Using Objects for Structuring Multiparadigm Programming Environments [*][†]

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

April, 1993

Abstract

Multiparadigm programming allows the programmer to write the implementation of a system in a number of different paradigms. We describe our approach to multiparadigm programming based on modeling programming paradigms as object classes. In particular, objects can be used to encapsulate program modules, and classes to encapsulate their respective paradigms. The paradigm class hierarchy can then be used to abstract common paradigm characteristics and the *call-gate*, a local inter-operation abstraction can be used to flatten the class hierarchy into a collection of paradigms. We use this object-oriented structuring mechanism to provide the base for designing multiparadigm environment generators. In order to demonstrate our approach we develop MPSS, a multiparadigm environment generator, use it to implement *blueprint*, a six paradigm programming environment, and use all its paradigms in a numerical and symbolic integration package.

1 Introduction

The word paradigm is used in computer science to refer to a family of system implementation notations that share common linguistic abstractions or theories. Thus, we talk about the *functional*, *logic*, *object-oriented*, and *imperative* paradigms. 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 the *object-oriented* paradigm can be advantageous for graphics work. As the subparts of typical real world systems can often be best implemented in different paradigms, a number of researchers have introduced the idea of *multiparadigm programming*: a programming methodology where the virtues of different paradigms are combined to ease the system implementation task. In this paper we will outline, how object-oriented technology can be utilised for structuring multiparadigm systems. We will first examine the problems that a multiparadigm programming environment has to address, and explain how objects, classes and inheritance can be used to overcome these problems. We will also describe the design and implementation of the experimental prototype systems we built to demonstrate the viability of our approach.

2 Object Based Multiparadigm Systems

The potential advantages of multiparadigm programming can be tapped only if some significant problems are overcome. These include the following:

Accommodation of different syntactic notations: Different paradigms typically deploy different syntactic notations. This allows for the most suitable notation for each paradigm to be used and for the programmer to receive the

[*] *Journal of Object-Oriented Programming* 8(1):33–38, March/April 1995.

[†] This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. This material is presented to ensure timely dissemination of scholarly and technical work.

appropriate paradigm related guidance when reading the code. This makes porting code written in a given paradigm easier.

Accommodation of diverse execution models: Different paradigms have different execution models. Almost[1] all of them can be modeled using a Turing machine; therefore they pose, in theory, no implementation problems. In practice, the execution model of the multiparadigm system must be flexible enough to provide a base for their *efficient* implementation.

Support for different implementation strategies: Some paradigms are usually implemented by being directly compiled to target machine code, others have their code interpreted, yet others have their code translated to some abstract machine notation and provide the abstract machine execution mechanism as part of the runtime support system. The choice of the execution mechanism depends on the paradigm, the target architecture and space versus time efficiency considerations. The system structure must support all these execution mechanisms to make possible an optimum implementation.

Ability to use existing tools: Implementing a programming paradigm can be a complicated process. If the system structure makes the use of existing tools that handle that paradigm (such as interpreters, compilers or runtime systems) possible, then the resulting system can be implemented in less time and, by capitalising on the existing tool investment, be more reliable and efficient.

Arbitrary paradigm mixing and matching: For a multiparadigm environment to be used by application developers, its internal operation should be transparent, and different paradigms should be easily combined.

We decided to focus our research on the *structure* of multiparadigm systems. A powerful structuring mechanism can solve many of the problems mentioned above, and create synergistic[2] effects through paradigm combination. The structuring mechanism that we have adopted is based on the structuring techniques deployed by object-oriented programming. This mechanism provides the possibility of *multiparadigm environment generators*: tools for designing and implementing multiparadigm programming systems. In the following sections we will expand on the way object classes can be used to represent and encapsulate programming paradigms, and how this representation can be exploited to create flexible multiparadigm environment generators.

2.1 Paradigms as Object Classes

A programming paradigm is really a notation for describing an implementation for solving a specific problem. This notation may resemble the notation used by the machine that will execute the implementation or it may resemble a more abstract notation suitable for describing implementations in that problem domain. At some point however, the implementation *will* be executed on a real machine and for this reason the semantic gap between the implementation paradigm and the programming paradigm of the target architecture must be bridged. This is usually done by an interpreter, a compiler or a hybrid technique. We regard all these methods as linguistic transforms from the paradigm notation to the target architecture notation. This view, although simple provides us with two insights:

1. The target architecture plays an important role when thinking of programming paradigms. The concept of the target architecture should be an integral part of multiparadigm programming environments and not an externally imposed specification, or an afterthought.

