JNC TN9420 2002-004

工学系モデリング言語としての次世代解析システムの開発(I)

—課題および要素技術の調査—

(調査報告書)

2002年11月

核燃料サイクル開発機構
大洗工学センター

JNC TN9420 2002-004
2002年11月

工学系モデリング言語としての次世代解析システムの開発（I）

—課題および要素技術の調査—

（調査報告書）

横山 賢治[1]、細貝 広視[2]、宇都 成昭[3]、笠原 直人[4]
名倉 文則[5]、大平 正則[6]、加藤 雅之[6]、石川 眞[1]

要 旨

高速炉開発において、解析コードを用いた数値シミュレーションは、理論、実験を補足するための重要な役割を果たしている。研究開発に対するニーズが多様化し、解析対象がより複雑化している現在では、工学的なモデルや解析手法を柔軟に変更したり、新たなモデルや手法を開発して容易に従来のシステムを拡張したりできることが、解析コードに求められる重要な要素となってきている。また、原子力に関連する技術分野は多岐にわたるため、多くの異なる分野の物理データや工学的モデル・手法を、いかにうまく結合して利用できるようにするかという点に大きな課題がある。

本研究では、原子炉の解析で必要となる物理量・解析手法等の工学上のモデリングの概念を、人間・計算機ともに理解できるプログラミング言語、あるいは、ダイアグラムの形で表現できるようにすることで、柔軟性が高く、かつ、汎用的な次世代解析システムを開発することを目標とする。この次世代解析システムの新しい概念を、工学系モデリング言語と名づけることとした。

本報告書は、この工学系モデリング言語としての次世代解析システムの実現のために利用可能と考えられる最新の計算機技術、ソフトウェア開発技術等を調査した結果をまとめたものである。

---

1 大洗工学センター システム技術開発部 中性子工学グループ
2 常陽産業株式会社
3 大洗工学センター システム技術開発部 炉心・燃料システムグループ
4 大洗工学センター 要素技術開発部 構造信頼性研究グループ
5 大洗工学センター システム技術開発部 管理グループ
6 原子力システム株式会社

JNC TN9420 2002-004
November, 2002

Development of the Next Generation Code System As an Engineering Modeling Language (I)

Kenji YOKOYAMA[1], Hiromi HOSOGAI[2], Nariaki UTO[3], Naoto KASAHARA[4]
Fuminori NAGURA[5], Masanori OHIRA[6], Masayuki KATO[6], Makoto ISHIKAWA[1]

Abstract

In the fast reactor development, numerical simulation using analytical codes plays an important role for complementing theory and experiment. It is necessary that the engineering models and analysis methods can be flexibly changed, because the phenomina to be investigated become more complicated due to the diversity of the needs for sresearch. And, there are large problems in combining physical propaties and engineering models in many different fields.

In this study, the goal is to develop a flexible and general-purposive analytsis system, in which the phisical propaties and engineering models are replesented as a programming languare or a diagrams that are easily understandable for humans and executable for computers. The authors named this concept the Engineering Modeling Language (EML).

This report describes the result of the investigation for latest computer technologies and software development techniques which seem to be usable for a realization of the analysis code system for nuclear engineering as an EML.

---

1 Reactor Physics Research Gr.、 System Engineering and Technology Div.、 OEC、JNC

2 Joyo Industry Co. Ltd.

3 Core and Fuel System Gr.、 System Engineering and Technology Div.、 OEC、JNC

4 Structure and Material Research Group、 Advanced Technology Division、 OEC、 JNC

5 Administrative Management Gr, System Engineering and Technology Div.、 OEC、JNC

6 Nuclear Energy System Inc.

目　次

表リスト

図リスト

第1章　緒言

21 世紀においても、ますますの科学技術の発展が期待されるが、その発展の形態については、時代の流れや多様化するニーズに柔軟かつ適切に対応することがこれまで以上に求められている。このような傾向は、原子力分野においても例外ではなく、その研究開発においては、時代の流れや多様化するニーズに柔軟かつ適切に対応できることが必要となっている。

一方、現在、解析コードを用いた数値シミュレーションは、理論、実験を補足するための、科学技術活動の一環であることが一般に認められるようになっているが、このことは原子力分野においても例外ではない。原子力開発においても、数値シミュレーション技術はますます重要度を増している。

研究開発に対するニーズが多様化し、解析対象がより複雑化している現在では、解析対象に応じて、工学的なモデルや解析手法を柔軟に変更したり、新たなモデルや手法を開発して従来のシステムを拡張したりできることが、解析コードに求められる重要な要素となってきている。従来、原子力関連の解析コードに求められる性能は、計算精度や計算速度が主なものであったが、これらの性能に加えて、柔軟性、再利用性、拡張性、保守性といった性能が強く求められる。

一般に、個々の解析コードは多くの高度な計算機能を有しているが、従来のように、ある解析コードに異なる解析コードの機能を単に組み込んで、新たな解析コードを作成するという方法では、組み込まれた解析コードは肥大化・複雑化する一方である。また、組込まれた解析コードは、オリジナルとは少し異なったコピーが増え続けることにもなる。このような解析コードの構築方法は、工学システムを設計する上で考慮すべき要求事項が増えるに従い、必要となる新たなプログラム改修や保守・管理等の労力を著しく増大させるものであり、解析コード利用の柔軟性という観点からはもはや開発投資に見合うだけの利用価値がないものとなる恐れがある。

本研究では、このような課題を受けて、主に原子力に関連する様々な専門分野で開発されてきた解析コードを、ニーズにあわせてより柔軟にかつ迅速に組み合わせて利用できるような新しいタイプの解析システム（次世代解析システム）の開発を目的とする。

ここで目標としている次世代解析システムは、基本的に原子力分野をターゲットとしているが、原子力に関連する技術分野は多岐にわたるため、多くの異なる分野の物理データや工学的モデル・手法を、いかにうまく結合して利用できるようにするかという点に大きな課題がある。

このような要求に的確に応えるために、本研究では、解析コードや解析システムに対して、新しい概念を導入する。既存の解析コード群を接続・統合して、大型の統合型解析システムを開発するのチではなく、「対象としている物理現象やデータを、いかにして人間とコンピュータの両者に分かるように記述するか」というアプローチを採用する。この概

念は、工学的な現象を記述・モデリングするための言語と捉えることができるため、本報告書では、この概念を工学系モデリング言語（Engineering Modeling Language）と呼ぶこととする。

工学系モデリング言語という概念は抽象的であるが、解析コードと、その開発に用いられてきたプログラミング言語の発展の歴史を大きく捉えて考えると理解しやすい。

図 1-1 に示すように、基本的なブール演算しか取り扱えない機械語の時代から、Fortran 等の高級言語が開発され、四則演算や初等関数、繰り返し計算、条件分岐等が容易に取り扱えるようになったことで、プログラム開発効率が劇的に向上し、多くの科学技術計算の分野で革新的な進歩が達成されたと考えることができる。

これまでに開発された解析コードの多くは Fortran で書かれているものが多いが、解析コードの開発に Fortran が好んで用いられた理由は、Fortran では、代数式を人間が使うのとほとんど同じ形で表現するだけで、コンピュータに計算をさせることが可能だからであると考えることができる。すなわち、Fortran は、代数式で表現された物理現象を記述するのに優れたプログラミング言語であったため、多くの分野における技術革新を支えてくることができたのであると考えることができる。

この段階では、微分方程式等の代数式で記述されたある特定の分野の物理現象を、いかにして四則演算や初等関数、繰り返し計算、条件分岐等に置き換えるかということが主な課題であった。この課題は今後も重要であり続けることは間違いないが、現在、新たに生じている課題は、各分野で開発された解析コードをいかにつなぎ合わせるかである。解析コード（ある特定の物理現象）の結合という概念は、代数式、すなわち、Fortran よりも、どちらかというと自然言語やダイアグラム、あるいは、他の数学的表現の方が適していると考えられる。

このように考えると、この課題に解決を与えて技術的なブレークスルーを起こすためには、解析コードをつなぎ合わせるという概念を、人間が理解するのと近い形で表現でき、かつ、コンピュータにも理解可能（実行可能）なプログラミング言語のようなものがあればよいと考えることができる。すなわち、この概念が、工学系モデリング言語である。

既存の要素技術や今後開発される技術、解析コードを、柔軟に再利用できる工学系モデリング言語としての新しい解析システムを開発することで、より高度な解析システムの開発を促すことにつながると考えられる。

例えば、原子炉システムの開発では、新型炉を軸とした核燃料サイクルに関する研究開発を、より統合的、かつ効率的に進めるための技術として、原子炉プラント内部で生じる現象を統合的にシミュレートできる解析コードが切望されている。プラント内で生じる個々の現象をシミュレートするコード群はほぼ整いつつあることから、これらを自由に結合して、プラント全体で起こる現象を工学系モデリング言語で記述することができれば、計算機上で「仮想プラント」が実現することになる。

更に、解析コードだけではなく、関連するプラントデータ、実験データ、解析データを

利用性の高い「情報データ」として蓄積・共有化すれば、計算機上の仮想プラントによる実験が大規模モックアップ実験を代替することが可能になると考えられる。このことにより、核燃料サイクルの研究開発にかかるコストを大幅に削減することが可能であると考えられる。

本報告書では、工学系モデリング言語としての次世代解析システムの開発におけるニーズと課題を検討し、課題の解決を図るために利用可能な要素技術を調査した結果をまとめる。第２章では、次世代解析システムに求められるニーズと課題を分野ごとに検討し、第３章で、課題を解決するための方策について検討する。第４章では、次世代解析システムの開発課題を解決するための要素技術の最新の動向について調査した結果をまとめる。第５章では、今回行った調査研究のまとめと今後の課題について述べる。

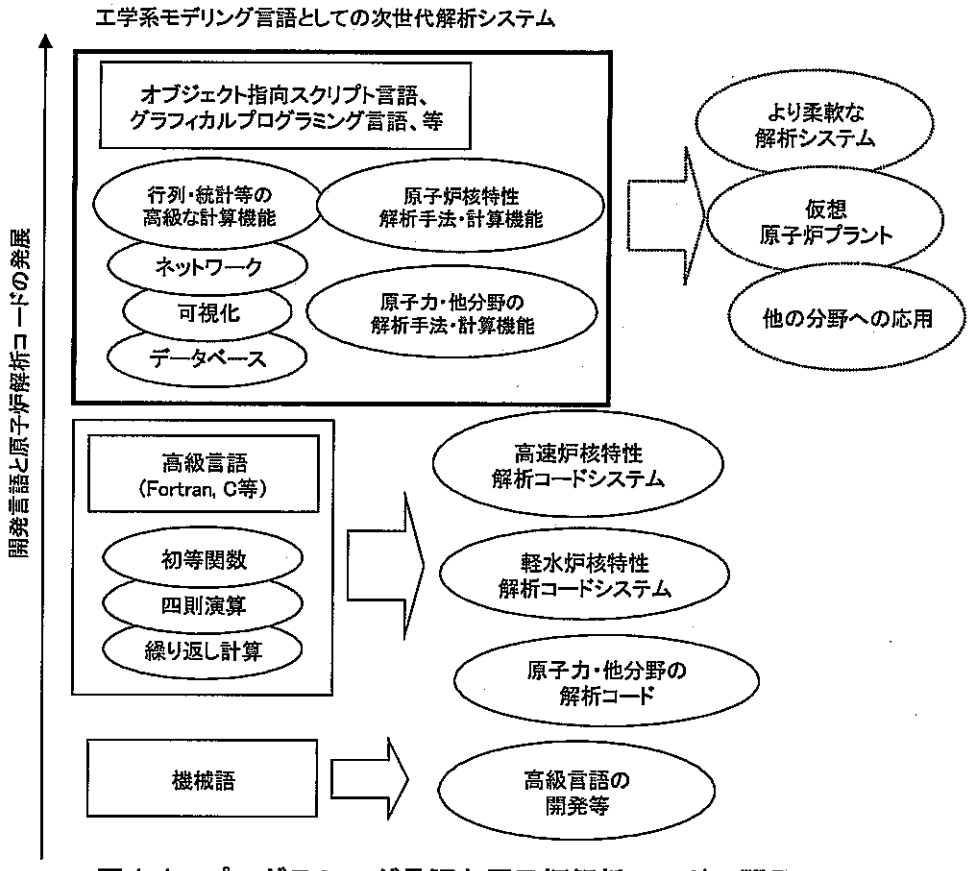


図 1-1　プログラミング言語と原子炉解析コードの開発

第２章　次世代解析システム開発のニーズと課題

本章では、次世代解析システム開発のニーズと課題について、熱過渡応力解析、炉心過渡解析、炉心核特性解析の各分野について、現状とニーズ・課題を整理し、各分野共通の課題を摘出する。

2.1　熱過渡応力解析

2.1.1　高速炉機器で支配的な熱荷重の緩和

高速炉で生じる系統熱過渡荷重はプラント機器にとって主要な荷重であることから、その緩和方法を提案している。系統熱過渡荷重はプラントの運用法とシステムパラータの組み合せによって決まる。個々のシステムパラメータは変動範囲を持っていることから、系統熱過渡荷重の大きさもそれに応じて変化する。プラント機器はこれらの中でも最も厳しい荷重に耐え得るように設計する必要がある。

従来の機器設計は、システムパラメータの変動を考慮して厳し目になる熱過渡条件を熱流動解析から設定し、このようにして得られた熱過渡条件の下で発生熱応力が許容値以下となる形状を構造解析によって求めるという手順を踏んでいた。

これに対して、本研究では熱流動－構造統合解析によりシステムパラメータと発生熱応力の関係を直接把握する方法を提案する。これによりシステムパラメータの組み合せから合理的に熱過渡条件を決定することができる。さらに、安全性や性能へ影響を及ぼさない範囲で、系統熱過渡荷重に感度が大きいシステムパラメータを調整することで、系統熱過渡荷重を緩和することができるようになる。

熱流動－構造統合解析を設計へ適用する上での課題は、膨大な解析ケースと計算時間の削減である。本研究では実験計画法と熱流動－構造統合解析コードを使用してこの問題の解決を図ろうとしている。これらの手法の有効性は、実証炉の中間熱交換器の系統熱過渡荷重解析への適用を通して示される。

2.1.2　高速炉の系統熱過渡荷重

原子力プラントに負荷される荷重には、圧力、自重および地震力に加えて、冷却材の温度変化によって生じる荷重がある。冷却材温度変化が構造材に伝わり内部に温度差が生じると、膨張と収縮する部材のせめぎ合いで熱応力が発生する。

高速炉では、軽水炉に比べて200℃程度運転温度が高いことから系統の温度変化幅が大きくなる上、冷却材が構造材へ熱を伝え易い液体金属ナトリウムであることから、系統温度変化が機器に与える熱応力（系統熱過渡荷重と呼ばれる）が厳しくなる。

系統熱荷重荷重には運転中に生じる事象の数だけ種類がある。図 2.1-1 に示す高速炉の系統図によりそれを説明しよう。高速炉は炉心で発生する熱を安全に蒸気タービンに伝えるために燃料棒の隙間に熱伝導の良い液体ナトリウム（１次ナトリウムと呼んでいる）を流すことで熱を取り出し、その熱を中間熱交換器で別の液体ナトリウム（２次ナトリウムと呼んでいる）に受け渡し２次ナトリウムの熱が蒸気発生器内を流れる水を加熱する構成となっている。ここで、起動または停止により炉心の発熱量が変化すると、１次と２次のナトリウムの温度が順次変化するためプラント内に温度差ができるため、それぞれの事象で系統熱過渡荷重が発生する。もし何らかの理由で原子炉を緊急停止したとすると、炉心の発熱は急激に低下するため、より厳しい系統熱過渡荷重が発生する。このように運転中には通常時から故障時まで様々の事象があり、もんじゅの場合は 22 種類、実証炉では 14 種類の事象が設定された。

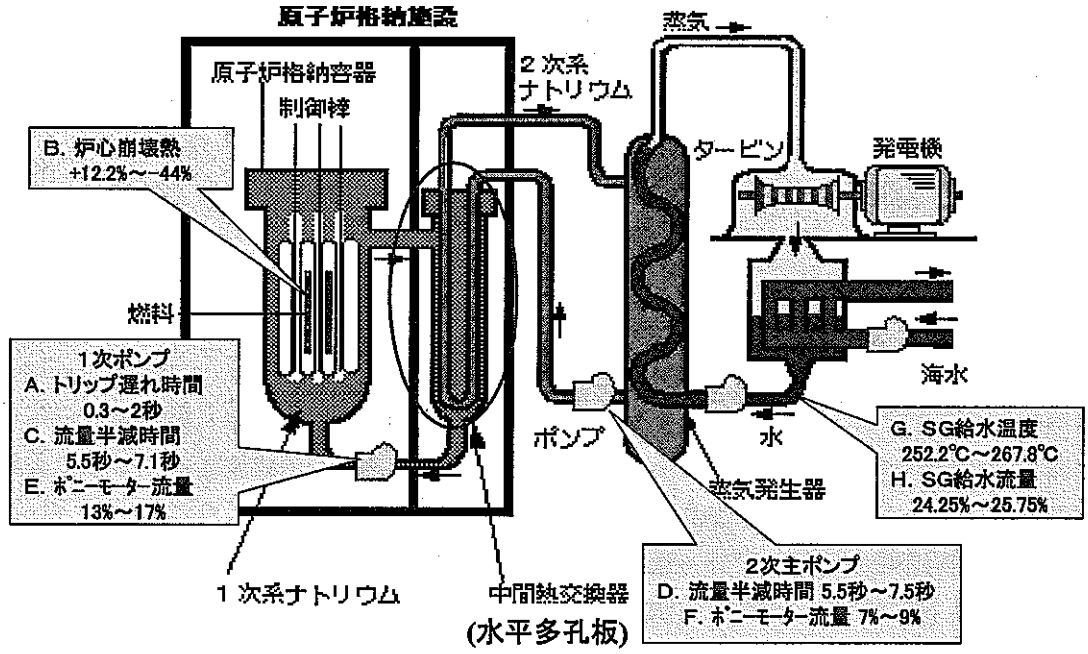


図 2.1-1　高速炉の系統構成とシステムパラメータの例

系統熱過渡荷重の発生機構は冷却材温度変化に対する構造の温度と応力の応答であることから、その大きさは、冷却材温度変化の要因と構造応答側の要因の両者の影響を受ける。

先ず冷却材温度の要因について説明しよう。冷却材の温度はプラント中の単位時間あたりの発熱量と除熱量のバランスで決まる。発熱と除熱に関与する因子を取り上げると、図 2.1-1 の例では A～H に示すシステムパラメータである。従って、これらは全て系統熱過渡荷重の大きさに影響を与える。これらのシステムパラメータは製作誤差や評価誤差により変動する可能性があることから、プラントの基本的な系統と運用を決めるシステム設計か

らはアウトプットとしてその変動範囲が示される。

次は構造応答側の要因であるが、こちらは温度の応答と応力の応答の２種類がある。温度応答は部材の熱容量と熱伝達係数に依存する。熱容量は主として板厚に依存し、熱伝達係数は冷却材の流速の関数である。その流速は図2.1-1中のA、C、D、E、Fの因子の影響を受ける。応力の応答は部材間に温度応答の違いによる温度差が生じ、さらに温度差に比例する熱膨張差が拘束を受けた結果生じるものである。従って応答特性は形状に依存する。

これらの結果、冷却材温度変化が同じであっても、応力の応答特性は構造毎に大きく異なることになる。図2.1-2にその例として、炉心出口の同じ冷却材温度変化に対する配管構造とYピース構造の応答の違いを示す。配管は板厚の内面と外面の応答の差に起因する板厚方向温度差が熱応力の要因となる。この場合、温度変化速度が大きい程、温度差がつきやすいことから、冷却材の温度変化速度に敏感な応答特性を有する。一方、Yピース構造では冷却材に液する容器壁と接しない支持部の温度差が応力の要因である。時間が経過すると容器壁の温度は冷却材の温度に追従することから、冷却材の温度変化巾に敏感な応答特性を有することになる。

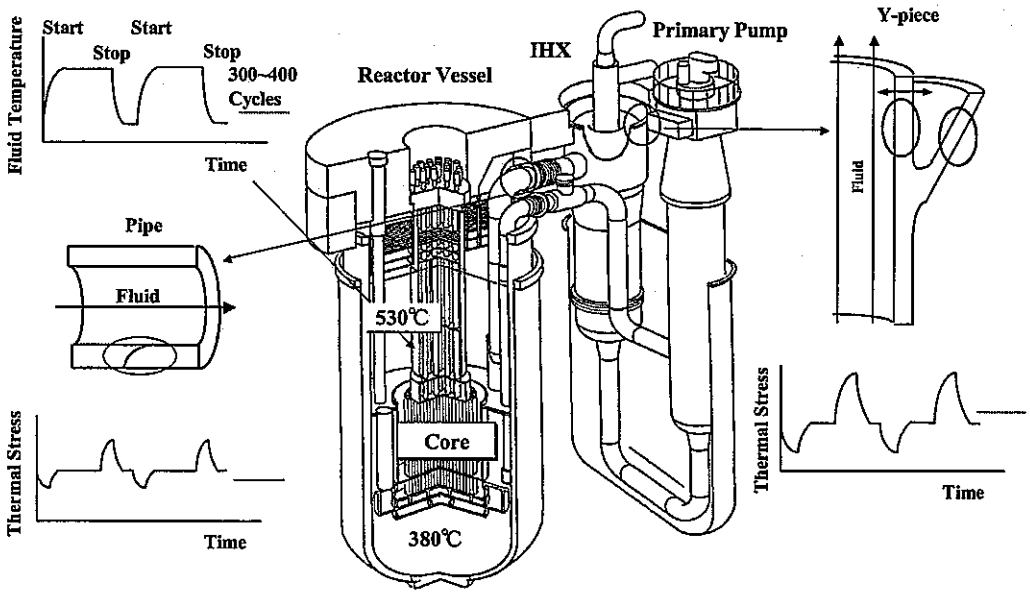


図 2.1-2　形状に依存する熱応力の応答特性

2.1.3　従来の熱流動・構造個別設計法

系統熱過渡荷重に対する機器構造設計の目的は、系統設計側から運転中に考慮するべき事象とシステムパラメータの変動範囲を受け取り、これに耐えうる構造形状を設計することである。従来の設計は、図 2.1-3 に示すようにシステムパラメータの変動を考慮して厳し目になる熱過渡条件を設定する熱流動設計と、設定した熱過渡条件のもとで成立する形状を求めるという、熱流動と構造の個別の手順がとられていた。

熱流動設計の目的である厳し目の熱過渡条件設定には２つの課題がある。一つは、冷却材の温度変化は発熱と除熱に係るシステムパラメータの複雑な相互関係で決まるため、システムパラメータの値を比較しただけでは熱過渡条件への影響度が予想できないことである。もう一つは同一温度に対し各機器毎の応力の応答が異なることから、プラント全体に対して安全側となる熱過渡条件を設定することは難しい点である。そこで、前者に対してはシステムパラメータの変動による流体温度の振れ幅を把握するためパラメトリックな熱流動解析が実施される。後者に関しては、熱流動解析の結果に対して安全係数を考慮した多直線処理を施し、実際より厳しい熱過渡条件を設定するようにしている。具体的には、熱流動解析の結果を時間領域でブレークポイントを設けて区分し、各領域の最急勾配で接続した多直線による温度変化図を作る。さらに各線分に温度変化幅を拡幅する係数と温度変化速度を速くする係数を掛け、接続することによって最終的な熱過渡条件をつくる。ここで、温度変化幅と勾配に対する係数は熱流動解析によって得られた流体温度の振れ幅に工学的判断を加えて設定する。

次の段階の構造設計では、冷却材温度変化を前提条件として、その条件の下での健全性の確認作業が行われる。つまり、形状を設定して与えられた熱過渡条件に対する構造解析を実施して熱応力を求める。応力が制限値を越えている場合は、それが許容値以下となるまで形状の変更と構造解析を繰り返す。ここでの課題は、発生応力が厳しい場合でも原則として熱過渡条件を決めている設計の上流には戻れないことである。

例として、実証炉の中間熱交換器の内部構造である水平多孔板の手動トリップに対する熱過渡荷重評価を取り上げると、冷却材の温度変化に影響するシステムパラメータとその変動範囲が表 2.1-1 のように与えられた場合の水平多孔板の手動トリップに対する熱応力を評価することが課題である。表中のノミナル条件(NOM)に対する熱流動解析の結果と多直線化によって得られた熱過渡条件を図 2.1-4 に示す。設定した熱過渡条件に対する中間熱交換器水平多孔板に生じる最大熱応力は 462MPa であった。

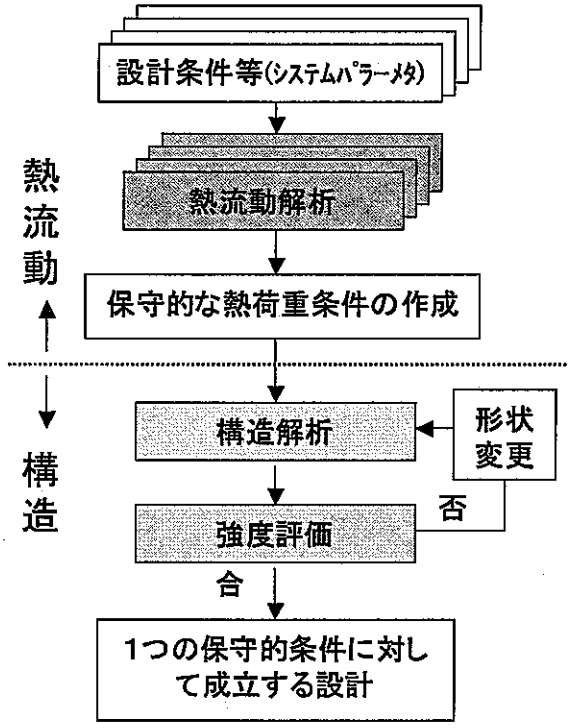


図 2.1-3　従来の熱流動・構造個別設計法

表 2.1-1　実証炉中間熱交換器の熱過渡条件に影響するシステムパラメータ

設計因子	パラメータ(水準)		
	MIN	NOM	MAX
A. ポンプトリップ遅れ時間		0.3秒	2秒
B. 炉心崩壊熱	−44%	NOM	+12%
C. 1次ポンプ流量半減時間	5.5秒	6.5秒	7.1秒
D. 2次ポンプ流量半減時間	5.5秒	6.5秒	7.5秒
E. 1次ポンプポニーモータ流量	13%	15%	17%
F. 2次ポンプポニーモータ流量	7%	8%	9%
G. トリップ後のSG給水流量	22%	25%	28%
H. トリップ後のSG給水温度	−3℃	トリップ時温度	+3℃

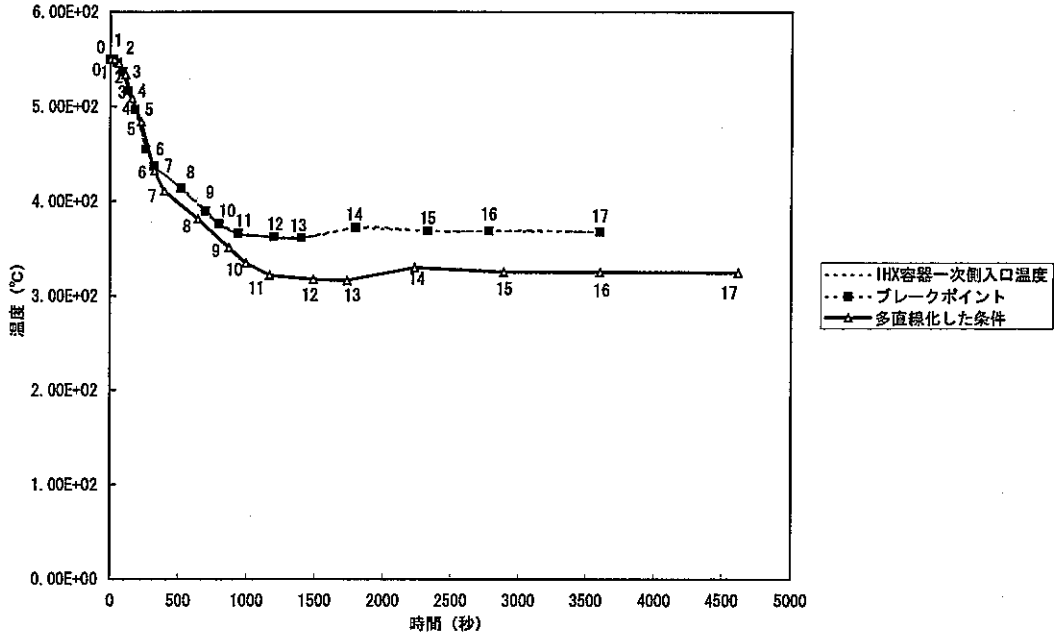


図 2.1-4　従来の熱流動・構造個別評価法で得られた実証炉中間熱交換器の熱過渡条件

2.1.4　熱流動－構造統合解析による熱荷重緩和設計法の提案

システムパラメータの組み合せに対する冷却材温度と熱応力の両者の応答を解析することによって、システムパラメータと発生熱応力の関係を直接把握する熱流動－構造統合設計法（図 2.1-5）を提案する(2.1-1)。この方法によると次章以降で述べる系統荷重の緩和設計が可能となる。

図 2.1-5 のフローを実現させる上での課題は、すべてのシステムパラメータの組み合せに対して熱流動解析と構造解析を実施するとなると、膨大な計算量が発生することである。

これを現実的な計算時間で実効するための方法として、実験計画法と熱流動－構造統合解析コードを組み合わせた方法を提案する。

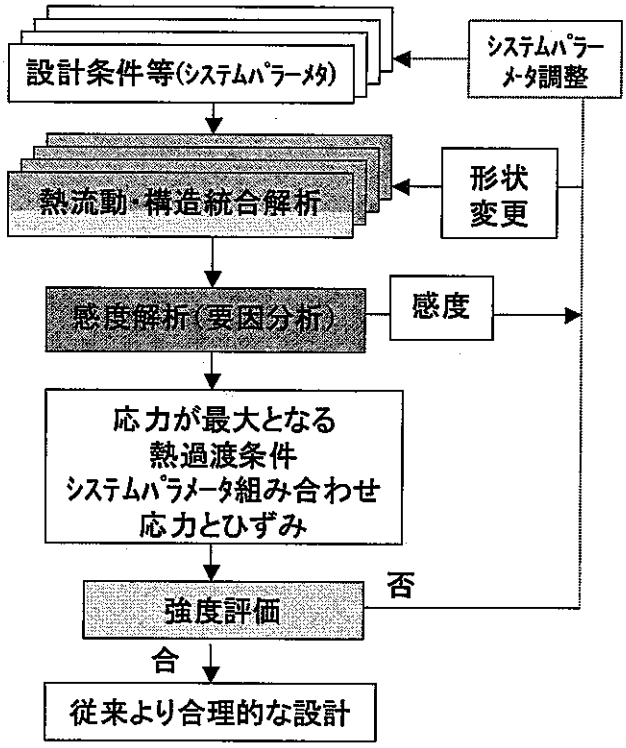


図 2.1-5　提案する熱流動・構造統合設計法

品質管理の著名な手法である田口メソッド(2.1-2)に採用されている直交表を用いた実験計画法を適用すると、システムパラメータの組み合せによって生じる膨大な解析ケースを大幅に削減した上で、パラメータに対する感度を得ることができる。実験計画法とは、条件割付の直交性を利用し、複数の因子が結果に与える影響を少ない試行回数で合理的に得る手法である。また、同時に要因効果分析の機能により因子が結果に与える影響度も定量評価できる。ここではその考え方を簡単に述べる。

最小構成の例として、結果に影響する因子がA、B、Cの３つの問題を考える。実験計画法ではそれぞれの因子の変動範囲を水準と呼ばれる離散値で表す必要があるため、ここでは水準が最小の２つの場合を考える。これらすべての因子の組み合せについて実験を行うとすると、$2^3=8$通りの実験が必要だが、表 2.1-2 に示す直交表Ｌ４を用いると、表中に示す４通りで済む。たとえば実験１はＡ１(Ａが水準１)、Ｂ１、Ｃ１という組み合せで実施する。ここで、２つの列を見ると、いずれも１と２という水準の組み合せであり、４通りの組み合せは(１、１)、(１、２)、(２、１)、(２、２)が同じ回数で現れている。この２列を「直交」していると呼ぶ。直交表ではすべての列が直交している。直交表を見ると、Ａが１の実験(上２段)では、Ｂは１と２が１回ずつ、Ａが２の時の実験(下２段)でもＢは１と２が１回ずつである。上２段と下２段を比べると、上２段も下２段もＢは１、２が１回ずつ出てきており、相違点はＡが１か２であるかという点だけである。従って、上２段と下２段の平均を比べると、Ｂの影響は等しく入っており、Ａの効果だけを評価できる。厳密な説明は文献(例えば 2.1-2)を参照いただきたい。

表 2.1-1 に示した実証炉中間熱交換器の熱過渡条件に影響するシステムパラメータは、因

子が８種類で水準は３または２であるため、これに対応するにはL18の直交表を利用する。表2.1-1のパラメータのすべての組み合せは4374通りとなるが、L18の直交表によってこれを18通りに削減できる。

表2.1-2　直交表　L４(２^3)の例

	A	B	C
実験１	水準１	水準１	水準１
実験２	水準１	水準２	水準２
実験３	水準２	水準１	水準２
実験４	水準２	水準２	水準１

熱流動解析－構造統合解析コード PARTS(Program for Arbitrary Real Time Simulation)を開発[2.1-3]することにより、種々のシステムパラメータの入力から熱応力の応答計算を短時間で実行できるようにした。

系統熱過渡は熱流動と構造の複合現象であることから、その解析には現象毎に以下の手順を踏んだ作業が必要である。

(1) 系統の動特性モデルの作成とシステムパラメータの入力
(2) 動特性解析による系統の冷却材温度温度変化の予測（必要に応じて多次元熱流動解析による機器内部の局所的温度変化の予測を実施）
(3) 構造温度応答解析モデルの作成と冷却材温度変化と流速から評価される熱伝達係数の入力
(4) 非定常熱伝導解析による構造温度応答の予測
(5) 構造熱応力解析モデルの作成と構造温度応答の入力
(6) 熱弾性解析による熱応力応答の予測

上記の解析を個別の計算コードを逐次起動して実行すると、主として以下の３つの理由で、１ケースあたり月オーダの時間がかる。

(a) コード間でのデータの受け渡しに必要な中間データの蓄積と様式変換に時間がかかる。例えば、動特性解析からはプラント全体の温度変化、流速等が時刻歴として得られるが、これを次の段階の構造温度応答解析モデルに受け渡すには、場所毎に全時刻歴データを記憶しておき、流速から熱伝達係数への変換等の、構造解析コードに適合する形式への変換が必要である。
(b) プラントの動特性モデルとシステムパラメータの設定には、ループ数や機器のレイアウトといった系統全体のレベルから、個々の機器の熱容量等の局所的なものまで、入力データの種類と数が多いため、システムパラメータの組み合せを変更しつつ解析を実施す

る場合に入力に時間がかかる。

(c) 熱応力の解析に通常の有限要素法を使用すると計算時間がかかる。尚、動特性解析は一般に1次元のフローネットワークコードが使用されるため、応力解析に比較すると時間がかからない。

これらの課題を解決するため、PARTS コードに以下の機能を実装した。

(A) コード間のデータの受け渡しを省くため、必要な熱流動と構造コードを統合した。内部では計算モデル間のデータ変換と転送は自動的に行われ、中間データは特に指定しない限り記憶されない。

(B) コードを統合すると、初期に必要な入力データの数が増え上記(b)の課題を助長することになる。そこでオブジェクト指向技術によって計算コードを大きく、流体計算グループ(2.1-4)と構造計算グループ(2.1-5)に分けた上で、それぞれのグループをさらに配管や熱交換器といった機器単位の自律計算部品で構成し、それらをビジュアルに扱えるようにした。このようにして実現したコードの入力画面を図 2.1-6 に示す。ユーザは予めカタログに用意された機器単位の標準部品を中央のシミュレーション台に載せマウスで連結することで解析モデルを構築できる。システムパラメータ等の局所的なデータは部品をクリックするとその場で入力できる。

(C) 熱応力計算時間は、計算法を有限要素法から Green 関数法(例えば 2.1-6)に変更することで短縮した。Green 関数法は図 2.1-7 下に示すように、単位ステップ入力に対する応答を予め用意しておき、時々刻々の入力データに対してこれを畳み込み積分することによって、任意波形に対する応答を得るものである。流体温度の単位ステップ入力に対して構造の温度応答と応力の応答を直接計算することができる。Green 関数法の課題は単位ステップ入力に対する応答を評価する方法であり、有限要素法によって計算することも可能であるが時間短縮に反する。そこでここでは、典型的な形状に対する応答特性をニューラルネットワーク (NN) (例えば 2.1-7)に記憶させ、類似の形状に対する応答を推論させるようにした(2.1-8)。

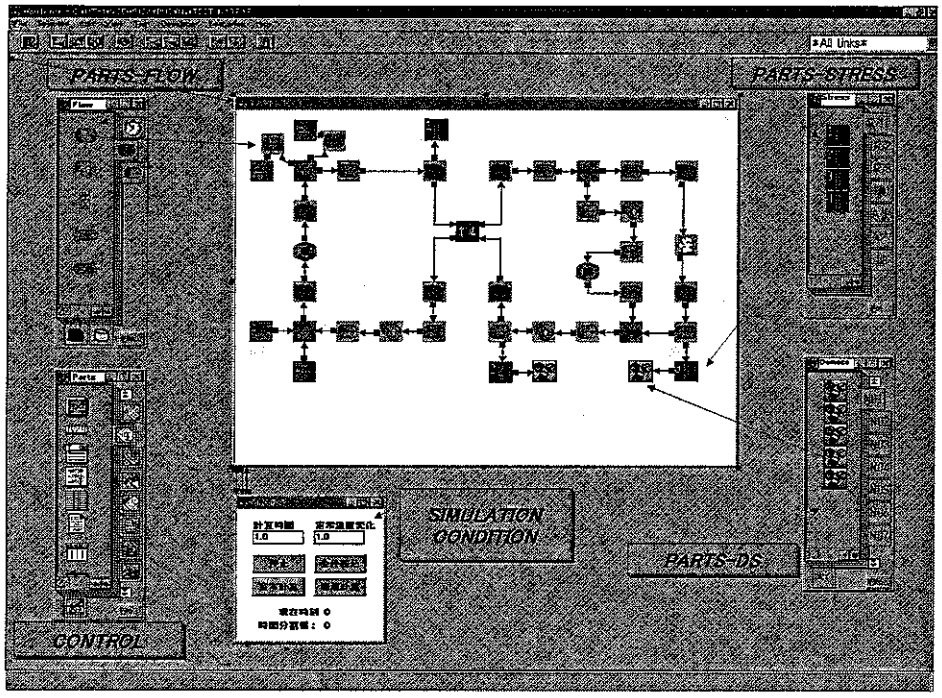


図 2.1-6　オブジェクト指向によってモデル化が簡単な PARTS コードの入力画面

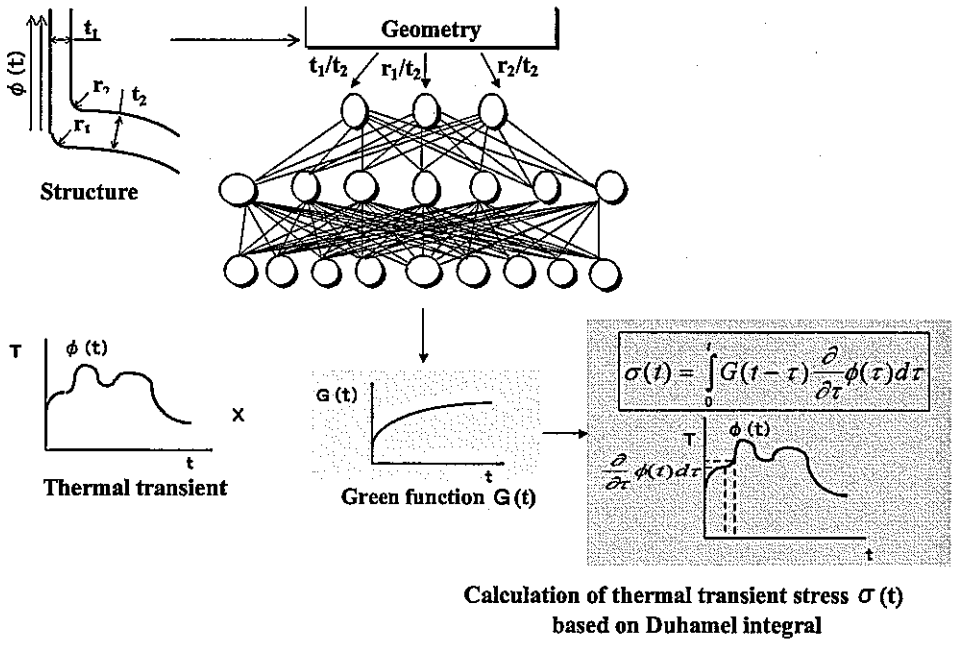


図 2.1-7　NN と Green 関数法による高速計算

実験計画法と熱流動－構造統合解析コードによる解析法を、図 2.1-1 で述べた実証炉の水平多孔板の手動トリップに対する熱過渡設計に適用した例(2.1-9)(2.1-10)(2.1-11)を示す。表 2.1-1 のパラメータを実験計画法による直交表 L 18 へ割り付けると表 2.1-3 のようになる。次に PARTS コードで実証炉モデルを構築し、表 2.1-3 に示された 18 ケースのシステムパラメー

タの組み合せに対する手動トリップ時の中間熱交換器水平多孔部の冷却材温度変化と熱応力の応答を計算した。PARTS コードの出力結果を図 2.1-8 および図 2.1-9 に示す。システムパラメータによって、温度と応力の応答に変化が生じることが分かる。

次に実験計画法の要因効果分析の機能により評価された熱応力の最大値と最小値を与えるシステムパラメータの組み合せとその時の発生応力を予測した結果を表 2.1-4 に示す。これらの組み合せは 18 ケースの解析には含まれておらず、応力は推定値である。これを確かめるため、実際に最大値と最小値を与えるシステムパラメータを PARTS コードに入力して計算を実行したところ、図 2.1-8 および図 2.1-9 に太線に示す応答と表 2.1-4 下段の応力値が得られた。PARTS コードの計算結果は実験計画法の推定値とほぼ近い値であったことから、実験計画法が妥当な応力推定を行うことが確認できた。

最後に、実験計画法の要因効果分析機能で評価した熱応力のシステムパラメータに対する感度を図 2.1-10 に示す。この結果から、最も感度の高いシステムパラメータは２次ポニーモータ流量であることが分かる。

表 2.1-3　システムパラメータの実験計画法による直交表 L18 への割り付け

ケース No.	A. ポンプトリップ遅れ時間	B. 崩壊熱	C. 1次主ポンプ流量半減時間	D. 2次主ポンプ流量半減時間	E. 1次ポンプポニーモータ流	F. 2次ポンプポニーモータ流	G. トリップ後のSG給水温度	H. トリップ後のSG給水流量
1	1 NOM	1 MIN	1 MIN	1 MIN	1 MIN	1 MIN	1 MIN	1 MIN
2	1 NOM	1 MIN	2 NOM	2 NOM	2 NOM	2 NOM	2 NOM	2 NOM
3	1 NOM	1 MIN	3 MAX	3 MAX	3 MAX	3 MAX	3 MAX	3 MAX
4	1 NOM	2 NOM	1 MIN	1 MIN	2 NOM	2 NOM	3 MAX	3 MAX
5	1 NOM	2 NOM	2 NOM	2 NOM	3 MAX	3 MAX	1 MIN	1 MIN
6	1 NOM	2 NOM	3 MAX	3 MAX	1 MIN	1 MIN	2 NOM	2 NOM
7	1 NOM	3 MAX	1 MIN	2 NOM	1 MIN	3 MAX	2 NOM	3 MAX
8	1 NOM	3 MAX	2 NOM	3 MAX	2 NOM	1 MIN	3 MAX	1 MIN
9	1 NOM	3 MAX	3 MAX	1 MIN	3 MAX	2 NOM	1 MIN	2 NOM
10	2 MAX	1 MIN	1 MIN	3 MAX	3 MAX	2 NOM	2 NOM	1 MIN
11	2 MAX	1 MIN	2 NOM	1 MIN	1 MIN	3 MAX	3 MAX	2 NOM
12	2 MAX	1 MIN	3 MAX	2 NOM	2 NOM	1 MIN	1 MIN	3 MAX
13	2 MAX	2 NOM	1 MIN	2 NOM	3 MAX	1 MIN	3 MAX	2 NOM
14	2 MAX	2 NOM	2 NOM	3 MAX	1 MIN	2 NOM	1 MIN	3 MAX
15	2 MAX	2 NOM	3 MAX	1 MIN	2 NOM	3 MAX	2 NOM	1 MIN
16	2 MAX	3 MAX	1 MIN	3 MAX	2 NOM	3 MAX	1 MIN	2 NOM
17	2 MAX	3 MAX	2 NOM	1 MIN	3 MAX	1 MIN	2 NOM	3 MAX
18	2 MAX	3 MAX	3 MAX	2 NOM	1 MIN	2 NOM	3 MAX	1 MIN

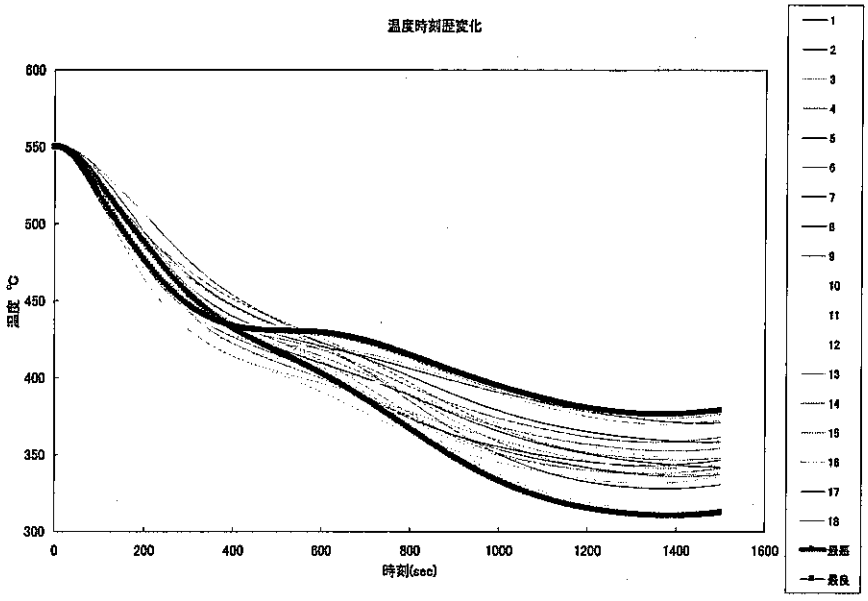


図 2.1-8　PARTS コードによって計算した冷却材温度変化

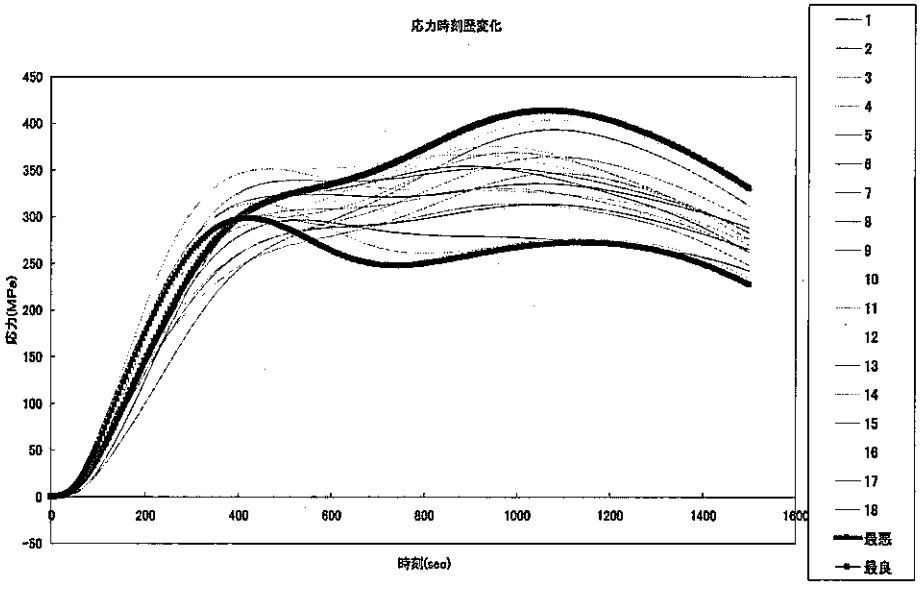


図 2.1-9　PARTS コードによって計算した熱応力変化

表 2.1-4　実験計画法で評価した熱応力の最大値と最小値

システムパラメータ	最小	最大
ポンプトリップ遅れ時間	MIN	NOM
崩壊熱	MAX	MIN
1次ポンプ流量半減時間	MAX	MIN
2次ポンプ流量半減時間	MIN	NOM
1次ポンプポニーモータ流量	MAX	MIN
2次ポンプポニーモータ流量	MIN	MAX
トリップ後のSG給水温度	MAX	NOM
トリップ後のSG給水流量	MIN	MAX
実験計画法による予想応力	277.8	420.9
パラメータ入力計算による確認	298.1	413.6
従来法によって評価した応力	-	462.0

（単位　MPa）

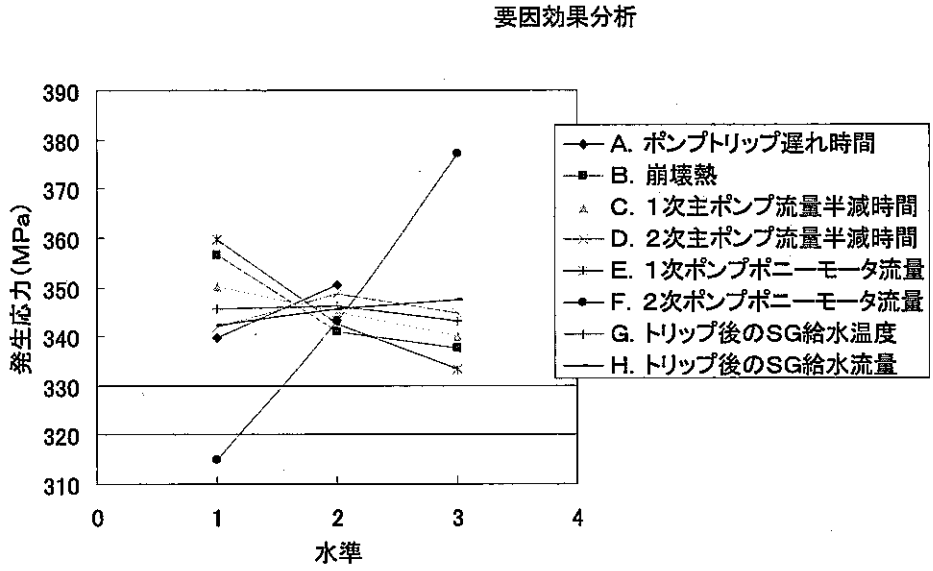


図 2.1-10　実験計画法で評価した熱応力のシステムパラメータに対する感度

システムパラメータと発生熱応力の関係が把握されると、発生熱応力が最大となる熱過渡条件をシステムパラメータの組み合せから客観的に決定することができる。このようにして求めた熱過渡条件の例である図 2.1-8 の流体温度変化を、同じ問題に対して従来の熱流動・構造個別評価法で得られた熱過渡条件である図 2.1-4 と比較すると、温度変化率が緩和されていることが分かる。さらに、発生応力を比較すると表 2.1-4 に示すように、従来法による 462.0MPa から提案法による応力は 420.9MPa に緩和されている。このことから、従

来法の工学的判断で決められた安全係数には過剰な裕度が含まれていたものと考えられる。

表 2.1-4 中の予想応力の最小値 277.8MPa は、システムパラメータを与えられた変動巾の中で組み合わせた場合に実現される最小の発生応力を意味している。従って、システムパラメータの誤差を小さくして応力最小となる値に近づけることによって、理想的にはこのレベルまでの荷重緩和が可能性である。

また、システムパラメータの中にはプラントの性能への影響よりも熱荷重に対する影響が大きいものが存在する。従って、安全性や性能へ影響を与えない範囲で系統熱過渡荷重に感度が大きいシステムパラメータを積極的に調整することで、系統熱過渡荷重を緩和することができる。

例えば、ポンプコーストダウン特性やポニーモータ流量は、プラントの性能を左右する定常運転には無関係であることから、安全性に影響のない範囲で調整可能である。実際に、Super-Phenix 等では、建設後にトリップ後のポンプ流量が変更されており、こうしたシステムパラメータの設定には自由度が存在することが示された。実証炉の水平多孔板の例では、発生熱応力に最も大きな影響を与えるのは２次ポンプポニーモータ流量である。ポニーモータ流量は原子炉停止時に補助的に使用するモータによる流量であり、発熱部と冷却部の熱の仲立ちに間接的に関与する。炉心の崩壊熱の除去性能に影響するため、安全性の観点からの許容範囲を定めた上で調整することによって応力を低減することが可能である。

以上のように実験計画法と熱流動－構造統合解析コードを組み合わせることによりシステムパラメータと発生熱応力の関係を直接把握する方法を提案した。本方法により、従来の熱流動・構造個別設計法による条件より緩和された系統熱過渡条件を、システムパラメータの組み合せから客観的に決定することができる。さらに、安全性や性能への影響の無い範囲で、系統熱過渡荷重に感度が大きいシステムパラメータを調整することで、系統熱過渡荷重を緩和することができる。

2.1.5 熱流動－構造統合解析コード PARTS の課題

高速炉の系統熱過渡荷重の緩和に有効な熱流動－構造統合解析を実現しようとすると以下の(A1)～（A3）の課題が生じる。

(A1) コード間でのデータの受け渡しに必要な中間データの蓄積と様式変換に時間がかかる。例えば、動特性解析からはプラント全体の温度変化、流速等が時刻歴として得られるが、これを次の段階の構造温度応答解析モデルに受け渡すには、場所毎に全時刻歴データを記憶しておき、流速から熱伝達係数への変換等の、構造解析コードに適合する形式への変換が必要である。

(A2) プラントの動特性モデルとシステムパラメータの設定には、ループ数や機器のレイアウトといった系統全体のレベルから、個々の機器の熱容量等の局所的なものまで、入力データの種類と数が多いため、システムパラメータの組み合せを変更しつつ解析を

実施する場合に入力に時間がかかる。

(A3)熱応力の解析に通常の有限要素法を使用すると計算時間がかかる。尚、動特性解析は一般に1次元のフローネットワークコードが使用されるため、応力解析に比較すると時間がかからない。

PARTS コードは上記の課題を解決するため、に以下の(B1)～（B3）機能を実装した。

(B1)コード間のデータの受け渡しを省くため、必要な熱流動と構造コードを統合した。内部では計算モデル間のデータ変換と転送は自動的に行われ、中間データは特に指定しない限り記憶されない。

(B2)コードを統合すると、初期に必要な入力データの数が増え上記(b)の課題を助長することになる。そこでオブジェクト指向技術によって計算コードを大きく、流体計算グループ(2.1-4)と構造計算グループ(2.1-5)に分けた上で、それぞれのグループをさらに配管や熱交換器といった機器単位の自律計算部品で構成し、それらをビジュアルに扱えるようにした。このようにして実現したコードの入力画面を図 2.1-6 に示す。ユーザは予めカタログに用意された機器単位の標準部品を中央のシミュレーション台に載せマウスで連結することで解析モデルを構築できる。システムパラメータ等の局所的なデータは部品をクリックするとその場で入力できる。

(B3)熱応力計算時間は、計算法を有限要素法から Green 関数法(例えば 2.1-6)に変更することで短縮した。Green 関数法は図 2.1-7 下に示すように、単位ステップ入力に対する応答を予め用意しておき、時々刻々の入力データに対してこれを畳み込み積分することによって、任意波形に対する応答を得るものである。流体温度の単位ステップ入力に対して構造の温度応答と応力の応答を直接計算することができる。Green 関数法の課題は単位ステップ入力に対する応答を評価する方法であり、有限要素法によって計算することも可能であるが時間短縮に反する。そこでここでは、典型的な形状に対する応答特性をニューラルネットワーク（NN）(例えば 2.1-7)に記憶させ、類似の形状に対する応答を推論させるようにした(2.1-8)。

上記(B1)～(B3)を実現する上でいくつかの制約を受けており、機能上の課題として以下に示す(C1)～(C3)が残されている。

(C1)熱流動と構造の解析機能に関しては、既存コードに存在するものであるが、PARTS コードの部品として再プログラムする必要があったことから、特に熱流動解析の機能を絞り込んでいる。具体的には、モジュール型プラント動特性コード Super-COPD(2.1-12) の Na 系の熱計算モジュールの機能のみを取り込んでおり、水蒸気系や流量に関する情

報は入力データとして与えるものとしている。その結果、これらが関与する主要な熱過渡事象である、給水管破断や主循環ポンプ軸固着（逆流が生じる。）を適切に模擬することが出来ない。

(C2)シンプルな GUI（グラフィカルユーザインターフェース）によって、計算部品間の結合を行うために、計算部品間で受け渡す情報を基本的にスカラー量に限定している。必要な部分に多次元解析を適用することができない。

(C3)応力解析については、Green 関数法の制約を受ける。すなわち流速が変化する等の温度以外の境界条件が変化すると計算誤差が大きくなる。部分的に FEM 解析を利用したいというニーズに応えられない。

さらに、上記(B1)～(B3)を実現する上で、プログラムに関する課題として以下に示す(D1)～(D3)が残されている。

(D1)コード間インターフェースは、Smalltalk 言語のメッセージ通信機能で実現しているため、固定化されている。言語に依存しない標準的で柔軟なインターフェースが望まれる。Super-COPD や FINAS 等の既存の計算コードを部品として利用するため、これらとのインターフェースをとるニーズもある。

(D2)グラフィカルユーザインターフェースは Digitalk 社の Visual-Smalltalk の機能を利用して実現したが、同言語のサポートが中止されたため、安定した言語で同機能を実現する必要がある。また、主としてユーザインターフェースの観点から選定した同言語に、計算モデルのプログラミングも制約を受けている。計算モデルのプログラミングはその他の部分から独立化した方が望ましい。FORTRAN 言語で書かれた Super-COPD 計算モジュールについては、できるだけそのままの型で取り込みたいというニーズもある。

(D3)計算精度向上のためには、部分的な２相流計算、多次元熱流動解析、および有限要素熱応力解析が必要であり、これらは計算資源を要求する。ネットワーク上での負荷バランス等の計算高速化の方策が必要となる。

2.2 炉心過渡解析

2.2.1 炉心過渡解析手順とデータフロー

核反応、冷却材熱流力挙動、構造物応答、燃料材料照射挙動が複合する原子炉内物理挙動に着目し、それらを高精度に模擬する数値モデルの開発、ならびにそれらを結合した実規模炉心システムを計算機空間に構築する手法の開発を進めている。

このような複合物理挙動は原子炉の安全性を評価する上で重要となる。一例として、図2.2-1は実際の高速炉の安全評価で評価の対象としている仮想事故事象のメカニズムを示したものである。例えば、蒸気タービンのトリップに伴い除熱源が喪失し、さらに原子炉停止信号が作動しないという仮想事故を考えた場合、まず2次系の冷却材温度が上昇する。続いて2次系冷却材と熱交換を行う1次系冷却材の炉心入口温度が上昇し始め、それに伴って炉心全域にわたり1次系冷却材の温度も徐々に上昇する。高速炉では、ラッパ管と称するステンレス鋼製の六角管の中に数百本の燃料ピンを束ね、それを数百体炉心に装荷することで炉心を構成するが、このラッパ管は炉心冷却材に接触しているため、1次系冷却材の温度上昇により熱変形を受けることになる。この熱変形と1次系冷却材の温度上昇により冷却材や構造材の密度が減少し、これが炉心の核的な反応度フィードバック効果となって現れ、原子炉出力の変化を引き起こす。原子炉出力が変化すると、原子炉出力と除熱に必要な冷却材流量との間のバランスが崩れるため、1次系冷却材温度が変化するとともに、燃料温度も変化する。前者は再びラッパ管の熱変形とそれに伴う冷却材や構造材の密度変化を、後者は燃料密度変化と燃料構成核種による中性子共鳴吸収量の変化（ドップラー効果）をそれぞれ引き起こす。これらはいずれも反応度フィードバック効果となって現れ、再び原子炉出力の変化を招く。このような現象が互いに連成しながら事象が推移していく。すなわち、安全評価ではこの互いに連成しながら推移するという事象進展の特徴を適切に考慮した上で各事象を解析し、それに基づいて物理現象を正しく理解するとともにその結果を適切に設計に反映することが重要となる。このような解析を行うには複数の専門分野をつなぎ合わせなければならないため、個々の専門分野で開発・整備されている解析コードをコード間のデータ授受を含めて適切に結合する方法を構築することが必要となる。

2.2.2 FANTASIの開発と摘出された課題

2.2.1で述べたようなニーズに応えるため、サイクル機構では図2.2-2に示すFANTASIと称する解析システムを開発した[2.2-1][2.2-2]。FANTASIは既存の大型解析コードを有効利用する観点から、これらが計算機上で互いに結合し動作するシステムである。核反応については、中性子束分布等の計算にCITATION[2.2-3]、反応度分布の計算にPERKY[2.2-4]を用いる。熱流動および過渡特性はSSC－L[2.2-5]、ラッパ管の熱変形はサイクル機構が開発した

PRECISE[2.2-6]をそれぞれ用いて計算する。

FANTASIはシステム構成の観点から主に2つの特長を有する。1つは、複数の解析コードをインターフェースプログラムにより結合し、全体をマスタープログラムで制御する方法を採用した点である。このため、各解析コードのモデル改良等を行う場合には、他の解析コードを改良する必要がほとんど生じず、システムの保守・管理がより容易となる。2つ目は、プログラム間のデータ通信を MPI[2.2-7]というメッセージ通信ライブラリを用いて行うこととした点である。これにより、あるプログラムの計算で得られたデータを一旦外部記憶装置に格納し、そのデータを必要とする別のプログラムが読み込むという従来の方法を抜本的に改善し、高効率のデータ送受信を実現している。実際、従来のデータ送受信方法では解析を終了するまでに解析対象によっては 1 ヶ月近くも要するとされていたが、FANTASIを用いれば同種の解析を数時間から1日程度で終了させることが可能であることが示されている。なお、FANTASIを構成する各々のプログラムは、ネットワーク結合された複数台のワークステーションに割り当てられる。

FANTASIの開発により、インターフェースプログラムと専用のメッセージ通信ライブラリを活用したシステム設計が複合する物理現象の連成解析を行うのに有効であることが示された。したがって、前述のように FANTASI を構成するある 1 つの解析コードを改修したとしても、その影響が及ぶ範囲はインターフェースプログラムまでに留められ、他の解析コードにはほとんど影響が生じない。ところが、新しい解析モデルを開発し、それに伴い解析コードを改修したり、全く別の解析コードを FANTASI に導入しようとする場合は、個々の解析コードを専門分野毎に更新しなければならない。現行の FANTASI を構成する各解析コードは、各々の専門分野で豊富な使用実績を有しているものの、そのほとんどが20～30年近くも前に開発されたものであり、バグ取り、改修、機能付加等の繰り返しによりコード自体が肥大化していることに加え、解析コードのステップ数が比較的多い（最小の PRECISE で約 8,000、最大の SSC-L で約 100,000）ことから、保守・管理に相応の手間を要しているのが実情である。一方で、実際の設計の現場においては、多様な型式の炉心やプラントを対象とした解析評価作業を日常とし、その評価方法も多様化してきており、このような幅広いニーズに柔軟に対応した解析を行うには、個々の解析コードを改修や保守・管理が現状よりも格段に容易となるような構成とすることが強く求められる。2.1.1 で述べたオブジェクト指向によるプログラミングはこの要求に応え得る良い例であり、FANTASI を構成する各解析コードをステップ数の小さな部品の集まりとして構成できれば、例えば原子炉過渡解析において冷却材の種類を変えた炉プラントについて解析したり、構造側の解析モデルをビームモデルからシェルモデルに変えたいといった要求に対しても、部品を必要なものに置き換えるだけで自由に解析条件を設定することが可能となる。これにより、ユーザーの解析作業にかかる負荷は大幅に低減され、解析コードの保守・管理も小さな部品を対象とすることから格段に容易となることが期待される。

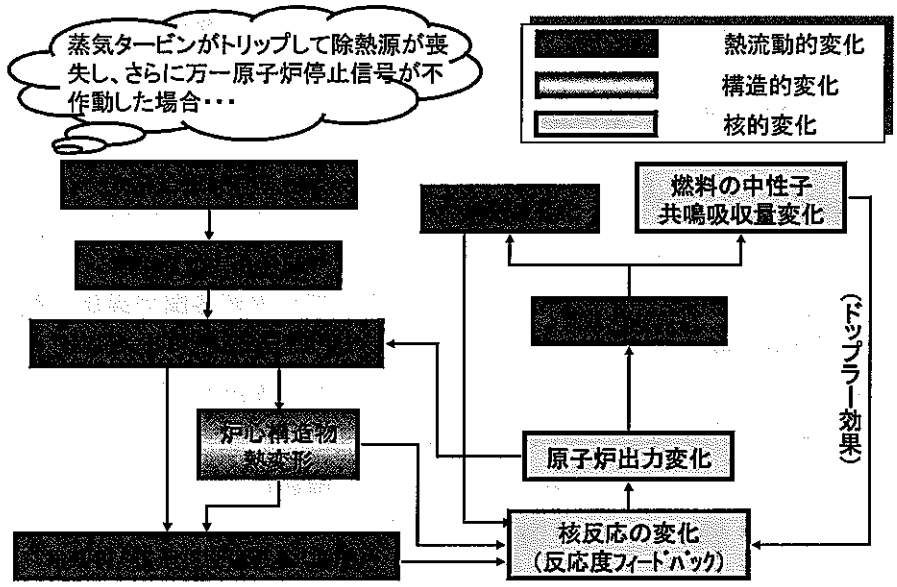


図2.2-1　　核-熱流動-構造連成事象メカニズムの例

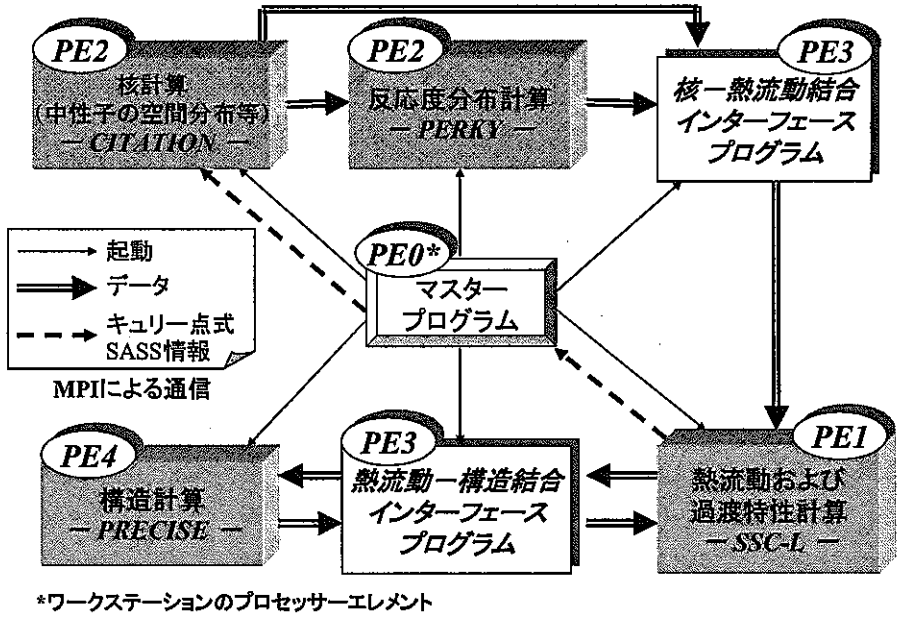


図2.2-2　　FANTASIのシステム構成

2.3 炉心核特性解析

2.3.1 高速炉炉心核特性解析スキームにおけるコード連携のニーズ

2.3.1.1 高速炉核特性解析スキームの概要

原子炉の核特性を計算で予測するためには、断面積データ、あるいは、核データと呼ばれる中性子と原子核の反応確率を表すデータが必要となる。これらのデータを核分裂炉の計算や遮蔽計算、核融合炉の計算等に利用するためのデータが各国で評価・整備されており、これらは評価済み核データライブラリーと呼ばれている。

評価済み核データライブラリーには数多くのエネルギー点のデータを含んでおり、このデータをそのまま原子炉の計算に用いることはできない。このため、断面積処理と呼ばれる操作を行い、これらのデータを中性子エネルギーで離散化して、群定数、あるいは、炉定数と呼ばれる形にまとめられる。原子炉で扱う中性子エネルギーは 0eV～20MeV の範囲に及ぶが、、このエネルギー範囲を 100 群～2000 群程度に分割する方法が用いられている。この炉定数は炉心体系に依存するため、対象とする炉心体系にあわせたパラメータを用いて作成されることになる。

高速炉の解析では、高速炉解析用に作成された炉定数がを用いて、解析が行われる。炉定数作成後の高速炉核特性解析の流れを図 2.3-1 に示す。

まず、燃料集合体を対象として、その領域における平均の群定数（実効断面積）を求める必要がある。通常、この計算は通常、格子計算と呼ばれる。ここでは、燃料集合体内の中性子束の空間分布及びエネルギー分布が計算される。

次に、この格子計算で求められた実効断面積を用いて、炉心全体の中性子束の空間分布及びエネルギー分布を計算する。この計算は格子計算に対して、通常、炉心計算と呼ばれる。図中では、格子計算と炉心計算の間に群縮約計算と呼ばれる過程が含まれているが、これは、炉心計算における計算量を低減するために、断面積をエネルギー領域で平均化してエネルギー群数を少なくするための計算である。炉心計算で得られる結果は、臨界性を表す中性子実効増倍率、中性子束、随伴中性子束等であるが、冷却材ボイド係数やドップラー反応度係数等を求めるためには、炉心計算で得られる中性子束等の計算結果を用いて、摂動計算と呼ばれる計算が行われる。また、燃焼に伴う燃料の組成変化を求めるためには、燃焼計算と呼ばれる計算が必要となる。

2.3.1.2 高速炉核特性解析スキーム構築における解析コード連携

以上のように、原子炉の核特性解析では、断面積処理、格子計算、群縮約計算、炉心計算、摂動計算、燃焼計算等の異なる概念の計算を連携して行うことにより、最終的な結果が得られることになる。また、各計算において、様々な解析モデルや解析手法が開発され

ており、解析対象の特徴に応じて使い分ける必要がある。このような解析モデルの違いによる影響（補正係数）を評価するために補正計算と呼ばれる計算も行われる。

このため、各段階の計算手法を様々に組み合わせて計算を行う必要が生じる。解析手法によって解析コードは別々になっていることが多く、解析コードを連携させることによって初めて、核特性解析が実施可能となる。このため、解析コードの連携は、核特性解析にとって基本的に必要不可欠な技術である。

2.3.2 炉心核特性解析の分野間におけるコード連携のニーズ

2.3.2.1 高速炉の臨界実験解析と実機解析との連携

サイクル機構では、大型高速炉の臨界実験であるJUPITER実験を主な解析対象として、核特性解析コードが開発されてきた。このため、臨界実験解析に対しては、解析コードの連携はよく整備されている。実機解析においても、格子計算、炉心計算、摂動計算で用いられる解析手法や解析モデルは、臨界実験解析と同じものを使うことができるが、データの受け渡しの使われるデータ形式が複雑化していること等から、現在のシステムをそのまま大幅に拡張して行くのは難しい状態となっている。これは、図2.3-2に示したように、各々の解析コードが解析機能やデータ形式を併せ持っているために、必要に応じて機能やデータを取り出すのが難しくなっているのが原因であると考えられる。

炉物理研究用の詳細解析システム、炉心設計コードシステム、炉心管理コードシステム等は、目的が異なるため、すべての要件を満たす解析システムを作るのは不可能のように思われるが、炉物理研究用の詳細解析システムでは、最も詳細な解析手法だけではなく、解析手法の差による効果を調べるために、簡易化された解析手法も使えるようになっていなければならない。このため、炉物理研究用の詳細解析システムを注意深く設計し、様々な手法が自由に組み合わせられるようにしておけば、その一部を取り出して、炉心設計コードシステムや、炉心管理コードシステムにそのまま再利用するといったことが可能であると考えられる。

欧州炉物理解析システムERANOSはこのような方式を採っており、システムの設計当初から、炉物理研究用の詳細解析システムとしてだけではなく、高速炉実機の炉心設計にも利用することが目的とされていたことが分かる。

2.3.2.2 高速炉と軽水炉等の他の分野との連携

前節では、高速炉の核特性解析にのみ言及したが、軽水炉や加速器駆動未臨界炉の核特性解析や遮蔽計算等とも連携できる可能性がある。例えば、サイクル機構の核特性解析システムでは、炉心計算にCITATIONと呼ばれるコードを用いているが、このCITATIONは、軽水炉の解析システムであるSRACでも用いられている。一方、長寿命核種の消滅処理を目的として、加速器駆動未臨界炉（ADS）の研究も盛んに行われているが、取り扱う

エネルギー領域や中性子源の有無等に異なる点があるが、中性子輸送の基礎方程式は同じであり、特に高速中性子の輸送を取り扱うので、高速炉核特性解析と重複する技術的な領域は多い。また、サイクル機構の核特性解析システムでは、Sn 法に基づく中性子輸送計算コードが用いられているが、同様にサイクル機構で整備されている遮蔽設計解析システムにおいても、Sn 法に基づく輸送計算コードが用いられている。両者で用いられているコードは異なるものの、基礎となる理論は共通しており、遮蔽計算に関しても共通化できる部分は多いと考えられる。

実際、フランス原子力庁が主体となって開発を進めている欧州炉物理解析システム ERANOS は、基本的に高速炉の炉物理解析システムであるが、遮蔽計算や ADS の計算にも応用できるように整備が進められている。高速炉や軽水炉、速器駆動未臨界炉の解析、遮蔽計算では、同じような技術が使われているにもかかわらず、解析コードが別々に開発されているため、お互いの成果を反映させるのに大きな労力が必要となるため、各分野の交流はあまり活発ではない。

ちなみに、軽水炉の炉物理解析の分野では、原研で開発された SRAC と呼ばれるコードシステムがよく使われているが、SRAC の開発者は、SRAC が統合型の大型コードシステムであり、拡張性が悪くコードの管理が大変であることを認めている。これを受けて、現在、原研では、軽水炉用炉物理解析システム SRAC を今後も拡張、保守していくことは難しいと判断し、MOSRA (Modular Code System for Reactor Analyses) と呼ばれるモジュラー型の炉心解析コードシステムの開発を進めている。MOSRA では、核計算だけでなく、冷却材計算（熱水力計算）、燃料温度計算、フィードバック断面積供給等の機能も含められる予定である。このように、軽水炉分野では、既に、モジュラー化を目指した解析システムの再構築が開始されている。

2.3.3 高速炉核特性解析におけるコード連携手法の現状と課題

高速炉核特性解析におけるコード連携手法としては、公開されているものでは、日本原子力研究所で開発された PDS ファイルと JOINT コードによる手法と、米国で開発された CCCC と呼ばれる手法が有名である。また、欧州では ERANOS と呼ばれる解析コードシステムが開発されている。ここでは、これらの特徴についてまとめる。

2.3.3.1 JOINT コードと PDS ファイル

サイクル機構の核特性解析システムでは、PDS ファイルと呼ばれるファイル形式と JOINT と呼ばれるファイル形式変換プログラムにより、解析コードの連携が実現されている。PDS ファイルは、原子数密度、実効断面積、中性子束等を取り扱うことができるが、バイナリファイルであり、もともとアセンブラ言語で開発されたという経緯もあり、現在の視点から見ると拡張性、移植性の点で問題がある。また、中性子束、原子数密度なども取り扱うことは可能であるが、摂動計算への接続部分で必要な中性子束データ等としては

適用することができず、各解析コード独自のフォーマット等が使われている。他にも同じデータであるにもかかわらず、様々なデータ形式が存在してしまっているのが現状である。

PDS、JOINTは、もともと原研で開発されたものであるが、現在、PDSやJOINTコードは各機関で独自の改良が加えられているため、各機関で異なっているのが現状である。

PDSとJOINTは現在の視点では、改良する点があると考えられるが、これらが目指した理想は非常に参考になると考える。以下に筆者が分析したJOINTシステムとPDSファイルの長所と短所についてまとめる。

<u>JOINTシステムとPDSファイルの長所と短所</u>

＜長所＞

(1) 断面積フォーマットをPDS形式で統一化したこと

(2) PDSのデータを表示・プロットするためツール等の機能を分離したこと

＜短所＞

(1) PDSの取り扱いにアセンブラを使ったこと

(2) 炉物理解析コード間で連携するデータは断面積データだけではないこと

(3) PDS形式は拡張したり、断面積以外のデータを表現するのが難しいこと

長所の(1)はデータを中心とした設計であり、現在のソフトウェア開発の主流であるオブジェクト指向の考えに通ずるものである。また、(2)は後述のMVCに共通する考え方である。

短所の(1)は移植性に問題が生じる。(2)は体系データや燃焼チェーン、反応率、反応度、随伴中性子束、解析スキーム等、炉物理解析上の他の概念は取り扱えないことを意味する。(3)は、PDSファイルの形式を拡張して、炉心体系のデータを保存するために使ったりすることは難しいため、せっかくPDSファイルを中心として設計されていたにも関わらず、PDSファイルでは取り扱えないデータ形式を受け渡す必要が生じてしまう。このため、独自のデータ形式を追加せざるを得ないという事態が発生し、独自のデータ形式が増えてシステムが複雑化してしまう。

この短所の(3)は注目すべき点であると考える。(3)が解決できれば、(2)は自然に解決でき、また、(1)は(3)の原因のひとつであるが、単にFortran95のような高級言語に変更するだけで解決できる問題ではないからである。

図2.3-3に、現在の高速炉核特性解析システムで利用されているデータ形式やプログラミング言語という観点から図式としてまとめた。本来は、PDSとJOINTにより計算の制御が行われるべきであったが、システムの拡張に伴い、それ以外の独自のデータ形式を使ってコード連携が行われるようになり、次第にシステムが複雑かしていったと考えられる。システムの変化すべてに対応可能にするのは不可能と思われるが、システムの変更に対して柔軟に対応できるデータ形式や計算制御方式が必要になると考えられる。

2.3.3.2 CCCC フォーマット

CCCC (the Committee on Computer Code Coordination) は、高速炉の炉物理コード開発の調整を目的として組織された。9期間が参加し、当時問題となっていた異なるコンピュータシステム間のコードの非互換性に起因する問題を軽減、解消するために、標準化された技法と手続きを採用することになった。作業は 1970 年から開始され、1973 年に VersionIII、1977 年に Version IV が発表されたが、米国での高速炉計画の停止によりが発表されたが、米国での高速炉計画の停止により CCCC システムの開発は中止されている。ただし、現在のコードでも CCCC の方針が活かされているものがある。

CCCC では、Fortran がベースとなっており、標準化インターフェイスファイルを定義し、この標準化インターフェイスファイルにより、単独コードの置き換え、コードシステムへの追加を容易にすることを目的とした。

前述のように、CCCC はその開発が停止されており、また、1970 年代に開発されたものであるので、現在の視点からは技術的に古い点もあり、そのまま採用することは現実的でないと考えられる。FORTRAN のみに限定している点やそれに関連した制限等は、あまり参考にならないかもしれないが、以下のような方針で設計されており、これらの考え方は今でも有効であり参考になると思われる。

(1) 入力・計算・出力の分離
(2) 標準化フォーマットによるデータの利用 (バイナリー順編成)
(3) 標準化サブルーチンによるデータ管理と転送
(4) メモリを最大限利用できるように配列、型は可変
(5) ワードサイズの互換性への配慮
(6) ドキュメント化 (internal-to-the-code documentation)
(7) エラー手続きの統一化

(1)は後述する MVC に通ずる考え方である。(2)、(3)は、オブジェクト指向のデータとメソッドに通ずる考え方であると思われる。(4)はメモリ管理、(5)は互換性、移植性、(7)は例外処理の考え方であり、いずれもオブジェクト指向で重要とされる項目である。また、(6)はドキュメントをコードの中に埋め込むことにより、ドキュメントとコードの不一致を極力避けるという発想である。後述するように、Java における JavaDoc、Python における pydoc、Ruby における RD-tool 等に近い考え方である。CCCC では、ソースファイルにコメントとして書き込むだけであり、最近のプログラミング言語がもつ機能のように、これらのコメント文を自動集積して、ドキュメント化するといったことまでは行っていないようである。

2.3.3.3 欧州炉物理解析コードシステム ERANOS

ERANOS は公開コードではないが、公開文献(2)にソフトウェア環境としての ERANOS についての記述があり、この内容をもとにして簡単にまとめる。

欧州での高速炉開発のための共同研究は 1970 年代から開始されたが、1984 年までは、各国が、解析手法と解析コードを各国で開発し、独自の解析スキームが構築されていた。しかしながら、このような重複開発は非効率的であるため、解析スキームは、各国で統一されたものを利用することで合意された。この結果開発された高速炉解析システムが ERANOS である。ERANOS では、臨界実験解析だけでなく、実機の核設計計算にも利用できるように、高い汎用性を持ったモジュラー化されたシステムを開発することが目標とされ、このために、その汎用性と携帯性に大きな注意が払われたとのことである。

ERANOS の開発では、解析コードシステムが長期間に亘って拡張可能であることを目標とされた。ここで「拡張可能」といっている意味は、既にある計算機能を拡張するだけでなく、全く新しいモジュールを追加することや、全く新しい計算スキーム（計算の流れ）を追加することで拡張できるということを目指したということであり、新しい機能といった場合、熱水力などの炉心設計の他の分野も視野に入れていたことが明記されている。

ERANOS の実装では、MASTER（Mighty Advanced Software Tools for ERANOS）と呼ばれるソフトウェアツールパッケージが用いられた。解析システムで用いられる基本データ（体系、断面積、中性子束等）は、SET（Self Descriptive Eranos Tree）と呼ばれるデータ形式で取り扱われる。SET は、枝分かれした階層構造を持っており、自己記述的なデータであるとされており、この SET の特徴のおかげで、新しい枝が使われていない既存のモジュールを変更することなく、SET に新しい枝を追加することが可能であると述べられている。

MASTER ソフトウェアツールパッケージについては、FLOWER、RULER、SABER、OPALE と呼ばれるツール群が含まれている。

(a) FLOWER

FLOWER は拡張版の FORTRAN 言語であり、前処理プログラムの形で通常の FORTRAN 言語を生成する。このことにより、通常の FORTRAN に比べて以下のような機能が追加される。

- ADA 言語の RECORD に似たデータ構造の宣言
- ポインタ型の変数
- メモリ管理を行う HAMMER ツール
- ユーザー言語 RULER を通したオペレータ間のデータ交換の命令

なお、HAMMER ツールは、データ構造の生成・削除をしたり、フォアグラウンドメモ

リ・バックグラウンドメモリ（コアメモリとハードディスクのことと思われる）への保存・復元を担っており、仮想メモリを作り出すことで最大限の性能を発揮することを可能にしているとのことである。

(b) RULER

RULER は、ユーザー言語と呼ばれており、SET を取り扱ったり、モジュール間の通信を行うために用いられる。RULER は do 文、if 文、print 文等を装備しており、高度なプログラミングを可能としている。

