〈論　　文〉

コンピュータ・ソフトウェア史の研究課題（1）

――科学，工学，信頼性の側面から見たソフトウェア――

吉　田　晴　代

はじめに

2000年4月にドイツのPaderbornで，「Mapping the History of Computing–Software Issues（コンピュータ史研究の現状と課題を探る－ソフトウェア問題を中心に）」というテーマの国際会議が，Heinz Nixdorf Museums Forum, Charles Babbage Institute（以下CBIと略），Paderborn大学Heinz Nixdorf研究所三者の後援により開催された[1]。会議ではソフトウェア史の現状と課題が多方面から議論された。本稿の目的は，議論の概要を筆者の関心に基づき理解した範囲で紹介することにある。

なぜ，今コンピュータソフトウェア史か？会議全体の概要を簡単に説明しておきたい。「現代のラジオやテレビ装置，それにインターネットの10～20％はハードウェアで構成されるが，残りはソフトウェアだ」という言葉[2]を待つまでもなく，我々はコンピュータシステムに囲まれソフトウェアに依存して日常生活を送っている。だが，CBI所長のArthur L. Norbergが開会の辞で述べたように，コンピュータ・ユーザーの大部分で，技術的な訓練を受けた専門家ではない人々は，ソフトウェアとはいったい何かということにも，その歴史にも驚くほど無関心だ。ソフトウェアを与えられ，いかに使いこなすかを教えられはしても，ソフトウェアがこれまでどのように発展して来てこれからどういう方向に向かおうとしているのか，そのことは我々の生活にとってどんな意味を持つのかを理解しない。そのために，人間は「ソフトウェアとなって」自分の好きなハードウェアに棲むことができるようになるといった，ソフトウェアという人工物と人間の知性との区別すらつかない滑稽な主張も生まれる[3]。コンピュータの一般ユーザーであっても，ソフトウェアをいかに使いこなすかだけでなく，その本質や意味，歴史についても学ぶ必要があるのだ。ところが，そのために重要な役割を発揮すべきコンピュータソフトウェアの歴史的研究は大きく立ち遅れているので，その立ち遅れを取り戻し研究を大きく前進させる条件を整えること，それがこの会議の目的だった。

コンピュータソフトウェア史研究の現状について説明する前に，その背景としてコンピュータ史一般の動向を簡単に見ておこう。現代のコンピュータが登場してすでに半世紀以上たち，最近の20年間でコンピュータとその利用に関する歴史的研究は目覚しく進歩した。情報技術史を専門とするCBIの創設と活動を始め，コンピュータを中心とする情報技術史を研究しようとする研究者はアメリカを始め量・質ともに拡大した。1980年代初めにコンピュータ史を研

1）会議の報告集は2002年7月に出版された。Ulf Hashagen, Reinhard Keil-Slawik, Arthur Norberg (Eds.), *History of Computing : Software Issues*, Springer, 2002.

2）前掲書，p. 57.

3）前掲書，p. 8.

究しようとしたパイオニアは，コンピュータにそもそも歴史などあるかと問われたそうだ。彼らが90年代になって中堅研究者に成長し，例えばSociety for the History of Technologyの年会では情報技術史関係の分科会が複数開催されるようになった。IEEEから刊行されている専門誌*Annals of the History of Computing*を見ると，かつてコンピュータ史と言えば，ハードウェア開発が中心テーマだったが，今では，コンピュータの利用が与えた社会的，経済的，政治的，文化的インパクトへと対象が広く拡大した。1980年代初め，日本語で読めるコンピュータ史と言えば，Herman Goldstineの『計算機の歴史』[4]ぐらいだったが，これはvon Neumannとともに初期のコンピュータ開発に参加した当事者の回顧録だ。それが，今日ではCampbell-KellyとAsprayやCeruzziなど歴史研究者による一般史が手軽に読めるようになった[5]。MIT Pressからはコンピュータ史のシリーズが刊行されている。コンピュータ史をめぐる状況は大きく変わったのだ。

ところが，ソフトウェア史の現状を見ると，研究自体が少ない。しかも大部分は一部の技術的発展に狭く焦点を絞り過ぎ，特定のソフトウェア（例えば，プログラム言語，OS，アプリケーションソフト）の発展の回顧にとどまっている。ソフトウェアが重要な意味を持つプロジェクトが取り上げられたといっても，1950年代のSAGE防空プロジェクトやバンクオブアメリカのERMAシステムぐらいだ。最近のソフトウェアの発展を反映し，ソフトウェアの重要性と歴史の全体像が理解できるような研究がなされていない。そこで，ソフトウェア史研究の現状と今後の研究課題（アジェンダ）について再検討し，研究の主題と方法を探求することが，会議の目標となる。目標を実現するために，コンピュータ科学者やソフトウェア開発の実務家の観点と歴史学者や社会学者の観点の両方を総合して，ソフトウェア史研究の体系をつくるという方針をとる。コンピュータ科学者は重要な出来事や発展を当事者として良く理解していて，歴史学者や社会学者はそのことを良く学ぶべきだが，他方当事者の関心は分野内部の専門知識の発展に偏りがちだからだ。それに対して歴史学者や社会学者は技術や知識のみならず，その応用，組織や制度への影響など関心を広く持っている。以上のような会議の趣旨が，Norbergの開会の辞で説明された。

選ばれたテーマは

1．科学としてのソフトウェア，
2．工学としてのソフトウェア，
3．信頼できる人工物としてのソフトウェア，
4．労働過程としてのソフトウェア，
5．経済活動としてのソフトウェア，
6．博物館における歴史的ソフトウェアの収集と展示，

の6つであり，各々分科会が構成された。各分科会の報告者とコメンテーターは，コンピュータ科学者，企業など現場の実務家とそのOB，歴史家や社会学者と多様であり，コンピュータ史を専門とする研究者以外にも，数学史のMichael S. Mahoney，社会学のDonald MacKenzieとDavid Edge，技術史のDavid A. HounshellとBruce E. Seelyなど多彩な顔ぶれが揃った。会議の参加者はヨーロッパとアメリカを中心に60人ほどと小規模ではあったが，これだけ多様な参加者が一堂に会してソフトウェア史の現状と課題について3日間かけて集中

4）ハーマン・H・ゴールドスタイン著，末包良太・米口肇・犬伏茂之訳『計算機の歴史－パスカルからノイマンまで』，共立出版，1979年（原著：Harman H. Goldstine，*The Computer from Pascal to von Neumann*, Princeton U.P., 1972）。

5）Martin Campbell-Kelly and William Aspray, *Computer : a history of the information machine*, Basic Books, 1999（邦訳:山本菊男訳『コンピューター200年史－情報マシーン開発物語』，海文堂，1999年）．Paul E Ceruzzi, *A History of Modern Computing*, MIT Press, 1998;1ST ed., 2003;2nd ed.（邦訳：近刊予定）．

的かつ自由に討論したことは，この問題への期待の大きさを示すものであり，これからの発展が期待される。

本稿ではとりあえず，会議の6つのテーマのうち最初の3つを取り上げ，報告と議論の概要を紹介したい。なお，「科学としてのソフトウェア」と「信頼できる人工物としてのソフトウェア」の原稿の一部は，2002年10月の津田塾大学数学史シンポジウムにおける私の講演をもとにしている。シンポジウム主催者にあらためて御礼申し上げる[6]。

第1部　科学としてのソフトウェア

科学としてのソフトウェアの歴史，ソフトウェアの科学史とは奇妙な感じもするが，ソフトウェアの理論的基礎を提供する理論コンピュータ科学の歴史とも，あるいは科学におけるコンピュータ利用の歴史とも解釈できるだろう。科学史におけるコンピュータ研究のパイオニアMichael S. Mahoneyの講演「科学としてのソフトウェアーソフトウェアとしての科学」は理論コンピュータ科学形成の壮大な図式を展開し，理論コンピュータ科学の認識論的基礎の考察に及ぶものだったのに対し，イギリスの社会学者David Edgeとドイツのコンピュータ科学者Gerhard Goodsがコメントを加えた。

Mahoneyの講演「科学としてのソフトウェアーソフトウェアとしての科学」

Princeton大学歴史学部のMichael S. Mahoneyは近世数学史の専門家であり，フェルマー研究の単著がある[7]。他方，理論コンピュータ科学の形成や技術史として見たコンピュータなど，活発に論文を発表している[8]。Mahoneyの講演は，3つの問題を論じた。第1に，理論コンピュータ科学（科学としてのソフトウェア）の形成とその研究手段としてのアジェンダ(agendas)，第2に，理論コンピュータ科学とソフトウェア工学との関係，第3に，コンピュータを研究手段として利用する科学（ソフトウェアとしての科学）から見た理論コンピュータ科学だ。

理論コンピュータ科学の形成と「アジェンダ」

科学の本質は知識と実践，つまり科学者の共同体が共通に何を知り何を行なうのかにあると考える。そうすると，科学の形態は，グループが違い，時代や場所が違えば異なることになり，学問的であると同時に社会的な性格を持つ。そこで，科学の実践に焦点を当て，「アジェンダ

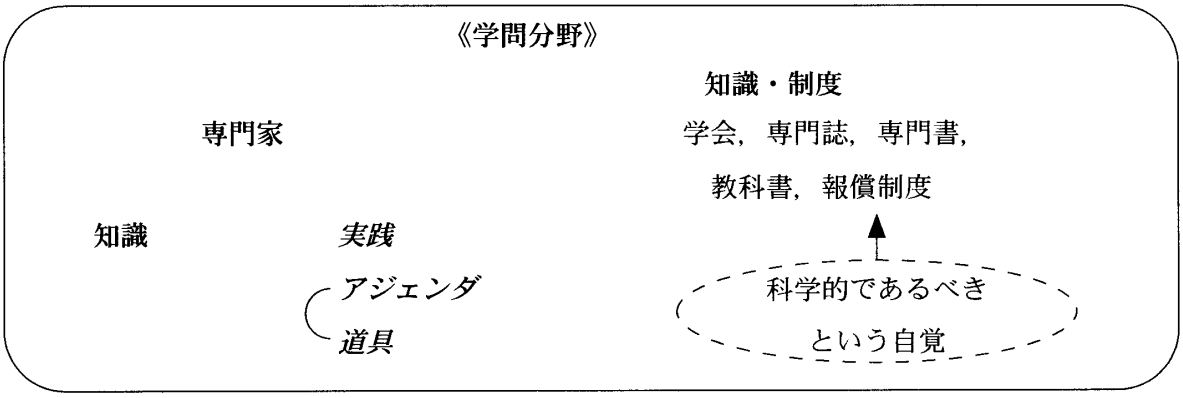
6）吉田晴代「コンピュータソフトウェア史の現状と課題」，『津田塾大学数学・計算機研究所報』24，2003年，pp. 158-175.

7）Michael S. Mahoney, *The MathematicalCareer of Pierre de Fermat 1601-1665*, 2nded., Princeton University Press, 1973.

8）主な論文を挙げると以下の通り。Michael S. Mahoney, "The History of Computing in the History of Technology," *Annals of the History of Computing*, 10 (2), 1988, pp. 113-125（吉田晴代訳「技術史におけるコンピューティングの歴史」，『産研論集』，札幌大学経営学部附属経営研究所，23，2000年，pp. 67-83）；"Computers and Mathematics: The Search for a Discipline of Computer Science," J. Echeverria, A. Ibarra, T. Mormann (eds.), *The Space of Mathematics-Philosophical, Epistemological, and Historical Explorations*, De Gruyter, 1992, pp. 349-363; "Computer Science: The Search for a Mathematical Theory," J. Krige and D. Pestre (eds.), *Science in the 20th Century*, Harwood Academic Publishers,1997, Chap. 31; "The Structures of Computation," R. Rojas and U. Hashagen (eds.), *The First Computers-History and Architectures*, MIT Press, 2000, 17-32. マイケル・S・マホーニィ著，佐々木力解説，林知宏訳，「コンピュータ科学の形成」（上）・（下），『UP』，2001年343号及び344号，東京大学出版会。

(agendas)」[9]という概念を用いると，コンピュータ科学のような新生の学問分野の形成をうまく理解できる。アジェンダとは，その学問分野で「何を行なうべきか？」ということであり，「何が問題か？ 問題の優先順位，問題を解く道具（手段・方法）は何か？ 何が解か？」といったことから構成される。従って，その学問分野で専門家であるということは，アジェンダを知り，その実行を助け，アジェンダを確立する能力を持つこと，つまり何が問題かを知り，アジェンダを拡張・再形成するような仕方で問題を解いたり，深い問題を提出することだ。アジェンダとアジェンダを実行する道具は不可分であり，道具はアジェンダを体現する。アジェンダに関するMahoneyのこの説明をもとに図式化すると，おおよそ次のようになる。

理論コンピュータ科学は1955年から1975年の間に，独立した学問分野として形成された。電子工学から言語学まで様々な分野出身の専門家が関連するアジェンダの小さな集まり（例. オートマトン，形式言語，計算量，形式的意味論）を形成し，その集まりが核となって理論コンピュータ科学を形成した。これらのアジェンダは1955年以前にはなく，専門家のグループが元々属していた専門分野のアジェンダとは異なる新しいアジェンダに合流することで形成された。彼らが出身分野からどんな道具を持ちこみどう適用したか，それによって新しいアジェンダはどのようにして形成されたかを明らかにすることが研究課題となる。こうしたアジェンダへの収束の各点で，新生の分野は親に当たる分野からどんな道具を獲得したか，道具は元の分野で何のためにつくられたか，コンピューティングに利用することで新分野の形成にどう寄与したか，適用の結果その道具はどう作り直され元の分野における位置に磨きをかけるのにどう役立ったか，といったことが問題になる。理論コンピュータ科学におけるアジェンダはどう形成されたのか，Mahoneyは，自分のこれまでの研究に基づいて２つの事例[10]を説明しながら，研究課題を提示した。



9）元々は「なされるべきもの（things to be done)」というラテン語の複数動名詞だが，英語では「なすべきもののリスト（list of things to do)」という意味の単数名詞だ。Mahoneyは「なすべきもの多数の集まり」という意味で複数名詞を使う。注２の文献（Mahoney2000)，p.20より。

