内容

上記、目次は私が作成しました。

下記においてこの色字は私が注目した箇所です。

エドガー・W・ダイクストラ著『**謙虚なプログラマー**』

原文：http://www.cs.utexas.edu/~EWD/ewd03xx/EWD340.PDF

以下は GoogleTanslate による翻訳です

序論

さまざまな偶然が重なり、私は 1952 年の春の初日の朝に正式にプログラミングの世界に足を踏み入れました。私が調べた限りでは、オランダでプログラミングの世界に足を踏み入れた最初のオランダ人でした。振り返ってみると、最も驚いたのは、少なくとも私が住んでいた地域では、プログラミングの業界がゆっくりと誕生したことでした。今ではその遅さは信じがたいほどです。しかし、その時代について、その遅さを疑う余地なく証明する 2 つの鮮明な記憶に感謝しています。

プログラミングを 3 年ほど続けた後、アムステルダムの数学センターで当時私の上司だった A.vanWijngaarden と議論を交わしました。この議論には、私は生きている限り感謝し続けるでしょう。要点は、私はライデン大学で理論物理学を同時に学ぶことになっていたのですが、この 2 つの活動を組み合わせることがますます難しくなってきて、プログラミングをやめて本物の立派な理論物理学者になるか、最小限の努力で物理学の勉強を形式的に完了させるだけにして、……ええ、何ですか？プログラマー？でも、それは立派な職業ですか？結局のところ、プログラミングとは何だったのでしょう？知的に立派な学問としてプログラミングを支えることができる確かな知識体系はどこにあるのでしょうか？ハードウェアの同僚が、自分の専門能力について尋ねられたときに、少なくとも真空管やアンプなどについては何でも知っていると指摘できるのに、

私はその質問に直面したら何も知らないだろうと感じていたことを、私はとても鮮明に覚えています。不安でいっぱいの私は、ファン・ウィンガーデンのオフィスのドアをノックし、「ちょっと話をしてもいいですか」と尋ねました。数時間後、オフィスを出たときには、私は別人になっていました。というのも、私の悩みを辛抱強く聞いた後、彼は、その瞬間までプログラミングの分野はあまりなかったことに同意しましたが、その後、自動コンピューターは今後も存在し続けるだろう、私たちはまだ始まったばかりであり、私は今後数年間でプログラミングを立派な分野にするために呼ばれる人物の一人になれるのではないか、と静かに説明し続けました。これが私の人生の転機となり、私はできるだけ早く物理学の勉強を正式に終えました。もちろん、上記の話の教訓の 1 つは、若い人にアドバイスをするときは非常に注意しなければならないということです。彼らは時々それに従います!

さらに 2 年後の 1957 年に私は結婚しました。オランダの結婚の儀式では職業を申告することが義務付けられており、私はプログラマーであると言いました。しかし、アムステルダム市の市当局は、そのような職業は存在しないという理由でそれを受け入れませんでした。そして、信じられないかもしれませんが、私の結婚証書には「職業」の見出しの下に「理論物理学者」というばかげた記載があります。

私の国でプログラミング専門職がゆっくりと登場するのを見たのは、このくらいのことでした。それ以来、私は世界をもっと見てきましたが、他の国でも、時期のずれは別として、成長パターンはほとんど同じであるというのが私の一般的な印象です。

今日の状況をよりよく理解するために、昔の状況をもう少し詳しく把握してみたいと思います。分析を進めていくと、プログラミング作業の本質に関する一般的な誤解が、今では遠い過去にまで遡ることができることがわかります。

最初の自動電子計算機はすべてユニークな単一コピー機であり、実験室のような刺激的な雰囲気のある環境で使用されていました。自動計算機の構想が浮かんだ後、それを実現することは当時の電子技術にとって大きな挑戦でした。そして、1 つ確かなことは、このような素晴らしい機器を作ろうと決心したグループの勇気を否定することはできないということです。それらは素晴らしい機器でしたが、振り返ってみると、最初の機械が少なくとも時々は動作したことに驚かされるばかりです。圧倒的な問題は、機械を正常に動作させ、維持することでした。自動計算の物理的側面へのこだわりは、計算機協会や英国コンピュータ協会など、この分野の古い科学団体の名前に今でも反映されており、その名前では物理的な機器が明示的に参照されています。

かわいそうなプログラマーはどうなったでしょうか。正直に言うと、彼はほとんど注目されませんでした。第一に、最初のマシンはあまりにも大きくて動かすこともできず、その上、メンテナンスが大変だったので、人々がそのマシンを使おうとする場所が、そのマシンが開発された研究室と同じ場所だったのは当然のことでした。第二に、彼のあまり目立たない仕事には魅力がまったくありませんでした。訪問者にそのマシンを見せることはできましたが、それはコーディングのシートよりも何桁も見事でした。しかし、何よりも重要なのは、プログラマー自身が自分の仕事について非常に控えめな見方をしていたことです。彼の仕事の意義はすべて、その素晴らしいマシンの存在から

