ソフトウェア工学との類似性に着目した立法支援方法（三）

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
第一節　ソフトウェア工学における要求分析の手法
一　要求分析の目的
二　要求獲得手法の概要
三　シナリオに基づく要求獲得
四　要求定義の手法
第二節　政策検討段階への応用

（以上二三五号）

一　要求分析手法の応用
二　シナリオを用いた立法過程への要求分析
三　要求表現方法の応用
第三節　立法事実の収集
第五章　設計―要綱の作成
第一節　ソフトウェア開発における設計手法
第二節　オブジェクト指向に基づく設計手法
一　オブジェクト指向パラダイム
二　オブジェクト指向のプログラム例
三　仕様記述方法
第三節　構造の表現
第四節　政策立案への応用
一　要綱作成過程への応用
二　要綱表現への応用
第六章　製作―法令・条例案の作成
第一節　ソフトウェア開発における製作
第二節　立法過程における製作
一　条文作成の支援

（以上二三六号）

二　条文作成の自動化
第三節　非落水型開発の応用
第七章　試験―意見収集と議会審議
第一節　ソフトウェア開発における試験技法
一　ソフトウェア試験の目的
二　ソウトウェア試験の種類
第二節　作成された法令の検査
一　立法過程における試験
二　立法過程におけるテストケース
三　高階テスト技法
四　立法過程における試験支援
五　試験としての議会
第八章　提供と保守―公布・施行・改廃
第一節　公開とバージョン管理
第二節　法制執務支援システムの実例
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）

第五章　設計――要綱の作成

第四節　政策立案への応用

一　要綱作成過程への応用

本章でこれまでに説明を行ったソフトウェア設計技法を政策立案過程に応用する方法について提案することが本節の目的である。ソフトウェア開発における設計段階の目的は、要求定義された問題に対する解法を与えることであり、そのアウトプットは各種の仕様書である。これを政策立案過程に対応付けるなら、この過程の目的は、立法事実などによる政策への要求に対して、対応策を与えることに相当し、アウトプットは各種の要綱に相当する。そこで、この対応策の与え方をソウトウェア開発の手法から応用することが本節の目的である。なお、本節では、この対応策の与え方（法政策の設計）自体とその結果を表現する方法については分けて論じ、まず、前者の設計作業方法への応用、すなわち、アウトプットである各種の要綱の作成過程への応用方法について示す。

ソフトウェア開発では、まず、要求を実現するために、システムの持つ機能やパーツを洗い出し、パーツの関係性を明確にする。すると、その様子はシステム内部のサブシステムの結び付きとして捉えることができる。その後、それらの各サブシステムをブラックボックスとして捉え、それぞれのインプットとアウトプットをはっきりさせる。これを一旦確定させた後で、今度はそれらの各ブラックボックスと見ていたサブシステムの内部に目を向け、そのブラックボックスの内部にさらに詳細な機能やパーツを洗い出し、同様の作業を操り返すことによって、当該サブシステムの機能を実現するように表現を試みる。このように段階的にブラックボックスの内部に目を向けて詳細化

を図り、設計を進める。この詳細化は、コンピュータが実行できる命令や既存のプログラムを呼び出して実現できるレベルまで進める。この最も詳細なレベルでは、コンピュータへの命令や既存プログラムが最小単位のブラックボックスに当たる。これらは既存のものなので、内部を知る必要もなければ、もちろん、設計する必要もなく、機能だけ確認し、それを利用すれば良いだけである。よく使われる機能のほとんどは、ウィンドウズのようなＯＳによって、既存プログラムとして提供されている。従って、プログラマはコンピュータ自体の命令セットを知る必要はなく、ウィンドウズの提供するプログラムの外部仕様を調べておくだけで良い。さらに、各プログラミング言語処理系には、ＯＳに依存しない形で頻繁に利用される命令群が、対応する既存プログラム群の入ったパッケージ（「ライブラリ」と呼ぶ）として用意されており、その命令名とパラメタを与えるだけで利用できるしくみになっている。従って、ソフトウェアの設計の場合であれば、このプログラミング言語で提供される基本命令（「プリミティブ」と呼ぶ）とライブラリとして提供される命令のレベルまで詳細化すれば良いことになる。実は、この基本命令は、どのプログラミング言語でも、大抵同じようなものが用意されているので、通常は、適当なプログミング言語を意識して、それを仮の典型的な言語と見て、そのプリミティブのレベルまでを設計すれば良い。ただし、この時、アルゴリズムなど、本質的な解法自体は、このような詳細化さえ行えば浮かんでくるというものではないが、このような詳細化による問題の分割によって、各部分は、かなり簡単な問題に置き換えられる。そして、大抵の場合、プリミティブの組合せ、あるいは、場合分けや繰り返しなどを含めた組合せに帰着される。

　政策立案でも、ソフトウェア開発と同様の手順で作業を進めることができる。もちろん、効果的な解決方法の着想を得ることは、ソフトウェアの設計と同じで、段階的詳細化によって直ちに可能となるものではない。しかしながら、複雑な政策実現も、原理的には基本的な行為や実体のみの組合せに還元して構成できるはずである。そこで、

政策も段階的に詳細化することでその構造を示すのである。例えば、「こうあって欲しいならば、その前に、これとこれが必要であり、そのためには、さらに細かく見れば、これとこれが必要だ」という具合に、段階的に必要な事柄を細かく見ていった時、プリミティブとなるような最小単位の要求事項が浮かび上がる。それらを組み合わせて政策実現の状態を表現するのである。

さらに、ソフトウェア開発との差異から、例えば、次のような配慮が必要である。

・上位法や競合法規による制約から、法的に可能なプリミティブで構成する必要がある。コンピュータの命令であれば、命令されるまでの状況（文脈）の影響を強く受けないため、同じ入力による同じ命令ならば、同じ処理結果が返ってくるが、人間の世界、特に、法的な世界では、一見、同じ条件での同じ行為であっても、隠れた条件や制約が存在し、それまでの文脈の影響を強く受けて、結論が変わることがしばしば起る。例えば、未成年であったとか、違法性が阻却されたとか、他に特別法があった、などという場合である。

・プリミティブが厳格に決まっている訳ではなく、多くの場合、解釈に依存する余地ができてしまう。意図的な場合は構わないが、不用意にこのような解釈の曖昧な部分ができることには注意すべきである。

・単純に行為を記述するのではなく、当該行為をさせたり、禁じたりするような記述がプリミティブとなることが多い。論理的な関係が複雑になることがあるので、注意が必要である。

こうして、ある程度の配慮は必要であるが、政策立案の場面でも、このように段階的詳細化によって、トップダウンに設計を進めることは可能である。実際には、現場でもこのように作業が進められているケースがあるかも知

れない。この設計過程で用いられる各プリミティブは、様々な実体や既存システムであるから、結局、実体間の関係を描いたシステムの図として表現可能である。すると、ソフトウェア開発におけるオブジェクト指向の考え方やその分析手法、あるいは、ＥＲ図やインスタンス図のような表現方法の流用も可能である。このような図を設計段階で描いたり、イメージしたりしながら考察を進めることで、段階的詳細化の作業を円滑に進めることができるであろう。そこで、設計段階のアウトプットとしてではなく、作業の中間で用いるツールとして、これらの表現方法を導入することを提案したい。なお、ＥＲ図の例については、第二章で示した図1を参照のこと。このような図を利用して、制度や手続きの内部を記述する際に、例えば、図1の場合ならば、さらに、申請書の中身や欠格条件などを徐々に詳細化して表現することで段階的に政策の状況を設計できる。ただし、ＥＲ図で表現できることには限界がある。そもそもＥＲ図は、データベースの設計分野で用いられることが多く、「関係」と「属性」にはっきり分けてしまうことを前提としている。しかしながら、「属性」の値は単なる数字や記号であるとは限らない。他の実体を示している場合もある。すなわち、「属性」と言うものは、それ自体、ある種の「関係」を表しているので、柔軟に表現するためには、「属性」として実体に内在化させてしまうのではなく、「関係」として明示的に実体の外に出す方が柔軟な表現ができる。つまり、ＥＲ図で、一切、「属性」を使わないで表現を行うことと同一になる。実は、知識工学の分野の古典的な知識表現形式である、「意味ネットワーク」と呼ばれる表現方式では、ＥＲ図のように「属性」と「関係」を区別せず、すべてを「属性」あるいは「関係」として表現する。こうしておけば、その「関係」を「述語」と見なすことで、述語論理式にも容易に変換できる。ただし、意味ネットワークでは、実体間の「関係」ばかりではなく、概念間の「関係」も表記している。そこで、概念と実体を明確に区別する必要もあるし、「関係」の中には、例えば、「親」のように、一見概念のようなものも含まれる。もちろん、「親」は概念で

あるとも言えるが、実体や概念の関係を描く中で、ある親子の各個体について、「親」はその「子」に対する一種の役割であり、「子」に対する「関係」として表現される。実際、「親」であるか否かは状況によって決まる、一種の役割なので、自分の子供に対してはたしかに「親」であるが、同時に自分の両親に対しては「子」であるので、その個体（その人）が固定的にその概念（「親」）に属すると考えることはできない。この「親」のような概念に対しては、少なくとも図式の中では、概念と言うよりも「関係」とみなすべきである。なお、この「親」や「子」のような役割は、関係の一種で「ロール」と呼ばれている。

この意味ネットワークの記法に従って、許可制度の一部を詳細に表現してみると図８のように描くことが可能である。なお、図１は制度というよりも作業局面を表現していたので、そもそも描いている対象が図８とは若干異なる。しかしながら、両図式を比較すると、意味ネットワークによる表現の方が、広く表現可能であるので、個別ケースだけでなく制度自体の概念的な部分も描けるのである。この図では、「事業」概念とその個体である「◇◇事業」が概念とその実例の関係を表しており、この場合は特別に IS-A と言う表記を使う。この他にも図の中には記入されていないが、山田太郎は自然人であることや自然人が生物であることなども記入できる。これらも、IS-A

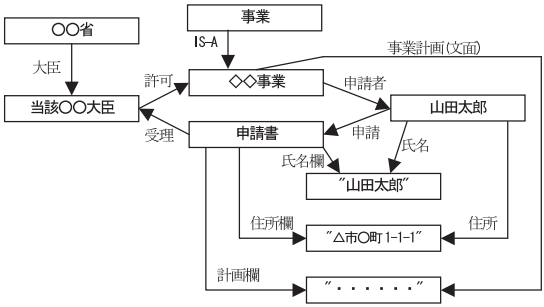


図８　意味ネットワークによる許可制度の図（一部の例）

という関係で表記される。この意味ネットワークでは、基本単位を〈概念、関係、値〉という組と考えることから始める。このような三つの要素からなる基本単位で表現する方式は、現在のＸＭＬ系の知識表現の一つであり、ウェブでデータを扱う際の標準になりつつあるＲＤＦでも採用されている。(1)図８では、各矢印前の四角部分は概念を表し、各矢印上のラベルは関係を表し、各矢印の先の四角部分は値を表す。それぞれの値もまた概念として矢印の起点となることができる。ただし、ここで言う「概念」を表す四角部分には、本来「個体」として考えるものも含まれており、実は、相対的な関係の中で「概念」か「個体」か、という区別が判断されるので、この点が問題視されることもある。この図では、この場合の「申請書」は個体のはずであるが、このように単に「申請書」とだけ書いてしまうと、概念か個体かは判別できなくなる。だからと言って、一々個体に固有名詞やＩＤを割り振るのも不便である。そこで、実用上は、例えば、個体部分は円形にする等、区別するための何らかの表記を導入すれば良い。

