解　説

ソフトウェア製品生産における知的分業†

小　林　　要††

1. まえがき

ソフトウェアは，それが実行されるハードウェアとそれを利用しようとする人間の意図との間に位置づけられる．ソフトウェアを開発するには，ハードウェアを制御する手段や，利用目的を達成するためのデータとアルゴリズム，さらに，ソフトウェアを記述する言語，などの技術知識が必要になる．

ソフトウェアを開発する活動は，こうした知識を集約しながら，与えられた問題に対して，適切な解決手段を提供しようとする問題解決の活動である．活動の中心的役割を演じているのは，人間の創意工夫にもとづいた知的な作業であり，開発されたソフトウェアは，知的成果物であるといえる．

産業構造審議会[1]によれば，知識産業とは，「経済社会全般において生じている知識，情報の効用，および需要の増大に応じて，知識，情報を生産し，提供する産業」であるとされ，情報処理サービス，情報提供サービス，ビデオ産業，教育に関連する機器産業，コンサルティング，システム・エンジニアリングなどのほか，ソフトウェア関連がこれに含まれている[2]．ソフトウェア関連産業を知識産業とみなすならば，ソフトウェアの生産をどのように管理すべきかという観点は，人間の知的な生産活動をどのように管理すべきかという大胆な観点であることに気がつく．

産業構造審議会の人間能力部会(大来佐武郎部会長)は，その中間答申[3]で，知識の生産と流通に携わる人間の労働としての知識労働についてふれ，通商産業省が推進しようとしている産業構造の知識集約化のためには，知識労働者の能力開発が不可欠の条件であると強調した．その指摘によれば，(1)大学などの機関による専門教育への配慮，(2)就業時間の弾力化をはかるなど時間面の障害克服，(3)雑用などの負担の軽減，の三つの対策が必要であるとされている．

ソフトウェア開発においては，定められた納期に向けて，品質の保証された製品を，定められた予算の中で，複数の人間によって効果的に生産することが要請される．したがって，知識労働者個人の能力開発はもちろんのこと，さらに，知識労働を適正な負担のもとに分担して，組織的な効果をねらう必要がある．

知識労働の分担方式については，現在のところ，十分な研究がなされているとは言い難く，ソフトウェア開発における作業分担方式なども定説があるとは思えない．Gunther[4]は，ソフトウェア開発をとらえる視点は時代とともに変ってきたとして，**表-1**のような三つの時代分けを考えた．各々の時代において，つくられる物の名称，開発組織などがどのように特徴づけられるかを示している．このような時代分けが，そのまま，わが国の場合にもあてはまるかは疑問である．しかし，開発対象，開発手段，開発方法などの変遷に伴って，開発組織の構成に対する試行錯誤がなされてきたことは事実である．

技術の進歩は，作業知識の分化をもたらし，専門家の種類も多様化させた．Gibson 等[5]は，**図-1**に示すような四つの段階に時代分けしたうえで，ソフトウェア関連の専門家も，ある流れにそって分化してきたと説明している．こうした専門分化が，歴史的には割に短期間になされてきたことを考慮すれば，ソフトウェア関連の技術移行が，徒弟制度的な伝授によるものではなく，次々に開発される新技術を外からとり入れながら，自己を改革してゆく過程でなされてきたと言えよう．

このようなソフトウェア開発の分野において，知識を効果的に集約し，生産性を確保し，向上せしめるには，知識労働をどのように組織化するべきであろうか．具体的な方式を，思いつきや，その場しのぎ的発想で提案するよりはむしろ，作業分担を考えるための基軸を整理することが先決であろう．本論文では，単

† Division of Intelectual Work in Software Production by Kaname KOBAYASHI (International Institute for Advanced Study of Social Information Science (IIAS-SIS) Fujitsu Limited).

†† 富士通(株)国際情報社会科学研究所

表-1 Gunther[4]による三時代区分

年代	時代名称	作られるもの	開発組織の特徴
1950～1962	プログラミングの時代	計算機プログラム	個人の努力
1963～1971	ソフトウェア開発の時代	ソフトウェア	プロジェクトチーム
1972～	ソフトウェア工学の時代	ソフトウェア製品	チーフプログラマチーム

