ソフトウェア工学 118−21
（1998. 3. 11）

クリーンルーム手法の味見

谷津弘一　染谷誠

日本ユニシス株式会社ソフトウェア開発部

限りなく無欠陥に近い高品質のソフトウェアを開発する方法として、クリーンルーム手法が注目されているが、その仕様の記述の仕方には問題がある。拙文では、クリーンルーム手法の仕様記述の問題点を明らかにし、その改善策について議論する。

A Taste of Cleanroom Software Engneering

Hirokazu Yatsu　Makoto Someya

Systems Engineering Division, Nihon Unisys, Ltd.

Cleanroom is a promissing managerial and technical process for the development of high quality software with zero defects. However, Cleanroom has some problem in describing functional specification of software to be developed. In this paper, the problem is pointed out and its solution is described.

1 はじめに

クリーンルーム手法は、欠陥の極めて少ない高品質のソフトウェアを開発するために IBM 社の H.Mills 博士により考案されたものであり、ソフトウェア開発者から高い注目を集めている手法である。情報処理学会ソフトウェア工学研究会でもクリーンルーム手法ワーキンググループを発足させ、1996 年度の始めからクリーンルーム手法の調査研究活動を行なってきた。

筆者のうちの一人 (染谷) は発足時からグループメンバーとして、もう一人 (谷津) は 1997 年 4 月からオブザーバとして、ワーキンググループに参加している。拙文は、クリーンルーム手法に対する筆者らの個人的な感想を綴ったものである。

拙文の構成は以下の通りである。まず第 2 節にてクリーンルーム手法の「レシピ」を簡単に紹介した後で、第 3 節にて、拙文の目的であるクリーンルーム手法の「味」について述べる。そして、最後に第 4 節にてクリーンルーム手法の課題について若干の議論を行なう。

クリーンルーム手法の適用範囲は、顧客の要求分析から品質評価に至るまで、ソフトウェアライフサイクル全般にわたる。しかし、我々の興味はソフトウェアの仕様記述にある。拙文で述べさせていただく意見はクリーンルーム手法の仕様記述に関する部分に対するものであることを、ここでお断わりしておく。

2 クリーンルーム手法

本節では、クリーンルーム手法の「レシピ」として、その手順と体制について簡単に述べ、仕様記述～コード開発～検証を支える要素技術である、ボックス構造分析について説明する。ここの説明は、[6, 7] による。

2.1 手順

クリーンルーム手法に基づく開発は図 1のようにして進められる。

1. まず最初に、顧客の要求分析を行なう (要求分析)。
2. 次に、顧客の要求を満足するシステムの機能の初期仕様を作成する (機能仕様作成)。この仕様では、システムは、与えられた「刺激」(入力等のユーザによる働きかけ) の履歴から応答を返す関数として表現される。ここで定められる機能の仕様をブラックボックスと呼ぶ。
   それと並行して、どのようなユーザがどのような環境でどの程度の頻度でシステムを使用するかを想定する (使用状況の仕様化)。
3. 作成した初期仕様と想定した使用状況を基にして、段階的な開発計画を立てる。その際、段階を経る毎にシステムの機能が累積的、拡充的に開発されるよう、各段階で開発される機能を決定する (インクリメント開発計画立案)。各段階で開発される機能をインクリメントと呼ぶ。
4. 定められた順序に従って、インクリメントを以下の手順に従って開発する。
   (a) インクリメントの仕様に対してモデル指向アプローチに基づく段階的詳細化を行ない、最終的なコードを作成する。この段階では作業者はテストを一切行なわず、作業者間の厳密なレビューを通して段階的詳細化に義務づけられる証明を行ない、作成したコードの信頼性を高める (デザイン・検証)。段階的詳細化して得られる中間的な仕様やコードを、ステートボックス、クリアボックスと呼ぶ。
   (b) デザイン・検証と並行して、想定されたシステムの使用状況からユーザがシステムを使用する詳細なシナリオ (イベントの推移とその生起確率で表される) を作成し、さらに、シナリオに基づき統計的なテストケースを作成する (テストケース生成)。
   (c) 作成された統計的なテストケースを用いて、開発されたコードのテストを行なう (統計的テスト)。テスト結果を評価して、問題点が検出されれば、適当な段階に適宜フィードバックされる。

