

ソフトウェア設計の分析・原理・論理

日 野 克 重
(富士通株式会社)

1. まえがき

現在のわれわれのソフトウェア開発のやりかたは、はなはだ試行錯誤的であるといわざるをえない。わたくしがこう言うのは、たとえば、次のような状況のことを指している。すなわち、(i) 設計の過程で採られる数々の選択に対して正当性の証明がつけられることは皆無に等しい。(ii) レビューをやっても、完全にレビューしきったという確信がもてることはない。(iii) 計算機上でのテストは、膨大な労力と時間をかけて行っても、所詮、完全網羅的ではあり得ない。

それでは、このような状況を好転させるために、現在のソフトウェア工学は十分助けになるだろうか。わたくしにはそうは思えない。わたくしは、現在のソフトウェア工学には、一言でいえば、「深み」が欠けていると思う。殊に、ソフトウェア工学の中核ともいえるべきソフトウェア設計理論についていえば、わたくしは次の3つの不満を感じている。そしてこの3つは相互に関連している。

<不満1> 個々の設計事例についての深い分析が欠けている。個々の設計事例をよく見て、まずは決定論としてのソフトウェア設計理論を確立すべきだと思う。

<不満2> 設計原理への志向性が欠けている。ソフトウェア設計理論がひとかどの理論となりおこせるためには、そこに、理論の核としての原理が必要であるが、この原理への志向性が乏しい。

<不満3> 設計の論理が脆弱である。現在のソフトウェア設計理論には、設計の論理あるいは証明の概念がほとんど決定的に欠けている。プログラムの正当性の証明の研究もあるが、それらは、多くのソフトウェア開発者にとって、あまり現実性のないものとして映っている。

本稿では、これらの不満点のそれぞれについて、それを埋めるための具体的な提案をする。なお、本稿で使用する「設計」という言葉は、ソフトウェア開発における知的活動のすべてを含んでいる。したがって、通常は設計と区別されるプログラミングの作業なども、ここでは設計の一部とみなされている。

2. ソフトウェア設計の分析

設計の明らかな失敗事例として、バグがある。このバグの成り立ちをよく見ることは、ソフトウェア設計にあたって人間の思考がどのように働くものかをよく見ることにつながる。本章では、バグ分析の一手法として、深層的バグ分析を提案する。

2.1 深層的バグ分析

バグの分析なるものは、ひろく行われている。しかし、それらは、バグの集合についての単なる統計的分析や分類であったり、あるいは、「バグを作らないようにもっとガンバッていればバグを作らなかった」というようなトートロジカルな結論しか導きだしていないいわば浅い分析であることが多い。これに対して、深層的バグ分析では、個々のバグについて、それを作り込んでしまった根本の原因を、深くかつ決定論的に分析する。そしてそうすることにより、力を使わないでそのようなバグを作り込まないようにするためのソフトウェア設計の術(ワザ)

を発見することを目的としたものである。

2. 2 深層的バグ分析の方法

(1) 分析の対象はなにか

一つ一つのバグが分析の対象である。バグの集合を対象としてそれを統計的に分析するのではない。

(2) 分析の材料はなにか

ソフトウェア設計者のバグ作り込みの過程を分析の材料とする。そのために、担当設計者に対してしつこくインタビューする。そしてそれにより、そのバグを作り込んだときの思考過程を再現する。

(3) どのようにして分析するか

バグを作り込んだ過程をよく見ながら、「なぜ間違えたのだろうか?」、「なぜ間違え易かったのだろうか?」、「自分も同じ間違いをするだろうか?」、「自分なら間違えないとすれば、それはどうしてだろうか?」などと自問自答しながら分析をすすめる。そしてそのときに、「なにかガンバリ過ぎたはずだ」、「なにか変に考え過ぎたはずだ」、「なにか不利な条件があったはずだ」といった仮説を脳裏においておく。人間の能力（たとえば注意力や論理能力など）には期待をかけないようにする。

