ソフトウェアシステム設計後の知的分業

富士通株式会社 国際情報社会科学研究所　小林　要
Kaname Kobayashi

1．はじめに

ソフトウエアのシステム設計が終了した後の開発工程に着目する。　システム設計が終了した時点では、システムの基本要素となるべきコンポーネントの仕様が決められ、システムの動作環境もほぼ決定されているものとする。システム設計の終了後、各コンポーネントの作成、コンポーネントテスト、システムの実現、システムテスト、運用、評価、保守、と続くと考える。これらの工程のうち、各コンポーネントを実現し、コンポーネントテストを終了するまでを、コンポーネント作成工程と呼ぶことにする。

コンポーネント作成工程は、ソフトウエア開発の全工程の工数に比して、最も工数を要する工程である。特にモジュール分割により、複数の作業者が同時に作業を分担して進めると、分業の効果を得る可能性の大きい工程である。本報告では、コンポーネント作成工程においてどのような観点から分業を行うのが効果的であるか、またそのための方法として、どのようなものがあるかについて問題を提起し、一つの方法を提案する。

2．分業を考える観点

分業のしかたには様々の種類があろう。立場によって分業の種類を分ける観点が違う。組織を構築し管理する立場から考えれば、「どういう部門構成が望ましいか」を考え、大がかりな分業体制を問題にするであろうが、現場の監督をする立場から考えると、「誰にこの仕事をさせようか」と考えるに違いない。一方、作業を実際に行う作業者本人からみれば、自分の得意な、あるいはよく知っている作業を分担するかどうかや、作業の易しさ、難しさ、作業の内容そのものなどを問題にする。

分業の一般的な意味での〝効果〝は同じ工数で同じ製品をつくる場合に比較して、

（a）作業の単純化による個人の技能の改善、
（b）ある仕事から別の仕事へと手をかえる場合に失われる時間の節約、
（c）作業目標の明確化により、作業手段が向上、

などを得て、品質と生産性をともに向上しようとする点にある。したがって分業のしかたを考えるには、まず作業そのものの性格と作業に必要な技術背景を問題にしなければならない。プログラミング作業を問題解決の作業と考えると作業に必要な考慮対象の範囲をどのようにして限定するか、問題自体をどのように整理するか、そして作業をどのように単純化し作業目標を明確化するかなどが検討されるべきであろう。本報告では、「作業に必要な知識の種類」に注目した分業を考える。

3．コンポーネント作成工程における作業知識

計算機プログラムは計算機システムを駆使して所望の機能を実現するための手段である。したがってプログラムには、機能本来の主旨があり、実体があり、かつその実行環境がある。実際のプログラムにはこれらに関連する情報が渾然一体となって含まれている。機能本来の主旨に関する情報はそのプログラムの応用分野における目的に依存した情報であり、基本的な入出力関係とアルゴリズムに反映する。この種の情報のことをＡＰＤＩ（Application Purpose Dependent Information)（応用目的情報）と呼ぶことにする。

プログラムの実行を行うには必ず実行環境が定まらなくてはならない。計算機システムにおけるプログラムの実行環境とは、その時々の計算機ハードウエアの構成やオペレーティングシステム、および他の関連プログラムとの結びつきの状況であり、プログラムの動作上の"意味"を理解するために不可欠である。実行環境に依存する情報はプログラムの中で用いている部品とそのためのインターフエースの部分、様々のエラー処理の部分、および障害時の回復処理の部分などに反映する。この種の情報のことをＥＥＤＩ（ Ｅｘｅｃｕｔｉｏｎ Ｅｎｖｉｒｏｎｍｅｎｔ Ｄｅｐｅｎｄｅｎｔ Ｉｎｆｏｒｍａｔｉｏｎ ）（実行環境情報）と呼ぶことにする。

作業者にとってプログラムの具体的実体は、何らかの言語で記述されたテキストである。したがって先に述べた応用目的に依存する情報（ＡＰＤＩ）と、実行環境に依存する情報（ＥＥＤＩ）のほかに、プログラミング言語それ自体の運用状況やモジュールの分割のしかたなど、プログラミングのテクニックに関する付随的な情報がある。この種の情報のことを ＰＴＤＩ （ Ｐｒｏｇｒａｍｍｉｎｇ Ｔｅｃｈｎｉｑｕｅ Ｄｅｐｅｎｄｅｎｔ Ｉｎｆｏｒｍａｔｉｏｎ）（プログラミングテクニック情報）と呼ぶことにする。

以上の三種類の情報（ＡＰＤＩ，ＥＥＤＩ，ＰＴＤＩ）は渾然一体となってプログラムの中に反映されているために、プログラムを理解するには、これらの情報を巧みによりわけてやる必要がある。ところで、ＡＰＤＩ，ＥＥＤＩ，ＰＴＤＩ を自由に操るには、各々に関する知識が必要である（図１）。これらの知識は相互にかなり独立であり、かつ底が深い。各々の知識を次のように呼ぶことにする。