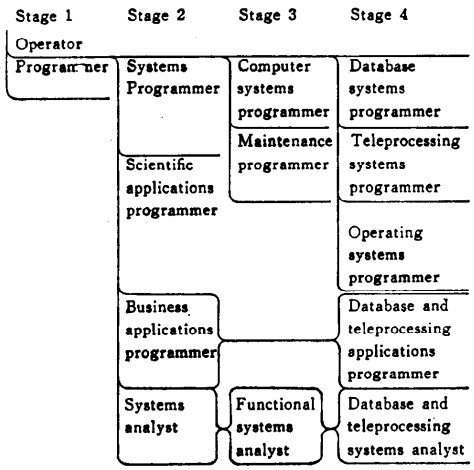

図-1 Gibson 等[5]による専門家の種類の変遷図

純協業の自然発生的分業からマニュファクチャ的分業を経て，近代工業化を実現せしめた要因の一つである分業に着目し，ソフトウェア開発における知識労働の分業を模索する上で必要と思われるいくつかの基軸について整理を試みた．ここでは，知識労働の分業のことを「知的分業」と称するものとする．

2. ソフトウェア開発における知識労働

ソフトウェア開発における知的分業を考える場合，ソフトウェア開発における知識労働の特徴や問題点を，どのように認識するかが大きな問題である．ここでは，知識労働の問題点を，一般的に，かつ網羅的に整理した報告にもとづきながら，ソフトウェア開発の場面における問題を整理してみた．

2.1 知識労働の問題

知識労働の問題点を整理した研究は意外に少ないようであるが，ここでは，松井[6),7)]による整理結果をもとに，ソフトウェア開発作業上の知識労働の特徴と問題を考えることにする．松井（文献6)，p.35）は知識労働の問題として，次の6項目を挙げている．

（1） ヒューマンファクタ中心の仕事

基本的には，各種のサービスを行うことであり，その対象も，またそのプロセスにおいても，ヒューマンファクタに支配される傾向が強い．

（2） 自由と統制の自己管理が必要

知識労働は創意工夫という知的活動を通じてサービスするものであり，その知的活動には自由な発想を助ける自由な雰囲気が必要である．しかし半面では，組織の統制に服する義務があり，自己管理を行う必要がある．

（3） 専門知識のライフサイクル

知識労働の基礎となる専門知識にはライフサイクルがあって，知識が陳腐化するサイクルがある．このため，生涯教育は不可欠となる．

（4） 経験法則優位の環境

企業などの経営組織は，知識労働者がよりどころとする理論が優位の社会ではなく，経験法則が優位の社会となりがちである．

（5） プロフェッショナル・ワークの混在

知識労働者の仕事には，高度な頭脳労働を中心とするプロフェッショナル・ワークと，比較的単純なオペレーショナル・ワークとが混在する．一般に，プロフェッショナル・ワークは管理困難である．

（6） 水平運動の必要性

知識労働者は，マネジャに対するスペシャリストとして位置づけられるが，両者の間には多くの不平等が残存し，平等水平にする運動が必要である．

これらの知識労働上の一般的問題が，ソフトウェア開発のどのような問題と関連があるかについて，次に考察する．

2.2 設計思想の伝達努力

(1)のヒューマンファクタの問題は，作業に対して作業主体の個性が強く反映したり，人と人との意志疎通を悪くする，などの問題に関連がある．とくにソフトウェアの設計段階で抽象的な事柄を対象にする場合，意志疎通が不十分だと，Brooks[8)]の主張する「ソフトウェアの設計思想の統一」が困難になる．

一般に，意図したこととその表現とは，微妙にずれている．表現したものは，意図とはかかわりなく存在でき，さまざまの解釈を許す．また，表現したものは意味が理解できるものであったとしても，どこかに不完全な部分が生じうる．こうした不備を補うためには，本来の目的がどこにあったかという点に立ちかえらねばならない．設計において，本来の目的を位置づける原点は設計思想である．設計思想にもとづけば，

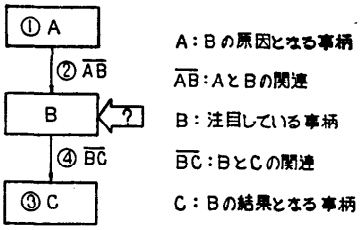

図-2 「なぜBなのか？」に関する項目

人間は高度な推論を展開することによって，不備をかなり補うことができる．

ハードウェアの設計の場合には，物理法則等の基盤の上で推論が可能であり，実世界にたちもどりながら表現上の不備を補うことができる．しかし，ソフトウェアの場合には，対応する法則が人為的であり，「なぜなのか？」を注意深く追跡しなければならない．事柄Bに対し，「なぜBなのか？」を把握するには，図-2 に示すように，少なくとも四つの項目にわたる理解が要請される．それらは．①Bの原因となる事柄Aが何か，②AとBとの関係$\overline{AB}$がどのような理屈で説明されているか，③Bによってもたらされる結果の事柄Cが何か．④BとCとの関係$\overline{BC}$がどのような理屈で説明されているか，などである．

