ソフトウェア工学 77－2
（１９９１．２．７）

オブジェクト指向環境におけるジェネリックオブジェクト

森本康彦* 佐藤康臣* 岡本康介* 宮永靖之* 田中稔** 市川忠男**

広島大学大学院* 広島大学工学部**

従来のオブジェクト指向にオブジェクト間の関係とメソッドの実行の為の条件を宣言的に記述する制約表現を導入した制約オブジェクト指向システムISL-xschemeについて述べる。本システムでは、制約表現を利用することにより、関連するオブジェクトとの関係から、自らの状態や振舞いを変えるジェネリックオブジェクトを利用することができる。ジェネリックオブジェクトは、実行時に与えられたドメイン内の的確なクラスのオブジェクトに自動的にオブジェクト変換され、実行される。このジェネリックオブジェクトを利用したソフトウェア部品は再利用性、生産性が高く、これを用いて信頼性、保守性の高いソフトウェア開発が可能となる。

Generic Object in an Object-Oriented Environment

Y. Morimoto, Y. Sato, K. Okamoto, Y. Miyanaga, M. Tanaka, and T. Ichikawa

Faculty of Engineering, Hiroshima University
Shitami, Saijo-cho, Higashi-Hiroshima, 724, Japan

This paper introduces a generic object which behaves flexibly according to related objects and executions in an Object-Oriented Programing System (OOPS). Generic object is an instance of Generic class and behaves like a suitable object in its domain. In order to decide a suitable object which can meet a demand, we must take into account the relations between objects and the executions. We,therefore, have introduced constraints (invariant, precondition, and postcondition) like Eiffel's assertions in OOPS, which perform an important role to guarantee the correctness of behavior and state of a generic object. By using generic objects, we can easily construct reusable and reliable software.

1 はじめに

近年、ソフトウェアの生産性、信頼性、保守性の向上を目指し多くの研究が行なわれている。なかでもオブジェクト指向プログラミング環境[1]は、対象の自然なモデリングができ、ソフトウェアの部品化や再利用にも適している。しかし、従来のオブジェクト指向環境ではオブジェクト間の関係を宣言的に表現することができず、対等な密接に関連する複数のオブジェクト間にまたがる仕事の記述は苦手である。そこで、本研究では、オブジェクト指向に関係の宣言的な記述である制約記述を導入した制約オブジェクト指向システムISL-xschemeを開発し、そのシステム上で、制約と実行環境から自らの状態と振舞いを変えるジェネリックオブジェクトを実現した。

ジェネリックオブジェクトは、ジェネリッククラスにあるドメインを与えることによりインスタンシエイトすることができ、関連するオブジェクトとの関係やメッセージの実行条件から、実行時にドメイン内の適切なクラスのインスタンスに自動的に変換される。

本システムでは、制約を記述することで、より自然にオブジェクト間の関係を表現することができる。さらに、プログラマの思考の抽象的な概念を、ジェネリックオブジェクトを用いることにより、それに近いかたちでモデリングすることができる。また、ジェネリックオブジェクトを用いて作られたソフトウェア部品は、実行形態に従って柔軟に変換され、使用されるため、再利用性が高いと考えられる。

以下、2章で一般的なオブジェクト指向の概念と制約を導入した関連研究、3章で本システムの仕様とその振舞い、4章でジェネリックオブジェクトとその振舞いや有効性、そして、5章でジェネリックオブジェクトを利用した本システムのソフトウェア開発環境について述べる。

2 オブジェクト指向

2.1 オブジェクト指向環境

オブジェクト指向を特徴づける概念として、オブジェクト、メッセージ、クラス、継承などがあげられる。

・オブジェクトとメッセージ
　オブジェクトはプライベートデータを保持し、そのデータはメッセージにより起動されるメソッドによってのみアクセスできる。

・クラスと継承
　同様の特徴を有するオブジェクト群はクラスによってまとめて管理される。クラスを階層化することにより、サブクラスはその上位クラスのインスタンス変数、メソッド等を継承できる。