2.2 体制

クリーンルーム手法に基づくソフトウェア開発は、プロジェクト管理者を除いて、原則として

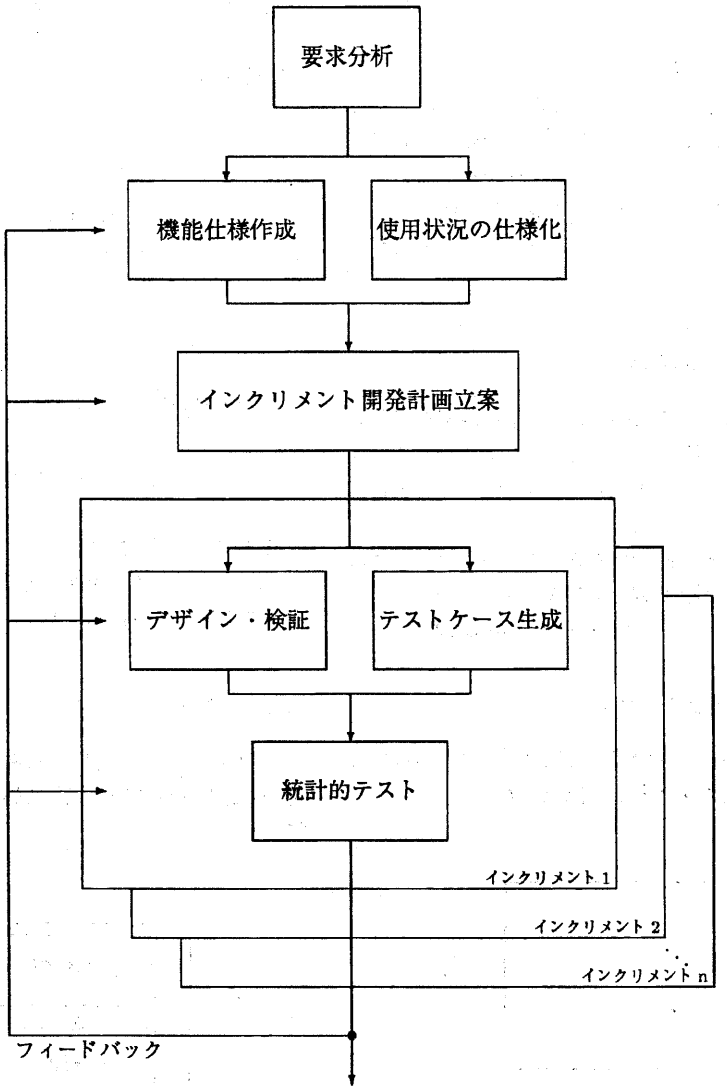


図 1: クリーンルーム手法の手順

- 機能仕様作成チーム
- 開発チーム
- 品質保証チーム

で行なわれる[1]。

機能仕様作成チームは、要求分析、機能仕様作成、使用状況の仕様化、インクリメント開発計画立案に携わる。開発チームはデザイン・検証を行ない、品質保証チームはテストケース生成と統計的テストを行なうが、この2チームはインクリメント開発計画立案にも携わる。