(c) SABER

SABER は SET を保存・復元するための機能である。この機能はアプリケーションとは独立しており、オブジェクト管理を可能としていると述べられている。また、この機能により、セッションを越えた研究を行うことを可能にしているとも述べている。

(d) OPALE

OPALE は、FORTRAN 言語と FLOWER 言語を理解することができる静的な文法解析器であり、ふたつの大きな目的を持っているとのことである。ひとつは、コンパイラに渡すときのコーディングエラーを検出することであり、もうひとつは、コードの構造やプログラムの内部に書かれた文書を自動的に処理してマニュアルの一部を作成することであると紹介されている。

ERANOS では、これらの MASTER に含まれるツール群により以下のようなことが可能となると述べられている。

①コーディングの標準の厳密な仕様：

- FLOWER 言語に基づく容易さと簡明さ
- Fotran77 の厳密な適用
- FLOWER 言語による do ループとデータ管理のインデント化
- ラベルの追加（SET の枝の追加のことと思われる）
- すべてのサブルーチンにおける内部文書化（機能、引数、変更等）
- エラー管理やエラー診断のシステム
- ユーザー関数のライブラリー

②管理の容易さと文書の生成：

- コードの内部文書化（マニュアルがソフトウェアとともに存在する）
- モジュラー化されたユーザーマニュアル（計算機能と同様に構造化される）
- SET データに関する完全な記述
- 利用可能な計算機能の一覧

③テストのための厳格なルール：

- 検証期間（validation time）：すべてのモジュールに対して、hostile テスト、logical テスト、referencing テストがお粉和得る。すべての古典的な計算セットに対して、reduced sequencing テストが行われる。
- 適正試験期間（during qualification time）：プロジェクト計算に相当する real sequencing テストが行われる。

④管理の容易性と効率性：

- 認識されたエラーの修正
- システムの変更可能性に対するコードの順応性

これらの特徴により、開発者は「ユーザークラブ」と緊密な関係を保つことが可能となると主張している。また、様々な計算機に対して移植作業や検証作業を行わなくても、同時に多くの計算機で検証済みのバージョンとしてすぐに利用できることや、必要に応じて、前のバージョンへ復元することもできることが述べられている。

その他、詳細にはふれられていないが、ERANOS の開発では、CASE（Computer Aide Software Engineering）ツールや、CAPRA（Changes And Problems Recording Aid）と呼ばれるバグ追跡システム等が使われていることも紹介されている

2.3.4 日本の原子力界における共用炉物理コードシステムの検討

最近、日本原子力学会及び日本原子力学会の専門委員会において、「共用炉物理コードシステムの構築」というキーワードの下、コード連携に関連した議論が行われている。この中で議論されている、コード連携のニーズ等について簡単にまとめる。

1999 年 3 月に行われた日本原子力学会春の年会において、炉物理部会企画セッション「共用炉物理コードシステムの構築に関するパネルディスカッション」が開催された。これを受けて、2000～2001 年度にかけて、日本原子力学会において、「共用炉物理コードシステム特別専門委員会」が設置された。この場において、軽水炉、高速炉を問わず、日本の炉物理関連の専門家が参加し議論が行われた。

以下に、この特別専門委員会の最終報告書の要旨の一部を引用する。

日本の各大学や公的な研究期間等で、開発された炉物理解析コードやデータは、個々に管理されており、利用目的や機能、データ形式の違いに等により、必ずしも、国内の利用者が共通して便利に利用できる状況になっていない。このため、利用者の目的に応じたコードへの入出力の改良やコード間のインターフェイスの作成に少なからぬ労力がかけられている。日本で公開される多くの炉物理コードに関して、共通のデータフォーマットやインターフェイスなどを整えることにより炉物理の研究環境は大幅に改善され、研究成果も

上がるであろう。また、最近の情報技術（IT）を利用すれば、システムの最新情報の発信、利用マニュアルの配付、システムの分散開発、共通検証データの閲覧などを効率的に行うことができよう。

以上のような主旨の下、大学、公的研究機関、産業界から、19 名の専門家が集まり議論が行われた。この委員会の最終的な結論としては、大学や公的研究機関、産業界が期待する共用炉物理コードシステムの具体像が必ずしも同じものではなく、以下のように二分されると結論された。

①大学、公的研究機関からの主なニーズ
自作のコードを自由に組み込め、他のコード群と組み合わせて汎用的に利用できる環境としてのシステム

②産業界からの主なニーズ
利用目的を明確にし、可能な限り最新かつ詳細な手法に基づき、自前コードの検証をすることができる信頼性が高い参照解提供用のコードシステム

この報告書では、ニーズがひとつに定まらなかったため、更に議論が必要であると結論づけているが、上記の二つのニーズをある程度包括した解析システムを構築することは可能であると考える。「自作のコードを自由に組み込」むことと、「可能な限り最新かつ詳細な手法」は、同じ意味であるし、「他のコード群と組み合わせて汎用的に利用できる環境」であれば、「利用目的」にあわせて組み合わせて使えば良いわけである。図 2.3-4 に示すような形で、各計算手法（コード）とデータの取り扱いや形式をうまく分離してモジュラー化しておき、必要に応じて解析スキームを組み立てることが可能で、かつ、新しいモジュールを組み入れるための機構が備わっていれば、両者のニーズを満足する解析システム像が浮かび上がる。

この特別専門委員会で結論づけられたニーズは、炉物理解析コードの「コード連携」の問題に他ならない。このように考えれば、この報告書が示しているニーズは、大学、公的研究機関、産業界（産官学）で一致しているわけであり、より柔軟なコード連携の技術は、非常に大きなニーズとなっていることが分かる。

2001 年度末に、この専門委員会は解散となったが、その後、この内容は、原研の炉物理専門委員会の「共用炉物理コードシステムの構築 WP」に引き継がれた。先の特別専門委員会に比べると少し規模が小さくなっているが、現在も議論が行われている。

日本原子力学会の特別専門委員会において、多くの専門家が集まって二年間議論されたにも関わらず、実際の作業に入れなかった理由はニーズ以外にあると考えられる。前述のように、産官学に共通したニーズがあることは明らかである。先の報告書の中を注意深く読むと分かるが、解析コードやデータ形式の標準化という作業は、非常に膨大な作業が必要で非常に難しい問題であり、かつ、そのような問題を解決しなければ、本来の研究を円

滑に進めることができないにもかかわらず、大学や公的研究機関では、成果として認められないという現実があるため、解析コードやデータ形式の管理者にはなりたくないという本音がうかがえる。このような仕事は、一企業では対応できない問題であり、本来、大学や公的研究機関が率先して行うべきであると考えられる。産業界に共通して、かなり強いニーズがあるにも関わらず、大学や公的研究機関がこのようなニーズに応えることが必ずしも成果として認められないという状況にも問題があると考えられる。

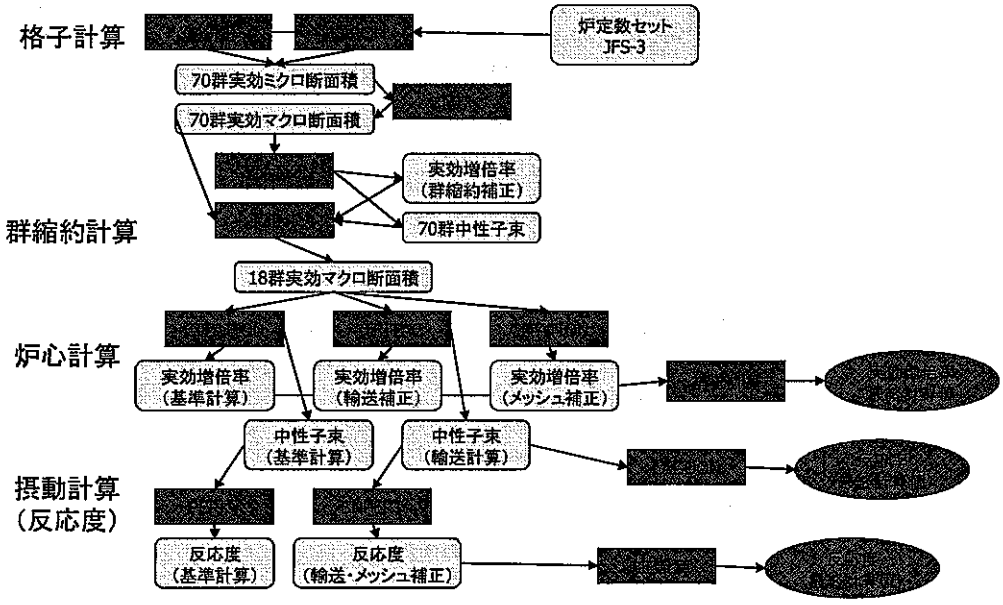


図 2.3-1　JNC 解析システムにおける解析スキーム

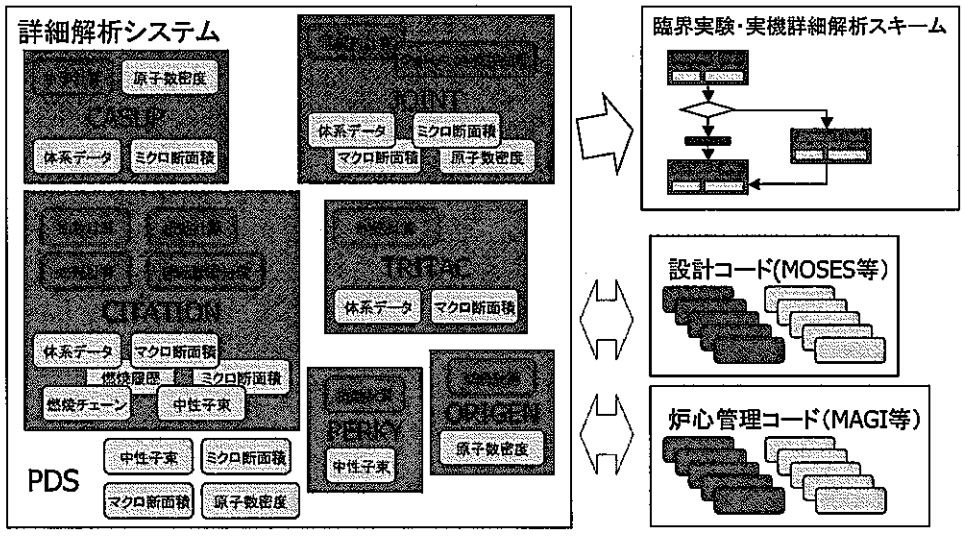


図 2.3-2　JNC 解析スキームにおける課題

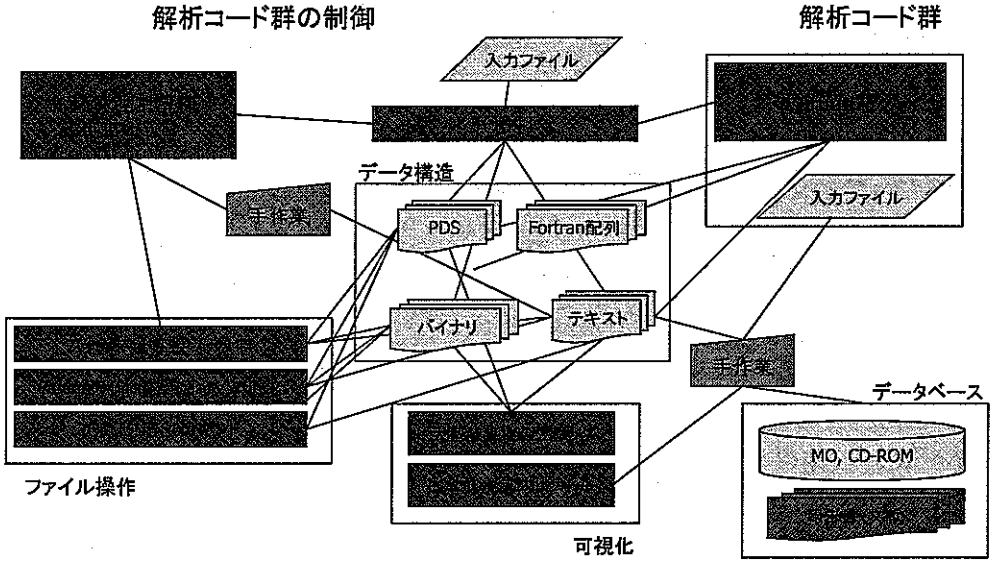


図 2.3-3　現在の JNC 核特性解析システムの構成

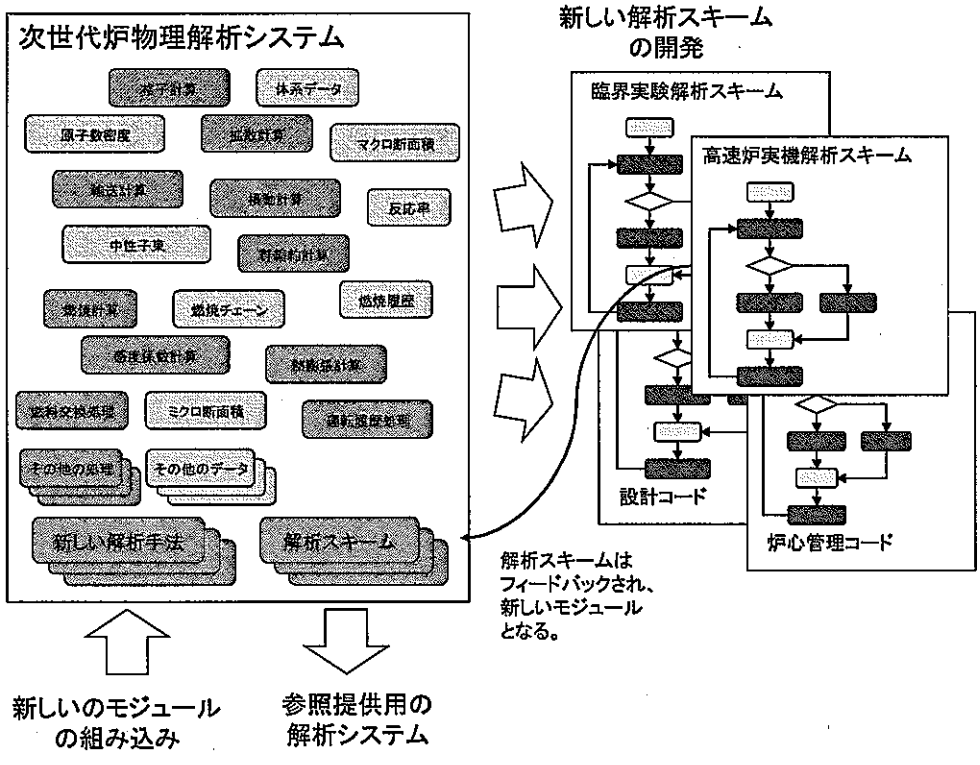


図 2.3-4　共用炉物理コードシステムに求められるニーズ

2.4 共通課題

本節では、前節までに述べた各分野におけるコード連携のニーズと課題を受け、各分野における共通課題をまとめる。ここでは、共通課題を、ニーズ（A)、機能上の課題（B)、実装上の課題（C）の３段階に分けて考える。

2.4.1 次世代解析システムに求められるニーズ（A)

まず、次世代解析システムに求められるニーズについてまとめる。各分野で挙げられたニーズは、主に以下の３つであると考えられる。

A1 自作コードの組み込み：
研究のため、自作のコードを自由に組み込んで使用したい。

A2 参照解としての利用：
設計へ適用するため、精度が保証されているものをリファレンスとして使用したい。

A3 異なる専門分野間での統合解析：
原子炉プラントの連成現象を解くため、核・熱流動・構造等の個別現象を統合したい。

なお、A3に関しては、再処理プラントの臨界事故の解析などでも同様のニーズもあり、原子炉プラントのみに関わらず、原子力関連のプラントにおける連成現象を解くために、各分野の個別現象を統合したいというニーズは今後も発生する可能性が高いと考えられる。

2.4.2 ニーズを満たすための機能上の課題（B)

上記のAで挙げられたニーズに対応するために必要となる機能上の課題について考えると以下のようなことが挙げられる。

B1 物理的意味を伴った中小規模モジュールの集合体としてのシステム：
上記のA1のニーズに応えるためには、プログラムを組替え可能な中小規模モジュールの集合体として実現する必要がある。この場合、モジュールの分割は開発者に左右されない、客観的（可能であれば物理的）な意味を伴ったものであることが望ましい。

B2 システムの信頼性確保：
解析の精度を示すには、少なくとも誤差の観点から性格の異なる、外部からの入力データ、プログラム組み込みの工学モデル、および数値解法を分けて取り扱う必要がある。さらに、複雑化・肥大化するプログラムの信頼性を保つ工夫が必要がある。また、信頼性を確保するためには、保守・管理を容易にする工夫も必要である。

B3 モジュラー型コードシステム：
連成現象に対する解析には、個別現象毎のコードを連携させる方法とすべての現象を取り込んだ統合コードを用いる方法とがある。上記のB1とB2の課題と両立し得るのは前者である。コード連携により連成現象を解くための課題は、コード間のインターフェースと制御法、および計算速度の向上が課題となる。また、連携法に関して、手法が結果に影響することから、入力データと同様に記録を残す必要がある。

2.4.3 機能を満たすための実装上の課題（C）

上記Bで挙げられた機能上の課題と解決するためには、以下のような実装上の課題を解決していく必要があると考えられる。

C1 異なるプログラミング言語間での結合：
上記B1のモジュール分割とB3のコード連携を実現するには、異なる言語で記述されたプログラムと多様な形式のデータを、柔軟に制御する技術が必要である。これらに活用できそうな技術として、Javaを核としたフレームワークであるJ2EE、Windows上のフレームワークである.NET、多様な言語と結合可能なスクリプト言語であるPythonやRuby等が存在する。

C2 自律したモジュール（オブジェクト指向技術の適用）：
上記B１のモジュールの分割をB2の精度保証やB3のコード連携を考慮した上で客観的で柔軟なものとするためには、目的に照らした分割法の試行錯誤と、組替えを可能とする自律したプログラム実装が必要である。これにはオブジェクト指向分析・設計・実装が有効と考えられる。この分野は急速にツールが整いつつあり、統一モデル化言語UML（Unified Modeling Language）と、UMLから多言語による半自動プログラム生成が可能になりつつある。

C3 ネットワーク対応・携帯性：
現在のハードウェア環境は、ネットワークで結合されたPCが標準となりつつある。解析システムを構成するモジュール群はネットワークで結ばれた複数の研究者やプログラマーによって開発・利用されることから、プラットフォームに依存しないことが望ましい。また大規模解析や上記B3の連成解析では計算負荷が大きくなることから、ネットワークコンピュータ上での高速化が必要である。高速化を主目的としたものにMPI等、オブジェクト化を主目的としたものにCORBA等があるが、両者を満たす既存ツールは無い。

C4 オープンシステムによる共同開発：

上記B1のようなモジュールの組替えを柔軟に行うには制御システムをユーザに開放する必要がある。また、B3の統合解析は異なる分野の専門家の協力が必要となる。これらをB2の信頼性を保ちつつ実現するには、システムのすべてに精通してなくとも確実な入力作業が行える支援機能を持つ優れたユーザインターフェースと、メインテナンスが容易な自己記述性に優れたプログラミングが必要となる。

C5 モジュールの結合関係の記述：

解析手法の物理的なモデルは、通常、数式などを使って表現されるが、計算制御や解析スキームといった物理概念は、ダイアグラムや自然言語で記述されることが多い。これは、コードの連携で表現しなければならない物理現象は、より高度に抽象化された概念であるからと考えられる。数式をプログラミング言語に変換するのは、比較的容易であるが、ダイアグラムや自然言語で表現しなければならない概念を、人間、コンピュータともに理解できる形で記述できるような、プログラミング言語やグラフィカルユーザーインターフェイスが必要となる。

第３章　次世代解析システム開発のための方策

異なる実行環境で動作するシステム同士を連携する方法として幾つか考えられるが、ここでは３種類の方法について述べ、また３種類の方法全てに対応したシステム構成を示す。それぞれの方法には長所/短所があり、求められる条件（機能・性能・操作性・拡張性等）と良く照らし合わせトレードオフを検討した上で連携方法を選択する必要がある。

3.1　制御言語と計算言語の分離

システムの複雑さを意識することなく、誰もが共通に思い描く物理イメージで計算コードを接続し同様に物理イメージでシステムからの応答を得られるオブジェクト指向制御言語を開発する。本言語への個々の計算コードの接続は、計算機の都合から決まる規約をすべて隠蔽し、物理イメージだけで記述された簡明なオブジェクト指向ソケットを使用する。これによって第 2.3 節で挙げた共通課題の（Ａ１）と（Ａ３）に関してユーザ側の課題が解決される。以下に具体的提案内容を示す。

3.1.1　オブジェクト指向設計の考え方である MVC モデルの導入

3.1.1.1　MVC モデル概要

MVC モデル（Model / View / Control）[参考文献]は、Smalltalk-80 言語から発生したもので３つの要素により構成され、それぞれは相互に依存せず独立・分離しており Model 部のみ別計算機で実行するという処理形態も可能とする非常に汎用性の高い概念である。

Smalltalk-80 では、ユーザインターフェース（U/I）を構築するために MVC モデルを用いたがその際３種類のオブジェクトにより U/I を構成した。Model オブジェクトはアプリケーションオブジェクト、View オブジェクトは画面の表現、Contol オブジェクトはユーザ入力に対するユーザインターフェースの働きを定義している。

MVC の概念が生まれる前は、ユーザインターフェース設計ではこれらを１つのオブジェクトにまとめる傾向があったが、MVC では、柔軟性と再利用性を向上させるためにこれらをお互いに独立させて扱っている。

この考え方は、広く応用されさまざまな場面で活用されている。例えば、Web サーバと DB サーバが連携して動作するインターネットシステムの殆どが、この考え方に基づいて構築されており、高い信頼性と可用性を実現している。

MVC モデルの特徴

・各要素の明確なる分離。
・分離する事で各要素は　環境（機種・OS）に依存せず実行出来る。
・汎用的で信頼性のあるインターフェース技術。
・要素間は、各種プロトコルに対応。
・パラメータも汎用的で拡張性に強い。
・シームレス（透過的）に繋がる。

3.1.1.2　PARTS-FLOW への MVC モデルの適用

複雑で大規模なシステムを構築する場合、計算処理部（Model）をさらに、細かな機能毎に上手く分割することが出来れば、機能の共有や柔軟な追加改訂が可能となる。ここで機能分割を誰もが共通に思い描く物理イメージに従って行い、モジュール化することが出来れば非常に使い易いものとなる。それにはシステム設計の初期段階で入念なオブジェクト指向設計を行う必要がある。

オブジェクト指向設計において「機能の明確なる分離」を行うための技術である MVC モデル（Model / View / Control）[参考文献]を導入することにより、システムを以下の 3 つの機能要素に分割し各要素の独立性・汎用性を高め、計算処理部（Model）を別計算機で実行させるという形態にも対応出来る事を目的とする。

① 計算処理部（Model）
② ユーザインターフェース部（View）
③ 制御部（Control）

MVC モデルを PARTS-FLOW に適用した場合の図を図 3.1.1-1 に示す。これら 3 つを独立したアプリケーションとして構築し、実績のあるインターフェース技術により組み合わせることで、以下に示すソフトウェア開発上の利点が生まれる。

ソフトウェア開発上の利点

- 機能毎に最も望ましい開発言語及びツールが活用出来、ユーザの操作性とそして実行性能が高まる。
- 機能の追加・修正を行う際、作業個所が限定でき他の部分に影響を与えず、保守性と拡張性が高まる。
- 機能により処理を別計算機に移し、負荷の分散が図れる。
- また、インターフェースプログラム（Wrapper と呼ばれる）を介して他アプリケーションを呼び出す事で古い言語で記述された既存のシステムを呼び出すことも可能になる。

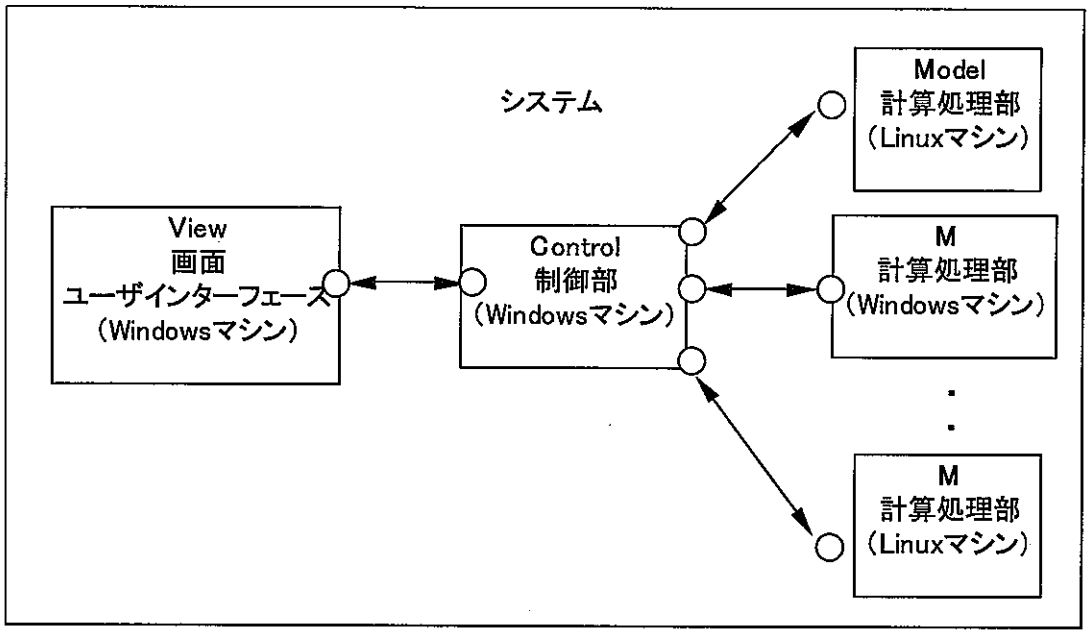


図 3.1.1-1　PARTS-FLOW の MVC モデル化（機能の分離）

※ 実行環境に依存しないシステム構成方式（機能毎に異なる計算機で動作する）

Model = 各機器部品（IHX：中間熱交換機等）
View = 画面インターフェース
Control = 制御部

3.1.2 機能のコンポーネント化

システムが明確な機能を持つモジュールに分割されていれば、各モジュールをコンポーネント（オブジェクト）化することで、更に以下の効果が発生する。

- 新たな計算処理を作成する場合も既存コンポーネントを組み合わせ、不足する部分のみを作り込む事で追加を容易に実現出来る。
- 既存プログラム（FORTRAN 関数）もコンポーネント化する事でオブジェクト指向メソッドと同じ容易さで扱える。
- コンポーネント単位で異なる環境で開発・実行することも可能であり、ネットワークコンピュータに適している。

ここで、コンポーネントとは自己完結した機能を持つソフトウェア部品のことである。

a. 1つ以上のエンタプライズ Bean とそれらに関連する 0 個以上のクラスまたはインタフェースを含むこと。
b. コンポーネントが外部に提供する API とその機能が定義されていること。

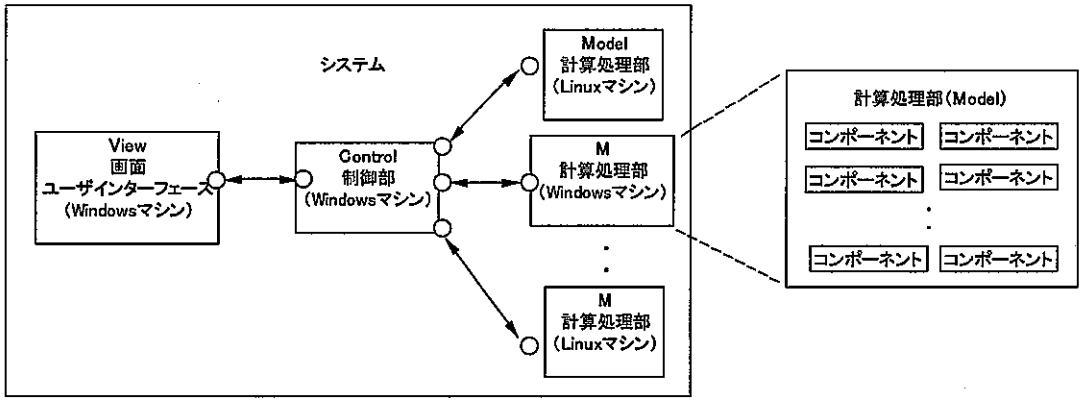


図 3.1.2-1 計算処理部の内部構成（コンポーネントの集合体）

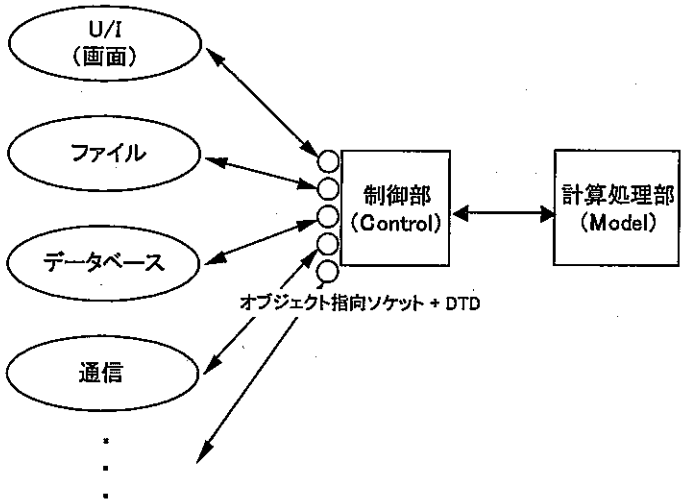
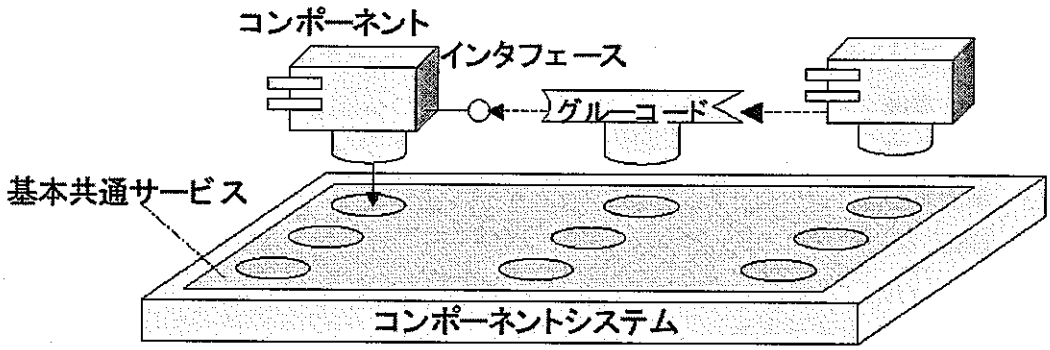


図 3.1.2-2 システムで扱うデータ種別



・コンポーネントシステム：コンポーネントを受入れる基盤環境、基本共通サービスの提供（制御・保存）
・コンポーネント：内部を隠蔽(Black-Box)、インタフェースを公開
・グルーコード：コンポーネント間を補完する“糊（のり）”

図 3.1.2-3　コンポーネントによるシステム構成[10]

再利用可能なソフトウェア部品である。もともと、ソフトウェアの再利用は、関数やサブルーチンの再利用、ソースコードの流用など、さまざまな形で行われてきた。しかしながら、例えばソースコードの流用を行った場合、もともとのプログラムにバグがあったり、仕様に変更があった場合、流用／コピーしたソースを個別に修正する必要がある。

これに対して、コンポーネントにおいてはコンポーネント内部のコードを隠ぺいし、インターフェイスのみから個別アプリケーションがアクセスできるので、コンポーネントと個別アプリケーションとの独立性は保ちながら高度な再利用を図ることができる。具体的なコンポーネントとしては、画面部品のような小さなコンポーネントから、ビジネス機能を部品化したビジネスコンポーネントのように大きなものまで存在する。

コンポーネントは再利用することを前提に作成された部品である。また、コンポーネント内に新たな機能の追加・修正を行う場合も、容易に実現される内部構成であることが望ましい。

コンポーネントの作者以外の人が追加・修正を行う場合、まず最初にコンポーネントに関するさまざまな情報（仕様・動作環境・制限事項等）が必要とする。

こうした情報が不足していると手探り状態の作業となり、コンポーネント情報を調べるだけでかなりの時間を必要とし、更に誤解も基に作業を行えば、誤った追加・修正を実施し要らぬ問題点（Software Bug）を生み出すことにもなる。

以上の事態を防ぎ効率的に作業を実現するためには、追加・修正作業が必要とするコンポーネントに関する情報を充実させ、コンポーネント実体とペアで公開することが重要である。

コンポーネントに関する公開情報としては、

- コンポーネントセット名
- バージョン
- 機能説明
- 動作環境
- 適用分野
- 制限事項
- UML 図

等の項目を含めるべきである。

3.1.3 システム言語とスクリプト言語

解析コードを連携させて利用するためには、計算の流れを制御するための制御用のプログラミング言語（制御言語）と、実際の計算を行わせるための計算用のプログラミング言語（計算言語）という概念が必要となる。この考え方に似た概念として、Ousterhout によるプログラミング言語の分類法(3.1.2-1)がある。本節では、この分類法についてまとめ、解析コードシステムとの関連を議論する。

3.1.3.1 システム言語とスクリプト言語の定義

Ousterhost の分類によると、プログラミング言語には大きく分けて二種類あり、スクリプト言語（Scripting Language）とシステム言語（System Programming Language）に分けられる。文献(1) の中では、スクリプト言語とシステム言語の特徴について、以下のように説明されている。

①スクリプト言語（Scripting Language）：

- 既存の強力な機能や要素を結合（Gluing：糊付け）して利用するための言語。
- データの型付けが緩いため、要素の組合せを簡素化することが可能であり、迅速なアプリケーション開発を促進する。
- 短いプログラムで高度な処理が可能であるが、実効効率は悪い。

②システム言語（System Programming Language）：

- メモリ等のコンピュータの基礎レベルから、データ構造やアルゴリズムを構築するための言語。
- データの型付けが強いため、複雑なデータ構造を管理するのを援助する。
- プログラムは長くなるが、実行効率は良い。

スクリプト言語は、短いプログラムで高度な処理が可能であることを特徴としており、プログラム一行あたりに処理可能な仕事量が大きいことが特徴である。その代わりとして、システム言語に比べて実行速度は遅くなる。もう一点、スクリプト言語の特徴として、データの型付けが弱いということが強調されている。

3.1.3.2 スクリプト言語とシステム言語の特徴

更に、スクリプト言語とシステム言語はそれぞれ、以下のような特徴を有しており、必要とされる条件に対して、それぞれ適正があると述べられている。

また、以下では、解析コードシステムに対する要件という観点からは、どちらの言語が適しているかについて考察した結果を、簡単にまとめた。

①スクリプト言語を選択すべき要件

- 既存の要素を組合せることが主な仕事か？　Yes（解析コード群の連携）
- 様々な異なった事象の操作が必要か？　Yes（対象は複雑で様々）
- グラフィカルユーザーインターフェイスを含むか？　Yes（結果表示、入力等）
- 機能は速い速度で進化するか？　Yes（解析手法は進歩する）
- 拡張性は必要か？　Yes（解析対象は変化する）

②システム言語を選択すべき要件

- 複雑なアルゴリズムやデータ構造が必要か？　Yes（ともに複雑）
- 大きなデータ処理や計算の速度は重要か？　Yes（ともに重要）
- 機能はよく定義されており、変化は遅いか？　No（解析手法は進化する）

このような観点から、解析コードシステムについて考えると、ほとんどが必要な条件であることが分かる。どちらかというと、従来の解析システムに求められる要件は、システム言語を選択すべき要件である、「複雑なアルゴリズムやデータ構造」、「大きなデータ処理や計算の速度」が重要視されてきたと考えられる。しかし、現在検討している次世代解析システムでは、解析コードの連携や新しいモジュールの組み込み等の変化への対応等が重要な課題となっており、スクリプト言語を選択すべき要件も強くなってきていると考えられる。しかしながら、計算速度等の要件は満たさなければならないので、両方の特徴を兼ね備えた開発環境が理想的であると考えられる。

3.1.3.3　スクリプト言語とシステム言語の分類

Ousterhost は、プログラム一行あたりの処理量とデータの型付けをパラメータにして、実際に存在するプログラミング言語の分類を行っており、図 3.1.2-1 にその結果を引用する。この図で、左上にくるものがスクリプト言語、右下にくるものがシステム言語に分類される。

ここでパラメータとして使われている、「データの型付け」という用語は抽象的で分かりにくい概念であるが、具体的には、整数型、実数型、文字列型といった計算機上で取り扱うデータの型の取り扱いのルールが緩いということである。文献では、例として、Unix シェルのパイプ機能を紹介している。Unix システムでは、標準入力、標準出力という概念により、各コマンドの入出力は、バイナリデータの一筋の流れ（a stream of bytes）として規格化されている。ここでやり取りされるデータは、テキストデータであってもバイナリデータであっても問題はなく、コマンド間でやり取りされるデータの型を先に宣言したりする必要もない。また、スクリプト言語に分類される Tcl 言語や Perl 言語においても、データの型宣言は不要であり、文脈に応じて言語がデータ型を解釈する。Ousterhost によれば、このようなデータの型付けが弱いという特徴が、各要素と要素をつなぎ合わせるためのグ

ルー（glue）言語としての特徴を支えているとのことである。

3.1.3.4 スクリプト言語とシステム言語の変遷と今後の展望

表 3.1.2-1 に、文献で言及されているプログラミング言語の変遷をまとめた。スクリプト言語は最初、Do ループのような基本的なプログラミング機能も持っていなかったが、その後、発展を遂げ、現在では、オブジェクト指向プログラミングのサポートも可能なもの（オブジェクト指向スクリプト言語）も多い。Ousterhost は、Tcl というスクリプト言語の開発者であるので、スクリプト言語に好意的である点は差し引いて考える必要があるが、現在のシステム開発で複雑さを増しているのは、結合という点においてであり、glue 言語としての特徴を持つスクリプト言語は、今後その価値を増すことであろうと結論づけている。

なお、文献では言及されていないが、Visual Basic も、最近になって、オブジェクト指向プログラミングのサポートができるように改良されている。また、同様に文献では言及されていないが、オブジェクト指向スクリプト言語の一種として、日本で開発された Ruby と呼ばれる言語も存在する。

また、Ousterhost は、すべての言語をシステム言語とスクリプト言語に分けることはできないとも述べており、その一例として、Lisp 系の言語を挙げている。Lisp 系の言語は、スクリプト言語とシステム言語の中間的位置づけであると述べている。Lisp 系の言語としては、Scheme、Haskel 等が存在する。また、Lisp 言語と同様に、関数型言語の特徴を持つものとして、ML 言語と呼ばれるものが存在する。ML 言語としては Standard ML や Objective Caml、等が存在する。

3.1.3.5 制御言語としてのオブジェクト指向スクリプト言語の利用

次世代解析システムにおける大きな課題は、解析コードという要素をいかにして連携させて、制御するかということである。スクリプト言語は、グルー言語と呼ばれ、コンピュータ上の要素をつなぎ合わせることを得意としており、次世代解析システムの制御言語として、スクリプト言語を採用できる可能性は高いと考えられる。

また、スクリプト言語とシステム言語という観点から、既存の解析コードシステムを分析すると、高速炉核特性解析システムでは、スクリプト言語として JCL と Unix shell、システム言語として Fortran が使われていると考えることが可能であり、スクリプト言語とシステム言語のハイブリッドシステムであると見ることが可能である。更に、欧州炉物理解析システムでは、RULER 言語がスクリプト言語、FLOWER 言語がシステム言語であると見ることができる。しかしながら、JCL や Unix shell の能力は限られている。ERANOS の開発において、RULER 言語や FLOWER 言語を独自開発したのは、このような背景があったのではないかと推測される。

従来の解析コードでは、制御言語（スクリプト言語）、計算言語（システム言語）といった概念は、あまりはっきりとは認識されていなかったと思われるが、現在、解析コードの

連携や制御という点で課題が挙げられているので、制御言語（スクリプト言語）に関して、オブジェクト指向スクリプト言語等の新しい技術の導入を検討するのは、自然な流れであると考えられる。

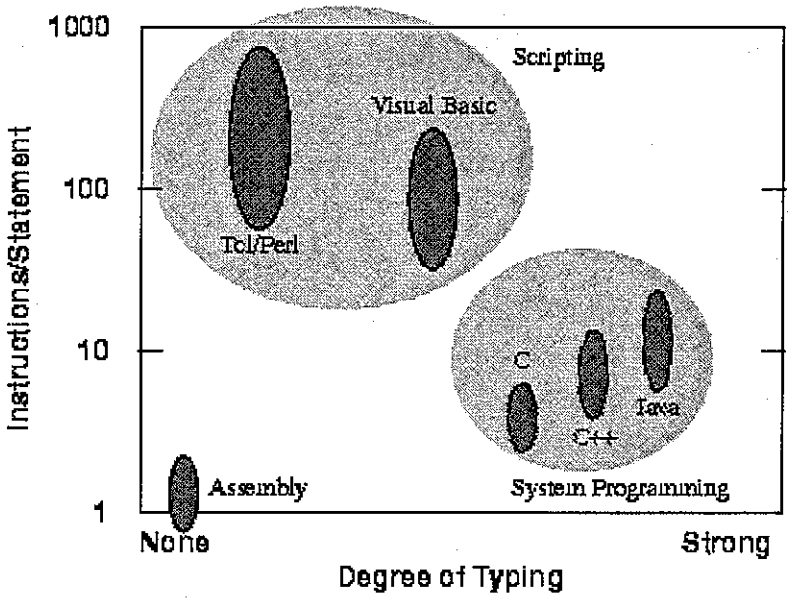


Figure 1. A comparison of various programming languages based on their level (higher level languages execute more machine instructions for each language statement) and their degree of typing. System programming languages like C tend to be strongly typed and medium level (5-10 instructions/statement). Scripting languages like Tcl tend to be weakly typed and very high level (100-1000 instructions/statement).

図 3.1.2-1　プログラミング言語の比較

表 3.1.2-1　Ousterhost によるプログラミング言語の分類と変遷

	スクリプト言語	システム言語	計算機システム
1970 年代	JCL (Do ループ機能なし)	PL/1、 Fortran、 assembly	OS/360
1980 年代	Unix shell (サブルーチン機能なし)	C	Unix
1990 年代	Visual Basic	C、 C++ (Active X)	PC (Windows)
現在 (1998 年論文発表時)	JavaScript、 Perl、 Tcl、 Rexx、 ...	Java	Internet World
将来	Python、 Perl Ver.5、 Object Rexx、 Incr Tcl、 ... (オブジェクト指向のサポート)	C、 C++、 Java、 ...	---

3.2 シンプルで汎用的なプログラム間インターフェース

3.2.1 データ交換によるプログラム間の連携

機能、さらにはコンポーネントとして分割され、ネットワーク上に分散配置されたプログラム同士をスムーズに連携させる方法を開発する。

様々な計算機あるいは OS 上で動作する既存システムを連携する方法は幾つか考えられるが、汎用性と拡張性の観点からは、プログラムを直接結び付ける方法に比べ、結果データを流用しあうデータ交換方式が最も現実的で効果がある（図 3.2.1-1 参照）。そのデータの形式・定義としては、以下が望まれる。

- テキスト形式の共通フォーマットデータ
- 構造化されたデータ定義（図 3.2.1-2）

典型例は「XML による異なるシステム間のデータ交換方式」でインターネットを介し、独自に開発され専用のデータフォーマットで運用しているシステム同士が XML 技術を介することで、拡張性の高いシステム連携を実現している。

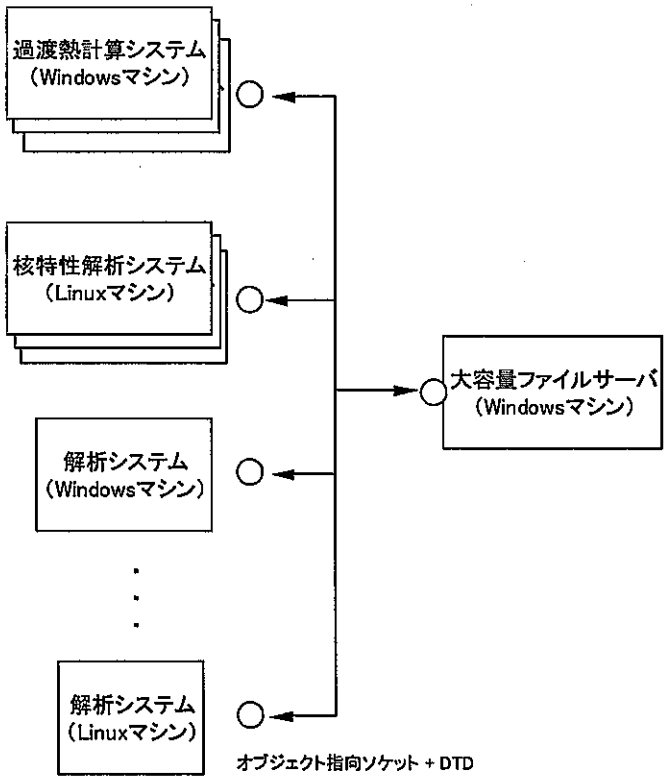


図 3.2.1-1　各種既存コードの連携

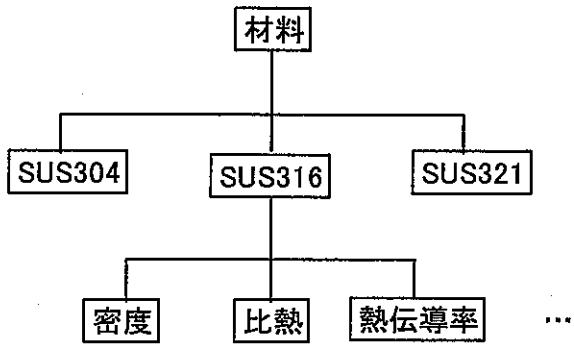


図 3.2.1-2　構造化されたデータ定義

この方式により拡張性・運用性の面で大きな効果が得られる。

- テキスト形式であれば計算機そして OS を問わずあらゆる実行環境のシステムに適応可能である。
- 構造化されたデータ定義であるため、データの意味も理解しやすく共通フォーマットの変更も容易に対応出来る。
- 更に「構造化されたデータ定義」を XML 準拠で行えば、インターネットを介したデータ交換にも即対応することが可能である。

3.2.2　プログラム間で交換するデータの抽象化

プログラム間で交換するデータは、インターフェースプログラムを介して、個々の計算環境に依存せず人間に理解し易い、抽象化されたものとする。

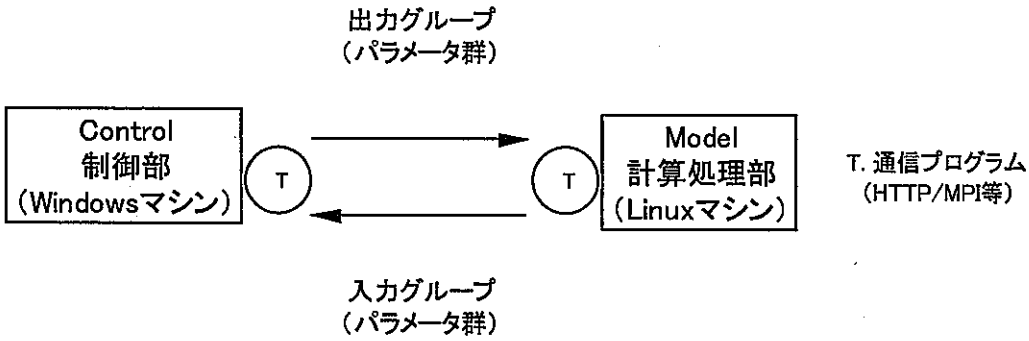


図 3.2.2-1　機能間のインターフェース

※ 個々のパラメータを抽象化した２つのグループとして扱う。

具体的には以下のポイントを実施する。

- インターフェースと処理部の分離
- 受け渡されるインターフェースパラメータ群をグループ化する
  機能間で受け渡されるパラメータ項目は個別に設定するのではなくグループ化してグループ名を受け渡す。
- 既存インターフェース技術の活用
  通信プロトコルとして既存の HTTP 又は MPI 等を活用する。

これらにより以下の効果が得られる。

- 実行環境に依存しないインターフェースを実現する。
- 実行時間の掛かる計算処理部の場合、別の計算機に処理を移すことも可能である。
- インターフェースと処理部を分離することで処理部の変更がインターフェースに影響を与えず汎用性が増す。
- 必要に応じて通信プログラムを入れ替えることも可能である。
- インターフェースパラメータをグループ化するため項目の追加 / 修正が発生しても最小の作業で済む。
- 既存インターフェース技術の活用により通信部分はブラックボックスとして扱うことが出来る。

3.2.3 汎用的なデータ I/O 形式

3.2.3.1 概要

ここでは汎用的なデータ I/O 形式の採用によるコード連携の方策について述べる。コードの連携では、データの受け渡しが重要となるため、データの形式が異なるとデータ形式の変換が必要となり、システムを不必要に複雑化してしまう。

取り扱う専門分野が異なっても、統一したデータ I/O 形式を採用できるためには、ある分野に特化した形式ではなく、一般的な事象に対して記述可能なより高度に抽象化されたデータ形式を採用する必要がある。また、システムの維持・開発をスムーズにするためには、以下のような特徴を有している必要があると考えられる。

- 拡張可能であること
- 自己記述的であること

データ形式としては、以下のようなものを検討している。

- XML

- HDF、NetCDF
- ADVENTURE 計画、GeoFEM 等における試み

複数のシステム（例：FORTRAN プログラム）を連携実行する場合、問題になるのがシステム間で受け渡されるデータをどのように扱うかということである。結果ファイルを受け渡す「疎結合」方式であればさほど問題にはならないが、関数同士が連携して実行する「密結合」方式の場合、受け渡すデータ項目の扱いに注意を払わなければ、正しい連携は実現出来ない。開発言語が異なれば、同じデータ型（例：浮動小数点/double）でも認識は違ってしまうのが現実である。

科学技術計算システムでは、膨大な数のデータ項目があり１つの項目自体のデータ容量もかなり大きいものになる。既存の FORTRAN プログラムにおいてモジュール間で共用する変数は通常、共通変数（Common 変数）として扱っている。この特徴は、膨大な数のデータ項目をパラメータとして受け渡すのに比べ、プログラムコーディング上の煩雑さや実行時のオーバヘッド（呼び出し負荷）が軽減出来ることである。

しかし、機能拡張等の修正を行った場合その影響はシステム全体に及んでしまい、拡張作業自体が大変困難なものになってしまうという欠点も併せ持っている。

こうした問題を克服し分散 / 並列環境にも対応出来るデータ I/O 方式を考察した。

まず、データ I/O 形式に求められる条件を以下に示す。

- 大容量データの取り扱いが可能であること。
- 分散 / 並列環境で使えるものであること。
- I / O 効率の高いものであること。
- 解析モジュールの連携に応用可能であること。

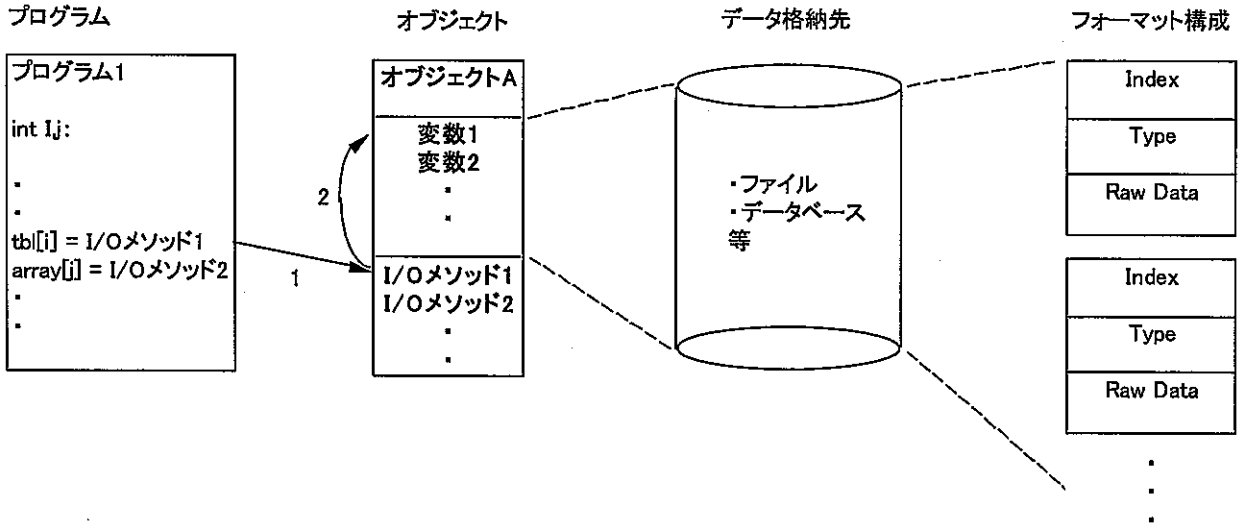


図 3.2.3-1 汎用的なデータ I/O 形式

3.2.3.2 汎用的データ I/O 形式を実現する方策

汎用的データ I/O 形式を実現するために策として以下の 2 点が考えられる。その全体概念を図 3.2.3-1 に示す。

(a) 汎用的な I/O フォーマット
(b) I/O ライブラリ

(a) 汎用的な I/O フォーマット

様々なデータ項目を格納出来るよう、シンプルなフォーマット構成を基本とする。但し、大容量データを効率良くアクセス出来るよう分類されたデータ（ブロック）容量は大き過ぎないように配分する。

具体的な構成を図 3.2.3-2 に示す。

フォーマット構成

① Index（索引情報）
　どういうデータが格納されているかの索引的情報を保持する。

② Type（属性情報）
　各データ項目の属性情報を保持する。この情報により個々のデータ項目の意味が理解出来る。

③ Raw Data（実際のデータ）
　実際のデータはこの領域に保持する。

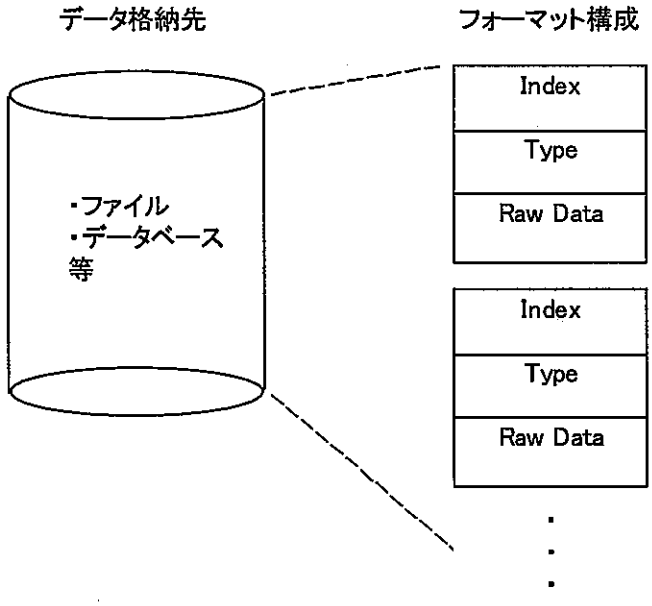


図 3.2.3-2 データのフォーマット構成

（注意点）

汎用的なフォーマット構成による媒体に膨大なデータ項目を格納する訳であるが、ここで注意しなければならないことが１つある。それはデータ項目自体の整理・分類をしっかり行うことである。同じデータ項目があちこちに存在していては、データを更新する度に項目全てを更新する必要が発生し非常に効率が悪く、アクセスするタイミングによっては正しい処理が行われない可能性があり、極めて不都合である。

そこで必要となるのは、データベース設計で行われるような「データの正規化」という作業である。これは、データ項目の重複を無くし、意味毎に構造化を行う訳であるが、最終的には全てのデータの中から１つのデータを一意に識別出来るまで整理・分類を推し進めることである。

こうしたデータ正規化の作業を経て、はじめて汎用的なI/Oフォーマットの効果が生かされることになる。

(b) I/0ライブラリ（メソッド）

プログラムにおいてデータの読み書きを行う場合は、直接ファイルやデータベースのデータにアクセスするのではなく、専用のI/Oライブラリ(メソッド)を介して行うことになる。

具体的には図3.2.3-3のイメージである。これは「データは全てオブジェクトとして扱う。」という考え方に基づいている。データには専用のI/Oメソッドが付随しており、1対1の関係になっている。直接変数にアクセスすることは出来ず、必ず専用のアクセスライブラリ（メソッド）によって参照することになる。一見回りくどく感じられるが、実際はプログラムの汎用性・拡張性を大きく高める効果を持っている。I/Oメソッドによってデータにアクセスする方式には、幾つかの利点がある。現状は、データの格納先がファイルであるがこれをデータベースに変更した場合でも、I/Oライブラリを呼び出すプログラム側の修正は一切必要無い。

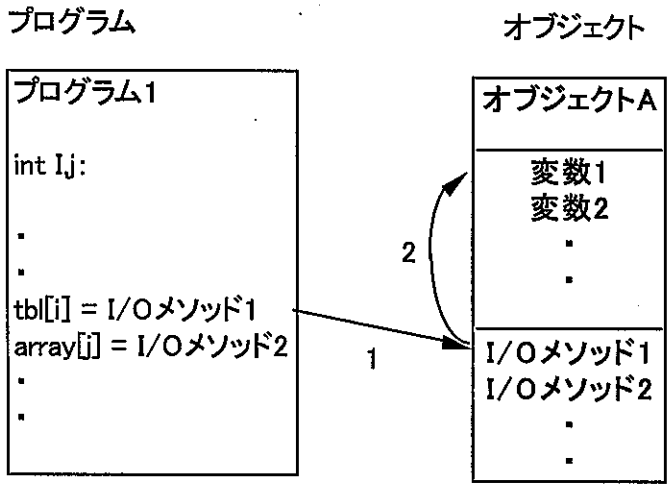


図3.2.3-3 I/0メソッド及びオブジェクト変数

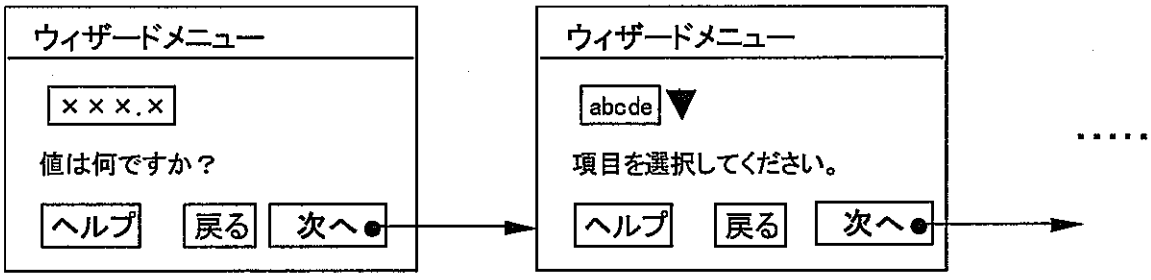
3.3 支援機能を持つ優れたユーザインターフェース

2.4 節で摘出された共通課題の C4 において、異なる分野間での共同開発を行うためにオープンシステムが必要であり、そのためには、支援機能を持つ優れたユーザインターフェースが必要であることを述べた。支援機能を持つユーザインターフェイスとして、自己誘導システムやグラフィカルユーザインターフェイスといったものが考えられる。なお、ここでいう自己誘導システムとは、ユーザが操作を間違えることなく正しくシステムを使えるようにシステム自身がユーザを誘導してくれるシステムのことを指す。

3.3.1 自己誘導システム

3.3.1.1 PARTS システムにおける自己誘導システム

システムの複雑さ煩雑さを出来る限り隠蔽し、容易に実行出来るユーザインターフェースを開発する。操作画面の至る所にヘルプを貼り付けこのオンラインヘルプによりユーザの理解を助け誤操作を防ぐ。また、実行に必要な一連の操作の流れはシステムが誘導する事で操作性の向上を図る。



※ システムが操作を誘導するためユーザは一連の流れを覚える必要が無い。

図 3.3.1-1 ウィザード方式による操作画面

具体的には以下のポイントを実施する。

- 必要最低限の操作
- ウィザード方式による操作画面
- 操作画面の至る所にヘルプを貼り付ける。
  報告書の電子化

これらにより以下の効果が得られる。

- 操作自体を最低限に減らす事でシステムの複雑さ煩雑さが隠蔽される。
- 充実したヘルプにより不明な点が解消され、システムの誤操作が防げる。
- 誘導型のメニュー画面を表示しユーザに入力を施すウィザード機能により一連の流れを覚える必要が無く操作が簡単になる。

3.3.1.2 核特性解析システムに求められる自己誘導システム

核特性解析システムでは、解析スキームを構築する際に、物理的にあり得ない計算オプションの組み合わせが発生することがある。現在のシステムでは、前述の PARTS のような GUI による自己誘導システムが整備されているわけではないが、明らかに計算オプションの組み合わせが間違っている場合には、警告メッセージが出力されるなどの工夫がなされている。このような工夫も、一種の自己誘導システムと考えることができる。

このように考えると、自己誘導システムは、特に GUI でなければならないという訳ではなく、テキストベースのインターフェイスに対しても、自己誘導システムを装備することは可能である。例えば、計算オプションの設定が明らかに間違いである場合などには、警告を出すだけではなく、計算の実行を止めて、適切なメッセージや参照すべきマニュアルを表示することで、ユーザを正しい方向に導くことが可能である。理想的には、ある計算オプションを指定した場合、他の計算オプションが一意に決まってしまうような場合には、システムが自動的に判断して他の計算オプションを指定するべきである。ユーザの操作は必要最低限で済むことになり、システムの複雑さや煩雑さを隠蔽することができる。もちろん、UNIX の man コマンドのように対話的に利用できるヘルプ機能等を整備することで、いつでもユーザが確認したい情報にアクセスできるようにしておくことも、有効であると考えられる。

異なるモジュール間でのエラーの組み合わせ数をすべて数え上げて、それぞれに対応することは現実的に不可能であるが、オブジェクト指向プログラミング言語の「例外」の機能を利用することで、複数の解析コード間で物理的にあり得ない組み合わせを指定したときに適切な例外を発生させるようにすることは可能であると考えられる。

3.3.2 ビジュアルインターフェース

3.3.2.1 概要

操作性は、システムの評価を決定付ける最も重要な要素である。煩雑な操作を必要とせず簡単でしかも直感的に操作できることが理想であるが、こうしたニーズを満たすのはなかなか容易ではない。シミュレーションシステムの場合、さまざまな条件での実行が必要となり、その度にソースプログラムを書き換えるのでは操作性が大きく損なわれる。

１つの解決策としてビジュアルプログラミングという手法がある。

ビジュアルプログラミング（ビジュアルインターフェース）とは、複雑な事象を単純化・抽象化したシンボルで表わすなどして視覚化し、これらを直接操作することによってプログラミングを行うことであり、近年急速に注目が集まっている。単にプログラムがグラフィクスを使うだけではなく、プログラミングやデバッグまでもビジュアルかつインタラクティブに行えるような「ビジュアルプログラミング・システム」も一部の開発言語用に製品化されている。

ビジュアルプログラミング・システムにおいては、従来のようなテキスト表示でなく、抽象化された視覚的表現によりプログラミングを行い、それをアニメーションさせることによって実行を表現する。これにより、テキストベースの言語に比べてプログラムの動作がわかりやすくなることが期待できる。

更に図的な表現を導入することによって、テキストのみで行なう現在のプログラミングより直観的能力を有効に活用することができるようになる。しかし、現状では、複雑なプログラムを記述するには、画面上が複雑になり過ぎるという問題が出てきている。

こうした問題に対しては、プログラムをシンボル（アイコン）化する際、従来の開発言語の命令と１体１で対応させるのではなく、関数単位でシンボル化することにより、制御の流れをよりシンプルにする事が可能となる。

ソースプログラムは予め部品（アイコン）化しておき、ユーザはこの部品をマウスで配置・結合することで全体プログラムが作成でき、しかも即実行することが可能である。条件あるいは部品構成を変えたい場合もマウス操作のみで対応可能である。

以下に示す SoftWIRE は、そうしたビジュアルプログラミングツールの１つである。

3.3.2.2 ビジュアルインターフェース製品「SoftWIRE」

ビジュアルプログラミング技術が製品化された例としては、Smalltalk 言語上で動作する製品が一部存在した。しかし、Smalltalk の言語仕様は非常に独特なものであり、習得にかなりの時間を要するという難点があった。

もっと身近で理解しやすい開発言語上で動作するビジュアルプログラミング製品はないかということで登場したのが、SoftWIRE for VisualBasic である。

最大の特徴は VisualBasic 上で動作することであり、VisualBasic であれば比較的容易な言語仕様であるため、習得期間を必要とせずシステムを構築する事が可能である。

操作方法は、到って簡単でシンプルなメニューから必要なファンクションを選び、オブジェクトを画面の好きな場所に配置し、ドラッグ・アンド・ドロップでさっとワイヤを描いてオブジェクトをつないでゆけば、あとはプログラムを実行するだけである。それだけでプログラムは完成する。

主な特徴を以下に示す。

・ マウスによるドラッグ・アンド・ドロップ式の強力なワイヤ機能を使って、部品同

士を接続出来る。

- ユーザインタフェースは非常にシンプルで、見ただけで直観的に理解できる。
- 予め SoftWIRE 内に豊富な部品集が用意されているため、全くソースプログラムを書かなくとも、実行可能なシステムを作成することが可能である。
- 拡張性が高い
- 既存部品を流用できる
- 既にある ActiveX コントロール等のプログラムも流用可能である。
- 独自仕様の部品が作成可能
- 大規模システムの構築も可能

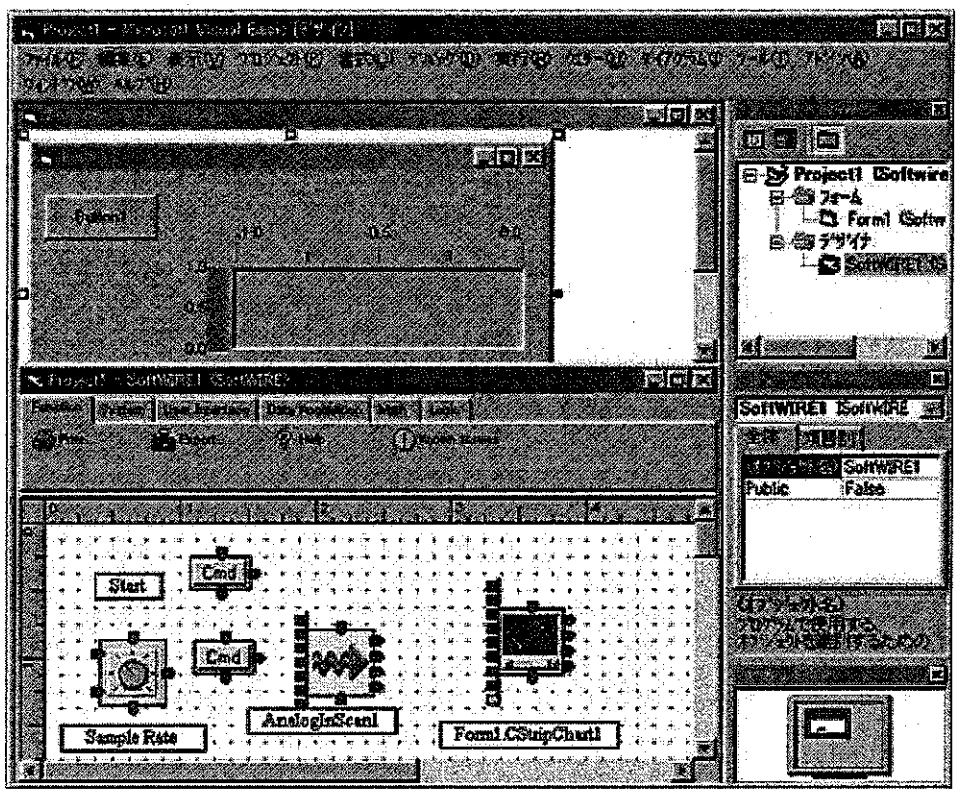


図 3.3.2-1　SoftWIRE による編集時画面

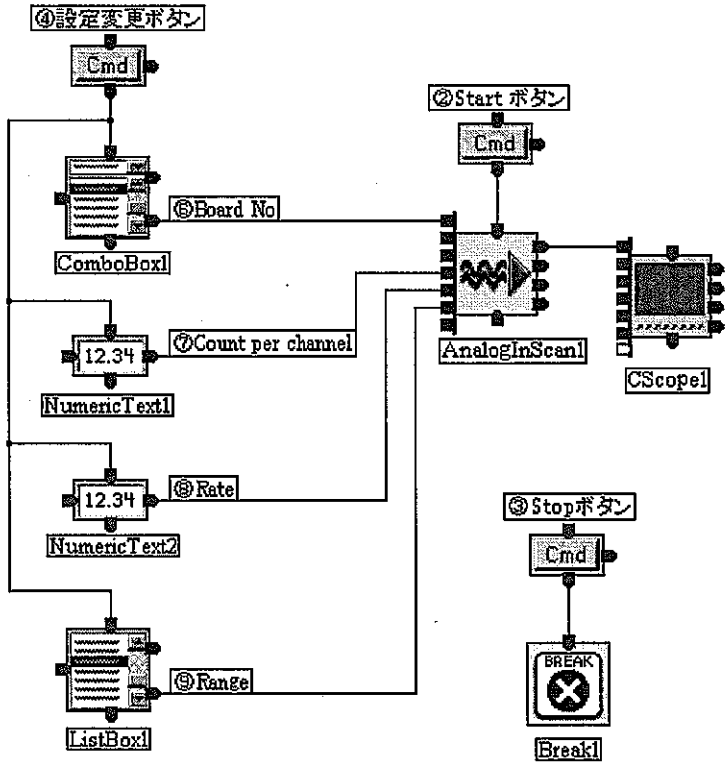


図 3.3.2-2　SoftWIRE 部品の配置図

3.3.2.3 ビジュアルプログラミングツールのPARTSシステムへの適用

PARTS-FLOWシステムをビジュアルプログラミングツール「SoftWIRE」で開発した際のイメージ図を示す。

アイコンの1つ1つがプラントの実機に対応しており（炉心アイコン、IHXアイコン、ポンプアイコン等）これらをリンクで結び付けられることで、実機プラントと同様の系統が構築できる。各機器に必要なパラメータ項目（材質・寸法等）は個々のアイコンをダブルクリックすることで入力画面が表示される。

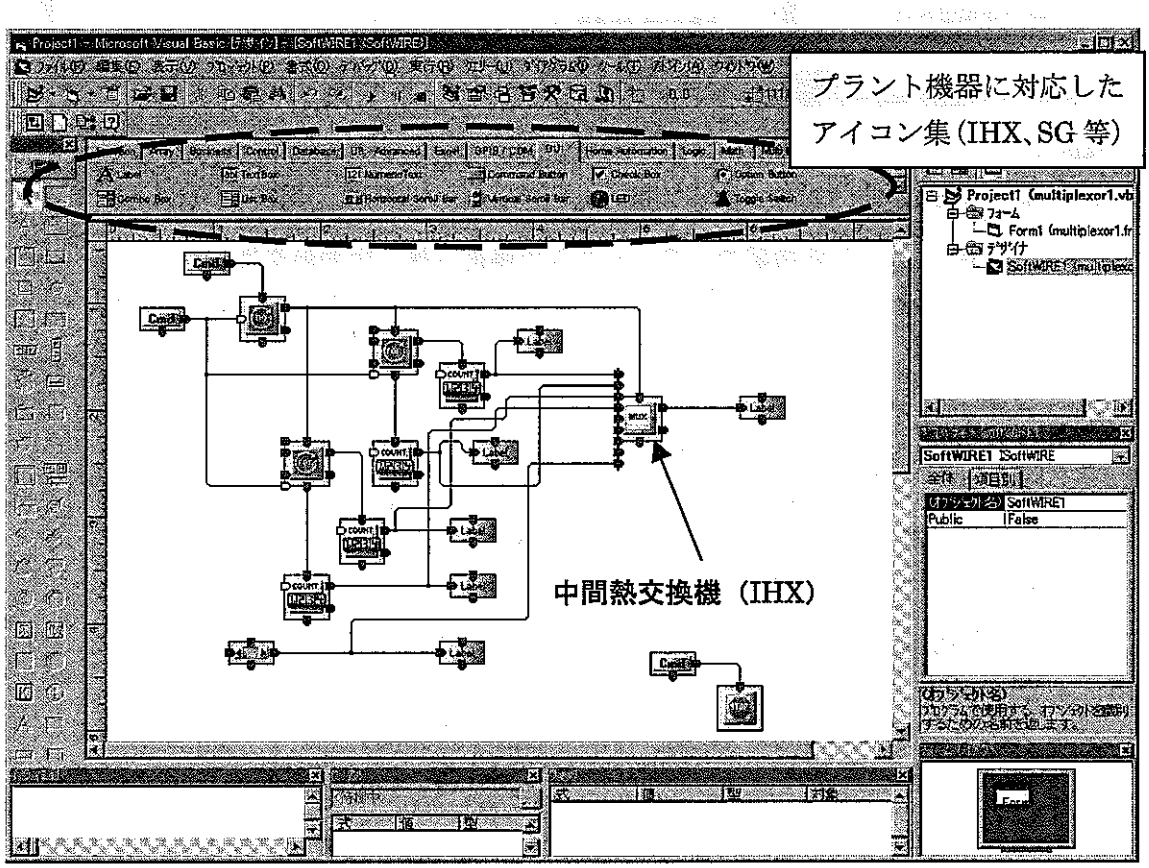


図3.3.2-3 PARTSシステム 機器部品編集時画面

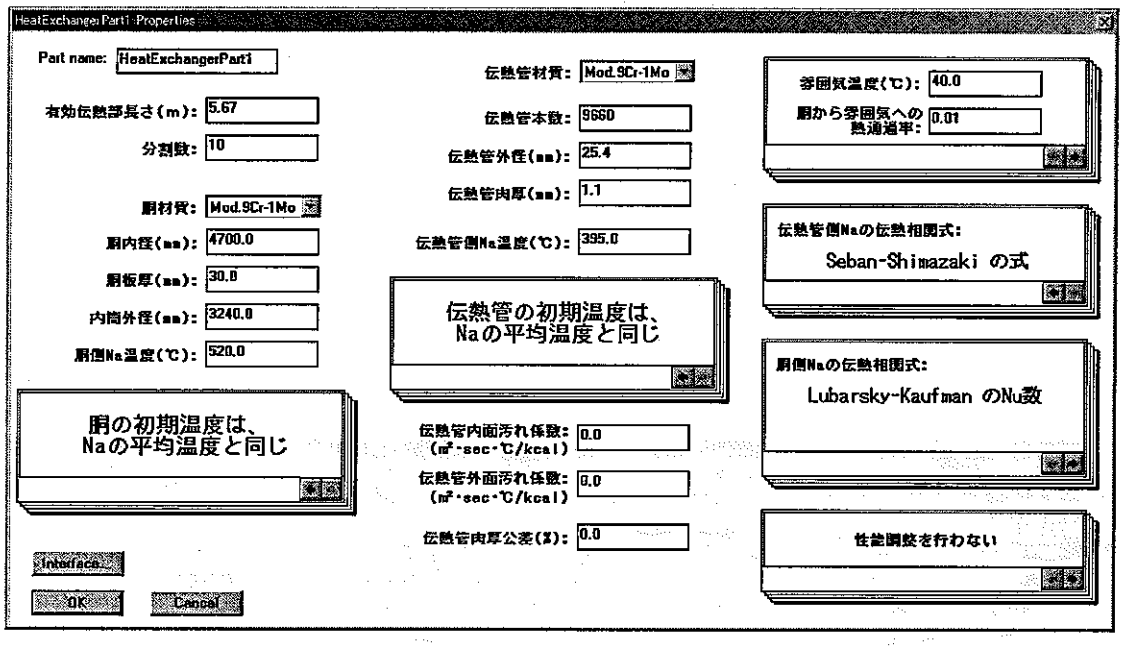


図 3.3.2-4 各機器のパラメータ入力画面（例：中間熱交換機）

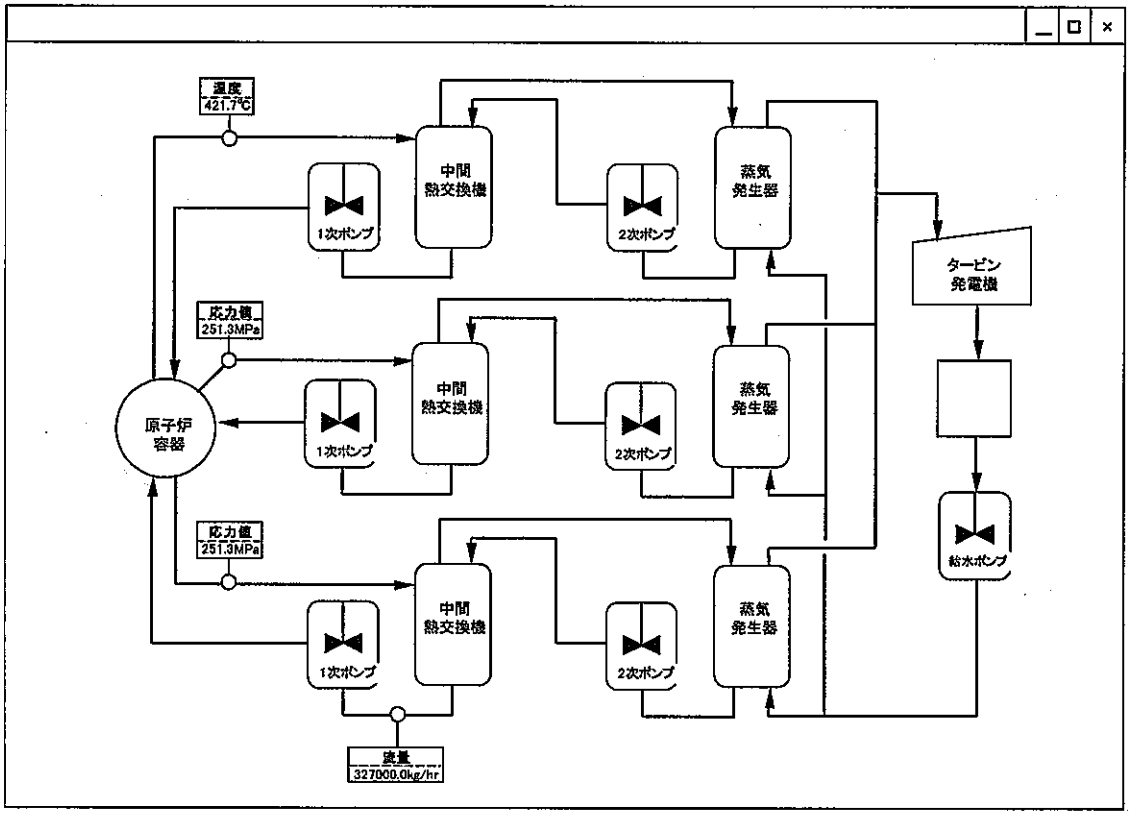


図 3.3.2-5 PARTS システム実行時画面

PARTS システム実行の流れ

① プラントモデルの編集

図 3.3.2-3 が PARTS システムの編集画面であるが、これは VisualBasic 標準のものではなく、ビジュアルプログラミングツール「SoftWIRE」の画面である。破線で囲まれた領域にプラント機器に対応したアイコンが多数登録されている。炉心、中間熱交換機（IHX）、蒸気発生器（SG）、配管、ポンプ等、これらをマウス操作で下の画面に貼り付け、更に機器間をリンクで結ぶことでプラント系統図を作成される。

各機器固有のパラメータに関しては、図 3.3.2-4 の画面において機器の形状、材質等を入力する。例：中間熱交換機（IHX）これで即実行が可能な状態になった。この間ソースプログラムの入力・変更等は一切行っていない。全てマウス操作のみである。

② 実行

実行ボタンを押すことで、過渡状態（手動トリップ）のシミュレーションが開始される。結果データは、リアルタイムで外部プログラムである Excel にプロセス間通信により、受け渡される。

尚、実行時画面を図 3.3.2-5 に示す。

3.4 ＰＣクラスタ上の分散コンピューティング

現在最も注目されている高速計算環境は、コストパフォーマンスの上昇が著しい多数のＰＣを高速ネットワークで結合したPCクラスタである。機能別にコンポーネント化されたプログラムは、コンポーネント毎に個別の計算ノードで計算実行を行う分散コンピューティングに適している。計算実行とデータ転送の負荷バランス機能を付加したＰＣクラスタ上の分散コンピューティングにより2.4節の課題を解決する。

3.4.1 分散コンピューティング

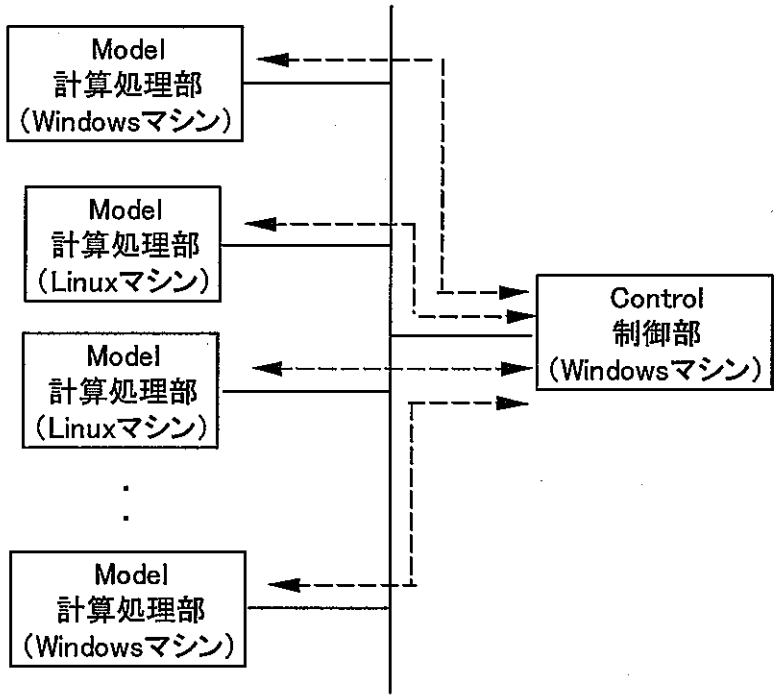


図3.4.1-1 分散コンピューティング技術（並列計算）

具体的には以下のポイントを実施する。

- 制御部と計算処理部という機能の明確なる分離
- インターフェースと処理部の分離
- 既存の分散コンピューティング技術の活用
  - Microsoft .NET、MPI
- 既存のインターフェース技術の活用
  - HTTP、TCP/IP

これらにより以下の効果が得られる。

- 汎用的で実績のある分散コンピューティング技術の活用により、さまざまな計算機との間で並列計算が可能になる。
- 機能を分離する事で実行時間の掛かる処理を高速計算機に移し負荷の分散が図れる。
- 計算機間の通信には既存インターフェース技術を活用するため、新たな作り込みが必要なく、信頼性の高い通信が確保出来る。

3.4.2 並列処理

並列ライブラリ（MPI 等）を用いた並列処理を行う上で最大の処理効率を得るために考慮しなければならない点を以降にまとめる。

最も重要な点は、対象問題を抽象化し概念モデルを作成する際、如何に並列性を抜き出しモデル化出来るかという点である。

3.4.2.1 並列処理の効率

並列処理において最も重要な観点は複数のプロセッサを効率的に動作させることであり、このためには対象問題、アルゴリズム、実装などの種々の局面で内在する並列性を抽出しなければならない。図 3.4.2-1 は問題領域からハードウエアに至るまでにおける並列性の度合いを示したものである。縦軸は並列して行える独立なオペレーションの数を示している。ＯＳとハードウエアでのギャップは、たとえばプロセッサの処理速度とプロセッサネットワークのバンド幅の相違による。

