ビジュアル言語による仮想クラス生成

ビヨーン ミカエル，穂鷹良介
筑波大学
michael@wiz.sk.tsukuba.ac.jp; hotaka@shako.sk.tsukuba.ac.jp

概要

JDMF-92 v.1.2 のクラス階層を応用データモデル開発の静的な基礎として利用した．データ操作，蓄積，検索の高速化のためには静的な型の言語を利用するのが最も適している．しかしある応用データモデルを作り出すためには利用者はJDMF-92の階層中に新しいクラスを作る必要がある．そのクラスは利用者が属性またはメソッドを追加/削除できるという意味で変更可能であるため，本来ならば動的な型をサポートする言語が必要とされる．
JDMF-92の核部分の実現に静的な型の言語を用いながら，利用者に動的な型の言語を見せるという目的のために仮想クラス概念を導入した．

キーワード：JDMF, データモデル，仮想クラス，クラス階層，クラス変更，ビジュアル言語

Virtual Class-Generation in Visual Languages

Michael Björn, Ryosuke Hotaka
University of Tsukuba
michael@wiz.sk.tsukuba.ac.jp; hotaka@shako.sk.tsukuba.ac.jp

Abstract

The class hierarchy of JDMF-92 v. 1.2 is used as a static base for developing application data models. It is best implemented in a static type language, since this would allow speed optimization for data manipulation, storage and retrieval. However, to create an application data model, the user creates new subclasses in the JDMF-92 hierarchy. These classes are user-modifiable (e.g. attributes and methods can be added/deleted) and thus would have to be implemented in a dynamic type language.
The use of virtual classes is an attempt to bridge the ideas of implementing the core of JDMF-92 in a static type language and user-classes in a dynamic type language.

Keywords: JDMF, data model, virtual class, class hierarchy, class modification, visual language

Introduction

JDMF-92 is a data modeling facility intended primarily for business use – in the sense that it is not only a theoretical model but explicitly intended for actual use. The requirements on software in the business world are quite different than those in the academic world. Some of these different requirements are:

(1) **performance:** Data must be available in the right time and place

(2) **avoidance of ambiguity:** Misinterpretation of data must be avoided. Unclassifiable data is not allowed in the system – strong typing is required.

(3) **ease of use:** A data modeling facility is not strictly aimed at programmers – the need of code writing when designing an application data model should be minimized.

The requirements of the academic world, would on the other hand emphasize:

(4) **theoretical soundness:** Rigid definitions make for better understanding and classification.

(5) **ease of development and experimentation:** Even though any one implementation of JDMF-92 has a static core class-hierarchy, it should be easily extendible and modifiable between versions. JDMF-92 is intended to be a standardization vehicle for data models. As standards develop, the core class-hierarchy will be extended.

Finally we have one requirement intrinsic to the idea of a data modeling facility:

(6) **user-definable classes:** Users design application data models by adding their own classes to the core JDMF-92 hierarchy, so user-definable and modifiable classes must be supported.

Meeting the design requirements

To meet the design requirements, we have decided to develop JDMF-92 in a visual language called Prograph (see Appendix 1). Prograph uses a model of computing called data flow (see Appendix 2). This language meets our requirements in the following way:

Performance (1) is fairly high because Prograph is compiling and has an integrated engine for multi-user access to secondary storage. The data flow model also promises great speed enhancements as parallel and distributed processing becomes more generally available, while keeping backward compatibility with conventional serial computers.
Prograph is a strongly typed language (2).
Ease of use (3) can be achieved by building a menu-driven method generator. This approach was demonstrated in [Björn93] where a totally menu-driven query-language was implemented for relational algebra in a relational database system. (The full Prograph source code for this RDBMS, called Re(ve)lations, is published in [TGS92].)
Also, Prograph adheres to the data flow and object-oriented models and can be considered theoretically sound (4). Arguably less so than SmallTalk, but more than C++.
In [Björn93] the ease of development (5) in Prograph was also demonstrated. More specifically, it was argued that the development process was simplified because of visual code advantages in: coding speed, debugging, readability of code and reusability of objects.

Requirement (6), user-definable classes, can however not be met as easily. The reason for this is that (6) implies a dynamic type language and is orthogonal to (1), implying a static type language. These terms are precisely defined below, but the essence of the problem is that run time modification can only be done in an interpreted language whereas

performance is best optimized in a compiled language. This paper presents a prototype implementation of JDMF-92 in which this problem was solved by extending the compiled language with virtual classes.

