INGI
Département d'ingénierie informatique

Concepts First in Introductory CS Courses
Working Group Report ITiCSE 2003

Juris Reinfelds, Peter Van Roy, Joe Bergin,
Jonathan Bredin, Richard Rasala, Kirk Scott

Research Report 2003-08
October 9, 2003

Faculté des sciences appliquées

UCL Université catholique de Louvain

Concepts First in Introductory CS Courses
Working Group Report ITiCSE 2003

Juris Reinfelds (co-chair)
New Mexico State University, USA
juris@nmsu.edu

Peter Van Roy (co-chair)
Université catholique de Louvain, Belgium
pvr@info.ucl.ac.be

Joe Bergin
Pace University, USA
berginf@pace.edu

Jonathan Bredin
Colorado College, USA
jbredin@coloradocollege.edu

Richard Rasala
Northeastern University, USA
rasala@ccs.neu.edu

Kirk Scott
University of Alaska, USA
kirk_scott@yahoo.com

ABSTRACT
Multi-language, multi-thread, multi-paradigm, net-centric programming is becoming widely used. Our teaching of programming has to adapt to the requirements of these new directions. How can we do that without an explosion in required course-hours?

This WG explores a *concepts first* approach to introductory programming courses that attempts to describe important ideas not simply in terms of a particular programming language but rather in terms that will permit the student to gracefully work with multiple programming paradigms. The paradigms appear naturally depending on which concepts are used for the problem being solved. The student is able to situate the paradigms in a more general framework that shows their relationships and how to use them together.

We discuss one way in which *concepts first* may be taught based on the kernel language hierarchy and its implementation as a subset of the programming language Oz [10]. We also discuss how *concepts first* may be introduced in situations where Java or a similar OO language is the base language. We comment on the impact of *concepts first* on existing courses and problem solving methodology.

We are proposing here an additional ingredient in freshman teaching, namely, language-independent concept descriptions. This approach has yet to be tried at the freshman level. Hence, we have no evidence that what we propose will make a difference. Nevertheless, we are hopeful that it will make a difference.

Categories and Subject Descriptors
D.1. Programming Techniques. D.3. Programming Languages.

Keywords
Concepts first; programmer's theory of programming; kernel language approach; multi-language, multi-thread, multi-paradigm programming; encapsulation, orthogonality, polymorphism, and abstraction; paradigms; toolkits.

1. INTRODUCTION
The working group, although small in number, was blessed with a rich set of strong viewpoints from an almost orthogonal variety of backgrounds. Therefore it first engaged in a vigorous, extended brainstorming phase both electronically prior to the conference and in person at the working group sessions.

From day one we made strong and valid points to substantiate our positions, but it took some time before we became willing and able to listen to the viewpoints of others that were equally strong and valid, but not necessarily in complete agreement with our own views. This report reflects both those ideas that we came to agree upon and some of the divergences of opinion that remain.

The original call for participation in the WG was posted on the ITiCSE'2003 website as follows:

> Byte-code has enabled ubiquitous multi-platform programming. Dot NET and its Open Source implementation MONO are enabling multi language, multi-paradigm code. Our teaching of programming has to adapt to the requirements of these new directions, but how to do it without an explosion of required course hours?
>
> Programming languages dwell on syntactic and semantic differences that distinguish one language from the others. Programming theory extracts and studies the concepts and mechanisms that are common to programming languages. A Programmer's Theory of Programming deals with concepts that programmers use to reason about programs. An

> approach through a Programmer's Theory of Programming is one way that an introduction to multi-language, multi-paradigm, distributed computing can be fitted into the class-hours currently allocated to CS-1 and CS-2.
>
> This Working Group will explore and define a concepts-first approach to introductory programming, compare it to the current approaches and contrast it with other possible approaches that are intended to equip CS graduates to better deal with the programming needs of tomorrow.

Although the invitation to the WG explicitly stated that the concepts for the *concepts first* approach should be chosen with a specific purpose in mind, WG members could not resist the temptation to make an attempt to produce a list of all concepts that are relevant to programming and problem solving. However, since no one could come up with a taxonomy of the concepts of programming and problem solving, this approach was abandoned by most members of the WG when the ad hoc list reached several pages in length without any sense of completeness.

The "natural" next step was to determine a short list of "the most important concepts". To do this, only two members of the WG addressed the challenge issued in the WG invitation, the rest of the members did not. *Abstraction* and *encapsulation*, perhaps the most fundamental concepts of problem solving, as well as *objects*, *inheritance,* and *polymorphism*, the most interesting concepts of the currently dominant object-orientation paradigm, emerged as clear winners. Although important in their own right, these concepts do not directly show us a way to include multi-language, multi-thread, multi-paradigm, net-centric programming into introductory programming courses which was the challenge issued by the invitation to the WG.

As we prepared for the WG, we kept in mind the strong warnings about the current state of introductory computer science education that were issued by Niklaus Wirth in the ITiCSE 2002 Keynote [20] and David Gries in SIGCSE Bulletin [5]. It turned out that equally strong warnings were made by Don Knuth [8] in the ITiCSE 2003 Keynote. It is worth extracting a few quotations from each of these well-known computer scientists.

In his keynote *Computer Science Education: The Road Not Taken*, Wirth deplores the complexity of current languages such as C, C++, and Java and the baroque nature of many of the details of these languages. He then criticizes the current introductory textbooks as merely attempting to describe what is a mess to start with rather than attempting to explain clear and solid principles. He then goes on to state what he believes should be the focus of introductory education.

> Considering our own subject, the field of computing and programming, an academic institution's ultimate goal must be much wider than the mastery of a language. It must be nothing less than the art of designing artifacts to solve intricate problems. Some call it the art of constructive thinking. It is in this context that the availability of an appropriate tool, a properly designed language, is of importance. It assumes the role of a theory on which we base our methods. How can anyone learn to design properly and effectively, if the formalism, the foundation, is an overwhelming, inscrutable mess? How can one learn such an art without master examples worth studying and following? Surely, some people are more, some less gifted for good design, but nevertheless, the proper teaching, tools, and examples play a dominant role.

