ABSTRACT

Objects first is a pedagogy that tries to introduce the core concepts of object-oriented programming - classes, objects, and methods - as early as possible in a programming course, even before variables, types, assignments and control structures are explicitly introduced. The concept of a named interface is typically introduced at a much later stage, usually in connection with inheritance, polymorphism, and abstract classes. In this paper we point out that interfaces as a language mechanism can be introduced much earlier, even before inheritance. This way the concept of an explicit class interface can be decoupled from the more complicated issues of inheritance and subtype-polymorphism.

Categories and Subject Descriptors

K.3.2 [Computers & Education]: Computer & Information Science Education - Computer Science Education
D.1.5 [Programming Techniques]: Object-Oriented Programming.
D.3.3 [Programming Languages]: Language Constructs and Features – Abstract Data Types, Interfaces, Polymorphism, Inheritance.

General Terms
Languages

Keywords
Pedagogy, Objects-First, Java, Interfaces, Type Abstraction, Subtyping, Polymorphism, Inheritance.

1. INTRODUCTION

Objects first is a pedagogy that tries to introduce the core concepts of object-oriented programming - classes, objects, and methods - as early as possible in a programming course, even before variables, types, assignments and control structures are introduced [1]. BlueJ [4] is a programming environment tailored for teaching that ideally supports the objects first approach [10]. We successfully applied this approach in an undergraduate course (first year) on imperative and object-oriented programming during the last three years.

While objects first worked well for us, we encountered more problems in teaching one particular language mechanism of Java: named interfaces. Named interfaces are an important means to raise the level of abstraction in software projects. Canning et al. quote Burstall and Lampson in [5]:

"The most important recent development in programming languages is the introduction of an explicit notion of interface to stand between the implementation of an abstraction and its clients."

Wise use of interfaces can significantly improve the structure of software and lower the degree of coupling in an application [7]. Despite this, interfaces are less commonly used than one could expect, even in production code of large projects [13].

We suspect the pedagogy of programming courses to be partly responsible for this disregard: named interfaces are typically introduced late and in connection with the more complicated notions of inheritance and subtype-polymorphism. In this paper we point out that named interfaces can and should be introduced much earlier, independently of inheritance.

For the following discussion we distinguish the concept interface from the mechanism interface by setting the terms in different fonts:

- The concept interface (set in italics in the following) denotes the conceptual interface of a class, i.e. all the methods that are public and are thus available for (regular) clients of the class. The interface of a class describes what instances of the class can do, not how they do it.

- The mechanism interface (set in typewriter font in the following) is a construct of the programming language Java (and of other languages as well) that allows to describe just the signatures of the methods of a type. An interface can be used to explicitly describe the complete interface of a class or it can be used to partially describe a role that a class can play in certain contexts [13].

In the next section we take a closer look at how interfaces (here we refer to both notions) are taught in typical courses. The subsequent section outlines a different approach we applied for the first time in our undergraduate course in Spring 2003. Section 4 summarizes the advantages of the new approach. Section 5 concludes the paper.
2. THE CLASSICAL APPROACH

The introductions of interfaces and interfaces are typically placed at different ends of a programming course.

Interfaces can be taught quite early in a programming course. Typically they have to be mentioned when library classes are used for the first time, as the documentation generated by javadoc describes exactly the interface of a class. Interfaces have to be picked as a central theme again in connection with concepts such as coupling, cohesion and information hiding.

Interfaces, on the other hand, are introduced quite differently, usually along the following line of argument, as for example in [1]:

“Avoid duplication, use inheritance. Make software more flexible and extensible, use subtype-polymorphism. Apply reuse with “Avoid duplication, use inheritance. Make software more flexible and extensible, use subtype-polymorphism. Apply reuse with adherence. This way the construct appears as just a shortcut, not very appealing to students.

Other examples of this approach can be found in [3], [14], [8] and [9].

This argumentation connects interfaces closely with inheritance and appears to stem from the historical roots of the mechanism:

- In the beginning of the 1990s, the necessity to distinguish subtyping and inheritance in object-oriented programming languages became evident [6]. This fundamental distinction allowed designers of new languages to offer multiple subtyping without having to offer the complexity of multiple inheritance. Another example of this design rationale besides Java is the language Theta [12].

- The usefulness of fully abstract classes in C++ lead to the new language construct signature, as described in [2]. Signatures are similar to interfaces in Java as they allow to bundle signatures of operations under a new type name, but are more flexible, as conformance of a class to a signature is defined structurally (matching signatures), not by explicit declaration in the class. This allows a polymorphic abstraction over a set of classes to be defined after the definition of the classes, without the need to change them.

Signatures can be seen as one predecessor of the interfaces in Java. They never found their way into the language definition of C++, but are part of the GNU gcc distribution since 1996.

Even though interfaces are historically motivated by inheritance concerns, they need not necessarily be introduced in connection with inheritance. We encountered several problems with this inheritance-related introduction of the mechanism:

- Introducing interfaces as abstract classes with just abstract methods totally underrates the power of the mechanism. This way the construct appears as just a shortcut, not very appealing to students.

- Quite often students ask: “What are interfaces for? They do not contain any code, so they do not do anything, so what are they good for?”

- We even heard statements like: “Interfaces are needed for multiple inheritance, multiple inheritance is complicated, so better I do not touch interfaces.”

- The idea that an interface can be used to explicitly model the interface of a class before the class is actually implemented is not well adopted by students, if at all.