これらの図式は、あくまでも静的な状況を表現したものであり、この例では、許可が与えられている状況を表している。従って、この図の時点から変化したものや別のケースを描く場合については、一緒に描かないで、別途、描く必要がある。例えば、許可が与えられなかったケースや、事業内容が変

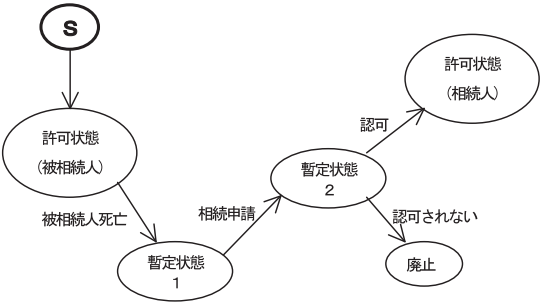


図９　状態遷移図の例

化したり、申請者が変化したりする場合である。あるいは、手続きが複雑で、いくつもの段階を経る場合には、このような静的な図だけでは十分に表現することができないこともある。そこで、このような静的な図に加えて、状態が移り行く様子を図示する「状態遷移図」なども必要になるであろう。状態遷移図もソフトウェア開発ではポピュラーな図式表現方法であり、それを流用すれば良い。なお、状態遷移図とは図9のようなものである。この図においては、起り得る状態は丸印で表現され、それらの遷移の様子は矢印で結ぶことで表現されている。各矢印のラベルは、その状態間の遷移を起こすきっかけなる行動やイベントをラベル化したものである。こうして、ある時点のスナップショットとしての静的な状態を描く意味ネットワーク等の図とそのような各状態を状態遷移図で時間的な順序関係も考慮して結ぶことで、両者を併せれば、時間的・空間的な政策の状況を表現することも可能となる。これらを用いて、段階的詳細化を図ることによって、政策立案過程の作業支援に貢献できると考えられる。

二　要綱表現への応用

本小節では、ソフトウェア開発における設計段階のアウトプットである仕様書の表現方法を法政策の設計における要綱記述のための表現方法や形式に生かすことについて考察を行う。

先に示したように、ER図や意味ネットワーク、あるいは状態遷移図などは、政策設計の作業局面にも利用可能であり、行為の手順や関連する概念構造などを考察するためのツールとして利用できる。しかしながら、これらは、作業中のツールとして提案されたものであって、法令案要綱などのように法制度や条文形式を意識した要綱記述が想定されたものではない。もちろん、これらの図を要綱記述形式としてしまう、という方法もあるかも知れない。たしかに、本稿の究極の目的からすれば、ユーザは要綱の代わりに図だけを描いておき、後はコンピュータが自動

で条文を作成する、ということも目指したい。しかしながら、それでもコンピュータ内部では、図の表現のまま蓄えられている訳でなく、また、自動で変換にするにしても、細部まで記述規則を決めておく必要もあるため、何らかの形で、コンピュータで扱うことができるレベルの記述形式を考えておく必要がある。ただし、これはユーザ自体が直接扱わなくても良いものなので、システムのインターフェースが分りやすい表示や簡単な操作を実現する設計になっていれば、厳格で複雑な規則であっても構わないであろう。そこで、大きく二つの提案を行いたい。それは、述語論理式による記述とＸＭＬ表記による記述である。なお、これらは、その表記方法をうまく設計し、必要に応じて多少拡張するだけで、相互に変換できるようにすることも可能である。そこで、最終的にどちらを採用するかは、実現方式に依存した判断をすれば良い。

まず、述語論理式の場合であるが、通常の述語論理式に加え、各命題にはＩＤを割り振る記述方法にする。また、ＸＭＬとの互換性を図りやすいように、別の箇所に各命題や各述語のメタ情報を保持させることにする。メタ情報としては、各命題の様相的な情報、例えば、義務や禁止、場合によっては、時間的な順序関係などを与えておく。さらに、述語に対するメタ情報としては、各述語の各引数の持つ意味を定めて、その意味を表記するラベルを記しておく。このラベルは「引数ラベル」「属性名」などと呼ぶことにする。以上の表記をまとめると、例えば、「甲さんが△△市長に○○事業許可の申請を申請書Ａによって申請する」をＩＤ番号1として、次の例のように表記することができる。

対象の命題の記述例：
1:　許可申請する（甲, △△市長, ○○事業, 申請書Ａ）

命題のメタ情報の記述例：
1: 義務
述語のメタ情報の記述例：
許可申請する: 申請者, 提出先, 事業名, 提出物

述語の引数ラベルの導入については、ＸＭＬとの互換性を維持することが主要な意図であるが、ＩＤの導入については、二つの背景となる事情がある。そこで、これらについて説明する。

一つは、条文として書かれる文は、単なる事実の表記だけではなく、多くは「～できる」「～してもよい」「～しなければならない」などの様相的な意味を持つ言明であり、また、仮定の記述や時間的な関係の表記が必要になる場合も多い。さらに、例外記述なども行わなくてはならない。そこで、絶対不変の事実の表記を前提とした数学的な意味論が与えられている通常の述語論理式の考え方や表記法では、法律学的な分野に対しては、不便なケースが生じてしまう。実際、「人工知能と法」の分野では、この問題に当初から主要な関心があり、現在に至るまで様々な提案が行われてきたという経緯もある。そこで、推論のメカニズムや数学的な解釈は後付けで行うにしても、最低限の情報を与えておけるような枠組みが必要である。これが、各命題にＩＤを割り振って付加情報を与えることで実現できるのである。

もう一つの背景となる事情は、高階の表現の問題である。この問題は、文を要素とする一段高い視点から見た文のようなものである。「『これはペンです』と言う文は平常文です。」のような表現である。特に問題視されるのは、簡単に自己言及的な記述を許してしまうことである。この場合、真偽不明の命題が存在できてしまうことがある。

例えば「『クレタ人は嘘つきだ』とクレタ人が言った」と言うような言明である。この言明を命題として真偽を考えると、クレタ人が嘘つきだとしても、正直だとしても、この命題全体では常にその逆になってしまう。こうした問題を避けるためには、記述対象の世界を層状に分けることで対応できることが知られている。この対応策の場合、確かに機械的に推論を進める場合には、別の世界のこととして取り扱われるので、問題のある結論は出てこないが、意味的には同じ対象を違うものとして扱っている分、推論できる能力が劣ってしまう。しかしながら、そもそも本稿で行おうとしているのは、コンピュータが自動的に行う推論機構を提案することではなく、客観的に政策や法的状況の知識を記述する方式を与えることであり、最低でも、条文作成時に用いられる要綱を記述するための形式的表現方法であるから、層を分けておいても、人手で解釈するときに認識可能であれば良い。そのための方法が、やはり、命題にＩＤを割り振っておき、対象として命題を扱う時は、当該命題を直接操作しないで、それは別世界のものとして、その命題のＩＤを操作対象にすることで対応するものである。先の例であれば、「その申請を△△市長が許可する」の場合は次のように記述することができる。

許可する（△△市長, #1）

ここで、「#」記号は、参照している命題のＩＤの記述であることを示すためにＩＤ番号の前に付されるものである。

次にＸＭＬ記法を導入する場合であるが、最近のＩＴ分野の動向からすれば、非常にポピュラーなＸＭＬ記法を導入することは、それだけでも意義がある。すなわち、互換性も高く、ＸＭＬ向けに作成されたソフトウェアも多

いからである。特に、第二章第二節でも言及した、最近注目されている、セマンティック・ウェブ上での知識（オントロジー）の記述に使われているＯＷＬ[(2)]と言う人工言語のように、意味論のはっきりした言語を用いたり、あるいは、それらをベースにした変形版を再設計して用いたりすれば、意味論や推論の問題も解消できる可能性が高い。ここでは、第二章第二節で例示したような、最低限必要な表現を組み込んだＸＭＬによる表現例を示す。最低限の表現なので、まだ意味論や推論のことには触れていないが、ＥＲ図等の図式との互換性は保たれている。この表現例は、今後の研究・開発の進展に伴って、逐次改良が行われるものなので、この具体的表現方法に固執して提案する訳ではない。ここでは、先の述語論理式の例と同じものをＸＭＬで表現している。

XML形式による表現：

```
<申請 制度＝"許可" 事業名="○○事業" 提出先="△△市長">
    <申請者>甲</申請者>
    <提出物>申請書A</提出物>
</申請>
```

詳細な解説は省略するが、述語論理式と表現を比較すると、ＸＭＬでは、述語論理式の各引数についてメタ情報として、別の場所に定義されていたことも含めて記述しているので、直観的に分かりやすいが、一方で冗長で見難いとも言える。また、ＸＭＬの場合には、「事業名」「提出先」の記述場所と「申請者」「提出物」の記述場所が異なることに気付くであろう。これはデータ構造の設計方針に依存するものなので、最初から決まりがある訳ではな

いが、「△△市の○○事業の申請」という形で多くの申請が予想されるので、これを一つの型として考える時、その型自体の付随情報が「事業名」「提出先」であり、個別の申請に関する部分が「申請者」「提出物」である。前者の記述部分を「属性」と呼び、後者を「要素」と呼ぶ。また、述語に対応する「申請」を修飾する「許可」と言う情報も、型自体の情報として属性部分に記述されている。もちろん、この例とは異なる設計思想で記述場所を決めることもできる。なお、そのようなデータ構造自体の設計は、ＸＭＬ形式のデータとは別にＤＴＤ（Document Type Definition）と呼ばれる形式で記述され、別のファイルに保存される。

このようにＸＭＬや述語論理式を利用した表現のメリットを挙げると、直観性は減じられるが、図式の場合よりもより細かく記述でき、データとしての取扱いも容易になる。特にＸＭＬは、現在、世界中で標準的に利用され始めている表現形式なので、応用性や互換性にも優れている。データとしての取扱いが容易になる理由は、基本的にテキスト・データの形式なので、特別なソフトウェアが準備できない場合でも、ウィンドウズのメモ帳やワープロソフトなどによって、手軽に閲覧や編集ができるからであり、もちろん、電子メールの本文に直接記すこともできる。このような点で、技術的なデータ処理の観点だけでなく、日常的な取扱いの局面からも利便性が高いものとなる。

第六章　製作――法令・条例案の作成

第一節　ソフトウェア開発における製作

ソフトウェア開発において、製作段階のほとんどの作業はコーディングである。そして、ソフトウェアの本体がプログラムであることを考えれば、ソフトウェア開発過程の本質的な作業でもある。しかしながら、設計段階までがしっかりしていて、詳細仕様書ができあがっており、そこにアルゴリズムまで書かれている場合には、コーディング作業として行うことは、単にプログラミング言語に合わせてそのアルゴリズムを定型的に書き直すだけの作業である。その意味では、あまり創造性の高い作業ではない。この状況を示す具体例として、次のような詳細仕様書内に記述されたアルゴリズムの例を使って、これが各言語のプログラムに変換される例を示す。

詳細仕様書内の構造化チャートの記述例：

形式：hello（表示回数）

機能：与えられた整数値の分だけ、「こんにちは！」を表示する

処理：

（開始）

｜

｜――カウンタCに指定の表示回数を代入

```
―
―――[W]　C>0
―　　―
―　　―――「こんにちは！」を表示
―　　―
―　　―――Cを1つデクリメント
―
（終了）
```

