設計と開発の効果的な実現
— Bjarne Stroustrup からの教訓 —

渡辺大地

本文書では、ソフトウェアの開発と設計のありかたに関して述べる。ただし、ここで述べている話は教訓からなるアナロジー (analogy 類推) であることを留意すること。

1 序説

1. この章で述べられる内容の要約は、次のようなものである。
   - ソフトウェア開発でもっとも重要なポイントは、何を作ろうとしているのかを明確に**把握している**ことである。
   - ソフトウェア開発は、**長期的**な仕事である。
   - システムは、開発者やツールが処理できる複雑さの**限界に達する**ことが多い。
   - 設計、プログラミングにおいて、知性、経験、センスに代えられる “クックブック” 的な方法は**存在しない**。
   - あらゆるソフトウェア開発において、**実験**は非常に重要である。
   - 設計とプログラミングは、**反復的**な作業である。
   - ソフトウェア開発は、設計、プログラミング、テストなどの異なるフェーズに分けられているが、これらを完全に**切り離すことはできない**。
   - プログラミングと設計は、それらの作業の**管理**ということを考えずに出来る仕事ではない。
2. 上記を実際に実践することはとてつもなく難しく、豊富な経験が必要である。理論的な学習のみで身につくものではない。
3. 「プロジェクト」は、往々にして野心的なものが多い。本文書では、そのような場合の設計と開発を前提に置く。(そうでないなら、以前に行ったプロジェクトを真似れば終りである。)
4. あらゆる設計と構築には「唯一の正しい方法」など存在しないし、それを追い求めるのは無駄な努力である。取り組む度に、最良の方法を考えるしかない。
5. ソフトウェアにおける根本的な問題は、**複雑なこと**である。複雑さは、分割して単純にしていく以外に解決策はない。分割は、共同作業を行う唯一の方法でもある。従って、複雑さを分割することで設計と開発の半分以上は終わったと言って良い。
6. 実は、分割自体は易しい。難しいのは、分割した後に双方が効果的な (必ず必要で、かつ余分な部分がない) **コミュニケーション**を実現可能にすることである。

2 目的と手段

1. プログラミングの目的は、ユーザを満足するものを作ること。
   → クリーンな内部構造と柔軟な反応がそれを可能にする。

2. しかし、本来内部構造とエンドユーザは無関係であるべき。なぜ内部構造が重要なのか？
   → テスト、移植、保守、拡張、再構成、理解を容易にするため。

3. 大規模なソフトウェアが成功すると、長い生命を持つ。作業の継承、委託や、新たなハードウェアへの移植、当初は想定外の利用法などが発生する。

4. エンドユーザは、しばしば内部構造を知りたがる。例えば、信頼性のチェックや改訂拡張の可能性の判断など。あるいは、マニュアルには記述されない 'Tips' の存在などにも貪欲である。

5. ソフトウェアは、**全体設計を欠如** させてはならない。同時に、**構造の過度な強調** も避けるべきである。
   → 前者例「今はこれを出荷しておいて、問題は次のリリースで解決しよう」
   → 後者例「今度の版は、古いものより**はるかに**優れているからユーザは遅れることを許容してくれるだろう」
   後者は、大量の計算機資源を必要とする場合が多い。この両者のバランスを取ることは大変難しく、経験のなせるものである。大規模ならなおさらである。

6. 多くのシステム開発現場での分担方法として、デザイナ (設計者) とコーダ (開発者) に分けるということがある。デザイナは開発の詳細には手を触れないし、コーダは意思決定に口出ししない。これには次のような問題が発生する。

   - デザイナとコーダのコミュニケーションが不充分で、経験を生かせず、新機能や迅速な開発や効率を改善できない。
   - コーダに成長がなく、創意もなく、プロジェクトが杜撰になる。

7. 一方、システムの構築、ドキュメンテーション、維持は、何らかの公的な構造で実現する必要がある。「公的な構造」とは、記法や命名法、テストなどのガイドラインのことである。しかし、厳格すぎるシステムは成長と革新を阻害する。「賢い」ことと「本に書かれているとおり」のどちらを優先するかの選択は多くのジレンマを生む。

8. 解決の糸口は、長期的な視野に立って計画することである。

9. 設計の目的は、**明確で単純な内部構造**をプログラムに与えることである。これをプログラムの**アーキテクチャ**と呼ぶことがある。

10. 設計は、デザイナとプログラマ、そしてプログラマ同士のコミュニケーションの中心に置かれるものである。設計の実体は、時と場合によって最適なものが異なる。個人のレベルであれば走書きで充分だが、数百人のプロジェクトでは分厚い設計書が何冊も必要となる。適切な水準の判定も管理者の難しい仕事である。

