報告

大規模プロジェクトにおけるプログラム開発新技法の適用例*

田坂 宏**

1. まえがき

ストラクチャード・プログラミングが提唱実用化され始めたのは1972年頃からであった．(E. W. ダイクストラ著 "Notes on Structured Programming" (1972年)．H. ミルズ他，ニューヨーク・タイムズ社データ・バンク・システム完成(1972年11月))，それから1年を経た1973年（昭和48年）に当行では，ストラクチャード・プログラミング，トップダウン開発方式，開発援助ライブラリー使用等，一連の新技法を，銀行の主要業務の一つである貸付業務のオンライン化プロジェクトで適用した．これは恐らく，日本で最初の新技法適用による開発の一つであると思われる．

システムは1972年10月の基本設計開始から2年3ヵ月かかって1974年12月に完成した．システムの規模は総開発プログラム・ステップ（インストラクション）数で約353,000ステップ，開発延人員は間接人員を含め約2,000人月（ピーク人員110人）を費した大規模プロジェクトであった．ストラクチャード・プログラミングを使った部分は，そのうちのオンライン・アプリケーション約157,000ステップであったが，オフライン・アプリケーションその他の部分でも，その他の新技法を使い開発を行った．

当時当行にプログラム開発新技法を提唱したのはIBM社のFSD (Federal Systems Division) であったが，その当時新技法の適用の成功例として引き合いに出されたのは，ニューヨーク・タイムズ社データ・バンク・システムのプロジェクトであった．これは10数名で約80,000ステップのシステムを作りあげた中規模プロジェクトである．当行の場合は，チーフ・プログラマー制不採用あるいは，ストラクチャード・プログラミングが部分的適用である等，種々の条件の差はあるが，敢えて完成後のプロジェクトの計数をNYタイムズ・プロジェクトと並べて比較すれば**表-1**のようになる．これを見るとNYタイムズ・プロジェクトの生産性の高さが際立っている．この694ステップ/月/人，(1人1日当り35ステップ)というNYタイムズ・プロジェクトの生産性は，当時旧方式で開発されたものの6倍であったという評価もあるので，IBM社の専門家スタッフによる中規模プロジェクトであるNYタイムズ・プロジェクトに比して，平均経験年数約3年の非専門家スタッフ中心，且つ大規模プロジェクトの当行プロジェクトがそのほぼ1/4の生産性をあげ得たことは，新技法の利点を享受出来たといえるであろう．

表-1 NYタイムズ，三井銀行貸付プロジェクト生産性比較

	YNタイムズ	三井銀行貸付	比較（三井銀行/YNタイムズ）
総開発延人員	120人月	2021人月	16.8倍
（投入人員）	(5～10人)	(80～110人)	
開発ステップ数	83,324	353,060	4.2倍
（同上難易度を易に換算）	(147,186)	(593,150)	(4.0倍)
生産性（1人1月当りステップ数）	649	175	1/3.97

また本番開始後当行システムは機能追加及び変更が多く発生したが，全く新しい担当者が既に出来ているプログラムを追い，機能追加，変更を加える場合にも読み易く論理が把え易いという利点が報告されている．バグの出方も後述のように非常に少ない．保守性，信頼性は，文書化及び標準化の推進（後述のモジュール・サイズの量的制限等）との併用により高められた面もあるが，恩恵の最大の原因は，構造化するための要素（順序・繰返し・分岐）のみを用いて論理の構成がなされているストラクチャード・プログラミングの技法を適用したことによると考えられる．

我々の経験は，大規模プロジェクトでの新技法適用という点に特色があったと思われるので，この点を中心に若干の考察を加えたい．

---

* An Experimental Application of the New Programming Techniques to the Large Scale Programming System by Hiroshi TASAKA (The Mitui Bank Ltd. EDP Administration Division)

** (株)三井銀行事務部

表-2 IPT のプログラム開発諸技法　　(略号)