ここで、「構造化チャート」とは、プログラムのアルゴリズムを「逐次」「分岐」「繰り返し」と呼ばれる単位だけによって記述する図式の一種である。様々な構造化チャートが提案され、用いられているが、それらは見やすさや取扱いやすさの問題を除けば、どれも大差はない。本チャート図も、よく見かけるチャート図を筆者が簡略化して提示したものである。この図の見方は、開始から終了に向かって、上から順に処理を進め、<W> の部分から下に伸びている部分が繰り返し処理を表している。<W> の横には、繰り返しを続けるための条件式が書かれる。この図式の便利な点は、テキスト・データとしてウィンドウズの「メモ帳」などでも記述できるので、図形描画機能や図形描画用のソフトウェアを用いなくても済む点である。例えば、アルゴリズムと言えばフローチャート図を思い浮かべる読者がおられるかも知れないが、フローチャート図をコンピュータ上で用いる場合には、ワープロ描画機能や専用ソフトウェアを必要とし、そもそも、データ形式もテキスト形式ではないので、互換性の問題を引きず

ることになる。さらに、フローチャート図は構造化設計を強いるものではないので、現在では、通常のプログラミングの方法論上、禁じ手となっているＧＯＴＯ文に対応する汚いアルゴリズムが記述できてしまう点でも問題がある。構造化チャートでは原理的にこのような記述ができないので、安心である。

ここで示した構造化チャートの例に対応するコーディングの例として、次のように二つのプログラミング言語における例を示す。

Pythonによる記述例：

```
def hello（n）：
    C=n
    while C>0：
        print 'こんにちは!'
        C=C-1
```

C言語による記述例：

```
void hello（int n）
{
    int C=n;
    while（C>0）{
```

```
        printf（"こんにちは！¥n"）；
        C=C-1;
    }
}
```

このように、プログラミング言語によって、多少の違いはあるが、詳細仕様書に構造化チャートが書かれていれば、かなり似た形のままでコーディングを進めることができる。こうして見ると、詳細仕様までを含めた設計作業が創造的な作業であったのに対し、コーディングは、むしろ、事務的な作業であると言えるのである。実際、特定のチャート図に限って、特定のプログラミング言語のプログラムを自動生成するようなソフトウェアも二〇年以上前から存在している。

こうしたコーディング作業の時に利用されるツールは、「テキスト・エディタ」と呼ばれる、ウィンドウズの「メモ帳」のようなソフトウェアである。特に、プログラミング言語の記述に特化されたオプション機能を備えているものが多い。例えば、特定のプログラミング言語で頻繁に使われる構文が自動的に挿入されるとか、プログラミング言語で予約されている特殊な単語については、自動的に強調表示やカラー表示がされるようなものである。その他、行番号を表示させる機能や、「キーボードマクロ」と言って、定型的なキーボード操作をコンピュータに覚えこませておき、自動的に編集操作をさせるような機能も大抵のエディタが備えている。また、構文の文法をチェックする機能を持っているエディタもある。例えば、括弧の対応関係を書き込むと同時にチェックして強調してくれる機能を持つエディタは多い。より詳細な構文チェックを行って、エラー箇所や疑わしい箇所を表示するよ

うな単体のソフトウェアも存在している。さらに、既に存在しているプログラムへの参照の記述を補助する機能を持つエディタもある。これは、既存のプログラム群を後から別のソフトウェアでも使いやすいようにまとめた「ライブラリ」と呼ばれるものを自動的に参照して、そのライブラリ内に登録されているプログラムの一覧がメニューのように表示され、そこから選ぶと、そのライブラリ内の該当プログラムへのリンク情報（関数呼び出し表記等）を自動的に自分の編集中のプログラム中に挿入してくれるものである。

このようなコーディング時の直接の記述作業以外にも、この製作の作業段階では、簡単な試験を行ったり、ドキュメントを整理したりする作業が付随している。試験に関しては、各プログラムの単位をコーディングするごとに、その部分が正常に動作するかを検査する、「単体テスト」を行わなければならない。その単体テストでエラー（バグ）が検出されれば、直ちに、そのエラー箇所を修正する必要がある。この修正作業は「デバッグ」と呼ばれる。デバッグを支援するソフトウェアを「デバッガ」と呼ぶ。詳しくは第三章第二節の七を参照頂きたい。なお、ここまではコーディング作業の一貫であるため、単体テストは、通常、この製作段階に含まれる作業であるが、第七章においてソフトウェア開発における試験について説明するので、その際に、まとめて述べる。一方、ドキュメントの整理については、例えば、実際に作成したプログラムの各単位（関数、サブルーチン、クラス等）の情報やそれらの参照関係などを自動的にチェックしてグラフィカルに表示したり、ドキュメントとして自動的にまとめたりする機能を持つツールがあり、これらを利用して、製作段階のドキュメントのまとめを行う。こうすることで、他のモジュールの開発者が参照することが可能となり、また、後に改良を行う必要があった際にも、容易にプログラムの解読や再利用が可能になるのである。

第二節　立法過程における製作

立法過程においては、政策設計段階で作成された要綱に基づいて、条文記述のための作法に則って、具体的に条文形式に記述する作業が、ソフトウェア開発の場合における製作、すなわち、コーディングに対応する作業である。大抵の場合、この作業と公布後の条文の更新作業をまとめて「法制執務」と呼んでいる。ここで行われている作業の本質は、形式的に表現することであり、当該政策の意味的な側面に対する製作作業は、前段階の政策設計の段階でほぼ終了している。そこで、この段階での課題は、要綱に記された政策設計の意図をいかに正確に、かつ、条文のスタイルに合わせて表現できるか、という点に集中する。もちろん、近年の社会的な流れに沿うためには、分かりやすさという視点も考慮すべきであろう。特に、筆者らが進めてきた日本法令の英訳支援プロジェクトでも、度々、翻訳し難い条文表現が問題視されており、今後も増え続ける外国人労働者の雇用や外国企業の進出の状況を考えると、やはり翻訳しやすさという点からも、分かりやすさの視点は重視すべきであろう。いずれにせよ、この段階の関心は表現の問題にある。ただし、文学のようにレトリックに凝る必要はないので、むしろ、客観的な形式的基準を作って、その基準に準拠させる仕組みにする方が、将来的には有益だと思われる。実際に、我が国の条文表現には、事実上の基準が存在しており、細部まで決まりごとがあり、さらに、そのマニュアルとなる書籍も存在している。しかしながら、分かりやすさよりも正確性に重点が置かれている傾向が見られ、一方で、もちろん恣意性を完全に排除できるような正確な形式化とまでは言えない。そこで、このような法制執務に関する現存する基準のうち、機械的にも扱えるくらい形式的なものについては、実際に、専用のソフトウェアなどを用意すれば、ある程度、条文の記述作業を補助することも可能である。本節の以降では、このような作業支援方法について、前節で

言及したソフトウェアの製作段階で用いたツールなどにヒントを得ながら提案する。なお、ここで行う提案は、人手による作業を支援するものと、人手による作業をなるべく廃して自動化を目指すものと大きく二種類に分けて行う。以下に、それぞれについて示す。もちろん、現実的には完全な自動化はまだまだ困難なので、両アプローチを併用することになると思われる。

一　条文作成の支援

条文作成作業もソフトウェアのコーディング作業と同様に、本質的な作業はテキスト、すなわち、文字列の記述である。条文の表記は自然言語で行われるため、人工言語であるプログラミング言語ほど構文規則や語彙などが厳密に形式化され、定義されている訳ではないが、一般の文書の記述に比べれば多くの形式的な決まりごとや慣習が存在し、書き方には一定の制約がある。すると、コーディングの場合と同じように、記述形式の検査（「シンタックス・チェック」と呼ぶ）や典型的な記述テンプレートによる入力補助などの支援方法が流用できるであろう。少なくとも、技術的には応用可能である。「技術的には」と前置きしたのは、法制執務の多くの形式的な決まりごとそのものが、現時点では、直ちにコンピュータで取り扱える形式のデータとして蓄えられていないので、その部分については、今後徐々に蓄積を進める必要があるからである。そして、このような形式的な決まりごと自体の形式的情報を蓄えておけば、ソフトウェア開発で用いられたツール群と同じ処理方式で条文作成用の支援ツールを提供できる。その意味で「技術的には」可能な状態になっていると言えるのである。

次に、ソフトウェア開発において頻繁に利用される典型的なプログラムは「ライブラリ」と言って、他のプログラムからも利用されやすいように、汎用に利用できる形で、あらかじめまとめて用意されており、それを再利用す

る形でプログラミングを進めることが多い。このライブラリの考え方やツールも条文作成の作業に流用可能である。例えば、条文を記述する時に、目的規定の書き方のテンプレートや罰則規定のパタンなどを蓄えておき、それらの一覧を見せることで、そこから選択させて、現在記述中の箇所に挿入できるようにするのである。このために、あらかじめ、定型的な部分を整理して蓄積すれば、制度や条文のライブラリとなる。

最後に、直接条文を記述する時ではなく、その後始末や整理のための作業の支援についても簡単に述べる。本稿で想定しているような条文作成の支援システムが整ってくれば、法令間の参照関係、条文間の参照関係、用語の参照関係、あるいは、指示語の参照関係についてもシステム側で認識しておくことが可能になる。そうすれば、依存する法令や条文の参照関係のチェックなども自動化が可能となり、さらに、そのような参照関係を自動的に分かりやすくグラフィカルに表示することも可能になる。また、法令案を一通り記述した後に、問題が発覚した時、参照関係にある条文を自動的に洗い出すことで、根本的な問題箇所を見つけ出すことも可能である。このように、エラーの発見や修正のためのツールとしても参照関係の情報は有効である。さらに、条文中で定義されている語の定義関係を網羅的にチェックして、本来定義すべき語句が未定義語のままであることなども発見可能である。これらの作業支援はソフトウェア開発で言う「デバッガ」に対応するツールの機能である。

二　条文作成の自動化

前章の提案のような形式的な仕様書ができているのであれば、条文を記述する製作段階の自動化が比較的容易になる。例えば、次のように、ＸＭＬ形式で要綱の一部が記述されていたとする。

<or>
<要件>法律</要件>
<要件>政令</要件>
<要件>規則</要件>
<要件>指令</要件>
</or>

もし、このように形式的に記述されていたならば、第二章第四節で述べたような、自然言語処理の比較的古典的な構文解析の技術を用いても、自動的に次のように変換することは容易である。

法律、政令、規則、又は指令

この場合、意味的に論理和であることを<or>◇◇◇・・・◇</or>で表現している。さらに、各◇部分は要件のタグによって<要件>○</要件>と表現しているので、このパタンに着目して、「○、○、○、・・・、又は○」と言う形式に変換するようにプログラミングしておけば良い。その場合は、プログラムの中に<or>などの記述に応じて、場合分けをしてプログラミングを進めることになる。しかしながら、このように一々構文パタンをプログラミングするのでは、大変な労力になってしまうし、そのプログラムにもエラーが含まれる確率が高くなってしまう。そこで、このように構文パタンごとにプログラミングをするのではなく、このような構文パタン（文法情報）

をあらかじめデータとして蓄えておき、それらを参照しながら変換を行うようにプログラミングすることが望ましいだろう。もちろん、構文を直接プログラミングするよりも高度なテクニックを要するが、それでも古典的な方法であり、十分可能である。

