Vol. 35 No. 9　　情　報　処　理　　Sep. 1994

解　説

無故障ソフトウェアを開発するための…「クリーンルーム手法」紹介†

佐　藤　和　夫††

1. はじめに

無故障ソフト，つまりユーザからみて故障のないソフトウェアは本当に作れないものであろうか？

「クリーンルーム手法」は，1987年に，ソフトウェアの信頼性を高める革新的な方法論として，当時 IBM フェローであった Mills, H. D. らにより提唱された[2]．最近では，各企業での具体的な適用事例も発表されはじめているが，テスト中に検出される欠陥の平均数はわずか 2.4 個/KLOC☆と報告されている．たとえば，スウェーデンのエルムテル社では35万行の交換機用OS開発に適用したが，全テスト工程中に検出された欠陥は 1.35 個/KLOC と発表している[9]．この数値は従来の一割にも達しない数字であり，その結果，後述するように，本番稼働中の無故障の世界にかなり近づいたことを示している．

「クリーンルーム手法」は IBM のソフトウェア工学の永年の研究の一つの集大成というべきもので，すでによく知られた定評のある技術の集合であるが，運用上のさまざまな工夫がこらされており，プロセスとして特色がある．たとえばチームワークを非常に重視した体制をとるが，これが技術の適用効果をあげて信頼性の向上に重要な役割を果たしている．

開発コストの面からみると，「クリーンルーム手法」は初めて適用したケースでも従来手法とほとんど変わらない，習熟すればまさっているとする報告が多い．さらに，保守コストを含めたソフトウェアのトータルコストで考えると，従来の手法による場合は，50〜70%が実は保守コストという報告もあり，クリーンルーム手法によりトータルではかなりのコスト削減が期待できる．

本稿では，クリーンルーム手法の概要，および三つの適用事例とその効果について紹介する．なお，参考文献を読むときの一助として，クリーンルーム手法で用いる独特の用語の英訳を {英訳} の形で本文中に提供する．

2. クリーンルーム手法はどんな技術か

2.1 クリーンルーム手法のねらい

従来のソフトウェア開発手法では，故障の発生は不可避なもの，故障の原因である欠陥 {Defects} はテストで除去するものと考えられていたようにみえる．そのために，テストに多くの労力を割いており，たとえば IBM 開発製造部門の開発プロセスでは，全体で 12 個の開発局面のうち4個はテストのための局面となっている[3]．おそらく多くのプログラマは一刻も早くコーディングを終了し，テストを開始したいと思っている．欠陥をテストで除去していると，いかにも仕事を精力的にこなしているようにみえる．しかし，この技術には次の三つの大きな問題点がある．

- テストは欠陥を増やしている．

なぜなら，故障の発生原因は仕様（外部設計）や設計（内部設計）に遡る必要がある場合も多いが，「後戻りの長さ」が長ければ長いほど，修正を間違いやすい傾向がある；これが信頼性の悪化をもたらす最大の原因であると言われている[6]．テストで欠陥を見つけるのは，仕様や設計局面で欠陥を見つけるのに比べ，「後戻りの長さ」がはるかに長く，信頼性を悪化させている．

- テストで欠陥を検出する方法は実はコストの高

† Introduction to the Journey to Zero Failures with Cleanroom Software Engineering by Kazuo SATOH (IBM Japan, Ltd., Asia Pacific Technical Operations, Cleanroom Technology Center).

†† 日本アイ・ビー・エム(株)開発製造本部　クリーンルーム技術センター

☆ KLOC の定義は，modified Kilo Line Of Codes without comments である．コンパイル中のシンタックスエラーも含めすべての欠陥を計測している．

くつく技術を選択している.

なぜなら,「後戻りの長さ」が,開発／保守コストの高さに比例する.これは,欠陥の修正コストの経験則 1：10：100 の法則として知られている[5].設計中に検出した欠陥の修正コストを1とすると,テストで修正すると10,本番稼働後に修正すると100倍のコストとなる.

- テストで全欠陥をとるのは不可能である.

なぜなら,テストはプログラムfへ具体的な数値もしくは文字列を入力し,予期した値を得ればfは正しい機能を実現したとする技術である.ところが,たまたま入力した値に対し予期した答えが返ってきても,fが常にすべてのとりうる値に対し正しく動くという証明にはならない.入力可能な値は多くの場合無限近くあるから,すべての場合をつくすことは実用的な時間の長さでは不可能である.