オブジェクト指向はモジュール性、再利用性、拡張性に優れ、ソフトウェアの生産性が高い。また、問題の対象をオブジェクトとして自然に表現できる[2, 3, 4]。

2.2 制約オブジェクト指向システム

オブジェクト指向言語はモジュール性、再利用性、拡張性に優れる反面、オブジェクト間の関係を直接宣言的に表わすことができず、それらをメソッドを用いて手続的に表現しなければならない。このような手続的な表現は、ユーザが直感的に理解しにくく、特定の利用法に依存した形でしか関係を表現できないため柔軟性に欠けるという問題がある。

そこで近年、構造記述に優れたオブジェクト指向に、関係の宣言的記述ができる制約表現を導入したThingLab[5]、ThingLabII[6]やCoral[7]、Garnet[8]などのシステムが考案され、シュミレーションやグラフィックユーザインタフェース等の分野で成果をあげている。また、Eiffel[9]には、ソフトウェアの正確さと頑丈さに加え、デバッグに利用するために宣言的なassertion(class invariant, precondition, postcondition)を記述できるという特徴がある。本研究では、オブジェクト間の満たさなければならない関係とメソッドの実行に必要な条件とを制約により記述し、これを抽象的に記述されるジェネリックオブジェクトの振舞いや状態を自動的に具体化するための条件として利用する。

3 制約オブジェクト指向システム：ISL-xscheme

本章では、xscheme言語を拡張して開発した制約オブジェクト指向システムISL-xschemeについて述べる。

3.1 クラス定義と制約記述

クラス定義は、図1(a)のように、クラス名、インスタンス変数、クラス変数、スーパークラスを与えることにより行う。また、このようにして作られたそれぞれのクラスに対し、図1(b)のようにメソッドを定義できる。インスタンス変数に対して初期値を与えたい時は、インスタンスを生成する"new"メソッド起動時に必ず実行される"isnew"メソッドを記述し、初期値を与える。

```
(class 'new
   'Tank                ;class-name
   '(fuel max-fuel)     ;instance-variable
   '()                  ;class-variable
   object)              ;super
```

(a) Definition of Tank Class クラス定義

```
(Tank 'answer 'isnew '()
  '((set! fuel 0)
    self))

(Tank 'answer-cond 'charge! '(f)
  '((<= (+ fuel f) max-fuel)
    (<= 0 f))                  ;precondition
  '((set! fuel (+ fuel f)))    ;body
  '((= fuel (+ old.fuel f)))) ;postcondition
```

(b) Definition of Tank Class method メソッドの記述

```
(Tank 'definvariant!
  '((<= fuel 0)
    (<= fuel max-fuel)))
```

(c) Definition of Class invariant クラスインバリアント

図1 クラス記述
Fig.1 Class Description

メソッドの定義では、まずメソッド名と引数を与え、次にメソッドの実行に必要な条件precondition、メソッド実行部

のbody、メソッド実行後に保証する状態postconditionをそれぞれ記述する。precondition、postcondition中では、そのメソッドの引数や他のオブジェクトのインスタンス変数値を参照できるほか、メソッドの実行前の値をoldという特別のオブジェクトを用いて参照することができる。

クラスが本質的に満たさなければならない制約であるクラスインバリアントは、図1(c)のように"definvariant!"を用いて各クラスに定義できる。

インバリアントや各condition等の制約文(<constraints>)は、図2(a)のようなシンタックスで、各クラスやメソッドに宣言的に記述する。各制約(<constraint>)中の関係(<relation>)は、システムに用意してあるプリミティブな関係に加え、図2(b)の「サブクラス関係」のように"defrelation"を使って必要に応じて定義することができる。

<Constraints> ::= (<Constraint> {<Constraint>})

<Constraint> ::= (Relation instance-variable {instance-variable})

<Relation> ::= <|<=|>|>=|==|...