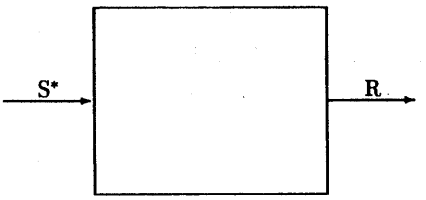
2.3 ボックス構造分析

クリーンルーム手法の「レシピ」の説明を終える前に、手順の説明で出てきたブラックボックス、ステートボックス、クリアボックスについて、触れておこう。

機能仕様記述段階で作成されたブラックボックスは、デザイン・検証段階においてステートボックスを経てクリアボックスへと段階的に詳細化されていく。

2.3.1 ブラックボックス

ブラックボックスでは、対象は「刺激」の履歴を得て応答を返す関数として記述される。



対象:「刺激」の履歴 ⟶ 応答

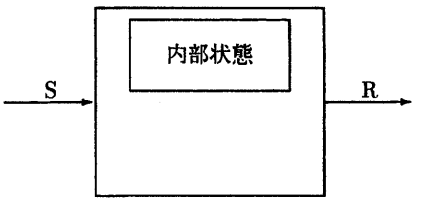
図 2: ブラックボックス

内部状態を具体的に記述しない、性質指向の記述である。

[1]機能仕様作成とデザイン・検証を別々のチームが行なうやり方 [6, 7] と一つのチームが行なうやり方 [8] がある。筆者は分ける方が自然だと考えたので、拙文ではこのような説明をしている。

2.3.2 ステートボックス

ブラックボックスでは対象の内部状態の記述をしなかったが、ステートボックスでは対象の内部状態を記述する。そして、各「刺激」毎に外部への応答だけではなく、内部状態がどのように遷移するかをも記述する。



対象:「刺激」×内部状態 ⟶ 応答×内部状態

図 3: ステートボックス

ここでは、「刺激」はカレントのものだけになる。ブラックボックスは性質指向であったが、ステートボックスはモデル指向である。

2.3.3 クリアボックス

クリアボックスでは、「刺激」と遷移前の内部状態から応答を作り内部状態を遷移させる手続きを記述する。

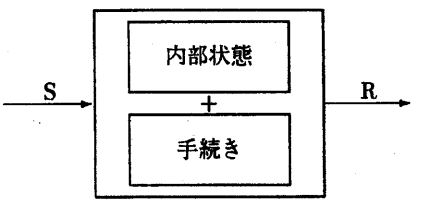


図 4: クリアボックス

クリアボックスで記述される手続きの中に、ブラックボックスが現われることもある。その場合には、改めてブラックボックスをステートボックスからクリアボックスへと詳細化させることになる。最後に、ブラックボックスのない手続きの記述を得て、コードが開発されたことになる。

3 クリーンルーム手法の味

クリーンルーム手法には、

- 段階的詳細化の証明を、形式的に行なうことをせず、レビューを通して行なう。
- デザイン・検証の際に、開発チームはテストを一切行なわない。

などの点で、筆者らは大いに共感を覚える。

まず、段階的詳細化の正当性の証明が形式的に行なわれなければならないとは、筆者には思えない。というのも、形式的な証明というのは、まことに手間がかかるものだからである。たとえ証明の支援系があったにせよ。

証明に必要な理屈をわきまえている人が段階的詳細化が正しく行なわれていることをレビューを通して納得することは、その証明を形式的に行なうことと同等の効果がある。しかも、レビューの手間は形式的な証明を行なうよりも少ないだろうし、レビューの参加者のシステムに対する理解が深まることが期待できる。

また、実際に検証を行なう開発チームは不安を覚えるだろうが、テストを一切行なわないことにより必然的にレビューが厳密になる。生半可にテストを行なうよりも、信頼性を含めた品質の高いコードを開発できる。

しかし、クリーンルーム手法に対して不満がないわけではない。クリーンルーム手法ではシステムの機能の仕様をまず最初に性質指向で記述するが、内部状態を意識することなく性質指向でシステムの振る舞いを表現するのは非常に難しい。

