ソフトウェア工学との類似性に着目した立法支援方法（一）

角田篤泰

第一章　はじめに
第一節　概要
第二節　背景
第二章　ソフトウェア工学の応用例
第一節　要求分析
第二節　仕様記述
第三節　設計方法論
第四節　テキスト処理
第五節　バージョン管理
第六節　再利用
第七節　簡易開発
第三章　ソフトウェア開発と立法過程

第一節　プログラミング
第二節　ソフトウェア開発過程
一　ソフトウェア開発概要
二　要求分析
三　設計
四　製作（コーディング）
五　試験
六　提供
七　保守
第三節　立法過程との比較と類似性
第四節　非落水型の開発方法
第四章　要求分析—政策の検討と立法事実の収集
第五章　設計—要綱の作成
第六章　製作—法令・条例案の作成
第七章　試験—意見収集と議会審議
第八章　提供と保守—公布・施行・改廃
第九章　立法支援システムと政策知識ベース
第十章　まとめ

（以上本号）

第一章　はじめに

第一節　概要

本稿では、ソフトウェア工学の知見を法令作成の方法論に取り入れて、システム化を行うことを提案する。法令作成も一種の「もの作り」であり、そのアウトプットは法令を表現した条文という文字列（「テキスト」と呼ぶ）であることから、客観的な作業的側面に着目すると、プログラムとして表現された文字列をアウトプットとするソフトウェア開発と重なる部分が多いこと、さらに、筆者の担当する政策立案の実習教育の経験の中で、ソフトウェア工学と同様のスキルを教授するケースに何度も直面したことから、両者の類似性に着目するという着想に至った。実際、第三章の図3のように作業プロセスも対応している。なお、本稿では狭義の法令だけでなく、条例等の例規を含めて「法令」と表現することにする。

現在、多くの自治体では業務の無駄を省く努力、さらに合理化や効率化の努力がなされ、例えば、条例の扱いに関しては、条例データベース・システムを用いて法制執務を行うことが普及している。少なくとも、システム化自体を目指す流れやモチベーションは広く存在していると言えよう。しかしながら、法制執務のような立法の現場の作業では、アナログでアドホックなものや、ベテラン職員の職人的技能に依存しているケースがほとんどであり、書類のデジタル化やデータベース導入程度のシステム化はなされていても、立法過程の作業の本質に迫るシステム化に至っているとは言い難い。そこで、本稿では、立法の本質部分である「もの作り」に対する方法論としての手法や支援技術を提案する。この方法論に付随するツール群も、ソフトウェア開発の現場で実際に利用されているよ

うなものを法制執務用に、言わば「翻訳」して用いる。

本稿で示される方法論については、行政や教育の現場の作業に貢献できることが最も重要な意義であるが、その他にも強調しておきたい意義として、「法情報学」の再構築に貢献できることを指摘しておきたい。従来、法情報学の分野で採り上げられてきたテーマは、司法分野におけるリーガル・リサーチやその支援に関するものがほとんどであり、立法過程の方法論に踏み込んだものはほとんど存在しない。しかしながら、情報科学のほとんどがソフトウェアやデータ構造の開発をともなう研究である点を勘案すると、「法」の「情報学」と呼ぶからには、このような研究成果が取り入れられて然るべきである。情報科学の分野が、開発をともなうクリエイティブな分野の知見を前提にした学問分野なのであるから、法情報学においても、立法過程のような物事の製作に関わる分野にもっと着目すべきであるし、そもそも司法の分野においても、調査のような受動的な情報処理だけではなく、立証構造を組み上げたり、法廷戦術をプランニングしたりするような能動的でクリエイティブなプロセスとして捉え直した上で取り扱うことで、実用的な知見も得られると考えている。そのような法情報学のあり方に対する新たな仮説を検証していくための最初のステップとしても、本稿の意義がある。この点については第二節でも議論する。

本稿と密接な関連研究としては、本稿と同様にソフトウェア開発と立法過程との類似性に着目した工学分野向けの拙著研究報告[1]と、法律事項を中心とした論理構造の視点から条文を組み立てる方法論を示した拙著論説[2]がある。これらは、情報科学的、あるいは数学的な理論的側面から記されたものである。これらの先行研究に対し、本稿では実践的な側面から工学的手法を取り入れた方法論の提案を行っている。

本稿では、まず、本章第二節で、背景について示した後、第二章において、ソフトウェア工学の知見を取り入れることで、期待できそうな具体的な利点、あるいは、そもそもどのような点が類似しているのかという点について、

簡単にその期待される応用例や類似点を紹介する。次に第三章において、ソフトウェア作成の概要を導入的説明や立法過程との類似点とともに記す。第四章から第八章までは、立法過程やソフトウェア開発の手順に沿って、順に各作業局面（「フェーズ」とも呼ぶ）でどのような方法論の導入が可能であるか議論する。第九章では、これらを想定した統合環境を実現する試作システム（eLe: e-Legislation Environment）や政策知識の表現方法について説明する。[(3)] 最後に第十章において、本稿のまとめを行う。

なお、本稿で行われる提案に当たっては、法制執務の現場の方々のヒアリングも行っており、自治体の担当の方へのインタビュー、自治体に条例管理システムを提供している業者の開発担当の方へのインタビュー、および中央省庁の方に行って頂いたレクチャーなどを基に考察を行った。[(4)] ご協力頂いた方々には、感謝の意を表したい。

第二節　背景

本研究を開始した初期の動機については、現実的な要請として、法学教育の現場において、立法過程に関する実習科目を運営するための方法論や支援システムが必要であったため、立法作業の内容を調査し、システマティックに整理しようと試みた際に、いくつかの問題点が見つかったことにあった。さらに、時を同じくして、訴訟の実習科目を運営するためにも同様のニーズがあり、本研究と同時並行で研究・開発を進めてきた。もちろん、その延長線上には、教育に限らず、立法と司法の両分野のそれぞれの現場作業支援に役立つ方法論やシステムの提供を目指すという目標があった。いずれにせよ、実学的な観点からの本研究の位置付けである。

一方、より学問的な面でも、本研究をある大きな研究の流れの一部として位置付けている。それは、前述のよう

な実習科目は、いずれも「法情報学」の科目の一つとしてそれぞれ開講されているものであるが、この「法情報学」の学問的な再構築や再定義に関わる問題意識も本研究の重要な動機となっている。そこで、以下に本稿の趣旨と大きく外れない範囲で、この「法情報学」の捉え方を示す。

法情報学は、従来、リーガル・リサーチを中心とした、法令や判例の検索を始めとする各種調査やそのまとめ方の実学的技能を学ぶ分野として多くの人々に認知されてきた。そこでは、法学の領域と情報を扱う学問領域が密接に結び付いていた訳ではない。しかしながら、本研究で最終目標としたい法情報学とは、情報学的な分野とさらに深く結び付けることで、様々な有効な知見と方法論を示すような分野である。

まず、法を取り巻く現象に着目して、法と情報との結び付きを考えてみる。結果的に言えば、法学が扱う事象の多くは問題解決的な側面が強いため、自然科学よりも工学に近く、その操作対象は言語的な情報がほとんどである、という点から、法が情報学と結び付くことを提唱したい。順に述べると、法学の扱う分野や現象は、法の役割を考えれば、その作り方や守ること自体が実質的に重要なはずである。しかしながら、多くの法学の現場では、裁判を中心とする司法の分野に重く力点が置かれ、実定法の解釈論が最も重要視されている。それ自体に異議を唱えるつもりはないが、それ以外の部分の方が、多くの一般の人々に共有されており、解釈論の部分は専門性の高い話であり、一般の人々の前に顕在化するのは、訴訟やそれに順ずるトラブルに発展した時だけである。最近盛んな「法教育」の理念や司法制度改革の理念を掲げるまでもなく、元々多くの人々にとって、日常的にも、法を守ること、あるいは、法を知り、活用するということは必須であり、専門家の専権事項ではない。さらに、法を作る立法の場面でも、このような歪みがある。それは、司法の場では、裁判官や弁護士など高度の専門家が活躍するのに対して、立法の場には、そのような専門家がほとんど不在なことである。特に小さな自治体で条例を作成する時には、例え

ば、昨年まで土木課に所属して外回りをしていた職員が今年はいきなり法務を扱う部署に異動させられて、条例作成に携わることもある。つまり、もっと立法分野においても、できれば一般の人々の間においても、法に関わる実用的な側面に目を向ける必要がある。そこで、法を守ること、法を知り、活用すること、さらに、法を作ることに目を向けると、それらのプロセスは、情報の発信・収集・加工という作業が本質になっている。さらに、法曹の専門家が活躍する分野でもこれらが必要な局面がある。それは、実際の裁判では、法解釈論が展開される訳ではなく、多くは要件事実の立証に費やされるのであり、立証とは、議論を言語的に組み立てるという、かなりクリエイティブな作業である。このように法を取り巻く現象を観察すると、法的現象の解釈としての自然科学的アプローチを採るよりも、「もの作り」的、すなわち、工学的な学問として捉えること、さらに、言語情報を扱うという特徴に着目すると、情報の取扱いを中心とした工学的作業として考えることも有効であると思われるのである。なお、工学的に捉えるには、情報科学的な知見だけでなく、システム科学一般で用いられる理論やツール群、あるいは、ゲーム理論を始めとする数学や経済学系の分野の理論なども導入した方が良い局面に数多く遭遇する。これらも法情報学の対象として適宜射程圏内に入れる必要はある。

次に、法と情報との本質的な関わりにも簡単に言及しておく。このような観点は「法と情報」のようなテーマでしばしば議論されることであるので、ここでは、本稿の後の部分と関わるものだけ示しておく。まず、本稿の趣旨から先に強調しておきたい点は、①「法＝プログラム」という視点での本質性である。これは、法はコンピュータの世界で喩えるなら、動作の指示が書かれたプログラムに相当する、という見方である。これは本稿でも採用する見方である。この他にも見方はあり、本稿の趣旨から離れ、哲学的な議論に発展するため、多くを述べないが、例えば②「法＝情報」という観点もある。これは「法＝無体物＝情報」としても良いかも知れない。さらに、③「法

の取り扱うものの関心が実体的な存在から情報的存在（無体物）に移っていく」という見方でも本質的な関連性の指摘になるかも知れない。これは、法の扱うものが実は「物（もの）」的な存在ではなく、「事（こと）」的な存在である、という考え方である。元々法が規制するのは人々の行為であり、実体そのものではない。実体でないものは、今日的には大抵情報に還元して議論することができる。そもそも、量子力学などの先端の自然科学分野でも、今日、「物質自体を構成する原子が物として実在している」と考えている物理学者は皆無であろう。物質（を構成する量子）は空間の特殊な「状態」と考えられている。日常的な直観とは異なるであろうが、これが現在の真実である。実際、我々の生活を支える原子力発電もこの理論から生まれ、太陽エネルギーもこの理論によって解釈されるのである。例えば、十九世紀には蝋燭の炎は物質だと考えられていたが、二十世紀には酸化還元現象の見え方だと判明し、このことは現在では周知されている。これと同様のことが二十世紀には物質存在の根本に及んだと考えて頂けば良い。同様に、法の対象はそもそも行為であるというだけでなく、さらに、一見「もの」的な対象でも、その究極は「こと」的な対象なのである。この観点からは、例えば、物権の排他性のような考え方は、人間の日常的直観を補足するには便利で実用的ではあるが、必ずしも事物の本当の姿を捉えている訳ではない。所詮、便宜的な考え方なので、必要に応じて解釈を変えないと、いくらでも問題が生じてしまう類いの考え方である。権利とは観念的に操作される情報的存在であるから、コピー可能であり、元々排他性のないものである。それを法律で排他性があるかのごとく擬制しているだけであるから、本当に排他性があると信じてしまう考え方は危険である。著作権や個人情報の考え方にも似たような問題が内在している。いずれにせよ、こうして、先に示した②や③の見解は、本稿の趣旨を離れ、やや大袈裟な議論になってしまうが、法の対象がそもそも情報であるという見地からのものであり、その観点からも情報が法の本質に密接に関わっていると言える。

