ソフトウェア工学に於いて
「数学的である事」とは如何なる事か？

― 精密ソフトウェア工学を目指して ―

こんどう　ひでたか
金藤 栄孝

（株）日立製作所　システム開発研究所

ソフトウェアでの「理論」と「実践」との関係

理論計算機科学　：　ソフトウェア工学

≅　？　：　？

余談

私は「(理論) 計算機科学」(“(theoretical) computer science”)
という用語が大嫌いである.

何故なら,「計算機」という製品分野の名称と，本質的には真理探求の学を
表わす「科学」を繋ぐのは，カテゴリ・エラー以外の何物でもないから.
(「船舶工学」とは呼んでも「船舶科学」とは呼ばないでしょう.)

最近，“informatics”(「情報学」？) なる語が使われているが,
この語の語幹の “information” (「情報」) という言葉も
ソフトウェアの基礎分野を表わす目的には範囲が広すぎる.

計算機ソフトウェアの本質は「計算できる事」にこそ有るのだから,
それを基礎付ける科学分野を表わす名称には,

“computationics”(訳語としては「算理学」)

を提案したい.

Hoare & He (1998) の立場

≅　　物理科学の各分野　　：　　従来の各工学分野

しかし,

物理法則は**帰納的**に基礎付けられる. 一方,

計算は**演繹的**理論に基づいている.

この他にも, Hoare & He は,

物理科学に対する**素朴過ぎる「信仰」**

が散見される.

例えば, 現状の物理学は, 彼らがその本で主張している様に,

統一的でも**完全な還元主義的**成功を収めている訳でもない!

例えば,「何故, 飛行機が飛ぶのか?」の原子レベルでの説明, 即ち,

Navier-Stokes 方程式が, 何故, あの形なのかの原子レベルでの説明は,

未だ判っていない.

我々の提案

	理論計算機科学	:	ソフトウェア工学
$\cong$	数理論理学	:	数学
$=$	数学を 分析する 事	:	数学を 行なう 事

即ち,

	ソフトウェアを 基礎付ける 事	:	ソフトウェアを 開発する 事
$\cong$	数学を 基礎付ける 事	:	数学理論を 生み出す 事

言い替えると，**我々のテーゼ**とは：

ソフトウェアは 数学的に 開発されるべきである．

しかしながら，

「数学的である事」とは如何なる事か？

「数学的である事 $(Mathematicalness)$」は，

通常の数学者の視点

に基づいて定義する．

即ち，部外者である我々ではなく，

普通の数学者が「数学だと思う」ものが数学！

形式的仕様は，しばしば

数学的である

と主張される．しかし，

形式的手法は**本当に数学的なのか？**

(前に述べた我々の意味に於いて)

FM は何らかの（数理）論理学 の体系に基づいている

という事を勘案すると，前の質問は，次の様に言い替えても良い．

論理的である事」 $\Longleftrightarrow$ 「数学的である事」？

「論理学的である事」 $\Longleftrightarrow$ 「数学的である事」？

($\Leftarrow$): 自明に**真**；

∵）数学は本質的には**演繹的**科学であるから.

($\Rightarrow$): ？！？これは真とは限らない！？！

論理学者の視点 vs 数学者の視点

論理学者の（数学を分析する時の）視点は，

還元主義的である！

例：

$$\forall x \in G.\, e \cdot x = x,$$

$$\forall x \in G.\, (x)^{-1} \cdot x = e,$$

$$\forall x, y, z \in G.\, x \cdot (y \cdot z) = (x \cdot y) \cdot z.$$

論理学的な視点からは，これらの論理式は，定数と関数記号とを持つ一階の理論の文以上の何物でもない.

数学者にとっては，これらの公理の定める数学的構造は，殆ど，数学として有意で面白いか否かの基準ですらある.

冗談

論理学者とは，帰納法に最も関心を持っている人種である.

数学者とは，帰納法に関心のない（帰納法が出て来たら自分の仕事は終わったと思う）人種である.

形式化の非効率性

例 1 :

X を集合, $\sqsubseteq$ と $\lesssim$ を X 上の半順序, $\simeq$ と $\approx$ を, 各々, それらの半順序から導かれる同値関係とする. この時,

$$\forall x, y \in X.\,(x \sqsubseteq y \supset x \lesssim y)$$
$$\supset \forall x, y \in X.\,(x \simeq y \supset x \approx y)$$

この言明は直感的には自明であるにも拘らず, NK での形式的な証明では, 約 10 ステップを要する.