What are virtual classes good for?

Before giving a more rigorous description of virtual classes, we give an intuitive explanation of what we want to achieve.
Let us look at SmallTalk/V, which supports user-definable classes (but does not support some other requirements, notably (1) and (2)). SmallTalk/V consists of three parts: a very small compiled application defining the core of SmallTalk, a source file containing the code for the SmallTalk language, and an image which contains the SmallTalk objects that make up the environment ([DgTlk88]).
The compiled SmallTalk/V application itself has been programmed in another language – let us call this language (which could be for instance C, C++, Pascal – or Prograph) the implementation language. The classes which are defined in the image do not correspond to classes (or data types – classes are treated as data types in this paper) in the implementation language. In this sense they are virtual to the implementation language.
To meet our other requirements, what we would like to do is to extend the part which is directly defined in the implementation language, and minimize the part which is virtual to the implementation language, as in the following figure:

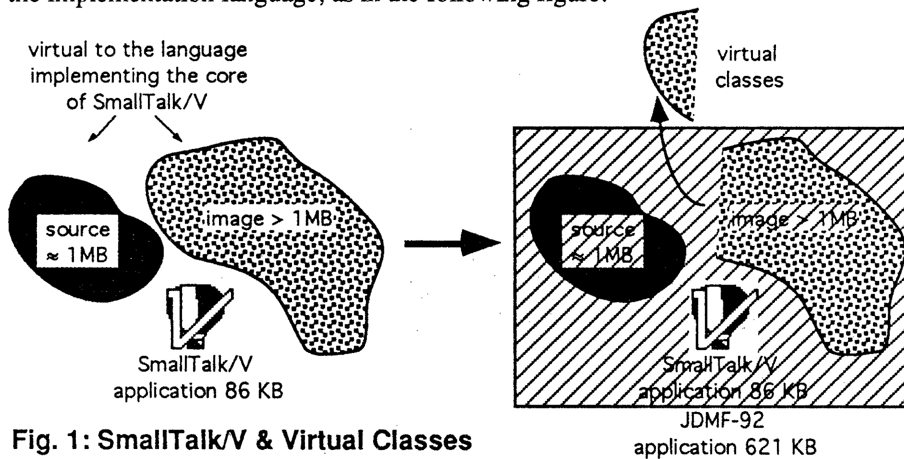


Fig. 1: SmallTalk/V & Virtual Classes

Also notice that in this way we get an elegant separation of JDMF-92 classes and user-defined classes.

A prototype implementation

A prototype implementation of JDMF-92 in Prograph has been successfully completed, showing that it is indeed possible to implement virtual classes without any disadvantages.

Now, let us look at a more formal description of virtual classes used in the implementation. All definitions that are given should be seen as extensions to the definitions of JDMF-92 given in [JDMF-92]. A notable limitation is that although the JDMF-92 allows unlimited levels of meta classes, for practical purposes the meta class implementation has been limited so that every class is an instance of an ObjectClass. In particular, any ObjectClass is an instance of itself. This self-describing instance is called a metameta instance in this paper. (See [Nogu93] for a more complete treatment of the ObjectClass.)

Static type and dynamic type languages

A static type language is a language where type-checking is done at compile time. Static typing absolves the system of the necessity to carry out run time type checks, thereby reducing execution time, and removes the need to store type information with data, which economizes on storage. However, any subsequent change in the class-hierarchy requires recompilation of the class-hierarchy ([Hugh91]). Prograph is a static type language.
A dynamic type language is similarly a language where type-checking is done at run time. A dynamic type language allows changes to the class-hierarchy at run time. A typical example would be SmallTalk ([Hugh91]). In SmallTalk everything can be changed at run time, to the point where the environment is destroyed ([DgTlk88]).

Definition of virtual classes

We use the term static_types to refer to all classes enumerated in [JDMF-92]. That is; static_types = {Object, PrimitiveObject, DeclaredObject, AtomicObject, Structured-Object, SetObject, ListObject, AttributedObject, NamedObject, MetaObject, Attribute, SuperSubRelation, ObjectClass, PrimitiveObjectClass, DeclaredObjectClass, AtomicObjectClass, StructuredObjectClass, SetObjectClass, ListObjectClass, AttributedObjectClass, NamedObjectClass}

A class X is a virtual class iff:
$X \subset \text{Object} \wedge X \cap \text{ObjectClass} = \emptyset \wedge X \in \text{ObjectClass} \wedge X \notin \text{static_types}$
Or, using words, a class is a virtual class if and only if:
the class is a subclass of Object, and the class and ObjectClass are mutually disjoint, *and* the class is an instance of an ObjectClass which is not enumerated in the definition of JDMF-92.