2. The target architecture naturally suggests a paradigm object hierarchy, with the target architecture forming and root of the hierarchy and other paradigms forming subclasses. Subclassing is used to create new paradigms, and inheritance to combine common features between paradigms.

It turns out that the object metaphor suits the abstraction of a “programming paradigm”, and that by using it many of the problems outlined above can be solved.

In the following paragraphs we will examine how important aspects of object-oriented programming can be related to programming paradigms and multiparadigm programming. We will present the elements of the equation [Weg87]:

object-oriented = objects + classes + inheritance

[1] An exception would be a paradigm based on real non-determinism.

[2] *Synergy* is the notion of a system having greater value than its parts.

and in addition present the definition of class variables, instance variables and methods [Nel91], in the context of multiparadigm programming.

In an object based multiparadigm programming environment programming 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 architecture being the root of it. Inheritance is used to bridge the semantic gaps between different paradigms.

2.1.1 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.

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 could correspond to a Modula-2 module. The *source code* variable of that object could contain the source code of the module, the *object code* variable could contain the compiled source, and the *module state* variable could contain the contents of the global variables. In addition, the *instance intialisation method* would be the initialisation code found delimited between `BEGIN` and `END` in the module body.

2.1.2 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 four *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.

4. *Documentation method.* The documentation method provides a textual description of the class functionality. It is used during the building phase of the multiparadigm environment, in order to create an organised and coherent documentation system.

The compilation and execution methods also contain the machinery needed to implement the import and export call gates described in section 2.2.

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

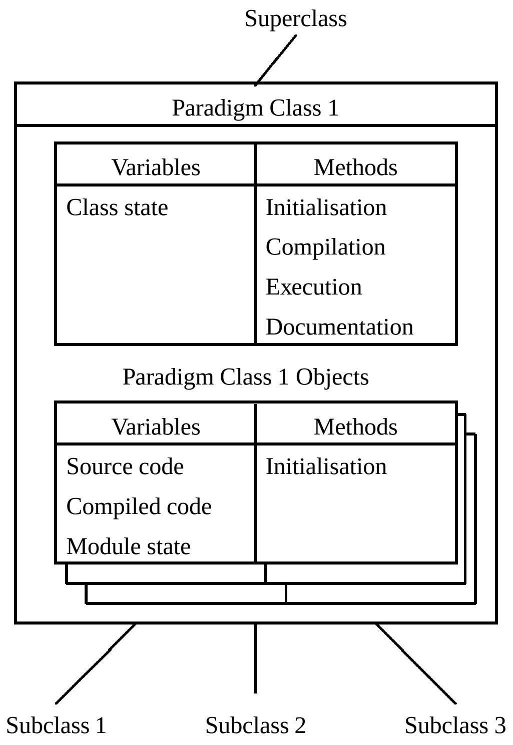

Figure 1: Programming paradigm classes and objects

2.1.3 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 created 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 could 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. A paradigm class tree based around these examples is show in figure 2.

2.2 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. Both can typically be implemented as methods of the particular paradigm class. 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 could 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

