解析と比較的大規模な開発を行う必要があった．

しかし，限られた予算の中では，プログラムの部分的な改良しか行えない．ソフトウェアのリエンジニアリングを行う場合，改良により得られる効果と開発のリスクを考慮しなければならない [47]. 今後の市場性，競合他社の動向，利益等を考慮して予算を決める必要がある．限られた予算の中で改良を行うには，プログラム全体を改良するのではなく，特に問題のあると思われるところに限定して改良すべきである．文献 [15] ではプログラムの規模や分岐の数といった尺度を用いて改良する範囲を限定している．しかし，外部変数が用いられることは考慮されておらず，組込みソフトウェアに適用可能であるかを調査する必要がある．

また，常に高い保守性を維持するには，プログラミングを行った部分を常に見直しておくことが必要である．12のプラクティスを強調して実施する XP(eXtreme Programming) においては「リファクタリング」というプログラムの改良を常に

行うことで，保守性を維持している [5]．この様に徐々に改良する方法は，一度に大きな費用が発生しないので，投入可能なコストが少ない．多くの改造が行われているレガシーな組込みソフトウェアでは，改良された部分が徐々に広がるので，有効な方法であると考えられる．

プログラムの改良を効率よく行うためには，新しい評価方法が必要であり，それに基づく作業を支援するツールも必要である [11]．文献 [27] および文献 [39] では，プログラムの評価法を示し，その評価法を満たすプログラムを設計することで保守性の高いプログラムを実現しようとしている．しかし，これらの方法論では外部変数によるデータの受け渡しは良くないとされており，外部変数を用いざるを得ないプログラムを改良する方法は示されていなかった．また，文献 [12] のような外部変数の参照を表示可能なツールであっても，従来の方法論に基づいているので，外部変数を参照する関数を示す機能のみであった．

4.2 提案する評価法

本研究では，レガシーな組込みソフトウェアの保守性を向上させる目的で，外部変数によるデータの受け渡しを行うプログラムの評価法を提案する．本研究では，既存のプログラムの保守性の低下した部分を調査した上で，従来の評価法を拡張した．

4.2.1 レガシーソフトウェアの調査

(1) レガシーソフトウェアの調査方法

組込みソフトウェアには他のソフトウェアにはない特徴があるため，既存の方法論のみではプログラムを改良することはできない．そこで，レガシーソフトウェアを調査し，保守性の低下したパターンを以下の順に調査した．対象は金銭の入出力を行う端末機器のレガシーソフトウェアである．C 言語で開発されており，6748 ファイルから構成されていた．これらのうち，保守が困難な部分を対象ソフトウェアのエキスパートにインタビューし，それらを中心に調査した．調査は実行ステップだけでなく，コメントアウトされた過去のソースコードや修正履歴も調査した．

STEP 1 インタビュー

まず，保守性の低下したところはどのような部分であるかを，エキスパートにインタビューした．保守の困難なところは，改造が繰り返された部分であり，入力データや外部変数などに対して，多くの判定が必要な部分であった．条件分岐の多いところの保守性が悪いという評価は文献 [15] と同じであった．

STEP 2 条件分岐を多く含む関数の調査

インタビューの結果を元に条件分岐を多く含む関数を調査した．条件分岐を多く含む順に並べ替えた関数のリストをエキスパートに示したところ，保守が困難な部分はすべてリストの上位にあったが，条件分岐を多く含む関数すべての保守性が低下しているとは言えなかった．

STEP 3 保守性の悪いプログラムの調査

条件分岐を多く含むプログラムのうち，保守性の低下したプログラムとそうでないプログラムを比較したところ，保守性の低下したプログラムには，以下の特徴があった．

- 多くの外部変数を参照する．
- モジュール構造が複雑である．

(2) 外部変数

条件分岐を多く含むプログラムのうち，保守性の低下したプログラムは，多くの外部変数を参照するという特徴のものがあった．機能が追加された際に，追加された処理に必要となる動作モードの設定や障害復旧対策のために，各関数で参照する外部変数が徐々に増えたのである．保守性の低下した関数は単純な機能を実現する関数ではなく，(1) 複雑な判定を行う，(2) 多くの外部変数を判定し順番に処理を実行する「手順的な凝結度」[39] で，複数の機能が一つの関数になっている，という特徴があった．(1) は徐々に増加した外部変数がまとめられずに，多くの外部変数を参照していた．(2) は手順的強度であっても比較的単純であった関数が，長年にわたる機能追加により徐々に複雑になったものである． 特に (2) では，機能が追加された際に，既存プログラムのコピーを改造して類似した関数を作るということも行われていた．既存のプログラムの再テストを避けることで，期間短縮をはかっていたのである．

(3) モジュール構造

条件分岐を多く含むプログラムのうち，保守性の低下したプログラムには，モジュール構造が複雑であるという特徴のものもあった．単純でない比較的上位の関数を再利用して，共通ルーチンとして使っていた．モジュールを再分割しないで，コードの共有化が行われており，再帰処理になっているところもあった．これは，すでに実績のあるソースを用い，既存の処理への影響を最小限にすることで，期間の短縮やコストを削減しようとしていたのである．