(4) いつまで分析するか

「こうやっていればこのバグは作り込まなかった。しかもそれは、特に努力しなくても容易に実行できる、むしろ実行できない方がおかしい」と思えるほどの、ソフトウェア設計上の術（ワザ）が見つかるまで分析する。「こうやっていればこのバグは作り込まなかった」というなんらかの対策が見つかって、それを実行するために努力や労力や能力を要するものであれば、それは、われわれのいう術（ワザ）ではない。

2. 3 深層的バグ分析の実施例

深層的バグ分析の実例を図-1および図-2に示す。

2. 4 深層的バグ分析の結果

われわれ自身が過去に作り込んだバグの中からランダムに70件のバグを選び出し、それらについて深層的バグ分析を実施した。なんといっても、2.3で実例を示したような1件1件のバグ分析がその最大の成果である。ちょっと意外なところに根本原因が見つけ出されていることに気がつくられるだろうと思う。

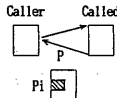
表-1に、われわれの深層的バグ分析の結果見つけ出された根本原因の分類を示す。これから、次のことがいえる。

① ほとんどのバグ（われわれの実施例の場合は約95%）について、深層的バグ分析が可能であった。そしてそれにより、力を使わないでバグを作り込まないようにするためのソフトウェア設計術、作業のやりかたの基本、および、コンパイラの改良すべき点が見つかった。

② ソフトウェア開発者がバグを作り込む原因は、論理的誤りや注意不足にあるのではなく、むしろ、変に深く考え過ぎていること、いいかえれば、かれらのもっているはずの健全な形式感覚や常識感覚をソフト開発時には十分働かせていないことにある。

〔バグの内容〕

呼出しプログラム (Caller) と被呼出しプログラム (Called) があり、両者間の呼出しインタフェース用パラメタリストは P である。Caller が P 内の P1 フィールドにパラメタ値を設定しなかったために、Called は、Called の期待した動作をしなかった。いまの場合 Caller は、P1=X'00' を設定しておかなければならなかった。



〔バグの分析〕

- ・このバグは、障害修正時に作り込んだバグであった。
- ・この修正担当者は、完全を期すために、全く同じ呼出し処理をしている既存のコーディングをそのままマネてコーディングした。ところが、その既存部分には P1=X'00' の命令がなかった。したがって、P1=X'00' の命令はなかった。
- ・それは、既存部分はなぜうまく動作しているか。それは、既存部分では、パラメタリスト P の領域をバッファオーバーから取り出した直後にそれをパラメタリストとして使用しているためであった。(というのは、このプログラム系では、バッファオーバーから取り出した直後のバッファは、それ全体が 0 クリアされるので、明に値を設定しなくても P1=X'00' となるから。)
- ・修正担当者は、実績のあるコーディングをマネたという意味で基本に忠実であった。
- ・お手本になった既存部分のコーディングがよくない。
- ・パラメタリストへのパラメタ値の設定は (たとえその値が 0 でも) 明にそれを行うべし。これは基本である。後から、その部分だけに着目しお手本にする人もいる。
- ・プログラムは初所完結的であるべし。

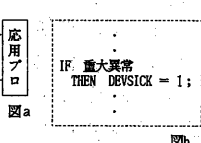
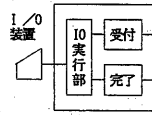
図-1 バグ分析例-1

