特別研究報告

題目

機能実現期間の測定によるプログラマ能力の実験的評価

指導教官
井上 克郎 教授

報告者
三谷 幸久

平成 15 年 2 月 19 日

大阪大学 基礎工学部 情報科学科

平成 14 年度 特別研究報告

機能実現期間の測定によるプログラマ能力の実験的評価

三谷 幸久

内容梗概

プログラマ能力は, プログラム開発の生産性や品質に大きな影響を与えることが指摘されている. プログラム開発における見積もりやスケジューリングを正確に行う上でも, 開発に携わるプログラマの能力を客観的に評価することが求められている. これまでに様々なプログラマ能力の評価方法が提案されている. その中の一つにエラー寿命 (エラーが作り込まれてから除去されるまでの時間) に着目した方法が提案されている. しかし, 能力評価のためのコストや手間が膨大なものとなり, 実際に適用することは難しい. 本研究では, より簡便にプログラマの能力評価を行う手法について提案する. 提案する手法では, まず, 開発すべき機能をチェックリストの形で用意する. チェックリストは, レビューで使用されるチェックリスト, テストデータ定義書,WBS (Work Breakdown Structure) から作成する. 次に, 開発過程から収集したデータに基づいて, 各機能の作成開始時点から作成完了時点までの時間を計測することで, 機能作成全体に要した時間を合計し, 能力を評価するための基本値とする. 提案した評価手法をある企業の新人研修プロジェクトに適用し, 能力評価を行った. 結果, 評価値と研修における成績 (プログラマの素質を間接的に評価するもの) の間に 0.7 の相関が見られ, 評価結果の妥当性を確認した.

主な用語

プログラマ能力
チェックリスト
機能実現時間
実験的評価

目 次

1　まえがき

コンピュータシステムへの依存性が高まるにつれて, その主要な構成要素であるソフトウェアは, ますます大規模化, 複雑化, 多様化している. その一方でソフトウェアの需要量は増大し, 開発期間の短縮化の傾向が強まっている. 結果としてソフトウェアの欠陥や故障が社会に大きな影響を与えることが多くなってきている. このような状況下で, 信頼性や生産性を保ちながらソフトウェアを開発するためには明確な開発計画の下で開発プロセスの工程を系統づけて管理することが不可欠となる. 現実問題として無理に開発期間を短縮することで, ソフトウェアの品質の低下や開発が納期に間に合わないという事例も発生している. このような事例の場合, 特に見積もりの精度が求められているソフトウェア開発の精度の高さに応え切れていないと考えられる.

現実の開発工程では, 新たにソフトウェアを作成するよりも, 既存のソフトウェアの再利用がなされることが多い. その現状において, 開発期間を正確に見積もる上で必要なのは人的資源の正確な管理であると考えられる. 人的資源の重要な属性として, プログラマ能力がある. プログラマ能力は, ソフトウェアの信頼性や開発期間に大きな影響を与えるものであり, その優劣の間には非常に大きな差が存在していることが指摘されている. 現実の開発では納期に間に合わせるために必要な人的資源が確保されているかを確認し, 不足すれば新たに確保するといった措置が取られる. 人的資源の補充ができないのであるならば, 開発期間の見積もりを計算し直す必要がある. また, 人的資源を開発途上で新たに投入すると効率は急激に悪化することが知られており [21] 人的資源の見積もりは, 極力, 初期の段階で見極めるべきであるとされている. 人的資源の見積もりを行う際の問題として, プログラマ一人一人の能力および熟練度には差があり, それらを正確に把握するのは困難である.

このような背景のもとで, プログラマの能力を定量的に評価するためのモデルが提案されている. 代表的なものとして, プログラマが作り込んだエラーの寿命を用いて評価するモデルやプログラマのキーストローク情報を用いて, プログラマのデバッグ作業の効率を求めてプログラマ性能を評価するモデルがある. それぞれのモデルの特徴は 2 節で述べる. どちらのモデルについても, 評価結果はある程度プログラマの能力を表していることが確認されているが, 実際に適用を行う上では, データ収集の手間や解析のコストがかかってしまうという問題がある.

そこで本研究では, より簡便にプログラマの能力評価を行う手法について提案する. 提案する手法では, まず, 開発すべき機能をチェックリストの形で用意する. チェックリストは, レビューで使用されるチェックリスト, テストデータ定義書,WBS (Work Breakdown Structure) から作成する. 次に, 開発過程から収集したデータに基づいて, 各機能の作成開始時点から作成完了時点までの時間を計測することで, 機能作成全体に要した時間を合計し, 能力を評価す

るための基本値とする. 提案した評価手法をある企業の新人研修プロジェクトに適用し, 能力評価を行った. その結果, 評価値と研修における成績 (プログラマの素質を間接的に評価するもの) の間に 0.7 の相関が見られ, 評価結果の妥当性を確認した.

以降, 2 章ではプログラマの能力測定方法について, 既存のモデルを紹介し, その概要, 特徴を分析する. 次に, 3 章では提案するプログラマ能力評価手法について説明する. 4 章では, ある企業の新人研修プロジェクトで実施した提案手法の評価実験と分析結果について述べる. 最後に, 5 章でまとめと今後の課題について述べる.

2　プログラマのデバッグ能力に着目した測定モデル

2.1　プログラマ能力測定の前提

プログラム能力測定の困難さ　ソフトウェアの生産性や信頼性は, プログラマ性能に大きな影響を受けることが指摘されている. プログラマ性能の優劣は, 開発期間やデバッグ作業時間, 開発コストを何倍にも変えてしまう. Sackman らが行った実験ではプログラマ性能の違いによって, 個人の間でのデバッグ作業時間に 20 倍以上の差が存在していたという報告もある [16]. Boehm のソフトウェアの開発コスト予測モデル COCOMO においても, プログラマ能力によって, 予測される開発コストに最大 2.03 倍もの差がある [2]. そういった重要性にも関わらず, プログラマ能力の測定は, 一般的に困難と考えられていた. プログラマ性能とは, 基本的には, プログラマをソフトウェアを生産する 1 つの機械 (装置) をみなした上で, 誤りのない設計, コーディング, 及び, デバッグをいかに効率よく行いうるかを表す尺度である. プログラマ性能は, プログラミング性能, プログラマ生産性, プログラマ能力等と呼ばれることもある. プログラマ能力は, 共通の仕様を対象とするという制約のもとでプログラムを実際に作成する試験を行えば, 作成に要した時間などにより, 能力の定量化が可能である. このような試験には多くの費用が必要となることが知られている.

プログラマとしての経験年数や経験したプロジェクトの回数などを能力の尺度として利用することも考えられた. しかし, そのような尺度はプログラマとしての適性を考慮していないので, 正確な評価とは言えない. 実際の企業では, プロジェクト管理者が, 過去のプロジェクトの中で断片的に観察した作業の様子から直感的に評価を下す事例も存在した. しかし, このような評価は, 評価者の経験や主観性に依存する危険性が高い.

以上の観点から, プログラマの能力を評価するためには, できるだけ制約のない条件のもとで実際の作業の様子を観察すること, 及び, 観察から得られるデータを客観的に分析することが要求される. また, プログラマ能力には技術的な面だけでなく, 適性も確かに存在する. そのため, 適性を評価できるようなモデルが必要とされている.

プログラム能力モデルの実験的評価　経験や主観に左右されずに, 平等な条件, つまりプログラマを経験した年数や経験したプロジェクトの回数や質が均質な集団がモデルの妥当性を測るうえで求められる. そのような集団として, プログラマの教育機関が挙げられる. 例えば学生や企業での新人研修ならば環境的な差異は少なく, モデルの妥当性も正確に測れると考えられる.