以下は、Zの記述例としてよく出てくる誕生日帳[5]をブラックボックスで表現したものである[4]。ちなみに、誕生日帳に対する要求は以下の通りである。

- 誕生日帳には名前と誕生日を登録できる。
- 誕生日帳から名前と誕生日を削除できる。
- 名前を与えるとその人の誕生日がわかる。
- 日付を与えるとその日に生まれた人の名前がすべてわかる。

define BB Birthday Reminder
stimulus
InitSystem
Add(name: NAME, date: DATE)
Delete(name: NAME)
FindBirthday(name: NAME)
Remind(today: DATE)
response
⟨add_confirm⟩ : MESSAGE
⟨already_known⟩ : MESSAGE
⟨delete_confirm⟩ : MESSAGE
date : DATE
⟨not_known⟩ : MESSAGE
reminders : **P** NAME
transition
1.InitSystem
V1: true
R1: null response(*stim_hlist* = ⟨⟩)
2.Add(name: NAME, date: DATE)
V1: Name not known to system
R1: ⟨add_confirm⟩
V2: Name known to system
R2: ⟨already_known⟩
3.Delete(name: NAME)
V1: Name known to system
R1: ⟨delete_confirm⟩
V1: Name not known to system
R2: ⟨not_known⟩
4.FindBirthday(name: NAME)
V1: Name known to system
R1: date
a birth date is displayed which is the date entered for the name
V1: Name not known to system
R2: ⟨not_known⟩
5.Remind(today: DATE)
V1: true
R1: reminders
Display every name known to system such that the birthday corresponding to the name is equal to today

ここで、**BB** はブラックボックスを意味する。**stimulus** 部と **response** 部では、システムに与えられる「刺激」とシステムが返す応答がそれぞれ列

挙されている。**transition** 部で「刺激」の履歴とそれに対する応答が対応付けられる。「name known to system」とは「その名前がシステムに既に登録されており、その後削除されていない」ことを、「name not known to system」とは「その名前がシステムに登録されたことがないか、あるいは、既に削除されている」ことを、それぞれ意味している。

このような簡単な記述にも、つつける重箱の隅はある。例えば、既に登録されている名前で異なる誕生日を登録するとエラーメッセージらしきものを返すが、実際にはどうなるのだろうか。その誕生日は無効になるのか、それとも、上書きされるのか。FindBirthday でその名前で登録されている誕生日を検索すると、どちらの日付が返ってくるのだろうか。この記述だけでは判断がつかない。また、InitSystem という「刺激」がいつでもシステムに与えられ得ることにも多少不満があるし、InitSystem が行なわれる度に「刺激」の履歴が空なるのは (常識的には納得できるが) いただけない。空の履歴の場合と同じ振る舞いをする、というだけで十分ではないだろうか。何よりもここで履歴を内部状態のごとく扱っているのが気に入らない。

上で挙げたことを踏まえたブラックボックスの記述を Z で行なうと次のようになる。ここで、Z を持ち出したのは、へたに自然言語で記述するよりも正確だからである。

まず、基本的な型として、Name(名前)、Date(日付)、Message(メッセージ)、そして BirthdayBook(誕生日帳) を考える。後で作られるステートボックスの内部状態との対応を考えると、「刺激」の履歴を考えるよりも誕生日帳の型を挙げておく方が都合がよい。

$$[Name, Date, Message]$$
$$[BirthdayBook]$$

システムから出されるメッセージは次の 4 つである。

$$add\_done : Message$$
$$delete\_done : Message$$
$$not\_known : Message$$
$$already\_known : Message$$

システムの応答には、メッセージ、日付、名前の集合の 3 種類がある。

$$\begin{aligned} Response ::= & \; message\langle\!\langle Message \rangle\!\rangle \\ | & \; date\langle\!\langle Date \rangle\!\rangle \\ | & \; names\langle\!\langle \mathbb{P}\, Name \rangle\!\rangle \end{aligned}$$

システムに対する「刺激」は、add(登録)、delete(削除)、find(名前による検索)、remind(日付による検索) の 4 つである。init は「刺激」ではなく、誕生日帳の初期値である。

$$init : BirthdayBook$$
$$add : NameDateBirthdayBook \longrightarrow BirthdayBook$$
$$delete : NameBirthdayBook \longrightarrow BirthdayBook$$
$$find : NameBirthdayBook \longrightarrow BirthdayBook$$
$$remind : DateBirthdayBook \longrightarrow BirthdayBook$$

