SEAMAIL

Newsletter from Software Engineers Association

Volume 11, Number 7 1998

目　次

ソフトウェア技術者協会
Software Engineers Association

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飛躍的に増加し，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30社を数えます．支部は，東京以外に，関西，横浜，長野，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．１１，Ｎｏ．７　　１９９８年６月２０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　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

☆

ほんとうにひさしぶりの船便 SEAMAIL です.

☆☆

今年の前半は, 数としては, たいしたイベントはなかったのですが, 4 月の京都 ICSE 併設の Asia Pacific Forum の企画と運営で, 事務局兼編集部はかなりのエネルギーを消耗してしまいました.

☆☆☆

それが終ってやれやれという間もなく, ソフトウェア・シンポジウム'98. 今年は, 2 つの併設チュートリアルと 3 つの特別セミナーの申し込みを同時並列に受けつけるというクレージーな事態で, ようやくそれも峠を越したようなので, シンポジウム開幕までのわずかな暇を盗んで, 1 冊編集することができました.

☆☆☆☆

新谷さん, 大木さんのエッセイは, 前々号に載った玉井先生の論文 (もともとは昨年夏に情報処理学会の機関誌に掲載された大須賀先生の論文) をきっかけとするソフトウェア工学およびソフトウェア産業の現状についての議論です. 他の会員の方々もふるって御意見をお寄せください.

☆☆☆☆☆

インテックの清水さんからは, いま流行りの ERP と, 御自分が昔関係されたナショナル・プロジェクトでの経験を対比させた長編のエッセイを寄稿いただきました. これもまた, 技術の現状についての 1 つの御意見をあらわしていると感じます.

☆☆☆☆☆☆

最後に載せた英語の Paper (これも長編) は, 京都の Asia Pacific Forum で基調講演をお願いした国連大学ソフトウェア研究所の前所長 Bjorner 先生の原稿です. 欧州協同体の国際プロジェクト Esprit と同じようなことが, なぜアジア (アフリカまたは南アメリカ) ではできないのか? というのは多少われわれにとって耳の痛い指摘のように思われます.

☆☆☆☆☆☆☆

「情報処理」掲載の大須賀論文に触発されて ソフトウェア工学について最近思うこと

新谷 勝利
(日本アイ·ビー·エム)

1. はじめに

この小論は「情報処理」1997年8月号に記載された大須賀論説:「日本のソフトウェア問題について－現状分析と将来対策」に対する小生から大須賀教授へのメモとして，当初SEA幹事会のメーリングリストに転送したものです．転送した理由は，大須賀教授からはたった1行の返事(小生がコメントしたこと自体へのアクノレッジメント)しか返ってこなかったたからでした．大須賀教授とのダイアログができないとしても，内容的にSEAの友人たちと討議したいなと思ったのです．その後，残念ながら，期待したダイアログは，SEA幹事会のメーリングリストでも起こりませんでした．反応は二つ：Seamailに転載してはというお勧めと，玉井先生のSeamail, Vol.11, No.3への関連エッセイの発表でした．そこで，編集部からのお勧めにしたがい，若干の手直しをした形で，この小論をまとめることにしました．ぜひとも忌憚のない意見が当誌で聞けることを期待しております．

2. ソフトウェア問題に関係すると思われる二つの身近なエピソード

現在小生のいる部門で「アイディア募集」と称して，オープン·フォーラムをロータス·ノーツ上で実施していますが，参加者は，ほとんど入社して10年未満の若年層です．このフォーラムに管理者が本気かどうかは，管理者本人の参加の度合いで評価できると，小生はアジっているのですが，それを受けて反応した管理職は1人だけでした．

若者の熱心さには，心うたれるものがあります．と同時に，管理者層には失望しています．このフォーラムは，単なるアイディア募集ではなく，小生のいる部門が今後進むべき方向等に関しても，何らの制約なしに討議できるように設定されていました．

そこでの議論を通読したとき，明らかな変化として感じるのは，技術至上主義ではなくなったということです．もちろん，社内論文発表会で優秀賞をもらった技術をどう生かすかといった意見もありますが，使用者がどのようにコンピュータを使用するのか，どのような問題を解こうとしているのか，シナリオはどうか等に，多くの議論が振り向けられています．

技術者が使用者の要求をほんとうに理解しているかどうかは，古くて新しい問題です．もちろん，使用者自身がコンピューターによる問題解決をどう認識しているかも同様に問題でしょうが，開発者が自分のアイディアを説明するのに使用者側に立つ努力，使用者の問題解決シナリオを理解しようとする努力，自分の解法の理由を説明しようとする努力といったことがらに多くの議論がなされているという状況は，数年前に同種の試行を実施した時に，技術論がほとんどどであったことと比較すると，大きな変化だといってよいでしょう．

また，今回まったく新しく見られた現象として，経営層の数語の発言を受けて，それを具体的にどのようなソリュウションに結びつけるか，自分たちの能力，技術的蓄積がどのように活用できるか，それをどの顧客で検証できるか等の問題が，延々と討議されてもいました．若年層に期待できる新しいうねりのようなものを感じました．その反面，管理者層には問題を感じましたが....

弊社のコスト分析によれば，ソフトウェア製品のエンドーツーエンドのライフサイクル·コストの配分で，開発コストは約30%を占めるのみです．マーケティング，流通，その他が約70%です．ベンチマーキングをしたところ，マイクロソフト，HP等の成功例ではこの比率は20% vs 80%になっています．開発コストのうち，コードを作っている部分は20%弱，UTを含まないテストに約30%，管理に約6%，残りは仕様·設計で45%という平均的な数字があります．

ベンチマーキングによると，これらの数字そのものには特異性があるようには見えません．コスト配布の多いものから見直すとすれば，マクロ的には，あるい

はソフトウェア･ビジネス的には，開発以外からということになります．開発行為そのものに着目すれば，仕様･設計，次いでテストということになるのでしょうか．波及効果あるいは因果関係もありますから，単にある局面の数字のみを問題視するのは問題かもしれません．

ソフトウェア問題は今まで後者が主として着目され，それをカバーするものとしてソフトウェア工学が考えられてきたのですが，広い意味でのソフトウェア問題は，いわゆる開発以外の局面にありそうです．ソフトウェア工学が余りインパクトを持たなかったのは "30 vs 70" あるいは "20 vs 80" の小さい数字の分野を対象にしていたせいでしょうか．

3. ソフトウェア問題とは何か

ここで興味深いのは，「開発部門のみならず研究部門にも影響を及ぼしている」(情報処理，1997 年 8 月号，p.669．以降「」で引用するのは特に断らない限りすべて大須賀教授の論文なので，ページ数のみ記述します) という記述です．読み続ければ気づくことではありますが，使用者のいないソフトウェアはビジネス上存在できないものです．ソフトウェア問題は開発の前から，そして開発後の価値評価にも問題があると考えるべきではないでしょうか．

ここで「開発の前」というとき，それは，研究･開発の研究段階を意味するというより，「何を開発すべきか？」というマーケティング段階と考えるのが自然であるような気がします．特に「最近の市販のソフトウェアは米国製で占められ，日本勢の衰退が著しい」(p.669) ことが問題意識の出発点であるとするならば，ウィンドウズが PC の OS 世界を席巻したのは，必ずしも研究の成果ではないことを思うにつけ，ソフトウェア問題への視点は「それが使用されている場はどこ？」という視点で，「求められている事項に対応しているのか？」というものを特定しなければならないのではないでしょうか．

別ないい方をすれば，ソフトウェア技術者にソフトウェア･ビジネスの経営感覚を身につけるにはどうすればよいのかということでしょう．純粋な技術論の立場からの追求が「ソフトウェア問題」の解決に結びつくというのは，上記のエピソードをみても実際的ではないように思われます．

4. 現象面に現れた日米の相違点

米国のソフトウェア開発現場で，日本との違いに気づくことの一つに，COOP 制度があります．学生は，そこではすでに一人前のソフトウェア開発者です．この制度が，日本でも，就職選択時期が早まったとこと合わて，せ新聞紙上で最近話題にのぼっています．日本でなぜこの制度が採りいれられていないのかは，興味があるところです．実際の開発現場を早くから知ることにより，前節で述べたことが，大学においても理解されるのではないかと考えます．

「大学の関心分野と企業の関心分野が乖離」(p.670) ということの責任は，現場を見ようとしない大学にもその一端があるのではないでしょうか．日本の大学からの論文で，現場で何が行われ，何が問題で，どうしようとしているのかという調査に基ずくものは一例を知るのみです．INSP で検索すれば，海外では，数多くの論文がヒットします．小生のところにスタンフォード大学，シラキュース大学，エディンバラ大学から研究者が調査にきたことがあります．それが工学的かどうかは別にして MIT のクスマノ研究室では，継続的に博士論文のテーマになっています．

もちろん企業にも責任はあるのでしょうが，日本の大学における研究アプローチに何らかの問題はないのでしょうか．米国においては，企業はベンチマーキングにきわめて熱心です．「自分を競争相手と比べたとき，ほんとうはどのように位置づけられるのか？ 何を改善すればよいのか？」といったことにセンシティブです．

ソフトウェア問題を技術論に絞ったとしても，米国の大学においては，マーケットが何を求めているのか，現行の技術あるいは技術動向がそれに対してどう応えられるか，といったことがらについての理解が，日本の大学におけるそれよりも進んでいるように感じられます．だからこそ，リスクの高いベンチャー･ビジネスにも，大学が対応できるのではないでしょうか．「日本では大学は孤立し，大企業は新しいソフト開発の遅れを米国の大学やベンチャー企業との提携で取り戻そうとしている」(p.670) ことが問題視されるのであれば，その対象は企業ではなく，むしろ大学であろうと思われます．.

5. 立ち遅れの原因

いま，アジアにおけるソフトウェア先進国はといえば，多分インド，シンガポール，台湾，中国でしょう．共通点が一つあります．大学および企業の中心的人物

が例外なく米国の大学院教育を受けていることです．米国でも「コンピュータ·サイエンスは役に立っているか」という議論が学会誌で展開されていたことは記憶にありますが，大学における教育そのものが問題視されていたのではなかったと思います．常に，投資に対するリターンを評価するという米国の文化的·社会的側面にもとずく調査研究の延長線上の議論であったと考えられます．同じような視点から，「日本の大学は何をしているのか」という議論が必要なのではないでしょうか．弊社の大和研究所にかなりの人数のソフトウェア開発者がいるにもかかわらず，世間で話題になるようなソフトウェア製品をまだ産み出していないことが，私たちの反省点なのですが，そのような場においても，最近入社してくる学生はきわめて優秀で，かれらにまかせたソフトウェア製品はPC雑誌のヒットチャートにのるようになってきていますので，必ずしも，日本の大学教育にのみ問題があるとは考えてはいません．「日本では物に即した実験·観測には力を入れてきたが概念化·抽象化の訓練が欠如」(p.671) しているという指摘には同感します．こうした訓練を提供する場は，やはり大学においてでしょう．小生は，弊社がこれからのソフトウェア社会で生き残るべく，すべてのソフトウェア開発者を再教育するというプロジェクト (コロンビア大学，CMU，ワシントン大学等のコンピュータ·サイエンス担当教授がアドバイザリボードのメンバーでした) に1984年から1989年にかけて参加しましたが，その時の最重要課題は，いかにして「概念化·抽象化」の力をつけるかでした．Polyaの著作も教科書のうちの1冊でした．クリーン·ルームの手法は，実は「概念化·抽象化の成果」が小集団において共有できるというチーム開発の方法論であると認識されており，そのような観点で社内教育が展開されていました．

6. 現象の理解

「仮に次の技術の飛躍点を5～10年後とみなして，新しい技術を開発する必要がある」(p.674) というアプローチは，現在の役立ちませんし，きっと5～10年後にも役に立たないでしょう．いまから5～10年前の技術がいまの時点においてさえも実践的なものとして認識されなければ，その時になってのビジネスには間に合わないことになるでしょう．

ソフトウェア工学の世界でいまホットな技術論として認識されているオブジェクト指向は，約20年も前の1970年代後半のバーバラ·りスコフらの研究に始まると考えることができます．構造化プログラミングや構造化設計にしても，すべて一般に普及し実践されるまでには長い年月を必要としています．インターネットの利用が本格化した現在，この時間を短縮するために，テクノロジー·トランスファーの技術がさらに研究されるべきでしょう．そして，このテクノロジー·トランスファーに責任を持つべきは同じく大学ではないでしょうか．

7. 将来の開発体制

ブルックスの "No Silver Bullet" の論文を思い起こさざるをえません．

8. 結び

第5節で述べたように，小生は大学での教育に期待するところが大です．しかし，大須賀論文には「では大学は何をするのか」という記述がないところが気になります．もう一つ気になるのは，一昨年であったかに行なわれた「創造的ソフトウェア開発」プロジェクトの帰結です．通産省が実施したのが筋違いであったのかどうかは定かではありませんが，「研究評価機構」は，予算の戦略的配置から考えても，特に研究予算や人的制約のある大学では重要ではないかと思われます．

大須賀論文が情報処理学会誌でこのように大きく取り上げられたのは，そこで取り上げられている事項が，まさに，数多くのソフトウェア関係者が悩み，苦しんでいることであることからすれば，画期的なことだと思います．しかしながら，「では大学では何をするのか」という視点が欠けていることや，「技術論以外，例えばマーケティング」の議論が少ないのが，気になるところです．これらを含めて「ソフトウェア問題」が継続的に討議されることを期待します．

これからのソフトウェア工学

大木 幹雄
(日本工業大学)

1. 沈滞感の原因

Seamail vol.11 No.3 で，玉井さんが，最近のソフトウェア工学の沈滞感について述べておられた．最近のマイクロソフトやオラクル等の製品発表のスピードと比較すると，たしかに沈滞しているといわざるをえない．ソフトウェア工学の目指すところは，一口にソフトウェアのライフサイクルを通した作業効率の向上や，開発プロセスの実態やその特性を客観的に把握し制御すること等であるといえようが，Windows 上のパッケージソフトの大洪水の中にあって，ソフトウェア工学の意義や期待感もかつて程感じられなくなっている．

ソフトウェア工学の成果が活用できる場面も，ビジュアルなプログラミング環境を提供するツールや，クラスライブラリを利用して初心者でもそれなりのプログラムを簡単に作成できる状況では，少なくなっていることは間違いない．結局，現在のソフトウェア工学は管理技術が中心になってしまい，「おもしろくない」ということになる．

大学でソフトウェア工学の講座を持つ者として，もちろんこのような状況をよしとしているわけではない．ソフトウェアの計測や予測，あるいはプロセス管理の重要性は，メインフレームの時代から情報システム開発を手がけてきたものにとって，その重要性は肌身にしみて感じているし，パソコンベースのシステム開発の時代にあっても，方向性として間違っているとは思わない．

しかし，マルチメディアやらコンテンツやらですそ野が広がったパソコンベースのソフトウェア開発 (一昔前のソフトウェア技術者が夢としていたハードウェア環境や開発環境) が目の前に出現し，一昔前に比較して十倍以上の効率でソフトウェア開発できる環境があたりまえとなり，かつ，それがものすごいスピードで進化 (?) している状況にあって，ソフトウェア工学が十分対応できずにいることはたしかであるし，それが沈滞感の主たる原因であるようにに思える．

2. 打破を目指して

私としては前述のような沈滞感を打破するためには以下のような活動が大切ではないかと考える．

(1) 細分化された研究分野の統合

ソフトウェアの意味する対象の広がりに応じて，研究分野の細分化が進み，ソフトウェア工学には，前述のような工学のエッセンスたるべき計測性の問題しか残されていないように思われる．これでは，次世代のソフトウェアのパラダイム革命など生まれようがない．

種々の研究分野を縦断し統括する，たとえば，一時期流行した「未来学」的な要素も取り入れてゆくべきではないか．

(2) ソフトウェア教育工学分野の研究

専門家集団から素人集団によるソフトウェア開発へという流れが定着する中，Smalltalk が子どもにもできるプログラミング環境を一つの目標にしたのと同様に，素人でもできるシステム開発を達成する教育ツール/環境の研究が，一層望まれているのではないだろうか．最近のヘルプデスクに対するニーズの高まりも，このような状況を反映しているものと思える (SEA でも技術移転の問題が取り上げられた時期がありましたが)．

(3) インタフェース中心のソフトウェア設計理論の確立

従来のソフトウェア工学は，大まかにいえば，情報システム開発をスクラッチから開発することを前提に，システム構成あるいはソフトウェア構成のあり方を追求することが一つの目的であった．最近のように，パソコンをベースにした分散システムや，自動車，電化製品，事務機器等の組み込み型コンピュータ等のソフトウェアでは，コンポーネントの組み合わせやコミュニケーションのインタフェースを正しく設計できるような理論が必要となる．残念ながら，関係データベースの

正規化のようにきれいに整理されるところまで至っていない.

2. 実験授業を通した将来のソフトウェアの形態

ソフトウェア工学の沈滞感のもう一つの背景として, 次世代のソフトウェアの形態が見えてこないことも影響していよう. そこで, 限られた範囲の事例で恐縮であるが, 図式化したコンポーネントを結合し組み立てる Visual Smalltalk Workbench を利用した記述実験 (情報工学科 1 年生 150 名) の結果から, ソフトウェア開発環境の将来イメージについて, いまの学生がどのように感じているかを多少紹介してみたい.

もちろん, これはかれらの願望を述べただけのものであり, 詳細を詰めれば穴だらけではあるが, メインフレーム時代のソフトウェアしか頭に浮かばない者には, 時として新鮮である.

(1) コンポーネントが, バーチャルな 3 次元空間で, 家具や電気製品を部屋の中で配置するように, 自由に配置できコードで結合できるようなプログラム開発環境. 対象年齢や嗜好に応じてコンポーネントが変化するとさらによい.

(2) バーチャル空間で, 開発スタッフ全員が一つの空間にアクセスして, 共同して操作しながらプログラムができるようになっている開発環境.

(3) ゲーム感覚で, コンポーネントに役割や得点を与え, うまくゲットしたとき高得点を得られるような開発環境. こうすれば, 何度も試行錯誤しながら最終的なプログラムを組み立ててゆくプロセスが苦にならない.

(4) 最終的な結果のイメージを示し, ウイザード形式で途中で何を穴埋めすべきかという形の虫食いプログラムが出てくる. それを埋めてゆくだけで目的のソフトウェアができあがる「おまかせコース」の開発環境.

将来のソフトウェア工学の新たなテーマが, このような素人のアイデアをきっかけとして生まれ出てくることを期待したいが, 単なる戯言に終るかもしれない.

以上、ソフトウェア工学の沈滞感を払拭する方法を述べたいと思った当初の気持ちとは裏腹に, 結局まとまりのないものになってしまった感があるが, これをもって玉井論文へのコメントとさせていただきたい.