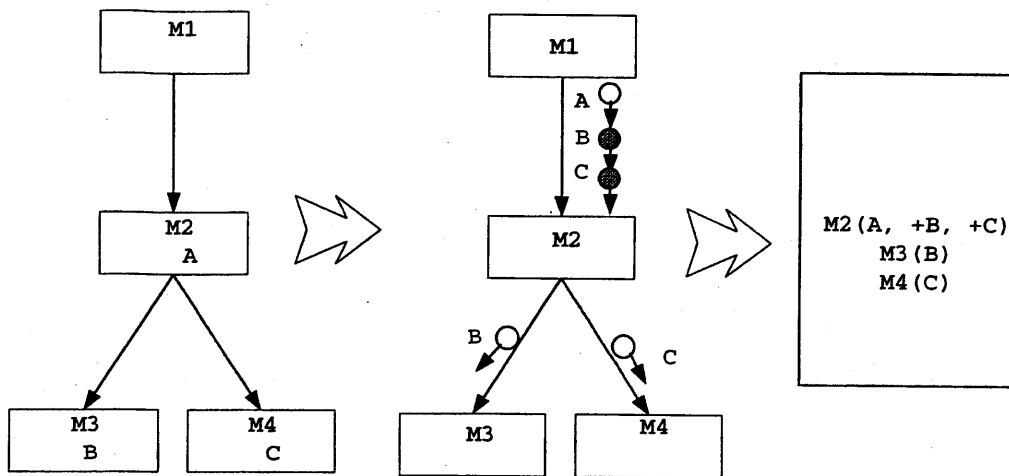


図 10 基本的なアイデア

このような，保守性の低下した部分は改良される機会はあったが，正しく改良されなかった．システムの大規模な見直しの際には保守性の低下した部分を含めて改良が試みられている．対象としたソフトウェアは文献[13]と同じようにカスタマイズで追加された新機能を標準化し，複数バージョンの管理を減らす努力が行われている．しかし，機能の統合は行われるものの，外部変数でデータを受け渡すプログラム構造全体を鳥瞰した上で，ソースコードから保守性の低下した部分をもれなく判定することは難しかった．

すなわち，以下が不足していたので，保守性が改善されず，レガシーソフトウェアになったと考えられる．

- 外部変数によるデータの受け渡しを用いたプログラムを評価する方法論．
- プログラム全体を鳥瞰しながら，方法論の実施に必要とされる情報を表示するツール．

4.2.2 レガシーな組込みソフトウェアの評価法と改良法

本研究で提案する評価法の基本的なアイデアは，外部変数によるデータの受け渡しを引数と同じように扱うというものである (図 10)．引数が外部変数と違う点は，以下の 3 点である．

- インタフェースが関数の定義として可視化される．
- 引数は上位の関数に与えられるので，同一の引数を使用する関数がグループ化される．
- スタックとして実装されるので，名前渡しの場合は，データが破壊されない．

対象の関数及びその下位の関数で使用している外部変数を関数のインタフェースとして表示することで，3 点目のデータの保護は実現できないものの，引数と同じ方法論を用いることができることになる．4.1.1 で示したように「データベース結合」は，不揮発性であり，方法論に基づき，一度の処理では必要なデータのみ操作する共通結合である．組込みソフトウェアで用いられる外部変数は不揮発性であり，引数に関する方法論は文献 [39] を用いることで，参照するデータごとにグループ化 (局所化) され，小さく汎用的なモジュール (関数) を実現できるようになる．すなわち，データベースは用いないもののデータベース結合に近い形になり，より保守の容易なプログラム構造になる．

本研究で提案する評価法に基づくと，4.2.1 で挙げた問題は図 11 の様に改良できる．A-S は各関数で参照する外部変数，四角は関数，細矢印は呼び出しを示す．多くの外部変数を参照している場合には，外部変数の統合が必要な場合 (a-1) と，共通処理を抽出する必要のある場合 (a-2) がある．参照されている外部変数がどのような分類に属するかで評価が変わり，改良方法が変わる．同一の分類の外部変数が多い場合は，それらを構造体に統合し，よりわかりやすいインタフェースに変更できる可能性がある．また，異なる分類の外部変数が多い場合には，外部変数の判定処理を抜き出して，共通処理の処理として抽出できる可能性がある．また，図 11-b のように，再帰処理で実装されたために，モジュール構造が複雑になった場合には，上位関数の共通処理を抽出して，新しい共通関数を作成し，上