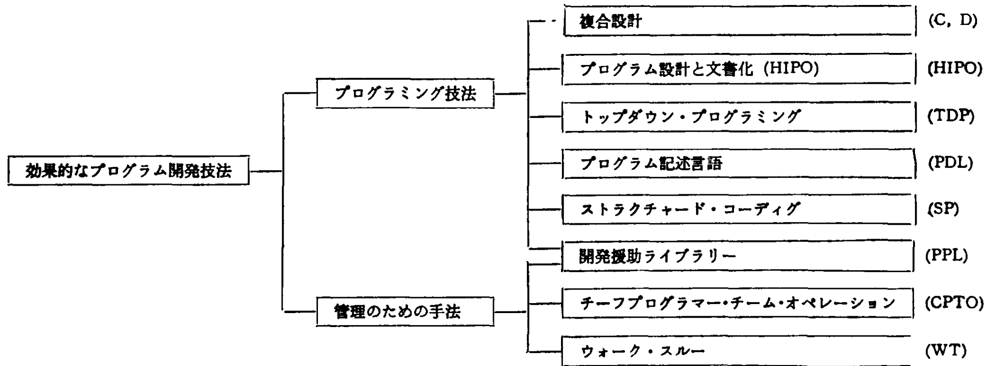

表-3 新技法適用方式及び範囲

技法 / サブシステム	TDP	PDL	SP	PPL	CPTO
コントロール・プログラム	×	×	×	△	×
サポート・プログラム	×	×	×	○	×
オンライン・アプリケーション	△	○	○	○	×
オフライン・アプリケーション	△	×	×	○	×

2. 新技法の各手法と当行プロジェクトでの適用法

プログラム開発の新技法は，これら手法の実用化を早くから手がけてきた IBM 社では技術面での手法と管理面での手法の2つを合わせて，IPT (Improved Programming Technologies) として体系化しており，現在では表-2 のような体系になっている．

当行でプログラム開発技法を最初に適用した時に存在していた技法はこれらのうち，トップダウン・プログラミング (以下 TDP と略す)，プログラム記述言語 (以下 PDL と略す)，ストラクチャード・コーディング(プログラミング) (以下 SP と略す)，開発援助ライブラリー (以下 PPL と略す)，チーフ・プログラマー制 (以下 CPTO と略す) の5つのみであった．従ってこの5技法をシステムの各分野の開発に対応させて採用，不採用を決定した．その結果は表-3 の通りである．

ここでみられる特色は，PPL がほとんどのサブシステムで適用され，CPTO が全く適用されていない点と，オンライン・アプリケーションが CPTO を除くすべての手法を適用している点である．各サブシステムの内容とそれぞれのステップ数は表-4 の通りであるが，採用，不採用は，次のような理由による．

a. PPL の全システムでの採用

表-4 開発プログラム内容及び規模

サブシステム	内容	使用言語	ステップ数
コントロール・プログラム	CICS修正，リカバリー／リスタート，預金インターフェイス，アプリケーションインターフェイス	アセンブラー	24,450
サポート・プログラム	テストドライバー，共通サブプログラム，画面コントロール	アセンブラー 一部 MARK IV	9,830
	移行用プログラム	コボル	29,170
オンライン アプリケーション	起票，照会，登録	アセンブラー	156,900
オフライン アプリケーション	ファイル整理，一括処理，管理資料	コボル	132,700
合計			353,060

SP で書かれていないサブシステムでもモジュール化されていれば PPL は効果を発揮する．最大の利点は，コーディング・テスト期間にプログラマーが，JCL カードの作成，テストラン，ファイリングといった雑務から開放されてプログラミングに専念出来る点である．またカタログド・プロシジャーを使うので，JCL エラー等による時間のロスが少ない点である．PPL による開発効率アップを FSD は5％とみていたが，少くとも5％以上の効果は各サブシステムで得たと確信している．当行ではプログラマー約 30 人に1人の女性ライブラリアンを置いた．全く EDP に関して素人の女性に約2週間の訓練で可能である．またテスト機が開発員の居場所から離れていたため，テストシステムの発送受付窓口の仕事も行わせた．(FSD はプログラマー約10人当りに1人が適当と言っている．)ただ，インストール等の環境設定と使う側の教育のために初期に若干手間がかかることと，ライブラリアンの女性以外に，PPL を司り緊急の際の対処 (PPL 破壊の修復等) が出来る男性を必要としたと