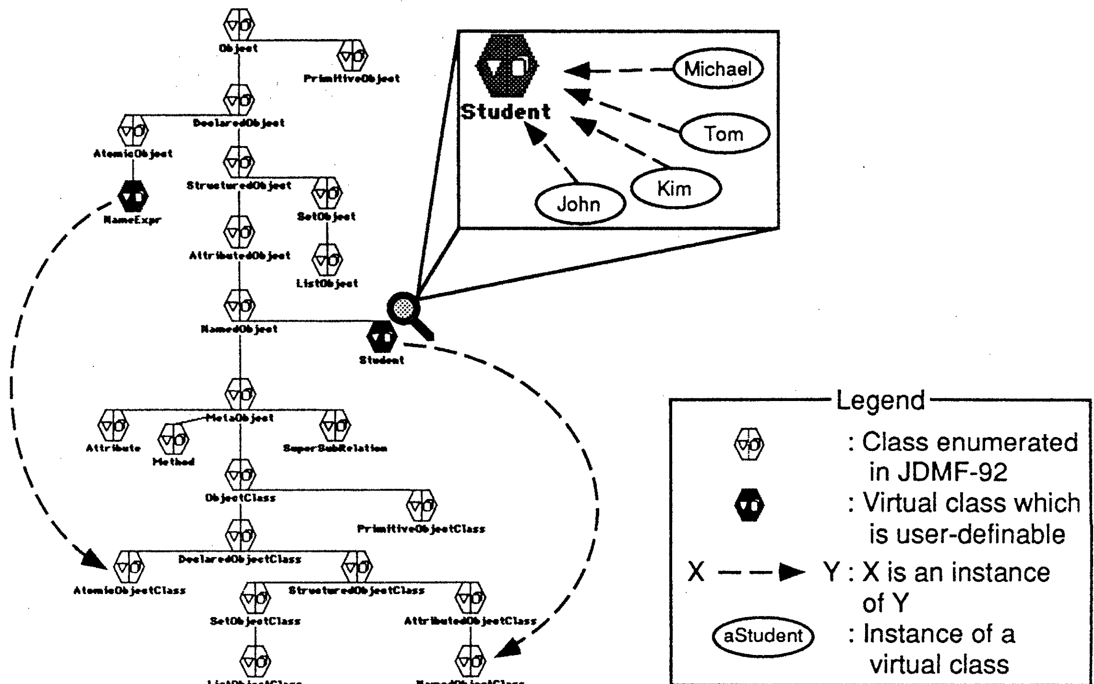


Fig. 2: JDMF-92 core classes & virtual classes

Virtual classes have explicit references to the ObjectClass of which they are instances (i.e. any virtual class X $\in$ ObjectClass), they are totally described in the JDMF-92 model and can for this reason be manipulated, stored and retrieved without the usual restrictions imposed by static type checking.

A similar design is found in [Card88], where a *dynamic object* is defined as a pair consisting of a type and an object of that type. Such an object is dynamically type-checked in the context of an otherwise statically typed language. The main differences are that dynamic object definitions in [Card88] are not necessarily seen as class definitions, and that the concept 'pair' is used instead of the 'instance of' concept.
Similar to the description in [Card88], a virtual class can be coerced back to the type (i.e. a JDMF-92 core class) it is based on.

According to [Card88], advantages of the above design are:
- suitability for export to persistent storage, and
- preservation of strong typing when static typing is impossible

Definition of an instance of a virtual class

An instance of a virtual class is an instance which has an explicit reference to that virtual class of which it is an instance.
An instance of a virtual class can be coerced to an instance of the type (i.e. a JDMF-92 core class) which the virtual class is based on.

Creation of a virtual class

The prototype implementation uses a polymorphic instance creation method called "new". Since creating a virtual class is equal to creating an instance of an ObjectClass, a virtual class is created by sending a "new" message to the metameta instance of an object class.

Sending "new" to an ObjectClass...