いずれにせよ、このような方法に発展させるためには、構文パタンを具体的な記述形式に合わせて蓄積する必要がある。この時に重要になるのは、入力データである要綱の書式である。どのような表現形式で条文として出力するかという点については、法制執務のマニュアルにも詳解されているが、それらの入力形式については定めはなく、法令案要綱として、ほとんど条文に近い形で記述されたものもあれば、メモ書きのようなものもある。さらに、多くの場合、担当者の頭の中にある意味的なものであり、明示的な形で示されていないことも多い。しかしながら、それらを明記し、さらに、自然言語だけに依らず、形式的な書式で記述しておけば、入力と出力の対応パタンが確定するので、自動化のためのシステム構築も可能となるのである。そこで、前章で提案したように、やはり形式的な要綱記述の方式が鍵になるのである。

さらに、このような自動化を支える核となるデータには「辞書」がある。これは、もちろん、電子的な辞書ではあるが、いわゆる手帳サイズの電子辞書や、書籍の辞書の代わりのソフトウェアではなく、コンピュータが可読な形式で意味的な情報を蓄えたものである。例えば、日英で対応する名詞のペアを一定の形式で並べたリストであるとか、語幹と活用形を対応させたリストをデータ化したものである。最も単純なものとしては、単語ごとの対訳形式のペアのデータや、単語とその品詞とを組にしたデータなどがある。

そのような辞書の中でも、特に動詞の持つ格についての情報を蓄えた辞書は、今回のような一定の表現を自動的に出力する場合には有効である。例えば、「走る」という動詞であれば、主体、場所、方向、手段などの格が考え

られるが、これを次のようにデータとして蓄えておく。

走る：主体（が），場所（を），方向（へ），手段（で）

これが格情報の辞書である。これは第五章第四節の二で示した述語のメタ情報と同様の構造で、さらに格助詞の情報が付加されているだけである。この辞書情報を前提にして、例えば、述語論理式を用いて、

走る（太郎，高速道路，大阪方面，自動車）

と記述されていた場合ならば、簡単なプログラムによって、自動的に、次のように変換することができる。

太郎が高速道路を大阪方面へ自動車で走る。

もちろん、述語論理式でなくても良いし、格情報の辞書の形式についてもより高度なものを含め、様々なものが考えられる。

このようにして、最終的に入力形式の定義や辞書的なデータが整えば、第三章第二節の四でも言及したように、形式的な要綱から条文に自動変換するようなシステムの実現も可能になってくる。ソフトウェア開発の過程では、人手による最後のコーディング作業で作成されたソースプログラムをコンピュータが直接解釈実行する電子的な信

号の羅列である機械語に変換することで、最終的なソフトウェアを製造している。その変換を行うソフトウェアが「コンパイラ」と呼ばれることから、形式的な要綱から条文へ自動変換するソフトウェアは「条文コンパイラ」と呼べるであろう。

第三節　非落水型開発の応用

非落水型のソフトウェア開発方法は、設計段階を飛ばして、いきなり製作段階から作業を始めるようなものである。既存のプログラムを再利用したり、修正したりして新たなプログラムを記述するからである。あるいは、とにかく簡単に作成できるプロトタイプ版のプログラムを書いて、それを元にして、テストと修正を繰り返す中で、徐々に完成度の高いソフトウェアに仕上げるという、テスト・ファーストの手法もあるが、これも非落水型の開発方法と言える。いずれにせよ、細かい設計などは行わずに、製作段階から取り掛かっている。もちろん、要求分析はある程度行っているであろうし、テスト仕様書を作成している場合もあるので、すべての作業の起点が製作にある、という訳ではないが、落水型の開発手順では、設計段階にかなりの労力や技術力が傾けられ、製作段階は事務的な作業に近かったことに比べて、非落水型の開発手法では、製作段階が中心的段階となると言える。本節では、このような非落水型の開発方法を立法過程に応用できる可能性のあるいくつかのケースを示す。

まず、自分達の自治体の類似条例や他の自治体の条例を参照して、要綱などは書かずに、いきなり、条文から書いてしまい、修正を重ねる中で徐々に完成させるという方法である。これは実際には多くの自治体で行われている手法である。表記上の便宜のために、この作業方法を仮に「模倣型」の作業と呼ぶことにする。なお、中央省庁で

も個人レベルの作業の時には類似の作業を行っているかも知れないが、多くは自治体での例規作成の局面で行われているため、ここでは、自治体での作業に焦点を当てて論ずることにする。このような模倣型の例規作成作業を大きく二つのフェーズに分けると、①類似法令の検索、②テンプレートの利用、という作業から成り立っていると見ることができる。そこで、これらについて順に述べる。

①のフェーズに関しては、極力多くの自治体の例規を集めたデータベースを作成しておき、自分達が参照したいような例規を見つけ出す支援機能が必要であろう。実は、筆者らの研究プロジェクトでは、既に、ウェブ上でアクセス可能なすべての自治体の例規を収集・蓄積しており、現在では単純な検索機能ではなく、参考にしたい例規を見つけ出すような検索機能の開発を進めている。例えば、自分達の自治体と同じくらいの規模の自治体に絞り込んで検索したり、例規の典型的な項目ごとに、そこに現れる言い回しをキーにして検索したり、足並みを揃えている近隣の自治体に絞って検索したりする機能である。このような機能も一部実現しており、我々



図10　法制執務支援のための条例検索システムの画面

に研究協力して頂いている自治体には、試験的にリリースしてフィードバックを得ることで、さらなる改良を進める予定である。この条例検索システムのハードコピーを図10に示す。なお、この支援方法では、情報科学的な技術を使っていることには違いないが、ソフトウェア開発の技術を流用している訳ではない。しかしながら、多くのプログラマも個人レベルでは似たような作業を行っていると思われる。実例としては弱いが、筆者もプログラミングする際は、過去の自分のプログラムを検索して、類似ケースの場合にどのようにコーディングしていたか、一覧し比較してみて、適切なものをコピーし、一部を書き換えて作業を進めることが多い。これが安心・確実で、しかも、素早く対応できるからである。いくつかの自治体職員の方々にヒアリングをした限りでは、例規作成時にも同じ思いからこのアプローチを採っていた。

②のフェーズに関しても、その支援システムの開発を現在進めている。このシステムについては第九章において紹介する立法支援システムの一部機能として実現されるものである。その支援方法は二つのアプローチが考えられる。一つは条例の準則のように、文字列としての条文の一部が、言わば変数化していて、例えば「○○市」のような形式によって、「○○」部分に当てはめることで条文が完成するようなものである。あるいは、仮想の自治体名や数値的な部分など、必要に応じて書き換えることで条文が作成できるようなものである。もう一つのアプローチは、本稿独自の提案でもあるが、知識構造に基づくものである。これは、条文のような文章としてテキストとして与えるのではなく、条例によって描かれている法政策の構造自体をデータ化して蓄えておき、その構造上の各部に該当する値を書き込む方法である。これならば、無理矢理語調を合わせることもなく、また、論理的整合性を保った応用も可能になる。現在、筆者らの研究プロジェクトでは、前者をサポートすることを先に考えているが、同時にプロトタイプのシステムには後者のアプローチによる支援機能も試験的に実現し、改良を進めている。

次に、非落水型と考えられる立法過程のケースの中で、テスト・ファーストに該当にするケースは、第三章第四節でも示したように、様々な自治体で実際に実施されているプロセスの中にも見られる。すなわち、法制執務の担当者などから構成される検討会議に、原課よりいきなり条文を記述した案が提出され、添削や修正依頼を受けて、修正・再提出し、これを何度も繰り返して、徐々にきちんとした条例案に収束させるようなやり方である。しかし、細かく見てみると、その原課の職員はどのようにして、いきなり条文を列挙できたのかと言えば、やはり、自分達の自治体の過去の類似法令や、他の自治体の類似法令を真似て作っている。つまり、先に示した、見本やテンプレートを用いるケースと同じである。そこで、テスト・ファーストに相当する場合も、技術的には先に示した方法論やツールと同じものが流用できるであろう。

最後に、プロトタイプから作成するケースに該当するものであるが、筆者らがヒアリングをした限りでは、実施している具体例は見当たらなかった。ただし、どのような作業になるか想定することはできるので、そこから考察を進めることにする。プロトタイプから作成するケースでは、プロトタイプさえ作成してしまえば、「模倣型」におけるテンプレートが与えられたことと同様であり、それを徐々に改良して完成に向かえば、結局同じ作業となる。そこで、プロトタイプを作る方法さえ検討しておけば良い。もし、作成しようとしている法令が全く新たな法令タイプであり、しかも、基本法のように抽象度が高く、法的にもテクニカルなものについては、そもそも、プロトタイプから作成することは推奨できない。このような場合には、非落水型ではなく、体系的に設計すべきであるし、少なくも、自治体レベルでそのような例規を作成する機会はほとんどないと思われる。それでも、稀には、全く新たな例規を作成しなくてはならない局面もあるだろう。その場合は、拙著論文[3]でも理論的側面と作業の骨組みを示したが、中心となる法律事項をまず先に定めて、そこに記載されている各部を段階的に詳細化し、その法律事項で

定められる条文の取扱いについて、隠れた条件を洗い出すことによって、当該法令の最低限の骨格を作ることができる。これをプロトタイプとすれば良い。この点では、ソフトウェア開発のツールや方法論が直接応用できる点はないかも知れない。しかしながら、人工知能技術、とりわけ、論理式の扱いや推論などのメカニズムを流用すれば、支援システムを提供できる可能性はある。特に、法律事項に含まれる概念の定義の部分などは、オントロジー工学の分野の知識表現の支援ツールが利用できる可能性が高い。

第七章　試験 ── 意見収集と議会審議

第一節　ソフトウェア開発における試験技法

一　ソフトウェア試験の目的

ソフトウェア開発における「試験」あるいは「テスト」とは、簡単に述べれば、作成されたソフトウェアが実際に意図通りに動くものであるかを検査することである。しかしながら、試験技法そのものを考案したり、議論したりする場合、あるいはテスター（試験を行う人）の心構えとしては、「テストとは、エラーをみつけるつもりでプログラムを実行する過程である」[(4)]と考えるべきであろう。つまり、試験の目標を「エラー検出」に置き換えるのである。これは、ソフトウェア試験の分野の古典的な名著であり、最近の情報処理環境に合わせて第二版も出版されたG・J・マイヤーズの『ソフトウェア・テストの技法』による定義である。この定義は、心理学的な分析も含む

議論に基づいて主張されたものであり、古典的であると同時に、現在でも支持されている。すなわち、意図通りに動いて欲しいという観点からテストケースを考えて試験を行っても、試験過程でのエラーの発見が難しくなり、結局、一旦ソフトウェアを提供した後に、ユーザからエラーの苦情が寄せられるという事態を招いてしまうからである。これは、結果として、意図通りに動いていないソフトウェアである、ということを広く露呈することになり、言わば「品質の低い」ソフトウェアを提供したことにもなってしまう。そこで、ソフトウェア開発の分野では、体系的・理論的に試験技法を考える時には、エラー検出を目的にすることを前提とした議論がなされている。本稿もそれに従う。

ソフトウェアの試験を行う際の最大の問題は、当該ソフトウェアの動作に関するすべての可能性を試してみることが困難である、という問題である。ほとんどの現実的なソフトウェアでも、数字以外のものを入力したり、何文字も入力したり、といくらでも、利用方法のパタンを考えることができる。これら可能なすべてのパタンをテストして、数字以外の入力はすべてキャンセルされるということを確かめなければ、すべてのパタンを試したことにはならない。すなわち、「0」から「9」までの数字を一文字ずつ入力して動くことを証明する、という問題設定ではなく、それ以外のすべてのパタン（アルファベットや各国の文字とその複数文字列）について動かないことも証明する、という問題設定となる。一種の「悪魔の証明」であり、これが原理的に不可能なことはすぐに分かるであろう。例えば、文字数にしても、一〇〇文字までなら正常に動作しても、一〇一文字ならばエラーとなること（例えばキャンセルされないで異常な動作をすることなど）もあり得るだろう。それが、本当にないことを実施して示すことが試験なのである。そこで、マイヤーズの表現を借りれば「テストの経済学[5]」を考えることも重要になる。つまり、すべての