Moher と Scheneider の実験結果より, プログラマの「経験 (experience)」と「素質 (aptitude)」がプログラマ性能を決定すると考えられる. 特にプログラマが学生の場合,「経験」は大学で習得したコンピュータサイエンスおよびプログラミングに関する講義の総数で近似的

に測れると考えられている [20]. 一方,「素質」は, それらの講義の成績の平均点で測れると考えている. そして学生や企業での新人の場合は, 経験, すなわち受講したコンピュータサイエンスおよびプログラミングに関する講義の時間数は, ほぼ同じである. 従って, プログラマ性能に差があるなら, それは各学生の素質の差と考えられる.

2.2 エラー寿命に着目した測定モデル [12]

プログラマのデバッグ能力に注目してプログラマ能力を測定する方法として, まずエラー寿命に着目して, プログラム能力を測定する方法が提案された. 開発されたプログラムには高い信頼性が求められていることからデバッグ能力の高いプログラマが高い能力を持っていると考えられる.

基本的アイディア このモデルは, ソフトウェアの開発過程で発生したエラーに関して、次の 2 つの要因に着目してプログラマ性能の定量的評価を行っている.

- 作りこんだエラーの総数
- エラーが作り込まれてから取り除かれるまでの時間

この評価の基本的な考え方は「性能の高いプログラマほど, エラーを作り込まず, 仮にエラーを作り込んだとしても短い期間でそれを取り除くことができる」というものである. このとき, プログラマ性能は各エラーが作り込まれてから取り除かれるまでの時間の総和で表される. そして各エラーが作り込まれてから取り除かれるまでの時間をエラー寿命と定義している.

エラーとフォールトの違い IEEE 標準の定義によると [4], 与えられた問題を解くために必要な情報を理解する過程や, 手法やツールを使用するための思考の過程でプログラマがおかす誤りをエラーと呼ぶ. 一方, エラーがソフトウェア中に具体化したものがフォールトである. ただし, エラーとフォールトは必ずしも 1 対 1 に対応せず, 通常, 一つのエラーが原因で複数のフォールトが発生する (図 1).

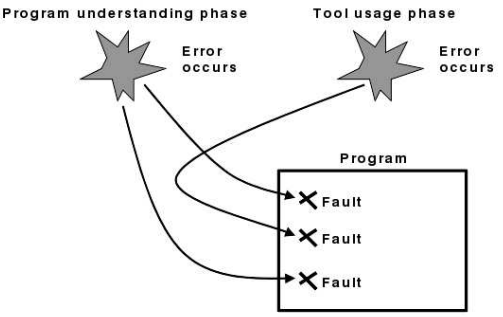

図 1: エラーとフォールとの関係

プログラマ性能を評価するためには, エラーの数を数えることが必要である. しかし, エラーは抽象的なものとして定義されているため, 実際に測定することは非常に困難である. これに対し, フォールトは具体的に測定可能である. そこで IEEE 標準でいうフォールトを測定し, そこから推定した誤りをエラーと呼ぶ. ただし, 明らかに一つの誤りが原因と考えられるフォールトがいくつかある場合には, それらのフォールトを, まとめて一つのエラーとみなす.

エラー寿命　設計, コーディング, デバッグの一連のソフトウェアの開発過程を考える. この各過程においてエラー e_i(i=1,2...) がソフトウェア中に具体化してから, 取り除かれるまでの時間をエラー寿命と定義する. 図 2 にエラー寿命の例を示す. 図で×印はエラーがソフトウェア中に具体化した時刻を, ○印はそのエラーが取り除かれた時刻を示す.

このエラー寿命と同様の概念は, この他にも提案されている. Mills はソフトウェアの品質評価の尺度としてエラー日数を利用している. ここでエラー日数とはエラーがソフトウェ中に作り込まれてから取り除かれるまでの日数の総和である.[20] また,Weiss と Basili はプログラムテキストの変更情報に基づいたソフトウェア開発過程の評価を行っている. そこでソフトウェアシステム中にエラーが存在していた時間の長さが有益な情報になり得ることを指摘している.[20]

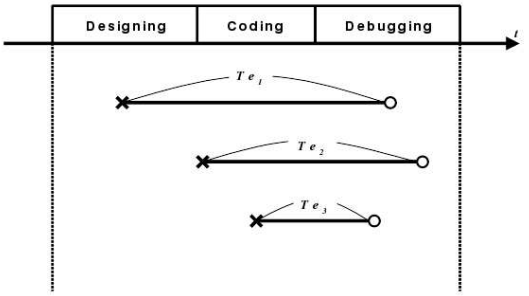

図 2: エラー寿命

重み付けの必要性　プログラマの性能を評価する単純な方法はエラー数を数えることであるが, どのエラーもソフトウェアの生産性や信頼性に同程度の影響を与えるとは考えにくいため, エラー数を数えるだけでは評価として不十分である. エラーが持つ影響度に応じた重み付けが必要となる.

そこで, その重みとしてエラー寿命を用いる. エラー寿命が長ければ, そのエラーに関連するプログラムコードの細部について忘れてしまう可能性が大きい. またエラーが影響を及ぼすコードの量も多くなり, エラー除去に必要な労力も大きくなってしまう. 作業内容も, それだけ複雑になってしまう. 上流工程で混入したエラーほど, 除去に必要な労力が大きいことは指摘されている.

エラー寿命による重み付けがプログラマ性能の評価に有用であるかは, プログラマ性能と直接的に対応していると考えられる, プログラム作成に要した時間 (総端末使用時間) との相関を見ることで確かめられる. 実験データに基づいてエラー寿命の総和と総端末使用時間の比較を行った結果, 両者の間の相関係数の値は 0.82 であった.[12] また, エラー数と総端末使用時間の比較を行った所, 両者の間の相関係数の値は 0.45 であった. これらのことから考えて, 単純にエラー数を評価するよりも, エラー寿命による重み付けを行って評価した方が, プログラム作成の効率を良く表現していると考えられる.

プログラマ性能の評価尺度 s　この考えに基づき, プログラマ性能の評価尺度 s を次のように定義する. まず, エラー寿命 T_e は与えられた問題の難しさ p にも依存すると考える. すなわち問題が難しいほどプログラマの開発作業は複雑になりエラーは発生しやすく, エラー寿命も長くなると仮定する. さらに性能の高いプログラマほど s の値が大きくなるように逆数を用いて次式のように定める.

$s = (\text{エラー寿命 } T_e \text{の総和}/f(p))^{-1}$

ただし,

f :正規化関数

p :与えられた問題の難しさ

とする.

この s の定義では, 開発すべきプログラムの要求仕様に基づき, プログラマが一人で, 設計, コーディングからテスト, デバッグまで行うことを前提としている. 更に, プログラムはあらかじめ定められた期限までに完成するものとし, その期間内に要求仕様は変更されないものとする.

評価値の算出方法　エラー寿命の測定の実験では, プログラムテキストが変更される度に, プログラマに, その変更理由を端末から入力してもらった. プログラマが実験中に作成したプログラムテキストの変更情報 (変更時刻, 変更箇所, 変更理由) を, 人手を介して分析してエラー寿命 T_e を求めた. そのため, プログラムテキスト中に具体化しなかったエラー (例えば, コーディング以前に発見されて取り除かれたエラーなど) はエラー e に含めない. また, シンタックスエラーをエラー e に含めていない.