＊応用目的知識＊

ＡＰＤＫ（Ａｐｐｌｉｃａｔｉｏｎ Ｐｕｒｐｏｓｅ Ｄｅｐｅｎｄｅｎｔ Ｋｎｏｗｌｅｄｇｅ）

応用分野における応用目的と応用アルゴリズムに依存する部分の知識、および応用目的に関する知識。

＊実行環境知識＊

ＥＥＤＫ（Ｅｘｅｃｕｔｉｏｎ Ｅｎｖｉｒｏｎｍｅｎｔ Ｄｅｐｅｎｄｅｎｔ Ｋｎｏｗｌｅｄｇｅ）

プログラムの実行環境に依存する部分の知識、および実行環境の知識。

＊プログラミングテクニック知識＊

ＰＴＤＫ（Ｐｒｏｇｒａｍｍｉｎｇ Ｔｅｃｈｎｉｑｕｅ Ｄｅｐｅｎｄｅｎｔ Ｋｎｏｗｌｅｄｇｅ）

プログラミングテクニックに依存する部分の知識、およびプログラミングテクニックの知識。

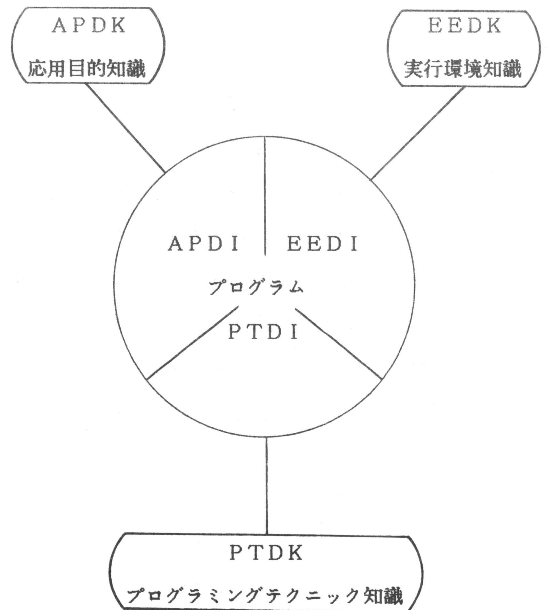

図１：ＡＰＤＫ，ＥＥＤＫ，ＰＴＤＫと ＡＰＤＩ，ＥＥＤＩ，ＰＴＤＩ．

作業者はプログラムを自由に操るためには、APDI，EEDI，PTDIを巧みによりわけながら理解し、対処するのに十分な　APDK（応用目的知識）、EEDK（実行環境知識）、PTDK（プログラミングテクニック知識）を持っていなければならない。しかし実際には応用分野における知識体系と計算機実行環境の知識体系とプログラミングの知識体系とをすべて自分のものとすることは困難である。最近はAPDK，EEDK，PTDKそれぞれが多様化し深化しているので、すべてに通じている人を期待できない。現在はむしろAPDKに通じている人、EEDKに通じている人、PTDKに通じている人、などそれぞれ専門分化している。作業者の知識バランスが図2のように三者三様になってきたと言える。

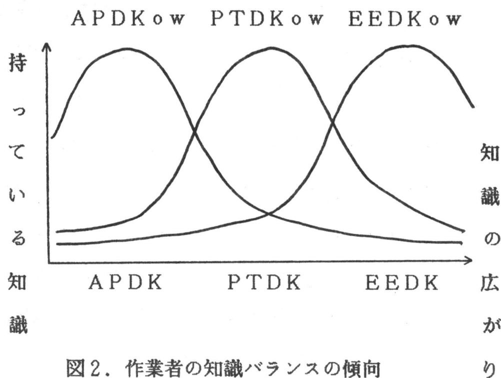

図2．作業者の知識バランスの傾向

プログラミング作業の分業を意図した場合、作業に必要な知識の有無、知識の獲得能力などは、作業を分担してから、実質的な作業に入るまでの時間にきいてくる。さらに、推論能力の程度や獲得された知識の内容によって品質が左右される。APDKが不十分だと、目的の軽重をはかりかねて、利用者の本来の意図が理解不十分になり、使い物にならないことが多い。EEDKが不十分だと、環境条件の認識不足からとんでもない事故につながったり、信頼性の低いものになりがちである。さらに、PTDKが不十分だと、性能が悪かったり、保守性の悪いものやバグの多いものになりがちである。

ここでいう知識とは、主に技術的な知識であり、それをもとに推論が可能となる情報であって、かつ作業者が自分のものとしてこれを活用しうるものである。俗にいう″センスのよいプログラマー″は、APDK，EEDK，PTDK，のそれぞれについて、推論の中核になる有用な知識を上手に会得し、巧みに推論する人のことではなかろうか。

4．コンポーネント作成工程における知的分業

図2のような作業者の知識バランスを考慮に入れると、APDKに通じた者、PTDKに通じた者、EEDKに通じた者の三者がそれぞれの持ち味を生かしながら協力しあう作業形態が望まれる。その場合、作業者相互の情報交換にはできるだけ少ない知識ですむような工夫が必要である。ここでは、空間分業、時間分業〔1〕の各々についてAPDK，PTDK，EEDK，との関連を考察する。

コンポーネント作成工程における空間分業は、最終的に出来上がったコンポーネントに対し、「この部分は自分が担当した」と直接プログラムテキストを指示することのできる分業である．分業の第一のポイントが「作業対象の空間の分割」にある。プログラムをいくつかの空間に分割する方法はさまざまあるであろうが、APDI，EEDI，PTDI，にそってモジュール分割することは必ずしもできることではない。しいていうならば、EEDK志向作業者（　EEDK　oriented　worker）はできるだけ実行環境の個別的な相違を表面に出さないように工夫した、実行環境の詳細を知らなくても安心して使える信頼性の高い基礎的な部品を提供することによって役割を果たすことができる。

一方、応用目的志向作業者（　APDK　oriented　worker）はいわゆる高級言語を用いて、応用機能別の各モジュールを作成することに

なるが、ＰＴＤＫが必ずしも十分でないために、諸種のプログラミング上の問題が生じる。

空間分業のよさは作業対象に関する知識の範囲をしぼりこめることである。応用分野を限定したり、実行環境を限定することは作業の能率を上げるのに役立つ。しかしながら、プログラミングテクニックは作業対象ではないために、空間分業のワク組みからはずされてしまうのである。したがって、プログラミングテクニック志向作業者（ＰＴＤＫ　ｏｒｉｅｎｔｅｄ　ｗｏｒｋｅｒ）が明確な位置づけを与えられにくい。