クリーンルーム手法 {Cleanroom Software Engineering}（以下 CR と呼ぶ）は,上記の問題点を抜本的に解決すべく提唱された統合的な手法で,仕様作成から保守までを含んだ方法論である.

図-1 に技術の把握のために,CR の基本的な考え方を,単純化した形ではあるがプロセスからみた切り口で紹介する.

CR では開発チームが,初期仕様局面 (HLS),最終仕様局面 (LLS) および設計・検証局面 (DV) までを担当する.開発チームの仕事は図-1 左に示すとおりであるが,メンバの教育を徹底するしくみを作り,開発中に欠陥を作り込まない,作り込んだ場合は即座に検出し修正することにより,従来に比べ一桁良い非常に高品質のコードを品質保証チームへ渡す.コンパイルからあとは品質保証チームが品質保証局面 (CT) で図-1 右の仕事を担当する.ユーザの使用頻度順に重みを付けたサンプリングテストを実施することにより,ごくわずか残っている欠陥のうち,必要なものを効果的に除去し,本番稼働後の品質を推定することにより,本番稼働時の無故障を保証する.

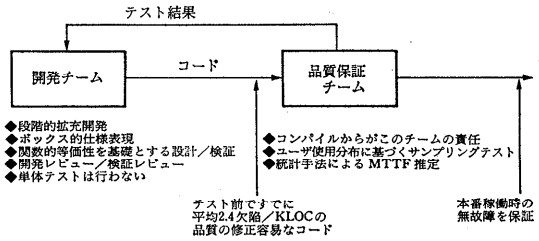


図-1 プロセスからみた CR の考え方

2. の以下の節でおのおのの要素技術について,時系列的に開発の局面に沿って解説する.

2.2 仕様局面における要素技術

2.2.1 ボックス的仕様表現

初期／最終仕様局面は,開発対象の中の「物{object}」に注目し,分析する局面である.開発対象全体を,最初は一つのボックス {Box} として表現し,厳密な逐次詳細化 {Stepwise Refinement} 手続きでもって,トップダウンで仕様から設計までを実施し,最後にはボックスの中に入れ子でたくさんのボックスが存在する形となる.分析の仕方がオブジェクト指向と異なるので,オブジェクト指向でいうオブジェクトをボックスと呼ぶわけである.

3種のボックス構造 {Box-Structure},すなわちブラックボックス,ステートボックスおよびクリアボックスは,開発対象もしくはその一部を,対象の外からみた振舞いからだんだん内部設計へと導く道具である（**図-2** 参照）.ブラックボックスは対象の振舞いを外から見て,対象への刺激 {Stimulus} の履歴に対するボックスの機能の推移 {Transition} および反応 {Response} として記述する.ステートボックスは刺激履歴を,最新の刺激と過去の刺激履歴に分割し,過去分をボック

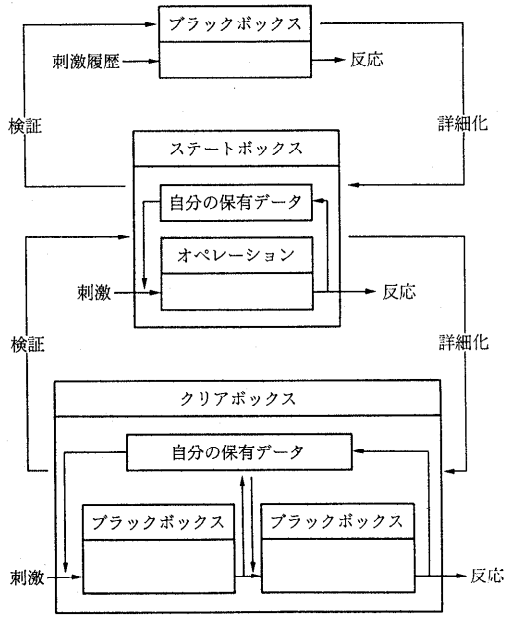


図-2 ボックス構造の詳細化と検証

ス自身の保有データすなわち状態 {State} という形で表現する．クリアボックスはステートボックスを，開発者からみて反応を実現するために必要な手続き，およびより詳細化が必要なブラックボックスへの分解という観点から詳細化する．発見されたブラックボックスはクリアボックスまでさらに詳細化され，もうこれ以上ブラックボックスが発見されなくなると，仕様作成作業は完了となる．この技術のねらいを次に示す．