エラー寿命 T_e を測る時間の単位としてはプログラム作成者が端末を操作していた時間 (端末使用時間) の累積を測定する. 厳密に言えば端末使用時間はプログラム開発に費やした実際の時間を表さない. しかし端末使用時間が自動収集可能なデータであること, 及び, 机上で考えた上で端末を操作するプログラマーほどエラー寿命の評価が有利になることを踏まえて, 端末使用時間を使用することにした.

エラー寿命 T_e の総和は次式のように書き換えられる. ただし,avg はエラー寿命 T_e の平均値,N はエラーの総数である.

$$\sum T_e = avg \cdot N$$

ここで,avg と,N は共に問題の難しさ p に依存すると考えられる. 従って, avg の正規化関数 $f_1(p)$ と N の正規化関数 $f_2(p)$ を導入して, 以下の式を求める.

$$\frac{\sum T_e}{f(p)} = \frac{avg}{f_1(p)} \cdot \frac{N}{f_2(p)}$$

ただし

$$f(p) = f_1(p) \cdot f_2(p)$$

とする.

各実験において各作成者が用いたアルゴリズムやデータ構造は, ほぼ同じである. また, 問題の難しさ p も各作成者の間で大きな差はない. そこで,p を実験終了時のプログラムサイズ (行数)L で近似的に表すことにする. 更に avg,N は共に p に同程度依存すると仮定する. 以上により, ここでは, 単純に

$$f_1(p) = f_2(p) = p \fallingdotseq L$$

$$f(p) = f_1(p) \cdot f_2(p) \fallingdotseq L^2$$

と置くことにする.

今回の評価値の算出方法では実験終了時のプログラムサイズで正規化を行っている. これには,

- 与えられた問題の難しさに大きな差がない
- プログラム中で用いるアルゴリズムやデータ構造が, ほぼ同じである

ということが仮定されている.

従って, 例えば簡単なファイル処理を実行する事務処理と複雑な処理を実行するシステムプログラム (OS, コンパイル) の場合のように比較すべきプログラムの間でアルゴリズムやデータ構造の複雑さに大きな差が存在するときには, その点を十分に考慮した正規化が必要になる.

また, プログラム中に多くのエラーが残ったままでは精度の高い評価値は期待できない. そこで, テストの手法, 残存エラー数, 等も考慮すべきである.

評価　こうした尺度で測定されたプログラマ評価値は, 実際のプログラム能力 (プログラマの成績) と高い相関を示した. このことからエラー寿命からプログラマ能力を判定できる.

モデルの特徴　プログラムの作成過程を追うことでエラーの寿命を測定しているが, そのエラー寿命の測定に多くの労力と時間が必要になってしまう. しかしエラー寿命を追うことでプログラムの信頼性も考慮することもできる. また, このモデルは, 最終的な成果でなく, 作成過程を時系列に追うことがプログラマ能力を測定する上で重要な役割を果たしていることを示している.

2.3 プログラマのデバッグ能力をキーストロークから測定する方法 [18]

ここで言われているデバッグ能力とは 1 個の故障 (software failure) を発見してから, その原因を修正するまでの時間的効率である. プログラミングとは修正の積み重ねと見ることができるので, デバッグ能力を測定することは重要な意味を持つ.

基本的なアイディア この方法では, プログラマの作業を観察するために, キーストロークに着目する. プログラマの作業の大部分は計算機に向かって行われるので, キーストロークの観察は有効な手段だと考えられる. プログラマの入力する全ての命令をキーストロークから解析し, その結果から各時刻におけるプログラマの活動を分類する. 特にデバッグ能力に注目しているので, 最初のコンパイルから以降の活動を数時間にわたって分類する. 分類の結果, デバッグ作業中のプログラマ活動の時系列 (プログラマ活動系列) が秒間隔で得られる. なお心拍などの生理的情報からプログラマの様子を観察することも試みられたが, プログラム能力を評価するまでの詳細なデータは得られていない.

熟練者と非熟練者の特徴を抽出するために, キーストロークから得られるプログラマの活動系列に対して, 更なる分析を行う. その分析を行う手段として, プログラマの確率的な状態遷移モデルを導入する. このプログラマモデルは活動系列がマルコフ過程であると仮定している.[18] マルコフ性を仮定しても実際の活動を十分に表現できるし, マルコフ性を仮定することによって, 分析が容易になってくるからである. 状態の個数や状態遷移の規則は, 活動系列を実際に観測した結果に基づいて決定された. 熟練者と非熟練者の特徴は遷移確率などのモデルのパラメータにおける差として表される. 観測される活動系列から, これらのパラメータを推定することにより, プログラマの特徴を抽出する. このプログラマモデルのパラメータから,1 個の欠陥 (バグ,software fault) の修正に要する時間の推定値をはじめとする 3 種類の評価値が導かれる.

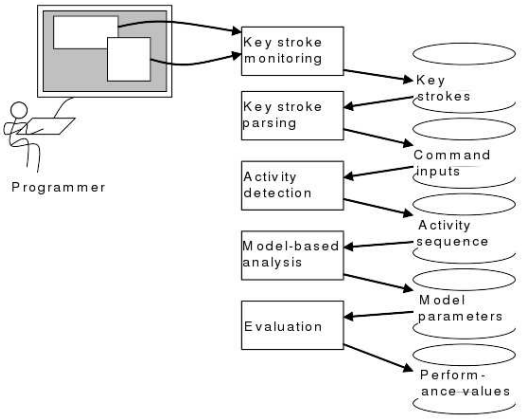

図 3: 方法の概略

測定方法　この方法では,1 人のプログラマがデバッグ作業を続けている様子を数時間に渡って観察, 分析する.1 人のプログラマが 1 個のプログラムを作成する場合には, 最初のコンパイルから以降の作業を観察する. 図のように観察, 分析の手順は 5 ステップからなる. なおステップ 1 から 3 までは測定ツールによって自動的に実行が可能である. またステップ 4，5 もツールによる自動化が可能である.

ステップ 1 キーストロークを監視する. 画面上に複数のウィンドウを開くことができる. プログラミング環境でも, 各ウィンドウへのキーストロークを全て記録する. なお, そのような環境では, 一般にマウスの操作による文字列の複写と貼込みが行われる. そのような文字列の張込みもキーストロークと同様に記録する.

ステップ 2 記録したキーストロークを解析することにより, 入力された命令 (comannd) と, その時刻を検出する. ここで言われている命令とは, プログラムやツールを起動するための命令 (シェル命令) だけでなく, 各プログラムやツールの内部の命令を含む. 例えばエディタの文字列の挿入命令や削除命令も検出する.

ステップ 3 デバッグ作業中の各時刻におけるプログラムの活動を, 入力された命令に基づいて分類する. このときのプログラマの活動はコンパイル, プログラム実行, プログラム変更の 3 種類に分類できる. このステップの出力は, それらの 3 種類の活動の秒間隔の時系列である. これをプログラマ活動時系列 (programmer activity sequence) と呼ぶ.

ステップ 4 プログラマモデルを時系列に適合させる. つまり観測された活動系列からモデル

のパラメータを推定する. これらのパラメータによりプログラマの特徴が表される. なお, パラメータを高精度で推定するためには, 活動系列が長い方が良い.

ステップ 5 推定したモデルのパラメータから, プログラムの評価値を算出する.1 回のプログラム変更で欠陥を修正できない確率,1 回のプログラム変更に要する時間, および,1 個の欠陥の修正に要する時間の 3 種類を出力する.

外部からの観測によりプログラマの活動がどのように分類できるかを調べるためにビデオカメラを使用した予備実験を行った. 結果として, デバッグ作業中の各時刻におけるプログラマの活動がコンパイル・プログラム実行・プログラム変更の 3 種類に分類できることが分かった.

