〈記録・文献・文書・資料〉

# ソフトウェア開発における上流工程設計技法入門

## ――システム科学的着想に基づくオブジェクト指向モデリング――

角　田　篤　泰

# Introduction to design methodology in the upstream process of software development

## —object oriented modeling based on systems scientific inspiration

Tokuyasu KAKUTA

**Abstract**

This paper is an introduction to design methodology in the upstream process of software development. Various theories proposed so far have become complicated because the design methodology should be practical. Therefore, it is difficult to serve as a guide for beginners in design work, and there are few that can be used as textbooks. In order to deal with this problem, this paper presents a design methodology focusing only on the orthodox essence of software development with a consistent view from the perspective of system science and abstraction. Moreover, the methodology is restricted to modeling activities only in the design process. That is, it can be also said that this paper provides a simple guideline for beginners.

**KeyWords**
Modelling, Software development, Object oriented, Abstraction, Systems science

目　　次

1. はじめに
2. モデリングとは
　2.1　モデリングの意味
　2.2　抽　象　化
　2.3　モデリング方法の概要
3. システムとは
4. システムの設計の方法
5. オブジェクト指向の着想
　5.1　初学者向けの具体例と開発手順
　5.2　背　　景
　5.3　典型的特徴
　5.4　人間の認識との親和性
　5.5　論理学との関連
6. ソフトウェア開発プロセスの進め方
7. ま　と　め

## 1. はじめに

本稿はソフトウェア開発における上流工程設計技法について，とくにモデリングについて入門的に示すものである．主に想定している読者は，ソフトウェア開発の上流工程における設計技能を修得することを目指す初学者，あるいは IT コンサルタントなどを目指す入門者である．そこで，本稿はソフトウェア開発の過程において，最低限の

オーソドックスな開発の各段階に通じる基本事項に留めて示すものとする．なお，本来豊富な実践例などを提示すれば，局所的な分かりやすさに貢献できるが，本稿で狙っているようなプロセスの全体を貫く考え方から初学者の注意を逸らしてしまうことが想定されるので，抽象度の高い文面になってしまうが，なるべく理論的本質から関心が外れないように記している．

ソフトウェアの設計はしばしば「システムの設計」と言われることがあるが，では，システムとは何かと問われれば，実は分からないまま，口にされていることが多い．本稿ではシステム科学的なシステムの捉え方から掘り下げてソフトウェア設計方法について示す．

ソフトウェアの学術的分野や開発現場では，「オブジェクト指向」という言葉を耳にすることが多いであろう．このオブジェクト指向による分析方法については，日々触れている技術者でも「システム」と同様に感覚的にのみ理解していたり，手順のみを覚えたりしていることが多い．しかしながら，設計センスを磨くためには，なぜオブジェクト指向が適切なのか，あるいは適切なのはどういう場合かなどのオブジェクト指向そのものに対する知見も必要である．そこで，本稿ではその理論的背景や哲学的背景を前提とした上でのソフトウェア設計を推奨したい．そのため，本稿ではこれらの背景についても解説する．なお，本稿では，ソフトウェア設計のモデリングとして，多くのオブジェクト指向プログラミング言語の前提に共通するモデリングはどのようなもので，どうのように行うかについて記している．

本稿の構成としては，まず，2～4章において，ソフトウェア設計の要となるモデリングの原理やシステム科学的見地から見たITシステムおよびその設計について示す．5章においてオブジェクト指向の基本概念や本稿におけるオブジェクト指向モデリングの方式について示す．6章において基本となるソフトウェア開発プロセスについて簡単に示した後，7章においてまとめる．

## 2. モデリングとは

### 2.1 モデリングの意味

モデルとは，語源的にはラテン語などの「小さい」と「ものさし」の意味の語の合成語が起源と言われているが，そもそも「模型」の意味も持つ語であり，現在でも「模型」という意味で使われることが多い．モデルは，例えばプラモデルのように，プラスティック素材で本物の3D外形のみ（中身が再現されることもあるが）を保ったものである．つまり，プラモデルでは形状という性質以外は本物とは異なる．このように，ある対象から特定性質を抽出することを「抽象化」という．逆に，抽出されないことを「捨象」という．飛行機のプラモデルであれば，大きさ，重さ，材質などほとんどすべては捨象されている．つまり，モデルとは，抽象化して出来上がったもののことを指す．そして，モデル化を実施する操作はモデリングと言われるので，抽象化操作こそがモデリングの本質である．日常的な言葉では，雑誌のモデルやファッションモデルという言い方で「モデル」という言葉が使われるが，これらでもある種の抽象化操作が実施されている．雑誌制作者が想定しているタイプの性質を兼ね備えた人，たとえば，理想の人，時の人，何より一般読者の典型の人をモデルとすることで，そのような人々の性質を抽象しているのである．この一般読者の典型を元々あこがれに近い方の性質にずらしておくことで，読者をそのモデルの気分にさせたり，あるいは目指すべき対象として設定させたりする効果もあるだろう．ここで，気をつけておきたいのは，その雑誌モデルとして紙面に現れると，俳優の演技と同じで，そのモデル個人の全性質ではなく，切り取られたいくつかの性質のみに着目される，ある種の非現実世界のできごとになっているという点である．これらの様子は，ファッションモデルでも同様である．新作の服の発表や服の販売促進が目的なので，重要視されるのは，服をよく見せることができる抽象化された人物の性質であり，その性質をまとった人物像をファッションモデルは担っ

ている．例えば，服をよく見せる形という観点からスタイルがよいことなどは必須なのであろう[1)]．

### 2.2 抽　象　化

抽象化操作で重要なことは，どの性質を抽象するかという観点があることと，さらに，その観点には人の認識が含まれるので，どういう意図があるのかという点である．もちろん，偶然なんらかの抽象化されたものになっているケースもあり得るであろう．芸術作品などであれば，例えば現代アートでは，その偶然的邂逅が本質となる作品もある．しかしながら，本稿ではモデリングという，人の行為のとしてわざわざ抽象化操作を実施する話なので，行為というからには，偶然ではなく，何らかの意図や目的を伴うものを対象とする．したがって，その意図や目的に基づいて，抽象される性質を選択し，それを抽出することでモデルを作成する（モデリングする）．すると，抽象化とは，観点に依って切り替わる類のものとも言える．そして，同じ対象であっても，何のためにモデルを作るのかという観点によって，どれが適切なモデルか，あるいは，どのようなモデリングが適切なのか，切り替わることになる．この意味で，曖昧模糊になってしまうが，モデリングには人の認識や志向性が付随するのである．