- 従来作成していた仕様書に比べ，かなり詳細かつぬけのない厳密な仕様を作成する．
- 効果的なレビューが可能なよう，レビューに必要な情報を物理的に近くに置く．以下これを情報カプセル化と呼ぶことにする．
- 段階的拡充をしやすくする．

2.2.2 開発レビュー

仕様のレビューは，開発レビュー {Development Review} と呼ばれる方法で次のように実施される．

- 開発チームの結成

開発対象がある程度詳細化されると，サブシステムごとに開発チームを結成し，それ以降の仕様作成からテスト直前まで緊密に共同作業を実施する．

- レビュー会議の開催

開発チームが作成する仕様は，基本的にはそのチーム全員参加のもとに，逐次詳細化されるたびに，会議でレビューを受ける．

- レビューの方法

情報カプセル化された，レビュー対象ボックス群のみについて，仕様作成者が仕様を「なぜこのように詳細化したか」の観点から説明する．一度のレビューの範囲として，(親)ブラックボックスから一段階詳細化された(子)ブラックボックスまでとし，その他は別のレビュー範囲とする．前にレビューした上位ボックスについては，変更がないかぎりさかのぼり検討はしない．

- 全員理解の原則

親ブラックボックスの記述内容と詳細化された記述が，等価であることをレビューに参加した全員が理解したら，次のレビューに移る．

この技術のねらいを次に示す．

- 仕様を作成する途中で逐次レビューすることにより，「後戻りの長さ」を減らす．
- 情報を局所化して検討することにより，集中力を高めレビューの実効化を図る．
- 説明するためには，頭を理論的に整理する必要があり，説明中に仕様作成者自身がまちがいに気がつく．これによる，開発スキルの向上を図る．
- 開発チーム全員が，サブシステム内部の情報を高度に共有化する．
- そのことにより，他人とのインタフェースの仕様作成の間違いを減らす．

2.2.3 段階的拡充開発

初期仕様局面で，サブシステムが明確になるまで詳細化されたら，開発計画を作成し，開発チームおよび品質保証チームを編成し，チームごとの段階的拡充開発 {Incremental Development} のためのスケジュールを確定する．

CR のプロジェクト管理法は，次のような段階拡充開発プロセスを採用している．

プロジェクトの早い時期から，できるだけサブシステムを統合した，かつシステムの実際に使用される環境のもとでテストを実施できるようにするために，システムのユーザからみた機能を少しずつ段階的に拡充するように計画する．

1. サブシステムを機能からみて，いくつかのグループに分割する．
2. グループごとに，どの開発サイクルで開発するか検討する．
3. 実際の実施計画として，開発チーム，品質保証チームの編成をおこない，開発スケジュールをたてる．

開発サイクルが一つ終了するごとに，システムの利用者に実際に開発途中のシステムを使用させて意見をすいあげることが望ましい．

この技術のねらいを次に示す．

- 開発／品質保証チームに，対象システムに対する理解を早い時期からもってもらうこと．
- できるだけ，テスト・ドライバをつくらないこと，およびテスト・ケースを全サイクルで再利用することによる，生産性の向上を図ること．
- できるだけ，システムのユーザからの反応を早期に得て，開発中の仕様変更による「後戻りの長さ」を短くすること．

2.2.4 チームワークを重視した開発体制

前述の開発チームおよび後述の品質保証チームを（図-1 参照）結成する．大規模なプロジェクトの場合は，複数のチームが同時平行的に段階的拡

充をおこなう．開発チームは開発レビューおよび検証レビューを基本的に全員参加で実施し，開発スキルの向上および情報の共有化を徹底して図る．そのため情報共有化の必要な機能ごとにサブシステムとして開発チームを結成することが望ましい．品質保証チームは信頼性の観点からは，ユーザの使用分布に応じたサンプリングテストを実施するのみであるため，従来よりも必要な人員はかなり少なくてすむ．