デバッグ中の作業状態　提案する方法では, キーストロークを利用して, 各時刻のプログラマの活動を上述の 3 種類に分類する. その為に, ステップ 3 では入力される全ての命令の中から各活動の開始時, および, 終了時に入力される命令を判別する. 以下では, 各活動の定義と具体的な判別方法を与える.

コンパイル ソースプログラムから実行可能ファイルを作成する間の活動である. コンパイラの起動だけでなく, ソースプログラム中の文法誤りをエディタにより除去する活動を含む. この活動の開始はプログラムの起動命令により判定する.(例:cc,gcc,make) 終了はコンパイラとエディタと以外のプログラムやツールの起動命令により判別する.

プログラム実行 故障を探す活動である. コンパイルしたプログラムを起動したり, 操作したり, 出力の正しさを確認することを繰り返す. この活動の開始はプログラムの起動命令により判別する.(例:a.out) プログラムの名前は必ずしも a.out ではないが作業ディレクトリ中の実行ファイルを検索することにより, 容易に推定できる. 終了は, その他のプログラムやツールの起動命令により判別する.

プログラム変更 発見した不都合を修正する活動である. 故障の原因となる欠陥を探したり, 発見した欠陥の修正を試みる. この活動の開始はエディタ, またはデバッガの起動命令により判別する.(例:vi,emacs,dbx) プログラマによっては, あるウィンドウにエディタを起動したまま別の活動を行うことがある. そのような場合は, エディタのウィンドウへの入力の切り換えにより判別する. 終了にはコンパイルの起動命令により判別する.

2.4 プログラマモデル

観測結果から, 次の特徴は, どんなプログラマについても共通であることが分かった.

1. コンパイル・プログラム実行・プログラム変更という単調な作業を反復する.
2. 最初のコンパイルは, 以降のコンパイルと比較して例外的に長い. それ以前の作業で作り込まれた多くの文法誤りが 1 度に検出されるためである.
3. プログラムの変更が行われる度に,1 回のプログラム実行の継続時間が長くなる傾向がある. 欠陥の個数や故障の頻度が減少するためである.

逆に, プログラマによって異なる特徴は以下の点である.

1. 状態遷移の頻度が大きく異なる. 一般的に経験を積んだ (または, 性急な) プログラマほど反復が速い傾向がある.
2. プログラム実行の継続時間が増加する様子がプログラマに依存して大きく異なる. 一般的に経験が不足している (または不注意な) プログラマは, 欠陥を完全に修復しないままコンパイルを行うことがある. そのような場合, プログラム実行の開始直後, 以前に発見された故障が再発する, プログラム変更が即座に再開されることになる. このような不完全なデバッグは不規則に短いプログラム実行として活動系列中に現れる. 一方, 経験を積んだ (または, 慎重な) プログラマは 1 回のプログラム変更で 1 個の欠陥を確実に修正する. 結果としてプログラム実行の継続時間が単調に増加する.

以上の分析から分かるように, デバッグ能力の高いプログラマとは, 活動の反復が速く, かつ, 欠陥を確実に修正するプログラマであると考えられる. 逆に能力の低いプログラマとは逆の特徴を持つプログラマである. しかし, 活動の反復は速いが, プログラム変更に対してデバッグが不完全である可能性が高いという特徴を持つプログラマも存在する.

2.5 モデルの定義

状態遷移表 観測された活動系列を, 各プログラマの特徴を抽出するために, プログラマモデルに適合させる. このモデルは前述したプログラマの相違点を表現できることを念頭において作成された.

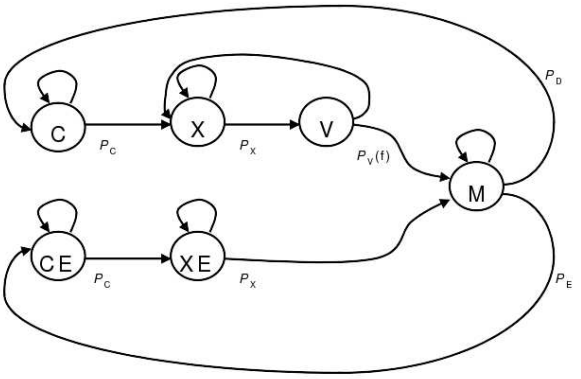

図 4: 状態遷移図

このモデル (図 4) では, プログラマ活動系列をプログラマ状態 C,X, V,M,CE,XE, からなるマルコフ過程と仮定している. 各時刻において, プログラマは 6 個の中の 1 個の状態にある. そして 1 秒間隔で, ある確率に従って, 他の状態かもしくは同一の状態に遷移する. 図の中の各矢印は 1 個の状態遷移を表し, 矢印のラベルは, その状態遷移の確率を表す.

観測されるプログラマ活動系列とプログラマ状態間の遷移との関係は次のとおりである.

- まず最初のプログラム実行において, プログラマは状態 X にあり, 作成したプログラムを起動する. 前述したように, 最初のコンパイルは例外的であるので, モデルには含まないことにした.
- 何らかのプログラムの動作が観察されるとプログラマが, それが正しいかどうかを確認し, 状態 V に遷移する.
- その際, もし故障が発見されれば, プログラマは状態 V から M に遷移し, プログラム変更を開始する.
- 発見されなければ, 状態 V から X に戻り, プログラムを再起動したり, 別の入力や操作を行う.

- 状態 M においてプログラム変更を終了すると, プログラマは状態 C か CE のどちらかに遷移する.
- 状態 M から C への遷移は発見された欠陥が完全に修正された後にコンパイルが開始されたことを意味する.
- 状態 C においてコンパイルが終了すると, プログラマは状態 X に遷移し, プログラム実行を繰り返す.
- 一方, 状態 M から CE への遷移はデバッグが不完全なままコンパイルが開始されたを意味する.
- 状態 CE においてコンパイルが終了するとプログラマは状態 XE に遷移し, プログラム実行を開始する. しかし, すぐに同一の故障が発見されるので, 直ちに状態 M に移動し, プログラム変更を再開する.

各状態遷移の確率は以下のように定義する.

p_c :状態 C から状態 X への遷移と状態 CE から状態 XE への遷移の確率は同一の定数であり,p_c と表す. コンパイルに対応する状態から抜け出る確率であり, 1 回のコンパイルの継続時間を逆数的に表す. 具体的には,$1/p_c$ がコンパイルの平均継続時間となる.

p_x :状態 X から状態 V への遷移と状態 XE から状態 M への遷移は確率も同一の定数であり,p_x と表す. プログラムの 1 回の操作に対応する状態抜け出る確率であり,1 回のプログラム操作の継続時間を逆数的に表す. 具体的には, $1/p_x$ がプログラム操作の平均継続時間となり, 作成するプログラムの特性に依存する.

p_D,p_E :状態 M から状態 C への遷移と状態 M から状態 CE への遷移の確率は独立な定数であり, それぞれ p_D,p_E と表記する.2 つのパラメータを 1 組で, プログラム変更の継続時間と不完全なデバッグの割合の両方を表す. $(p_D + p_E)$ でプログラム変更に対応する状態から抜け出る確率であり, $1/(p_D + p_E)$ がプログラム変更の継続時間となる. $p_E/(p_D + p_E)$ はプログラム変更に対する不完全なデバッグの回数の割合である.

$p_v(f)$:状態 V から状態 M への遷移は故障の発見を意味するので, この遷移確率は定数とするべきではない. そこで時刻 t 秒までに状態 V から状態 M に遷移した回数を f として遷移確率を関数 $p_v(f)$ として定義する.