以上のように法と情報の密切な関連性を踏まえ、法情報学を捉え直す時、本研究がその一部を支えるものになっているると言えるであろう。なぜなら、法令をプログラムや情報の一種と考え、その作成・加工・運用として、それらを扱うことを目指した提案が本研究だからである。

第二章　ソフトウェア工学の応用例

本章では、次章以降で提案するソフトウェア工学の応用について、先に、立法過程に応用可能な局面をいくつか採り上げ、その例を示す。これらによって、本提案により期待される効果の概要を直観的に示したい。

立法過程とソフトウェア開発の類似点の一つは、本稿冒頭で示したように、そのアウトプットとして、文字で記述された一種の文書を作成するという点である。しかしながら、小説や論文を作成する場合とは異なり、法令もソフトウェアも、ともに、その文に従ったシステム（社会あるいはコンピュータ）の振る舞いに本来の目的があり、その記述形式には詳細な制約がある。このような点でも類似していると言える。さらに、第三章でも詳細に示すが、法令もソフトウェアも、同様のプロセスを経て作成される点でも類似している。そのプロセスは順に、その目的や要望を分析し、設計図と言える要綱や仕様書を記述し、その意図に従って特定書式のアウトプット（法令案あるいはプログラム）を作成する工程が中心であり、その後、ソフトウェアにおける試験工程に対応して、立法過程でも、議会に図ったり、パブリック・コメントを集めたりする局面がある。こうして作成されたものがリリース（立法の場合は公布）され、実施される。さらに、このように一旦作成されて運用が始まった後でも、通常は、ソフトウェ

ア保守のように、法令の改廃の作業が付随している。特に、法令の附則に記されるようなバージョン管理はソフトウェアの保守プロセスでも中心的なテーマの一つである。このように作業全体の流れも類似している。

第一節　要求分析

立法過程では、国民や住民のニーズを把握して、それに対応した政策を施す必要がある。問題は、立案者がそのニーズを正確に把握していない場合や複数の人間のニーズが衝突し、要求が絞り込めない場合に生じる。これは、ソフトウェア開発を始め、経営や軍事などの幅広い分野で観察される現象である。むしろ、ソフトウェア開発の方が、立法に比べ、利害関係の一致する組織や個人のニーズから要求が生じることが多いため、このような問題は少ないかも知れない。しかしながら、それでも、曖昧な要求を絞り込む形式的手法を用いたり、観念的な要望を特定書式で記述して客観化を図ったり、あるいは、経営学等でも利用されている意思決定の方法論やそのための支援ツールを利用したりすることもある。一方、立法の分野では、政策シンクタンクなどの内部では、意思決定の方法論が使われるケースもあるが、自治体レベルではまだまだ普及していないようである。

意思決定の支援ツールとは、複数の目的や利害が関係するときに、それらを調整するための枠組みのことである。経営学やオペレーションズ・リサーチの分野で開発された方法論であるので、多くの場合、数学的取扱いが可能なものである。例えば、住居を借りる時に、条件として、(a)家賃、(b)立地場所、(c)広さ、(d)設備などを考えることができるが、最終的には、(A)安価で、(B)利便性が高く、(C)快適なところに住みたいと考えるであろう。(A)～(C)のすべてが満足できる場合ならば問題ないが、大抵はトレードオフ関係になる。そして、各条件が(a)～(b)の各目的に結び

付く組合せやその強さのパタンは数多く考えられる。それらは個人の場合であれば、主観的に決められるかも知れないが、集団の場合は、その構成員ごとにそのパタンが異なる。これらの主観的な結び付き関係の評価を全員から収集し、何がベストかを決めるための手法が意思決定の典型的な方法論である。例えばＡＨＰ(5)と呼ばれる手法を始め、いくつも提案されている。これらは、その主観的な評価を階層的に整理し、それらを数学的な方法によって計算し、全体として満足度が最大となるようなパタンを算出するものである。手計算では大変な手間となるので、現実的な分析を行う時はコンピュータ支援が必須となる。そのためのソフトウェアも開発されており、最近ではウェブ上で直ちに利用できるようなページも公開されている。

ソフトウェア開発では、このような要求分析の結果として、それらを客観化する「要求定義」を行ったり、それを形式化した要求仕様書を作成したりすることが多い。この時の技法やツールについては、立法過程においても流用できるものが多い。例えば、住民票の写しの発行業務のＩＴ化を考えてみると、人手による受付作業のプロセスや体制を分析し、新システムへの要望として記述する訳であるから、コンピュータ介在以前の人の作業や体制を記述する仕様を書くことで行うことになる。従って、ソフトウェア工学分野で利用されているツールをそのまま立法過程で流用できる可能性が高い。図１は、この業務を図示したものであるが、これはＥＲ図と言って、ソフトウェア設計時のモデル化に使われる記述法であり、要求分析の時にも利用される。矩形は実体を表し、楕円はその属性を表し、さらに、ひし形は実体間の関係を表す。これを見て分かるように、ソフトウェアの設計用であっても、コンピュータとは関係のない人手の作業システムや体制を記述しているだけである。つまり、ソフトウェア開発の局面でなくても、業務分析やその可視化に利用可能なのである。さらに、これを上手く作成できると、場合によっては、エージェント・システムと言って、この図に沿った住民や職員の行動をシミュレートできるようなシステムに

発展させることも可能である。なお、ＥＲ図はほんの一例であり、利用局面に応じて、ソフトウェア工学分野には、より適切な表記方法もある。例えば、手続きの方法や詳しい作業のタイミングなどの表記方法も数多く存在している。

このような図を作成するために、わざわざワープロソフトや図形描画ソフトを用いて記述するのでは、返って手間となる場合がある。そこでＥＲ図作成専用のソフトウェアを導入することも可能である。しかしながら、それでも日常的には、例えば、電子メールを用いる場合、添付ファイルとしてしか送ることができず、さらに、ウェブ上の掲示板システムなどで、議論しながら修正する場合にも直接書き込むことができないのでは不便である。そこで、筆者は、次に示すように、テキスト・データだけを用いて表記することがある。

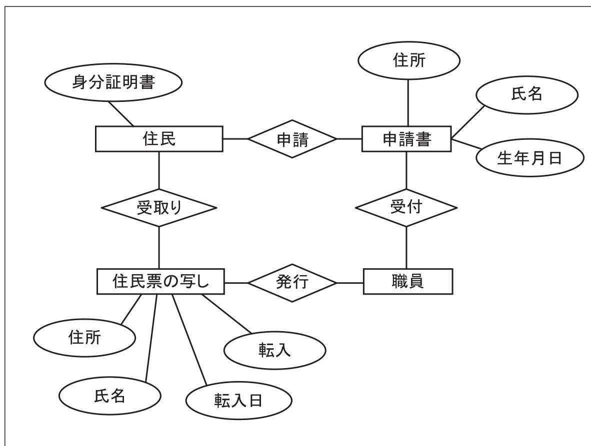
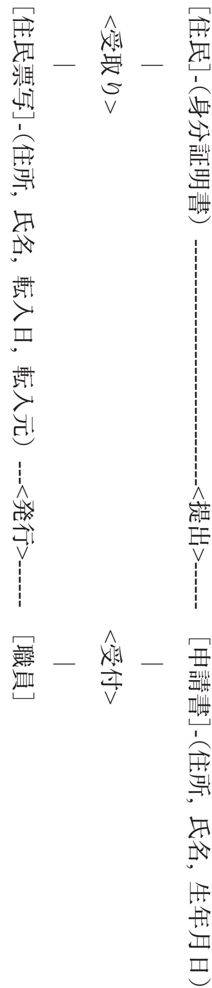


図１　ＥＲ図による住民票の写し（本人）の発行業務



立法過程における意思決定の材料として有力なものは、「立法事実」である。裁判の立証において、証拠が必須であることと同様に、政策立案においても、その動機となる事実がなければ、そもそも政策の必要性が疑問視されてしまう。そこで、要求分析の過程ではこのような立法事実収集のための作業が必須となる。住民の要望や専門家の意見を聞いたり、関連する文献的な資料を始め、その地域の現場資料を集めたりすることも必要になる場合が多い。これらのような動機を裏付けるような事実に対しては、ソフトウェア開発では、特に積極的に収集する必要性がほとんどない。多くはクライアントの方から先に提示されるし、問題があればユーザからの指摘も受けることになるからである。そこで、ソフトウェア工学の対象の中には、対象ソフトウェアのニーズの裏付けを取るための手法については特筆すべき手法が見当たらない。しかしながら、法令を作る動機の中には、直接、住民のニーズから生まれる立法事実に基づくもの以外にも、動機が与えられる場合がある。例えば、上位法との関係で立法を余儀なくされる場合や、首長の指示により、作成せざるを得ないケースである。このように動機となる要望の存在に対しては疑問がない場合でも、その要望に応じて打ち立てた対応策や方針の正当性について、立法事実のような証拠が

必要となる場合がある。このような時に根拠として提示される資料のうち典型的なものは、他の自治体の条例や関連する法令、行政訴訟の判例などのデータである。このようなデータは既に多くがデータベース化されており、ウェブ上でも素材としては多くが整っている。しかしながら、国が定めた法令は一箇所に集められているが、条例については、自治体ごとに様々なウェブサイトに散在しており、形式もまちまちで、根拠付けの資料とするには、工夫が必要である。法情報学分野で示されるようなデータベース技術や検索技法を用いることで、解決される問題も多いが、それらの技術は立法用ではなく、司法用であり、立法の根拠としてのデータベース化や検索技術は今後開発する余地がある。このような技術については、例えば、情報科学分野のデータマイニングの技術（膨大なデータの中から有益そうなデータを自動で発見する技術）や自動要約の技術を応用するなど、情報科学分野の成果を応用できそうである。これらはソフトウェア工学の技法そのものではないが、本稿の背景としても述べたように、本研究の最終目標には、法情報学の再構築も視野に入れているため、今後、本研究がカバーする対象として考えている。

第二節　仕様記述

立法過程では、政策案要綱や法令案要綱を始め、様々な要綱が作成されるが、これらは、法令案を作成する前段階の設計書であり、説明書でもある。ソフトウェア開発においても、このような設計書があり、それらは「仕様書」と呼ばれる。立法過程における要綱と同様に、様々なパタンの仕様書が存在する。それらの仕様書の書き方は、開発会社などによって千差万別であるが、多くの場合、内容的には同じタイプのことを表現している。その内容を大きく分類すると、作成段階の順に、外部仕様書、内部仕様書、詳細仕様書に分かれる。外部仕様書はユーザから見

たシステムの見え方を設計したものであり、内部仕様書は、その外部的に見える動作に対し、内部的な構成や実現方法を設計したものである。内部仕様は「アーキティクチャ」と呼ばれることもある。詳細仕様書は、その内部仕様に沿って、最終的に記述するプログラムと同程度の具体的粒度で記したものである。よく知られたフローチャートは詳細仕様書で用いられる記述形式の一つである。詳細仕様を特定のプログラミング言語の形式に書き変えることでプログラムが出来上がるのである。もちろん、個人で開発する場合や、小規模集団で開発する時には、このような設計過程を省略して、頭の中で詳細仕様まで考えて、いきなりプログラムを記述する者もいる。しかしながら、管理の厳しい大規模なプロジェクトでは、これらの仕様書を作成する。

仕様書は、通常、自然言語で記述される。自然言語とは、英語や日本語のように自然に社会の中で生まれた言語である。そこで仕様書に記述された仕様はコンピュータが実行時に参照する人工的な言語で記述されたプログラムのような素人にとって可能性の低いものとは異なり、設計過程や保守過程で参照される道標となるものであるから、可読性の高い記述が望ましい。もちろん、いくら可読性が高いと言っても、意味的に曖昧性がなく、最終的にプログラムとして記述可能、すなわちソフトウェアとして動作可能な範囲の設計でなくてはならない。しかしながら、自然言語で記述される以上、表現の曖昧性の問題や正確性を欠くという問題はなかなか消えることはない。そこで、まず、そのような表現の曖昧さや不正確さを避けたい、という要望がある。このような要望に対して、ソフトウェア工学では、あらかじめ書式の定まった図式を用いたり、「仕様記述言語」と言って、仕様書自体も人工的に設計された言語を用いたりして、仕様記述の明確化を図ることがある。仕様記述言語は、通常、仕様を数学的、あるいは形式論理学的に扱えるような形式で表記するための人工的な言語である。多くの場合、数学的には容易に扱えても、素人にはただの数式かプログラム本体と変わりがなく、必ずしも、可読性の高いものとは言えない。そこで、