モデリングは抽象化操作である，とした場合，ソフトウェア開発におけるモデリングとはどのようなものであろうか．これも抽象化の観点から見直していく．まず，大きく２つの局面がある．特に用語法があるわけではないので，便宜的に「記述的モデリング」と「設計的モデリング」とする．記述的モデリングとは，プラモデルなどと同様に，現存する（した）ものに対する抽象化操作である．これに対し，設計的モデリングとは，建築物の設計図を描くときのように，まだ存在しないものについて，必要な性質を選択する場合である．具体物が存在しないので，モデリング時には，考えられ得る諸性質の中から必要な性質を抽出するという操作になる．もちろん，実際にそのモデルに基づいて制作された実物ができて上がってしまってからは，その制作物に対し，設計時のモデルはその制作物から抽象されたものになっている．

このような２つの局面について，ソフトウェア開発も制作物を作成する作業であるから，設計的モデリングが必要なことと考えるのは自然であろう．その一方で，記述的モデリングはソフトウェア開発の場面で必要なものなのだろうか．過去のソフトウェアを分析するために必要なのだろうか．その場合には，なぜ新たなものを作成する設計というプロセスで過去のものが必要なのかという疑問が残る．実は，過去のソフトウェアを分析することもあるが，多くの場合，現在アナログに実施されている業務，従来の旧システム，あるいはそれらの環境など，デジタル化（あるいはDX）の前提環境や対象についてモデリングする際に用いられるのである．そのモデルを基にして，目的に合わせた新しいモデルに改良していくことで，設計を進めるのである．このようなモデリングは，設計段階以前の要件定義段階でも登場することがある．すなわち，どのようなソフトウェアを開発すればよいのか，その条件を明確化する作業段階における，現状把握のためのモデリングとも言える．目標とするソフトウェアの備えるべき条件は，ある意味で現状に足りないものと言える．その現状の方を構成する諸性質のうち，条件記述やその考察に必要な限りにおいて，抽出する作業となる．

ここで，抽象化の特質を付記しておく．抽象化とは，実は，①匿名化であり，②置き換え可能であり，③共通性質を持つもののグルーピングでもある．①の匿名化とは，例えば，ある人の個人情報を開示できないときにその人について言及する場合，「ある男性」や「ある人」という表現で，その個体の持つほんの一部の属性に着目して，男性

1)　もちろん，ファッションショーと言いながら，旬な人々をステージやランウェイに登場させることで，服よりも，そのモデル自身にフォーカスするイベントもあるが，その場合は無理矢理「モデル」とも言えなくもないが，多くの人にとっては「出演者」くらいの意味で使われているので本稿で言うところのモデルとはずれが生じている．

であることや人であるという性質を抜き出して表現していることになる．これは本稿で示してきた抽象化操作そのものである．②の置き換え可能であるとは，抽象化された同じ性質を持つもの同士は，その抽象化の文脈では，区別できる他の性質に着目しないので，その文脈上は置き換えても問題ないということである．そもそも区別できないのであるから．③の共通性質を持つもののグルーピングとは，抽象化しておいて，結局，その抽象された性質を持つもの全体について言及しているのと同じことになるので，ほとんどの場合，その性質を共有するグループ（集合）を指しているのと同義になる．「抽象化」と言ったときには，これらの性質を自動的に伴う点にも注意して欲しい．

### 2.3　モデリング方法の概要

記述的モデリングの方法は，基本的に対象の持つ諸性質を確認して，それを記述することである．外部から観測できる性質を列挙すればよいが，可能なすべてを列挙するわけではない．物事は考えようによっては限りなく性質を見出すことが可能なので，ある程度は関係ありそうな性質に絞って確認していく必要がある．なお，対象物が内部構造を持つ場合，その構造も確認する必要がある．さらに，対象物の外部環境を前提にした方がよい場合も多々あり，その場合は，対象と外部の事物との関係性も確認することになる．これらの確認作業では，通常は記述作業が含まれるので，言語化（数式も含む）や図式化を伴う．もう少し詳しい方式については，本稿ではオブジェクト指向の着想に基づいた方法として5.章にて示す．

こうして確認された性質群の中から，自分たちの設計の目的に関して必要なもののみを取り出す．この抽出結果がモデルである．なお，記述的なモデリングの場合には，目的を多数見出し得たり，汎用な用途を目的とするモデルであったりすることも多いので，抽象化すべき性質を正確に確定するのは一般には難しいだろう．そのように目的を明確化できない場合は，標準性や典型性の観点から関連性質を抽出することになる．

一方，設計的モデリングの方法は，目的や意図を達成するために具体的に必要な性質や構造をすべて洗い出すことから始まる．もちろん，言語化も含む．この操作がそのままモデリングになっている．ただし，性質記述が曖昧だったり，実現性のない性質だったりすると，対象物の作成に至ることが困難となる．

これらのモデリングは文章のみで記述することも可能であるが，通常は図式化を伴う．近年典型的な図式としては，UML（Unified Modeling Language）が用いられることが多い．

## 3.　システムとは

システムと言っても，システムの種類にはいろいろある．機械的なシステム，例えば，自動車や列車，ロケットなどの乗り物，工作機械，ロボットなどを始め，その他にも，金融システム，医療システム，社会制度などもシステムであり，何より本稿読者には，ソフトウェアシステムが典型的であろう．システムの定義も様々な提案がなされているが，本稿では，共通項的な条件を取り上げて，次のように定義しておく．

［定義］

システムとは，何らかの意図・目的を持って動作するものであり，その各要素間の何らかの結びつきによって全体を構成する構造物である．

システムのイメージを図示すると図1のようになる．

なお，本来のシステム科学の学術書や論文などであれば，このような定義は，たとえ共通項から定立したと言えども，正確に数式化するものなのであるが，本稿では技術的基礎を学ぶ初学者という想定があるので，このような自然言語による定義とする．

図中のシステム内外を分ける境界線の部分は，実際には線や面が存在している訳ではなく，「界面」と翻訳されることもあったが，いわゆる「インターフェース」を意味している．システム内外