表-1 われわれの作り込んだバグの根本原因の分類

根本原因	<件数小計>	件数 (比率)
A. 設計・作成上の誤った概念や習慣に囚われている。		
A.1 形式 (一貫性、対称性、階層性) を整えることを第一優先に考えればよいのに、何やら深く考えてしまっている。あたかも、深く考えないとプログラムを作っている気がしないというかのように。	<15>	
A.2 標準リネージ規約に沿えばいいのに、何やら深く考えて、そうしていない。	<10>	
A.3 単純性を志向してさえいれば容易にそうできたのに、なぜかゴチャゴチャと複雑なログクにしている。	<5>	
A.4 明瞭でないログクなどに早分してしまっている。	<5>	
A.5 高級言語流のコーディングをしていない。アセンブラの発想のコーディングをしている。	<4>	
A.6 パラメタ値 (特に U) を明に設定していない。	<3>	
A.7 DEVSIC 系の処理は多量に行って構わないのに、わざわざ多量に行わないように几帳面に削削している。	<2>	
A.8 新しいプログラムをマネれば一番確実でかつ楽でもあるのにそうしていない。	<2>	
A.9 何事も制限や限界があるのが当然で、機械的に制限や限界を設定していればよかったのに、それをためらってしまった。	<2>	
A.10 必然性のないところや曖昧なところは、ムズミに作っておけば安全なのに、キチンと作ろうとしている。	<1>	
B. 作業のやりかたの簡単な基本を守っていない。		
B.1 小規模の修正でも機械的にグループレビューにかけていねばよかったのに、それをやめてしまっている。	<6>	
B.2 一人一人の経験をきちんと行っていない (ex. 文書で確認していない、確認すべきところをしていないなど)。	<5>	
C. コンパイラのチェックが慣性的過ぎる (開発者にとっては甘過ぎる)。	6	(9%)
D. 人間がプログラムを作る限り、完全には防ぎ得ない。	3	(4%)
E. 分析できない (バグの作り直し過程を再現できない)。	1	(1%)

〔バグの内容〕

I/O制御プログラムのバグ。この I/O制御プログラムの構成は下図 a のようになっている。I/O処理が完了すると、I/O実行部はその I/Oの完了状況 (正常完了または異常完了など) を記憶し、その後制御を受ける I/O完了通知部の情報をもとに、応用プログラムに I/O完了状況を通知する。そして異常完了の場合には、「異常」をさらに「一般異常」と「重大異常」とに分け、それぞれ異なった通知をしなければならない。ところが、I/O実行部では下図 b のようにコーディングしてあった。DEVSIC は重大異常を要するフラグで、I/O装置対応に存在する DEV という制御表の中にある。このフラグをどこでもリセットしていなかったために、ある I/O装置で一度重大異常が発生すると、その後は、一般異常が発生しても必ず重大異常として通知されてしまう。



〔バグの分析〕

- ・I/O処理を開始するときこのフラグ (DEVSIC フラグ) を初期リセットすべきであったのだろうか。はたまた、I/O完了通知時にこのフラグをリセットすべきであったのだろうか。どちらを行ってもこのバグは作り込まなかっただろう。
- ・でもそれは結果論。いずれかでも、当然のように行うことができるためには、DEV の中の諸フィールドが恒常的なフィールド (たとえば装置アドレスフィールド) と、一つ一つの I/Oに依存して変動するフィールドとに整理と分けられている (必ずしも物理的にではなく、少なくともプログラムの頭の中だけでも) が前提となる。しかし、それは現実的に見て、過度な要求と推察される。
- ・答は簡単であった。単純な定石を守っていない。I/O実行部で、

```
IF 重大異常 THEN DEVSIC = 1;
ELSE DEVSIC = 0;
```

図 c

とやっておくだけでよかった。DEVSIC = 1 は重大異常発生を意味するのだから、重大異常でなかったときは、DEVSIC=0 とするのは考えてみれば当然である。

・「E.I.S.E」を定数せよ。・「0」を明に設定せよ。

・対称感覚と 0 の効用。

図-2 バグ分析例-2

表-2 7つの設計原理と深層的バグ分析結果との対応

① 単純原理	分析例-1が代表的な例である。表-1では、A.2、A.3、A.5およびA.6などがこの原理に対応する。70件のバグのうち22件がこの原理から説明できた。
② 同型原理	表-1では、A.1の一部、A.2およびA.8などがこの原理に対応する。70件のバグのうち27件がこの原理から説明できた。
③ 対称原理	分析例-2が代表的な例である。表-1では、A.1の一部などがこの原理に対応する。70件のバグのうち6件がこの原理から説明できた。
④ 階層原理	表-1では、A.1の一部などがこの原理に対応する。70件のバグのうち10件がこの原理から説明できた。
⑤ 線型原理	表-1では、A.7などがこの原理に対応する。70件のバグのうち2件がこの原理から説明できた。
⑥ 明確原理	表-1では、A.4およびA.9などがこの原理に対応する。70件のバグのうち7件がこの原理から説明できた。
⑦ 安全原理	表-1では、A.10などがこの原理に対応する。70件のバグのうち2件がこの原理から説明できた。