After further discussion, Wirth imagines the characteristics of an exemplary textbook:

1. It starts with a succinct introduction into the basic notions of program design.
2. It uses a concise, formal notation. This notation is rigorously defined in a report of no more than some 20 pages.
3. Based on this notation, the basic concepts of iteration, recursion, assertion, and invariant are introduced.
4. A central topic is the structuring of statements and typing of data.
5. This is followed by the notions of information hiding, modularization, and interface design practiced on exemplary applications.
6. The book establishes a terminology that is both intuitive and precisely defined.
7. The book is of moderate size.

We have quoted Wirth's wish list at length because its ideas occur throughout this working group report.

The article *Where is Programming Methodology These Days?* by Gries [5] emphasizes the conscious application of principles to the problem solving process. He states:

> In any case, teaching programming as a skill means more than teaching facts. It means teaching students how to think when designing, developing, testing, and debugging a program. It means extending their problem-solving abilities. It means giving them effective strategies and principles that will shorten programming time and reduce the need for debugging (but not for testing). It means teaching good thought habits. In addition, it means teaching basic theory that provides understanding and that they can put into practice.

Although Gries does not advocate the development of new formal notations as does Wirth, Gries does advocate a focus on the fundamental organizing concepts and strategies and is thus quite in agreement with Wirth.

In his keynote entitled *Bottom Up Education*, Knuth is also critical of computer science education. Although the full text of this address is not yet available, here is the opening portion of his abstract:

> People who discover the power and beauty of high-level, abstract ideas often make the mistake of believing that concrete ideas at lower levels are relatively worthless and might as well be forgotten. The speaker will argue that, on the contrary, the best computer scientists are thoroughly grounded in basic concepts of how computers actually work, and indeed the essence of computer science is an ability to understand many levels of abstraction simultaneously.

In the keynote, Knuth then went on to explain his MMIX simulation of a RISC machine as a tool for computer science educators to explain *how computers actually work*. Although

MMIX is not relevant to this working group report, Knuth's more general advice that understanding abstractions is always based on intellectually integrating the more concrete levels that underlie these abstractions is important as we propose a *concepts first* orientation to the introductory curriculum.

This working group report consists of two fairly distinct parts. The first part addresses the challenge of the WG invitation directly. The second part raises issues that might impact the programming component of introductory CS courses and either help or hinder early introduction of multi-thread, multi-paradigm programming, based on a *concepts first* approach.

2. WHAT CONCEPTS ?

The challenge here is to avoid a union of the concepts of Imperative First, Functional First, Logic First, Objects First, and so on. Programming languages and programming paradigms justify their existence by emphasizing how they differ from other languages and paradigms. This leads to the erroneous, but widely held, view that there is little that these paradigms have in common and that what there is in common is not worth talking about.

In this section we present two approaches that look for what is common to programming languages and paradigms. We will see that there is so much that is common or only slightly different that the teaching of multi-thread, multi-paradigm programming becomes feasible without any significant increase in the class-time that is currently allocated to the teaching of single-thread, one-paradigm programming.

Both approaches lead to the same Kernel Languages of Programming of Van Roy & Haridi [18] that capture the essence of all major paradigms as well as distributed and parallel programming. The first approach starts with a small functional language, the core Kernel Language, and a set of functionally oriented concepts. This approach is well suited for students who already have had some exposure to functional programming or to the mathematics that goes with functional programming.

The second approach critically examines and redefines the most basic concepts of programming, such as *value, type, variable, expression, statement, assignment statement, procedure, function, thread, program* and a few others to create a Programmer's Theory of Programming that provides a beginner's path to the same Kernel Language that is introduced in the first approach.

2.1 The Kernel Language Approach

The kernel language approach came out of research into programming language concepts. We observed programming as a splintered discipline, divided into programming paradigms, and taught with little rigor (more like a craft than a discipline with an underlying scientific theory). The basic research question was to find the unity in this diversity and to put it on a solid scientific foundation. The focused attack on this question started in the late 1980s, and was led by people like Hassan Aït-Kaci [2], Gert Smolka [6], and Seif Haridi [7] and their students and colleagues. In 1999 we suddenly realized that we had made enough progress to teach programming in this way. We set about distilling our results in a textbook and teaching with it [19].

The basic idea of the kernel language approach is to translate practical languages (with all their rich expressiveness) into simple kernel languages. Widely different languages and programming paradigms can be translated into closely related kernel languages. The next step is to organize the kernel languages into a sequence, such that successive kernel languages differ only in one concept. For example, it is well known that object-oriented programming can be considered as a set of programming abstractions built on functional programming with state. We have proposed an organization principle (the "creative extension principle") that allows us to decide when and what concepts to add to a kernel language. With this principle we have organized several dozen programming concepts into a coherent progression, covering all well-known programming paradigms.

Based on this research, the textbook gives a comprehensive view of the whole discipline of programming. It is therefore not targeted for first-year courses. We have used it successfully from second-year to graduate level courses. It is an interesting question whether the approach can be adapted to first-year courses. The main questions are what subset of concepts should be chosen and in what order they should be introduced. Several authors have proposed answers to these choices.

We see three obvious ways to organize the concepts for a first-year course. The first possibility is exemplified by Abelson & Sussman [1] and Felleisen et al [4], who both propose to start with a small functional language and to extend this language with state and/or objects later in the course. A second possibility that has been used with success is to start with the basic concepts of object-orientation (encapsulation, polymorphism, state, and inheritance) so that the student can quickly get started in modeling real-world problems. Later in the course, these concepts can be defined precisely in terms of an underlying functional language with state. A third possibility is to start with active objects that send each other messages. This would bring the concept of concurrency to the forefront. We believe that this approach can be used successfully if an appropriate concurrency-oriented programming language is used.

2.2 A Programmer's Theory of Programming

Programming languages come and go, programming paradigms fall in and out of favor. A programmer's theory of programming provides a stable framework of concepts that programmers may use when they design programs or reason about programs. Research work is under way [13] that studies the partially overlapping programming concepts of various programming languages and then extends or redefines these basic programming concepts so that they can cover as much of the programming language landscape as possible. In this way, beginning students can get equally ready for subsequent specialization in any programming language or paradigm.

The development of programmer's theories of programming have already had successful applications to the conceptual simplification of introductory as well as advanced CS courses. Edsger W Dijkstra introduced the first Programmer's Theory of Programming in the late sixties [3]. Dijkstra's theory was concise, precise and elegant, but it only applied to imperative programming.

The Kernel Language approach of Van Roy and Haridi [19], includes all major paradigms including distributed and parallel computations and concepts of internet-wide computations. The Kernel Language approach, initiated through a small functional core kernel language, has been successfully class tested in a number of courses at the second year and higher level in several universities [18], [14].

