# INTEGRATING DIFFERENT PROGRAMMING PARADIGMS

## 1 Programming Paradigms

Programming is bridging the gap between man and machine. Already since the early days of the computer, people have struggled with the question how to express problem solving solutions into computer understandable notations. Soon after the introduction of machine language and associated assembler the need arose to program in terms of particular application domains rather than in low-level computer instructions. As a result, in the 1950s and 1960s high level languages were developed for scientific work (Fortran, Algol), for business applications (Cobol), for symbol manipulations (Lisp), and for many other areas. The goal of all those languages was to bridge the gap between application domains and the computer.

In the 1970s and 1980s new ways of programming were developed by the introduction of different so-called programming paradigms. Each programming paradigm represented a set of principles and rules for writing programs. While the languages of the earlier decades were mainly procedural, new paradigms such as object-oriented, functional, and logic programming emerged. All these paradigms were developed because the procedural approach was felt to be inadequate for solving problems in new areas. Bridging conceptual gaps, which was earlier one of the driving forces for developing new languages, was now pushing for new paradigms.

These developments also started discussions about the differences between imperative languages and declarative languages.

Imperative languages express their programs as sequences of explicit steps. These steps may be instructions at the assembler level, statements in a high-level language, or commands for an operating system. They all imply an underlying machine with memory which is manipulated by individual steps. In the early days, almost all languages, with the exception of the first version of Lisp, were imperative languages.

In contrast, declarative languages describe what is to be computed rather than how the computation should proceed. How the computation is performed depends on the underlying implementation, which embodies imperative steps not found directly in the declarative language.

The distinction between imperative languages and declarative languages is very coarse and nowadays it is recognized that most languages are lying at a continuum between both extremes. Still it can be used as a global categorization of the different programming paradigms.

Within these two major categories several programming paradigms, representing different styles of programming, have emerged since the advent of high-level languages. Most languages are belonging to one of the following classes:

Procedural Languages
As said before, procedural languages were the first and are still most dominantly present. They introduced assignment statements, conditional statements, repetition statements, etc. Procedural languages are combining imperative programming facilities with abstraction mechanisms to build procedures and functions. Languages such as Fortran, Cobol, Pascal and C are examples of procedural languages.

Functional Languages
One of the first languages which deviated from the imperative style was Lisp. Conceived in the 1950s, it started with a pure mathematical approach, based on recursive functions. Its main application area was symbol manipulation, including algebraic manipulations of symbolic expressions, and proving theorems of mathematical logic. The original language, published in 1960 by John McCarthy, was based on five elementary functions operating on uni-directional lists. All other functions of the language could be expressed in these five elementary functions, which was a remarkable achievement at that time. Later on, many language features including imperative constructs were added for improving the efficiency or functionality of "pure Lisp", as the original version has been called. Since then, other functional languages, such as Miranda, Hope, Haskell, and others, have been introduced.

In general, functional languages are belonging to the category of declarative languages, and are based on the principles of mathematical functions. Functions are the major constructing elements of such languages. The result of a function is determined solely by the values of its arguments, so a function called with the same arguments will always yield the same result and will have no side-effects. One of the main motives for developing functional languages is that mathematical reasoning about programs is rather simple, because all functions are mathematical functions.

Logic Languages
The concepts of logic programming have been developed in the early 1970s by two groups: one under the direction of Alain Colmeraurer in Marseille, working on natural language processing, and the other group under the guidance of Robert Kowalski at the university of Edingburgh, exploring automatic theorem proving. The combined efforts resulted in the development of Prolog.

Logic programming is based on a particular form of mathematical logic known as the first-order predicate calculus. The term 'first-order' means that the logic cannot deal directly with statements about statements but only with statements about single objects. First-order logic allows quantification over individual objects; second-order logic further allows quantification over predicates.

Logic programming languages are belonging to the category of declarative languages. A logic program consists of a number of rules which are used to derive conclusions from a number of conditions. The logic programing paradigm introduced a number of new and interesting programming concepts. In order to derive proper conclusions from a set of rules, the underlying implementation uses unification to invoke a rule and uses backtracking in case a rule fails. Furthermore, a rule may produce a set of answers, not necessarily a single answer. The underlying implementation uses a search mechanism to obtain the answers.