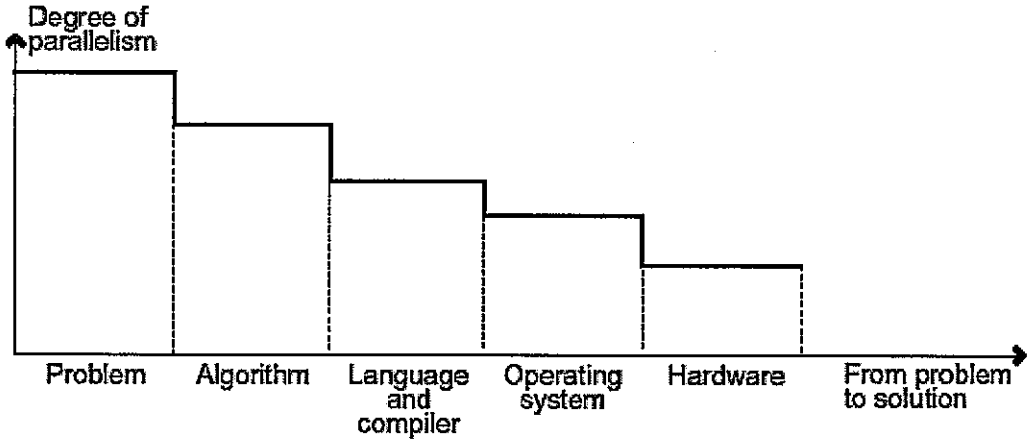


図 3.4.2-1 各レベルにおける並列性

ここで、マトリックス演算 $C=AB$ を考える。各マトリックスのサイズを $n \times n$ とすると、並列計算の複雑度は $O(\log n)$ である。すべての要素の乗算は並列で行うことができるが、n 回の加算に時間 $\log n$ が必要となる。しかし、n 回の加算を逐次的に行うとこのアルゴリズムの複雑度は $O(n)$ となる。ここに問題とアルゴリズムとの並列度のギャップが生じる

種々の問題を並列計算機に写像し、プロセッサ間のロードバランスをとるのは極めて難しい問題である。さらに、これがうまくできてもどこかの段階に修正が生じると全体を見直す必要が生じる。すなわち、並列計算は図 3.4.2-1 のすべての段階が密に関連している。

$$E_p = \frac{T_1}{pT_p} \qquad (3.4.2\text{-}1)$$

並列処理の効率は簡単には式 2.1 で表される。すなわち、p 個のプロセッサを用いたときの効率は 1 個のプロセッサで行ったときの逐次処理計算時間 T_1 を並列処理したときの計算時間 Tp と p の積で除したものである。ここで、逐次処理では最良のアルゴリズムを用いる必要がある。

$$R(f) = \frac{1}{(1-f) + f / R_p} \qquad (3.4.2\text{-}2)$$

並列処理の数学的な効率としては Amdahl の法則がある。これは並列処理の部分とそうでない部分の比率の関数として全体としての処理速度を表したもので、全体の処理速度が低速部分に大きく影響されることをよく理解することができる。

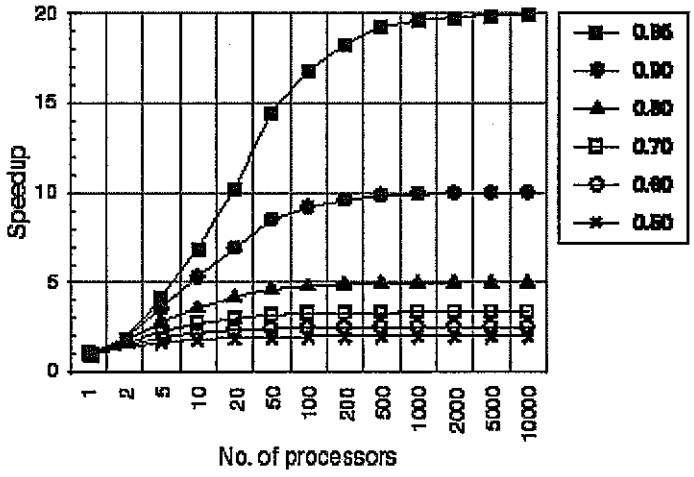


図 3.4.2-2 並列化の速度向上

ここで、f は全体の処理における並列化可能な処理の割合であり、$R(f)$ は総合的な速度向上率、Rp は並列処理の速度向上率である．これを図示すると**図 3.4.2-2** となる．

3.4.2.2 並列処理の効率に影響を与える要因

① 処理全体における逐次処理の割合

たとえば逐次部分の割合がたとえ 5 %と少なくても、逆に言えば並列処理部分が 95 %でも、1 万台のプロセッサで得られるスピードアップはわずか 20 倍にしかならない。いかに逐次処理部分の割合が全体の効率を下げているかが分かる。

②プロセッサ間の通信速度

並列処理における効率に重大な影響を及ぼすもう一つのファクターはプロセッサ間通信の速度である。この速度が遅く、しかもプロセッサ間通信が多い場合には並列処理の効率は著しく悪くなる。

プロセッサ間通信の頻度は処理の粒度（grain size or granularity）に関係している。粒度とは複数のプロセッサに与えられるタスクの大きさのことである。一つのステートメントあるいはそれ以下のタスクを細粒度（.ne grain）、ループまたはサブルーチンレベルのタスクを粗粒度（coarse grain）、それらの中間的な大きさのタスクを中粒度（medium grain）とよぶ。粒度が小さくなればプロセッサ間通信が多くなり、反対に粒度が大きくなればプロセッサ間通信は少なくなる。このため、プロセッサ間通信の速度が遅く、細粒度の場合は並列処理の効率が著しく悪くなる。特に、ＰＣクラスタの場合、一般にはプロセッサ間通信の速度が遅く、このため、細粒度での並列処理を行わせると複数台のプロセッサを用いているにも拘わらず、１台のプロセッサを用いたときより遅くなることもある。このため、特にＰＣクラスタを用いる場合には、できるだけ粒度を大きくしなければならない。

③ロードバランス

並列処理の効率に大きな影響を与えるもう一つのファクターはロードバランスである。ロードバランスとは並列計算機の各プロセッサに与えられる計算負荷の均等性のことであり、並列処理の効率を高めるには、できるだけロードをバランスさせなければならない。

各プロセッサに与えられる負荷が不均一であると、処理に不可避な同期待ちが多くなり、処理効率は著しく悪化する。たとえば、負荷が２倍異なれば計算時間が２倍異なり、同期をとるために遅いプロセッサを待つことになり、いくら他のすべてのプロセッサが早く処理を済ませても一番遅いプロセッサに律速される。このため、処理効率はこのことだけで 50 %程度に悪化する。

こうした問題を克服するため、各プロセッサに与える負荷は出来るだけ均一になるようにしなければならない。しかしながら、これは難しい問題である。なぜなら、画像処理などのようにデータの量に応じて負荷が決まる場合はデータを分散させればロードバランスはとれるが、連立一次方程式の Gauss-Seidel 法に基づく解法や傾斜法に基づく最適化計算など、多くの繰り返し計算手法ではデータの質（内容）によって収束までの時間が異なるからである。

そのために、最初から静的に負荷を分割するのではなく、その時々の各プロセッサの負荷を測定し、負荷を動的に均一化させる方法も有用である。しかし、この方法では負荷を管理するプロセッサが別に必要となり、しかも、負荷の測定のためにプロセッサ間通信が必要となり、別のオーバーヘッドが大きくなる。

3.4.2.3 数値計算の構造

次に考えなければならない点は、数値計算に用いる概念モデルと計算モデルの組み合わせを数値計算の構造と呼ぶことにすると、数値計算の構造と並列処理の適合性はどうか、ということである。

図 3.4.2-3 はこの関係を示している。最初に特定のドメインでの問題があり、それを解くための概念モデルが作られる。このモデルの種類によってこの図でいう数値ソルバーを用いるか、 自然ソルバーを用いるかが決まる。そしていずれの場合にも分割と写像が重要なポイントとなる。このブレークダウンができれば仮想的な計算機モデルを作ることができ、そのモデルに依存してどのようなアーキテクチャの計算機を用いるかが決まる。たとえば固体解析に有限要素法を用いるとして、それを並列化する場合には粗粒度の分割、すなわち領域分割により計算の並列化を行うこともでき、一方、細粒度の分割、すなわち内部でのベクトル計算のループを並列化することもできる。最近ではシミュレーテッドアニーリング、遺伝的アルゴリズム、あるいはニューラルネットワークなどの、この図でいうところの自然ソルバーを用いる場合も多く、この場合にも異なった並列化の計算モデルが考えられる。たとえば、遺伝的アルゴリズムで、全体的母集団における個体を複数プロセッサに割り当てて進化を進める方法や島モデルで個体群を分散させて独立して進化させ、ときどき移民させるというモデルもある。いずれにせよ、シミュレーションで重要な点は最初の概念モデルであり、そして次の計算モデルであり、これらの組み合わせで多くのタイプの並列処理が可能となる。

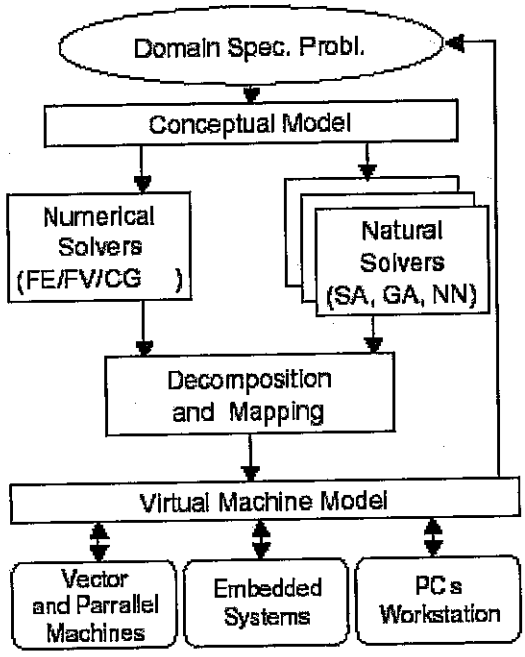


図 3.4.2-3 数値計算の構造

3.4.2.4 数値計算の並列化

① 差分法の並列化

ラプラスの方程式やポアソン方程式で表される種々の物理現象（重力場、静電場、静磁場、定常熱伝導、定常物質拡散、渦なし非圧縮性流れ、弾性体の平衡、結晶の成長など）のシミュレーションは差分法に基づいて行われることが多い。この場合、並列化は細粒度では Red-Black 法などによって色分けされた並列化可能な格子点をグループ化したものをプロセッサに割り当てるか、あるいは中粒度では複数の格子点を格子状にグループ化した領域を並列可能性に基づいてプロセッサに割り当てる。もしくは大粒度では全体をプロセッサ数だけの領域に分割して行う。いずれの方法でも Jacobi 型の緩和法を用いるのか Gauss-Seidel 型の緩和法を用いるのかによって計算モデルが異なり、計算速度と並列化効率のバランスが変わる。

② 有限要素法における並列化

有限要素法で定式化されたシミュレーションモデルでは細粒度ではベクトル計算の並列化となり、これは HPF などの並列プログラミング言語で比較的容易に並列化できる。大粒度での並列化では領域分割法（Domain Decomposition）が中心となっている。 領域分割法は有限要素法におけるデータ分割あるいは負荷分散である。この方法では対象を幾何学的に複数の領域に分割し、それらを対応するプロセッサに割り当てる。そして、領域内の

計算が終了すれば互いに境界条件を交換し、再度、領域内の計算を行う。これを反復することで全領域の計算を行う。領域分割法では領域の分割の方法によってプロセッサのロードバランスが大幅に異なるため、並列化効率を上げるにはこの点に注意する必要がある。

③ 粒子法による物質の挙動シミュレーション

多数の粒子の相互作用を基本として各粒子の位置や速度を求め、粒子全体からなるシステムの挙動を計算するアプローチを粒子法（particle method）という。相互作用が重力のときは天体力学における N 体問題となる。一方、相互作用がクーロン力の場合には分子動力学などのアプローチを用いて原子あるいは分子の配置を決定する解析が行える。

N 体問題の場合、重力は長距離力であり、N 個の粒子の計算には $N(N-1)$回の相互作用を求める必要がある。これは N が大きくなると膨大な計算となる。これを回避するには空間を複数のセルに分割し、そのセルに含まれる粒子の平均的挙動をセルの状態として考え、ある一定の距離以上のセル内の粒子についてはセルとしての代表的な状態で近似する高速粒子法が必要となる。これによって計算回数は $N \log N$ ぐらいまで減少させることができる。ただし、この場合には空間に固定されたセル内の粒子の出入りを監視することが必要になる。

一方、分子動力学では原子間のクーロンポテンシャルが長距離で減衰が激しいことを利用し、ある一定の距離以上では他の粒子を考えなくても良い。これによって膨大な数の原子を取り扱うことができる。この場合に原子間距離の計算回数はやはり $N(N-1)$ 回となるため、たとえポテンシャルを計算しなくても距離計算の負荷が大きく、これを避けるには着目している原子の近傍粒子とそうでない粒子を分類し、その分類自体は時折行うことで毎回の距離計算自体を無くす粒子登録法（Bookkeeping method）が用いられる。

このような粒子法の並列化は大粒度では空間でのセルあるいはクラスター化した粒子群を一つのプロセッサに割り当てる方法であり、細粒度での並列化はベクトル計算における inner loop の並列化である。ただし、後者においてもプロセッサのロードバランスを良好にするために単一プロセッサの場合とは異なるアルゴリズムが必要となる。

現在の自動並列化プログラミング言語は基本的に DO-LOOP の並列化が中心である。特に innerloop の並列化には効果的である。しかしながら、大規模シミュレーションの多くが最終的には膨大なベクトル計算に帰着するのはあくまでも図 3.4.2-3 でいうところの数値ソルバーを用いる場合である。しかしながら、同図でいう自然ソルバーにはまだまだ大きな発展の可能性があり、それらのアプローチでは種々の並列化の方法が可能となる。しかも、そうした自然ソルバーは本質的に並列処理を内在している場合が多い。このため、そうした自然な並列性を抽出して並列処理モデルを構築することはこれからの並列処理の可能性を大きく広げるものと考えられる。

第4章　オブジェクト指向技術

4.1　オブジェクト指向技術の概要

4.1.1　オブジェクト指向の特徴

4.1.1.1　概要

オブジェクト指向の考え方の源は Simula というプログラミング言語にさかのぼる。Simula は 1960 年代後半にノルウェーで生まれたシミュレーションのための言語であり、現在のオブジェクト指向の基本的な考え方を兼ね備えていた。この考え方は、Xerox パロアルト研究所で開発された Smalltalk という言語に受け継がれ、Lisp や C などにも影響を及ぼし、現在に至っている。したがって、25 年以上の歴史を持つ考え方で、けっして新しくはないのである。

Simula が生まれたのは、数千を越える部品から構成される複雑なシステムのモデルを作り、その動きを洞察したいという願いからであった。そのために、現実世界をそのまま投影できることを方針にして言語が設計された。ここで重要なのは、現実世界をそのまま投影するということである。

もし、1960 年代後半という時代におかれたならば、ソフトウェア技術者は、コンピュータ上に作成しやすいようにプログラムを作りはするが、現実世界をそのまま投影しようなどとは思わなかっただろう。むしろ、現実世界の方をコンピュータに合わせようとしたのではないだろうか。コンピュータの速度も資源も、今とは比べものにならない時代に、たいそうなことを考えたものである。

この現実世界をそのまま投影するという考え方を踏襲した言語に Smalltalk がある。この言語も、子供の学習過程を研究する目的で設計された。Dynamic Media (Dynabook)という携帯型の視聴覚入力装置が具備されたコンピュータを作り、それを子供たちに使わせてみようという発想である。その記述言語として生まれたのが Smalltalk である。この言語は研究所内で 1970 年代の全期間を通して段階的に改良され発展した。オブジェクト指向とは、現実世界のものを「=オブジェクト」として捉え表現することであるが、人にしても車にしても全てはもの（オブジェクト）であり、ものには働き（作用）が伴う、例えば人であれば呼吸をする、会話をする、飲食をする、車であれば走行する、給油をする等である。そしてこれらものは何らかのルールにより結びつき相互作用を実現している。こうした考え方によって現実世界をもの・作用・結びつきに置き換え、コンピュータにより代替(シミュレート)するのがオブジェクト指向技術の目標なのである。

4.1.1.2 オブジェクト指向の代表的考え方

オブジェクト指向の代表的な3つの考え方を以下に示す。

① 抽象データ型(オブジェクト)
② 継承（インヘリタンス）
③ 多相（ポリモルフィズム）

① 抽象データ型（オブジェクト）

抽象データ型は1970年代の中頃に提唱され、その利点は情報隠蔽(information hiding)やカプセル化(encapsulation)など、さまざまな言葉で表現される。データとそれに対する手続きを一体化し、データへのアクセスは決められた手続きのみによって行なわれるようにしたものである。この一体化することをカプセル化と呼び、あるデータを扱っている手続きがプログラム全域に散在することを防ぎ、情報の局所化を促進する。また、決められた手続き以外には、内部のデータをアクセスできないので、必要以上に情報が漏れることがない。このことを情報隠蔽と呼ぶ。これにより、データの構造などを変更しても、それを利用する場所に変化を及ぼさなくなる。抽象データ型は、オブジェクト指向でいうところのオブジェクトとほぼ等しい。オブジェクトとは次の式で表される。

オブジェクト = データ + 手続き + メッセージ送受信機能

オブジェクトは、データ、手続き、メッセージ送受信機能をパックしたものであり、データは独立に存在することが許されず、また、手続きも独立に存在することが許されない。データと手続きは常に一緒であり、そこにメッセージ送受信機能が付加される。オブジェクト指向の言葉を使えば、隠蔽された三つはそれぞれ、インスタンス変数(私的領域)、メソッド(私的操作)、メッセージパッシング(会話能力)と呼ばれる。

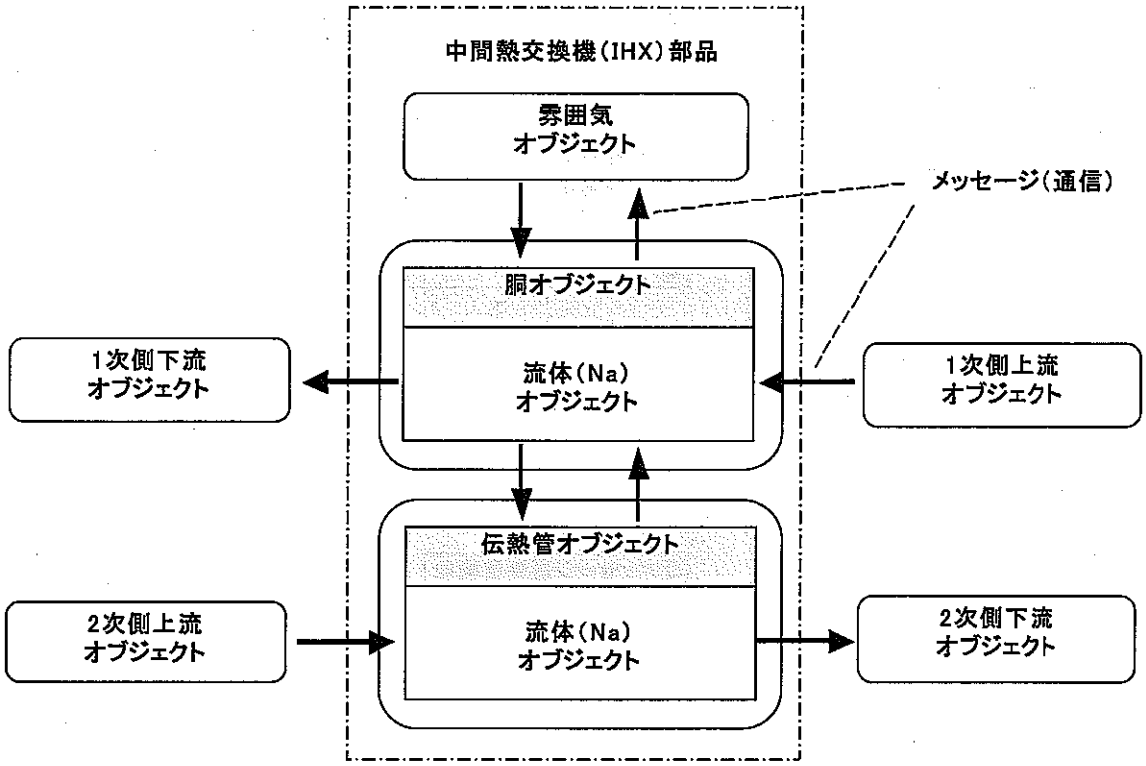


図 4.1.1-1　オブジェクトによる実機プラントの置き換え（例：中間熱交換機-IHX）

プラント機器は、複数のオブジェクトにより構成されている。

各オブジェクトは相互に関連しておりメッセージ通信（処理依頼）により制御の流れ・データ受け渡しを行っている。

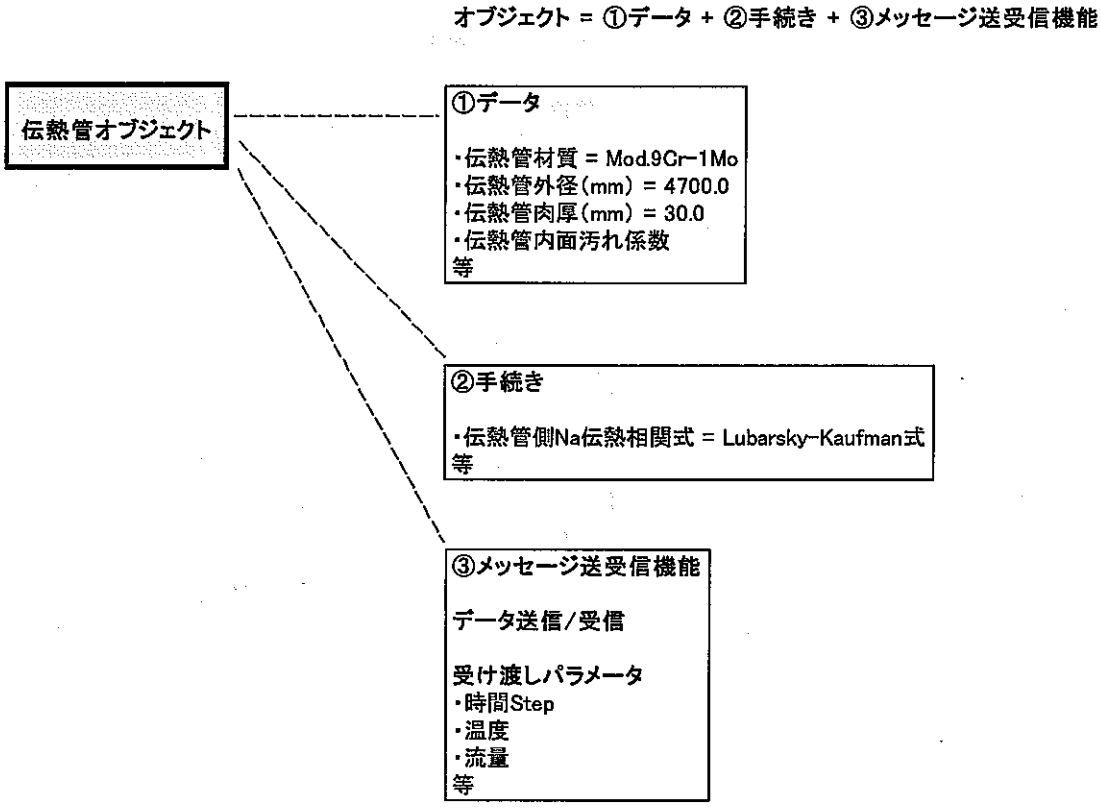


図 4.1.1-2　オブジェクトの構成要素（例：IHX 胴オブジェクト）

```
PF_HeatExchangerPart
        subclass: #PF_DHXPart
        instanceVariableNames:                    'velocityInPrimaryCircuit
previousVelocityInPrimaryCircuit     coefficientForConversionOfFrictionalLoss
componentsInPrimaryCircuit  dhxPlenumPart  averageTemperatureInPrimaryCircuit
averageTemperatureOfOrigPartInPrimaryCircuit
averageTemperatureOfTermPartInPrimaryCircuit
ratioOfInitialFlowInPrimaryCircuitToMainFlow    indirectPartsInPrimaryCircuit
flowInPrimaryCircuit multiplier'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'PF Parts - Concrete'!

PF_PrimaryPart
        subclass: #PF_HeatSourcePart
        instanceVariableNames: 'heatComponents regularHeat heatSchedule'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'PF Parts - Concrete'!
!PF_HeatExchangerPart methodsFor: 'Calculations' !

blockForOverallHeatTransferCoefficientFromSecondToShell
        "Answer the block for the overall heat transfer coefficient from the shell
side fluid to the shell."

        ^self blockForOverallHeatTransferCoefficientFromFluidToSolid!

blockForOverallHeatTransferCoefficientFromShellToAtmosphere
        "Answer the block for the overall heat transfer coefficient (netsu tsu-ka
ritsu) from the shell to the atmosphere."
```

図 4.1.1-3 Smalltalk プログラムにおける抽象データ型

```
        (self ignoreHeatTransferFromShellToAir or: [self
ignoreHeatTransferFromSecondToShell])
                ifTrue:
                        [^[:aComponent |
                                0.0]]
                ifFalse:
                        [^[:aComponent |
                                self
overallHeatTransferCoefficientFromShellToAtmosphere]]!

blockForOverallHeatTransferCoefficientFromTubesToSecond
        "Answer the block for the overall heat transfer coefficient from the tube
to the shell side fluid."

        ^self blockForOverallheatTransferCoefficientToOuterComponent2!

calculateOverallHeatTransferCoefficientFromFirstToTube: aComponent
        "Private - Answer the overall heat transfer coefficient from the tube side
fluid to the tube."

        ^self calculateOverallHeatTransferCoefficientFromFluidToSolid2!

calculateOverallHeatTransferCoefficientFromFluidToSolid: aComponent
        "Answer the overall heat transfer coefficient (netsu tsu-ka ritsu) from
the fluid to the solid."
```

図 4.1.1-3　Smalltalk プログラムにおける抽象データ型

② 継承（インヘリタンス）

インヘリタンスとは、属性や機能を別のものから受け継ぐことである。属性とは、オブジェクトの内部に閉じ込められたデータ構造や状態のことであり、インスタンス変数や私的領域などと呼ばれる。機能とは、オブジェクトの内部に閉じ込められた手続きや挙動のことであり、メソッドや操作などと呼ばれる。属性や機能を与える側(スーパクラス)と受け取る側(サブクラス)の特別の関係とは、どのような関係であろうか。オブジェクト指向では、世の中に存在するさまざまな関係の中から、もっとも汎用な is-a や kind-of 関係だけを特異的に取り出し、抽象データ型(オブジェクト)の整理に利用したのである。整理された抽象データ型群がとる関係構造をインヘリタンスと名付けたのである。

結局、インヘリタンスは、問題領域に登場する複数のオブジェクトすなわち抽象データ型を、is-a や kind-of 関係に従って、分類し整理するための機構だとみなせる。分類し整理する際には、差分プログラミングが良好に働くようにする。また、分類し整理した結果は、問題領域によって異なるのが普通であろうが、できる限り異なった問題領域へも対応できるように、インヘリタンスを成長させることが肝要である。そうすれば、複雑極まりないソフトウェアの認知的経済性を指向する道が開けるであろう。

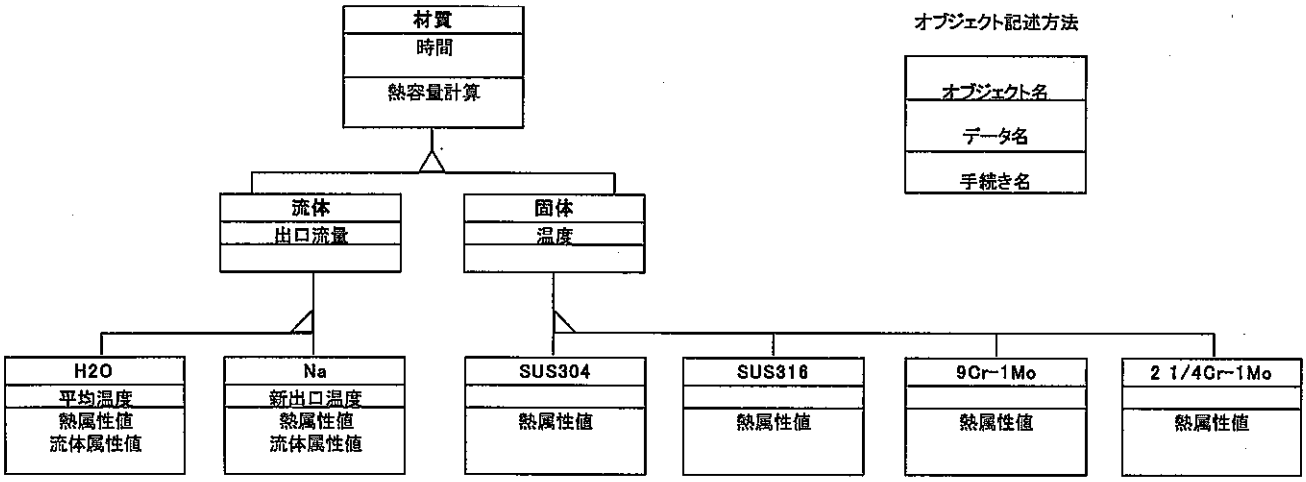


図 4.1.1-4 オブジェクトの継承（例：材質）

```
PF_Material
        subclass: #PF_Fluid
        instanceVariableNames: 'exitFlow newExitFlow exitTemperatureBlock
exitFlowBlock origFluids termFluids valveBlock steadyExitFlow'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'PF Material - Abstract'!

PF_Fluid
        subclass: #PF_H2O
        instanceVariableNames: 'exitEnthalpy volume oldDensity
oldAveragePressure exitPressureBlock averageTemperature oldDTime
newExitEnthalpy crossSection totalExitCrossSection averageEnthalpy
isEndTerminal isFrontTerminal closedFluids'
        classVariableNames: 'Scoef1Table Scoef2Table Scoef3Table Scoef4Table
Scoef5Table TscoefTable Scoef2ReverseTable'
        poolDictionaries: ''
        category: 'PF Material - H2O'!
PF_Fluid
        subclass: #PF_Na
        instanceVariableNames: 'flowControler exitTemperature
newExitTemperature steadyExitTemperature'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'PF Material - Na'!

PF_Na
        subclass: #PF_Air
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'PF Material - Na'!
```

図 4.1.1-5　Smalltalk プログラムにおける継承（インヘリタンス）

③ 多相（ポリモルフィズム）

オブジェクト指向では、オブジェクト(データ)にメッセージ(手続きの名前)を送る(渡す)ことが基本である。問題を解決するのは、手続きではなく、オブジェクト(データ)自身である。この考え方に基づけば、ポリモルフィズムが実現できる。ポリモルフィズムとは、同じ処理依頼(メッセージ)でも、メッセージの受け手によって、適切なメソッドすなわち手続きが選び出されることであるが、これを実現するためには、メッセージが送られるオブジェクトによって対応する手続きを捜さなければならない。そのため、手続きの名前と手続き本体はコンパイル時に結合することができず、実行時にオブジェクト(データ)にメッセージ(手続きの名前)を送って(渡して)、手続きの名前と手続き本体を動的結合(遅延束縛)するのである。この結合方法をメソッドサーチと呼ぶ。この結合方法を実現できれば、再配置可能モジュールを結合編集して、手続きのエントリのアドレス解決をする必要がない。本来のオブジェクト指向にはリンケージエディタ(リンクするという作業)は不要なのである。

4.1.1.3 自動メモリ管理（Automatic Memory Management）

(1) 概要

従来の開発言語（C・C＋＋等）では、取得したメモリを解放するタイミングは、プログラマが明示的に記述する必要があり、もし解放し忘れると、メモリ領域を無駄に消費しつづけることになる。このことを「メモリリーク」と言う。メモリリーク自体はメモリを圧迫するだけで、それほど危険ではないとも言える。現在のコンピュータは仮想メモリが実装され、物理メモリもふんだんに用意されているため、少々メモリリークしたとしても問題が起こるのは特定の場合に限られるからである。長時間作業しつづけるようなプログラムや、メモリを頻繁に確保&解放するようなプログラムでは、もちろんメモリリーク自体にも大きな問題があるが、それ以外の場合、メモリリーク自体に問題があるのではなく、メモリリークが生じているようなプログラムには、単純にメモリの解放を忘れているだけではなく、別のメモリを誤って解放してしまっているとか、ポインタの指しているメモリを解放する前にポインタの内容を書き換えてしまっているなどの問題が眠っている場合が多いのが問題なのである。特に、別のメモリを解放しているとすれば、誤った領域への書き込みがほぼ必ず生じていると考えられ、非常に危険である。つまり、メモリリークの問題と言うのは、メモリが足りなくなっていくという問題そのものよりもさらに悪い問題を暗に持つ場合が多いのである。実現できるかどうかは別に考えるとして、もし仮に、C言語の free() 関数やC＋＋言語の delete 命令がもともとプログラマが記述する必要が無かったとすれば、このような問題は生じないでしょう。つまり、メモリの解放タイミングを自動決定できる方法があるとすれば、メモリリークの問題はなくなるのである。

メモリリークとは別に、確保したメモリを解放したとしても、その領域が再利用されるとは限らないという問題がある。まとめて確保した複数の領域を、解放時にもまとめて解放した場合は別として、一部の領域だけを解放したとすると、メモリ空間の中で空き領域

が点在するようになってしまうからである。細かく点在した空き領域が再利用されるには、新しく確保したい領域のサイズが内部的なヘッダ部分も合わせて空き領域のサイズと丁度等しいか、それ以下の場合で無ければならないからである。このような、確保されているメモリ領域の中に解放後の空き領域が点在する現象は、メモリブロックの分断化と呼ばれる。C言語やC＋＋言語では、メモリ確保と解放のタイミングを非常にうまく行なわない限り、このような現象を防ぐ手立ては無い。他の高級言語では、この状態を時々修正する機能が備わっている場合が多いのである。メモリが不足した場合などに、「ガーベッジコレクション（塵集め）」と呼ばれる作業が行なわれる。ガーベッジコレクションでどのような処理が行なわれるかは、処理系依存であるが、一般的には使用領域の中から空き領域を取り出してまとめて集合させ、集合させた空き領域を使用領域の後方に移動してしまい、最終的に空き領域はメモリ空間から除外してしまうような作業が行なわれる。

プログラム実行中必要に応じてリソース（メモリ領域）を自動で確保し、更に使われなくなった領域を自動で解放し、メモリに関する問題を解消するのが自動メモリ管理である。

自動メモリ管理は、開発言語の仕様として実現されており、具体的には以下に示す 3 つの作業を行う。

① リソースの自動確保

プログラム実行中（動的）必要に応じてリソース（メモリ等）を自動で確保する。

② メモリ管理

③ リソース自動解放

使われなくなったリソース（メモリ含）を自動で解放する。ガーベジコレクター（GC: Garbage Collector）という処理系が使われなくなったという判断、解放のタイミング、性能を考慮したリソース解放を行う。

なお、リソースには、ファイルリソース、メモリバッファ、画面空間、ネットワーク接続、データベースリソース等が含まれる。

ちなみに Microsoft .NET の場合は、共通言語ランタイム（Common Language Runtime ） 環境がこの機能を担っている。Python、Ruby 等では、言語の固有の機能として自動メモリ管理の機能を備えている。

ガベージコレクタに管理された環境を作るのは、プログラマのためにメモリ管理を単純化するためである。

メモリに関する主な問題

・メモリリーク

・メモリブロックの分断化

・メモリ取得 / 解放時の CPU への負荷

解決する技術
・ガーベジコレクション（GC : Garbage Collection）

(2) GC（Garbage Collector）の概念と内部メカニズム

(a) リソースの確保

すべてのリソースは、OS 管理のヒープから確保しなければならない。オブジェクトは、アプリケーションにとって不要になったときに自動的に解放される。

今日使われている GC アルゴリズムは数種類ある。例えば、Ruby ではマークスイープ法、Python では参照カウント法が用いられている。本稿では、Microsoft .NET の共通言語ランタイムが使っている GC アルゴリズムに焦点を絞る。それでは、基本的な考え方から見ていこう。

プロセスが初期化されるときに、共通言語ランタイムはアドレス空間内の連続領域を予約するが、初期状態ではこの領域には物理メモリはいっさい割り当てられていない。このアドレス空間内領域が管理ヒープである。管理ヒープはポインタも管理している。本稿では、このポインタを NextObjPtr と呼ぶことにする。このポインタは、ヒープ内のどこから次のオブジェクトを確保すべきかを示す。初期状態では、NextObjPtr は、予約済みアドレス空間内領域の先頭アドレスにセットされている。

アプリケーションは、new 演算子を使ってオブジェクトを作成する。new 演算子は、まず、予約済み領域内に新オブジェクトが必要とするスペースがあるかどうかを確認する（必要に応じて、物理メモリをコミットする）。もしあれば、NextObjPtr はヒープ内の新オブジェクトを作成すべき位置を指しているので、オブジェクトのコンストラクタを呼び出し、new 演算子はオブジェクトのアドレスを返す。

この時点で、NextObjPtr は、次のオブジェクトを配置すべきヒープ内の位置を指すようにオブジェクトの後ろにインクリメントされる。図 4.1.1-5 は、A、B、C の 3 個のオブジェクトを含む管理ヒープを示したものである。次のオブジェクトは、NextObjPtr が指している位置（オブジェクト C の直後）から確保される。

実際には、ガベージコレクションは、ジェネレーション 0 がいっぱいになったときに発生する。簡単に言えば、ジェネレーションとは、ガベージコレクタが処理性能の向上のために実装しているメカニズムである。新しく作成されたオブジェクトは若いジェネレーションの一部となり、アプリケーションのライフサイクルの初期に作成されたオブジェクトは古いジェネレーションの一部となる。オブジェクトをジェネレーションに分割すれば、ガベージコレクタは、管理ヒープ内のすべてのオブジェクトではなく、特定のジェネレーションだけを集めれば済むようになる。

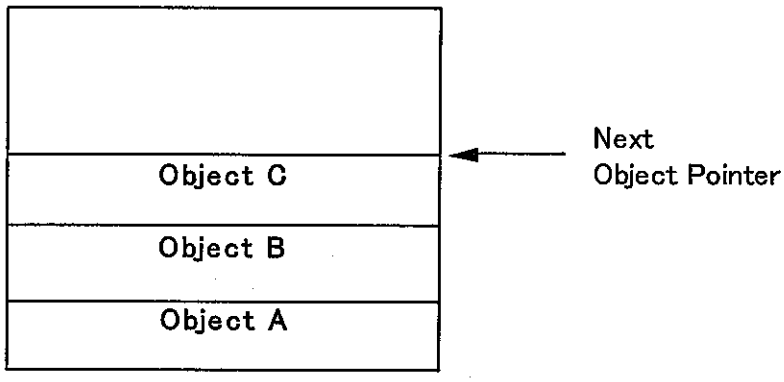


図 4.1.1-6　管理ヒープ

(b)　ガベージコレクションのアルゴリズム

ガベージコレクタは、もうアプリケーションからは使われていないオブジェクトがヒープ内にあるかどうかをチェックする。そのようなオブジェクトがあれば、それらが使っているメモリは回収できる（ヒープに使えるメモリがなければ、new 演算子は OutOfMemoryException を起こす)。では、ガベージコレクタは、アプリケーションがオブジェクトを使っているかどうかをどのようにして知るのだろうか。

これは簡単に答えられる問題ではない。

すべてのアプリケーションは、ルーツのセットを持っている。ルーツは記憶位置を識別するもので、管理ヒープ上のオブジェクトか、null をセットされたオブジェクトのどちらかを参照する。たとえば、アプリケーション内のすべてのグローバルな静的オブジェクトポインタは、アプリケーションのルーツの一部であるとみなされる。さらにスレッドスタック上にあるローカル変数／引数オブジェクトポインタも、アプリケーションのルーツの一部であると考えられる。最後に、管理ヒープ内のオブジェクトを指すポインタを格納する CPU レジスタも、アプリケーションのルーツの一部とみなされる。アクティブなルーツのリストは、JIT (just-in-time) コンパイラと Common Language Runtime によって管理されており、ガベージコレクタのアルゴリズムからアクセスできるようになっている。

起動時のガベージコレクタは、ヒープ内のすべてのオブジェクトをごみだと考えている。言い換えれば、アプリケーションのルーツのなかには、ヒープ内のオブジェクトを参照しているものはないとみなしているのである。次に、ガベージコレクタはルーツをたどり、ルーツから手の届くすべてのオブジェクトのグラフを作る。このとき、ガベージコレクタは、たとえばヒープ内のオブジェクトを指しているグローバル変数を見つける。

図 4.1.1-6 は、いくつかの確保済みのオブジェクトを持つヒープを示したものである。このヒープでは、アプリケーションのルーツがオブジェクト A、C、D、F を直接参照している。これらのオブジェクトはどれもグラフの一部になる。ガベージコレクタは、グラフにオブジェクト D を追加するときに、このオブジェクトがオブジェクト H を参照しているこ

とに気づく。そこで、オブジェクト H もグラフに追加される。ガベージコレクタは、すべての到達可能オブジェクトを再帰的にたどっていく。

すべてのルーツをチェックしたら、ガベージコレクタのグラフにはアプリケーションのルーツから何らかの形で手が届くすべてのオブジェクトの集合が含まれている。グラフに含まれていないオブジェクトはアプリケーションからはアクセスできないので、ごみとみなされる。ガベージコレクタは、今度はヒープを線型にたどり、ガベージオブジェクトの連続ブロックを探す（この部分は、フリー空間とみなされる)。次に、ガベージコレクタは、ごみではないオブジェクトをメモリの下位アドレスにシフトし（誰もが大昔から知っている標準の memcpy 関数を使う）、ヒープのなかのギャップをすべて取り除く。もちろん、メモリ内のオブジェクトをすべて移動すれば、オブジェクトを指すポインタもすべて無効になる。そのため、ガベージコレクタは、ポインタがオブジェクトの新しい位置を指すようにアプリケーションのルーツを書き換えなければならない。また、ほかのオブジェクトを指すポインタを持つオブジェクトがあれば、ガベージコレクタはこれらのポインタも修正する。図 4.1.1-6 は、ガベージコレクション終了後の管理ヒープを示したものである。

ごみをすべて見分け、ごみ以外のすべてのオブジェクトをメモリコンパクションにかけ、すべてのポインタを修正したら、NextObjPtr は最後のごみではないオブジェクトの直後に配置される。この時点で new 演算子は再び処理を試み、アプリケーションが要求したリソースの作成は成功する。

以上からも想像されるように、GC は処理性能をかなり損なう。管理ヒープを使うときの大きなデメリットは、これである。しかし、GC が発生するのはヒープがいっぱいになったときだけであり、それまでは管理ヒープは C ランタイムヒープよりもかなり高速だということを忘れないでいただきたい。Common Language Runtime のガベージコレクタは、処理性能を大幅に向上させるメカニズムも組み込んでいる。これらの最適化については、Part□でジェネレーションについて論じるときに取り上げる。

ここで注意しておくべき重要なポイントがいくつかある。もはや、アプリケーションが使うリソースの寿命を管理するコードを実装する必要はない。そして、本稿の冒頭で取り上げた 2 つのバグが消えたことに注意していただきたい。まず、アプリケーションのルートからアクセスできないリソースはいつかの時点でガベージコレクタに収集されるため、リソースリークが発生する可能性はなくなる。手が届かなければ、アプリケーションからそのリソースにアクセスする手段はないのである。次に、手の届くリソースは解放されないので、解放されたリソースにアクセスする可能性もなくなる。GC がそんなに偉大だというのなら、なぜ ANSI C++には GC が含まれていないのだろうか。その理由は、ガベージコレクタがアプリケーションのルーツを識別するとともに、すべてのオブジェクトポインタを見つけることもできなければならないからである。C++は、ある型から別の型へのポインタのキャストを認めるため、ポインタが何を参照しているのかを知ることができない。Common Language Runtime では、管理ヒープはオブジェクトの本当の型を知っている。また、オブジェクトのどのメンバーがほかのオブジェクトを参照しているかは、メタデータ情報によって分かる。

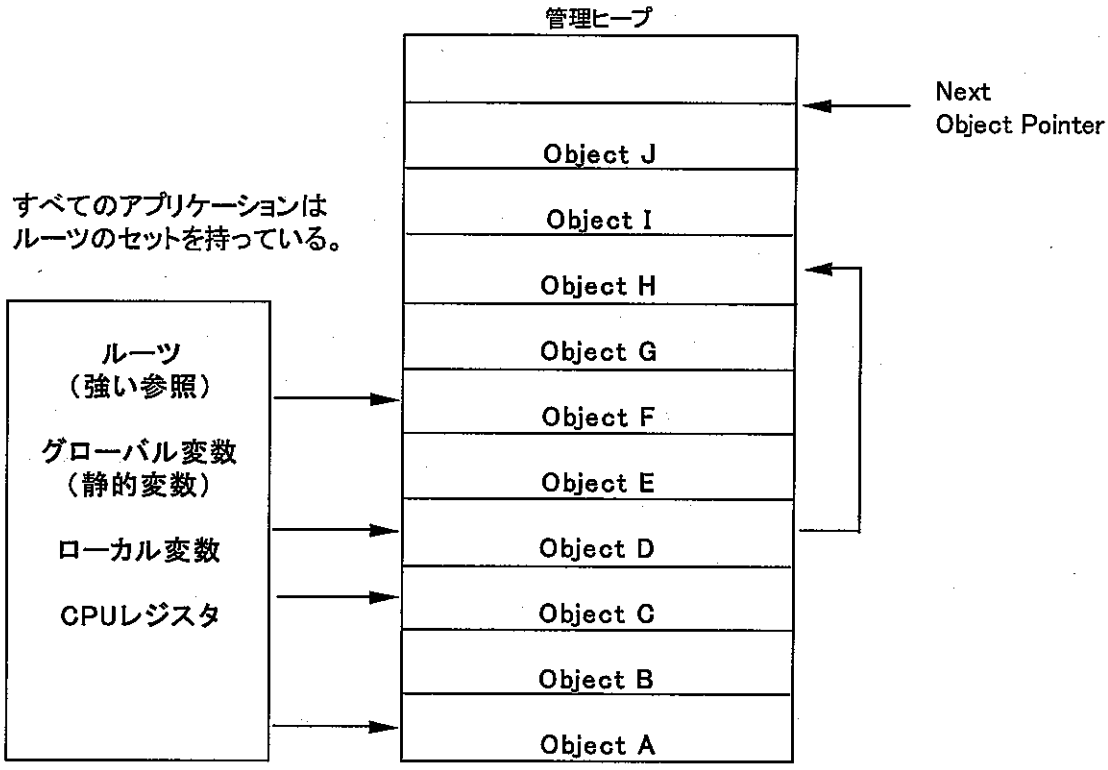


図 4.1.1-7　ヒープ内に確保されたオブジェクト

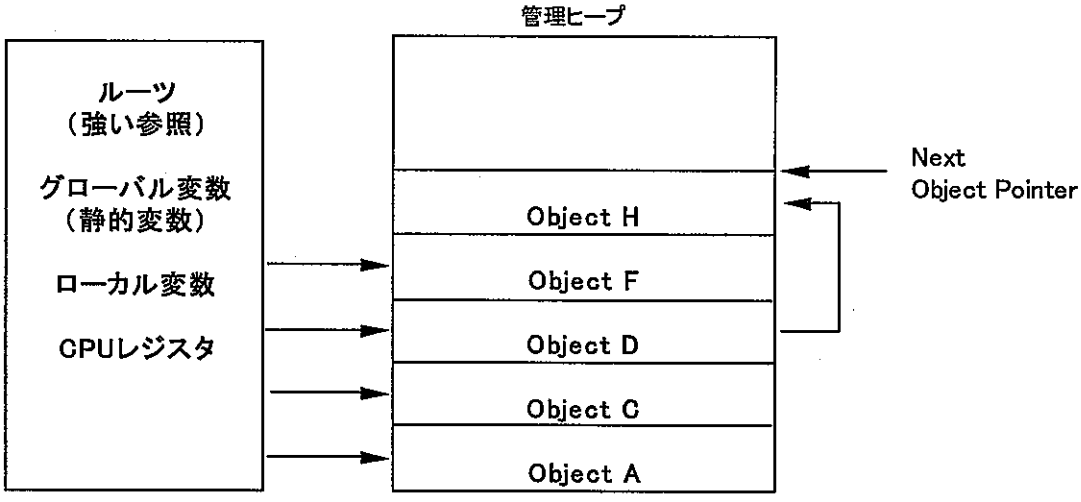


図 4.1.1-8　ガベージ コレクション終了後の管理ヒープ

(c) 完結化

ガベージコレクタには、プログラマの役に立つもう 1 つの機能、完結化機能がある。完結化により、ガベージコレクションにかけられたリソースが、自分自身を穏便にクリーンアップできるようになる。完結化機能があれば、ガベージコレクタがファイルやネットワーク接続を表現するメモリを解放することになっても、ファイルやネットワーク接続を表現するリソースは、ガベージコレクタがリソースのメモリを解放しようと判断したときに、自分自身を適切にクリーンアップできる。

完結化のために行なわれていることを実際よりも単純化して説明しよう。ガベージコレクタは、オブジェクトがごみになっていることを検出すると、オブジェクトの Finalize メソッドを呼び出し（存在する場合）てから、オブジェクトのメモリを回収する。たとえば次のような型があったとする（記述には C#を使っている）。

```
public class BaseObj {
    public BaseObj () {
    }

    protected override void Finalize() {
        // ここにリソースクリーンアップコードを入れる
        // 例： ファイルのクローズ／ネットワーク接続の切断
        Console.WriteLine("In Finalize.");
    }
}
```

そしてプログラムは次のコードを呼び出してこのオブジェクトのインスタンスを作成する。

```
BaseObj bo = new BaseObj();
```

将来のいずれかのとき、ガベージコレクタはこのオブジェクトがごみになっていることに気づく。ガベージコレクタはこの型が Finalize メソッドを持っていることを検出し、そのメソッドを呼び出す。コンソールウィンドウに“In Finalize.”という文字列が表示され、このオブジェクトが使っていたメモリは回収される。

C++プログラミングになれているプログラマの多くは、デストラクタと Finalize メソッドを直接結び付けて考えようとするだろう。しかし、オブジェクトの完結化とデストラクタとでは、セマンティクスがまったく異なっているので、完結化について考えるときには、デストラクタについて持っている知識はすべて忘れたほうがよい。管理オブジェクトはデ

ストラクタを持たない。それだけである。

```
update
        "Update the components."
        self subParts
                do:
                        [:aPart |
                        aPart update]! !
update
        "Update the components."
        super update.
        previousVelocityInPrimaryCircuit := velocityInPrimaryCircuit.
        velocityInPrimaryCircuit := self calculateVelocityInPrimaryCircuit.
        flowInPrimaryCircuit := nil.
        averageTemperatureInPrimaryCircuit := nil.
        averageTemperatureOfOrigPartInPrimaryCircuit := nil.
        averageTemperatureOfTermPartInPrimaryCircuit := nil.! !
update
        "Update the components."
        "currentTimeStep = 0 ifTrue: [^self]."
        super update.
        self components2
                do:
                        [:aComponent |
                        aComponent update]! !
update
        "Update the components."
        "currentTimeStep = 0 ifTrue: [^self]."
        super update.
        self heatComponents
                do:
                        [:aHeatComponent |
                        aHeatComponent update]! !
```

図 4.1.1-9　Smalltalk プログラムにおける多相（ポリモルフィズム）

4.1.2 オブジェクト指向分析・設計

4.1.2.1 UML によるオブジェクト指向分析とモデリング

(1) 概要

1980 年代の米国で誕生したオブジェクト指向技術は、何とか効率の良い開発は出来ないかというエンジニアの課題を克服するため生まれたものである。このオブジェクト指向技術に対応しさまざまな方法論（50 種類以上）が編み出されたが、方法論によって表記法が異なるためエンジニアの間で混乱が起こり始めた。そこで 1994 年これら乱立した方法論を統合する目的で UML（Unified Modeling Language：統一モデリング言語）が誕生した。

UML は、オブジェクト指向の考え方に基づいて問題の分析・設計を行い、更にプログラム開発までをサポートする統合的な表記法である。UML の結果生成物は、建築工事に例えれば設計図のようなものでシステムの青写真ともいうべき非常に重要なものである。

従来の分析・設計手法との大きな違いは、対象問題を機能として切り出し捉えるのではなく、そのままモノ（Object）として捉え、複雑な対象問題を出来る限り単純化するところに大きなポイントがある。

(2) オブジェクト指向分析 / 設計

UML はオブジェクト指向に基づいて 4 つの観点から対象問題を分析し、その結果を 10 の図表として記述する方法であくまで表記法であり、重要なのは 4 つの観点から問題を分析するそのやり方である。

まず 4 つの観点とは以下のものを指す。

① モデル化対象に対する要求を表す。
UseCase 図
② モデル化対象の静的な振舞いを表す。
Class 図、Object 図、Package 図
③ モデル化対象の動的な振舞いを表す。
Sequence 図、Colabolation 図、Activity 図、Statechart 図
④ モデル化対象の物理的な実装を表す。
Component 図、Deploy 図

対象問題をオブジェクト指向に基づいて分析 及び 設計を行う方法は、OOA（Object Orinted Analysis：オブジェクト指向分析）、OOD（Object Orinted Design：オブジェクト指向設計）と呼ばれており、OOA の代表的なものとしては、OMT（Object Modeling Technique）法・Booch 法等がある。こうした手法により分析を行う事が対象問題のモデル

化においては極めて重要であり、UML の各図表は OMT 法等の OOA を実施した結果の生成物と捉えるのが、一般的な見方である。

OMT (Object Modeling Technique) 法

Object Modeling Technique (OMT) は General Electric 社の研究開発センターで開発された手法であり、ほとんどの種類のアプリケーションの開発に役だったと主張されている方法論である。現時点で最も完成された技法なので、多少詳しく解説する。OMT では、オブジェクト指向を以下のように捉えている。

① 抽象
② カプセル化
③ データと振舞いの結合
④ 共有

継承はいくつかのサブクラス間のデータと振舞いの共有であり、継承を使ったコードの共有はオブジェクト指向言語の最も有利な点であるが、他にも、将来のプロジェクトとの設計の共有や、再利用可能なライブラリーの構築が容易といった利点がある。OMT は、SA からデータフローと状態遷移図を引き継ぎ、オブジェクトモデリングは E-R 図やデータベース設計技法から引き継いでいる。オブジェクトの認識方法は JSD 法から一部影響を受けている。他の OOA/OOD 技法のうち、特に Shlaer／Mellor の手法や Coad 法とは、オブジェクトモデリングの方法・記法が似ている。対象とする言語は、ほとんどすべての言語である。

OMT の場合特徴的なことは、OMT モデルをリレーショナルデータベースシステム (SQL 言語相当のデータベース照会言語を含む) で実現する際の詳細なガイドがある点である。OMT モデルは、オブジェクトモデル・動的モデル・機能モデルからなる。オブジェクトモデルはオブジェクト図からなり、オブジェクトやクラスの静的構造を表す。動的モデルは状態遷移図で表し、オブジェクトの状態・動作を記述する。機能モデルはデータフロー図で表し、機能の詳細を表す。OMT のオブジェクト図が表現できるオブジェクトは、以下のように豊富である。

① クラスとオブジェクト

クラスは Coad 法と同じように内部を分割した箱で表す。Person、 Company などがクラスであり、クラスを表す箱の一番上の部分がクラス名、２番目が属性、３番目が操作を表す。

② 属性

属性はクラスの記号の中央の部分に列記するが、データの型と初期値を記述することもできる。

③ 操作

操作はクラス記号の一番下の部分に列記するが、引数リストと返す値も記述でき

る。

④ 継承

継承は白抜きの三角形で表される。図 2.5-1 では Worker と Manager が、Person の性質を継承していることが示されている。多重継承を表すこともできる。（5）結合（association）　結合はクラス間を結ぶ線分で表される。

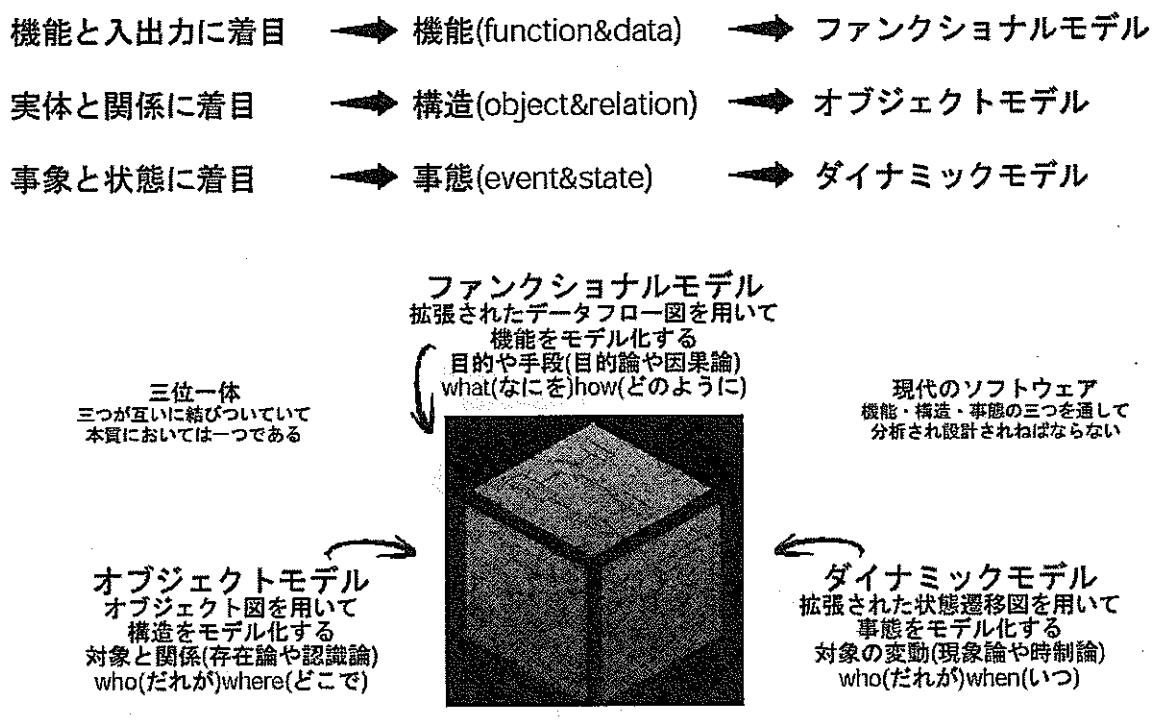


図 4.1.2.1-1　OMT 法による対象問題の分析及びモデル図の作成

(3) UML による効果

UML の利用により以下の効果が得られる。

① ユーザと開発者そして開発者間のコミュニケーションが向上する。
② リスク軽減と品質向上である。
③ 再利用性が高まり、生産性が向上する。

但し、オブジェクト指向=再利用という効果については決して自然に達成されるものではなく、どこまで、なにを再利用したいかを必ず意識していないとければならない。UML により、①～③の効果をあげるために必要なことは、UML 図作成、ソースプログラムへの自動

変換、開発・デバッグが統合的に実践出来る開発ツールを最大限に活用することである。

(4) UMLのダイヤグラムタイプ

UMLによる分析・設計を実施すると以下の①～⑧の8種類のダイヤグラム（図）が生成される。これらによりユーザ・開発者間、及び、開発者間のコミュニケーションが確実に行われ、スムースなシステム開発が実現される。

以下、レンタカーを管理するシステムを例にして各UML図について説明する。

① 機能ケース（ユースケース）ダイヤグラム

使用ケースは、ユーザーとシステムの間の、ユーザーが目標を達成するために行う対話を指定する。典型的なソフトウェア システムには単純な使用ケースが数百種類も含まれていることがある。

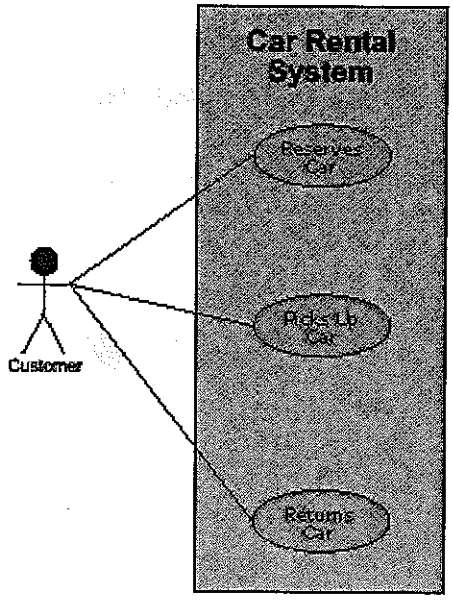


図4.1.2.1-2 ユースケース図

② 静的構造（クラス）ダイアグラム

次のタスクは、関連するオブジェクトとそれらの間の関係を分類することである。使用ケースを調べることで、クラスを識別することができる。オブジェクトのクラスは、システムの全体的な構造と、その関係上および動作上のプロパティを示す静的構造ダイアグラム、すなわちクラス ダイアグラムを使ってモデリングされる。

クラス ダイアグラムでは、カー レンタル システムに関わっているオブジェクトがクラスにグループ化される。各クラスは名前セクションと属性セクションを含んでいる。一部のクラスは、そのクラス内のオブジェクトがどのように振る舞えるかを指定する操作セクションも含んでいる。

Customer クラスの属性には、名前、電話番号、運転免許証番号、および住所などがある。誕生日は、カスタマが車を借りるための年齢制限を満たしていることを確認するために必要となる。また、Customer クラスは予約などの操作を含んでいる。

クラス ダイアグラムは継承をサポートしている。たとえば、次の図では、Mechanic クラスと Rental Agent クラスは、Employee クラスから名前や住所などの属性を継承している。

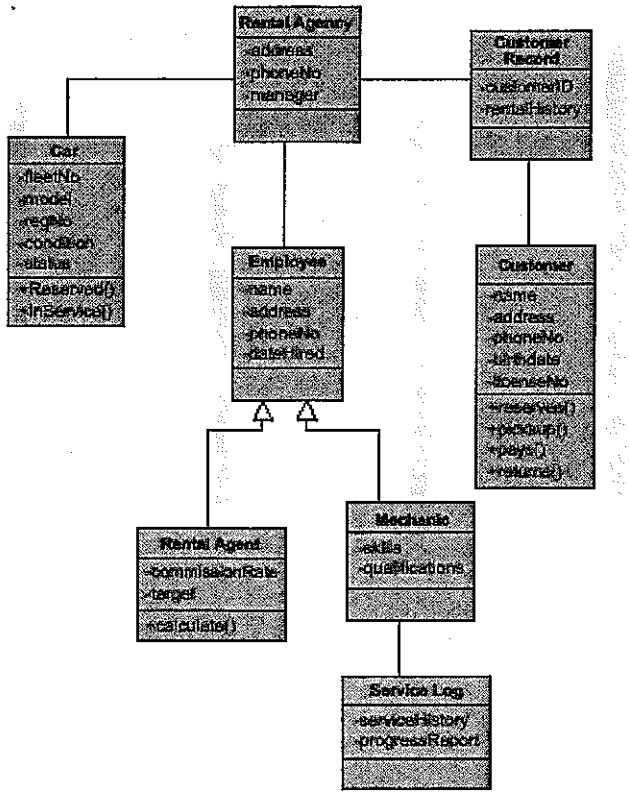


図 4.1.2.1-3　静的構造（クラス）図

③　シーケンス ダイアグラム

シーケンス ダイアグラムは、使用ケースの詳しいビューを提供する。このダイアグラムは、時系列的なシーケンスに並べられた相互作用を示し、アプリケーション内のロジックの流れをドキュメント化するのに役立つ。参加者は、それらの間で受け渡されるメッセージのコンテキストで表示される。包括的なソフトウェア システムでは、シーケンス ダイアグラムはかなり細かいものになり、数千のメッセージを含んでいることもある。

カスタマが車を予約したいと考えたとする。レンタカー会社は、まずカスタマのレコードをチェックして、予約を受け付けるかどうかを確認しなくてはならない。カスタムが以前にこの会社から車を借りていた場合、そのレンタル履歴はすでに記録されているので、レンタカー会社はそれ以前のすべてのトランザクションがスムーズに行われたことを確認しなくてはならない。たとえば、レンタカー会社は、カスタマのこれまでのレンタル カー

が予定どおりに返却されたことを確認する。カスタマのレンタカー ステータスが承認されたら、レンタカー会社はカー レンタルの予約を承認する。このプロセスは、次のようなシーケンス ダイアグラムとして表現することができる。

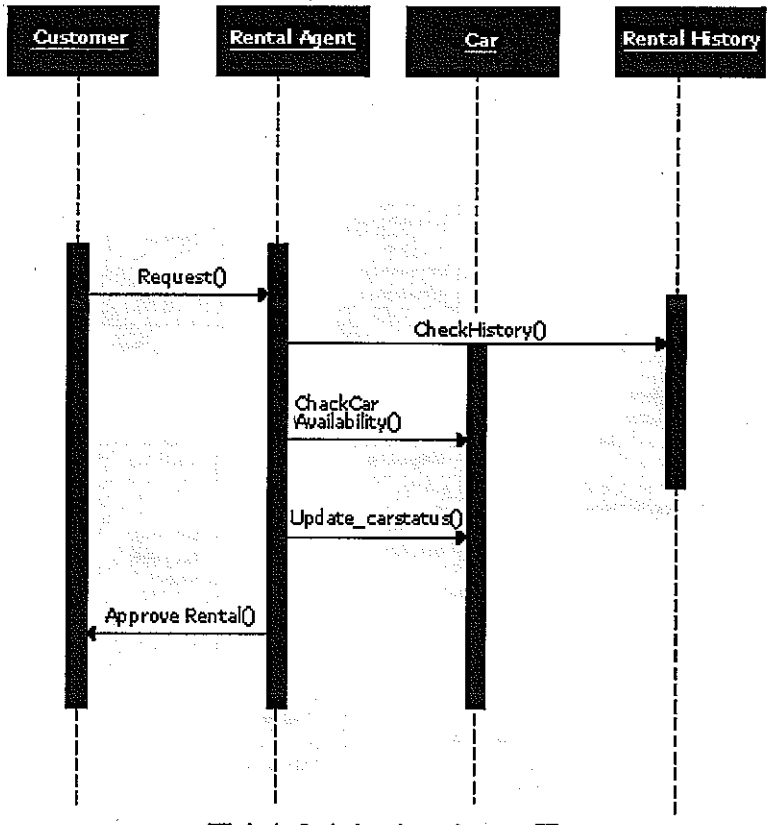


図 4.1.2.1-4　シーケンス図

④ コラボレーション ダイアグラム

コラボレーション ダイアグラムは、別のタイプの相互作用ダイアグラムである。シーケンス ダイアグラムと同様に、使用ケースの中のオブジェクトのグループが、互いにどのように相互作用を行うかを示す。個々のメッセージには、その発生順序を示す番号が付けられる。

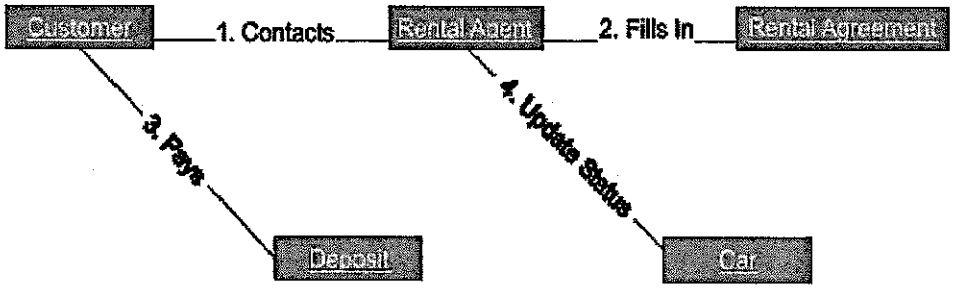


図 4.1.2.1-5　コラボレーション図

⑤ 状態チャート ダイアグラム

オブジェクトの状態は、特定の時点における属性によって定義される。オブジェクトは、外部からの刺激の影響を受けて、各種の状態の間を移行する。状態チャート ダイアグラムは、これらの状態と、オブジェクトを特定の状態にするトリガリング イベントを表示する。たとえば、カー レンタル システムでのオブジェクトは車である。車はレンタル システムの中を通る過程でさまざまな状態を取り、複雑な、しかし情報量の多いダイアグラムを生成する。たとえば、車はまず在庫に追加される。その後、レンタルされるまでは InStock 状態にある。レンタルされた後、車は在庫に戻され、InStock 状態に戻る。その寿命の中で、車は修理が必要になることがある。(InService)。耐用期間が終了した車は、新しい車のスペースを空けるために、売り払われるかスクラップされる。

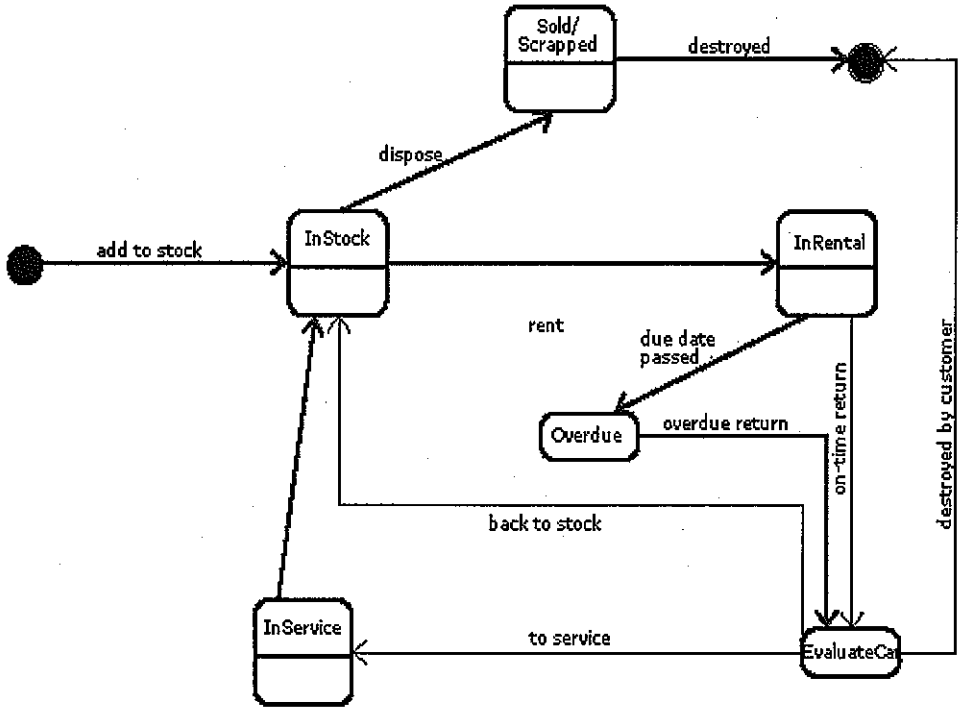


図 4.1.2.1-6 状態チャート図

⑥ アクティビティ ダイアグラム

アクティビティ ダイアグラムは、内部で生成されたアクションに応じて発生するロジックを示す。アクティビティ ダイアグラムは特定のクラスまたは使用ケースについて作成され、特定の操作を実行するためのステップを示す。

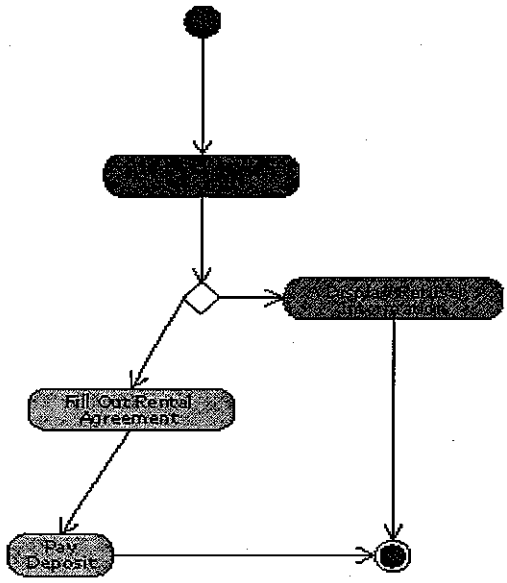


図 4.1.2.1-7　アクティビティー図

⑦ コンポーネント ダイアグラム

コンポーネント ダイアグラムは、各種のソフトウェア サブシステムがシステムの全体的な構造をどのように構成しているかを示す。このシステムは、過去のレンタカー記録、車の詳細情報、修理レコード、およびカスタマと従業員の詳細情報を含んでいる中央のデータベースの上に構築されている。在庫レベルは刻々と変化し、関係者全員が最新情報にアクセスできなくてはならないので、このデータを 1 つのデータベースに集中化することは非常に重要である。データを最新の状態に保つためには、関係者全員がリアルタイムで情報を更新する必要がある。この例のソフトウェア サブシステムには、Car Records、Service Records、Sales Records、Customer Records、および Employee Records がある。

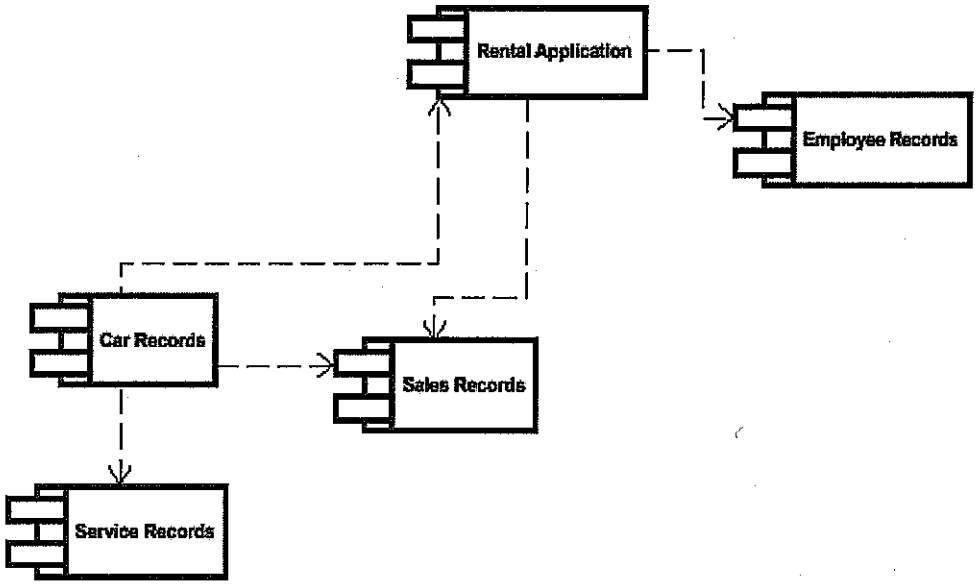


図 4.1.2.1-8　コンポーネント図

⑧ 導入ダイアグラム

導入ダイアグラムは、システム内のハードウェアとソフトウェアがどのように構成されているかを示す。レンタカー会社は、スタッフがアクセスできる、レコードが格納された中央のデータベースを使用するクライアント サーバー システムを必要としている。レンタカー会社は、車両が空いているかどうかというデータにアクセスできなくてはならない。また、メカニックは特定の車が InService 状態にあることを示すフラグを設定できなくてはならない。