生まれたものでした。そのマシンはユニークなものだったので、彼は自分のプログラムが局所的な意義しか持たないことを非常によく知っていました。また、このマシンの寿命が限られていることは明白だったので、自分の仕事で永続的な価値を持つものはごくわずかであることを知っていたのです。最後に、プログラマの仕事に対する姿勢に深い影響を与えた状況がもう 1 つあります。一方では、信頼性が低いだけでなく、マシンが通常非常に遅く、メモリが通常非常に小さいため、プログラマは窮地に陥っていました。他方では、通常、マシンのやや奇妙な命令コードが最も予期しない構成に対応していました。そして当時、多くの賢いプログラマは、不可能を自分の機器の制約に押し込む巧妙なトリックから、計り知れない知的満足感を得ていました。

プログラミングに関する 2 つの意見は、その時代に遡ります。今、それらについて触れましたが、後でまた取り上げます。1 つの意見は、本当に有能なプログラマーは、パズルを解くのが得意で、巧妙なトリックが大好きだというものでした。もう 1 つの意見は、プログラミングとは、計算プロセスの効率をいずれかの方向に最適化すること以外の何ものでもないというものでした。

後者の意見は、実際に利用可能な機器が足かせになるという状況が頻繁に発生したことによるもので、当時は、より強力なマシンが利用可能になればプログラミングはもはや問題ではなくなるという素朴な期待によく遭遇しました。なぜなら、そうなればマシンを限界まで押し上げるための苦労はもはや必要なくなり、プログラミングとはまさにそれだからです。しかし、その後の数十年間でまったく異なることが起こりました。より強力なマシンが利用可能になり、1 桁どころか、数桁も強力になったのです。しかし、すべてのプログラミング問題が解決されたという永遠の至福の状態にいる代わりに、私たちはソフトウェア危機に首までつかっていることに気付きました。どうしてでしょうか。

ちょっとした原因があります。1、2 点において、現代の機械は基本的に古い機械よりも扱いにくいのです。まず、予測不可能で再現不可能な瞬間に発生する I/O 割り込みがあります。完全に決定論的なオートマトンを装っていた古いシーケンシャルマシンと比較すると、これは劇的な変化であり、多くのシステムプログラマーの白髪が、その機能によって生じる論理的な問題について軽々しく語るべきではないという事実を証明しています。次に、マルチレベルストアを備えたマシンがあり、管理戦略の問題が生じています。このテーマに関する膨大な文献があるにもかかわらず、この問題は依然としてかなりとらえどころのないままです。実際のマシンの構造変更によって複雑性が増した理由はこれだけです。

しかし、私はこれをマイナーな原因と呼びました。主な原因はマシンが数桁も強力になったことです。はっきり言って、マシンがなかった間はプログラミングはまったく問題ではありませんでした。私たちが少数の弱いコンピュータを持っていたときは、プログラミングはささいな問題になりましたが、今では巨大なコンピュータがあり、プログラミングは同様に巨大な問題になっています。この意味で、電子産業は単一の問題を解決したわけではなく、問題を作り出しただけです。製品の使用の問題を作り出したのです。別の言い方をすると、利用可能なマシンのパワーが 1000 倍以上になったため、

これらのマシンを適用したいという社会の野心もそれに比例して大きくなり、目的と手段の間の緊張が爆発したこの分野で仕事を見つけたのは、かわいそうなプログラマーでした。ハードウェアのパワーの増加と、おそらくさらに劇的な信頼性の向上により、プログラマーが数年前には夢にも思わなかったソリューションが実現可能になりました。そして今、数年後、プログラマーはそれらのソリューションを夢見なければならず、さらに悪いことに、そのような夢を現実に変えなければなりませんでした。私たちがソフトウェア危機に陥ったことは不思議なことでしょうか?いいえ、決してそうではありません。ご想像のとおり、これはかなり前から予測されていました。しかし、もちろん、小予言者の厄介なところは、5 年後になって初めて、彼らが正しかったことが本当にわかるということです。

そして、60 年代半ばに、恐ろしいことが起こりました。いわゆる第 3 世代のコンピュータが登場したのです。公式の文献によると、価格と性能の比率は主要な設計目標の 1 つでした。しかし、マシンのさまざまなコンポーネントのデューティサイクルを「性能」と見なすと、パフォーマンス目標の大部分が、必要性が疑わしい内部のハウスキーピングアクティビティによって達成される設計になってしまうのを防ぐことはほとんどできません。また、価格の定義をハードウェアに支払う価格とすると、プログラミングが非常に難しい設計になってしまうのを防ぐことはほとんどできません。たとえば、注文コードによって、プログラマーまたはシステムに対して、実際には解決できない競合を生じさせるような早期の拘束決定が強制される可能性があります。そして、これらの不快な可能性は、かなりの程度まで現実のものとなったようです。