(a) Syntax of Constraint 制約のシンタックス

```
(Class 'defrelation 'Subclass
  '(lambda (X Y)
     (or (equal (get-class X) (get-super-class Y))
         (and (equal (get-class X) (get-super-class $Z))
              (Subclass $Z Y)))))
```

(b) Definition of Relation 関係の定義

図2 制約と関係
Fig.2 Constraints and Relation

3．2 制約の種類と処理

本システムで扱う制約は、クラスインバリアント、precondition、postconditionの3つであり、次のような制約を表わす。

- class invariant
  各クラスのインスタンス変数間の満たさなければならない関係を表わし、クラスが本質的にもつ性質を意味する。クラスの全てのインスタンスに適用され、サブクラスにも継承される。
- precondition
  メソッドを正しく実行するための必要条件を表わし、この制約が満たされた状態で実行されたなら、postconditionと実行直後のクラスインバリアントが保証される。
- postcondition
  メソッド実行後の状態を表わす。preconditionが満たされた状態で実行された場合、メッセージセンダに対してこの状態を保証する。

従来のオブジェクト指向システムと同様にメソッドは下位クラスに継承され、その際メソッドをオーバーライドせずに継承している場合、preconditionとpostconditionもそのまま下位クラスに継承される。

これらの制約はオブジェクトにメッセージが送られてから評価される。オブジェクトにメッセージが送られるとanswer、new、isnewなどのシステムメソッド以外の全てのメソッドは実行前にinvariantとそのメソッドのpreconditionをチェックする。正しければメソッドが実行され、実行後postconditionとinvariantをチェックし、メッセージが正しく振舞ったかどうかを検査する。違反が発見された場合、それを警告して適切な例外処理を行う。ジェネリックオブジェクトはこれらの制約の違反が起こらないように自らの状態を変化させる。ジェネリックオブジェクトによる制約充足の詳細は次章で述べる。

3．3 ジェネリッククラス

ジェネリックオブジェクトは、すべてジェネリッククラスのインスタンスである。ジェネリッククラスは図3のように定義されており、以下のインスタンス変数をもつ。

- currect-object…現在のオブジェクトを保持する
- domain…カレントオブジェクトが属し得るクラスの範囲を規定する
- promise…送られてきたメッセージを保持する
- data…カレントオブジェクトに保持しきれないデータを格納する

ジェネリッククラスのオブジェクトには「送られたメッセージをカレントオブジェクトが受け付けなければならない」という制約と、「関連する制約を満たすようにカレントオブジェクトの状態を変化させなければならない」という制約とを持つ。

ジェネリックオブジェクトは、このジェネリッククラスに適切な抽象クラスをドメインとして与えることによりインスタンシエイトできる。

```
(class 'new Generic
   '(current-object
     domain
     promise
     data)
   '()
   object))

(Generic 'definvariant!
 '((accept-method current-object)
   (satisfy-constraint current-object))
```

図3 ジェネリッククラスの定義
Fig.3 Definition of Generic Class

4 ジェネリックオブジェクト

本章では、制約とメソッド実行時の条件、状態により柔軟に自らの状態、振舞いを変えるジェネリックオブジェクトについて述べる。

4．1 ジェネリックオブジェクトの概念

ジェネリックオブジェクトはジェネリッククラスのインスタンスであり、ドメインとして与えられたある抽象的クラスの全ての下位クラスに属し得る総称オブジェクトとして振舞う。ジェネリックオブジェクトは、制約とメッセージに従い、実行時に具体的なインスタンスへとオブジェクト変換され、実行される。