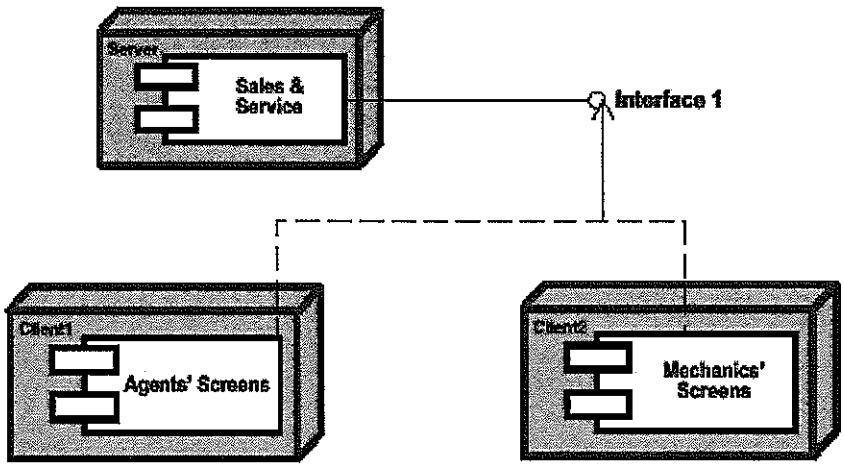


図 4.1.2.1-9　導入ダイヤグラム図

(5) UMLによるPARTSシステムのモデル化

PARTSシステムについてUMLによるオブジェクト指向分析を実施しその結果として作成された各種モデル図を以降に示す。

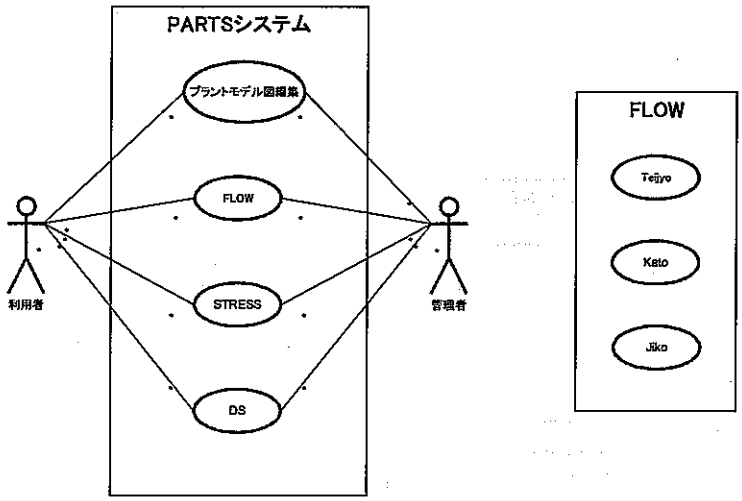


図 4.1.2.1-10　UseCase 図ーシステム全体

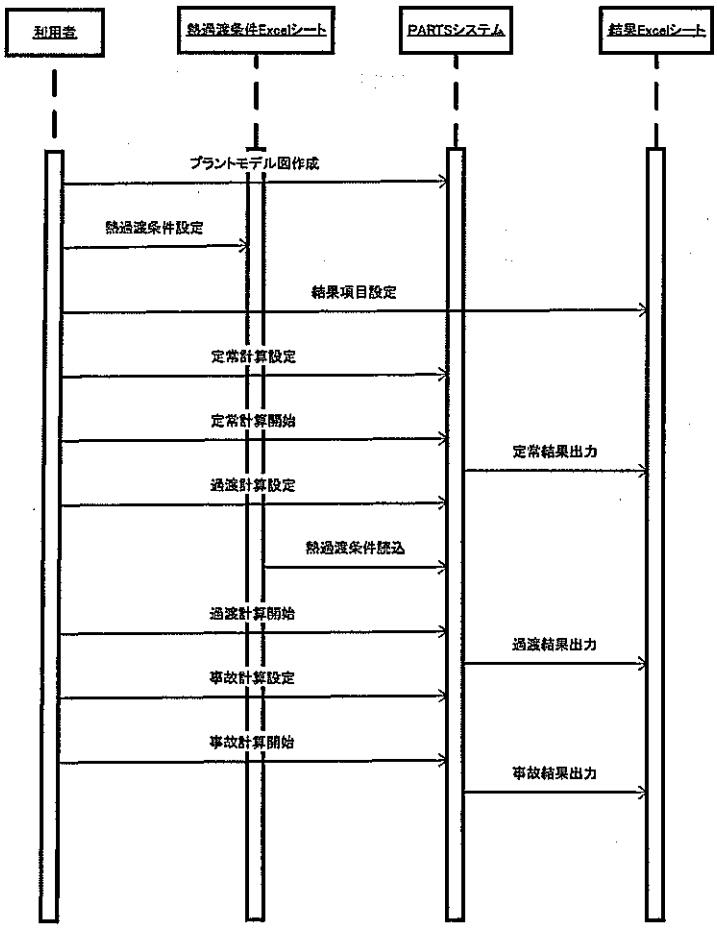


図 4.1.2.1-11　Sequence 図ーシステム全体

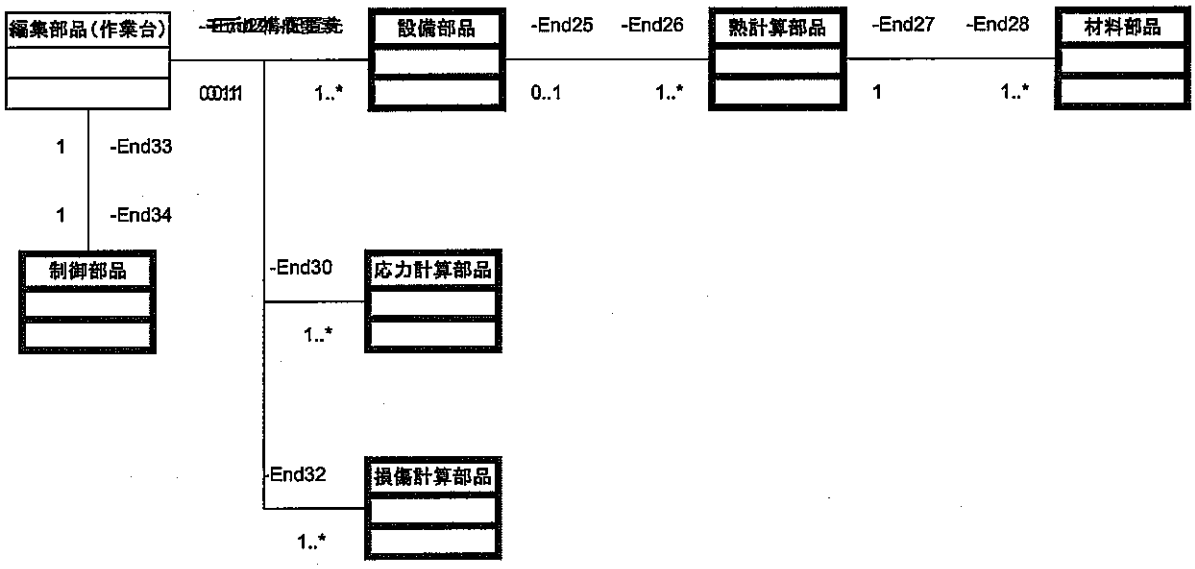


図 4.1.2.1-12　PARTS システムの静的クラス図ーシステム全体

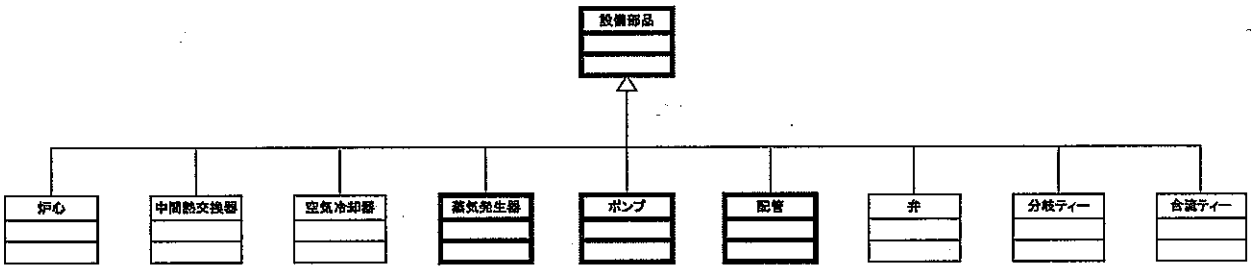


図 4.1.2.1-13　PARTS システムの静的クラス図ー設備部品

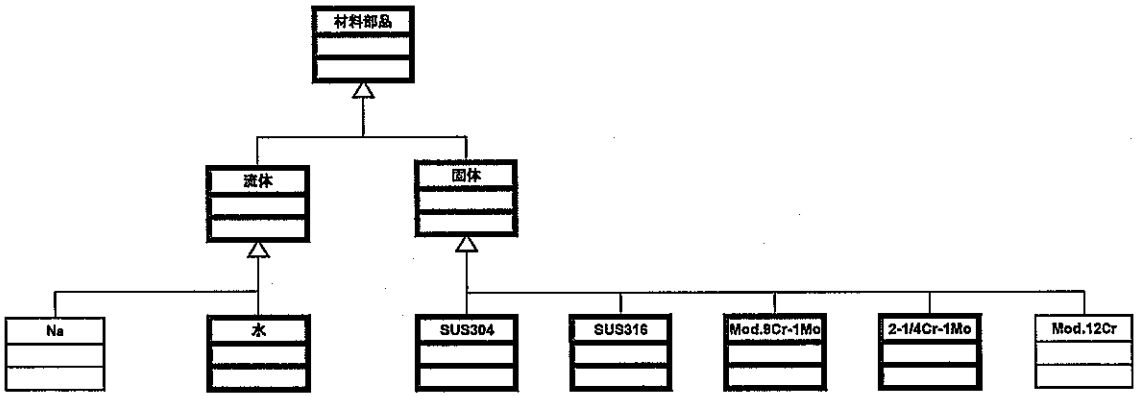


図 4.1.2.1-14　PARTS システムの静的クラス図ー材料部品

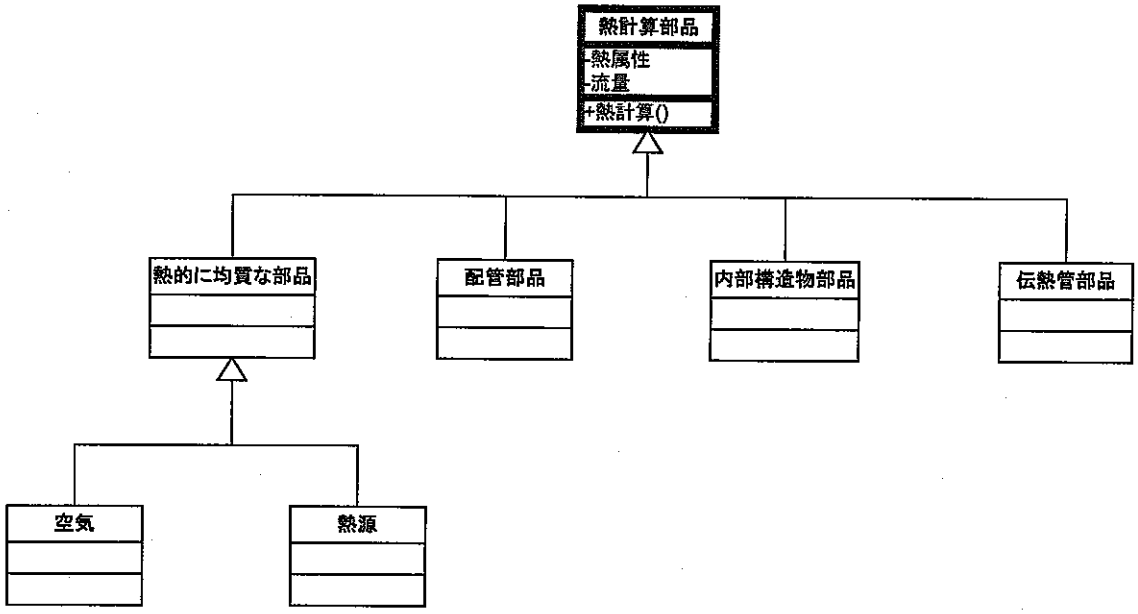


図 4.1.2.1-15　PARTS システムの静的クラス図ー熱計算部品

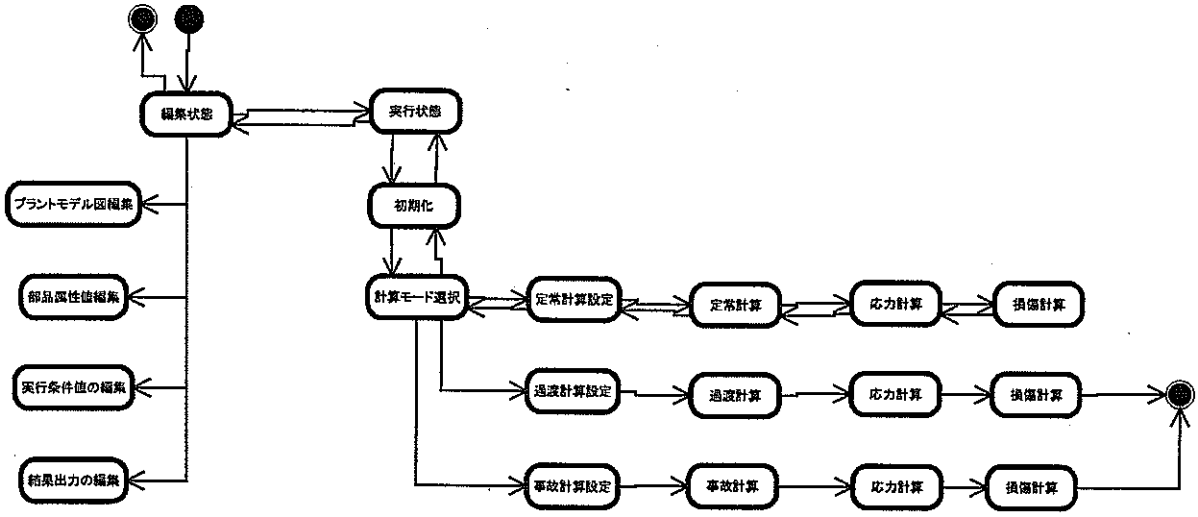


図 4.1.2.1-16　PARTS システムの StateChart 図ーシステム全体

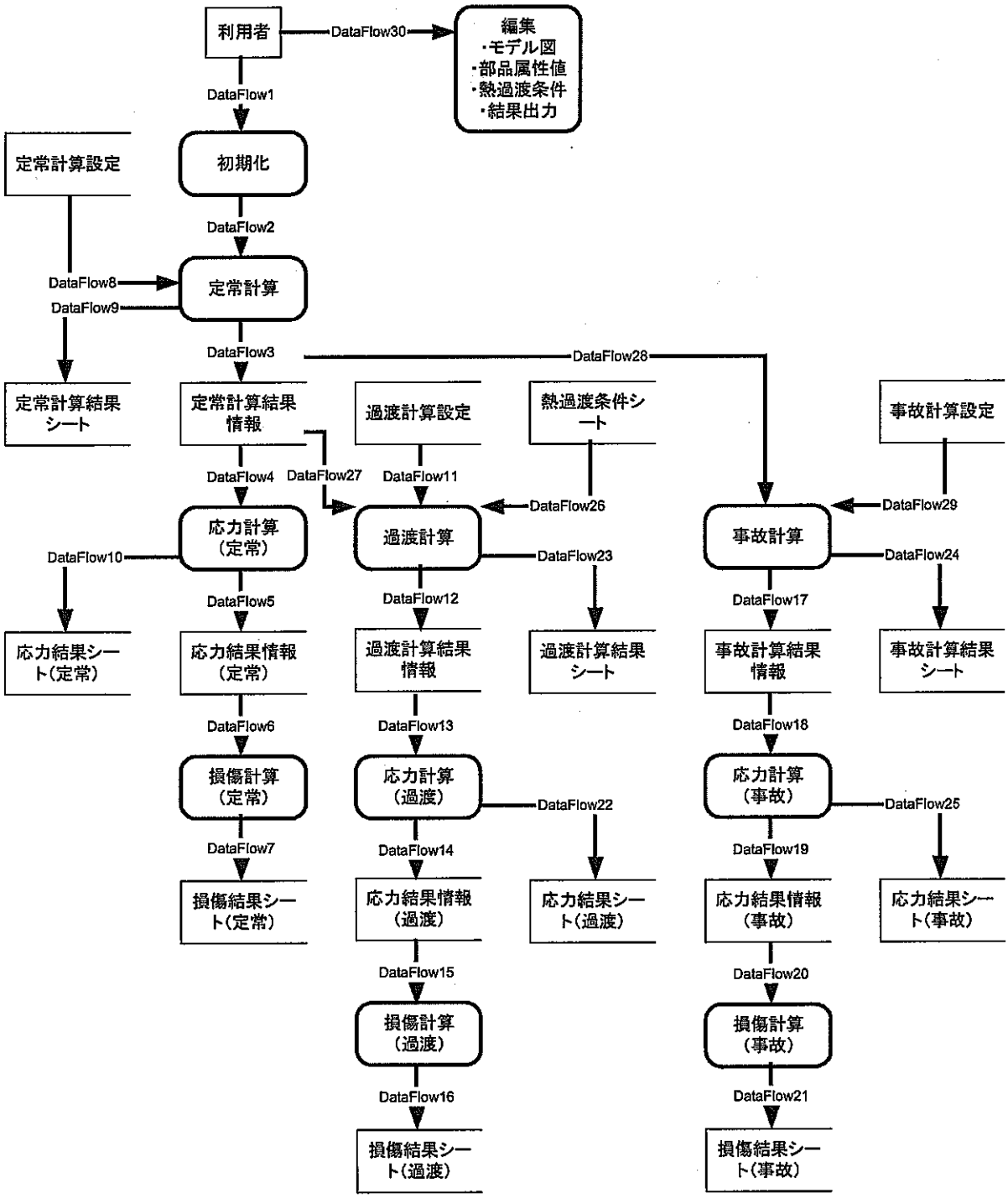


図 4.1.2.1-17　PARTS システムの DataFlow 図－システム全体

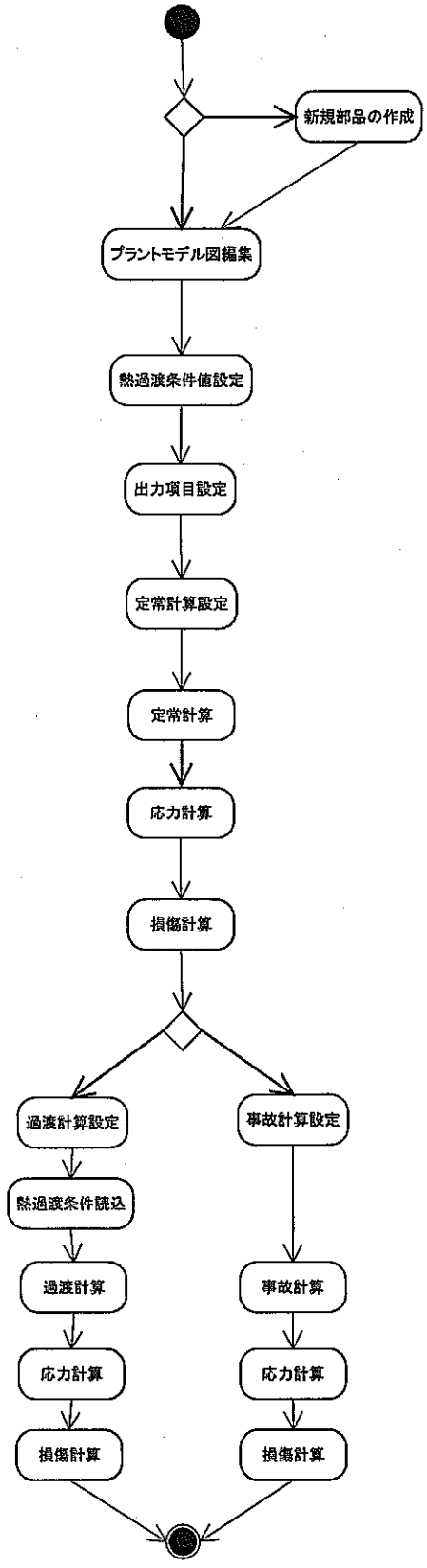


図 4.1.2.1-18 Activity 図ーシステム全

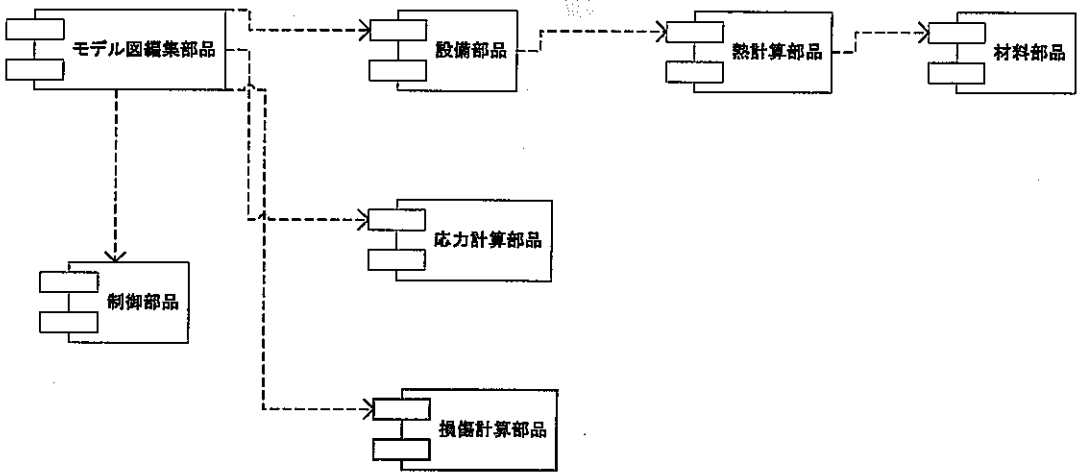


図 4.1.2.1-19　Components 図ーシステム全体

4.1.2.2 デザインパターンと開放/閉鎖原則

(1) 概要

デザインパターンの考えは、もともと建築学の分野から来た。建築家である Christopher Alexander は 40 年以上にわたって万物の型を探り続けた。彼はよい家とよい空間に繰り返される構造的な関係を示すのに言葉のパターンを採り、建築学と都市計画におけるパターンを記述した『パターン言語：Towns、Buildings、Construction』（オックスフォード大学出版部、1977）と『The Timeless Way of Building』[訳書名は「時を超えた建設の道」]（オックスフォード大学出版部、1979）の 2 冊の本を執筆した。これらの本において彼は「それぞれのパターンは我々の身のまわりで何回も起きる問題、および、それぞれの問題に対する解法のポイントを記述している。そこで我々は、これらの解法を何万回でも使うことができる。同じ問題に対する同じ解法を何度も何度も最初から考え直さずに済むというわけだ」と述べ、提示された考えはソフトウェアを含む建築学以外の多くの分野に適用できるものである。

『デザインパターン』にはソフトウェアの再利用性や柔軟性を高めるために、開発者が設計やコーディングを何回も繰り返した結果、徐々に改良されてできた解法が体系的に記されている。主にパターンは 4 つの基本的な要素（パターン名・問題・解法・結果）を記述し、種々の状況における設計上の一般的な問題解決に適用できるよう、オブジェクトやクラス間の通信を記述している。

『デザインパターン』には、オブジェクト指向システムにおいて重要でかつ繰り返し現れる技法が 23 個挙げられ、その利用により設計者が適切な設計により速く到達できるようカタログの形で表示している。

		目的		
		生成	構造	振る舞い
適用範囲	クラス	Factory Method	Adapter	Interpreter Template Method
	オブジェクト	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

図4.1.2.2-1 デザインパターンカタログ

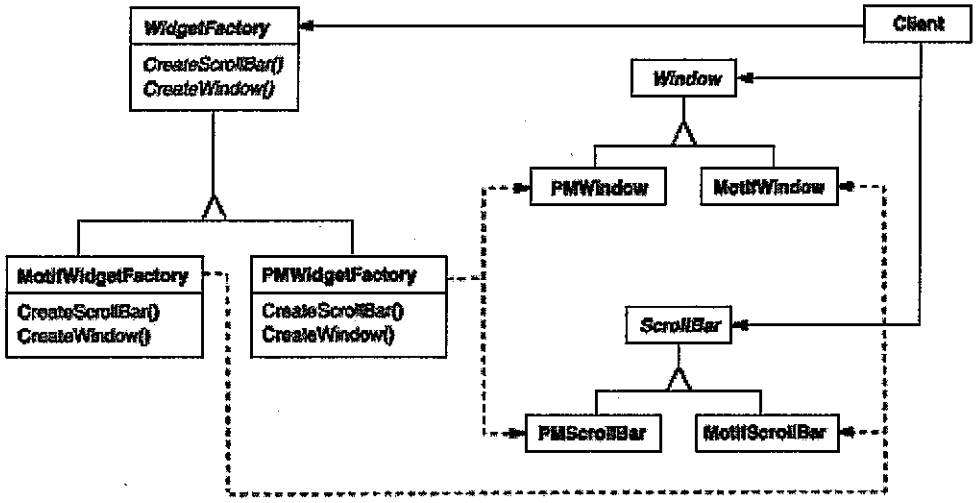


図 4.1.2.2-2　デザインパターン構造図（例：Abstract Factory）

(2)　必要性と効果

オブジェクト指向に基づいてソフトウェアを設計することは簡単なことではない。オブジェクト指向に基づいて「再利用可能な」ソフトウェアを設計することは更に難しい。設計の際には、まず適切なオブジェクトを見つけ、適切な粒度でクラスとしてまとめ、クラスのインターフェースや継承構造、そしてそれらの間の主要な関係を確立しなければならない。設計は、現在遭遇している問題に的確に対処したものであると同時に将来起こりうる問題・要求にも対処出来るものでなければならない。

デザインパターンは、問題に対しこうした諸々の要求を満たしながら解く考え方を提示している。つまり対象問題を細かく分類すれば、後はデザインパターンカタログ集の問題と比較し、似通っていればそのまま優れた解法を適用することができるのである。こうした取り組み方を「問題指向のアプローチ」という。

個々のパターンは、世界中の技術の高い設計者が時間を掛け徐々に改良されてきた解法で、さまざまな実践の場で適用し効果の検証も行われてきた。まさに先人の知恵の結晶といえる。

ちなみにデザインパターンを１つも用いていないオブジェクト指向システムを見つけることは困難であり、特に大きなシステムの場合は、デザインパターンのほとんどを用いている。

(3) 開放／閉鎖原則 (Open-Closed Principle)

システムの仕様変更やプログラムの修正に対し、効率的に対処出来るシステムとはどういうものかをここで説明する。

著名なオブジェクト指向設計者である Bertrand Meyer によれば、開放／閉鎖の原則 (Open-Closed Principle) とは次のことを意味すると説いている。

「モジュールは拡張性について開いて (Open) おり、修正について閉じて (Closed) いなければならない」

この「開放/閉鎖の原則」は、オブジェクト指向設計を考える際、その設計が正しいかどうかの指針を与えてくれるもっとも重要な原理である。あるモジュールについて、その機能を拡張できるとき、そのモジュールは「開いている」という。開いているモジュールはソフトウェアの機能追加、仕様変更に応じて異なった振る舞いをするようにできている。

モジュールは、将来どんな風に拡張されるかどうかは予想できない。したがって、そのモジュールには柔軟性 -- 開いていることが要求される。また、あるモジュールが他のモジュールから利用でき、そのソースコードを修正することが許されないときそのモジュールは「閉じている」といい、場合によってはそのモジュールはライブラリとして提供することもできる。

モジュールがひんぱんに修正されると、そのモジュールに依存している他のモジュールはその度に更新することになり、ソフトウェアが安定するためには、修正に対して閉じていることが要求される。

(4) 必要性と効果

システムの殆どは、運用後何らかの仕様変更に遭遇する。大幅な仕様変更では、修正部分が多くなり、そのためのコストがかさむのは当然のことである。では、少しの仕様変更があった場合はどうなるか？

うまく作られているソフトウェアは、この場合でも少しの変更で済むが、よくないソフトウェアは、この少しの変更が大幅な修正を引き起こす場合がある。

仕様変更があったとき、どう対応できれば一番いいのか？

バグを減らすためには、なるべく修正個所を少なく押さえる必要があるが、修正個所が広範囲にわたってしまうと、それだけでバグの可能性が多くなり、修正コストも大きくなってしまう。なるべくなら、修正個所は一個所にしぼりたいところで、共通する部分をうまく共有しているプログラムなら、このことを実現するのはそんなに難しいことではない。

でも、これですべて終わりか？ほかにもっといい方法がないのか？

それは、コードの追加である。コードの修正ではなく、コードの追加だけで対応できれば、バグを生む可能性はかなり減る。さらに、コードを修正する必要がなくなったモジュールは、再利用できるというメリットもある。

コードの修正ではなく、コードの追加で変化に対応するという方法は、従来の構造化プログラミングでは簡単には実現できなかったことである。けれどもオブジェクト指向の道具である継承とポリモルフィズムを使えば、それが実現できる。

この「機能拡張をコードの修正ではなくコードの追加によって行う」というのが、ここで述べる Open-Closed Principle にしたがうソフトウェアがもつ最大の特徴である。

具体的には、以下の手順により追加による機能拡張を実現する。

① 変更の可能性があるところを見つける 。
② その部分を抽象化してクラスにする。
③ 抽象クラスのサブクラスで変化に対応する。

4.1.2.3 オブジェクト指向開発環境

(1) 概要

開発環境とは、開発言語（C/FORTRAN/VisualBasic/Java 等）によるシステム開発を効率よく行うための支援ツール及びソフトウェア環境の総称である。

これまでのシステム開発は、データベースの設計ツール、アプリケーションのモデリングツール、開発ツール、テストツールなど異なるベンダーが提供する単機能の製品を、それぞれ使いこなし、更に連携させなければならなかった。ツール間でデータをやりとりするにも相互連携に関する仕様が異なっているため上手く連携するのは難しく、１つの製品のように使うことはできなかった。

ツール間でデータの受け渡しをするのに手間がかかるようでは開発者にストレスを与え、開発生産性も下がるばかりで、効率的とは程遠いのが現状であった。

従来は、開発・テスト工程をサポートする製品あるいは開発のみテストのみといった単機能製品が殆どであったが、全体計画・要件定義に始まりテストに至るまでの一連の作業工程をサポートした製品が望まれていた。ここにきてそれら一連の工程をサポートする製品が登場している。

大きな特徴は、上流工程の開発支援をサポートしている事であり、これによりシステム開発において最も重要な部分がサポートされ、開発効率及びソフトウェア品質の向上が期待されている。また、モデルのビジュアル化についてはGUIのようなビジュアル表示ではなく、プログラムの構造をモデル図でビジュアルに表示するという開発方法が主流なりつつある。モデル図に加えた変更がソースプログラムに反映され、さらにソースプログラムへの修正は、モデル図に自動的に反映される。

こうした開発環境（開発ツール）が目指す目標は以下のものである。

開発環境の目的

- 開発期間の短縮
- コスト削減
- ソースプログラム品質の向上 等

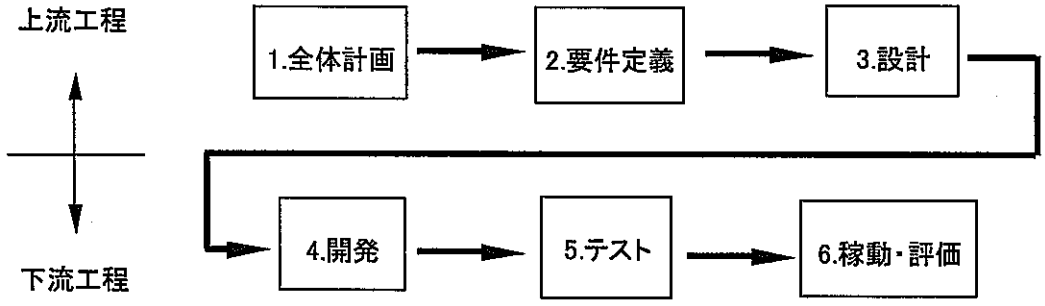


図4.1.2.3-1 システム開発における作業工程

(2) 開発環境の重要性

システム開発をより効率的に行うには、開発言語あるいは開発環境だけがあればよいというものではなく、幾つかの要素が結び付いてはじめて開発期間の短縮やコスト削減などといった目標が実現されるのである。

その必須要素を以下に示す。

① 開発方法論（開発プロセス）
② 開発言語
③ 開発環境（開発ツール）

① 開発方法論（開発プロセス）

開発プロセスは、1つのソフトウェアをどのような手順で作り上げるかの取り決めであるが、厳密にいうと、システム開発を成功へと導くプロジェクト運営の、一番基となる基本的なプロジェクトの進め方に関する基本的な考え方をまとめたものと言える。

したがって、通常、開発プロセスには、開発を構成する「アクティビティ」とその時間順序や依存関係を指定した「ワークフロー」、各アクティビティの作業内容の中で生成されるUMLモデルやドキュメントなどの「成果物」、それに作業や管理の指針となる「ガイドライン」が含まれるのが普通である。そのガイドラインに、要求のまとめ方や分析クラス図作成上の注意点、アーキテクチャ設計の技法、個々の局面でのテスト手法、コーディング規約などがまとめられているわけである。

代表的な開発方法論として以下のようなものがある。

- RUP（Rational Unified Process：ラショナル統一プロセス）
- XP（eXtreme Programming：エクストリーム・プログラミング）
- UP（Unified Process：統一プロセス）

② 開発言語

①の開発方法論（開発プロセス）を効率良く実践するには、従来の逐次型（手続き型）言語（COBOL/FORTRAN/C 等）ではなく、オブジェクト指向言語（Java / Python / VisualBasic.NET / C++ / Smalltalk 等）であることが望ましい。オブジェクト指向言語であれば、UMLモデル図により行う上流工程（分析・設計）作業の結果がそのまま下流工程に反映させることが出来るからである。つまり、モデル図からソースコードが自動生成される機能が有効活用出来る訳で、非常に効率に良い開発が実現出来るのである。

代表的なオブジェクト指向言語としては以下のようなものがある。

- Java
- VisualBasic.NET
- C++ / C#
- Smalltalk
- Python

③ 開発環境（開発ツール）

①の開発方法論を正しく理解することが何よりも重要なことであるが、その開発方法論を実践する上で欠かせないのが「開発環境（開発支援ツール）」である。

開発方法論は、リスクを解消し効率良く開発を行うための工程に関する進め方は論じているが、具体的にどういうやり方で分析あるいは設計を行い、プログラミングやデバッグ等の効率化を図るにはどうしたら良いかということについてはサポートしていない。

開発方法論そしてオブジェクト指向言語に対応しており、しかも複雑になりがちな開発手順をより単純化し簡便に操作できるようにするのが、開発支援ツールの役目である。特に求められるのが上流工程をサポートする機能で、分析/設計を支援する機能である。システムのロジックを図で表示する機能（UML モデル図による分析・設計支援）は、必須の機能となりつつある。

代表的な開発支援ツールとしては以下のようなものがある。

- Suite v2001A
- Control Center
- VisualStudio.NET
- Jbuilder

(3) 個々の開発環境製品の特徴

開発ツールは大きく２つのタイプに分類できる。１つは「統合開発ツール」で UML ツールを含んでいるタイプである。UML による分析/設計機能のほか、データベース設計やコーディングなどシステム開発のほとんどの作業をカバーする。同じ操作方法で UML をベースに上流から下流までの開発作業をおこなえることがメリットである。

このタイプの代表は、日本ラショナルソフトウェアの「Suite v2001A」で UML の分析/設計ツールとして、同社製の「Rose」を含んでいる。Rose は UML ツールの定番として地位を確立している老舗ツールである。この他の統合型ツールとしては、トゥゲザーソフトジャパンの「Control Center」、Microsoft の「VisualStudio.NET Enterprise Architect」等がある。もう１つのタイプは、開発やテストの専用ツールである。こちらは統合開発ツールに比べと安価に利用できることがメリットである。

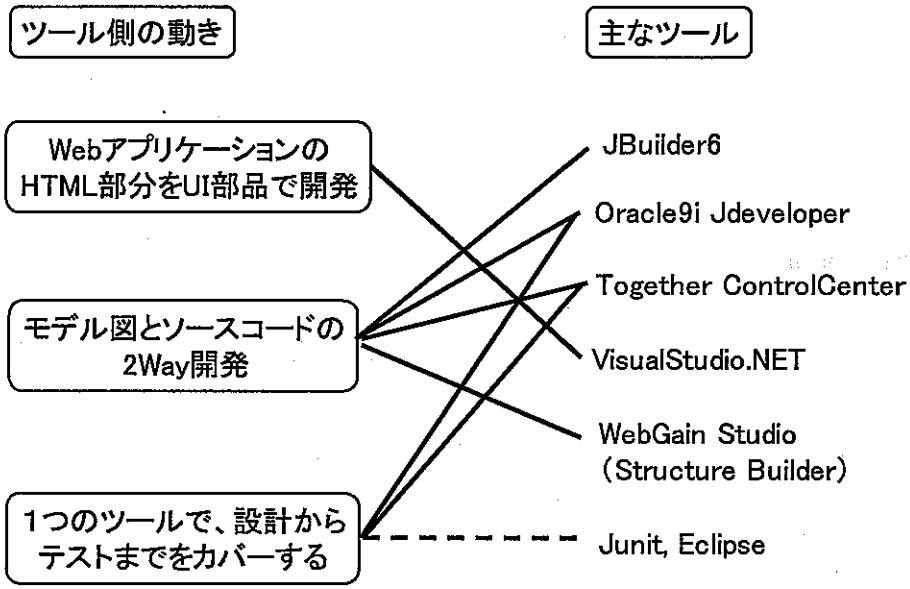


図 4.1.2.3-2　既存の設計・開発ツールにおける分類

ツール名	販売会社	主な機能				URL
		ソースコード生成	リバースエンジニアリング	レポート作成	その他	
Suite v2001A	日本ラショナルソフトウェア	C++, Java, VisualBasic	○	○ (HTMLなど)	作成したクラスを, Jbuilder VisualAgeといった開発ツールと共有できる	http://www.rational.co.jp/products/rose/
ControlCenter	トゥゲザーソフトジャパン	Java, C++, VisualBasic.NET, VisualC#.NET	○	○ (HTML, PDFなど)	図の記号などにハイパーリンクを付けて、クリックするだけで関連する図を呼び出すことが出来る	http://www.togethersoftl.co.jp/products/controlcenter/
VisualStudio.NET Enterprise Architect	マイクロソフト	Visual C++.NET, VisualBasic.NET, VisualC#.NET	○	○ (HTML, Wordなど)	UML以外にWindowsアプリケーションの画面やE-R図など、様々な図を作成できる	http://www.microsoft.com/japan/msdn/vstudio//

図 4.1.2.3-3　UML による分析/設計機能を持つ統合開発ツール

具体的には、どのような開発ツールがあるかを以降に示す。

(4) 選択する際の基準

オブジェクト指向の開発環境（開発支援ツール）を選択する際に、どういう基準を基づいて検討すれば良いかを以下に示す。

まず第一に上流工程から下流工程に至るまでの全ての開発工程をサポートされた統合開発ツールであることが前提条件となる。これは分析・設計作業時に作成したモデル図がそのまま開発・テストに反映されるということで、アーキテクチャの一貫性という観点で非常に重要である。

オブジェクト指向開発環境の選定基準

1 開発言語の選択
　Java、 Python、 VisualBasic.NET、 C++等
2 機能
2-1 上流工程（分析及び設計機能）のサポートの有無
　UML による分析・設計の支援機能
　① UML による作図機能
　② クラス図からソースプログラムを自動生成機能
　③ リバースエンジニアリング機能
　④ レポート作成機能

3 簡便な操作性
4 導入価格
5 性能

UML による分析・設計の支援機能

以降に UML による分析・設計の支援機能の具体的項目を説明する。

① UML による作図機能

UML の図を描画する機能である。UML ツールには UML を描画するための各種の記号があらかじめ用意されている。これらの記号をマウス操作で画面上に配置して、線で結んだり文字を記入したりすることで、図を作成できる。

② クラス図からソースプログラムを自動生成機能

作成したクラス図から、C++や Java といったプログラム開発言語のソースコードを自動生成する機能である。といっても、もちろんプログラムの全てが生成され

るわけではない。自動生成できるのは、メソッド名や属性名、クラス間の依存関係などの大まかな枠組みを、各プログラミング言語の文法に沿って記述した「スケルトン」にすぎない。具体的な処理は、このスケルトンを基にエディタや開発ツールなどを使ってコーディングしていく。

どのプログラミング言語のソースコードを自動生成できるかはツールによって異なるが、Java は殆どのツールが生成できる。Microsoft の「VisualStudio.NET Enterprise Architect」だけは「VisualBasic.NET」などの同社製のプログラミング言語のみ生成できる。

③ リバースエンジニアリング機能

これはソースコードの自動生成機能とは逆にソースコードから UML のクラス図を自動生成する機能である。ソースコードを修正した結果を検証するときや、既存のソースコードを解析したりするときに利用する。

④ レポート作成機能

HTML（Hyper Text Markup Language）形式や PDF（Portable Document Format）形式、Word 形式などのレポートを作成する機能である。

このレポートを印刷して、会議の資料などに利用する。多くのツールは UML の図を挿入できるだけでなく、各図中のある記号数の集計結果やコメントを挿入したり、図の変更履歴を記入したいすることが出来る。

(5) UP (Unified Process : 統一プロセス)

開発方法論(開発プロセス)には幾つかあるが、代表的なプロセス UP について説明する。

UP (統一プロセス) は、Booch 法、OMT 法、Objectory 法のそれぞれの長所を取り込み、これら各手法を統合化した開発プロセスである。従来からの開発プロセスであるウォーターフォール型プロセスとの違いは、幾つかありが最も大きな違いは「反復的に開発を繰り返す。」という点である。

システム全体の要求を一度に聞いて、一辺に設計し、更に一辺に実装するのは無理があるため、少しずつ分析し、少しずつ設計し、少しずつ実装し、というサイクルを段階的に繰り返していこうというプロセスである。こうすることで、理解できたところ、要求が固まったところから作業を前に進められ、誤りの修正も 1 サイクル以内に行えるという柔軟性が手に入る訳である。

その他 UP で開発を行うときの根本的な考え方といえる UP の特徴を以下に示す。

① ユースケース駆動
② アーキテクチャ中心
③ 反復的でインクリメンタル

① ユースケース駆動

従来オブジェクト指向開発の難しさとして、オブジェクトの世界と実際のシステムの機能が結び付きにくいということがいわれていた。オブジェクト同士のやりとりをシステム化したとしても、一体そのシステムがどのように使われるのか、そのシステムを使うと何ができるのか、というようなことが分かりにくいという面があったわけでである。そのギャップを埋めるのがユースケースである。ユースケースというのは、一言でいえばユーザーから見たシステムの使い方であり、対象のシステムを、ユーザーが何のためにどのように使うのかを表現し記述したものである。従って開発者はもちろん、ユーザーから見てもシステムがどのようなものになるのかが大変分かりやすく、相互のコミュニケーションの促進に非常に有効である。そして、このユースケースをベースに開発の進め方を考えて、要求から分析・設計、実装、テストだけでなく、プロジェクト管理といったものまでをユースケースを通じて一貫してとらえるのが、ユースケース駆動という考え方になる。

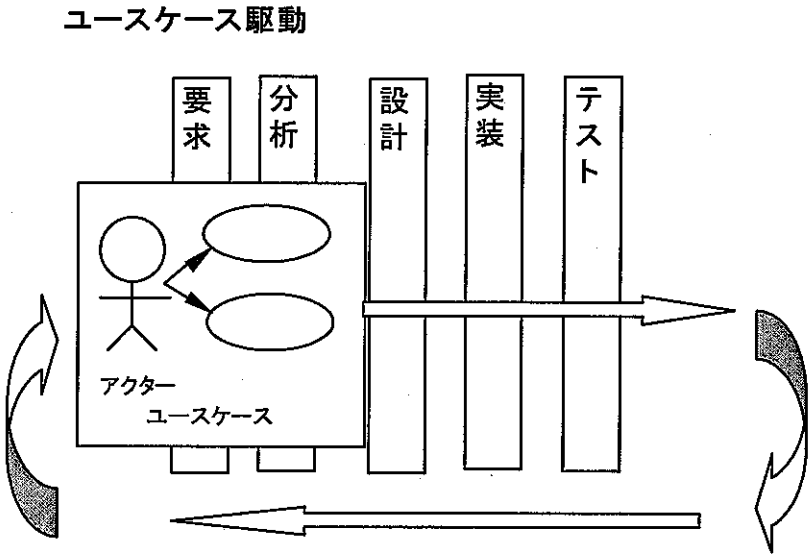


図 4.1.2.3-5　ユースケース駆動

② アーキテクチャ中心

アーキテクチャという言葉はいろいろな解釈ができてしまうが、一言でいえばシステムを支える骨組みであるといえる。ユースケースは機能を中心としたとらえ方であるが、それだけでシステムを構築することはできない。そこで、機能以外の要求、例えば性能や信頼性、拡張性などを考慮したうえで、システムの骨組みをしっかり考えて構築していこう、というのがアーキテクチャ中心という考え方である。UP ではコンポーネントベースのアーキテクチャを採用し、再利用性を重要視している。このアーキテクチャとユースケースという両面から取り組むことにより、柔軟かつ頑強なシステムを構築することができるわけである。

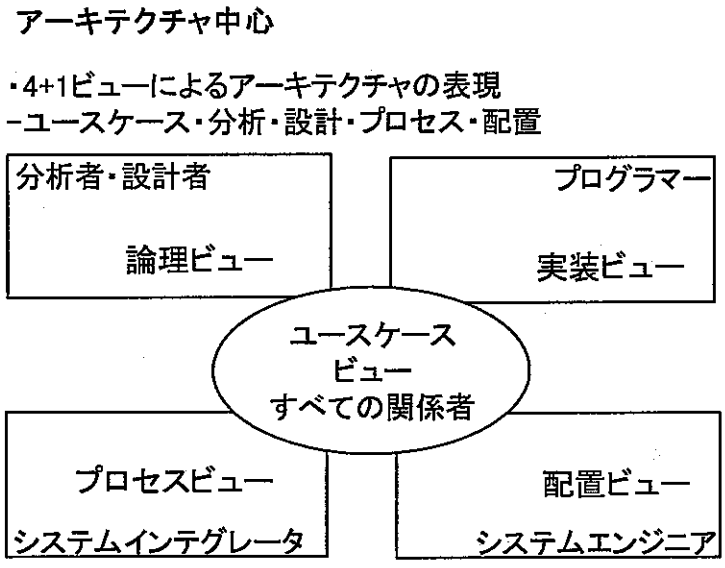


図 4.1.2.3-6 アーキテクチャ中心

③ 反復的でインクリメンタル

反復型開発というのは、小さなリリースを繰り返しながら全体的なシステムを構築していくというアプローチである。この小さなリリースのたびに統合作業が繰り返され何度も検証されるため、最後になってから致命的な問題が発覚するという可能性は極めて小さくなる。このリリースのことを反復（Iteration：イテレーション）と呼ぶ。ただし、有効に思えるこの反復開発も、ただ無計画に反復させているだけでは決して機能しないだけでなく、逆に混とんとした状況を招いてしまう危険がある。そこで、UP ではこの反復を重要な管理対象として扱い、開発のライフサイクルを支えるキモとして評価しているのが特徴である。つまり、エンジニアリングステージと製造ステージをさらに管理しやすく分割したフェーズと、フェーズおよびそれぞれの反復の主要な目標であるマイルストーンという概念を用いて、プロジェクト管理的にも管理可能な単位として反復を位置付けているのである

反復的でインクリメント

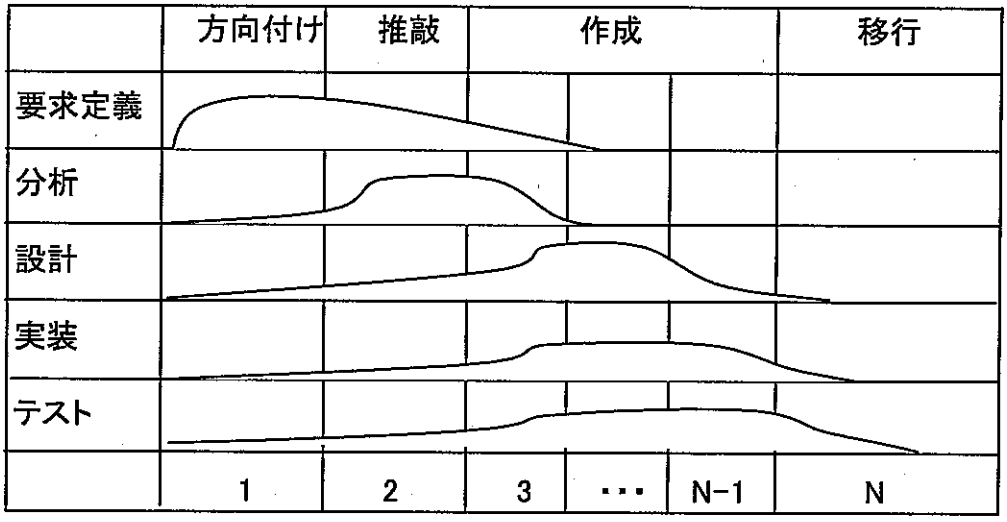


図 4.1.2.3-7 反復的な開発サイクル

4.1.3 最近のプログラミング言語に共通する特徴

4.1.3.1 異言語間結合

(1) 共通言語ランタイム

(a) 概要

マイクロソフトが提供するソフトウェア・フレームワークである.NET Framework 上で、アプリケーションやサービスを動作させるための実行エンジンで共通言語ランタイム（CLR: Common Language Runtime）は、コードの実行を管理し、アプリケーションに対してさまざまなサービスを提供する。Java で言うところの Java 仮想マシン（Java VM）に相当する。従来の Windows 環境では、アプリケーションから実行時に呼び出されるランタイム・ライブラリとして、MFC ランタイム・ライブラリや Visual Basic ランタイム・ライブラリ、ATL（Active Template Library）などのように、使用する言語やプログラミング・モデルごとに独立したランタイム環境が提供されていたが、CLR はその名が示すように共通の言語ランタイムであり、.NET をサポートする各言語のコンパイラによって生成された実行ファイルはすべて CLR 上で実行される。

CLR 上で実行されるコードは、マネージド・コードと呼ばれ、CLR により完全に管理されて実行される（これに対し、CLR では管理されない従来のものは、「アンマネージド・コード」と呼ばれる）。.NET をサポートする各言語コンパイラは、すべてマネージド・コードを生成するため、プログラマは任意の言語でアプリケーションを記述し、それらを組み合わせて使用することができる。CLR に準拠したマネージド・コードでは、言語間でのクラスの継承やメソッドの呼び出し、言語間での例外処理などが可能となっている。また.NET Framework のクラス・ライブラリもマネージド・コードで構成されているため、どの言語からも共通して使用することができる。

CLR によるアプリケーション実行のおおまかな流れは次のようになる。

アプリケーションが実行されると、まず CLR のクラス・ローダによってアプリケーションのコンポーネントがメモリ上にロードされ、配置される。このときにはコードがタイプ・セーフかどうかのチェックも行われる。誤ったメモリ領域への参照や型が一致しないパラメータの受け渡しなどを含むタイプ・セーフでないコードは実行されない。

.NET のアプリケーションの実行ファイルは、すべて MSIL（Microsoft Intermediate Language）と呼ばれる中間コードで構成されており、これは実行時に CLR の JIT コンパイラによりネイティブ・コードに変換される。ただしこの場合でも、アプリケーションのすべての部分がコンパイルされるわけではなく、メソッドの呼び出しに応じて必要な個所が効率的にコンパイルされ、実行される。

アプリケーションの実行中は、オブジェクトのライフタイム管理や、オブジェクトへの参照を管理することによってガーベジ・コレクションが行われる。また、システム・リソ

ースや他のプログラム・コードへのアクセスに対するセキュリティのチェックも行われる。
これら以外にもCLRは、開発サポートとしてデバッグやプロファイルなどのサービスも提供する。

(b) CLRが提供する主なサービス

CLRは以下のようなサービスを提供している。

- コード管理（ロードと実行）
- アプリケーション メモリの隔離
- タイプ セーフティの検証
- IL からネイティブ コードへの変換
- メタデータへのアクセス（拡張された型情報）
- マネージド オブジェクトのメモリの管理
- コード アクセス セキュリティの実施
- クロス言語例外を含む例外処理
- マネージド コード、COM オブジェクト、および既存の DLL（アンマネージドのコードとデータ）の間の相互運用
- オブジェクト レイアウトのオートメーション
- 開発者サービスのサポート（プロファイリング、デバッグなど）

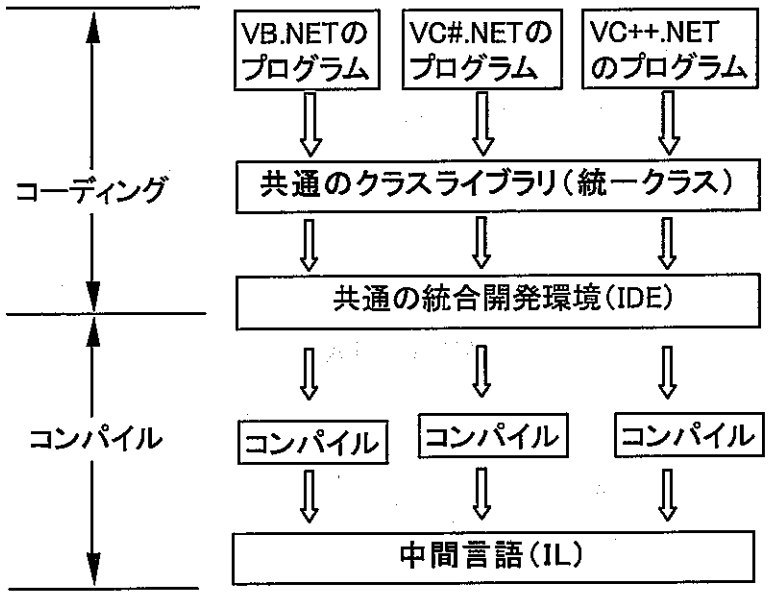


図 4.1.3.1−1　VisualStudio.NET での開発の流れ

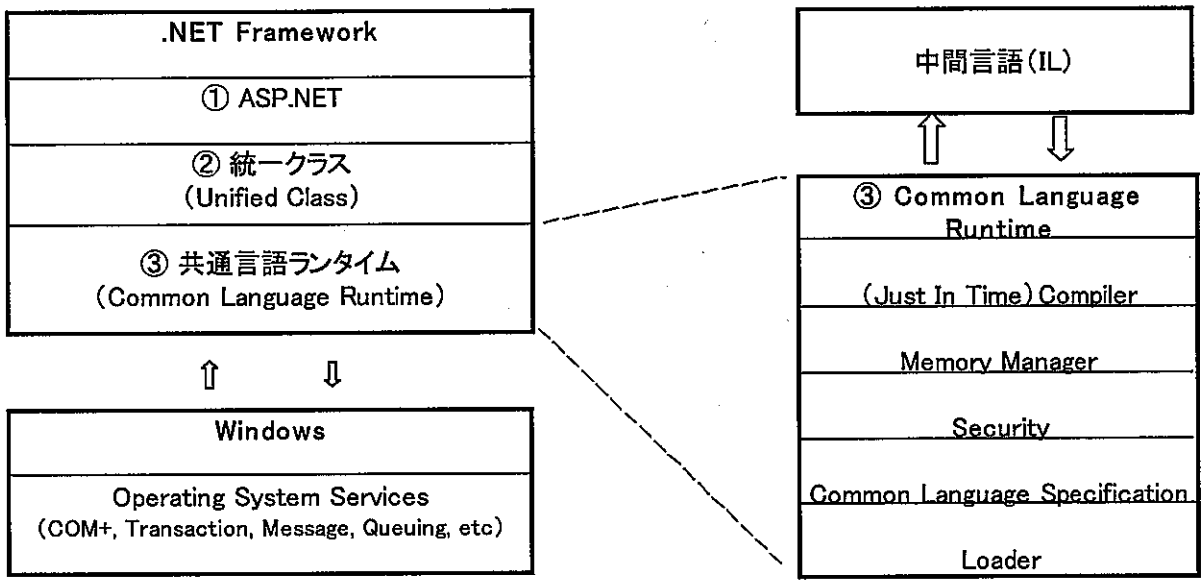


図 4.1.3.1−2 アプリケーションから見た.NET Framework 及び
共通言語ランタイム（CLR）の内部構成

(2) Java による他言語実装

Java はその汎用性と柔軟性という特徴を生かし、多くの言語が Java により実装されている。その中でもはじめに近年注目を集めている、オブジェクト指向型スクリプト言語 Python を実装した Jython を中心に取り上げる。

(a) Jython

Jythonとは、JVM(Java Virtual Machine)上で動作するスクリプト言語である。Jythonは、Pythonというスクリプト言語の処理系の一つで、Jythonの処理系はJavaで記述されている。Jythonを使うと以下のようなメリットがある。

- Javaと違ってコンパイルする必要がない。
  「ちょっと書いて実行」というRAD(Rapid Application Development)開発に有利。
- 「擬似コードを使って設計するのに近い感覚」で実装できる。
  シンプルなオブジェクト指向スクリプト言語のため、同じ処理内容を記述するときに、Javaより楽に直感的に記述できる。
- JythonからJava、JavaからJythonを利用できる
  Javaのクラスのテストを Jythonで記述できる。アプリケーションのカスタマイズ部分をJyhtonで記述できる。

例えば、システム内の安定した部分にはJavaを使い、ホットスポット*1と呼ばれる部分を、Jythonスクリプトを使って記述することができる。スクリプト言語は変更するためのコストが低いので、このような使い分けは効率がよいはずである。言語の特性に応じた、適材適所な使い方ができれば、システム全体の設計がよりシンプルになり、開発効率が上がると思われる。

また、JythonはJavaと同じオブジェクト指向言語である。そのためシステム内のJava部分の設計と、スクリプト部分の設計のインピーダンス*2のミスマッチが少なく、より設計しやすくなるという利点がある。

---

*1 システムやフレームワークの中で、要求が変更されたり追加されたりすることが、あらかじめ予想される部分。ホットスポット部分は変更に対して柔軟に対処できるように、最初から意識して設計する。

*2 オブジェクト指向の分野では、「オジェクト指向で設計された部分」と「それ以外の方法で設計された部分」の整合性が、うまく保たれない場合に使う場合が多い。例えばRDBのモデルとオブジェクト指向モデル間のインピーダンスのミスマッチがよく話題になる。

JythonはPython言語(と、そのC言語で書かれたインタープリタ)の進歩に追随するように開発が進んできた。Python 1.6まではJPythonという名前であったが、Python 2.0の登場に合わせて、Jythonと名前を変えた新プロジェクトとなった。現在の総本山はhttp://www.jython.orgで、このサイトから最新のバージョン2.1が無償で入手できる。リリース履歴は以下の通りである。

- 2.1 ... 2001/12/31
- 2.1b2 ... 2001/12/21
- 2.1b1 ... 2001/12/03
- 2.1a3 ... 2001/07/28
- 2.1a2 ... 2001/07/17
- 2.1a1 ... 2001/03/13
- 2.0 ... 2001/01/16

なお、Jythonは100%純粋なJava言語で書かれているため、MS-WindowsでもUNIXでも同じように使うことができる。

(b) JythonとJDBC*3

JDBCは、Javaからデータベースに接続するためのAPIである。JythonでJDBCを使うには、本家JavaでJDBCを使う方法を知っていることが前提となる。JavaでJDBCを扱う方法を知っていれば、そのままPythonで書き下すだけでJython上で動く。

また、JDBCテクノロジは、広範なSQLデータベースへのDBMSの接続性を提供するだけでなく、現在では新しいJDBC APIを使用して、スプレッドシートやフラットファイルなどの他の表形式データソースへのアクセス手段も提供する。

JDBC APIを使用すると、開発者は企業データへのアクセスを必要とする強力な、異なるプラットフォーム間で動作するアプリケーションに対して、「Write Once、Run Anywhere™(一度記述すればどこででも実行可能)」というJavaプラットフォームの特色を利用することができる。JDBCテクノロジ対応ドライバを使用すると、開発者は異機種システム混在環境のすべての企業データに簡単に接続できる。

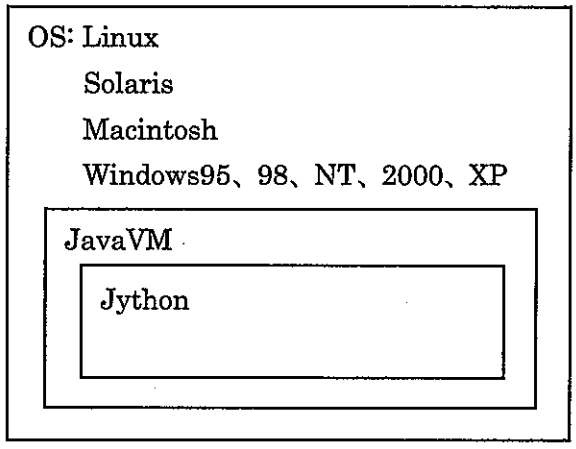
---

*3 一般的に、Java Database Connectivityの略と言われているが、JavaSoftの資料には「JDBCはトレードマーク(TM)であって、何かの略語ではない」と書かれている。

(c) Jython 動作環境

Jython の動作環境は、以下のとおりである。

- Linux
- Solaris
- Macintosh*4
- Windows95、98、NT、2000、XP



(d) JavaVM 上で動く Java 以外の言語

JavaVMはJavaをコンパイルして生成されたバイトコードを解釈して動作を行っている。ここでよく考えると、バイトコードさえきちんと生成できれば、元の言語は何でもいいのではないか、というアイディアが浮かんでくる。マイクロソフトの.NET も基本的には同じ考え方である。.NET ではいろいろな言語が使えることを強調しているが、結局バイトコードにコンパイルして実行する点は、Java と大体一緒である。

参考までに、以下の URL には、Java 処理系で動く、さまざまな言語がコレクションされている。

http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html

---

*4 Macintosh には、以下のものが必要となる。

JPython Runnre for Macintosh：マッキントッシュ Java プラットフォーム上の JPython クラスあるいはスクリプト・ファイルを実行するためのラウンチャ・プログラム

例：

Basic	JBasic、JavaBasic 等
smalltalk	smalltalkJVM、Talks2、Bistro
COBOL	PERCobol
Scripting	JRuby、Jython 等

(3) SWIG

SWIG (Simplified Wrapper and Interface Generator)、Tcl、Perl、Python 等のスクリプト言語から、C、C++、Objective-C 等のシステム言語を結合するためのツールである。もともと、SWIG は、米国 Los Alamos 国立研究所の理論物理の分野で開発されたシステムであり、複数台のスーパーコンピュータで動作している物質科学の統合シミュレーションを行うためのインターフェイス作成用に開発されたものである。

Tcl、Perl、Python 等のスクリプト言語自体は C 言語で書かれており、C 言語で書かれたプログラムを結合するインターフェイスを独自に持っている。このため、SWIG を使わなくても Python 等のスクリプト言語と C 言語を結合する方法はある。しかしながら、この方法は、スクリプト言語の開発にかかわるものであるため、単なるスクリプト言語のユーザーには使いこなすのが難しい。SWIG は、このような異言語結合の作業をなるべく簡単に行うために開発されたツールである。SWIG は、多くのスクリプト言語に対応しており、ドキュメントもよく整備されている。

SWIG の主要開発者である Beazley 氏によると、C 言語によるプログラミングは次の部分の組み合わせで成り立っているとされる。

- いろいろな関数と変数の集合
- プログラムを開始するための main() (メインルーチン)
- プログラムを適切に使えるようにするための各種の工夫

現実には、本質的な部分である 1 よりも 2 や 3 が複雑になって扱いきれなくなることが多い。1 の部分だけを C 言語で実装し、残りはスクリプト言語を使うことにすれば、プログラムは、

- いろいろな関数と変数の集合
- それを使うためのスクリプト言語

という構成になり、プログラムが格段に使いやすくなると主張している。

また、現在のプログラミング言語では状況に応じた言語の使い分けが難しいので、通常は、1 つの言語ですべての処理をおこなうことが多い。しかしながら、例えば、以下のように目的に応じて、

- 数値計算は Fortran
- システム関数は C 言語
- 3 次元表示は OpenGL (C 言語で書かれたライブラリ)
- パターンマッチやテキスト処理は Perl
- 簡単な GUI は Tcl/Tk

のように、プログラミング言語を選択することが可能であれば、システム開発はかなり楽になると考えられる。SWIG は、Fortran には対応していないが、Fortran と C 言語との結合は可能であるので、上記の言語を SWIG を利用して結合することが可能になる。

以下では、SWIG がどのようなツールであるかを示すために、実際に簡単なサンプルを用いて行った SWIG の利用テストを紹介する。なお、SWIG は Unix だけではなく、Windows にも対応しており、Windows 用の実行ファイルも配布されている。ここでは、Linux と Windows 上で同様のテストを行った。スクリプト言語としては、Python を利用した。

(a) SWIG のテスト

SWIG についても、配付されているパッケージに含まれるサンプルのうち、最も簡単なものについてテストを行った。ここで結合する C 言語のプログラムは、初期値が 3.0 にセットされた Foo という大域変数と、最大公約数を求める gcd という関数を含む、example.c というファイル(図 4.1.3-2)である。この C 言語のプログラムから、example という Python の拡張モジュールを作成する。

SWIG では、ユーザーは拡張子「.i」を持ったファイル（図 4.1.3-3）を別途用意する必要があり、f2py の「.pyf」拡張子ファイルのように自動生成はされない。しかしながら、作成したい拡張ライブラリーのモジュール名やデータ型を定義するだけのファイルであり、ユーザー自ら定義せざるを得ない最低限の情報で済むようになっている。Python に C 言語のプログラムを結合するために作成しなければならないのはこのファイルだけであり、後は、SWIG が自動的にソースファイルを分析して処理してくれる。

以下に、SWIG のテストとして行った手順をまとめる。以下の手順は、Linux 上でのものであるが、基本的に Winodws での作業も、(2)と(3)で、Linux 上では GNU C コンパイラ（gcc）であるものが、Microsoft Visual C++（MSVC++）コンパイラになるだけで、Windows でも同じである。もちろん、MSVC++での作業は、通常どおり Windows の GUI 環境で作業することが可能である。

なお、本文でも記したように、SWIG を利用するためには、ユーザーは example.i というファイルを作成する必要がある。以下は、example.i 作成後の作業手順である。ちなみに、添付されていたファイルには Makefile があるため、実際には、make コマンドを実行するだけで拡張ライブラリーが作成されるが、以下では内部で実行されている手順を詳しく述べる。

① 拡張ライブラリーのラッピング情報の作成

swig コマンドを実行して、インターフェイス情報（図 4.1.3-3 の example.i）からラッピング情報を作成するために以下のコマンドを実行する。

```
% swig -python example.i
```

② C言語のソースファイルのコンパイル

拡張ライブラリーにしたいC言語のソースファイルであるexample.c（図4.1.3-2）をコンパイルする。

```
% gcc -c example.c
```

SWIGが生成したラッピング情報（example_wrap.c）をコンパイルしてオブジェクトファイルを作成する。なお、このとき、Pythonのincludeファイルを参照する必要があるため、Pythonのライブラリーへのパスが通っている必要がある。

```
% gcc -c example_wrap.c -DHAVE_CONFIG_H ¥
    -I/usr/local/include/python2.2 ¥
    -I/usr/local/lib/python2.2/config
```

③ 拡張ライブラリーの作成

作成したオブジェクトファイル（example.o、example_wrap.o）をリンクして、拡張ライブラリーを作成する。

```
% ld -shared example.o example_wrap.o -o examplemodule.so
```

④ 拡張ライブラリーの利用

作成した拡張ライブラリを利用するためには、Pythonで、「import foobar」とするだけでよい。これは、Pythonで書かれたライブラリーと全く同じ扱いであり、基本的にユーザーはC言語で書かれていることは意識する必要がない。図4.1.3-1に実行例を示す。「>>>」はPythonのインタラクティブモードでのプロンプトであり、「>>>」の行が実際に打ち込んだコマンドである。プログラムとして実行させる場合も同様に書けばよい。

基本的な作業の流れは、Linuxであっても、Windowsであっても同じである。以下、異なる点を簡単に説明する。

(b) Linuxでのテスト

Linux上では、SWIGの出力としては、examplemodule.soというファイルが得られる。このC言語で作成されたモジュールは、通常のPythonのモジュールと同様に、「import example」とするだけで利用可能となる。

(c) Windows でのテスト

SWIG は、Microsoft Visual C++（MSVC++）に対応しており、MSVC++でコンパイルしたライブラリーを Windows 上の Python から利用することが可能である。Linux と同じサンプルでテストを行い、同様の結果が得られることを確認した（図 4.1.3-4）。

SWIG のコマンド自体は、Windows の DOS プロンプトを用いて、Unix の Shell 上での作業と同じようにして実行する。この後は、MSVC++を使って、SWIG が作成したファイルとともにコンパイルするだけでよい。なお、当然のことながら、Windows の場合は、最終的な出力として、Windows に対応した DLL ファイル（ここでは、example.DLL）が生成される。

作成した拡張ライブラリーを Python から使うときは、「import example」とすればこの DLL ファイルが読み込まれるようになっており、利用方法は、Linux の場合とまったく同じである。

```
% python
Python 2.2.1c2 (#1、 Apr  4 2002、 13:07:25)
[GCC 2.96 20000731 (Red Hat Linux 7.1 2.96-98)] on linux2
Type "help"、 "copyright"、 "credits" or "license" for more information.
>>> import example                    #拡張ライブラリーの読み込み
>>> example.gcd(42、105)                      #42と105の最大公約数を計算
21
>>> example.cvar.Foo                  #大域変数Fooの内容確認
3.0
>>> example.cvar.Foo = 3.141592654    #大域変数Fooの内容を変更
>>> example.cvar.Foo
3.1415926540000001                    #大域変数Fooの変更成功
```

図 4.1.3.1-3　作成した example 拡張モジュールの利用例

```
/* File : example.c */

/* A global variable */
double Foo = 3.0;

/* Compute the greatest common divisor of positive integers */
int gcd(int x、 int y) {
  int g;
  g = y;
  while (x > 0) {
    g = x;
    x = y % x;
    y = g;
  }
  return g;
}
```

図 4.1.3.1-4　example.c のプログラム
（大域変数 Foo と最大公約数を求める関数 gcd を含む）

```
/* File : example.i */
%module example

extern int    gcd(int x、 int y);
extern double Foo;
```

図 4.1.3.1-5　example.i のソース
(example.c をコンパイルするためのユーザーが作成するインターフェイス情報)

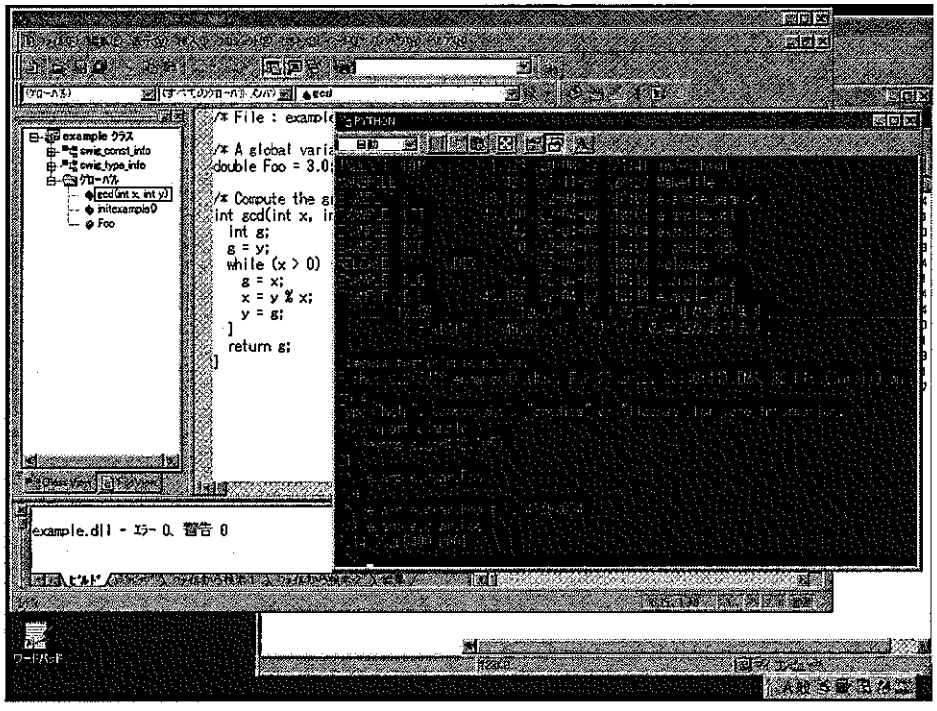


図 4.1.3.1-6　Windows 上での SWIG のテスト

手前の DOS プロンプトで python を起動し拡張ライブラリーexample を利用している。後ろに見える Microsoft Visual C++でコンパイルして example.dll を作成した。

4.1.3.2 内部ドキュメント化

解析コードシステムにおいて、ドキュメントの整備は、品質保証の観点からも重要な課題である。通常、解析コードが開発された後に、紙に印刷された報告書の形でマニュアルが整備されることが多い。しかしながら、解析コードは開発された後も、常に機能拡張やエラー修正に伴った変更が行われ続けるため、このような文書はすぐに古くなるという問題がある。しかしながら、報告書を電子ファイル化するだけでは、最新のマニュアルを常に整備できるとは限らないのが現状である。

このような問題は解析コードに限らず、一般のソフトウェア開発では問題となることが多い。このような問題に対応する方策として、C 言語の開発者であるカーニハンは「文芸的プログラミング」と呼ばれる方法を提唱している。このような概念は、さまざまな方法で取り入れられている。

(1) 炉物理解析コードにおける内部文書化

第２章で紹介したように、CCCC フォーマットでは、プログラムのコメントとして、サブルーチン毎にドキュメントを挿入するという方針が採られている。一方、欧州炉物理解析システム ERANOS でも、同様の方針が採られているようであり、文献によるとプログラムの中にマニュアルが書き込まれているとの記述がある。更に、CCCC より一歩進んで、OPALE と呼ばれるプログラム解析器が用意されており、プログラム中に書き込まれた文書を取り出したり、データ構造（SET）を処理したりして、自動的にマニュアルを生成することが可能のようである。このように、マニュアルとプログラムを同じファイルに管理しておき、マニュアルの自動生成を行うことで、解析システムに関するマニュアルとプログラムを、一元的に管理することが可能となる。このため、プログラムの修正時に、すぐそばに書かれているマニュアル用のテキストを改訂することさえ忘れなければ、常に最新の文書を提供することが可能となる。また、ERANOS の文献では、このように文書を管理することで、モジュラー化された計算機能が持つ階層構造と同じように、マニュアルも階層構造を持つことができるという利点も指摘している。

(2) プログラミング言語における内部文書処理機能

上述のような文書管理方法は、最近のソフトウェア開発手法では、一般的になっているようであり、いくつかのオブジェクト指向プログラミング言語では、言語の基本機能としてこのような文書管理機能を持っているものがある。最近のプログラミング言語では、単なるコメントのレベルではなく、後からマニュアルを自動生成することを目的として、プログラムソースの中にマニュアルを書き込むことをサポートしているものが多い。例えば、Java 言語では javadoc、Ruby 言語では RDtool、Python 言語では、PyDoc、HappyDoc と呼ばれるツールが整備されている。

ここでは、Python 言語を例にとって、プログラミング言語が備える内部文書処理機能について述べる。Python 言語では、プログラミング言語の基本機能として、内部文書化の機能を備えており、クラスやメソッドの宣言の次にはクラスやメソッドを説明するための文字列を挿入することが可能である。これらのコメント文は、__doc__という名前の特殊な変数に収められており、Python プログラム中でもこの情報を利用することが可能となっており、Python 言語は自らが自分自身のマニュアルを処理することができる。

図 4.1.3.2-1 には、対話形式での利用方法の例を示す。ここでは、文字列オブジェクト a が持つメソッド upper に関する説明を表示させている。ユーザーは、Python のマニュアルを探さなくても、__doc__が持つ情報を見ることで、upper というメソッドが、すべて大文字に変換した文字列オブジェクトのコピーを返すことが分かる。

また、Python 言語には、PyDoc と呼ばれる機能が標準で添付されており、Python のプログラム中にに記述された文書を自動的に集積、整形して表示する機能も持っている。図 4.1.3.2-2 には、Python の標準モジュールである pickle モジュールに関するマニュアルを自動生成した結果を示す。実際のプログラム中では、クラスやメソッドの実装部分があるので、埋め込まれた文書は、分散されて格納されているが、PyDoc により、ひとまとめにして表示してくれる。

また、HappyDoc と呼ばれる機能も整備されており、PyDoc がオンラインマニュアル的な利用を前提としているのに対して、HappyDoc では、きちんとまとめられたマニュアルを生成することを目的としている。HappyDoc を使うことにより、モジュールやクラスの収められたディレクトリを自動的に捜査して、モジュールやクラスの階層構造と同様の階層構造を持った HTML 形式のファイルを自動生成することが可能である。

(3) 解析システムへの内部文書機能の応用

このような機能を利用することにより、解析システムに自己記述的な特徴を持たせることが可能になると考えられる。

図 4.1.3.2-3 には、独自に作成した Python プログラム（核種を表すクラス Nuclide）に対して、説明文を埋め込んだ例である。このソースファイルを PyDoc で処理して作成したマニュアルを図 4.1.3.2-4 に示す。また、図 4.1.3.2-5 には、HappyDoc で処理して作成した HTML 形式のマニュアルを示す。この例では、プログラム量が非常に小さいので、このようなことをする利点はあまり感じられないかもしれないが、HappyDoc を使えば、複雑な階層構造を持ったクラスを定義した場合にも、その階層構造と同じ構造を持った HTML 形式のマニュアルを生成することが可能であるので、ユーザーはシステムの構造を理解しやすくなると考えられる。

Python 言語に限らず、最近のプログラミング言語が持つ内部文書処理の機能を用いることにより、ドキュメントとプログラムソースを一元管理し、プログラムの開発とドキュメント作成を同時進行しやすく、かつ、マニュアル作成のコストを低く抑えられる可能性がある。また、異なる分野の専門家が解析システムを共同開発するためには、すべての分野

の詳細を熟知することは不可能であるので、各々が必要に応じて、各モジュールのインターフェイス部分の最新情報だけを自由に確実に取り出せるような工夫が必要である。複数の開発者による共同開発を効率よく進めるためにも、このような内部文書処理機能等を有効活用する必要があると考えられる。

```
>>> a = "abcdefg"
>>> print a.upper.__doc__
S.upper() -> string

Return a copy of the string S converted to uppercase.
```

図 4.1.3.2-1　Python における内部文書の例

```
Python Library Documentation: module pickle

NAME
    pickle - Create portable serialized representations of Python objects.

FILE
    /usr/local/lib/python2.2/pickle.py

DESCRIPTION
    See module cPickle for a (much) faster implementation.
    See module copy_reg for a mechanism for registering custom picklers.

    Classes:

        Pickler
        Unpickler

    Functions:

        dump(object, file)
        dumps(object) -> string
        load(file) -> object
        loads(string) -> object

    Misc variables:

        __version__
        format_version
        compatible_formats

CLASSES
    exceptions.Exception
        PickleError
            PicklingError
            UnpicklingError
        _Stop
    Pickler
    Unpickler
    _EmptyClass

    class PickleError(exceptions.Exception)
     | Data and non-method functions defined here:
     |
     | __doc__ = None
     |
     | __module__ = 'pickle'
     |     str(object) -> string
     |
     |     Return a nice string representation of the object.
     |     If the argument is a string, the return value is the same objec

 (以下省略)
```

図 4.1.3.2-2　PyDoc により自動生成されたマニュアル（Python 標準ライブラリ Pickle）

```
class Nuclide:
    """このクラスは核種を表します。引数には、原子番号（第１引数）と質量数
    （第２引数）を渡します。

    例：Pu-239 のインスタンス生成と名前の表示
        >>> p9 = Nuclide(94、239)
        >>> print p9
        Pu-239

    核種が核変換した場合や中性子と反応した場合に、どの核種になるかと
    いう情報を持っています。

    例：U-238 から Pu-239 への核変換
        >>> u8 = nuclide.Nuclide(92、238)
        >>> print u8
        U-238
        >>> print u8.capture().betaDecay().betaDecay()
        Pu-239

    将来、各解析コードで利用される ID 番号を返すメソッドを実装する予定です。
    """

    def __init__(self、 an、 am):
        self.atomicNumber = an
        self.atomicMass = am

    def __repr__(self):
        return self.nameOfNuclide()

    def nameOfElement(self):
        "元素名（例：Pu-239 の場合、Pu）を返します"

        __list = ['H'、'He'、'Li'、'K'、'B'、'C'、'N'、'O'、'F'、'Ne'、
                  'Na'、'Mg'、'Al'、'Si'、'P'、'S'、'Cl'、'Ar'、'K'、'Ca'、
                  'Sc'、'Ti'、'V'、'Cr'、'Mn'、'Fe'、'Co'、'Ni'、'Cu'、'Zn'、
                  'Ga'、'Ge'、'As'、'Se'、'Br'、'Kr'、'Rb'、'Sr'、'Y'、'Zr'、
                  'Nb'、'Mo'、'Tc'、'Ru'、'Rh'、'Pd'、'Ag'、'Cd'、'In'、'Sn'、
                  'Sb'、'Te'、'I'、'Xe'、'Cs'、'Ba'、'La'、'Ce'、'Pr'、'Nd'、
                  'Pm'、'Sm'、'Eu'、'Gd'、'Tb'、'Dy'、'Ho'、'Er'、'Tm'、'Yb'、
                  'Lu'、'Hf'、'Ta'、'W'、'Re'、'Os'、'Ir'、'Pt'、'Au'、'Hg'、
                  'Tl'、'Pb'、'Bi'、'Po'、'At'、'Rn'、'Fr'、'Ra'、'Ac'、'Th'、
                  'Pa'、'U'、'Np'、'Pu'、'Am'、'Cm'、'Bk'、'Cf'、'Es'、'Fm'、
                  'Md'、'No'、'Lr']
        return __list[self.atomicNumber-1]

    def nameOfNuclide(self):
        "核種の名前（例：Pu-239）を返します"
        return self.nameOfElement() + '-' + str(self.atomicMass)

    def alphaDecay(self):
        "α崩壊後の核種（Nuclide）のインスタンスを返します。"
        return Nuclide(self.atomicNumber-2、 self.atomicMass-4)

    def betaDecay(self):
        "β崩壊後の核種（Nuclide）のインスタンスを返します。"
        return Nuclide(self.atomicNumber+1、 self.atomicMass)

    def n2n(self):
        "(n、2n)反応後生成する核種（Nuclide）のインスタンスを返します。"
        return Nuclide(self.atomicNumber-1、 self.atomicMass-2)

    def capture(self):
        "中性子捕獲反応後生成する核種（Nuclide）のインスタンスを返します。"
        return Nuclide(self.atomicNumber、 self.atomicMass+1)

    def idOfJFSLIB(self):
        "JFS-3 ライブラリの核種 ID を返します（未実装）。"
        raise exceptions.NotImplementedError
```

図 4.1.3.2-3　プログラムソースファイルへの説明文書の挿入例（Nuclide クラス）

```
Python Library Documentation: module nuclide

NAME
    nuclide

FILE
    /home2_ofbrf96/yokoyama/Work/isotope/nuclide.py

CLASSES
    Nuclide

    class Nuclide
     | このクラスは核種を表します。引数には、原子番号（第１引数）と質量数
     |  （第２引数）を渡します。
     |
     | 例：Pu-239 のインスタンス生成と名前の表示
     |     >>> p9 = Nuclide(94、239)
     |     >>> print p9
     |     Pu-239
     |
     | 核種が核変換した場合や中性子と反応した場合に、どの核種になるかと
     | いう情報を持っています。
     |
     | 例：U-238 から Pu-239 への核変換
     |     >>> u8 = nuclide.Nuclide(92、238)
     |     >>> print u8
     |     U-238
     |     >>> print u8.capture().betaDecay().betaDecay()
     |     Pu-239
     |
     | 将来、各解析コードで利用される ID 番号を返すメソッドを実装する予定です。
     |
     | Methods defined here:
     |
     | __init__(self、 an、 am)
     |
     | __repr__(self)
     |
     | alphaDecay(self)
     |     α崩壊後の核種（Nuclide）のインスタンスを返します。
     |
     | betaDecay(self)
     |     β崩壊後の核種（Nuclide）のインスタンスを返します。
     |
     | capture(self)
     |     中性子捕獲反応後生成する核種（Nuclide）のインスタンスを返します。
     |
     | idOfJFSLIB(self)
     |     JFS-3 ライブラリの核種 ID を返します（未実装）。
     |
     | n2n(self)
     |     (n、2n)反応後生成する核種（Nuclide）のインスタンスを返します。
     |
     | nameOfElement(self)
     |     元素名（例：Pu-239 の場合、Pu）を返します
     |
     | nameOfNuclide(self)
     |     核種の名前（例：Pu-239）を返します
     |
     | ----------------------------------------------------------------------
     | Data and non-method functions defined here:
     |
     | __doc__ = '¥xa4¥xb3¥xa4¥xce¥xa5¥xaf¥xa5¥xe9¥xa5¥xb9¥xa4¥xcf¥xb3¥xcb¥xb...
     |     str(object) -> string
     |
     |     Return a nice string representation of the object.
     |     If the argument is a string、 the return value is the same object.
     |
     | __module__ = 'nuclide'
     |     str(object) -> string
     |
     |     Return a nice string representation of the object.
     |     If the argument is a string、 the return value is the same object.

DATA
    __file__ = './nuclide.pyc'
    __name__ = 'nuclide'
```

図 4.1.3.2-4　PyDoc により自動生成されたマニュアル（Nuclide クラス）

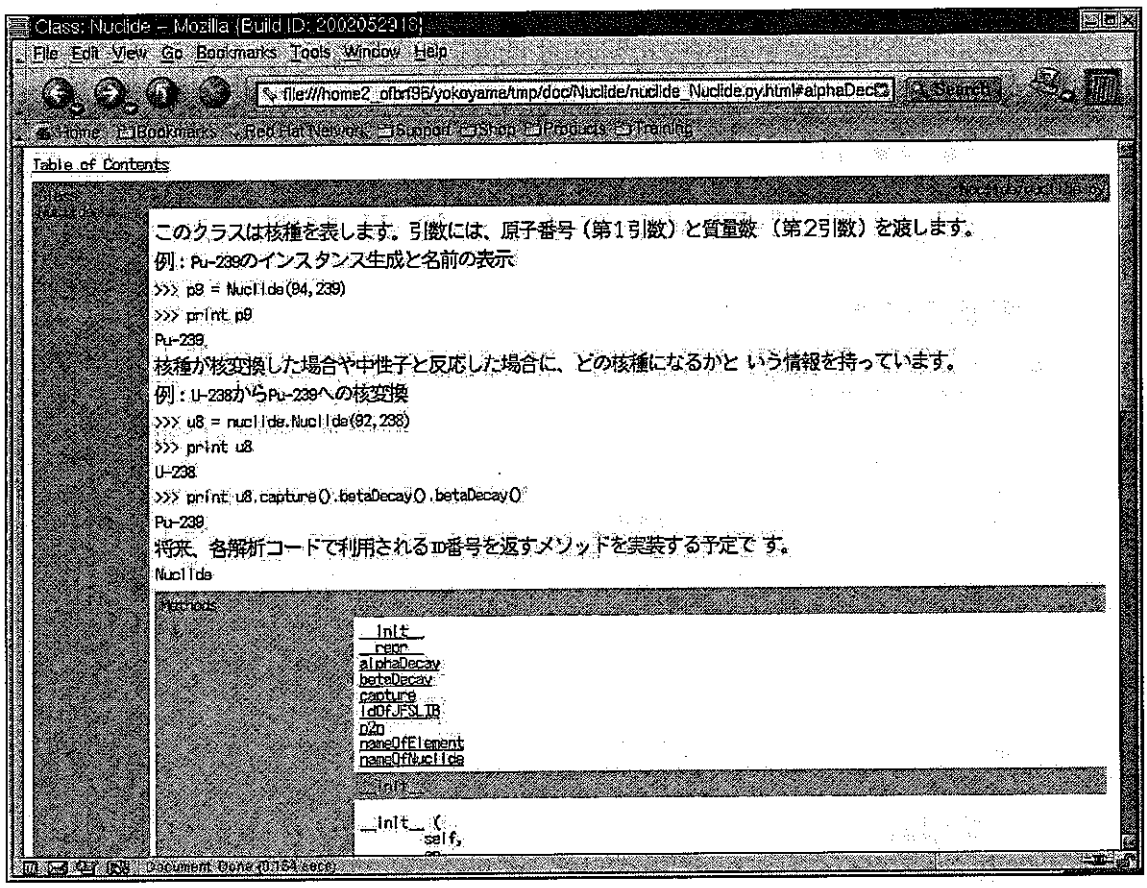


図 4.1.3.2-5　HappyDoc により自動生成された HTML 形式のマニュアル（Nuclide クラス）

4.1.4 代表的なオブジェクト指向言語

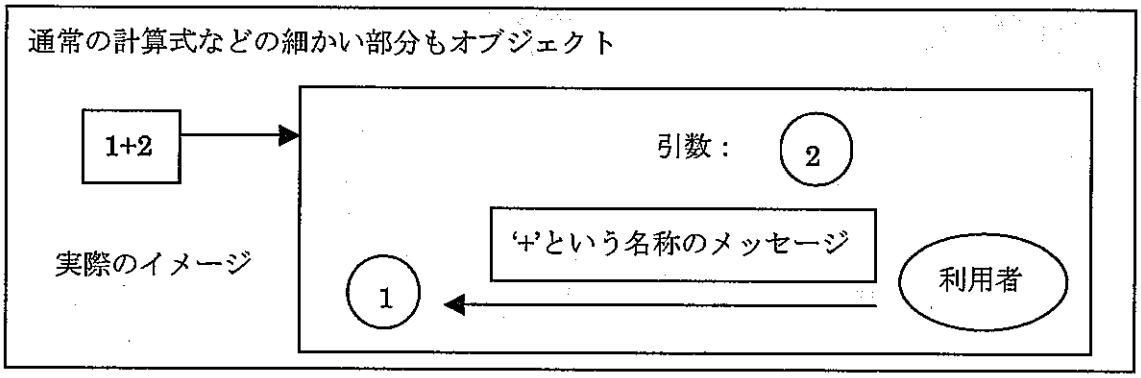
4.1.4.1 Smalltalk

Smalltalk は、1970 年代 米国ゼロックス社により誕生した。やがて開発が進み 1980 年に Smalltalk-80 として言語仕様は完成し、オブジェクト指向概念を実際にコンピュータ上で表現することに成功した。

現在の数多くあるオブジェクト指向言語は、全てこの Smalltalk-80 を参考にし誕生したものである。

それでは、Smalltalk の特徴を以降に示す。

Smalltalk システムは、システム構成要素、プログラム表記上の要望が全てオブジェクトとして考えられている。システム機能がばらばらの API で公開されているものに比べると、かなり使いやすいが、プログラミング学習を行う上ではかなりの労力を必要とする。