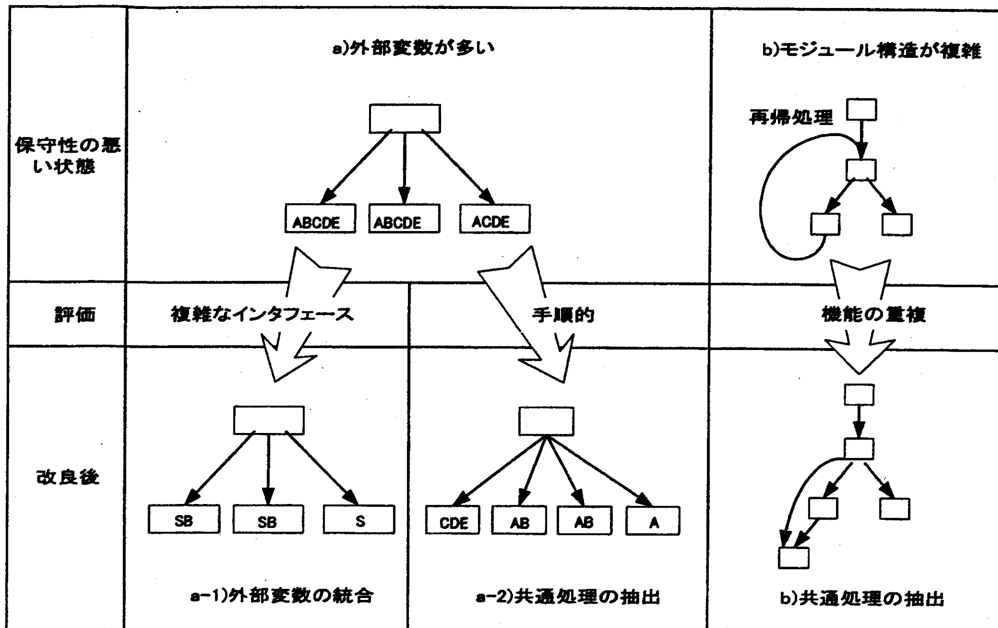


図 11 評価と改良法

位関数と下位関数を新しい共通関数を呼び出す様に改造することで，モジュール構造を単純にできる可能性がある．これは文献 [5] で"the Once and Once rule"と呼ばれるリファクタリングと同じ方法である．これらの評価を行って改良するには，外部変数の参照関係を引数と同様に表示するだけでなく，全体を鳥瞰でき，構造体への統合をシミュレートできる必要がある．構造図は大規模になると作成や管理の負担が増えるだけでなく，全体を見渡すことが難しくなる．そこで，図10 の右に示すような関数の引数形式での表示が有効である．ただし，本来その関数で扱わない外部変数は区別する必要がある (図 10 では＋マークを付けている)．また，構造体への統合は修正範囲が全体に及ぶので，実際の修正が困難な場合がある．そこで，外部変数を実際に構造体に統合せずにどのような結果になるかを表示できれば，全体の構造がわかりやすくなり，共通処理の抽出など他の改良が容易になると考えられる．

4.3 改良支援ツール

4.3.1 ツールの設計と実装

ソフトウェア改良支援ツールは，以下の機能を実装している．

- 外部変数によるデータの受け渡しを上位関数とのインタフェースとして可視化できる．
- 外部変数を構造体として統合した場合のシミュレーションができる．
- 全体を鳥瞰しやすいように，プログラム構造をシンプルに表示できる．

具体的には，外部変数を構造体に統合する場合の定義を記述した「外部変数分類定義」とC言語の「ソースファイル」を入力とし，外部変数をグループ化し，関数のインタフェースとして出力する．

「外部変数分類定義」の内容は外部変数をどのようにグループ化するかを示したものである．テキストファイルとして記述されており，その形式は，一行が一つの外部変数グループに対応し，各行にはグループの名称と属する外部変数の名称がタブコードで区切られ列挙されているというものである．支援ツールは,「外部変数分類定義」に基いて，外部変数の名称を所属するグループ名称に置き換え，外部変数を構造体として統合した場合のシミュレーション結果を出力する．

なお，下位の関数が参照した外部変数も引数の形で (+記号により) 区別して表示できる．また，関数の呼び出し関係を字下げにより示す．すでに出力した関数以下は階層を展開して出力するか，あるいは [既] マークを付けて省略するかを指定できる．なお，再帰処理の場合は警告のために [再] マークを表示する (図 12).
本ツールは，組込みソフトウェアのプロジェクトの一環として開発したので，高い信頼性，短期間，低コストの制約を満たす様に設計した．

- 不具合の生じやすい構文解析にTAG生成ツールを流用した．
- 短期間で開発するために，スクリプト言語を用いた．
- 上記のツールのコストを削減するためにオープンソースのツールであるECTAGS[14]，Ruby[30] をそれぞれ用いた．

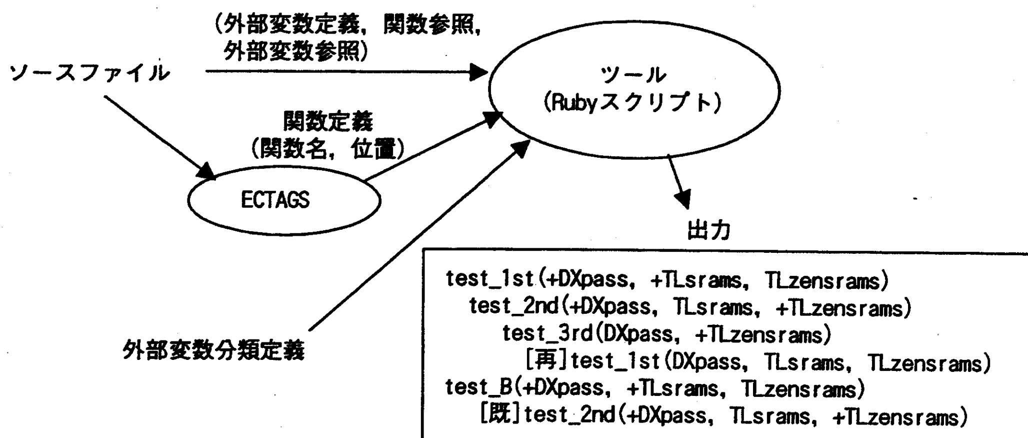


図 12 ツールのデータフロー

本ツールの機能を実現するには，関数の定義，外部変数の定義，関数の参照，外部変数の参照の情報をソースファイルから得る必要があるほか，外部変数をどのように統合するかを定義した外部変数の分類の定義が必要である．図 12 に示すように関数の名前と位置の定義を ECTAGS から，外部変数の分類の定義を外部変数分類定義ファイルからそれぞれ入力とし，ソースプログラムを部分的に構文解析して外部変数の参照と関数の参照を検索する，というスクリプトを Ruby で記述した．C 言語の構文のうち関数名とそのファイル上の位置がわかれば，