前者の図式の方を用いたり、仕様記述のための支援ソフトウェアを用いたりして、操作や見え方については図式的で分かりやすくし、内部では自動的に仕様記述言語に変換する方法が用いられる。こうすれば、その支援ソフトウェア内部では、仕様の正当性や正確さをある程度自動チェックすることが可能だからである。

政策設計に関しても、必ずしもこれらの仕様書に対応する要綱が必要な訳ではないが、似たような手順を導入することができる。そして、前節の要求仕様の場合と同様に、仕様記述言語や図式を用いることで客観化や可視化を図ることができ、特に外部仕様に関してはコンピュータの内部動作を前提とするものではないので、支援ツールも流用できる可能性がある。少なくとも、方法論の考え方には参照可能なものもあるであろう。例えば、許可制度の設計を行う際に、申請者が行う手続きの手順は外部設計として行い、申請者からは職員の作業をブラックボックス化して見えないものとした上で、申請者がどのようなアクションを行い、どのようなサービスを受けるのか設計することから始める。もちろん、波及する影響もアウトプットと考えておかなくてはいけない。一方、それらの入出力に対して、内部仕様として、自治体の担当者や首長がどのような手続きを踏むとその行政上の対応が実現できるかを設計する。その動きを条文で表現できる範囲のものに落とし込むことが詳細仕様の段階である。ソフトウェア工学に限らず、システム科学からの知見によれば、システムを外部から見て内部との境目をはっきりさせることと、それらを分けて認識することはシステム設計上、重要なことである。制度が良質なシステムとして機能するためには、制度設計もシステム設計の例外ではなく、このようなシステムの境界をはっきりさせた上で設計を進めるべきであろう。そのような時にも、境界の認識の仕方から始まって、モデル化の方法など、ソフトウェア工学から流用できるものはいくつもある。それらに付随して、仕様書の書き方や考え方、および書式などが応用でき、その支援ソフトウェアも流用できるのである。

次に、複雑な政策の場合には、政策の手続き的な記述だけではなく、対象とする物事の構造に言及した法令を記述しなくてはならない場合もある。そのような物事を表す言葉の意味付けが曖昧であったり、アドホックに用語を作り出したりしていては、混乱を招くことになる。そこで、手続きに関する書式だけではなく、法令で用いられる様々な概念についても、客観化を行い、曖昧さを減じたいという要望が生まれる。このような要望に答えて、当該用語について、表現と意味との対応を事前に定義したり、その定義方法自体の書式やメタな用語法を定義したりするための工学的手法が伝統的には知識工学という分野で古くから研究されている。これらは、ソフトウェア工学として直接扱われている方法論の応用ではないが、人工知能などを始めとする高度なソフトウェア開発分野では必須の方法論である。特に、「セマンティック・ウェブ」と言って、インターネット上のデータを単純な情報記述の羅列ではなく、体系化され標準化された形で置いておき、単なる検索ではなく、人間の推論のように高度な情報獲得や加工を複数のウェブ上で連動させて行う技術が注目を集めているが、そこで中核をなす技術が知識工学的な手法である。最近では、このような知識の形式化に関する情報科学的分野は「オントロジー工学」と呼ばれている。そして、そのような対象知識を記述する際の仕様記述方法を「オントロジー」と呼んでいる。さらに、正確な用語法ではないが、そのようなオントロジーとしての記述方法に則って書かれた記述自体も「オントロジー」と呼んでいるケースが多い。なお、今日の情報科学の分野では、「オントロジー」という言葉が元々の語源である哲学の「存在論」を意味することはほとんどない。

こうして、先の仕様記述もオントロジーを用いて、書式や用語法を仕様として事前に定義して関係者で共有することで客観化を図ると、機械的な処理には非常に都合が良い。ただし、一般には、可読性が低くなることは避けられない。例えば、政策についても、オントロジー工学の分野でよく見かけるような書式で記述したとすると、その

一部は次のような記述になる。

```
<pub_policy type="資格付与" lib="免許制">
  <name> ○○ </name>
  <criteria type="human">
    <test lib="資格試験" name="○○" year="1" ></test>
    <registry type="登録簿" ref="◇◇"></registry>
    <except lib="前科" years="3"></except>
  </criteria>
  <monitor>
      <committee member="lawyer" ref="△△">
        <name>資格監査委員会</name>
      </committee>
  </monitor>
</pub_policy>
```

これは、資格付与制度の一部を表現した例である。この例は「○○資格付与の制度は、三年以内に前科がない者が、その資格試験をパスして登録することによってなされるもので、その試験が年一回であること、および資格監

査委員会の監査を受けることが条件となっている。」ということを表現したものである。もちろん、この表現例は、ほんの一例であり、より優れた表現方法もあり得る。

この他にも数式や論理式を拡張したような表現方法も様々なものが提案されている。例えば、論理式を用いる手法は筆者も報告している。[(6)] そのような表現方法を用いて、政策の分類階層の一例を示すと次の通り。

資格付与 ⊑ 法政策
経済的誘導 ⊑ 法政策
許認可 ⊑ 法政策
登録制度 ⊑ 資格付与
免許制度 ⊑ 資格付与
免税制度 ⊑ 経済的誘導
補助金制度 ⊑ 経済的誘導
融資制度 ⊑ 経済的誘導
許可制度 ⊑ 許認可

ここで、「X⊑Y」という表記は、XがYに含まれていることを示す。つまり、YがXの上位概念であることを意味する。

このような分類階層の表記があれば、分類構造そのものを電子化しておくことができる。その結果、例えば、ソ

フトウェア工学では今日広汎に用いられている、「属性継承」というテクニックが利用できるようになる。それは「分類階層上の上位概念が持つ性質はその下位概念も持って（継承して）いるとみなして良い」という法則に基づくものである。一般的な例で考えると、「人間」は上位概念の「哺乳類」の性質である動物的な諸性質や授乳などの性質を継承していることから確認できるであろう。そこで、最近のソフトウェア開発過程ではこの原理を取り入れて、下位概念については上位概念の持つ性質が記述されなくても、自動的に継承されて内部的に処理される、というメカニズムが実現されている。これを立法過程の支援システムに導入できたとすると、例えば、資格付与制度の一般的な項目を一旦定義しておけば、その下位概念である免許制度では、資格付与制度一般の項目と同じ部分については、人手で新たに記述する必要はない。システムが自動で記述してくれるのである。現在目にする多くのソフトウェアもこのような仕組みで開発されている。このような便利な恩恵も、概念構造が客観的に電子化されていることが大前提である。その意味でも、このような表現方法が役立つのである。

その一方で、これらの表現を一見して分かるように、これらも仕様記述言語と同様で、通常の人間にとって可読性の高いものではない。そこで、多少でも分かりやすくするためには、やはり、グラフィカルな表示や操作を補助する政策設計用のソフトウェアを提供するべきであろう。つまり、それらを操作して政策設計を行い、数式や記号表記によるストレスやエラーを減らし、一方で、内部的には厳密な政策構造の記述に自動変換させて、システムに蓄えておく、という方法が実用的だと思われる。本稿の第九章では、実際にそのような機能を持つソフトウェアの試作版を紹介する。

第三節　設計方法論

一人の優秀な人間がすべてを作成する場合には、法令であっても、プログラムであっても、作成の方法論は問題にならないかも知れない。その場合の設計は、その人の頭脳の中か、一部のメモ書きとして、描かれるだけであり、なかなか客観化されるものではない。それでも、全工程を通して一人で行うのであるから、この場合は、作成段階での支障はほとんどない。しかしながら、通常の能力の人間が、複数人数のチームで作成する場合には、設計段階から、意思疎通が必要であり、何らかの形で客観化される必要がある。そうでないとコミュニケーションも図れないし、考えもまとまらない。ソフトウェア工学の分野でも、設計段階の方法論は古くから様々なものが提案されている。従来から存在する方法論の中で共通に扱われているテーマのうち主要なものは、①データやシステムの**モデリング**と、②処理単位の**グルーピング**に関するものである。

データやシステムの**モデリング**とはこれから作成する対象について、どのようなモデルを通じて捉えるかを考えることである。ソフトウェアにしろ、制度にしろ、そのものが物質的実体として存在する訳ではないので、何らかの形で可視化するには、議論の対象として抽象的に必要な性質を備えた、目に見える仮の形状が必要になる。それがモデルである。この場合の「可視化」は文章や数式による記述でも構わない。

ソフトウェア開発におけるモデリングとは、例えば、連絡先データベース・システムを作成する時に、日常的に利用する実体的な住所録のメタファを使って、一つのエントリごとに、氏名、所属、住所、自宅電話、携帯電話の番号が記述できるように設計しておく、というような作業である。また、データとプログラムを一体化して、あたかもコンピュータ内に人間の分身ロボットがおり、人に代わって作業を行ってくれるようにモデル化する設計方法

もある。その場合は、住所録というようなもののモデルはもちろん、その住所録の読み書きをする所有者のような動作主体もモデリングする。コンピュータの中に人がいる、というと、ＳＦ的なイメージを持つかも知れないが、そのような話ではなく、あくまでも、抽象レベルでの模型の話であり、具体的に実現する時には、そのメタファが持つような姿ではなく、機能だけが実現されるようにプログラムを書くのである。例えば、「新たに住所録に氏名を書き込む」というと、人間が書き込み動作を行う様子をイメージすることができると思うが、プログラム上では、例えば、実際には、次のように記述されるだけである。

TheEntry=TheAddBook.newEntry()
TheEntry.writeName("山田太郎")

このような記述によって、具体的にシステム内部の住所録や各エントリへの操作が起動されるわけであるが、もちろん、それは、実際には、コンピュータ上のメモリやハードディスク上のファイルに対する操作であるから、元々人手でなく、電気信号によるものである。そこで、具体的なハードウェアの操作を意識しなくても、分かりやすいメタファにつながるようなモデリングを行っておけば、詳細が分からなくても設計は可能なのである。実際に、ほとんどのプログラマは電気信号のイメージでプログラミングしている訳ではなく、抽象的なモデルを想定して、それを操作しているイメージで開発を進めている。そこで、このモデリングのための方法論が設計段階から重要なファクタとなるのである。

政策もある種のシステムであり、システム自体は、必ずしも、物質的実体として目に見えるようなものではない。

政策とは、物理的な自動車、細胞、太陽系のような、対象そのものが実体として見えるようなものではなく、目に見える部分はその政策に従って動く人々の行為やその結果引き起こされる様々な現象だけである。それらを一つの政策として実体のごとく観る訳であるから、設計者はもちろん、広く国民や住民に理解できるように上手く表現する必要がある。つまり、政策もイメージしやすいモデルとして表現する必要性があるだろう。たしかに、単純な政策も多いし、規制される行為自体は目に見えるものであるから、常にモデリングが必要とまでは言えないかも知れない。しかしながら、手続きが複雑になっていたり、他の法令と関連性があったり、あるいは、法令の扱う対象が複雑な実体であったりする場合には、明確にモデリングした方が良い場合もある。極端な例だが、税率や控除率などの関係は数式化して、できれば、シミュレート可能なグラフィック化を行って検討した方が良いだろう。このような作業も一種のモデリングである。