コンポーネント作成工程における時間分業は、最終的に出来上がったコンポーネントに対し、「何という工程は自分が担当した」と言って、時間軸上の関与を示すことのできる分業である。　ＳＤＥＭ〔２〕（Ｓｏｆｔｗａｒｅ　Ｄｅｖｅｌｏｐｍｅｎｔ　Ｅｎｇｉｎｅｅｒｉｎｇ　Ｍｅｔｈｏｄｏｌｏｇｙ）に従うと、システム設計以降（システム設計レビュー以降）は次のように工程が分けられる。

（１）ＰＳ（プログラム構造設計）

モジュール構造、モジュール内処理手順概要、モジュールの入出力定義、物理的なデータ設計（共通ファイルやテーブル）などを定めて、プログラム構造設計書をつくる。

（２）ＭＤ（モジュール設計）

モジュール内のデータ設計を行い、モジュール内処理手順を詳細化する。モジュール設計書、モジュールテスト仕様書、結合テスト仕様書、および解説書を作成する。

（３）ＰＤＲ（プログラム設計レビュー）

プログラム設計内容をレビューし、不都合な部分を発見したら再設計にもどす。

（４）ＰＧ（プログラミング）

コーディング、デバッグ、モジュールテストを行う。ソースリスト、モジュールテスト成績書、システム（コンポーネント）テスト仕様書を作成する。

（５）ＩＴ（結合テスト）

モジュールを結合し、プログラム（コンポーネント）をつくり、テストする。結合テスト成績書が出来上がる。

以上の後、システムテスト（ＳＴ）、テスティングレビュー（ＴＧＲ）、運用テスト（ＯＴ）、保守、システム評価（ＭＥ）と続くわけであるが、コンポーネント作成工程の部分は上記　ＰＳ－ＭＤ－ＰＤＲ－ＰＧ－ＩＴ　までである。

ＳＤＥＭの工程に従う場合、ＰＳ，ＭＤ，ＰＤＲ，ＰＧ，ＩＴの各々に作業者を割りふることが時間分業に相当する。プログラム構造設計者は、入出力関係のあるべき姿と、性能や保守性のよいモジュール分割をまかされる。ＡＰＤＫ，ＰＴＤＫ，ＥＥＤＫのいずれにも通じていればよいというわけにもいかず、実際は勘と経験が必要になる。モジュール設計者は、ＡＰＤＩとＥＥＤＩの狭間で苦労する。理解性をよくするにはＡＰＤＩが明確に見えるようにすればよいが、性能や保守性、信頼性などのために、ＰＴＤＩ，ＥＥＤＩが混入し、すっきりしたものにはなかなかならない。ＰＧ段階の作業者は、同一テキストの中に渾然一体となって存在するＡＰＤＩ，ＰＴＤＩ，ＥＥＤＩの扱いに苦労する。

時間分業のよさは作業工程に関する知識の範囲をしぼりこめることである。作業環境を規定し技術向上をねらえる。しかし単なる工程分割に基づく時間分業ではＡＰＤＫ，ＰＴＤＫ，ＥＥＤＫのすべてを依然として強く要求することになる。勘と経験にたよることにもなりかねない。

空間分業のよさである作業対象分野の知識別分業の考え方と、時間分業のよさである作業工程技術の知識別分業の両方を統合したような分業方式が望ましいのではなかろうか。次に一つの分業方式を提案する。

５．コンポーネント作成工程における設計変換法

コンポーネント作成工程を次の三つに分ける。

（ＩＤ）初期設計：Ｉｎｉｔｉａｌ　Ｄｅｓｉｇｎ

（ＤＴ）設計変換：Ｄｅｓｉｇｎ　Ｔｒａｎｓ－ｆｏｒｍａｔｉｏｎ

（ＩＭ）実体化：ＩＭｐｌｅｍｅｎｔａｔｉｏｎ

初期設計段階ではＡＰＤＫ志向作業者が所定の設計言語を用いて、専らプログラムの入出力関係の保証を行う。入出力の関係に注目するのであって、入出力の具体的なサイズや型まで決める必要はない。実行環境への配慮もあまりしなくてよい。性能については、基本的なアルゴリズムの評価を行うが、プログラミングテクニックによる性能向上までは考えなくてよい。初期設計におけるモジュール分割は、最終的なプログラムのモジュール分割と一致しなくてよい。モジュール（設計モジュール）の分割のポイントはＡＰＤＩの理解性である。そのため、中間的な変数を多用して入出力関係が容易に把握できるようにすることが望ましい。ＰＴＤＩ，ＥＥＤＩを極力必要としないようにする。

設計変換段階ではＰＴＤＫ志向作業者が初期設計を受け取り、システム設計書とも照らし合わせながら、これを変換する。変換作業は同じ設計言語の上で行い、最終設計を得る。変換の主な目的はＰＴＤＩの注入である。初期設計にはＡＰＤＩが反映しているとし、その入出力関係を保ちながら、プログラムとしての構造を整える。中間変数の消去、冗長な条件判定の除去、既存部品の利用のための再構造化、スタックやファイル、バッファ、その他のデータ構造の導入、保守性を考慮したモジュール再統合と再合成などを行う。設計言語（後述する）の特徴から、処理内容はそのすべてを記述するが、ターミナルモジュールは次の実体化段階で用いるインプリメンテーション言語の基礎部品を意識してよい。すなわち最終的に出来上がるであろうプログラムの内部における処理と外部における処理（外部部品）との区別を意識してよい。あるいは意識しなくてもよい。