For a surprising glimpse at how many new insights can be found and old misconceptions laid to rest by an examination and redefinition of the most basic programming concepts, the ones that most programming textbooks gloss over, let us discuss a couple of these concepts.

2.2.1 The Programming Concept "value"

When we say "value", the typical response is *int* or *float*, or *"a string of characters"*. Let us extend the definition of the concept of "value" to:

A value is any piece of information
that is capable of encapsulation.

The extended definition includes much more than *int* or *float*. The extended definition includes functions, procedures, records, classes, objects, and methods as well as integers, floats and strings. Classifying all these as values suggests that, ideally, they should all follow similar usage rules. Today, usage rules differ widely, especially for procedures, functions, classes, objects, and methods, but this is largely because compiler writers could not efficiently handle procedures as first class values for a long time and language designers have not yet woken up to the fact that today that restriction is no longer necessary.

With the expanded definition of a "value", exposure to a new programming language or paradigm will raise the question: "How are the different types of values treated here, compared to what I used before?". The student will not be shocked or surprised if the new paradigm treats procedures as values that can be assigned to variables and passed as parameters to other procedures. Instead, the student might think: "If this language lets me introduce the source code text of procedures or functions into the program in the same way as it allows me to introduce the source code text of integers, then this language will be easier to learn and use." In addition, the student will cease to think of any particular language as the standard and will instead begin to think of language evolution as just as natural as the evolution of individual programs albeit on a longer time frame.

2.2.2 The Programming concept "variable"

From the programmer's point of view, a **variable**

- has a **scope** that determines where in the program this variable is valid.
- may meaningfully exist unbound to any value or may be bound to a value via some reference mechanism.
- .has a **name** with which programmers refer to it.

For a long time we have claimed that imperative languages have assignments to variables and functional languages do not. This is actually not quite true, as a precise and somewhat more general definition of the assignment shows.

An assignment statement links a variable to a value.

This definition includes all paradigms. All programming languages have variables. Variables may be linked to values or may be unbound. The difference between paradigms is in the linking rules. For example, in imperative languages variables may be linked to values in any place in the program, any number of times. In functional languages a variable may be linked to a value exactly once. In logical programming, variables may be re-linked to values at specific points (choice points) in the program.

2.2.3 Procedures as Values

We have no problem with the source-code description of an integer by a sequence of ASCII characters that may appear whenever there is an *"expression situation"* in a program where an integer value is needed.

Similarly, we should expect that future compilers will allow us to place the source code of a procedure, which is also a sequence of ASCII characters, in any *expression situation* in the program, where a procedure value is required. The typical *expression situation* where a procedure-value is used is on the right hand side of an assignment statement that assigns a variable to the procedure. This sounds strange only because most programming languages introduce a procedure definition via a special construct called *declaration* that automatically binds a variable to the procedure and discourages or prevents redefinition. To refer to a procedure in a different way from its declared name in such a language or to pass such a procedure as an argument, we need to introduce either the concept of pointers or to learn about "strategy patterns" that perform the encapsulation of procedures as first class objects. In either case, we need a complicated implementation mechanism for an idea that is actually relatively simple.

Our approach provides a gentle introduction to the use of procedures and functions as values. Higher order programming and other "heavy duty" concepts of functional programming can be introduced later.

2.2.4 Programming Aspects of OO concepts

The OO paradigm offers a rich set of problem solving concepts. For example, objects are model entities of the problem area and methods define behaviours that these entities should exhibit. Of course, one can also view objects as bundles of functional behavior whose specific actions are influenced by the internal data of the object.

From the programmer's point of view, an object has a global existence that transcends the particular place in the program where the object was created. An object may be passed as an argument and returned as a value. Once created, an object can exist for the duration of the execution of a program. Of course, for storage efficiency, objects that cannot be accessed by any part of the running program are deleted via garbage collection.

It is common though not always essential to link objects to variables. These variables have the same block-based scope of definition as all the other named entities of the program, so if a programmer asks the question: "Where can I use a variable that is bound to an object?" the answer is "According to the same block-scope rules as for any other entity that is bound to a value." It is important to recognize that the scope of the object which is global is not the same as the scope of any particular variable that may be used to refer to the object.

If a programmer asks the question: "What methods are available for this object?" the block-based scope does not apply, but an extension of the scope concept does. Whenever it is necessary to decide whether an object may call a particular method, the programmer has to look at the scope of method names of the class from which that object was constructed. In this sense, the scope-concept extends to method calls.

With inheritance (both explicit and anonymous), the number of available methods increases because the methods of the superclasses are available also. The basic idea is simple, but current languages complicate this issue with ad hoc rules as to what method-names are actually available, which ones are blocked in certain situations, and so on. Note that the type of an object is determined by its own construction not by the type of any particular variable that refers to the object. Object-oriented languages normally permit an object of a derived class to be referenced by a variable with the type of a base class while maintaining the object identity. In this situation, if the object is asked to execute a method via a call using the base type variable, then the normal object-oriented response is to dynamically execute the corresponding method of the derived class rather than any related method of the base class. This is part of what is called dynamic polymorphism. Unfortunately, some so-called object-oriented languages really muddy the waters by defining situations in which the base class method will be called rather than the derived class method. Such language designs make pedagogy extremely difficult not to mention the problems that arise in practical use of such languages in industry.

3. ISSUES OF CONCERN

In this section we discuss a number of issues that positively or negatively impact a *concepts first* approach to introductory programming courses. We examine the most important problem solving concepts (encapsulation, orthogonality, polymorphism, and abstraction) and we consider the role of programming paradigms. Then we examine how we may apply the theory of programming to large, messy languages such as Java. We argue that the use of toolkits such as the Java Power Tools [11] can provide a bridge that illustrates how the simple concepts may be expressed within the constraints of an existing commercial language. Finally, we consider some of the reservations expressed within the working group about how the *concepts first* approach can work within a language such as Java.

3.1 Encapsulation, Orthogonality, and Polymorphism

Understanding of the fundamental elements of the programmer's theory of programming will be of little value without a vision of how these elements can and should be composed to form a complete program that is organized in a coherent fashion and capable of being extended in the future. Although it is not easy to solve problems using programs, it is even more difficult to solve problems in a fashion that permits these programs to be adapted cleanly to handle future problems or additional requirements. The essence of program design is to create a program that is so well organized that adaptations are straightforward.