... means sending "new" to its metameta instance. The result is an instance of the ObjectClass – in this case a virtual class definition 'Student'.

The "new" message is sent to this instance

This is the resulting instance

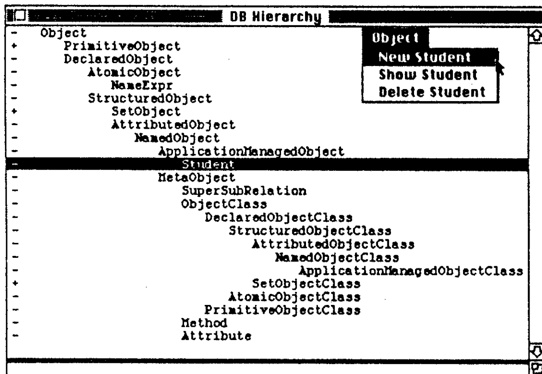
Fig. 3: Creation of a virtual class

Deletion of a virtual class is performed by sending a "delete" message to the class of which the virtual class is an instance. Checks are performed to ensure that no instance of the class to be deleted exist. The "delete" method is polymorphic and behaves differently in different classes.

Creation of instances of a virtual class

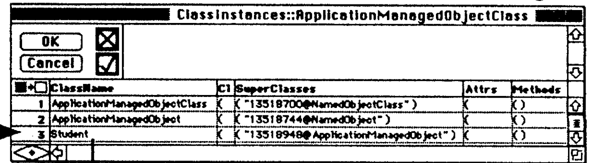
Creating an instance of a virtual class is the same as inputting real data. To do this a "new" message is sent to the ObjectClass instance defining the virtual class of which we want an instance.

An instance of the virtual class 'Student' is created...



new

... by sending "new" to the ObjectClass instance 'Student' (which was created in Fig. 3)



We then input our data concerning this instance of Student, in nested spreadsheets.

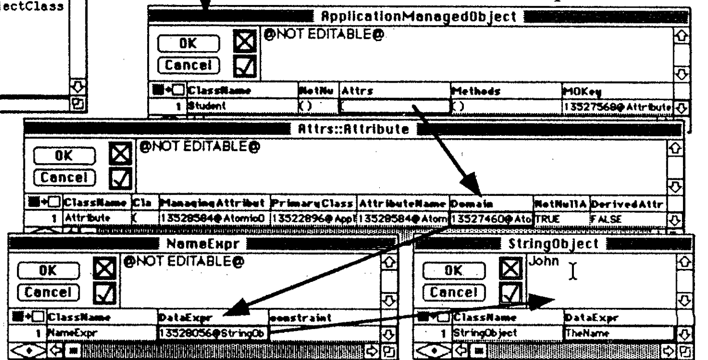


Fig. 4: Creation of a virtual class instance

By repeating the process described in fig. 4, we can create many instances of the virtual class 'Student'. These instances can be viewed in a spreadsheet:

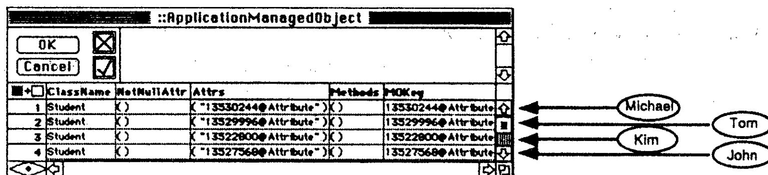


Fig. 5 Viewing virtual class instances

Deletion an instance of a virtual class is performed by sending a "delete" message to that virtual class which the instance belongs to.

Schema and database

A schema in the JDMF-92 prototype implementation is a set of virtual class definitions. A database is a set of instances belonging to classes which fulfill the condition that if X is

a class then $X \subset Object \wedge X \cap ObjectClass = \emptyset \wedge X \in ObjectClass$ (i.e. classes which are not in the ObjectClass hierarchy). The schema and the database are stored separately on secondary storage.
Further, since a database may contain instances of classes which are stored in the schema, the schema must be loaded before the database is queried.

Using standard terminology, defined in [ISO10032], the schema and the database constitue a level pair, where if the database is at level N the schema at level N+1.

Definition of attributes

User-defined attributes in JDMF are all instances of the class Attribute. Virtual classes can have attributes which are instances of the class Attribute.
Attributes of a virtual class are inherited by all subclasses of that class.