3. ソフトウェア設計の原理

本章では、バグを作り込まないための7つの設計原理を仮説として提示する。これらの設計原理は、深層的バグ分析の過程でつねに意識していたバグゼロ仮説でもあった。

3.1 7つの設計原理

バグの実態を見ればよくわかるが、バグを作り込む原因の大半は、ソフト開発者の論理的な過ちやソフトウェアについての無知やテスト不足などにあるのではなく、むしろ、一般成人ならだれもが持っているはずの、形式感覚や常識感覚を、ソフトウェアの設計の際には十分働かせていないことにある。逆に、おもしろいことに、これらの感覚を働かせることには、無意識のうちにバグを予防する効果がある。ここで示す7つの設計原理は、これらの形式感覚や常識感覚をもう少し具体化し定式化したものである。

ソフトウェアを設計することは、なにかの理論を構築することに似ている。殊に大規模の制御プログラムの設計についてはそうである。そういう観点から、下の7つの設計原理を眺めてみると、たとえばこの中の、同型原理と対称原理と階層原理は、既存の

いくつかの美しく堅固な理論の中にも見つけ出すことができる(たとえば、ニュートンの運動法則や形式論理の法則や群の公理など)。また、単純原理は、理論構築の上での基本原理である「オッカムの剃刀」にほぼ対応するし、線型原理は、現代の物理学理論を、あまねく貫いているようにみえる。その他、明証原理はデカルトの精神であるし、安全原理は、明証原理の裏返しである。

3. 2 7つの設計原理の現実的妥当性

いま提示した7つの設計原理は、わたくしの過去のソフトウェア設計の経験から、しだいにこのような形に凝集してきたものである。そしてこれらの7つの設計原理は、先の深層的バグ分析を通じてさらに例証され強化されていった。これらの点から、その現実的妥当性は、きわめて高いといえる。

表-2に、これら7つの設計原理が、深層的バグ分析の結果とどのように対応しているかを示す。

4. ソフトウェア設計の論理

第2章で、バグの大半は、われわれが、われわれ自身もっている健全な形式感覚や常識感覚を不当に軽んじているために作り込んでしまっていることを示した。さらに第3章では、それらの形式感覚や常識感覚を7つの設計原理の形に定式化した。本章では、この7つの設計原理にもとづくソフトウェア設計の論理として、感覚の論理を提案する。

考えてみれば神でさえ、この世界を秩序あるものとして創造するとき、論理(すなわち、形式論理や記号論理のような伝統的論理)だけを使ったのではなかった。物理法則をはじめとするいくつかの制約をつけられた。ましてやわれわれ