計算式においては、数値 1、2 は、Smalltalk システム上、クラスのインスタンスとして、演算子+は、オブジェクトへのメッセージ式として理解される。

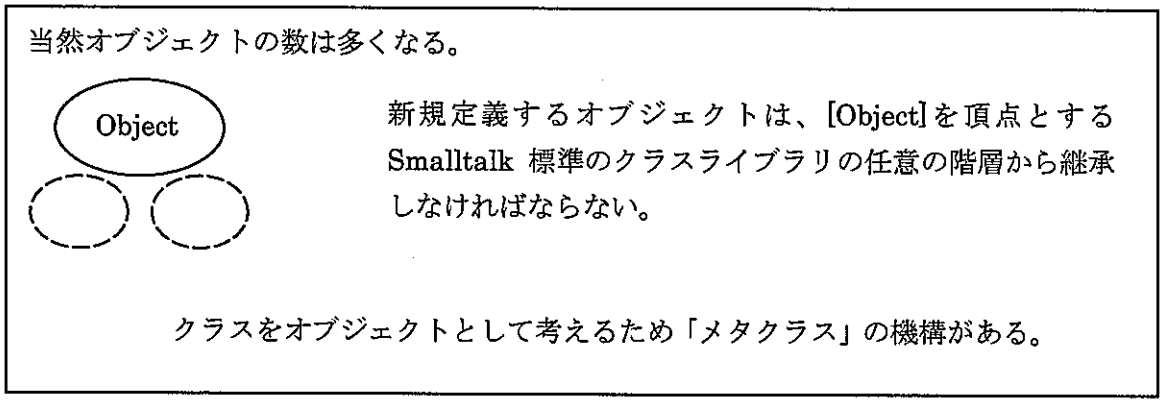


図 4.1.4.1-1 Smalltalk の特徴

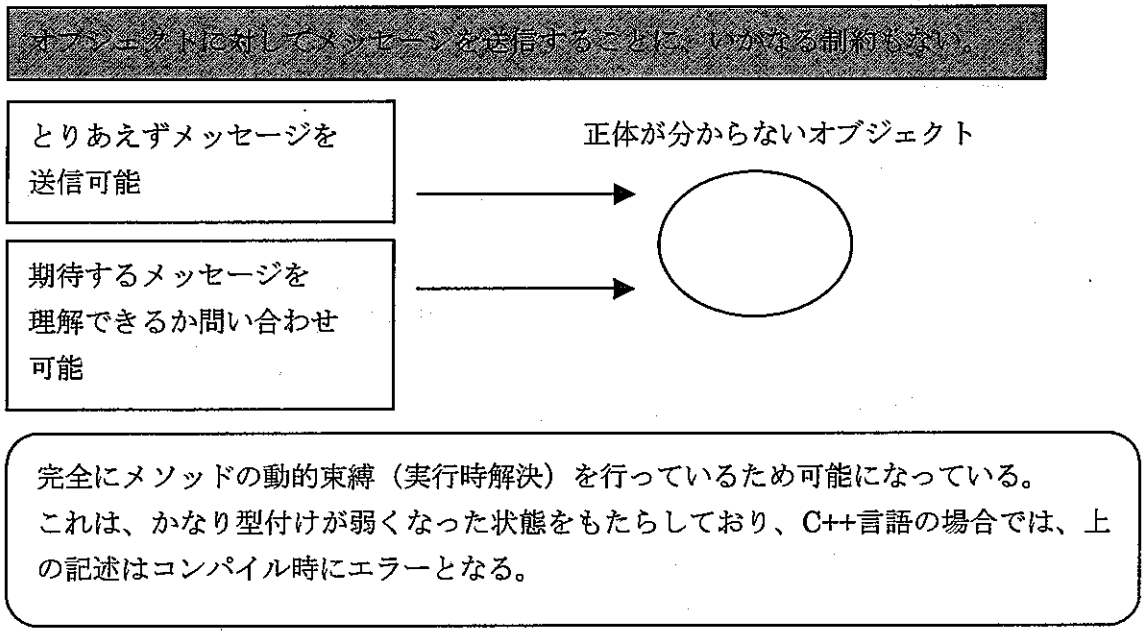


図 4.1.4.1-2　Smalltalk におけるメッセージ送信

Smalltalk においては、クラス階層におけるメソッドの再定義は、クラスの型が分からなくても、インターフェースが分かれば、メッセージを送信することを可能とする。（ポルモルフィズム）

C++言語プログラムとの比較

全てのインスタンスメソッド（メンバ関数）が、C++言語でいうところの仮想関数として定義されており、それがデフォルトになっていることが挙げられる。
（C++言語は、基本的に静的結合であるため、動的結合の実現のための仮想関数という機構を提供している。つまりプログラマがそのことについて意識する必要がある。）

Smalltalk においては、インヘリタンス（継承）は似たような振る舞いを持ったクラスを階層的に組織化するために使用されるべきものという考えがある。

Smalltalk 継承関係の特徴

	C++	Smalltalk
継承時のアクセス制御	○	全てのメソッドと変数を継承
多重継承	○	禁止

図 4.1.4.1-3　動的束縛とクラス階層構造によるポルモルフィズムの実現

定数標準が可能なクラス（定数表記によりクラスのインスタンスおして生成される）	
数（Number）	「123」「-123」「1.23」「3.45e21」のような記述を行う。
文字列（String）	Smalltalk の文字列は、シングルコーテーションで囲む。（例'ABCDE'、'123456'）ダブルコーテーションで間違って囲むと、それはコメントとして認識される。
文字（Character）	文字の前にダラー記号を付けて記述する。（「$a」「$9」）'A' と$A はオブジェクトとしては、違うことに注意しなければならない。
シンボル（Symbol）	「#cat」「#Red」のような形式で記述する。C 言語での enum 変数がシステムでユニークになったようなものと考えられる。ユニークであることを除けば文字列と同じであるが、並びが同じ文字列を、シンボルとして定数記述したものは、全て同一のインスタンスである。
配列（Array）	通常の言語にはない便利な定数。「#(10 $A '12')」と記述する。この例では整数オブジェクト、文字オブジェクト、文字列オブジェクト、を持つ集合オブジェクトを生成する。
ブロック式（BlockClosure）	これは Smalltalk 独特の考え方で、式をオブジェクトとして表現できるものである。式を[]表記でくくることで内部の式は、ブロック式のオブジェクトとして認識される。 最初は、理解に苦しむが Smalltalk のパワーを感じさせる定数表現である。

図 4.1.4.1-4　Smalltalk の基本文法

4.1.4.2 Java

UNIX とともに広まった TCP/IP のネットワークが、インターネットに発展し、商用化されたのは 1995 年だが、これとほぼ同時に登場したのが Java である。Java は米サン・マイクロシステムズによって発表された仕様であり、インターネットでの利用を強く意識している。Java をプログラミング言語として見た場合には C++言語の発展版のように思えるが、Java がそれまでの言語と最も違うのは、マルチプラットフォームで動かすことを最初から考慮し、実行環境や開発環境などが仕様として存在している点にある。そして、それらが実際に受け入れられ、多くのプラットフォームで移植性の高いプログラミング環境が実現されていることが、高く評価されている一因になっている。つまり、Java はこれまでのプログラミング言語の集大成といえるのだ。ライブラリ方式、構造化、オブジェクト指向などの思想はすべて盛り込まれているし、ファイル I/O、ネットワーク、リレーショナルデータベース（RDB)、多言語、ウィンドウ GUI などがすべて言語仕様そのものに含まれている。プログラミング言語仕様自体にこのような考慮がなされたものは、実はこれまでなかったのである。

(1) Java 技術の構成要素

Java 技術は以下の 3 つの要素から構成されている。

- Java 言語：新たに設計したオブジェクト指向プログラミング言語
- Java VM：Java バイトコード（実行用の中間形式）を実行する仮想マシン
- Java 言語から利用できるクラス・ライブラリ群

Java が登場してからの歴史を以下に示す。

1995 年 5 月	Sun Microsystems が Java を発表 Netscape Communications 社が Java の採用を表明
1995 年 9 月	Netscape Navigator2.0 ベータ版を公開。Java 実行機能を標準で搭載
1995 年 12 月	Microsoft 社が Java ライセンス供与を受ける
1996 年 1 月	米 Sun Microsystems の JavaSoft 事業部門が発足 基本開発キット JDK1.0 正式版を公開
1996 年 2 月	JDBC を発表

1996年3月	Microsoft社がActiveX技術を発表 SunSoftがJava WorkShopを発表
1996年4月	OSベンダー10社がJavaの組み込みを表明 シマンテック、WinCafe国内出荷開始。Windows環境で最初の商用開発ツール
1996年5月	開発者会議JavaOne開催。新API群の構想などを公表 Java OS発表、20社以上がライセンスを受けるJavaチップを発表 Oracleらが発表したNetwork Computerの仕様にJava実行機能を取り入れ
1996年6月	Microsoft、Internet Explorer3.0ベータ版の付加ソフトJava Support公開。ActiveX技術を取り入れたJava VM
1996年7月	Netscape、開発環境Netscape ONE発表。その中でJavaクラス・ライブラリIFCを主要技術の一つと位置づける
1996年9月	Netscape Navigator3.0製品版を公開。Windows版はBorland製JITコンパイラを装備
1996年10月	Java Beans APIの仕様策定プロセスが終了 JavaSoftがJava Card APIを発表 米Marimba、Javaベースのコンテンツ配信システムCastanetを発表
1996年10月	米Sun Microsystems、JavaStationを発表。独SAP、カナダCorelなどサード・ベンダー35社が対応ソフトを展示
1996年11月	IBMがJava OSのライセンス供与を受け、同時にJava検証センターの設立計画を発表
1996年12月	100% Pure Javaプログラム認定制度の発表 基本開発キットJDK1.1ベータ版を公開
1997年2月	JDK1.1正式版を公開
2002年2月	JDK1.4を公開

Webアプリケーションを構築するためのサーバー側Java技術に注目が集まっている。鍵と

なるのは、(1)Servlet、(2)JSP(JavaServer Pages)、(3)EJB(Enterprise JavaBeans)、の三つ。これらの技術が登場してきたことで、大規模な Web アプリケーションを構築する枠組みが整った。実際、多数の Web アプリケーション・サーバー製品が、EJB を始めとする各種技術を取り入れつつある。

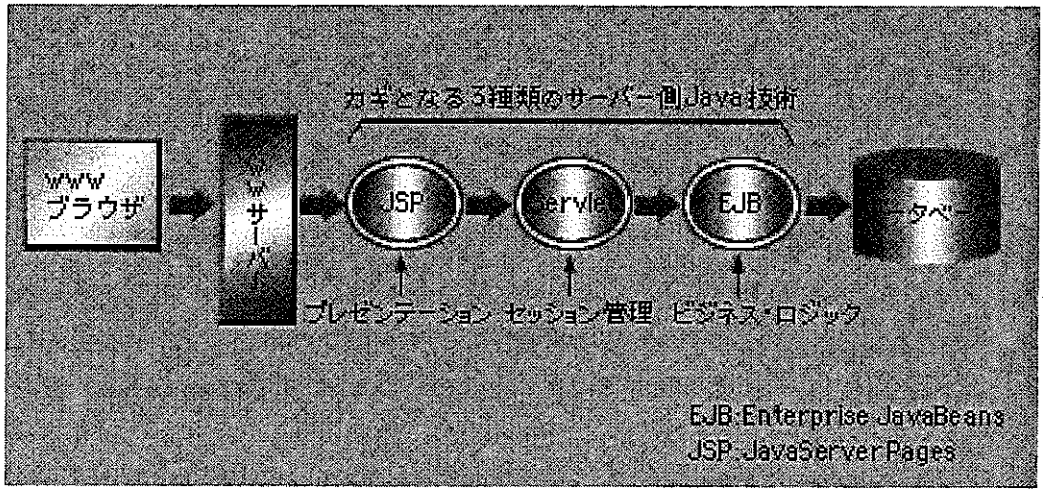


図 4.1.4.1-5 サーバ側 Java 技術の構成

(2) Java で使用される主な用語

① Java VM(JVM)

Java 仮想マシン(VM: Virtual Machine)といい、Java プログラムを実行するには、まずソース・コードをバイトコードと呼ぶ中間形式にコンパイルし、Java VM がバイトコードを解釈して実行する。バイトコードを実行させる環境が Java 仮想マシン(Java VM)である。Java VM は、各種 UNIX、Windows、MacOS など、ほとんどの主要 OS の上で提供されているために、Java で作成したプログラムは稼働環境を選ばない(アーキテクチャ独立)。またプログラムがシステムに対して有害な動作をしないことを保証するため、Java VM にはバイトコード検証機構が組み込まれている。

② JDK(Java Development Kit)

米 Sun Microsystems 社が提供する Java 基本開発キット。サード・ベンダーの Java 関連製品も、JDK を元にして開発される。このため、JDK のバージョン番号は、事実上「Java のバージョン番号」としての意味を持つ。

JDK の最新版は 2002 年 2 月に出荷開始した JDK1.4。ただし、2002 年 7 月現在、WWW ブラウザとして普及している Netscape Navigator6 で JDK1.3、Internet Explorer5 以上で JDK1.1.4、HotJava Browser3.0 では JDK1.1.6 の実行環境である。

③ JavaScript

米Netscape Communications社と米Sun Microsystems社がライセンスするスクリプト言語。WWWブラウザ 製品Netscape Navigatorや、Internet Explorerに搭載されている。「Java」の名前を冠しているが、JavaScriptはJavaとは異なる言語である。文法がC++類似である点はJavaと 似ているが、Javaと異なり「型に甘い」言語である。HTMLに埋め込んで利用でき、Webに対話性を与えるなどの応用に用いる。
Netscape Navigatorに搭載されているLiveConnect機能を利用することで、Java言語と連携できる。

④ JavaBeans

Java プログラムから利用できるソフトウエア部品の仕様。個々のソフトウエア部品をBeanと呼ぶ。JDK1.1以降、Java Beansを実現するためのJava Beans APIを取り入れた。他のコンポーネント・アーキテクチャ (ActiveX、OpenDoc、LiveConnect)と連動させることができる。

1997年4月、米Sun MicrosystemsはGlasgow(コード名)と呼ぶ次期Java Beansの仕様案を公開した。複数Beanをグループ化して扱う機能を盛り込む。オブジェクトの集合をあたかも単一のオブジェクトのように扱う仕様(Aggregation)や、あるオブジェクトが代表として他のオブジェクトと通信する仕様(Delegation)を盛り込む。
Java Beans ページ→http://java.sun.com/beans/

⑤ Enterprise JavaBeans(EJB)

3層モデルの中間層(アプリケーション・サーバー)のためのソフトウエア・コンポーネント仕様。「JavaBeans」をサーバー・ソフト向けに拡張したコンポーネント・モデル。1998年 3 月開催の JavaOne'98 に合わせ仕様書第 1.0 版が公開された。同時に Borland International、IBM、Netscape Communications、Novera、WebLogic などアプリケーション・サーバー・ベンダー各社が採用を表明した。

ソフト部品のコンテナをEnterprise JavaBeans Server(EJB Server)と呼び、このEJB Serverが分散メッセージング機能や2フェース・コミット処理を実行する機能、セキュリティ機能などをあらかじめ備えている点がJavaBeansとは異なる。ソフト部品の開発者は、サービスをほかのソフト部品へネットワーク経由で提供するための機能やトランザクション処理エンジンを使うための機能を実装する必要がない。ソフト部品として提供する機能の開発に専念できることがEJBを使うことの開発者にとってのメリットになる。
EJB では、ソフト部品の永続性(persistence)を本格的にサポートする点も既存のJavaBeansとは大きく異なる。永続的なソフト部品を「Entity Bean」、一時的なソフト部品を「Session Bean」と呼び、データベース上に登録してあるデータやファイルなど、さまざまなオブジェクトをEntity Beanとして表現できるようにした。

既存のJavaBeansではソフト部品の永続性をオブジェクトをシリアル化する機能で代用

しているだけである。ソフト部品を保存すべきかどうかの決定は開発者にゆだねられているし、コンテナとなるアプリケーションがなんらかの理由で異常終了した場合に Bean が備えていたデータの内容などはすべて消えてしまう。Entity Bean では、表すデータの一貫性を EJB Server が保証する。EJB 1.0 の仕様ではこの Entity Bean を扱うための機能がオプションとなっていたが、2000 年 6 月に発表された EJB 2.0 では標準で実装された。

⑥ アプレット

アプレット(Applet)は、他のアプリケーションに読み込んで動作させる形式の Java プログラム。「小アプリケーション」という意味。アプレットに対して「Java アプリケーション」は単独で実行できるプログラム形態を指す。

WWW ブラウザ(HotJava、 Netscape Navigator、 Microsoft Internet Explorer)に読み込んで動作させることが可能。プログラミング上は、java.applet.Applet クラスを継承して作成するクラス(Class 定義文に extends Applet の節が加わる)。

アプレットは、セキュリティ上の配慮から、ブラウザによって厳しく機能が制限されている。例えば、Netscape Navigator ではローカル・ファイルへのアクセスはできない。また、ダウンロード元のホストとしか通信できない。これは、アプレットによる悪質なプログラム（コンピュータ・ウイルスや、致命的なバグを含むプログラム）を締め出すためである。

アプレットとして作成したプログラムを業務システムでも応用しようという動きがある。メリットは主に 2 点(1)ソフト配布の手間を省ける。実行ファイルを WWW サーバー側に配置するために、多数のクライアントにいちいちアプリケーションをインストールする手間がなく、マルチプラットフォームであり UNIX、 Windows、 Macintosh、JavaStation(Java 実行専用のネットワーク・コンピュータ)など多様な動作環境で共通のプログラムを動作させることができる。

⑦ Java 3 D

Java 3D は Java の標準的な 3 次元グラフィックス API("Application Program Interface" であり、正式名称は Java 3D API という(API:アプリケーションプログラムからシステム(OS)やライブラリ、言語処理系などに対する標準的な機能呼び出しの方法について定義したものが API)。Java API には Core API、標準拡張 API があり、Core API は Java 2 SDK、JRE(Java Runtime Environment)に含まれて配布されている。

Java 3D の歴史

1996　　JavaOne で Java Media API 発表。
　　Java 3D もこの中に含まれていた
1998　　Java 2(JDK 1.2)リリースとほぼ当時に Java 3D 1.1 をリリース
2002　　Java 3D 1.3 リリース

Java 3D は 1998 年に Sun(JavaSoft)によって開発された。1996 年の発表当初は SGI(シリコン・グラフィックス)、Intel、Apple も開発に参加するとアナウンスされたが、 正式リリース版は Sun の開発チームによるものであった(リリースバージョンは 1.1 から。バージョン 1.0 は実装されていない)。

2000 年に 1.2 がリリースされ、描画内容が再利用できるようになるオフスクリーンレンダリング機能や、物体に貼り付けた画像を交換できるマルチテクスチャー機能などが追加さ、本格的な 3 次元画像を作成できるようになった(現在 1.3 がリリースされている)。

Java では" Write once run anywhere "というスローガンがよく使われるが、これは、一度コンパイルすればJavaをサポートするどんな環境でも実行可能になることをうたっており、Java 3D にはこれによく似た"Write once、 view anywhere"というスローガンがある。

Java 3D では、一度コンパイルされれば Java2 をサポートするどんな環境でも表示可能になることを目的として開発されている。

ただし、現在 Sun が正式にサポートしているプラットフォームは、Windows95/ 98 / NT4.0 / 2000 と Solaris である。この他 SGI の IRIX、IBM(AIX)、Hewlett-Packard(HP-UX) そして Linux(一部条件あり*1)にも対応している。

Java 3D は描画に際して OpenGL、DirectX などプラットフォーム依存の 3D 描画 API を使用するが、 Java 3D 利用者はこれら下位 API に依存せずにプログラミングできる。

---

*1:米国 Blackdown 社が Sun の Java に手を加え、リリースしているバージョンを使用する。

4.1.4.3 C++

(1) 概要

文献 4.1.4.3-1 を参考に、まず C++の概要について述べる。

C++は 1980 年代の初頭に、Bjarne Stroustrup により最初の実装が行われたオブジェクト指向プログラミング言語である。Bjarne Stroustrup は、C 言語に抽象データ型の便宜を提供するものとしてクラスを考案し、C++は当初「クラスを備えた C 言語」と呼ばれていた。すなわち、C++のインクリメント部分（++）は「C 言語のオブジェクト指向的な拡張」を表しており、その要となるクラスは、C 言語における構造体定義の拡張技法を用いて定義することができる。

その後、さらに拡張が施され、数年後には現在の「オブジェクト指向プログラミング言語 C++」の原形が出来上がったが、1990 年代になり C++を標準化する試みが開始された。紆余曲折を経た結果、1997 年末に米国規格協会 ANSI と国際標準化機構 ISO により、C++の標準規格最終草案が採択され、翌 1998 年 9 月に公式ドキュメント「Programming Language C++、 ISO/IEC Standard 14882」が明らかにされた。こうして標準 C++が出来上がり、Bjarne Stroustrup が構築した当初の C++に比べ、STL（Standard Template Library：標準テンプレートライブラリ）と呼ばれるテンプレートベースの強力なライブラリ機能が標準装備として追加された。ここで、テンプレートとは、プログラミング時には型が未決定であって、コンパイル時に実際の用法に基づき型を決定できるという汎用型のことを指す。

なお、フランスでは CEA と EDF が共同で、炉物理、熱流動、構造等の原子力工学の基盤となる要素分野全般にわたる解析システムを構築するという大規模なプロジェクトが計画されている。この中で、オブジェクト指向データやソルバーの構築などには C++が使われる予定であり、今後の動向が注目される。

以下で、C++関連ツールのうち、Visual C++.NET の特徴について述べる。

(2) Visual C++.NET

Visual C++ .NET は、Web アプリケーションと XML Web サービスの作成と利用を可能にするための新機能と機能強化を多数含んでいる。Visual C++ .NET は、より小さく高速で高機能のプログラムを、より短期間に作成できるようにすることを目標としている。

Visual C++は、パワーと柔軟性の点で優れている。C++の開発者はアプリケーションを構成するコンポーネントを最大限に制御したいと考えており、開発ツールには可能な限りコンパクトでパフォーマンスの高いプログラムを生成することを期待している。Visual C++ .NET により、開発者はこれらの目標をより簡単に実現することができる。

①パワーと柔軟性

Visual C++は、.NET Framework が提供するマネージド コード モデルと、アンマネー

ジのネイティブな Windows コード モデルの両方をサポートしているという点で、.NET 言語の中で独特の地位にある。Visual C++ .NET は両方のプログラミング モデルをサポートすることで、従来の Visual Studio 環境で開発された既存の投資を守り、有効に活用して、開発者に最大限の選択肢を提供している。

Visual C++ .NET には、コンパクトで高性能の Web アプリケーションとサービスを作成できる、 ATL (Active Template Library) サーバーが含まれている。ATL サーバーは、高性能の Web アプリケーションの開発のベスト プラクティスを抽出し、それらを開発者が再利用できる単純で拡張性のある ATL クラスのセットにカプセル化している。

Visual C++の機能強化により、クライアント サイドまたはサーバー間プログラムから他の XML Web サービスを簡単に呼び出せるようになっている。これは、開発者が .NET Framework、Microsoft Foundation Class (MFC)、ATL のどれを使用していても、あるいは Windows API を直接プログラミングしている場合でも可能である。

新しい Visual C++のマネージ拡張により、開発者は .NET Framework をターゲットとする C++プログラムを作成することができる。またマネージ拡張は、Visual C++のコードと、Visual C# .NET および Visual Basic .NET を含む他のマネージド言語の間のブリッジの役割も果たす。

また、Visual C++ .NET には、既存の Windows ベース アプリケーションの保守と拡張を容易にする機能も多数含まれている。MFC のアップグレードにより、開発者は Windows プラットフォームの最新の機能にアクセスすることができる。Visual C++は、生成されるコードの品質とパフォーマンスの点で、依然としてリーダーとしての地位を保っている。さらに、コンパイラの新しい最適化機能により、これまでのリリースよりも小さく高速なコードが生成されるようになっている。

主な要点を以下に挙げる。

- 属性によって、開発者が書かなくてはならないコードの量が減る
- C++マネージ拡張を使って、既存のアプリケーションを.NET Framework に統合できる
- ATL サーバーを使って高性能の XML Web サービス インフラストラクチャが作成できる
- エラー修正機能により、より堅牢なコードを作成できる
- 強化された最適化コンパイラによって、強力なアプリケーションを作成することができる

4.1.4.4 C#

(1) 概要

C#(Visual C#.NET)は、Microsoft 社が 2002 年に発表した開発言語パッケージ「VisualStudio.NET」に含まれる C/C++の言語仕様を基に新たに開発された言語バージョンである。

過去 20 年間、C と C++は主要なソフトウェアとビジネス ソフトウェアを開発するためのプログラミング言語として、最も広く使われてきた。どちらの言語でも、プログラマは非常に細かいレベルで制御を行うことができるが、この柔軟性には生産性の面でのコストがある。Microsoft Visual Basic などの言語と比べると、それと同等の C および C++アプリケーションは、一般に開発により多くの時間がかかる。このような複雑さと開発時間の長さのために、多くの C および C++プログラマは、パワーと生産性をうまくバランスさせている言語を探してきた。

一部の言語は、C および C++プログラマが必要とする柔軟性を犠牲にすることで生産性を高めている。このような解決方法は、開発者をあまりに制約し(たとえば低水準のコードを制御するメカニズムの省略)、最大公約数的な能力しか提供できない。また、既存のシステムとの相互運用が難しく、必ずしも現在の Web プログラミングの状況に適合しない。

C および C++プログラマにとっての理想的な解決方法は、ラピッド開発と、下位のプラットフォームのすべての機能へのアクセスの組み合わせである。プログラマは、新しい Web 標準に完全に対応し、既存のアプリケーションとの統合が簡単に行える環境を望んでいる。さらに、C および C++開発者は、必要ならば低水準でコーディングを行える機能を必要とする。

この問題に対する Microsoft のソリューションは C#である。C#は、Microsoft .NET Framework 用の幅広いアプリケーションを短期間で構築できる現代的なオブジェクト指向言語で、コンピューティングと通信における最新の進歩を利用するためのツールとサービスを提供している。

オブジェクト指向デザインを持つ C#は、高水準のビジネス オブジェクトからシステムレベル アプリケーションまでの幅広いコンポーネントの構築に適している。コンポーネントは単純な C#言語要素を使って XML Web サービスに変換し、任意のオペレーティング システム上で実行されている任意の言語から、インターネット経由で呼び出すことができる。

特に重要なのは、C#が、C と C++の特徴だったパワーとコントロールを犠牲にせずに、C++プログラマにラピッド開発機能を提供できることである。C#は C と C++にきわめてよく似ているため、これらの言語を知っている開発者は、C#でもすぐに生産的な活動を始めることができる。

Visual C# .NET は Microsoft Visual Studio .NET の一部として提供され、共通の実行エンジンと豊富なクラス ライブラリを含む、Microsoft .NET Framework へのアクセスを提供する。Microsoft .NET Framework が定義している共通言語仕様(CLS)は、CLS 準拠の言

語とクラス ライブラリの間のシームレスな相互運用性を保証する一種の共通言語である。開発者にとっては、Visual C# .NET は新しい言語であるにもかかわらず、Visual Basic .NET や Visual C++ .NET などの昔からのツールで使われていた豊富なクラス ライブラリに自由にアクセスできるということになる。

主な要点を以下に挙げる。

- C および C++開発者の生産性の向上
- .NET コンポーネント、XML Web サービス、および Web アプリケーションの構築が可能
- 耐障害性と堅牢性の高いプログラムの開発
- バージョン管理のサポートを内蔵
- ソフトウェア コンポーネントの構築の第一級のサポート
- よく見られるメモリ管理エラーを解消する自動ガーベジ コレクション
- より単純で理解しやすいコードを書くことができるクリーンな言語デザイン

Visual C#.NET の基本的な特徴

①.NET Framework アプリケーション開発向けのプログラミング言語
② 高い生産性を持つ VisualBasic と、オブジェクト指向言語 C++の特徴を併せ持つ
③ C/C++よりも馴染みやすい構文
④ C++よりも精錬された言語
　スコープ、ポインタに関する構文（:: . ->）を簡素化
　ポインタ型による直接的なアドレス操作は出来ない
　オブジェクト指向を強制
⑤ 共通言語ランタイム環境対応
　CTS/CLS 準拠によるタイプセーフなデータ構造
　動的なメモリ自動管理、ガーベジコレクション
⑥ .NET Framework クラスライブラリを利用
⑦ C#言語仕様及び.NET Framework のサブセットは、ECMA によって承認されている
　2001 年 12 月、ECMA-334、ECMA-335

(2) ボクシングとアンボクシング

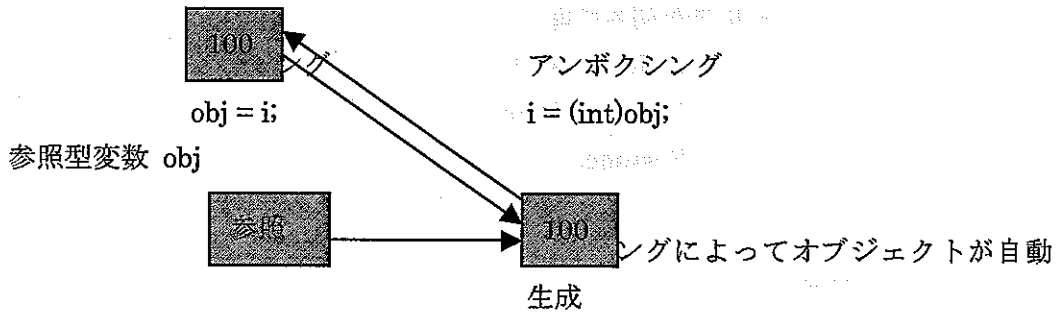
C#では、新たにボクシングとアンボクシングという機能が取り入れられている。

ボクシングとアンボクシング

値型と参照型間でキャストが可能

- ボクシング：値型データをコピーしたオブジェクトを作成し、それを参照する。
- アンボクシング：参照するオブジェクトから、明示的に設定された値を取得する。

```
int i = 100;    //値型
Object obj;     //参照型
Obj = i;        //ボクシング
i = (int)obj;   //アンボクシング
```



ボクシングは、値型変数から参照型変数に代入しようとすると、その値を持つ新しいオブジェクトが自動的に生成されることをいう。そして、そのオブジェクトへの参照情報が参照型変数に代入される。これによって、誤ったメモリ位置を参照してしまう事を防ぐ事が出来る。

アンボクシングは、参照型変数から値型変数に代入しようとした場合、参照型変数が示すオブジェクトから、適切がデータが取得され、値型変数に代入される。もし受け取り側の値型変数のデータ型にみあうデータが取得できなければ、例外が発生する(InvalidCastException)。

(3) 例外処理

VB.NET の例外処理では、従来 Visual Basic で採用されていた On Error ステートメントを利用した Goto 文形式の制御を行うスタイルのほかに、Try ～ End Try のようにブロック形式の構造化例外処理が導入された。

① Try ～ End Try による構造化例外処理
② Throw による例外の投入
③ Throw によるきめ細かい制御
④ On Error ステートメントによる例外処理

構造化例外処理
(ブロック構造の例外処理)
Try ～ End Try
Catch、 Finally
Throw

非構造化例外処理
(GoTo 文制御の例外処理)
On Error Goto...
Resume

(4) Checked と unchecked

Checked と unchecked

例外発生の制御

- オーバフロー、アンダーフロー発生時に適用
- コンパイルオプション /checked /checked- でも制御可能

```
int x = 10000000;
int y = 10000000;
x *= y;

x = 20000000;
y = 30000000;
unckecked {
  x *= y;
}
```

例外は発生しない

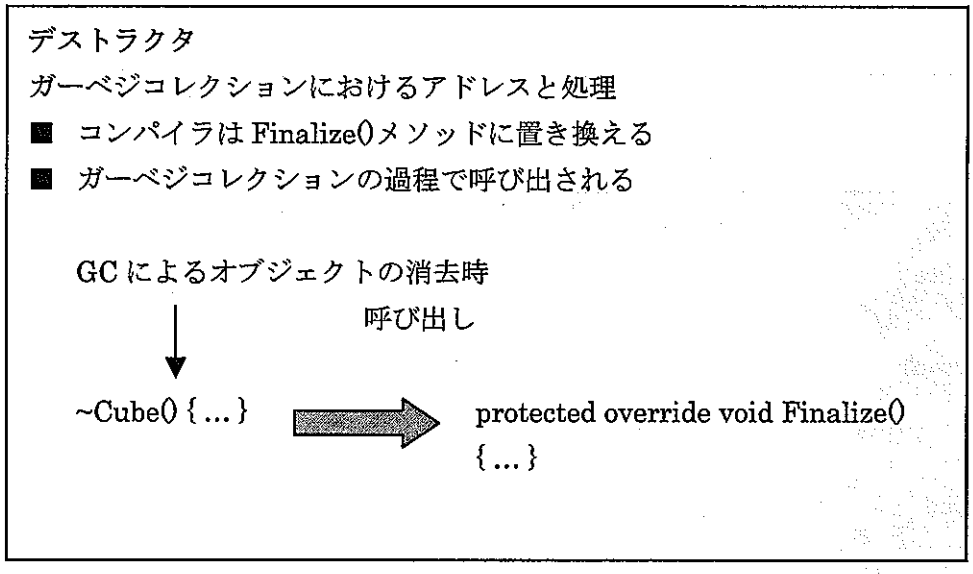
```
int x = 10000000;
int y = 10000000;
checked {
  x *= y;
}
```

例外発生!!
OverflowException

(5) 例外発生の制御

C#では、C/C++と同様に既定では、オーバフローやアンダーフローに対して例外は発生しない。例えば、int 型に加算を行いオーバフローになっても計算はそのまま続行される。もし、オーバフロー時に例外を発生させたい場合には、その部分を checked ブロックで囲みます。

この制御はコンパイラレベルでも行うことが可能で、/checked オプションで制御します。もし、コンパイラレベルで/checked+オプションを使うと、プログラム全体について、オーバフロー時に例外発生の対象となるが、unchecked ブロックで囲った部分は、例外発生の対象外になる。



(6) デストラクタ

C#では、C++と同様にデストラクタを記述することができる。しかし、コンストラクタが中間言語レベルでコンストラクタ専用のメソッドに展開されるのに対して、デストラクタの場合は、コンパイル時に単に Finalize メソッドに置き換えられるだけである。

ガーベジコレクションの過程でオブジェクトが開放されるときに、ガーベジコレクタがそのオブジェクトに Finalize メソッドを見つけると、そのメソッドが呼び出される。

ガーベジコレクションを明示的に起動することは可能であるが（GC.Collect()メソッド）、ガーベジコレクションの処理過程の中で、いつ Finalize メソッドが呼び出されるか、正確な時間は特定できない。また、複数のオブジェクトを削除する際に、どのオブジェクトの Finalize メソッドから呼ばれるか順番を特定することはできない。

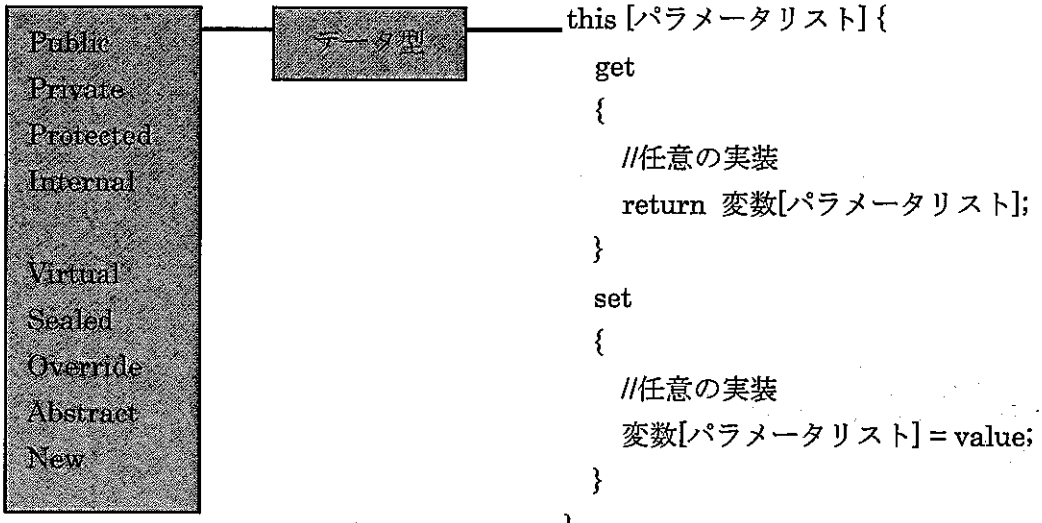
そのため、Finalize メソッド内にリリースの解放コードを書いた場合、リソースを使わなくなったタイミングと、実際に Finalize メソッドが呼び出されるタイミングにタイムラ

グがあり、必要以上にリソースを占有する可能性がある。リソースの後処理忘れを防ぐためには、Finalize メソッドに後処理を記述することは有効であるが、スケーラビリティの点からすると、リソースの解放は、明示的に後処理メソッドを必要に応じて呼び出すよう実装することも重要である。

また、コンストラクタとデストラクタでは実行されるスレッドが異なる。コンストラクタの場合プログラムが new によってコンストラクタを呼び出すとき、プログラム本来のスレッド上でコンストラクタが実行される。一方デストラクタの場合は、ガーベジコレクションのスレッドが実行するので、プログラム本来のスレッドとは異なる。マルチスレッドアプリケーションやスレッドに依存したリソースを扱う場合、Finalize メソッドの扱いには注意が必要である。

(7) インデクサ

インデクサの記述方法
オブジェクトに対してインデクシングが可能



※ 戻り値の値や値の格納方法は任意

インデクサは、オブジェクト変数をあたかも配列のようにインデックスを付けて操作できるものである。インデクサを記述するには、プロパティと同様に get アクセサと set アクセサを記述する。但し、プロパティと異なりプロパティ名の代わりに this キーワードとインデックスとして指定するパラメータの書式を指定する。

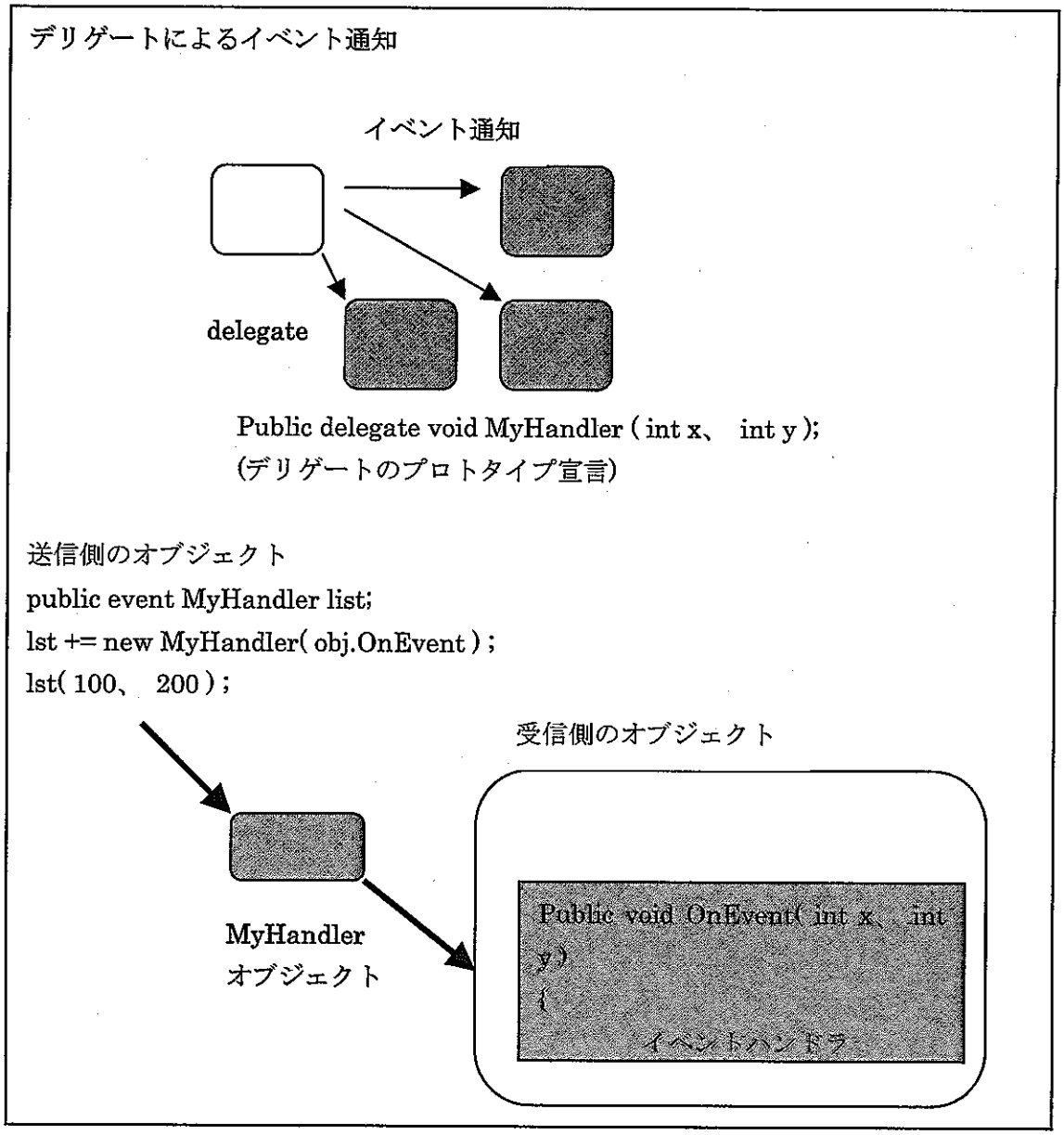
パラメータの部分は整数である必要もなく、複数のパラメータを指定することもできる。

そのため、仮想的な二次元テーブルを表現したり、インデックスに文字列を渡す連想配列などを表現することができる。

修飾子は、メソッドの修飾子と同様に扱います。

継承にともなうオーバライドや、パラメータリストを変えたオーバロードを行うこともできる。

(8) イベント / デリゲートによるイベント通知



イベントに関してオブジェクトが発信するイベントを表現するには、デリゲートを使う。このイベント発信では、1対多のマルチキャストを行うことも出来る。デリゲートは、いわばC言語の関数ポインタをタイプセーフにしたラッパーオブジェクトと見ることが出来る。

Delegate キーワードを使って、まずイベントを受け取るハンドラの引数や戻り値のパターンを宣言する。このとき宣言に使われる識別子（前記の MyHandler）は、関数ポインタをラップしたクラスと見る事が出来る。

イベントを受け取る側は、デリゲートの宣言に見合うメソッドを用意する必要がある。

送信側には、event キーワードとデリゲートの識別子（前記の MyHandler）を伴ういわば変数宣言（list）がある。これは、関数ポインタをラップしたオブジェクトのコレクションとして扱われる。

4.1.4.5 VisualBasic

(1) 概要

VisualBasic.NET は、Microsoft 社が 2002 年に発表した開発言語パッケージ「VisualStudio.NET」に含まれている VisualBasic の新たなバージョンである。今回のバージョンアップで特に重要なのは、Visual Basic が完璧にオブジェクト指向プログラミング言語になったということである。インプリメンテーション継承や構造化例外処理などの機能が完全にサポートされている。

このようなオブジェクト指向機能が追加されただけでなく、Visual Basic は一層ユーザーフレンドリな言語になるようにアップデートされている。言語をより単純化するために、歴史的な理由のみから存在していた機能は削除された。言語の構文は、より論理的に、理解のしやすいものになるように変更されている。このため、コードを見たときに、そのコードが何を実行するのかを理解しやすくなっている。

主な要点を以下に挙げる。

- 継承によるコードの再利用の最大化
- 構造化例外処理によるエラー処理の合理化
- フリースレッド プログラミング モデルによる高性能なアプリケーションの作成
- .NET Framework の利用
- 開発者に必要な機能のみを使用することで、単純さを保つ
- 既存の投資を守るアップグレード ツール

Visual Basic は、オブジェクト指向プログラミング言語、かつ .NET Framework 上の主要な開発言語となっている。開発者は、明示的な軽量プロセスに依存しない手法を使って、スケーラビリティの高いコードを作成することができ、構造化例処理などの言語要素が追加されたために、開発者が作成したコードはきわめて保守しやすくなっている。

VisualBasic.NET の特徴を示す。

① .NET Framework アプリケーション開発向けプログラミング言語
共通言語ランタイムに対応したアプリケーション
複数プログラミング言語による相互運用、バージョン管理、セキュリティ
.NET Framework クラスライブラリを利用
Windows フォーム、Web フォーム、XML Web サービスなどの実装が可能
② 従来の VisualBasic6.0 よりもオブジェクト指向の側面を強化
クラス継承、オーバライド、オーバロードなどの導入
他言語（C#、C++等）と統一のとれたプログラミングモデル
③ VisualBasic6.0 の後継
VB6 プログラマに馴染みやすい構文
従来の VB6 が持つ高い生産性を受け継ぐ、VB6 と同様の開発スタイル
初心者から、高度な実装を必要とする上級者まで幅広い適用範囲

(2) 例外処理

VB.NET の例外処理では、従来 Visual Basic で採用されていた On Error ステートメントを利用した Goto 文形式の制御を行うスタイルのほかに、Try ～ End Try のようにブロック形式の構造化例外処理が導入された。

① Try ～ End Try による構造化例外処理
② Throw による例外の投入
③ Throw によるきめ細かい制御
④ On Error ステートメントによる例外処理

構造化例外処理
(ブロック構造の例外処理)
Try ～ End Try
Catch、 Finally
Throw

非構造化例外処理
(GoTo 文制御の例外処理)
On Error Goto...
Resume

(3) オーバロード

メソッドのオーバロード
オーバロード
メソッド名は同じだが引数リストが異なる。

```
Dim X As Integer = 10、 Y As Integer = 20
Dim S As String = "hello"
Draw( X );
Draw( S );
Draw( X、 Y );
```

引数のリストによって
呼び分ける

```
Over loads Sub Draw( ByVal X As Integer ) ...

Over loads Sub Draw( ByVal S As String ) ...

Over loads Sub Draw( ByVal A As Integer、 ByVal B As Integer ) As Integer
```

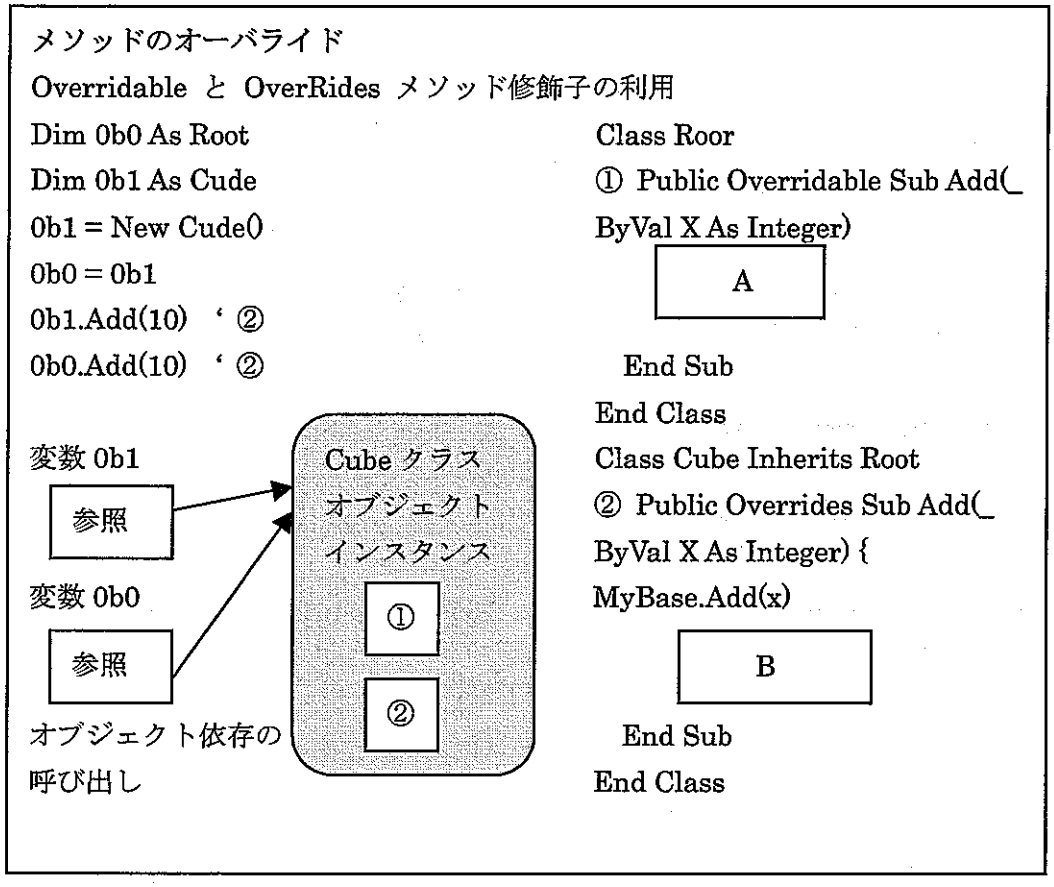
引数リストのパターンを変えて、1つのクラス内に名前が同じメソッドを重複して定義する「オーバロード」を行うことができる。但し、引数リストのパターンが同じで、戻り値の型だけ違うというオーバロードは出来ない。

オーバロードを行うには、メソッドの修飾子として Overloads を付ける。但し、1つのクラス内において同じ名前のオーバロードに関する全てのメソッドに Overloads 修飾子をつけない場合に限り、この修飾子を省略する事が出来る。どれか1つでも Overloads 修飾子を付けた場合は、そのオーバロードに関する同じ名前のメソッドは全部 Overloads 修飾子を付けなければならない。また、継承するとき基底クラスが持つメソッドと同じ名前のメソッドを引数を変えてオーバロードするときも、Overloads 修飾子を付けなければならない。

また、引数のパターンを変えれば、Function と Sub を共存させることができる。

オーバロードされたメソッドを呼び出す際には、渡す引数リストと同じパターンの引数リストを持つメソッドが呼び出される。

(4) メソッドのオーバライド



オーバライドは、継承において基底クラスが持つメソッドと全く同じメソッドを派生クラスで定義することである。これによって単に基底クラスの機能を引き継ぐのではなく、必要に応じて機能を変更することも出来る。オーバライドは、オーバロードと違って戻り値と引数リストのパターンも全く同一でなければならない。

派生クラスオブジェクトを参照する変数は、派生クラス型変数でも基底クラス型変数でも構わない。どちらの型の変数を使う場合でも、オーバライドされたほうのメソッドが呼び出される。つまり変数の型で呼ばれるメソッドが決まるのではなく（型依存）、実際に参照しているオブジェクトによって、呼ばれるメソッドが決まるのである（オブジェクト依存）。

VB.NET でオーバライドをするためには、基底クラスのメソッドには Overridables 修飾子がついていなければならない。また、派生クラスではメソッドに Overrides 修飾子を付けなければならない。但し、Overrides 修飾子を付けたメソッドを更に派生クラスでオーバライドすることも出来る。

または、基底クラスが抽象クラス（MustInherit 修飾子付きクラス）で、そのクラスのメソッドに MustInherit 修飾子が付いている場合も、派生クラスで Overrides 修飾子を付けてオーバライドすることができる。（オーバライドして実装しないとそのインスタンスが作

れない。)

つまり、オーバライドするメソッドの修飾子の組み合わせは、Overridable － Overrides が、MustOverride － Overrides、または、オーバライドされたメソッドをさらにオーバライドする Overrides － Overrides である。

Not Overridable はオーバライドしたメソッド（Overrides 付きメソッド）をもうこれ以上オーバライドされたくない時につけます。

なお、オーバライドしたメソッドから、同名の基底クラスのメソッドを呼び出すためには、MyBase キーワードを付けて自己のメソッドと区別する。

4.1.4.6 Python

(1) 概要

Python は Guido van Rossum 氏というオランダ人が開発したオブジェクト指向プログラミング言語であり、オープンソースのフリーウェアとして開発が進められている。日本では日本語の文献が少ないこともあり、それほど知られていないが、Microsoft 社の.NET の中でも Python は取り込まれることが決定しており、欧米では比較的よく知られたプログラミング言語である。また、Redhat 系の Linux ではインストーラや環境設定ツールの開発言語として利用されており、Redhat Linux では標準的に利用できるようになっている。その他、米国の NASA において管制システムの開発や、日本の高エネルギー加速器研究機構の加速器制御システムの開発などに利用されている例もある。最近では、映画のコンピュータグラフィックスの作成に利用された例もあり、幅広い分野で利用されているプログラミング言語である。

なお、フランスでは、工学系モデリング言語とよく似た考え方に基づき、原子力工学を含む工学系の数値シミュレーションの共通プラットフォームの開発を進めている。この計画で開発が進められている共通プラットフォームは SALOME と呼ばれており、この SALOME では、C++で書かれたソルバーや可視化ツールなどの要素を結合するための言語（グルー言語）として、Python を採用している。

以上のように、Python は、科学技術の分野で利用例が比較的多いことは特筆すべき点である。

Python の開発は 1990 年ごろから開始され、開発者の Guido van Rossum 氏 は教育用のプログラミング言語「ABC」の開発の経験を生かし、「シンプル」で「習得が容易」という目標に重点をおいて設計を行った。多くのスクリプト言語ではユーザの目先の利便性を優先して色々な機能を言語要素として取り入れる場合が多いが、Python では言語自体の機能は最小限に押さえ、必要な機能は拡張モジュールとして追加する、ということが 基本設計方針となっている。また、同氏は、「ABC 言語は教育用として優れていたが実用に適さなかった」という反省を込めて、習得しやすいと同時に、強力な機能を兼ね備えた実用性のある言語として Python の開発を進めたため、実用性も考慮された設計となっている。実際、米国では Python を学習用言語とするプログラミング教育プロジェクト CP4E(Computer Programming For Everybody)が、DARPA(米国防衛高等研究企画庁)からの出資を得て活動を行っている。

Python の主な特徴としては一般に以下のようなことが挙げられる。

- オープンソース(商用利用も含め、無償で利用・再配布可)
- マルチプラットフォーム(Unix、 Windows、 Macintosh、 etc...)
- インタプリタ型言語
- オブジェクト指向言語

- モジュール機構
- リストや辞書、複素数など、豊富な組み込みデータ型
- 例外処理
- クラスや関数などもオブジェクトとして扱うことができる
- マルチスレッド対応
- 豊富な拡張ライブラリ
- C/C++ による拡張が簡単
- C/C++ アプリケーションへの組み込みが簡単

また、Python には、テキスト処理から、グラフィカルインターフェイス、インターネットプロトコルに至るまで、豊富なライブラリが標準で添付されている。以下が標準添付されているライブラリーの一部である。

- Tkinter :
  Tcl/Tk の Tk ライブラリを使用した GUI ライブラリ。
- ftplib、 httplib :
  ftp、 http クライアント機能を提供。他にも telnet、nntp、 gopher クライアント等のライブラリを提供。
- xmllib、 htmllib :
  XML、 HTML のパーサ（構文解析）用ライブラリ。

Python はグルー言語としての特徴を持っており、C や C++で書かれたサブルーチンを容易に取り込むことが可能となっている。このため、C 言語等で書かれたライブラリが非常に多く取り込まれている。標準モジュール以外にも多くのモジュールが世界中で開発・公開されており、多くの GUI ライブラリや Oracle、 SyBase などの商用のリレーショナルデータベースは、もちろん、PostgreSQL や MySQL 等のオープンソースのデータベースにアクセスするためのモジュールも公開されている。

また、Python は、さまざまな Operating System（OS）で動作しており、OS に依存しないプログラムがかけるように工夫がされている。例えば、ファイルの取り扱いなどは、OS に依存せざるを得ない部分であるが、このような OS 依存のモジュールは、OS 非依存のモジュールでラッピングされており、あえて OS 非依存のモジュールを直接呼び出すといったことをしない限り、OS が異なっても、ソースファイルをまったく変更することなく動作する。また、Python は、Java 等と同様に、中間ファイル（拡張子.pyc）を生成してから実行するという方式をとっており、この中間形式ファイルには、OS 依存性はなく、UNIX、Windows、Macintosh 等異なる OS で作成されたものを共有することができる。このため、他の OS で作成した中間実行ファイルだけコピーしてきて実行することも可能である。

Python は、インタプリタであり、基本的に Fortran 等のコンパイラ言語に比べると計算

速度は遅く、一般に、数値計算には向かないと考えられている。しかしながら、Python は C 言語との結合が簡単に行えるという利点があるため、大きな配列に対する数値計算部分だけを C 言語で記述して、数値計算用に Python を拡張したモジュールが米国のローレンス・リバモア国立研究所を中心として開発されており、これは Numerical Python と呼ばれている。この、Numerical Python は、Python で大規模な配列を取り扱う場合の標準的な拡張モジュールとなっている。

(2) Python のプログラム例

Python の文法は一般的なもので、親しみやすいものとなっている。手続き型言語になじんでいる人にとっても違和感はない。例えば、Hello World プログラムは以下のようになる。

```
print "Hello World!"
```

また、Python はインタプリタ型の言語であり、スクリプトファイルを読みこんで実行するだけでなく、以下のように対話モードで使用することも可能である。

```
% python
Python 1.5.2 (#15、 Apr 14 2000、 12:44:21) [GCC 2.7.2.3] on linux2
Copyright 1991-1995 Stichting Mathematisch Centrum、 Amsterdam
>>> print "Hello World!"
Hello World!
>>> a = 1
>>> b = 2
>>> print a+b
3
```

ただし、文法に関しては、ひとつ変わった特徴があり、「ブロックの範囲をインデントで決定する」というものである。通常、人間が見たときのプログラムの可読性を向上するために、インデントを使ってブロックを表すが、このインデントはコンピュータにとっては何の意味も持たず、例えば、C 言語では「{ }」、Fortran 等では end 文等を使って、このブロックを表す。しかしながら、Python ではインデント、つまり行頭のスペースとタブの数で指定する。以下は、引数 a が 100 より大きければ 1 を、それ以外の場合 0 を返す関数 foo を定義する Python のプログラムである。

```
def foo(a):
    if a > 100:
        return 1
```

```
else:
    return 0
```

このような方式を採用しているプログラミング言語は多くはなく、あまり見かけない書き方であるが、誰が書いても統一されたフォーマットとなるのでプログラムの保守が容易になる、というメリットが得られる。また、この程度のインデント処理であれば、通常、エディタで対応できるので、編集が著しく面倒になるということもない。

一方、Python はリスト、 辞書といった高レベルなデータ構造を組込み型として備えており、このようなデータ構造を利用すれば、複雑な処理を簡潔に記述することができる。

```
aList = [1、2、3、4]  # リストを作成
print aList[1:4]    # aList の要素のうち、1番目から3番目まで印刷
print aList[2:]     # aList の2番目以降の要素を印刷

aDict = {'first':'No.1'、  2:'No.2'}  # 辞書を作成
print aDict['first']      # 'No.1'と印刷
aDict[3] = 'No.3'         # 辞書に 3->'No.3'を登録
```

Python は伝統的なプログラミングモデル以外にも、モジュール機構・オブジェクト指向・例外処理のような、現代的なプログラミング言語に必要な機能を提供している。
特に Python のオブジェクト指向的な機能は、多重継承や動的な型情報の参照を含む、非常に柔軟性に富んだシステムとなっている。

また、Python は関数型プログラミングの要素も取り込んでおり、lamda というキーワードを用いて、無名の関数を定義することも可能であり、map、reduce 等、関数型プログラミングでよく利用される関数も標準で整備されている。

(3) f2py

第 4.1.3.1 節で、スクリプト言語と C、C++言語を結合する SWIG というツールを紹介したが、Python に関しては、Fortran とと結合させるためのインターフェイスプログラムとしては、報告者が調査した範囲では２つ存在する。ひとつは、米国のローレンス・リバモア国立研究所で開発された Pyfort と呼ばれるインターフェイス、もうひとつは、エストニアの Pearu Peterson 氏が中心となって開発を進めている f2py と呼ばれるツールである。なお、この Peterson 氏自身は、この f2py を航空宇宙工学へ適用することを目的としているようであり、米国スタンフォード大学との共同研究の一環として開発しているようである。

f2py の開発者によれば、現在の Pyfort は基本的な機能確認だけを行った段階で、Fortran

の内部処理に詳しくなければ使いこなせないとのことである。f2py ではなるべく少ない情報で、容易に Fortran プログラムを結合できるように、整備を進めているとのことである。しかしながら、f2py も、現在、発展途上にあるようで、報告者がテストした感想としては、動作が安定しているとは言い難いが、Pyfort よりも高機能で使いやすいと感じている。今後発展していく可能性も高いと思われるため、ここでは f2py について調査することにした。

なお、残念がなら、f2py は、現在のところ UNIX にのみ対応しており、Red Hat Linux 上でのテストのみを行うことし、Windows での動作確認は行っていない。

f2py のテスト

f2py のマニュアルに掲載されているサンプルプログラムを実際に実行し、マニュアル通りの動作が得られることを確認した。

このサンプルでは、bar という Fortran の関数（図 4.1.3.1.d-2）と、foo という Fortran のサブルーチン（図 4.1.3.1.d-3）を Python から利用できるようにするためのものである。bar は２つの引数を受けてその和を求めるものであり、foo はひとつの引数を与えその値に 5 を足して返すという単純なものである。この関数とサブルーチンを持つ foobar という拡張ライブラリーを作成した。単純な例ではあるが、このテストにより、Python から、他の Python のモジュールのように、Fortran の関数を呼び出すことができることが確認できた。

f2py では、拡張子「.pyf」をもつインターフェイス情報を記述するファイルを自動的に生成してくれるため、Python と結合するために新たに作らなければならないファイルは特にない。ただし、引数が入力として扱うか出力として扱うかを指定する部分だけは、自動的に判断できないので、ユーザーが指定する必要がある。この部分は、Fortran90 以降で用意された INTENT 文で指定するだけであり、Fortran90 の知識があれば使えるように配慮されている。

なお、f2py の出力としては、foobarmodule.so というファイルが得られる。この Fortran で作成されたモジュールは、通常の Python のモジュールと同様に、「import foobar」とするだけで利用可能となる。

f2py に関して行ったテストの詳細手順を以下にまとめる。

①インターフェイスファイルの作成

以下のコマンドを実行することにより、foo.f と bar.f が含まれた拡張ライブラリーfoobar を作成するためのインターフェイスファイル（foobar.pyf）が自動的に生成される。

```
% f2py foo.f bar.f -m foobar -h foobar.pyf
```

②Makefile の作成

以下のコマンドを実行することにより、foobar.pyf から、拡張ライブラリーをコンパイル

するための Makefile が自動的に生成される。Makefile のファイル名は、Makefile-（拡張ライブラリー名）となる。

```
% f2py foobar.pyf
```

③拡張ライブラリーのコンパイル

前項で作った Makefile を使ってコンパイル（make）を実行する。foobarmodule.so というライブラリーが生成される。

```
% make -f Makefile-foobar
```

④拡張ライブラリーの利用

作成した拡張ライブラリを利用するためには、Python で、「import foobar」とするだけでよい。これは、Python で書かれたライブラリーと全く同じ扱いであり、基本的にユーザーは Fortran で書かれていることは意識する必要がない。

図 4.1.3.1.d-1 に作成した拡張ライブラリーの利用例を示す。なお、「>>>」は Python のインタラクティブモードでのプロンプトであり、「>>>」の行が実際に打ち込んだコマンドである。プログラムとして実行させる場合も同様に書けばよい。

```
% python
[GCC 2.96 20000731 (Red Hat Linux 7.1 2.96-98)] on linux2
Type "help"、 "copyright"、 "credits" or "license" for more information.
>>> import foobar                #ライブラリの読み込み
>>> print foobar.bar(2、3)       #Fortran の関数 bar（和算）を呼び出す
5                                #2+3=5 であるので正しい答え
>>> from Numeric import *
>>> a = array(3)                 #数値を書き換える場合は Numeric.array を使う
>>> print a                      #初期値の確認
3
>>> foobar.foo(a)                #Fortran の関数 foo（5 を加算）を呼び出す
>>> print a
8                                #3+5=8
>>> foobar.foo(a)
>>> print a
13                               #8+5=13
```

図 4.1.4.6-1　作成した foobar 拡張ライブラリーの利用例

```
! Fortran file bar.f:
      function bar(a、b)
      integer a、 b、 bar
      bar = a + b
      end
```

図 4.1.4.6-2　bar.f

```
! Fortran file foo.f:
      subroutine foo(a)
      integer a
      a = a + 5
      end
```

図 4.1.4.6-3　foo.f

4.1.4.7 Ruby

Ruby 言語は、まつもとゆきひろ氏という日本人が開発したオープンソースのオブジェクト指向スクリプト言語である。日本人が開発しているということもあり、最近日本でのユーザーが急速に増えている。現在最新のバージョンは、Ruby1.6.6 であり、現在、Ruby1.7 が開発中である。Ruby が登場してからの歴史を以下に示す。

1993 年 2 月 24 日	Ruby 誕生
1994 年 11 月	実用的な言語に育つ。NetNews で協力者募集
1995 年 12 月	fj.sources で初の一般公開（バージョン 0.95）
1996 年 12 月 25 日	Ruby 1.0 リリース
1997 年 12 月	オンラインソフトウェア大賞'97 Linux 部門入選
1998 年 12 月 25 日	Ruby 1.2 リリース
1999 年 8 月 13 日	Ruby 1.4 リリース
1999 年 11 月	解説書「オブジェクト指向スクリプト言語 Ruby」刊行
2000 年 9 月	Ruby1.6 リリース

Ruby は、多くの他のオブジェクト指向言語の良いところを取り入れて設計されており、設計の方針は、「使っていて楽しく、かつ、本格的なプログラミングができる」である。Ruby の名前は、有名なスクリプト言語である Perl（Pearl：真珠）に由来しており、Perl の特徴を多く持つ。しかしながら、Perl は読みにくいコードとなることが多い、あるいは、オブジェクト指向の機能が後付であるためオブジェクト指向的なプログラムを書きにくいという批判を受けることが多い。

これに対して、Ruby では、Eiffel や Ada の構文が参考にされている点や、Smalltalk のように、基本的なデータ型を含めてすべてがオブジェクトとして扱われている点から、一般に読みやすいコードとなり、また、Ruby 自体がオブジェクト指向で設計されているため、オブジェクト指向プログラミングに適していると考えられている。Ruby は、Perl の実用性と Smalltalk の純粋なオブジェクト指向プログラミング言語の特徴を併せ持つため、「Perl 的な Smalltalk」と呼ばれることもある。また、Ruby 作者自身、Python にも影響されたと述べており、Ruby には、Python の特徴も取り入れらている。

Ruby の特長としては以下のようなことがあげられる。

- シンプルな文法
- 普通のオブジェクト指向機能(クラス、メソッドコール等)
- 特殊なオブジェクト指向機能(Mixin、 特異メソッド等)
- 演算子オーバーロード
- 例外処理機能
- イテレータとクロージャ

- ガーベージコレクタ
- ダイナミックローディング（アーキテクチャによる）
- 移植性が高い．多くの UNIX 上で動くだけでなく、DOS や Windows、 Mac、BeOS などの上でも動く

Ruby には他のオブジェクト指向言語が提供する基本的な OO 機能（クラス、メソッドなど）がすべて含まれいる。また、継承、ポリモルフィズム、特異メソッド、Mix-in といった機能も、Ruby の言語仕様として実装されている。また、整数や実数などの基本的な型は特別扱いされるオブジェクト指向言語も多いが、Ruby では、クラス、整数、文字列、配列、およびコード・ブロック等が、すべてオブジェクトとして扱われている。このような点から、Ruby は純オブジェト指向言語と呼ばれている。

Ruby は 当初、NEWS-OS (Sony 版 BSD) 上で作成されたが、以後は SunOS 用、Linux 用に作成され、多くのプラットフォーム間での移植性を持つ。Ruby は、ほとんどのバージョンの UNIX、MS-DOS、Windows、Macintosh、BeOS、OS/2 などで実行することが可能である。

Ruby が Perl や Python よりも優れている点に、その読みやすさがあります。Ruby の持つ単一継承機能と、純オブジェクト指向言語により、そのコードは通常よりもはるかに明快で簡単です。

複雑で扱いにくい構文を避けるために、Ruby では単一継承のみを提供しており、サブクラスが複数の親クラスから派生する多重継承は提供していない。多重継承に相当する操作が必要な場合には、モジュールと Mixin の機能を利用することになる。

(1) Ruby と Python の比較

Ruby と Python は、その設計と目標はよく似ているが、一般に Ruby は Python に比べて、以下のような違いがあると考えられている。

- 構文の構造が Python よりも保守的である。
- オブジェクトの属性にアクセスするために "self" を記述する必要がない。
- Python のようにオブジェクト属性をデフォルトで公開しない。
- Python とは異なり、関数とメソッドがファーストクラス・オブジェクトではない。
- 短整数と長整数を自動的に変換する。
- タプルがない。
- データはすべてクラス・インスタンスである。

ただし、Ruby も最近は海外でも有名になりつつあり、Python においても、Ruby の特徴を意識した開発が現在も進められている。一般的に、どちらが優れているという判断は難しいが、ともに、オブジェクト指向スクリプト言語であり、コードの読みやすさ、実用性を設計目標としている点では、よく似たプログラミング言語であると考えられる。

4.1.4.8 Fortran90

FORTRAN については、文献 4.1.4.8-1 に FORTRAN90 によるオブジェクト指向プログラミングの可能性について述べられている。ここではその内容について調査した。

調査の結果、以下のことがわかった。

- 言語分類上、FORTRAN90 はオブジェクト指向言語ではないと称されるが、そのように称されるのは FORTRAN90 がオブジェクト指向の仕様をフルサポートしていないためである。
- クラス（class）、抽象データ型（abstract data type）、カプセル化（encapsulation）、関数多重定義（function overloading）機能は FORTRAN90 で直接サポートされている。
- 継承（inheritance）、多様性（polymorphism）については FORTRAN90 で直接サポートされていないが、同等の機能を模擬することは可能である（これらについては FORTRAN2000 において直接サポートされる）。
- クラスは、抽象データ型（属性）と手続き（メソッド）を MODULE 文に組み込むことで実現される。手続きは CONTAINS 文の中にまとめて内包されるのが一般的のようである。
- 抽象データ型は、TYPE 文を用いて構造型名（INTEGER や CHARACTER 等、基本データ型をユーザの要求通りに組み合わせて作った新しいデータ型の名前）を定義することで実現される。
- カプセル化は、MODULE 文（クラス）に内包する構造体要素（構造体を形成するデータ要素）や手続きに対し、必要に応じて PRIVATE 宣言を設けることで実現される。
- 継承は、各 MODULE 文（クラス）に USE 文を用い、この USE 文にスーパークラスを定義する MODULE 文を組み入れることで実現される。これにより、サブクラスが生成されるが、そのサブクラス内でデータ型を新たに定義する場合、スーパークラスのデータ型（構造体型の名前）も記述する必要がある、といった煩雑さが生じる。

文献 4.1.4.8-1 には FORTRAN90 によるサンプルプログラムが C++によるプログラムと比較の形で掲載されている。これら文献に掲載されているサンプルの一部を図 4.1.4.8-1～5 に示す。

①サンプル 1　：　クラス生成

このサンプルは FORTRAN90 によるクラス生成の方法を C++による方法と比較することにより説明するために設定されたものである。

- FORTRAN90 バージョン　：　図 4.1.4.8-1（1/3～3/3）
- C++バージョン　：　図 4.1.4.8-2（1/2～2/2）

- クラス名は Personnel_class で、これは MODULE 文で宣言されている（図 4.1.4.8-1 (1/3)）。C++では Personnel という名前でクラス名が定義されている（図 4.1.4.8-2 (1/2)）。
- 図 4.1.4.8-1 (3/3) は FORTRAN90 によるメインプログラムである。ここでは、USE 文により Personnel_class を用い、まず構造型名 Personnel の構造体 person（これがオブジェクトに相当）を生成する。次に、構造体 person に対する記録（記録数、姓、名）を生成する（ここでオブジェクト Person は属性を持つことになり、インスタンスが生成されたことになる）。さらに、姓、名を出力し、それを消去し、現時点での記録数を出力する。

②サンプル 2　：　継承の方法

このサンプルはFORTRAN90による継承の方法をC++による方法と比較することにより説明するために設定されたものである。

- FORTRAN90 バージョン　：　図 4.1.4.8-3 (1/3～3/3)
- C++バージョン　：　図 4.1.4.8-4 (1/2～2/2)
- 図 4.1.4.8-3 (1/3) の MODULE 文で定義されている Student_class は、クラス Personnel_class をスーパークラスとするサブクラスである。C++では Personnel をスーパークラスとし、そのサブクラスとして Student が定義されている（図 4.1.4.8-4 (1/2)）。
- 図 4.1.4.8-3 (3/3) は FORTRAN90 によるメインプログラムである。ここでは、USE 文により Student _class を用い、まず構造型名 Student の構造体 studentA（これがオブジェクトに相当）を生成する。次に、構造体 studentA に対する記録（記録数、姓、名）を生成する（ここでオブジェクト Personnel は属性を持つことになり、インスタンスが生成されたことになる）。続いて、studentA に授業科目情報 (MATH) を追加する（インスタンスの属性が変更したことに相当）。さらに、姓、名、授業科目名を出力する。
- 図 4.1.4.8-3 (2/3) は FORTRAN90 によるサブクラス Student_class の具体的な構造を記したものである。USE 文でスーパークラス Personnel_class を継承し、type Student ～end type Student でサブクラス Student_class のデータ型を定義している。その際、type(Personnel) :: personnel というように、スーパークラス Personnel_class のデータ型も記述している。
- 図 4.1.4.8-4 (1/2) は C++によるサブクラス Student の具体的な構造を記したものである。Class Student : public Personnel でスーパークラス Personnel を継承し、int nclasses と char *classes[10]の 2 行でスーパークラス Personnel のデータ型に対し追加すべきデータ型を定義している。すなわち、FORTRAN90 バージョンと異なる点は、サブクラスでそのデータ型を定義する際、スーパークラスのデータ型に追加すべきデータ型のみ追記するだけでよく、スーパークラスのデータ型まで新たに記述する必要がな

い、ということである。

③サンプル3　：　多様性

このサンプルはFORTRAN90による多様性について説明するために設定されたものである。

- FORTRAN90バージョン　：　図4.1.4.8-5（1/3～3/3）
- 図4.1.4.8-5（1/3）のMODULE文で定義されているpoly_Personnel_classは、クラスStudent_class、Teacher_class（図の掲載は省略。Student_classと同様、Personnel_classのサブクラス。）を多重継承するサブクラスである。
- 図4.1.4.8-5（3/3）はFORTRAN90によるメインプログラムである。ここでは、USE文によりpoly_Personnel_classを用いて、まず構造型名Studentの構造体studentA、構造型名Teacherの構造体teacherA、構造型名poly_Personnelの構造体person（これら構造体がオブジェクトに相当）をそれぞれ生成する。次に、同一の総称手続き名newを用いて、構造体studentAに対する記録（記録数、姓、名）、構造体teacherAに対する記録（記録数、姓、名、給与）を生成する（ここでオブジェクトstudentA、teacherAはともに属性を持つことになり、2つのインスタンスが生成されたことになる）。続いて、インスタンスstudentAを総称手続きpolyによりオブジェクトpersonに割り当て、姓、名を出力する。全く同様に、インスタンスteacherAを同じく総称手続きpolyによりオブジェクトpersonに割り当て、姓、名、給与を出力する。

以上、FORTRAN90によるオブジェクト指向プログラミングの可能性について、簡単なプログラムを例に検討した。ここで示した例からもわかるように、FORTRAN77からFORTRAN90に機能拡張されたMODULE文、USE文、TYPE文、interface文等を活用することにより、クラス生成、継承の方法、多様性といったオブジェクト指向のキーとなる特徴を含めてC++と同等の機能を持つプログラミングが原理的には可能であるといえる。

ただし、上述したように、サブクラス内でデータ型を新たに定義する場合、FORTRAN90ではスーパークラスのデータ型（構造体型の名前）も記述する必要がある点がC++とは異なる。本研究のテーマである次世代解析システムの開発の観点からは、このことは例えば、物性データをクラス階層化する際に、その物性データの特性によってはユーザ理解や保守・管理の容易さへの要求も含め、クラス階層の深さが深くならざるを得ないような場合に問題が生じ得ると考えられる。このような場合にFORTRAN90を活用することは、プログラミングが大きく煩雑になる恐れがあり、プログラムの保守・管理の容易化の観点からは必ずしも望ましい方策とはいえない。

一方で、元来FORTRANは科学技術計算において豊富な実績を有することが従来から広く知られている。したがって、FORTRANを用いたオブジェクト指向相当のプログラミングを試みる際は、例えば深いクラス階層化を必要とせず、かつ高速の数値演算を行う箇所に限定してFORTRAN90を活用し、その他、例えば物性データのように、数値演算処理と

は直接関係しないような部分については C++等の他の言語の特徴を活用し、これらをスクリプト言語により有機的に結合することにより、FORTRAN を含め個々の言語の特長を活かしつつ、プログラム全体のバランスおよび保守・管理の容易性に優れたプログラミングを目指すといった方法が一つの有望な考え方ではないか、と考えられる。その際、C++と FORTRAN90 の親和性についての配慮が必要となるが、FORTRAN を C++から呼び出すことについて検討した例を参考にすれば [4.1.4.8-2]、FORTRAN77 の SUBROUTINE 文や FUNCTION 文といった副プログラムについての呼び出しは可能であり、両言語で構造が大きく異なるデータの扱いについても、FORTRAN90 に TYPE 文や MODULE 文の機能が設けられたことで両者のリンクは可能と考えられる。したがって、FORTRAN90 の C++との親和性については特に問題にならないといえる。

また、同じ FORTRAN でも FORTRAN77 から FORTRAN90 に変更することで、例えば FORTRAN77 では副プログラム間で共通して用いるデータを COMMON 変数として定義する場合、それを使用するすべてのサブルーチンに明記する必要があるが、FORTRAN90 におけるクラスで用いるデータ構造は他のクラスとは独立した存在であるため、異なるクラス同士でのデータ構造の共有といったことに留意する必要は基本的にない。当然ながら、継承関係にあるクラス間においては、スーパークラスが有するデータ構造をそのサブクラスも共有することになるが、そのための記述にあたっては、上述のようにスーパークラスのデータ型はサブクラスに記述する必要があるものの、そのデータ構造の詳細までは書く必要がない。したがって、副プログラムやクラスといった、プログラム全体を構成するいわゆるモジュールの数が増えるに従い、そのプログラミング作業や作成したプログラムの保守・管理作業に要する負荷が FORTRAN90 の利用により大きく緩和されることが期待される。

```
      module Personnel_class
      implicit none
      private :: init_Personnel, term_Personnel, print_Personnel
! Define Personnel type
      type Personnel
         private
         integer :: ssn
         character, dimension(:), pointer :: firstname, lastname
      end type Personnel
! Number of database records
      integer, save, private :: NUM_FILES = 0
      interface new
         module procedure init_Personnel
      end interface
      interface delete
         module procedure term_Personnel
      end interface
      interface print
         module procedure print_Personnel
      end interface
      contains
         subroutine init_Personnel(this,s,fn,ln)
! Constructor
         type (Personnel), intent (out) :: this
         integer, intent (in) :: s
         character*(*), intent (in) :: fn, ln
         this%ssn = s
         allocate(this%firstname(len(fn)),this%lastname(len(ln)))
         call strcpy(this%firstname,fn)
         call strcpy(this%lastname,ln)
         NUM_FILES = NUM_FILES + 1
         end subroutine init_Personnel
!
         subroutine term_Personnel(this)
! Destructor
         type (Personnel), intent (inout) :: this
         deallocate(this%firstname,this%lastname)
         NUM_FILES = NUM_FILES - 1
         end subroutine term_Personnel
```

図 4.1.4.8-1　サンプル 1：FORTRAN90 によるクラス定義

```
        subroutine print_Personnel(this,printssn)
        type (Personnel), intent (in) :: this
        logical, optional, intent (in) :: printssn
        if (present(printssn)) then
           if (printssn) write (*,'(i2,a2)',advance='no')
     &     this%ssn,': '
        endif
        Print *, this%firstname, ' ', this%lastname
        end subroutine print_Personnel
!
        function getssn_Personnel(this) result(ssn)
        type (Personnel), intent (in) :: this
        integer :: ssn
        ssn = this%ssn
        end function getssn_Personnel
!
        integer function get_num_files()
        get_num_files = NUM_FILES
        end function get_num_files
!
        subroutine strcpy(s,c)
        character, dimension (:), intent (out) :: s
        character*(*), intent (in) :: c
        integer :: i
        do i = 1, max(size(s),len(c))
           s(i) = c(i:i)
        enddo
        end subroutine strcpy
     end module Personnel_class
```

図 4.1.4.8-1　サンプル 1 : FORTRAN90 によるクラス定義

```
        program personnel_test
        use Personnel_class
        type (Personnel) :: person
        call new(person,1,'PAUL','JONES')
        call print(person)
        call delete(person)
        Print *,'NUM_FILES=',get_num_files()
        end program personnel_test
```

図 4.1.4.8-1　サンプル 1 : FORTRAN90 によるクラス定義

```
// ************************
// ****  personnel.h  ****
// ************************

#include <stream.h>

class Personnel {
  static int NUM_FILES;
  char *firstname, *lastname;
 protected:
  int ssn;
 public:
  Personnel(const int s, const char *fn, const char *ln);
  ~Personnel();
  virtual void print(const int printssn = 0);
  virtual int getssn();
  static int get_num_files();
```

図 4.1.4.8-2　サンプル 1：C++によるクラス定義

```
//******************************
//****   personnel.cc   *******
//******************************

#include <stream.h>
#include <string.h>
#include "personnel.h"

// Initialize static class member
// Number of database records
int Personnel::NUM_FILES = 0;

// Constructor
Personnel::Personnel(const int s, const char *fn, const char *ln)
{
  ssn = s;
  firstname = new char[strlen(fn)+1];
  lastname  = new char[strlen(ln)+1];
  strcpy(firstname, fn);
  strcpy(lastname, ln);
  NUM_FILES++;
}

// Destructor
Personnel::~Personnel()
{
  delete firstname;
  delete lastname;
  NUM_FILES--;
}

void Personnel::print(const int printssn)
{
  if (printssn)
      cout << ssn << ": " << firstname << ' ' << lastname << endl;
  else
      cout << firstname << ' ' << lastname << endl;
}

int Personnel::getssn() { return ssn; }

int Personnel::get_num_files() { return NUM_FILES; }
```

図 4.1.4.8-2　サンプル 1：C++によるクラス定義

```
      module Student_class
! bring Personnel_class into scope
      use Personnel_class
      implicit none
      private :: Personnel,init_Student,term_Student,print_Student
      private :: getssn_Personnel, getssn_Student
! define String type
      type, private :: String
         character*1, dimension(:), pointer :: stringptr
      end type String
! define Student type
      type Student
         private
         type (Personnel) :: personnel
         integer :: nclasses
         type (String), dimension (10) :: classes
      end type Student
      interface new
         module procedure init_Student
      end interface
      interface delete
         module procedure term_Student
      end interface
      interface print
         module procedure print_Student
      end interface
      interface getssn
         module procedure getssn_Student
      end interface
      contains
         subroutine init_Student(this,s,fn,ln)
! Student class constructor
         type (Student), intent (out) :: this
         integer, intent (in) :: s
         character*(*), intent (in) :: fn, ln
         call new(this%personnel,s,fn,ln)
         this%nclasses = 0
         end subroutine init_Student
!
         subroutine term_Student(this)
! Student class destructor
         type (Student), intent (inout) :: this
         integer :: i
         call delete(this%personnel)
         do i = 1, this%nclasses
            deallocate(this%classes(i)%stringptr)
         enddo
         end subroutine term_Student
```

