An Object Model for Multiparadigm Programming

Diomidis Spinellis, Sophia Drossopoulou, and Susan Eisenbach
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ
e-mail: {dds,scd,se}@doc.ic.ac.uk

July, 1994

1 Introduction

We became interested in multi-language object models while researching problems related to *multiparadigm programming* [SDE94a,SDE94b]. It is widely accepted that different types of tasks can be best implemented in different paradigms. As an example the logic programming paradigm is particularly well suited for implementing expert systems, while many operations on lists can be elegantly described in the functional programming paradigm. Multiparadigm programming can allow each part of a system to be implemented in the most suitable paradigm. Some of the problems in achieving this ideal are:

- accommodation of different syntactic notations,
- accommodation of diverse execution models,
- support for different implementation strategies,
- ability to use existing tools, and
- arbitrary paradigm mixing and matching.

Some of these problems can be overcome by designing around an *object-based multiparadigm programming environment*. In an object based multiparadigm programming environment every paradigm forms a class, and every module written in that paradigm is an object member of that paradigm's class. Paradigms form the class hierarchy with the target machine architecture being the root of it. Inheritance is used to bridge the semantic gaps between different paradigms. In the following paragraphs we will present these aspects in greater detail and discuss a possible design abstraction for multi-language module inter-operation.

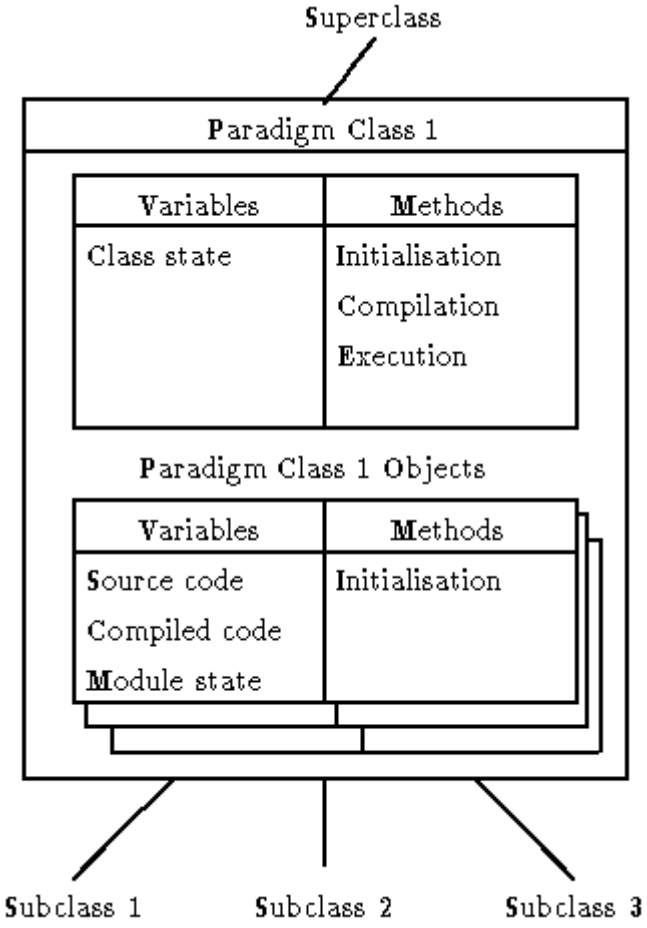
2 Paradigms as Object Classes

2.1 Objects

An object can be used as the abstraction mechanism for code written in a given paradigm. Such objects need to have at least three *instance variables* (Figure 1):

1. *Source code.* The source code contained in an object is the module provided by the application programmer.
2. *Compiled code.* The compiled code is an internal representation of that specification (generated by the class *compilation method*) that is used by the class *execution method* in order to implement the specification.
3. *Module state.* The module state contains local data, dependent on the paradigm and its *execution method*, that is needed for executing the code of that object.

Figure 1: Programming paradigm classes and objects



Every object has at least one *method*:

1.

Instance initialisation method. The instance initialisation method is called once for every object instance when the object is loaded and before program execution begins. It can be used to initialise the module state variable.

As an example, given the imperative paradigm and its concrete realisation in the form of Modula-2 [Wir85] programs, an object written in the imperative paradigm corresponds to a Modula-2 module. The *source code* variable of that object contains the source code of the module, the *object code* variable contains the compiled source, and the *module state* variable contains the contents of the global variables. In addition, the *instance initialisation method* is the initialisation code found delimited between `BEGIN` and `END` in the module body.

2.2 Classes

All classes contain at least one class variable (Figure 1):

1. *Class_state*: contains global data needed by the *execution method* for all instances of that class.

In addition paradigm classes contain at least three *methods*:

1. *Compilation method.* The compilation method is responsible for transforming, at compile-time, the source code written in that paradigm into the appropriate representation for execution at run-time.
2. *Class initialisation method.* The class initialisation method of a paradigm is called on system startup in order to initialise the class variables of that class. It also calls the instance initialisation method for all objects of that class.
3. *Execution method.* The execution method of a class provides the run-time support needed in order to implement a given paradigm.