- | | |
|--------|---|
| ① 単純原理 | : 多よりは一がよいとする原理。
たとえば、「全体的な関連性を意識しながら」プログラムを作ったりせず、局所完全なプログラムで満足するようになり、フィールドの2重定義のような高級なテクニックは極力使わないようにしたり等々、単純に、素人っぽくプログラミングすること。 |
| ② 同型原理 | : 形にこだわるという原理。意味には無頓着であること。
たとえば、同じことは同じようにやることにこだわる。また例外は設けないように固く決意したりすること。一様性を重んじることもこの原理の一部である。 |
| ③ 対称原理 | : 形の対称性にこだわるという原理。
たとえば、上があれば下、右があれば左、アクティブがあればインアクティブというような対称性にこだわる。あるいは、この対称感覚に身を委ねて設計してみるということ。 |
| ④ 階層原理 | : 形の階層的美しさにこだわるという原理。
たとえば、ものごとの主従関係、前後関係、本末関係などの階層関係をつねに意識し、あるべき姿を求めるということ。構造化プログラミングの考えかたもこの原理の一部である。この原理はまた関係性の原理といってもよい。 |
| ⑤ 線型原理 | : 形は直線だけによって描かれており、さらに矩形であるのがよいとする原理。
たとえば、ある機能は、いくつかの機能の重ね合わせによって実現される(線型結合的である)のがよいとすること。あるいは、それぞれの機能は、いくつかの有限個の命題によって規定されるものでなくてはならないとしたりすること(機能の形が曲線でかたづけられている場合には、有限個の命題でその機能を規定することはできないであろう)。その他、RESET系の処理は多重に行ってもかまわないとするのも、この原理の一部である(多重に行うことを避けようとするれば、条件判定が必要となり、制御の直線性が乱れる)。 |
| ⑥ 明証原理 | : ロジックの明証性にこだわるという原理。
たとえば、すこしでも不透明なロジックは証明しておくように努めること。どうしても証明できないときや証明しにくいときには、そのロジックあるいはそれが依っている基本方式を捨てることを覚悟すること。 |
| ⑦ 安全原理 | : 必然性を意識するという原理。
たとえば、必然性のないところや曖昧なところは、ちょっとルーズに、安全サイドで設計しておくというようなこと。 |

図-3 7つの設計原理

人間が、ソフトウェアというあらたな世界を秩序正しく創るとき、伝統的論理一本槍でそれがかなうわけがない。しかし、まさにそうしようとしているのが現在のソフトウェア開発者である。かれらは、伝統的論理にくわえて設計の論理という味方を得ないことにはソフトウェアを自らの知的制御下におくことができないということ、ならびに、この味方の助けを求めることが決してアンフェアなことなのではなく、むしろそうしない方が無責任なことであることに未だ気がついていないように見える。

4. 1 感覚の論理

感覚の論理は、伝統的論理と違う一種の準論理である。感覚の論理は、形式感覚や常識感覚に依拠し、日常語で推論する設計の論理である。対比的に言えば、伝統的論理が、正しく考えるための論理であるのに対し、感覚の論理は、美しく考えるための論理である。そして一般に、設計というものは、美に関するものであり、美しさは正しさを包含する。

4. 2 感覚の論理の推論法

(1) 何を正当性の根拠とするか

7つの設計原理に適合していることを、正しさの根拠とする。

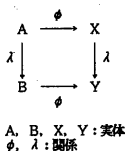
(2) 推論するとき使う言葉はなにか

形式感覚と日常語とを使う。計算機の世界の用語や概念は極力使わないようにする。この制約は、感覚の論理にとって本質的である。こうすることには、われわれの慣れ親しんだ、そして、知恵の結晶であろう認識の道具だけを使い、まだ日の浅い、われわれ自身未だ十分咀嚼し、身につけていない道具に感わされることを避けるという意味がある。

(3) どのように推論するか

形式感覚と日常語とを用いながら、7つの設計原理に適合しているか否かと推論していく。また、通常は厳密でないとして消極的な使われかたしかされないアナロジーは、感覚の論理では積極的な使われかたをする。アナロジーは常識の世界では立派な推論であるからである。

ここでいうアナロジーは次の図-4のような図式で定式化できる。



A, B, X, Y: 実体
φ, λ: 関係

この図式は、たとえば次のように使う。いま地図の設計をしようとしているとする。海に対して陸は対称的な概念といってもよいだろう。(したがって左の図式にあてはめると、Aは海、Bは陸、そしてλは、〜に対して対称的という関係、ということになる。)一方、青に対して赤は、色として対称的であるといってもよいだろう(したがって右と左図式のXとYとしてそれぞれ青と赤をあてはめ)。ここで、青が、海を表す色としてふさわしいとすれば、(したがってφは、〜に対してふさわしい色という関係を表すことになる)赤も、陸を表す色としてふさわしいと推論する。(この結論は、直感的に真といえる。海も陸も青い地図は、伝統論理的には偽ではないかもしれないが、設計的には偽であろう。)