10）これらの事例の詳しい内容については，注２の文献（Mahoney, 1992)，(Mahoney, 1997)，(Mahoney, 2000)，及び（マホーニィ，2001）を参照。

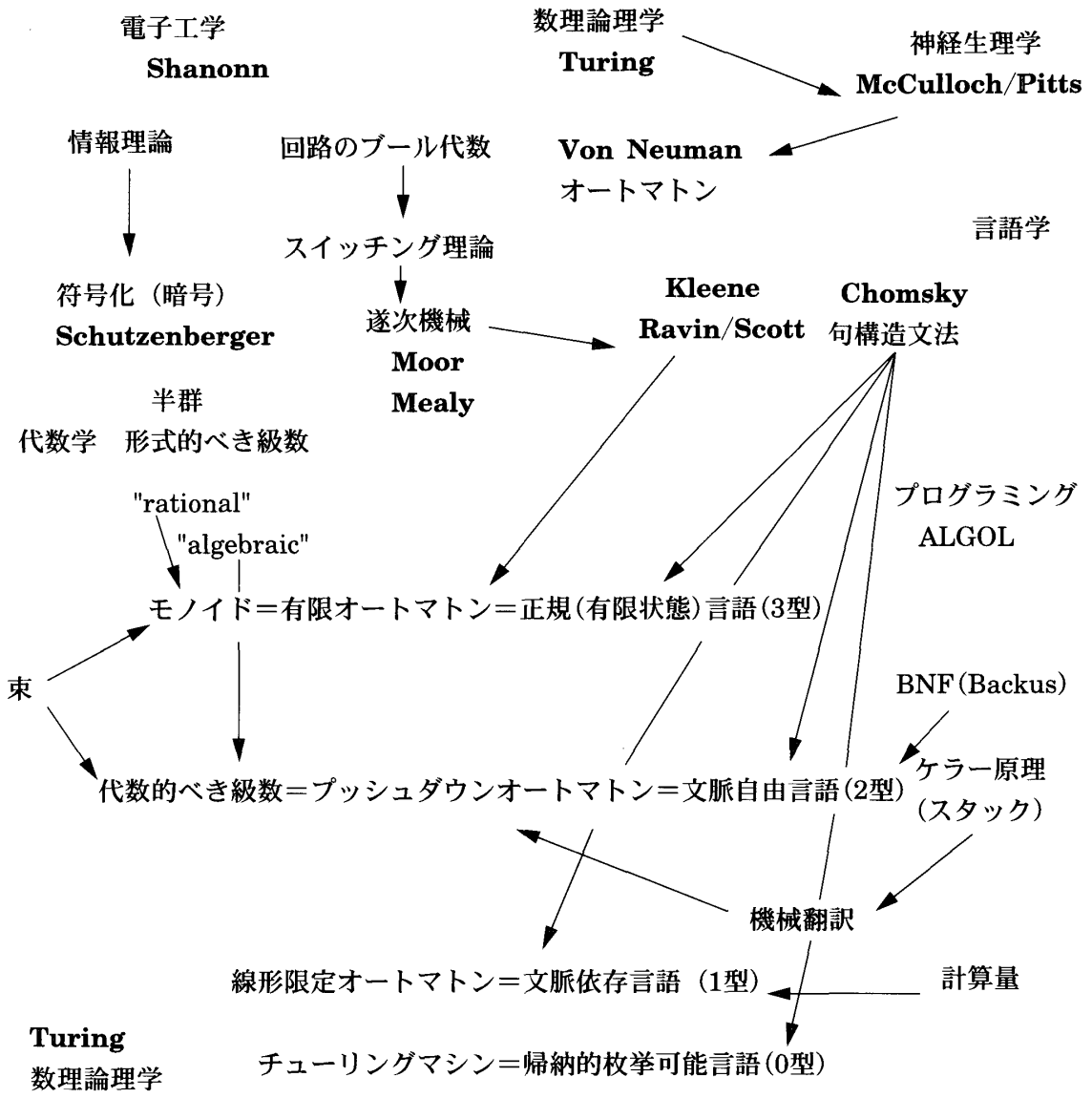


図1．オートマトンと形式言語のアジェンダ（実行計画）

《例 1. オートマトンと形式言語のアジェンダの形成》（図1）コンピュータ科学は独自のアジェンダを持たずに出発した。出発点は科学理論ではなく物理装置としてのコンピュータだった。コンピュータ科学にとって親にあたる数理論理は計算可能性の問題（すなわち，時間・空間が無限でもコンピュータに何ができないか）を明らかにし，スイッチング理論は基本演算のために回路の総合・分析の手段を提供した。だが，有限なランダムアクセス式メモリーを持つ有限なマシンに何ができるか，それを実行するにはどうしたら良いかを明らかにする理論はなく，それが必要とされた。重要なアジェンダの収束の例として，形式言語とオートマトンの理論の形成がある。文法能力の数学的理論という言語学者Chomskyのアジェンダと，代数学と数論を出発点とするブルバキ派数学者Schutzenbergerの数学的アジェンダがあった。ブルバキ数学の半群のような数学的基礎構造が道具として符号の数学的解析に役立ち，符号の問題は有限オー

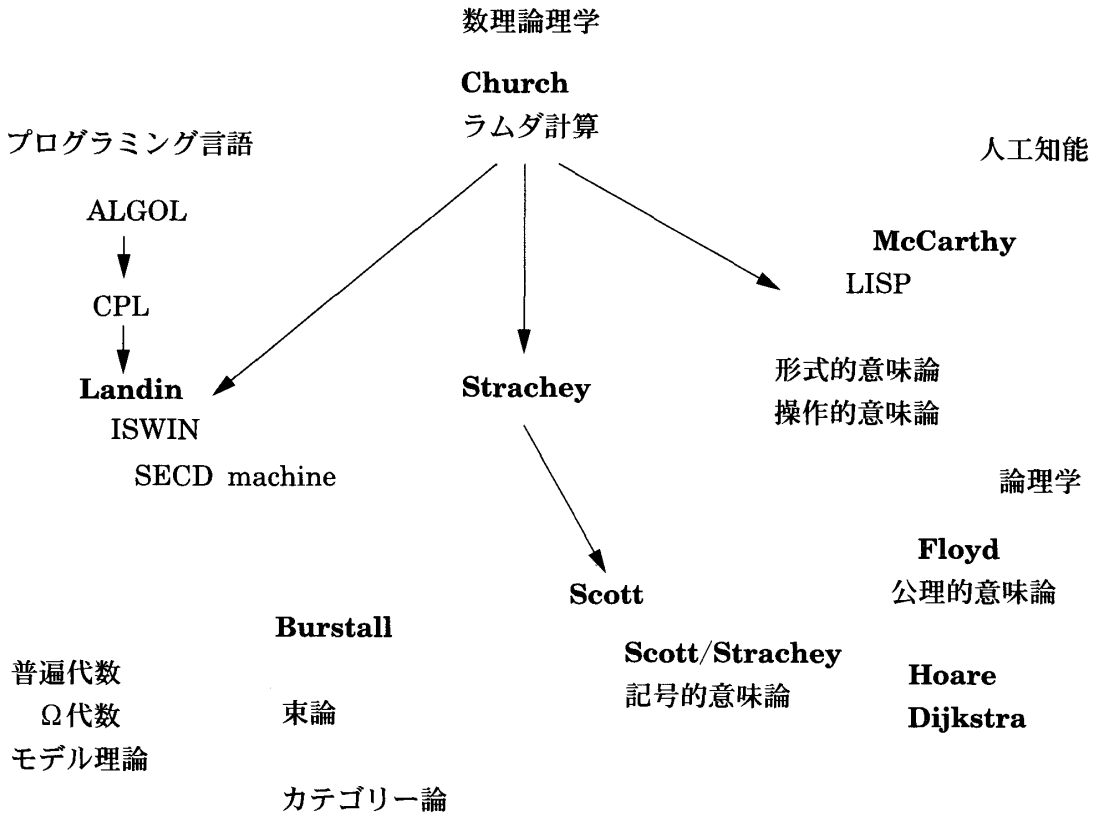


図２．意味論のアジェンダ

トマトンと関係することが明らかになると，Schutzenbergerは数学的一般化を追求すると同時に，句構造文法では自然言語の研究に不十分なことを理解したChomskyと共同研究を推進した。その結果，Chomskyの言語学のアジェンダ，機械翻訳のアジェンダ，代数的プログラミング言語のアジェンダが合流し，代数的べき級数とプッシュダウンオートマトンと文脈自由言語の等価性が明らかになった。この等価性は，ALGOL60が文脈自由言語を構成するとわかると，コンピュータ科学の基礎となった。アジェンダはいろいろな場所で形成された。形式的べき級数とプッシュダウンオートマトンと文脈自由言語を同一視することで，パリのSchutzenbergerの代数的符号理論からミュンヘンのFritz Bauer達の逐次形式翻訳に至るアジェンダがMITにおいて合流し，自動プログラミングの取組みに発展するという具合にだ。

《例２．意味論のアジェンダ形成》（図２）もう１つの重要な例が，（普遍）代数，数学的論理学，プログラム言語，人工知能の相互作用による意味論のアジェンダの形成だ。λ計算は数理論理学の概念として1930年代にChurchがつくった。1950年代後半になって，人工知能を研究していたMcCarthyが，定理の機械証明とコンピュータによる常識的が推論に関する研究の手段として復活させた。その後，統語論よりは意味論を重視した計算の数学的理論の基礎としても活用した。McCarthyの形式的意味論のアジェンダ（λ計算は道具）はヨーロッパに伝わり，ケンブリッジのLandinやStracheyの所に定着，さらにアムステルダムやウィーンのScottなどにも伝わり，プログラム言語の意味論やプログラムの証明に関する研究で普遍代数とも結びついて，実り豊かな研究成果を生み出した。だが，McCarthyの地元アメリカでは実践的な関心が

支配的で，McCarthyのアジェンダには冷淡だった。

以上２つの事例の検討から以下の有用な点が明らかになるとMahoneyは言う。

（1）コンピュータ科学とそれへの数学の応用との関係は双方向的だ。数学はコンピュータ科学に数学的基礎を提供する一方，コンピュータ科学は数学の抽象的概念に「物理的」意味を与えたという具合に相互作用した。例えば，コンピュータ科学が，半群，束，圏といった最も抽象的な数学的構造に実際的な意味を与えた。コンピュータのことを考えてつくられたわけでもなく，数学においてさえ抽象的で役に立たないと言われたのに，コンピュータ科学に応用された結果，予想されなかった性質や関係が明らかになり数学研究を刺激した。λ計算により，意味論と普遍代数や圏論の間にも同様の相互作用が生じた。

（2）アジェンダ形成の初期には関心や方法に地域性が強く反映した。研究グループが異なれば，教育も好みも構想の混ざり具合も違い，違いには文化的および制度的背景の違いが反映した。そこで，地域的グループがどのようにして共通のプロジェクトに合流し，新しい学問分野としてのアジェンダを形成していくのか，実践がいかに移動していくのかが研究課題になる。

アジェンダの形成は，研究助成の獲得[11]や教育過程といった，学問的実践のもっと異なる側面からも解明される。中でも注目されるのは，教育の側面から見ると，Mahoneyも認めるように，アジェンダは，Thomas Kuhnの「パラダイム」概念[12]とほぼ重なることだ。その学問分野の専門家になるというのは，アジェンダを知ること，つまり何がなされるべきかを知ること，その分野の問題が何であり，どう取組まれ解かれるべきか，解は何かを学ぶことだ。これは，模範例とその解，つまり問題解決のモデルが次世代へと伝えられていくという，Kuhnのパラダイムの考え方とぴったり重なる[13]。だが，両者は全く同じというわけではない。パラダイムが同一学問分野における，例えば地球中心説から太陽中心説へといった転換（学問分野の発展とはどういうことか，それを促す要因は何か）を説明するのに適しているのに対し，アジェンダは，コンピュータ科学のような新生の学問分野形成を研究するのに適している。アジェンダと道具の組合せにより他分野との相互作用の中で新分野形成をダイナミックに捉えることができるし，またアジェンダ形成に焦点をあてることで，学問分野形成の社会的，政治的，経済的，文化的要因をも考慮して多面的な理解が可能になる。

11）Mahoneyによれば，研究助成の問題は新生の学問分野のアジェンダ形成につながる重要なファクターだ。研究助成は単なる金の問題ではなく，資金援助者である政府機関（コンピュータ科学の場合，国防総省高等研究計画局や全米学術財団）と科学者のコミュニティの相互作用はアジェンダの形成に重要な影響を及ぼす。自分達がどのような学問的実践を追及しようとしているのかを資金援助者の要求や期待に応える形で明らかにすること，その際，他の関連学問分野（コンピュータ科学の場合，数学）からの評価も，新生の学問分野にとって重要だ。

12）Thomas S. Kuhn, *The Structure of Scientific Revolutions*, Chicago U.P., 1962；2ed ed, 1970（邦訳:中山茂訳『科学革命の構造』，みすず書房，1971年）．トーマス・クーン著，安孫子誠也・佐野正博訳『本質的緊張』第２巻，みすず書房，1992年（原著：Thomas S. Kuhn, *The Essential Tension*, Chicago U.P., 1977），第12章「パラダイム再考」。

13）そこで，Mahoneyによれば，教科書や教育課程は新生の学問分野の発展を研究するための重要な資源となる。教育課程をつくるには，その学問分野がどんなものか，何が中心で何が周辺か，何が誰でも知るべき問題で何が選択科目かといった問題について学問分野内部で合意形成が必要なので，そこにアジェンダが反映する。歴史家から見れば，完成した教育課程のみならず，それをつくる過程もまた重要な研究対象なのだ。

数学とソフトウェア工学

理論コンピュータ科学が，ソフトウェアの開発や利用といった日常の実践にどこまで役立っているのかということは，しばしば問題になる。HoareやDijkstraのようなコンピュータ科学者は，これまでの科学技術における理論と実践の伝統的関係に従って，理論コンピュータ科学がソフトウェアの実践に対して基礎科学の役割を果たすことを期待した。だが，現状は理想から程遠く，両者の間には埋め難いギャップが存在する[14]。だが，歴史家から見れば，この一貫したギャップがあることこそ研究課題だ。Mahoneyはこの問題にどう取組むか，いくつか研究の方向を示唆した[15]が，そのうちの１つを紹介したい。ソフトウェア開発のウォーターフォールモデルを取り上げ分析して，こうした理論と実践のギャップは，Hoareのようなコンピュータ科学者の理想を追求しさえすれば解決するものではないという，ソフトウェアの別の側面を明らかにしたのだ。