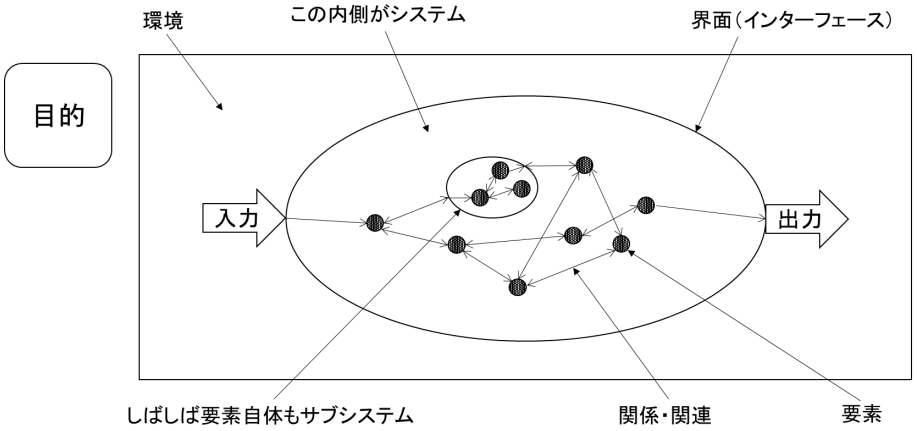


**図1**　システムのイメージ

のやり取りをする部分である．情報システムの場合には，入力と出力のことを指す．また，図中のシステムの内部に目を向けると，その要素の一つが拡大されて，内部構造が示されているが，このように要素自体がサブシステムになっていることも頻繁にあり得る．一方で，システムの外側の部分に目を向けるとそれは「環境」と呼ばれる．

この定義から，システムとは要するに，目的動作をする構造物ということになるが，ここで，注意して欲しいことは次の点である．

＜注意点＞システムとは認識論的存在である．

つまり，システムとは人の認識がなければ存在しないのと同じである．しかし，通常，認識できていないことはあまりないので，重要なことは，確固たる客観的なシステムは存在していない，という事実である．それはシステムとは常に人の認識に依存する部分があるので，その存在というより，むしろ存在形態について，安住できる合意点があるとは限らない，ということである．つまり，違う捉え方で対象システムを見ている人がいる可能性が常に存在しているということである．

なぜ，システムが認識論的であるかというと，まず，目的や意図という客観的には観測できないものが条件に入っていることである．次に，そのシステムを全体として中身をブラックボックス化しているときには，人々から同じように見えていても，その中身の構造については，何を要素と見るか，どういう要素間の結びつきを本質的なものと見るか，その点については見解が分かれる可能性があるからである．太陽系は英訳すれば solar system であるが，そもそも目的が不明なので，本稿で言うシステムかどうかがそもそも怪しいが，これを何らかの意図によってシステムと見るとき，太陽の周りを惑星が太陽の引力で回っていると見るような大雑把な認識とするか，各惑星の衛星やアステロイドベルト，あるいはその他の小天体も含めるのかでシステムの見え方は異なってくるし，いや，その空間も含める，ということであれば，太陽系自体も銀河の中で動いているので，一体，何を持って太陽系とするかは，その認識をする者ごとに状況や議論の文脈に応じて定義が変わる類のものである．もちろん，太陽や各天体は，自分のことをシステムの要素だと認識しているわけではない．太陽系のような物理的に強大で確固たるものであっても，システムとしての存在物と考えた瞬間から，考える対象，すなわち認識の対象になってしまうのである．このようなシステムに関する陥りやすい客観的絶対的な物体としてシステムを捉えてしまうという錯覚には常に注意すべきである．

もう一点，システムには重要な性質が指摘されている．それは，システムの持つ階層性である（サイモン 1999）．

＜システムの性質＞システムは階層的である．

これは，定義的なものではないし，帰納的な法則でもないし，数学的に厳密に導出された定理でもないが，典型的に観測される性質であり，ソフトウェアの場合には，ほぼ前提と言える性質である．すなわち，システムはサブシステムを要素とし，そのサブシステムもさらなるサブシステムを要素としている．そのさらなるサブシステムも要素としてサブシステムを持つ，のように階層的な構造になっているという事実である．なお，実は，対象としているシステムが，何かその外側に存在する大きなシステムのサブシステムになっているということも当然あり得る．

## 4. システムの設計の方法

前述のように，システム自体が認識論的存在であり，その認識のしかたによって，着目される性質が異なるような，抽象化されて認識されるものであるので，その設計にはモデリングが行われるのが自然である．ただし，システムというものの性質上，どのような設計方法が合理的かという点まで含めると，結果としてモデリングになっていればよい，というわけではない．前述のシステムの階層性やシステムは目的を持つという性質（志向性）より，設計は目的実現のための「段階的詳細化」によって実施されるのが自然であろう．

ソフトウェアのシステム設計における「段階的詳細化」とは，対象システムが完成していると仮定して，それを実現させる要素（サブシステム）をブラックボックスのまま洗い出し（まずは要素の内部は知らないままでよい），さらにその要素間の関係も洗い出し，その後で，各ブラックボックスの内部を対象にして同様の作業を繰り返し，この繰り返しでコーディング可能なレベルまで実施することである．これはトップダウンに実施されていると言える．なお，「洗い出す」としたが，設計作業は分析活動ではないので，新たに「作り出している」とも言える．一旦ブラックボックス化しているということは，内部の詳細な性質は一旦捨象して，トップダウンに必要な性質のみに着目しているので，これは抽象化していることになる．この抽象化されたブラックボックス部分を段階的に具体化していることになる．具体化とは，要するに性質を追加して，より細かく対象を捉えていることなので，詳細化していると言える．多分，直観的にもこの具体化と詳細化は同義に捉えられると思う．

もちろん，段階的詳細化でなくても，いきなり詳細なものから設計して，偶然，目的が達成できる設計になることもあるだろうし，同様に一番具体的なものから設計して，組み合わせ方を工夫しながら，目的にアジャストしていくという設計方法もあるだろう．例えば，アジャイル設計では，この後者の設計方法に近い手法が用いられている．しかしながら，これらの方法は試行錯誤的な手法で，具体的な細部の実現をベースに組み上げるものなので，少しだけ実現してしまうことになり，設計そのものの範疇をはみ出る部分もある．もちろん，そのような実験的な実現はまさにシミュレーションであり，それらを含めて「設計」と呼べるのであるが，そのようなシミュレーションの中でも，結局シミュレーションの前に，思考上の設計は実施されている．どのような詳細部分でも，必ず具体的作成の前に思考の中では設計フェーズが存在している．もしそうでない例外があるとすれば，それはランダムに作成して，偶然正しくなるのを待つという方法くらいであろう．これでは方法論とは言い難い．本稿で扱っているのは，まさに，その設計時の思考方法の部分なので，シミュレーションの場合，本稿で扱う設計の範疇をはみ出していると指摘しているのである．したがって，純粋な設計自体の方法論としては，トップダウンな段階的詳細化が合理的であると考えられる．

## 5. オブジェクト指向の着想

### 5.1 初学者向けの具体例と開発手順

本小節では，初学者向けに，オブジェクト指向に基づくプログラミングの本質部分にのみ着目した簡単な例を示しながら，プログラミング手順を概観する．

まず，現実のプログラミング言語は，様々な実用上の要請や事情から，話が見えにくくなってしまうので，Python 言語とほぼ同様の架空のオブジェクト指向言語 TinyObj を使うことにする．Python に詳しくなくても，何らかのプログラミング言語を少し知っていれば，特に難解なことはないであろう．