11. これより後の解説では、クラス構造で設計を解説するが、実際の設計にはもっと多くの要素 (並列処理、データベース、移植など) が関わってくる。

3 開発プロセス

1. 一般に、開発プロセスには**始まりも終わりもない**。

   - はじめは、誰かが作った設計、サンプル、ライブラリ、開発用ソフトウェアを基礎としてスタートする。

- 作成後のプログラムは、他人や自分によって利用、改訂、改良、参照、移植などが行われていく。

2. 従って、自分がまっさらなところからスタートする**つもり**になっていることは重大な問題を引き起こす。

3. 同様に、“最終出荷” によって世界が終わるつもりになっていることも、その**後継者たち** (未来の自分かもしれない) に重大な問題を押し付けることになる。

4. つまり「設計」とは、以前の設計と経験に基づく**再設計**なのである。

5. 開発プロセスには、次の三つのステージがある。

   - 分析: 解決すべき問題の範囲を明確にする。
   - 設計: システムの全体構造をつくる。
   - 実装: コードを書き、テストする。

   このプロセスが**反復的**な性格を持つことを忘れてはならない。(ステージには番号はついていない!!) ソフトウェアの「保守」とは、このプロセスをひたすら反復するだけのことに過ぎない。

6. 最も重要なことは、「分析」「設計」「実装」が解離しないこと
   → これらの作業に関わる人々が効果的なコミュニケーションのために文化を共有することである。しかし、大きな組織やプロジェクトではそのようにいかないことが多い。

7. 小、中規模プロジェクトでは、分析と設計は一つに統合されている場合が多い。さらに小規模プロジェクトでは、設計とプログラミングの間に区別を置かないこともよくある。

   - メリット: コミュニケーションの問題を解決できる。
   - デメリット: 不充分な設計や杜撰な運用を生む。

   正しい方法は**一つではない**。

8. 伝統的なソフトウェア開発モデルは、各ステージを完全に消化してから次のステップに進む (そして後戻りしない) もので、これを「**カスケードモデル**」とか「**トップダウンモデル**」と呼ぶことがある。

9. カスケードモデルは、情報が**一方通行**になるがちだという根本的な欠陥を抱えている。“下流” で問題が見つかっても、**局所的に解決せよ** という組織的な強い圧力がかかるのである。このようなフィードバックの欠如は**不充分な設計**を導き、局所的な訂正は**歪んだ実装**を招く。

10. 上流の変更には時には**膨大な手間**を生ずるときがあるので、必ずしも「局所的な訂正」が悪とは言えない。しかし、カスケードモデルでは生じた問題が手に負えなくなる可能性を大幅に上げてしまう。

11. 一方で、開発プロセスの各ステップを反復的に行う開発モデルを「**スパイラルモデル**」とか「**反復的モデル**」と呼ぶことがある。スパイラルモデルでは、下流で生じた問題への対応が**飛躍的に向上する**。

12. しかし、スパイラルモデルではプロジェクトがいつまでも**終わらない**という危険性を持つ。この問題もカスケードモデルの問題も、解決するよりは診断の方が簡単である。

13. どちらにしろ重要なことは、プロジェクトがどのような段階であっても分析・設計・実装のうち 1 つだけ (他を排除するような形で) 重点を置くようにはしないこと である。

14. 何をやるにしても、**明確な目標や中間目標**を定義すべきである。それを怠ると、どんなに適切な管理テクニックを駆使しようが、高度なテクノロジーを潤沢に活用しようが、**何の役にも立たない**。

15. 一方で、適切なテクノロジーは例え金額的/人材的/労力的投資が必要な場合でも積極的に利用すべきである。適切なツールと良好な環境の元では、人はよりよい仕事をするものだ。しかし、それ (適切なテクノロジーの採用) を**簡単なことだ**などと**勘違い**してはならない。

3.1 開発サイクル

1. システム開発は、反復的な作業でなければならない。メインループは、以下のシーケンスの繰り返しである。

   (a) 問題を解析する。

   (b) 全体設計を作る。

   (c) 標準コンポーネントを見つける。
   → この設計に合わせてコンポーネントをカスタマイズする。

   (d) 新しい標準コンポーネントを作る。
   → この設計に合わせてコンポーネントをカスタマイズする。

   (e) 最終設計を組み立てる。

2. コンポーネントの想定と作成は非常に重要な仕事である。

   - 前のプロジェクト中のコンポーネントは利用できるか？
   - 必要なコンポーネントは外から調達できないか？
   - 調達したコンポーネントはどの位今回のプロジェクトで**満足**できるか？ もし利用するとなると、**妥協点**はどんなものか？ それは許容範囲か？

3. プロジェクト中で新たに作るコンポーネント (ユニット) は、外でも使い物になるほど**汎用性**が高いことが望ましい。しかし、汎用性の追求は機能性を阻害する可能性もある。