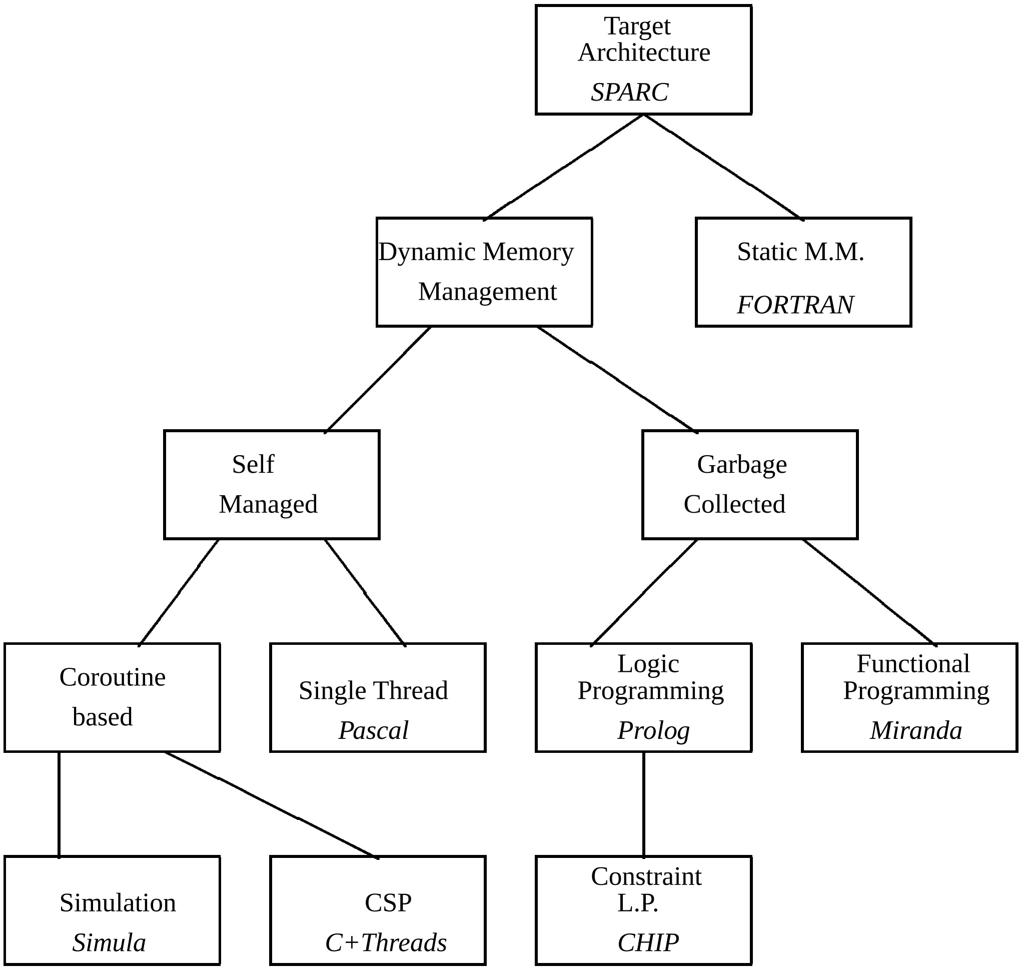

Figure 2: Paradigm class tree structure example

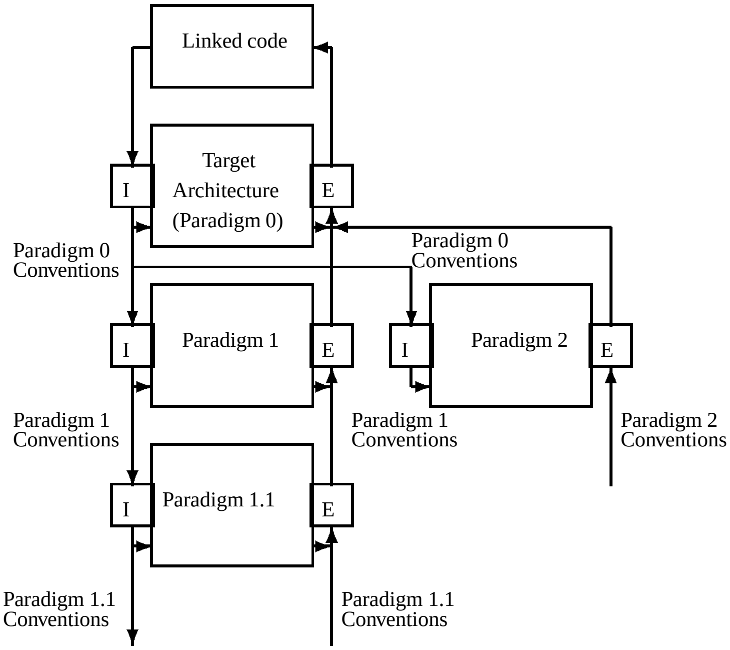

Figure 3: Paradigm inter-operation using call gates

same service must pass through its export gate. The call gates are design abstractions and not concrete implementation models. They can be implemented 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. 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 3 illustrates an example case. Assume that a module written in *paradigm 2* is using a facility implemented in *paradigm 1.1*. The module written in *paradigm 1.1* will export that facility (using the syntax and semantics appropriate to *paradigm 1.1*) to its superclass (*paradigm 1*) throught its export gate, thus converting it to the data types and calling conventions used by paradigm 1. *Paradigm 1* will again pass it through its export gate, converting it to the conventions used by *paradigm 0,* the target architecture. (For example the calling conventions of the Unix system, can include the passing of parameters through a stack frame, and the naming of identifiers with a prepended underscore.) In this form the facility will again be imported from the pool of linked code by *paradigm 1* and made available to its subclasses using its conventions. The facility can then be imported and used by *paradigm 2* which can understand the calling conventions of *paradigm 1*. Although during the path described the facility crossed three paradigm boundaries, in all cases the paradigm just needed to be able to map between its calling conventions and data types and those of its superclass.

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.

2.3 Multiparadigm Environment Generators

The object-oriented system structure described above, introduces the possibility of *multiparadigm environment generators*: systems that can be used in order to design and implement a multiparadigm programming environment. A multiparadigm environment generator can provide the following services:

- convert a system described by paradigm objects into a multiparadigm programming environment, and
- provide support for using existing tools.

The use of a multiparadigm environment generator eases the development task of multiparadigm programming environments by reducing the development time and implementation errors. Such systems may even be used to create specialised paradigms for one specific application. In this way a solid software engineering foundation for the concept of "little languages" [Ben88, pp. 83-100, 128-131] can be provided.

3 Prototype Implementation