いうことは，使用の際考慮すべき点であろう．当行はプロジェクト内にテストコントロール班を置き，そこに PPL をかなり深く突込んで知っている男性責任者を2名配置した．

b. CPTO の不採用

CPTO の制度は適当な人材が居ない点と大規模プロジェクトでは無理という2点で採用を見送った．FSD の言うチーフ・プログラマーとは，EDP 経験少なくとも8年以上，システム設計及びプログラミングに精通したスーパーマンで，全て仕様書，プログラムを自ら手を下して書くという人間であり，単なるプログラミング・チームのリーダーとは違う．その意味から百人を越す大規模プロジェクトでの適用は疑問がある．また全権力をそこに集中して，バックアップ・プログラマー，秘書的なライブラリアンを置いて仕事を進める環境風土が，アメリカのように日本に合うかも問題がある．FSD も今回の開発には CPTO は実状から見て不採用を勧告している．

c. オンライン・アプリケーションでの SP 及び PDL の採用

SP と PDL をオンライン以外のサブシステムで使わなかった理由は下記の通りである．

コントロール・プログラムは既存のものの修正をして使ったために利用不適当．

サポート・プログラムは他のものに先がけてコーディングを行い完成する必要があったため，とりあえず旧方法で開発．

オフライン・アプリケーションは，プログラマーの経験年数が他のサブシステムに比べて低く（約2.5年）新ソフトウェア多用のため新技法習得ロードを加えることを避け安全度をみた．（テスト・ツールとしての MTS，ファイル・ハンドリングのソフトとして IMS/DL 1 を初使用した．）

COBOL では DO を予約語として使うことはプログラミング上難があって出来なかった．（PERFORM 命令にして文章構造を変えた上で初めて使用可能．）

以上の理由で SP はオンライン以外では不採用としたが，モジュール化とセグメンテーションは全サブシステムで推進した．

d. トップ・ダウン・プログラミングの採用方法について．

開発の順序としては，オンライン・オフライン共トップダウンで行った．即ち上位・下位とモジュールを規定して上位モジュールから順にプログラミングを行った．しかし，単体モジュールのテスト，リンケージ・テスト，全体パッケージのテストと手順を踏んだため，FSD のいう完全なトップダウン・プログラミングではなく，思想をとり入れるに止まったといえる．FSD の言うトップ・ダウンとは，上位モジュールからデザイン，コーディング，テストを行い，上位モジュールのテスト完了後に下位モジュールのデザインを行うことをいう（トップダウン・デザイン）．この方式では，コーディングは期間的に重なる．従って単体テストは，上位と下位のモジュールのリンケージ・テストで行われてしまう．我々は FSD とのプログラマーの技術力の差を考え，モジュール・テストは単体ごとに（原則として上から下の順序で）逐一行った．ただし，下位モジュールが出来ていなくてもそれをダミー・モジュールと見立てて，リンク/リターンを行ってリンケージ・テストを終了させる機能をテスト・ドライバーの中に持ち，それをフルに利用した．

以上4つのことから，新技法を一部のサブシステムで適用しても，その他の旧方式による開発部分とのインターフェイスでは何ら問題なく，新技法の部分的適用が可能であることが立証された．

当時体系化されていなかった複合設計，HIPO，ウォークスルーの3技法については，我々が意識せずにとっていた方法が代替方式となっている．即ち，機能ごとに分解したモジュラー・デザイン，インプット→処理→アウトプットの項目を各機能ごとに明示した機能要求書と基本設計書，詳細設計のチェック等がそれである．この詳細設計のチェックは，プログラムのチェックであるウォークスルーとは全く異なるアプリケーションの機能検証であるが，我々は詳細設計が終了して検証が終るまではアプリケーションのプログラミングには全くとりかからなかった．

3. 開発効率面での効果

3.1 IBM (FSD) 提示の効率アップ

表-5 経験値からみた開発効率アップ率（当行への提示）

適用技法	効率アップ率	採用のもの計	当行への適用
ストラクチャード・プログラミング トップダウン・プログラミング/テスト 開発援助ライブラリー	20% 20% 5%	45%	30%
チーフプログラマー制	20〜40%	今回不採用	今回不採用
全て採用の場合	65〜85%	—	—