これらのマシンが発表され、その機能仕様が明らかになったとき、私たちの中のかなりの数の人がかなり惨めになったに違いありません。少なくとも私はそうでした。そのようなマシンがコンピューターコミュニティに溢れかえることは当然予想できたことであり、したがって、その設計が可能な限り健全であることがさらに重要でした。しかし、その設計には重大な欠陥が組み込まれていたため、私は、一撃でコンピューターサイエンスの進歩が少なくとも 10 年遅れてしまったと感じました。そのとき、私は職業人生で最も暗い 1 週間を過ごしました。おそらく今最も悲しいことは、長年の苛立たしい経験を経てもなお、マシンはそうなるべきだということを自然の法則が教えてくれると心から信じている人がまだ非常に多いことです。彼らは、これらのマシンがどれだけ売れたかを見て疑念を黙らせ、その観察から、結局のところ、設計はそれほど悪いはずがないという誤った安心感を得ています。しかし、よく調べてみると、その防御線は、多くの人が喫煙しているから喫煙は健康に良いに違いないという主張と同じ説得力を持っています。

この点に関して、コンピューティング分野の科学雑誌が、科学出版物をレビューするのとほぼ同じように、新しく発表されたコンピュータのレビューを掲載することが慣例になっていないことを残念に思います。マシンをレビューすることは、少なくとも同じくらい重要です。ここで告白します。60 年代の初めに、私はそのようなレビューを CACM に提出するつもりで書きましたが、アドバイスを求めてその文章を送った数人の同僚が全員にそうするように勧めたにもかかわらず、私自身または編集委員会にとって困難が大きすぎると判明することを恐れて、敢えてそれをしませんでした。この抑制は私

の側の臆病な行為であり、私はますます自分自身を責めています。私が予見した困難は、一般に受け入れられている基準が存在しない結果であり、適用することを選択した基準の妥当性に確信を持っていましたが、私のレビューが「個人的な好みの問題」として拒否または破棄されるのではないかと恐れました。私は今でも、このようなレビューは非常に役立つと考えており、そのようなレビューが掲載されるのを待ち望んでいます。なぜなら、そのようなレビューが掲載されることが、コンピューティングコミュニティの成熟の確かな兆候となるからです。

私がハードウェアの分野に上記の注意を払った理由は、あらゆるコンピューティングツールの最も重要な側面の 1 つは、それを使用する人の思考習慣に与える影響であると感じているからです。また、その影響は一般に考えられているよりも何倍も強いと信じる理由もあります。では、ソフトウェアの分野に目を移しましょう。

ここでは多様性があまりにも大きいため、私はいくつかの踏み石にとどまらざるを得ません。私は自分の選択が恣意的であることを痛感しており、言及されないまま残る多くの努力に対する私の評価に関して、いかなる結論も導き出さないようお願いします。

最初はイギリスのケンブリッジに EDSAC がありましたが、最初からサブルーチンライブラリの概念がそのマシンの設計とその使用方法において中心的な役割を果たしていたことは非常に印象的だと思います。それから 25 年近く経ち、コンピューティングの状況は劇的に変化しましたが、基本ソフトウェアの概念は今でも存在しており、クローズドサブルーチンの概念は今でもプログラミングの重要な概念の 1 つです。クローズドサブルーチンは、ソフトウェアの最も偉大な発明の 1 つであると認識する必要があります。これは 3 世代のコンピューターを生き延びており、さらに数世代も生き延びるでしょう。なぜなら、これは抽象化の基本パターンの 1 つを実装するからです。残念ながら、その重要性は第 3 世代のコンピューターの設計で過小評価されてきました。第 3 世代のコンピューターでは、明示的に名前が付けられた演算ユニットのレジスタの数が多いため、サブルーチンメカニズムに大きなオーバーヘッドが生じます。しかし、それでもサブルーチンの概念が消滅したわけではなく、突然変異が遺伝性でないことを祈ることしかできない。

私が言及したいソフトウェア界における 2 つ目の大きな進歩は、FORTRAN の誕生です。当時、これは非常に大胆なプロジェクトであり、その責任者たちは大いに賞賛に値します。10 年ほどの広範な使用を経て初めて明らかになった欠点について、彼らを責めるのはまったく不公平です。10 年先を見据えて成功したグループは非常にまれです。振り返ってみると、FORTRAN は成功したコーディング手法であると評価せざるを得ませんが、構想を練る上で効果的な補助はほとんどなく、現在ではその補助が緊急に必要とされているため、時代遅れと見なすべき時期が来ています。FORTRAN が存在したことは早く忘れた方がよいです。思考の手段としてはもはや不十分だからです。FORTRAN は脳力を浪費し、リスクが大きすぎて、使用するには費用がかかりすぎます。FORTRAN の悲劇的な運命は、広く受け入れられ、何千人ものプログラマーを過去の失敗に精神的に縛り