- 関数定義の抽出に必要な C 言語の多くの構文の解析が不要になる．
- 個々の関数のコード上で関数名を検索すれば関数の参照情報が得られる．

となり，ECTAGS の出力により，関数に関する構文解析の工数の多くを軽減することができた．なお，外部変数の定義は extern 宣言分を解析するだけで容易に抽

出でき，参照情報は容易に抽出できた．ECTAGSの利用とRubyにより，開発規模は約1000Step程度になり，約1人月で作成できた．

4.3.2 ツールの導入

利用者の経験度を問わずソフトウェア改良法を短時間で習得できるドキュメントを整備した．改良法に関しては，設計の指針を示した設計ガイドを作成したほか，具体的なソースを元に改良法の勉強会を主要なメンバーで実施した．設計ガイドには4.2.2で述べた内容のほか，4.1.1の記述も行い，本ツールの使用方法を中心になるべく一般的な内容で作成した．また，改良後の品質を維持できるように，設計とコーディングのチェックリストを，文献に基づいてそれぞれ作成した．既存の文献のチェック項目は，文献毎に注目する内容が異なるため，チェック項目の統合は難しく，設計で82項目，コーディングで98項目になった．これらをレガシーソフトウェアに特有の内容，一般的な内容，基本的な内容の3種類に分類し，開発者の経験にあわせて，重要な部分だけを短時間で読めるようにした．

4.4　結果と考察

4.4.1　試行

改良支援ツールを本格的に導入する前に試行を行い，ツールの効果を確認した．試行では，既にコードレビューの行われているプログラムをツールで評価（改良すべき点を指摘）し，コードレビューでの評価と比較した．

対象： 10年以上改造が繰り返されてきた組込みソフトウェアで規模は約5500ステップ，試行前にプログラム構造を熟知したエンジニアが8時間かけてコードレビューした．

試行： プログラム構造の知識のないエンジニアが，ツールの出力を元に約30分間かけて評価した．

結果： ツールの出力に基づき，図3に示した2つの「保守性の悪い状態」を次の通り指摘することが出来た．ツールによる評価は，コードレビューに取って代わるものではないが，レビューでの指摘漏れを防止する効果は高いと考えられる．

(a)「外部変数が多い」

モジュール構造図で最下位に位置する関数(単機能な汎用関数)を除いたほぼ全ての関数で，参照される外部変数の数が多いことが分った．この点は，コードレビューにおいても指摘されていた．なお，該当箇所を改良するためには多くの工数が必要となるため，必要性は認められるが実際には改良は行わない，とコードレビューでは判断されていた．

次に，ソースファイル上でのデータ種別の分類を元に作成した「外部変数分類定義」を用いてシミュレーションを行った．その結果，外部変数の数が多い関数のうち3つの関数については，外部変数が多数のグループに分類される（構造体として統合しても，数多くの構造体を用意しなければならない）ため,「共通処理の抽出」が必要であると判断された．これら3つの関数は，コードレビューにおいてもその問題点

が指摘されており，うち2つの関数は，処理が手順的であり改良（共通処理の抽出）が必要であると判断されていた．残る1つの関数は，プログラムの実装を考慮すると改良する必要はないと判定されていた．

(b) 「モジュール構造が複雑」

プログラム全体を鳥瞰した結果，再帰処理マーク（[再]）がツールにより付加されており，かつ，自己再帰でなく複数の関数で構成される部分をプログラム中に1ヶ所発見することが出来た．単純な機能でない上位関数が共通ルーチンとして使われており，本来は，共通処理を抽出して共通関数を作成すべき部分である．しかし，コードレビューでは，改良すべき点としての指摘はなされていなかった．

4.4.2 導入結果

試行後に実際の開発作業に導入し，開発が最終テスト段階になった時点で，アンケートを取った．このツールは，レガシーソフトウェアを改造して新しいシステムを作るプロジェクトに導入された．導入にあたり，設計ガイドとチェックリストを元に，より具体的な改良法の勉強会を中心メンバー3～4人で2時間ずつ，5回実施した．最終テスト段階のアンケートで，以下の意見が得られた．

- 改造元ソースの悪い構造が見えたので，元々開発された時と同じ構造にならないように参考にできた．
- 設計時に「保守性の高いソースを作る」という意識で開発・レビューできた．
- 元のプログラムに比べ，データを隠蔽したり，シンプルな構造にすることができた．その効果はその後の保守で実感した．
- 勉強会の実施により，知識が向上した．
- キャラクタインターフェースなので，あまり使わなかった．
- プログラム構造を大幅に変更することが恐かったので，直接コードを見てレビューをした．