遷移関数 $p_v(f)$ は以下の式により定義される.

$$p_v(f) = \alpha \cdot \max[\epsilon, f_0 - f]$$

f_0:最初のコンパイルの際に潜む欠陥の個数

α:1 回のプログラム操作で故障が発見される確率であり, 故障発見の容易さを表す. プログラマ能力に依存すると共に仕様の難しさやプログラムの複雑さに依存すると考えられる.

上式中の小正数 ϵ は遷移確率が非正数となるのを防ぐ役割を果たす. つまり $(f_0 - f)$ は時刻 t における残存する欠陥の個数であり,$p_v(f)$ は, その個数に比例すると仮定している.

モデルの制約　このプログラマモデルは, 実際のプログラマ活動系列の分析結果に基づいて定義されているが, このモデルが妥当であり, 提案する方法が有効に働くためには以下の条件が必要である.

- 故障を発見しても直ちに修正しようとせず, 複数の故障を発見した後にまとめて修正しようとした場合に, このモデルや方法は適用できない.
- コーディングとデバッグが明確に分かれていない場合に適用しては, ならない. 例えば, 仕様の一部だけが実行された後に, コンパイル, 実行, 修正, プログラムの拡張などが不規則に繰り返される場合がある.
- 独立性の高い複数のモジュールから構成される大規模プログラムを対象とした場合に, 適用してはならない. そのような場合は, 複数のプログラムを順番にあるいは, 交互にデバッグする作業になると考えられる.
- 仕様の誤解や設計に根本的な誤りに起因する大幅な変更があった場合に適用してはならない.
- デバッグ環境の不備のために, プログラマに余計な負担がかかった場合に, 適用してはならない. 例えば, プログラマが使い慣れているデバッガが使用できない場合などである. しかしツールの使用について, プログラマが有効な選択ができなかった場合には, それもデバッグ能力の一部と見做しても問題は無いと思われるので, 特に配慮を必要としない.

評価値の算出　これらのパラメータから以下の 3 種類のデバッグ能力の評価値を算出する.

r :プログラムの変更に対してデバッグが不完全である回数の割合. $p_E/(p_D + p_E)$

d :1 プログラム変更に要する時間の平均.$1/(p_D + p_E)$

D :1 欠陥の修正に要する時間の平均

1 回のプログラム変更で 1 個の欠陥が修正される (つまり,$r = 0$) とは限らないので,d と D は必ずしも同じではない. 評価値 D は d と r の両方を考慮した値である.

D は p_C,p_D,p_E,p_X から算出される. まず不完全なデバッグの割合は r であるので, デバッグの完了までに n 回のコンパイルを行う確率は $r^n(1-r)$ となる. 従って 1 個の欠陥当りのコンパイルとプログラム実行の平均回数は共に $r(1-r)$ となる. 従って 1 個の欠陥当りのプログラムの変更の回数は, それにより 1 回多いので, $1/(1-r)$ となる.1 個の欠陥の修正に要する時間 D はコンパイル・プログラム実行・プログラム変更に要する時間の和として, 次のように求まる.

$$D = \left(\frac{r}{1-r}\right)\frac{1}{p_C} + \left(\frac{r}{1-r}\right)\frac{1}{p_x} + \left(\frac{1}{1-r}\right)\frac{1}{p_D + p_E} = \frac{p_E}{p_D}\left(\frac{1}{p_C} + \frac{1}{p_X}\right) + \frac{1}{p_D}$$

評価　これらの計算により求められた評価値 r,d,D とプログラムの作成時間を比較した結果, プログラムの作成時間は評価値 D と最も高い相関を示した. 評価値 d とも相関は強かったが, 評価値 D 程高くなく, 評価値 r は余り相関は見られなかった. このことから, 評価値 D が, より正確なデバッグ能力を表していると言える.

このモデルの特徴　プログラマのデバッグ能力をキーストロークから測定する方法は, 実際の作業の様子を監視するので, 客観的かつ正確に能力を測定できる. またキーストロークを監視するだけなのでプログラマにも測定者にも余分な負担をかけない. さらに測定の手順をツールによって自動的に実行できるという利点をもつ. しかし, 作成者の作業工程の全てを監視する必要があり, 実用性という面では現実的な手段ではない.

この章で紹介した, これらの分析方法はプログラマの作業形態や進捗状況を, かなり細かく追う必要があり, 大量のプログラマを一度に分析しようとする場合, 大量のバックアップが必要になる. これらの手法は判定に膨大な時間とコストがかかってしまう. また, モデルの制約で述べたように, 大規模な開発には適用できない.

3　提案するモデル

3.1　アプローチ

2 章で紹介したモデルは実験結果からもプログラマの人的評価を導き出す上で有効と考えられる. そこで, これらのアイディアを基にプログラマ能力を簡潔に評価する手法を考える.

デバッグ能力を, 以下の 2 つの能力の組み合わせとして考える.

- バグに気が付く能力
- バグを修正する能力

2.2 のエラー寿命を測定するモデルの方では, 図 2 でも分かるように, バグを見落としている期間とバグの修正にかかる期間の合計をエラー寿命として設定している. エラー寿命が短い程プログラマの能力は高くなる. そして, バグに気が付く能力やバグを修正する能力が高ければ, エラー寿命は短くなり, プログラマ能力は高くなる.

また 2.3 のキーストロークから測定する方法では, 主にプログラムの修正時間と, その修正が適切であるかの確率をもって評価値を求めているので, バグを修正する能力が基本になっているが, バグを適切に修正できるかが評価値を求める式の中で大きな重みを持っている. そして, バグを適切に修正するには, バグの内容を適切に把握する必要がある. つまりバグの内容に正確に気が付いていなければならない. 例えばデバッグが不完全ということプログラムのバグに気が付いていないということまれていると見做して良い. 従ってバグに気が付く能力は, バグを修正する能力と同じ位に重要だと考えられる.

ここで, バグに気が付くとは, プログラムがフォールトを起こすことが付いているということでは無く, プログラムのどこに問題が存在するのかを理解することを意味する. 従来のバグの定義より広い意味になるが, プログラムの限界を超えた範囲 (処理しきれない範囲) について, 何の対処もされていないのであれば, そのような場合もバグに含めるべきである. 仕様外の処理について, エラーメッセージを表示させるのがバグの修正であり, 仕様外の入力の可能性について想定すること, それがバグに気が付くということである.

具体的には, あるデータを処理するプログラムで, 入力値がプログラムの許容範囲を超えていた場合, 許容範囲が存在することは当然のことなのだが, その限界に対して, 何の表示もされず, プログラムが無限ループに陥ったりしてしまえば, それはバグであると判定する. この場合, 入力をはじきエラーメッセージを表示させればバグの修正はされていると言えるが, 何の対処もされていなければバグに気が付いていないとみなす.

優秀なプログラマは, コードを作成する時点で, 充分な設計を行い, 作成段階の初期の時点で, プログラムの仕様書に明示的に書かれていなくても, プログラムの仕様を守る上で役立

つの機能を作り込むと考えられる. その結果, 想定外のエラーが少なくなり, 仮に起きたとしても, すぐに気が付き, 修正するので, プログラムフォールトの潜伏期間は短くなり, 長期的には機能実現までの時間は早くなるはずである. また, 仮にコードの作成途中で対応処理ができていなくても, それらの処理に対する注意はされていると考えられる. その結果, プログラムの内部処理が完成されていなくても, プログラムの概要が固まっている可能性は高いと考えられる.

つまりプログラマ能力の高さに対して, 作成されたプログラムで起こるフォールトの数は比例関係にあると仮定する. これらのことからプログラマのバグに気が付く能力の測定を基に, その作成者のプログラマ能力を測定するモデルを提案する.