誕生日帳のブラックボックスを BB とすると、BB は図 5のように書ける。

改めて言うが、記述対象の振る舞いをその内部状態を意識せずに記述するのは難しい。誕生日帳のような単純なはずの例でもかなり複雑な記述になる。ましてや、現実のシステムについては言わずもがなであろう。

機能仕様の記述を疎かにしておくと、その後のインクリメント開発計画やデザイン・検証に大きな悪影響を及ぼす。機能仕様の作成にはもう一工夫欲しいところである。

4 議論

本節では、前節で問題にした、機能仕様の記述について議論する。

ソフトウェア開発を「数学する」ことのアナロジーで考えるべきだ、という指摘がある [1]。特殊から一般へ。これは「数学する」際の基本的な姿勢である (最も、これは数学に限らず、学問一般に共通の姿勢であろうが)。つまり、いきなり一般的な問題を相手にするのではなく、具体的な問題から始めて、それを次第に一般化していくわけである。

こういうとこじつけの誹りを免れないだろうが、クリーンルーム手法では、「特殊」はステートボックス (i.e., モデル指向記述) に、「一般」はブラックボックス (i.e., 性質指向記述) にあたるのではないだろうか。

$$BB : BirthdayBook \longrightarrow Response$$

$$\begin{aligned}
&\forall p, p' : Name;\ d, d' : Date \mid bb : BirthdayBook \bullet \\
&\quad BB(find(p, bb)) = message(not_known) \Rightarrow \\
&\qquad BB(add(p, d, bb)) = message(add_done) \wedge \\
&\quad BB(find(p, bb)) = date(d') \Rightarrow \\
&\qquad BB(add(p, d, bb)) = message(already_known) \wedge \\
&\quad BB(find(p, bb)) = message(not_known) \Rightarrow \\
&\qquad BB(delete(p, bb)) = message(not_known) \wedge \\
&\quad BB(find(p, bb)) = date(d') \Rightarrow \\
&\qquad BB(delete(p, bb)) = message(delete_done) \wedge \\
&\quad BB(find(p, init)) = message(not_known) \wedge \\
&\quad BB(add(p, d, bb)) = message(add_done) \wedge p = p' \Rightarrow \\
&\qquad BB(find(p', add(p, d, bb))) = date(d) \wedge \\
&\quad BB(add(p, d, bb)) = message(add_done) \wedge p \neq p' \Rightarrow \\
&\qquad BB(find(p', add(p, d, bb))) = BB(find(p', bb)) \wedge \\
&\quad BB(add(p, d, bb)) = message(already_known) \Rightarrow \\
&\qquad BB(find(p', add(p, d, bb))) = BB(find(p', bb)) \wedge \\
&\quad BB(delete(p, bb)) = message(delete_done) \wedge p = p' \Rightarrow \\
&\qquad BB(find(p', delete(p, bb))) = message(not_known) \wedge \\
&\quad BB(delete(p, bb)) = message(delete_done) \wedge p \neq p' \Rightarrow \\
&\qquad BB(find(p', delete(p, bb))) = BB(find(p', bb)) \wedge \\
&\quad BB(delete(p, bb)) = message(not_known) \Rightarrow \\
&\qquad BB(find(p', delete(p, bb))) = BB(find(p', bb)) \wedge \\
&\quad BB(find(p', find(p, bb))) = BB(find(p', bb)) \wedge \\
&\quad BB(find(p, remind(d, bb))) = BB(find(p, bb)) \wedge \\
&\quad BB(remind(d, init)) = names(\varnothing) \wedge \\
&\quad BB(add(p, d, bb)) = message(add_done) \wedge d = d' \Rightarrow \\
&\qquad BB(remind(d', add(p, d, bb))) = names(\{p\} \cup names^{\sim}(BB(remind(d', bb)))) \wedge \\
&\quad BB(add(p, d, bb)) = message(add_done) \wedge d \neq d' \Rightarrow \\
&\qquad BB(remind(d', add(p, d, bb))) = BB(remind(d', bb)) \wedge \\
&\quad BB(add(p, d, bb)) = message(already_known) \Rightarrow \\
&\qquad BB(remind(d', add(p, d, bb))) = BB(remind(d', bb)) \wedge \\
&\quad BB(delete(p, bb)) = message(delete_done) \Rightarrow \\
&\qquad BB(remind(d, delete(p, bb))) = names(names^{\sim}(BB(remind(d, bb))) \setminus \{p\}) \wedge \\
&\quad BB(delete(p, bb)) = message(not_known) \Rightarrow \\
&\qquad BB(remind(d, delete(p, bb))) = BB(remind(d, bb)) \wedge \\
&\quad BB(remind(d, find(p, bb))) = BB(remind(d, bb)) \wedge \\
&\quad BB(remind(d', remind(d, bb))) = BB(remind(d', bb))
\end{aligned}$$