The compilation and execution methods also contain the machinery needed to implement the import and export call gates described in section 3.

Taking as a paradigm class example, the logic programming paradigm realised as Prolog compiled into Warren abstract machine instructions [War83], the *class state* variable contains the heap, stack and trail needed by the abstract machine. In addition, the *compilation method* is the compiler translating Prolog clauses into abstract instructions, the class initialisation method is the code initialising the abstract machine interpreter, while the execution method is the interpreter itself.

2.3 Inheritance

Inheritance is used to bridge the semantic gap between code written in a given paradigm and its execution on a concrete architecture. We regard the programming paradigm of the target architecture as the *root class*. If it is a uniprocessor architecture it has exactly one object instance, otherwise it has as many instances, as the number of processors. The *execution method* is implemented by the processor hardware and the *class_state* is contained in the processor registers. The *compiled code* and *module state* variables are kept in the processor's instruction and data memory respectively.

From the root class we build a hierarchy of paradigms based on their semantic and syntactic relationships. Each subclass inherits the *methods* of its parent class, and can thus use them to implement a more sophisticated paradigm. This is achieved because each paradigm class creates a higher level of linguistic

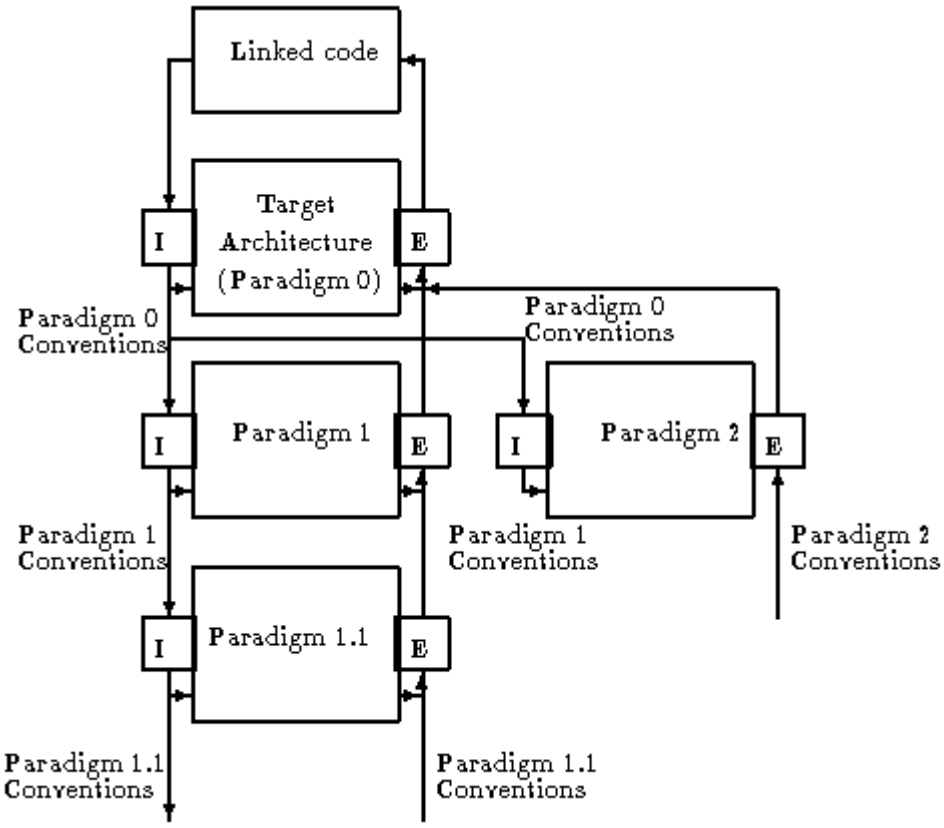
abstraction, which its subclasses can use.

As an example most paradigms have a notion of dynamic memory; a class can be created to provide this feature for these paradigms. Two subclasses can be derived from that class, one for programmer-controlled memory allocation and deallocation and another for automatic garbage collection. As another example a simulation paradigm and a communicating sequential processes paradigm can both be subclasses of a coroutine-based paradigm. Subclassing is not only used for the run-time class execution methods. Syntactic (i.e. compile-time) features of paradigms can be captured with it as well. Many constraint logic languages share the syntax of Prolog, thus it is natural to think of a constraint logic paradigm as a subclass of the logic paradigm providing its own solver method, and extension to the Prolog syntax for specifying constraints.

3 Paradigm Inter-operation

Paradigm inter-operation can be designed around an abstraction we name a *call gate*. A call gate is an interfacing point between two paradigms, one of which is a direct subclass of the other. We define two types of call gates: the import gate and the export gate. In order for a paradigm to use a service provided by another paradigm (this can be a procedure, clause, function, rule, or a port, depending on the other paradigm) that service must pass thought its import gate. Conversely, on the other paradigm the same service must pass through its export gate. The call gates are design abstractions and not concrete implementation models. They can be implemented manually or automatically by the paradigm compiler, the runtime environment, the end user, or a mixture of the three. Each paradigm provides an import and export gate and documents the conventions used and expected. The input of the export gate and the output of the import gate follow the conventions of the paradigm, while the output of the export gate and the input of the import gate follow the conventions of the paradigms' superclass (Figure 2). The target architecture paradigm combines its import and its export gate using the linked code as the sink for its export gate and the source for its import gate. Call gates can make the paradigm inter-operation transparent to the application programmer and provide global scale inter-operation using only local information.