3.2 モデルのキーアイディア

プログラムが内在しているバグに気が付いているかどうかを外部から判定するのは難しい. そもそもプログラム中のバグの数を特定するのは非常に困難である. そこで, バグに気が付いているのならば, そのバグに対して何らかの対処を行う機能がプログラムに作り込まれるという仮説を立てる. 仮に作成過程のプログラムでも, 設定された入力に対して, 何の出力も反応も処理もされなければ, それはバグに対して配慮がされていない, つまり気が付いていないということであると考えられる.

この仮説に立てばバグに気が付いているかどうかは, 作成されているプログラムの中である入力や処理に対して機能が実装されているかどうかで判定できる. 従ってバグに気が付く能力は, バグの存在やデバッグ状況を見るだけでなく, プログラム中の機能の実現状況を測定することでも評価できる.

しかし一般的に考えて, プログラム中の全ての機能を洗い出すのは簡単なことではない. 本研究で提案するモデルでは, プログラムの機能の実現状況を測定する実際的な手法として, チェックリストを利用して確認する方法を取る. そしてチェックリストは,3 種類のドキュメントを作成してプログラムの機能を洗い出すことで行う.

つまり本研究で提案するモデルではチェックリストで挙げられた各機能に対して, それぞれの機能の作業開始から実現するまでに必要とした時間 (機能実現時間) を測定し, それらの時間を用いて評価値を算出する.

3.3 モデルの具体的観測方法

機能実現時間の測定は以下の手順に沿っておこなう.

Step1 作成するプログラムの機能を洗い出す

Step2 挙げられた機能をまとめてチェックリストを作成する

Step3 実際の開発過程からデータを収集して各チェック項目を確認していく

3.3.1 チェックリストの作成

作成するプログラムの機能を洗い出すために, まずプログラムの仕様書から以下の三種類のドキュメントを作成する.

コードレビューのチェックリスト :プログラムの全ての欠陥を発見して修正することが必要な場合において正確な手順に従っているかを確認するために作成するドキュメント. プログラムの基本的な処理について, 単純なエラーが存在しないかを確認するために用いられる. 例えば, ポインタを使用した場合には使用後に削除されているか, ファイルをオープンした場合にはクローズされているかといったことを列挙する.

Work Breakdown Structure :必要な作業項目をトップダウン方式で, 細かく分析し, 階層構造として表現したドキュメント. プロジェクトの全体スコープを定義し, 体系化するためのプロダクト指向のツリー構造体. 仕様の機能を満たすのに必要な処理を挙げて作成する.

テスト仕様書 :仕様の機能を守るのに必要な付加的な処理について考慮して作成するドキュメント. 例えば, 仕様外の入力によりプログラムがフォールトを起こさないように入力をはじく処理などである. このような処理を列挙した文章である.

これらのドキュメントによって挙げられた機能をまとめることでチェックリストを作成する.

3.3.2 チェック項目の確認作業

チェック項目の確認は作業は図 5 のような図を作成して行う. 縦軸はチェックリストに挙げられたプログラムの各機能をチェック項目として並べる. 横軸は時間を表している. 各チェック項目に対して, その機能の作成を開始した時刻と, その機能が完成した時刻を確認し, 各機能の実現までの時間を測定する. そして, 各チェック項目の機能実現時間を合計したものがプログラマの評価値となる. この評価値が低いほど, プログラマ能力は高いということになる.

3.4 適用条件

この評価値を求める上で以下のことを前提としている.

- プログラムの作成過程に矛盾が存在しないこと

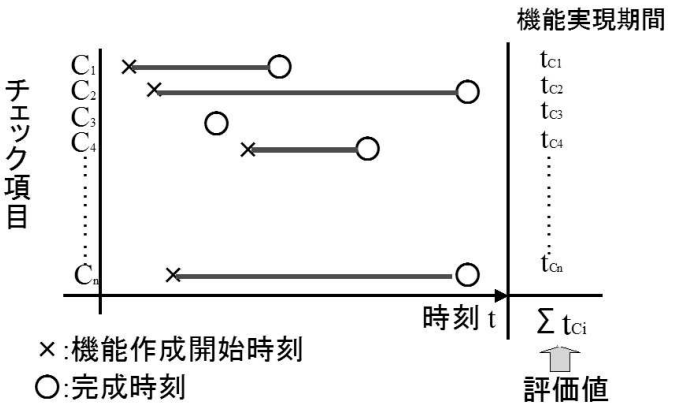

図 5: モデル

- プログラムの作成過程は全て把握していること
- プログラムの作成者が作成開始時から変わっていないこと

プログラムの作成過程に矛盾が存在しないこととは, 作成されたプログラムがある期間に急激に変化を起こさないことである. 具体的に言えば, 急なアルゴリズムの変化や一般的でない処理に対応するコードが急激に増えることを指す. 例えば, 他のプログラムからコピーを行った場合には評価は不可能である.

プログラムの作成過程を全て把握しているとは, 例えば, 単位時間ごとにデータのバックアップを取っている場合に, そのデータが全て揃っていること, そしてプログラムの作成をデータのバックアップの取れない場所で行っていないことを指す. 従って, プログラムの作成過程を全て把握していない状況には適用できない.

プログラムの作成者が作成開始時から変わっていないとは, 作成過程でアドバイスを受けたりして, 自分の気が付いていないバグを指摘されて気が付いた場合を除くということである. 例えば, プログラムのコピーをした者も含まれるが, それ以外に他人が準備したテストデータを使っただけなど, コード作成過程上には表れない他人のサポートがあった場合が, これにあたる.

以上をまとめると, 個人として作成されたプログラムならば, その作成者以外の人は関与していない状態を前提としている.

4　評価実験

4.1　概要

4.1.1　課題内容

ここでは, ある企業の新入社員に対して行われた新人研修で作成されたプログラムを使用して評価実験を行った. 作成されたプログラムは C 言語で書かれていて, 課題は全部で 3 つ行われた. 課題内容を以下に示す.

課題 1 英語の文章 (テキストファイル) を読み込み, その中に現れた単語の一覧表を作成する. 入力はテキストファイルを指定して行い, 出力は標準出力とする. 単語の一覧表は ASCII コードの昇順にソートして,1 行につき 1 単語を出力する. また各単語の出力回数も出力する. 単語は指定された区切り子と空白によって分離された語と設定されている. プログラムの規模は約 300 行. 作成期間は 3 日間.

課題 2 C 言語のソースプログラムから字句 (トークン) を切り出して,1 行に 1 つずつ表示する. 入力はソースファイルを指定して行い, 出力は標準出力とする. トークンは出現順に 1 行につき 1 つ出力され,1 行はトークンと, その種類を出力する. トークンの種類は 7 種類に分類されていて, 複数の種類に属するトークンも存在している. しかし種類が重複するトークンの処理や入力されるソースのエラー等は考慮しなくても良いと設定されている. プログラムの規模は約 500 行. 作成期間は 3 日間.

課題 3 クロスリファレンス (相互参照表) を作成する. 入力は C 言語のソースファイルを指定して行い, 出力は標準出力とする. 出力されるのはファイル中の識別子である. 識別子は, 英字または下線で始まり英数字または下線が続くトークンのうち予約語をもので有効数字は 31 文字まで, と定義されている. プログラムの規模は約 600 行. 作成期間は 3 日間.

4.1.2　プログラムの作成環境

研修生の課題への取り組み方は以下のような形を取っている.

- 各演習課題は提出期限が定められており, 基本的にその期限内に提出する.
- 提出先はホームサーバーの下の指定されている各自のディレクトリの中となっている.
- 提出物は各自のソースコード, 行ったテスト内容と障害記録, 使用したアルゴリズムの説明, そして作業内容報告書である.