Object-oriented Languages
The history of object-oriented programming starts with the development of the discrete event simulation language Simula 67 in Norway as early as 1967. Based on concepts found in Simula 67 the language Smalltalk was developed in the 1970s at Xerox Palo Alto Research Center in California. Later on, in the 1980s, other object-oriented languages such as C++, Eiffel, and Java were developed.

All these languages are using the concepts of objects and classes of objects. Classes may be organized in hierarchies allowing the use of inheritance to capture commonalities of sub-classes in super-classes. Most object-oriented languages are imperative languages. They describe the behavior of an object by means of methods which are associated with the class of the object. Interaction between objects is performed by means of messages.

Because of its roots in simulation, the same principles used in object-oriented programming can also be used for object-oriented analysis and object-oriented design. Furthermore, object-oriented databases is an important research area.

In the following table the most important features of these paradigms are listed:

| Procedural Programming | Object-Oriented Programming | Functional Programming | Logic Programming |
|---|---|---|---|
| • procedures<br>• assignments<br>• side-effects | • objects<br>• classes<br>• encapsulation<br>• inheritance<br>• polymorphism | • mathematical functions<br>• symbolic expressions<br>• lambda functions | • relations<br>• backtracking<br>• unification<br>• multiple answers<br>• reasoning |

## 2 Object Orientation

Because of the major impacts of the object-oriented paradigm we will discuss in more detail some important aspects. Object orientation has become a new major direction in software development in the past few years. It not only changed thoroughly the ideas about programming but it also affected others phases of the software life-cycle. Object-oriented analysis and design became a preferred way of approaching new applications.

Numerous books and articles have been published on the subject. In addition, many object-oriented products have appeared on the market. The suppliers of object-oriented languages, object-oriented user interfaces, object-oriented databases, and object-oriented operating systems are all trying to convince the users that the new approach is solving many problems, if not all.

Why has object orientation become so popular? To answer that question we have to go back into a little bit of history. In the early days of computing, programming was the most important activity in software development. Programming was bridging the gap between man and machine, and between problems and solutions. To reduce the gap, first the assembly language was invented, and then, later on, several high-level languages appeared to diminish the effort to write programs. The history of programming languages shows a progression from low-level computer-oriented constructs towards high-level application-oriented abstractions.

After the introduction of higher level languages it became clear that structured methods were needed to avoid spaghetti code and to write better maintainable programs by using different levels of abstractions. In the same period software engineers became aware of the fact that many more steps are required in software engineering. As a consequence, structured analysis and design were developed in the 1970s as a natural outgrowth of structured programming.

In the 1980s and 1990s the history was repeated by object-oriented programming followed by object-oriented analysis and design.

Object-oriented programming has its roots in the discrete event simulation language Simula 67 which was developed in Norway as early as 1967. Based on concepts found in Simula 67 the language Smalltalk was developed in the 1970s at Xerox Palo Alto Research Center in California. Later on, in the 1980s, other object-oriented languages such as C++ and Eiffel were developed.

All these languages are using the concepts of objects and classes of objects. Classes may be organized in hierarchies allowing the use of inheritance to capture commonalities of sub-classes in super-classes. Most object-oriented languages are imperative languages. They describe the behavior of an object by means of methods which are associated with the class of the object. Interaction between objects is performed by means of messages.

Because of its simulation background object orientation provides a connection between objects in the real world and objects in the software. It also forms the basis for later developments as object-oriented analysis and object-oriented design. The ability of object orientation to map real-world objects directly into corresponding software objects was one of the driving forces in the acceptance of object orientation.

## 2.1 Characteristics of Object Orientation

Object orientation has enriched the software discipline with a different way to view the world. While in conventional software engineering the main emphasis was on actions, object orientation made clear that actions and data are inseparable. This change in direction requires a different mind-set to analyze and to solve problems. Instead of concentrating on the question of what should be the sequence of actions to solve the problem, the new challenge is how to divide the solution space into a set of objects that could solve the problem. This requires a higher level of structuring capabilities of software developers than demanded by conventional methods.