この技術のねらいを次に示す．

- 開発レビューをとおした開発スキルの向上を徹底させる．
- チームのなかで情報共有化を徹底することにより，他人とのインタフェースのまちがいを減少させる．
- また，開発チームはコンパイルを許されないため，単体テストで欠陥を取り除くことができず，「背水の陣」でレビューすることになる心理的な効果もある．

2.3 設計・検証局面における要素技術

2.3.1 関数的等価性を基礎とする設計

設計・検証局面を通じて，クリアボックスを逐次詳細化していき最終的にはコードまでおとすのが開発チームの役割である．手続きの設計／コーディングは構造化設計／プログラミングと基本的には同じ方法論を使用する．すなわち，順次処理，条件分岐，ループのみからなる制御構造を用いて，トップダウンに制御構造の連続と入れ子のみを用いて，手続きの逐次詳細化をおこなう．

ただし，CR では，詳細化の必要な制御構造を []= で表し，一段詳細化されたものを []= の直後に，通常一段さげて記述する．たとえば，次のように記述する．

```
[xの絶対値をとってyへ代入する]=
    if x<0 then y := −x else y := x endif
```

関数関係にみたてて，左辺を意図関数 {Intended Function}，右辺を要素プログラム {Prime Program} と呼ぶ．要素プログラムのなかに，次に詳細化すべき意図関数が混じっているのが，設計中ということとなる．設計の途中でも仕様と同様に，たとえばよく使うルーチンなどはブラックボックスとして記述し，ステートボックス，クリアボックスへと逐次詳細化していく．ボックスも []= で表現する．逐次詳細化中も終了後も，ボックス，意図関数はそのまま残しておき，仕様，設計となる．

私見であるが，ステートボックスで検出されたデータの詳細化には，他の方法論たとえば E—R や正規化を併用するとさらに効果的であろう．

この技術のねらいは，サブシステムの仕様，設計，コードがすべて近所にあり，逐次詳細化の歴史が分かり，必要な情報を情報カプセル化することである．このことにより，レビューのしやすい，そして検出された欠陥の修正／変更のしやすい文書作りをねらっている．

2.3.2 開発レビュー

仕様のレビュー(**2.2.2** 参照)と基本的に同じ方法で，逐次詳細化されるたびに開発レビューミーティングが開催される．ただし，設計局面での開発レビューでは，レビュー方法が厳格に決められている．

- 関数的等価性を基礎とする検証の実施

正当性定理 {Correctness Theorem} により，制御構造が順次処理，条件分岐，ループのみで構成されている場合，要素プログラムが意図関数と等しいことを逐次証明すれば，サブシステムの仕様と等価であることが証明される[11]．つまり，局所的に要素プログラムが正しいことを証明することを積み重ねれば，全欠陥を完全に検出することが可能である．このことを利用して，順次処理で一つ，条件分岐で二つ，ループで三つの観点で「式として左辺と右辺が等価か」どうか検証 {Verification} する．これはテストでの証明と異なり必ず有限回で終了する．

- プログラム関数の導出

証明を機械的に実施するために，CR では，要素プログラムからプログラム関数を導出し，それが意図関数と等価かどうかという方法論で証明する．導出の助けとして，トレース・テーブルおよび条件付きトレース・テーブルが提供されている．たとえば先ほどの要素プログラム，

```
if x<0 then y := −x else y := x endif
```

から，プログラム関数

```
[(x<0) -> y := −x|(x>=0)−> y := x]
```

を導出し，それを意図関数と等価かどうか検証する．

なお，これらの検証方法は，仕様のレビューのさいにも使用できる場合がある．

この技術のねらいは，なるべく機械的に作業しながら，欠陥を完全に検出することである．

2.3.3 検証レビュー

検証レビューは，コード完成後に，開発チームメンバ全員参加のもとにコードから仕様まで逆に遡り検証する，レビューミーティングである．基本的には，前節の開発レビューと同じ運営をするが，次の3点が異なる．

- 開発レビューは，逐次詳細化されるたびに，その部分のみをレビューするが，検証レビューは，自分のチームのコード，設計から仕様までを一度にくまなく検証する．
- レビューはコードから始まり，前節の検証の方法論を厳密に適用しながら抽象化してゆく．開発レビューが詳細化の過程をレビューしてゆくのに対して，抽象化の過程をレビューする役割をになっている．
- 全員一致の原則