Definition of methods

In the prototype implementation discussed in this paper, methods are implemented as instances of an immediate subclass "Method" to the class MetaObject, since no exact specification of methods yet exists for JDMF-92. This approach is influenced by the programming language Eiffel, where attributes and methods are abstracted into a single concept called "feature" [Mey88].
Assuming that there is a subclass "Method" to the class MetaObject, the inheritance of methods will be exactly the same as the inheritance of attributes. Thus remains the specification of a language (i.e. a specification of syntax) for assembling methods.
The goal is to design a language which is optimized for data manipulation and supports set-oriented database queries (JDMF-92 currently only supports navigational queries).
As already mentioned, the implementation will be an extension of the menu-driven method generator developed in [Björn93], requiring little or no coding by the user. We are currently working on defining a set of basic primitive methods (which will be selectable through menus), basic logical connectives and a syntax for combining the primitive methods with the logical connectives into user-defined methods.

Summary

We have defined several business-world as well as academic design requirements which an implementation of a data model must fulfill: high performance, avoidance of ambiguity (strong typing), ease of use, theoretical soundness, ease of development and experimentation, and support for user-definable classes. However, the requirement of high performance implies implementation in a static type language whereas user-definable classes imply implementation in a dynamic type language.
By using the concept of virtual classes we can preserve the strong typing requirement and at the same time have dynamically type-checked class definitons in an otherwise statically typed environment. Virtual class definitions are stored separately on secondary storage and are not part of the enumeration of classes in the definition of JDMF-92.
A prototype implementation of JDMF-92 using virtual class definitions has been completed, showing that this approach is indeed feasible.
Further, using a visual development environment helps us attain the requirements of high performance, avoidance of ambiguity, and ease of development and experimentation.

References

[Björn93] Björn M., Designing Database Management Systems in Prograph, Master's Thesis, University of Tsukuba, 1993

[Card88] Cardelli M., MacQueen D., Persistence and Type Abstraction, pp 31 - 41, Data Types and Persistence, Atkinson M.P., Buneman P., Morrison R. (Eds.), Springer-Verlag, 1988

[DgTlk88] Digitalk, SmallTalk/V Mac, Tutorial and Programming Handbook, Digitalk, Inc, 1988

[Gur87] Gurd J.R., Dataflow Architectures, pp 51-68, Major Advances in Parallel Processing, Technical Press, 1987

[Hugh91] Hughes J.G., Object-Oriented Databases, Prentice-Hall, 1991

[ISO10032] ISO/IEC 10032: Reference Model of Data Management

[JDMF-92] Information resource schema research and standardization committee, A Data Modeling Facility: JDMF-92

[Mey88] Meyer B., Object-oriented Software Construction, Prentice-Hall, 1988

[Nogu93] On the implementation of class methods and class variables in JDMF-92 (In Japanese), 1993

[Rum77] Rumbaugh J., A data flow multiprocessor, IEEE Trans. on Computers, pp 138-146, C-26, 1977

[Rum91] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., Object-Oriented Modeling and Design, Prentice-Hall,, 1991

[TGS90] TGS Systems, Prograph Reference, TGS Systems Ltd, 1990

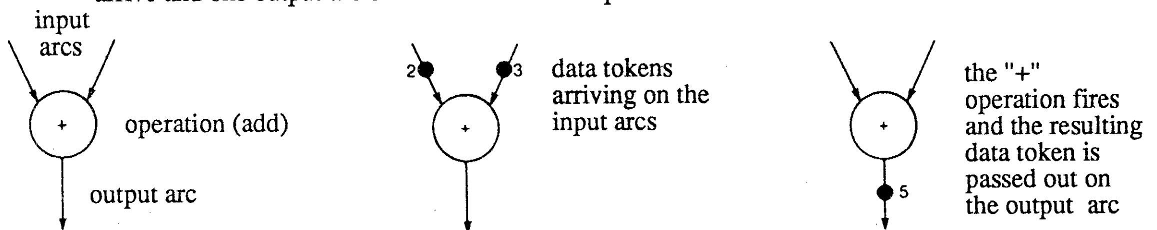
[TGS92] TGS Systems, Prograph Essentials, TGS Systems Ltd, 1992

Appendix 1 – Introduction to Data Flow

According to [Gur87], data flow is a technique for specifying parallel computations at a fine-grain level, in the form of two-dimensional graphs in which operations that are available for concurrent execution are written alongside one another, whilst those that must be executed in sequence are written one under the other. Data dependencies between individual operations are indicated by directed arcs linking the operations together.
The resulting data dependence graph shows how instructions are dependent on data – an operation is executed (or 'fired') first when all required data is available, and is said to be data-driven. This can be illustrated by depicting data-carriers (or 'tokens') which carry data values on the arcs of the graph. Once all required tokens have arrived at the input nodes of a specific operation, it can be executed independent of other operations, and several independent operations can thus be executed in parallel.