In addition, object orientation introduced a whole new terminology. Objects, classes, methods, messages, instances, encapsulation, inheritance, polymorphism, dynamic binding are new terms which should be mastered by those who want to understand object orientation. Part of the terms are naming new concepts, another part is related to the metaphor of message passing between objects.

Because of the change in mind-set and the difference in terminology the new technology of object orientation requires a steep learning curve in particular for experienced programmers because the new direction means a break with the past and a considerable reorientation for the future.

## 2.2 Problems of Object Orientation

As we have seen, one of the claimed benefits of object orientation is that real world objects can easily be mapped onto corresponding software objects. Real-world objects may be real or abstract things, such as tables, chairs, books, drawings, customers, accounting records. They can be represented in the computer by related objects naming tables, chairs, and so on.

### What are Objects?

But there are also things which are not so easy to map. Is wind an object? And what about emotion, color, velocity, and weight? Some advocates of object orientation are stating: all things are objects. But that may be a confusing statement for a car designer, for he will agree that a car is an object, but in his perception color, velocity, and weight are different qualities. They may be characteristics of an object, but are certainly not objects. The confusion arises because we are confronted with two different languages. The car designer talks about a real world object and its characteristics, while the software designer talks about objects as software solutions for all kinds of real world concepts including object characteristics or attributes. This difference in terminology causes a lot of misunderstanding between domain specialists and software engineers because not all concepts used in the real world are objects.

### Relationships

Another problem are relations between objects. In the real world, objects are not living in isolation but are interacting with other objects. For example, a marriage is a mutual relationship between husband and wife. While husband and wife may be considered to be objects, in real life a marriage is of a different quality. Another example is ownership. For example, Peter owns a car, a house, a boat, and an airplane. Peter, car, house, boat and airplane are all examples of objects, but the ownership relations are not considered to be objects. As a consequence, relations between objects are often difficult to map on software objects because relations are not objects.

There are two different forms of relationships which can be recognized in many situations. These are the has-a relationship and the is-a relationship.

The has-a relationship describes object composition. For example, a car has an engine, a customer has a name. In daily life, the description of the relationship may take many forms such as "contains", "holds", "consists-of", and so on.

The is-a relationship describes a generalization relationship. For example, a car is a vehicle, a customer is a person.

### Inheritance

Because of historic reasons, object-oriented languages are explicitly supporting is-a relationships by means of inheritance, while the has-a relationships are not explicitly supported. As a consequence, inheritance has attracted much more attention in the literature than the has-a relationships. Many articles and many training hours for object-oriented programming have been devoted to inheritance and its proper use. However, because of the excessive focusing on inheritance other structuring capabilities of object orientation has become underexposed. As a consequence it creates the false impression with software developers that inheritance is the only way to structure the software.

Inheritance is a complex issue. From a conceptual point of view, it seems that inheritance is a rather simple generalization relationship which can easily be established. But if you try to use it in practice, inheritance appears to be a difficult concept. Take, as an example, the simple question: do squares form a subclass of rectangles? It appears that finding the answer is not easy (see Ryant (1997)).

## 2.3 Problems associated with Inheritance

There are also problems with inheritance at the technical side. Implementations of inheritance have been complicated by many factors. In almost all object-oriented languages inheritance, polymorphism, and dynamic binding are closely interconnected. This has the advantage that one mechanism can be used to support all three concepts, but it has the disadvantage that the interdependencies between the different features are causing a number of problems:

1. Because inheritance, polymorphism, and dynamic binding are coupled, inheritance is not only used to describe generalization relationships but is also used for quite a number of other purposes. As a result many variations of inheritance are recognized. For example, inheritance is used for extension because the properties of a child class may be extended with the properties of a parent class, but inheritance is also used for contraction because a child class may be more specialized (or restricted) than the parent class. One author (Budd (1997)) has counted 8 forms of inheritance, while another (Meyer (1996)) has discovered 12 kinds of inheritance grouped into 3 broad categories. For a software designer it is not always obvious which variation should be used.
2. Another area is the run-time behavior in case of inheritance. A message to an instance of a child class may be handled by a method of one of its ancestor classes. An ancestor class may be a parent class, a grandparent class, or any other ancestor class. Which class will be selected depends on the dynamic conditions of the program. Because of the interactions between inheritance, polymorphism, and dynamic binding on the one hand and the interrelations between ancestor classes and child classes on the other hand, the implications of a method lookup are difficult to follow. In particular, the dynamic interaction between methods of a super-class and its sub-classes may cause what is called by some authors the yo-yo problem where control flow alternates between methods on different levels. Programmers who are responsible for maintenance, have often a difficult time to understand the dynamic behavior of a program. They need to understand how messages are propagated along a class hierarchy, up and down, to discover where the processing is getting done. Experience shows that such tracing is time consuming and error prone.
3. Another problem is that inheritance may break encapsulation. With inheritance, the internals of parent classes are often visible to child classes. As a consequence, a child class can access and modify the instance variables of its parent class directly, thereby violating the principles of information hiding and encapsulation. If, later on, the implementation of a parent class must be changed, the child classes may also be forced to change. Because reuse of classes is one of the major goals of object orientation these implementation dependencies can cause problems when you're trying to reuse a child class which uses the internal data of a parent class. Should any feature of the inherited implementation not be appropriate for another application, the parent class must be rewritten or replaced by something more appropriate. This dependency limits reusability of classes.
4. Inheritance requires specialized language constructs and implementations. In object-oriented applications objects may send messages to other objects. Such a message may contain a number of arguments. The first argument of the message specifies the destination: the receiving object. It is treated in a special way because the underlying mechanism for handling inheritance, polymorphism, and dynamic binding is associated with the receiving object as specified by the first argument. As a consequence, symmetric treatment of all arguments in a message is not possible. This is particularly disturbing in cases where arguments have equal rights. Examples are binary operations, and relationships between equivalent partners. It also excludes the use of pattern matching for the first argument.

## 2.4 Alternatives to Inheritance

Because of the complexities of inheritance many researchers have tried to simplify or to avoid inheritance. From the many alternatives which have been investigated two approaches have appeared to be the most effective. These alternatives, delegation and decomposition, are both based on object composition.

Object composition means that an object is composed of other objects which implies that such an object is able to make use of the other objects. Thus, a child object may use a parent object. One may say that the has-a relationship and the is-a relationship, as discussed earlier are both be implemented by means of a using relationship.

Object composition requires that the objects being composed have well-defined interfaces. No internal details of the referred objects are visible. The differences between inheritance and object composition have been investigated by a number of authors (Budd (1997), Gamma (1995)).

Delegation is one of the alternatives to inheritance. In delegation, two objects are involved in handling a request; a receiving object delegates operations to another object, its delegate. This is comparable with child classes deferring requests to parent classes.

Another approach is to use decomposition. With decomposition, elements of an object are made accessible to the user. That means that a reference to a parent object in a child object can be made available to the user of the child object by means of an accessor operation.

While delegation and decomposition are in many cases alternatives of inheritance, the mechanisms used inheritance are still required in object-oriented languages for polymorphism and dynamic binding.

### 2.5 Well-defined semantic model is missing

One of the main reasons of the problems with object-oriented languages is that there is not a well-defined semantic model for object-orientation. There are only several losely defined implementation models for the different languages. Most problems as we discussed are so interrelated that many attempts to remedy the flaws in one of the implementation models introduces other problems. Some people are therefore stating that objects are pass because further developments are stuck in a blind alley. However, this may be too pessimistic. In the following chapters we will discuss how elements of object-orientation can be used in Domain Orientation.

## 3 Integration of Different Paradigms

All these paradigms have proved to be useful in formulating solutions in particular problem domains. Each paradigm favors specific formulations which often result in compact and clear programs.

Many applications contain a number of (sub-)problems which could successfully be solved by a specific paradigm. However, in a single-paradigm language problems can only be worked out within the framework of that paradigm. In many situations the solution to a problem has to be distorted to fit the single paradigm.