この言語で，例えば「犬」をモデリングして，その抽象した性質を実現する dog という概念（クラス）としてプログラミングし，その「犬」概念に属する「ポチ」という名の実例（インスタンス）を TinyObj システム上で作成して，そのポチを操作してみる．なお，個々のクラスやインスタンスのことを「オブジェクト」と呼ぶ．もちろん，オブジェクトはコンピュータ上で操作される実体なので，仮想的なものであり，本物の生物の犬のことではない．まさに，モデリングされた（抽象化された）実体なのである．もちろん，モデリングの対象は，生物である必要はない．図書や図書館，自動車やその部品，鉄道の場合なら，その路線や駅，世界に存在する様々な実体はもちろん，SF 的な架空の事物でもよいし，権利や法令，政治機構，ビジネスフローなどの概念的な存在でも構わない．

まず，モデリングとプログラミングを行ってみよう．ここでは，簡単な実験用のモデリングなので，コンピュータ上の仮想の犬，すなわち犬のオブジェクトに対して，どのような実験をするのか，それに必要な性質を抜き出すモデリングをすればよい．例えば，犬の名前を呼ぶと「ワン」と吼える様子をコンピュータ上で実現するようにモデリングしよう．この場合，犬のオブジェクトは犬ごと，すなわちオブジェクトごとに名前を持つことになる．つまり，名前という性質の値として犬ごとに具体的な名前が割り当てられるものである．また，名前を呼ぶという操作も用意しなければならない．この操作に対して，もし自分の名前と一致したら，画面に「Wann!」と表示する処理となるような操作をプログラミングする必要もある．もちろん，そのような音を発する実現も可能であるが，今回は簡単のために「Wann!」の表示だけとする．これらは，今回想定している犬の実例については，すべてについて一般化して備えるべき性質や操作である．そこで，個々の犬としてではなく，その汎用のテンプレート，あるいは個体の犬ではなくて，概念としての犬のイメージで捉え，「クラス」という形にまとめて，これらの性質や操作を記述することになる．この例であれば，TinyObj では，次のように表記する．

```
class dog:
  def self.dog（Name）:
    self.name = Name
  def self.call（Name）:
    if Name==self.name:
      print（'Wann!'）
    else:
      print（'?'）
```

半角文字に限定された仕様なので，クラス名は dog にしている．最初の def の行は，インスタンスを作成する時に初期値を設定するための操作定義である．後で説明しやすくするために TinyObj では，クラス名と同じ操作名としている．self というのは，インスタンスが自分自身を表すために使われる変数である．次の行がその操作の処理のプログラム本体である．左辺は自分自身に name という性質（プロパティ）を持たせるという意味であり，そこに初期値に設定する Name という変数に入れられた文字列を name というプロパティに設定している．これで，プログラム内部では，self.name と書けば，自動的にここで設定した名前の文字列を得ることができる．2 番目の def の部分は名前を呼ぶという操作 call である．ここでも漢

字が使えないので「呼ぶ」でなく「call」とした．なお，ここでの Name という変数は上の def 内の Name とは別の話として使われている．もちろん，同じ値が入ってくる場合もあり，その場合は，まさに次の行の if 文で，name プロパティに設定された名前の文字列とその Name 変数の値の文字列が一致する場合であり，「Wann!」と表示させる．一致しない場合は「?」を表示させる．こうして，作成したクラス定義のプログラムを TinyObj インタープリタ上でシステムに登録すると，システム内には dog というクラスが作成される．

次に，実際に操作してみる．インタープリタ上で，dog を登録した後，次のように入力する．

```
X = dog（'Pochi'）
```

「ポチ」という名前のインスタンスを作成するので，dog のクラスの生成・初期化の操作の方の dog を呼び出して，その結果作成されるオブジェクトを X という変数に入れている．ここで，X の値を直接表示させても，システム上の ID が表示されるだけである．犬の画像や設定した名前が見られるわけではない．ここで，Hachi という名前の犬のインスタンスも作成しておこう．

```
Y = dog（'Hachi'）
```

この状態で，次のように X に「Pochi」と呼びかける操作（call）をする．

```
X.call（'Pochi'）
```

すると，「Wann!」と表示される．そこで，今度はその X に別の名前で「Hachi」と呼びかける．

```
X.call（'Hachi'）
```

こうすると，今度は「?」と表示されてしまう．インスタンスが違うからである．これに対し，当然，

```
Y.call（'Hachi'）
```

とすれば，「Wann!」と表示され，

```
Y.call（'Pochi'）
```

とすれば，「?」と表示される．このように，クラス定義は同じでも，インスタンスを生成するごとに別の実体として，コンピュータ上に存在することになる．

ここで，継承というメカニズムを利用してみる．例としては，動物のクラス animal を作成して，犬は動物の下位概念（サブクラス）なので，犬は動物の性質や操作を継承するものであるから，その継承の様子を実現するのである．

例として，動物なら成長して，歳を重ねるものなので，1 年分の成長を一気にさせる操作を grow として定義して，年齢の値が設定されるプロパティ old を grow 操作 1 回につき，1 年加算して増やす（歳を取る）処理にしておく．また，その年齢を確かめる操作 howOld も用意しておき，その操作によってその時点の年齢，すなわち old の値が表示されるものとする．animal のクラスのプログラム例は次の通りである．

```
class animal:
  def self.animal（）:
    self.old = 0
  def self.grow（）:
    self.old = self.old + 1
  def self.howOld（）:
    print（self.old）
```

こうしておいて，犬のクラスのプログラムの第 1 行目だけを次のように書き換えたプログラムに更新しておく．

```
class dog（animal）:
```

これで dog は animal のすべての処理やプロパティ

を自動的に継承してくれる．インタープリタを再度立ち上げなおして，まず，animal のクラスを登録し，次に，それを継承する形の dog のクラスを登録すれば，次のように動作する．画面入力や出力をそのまま示したものである．

```
Z = dog（'Taro'）  #1
Z.howOld（）       #2
0                  #3
Z.grow（）         #4
Z.grow（）         #5
Z.howOld（）       #6
2                  #7
Z.call（'Taro'）   #8
Wann!              #9
```

それぞれの行末の「#」と番号はコメントであり，実際には入力もしないし，表示もされない．それらの番号に対する具体的コメントは次の通り．

#1: Taro という名前の犬のインスタンスを生成する．
#2: dog が継承する animal の操作を呼び出す．
#3: まだ成長していないので，0 歳の 0 と表示される．
#4: 1 歳分，成長させる．
#5: もう 1 歳分，成長させる．
#6: 再び年齢を問い合わせる操作．
#7: 今度は 2 年分成長させられたので，2 歳の 2 が表示される．
#8: もちろん，犬の操作も使える．
#9: 名前どおりなので，Wann! と吼えた．