In order to demonstrate our object-oriented approach to multiparadigm programming we designed and implemented three prototype systems. In the following sections we will describe *integrator*, an application written in a number of paradigms, *blueprint*, the multiparadigm programming environment used to implement the *integrator*, and MPSS, the object-based multiparadigm environment generator that was used to develop *blueprint*. The relationship of the three systems is illustrated in figure 4.

3.1 MPSS: an Object-based Multiparadigm Environment Generator

MPSS consists of a number of separate tools that aid the implementation task of a multiparadigm programming environment. This is consistent with the Unix philosophy of small individual tools that can be combined with each other [Rit84].

The design philosophy behind MPSS, is that of paradigm object classes, described in the previous section. Every programming paradigm forms a class, with the target architecture paradigm being the root of the class structure tree. For every paradigm, the implementor provides a paradigm class description file, that defines the variables and methods of the paradigm class. This is then compiled, by the paradigm description compiler provided by MPSS, into a compiler for that paradigm and its manual page. The multiparadigm programming environment (e.g. *blueprint*) user, can use that compiler, to convert source code from the given paradigm into object code. When the source code of all paradigms has been compiled a special link editor, the *multiparadigm link editor* can be invoked to link all the paradigm objects, and associated support libraries together into a runnable system. The link editor is also responsible for initialising the classes and their objects by calling the respective initialisation methods. Two additional tools aid the incorporation of existing compilers into the system by detecting and "protecting" global variables and functions. In this way compilers and code generators that use fixed names for global variables or functions can be incorporated into a multiparadigm programming environment.

3.2 *Blueprint*: a Class-structured Multiparadigm Programming Environment

Blueprint is a multiparadigm programming environment, built using the MPSS tools. Its name is derived from the acrostical spelling of the paradigms provided, [3] namely:

- **B**NF grammar descriptions (*bnf*),
- **l**azy higher order functions (*fun*),
- **u**nification and backtracking (*btrack*),
- **r**egular expressions (*regex*),
- **i**mperative constructs (*imper*) and,

[3] In order to find the name the Unix command "`grep b /usr/dict/words | grep u | grep l | grep r | grep i | grep t`" was executed. *Blueprint* was selected from the 29 words that matched the specification.