プログラム関数と意図関数が一致することを，レビューに参加した全員が合意したら，次のレビューに移る．また，全員が合意してから，完成したコードを，コンパイルもせずに品質保証チームへ渡す．

この技術のねらいは，品質保証チームにコードをわたす前の最後の歯止めである．

2.4 品質保証局面における要素技術

2.4.1 ユーザ使用分布に基づくサンプリングテスト

未テスト状態のコードに対し，あらかじめ計画された順序で統計的なランダムサンプリングテストを実施することにより，科学的な信頼性推定が実施できるようになる．開発チームから渡されたコードは，品質評価チームによりはじめてコンパイルされ，テストが開始される．コードはすでにかなり品質の良いものなので，いきなり，従来のプロセスでのテストの最終局面であるシステムテストに相当するテストが可能である．

テストは次の順序で実施する[4]．

1. 各機能ごとのユーザ使用分布を調査する．新規システムの場合は，類似のものから推定する．
2. 使用分布に応じた重みを付けたランダムサンプリングをおこない，テストシナリオを作成する．作成には STGF (Statistical Test case Generation Facility) と呼ばれるツールの使用も可能である．
3. テストシナリオより，具体的なテストケースを作成する．
4. サンプリングで指定された順序でテストを実施する．

この技術のねらいを次に示す．

- 従来のパス／カバレージテストにくらべて，より効果的な，本番稼働で発見されるはずの故障の事前検出率の高いテストケースを作成する．すべての欠陥を除去するのではなく，ユーザがよく使用する部分の無故障を実現しようとする．
- 平均故障発生間隔 MTTF {Mean Time To Failure} の推定が可能なデータを収集する．

2.4.2 統計的手法による MTTF 推定

テスト中に検出された各故障ごとに，前故障からの故障発生間隔を記録していき，本番稼働後 MTTF すなわち，何カ月に一度故障が発生するかの推定をテスト中つねに実施する．推定 MTTF が順調に成長し，予定 MTTF を越えると本番稼働が可能と判定する．逆に推定 MTTF がいつまでも予定 MTTF に近づかないか，4欠陥/KLOC を越えた場合は開発チームへさしもどしになる．推定には CCM (Cleanroom Certification Model) と呼ばれるツールの使用も可能である．

この技術のねらいは，従来よりも正確な，本番稼働後の故障状況の把握である．そのために，本番稼働中の使用状況に極力似せたテストを実施し，そこで発生する故障状況からの統計的な推定がより有意になるように図っている．局面ごとの，各種観点からのテストをじゅうたん爆撃する従来のテスト法では，本番稼働時の故障発生状況との関連は薄い．そのために，テストをいつまで実施すれば十分か？ 検出欠陥数が多ければ本番稼働時に故障が多いのか少なければよいのか？という疑問に答えることは困難であった．

2.5 保守局面における要素技術

本番稼働後に，処理速度改善や機能追加を目的として，システムの変更をおこなう保守は次の手順にしたがう．

1. 変更箇所の発見

仕様，設計，コードのどれかで，変更の必要な箇所を一カ所見つける．処理速度改善の場合は設計，機能追加の場合は仕様のなかで発見することが多いであろう．

2. 最上流変更箇所の特定

変更による影響を見さだめるために，上流の変更の必要性を検討する．すなわち，1. で発見した箇所から設計，仕様と上流へ遡り，それから上流は変更がなく，それから下流は洗い直す必要のある箇所を特定する．

3. 変更必要箇所の特定

最上流変更箇所から下流へ，すべての変更の必要箇所を洗いだす．

その際に，CR を用いた開発がされていない場合は，コードから変更部分の設計あるいは必要に応じ仕様を新たに作成することが必要になる．以下にその場合の方法を述べる．まず，最上流変更箇所からその配下のコードの範囲を特定する．次にコードから上流へ，「関数的等価性を基礎とする設計」および「ボックス的仕様表現」を用いて，逐次詳細化の逆の作業すなわち逐次抽象化{Stepwise Abstraction}をおこない，最上流変更箇所へいたる．

4. 変更の実施

最上流変更箇所から下流に向けて，逐次詳細化の方法で変更する．

CR を用いた開発がされている場合は，上記の方法は簡単に実施できる．なぜなら，CR では仕様，設計，コードは階層構造をして，一つのファイルのなかですべて近所に集まっているため，影響範囲の特定が容易である．