- 作業可能時間は平日の午前 9 時から午後 8 時まで, データの持ち出しは原則的に禁じられている.

4.2 収集データ

プログラムの作業過程から以下のようにデータを収集した.

- 各プログラマのホームディレクトリの中で新しく更新されたファイルを 1 時間おきに取得する
- 1 日の最後にホームディレクトリの内容の完全なバックアップを取得する
- 指定されている各自の提出先のディレクトリにも, 同様のことを行う

4.3 分析

今回の実験では, 全部で 240 人の研修生が参加したが, 用いるデータはモデルの条件を明らかに満たしている研修生に絞った. また, 演習は 13 のグループに分かれて行われたが, グループ内での共同作業の可能性を踏まえて, グループの研修生, 全てのデータが完全に揃っているグループのみのデータを分析した.

また, 課題 3 に関しては以下の理由から, 今回の分析から外すことにした.

- 課題の内容が課題 2 と課題 1 を合わせれば容易にできる課題であった.
- 課題 3 の開始前にグループ内の小グループで課題 2 の内容についてディスカッションとプレゼンテーションを行っている関係で, 殆どのプログラマの間で満たしているチェック項目に差が生じていなかった.
- 課題の成績点も大半の人が前の 2 つの課題に比べて点数が上昇し, 平均点も大きく上昇しているので, 純粋な資質を計るには不適切と判断した.

課題の完成時間は提出されたプログラムと, 同一のコードがプログラマのホームディレクトリで作成された時刻を探し, その時刻を完成日時とした. また, 提出書類の作成をプログラムの作成時間に含めるのはプログラマ能力の測定の意図と反するので, あくまでプログラムに着目し, そのプログラムが完成した時刻とした.

結果として,11 人の研修生のデータを得られた. 課題 1 のチェック項目は表 1 に, 課題 2 のチェック項目は表 2 に挙げる. また 11 人の得点を表 3 に示す.

チェック項目番号	機能内容
1	ファイル名は指定されているか
2	指定されたファイルが存在しなかった場合や開けられない場合の処理はされているか
3	引数の数が正しいか
4	ファイルを閉じているか
5	入力されたファイルが空の場合の処理はされているか
6	入力されたファイルが空白のみの場合の処理はされているか
7	入力されたファイルが改行のみの場合の処理はされているか
8	英数字以外の文字を不適切な入力として検出しているか
9	メモリ領域を確保されているかを確認しているか
10	余分なメモリ領域を開放しているか
11	挿入が終わった後に単語の格納領域を開放か初期化をしているか
12	リストが空の場合の処理はされているか
13	出力したリストのメモリ領域は開放されているか
14	数字の出力は最後までされていて,10 進表現でされているか
15	指定されたファイルの文字数が 32 文字以上の場合の処理はされているか
16	32 文字以上の文字列を読み込んでも正しく出力されるか
17	識別子, 演算子に対する処理を行っているか
18	仕様書に書かれていないが区切り子として判定する必要のある記号の処理はされているか

表 1: 課題 1 のチェック項目

チェック項目番号	機能内容
1	入力されたファイルが存在しなかった場合や開かなかった場合の処理はされているか
2	引数の数が正しいか
3	ファイルを閉じているか
4	空白に対する処理はされているか
5	改行に対する処理はされているか
6	バッファサイズ (32 文字) を越えた場合の処理はされているか
7	英文が入力された場合の配慮はなされているか
8	日本語入力に対する対応はされているか
9	指導書で定義されていない文字列に対する処理はされているか
10	コメントに対する処理はなされているか
11	全ての予約語に対応しているか
12	大文字で入力された予約語を識別子として扱うか
13	全ての識別子に対応しているか
14	”.” を小数点として扱っていないか
15	文字列中の数値にも対応しているか
16	16 進数表現に対応しているか
17	指数表現に対応しているか
18	不適切な小数点に対して対応しているか
19	数値の間に空白が挟まっていた場合に別々の数値として扱っているか
20	￥が出てきても対応できるか
21	” に対応できるか
22	入れ子になっている場合の対応はされているか
23	文中に￥が出てきても対応できるか
24	ヌル文字列であった場合にも対応できるか
25	入れ子になっている場合の対応はされているか
26	全ての演算子に対して対応しているか
27	演算子を正しく分割できるか
28	負の数との違いを認識しているか
29	全ての区切り子に対応しているか
30	メモリ領域を確保されているかを確認しているか
31	リストが空の場合の処理はされているか
32	出力したリストのメモリ領域は開放されているか

表 2: 課題 2 のチェック項目

作成者	課題 1	課題 2
A	75	142
B	84	138
C	100	240
D	82	163
E	87	173
F	120	223
G	87	263
H	93	218
I	101	185
J	104	196
K	217	215

表 3: モデルによる評価値

4.4 考察

今回の評価実験では, プログラム作成者は企業の新人研修の受講者であるので, 2.1.2 で述べた理由により作成者のプログラマ能力は課題の成績から判断できる. 今回は作成したプログラムの課題の点だけでなく, コンピューターサイエンス系科目の成績の総合評価も分析の考慮に使用する.

4.4.1 評価値と成績の比較

全ての条件を満たした 11 人の作成者の評価値と課題の成績の間でスピアマンの順位相関係数を取った結果, 課題 1 で 0.71, 課題 2 で 0.79 という結果になり, 相関があることが示された. この結果から, 今回, 提案したモデルはプログラム能力測定としての機能を果たしていると言える. 課題 1, 課題 2, の課題の成績とモデルの評価値については表 4, 表 5 のようになった.

また, 相関から外れた作成者に限定してコンピューターサイエンス系科目の成績の総合評価を見たところ, モデルの評価値との間に相関関係が成立した. モデルの評価値の高い作成者はコンピューターサイエンス系科目の成績の総合評価が高く, モデルの評価値の低い作成者はコンピューターサイエンス系科目の成績の総合評価が低かった.

作成者	課題 1 の成績	課題 1 の順位	モデルの成績	モデルの順位
A	90	1	73	1
D	80	4	82	2
B	83	2	84	3
E	80	4	87	4
G	77	7	87	4
H	76	8	93	6
C	83	2	100	7
I	76	8	101	8
J	63	10	104	9
F	80	4	120	10
K	58	11	217	11

表 4: 課題 1 の結果

作成者	課題 2 の成績	課題 2 の順位	モデルの成績	モデルの順位
B	95	1	138	1
A	88	2	142	2
D	84	6	163	3
E	86	3	173	4
I	86	3	185	4
J	82	7	196	6
K	80	9	215	7
H	85	5	218	8
F	79	10	223	9
C	82	7	240	10
G	51	11	263	11

表 5: 課題 2 の結果

4.4.2　条件外のデータについて

順位相関を取ることで, 条件を満たした作成者のプログラマ能力評価が正しいということが実験的に確かめられた. ここでは, 条件を満たすことができず使用できなかったデータに対して, どのような問題が生じたかについて触れておく.

今回の課題は個人で作成されていたが, 授業の関係上, グループに分けられていた. そしてテストデータが提供されていたグループが 2 つ存在した. 今回のモデルは個人の評価を比較するので, テストデータの提供も他人の助けとしてサンプルから外した. この 2 つのグループについては対照的な結果が出た.

まず 1 つ目のグループに関しては,17 人の作成者の中で課題の仕様を満たしていたプログラムが 1 つしか存在しなかった. 全てのプログラムはテストデータに対して, 満足な結果を返したが, テストデータにのみ注目がいってしまったようで, 仕様に対する設計が不充分になってしまった.