図３のモデルで図式の下半分は，数学的に最も良く理解できる部分，つまり理論コンピュータ科学の対象となる部分であり，ここで重大な誤りが生じることはほとんどない。モデルの図式上半分はソフトウェア工学が注目する，大量の決定的誤りが生じる場所だ。この分析は，ソフトウェア開発の立場から，Frederick Brooksがソフトウェア構築の「偶有的」作業と「本質的」作業と呼んだものに対応する[16]。アリストテレスの用法に従って，「本質的」とはソフトウェアの性質に固有なもの，「偶有的」とは実現するとき派生する付随的，副次的なものを意味する。Brooksは，ソフトウェア生産性向上の可能性を探るためソフトウェア構築上の主要

---

14) コンピュータ科学者のHoareは，「Hoareの原理」とも言うべき自分の学者としての信念と現実の乖離についてこう述べた。

我々の原理は以下の４つの見出しに要約される。

（1）*コンピュータは数学的機械である。*その働きのあらゆる側面が数学的厳密さで定義でき，どの細部も純粋な論理学の法則を用いて，この定義から数学的正確さで導かれ得る。

（2）*コンピュータプログラムは数学的表現である。*それらは，先例のない厳密さで，どんな微小な細部でも，それらが実行するコンピュータの働きを，意図されたものかどうかに関わりなく，記述する。

（3）*プログラム言語は数学的理論である。*それは，概念，記号，定義，公理および定理を含み，プログラマーが仕様に合ったプログラムを発展させ，プログラムが仕様通りに動くことを証明するのを助ける。

（4）*プログラミングは数学的活動である。*応用数学や工学の他の分野同様，数学的理解，計算，証明の伝統的方法を明確かつ細心に適用することが実践を成功させるために必要だ。

これらは一般的な哲学的かつ道徳的原理であり，私には自明なことに思える。そう言った方が良いのは，現実の証拠がそれらの原理に反するからだ。私が述べたようなことは現実には成り立たない。コンピュータについても，プログラムについても，プログラム言語についても，プログラマーについてもだ。C.A.R.Hoare, "The Mathematics of Programming," C.A.R.Hoare, *Essays in Computing Science,* Prentice Hall, 1989, p. 352. 講演は1985年。

15) １つは，工学としてのソフトウェアの側面，つまりアルゴリズムに解消されないコンピューティングの機構，手段そして非決定論的に重要な側面に注目することであり，これは，社会的，政治的，経済的要素とも関係する。もう１つは，科学としてのソフトウェアの側面に注目することだ。数学が，定理，無限過程，静的関係を扱うものであるのに対し，コンピュータ科学の対象はアルゴリズム，有限構成，動的関係である点で，数学とコンピュータ科学は異なる性格の学問なのだ。

16) Frederick P. Brooks, *The Mythical man-month: essays on software engineering,* Addison-Wesley Publishing Company, Anniversary edition 1995 (邦訳:滝沢徹・牧野祐子・富澤昇訳『人月の神話－狼人間を撃つ銀の弾はない』，アジソンウェスレイ，1996，第16章「銀の弾などない－ソフトウェアエンジニアリングの本質と偶有的事項)。

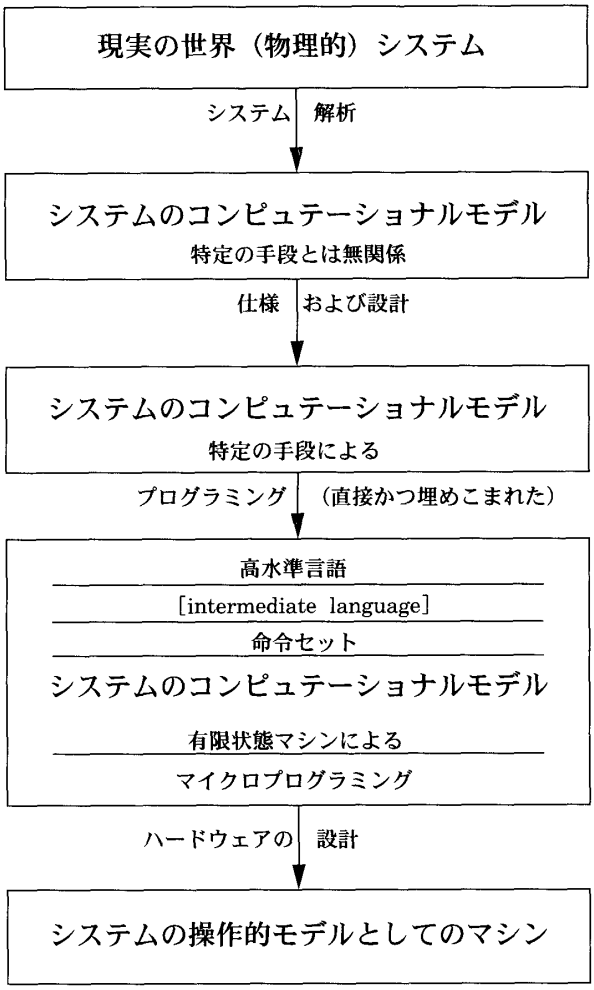


図3．ソフトウェア開発：抽象の各段階

な困難を分析した。その結果，難しいのは何を表現するのか決定すること（本質的作業）であって，表現することそれ自体（偶有的作業）ではないとわかった。

MahoneyはこのBrooksの分析結果を，ソフトウェアとしての科学，つまり科学研究におけるコンピュータ利用の問題として読み替えた。するとこの場合にもやはり，ソフトウェア開発と同様，現実の世界の一部をソフトウェアつまりコンピュータを利用したモデルによってどう表現するのか，そのモデルが適正か否かといった問題が生じる。だが，それは，Mahoneyが論じている科学としてのソフトウェアつまり理論コンピュータ科学の問題（つまり，プログラムが行なうべきこととして我々が書いたことをプログラムが実行していると我々はどのようにしてどの程度まで確かめられるのか）ではない。ソフトウェア開発者もしくはコンピュータを研究手段としで利用する科学者にとって，対象とする現実世界の問題なのだ。この問題は，コンピュータの利用が大きく進み，科学のあり方にまで影響を及ぼすかのような現代科学にあっては，科学史研究の重要な課題だ。そうMahoneyは言う。

コンピュータ科学と数学

ソフトウェアとしての科学と科学としてのソフトウェア（理論コンピュータ科学）との関係が今述べたようなものであれば，理論コンピュータ科学とはどのような学問であるべきかという本質的問題に立ち戻ることになる。科学研究においてコンピュータを用いてモデルをつくることの意味が問われるからだ。

伝統的なモデル観では，モデルとは，より良く理解できるか直接操作可能なメカニズムもしくは数学的関係であり，モデルの目的は現象の働きを理解するのに役立つこと，モデルの優秀さの基準はモデルと観測された現象との経験的な適合度にあった。17世紀以来の近代科学では，自然は物理モデルに還元され，物理モデルは数学的関係に還元された。その際，数学的関係は物理モデルの構造を反映し，物理モデルは自然の構造を反映するものとされた。それに対して，von Neumannは，自然と数学とを物理モデルが仲介するという近代科学のモデル観を否定し，数学的構造自体が物理的世界でも社会でもモデルになるという新しいモデル観を主張した。科学者は現象にぴったりのモデルをつくりさえすれば良く，モデルが現実世界の本当の構造の反映であるが否かに関心を持つ必要はないのだ。

> まず初めに強調しなければならないのは，もう前に聞いたことがきっとあると思うのだが，やはり繰返し述べておかなければならない言明だ。科学は説明しようと

しなくて良い。解釈しようとさえしなくて良い。科学とはモデルをつくることだ。モデルが意味するのは，一定の言葉による解釈と一緒に，観測された現象を記述する数学的構造だ。そのような数学的構造の正当化は，機能するように期待されるということ，つまり正当な広さの範囲から現象を正しく記述するということだけだ。それに加えて，ある種の審美的基準を満たさなければならない。つまり，どのくらい多くを説明するかに関して，むしろ単純でなければならない。例えば，言葉を使うより，こうした漠然とした事柄を主張する方が価値があると思う。「単純」がいかに単純かを正確に説明することはできない。我々が採用する理論のあるもの，我々がそのおかげでとても幸福でそれを誇りに思っているモデルのいくつかは，初めてそれらに接した人々にとくに単純なものとして感銘を与えるようなことはなかっただろう。[17]

要するにvon Neumannにとって，科学とはモデルをつくることであり，モデルとは観測された現象を正しく記述する数学的構造であり，審美的観点からモデルは単純であることが望ましいのだ。重要なことは，von Neumannは，伝統的モデル観を完全に否定したのではないということだ。モデルの数学的構造が解析学的に取扱い可能なものであり，研究者はモデルがどう働くか理解することは前提だった。

だが，この新しいモデル観がvon Neumannを困らせることになった。当時の主流の数学つまり解析学で解けない非線形偏微分方程式に関連した問題を解くためにコンピュータを利用することになったからだ。問題はコンピュータの与える数値解がモデルの条件を満たすかということだ。数値解は，解析的モデルのように研究者に理解可能な構造を提供しない。解が予想に反したときどこに問題があるか確定するのが難しいし，打ち切りや丸めによる誤差などコンピュータを利用することで生じる独自の問題もあった。コンピュータに適した数値解法も研究された[18]が，それだけでなく新しい計算理論が必要になった。そこで，von Neumannはコンピュータの数学的基礎としてオートマトンの理論を考えたのだ。オートマトンに関するHixon Lecture（1948）でこう述べた[19]。

今日では，形式論理学，とりわけ数学に応用される論理学のとても入念に仕上げられた体系が存在する。このことは，学問分野として優れた側面でもあるが，同時に深刻な弱点でもある。これは，すぐれた側面を拡大する機会ではない。それを軽視するつもりもないが。だが，不充分さについてはこう言える。形式論理学を研究してきた者は誰でも，数学の技術的に最も手に負えない部分の一つであることを認めるだろう。その理由は，形式論理学が厳密な全か無かの概念を扱うからであり，実数や複素数の連続の概念，つまり数学的解析とほとんどつながりを持たないからだ。だが，解析学は数学の中で技術的に最も成功した最も入念につくられた部分である。それゆえ，形式論理学は，そのアプローチの性格からして，数学の最も良く耕された部分から切離され，数学の最も困難な部分である組合せ数学に組込まれざるを得ない。デジタル式の全か無かタイプのオートマトンの理論は，今まで考察したことから，形式論理学の一章であることは間違いない。その理論は数学的観点からすると解析学ではなく，組合せ論に属さなければならない。

ここでvon Neumannが言いたかったことは，

17) John von Neumann, "Method in the Physical Sciences," *The Unity of Knowledge*, ed. L. Leary, 1955; John von Neumann, *Works*, VI. P.492.

18) ウィリアム・アスプレイ著，杉山滋郎・吉田晴代訳，『ノイマンとコンピュータの起源』，産業図書，1995（原著：William Aspray, *John von Neumannand the origins of modern computing*, MIT Press, 1990），第5章。

19) John von Neumann, "On a Logical and General Theory of Automata," *Cerebral Mechanisms in Behavior-The Hixon Symposium*, ed. L. A. Jeffries (New York, 1951), pp.1-31; *Papers of John von Neumann on Computing and Computer Theory*, William Aspray and Arthur Burks, (eds.) MIT Press, 1987, pp. 391-431のp.406.

コンピュータを利用して現実世界のモデルをつくる前提として，新しい計算理論（コンピュータの抽象的な働きを数学的に理解する理論）が必要であり，それがオートマトンの理論だということだ。解析学で解けない問題をコンピュータを利用して解くというのであれば，コンピュータを利用する計算過程に関する厳密な数学的理論としてオートマトンの理論が必要なのだ。Mahoneyはこれを解析学の失敗を穴埋めすると説明した。オートマトンの理論は，TuringやMcCulloc／Pittsのような先行研究を考えると，形式論理学に属するように思えるが，解析学に頼らず，形式論理学（数学的に言えば組合せ数学）だけではオートマトンを十分理解することはできない。そこでvon Neumannは，オートマトンの理論を「解析学の領域に引き戻したい」と考えた[20]。「組合せ論的ではなく，もっと（数学的な意味で）解析学的な数学的理論を得たいと思った」[21]のだ。コンピューティングの基礎となる新しい数学理論として，解析的モデルが研究者にとって理解可能なのと同じくらい理解可能（で厳密な数学的関係に基づいた）な数学的モデルを提供できるオートマトンの理論が必要だ，そういうvon Neumannのコンピュータ科学のアジェンダが継承され発展した流れを示すのが図４だ。この図で注目されるのは，研究のもとになる関心が一貫して，コンピュータによる数値解法，つまりコンピュータを利用して現実世界のモデルをつくるということにあり，プログラミングへの関心から生じたものでないことだ。図の中でアジェンダの発展の終点近くに位置するJohn Hollandの次の言葉もそのことを示す。

> 数学は旅のこの部分にぜひとも必要だ。…数学が決定的な役割を果たすのは，厳密な一般化かもしくは原理を定式化できるようにするからだ。物理学の実験もコンピュータに基づいた実験もそのような一般化を提供できない。物理学の実験は一般に厳密なモデルのための入力と制約を供給するだけだ。実験自体が演繹的な説明を許容する言語で記述されることはほとんどないからだ。コンピュータに基づいた実験は厳密な記述を持つが，特定の場合を扱うに過ぎない。よく設計された数学的モデルは，何といっても，物理学の実験，コンピュータに基づいたモデル，それに学問分野を超えた比較によって明らかにされる詳細を一般化する。…数学だけが我々を十分遠くまで連れて行ってくれる。[22]

ここに示された数学的モデルの役割は，デカルト以来の数学像を反映している。「モデルの数学的構造が解析学で取扱い可能で，研究者はモデルの働きを理解できる」ことを前提しているのだ。解析の手段が解析学からコンピュータに変化したことで，「関心を持つ世界の部分を信頼性をもってシミュレートし，プロセスの解析と理解とを可能にする仕方でシミュレートする」モデルの問題となったのだ。世界を理解することがコンピュータの計算過程の理解に左右され，ソフトウェアが科学の象徴になろうとしている今，20世紀後半にそれがどのようにして起こったのかが，間もなく科学史の大問題になる。そのとき「歴史感覚に優れた」ソフトウェア史は問題を解く助けになる。これがMahoneyの結論だ。Mahoneyは，理論コンピュータ科学（科学としてのソフトウェア）の形成のみならず，コンピュータを利用する科学研究（ソフトウェアとしての科学）とそこから生じるモデルの問題を提起することで，科学史研究におけるソフトウェア史の意義を明らかにした。

コメント及び議論で問題になったこと

David Edgeは科学知識の社会学のエジンバラ学派の一員として，コンピュータ科学の内容

20）マイケル・S・マホーニィ著，佐々木力解説，林知宏訳，「コンピュータ科学の形成」（下），『UP』，2001年343号及び344号，東京大学出版会，p.29.