実体化段階では、ＥＥＤＫ志向作業者が上記設計変換段階で作成された最終設計を受け取り、システム設計書と照らし合わせながら、最終設計を実際のインプリメント言語（コンパイラ言語など）に写像する。写像作業における主な目標はＥＥＤＩの注入である。最終設計は、実行環境が意識されているいないにかかわらず、設計言語ですべて記述されているために、具体的な実行環境を選ばない。ＥＥＤＩが欠如しているともいえる。実体化の作業では、インプリメント言語の特徴を生かしながら個々の変数に対して型やサイズを与え。さらに実行環境との整合性をとり、エラー処理ルーチンの追加、リカバリー処理ルーチンの追加などを行う。

このアプローチのことを設計変換アプローチ（ＤＴＡ：Ｄｅｓｉｇｎ　Ｔｒａｎｓｆｏｒｍａｔｉｏｎ　Ａｐｐｒｏａｃｈ）と呼ぶ。ＤＴＡの工程分割の方針はＡＰＤＫ志向作業者、ＰＴＤＫ志向作業者、ＥＥＤＫ志向作業者の三者が、それぞれの互いに独立した技術知識を背景に知的分業（知識労働の分業）をすることにある。ＤＴＡの全体を図３に示す。

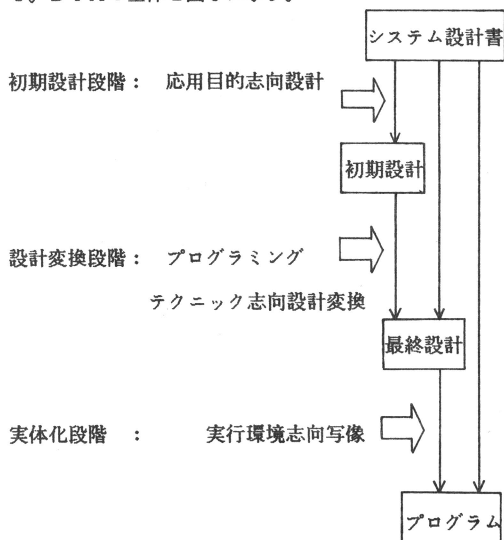

図３．設計変換アプローチ（ＤＴＡ）

6．PDL／DTA（Program Design Language for DTA）

ＤＴＡで用いる設計言語は次の性質をもつ必要がある。

（a）ＡＰＤＫ，ＰＴＤＫ，ＥＥＤＫ志向の各作業者が共通に知らなければならない言語であるので、ＰＤＬ／ＤＴＡの言語規則は少量であること。

（b）ＡＰＤＫ志向作業者がアルゴリズムを記述できること。

（c）ＰＴＤＫ志向作業者が設計変換できること。

（d）ＥＥＤＫ志向作業者が写像する先のインプリメント言語と整合性がよいこと。

ＰＤＬ／ＤＴＡはこれらの要請を考慮して次のように定められた。

（1）設計は一つ以上の「設計モジュール」からなる。各設計モジュールでは「何を参照し、何を定義する」のかを定める。参照されるものを参照変数と呼ぶ。定義されるものを定義変数と呼ぶ。設計モジュールは参照変数と定義変数との間の関数として見ることができる。設計モジュール名（関数名）は唯一でなければならないが、設計変換を考慮して、バージョン番号を各設計モジュールにつける。たとえば、設計モジュールＦはＸを参照してＹを定義するならば、バージョン番号を０とすると、

Ｄｅｓｉｇｎ ｏｆ Ｆ：０；

Ｙ：＝Ｆ（Ｘ）；

のように宣言する。

（2）変数はＣＯＢＯＬやＰＬ／Ｉなどに用いられている構造体の概念を利用する。構造体変数の親子関係は、親変数名＝＝（子変数名リスト）；の形式で宣言し、変数名の引用にあったては、ピリオドを用いて修飾する。たとえば、Ｘ＝＝（Ａ，Ｂ）；と宣言されているときには、Ｘ．Ａ あるいは Ｘ．Ｂ のように用いる。このＸが設計モジュールＦの中で用いられている場合には、「設計モジュールＦのＸのＢの内容」を示すための、設計モジュールから内容への写像関数と解釈する。変数名のスコープは各設計モジュール内で閉じている。変数名リストのならびは順序に意味があり、位置的対応を問題にする。インデックス付きの変数名も許す。その場合は変数名（インデックス）：インデックス＝初端．．終端の形式で表わす。たとえば、Ａ（Ｉ）：Ｉ＝１，Ｍ は、Ａ（１）からＡ（Ｍ）までを示す。Ｍは不定として扱うことができる。インデックスに用いた名前は（例ではＩ）変数名リスト内で閉じている。初端、終端に用いた名前は設計モジュール内で閉じている。

（3）制御構造は、ｂｅｇｉｎ－ｅｎｄブロック、ｉｆ－ｅｌｓｅ、ｗｈｉｌｅ、ｆｏｒおよびｉｆ－ｅｌｓｅにおけるｅｌｓｅが空のものをとくにｗｈｅｎとした５種類に限っている。個々のステートメントは左辺：＝右辺；の形式を用いる。左辺には定義変数リスト、右辺には引用関数名と参照変数リストを書く。引用関数が加減乗除の場合は＋－／＊を用いてインフィックスに記述してよいとする。引用された関数はやはり設計モジュールとして定められていなければならない。個々のステートメントは「その引用関数が右辺の参照変数の内容を参照して、左辺の定義変数の内容を定義する」と解釈される。