テストケースを想定し、すべてを試みることはそもそも不可能なので、「エラー検出をより効率良く行う方法」を考えることこそが、ソフトウェアの試験技法の研究の本質的課題である、ということである。そこで、試験技法の良し悪しの判定はこの観点から行われている。

ソフトウェアの試験を実施する際の本質的な枠組みは、ある入力に対して、当該ソフトウェアの出力が要求通りのものであるかどうかを調べる、というものである。この時、入力とその要求通りの期待される出力を組み合わせたものを「テストケース」と呼ぶ。もちろん、通常は、一組だけということはなく、多くの組をセットにして、テストケースと呼んでいる。このテストケースさえ作成しておけば、実施自体はどのような方法論でも良い。期待通りの出力でないテストケースが見つかれば、それがエラー検出をしたことになる。そこで、このようなテストケースを考えることが、試験技法の中心的な課題となる。さらに、テストの経済学の発想からすれば、より少ない試験によって、より多くのエラーが発見できるような（集合としての）テストケースであれば、それは優れたテストケースを考案したことになる。

以上、ソフトウェアの試験技法の理論的側面の問題から、その目的を示してきた。厳密に試験対象のソフトウェアの仕様が定まっている場合であれば、テストケース上の問題に帰着して議論ができるので、たしかに、理論通り

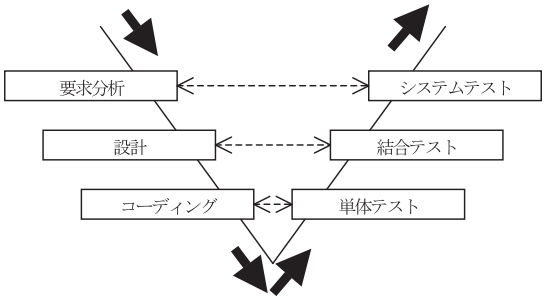


図11　Ｖ字型の開発工程図

の方法論を活用することができる。この場合は、仕様通りに動くか否かを確かめる試験を行うものであり、「検証（verification）」と呼ばれている。しかしながら、一応、開発者側で設計・製作して提供したソフトウェアが、ユーザ側としては、実際の現場では使い難い、あるいは、事実上は不都合な事態を招いている、ということもある。これは、仕様に書いてある必要条件以外の部分で、どれだけ実利用に供することができるかという、「妥当性評価（validation）」の問題である。この妥当性評価もソフトウェア試験のプロセスの中に取り入れることによって、より品質の高いソフトウェアを提供できることになる。これは、多くの場合、仕様自体が評価を受けている、と捉えることもできる。こうして、ソフトウェア開発においては、検証はもちろん、妥当性評価も試験工程に加えることが望ましいとされている。

二　ソウトウェア試験の種類

ソフトウェア開発の後半では、試験を行う作業が中心となる。試験の段階を位置付けるために、落水型の開発工程を採用している場合には、しばしば、「V字型」の開発工程と呼ばれる考え方が採用される。これは、まず、設計製作段階までについては、Vの字の左半分を下る方向と見て、次に、テスト段階以降をVの字の左半分を上がる工程と見る考え方で進めるものである。この様子を図11に示す。ここで、「単体テスト」とは、最小単位のプログラムを作成した後、その最小単位のプログラムが意図通りに動作するか検査するための試験である。モジュール単位で行う場合も単体テストと呼ぶことが多い。通常、単体テストはコーディングしたプログラマ自身が行う。そのため、この試験を製作工程の一部に含める場合もある。「結合テスト」とは単体のプログラムを当該ソフトウェアの構成に従って、結合した上でその動作を試験するものである。「システムテスト」とは、ソフトウェアとしてす

べてを結合した状態で、検証としての試験はもちろん、様々な妥当性評価も行う試験を指す。このV字型の開発工程図からも分かるように、コーディング作業と単体テストが対応し、設計段階と結合テストが対応し、さらに、要求分析段階とシステムテストが対応する。このように、ソフトウェア開発過の前半である製作までの過程は、全体的で抽象的な段階から具体的で詳細な段階までをトップダウンに進め、後半の試験の過程に移ると、具体的な段階からボトムアップに進めるという方法が一般的な開発方式である。

なお、単体テストや結合テストは、そのままでは、連動する他のプログラムが未開発の場合、試験ができない場合もあり得る。例えば、データベース・システムのソフトウェアを作成する時に、データベース内のデータ検索を行うプログラムを検証するには、その前に、データを入力するプログラムが動いていて、データを蓄積している必要があるし、データを検索する時に、具体的に一つ一つのデータにアクセスするための下位の処理も必要である。そこで、データを入力するプログラムやアクセスするプログラムがない状態で、当該検索プログラムを単体でコーディングした場合、試験ができないことになってしまう。このような時のために、「ドライバ」や「スタブ」というものが使われる。「ドライバ」は、前提となるような入力データがあたかも存在しているかのような環境に一旦設定した後で、当該テストケースに則った入力を試験対象のプログラムに与えて、試験対象プログラムを実行させるテスト用プログラムである。一方、「スタブ」は、当該プログラムが別のプログラムを呼び出して、処理させた結果を利用して自らの処理を進める場合、その別のプログラムがまだ存在しない時に、その代わりに利用する中身のないダミー・プログラムである。通常は、関数名や引数の数など外側の形式だけ整えておき、あらかじめ与えられる予定の入力に対してのみ、先に手計算で出力値を算出しておき、その値を直接に結果として返すようなプログラムである。これも基本的にテストケースごとに専用に作成されるものである。ただし、ドライバもスタブも通常

動作するプログラムではなく、テストケースのパタンでしか動作しないものなので、試験できる範囲には必然的に限界がある点に注意すべきである。

このような試験の段階による分類の他に、特に種類や分類と呼ぶものではないが、テスト作業の方法による区別を行うことも可能である。これは、要するにコンピュータを使う試験方法か、あるいは人手による方法かという観点による分類である。コンピュータを使う場合とは、先に述べたようなテストケースを用いて、実際にソフトウェアや一部のプログラムを動作させて検査することであり、ソフトウェアはコンピュータで動作するものなのであるから、当然の試験方法である。これに対し、人手による試験の有効性もしばしば指摘されている(6)。人手の試験とは、例えば、ソースコードを直接見てエラーを探すもので、チェックリストなど用いると効果的である。さらに、「ウォークスルー」と言って、あたかも試験担当者がコンピュータになったつもりでロールプレイのように、シミュレートしながら、ソースコード上の処理の流れを追跡する方法もある。人手による試験を個人で行う場合には「机上テスト」などと呼ばれることもある。これに対して、複数の関係者やテストの専門家から少数チームを構成し、「レビュー」形式で検査をする試験形態もある。なお、人手によって試験を行う場合でも、コンピュータを用いる場合と同様に、単体テストや結合テストでは、一部にスタブやドライバの役割の処理が必要となる。これは、もちろん、人手なので、コンピュータを動かす必要はなく、スタブの場合なら、手計算で、あらかじめ分っている結果を返せばよく、ドライバの代わりならば、そもそもテストケースを手入力するだけのことなので、特別な手続きは不要である。

以上のソフトウェア試験の分類方法に対して、技術的な内容の観点からも様々な種類が存在するが、大抵、これらは組み合わされて実施されている。もちろん、前述した分類による試験とも別のものではなく、重なっている。

例えば、システムテストにおいて、コンピュータを使用してストレステスト（後に説明する）を行う、という試験は極めて典型的なケースである。そこで、以降では、組み合せられる各要素となるような技術的観点からの典型的な試験技法について説明する。

技術的な観点から見ても、試験方法にはさらにいくつかの観点が存在する。まず、典型的なものには、試験を行う側がプログラムの内容を参照して行う「ホワイトボックス・テスト」と呼ばれるものと、逆にプログラムの内容を見ずに仕様だけから考えて試験を行う「ブラックボックス・テスト」と呼ばれるものがある。単体テストのようにプログラミングした本人が行う場合が多い試験の場合には、ホワイトボックス・テストとなるが、多くのシステムテストのように、ユーザ操作の側に立って行う試験の場合にはブラックボックス・テストとなることが多い。もちろん、より経済的なテストケースを案出するためには、両方を組み合わせることもしばしば行われる。

次に、テストケースを作る際に何に着目するか、という観点から典型的なものを挙げておく。それは、条件分岐に着目する方法と同値分類を行う方法である。前者は、典型的なホワイトボックス的な試験方法であり、まず、プログラムの記述内容を見て分析することから作業を始める。それは、プログラムの中の条件分岐処理に着目して、すべての場合分けを明らかにする作業である。その上で、すべての条件分岐のパタンを網羅できるようなテストケースを作ることが、最も典型的な条件分岐に着目したテストケース作成の技法である。これに対して、後者は、ブラックボックス的な試験方法の典型であり、当該プログラムの仕様を参照して、入力データと出力データを分析し、同じカテゴリに入るデータをまとめることから作業を始める。そうして、実際に試験を行う時は、各カテゴリについて一つだけ試験を行えば、効率的で網羅的なエラー検出が可能になる。これらはさらに高度な試験技術の前提となっており、また、組み合わせて使うことも可能であるが、本稿の関心からすると、技術的に立ち入り過ぎる

ことになるので省略する。

最後に、マイヤーズによれば、「上級テスト（Higher-Order Testing）」と呼ばれる範疇に属する一五種類の試験が挙げられている[7]。これらは、現代風の用語法に従えば「上流工程のテスト」と呼ぶべきかも知れない。本稿では、論理学における翻訳に揃えて「高階テスト」と呼ぶことにする。いずれにせよ、検証というよりも、主に妥当性評価を目指した試験方法であり、システムテストの段階で行われるケースが多い試験方法である。一五種類の試験は、大容量テスト、ストレス・テスト、ユーザビリティ・テスト、秘密保護テスト、手続きテスト、性能テスト、信頼性テスト、回復テスト、機能部分テスト、記憶域テスト、構成テスト、互換性テスト、設置テスト、サービス性テスト、文書テストである。ここでは、本稿で後に参照するもののみ簡単な説明を行う。それらを示すと次の通りである。

・大容量テスト

ソフトウェアへの入力量を非常に大きくして動作を試みる。

※ただし、単に、多くのテストケースを行う、ということではない。一個のテストケースの場合でも、大容量テストとなることがある。例えば、物理的な入力量を増やしたケースを試験すれば良いので、テストドライバで何千万回もループさせてコマンドを実施する、大容量の記録媒体を読み取る、一行の入力欄に書籍一冊分のデータを書き込む、等のケースであれば、これら自体は一個のテストケースであっても、大容量のテストを実現できる。

・ストレス・テスト
短い時間にソフトウェアへの入力量を大きくして動作を試みる。
・ユーザビリティ・テスト
利用者の使い勝手の観点から試験を行う。
・秘密保護テスト
機密性のある情報や個人情報がなどが保護されているか検査する。
※通常、ソフトウェアは各機能が正常に働くことやその動作の効率性、あるいは、使い勝手などの観点から評価されることが多い。しかしながら、機密性という観点からは、それらに反して、冗長で非効率的なアルゴリズムであったり、使い勝手が悪くなったりすることもある。それでも、このことによって機密性が頑健に守られているなら、トレードオフとなるので、そのソフトウェアの適用対象によって総合的に評価されるべきである。
・手続きテスト
人手による作業の部分も含めて、ユーザ側の一連の作業手続きの中で試験を行う。