この技術のねらいは，変更による別の箇所への影響をきちんと調べ，変更忘れをなくすることである．

3. 従来の開発プロセスをどう変更すればよいか

3.1 従来の開発プロセスの例—PPA

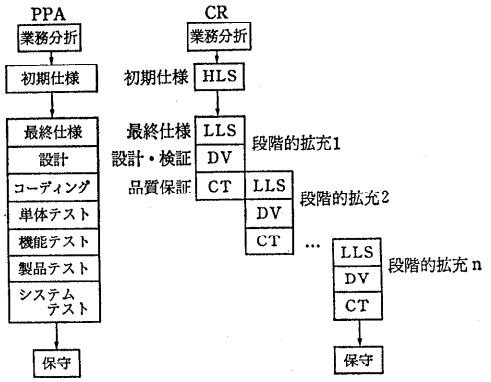
IBM 開発製造部門のソフト開発における伝統的なプロセスは，PPA (Programming Process Architecture) と呼ばれ，いわゆるウォーターフォールモデル（滝型モデルとも呼ぶ）を採用している[3]．このモデルは現在多くの企業の開発プロセスとして採用されている．PPA は**図-3** 左に示すように，仕様作成からテスト終了までの活動を8局面に分け，次の局面に行くまでにその局面の活動は完璧なものとするチェックポイントを置くことで，局面をまたがる後戻りはしない．チェックポイントとして，たとえば仕様レビュー，設計／コードのインスペクション，テストケースレビューなどが実施される．これらのチェックが理想的にゆけば，「後戻りの長さ」は短くてすむはずであるが，現実にはテスト局面が4つも用意されていることで暗に示されているように，レビュー，インスペクションで欠陥を除くことは十分にはできないという前提にたっている．

3.2 PPA からの変更点

一方，CR では，図-3 右に示すように局面の数を4と半分に減らし，しかも段階的拡充プロセスにより，短時間に機能を追加しながら，何回も開発サイクルを回しながら開発する．短期間に開発する必要上，チェックポイントは DV と CT の間に 1 カ所のみで，「検証レビュー」という方法論を用いることとなっている．また，最初のサイクルから，一部であるがシステムが使用可能となり，早期に利用者の意見を聞く局面を設けることが可能となっている．

3.3 変更のねらい

● できるだけ，ユーザからの反応を早期に得て，開発中の仕様変更による後戻りの長さを少なくする．従来は，仕様書にたいする利用者のレビューをすることで，仕様は完璧と考え，あとは，いかに正確に仕様を実現するか，という発想法をとっていた．しかし，ユーザが仕様書を理解することは困難であり，システムが動きはじめてから仕様変更となることがしばしば発生した．



注 HLS: High Level Specification
LLS: Low Level Specification
DV: Design and Verification
CT: CerTification

図-3 PPA と CR プロセスの比較

●開発期間が長期にわたると，その間にユーザの要求が変わることがあるので，それに対応しやすくする．

●開発サイクルを繰り返すことで，開発／品質保証チームに，開発対象に対する理解を早期から深めさせる．従来は，理解したころに開発は終わっていたということになりがちであった．

●開発／レビュー／テストの方法論をより具体的に指定することで，プロジェクト・メンバごとのばらつきを少なくする．

4. 適用事例紹介

4.1 統合診療支援システム

IBM がパッケージソフトとして提供をしている，統合診療支援システム（IDSS）の第2版の一部の開発に CR を採用した．このシステムは，ゲートウェイ，ホストおよびワークステーションからなる大規模システムであるが，診療システムの欠陥の影響は人命におよぶ場合があるので，無故障は重要な開発要件である．このプロジェクトでは，方法論の効果からスキルによる影響を除くために，プロジェクトのコンポーネントを二つに分け，同一メンバが CR と従来の方法論の二つで開発している．その結果，テスト中に検出された欠陥は 2.0 個/KLOC と従来の5分の1であり，たしかにスキルによる差でなく，方法論による差が顕著に認められたと報告している．開発期間は，最初の適用でありかつテストは従来法を用いたにもかかわらず，従来より短かく生産性は高かったと報告している．現在までのところ，客先からの故障発生の報告はない．