次に、処理単位の**グルーピング**も設計時点で考慮すべき重要なファクタである。特に、複数の人間が設計に関わるのであれば、これを上手く行うことが後のプロセスの混乱や品質の低下を防ぐことにつながる。立法の局面ならば、多くの自治体で行われている、他の自治体や法令を再利用するというケースでも、円滑にこれを行うために処理単位をうまくまとめて、パーツ化しておくことができれば、利便性を向上させることにつながる。これらのような処理単位のグルーピングをソフトウェア工学では「モジュール化」と呼んでいる。複数の人間が協調して開発を行う際に、製作する全プログラムをいくつかに分割して、分業で開発作業を進める必要がある。そのための分割もモジュール化と考えることができる。また、共通要素をくくり出すことで汎用の部品的な処理をまとめておくこともあり、その場合には、当該開発以外でも、再利用できる可能性が生まれる。その意味ではパーツ化の作業もグルーピング作業に含まれる。なお、最小の処理単位から見ると、それらを集めてまとめるのであるから、グルーピング

とみなすことができるが、システム全体から見ると「分割」しているとみなすこともできる点に注意されたい。

このモジュール化によって、各作業者が担当部分を円滑にプログラミングするためには、他の作業者の担当部分との独立性が高い方が開発効率を高めることができ、品質も向上する。例えば、自分の担当するプログラム内部で用いるローカルなデータを他のモジュールの担当者がいちいち参照しなければならなくなると、打合せのためのコミュニケーションのオーバーヘッドが大きくなり、進捗も低下し、誤解やエラーが混入する率も増える。そこで、モジュールの外部に公開すべき最低限のデータ構造については、その規約を先に決めておき、以降はそれに従い、公開しない部分は、担当者のみがローカルに設計を行うことで各自が円滑に開発を進めることができるようになる。喩えるなら、部署間のやり取りの連絡用の書式は共通にしておき、内部の作業は各部署に任せるようなものである。そして何か情報が欲しい時には、各窓口を通じて、その書式に従ってやり取りをする、そのような作業方法の本質が、実は、このような独立性の向上にあるのである。なお、このような公開するやり取りのための規約、すなわち、仕様も「インターフェース」と呼ばれることがある。他のモジュールとの直面する結合の部分の仕様のことなので、内部と外部の境界面という意味から生じた用語法である。

モジュール化の方法は多数あり、ソフトウェア工学の古典的著作にも、その分類や評価が詳細に議論にされている[(7)]。現在の主流となっている手法に直結する考え方だけ示すと、ある共通のデータを中心に処理単位をまとめて一つのモジュールにしておき、そのモジュールの外部からは、当該データ自体を見えなくしてしまう「情報隠蔽」という考え方を使ったモジュール化技法が挙げられる。これは「情報的強度」の「モジュール内粘着度」と言われ、最も内部的結び付きの強いモジュールの構成方法であるとされている[(8)]。この考え方をベースに「データ抽象」と呼ばれるモデル化方法も提案された。これは、具体的な内部のデータ構造を知ることなしに、抽象的なモデルのみを

意識しておき、そのモジュールの操作用プログラムとして公開されたものだけを用いて、プログラミングを進める手法である。現在、ソフトウェア開発分野において主流となっている「オブジェクト指向」と呼ばれる一連の方法論も、「カプセル化」と呼ばれる一種の情報隠蔽技術がその本質的特徴の一つとなっており、データ抽象と同じような操作体系を想定している。そこでは、中身を見せないままで外部に提示される抽象データを「オブジェクト」と呼び、その操作プログラムを「メソッド」や「メッセージ」と呼ぶ。例えば、パソコンの画面上のウィンドウはオブジェクトとしてプログラミングされており、他のプログラマは画面一杯に最大化する場合であれば、そのウィンドウ・オブジェクトに対し、「最大化せよ」という旨のメソッドを送るだけで実行させている。

こうして、ソフトウェア開発の分野では、モジュールの独立性を高めている。立法過程においても、本来、局所化して集めておくべき施策が分散してしまうと、思わぬエラーが混入してしまうし、チームで立案することや考えをまとめることが難しくなるので、このような手法を取り入れるメリットはある。

なお、モジュール化の利点としては、開発効率や品質問題だけに貢献する訳ではなく、このデータ抽象のように、内部の複雑でテクニカルなことを意識しなくとも、理解しやすいモデルだけに意識を集中できるという観点からも有効である。例えば、まだ存在していない下位法の立法以前に上位法を作成しなくてならない場合など、このデータ抽象のような技法が効いてくると思われる。なぜなら、具体的な法令が存在しない状態で、法令の機能を予定して、立法を進めるのであるから、その下位の法令を抽象的に捉えてモデル化しておき、そのモデルの状態で、必要に応じて参照すれば、実際に下位法令ができた時にも整合性を維持できるからである。さらに、下位法令を設計する際には、モデルが先に存在することになり、整合的に制度設計や条文記述を行うことができるのである。多分、潜在的には同様の考え方で、法令も作成されていると考えられるが、本稿では、それを職人技としてではなく、誰

でも利用できる技術として提示する必要があると考えており、そのためにこのようなソフトウェア工学の手法をベースとする方法論を提案しているのである。

第四節　テキスト処理

これまで指摘したように、法令や例規というものは、物理的に見れば、文書であり、言語で記述された文字列、すなわち、テキストである。一方で、ソフトウェア開発の中で利用されているツールの中にも、その実体としてのテキスト性に着目したものが数多く存在する。従って、これらのツールを応用できる可能性が高い。多くの読者も、日常的な事務作業において、パソコン上の文書（テキスト）内の文字列検索程度は経験されているであろう。これもテキスト処理の例である。もちろん、この他にも、テキストの取り扱いを支援するツールは存在する。ソフトウェア開発に特化される訳ではないが、自動的に書式を整えたり、漢字コードを変換したり、索引を作ったり、文書の差分を取ったり、特定文字列を置換したり、様々なツールがソフトウェアとして、それも大抵はフリー・ソフトウェアとして公開・配布されている。これらは、当然、条文を書き記す時にも必要に応じて利用でき、人手で清書を行う場合や法令から様々な加工データを作成する場合には有効である。このようなツールであれば、既に利用している自治体や中央省庁の職員がいるだろう。

もちろん、ソフトウェア工学的な観点から支援するシステムも存在する。例えば、プログラムは、通常、モジュール、クラス、あるいは、関数と言った、処理の塊のようなものごとに分けて記述されているが、編集中にそれらを相互に切り替えたり、その塊ごとの情報を収集したり、他の塊との参照状況を自動でチェックしたり、まとめたり

するツールである。これらのツールは、法令用にカスタマイズする必要はあるが、立法過程の作業でも応用できる。参照している他の法令や条文をコンピュータが自動的に表示してくれたり、ある部分を修正すると、それに合わせて、関連しているすべての部分を自動的に修正したりすることも可能である。このような手法を導入すれば、自動化によって、人手によるエラーや見落としを防止することができる。加賀山・松浦らの編著による著作『法情報学』[9]の付録ＣＤ-ＲＯＭにも、テキスト処理のためのソフトウェアがいくつか添付されている。その中には、年号の一括変換や送り仮名の統一を行うツールなどが含まれている。

しかしながら、本稿で強調したいことは、それらのようなヒューマンエラーの防止といった、消極的な理由としてのみではなく、コンピュータにこのような作業をさせるということは、コンピュータが判別できるくらい、曖昧性や不正確さを除去した、形式的な明確化が必要になり、そのような条文記述方式を推進することに貢献できる点である。例えば、相互参照が複雑過ぎて、参照がループ（循環参照）になっている、あるいは曖昧な記述で分かり難いといった問題が、機械的に検査できるようになるので、分かりやすく品質の向上した条文を提供できることにもつながる。

その他の局面でも、法令の改正を行う場合には、旧法と新法の差分を取る必要がある。その際、新旧対照表を自動生成したり、溶け込まし文を自動作成したりするツールもテキスト処理技術で実現できる。このようなツールは、既に、実際に多くの自治体で用いられているソフトウェア製品にも導入されている。[10]

単なる文字列の処理だけではなく、音声認識や翻訳技術などの中核をなす、自然言語処理技術を使うと、条文の構文や語彙のチェックなど、かなり高度な作業を支援することも可能となる。最終的には、条文の清書に関しては、人手でなく、コンピュータによって、要綱を基に自動的に法制執務の様式に則った形で記述させることも可能にな

るであろう。例えば、「若しくは」「又は」「及び」等の使い方の関係を検査したり、助詞の使い方の自動修正をしたり、係り受けをチェックしたりすること等も可能になる。さらに、シソーラス情報や概念の構造情報などを用いた、ある程度の意味処理も導入できれば、矛盾の可能性の指摘などの論理的な整合性を検査したり、不自然な用語法を指摘したりするような支援ツールも提供可能になる。例えば、マイクロソフト社のExcelで数式を入力したことのある読者も多いと思うが、その数式内の括弧や演算順序の認識の技術は、古くから存在する「構文解析（parsing）」処理と呼ばれる典型的な処理技法の一種である。この技術を用いれば、「若しくは」「又は」などの結合の仕方の規則を「＋」「×」などの演算順序の扱いと同種の処理技法によって、コンピュータに認識させることが可能となる。技術的には比較的簡単な方法であるが、問題は、参照される数多くの文法や用語法をソフトウェアの中に事前に設定しておかなければならない点である。つまり、このような支援ツールの実現を推進するための問題点は、技術的に困難な点よりも、むしろ、書式、文法、あるいは、語彙に関する情報が文書としては存在するが、電子データとしては全く不足している点である。しかしながら、これらは、開発の初期に文法などのデータを作るプロセスが面倒であるということであり、高度な技術者でなくても、一定の素養を持つ人々による人海戦術で解決可能な類いの問題である。

第五節　バージョン管理

法令の改正はしばしば行われる。これはソフトウェアも同じである。ソフトウェアの場合、その変更履歴は、差分に説明を付けて記録しておく。このような作業のためのツールは古くから存在している。最も古いものは単純に

新旧の両テキストを入力して、その差の部分を抜き出すプログラムであった。それらは「diff」と命名されたプログラムである。このようなプログラムを基本部品としたバージョン管理専用のフリー・ソフトウェアも存在しており、今日でも多くのプログラマによって利用されている。[11] 特にチームで開発する場合には必須のツールとなっている。

これらのソフトウェアは、その処理対象がプログラムの形式でなくても、すなわち、自然言語による通常の文書であっても、テキストファイルとして作成されていれば、直ちに利用できる。先に言及した、新旧対照表や溶け込まし文を自動生成する機能を備えたソフトウェアにおいても、このようなテキスト処理技術をベースにしたバージョン管理の機能がその中核をなす機能として備えられている。現在の技術水準からすると、法令のバージョン管理はコンピュータを使って行われるべきであり、むしろ、職人的な手作業でこれを行う方が行政のスリム化に逆行しかねない無駄な労力であろう。

なお、法令のバージョン管理を行うには、単に差分を取るだけではなく、その差がどの範囲に及ぶかも考慮された管理が必須となる。そこで、差分を取る仕掛けだけでなく、法令の参照関係なども同時にシステムに保持させておき、整合性が維持できるように参照関係の情報も記録しておくことになる。人手では、作業ミスや見落としの可能性もあるし、そもそも膨大な量を見直す必要があり、労力がかかり過ぎる。しかしながら、これらは、機械的に行える作業であり、ソフトウェアに任せることができる作業であるので、質的にも量的にも作業プロセスを改善できる。ただし、現在の法令管理のためのソフトウェアは、自分達の管理する法令しか蓄積していないため、上位法が変わった時や、所管の異なる法令の影響を受ける時など、その参照関係の検査が網羅的にできない場合がある。

そこで、網羅性を確実なものとするには、ネットワークの利用を前提にした、相互参照のできる仕組みや標準化が

必要となる。いずれにせよ、人手ではなく、自動化が可能な処理であることに変わりはない。強いて、人手が必要となる部分があるとすれば、コンピュータが形式的に判断できない意味的な参照を避けて、誰でも形式的に判断できるような参照を行うように法令の記述スタイルを定めておき、そのスタイルを守るようにする必要がある点である。