図 4.1.4.8-3　サンプル 2：FORTRAN90 によるクラス継承

```
          subroutine print_Student(this,printssn)
! Print a student file
          type (Student), intent (in) :: this
          logical, optional, intent (in) :: printssn
          integer :: i, j
          call print(this%personnel,printssn)
          if (this%nclasses==0) then
             Print *,'-- Not Enrolled'
          else
             Print *,'-- Enrolled'
             do i = 1, this%nclasses
                do j = 1, size(this%classes(i)%stringptr)
                   write (*,'(a)',advance='no')
     &             this%classes(i)%stringptr(j)
                enddo
             enddo
             Print *
          endif
          end subroutine print_Student
!
          integer function getssn_Student(this)
          type (Student), intent (in) :: this
          getssn_Student = getssn_Personnel(this%personnel)
          end function getssn_Student
!
          subroutine addclass(this,c)
! Add a class to a student file
          type (Student), intent (inout) :: this
          character*(*), intent (in) :: c
          this%nclasses = this%nclasses + 1
          allocate(this%classes(this%nclasses)%stringptr(len(c)))
          call strcpy(this%classes(this%nclasses)%stringptr,c)
          end subroutine addclass
       end module Student_class
```

図 4.1.4.8-3　サンプル 2：FORTRAN90 によるクラス継承

```
          program student_test
          use Student_class
! create a record
          type (Student) :: studentA
          call new(studentA,0,'PAT','SMITH')
          call addclass(studentA,'MATH')
! print a record
          call print(studentA,.true.)
          end program student_test
```

図 4.1.4.8-3　サンプル 2：FORTRAN90 によるクラス継承

```
// ***********************
// ****   student.h   ****
// ***********************

class Student : public Personnel {
  int nclasses;
  char *classes[10];
public:
  Student(const int ssn, const char *firstname, const char
*lastname);
  ~Student();
  void print(const int printssn = 0);
  void addclass(const char *c);
};
```

図 4.1.4.8-4　サンプル 2：C++によるクラス継承

```
//****************************
//**** student.cc   **********
//****************************

#include <stream.h>
#include <string.h>

#include "personnel.h"
#include "student.h"

// Student class constructor
Student::Student(const int s, const char *fn, const char *ln) :
Personnel(s,fn,ln)
{
  nclasses=0;
}

// Student class destructor
Student::~Student()
{
  for (int i=0; i<nclasses; ++i) delete classes[i];
}

// Add a class to a student file
void Student::addclass(const char *c)
{
  classes[nclasses] = new char[strlen(c)+1];
  strcpy(classes[nclasses], c);
  nclasses += 1;
}

// Print a student file
void Student::print(const int printssn)
{
  Personnel::print(printssn);
  if (nclasses == 0)
    cout << "-- Not Enrolled" << endl;
  else {
    cout << "-- Enrolled:" << endl;
    for (int i=0; i < nclasses; ++i) cout << classes[i];
    cout << endl;
  }
}
```

図 4.1.4.8-4　サンプル 2：C++によるクラス継承

```
      module poly_Personnel_class
! bring Student_class into scope
      use Student_class
! bring Teacher_class into scope
      use Teacher_class
      private :: poly_init, assign_student, assign_teacher
      private :: poly_print, poly_getssn, poly_addclass
      private :: poly_updatesalary
! define poly_Personnel type
      type poly_Personnel
         private
         type (Student), pointer :: ps
         type (Teacher), pointer :: pt
      end type poly_Personnel
      interface new
         module procedure poly_init
      end interface
      interface poly
         module procedure assign_student, assign_teacher
      end interface
      interface print
         module procedure poly_print
      end interface
      interface getssn
         module procedure poly_getssn
      end interface
      interface addclass
         module procedure addclass, poly_addclass
      end interface
      interface updatesalary
         module procedure updatesalary, poly_updatesalary
      end interface
      contains
         subroutine poly_init(this)
! Initialize poly_Personnel with null pointers
         type (poly_Personnel), intent (out) :: this
         nullify(this%ps)
         nullify(this%pt)
         end subroutine poly_init
!
         function assign_student(ps) result(pps)
! assign Student to poly_Personnel
         type (poly_Personnel) :: pps
         type (Student), target, intent(in) :: ps
         pps%ps => ps
         nullify(pps%pt)
         end function assign_student
```

図 4.1.4.8-5　サンプル 3 : FORTRAN90 による多様性

```
        function assign_teacher(pt) result(pps)
! assign Teacher to poly_Personnel
        type (poly_Personnel) :: pps
        type (Teacher), target, intent(in) :: pt
        nullify(pps%ps)
        pps%pt => pt
        end function assign_teacher
!
        subroutine poly_print(this,printssn)
! Print poly_Personnel
        type (poly_Personnel), intent (in) :: this
        logical, optional, intent (in) :: printssn
        if (associated(this%ps)) then
           call print(this%ps,printssn)
        elseif (associated(this%pt)) then
           call print(this%pt,printssn)
        endif
        end subroutine poly_print
!
        integer function poly_getssn(this)
        type (poly_Personnel), intent (in) :: this
        if (associated(this%ps)) then
           poly_getssn = getssn(this%ps)
        elseif (associated(this%pt)) then
           poly_getssn = getssn(this%pt)
        endif
        end function poly_getssn
!
        subroutine poly_addclass(this,c)
! Add a class to a student poly_Personnel file
        type (poly_Personnel), intent (inout) :: this
        character*(*), intent (in) :: c
        if (associated(this%ps)) call addclass(this%ps,c)
        end subroutine poly_addclass
!
        subroutine poly_updatesalary(this,sal)
        type (poly_Personnel), intent (inout) :: this
        integer, intent (in) :: sal
        if (associated(this%pt)) call updatesalary(this%pt,sal)
        end subroutine poly_updatesalary
     end module poly_Personnel_class
```

図 4.1.4.8-5　サンプル 3 : FORTRAN90 による多様性

```
      program poly_test
! bring in poly_Personnel_class into scope
      use poly_Personnel_class
      type (Student), target :: studentA
      type (Teacher), target :: teacherA
      type (poly_Personnel) :: person
! initialize student and teacher
      call new(studentA,0,'PAT','SMITH')
      call new(teacherA,2,'JOHN','WHITE',1000)
! assign a student to person and print record
      person = poly(studentA)
      call print(person,.true.)
! assign a teacher to person and print record
      person = poly(teacherA)
      call print(person,.false.)
      end program poly_test
```

図 4.1.4.8-5　サンプル 3 : FORTRAN90 による多様性

4.2 インターフェース技術

4.2.1 プログラム間交換データを記述するための XML

4.2.1.1 概要

3.2.1 項におけるデータ交換の方法として XML のデータの持ち方及び形式を参考にしている。既存のシステム間でデータをやり取りする際、さまざまな制約がある中で最も効率的にしかも拡張性が高いデータ交換を実現する方法としての実績がある。

XML（Extensible Markup Language：拡張可能なマークアップ言語）は、汎用的なデータ記述言語で元は出版業界で使うために文書記述言語として ISO が標準化した SGML（Standard Generalized Markup Language）から派生したものだが、特にインターネット上でのデータ交換を意識して設計されている。したがって、XML は SGML のころからの用途である文書の記述だけでなく、電子商取引データをはじめとしてインターネット上で交換可能なあらゆるデータの記述に使われようとしている。

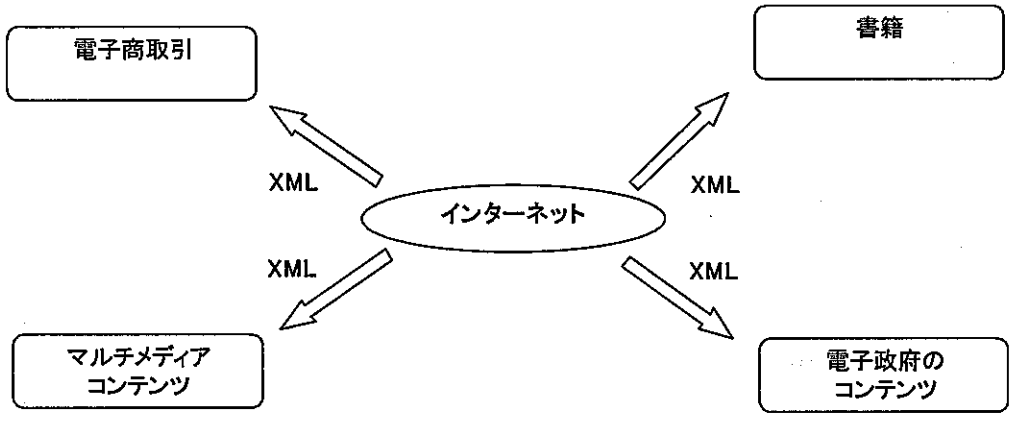


図 4.2.1-1 インターネット（XML 方式）によるデータ交換

4.2.1.2 テキスト形式による記述

XML はテキスト形式で記述されるため、マルチプラットフォーム環境でのデータ交換に適している。

例を使って説明しよう。A 社製のワープロソフトで作成した文書は、そのままでは B 社製のワープロで開けないのはなぜだろうか。それはワープロ文書が各社独自のバイナリ形式で保存されているからだ。異なるベンダのワープロ間で文書を交換するためには、いったんテキストファイルにする必要がある。テキスト形式のデータなら異なるプラットフォ

ームでも読み込み可能だからだ（図 4.2.1-2）。

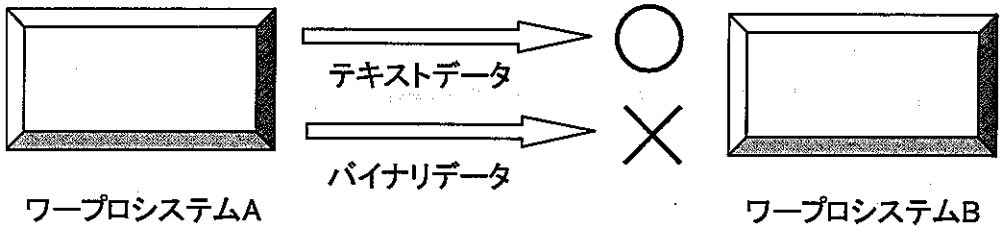


図 4.2.1-2 テキスト形式データによる異種システム間の互換性の確保

同様に、XML 文書もテキスト形式で記述されるため、プラットフォームの壁を越えてデータ交換が可能だ。もっとも、テキスト形式でも文字コードの相違が問題になることがある。XML ではその点でも工夫がなされているが、詳しくは、この連載の別の回で説明しよう。

4.2.1.3 タグによる記述

XML では、文書構造を構成する個々のパーツを「要素」（エレメント：Element）と呼び、要素はタグ（tag）を使って記述する。タグを使った記述方式を採用することで、データの意味やデータ構造を保持したまま、インターネット上でデータ交換ができる。さらに仕様変更や異なるシステム間でのデータ交換に柔軟に対応できるようになる。

こうしたメリットが生まれる理由を以下に説明する。

タグを使ってデータを記述するとは、具体的にいうと要素の始まりを示すタグ（「開始タグ」という）と、終わりを示すタグ（「終了タグ」という）の２つのタグでデータを挟み込んで、要素を表現するということだ。

```
<社員番号>1000</社員番号>
```

上の例でいうと、<社員番号>が開始タグ、</社員番号>が終了タグ、“1000”が要素内容、<社員番号>から</社員番号>までが要素だ。また要素を名前で呼ぶ場合、タグ名“社員番号”に合わせて、“社員番号”要素というように呼ぶ。このように文書を記述するためにタグ付けをすることを「マークアップ（markup）する」という。

HTML もマークアップして文書を記述する言語だが、XML が HTML と大きく異なるところは、使用するタグ名を“住所”とか“お住まい”とか“address”など、ユーザーが決められる点だ。タグ名を要素内容の意味を反映させたものにすれば、その XML 文書を読む人間やシステムに、データの内容だけでなくデータの意味も伝えることができる。

4.2.1.4 入れ子のタグ

さらにXMLは、タグでマークアップされた要素を入れ子構造にすることでデータ構造を表現できる。図4.2.1-3で示されるデータ構造を持つ社員情報を例にして説明する。

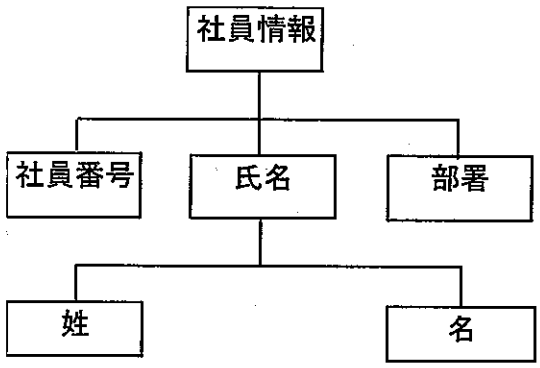


図4.2.1-3 社員情報のデータ構造

図4.2.1-3を見てお分かりのとおり、社員情報を構成する社員番号や氏名などの各要素は、ツリー構造として表現されている。

そこで、ある社員の社員情報をXMLで記述すると次のようになる。

```
<社員情報>
  <社員番号>1000</社員番号>
  <氏名>
    <姓>田中</姓>
    <名>次郎</名>
  </氏名>
  <部署>総務</部署>
</社員情報>
```

図4.2.1-3のツリー構造で表されていた要素間の親子関係が、要素の入れ子構造によってうまく記述されているのにお気付きいただけるだろう。

このようにXMLは、タグ名と要素の入れ子構造というシンプルな方法で、データの意味とデータ構造を保持したままでのデータ交換を可能にしている。

4.2.1.5 ユーザー定義によるタグ名

「XML はタグで記述する」で説明したとおり、XML はタグの名前やタグの階層構造などをユーザーが定義できる。これは特定の用途に特化したタグセットを定義することによって、XML 文法に従う新しいマークアップ言語を定義できるということを意味している。XML に「拡張可能な（Extensible)」という名前が付けられた理由はここにある。

「言語を定義する言語」のことをメタ言語という。XML は、メタ言語機能を持っているため、電子商取引データの記述や、マルチメディア・プレゼンテーションのための記述や、数学で用いる数式の記述など、さまざまなデータを記述する応用言語のベースとなる言語だ。参考までに XML をベースにして作成されたデータ記述言語（本連載では、これを XML 応用言語と呼ぶことにする）のいくつかを以下に示す。

出版・メディア：	Open eBook（電子書籍)、NewsML（ニュースメディア）
科学：	MathML（数式)、CML（化学）
電子商取引：	cXML（電子商取引)、FpML（金融）
マルチメディア：	SMIL（マルチメディアプレゼンテーション)、BML（BS データ放送）

ところで、人間同士で使われる自然言語は、その言語を話す人が多くいて初めてコミュニケーションツールとしての意味を持つ。

同様に、特定の企業間や業界内でひとそろいのタグセット、すなわち特定の XML 応用言語を使用するという約束事があって初めて、その XML 応用言語はコミュニケーションツールとしての存在価値が出てくる。従って、XML のメタ言語機能について語ったが、XML ユーザーの多くは、業界団体が標準化した XML 応用言語を使用することがほとんどで、自分で新しい XML 応用言語を作ることはまずないだろう。

4.2.1.6 XML の適用範囲

XML はデータ記述言語として優れているが、あらゆる電子データを XML 化すればよいわけではない。これまで挙げた XML の特長は、裏返してみれば XML のデメリットにもなる。

例えば XML がテキスト形式であることや、タグでマークアップされていることは、バイナリデータに比べて冗長という意味だ。データの交換量が膨大な場合、これはパフォーマンスの点で大きな問題となる。つまり、XML 導入にあたっては XML のメリット・デメリットを比較・考慮し、XML のメリットを生かせる領域でのみ XML 化を図るべきだ。

4.2.2 データのシリアライゼーション（永続化）

オブジェクト指向プログラミング言語では、プログラミングの中で利用されるオブジェクトをそのままの形でファイルに保存するための機能を持っていることが多い。このような機能をデータの永続化（英語では serialization、marshalling 等）と呼ばれている。データの永続化は、解析コード間のデータの受け渡しにも利用できると考えられるため、個々で、データの永続化についてまとめ、解析コードへの適用可能性について議論する。

4.2.2.1 オブジェクト指向プログラミング言語におけるデータの永続化

オブジェクト指向プログラミング言語では、標準機能としてデータの永続化の機能を装備している。例えば、Java では、Serializable と呼ばれるインターフェイスが用意されており、Python では pickle、Ruby では marshall と呼ばれるモジュールが用意されている。これらの機能を用いることにより、プログラミング言語内で利用されるオブジェクトをそのままファイルに保存したり、復元したりすることが可能となる。

このようなデータ永続化の機能を用いることにより、プログラマは、オブジェクト指向プログラミング言語内で利用される複雑なオブジェクトを、ファイルの形式を全く気にすることなく、ファイルに保存・復元することが可能となる。

4.2.2.2 解析コードにおけるデータの永続化

次に、データの永続化という概念から、既存の解析システムについて分析を行う。例えば、サイクル機構の核特性解析システムで採用されている JOINT システムで利用される PDS ファイルは、核反応断面積データをバイナリファイルに変換したものであり、データの永続化という概念とよく似ていると考えることができる。しかしながら、取り扱う対象は、基本的に核反応断面積データであり、オブジェクト指向プログラミング言語におけるデータの永続化ほどには抽象化されてはいない。

前述のように、ERANOS システムでは、ファイルへの保存に関しては、SABER と呼ばれるツールが用意されており、ERANOS システム中で利用されるデータ（体系、断面積、中性子束等）はすべて SET で取り扱われ、SABER の機能は、アプリケーションから分離されているとの記述から、かなり高度な抽象化が行われていると考えられる。これは、オブジェクト指向プログラミング言語におけるデータの永続化にかなり近い概念となっているのではないかと思われる。

4.2.2.3 JAVAにおけるデータ永続化の利用例

(1) ファイルベースの永続化

ファイルベースの永続化を実現するための方法として、フラットファイル（単なるテキストファイル）を使用するものがある。これは、テキストを使って簡単に表現できる少量のデータを扱う場合に都合がよい。ファイルを作成したアプリケーションを使わなくても、データの確認や修正が可能となる。例えば自作のアプリケーションで、最初に表示したダイアログに有効なユーザーID、ホスト名等、ユーザーが入力しないとアプリケーションの使用を認めない、とする機能があるとする。この場合、これらのフィールドのデフォルト値をフラットファイルに保存(図 4.2.2-1、図 4.2.2-2 参照)しておき、ダイアログでそのファイルを読み込んで該当するフィールドに表示（コマンドプロンプト上で確認　図 4.2.2-3 参照）するようにすると便利である。こうすれば、ユーザーはログオンのたびに内容を入力する必要がなくなるうえ、デフォルト値をテキストエディタで簡単に変更することができる。

フラットファイルは、少量のテキストデータを扱うには適しているが、大量のデータには適していないことが多い。大量のデータを扱う場合には、検索機能が必要となることがよくあるが、フラットファイルでは、シーケンシャルにしか検索できないからである。さらに、データを挿入したり削除したりすることができない。ただし、必要な変更を加えたファイルを新しく作り直せば、挿入や削除と同じ機能を実現することができる。

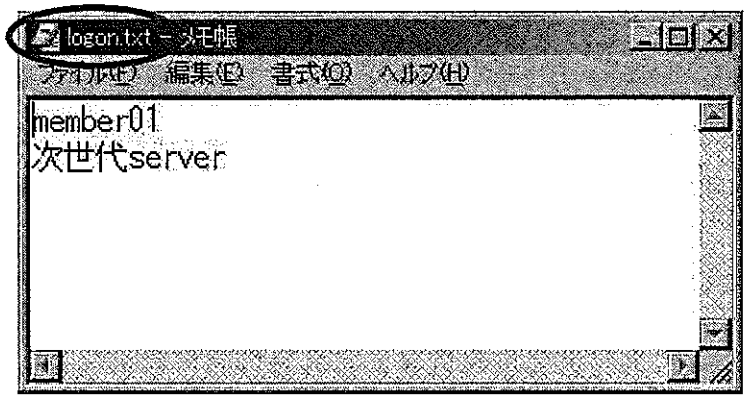


図 4.2.2-1　フラットファイルの内容

```
1  import java.io.*;↓
2  ↓
3  public class ReadHostData {↓
4  ↓
5    public static void main(String[] args) throws IOException {↓
6      String userid = "";↓
7      String hostName = "";↓
8      File file = new File("logon.txt");↓
9      if (file.exists()) {↓
10       String value;↓
11       FileReader fileReader = new FileReader(file);↓
12       BufferedReader reader = new BufferedReader(fileReader);↓
13       value = reader.readLine();↓
14       if (value != null) {↓
15         userid = value;↓
16       }↓
17       value = reader.readLine();↓
18       if (value != null) {↓
19         hostName = value;↓
20       }↓
21       fileReader.close();↓
22       System.out.println("ホスト '" + hostName + "' にユーザー '" + userid +
23           "' としてログオン");↓
24     }↓
25   }↓
26 }↓
27 [EOF]
```

```
ReadHostData
D:¥次世代解析ｼｽﾃﾑ¥ｼﾘｱﾗｲｾﾞｰｼｮﾝ¥まとめ¥source_pro_java2¥chapter2¥ReadHostData>java
ReadHostData
ホスト '次世代server' にユーザー 'member01' としてログオン

D:¥次世代解析ｼｽﾃﾑ¥ｼﾘｱﾗｲｾﾞｰｼｮﾝ¥まとめ¥source_pro_java2¥chapter2¥ReadHostData>_
```

図 4.2.2-3　フラットファイルから読み出した状態

(2)　リレーショナルデータベースを使った永続化

フラットファイルと同様に、シリアライズ*1)は比較的少量のデータを永続化するには適切な方法である。しかし、アプリケーションで大量のデータを作ったり使ったりする場合には、通常はそれらのデータはデータベースに格納する必要がある。データベースを使ってオブジェクトの状態の保存や復元を実現するには、次の 3 つの方法がある。

・オブジェクトをシリアライズしたデータを生のバイナリデータとして保存する。
・通常のオブジェクト／リレーショナルマッピング*2)を使う。
・構造化型を JAVA のクラスにマッピングする。

このうち、1つ目と2つ目の方法は、JDBC1.x と 2.x のどちらでも利用可能だが、3つ目の方法には、構造化型をサポートする DBMS と JDBC2.x 準拠のドライバが必要となる。

DBMS 製品では、データベースの各列のほとんどには特定の型のデータを格納していくことになる。例えば、文字、数値、日付といった型である。しかし通常はこのほかにも、バイナリデータ（つまり任意のバイトデータの集まり）を格納するためのデータ型が用意されていることが多い。このデータ型の列を使えば、シリアライズされたオブジェクトを格納したり取り出したりすることができる。例えば、MYTABLE というデータベーステーブルを定義したとする（図 4.2.2-4 参照）。この中に BINDATA という列があり、ここにはバイナリデータを格納できるものとする。この場合、図 4.2.2-4 のようなコードを使えば、Vector のインスタンス BINDATA 列に格納することができる。シリアライズ可能なオブジェクトであればどれも同様の手順で格納が可能である。格納された状態を図 4.2.2-5、また、格納されたデータベースから読み出した状態を、図 4.2.2-6 に示す。

---

*1) シリアライズ（Serialize：直列化）

Java ではオブジェクトをバイナリに変換、あるいはバイナリをオブジェクトに変換する処理をいう。

*2) オブジェクト／リレーショナルマッピング

クラス内のフィールドを、データベーステーブルの列にマッピングする方法。

```
1  import java.io.*;↓
2  import java.sql.*;↓
3  import java.util.*;↓
4  ↓
5  public class WriteBinary {↓
6  ↓
7    public static void main(String[] args) throws Exception {↓
8  ↓
9      new sun.jdbc.odbc.JdbcOdbcDriver();↓
10 ↓
11 //     Connection connect = DriverManager.getConnection("jdbc:odbc:LocalServer");↓
12     Connection connect = DriverManager.getConnection("jdbc:odbc:BINARYDATA");↓
13     Vector v = new Vector();↓
14     v.addElement("次世代解析システム");↓
15     v.addElement("データの永続化(JAVA)");↓
16     v.addElement(new java.util.Date());↓
17 ↓
18     PreparedStatement pstmt = connect.prepareStatement(↓
19         "UPDATE BINARYDATA SET BINDATA = ? WHERE ITEMKEY = 123456");↓
20 ↓
21     // バイト配列に書き込む出力ストリームを作成する↓
22     ByteArrayOutputStream baos = new ByteArrayOutputStream();↓
23 ↓
24     // ObjectOutputStreamを被せる↓
25     ObjectOutputStream oos = new ObjectOutputStream(baos);↓
26     // Vectorとその中のデータを直列化する↓
27     oos.writeObject(v);↓
28     // ストリームをクローズする↓
29     oos.close();↓
30 ↓
31     // ストリームからバイト配列を取得する↓
32     byte[] binaryData = baos.toByteArray();↓
33 ↓
34     // バイト配列をBINDATA列に格納する↓
35     pstmt.setBytes(1, binaryData);↓
36     pstmt.executeUpdate();↓
37 ↓
38     connect.close();↓
39   }↓
40 ↓
41 }↓
42 [EOF]
```

図 4.2.2-4　シリアライズされたオブジェクトの格納（バイナリデータとして格納）

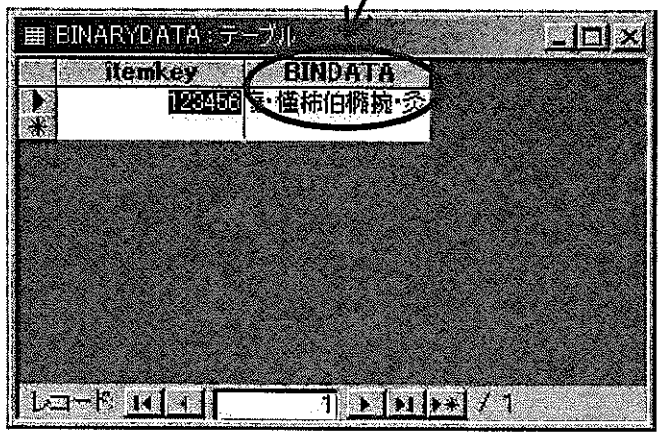


図 4.2.2-5　格納された結果

```
ReadBinary
D:¥次世代解析ｼｽﾃﾑ¥ｼﾘｱﾗｲｽﾞ¥まとめ¥source_pro¥source_pro_java2¥chapter2¥Binary
>Java ReadBinary
次世代解析システム
データの永続化(JAVA)
Tue Jul 02 08:48:28 JST 2002

D:¥次世代解析ｼｽﾃﾑ¥ｼﾘｱﾗｲｽﾞ¥まとめ¥source_pro¥source_pro_java2¥chapter2¥Binary
>
```

図 4.2.2-6　データベースから読み出した内容

永続化は、シリアライズやリレーショナルデータベースといった何らかの既存の機能の助けを借りる必要がある。したがって、これらの機能を使わずに永続化の機能を独自に実装するのは、通常は好ましくない。少量のデータを格納したり復元したりするだけの単純なアプリケーションの場合には、シリアライズを選んでおけばたいていの場合は十分間に合う。一方、リレーショナルデータベースを使えば、より堅牢でスケーラブルなソリューションを実現することができる。

4.2.2.4　Python におけるデータ永続化の利用例

続いて、Python を例に具体的な使用方法について検討する。図 4.2.2-7 にサンプルプログラムを示す。ここでは、Python 言語の標準モジュールである Pickle モジュールを使ってデータを永続化している。

サンプルとしては、核特性解析において、解析コード間で頻繁に受け渡しが必要な断面積を考えた。サンプルプログラムでは、断面積データは、かなり単純化してあり、核分裂断面積、v 値、中性子捕獲断面積、(n、2n)反応断面積しか考慮しておらず、エネルギー群数も 1 群である。しかしながら、反応の種類やエネルギー群数を増やしたり、マトリックスデータを同等に扱うことはそれほど難しくないと考えられる。また、実際の解析ではあまり使われることはないが、断面積に関するメソッドとして、α 値と η 値を計算するメソッドをつけてある。実際には、全断面積や輸送断面積、拡散係数を計算するメソッド等が実装されることになると思われる。

サンプルプログラムでは、以下のような手順を行っている。

(1)　断面積（MicroscopicCrossSection）クラスののインスタンス（xs）を生成する。
(2)　xs の α 値と η 値を計算して表示する。
(3)　xs インスタンスをファイル（xs.pcl）に保存する。
(4)　ファイル（xs.pcl）を復元し、新しいインスタンス（xs2）を生成する。
(5)　xs2 の α 値と η 値を計算して表示する。

ここで、永続化された断面積のインスタンスは、複数の反応断面積に関するデータを持ち、更に、そのデータに密接に関連した計算式（α値とη値を計算するメソッド）に関する情報も持っている。このような複雑な情報をクラスとして定義することで、xs、xs2という変数の形で情報を保持することができる。更に、データの永続化機能を使うことで、簡単にファイルに保存することが可能であることが分かる。

解析コードの連携で、クラスで定義されたデータを受け渡しするように設計しておけば、非常に複雑で膨大な量のデータであっても、参照名（変数）ひとつで受け渡すことが可能である。また、ファイルに永続化して、ファイルの形で渡すことも簡単になる。更に、計算機のメモリ容量が足りない場合には、不要なデータをファイルに永続化して、メモリを解放するということも可能である。オブジェクト指向言語が、ガーベジコレクションの機能を使って、自動的にメモリを解放してくれる。

データの永続化のファイル形式

図 4.2.2-8 には、実際に永続化されたファイルの内容（xs.pcl）を示す。この内容を理解するのは難しいが、開発者・ユーザーとも、永続化のために用意されたモジュールやインターフェイスを使うだけでよく、永続化に用いられるファイル形式を理解する必要はない。

しかしながら、このフォーマットは Python の Pickle モジュール特有のフォーマットであるため、Python を使っている限りは問題は生じないが、他の言語へ渡す場合にはこのフォーマットを認識できるにしなければならない。

永続化に使われるフォーマットとして、XML のように標準化されたものを使うことができれば、異なる言語間でのやり取りも難しくなくなるのではないかと思われる。このような試みは既に行われているようであり、Python 言語では、xml_pickle、xml_objectify といったモジュールの開発が検討されているようである。ただし、現在のところ、実用の段階には至っていないようである。

```
#!/usr/local/bin/python

import pickle

class MicroscopicCrossSection:                                  # ミクロ断面積クラスの定義

    def __init__(self、 fission、 nu_value、 capture、 mu_value、 n2n):

        self.fis = fission
        self.nu  = nu_value
        self.cap = capture
        self.n2n = n2n

    def alphaValue(self):                                       # α値の計算
        return self.cap / self.fis

    def etaValue(self):                                         # η値の計算
        return self.nu * self.fis / self.cap

if __name__ == '__main__':

    xs = MicroscopicCrossSection(1.8、 2.5、 0.5、 0.9、 0.004)          # (1) ミクロ断面積インスタン
ス (xs) 生成
    print xs.alphaValue()、 xs.etaValue()                       # (2) α値とη値の表示

    pickle.dump(xs、 open("xs.pcl"、 "w"))                              # (3) xs の永続化 (ファイル名
xs.pcl)

    xs2 = pickle.load(open("xs.pcl"))                           # (4) xs2 として復元
    print xs2.alphaValue()、 xs2.etaValue()                     # (5) xs2 のα値とη値の表示
```

図 4.2.2-7　Python におけるオブジェクトの永続化の例

```
(i__main__
MicroscopicCrossSection
p0
(dp1
S'mu'
p2
F0.90000000000000002
sS'fis'
p3
F1.8
sS'cap'
p4
F0.5
sS'nu'
p5
F2.5
sS'n2n'
p6
F0.0040000000000000001
sb.
```

図 4.2.2-8　永続化されたファイルの内容 (xs.pcl)

4.2.3 可視化技術

文献 4.2.3-1、4.2.3-2 を参考にしつつ、可視化技術について調査した。

4.2.3.1 可視化技術の現状

可視化技術は計算機を用いた視覚的表現技術であり、その目的・用途によって大きく以下の 3 つに分類される。

① コンピュータグラフィックス (CG)

CG は、計算機に収められているデジタル情報を人間が視覚的に理解できるようにディスプレイ等の出力装置上に表現する技術である。

1950 年代の頃は非常に簡単な図形描画ができる程度であったが、その後計算機能力の飛躍的な進展に伴い、写真との区別がつかないほど自然界を描画できる程度にまで急速な進歩を遂げている。表現技術としては高画質を実現するレイトレーシング法、ラジオシティ法や、表面だけでなく内部もモデリングされた物体を表現するボリュームレンダリングなどがあり、これらの基本原理は 1980 年代に提案・開発された。その後 1990 年代に入ってからは、これらの手法の拡張・改良・高速化が加えられ、現在に至っている。

CG の実用化における成功例として、映画への応用を抜きにして語ることはできないであろう。1993 年に公開された映画「ジュラシックパーク」でのリアリティ追求に始まり、以後の映画制作における CG の活用は、映画業界のみならず一般市民の間でも、もはや当たり前のように考えられてきている。これを発展させ、CG と実写を融合させた映画作りのための技術も年々進歩しており、CG は既に、我々の日常生活にすっかり融け込んできている技術であるといえよう。その他にも、高層ビルなどが林立する都市空間に携帯電話・PHS 等の移動体通信の基地局を適切に配置するための、電波伝播を可視化する手法についての研究や [4.2.3-3]、日本近海の海底地形を描画する技術の開発 [4.2.3-3] などが進められており、CG 技術は広い分野で応用されている。

② サイエンティフィック・ビジュアリゼーション (SV)

SV は、科学技術分野における様々な物理現象を理解するために行う数値シミュレーションや物理量の計測によって得られる出力結果を、数値の羅列ではなく画像によって表現し、人間がその現象を容易に理解できるようにするための技術である。

近年、計算機能力の益々の増大、新規計算手法の開発により、計算機による物理現象のシミュレーションをより精密に行うことが可能になってきている。それに伴い、実物相当のものを用いた試験の遂行頻度を抑えて開発期間・コストの縮小化を図る目的で、例えば航空機や車両の設計においては、従来から行われている風洞実験に代わり数値風洞を積極的に活用することで流力特性を評価したり、車の設計においては、実物を用いた衝撃試験の多くを数値実験で代用することで衝撃特性を把握するなどの動きが見られてきている。こ

うした数値シミュレーションにより得られる結果は、一般に大量の数値データを出力することになるが、それらを数値データのまま表現していたのでは、大規模な工学物を対象とした設計・安全評価において対象物のメカニズムを理解することは事実上不可能に近いと考えられる。そのため、結果の可視化は必須事項の一つといえる。

こうした可視化へのニーズは解析分野に限ったものではなく、実験・計測の分野においても高い。医療分野における可視化技術の応用はその一例であり、核磁気共鳴（MRI）を用いた脳の断層写真撮影は脳の異常の早期検知や、手術を行う箇所の正確な同定に大きな役割を果たしている。また最近では、互いに遠隔された医療機関同士をネットワークで結び、一方の機関Aで手術を行いながら患者の体内の様子を透視撮影してもう一方の機関Bにリアルタイムで動画配信し、Bがその画像を見ながら逐一Aに手順を指示し手術を進める、といったことが実用化されつつあり、この画像処理技術により患者は医療機関を渡り歩く必要がなくなり、患者の移動への負担を大幅に軽減することが期待されている。また、サイクル機構では高速炉の実用化を目指して様々な要素試験を行っているが、例えば原子炉の炉心を構成する燃料集合体の照射後の状態を把握するために、X線CT技術を用いた燃料集合体の断層画像を作成することが行われており[4.2.3-4]、これにより燃料ピンやラッパ管等、燃料集合体を構成する部材の健全性等を適切かつ容易に把握することが可能となっている。

以上のように、科学技術分野において可視化技術の適用はもはや必須のこととなりつつあり、その対象規模が複雑になればなるほど、また設計・製作に求められる精緻さが増せば増すほど、可視化へのニーズは今後ますます高まるといえる。

③ バーチャルリアリティ（VR）

VRは、コンピュータ内に構築した仮想的な空間を、各種のインタフェース機器を用いて視覚や聴覚、触覚などの人間が知覚できる情報として再現する技術である。この技術は、実在はするもののそれを直接用いることが困難な場合に、その実在物の中の様子や実際の挙動を人間が仮想的に体感できるようにすることなどに用いられる。航空機のフライトシミュレーションや原子炉の運転シミュレーションはその例で、これらは運転員の教育訓練等に実際に用いられている。施設内をコンピュータ上で仮想的に渡り歩き、中の様子を擬似的に体感するウォークスルーも応用例の一つで、サイクル機構では大洗にある情報センターのウォークスルーを一般に公開している[4.2.3-5]。

以上述べたように、可視化技術は技術そのものもさることながら、その応用範囲も年々着実に広がりを見せていることが伺える。解析コードに供給する入力データの作成や昨今のOS環境等にも、従来の文字をベースとしたインターフェースに代わり、グラフィカルユーザーインターフェースの利用が定着しつつある。さらに計算機能力の飛躍的向上により、音楽・音声、画像・映像等の文字以外の情報をデジタルに扱うマルチメディアにも可視化技術は登場してきている。したがって、科学技術分野を含め、今後、我々の生活の至るところに可視化技術がますます入り込んでくることは容易に想像できる。

4.2.3.2 可視化技術と情報技術の相互進展

上記(1)で述べた可視化技術の発展・拡大は、新規アルゴリズム・概念の開発もさることながら、可視化技術を取り巻く周辺技術の発展に依存するところも大きい。可視化技術の発展が周辺技術の発展を促し、逆に周辺技術のさらなる発展がより高精度・高機能の可視化技術を生み出す。このサイクルは年々速度を増し、数年前までは高価なワークステーションを必要としていた描画計算も、今やパーソナルコンピュータで十分可能となっている。

このような可視化技術を取り巻く技術の中で、特に計算機ハードウェア技術およびネットワーク技術は可視化技術の発展に大きく寄与している。

① 計算機技術の進歩

これまで画像処理といえば、高価な専用のグラフィックワークステーションを用いるという考え方が一般的であった。ところが、計算機ハードウェア技術の飛躍的な向上によるパーソナルコンピュータの一般ユーザへの著しい普及に伴い、様々なユーザが比較的安価にパーソナルコンピュータを入手することが可能となり、今や描画したいものを各家庭のパーソナルコンピュータで高画質に画像処理することが可能になった。そのために必要なグラフィック・エンジンやグラフィック・ボードも年々充実度を増しており、複雑な 3 次元画像をパーソナルコンピュータで描写することも現在では決して珍しいことではなくなってきている。

② ネットワーク技術の進歩

近年のインターネットを中心としたネットワーク技術の発達も急速な可視化技術の進展に有効に寄与している。従来、電子メールなどに代表されるように、計算機間の通信は文字ベースのものが中心であったが、WWW 技術の普及により画像を通信する頻度が急激に増加し始めた。ビジネスや研究の場では、100Mbps から最近では 1Gbps 規模の高速通信が普及し、これにより解析結果の可視化画像を互いに通信したり、それを論文に貼り付けて国際会議等に提出するなどの作業が短時間で行えるようになってきた。さらに、一般家庭においても、数 Mbps 程度の通信速度を提供する ADSL 等のサービスが利用できるようになったことで、従来に比べはるかに高速の画像通信ができるようになっている。パーソナルコンピュータを使った家族や友人同士での記念写真の送受信や、最近急速に普及しつつある携帯電話を用いた本人の顔や周囲の状況の動画配信などはその良い例である。こうした流れは画像圧縮の技術や、安価なグラフィックス・ボードの急速な技術開発をも刺激している。

4.2.3.3 次世代解析システムの開発との関係

次世代解析システムの開発においては、その主たる対象が原子力という科学技術分野であることから、(1) で述べた 3 種類の可視化分野のうち、特にサイエンティフィック・ビジ

ュアリゼーションが大きく関係するといえる。

2章の2.2でFANTASIシステムについて述べたが、このシステムによる数値シミュレーションで得られる結果を可視化するための画像処理システムもサイクル機構で開発しており、これらは既に同機構大洗工学センターのFセルボ1階シミュレータ室に実装され、現在社内外に広く活用されている。この画像処理システムの開発にあたって着目した点は、FANTASIシステムが対象とする物理現象が核、熱流動、構造という複数にわたっているため、各々の専門家がシミュレーション結果の分析のために共有すべき情報を数値や文字ではなく、画像で表現すれば互いの理解の促進に効果的であるとの考えであった。例えば、炉心冷却材の温度変化に基づく炉心構造物の熱変形とそれに伴う反応度変化の関係を構造専門家と炉物理専門家の間で議論する場合、構造分野における専門用語と炉物理分野における専門用語は互いに専門の異なる研究者にとっては必ずしもその意味の理解が容易でない場合が多い。このような状況下では、構造物の変形という現象と反応度変化という現象を画像にして表現することにより、難解な用語で現象を理解しようとする場合に比べ、現象の理解は格段に容易となる。

次世代解析システムの開発では、オブジェクト指向技術を活用して、解析システムというものをオブジェクトという一種の部品の組み合わせで構築する手法の確立を目指している。これら部品の種類や数が増えるにつれ、それらの組み合わせのパターンもますます増大し、それによって異なる専門分野同士の情報交換もさらに活発化することが予想される。また、画像処理のみならず、昨今の計算機技術の目覚しい発達とそれに伴う数値演算の高速化を考えれば、大規模モックアップ試験を数値解析で代替することにより、パラメータ変更の柔軟性やコスト削減効果に富む数値実験を軸とした新しい研究開発アプローチへの期待もますます高まっていくものと考えられる。こうした近未来の状況を見据えれば、可視化技術の利用はもはや研究開発の分野においても必須になるものと思われる。

その際、どのような可視化環境を構築するかが重要であるが、複数専門分野間でのデータ通信における互換性にまで配慮することが必要との認識に立てば、市販品として多くの研究機関で広く用いられているAVSのような可視化ソフトを共通的に用いるのが望ましいといえよう。

また、サイクル機構のみならず科学技術に携わる研究機関においては、実験と解析の有機的な連携は研究開発を効率的に進める上で重要である。(1)では、実験、解析ともに可視化技術適用のニーズが高いことを述べたが、可視化を行うに際して、両者をできるだけ同じ可視化環境の下で比較することは研究開発の効率化を図る上で重要な視点であると考えられる。次世代解析システムの開発においては、こうした実験、解析の両面から可視化環境の構築方法について検討することも今後の重要なテーマの一つであるといえる。

4.2.4 インターフェース技術の例

4.2.4.1 ADVENTURE システム

日本学術振興会未来開拓学術研究推進事業「計算科学」分野の ADVENTURE プロジェクトとは、次世代の汎用計算力学システムの基盤とすべく、Beowulf 型の廉価な PC クラスタから超並列計算機までをカバーする汎用的な並列環境で稼動する汎用並列有限要素法解析システム（ADVENTURE システム/フリーウェア）の開発を行うものである。

このシステムの特徴は以下の通りである。

(1) 数百万～1 億自由度級メッシュを用いて詳細な丸ごと解析が行える。
(2) プロセッサ数が数個から千オーダーの幅広い並列～超並列計算機環境においてスケーラブルに高い並列性能が得られる。
(3) PC クラスターから超並列計算機まで多様な並列分散環境へ容易に移植可能である。
(4) 並列分散環境において、単一現象の解析はもちろん、連成現象解析や設計解析を柔軟かつ効率的に行える。

(1) Adventure I/O

この Adventure システムは、分散 / 並列環境で動作し大容量のデータを扱う、データ入出力形式に求められるさまざまな要求を満たすべく設計されたのが Adventure I/O フォーマットそして Adventure ライブラリである。

Adventure システムのモジュールとモジュールを結合するためのフォーマット及びライブラリ

- Adventure フォーマット
- Adventure ライブラリ

Adventure I/O に要求される事

- Adventure システムの各モジュールで共通に使える事。
  フォーマットが定義されている事。（汎用性があること。）
- 大容量データの取り扱いが可能であること。
- 分散/並列環境で使えるものである事。
- I/O 効率の高いものである事。

Adventure I/O の応用

- 複数の解析モジュールの連携による連成解析への応用

・ 他の並列解析システムからの、AdvMetis モジュールの効果的な利用

① Adventure I/O フォーマット

以下に Adventure I/O フォーマットの概要を記す。

データ構造

- File
  - Document
    - Document ID
    - Property
    - Raw Data
  - Document
    - Document ID
    - Property
    - Raw Data

File の構造

- File は 1 つ以上の Document で構成される。
- File には通常の Disk 上の File の他にメモリ上の File、遠隔計算機上の File などが指定できる。(CORBA を用いた応用)
- ファイルサイズの上限が 2GB に制限されている OS においてファイルサイズが 2GB を超えた場合は自動的に複数ファイルに分割される。(ライブラリの機能)

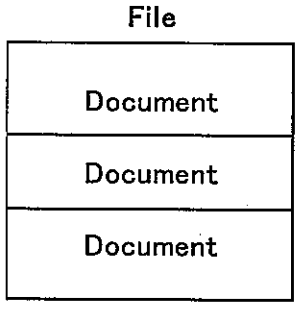


図 4.2.4.1-1 File の構造

Document の構造

- Document ID 部分及び Property 部分と Raw Data 部分で構成される。
- Property 部分は RawData 部分のデータの意味を表す情報や領域分割ツールに渡すオプションなどで構成される。
- RawData 部分には各接点の座標値や物理量などのマスデータが記されている。

Document ID の構造

- Document ID は Document 名、作成日時、乱数等を組み合わせた固有の文字列で構成される。
- 全ての Document ID は必ず固有な文字列となり、Document ID のみで任意の Document を指定する事が出来る。

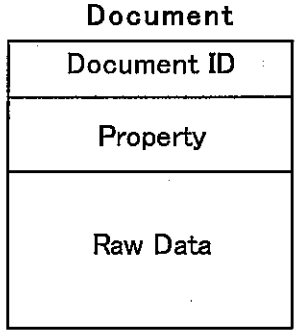


図4.2.4.1-2 Document の構造

Property の構造

- Property は key と value の組み合わせによって構成される。
- Key の名前小文字で区切りには_を使う。
- Value の名前には区切りに大文字を使い、言葉は略さず、複数形の s は付けない。

Raw Data の構造

- Raw Data はバイナリ形式のデータの集合で構成されている。
- データの並び方は property 部の format の値で決まり、データのサイズは num_items の値で決まる。

サンプル（advinfo の出力）

② Adventure I/O ライブラリ

以下に Adventure I/O ライブラリの概要を記す。

使用方法

ファイルと Document を開く

```
#include "AdvDocument.h"
int main(int argc、 char** argv)
{
  AdvDocFile* dfile;
```

```
AdvDocument* doc;
dfile = adv_dio_file_open(testfile、 "C");
doc = adv_dio_create(dfile、 adv_dio_make_document("document_name"));
```

開いた Document に property を書き込む。

```
Adv_dio_set_peoperty        (doc、 "prop1"、 "string");
Adv_dio_set_peoperty_int32    (doc、 "prop2"、 "12345678");
Adv_dio_set_peoperty_float64 (doc、 "prop3"、 "1.2345678");
```

開いた Document に RawData を書き込む

```
items = 100;
adv_dio_property_int32 (doc、 "items"、 items);
off = 0;
for( I = 0、 I < items、 I++ )
  off += adv_dio_write_int32 (doc、 off、 i);
```

Adventure I/O フォーマットとライブラリの特徴

- Adventure I/O フォーマット及びライブラリ（AdvIO）は、Adventure システムの各モジュールを結合する目的で開発されたモジュールである。
- 大規模データの分散並列環境における入出力が効率よく行えるような設計がなされている。
- FEGA によりデータの抽象化がなされ、汎用性に優れた仕様になっている。
- リリースされた Adventure I/O の詳細を意識する必要はない。
- 複数の解析モジュールの連携による（弱）連成解析を行える仕様になっている。

4.2.4.2 GeoFEM システム

(1) 概要

GEoFEM プロジェクトは、「科学技術振興調整費：高精度の固体地球変動予測のための並列ソフトウェア開発に関する研究」の一部として（財）高度情報科学技術研究機構で開発されている、固体地球分野を対象とした並列有限要素解析システムである。

GeoFEM は、大規模かつ複雑なマルチフィジクス／マルチスケールに及ぶ固体地球の諸モデルをプラグイン形式で取り込むことの出来る、並列プラットフォームの構築を目指している。これは、固体地球分野の研究者の持っている既存のコードを GeoFEM にプラグインすることによって、並列化による解析性能の向上と共通基盤によるデータの共有化を図ろうという意図に基づいている。

(2) GeoFEM プラグインの概念

GeoFEM のプラグイン機能とは、既存の逐次式処理型の有限要素法コードを GeoFEM に組み込む機能であり、プラグインすることによってコードが並列化され、入出力データが GeoFEM と共有化できる。領域分割法に基づき並列反復ソルバを使った GeoFEM のプログラム構造の概要を図 4.2.4.2-1 に示す。GeoFEM は SPMD (Single Program Multiple Data) 型のプログラムであり、各 PE が領域分割された部分データを読み込む。その後、要素毎の処理を行い PE 間の通信はソルバのみで発生する。よって既存の有限要素法コードを GeoFEM にプラグインするには図 4.2.4.2-2 に示すようなソルバ及びデータの入出力部分を GeoFEM 側で行い、要素毎の処理である全体マトリックスの作成と状態変数の更新については、既存の有限要素法コードをそのまま利用する。

また、並列反復ソルバでは PE 間通信のためのデータを必要とするが、これらは GeoFEM の入力データに含まれており、GeoFEM のユーティリティプログラムとして用意されている領域分割プログラムにより作成する事が出来る。よって利用者が通信データを直接意識する必要はない。尚、GeoFEM は MPI と FORTRAN90 によりプログラムされており、特に FORTRAN90 のモジュール文、構造体、ポインタ等の機能を利用しているため、組み込むコードが FORTRAN90 で書かれている必要はないが、GeoFEM として動作させるためには、FORTRAN90 のコンパイラが必要となる。

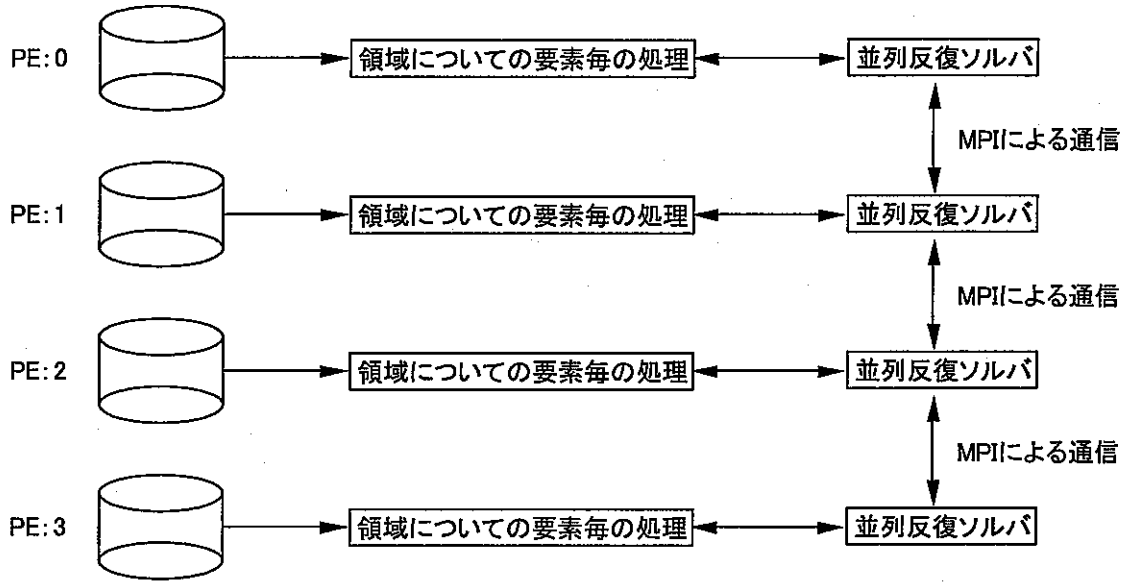


図 4.2.4.2-1 GeoFEM による SPMD 型並列有限要素法の概念

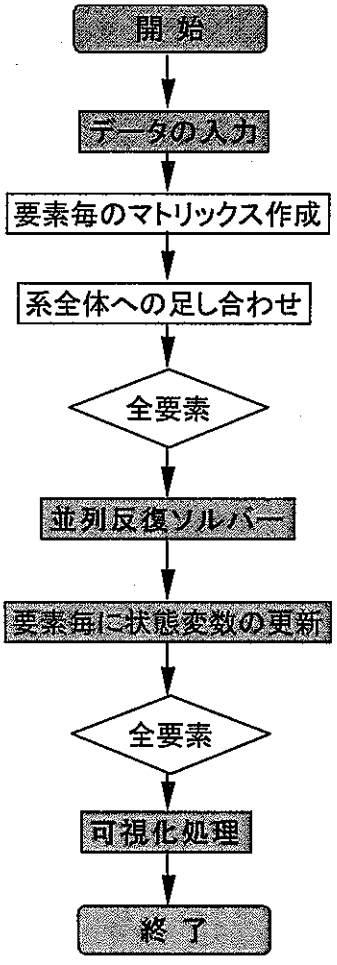


図 4.2.4.2-2 既存の FEM コードのプラグイン

(3) GeoFEM プラグインの手順

GeoFEM のプラグインは、以下の３つの手順により実現されている。これらは、仕様が決められているプログラムのインターフェースつまりプラグであり、プラグインとはユーザのプログラムをこのプラグに接続することに他ならない。

①プログラム構造の定義

GeoFEM では、プラグインの単位であるサブプログラムをモジュールと呼び、解析用のモジュールを解析器、可視化用のモジュールを可視化器と呼んでいる。ユーザの既存の有限要素法コードは解析器としてプラグインするが、このとき解析器を定義するためにモジュール名として analyzer、初期化サブルーチン名として init_analyzer、解析ルーチンの入口として analyzer とすることが決められている。

②データ入出力のサポート

有限要素法モデルを定義するためのデータとして、メッシュデータ（節点座標、要素接続等）の書式が決められており、その読込みルーチンが提供される。また、各々の解析器に特有なデータは GDL（GeoFEM Data description Language）を使って定義すれば、読込みルーチンが作成され、解析器側に渡される仕組みを持っている。更に可視化のための出力は、節点、要素について指定された構造体を作り、特定のサブルーチンを呼べば、可視化器へ接続される仕組みが提供される。これらは、GeoFEM のユーティリティプログラムである geoutil によりサポートされている。

③並列反復ソルバのサポート

GeoFEM は有限要素法の並列初期をソルバのみに局所化し、並列化のための通信テーブルはメッシュデータの一部として、その構造が定義されている。従ってユーザのプログラムはソルバを GeoFEM 提供のものに変更し、通信テーブル（GeoFEM が用意している領域分割プログラムによって自動的に作成される）を用意すれば、並列計算が可能になる。

このように、GeoFEM のプラグインでは入力、処理、出力の各々についてプラグが用意されており、つまり、有限要素法を並列化し、かつ入出力を共有できるシステムであり、まさしく並列有限要素のプラットフォームということが出来る。また、これらのプラグを使う事によってユーザが並列処理を意識することなく、有限要素法コードの並列化が達成できる仕組みを実現している。

4.2.4.3 電脳 davis プロジェクト

地球流体分野では、科学技術振興事業団の計算科学技術活用型特定研究開発推進事業として、平成 10 年度に「地球惑星流体現象を念頭においた多次元数値計算データの構造化」というテーマでプロジェクト研究が行われた。

地球や惑星での流体現象（大気や海洋、マントルや中心核）における研究では、得られた多次元データを認識可能な形に整形するのに多大な労力がかかることが問題となっており、これらの問題を解決するために、データのオブジェクト指向的な構造化の導入により、数値計算や観測データ解析で関連するも代を整理合理化することを目的として、このプロジェクト研究が実施された。

このプロジェクト研究の報告書では、地球惑星流体現象にまつわる問題点は、

- 大規模、すなわち、サイズが大きい、あるいは、数が多い
- 多様、すなわち、個人、グループ、プロジェクトなどの生産元ごとにデータの形式構造が異なっている

の二点にまとめることができるとしている。この問題点は、現在、我々が課題としている原子力関連の解析コードの連携の問題と非常によく似ている。

これらの課題に対して、電脳 Davis プロジェクト研究では、標準データ構造と対応した処理手法を提案することにより、データの加工交換を容易にし、問題考察能力を高めたいという要求があった。この要求を達成するために、必要となるデータ構造の性質は、

- ネットワーク透過性の確保、すなわち、機種/OS 非依存のバイナリデータであること
- 自己記述的なデータ構造とデータ処理、すなわち、データ自身がその中身を語ってくれる自己記述性とデータ自身が同所利して欲しいか語ってくれるオブジェクト指向であること

の二点であるとしている。

このプロジェクト研究では、上記の目的を達成するために、多次元データに対して、以下の三層に分けたアプローチを採用した。

- データの構造
  - データの処理と表示を念頭においた属性リストの整備
  - NetCDF を機種/OS 依存しないバイナリ形式として採用
- データの処理
  - 地球流体での数値処理に適した Fortran90 によるライブラリ（Fortran によるオブジェクト指向的実装）
  - オブジェクト指向スクリプト言語 Ruby によるライブラリ

- データの可視化
  - 研究教育現場で自由に流通させることが可能（学生への配付が可能）な最低限の可視化資源の確保

この研究では、NetCDF と呼ばれるデータ形式を採用し、多次元データの格納形式を提案している。このデータ形式の特徴としては、

- 自己記述的
- 非格子状データのサポート
- 数値データと可視化情報の統合
- 機種・OS 非依存性

といったことが挙げられている。

このプロジェクト研究では、地球流体科学における大規模な多次元データの可視化処理が主目的である。原子力関連の解析コードの連携においても、大規模な多次元データが扱われており、これらのデータに対する連携処理が、現在、我々が解決すべき課題であるので、この研究によって得られた知見を活用できる可能性は高いと考えられる。

この研究では、データ処理に関しては、従来から整備されていた Fortran77 ベースのツールと親和性の良い Fortran90 や C 言語による実装と、オブジェクト指向スクリプト言語である Ruby による実装の、二種類の実装が試された。Fortran90 による実装がメインで、Ruby による実装は、試験的なものであったようであるが、この試験的実装により、Ruby により、柔軟で軽快なデータ解析環境が提供される可能性が示せ、かつ、netCDF のようなデータ形式も容易に扱うことができるとしている。

ただ、現状の問題点として、Ruby のコミュニティーにおいて、数値処理の経験が少ないこと、スーパーコンピュータ上への効率の良い実装、巨大データに対する実装等を挙げている。

なお、この研究では、当初、Ruby ではなく、欧米で既に流通している Python を利用する計画であったらしいが、Ruby の方がより純粋にオブジェクト指向であり、かつ、言語制作者が国内にいるので、より緊密な連係がとれるであろうとの判断から、Ruby を採用することにしたとのことである。開発者が日本人であるので、Ruby の開発グループと日本語で直接的なコミュニケーションがとれるということは大きなメリットであると強調されている。

4.2.4.4 計算機支援問題解決環境CAPSE

(1) CAPSE (Computer Aided Problem Solving Environment)とは

CAPSEとは、分散コンピュータ環境上でシミュレーションを効率よく行うための支援システムである。日本では、いくつかPSE(Problem Solving Environment：問題解決環境)システムが開発されているが、このCAPSEは、(財)情報処理振興事業協会(IPA)の協力により、開発が始められたシステムである。

このCAPSEについて、以下のとおり解説されている。

(2) 背景

数値シミュレーションは、理論と実験を補足する科学活動の一環であることが一般に認められるようになった。CAPSEでは、計算科学者がシミュレーションを利用して、通常行う問題解決作業について、

1) 考察している現象の数学モデルを構築する。
2) 適切な物理的・幾何学的手法を選択する。
3) 方程式と付随する条件を操作して、適切な解法を適用できるように問題を簡略化する。
4) 解析的・近似的手法による解法を指定する。
5) 適切な仕様記述言語及びプログラミング言語を用い、解法実行のためのプログラムを作成する(新規作成及び既存資源の拡張)。
6) 試行問題及び試行データを作成する。
7) プログラムに試行データを適用する。
8) 結果を検証する。
9) 結果と性能を別の解法によるものと比較する。
10) 出力データの収集及び加工を行う。
11) 実験手順を記録する。
12) 結果を学会等に報告する。

という作業項目が考えられており、問題解決するための作業として、既に適用されている。問題解決作業は、上述のプロセスの一部あるいは全部が使われて構成される(図4.2.4.4-1)。

また、「研究者や技術者が自分たちの言葉／用語で問題やその問題の着眼点をPSEに伝達すると、問題が解かれ、彼らの言葉／用語で解答が提示される」という理想形が考えられており、この理想形の実現が期待されている。

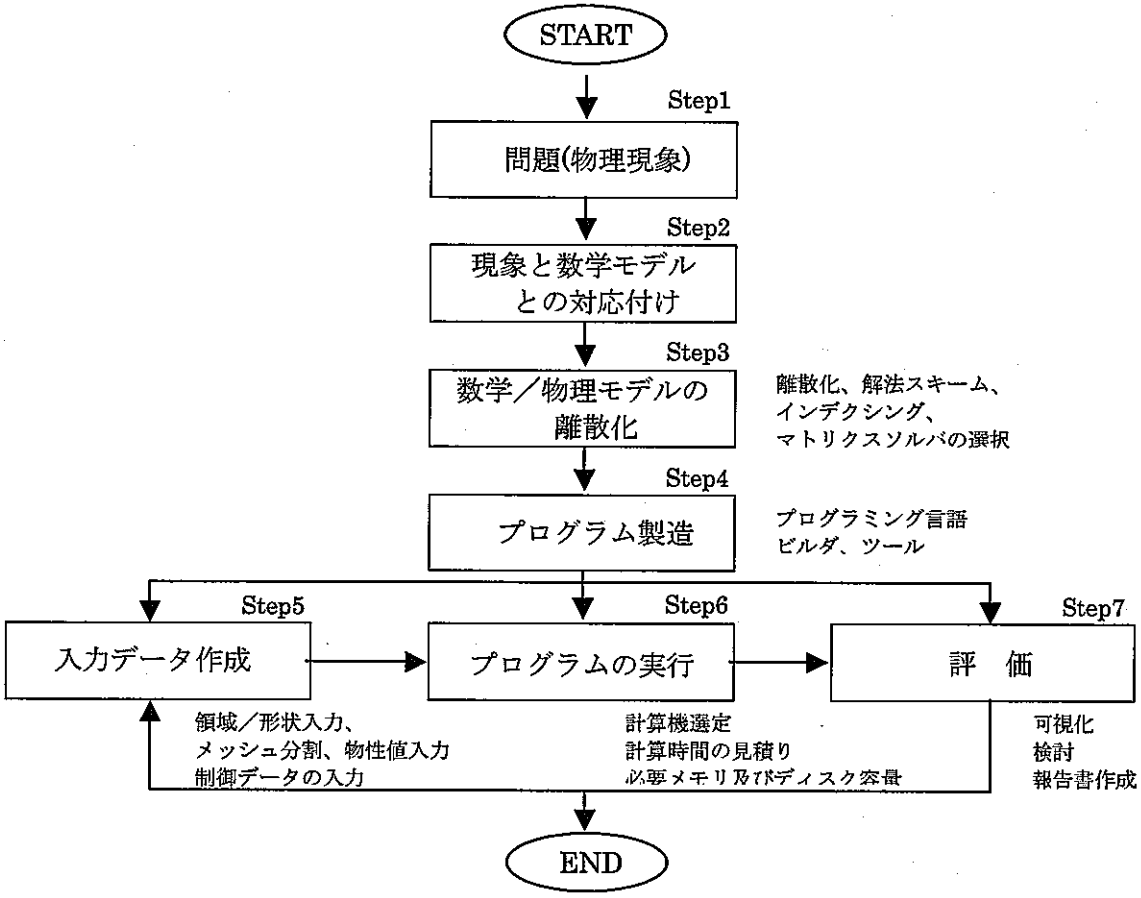


図 4. 2. 4. 4-1　問題解決作業のプロセス

(3)　目的

CAPSE の開発は、将来に渡る各種の科学・工学問題に適応可能な汎用的な PSE を実現することを目的とされている。すなわち、本開発は、研究者・技術者に対し、容易にかつ効果的に個々人の科学・工学問題を解くためのワークベンチを提供することを目指すものであり、米国に比して大きく遅れている日本の PSE 研究開発を前進させるものである。これにより、今後ますます重要とされる日本の計算機シミュレーション分野の発展に寄与すると同時に、高度な技術を駆使する産業の発展と国際競争力強化に貢献することを目指すものとされている。

(4)　開発概要

科学技術計算分野の問題解決を支援するプラットフォームのシステム要件を、図 4.2.4.4-1 の問題解決プロセスを基礎として、以下に示すように定義した(図 4.2.4.4-2)。

1) 問題定義
2) ツールジェネレータ
3) インテグレータ
4) ライブラリ(最適化エンジンを含む)
5) 知識ベース
6) ユーザインタフェース

「問題定義」は、対象問題を物理現象とともにシステムに知らせることである。問題を計算対象領域、支配方程式及び境界条件として定義する方法や実現象を実験装置やイメージ図等で表現し認識させる方法等が考えられる。

「ツールジェネレータ」は、対象問題を解くツールが既存ツールとして存在しない場合に利用する。物理現象を表す保存則を偏微分方程式系で入力し、自動的にプログラムソースを生成する方法と既存のモジュールを再利用するためのワークベンチを提供し、新規モジュールと組み合わせて新たな機能を有するツールを半自動的に生成する方法等が考えられる。

「インテグレータ」は、複数の既存ソフトウェアを組み合わせて、科学技術計算分野のある1つの問題解決システムをコンピュータ上に実現するためのワークベンチである。既存の「領域定義及びメッシュ生成プログラム」、「求解プログラム」、「可視化プログラム」の他、外部仕様の明確な市販パッケージや「ツールジェネレータ」で生成したプログラムを組み合わせて利用するための仕組みである。

「ライブラリ」は、独自開発、市販を含む既存ツールであり、プログラムの貯蔵庫である。「ツールジェネレータ」で生成されたプログラムもコンポーネントとしてこの貯蔵庫に格納される。また、本「ライブラリ」には、最適化エンジンを搭載し、パラメータスタディや感度解析、設計変数及び特性値の最適化、最適解の探索を支援することにより、技術者・設計者の材料・製品開発を大幅に効率化することができる。

「知識ベース」は、対象問題が与えられたときに、「その問題を解釈し」、「その問題を解くための最適な計算手法または既存ツールを推測／助言し」、「既存ツールに対しては、最適な入力データを予測／助言し」、「計算結果を評価する」等の機能であり、その役割は多岐に渡る。ただし、AI／エキスパートシステムを含む知識ベースは、数値シミュレーション分野への適用という意味では、最も立ち後れた技術であり、今後の技術開発と運用ノウハウの蓄積が待たれる研究分野である。

「ユーザインタフェース」は、インテグレータやツールジェネレータ、知識ベース等を統一的に利用するためのGUIである。また、問題定義時にシステムに問題を認識されるための音声認識や画像読み取りと言ったコンピュータとの新しいI/Fの適用も考えられる。

上述のように定義したプラットフォームは、「ライブラリ」をさらに充実することにより、多くの問題分野に適用可能なものとなる。また、「インテグレータ」、「知識ベース」は、適用可能なライブラリが増えれば増えるほどその効力を発揮することになる。本プラットフォームは、より汎用的ないわゆる「計算科学のための問題解決環境(PSE)」への発展の可能

性を持つものである。

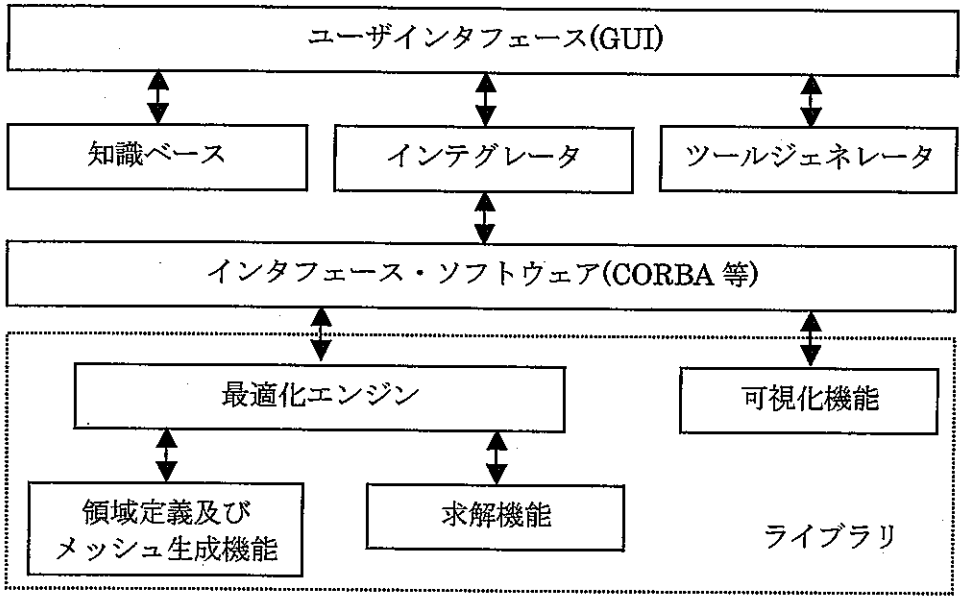


図 4.2.4.4-2　システム構成

本稿では、上述のシステム構成を踏まえ、システム要件と対応するように、下表に示す機能を開発した。以下では、表の開発機能に示す 5 つのサブプログラムの概要について述べる。

表 4.2.4.4-1　PSE のシステム要件と開発機能

システム要件	開発機能
(1) 問題定義	統合化環境機能 プログラム自動生成機能
(2) ツールジェネレータ	プログラム自動生成機能
(3) インテグレータ	統合化環境機能
(4) ライブラリ	最適化問題対応機能 求解及び求解支援機能
(5) 知識ベース	利用支援機能
(6) ユーザインタフェース	統合化環境機能

(5)　CAPSE の機能

CAPSE の機能として、前述のとおり、システム要件に対応した機能が開発されている。その機能について、以下のとおり、解説されている。

① 統合化環境機能

統合化環境機能は、ネットワーク上に散在する科学・工学分野のアプリケーションプログラムを単一の操作環境に統合化し、それらのプログラムを利用して行う一連のシミュレーション作業の流れを一元的に管理あるいはその作業プロセスを制御する機能である。この機能は、「分散コンポーネント管理機能」及び「分散オブジェ

クト管理機能」により達成される。「分散コンポーネント管理機能」及び「分散オブジェクト管理機能は、それぞれネットワーク上に散在するアプリケーションプログラム及びデータファイルを分散コンポーネント及び分散オブジェクトとして抽象化し、統合化環境機能上に取り込む。両機能により、利用者は、ネットワーク上のアプリケーションプログラム及びデータファイルの所在地を意識することなく、統合化環境機能の統一された GUI を利用して問題解決システムを構築し、利用することができる。

また、統合化環境機能を含む以下に示す各種サブシステム(CAPSE ソフトウェア群)は、Web サーバを介してインターネットに接続されており、インターネットに接続された全ての Web ブラウザからログオンしてその機能を利用することができる。すなわち、利用者は、プリプロセッサ、ソルバ、ポストプロセッサ等からなる科学技術計算のためのアプリケーションプログラムを、LAN 及びインターネット上から統一的に利用し、自らの問題を解くことができる(図 4.2.4.4-3 参照)。

さらに本機能は、計算プロセスの自動実行、繰り返し実行機能等からなる「プロセス実行制御機能」、サブルーチンレベルでの統合化及びビジュアルプログラミングを支援する「メタモジュール統合化機能」、プログラムへの入力データを画面から入力するための支援機能やプログラムの計算結果を可視化するための支援機能、プログラム間のファイル情報を変換するためのファイル変換機能からなる「分散コンポーネント開発支援機能」を有し、利用者のコンピュータ・シミュレーションによる問題解決を支援する。

本機能は、図 4.2.4.4-1 に示す科学・工学問題の問題解決プロセスのすべての Step (プログラムの作成及び運用両フェイズ) を支援するワークベンチである。

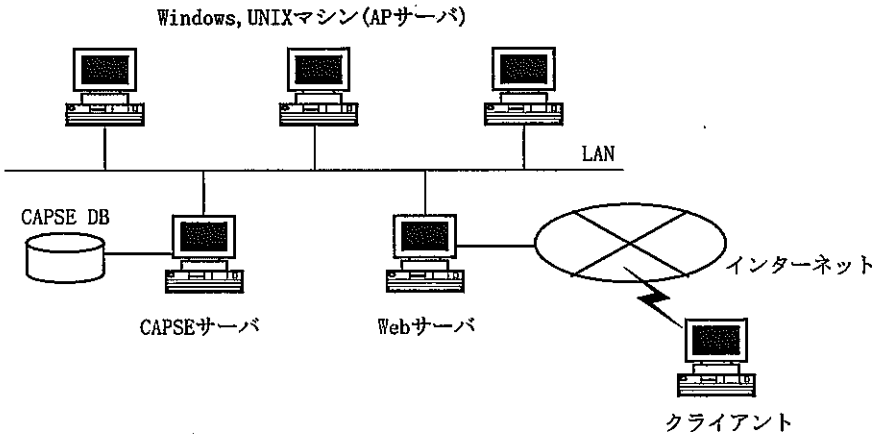


図 4.2.4.4-3　統合化環境機能の動作環境

② プログラム自動生成機能

本機能は、偏微分方程式や計算領域、境界条件などを入力し、数値計算プログラムを自動生成する機能である。本機能は、図 4.2.4.4-4 に示すように、既存の偏微

分方程式問題のプログラム生成支援環境である NCAS(長岡技術科学大学)をベースに開発した。本開発では、この NCAS の操作性向上を目的に、計算モデル、支配方程式、計算領域及びメッシュ情報からなる問題定義情報を GUI を用いて定義し、容易にこれら問題定義情報を NCAS に供給するための以下の機能を開発した。

- モデル化支援機能
  次元数、時刻情報(Δt やサイクル数)、未知数等からなる計算モデルを定義する機能。
- 数式入力支援機能
  支配方程式を構成する未知数変数名、偏微分記号、算術記号等を数式パレットから選択して入力する機能。
- ライブラリ連携機能
  既存のプリ・ポストプロセッサと連携し、複雑な計算領域及びメッシュを NCAS に供給し、可視化するための支援機能。

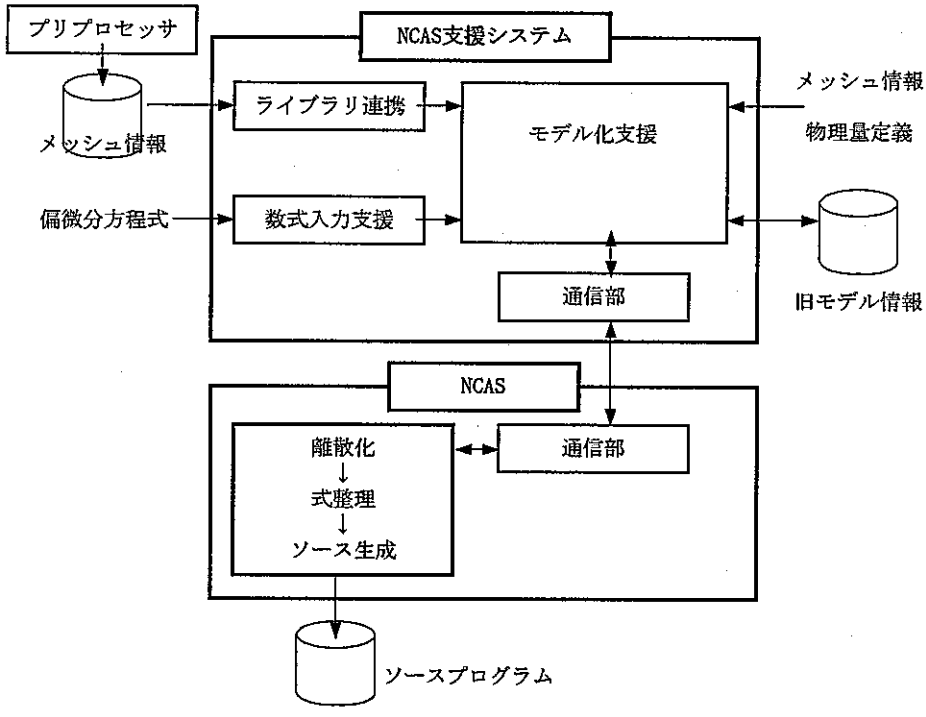


図 4.2.4.4-4　プログラム自動生成機能

本機能で自動生成される個々のプログラムは、統合化環境機能上に分散コンポーネントの 1 つとして統合化され、数値計算による問題解決のための利用に供せられる。

本機能は、図 4.2.4.4-1 に示す科学・工学問題の問題解決プロセスのうち、Step1～4 のプログラム作成フェイズを支援するワークベンチである。本機能は、プログラミングすることなしに、研究者・技術者に、彼ら個々の持つ問題を解決するための

数値計算プログラムを提供する。したがって、研究者・技術者は、情報技術の習得及び運用するための時間を削減することができ、技術開発や製品設計等の問題解決のために優先的に自らの時間を割くことができる。

③ 最適化問題対応機能

本機能は、図 4.2.4.4-5 に示すプログラム群により設計最適化問題を解決する。

通常、構造解析、伝熱解析、熱流動解析等の数値計算プログラムは、応力、変位、温度、流速といった物理量(従属変数)についてのみ結果を算出する。これに対し実際の設計プロセスでは、設計目標を満たすような設計変更を繰り返しながら、複数の物理量や形状等をコストを含め最適な値またはその値がある許容範囲内に入るよう模索する手順が用いられる。本機能では、設計者が複数のケーススタディを行うことなしに、最適な目標設計値を取得できるよう上記プロセスを自動化する。これにより、数値計算による問題解決の生産性を飛躍的に向上することができる。

本開発では、以下に示す 5 つの機能を開発し、最適化問題に対応した。

- 最適化プロセス制御機能
  最適化問題を定義するファイルを読み込み、以下のパラメータスタディ自動化機能、感度解析用簡易解析モデル生成機能、NLP や GA による最適解探索機能を制御する。さらに、形状最適化のための適応メッシュ生成機能を起動する。
- 最適解探索機能
  数理的探索方法として、代表的な非線形計画法ソルバであるSQP、AGL、QNMを有する。また確率的探索法としてGAを用いる。
- 感度解析用簡易解析モデル生成機能
  L9、L27 直交表を用いた応答曲面近似式を生成する。作成された応答曲面を用いて、応答値及びその微分値が計算される。
- 適用メッシュ生成機能
  既存の CAD データから 2 次元及び 3 次元形状定義データを生成する機能、及びこの形状定義データから有限要素メッシュデータを生成する機能。
- パラメータスタディ自動化機能
  既存の求解プログラムに設計変数と入力データを与え、逐次求解プログラムを起動し、計算結果及びその中の特性値を抽出する。

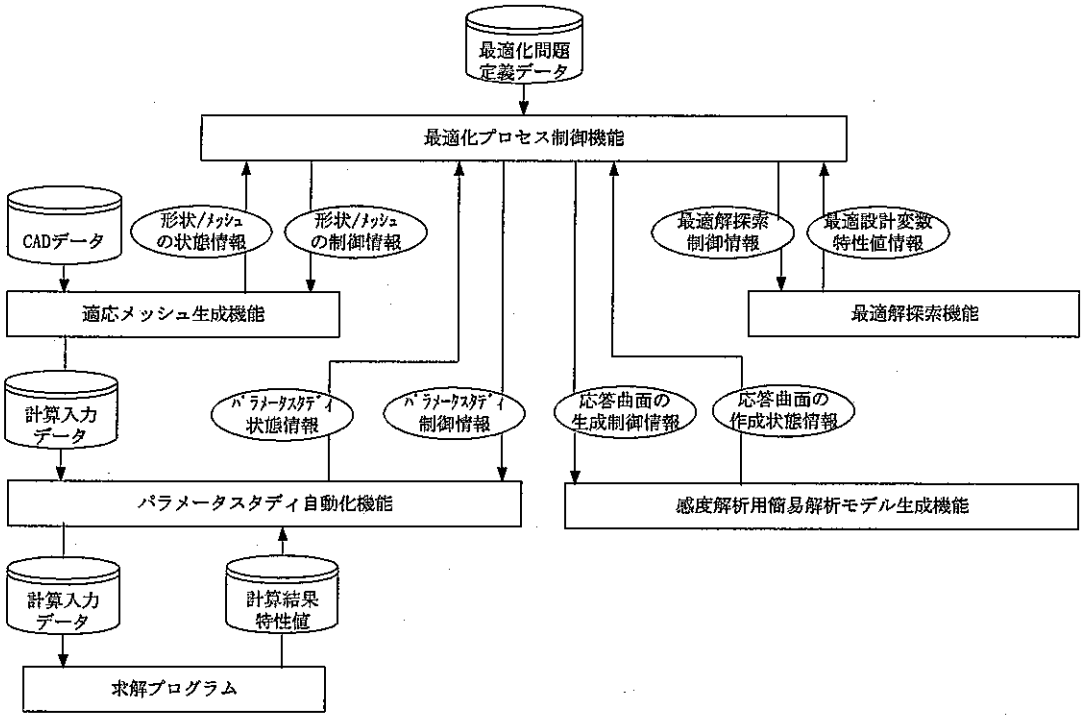


図 4.2.4.4-5　最適化問題対応機能

ここで、パラメータスタディ自動化機能から呼び出される求解プログラムは、「④求解及び求解支援機能」で開発し、目的関数の評価に利用される。

本機能の実装により、図 4.2.4.4-1 に示す科学・工学問題の問題解決プロセスのうち、Step5～7 のプログラム運用フェイズを強力に支援することができる。

④　求解及び求解支援機能

本機能では、「①統合化環境機能」上での問題解決に当たって必要とされる各種プログラム及び「③最適化問題対応機能」で目的関数を評価するために必要となる求解プログラムを開発した。以下に、本機能で開発されたプログラム群を示す。これらのプログラムは、すべて「①統合化環境機能」上に実装され、問題解決システムの構築に利用されるが、このうち、伝熱解析機能、熱流動解析機能、構造解析機能は、「③最適化問題対応機能」の求解プログラムとして利用される。

- プリプロセス部
  - a.　差分メッシュ生成機能
- ソルバ部
  - a.　乱流解析機能
  - b.　伝熱解析機能
  - c.　熱流動解析機能
  - d.　構造解析機能

・ ポストプロセス部

a. 乱流解析用可視化機能

上記のプログラムは、ある与えられた問題に対して、その解析領域や物体の形状、計算メッシュ、解析条件及び境界条件を定義し(プリプロセス部)、その問題を表す支配方程式を数値計算により解き(ソルバ部)、出力された結果を画面表示する(ポストプロセス部)という数値計算技術を利用して問題解決をするための想定プログラムである。したがって、この部分のプログラムが増えれば増えるほど本プラットフォーム(CAPSE)全体の問題解決能力が向上することになる。

本機能は、図4.2.4.4-1に示す科学・工学問題の問題解決プロセスのうち、Step5～7のプログラム運用フェイズを支援することになるが、ここで開発される機能に加えさらに多くの求解及び求解支援機能を追加／実装していくことが適用範囲の拡大という意味で重要となる。

⑤ 利用支援機能

利用支援機能は、図4.2.4.4-6に示すように既存のWebベースの質問応答システム(富士通株式会社)を利用して開発する。本機能は、作成された文書及びその目次を利用して文書を分割し、この分割された文書に対し有効なインデックスを割り当て、効率的な検索を共通のGUIで行うための検索エンジンである(知識ベース検索表示機能)。

本開発で作成された各種ドキュメント(外部設計書、取扱説明書等)を文書として登録し(知識ベース登録機能)、本検索エンジンを使用することにより、利用者は、CAPSE全体の利用及び運用に関する情報を迅速かつ的確に得ることができる。したがって、本機能は、図4.2.4.4-1に示す科学工学問題の問題解決プロセスのうち、Step1～7のプログラム作成及び運用フェイズを後方支援することになる。