4.4.3 オープンソースを用いた内製の効果

今回，オープンソースであるRubyとECTAGSを用いたツールを内製することで，市販のツールには無い，以下の効果があった．

- 利用者が多いツールであり，信頼性が高かったので，ツールの欠陥によるトラブルがなかった．将来，欠陥が生じた場合も，対応策を取れる可能性がある．
- スクリプト言語を用いて内製したので，必要な情報を必要なフォーマットで得られ，仕様の追加などの保守も容易であった．
- 社内に展開する際もコストが掛からない．
- WindowsとUNIXの2つの環境で実現でき，クロス環境でも用いることができた．

4.4.4 考察

試行では8時間かかったコードレビューに30分追加するだけで，新たな問題を発見できた．ソフトウェアの規模に対して比較的小さい運用工数であり，以降の工程でのデバッグ作業効率の向上という効果を考慮すると十分な利益があったといえる．

導入時には各人10時間を改良法の学習に費やしたが，広く普及している方法論であり，本来はツールの導入工数ではなく，研修として行われるべき内容である．組織の大小に関わらず，単年度の研修時間として実施が可能な工数である．

アンケートの結果を見ると，支援ツール導入の効果はあったが，担当する作業により評価は分かれていた．高い評価をしていたのはリーダであった．改良法が分かること，改良後の構造を確認できることから評価が高かった．従来は評価の方法論が確立していなかったので，レビューを行っても保守性を高められなかった．本研究では外部変数によるデータの受け渡しを含めたプログラム構造の評価法を提案し，新たなツールを開発し，ツールで支援することで，大きな負担をかけずにレビューができるようになった．一方，評価が低かったのはメンバであっ

た．メンバにとって，今回のツールは悪いところを見つける手助けはするが，改良作業そのものを支援しないことから評価が低かった．開発されたツールは，ユーザ・インターフェースを改良することで，利用率が向上すると考えられるが，大幅な変更に対する不安を取り除くためには，実績を積むほか，XPのリファクタリングの技術を導入するなど改良作業そのものの支援の検討が必要と考えられる．

オープンソースのツールを用いて，支援ツールを内製することは信頼性，納期，コストの面で効果があった．オープンソースは無料であるというだけでなく，信頼性が高いことから用いられることが多く [50]，本研究で使用したツールにおいても信頼性は高かったといえる．また，変更が容易で，クロス環境が実現できたなど，自由度が高かった．ユーザ・インターフェースが使いにくいなど，必ずしもすべての要求を満たすものではなかったが，最低限必要とされる機能を，少ないコストで，柔軟な仕様で実現することができた．既存のツールでは対応していないようなツールが必要な場合，不具合の生じやすい部分になるべくオープンソースのツールを用いて，内製することが効果的であると考えられる．

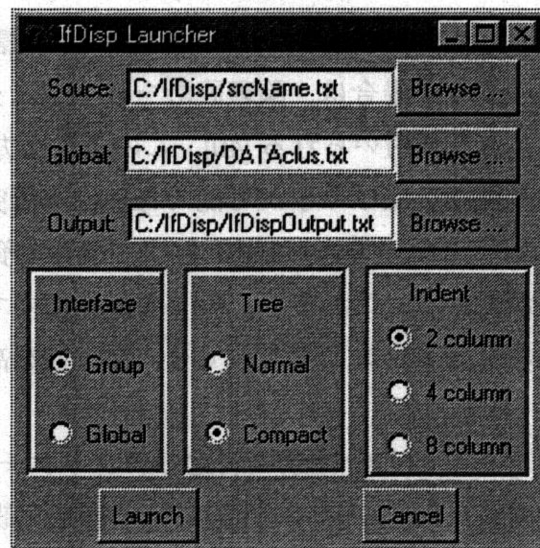


図 13 ツールのユーザ・インタフェース (GUI 版)

4.5 まとめ

レガシーソフトウェアを調査し，外部変数によるデータの受け渡しを含めたプログラム構造を評価する方法を提案した．また，オープンソースのツールを用いて支援ツールを作成した.このツールは，提案した評価法に基づき，外部変数によるデータの受け渡しを引数によるインターフェースとして表示すると共に，モジュール構造を表示するものである．従来レビューが困難であった，外部変数によるデータの受け渡しを含めたプログラム構造を示すことで，改良すべき点の検討や改良後のソフトウェアの確認に有効であった．また，オープンソースのツールを用いて内製することで，少ない工数で利用者の意見に合わせたツールを作ることができた．しかし，リーダだけでなくメンバが進んで使うようにするには，他の方法論との併用やユーザ・インターフェースの改良が必要である．現在，UNIX およびウィンドウズで動作するフリーウェアであるTcl/Tkを用いて，図13に示すグラフィカル・ユーザ・インタフェースを開発する等の改良を進めている．

本研究の特徴は，組込みソフトウェアの信頼性，納期，コストの制約の中で，よ

り保守性の高いソフトウェアを得る方法を示していることである．オブジェクト指向においても何らかの制約によりオブジェクト指向言語を用いることができない場合がある．このような場合は，論理設計をオブジェクト指向で行い，それを開発言語で写像する方法が用いられる [40]．このような方法は消極的ではあるが，方法論の適用できる範囲を広げる事が可能である．本研究も組込みソフトウェアの制約を満たす比較的少ない導入/運用工数で，既存の資産を最大限に生かしながら実施することができる．これまで，多くの制約の中で大規模な改良をあきらめざるを得なかったソフトウェアであっても，保守性を高めることができる可能性がある．