ソフトウェアの設計思想を伝達するには，こうした項目に関する注意深い伝達努力を要する．作業を分担する際の作業分割のしかたとして，伝達努力が比較的少なくて済む切れ目に注目するしかたを考えることは，きわめて大切である．

2.3 作業目標と作業進捗メルクマールの客観性

(2)の自己管理の問題は，時間の使い方や，文書の読み書きのしかた，健康管理など，個人の日常の行動に関する問題である．ソフトウェア開発の場面では，納期があるので，個人の時間の使い方は，組織運営者にとって重大な関心事である．知識労働における生産性の概念は，時間の効果的な使い方と結びつけられることが多い．

ソフトウェアの生産性の尺度として，生産されたプログラムの行数を，生産に要した時間で割った値を用いようとすることがある．しかし，この尺度を生産性と称することは危険なことで，誤解を招きやすい．同一の機能を果すプログラムを，同一の言語で表現した場合でも，実際，倍以上の実効行数（コメント行を除いた行数）で書き表わせるほどの自由度がある．品質の点からは，実効行数の少ないほうが，一般に性能は良く，コンパクトになるので，速やかに全体を見渡すことができるなど，利点が多い．文体改良技術[9]は実効行数を要領良く減らし，悪文を追放して品質を向上させようとする技術である．他の人よりも多くの時間をかけた割には，少ない行数しか生産していない，などときめつけるのは早計で，品質を考慮に入れた生産性尺度を用意することが大切であろう．

ソフトウェア開発の作業は，先に述べたように，問題解決の作業である．したがって，作業工程をあらかじめ細分化しておこうとしても限度がある．日常活動のレベルにおいては，細かい作業目標は作業者自身が設定してゆかねばならない．しかし，自己を管理するためにも，そして納期を守るためにも，客観的な作業目標と作業進捗メルクマールが必要である．

作業が具体的な，コーディング段階などでは，たとえば「コンパイルエラー無し」といった客観的メルクマールがある．作業が比較的抽象的な設計段階においても，何らかの客観的メルクマールが望まれる．そのためには，設計手法と設計手段を定め，客観的に内容をチェックしてくれる設計支援システムを用いることが考えられる．ISDOS[10]や，SSD[11]，あるいは，SREM[12]におけるREVS[13]などは，設計のかなり初期から，内容を自動チェックする支援システムとして実現されている．SDSS[14]，UDDT[15]などは，プログラミング段階における開発支援システムとして用いられている．

自動チェックメカニズムを備えた設計支援システムは，近年，かなり提案され開発されているようであるが，作業を積極的に分担する意図のもとに開発しているとは言い難い．今後に残されている課題ではなかろうか．

2.4 知識集約単位と情報入手環境

(3)の知識陳腐化現象は，知識内容自体の陳腐化という現象と，古い知識以外に新しい知識がさらに必要になってくるという現象との両方を含む．作業の現場において大切なことは，生涯教育もさることながら，作業に必要な知識，情報を，作業者自らの意志によって，いつでも容易に入手し獲得できるような環境を整備しておくことである．

情報入手環境の整備などには，それなりの支援システムを構築し，管理することが望ましい．PRISM[16),17]は，OS (Operating System) の構造情報を内蔵したOS 修正制御情報の管理システムとして，実際に役立った経験を持つ．

ソフトウェアの内容に関する情報を効果的に取り扱うには，プログラムのテキストに注目する従来の取り

扱い方ばかりでなく，図や表，グラフなど視覚に訴える方法も有効であろう．SADT[18]，SARA[19]，R-Graph[20]，SDD[21]などは，設計の段階においてソフトウェアを図面化しようとする意図を持っている．

作業に必要な知識が獲得容易であるためには，作業目的に沿った知識集約がなされていることや，知識のまとまりが，流通させやすい単位になっていることなどが必要である．知識集約の単位として，どのようなものを考えるかによって，知識獲得の効率に違いが生じ，その結果，生産性が左右されると考えられる．ソフトウェア開発に要する知識は，いわゆる学問的知識などの普遍性の高いものに比較すれば，ライフサイクルの短いものが多い．したがって，知識獲得の効率の良い知識集約単位が何であるかを知ることが，より重要である．

2.5 組織編成単位と組織の弾力性

(4)の経験法則優位環境という指摘は，経験に対する評価や考え方に関係した指摘である．経験年数なる尺度に関していえば，ソフトウェアの場合，経験優位な作業と，そうでない作業とがある．プログラムの設計力やデバッグ力は経験年数に比例しないが，コーディング力は経験年数に比例するという報告[22]がある．