表-6 開発生産性テーブル　(単位：人月)

		基本設計	詳細設計	コーディング/ユニットテスト	統合テスト	システムテスト	合計
当行実積値		172 (期間22%)	495 (期間26%)	726 (期間26%)	376 (期間15%)	252 (期間11%)	2,021 (期間100%)
見積値1 (新技法適用せず)		242 (期間15%)	363 (期間15%)	1,331 (期間40%)	484 (期間15%)	193 (期間15%)	2,613 (期間100%)
見積値2 (新技法適用)		208 (間期15%)	312 (期間15%)	1,146 (期間40%)	416 (期間15%)	169 (期間15%)	2,250 (期間100%)
効率比較	実積値/見積値1	0.71 (29%効率アップ)	1.36 (36%効率ダウン)	0.55 (45%効率アップ)	0.78 (22%効率アップ)	1.30 (30%効率ダウン)	0.90 (23%効率アップ)
	実積値/見積値2	0.83 (17%効率アップ)	1.58 (58%効率ダウン)	0.63 (37%効率アップ)	0.90 (10%効率アップ)	1.49 (49%効率ダウン)	0.90 (10%効率アップ)

表-7 J. ARON の方式に基づくマンパワー見積算式

1. 開発期間（基本設計～本番開始）の選択
　(下記では期間1年～2年のものを適用)
2. 必要人員
　標準生産性 Hard なプログラム：125 instructions/月
　　　Medium 〃 ：250 〃
　　　Easy 〃 ：500 〃
　必要直接人員（人月）D=hard instruction数/125+medium instruction 数/250 +easy instruction 数/500
　基本設計～統合テストまでの必要総人員 A=D×2 (注1)
　基本設計～システム・テストまでの必要総人員=A×1.13
3. 期間割振
　Design (基本設計・詳細設計)=30%
　Implementation (コーディング・ユニットテスト)=40%
　Test (統合テスト・システムテスト)=30%
4. 期間別人員割振
　基本設計 A×10% 詳細設計 A×15% コーディング・ユニットテスト A×55% 統合テスト A×20% (以上小計 100%=A)
　システムテスト A×13% (以上合計 A×113%)
5. 生産性テーブル作成
　I. 新技法適用の場合 II. 新技法を適用しない場合 (注2)
(注1) 当行に適用の場合D×2-α とした. (間接人員の一部削減)
(注2) 当行に適用の場合 オンライン・アプリケーション=SP 等の効果30%
　コントロール・プログラムを除く全サブシステム=PPL の効果5%
　開発環境の不利による効率ダウン全サブシステム △25%

FSD は新技法適用によって得られる効果として次の4点をあげた,

- 開発生産性の向上 (Productivity)
- 信頼性の確保 (Reliability)
- 保守の容易性 (Maintenability)
- 管理の容易性 (Manageability)

上記のうち生産性に関しては, FSD は, 総体で当プロジェクトの効率は新技法の適用によって適用したサブシステムごとに5%～30% 上るであろうと提案した. 当時の経験値から見た提案の効率アップ率を表-5 (前頁参照) に示す.

3.2 当行での生産性向上の結果

当行での新技法適用による開発効率向上は, 結果的に表-6 の生産性テーブルのような形で表われた.

生産性の向上は, 開発投入人員が少ないこと及び期間が短かくなることで表わされると考え, 基本設計から完成までの開発延人員（投入Man-month）を基準としている. 上記のテーブルは, IBM(FSD) の J. ARON の "Estimating Resources for Large Programming Systems" という小論文中の算式の結果を見積値として, それと実績を対比させたものである. 以前に類似の規模の適用業務システムの開発を経験していない当行は, 新技法導入以前の基本設計時の見積りの際から, ARON の算式を根拠として人員を割当てていた. (表-7 参照)

たまたま新技法導入の際, 同一算式を使って新技法による生産性向上勘案後の見積りが提示され, 人員を修正して割当てたので（期間は不変), 基本設計から通して生産性テーブルを作ることが可能となった. ただ当時は IPT による開発例が少なく, IPT 適用下のシステム開発のライフサイクルのパターンが確立していなかったため, ARON のライフサイクル・パターンは旧来開発方式に基づいていたが使わざるを得なかった.