4. このような開発モデルが真価を発揮するのは、長期的な視野に立ったときだけである。「今回のプロジェクト」だけで見れば、単なる**オーバーヘッド**となる。

5. 一方、“標準コンポーネント” が**普遍的な標準**であると期待するのは合理的な考え方ではない。適用範囲と機能は多くの場合で**反比例**するので、そのコンポーネントがどの程度の適用範囲と機能を持つかを適切に判断する必要がある。

6. 最初の設計で普遍性を狙えば、そのプロジェクトは確実に**終わらない**。経験を得るためには**動く**システムがあることが本質的に重要な意味を持つ。開発サイクルが「サイクル」である理由の一つはこれである。

3.2 設計目標

1. 設計は、発展できなければならない。つまり、システムはすべて**予見できるわけではない**形で拡張、移植、チューニングなどの様々な変更を受けなければならなくなる。

2. 従って、システムは数回の変更を経ても、出来る限り**単純なまま**でいられるように設計しなければならない。

3. これを実現するには、変更を受けそうな領域を**カプセル化**しておく必要がある。そのためには、アプリケーションの主要コンセプトを見極め、単一のコンセプトに関連したすべての情報を 1 つのクラスに全面的に管理させる。そうすれば、変更を加えるために必要な仕事はクラスの変更だけになる。

4. 例: 気象シミュレーションでの雨雲の表示

   - 雲を表示する汎用ルーチンの作成。
     → 「雲」クラスの内部状態が規定できないので作成不能。
   - 雲に自分自身を描画させる。
     → 多くの場面でそれなりには適切なアプローチである。しかし、描画環境 (色数や表示機能など) の違いに対応できない。
   - 雲に環境を意識させた上で自分自身を描画させる。
     → 異なるコンセプトが 1 つのクラスに共存してしまい、よいアプローチではない。(このクラスが**何を意味しているのか**よくわからなくなる。)
   - 雲がビューワー (の API) に対して自分自身を記述する。
     → よいアプローチだが、複雑さや速度に問題が生ずることがある。

5. 一般に、長い寿命を持つプログラムの適切な抽象レベルは、作成者が理解でき、**実現できる**なかで最も一般性の高いものなのであって、絶対的に最高の一般性を持つものでは**ない**。プロジェクトの守備範囲を超え、関わっている人々の経験を超えるような一般化は、**害**をもたらすことがある。

3.3 設計の手順

1. 単一のクラスだけを設計するという方針は、よい考えでは**ない**。コンセプトは独立して存在するものではなく、他のコンセプトとの**関わり**の中で定義されるものである。

2. 一般に、デザイナやプログラマは一連の関連しあうクラスを相手に仕事を進める。このクラスの集合は**クラスライブラリ**とか**コンポーネント**と呼ばれることが多い。

3. あるコンポーネントの中のクラスを使うときに、プログラマはできるだけ少ないクラスで仕事を実現しようとする。この行為は正しい。しかし、コンポーネントの一部を利用するときには、コンポーネントを定義する**論理基準**、コンポーネントの背景を流れる**習慣**や**スタイル**を理解する必要がある。

4. コンポーネントの設計を、次のようなステップに分割して考える。

   (a) コンセプト/クラスを見つけ、それらの間の中で最も根本的な意味のある**関係**を探す。

   (b) 加えられる演算の集合を規定しながら、クラスを磨いていく。

   - 演算を分類する。
   - 構築、コピー、解体のニーズを考える。
   - サイズを小さくすることを考える。
   - 完全を期することを考える。
   - 使いやすさを確保することを考える。

(c) 依存関係を規定することによってクラスを磨いていく。パラメータ化、継承、依存関係の利用を考える。

(d) インターフェース (public メソッド) を規定する。

- 関数 (メソッド) を公開、限定公開関数に分類する。
- クラスに加えられる演算やメソッドのタイプを明確に規定する。

5. 上記の設計手段も、**反復的**なプロセスであることを忘れてはならない。一般に、初期実装、再実装で快適に使える設計を完成させるためには、このシーケンスを**数回**繰り返す必要がある。

6. この作業のメリットの 1 つは、コードを書いてしまった後でも、クラスの関係の組み替えが**比較的楽**になることである。しかし、クラスの関係の組み替えが**とても簡単になる**ことは決して**ない**。

3.4 ステップ 1: クラスを見つける

1. 最初に見るべきところは、コンピュータ科学者の抽象化、コンセプトのコレクションではなく、**アプリケーション自体**である。完成後の**ユーザ**になるべき人や、使っているシステムに不満を持つ**ユーザ**に聞くとよい。彼らが使う**語彙**に注意する。

2. コンセプトの共通性は、**継承**を使って表現することができる。特に、継承ではコンセプトを**階層構造**で表現できることが重要である。これは、洞察力を必要とするレベルの高い作業である。