Figure 2: Paradigm inter-operation using call gates



We must note at this point that the class hierarchy is not visible to the application programmer. The hierarchy is useful for the multiparadigm programming environment implementor, as it provides a structure for building the system, but is irrelevant to the application programmer, who only looks for the most suited paradigm to build his application. This is consistent with the recent trend in object-oriented programming of regarding inheritance as a *producer's mechanism* [Mey90], that has little to do with the end-user's use of the classes [Coo92].

Using to our approach a multiparadigm programming environment consists of a set of classes, one for each paradigm. The classes are ordered in a hierarchy whose root is the target architecture. Every class is self-contained and only needs to handle the calling conventions of its superclass and provide a mechanism for interfacing with its subclasses. Code in different paradigms is written in different source modules, which are then handled by the appropriate methods of the respective paradigm class.

4 Prototype Implementation

In order to demonstrate the validity of our approach we have implemented three prototypes:

The *integrator* is a multiparadigm application dealing with the numeric and symbolic evaluation of integrals. The symbolic evaluation is based on the backtracking resolution mechanism offered by the logic programming paradigm, and the numeric evaluation on the infinite streams implemented in the functional paradigm. Additionally, lexical analysis of the input expressions is described using regular expressions, and the expression grammar is described using a BNF syntax. Finally, expression simplification uses a term rewrite system and graphing of functions is done by directly interacting with Unix tools.

Integrator is implemented in the *blueprint* multiparadigm programming environment, a prototype implementation of the six paradigms based on our object-oriented approach. Many different implementation techniques have been applied in order to demonstrate the wide applicability of our

approach. Some of the paradigms are implemented as compilers (using existing implementations where possible); others are implemented as interpreters.

Finally, the implementation of *blueprint* is based on a multiparadigm environment generator. This allows the description of paradigms as object classes using paradigm description files. Additionally, it provides a multiparadigm link editor and support for incorporating existing compilers into multiparadigm programming environments.

5 Proposed Workshop Topics

Some topics that are interesting in the context of the work described above are the following:

Object Size Granularity
: Least common denominator systems [Mee94] are efficient in their implementation at the cost of limited flexibility in the inter-operation between modules written in different languages. Systems that introduce some sort of translation services such as the ones described in section 3 can allow for more sophisticated transactions between modules at the cost of run-time efficiency. The efficiency penalty need not be great if a substantial part of the problem is solved within a module written in one language. Thus the object size granularity pays an important role in the efficiency and flexibility tradeoffs of such systems. Design methods, environments, and techniques that affect this granularity can be discussed.

Type Systems and Dynamic Typing
: Although least common denominator approaches together with some sort of type checking link editors [Spi91] can solve many of the problems in this area dynamic typing introduces new problems that need be discussed. We propose the use of *type gates* similar in design to the *call gates* described in section 3.

Heterogeneous Architectures
: Finally, in the area of architectures we believe that an architecture can be abstracted as an object by realising that the machine language of that architecture is part of our multi-language model, and that the machine's register and memory form that object's state. A suitable abstraction should be devised for the case of networked machines or connected processors running a multi-language implementation.

References

Coo92
: William R. Cook.
Interfaces and specifications for the Smalltalk-80 collection classes.
ACM SIGPLAN Notices, 27(10):1-15, October 1992.
Sevent Annual Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '92 Conference Proceedings, October 18-22, Vancouver, British Columbia, Canada.

Mee94
: Brian L. Meek.
Programming languages: Towards greater commonality.
ACM SIGPLAN Notices, 29(4):49-57, April 1994.

Mey90
: Bertrand Meyer.
Lessons from the design of the Eiffel libraries.
Communications of the ACM, 33(9):68-88, September 1990.

SDE94a

Diomidis Spinellis, Sophia Drossopoulou, and Susan Eisenbach.
Language and architecture paradigms as object classes: A unified approach towards multiparadigm programming.
In Jürg Gutknecht, editor, *Programming Languages and System Architectures International Conference*, pages 191-207, Zurich, Switzerland, March 1994. Springer-Verlag.
Lecture Notes in Computer Science 782.

SDE94b

Diomidis Spinellis, Sophia Drossopoulou, and Susan Eisenbach.
Object-oriented technology in multiparadigm language implementation.
Journal of Object-Oriented Technology, 1994.
Accepted to be published.

Spi91

Diomidis Spinellis.
Type-safe linkage for variables and functions.
ACM SIGPLAN Notices, 26(8):74-79, August 1991.

War83

David H. D. Warren.
An abstract Prolog instruction set.
Technical Note 309, SRI International, Artificial Intelligence Center, Computer Science and Technology Division, 333 Ravenswood Ave., Menlo Park, CA, USA, October 1983.

Wir85

Niklaus Wirth.
Programming in Modula-2.
Springer Verlag, third edition, 1985.