以上の事柄を承知の上で開発生産性の比較を行うと, 我々の開発投入の人的資源は新技法を全く利用しない場合に比して23% 少なく（効率向上), 更に新技法を利用した場合の開発効率より10% 良好という結果が出ている. (見積値2には新技法の部分的利用による5～30% の効率向上が見込んである).

その他, 表-6 で特徴的なこととして次の3点が挙げられる.

a. 総体では効率アップになっているが, 詳細設計とシステム・テストでは逆に効率が落ちている. 特に詳細設計でそれが著るしい.

b. 開発各フェイズの期間の割り振りが従来の開発方式のそれと異なっており, 基本設計～詳細設計まで

表-8　コーディング・ユニットテスト進捗度のマイルストーン表

モジュール・コーディングのマイルストーン			単体テストのマイルストーン		
Activity	%	累計%	Activity	%	累計%
1. プログラムデザインの理解	5	5	1. テストデータの生成	10	10
2. 擬似コードのコーディング*	5	10	2. テスト・ドライバーのコントロールカード作成	10	20
3. 擬似コーディングの Review	3	13	3. 最初の成功のケース	15	35
4. 単体テスト計画の完了	10	23	4. 全正常パスのチェック完了	25	60
5. テストデータのコーディング	10	33	5. 全エラーパスのチェック完了	25	85
6. テストの Review	2	35	6. テストの文書化**	10	95
7. プログラムのコーディング	30	65	7. テスト文書の最終 Review	5	100
8. プログラム・コーディングの Review	5	70			
9. 最初の完璧なアセンブリー	20	90			
10. デスクチェックの完了	10	100			

* もしプログラム・デザイン仕様書が構造的コンバージョンが要求されるように書かれていれば10%まで上昇する
** 訂正後のソース及びコメントを含む

に時間がかかっている．(全体の約半分の48% を設計が占める)．逆にコーディング・ユニットテスト期間は見積値の40% に比して26% と極端に短縮化されている．

c.　上述の23% の効率アップという結果が表われたのは，プロジェクトの早期完成という形ではなく，生産したコーディング・ステップ数が投入人員に比して相対的に多いという形としてである．

上記のような結果が出たのは次の理由によると考えられる．まずa の結果は，詳細設計のブレイクダウンの度合が著るしいことに起因すると思われる．我々は詳細設計の範疇に入れたモジュール仕様書をコーディングの寸前まで落して書いた．プログラマーはその仕様書をそのまま擬似コーディングすればよいだけになっていたので，論理的にはモジュール仕様書，擬似コーディング，ソース・プログラムの3者がほぼ1対1で対応したモジュールが多かった．これにより擬似コーディング・シートはプログラム完成後不要になった．

b の結果は，人員投入，期間割振りを古いライフサイクルのパターンに基づいた ARON の方式で行い，それを最後まで変更せずにきたことに起因する．結果的には投入人員のカーブは旧ライフサイクルの型と IPT 下の新ライフサイクルの型との折衷の型となっており，設計段階の人員過少投入，時間の延引をコーディング以後でとり戻したことになった．新技法導入当時から言われた通りの習熟曲線の上昇がみられたが，デザインに時間をかけた方がコーディング時の生産性が急上昇し，トータル効率は良いことが分かったのは大規模プロジェクトへの適用の利点として有益であった．

c の結果は，リソース見積り方法の不慣れと関連する．開発ステップ数は当初見積ったものより大きくなり，途中で機能カットをしたが，当初の計画通りの投入人員で（期間は変えずに）予測以上の規模の開発を行っていたのである．逆に言えば 23% という効率アップがなければ更に多くの機能カット(開発規模縮小)を余儀なくされたであろう．これには SP に伴う冗長性が関連するが，SP で開発するが故のステップ数増は我々の経験では 5〜15% であるという見方である．

いずれにしても開発効率を基本設計から完成までの基準延人員（難易度加味後）の見積値と実績値の比較で表わせば，大規模プロジェクトでは我々並みの 20%〜60%，中規模以下のプロジェクトでは 40〜100% 程度の効率アップが新技法適用の効果として他のプロジェクトでも出てくるのではなかろうか．