21）ウィリアム・アスプレイ，前掲書，pp. 207-208.

22）John H. Holland, *Hidden Order: How Adaptation Builds Complexity,* Perseus Books, 1995.

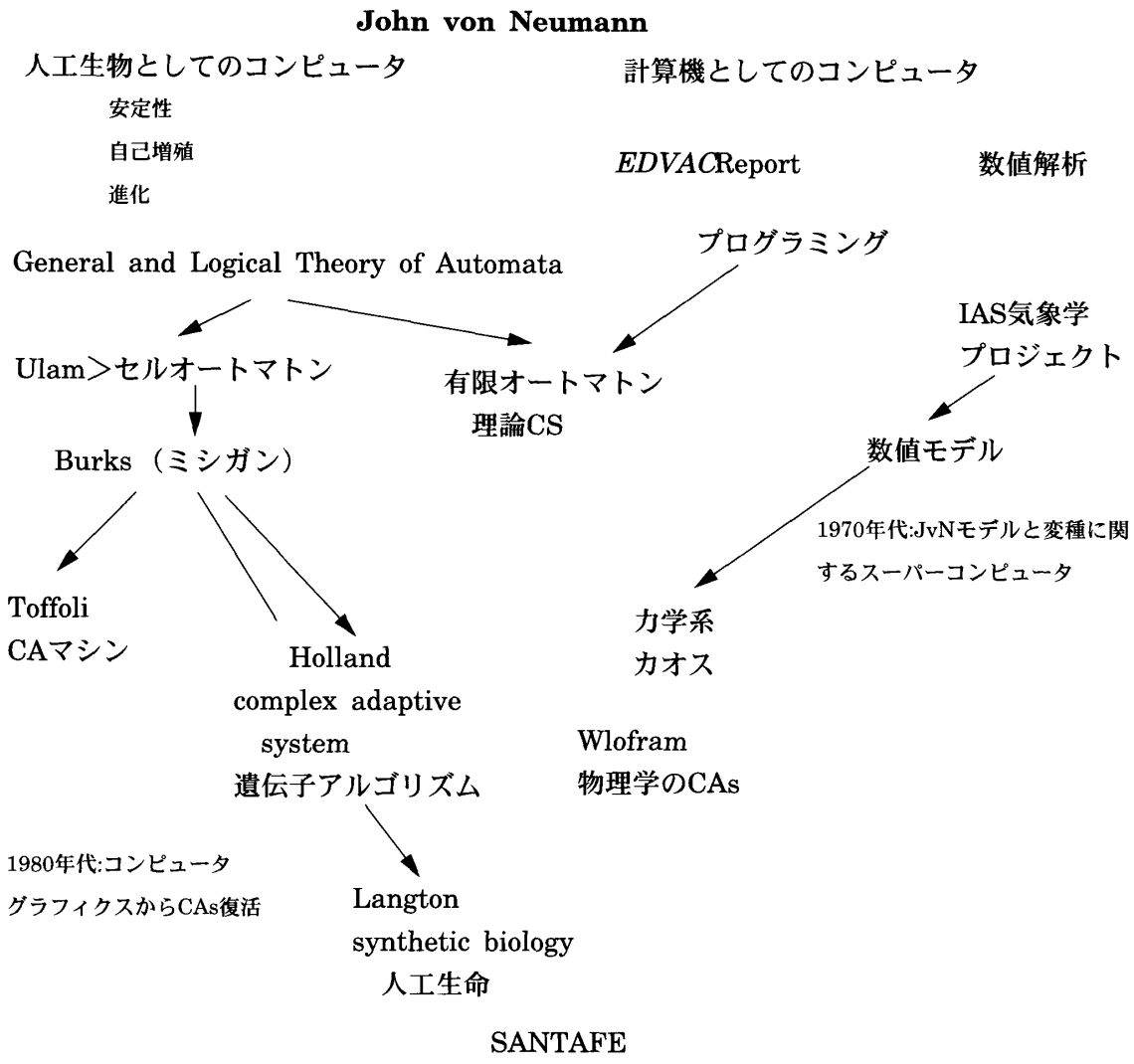


図4．Von Neumannのコンピュータ科学のアジェンダ

にはそれほど踏み込まず，科学技術史研究一般の観点から自分の研究を踏まえてMahoneyの報告にコメントを加えた。要点は2つある。第1に，理論コンピュータ科学とソフトウェアの発展に伴って，他の学問分野との間，および科学と技術としての実践との間に各々生じる境界確定の問題だ。第2に他の新生の学問分野との比較研究が有用だということを，サイエンススタディーズや電波天文学を例に説明した。また，自分が編集している雑誌*Social Study of Science*の掲載論文を例に，国同士の比較や修辞分析など有効と思われる研究方法を示唆した。

Gerhard Goodsは，大陸ヨーロッパ（ドイツ）のコンピュータ科学者らしく，情報科学を意味するInformatik（フランス語Informatique）という用語に焦点を当てた。これは，このヨーロッパ独特の用語とcomputer science（米;日本語で計算機科学）という用語の対立をめぐる論争の種になった。Mahoneyは，シェークスピアの"What's in a name?" を引用しつつ両者の内容にはほとんど違いはないと反論しているが，哲学的背景など考えると問題はそう簡単ではないだろう。Goodsは，モデルと実在，ソフトウェアやコンピュータ科学の性格などにつ

いても，Mahoneyとはかなり異なる角度から考察を加えた。例えば，モデルはもともとプラトン哲学の対象だが，ソフトウェアで表現されると実在になるとか，他の科学分野が科学的進歩により評価されるのと違い，情報科学の進歩が応用の発明や改良により評価されるのはなぜかといった具合だ。いずれにしてもコンピュータ科学に対する考え方にも地域性が反映することの1つの事例だ。

締めくくりの討論で，とくに驚いたのは，Mahoneyが提示したコンピュータ科学のアジェンダ形成の壮大な図式にもかかわらず，Mahoneyの講演がカバーした理論コンピュータ科学の範囲が狭過ぎるという指摘がなされたことだ。例えば，アルゴリズムやデータ構造など重要な数学的研究が含まれていないというのだ。Mahoneyはこれを認めた上で，自分の提案したアジェンダとその収束という概念が計算量，データベース，計算幾何学，並列計算のようなテーマに関する研究にも有効なはずだと主張した。この議論からうかがえるのは，理論コンピュータ科学史の研究対象が豊富でしかも，まだ未開拓の部分が大きく残されていることだ。もう1つ議論の焦点になったのが，ソフトウェアの科学があり得るか否かだ。講演の冒頭でMahoneyは，Herbert Simonの「人工物の科学」[23]を引き合いに出し，ソフトウェアにもソフトウェアの科学にも自然は存在しないと述べた。コンピュータもプログラムも，プログラムがつくり出す世界も人工物だからというのだ。これが議論の引き金になった。科学は自然に関するもので，人工物に関するものではないというのだ。だが，こういう議論は，Mahoneyにしてみれば不本意であり，歴史家として誤った問題の立て方なのだ。17世紀科学革命は自然と人工物という古典的区別を拒否することから生じた。機械論哲学における「時計仕掛けの宇宙」というメタファーはその証拠だ。それに蒸気機関から熱力学が生まれ，通信技術から情報科学が生まれた事実を無視するものだ。それに科学の定義は，様々な時代，環境，場所に応じて異なっていた。だから大切なことは，なぜコンピュータ科学と呼ばれるようになったか，コンピュータ科学者は何を考えたか，実際に何が進行したのか，を見守ること，それが歴史家の任務だ。こうしたことが議論になるのは，ソフトウェアに関する歴史的研究のみならず哲学的研究の展開が不十分なことを意味するのかもしれない。

第2部　工学としてのソフトウェア

工学としてのソフトウェアの歴史，つまりソフトウェアの技術史はソフトウェアの専門家ではない歴史家にとって研究するのがもっとも困難な分野の1つであるように思われる。事実，1980年代以後コンピュータ一般の技術史研究の成果は増大の一途だが，ソフトウェアの技術史研究は少ない。あまり少ないのでソフトウェアの意味を拡大解釈し，コンピュータ利用の歴史まで含めることにしても，社会史的な研究が大部分で，サブルーチンの開発といったテーマの本格的な技術史研究は少ない[24]。この困難な歴史研究の課題に対してどう取組んだら良いのか，アメリカ技術史・技術論研究における「工学とは何か？」という議論を踏まえて研究のアジェンダを示そうとしたのが，James E. Tomaykoの講演「工学としてのソフトウェア」だ。これに対し，Albert Endresがソフトウェア技術者の立場から，そしてBruce Seelyが技術史家としてコメントを加えた。

23) Herbert Simon, *The Science of the Artificial*, MIT Press, 1969; 2nd ed., 1981; 3rd ed., 1996（邦訳：稲葉元吉・吉原英樹訳『システムの科学』，パーソナルメディア，1987年）.

24) James W. Cortada, *A Bibliographic Guide to the History of Computer Applications, 1950-1990* (Westport, Conn., 1996) に基づくTomaykoの分析。

Tomaykoの講演「工学としてのソフトウェア」

James E. Tomaykoは現在Carnegie Mellon大学でソフトウェア工学の研究と教育に従事しているが，ソフトウェア技術者として長年の経験を積む一方，技術史の学位を持ち，NASAにおけるコンピュータ利用の歴史に関する*Computers in Space Flight: the NASA Experience*と*Computers Take Flight: A History of NASA's Pioneering Digital Fly-by-Wire Project*の著者でもある。

ソフトウェア工学を他の伝統的な工学の分野と同列にみなせるのか？という懸念がTomaykoの考察の出発点だ。そこでソフトウェア工学の工学としての側面をより深く理解するために，工学史研究一般に立ち戻って，そこにおける重要な主題に注目し，それらをソフトウェア工学の文脈に置いて考えてみる。それによってソフトウェアの工学としての特徴が把握できると同時に，ソフトウェア工学史研究のアジェンダが導かれるというわけだ。さて，技術史家Eugene S. FergusonやHenry Petroskiは「工学とは何か？」をめぐる議論に対して重要な一石を投じた[25]。中でも注目されるのは，工学は技芸としての特徴も科学としての特徴も併せ持つこと，工学の中でも設計という創造的仕事は技芸としての性格を持ち，視覚的で非言語的なプロセスが重要な役割を果たすこと（Ferguson），工学は成功と失敗の両方から知識を獲得するが，とりわけ失敗は工学の進歩に重要な役割を果たすこと(Petroski)，こうして様々なタイプの知識を長い間かかって集積し学問的に伝達しうる知識の形態をもつ実体として工学は形成されたこと等だ。そこでTomaykoは工学に関するこうした議論をモデルに，1)技芸として工学，2)応用科学として工学，3)工学における失敗の役割という3つの主題に沿ってソフトウェア工学の発展を考察する。

技芸としてのソフトウェア

技芸としての工学という考え方の核になるのは設計だ。Fergusonによれば，「不確実な世界の中で現実のものの設計が成功したなら，それはどんな場合にも科学よりはむしろ技芸に基づいている」[26]。それが証拠に工学におけるハンドブックは科学的知識の典型だが，ハンドブックや学校で学んだ知識が設計にはほとんど役立たないこともまた工学の常識だ。ところがソフトウェア工学ではハンドブックが盛んに出版されるようになったのは10年前位からであり，ハンドブックが少な過ぎることがソフトウェア技術者を困らせている。これは，ソフトウェア工学が専門学問分野としてまだ未熟な段階にあり，技芸としての性格を強く残していることの証拠だ。そこでソフトウェア工学における設計手法の発展を見ると，工学の設計においては技術者の直感，視覚的で非言語的なプロセスが重要な役割を果たすというFergusonの指摘がぴったり当てはまることがよくわかる。

ソフトウェア開発における設計の起源は，ケンブリッジのMaurice Wilkesチームによる世界初のコンピュータEDSAC研究のサブルーチンの考え方にまでさかのぼる。これが機能分割によるプログラム設計を目指した第一歩だった。次の飛躍は，高級言語と大規模プログラムの登場により，新しい設計ツールとして視覚化の手段を利用するようになったことだ。フローチャートや，データフロー図，IBMのハイポ図，ワー

25) Eugene S. Ferguson, *Engineering and the Mind's Eye*, MIT Press, 1992（邦訳：藤原良樹，砂田久吉訳『技術屋の心眼』，平凡社，1995）。Henry Petroski, *Design Paradigms*, Cambridge U.P., 1994（邦訳：中島秀人，綾野博之訳『橋はなぜ落ちたのか－設計の失敗学』，朝日新聞社，2001）；*To Engineer is Human : The Role of Failure in Successful Design*, St. Martin's Press, 1985（邦訳：北村美都穂訳『人は誰でもエンジニア－失敗はいかにして成功のもとになるか』，鹿島出版会，1988）。

26) Eugene S. Ferguson, 藤原良樹，砂田久吉訳，前掲書，p.249。

ニエ法などだ。構造や論理とフローの両方を同時に示すことで，機能分割がよりいっそう発展した。1970年代になると設計について意識的な考察がおこなわれるようになり，成果としてDavid Parnasの情報隠蔽の考え方，抽象データ型の研究，それに結合度と結束性の重要性が明らかになった。こうした成果がもとになって，1980年代のオブジェクト指向設計や1990年代のソフトウェアアーキテクチャの考え方が発展したのだ。