もう 1 つのグループに関しては, 上述したグループのようなことは起きなかったが, 分析結果の表を見ると, 殆どの作成者がチェック項目をプログラムの完成直前に満たしており, モデルによる評価値は全体的に低くなってしまった. それに反して課題の評価値は高く, またコンピューターサイエンス系科目の成績の総合評価は低かった.

他にも課題の模範解答が示されていたと思われるグループが存在したが, そのグループで作成されたプログラムの大半が作成過程での満たされているチェック項目の数に変化が無かった.

これらのことを踏まえると, このモデルによる測定はプログラムの作成過程に対する評価や分析にも活用できる. このような評価方法は最終的な成果だけでなく, 途中経過に対しても評価を付けられるので, 教育機関で有効に活用できると考えられる.

5　むすび

今回の評価実験で, プログラムの機能を実現する過程をチェック項目で調べていくことでプログラマの能力値を判定できることを示した.

プログラム開発においてはテストは莫大な時間と費用を必要としている. しかし作成した時点で機能が充分に, その役割を果たすのであれば, テストの必要性は減少し, また精度も高めることができる. そういった意味でも, チェック項目を基本に作成過程を確認する作業は, プログラム作成のコストを抑えることに繋がる.

また, 本研究で提案するモデルはドキュメントの作成が重要であることから, 教育機関でのプログラム作成の課題で作成者自身に行わせても, 教育者側が進行状況の把握に使っても有効と考えられる. 作成者はドキュメント作成の大切さ, 課題作成の過程や流れ, 進行の傾向を自覚し, 自分の作業を管理することができる. また教育者側はドキュメントを作成者の成績評価にも使用できる.

現実のプロジェクト開発では, ドキュメントをプロジェクトリーダがプロジェクトの進行状況を確認するために使える可能性もあるが, そのためにはチェック項目の作成方法や, チェック方法の簡略化, 自動化が望まれる. またチェック項目の重み付けも必要になってくると考える.

今回の評価実験では以下の理由から各機能の重みを等しくして評価値を求めた.

- 課題のプログラムの規模が小さかったので, 機能を満たしていく過程で大半のチェック項目を満たしてしまうこと. そしてプログラム作成過程で行われた授業で例として提供されたプログラムを参考にすることで防げたエラーも多く, それらに対するチェック項目は異なる重みを持たせる意味はなかったと考えられる.

- 課題の作成期間が短く, 最大でも 4 日, 大半の人が 2 日でしあげていたので, プログラムを熟考する程ではなかった. そのため, 特に何の問題も起きずに課題を終えてしまった人も多く, そのため実際のテストで作成されるような詳細なドキュメントが作成されずに課題を終えてしまった可能性が高い.

現実的には, 大規模なプログラムにおいては各機能のプログラム本体に与える影響が同一とは考えにくい. 各機能に対する重み付けは今後の課題である. また, 確認方法の自動化, 異なるプログラムを異なるプログラマが作成した場合の比較を容易にするためのモデルの改善と具体化は今後の課題である.

謝辞

本研究において, 常に適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に深く感謝致します.

本研究において, 常に適切な御指導および御助言を頂きました 同 楠本 真二 助教授に深く感謝致します.

本研究において, 常に適切な御指導および御助言を頂きました 同 松下 誠 助手に深く感謝致します.

本研究において, 常に適切な御指導および御助言を頂きました科学技術振興事業団 若手個人研究推進事業 (さきがけ研究 21) 研究員 神谷 年洋 氏に深く感謝致します.

本研究において, 評価実験関してお手伝い頂いた日本ユニシス株式会社 毛利 幸雄 氏に深く感謝致します.

本研究において, 様々な御指導, 御助言、御手伝いを頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期過程 1 年 肥後 芳樹氏に深く感謝致します.

本研究において, 様々な御指導, 御助言、御手伝いを頂きました 同 松川 文一氏に深く感謝致します.

最後に, その他様々な御指導, 御助言を頂いた大阪大学 大学院情報科学研究科コンピュータサイエンス専攻 井上研究室の皆様に深く感謝致します.

参考文献

[1] B.W. Boehm, “Improving software productivity”, IEEE Computer, 1987.

[2] B.W. Boehm, “Software Enginering Economics”, IEEE Computer, 1981.

[3] 福田, “バグ重大度の定量化の試み”, 情報処理学会研究報告, SW-32-2, 1983.

[4] IEEE, “IEEE Standard for Software Reviews and Audits”, ANSI/IEEE, Std:1028-1988, 1988.

[5] 伊藤 潔, 廣田 豊彦, 冨士 隆, 熊谷 敏, 川端 亮, “ソフトウェア工学演習”, オーム社, 2001.

[6] I. Jacobson, M. Griss and P. Jonsson 著, 杉本 宣男, 吉田 幸彦, 落合 修, 田中 正仁 監訳, “ソフトウェア再利用ガイドブック”, トッパン, 1999.

[7] C. Jones 著, 富野 壽 監訳, “ソフトウェア品質のガイドライン”, 共立出版, 1999.

[8] N. Juristo, A.M. Moreno, “Basics of Software Engineering Experimentation”, Kluwer Academic Publishers, 2001.

[9] 鍵和田 京子, 石村 貞夫, “よくわかる卒論・修論のための統計処理の選び方”, 東京図書, 2001.

[10] B.W. カーニハン, D.M. リッチー, 石田 晴久 監訳, “プログラミング言語 C”, 共立出版, 1989.

[11] 工藤 英男, 内田 真司, 門田 暁人, 松本 健一, “保守工程におけるデバッグ作業者のバグ特定プロセス分析”, 奈良工業専門学校研究紀要第 35 号,pp.75-80,2000.

[12] 松本 健一, 井上 克郎, 菊野 亨, 鳥居 宏次, “エラー寿命に基づくプログラマ性能の実験的評価 -大学環境におけるプログラム開発-”, 電子情報通信学会論文誌,Vol.J71-D,No.10,pp.1959-1965,1988.

[13] 松本 健一, 楠本 真二, 菊野 亨, 鳥居 宏次, “プログラム開発におけるチーム性能のモデルに基づく実験的評価-プログラマ性能モデルの拡張-”, 情報処理学会論文誌,Vol.31,No.12,pp.1812-1821,1990.

[14] 松村 知子, 門田 暁人, 松本 健一, “潜在コーディング規則に基づくバグ検出方法の提案”, ソフトウェアシンポジウム 2002 論文集,pp.105-114,2002.

[15] T. Moher and G.M. Schneider, “Methods for improving controlled experimentaion in software engineering”, Proc. 5th ICSE, 1981.

[16] H. Sackman, W.J. Erickson and E.E. Grant, “Exploratory experimental studies comparing online and offline programming performance”, Commun.ACM, pp.3-11, 1968.

[17] 重松 英二郎, 水野 修, 菊野 亨, 高木 徳生, “フィールド不具合数を許容値以下に抑えるためのソフトウェアテスト工数の推定モデルの提案”, 電子情報通信学会技術研究報告,Vol.SS2001,No35,pp.9-15,2002.

[18] 高田 義広, 鳥居 宏次, “プログラマの能力をキーストロークから測定する方法”, 電子情報通信学会, 1994.

[19] S. Uchida, A. Monden, H. Iida, K. Matsumoto, H. Kudo, “A Multiple-View Analysis Model of Debugging Process”, Proc.1st ISESE, 2002.

[20] 山田 茂, 高橋宗雄 著, “ソフトウェアマネジメントモデル入門”, 共立出版, 1993.

[21] Frederick P., Jr. Brooks,”The Mythical Man-Month: Essays on Software Engineering”Addison-Wesley Pub Co,1985.