第六節　再利用

自治体で条例を作成する場合などには、第三章で示すように、実際には、逐次的・段階的に作成されるとは限らない。小規模な自治体の場合は、特にその傾向が顕著である。その場合は、ほとんどが他の自治体の条例を元に作り直したり、政府等から提示される雛型をカスタマイズしたりして利用している。これは、作業効率を向上させるための理由もあるが、他の自治体での施行実績がある場合や中央省庁の指示に基づく場合は、いずれも、一種の保証が付いたようなものであるから、安心して利用できるという、品質の維持という側面からの理由もある。このような場合の方法論は、ゼロから作り出す訳ではないので、設計から逐次に製作を進める方法論とは別途考慮する必要がある。

実は、ソフトウェア開発の現場でも、予算や人数の限られた小規模プロジェクトにおいては、ゼロから作成するような段階的な工程を経ずに、最終的な製品こそが重要、という考えから、過去に実現したシステムを作り変えたり、それに使われた部品を集めて別のものを組立てたりして、品質を保ちながらも、効率化を図ることがしばしば行われる。そのために配慮すべき留意点や用いる手法は、立法過程で、既存法令や雛型を再利用する場合にも流用

できるものがある。例えば、全く同じ政策を実現する条例の場合であれば、条文中のその自治体独自の部分を書き換えるだけで良い。特に、申請書式などを例規の中に含めて規定する場合は、自治体名を置き換えるだけで流用できるものもある。しかしながら、通常、自治体で行われているように、単に書式の図案やワープロ文書のような形で保存されたデータのみを扱う場合には、意味的に変更が必要な箇所などに気付けないまま、エラーが紛れ込んでしまう場合もある。ソフトウェア工学でプログラムを再利用する場合には、そのような表面的な流用を行う場合もあるが、ほとんどの場合、元々流用可能なようにパーツ化されて設計されていることが多いため、そのパーツ単位で用いる場合はこのようなエラーを回避できる。なぜなら、各パーツの内部はブラックボックス化されて、「パラメタ（媒介変数）」と呼ばれる可変部分だけが外部に示されており、そのパラメタ部分だけ指定すれば、後は自動的に処理されるからである。法令における「準用」の発想をシステム化したものと考えれば良い。あるいは、判例の文章中で人物の固有名詞を「X」や「Y」に置き換えているのと同じで、必要に応じて、その中に本名でも仮名でも入れることができる状況を考えて頂きたい。この時の「X」や「Y」をあらかじめ括り出しておいたものがパラメタである。現在、立法過程で行われる設計は、ほとんどの場合、再利用を前提としたパーツ化などは意識されていない。実情としては、他の自治体での再利用を考えている余裕などないのであろう。しかしながら、本稿で提案するようなシステムを導入したり、そのシステムを前提とする、いわば、「政策テンプレート」のようなものを電子的なデータとして用意して配布したりすることで、政策設計に関しても、このような再利用やパーツ化を図ることが可能となる。

第七節　簡易開発

前節でも言及したが、小規模な自治体の場合、必ずしも段階的に条例が作成されない場合が多い。その場合、既存条例を再利用することが多い。この時、先に作成されるのは、要綱ではなく、条例案であり、その後に説明資料として要綱を作成することもある。そのようなケースでは、検討会議（会議の名称は自治体によって異なる）へ何度も繰り返し提出して、そのやり取りの中で、徐々に修正が加えられるという方式を取る。

ソフトウェア工学でも、例えば、近年、「テスト・ファースト」という手法が着目されており、この手法では、先に、プログラムを利用した結果として得られる予定のアウトプット（「テスト仕様」と呼ぶ）を事前に決めておき、とりあえず、作成したプログラムをテストして、その予定通りのアウトプットとなるか検査を行う。こうして、そのテストをパスするまで何度も修正を続ければ、最終的には期待していたようなソフトウェアが完成する、という手法である。この場合は、設計書よりも、その予定されたアウトプットを記述したテスト仕様書が重視される。自治体では、テスト仕様書とまで言えるものがあるとは限らないが、政策に期待される効果というものは、関係者らが各自で思い描いているのであるから、その関係者らが、提示された条例案通りに物事が動いたと仮定してみた場合、その効果が得られると思うか否かを検討してみれば、テスト・ファーストと同様の手続きと考えることができる。もちろん、さらに、客観化を図ってチェックリストのようなものを作るのであれば、それはテスト仕様書と同じである。

ソフトウェア工学の分野では、テスト・ファースト以外にも、仰々しい段階を経ることなく、簡易、あるいは、迅速に実質的開発を進める手法が数多く提案され、実施されている。詳しくは第三章で示すが、それらの開発手法

は、九〇年代から注目され始めた非段階的な「アジャイル開発」と呼ばれる種類の開発方法に代表される。これらは、当然、利点もあれば欠点もあるが、やはり、小規模開発チームであったり、時間的に余裕のない開発であったりする場合には有効である。これらの簡易な開発技法に共通する考え方は、先に完成品に近いものを作ってしまうこと、無駄を除去すること、等である。

立法過程でも、「もの作り」という観点からすれば、これらの方法論に学ぶところは多い。これらの手法に用いられた元々の着想には、トヨタ自動車などの効率的な生産手法からヒントを得たものもあり、そもそも、ソフトウェア業界に限定された考え方ではない。もちろん、適用領域によって、多少の修正は必要だが、本質はほとんど変わらないと考えられる。したがって、立法過程にも、このようなアジャイル開発の手法やツールと同様のものを導入して、効率性や品質の向上を図ることができると期待している。

第三章　ソフトウェア開発と立法過程

第一節　プログラミング

本稿の冒頭で述べたように、法令の作成がソフトウェア開発と類似しているという着想が本稿の核となるものなので、本節では、その元となるソフトウェア開発の中心であるプログラミングについて導入的な説明を行う。

「もの作り」の技術に関する学問分野は「工学」である。「ソフトウェア工学」とは、ソフトウェア作りを対象

にした工学である。ソフトウェアを作成する際の実体的な対象物は、通常、コンピュータ用のプログラムである。このプログラムと言うものは、要するに、人工的に作られた特定の文法や用語法に則って記述された、文字列や記号列の集合体であり、文字で記述された文書（テキスト）の一種である。人工的に作られた文法や用語法はまとめて「人工言語」と呼ばれ、日本語や英語などの自然発生的な「自然言語」とは区別された呼称を持つ。特に、プログラムを記述する目的の人工言語は「プログラミング言語」と呼ばれる範疇に入る。このプログラミング言語で記述されたプログラムの中でも、特に、人間が直接に記述するものは「ソースプログラム」や「ソースコード」と呼ばれ、素人にとっては可読性の低いものであるが、慣れた者にとっては、それなりに可読性も高いものである。これに対し、コンピュータが直接電気信号として読み取るプログラムは「機械語プログラム」や「オブジェクト・コード」と呼ばれる。これはプログラマでも読みこなすことが困難なものである。そこで、実際には、ソースプログラムを機械語プログラムに自動変換する「コンパイラ」と呼ばれるソフトウェアが用いられる。ソフトウェアを作るソフトウェアである。コンピュータの世界では、データもプログラムも電子的な信号の羅列としては全く同一の形態なので、しばしば、このような自己言及的なことが行われる。最近ではそのような自己言及的な技法を駆使して、エミュレータや仮想化技術と言って、他のコンピュータや専用ゲーム機の振る舞いをソフトウェア的に別のコンピュータで実現してしまう技法が盛んに用いられている。

プログラミングの説明のために、まず、簡単なプログラムの例を示すと次の通りである。

［ソースプログラムの例］

x　=　a+b

y ＝ c+d

z ＝ max(x,y)

print z

［機械語プログラムの例］

0001010111010100010100101000101011111011011110000010101010101……

このようなプログラムをコンピュータのメモリに事前に入力しておき、コンピュータがそれに従って動作する、という方式が、現代のほとんどのコンピュータで採用されている「ノイマン型」と呼ばれるコンピュータの動作方式である。そこで、ソフトウェア工学の「もの作り」の対象は、このようなコンピュータ動作を指示する内容が書かれたテキストとしてのプログラムである、と考えて良い。

一方、動作主体がコンピュータではなく、人間の場合を考えてみよう。例えば、ある式典の進行プログラムに沿って、主催者側の人々や参加者が行動を行うことがある。あるいは、誰かによって記述されたある特定のルールに従って、スポーツやゲームを行うこともある。そして何より、国家や自治体を始め、企業や学校、あるいはサークル等にも、多くの場合、明文の規則があり、構成員は、それらに従って行動する。もちろん、人はコンピュータとは違い、書かれている通りに行動しないこともあるし、そもそも、記述されたプログラムやルールの解釈も一通りではないので、同じ記述内容であっても、常に一定の動きとなるとは限らない。しかしながら、プログラムやルールを記述する時点の人々の作業に目を向けると、コンピュータ・プログラムを作成する時点で行われる作業と大差はないのである。例えば、目的を考え、資料を集め、構成を考え、下書きを行い、最後に清書を行う。これは、ソフト

ウェア開発でも、立法過程でも同じである。それどころか、今回の研究対象からは外れるが、論文や小説、あるいは催し物の企画でさえ、これに似た手順を踏むであろう。ただし、論文や小説に比べると、法令の場合、最終的な書式の厳密さや公布後のメンテナンスの点で、よりソフトウェアに近いものである。そもそもの目的も、コンピュータ・システムや人間の社会システムという、あるシステムの動作を規定することにあり、その点でも、相対的に類似度が高いテキストを扱っていると考えることができる。

次に、情報処理の専門でない読者のために、プログラミングについて、ソフトウェアの動作メカニズムとともに、説明を行う。ある程度予備知識のある方は本節の以降をスキップして頂いて構わない。

まず、知っておいて頂きたいことは、現在のコンピュータ・システムを前提にする場合、ほとんどのプログラマは、自分の書いたプログラムによって、コンピュータのハードウェアが詳細にどのように動いているかということを知らない、ということである。これは、アプリケーションの動作を即座にハードウェアの電気信号に置き換えて考えられるプログラマはほとんどいない、という意味でもある。筆者の予想では多分皆無だと思う。もちろん、四半世紀前ならば、特定のパソコンを対象に、そのようなハッカーも確かに存在した。しかしながら今日の複雑で巨大なコンピュータ・システムではほぼ不可能であろう。いずれにせよ、このような状況でも、現在多くのソフトウェアが開発されているのであるから、ソフトウェア開発に際しては、プログラマは具体的な動作状態を知らなくても、抽象的な理解のままで、開発を進めることができるのである。

このようなことが可能な理由は、人間にも容易に理解可能なコンピュータへの指示書き（プログラム）を機械語による具体的な動作命令の電気信号に変換（言語の変換なので「翻訳」と言う）する作業をソフトウェアを用いて自動的に行っているからである。遂次変換するためのソフトウェアを「インタプリタ」、あらかじめ一括して機械

語に変換するためのソフトウェアを「コンパイラ」と呼ぶ。

この状況を理解して頂くために、簡単なプログラムを例に採り上げる。ここでは、ウェブ上でそのホームページへの何人目の訪問者であるのかを表示するプログラムを考える。これは、次のようなプログラムとなる。

あなたは、<dtml-var n>人目の来訪者です。
<dtml-call "manage_changeProperties(n=n+1)">

あとはコンピュータ上には、そのホームページの属性としてｎという仮想の「メモ」を用意することができるので、そのメモの初期値を1に設定しておくだけで動作する。コンピュータの本質的機能は、このような仮想のメモを物理的な記憶装置の容量の範囲内でいくらでも用意して読み書きを行える点である。