3.3　管理容易性にみられる開発面の効果

ここで言う管理容易性とは，主にトップダウン・プログラミングとストラクチャード・コーディングによって，コーディング以降の進捗の把握が確実に出来て狂いがなくなるということである．我々はピーク時で約 70 名のプログラマーに数本のプログラムを同時に持たせて開発を行い，擬似コーディング/コーディング/テスト/デバッグ完了といった節々に進捗度の%をつけ管理を行ったが，各々の積上げの累計が全体の進捗度とほとんど狂わず，平均的なジョブ・ショットの回数で類推が可能であった．また統合テスト，システムテストでは，テスト完了のものの割合で全体の進捗が測れた．これはボトム・アップ方式では出来ないことであろう．即ち，トップモジュールの最後のリンケージが通るまでは 90% 以上の完成度は保障されにくく，全体の進捗率が完成予定の数日前に急に下ることもありうる．しかしトップダウン・プログラミングによって上位のリンケージが保障されているので，プロ

グラミング/テストの終期の完成度が 90～95% に保障されており管理が容易であった．この点は，トップダウン方式の利点として，SP による各モジュールの品質とモジュール間のインターフェイスの固さとの相互効果で，未熟練プログラマーを多くかかえている大規模プロジェクトにとって有効に働いた．（表-8 前頁参照）

4. 保守容易性，信頼性向上の面での効果

前章で挙げた効果は完成までの開発段階で現われるものであるが，保守容易性と信頼性は完成後運用の段階に入って現われる問題である．大規模プロジェクトで開発し保守でも多数の要員を必要とする大きなシステムでは，新技法のこの面での効果が最大の恩恵となる．開発段階の効果と違い，運用段階の効果はシステム完成後，運用を少なくとも1年以上経なければ評価出来ない．当システムは完成後2年以上経ているが，その間で，SP を使用したオンライン部分約16万ステップの範囲での本番後のバグは 10～12 件である．(15,700 ステップ当り1件)．大規模システムに多いインターフェイスによって生ずるミスや行き違い等はほとんどない．このシステムは完成後加えた機能追加と変更が非常に多く，完成後1年間にアプリケーションの全モジュールの 60% に何らの追加・変更を加えたが，このすべての追加・変更と定例メインテナンスを，開発のコーディング/ユニットテスト時の平均人員 103 人の 40% に当る 41 人で行っており，うち SP で書かれた約 16 万ステップの部分は，定例メインテナンスのみでは3人（1人当り5万3千ステップ）でカバーしている．

当行は1モジュール=250 ステップ以内という量的制限の厳守，50 ステップ以内で論理の最小単位をまとめるセグメンテーションの実行を行ったが，論理的に追い易い SP の特性と相まってシステムテストや完成後の保守の際に偉力を発揮した．当行では独自の SP 講習用テキストを作りその第1頁に量的制限のルールを入れて徹底している．必ずしも熟練者ばかりでない多くの要員によって開発，保守される大規模システムでは，特に人間の心理的（視覚的）制約と論理解析能力の制約を考慮する心要がある．

5. 適用面での利点，問題点

5.1 開発各フェイズにおける利用の実際

a. 仕様書作成段階

ユーザ・プランナーの作成した機能要求書より，トップダウン方式による論理的構造の明確なパッケージ仕様書，モジュール仕様書を作成した．パッケージの中をモジュール化して機能別に分類し，モジュール内を更にセグメンテーションした．これらの作業に関しては，業務知識のない仕様書作成者でも比較的論理の明確な仕様書を作成する事が可能となった．この段階では TDP と PDL の技法の一部を適用した．SP のキーワードを仕様書の表現の中でも直接使用し，これによってストラクチャードな論理の構成が可能になった．この段階でみられた利点は，TDP, PDL の採用によってモジュール機能の把握が短期間で可能となった事と，フォーマットのルールが必然的に画一化されるため論理の曖昧さや誤解が非常に少なくなった事である．ただ仕様書作成者が PDL コミットする度合によっては，仕様書を作成しているのかコーディングしているのか不明瞭になる．また同様に，仕様書作成にロードをかけるのかプログラマーのプログラミングのロードに任せるかによって仕様書のレベルに差が出来た．コミットを多くしロードをかけた仕様書は，予約語を使い，桁ずらしも行い，擬似コーディングの中身とほとんど同じく1対1に対応するものとなった．