エッセイ的小論

仏教的システム開発論とERP
― もっと道具を作ろう！ ―

清水 求
インテック
(ERP研究推進フォーラム出向)

1. まえがき

物議を引き起こすようなタイトルをつけてしまったが，うさん臭い内容ではないかと思われる方が多いのでなかろうか．ご推察のとおりかもしれないが，しかし，本人はいたってまじめである．齢54にもなると，多少はうさん臭いものも平気で人前出せるようになる．SEAMAILへの投稿が少ないようであり，なにか議論のネタを提供したいと考えて投稿した．ここでは，何が仏教的であるかということは論じない．仏教的かそうでないかは読者に委ねる．SEAMAILへの投稿は初めてである．

2. 投稿の動機

筆者はいまERP研究推進フォーラムへ出向し，ERP(Enterprise Resource Planning)の啓蒙･普及を推進している．ERPについては，すでに多くの雑誌･新聞などで紹介されているので，特に紹介はしない．名前のとおり「統合的企業資源計画」とでも呼んでおこう．拡張解釈すれば企業の統合的情報処理システムの構築であり，多くの企業でいままでも構築されてきているので，ことさら新しいものではない．

ERPの実現にはERPパッケージが多く使われているが，それだけがすべてではない．パッケージを利用することが効果的であることはいうまでもない．ところが今日，日本においてはERPパッケージの利用にかなり躊躇しているようである．宣伝されているわりには効果が出ないのではないかという疑問が提示されているようである．日本における導入事例では，ERPパッケージの機能が業務処理を満足する度合いが低く，思った以上にコストがかかってしまったとかいったような問題が出ているようである．だんだん実態がわかるにつれ，ERP導入の検討が多くなりつつも，逆にERPパッケージの導入意欲を阻害していることも多くなっているように見受けられる．

ERP導入の成功要因には，先ずトップのリーダシップがあげられているが，筆者からみると，システム部門の力も相当影響している．トップのリータシップだけのせいにするのではなく，システム部門も自分たちの能力のレベルアップをはかることが，ERPパッケージに不足する機能を効果的に補完できることに，注意を払う必要があるだろう．そもそも，情報処理システムの実現は，企画，開発･運用･保守自体が，システムライフサイクルの視点でコストミニマムを追求すべきものである．ERPパッケージは道具の1つにすぎない．道具論については後述するが，複数のよい道具を使わないかぎりコストダウンは不可能である．

日本においては，道具を使うことに対して，いままでは消極的であったように思われる．ことに，基幹業務パッケージの導入ともなると，さらに消極的である．このような状況は，パッケージの提供者側にも問題があるが，ユーザをサポートするシステムインテグレータの能力レベルにも，大きな問題がある．ERPパッケージの機能不足により，パッケージの導入を見送り，従来のように手作りで開発しようという空気が強く感じられるようになってきている．パッケージの適合率が低ければ，手作りで開発することには異論はないが，問題はやり方である．品質をどのように満足させるかである．筆者は，道具を使わないかぎりよい品質の達成はできないと思っている．多くの企業で，道具の利用はあたり前のこととして検討されてきたであろう．しかし，現実には，ERPパッケージだけでなく，CASEの利用にも躊躇しているようでもある．このような状況は，日本の情報化において好ましいことではない．したがって，筆者の道具の開発経験を踏まえ，システム開発における道具の使い方を述べるものである．当然，道具は開発方法論でもある．

筆者が述べることは，格別新しいことでなく，30年前の思考と同じものである．30年前の思考を持ち出さなければならない状況は悲しいことである．コンピュータのハードウェアは驚異的な発展をしたが，ソフトウェア開発はあまり進歩していないように思われる．筆者の経験では，ある1つの処理プログラミング

に要する時間は，機械語とオブジェクト指向言語を使うのとそんなに大差がないように思うからである．システム開発がなぜ進歩していないかを考えてみると，どうも情報処理システム開発の本質が理解されているようで理解されていないのはないかと思う．したがって，あえて分かり切ったことについて述べるものである．

筆者の持論には非常識なきらいがあるかもしれない．だが，問題解決には常識も非常識もいないはずである．要は品質特性を満足する答えが得られればいよいからである．

ソフトウェア開発の現場では，人びとがまだ多くの悩みをもっており，答えを探しているようである．先日，ラジオの宗教番組で幸福の科学の大川隆法が「悩みの解決」について話をしていた．宗教家だから変わったことをいうのかと期待していたが，いったことは「悩んでいる暇があったら知識と経験を多く持ちなさい」というものであった．筆者もそのように思っている．少なくとも筆者は，自分の方法論がプロジェクトの企画段階から適用できるとすれば，開発における悩みの多くは解消されるはずだと信じている．われわれが高品質の情報処理のために開発をしなければならない課題がたくさんあることを考えれば，過去に解決されていることすら実践されていないプロジェクトがいかに多いことであろうか．

ISOソフトウェア品質特性

特性	副特性
機能性	合目的性 正確性 相互運用性 標準適合性 セキュリティ
信頼性	成熟性 障害許容性 回復性
使用性	理解性 習得性 運用性
効率性	時間効率性 資源効率性
保守性	解析性 変更性 安定性 試験性
移植性	環境適応性 設置性 規格適合性 置換性

図1 ソフトウェアの品質特性

筆者がここで述べることは，自分自身の道具の開発経験である．筆者の開発した道具は，結果として世間から支持されなかったものであり，ビジネスとして敗北したものである．したがって，筆者の論には説得力がないかもしれないが，それを承知で述べざるをえない状況であると思っている．それは，ERPパッケージをはじめとする道具類が必ずしもよい方向で使われているとは限らないからである．また，筆者の開発した道具は敗北したとはいえ，その道具と類似したものが，多くのERPパッケージシステム(道具を含んでいるので「システム」という言葉を付加した)の核となっている．多くの技術者にとってERPパッケージの理解は困難なようであるが，悩みを解決するには知識の幅を広げてもらいたいと思い，あえて思考のプロセスを含めて問いかけるものである．道具を効果的に使うためには思想が大切である．

日本でも何社かは，道具作りをして商売としている．外国製品と比べると苦戦を強いられている．道具は文化でもあるので，グローバルスタンダードに飲み込まれてしまうことは好ましいことではない．筆者のこの稿が，開発に努力されている方々へのささやかな応援歌となれば幸いである．

3. 道具論

どこの企業も道具を利用しようと考えてはいるが，実際にそれが進展していないケースが多いのではなかろうか．適合する道具がないので利用できないという理由もあるだろう．しかし，筆者は，何か道具に対しての考えちがいをしているところがあるのではないかと思う．

読者が経験されているように，道具には，それぞれ特性がある．たとえば，包丁は料理の道具であると同時に凶器にもなりうる．道具の本質は，長所と短所との両方をもつことである．したがって，1つの道具に多くを期待すること自体がそもそも何かおかしい．多機能な道具は，それを使いこなすためには，全部の機能を習得しなければならない．また，必ず無理がある．これは，図1の品質特性を考えれば明白であろう(ある特性を高めようとするとある特性が低下するという矛盾の存在)．日常の世界では，ある目的を達成するのに多くの道具が存在していることは理解されている．しかし，情報処理システムの開発の現場ではどうであろうか．日常の常識が飛んでしまっているように思われる．

筆者の感触であるが，情報処理システムの開発に携わっている人間は，なぜか完璧主義者が多いように思われる．1つの道具でなにもかにも満足しなければ気がすまないのようになっているのではなかろうか．た

だし現実には，道具を提供する側にも問題があることはいうまでもないので，道具の選択の責任者に全面的に非があるといっているのではない．しかし，手をこまねいている暇があれば，どうして補完する道具を作らないのであろうか．もちろん，道具作りをされている企業もある．しかし，大勢はそうでないように見受けられる．製造業における現場においては，道具に対する技術･技能が品質の決定的要因となっている．道具･工具･ジグ (治具) に対して並々ならぬ情熱を注いでいるが，このような例と対比すると，情報処理システム開発現場は異常ではないだろうか．

その異常さはどこから来ているのだろうか．日本においてもある一時期盛んに道具作りが行われた．しかし，今日ある種の挫折感があるようであり，気がついてみたら欧米に席巻されていたという状況である．筆者は，日本の状況には，情報処理システム開発現場において，道具開発に対して偏見があるように感じている．極論のきらいもあるが，その偏見とは，

1) 道具開発をするような人間は職人である．したがって，彼らの品質はいたって悪く，近代工業からみると職人に仕事をさせてはならない．
2) 道具作りは難しい．
3) 道具を作らなくても商売できる．

というような考えはないだろうか．もしこのような偏見があるとすれば，それはかなり根が深いので，それを除去するための説得は容易ではないと思われるが，簡単に反論しておこう．3) の反論は本稿の全体である．

3.1. 職人に対しての偏見はないか

先ず，職人に対しての偏見であるが，もしそのような考えがあれば，それは職人に対して失礼である．また，悪い職人しかみていないからである．本物の職人とはいかに合理的に物を作っているかを例にあげればきりがないが，ここに職人について定義した方がいらっしゃるので紹介しておこう．名前は失念したが，かって NHK ラジオの人生読本で，職人のお話があった．その方の定義は，正確ないい回しは記憶にないが，「職人とは，一瞬にして複雑な問題事象を単純にし，問題解決ができる者をいう」ということであったと思う．IC 技術など最先端の技術を支えているのはだれであろうか．道具である工作機械のもとを作るのはだれであろうか．

情報処理システム開発の現場において，複雑な事象を一瞬にして単純化できる人間はどれだけいるのであろうか．システムエンジニアやアナリストと称していながら職人にも及ばない人たちがどれだけいることであろうか．職人は問題解決を体でもって実現する者をいう．空論を振り回している人間は技術者･技能者と呼ばないのが常識である．職人は体によって合理化の 3 要素である 3S(単純化･標準化･専門化) を実現しているのである．

ソフトウェア開発において，品質特性を満足できないないような者は，技術者でもなければ職人でもない．職人とはその名前のとおり「プロフェッショナル」である．プロは厳しい訓練によって達成できることは周知であろう．もし，職人が「近代的」でないという見方があるとしたら，われわれの生産の基本は何であるかを知らないといえるだろう．並のサラリーマンよりも付加価値の高い仕事をしているのが，芸術家であり職人である．芸術家や職人とて商取引における社会通念を逸脱すれば，社会から淘汰されてしまう．よい作品を得るために，数年も待っているお客様も世の中にはいるのである．これは，極論にしても，1 つの職人論である．ただし，現実には情報処理システム開発現場において職人もどきの人が多くいることも事実である．われわれは何か勘違いをし，真の職人を育成してきていないのではなかろうか．

3.2. 道具作りは容易

次に道具作りは難しいということに対してであるが，道具作りが難しいというのは，基礎ができていない証拠である．職人は生産性をあげるために，また，できないものを可能にするために自ら道具を作る．基礎ができてしまえば，道具作りは容易にできるものである．読者の中は 4GL などプログラムジェネレータを作るのは大変だと思っていらっしゃる方が多いのではなかろうか．筆者のジェネレータ開発の経験からは技術と呼べるしろものではない (ジェネレータ作りに情熱をかけている方がいらっしゃったらゴメンナサイ．あくまでも筆者が開発したもののことです)．コンパイラ作りのように難しい技術は必要ない．情報処理技術者試験の 2 種に出てくるプログラミングアルゴリズムで十分である．

道具作りを難しくしてしまうのも，完璧主義者の影響かと思われる．先述のように 1 つの道具に何でもかんでも実現させようとすると極端に難しくなってしまう．当面実現したいもののために的を絞れば容易に実現できる．いまの ERP パッケージの中には，ERP パッケージそのものを理解するのに多大な努力をしな

ければならないものが多い．それはそれである面では仕方のないことでもあるが，それに対して疑問ももたなくてはならない(最近のERPパッケージは，当初のものに対して疑問をもち，改善されつつあるようである)．筆者は，決してそのようなパッケージを非難しているのではない．前述したように，道具は，利用者が開発者の思想と合致する場合に，多大な効果をもたらすからである．あくまでも管理量の問題として捉えればよい．すなわち，単純な多くの道具を管理･駆使するのと，多くの機能をもった１つの道具を駆使するのと，利用者にとってはどちらが効率がよいかという問題である．

一般の企業においては，他の会社の機能までも管理する必要はない．自分たちが目標とするシステムを実現できればそれでよいのである．自社だけを考えれば，管理情報はそんなに大きいものではない．したがって，明確に目的が定まっていれば，道具作りも簡単になる．また，仏教的システム開発論の基本思想であるが，ソフトウェアとは，その名前のとおり，状況の変化に対して柔軟に対応できるものではなくてはならず，要求機能のうちいまきっちりと決める必要はかならずしもないというものがあるということである．わけのわからないものは，人間の知恵にゲタを預けてしまえば問題解決が早くなる．道具による作業と人間による作業のバランスのとれた配分である．ただし，自動化されていない部分の情報管理が適切に行われなければないないことはいうまでもない．

ちなみに，特定企業の事務処理のプログラムジェネレータ(COBOL)は，どのくらいで作れるものであろうか．基礎の論理がわかってしまうと，１ヶ月も必要ないといえるだろう．与えられた問題に対して別の情報に転化する(ジェネレート)することは，100%の完璧性を期待しないとすれば，容易に実現できるものである．たとえば，ERPパッケージシステムにおいてリポジトリがあり，リポジトリの利用インタフェースが用意されていれば，われわれはその情報を道具によって加工することが容易にできるのである．

もし読者からの御要望があれば，筆者のジェネレータの処理方式を参考として掲載してもよい．

3.3. 道具の限界の設定

道具についての考え方で重要なことは，道具の機能に不足するところは，人間の知恵で解決できるということである．われわれが目にする現象としていろいろな特性がある(図２参照)．これらの特性については周知のことであろう．しかし，周知であっても現実は，飽和特性を無視した計画を立てる者がいる．たとえば，安全にかけるコストは，飽和領域に入れば，1%の安全率を高めるためにとてつもないコストがかかることになる．システム開発においても同様である．ある機能を実現するために飽和領域に入ってしまえば，際限なくプログラミング量が増えてしまう．図１の品質特性とは，システムとは，常に多くの要因･要素を考慮した全体最適のバランスの上で考えるべきものである．論理の完璧性を求めることだけがシステムではない．システムの中には不合理や不条理が包含されているものである．あくまでも，全体最適の視点で判断すべきものである．

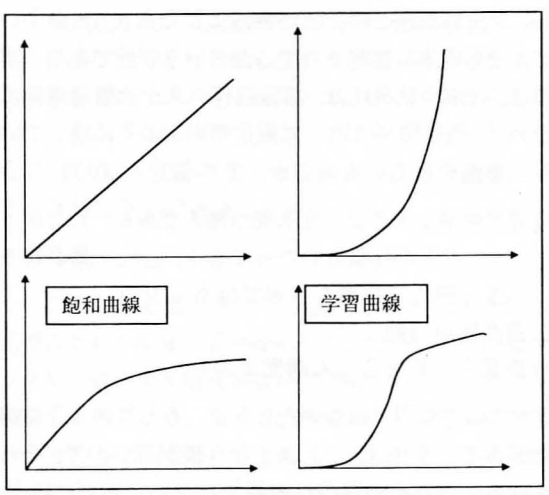

図２　事象の主な特性

情報処理技術の世界では，構造化(構造的)プログラミングは常識であろう．すなわち，構造化プログラミングは３つの基本形でプログラミングしようとするものであるが，この思想がシステム分析や設計で生かされていることが少ないのはなぜであろうか．いわゆるアナリストやシステムエンジニアの多くは，構造化プログラミングによって，複雑なものを単純にするという能力の養成が行われてきていないのではなかろうか．先述の職人の能力のように，単純化とは，実現のイメージがわからないかぎりできるものではないからである．道具作りが難しいといっている人は，情報処理システムの実現において最終的には何が重要であるかということの理解が不足しているのでなかろうか．

たとえば登山の例を考えてみれば(図３参照)，初めての山であっても，目標物が見えていて，地図があって，登山道具があれば，容易にいくつものルートを見

出し，命を落とすことなく目標物に到達することができる．もし，ガスがかかっていて，目標物が見えないとすれば，われわれはどのように目標物に到達できるのであろうか．超ベテランといわれる登山家が経験した山であっても，こんなところでと思われる場所で遭難死している例がある．つまり，道具作りが難しいといっているのは，目標物が見えていないか，道具作りのための知識と経験を持ち合わせていないからであろう．

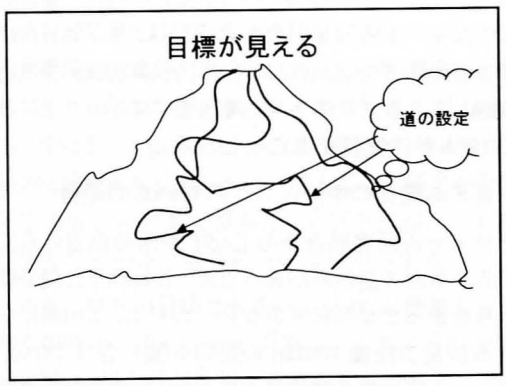

図3　目標の設定

4. パッケージが具備すべき条件 (道具作りの思考の過程)

ここから，パッケージをどのように作るべきかを，筆者の開発方法論を交えて述べる．パッケージや道具作りに興味のある方は参考にされたい．

日本における事務処理プログラミングの自動化研究は，1973年の通産省のモジュール研究組合から始まる．「モジュール」という言葉にみられるように，当時ハードウェアの製造においては「部品モジュール」というように，1つの機能単位をまとめて実現する状態に技術は進展しており，ハードウェアの設計思想にならい，ソフトウェアも部品化をはかり，生産性，品質向上を図ろうとするものであった．このプランは世界にさきがけた画期的なものであったが，その結果についてはほとんど評価されていない．モジュール研が目標としてものは，今日ようやく多くのERPパッケージで実現されるようになった．分野別の6つのモジュール研究組合のうち，事務処理モジュール研究組合の1研究員として携わった筆者からみれば，今日のERPパッケージをオブジェクト指向で作るようなものであったといえよう．「まじめに」といったら語弊があるかもしれないが，研究の機会を与えられた筆者にとっては，まじめに研究をし，パッケージ作りやシステム開発の方法論の基礎を確立することができたといってよいだろう．

当時はすでに多くのパッケージが出現しており，特に1971年に出たIBMのCOPICSは，統合情報処理システムという観点から画期的なものであった．ERPパッケージの条件に照らし合わせれば，ERPパッケージといえるだろう．COPICSを目の当たりにして，モジュール研究組合の限られた資源の中ではCOPICSのようなものを作ることは大変なことであり，いまさら何を研究すればよいのかというとまどいがあった．