以上のような設計を核とするソフトウェア開発の歴史的流れの中で研究の手がかりとなるのは，視覚化の発展，段階的詳細化と機能分割からヒントを得た個別設計手法の発展，それにオブジェクトをめぐる考え方等だ。プログラム言語の歴史においては，言語の基礎にある設計思想（パラダイム）が重要だ。例えば，オブジェクトとC＋＋，データ抽象とAdaという具合にだ。こういう研究はまだないが，設計思想の源泉にまで迫るためには，膨大な技術文書の解析だけでなく，オーラルヒストリーやノート・日記類の資料発掘が必要だろう。

（応用）科学としてのソフトウェア工学

ソフトウェア工学が，伝統的な工学分野のように確固とした科学的を基礎を持つのかについては，異論もある。だが，どんな工学的知識も最初は実践によって獲得される。ソフトウェア工学の科学的基礎が不十分とされるのは，ソフトウェア工学がまだ成熟した学問分野ではないというだけのことだ。そうだとしてもソフトウェア工学において，科学が意識的に適用され成功を納めた事例が皆無というわけではなく，その場合の基礎科学は数学だ。

重要な事例として，クリーンルーム法の発展がある。クリーンルームとは，プログラム設計と静的テストのための正当性証明を基礎とする，大規模ソフトウェア開発のアプローチだ[27]。正当性証明とは，計算機プログラムの実行が正しい結果をもたらすことについての数学的証明のことだ。これをクリーンルームソフトウェア開発の数学的基礎として発展させたのが，クリーンルーム法の創始者の一人Harlan Millsだという。Millsは，プログラムの各モジュールを証明に従う数学的関数と考え，この考え方を発展させプログラム構造の証明を行なった。プログラム構造とは，「連接，選択，反復」構造のことであり，そういう構造があるという考えは1930年代の計算可能性の議論にまでさかのぼる。それに証明が行われる以前から，プログラマーはそういう構造のプログラムを書いていたし，プログラム言語もこの構造に基づいて設計されていた。だがMills達によって数学的基礎が確立し，プログラム構造の証明が発見され，プログラム構造が厳密に適用されるようになると，それまでに比べてソフトウェアのエラー発生率は減少し，ずっと高品質になった。今では正当性証明はクリーンルームのようなソフトウェア開発技法の普遍的な数学的基礎として確立し，証明可能なプログラム構造は，土木工学における材料強度に匹敵するソフトウェア工学の基本要素となった。だが，クリーンルーム法についての歴史的研究はまだない。科学としてのソフトウェアというこの主題はMahoneyの講演「科学としてのソフトウェア」の内容とも深く関わる。

ソフトウェア工学における失敗の役割

Petroskiによれば，工学とは失敗を通じて成功した設計の限界を見つけることだ。Thomas Kuhnのパラダイムを工学にあてはめると，パラダイムの利用は工学の成熟を示し，パラダイムが駄目になったとき新しい工学知識の獲得が

27）『ソフトウェア工学大辞典』，朝倉書店，1998年（原著：John Marciniak ed., *Encyclopedia of Software Engineering*, John Wiley & Sons, 1994），p.527。

28）日本でも最近，工学における失敗の研究が見直されるようになった。詳細は，畑村洋太郎『失敗学のすすめ』，講談社，2000年を参照せよ。

加速される。例えば，タコマ海峡橋が崩落しても，吊り橋設計のパラダイムは無傷だったが，風などの環境に対する橋の長さと橋床の厚さに関するそれまでの考え方は別の新しい考え方にとって代わられた。Petroskiは，失敗から新しい工学知識が獲得される事例を多数考察しだ[28]。

ソフトウェア工学でも失敗から学ぶ機会はいくらでもある。開発プロジェクトは失敗し，製品が「バグ」だらけで提供されるのは，珍しいことではない。だが，ソフトウェアのこの種の失敗はPetroskiが注目したタイプの失敗とは違う。ワープロ用ソフトウェアのバグは防ごうと思えば防げるが，経済的事情で許されないだけだ。Petroskiが書いたように，「大量生産の機械製品やエレクトロニクス製品の場合には，欠陥の是正や改良が，製品が消費者の手に渡ってからなされることが多い」[29]。航空機のような一品生産の製品が自動車会社の平均的製品に比べはるかに高品質なのと同じように，スペースシャトルのソフトウェアはワープロ用ソフトウェアほどに年中壊れたりしない。

研究に値する失敗のタイプとは，技術的可能性の限界を明らかにするものだ。土木工学史における橋の設計は失敗を経験すればするほど洗練された。何年もかかって石橋は吊り橋に変わり，橋の材質は石から鋼鉄に変わった。ソフトウェア工学でこれに相当する変化は高速のプロセッサーの登場だ。問題なのは，こうした変化がソフトウェア工学では土木工学と違って始終しかも急速に生じることだ。このためソフトウェア工学では失敗から学ぶことは難しい。もう１つ不利なのは，ソフトウェア工学が他の工学に比べて自然とのつながりが希薄で，自然的制約を認識する機会が少ないことだ。

そうではあってもソフトウェア工学史の研究において失敗というテーマは有用だ。異なる時代毎にどんな技術的制約があったかを理解し，学ぶべき教訓を理解する手がかりになる。失敗したソフトウェア開発プロジェクトの歴史的研究はソフトウェア工学にとっても重要だ。これまで大部分のプロジェクトは成功せず，製品が提供され実際に使えるようになるまでユーザーは堪え忍ばねばならなかった。スケジュールと予算超過という「ソフトウェア危機」，つまりソフトウェア開発の失敗の原因は何か？ ソフトウェア技術者は他の工学分野に比べて失敗を簡単に修正できるために却って失敗を繰り返すのか？ いずれにしても失敗したプロジェクトの歴史研究はまだない。

まとめ

以上の考察から工学史一般において議論されている主題は，ソフトウェア工学史においても有用な研究の指針となることがわかった。ソフトウェア工学を専門学問分野として早く確立させようとするあまり，ソフトウェア工学の技芸としての側面を否定しようとする動きもある。それに対して，ソフトウェア工学においてはハンドブック的科学よりは技芸が優位にあることを明快に示したのが，*The Mythical man-month*[30]の著者Frederick Brooksだ。Brooksによると，その時代の「主導」原理を使って委員会形式で開発されたシステム（COBOL，VMSオペレーティングシステム，Ada言語）は人々に好まれず，人々が好むシステム（C言語，Unixオペレーティングシステム，Java）は一人か二人の手で開発されたものだ。ソフトウェア開発に職人芸が有効な証拠だ。だが，一人の名人が開発したソフトウェアを歴史研究のテーマにしようとしても，職人芸の常で，実験科学でアイデア誕生を捉えたノートの類がソフトウェア工学の分野で保存されることはまずない。この場合オーラルヒストリーが唯一の手がかりだ。組織的な大規模ソフトウェア開発の場合には，プロジェクトの構想やメモの類まで記録は残されているが，とくに失敗したプロジェクト

29) Henry Petroski，北村美都穂訳，前掲書，p.32。

30) Frederick P. Brooks，前掲書。

の場合，新しい知見が得られたとしても，組織はそういう記録を公開したがらない。やはり組織を退職した技術者や管理者を相手にオーラルヒストリーに頼らざるを得ない。産業界には個々のプロジェクトで残された報告書類から「学べる教訓」を見つけ集積しようという構想もある。だが，企業はこうした情報を顧客にさえ公開したがらないので，失敗から学んだ知識も企業の私有物のままだ。もしこうした報告書類や情報にアクセスできれば，ソフトウェア工学にとってもソフトウェア工学史にとっても研究の基礎として有用なはずなのだが。

もう1つ歴史研究の有力な情報源に技術報告がある。ソフトウェア工学の専門家ではない歴史家にとって内容は難解だが，どう利用するかも問題だ。技術史において重要な出来事を考察するには技術的事項の理解は必須だ。問題となるのは，歴史家が技術の内容をどこまで詳しく解明し，そのうちどれだけを読者と共有するかだ。ここでTomayko自身の歴史家としての経験がものを言う。NASAにおける包括的なコンピュータ利用の歴史を書いた1冊目の著書*Computers in Space Flight*(1987)では技術の進化を制度史に劣らず主要なテーマと位置づけ，技術の詳細な説明に力を入れ過ぎたため，一般読者が理解するには難解な内容になった。10年後に2冊目の著書*Computer Takes Flight* (2000)を書くときには周囲の助言を聞き入れ，歴史的な出来事を解析するための基礎として技術を位置づけた。一般にコンピュータ史の分野では，技術の内容を容易に理解でき，自分達の歴史を書くという動機から，コンピュータの専門家が活躍してきた。だが専門家の歴史は技術過剰だ。専門家にも一般読者にも面白くかつ有用なバランスのとれた歴史が必要であり，そのために専門家ではない歴史家がソフトウェア工学史の分野に参加することが望まれる。

Endresのコメント

Stuttgart大学のAlbert Endresは，長年ソフトウェア開発に携わった現場の技術者の立場から，Tomaykoの提示した3つの主題の各々にコメントを加えた。コメントにはTomaykoのようなアメリカ人とは異なる見方が反映され興味深い。

ソフトウェア工学の技芸

設計と実際につくることをはっきり切り離せない点で，ソフトウェアは他の工学分野と違う。一般の工学では，エンジニアは設計に専念し，テクニシャンが実際につくるという役割分担ができている。エンジニアが設計に劣らず実際につくることにも関与すれば，ソフトウェアの場合に典型的にそうであるように，設計自体がずさんなものになることが多い。

工学と技芸は対立するわけではない。すぐれた技芸が発揮されることで設計は素晴らしいものになる。使い易いものが美しくないということもない。建築家と画家のグループ，バウハウスは日用品に「魂を吹き込む」ため芸術的形式を与えようとした。バウハウスの精神がソフトウェア，とくにウェッブ製品に生かされることを期待したい。

応用（コンピュータ）科学としての（ソフトウェア）工学

コンピュータ科学はポパーの科学の定義を満たしている。理論を確証する科学的説明の過程で，観測，実験，証明のいずれも利用するからだ。ソフトウェア工学で問題になるのは，実務家が必要とする（成功もしくは失敗した）実践の理論的説明をコンピュータ科学が提供しないことだ。ソフトウェアの科学的基礎としては数学と生物学が有望だ。数学は暗号やプログラム証明の自動化に現に役立っているし，生物学は人間の心の機能を理解することでヒューマン・インターフェースの設計の助けになる。

ハンドブックのような体系立てられた知識はソフトウェア工学にとって重要だ。実務家にはしっかりした基礎に基づいているという自信を

与えるし，時間とお金を節約してすぐれた解決法を提供する。それに教育にも役立つ。ソフトウェア設計でさえ科学的基礎を必要としている。コンピュータ科学が新し過ぎるのに加え，ハードウェア技術とシステム利用が加速度的に進展するために，コンピュータ科学への要請は過大なものとなり，分野全体を相手にできる専門家もいないし，専門家同士の交流も難しくなっている。だからといって，基礎科学によって工学が正当化されるとは思わない。科学的方法は自然科学の専有物ではなく，工学知識の体系化に役立てられている。それに，技術と工学は昔から科学より先行してきたし，ソフトウェア工学だってコンピュータ科学より先に無関係に発展してきたからだ。

（ソフトウェア）工学における失敗の役割

Windowsのようなパソコン用ソフトウェア製品は「大量生産」されているわけではない。一度だけ生産されると，あとは主要言語の数だけ増産され，複製され，製品として包装され，「大量販売」される。生産という点では，航空機など一品生産の工業製品に近いかも知れない。したがって，パソコン用ソフトウェア開発からでも，「現在の設計と方法の限界」を示す失敗の教訓を引出せる。例えば，Windows2000には６万のエラーがあるという予測が心配の種になっている。1000命令につき１つのエラーがあることになるからだ。だがこれは，Windows2000のような巨大システムに関するエラー予測の方法がまだ不完全で，「操作的プロフィール（operational profile)」についてもっと研究する必要があることを示すものだ。操作的プロフィールとは，テストのカバーする範囲がシステムのライフタイムにおける実際の利用とどう関係しているかを示すものであり，この助けを借りればWindows2000の利用者は100万回に１回以上のエラーを経験することはないとわかるのだ。

（ソフトウェア）工学史研究者への助言

新しい工学知識は価値あるもので，道端に転がっているわけではない。企業はいつでも新しい工学知識を独占しようとするし，技術者の知識は論文ではなく製品として表現される。工学知識を文書化するのは設計者や開発者ではなく，いろいろな設計を比較しようとする人々の仕事だ。ソフトウェア工学ではこういう仕事はほとんど行われていないが，実務家にも歴史家にも歓迎されるはずだ。

あらゆるソフトウェアをひとまとめにして考えることはできない。宇宙飛行，テキストエディター，学生用Java入門といったソフトウェアは全然違うものだ。あるタイプのソフトウェアを書いたからといって，別のタイプのソフトウェアの専門家とは言えない。航空機の保守管理マニュアル，ベストセラー小説，地下鉄の落書きを，書かれたものだからといって同列に比較できないのと同じことだ。そこにソフトウェア工学を教えることの難しさがある。

要するに，Endresのコメントは，現場から遊離し，やや抽象的に過ぎたTomaykoの講演を補うものだったと言える。

Seelyのコメント

Michigan工科大学のBruce E. Seelyは交通輸送業における技術者教育と技術者の活動の歴史を研究テーマとする技術史家だ。ソフトウェア工学は工学の一分野として認められていないというTomaykoの懸念に焦点を絞り，その根拠について検討した。Tomaykoが提示した伝統的な工学における３つの主題について他の工学分野と比較するだけでなく，専門職業としてのソフトウェア技術者の位置について考察した。以下その要点を見ていく。

Tomaykoの懸念は，工学とは有形で物理的な仕事に関係するとか，工学とは触ることのできるものを対象とするという工学の伝統的な考え方に原因がある。技術者は非言語的で視覚的