また，組込みソフトウェアの特徴は最近のアプリケーションの特徴でもあり，今回の成果が生かせる可能性がある.外部変数は構造化設計やオブジェクト指向設計の普及により，少なくなってきている．しかし，最近のアプリケーションにおいても，終了時の情報を不揮発メモリであるレジストリに保存しておき，再び実行された際に，前回の続きから実行できるように作られる場合が多い．この様なアプリケーションでは，システム関数を介しているが，実質的にレジストリのデータを直接操作している場合が多い．この場合のレジストリは，組込みソフトウェアにおける外部変数と同じ使い方であると考えられる．レジストリが複雑になり，プログラムの保守性が悪くなったプログラムに，今回の方法論とツールを適用できる可能性がある．

5. おわりに

本論文では，開発支援の導入工数と運用工数に制約を考慮した2つの支援方法を提案した.

まず，小型計算機上での開発プロセスの改善法を提案した．提案した改善法はソフトウェア開発で広く用いられている作業報告書の記述から開発上の問題とその具体的な解決法を収集，蓄積し，形式化して開発プロセスモデルに組み込むものである．作業報告書からの解決法の収集には僅かな運用工数しか必要とならない．また，解決法を蓄積するデータベースのデータ構造は比較的単純で，導入工数も小さい．蓄積された解決法は，開発上の新たな問題に対する解決法の創造にも利用可能で，より現実的で具体的な解決法を発想する手助けとなる．提案方法を実際のソフトウェア開発プロジェクトに適用した結果，小型計算機上のソフトウェア開発でよく発生する6つの問題に対する具体的な解決法が収集でき，それらを組み込んだ新しいプロセスモデルを得ることができた．また，提案する方法に基づくデータベースの構築と，作業報告書に基づく解決法の収集と蓄積の試行の結果，改善法の導入と運用にはわずかな人員と労力しか必要とならないことが確かめられた.

次に，再利用が繰り返された組込みソフトウェアの改良法を提案した．提案した改良法は外部変数によるデータの受け渡しを，引数と同様に上位関数とのインタフェースとして扱うことで，プログラム構造の評価を可能にする．提案方法は，(a) 広く知られている方法論である構造化設計が定める基準により，プログラム構造を評価できる，(b) 利用者の経験度を問わず短時間で習得できるように，設計ガイドとチェックリストが整備されている，ことで導入工数を少なくしている．提案方法に基づく支援ツールを開発現場で試行した結果，コードレビューでは指摘できなかったプログラム構造の問題点を，短時間の評価で指摘することができ，運用工数が少ないことも確認された.

本研究では，実証的ソフトウェア工学 (Empirical Software Engineering) の考え方を基に [4][20]，ツールの設計や実装を行うことで，より具体的な議論を行った.

ソフトウェア工学には，産業界からの要求 (ニーズ) から研究されるものと，学界にある理論 (シーズ) の応用として研究されるものがあると言われている [32].

生産活動から理論への方向のものと，理論から生産活動への方向のものであるが，近年では産業と学界の関係の問題として捉えられ[37][38]，産業と学界の歩み寄りが必要であると言われるようになった[21]．近年では産学共同プロジェクトや社会人の留学などの人的な交流により，産業界と学界の交流は盛んになりつつある．大学においては，社会人を迎え入れることで現場のニーズに触れられるような環境が作られるようになってきている[22]．

しかし，二者間の関係をソフトウェア開発に当てはめてみると，交流が盛んになるだけでは不十分であり，具体的な議論の積み重ねが必要なことがわかる．ソフトウェア開発において，利用者と開発者の交流が頻繁に行われても，期待したソフトウェアが開発できるとは限らないからである．そこで，本論文では文献[6]に示されるプロトタイピングの手法を研究方法としてに導入した．ソフトウェア開発における問題の解決法を提案するだけでなく，実際に開発支援ツールを開発することで，より具体的な議論を行えるようにした．今後も，ソフトウェア開発に関する提案だけではなく，具体的なツールの開発を行い，研究を進めていく予定である．

謝辞

本研究を進めるに当たり，常日頃より適切な御指導を賜わりました，奈良先端科学技術大学院大学 情報科学研究科井上 克郎 教授に，心から深く感謝申し上げます．

本研究を進めるに当たり，貴重な御指導を賜わりました，奈良先端科学技術大学院大学 情報科学研究科小山 正樹 教授に，心から深く感謝申し上げます．

本研究を進めるに当たり，貴重な御指導を賜わりました，奈良先端科学技術大学院大学 情報科学研究科関 浩之 教授に，心から深く感謝申し上げます．

本研究を進めるに当たり，研究テーマの決定，論文作成，発表準備まで，本研究を進めるに当たり，常に適切な御助言と根気強く熱心な御指導を頂きました，奈良先端科学技術大学院大学 情報科学研究科松本 健一 助教授に，心より感謝致します．