図4に示されるクラス階層中でOrderedDataを引数としてジェネリックオブジェクトmydataを生成した場合、そのジェネリックオブジェクトは実行条件と制約に従ってList、Vector、Stringとして振舞う（図5）。例えば、このmydataに"m

ake-vector! 'a "b" 3"というメッセージを送ると、mydataのカレントオブジェクトがVectorとして生成される。このVectorに具体化されたmydataは、従来のVector同様、"ref"メソッドにより値を参照することができる。また、従来のVectorではappend等のサイズの変更を伴うメソッドは使えなかったが、ジェネリックオブジェクトmydataでは、"append! 'd"が送られた場合、カレントオブジェクトがVectorからこのメッセージをacceptできるListへオブジェクト変換されて実行される。つまり、このオブジェクトは、コンパクトでデータの様々な操作に富むListやString、固定長で効率のよいVectorの両方の特徴を持つ。

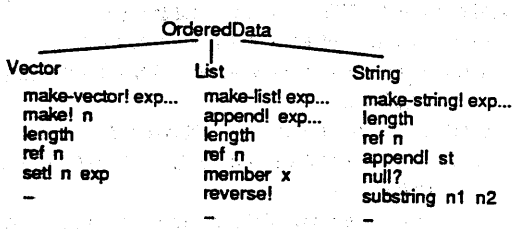

図4 クラス階層

Fig.4 Class Hierarchy and protocols

```
> (define mydata (Generic 'new OrderedData))
mydata

> (mydata 'make-vector 'a "b" 3)
— object transformation
— mydata: OrderedData to Vector
#(a "b" 3)

> (mydata 'ref 0)
a

> (mydata 'append 'd)
— object transformation
— mydata: Vector to List
(a "b" 3 d)
```

図5 ジェネリックオブジェクトの概念

Fig.5 Concept of Generic Object

4．2 制約の処理

ジェネリックオブジェクトにメッセージが送られた時、クラスインバリアント"(accept-method current-object)"により、まずcurrent-objectが、送られたメッセージに対応するメソッドを持ち、関連するインバリアントとそのメソッドを実行するためのpreconditionを満たすかどうかをチェックする。違反する場合は、ドメイン内のクラスを検索し、適合するクラスにcurrent-objectをオブジェクト変換して実行する。

また、メッセージの実行に伴ってインバリアントやprecondition等の制約に違反が生じた場合、まず、違反を生じさせる制約中のジェネリックオブジェクトを"(satisfy-constraint current-object)"に従ってドメイン内の別のクラスにオブジェクト変換し、制約を満足させる。満足させることができない場合、違反がpreconditionならばメッセージセンダ、postconditionならばそのメソッド、インバリアントならばクラスの設計の制約違反を警告し、例外処理を行なう。

クラスを検索する際、ジェネリックオブジェクトは、まず、現在のオブジェクトのクラスのサブクラスを広さ優先探索する。満足するものがなければ1つ上位のクラスから同様の探索をする。そのジェネリッククラスのドメインクラス内に満足できるクラスがない場合、違反を起こす制約と共にエラーを示す。

4．3 オブジェクト変換

全てのオブジェクト変換は次の2つの変換、又はその組合せで実現できる。

・サブクラスへの変換

インスタンス変数の値をそのままコピー

但し、サブクラスでオーバーライドされるものに関してはサブクラスの定義を優先させる。

・上位クラスへの変換

上位クラスで定義されたもののみコピー

この時、コピーされない値は、ジェネリックオブジェクトのdataにクラス名とインスタンス変数名をキーとして保持され、次にそれを必要とする時にはその値が使われる。必要であれば、ユーザが宣言することにより、他のインスタンス変数値として使うこともできる。

5 ISL-xschemeのソフトウェア開発環境

制約とジェネリックオブジェクトを備えたオブジェクト指向システムISL-xschemeでの、ソフトウェア開発とその開発支援環境ついて述べる。

5．1 抽象性を利用したソフトウェア開発

従来、プログラマは、抽象的なものを含む要求であっても、それを具体的な細かい仕様に変換し、プログラムを作成していた。ジェネリックオブジェクトは、そのような抽象的な要求を、それに近い形でプログラムできるように設計され、プログラムの読みやすさ、生産性の向上を目標としている。また、ジェネリックオブジェクトを用いて抽象的にソフトウェアを構築することにより、あるソフトウェア部品が多くのアプリケーションに適応可能となる。さらにジェネリックオブジェクトは実行時の条件や実行環境に従って状態を自動修正するため、変更、移植も容易に行なえ、信頼性、保守性も向上する。