7．実例

以下では、初期設計、最終設計、そして実際に作成されたプログラムを例示する。ここにとりあげられた例はＢＩＮＤＥＲプログラムというもので、ラインプリンター出力形式を元の出力イメージを半ページだけずらして再編集するだけの単なるツールである（２つ折りして綴じると製本できて、かつ元の出力イメージが再現きる）。したがってこのプログラムを理解するために必要なＡＰＤＫ，ＰＴＤＫ，ＥＥＤＫはオーバーラップしているので、知識背景の違いを示すには悪い例ではあるが、逆にＡＰＤＩ，ＰＴＤＩ，ＥＥＤＩの違いを比較するには都合がよい。図４に初期設計、図５に最終設計、図６にプログラム（たまたまＦＯＲＴＲＡＮ７７）を示す。

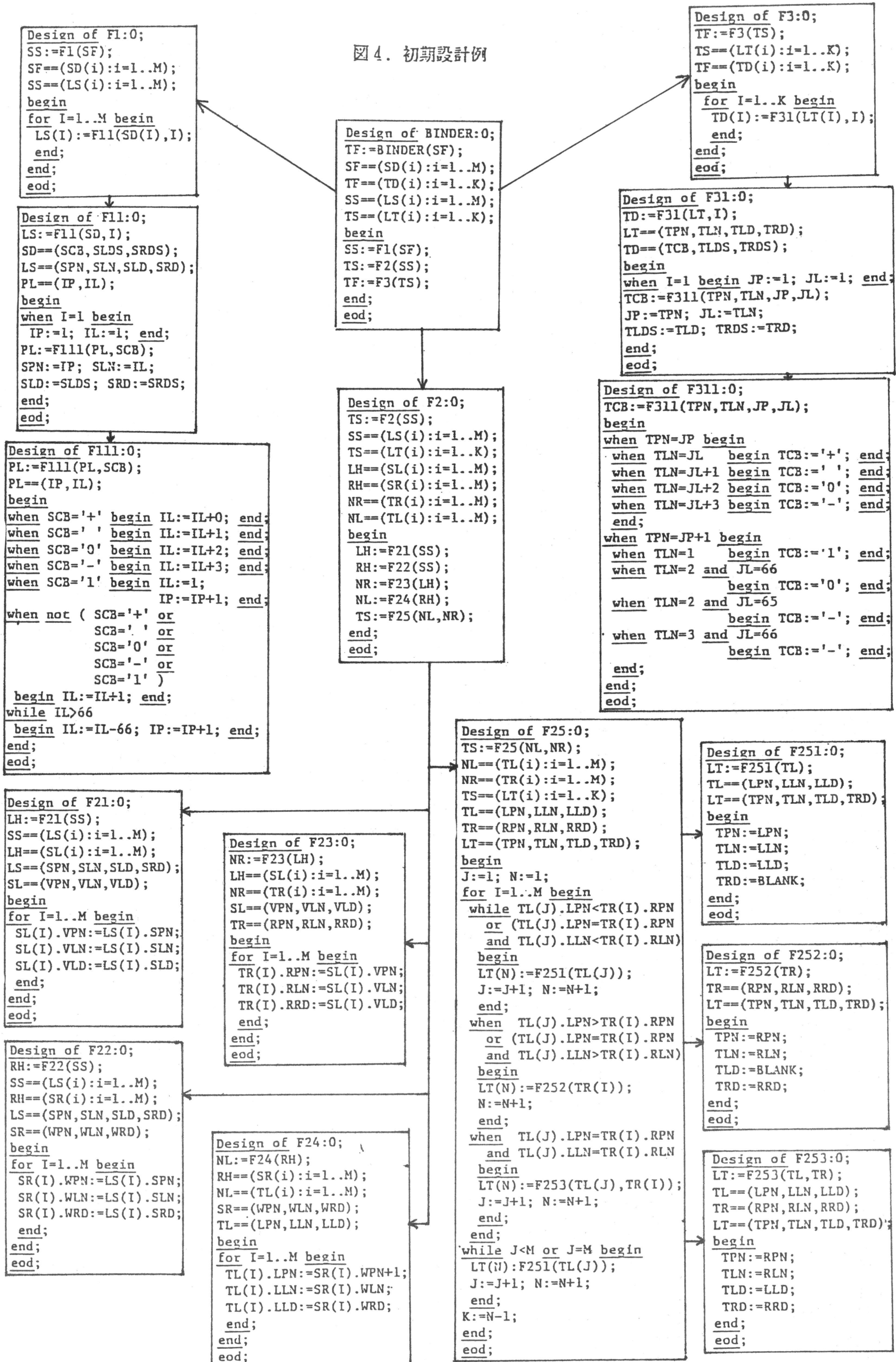

図4. 初期設計例

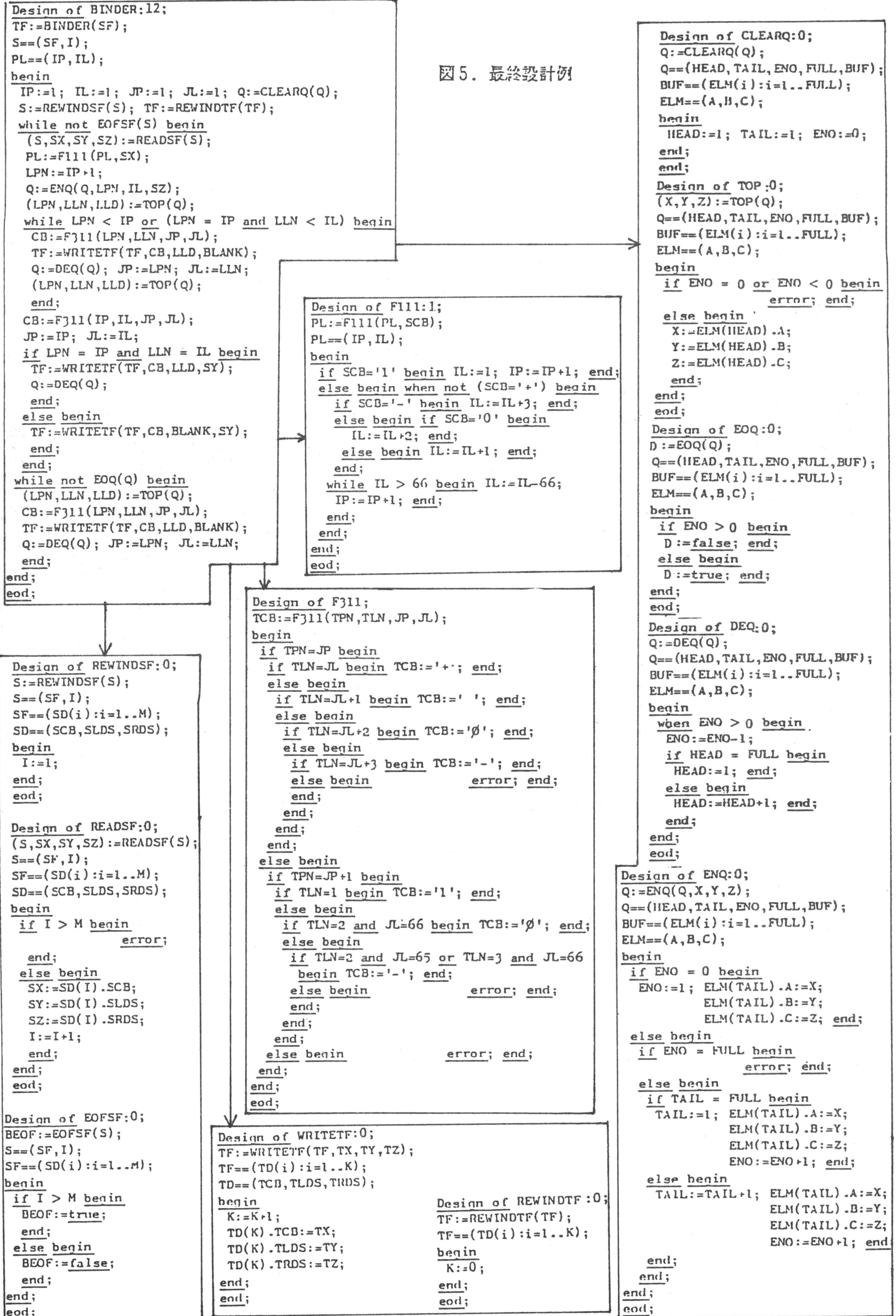

図5. 最終設計例

図6. プログラム例

```
      PROGRAM BINDER
      LOGICAL*1 EMPTY
      CHARACTER SX,CB,SY*68,SZ*68,LLD*68
      CHARACTER BLANK*68
      DATA BLANK(1:34)/'
      DATA BLANK(35:68)/'
      CALL ERRSET(212,256,-1,1)
      CALL ERRSET(219,2,-1,1)
      IP=1
      IL=1
      JP=1
      JL=1
      CALL CLEARQ
      REWIND 1
      REWIND 2
      SY(1:68)=BLANK(1:68)
      SZ(1:68)=BLANK(1:68)
1Ø     READ(1,FMT='(A1,A68,A52,A16)',END=4Ø) SX,SY,SZ(1:52),SZ(53:68)
       CALL SXDEC(IP,IL,SX)
       LPN=IP+1
       CALL ENQ(LPN,IL,SZ)
       CALL TOP(LPN,LLN,LLD)
2Ø      IF(LPN.GT.IP.OR.LPN.EQ.IP.AND.LLN.GE.IL) GO TO 3Ø
         CALL CBENC(CB,LPN,LLN,JP,JL)
         WRITE(2,FMT='(A1,A68,A68)') CB,LLD,BLANK
         CALL DEQ
         JP=LPN
         JL=LLN
         CALL YOP(LPN,LLN,LLD)
         GO TO 2Ø
3Ø      CONTINUE
        CALL CBENC(CB,IP,IL,JP,JL)
        JP=IP
        JL=IL
         IF(LPN.EQ.IP.AND.LLN.EQ.IL) THEN
          WRITE(2,FMT='(A1,A68,A68)') CB,LLD,SY
          CALL DEQ
         ELSE
          WRITE(2,FMT='(A1,A68,A68)') CB,BLANK,SY
         ENDIF
        GO TO 1Ø
4Ø     CONTINUE
       CALL EOQ(EMPTY)
       IF(EMPTY) GO TO 5Ø
        CALL TOP(LPN,LLN,LLD)
        CALL CBENC(CB,LPN,LLN,JP,JL)
        WRITE(2,FMT='(A1,A68,A68)') CB,LLD,BLANK
        CALL DEQ
        JP=LPN
        JL=LLN
       GO TO 4Ø
5Ø    CONTINUE
      STOP
      END
```

```
      SUBROUTINE SXDEC(IP,IL,SCB)
      CHARACTER SCB
      IF(SCB.EQ.'1') THEN
       IL=1
       IP=IP+1