このような継承のメカニズムのありがたみは，犬のクラスしか動物を継承するものがないと，あまり感じないが，これが，猫のクラスや猿のクラスなど，沢山の動物の種類のクラスがあった場合，animal の内容を何度も記述する必要もないし，animal のクラスの仕様変更があった場合でも，すべてのクラスの該当部分を書きなおす必要はなく，animal のクラスだけ書き直せば済む．継承メカニズムにはこのようなメリットがある．

以上のように，対象物（オブジェクト）を設計して作成しておいて，その対象物を操作する，このようなイメージでソフトウェアを構成していくと，実世界の箱庭のようなモデルを構成するようにして，作成・操作できることが分かるであろう．

### 5.2 背　　景

オブジェクト指向とは，ソフトウェア開発手法やプログラミング手法における物事の考え方（「パラダイム」という）の一つであり，ソフトウェアを構成する対象物を「オブジェクト」というデータと操作を一体化したものと見て設計や開発を進めるものである．なお，その際に，一体化したことで内部をブラックボックス化して，むしろ，外部からオブジェクトを操作する際には，具体的な内部状況や内部の性質は見ないことにして，外側に提示された性質や操作のみを用いることが求められる．これは，いわゆる「カプセル化」や「情報隠蔽（information hiding）」と呼ばれる．このことによって，そのオブジェクトを使う側からすると，必要な性質のみを抽象してモデリングしていることになり，その抽象された性質のみに注力するだけで済むのである．

本小節では，オブジェクト指向自体の誕生や様々な経緯については他書に譲ることにして，オブジェクト指向が受け入れられ，むしろ盛ん取り入れられるようになった背景について概観を記すことで，オブジェクト指向が使われる意図について確認しておきたい．

ソフトウェア開発方法の歴史は，最初はプログラマがアルゴリズムを考えて，それをプログラミングすることで完了するだけのシンプルな方法であった．そもそも初期のコンピュータの時代には，大きなソフトウェア開発はハードウェアの制限からできなかった．しかしながら，1950 年代後半～1960 年代になると，次第に大規模なソフトウェア開発が始まり，一つのソフトウェア開発に複数の人間が関わったり，長いスパンの中でソフトウェ

アを何度も利用したりする機会が増大することとなった．そうなると，まず，アルゴリズムやプログラムの作成について，GOTO 文を使った処理の流れが入り乱れる，いわゆる「スパゲティプログラム」と呼ばれるプログラムが問題視され，E・ダイクストラによって，「構造化プログラミング」が提唱されて（Dijkstra 1968: 147-148，Dijkstra 1970: 84-88），順接，分岐，反復のみでソフトウェアを記述しよう，という方法論が生まれた．彼は，他にもトップダウン設計や段階的抽象化，そして階層的なモジュール化，および「抽象データ型」という，現在にも通じる，そして本稿でも中心的に紹介している概念も提唱した．なお，この後，IBM 社主導で「構造化設計」という技法も発展している[2)]．「構造化設計」を題名に取り入れて，ソフトウェアをどのようにモジュールに分けて設計・開発していくべきか，上記の考え方をベースにまとめられた G・マイヤーズらの著作は教科書のように用いられた（マイヤーズ 1979）．そこで解説されているモジュール設計に関わる「モジュール強度」と「モジュール間結合度」の指標は著名である．モジュールとは簡単に言えば，ソフトウェアを構成する各パーツのことである．本稿の用語法にしたがうと，ソフトウェアをシステムと見た場合のサブシステムと言うこともできる．

ここでは，オブジェクト指向の発想に通じる，モジュール設計や抽象データ型（あるいデータ抽象）の考え方について簡単に示しておく．

まず，「モジュール」とはソフトウェアを部分に分けて設計したり，開発したりする際のその分けられた部分のことをいう．このように部分に分ける目的は非常に大雑把な言い方をすれば，分かりやすく設計し，分かりやすく動作を理解できるようにすることである．動作理解がしやすいとデバッグ（＝ソフトウェアの不具合修正作業）もやりやすくなる．大規模になり，対象物が大きくなることに加え，関係者も増えて，複数の人々の共同作業となることから，コミュニケーションを円滑することは必須である．現在でも改訂を重ね，読み継がれているソフトウェア開発の古典的名著，F・ブルックスによる『人月の神話』（ブルックス 1995）でも，旧約聖書のバベルの塔の逸話を挙げて，複数の人手による作業の本質がコミュニケーションにあることを指摘し，ソフトウェア開発でのコミュニケーションを円滑にすることの重要性を説いている．

では，そのモジュール分けの目的に沿って，より適切な分け方はどのようなものであろうか．当然，コミュニケーションの円滑化が目的であるが，そもそもコミュニケーションコストを減らせれば，問題が減少する可能性が高い．とくに必須のコミュニケーションを減ずる方法があれば，全体的に問題を減少させることに寄与できるであろう．そのような方法は，モジュールの独立性を高めることにある．モジュール間での操作や参照のやり取りが減れば，モジュールごとに割り当てられている開発者の人々は，他のモジュールのことを気にしないで，すなわち，コミュニケーションをとらないでも，自分のモジュールの開発に専念できる．そのためには，各モジュールが最低限，外部に操作させたり，参照させたりするものは何かを確定して，それ以外は隠してしまえばよい．これはカプセル化である．こうしないと，あるモジュール A のデータを別のモジュール B の操作によって書き換えたり，参照したりすることで，A の開発者が思ってもいない挙動をすることも起こり得るが，構造化設計が推奨される以前は，それを防ぐためには A も B も常にお互いのやることを連絡しあったり，何か設計を変更する際にもお互いの承認を取り合ったりする必要があった．それがモジュール数の分だけ必要となっていたのであるから，コミュニケーションはもちろん，設計・開発が混沌としてしまう．そこで，そのような混沌を避けるには，各モジュールの他のモジュールからの独立性を向上させること大切であり，それが提唱されたのである．そのような独立性を高めるための指標が「モジュール強度」や「モジュール間結合度」である．