3. 分類方法は、他の分野で正しいとされる「ことがら」ではなく、システムに実体化しようとしている「ことがら」に基づいたものでなければならない。例えば、長方形は多角形の**一種**だが、多角形から長方形を派生させたり、その逆を行うことは本当に**嬉しいことだろうか**？

4. 重要なコンセプト/クラスを最初に見つけるときに最も役に立つ道具は**黒板**である。初期段階での改良の最良の道具は、専門家や友人との**議論**である。一人でできるひとは**まずいない**。

5. 次に、システムをクラス構成に従ってデザイナが**シミュレート**し、漏れや欠陥を探す。

6. しかし、機能ばかりに重点を置くと、構造設計案が破綻することがある。利用例をいくら集めても、それが設計になるわけではない。システムの利用形態に強調を置くのと同じくらい、システムの構造にも焦点をあてる必要がある。

7. 開発チームは、**すべての**利用例を見つけて記述しようという努力に捕らわれがちだが、これは**無駄な努力**である。「一旦、手持ちのものを試してどうなるか試してみる時期が来た」と言わなければならないタイミングが**必ず**やって来る。しかし、このタイミングを見極めることは常に難しい。

8. 前節で説明したコンセプト、演算、関係は、応用分野の理解から自然に得られるものと、クラス構造を検討していく過程で明らかになるものがある。これは、(作成しようとしている) アプリケーションについての**理解内容**を示しており、基本コンセプトの**分類結果**であることが多い。

3.5 ステップ 2: 演算を規定する

1. 加えられる演算の集合を規定しながら、クラスを磨いていく。当然ながら、クラスを見つけていく作業と、そのクラスで必要とされる演算を明らかにしていく作業を分離することは**できない**。

2. 演算は、次のようなものに大別するとわかりやすい。

(a) 基本演算: コンストラクタ、デストラクタ、コピーなど

(b) 状態参照演算: オブジェクトの状態を (変更しないで) 参照する演算

(c) 状態変更演算: オブジェクトの状態を変更する演算

(d) 変換演算: オブジェクトの状態に基づき、別型のオブジェクトを生成する演算

(e) 要素取得演算: オブジェクト内にある要素 (子オブジェクトなど) を取得する演算

ただし、これらの要素は直交的[1] ではない。たとえば、要素取得演算は状態参照とも状態変更とも言えるケースがありえる。

3. どのような演算を提供すべきかを考えるときには、いくつかの考え方がある。一例として、次のようなものがある。

   (a) クラスオブジェクトが、どのように構築、コピー (必要なら)、解体するかを考える。

   (b) クラスが表現しているコンセプトが必要とする**最小限**の演算を定義する。

   (c) どの演算が仮想関数[2] になるかを考える。

   (d) コンポーネントのすべてのクラス全体に命名、機能上の統一性を得るためにはどうしたらよいか考える。

4. 重要なのは、できるだけ演算の数を**増やさない**ことである。確かに、役に立ちそうなものを全て追加作成し、全ての演算を仮想関数にしてしまうのが最も楽である。しかし、このようなクラスは実装段階、利用段階、再設計段階の全ての部分で大きな制限を設けることとなる。

5. かなり早い時点で仮想関数の検討を行うことには理由がある。それは、クラス同士の**関係**に対して決定的に重要な影響を及ぼすからである。

6. 演算を選択するときには、「どのようにクラスを**実装**するか」よりも、「クラスの**望ましい動作**とはどんなものか」に集中すべきである。

3.6 ステップ 3: 依存関係を規定する

1. 設計の文脈で考慮すべき重要な依存関係は、**パラメータ化**、**継承**、**利用**関係である。これらはそれぞれ、クラスが取るべき責任とは何かということについての考察を反映している。

2. 「責任を取る」ということは、クラスが全てのデータを抱え込むとか、メンバ関数が必要な演算をすべて直接行うという意味ではない。あくまで「自分の仕事をこなす」という点が重要である。当然、「特定の仕事を他のクラスに**委託**する」ことも多く存在する。

3. しかし、あまりに**委託**の過ぎる設計は、当然のことながらクラスの種類が膨大になり、効率や理解しやすさに問題が出る。

[1] それぞれのカテゴリー内の要素が、他のカテゴリーには入らないこと。
[2] C++ では「virtual」キーワード、Java では「abstract」キーワードとなる。

3.7 ステップ 4: インターフェースを規定する

1. 'インターフェース' とは、最終的にクラスに対してアクセスするための手段一般のことで、メンバ関数、継承、コールバックなどの手段がある。

2. インターフェースを規定するときには、メンバ関数が異なる抽象レベルをサポートするように見えるクラスに注意する。例えば、ファイル名文字列とファイルデスクリプタの両方をサポートする場合、本当に一つのクラスで扱うべきなのかどうかを考慮する必要がある。

3.8 クラス階層の再構成

1. ステップ 1 と 3 で、クラスとクラス階層がプロジェクトのニーズに充分応えられているか検討したが、応えられていない場合 (通常はこちらである) はクラス階層、設計、実装の**再構成**が必要となる。