図 5: Z による誕生日帳のブラックボックス記述

最も、ステートボックスの記述に使われる言語はDijkstraの小言語[3]のサブセットのような、ガード付きの単一代入文である。これでは表現力に乏しく[1]、ブラックボックスの記述を得るための出発点にはなり得ない。[4]でも試みられているように、Zによるモデル指向記述をステートボックスの記述に用いたい。

前節で挙げた誕生日帳にしても、「特殊」であるモデル指向のZ記述[5]は具体的でわかりやすい。記述も極めて単純である。そこから筆者が挙げた性質指向(もどき)の記述を導くことは容易い。

顧客は整理された要求を持っているわけではない。顧客から出される要求を理解し整理するためにも、特殊から一般へ、すなわち、モデル指向から性質指向へ。

このようにすれば、機能仕様の作成が終わる頃には、ある程度の細かさでモデル指向の仕様も作成されている。そうなれば共通部品の割り出しも容易に行なえるだろうし、インクリメントの開発計画の立案も容易になる。デザインの途中でブラックボックスが出現することもほとんどなく、コードの開発がスムーズに進む。

5 おわりに

的外れかも知れない意見を述べてきたが(グルメの振りをした味音痴は世の中にごまんといる。許されたい)、筆者らは第3節の冒頭でも明らかにしたようにクリーンルーム手法に大いに共感を覚えている。

集合論に基づく形式仕様言語やDijkstra算[3]といった、四半世紀も前に考案されている形式的方法を現実の開発に(常識的に)使えるようにするための橋渡しの役目を果たしてくれるものと期待している。

最後にちょっとした疑問を一つ。分散処理システムの開発にクリーンルーム手法を適用するにはどうすればよいのであろうか?

謝辞

拙文を書く上で、(株)日立製作所 システム開発研究所の金藤栄孝さんからいろいろとお教えを賜りました。ここに深く感謝いたします。

参考文献

[1] 金藤栄孝, クリーンルーム手法ワーキンググループ第4回会合における講演, 於東工大, 1997年11月17日.

[2] 西橋幹俊, ソフトウェア開発クリーンルーム手法 — その概要と留意点 —, 電子情報通信学会誌, vol.80, No.5, pp 470 - 478, 1997.

[3] Gries, D., *The Science of Programming*, Springer-Verlag, 1981.

[4] Fetzer, D.T. and Poore, J.H., Using Box Structures with the Z notation, Proceedings of the Hawaii International Conference on System Sciences, vol.2, pp 394 - 405, IEEE, 1992.

[5] Spivey, J.M., *The Z Notation: A Reference Manual 2nd ed.*, Prentice-Hall, 1992.

[6] http://www.asset.com/stars/loral/cleanroom/tutorial/

[7] http://www.cs.utk.edu/ trammell/MFCT/process.html

[8] http://www.clearlake.ibm.com/MFG/solutions/cleanrm.html