プログラミングにおいて、抽象的な要求にはジェネリッククラスを用いて抽象クラスを使い、具体的な要求には具体的なクラスを使う。このように、本環境では、要求をそれに近い形でプログラムに反映させることができる。従来、オブジェクト指向環境では、抽象クラスはインスタンシエイトすることができず、類似したサブクラスをクラス階層中に効率的に管理するために存在していたが、ジェネリッククラスを用いることにより、従来プログラムに使うことのできなかった抽象クラスを利用してソフトウェア開発ができる。

5．2 プログラム例と実行

抽象的な要求に基づいたプログラムがインスタンシエイトされる際、抽象的な部分はジェネリッククラスのインスタンス（ジェネリックオブジェクト）として実現する。

図6a,b,cの様なクラス階層を考える。要求が、「クラスVehicleを用いてエンジンとタンクを持つSampleクラスを作る（エンジンとタンクに対して具体的要求はない）」である場合、本システムではジェネリッククラスを用いて図7の様に

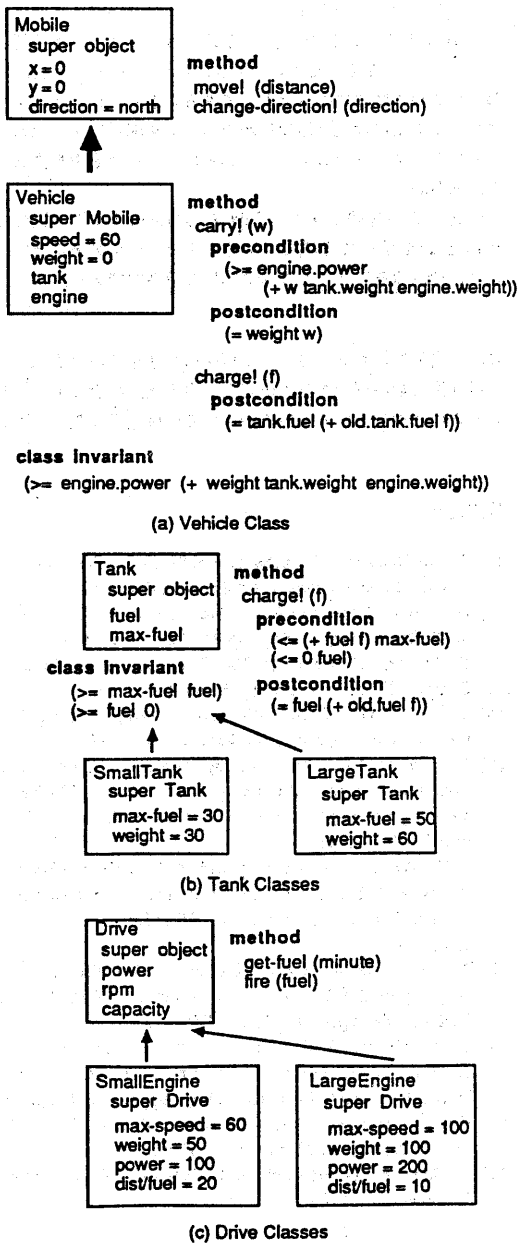

図6 ソフトウェア部品
Fig.6 Software Components

抽象表現を含むSampleクラスを定義できる。このクラスをインスタンシエイトしてオブジェクトmycarを生成すると、mycarのengineとtankにはそれぞれDrive、Tankクラスをドメインとするジェネリックオブジェクトが生成される。