b. プログラミングのうち擬似コーディング段階

この段階では TDP と PDL の技法を適用して，ストラクチャードなプログラミングとセグメント化を行った．PDL はフローチャートの代替として考え，キーワードを使用してロジックを固めた．ここでは1 in 1 out の原則がとられ，すべての論理は SP の3基本形 (DO UNTIL と CASE を加えれば5基本形) に統一された．その結果プログラム・ソースとほぼ1対1で対応したコーディングとなった．この段階で得られた利点は，簡潔明瞭かつ短時間に当該モジュール機能と論理が判明する点と，セグメント内の修正については，擬似コーディングを当該セグメントについてのみ実施すれば良く，修正ロードが軽いという点であった．一方文字数が多いため，擬似コーディングのための下準備をする必要があり生産性が若干落ちた．また修正のレベルによっては全面的に書き直す必要があり，その点ではフローチャートと変りなくなった．当プロジェクトでは，スケジュールの制約上と修正が生じた場合仕様書と擬似コーディングの両方を保守する必要があるという欠点のため，擬似コーディングシートは完成後，制定文書として残さないことに決定した．SP ではプログラム・ソースリストが擬似コーデ

ィングの形式をもつため，ソース・リストをドキュメントとし得る．

c.　プログラム・ソース・コーディング段階

この段階では標準化 (1 in 1 out，基本型へのロジック統一等) を心掛け，SP マクロを使用しながら擬似コーディング・シートからソース・プログラムを作成した．その結果，完全な GO-TO なしのプログラムになった．利点は次の諸点に表われた．第1にキーワードがそのまま SP マクロ/コントロール・プログラムとのインターフェイス・マクロとなるので，擬似コーディングと同等のレベルで，論理及び機能的に明確なソースが作成可能であった．このマクロの整備によって，使用頻度の高いロジックや誤り易いロジックが簡単になった．第2にプログラマーの能力差に関係なく，全体として画一的なソース・プログラムが作成出来た．以上の諸々の利点から，前述のように，テスト・フェイズ～本番稼動後でもバグは総じて少なかった．

5.2　各技法の短所又は問題点とその改善策

a.　PPL の問題点と改善策

PPL はプログラマーの事務的負担からの解放，イージー・ミス減少による生産性向上の面で効果を上げたが，反面 PPL サポート担当の作業 (PPL 管理体制，バックアップ体制等) が多くなる．また PPL プロシージャーに使いにくいものがあるためユーザが修正しなければならない，という問題点がある．この改善策としては，PPL の教育の徹底，プログラム関連のリスト類のファイリングがあげられる．また当行のような 30 人のプログラマー当り1人のライブラリアンより，10 人に対して1人程度のライブラリアンが適当であろう．

b.　SP の問題点と改善策

プログラミングの標準化によって生産性，信頼性，保守性に富むプログラムが出来る SP も，問題点として，慣れるのに時間がかかる，余分なロジックが増す，修正内容によってはセグメントの再構築を要するという問題点がある．更に冗長性によるプログラム・サイズの膨張 (我々の経験では5%～15%) の問題，プログラマー，特にベテランが，自己の卓抜したプログラミング技術が使えないためにモラールが低下するという問題がある，前半の問題点の対策としては，モジュール仕様書を徹底的に擬似コーディング・レベルまで落すことが良いようである．余分なロジックによる冗長性は，他のメリット (保守容易性，信頼性等) との引き換えであろう．プログラマーのモラール低下の問題については，プログラム論理を組み上げる所では腕が発揮出来る筈との反論がある．

c.　TDP の問題点と改善策

モジュール間のインターフェイスが明確になり，結合テストは単体テストと同時に出来てしまう利点をもつ TDP も，下位モジュールのテストほど雑になる傾向が出るという問題点がある．また，設計を fix していないと変更により下位レベル全てに影響する．これについては，当行が行ったような単体テストによる最少限の確認も一策であろう．適切なモジュール化が TDP のうまくいく条件である．更に TDP をレベルアップして効率よく適用する方法としては，トップ・ダウン・デザインと併用すれば，工程数を減少させることは可能である．その前提は各レベルごとに設計が前もって完全に fix されている必要がある．