2)　ダイクストラの意図とずれてきたという見解もある．

「モジュール強度」はモジュールが提供する諸機能やデータのうち，一つのモジュールにまとめておくべき度合いが強い状態を良いものとする指標である．この強度が高いほど，まとまりとして適切であることになる．最上位の結合度合いは「機能的強度」と「情報的強度」とされている．前者は，単一の機能実現のためのみの諸機能をまとめたものである．後者は，特定のデータを中心に，そのデータにアクセスできる操作・参照用の機能をまとめたものである．これによって，内部を隠すことができるのでカプセル化と言えるが，データの具体的な態様を隠して，与えられたモデルに対する操作として抽象化されたデータを操作することになるので，「抽象データ型」と呼ばれるタイプのデータを提供していることにもなる．抽象データ型とは，まさに具体的な内部のプログラミングに依存しないモジュール外部に提示した操作によって現れる性質だけが抽象化されたデータタイプである．もちろん，モジュール内部では具体的に実現されている．なお，これらより低い強度のモジュールも段階的にいくつも提示されてはいるが，特に事情がない限りは，そのようなモジュール設計はやり直して，機能的強度か情報的強度になるように再設計すべきである．

「モジュール間結合度」は，モジュール強度が内部の結合の観点であったのに対し，モジュールを外から見て，モジュール同士ではどのような結びつきが好ましいのか，適切さの度合いである．もちろん，強く関連しているモジュールは，そもそも一つのモジュール内にまとめて強度を増すべきであるから，それを前提とするなら，個別のモジュールになっている時点で，モジュール同士では結びつきが弱い方が，各モジュールの独立性を高めることに貢献できる．そこで，最も適切である，モジュール間結合度の弱い形態の指標は「データ結合」と呼ばれるものである．データ結合とは，例えば，関数呼び出しで，f（x）の x に数値データや文字列などの構造を持たないデータを用いて，他のモジュールから操作・参照させる場合である．つまり，特に詳細な取り決めをしないで済むデータのみが，モジュール間でやりとりされることになり，設計・開発時の各モジュール担当者のコミュニケーションコストを下げることができる．これがもし，構造データのやり取りとなってしまうと，データについての仕様をモジュール間で共有することになり，独立性は少し低くなる．もちろんコミュニケーションコストも増える．このような少し高い結合状態を「スタンプ結合」と呼ぶ．さらに，独立性が低まり，結合度が高くなるものは「制御結合」と呼ばれるもので，対象モジュール内の処理の方式を選択させるような引数で関数呼び出しをさせる時の結合度合いである．そもそもこのような設計はよほど特殊な事情がない限り，採用すべきではない．さらに高い結合度のものも指標としては示されているが，そうなる前に設計を見直すべきである．

こうして，モジュール間の結合度を下げ，各モジュールについてはその強度を上げると，独立性が高まるのである．また，これを感覚的言えば，仲間はより近く，仲間でない者はより遠くに置くことになり，メリハリがついた状態になる．これは人にとっても，認識がしやすい状態と言える．

こうしたモジュール設計の方法論の中で推奨されてきたモジュール，すなわちカプセル化された抽象データ型で存在するパーツこそが，オブジェクト指向で言う，オブジェクトである．

オブジェクト指向自体の誕生の意図や発想の源泉は，実はこのような抽象データ型のような話よりも，むしろ，シミュレーションのエージェントのようなものとしてオブジェクトを捉え，それらのメッセージのやり取りで，自動的にソフトウェアの動作が進むという発想であった．その発想のような動作も実現可能ではあるが，多くのオブジェクト指向設計方法の主眼にはなっていない．それゆえ，何がオブジェクト指向の本質か，という論争になることもある．本稿では，オリジナルのオブジェクト指向の生みの親には申し訳ないが，抽象データ型のイメージや実践的に利用しやすい性質に着目し（本質性を見出し），説明を進めていくものとする．

### 5.3 典型的特徴

オブジェクト指向の設計やプログラミングが採用される背景事情は前述の通りであるが，その経緯にはここで示した以外にも様々な思惑や理念が含まれているので，ここでは，オブジェクト指向の設計において，典型的に観察される特徴を挙げることする．オブジェクト指向の定義としてしまうと議論が分かれて初学者には混乱を招き兼ねないからである．

オブジェクト指向の典型的特徴は次の通り．

- **オブジェクトという見かた**
  一つの個体として捉えることのできる対象をプログラム（操作）群とデータ群からなるオブジェクトという対象物としてまとめて見る．特徴というより，定義的な不可欠な性質（＝本質）である．
- **カプセル化**
  オブジェクトの中身を外部に見せない．オブジェクトは抽象データ型として扱うことが可能である．
- **クラスとインスタンス**
  オブジェクトはプロパティと操作を定義した，クラスという雛形で定義される．プロパティとは，オブジェクトごとに内部状態として保持できる変数のようなものである．プロパティはいくつでも定義できる．実際の動作時は，クラスの実例をインスタンスというオブジェクトとして生成し，そのインスタンスに対する操作としてプログラムが実行されていく．操作はオブジェクト指向のプログラミング言語では「メソッド」と呼ばれることがしばしばある．ほとんどの場合，対象のインスタンスを第1引数にすることと，そのインスタンスに属する操作である，というだけで，プログラムの記述は通常の関数と同様に行うことができる．
- **継承機能**
  上位クラスとして指定したクラスの操作や性質を引き継ぐことができる．

この他，「多相性（ポリモルフィズム）」が挙げられることもあるが，一方で多相性は抽象的には扱いやすくなるが，現実のプログラミングではむしろ混乱を招くこともあるので，議論の分かれるところでもあるし，実装されないこともあるので，本稿は初学者を対象にしていることも勘案して，ここで列挙した4点のみを特徴としておく．

これらのうち，カプセル化については，オブジェクト指向の興隆に至るまでの経緯からしても，前述のように元々必要性の高かった性質と言えよう．また，クラスとインスタンスというソフトウェアの捉えかたは，概念とその実例という，我々の認識にも適合した物事の捉え方であるので，ソフトウェアの可読性や再利用性などに貢献できる．ダイクストラも検証できること，すなわち可読性の良いプログラムを記述すべきである，ということが元々の大きな主張であった．これらに対し，継承機能は，ほとんどのオブジェクト指向プログラミング言語でも実装されてはいるし，うまく使われれば便利な機能ではあるが，そもそも多重継承問題や，モジュール独立性に本当に貢献できるか，などの議論も存在する．プロトタイプのプログラミング時にはとても有効なのだが，大規模な現実的ソフトウェアになると逆に手間が増えてしまうこともある．なお，多重継承問題とは，2つのクラスを継承するとき，その2つのクラスが矛盾する操作や性質を伴っていたときの競合解消問題である．通常，プログラムには継承優先順位を書くので，その順，あるいは逆順で，上書きされたり，優先度が決まったりすることで解決されるが，そもそも継承を実現するメカニズムは，矛盾を孕むものを導き入れてしまうものなので，安易に利用されると混乱を招きやすい．しかしながら，実際には多くの現場で安易に利用されており，後から不満が出ることもある．