このmycarに"charge! 20"というメッセージを送る。メソッドの実行前にインバリアントとpreconditionがチェックされ、ジェネリックオブジェクトのtankとengineは、それぞれ、制約を満足するSmallTankとSmallEngineに具体化される。Sampleクラスのメソッドcharge!の実行によりジェネリックオブジェクトのtankに"charge! 20"が送られ、同様にインバリアントとpreconditionをチェックし実行する（図8a）。

さらにmycarに"carry! 30"を送る。この時、carry!メソッドのpreconditionのweightに対する制約から、現在の状態（engineがSmallEngine、tankがSmallTank）ではこのメソッドを受け付けることができない。しかし、制約中にあるengine、tankはジェネリックオブジェクトなので、制約を満足するようにengineをLargeEngineにオブジェクト変換し、違反を起こすことなくこのメソッドが実行される（図8b）。

このように、ジェネリックオブジェクトを用いたプログラムは要求に近く、読みやすいだけでなく、応用や実行に応じて柔軟に対応可能である。また、変更に対しても寛容であり、保守性も優れる。

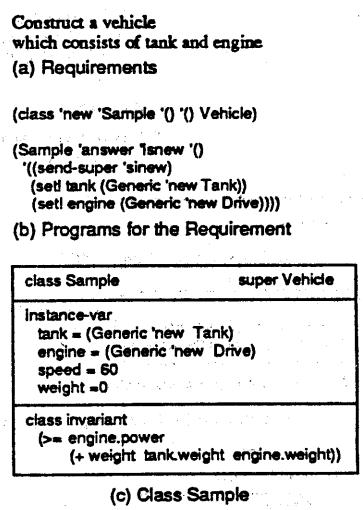

図7 プログラム例
Fig.7 Abstract Program

5．3 モードの切替と自動詳細化

ジェネリックオブジェクトの特性を生かし、本環境ではメソッドの実行を通じてソフトウェア開発を行える。そのようなソフトウェア開発を有効に行うため、本環境ではGeneralモードとSpecifyモードの切替機能を提供する。

Generalモードでは実行毎に制約を評価してジェネリックオブジェクトを柔軟に変化させる。そのため、このモードで実行するときはジェネリックオブジェクトの総称性を保つことができる。

Specifyモードをonにすると、実行される時に必要な条件（制約）がandで結合される。このモードでは、実行毎にジェネリックオブジェクトは具体化、詳細化される。

プログラマは、Generalモードで実行を確認しながらプロトタイプを作成し、Specifyモードに切り換えて具体的に必要なテストケースを実行することによって適切なソフトウェアを簡単に開発することができる。

5．4 クラス選択とブラウザ

本システムでは、何も指定がなければ、ジェネリックオブジェクトのcurrent-objectのクラスを選択する時、先に述べた様にクラス階層に対して広さ優先探索を行い、最初に制約を満足するクラスを選ぶ。