      ELSE
       IF(SCB.NE.'+') THEN
        IF(SCB.EQ.'-') THEN
         IL=IL+3
        ELSE
         IF(SCB.EQ.'Ø') THEN
          IL=IL+2
         ELSE
          IL=IL+1
         ENDIF
1Ø       IF(IL.LE.66) GO TO 2Ø
         IL=IL-66
         IP=IP+1
         GO TO 1Ø
2Ø       CONTINUE
        ENDIF
       ELSE
       ENDIF
      ENDIF
      RETURN
      END
```

```
      SUBROUTINE CBENC(TCB,TPN,TLN,JP,JL)
      CHARACTER TCB
      INTEGER*4 TPN,TLN
      IF(TPN.EQ.JL) THEN
       IF(TLN.EQ.JL) THEN
        TCB='+'
       ELSE
        IF(TLN.EQ.(JL+1)) THEN
         TCB=' '
        ELSE
         IF(TLN.EQ.(JL+2)) THEN
          TCB='Ø'
         ELSE
          IF(TLN.EQ.(JL+3)) THEN
           TCB='-'
          ELSE
           WRITE(6,6)
           STOP
          ENDIF
         ENDIF
        ENDIF
       ENDIF
      ELSE
       IF(TPN.EQ.(JP+1)) THEN
        IF(TLN.EQ.1) THEN
         TCB='1'
        ELSE
         IF(TLN.EQ.2.AND.JL.EQ.66) THEN
          TCB='Ø'
         ELSE
          IF(TLN.EQ.2.AND.JL.EQ.65.OR.
     *        TLN.EQ.3.AND.JL.EQ.66) THEN
           TCB='-'
          ELSE
           WRITE(6,6)
           STOP
          ENDIF
         ENDIF
        ENDIF
       ELSE
        WRITE(6,6)
        STOP
       ENDIF
      ENDIF
      RETURN
    6 FORMAT(1H ,'CONTROL BYTE ERROR')
      END
```

```
      SUBROUTINE CLEARQ
      LOGICAL*1 EMPTY
      INTEGER*4 HEAD,TAIL ENO,FULL
      INTEGER*4 A(264),B(264),X,Y
      CHARACTER C(264)*68,Z*68
       FULL=264
       HEAD=1
       TAIL=1
       ENO=Ø
       RETURN
      ENTRY ENQ(X,Y,Z)
       IF(ENO.EQ.Ø) THEN
        ENO=1
        A(TAIL)=X
        B(TAIL)=Y
        C(TAIL)(1:68)=Z(1:68)
       ELSE
        IF(ENO.EQ.FULL) THEN
         WRITE(6,6)
         STOP
        ELSE
         IF(TAIL.EQ.FULL) THEN
          TAIL=1
          A(TAIL)=X
          B(TAIL)=Y
          C(TAIL)(1:68)=Z(1:68)