パッケージは機能が多くなれば，それを制御する情報(たとえばパラメータ)も必然的に多くなってしまう．したがって，制御情報の管理が十分でないとパッケージを使った実際のシステム構築には，大変な工数がかかることが予想される．現実には，システムインテグレータの優劣は，いかにパッケージの制御情報に熟知しているかとうことになっている．パッケージの機能が実現したいシステムとかなり適合するのであれば，パッケージの利用効果が非常に高いものであるが，適合率が低い場合にはどのようになるかという根本的な問題が存在する．この根本的な問題をどのように解決すればよいのかという課題に突き当たる．

企業の業務を分析してみると，どの企業も他の企業の機能と同じではない．たとえば原価管理においては，先入れ先出し法か後入れ先出し法などいくつかの方式があるが，1企業にとっては1つだけの選択である．多機能な道具にみられるように，利用者にとって関係のない情報まで理解しなければならないのははなはだ迷惑である．今日においても，パッケージの大きな問題は，パッケージの機能に精通した人間が必要なことである．いわゆるパッケージ屋がいなければ，はなはなだ開発効率が悪くなる．パッケージ導入の重要な成功要因の1つとして，優秀なパッケージ屋をプロジェクトに取り込めるかどうかがある．パッケージ機能が多くなるに従い，ブラックボックスの部分が多くなり，実現への不安が大きくなってしまう．この不安のためは，ユーザが自らシステムを構築したいという誘惑にかられることになってしまう．

繰り返すが，パッケージ利用を否定しているいのではない．パッケージが利用できないときにどのようにすればよいかということを論じている．宇宙開発事業団が非難を受けながら自前のロケット開発をしたのも，ブラックボックスが存在する技術提携では，いろいろなデータをとることが試行錯誤でしかできなく，膨大な量の実験をしなければならないからである．試

行錯誤では，効率よく新しい機能や性能を開発することはムリである．同様にパッケージのモジュールの論理が不透明であれば，新しい機能を付加することは怖くてできない．特定企業において，自分たちの業務を把握しており，新しい業務設計もできるという前提があれば，あとはそれをシステムライフサイクルのコストミニマム (ソフトウェアの品質) という視点からどう作ればよいかという問題になる．

結論から先にいえば，今日のオブジェクト指向と合わせて CASE による開発ということになる．パッケージが CASE の中核であるリポジトリと 4GL によって実現されているとすれば，パッケージの機能拡張や，ユーザ自身が機能の追加開発 (アドオン) を行うことが比較的容易になる．また，開発されたものは，品質特性における，拡張性，移植性，保守性も格段に高くなる．このような観点から，モジュール研で筆者が得た結論は，まず，パッケージを開発する前に，その中核となる CASE が必要であるということであった．さらに CASE の重要機能であるプログラムジェネレータの開発が，将来の拡張性，移植性，保守性を保証するという観点と，ジェネレータがあれば，当面のプログラミングに対しても現世利益があるという観点でジェネレータの開発を行うこととした．

ジェネレータは，将来の拡張性，移植性，保守性を保証するという観点で，プロセッサの開発言語及び生成言語は，ANSI で標準化されている COBOL を使用すべきということで主張したが，それは責任者からは支持されなかった．今日，COPICS をはじめ，機種に依存したプログラミングはどのような運命をだどったかは明白であろう．システム開発は，部分最適ではなくライフサイクルにおける全体最適でなければならないという常識的なことさえ，当時は非常識とみられていたのである．全体最適ということは，品質特性やコンピュータの利用目的からみれば明白であったことであるが，日本の情報処理業界においては部分最適の思想が大勢として今日まで引き続いているといってよいだろう．ERP 普及の大きな課題の１つとして，先ずは全体最適ということを関係者にどう説得し理解してもらうかということがある．

筆者自身の本格的な CASE の実現は，モジュール研究組合の後続プロジェクトとして情報処理振興事業協会 (IPA) によるプログラム生産技術プロジェクトにおいて 1981 年に実現することとなる．

5. システム開発の中核としての CASE の開発

モジュール研究組合から CASE が生み出されるまでに約７年間の足踏みがあったが，それは単に開発費が足りなかったからにすぎない．ただし，この間にシステム開発方法論 PRIDE の影響を強く受けている．モジュール研究組合時代においては，開発方法論の理論的な根拠は漠然としたものであったが，PRIDE の導入により，CASE の理論的な根拠を強くしたといってよい．今日，RAD が話題になっているが，PRIDE を正しく解釈すれば，その本質は RAD に他ならない．システム開発に限らず，世の中で早く仕事をなしとげようとすると，ことわざ「急がば回れ」は真理であるようである．土台のできていないところには，ライフサイクルで通用するようなものは確立できない．

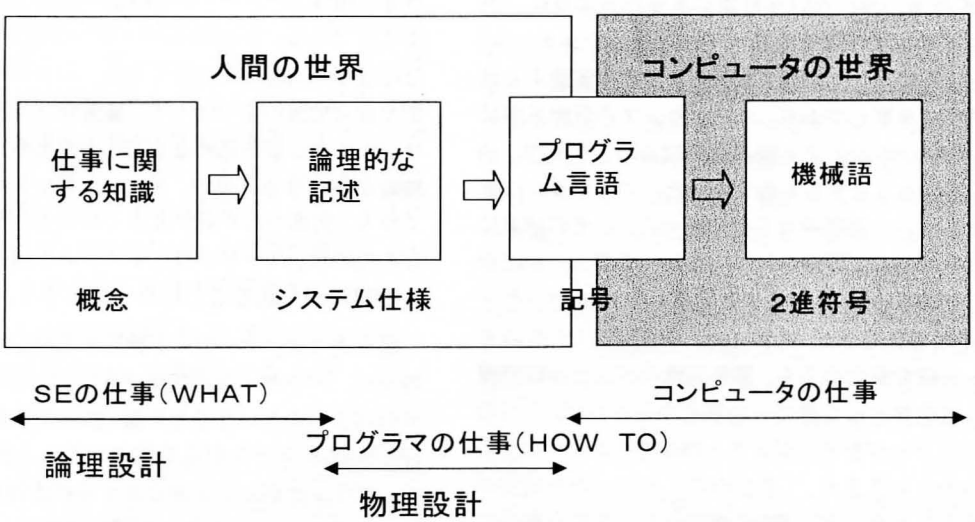

図4 システム開発プロセス

ちまたでは，PRIDE はウォータフォール型の方法論であるので，いまの時代には役に立たないという意見がほとんどである．筆者からみれば，そのような意見をいっている人は，本当に PRIDE を理解しているのかと首をかしげたくなる．ヒョッとして PRIDE の開発者自身がウォータフォール型でしか定義していないのかもしれない．現実のシステム開発ではウォータフォール型ではムリであるからである．少なくとも筆者はウォータフォール型では開発できなかった．もし方法論の運用に東洋的な思考をもち合わせていないとすれば，欧米のツールや方法論にみられるように硬直したものとなる．筆者は方法論もやはり道具であり，もっと自由に思考すべきものであるとの立場である．すなわち，筆者にとっては，無節操ではあるが，よい方法論であれば方法論は何でもよい．筆者が提唱する方法論とは，「無方法論」，「方円方法論」とも呼ぶべきものである．これについては後述する．

さて CASE の開発であるが，CASE の種類として上流 CASE と下流 CASE とがある．このような分類については，筆者は違和感を覚えるものである．システム開発をごく簡単に表すと図 4 のようになる．

図 4 は，南條優氏のもの (システム設計速修読本，学研) を引用 (一部加筆変更) しているが，簡単な図ほど本質の的を得ていると思っている．長い情報処理システム開発の歴史において，今日の日本の技術者不足や ERP パッケージにおいて欧米に遅れをとった原因は，いままでの指導者達がこの図をないがしろにしてきたことであろうと思っている．図 4 は SE の入門書に記載されているものであり，あまりにもあたりまえすぎるため，多くの指導者たちは本質的な問題を真剣に考えてこなかったのではなかろうか．

CASE にも見られるように，開発効率における最も大きな問題は，論理 (機能) 設計と物理 (プログラム) 設計の溝をどのように埋めるか (コミュニケーション問題) である．溝を埋めるには論理設計側 (SE) が物理設計をするか，物理設計側 (プログラマ) が機能を理解するしかない．もちろん共同作業ということにもなるが，どちらかが歩み寄らなければ，図 4 のプロセスは成立しない．通産省指導下の人材育成像においては，専門性ということでこのプロセスがズタズタに分断されており，それはそれで正しいことであるが，設計情報の連係をとることが前提条件となる．また，どのように連係をとったとしても非効率は避けられない．

図 4 からは，いわゆるスーパ SE の存在が期待されることがわかるだろう．コミュニケーションがいかに改善されたとしても，スーパ SE の存在と比較すれば劣ってしまう．現実には大規模開発は少数のスーパ SE ではできないことであり，多くの要員で作業をせざるを得ない．要員が 2 人以上のプロジェクトでは必ず軋轢があるため，スーパ SE がいるからといってうまくいくとは限らない．マネージメント問題が発生する．したがって，実際のプロジェクトでは，プロジェクトに携わる人たちの情報共有がなければ，はなはだ非効率なものになる．

その解決策として，グループウェアやネットワークコラボレーションが導入されているが，留意しなければならないことは，それらの導入前に，基礎としての情報共有を職能を踏まえて確立 (情報活用リテラシーのレベルアップ) しておかなければ，単に情報共有の道具が存在しているだけということになりかねない．

図 4 から明らかなように，仕事は少数精鋭で行うべきである．設計情報は，少数の人間が，できれば 1 人の人間が把握しなければトータル (全体) にはならない．もちろんリスク管理として人のペアシステムの導入は必須であるが，設計の本質論を考えれば，特に情報処理システムのように単品生産型においては，主設計者の存在はきわめて重要となる．

設計とは設計者の個性がモロに出るものである．矛盾する品質特性をどのようにバランスをとるかは，途中で複数人による議論の過程はあるにせよ，主設計者が意思決定するよりほかはない．したがって，主設計者は「プログラミングは知らない」ではすまされない．情報処理システム開発を建設業に類似させるなら，できあがり建物のイメージを描けないようでは，設計者として失格である．目的とする業務機能がプログラムとして実現されるためには，主設計者またはプロジェクトマネージャーが全体を把握するという視点で，適切に情報管理をする道具の利用は必須である．ERP パッケージを利用しようが，手作りであろうが，情報管理の道具として CASE の存在を必要とする．多くの CASE や ERP パッケージが出現している現在において，どれだけの企業で情報を一元管理する道具が使われているのであろうか．情報の一元管理による情報共有のシステムが確立していない限り，コスト増は避けられない．

筆者が開発した CASE は「PDL/1 (Progam Description Language One)」と呼ばれたが，これは，SE がプログラムを記述すべきだという観点に立ってい

る．情報処理システムの学習過程として，物理設計側(プログラマ)が機能を理解するような能力をもつ方が困難であるということがわかるだろう．もし，そのようなことができていたら，現在，業務を知らないSE(技術者)問題は発生していなかったはずである．そもそも業務を知らない技術者を生み出したことを問題にしなければならない．図4は，「業務を知らないでどうしてシステム開発ができるの？」といっているのである．SEがプログラムを記述すべきだといったが，SEが従来の言語(たとえばCOBOL)でそれ記述していたら身がもたないし意味がない．したがって，同じ処理(機能)を第3世代言語よりもはるかに少ない記述量で記述できる4GLが必要となる．しかも，その4GLは，システムの記述に近づいたものでなければならない．

理想的には機能の記述ができるものが欲しいのところであるが，現在のノイマン型コンピュータでかつ移植性を考えればCOBOLなどの第3世代言語への変換という処理方式が妥当であり，したがって，論理設計と物理設計とは別の構造とならざるを得ないため，機能の直接記述からのプログラム生成にはムリがある．また，そもそも機能の記述にも判断(if-then-else)構造はつきものであり，プログラムの基本構造と同じ構造の記述を避けてとおることはできない．「Program Description」は，「Problem Description」となる方が好ましいことであったが，ERPパッケージの開発構想までは当時としては開発費用の点で展開できなかったものである．

今日のERPパッケージの多くは，4GLを核としてモジュールが作成され，業務パラメータを制御するという方式である．現在のERPパッケージにおける制御情報(パラメータ)の管理はまだ不十分な点があると思われるが，今日のERPパッケージの姿は，1つの成り行きであろう．ただし，アドオンにおいてERPパッケージのもつ4GLがどの程度の記述ができるかを考えた場合に，プログラムの記述ができるレベルであろうと思われる．このようなレベルであっても，従来の第3世代言語を使うよりは，はるかに生産性が向上することはいうまでもない．したがって，ERPパッケージ導入は従来の開発よりも生産性が低いという論に対しては，4GLだけでも効果があるはずである．CASEをもつERPパッケージが単なる手作りよりもそんなに生産性が変わらないという論があるとすれば，道具に対する概念やリテラシーが確立されていないといえるだろう．

6. PDL/1の設計思想

くどいようだか，ここで述べることはPDL/1を理解して欲しいということではない．PDL/1はビジネスとして失敗したものである．しかし，開発方法論とツールをどのように考えるかということについて参考となればと思うものである．

6.1. プロセスによるコミュニケーションの一貫性(等価性)の確保

前述のように，PDL/1は，業務のわかったSEが，プログラム構造までを記述することを前提としている．SEによって記述された構造にもとづいて，詳細なアルコリズムやテストをプラグラマが行ってもよい．つまり，従来のように情報の連係を機械的(自動的)にできないような文書(ドキュメント)化を排除しようとするものである．図5のように，各プロセスにおいて文書が作成されたとすれば，文書の情報において等価性がなければ悲劇である．筆者は文書の不等価によりいかに多く泣かされたことか．文書は適切に保守されていない限りあてにはならない．ないよりはましだという考えもあるが，ない方がましという文書も存在している．そのみきわめが適切にできる能力が要請されていることになるが，この能力は高度な能力の部類にはいるのではなかろうか．

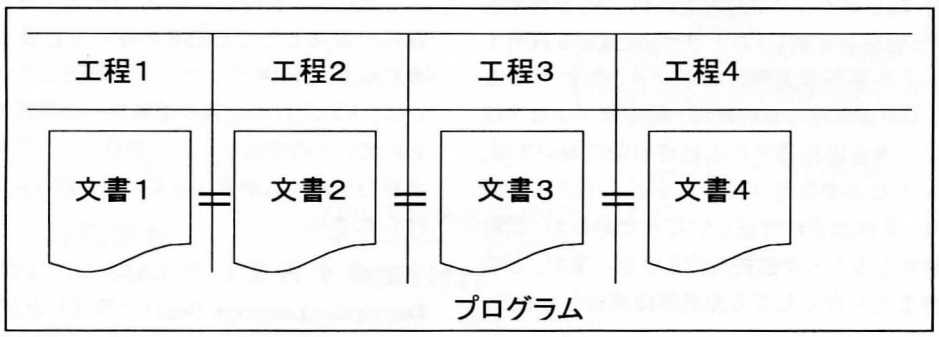

図5 設計内容の等価性の確保

文書の等価性ということから，CASE が上流と下流に分断されることは好ましいことではない．分断された CASE が個々において，情報が完璧に記述されることよりも，全工程を通じて情報の等価性をもつことが筆者は重要と考える．ERP パッケージにおいてもようやく一貫して記述できるものができているようである (実地に検証していない).

6.2. 機能記述の考慮

システムの記述とプログラムの記述という問題に触れるが，プログラム言語の構造から自動的にシステムの情報は取り出せないという問題がある．第 3 世代言語を使っている限りは，機能構造と物理構造とは異なったものにならざるをえなく，コメント欄にシステムの機能記述を行わないかぎり，プログラムからシステム (業務機能) の情報を取り出すことはできない．しかも，統合的な情報にするためには細工が必要である．

プログラムからシステムの情報を推測する最も簡単な方法は，データ名の命名に業務の名前を使うことである．たとえば，COBOL において,"Move A0001 To C095B" と記述するよりも，"Move 数量 In 販売ファイル To 数量 In 在庫ファイル" と記述する方が，システムの機能の推測が容易であることが理解できよう．本来，COBOL は，事務処理の自然言語をめざしたものであり，データ名に業務の単語を書けるようにしたものであったはずである．ところが，業務の言葉を記号化する方がよいと思ったのかどうかわからないが，データ名を記号で記述した有力な教科書もあり，記号化記述の方が入力が楽であったため，多くの者に影響を与えてきた (部分最適論者が指導者では困ったことである).