4.2 交換機用 OS

スウェーデンの大手通信機器メーカであるエレムテル社は，新 PBX システム用の制御用ソフト（交換機用 OS）を，全面的に CR を採用して新規開発中である[9]．これは35万行におよぶ大規模システムで C および自社開発言語 PLEX—M を用いて開発された．交換機稼働中に故障が発生すると社会的な影響が大きいので，前 OS に比べ44%欠陥を減らしたいということと，開発時の生産性を 30% 向上させたいという目標で開発が開始された．その結果，テスト中に検出された欠陥は 1.35 個/KLOC と 75% の減少であり，生産性は 70% 増加したと報告している．テストの様子から推定すると，ハードウェアを含めた本番稼働中の MTTF は前機種の3倍程度になるかと言って

表-1　適用結果一覧

組　織	完了年	サイズ KLOC	テスト中欠陥/KLOC	本番稼働後欠陥/KLOC	開発時の生産性
IBM—FSC	1987	33	2.3	今までの 1/5	—
IBM—STL	1988	85	3.4	3年で 0.082	740 LOC/PM, x5
NASA	1989	40	4.5	今までの 1/2	780 LOC/PM, x1.8
テネシー大	1990	12	3.0	—	—
Martin Mariotta	1990	1.8	0.0	—	—
IBM—STL	1991	0.6	0.0	—	—
IBM—ASD	1991	107	2.6	—	486 LOC/PM, x1.36
IBM—STL	1991	21.9	2.1	—	—
IBM—PRGS	1991	3.5	0.9	—	—
IBM—Baulder	1992	6.7	5.1	—	—
IBM—FSC	1992	17.8	3.5	—	—
NASA	1992	170	4.2	—	—
IBM—YAMATO	1993	0.7	0.0	—	従来と変わらず
IBM—YAMATO	1993	0.7	5.7	—	従来と変わらず
IBM—YAMATO	1993	1.0	2.0	—	従来より少しよい
IBM—FSC	1993	21.5	2.4	—	—
IBM—PRGS	1993	4.8	0.8	—	—
IBM—FSC	1993	3.0	4.1	—	—
IBM—Tucson	1993	150	1.2	—	—
Ellemtel (Sweden)	1993	350	1.35	—	x1.7

注　PM　人・月
　　xn.m 生産性が従来に比べ，n.m 倍
注意　テスト中欠陥とは，コンパイル開始後検出されたすべての欠陥（コンパイルエラーも含めて）を計測している．

いる.

4.3 意志決定支援システム構築用ツール

IBM により1991年に開発されたパッケージソフトである，AOEXPERT/MVS は全面的に CR を採用して新規開発された[8]．これは AI 技術を用いた，意志決定支援システム構築用プラットフォームの一種である．アセンブラー，PL/X, C, プレゼンテーション・マネージャ，JCL，REXX, PL/I および AI 用の言語である TIRS と実にさまざまな言語を用いて開発された．その結果，テスト中に検出された欠陥は 2.6 個/KLOC であり，生産性は開発前の予測に比べ，36% 上回っていたと報告している．

5. 適用効果はどの程度か

1987年に Mills, H. D. らにより提唱された CR は適用開始後7年目である．今までに効果についての報告があったプロジェクトについて，**表-1** に一覧をのせておく[8),10)]．

信頼性の効果は本来は稼働後の故障実績で確認すべきものであるが，本番稼働後3年程度経過しないと計測できないし，公表されているデータは限られている．そこで，信頼性の代用特性としてテスト中に検出された欠陥/KLOC の欄で効果を見てみると，現在平均 2.4 を下回ったあたりである．従来のわれわれの経験では，30〜60 欠陥/KLOC であるので，CR 開発チームは一桁欠陥の少ないコードが開発できる技術であると考える．CR 品質保証チームが，客先に発見されやすい欠陥については，特に念入りに除去することを考えあわせると，無故障が実現できると考えている．

開発時の生産性についての報告は少ないが，すべての報告で従来よりも生産性は向上している．IBM のクリーンルーム・コンサルタントである Hausler, P. A. によれば，初回の適用でも彼の知るかぎりにおいて生産性は従来以上であると言っている．経験をつめばもっと開発時の生産性は向上するはずである．また，CR は保守性のよい仕様，設計，コードを作成するので (**2.5** 参照)，保守コストの大幅な削減が可能であろう．