          ENO=ENO+1
         ELSE
          TAIL=TAIL+1
          A(TAIL)=X
          B(TAIL)=Y
          C(TAIL)(1:68)=Z(1:68)
          ENO=ENO+1
         ENDIF
        ENDIF
       ENDIF
       RETURN
      ENTRY TOP(X,Y,Z)
       IF(ENO.LE.Ø) THEN
        WRITE(6,6)
        STOP
       ELSE
        X=A(HEAD)
        Y=B(HEAD)
        Z(1:68)=C(HEAD)(1:68)
       ENDIF
       RETURN
      ENTRY DEQ
       IF(ENO.GT.Ø) THEN
        ENO=ENO-1
        IF(HEAD.EQ.FULL) THEN
         HEAD=1
        ELSE
         HEAD=HEAD+1
        ENDIF
       ELSE
       ENDIF
       RETURN

      ENTRY EOQ(EMPTY)
       IF(ENO.GT.Ø) THEN
        EMPTY=.FALSE.
       ELSE
        EMPTY=.TRUE.
       ENDIF
       RETURN
    6 FORMAT(1H ,'MISS-USE OF CLEARQ')
      END
```

8．むすび

本報告では、初期設計技術、設計変換技術、実体化技術、の内容については紙面の都合で述べなかったが、初期設計に関しては、ＡＰＤＩを中心としたモジュールの階層構造化、入出力関係に着目した設計を行えばよく、それぞれの応用分野に通じた者がプログラミングテクニックなどにわずらわされることなく、「参照」、「定義」の概念を用いてすみやかに記述すればよい。設計変換技術に関しては、多くの場合従来からのコンパイラーなどにみられるような最適化技術を用いればよいが、ＱＵＥＵＥやＦＩＬＥ、ＳＴＡＣＫ等のオペレーションクラスターを導入したり、モジュールの再構成を行うなど比較的高級な変換技術も必要である。プログラムの変換技術については、エジンバラ大学〔3〕、ＳＥＴＬグループ〔4〕、ＣＩＰグループ〔5〕、ＳＡＦＥグループ〔6〕などで野心的な研究がなされているが、本報告のような分業体制がらみで設計を変換しようとするアプローチは、著者の知る限りではあまり論じられていない。実体化技術に関しては、重要な技術にもかかわらず、実行環境に依存する個別の話題になることのせいか、表立って論じられていないのではなかろうか。本報告のように、最終設計からプログラム実体への写像という形で実体化技術を研究することが、プログラムの信頼性を確実に向上できる大きな手掛かりになると思われる。

例題として示したＢＩＮＤＥＲ（製本人）プログラムについていうと初期設計は、「ラインプリンター出力用ファイルの構成」と「行制御文字」、およびＰＤＬ／ＤＴＡの知識があれば十分読みこなせる。ＲＥＡＤ文やＷＲＴＥ文など外部システムとのインターフェースはいっさい無い。ＡＰＤＩを読みとるには過不足無い情報が示されている。　設計変換の過程は割愛するが、ＢＩＮＤＥＲプログラムの場合では、１０種類の変換技術が用いられている。　出来上がった最終設計は、ＱＵＥＵＥの概念、　ＦＩＬＥの概念をさらに含んでいる。これらの概念は初期設計にはなかった概念である。ただし最終設計といえども、ＰＤＬ／ＤＴＡによって記述されていることにはかわりなく、ＥＮＱＵＥＵＥ，ＤＥＱＵＥＵＥ，ＲＥＡＤ，ＷＲＩＴＥ等のオペレーションクラスターはしっかり定義されている。プログラム実体はたまたまＦＯＲＴＲＡＮ７７を用いて記述されているがＰＤＬ／ＤＴＡの制御構造はむしろＰＡＳＣＡＬやＡＬＧＯＬ，ＰＬ／Ｉに近いし、変数の構造体扱いはＣＯＢＯＬやＰＬ／Ｉに近い。ＢＩＮＤＥＲプログラムの場合、ＱＵＥＵＥに関するオペレーションクラスターをＥＮＴＲＹによって１つのサブルーチンにされているが、このようにプログラム実体のレベルでもモジュールの再構成がありうる。