付けたことです。私は、より多くのプログラマー仲間が互換性の呪いから解放される方法を見つけられるよう、日々祈っています。

3 つ目のプロジェクトとして、LISP について触れておきたいと思います。これはまったく異なる性質を持つ魅力的なプロジェクトです。ごく基本的な原則を基盤として、驚くべき安定性を示しています。それに加えて、LISP は、ある意味では最も洗練されたコンピュータアプリケーションのかなりの数を担ってきました。LISP は、冗談めかして「コンピュータを悪用する最も賢い方法」と表現されています。この表現は、解放の雰囲気を余すところなく伝える素晴らしい賛辞だと思います。LISP は、これまで不可能だった考えを思いつくのに、最も才能のある多くの人類を支援してきました。

4 番目に言及するプロジェクトは ALGOL60 です。今日に至るまで、FORTRAN プログラマーは、使用している特定の実装の観点からプログラミング言語を理解する傾向があり(そのため、8 進数ダンプや 16 進数ダンプが普及しています)、LISP の定義は、言語の意味とメカニズムの動作が奇妙に混在していますが、有名なアルゴリズム言語 ALGOL60 に関するレポートは、抽象化をさらに重要なステップに進め、実装に依存しない方法でプログラミング言語を定義するという真摯な努力の成果です。この点で、その作成者は非常に成功しており、そもそも実装できるかどうかについて深刻な疑問を抱かせたと言えるでしょう。このレポートは、形式手法 BNF(現在では Backus-Naur-Form としてよく知られています)の威力と、少なくとも PeterNaur のような優れた人物が使用した場合の慎重に表現された英語の威力を、見事に実証しました。これほど短い文書で、コンピューティングコミュニティに同様に大きな影響を与えたものはごくわずかだと言っても過言ではないでしょう。後年、ALGOL や ALGOL-like という名前が、保護されていない商標として、時にはほとんど関連のないいくつかの新しいプロジェクトにその栄光をいくらか与えるために簡単に使用されたことは、ALGOL の地位に対するいくぶんか衝撃的な賛辞です。定義デバイスとしての BNF の強さは、私がこの言語の弱点の 1 つと考えるものの原因です。つまり、過度に複雑であまり体系的ではない構文が、今では非常に少ないページの範囲内に詰め込まれてしまうのです。BNF ほど強力なデバイスがあれば、アルゴリズム言語 ALGOL60 に関するレポートはもっと短くできたはずです。それに加えて、私は ALGOL60 のパラメーターメカニズムに非常に疑問を感じています。ALGOL60 はプログラマーに非常に多くの組み合わせの自由を与えるため、自信を持って使用するにはプログラマーによる強い規律が必要です。実装に費用がかかるだけでなく、使用するのは危険に思えます。

最後に、この話題は楽しいものではありませんが、定義ドキュメントが恐ろしく大きく複雑なプログラミング言語である PL/1 について触れなければなりません。PL/1 を使うのは、コックピットで操作する 7000 個のボタン、スイッチ、ハンドルを備えた飛行機を操縦するようなものです。プログラミング言語(私たちの基本的なツールです!)が、そのまったくの奇抜さゆえにすでに私たちの知的制御を逃れているのに、成長し続けるプログラムをどのようにしてしっかりと知的制御下に置けるのか、私にはまったくわかりません。そして、PL/1 がユーザーに与える影響を説明する必要がある場合、私の頭に浮かぶ最も近い比喩は、麻薬です。高水準プログラミング言語に関するシンポジウムで、PL/1 を擁護する講演をした人物を覚えています。彼は、自らを PL/1 の熱心なユ

ーザーの 1 人だと称していました。しかし、PL/1 を称賛する 1 時間の講演の中で、彼は約 50 個の新しい「機能」の追加を要求しました。彼の問題の主な原因が、すでに「機能」が多すぎることにある可能性は十分に考えられませんでした。講演者は、依存症の憂鬱な症状をすべて示し、精神的に停滞した状態に陥り、もっと、もっと、もっととしか言えなくなっていました FORTRAN が小児疾患と呼ばれているのに対し、危険な腫瘍の成長特性を持つ完全な PL/1 は、致命的な病気になる可能性があります。