日本においては，COBOL の言語仕様においてデータ名の日本語記述は比較的早い時期から出現していたが，部分最適論者が影響力をもっていたのである．テキストのかな漢字変換や辞書による変換は容易な技術であるにもかかわらず，われわれにとって重要なことは，システムを管理しなければならないという発想が欠落していたのである．現在においても，外国製のツールの中には，日本語のデータ名を記述できないものがある．また，日本語のデータ名が記述できたとしても，単独で一意の名前にしなければならないものがある．この場合は英語のデータ名にしても識別できる一意のデータ名にしなければならず，機能名と物理名を一致させることはできない．

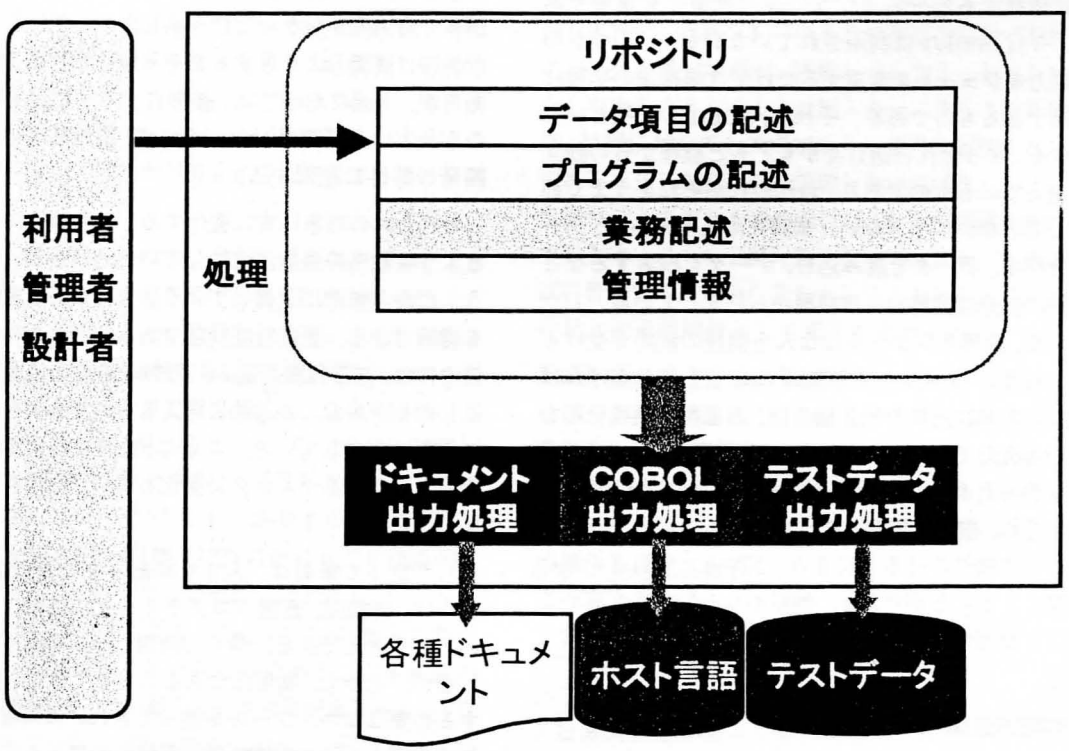

図 6 CASE(PDL/1)

COBOL のような第 3 世代言語は，ほとんど機能名で記述していくことが可能であるので，データベースを含め周辺の支援ツール類も機能名で記述できるようにして欲しいものである．現在出現している多くのツールは，データ名の記述が論理設計と物理設計とで分断されているものがほとんどであろう．

PDL/1 では，物理的記述は行なわざるをえないことを前提に，機能の論理を把握しやすいように，命令の表記及び命令の構造記述に配慮している．また，データリポジトリをもっているということは，テストデータの生成が，システムテストのできる状態で生成できることを可能とする．プログラムを解析してテストデータを生成するツールは多く提供されてきたが，意味のないデータがたくさん生成されたりしてあまり効果的といえない．PDL/1 では，システムテストデータ生成機能はパイロットとして作成してみたものであるが，このようなツールの出現が停滞しているように思われる．

6.3. 入出力インタフェースの抽象化

論理設計と物理設計の断絶を埋めるものとして，データ名の命名則が有効なツールであることを述べたが，モジュールとして入出力処理，つまりコンピュータに依存する部分のモジュールの抽象化も重要である．今日，PDL/1 は利用されていないが，いくつかの入出力モジュールを変更するだけで 3 層構造の処理に転換できるものである．それは，PDL/1 の仕様というよりも，第 3 世代言語自身がもともと機種からの独立を備えているものであり，われわれがそれをうまく利用していないにすぎない．業務を考えてみると，データを作る，データを読み込む，データを検索するなど動詞の部分は業務的には機種から独立している．したがって，論理モジュールはなんら機種の制約を受けることはない (ただし，高速処理の命令を使わなければ現実の応答に支障が出る場合は，ある程度機種依存をした入出力モジュールを使わなければならない場合があるかもしれないが，ほとんど必要ないだろう)．あらかじめ，機種に依存しそうなところは別のモジュールとして独立させるとにより，プロセッサ自身の寿命を長くすることができる．機種からの独立性を高めるたことを現実に生かすには，ジェネレータ技術が核となる．

7. 開発方法論

いままでに多くの開発方法論が出現し利用されてきている．ここではそれらの開発方法論のよし悪しについては言及しない．前述したように，筆者は，開発方法論も 1 つの道具であり，1 つの開発方法論にこだわることはしないという立場である．繰り返すが万能の道具 (1 つの方法論にたくさんの機能があるもの) は，その品質特性から必ず総合的な使い勝手は悪くなる．また道具の特性として，他人がよいというものであっても，自分の身体にはなじまないものもある．道具の利用において重要なことは，それを利用する前に，当事者が何をしたいか明確になっていることである．したいことがわからないのに道具が先にあるということはおかしなことである．ただし，道具の存在により道具を知るという効果があるので，道具を身の回りにおいておくことは否定しない．

自分のしたいことがわかれば (これが難しいのだが)，必要な道具を選択すればよい．道具がなければ作ればよい．ただし，複数の道具利用の場合の必須条件は，情報の連係がとれているかどうかということに注意を払うことである．現在のツールにおいて，異なったツール間において情報の連係をとることは結構困難かもしれない．したがって，現実的には 1 つのツールをベースにして，独自のツールで補完する方がよいかもしれない．

筆者は，外国製のツールについては，やたらに規約が多く自由度が少ないという感じをもっている．論理を厳密に展開しようとするとそうなってしまうわけであるが，上流においては，厳密にする必要があるのだろうかという疑問をもつ．それは，情報処理システム開発の特性に起因する．

われわれの対象は常に変化する．飽和特性にみられるように数%の機能は流動しているといってよいだろう．だから厳密に定義していく必要性があるという論も理解できる．要は程度問題である．われわれの事象の中には，ある程度普遍 (不変) 的なものと流動しているものとがある．普遍的に見えるものでも大きな流れの中では変化している．ことに，企業は経営環境の変化によってダイナミックに変化していかなければならない．

したがって筆者は，大きな枠組みさえ分析･設計できれば，詳細は，直接プログラミングに委ねた方がよいという考えである．多くの商用ツールの利用は否定しない．しかし，簡単にできることをことさら難しくする必要はない．ツールを使ってきれいに情報を整理してみても，その情報はだれが使うのであろうか．その情報が新しい業務革新の設計に反映されなければ

ならない．要は情報の管理･獲得能力の問題である．ERP の課題でもあるが，情報リテラシーのレベルアップを図られなければ，せっかくツールを導入してもツールの効果は小さくなる．

たとえば，いまどれだけの企業で業務の項目管理が行われているであろうか．前述したように，主設計者が全体を把握しなければならないという前提に立てば，数千項目にわたる業務項目は通常の人間の管理限界をこえているはずである．管理限界をこえたものは適切な判断が困難になる．したがって，ツールの利用は必須であろう．だが，ツールを利用したとしても，設計者は項目１つ１つの意味を理解しておかなければならない．

項目を管するというニーズに対しては，あえて高価なツールを買う必要はない (お金があれば買って下さい)．ツールがないからできないというのはやる気がない (または必要性の意味がわかっていない) だけである．RDB のような特別なツールを利用しなくてもできるものである (現在は DB 処理は RDB が主体になってしまったので，逆に RDB を使わない方が難しいかもしれないが)．目的が明確であれば，必要な情報収集は容易である．分析･調査をたくさんやって，アウトプットも多くできたが，実際のシステムはとんでもないものになってしまったという例がある．それは，調査のための調査になっており，結果として目的からはずれたものとなってしまったのである．当初は目的もはっきりしているが，人間はしばしば渦中に入ってしまうと目的や目標を見失ってしまうことがある．

さて，開発方法論について，先に「無方法論」,「方円方法論」という言葉を掲げたが，先述のように,「方法論なんかどうでもよいじゃないか」という無責任なものである．「無方法論」とは方法論が無いという意味ではない．多くの方法論を是認し，つまり無限に方法論がありそれを超越するということである．超越してしまえば無の状態である．「方円方法論」は「水は方円の器に従う」ということに由来する．つまり，システム開発対象物 (目標･目的) に合わせて方法論がしたがえばよいという考えである．いずれも，自由きままに，おおらかに，こだわりの心 (こころ) をもたないで対処したらどうかというものである．宮本武蔵の「五輪の書」に「守･破･離」という言葉があるらしい (自分で読んでいない) が,「無方法論」,「方円方法論」は「破･離」の状態である．どちらかというと「離」ともいえる．つまり，プロジェクトマネージャーが新しい方法論を設定すればよい．当然「守」の状態がクリアされていることが前提である．つまり基礎ができていなければ「破･離」はありえない．

人間の心は移ろいやすいものである．そもそも移ろいやすいものを固定しようとすること自体に無理がある．TQC はムリ･ムダ･ムラの排除が要点であるが，われわれはムリな状況で問題解決をしようとしてはいないだろうか．少しはムリをした方が緊張もありよい状態を生み出すことがあるが，ムリのし過ぎは失敗の大きな要因となる．われわれ人間は，こころの移ろいがありながら，ある面では基本は案外しっかりしているものである．また，昨日までの基本行動は，今日になって急に変えることはできない．したがって，移ろいの中にも普遍的な要素を多くもっている．

どのような方法論もある単位の仕事 (作業) から構成されている．その単位を目標の仕事に合わせて再構築すればよい．「よいとこ取り (BOB:Best Of Breed)」という言葉があるが，それとは異なる．計画の手法に PERT という大変よい手法があるが，これを応用すればよい．ただし，PERT もきれいに，厳密に記述しようとするとはなはだ厄介である．人間は，臨機応変ができる．パレート図にみられるように，大部分は少数の項目によって決定づけられる．大まかなことはきっちりと決め，詳細なものはある程度臨機応変にやればよい．生産性の問題は，詳細レベルできっちりと計画をしたからうまくゆくとは限らない．そもそも管理とは，目標に対してのアウプットの差異を制御することである．リアルタイムに管理しなければならないようなクリティカルなものを除けば，あまり詳細な管理にするとオーバヘッドが大きくなってしまう．おおざっぱに管理することから発生するリスクとオーバヘッドの量との選択問題として捉えた方がよい．

繰り返すが，方法論を必要としないということでない．１つの方法論にこだわらないということである．重要なことは，開発対象に合わせて適切に方法論を構築することである．方法論の選択の柔軟さは CASE ツールをもつことにより達成できる．

8. 要求仕様凍結問題

プロジェクトの失敗またはコスト増大問題として,「システムの要求が当初のものから変わったため」といういい訳がしばしば聞かれる．本来，システム開発とは要求が時々刻々と変わるものである．したがって，このような状況を解決するために方法論があったはずである．このような状況に対応できない方法論で

あったとすれば，それは欠陥のある方法論である．後述するが，方法論を支援する道具がないために方法論が悪いという見方もあるが，道具と方法論とは区別して考えてみる必要があるだろう．

ウォータフォール型開発では，工程のある段階で仕様の凍結ということを行う．ウォータフォール型開発でなくても仕様の凍結はどこかでしなければならない．お客様である発注者からみれば不満であろうが，これはこれで仕方のないことであろう．しかし，現実にはどうでもよいことまで仕様として決めようとする傾向がある．

前掲の特性図 (図 2) を思い出していただきたい．飽和状態の部分を決めようとすることはほとんど不可能に近い．発注者にとっては，いま，見えないところまで仕様を決めてくれという要求はいちばんつらいところである．画面や帳票の形式は大きな業務処理からみると本質ではない．ところがプログラミング作業という視点からは，大変工数のかかるところでもある．また，標準化という視点からは，ユーザインタフェースの統一化などは重要であるが，先に行うべきでもないだろう．現在の多くの開発は，論理設計と物理設計とが入り組んだ状態になっているため，物理設計を考えてはならない段階で物理設計を考え過ぎてしまう傾向がある．

ユーザインタフェースの検討の前にすべきことがある．物理設計に早く気を回しすぎるのは，プログラミングの有力なツールがないからともいえる．われわれは後からでもできることを，先の方に持ち込んでいないだろうか．先にやるべき重要なことは，基本処理に必要な情報 (データ項目) は何であるかということである．要求仕様を決めると称して工程の初期の段階で多くのムダな時間を費やしていないだろうか．

民家の建築の例をあげるなら，カーテンの種別·色·柄などを決めることは，引き渡し直前の住むことができる段階になってからも十分間に合うものである．むしろ，仮想のイメージでカーテンを選択するよりも，住む段階になって選択する方が適切なものを選択することができる．問題はカーテンが基本構造·コンセプトに影響を与えているところがないかを見極めておくことである．コストを厳密に見積もるためには，細かい仕様までも決めたいという考えは理解できるが，だからどうだともいえる．発注する側も，受注する側も欲をかいていてはお互いに損をするだけである．リスクとして予備費をみておけばよい．

人間は後から気づくことが多い．われわれのシステム作りにおいて，要求仕様の凍結ということで，カーテンの詳細まで決めさせているようなことはないだろうか．画面の形式や帳票などは後付でどうにでもなる部分である．利用者も本番データをみながら，つまり仮想ではなく現実のものを眼で見て考えること (ビジュアル設計) ができるので，オーバヘッドがかなり小さくなる．最近の ERP パッケージの CASE には，画面のインタフェースを各クライアントに合わせて変更できるものがある．

要求仕様をいい加減にしてよいという筆者の論は，支援ツール (分析·ジェネレータなど) があることを前提としている．もし，支援ツールがなければ作ればよい．前述のようにパッケージ仕様の必要な部分が公開されていれば，専用ツールの開発は容易である．ツールがないから，仕様をきっちりときめなければならないというのは，本末転倒である．他の製造業においては先述のようにツール導入やツール作りはあたりまえのことであり，そのような水準からみれば情報処理技術者は努力が足りないといえる．

システムが大きくなればなるほど複雑度は図 2 のように非線型で増す．普通の企業であればプログラムレベルまで必要な情報をまとめることは至難に近い．ムリな要求はよい結果を生み出さない．このためにモジュール設計という概念が必要になってくる．オブジェクト指向も然りである．モジュール設計·オブジェクト設計により単純化することができる．要求変更が基本構造に影響を与えないものであれば，過程において要求変更を取り込むことは小さなオーバヘッドである．だからといって，ユーザの要求を無定見で聞いてよいといっているのではない．あくまで全体最適という合理性のもとで要求を認めるというものである．部分最適に相当するところはユーザの責任が伴う EUC として実現すればよい．

ユーザが簡単に操ることができる EUC ツールがあれば，基幹の開発速度は速くなり，当然コストも下がるはずである．基幹システムの内容もスリムになり，今日の多くの品質特性問題もかなり軽減できるはずである．よい EUC ツールの存在がいかにシステム開発の負担を軽くしてくれるかを想像できかるかと思われる．繰り返しになるが，ツール作りは決して難しい技術ではいないので，われわれは自在な方法論の開発や採用ができるということである．また，ERP パッケージ本体または周辺としてツールが整備されつつあり，方法論の実装が格段に改善されている．

9. モジュール設計･オブジェクト設計

プログラムの基本構造は，連接，分岐，反復の構造であるが，論理設計においてもこのような構造にすることが重要である．つまり，論理設計にも合理化の3要素が必要であり，単純化の方法としてプログラムの基本構造の概念を取り込めばよい．総てが連接構造になればテストケースは2個である．連接構造の1箱の中が分岐構造であればテストケースは2個である．構造的プログラミングとは，全体の構造の構成要素を単純にし，テストケースを減らすことが本来の目的であった．コンピュータプログラミングの特徴は，時間が複雑な処理をしてくれるということである．瞬間瞬間は単純な処理しかしていない．しかし，1秒たち，2秒たちということにより複雑な処理をしてくれることである．われわれは論理を先のことまで考えて多くの組み合わせの論理を考える傾向があるのではなかろうか．繰り返しになるが，まず大きな構造を押さえ，基本を論理的に堅固にすることである．

単純化しなければならないしことはわかるが，どのように単純化したらいよいということに悩みをもつことがあるかもしれない．1つのヒントは，幾何の補助線のように共通の概念 (変数) を導入することである．言語翻訳は昔は，日英，日仏などのように直接的な翻訳処理であったが，今日は，日本語→共通語 (共通概念) →英語というように共通概念が単純化を促進する．このような観点でも，データの命名則や属性の管理が重要となる．

10. ツール評価に対する態度

コンピュータ情報処理における今日的な問題には，過去における言論が尾を引いている．ツールなどソフトウェア商品を売る立場からすると，他の商品について短所をことさら強調するきらいがある．本来はユーザは鑑識･識別眼をもたなければならないが，今日ではかなりの専門性が要求される．多くのユーザはまだ専門性が低いというのが実状であろう．また，ユーザにおいて影響力のある方はおおむねジェネラリストであり，部下などの第三者の言動によって判断することが多い．ツールには長所短所があり，人材の活用と同じように長所を引き出していくことが大切である．

たとえば，COBOL は生産性が低いという記事があちらこちらに載ると，たちまち COBOL の採用を手控えてしまう．ユーザの中には COBOL から新しい言語に切り替えて苦労されている方も多いであろう．情報処理学会誌などで論文に載るとさらにその影響力は大きいと思われる．COBOL が生産性が低いという論は，他にたとえるなら，「日本語は論理的な言語でない」という誤解と同じようなものである．われわれが思考するのは言葉によってである．もし，日本語が論理的表現が不十分であれば，日本の現在の学術･工業の繁栄もなく，とっくの昔に日本語は消滅していたはずである．

問題は言語を使う人間が論理的思考ができないだけのことである．COBOL プログラミングも同様であり，たまたま COBOL に精通しておらず，プログラミング構造の知識もない人がプログラミングをし，生産性が低い実績をもって低いといっているにすぎない．慣れていない言語は，著しく生産性を低下させる．科学者･技術者であれば，事実に忠実であることが基本姿勢であろうが，評価のような作業にさえ条件設定の適切さを欠いている．筆者は，あるジェネレータを COBOL で記述したが，いまはやりの言語よりは記述量は少なく，おそらく性能もよいし保守性･拡張性もよいのではなかろうかと思う．

ERP パッケージをはじめ道具の評価には，評価者が事実を把握することが大切である．また，商品の提供者は事実を情報公開すべきである．ユーザにとって厄介な問題は，ソフトウェアの許諾において，性能情報などは公開できないようになっているものがある．したがって，ユーザにおいては自己防衛をしなければならず，性能評価のための余分な工数と専門性をもった要員を手当しなければならないという問題をもつ．

さらに厄介な問題は，良心的な商品提供者であれば，商品の試用ということを認めてくれるが，海のものとも山のものともわからないものを購入しなければ評価できないという問題もある．情報が開示されていなく，表だってはユーザも守秘義務によって情報の交換ができない状況では，ユーザはどのように対処したらよいのであろうか．力のあるユーザは商品提供者に幾分かは圧力をかけることができるだろう．しかし，弱小のユーザにおいては，どうしようもない状態である．

ユーザが真によいシステムを作ろうという考えがあれば，他人まかせではなく自ら情報収集を行う態度が必要であろう．さらに，それを強力にするためには，ユーザが団結することが必要であろう．欧米に比べ，日本においてはユーザ会は，商品提供者の主導である．このような状況も日米の格差となっていることと認識すべきであろう．

ツールは長所･短所が同居である．歴史をみるなら，

情報開示がされていないクローズされている技術はその寿命は短い.

既存言語は, 技術転換の過渡期においては, 新しい転換の対応が遅れてしまい, 相対的に新しい言語に比べて機能を落としてしまう. 特に, パソコンにおける画面インタフェースは, 既存言語に大きな影響を与えたことは否定できない. プログラミングに力のあるユーザにおいては, 画面インタフェースを作成できたであろうが, 保守を考えると安定的に提供される商品を使わざるをえない. 道具論を考えると, 商品提供者が大きな市場をとった場合は, ある面では道具選択の幅がかなり狭くなっているという認識が必要である.

11. 技術者教育における道具の効用

PDL/1 は, 上述のように仏教的開発方法論を基本概念として開発したものである. 情報をリポジトリとして管理することによって, 設計情報を段階的に詳細化していく方法である. 欧米型のツールでは, 最近は少なくなったが昔は, 必要情報を完璧に入力しなければファイルに物理的に記録できないものがあった. これははなはだ不便である. 未知の情報が確定できるまでは時間のかかる場合が多い.