その報告ではさらに，作成効率の高いプログラマは信頼性の高いプログラムを作り，作成効率の個人差は4倍以上，信頼性の個人差は10倍以上あったと結論づけている．報告の精度はともかくとして，設計やデバッグなど，抽象性の高い作業では，経験が必ずしも優位でなく，コーディングなど具体性の強い作業において経験が優位であったことがわかる．

Weinberg[23]はプログラミングが知性よりはむしろ，個性や仕事をするときの習慣，あるいは訓練などに依存することを指摘している．設計などでは，機能やデータなどを抽象化して概念を作りあげ，その概念を用いて論理を構成する．したがって，設定された概念に慣れ親しむことが大切であり，自己を訓練する努力が要請される．

しかし，個人の努力にも自ずから限界がある．知識労働の質的向上を個人的な問題に帰着させてしまう考え方は，必ずしも生産的でない．むしろ，組織的効果によって個人の能力をカバーすることが望ましい．設計の段階においては，用いる機能を抽象化した概念があらかじめ整備されていれば，その概念に慣れ，概念を応用する訓練をしておけば，作業を比較的容易に分担できる．しかし，そのような抽象概念のライフサイクルは，ある程度長期のものでないと訓練が有効でない．したがって，比較的長期のライフサイクルを持つ抽象概念を中心とした知識集約単位を，組織単位の基盤とすることが大切である．

知識集約単位は，さまざまの角度から結合できる．組織単位も同様に，さまざまの角度から編成される．部門編成の考えと，プロジェクトチーム編成の考えとは，それぞれ，別な角度からの知識集約方針にもとづいている．さまざまの角度から組織を再編できるだけの組織的弾力性は，進歩の激しいソフトウェアの分野において，常に要請されていると見るべきであろう．

2.6 知識労働における作業の単純化

(5)のプロフェッショナル・ワークの混在に関していうと，プロフェッショナルということばはともかくとして，知識労働の中に，比較的単純な作業と複雑な作業とが混在している事実を考慮する必要がある．ワークサンプリング法によって，あるデータセンタでの70人のプログラマの時間の使い方を観察した報告[24]がある．これによると，「聞く話す読む書く」など，知識や情報の取り扱いに約7割もの時間を消費している．つまり，比較的単純な作業の混在という意味は，知識や情報の取り扱いの活動の中に比較的単純な作業もある，ということになる．

そうした作業としては，外部からの資料収集，書類の蓄積，検索，整理，作表，清書，複写，あるいは事務手続き上の文書作成，など，比較的作業分担が容易なものが考えられる．オフィス・オートメーションの効果には，このような作業をできるだけ，計算機に代行させて，人間はより知的な作業に専念できる，という側面があろう．

一方，ソフトウェア開発におけるプロフェッショナル・ワークは複雑な作業として対比されるが，単純化できる部分は無いであろうか．一般に，作業を単純化するには，作業を細分化する必要がある．作業の細分化は，作業対象の細分化と，作業工程の細分化とによって得られる．細分化と詳細化は類似であり，設計作業における対象の詳細化の過程は，作業の細分化の過程に対応する．

コーディング段階以降をプログラミング作業と呼び，設計はそれ以前に終了しているものとすれば，設計終了を境として，性格が異なる．設計終了までは，ソフトウェアを構成する要素を細分化する分解過程であり設計終了以降は，要素を組み立てて全体とする合成過程である．

合成過程における作業単純化は，作業対象の全貌が既知であるために，作業開始以前に，具体的に行うことができる．しかし，分解過程における作業単純化は，作業対象の全貌が未知であるために，作業開始以前に具体的に行うことが困難である．作業を単純化して分業を実現するためには，設計終了までの分解過程をどのように効率化するかという問題と，合成過程における単純化された作業をいかに効率よく処理するかという問題の二つに分けて，それぞれ考察すべきであろう．

2.7 知識労働に対する理解

(6)の水平運動の必要性の指摘は，知識労働の評価と報酬，および責務と権限に関連し，個人の生き甲斐などと深い関係がある．適正な負担とは何かを考えてみる必要がある．作成効率の高い作業者ばかりに仕事が集中したり，作成効率の低い作業者のために全体の作業が著しく阻害される場合がある．知識労働の内容は外見だけで判断できない．管理者は，製品の開発状況だけに注目することなく，総合的で多元的な視点から評価せざるを得ない．

一般に，作業負担が大きくなると，作業能率は低下する．雑用などを強要して，本来の仕事を中断させた場合，雑用自体は簡単なことであったとしても，一度中断した複雑作業を再開するまでに要する時間の浪費が大きいこともある．したがって，仕事内容の程度以外に，作業を中断させることによって生じる能率低下を，管理者は十分考慮に入れる必要がある．