過去についてはここまでです。しかし、その後にそこから学べない限り、間違いを犯しても意味がありません。実際、私たちは多くのことを学んできたので、数年以内にプログラミングは今までとはまったく異なる活動になる可能性があり、その違いがあまりにも大きいので、ショックに備えたほうがよいでしょう。考えられる未来の 1 つを概説しましょう。一見すると、おそらく近い将来のプログラミングのビジョンは、まったく空想的なものに思えるかもしれません。したがって、このビジョンが非常に現実的な可能性であるという結論に至る可能性のある考慮事項も付け加えておきます。

ビジョンは、1970 年代が終わるよりずっと前に、現在プログラミング能力に負担をかけているようなシステムを、現在の人年コストの数パーセントのコストで設計および実装できるようになり、さらにこれらのシステムは実質的にバグがなくなるというものです。これら 2 つの改善は密接に関連しています。後者の点では、ソフトウェアは他の多くの製品とは異なっているようです。他の多くの製品では、一般に品質が高いほど価格が高くなります。本当に信頼性の高いソフトウェアを求める人は、まずバグの大部分を回避する方法を見つけなければならないことに気付くでしょう。その結果、プログラミングプロセスは安価になります。より効果的なプログラマを求める人は、デバッグに時間を無駄にすべきではなく、まずバグを持ち込むべきではないことに気付くでしょう。言い換えると、両方の目標は同じ変化を指し示しています。

これほど短期間にこのような劇的な変化が起きれば革命となるでしょう。そして、最近の過去のスムーズな推定に基づいて将来を予想するすべての人々、つまり社会や文化の慣性の暗黙の法則に訴える人々にとって、このような劇的な変化が起きる可能性は無視できるほど小さいと思われるに違いありません。しかし、革命が時々起きることは誰もが知っています。では、今回の革命の可能性はどれくらいでしょうか?

満たされなければならない主な条件が 3 つあるようです。まず、世界全体が変化の必要性を認識すること、次に、変化に対する経済的必要性が十分に強いこと、そして、**3 番目に、変化が技術的に実行可能であることです。**この 3 つの条件について、上記の順序で説明しましょう。

ソフトウェアの信頼性を高める必要性の認識に関しては、もう異論はないと思います。ほんの数年前までは状況は違っていました。ソフトウェア危機について話すことは冒涜でした。転機となったのは、1968 年 10 月にガルミッシュで開催されたソフトウェアエンジニアリングに関する会議です。この会議では、ソフトウェア危機が初めて公然と認められ、大きな話題となりました。現在では、大規模で高度なシステムの設計は非常に難しい仕事になることが一般的に認識されており、そのような仕事の責任者に会う

と、信頼性の問題を非常に懸念していることが分かります。それは当然のことです。つまり、最初の条件は満たされているようです。

さて、経済的な必要性について。今日では、60 年代のプログラミングは高給取りの職業であり、今後数年間でプログラマーの給与は下がると予想されるという意見をよく耳にします。通常、この意見は不況に関連して表明されますが、これは別の、非常に健全な兆候である可能性があります。つまり、過去 10 年間のプログラマーは、本来すべきほど良い仕事をしなかったということです。社会は、プログラマーとその製品の成果に不満を抱き始めています。しかし、それよりはるかに重要な別の要因があります。現在の状況では、特定のシステムの場合、ソフトウェアの開発に支払われる価格は、必要なハードウェアの価格と同程度であるのが一般的であり、社会は多かれ少なかれそれを受け入れています。しかし、ハードウェアメーカーは、今後 10 年間でハードウェアの価格が 10 分の 1 に下がると予想しています。ソフトウェア開発が現在と同じように扱いにくく、費用のかかるプロセスであり続けると、物事は完全にバランスを失ってしまいます。社会がこれを受け入れることは期待できないので、私たちは桁違いに効果的なプログラミングを学ばなければなりません。言い換えれば、マシンが予算の最大の項目である限り、プログラミングの専門家は不器用なテクニックでやり過ごすことができますが、その傘はすぐに閉ざされてしまいます。つまり、2 番目の条件も満たされているようです。

さて、3 番目の条件は、技術的に実現可能かどうかです。私は実現可能かもしれないと考えており、その意見を支持する 6 つの論拠を挙げたいと思います。

6 つの論拠

プログラム構造の研究により、プログラム(同じタスクで同じ数学的内容を持つ代替プログラムでさえ)の知的管理性は大きく異なる可能性があることが明らかになりました。違反するとプログラムの知的管理性が著しく損なわれるか、完全に破壊される規則がいくつか発見されました。これらの規則には 2 種類あります。

最初の種類の規則は、

適切に選択されたプログラミング言語によって機械的に簡単に課すことができます。例としては、goto ステートメントや複数の出力パラメータを持つプロシージャの除外があります。