段階的な開発には, エラーがあっても記録できることや, 未知の情報に対して仮設定ができることが重要である. つまり最終段階の必要なときにきっちりとした情報になっていればよい. このような処理方法は, オーバヘッドが大きいように思われるが逆である. 未知のものを思考している方がはるかにムダな時間を費やしている場合が多い. 段階的詳細化は, 何回も情報を見ることになるので, 自然とレビューを繰り返していることになる. 情報が整えば一気にプログラムを生成 (ジェネレート) すればいよい.

素人が家の設計をするのに, 気にいったものになるには 3 回ほど経験しなければならないともいわれている. 筆者も同感である. 事前にいろいろなことを検討したつもりであるけれども, 現物ができてはじめて気がつくことが多い. 企業の情報処理システムの複雑さは民家の比ではない. いま, プロトタイピング手法が流行となっているが, それは必然である.

筆者は, コンピュータの処理性能がいまと比べると 3 桁以上も違う 30 年前からプロトタイピィング方式を採用してきている. 仮作成をし, 要求を確認するという方式である. 機械語やアセンブラではさすがにつらいが, コンパイラの出現によりプロトタイピィングも格段に容易になった. さらに, COPY 句はデータリポジトリであり, サブルーチンをオブジェクトととらえれば, クライアントサーバシステムの出現により技術転換のできなかった古い技術者も小さな努力で転換できるはずであった.

プログラマ 35 歳定年説とか, メインフレームとクライアントサーバの開発は異なるとか, 古い技術者の自信を喪失させるような物いいはだれが行ったのであろうか. 論理設計と物理設計の区分を明確にした上で, 両者を一体化させることにより, 総合的な能力を容易に育成できたはずである. 情報処理技術者試験 2 種のアルゴリズムの基礎ができていれば, コンピュータの基本原理が変わらない限り, 道具をうまく使うことにより, 古い技術者も当分は通用するはずである.

当事者が貴重な人材を育成する努力もせず, 時代の流れにちょっとついていけなくなっただけで切り捨ててしまうということは, 技術の責任者が行うべきことではなかったはずである. いま, ERP パッケージの世界でもそのようなことがあるかもしれない. ERP パッケージは確かに新しい技術かもしれない. しかし, 情報処理システムの開発の基本は３０年前と変わっていないはずである. ERP パッケージの多くの知識がなければシステム開発ができないというものではない.

新しい技術についていけなくなった技術者であっても, まじめに技術者として基礎を学ぼうという気があれば, かれらはツールを使うことにより存在を示すことができるはずである. そのツールも西洋的なものではおそらくつらいものがあるだろう. PDL/1 の狙いは, 技術者の技術能力転換を図る道具としての狙いが最も重要であった.

図 4 の基本的開発プロセスに戻るが, PDL/1 はコーダの排除をねらったものである. つまり, プログラマが業務を理解し, プログラミングするか, SE がプログラミングを理解してプログラミングするかということであり, １９８１年当時, コーダでさえプログラマと称して仕事をさせていたことへの警鐘でもあった. 本物のプログラマはいまの時代でもメシは食える. ますます本物のプログラマが求められている時代である.

PDL/1 は, 新入社員が COBOL を習得した段階で, 1ヶ月目に画面処理を含むプログラムを 20 本以上も単体テストを含めて完成させている実績がある. ある者は 2ヶ月目には 40 本以上の完成をしている. 従来方式であれば 1 本も完成させることはできなかった. こ

れは何を物語るものであろうか．2ヶ月目の新入社員にでもてきることが，経験のある古い技術者ができないことはない．ただし，人間は保守的防衛本能を打ち破って脱皮するには，強い動機付けやツールが必要である．1ヶ月に4本ほどしかプログラミングできなかった者が，20本ほどできるようになることは大きな動機づけであり現世利益(インセンティブ)となる．よい道具の効用は，仕事を楽しいものにしてくれることである．

12. むすび

PDL/1プロセッサには，いろいろな形式のデータを蓄積し，管理する論理が盛り込まれていた．このようなものを作ることは一見難しそうだが，アプリケーションの構築はそんなに難しいものではない．既知の要素技術を理解するまでにはちょっと時間がかかるが，だれか体系的に教えてくれる人がいれば1週間も必要ないだろう．

一般に日本人の情報処理技術者は多くの勉強しているようであるが，何か本質的なところをそれているような気がする．情報分析ツールにしろ，ジェネレータにしろ，基礎の論理を1つ1つ積み上げるだけである．決して難しいものではない．PDL/1の開発は，ユーザにとっても，システムインテグレータにとっても双方が利益になることを願って開発したものである．ERPパッケージの時代になり，また，次のパッケージが出た時代になってもだれかがつらい目にあうようでは，真の技術といえるだろうか．時代のどの段階においても道具を有効に活用することを考えない限り，だれかがつらい目にあうことは目にみえている．適度につらい目にあうことは，能力開発にとって好ましいことであるが，度を超さないようにしなければならない．

いま，日本製のERPパッケージをはじめとして多くのパッケージが出現しつつある．おそらく，外国製のパッケージには日本人の思想に合わないところがあるかもしれない．しかし，完璧なものを求めるより，現実的な対応をした方が得策である．ユーザ及びシステムインテグレータは，もっと柔軟な考えをもち，道具を活用することにより，さらには，自ら道具を作ることにより，多くの問題解決ができるということに注目をしてもらいたいと願うものである．幸い，今日のERPパッケージは，柔軟性を売り物にするようになってきた．道具に対する思考の幅がかなり広くなったように思われる．したがって，リスクはかなり小さくなっているはずである．ERPパッケージベンダーの1つであるBAAN社CEOヤン·バーン氏は，「われわれの競争相手はERPベンダーではなく“複雑”さなのだ」といっている．統合システムは，情報の連関が巨大なものになっていくだろう．われわれはその課題を解決するために，もっと道具を活用し，作らなければならない．

(なお，本稿は，1996年7月の情報処理学会ソフトウェア工学研究会サマー·ワークショップin立山の筆者の論文を大幅に加筆したものです.)

Issues in International Cooperative Research Why not Asian, African or Latin American 'Esprits'?*

Dines Bjørner, Technical University of Denmark†

ABSTRACT

Joint international research in software engineering stands low in contrast to joint research in computer science.

In this personal account of more than 25 years of experience in joint international research projects around or based on formal methods the author reviews a number of facets of past joint R&D projects. A proposal for a framework for regional collaborative research in software engineering: methods and technology is finally put forward.

An extensive, commented software engineering terminology is included. It can be referenced for terms not otherwise defined. An extensive bibliography gives useful references. An extensive index should likewise prove useful.

1 Introduction

This note briefly examines some of the issues of international research collaboration in software engineering.

In the main it seems that more resources, than for "one-on-one" computing science research, are needed in order to achieve useful results. Software engineering seeks practically useful results, so they require a large audience.

International collaboration in software engineering research received, it is claimed, a significant, positive boost, with the European Community's ESPRIT (European Strategic Programme in Information Technology) programmes, incl. the BRA (Basic Research Action). We will briefly review our own, mostly positive experience from our long and deep participation in this programme. And we will comment on what seems to be the current state of this programme.

The United Nations University established in 1992 an International Institute for Software Technology, UNU/IIST, located in Macau, but active on four continents: Asia, South America, (Eastern) Europe and Africa. We will briefly outline what we consider UNU/IIST's success in propagating research and software engineering methods to more than 20 developing countries and countries in transition.

The SEA (Software Engineering Association of Japan) has conducted a most commendable joint programme on sound software engineering practice by organising a number of international symposia in Far and South East Asia, notably China. UNU/IIST has been supported by these symposia. We very briefly review this SEA effort.

All this brings us to suggest that regions or continents, as in Europe, try establish local (regional) versions, i.e. adaptations of the ESPRIT/BRA programmes.

We finally asses the possibilities of success of such programmes.

2 Software Engineering vs. Computer Science Research

By software engineering we understand the practice of domain engineering, requirements engineering and software design.

By computer science we understand the study of and knowledge about the 'things' (data and pro-

*Invited presentation for a similarly titled panel at the Asia pacific Forum on Software Engineering Monday 20 April 1998, Kyoto, Japan — in connection with ICSE'98.

†Software Systems Section, Department of Information Technology, DK–2800 Lyngby, Denmark. Fax: ++ 45-45.88.45.30; E–Mail: db@it.dtu.dk; Web: http://www.it.dtu.dk/~db

cesses) that can reside inside computers (what they are).

How does software engineering relate to computer science? Well, through computing science, or as we shall here call it: programming methodology. By computing science, cum programming methodology we understand the study of and knowledge about how to construct these 'things' (how to build them).

Software engineering centers around programming methodology but incorporates a great many other facets, usually taken over from traditional engineering.

The problem is, however, that software engineering is not easily comparable to other engineering disciplines. There are many reasons for this:

- **Science, Engineering & Technology:**

  Engineering is the process of "walking" between science and technology, that is: of creating technology based on scientific insight, and, vice-versa, analysing technologies in order to ascertain their possible scientific value (e.g. UML, Java).

- **Computability vs. Mother Nature:**

  First we have that no laws of nature govern the properties (behaviour) etc. of the data and processes inside computers. They are instead subject to the mathematical laws of recursive function theory (meta-mathematics, mathematical logic).

- **A Different Experimental Science:**

  Secondly, since the inevitable experiments by the practitioners of computer science and programming methodology are not ruled by laws of natural sciences, these experiments are much more like those of mathematicians. But they are, especially for programming methodology researchers, different in that they are method and methodology oriented. The mathematician mostly wishes, risking a rather superficial statement, to capture new mathematics or exciting theories around older math. The programming methodology researcher is primarily — or ought primarily to be — interested in principles, techniques and tools of programming methods and relations between such.

  The (theoretical) computer scientist is much more like the mathematician: wishing to build new theoretical models of various computing aspects: type theories, concurrency, distribution, "knowledge representation", proof theories, etc.

  In my mind much too few researchers, everywhere, are concerned with programming methodology. In Europe we see England, with Oxford as a leading place, as a country where programming methodology stands very high on the agenda. France has in later years, where they had one of the strongest schools in theoretical computer science, also moved into formally based, or at least formally understood software engineering. In my opinion, quite a nice and good surprise. In Denmark, besides my own influence, and no doubt also due to some early directions taken by Peter Naur, we are also strong in programming methodology. In the US they are strong at the two ends of a perceived spectrum from theoretical computer science, at one end, to very practical, non-programming methodology oriented software engineering, at the other end — but not in programming methodology!

- **Method & Methodology:**

  The terms method and methodology are important in this paper.

  By a method we understand a number of principles for analysing, selecting and applying a number of techniques and tools in order efficiently to construct an efficient artifact (here software).

  By methodology we understand the study and knowledge about methods.

  Since there is no one method that will bring us safely through any reasonable sized software development project we have to rely on several methods: methods for the "grand state" design of a large, distributed system, methods for the "nitty gritty" design or real-time,

safety critical and electronic equipment embedded components of the larger system, and so on.

- **'Formal Methods':**

  A software development method may be characterised as formal if the following holds: (i) the specification and programming languages are formal (have a formal syntax, a precise formal mathematical semantics, and a formal, mathematical logical proof system); and (ii) there are sets of rules (i.e. design calculi) that help guide or support the developer in transforming abstract specifications to executable code. Thus, by a formal method we do not mean that the principles are formal, only that some of the techniques and tools are. One observes that we really mean: formal specification and calculation when we say formal method.

  Programming methodology researchers study both the informal and the formal aspects of formal methods. It is somehow easier to study and produce papers on the formal aspects: yet another formal specification language, yet another twist to the mathematical semantics or the proof system of such a language, etc. It is more difficult, it seems — and at least to this author, whose primary occupation it is — to combine the study of the formal aspects of so-called formal methods with the informal ones: the principles of specification, calculation, abstraction and of modelling.

- **Descriptions at All Levels:**

  There are no physical artifacts when a software developer is at work.There are descriptions and descriptions and descriptions and ...:

  - **Domain Descriptions:**

    In our mind, proper software development is based on, or first develops a proper description of the application domain — a description which is void of any reference to not-already existing computing (and IT) in the domain. Such a domain description is usually developed in stages of development: from gross assumptions ("big LIES") via lesser and lesser assumptions ("increasingly smaller Lies") to a domain description that captures "all that is relevant" at least to the next development stage of requirements (whereby the limit of all the lies is said to be the truth!). Usually we develop a domain description which covers a broader (sometimes and preferably much broader) spectrum than needed for the next, the requirements stage. The domain is relatively invariant.

  - **Requirements Descriptions:**

    From a proper domain description we develop, also in close interaction with the stake-holders of the domain, the requirements. Again a set of increasingly more precise descriptions. Terms of the requirements are terms of the domain and must therefore be precisely described in the domain description. The requirements description is void of any reference, in principle, as to how the desired (required) software is to be implemented. The requirements are relative fixed.

  - **Software Design:**

    Software design — like domain engineering and like requirements engineering — involve many steps of development (depending on the way in which these steps are carried out, referred to as refinement).

    * **Software Architecture:**
      Now, given the computing platform constraints, the software architecture describes, in committed ways, the external interfaces and the observable behaviour over these. That is: all the things that human users or other software (either already or to be elsewhere developed) can experience. Again descriptions transpire. The software architecture is much less fixed than the requirements.

    * **Program Organisation:**
      And we could go on: to program organisation descriptions which de-

scribe all internal interfaces — often imposed in order to exploit existing platforms, or to ensure efficiency, or to ensure security, etc., etc. The program organisation possibilities are legio!

* **Etc.**

The software developers are "creating" these descriptions, formally or informally — and if formally, then in a spectrum from systematic, via rigorous to formal. The software developer reasons about the text: proves — informally, model-theoretically or proof--theoretically, or checks (model-theoretically) — properties about the individual specifications (descriptions, texts) or about transitions (relations) between documents (such as correctness, validation, etc.).

Lest it should be overlooked, the author of this invited keynote paper, is of the strong opinion that:

- Much too little research in done in most countries in the area of programming methodology.
- Much too little research and transfer is done in many countries wrt. formal methods.
- Much too many so-called theoretical computer science researchers are wasting our time, their time and somebody's money in contributing to the above!

  Namely by not understanding that computer science is both a mathematical and an engineering discipline and they must alternate!

We owe our jobs in universities to the actual use of computers, to the actual production of both hardware and — to us, worldwide — of software.

3 Three International R&D Efforts

3.1 The European Community ESPRIT Programme

The ESPRIT[1] Programme, of which the 5'th three–four year Framework is now being initiated has changed the European research scene in computer science, programming methodology and software engineering. It has mostly benefited the software engineering, then the programming methodology, and — rather sparingly — the computer science communities, as could be expected.

ESPRIT projects were primarily concerned with pre-competitive R&D projects in IT, including Software.

3.1.1 The Main Programme I have myself had projects in the precursor to ESPRIT (the Multi--Annual [IT] Programme) and in ESPRIT. I was one of the instigators of the ESPRIT 415: RAISE, and the ESPRIT 5383: LaCoS projects. The former created the RAISE Method, the RAISE Specification Language, RSL, a successor to VDM (and hence also, really, VDM-SL), and the RAISE Tool Set. The latter project tested out RAISE on seven medium scale software industries in Europe. RAISE stands for *Rigorous Approach to Industrial Software Engineering.* LaCoS stands for *Large scale development of Computing Sysrems using formal methods.* I have taken part in other projects as well. Common to all these projects are (and were) that they involved at least one industry partner, at least two, legally unrelated entities from different community countries, and that often they had a large proportion of academic, typically university institute or department participation. Financing was 50% community coverage of standard costs plus overhead. For an industry it meant a 50% contribution if the project was in line with that industry's R&D plans. For a university the other 50% were our ordinary salaries plus the university overheads. Thus it meant that university groups often could hire several researchers to work almost exclusively of the ESPRIT projects. As European academics also staffed the ESPRIT boards and took part in the pre-, ongoing and post-

[1] ESPRIT: European Strategic Programme for Research in IT, Information Technology

evaluation it meant that we had a beneficial influence over topics and research methodologies.

Many were the initially uneven partners ("strange bed-fellows") that were brought together. As years went on partners became "in-bred", knew each other from beforehand and worked well together. In some instances one had to "sleep" with partners, i.e. endure such, which did little or no work. Because of differences in national accounting practices we all witnessed some rather creative such techniques.

On the whole I can recommend any region of the world to consider a transplant — a totally business process re-engineered version, however!

3.1.2 The Basic Research Actions The BRA (Basic Research Action) was a subpart of ESPRIT geared more to research than to actual prototype, pre-competitive product development.

I was involved with one such project: **ProCoS** for Provably Correct Systems. Instigated by Tony Hoare it had two phases of which I was the director of the first, very successful phase.

It brought leading programming methodology researchers together regularly from three different countries and six different universities. Although there were culture differences, nationally and scientifically (some were more computer science than programming methodology oriented, etc.), the research productivity was very high and probably much, much higher total than if the six had each been left to their own devices.

Through the external review process, often with scientists from other community countries than the partner countries, results were critically assessed and, when appropriate, rapidly disseminated.

I do not believe that any national R&D programme of the kind that ESPRIT enabled could be as successful. At least in my country I see such programmes being far too political with the result that they really do not produce much in terms of research, let alone new results.

3.1.3 Political Demands The ESPRIT and the ESPRIT/BRA programmes have now run for more than 12 years. On one side they have brought much scientific and technological progress, on another side they may not have brought Europe much farther away from what some European Commission staff may characterise as a US dominance in software.

The political demand, in all areas of ESPRIT have therefore been increasingly tuned to an agenda of significant impact on European IT by European Software, and a similar impact of European IT on the world.

When science and politics meet, science becomes the looser. When engineers and politicians gather nothing very good results.

3.1.4 An Assessment of ESPRIT + ESPRIT/BRA In the European Multi-Annual and in the ESPRIT programme projects (in which I was involved) we transferred our research into practical, engineering tools, but mostly methods and research results. Thus we were able to make a significant quantum step ahead, of a size that would have been unthinkable in a purely Danish technology or science support environment: (i) **Formal Definition of Ada** (jointly with notably Prof. Egidio Astesiano of Genova, Italy), (ii) **Ada Compiler Development**, (iii) **Formal Methods Assessment**, (iv) **RAISE** (jointly with notably STL [Standard Telephone Ltd., Harlow, UK]), (v) **LaCoS** (jointly with STL's successor: BNR [Bell Northern Research]), (vi) **ProCoS** (jointly with groups at Oxford Univ., Royal Holloway/London Univ., Kiel Univ., Oldenburg Univ., Århus univ.), etc. There is no doubt that were it not for these European Programmes formal methods might not have been as advanced as they actually are today — in Europe. Which brings me to also mention that for 10 years (1987–1998) the European Commission supported first VDM Europe, then its successor Formal Methods Europe (FME). VDM Europe/FME serves, still, to help industry be increasingly aware of formal methods and their tools. There is now a whole industry of formal methods consulting firms spread out over Northern Europe. I doubt it would have been there were it not for ESPRIT. And I am not exactly a fan of the EU!