知識労働者にとって，関連知識の獲得，新技術の導入などは，本質的なエネルギの注入に相当する．設計終了までの分解過程における作業単純化は，知識の効果的な集約によって可能になる．知識集約を徹底して，作業に必要な知識を持ち，知的訓練を行えば，それまで複雑に思えていた作業も，相対的に単純化され，見通しを持って作業できるからである．

したがって，管理者は，知識労働者の関連知識獲得意欲を確保しなくてはならない．また，作業経験を活かすための反省を客観的な形で行うことも，大切な知識集約作業であろう．

3. 知的分業の基軸

ソフトウェア開発における知的分業を考えた動機は，同一の人数で，同じ製品を生産するならば，分業を行うことによって，生産性を向上できるのではなかろうか，という漠然とした期待を抱いたことにある．しかし，知識労働の分業の方式，知識労働における生産性の尺度，および，知的分業による生産性向上の程度，といった事柄に対して，現時点において，確信の持てる解答を与えることが，はたして可能であろうか．このような問題に対するアプローチ方法自体を開拓する必要があるのではなかろうか．

ここでは，次のようなアプローチ方法を考えた．まず，分業の方式についての基軸を用意する．次に，分業によって期待される効果の基軸を整理する．さらにこれらの基軸がソフトウェア開発の場面で効果をあげるには，どのような前提条件があるかを考察する．この場合，生産性の概念は存在すると仮定するが，具体的な定義を保留する．その理由は，分業によって得られる生産性の概念が何であるかを逆に定義できるのではなかろうかと考えたからである．

3.1 時間分業と空間分業

作業を構成する要素として，作業主体，作業対象，作業工程(作業目標，作業手段，作業方法を含む)の三つにしぼって考える．その場合，分業は作業対象と作業工程とを細分し，複数の作業主体達に割り当てて，作業を分担することである．したがって，各作業主体にとってみれば，「自分は何という対象の何の工程を受け持つ」かが明らかになる．分業の方式を区別するために，作業工程を中心に分担する方式と，作業対象を中心に分担する方式とを考える．

作業工程を中心に分担した場合，一つの製品をとってみると，製品を直接扱った期間ごとに作業主体が定まり，「どの期間は誰が担当したか」が明らかになる．これは，製品開発の開始から終了までの時間軸上で作業主体を割り当てることに対応する．この方式を「**時間分業**」と呼ぶことにする．

作業対象を中心に分担した場合，一つの製品をとってみると，製品を構成する部分ごとに作業主体が定まり，「どの部分は誰が担当したか」が明らかになる．これは，製品を構成する空間上で作業主体を割り当てることに対応する．この方式を「**空間分業**」と呼ぶことにする．

たとえば，今仮りに，作業標準 SDEM[14] (Software Development Engineer's Methodology) を借りて，工程を，①計画工程，②設計工程，③プログラミング工程，④テスト工程，⑤評価／保守工程，に分けたとする．このとき，その各々の工程ごとに違った作業主体を割り当てる方式が時間分業に相当する．

また，仮りに，ソフトウェアの構成要素として，④

ユーザインタフェース部分，㈪データベース部分，㈫外部システムとのデータコミュニケーション部分，㈬応用目的達成のデータ処理部分，㈭全体を制御する部分，に分けたとする．このとき，その各々の部分ごとに違った作業主体を割り当てる方式が空間分業に相当する．

作業主体，作業対象，作業工程の各々について規模の概念を導入する．たとえば，上述の時間分業を部門レベルで行って，さらに，各部門の中では，上述の空間分業をグループレベルで行う方式とか，逆に空間分業には部門を，時間分業にグループを対応させる方式など，さまざまの組み合せが考えられる．

3.2 知的分業の効果

Adam Smithの国富論[25]第1篇第1章「分業について」によると，分業の効果は，(a)作業の単純化により，個人の技能が改善される，(b)ある仕事から別な仕事へと手をかえる場合に失われる時間が節約される，(c)作業目標が明解になり，作業手段の向上が図られる，の三点にまとめられる．同じく国富論第1篇第2章「分業をひきおこす原理について」では，分業は人間の本性にひそむ交換という性向から生じるものであると論じている．

(a)の作業単純化は，2.6節で述べたように，もともと比較的単純な作業と，合成過程における具体的作業の単純化と，分解過程における知識集約の効果にもとづく相対的単純化の三種類があった．これらのうち，作業の単純化によって技能改善が期待できるのは，前二者である．後者は個人の技能改善が結果ではなく，原因になる．

(b)の時間の節約は，2.2節の論点から考えると，設計段階など，抽象的な事柄を対象にする場合，作業の分担によって，新たに設計思想の伝達努力を要するため，必ずしも時間の節約にならない．しかし，2.7節で述べたように，雑用などを分担しておけば，本来の作業を中断／再開する時間の浪費を防ぐことができる．