2. クラス階層の再構成の最も一般的な形態は、次の 2 つである。

   - 2 つのクラスの**共通部分**を抽出して新しいクラスを作る。
   - 1 つのクラスを 2 つのクラスに**分割**する。

3. いずれにしても本質的な仕事は、クラスの共通部分を探してそれを抽出することである。共通性の明確な基準はないが、単に実装の便宜のためというのではなく、システムの**コンセプト**を反映していなければならない。

4. 共通性の抽出は、次のような点がヒントになる。

   - 利用時の共通パターン
   - 演算の集合の類似性
   - 実装の類似性
   - 設計の議論で 1 セットで扱われるかどうか

5. 一方、次のような点がある場合はクラスを分割するとよい。

   - 通常とは**異なる**利用パターンのメンバ関数の存在
   - クラスが表現しているコンセプトの中の、**特定の一部**のみにアクセスしているメンバ関数の存在
   - 設計の議論と明らかに**無関係**な文脈を含むクラス

6. 設計の最も重要な目標の 1 つは、変更を経ても安定性を保てるようなインターフェースを提供することである。これは、依存するクラスが多ければ多いほど、依存されているクラスの**一般性**は高くし、そのクラスが外に見せる**ディティール**は少なくすることで達成できる。

7. 多くのクラスから使われるクラスには演算 (及びデータ) を**増やしたい**という欲求が強く働く。そうすれば、そのクラスはより使いやすく、将来必要になる変更は減るような**気がする**のである。しかし、これは「ファットインターフェース」を導き、次のような問題点が起こる。

   (a) ファットインターフェースを持つ「A」というクラスを利用しているクラスは、「A」の機能に強く依存することになる。

   (b) サポートしている多くのクラスの中のどれかに、大幅な変更を加えると、「A」の方も変更しなければならなくなる。

(c) それにより、「A」を利用する全クラスで変更が必要となる。

8. 従って、クラス構成は根の近くでは最も一般性が高く、他のほとんどのクラスや関数がそれに依存するような抽象クラスを準備すべきである。同様に、枝葉のクラスは専門分化していて、それらのクラスに直接依存するコードはごくわずかとなるべきである。

3.9 モデルの利用

1. 例えば、ある書籍の第 3 判を出す場合、第 1 判、第 2 判をベースに記述される。このように、新しい設計のために既存のシステムを利用することは頻繁に行われる。むしろ、標準的な方法だと言って良い。設計とプログラミングは、可能な限り既存の仕事をベースとして扱うべきである。

2. モデルは、それを持つことによって**制約**を設けることでは**ない**し、そのモデルに**隷属的**に従う必要もない。モデルは、モデルの 1 つの側面に考えを集中させられるように、デザイナを解放するものである。当然ながら、初期モデルの選択はそれ自体重要な設計の**意思決定**である。

3. 多くの設計で繰り返し現れる一般的な要素と、それらが解決する設計問題、それらを利用できる条件の簡単な説明を「**デザインパターン**」あるいは単に「**パターン**」と呼ぶことがある。デザインパターンに関する書籍も多く存在する。

3.10 実験と分析

1. 野心的な開発プロジェクトでは、**ディティール**がシステムを構築、テスト、利用していく過程で**しか**明らかにならないので、「システムが何をすべきか」ということさえも正確には**知らない**ことが多い。

2. では、完全なシステムを作らずに、設計上の何が重要なのかを理解し、波及効果を見積もるために必要な情報を得るにはどうしたらよいか？

3. 「実験」である。分析**できる**設計や実装ができたならば、それらを分析することも重要である。

4. 実験のもっともポピュラーな形態は、**プロトタイプ**すなわちシステムの小規模版、あるいはシステムの一部を**実際に**作ることである。プロトタイプは一般に、教育、環境、経験、意欲全てが例外的に優れているデザイナ、プログラマの手によって作られることが多い。

5. このアプローチの最大の問題は、プロトタイプ開発の力点が「設計の選択肢の検討」から「できるだけ早く**稼働する**なんらかのシステムを手にいれること」に容易にシフトしてしまうところにある。そのため、プロトタイプの内部構造には注意が払われず (「どうせプロトタイプだから」)、実装ばかりを進めて設計の努力を無視する傾向が現れる。

6. そして厄介なことは、そのようにして出来たプロトタイプに対して「ほとんど完成した」という**幻想**を与えることである。そもそも、プロトタイプはスケールアップに耐えうる内部構造、処理性能、維持基盤を持たない。

7. 結果として、開発者や特に**マネージャ**がプロトタイプをそのまま**リリース**し、次のリリースまでに「パフォーマンス改善」を延期したい**衝動**が働く。このように、プロトタイプは方向性を誤ると設計のあらゆる根拠を奪う結果となる。