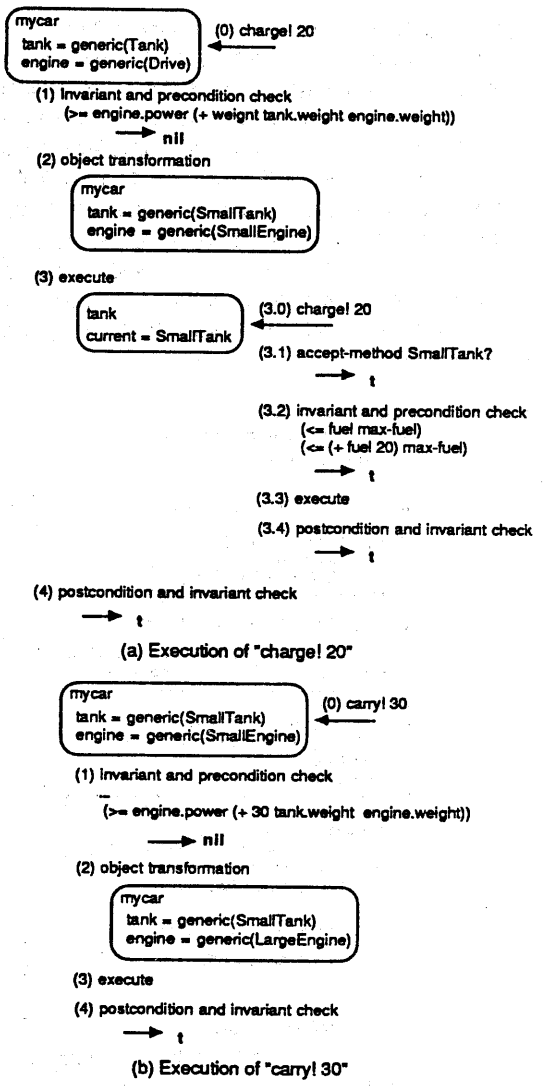

図８実行例

Fig.8 Behavior of Generic Object

また、ユーザは、ブラウジングを指定することにより、より最適なクラスを選択することができる。この場合、システムはクラス検索を行う際にドメイン内のすべてのクラスを調べ、適合可能なクラスを示しユーザに選択させる。

6　まとめ

制約オブジェクト指向システムISL-Xschemeを開発し、ジェネリックオブジェクトを実現した。ジェネリックオブジェクトは、関連するオブジェクトとの関係から、自らの状態を柔軟に変えることができる。このジェネリックオブジェクトを利用することにより、再利用性、信頼性の高いソフトウェア開発が行える。

ジェネリックオブジェクトの状態をシステムが自動的に変化させているが、プログラマの要求の抽象度をドメインとして保証しているため少なくともそのドメイン内のオブジェクトであればプログラマの要求を満たしており、また振舞い（メソッド）もそれぞれのprecondition、postconditionを満足する限り要求に従ったものである。但し、ジェネリックオブジェクトの存在と振舞いがデバッグを困難にするため、ジェネリックオブジェクトには専用のトレーサを用意する必要がある。

現在、この実験システムをUNIX、XWindow上にCと拡張したxschemeを用いて作成している。またこのシステム上で、より信頼性の高いソフトウェア部品を構築中である。

謝辞　研究を進めるにあたって有益な助言を頂いた広島大学工学部第二類情報システム研究室の平川正人博士および、システムの実現にあたって協力頂いた吉嶺良之氏に感謝する。

参考文献

[1] A. Goldberg and D. Robson: Smalltalk-80 -The Language and It's Implementation- Addison-Wesley, 1983.

[2] G. Booch, "Object-Oriented Development," IEEE Trans. on Software Engineering, pp.211-221, Feb., 1986.

[3] B. Meyer, "Reusability: The Case for Object-Oriented Design," IEEE Software, pp.50-64, Mar., 1987.

[4] B. J. Cox: Object-Oriented Programming: An Evolutionary Approach, Addison-Wesley, 1986.

[5] A. Borning, "The Programming Aspects on ThingLab, A Constraint-Oriented Simulation Laboratory," ACM Trans. on Programming Languages and Systems, vol.3, No.4, pp.353-387, 1981.

[6] J. H. Maloney, A. Borning and B. N. Freeman-Benson, "Constraint Technology for User-Interface Construction in ThingLab II," ACM Proc. of OOPSLA'89, pp.381-388, Oct., 1989.

[7] P. Szekely and B. Myers, "A User-Interface Toolkit Based on Graphical Objects and Constraints," ACM Proc. of OOPSLA'88, pp.36-45, Sep., 1988.

[8] B. A. Myers, D. A. Guise, R. B. Dannenberg, B. V. Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal, "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," IEEE COMPUTER, vol.23, No.11, pp.71-85, Nov., 1990.

[9] B. Meyer: Object-Oriented Software Construction, Prentice-Hall, 1988.