2.5節で述べたような，抽象概念の訓練を行った部門やグループに，その抽象概念と関係の深い仕事を分担すれば，明らかに時間の節約ができる．また2.4節で述べた情報入手環境の整備と分担によって，必要な情報を得るまでの時間が節約できる．知識自体がコンパクトにまとめられていることも時間の節約に効果があるであろう．

(c)の作業目標の明解性は，(a)で述べたことに関連して，もともと比較的単純な作業や，コーディング以降の合成過程における具体的作業において分業を行った場合に効果がある．設計終了以前の分解過程では，2.3節，2.4節で示したような，設計支援システムや図面化ツールを導入するなど，作業目標を明解にする手段を，積極的に用意しなくてはならない．

分業をひきおこす原理としての「交換」概念は，生産された「品物が存在」することを暗示している．情報も交換原理が成立するが，図面や文書など物理的媒体を通して交換することが，分業を確実なものにすると考えられる．ただし，図面化作業や文書化作業には時間がかかるので，文書化努力を多大に要求されると，本来の作業が停滞する恐れもある．文書化作業は，ソフトウェアの内容を決めない限り進まない．したがって，多くの場合，実質的作業の終了時点から，文書化努力が始まる．しかし，仮りに，実質的作業を行う者と，文書化作業を行う者とを分けて，2人が同時に，同じ品物を対象に作業を行えば，実質的作業が文書化作業によって停滞することは少なくなろう．

以上を整理すると，分業による効果は，次のようにまとめられる．

(A) 外部からの資料収集，書類の蓄積，検索，整理，作表，作図，清書，複写，あるいは事務手続き上の文書作成などの作業は，比較的容易に分離でき，作業単純化を進めることによって，技能の改善が期待できる．

(B) 設計終了以降の，コーディングとデバッグ，テスト，評価，保守，などの作業は，ソフトウェアの全貌がすでに知られていることから，作業の目標が明解になり，手段の向上が期待できる．

(C) 設計終了以前の，計画，設計などの作業は，構成要素を詳細化する分解過程であり，作業の見通しがつきにくい．作業の単純化，作業目標の明解化などは，作業者の能力向上，作業手段の整備などの結果，得られるので，作業を分割する考えにもとづく分業は，必ずしも効果があるとは言えない．

(D) 設計終了を境に時間分業を行う場合，設計情報を伝達するための文書や図面が重要な役割を果すが，本来の設計内容を決める作業と，文書化作業とを並行して進めておかないと，時間分業することによる新たな消費時間が表面化する．

3.3 知的分業の条件

前節の結論から，設計を終了するまでの段階における作業が，必ずしも分業によって生産性の向上を期待

できる作業ではない，ということがわかる．しかし，現実には，複数で行わなければ，間に合わないことが多い．したがって，計画段階や設計段階において，何らかの形での分業を行う必要がある．

この場合，空間分業と時間分業とでは，どの点が違うであろうか．設計を空間分業する場合はどうか．ソフトウェアを分割する方法としては，少なくとも二種類考えられる．第1の方法は，システムをいわば縦に分割する方法である．まずシステムをいくつかのサブシステム達に分け，次いで，サブシステムをいくつかのコンポーネント達に分ける．各コンポーネントは，コンパイルの最小単位としてのモジュール達に分解し，モジュール設計完了をもって，設計終了となる．

この縦割り方式によって細分化されるソフトウェアの構成単位を，空間分業の単位とすると，確かに，作業が並行して進められるので，設計終了までの時間が短縮される．HIPO[26]などは，このような縦割り方式の分業に適した道具立てである．しかし，この方式の弱点は，モジュールの設計の段階で，作業主体間の情報交換が減少することである．その結果，本来類似のモジュールであっても，統合されず，冗長なものになる傾向がある．

ソフトウェアを分割する第2の方法は，システムをいわば横に分割する方法である．この方式は，計算機システムを制御する手段に階層を設け，各階層において，それぞれ，計算機システムをモデル化する方式である．各階層ごとに，ある程度首尾一貫したモデルを設定する．各モデルは，それぞれ演算体系を持つ．各階層における設計は，上部階層で使われている演算系を，下部階層のモデルの演算系を用いて表現したものになる．この考え方は，特に新しいものではない．現実に，OS の内部を作成する作業と，その OS を用いた処理プログラムを作成する作業とを分業することができるのは，この横割り方式にもとづいている．