本研究を進めるに当たり，常に適切な御助言を頂きました，奈良先端科学技術大学院大学 情報科学研究科 島 和之 助手に，心より感謝致します．

本研究を進めるに当たり，常に適切な御助言を頂きました，奈良先端科学技術大学院大学 情報科学研究科 門田 暁人 助手に，心より感謝致します．

本研究を進めるに当たり，論文作成に関する適切な御助言を頂きました，奈良先端科学技術大学院大学 情報科学研究科 中小路 久美代 助教授に，心より感謝致します．

本研究を進めるに当たり，御助言や御協力をして頂いた鳥居研究室の皆様に，心から感謝致します．

本研究を進めるに当たり，御助言や御協力をして頂いたオムロン株式会社久保田 益史主査に心より感謝致します．

本研究を進めるに当たり，御助言や御協力をして頂いたオムロン株式会社沖田昌也様に心より感謝致します．

本研究を進めるに当たり，御協力をして頂いたオムロンソフトウェア株式会社の皆様，および，プロジェクトの皆様に心より感謝致します．

また，本学への進学に当たり，貴重な御助言を賜わると共に，在学中も適切なご助言を賜わりました，奈良先端科学技術大学院大学 情報科学研究科鳥居 宏次 副学長

に, 心から深く感謝申し上げます.

本学において研究する機会を与えて頂くと共に御助言を頂きました株式会社 SRA 丸森 隆吾 代表取締役社長, 岸田 孝一 最高顧問, 稲葉 実 取締役, 株式会社 SRA 先端技術研究所 松村 好高 代表取締役社長, 株式会社 SRA 鈴木 茂雄 主幹, 原田 一司 主席, 四谷事業所の皆様, および, 関西支社の皆様に心より感謝致します.

最後に, 経済的にも精神的にも苦しかった博士前期課程から, あまり良い夫・父親でなかった私に対し, 影ながら支えてくれた妻 佳子と, 安らぎを与えてくれた子供達に心から感謝します.

参考文献

[1] 赤間浩樹, 石垣昭一郎, "ダウンサイジングにおけるデータベース利用の課題と対策", Computer Today, Vol. 11, No. 60, pp.25–30, 1994.

[2] 青木淳, "オブジェクト指向システム分析設計入門", p.16, 1993.

[3] J. D. Ahrens, N. S. Prywes, "Transition to a legacy- and reuse-based software life cycle," IEEE Computer, Vol.28, No.10, pp.27–36, 1995.

[4] V. R. Basili, "The role of experimentation in software engineering: Past, Current, and Future," Proc. of ICSE-18, pp.442-449, 1996.

[5] K. Beck, "Extreme Programming Explained: Embrace Change," Addison-Wesley, pp.11-14, pp.21-25, pp.107–108, 1999.

[6] B. W. Boehm, TRW Defense Systems Group, "A spiral model of software development and enhancement," IEEE Computer, Vol. 21, No. 5, pp.61–72, 1988.

[7] G. Booch, "Object-oriented analysis and design with applications," Rational, 1994, 山城明宏, 井上勝博, 田中博明, 入江豊, 清水洋子, 小尾俊之 訳, "Booch法：オブジェクト指向分析と設計 第2版", アジソン・ウェスレイ・パブリッシャーズ・ジャパン, pp.57–63, 1979.

[8] J. G. Brodman and D. L. Johnson, "What small businesses and small organizations say about CMM," Proc. of ICSE-16, pp. 331–340, 1994.

[9] F. P. Brooks, Jr., "The mythical man-month: Essays on software engineering," Addison-wesley, 1975.

[10] E. Carmel, R. D. Whitaker, J. F. George, "Pd and joint application design: A transatlantic comparison," Communications of the ACM, Vol. 36, No. 4, pp. 40–48, 1993.

[11] G. Canfora, A. Cimitile, U. D. Carlini, "An extensible system for source code analysis," IEEE Trans. on Software Engineering, Vol.24, No.9, pp.721–740, 1998.

[12] L. Cleveland, "A program understanding support environment," IBM system journal, Vol.28, No.2, pp.324–344, 1989.

[13] 長谷川邦夫, "事例: 日本ユニシスにおける保守管理の実際", ソフトウェアプロジェクト管理 上, 菅野文友監修, pp.525–549, 1990.

[14] D. Hiebert, "EXUBERANT CTAGS," http: //home.hiwaay.net/ darren/ctags/, 1999.

[15] 平野一路, "遺産的 (Legacy) システムを対象にした進化的保守開発", 第 16 回ソフトウェア信頼性シンポジウム論文集, pp.16–19, 1996.

[16] 堀田美佐雄, "図解 会社の数字が読める本", 大和出版, pp.34–37, 1994.

[17] W. S. Humphrey, "Process maturity model," In J. J. Marciniak, editor, Encyclopedia of Software Engineering, pp.851–860, John Wiley and Sons, New York, 1994.

[18] W. S. Humphrey, "Managing the Software Process," Addison-Wesley Publishing, 1989.

[19] W. S. Humphrey, M. Kellner, "Software process modeling: Principles of entity process models," In Proc. of ICSE-11, pp. 331–342, 1989.

[20] D. Ross Jeffery, L. G. Votta, "Guest editor's spetial section introduction," IEEE Trans. on Software Engineering, Vol.25, No.4, pp.435-437, 1999.

[21] 神谷武志, "産・学・学会", 電子情報通信学会誌, Vol.81, No.2, 目次前, 1998.