2 番目の種類の規則

については、少なくとも私には(ただし、これは私の能力不足によるものかもしれません)、機械的に課す方法がわかりません。何らかの自動定理証明器が必要と思われるためですが、その存在証明はありません。したがって、当面、そしておそらく永遠に、2 番目の種類の規則はプログラマーに要求される規律の要素として現れます。私が考えている規則のいくつかは非常に明確であるため、教えることができ、特定のプログラムが規則に違反しているかどうかについて議論する必要はありません。例としては、終了

の証明を提供せずに、または繰り返しステートメントの実行によって不変性が破壊されない関係を示さずにループを記述してはならないという要件があります。

ここで、知的に管理可能なプログラムの設計と実装に限定することを提案します。この制限が厳しすぎて我慢できないのではないかと心配する人がいるかもしれませんが、安心してください。知的に管理可能なプログラムのクラスは、アルゴリズムで解決できるあらゆる問題に対する非常に現実的なプログラムを多数含むのに十分なほど豊富です。私たちの仕事はプログラムを作ることではなく、望ましい動作を示す計算のクラスを設計することだということを忘れてはなりません。知的に管理可能なプログラムに限定するという提案は、私が発表した 6 つの議論の最初の 2 つの根拠です。

第一の議論は

プログラマーは知的に管理可能なプログラムのみを考慮する必要があるため、選択する選択肢ははるかに簡単に対処できるというものです。

2 番目の議論は

知的に管理可能なプログラムのサブセットに制限することを決定した時点で、検討すべきソリューション空間の大幅な削減が達成されたということです。この議論は 1 番目の議論とは異なります。

3 番目の論点は、

プログラムの正しさの問題に対する建設的なアプローチに基づいています。今日、通常の手法は、プログラムを作成してからテストすることです。しかし、プログラムのテストはバグの存在を示すのに非常に効果的な方法ですが、バグが存在しないことを示すにはまったく不十分です。**プログラムの信頼度を大幅に上げる唯一の効果的な方法は、その正しさを説得力のある形で証明することです。**しかし、最初にプログラムを作成してからその正しさを証明するべきではありません。そうすると、証明を提供するという要件が、かわいそうなプログラマの負担を増やすだけだからです。それどころか、プログラマは正しさの証明とプログラムを手を取り合って成長させるべきです。3 番目の論点は、基本的に次の観察に基づいています。説得力のある証明の構造はどのようなものか自問し、それを見つけたら、この証明の要件を満たすプログラムを構築すると、これらの正しさに関する懸念は非常に効果的なヒューリスティックなガイダンスになります。定義上、このアプローチは、知的に管理可能なプログラムに限定した場合にのみ適用できますが、その中から満足のいくものを見つけるための効果的な手段を提供します。

議論 4 は、

プログラムを設計するために必要な知的努力の量がプログラムの長さに依存するという点に関係しています。必要な知的努力の量はプログラムの長さの 2 乗に比例して増加するという、ある種の自然法則があると言われています。しかし、ありがたいことに、この法則を証明できた人は誰もいません。これは、この法則が必ずしも真実である

必要がないからです。**ごく限られた推論で無数のケースをカバーできる唯一の精神的ツールが「抽象化」と呼ばれることは、誰もが知っています。**したがって、抽象化の能力を効果的に活用することは、有能なプログラマーの最も重要な活動の 1 つとみなされなければなりません。この点で、抽象化の目的は曖昧にすることで*はなく*、完全に正確になる新しい意味レベルを作成することであると指摘しておく価値があるかもしれません。もちろん、抽象化メカニズムが十分に効果的でない原因となる根本的な原因を見つけようとしてきました。しかし、どれだけ努力しても、そのような原因は見つかりませんでした。その結果、私は、抽象化の力を適切に応用すれば、プログラムを考案または理解するために必要な知的努力は、プログラムの長さに比例する以上のものになる必要はないという、これまでの経験では反証されていない仮定に傾いています。しかし、これらの調査の副産物は、はるかに大きな実用的意義を持つ可能性があり、実際、私の 4 番目の議論の根拠となっています。その副産物は、プログラムを作成するプロセス全体で重要な役割を果たす、いくつかの抽象化パターンの特定でした。これらの抽象化パターンについては、現在ではそれぞれについて講義を行うことができるほど十分に知られています。これらの抽象化パターンの親しみやすさと意識的な知識が意味するものは、15 年前にそれらが常識であったなら、たとえば BNF から構文指向コンパイラへの移行は、数年ではなく数分で済んだだろうと気づいたときに、私にはっきりとわかりました。したがって、**重要な抽象化パターンに関する最近の知識を 4 番目の議論として提示します。**