な流儀で働くと主張したFergusonは，それに加えて，物理的世界における経験こそ複雑なシステムを開発する際に技術者の取組みの成功を左右すると主張した。Tomaykoが言うようにソフトウェア設計と自然との間に緊密な関係がないとしたら，Fergusonの指摘は無視できない。

そこでまずTomaykoが提示した工学史における３つの主題について考察する。第一に応用科学としての工学について，工学は科学から大きく花開くというBannevar Bushの主張はまやかしだと技術史家は一貫して主張してきた。技術史においては明白な事実だからだ。だが，第２次世界大戦後，科学上の発見と技術革新とのつながりが強まると，技術者は科学への工学の従属を認めるようになった。そのためオペレーションズリサーチやシステム工学のような「手続き（procedure)」指向の学問は科学的基礎を持たないとして学問分野として軽視された。同じ「手続き」指向の性格をもつソフトウェア工学の場合はといえば，コンピュータ科学という立派な後ろ楯があるので，それらの学問分野と同列に論じるには無理がある。第２に工学における失敗の役割について，ソフトウェアにたくさんの失敗があるのは誰でも知っている。問題なのは，失敗に対するソフトウェア技術者の関係だ。Tomaykoが言うように，ハードウェアの変化が急速なためにソフトウェア設計者はあえてリスクを犯そうという意識が高いとか，ソフトウェアでは失敗の容易さが失敗を促進するというのであれば，ソフトウェア１ビットの誤りが橋やダムの失敗に劣らぬ破滅的結果につながるかも知れないのだから，恐ろしいことだ。厳密な完全主義者というステレオタイプの技術者像とソフトウェアの不完全な性能とにどう折合いをつけるのか，この点は大いに疑問が残る。第３に設計について，ソフトウェア開発者の設計の取組みは，システムを機能させるという困難に挑戦し，Fergusonが言うようにエレガントな解決を願っている点で他の工学と違わない。設計を，Fergusonのような単なる技芸ではなく，Bucciarelliのように「社会的過程」と見るならば[31]，ソフトウェア技術者が活動する環境にそっくりだ。Bucciarelliは事例研究として写真原版処理機の開発現場を克明に研究し，実物試験直前に処理時間が長過ぎる（30秒のはずが２分）ことを発見した開発チームの取組みを次のように生き生きと描いている。

> Markは15時間を費やした…コンピュータ操作卓の前に座り，対象となる世界に包み込まれて。手順に従ってコンピュータコードのブロックを通り抜け，各フェーズやセグメントの時間を設定し，不要な操作を見つけ出し，…必要な時間節約を達成するためセグメントを再構築する方法を見つけ出す。生活はビットとバイトの世界に凝縮され，そこでは，マシンコード１行の実行のような特徴的な出来事が100万分の１秒の間に生じ，1000分の１秒の割合で任意に選ばれる512のデテクターから生じるデータバイトが処理されるべく列をなして数を増し，そして処理を行なうコードはそのプログラムされた操作を通じて1000ピクセルのどれにも正しい灰色の色調を与えるために何百万回も走り，それらがみな一緒になって，モニターのスクリーン上に画像を生じるのだ…。
>
> Markは自分の診断的作業のために特別なツールを持っていた。ソフトウェア開発プログラムだ。それのおかげで，コードを１行ずつ走らせたり，命令の特定の部分集合が消費する「(実時間)」を測定するためにその部分を繰返し走らせたりすることができた。それらのツールによって，100万分の１秒のレベルまでマシンの応答時間に対する感覚を拡張する。それらのツールのおかげで，コードのすべての行を知ることができ，ある部分をまとめて除去し，プログラムを走らせ，さらに実行時間の差に気づく。100万分の１秒の目録を作るのだ。それらが彼を夢中にするのは，彼が対象としてのそれらの時間間隔を突き止め時間を節約するためだ。帆を巧みに操って前進するときのように前後に動きながら，画像を作り出す時間全体を短縮するため

31）Lois L. Bucciarelli, *Designing Engineers*, MIT Press, 1994。
32）同上，192-93.

に働く。建物の中に１人いて，暖房システムの騒音も時間が矢のように過ぎ去るのも気にならない。[32]

Bucciarelliの説明が教えてくれるのは，ソフトウェア技術者の仕事ぶりが伝統的な技術者と変わらないこと，それに重要なのは，ソフトウェアがいつでもマシンやプロセスの鍵を握っているという技術者の認識が物理的な現実世界とのつながりを提供するかも知れないことだ。

以上の考察は，ソフトウェア開発が工学的活動であることの証明にはなるが，しかしそれでも疑問が残るというのであれば，それはソフトウェア技術者の社会的評価，つまり専門職業化に問題があるのではないか。そうSeelyは考えた。コンピュータ史家のPaul Ceruzziによると，1968年のガルミッシュ会議（詳細は第３部参照）でソフトウェア工学がソフトウェア開発の理論的基礎として提唱されて以来，専門職業として分野への参入を規制したり，規範を設けたりする力をもたなかった。要するに専門職としてソフトウェア工学を確立する取組みは失敗したというのだ。これは重要な論点だ。伝統的な工学の分野では，社会的評価の向上を目標に専門職業組織を確立させてきたからだ。これに対してソフトウェア開発者は，伝統的工学のパターンを踏襲して，ソフトウェア技術者と呼ばれたいと考えているのか？ソフトウェア設計者の仕事に有形性が欠けていることが障害になるというのであれば，生産工学という注目すべき前例がある。生産工学は工場における組織化の活動と効率を重視する。生産物は書かれただけの情報やダイアグラム，組織図といったものが大部分であり，技術者の仕事が物理的なものの設計に直結しない点はソフトウェア技術者と同じだ。だが，生産工学は1900年以後機械工学から独立して発展し，1917年には専門家の組織として産業工学協会を樹立するまでになった。生産活動の効率を改善する科学的管理法などの技術に集まる関心をてこにし，Taylorの「科学的管理法」に見られるように科学的装いを求めた成果だ。ソフトウェア開発者と研究者の連携という点ではソフトウェア開発者David Parnasはこう述べた。研究者の仕事は開発者の問題を理解し，長期的視野にたって根源的な原因と根本的な治療法を探し出すことだ。締切日に追いまくられ強圧的な市場に応じねばならない開発者には長期的な開発策を探す余裕はない。こういう考え方は他の工学分野にも立派に通用する。

以上のようにSeelyのコメントは，ソフトウェア活動を工学と認めさせることができるかというTomaykoの懸念に答えようとしたものだった。だがSeelyのコメントは，第２部の締めくくりの討論においてCBIのNorbergから厳しい批判を受けた。専門職業化に関する考察自体は有意義だが，科学としてのソフトウェア（ソフトウェアの科学史）と対をなす要素である工学としてのソフトウェア（ソフトウェアの工学史）の議論にはふさわしくない。ソフトウェアの発展と急速な変化にこそ焦点をあてるべきなのだ。このNorbergの批判は，ソフトウェアの専門家ではない歴史家がソフトウェア工学史の分野に足を踏み入れることの難しさをあらためて認識させるものだ。

討論では他に，伝統的工学とソフトウェア工学における特許の問題，ソフトウェア工学と産業界との関係などが話題となった。

第３部　信頼できる人工物としてのソフトウェア

第３部のテーマは，ソフトウェアの信頼性の歴史に関する考察だ。第１部と第２部で論じられた，理論コンピュータ科学とソフトウェア工学がソフトウェアの開発や利用といった日常の実践にどこまで役立っているのかという問題を，MahoneyやTomaykoとは異なる社会的観点から考察する。コンピュータへの依存をますます深める現代社会にとって差し迫った問題だ。社会学者Donald MacKenzieの講演「Sonnenbichlからの眺め：ソフトウェアとシステムのデペンダビリティに関する歴史社会学について」はソ

フトウェア工学で有名なガルミッシュ会議を出発点にソフトウェア信頼性の歴史における研究課題を探求し，ドイツのコンピュータ科学者Bernd Mahrとアメリカのソフトウェア技術者Victoria Stavridouがコメントを加えた。

MacKenzieの講演「Sonnenbichlからの眺め：ソフトウェアとシステムのデペンダビリティに関する歴史社会学について」

Edinburgh大学社会学部のDonald MacKenzieは，技術の社会学，とくに技術の社会構成と呼ばれるグループの中でも今最も活発で注目を浴びている研究者の１人だ。

最初にMacKenzieの考察の前提になる用語の問題を見ておく。会議の主催者がMacKenzieに依頼したテーマは "Software as Reliable Artefact" であり，ソフトウェア工学の分野でもreliabilityを信頼性の意味に使う。だがMacKenzieは，IFIP Working Groupの用語法に関する研究成果[33]をもとにデペンダビリティ（dependability）を採用した。デペンダビリティは，アベイラビリティ（即利用可能性），信頼度（仕事の継続性），安全度（周囲の環境に致命的な結果をもたらすことのないようこれを回避），セキュリティ（無許可で情報をアクセスしたり取扱うことを防止）すべてを包含する用語だ。デペンダビリティに対応する阻害要因として障害（実行されるべき仕事の仕様から逸脱しているとき生じる。仕様はシステムの期待すべき機能および仕事についての合意文書），誤り（障害をもたらすシステムの状態の一部），フォールト（誤りのはっきりした，あるいはそのように推定された原因）がある。デペンダビリティを達成する手段は達成方法（フォールト予防とフォールトトレランス，仕様に定められた仕事を実行する能力をシステムに与える方法）と確証（フォールト除去とフォールト予測，仕様に示された仕事を実行するシステムの能力の範囲内で確実性を達成する方法）とに分かれる。これらの概念は，MacKenzieによると，ソフトウェアのデペンダビリティの歴史が直面する課題を研究する上で有用なガイドとなる（図5）。MacKenzieの考察の出発点は，「ソフトウェア危機」が診断され，解決策として「ソフトウェア工学」が提案された，有名な1968年のガルミッシュ会議だ。講演の表題にある 'Sonnenbichl' は南ドイツの景勝地ガルミッシュのホテル名で，会議の会場となった。会議では，ソフトウェアの生産性向上だけでなく，人間の安全がコンピュータシステムに頼ることになったときに予想される深刻な事態も話し合われた。その意味でMacKenzieは，この会議を，「ソフトウェア史における自己反省の明確な契機として，デペンダビリティの歴史に有用な出発点」と位置付けた。問題は，その後ソフトウェアのデペンダビリティの歴史はどう展開したかだ。考察の基礎となるのは、コンピュータが関係した可能性のある事故死の記録だ。総数約1100の事例を分析した結果を見ると，主要な原因（死の90％以上）は人間とコンピュータの相互作用における欠陥であり，ハードウェアの欠陥（フォールト）が約４％，ソフトウェアの「バグ（ソフトウェア設計の欠陥）」が３％強つまり30件だ。ソフトウェアが原因とされたもののうちに，放射線治療機や湾岸戦争中のパトリオット迎撃ミサイルシステムの事故も含まれる。要するに，ソフトウェア設計の欠陥による事故死はガルミッシュ会議で心配されたほどに多くない。

この分析結果から，ソフトウェアデペンダビリティの歴史にとって重要な研究課題が浮かび上がる。なぜ，大衆注視の中で，ソフトウェア設計の欠陥が明白に関与したような破滅的な事故はまだ起こっていないのか，という問題だ。MacKenzieは問題を「Hoareのパラドックス」と名づけた。問題を深く考察したのが，歴史家

33) Jean-Claude Laprie, ed., *Dependability : Basic Concepts and Terminology in English,French, German, Italian and Japanese* (*Vienna : Springer,* 1992)

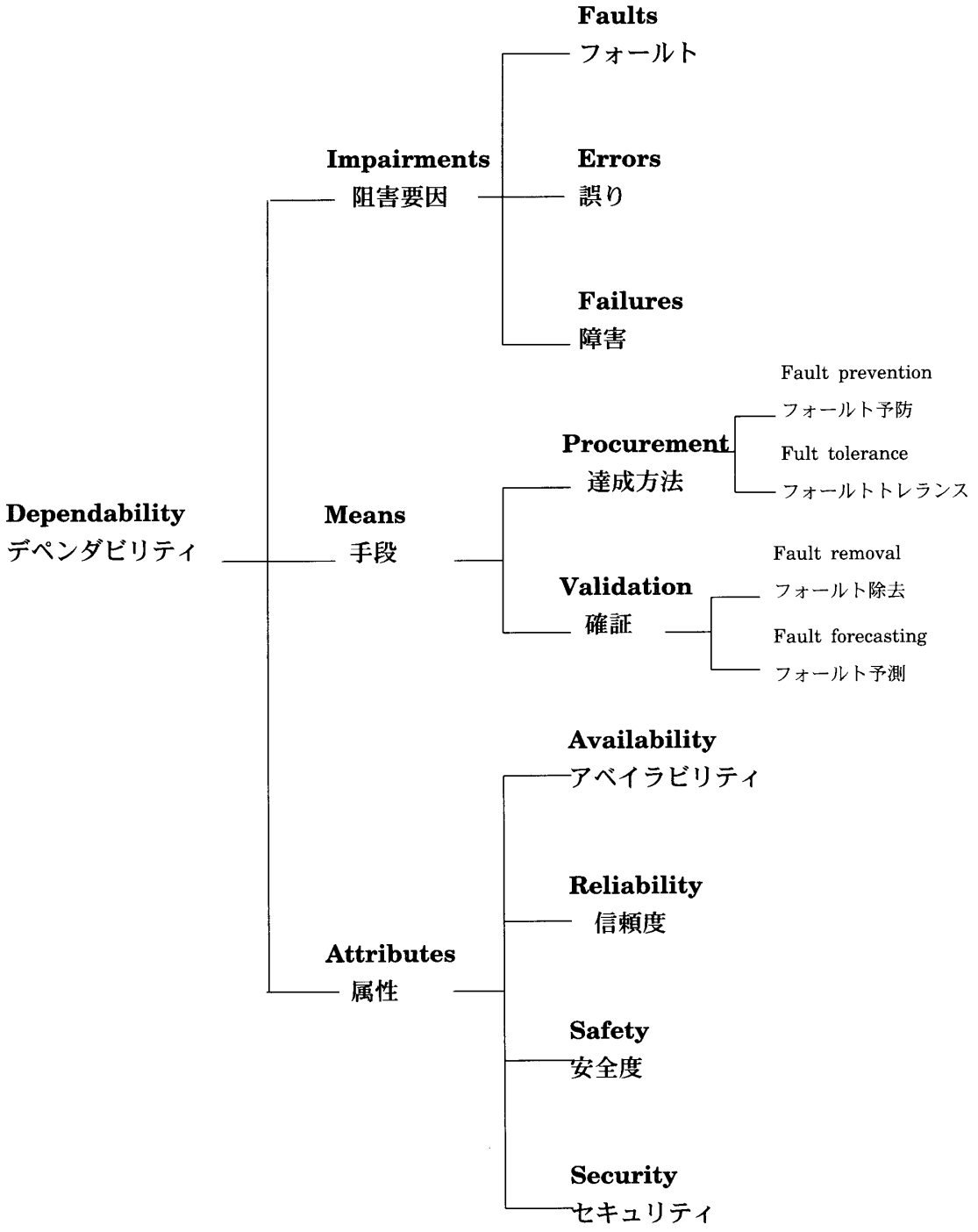


図5．デペンダビリティの関係図．J. C. Laprie, ed, Dependability（Vienna：Springer, 1992), P. 5 and P.104.