今後に残された問題としては、第一に初期設計、最終設計、プログラム実体の３つをどのように維持、管理、発展させていくかであろう。第二は初期設計、設計変換、実体化のそれぞれの作業を支援するシステムの実現である。設計変換アプローチでは応用目的をすみやかに記述しうる、実行環境の変化に対処しやすい、などの側面がある一方、設計変換技術への期待が大きくなるため、設計変換段階の強力なツールが必要になろう。

参考文献

〔1〕小林要：ソフトウエア製品生産における知的分業、情報処理．ＶＯＬ．２１，ＮＯ．１０，１９８０．
〔2〕N．Ｍｕｒａｋａｍｉ：ＳＤＥＭ and ＳＤＳＳ：Ｏｖｅｒａｌｌ Ａｐｐｒｏａｃｈ ｔｏ Ｉｍｐｒｏｖｅｍｅｎｔ ｏｆ Ｓｏｆｔｗａｒｅ ｄｅｖｅｌｏｐｍｅｎｔ Ｅｎｖｉｒｏｎｍｅｎｔ，'Ｓｏｆｔｗａｒｅ Ｅｎｇｉｎｅｅｒｉｎｇ Ｅｎｖｉｒｏｎｍｅｎｔｓ'，１９８１．
〔3〕Ｂｕｒｓｔａｌｌ，Ｒ．Ｍ．，ｅｔ ａｌ．：Ａ ｔｒａｎｓｆｏｒｍａｔｉｏｎ ｓｙｓｔｅｍ ｆｏｒ ｄｅｖｅｌｏｐｉｎｇ ｒｅｃｕｒｓｉｖｅ ｐｒｏｇｒａｍｓ，ＪＡＣＭ ２４，１９７７．
〔4〕Ｄｅａｋ，Ｅ．：Ａ Ｔｒａｎｓｆｏｒｍａ－ｔｉｏｎａｌ Ｄｅｒｉｖａｔｉｏｎ ｏｆ ａ Ｐａｒｓｉｎｇ Ａｌｇｏｒｉｔｈｍ ｉｎ ａ Ｈｉｇｈ Ｌｅｖｅｌ Ｌａｎｇｕａｇｅ．ＩＥＥＥ ＳＥ－７－１，１９８１．
〔5〕Ｂａｕｅｒ，Ｆ．Ｌ．：Ｐｒｏｇｒａｍｍｉｎｇ ａｓ ａｎ ｅｖｏｌｕｔｉｏｎａｒｙ ｐｒｏｃｅｓｓ，２ｎｄ ＩＣＳＥ，１９７６．
〔6〕Ｂａｌｚｅｒ，Ｒ．：Ｔｒａｎｓｆｏｒｍａ－ｔｉｏｎａｌ Ｉｍｐｌｅｍｅｎｔａｔｉｏｎ：Ａｎ Ｅｘａｍｐｌｅ，ＩＥＥＥ ＳＥ－７－１，１９８１．

本 PDF ファイルは 1982 年発行の「第 23 回プログラミング・シンポジウム報告集」をスキャンし、項目ごとに整理して、情報処理学会電子図書館「情報学広場」に掲載するものです。

この出版物は情報処理学会への著作権譲渡がなされていませんが、情報処理学会公式 Web サイトに、下記「過去のプログラミング・シンポジウム報告集の利用許諾について」を掲載し、権利者の捜索をおこないました。そのうえで同意をいただいたもの、お申し出のなかったものを掲載しています。

https://www.ipsj.or.jp/topics/Past_reports.html

過去のプログラミング・シンポジウム報告集の利用許諾について

情報処理学会発行の出版物著作権は平成 12 年から情報処理学会著作権規程に従い、学会に帰属することになっています。

プログラミング・シンポジウムの報告集は、情報処理学会と設立の事情が異なるため、この改訂がシンポジウム内部で徹底しておらず、情報処理学会の他の出版物が情報学広場 (=情報処理学会電子図書館) で公開されているにも拘らず、古い報告集には公開されていないものが少からずありました。

プログラミング・シンポジウムは昭和 59 年に情報処理学会の一部門になりましたが、それ以前の報告集も含め、この度学会の他の出版物と同様の扱いにしたいと考えます。過去のすべての報告集の論文について、著作権者（論文を執筆された故人の相続人）を探し出して利用許諾に関する同意を頂くことは困難ですので、一定期間の権利者捜索の努力をしたうえで、著作権者が見つからない場合も論文を情報学広場に掲載させていただきたいと思います。その後、著作権者が発見され、情報学広場への掲載の継続に同意が得られなかった場合には、当該論文については、掲載を停止致します。

この措置にご意見のある方は、プログラミング・シンポジウムの辻尚史運営委員長 (tsuji@math.s.chiba-u.ac.jp) までお申し出ください。

加えて、著作権者について情報をお持ちの方は事務局まで情報をお寄せくださいますようお願い申し上げます。

期間： 2020 年 12 月 18 日 〜 2021 年 3 月 19 日
掲載日： 2020 年 12 月 18 日

プログラミング・シンポジウム委員会

情報処理学会著作権規程
https://www.ipsj.or.jp/copyright/ronbun/copyright.html