[22] 嵩忠雄, 千原國宏, 鳥居宏次, 山本平一, 渡邊勝正, "大学院における情報処理教育の一つの取り組み", 情報処理, Vol.35, No.7, 情報処理学会, pp.593–598, 1994.

[23] 加藤久蔵, “会社経理の知識”, 日本経済新聞社, pp.73–79, 1967.

[24] M. Kellner, “Software process modeling example problem,” In Proc. of ISPW-6, pp.7–18, 1990.

[25] 岸田孝一, “プロセス成熟度に関する議論”, SEAMAIL, Vol. 9, No. 1, pp.35–39, 1994.

[26] J. L. Koldner, “Improving human decision making through case-based decision aiding,” AI Magazine, Vol. 12, No. 2, pp.52–66, 1991.

[27] G. Lenford, J. Myers, “Composite / structured design,” Litton Educational Publishing, 1978, 国友義久, 伊藤武夫 訳, “ソフトウェアの複合/構造化設計”, 近代科学社, pp.61–67, 1979.

[28] W. C. Lim, “Effects of reuse on quality, productivity, and economics,” IEEE Software, Vol.11, No.5, pp.23–30, 1994.

[29] J. Martin, “Recommended Diagramming Standards for Analysts and Programmers,” Prentice-Hall, 1987.

[30] まつもとゆきひろ, “オブジェクト指向スクリプト言語 Ruby”, アスキー, 1999.

[31] 宮本勲, “ソフトウェア保守の管理”, TBS 出版会, p.15, 1984.

[32] 宮本勲, “ソフトウェアエンジニアリング：現状と展望,” TBS 出版会, p.4, 1982.

[33] 村井修造, “ダウンサイジングとオープンシステム，何がそうさせるのか？”, 情報処理, Vol. 34, No. 10, pp.1234–1239, 1993.

[34] 中島丈夫, “ライトサイジングへの道”, 情報処理, Vol. 34, No. 10, pp.1229–1234, 1993.

[35] 内藤裕幹, 飯田元, 松本健一, 鳥居宏次, "役割別工数投入計画のための見積もりモデルの提案", 情処ソフトウェア工学研報, 情報処理学会, Vol.96, No.6, pp.1–8, 1996.

[36] 中本幸一, 高田広章, 田丸喜一郎, "組込みシステム技術の現状と動向", 情報処理, Vol.38, No.10, 情報処理学会, pp.871–878, 1997.

[37] 日本電子工業振興協会, "日米ソフトウェアギャップに関する調査報告書", 日本電子工業振興協会, 94-計-6, p.41, 1994.

[38] 大須賀節雄,"日本のソフトウェア問題について", 情報処理, Vol.38, No.8, 情報処理学会, pp.669–681, 1997.

[39] M. Page-Jones, "Practical guide to structured system design second edition," Prentice-Hall, 1988, 久保未沙, 新谷勝利 訳, "構造化システム設計への実践的ガイド", 近代科学社, pp.57–142, 1991.

[40] J. Rumbaugh, "Object-oriented modeling and design," Prentice-Hall, 1992, 羽生田栄一 訳, "オブジェクト指向方法論 OMT", トッパン, pp.310–311, 371–397, 1992.

[41] M. Sakai, K. Matsumoto, K. Torii, "A new framework for improving software development process on small computer systems," International Journal of Software Engineering and Knowledge Engineering, vol.7, no.2, pp.171-184, 1997.

[42] 阪井 誠, 久保田 益史, 沖田 昌也, 松本 健一, 鳥居 宏次, "レガシーな組込みソフトウェアの改良支援ツール", 電子情報通信学会論文誌 (条件付採録).

[43] 阪井誠, 久保田益史, 沖田昌也, 松本健一, 鳥居宏次, "レガシーな組込みソフトウェア改良のための支援ツール", ソフトウェア・シンポジウム'99 論文集, pp.59–66, 1999.

[44] M. Sakai, K. Matsumoto, K. Torii, “A new framework for improving software development process on small computer systems,” Proceedings of International Symposium on Software Engineering for the Next Generation, pp.151-160, Feb. 5-7 1996.

[45] 阪井 誠, 松本 健一, 鳥居 宏次, “ダウンサイジング時代のプロセス改善モデル”, ソフトウェアシンポジウム’95 論文集, pp.131-140, 1995.

[46] 阪井誠, 松本健一, 鳥居宏次, “ソフトウェアプロセスの動的な変更部分の分析によるプロジェクト運営実践技術のモデル化とプロセス改善”, 情報処理学会研究会報告, Vol. 94, No. 43, pp. 41–48, 1994.

[47] H. M. Sneed, “Economics of software re-engineering,” Journal of software maintenance: Research and practice, Vol.3, No.3, pp.163–182, 1991.

[48] 高橋良英, 中村行宏, “流用率と流用部・改造部間のインタフェースの複雑さによるソフトウェアの保守品質評価方法,” 信学論, Vol.J80-D-I, No.5, pp.441–449, 1997.

[49] G. M. Weinberg, “Quality Software Management: Volume1 System Thinking,” Dorset House Publishing, 1992.

[50] 山中勝, “オープンソースとしてのリアルタイムシステム”, ソフトウェアシンポジウム 2000 論文集, pp.55–59, 2000.