これらは，最小3名，最大 73 名のプロジェクトで実施された結果である．システムとしては，マイクロコード，バッチ，分散型，対話型，リアルタイムなど多岐にわたり適用されている．

そのほか，現在 AT&T, Chase Manhatten Bank, Merrill lynch, Prudential (UK), および Sprint & Sterling Software から，CR の適用を開始したという発表がされ，ここにきて急激に適用例が増加している．

6. ま と め

CR は，従来のテスト重点志向のかわりに，設計重点志向をとる．すなわち，コーディング終了までは欠陥を作り込まない努力をし，欠陥を作り込んだ局面で即座に検出することで，ほとんど無欠陥を実現する．さらに，コンパイル直後のコードに対し，ユーザの機能別使用分布に基づくランダムサンプリングしたテストケースを適用することで，本番稼働後の品質を統計理論に基づいて推定することにより，本番稼働時の無故障を科学的に保証しようとする方法論である．

目　的	要素技術
欠陥を作り込まない	● ボックス的仕様表現 ● 段階的拡充開発 ● 関数的等価性を基礎とする設計
欠陥を即座に検出する	● 開発レビュー／検証レビュー（カプセル化された情報に対する，チームによる逐次検証） ● 関数的等価性を基礎とする検証
本番稼働時無故障を保証	● 統計的品質保証（ユーザ使用分布に基づくサンプリングテストと MTTF を使用した信頼性推定）

その結果，次のような成果が得られている．

- 現有の開発要員で，信頼性の顕著な改善がみられ，無故障にかなり近づいた．
- 開発コストははじめての適用でも現状以下ですんだ．

また，次のような効果も予想される．

- なれてくると，生産性はかなり良くなる．
- 保守コストが大幅に減る．すなわち，欠陥の修正コストがなくなり，保守の容易な仕様書，設計書，コードが作れる．

読者の方々も，一度試行してみることをおすすめする．

参 考 文 献

1) Currit, P. A., Dyer, M. and Milles, H. D.: Certifying the Reliability of Software, IEEE Trans. Softw. Eng., pp. 3-11 (Jan. 1986).
2) Mills, H. D., Dyer, M. and Linger, R. C.: Cleanroom Software Engineering, IEEE Software, pp. 19-25 (Sep. 1987).
3) Radice, R. A. and Phillips, R. W.: Software Engineering: An Industrial Approach Volume 1, p. 457, Prentice-Hall, Inc (1988).
4) Mills, H. D.: Certifying the Correctness of Software, Proceeding of 25th Hawaii International Conference on System Sciences, IEEE Computer Society Press, pp. 373-381 (Jan. 1992).
5) The Transformation of IBM: A Market-Driven Quality Reference Guide, IBM Corporation, pp. 22-23 (June 1992).
6) Linger, R. C. and Hausler, P. A.: The Journey to Zero Defects with Cleanroom Software Engineering, Creativityl, Vol. II, No. 3 (Sep. 1992).
7) McGarry, F.: Experimental Software Engineering—17 Years of Lessons in the SEL, Seventeenth Annual Software Engineering Workshop in NASA, pp. 47-49 (Dec. 1992).
8) Hausler, P. A.: A RECENT CLEANROOM SUCCESS STORY: THE REDWING PROJECT, Seventeenth Annual Software Engineering Workshop in NASA, pp. 256-273 (Dec. 1992).
9) Tann, L. G. : OS 32 & Cleanroom, 1st European Industrial Symposium on Cleanroom (Oct. 1993).
10) Hausler, P. A.: Cleanroom Project Results, 私信 (Nov. 02 1993).
11) Linger, R. C., Mills, H. D. and Witt, B. I.: Structured Programming, pp. 229-234, Addison-Wesley Publishing Company (1979).

(平成6年3月15日受付)

佐藤 和夫 (正会員)

1970年関西学院大学理学部物理学科卒業．1972年同大学院理学研究科修士課程修了．同年日本電子計算(株)入社．1984年より日本IBM(株)入社，製品保証を経て，開発製造本部・品質プロセスプログラム担当．現在クリーンルーム技術センター・リーダーとして大和研究所内のクリーンルーム手法の普及活動に従事．1973年バロース研究会最優秀論文賞受賞．本会学会誌編集委員．