If we had a multi-paradigm language then each paradigm could be applied where most appropriate. For example, procedural programming could be used to express sequences of imperative steps, object-oriented programming could be applied to reflect objects in the problem domain, logic programming could be used to describe the relations between those objects, and functional programming could be applied to express mathematical functions applicable to those objects.

Integration of different paradigms may also create new possibilities of solving problems and may open new ways of programming. It may create synergistic effects where the power of an integrated language is larger than the sum of the different paradigms. For example, logic programming applied to objects may open new ways of expressing solutions in terms of the problem domain. One of the most important means to reduce the gap between man and machine, and applications and computers, is to express a solution in terms of its problem domain and that may require the exploitation of different programming paradigms.

In the past, many attempts have been made to integrate two or more paradigms into one single language. Some were successful, such as the integration of object-orientation and Lisp in CLOS (Common Lisp Object System), but many other attempts failed.

There are various obstacles on the way of integrating different paradigms into one single linguistic framework. The main cause is that a number of paradigms are based on mutually conflicting characteristics. For example, assignment statements are essential for the procedural and object-oriented paradigms, but are forbidden in the functional and logic paradigms. Or, as another example, functional and logic are based on a declarative way of programming, while procedural and object-oriented programs have an imperative structure. In general, features needed by one paradigm may conflict with the requirements of another paradigm.

Because incompatible requirements cannot be satisfied by a single and coherent linguistic framework without accepting flaws in the language, one has to find ways to remove those incompatibilities. One way is to extend a given language with a number of features from another paradigm, as has been done in the cases of CLOS, and C++. Another way is to define a new language as exemplified by Eiffel and Java. However, both approaches will require compromises in language design to remove inconsistencies.

## 4 Execution Models

One of the main obstacles in designing a coherent linguistic framework for an integrated language are the underlying execution models as dictated by the different paradigms. For example, the object-oriented programming paradigm assumes an imperative, single-value execution model with implied facilities for inheritance and polymorphism while the logic programming paradigm assumes a declarative, multi-value execution model with implied facilities for unification and backtracking.

Because, for several reasons, a programming language should be based on one single execution model, an execution model for an integrated language has to be found which could fulfill the stated requirements and which could support the different language concepts. To define such an execution model, all essential features of the different paradigms have to be analyzed and mapped, where possible, on a single execution model.

An important requirement for an execution model is that it must be simple and easy to understand. Complex execution models are difficult to program because of intricate interactions. Reasoning about programs written for complex execution models is often hard. The same applies to debugging.

Another essential factor in designing an execution model is how efficient the different language elements can be implemented, because the expressive power of a language is one thing, but efficient execution is another important property. Significant performance questions had to be considered, such as: Can a language feature be mapped directly on an underlying machine or are run-time routines necessary? Where possible, for Elisa the choise was made in favor of compilable features with a minimum of run-time support.

Because of the need of a single and efficient execution model, not all properties of a paradigm could be retained in their original form. Some language features had to be dropped or modified in order to be supported by the underlying execution model.

We will give some examples of the kinds of considerations and the decisions we took for Elisa.

In case of the assignment statement some investigations made clear that an assignment statement can be quite useful in the body of a pure function, as long as side-effects are avoided. This brought us to the pragmatic decision to accept assignment statements in the language, although within the functional and logic paradigms they should be used with care.

Regarding the imperative and declarative controversy, it was decided to favor the declarative style of programming without excluding imperative constructs.

Unification and backtracking are important elements of logic programming. Because unification is a complex and expensive mechanism, it was decided to replace it by the simpler mechanism of pattern matching which is sufficient in many circumstances. Backtracking, however, is needed to handle multiple answers and can also be useful in many other situations, thus we decided to include it in the language in such a way that it can be applied selectively.

Inheritance, polymorphism, and dynamic binding are strongly coupled in object-oriented languages. Because of their impact on the execution model and on the language it was decided to decouple them in the language and to replace the inheritance implementation model by facilities for delegation and decomposition.

By using delegation and decomposition instead of inheritance a large number of the problems associated with inheritance are disappearing. Furthermore, polymorphism and dynamic binding can be supported by orthogonal language facilities.