So perhaps we have seen the good days of ESPRIT and ESPRIT/BRA in Europe. While it lasted it was good. Many closet engineers came out of the university cloak rooms, but much good research

also got a fine chance to be transferred into industry. What Europe, as a whole, lacked was the close relations that US universities, when among the best, have with US industry.

I seriously believe that these European IT cum CS+PM+SE programmes have helped and will continue to help European industry.

3.2 UNU/IIST

I was the first and founding UN Director of UNU/IIST. From its start 2. July 1992 and for exactly five years. As for our strategy and tactics, we took, guided also by my then Deputy, now the full, Director, Professor Zhou Chaochen, a research and post-graduate direction that owed much to the "spirit" in which we had seen several ESPRIT and especially ESPRIT/BRA projects flourish.

We established a number of joint university, industry and UNU/IIST R&D projects. In China, Vietnam, the Philippines, and elsewhere. With rather practical goals: software to support the dispatch of trains in China, software for the Ministry of Finance in Vietnam, models of and a foundation for software for a radio telephony system for The Philippines, an analytical basis for software for a generic manufacturing industry in The Philippines, China and Brazil, a formal basis for a multi-script processing system for Mongol (combined with any other language you may so wish), etc. In these projects Fellows from these and other countries would visit UNU/IIST for periods of at least 10 months, some more, to learn formal software engineering methods (PM) and to draft the most significant documents of their engineering life so far: extensive domain models of the application area, of requirements as derived from these, etc., on to software. RAISE [34, 35] became a focal point as it, as a method and with its specification languages, RSL, and tools, provided amply sufficient support for almost all the work. Where RAISE was insufficient we turned, naturally, to the Duration Calculi, see next!

In addition we established a training programme in computing science research around the scientific and programming techniques for software for real-time, embedded systems. Here the theoretical work centered around the continuous time, integrable interval temporal logics of the Duration Calculi [20, 24, 23, 18, 19, 39, 22, 40].

Finally we established a programme in which we trained post-graduate fellows, usually lecturers, in transferring our advanced computing science and software engineering courses to their local universities.

UNU/IIST was, and is, very successful in this endeavour. A five-year review commission set down by the previous rector of UNU, came, however, to a rather different conclusion: we shouldn't propagate formal methods since they were not generally accepted in industry. We should give courses in object oriented programming, Java and the like. Well, I am glad that the board of UNU/IIST had the good senses to see through this strangely put request. The current UNU/IIST is — so far — continuing the good work of the past UNU/IIST. But I trust that the new Director, Prof. Zhou Caochen, will bring needed renewal and new facets to UNU/IIST. *Why shouldn't we give the best we know to the world? Why withhold from developing countries a leading scientific and technological direction? Why give them "crap"?* We are glad that all the groups with whom we worked in more than 20 developing countries, besides being acutely aware of the situation wrt. possible 'formal methods' controversies, all fully and enthusiastically adopted our programme: from China to Argentina, from Russia to Indonesia, from Gabon to India!

People from Asian industry and, especially Chinese universities, often asked me: how big is UNU/IIST. It was as if they expected to hear: 50–70 staff, etc. So when I proudly replied ten times less, they wondered. The trick behind our unquestionable success in propagating sound software engineering methods was that all my staff were tuned to exactly the same melody: they basically all understood computer and computing science and software engineering in the same way, they basically all used formal techniques: specification and calculation, as a matter of fact. We did not believe in formal methods — we just use it, further researched it, and taught it! Small is not only beautiful, but small is also very effective! And with the right tools and techniques one can convincingly tackle very large scale

systems.

3.3 SEA's Outreach Programme

The SEA: Software Engineering Association of Japan is — seen in the international light of ACM and IEEE — a truly unique institution. Over the years the rank and file of SEA have steadfastly organised an amazingly refreshing series of software engineering (CASE and other technology) symposia in China, and I have been the lucky participant of several of these events: in Urumqi, in QuFu, in Kunming. My colleagues at UNU/IIST have now taken over my participation. These symposia — conducted at a liberating low-key level — brought together leading software engineering researchers from China and Japan, and, thanks to SEA's strong network, also featured top US software engineering researchers. Thanks to a open minded spirit many young Chinese and Japanese researchers first presented their papers at these symposia which also featured good, professional discussions.

We all owe a great deal to the opening up of international collaboration in software engineering research to Kouichi Kishida–sensei's steadfast guiding spirit, gentle mind, philosophical inclination and long-range foresight.

4 A Critique

Not all is 'rosy': perceived problems hinder internationalisation of software engineering research.

Which, then, are these problems? Some are:

- **Software Engineering & Technology Hypes**

  Engineering "walks the bridge" between science and technology — both ways: creating technology based on scientific insight; analysing ("industrial archaeology") some technologies in order to retrieve scientific values.

  Many so-called software engineering researchers are guided more by current commercial technologies than by any substance of scientific content. Hypes such as Object--orientedness are considered hypes by a scientific community which fails to see how these so-called technologies relate to serious scientific insight. As an example, most, if not all object-oriented "methods" fail to deliver abstraction, compositionality (refinement) and logical reasoning. That is: where software abstraction techniques has been seen as a major contributor to cleaner and clearer software developments and designs most object--oriented techniques do not provide for abstraction. Once an object-oriented technique has been applied and a document created, this technique does not, in general, allow for refinement: transforming easy-to-understand abstractions into descriptions for presumably efficient software. And finally no techniques are offered by any object-oriented technique (that I am aware of) for reasoning over object-oriented specifications (i.e. no means for proving properties). This puts object-orientedness significantly apart from the main stream of programming methodology, or is it vice-versa ? There may be wonderful insight in these object-oriented techniques, but is it not sad that they are primarily carried by such commercial, anti-science attitudes?

- **Formal Methods vs. Ad Hoc Methods**

  Tony Hoare has expressed:

  - **Maturity:** Use of a formal method is no longer an adventure; it is becoming routine.
  - **Convergence:** The choice of a formal method or tool is no longer controversial; they are chosen in relation to their purpose and they are increasingly used in effective combination.
  - **Cumulative progress:** Promise of yet further benefit is obtained by accumulation of tools, libraries, theories, and case studies based on the work of scientists from many schools which were earlier considered as competitors.

  In [38] Anthony Hall lists and dispels the following seven formal method "Myths":

  1. *Formal Methods can Guarantee that Software is Perfect*

2. *Formal Methods are all about Program Proving*
3. *Formal Methods are only Useful for Safety-Critical Systems*
4. *Formal Methods Require highly trained Mathematicians*
5. *Formal Methods Increase the Cost of Development*
6. *Formal Methods are Unacceptable to Users*
7. *Formal Methods are Not Used on Real, Large-Scale Software*

In [15] Jonathan P. Bowen and Michael G. Hinchey continue dispelling formal method myths:

8. *Formal Methods Delay the Development Process*
9. *Formal Methods are Not Supported by Tools*
10. *Formal Methods mean Forsaking Traditional Engineering Design Methods*
11. *Formal Methods only Apply to Software*
12. *Formal Methods are Not Required*
13. *Formal Methods are Not Supported*
14. *Formal Methods people always use Formal Methods*

And in [16] Jonathan P. Bowen and Michael G. Hinchey suggests ten rules of formal methods software engineering conduct:

I. *Thou shalt choose an appropriate notation*

II. *Thou shalt formalise but not over-formalise*

III. *Thou shalt estimate costs*

IV. *Thou shalt shall have a formal methods guru on call*

V. *Thou shalt not abandon thy traditional development methods*

VI. *Thou shalt document sufficiently*

VII. *Thou shalt not compromise thy quality standards*

VIII. *Thou shalt not be dogmatic*

IX. *Thou shalt test, test, and test again*

X. *Thou shalt reuse*

Despite the above well-worded de--mystifications based on rather extensive evidence we still see parts of the software engineering world being very slow on the uptake.

Ad hoc ways of developing software, i.e. ways in which no proper use is made of available mathematics, still abound. In Denmark we say: OK, that's fine, then our software industry will indeed have at least one competitive edge when the real fight for software house survival sets in.

Examples of formal specification techniques and design calculi are:

B/AMN	[1, 49, 61]
CafeOBJ	[25, 54, 53, 31]
CoFI/CASL	[57, 58, 52, 17]
Duration Calculi	[20, 18]
Larch	[36, 37]
RAISE	[34, 35]
STeP/React	[50, 51]
VDM	[7, 47, 29]
Z	[59, 60]

- **Software Technology vs. Methodology**

Programming methodologists, in addition to research into principles for selecting and applying techniques for the analysis and synthesis of domain, requirements and software design models, also need create tools. Typically such tools center around tools for the creation, analysis and transformation of domain, requirements and software models.

Many so-called software engineering researchers waste their time, in my opinion, on building "tools" that do not have a proper methodological or scientific foundation. When they try show this "gadgetry" to me — "demo", I believe they call it — I have a very hard time understanding what is going on. While they are frantically hammering away at the keyboard and nervously clicking the mouse

etc., while they are excusing that the system reacts strangely just now, I am all the while asking for the one, two or at most three key concepts that best characterise their system. After half an hour the session usually is abandoned.

- **Pre-competitive Research:**

  Software engineering research typically requires a large-scale experiment around the development of actual, realistic software systems. This often necessitates commercial software house partners in addition to academic research groups. These commercial enterprises usually have a hard time understanding that one can indeed single out facets of the systems they wish to develop, facets that are "pre-competitive", that is: Whose openly inspectable development and the general awareness of this development does not hinder these enterprises in retaining their commercial rights nor their competitive edge. The European ESPRIT projects were very successful in this.

- **Confusion of Academic and Industrial Work**

  In many developing countries this is a serious problem and it is hard to tackle. But it is also a problem in most industrialised countries. The problem is that otherwise clever and scientifically serious research groups waste their talent on projects that ought better be carried out in industry. In developing countries such is often the case because the academy institutes and the university departments are extremely hard pressed to earn money. In industrialised countries some researchers — due to "fear of flying" — practice, under university protection, development which had better be done in industry. These "frustrated engineers"[2] attract the attention of the media and are often the "darlings" of their university chancellors. But no scientific progress is made!

  I pity those developing and other countries which tolerate this serious diversion, away from science, and in clear, but in all ways unfair and uneven competition with industry. The universities and academies are shooting themselves in the foot: they will loose years of research insight. They will miss generations of sufficiently well-educated candidates.

- **Cultural Differences?**

  In my 20 years of lecturing around the world, outside the Judean/Hellenic/Christian world, I have found little evidence pointing in the direction that cultural: philosophical, religious, ethnic and other such factors materially influence basic attitudes towards the conception of software, research into its scientific foundations and its methodologies — in any but at most superficial ways. One such is treated next!

  But I may just be kidding myself. At least it could be a topic for discussion at the Asia Pacific Forum on Software Engineering Workshop, April 20–21, 1998 at ICSE'98.

- **One Country's Seeming Dominance?**

  The US, for better or worse, dominates the software engineering research scene, at least when seen from other places than Europe. Some years ago one could say, with some truth to it, that the disciplines, the topics, of theoretical computer science (complexity theory, etc.) and practical software engineering were the two main characteristics of US research in our area. It is only recently that US universities (etc.) have made significant contributions to programming methodology — except for significant research groups at CLInc. (the no longer existing Computational Logic Inc., Austin, Texas), SRI International (Menlo Park), Digital Equipment Corporations SRC (Systems Research Centre, Palo Alto), Stanford Univ. and CMU (Carnegie Mellon Univ.). There is something refreshing about US research in our area, but I seriously doubt its "hard nosed" attitude: *must have conclusive, experimental proof of superiority of formal methods over ad hoc techniques.* It seems that no allowance is given for the individual developers' intellectual satisfaction, feeling of elegance and sense of joy of conducting software development using formal techniques.

[2] Just like the many "frustrated mathematicians" who practice rather esoteric, some would say useless, theoretical computer science work

Although [38, 15] stands basically irrefutable many refuse to listen.

But the US state of affairs is rapidly changing.

5 Proposals

So what is our proposal. It is not formulated as a specific programme, only as an advice:

- **Advice #1:** *Intra– & Intercontinental Joint R&D*

  To boost local university research, and to make that research relevant to local industry while securing international relevance, create a number of "Path-finder" R&D projects that will bring university, academy and industry groups together around a common set of three objectives:

  1. New Software Technology for Industry
  2. New Software Methodology
  3. Increased Production of Professional Software Engineers

  Seek possible support from The World Bank *info***Dev** programme, from UNDP, ADB (Asian Development Bank) and others. Involve UNU/IIST in any such effort.

- **Advice #2:** *"Formal Methods"*

  To make a difference, and you'd better admit and accept it, base any such "path-finder" project on the simultaneously use of and research into further improved formal software development techniques (formal specification and design calculi) and tools.

  There are enough, and there will remain enough, software engineers that will try — in vain — to "fight" for the use of ad hoc methods. So to make a real difference join the real professional endeavour for responsible, trustworthy methods!

- **Advice #3:** *Infrastructure System Software*

  Let the "path-finder" projects evolve around:

  – **Infrastructure Systems Support**

    Examples of such infrastructure systems are:

    – *Transport Systems:*
      * *Railways* [9, 14, 21, 28, 13]
      * *Air Traffic* [5]
      * *Metropolitan Transport* [62, 55, 27]
      * *Shipping*
      * *&c.*
    – *Airline Business* [4]
    – *Manufacturing Systems:*
      * *The Market* [2, 45, 3, 44, 46]
      * *"Beyond CIM"* [32, 33]
      * *— including Robotics!* [8, 30]
    – *Government Administrative Systems*
      * *Ministry of Finance* [26]
      * *Ministry of Social Welfare*
      * *Health Care Systems*
      * *&c.*
    – *Financial Service Industry* [12, 48, 6]
      * *Banks*
      * *Insurance*
      * *Securities*
      * *Portfolio Management*
      * *Check &c. Clearing*
      * *&c.*
    – *Strategic, Tactical & Operational Resource Management:* [10, 56]
    – *&c.*

    To make sure that you understand what I am driving at, let me elaborate. The example is that of Railways.

    What is needed here, as in all the other examples, is a thorough understanding of the application domain void of any reference to requirements, let alone desired information technology. For railways this means, we believe, that domain models must be painstakingly established, models that cover "the entire railway spectrum": from the railway net with their lines and stations, with their rail units (linear, switches, crossovers, etc.), with the signalling that changes the states of paths through units and train routes, etc., via trains,

the movement of trains across the net, time--tables, scheduled and rescheduled traffic, including accidents, to passenger and freight services, rolling stock monitoring & control, marshaling, and net development: rail and signal equipment maintenance, new such, etc. The etcetera includes models of railway system rules & regulations, staff and user behaviour, economics, performance, etc. (again!). In other words: a rather sizable "chunk" of the domain. It can be done, and it must be done if we are to have any trust in any future software for even a tiny bit of railway system functioning. Also: there is no way to accomplish the goals, anyway near, other than using formal (combined, of course, with informal, synopsis and narrative) specification!

- **Advice #4:** *Domain/Requirements/Software*

  The projects (thus) should each cover the following research and development spectrum:

  - **Domain Engineering**
  - **Requirements Engineering**
  - **Software Architecture Design**
  - **Program (Structure) Organisation**

    and
  - **Formal Methods–based Tools (CASEs)**

- **Advice #5:** *Problem Frames*

  Software engineering is too broad a term. Professionalism can be achieved primarily for a well-delineated subset of applications. Such subsets could be linked directly to Michael Jackson's concept of problem frames [42, 43].

  We therefore suggest that an international research effort register a number of otherwise distinct consortia, say each working on a particular infrastructure support system, but such that across the consortia a non-trivial number of problem frames are represented:

  - Reactive system frame
  - Information system frame
  - Connection frame
  - Workpiece frame
  - Transaction processing frame
  - Decision support system frame
  - *&c.*

  The aim is to contribute, in a systematic way, to specialised frame-oriented methods: principles, techniques and tools.

- **Advice #6:** *"Pre–competitive"*

  Choose "pre–competitive" projects!

  Now it is easier to see that the above suggested class of joint R&D projects allow for much joint R&D of a nature that does not infringe upon competitive and proprietary aspects of participating commercial enterprises.

6 Summary

A plea has been made for medium-scale, 3–5 partner, 12–20 person international projects covering at least two countries of a region and at least one commercial and competitive software house — all as a means to further sharpen research into techniques in software engineering and into bases for new software systems. The former so as to result in the production of increasingly professional software engineers. The latter so as to result in the competitive survival of local software houses.

A plea has been rather personally and forcefully advanced that such projects be based on the increasingly accepted paradigms of formal methods and that they focus on the full spectrum of software engineering: from domains, via requirements, over software architectures to program structures (program organisation).

A World Congress on Formal Methods

The reader is kindly invited to take part, actively by submitting a paper (or two!), or "passively" by also participating in:

- **World Congress on Formal Methods**

  FM'99: : Toulouse, France, 20–24 September 1999

FM'99 is co-sponsored by ACM, AMAST, EATCS, ETAPS, EU (European Union), FME (Formal Methods Europe), IEEE Computer Society, IFIP, and many other societies.

FM'99 will feature a (i) Technical Symposium, (ii) a Tools Fair & Applications Demo Forum, (iii) a set of Users Group Meetings, and (iv) a set of Industry Tutorials (Formal Methods for Railways, Formal Methods for Telecommunications, Formal Methods for Avionics, Formal Methods for Hardware, etc.).

Program committee chairpersons of the Technical Symposium are Profs. Jeannette Wing, CMU (wing@cs.cmu.edu), and Jim Woodcock, Oxford (jim.woodcock@comlab.ox.ac.uk). The Symposium is planned around up to ten parallel sessions with a total of more than 150 papers and some 10 invited speakers. It will be the premier event in the now mature field of formal methods.

"Surf" to:

`http://www.it.dtu.dk/~db/fm99/cfp.ps`

for regular information on this congress.

B Software Engineering Terminology

B.1 Special Terminology

The wording of many of the definitions of this report may sound dogmatic. Prudent reflection will soon reveal that it is merely a set of reasonable and useful delineations.

1. Software Development:

   To us software development consists of three major components: domain engineering, requirements engineering and software design. Together they form software engineering.

   > Discussion: *This is a somewhat "bureaucratic" characterisation. Namely one given in terms of its "way of being handled" — who does it, rather than what it does!*

   Therefore: Software Development aims at constructing software — or as we shall later "enlarge" it: machines. It does it by also constructing models of the domain in which the software will reside, the requirements that the software must satisfy, etc. The present report will deal with the processes of software development.