### 5.4 人間の認識との親和性

コンピュータというものが誕生した当時のプログラムは命令を列挙するものであり，表記上，自分が読みやすいようなパラグラフ化などはしてい

たかも知れないが，最初から最後まで繋がったものであり，その後，サブルーチンや関数を一部に使うことで，分節化が図られ，部分の認識もしやすくなり始めたと言える．このように連続的に繋がったものをどう分節化して切り分けて，部分ごとに認識をしていくか，これがうまくいけば，プログラムの理解，そしてそれを前提とする共有や共同作業が円滑に行われるようになる．

そこで，物事の認識の原理や認識のしやすい状況について考察を進める．本稿ではシステムの捉え方の中で，システムが認識論的存在であることを強調しているので，その観点からも認識について述べることにする．ただし，本稿は哲学書でもなければ，心理学研究書でもないので，本稿の趣旨に関係する，とくに物事の区別との関係性から，浅いレベルの認識について記すことにする．

まず，初期のコンピュータの時代のプログラムに限らず，我々の日常的な生活空間も実は連続的に繋がっているが，それを物体の配置などによって，区切って認識している．そこで，物体がうまく配置されていないと，例えば，海の上では肉眼での国境の線は判別できず，直ちには国の区別ができないであろう．このように我々は物体をうまく使って区切っている．そもそも物体としての認識自体も，空間と物体を切り分けていると言える．下等生物にとっては，山腹に建てられた家も山肌に沿って張り出した岩も区別はないだろう．哲学や言語学の話だけではなく，量子力学的にも物体の持つ排他性や自己同一性などは超ミクロのレベルでは意味がなくなってしまう．結局のところ，あくまでも我々の便宜的な都合や習慣，あるいは本能的に組み込まれた脳や器官のしくみがモノ的な認識や分節化を成立させているのである．

19 世紀にはロウソクの炎は物質であると仮定されていて，その物質は「フリギストン」と呼ばれていた．しかし，その後，酸化還元現象で光っている状態が連続的に続いているので，物体のように見えてしまう，ということが分かったのである．このように人は，本来は出来事などの「コト」的に存在しているものを「モノ」的に見てしまうことで，むしろ認識しやすくしている．つまり，真実はともかく，真実を曲げても，日常的にはモノ的に見てしまいがちであるし，むしろ物事をモノ的に見る方が認識しやすいのである．実際，PC やスマートホンの画面に，ボタンやウィンドウが表示され，それをあたかもモノが存在しているかのような錯覚を利用して，我々は操作を行っている．これらのボタンやウィンドウはご存知のように細かいドットが様々な色や明るさで点灯していて，その集合体による状態なのであって，物理的なボタンのような「モノ」ではない．現象として状態が変化している「コト」なのである．それでもやはり，画面上のボタンはモノとして認識しておいた方が日常的には都合がよいだろう．

こうして，プログラミングにおいても，プログラムがシステムの連続的状態を表すような表現であったりするよりも，「モノ」的発想の上に立って，その構成物としてプログラムやソフトウェアを捉えることができる方が，人間の認識，あるいはその認識方法の傾向に対して，親和性が高いと言える．このような観点もオブジェクト指向を採用する一つの根拠になり得る．

### 5.5　論理学との関連

既にオブジェクト指向の特質について記したが，その中で，そもそもオブジェクトをクラスで定義したり，その定義の中でも，プロパティと操作という 2 つが定義すべき項目になったりしたのはなぜだろうか．偶然，このような定義の仕方になったのだろうか．実は，これも人間の思考方法に沿った方法論になっていて，我々の認識方法とも相性がよい．さらに，それは形式論理学で最も典型的な述語論理表現とも相性がよい．この他にも，情報数理系のオートマトンの理論などにも親和性がある．本小節では，特に述語論理および日常的な我々の言語活動との親和性について記す．

人間が用いる様々な自然言語の多くが，文の中に主語と述語を持つ．何百年も前は，特に主語や述語という文法的捉え方をしていなかった言語であっても，さすがに現代のようにグローバル化さ

れ，西欧の言語との交流がある中では，主語や述語の概念は浸透しているだろう．この日常言語的に使われる文の中にある，主語や述語の役割こそ，クラスやインスタンス，それらに属する，プロパティや操作などの概念の性質に対応するものなのである．

まず，物事を言語表現するための考え方の枠組みとして，「概念」「個体」「性質」について述べる．「概念」とは，種類，カテゴリ，型（タイプ），クラスなどとほぼ同義であり，具体的個別的な実体である「個体」をまとめて考えた時の観念上のものを指す．まとめて考えることは，本稿で示してきた抽象化を行うことに他ならず，概念に属する各個体がその抽象された性質を共有していることになる．つまり，特定の性質，あるいは特定の性質の集合から概念は形成されていると言える．

具体例で考えてみよう．日本語で「りんごは赤い」という文があったとき，正確には，りんごであるようなものがあれば，それは赤い，という意味である．このとき，「りんご」は概念であり，「赤い」はその概念の持つ性質である．その概念に属する「個体」でありその概念の「実例」とも言える各りんごも当然「赤い」性質を持っている．この例文の場合，もし，「りんご」なるものである個体が存在すれば，それは当然「りんご」の実例である．このような個体は世の中，過去・未来，想像上の世界，すべてに登場する「りんご」と呼ばれる個体すべてに対して言及していることになる[3)]．

ここまでの説明で，オブジェクト指向の「クラス」は「概念」に相当し，「インスタンス」はそこに属する個体である「実例」に相当し，さらに性質は「プロパティ」に相当することが分かるであろう．

主語・述語の関係で述べると，述語や述部に来るものは，性質や概念を指す．主語は通常は個体である．ただし，何の概念の個体かが特定されていない場合，日本語であれば，そのまま概念名を主語にしてしまい，英語であれば，不定冠詞を概念名に付す．特定のりんごでなければ，「an apple」が主語になる．apple は概念名である．つまり，英語だとそもそも概念と個体を分けているのである．日本語は雑なので，概念のりんごも個体のりんごも「りんご」と言うし，とくに具体的に特定のものであることを明記する場合は「そのりんご」などと指示語を付す．なお，概念自体について言及する場合もあるので，その場合は，日本語でも英語でも，概念名のままになっている．