ではなくコンピュータ科学者のC. A. R. Hoareだからだ。Hoareは，ソフトウェアデペンダビリティを達成する手段はソフトウェアを形式的推論の支配下に置くことだという，ガルミッシュ会議の精神の強力な推進者であり，コンピュータ科学者の中で厳密で明確な思考を重んじる形式主義者の1人だ。論文「コンピュータプログラミングの公理論的基礎」（1969年）[34]で，自分の考えをこう説明した。「コンピュータプログラミングが精密科学であるのは，プログラムの性質全体と，所与の環境でプログラムを実行した結果全体とが，原則としてプログラムのテキスト自体から純粋に演繹的推論によって求まる点にある。」1つのプログラムが達成しようとすることについて，形式的仕様をつくることができ，関係するプログラム言語をうまく設計できれば，プログラムの「正しさ」を定理として表すことができ，定理を証明することができるはずだ。「プログラムの証明」（つまり「プログラムの正しさの証明」）は，「理論的探求」であると同時に，プログラム設計の欠陥を防ぎ，欠陥が存在しないという信頼を達成するためにとても重要な実地の技術であり，従ってデペンダビリティへの重要な貢献にもなる。これが，Hoareの信念だった。

ところが四半世紀後，ソフトウェアデペンダビリティの問題を再検討したHoareが認めたのは[35]，電話交換システムや航空機の飛行を制御する現代のソフトウェアがかつてのソフトウェアに比べてはるかに巨大でありながらはるかに信頼度を増していることだ。だが，このソフトウェアデペンダビリティの成功は，Hoareが主張してきた「プログラムの証明」のおかげではない。1990年代半ばになっても，そのような証明が産業界で実行されることはほとんどないからだ[36]。「ソフトウェアは証明なしにどうやってそれほどの信頼性を獲得したのか？」とHoareを悩ませた問題，それがMacKenzieの言う「Hoareのパラドックス」だ。証明なしでも現代のソフトウェア工学の実践が機能している理由についてHoareは考えた。コンピュータハードウェアの能力の増大と相対コストの減少のおかげで，プログラミングでかつて禁止されていたことができるようになり，誤りを予想した「防御的プログラミング」が実行され，設計とプログラムをそれらに直接責任を負わないチームメンバーに見直させることで誤りの検出に目覚しい成果を挙げるようになったことなどだ。

これに対して，「Hoareのパラドックス」を歴史家及び社会学者の目で見て，MacKenzieは，3つの研究課題を定式化した。第1に，デジタル電話交換やフライバイワイヤのような現実世界の信頼できるシステムの詳細な歴史を徹底的に調べることだ。それらのシステムでどんな欠陥が生じ，どれが誤りにつながり，誤りから障害が生じるのをどのようにして防いだか？システムの長い進化においてデペンダビリティはいかにして保たれ改善されたか？異なるシステムでデペンダビリティに優劣があるのはなぜか？を調べることだ。第2に，デペンダビリティを達成する手段についてもっと視野を広げる必要がある。ソフトウェアの証明とは別に，もっとありふれた防御的プログラミング，ソフトウェア（開発の）監査，リカバリーブロックの

34）C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, 12, 1969, pp. 576-83.

35）C. A. R. Hoare, "How Did Software Get So Reliable Without Proof?", Presentation to Awareness Club in Computer Assisted Formal Reasoning, Edinburgh, 21 March 1994 ; Hoare," How Did Software Get So Reliable Without Proof?", typescript, December 1995, p. 3.

36）例えば，Donald MacKenzie and M. Tierney, "Safety-Critical and Security-Critical Computing in Britain : An Exploration," *Technology Analysis & Strategic Management*, 8, 1996, pp. 355-79参照。

歴史，それにソフトウェアの見直しとテストの役割に注目すべきだ。それらは現実世界のシステムデペンダビリティに大きく役立った可能性が大きいからだ。第3に，デペンダビリティへの形式主義の影響を，単なる技術を越えたもっと広範なものと見るべきだ。ソフトウェアとはどのようなものでありあるべきかという考え方や文化の問題も含めてだ。MacKenzieが提起したこれらの問題は，コンピュータソフトウェア史はもちろん，コンピュータ史でも今までほとんどとり上げられてこなかった。

それでは，MacKenzie自身はこの問題にどう応えようとしているのか？それを知るよい手掛かりが，会議後の2001年出版された著書*Mechanizing Proof－Computing, Risk, and Trust*[37]だ。著書の主題はコンピュータを利用したソフトウェアの証明であり，演繹的方法という人類の歴史に深く根付いた文化的伝統がコンピュータの利用によってどう変化しようとしているのかを問題にしている。だが，表題からもわかるように，それだけに止まらず，問題を，コンピュータへの依存を深める現代社会にとって決定的な問題であるコンピュータデペンダビリティの問題と結びつけた所に，MacKenzieの優れた問題設定能力が示されている。著書では，この報告でも取上げた1960年代のソフトウェア危機とソフトウェア工学を始め，人工知能と自動化された定理証明，コンピュータを利用した数学の定理の証明（4色問題），米国国防総省の支援で発展したコンピュータセキュリティ研究，自動化されたソフトウェアの証明に対する批判，航空管制システムのプログラム証明のような実践における証明と現実世界の関係，機械化された証明の鍵を握る手段としての証明システムの発展やそういうシステムを証明に利用することの意味と問題点など，多彩な話題が取り上げられている。しかも文献のみならず関係者からの膨大なインタビューに裏付けられた，まさに現代コンピュータソフトウェア史の圧巻といえる。

ここで注目されるのは，最後の章でやはりコンピュータ関連事故の歴史とHoareのパラドックスを取上げていることだ。そしてHoareのパラドックスに対して社会学の立場から2つの説明を与えた。1つは，Hoareのような「道徳事業家」の活躍が功を奏して，ソフトウェアの危険を警告することで結果的にリスクを減らしてきたというものだ。これは，逸脱の社会学，とくに1960年以来のHoward Becker等のラベリング理論に基づいた説明だ[38]。ラベリング理論では，逸脱，つまり社会的規則違反のような行為を，その前提となる社会集団が規則をつくり執行することから考察する。「逸脱とは，人間の行為の性質ではなく，他者によってこの規則と制裁とが『違反者』に適用された結果なのである。逸脱者とは首尾よくこのレッテルを貼られた人間のことであり，また逸脱行為とはこのレッテルが貼られた行動のことである。」この理論で，行動に対する反応および行動自体にとって重要なのは，それが（「社会の聴衆」によって）逸脱とみなされるかどうかだ。MacKenzieによれば，これはコンピュータデペンダビリティの領域にもそのまま当てはまる。つまり，コンピュータデペンダビリティは絶対的な問題ではなく，どの程度の信頼で十分かとか，どんな出来事が障害とされるのかを判断するのは，コンピュータシステムの「聴衆」なのだ。MacKenzieが格好の事例としてとり上げたのは，1994年に起きたインテル製マイクロプロセッサーの「割り算バグ」事件だ。インテルは出荷前に

37） Donald MacKenzie, *Mechanizing Proof－Computing, Risk, and Trust*, MIT Press, 2001.

38） ハワード・S・ベッカー著，村上直之訳，『新装アウトサイダーズ－ラベリング理論とはなにか』，新泉社，1993年（原著：*Outsiders : Studies in theSociology of Deviance*, The Free Press, 1973）。とくに第1章「アウトサイダー」，第7章「規則とその執行」，第8章「道徳事業家」参照。

バグを検出していたが，深刻な障害を生み出す可能性は小さいと判断した。ところが，ある数学者がこの欠陥を発見し，欠陥とそれに対するインテルの態度をスキャンダルと考え，インターネットのニューズグループで社会に広く知らせた。すると，別の技術者がそれに応えて，普通のユーザーが自分のプロセッサーに欠陥があるかどうか簡単にチェックできる深刻な事例を考え出し，やはりインターネットを通じて警告した。おかげでうわさはインターネットを越えて，一般のマスコミに取り上げられるまでになり，インテルは製品の出荷停止と回収に加えて，株価低落と大きなダメージを受けることになった。ラベリング理論では，欠陥を告発した数学者や技術者を「道徳事業家」と呼ぶ。なぜなら「規則は自動的につくられるものでない。たとえ，ある種の活動が客観的な意味から集団にとって有害であるとしても，その害悪は発見され摘発される必要がある。人々は，何らかの手段を講じるべきことを思い知らされる。」コンピュータシステムのデペンダビリティというのは一般原理であって，例えばこの場合インテルとユーザーとで異なるように様々に解釈可能であり，具体的な規則が自動的に生み出されるわけではない。何者かが問題の事態に大衆の注意を喚起し，規則（この場合，どこまで欠陥を許容できるか）の創設に導いていかなければならない。彼等を道徳事業家という。なぜ「道徳」というのかは，社会の道徳体系，つまり社会の善悪の掟に関わる問題だからだ。こうした「道徳事業家精神」は，ガルミッシュ会議やHoareのようなコンピュータ科学者の活動，最近では2000年問題をめぐる動向を始め，ソフトウェアデペンダビリティの領域でもあちこちで見られ，重要な役割を果たしている。これがMacKenzieの説明だ。

Hoareのパラドックスに対するMacKenzieのもう1つの説明は，知識社会学[39]のアプローチによるものだ。MacKenzieによれば，2000年問題からも明らかになように，我々がコンピュータに望むことは，安全で確実であるだけでなく知ることなので，ソフトウェア信頼性は知識社会学の問題となる。知識社会学では，コンピュータのような人工物の性質に関する知識を3つに分類する[40]。第1に権威，つまり我々の信頼する人々が，それらの性質は何であるかを我々に話す。第2に帰納，つまり人工物やシステムをテストし利用することで，我々がそれらの性質を学ぶ。第3に演繹，つまり科学理論などから我々がそれらの性質を推測する。知識社会学の重要な鍵は，事物の性質に関する知識は人々の性質に関する知識と完全には分離できないということ，つまり知識はそれを生産したり所有する人間に属していると考えることにある，プログラムの証明の場合で言えば，演繹は人間に簡単に理解できる短い推論の連鎖ではなく，自動化された検証システムに頼ることが多い。従って，知識の生産と所有はシステム（とそれを設計した人間）に移るので，その場合に知識の性格が変わるのかというのがMacKenzieの著書の主題だ。だが，ソフトウェアの開発・生産のような社会—技術的実践においては，コンピュータシステムの性質に関する知識のうち，帰納と権威に基づいた形式の方がより効果的であるように思われる，それがパラドックスに対するMacKenzieの第2の説明だ。

こうしたMacKenzieの説明には，不十分な所もあるかもしれない。だが，重要なことは，コンピュータソフトウェアの信頼性のように，

39) MacKenzieが挙げた科学の知識社会学の入門書として，Barry Barnes, David Bloor, and John Henry, *Scientific Knowledge : A Sociological Analysis*, Chicago U.P., 1996.

40) Donald MacKenzie, "How Do We know the Properties of Artefacts? Applying the Sociology of Knowledge to Technology," *Technological Change : Methods and Themes in the History of Technology*, ed. Robert Fox, Harwood, 1996, pp. 247-63.