6.　今後への提言

大規模プロジェクトでの新技法適用を前提に，今後更に有効な使い方が出来ると思われる点を我々の経験から述べると以下のようになる．

第1に適用のポイントを明確にすることである．即ち開発効率主眼か保守主眼かパフォーマンス主眼かということであるが，それによって新技法の適用の仕方が異なってくる．我々の場合は，メインテナンスでの追加・修正の多いシステムであるとの認識から，他のものは多少目をつぶっても保守性第一義の適用法とした．

第2に新技法の適用に当っては，独自の自社用マニュアルを作成し，独自の教育体系によって要員をまず教育すべきである．演習を含んだ教育がよい．大規模プロジェクトは全員が同じスタートラインに立つことが重要であり，これがうまくゆけばあとは習熟曲線の向上を信用してよい．各技法の採用／不採用の判断や自社の標準化ルールとの組合せをはっきりさせた上で，当該開発担当者以外のサポート・グループに新技法の適用推進をさせる方が効果的と考える．

第3は，我々は利用出来なかったが，同種同規模のプロジェクトでの経験値との対比で開発規模見積りの精度を上げたうえで，新技法適用下でのシステム開発ライフサイクルのパターンに従った開発要員計画を行うとよい．勿論，新技法による効率向上を織り込んでである．1人当り開発効率は個人差はあっても，多人数となれば標準値とは大きなかい離は生じなくなるよ

うである．ただしその前提はほぼ同じ大きさのモジュールにシステムが細分されていることを要する．

第4は，前項の新技法適用下でのシステム開発ライフサイクル曲線（設計に人的資源と期間を多くつぎ込んだ型）を使いスケジュールを進めるならば詳細設計〜モジュール（プログラム）設計の徹底的ブレークダウンと設計の前倒しを行うべきである．設計に時間をやや過重にかけて丁寧に行った方が後が楽になる．

第5は，更に進めて上位下位のモジュールの規定とインターフェイスが固まり，それ以降の設計変更がなければ完全なトップ・ダウンデザインとトップダウン・プログラミングの併用での開発を行うとよい．開発能力に自信があれば，この方法は単体テストとリンケージ・テストが同時に出来，開発期間は早まる．

第6に，開発用ツールとして適用した PPL による外部/内部ライブラリー管理を，我々は本番後も開発済のプログラム管理用ツールとして利用し続けていることを参照に供したい．

最後は学者やコンピュータ・メーカへの願いごとになるが，高級言語用の SP マクロの展開を容易にしてもらえば SP はより使い易くなるであろう．COBOL における DO の容易な利用を始めとして，構造化しにくいと言われている FORTRAN での SP の利用等考慮してほしい．アセンブラー言語の方がストラクチャード・コーディングし易いというのは矛盾のような気がする．

この他，既に述べた当行独自の新技法適用例で成功したと思われるもの，即ちモジュールの大きさの量的制限，制定文書としての擬似コーディング・シートの破棄，標準化の一環としての新技法の利用等も今後新技法の適用を試みるプロジェクトに参考として提言しておきたい．

以上述べた提言の中には SP の本質から外れた適用と言われるものもあるが，最高技術者ばかりを揃えられない大規模プロジェクトでは有効であると思われる．

日本への導入以来改善されてきたプログラム開発新技法は，今後共ますます有効に使われ，開発生産性，保守容易性，信頼性面での実績をふやし続けていくものと思われる．

参 考 文 献

1) J. ARON : Estimating Resourses for Large Programming Systems, (IBM FSD Report 1970)
2) E. W. ダイクストラ，C. ホーア，O. ダール共著「構造化プログラミング」p. 250 サイエンス社，(1975年)
3) E. W. ダイクストラ他「'76 情報化国際講演・討論会 会議録」日本情報処理開発協会，(1977年)

(昭和52年7月1日受付)
(昭和52年9月30日再受付)