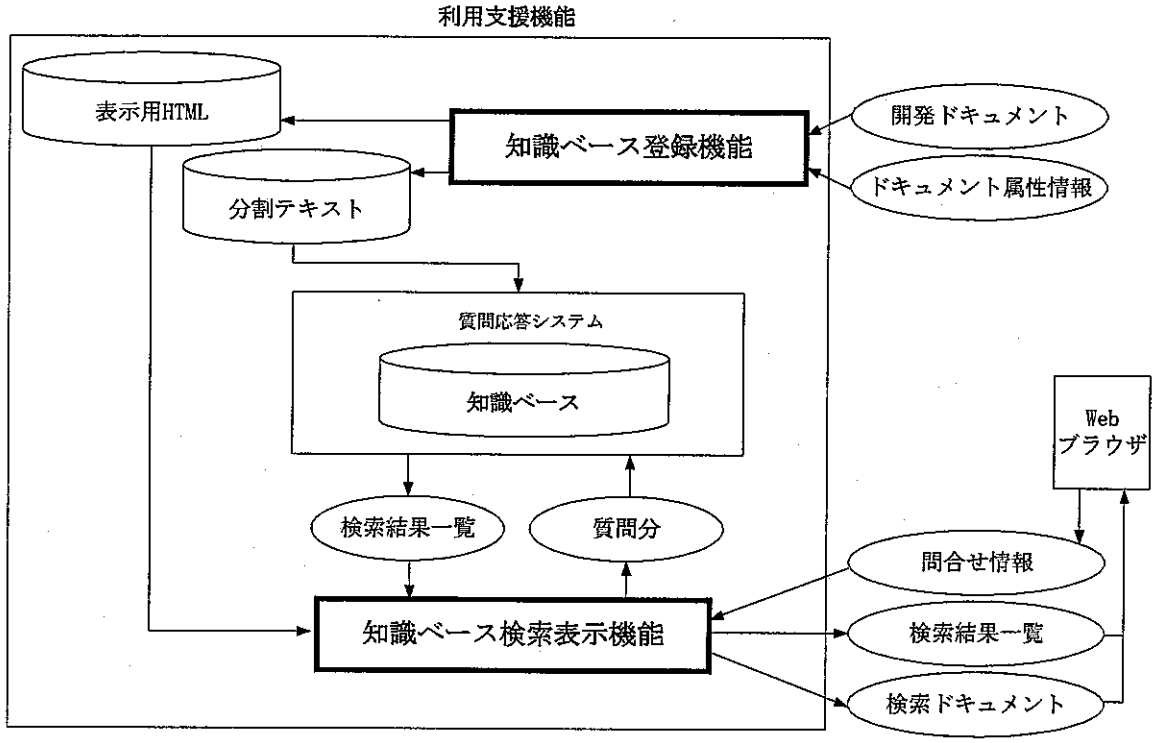


図 4.2.4.4-6　利用支援機能

4.2.4.5 バイオインフォマティクス

(1) データベースの例

表 4.2.4.5-1 は、ゲノム研究等で利用される代表的なデータベースの例であり、この表は、科学技術動向研究センターにより作成された表である。これについて、以下のように、説明されている。

一般に、タンパク質に比べて DNA の方が精製しやすく、DNA シークエンサーを用いることにより配列を比較的容易に決定できることから、DNA 塩基配列のデータベースが最も規模の大きいものになっている。DNA 塩基配列の公共データベースには、ヒト遺伝子の DNA 塩基配列や一塩基多型（SNPs）のデータベースなどがある。

また、タンパク質のアミノ酸配列、機能を予測するために有用なモチーフ配列、立体構造のデータベースなどがある。このようなデータベースは世界に約 400 種類あると言われている。

日本に関して言えば、パスウェイデータベースなどいくつかのデータベースを除いて、外国からも頻繁にアクセスされるデータベースは数少ないと言われている。

表 4.2.4.5-1 データベースの例

データベースの主な内容	データベース名称（運用国）	データベースの主な内容	データベース名称（運用国）
DNA 塩基配列	GenBank（米）、EMBL（欧）、DDBJ（日）	アミノ酸配列	SWISS-PROT（欧）、PIR（米）
ヒト遺伝子の DNA 塩基配列	UniGene（米）	アミノ酸配列ドメイン	Pfam（欧）
一塩基多型	dbSNP（米）、JSNP（日）	アミノ酸配列モチーフ	PROSITE（欧）、BLOCKS（米）
遺伝病	OMIM（米）、Mutation Database（欧）	タンパク質立体構造	PDB（米）、SCOP（欧）、CATH（欧）
総合的なヒトの配列情報	HGREP（日）、Ensembl（欧）	パスウェイ	KEGG/PATHWAY（日）
ヒトの総合情報	LocusLink/Refseq（米）、GDB（カナダ）	文献	MEDLINE（米）

(2) データベース検索システムの例

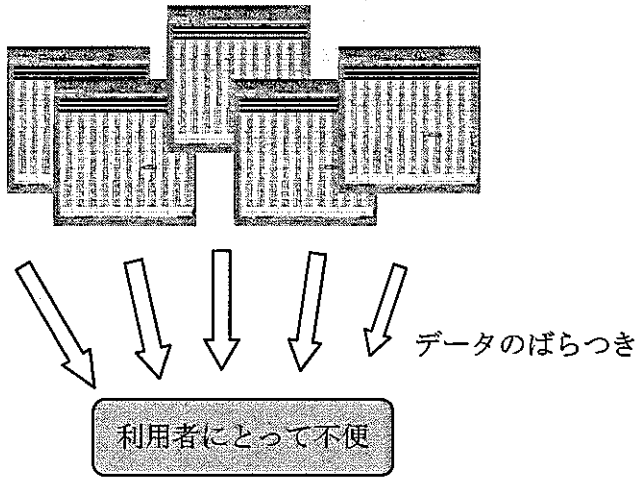
バイオインフォマティクスを用いた解析では、一種類のデータベースだけを用いることは少なく、様々なデータベースや検索システムを組み合わせて行われている。科学技術動向研究センターにより作成された、代表的なデータベース総合検索システムの例を表 4.2.4.5-2 に示す。

表 4.2.4.5-2　データベース総合検索システムの例

データベース総合検索システム	サービス提供機関
DBGET（日）	京都大学化学研究所、東京大学医科学研究所
Entrez（米）	米国国立バイオテクノロジー情報センター（NCBI）
SRS（欧）	欧州分子生物学研究所（EMBL）

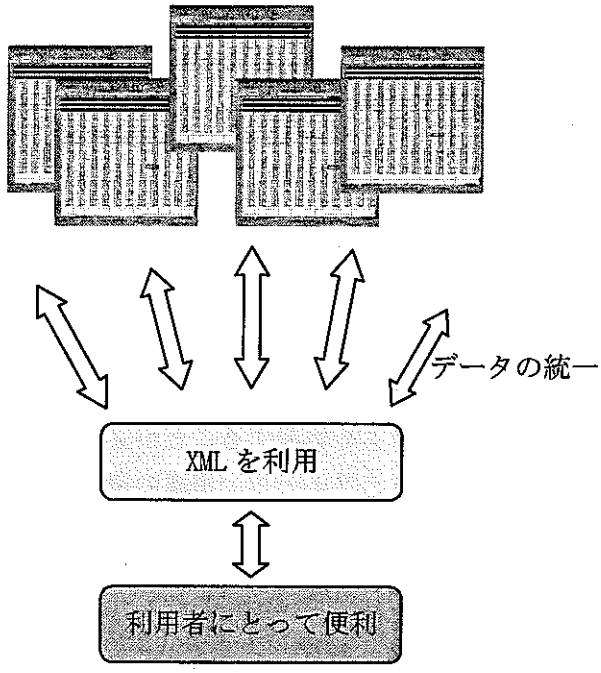
(3)　XML 利用の経緯

ゲノム解析によるデータは、塩基配列、制御情報等、多岐にわたり、各研究機関による成果がインターネット上でも多く公開されている。ゲノムの機能を解明するためには、これらの大量の情報を統合的に効率よく処理することが必要不可欠である。しかし、これらのデータベースはスキーマやインタフェース、用語の持つ意味などが統一されていないことや、更新修正が頻繁であることなどの問題がある。



上述の問題を解決するために、データ交換フォーマットとしての XML が注目され始めている。すでにゲノム研究のコミュニティでは用語や概念体系の統一の動きが大きくなっているらしく、データ表現のための標準 XML タグセットも将来広く使われるようになる可能性があるという。

XML を利用することにより、データの交換・再利用、プログラムによる効率的な処理が容易になるなどのメリットが考えられる。たとえば、利用者は各データベースの構造やそれらの違いをあまり意識することなく複数のデータベースへの問合せを行うことが出来るようになる。また、得られた問合せの結果を、そのままプログラムに渡して各利用者に必要な処理を行うための、プログラムの開発も容易となることが期待されている。



XML をバイオインフォマティクスに利用する際に必要な、XML データの格納法や問合せ、データ表現の仕方などについて研究が行われている。

(4) 最近の動向

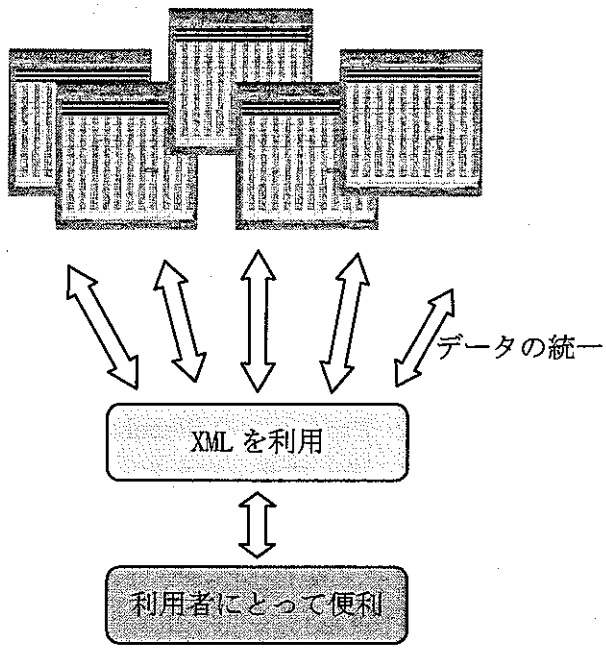
最近では、XML のバイオ版である BSML (Bioinformatic Sequence Markup Language) が普及しつつあり、この BSML とは、ゲノム研究のための標準 XML データフォーマットである。

1997年に開発された公開標準規約であり、ライフサイエンス分野の共通プロトコルを推進するI3C(The Interoperable Informatics Infrastructure Consortium)に採用されている。

BSML は、遺伝子の塩基配列やバイオインフォマティク・コンテンツの統合、注釈付け、視覚化の際に、必要となる情報を XML で規定したもので、新しい表示形式の技術である。

BSML はバイオインフォマティクスに関係するコミュニティでの利用を想定し、オープンなパブリック・ドメイン(フリー)標準規格として作成している。

BSML を利用することで、複数の異なるシステム間でデータの交換や再利用が容易になる。さらに、データベースの登録／検索のインタフェースが共通化されることで、データベースの違いを意識せずに利用できるようになる。このため、今まで充分に活用できなかった多種・多量データの有効活用が可能となり、研究開発のスピードアップにつながる。



導入事例

ソフト

・株式会社日立製作所　ライフサイエンス推進事業部

http://www.hitachi.co.jp/LS/index.html

解析サービス

ＤＮＡ塩基配列解析サービス

遺伝子多型解析サービス

遺伝子発現解析サービス

蛋白質機能解析サービス

バイオインフォマティクス支援サービス（整備中）

・日立ソフトウェアエンジニアリング株式会社

http://www.hitachi-sk.co.jp/

バイオインフォマティクスソフトウェア

バイオ情報処理環境構築システム

ヒトゲノム情報閲覧・検索ソフトウェア　他

http://bio.hitachi-sk.co.jp/product_2.html

・他

4.2.4.6 企業アプリケーション統合（EAI：Enterprise Application Integration）

(1) 概要

EAIとは、「企業内アプリケーション統合（Enterprise Application Integration）」と直訳そのままだが、おおまかには複数のアプリケーションのデータを共通化する（もしくは統合する）「データ統合」の側面と、アプリケーションを連係させる「アプリケーション統合」の側面がある。

企業内には、さまざまなアプリケーションが稼働している。例えば、営業部門ではSFA（Sales Force Automation）アプリケーションのSiebelが、工場ではSCMのためにi2が、本社ではERPとしてSAP R/3が、そしてサービスセンターにはOracleベースの独自アプリケーションが稼働している、といった具合である。しかし正確な販売戦略を立てるためには、生産計画と顧客動向の両方を分析すべきであり、別々に稼働している営業部門と工場の情報システムの連係が必要である。

このために有効なのが、両方のシステム間で矛盾していたり、重複して管理されたりしているデータをあらためるデータ統合であり、生産計画から導き出された結果を販売計画に反映させるといった、アプリケーションの連係を可能にするアプリケーション統合である。このように、既存のシステムを作り替えるのではなく、そのままのシステムでデータやアプリケーションの統合を実現し、新しい価値を作り出すことがEAIの目的といえる。

①EAIの実現に必要な機能

単純なEAIとは、複数のデータソース間（すなわちアプリケーション）を結んで夜間などにデータ交換を行い、バッチ的なデータ統合を実現する、といったものになる。全社的にデータを統合するだけでも、データの精度が高まり、重複がなくなることで効率が高まり、プロセスの速度をあげることが可能になる。

これに対し高度なEAIとしては、複数のアプリケーション間でのデータ受け渡しをリアルタイムに行い、あるアプリケーションの結果をほかのアプリケーションに反映する、といった方法で密なアプリケーション統合を実現するものがある。さらに、中央に統合データベースを構築し、常にそこに全社のアプリケーションの最新データを蓄積して分析に利用する、データウェア的な機能を含むものなどもある。

上記のような機能を備えたEAIソフトウェアを提供する専業ベンダは、以前から存在していた。代表的なものは、Tibco、Vitria Technology、BEA Systems、WebMethods、SeeBeyondなどだ。これらの専業ベンダは、特にメインフレームなどとの接続に強く、リアルタイムなデータ統合・アプリケーション統合のためのアーキテクチャといった高度な機能を備えている。また、SAP/R3などERPアプリケーションも、外部のアプリケーションと連係するためのアダプタやAPIなどが用意されているため、ERPなどのアプリケーションを中心としたEAIの実現も選択肢の1つではある。

こうしたEAIに求められる機能を簡単にまとめてみると、下記のようになる。

① インフレーム、SAP/R3、MQSeries、Oracle、PeopleSoft、SQL Server、CORBA プレーンテキストなど、さまざまなデータソースへの接続機能
② データソースからのデータを別の形式に変換する変換機能
③ アプリケーションの接続方法、統合方法や、データ変換に関するロジックやフローを記述するビジネスロジック機能

②EAIの導入効果

EAIを導入すると、従来のレガシーシステムがWebシステムなどと連携できるため、資産の有効活用が行えるようになる。また、これまで別々に存在していた業務アプリケーションが統合されることによって、あるデータを1カ所に入力すれば、他のアプリケーションにおいてもリアルタイムに更新されるようになり、データの二重入力が回避され、入力作業が軽減されるといったメリットももたらされる。

そして、データの精度も向上される。これを簡単に説明すると、異機種間のデータ連携を行なうためには、まず、それぞれのデータやフォーマット形式を整えなければならない。

例えばAシステムでは「B」、Cシステムでは「D」というようにデータ形式が異なる場合、「E」というデータ形式に統一する必要がある。そこで、データのクレンジングが行なわれ、必然的にデータの精度が上がるというわけである。さらに、システムやビジネスプロセスが統合されることにより、お金の流れや製品の動きが明確になり、リアルタイム処理が可能となるため、意志決定スピードの加速化や迅速な顧客へのサービス対応なども可能になる。

(2) EAI ツールの接続形態

EAI ツールの接続形態は、自転車のハブ＆スポークの関係に似ている。
EAI が実現する統合の世界は、従来のようなデータベースとデータベース、データとデータといった１対１の関係ではなく、多対多の世界を実現するのである。
EAI ツールをハブ（核）とし、その先に各業務アプリケーションや既存システムなど、さまざまな要素が存在する。

つまり、EAI ツールを中心に置くことで連携部分の処理を集中化し、データウェアハウス、ERP、メインフレーム、各業務アプリケーションなど個々に存在するシステムの接続インタフェースを最小限に抑え、有機的な連携を保つのである。

このハブ＆スポークの接続環境が整うと、次のステージはプロセスの統合になる。

いくら多対多の関係が成立しても、ユーザーの立場からみると連携を取るシステム間の連携は多対 1、あるいは１対多という関係になるであろう。

問題は、接続後のプロセス連携をどのように実現するかに集約される。

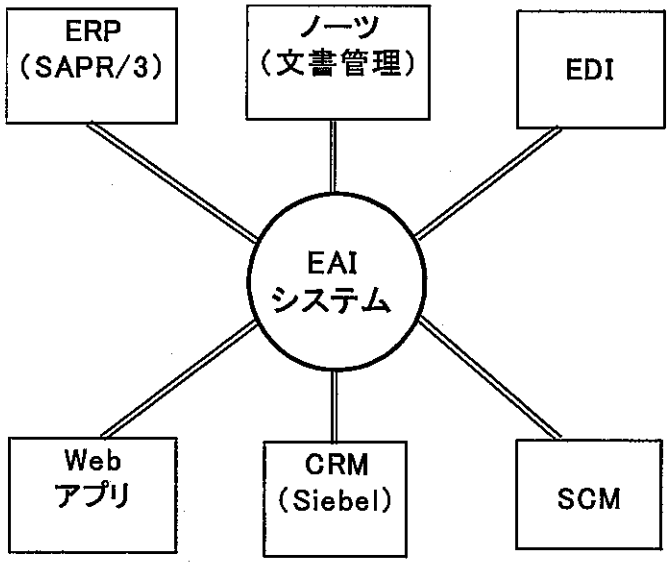


図 4.2.4.6-1　EAI による連携（ハブ＆スポーク型）

(a) EAI の構成

EAI を実現するソフトウェア(EAI ツール)はミドルウェアの一種で各システムへのインターフェースを提供する「アダプタ」システムごとのデータ形式やプロトコルの違いを吸収する「フォーマット変換」あるシステムから受け取ったデータを内容に応じて他のシステムに振り分ける「ルーティング」これらの機能を組み合わせ、実際の業務に合わせたビジネスプロセスを構築する「ワークフロー(プロセス制御)」などの機能から構成される。

EAI は企業内システムの統合にとどまらず、企業間の電子商取引を実現するためのシステム間接続や企業買収・合併に伴う情報システムの統合を効率よく行なうための手段とし

ても利用される。

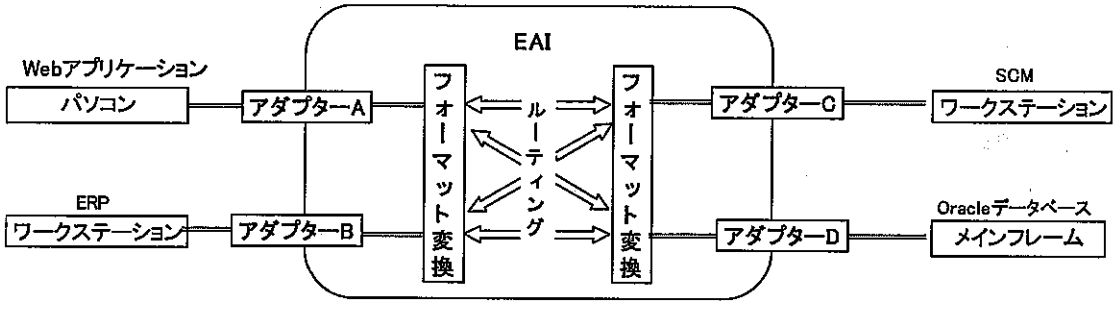


図 4.2.4.6-2　EAI 構成図

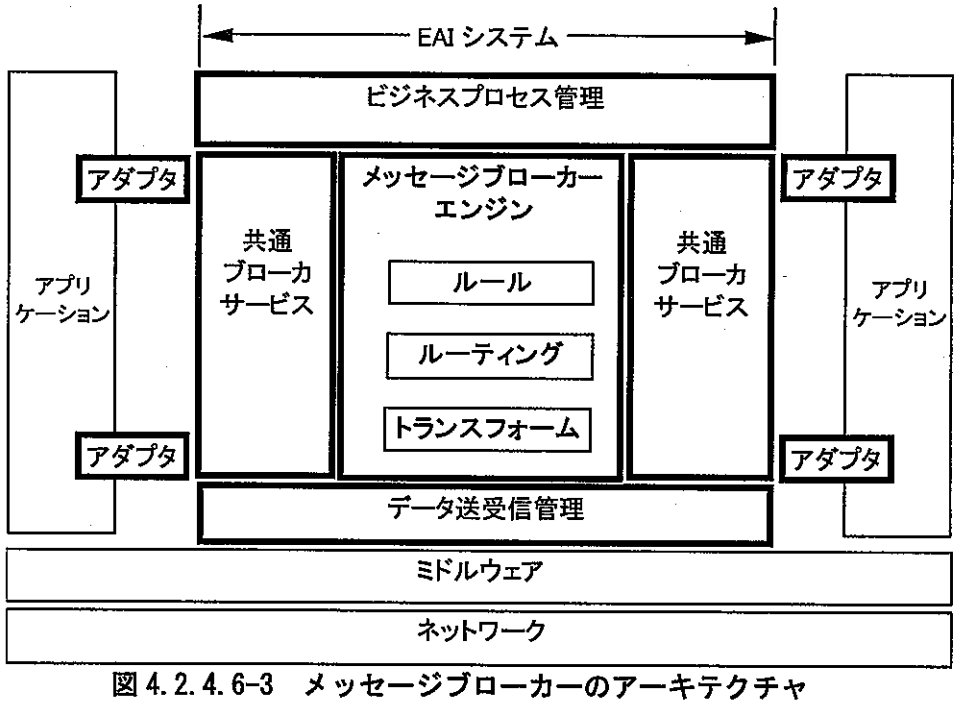


図 4.2.4.6-3　メッセージブローカーのアーキテクチャ

(b)　各構成要素の説明

①メッセージブローカー

メッセージブローカーは、複数のプラットフォーム上に存在するパッケージアプリケーションやビジネスロジック、各種のデータ、そしてミドルウェアなどを連携するための仕組みを提供する EAI エンジンである。

メッセージブローカーは以下の機能を提供する

- ルール機能
- ルーティング機能

・トランスフォーム機能

メッセージブローカーは、上記の 3 つの機能をベースに「EAI のエンジン」として動作して、アプリケーション統合環境を実現する。

②ルール機能

データ送信の際に、業務によって異なるルールを正しく処理する機能である。 企業間データ送受信の際に、特定形式の送信データを相手方である受信先企業が拒否するような場合のある。 たとえば、「EXE 形式のデータは受け取らない」という取り決めがある送信先が相手の場合、このルール機能を使って該当取引先には EXE ファイル形式 のデータが送信されないよう対処し、また、後述のトランスフォーム機能で妥当な形式に変換することができる。

③ルーティング機能

送信元のアプリケーションからのデータを、指定された適切なターゲットアプリケーションに送信する機能である。 ビジネス上でのデータのやり取りでは、1 対 1 のデータ送信ばかりでなく、1 対多(n)というような形式のデータ送信が発生する。 こんな際、ルーティング機能が適切な送信先へのルーティングを行う。

④トランスフォーム機能

最新先アプリケーションに対応したデータフォーマットへの変換機能である。例えば、送信元から HTML 形式で送信されたデータに対し、受信する側がテキストメールしか扱えない状況だった場合、このデータを受信側が利用できるフォーマット（この場合はテキストメール形式）に変換する機能である。

⑤アプリケーションアダプタ

アプリケーションアダプタ（図中のアダプタ）は、メッセージブローカーのインターフェースである共通ブローカーサービスとアプリケーションの間でデータ交換を行う。
アダプタの種類は、アプリケーションがメッセージブローカーにデータを送信するための「送信アダプタ」、逆にデータを受信するための「受信アダプタ」、そして、アプリケーションがメッセージブローカーからデータを受信して、その結果をメッセージブローカーに送信する「送受信アダプタ」がある。

⑥共通ブローカサービス

アプリケーションとやり取りを行う際には、アプリケーション側にアダプタが必要となる。このアダプタとメッセージブローカーの共通インターフェースを共通ブローカーサービスと呼ぶ。

⑦データ交換時の接続インターフェース

BizTalk Server と WebLogic との接続では双方に HTTP プロトコルを使用する。
これに対し BizTalk Server と SAP R/3 でのデータ交換では
BizTalk Server→SAP R/3 への通信では COM インターフェース（BAPI、RFC）を使い
逆にの通信では FTP を使ってファイル転送する。

EAI システムが備える主な接続インターフェース

- COM、MSMQ（Microsoft Message Queuing）
- Oracle、SQL Server 等の主要なデータベース製品
- SAP、SAP R/3 等の MQSeries
- メインフレーム
- CORBA、EJB（Enterprise Java Beans）、JIMS（Java Message Service）等

これ以外の場合は、アダプタ（接続インターフェース）を独自に作成し対応する。

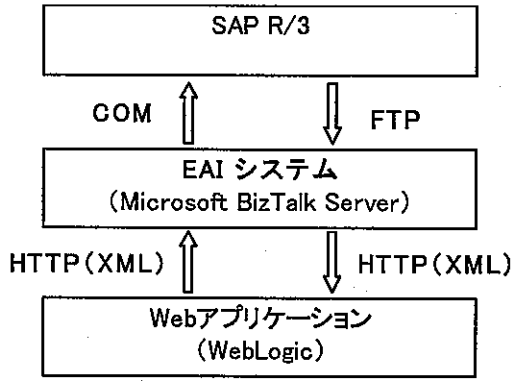


図 4.2.4.6-4　データ交換時の接続インターフェース

(c) EAI のレベル

EAI にはさまざまなレベルがあり、それぞれどのようなレベルの EAI があるのかを説明する。

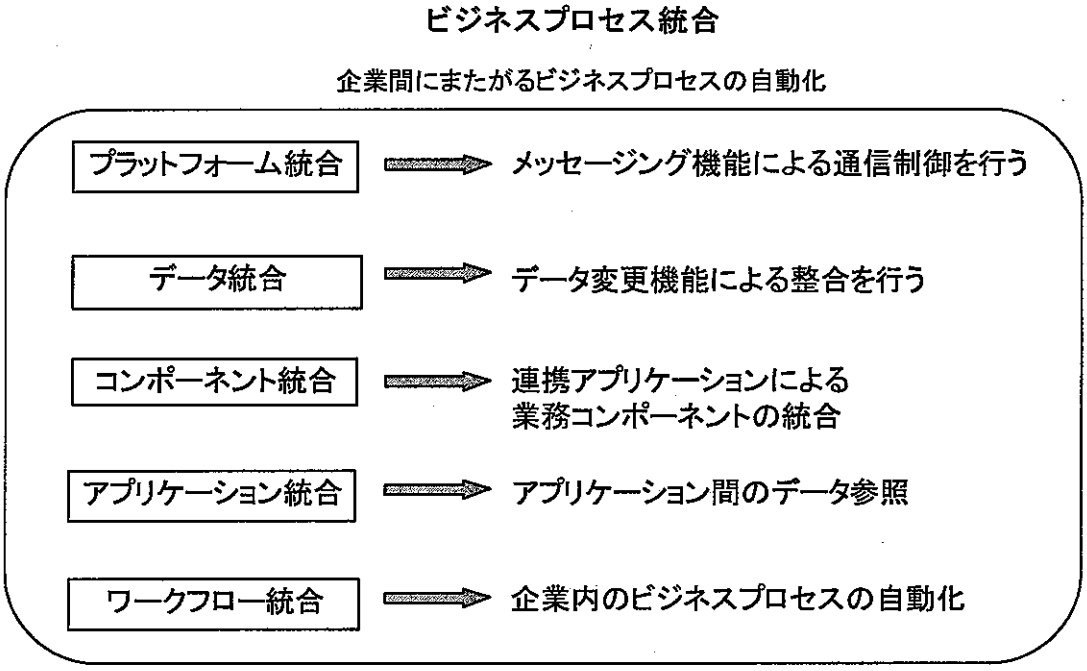


図 4.2.4.6-5 EAI の分類

■レベル 1：プラットフォーム・インテグレーション

（メッセージによる通信制御を行なうプラットフォームの統合）

このレベルではシステム間、アプリケーション間の通信プロトコルの統合が行われる。
例えば、メインフレームとクライアント／サーバーなどを連携したい場合
互いに異なるプロトコルが使われているため、そのままでは連携ができない。
そこで、MQ（メッセージキューイング）技術や通信プロトコル変換機能を使って統合を実現するのである。

■レベル 2：データ・インテグレーション

（データ変換による整合をとるデータ統合）

データレベルの統合では、データの正規化やデータマッピングを行なうために
ETL（Extract Transformation Loading：データを抽出し、変換、加工、流し込みを行なうこと）ツールが用いられる。
異なるアプリケーション間におけるデータの統合を実現するのが、このレベルの特長である。

■レベル3：コンポーネント・インテグレーション

（ゲートウェイなどのアプリケーションによる業務コンポーネント統合）

コンポーネントの統合に必要なものが、アプリケーションサーバーである。
例えば、パソコン上のWebブラウザでホストシステムのデータの入出力を可能にする場合ホストシステムに接続されたゲートウェイサーバーを経由し、Web サーバーに端末エミュレーターを導入することで解決できる。
つまり、このレベルは、レガシーシステムを変更することなく
新しい技術を使ったシステムのように見せかけることで統合を実現するのである。

■レベル4：アプリケーション・インテグレーション

（アプリケーション間のデータ参照レベルでの整合によるアプリケーション統合）

このレベルでは、例えば、ERPとCRMなどのアプリケーション同士の連携が可能になり、しかもそれぞれ異なるはずのデータレベルの統合までも可能にする。
いわゆる1対1のアプリケーションにおいては、完全に一つのシステムとして統合されるのである。

■レベル5：ワークフロー・インテグレーション

（単一企業内のビジネスプロセスの自動化を伴うプロセス統合）

この統合はワークフローモデリング、プロセス・サーバーなどで実現する。このプロセス・インテグレーションでは企業内のプロセスが統合されるため、ビジネス効率の向上が期待できる。

■レベル6：ビジネスプロセス・インテグレーション

（複数企業間にまたがってビジネスプロセスの自動化を伴うB2B統合）

この統合レベルは、EDIやSCM（サプライチェーン）などの導入によって実現できる。
企業内を越えた企業間のビジネスプロセスまでも統合可能としているため、SCMなどにおいても本格的な効率化が図れる。
このレベルはEAIの最終目標となる。

4.3 ネットワークコンピューティング技術

現実的な数値計算は、膨大なメモリ容量と計算時間が必要となり、ネットワークコンピューティング技術を用いた並列・分散処理が不可欠である。ここでは、それらを実現するための、ソフトウェア環境及び方法について取り上げる。

4.3.1 MPI

4.3.1.1 概要

MPI(Message Passing Interface) とは、分散メモリ環境における並列プログラミングの標準的な実装である。その名の通りメッセージパッシング方式に基づいた仕様であり、MPI の仕様に準じた実装ライブラリは、複数存在する。その中の幾つかはフリーで配布されており、UNIX 系を中心として Windows、Mac とほぼ全ての OS、 アーキテクチャに対応している。そのため、どのようなクラスタ環境においても MPI はフリーで使用することができる。

PC クラスタのような分散メモリシステムにおいて、並列化インタフェースは大きく 2 つに分類される。
一つは、 メッセージパッシング方式であり他方がデータ並列方式である。以下、MPI が用いているメッセージパッシング方式について説明する。

<u>メッセージパッシング方式</u>

メッセージパッシング方式とは、プロセッサ間でメッセージを交信しながら並列処理を実現する方式である。並列処理では、複数のプロセッサが通信を行いながら同時に処理を進めていく。そこで問題になるのがプロセッサ間の通信であるが、メッセージパッシング方式ではプロセッサ間での通信をお互いのデータの送受信にて行う。
そのため、

- プロセッサ i 上でプロセッサ j にデータを送る。
- プロセッサ j 上でプロセッサ i からデータを受ける。

が必ずペアとして成り立たなければならない。

MPI による並列プログラミングの基礎 34 メッセージパッシング方式は、分散メモリ環境において現在、最も主流の方式であり、かなり本格的な並列プログラミングが実現することができる。この方式をプログラムで実現するためのライブラリが、 メッセージパッシングライブラリである。 その最も代表的なライブラリとして MPI と PVM がある。

これら並列ライブラリを用いたプログラムを複数のパソコンにインストールし、ネット

ワーク経由で接続すれば並列処理システムが出来上がる訳である。

4.3.1.2 必要性と効果

かなりの計算能力を必要とする大規模問題を解くためには、従来スーパコンピュータを使用してきたが、計算機（ハードウェア）費用、ソフトウェア費用、構築費用そして保守運用費用等の莫大なコストが掛かり、利用出来るのは国の研究機関・大学・大企業といったごく一部のユーザに限られてきた。

しかし、大規模問題解決に対するニーズは高く、もっと安く手軽に個人レベルで利用出来る計算機環境が欲しいという要求に基ずき、世界中の有志により「パソコンによるクラスタ（並列処理）システム」に必要な並列計算ライブラリ（MPI）が開発された。

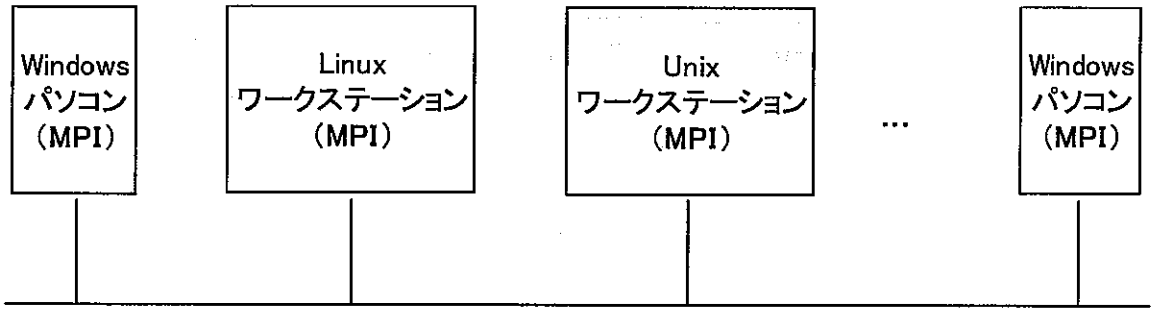


図 4.3.1-1 クラスタシステム構成図

① PC クラスタの構成要素

・ パソコン ×n 台

・ OS

Linux、WindowsNT

・ 並列ライブラリ

MPI、PVM ... 何れもフリーウェア

・ ネットワーク

100Mbps スイッチングハブ等による LAN

PC の台数にもよるがスーパーコンピュータにも匹敵する性能を持つシステムを構築することが可能である。

クラスタを所有すると以下のようなメリットがある。

- 数値計算の向上(「HPC(High Performance Computing)」と呼ばれる)
- トータルコストは、スーパコンピュータに比べ格段に安価で済む
- 可用性の向上(「HA(High Availability)」と呼ばれる)
- 計算機環境の専用(スループットの向上)

・ 古いマシンの有効活用
・ 技術進展に追従したシステムのバージョンアップが可能

等

② PC Cluster

PC Cluster は、汎用的な PC を多数集めて構築される並列計算機のことである。PC Cluster はコストパフォーマンスが高いことから、最近では数多く作られるようになった。それに伴い、Cluster を構築・運用する際の技術も充実しつつあり、用途に合わせた Cluster を構築することが可能になった。

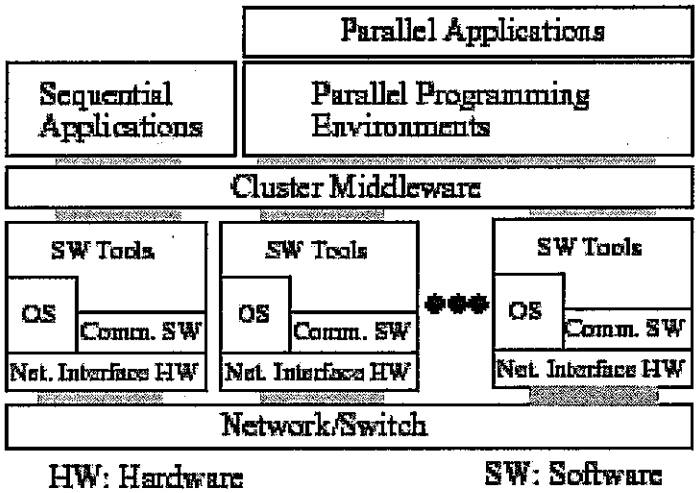


図 4.3.1-2 C クラスタのソフトウェア構成

4.3.1.3 MPI と他の並列ライブラリとの比較

① MPI と PVM の比較

MPI と PVM に関する議論は、これまでにも幾つか行われている。最近の動向としては、MPI が新規格である MPI-2 の仕様において、それまでの PVM にしか無かった「動的なプロセス管理」の機能を取り入れるなど活発な動きを見せており、MPI がかなり優勢な状況となっている。ここで両ライブラリの利点・欠点についてまとめる。

PVM は、ワークステーションをクラスタにすることで再利用するという世界で育ってきたものである。そのために不均質ないくつものマシンやオペレーティング・システムを直接に管理する。そしてダイナミックに仮想マシンを形成することができる。PVM のプロセス管理の機能では、アプリケーションの中から動的にプロセスを生成したり、停止したりすることができる。この機能は MPI-1 にはなく、MPI-2 で採り入れられている。さらに利用可能なノードのグループを管理する機能では、ノード数を動的に増減したり調べたりすることができる。これは、MPI ではまだ採り入れられていない。

PVM の利用における最大の問題点は、先にも述べた移植性である。それに対してMPI は、その実装がMPP (Massively Parallel Processors) やほとんど同一な特定のワークステーション・クラスタを対象としている。MPI は PVM の後に設計されたために、明らかにPVMにおける問題点を学んでいる。つまりMPI の方がPVM に比べて、高レベルのバッファ操作が可能であり、高速にメッセージを受け渡すことができる。そして、オープンなフォーラムによって達成された「標準」であるために、MPI で作成されたプログラムは非常に移植性が高い。ただし MPI の実装は数多くあるが、MPI-2 を完全にサポートしたものはまだない。また、Web や書籍などでの情報も MPI の方が入手しやすく便利であるといえる。

ここで、MPI のライブラリが集まる http://www.mpi.nd.edu/lam/lam-info.php3 では、「MPI が PVM より優位である 10 の理由」と題して興味深いリストが載っていたので紹介する。ここでは、MPI がPVMに勝っている点として、

1. MPI はフリーの実行可能かつ高品質なインプリメンションを 1 つ以上持っている。
2. MPI は第 3 者的な決定機関を持っている。
3. PC クラスタ超入門 2000 37
4. MPI グループは完全に非同期通信をサポートしている。
5. MPI グループはしっかりとした基盤と持ち能率的で決定論的な側面を持っている。
6. MPI は効果的なメッセージバッファを扱うことができる。
7. MPI における同時性は第 3 者的なソフトウェア機関により保護されている。
8. MPI は効果的なMPP、クラスタのプログラミングを行うことができる。
9. MPI は全体的にポータブルである。
10. MPI は正式に明示されている。
11. MPI が標準である。

を挙げている。上記の理由の主な論拠は、

- 複数の信頼性の高いライブラリの存在
- MPI フォーラムの存在
- MPI の移植性の良さ

などである。このことより、MPI において、MPI フォーラムの存在がMPI への信頼性の向上に大きく関与していることがわかる。また、PVMにおける最大の問題点である移植性の良さもMPI の最大の利点の一つとなっていることも分かる。

4.3.2 CORBA

4.3.2.1 概要

CORBA (Common Object Request Broker Architecture)とは、OMG が策定する標準仕様で、分散システム環境のインフラを整備、標準化するものである。CORBA は、OMG の制定する OMA (Object Management Architecture) と呼ばれるアーキテクチャを基盤とし、OMA は以下の 3 要素により構成されている。

① ORB (Ojbect Request Broker)
ORB は CORBA の最も基本となる要素であり、オブジェクトバスと呼ばれる。ORB を用いることで、ネットワーク上に分散した各オブジェクトはローカル、あるいはリモートのオブジェクトに対して透過的にアクセスすることが可能になる。

② CORBA サービス
CORBA サービスとは、ほとんどのアプリケーションで必要とされるシステムレベルのサービスの集合であり、ORB の機能を拡張し、補うものである。代表的なものにネーミングサービス (オブジェクトに対し名前でアクセスする)、イベントサービス (オブジェクト間のイベント送受信を行なう) などがある。

③ CORBA ファシリティ
CORBA ファシリティは CORBA サービスよりも抽象度が高く、アプリケーションに近い上位のサービス群である。CORBA ファシリティは、様々なアプリケーションに適用可能な水平型と、特定の産業分野 (ドメイン) に特化した垂直型の 2 つに分類することができる。

CORBA は、あらゆるオブジェクトのインタフェースを IDL (Interface Definition Language) と呼ばれるインタフェース定義言語で定義する。この IDL の存在により、異機種異言語相互運用性 (様々なプログラミング言語、プラットフォームにおける透過性) を達成している。この異機種異言語相互運用性こそが CORBA の持つ最も大きな特徴である。

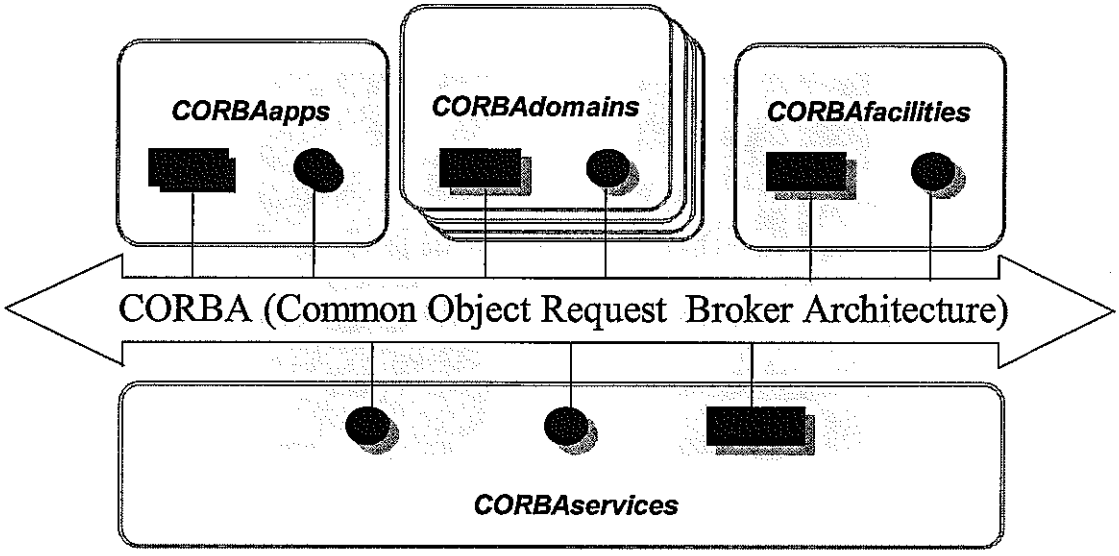


図 4.3.2-1　ORBA アーキテクチャ

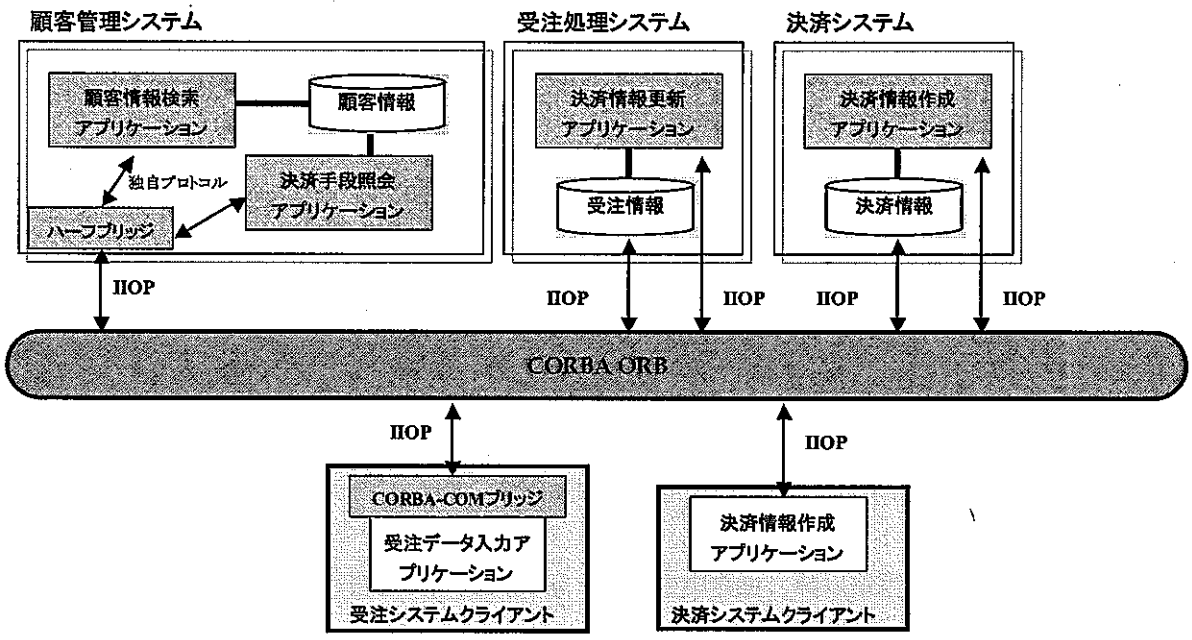


図 4.3.2-2　ORBA によるシステム例

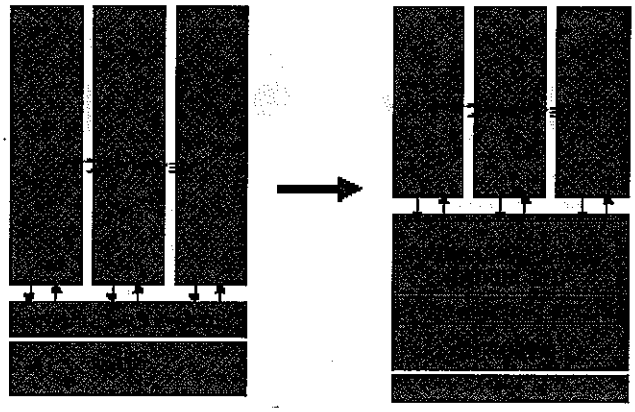


図 4.3.2-3　抽象度の高い通信モデル

4.3.2.2　必要性と効果

異なる環境で構築した幾つかのシステムを統合したいという強いニーズは以前からあった。これを実現するには、あらゆる計算機 OS そして通信プロトコルの違いを吸収した上でデータのやり取りを実現する共通基盤（フレームワーク）が必要である。将来の拡張性を考慮すれば標準化された技術である事が望ましい。しかし、統合化には以下に示すような要素の違いを吸収した技術仕様が必要である。

環境要素

- ハードウェアプラットフォーム
- オペレーティング・システム
- ネットワークプロトコルアプリケーション・フォーマット

そこでこうした問題を解決すべく、OMG（Object Management Group）という世界的な標準化団体がソフトウェアの相互運用を目的とした共通基盤（フレームワーク）CORBA を策定した、これを活用すれば統合化の問題を解決出来、しかも将来に渡る拡張性も保証される。

実際に CORBA1.0 策定以降、世界中の企業・研究機関等でさまざまな既存システムの統合化が実現され大きな効果をあげている。

そもそも CORBA の目的は、あらゆる技術を繋ぐ効果的なインフラを提供することにある。COM が登場した 1995 年、OMG は COM と CORBA の相互運用を実現した。DCOM や Java、ERP システムとも相互運用を行うことができる。そして現在、OMG は XML と CORBA を統合しようとしている。

4.3.2.3 CORBAのメリット

CORBAのメリットとしては以下のようなことが挙げられる。

・相互運用性インタフェース
・すでにCORBAは企業の異機種分散システム構築における デファクト・スタンダードとなった。
・システムインテグレーションを経済的に行うためのインフラ
・いかにうまく将来を管理するか
・抽象度の高い通信モデルを提供
・インフラを共通化出来る、つまりアプリケーション開発に専念出来る。
・すべてを自前で作ることではなく、標準技術を使っても他社と差別化すること
・負荷の分散ではなく、協調して実行する関係である

・CORBAによる相互運用性
. プロセス／マシン間
. 言語間
. ORB間
CORBA2.0で規定
. GIOP (General Inter-ORB Protocol)
. IIOP (Internet Inter-ORB Protocol)
. IOR (Interoperable Object Reference)
. オブジェクトモデル間
. COM、OLEオートメーション、DCOM
. CORBA2.0で規定
. 通信メカニズム間
. DCE
. CORBA2.0で規定

CORBAの進化を評価するのにキーとなる概念は以下のものである。

・自律、分散、協調
・瞬発性、柔軟性、持久性（人間の運動能力）

CORBA最大の特徴である異機種間相互運用を実現している重要な概念がある。
それがオープンな「ソフトウェア・バス」という考えであり、

・迅速なシステム間／アプリケーション間の統合を支援

・クライアント／サーバ・アプリケーションの開発を容易にする
・オブジェクトのインタフェースが CORBA に準拠するならば、相互運用性が保障され、オブジェクトは互いに通信できる

というものである。

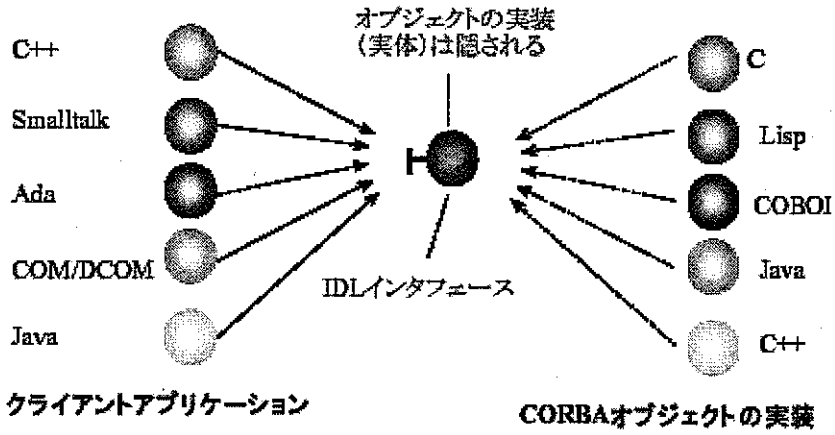


図 4.3.2-4 インターフェースと実装の分離
（ソフトウェアバスのキーとなる概念）

なお、CORBA に類似する技術として以下のようなものがある。

・CORBA＝実装非依存デファクト・スタンダード
・DCOM＝マイクロソフト Windows 環境依存
・Java RMI＝Java 依存

4.3.3 SOAP

SOAP (Simple Object Access Protocol) は、既存のインターネット・インフラストラクチャ(HTTP や SMTP)と XML を使って、RPC (Remote Procedure Call)を実現しようというものである(1998 年当時マイクロソフトに在籍していた、Dave Winer により設計された XML-RPC を拡張したものである)。HTTP を使って RPC を実現することにより、インターネット・ファイアウォールに意図しない形でブロックされることはなくなる。ここが、既存の分散オブジェクト・プロトコルと大きく異なっているところで、SOAP の最大の利点である。また XML を使っているので、interoperability も実現される。SOAP は名前の通りシンプルさと拡張性を設計目標にしているため、分散オブジェクト・システムの全ての特徴を満たしているというわけではない。

SOAP は、DevelopMentor 社、IBM 社、Microsoft 社、Lotus Development 社、UserLand Software 社の各社によって仕様が策定され、W3C に提出された。

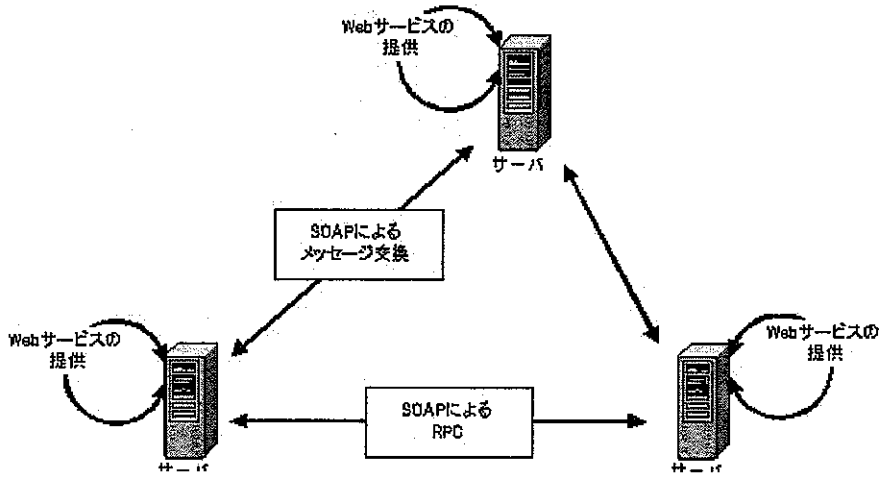


図 4.3.3-1 Web サービスにおける SOAP の役割

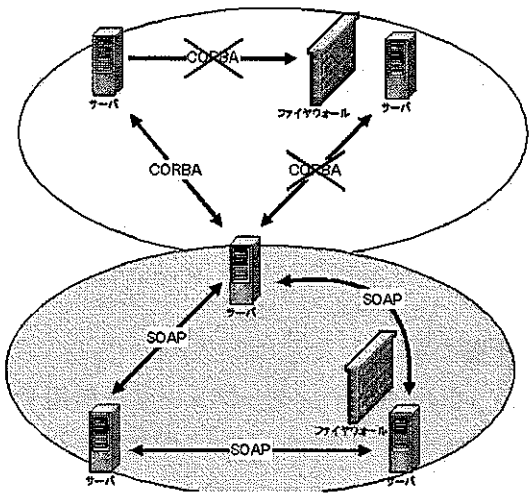


図 4.3.3-2 SOAP と既存技術の比較

SOAP に規定されている事柄

● エンベロープ構成要素
├ メッセージ構成
└ 処理仕様
● エンコーディング規則
└ データのシリアライズメカニズム
● RPC 表現規則
└ 要求と応答の規則

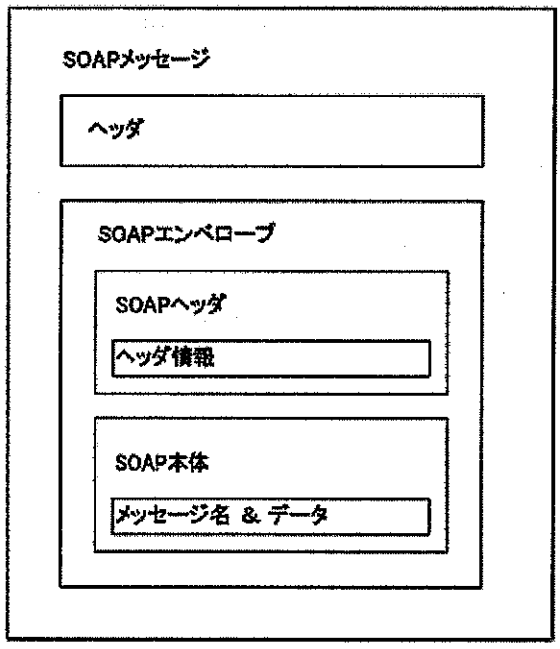


図 4.3.3-3 SOAP メッセージ構成

SOAP が有効に活用される領域は、インターネットを介したシステム間の連携においてである。HTTP 及び XML という一般化されたインターフェース技術を活用するため、その部分に関しては特別な作り込みを必要とせず、プロセス間の通信部分のみを作成すれば良い。

名称が示す通りシンプルな技術仕様であるため、プロセス間の通信部分についても比較的簡単に開発する事ができる。

主な特徴を示す。

- シンプルな技術仕様である。
- HTTP/XML という汎用的なインターフェースを活用するため、CORBA のようにファイヤウォールを通過出来ないという問題も発生しない。

但し、技術仕様が確定していないため、今後大幅な変更が発生するかもしれないという課題も抱えている。

4.3.4 Microsoft.NET

4.3.4.1 概要

2002年Microsoftが発表した.NET Technologyは、「.NETプラットフォーム」という基盤技術の上に成り立っている。.NET Technologyは広範囲な技術体系で、インターネットの標準プロトコル（HTTP）やオープンな規格（XML、SOAP等）に基づいてさまざまなWebサービスを提供することを目的としている。

.NETプラットフォームは、インターネットでの標準プロトコルやオープンな規格に基づいて構築されたプラットフォームである。Windows OSは、.NETプラットフォームを実装するための重要な手段の１つであるが、.NET プラットフォームのターゲットは、WindowsOSではなくインターネット自身なのである。

また、まだ始まったばかりではあるが、オープンソースの.NETプラットフォームを開発しようとしているMonoプロジェクトとよばれる計画も存在する。このMono計画では、UNIX（Linux）上で動作する.NET プラットフォームの開発を目標としている。Mono計画は、Ximianという企業が中心となって進めており、オープンソースによるC#の開発も行われている。（参考文献：オープンソース版.NET プロジェクト（Mono）、http://www.go-mono.com/）

「サービス指向」は、.NET プラットフォームの本質を表す重要なキーワードで、.NETプラットフォームはユーザ本位のサービスを提供するものであり、特定のOSやアーキテクチャに関係なく、ユーザは透過的にサービスを利用できるようになる。.NET プラットフォームは特定の実装手段に依存しない、インターネットを基盤としたオープンなプラットフォームである。

その目標は、既存システムを含む複数組織（複数企業）を、XMLやSOAPなどのオープンな規格を使って統合することにある。それぞれのシステム内部は、異なるテクノロジーを使って実装されている場合もあれば、基盤となるOSが異なることもありえる。

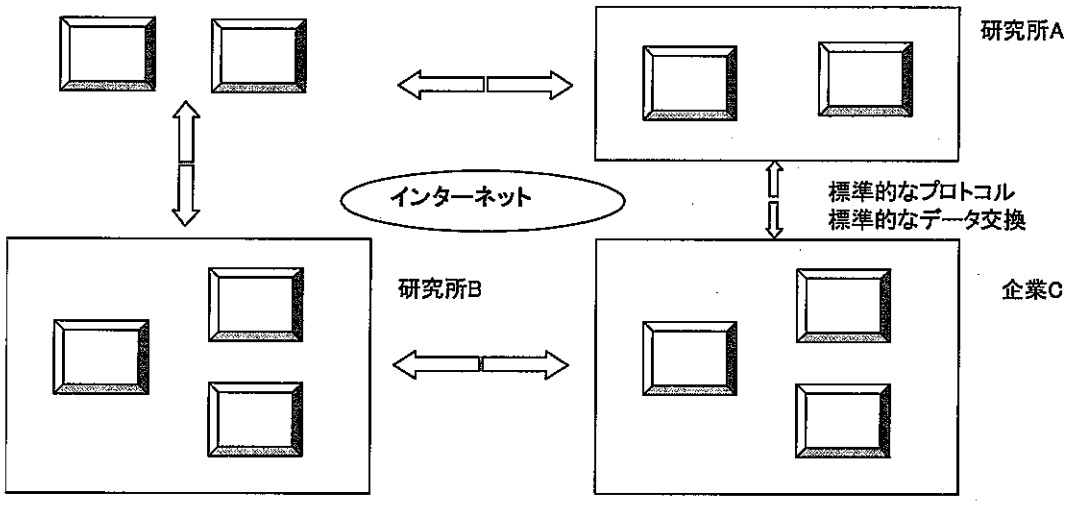


図 4.3.4-1 .NET プラットフォームによる既存システムの統合

4.3.4.2 .NET プラットフォームの構成要素

.NET プラットフォームでは、Web 上に公開される様々なシステム（サービス、機能）や既存のコンポーネントを統合して、１つの付加価値あるサービスを提供する。このような統合を「オーケストレーション」と呼ぶ。

① .NET Framework

.NET プラットフォーム向けアプリケーションのための開発環境及び実行環境を提供するフレームワークである。このフレームワークに対応した開発ツールが VisualStudio.NET である。

② .NET Enterprise Servers

.NET プラットフォームに基づいて構築されるサービスのバックエンドを支えるサーバ製品群である。Web 上に構築されるサイトを支えるため、データベース管理、メッセージング、ワークフロー管理、負荷分散などを行うサーバ群である。

③ ビルディングブロックサービス

「ビルディングブロック」とは、もともと積み木の意味である。.NET プラットフォームでのオーケストレーション（統合）をするための部品であり、一塊のサービスである。

Web 上に公開された一塊の「Web サービス」であるともいえる。.NET プラットフォームでは、ビルディングブロックと呼ばれる部品を単位として再利用も行われる。

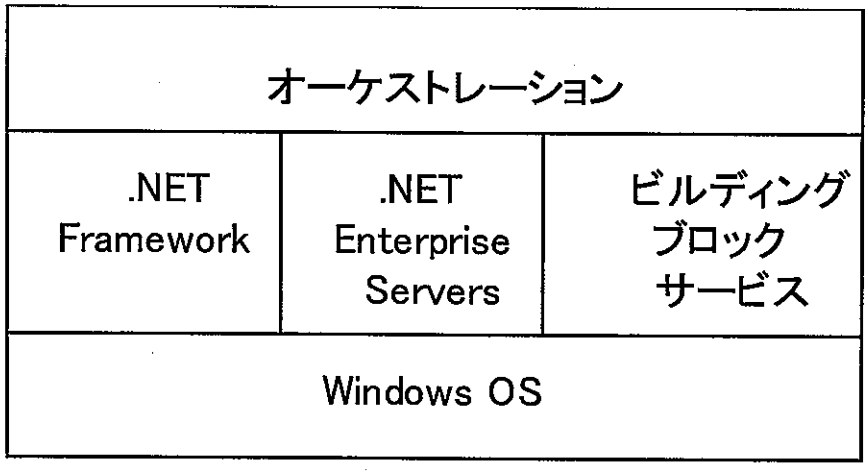


図4.3.4-2 .NET プラットフォームの構成要素

4.3.4.3 関連テクノロジーと実装方法

.NET プラットフォームを構築するための様々な要素

① .NET Framework
共通言語ランタイム、CTS、CLS などの実行環境/仕様
XML Web サービス、Web フォーム、ASP.NET、ADO.NET、Windows フォームなど
VisualStudio.NET、VisualBasic.NET など

② .NET Enterprise Servers
Application Center、Biztalk Server、Commerce Server
Content Management Server、Exchange Server、Host Integration Server
Internet Security and Acceleration Server、Mobile Information Server
SharePoint Portal Server、SQL Server

③ ビルディングブロックサービス
.NET Framework を元に実装したサービス
Win32API など Windows のネイティブな機能を利用したサービス
異種 OS/プラットフォームに基づくサービス

4.3.4.4 .NET Framework

Microsoft .NET Framework は、Web サービスおよびアプリケーションの構築、導入、および実行のためのプラットフォームで、既存のシステムを次世代のアプリケーションおよびサービスと統合するための生産性の高い標準ベースの複数言語環境と、インターネット スケールのアプリケーションの導入と運用に関連する問題を解決するための敏捷性を備えている。.NET Framework は、共通言語ランタイム、階層的な統一されたクラス ライブラリのセット、および Active Server Pages のコンポーネント化されたバージョンである ASP.NET の 3 つの部分から構成されている

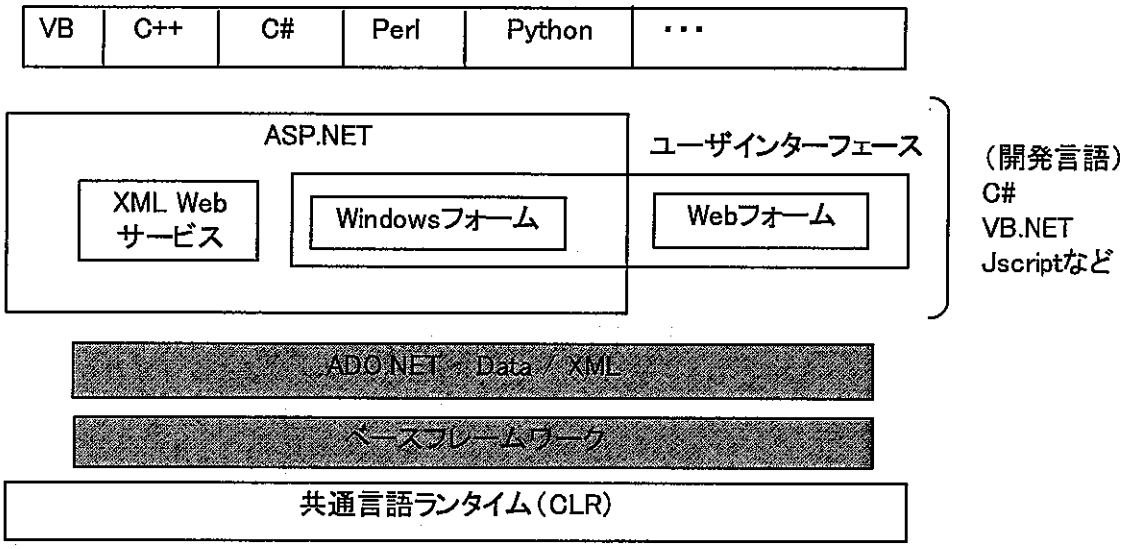


図 4.3.4-3 .NET Framework の構成要素

4.3.4.5 共通言語ランタイム

共通言語ランタイムはオペレーティング システム サービスの上に作られている。これはアプリケーションを実際に実行する責任を負っており、たとえば、アプリケーションのすべての依存関係が満たされていることの確認、メモリの管理、セキュリティの処理、言語の統合といった作業を行う。共通言語ランタイムは、アプリケーションの信頼性を高めると同時に、コードの開発とアプリケーションの導入を単純化するさまざまなサービスを提供している。.NET Framework はメモリ管理を自動化する機能を実装することで、メモリリーク等の問題を解決する事が可能になる。

ただし、開発者は実際に共通言語ランタイムと対話を行うわけではなく、共通言語ランタイムの上に作られた、統一されたクラスのセットを使用する。これらのクラスは任意のプログラミング言語から使用することが可能である。

ここで主な機能を以下に示す。

- 自動メモリ管理
- プロセス管理
- スレッド管理
- セキュリティの実施

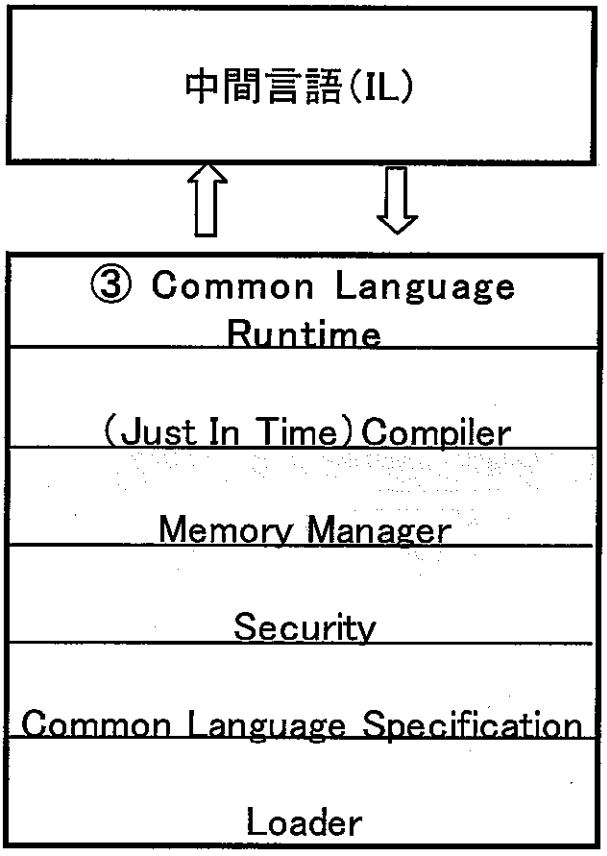


図 4.3.4-4　共通言語ランタイムの構成

4.3.4.6　階層的な統一クラス

.NET Framework のクラスは、図 4.3.4-5 に示すように、さまざまな種類のクラスを統一し、これらの機能のスーパーセットを作成する。その結果、開発者は複数のオブジェクト モデルやクラス ライブラリを学ぶ必要がなくなり、すべてのプログラミング言語に共通する API のセットを作成することで、.NET Framework はクロス言語の継承、エラー処理、およびデバッギングを可能にしている。実際、JScript から C++までのすべてのプログラミング言語が同等になり、開発者は自分の仕事に合った言語を自由に選ぶことが出来る。

.NET Framework 以前の Windows 開発環境では、COM を使ってプログラム間の連携を行っていたが、この技術はバイナリレベルの実装技術であるため、利用には最新の注意が必要であった。例えばデータを受け渡す場合、それぞれの言語でのデータ型の定義が違うため、送受信データの調整が不可欠であった。また、デバッグを行う時は、それぞれの開発ツールを使ってデバッグしなければならなかった。

.NET Framework では、COM を使わない。バイナリレベルの技術ではなくコードレベルの技術を使って複数言語間で連携できるため、開発の難易度は大幅に低減された。いちいち COM 化しなくても、他言語からメソッド等を呼び出すだけでなく、クラスの継承も可能である。データ型が統一されているためデータの受け渡しに気を使う必要も無い。

.NET Framework 上での開発では、言語が異なっても利用するクラスライブラリや IDE

（統合開発環境）は同じである。しかも、言語が異なってもコンパイルすると同じ中間言語「IL（Intermediate Language)」に展開される。つまり、どの言語で開発しても同じ開発方法で、同じ実行性能になる。[RPRG62]

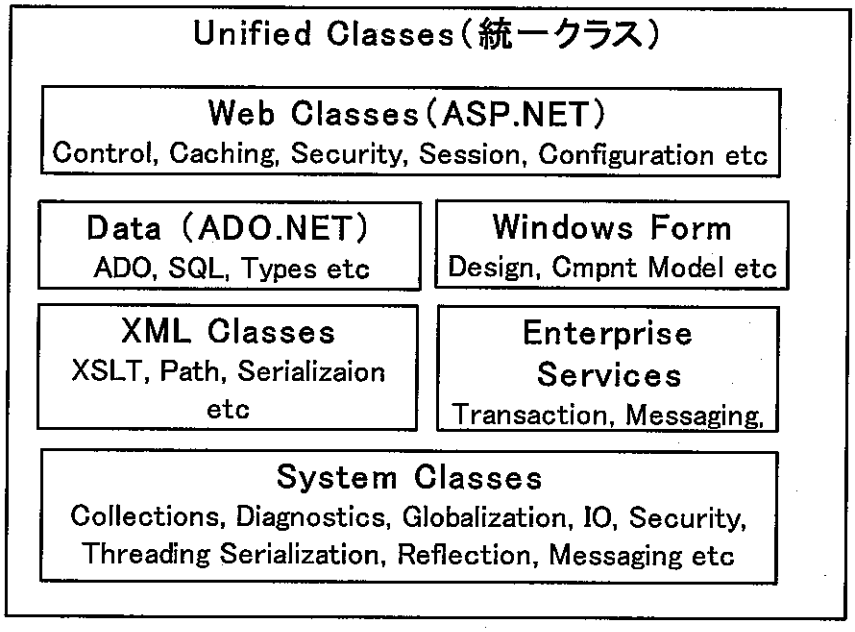


図 4.3.4-5 統一クラスの構成

4.3.5 Javaのネットワーク技術

Javaは先の4.1.4(2)節でも述べたとおり、その特徴はマルチプラットフォーム上で開発・動作することである。このJavaの特徴を生かした、種々のネットワーク技術が存在する。

また、Java という限定された言語での技術範囲ということで、あるので RMI、HORB については具体的な実装を行った(HORB との比較のため、CORBA についても行っている)。それらの実行環境を以下に示す。

実装環境

サーバ側	クライアント側
・OS：Linux(Red Hat 7.1J)	OS：Windows2000 SP2
・Java：JDK1.4.0	Java：JDK1.4.0
・その他：HORB2.0.1	その他：HORB2.0.1

4.3.5.1 RMI (Remote Method Invocation)

JDK 1.1 から標準でサポートされ、Java の開発元である Sun Microsystems が提供している、Java 専用の分散オブジェクト環境である。低レベルのネットワーク部分を意識せずに、簡単にクライアント/サーバ・プログラムを作成することができる。

① RMI の特徴

(a) 変更に対する柔軟性

RMI では、サーバとクライアントは Java のインタフェースで結合されている。これはインタフェースが変更されない限り、サーバの処理に変更があっても、クライアントを変更する必要がなく、また、サーバの変更なしでクライアントを変更できることを意味している。

したがって、RMI では、このようにサーバ、クライアントの変更に対し柔軟なシステムを構築が可能。

(b) 記述の容易性

RMI を用いてサーバ、クライアント・プログラムを記述することが容易（ただ、実行手順は複雑)。

サーバ・プログラムでは、それが RMI のサーバであることを宣言するための数行のコードを記述するだけで良い。

クライアント・プログラムでは、サーバ・プログラム、すなわちサーバ・オブジェクトは、リモートオブジェクトへの参照として得られるが、これが通常のオブジェクトと何ら変わりなく取り扱うことができる。

(c) 可搬性

JDK1.1 以降の環境があれば、他のどのようなシステムにおいても動作させることができる。ただし、JDK1.2 から新しく加わった機能（Activation など）は JDK1.2 が必要。

(d) データ送受信の容易性

RMI ではメソッドを呼び出す際の引数、戻り値として、オブジェクトを受け渡すことができる。そのため複雑なデータ構造でも、ただ１つの引数として取り扱うことが可能。

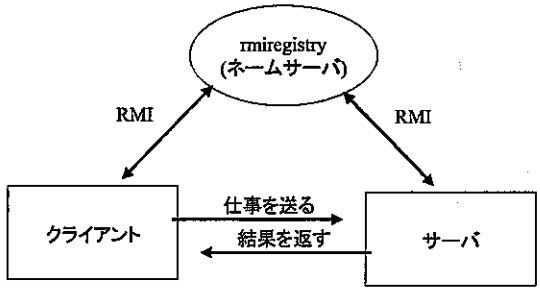


図 4.3.5-1　RMI 構成

JavaRMI を利用する際のツール(いずれも JAVA2(JDK1.2 以降)に標準添付されている)

①rmic　　RMI スタブコンパイラ(RMI 用のインターフェース生成用)

②rmiregistry　　リモートオブジェクト登録(単純な RMI 用ネームサーバ)

③rmid　　JDK1.2 から新たに加わった Activation という機構を実行するためのサーバ(クライアントからの要望に応じて、リモートオブジェクトを呼び出せるようにする)

次に RMI を用いた実装例を示す。

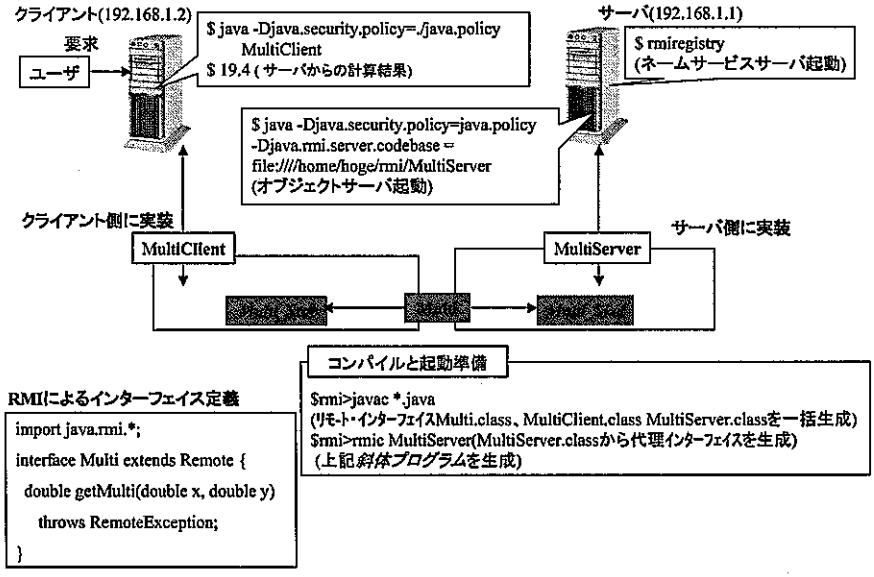


図4.3.5-2　RMIによる実装例

クライアント PC、サーバ PC と 2 台の PC 間において実施した。ネームサーバである rmiregistry はサーバ上にて起動しておこなった。このときのクライアント及びサーバ・プログラムを図 4.3.5-11 に示す。

RMI は CORBA インターフェースと似せながらも、実装における簡略化に成功しているといわれ、また RMI-IIOP (Invocation over Internet Inter-ORB Protocol)という方法で、CORBA 接続も可能としている。

Java での開発を行う場合、RMI は後述する HORB と使いやすさを比較するとやや劣ると思われるが、CORBA よりも手軽で使いやすく、Sun が提供しているだけに Java と共にその発展性が期待されている。

4.3.5.2 HORB

HORB は 1996 年工業技術院、電子技術総合研究所(現：産業技術総合研究所)の平野博士により開発され、世界で始めて実現した Java 用分散オブジェクト技術である。Java を用いて、ネットワーク上の複数のコンピュータをあたかも一つの巨大なマシンに見えるようにするため、広範囲のマシンでポータビリティ portability（移植性）とインタオペラビリティ interoperability （相互運用性）を達成した分散言語システムである。

HORB でプログラムを書けば、市場に出回っているほとんどの OS 上で動く。その特徴は HORB のベースとなった Java から受け継がれたものであり、プログラムは再コンパイルなしに多くのマシン上で動作する。また、HORB のオブジェクト間通信は、

- リモートオブジェクトの代理として Proxy、実装として Skeleton オブジェクトが存在
- ソケットによる複雑なネットワーク通信を隠蔽
- わずかなコードでリモートオブジェクトをあたかもローカルオブジェクトのように扱うことができる

という特徴があり、他の ORB 製品よりも高速であると言われている。

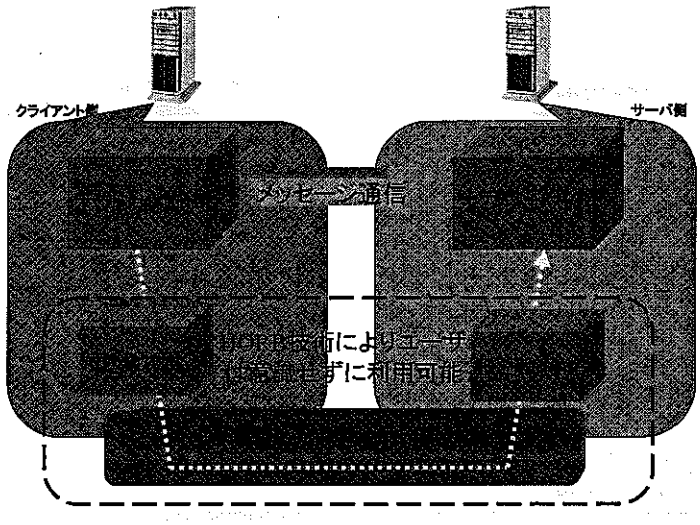


図 4.3.5-2　HORB の動作メカニズム

次に HORB の特徴を CORBA と比較するため、簡単な計算を行う実装例を示す。(CORBA プログラムは図 4.3.5-12 参照)

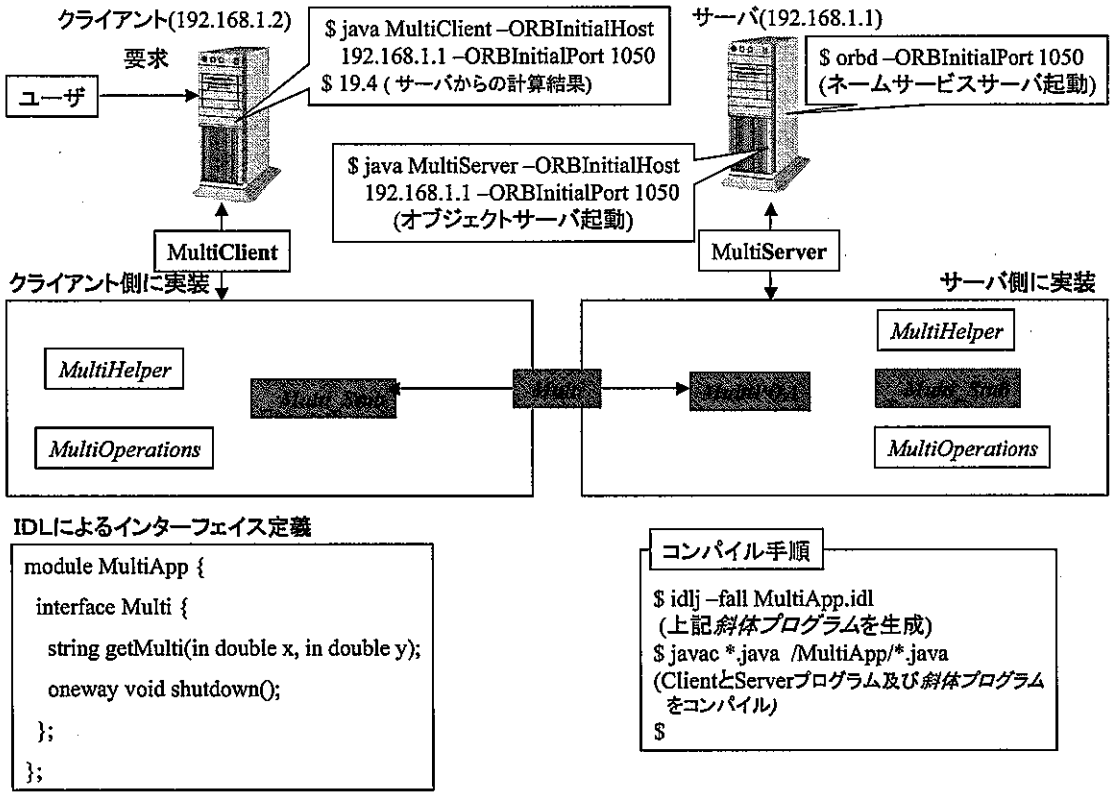


図 4.3.5-3　CORBA による実装例

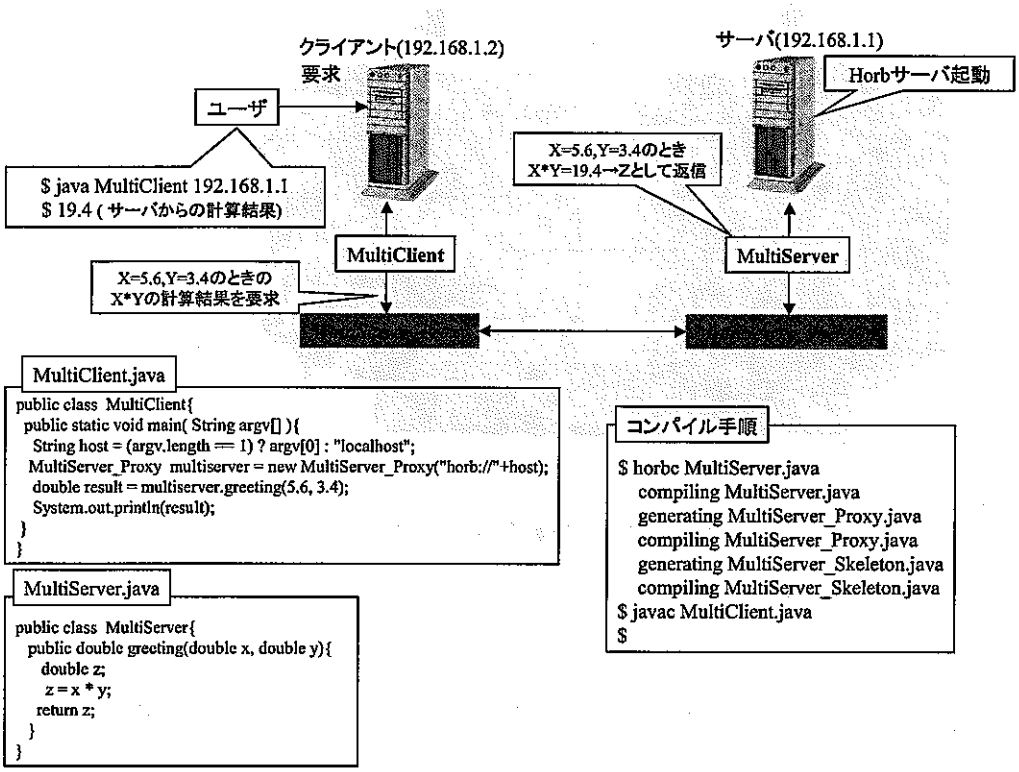


図 4.3.5-4 HORB による実装例

CORBA の実装には訓練と専門知識、プログラミングが必要である。そのため実装にコストがかかり、中規模以下のシステムではあまり使用されない。一般的に大規模な EAI (Enterprise Application Integration) プロジェクトでの利用に限られる。一方、HORB は原則 Java を習得した者であれば、容易に利用できるといわれている(HORB のプログラミングインターフェイスについて、「Java を知っている小学生や中学生でも使える、やさしい分散オブジェクト環境を目指そう」という考えの基で開発されている)。