さて、5 番目の議論です。

これは、私たちが使用しようとしているツールが私たち自身の思考習慣に与える影響に関するものです。この影響を無視し、人間の精神をその人工物の最高かつ自律的な支配者と見なすという、おそらくルネッサンスに起源を持つ文化的伝統を私は観察しています。しかし、私自身や私の仲間の人間の思考習慣を分析し始めると、好むと好まざるとにかかわらず、まったく異なる結論に達します。つまり、私たちが使用しようとしているツールと、思考を表現または記録するために使用している言語または表記法が、私たちが何を考え、表現できるかを決定する主な要因であるということです。プログラミング言語がユーザーの思考習慣に与える影響の分析と、現在までに脳力が圧倒的に最も希少なリソースであるという認識は、さまざまなプログラミング言語の相対的なメリットを比較するための新しい基準の集合を私たちに与えてくれます。有能なプログラマーは、自分の頭蓋骨のサイズが厳密に制限されていることを十分に認識しています。したがって、彼はプログラミングの課題に謙虚に取り組み、とりわけ巧妙なトリックを疫病のように避けます。よく知られている会話型プログラミング言語の場合、プログラミングコミュニティに端末が装備されるとすぐに、よく知られた名前さえある特定の現象が発生すると、さまざまな方面から聞きました。それは「ワンライナー」と呼ばれています。それは 2 つの異なる形式のいずれかをとります。1 人のプログラマーが 1 行のプログラムを別のプログラマーのデスクに置き、誇らしげにその動作を説明し、「これをより少ない記号でコーディングできますか?」という質問を追加します(これは概念的な関連性があるかのように!)。または、単に「何をするか推測してください!」と尋ねます。この観察から、ツールとしてのこの言語は巧妙なトリックを歓迎していると結論付けなければなりません。そして、まさにこれが、自分がどれだけ賢いかを誇示したい人

にとって、この言語の魅力の一部を説明するかもしれませんが、申し訳ありませんが、これはプログラミング言語について言える最も非難すべきことの 1 つであると見なさなければなりません。最近の過去から学ぶべきもう一つの教訓は、「よりリッチな」または「より強力な」プログラミング言語の開発は、これらのバロックな怪物、これらの特異性の集合体は、機械的にも精神的にも実際には手に負えないという意味で間違いだったということです。私は、非常に体系的で非常に控えめなプログラミング言語に大きな未来があると考えています。「控えめ」というのは、たとえば、ALGOL60 の「for 句」だけでなく、FORTRAN の「DO ループ」でさえ、バロックすぎるとして捨てられる可能性があることを意味します。私は、非常に経験豊富なボランティアと小さなプログラミング実験を実行しましたが、まったく意図せず、まったく予期しないものが発生しました。私のボランティアの誰も、明白で最もエレガントな解決策を見つけませんでした。さらに詳しく分析すると、これには共通の原因があることがわかりました。彼らの反復の概念は、制御変数を段階的に増加させるという考えと非常に密接に結びついていたため、彼らは明らかなことに気づかずに精神的にブロックされていました。彼らの解決策は効率が悪く、不必要に理解しづらく、解決策を見つけるのに非常に長い時間がかかりました。私にとってそれは啓示的で、また衝撃的な経験でした。最後に、ある意味では、明日のプログラミング言語が、現在私たちが慣れ親しんでいるものとは大きく異なることを期待します。つまり、これまでよりもはるかに大きな程度で、私たちが記述するものの構造に、設計しているものの複雑さに概念的に対処するために必要なすべての抽象化を反映させるように促すはずです。これが、5 番目の議論の根拠であった、将来のツールのより適切な使用法についてのことです。

余談ですが、プログラミング作業の難しさを現在のツールの不十分さとの闘いと同一視する人たちに警告をしたいと思います。なぜなら、ツールがもっと適切になれば、プログラミングはもはや問題ではなくなると結論付けるかもしれないからです。プログラミングは依然として非常に難しいままです。なぜなら、状況的な煩雑さから解放されれば、現在のプログラミング能力をはるかに超える問題に自由に取り組めるようになるからです。

私の 6 番目の議論に異論を唱えることもできます。なぜなら、その裏付けとなる実験的証拠を集めるのはそれほど簡単ではないからです。しかし、その事実が、私がその妥当性を信じるのを妨げることはありません。これまで私は「階層」という言葉には触れていませんでしたが、これは、うまく因数分解されたソリューションを具体化するすべてのシステムにとって重要な概念であると言っても過言ではないと思います。さらに一歩進んで、それを信条にすることもできます。つまり、私たちが本当に満足のいく方法で解決できる唯一の問題は、最終的にうまく因数分解されたソリューションを認める問題であるということです。一見すると、人間の限界に関するこの見方は、私たちの苦境に対するかなり憂鬱な見方のように思われるかもしれませんが、私はそうは思いません。むしろその逆です。私たちの限界とともに生きることを学ぶための最良の方法は、限界を知ることです。他の努力が私たちの知的把握から逃れるため、因数分解されたソリューションだけを試すほど謙虚になる頃には、システムを有益な方法で因数分解する能力を損なうすべてのインターフェイスを回避するために最善を尽くすでしょう。そして、私は、これが、最初は手に負えない問題が結局は因数分解できるという発見に何度