Three key concepts are central to quality program design: encapsulation, orthogonality, and polymorphism. The practical usefulness of a programming language is often determined by the degree to which these concepts are supported. The coherence of a program or family of related programs is likewise determined by the degree to which these concepts are utilized.

Encapsulation involves capturing some aspect of programming knowledge into an entity that is named and reusable. Here are some simple object-oriented examples:

1) Collecting data elements into an object.
2) Collecting the operations on a set of data elements into an object.
3) Defining a method or function to perform some task based on its own internal state and on its parameters. This function may or may not return a value for further processing as desired.
4) Collecting a set of related behaviors into an object and providing internal parametric data to support these behaviors.
5) Defining the contract required of a given set of related objects or related behaviors.
6) Abstracting a common set of behaviors into some entity such as an interface or an abstract class that may be used to specify more concrete entities.

Programming languages vary as to what forms of encapsulation are directly supported and how well they are supported. The best form of encapsulation will create a first class entity that may be saved, assigned, and passed as a parameter to other entities. The reason "first class" is so useful is that first class entities enable *substitution*, that is, one first class entity may be replaced by a similar first class entity with almost no effort. To take Java as a specific example, items 1, 2, 4, and 6 provide first class encapsulations. Methods and functions (item 3) are definable via declarations but are not first class. This is a major source of annoyance in using Java and workarounds, also known as design patterns, are needed to overcome this gap. Finally, contracts (item 5) are only weakly supported in Java so contracts must often be specified in comments not code.

Orthogonality is a term borrowed from mathematics to describe the ability of some entity to be varied independently along several axes of parameters. Orthogonality is desirable in programs since it allows the programmer to make independent changes to decisions and settings. Data orthogonality is the ability to modify or substitute data values independently as long as the meaning of the program requires no mutual constraints. Data orthogonality is widely supported in programming languages. Functional or algorithmic orthogonality is the ability to vary functions or algorithms. This type of orthogonality is not so widely supported and so it often becomes important to develop patterns that enable such orthogonality to be achieved in spite of the language.

Suppose, for example, that an algorithm is embedded inline in the body of a method in a Java class. It is not altogether easy to change that algorithm. The most common solution to this problem is to define a derived class in which the method in question is overridden to use a different algorithm. This inheritance mechanism may be acceptable if there are only one or a few possible replacement algorithms but becomes unwieldy if there are many possible options. Moreover, if there are two methods with separate algorithms each of which may possibly be replaced, the entire process of inheritance breaks down in excessive complication.

A second solution is to use inline definitions that combine the definition of an object with the replacement of one or more of its algorithms. Since this technique is not well known, we will illustrate it by sketching an example. Consider:

```
public class Foo {
    ...

    public void bar(...)
        { ... original body ... }
    ...
}
```

Suppose we want to build a **Foo** object but replace its **bar** method inline. We may do this as follows:

```
Foo alternateFoo =
    new Foo( ... parameters ...)
{
    public void bar(...)
        { ... replaced body ... }
};
```

We can explain this to students as follows. In the **new Foo** operation, the ordinary parentheses enclose zero or more constructor data parameters that set or modify the internal member data of the object. If the ordinary parentheses are followed by braces, then the methods defined within the braces override corresponding methods in the original definition of the **Foo** class. We also point out to students that it is valid to add additional member data and additional helper methods within the braces. These helpers cannot easily be visible outside of the object but can nevertheless make the code more readable.

The technique of inline definition is obviously most useful when the number of replaced methods is small and the object created is one-of-a-kind. It is clearly less useful if we wish to make several objects with exactly the same method changes. It is worthwhile to note that the same technique works if **Foo** is an interface. In that case, use

```
Foo object = new Foo() { ... };
```

and supply all of the necessary methods of the interface.

The cognoscenti will observe that the technique of inline definition relies on anonymous inner classes for its implementation but there is never any need to say this to students. Instead, the construction **new Foo(...) { ... }** may be introduced as a syntactic idiom that permits the programmer to initialize both data and methods to a desired state.

The third way to achieve orthogonality for algorithms is not to embed alogorithms inline into functions in the first place. Rather, it may be better to have separate encapsulations for the algorithms needed by a class, to store an encapsulated algorithm as member data, and then to simply have the corresponding member function call on its current algorithm as needed.

As an example, let us show the setup for an abstract **ArrayAlgorithm** class that encapsulates algorithms that can operation on an abstract **Array** class.

```
public abstract class ArrayAlgorithm {

    public final void perform
        (Array data)
    {
        if (data != null) {
            int max = data.length() – 1;

            perform(data, 0, max);
        }
    }
    public abstract void perform
        (Array data, int min, int max);

}
```

Notice that the 1 argument method is **final** and simply calls the 3 argument method using the ends of the array as the **min** and **max** parameters. The 3 argument method is the one that must be supplied by the algorithm designer. This is an instance where a contract between two methods (1 and 3 argument versions) can be written into code and enforced.

In this situation, it is now quite convenient to apply the technique of inline definition to actually specify algorithms.

```
ArrayAlgorithm algorithm =
    new ArrayAlgorithm() {
        public void perform
            (Array data, int min, int max)
        { ... definition ... }
    };
```

Once these definitions are in place, **ArrayAlgorithm**'s may be passed to, stored in, and used by any object interested in doing something on another object that implements the **Array** class.

This technique, which is an instance of the *Strategy Pattern*, has been used successfully in teaching array algorithms and in providing automatic array algorithm animations. The students write the algorithms and supply them to a test suite. The test suite executes the algorithms on a **BarChart** class derived from the **Array** class that automatically repaints itself on the screen when any data element in the bar chart is changed.

Polymorphism is closely related to encapsulation and orthogonality. Polymorphism occurs if two behaviors can act differently even though syntactically in the language the invocations of the behaviors have the same structure. In an object-oriented setting, polymorphism is usually applied to the situation in which two calls **a.bar(...)** and **b.bar(...)** have different behavior. Of course, if **a** and **b** are the same type of object with different dumb data parameters (numbers, ...) then although the behavior may vary we do not normally use the word polymorphism. We restrict polymorphism to those cases where the difference in behavior is due to deeper reasons.