実際、同様の処理を行うプログラム(HORB は図4.3.5-4又は図4.3.5-13参照)を見ると、双方ともに自動生成機能を有しているが、プログラムを自ら作成する部分に大きな差あることがわかる。

また、現在の計算コードはその多くが並列化されている。並列化を行う際に必要なのが同期・非同期処理である。

CORBA の場合その実装に専門性等が要求され、並列プログラムになるとさらに複雑になるといわれている。その点について HORB はどうか。

基本的なハードウェア構成は先の CORBA と比較した実装例と同一である。

まず同期処理プログラムを上げ、そこから非同期処理プログラムへ移行する例をあげる。

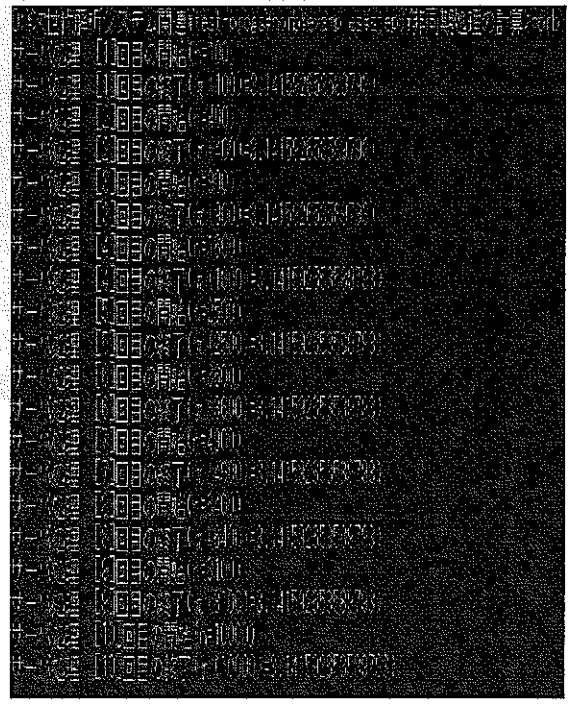
$$\pi(k)=\sum_{k=0}^{\infty}\{4/(8*k+1)-1/(4*k+2)-1/(8*k+5)-1/(8*k+6)\}/(16^k) \qquad (4.3.5\text{-}1)$$

πのBBP公式

ここでの計算は円周率を求めるBBP公式(式1)というものを用いたが、kの値が100を超える程度で、収束してしまうため、全て同じ計算結果となってしまっている。

```
[nagura@ofbra120 oneway]$ java ClientC1
クライアント処理　[1]回目の開始
クライアント処理　[1]回目の終了
クライアント処理　[2]回目の開始
クライアント処理　[2]回目の終了
クライアント処理　[3]回目の開始
クライアント処理　[3]回目の終了
クライアント処理　[4]回目の開始
クライアント処理　[4]回目の終了
クライアント処理　[5]回目の開始
クライアント処理　[5]回目の終了
クライアント処理　[6]回目の開始
クライアント処理　[6]回目の終了
クライアント処理　[7]回目の開始
クライアント処理　[7]回目の終了
クライアント処理　[8]回目の開始
クライアント処理　[8]回目の終了
クライアント処理　[9]回目の開始
クライアント処理　[9]回目の終了
クライアント処理　[10]回目の開始
クライアント処理　[10]回目の終了
[nagura@ofbra120 oneway]$
```

同期時クライアント側出力



同期時サーバ側出力

同期時のクライアント側要求(出力)に準じて、サーバ側出力も順序良く処理を行っている様子がわかる。サーバ側処理が完了してから、クライアント側から次の計算を要求しているためである(プログラムは図4.3.5-3参照)

次に非同期処理プログラムを実行してみる。ちなみに、先のRMIには非同期呼出し機能は提供されていない

```
[nagura@ofbra120 oneway]$ java ClientC1
クライアント処理　[1]回目の開始
クライアント処理　[1]回目の終了
クライアント処理　[2]回目の開始
クライアント処理　[2]回目の終了
クライアント処理　[3]回目の開始
クライアント処理　[3]回目の終了
クライアント処理　[4]回目の開始
クライアント処理　[4]回目の終了
クライアント処理　[5]回目の開始
クライアント処理　[5]回目の終了
クライアント処理　[6]回目の開始
クライアント処理　[6]回目の終了
クライアント処理　[7]回目の開始
クライアント処理　[7]回目の終了
クライアント処理　[8]回目の開始
クライアント処理　[8]回目の終了
クライアント処理　[9]回目の開始
クライアント処理　[9]回目の終了
クライアント処理　[10]回目の開始
クライアント処理　[10]回目の終了
[nagura@ofbra120 oneway]$
```

非同期時クライアント出力

```
サーバ処理　[2]回目の開始(n=400)
サーバ処理　[3]回目の開始(n=900)
サーバ処理　[4]回目の開始(n=1600)
サーバ処理　[5]回目の開始(n=2500)
サーバ処理　[6]回目の開始(n=3600)
サーバ処理　[7]回目の開始(n=4900)
サーバ処理　[8]回目の開始(n=6400)
サーバ処理　[9]回目の開始(n=8100)
サーバ処理　[10]回目の開始(n=10000)
サーバ処理　[1]回目の開始(n=100)
サーバ処理　[2]回目の終了(π(400)=3.1415926535897931)
サーバ処理　[3]回目の終了(π(900)=3.1415926535897931)
サーバ処理　[6]回目の終了(π(3600)=3.1415926535897931)
サーバ処理　[7]回目の終了(π(4900)=3.1415926535897931)
サーバ処理　[1]回目の終了(π(100)=3.1415926535897931)
サーバ処理　[10]回目の終了(π(10000)=3.1415926535897931)
サーバ処理　[9]回目の終了(π(8100)=3.1415926535897931)
サーバ処理　[4]回目の終了(π(1600)=3.1415926535897931)
サーバ処理　[5]回目の終了(π(2500)=3.1415926535897931)
サーバ処理　[8]回目の終了(π(6400)=3.1415926535897931)
```

非同期時サーバ側出力

これらの結果を見るとクライアント側はサーバ側の処理を待たずに、計算要求を行っているため、サーバ側での処理の開始･終了順序が先の同期時と比べばらばらで、クライアント側と非同期に処理されている様子がわかる。

次に表 4.3.5-1 でプログラムの一部に注目して見てみると、

表 4.3.5-1　HORB における同期・非同期処理の記述の違い

	クライアントプログラム	サーバプログラム
同期処理	remote.Calc(i、 (i*I)*100);	public void Calc(int i、 long n)
非同期処理	remote.Calc_OneWay(i 、 (i*i)*100);	public void Calc_OneWay(int i、 long n)

同期処理から非同期処理に移行するための処理は、"OneWay"というこの単語を追加するだけで良い。

これらの結果から見ると、HORB は非常に簡単な手続きだけで、非同期処理を実現することができる。今回の非同期処理プログラムは"クライアント→サーバ"と一方通行(OneWay)の非常に簡単な例であったが、もちろん HORB ではサーバからの結果を受取ったり、コールバック呼出しなどの機能があり、MPI や PVM などで行われている並列処理と同等の処理も可能である。

ただし、HORB の利用には Java 言語を使用する必要があるが、RMI-IIOP を経由した

CORBA(他言語)との接続も可能である。

今回は簡単な実装例を用い CORBA と比較しながら HORB の特徴と有効性を示したが、ここで言えることは、

・CORBA に比べプログラミング負荷が少ない。
・HORB は容易に分散／並列処理を実現できる。

という機能を有しており、Java 言語でのシステム開発には非常に有効な手段であるといえる。

4.3.5.3 J2EE

J2EE とは、Java2 Platform Enterprise Edition の略で、Sun の提唱による企業情報システム向けの Java プラットフォームの規格・仕様であり、1998 年の発表以来に多くのベンダが互換製品を提供している実績ある技術である。

Java には現在、3 種類のプラットフォームが用意されており、基本仕様を提供する J2SE (Java2 Platform Standard Edition) と、携帯電話などの機器組み込み向け仕様の J2ME (Java2 Platform Micro Edition)、そして企業情報システム向け仕様の J2EE である(図 4.3.5-5)。

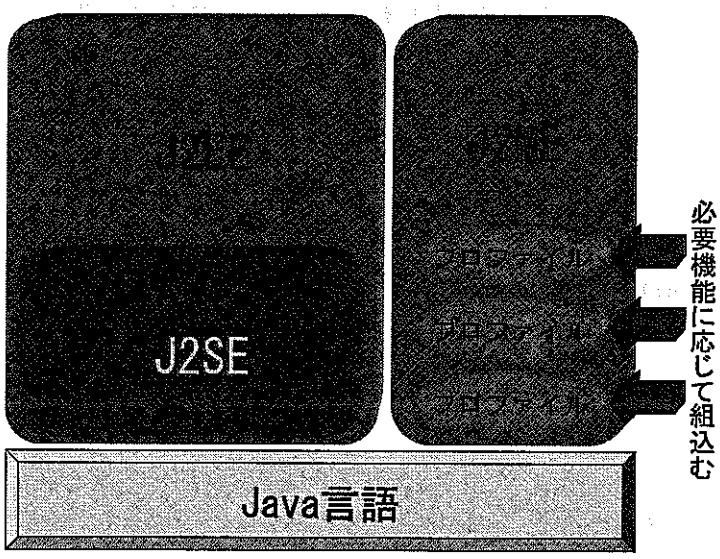


図 4.3.5-5　3 種類のプラットフォームの関係

J2SE は、エンドユーザーが直接操作するデスクトップ・アプリケーションを構築することを主軸としており、AWT や Swing*1 という GUI コンポーネントの充実が図られている。

これに対して、企業情報システムの構築を目的とした J2EE では、仕様の規模もカバーする範囲も膨大なものとなっている。J2SE による基本仕様の上に、Web やデータベースへの対応、ERP*2 などの外部システムとの連携や分散処理が積み重ねられ、幾重にも層をなす構造になっている（図 4.3.5-6）*3。

---

*1：本格的な GUI を Java で実現するために開発されたコンポーネント。Swing は AWT(Abstract Window Toolkit)を継承しとして、1997 年にリリースされ、最近では Swing へ移行するプログラマが増えている。

*2：企業資源計画(Enterprise Resource Planning)の略で、企業全体を経営資源の有効活用の観点から統合的に管理し、経営の効率化を図るための手法・概念。

*3：J2EE は、J2SE のようにクライアント側にアプリケーションを作らず、サーバ・サイドにアプリケーションを構築する技術のため、サーバ・サイド Java ともいう。

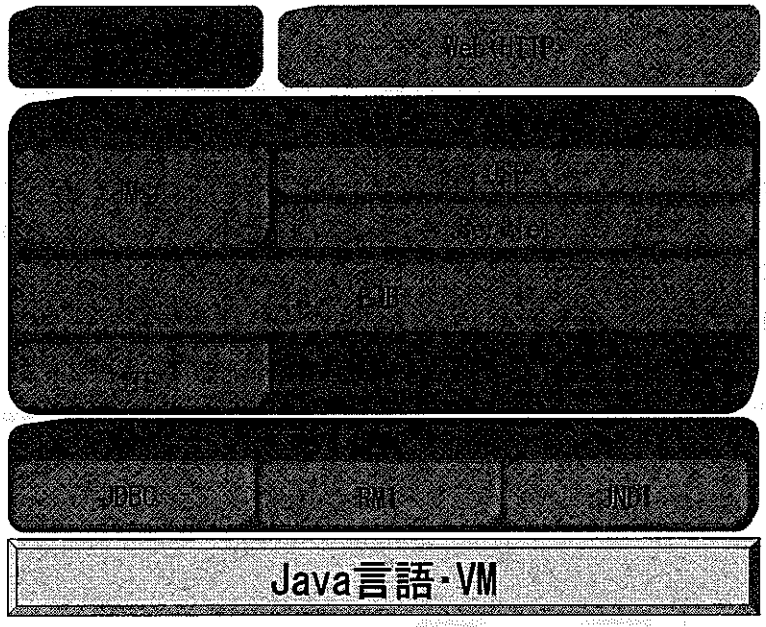


図 4.3.5-6　各プラットフォーム構成

① 従来のクライアント・サーバシステムとの違い

J2EE では、特に Web とデータベース管理システム（DBMS）との連携が重視されている。これもまた、柔軟性や拡張性、費用対効果を目指しているからである。

しかし、従来多く用いられていたクライアント・サーバ構成のシステムでも、DBMS との連携は行われていた（**図 4.3.5-7**）。

では、J2EE はどのあたりが違うのか。

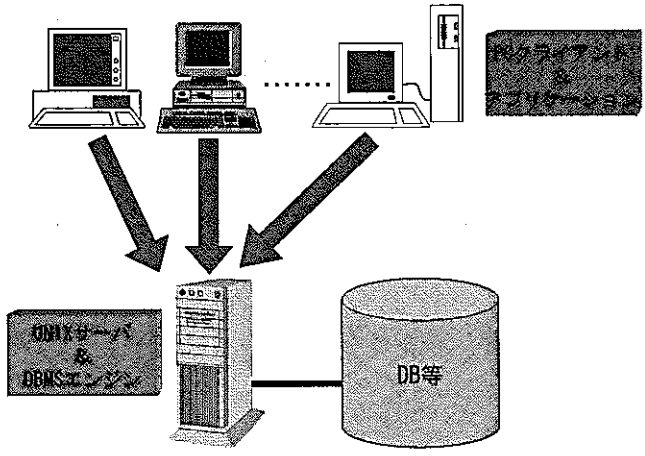


図 4.3.5-7 従来のクライアント・サーバの構成

クライアント・サーバの単純な 2 層構造では、処理が複雑になっていくとどちらかに処理が集中し、問題を引き起こす。

通常のプログラミング言語で作られた、クライアント・アプリケーションに処理が集中すると、複雑で大きなアプリケーションを全てのユーザのマシンにインストールする必要があり、膨大な保守費が発生する。

一方、DBMS サーバに処理を集中させることも問題を引き起こし、どんなに高性能なサーバであっても、様々な処理を 1 台で行うことは不可能である。しかし、DBMS は本来、データの一元管理を行うためのシステムであるため、簡単に台数を増やして処理を分担させることも困難となる。

以上のように、従来の構成には柔軟性や拡張性に問題があった（図 4.3.5-8)。これらのを改善する方法として考えられたのが、3 階層以上の構成による多階層構造である（図 4.3.5-9)。

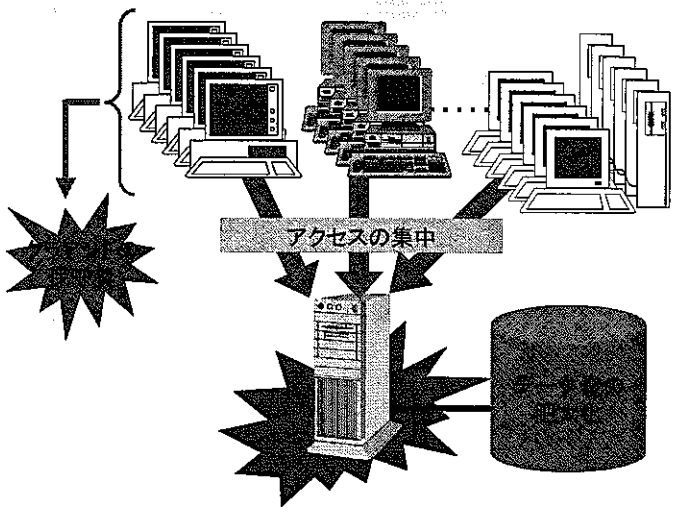


図 4.3.5-8 2 層構造では、クライアントの保守費、拡張性の点で問題

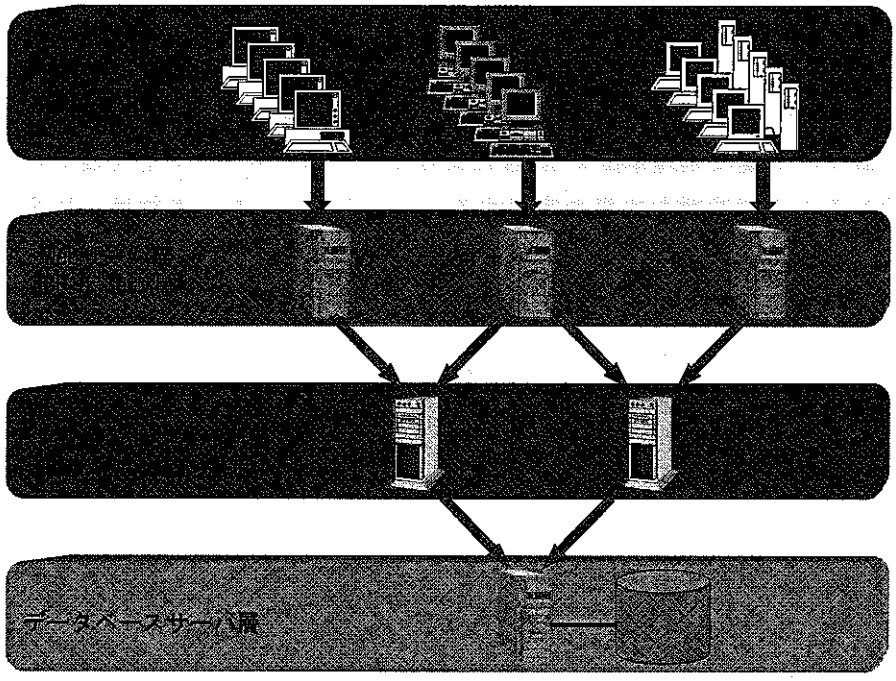


図 4.3.5-10 J2EE は 3 層以上の多階層構造を実現する

J2EE を用いたアプリケーションは、ユーザが直接操作するのは Web ブラウザのみというのが一般的であり、これによって、膨大な数のクライアント・アプリケーションの保守費を削減できる。

そこで、ユーザとのインターフェイスとなるのが Web サーバである。ここではユーザからの要求に応えて Web 画面を用意して提示する必要があり、実際にこれを行うのが Servlet*3 と JSP(Java Server Pages)*4 である。これらは Web サーバ上で動作する Java ベースのコンポーネントである。

---

*3：アプレットが WWW からユーザのマシンにダウンロードされ、ブラウザ側で動作するプログラムであるのに対し、Web サーバ上で直接動作するプログラムである。

Servlet はアクセスされると Web サーバ上で定められた処理を行い、その結果だけをブラウザへと返す。サーバ側で動作するためダウンロードにかかる時間が短縮でき、プログラム自体も高速になる。Java のサーバ・サイド技術の一つとして、昨今注目されている。

*4：サーブレットを動的に生成するための技術であり、HTML ドキュメントの中に専用のタグを使って Java プログラムを埋め込むことができる。マイクロソフトの ASP(Active Sever Pages)と類似の技術と考えることができる。

ただし、Web サーバは 1 台で多くのユーザからのアクセスを処理するため、入力や表示等、低負荷の処理のみにする必要がある。そこで、高負荷の処理を分散して行う中間層のアプリケーション・サーバが必要となる。この上で動作するのが EJB コンポーネントである。

J2EE に対応したアプリケーション・サーバは、複数台を用意して分散処理できるのが特徴であり、個々の単純な処理単位を EJB によってコンポーネントとしてまとめ、これらを自在に組合せることで複雑な処理を実現することが可能となり、柔軟性と拡張性に対応することができる。

```
1  //リモートインターフェイス Multi.java
2  import java.rmi.*;
3  interface Multi extends Remote {
4  double getMulti(double x、 double y) throws RemoteException;
5  }
```

```
1  //クライアントプログラム MultiClient.java
2
3  import java.rmi.*;
4
5  public class MultiClient {
6    public static void main(String args[]) {
7      try {
8        System.setSecurityManager(new RMISecurityManager());
9        Multi obj = (Multi)Naming.lookup
               ("rmi://192.168.1.1/MultiServer");
10       double z = obj.getMulti(5.6、 3.4);
11       System.out.println("z = " + z);
12     }catch(Exception e) {
13       e.printStackTrace();
14     }
15   }
16 }
```

```
 1 //サーバプログラム MultiServer.java
 2
 3 import java.rmi.*;
 4 import java.rmi.server.*;
 5
 6 public class MultiServer extends UnicastRemoteObject implements Multi {
 7   double z = 0.0;
 8   public static void main(String args[]) {
 9     if (System.getSecurityManager() == null) {
10       System.setSecurityManager(new RMISecurityManager());
11     }
12     try {
13       MultiServer obj = new MultiServer();
14       Naming.rebind("rmi://192.168.1.1/MultiServer"、 obj);
15     } catch (Exception e) {
16       e.printStackTrace();
17     }
18   }
19   public MultiServer() throws RemoteException {
20   }
21   public double getMulti(double x、 double y) throws RemoteException {
22         z = z + x * y;
23         return z;
24   }
25 }
```

図 4.3.5-11(RMI 実装例プログラム)

```
1  //クライアントプラグラム MultiClient.java
2  import MultiApp.*;
3  import org.omg.CosNaming.*;
4  import org.omg.CosNaming.NamingContextPackage.*;
5  import org.omg.CORBA.*;
6
7  public class MultiClient {
8    static Multi multiImpl;
9
10   public static void main(String args[]) {
11     try{
12       // ORBの生成と初期化
13       ORB orb = ORB.init(args、 null);
14
15       // ネームサービスを検索してネームサービスの参照を取得する
16       org.omg.CORBA.Object objRef =
17           orb.resolve_initial_references("NameService");
18       NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
19
20       // ネームサービスからMultiオブジェクトwの参照を取得する
21       String name = "Multi";
22       multiImpl = MultiHelper.narrow(ncRef.resolve_str(name));
23
24       // getMulti()メソッドを実行する
25       System.out.println("x1 * y1 =" + multiImpl.getMulti(6.7、 7.1));
26       System.out.println("x2 * y2 =" + multiImpl.getMulti(4.5、 2.3));
27
28     } catch (Exception e) {
29       e.printStackTrace();
30     }
31   }
32 }
```

```
 1 //サーバプログラム MultiServer.java
 2 import MultiApp.*;
 3 import org.omg.CosNaming.*;
 4 import org.omg.CosNaming.NamingContextPackage.*;
 5 import org.omg.CORBA.*;
 6 import org.omg.PortableServer.*;
 7 import org.omg.PortableServer.POA;
 8
 9 import java.util.Properties;
10
11 class MultiImpl extends MultiPOA {
12   private ORB orb;
13
14   public void setORB(ORB orb_val) {
15     orb = orb_val;
16   }
17
18   public String getMulti(double x、 double y) {
19     return (new Double(x * y)).toString();
20   }
21
22   public void shutdown() {
23     orb.shutdown(false);
24   }
25 }
26
27 public class MultiServer {
28
29   public static void main(String args[]) {
30     try{
31       // ORBの生成と初期化を行う
32       ORB orb = ORB.init(args、 null);
33
```

次ページへつづく

前ページからの続き

```
34          // RootPOA の参照取得と POAManager を使用可能にする
35         POArootpoa=POAHelper.narrow(
                        orb.resolve_initial_references("RootPOA"));
36          rootpoa.the_POAManager().activate();
37
38          // サーバントを生成し、それに ORB を登録
39          MultiImpl multiImpl = new MultiImpl();
40          multiImpl.setORB(orb);
41
42          // サーバントから Multi オブジェクトの参照を取得する
43          org.omg.CORBA.Object                    ref                    =
rootpoa.servant_to_reference(multiImpl);
44          Multi href = MultiHelper.narrow(ref);
45
46          // ネームサービスを検索してネームサービスの参照を取得する
47          org.omg.CORBA.Object objRef =
48              orb.resolve_initial_references("NameService");
49          NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
50
51          // Multi オブジェクトの参照をネームサービスに登録する
52          String name = "Multi";
53          NameComponent path[] = ncRef.to_name( name );
54          ncRef.rebind(path、 href);
55
56          System.out.println("CorbaMultiServer が起動した");
57
58          // クライアントからの呼び出し待機
59          orb.run();
60        } catch (Exception e) {
61          e.printStackTrace();
62        }
63
64        System.out.println("CorbaMultiServer を停止します");
65      }
66    }
```

図 4.3.5-12(CORBA 実装例プログラム)

```
 1 //クライアントプログラム ClientC1.java
 2 public class ClientC1{
 3   public static void main( String args[] ){
 4     String hostName = (args.length == 1)? args[0] : "localhost";
 5         int     i;
 6         long    n;
 7
 8     RemoteC1_Proxy remote = new RemoteC1_Proxy("horb://"+hostName);
 9
10         for(i = 1; i <= 10; i++){
11         System.out.println("クライアント処理  ["+i+"]回目の開始");
12
13         remote.Calc(i、 (i*i)*100); //リモート･オブジェクト呼出部分
14
15         System.out.println("クライアント処理  ["+i+"]回目の終了");
16         }
17   }
18 }
```

```
 1 //サーバプログラム RemoteC1
 2 import java.io.*;
 3 public class RemoteC1{
 4   public void  Calc(int i、 long n){
 5         double z = 0.0;
 6         double x = 0.0;
 7         long j;
 8     System.out.println("サーバ処理  ["+i+"]回目の開始(n="+n+")");
 9     for(j = 0; j <= n; j++){
10         x = (double)j;
11         z = z + ((4.0 / (8.0 * x + 1.0) - 1.0 / (4.0 * x + 2.0) - 1.0 / (8.0
               * x + 5.0) - 1.0 / (8.0 * x + 6.0)) / (Math.pow(16.0、 x)));
12         }
13     System.out.println("サーバ処理["+i+"]回目の終了(π("+n+")="+z+")");
14   }
15 }
```

図 4.3.5-13(HORB 実装例：同期処理)

```
 1 //クライアントプログラム ClientC1.java
 2 public class ClientC1{
 3   public static void main( String args[] ){
 4     String hostName = (args.length == 1)? args[0] : "localhost";
 5         int            i;
 6         long    n;
 7
 8     RemoteC1_Proxy remote = new RemoteC1_Proxy("horb://"+hostName);
 9
10         for(i = 1; i <= 10; i++){
11         System.out.println("クライアント処理  ["+i+"]回目の開始");
12
13         remote.Calc_OneWay(i、(i*i)*100);//リモート・オブジェクト呼出部分
14
15         System.out.println("クライアント処理  ["+i+"]回目の終了");
16         }
17   }
18 }
```

```
 1 //サーバプログラム RemoteC1
 2 import java.io.*;
 3 public class RemoteC1{
 4   public void  Calc_OneWay(int i、 long n){
 5         double z = 0.0;
 6         double x = 0.0;
 7         long j;
 8        System.out.println("サーバ処理  ["+i+"]回目の開始(n="+n+")");
 9     for(j = 0; j <= n; j++){
10         x = (double)j;
11         z = z + ((4.0 / (8.0 * x + 1.0) - 1.0 / (4.0 * x + 2.0) - 1.0 / (8.0
                 * x + 5.0) - 1.0 / (8.0 * x + 6.0)) / (Math.pow(16.0、 x)));
12         }
13     System.out.println("サーバ処理["+i+"]回目の終了(π("+n+")="+z+")");
14   }
15 }
```

図 4.3.5-14(HORB 実装例：非同期処理)

4.3.6 エージェント指向技術

エージェント指向技術について、文献 4.3.6-1 を参考に調査した。

エージェント指向技術はオブジェクト指向技術を発展させたものといわれている。エージェント指向技術はまだ研究段階にあり、オブジェクト指向で言う分析・設計・プログラミングのソフトウェア開発工程というものも、必要とするかどうかさえ判然としていないのが現状である。ただ、こうしたエージェントがソフトウェアの分野で注目されているのは、エージェントが以下に挙げるようなオブジェクトにない特徴を有することによる。

・「成長性」

オブジェクトは自身の属性の種類やメソッドを変えないが、エージェントはそれらを自由に変化させて自身を成長させるという「成長性」を有する。

・「自律性」

オブジェクトは受信したメッセージに忠実に行動し、メッセージと周囲の状況が矛盾する場合は結果に到達することをあきらめるのに対し、エージェントは仮にメッセージと周囲の状況が矛盾する場合でも、自らが判断してプランを立て直し、試行錯誤を繰り返しながらあくまで結果にたどり着くことを目指すという「自律性」を有する。

・「移動性」

ネットワーク上を移動する点はオブジェクトもエージェントも同じであるが、オブジェクトは移動それ自体には大きな意味がないのに対し、エージェントは移動した先で何かしらオブジェクトにはできないことを行う。

・「外見」

音声、画像、映像処理などの、いわゆるマルチメディアを駆使して人間の表情等を表現しようとするものである。エージェントにとっては大事な側面の一つであるとされる一方で、人間の表情というものが人間の持つ感受性を研究対象に含む可能性があり、話が高度になりすぎる恐れがあるため、エージェントの持つべき資質としては強くは意識されていないというのが実情のようである。

こうした特徴は、例えば高齢化社会を迎えるにあたって、いわゆる情報弱者といわれる高齢者まで含めたあらゆる世代層が昨今の情報化社会の中で生活を満喫するための有効な道具を提供するといえる。すなわち、コンピュータという道具を、従来は人間がコンピュータに歩み寄り、自らがその使い方を習得する必要があった。それに対し、エージェント指向技術の導入により、コンピュータに関する知識とそれを利用する知識をエージェントが持ち、そのエージェントにユーザの要求を理解させ、必要に応じて作業の肩代わりや的確なアドバイスをさせるといった、いわばコンピュータという道具を使いこなすための道具の提供を可能にするものがエージェント指向技術であるといえよう。

実際にエージェントが活躍しているシステム例としては以下のようなものがある。

① インターフェース・エージェント

エージェントが持つべき資質のうち、「自律性」や「外見」などに重点を置き、人間とのコミュニケーションを重要な目的とするエージェント。すなわち、「情報リテラシー」のうち、「コンピュータを活用する」手助けを行うものである。

一つの応用例として電子メールの処理が考えられる。エージェントは電子メール処理システムに存在し、その内部状態を顔の表情でユーザに示す。エージェントはユーザがメールを処理する手順を絶えず観察、学習し、その効果に基づいてユーザに送られたメールのうちどれをどの順で読むか、どのメールを削除、転送、保存するかなどをユーザに助言する。ユーザはエージェントからの提案をいくつか選択し、エージェントにそれらを実行するように依頼する。この繰り返しでエージェントはメール処理に関して成長し、最終的には本当に重要なメールのみを選択できるようになる。

② モバイル・エージェント

「移動性」に重点を置き、「大量の情報の中から必要な情報を選別して取り出す」ことを重要な目的とするエージェント。すなわち、何とかして結果にたどり着くためにネットワーク上を移動することが特徴である。

既に開発された例としてプランジェント[4.3.6-2]がある。このプランジェントは、

- ネットワークを移動しながら処理を行う「移動性」
- 自分でプランを立てる「自律性」
- あるプランを実行して失敗すると状況に応じて再プランニングする「柔軟性」

を有している。プランジェントは Java で組まれており、Java が起動するワークステーションまたはパソコンをノードとして、これらノード間をネットワークを介して自由に移動する。

上記の「柔軟性」はプランジェントの最大の特徴である。プランジェントは自身の持つ知識ベースとプランを抱えながら、到着したノードが固有に持つ知識ベースと自身の知識ベースを活用してそのノード上でプランを実行する。そこでもし「プラン実行に失敗したら再プランニングする自律性」と「新たに立て直したプランが実行可能かもしれないノードに飛んでいく移動性」とを併せ持つのが大きな特徴である。

プランジェントは電力系統保護制御装置の巡回・点検に既に適用されている。電力所の中央司令室からネットワークに飛び出したプランジェントは最初の保護制御装置に向かい、到着後、あらかじめ決められた検査をその場で行う。まだ成長を遂げていないエージェントは保護制御装置の近くにある知識ベースを適宜参考にし、装置の異常の有無を検査する。検査の結果、異常がなければネットワークを移動して次の保護制御装置に向かい、そこで再び検査を行う。このようにエージェントは順次巡回しながら自信を成長させ、すべての保護制御装置について検査を行い、完了すれば再び中央司令室に戻る。この過程で異常を発見した場合、その異常を示した保護制御装置が有する知識ベースを参考に、エージェン

ト自身のノウハウを活かしながらエージェント自らがその対策を考える。対策が見つかればそれに適した新たなソフトウェアをこの保護制御装置に送り込み、装置のソフトウェアを作業員の移動を伴わずに更新する。周辺の保護制御装置を検査する必要がある場合には分身を生成し、並行して検査を行わせる。分身は自発的に収集した情報を抱えて中央司令室に戻る。

従来は、広域に分散する保護制御装置に対して検査・点検・交換を行う際、作業員が現地に出向いて作業を行っていたが、プランジェントを活用することによりその必要性が基本的になくなり、作業効率が格段に改善されている。

以上、エージェント指向技術についてその特徴を述べた。本研究で目指す次世代解析システムへの適用性の観点からは、オブジェクトをエージェントに置き換えることにより、将来的には解析システムを構成する際、それに必要な部品を結合する作業でユーザに課せられる負荷を大きく軽減することが期待される。まだ研究の途上にある技術ではあるが、今後の研究成果の進捗を見守りつつ、次世代解析システムへの適用を中長期的視野に立って検討を進めるのが妥当と考えられる。

4.3.7 異なる計算機資源での移植性に関する考察

現在のOS環境は大きく分けて、Windows系、Unix系、Macintosh系、Linux(FreeBSD含む)がある。ハードウェアにいたっては数えきれない。

これらの異なった環境において、あるOS環境用に開発されたアプリケーション(システム)を改変すること無く、相互で運用することは不可能である。

通常、多くのアプリケーションは各OS環境に対応した開発環境で、そのプラットフォームの特性を意識して、設計・開発される。

我々が日ごろ使用している計算コードでは、同一機種間であってもOS、コンパイラのバージョン、メーカの違いで、ソースコードレベルでの改変が付きまとい、さらにはその改変後の結果検証まで強いられることになる。これらに係る費用、人的資源投入は膨大なものとなり、様々な計算機が混在するネットワーク上での利用に大きな障壁となる。

4.3.7.1 現状の開発環境

開発環境を左右する最大の要因、それはOSである。ご存知の通りこのOSは我々が計算機を操作(使用)する上で大変便利なものであり、キーボードからの入力や日本語表示等我々が計算機を操作する上で欠くことのできないものである。しかしOSはある特定のプラットフォーム上で理解できるように作られているため、そのOS上で開発されたアプリケーションも自ずと特有のものになってしまう。

現在主流の開発方法はまさしく、開発するアプリケーションの対応プラットフォームに即したもので行われている。身近なところではWordやExcelではWindows系OS上で、計算コードなどではそれが実行されるであろう、ある特定のUnix系OS上で行われている。

これらは、開発ツール等はもとより、将来そのアプリケーションが動作するプラットフォーム上で開発すれば、その開発過程においてより多くのバグや不具合を取り除くことができ、製品化までの期間とコストに対し、優位性を確保できるという極当たり前のことである。

しかし、このような開発環境に縛られた状態では、スタンドアロン計算機やせいぜい同一プラットフォーム同士でネットワークを通じ使用するアプリケーションに留まり、他のプラットフォームでは利用することができ無いものとなる。

4.3.7.2 汎用開発環境

それではどのようにすれば、この異なったプラットフォーム、OSにおける違いを乗り越えられるか。

OSはプラットフォーム固有のものであり、非現実的な「共通OS開発をする」という考えを除けば現時点では動かすことのできない。

それでは共通に利用でき、尚且つプラットフォームやOSの違いを吸収できるものがない

か。

現存する全てのOSに対してというレベルではないが、登場以来、ネットワークコンピューティングの世界で一躍注目を浴び、今では確固たる地位を築いた、オブジェクト指向言語Javaである。

Java は今までにはない、マルチプラットフォームを実現し、Windows、Mac、Unix、Linux等それぞれ異なるプラットフォーム上開発されようが、一度コンパイルすれば、その後はどこへ移植するにも、ソースの改変、再コンパイルなしに利用することができるという言語である。

また、Java と同様の特徴を持った言語「Python」と「Ruby」がある。ともに両者はマルチプラットフォームに対応、オブジェクト指向型言語という点でJava言語と共通であり、唯一異なるのがスクリプト言語という点である（Java、Python、Ruby に関する詳細な説明は4.1.4(2)、(6)及び(7)節を参照)。

4.3.7.3 デメリット(処理速度)

(2)で移植性に優れ、異なる計算機資源を有効活用できる言語について述べた。しかし、これらの言語も決して万能ではなく、その有益な特徴がゆえにいくつかの欠点もある。

まず 1 つ目、「Python」や「Ruby」はスクリプト言語であるため、そのプラットフォーム用の処理系上でインタープリタ(コンピュータが実行できる形式への変換)形式で実行されるため、その処理速度に問題がある(ただし、スクリプト言語系は一般に小規模なプログラムを構築する際に利用されており、計算制御や RAD 的な使用が主である。ここでの比較は参考まで)。

Java はバイトコード(中間言語)を生成し、それを各プラットフォーム用の JavaVM 上で実行する。そのため、半ばインタープリタに近い形で逐一 CPU が理解できるマシン語に変換し実行されるため、同様に処理速度が遅いと言われている(Fortran や C 言語では、CPU が直接理解できる実行形式(ネイティブコード)が生成されるため、高速な処理が可能である。Java は C++より約 3～4 倍遅いと言われている)。

2 つ目は、各プラットフォームの特徴に応じた、アプリケーション開発が行えないため、その特徴を十分に生かすことができない(Intel 社より、Intel 系 CPU 用に最適化された Fortran コンパイラが提供されている)。

4.3.7.4 マルチプラットフォームの可能性

これらの計算処理速度の問題については、最近 Java では JIT コンパイラ(Just in time Compiler)*1 がブラウザに実装され、その性能も日進月歩で改善されており、C 言語並みの計算速度が得られるようになってきている。しかし、これらの言語「マルチ」という特徴を生かせば計算速度を向上させる手法「並列計算」に利用できるのではないか。

主に行われている並列計算は、「MPI」や「PVM」等のライブラリを用いて Fortran や C 言語等で開発が行われている。また、実行環境は通常クラスタ計算機を使用している。このクラスタ計算機は複数の計算機を 1 台の巨大な計算機として扱えるようにしているが、この各計算機のプラットフォームは全て同一構成になっている。

そのため、一定の計算処理能力は得られるものの、それ以上の能力向上は望めない。

ここで先に上げた「マルチ」である。アプリケーション(計算コード)を Java 等の言語で開発した場合、これらはどの計算機上でも動作することから、上記のクラスタ計算機の範囲を超えた、分散コンピューティング、強いてはグリッドコンピューティング*2 としての利用が可能になってくる。もちろん、グリッドコンピューティングを行うには、アプリケーションの並列化手法やロードバランスなどまだ技術的な課題が多いが、今年 2002 年 6 月に産官学を交えた「グリッド協議会」が日本で設立され、グリッド技術の普及や標準化等の活動が計画されている。また、実際に計算機メーカでもその計画を打ち出しているこのような動きの中で、Java をはじめとしたこれらの言語は一役買うことができる技術ではないかと考える。

---

*1:Java プログラムを実行する際に、プラットフォームから独立した形式のプログラム(Java バイトコード)を、実行前にまとめて一気にそのプラットフォームで実行可能なプログラム(ネイティブコード)に変換し、実行する機構のこと。少しずつ変換しながら実行する従来の方式より実行速度は速いが、変換に時間を要するので実行を始めるまでにかかる時間は従来より長くなる。

*2:ネットワークを介して複数のコンピュータを結ぶことで仮想的に高性能コンピュータをつくり、利用者はそこから必要なだけ処理能力や記憶容量を取り出して使うというシステム。複数のコンピュータに並列処理を行わせることで、一台一台の性能は低くとも高速に大量の処理を実行できるようになる。ビジネス利用や学術研究など、多くの可能性が模索され、実現に向けてさまざまな試みが行なわれている。distributed.net や SETI@home など、インターネットを通じて家庭のパソコンの空いている CPU パワーを集め、暗号解読や医療研究などの複雑な処理を行わせる分散コンピューティングプロジェクトは既にいくつかあるが、グリッドコンピューティングはこうした技術を含む、より包括的な概念といえる。グリッド技術の普及や標準化を進めている The Global Grid Forum を始め、学術機関を中心に研究が進められていたが、最近では IBM 社が商用化を目指すプロジェクトを立ち上げている。

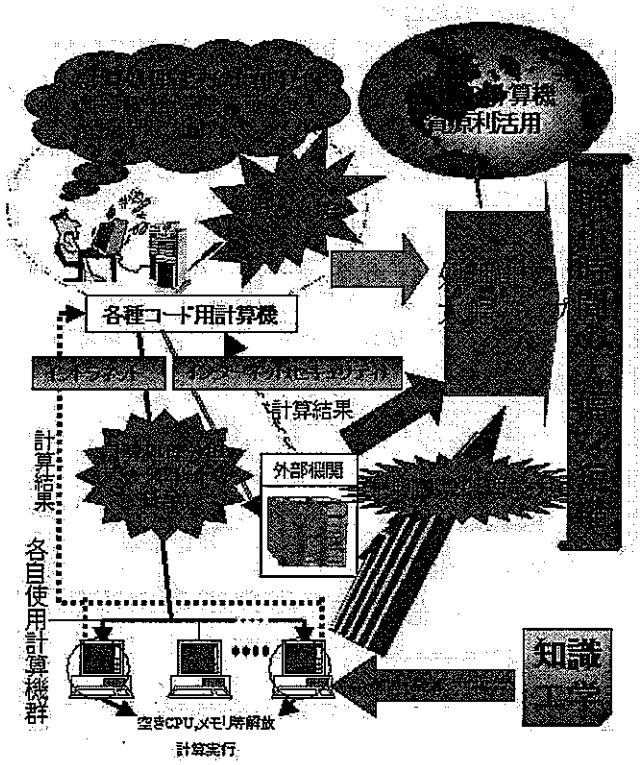


図 4.3.7-1　グリッドコンピューティング構想図

4.4 データベース技術

4.4.1 データベース概要 [4.4.1-1]

ここでは、データベースの概要や主なデータベースの種類について説明する。

4.4.1.1 データベースの概要

データベースとは、大量の情報をまとめて整理し、いつでも必要な時に取り出せる情報の固まりのことを指す。データベースが登場するまではプログラム毎にファイルが存在しており、プログラムとデータは１対１に対応していたことにより、以下の問題点を抱えていた。

① データの冗長性が高い
　同じデータが複数のファイルに存在する
② データの整合性が低い
　データ重複があると、データを更新する際に漏れが起こり、ファイル間のデータに矛盾が発生する
③ データの独立性が低い
　ファイルを修正することによって、プログラムにも影響が出てくる場合もある

データベースは、以上の問題点を解決するために登場した統合的にデータを一元管理する技術と言える。なお、統合化することにより、データを共有でき、多くの利用者が使えるメリットも生まれた。
データベースの特徴としては、以下の点が挙げられる。

① データの共有化・標準化が促進される
② プログラムからの独立性が向上する
③ データの安全性が高まる

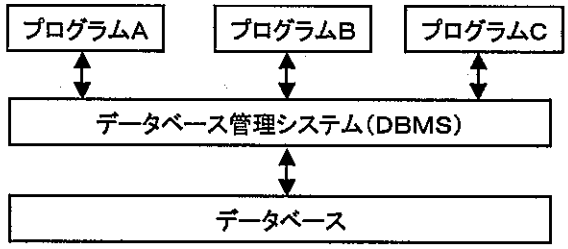


図 4.4.1-1　データベース概念図

4.4.1.2 データベースの種類

データベースは使用するデータ構造によって以下の３種類に分類される。

① ツリー（階層）型データベース
親データは複数の子データをもつことができるが、子データは一つの親データしか持てないという型。

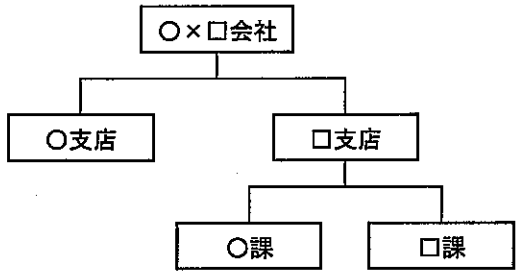


図 4.4.1-2 ツリー型データベース例

② ネットワーク（網）型データベース
ツリー型との違いは、子データにも複数の親データを持つ事ができる。

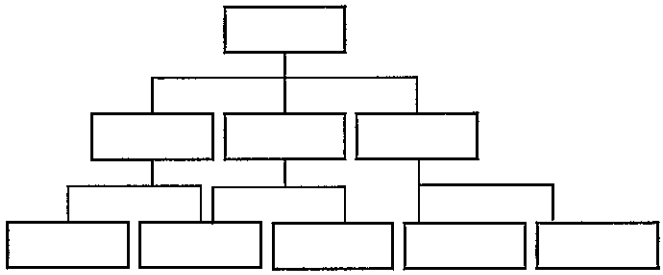


図 4.4.1-3 ネットワーク型データベース例

③ リレーショナル（関係）型データベース
現在の主流なデータベースである。ツリー型やネットワーク型のように、データに親子関係を作ることはせずに表形式でデータを管理する。基本処理としては、選択・射影・結合がある。

- 選択 ・・・ 表の中から条件にあった行を抜き出すこと
- 射影 ・・・ 表の中から条件にあった列を抜き出すこと
- 結合 ・・・ 複数の表をまとめて一つの表にすること

No	氏名	年齢	電話番号
10	伊藤	35	090－＊＊＊
20	田口	40	080－＊＊＊
30	原	19	070－＊＊＊

図 4.4.1-4 リレーショナル型データベース例

4.4.2 オブジェクト指向とデータベース[4.4.2-1、4.4.2-2、4.4.2-3]

ここでは、データベース分野でのオブジェクト指向技術への取り組みとして、データベースにオブジェクト指向技術を取込んだ「オブジェクト指向データベース」とリレーショナル型データベースをベースにオブジェクト指向技術の拡張を行った「オブジェクトリレーショナルデータベース」の概要について説明する。

4.4.2.1 オブジェクト指向データベース概要

オブジェクト指向データベースは、元々オブジェクト指向システム開発および、オブジェクト指向プログラミングの考え方をデータベースに取り入れたものである。リレーショナル型データベースは主に計数データを扱うアプリケーションに有利な構造となっているのに対し、オブジェクト指向データベースは複雑なデータ構造やマルチメディア・データ（文書／イメージ／画像等の多種多様なデータ）を扱いつつ、オブジェクト指向の概念により開発効率の向上も狙ったものである。オブジェクト指向の基本的な考え方には、抽象データ型(abstract data type)、継承(inheritance)およびオブジェクト識別性がある。抽象データ型は、現実世界の実体をデータとそれに対する操作または、手続きとでカプセル化して表現するという考え方である。以下にオブジェクト指向データベースの特徴を記す。

① 複合オブジェクト

オブジェクト指向データベースでは、配列や集合など多様なデータ構造をそのまま格納することが可能である。銀行の融資業務を想定した場合には、取引先／融資額といった文字／数値データに加え、取引先の情報を記録した書類／取引先の経営状況を表したグラフ／担保物件の写真などのマルチメディア・データについても一元的に管理することができる。

一方リレーショナル型データベースでは、データ構造が複雑になるとデータの正規化やテーブル間の複雑なリンク付けが必要となり、データベース設計が困難なばかりか結果的に表形式では表現できない場合もある。オブジェクト指向データベースでは、人間が「共通な性質を持つものを一つのまとまりとして考えると扱いやすい」と思うものをクラス(class)として定義する。

```
class 社員
  type tupple(社員コード:integer、
              氏名: string、
              写真: Bitmap、
              部課 : 部課、
              職位: string、
              クラブ: set(クラブ)、
              給与:list(tupple(基本給:integer、
                               手当:integer、
                               残業:integer、
                               控除:integer、))
  method 月給(base:基本給、grade:職位、add1:残業、add2:手当、minus:控除)
  end;
```

図 4.4.2-1　複合オブジェクト定義例

図 4.4.2-1 の例で、set は、ゴルフクラブ、テニスクラブなど別の所で定義されているオブジェクトの集合を指定している。tupple は、リレーショナル型データベース同様の属性の並びを表している。この例に見られるように、リスト、タプル、集合などの構成子(constructor)を再帰的に使って定義した型を持つオブジェクトを複合オブジェクト(complex object)と呼んでいる。この例から、リレーショナル型データベースでは、全てを表形式に直して表現するので、実世界にある物事をそのままでは表現しにくい問題をオブジェクト指向データベースでは解決しているのが分かる。

② カプセル化

オブジェクトは、データとそれに付随する処理を一体化して構成される。処理はメソッドと呼ばれ、このメソッドを働かすためには、このオブジェクトに対してメッセージを送信し、結果は、メッセージで受け取るようになっている。オブジェクト指向では、このようにデータとメソッドをカプセル化 (encapsulation)することで情報の隠蔽を図っている。このオブジェクトが一旦データベースに格納されると、複数のアプリケーションで共有できるようになり、再利用が可能となるためアプリケーション開発時間が大幅に削減される効果が生まれる。また、カプセルに隠蔽されたデータは、外部参照によって破壊されることがない。

③ 継承

データと処理方式を組み合わせたオブジェクトを個々に準備していくと、その量は膨大になる。そこで同じ性質を持つオブジェクトをクラスとしてまとめ、クラス間に階層関係を

つけることを可能としている。下位のクラスは、上位のクラス属性を継承することにより、意味を持った階層的データ構造を表現できる。これにより、各オブジェクトに共通な性質は一ヶ所に持てば良く、データベース定義の簡略化とデータベース格納領域の削減に繋がる。

更に、一度定義したクラスは他のデータベースシステムを作成する際にも再利用が可能である。継承の様子を図 4.4.2-2 に示す。

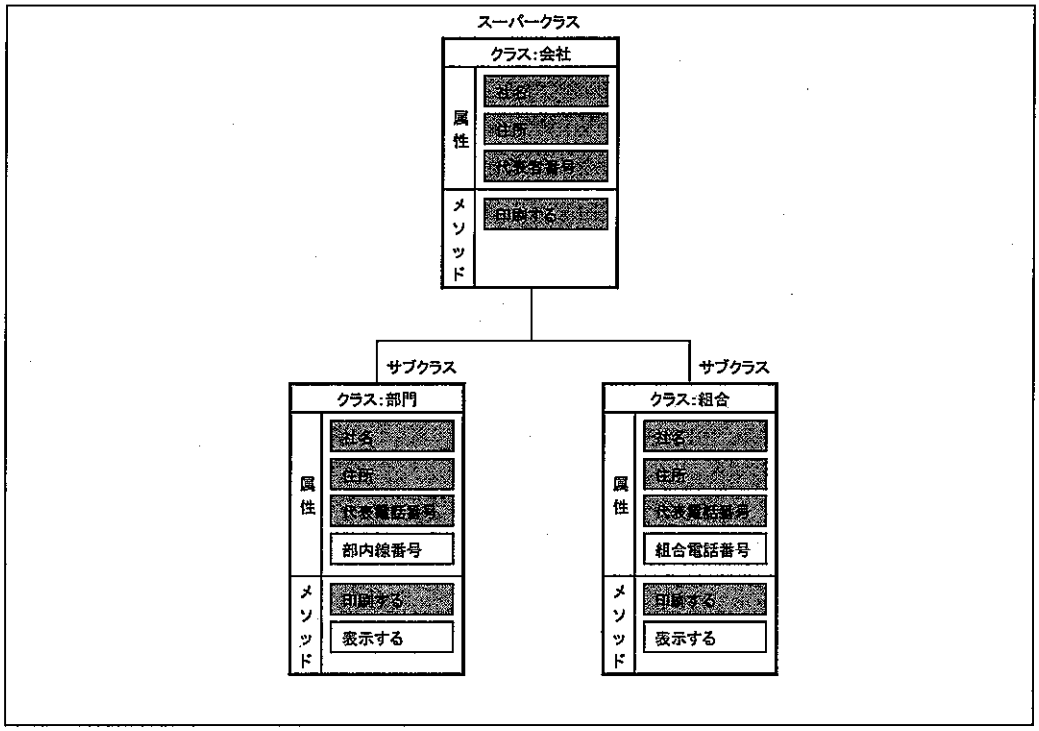


図 4.4.2-2　クラス継承例

④　オブジェクト識別子(object identifier)

オブジェクト指向データベースでは、オブジェクト・インスタンスを一意に識別するためにオブジェクト識別子を使用する。このオブジェクト識別子はインスタンスが作られると自動的にオブジェクト指向データベースが生成するようになっている。　アプリケーションプログラムはオブジェクト識別子を直接参照することはないが、オブジェクト同士の関連づけを行う場合は、相手オブジェクトの識別子をリンクポインタとして保持する。

例えば、ある会社の社員データベースにおいて「システム開発部は社員Ａ、社員Ｂ・・を含む」は、１：ｎの関係を表し「社員Ａ、社員Ｂは、複数のプロジェクトに参加している。」は、ｎ：ｍ の関係となる。　オブジェクト指向データベースでは、「～は、～を保有する」、「～は、～に所属する」などの関係をそのまま記述でき、オブジェクト間に直接リンクが張られる。リレーショナル型データベースでは、部門の表、社員の表、およびプロジェクトメンバー表の 3 つのテーブルが必要となる。　この機能により複雑なデータ構造が表現

できるだけでなく、オブジェクト間が直接リンクされている事で、関連するデータを取り出す処理を高速化できる。

リレーショナル型データベースでは関連データの取り出しをテーブル間のジョインで実現するが、テーブル間に跨がる処理が多い場合は、オブジェクト指向データベースの方がパフォーマンスは良くなる。

4.4.2.2 オブジェクトリレーショナルデータベース概要

オブジェクトリレーショナルデータベースは、リレーショナル型データベースのアクセス言語ＳＱＬをオブジェクト指向の仕様へ拡張したものである。(規格としては、ＳＱＬ－９９となる)

この仕様拡張では、データ型（データタイプ）として、基本的な組込み型（標準で提供されている文字型、数値型等）に加え、ユーザ定義型（組込み型をもとに新たに生成したデータ型）とオブジェクト指向のための構造型（組込み型やユーザ定義型を組み合せて新たに生成したデータ型）が追加されている。

この構造型を用いることにより複合データ型（複合型オブジェクト）やスーパー型／サブ型の型階層（データ型の継承）を構成することができる。

構造型とその型階層の定義例を図 4.4.2-3 に示す。**図 4.4.2-3** の例では、データ型「従業員」の現住所に別なデータ型「住所」を指定することにより、データ型「従業員」はデータ型「住所」の構造を持つ複合データ型として
定義されたことになる。　この定義により、データ型「住所」はデータ型「従業員」として取り扱うことができるようになる。

また、データ型「事務員」に「UNDER 従業員」を指定することにより、データ型「従業員」がスーパー型で、データ型「事務員」がサブ型であるという関係が定義されたことになる。　この定義により、データ型「事務員」は上位のデータ型「従業員」のすべての定義を継承していることになる。

```
CREATE TYPE 住所 AS
        (都道府県名      NCHAR(10)、
         市名            NCHAR(10)、
         町名            NCHAR(10)、
         番地            CHAR(10)、
         郵便番号        CHAR(7));

CREATE TYPE 従業員
        (従業員          NCHAR(10)、
         部門名          NCHAR(10)、
         現住所          住所
         給料            DECIMAL(8、0))
METHOD 給料計算() RETURNS DECIMAL(8、0);

CREATE TYPE 事務員 UNDER 従業員 AS
       (特技            NCHAR(100)):

CREATE METHOD 給料計算 FOR 従業員
       BEGIN
          :
       END;
```

図 4.4.2-3　構造型定義例

4.4.3 データベースの実装例 4.4.3-1、4.4.3-2、4.4.3-3

ここでは、図 4.4.3-1 の構造化されたデータ定義例をオブジェクトリレーショナルデータベース PostgreSQL の継承機能を利用し実装する方法を説明する。

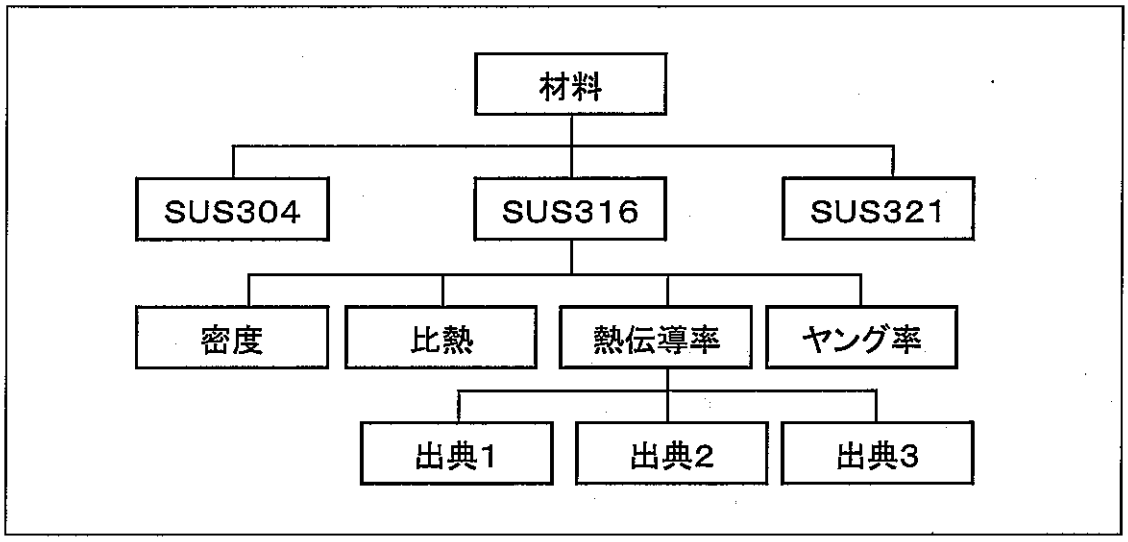


図 4.4.3-1　構造化されたデータ定義例

図 4.4.3-1 のデータ定義例を PostgreSQL の継承機能を利用し、データベース化した例を図 4.4.3-2～3 に示す。

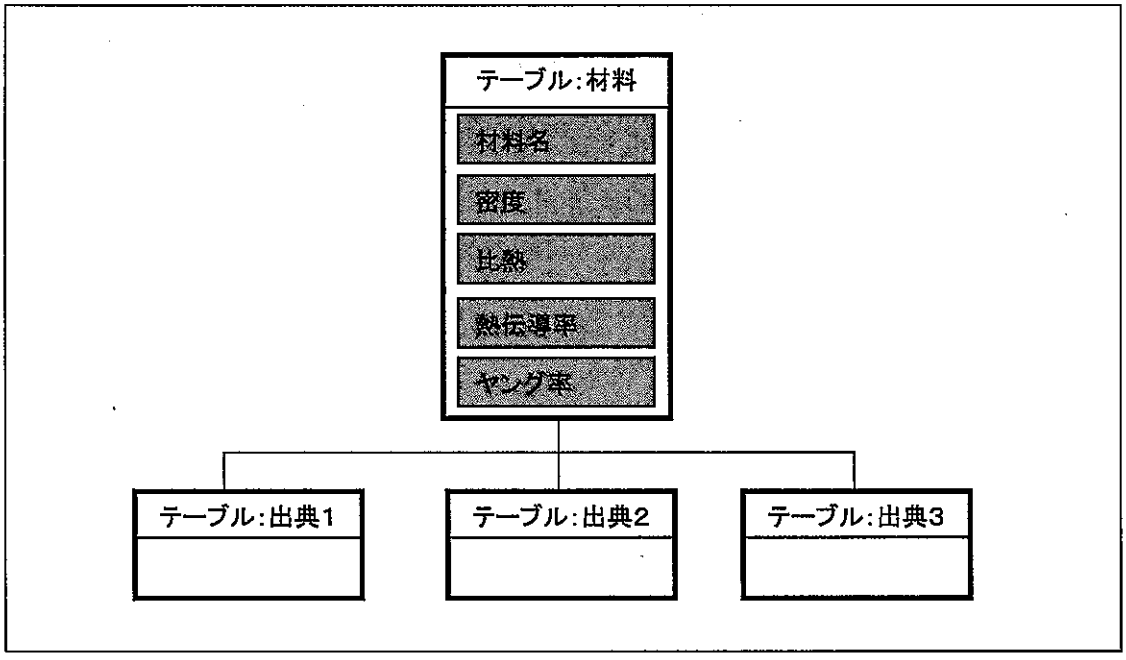


図 4.4.3-2　データベース構成図例

```
create table 材料 (
  材料名 text、
  密度 float、
  比熱 float、
  熱伝導率 float、
  ヤング率 float
);

create table 出典１ ( ) inherits(材料);
create table 出典２ ( ) inherits(材料);
create table 出典３ ( ) inherits(材料);
```

図 4.4.3-3　データベース定義例

図 4.4.3-3 の例では、出典１～３のテーブルに「inherits(材料)」を指定することにより出典１～３のテーブルは、材料テーブルの定義を継承することになる。

次に、作成したデータベースの検索方法について説明する。

PostgreSQL は、データベースアクセス言語ＳＱＬでデータベースへアクセスする方法以外にスクリプト言語 Perl、Tcl/Tk、Python、Ruby 等からもアクセスすることができる。図 4.4.3-4～6 に登録したデータ内容と Python による検索例を示す。

【出典1】

材料名	密度	比熱	熱伝導率	ヤング率
SUS301	8.03	14	0.12	19700
SUS304	8.04	15	0.13	19800
SUS316	8.05	16	0.14	19900
SUS410	7.75	21	0.15	20400

【出典2】

材料名	密度	比熱	熱伝導率	ヤング率
SUS301	8.12	18	0.21	20100
SUS304	8.13	19	0.22	20200
SUS316	8.14	20	0.23	20300
SUS410	8.02	22	0.24	21000

【出典3】

材料名	密度	比熱	熱伝導率	ヤング率
SUS301	8.08	10	0.17	19200
SUS304	8.09	11	0.18	19400
SUS316	8.16	12	0.19	19500
SUS410	7.02	23	0.26	20000

図 4.4.3-4　登録データ例

```
>>> import pg
>>> con = pg.connect('zairyo')
>>> qobj = con.query("select * from 出典1 where 材料名='SUS304'")
>>> result = qobj.getresult()
>>> print result
[('SUS304'、 8.04、 15.0、 0.13、 19800.0)]
```

図 4.4.3-5　検索例 1

図 4.4.3-5 の例では、出典 1 テーブルを材料名 SUS304 で検索すると、他のテーブルの情報は検索されずに出典 1 テーブルの情報のみが検索される。

```
>>> qobj = con.query("select * from 材料 where 材料名='SUS304'")
>>> result = qobj.getresult()
>>> print result
[('SUS304'、8.04、15.0、0.13、19800.0)、('SUS304'、8.13、19.0、0.22、20200.0)、
(
'SUS304'、 8.09、 11.0、 0.18、 19400.0)]
>>> con.close()
```

図 4.4.3-6　検索例 2

また、図 4.4.3-6 の例では、材料テーブルを材料名 SUS304 で検索すると、出典 1 ～ 3 のテーブルの情報が検索される。

第5章　結言

現在の高速炉開発に関連する熱過渡応力解析、炉心過渡解析、炉心核特性解析の分野における解析コードに対するニーズと課題について検討し、各分野共通のニーズと課題を摘出した。この結果を踏まえた上で、次世代解析システムに求められるニーズと課題について、（A）次世代解析コードに求められるニーズ、（B）ニーズを満たすための機能上の課題、（C）機能を満たすための実装上の課題、の三段階に分割して検討を行った結果、以下のようなニーズと課題があることが明らかとなった。

(A)　次世代解析システムに求められるニーズ

1. 自作コードの組み込み
2. 参照解としての利用
3. 異なる専門分野間での統合解析

(B)　ニーズと満たすための機能上の課題

1. 物理的意味を伴った中小規模モジュールの集合体としてのシステム
2. システムの信頼性確保
3. モジュラー型コードシステム

(C)　機能を満たすための実装上の課題

1. 異なるプログラミング言語間での結合
2. 自律したモジュール（オブジェクト指向技術の適用）
3. ネットワーク対応、携帯性
4. オープンシステムによる共同開発
5. モジュールの結合関係の記述

更に、（C）の課題を解決するための要素技術としての最新の計算機関連技術の調査を行った。この結果、（C）の課題を解決する方策として、Microsoft .NET フレームワーク、Java 関連技術、Python や Ruby 等のオブジェクト指向スクリプト言語関連技術等、いくつかの実装上の選択があることが分かった。また、（B）の機能上の課題を解決するためには、オブジェクト指向技術の導入が不可欠であると考えられるが、オブジェクト指向分析・設計・開発をサポートするための UML（Unified Modeling Language）を中核とする開発支援環境なども近年急速に発達している。UML により記述したモデル図から、自動的にオブジェクト指向言語に変換するツールなども存在しており、このような機能を使用することで、異なる分野の研究者たちで、共通のモデル化を共有することも可能になると考えられる。

本調査研究の結果を踏まえ、今後は、工学系モデリング言語としての次世代原子炉系解

析システムの開発を現実のものとするために、従来から使われてきた Fortran ベースの開発環境だけではなく、今回の調査で候補として挙がった、いくつかの開発環境を利用し、次世代解析システムのプロトタイププログラムの作成に着手する予定である。このプロトタイププログラムの作成を通して、より具体的な課題を明らかにすることが可能であると考えられる。

参考文献

2.1-1 Naoto Kasahara and Masaaki Inoue, "Object Oriented Design Procedure for Nuclear Components Against Thermal Transient Stress," ASME, PVP-Vol.360, Pressure Vessel and Piping Codes and Standards(1998)

2.1-2 William Y. Fowlkes and Clyde M. Creveling, "Engineering Methods for Robust Product Design: Using Taguchi Methods in Technology and Product Development," Prentice Hall PTR (1995).

2.1-3 笠原直人、井上正明、"オブジェクト指向過渡熱応力リアルタイムシミュレーションコードPARTS (1) プロトタイプの設計"、機械学会、第8回計算力学講演会講演論文集、516、(1995)

2.1-4 井上正明、笠原直人、"オブジェクト指向過渡熱応力リアルタイムシミュレーションコード PARTS の開発 (2) 熱流動計算サブシステムの開発"、 機械学会、第9回熱工学シンポジウム講演論文集 (1996)

2.1-5 笠原直人、井上正明、"オブジェクト指向過渡熱応力リアルタイムシミュレーションコード PARTS の開発 (3) 構造計算サブシステムの開発"、機械学会、第9回熱工学シンポジウム講演論文集 (1996)

2.1-6 Steeevens, G.L. and Ranganath, S), "Use on on-line fatigue monitoring of nuclear reactor components as atool for plant life extension," PVP Vol.171, ASME (1989)

2.1-7 吉村 忍、"材料・構造問題におけるニューラルネットアプローチ"、機械学会材力講演会論文集、500 (1993)

2.1-8 笠原直人、吉川信治、"ニューラルネットワークと Green 関数法による過渡熱応力高速計算法"、機械学会、計算力学講演会講演論文集、114 (1996)

2.1-9 井上正明、平山 浩、木村公隆、神保雅一、"系統熱過渡荷重への影響因子の調査"、JNC TJ9440 99-017 (1999)

2.1-10 井上正明、 神保雅一、小沢正則、"熱流動-構造統合解析コードの系統熱過渡解析への適用性調査"、JNC TJ1420 2001-001 (1999)

2.1-11 神保雅一、井上正明、小川正則、笠原直人、"統計的手法を用いた熱流動ー構造統合設計手法の開発"、 原子力学会、 秋の大会、J28 (2001)

2.1-12 岩崎 隆、仲井 悟、島川佳郎、丹治幹雄、"モジュール型プラント動特性コードSuper-COPD コード説明書"、 PNC TN9520 89-001、(1989)

2.2-1 宇都、月森、根岸、江沼、菅谷、堺、"ヴァーチャル炉心ラボラトリーの開発に関する研究(1) ー 核 - 熱流動 - 構造結合解析プロトタイプシステムの開発 ー"、JNC TN1400 99-018 (1999年9月)

2.2-2 N. Uto, K. Tsukimori, H. Negishi, Y. Enuma, T. Sugaya, and T.

Sakai, "Development of A Computational System for Coupled Neutronic, Thermal-Hydraulic and Structural Analysis Using Message Passing Interface," Proc. of The Fourth Intl. Conf. on Supercomputing in Nuclear Applications (SNA2000), Tokyo, Sep. 4-7 (2001)

2.2-3 T. B. Fowler, et al., "Nuclear Reactor Core Analysis Code : CITATION," ORNL-TM 2946, Rev. 2 (1971)

2.2-4 飯島 進他、"高速炉設計計算用プログラム・2（2 次元・3 次元拡散摂動理論計算コード：PERKY)"、JAERI-M-6993（1977 年 2 月）

2.2-5 J. G. Guppy, "Super System Code [SSC Rev. 2] An Advanced Thermohydraulic Simulation Code for Transients in LMFBRs," BNL-NUREG-51650, April (1983)

2.2-6 H. Negishi, K. Tsukimori, K. Matsubara and T. Sato, "Development of Core Deformation Analysis Program by Using Parallel Algorithm," HPCN Europe '98 Proc., pp999-1001 (1998)

2.2-7 William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum, "A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard," Parallel Computing, 22 : pp789-828 (1996)

2.3-1 共用炉物理コードシステム専門委員会："共用炉物理コードシステム特別専門委員会報告書"、JNC TJ9420 2001-007（2001 年 3 月）

2.3-2 J.Y.Doriath, et al., "ERANOS1: The Advanced European System of Codes for Reactor Physics Calculations," Proc. Int. Conf. on Mathematical Methods and Supercomputing in Nuclear Applications (M&C + SNA'93), 19-23 Apr. 1993 Karlsruhe, Germany, pp.177 (1993)

3.1.1-1 Erich Gamma, Richard Helm, "Design Pattern Elements of Reusable Object-Oriented Software", SOFT Bank Publishing, (September 2001)

3.1.2-1 EJB コンポーネントに関するコンソーシアム、"コンポーネント仕様／品質公開情報規約 第 1 版"、EJB コンポーネントに関するコンソーシアム、(July 2001)

3.1.2-2 EJB コンポーネントに関するコンソーシアム、"ポータブルコンポーネント規約第 2 版"、EJB コンポーネントに関するコンソーシアム、(September 2001)

3.1.3-1 J.K.Ousterhout, "Scripting: Higher Level Programming for the 21st Century", IEEE Computer Magazine, (1998)

3.2.3-1 S.Yoshimura, "Annual Report of ADVENTURE Project", Computational

Science and Engineering Research for the Future Program Japan Society for the Promotion of Science, (September 1999)

3.3.1-1 "VisualBasic でグラフィッカルプログラミング"、 アストロデザイン (株)、(2001)

3.4.2-1 三木、"超並列計算研究会 PC クラスタ超入門 2000"、 超並列計算研究会、September 2000、http://is.doshisha.ac.jp/SMPP/

4.1.1-1 A.Aoki : "オブジェクト指向システム分析設計入門"、コンピュータリサーチセンター、(1999)

4.1.1-2 "Garbage Collection Part Ⅰ & Ⅱ: Automatic Memory Management in the Microsoft .NET Framework", Microsoft Corporation, (November 2000)

4.1.2.1-1 A.Aoki、"オブジェクト指向システム分析設計入門"、 コンピュータリサーチセンター、(1999)

4.1.2.1-2 "Microsoft VisualStudio.NET Beta2 Reviewer's Guide", Microsoft Corporation, (2001)

4.1.2.2-1 M.Ishii、 "デザインパターンと Open-Closed Principle"、 (June 1999) http://member.nifty.ne.jp/masarl/article/dp-ocp-2.html

4.1.2.3-1 "Nikkei IT Professionals 2002.6", Nikkei Buisiness Publicatuons Inc、(June 2002)

4.1.2.3-2 奥村、 "オブジェクト指向分析設計手法による Web アプリケーション設計"、(株)豆蔵、 (2001)、http://www.mamezou.com/

4.1.2.3-3 羽生田、"いまなぜ開発プロセスを注目するのか"、(株)豆蔵、(January 2002)、http://www.atmarkit.co.jp/fjava/devs/process01/process01.html

4.1.3.1-1 松山、 "NIKKEI OPEN SYSTEM 2001 年 12 月 (No.105) 調査&検証 VisualStudio.NET"、 日経 BP、 (September 2001)

4.1.3.1-2 Jython、http://www.jython.org/

4.1.3.1-3 オージス総研 おれは Jython 第 1 回 http://www.ogis-ri.co.jp/otc/hiroba/technical/jython/jython1/html4/

4.1.3.1-4 Jython=Java+Python 複合言語プログラミング http://isweb23.infoseek.co.jp/computer/paison/paison/

4.1.3.1-5 JDBC Data Access API、http://java.sun.com/j2ee/ja/jdbc/index.html

4.1.3.1-6 Art of Java Programming、http://www.bekkoame.ne.jp/~mizutori/java/index.html

4.1.3.1-7 David M. Beazley: "SWIG Reference Manual Version 1.1", Department of Computer Science University of Utah Salt Lake City, Utah 84112, beazley@cs.utah.edu (June 1997)

4.1.3.1-8 David M. Beazley: "SWIG User Manual Version 1.1", Department of Computer Science University of Utah Salt Lake City, Utah 84112, beazley@cs.utah.edu (June 1997)

4.1.3.1-9 増井俊之：「インターフェイスの街角(27)—SWIG」、UNIX MAGAZINE、2000年3月号

4.1.4.2-1 "Smalltalk 入門"、Object Technology Group、(June 1995) http://www.njk.co.jp/otg/

4.1.4.2-2 「The Java 3D API 仕様」Henry Sowizral、 Kevin Rushforth、 Michael Deering 著　竹内里佳訳 ㈱アスキー

4.1.4.2-3 http://www.antun.net/index.html

4.1.4.2-4 http://java.sun.com/products/java-media/3D/

4.1.4.2-5 http://www.blackdown.org/java-linux/jdk1.2-status/java-3d-status.html

4.1.4.2-6 http://www.javaopen.org/j3dbook/text/welcome.html

4.1.4.2-7 http://www.javaopen.org/jfriends/index.html

4.1.4.3-1 柏原正三：「標準プログラマーズライブラリ　C++効率的最速学習徹底入門」、技術評論社（2002年8月）

4.1.4.3-2 "Microsoft VisualStudio.NET で学ぶ.NET プログラミング(Visual C++.NET 編）"、マイクロソフト株式会社、(May 2002)

4.1.4.4-1 "Microsoft VisualStudio.NET で学ぶ.NET プログラミング（Visual C#.NET 編）"、マイクロソフト株式会社、(May 2002)

4.1.4.5-1 "Microsoft VisualStudio.NET で学ぶ.NET プログラミング（VisualBasic.NET 編）"、マイクロソフト株式会社、(May 2002)

4.1.4.6-1 Mark Lutz、David Ascher 著、紀太 章訳：「初めての Python」、オライリー・ジャパン、ISBN4-87311-022-X (2000年9月)

4.1.4.6-2 Guido van Rossum, Fred L. Drake, Jr.: ``Python Reference Manual", Python Documentation, Online Document, http://www.python.org/

4.1.4.6-3 David Ascher, Paul F. Dubois, Konrad Hinsen, Jim Hugunin, TravisOliphant: ``Numerical Python", Lawrence Livermore National Laboratory, Livermore, CA 94566 UCRL-MA-128569 (Dec. 2000)

4.1.4.6-4 Konrad Hinsen: ``Scientific Python User's Guide", Centre de Biophysique Mol eculaire CNRS Rue Charles Sadron 45071 Orl eans Cedex 2 France (March 2001)

4.1.4.6-5 Fredrik Lundth: ``An Introduction to Tkinter", Online Document, http://www.pythonware.com/library/tkinter/introduction/index.htm

4.1.3.6-6 Pearu Peterson: ``f2py Fortran to Python Interface Generator Second Edition"Revision：1:16 (May 9, 2001)

4.1.3.6-7 Paul F. Dubois: ``Pyfort Reference Manual Version 6", Program for Climate Model Diagnosis and Intercomparison Lawrence Livermore National Laboratory Livermore, CA (August 26, 2000)

4.1.4.7-1 まつもとゆきひろ、石塚圭樹：「オブジェクト指向スクリプト言語 Ruby」、ASCII SOFTWARE SCIENCE Language〈11> ISBN:4756132545（1999年

11月)
4.1.4.7-2 まつもとゆきひろ:「オブジェクト指向スクリプト言語 Ruby リファレンスマニュアル」、http://www.ruby-lang.org/
4.1.4.8-1 V. K. Decyk, C. D. Norton and B. K. Szymanski,"How to Express C++ Concepts in Fortran 90
4.1.4.8-2 http://www.lctn.u-nancy.fr/design/hcomp.html#Heading1
4.2.2-1 プロフェッショナル Java 下、JDBC、XML、分散オブジェクト、セキュリティ、チューニング編、Brett Spell 著 アクロバイト 監訳
4.2.2-2 じゃばじゃば
http://www.asahi-net.or.jp/~DP8T-ASM/java/tips/Serializable.html
4.2.4.1-1 吉村、中林、"設計用大規模並列有限要素法解析システム ADVENTURE システム講習会資料集"、日本学術振興会 未来開拓学術研究推進事業「計算科学」分野 ADVENTURE プロジェクト、(May 2001)
4.2.4.2-1 柄谷、"GeoFEM のプラグイン / 並列有限要素法プログラミング No.01-51"、機械学会講習会、(September 2001)
4.2.4.3-1 林洋介、他:「地球惑星流体現象を念頭においた多次元数値データの構造化」、平成10年度計算科学技術活用型特定研究開発推進事業(短期集中型)研究開発終了報告書(2000年4月)
http://dennou-h.gfd-dennou.org/library/davis/ workshop/2000-03-09/
4.2.4.3-2 地球流体電脳倶楽部、http://dennou-h.gfd-dennou.org/
4.2.4.4-1 藤井義巳、他:科学・工学のための計算機支援問題解決環境 CAPSE の構築
http://www.ipa.go.jp/NBP/Digital/comp/pdf/003.pdf
4.2.4.4-2 平成10年度 Activity Report 金沢大学理学部 計算科学科 計算機実験学講座、1999年10月発行
4.2.4.4-3 田子精男:問題解決環境
http://cmpsci.s.kanazawa-u.ac.jp/~actrep/98/es.tago.html
4.2.4.5-1 科学技術動向 2001年12月号 文部科学省 科学技術政策研究所 科学技術動向研究センター 特集 バイオインフォマティクスの動向
http://www.nistep.go.jp/achiev/ftx/jpn/stfc/stt009j/feature1.html
4.2.4.5-2 データベース学研究室
http://db-www.aist-nara.ac.jp/ResearchThemes/genome.html
4.2.4.5-3 NetLaboratory、http://venus.netlaboratory.com/
4.2.4.6-1 "第2回 企業の情報アーキテクチャの中核的存在 EAI とは"、EAI Solution Firm、 2002、http://www.eaisf.net/main.html
4.2.3-1 http://www-fact.jst.go.jp/~vis/report/node3.html
4.2.3-2 http://robin2.ise.chuo-u.ac.jp/TISE/kyouyou/8makino1999120/
4.2.3-3 工業技術院地質調査所監修、日本列島の地質編集委員会編、「理科年表読本 コンピュータグラフィックス 日本列島の地質」、ISBN4-621-04272-6 (1996

年 12 月).
4.2.3-4 日刊工業新聞、「高速増殖炉実現へ/研究開発最前線⑫ 精密検査-X 線 CT 技術 - 燃料壊さず外から観察-」(平成 14 年 3 月 12 日付)
4.2.3-5 http://w3.jnc.go.jp/~jsystem/etc_sec/vr/gallery/movies/1F_mov.html
4.3.4-1 "Microsoft VisualStudio.NET で学ぶ.NET プログラミング (Visual C#.NET 編)"、マイクロソフト株式会社、(May 2002)
4.3.5.1-1 http://java.sun.com/j2se/1.4/ja/docs/ja/guide/rmi-iiop/index.html
4.3.5.1-2 http://www.asahi-net.or.jp/~dp8t-asm/java/articles/JavaDApp/article.html
4.3.5.1-3 http://www.logos.co.jp/development/
4.3.5.1-4 http://home.catv.ne.jp/dd/chiba/ken/Java/JavaMain.html
4.3.5.1-5 http://www.njk.co.jp/otg/Study/Javado-wake/index.html
4.3.5.1-6 「Java 分散オブジェクト入門」中山 茂 著 技報堂出版
4.3.5.2-1 HORB 2.0 Flyer's Guide、
http://horb.a02.aist.go.jp/horb-j/doc/guide/guide.htm
4.3.5.2-2 http://www.logos.co.jp/development/
4.3.5.2-3 http://www.atmarkit.co.jp/fjava/
4.3.5.2-4 http://www.njk.co.jp/otg/Ijaho/presen/
4.3.5.2-5 「Java 分散オブジェクト入門」中山 茂 著 技報堂出版
4.3.5.2-6 http://java.sun.com/j2ee/ja/index.html
4.3.5.2-7 http://www.atmarkit.co.jp/fjava/rensai2/j2eemap01/j2eemap01.html
4.3.6-1 本位田真一、大須賀明彦、「オブジェクト指向からエージェント指向へ」、ソフトバンクパブリッシング株式会社 (2001 年).
4.3.6-2 http://www2.toshiba.co.jp/plangent/index_j.htm
4.3.7-1 「Java グラフィックス完全制覇」 芹沢 浩著 技術評論社
4.3.7-2 「Java による流体・熱流動数値シミュレーション」峯村 吉泰著 森北出版株式会社
4.3.7-3 http://www.sgi.co.jp/visualization/van/news.html
4.3.7-4 http://nttcom.e-words.ne.jp/
4.4.1-1 m－s i z u' (DBMS)、http://www.m-sizu.com/dbmstop.htm
4.4.2-1 農林水産業の高度情報システム (オブジェクト指向データベース)
http://agrinfo.narc.affrc.go.jp/fs/cdrom/3syou/303st/t0302.htm
4.4.2-2 @ＩＴ (新しい業界標準「SQL99」詳細解説)
http://www.atmarkit.co.jp/fnetwork/tokusyuu/01sql99/sql99_0.html
4.4.2-3 情報データベース技術 (オーム社)
4.4.3-1 日本 PostgreSQL ユーザ会、http://www.postgresql.jp/
4.4.3-2 Cafe de Paison、http://isweb23.infoseek.co.jp/computer/paison/paison/
4.4.3-3 PostgreSQL 完全攻略ガイド (技術評論社)