もつながることを期待せずにはいられません。コンパイル段階の「コード生成」と呼ばれるトラブルの大部分が、順序コードの奇妙な特性にまでさかのぼることができることを知っている人なら、私が考えている種類のものの簡単な例がわかるでしょう。うまく因数分解されたソリューションのより広い適用性は、この 10 年間に起こるかもしれない革命の技術的実現可能性に関する私の

6 番目で最後の議論です。

原則として、私の考えをどの程度重視するかは、皆さん自身にお任せします。私が自分の信念を他の人に強制することはできないことは重々承知しています。各本格的な革命は激しい反対を招き、そのような発展に対抗しようとする保守派勢力がどこにいるのか自問自答することができます。私は、大企業、さらにはコンピュータ業界にさえ、そのような勢力が現れることを期待していません。むしろ、今日のトレーニングを提供する教育機関や、古いプログラムが非常に重要で、書き直して改善する価値がないと考えている保守派のコンピュータユーザーグループに現れることを期待しています。これに関連して、多くの大学のキャンパスで、中央コンピューティング施設の選択が、確立されているが高価な少数のアプリケーションの要求によって決定され、独自のプログラムを書く意思のある何千人もの「小規模ユーザー」がこの選択によって苦しむことになるという問題が無視されていることは残念です。たとえば、高エネルギー物理学は、残っている実験装置の価格で科学界を脅迫することが多すぎるようです。もちろん、最も簡単な答えは、技術的な実現可能性を全面的に否定することですが、それにはかなり強力な議論が必要だと思います。残念ながら、今日の平均的なプログラマーの知的限界が革命の実現を妨げるだろうという意見からは、何の安心感も得られません。他の人たちがはるかに効率的にプログラミングしているので、いずれにしても、プログラマーは排除される可能性が高いのです。

政治的な障害もあるかもしれません。たとえ明日のプロのプログラマーを教育する方法がわかっていても、私たちが住んでいる社会がそれを許してくれるかどうかはわかりません。知識を広めるのではなく、方法論を教える最初の効果は、すでに能力のある人の能力を高め、知能の差を拡大することです。教育システムが均質化された文化を確立するための手段として使用され、優秀な人がトップに上がるのを妨げられる社会では、有能なプログラマーの教育は政治的に不可能になる可能性があります。

結論を述べます。

自動コンピュータが私たちの身近に登場してから四半世紀が経ちました。ツールとしての能力で私たちの社会に多大な影響を及ぼしてきましたが、その能力では、人類の文化史上前例のない知的挑戦という能力で与える、はるかに深い影響に比べれば、私たちの文化の表面にさざ波のように広がるだけです。階層システムには、あるレベルでは分割されていない実体と見なされるものが、より詳細な次の下位レベルでは複合オブジェクトと見なされるという特性があるようです。その結果、各レベルで適用可能な空間または時間の自然な粒度は、あるレベルから次の下位レベルに注意を移すと、1 桁減少します。私たちは壁をレンガで、レンガを結晶で、結晶を分子で理解します。その結果、階層システムで意味のある区別ができるレベルの数は、最大粒度と最小粒度の比率

の対数に比例するため、この比率が非常に大きくない限り、多くのレベルを期待することはできません。コンピュータプログラミングでは、基本的な構成要素にはマイクロ秒未満の時間粒度が関連付けられていますが、プログラムの計算には数時間かかることがあります。1010 以上の比率をカバーする他のテクノロジーを私は知りません。コンピュータは、その驚異的な速度のおかげで、高度に階層化された成果物が可能で必要な環境を初めて提供したようです。この課題、つまりプログラミングタスクとの対決は非常にユニークであるため、この新しい経験から自分自身について多くのことを学ぶことができます。設計と作成のプロセスに関する理解が深まり、思考を整理するタスクをより適切に制御できるようになります。そうでない場合は、私の好みでは、私たちはコンピュータにまったく値しないでしょう。

すでに私たちはいくつかの教訓を得ていますが、この講演で強調したいのは次のこと

プログラミングの難しさを十分に理解した上でタスクに取り組む限り、控えめで洗練されたプログラミング言語に固執する限り、人間の心の本来の限界を尊重し、非常に謙虚なプログラマーとしてタスクに取り組む限り、私たちははるかに優れたプログラミング作業を行うことができます。

転写者

改訂 2013 年 2 月 5 日