Often, polymorphism has been associated with behavior variations created by inheritance. However, as we have seen, inheritance is only one way to achieve orthogonality of behavior. Thus, it would seem that behavior variation induced by inline definition or by a different choice of algorithmic data object should equally be viewed as instances of polymorphism.

As we said above, the goal of programming is to solve problems in a fashion that permits these programs to be adapted cleanly to handle future problems or additional requirements. The essence of program design is to create a program that is so well organized

that adaptations are straightforward. We can now see that encapsulation, orthogonality, and polymorphism are essential programming concepts that guide the process of program design. A programmer must not only be able to write correct code, he/she must also be able to stand back and assess where in the program are the critical points at which implicit choices are being made via inline code. Once these choice points are recognized, the design will become more flexible if explicit orthogonality is provided so substitution may occur either by later programmer choice or by selection of an option by the end user. Encapsulation and polymorphism provide the means to achieve the desired orthogonality in well-designed programs.

3.2 Abstraction

The three fundamental topics we have just discussed, encapsulation, orthogonality, and polymorphism, are all related to abstraction so it seems useful to say a few words on this more general concept. Abstraction is first of all a process. If a person is confronted with several concrete objects, actions, or scenarios, it is natural to ask: "Are these all different or is there something in common?" Abstraction is the process of identifying what is common.

For example, consider some children's toys, say, balls and blocks. If we drop some of the balls and blocks from the same height, we may notice that they hit the ground at the same time. Thus, we have found a common property that we may abstract. If we lift a small ball and a large block, we may notice that it takes more effort to lift the large block than the small ball. This is a difference. Further investigation, may show that objects that are about the same size take a similar effort to lift regardless of whether they are balls or blocks so now we have found something common in the difference we just observed. Next, if we want to put the balls and blocks into holes in a wooden board, we will notice that balls fit better into circles and blocks fit better into squares. Thus, we have another difference and another common feature among a different subset of the objects.

Notice that, in these examples, our identification of what is common depends on the kind of observation we are making or question we are asking, that is, on the context. Thus, our definition of abstraction above is itself incomplete. We should say that abstraction is the process of identifying what is common in a given context.

Abstraction is the most pervasive concept in computer science since we cannot push actual objects of the real world into a computer. We must always deal with actual objects or actual actions or actual scenarios by means of some representation that captures what we think are the "important properties" of the entity. Making a representation is already an abstraction process since it is both impossible to represent all properties of a real object and quite undesirable to be so verbose. This is one of the root causes of the difficulty of teaching computer science, namely, that we must deal with abstractions from the very beginning. Both as individuals and as members of the computing community, we decide "what we want to study", "what properties are important about these entities we want to study", and "how shall we represent the objects, the actions, and the scenarios we want to study in the computer using some programming language(s) and paradigm(s)".

So, the fundamental problem of computer science is to create abstract representations of many entities using a medium, programming language(s) and paradigm(s), that is full of it own set of internal abstractions. No wonder computer science pedagogy is so hard.

The main thesis of this report is that the programming languages in use in industry and the classroom are so complex that the important abstractions in these languages are themselves obscured by the complexity. This is exactly the position articulated by Wirth in his keynote to ITiCSE 2002. It is our proposal to emphasize the key programming concepts in their cleanest form and then to explain, if needed, how to implement these clean ideas in the baroque languages we are often forced to deal with.

These recommendations are synergistic with the approach of Felleisen [4]. What Felleisen accomplishes is teaching students how far you can go using a strict recipe for data definitions and a corresponding template for function definitions. What is left is the more creative programming that is not simply forced by the structural information.

3.3 Paradigms

There are several programming paradigms that are commonly in use or are coming into existence: procedural, functional, object-oriented, event driven, logic, concurrent, parallel, genetic, and quantum.

Different languages adopt the various programming concepts in different ways (often with restrictions) to support the paradigm on which the language is based. Not all concepts are in all paradigms and therefore not in all languages. For example, programming by extension or subclassing is not a part of the procedural programming and so it has no direct support there. Since almost all languages are equivalent to a bounded Turing Machine you can introduce the "left out" concepts via lower level programming in your language of choice, but usually only with difficulty. Interestingly, Lisp, being among the simplest of languages, simulating most other paradigms in Lisp is not much harder than doing any other Lisp programming.

However, many people believe that each paradigm requires a certain "mind set" on the part of the programmer in order to program at the expert level in a language built from that paradigm. Often this mind set will conflict with that of another paradigm (hence the word, paradigm). In particular, if you think like a procedural programmer, where problems are solved with assignment and loops, you will have difficulty in Lisp as these concepts are missing there. In many ways, however, paradigms that don't overlap much are easier to switch into and out of than those that overlap strongly. This is due to the fact that you must adopt the more appropriate tools of the different paradigm as the more familiar constructs are probably not available. In Lisp, for example, you must use recursion in all those places in which you would use loops in C. When paradigms overlap, however, it can be much harder, especially in the learning phase, because it is tempting to use known constructs rather than become skilled in using the alternate paradigm.

Object-orientation and procedural programming both have the idea of state maintained in something like a variable and most OO languages also have *if* statements. But *if* statements are much less essential to the OO paradigm and expert OO programs have few *if* statements, relying instead on dynamic polymorphism. While this seems like a simple distinction it is actually quite profound requiring a rather different mind set. In procedural programs in

which decisions are made with *if* statements, a lot of information about the structure of the data is distributed throughout the program. Everywhere that we must manipulate a complex data structure in a large program we need an *if* or *switch* to distinguish cases. This distribution of information has proven problematic in large programs as it decreases cohesion. On the other hand, in OO programming, information about a data structure and its operations are centralized in one or more related class definitions which implies that there is often a single point of change in the program as the problem evolves or is extended.

Viewed from a client-server standpoint the procedural programmer has to get the clients right: the code that calls the functions. Client code must know what function to call even as the data referenced may vary. This is actually improper information for a client to have for it requires to client to know too much about the associated data. This kind of knowledge is best encapsulated by making the data into a server object that knows about its own appropriate behavior. More precisely, there will usually be a set of server objects that may be supplied to the client and each of these servers can do one thing well without explicit decisions. So, the procedural programmer must get the clients (callers) right while the successful OO programmer thinks in terms of getting the servers right and also in getting a sufficiently rich set of servers to cover all options.