最近は，この横割り方式を，さらに発展させて，階層ごとに抽象化のレベル[27]を，抽象データ型[28]などの導入によって明らかにする方向が模索されている．この方法では，作業目標が比較的明解になり，分業する上では効果があろう．しかし，階層の数を増せば，それだけ，中間のインタフェースが増えるわけで，その分，必要作業が増えることになる．

一方，設計を時間分業する場合はどうか．設計工程は，抽象的な概念を具体的なプログラムで表現する作業であるため，工程を切り分ける軸も抽象的になる．たとえば，概要設計，機能設計，詳細設計であるとか，概念設計，基本設計，あるいは，論理設計，物理設計など，どれも抽象的な表現と言える．

大ざっぱに分けると，ソフトウェアは次の三つの状況の中で存在していると言える．

(C_o) 人間の考える概念や意味の世界におけるソフトウェア (Concepts)

(S_y) 何らかの言語記号で記述された論理世界におけるソフトウェア (Symbolism)

(M_e) 実際に計算機システムの中に組み込まれた世界におけるソフトウェア (Mechanism)

したがって設計工程は，上記 C_o, S_y, M_e のそれぞれの内容を検討し決定する工程であると言えないだろうか．Dijkstra[29]は，「プログラムの正しさと実行コストの二つの問題を分離するべきである」と主張したが，ここでは，S_y のソフトウェアの正しさと，M_e のソフトウェアの実行コストを分離すべきである，という主張に翻訳できよう．添字記法[30]を設計に用いる目的は，S_y の世界で正しく設計されたものを，M_e の世界で，効率良く実行させるために，設計自体を変換しようとする点にある．

しかし，抽象的な基軸で工程を分業することは，一般に，誤解を生じやすく，現状では望ましくないと考えられる．現実には，図-1 で示されるように，空間分業が進んでおり，ソフトウェアの種類別の分業が多くみられる．

チーフプログラマチームにみられる分業は，空間分業に近い．Baker[31]の報告によると，チーフプログラマ，バックアップ・プログラマ，アナリスト（プログラマも兼る），および，プログラマ1名の計4名が，全工程に関与し，全工数の約67% をカバーしている．プログラミング工程で途中加わった他のプログラマ4人の工数は，全体の 13% である．残り 20% は，テクニシャン（ライブラリアン），セクレタリ，マネジャ達の工数である．

チーフプログラマチームの特徴の一つは，ライブラリアンやセクレタリ，マネジャなどが管理作業をかなり分担した点にある．また，もう一つは，少数精鋭のメンバーで，設計思想の統一を図った点にあろう．どちらも，分業の観点から良く理解できる．ただし，この場合のプログラム行数は約 83K であり，対象の規模がこれより大きくなれば，複数のチーフプログラマチームが必要になろう．

空間分業を行う前提条件は，工程に関する技術知識

を，時間分業を行う場合より以上に広い工程範囲にわたって，獲得しなければならないことである．逆に時間分業を行うためには，ソフトウェアを構成するさまざまの要素について，空間分業する場合より以上に，広く知識を持たねばならない．工程に関する必要知識の量と，対象に関する必要知識の量とを比較すると，現在では後者の，ライフサイクルの短い必要知識の量がかなり多いと考えられる．したがって，この点からも，空間分業が，やむを得ず進行していると考えられる．

今後，空間分業を，より積極的に推進しようとするならば，作業対象であるソフトウェアそのものを，相互に独立で，知識集約単位との整合性の良いものに，分解する基軸を研究し開発する必要がある．

4. むすび

知識集約を前提にした知識労働としてのソフトウェア開発作業を分業するためには，空間分業，時間分業，分業規模などの方式分類のもとで，どのような分業基軸があるかを考察した．設計の工程では，設計対象に関する必要知識の量が，工程知識に比較して多いこと，さらに，工程分離の基軸の抽象性，などから，空間分業の傾向が強い現状にあることを示した．

本論文では述べなかったが，ソフトウェア製品保守における知的分業が，もう一方にあって，これと合せることができれば，ソフトウェア製品生産における知的分業の体系ができると考える．

謝辞 本論文発表の機会を与えて下さった東京大学理学部情報科学科國井利泰教授に感謝の意を表します．また終始ご指導下さった富士通国際情報社会科学研究所北川敏男所長に深甚の意を表します．