8. 一方、プロトタイプは大きな価値を持つことがある。たとえば、ユーザインターフェースの設計では内部構造とは無関係であり、ユーザの反応を見る以外に改善策は考えられない。同様に、システムの内部動作の研究にはインターフェースは大雑把なものでよく、ユーザも本物である必要はない。

9. プロトタイプは、あくまで**実験手段**である。プロトタイプの成果は、それ自体ではなく、実装、実験による洞察である。プロトタイプを**不完全**なものに保つことで、それを製品版に転化する危険性を避けられる。使用後のプロトタイプは、**捨て去るべき**である。

10. 多くの場面では、プロトタイプ以外にも実験のテクニックが**ある**。それが存在するならば、そちらの方が**厳格**で、デザイナの時間やシステム資源に対する需要は低いことが多いので、そちらの方が望ましい。例えば、数学モデルやシミュレータなどである。

11. 仕様 (分析段階の出力) と設計は、実装と同じくらい**エラー**を含みがちである。実際、仕様や設計は実装よりも具体的ではなく、正確に規定されていないことが多く、実装以上にエラーを含む可能性が高い。

12. しかし、仕様や設計を**厳格な記法**で記述するのにも問題の原因となることもある。例えば、適当対象の実践的な問題に**形式主義**が向いていない場合や、厳格さがデザイナやプログラマの能力を**上回る**場合などがある。

13. 設計は本質的にエラーを**含む**。そのため、実験とフィードバックが重要な意味を持つ。結局、分析に始まりテストに終わる**直線的**なプロセスとして開発を捉えることは、根本的に間違っているのである。

3.11 テスト

1. テストを経ていないプログラムは**動作しない**。よほど簡単なプログラムでない限り、最初から動作するという理想は**到達不可能**である。

2. 「どのようにテストするか」は一般的な解答がないが、「いつテストするか」には明確な解答が存在する。「できるだけ早く、できるだけ頻繁に」である。

3. テストは、設計、実装作業の**一部**として作るべきで、稼働するシステムができると**同時**にテストを開始しなければならない。実装が完成するまでテストを先延ばしすることは、間違いなく**日程の遅れ**と**欠陥リリース**をもたらす。

4. システムは、比較的テストがしやすいようにシンプルな設計にすべきである。実行時テストは高くつくとか、チェックのためにデータ構造が異常に膨らんでしまうという批判があるが、これは一般的に**的外れ**である。リリース版にテスト部分を取り除くには様々なテクニックがある。

5. テストで重要なのは、それ自身よりも、エラーを容易に分析、修正ができるようなシステムの**発想**の方である。

6. どの程度テストすれば充分かの見極めは難しいが、テスト過小の方がテスト過多よりも頻繁に起きる問題である。

3.12 効率

1. D.E.Knuth は「**半端**な最適化は諸悪の根源である」と言った。これは確かに真だが、設計、実装作業では常に効率のことを忘れてはならない。

2. しかし、だからといってデザイナが細かな効率のことを考えなければならないということではない。システムに適したデータ構造など、一時的な効率問題を考えなければならないということである。

3. 効率を上げるための最良の戦略は、**クリーン**で**シンプル**な設計を作ることである。人々は、“**もしものとき**” のために機能を追加してしまい、システムのサイズと実行時間を 2 倍、4 倍にしてしまうことがあまりにも多い。そのようなシステムは、分析が困難になってしまい、効率を上げることが難しくなる。

4. 最適化は、基本分析段階では避けるべきである。これは、**分析**と**性能計測**の結果であるべきで、コードのランダムな操作によってはならない。特に、大規模なシステムでは開発者の**直感**は当てにならない。

4 管理

1. 開発作業は、実際には設計、開発、管理の三つに分割される。その 3 フェーズのどこかで**スタイル**が変わった場合、他の部分もそれに合わせてスタイルを変更する必要がある。

   悪い例 1:
   C で開発されたシステムを、オブジェクト指向の概念を無視して C++ に移植する。

   悪い例 2:
   C で開発されたシステムを、オブジェクト指向の概念を導入するのにも関わらず、C のままで開発を進める。

4.1 再利用

1. 新しいプログラミング言語や設計戦略を採用する理由として、コードや設計を**再利用**しやすくなることがよく挙がる。

2. しかし、多くの組織では 0 からやり直すような行為を推奨する体制になっている。

   例 1:
   コードの**行数**で報酬が決まる。

   例 2:
   プロジェクトの**人数**に比例して必要経費が算出される。

   このような文化では、再利用が評価されることはないだろう。

3. 再利用は、次のような条件が満たされれば可能となる。

   (a) **見つけられる**。
   (b) **動作する**。願わくば、なんらかの実績がある。
   (c) **理解できる**。
   (d) そのソフトウェアと共存を意識していないソフトウェアとも共存**できる**。
   (e) **サポート**されている。あるいは、自分でサポートする気になれる。
   (f) **経済的**に現実的である。(購入費、維持費など)