図-4 アナロジー図式

【設計の内容】
レビューをしているときに次図のようなコーディング箇所に出会った。

```
IF HFRDATA = '02'X THEN
  NLSIDTSA = '25'X ;
ELSE
DO ;
  NLSIDTSA = '29'X ;
  VBCSKVSM = OFF ;
  NLSYSRSP = OFF ;
  (さらに 5行)
END ;
```

図a

【設計の証明】
この部分は、THEN節の処理とELSE節の処理とが、端的に形として非対称である。つまり、それぞれの最初の1行だけは同じ形であるが、それ以外はTHEN節が1行なのに対してELSE節は10行近くもある。これは対称原理に反する。なにかよからぬことがありそうなので、さらに細かくレビューが必要である。もし、問題がなくても、適当な機会に、対称形に書き直しておくべきである。(本例の場合、実際にはバグであった。)

図-5 適用例-1

4. 3 感覚の論理のソフトウェア設計への適用例

図-5および図-6に適用例を示す。

4. 4 感覚の論理の有効性

感覚の論理には、以下に示す有効性が期待できる。

① 感覚の論理に従って設計することにより、ソフトウェア設計者は、無意識のうちに、高品質の（すなわち、正しく美しい）ソフトウェアを設計することができる。

② 感覚の論理を使用することにより、ソフトウェア設計者は、一段一段証明しながら設計をすすめること（いわば証明的設計法）が可能となる。

③ 難しくない常識的な論理であるので、少しの訓練によって、使えるようになる。

④ 感覚の論理を共通の設計論理として用いることにより、広範囲のソフトウェア設計者が、同じ設計問題について議論することが可能となる。

⑤ 伝統的論理と異なり、より全体的で、高速な論理であるので、大規模のソフトウェアの正当性の証明への道を開く。

【設計の内容】
同時に複数端末へのデータ転送要求を処理しなければならない制御プログラムがある。この制御プログラムは、一つ一つの端末への円滑なデータ転送という要件も満足させたいし、複数端末間のデータ転送の均等性という要件も満足させたい。しかし、ある制約があり、それらのうち一方は満足させられないとすれば、どちらを採るか。

【設計の証明】
一つ一つの端末へのデータ転送の方を採る。なぜなら、多より一の方がより基本的であるから（階層原理）。

図-6 適用例-2

5. むすび

深層的バグ分析と7つの設計原理と感覚の論理とを提案した。深層的バグ分析については、これ自体、バグの根本原因を発見し、ソフトウェア開発者に対して直接的な示唆を与えるバグ分析法として、非常に有効な手法であると確信する。そしてこれは、ソフトウェア設計の原理の妥当性をテストするための方法ともなる。7つの設計原理については、わたくし自身は、これらはほぼ自明な原理であろうと考えている。おそらく、今後さらにバグを分析しても、それらの根本原因の多くは、この7つの設計原理のいずれかに抵触したものであるだろう。むしろ問題は、これらの設計原理を、ソフトウェアの開発にどう活用できるかにあると思う。これは、わたくしのいうソフトウェア設計の論理の問題となる。さて、そのソフトウェア設計の論理に関しては、感覚の論理を提案した。これについては、残念ながらまだまだ実践例が乏しい。しかし、そのアイデアのもつ可能性は豊かであると思っている。ソフトウェア危機なるものがもし存在するとすれば、感覚の論理のような、伝統的論理を超えた設計の論理の確立が、そこからわれわれを救いだす正統的かつ唯一の方法のようにみえる。今後さらに実践を続けたい。

6. 謝辞

本稿をまとめるよう鼓舞して下さった当社の久保ソフトウェア技術部長、日頃から御指導いただいている山地第三開発部長、ならびにその生き生きとした発想に学ぶところの多いソフトウェア技術部の君島浩氏に深く感謝の意を表したい。ただし、本稿における議論の粗雑さや誤りは、すべて筆者の責任下にある。