The implication of all this is that when the basic concepts are matched to paradigms it is not enough to just present the concepts or the paradigms. Two additional things are needed. We need to show students how the concepts maps into the paradigms, including any restrictions, and we must also guide the learner in how to adopt the appropriate mind set that they need to be successful in the paradigm. Typically the standard idioms of the language help in this. More recently, design patterns have been shown to be even more powerful in capturing the mindset of the successful practitioner.

There is one aspect of this discussion of paradigms that may be altered as experience with a *concepts first* approach develops. A great deal of the difficulty that procedural programmers have with learning the object-oriented paradigm may be due the fact that the limitations of the procedural paradigm are so significant that the procedural programmer builds the necessary workarounds deeply into his/her mind and thereafter such workrounds are hard to dislodge. With a more balanced *concepts first* approach in which the idea of value is so general, students may learn to think flexibly from the very beginning and find that transitions from paradigm to paradigm are not as difficult as in the past.

3.4 The Role of Toolkits

One of the authors, Richard Rasala, together with his colleagues, Jeff Raab and Viera Proulx, has built a large toolkit known as the Java Power Tools [11] whose purpose is to dramatically reduce the time required to create graphical user interfaces in Java. We wish to discuss here how the Java Power Tools and similar toolkits can contribute to the goals of the *concepts first* approach.

As Wirth states, the fundamental problem with commercial languages is that they exhibit complexity far in excess of what is conceptually required to solve the problems they deal with. In Java, such complexity is evident in the design of the GUI tools. Although Swing is a significant advance over the earlier AWT, it is still that case that creating a GUI using Swing is a painstaking experience with myriad repetitious details. The difficulty is that the tools provided are too atomic, that is, that they provide the raw ingredients needed to build a GUI but not the encapsulations needed to match the higher level of thinking that a GUI designer would like to employ. This gap between the conceptual level of GUI design and the granularity of the actual Swing tools is the source of the extra complexity that is found in building Java GUIs.

The origin of the Java Power Tools was in the desire to provide quality GUIs for freshman computer science courses. However, from the very beginning, it was our plan that these tools would be of such high quality that students would wish to use them throughout their education and that we, as faculty, could use them as design models in upper-level courses on object-oriented design and software development. Therefore, every aspect of the design was carefully considered from the generality of the abstractions, the structure of the classes, and the names of the methods down to the format of the code so that it could be displayed in a large font for classroom presentation. No detail was left to chance. We did not want our students to view these tools merely as "training wheels" but rather as a serious model of object-oriented design.

An ITiCSE 2003 paper [12] discusses the design themes of the JPT. Let us briefly summarize some key points here. The most fundamental theme is extreme encapsulation, that is, the systematic effort to encapsulate all technical detail so that the user of the JPT classes need only provide those parameters that relate to the conceptual design level. One instance of this is the **TablePanel** class which can organize a one or two dimensional collection of objects into a panel with corresponding Java components. The conversion of objects to components is an important aspect of the power of this class. Another example is the systematic encapsulation of algorithms as Java **Action** objects which may then be used define the behavior of buttons and other components. A wrapper class **ThreadedAction** may be used to guarantee that each click of a button causes its associated algorithm to be run in a separate thread. Thus, basic multi-thread programming is achieved almost for free. A third instance of encapsulation is the **Paintable** family of classes that encapsulate shapes, images, and text in a uniform fashion, handle mutation by affine transformations, and arrange that property change listeners are automatically set up with no special manual intervention. Using encapsulations of this caliber, it is possible to build GUIs in minutes rather than in hours or days and, indeed, GUIs have often been created in lecture on demand in a short period of time.

There is a special set of classes in JPT called the *Java Power Framework* (JPF) that are used to create automatic test suites. The user builds a class, say, **Methods** that extends the base class **JPF**. When the constructor, **new Methods()**, is called, the base class constructor then creates two panels, a simple console for text I/O and a graphics panel with buttons to control the test process. Using Java reflection, there is a button in the GUI for each method in the class **Methods** that is public and has simple arguments and return values. Clicking a button executes its associated method. If the method requires arguments or has a return value then an auxiliary GUI is generated on the fly that permits to user to provide the arguments, execute the method, and see the return value.

Using JPF, a student can execute Java code with as little as one method. Thus, JPF is a great tool for exploration and learning. As the student begins to develop more complex classes and programs,

JPF may be used to create and save the methods that do systematic unit testing as well as testing of the integrated set of components. The beauty of this approach is that the test methods do not clutter up the classes being built but nevertheless are instantly available at the click of a button in the JPF graphical user interface. Finally, since a method may be defined to call the **main** program of an application, it is also possible to build test buttons for testing a complete application.

The Java Power Framework is related to but different from the well known Java environment for students called BlueJ [9]. In BlueJ, the focus is on building objects and then testing method calls on these objects. In JPF, the focus is on behavior, that is, on executing methods. Using methods, one can access member objects, create new objects, test algorithms, do all sorts of graphics, open auxiliary windows, and launch entire applications. We believe that JPF is flexible enough for the entire range of mature test activities and every project we do is undertaken using a JPF test framework from the start.

Having sketched the Java Power Tools and its test framework Java Power Framework, let us now explain why these toolkits are relevant to the themes of this report. The fundamental goal of the *concepts first* approach is to cut through the conceptual complexity of commercial languages and their libraries. We want students to understand the big ideas and not to be distracted by a maze of technical debris. We want students to be able to explore easily and to gradually build a vision of what can be encapsulated, that is, what may be considered as a *value* in a programming language. The Java Power Tools form, if you will, an existence proof of the contention that one can dramatically simplify the process of programming if one works with high level abstractions that combine power and generality. Before we built the Java Power Tools, many people who worked with graphical user interfaces and event driven programming believed that these subjects were inherently complicated and nothing could be done about this complexity. Now that we can build GUIs in minutes without ever visibly wiring an event listener, such beliefs have been proven to be incorrect.

From a *concepts first* viewpoint, the Java Power Tools demonstrate that it is possible to write elegant conceptual code in a language like Java. The JPT promotes an integration of the functional declarative style of Scheme and the traditional object-oriented style of Java. The traditional object-oriented style focuses on the data and indeed this is the foundation level of programming that must be mastered. Nevertheless, as programs become more complex, one does not want to continue to focus primarily on the data. The heart of a large program are the classes that encapsulate the mutually supporting behaviors of the program and that hide all information about the underlying program data that is being manipulated. At this high level, the functional programming style is much more important and the object-oriented style is simply the encapsulation infrastructure.