4. 再利用を実現するには、それを実現することを**目的**とする人員が**いる**ときだけである。規模が小さければある**個人**が管理を担う。規模が大きければ、多くのグループが使うソフトウェアの収集、構築、文書化、普及、維持を**専門**とするグループあるいは部門が担うことになる。このような**標準コンポーネントグループ**の重要性は、いくら強調してもし過ぎることはない。

5. 大雑把に言って、システムはそれを作った組織を反映するものである。協力と共有を推進し、評価するメカニズムを**持たない**組織では、協力や推進は**実現できない**。標準コンポーネントグループは、コンポーネントの利用を積極的に推進しなければならない。それには、**教育**や**マーケティング**といった側面も持つ。

6. コンポーネント部門は、組織の**コンサルタント的**な役割を果たすべきである。従ってコンポーネント部門のメンバーは、可能な限り開発部門と密接に連絡を取り合って仕事をすべきである。

7. 「再利用可能なコンポーネント」とは、特定の枠組みの中ではほとんどあるいはまったく労力を書けずに再利用できるコンポーネントのことである。全てのコードが再利用可能である必要は**なく**、再利用が可能だという属性は普遍的なものではない。ほとんどの場合、異なる枠組みのなかでコンポーネントを利用するには**努力**が必要となる。

8. 再利用は、再利用を**最初から**意図した設計、経験に基づくコンポーネントの**改良**、(再) 利用できる既存のコンポーネントを**探す**意識的な作業の成果であることを忘れてはならない。特定の言語機能やコーディングテクニックを考え無しに使っても、再利用が**魔法のように**可能になるわけではない。

4.2 スケール

1. 個人でも、組織でも、

   「正しいことをした」

   ということに大喜びしてしまうのは**簡単**なことだ。処理を**改良したい**という純粋で熱狂的な意欲の、最初の犠牲になるのは

   常識

   である。不幸なことだが、一度常識が失われると、無意識のうちに受けるダメージには**際限がない**。

2. 前節で延々と述べた事項を忠実に守り、設計と開発を交互に綿密に進めることは、大抵のプロジェクトの成功を助けることになるだろう。しかし、このような設計メソッド、あるいは一連のコーディング、ドキュメンテーション標準への準拠が**「正しい方法だ」**という考え方が認められてしまうと、それを普遍的にあらゆるディティールに適用せよという**圧力**が生まれる。

3. 小さなプロジェクトでは、これが馬鹿げた**制約**や**オーバーヘッド**の原因になりかねない。特に、生産的な仕事ではなく、紙のやり取りや帳票の入力が、成功や進行の尺度になってしまうことがある。こうなると、本物のデザイナやプログラマはプロジェクトから離れ、官僚が集まるようになってしまう。

4. 設計の完全性をどのように保っていくかは、あらゆるプロジェクトにとって重大な問題である。巨大プロジェクトの全体的な目的を把握し、ずっと見つめていられるのは、**個人かごく少人数のグループ**に限られる。

5. ほとんどのメンバーは、時間の大半を部分プロジェクト、技術的ディティール、日常の管理に費やさなければならないため、プロジェクトの全体目標は**忘れ去られたり**、局所的な目前の目標の影に霞んでしまったりしている。

6. 従って、設計の完全性の維持を明示的に担当する個人やグループを持たないことは、失敗の大きな原因になる。同様に、それらの個人やグループにプロジェクト全体への**影響力**を与えられないことも、失敗の大きな原因になり得る。

7. 一貫した**長期的**な目標の欠如は、あらゆる種類の個人的な能力の欠如よりも、プロジェクトと組織に多大なダメージを与える。そのような全体目標を公式化し、その目標を常に忘れずに、主要な全体設計のドキュメントを書き、重要コンセプトの入門文書を書き、他の人々が全体目標を忘れないようにする活動全般を担当する、少人数のグループを置くことは是非とも必要である。

4.3 個人

1. マネージャは、組織が**個人**によって構成されているということを忘れがちである。プログラマはみな等しく、交換可能だという考え方が広く普及している。

2. しかし、これはもっとも有能な個人の多くを失い、残った人々にもそれぞれの能力以下のレベルで仕事をさせることを強制する考え方である。個人が交換可能なのは、当面の仕事で必要とされる絶対最小限以上に、能力を発揮することが**認められていない**場合だけである。

3. プログラミングの職務評価基準の大半は、**無駄の多い仕事**を評価している。最も顕著な例として、以下のようなものがある。

   - 書いたコードの行数。
   - ドキュメントのページ数。
   - パスさせたテストの数。
   - 修正したバグの個数。

   このような数字は、管理チャートでは**見栄えがする**かも知れないが、現実をあまりにも**希薄**にしか反映していない。

4. 作業の質の計測は、出力量の計測と比べて**遥かに難しい**が、個人とグループは乱暴な量の計測によってではなく、仕事の**質**によって評価されなければならない。