例２：簡単なプログラムの正当性証明からの抜粋

$$NotYetFound(i, j-1) \wedge a[i,j] = 0$$
$$\wedge\, i \in [1..m] \wedge j \in [1..n]$$
$$\supset (MinimalIndex(i,j) \vee NowhereZero \wedge i > m)$$

where

$$NotYetFound(i,j) \triangleq$$
$$\forall k \in [1..m], l \in [1..n].$$
$$\big(\langle k,l\rangle \sqsubseteq \langle i,j\rangle \supset a[k,l] \neq 0\big)$$

$\sqsubseteq$: the lexicographic order on $\mathbb{N} \times \mathbb{N}$

$$MinimalIndex(i,j) \triangleq$$
$$\forall k \in [1..m], l \in [1..n].$$
$$\Big(\big(a[k,l] = 0 \supset \langle i,j\rangle \sqsubseteq \langle k,l\rangle\big)$$
$$\wedge\, i \in [1..m] \wedge j \in [1..n]\Big)$$
$$NowhereZero \triangleq \forall k \in [1..m], l \in [1..n]., a[k,l] \neq 0$$

意味的に考えると自明にも拘らず，形式的にこれを証明するには約 20 ステップを要する.

教訓

> 人間にとっては極めて**自明な**「証明ステップ」でさえも，形式的証明では**10ステップかそれ以上**を必要とする.

即ち,

形式的証明は数学での証明と比べて，**規模が文字通り桁違いに大きい**

普通の数学者の「生産性」

形式的数学者 (補助概念)

（仮想的なクラスの）数学者で，通常の数学者と同様，**数学を行なう（即ち，新しい数学理論を生み出す）**事を生業とするが，それを**形式的に**行なう点が通常の数学者とは異なる.

注意：

形式的数学者を，既存の数学理論を形式言語で記述された形に翻訳／収集する活動をしている研究者（大抵は，自動定理証明等の研究者）とは**混同しない様に**せよ.

数学者の生み出す**数学的内容に関する彼の生産性は，彼が形式的数学者であると考える**事で測る事ができる.

（即ち，論文や書籍を形式的な言語に翻訳した時の量）

例 1　**Barendregt の “*The Lambda Calculus*”**：

約 600 ページ,

約 30 行 / ページ,

換算尺度 (informal⟶formal) 20

（この本の証明には殆ど自明でないギャップは無いので極めて小さい値）

⇓

この本の内容は約 400,000 formal lines (LOC) に相当すると見積もる事が出来る.

例２．

難しい数学の論文誌に載る長さが 20 ページで 30 行／ページの論文（そいつらの証明は，大抵，我々の様な**素人では到底埋め難い「ギャップ」だらけ**なのであるが）

$$\Downarrow$$

その数学的な内容の推定評価量は，約 $60,000 \sim 600,000$ LOC！

教訓

形式化（特に，**形式的証明**）は，内容に関する生産性を大幅に低下させる！

観察

Software Formalists（形式的な検証の研究者）は，彼ら自身の論文を形式的に書いた事もなければ，彼ら自身の（彼らが研究しているFMの性質に関する）メタ定理を形式的に証明した事もない！

事実

ソフトウェア技術者は，既にハンディキャップを背負っている．即ち，彼らの最終成果物であるプログラムは，完全に形式的な代物でなければならない．

社会的対応関係: ソフトウェア *vs* 数学

或いは，**連中は何で給料を稼いでいるのか？**

ソフトウェア技術者	数学者
プログラム	(新しい定理を含んだ) 新しい数学の理論 (に関する論文)
正当性の証明	定理の証明

⇓

何れの場合も，証明の位置付けは，生産物そのものではなく，

品質保証の手段

注記

ソフトウェアの正しさは，ソフトウェア技術者の主たる目的の一つである（或いは，「あるべきである」）が，

正当性の証明自身の（絶対的な）正しさは，彼らの主要な職務の範囲外である.

テーゼ

プログラムは，再帰的関数を実現する 形式的対象 である.

従って，**プログラムは形式的な枠組の下で開発されるべきである.**

⇓

形式的手法

このテーゼに基づく**最も極端なアプローチ**は：

構成的プログラミング

De Bruijn-Curry-Howard の対応: プログラム *vs* 論理

プログラミング	論理
仕様	命題
プログラム	証明

限界

この視点は，programming-in-smalls のスケールであり，現実のソフトウェア・システムの大規模スケールには対応していない．（純粋な数理論理学的視点では，「補題」の設定の善し悪しの判定が行なえないのと同様）

アンチテーゼ

プログラムは，単に計算機への指令の為だけでなく，人間同士のコミュニケーションの為のものでもある.

従って，**プログラムは楽しく読めるものであるべきだ**.

⇓

Knuth の文芸的プログラミング

人間の活動の中で，プログラミングに最も類似した活動は **文芸作品を書く事**である.

注記

この視点は**エンジニアリングでは無視できない**.

（即ち，保守の問題）

シンテーゼ

プログラムは**形式的な理論体系**に基づいており，同時に計算機だけでなく**人間によっても**読まれねばならない.

しかも，殆どのプログラムは**自明でない規模**である.

故に，**プログラムは，その論理的な特性が良く判っている「良い構造」を用いて設計されねばならない**.

⇓

精密ソフトウェア工学

ソフトウェア開発に最も類似した人間活動は**数学の本を書く事**である.