参考文献

1) 産業構造審議会編：70年代の通商産業政策，大蔵省印刷局 (1971).
2) 安田寿明：知識産業とその市場構造，講座「情報社会科学」第9巻「情報の社会経済学III 知識産業の展開」，学研，pp. 45—218 (1973).
3) 産業構造審議会・人間能力部会：産業構造の知識集約化と人間能力，(中間答申) (1973).
4) Gunther, R. C.: Management methodology for software product engineering, John Wiley and Sons, pp. 4 (1978).
5) Gibson, C. F. and Nolan, R. L.: Managing the four stages of EDP grouth, IEEE Tutorial, Software Management, D. J. Reifer ed., COMPCON '79, pp. 6—18 (Feb. 1979).
6) 松井 好：知識労働者のマネジメント，マネジメント，pp. 31—35 (11月 1973).
7) 松井 好：知識の「生産管理」をどうするか，マネジメント，pp. 113—117 (12月 1973).
8) Brooks, F. P.: The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley (1975). 山内訳：ソフトウェア開発の神話，企画センター，pp. 39—55 (1977).
9) 君島 浩：プログラムの文体の評価・改良，情報処理学会，ソフトウェア工学研究会資料，12—3 (10月 1979).
10) Teichroew, D. and Hershey, E. A.: PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems, IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, pp. 41—48 (Jan. 1977).
11) Uchida, H.: SSD: Software for Structured Development, FUJITSU Scientific and Technical Journal, Vol. 14, No. 2, pp. 129—142 (June 1978).
12) Alford, M.: A requirements engineering methodology for real-time processing requirements, IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, pp. 66—69 (Jan. 1977).
13) Bell, T. E., Bixler, D. C. and Dyer, M. E.: An extendable approach to computer-aided software requirements engineering, IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, pp. 49—60 (Jan. 1977).
14) 富士通株式会社システム開発部：FACOM ソフトウェアエンジニアリング，FACOM ジャーナル，Vol. 3, No. 12, pp. 25—31 (1977).
15) Matsumoto, Y., Yamano, K. and Nakata, I.: UDDT: A practical technique for structured programming and documentation, Proc. 3rd UJCC, pp. 112—116 (1978).
16) 三次 衛，武田 穣：ソフトウェアの保守，ビジネス・コミュニケーション，Vol. 14, No. 4, pp. 66—70 (4月 1977).
17) 辻ケ堂信，君島 浩：大規模オペレーティングシステムの開発管理，電気学会雑誌，Vol. 98, No. 1, pp. 20—23 (1978).
18) Ross, D. T.: Structured Analysis (SA): A Language for Communicating Ideas, IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, pp. 16—34 (Jan. 1977).
19) Estrin, G.: A methodology for the design of digital systems-Supported by SARA at the age of one, AFIPS Conference Proc., Vol. 47, pp. 313—324 (1978).
20) Harada, M., Kunii, T. L. and Saito, M.: RGT: The Recursive Graph Theory as a Theoretical Basis of a System Design Tool DESIGN-TOOL …with an Application to Medical Information

System Design…, Proceedings of the International Symposium on Medical Information System, Osaka, Japan, pp. 503—507 (Oct. 1978).
21) Kanda, Y. and Sugimoto, M.: SDD: Software Diagram Description, Proc. 3rd UJCC, pp. session 9. 3. 1—5 (1978).
22) 田村悦夫：通信制御ソフトウェア開発におけるプログラマの個人差の分析，情報処理学会，ソフトウェア工学研究会資料，3—3（9月 1977）.
23) Weinberg, G. M.: The Psychology of Computer Programming, Van Nostrand Reinhold (1971).
24) Mayer, D. B. and Stalnaker, A. W.: コンピュータ要員の選択と評価，「ソフトウェア生産の技術」，G. F. Weinwurm ed., 吉原賢治編訳，学研，pp. 168—197 (1976).
25) 大河内一男責任編集：国富論，世界の名著 31・アダム・スミス，中央公論社，pp. 59—569 (1968).
26) IBM: HIPO: Design Aid and Documentation Tool, IBM, SR 20-9413-0 (1973).
27) Liskov, B. H.: A design methodology for reliable software systems, Proceedings of the AFIPS, Vol. 41, pp. 191—199 (1972).
28) Liskov, B. and Zilles, S.: Programming with abstract data types, Proceedings of a Symposium on Very High Level Languages, Santa Monica, CA., pp. 50—59 (Mar. 1974).
29) Dijkstra, E. W.: Programming: From Craft to Scientific Discipline, represented for JIPDEC, Oct. 1976, Tokyo, 木村 泉，和田英一共訳「プログラミング——工芸から科学へ」，情報処理，Vol. 18, No. 12, pp. 1248—1256 (Dec. 1977).
30) 小林 要，内田裕士，高橋 浩：添字記法を用いた詳細設計支援ツールと機械チェック可能性について，情報処理学会，第20回全国大会論文集，pp. 331—332 (1979).
31) Baker, F. T.: Chief programmer team management of production programming, IBM System Journal Vol. 11, No. 1, pp. 57—73 (1972).

（昭和55年7月15日受付）