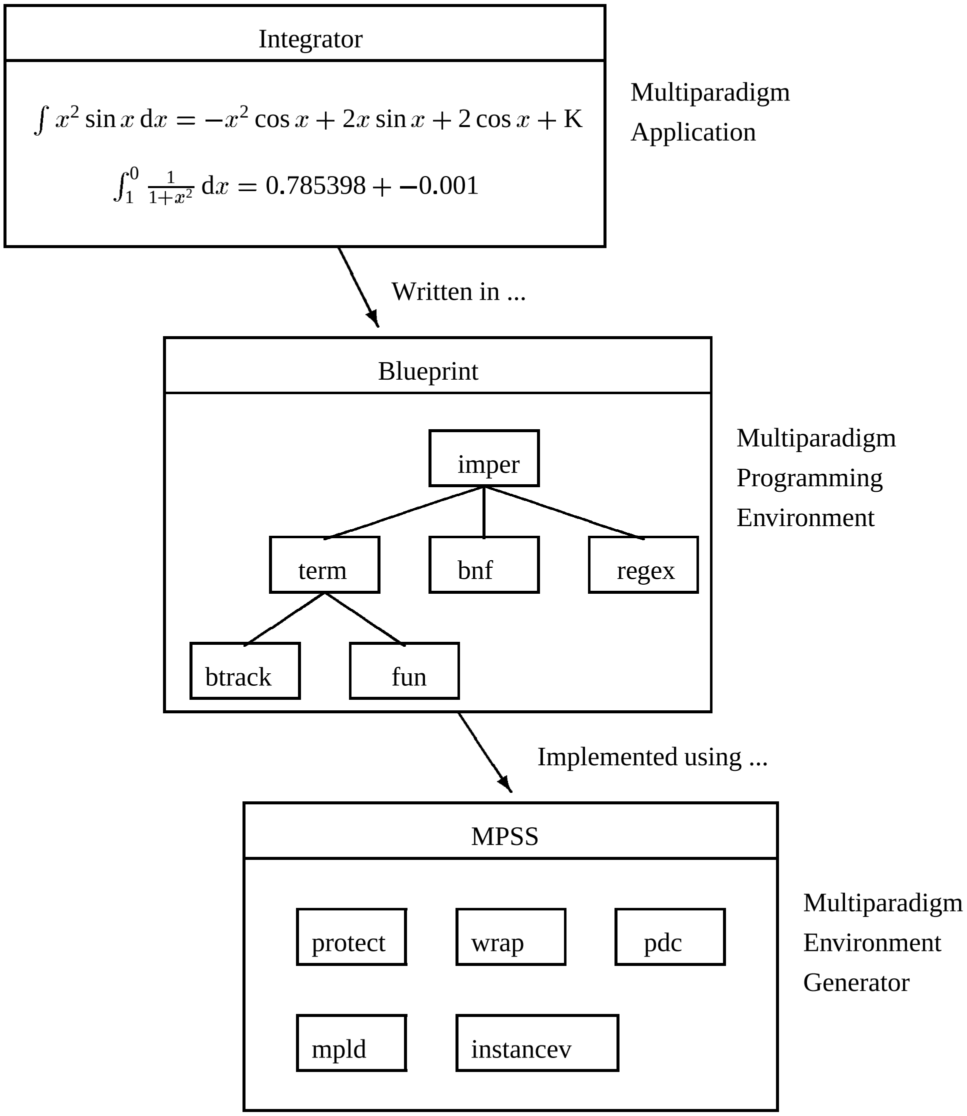

Figure 4: Entity-relationship diagram of the implemented systems

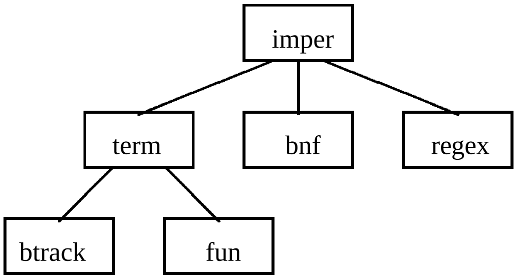

Figure 5: *Blueprint* class hierarchy

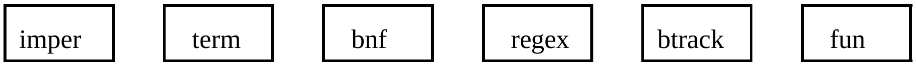

Figure 6: Programmer's view of *blueprint*

- **t**erm handling (*term*).