構造的対応：ソフトウェア *v.s.* 数学

アイデア

構成的プログラミングでのプログラミングと論理との対応関係を，より programming-in-larges のスケールで考えよう.

ソフトウェア開発	数学的活動
基本制御構造	論証の基本ステップ
イディオム	証明テクニック
抽象データ型	数学的概念に関する理論
デザイン・パターン	証明の戦術
アーキテクチャ・スタイル	理論構築の戦略
ソフトウェア・システム	数学理論
ドメイン	数学の分野

基本制御構造 *v.s.* 論証の基本ステップ

この対応は exact である. （*cf.* 構成的プログラミング）

イディオム *v.s.* 証明のテクニック

イディオムとは，複数の**基本制御構造**を特定のパターンで組み合せたものである.

証明テクニックとは，複数の**基本的な論証ステップ**を特定のパターンで組み合わせたものである.
例えば,

Fermat の無限降下法：（初等整数論で良く出て来る）

> **数学的帰納法**と**背理法**とを一定のパターンで組み合わせた論証法.

抽象データ型 *v.s.* 一つの数学的概念に関する理論

抽象データ型とは，（その表現が隠蔽されるべき）共通のデータに作用する**演算の集まり**である.

数学的概念に関する理論とは，その概念の性質を厳密な形で表わしている**命題の集まり**である.

抽象データ型	数学的概念に関する理論
演算の仕様	（概念の性質に関する）命題
抽象データ型の仕様	（概念に関する）理論＝正しい命題の集まり
演算の実現	命題の証明
抽象データ型の実現	理論を構成する命題群の証明の集まり

デザイン・パターン *v.s.* 証明の戦術

デザイン・パターンとは，複数の**抽象データ型（クラス）**とそれらの間の特定の**依存関係（継承）**との組み合わせ方である.

証明の戦術とは，（証明を進める上で必要に応じて導入される補助的な概念の性質を与える）**補題 — その集まりは，補助概念に関する「小さな」理論を構成する** —とそれら補題や補助概念の間の依存関係に関する知識である.

デザイン・パターン	証明の戦術
抽象データ型（クラス）	補助的な理論としての補題の集合
継承	補題群の間の依存関係

アーキテクチャ・スタイル *v.s.* 理論構築の戦略

アーキテクチャ・スタイルとは，ソフトウェア・システムを構築する為のテンプレートであり，**コンポーネントの部分的な仕様，内部インタフェース，外部インタフェース**を一定の形で組み合わせたものである.

理論構築の戦略とは，**基本的定義，理論が示そうとする目標のパターン（主定理されるべき generic な命題），重要な補題群**の有機的な纏まりである．例えば,

λ- 計算や類似の簡約理論／書き換え理論を構築する場合,

合流性（Church-Rosser 風の定理．即ち，これが理論の目標）を示すには，良く知られたものとしては，大別すると 2 通りの方法がある．（その各々が「理論構築の戦略」である）：

(1) Hindley-Rosen のダイヤモンド補題（この補題自体が generic）を用いる方法；

(2) 並列簡約（この概念自体が generic）を用いる方法.

ソフトウェア・システム *v.s.* 数学理論

ソフトウェア・システム	数学理論
システムの外部仕様	主定理として挙げられた命題
コンポーネントの仕様	補題として示された命題
コンポーネントの実現	補題の証明
ライブラリ・コンポーネント	文献から参照した補題
外部仕様の為の抽象データ型	主定理を述べる為の数学的概念
内部インタフェースの為の抽象データ型	補題を述べる為の補助概念

ドメイン *v.s.* 数学の分野

共に，仕事を進めて行く為に必要な基本的な語彙の集合と基本的な考え方を定める.

ソフトウェア工学で何が欠けているのか？

ソフトウェアのマクロ構造（イディオム，デザイン・パターン，アーキテクチャ・スタイル）の論理的に**厳密な性質を分析し，蓄積し，再利用する**事.

形式的手法は主として**記述する事**に関心がある．しかし
性質の分析の方が記述よりも遥かに重要である．

精密ソフトウェア工学を目指して

- ソフトウェアの様々な粒度の階層で繰り返し現れるマクロ構造の性質を分析し,
- それらの構造に関する**ミニ理論**（「数学理論」と同じ意味で）を構築する事.
  この活動は，表現や実現と独立に構造自身を調べるという点で，抽象代数と同じ意味で,

抽象ソフトウェア工学

と呼んでも良い．これこそが，**ソフトウェア開発を直接に支えるソフトウェアの為の学**となる.
（この「抽象ソフトウェア工学」の確立は，アカデミズムの方々のミッション）

その成果に基づく実践としての精密ソフトウェア工学での構造の利用手順は，概ね，

- それらの構造に関して分析から判った論理的な性質を活用して，**それらの構造を，安全に（即ち，バグの入る心配なしに）再利用し，**

- それらの構造を用いて構築されたソフトウェアの正しさを，**個々の構造に対するミニ理論に基づいて効率良く**証明する.

これが，将来あるべきソフトウェア開発の姿である！