このプログラムが書かれたホームページにアクセスすると、このプログラムが実行されて、

あなたは、１人目の**来訪者**です。

と表示され、再びアクセスすると（再読込みボタンを押すと）、再び実行されて、

あなたは、２人目の**来訪者**です。

と表示される。プログラム中の「<dtml-var n>」は「この部分にｎの現在値を表示せよ」というシステムへの命令であり、実際のホームページではこの行に対応して、最初の値である1が表示される。後半の「」と「」と記された部分は「それらに挟まれた部分を太字で表示せよ」という意味の命令である。そこで、この行は全体として、先に示したような一連の表示結果になる。次の行では、「ホームページの持つｎの値に1を加えた値を計算して、その値でそのホームページの属性ｎの値を変更せよ」という命令を実行している。この行に対応する表示指定は何もないので先の一行だけが表示されるのである。このプログラミング言語では、大抵の場合、「＜」と「＞」で囲まれた部分は非表示のままでシステムへの直接命令と解釈される。このような規約はプログラミング言語ごとに異なるので、詳細な表記法は本質的ではない。今回のプログラムは、Ｚｏｐｅ[12]という特殊なウェブ・システム上で動作する、ＤＴＭＬと呼ばれるプログラミング言語を用いた書き方であり、その記法に則って、プログラムを書いたものである。プログラマはウェブ・システムの詳細な仕組みを知る必要もなければ、コンピュータの仕組みの詳細を知る必要もない。単に、Ｚｏｐｅと言うシステムを使えば、ホームページごとにいろいろな値を設定することができ、それを特定の表記法で操作できる、ということだけ知っていれば良い。具体的な表記方法については、必要に応じてリファレンス・マニュアルを参照すれば良いだけである。特にこのＤＴＭＬの場合ならば、実際には、規約がさほどある訳ではないので、よく使うものは、自然に覚えてしまう。

　コンピュータの実際の動作を図2に示すと、このプログラムはＺｏｐｅシステムを介して、自動的に一段下の層のPythonと言う別のプログラミング言語に翻訳され、その翻訳されてPythonで再表現されたプログラムも、再びPythonインタプリタと言うソフトウェアによって、さらに自動的に機械語に翻訳されていることが分かる。なお、翻訳後によく利用される機能については、ＯＳ（オペレーティング・システム）にあらかじめ記述されていて、コ

ンピュータ内に保持されているので、コンピュータには、「その記述箇所に従え」という旨の命令を与えるだけで、詳細は省略できてしまう。

このようにして、自分が利用する最上層のプログラミング言語の規約だけ知っていればプログラミングできるのである。その規約は、コンピュータは、こう書けばこう動作する、という入出力関係だけを示したブラックボックスによって抽象化されており、その結果としての働き（機能）だけ理解できれば、内部動作まで知る必要はない。「本当に知らなくて大丈夫なのか？」と思われるかも知れないが、例えば実際、我々はＴＶ受像機の内部構造をほとんど知らなくても、ＴＶの視聴ができる。視聴に関わる機能（出力）とその最低限の操作方法（入力）だけ知っておけば、内部動作には関係なくその働きを理解でき、実際に操作してＴＶを視聴できるのである。プログラミングも同様である。

そこで、「プログラミングする」ということは、何か特別な技能を要する訳ではなく、与えられたプログラミング言語の規約に従って、自分がコンピュータにさせたい作業手続きのパーツを組み合わせるだけの作業である。その場合でも、弁護士や裁判官がすべての法律を暗記していないのと同じで、必要に応じて、六法全書を参照するように、その規約のマニュアルを読めば良いだけである。ただし、その規約に従った作業手続きの組合せの方法論を学ぶ必要はあるかも知れない。それはプログラミングというよりは、アルゴリズムの設計方法を学ぶことであるが、これは、具体的なコンピュータを想定することなしに学ぶことができるものである。しかも、高度なケースは稀であり、通常は、既に用意されているパーツを順に並べるだけである。条件分岐や繰り返しといった処理もあるが、むしろ、これだけが本質的な技法であり、ほとんどすべてのアルゴリズムに対応できる。

なお、図2のＯＳの上の層は、左側については、インタプリタ系のプログラミング言語でプログラミングを行っ

た場合を示しており、それ以外の通常のアプリケーション・ソフトウェアの場合は、図の右側のように、そのアプリケーションを作成する時には、それぞれのプログラミング言語で記述されていても、ソフトウェアとしてリリースする前には、先に、機械語で書かれたプログラムに変換されてしまい（「コンパイル」と呼ぶ）、その機械語で書かれたプログラムを販売したり配布したりしている。これをＯＳの上に載せて（インストールして）直接起動する。ウィンドウズであれば、「.exe」という拡張子の付いたファイル名のファイルは機械語で書かれたプログラムであり、これをダブル・クリックすると、ＯＳであるウィンドウズがこれを解釈して、ハードウェアを操作している。

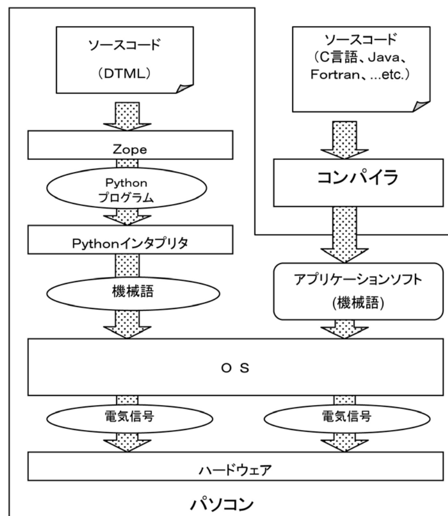


図２　プログラムの層

第二節　ソフトウェア開発過程

一　ソフトウェア開発概要

本稿で立法過程との類似性が指摘されているソフトウェア開発とは、実際にどのような手順で行われているかについて、まず最も典型的なパタンを示す。ソフトウェア開発工程の流れは、要求分析、設計、製作、試験、提供、

保守という順で進められる。本稿は作り方に焦点を当てているため「ソフトウェア開発」と表記しているが、提供や保守までを含む場合は、本来、「ソフトウェアのライフサイクル」と呼ぶ方が正しい。ただし、本稿では、煩雑さを避けるために、「開発」という表記で統一する。

この工程の流れをさらに詳細に示すと、次のような手順で進められている。

〔要求分析〕
(1)要求分析・定義
〔設計〕
(2)基本設計
(3)外部仕様設計
(4)内部仕様設計
(5)詳細仕様設計
〔製作〕
(6)コーディング
〔試験〕
(7)単体テスト
(8)結合テスト
(9)システムテスト

［提供］

⑽リリース（公開、導入、販売等）

［保守］

⑾メンテナンス（障害対応、バージョンアップ等）

なお、より大きな分類法としては、要求分析から内部仕様設計までのプロセスを「上流工程」と呼び、その後、リリースされるまでのプロセスを「下流工程」と呼ぶこともある。このような工程をこの順序で、後戻りが生じないように逐次段階的に作業を進めていく開発方法は、「落水（Water Fall）型の開発」と呼ばれる。これは標準的な開発方法であり、今日でも、基本的にこの進め方で開発が行われるケースは多い。ただし、落水型の手法には古くからいくつかの問題点が指摘されており、代替手段も提案されている。この問題は、立法過程と同じ論点を含むため、本章の第四節で議論する。

本節の以降では、大きく六つに区分した開発工程ごとに、その工程内の各プロセスの作業概要を示す。各工程に対応する立法作業も簡単に付記するが、詳細は次章以降を参照されたい。

二　要求分析

ソフトウェア開発における作業工程の最初の局面は「要求分析」やその分析に基づく「要求定義」を行うプロセスである。その作業内容は、クライアント等から得られる、製作されるソフトウェアに対する要望を実際に開発可能な要求として客観化することである。これらの客観化のためには、通常の自然言語による文書を用いる場合もあ

れば、図表を用いたり、特殊な記述規約のある図式的表記法を用いたり、要求仕様記述言語という種類の人工言語を用いたりすることもある。目に見える作業としては、このようなツールを用いて、要求されている事項を記述することに尽きるが、その過程では、様々な思考支援のためのツールもしばしば用いられる。あるいは、大規模なシステムを設計する場合には、どのようなソフトウェアを作るか、ほとんど決められていないこの段階で、とりあえず、様々なサンプルやプロトタイプを作成して、実験やシミュレーションを試みて、徐々に具体的な要求を探り出す場合もある。さらに、クライアント自身が迷っていたり、クライアント側で競合する要望が存在していたりして、意思決定を行う必要がある場合もあり、その意思決定支援のための方法論や支援ソフトウェアを利用する場合もある。以上のようなツール群の中には、第二章第一節で示したように、そのまま政策立案過程で利用できるものもあり、政策シンクタンクで利用されているものもある。

三　設計

ソフトウェア開発の場合は、設計段階は「基本設計」「外部仕様設計」「内部仕様設計」「詳細仕様設計」に細分化されていることが多い。もちろん、素人プログラマの場合には、設計フェーズを客観化しないで、自己の思考の中でのみ設計を進める者もいるが、複数メンバが関わるソフトウェア開発では、ほとんどの場合、このような明示的な設計を行う作業を経た後で、「製作（本稿では「コーディング」や「プログラミング」という狭い意味で用いる）」の作業段階に入る。なお、このような設計手順の用語法を一部用いた政策法務の文献も存在する。[13]

「基本設計」は要求定義に基づき、システムの実現目的、主要機能、利用環境（情報処理的な環境だけではなく、人・場所・業務も含む）を明示し、さらに、その構成方法や設計・開発方法自体の方針なども明示するものである。

また、特筆すべき前提条件なども示される。

「外部仕様設計」は、利用者から見た、そのソフトウェアの動きを設計するものである。例えば、ウェブ画面で、どのボタンを押したら、どのようなイメージで結果が表示されるのか、あるいは、どのようなデータを入力したら、どのようなデータに変換されるのか、このようなことを設計する。外部仕様を考える時は、ソフトウェア内部の動作をブラックボックス化し、入力と出力の部分だけを考える。このような設計の方法は、ソフトウェア開発に限らず、システムと見なされるもの一般に有効である。この入力と出力で物事を捉える方法は、システム科学の分野でも、「インプット－アウトプット・モデル（入出力モデル）」と呼ばれる最も典型的なシステムの捉え方である。(14) 例えば、ある役場の証明書発行業務を考える。利用者が必要事項を書き込んだ発行申請書を窓口に提出した後、しばらくすると、当該証明書が発行される。この場合、申請書がインプットであり、証明書がアウトプットである。そして、内部でどのように作業が進められているかは、利用者には見えず、ブラックボックス化されている。このような入出力による外側からのシステムの見え方を規定する設計を外部設計と呼ぶ。

「内部仕様設計」は、外部仕様設計が内部をブラックボックス化して考えていたことに対し、その内部での機能実現の方法を設計するものである。この実現方法や内部構造のことは「アーキテクチャ」と呼ばれることもある。ソフトウェアの場合には、コンピュータに用意されている基本機能の組み合わせによってすべての機能を実現する必要があり、原理的に可能であっても、具体的にその方法（アルゴリズム）を提示できなければ、設計は絵に描いた餅であり、思ったようなシステムを実現できない、あるいは、それ以前に、そもそもプログラミングの工程に移ることすらできない。その意味では、ソフトウェア開発にとっては、本質的な設計部分である。ただし、画面イメージやマウス操作なども含む入出力情報が客観化されていれば、コンピュータの行っている作業の本質は、所詮、単

なる「変換」作業であるので、実は、コンピュータの原理からすれば、必ず実現可能なものではある。極端な話、アルゴリズムなど全く考えず、膨大に存在するすべての電気的な入出力の変換パタンを事前に全部列挙して[15]、メモリ上に置いておくことができれば、この内部動作を設計する必要はない。外部の入出力の仕様が決まった段階で自動的に定まる。この点で、内部設計の問題は実際には非常に有効な意義を持つが、本質的には内包的なものであり、どのように簡潔に表現できたか、というレトリックの問題にも近い。最近では、一般の人々が日常的に利用するレベルのパソコンでも、潤沢なメモリ容量と高速な計算能力を備えているので、プログラムの記述は、単なる簡潔さよりも、可読性の高さが重視される傾向もある。可読性を高めるためには、内部仕様が人々に分かりやすく設計される必要がある。経験的には、想定しているシステムが、人々によく知られた事物のメタファの利用によってモデル化されていると可読性が向上する。

「詳細仕様設計」は、内部仕様を詳細に具体化する段階である。このプロセスでは、プログラムとほぼ同じ粒度で書かれた設計書が作成されるが、特定のプログラミング言語に依存しない、「チャート図」と呼ばれる類いの図式が利用されることが多い。また、詳細仕様書では、細分化された最小単位の機能についての入出力の定義を行っている。逆に言えば、詳細設計書に書かれた最小単位の機能を組み合わせると、内部設計で想定しているアーキテクチャが再構成されるようになっていなければならない。