All these paradigms are provided in the form of individual paradigm compilers: tools that convert the code expressed in a given paradigm, into object code that can be linked and executed together with code from other paradigms. In the following sections we present the design of the system and its parts.

3.2.1 Design Objectives

Blueprint was designed as an experimental prototype system in order to prove the viability of the object-oriented structuring methodology. Therefore our design was centered around the following objectives:

- realisation of a wide variety of diverse programming paradigms, and implementation methods,
- provision of a non-trivial class hierarchy, including the abstraction of common characteristics in a special superclass,
- usage of all the features provided by MPSS,
- incorporation of existing tools,
- suitability for the implementation of a useful application, and
- ability to bootstrap the system in order to test and use it as much as possible.

3.2.2 System Structure

Blueprint is designed using the MPSS paradigm class hierarchy notion. The target paradigm is the imperative paradigm provided by the target architecture, which in our case are Sun SPARC computers. The paradigm classes that were implemented, can be seen in figure 5. Term expressions are the natural data objects, for both functional and logic languages; the provision of the *term* class is based on this observation and, in addition, provides a practical vehicle for their implementation.

It is important to note, that the tree structure is only used in order to design and implement the system using MPSS. The structure is transparent to a programmer using *blueprint* who is presented with a flat structure of all the paradigms (figure 6). In the following sections we briefly describe each *blueprint* paradigm.

3.2.3 Imperative Paradigm

Imper, the imperative paradigm is provided in the form of the C programming language [KR88]. It is the one closest to the target architecture, and the calling and naming conventions of the language are used as a common interface for the other paradigms.

3.2.4 BNF Grammar and Regular Expression Paradigms

The *bnf* (BNF-grammar) and *regex* (regular expression) paradigms are used to encapsulate *yacc* [Joh75] grammar descriptions and *lex* [Les75] lexical analyser specifications, as objects. The main advantage of this encapsulation is the ability to use more than a single grammar description or lexical analyser specification within the same project. This is achieved by "protecting" the global variable and function names that *yacc* and *lex* define by prepending them their object (module) name. Both paradigms were implemented by using the MPSS tools to create multiparadigm environment conformant compilers out of the standard Unix *yacc* and *lex* generators. Inter-operation with the imperative paradigm is achieved by using the standard *lex* and *yacc* interfacing conventions, as modified by the object encapsulation scheme.

3.2.5 Rule-rewrite Paradigm

Term, the term-based rule rewrite paradigm abstracts the notion of a *term* used by both the functional and logic programming paradigms. Its syntax resembles that of Prolog, but it uses a deterministic rule-rewrite execution model with predefined argument mode declarations, resembling the functionality provided by Strand [FT90]. *Term* is implemented in *term*, *imper*, *bnf*, and *regex* as a compiler that translates *term* into C. It was bootstrapped using the SB-Prolog compiler [Deb88], and a semi-automatic translation process. Inter-operation with the imperative paradigm is provided by documenting the compiled form of the *term* "predicates" and providing access and constructor functions for the term abstract data type, in its converted form of C *structures*.

3.2.6 Logic Programming Paradigm

Btrack, the logic programming paradigm provides the backtracking execution model, deep unification and syntax, associated with implementations of the Prolog programming language. It is implemented in *term* as an encoded token interpreter based on a *solve/unify* loop [Coh85, p. 1313]. The *btrack* to *term* token conversion is performed by "compiling" the *btrack* predicates into *term* rules. Inter-operation with *term* is achieved by defining the predicates that are exported using *term* signatures. The *btrack* compiler then creates the necessary interfaces and entry ports.

3.2.7 Functional Programming Paradigm

Fun, the functional programming paradigm offers lazy higher order functions supporting currying and call-by-name, normal-order evaluation. Its syntax resembles that of Miranda [Tur85] omitting the guard and pattern matching constructs. It is implemented in *bnf*, *lex*, and *term* with function evaluation provided by an *eval/apply* interpreter [FH88, p. 205–211], written in *term*. Inter-operation with *term* is provided by allowing the import of single result *term* rules and exporting functions as *term* rules with a single result. Calls to and from *fun* need to take into account and respect the *fun* data structuring conventions, which are documented as *term* constructors.