ここで，操作（メソッド）についても記しておく．操作も対象クラスのインスタンスであれば，すべて同様に保持するものであるから，一種の性質であると捉えていい．プロパティであれば，例えば，「りんごは赤いという性質を持っている」のように表現できるので，直接保持している性質と言えるが，操作の場合は，モデリングの過程で，可能な操作の中から選択された操作であり，許可された操作とも言えるので，強いて言えば，例えば「りんごを剥く」とは「りんごは剥かれることができる性質をもっている」とすることで，これも一種の性質と捉えることができる．さらに，操作によって結局動作することになることになるので，述語が記述的なものではなく，動作的なものの場合には操作になるのだと考えれば，モデリング時の迷いもだいぶ解消されるだろう．物事の事実関係を主語・述語による基本構成を核に文で表現する際に，感覚的には「～である」状態表現に対応するものがプロパティに対応し，「～する」「～される」などの動作表現に対応するものが操作であると考えればモデリングの発想時のトリガになるであろう．

こうして，日常的な我々の言語的認識の中に自然にオブジェクト指向の考え方は入り込んでいるので，モデル化するときはそこを意識的に行うと理解しやすいモデルを構築することができる．

なお，主語・述語の話は，そのまま数理論理学の典型的な枠組みである，第一階述語論理にも通

3)　オランダでの国際会議で「りんごは赤い」と言ったら，西欧では「りんごは青い」と言うのがメジャーだ，と指摘された．そこで，本来は「りんごが赤い」の「りんご」は，「すべてのりんご」という意味にとらないほうが正確である．

じる．本小節では，これまで自然言語による説明を試みてきたが，主語・述語に着目しているので，ここで述語論理表現の説明を試みる．

先ほどのりんごの例の場合であれば次のような論理式表現が可能である．

∀ X りんご（X）→ 赤い（X）

この式は「すべての X について，X がりんごであれば，X は赤い」という意味である．

これから「りんご」概念は「赤い」という性質を見出してもよいし，実は赤いという性質を抽象した概念である「赤い（もの）」のサブクラスがりんごである，という解釈も可能である．英語はうまくできていて，「red」が名詞的に「赤」や「赤色」の意味に使われたり，形容詞的にまさに「赤い」という意味にも使われたりする．つまり，

∀ X apple（X）→ red（X）

と書けば，red という性質を持つとしても，red という概念に apple が所属するとしても，解釈を変えるだけで，日本語のように「もの」を補う必要なく，切り替えることができる．なお，このような様子から，実は，概念の実質は性質である，ということが露呈している点も付記しておく[4]．

先ほどの剥くという操作は単に，

∃ X りんご（X）→ 剥かれる（X）

と表現することも可能である．

こうして論理式や自然言語文での事物の表現ができれば，それに対応させてオブジェクト指向のモデリングも容易になる．

以上のような性質があるので，オブジェクト指向における表現は，知識工学という分野でかつて提唱された知識表現である，フレーム知識表現やこの論理式表現とも親和性が高いものであった．論理的に見ても妥当な定義がされていれば，論理的な推論の素材としてもオブジェクト指向に基づく表現で形成されたデータは生かされる可能性が高い．

## 6. ソフトウェア設計プロセスの進め方

ソフトウェア開発は次の図 2 のような工程で進む．このうち，内部設計までが上流工程である．本稿で紹介したモデリングが主に活かせるのは上流工程である．

このような手順に厳密に従う開発方式は「ウォーターフォール（落水）」型の方式と呼ばれる．実際にはこのような手順とは異なる開発フローを採用する方式もあるが，本稿では割愛する．この上流工程では，様々な局面でモデリングが必要になるが，下流工程のコーディングが始まるまでは，段階的詳細化によって一貫して進めることができる．つまり，各工程を進むごとに，前の工程を受けて，詳細化していることになる，ということである．これは，システムがサブシステムからなる階層構造であることを受けているものである．すなわち，上流ほど，抽象度が高いモデリングであり，どの工程でも内部をブラックボックス化するため，具体的な内部の性質が捨象されており，そこを工程が下流に進むごとに，拾い上げていくイメージで開発が進む．オブジェクト指向のモデリングは上流工程を中心に実施されるが，詳細設計はもちろん，オブジェクト指向プログラミング言語を用いれば，コーディングまで一貫してモデリング時の抽象化の視点や発想を維持できる．

## 7. ま と め

本稿ではソフトウェア開発における上流工程設計技法について，システム科学的着想に基づいた，設計の全体像と設計のためのモデリング方法論を示した．特にオブジェクト指向のモデリングの発想について示した．

本稿は，当初，モデリングだけでなく，ソフト

4)　このことから，ある概念 A の持つ性質群は，逆を言えば，その性質群の各性質と等価な概念の集合の交わりの部分の中にその A と言う概念が存在している，とも言える．

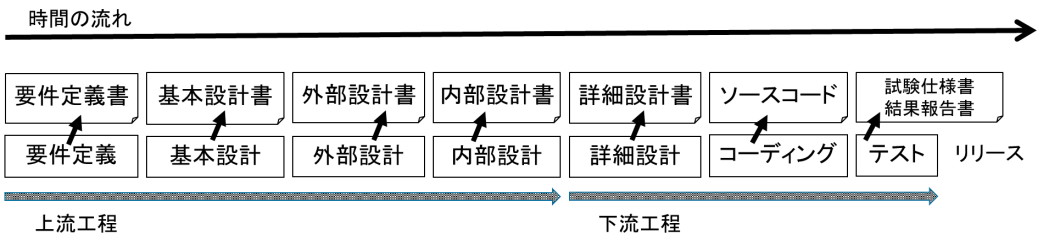


**図2**　ソフトウェア開発工程のフロー

ウェア開発全体のシステム科学的な理論を前提とした入門事項を提供するつもりであったが，紙面の関係で，モデリングのみに制限した．次の機会には，本稿で示した設計論の全体像とモデリングに基づいた各段階での設計方法を詳しく提示して初学者の勉学や開発に貢献したい．

## 参考文献

Edsger Wybe Dijkstra, “Go To Statement Considered Harmful”, Communications of the ACM, 11,3, 1968.

Edsger Wybe Dijkstra, “Structured Programming”, Software Engineering Techniques, B. Randell and J.N. Buxton, (Eds.), NATO Scientific Affairs Division, 1970, pp.84-88.

H・サイモン（著），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，3版，1999）．［※原著：Herbert A. Simon, “The sciences of the artificial”, MIT Press, 1996.］

G・マイヤーズ（著），国友義久，伊藤武夫（訳）『ソフトウェアの複合・構造化設計』（近代科学社，1979）．［※原著：Glenford J. Myers, “Composite/structured design”, Van Nostrand Reinhold, 1978.］

F・ブルックス（著），滝沢徹［ほか］（訳）『人月の神話：狼人間を撃つ銀の弾はない』（アジソン・ウエスレイ・パブリッシャーズ・ジャパン，増改版，1996）．［※原著：Frederick P. Brooks Jr. , “Mythical Man-Month, The: Essays on Software Engineering”, Addison-Wesley Professional, 2nd edition, 1995.］