第二節　作成された法令の検査

一　立法過程における試験

本節では、ソフトウェア開発における試験の考え方や技法を応用して、立法過程において作成された法令を検査

する方法および支援方法を検討する。まず、立法過程における試験とは何を指すのであろうか。この回答には、次のような三つの立場があるであろう。

(A) 法政策が期待された通りの効果を示したことの試験
(B) 法令通りに法政策が実施されていることの試験
(C) 要望や趣旨を反映した法令が作成されたことの試験

ここで、前提として、本稿の目的は、政策や制度自体の良し悪しを考えたり、優れた政策自体を案出する方法を目指したりすることではない。本稿の目的は、関係者が良いと考える政策を法令の形式に移し込む作業の方法論や支援を目指したものである。そこで、ソフトウェア開発における試験に対応させて、立法過程の試験の目的を考えてみると、本稿の趣旨から言えば、法令が公布される前、できれば法案を議会に提出する前に、より多くのエラーを発見して修正することであろう。そこで、本稿では、(A)や(B)ではなく、(C)のような試験について扱うこととしておきたい。なお、議決される前の段階までの様々な検査も試験の一種と考えることができるので、議会自体も試験の一種と考えることができるし、当然、議会にかかる前に実施されるパブリック・コメントも試験の一種であると考えることができる。

ソフトウェア開発における試験の考え方を踏襲すると、まず、一般に「すべてのテストケースを試すことはできない」という、ソフトウェア開発の分野で成り立つ大前提となる命題が、立法過程でも成り立つことを確認しておきたい。ソフトウェアの場合には、入力データが組み合わせ的爆発を引き起こすという問題があり、この命題が成

り立っていた。立法過程でも、法令を適用するケースは同様に組み合わせ的爆発を引き起こすので、この命題は同様に成り立つ。しかも、その組合せパタンの元となる要素を洗い出すだけでも限りがないという事情もあり、ソフトウェア以上にすべてを試すことが厳しいものとなるであろう。従って、ソフトウェア開発の場合と同様に、試験に対しては一種の経済学的観点が必要になってくる。すなわち、限られたテストケースや試験時間の中で、最大にエラーを検出できることが試験技法の優劣を決めることになる、ということである。

立法過程における試験の場合は、ソフトウェア開発における試験のように、前提となるV字型の工程イメージや単体テスト・結合テストなどに相当するものがあるのであろうか。この点についても、次のように対応付けることができる。要求分析、設計、製作の段階は、本稿でこれまで示したように、立法過程についても、同様に進めることができると考えて良く、トップダウンに作成したら、各条文単位から順に試験を試みて（単体テストに対応）、徐々にその項目のカテゴリごとにまとめて試験を行い（結合テストに対応）、最後に当該法令の単位で試験をすることで（システムテストに対応）、V字型の工程に対応付けることが可能である。実際に、現場でも、普通はそのようにチェックをかけているのではないだろうか。すなわち、単体テストとしては、法令案要綱に書かれた意図が対応する条文に表現されているか検査を行い、結合テストとしては、当該箇所が法政策要綱などの法令案要綱よりも抽象的な要綱の意図に従っているか検査を行い、さらに、システムテストとしては、そもそもステークホルダや首長の要求、住民の要望、立法事実から導かれる目的、合意のある制度趣旨などに照らして、当該法令を見た時に齟齬やエラーがないか検査を行うのである。

二　立法過程におけるテストケース

こうした試験作業に現れるテストケースについても、ソフトウェアのテストケースとの対応を示しておく必要があるだろう。そこで、まず、考え方の準備として、ソフトウェアの試験の最小単位である、関数やサブルーチンのような手続きの単位に対応するものを立法過程の対象で考えてみると、それは条文であろう。関数やサブルーチンが、入力に対する出力を行うものであると考えて、その正しい入力と出力の組合せがテストケースとなるので、条文の場合もこのような入力と出力に対応するものがあれば、テストケースを同様の枠組みで捉えることができる。実は、法令に本質的な条文とは、法律事項を表した条文であり、それは、当然、ルールの形態をとる。ルールは、条件（法律要件）部と帰結（法律効果）部で構成されており、ある状況をその条件に当てはめた時に、対応した帰結を導き出す様子を表している。その条件に当てはめる状況こそが入力であり、その結果導き出される帰結が出力と捉えることが自然であろう。このように考えることにすれば、単に、ソフトウェア開発と同型の話になる、というだけでなく、ルールの枠組みで捉えることで、形式論理学的な取扱いとの親和性も高くなるのである。そうすれば、将来的には条文の論理性を自動的に検証させる途も開けることになる。こうして、テストケースは、法律事項の条文の場合、条件部に当てはめたい状況や事実と、その場合に帰結させたい結論の組として作成すれば良い。この具体的な組合せをテストケースとして多数用意して試験を進めるのである。

しかしながら、ルールの形として捉えるという方法には問題もある。それは、例えば、目的や趣旨のように、容易にはルールの形として見ることができない条文が存在することである。本来、議会で決定されてしまえば、原理的にはどのような文面であっても、法令として存在することができるので、ルールの形で捉えられないものがあることは必定である。もちろん、圧倒的に多くの条文はルール形式で捉えることが可能であり、それに対して、この

ような非ルール的な条文は比較的少数の例外か、あるいは、パタン化されたものが多いので、個別に対応することになるだろう。例えば、目的規定については、意味的な構成にはパタンがあり、当該法令が採用している手段（aとする）、その結果直接的に実現しようとする目的（bとする）、さらに、その先にある目的（cとする）が記述されているケースが多いので、論理式で表現すると、a⇓b、b⇓cが記載されていることになり、仮にaが省略されても、それは当該法令が実施されること（'aとする）により、直接の目的が成り立つと解釈できるので、a⇓bというルールを考えれば良く、このように、ルールの形として見ることで法令の大抵の部分をカバーすることができる。なお、定義規定については、定義内容に書かれた要件が満たされた場合、その状況を当該定義の用語によって呼称する、というルールと見なせば良い。

　このような入出力によるテストケースの作り方についても、ソフトウェア開発の試験技法から応用できる。テストケースを考える時、立法過程における試験の場合は、通常、具体的な条文が曝された上での試験になるので、まず、ホワイトボックス・テストの手法を用いることができることは確かである。この場合、各条文の条件部の様々な入力組合せや複数の条文が関連する場合の組合せなどのケースを考えてテストケースを作ることに相当する。一方、あえて条文内容や法令案要綱などを見ないで、テストケースを作成するような、ブラックボックス・テストの手法を用いることも可能である。例えば、元々の立法事実や要求に答えたものになっているか、あるいは、経緯を知らない様々な住民に対しても妥当なものになっているか、等の観点からのみテストケースを作成するのである。さらに具体的には、適用対象の人に関するものであれば、外国人の場合はどうか、未成年の場合はどうか、所得の低い人の場合はどうか、家族の多い世帯ではどうか、等、様々なケースを想定して、そのケースごとに、本来の当該制度の趣旨からすれば、未成年は除外されるべき、外国人にも適用されるべき、等として、先に、各ケースと、

そのケースで期待される帰結をセットにしたものを何種類も用意しておく。これをルールへの入力と出力と考えて、後は試験を行えば良い。さらに、このようなケースを案出する時に、満十歳はどうか、満十一歳はどうか、というケースよりは、小学生や中学生という括りにする方が、さらにまとめて、未成年という括り方にする方が、より効率の良い試験を行うことができるだろう。このように、ケースをグルーピングして捉えることが前節で紹介した同値分類を考えることに相当する。こうしたブラックボックス・テストの手法でなく、ホワイトボックス・テストの手法で作成したテストケースばかりを想定している場合、例えば、当該条例の作成に関わった自治体職員が効率良く検査を行おうとしてしまう場合は、そもそも、内容を知っているので、その情報に照らして条例を見直してしまいがちであろう。しかしながら、それでは思いもよらない例外的なパタンはなかなか発見できない。まさに、経験と勘による職人技になってしまう。そこで、ここで提案したような体系的な技法を導入すべきなのである。もちろん、経験と勘も重要であり、体系的な方法と合わせて利用することに問題はない。ただし、その場合でも、それらの知識の伝承のためには、客観化して、テストケースの入力パタンを蓄積したり、チェックリスト化したりしておく必要があるだろう。

なお、試験作業は、コンピュータ上で実行するものでもなければ、本当に実社会で実験する訳にもいかないので、必然的に人手による方式になる。すなわち、レビュー方式やウォークスルー方式になる。ウォークスルー方式の場合は、テストケースに従って、関連条文に当てはめていき、その結果をチェックすることで行われる。シミュレーションのようなものである。これをテストケース分、繰り返せば良い。レビューの場合には、単体テストについては法令案要綱と見比べながら、異変などを検査することになる。結合テストの場合やシステムテストの場合は、もし、レビュー方式で進めるのであれば、従来型の会議等で一斉にチェックする場合と同じことになるであろう。そ

れでも客観性の高い方法論に基づく方がテストによる品質は高くなると考えている。

三　高階テスト技法

先に示したように、マイヤーズはソフトウェア開発における一五個の高階テスト技法というものを提示していた。本小節では、これらの中で、立法過程においても応用できそうな考え方について検討する。なお、高階テスト技法は、通常の試験とは評価の観点を変えたり、総合的に見て、現実世界においてどのように振舞うかを検査したりすることが中心である。そこで、立法過程における試験作業に移して考える場合も、条文の書き方のような表現の問題よりも、法令上で意図されている制度の内容に踏み込む部分もある。ソフトウェアの世界で喩えるならば、ある業務をＩＴシステム化した際に、実際の利用の局面を想定して試験を行うことによって、その新たな業務フロー自体の問題点を発見してしまう場合がある。それと同様に、条文の検査を通じて、場合によっては、手続きの設計の不備など、当該制度の問題点が発覚することもあるかも知れない。これは制度設計の問題であって、本稿の対象である、設計した制度を正確に法令として表現するための支援、という観点からすると、カバーする範囲を超えている部分も多い。そこで、ここでは簡単な提案例を示すだけに留める。

・大容量テスト

適用対象自体やその関連実体、あるいは行為などの量を極端に増やしたパタンを考えてテストケースを考える。

例えば、申請者が殺到するとか、金額が大き過ぎて対応できないようなケースを指す。

・ストレステスト
短時間に大容量テストのようなケースを扱う場合を想定したテストケースを用いる試験である。
・ユーザビリティ・テスト
国民や住民、あるいは関係機関が利用しやすいかという点からテストケースを考えた試験である。例えば、広く適用される可能性がある法令の場合、国民や住民、あるいは外国人にとって、読みやすく、理解しやすいか、手続きに混乱しそうな点はないか、等を検査するテストケースを用いる。
・秘密保護テスト
個人情報の漏洩や自治体や国家の機密事項が漏れることがないか等、情報の管理やセキュリティに関する問題点発見や脆弱度合いを検査する試験である。
・手続きテスト
法令の場合には、基本的に実施する主体も人手によるものであるから、ユーザビリティ・テストと重なる部分が多いが、主に、法令に規定されていない部分の運用に関するものである。例えば、国民や住民の行う手続きへの対応方法が整っていなかったり、現場の担当職員の裁量に任される部分で、判断に窮するような箇所が含まれていたりする点を検査する試験である。