3.3 *Integrator*: an Exemplar Multiparadigm Application

We decided to use *blueprint* as the implementation vehicle for a program dealing with function integration. Lexical analysis of the functions is provided by *lex*, the parsing by *bnf*, the numerical integration by *fun*, and the symbolic integration by *btrack*. We also used *term* to simplify and print the results, and *imper* to provide a graphing capability by interfacing to the X-Window system.

Numeric integration was performed by constructing an infinite list of better and better approximations, and eliminating the error terms using a higher-order method described in [Hug90]. The lazy function evaluation and functional style of *fun* allowed us to express the algorithm succinctly using function composition.

Symbolic integration on the other hand depends on heuristics, that are easily expressed as non-deterministic *btrack* predicates. In the process of integration one typically needs to dwell deep into a solution path to decide whether a specific method can be applied or not (e.g. integration "by parts"). The backtracking nature of the *btrack* predicates made this expression natural and close to the problem domain.

Integrator was implemented in three days. Table 1 has a list of the modules and their size, and figure 7 provides the paradigm inter-operation call graph of the system. We believe that each part of the system was implemented in the most suited paradigm. It seems, that implementing the system in the *blueprint* multiparadigm environment resulted in a concise and thus inherently (though always relatively) correct and easily maintainable system. Writing the *integrator* in an imperative language would result in an order of magnitude larger system, while choosing a single declarative language as the implementation vehicle would still make the system at least twice as complicated.

Function	Paradigm	Module	Lines
Symbolic integration	btrack	sint.pb	127
Lexical analysis	regex	scan.pl	47
Expression parsing	bnf	parse.py	76
Numeric integration	fun	aint.pf	75
Interfacing	term	ui.pt	131
Graph creation	imper	main.c	51
Total	blueprint		507

Table 1: *Integrator* line count table

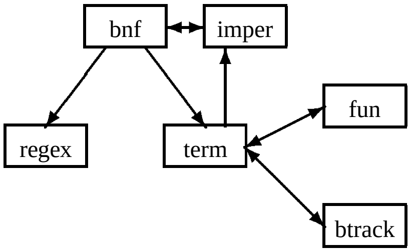

Figure 7: *Integrator* paradigm inter-operation call graph

4 Related Work

The usual approach to multiparadigm is to design languages that support many different paradigms [Pla92, Wel89, TOO86]. There are a number of languages that are based on this approach. Table 2 summarises the number of languages that are described in the literature supporting different paradigm combinations. Many of the problems described in section 2 are difficult to address in this way. Furthermore, the resulting systems are inflexible, and every system only solves the problem of combining the particular paradigms it supports. The combination of particular paradigms can however address important theoretical problems [Fro87, Han90].

Some of the multiparadigm programming languages support the object-oriented paradigm [TOO86, McC92, GM87], but do this as an additional paradigm, and not as a structuring mechanism for integrating other paradigms. In our approach object-orientation is used externally to the multiparadigm programming system in order to structure and encapsulate its components.

Finally, an approach that can be used for integrating arbitrary paradigms is described in [Zav89]. It concentrates however more on the aspects of validation and verification of the resulting system, and less on the structure of it.

Functional	●		●	●			●		●
Imperative		●	●			●	●		
Object-Oriented				●	●	●	●	●	
Logic	●	●			●		●	●	●
Distributed								●	
Constraint									●
Number of languages	23	10	9	8	10	7	5	4	5

Table 2: Number of languages found for the common paradigm combinations

5 Future Work and Conclusions

Applying object-oriented technology in the area of multiparadigm programming has given a number of promising results. Objects provide a powerful way for encapsulating and integrating modules written in different paradigms. The paradigm class hierarchy can be used to organise the implementation of a multiparadigm programming environment, and inheritance to decrease the implementationn effort. The approach can be used create multiparadigm environment generators: workbenches supporting the multiparadigm programming environment implementor.