5. しかし、プロジェクトの状態を不完全に記述するような計測方法は、開発の行方を**ねじ曲げがち**である。人々は、計測方法に規定された形で局所的な合格基準に順応しようとする。その直接の結果として、全体設計の完全性や性能が**犠牲**になる。

   例 1: バグの修正個数による評価　→　バグを修正するためには性能の低下を厭わない。

   例 2: ベンチマークによる評価　→　バグやエラーの確率は確実に上昇する。

6. 包括的で優れた品質計測基準が**ない**ということは、マネージャたちに専門的な能力を要求されるということである。マネージャも**個人**だということを忘れてはならない。マネージャも、新技術に対して少なくとも管理されている人々と**同様の教育**を受けていなければならない。

7. 個人の貢献度を判断することは、1 年分の仕事の出来だけでは本質的に不可能であり、長期的な視点が必要である。

8. ほとんどの個人は、長期的に**一貫**した記録を残しているものであり、それは技術的な判断を下すときの信頼できる材料になり得るし、最近の仕事ぶりの評価の基礎にもなる。

9. 長期的な視点を持ち、「交換可能な愚者達のマネージメントスクール」的な視点を避けると、(プログラマとマネージャの双方の) 個人の成長期間を**長め**に取らなければならなくなる。目標は、重要な技術者とマネージャの両方の配置転換を**少なくする**ことである。

10. 重要なデザイナ、ディベロッパとの**協力関係**や最新の重要な**専門知識**を持たないマネージャは成功できない。逆に、有能なマネージャの**サポート**がなく、自分が働いている環境の技術面以外の状況や文脈についての最小限の理解を持てなかったデザイナ、プログラマも、長期的には成功をつかめないだろう。

4.4 ハイブリッドな設計

1. 上級技術者、アナリスト、デザイナ、プログラマといった人種は、新しい技術を学ぶとともに、多くの場合で古い習慣を捨てなければならない人々である。

2. これは簡単なことではない。これらの人々は、かつて新しかった (そして今はもう**古い**) 方法を学ぶために個人的に**かなりの投資**をしていることが多く、これらの方法を使って勝ち取った成功をもとに現在の技術的な評価を築いている。これはマネージャにも当てはまる。

3. 当然ながら、これらの人たちの間には変化への恐怖心があることが多い。そのため、変化に伴う問題を**過大評価**し、古い方法による問題を**認めたがらない**傾向がある。同じように、変化の必要性を論じる人々は新しい方法のメリットを**過大評価**し、変化に伴う問題を**過小評価**する傾向を持っている。この二種類の人々が協力できないと、組織は麻痺し、最も有能な人々が両方のグループから離れていってしまう。

4. 組織への新しい方法の導入には、苦痛を伴うことがある。特に、一夜にして「古い (やり方をとってきた) 学校」の生産的で傑出したメンバーが「新しい学校」の**出来の悪い初心者**に変わってしまうような急激な変化は、反発を呼ぶ。しかし、変化なしで大きな成果が得られることは**稀**だし、大きな変化には**リスク**が伴うものである。

5. 多くの場合は、移行の管理方法に対する配慮によって改革の意欲は緩和されるものであり、緩和すべきものである。以下の要素を考えるべきだ。

   - デザイナとプログラマは、新しい技能を獲得するために**時間**を必要とする。
   - 新しいコードは古いコードと**共同**で仕事をしなければならない。
   - 古いコードは**維持**を必要とする。(**無限**に必要な場合もよくある。)
   - 既存の設計とプログラムの作業も (時間内に) **完成**させる必要がある。
   - 環境に新テクニックを**サポート**するツールを導入する必要がある。

   これらの要素は、必然的に**ハイブリッドスタイル**[3] の設計を要請する。たとえそれがデザイナの意図に**反する**ものであっても。最初の 2 つのポイントは特に**過小評価**されがちである。

6. C++ は、**複数**のプログラミングパラダイムをサポートすることによって、様々な形で段階的な導入という概念をサポートしている。

   - プログラマは C++ の学習過程でも**生産性**を維持できる。
   - C++ はツールが**貧弱**な環境でも大きな成果を上げられる。

[3] 複数のプラットフォームや言語、あるいは設計指針を同時にサポートすることを念頭においたスタイルのこと。

- C++ プログラムの断片は、C を始めとする**古い言語**で書かれたコードとうまく共存できる。
- C++ は、大規模な **C 互換サブセット**を持っている。

7. うまく設計されたライブラリを**利用**することは、それを設計、実装することよりも遥かに簡単である。そのため、初心者でも早い段階から**抽象**のより高度な使い方によるメリットを享受できる。