The transition from thinking about objects as data with associated behavior to objects as behavior with associated data is a bold step that is supported by the *concepts first* approach. The *concepts first* approach argues that we should frame the key operations in programming in as general a form as possible prior to committing to any particular paradigm. Indeed, individual paradigms make more sense if we understand their relationship to the other paradigms rather than what makes them distinct. The role of toolkits is then to provide the bridge between paradigms, that is, to provide the links that the language designers left out.

Finally, a word must be said to those who object to the use of toolkits on principle because students will not use the *real* language and will therefore be handicapped in industry. Those who have read the Wirth keynote quoted earlier will know that he would strongly disagree. The same question was addressed and answered more than 10 years ago by Eric Roberts [17] when the language in question wasn't Java but C. Roberts writes:

> These reviewers argued that I was changing the language or that I was likely to distract the students by providing a mechanism that they would "remember after the course is over, instead of remembering **scanf**." One reader felt sufficiently strongly about this question to set an entire sentence in italics: *"If you are going to teach ANSI C, teach ANSI C, not some modified local version of the language!"*
>
> I believe that the last comment cuts to the heart of the controversy. I was not trying to teach ANSI C. I was trying to teach programming.

If you look back from a vantage point of 10 years and see how silly it now seems to be fighting over **scanf** then you realize that our purpose as educators cannot possibly be to teach particular languages or technologies as they are used in industry today. This is one of the main points of this working group report. Our goal as educators is to teach programming and all of the grand ideas that surround this wonderful discipline. Our goal is *not* to teach a particular language (C, C++, Java, C#, or Scheme) and our goal is *not even* to teach a particular paradigm. Our goal is to provide a vision of what might be in the future given a strong framework of elegant programming concepts that crosses languages and paradigms.

3.5 Other Concerns

In this section we have collected some of the concerns that working group members expressed about the programming theory of programming approach to introductory programming courses. Brief responses are provided in typical FAQ style.

Concern: *For practical and political reasons, many colleges and universities will not be able to adopt the Oz language. Java is currently a very popular language, and the question arises of how or whether this set of concepts and theory of programming apply, or can be illustrated, in an introductory course which uses Java as the programming language.*

Answer: The main purpose of a theory is to provide a framework for better understanding of all of programming and to develop a mindset in students that ties together concepts that due to implementation distortions seem unrelated. Oz is a complete, powerful programming language, but lab-work to support the concepts of the programmer's theory of programming and the Kernel Language requires only a small subset of Oz [19]. A GUI based incremental compiler coupled to a run-time system is available that allows Oz [10] to be used as a theory support tool with a very high gain/pain ratio. Java can, therefore, still retain its place as the preferred real-world programming language for introductory computer science courses. In addition, toolkits such as the Java Power Tools [11] demonstrate that even in Java a more conceptually higher level of programming is possible.

Concern: *The programmer's theory of programming deals with fundamental programming concepts at a very low level. Object-oriented concepts can be regarded as higher-level, problem solving abstractions imposed on top of programming.*

Answer: True, except that the use of the phrase "very low level" versus "higher level" carries the, perhaps unintentional, suggestion that masters of higher levels should not be burdened with details of lower levels. To create an object or a method, the programmer has to write statements. A statement contains variables and values. Hence a thorough understanding of these concepts is as important as an understanding of the problem solving concepts of OO that help one to determine what classes and objects should be created and what behaviours should be incorporated into their methods. Moreover, the general concept of value as any entity that may be encapsulated is a framework that makes sense of the particular declarations used in a language like Java.

Concern: *Using a language at variance with the theory is a compromise and presents certain challenges. If Java is the language of instruction, the concept can be presented and an in-depth discussion of the theoretically "right" decision should be deferred to a more advanced course.*

Answer: Once a student forms the mindset that Java or any other first-language is the only way to program, it is almost impossible to expand that student's viewpoint in subsequent courses. The theory expands the definitions of the basic concepts of programming so that students can see relationships between programming languages and concepts that were not immediately obvious before. The key point in starting with these concepts is to get the student to separate the concept from the restrictions, extensions and other necessary or *ad hoc* decorations that disguise that concept in practical programming languages. For example, traditional languages emphasize stack-based allocation of data and the automatic recovery of such data when the stack frame is closed. This view actually obscures the fact that objects in Java are global and do not live simply in the context of a particular stack frame. Therefore, being too traditional can be a pedagogical problem.

Concern: *Java is a strongly typed language, and variables are declared with a given type. The beginning student can be safely told that a variable name, or identifier, is a shorthand for accessing or using the value stored in a particular location in the memory of the computer.*

Answer: Strong-typing can be achieved by static (compile-time) type checking or dynamic (run-time) type checking. Java "starts with" static type checking, but "gets into trouble" and has to introduce dynamic type checking to deal with polymorphism. A clarification of the concept of strong-typing simplifies the comprehension of polymorphism considerably and helps students to understand, in particular, that the type of an object and the type of a variable that refers to that object may not be the same.

Concern: *When describing variables and types in Java, in contrast to the theory of programming given above, it is necessary to talk about references. When objects are created, the "handles" for these objects are references. These are not pointers, but unique hash codes based on the location of the object in memory, and they can be associated with named identifiers which are declared with a class type that is compatible with an object that a reference refers to.*

Answer: By careful definition of the concepts of value, variable, assignment and the binding rules of variables and values, the theory explains both simple variable binding and object references with one set of concepts. This explanation needs no implementation concepts such as hash codes or memory locations.

Concern: *In the presentation of the theory of programming given above, there are certain concepts, which are included* ***without a separate definition****. Among these are functions, procedure declarations, and repetitive statements.*

Answer: The above section illustrates the theory of programming with a few pertinent examples. The whole theory and the Kernel Language contain precise and complete definitions of functions, procedures, repetitive constructs and other concepts that are essential to multi-thread, multi-paradigm programming.

Concern: *In the presentation of the theory of programming given above, there are also certain things, which are said to need a new definition. The handling of variables and types in Java differs significantly from that in the theory of programming, but it does not differ significantly from prior languages. Since over the previous 5 decades programming has progressed with this earlier treatment of variables, it seems safe to say that although perhaps not ideal, the treatment in Java does no harm, and will provide a sound foundation for further elaboration of the theory later in the curriculum. The point is that in Java, no new definition of this is needed.*

Answer: Not so. The theory accommodates static and dynamic binding and explains their differences and the reasons why one or the other was preferred by the designers of a programming language e.g. Java. The theory facilitates the introduction of "references to objects" especially to students with C-programming experience who have trouble with the conceptual separation of references from pointers.