In the previous sections we have outlined our approach to multiparadigm programming based on modeling programming paradigms as object classes. We described how objects and classes can be used to encapsulate program modules and their respective paradigms, how the paradigm class hierarchy can be used to abstract common paradigm characteristics, and how the *call gate* abstraction can be used to provide paradigm inter-operation and flatten the class hierarchy into a usable collection of paradigms. Furthermore we demonstrated how the object-oriented structuring mechanism provides the necessary base for designing multiparadigm environment generators. In order to demonstrate our approach we developed MPSS, a multiparadigm environment generator, used it to implement *blueprint*, a six paradigm programming environment, and used all its paradigms in a numerical and symbolic integration package. The advantages from using object classes as a structure for multiparadigm programming, exhibited themselves during all phases of the prototype development, and convinced us of the merits of this approach.

More work needs to be put into the semantics and type checking of multiparadigm systems based on object classes. Our structuring methodology can also be applied into other multiparadigm domains such as text processing, and can be used to model little "throw-away" languages and an associated software development process. We feel that many of the above mentioned goals are worth pursuing.

Acknowledgements

We would like to thank Bashar Nuseibeh for his helpful comments on earlier drafts of this paper. Support from the British Science and Engineering Research Council is gratefully acknowledged.

References

[Ben88] Jon Louis Bentley. *More Programming Pearls: Confessions of a Coder*. Addison-Wesley, 1988.

[Coh85] Jacques Cohen. Describing Prolog by its interpretation and compilation. *Communications of the ACM*, 28(12):1311–1324, December 1985.

[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.

[Deb88] Saumya K. Debray. *The SB-Prolog System, Version 3.0: A User Manual*. University of Arizona, Department of Computer Science, Tucson, AZ 85721, USA, September 1988.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[Fro87] Bertram Fronhöfer. PLANLOG: A language framework for the integration of procedural and logical programming. In John McDermott, editor, *IJCAI 87: Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 15–17, Milan, Italy, August 1987.

[FT90] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1990.

[GM87] Joseph A. Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[Han90] Michael Hanus. A functional and logic language with polymorphic types. In A. Miola, editor, *Design and Implementation of Symbolic Computation Systems: International Symposium DISCO'90 Proceedings*, pages 215–224, Capri, Italy, April 1990. Springer-Verlag. Lecture Notes in Computer Science 429.

[Hug90] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 2, pages 17–42. Addison-Wesley, 1990. Also appeared in the April 1989 issue of The Computer Journal.

[Joh75] Stephen C. Johnson. Yacc — yet another compiler-compiler. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, USA, July 1975.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[Les75] Michael E. Lesk. Lex — a lexical analyzer generator. Computer Science Technical Report 39, Bell Laboratories, Murray Hill, NJ, USA, October 1975.

[McC92] Francis G. McCabe. *Logic and Objects*. Prentice Hall, 1992.

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

[Nel91] Michael L. Nelson. An object-oriented tower of Babel. *OOPS Messenger*, 2(3):3–11, July 1991.

[Pla92] John Placer. Integrating destructive assignment and lazy evaluation in the multiparadigm language G-2. *ACM SIGPLAN Notices*, 27(2):65–74, February 1992.

[Rit84] Dennis M. Ritchie. Reflections on software research. *Communications of the ACM*, 27(8):758–760, 1984.

[TOO86] Ikuo Takeuchi, Hiroshi Okuno, and Nobuyasu Ohsato. A list processing language TAO with multiple programming paradigms. *New Generation Computing*, 4(4):401–444, 1986.

[Tur85] David A. Turner. Miranda — a non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 1–16, Nancy, France, September 1985. Springer-Verlag. Lecture Notes in Computer Science 201.

[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.

[Weg87] Peter Wegner. Dimensions of object-based language design. *ACM SIGPLAN Notices*, 22(12):168–182, December 1987. Special Issue: Object-Oriented Programming Systems, Languages and Applications, OOPSLA ’87 Conference Proceedings, October 4–8, Orlando, Florida, USA.

[Wel89] M. Wells. Multiparadigmatic programming in Modcap. *Journal of Object-Oriented Programming*, 1(5):53–60, January/February 1989.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, third edition, 1985.

[Zav89] Pamela Zave. A compositional approach to multiparadigm programming. *IEEE Software*, 6(5):15–25, September 1989.