2. Systems vs. Software Engineering:

   Perhaps the term 'software engineering' is too restrictive. Since any implementation of especially a larger software system entails procurement also of hardware, development will also include configuration and acquisition of hardware components. That larger concept: the development, procurement, installation, performance tuning, operation and disposal of computing systems (hardware $\oplus$ software) is therefore what we mean by systems engineering. Thus software engineering is part of systems engineering.

   > Discussion: *As eloquently pointed out by Michael Jackson [42] the term software engineering is probably much too broad a term, or it should be understood as a class term. As such it covers a set of specialised software engineer(ing specialtie)s. Mechanical engineering stands for rather separate groups of for example automotive, heat/water/ventilation, hydrological, nautical, aero-nautical, and many other engineering specialities. Software engineering is still far from having identified suitably specialised such groups — except perhaps for compiler designers. We refer to item 12 (page 20) for hints at what such groups might be.*

3. Linguistic Notions:

   (a) Descriptions & Documents:

   All stages of software development results in descriptions and documents. The two terms are almost synonymous: description refer to the semantic content of the syntactic document. We describe and document domains, requirements, software architectures, program organisations, etc. We sometimes also, again synonymously, refer to these descriptions as

Definitions (as f.ex. for a domain model or a requirements model), sometimes as Specifications (as f.ex. for a software architecture model), yes even as Designs (as f.ex. for a program organisation model).

software engineering management takes the syntactic, document view of development; whereas programming takes the semantic, description view.

(b) Concordant Documents:

A set of documents, spanning the spectrum of descriptions of domains, requirements, software architectures, program organisations, etc., form a set of *concordant* descriptions, and within each of these we may also need alternative, complementary descriptions — which form another set of *concordant* descriptions.

Two or more documents are said to be concordant wrt. each other if they all purport to present descriptions of basically the same thing — but each emphasising different, but related aspects.

We shall later introduce pragmatic notions of perspectives, facets, aspects and views. These represent equivalence classes of concordant documents.

(c) The Informal Languages of Indications, Options and Actions:

As pointed out by Jackson [43] the informal language of domain descriptions is indicative: "what there is", that of requirements descriptions is optative: "what there should be", and that of software design descriptions is imperative: "do this, do that — how to do it!".

(d) Descriptive and Prescriptive Theories:

We could also use the terms descriptive and prescriptive theories in lieu of indicative and optative descriptions.

(e) The Formal Languages of descriptions:

In contrast, the languages of formal descriptions are mathematical, and in mathematics we cannot distinguish between indicative, optative and imperative moods. Such distinctions are meta-linguistic, but necessary. Similarly with the various equivalence classes of concordant documents: perspectives, facets, aspects and views.

(f) Description Techniques:

We refer to Jackson [42, 43]: "Phenomenology — *recognising* and *capturing the significant elementary phenomena* of the subject of interest (domain, requirements, software) *and their relationships. Say as much as is necessary, with perfect clarity, but no more. ... Choose* and *express abstractions and generalisations* formally *in order* necessarily *to bring* an *informal reality under intellectual control.*"

Constituent techniques [43] are those of:

- `Designations:`
  That is: system identification. Establishing the informal relationship between real world phenomena and their description identifiers.
- `Definitions:`
  The definition of concepts based on real world phenomena.
- `Refutable Assertions:`
  The usually axiomatic expression of real world properties.

4. Machine:

The aim of software development is to create software. That software is to function on some hardware. Together we call the executing software ⊕ hardware for the machine. The machine is, in future, to serve in the (future) domain as part of the (future) system.

Since domain engineering and requirements engineering aim at descriptions that may eventually lead to procurement of both software and hardware we shall refer to software development leading to a machine.

5. Domain Concepts:

Two approaches seem current in today's 'domain engineering': one which takes its departure point in model–oriented, Mathematical Semantics specification work (and which again basically represents the 'Algorithmic' school), and one which takes its departure point in

knowledge engineering — an outgrowth from AI and Expert Systems. The latter speaks of Ontologies. For now we focus on the former approach.

(a) Domain = System ⊕ Environment ⊕ Stake-holders:

By domain we roughly understand an area of human or other activity. We "divide" the fomain into system, environment and stakeholder. All are part of a perceived world.

Discussion: *Examples of domains are: railways, air traffic, road transport, or shipping of a region; a manufacturing industry with its consumers, suppliers, producers and traders; a ministry of finance's taxation, budget and treasury divisions as manifested through government, state, provincial and city offices and their functions; the financial service industry, or just one enterprise in such an industry (a bank, an insurance company, a securities broker, or a combination of these); etcetera.*
Since we are developing software packages that serve in these domains it is important that the software developers are presented with, or themselves help develop precise descriptions (models, see later) of these domains.
Our argument here parallels that given for compiler development: we must first know the syntax and semantics of the (source, target and implementation) languages involved.

(b) System:

By system we understand a part of the domain. The system is typically an enterprise. Once the machine has been installed in the system then it becomes a part of a new domain wrt. future software development.

Discussion: *A railway System consists of the railway net (lines, stations, signalling, etc.), the rolling stock (locos, passenger waggons, freight cars, etc.) and trains, the time tables and train journey plans, etc. A description of the railway domain must make precise the structure and components of the railway systems as well as all the behaviours it may exhibit.*
Identification of the system is an art.
Please note that when we speak of a system we do not refer to a computing system.

(c) Environment:

By environment we understand that part of the perceived world which interacts with the system. Thus the system complement wrt. "the perceived world", i.e. the environment, together with the system and stakeholder makes up the domain of interest.

Discussion: *The Environment of an air traffic system includes the weather (the meteorology) and the topology of the geographical areas flown over.*
Identification of the Environment is an art.
Since the Environment interacts with the System (and hence potentially with the Machine to be built) it is indispensable that we describe (incl. formally model) that part of the Environment which interacts.

(d) Stakeholder = Clients ⊕ Customers ⊕ Staff:

By stakeholder we mean any of the many kinds of people that have some form of "interest" in the (delivered) machine: enterprise owners, managers, operators and customers of the enterprise: within the system or in the environment.

Discussion: *Stakeholders of a ministry of finance include government ministers, ministry staff and tax payers,*
Identifying all relevant stakeholders is an art.
Since also they interact with the System (and hence potentially with the Machine to be built) it is indispensable that we describe (incl. formally model) possible stakeholder interactions with the System.

(e) Client:

By client we understand the legal entity which procures the machine to be developed. The client is one of the stakeholders, and must be considered a main representative of the system.

Discussion: *A financial enterprise Client is usually the appropriate level executive who specifically contracts some software to serve in the enterprise.*

(f) Staff:

By staff we understand people who are employed in, or by, the system: who works for it, manages, operates and services the system. staff are a major category of stakeholders.

(g) Customer:

By customer we understand the legal entities (people, companies), within the system, who enter into economic contracts with the the client: buys products and/or services from the client, etc. customers form another main category of stakeholders: outside the system, but within the domain.

Discussion: *We have identified important components of a domain. The software engineers — in collaboration with domain stakeholders — face the further tasks of specifically identifying the exact components to be considered for a given Domain. That 'identification' is still an art: requires experience and cannot be settled before preliminary modelling experiments have been concluded.*

(h) Domain Engineering
= Recognition
↔ Capture
↔ Model
↔ Analysis
↔ Theory:

Domain Engineering, through the processes of domain acquisition and domain modelling, establishes models of the fomain. A domain model is — in principle — void of any reference to the machine, and strives to describe (i.e. explain) the fomain as it is. domain analysis investigates the domain model with a view towards establishing a domain theory. The aim of a domain theory is to express laws of the fomain.

Discussion: *The Domain Engineer could be a special version of a Software Engineer — one who could be specially trained both as a Software Engineer, in general, and as a "Domain Expert", in particular.*

(i) Domain Recognition:

System identification is an art! To recognise which are the important phenomena in the domain, and which phenomena are not (important) is not a mechanistic "thing".

(j) Domain Capture
= Acquisition
↔ Modelling:

Discussion: *We make a distinction between the "soft" processes of domain acquisition: linguistic and other interaction with stakeholders, and domain modelling: the "hard" processes of writing down, in both informal and formal notations, the domain model.*

The domain capture process, when actually carried out, often becomes confused with the subsequent requirements capture process. It is often difficult for some stakeholders and for some developers, to make the distinction. It is an aim of this report to advocate that there is a crucial distinction and that much can be gained from keeping the two activities separate. They need not be kept apart in time. They may indeed be pursued concurrently, but their concerns, techniques and documentation need be kept strictly separate.

(k) Ontology:

What we call domain models some researchers call ontology — almost!

In the 'Enterprise Integration and in the 'Information Systems communities ontology means: "formal description of entities and their properties". Ontological analysis is applied to modelling the domain of (manufacturing) enterprises and such systems (typically management systems) whose implementation is typically database oriented.

(l) Domain Model:

By a Domain Model we understand an abstraction of the Domain.

Discussion: *Usually we expect a Domain Model, i.e. a Description of the Domain to be presented both informally and formally.*

The informal Description typically consists of a Synopsis which summarises the Model, a Terminology which for every professional term of the Domain defines that term, and a Narrative which — in a readable style — describes how the terms otherwise relate. The formal Model is then expressed in some formal specification language and can be subject to Calculations using a Design Calculi of that notation. The model thus presents the syntax, semantics and, possibly also, the pragmatics of terms of the Domain. Not the syntax and semantics of the professional language spoken by Staff of the Domain System, but just the crucial terms.

(m) Domain Modelling Techniques

Domain modelling usually proceeds by constructing a partial specification (type space, functions and axioms) for each of a number of domain perspectives and similarly one for each domain facet.

(n) Description Technology:

Crucial concepts in domain modelling include:

- system identification,
- i.e. enumeration of designations [43],
- formulation of definitions and
- expression of possibly refutable assertions.

The latter typically in the form of constraints on types and functions.

(o) Domain Perspective:

Domain perspectives reflect the conception of the domain business as seen by various stake-holders.

(p) Domain Facet:

Domain facets reflect some more 'technical, pragmatic decomposition' of the domain together with a 'separation of concerns'. Specification typically proceeds from intrinsic facets, via support technology facets and rules & regulations facets to staff facets, etc.

(q) Domain Model Analysis:

By Domain Analysis we understand informal and formal analyses of the Domain and of the resulting Model — whether informal or formal.

Discussion: *The purposes of the analyses can be to ascertain*

whether a component and/or its behaviour qualifies as a component (etc.) of the Domain, and for such included components analyses may reveal Model properties not immediately recognised as properties of the Domain. Note the distinction being made here: the Domain as it exists "out there", and the Model as an abstraction thereof and which "exists" on the (electronic) "paper" upon which the Model is represented.

(r) Domain Theory:

The purpose of Domain Analysis is to also establish a Theory of the Domain, or rather: of the Models purported to represent the Domain!

Discussion: *Examples of theorems in a theory of railways could be: (1) (Kirschhoffs law for trains:) "Over a suitably chosen time interval (say 24 hours) the number of trains arriving at any station, minus the number of trains taken out of service at that station, plus the number of trains put into service at that station, equals the number of trains leaving that station"; (2) (God doesn't play dice:) "Two trains moving down a line cannot suddenly change place"; (3) (No Ghost Trains) "If at two times 'close to each other' (say seconds apart) a train has been observed on the railway net, then that train is on the railway net somewhere between the two original observation positions at any time between the two original observation times". Etc.*
Failure to record essential theorems may result in disastrously erroneous software.

Ability to identify and establish appropriate theorems is an art and takes years!

(s) Domain Model Validation

An informal process whereby informal and formal specification parts are related and where these again are related to the "real world" domain (system identification)

6. Requirements Concepts:

Requirements, as we have seen, form a bridge between the larger Domain and the "narrower" software which is to serve in the Domain.

(a) Requirements = System ⊕ Interface ⊕ Machine:

Requirements issues are either such which concern (i) machine support of the system, (ii) human (and other) interfaces between the system and the machine, or (iii) the machine itself.

Requirements describes the system as the stakeholders **would like to see it**.

(b) Functional & Non-Functional Requirements:

Functional requirements include the concepts and facilities to be offered by the desired software.Non-functional requirements emphasise such less tangible issues as performance, user dialogue interface, dependability, etc.

(c) Requirements Engineering
= Capture
↔ Model
↔ Analysis
↔ Theory:

Requirements Engineering, through the process of requirements capture, establishes models of the requirements. The "conversion" from requirements information obtained through requirements elicitation, via requirements modelling to requirements models is called requirements capture. Requirements Models are formally derived from and extends domain models. Requirements Engineering also

analyses requirements models, in order to derive further properties of the requirements.

Discussion: *We hope the reader observes the "similarity" in the components of domain engineering vs. those of requirements engineering.*

(d) Requirements Capture
= Elicitation
↔ Modelling:
Remarks similar to those under Domain Capture — item 5j (page 15) apply.

(e) Requirements Model:
A specification of the requirements. Usually in the form of a set of partial specifications, one for each requirements aspect.

(f) Requirements Modelling Techniques:
Requirements "reside in the domain", and are hence primarily projections of their type space and functions. Functional techniques deal with projections, resolving domain/requirements dichotomies and extending domains. Non-functional techniques deal with machine notions: computing platform, system dependability and maintainability, and with computer human interface issues: user-friendliness, graphic user interfaces, dialogue management, etc.

(g) Requirements Model Analysis:
By Requirements Analysis we understand informal and formal analyses of the Requirements and of the resulting Model — whether informal or formal.

Discussion: *The purposes of the analyses can be to ascertain whether a component and/or its behaviour qualifies as a component (etc.) of the Requirements, and for such included components analyses may reveal Model properties not immediately recognised as properties of the Requirements. Note the distinction being made here: the Requirements as it exists "out there" — among Stake-holders, and the Model as an abstraction thereof and which "exists" on the (electronic) "paper" upon which the Model is represented.*

(h) Requirements Theory:
The purpose of Requirements Analysis is to also establish a Theory of the Domain, or rather: of the Models purported to represent the Domain!

7. Software Concepts:

(a) Software Design
= Software Architecture Specification
↔ Program Organisation Specification
↔ Refinements
↔ Coding:
Software Design, through the process of design ingenuity, proceeds from establishing a software architecture, to deriving a program organisation, and from that, in further steps of design reification, also called design refinement, constructing the "executable code".

(b) Software Architecture:
A software architecture description specifies the concepts and facilities offered the user of the software — i.e. the external interfaces.
Usually functional requirements "translate" into software architecture properties.

(c) Program Organisation:
A program organisation description specifies internal interfaces between program modules (processes, platform components, etc.).
Usually non-functional requirements "translate" into program organisation design decisions.

(d) Refinement:
Design Refinement covers the derivation from the requirements model of the software architecture, of the program organisation from the software architecture,

and of further steps of concretisations into program code.

8. Creation — Acquisition, Elicitation and Invention:

   All stages and steps of the software development process involves creation: domain acquisition & domain modelling, requirements elicitation & requirements modelling, and design ingenuity. This human process of invention leads to the construction of informal as well as formal descriptions.

9. Systematic, Rigorous and Formal Development:

   The software development may be characterised as proceeding in either a systematic, a rigorous or even, in parts, a formal manner — all depending on the extent to which the underlying formal notation is exploited in reasoning about properties of the evolving descriptions.

   (a) *Formal Notation:*
   By a formal notation we understand a language with a precise syntax, a precise semantics (meaning), and a proof system. By "a precise ..." we usually mean "a mathematical ...".

   (b) *Systematic Use of Formal Notation:*
   By a systematic use of formal notation we understand a use of the notation in which we follow the precise syntax and the precise semantics.

   (c) *Rigorous Use of Formal Notation:*
   By a rigorous use of formal notation we understand a systematic use in which we additionally exploit some of the 'formality' by expressing theorems of properties of what has been written down in the notation.

   (d) *Formal Use of Formal Notation:*
   By a formal use of formal notation we understand a rigorous use in which we fully exploit the 'formality' by actually proving properties.

   (e) *Formal Method* $\approx$ *Formal Specification* $\otimes$ *Calculation:*
   We refer to item 3 (page 21) for a definition of 'method'.
   The methods claimed today to be formal methods may be formal, but are not methods in the sense we define that term! Since we do not believe that a method for developing software: from domains via requirements, can be formal, but only that use of the notations deployed may be, we (now) prefer the terms: formal specification and calculation.

   (f) *Design Calculi — or Formal Systems:*
   By a design calculus we understand a formal system consisting of a formal notation and a set of precise rules for converting expressions of the formal notation into other such, semantically 'equivalent' expressions.

10. Satisfaction = Validation $\oplus$ Verification:

    The domain acquisition and requirements elicitation processes alternate with domain modelling and requirements modelling, respectively, and these again with securing satisfaction.

    (a) Validation:
    In this report we are not interested in the crucial process of interactions between software developers (i.e. software engineers, which we see as domain engineers, requirements engineers and software designers) and the stakeholders. validation is thus the act of securing, through discussion, etc., with the stakeholders that the domain model correctly reflects their understanding of the domain.

    (b) Verification:
    Let $\mathcal{D}$, $\mathcal{R}$ and $\mathcal{S}$ stand for the theories of the fomain, requirements and software. Then verification:

    $$\mathcal{D}, \mathcal{S} \models \mathcal{R}$$

    shall mean that we can verify that the designed software satisfies the requirements in the presence of knowledge (i.e. a theory) about the fomain.

11. Software Engineering:

Software Engineering is the combination of domain engineering, requirements engineering and software design, and is seen as the process of going between science and technology. That is, of developing descriptions on the basis of scientific results using mathematics — as in other engineering branches — and of understanding (the constructed fomain of) existing (software) technologies by subjecting them to rigorous domain analysis.

12. Frame Specialisation:

> Discussion: *In item 2 (page 12) we discussed the problem of software engineering being seemingly as a too wide field. And we hinted that specialisation might be a natural way of achieving a level of professionalism achieved in traditional engineering fields. In this item we will briefly introduce the concept of problem frames and give example of distinct such frames.*

A problem frame is well-delineated part of all the problems to which computing might be applied — such that this frame offers a precise set of principles, techniques and tools for software development, and such that this 'method' fits the frame "hand in glove".

- *Principal Parts and Solution Task*

Following Jackson [42, 43] we think of a (problem) frame as consisting of its principal parts and a solution task. The principal parts are (1.) the domain — which exists a-priori — and (2.) the requirements. The solution task is that of developing the software — something relatively new! Tackling an application problem consists initially of analysing it **into** a frame, including a multi-frame with clearly identified part-frames.

We explain a few frames and otherwise refer to [42, 43, 11]:

(a) *Translation Frame:*

The principal parts are: (i) two formalised languages (syntax and semantics), source and target; (ii) the concrete form of the syntactic representations of either: the source usually in the form of a BNF grammar for textual input, the target usually in the form of an internal ("electronic") data structure; (iii) user requests for compilation from source to target; (iv) the compiler; and (v) the translation function. (i-ii) form the domain, (iii-iv-v) the requirements.