Concern: *To achieve learning, we have to engage the attention and awaken the interest of the students. For many students, computer programming is an abstract practice.*

Answer: As in conventional courses, visualization can aid understanding and motivate students to spend more time with class material. If each line of code has a visible effect, students immediately appreciate core concepts. Early in the course, a student can observe variable assignments through changing a quality of an on-screen object or by lighting pixels. With relatively little pre-existing structure, more complex behaviour can be implemented through message passing and through composition of simpler objects. Combined with time delay, students can observe sequential computations to better understand order of operations. Sometimes graphical environments require students to interact with relatively complex systems. The interaction can serve as an introduction to basic pattern use.

Concern: *A course project provides a culminating experience that can build a sense of satisfaction within students. It is also an opportunity for a student to demonstrate knowledge and for the instructor to assess whether the core concepts have been effectively taught. Does the theory approach leave enough time for class projects?*

Answer: The theory approach saves class time through better coherence and simpler explanations of programming concepts that are currently regarded as difficult. Time is also saved by use of

carefully chosen software tools that have high gain/pain ratios. As in current courses, a class project can be a programming project separate from the rest of the class programming assignments, or it can be a series of incremental programming assignments integrated into the class programming assignments. There are many obvious options for independent projects. For example, here is a project that can be integrated into a set of first-semester programming assignments:

Media database project: Through an introductory course, a class may develop a database to store book, movie, or music information. The development begins with the introduction of objects, constructors, and assignment. The assignments progress to add features incrementally to prompt the user for new entries, print out database entries, search the database, store entries to disk, to develop a graphical interface to the database. Each successive assignment allows the instructor to guide design decisions and publish the design of the previous assignment.

4. CONCLUSIONS

Earlier efforts to include several programming paradigms into introductory CS courses had to introduce a different programming language and a different program development system and compiler for each paradigm [15]. In spite of severe difficulties, this approach showed promising results [16]. Today, at least one highly usable and effective version of each key component for success with a multi-paradigm, multi-thread concepts-first approach to introductory programming courses is in place.

We have a programmer's theory of programming that uses the basic concepts of programming to gain unifying insights about programming as such, in any paradigm or programming language.

We have well defined Kernel Languages that introduce the essential program components of all paradigms in a surprisingly short evolutionary sequence starting with a simple functional Base Kernel Language.

We have a programming language of which the Kernel Languages are proper subsets. We have a programming system that provides a GUI based incremental compiler that can be used as a theory and Kernel Language study tool.

We have toolkits, like the Java Power Tools, that save time in teaching of Java programming and serve as models of good Java program design and implementation, thereby saving class-time for theory and Kernel Language intro to other paradigms.

Finally, we have increasing demand for net-centric GUI based distributed computation designers and programmers who will almost certainly have to program in several languages and paradigms sometime in their computer science careers.

All we need is courage and enthusiasm to use this approach to endow computer scientists with a good understanding of and enthusiasm for all of programming, unrestrained by any specific paradigm or programming language.

5. ACKNOWLEDGEMENTS

Many thanks to the ITiCSE 2003 Organizing Committee for an excellent conference, especially to Mike Goldweber for enthusiastic and excellent organization of the working group process, facilities and smooth operations.

Thanks to the working group members for sustained, sometimes hard, but always fair and honest discussions.

6. REFERENCES

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman, *Structure and Interpretation of Computer Programs, second edition*, MIT Press, 1996.

[2] Hassan Aït-Kaci and Andreas Podelski, Towards a Meaning of LIFE, Journal of Logic Programming 16(3-4), (1993), 195-234.

[3] Edsger W. Dijkstra, A Discipline of Programming, Prentice Hall, (1976)

[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi, *How to Design Programs: An Introduction to Computing and Programming*, MIT Press, 2001.

[5] David Gries, Where is Programming Methodology These Days ?, SIGCSE Bulletin 34(4), (2002), 5-7.

[6] Martin Henz, Gert Smolka, and Jörg Würtz, A Programming Language for Multi-Agent Systems, 13th International Joint Conference on Artificial Intelligence, Morgan Kaufmann, (1993), 404-409.

[7] Sverker Janson and Seif Haridi, Programming Paradigms of the Andorra Kernel Language, International Symposium on Logic Programming, (1991) 167-183.

[8] Donald E. Knuth, Keynote Address ITiCSE 2003, video at **http://iticse2003.uom.gr**

[9] Michael Kölling et al, *BlueJ* , **http://www.bluej.org**

[10] Mozart Programming System, **http://www.mozart-oz.org**

[11] Richard Rasala, Jeff Raab, and Viera Proulx, *Java Power Tools*, see: **http://www.ccs.neu.edu/jpt**

[12] Richard Rasala, *Embryonic Object versus Mature Object: Object-Oriented Style and Pedagogical Theme*, SIGCSE Bulletin, 35(2), 2003, 89-93.

[13] Juris Reinfelds, A Programmer's Theory of Programming, unpublished report, (2003).

[14] Juris Reinfelds, Teaching of Programming with a Programmer's Theory of Programming, Informatics Curricula and Teaching Methods, Editors Cassel & Reis, Kluwer Academic Publishers, (2003) 41-51.

[15] Juris Reinfelds, A Three Paradigm Course for CS Majors, Proceedings, 26th ACM SIGCSE Symposium on Computer Education, (1995) 223-227.

[16] Juris Reinfelds, 1996 Lecture Notes for CS 272 (new), Department of Computer Science NMSU, (1997).

[17] Eric Roberts, *Using C in CS1: Evaluating the Stanford Experience*, SIGCSE Bulletin, 25(1), 1993, 117-121.

[18] Peter Van Roy and Seif Haridi, Teaching Programming Broadly and Deeply: the Kernel Language Approach, in Informatics Curricula and Teaching Methods, Eds. Cassel & Reis, Kluwer Academic Publishers, (2003) 53-62.

[19] Peter Van Roy and Seif Haridi, *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004 (to appear).

[20] Niklaus Wirth, Computing Science Education: The Road Not Taken, SIGCSE Bulletin 34(3), 2002, 1-3.