As a result of this approach most of the required features could be incorporated in the language. In addition, some new features have been added. By combining the best concepts of different paradigms and by compensating for the missing features the Elisa language is a programming language with an own and specific character, as will be made clear in the following chapters.

## 5 References

Many references can be found in the bibliography.

About the criticisms of object-orientation interesting references are:

Al Davis (1998a), former editor-in-chief of IEEE Software wrote "We are now witnessing the fall of the Object era" (IEEE Software July/Aug 1998, pp.6-9). In his final interview (1998b) he says, "When I started consulting ten years ago, all my customers wanted to hear the word 'object'. Now, none do, and I find that clients are much happier when they don't. I think 'object' has now gone the way of 'structured'." (IEEE Software Nov/Dec 1998, pp.18-22).

Les Hatton(1998) states in "Does OO sync with how we think" (IEEE Software May/June 1998, pp.46-54), that object orientation is an imperfect paradigm for reliable coding.

Bertrand Meyer(1999) defends Object Technology against the attacks of the opponents in "A Really Good Idea". (IEEE Computer Dec 1999, pp.144-147). One of his conclusions is that most objections are related to C++.

Ivan Ryant(1997) "Why Inheritance means extra trouble", CACM October 1997, pp.118-119.

Taenzer (1989), Ganti, and Podar are describing the yo-yo problem.

There is a wealth of literature about programming languages and programming paradigms.

For example, Prolog as a representative of logic programming is described by Bratko (1990), Clocksin and Mellish (1984), Sterling and Shapiro (1986), and many others. Kowalsky(1980) showed how logic programming can be used for problem solving.

Object-oriented programming and Eiffel are discussed by Meyer (1988). Smalltalk is described by Goldberg and Robson(1983). C++ is defined by Stroustrup(1993). Many contributions about object-oriented programming can also be found in the yearly OOPSLA proceedings since 1986.

Object-oriented analysis and object-oriented design are discussed by Coad and Yourdon (1990, 1991). Object-oriented design with applications is described by Booch(1991). Object-oriented modeling and design are discussed by Rumbaugh(1991) and others.

Functional programming is discussed by Backus (1978), by Henderson(1980), by Darlington, Henderson, and Turner (1982), and by Bird and Wadler (1988).

Lisp, is described by McCarthy(1960), Winston and Horn(1984), Steel (1984), and many others. Applications of Lisp are also described by Abelson and Sussman(1985), and by Charniak and McDermott(1985).

DeGroot and Lindstrom (1986) contains a number of interesting papers about the unification of logic and functional languages.

Bobrow and others (1986) are describing how to merge Lisp and object-oriented programming. Moon (1989) describes the CLOS system. Gabriel, White, and Bobrow (1991) are discussing how object-oriented and functional programming are integrated in CLOS.

Petre and Winder (1990) classified different programming languages according to their ratio of imperative and declarative aspects.

Graham (1991) and Wegner(1990) are discussing the integration of different paradigms.

Placer (1988) described a multi-paradigm language and discussed the advantages of a multi-paradigm approach (1991).

Moss (1994) describes Prolog ++, which combines object-oriented programming and logic programming.

Budd (1995) describes Leda, which is one of the first languages designed for multiparadigm programming.

Budd(1997) describes and compares various features of different object-oriented languages.

Of the many papers about the problems of inheritance, the papers of Snyder(1986,1987) are worthwhile to study. The differences between inheritance and object composition have been described by Gamma (1995) and Budd (1997). Budd (1997) distinguishes 8 forms of inheritance. Meyer (1996) found 12 kinds of inheritance grouped into 3 broad categories.

Delegation was first described by Lieberman (1986). Later on, the term delegation was redefined by Wegner (1987).

Wilde(1993), Matthews, and Huitt are discussing maintenance problems of object-oriented software.

Many contributions about object orientation can also be found in the yearly OOPSLA proceedings (since 1986), and in the Journal of Object-Oriented Programming (since 1988).

Back | Home | Up | Next

Send me your comments

This page was last modified on 27-09-2012 12:11:26