Rumbaugh's data flow notation

The basic, generic node in Rumbaugh's notation as described in [Rum77] is an operation requiring certain input(s) and giving certain output(s). Consider for example the addition operation, which consists of a "+"-node with two input arcs on which the data tokens arrive and one output arc on which the result is passed out:



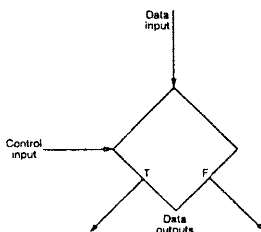
Rumbaugh's control nodes.

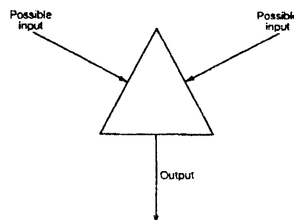
Three simple control nodes called *switch*, *merge* and *branch* (see [Rum77]), are needed for construction of more elaborate logical operations corresponding to IF-THEN statements and LOOP structures found in most programming languages. (The above described "+" operation is a merge node.)

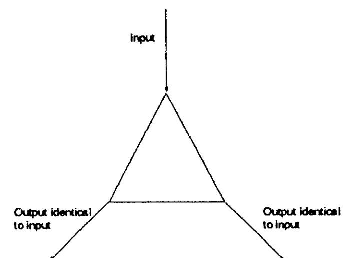
switch
The input token is placed on the output arc selected by the control input.

merge
The input arrives either on the left or on the right, and is placed on the output.

branch
The input is placed on both outputs.





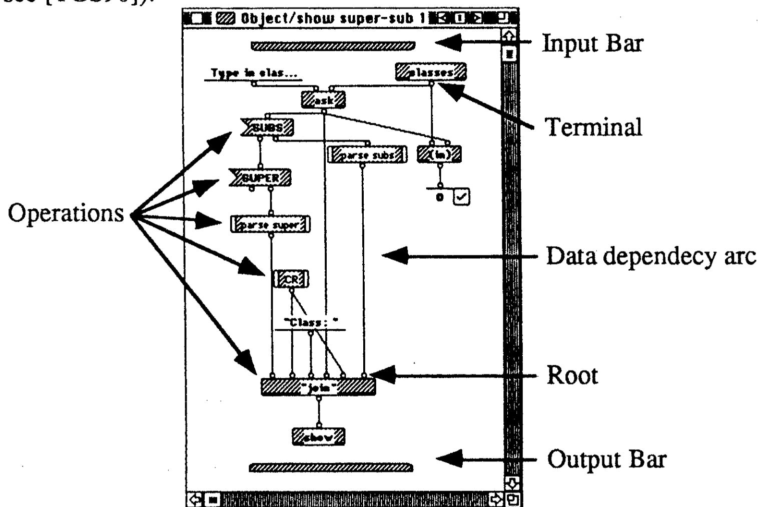


Appendix 2 – Introduction to Visual Languages

There is no general consensus as to what a visual language is, other than that such languages use graphical information such as diagrams, icons or tables in the actual process of programming. The program meaning should be as bound to its visual representation as a text-based language is tied to its syntax. In this paper, a visual language is taken to be a language which uses a special form of graphic notation developed for data flow (see Appendix 1).

Prograph, the implementation language used for the JDMF-92 implementation discussed in this paper, is one of the few currently available visual data flow languages. In Prograph the data flow model is extended with object-orientation a data token (described in Appendix 1) can be any kind of object.

An example of code (an instance method) written in Prograph (for EBNF specification, see [TGS90]):



Reasons why visual code can not be modified dynamically:

1. Difficulties of generating visual code as the result of an operation, since visual coding by definition involves visual coordination in the specification of syntax.
2. The graphical information makes code very big and cumbersome to process at run time. Thus, visual languages tend to be static type languages.