一見技術的に見える問題においても，様々な社会的ファクターや人間的ファクターが技術と複雑に相互作用していることだ。それを象徴的に示す事例が，MacKenzieの提示したHoareのパラドックスだ。これを，第1部及び第2部と併せて見ると，こう言えるのではないか。すなわち，理論コンピュータ科学やソフトウェア工学についてこれまでの科学や技術の閉じた概念によって，ソフトウェアの歴史を構築することはできない。もっと広範な社会的，政治的，経済的，文化的背景の中で問題をより総合的かつ多面的にとらえなければならないという，歴史研究の新しい方向を指し示すものだ。

Mahrのコメント

ベルリン工科大学のBernd Mahrは，1968年のガルミッシュ会議をMacKenzieがソフトウェアとシステムデペンダビリティの歴史的および社会学的考察の出発点としたことを，ソフトウェアに対する見方を狭め，開発や保守という商業的ファクターを無視するという危険をはらむもの，と批判した。それらのファクターはソフトウェアデペンダビリティを抑制も妨げもするだけでなく，ソフトウェアの様々なタイプ分け，それに応じたデペンダビリティの現象とそれを達成する方法をも左右するからだ。Mahrがもう1つ批判したのは，MacKenzieがデペンダビリティの考察の手がかりとしたLaprieの存在論のツリー構造だけではデペンダビリティという用語の背景に隠された方法論的内容は明らかにならないということだ。その代わりに，ソフトウェア工学におけるソフトウェアのライフサイクルモデルと商業環境，それに利用の文脈を考慮した，用語間の関係を示す自分の図式（図6）を提示した。さらに，MacKenzieが提起したHoareのパラドックスに答えるには，文化的，社会学的，経済的議論を基礎にする必要があり，Mahrから見て，MacKenzieの説明には決定的と思われる多数の論点が欠けている。歴史研究のためにも，ソフトウェアライフサイクルの諸局面にもっと注目する必要がある。テスト，理論とプログラミングスタイルに加えて，ソフトウェアライフサイクルは開発過程，契約者，開発者及びユーザーの関係，それに保守と利用の過程から強い影響を受け，こうした過程を適切に管理できたかどうかもまた成功・不成功の原因となったからだ。ソフトウェア及びシステムデペンダビリティの社会学という点では，MacKenzieが考察した逸脱，道徳事業家精神，知識と専門職業といった論点以外に，労働，組織，経済，市場及び消費者の振舞いなどもっと広範に考察されるべきだと，Mahrは指摘した。

Stavridouのコメント

シリコンバレーにあるシステム設計研究所のBictoria Stavridouは，現場のソフトウェア技術者としての経験に基づいて，デペダビリティの現状と将来について生き生きと語った。

まず，MacKenzieがLaprieに依拠して提示したデペンダビリティの概念自体進化しつつあることを具体例を挙げ説明した。例えば，MacKenzieが指摘したようなセキュリティ障害と安全度障害の違いは曖昧になりつつある。2000年春の米国の電子商取引サイトへのサイバーテロのように，国防のような特殊な分野だけでなく，セキュリティ一般への社会的関心も増す中で，両者の技術は融合し，欠陥があってもシステムの正常な運転継続を可能にするフォールトトレーランスのように，新しいアイデアに基づく強力な技術も開発されつつあるというのだ。この指摘を裏返せば，MacKenzieのように軍事技術にばかり注目しても，技術の全体像や問題全体を見そこなう危険を示唆するものだ。

次にStavridouが話題にしたのは，MacKenzieが取上げた30年前のガルミッシュ会議どころか，つい昨年，米国のコンピュータ科学者のグループが，埋め込みソフトウェアに基づく部品の領域で連合を結成し，そのような装置内部のソフトウェアを正しく理解することはハードウェアに比べてずっと難しいということを議論する

ことにしたというものだ。そこで，ソフトウェアを正しく理解し，信頼性を持たせることが，なぜそれほどまでに難しいのか？という理由を３つ挙げた。第１に，ソフトウェアがはるかに高度な信頼度を要求されることだ。例えば，エビオニックス（航空・宇宙・ミサイル用電子機器に関する電子工学）に利用されるソフトウェアにおいて危険な障害が数百万年にせいぜい１回といった高度の信頼度を達成することは可能でも，その信頼度をテストのような方法では測定することも，相手を納得させる根拠も構築できない。第２の理由は，ソフトウェアは複雑さの宝庫であることだ。第３に，ハードウェアが何であり，ソフトウェアが何であるかという境界は，過去も今も変化し続けていることだ。ソフトウェアの複雑さの恩恵を被っている側面もある。関連して，ソフトウェアが実現するものの複雑さと，コンピュータ関連の事故死の90％が御粗末なヒューマンーコンピュータインターフェースのせいだというMacKenzieの結論とを併せ考慮すれば，ヒューマンーコンピュータインターフェースもまたソフトウェアデペンダビリティの本質的な構成要素と言えることを認めた。この点では，MacKenzieの歴史社会学的分析はコンピュータの専門家を十分説得できるほど優れたものであったと評価できる。

さらに，「Hoareのパラドックス」に自分なりの解釈を加えるために，Stavridouが披露したのは，90年代初期の仕事でベテランの彼女が新米のような初歩的な失敗をした実に興味深い経験談だった。彼女は埋めこみソフトの専門家であるらしく，仕事は20年近くも使われていた有名な戦闘機のソフトウェア安全性証明であり，形式的検証技術が基本だった。だが，実際の作業の中で彼女が思い知らされたのは，戦闘機という「作動装置やセンサー，連動装置を持つ現実のシステムの文脈中に置かれることでソフトウェアは生き返り，そこで初めて安全度のような属性をも獲得する」という初歩的原則だった。そのことを忘れると，形式的検証技術を基本とする安全性証明のような仕事でもうまく行かない。ソフトウェアはそれだけでは何もできない，現実のシステム（コンピュータはもちろんそれが埋めこまれたシステム全体）の中に置かれてこそ生きて働くものであり，デペンダビリティや安全度のような属性もそのような環境の中でこそ実現されるという，ともすれば専門家でさえ見失いがちなソフトウェアについての基本的な理解を，彼女の体験談は教えてくれた。

Stavridouによると，システムのことを忘れて重大な結果を引起こしたのが，EUのロケットAriane５の処女飛行だ。旧型のAriane４で，ロケットの方向や姿勢，速度を自動制御するコンピュータシステムに使って成功した１組のコードが，最小限の変更を加えられた上で，Ariane５の制御用コンピュータに再利用された。ソフトウェアの再利用は珍しいことではない。だが，発射から40秒でAriane５は爆発してしまった。原因は，Ariane５が旧型のAriane４に比べ５倍もスピードが速かったことだ。旧型のAriane４ではとてもうまく機能していたコードが，もともと設計されたときに比べ５倍も高速のAriane５の制御システムに組込まれてトラブルを起こしたというわけだ。これは，ソフトウェアに限らず，システムのスケールアップによくある出来事ではないか。第２部で議論された工学における失敗のうち，ソフトウェアに典型的な事例として実に興味深い。

ソフトウェア工学の共同体は形式的方法に無理解だというHoareの嘆きに対しても，Stavridouは現場の技術者らしい的確な解釈を示した。どのソフトウェア技術者にもプログラムを書くたびに基本的な数学を使うことを期待するのは，土木技術者に橋を設計するのに力学の第１原理に立ち戻れと要求するに等しい。どちらの場合にも，手近な問題に適用可能な工学原理と公式として表現された知識とからなる確立された基盤があるのだ。その基盤を利用することで，日常の実践でも構築の基本原理に立ち戻るという

専門家の重荷は除去される。形式的方法は，プログラム言語やコンパイラーのような基盤を建設するときにはいつでも威力を発揮し，目覚しい成功を収めた。おかげで，それらの基盤を利用するときも形式言語理論のような基礎のことを忘れていられる。だが，証明のような形式的方法が実際的な問題に採用される場合もある。例えば，民間航空機用のフライバイワイヤのシステムは複雑さのため，通常のテストとデバッグでは手に負えないので形式的方法が採用される。他方，現在利用されている埋めこみソフトウェアの98％は，洗濯機，ストーブ，腕時計といった装置の中にあり，こうしたアプリケーションにまで形式的方法を適用する必要があるか否かに議論の余地はない。確かに，ソフトウェは今日までHoareの考えたような完全な信頼度を獲得したことはなかった。間違いなく世界一広く普及したオペレーティングシステムであるWindowsがその良い例だ。だが，安全が決定的なソフトウェアをつくるとき，難しい問題に取組むとき，人々は良心的で慎重であり，そのため形式的推論を利用する。それに，コンパイラのような日常の道具を使うとき，人々は無意識に（最良のやり方だと思う）形式的基礎を利用しているのだ。

最後に，Stavridouは，今日我々が恩恵を被っている基盤はポストPC時代になればもはや我々の取組みを支援できないことについて考察した。低価格で強力なマイクロプロセッサーの増殖，万能ネットワークの成長，それに企業と投資家に対して新しい成長市場に目を向けるよう強いる既存のIT市場の飽和状態に焚き付けられて，コンピューティングがあらゆる場面に浸透することは現実となりつつある。この新しいユビキタスコンピューティングの世界では，スーパーコンピュータ，PC，それに日用品に組込まれた小さなマイクロプロセツサーのような多様な装置のダイナミックな連合が，コンピューティング利用の手段としての現在のPCにとって代わる。この世界では，多様なコンピューティングを装置に埋めこむことになり，分散的な情報基盤への大規模に分散したアクセスが必要とされ，これが未来のコンピューティングシステムのアーキテクチャになろう。このようなシステムを機能させるために必要なソフトウェアとは，こうした連合を創り出し仕事を課し管理する，騒々しく誤りを免れない構成要素の集合の知覚と振舞いを一貫して監督する，そしてセキュリティ，安全及びプライバシーを危険にさらさない信頼できる方法でそれを行なうソフトウェアだ。この世界のためのソフトウェアを今日の我々がつくれないことは明白な事実であり，「ソフトウェア危機」が再び我々に襲い掛かることになろう。技術革新のおかげで，今日，1960年代の「ソフトウェア危機」はある程度克服され，デペンダビリティも大いに達成された。だが，今後も不可避的に進行する技術革新のせいで，現在の技術では実現できない高度なデペンダビリティが社会から要請される時代が間もなく到来する，それがStavridouの結論だ。

MacKenzieの講演と２人のソフトウェア専門家のコメントを比較してみると，社会的関心と技術的関心とがダイナミックに交叉する場面もあったが，両者の間にギャップを感じないわけにはいかない。Mackenzieが提示した，逸脱の社会学や知識社会学のアプローチは，確かにデペンダビリティの基礎にある信頼の問題やデペンダビリティに対する社会の反応など，これまでのコンピュータ史にはない深い銅察を与えてくれる。だが，２人のコメンテータと比較すると，現実を知らないだけでなく，方法論に重きを置くあまり，現実に即して考える姿勢が希薄なのではないかと思わせる所がある。既成の研究方法に現実を合わせるようなやり方では，現実から遊離した抽象的で観念的な結論しか引き出せないのではないかと危惧を抱かせる。それで広範で長期的な視野から問題を見るという歴史家や社会学者のソフトウェア専門家に対する相対的優位が活かせるのか？　また，

MacKenzieの提唱するデペンダビリティの概念だけで問題が十分考察できるわけでもない。締めくくりの討論では，ユーザーのデペンダビリティの認識の問題として，「信頼（trust)」が議論の中心になった。ソフトウェアは，社会的信頼を失った核燃料や遺伝子操作のような技術に比べれば，信頼されている。信頼はどのように達成されるかと言えば，どんなリスクをとるか人々が選択できれば，人々に決定が強制されるよりは信頼は増す。安全が決定的なシステムにとって信頼は不可欠だと，MacKenzieは認めた。また，デペンダビリティ向上のための形式的方法の役割も議論になった。いずれにしてもMacKenzieも強調するように，現実に即した体系的な研究のよりいっそうの積み重ねと，そのための歴史家と社会学者の協力が求められている。MacKenzieが提起した「ソフトウェアとシステムのデペンダビリティに関する」歴史的考察はソフトウェア専門家にも歴史家にも面白く刺激的だったことは間違いない。

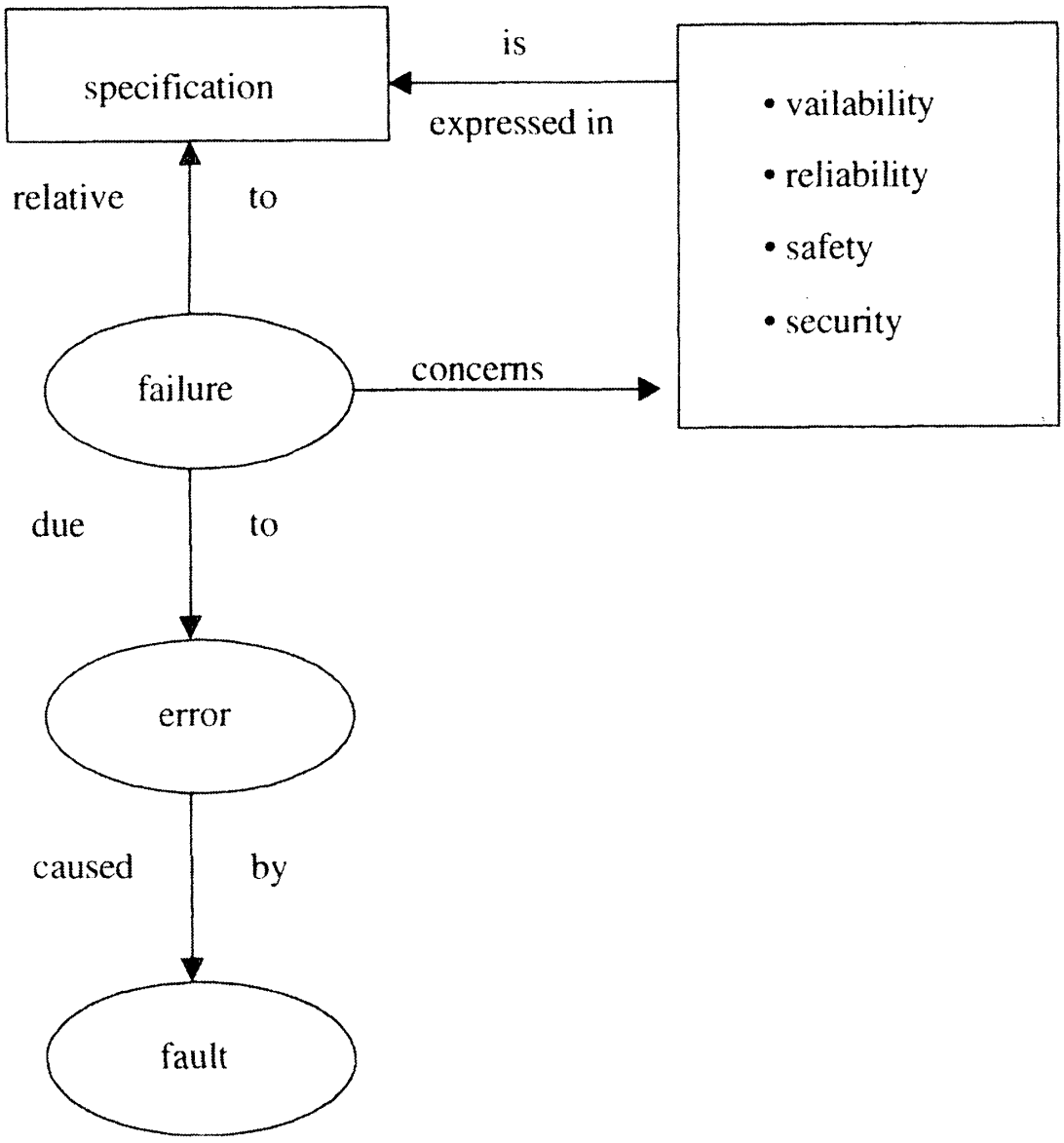


図６．Mahr提唱のfailure分類関係図（P.26参照）