このような各設計段階の作業では、それぞれに対応する仕様書が作成される。むしろ、各作業プロセスは、これらの仕様書を作成するための作業プロセスであると見ることもできる。従って、客観的には、このような仕様書が各成果物であり、次のプロセスに渡されることになる。そのような成果物自体の客観性を向上させることができれば、工程ごとに異なる人々が作業を担当することも可能になる。これは専門による分業につながり、作業の合理化

や効率化の基礎となる。また、仕様書の客観性の向上は、設計段階からの根の深いエラーの混入を防ぐことにも貢献できる。そこで、このような客観性の向上のために、前章の第二節で示したような人工的に設計された、仕様記述言語や図式の記述様式を導入して、仕様書が作成されることも多い。図式的なものは、記号も用いられているが、多少は可読性が向上しており、単に自然言語で書かれている場合よりは、むしろ、分かりやすい場合もある。前章で述べたように、これらのような記述方法は、政策設計に応用できるものもある。

四　製作（コーディング）

物質的な対象の場合、設計の次の段階は、例えば、自動車の開発であれば、自動車を製造し、建築物であれば建設を行うように、実際に物理的な製作を行う段階となる。しかしながら、ソフトウェア開発の場合であれば、この製作の段階は、物理的な実体を用いた製作を行う訳ではなく、プログラミング言語を用いて、設計に従ったテキスト記述を行う作業段階となる。すなわち、設計も製作もテキストとして記述する作業であるから、物理的には、似たようなデスクワークである。このような製造段階としての記述作業を「コーディング」（あるいは「プログラミング」）と言う。元々、コンピュータの記述形式に則ったコードとしてプログラムを作成していたので、このような呼び名が付いたものと考えられる。

設計がいくら優れたものであっても、コーディング段階で、誤りのあるプログラムを記述してしまえば、全く意味がない。この段階の些細なミスには、動作が少々期待と異なる程度の場合もあれば、ソフトウェアが全く動作しない場合もある。従って、ソフトウェア開発の本質部分でもある。この過程での品質の高さがソフトウェアの価値を大きく左右する。この段階のミスは、法令に喩えると、政策内容を十分検討して、立派な法案の要綱を作成して

も、条文作成段階でミスがあり、誤りのある法令を作成してしまうようなものである。もっとも、法令の場合には、プログラムのように機械的に自動処理される訳ではなく、最終形も人の目で確認されるのであるから、この問題はほとんど生じないかも知れない。あるいは、人の解釈を伴って実施されるので、明らかな表現ミスでない限りは、運用の局面で多少の融通が利くかも知れない。しかしながら、法令にミスがあるという状況は、成文法主義を取る国家にとっては、社会システムの根幹を揺るがすことにもなりかねないので、やはり、コンピュータ・プログラム以上に完璧さが要求されるであろう。ソフトウェア開発では、コーディングの段階において、ヒューマンエラーを駆除する様々な工夫が施されており、さらに、そのための様々な開発支援ツールも存在する。これらの方式の中で、政策設計に利用可能なものについては本稿でも紹介する。

前章で示したように、通常のソフトウェア開発では、人手で作成するものは、「ソースプログラム」という、比較的可読性の高いプログラミング言語を用いて記述されるプログラムである。これは「コンパイラ」や「インタプリタ」と呼ばれるソフトウェアによって、実際にコンピュータに処理される「機械語」と呼ばれる電子的な信号の列で表現されたプログラムに変換される。このような比較的可読性の高いソースプログラム用のプログラミング言語を用いても、構造を意識せずに、闇雲にプログラムを記述してしまえば、その可読性は下がってしまう。七〇年代には、可読性の低いプログラミングによる開発が工学的な問題を招くことに注目が集まり（「スパゲティ・プログラム」と呼ばれる混乱したプログラムの問題など）、それらに対応する方法論が研究され、その成果を基にした新たな開発手法やプログラミング言語も数多く登場している。さらに、プログラミング言語ごとの文法構造のチェック・システムなども登場し、品質の向上が図られている。

条文の表記方法も実際には職人技の領域に入っており、詳細な表記まで決まり事があり、一般の人々にとって可

読性が高いとは限らない。そこで、立法過程の最終段階においても、コンピュータの機械語ほどではないにしろ、可読性の高い表記から条文用表記に自動変換することも検討した方が良いと思われる。言わば「条文コンパイラ」を用意するのである。

五　試験

ソフトウェア開発では、試験のプロセスも何段階かに分かれる。通常は、大きく「単体テスト」と「結合テスト」に分かれる。まず、単体テストとは、プログラムの最小単位である、「サブルーチン」や「関数」と呼ばれる処理単位に対して、正常に動作しているか、コンピュータ上でテスト動作させて試験を行うものである。なお、単体テストは、その前段階であるコーディングのプロセスの一貫として含められてしまうこともある。

この最小単位での正常動作が確認されたら、他のプログラムと結合した上で、さらに動作テストを行う。これが結合テストである。組合せ的な問題点を発見することが主目的である。結合テストの際に検出される典型的なエラーは、プログラム間のインターフェースが上手く整合していないことに起因するものである。例えば、二つのプログラムを組み合わせて処理を行う時、最初のプログラムの出力を得て次のプログラムの入力とし、最終的な出力を行うようなケースを考えると、最初のプログラムの出力が文字列であったのに、次のプログラムの入力が数値データのみを予定してプログラミングされていると、当然、エラーとなる。これは、事前に互いの入出力のタイプや書式を一致させる作業を怠ったり、互いに誤解していたりする場合に生ずる現象である。両者が接する部分で整合されていない、という意味で、「インターフェース（界面）が整合していない」という風に表現される場合もある。

試験の最終段階としては、さらに、開発したプログラム全体を一つのソフトウェア・システムとして結合し、現

実の利用環境と同様の状況を設定して試験を行う。これが「システムテスト」と呼ばれるテストである。
立法過程の場合には、試験的に実社会での実験を行うことが困難なので、議会に諮ったり、パブリック・コメントを集めたりすることで、このような試験手続きに対応していると解釈できる。

六　提供

ソフトウェアを実際に利用するユーザに提供するための「提供」過程について示す。これは、ソフトウェア開発という「もの作り」の側面からすると後始末の話のようであるが、ソフトウェアはこの後も保守や改良を伴うことが常であり、広く捉えれば、このプロセスも開発の過程と考えることができる。この提供過程はソフトウェアの場合、商業的な製品ならば、通常の商品同様に販売ルートに乗せるだけであり、フリー・ソフトウェアとして公開する場合ならば、ウェブ等の所定のサーバに置いておくことによって実施される。
以上の場合は、ユーザから見れば、先にソフトウェアが存在して、それを利用するケースである。しかしながら、ソフトウェアの中にはオンデマンドで、先に具体的なユーザの要求があって、そこから開発され、そのユーザに特化されて提供されるものもある。このような開発形態の場合は、ソフトウェアの完成後に、実際に利用される現場に出向き、その現場環境のコンピュータ・システムへの導入作業を行う必要がある。特定の環境向けのソフトウェアなので、特定の組み込み操作が必要な場合もあるし、そのソフトウェアに関する書籍等も存在しないため、講習会を開くこともある。このようなプロセスのすべての作業がこの「提供」の段階の作業であると考えられる。
立法過程においては、この提供段階は公布に対応し、物理的には、官報や広報誌に載せるだけでなく、現在では、ウェブなどを通じて、電子化された公開がなされることがほとんどであり、その点では、ソフトウェアの公開方法

と重なる部分が多い。

七　保守

ソフトウェアの「保守」のプロセスで行われる作業は、さらに、「障害対応」「設定変更」「機能追加・変更・廃棄」の各作業に分かれる。もちろん、これらの複合的なものもあり得る。なお、これらの補足的な作業として、開発時と同様に要求分析や設計、あるいはコーディングの各作業が付随する場合もある。

「障害対応」とは、ソフトウェアが意図通りに動作しない場合の対応である。導入や操作のミスによるものもあり得るが、間違ったコーディング部分（「バグ」と言う）が含まれる場合もある。さらに、そもそも、要求分析段階や設計段階での誤解による根本的なエラーが発生する場合もある。これらに対応することが障害対応である。このような障害が発生すると、単にプログラムを修復するだけでなく、その修正履歴を記録に残すような作業も必要であり、さらに、マニュアルや仕様書など、関連する様々なドキュメントをソフトウェアの改修に合わせて修正する必要もある。バグによる障害対応の作業を「デバッグ」と言う。そのような作業を補助する「デバッガ」と呼ばれるツールにも様々なものがある。典型的なものは、実行時のコンピュータの内部状況をリアルタイムでモニタしたり、ログとして出力したりするツールと、プログラム実行時に、手動でプログラムのワンステップごとに停止させながら実行を進めるツールがある。残念ながら、立法支援を考えた場合、機械的なツールとしてこのようなものを実現することは難しい。人々の様々な行動をシミュレートする必要があるからである。それでも、一部の行動パタンに限定したツールであれば可能かも知れない。なお、デバッガの流用は困難でも、デバッグ方法論については、立法支援の方法論として流用可能な場合がある。例えば、問題箇所の同定方法や影響を局所化する修正方法などで

ある。

「設定変更」とは、ソフトウェア自体を改修する訳ではないが、ソフトウェアに事前に与える設定値や導入方法を変更する場合である。例えば、コンピュータやオペレーティング・システムが変更された場合に、それに対応して、適正なチューニングを行う場合である。あるいは、当該ソフトウェアが登録制で利用されている場合、その登録者情報を変更する場合も一種の設定変更と考えられる。しかしながら、法令を扱う場合は、同じ法令のままで運用方法を変えるということは、明示的には起らないと考えた方が良いであろう。たしかに、司法の局面では、解釈の変更という形で、ソフトウェア工学に対応することが起っていると考えることもできるが、本稿の対象範囲を越えるため、本稿では扱わない。

「機能追加・変更・廃棄」とは、対象ソフトウェアを変化させることである。法令の改廃と同じである。先に述べた障害対応によって、追加・変更がある場合もあり得るし、不要になって、コンピュータ上からアンインストールされる場合もある。このような作業プロセスで重要なことは、その変更の履歴が残されていて、必要に応じて直ちに参照できるように整備されていることである。このためのツールも存在する。その多くは、プログラムがテキストであることから、法令の改廃作業についても流用可能なものであり、条例の改廃の場合には、実際に利用されているソフトウェアもある。[16]

最近では、これらの他に「リファクタリング」という手法も注目を集めている。これは、とりあえず、正常に動作するソフトウェアを作成しておき、その後、プログラムの書き方を整理していく方法である。外側から見た機能を変えずに内部を書き換えることで、新たな障害を未然に防いだり、別のソフトウェア作成のパーツやテンプレートとして再利用したりすることに貢献できる。もちろん、可読性を良くすること自体も重要な目的である。法令の

場合でも、片仮名表記を平仮名表記に改めるなど、実質的な内容を変えない改正を行うことがある。このような場合は、リファクタリングの一種と言えよう。この処理も基本はテキスト処理に帰着できるので、立法過程においても、これらの方法論やツールを流用できる可能性がある。また、要綱を変えずに法令案を書き換えて整理する場合にも、この技術が流用できそうである。

第三節　立法過程との比較と類似性

前節で示したようなソフトウェア開発の過程を立法過程と並べて見ると、各フェーズの作業が図3のように対応付けられる。このように対応付けられた各作業は、抽象的には類似した思考方法やテキスト操作作業を行っていると考えられる。比較内容や各対応関係については、次章以降で詳細な説明を行う。

本節では、このような類似性の根源について言及しておきたい。これは両者をシステムとして捉えることによって立ち現れるものである。「システム」とは、システム科学的に言えば「構成要素の集合とその関係から成る総体として認識された」ものである[17]。システム科学の古典の一つとされるサイモンの著書『システムの科学』では、その著作で扱う対象を「人工物」としている[18]。システムとは実体的な要素を持つ場合が多いが、システム自体は人間によるものの観方によって形成される認識論的な存在であり、何らかの目的、役割、あるいは働き（これらを「機能」と呼ぶことにする）を持ってまとまって形成され、あるいは動作するものである。その各要素は「関係」という直接目に見えない、実体的ではない、解釈によって生まれているものによって結び付けられている。例えば、自動車をシステムと考えた時、その部品である両タイヤ部分はシャフトによって結合されており、実体が要素を関係