The solution task now involves developing the compiler using a well-defined set of techniques and tools: lexical scanner generators, possibly error-correcting parser generators, attribute grammar interpreters, etc.

(b) *Reactive Systems Frame:*

The principal parts are (i) the dynamic (temporal, real-time) "real world"; (ii) its observable variables [output], (iii) its controllable variables [input]; (iv) user (or other system) requests for the monitoring and/or control of the "real world"; (v) the monitoring & control (software etc.) system; and (vi) the specific monitoring & control functions (optimisation, safety, dependability, etc.). Items (i-ii-iii) form the domain, (iv–vi) the requirements.

The solution task now involves control theoretic and real-time, safety critical software design principles, techniques and tools.

It seems that Jackson refers to the reactive systems frame as the environment-effect frame [42].

(c) *Information Systems Frame:*

The principal parts are almost as for reactive systems (i-ii) except that there is no desire for control, and the issues of safety criticality, real-time and dependability are replaced by (vi) (observable) information security and the need for usually "massive" information storage (for statistical and other purposes); (iv) the requests are concerned with the visualisation of observed information and

computations over these; (v) the system is thus more of an information (monitoring) system; and (vi) the functions include specifics about the visualisation and other processing.

The solution task can perhaps best be characterised in terms of the principles, techniques and tools for example offered by Jackson's JSD method [41, 42].

(d) *Connection Frame:* See [42, 43, 11] for details.

(e) *Workpiece Frame:* See [42, 43, 11] for details.

(f) *Transaction Frame:* See [42, 43, 11] for details.

(g) *Multi–frame:* See [42, 43, 11] for details. Usually a problem is not reducible to a single of the frames mentioned above (and some of these, due to requirements, often "overlap"). In such cases we have a multi-frame, a frame being best characterised in terms of hopefully reasonable well-delineated (sub-) frames.

(h) *&c.*

B.2 General Terminology

Many more terms are used in the subject field of this report: in its science and in its engineering. Sometimes with unclear meanings, and not always with the same meaning from paper to paper. We shall therefore try delineate also important general concepts.

Some dogmas:

1. Computer Science:

   Computer Science, to us, is the study and knowledge of the foundations of the artifacts that might exist inside computers: the kinds of information, functions and processes (i.e. type theory), models of computability and concurrency; bases for denotational, algebraic and operational semantics; specification and programming language proof theories; automata theory; theory of formal languages; complexity theory; etc.

2. Computing Science:

   Computing Science, to us, is the study and knowledge of how to construct the artifacts that are to exist inside computers. Successful computing science results in a useful programming methodology.

   The present report "falls", subject-wise, somewhere between computing science and software engineering.

3. Method:

   By a method we understand a set of *principles* of *analysis*, and for *selecting* and *applying techniques* and *tools* in order *efficiently* to *construct efficient artifacts* — here software.

4. Methodology:

   By methodology we understand the study and knowledge about methods. Since we can assume that no one software development method will suffice for any entire construction process we need be concerned with methodology.

5. Software:

   By software we understand all the documentation that is necessary to *install, operate, run, maintain* and *understand* the executable code, as well as that *code* itself and the *tools* that are needed in any of the above (i.e. including the original development tools).

6. Software Technology:

   By software technology we understand sets of software tied to sets of specific platforms. (By a platform we mean "another" machine!)

7. Programming:

   Programming is a subset of activities within software engineering which focus on the systematic, via rigorous to formal creation of descriptions using various design calculi.

8. Engineering:

   Engineering is the act of constructing technology based on scientifically established results and of understanding existing technologies scientifically.

9. Engineer:

   Engineers perform engineering and use, as a tool, mathematics. It is used in order to model, analyse, predict, construct, etc. software engineers reason about the artifacts they construct, be they (fomain, requirements, software architecture, program organisation, etc.) model descriptions (i.e. definitions or specifications) or program code.

10. Technician:

    Technicians use technologies: they compose, use and "destroy" them — without necessarily using mathematics.

11. Technologist:

    Technologists are technicians who manage technologies: perceive, demand, produce, procure, market and deploy technologies.

This report views software engineering as hinted above: As the act of going between science and technology, using mathematics — wherever useful.

REFERENCES

[1] Jean-Raymond Abrial. *The B Book: Assigning Programs to meanings.* Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1996.

[2] Cleta Milagros Acebedo. An Informal Domain Analysis for Manufacturing Enterprises. Research Report 62, UNU/IIST, P.O.Box 3058, Macau, March 1996.

[3] Cleta Milagros Acebedo and Erwin Paguio. Manufacturing Enterprise Simulation: A Business Game. Research Report 64, UNU/IIST, P.O.Box 3058, Macau, March 1996.

[4] Dao Nam Anh and Richard Moore. Formal Modelling of Large Domains — with an Application to Airline Business. Technical Report 74, UNU/IIST, P.O.Box 3058, Macau, June 1996. Revised: September 1996.

[5] D. Bjørner. A Software Engineering Paradigm: From Domains via Requirements to Software. Research report, Dept. of Information Technology, Technical University of Denmark, Bldg.345/167–169, DK–2800 Lyngby, Denmark, July 1997.

[6] D. Bjørner. Towards a Domain Theory of The Financial Sevice Industry. Research report, Dept. of Information Technology, Technical University of Denmark, Bldg.345/167–169, DK–2800 Lyngby, Denmark, July 1997.

[7] D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science.* Springer-Verlag, 1978.

[8] Dines Bjørner. Formal Models of Robots: Geometry & Kinematics. Research Report 6, UNU/IIST, P.O.Box 3058, Macau, March 15 1993. For abbreviated version see Chapter 3 in *A Classical Mind*, Festschrift for C.A.R. Hoare, Prentice-Hall International, Publ., 1994, pp 37–58.

[9] Dines Bjørner. Prospects for a Viable Software Industry — Enterprise Models, Design Calculi, and Reusable Modules. Technical Report 12, UNU/IIST, P.O.Box 3058, Macau, 7 November 1993. Appendix — on a railway domain model — by Søren Prehn and Dong Yulin, Published in *Proceedings from first ACM Japan Chapter Conference*, March 7–9, 1994: World Scientific Publ., Singapore, 1994.

[10] Dines Bjørner. Models of Enterprise Management: Strategy, Tactics & Operations — Case Study Applied to Airlines and Manufacturing. Technical Report 60, UNU/IIST, P.O.Box 3058, Macau, January – April 1996.

[11] Dines Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In Li ShaoYang and Michael Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society.

[12] Dines Bjørner. Models of Financial Services & Industries. Research Report 96, UNU/IIST, P.O.Box 3058, Macau, January 1997. Incomplete Draft Report.

[13] Dines Bjørner, Chris W. George, Bo Stig Hansen, Hans Laustrup, and Søren Prehn. A Railway System, Coordination'97, Case Study Workshop Example. Research Report 93, UNU/IIST, P.O.Box 3058, Macau, January 1997.

[14] Dines Bjørner, Dong Yu Lin, and Søren Prehn. Domain Analyses: A Case Study of Station Management. Research Report 23, UNU/IIST, P.O.Box 3058, Macau, 9 November 1994.

[15] J.P. Bowen and M. Hinchey. Seven More Myths of Formal Methods. Technical Report PRG–TR–7–94, Oxford Univ., Programming Research Group, Wolfson Bldg., Parks Road, Oxford OX1 3QD, UK, June 1994. Shorter version published in LNCS Springer Verlag FME'94 Symposium Proceedings.

[16] J.P. Bowen and M. Hinchey. Ten Commandments of Formal Methods. Technical report, Oxford Univ., Programming Research Group, Wolfson Bldg., Parks Road, Oxford OX1 3QD, UK, 1995.

[17] M. Cerioli, A. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Permissive Sub-sorted Partial Logic in CASL. In *Proceedings, AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*. Springer–Verlag, 1997.

[18] Zhou Chaochen. Duration Calculi: An Overview. Research Report 10, UNU/IIST, P.O.Box 3058, Macau, June 1993. Published in: *Formal Methods in Programming and Their Applications*, Conference Proceedings, June 28 – July 2, 1993, Novosibirsk, Russia; (Eds.: D. Bjørner, M. Broy and I. Pottosin) LNCS 736, Springer-Verlag, 1993, pp 36–59.

[19] Zhou Chaochen and Michael R. Hansen. Lecture Notes on Logical Foundations for the Duration Calculus. Lecture Notes, 13, UNU/IIST, P.O.Box 3058, Macau, August 1993.

[20] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, 1991.

[21] Zhou Chaochen and Yu Huiqun. A duration Model for Railway scheduling. Technical Report 24b, UNU/IIST, P.O.Box 3058, Macau, May 1994.

[22] Zhou Chaochen, Dang Van Hung, and Li Xiaoshan. A Duration Calculus with Infinite Intervals. Research Report 40, UNU/IIST, P.O.Box 3058, Macau, February 1995. Published in: *Fundamentals of Computation Theory*, Horst Reichel (ed.), pp 16-41, LNCS 965, Springer-Verlag, 1995.

[23] Zhou Chaochen, Anders P. Ravn, and Michael R. Hansen. An Extended Duration Calculus for Real-time Systems. Research Report 9, UNU/IIST, P.O.Box 3058, Macau, January 1993. Published in: *Hybrid Systems*, LNCS 736, 1993.

[24] Zhou Chaochen and Li Xiaoshan. A Mean Value Duration Calculus. Research Report 5, UNU/IIST, P.O.Box 3058, Macau, March 1993. Published as Chapter 25 in *A Classical Mind*, Festschrift for C.A.R. Hoare, Prentice-Hall International, 1994, pp 432–451.

[25] Razvan Diaconescu and Kokichi Futatsugi. Logical Semantics of CafeOBJ. Research Report IS-RR-96-0024S, JAIST, 1996.

[26] Do Tien Dung, Le Linh Chi, Nguyen Le Thu, Phung Phuong Nam, Tran Mai Lien, and Chris George. Developing a Financial Information System. Technical Report 81, UNU/IIST, P.O.Box 3058, Macau, September 1996.

[27] Myatav Erdenechimeg, Richard Moore, and Yumbayar Namsrai. MultiScript I: The Basic Model of Multi-lingual Documents. Technical Report 105, UNU/IIST, P.O.Box 3058, Macau, June 1997.

[28] Yu Xinyiao et al. Stability of Railway Systems. Technical Report 28, UNU/IIST, P.O.Box 3058, Macau, May 1994.

[29] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques*. Cambridge University Press, 1997–1998.

[30] Yang Lu Fu Hongguang and Zhou Chaochen. A Computer-Aided Geometric Approach to Inverse Kinematics. Research Report 101, UNU/IIST, P.O.Box 3058, Macau, April 1997.

[31] Kokichi Futatsugi and Razvan Diaconescu. *CafeOBJ Report — Definition of the Language.* To appear in 1998 as book in the AMAST series at World Scientific

[32] Jan Goossenaerts and Dines Bjørner. An Information Technology Framework for Lean/Agile Supply-based Industries in Developing Countries. Technical Report 30, UNU/IIST, P.O.Box 3058, Macau, 1994. Published in *Proceedings of the International Dedicated Conference on Lean/Agile Manufacturing in the Automotive Industries*, ISATA, London, UK.

[33] Jan Goossenaerts and Dines Bjørner. Interflow Systems for Manufacturing: Concepts and a Construction. Technical Report 31, UNU/IIST, P.O.Box 3058, Macau, 1994. Published in *Proceedings of the European Workshop on Integrated Manufacturing Systems Engineering.*

[34] The RAISE Language Group. *The RAISE Specification Language.* The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[35] The RAISE Method Group. *The RAISE Method.* The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[36] J. Guttag, J.J. Horning, and J.M. Wing. Larch in Five Easy Pieces. Technical Report 5, DEC SRC, Dig. Equipm. Corp. Syst. Res. Ctr., Palo Alto, California, USA, 1985.

[37] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, , and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, Springer-Verlag New York, Inc., Attn: J. Jeng, 175 Fifth Avenue, New York, NY 10010-7858, USA, 1993.

[38] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, 1990.

[39] Dang Van Hung and Zhou Chaochen. Probabilistic Duration Calculus for Continuous Time. Research Report 25, UNU/IIST, P.O.Box 3058, Macau, May 1994. Presented at *NSL'94 (Workshop on Non-standard Logics and Logical Aspects of Computer Science, Kanazawa, Japan, December 5–8, 1994)*, submitted to *Formal Aspects of Computing.*

[40] Dang Van Hung and Phan Hong Giang. A Sampling Semantics of Duration Calculus. Research Report 50, UNU/IIST, P.O.Box 3058, Macau, November 1995. Published in: *Formal Techniques for Real-Time and Fault Tolerant Systems*, Bengt Jonsson and Joachim Parrow (Eds), LNCS 1135, Spriger-Verlag, pp. 188–207, 1996.

[41] M. Jackson. *System Design.* Prentice-Hall International, 1985.

[42] Michael Jackson. Problems, methods and specialisation. *Software Engineering Journal*, pages 249–255, November 1994.

[43] Michael Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices.* ACM Press. Addison-Wesley Publishing Company, Wokingham, nr. Reading, England; E-mail: ipc@awpub.addwes.co.uk, 1995. ISBN 0-201-87712-0; xiv + 228 pages.

[44] T. Janowski and C.M. Acebedo. Virtual Enterprise: On Refinement Towards an ODP Architecture. Research Report 69, UNU/IIST, P.O.Box 3058, Macau, May 1996.

[45] Tomasz Janowski. Domain Analysis for Manufacturing: Formalization of the Market. Research Report 63, UNU/IIST, P.O.Box 3058, Macau, March 1996.

[46] Tomasz Janowski and Rumel V. Atienza. A Formal Model For Competing Enterprises, Applied to Marketing Decision-Making. Research Report 92, UNU/IIST, P.O.Box 3058, Macau, January 1997.

[47] C.B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, second edition, 1990.

[48] Souleymane Koussoubé. Knowledge-Based Systems: Formalisation and Applications to Insurance. Research Report 108, UNU/IIST, P.O.Box 3058, Macau, May 1997.

[49] K. Lano. *The B Language and Method, A Guide to Pratical Formal Development.* Springer-Verlag, Formal Approaches to Computing and Information Technology (FACIT). Ed.: S.A. Schuman, 1996.

[50] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Specifications.* Addison Wesley, 1991.

[51] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Safety.* Addison Wesley, 1995.

[52] P.D. Mosses. COFI: The Common Framework Initiative for Algebraic Specification and Development. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97*, volume 1212 of *Lecture Notes in Computer Science.* Springer–Verlag, 1997.

[53] Ataru T. Nakagawa, Toshimi Sawada, and Kokichi Futatsugi. *CafeOBJ manual (for system version 1.3).* 142 pages.

[54] Kokichi Futatsugiand Ataru Nakagawa. An Overview of CAFE Specification Environment – An Algebraic Approach for Creating, Verifying, and Maintaining Formal Specifications over Networks. In *ICFEM'97: International Conference on Formal Engineering Methods.* IEEE Computer Society, IEEE CS Press, November 1997.

[55] Nikolaj Nikitchenko. Towards Foundations of the General Theory of Transport Domains. Research Report 88, UNU/IIST, P.O.Box 3058, Macau, December 1996.

[56] Roger Noussi. An Efficient Construction of a Domain Theory for Resources Management: A Case Study. Research Report 107, UNU/IIST, P.O.Box 3058, Macau, May 1997.

[57] CoFI Task Group on Language Design. CASL — The Common Algebraic Specification Language Summary. Available at http://www.brics.dk/Projects/CoFI/-Documents/CASL/Summary/, 1997.

[58] CoFI Task Group on Semantics. CASL — The CoFI Algebraic Specification Language (version 0.97) Semantics. Available at http://www.brics.dk/Projects/CoFI/Notes/S-4/, 1997.

[59] J. Michael Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, UK, January 1988.

[60] J. Michael Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1989.

[61] John Wordsworth. *Software Engineering with B.* Addison-Wesley Longman, 1996.

[62] Tan Xinming. Enquiring about Bus Transport–Formal Development using RAISE. Technical Report 83, UNU/IIST, P.O.Box 3058, Macau, September 1996.

第10回

テクニカルマネジメントワークショップ

ソフトウェア産業の二極化

ー参加者募集ー

「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，早いもので今年１０回目を迎えることになりました．今回は，北方領土返還を願いながら，昨年の隠岐島から日本海を北上して，新潟の粟島に場所を移し，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．

さて，情報サービス産業は93年度を底に拡大しておりますが，ソフト開発需要の続伸と情報処理需要の停滞という二極化が始まっています．さらに，昨年は大手金融機関が倒産した様に，従来では考えられなかった企業運営の難しさが出ており，勝ち組･負け組といった真の二極化が始まっています．

このような状況下にあって，IEEE Software 誌の January-February 1998 号に Ed Yourdon が発表した「A Tale of Two Futures」(二つの未来物語) は、来るべき未来は，はたしてソフトウェアの黄金時代？それとも，現状よりさらに悲惨な悪夢の時代なのか？が、彼の問題提起でした．マイクロソフト製品を使わざるをえない状況での技術進歩，ソフトウェア･エンジニアリングと経済社会の共存，ゲーム感覚でのシステム作り，等の当問題は，これからのマネージメントを考える良い機会ではないでしょうか．SEA では3月の Forum でも，この問題を取り上げましたが，さらなる議論をしたいと思います．

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： 1998年7月9日(木)午後から11日(土)正午まで(2泊3日)

場　所： 新潟県･粟島 松太屋旅館

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トップ･マネジメント．

費　用： SEA 会員　40,000円，一般　50,000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)，武田 淳男 (安川情報システム)

基調講演： 松原 友夫 (松原コンサルティング)

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： 裏面の参加申込用紙に必要事項を記入の上，**6月24日(水)**までに，E-mail または Fax で，下記宛にお送り下さい．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)　情報技術研究所　田中 一夫

Tel: 03-3623-5082　　Fax: 03-3623-8779　　E-mail: kazuo.tanaka@jfits.co.jp

***** 裏面にワークショップの申込用紙があります *****

第10回 テクニカルマネジメントワークショップ参加申込用紙

氏名：　　（ふりがな）

種別：□正会員（No.　　）□賛助会員（No.　　）□一般

年齢：　　（歳）　性別：□男 □女　血液型：

会社名：

部門：　　役職：

郵便番号：（〒　　）

住所：

TEL:（　　）－（　　）－（　　）内線（　　）

FAX:（　　）－（　　）－（　　）

E-MAIL:

ポジションステートメント

◆あなたは，今回のワークショップでどのようなことを討論したいですか?また、そのテーマについてどのような御意見をお持ちですか?

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