なお、このようなソフトウェア開発における試験技法の応用でなくても、そもそも、制度設計独自の高階テストと呼べるものが存在するものと思われる。例えば、他の法令との参照関係の試験や法律的な様々な関係の試験などは典型的であり、実際にチェックされているものである。そこで、ここで提案したものは、むしろ、通常はあまり

関心を払わない可能性のあるものであり、結果として検査時に見落としがちな項目を網羅的に発想することに貢献できる。

四　立法過程における試験支援

本節で述べたような試験技法を採用する場合に、そこで要求される立法支援システムが持つべき試験支援機能の例について記す。

［パブリック・コメントなどの意見収集支援］

・意見収集作業の支援

既に導入している自治体もあるが、ウェブや電子メールによって意見を求めるシステムの導入が可能である。ただし、その各意見のデータについて、正統性を担保できるようなセキュリティ保護や認証方法などの技術的な課題は残る。

・収集された意見の整理作業の支援

大量の意見が集められた場合、自治体関係者や議員がそれらをすべて読んだり、分析したりすることは困難であろう。この時、もし、ウェブや電子メールなどテキスト・データとして取り扱うことが可能な形式で収集されていたとすると、自然言語処理技術が役立つと思われる。最近では、大量の文面を一気に機械的に処理して、その関心や要望の傾向を掴むという技術がある。もちろん、一〇〇％の精度がある訳ではないが、大きな流れを見落とす可能性は低くなる。このようなソフトウェアは非常に高価なものがほとんどであるが、実際に、

マーケティングのためのアンケート集計などに導入されている。なお、試験という観点から見ると、承認や支持の意見でなく、本来は、問題点を指摘している意見を見逃さないことが重要である。すると、全体の主流から外れていると思われる少数意見を見つけることも重要になるが、このような全体の主流となる意見を検出するシステムがあれば、逆に主流でないという評価を得た意見だけを出力することによって、少数意見を取り出す可能性も見えてくる。

・提示資料の作成支援

パブリック・コメントに先立って、広く一般に対し資料提示を行う必要があるが、その際には、ステークホルダや議会への説明資料の作成支援と同じ手法を用いることができる。本稿で提案しているような法政策の構造データを扱っていれば、原理的に、説明に利用する政策の詳細な図なども自動的に作成できる。もちろん、ポンチ絵のようにうまく要点を捉えて、高度なレトリックの技能を要する支援は現在の技術レベルでは困難である。しかしながら、単に手間のかかるだけの煩雑で細かな資料に関しては、正確な作成を支援できる可能性がある。

［テストケース作成支援］

・条件分岐

ルールの要件や適用除外の条件の関係を明確に図示することでテストケースの作

表1　テストケースの例

	入力			出力
命題	a	b	c	d
テストケース1	真	真	真	偽
テストケース2	真	偽	真	偽
テストケース3	真	真	偽	偽
テストケース4	真	偽	偽	真

成を支援することが考えられる。本稿の提案の前提は、設計時点から、電子的に扱える形で政策構造を構築するものであるから、その構造の中に自動的にこれらの条件関係の構造も含まれている。さらに、条件の組合せのパタンを自動的に出力させることも可能なので、条件分岐を網羅するようなテストケースの自動作成も可能になる。条件を網羅するテストケースを作成する様子の簡単な例を示すために次のような状況を考える。

A: 申請書提出（a）⇒ 事業許可（d）
B: Aの例外：適格でない場合（b）
C: Aの例外：申請内容が許可基準を満たさない場合（c）

これは、Aのようなルールを試験するものであり、他の条文においてBやCという隠れた条件が提示されていた場合である。なお、括弧内のアルファベットは各原子命題のラベルとする。すると、この試験の入力は、a、b、cの各命題の真偽であり、出力はdの真偽と考えることができる。そこで、この場合、テストケースは、a、b、cの態様に応じて、dの真偽、すなわち、「事業許可」となるか否かが決まる。そのテストケースのパタンを表1に示す。これを要綱の内容から作成し、実際に、各テストケースについて、ウォークスルーで追跡する際に、条件部分に当該テストケースの入力を当てはめるとどのように扱われるか、条文を読み進んで行けば良い。その結果、導かれる態様が当該テストケースの出力と一致していれば、そのテストケースはクリアされたことになる。これをすべてのテストケースで行うことが条件分岐の網羅的な試験である。そこで、このようなテストケースを自動作成するツールがあれば役立つであろう。実は、命題論理の形やそれと互換な構造

データさえあれば、比較的簡単に実現できる。なお、この例のような簡単なパタンでは、わざわざコンピュータの支援を必要としないかも知れないが、当該法律事項の要件や対応する例外記述の要件が増えて複雑化すると、一気にこれらの組合せパタンが増え、人手では、場合分けに漏れが生じてしまうケースもあり得る。そのような点からも機械化することに意義があると考えている。

・同値分類

テストケースの入力となる様々なものを考える場合に、闇雲に具体例を当てはめて考えるのではなく、まず、ある程度抽象化して、すなわち、ある共通性質を持った同類をまとめ上げて、そのクラスを考え、その中から、少数の具体例を代表値として選んでテストケースを作成することが同値分類を利用したテストケースの作成方法である。この場合、ＩＴ化による機械的な支援によって、共有性質の抜き出しや具体例の列挙が促進されるようにすることは、少々困難である。筆者も法的類推の研究[8]の中で、概念の抽象化を行う機械的な仕組みを提案しているが、構造化された、質の良いデータが十分に存在することが条件になっており、理論上は可能であるが、現状では、実際の試験の局面に生かすことは難しい。しかしながら、抽象化や具体例の列挙自体は人手で行うにしても、その分類の後、テストケースの作成までの機械的な作業ならば支援できる。これも条件分岐の場合と同様に、各同値クラスから代表を選抜して、それらを組み合わせる必要があるが、この時に、自動的に整理して列挙するシステムがあれば、漏れを防ぐことができる。また、同値類から選ぶ時には、「限界値」と言って、そのクラスの中で、最も極端な値、境界に近い値を選ぶことが、ソフトウェアの試験ではポピュラーである。これは、立法過程における試験でも同じである。例えば、小学生と言う同値分類のクラスを考えた時、小学一年生に対しては、より幼児に近い配慮が必要になるかも知れないし、小学六年生ならば中学生並

みの扱いが必要となる場合もある。これを小学生という括りで抽象的に考えていると、人によって、イメージするものが異なってしまう。従って、これらをテストケース作成時に考慮することによってエラーを検出できる可能性が高くなると考えられる。そこで、例えば、テスト入力用データのカテゴリを定義した後、具体値を考えて入力し、一旦蓄えておく。その際、限界値となるような要素の入力促進を図る機能があれば便利である。さらに、分類クラス名を入力する時に、数値的な範囲であれば、その範囲も合わせて入力させ、自動的に限界値や、典型的な代表値を算出させることも可能である。先の小学生の例であれば、「小学一年生〜小学六年生」と入力させた時、その限界値は小学一年生および小学六年生であることは機械的に判別できることであるし、代表値は例えば平均や中央値をとって、小学三年生や小学四年生を算出させれば良いので、人手を使わなくても実現できる。

・典型的なテストケースのデータベース化

立法の対象は、法令であり、それによって社会制度の実現を目指すのであるから、必ず人手が関わる。テストケースに関しても、同値分類の時のように、結局人手で行う局面が含まれてしまう。そこで、これらのテストケースや同値分類のクラスのパタンを一旦作成したら、同じ労力を省略するために、他の法令の試験の際にも再利用できるようにデータベース化を行うことが、本提案である。同値分類については、このようなデータベースがあれば、シソーラスのような一種の類語辞典の働きをして、その人手による作業の際に、発想を生み出す手掛りを与えることも可能となる。

［ウォークスルー支援（テスト実施支援）］

・設定値と状態変化の記録

人手でウォークスルー試験を行っていると、テストケースでの入力側の値がいくつもある場合、ウォークスルーの過程で、中間状態に移行した時（許可申請後で許可が下りるまでの間等）、これらを記憶しながら、進めるのは大変な場合がある。そこで、テストケースごとにウォークスルーを行っている間、作業のプラットホームとなるシステムを用意しておき、ウォークスルーの進行に合わせて、現在値を書き込んでおく。どこまで作業を行ったかという情報が記録されることになるので、複雑なウォークスルーの場合、作業を中断しても、後にその中断箇所からスタートする際に、正確にその時点から再現できる。

・条文の参照関係表示

ウォークスルーのためのプラットホームには、当該テストケースで関連する条文の参照関係も表示させておく必要がある。ウォークスルーは条文を読みながら追跡して行く作業であるから、現在、全関連条文中のどの条文をチェック中なのか、位置付けが分るようにするためである。もちろん、このようなことを個人で実行する時には、意識しているであろうが、チームで行う場合には引継ぎも容易になる。さらに、例外規定に一致しないか、上位法で規定されていることに該当していないか、等のチェックが抜けてしまうことを防ぐことにも貢献できる。

［立法事実やステークホルダのデータベース］

試験過程でのみ利用される訳ではないが、ステークホルダの要求を蓄積したり、立法事実を蓄積したりするため

のデータベースが必要である。これは要求分析過程で必要であるだけでなく、試験段階で妥当性評価などを行う際に、客観的な拠り所となるものである。つまり、テストケースを考える際の重要な基準であり目標となる。なお、このようなデータベースとともに、何の要望に関する具体的な証拠・証言なのか、その要望との関係も記録しておくと便利である。さらに、樹形図のような形で、その論理的な依存関係を図示する機能なども必要であろう。立法分野ではないが、筆者は、既に、訴訟実習の学習システムの中で、ArgPLAN[9]と言う、証拠・証言と法律要件を結び付けるシステムを開発し、授業の中で利用している。これと同じメカニズムを留用して、法律要件ではなく、政策への要望を入力させるように変更するだけで直ちに実現可能である。

この他にも、字句の誤りや疑問箇所を機械的に検査する機能なども考えられるが、これらの機能は既に市販のシステムでもある程度実現されているので、ここでは省略した。

五　試験としての議会

本節の最後に、試験という観点から見た議会について述べる。議会の本質的な部分は人手で行われることに意義があるので、議会運営の事務手続き的なことや公開に関すること以外には、ＩＴ化になじまないと思われる。そこで、本稿では、積極的に試験技法自体を取り入れたＩＴ化のような提案は行わない。しかしながら、ソフトウェア開発における試験と同じように、本節で提案したような試験に対する考え方を議会にも適用してみたい。この観点から議会というものを見た場合には、話し合いの結果、多数決により支持者が多いという事実を導く側面よりも、法案の中で見逃している問題点を検査する側面に比重を置いて考える必要がある。なぜなら、試験の目的は、うま

く機能することを確認することではなくて、できる限り多くのエラーを発見することにある、という観点で考えているからである。そこで、十分な討議期間を経ていない法案の議決は、試験という観点から見ると、品質の低いものであり、しばしばそのような形で成立した法案が見られるが、仮に実質的に問題が生じていないとしても、高い「リスク」を内在したものとなってしまう。ここでの「リスク」とは危険という意味合いよりも、経済学における定義のように、不確実さの範囲が広い、という意味である。もちろん、単に時間さえかければ良いという訳ではないが、強力な反対派でなければ思い付かないようなテストケースには敏感になるべきであり、支持者が考えるテストケースよりも、効率良く良質なテストケースとなる場合がある。ソフトウェア開発の業界でも、「ゴッドハンド」と言う俗称で呼ばれる、試験の最終段階で致命的なエラーを発見してしまうテスター（試験担当者）が登場することがある。現場の開発者からは嫌われるが、それでも同時に、ゴッドハンドの試験を通過したソフトウェアの品質は高いとも考えられている。議会での反対派は、試験としての議会という観点からは、まさにゴッドハンドとなり得る立場にある。このような観点から、積極的に品質を高めるための場として、議会を位置づけることも可能である。