付けている、と思われるかも知れない。しかしながら、物理的につながっている、と言うことであれば、空気を媒介にしてつながっているとも言える訳であるから、なぜ空気ではなくシャフトなのか、それを説明するには、やはり、人間による解釈や判断が入り込まざるを得ない。そのような意味で、システムという場合には、必ず「人間」が対象をどのようなシステムとして「認識」しているか、という点が本質的になる。こうして、システムとして見ることのできるものは、何らかの「機能」を持って存在していることになる。そして、ソフトウェアも社会制度も、システムの一種であり、このような機能的な側面を持つ存在である。そこで、システム作り一般の手法がそもそも利用できるのである。高度に専門的なプロセスでなければ、例えば、要求分析などのプロセスで利用される思考ツールや支援ソフトウェアは、実際、ソフトウェア開発以外でも利用されるものであり、むしろ、元々、シンクタンクなどの業務のように、公共的でないものも含めて広く政策を考える際に利用されるものが数多く存在する。前述したように、これらの中の典型的なツールとしては、意思決定の数理的手法やそのための支援ソフトウェアなどがある。そもそも、政策や人々の行動指針を決定するために使われていたものをソフトウェア開発に持ち込んだだけであるから、むしろ、立法作業への応用の方が、それらツールとの親和性は高いであろう。

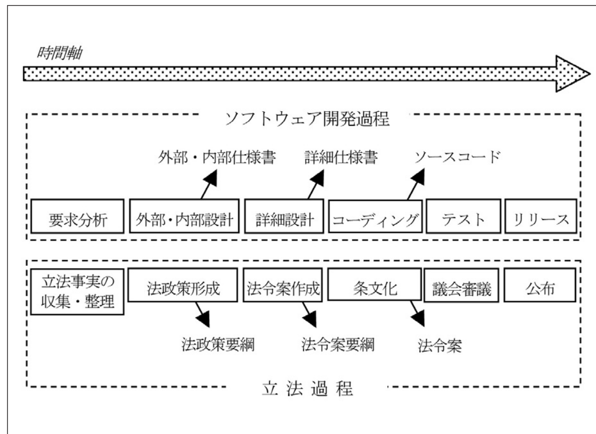


図３ ソフトウェア開発過程と立法過程の比較

またシステム一般ということで考えれば、例えば、マーケティング分野における流通ルートの構築やビジネスモデルの開発も一種のシステムの開発であり、その開発プロジェクトの運営方法論については、広く共通する部分がある。この点については、世界的な共通化の動きもあり、ＰＭＢＯＫ[19]と呼ばれるプロジェクト管理の方法論に関する世界的な標準も登場し、その知識に関する認定試験まで存在する。こういった手法は汎用的なものであり、当然立法過程にも導入可能なものが多い。このようにシステム一般の観点から見れば、ソフトウェア開発と共通する部分も数多くあり、ソフトウェア開発に限らず、流用可能な方法論やツールも数多く存在する。そして、システム的観点から見れば元々同じカテゴリの対象なので、流用が効く可能性が元々高いのである。ただし、本稿では、流用可能なものの中でも、特にソフトウェア開発との類似点に主眼を置いているので、システム一般やプロジェクト運営についての議論は、省略させて頂く。

第四節　非落水型の開発方法

第二節でも触れたが、ソフトウェア開発の標準的な手順に沿って、後戻りなく作業を進めていく手法は「落水型」と呼ばれる。これに対して、本節では、そのような手順を踏まない、言わば「非落水型」の手法について示す。

古くから落水型では後戻りができない点が問題視されている。あるいは、後戻りした場合、コストがとても大きくなってしまうため、無理にでも、前に進めることが多くなり、それを避けるために、設計段階でも、後の実現段階の問題を正確に予測しなくてはならず、初期の工程から慎重になり過ぎ、進捗も悪くなる。また、後になってからの責任逃れのために、言い訳のための余地を冗長に含ませた機能を設計に盛り込む点などの問題も指摘されてい

る。それでも、この手法で開発を進めることで、大規模なプロジェクトの場合、管理が行いやすくなるという利点があり、その観点から現在でも採用されるケースは多い。なぜ管理が行いやすいかというと、後戻りがないため、進捗状況が把握しやすい点、さらに、各サブ・プロセスのアウトプットが次のサブ・プロセスのインプットとなることで、作業プロセスの間で、一旦生産物が作成されて、開発作業が客観化される点、これらがその理由である。

しかしながら、小規模な開発や、緊急を要するような開発においては、落水型のような手間のかかる手法では、贅沢過ぎてしまい、予算オーバーになるとか、時間切れになるとかいった、ビジネスとしては致命的な問題を引き起こす場合がある。そこで、このような場合には、落水型のカテゴリに入らない手法が採られる。これらは、例えば、そもそも後戻りを許したり、短い期間で落水型の最初から最後までを一旦、通して行い、それを何度もスパイラルに繰り返したりするような手法である。そのような手法の多くは「アジャイル（agile）型」と呼ばれる開発モデルに分類される。厳密な定義がある訳ではないが、二〇〇一年に著名なソフトウェア開発方法論者が決起人となって結成されたアジャイル・アライアンスの宣言によれば、アジャイル型の開発とは、次のような特徴を持つ開発手法とされる。[20]

- コミュニケーション重視
- 動作するソフトウェア
- ユーザとの協力
- 変化への対応

いずれのアジャイル型の手法においても、「軽量さ」「俊敏な適応性」「無駄のなさ」などが強調されている。システムや工程を小規模化あるいは軽量化し、さらに、目的に対して、俊敏に適応して製造するような方式であることが所以である。ポッペンディーク夫妻の著書による解説では、[21]トヨタ社の自動車製造の方法論から多くのヒントを得ており、特に、様々な観点からの「無駄をなくす」という考え方から大きな影響を受けている。例えば、在庫を抱える無駄をなくすために、顧客の要求に「適応」して、オンデマンドで製造したり、あるいは、要求が実際に存在する機能だけに制限して設計・製造したりして、無駄な製品や機能を作成しないようにする、という手法もその一つである。この他にもアジャイル型に近い最近の開発手法の中には、第二章第七節で紹介した「テスト・ファースト」という考え方がある。

なお、アジャイル型の特徴が欠点となっている点も指摘しておく。アジャイル型の手法はコミュニケーション重視の面が強いため、ある一定規模より開発チームの人数が多いような大規模開発には向かないことが、当初から多くの人々によって指摘されている。アジャイル型開発が提唱されたのは一九九〇年頃であるが、既に、七〇年代には外科医療チームの手術時の作業のように、少人数で一人の優秀な執刀医を中心として作業を行う小規模チームのような構成が、ソフトウェア開発にも効率が良い、と指摘されており、[22]アジャイル方式と同様の軽量化や適応性がその特徴には含まれていた。しかしながら、そのような優れた方法と考えられるものでも、当時の大規模なソフトウェア開発というものには対応できず、従って、落水型のように、フェーズ間の中間の仕様書などの文書化が鍵を握るような開発方式を容認せざるを得ないことが指摘されていた。すなわち、アジャイル開発の利点も欠点も本質的には大型コンピュータ全盛の七〇年代から既に認知されていたと言える。そこで、アジャイル方式の問題は最近の問題でなく、ソフトウェア開発に古くから内在した問題であるとも言える。従って、両方式の利点・欠点に留意

して、ケースバイケースで方式を適用すべきであり、どちらかの方式を絶対視することは危険である。

このような非落水型の方式と同様の作業は、立法過程でも、しばしば行われている。それは、小規模な自治体の場合である。人的な面で費やすことができるコストには限界があり、それがさらに時間的コストに響いてくるため、結局、時間的な点でも、作業の手間の点でも、軽量化を図らざるを得ない。実際、自治体の担当職員にインタビューしてみると、立法事実を調べて政策を模索して、要綱を作成して、条例化して…という、言わば落水型の順番ではなく、最初の検討会議（会議の名称は自治体によって異なる）の場からいきなり、条例案とともに説明資料として書かれた要綱を提出して、チェックを受けるという自治体もある。このように原課とその会議との間での条例案のやり取りを繰り返すことによって、徐々に修正されて完成する、という手順によって進めるのである。第二章第七節でも示したが、その検討会議がテストになっており、広義のテスト・ファーストと言えるだろう。そこで、数多く存在する小規模自治体の法制執務支援を考えると、落水型の仰々しい方式だけでなく、非落水型の開発モデルを参考にした方式についても検討を加える必要があり、本研究では、状況に応じた提案を行いたいと考えている。ただし、その非落水型の多くの方式でも、個々の短いプロセスや一回のやり取りの中では、要求分析や設計などの作業が行われることもあり、落水型と同様のものを流用できる。そこで、本研究では、非落水型を意識しつつも、基本的には落水型の順序に従って、各段階の作業プロセスについて検討を進めることにする。

注

(1) 角田篤泰、齋藤大地、外山勝彦「ソフトウェア開発過程との類似性に基づく立法支援システム」人工知能学会第二三回論文集（二〇〇九年）2F2、4・1－4・4頁。以降、「JSAI」で参照。

⑵　角田篤泰「法律事項に基づく法令・例規作成の方法論─法政策の設計と法制執務を結ぶ論理的基礎付けの試み」名古屋大学法政論集二三二号（二〇〇九年）七五─一一九頁。以降、「方法論」で参照。
⑶　前掲、拙著ＪＳＡＩ参照。
⑷　ご協力頂いた方々は、自治体では名古屋市、山形県酒田市、神奈川県座間市、同高座郡寒川町、佐賀県杵島郡大町町、同三養基郡みやき町、中央省庁では厚生労働省、開発業者では㈱クレステックの方々である。
⑸　Saaty, T. L.,"The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation", McGraw-Hill（1980）.
⑹　前掲、拙著「方法論」八二─九一頁。
⑺　Ｇ・Ｊ・マイヤー著、国友義久他訳『ソフトウェアの複合／構造化設計』近代科学社（一九七九年）。以下『構造化』で参照。
⑻　前掲、マイヤー『構造化』五〇─五四頁。
⑼　加賀山茂、松浦好治編『法情報学』第二版補訂版、有斐閣（二〇〇六年）。
⑽　例えば、㈱クレスッテク製の「じょうれいくん」というソフトウェアがある。このシステムについては http://lawinfo.crestec.jp/joureikun/product_jo.html 参照。以降、「じょうれい」で参照。
⑾　例えば、RCSやCVSというソウトウェアがあり、ＵＮＩＸと呼ばれるＯＳには標準で添付されていることが多い。
⑿　http://www.zope.org 参照。
⒀　礒崎初仁「条例をつくる⑵」『ガバナンス二〇〇五─2』ぎょうせい（二〇〇五年）一二八─一二九頁。
⒁　高橋信吾『システム学の基礎』培風館（二〇〇七年）。以下『基礎』で参照。
⒂　組合せ的爆発となり、実際には時間もメモリも足りなくなってしまう。
⒃　前掲、「じょうれい」参照。

(17) 前掲、高橋『基礎』五頁。

(18) Ｈ・Ａ・サイモン著、稲葉元吉、吉原英樹訳『システムの科学』第三版、パーソナルメディア（一九九九年）。

(19) A guide to the project management body of knowledge（PMBOK guide）3rd ed., Newtown Square, Pa.:Project Management Institute（2004）.

(20) http://www.agilemanifesto.org/参照。

(21) Ｍ・ポッペンディーク、Ｔ・ポッペンディーク著、平鍋健児他訳『リーンソフトウエア開発』日経ＢＰ社（二〇〇四年）一八頁。

(22) Ｆ・Ｐ・ブルックス著、滝沢徹他訳『人月の神話』ピアソン・エデュケーション（二〇〇二年）二五−三三頁。