第八章　提供と保守――公布・施行・改廃

第一節　公開とバージョン管理

ソフトウェア開発におけるリリースや保守の段階は、既に完成した製品の扱いに関する局面なので、開発段階のように際立った方法論がある訳ではない。それでも、実際に、多くの開発者が頻繁に関わる業務がこのソフトウェアを提供した後の作業である。なぜなら、現実的な利用に供されているようなほとんどのソフトウェアは、提供後の不具合の修正のため、あるいは、機能・性能を向上させたものに置き換えるためにバージョンアップが行われるからである。本章では、この過程の中においても立法過程の作業に応用できるような技術の可能性を探ってみる。

ソフトウェアを提供する場合は、第三章第二節の六で示したように、パッケージ製品として販売するならば、販売ルートに乗せることが最もポピュラーであり、特定の発注者から受注してソフトウェアを作成した場合は、発注者に納入したり、発注者のシステムにインストールしたりすることで当該ソフトウェアを提供したことになる。近年は、フリー・ソフトウェアを中心に、ウェブ上で提供されるケースも多い。このようにウェブ上で提供される場合は「公開」されると言う。

こうして、一旦提供された後、何らかの修正や改良を行い、ユーザ環境に合わせて調整したり、再提供したりする作業が「保守」作業である。この保守作業の中で重要な役割を持つものは、当該ソフトウェアのドキュメント類とこれまでの改変ごとのプログラムである。改変は何段階も行われるので、その度ごとに、そのソフトウェアを構成する全プログラムが一セットとなってコピーされ保存される。この一つのセットが「バージョン（version）」で

ある。このバージョンが何代にも渡って、時系列に並べられて行くので、その改変履歴と該当バージョンのプログラム・セットが対応付けられて整理・蓄積されて管理されている必要がある。これがバージョン管理である。さらに、このような時系列に置き換わるような形のバージョンを「リビジョン（revision）」と呼ぶ。ただし、このようなリビジョンとは異なり、時系列に置き換えてはいけない種類のバージョンもある。例えば、全く同じ機能のソフトウェアで、内部のプログラムもほとんど同じものであっても、周辺機器やパソコン本体のハードウェア環境、利用しているＯＳの環境、あるいは、ユーザの使用言語、等に応じて、一部を置き換えた形で提供しているプログラムのセットが、対応する対象ごとに出来上がってしまうことがしばしば起る。このような時のバージョンは「バリエーション（variation）」と呼ぶ[10]。

ソフトウェア開発における提供と保守の作業のうち、ウェブを利用した公開については、法令データの公開の場合でも、直ちに同様の公開方法は可能であり、現在では、総務省の法令データ提供システムのホームページや多くの自治体のホームページで実際に公開されている。すわなち、既存の手法として定着している。一方、バージョン管理に関しては、法令の各時点での状態を蓄えたり、整理して管理したりすることに相当する技術であるが、これは現時点では定着しているとは言い難い。実際、多くの自治体では、紙面やワープロデータのような形態、あるいは、その管理の部分を民間業者に任せていることが多いようである。一部には次節に示すようなシステムを導入しているケースもあるが、いずれにせよ、法令の更新に応じたバージョンを管理することは、何らかの形で必要な技術である。そこで、本節の以降では、バージョン管理について示す。

まず、法令もテキストであり、ソフトウェア開発で中心となるプログラムもテキストである。そのようなテキストで書かれたファイルのバージョン管理については、ソフトウェア開発の分野では、差分をとって、日時やバー

ジョン番号ともに残しておく方式を採っている。このようなシステムは、比較的単純なアルゴリズムで実現できるため、かなり古くから存在している。例えば、ＲＣＳと呼ばれる古典的なソフトウェアがある。これは、あるファイルを最初にシステム内に保存しておき、その後、そのファイルを更新した新たなファイルをそのシステムに書き込むと、自動的に差分情報を控えておくものである。そして、必要なバージョンのファイルが欲しい時には、元のバージョンから、差分情報を順に適用することで、指定されたバージョンのファイルを再生成して提供する。そこで、法令の末尾に記述される附則に記載されているような各法令の履歴情報は、まさに、この技術で十分対応できる。このバージョン管理の技術の核心部分は、第二章第五節で紹介した二つのファイルの差分を抽出する処理の技術である。なお、単独の自治体で管理する場合にはリビジョン管理だけを考えておけば良いが、将来、複数の自治体のデータベースを統合するようなことが起これば、バリエーション管理も重要になる可能性がある。

次に、法制執務の支援のためのバージョン管理システムが備えるべき機能を考えると、このような、単なるバージョン管理だけでなく、差分データから溶け込まし文を自動的に出力したり、差分を分かりやすく表示する新旧対照表を自動的に作成したりする機能が必要になるだろう。これらも、技術的には十分可能なものであり、本稿で提案するまでもなく、既に実装されて自治体で使われているものもある。次節で紹介するシステムもこのような法制執務の現場で有用なバージョン管理機能を備えている。

第二節　法制執務支援システムの実例

本節では、法令の公布や改廃の作業である法制執務の支援のために、実際に広く利用されている支援システム

の実例を紹介する。世界的に見ると、このようなシステムで著名なものはオーストラリア・タスマニア州政府のEnAct[11]である。ただし、日本とは制度や言語が異なるオーストラリアのものであるから、もし、何らかの形で日本に導入されたとしても、国内の自治体職員が容易に利用できるシステムではない。これに対し、日本には、国内の一五〇以上の自治体で利用されている『じょうれいくん』[12]と呼ばれる法制執務支援システムが存在する。これは当然、日本語によるシステムであり、日本の制度に則ったシステムとして実現されている。そこで、本節では、このシステムの概要について紹介する。

法制執務支援システム『じょうれいくん』は主に自治体向けにクレステック社によって開発されたもので、条例の公布から改廃までのプロセスを支援するシステムである。もちろん、大学など例規類を膨大に抱える組織でも利用されている。このシステムの本質的な機能は、条例のデータベースを基盤にして、条例の改正を行う際のメンテナンスを支援する機能であり、具体的には、改正を行う際の記述作業を支援したり、溶け込まし文や新旧対照表を自動生成したりする機能である。そして何より、これらの作業を統一的に進め、ウェブ上で多くの職員が共同で利用できる点も、従来の作業環境を改善することに役立っている。なお、このシステムの設計・開発に当たっては、担当のエンジニアが何ヶ月もの間、実際に、自治体に出向き、法制執務を行う職員と毎日一緒に仕事を手伝わせてもらう中で、要求を抽出し、仕様を固めて開発したものである。そのため、ユーザビリティは非常に高い。

まず、このシステムの内部構成の概要について記す。このシステムは条例データベース・システムを核にして、条文の形式的構造をシステムが把握し、法令特有の形の更新情報を管理したり、溶け込まし文の構文や語彙などを自動で補ったりする機能を加えたものである。なお、実際には、このシステムで作成された条例は、ウェブ上で公開できる形に直ちに自動変換される。その後、職員の方で公布許可のタイミングに合わせて公開すれば良い。もち

ろん、予約した日時に自動的に公開させることも可能である。

次に、このシステムで内部的に扱っているデータについて記す。『じょうれいくん』の核にあるデータベースに蓄積されるデータの形式はＸＭＬ表現をベースにしている。通常の法令データのＸＭＬ表現に比べると、改正情報が付加されている点が特徴的である。さらに、このデータ構造の表現の考え方は、法務省の法令外国語訳データベースのデータ形式にも採用されている。(13)

最後に、図12に『じょうれいくん』の画面のハードコピーを示す。ここで想定されているケースは、ある町の例規を改正する場合であり、これまで「助役」と呼ばれていた役職が「副町長」と呼ばれるようになった時に対応する事例である。この画面では、その例規の第五条３が元は「助役に事故があるとき、又は助役が欠けたときは商工会長がその職務を代理する。」と規定されており、この最初の「助役」の文字列を選択して「副町長」と編集することで、同じ条文内のすべての「助役」の部分が赤くなり消し線が引かれ、隣に「副町長」と青く表示される様子を示している（本稿ではモノ



図12　じょうれいくん®の画面

クロ）。このようにして、同じ語句を自動的に修正する機能を持つと同時に、このように消し線付きの状態で「新例規表示」の画面に表示されることで、非常に分かりやすくなる。これは「見え消し表示」と呼ぶ表示方法である。このまま、印刷することもできるが、もちろん、実際に新たな例規として公開したり印刷したりする時は、このような消し線の部分が自動的に削除されたものを表示する。さらに、この操作を行うだけで、改正文や新旧対照表などの原議も自動的に生成されるので、人手によるミスや煩雑な作業を避けることができる。この他、入力支援機能も現場に配慮されたものとなっており、法制執務用語への言い換え機能や誤字脱字を指摘する精査機能も備えている。法制執務用語への言い換え機能とは、法改正によって用いられなくなった用語について、変換候補を自動表示するような機能である。例えば、「収入役」と言う用語ならば、入力する際に「しゅうにゅうやく」とキーボードからタイプした時点で、漢字変換候補として「収入役」と言う単語の前に「会計管理者」と表示されて、そこから選ぶことができる。もちろん、その時に「会計管理者」という用語の意味も合わせて一時表示される。

このようなシステムが既に存在し、広まりつつあることから、本稿で提案する方法論やツールは、このようなシステムがカバーしている「提供」の段階よりも前の段階に焦点を当てており、このようなＩＴ化を徐々に立法過程の早い段階へも進めて行くことに本稿の意義があると考えている。

注

(1) ＲＤＦ（Resource Description Framework）の公式サイトは、http://www.w3.org/RDF/である。

(2) ＯＷＬはＲＤＦの語彙拡張であり、"Web ontology language"の頭文字の順番を入れ替えたものが由来とされている。詳しくは、http://www.w3.org/TR/owl-features/を参照のこと。

(3) 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。

(4) G・J・マイヤーズ他著、長尾真、松尾正信訳『ソフトウェア・テストの技法』第二版、近代科学社（二〇〇六年）六頁。以下『テスト技法』で参照する。

(5) 前掲、マイヤーズ『テスト技法』九頁。

(6) 前掲、マイヤーズ『テスト技法』二一－二二頁。

(7) 前掲、マイヤーズ『テスト技法』一一九－一四九頁。

(8) Kakuta, T. and Haraguchi, M., "A Demonstration of a Legal Reasoning System Based on Teleological Analogies", Proceedings of the 7th International Conference on Artificial Intelligence and Law (1999) pp.196-205.

(9) 角田篤泰「法廷弁論の論理的基礎とＩＣＴを利用した教育」情報ネットワーク・ローレビュー第九巻第二号（二〇一〇年）一－二一頁。

(10) Ｗ・Ａ・バビック著、菊池豊彦訳『プログラムのチーム開発入門―Ｕｎｉｘ／Ａｄａによるソフトウェア構成管理』ＣＱ出版（一九八八年）四〇－四一頁。

(11) EnActシステムの概要については、http://www.thelaw.tas.gov.au/about/enact.w3p を参照のこと。

(12) 『じょうれいくん』の概要や仕様については、http://lawinfo.crestec.jp/joureikun/product_jo.html を参照のこと。

(13) 法務省が提供するシステムであるが、開発は名古屋大学が担当した。ＵＲＬは、http://www.japaneselawtranslation.go.jp/である。