- The advantages of changing an implementation without having to change the clients code can be demonstrated well with interfaces, but are buried under inheritance complexity.

The central question for us thus was: Why not use interfaces to better point out the notion of interfaces, but without talking about inheritance?

3. INTERFACES BEFORE INHERITANCE

Triggered by this question, the last time we conducted our course we took a slightly different approach. Before we can describe it, we have to explain some supporting terminology.

3.1 Operations and Methods

From the very beginning of the course we distinguish operations and methods. Operations are part of a type and describe the abstract notion of operations on the values of a type’s domain. Operations describe what can be done with the objects of a type, not how it is done. Methods on the other hand are structural parts of classes and contain code. Public methods can then be seen as realizations of operations defined by a type.

The advantage of this distinction becomes evident in connection with inheritance. Take the method equals of the Java class Object as an example. The type Object defines the operation equals, the class Object also supplies a realization of this operation (an implementation as a method). Subclasses of Object that redefine equals do not offer an additional operation, they just offer alternative realizations of still the same operation.

But even without this rather complicated argument the distinction did not seem unnatural to our students. Within BlueJ they interact with objects during their first lab classes. As interactive clients they call the operations of the objects on the object bench (we just talk about operations in our first assignments). Later, when they open class definitions and edit them, we “reveal” that the operations offered are realized by methods that implement the operations.

3.2 List: Interface and Implementations

It appears much more natural within this terminology to introduce a language construct that allows to describe just the operations of a type. We introduced this concept during the discussion of collection classes in week 6 with the interface List of the Java standard library. Only in week 7 we started with subtyping and inheritance.

For the lab classes of the course we gave out an assignment every week over the first eight weeks. Each assignment had a particular focus:

- week 1: interactively using classes and objects, calling operations, passing parameters
• week 2: opening class definitions, classes have methods and fields
• week 3: assignments and conditionals, objects use other objects
• week 4: loops, more on objects using other objects
• week 5: static collections of objects (arrays)
• week 6: dynamic collections of objects (lists and sets), interfaces
• week 7: subtyping
• week 8: inheritance

Weeks 9 to 14 were conducted as a mini project and do not contribute to this discussion. For the first assignment of week 6, we provided an implementation of a simple notebook for an arbitrary number of notes, based on a project taken out of chapter 4 in [1]. The class Notebook initially offers the operations storeNote(String note), int numberOfNotes() and showNote(int noteNumber). Internally, the class uses a field named `notes` of type `List`, denoting the interface `java.util.List`. In the class code, there is only one point where a concrete implementation of this interface is used, namely `ArrayList` in the constructor of `Notebook`. The students then have to add further operations (such as `removeNote`) to the `Notebook` class.

During this assignment we could demonstrate and discuss several aspects of this scenario:

• The one usage of `ArrayList` can be replaced with `LinkedList` without any problem, as the rest of the class code just uses the interface.
• Having a clear understanding of the needed functionality is more important than an efficient implementation of this functionality (“First make it right, then make it fast”).
• If efficiency becomes an issue, the choice of the concrete implementation depends on the profile of usage (for large lists, many insertions at the head of the list are more expensive with `ArrayList`s, whereas many random accesses on large lists are more efficient with `ArrayList`s).
• Changing an implementing class of an interface does not necessarily lead to a recompilation of client code of the interface. This can be visualized nicely in BlueJ, where a class diagram is provided and classes that need to be recompiled due to source code changes are marked as striped.

Consider a class `A` that directly uses class `B`. Any change to class `B` leads to `A`’s and `B`’s representation in the diagram being striped. If we introduce an interface `C` that models the interface of class `B` and let `A` just use `C`, then any change to the method bodies in class `B` only lets the representation of class `B` be striped.

This way, even before students have heard the term polymorphism, they quite naturally use one application of it for data abstraction (i.e. the concept of allowing different implementations for an abstract data type), see [11]. But the more important point is that the fundamental separation of interface and implementation is explicitly addressed and supported by an explicit language mechanism.

3.3 Evaluation

We have changed the course as described for the first time in Spring 2003. We did not explicitly gather feedback on this new approach, because first we wanted to see if it works at all. It could have happened that the students were not able to do what we expected them to do, complaining that they do not understand inheritance yet and thus are not able to use interfaces. Nothing like this happened, the course went quietly over this point, and we had no complaints or problems.

To get a better feeling for the effect of the change we will conduct a survey in the Spring 2004 semester. We will ask the students before the course what their understanding of the term interface is (as some students already have an understanding upfront). After the course we will ask again and compare if there is a significant change in relating interfaces to inheritance.

4. ADVANTAGES

We think that our approach of introducing interfaces before subtyping and inheritance has several advantages:

• The notion that a server’s interface for a client is independent from its implementation can be stressed, as it is supported by an explicit language construct.
• The important notion of “programming to an interface” [7] can be practiced much earlier and explicitly.
• It becomes more obvious that a common mistake of class documentation can be avoided: mentioning the private fields of a class in the javadoc comment of a method. If the respective operation is commented in an interface, there are no private fields that could be mentioned.

5. CONCLUSION

The concept interface of a class is such an important notion in object-oriented programming that it should be introduced as early as possible. Making use of the explicit language mechanism interface to make an interface more tangible for this purpose seems evident, but is not practiced in most courses. We have used this approach in an undergraduate course and introduced interfaces before inheritance. We think that this way students both get a better understanding of the concept interface and become more confident in using the mechanism interface.

6. ACKNOWLEDGMENTS

My thanks go to my colleagues in the Software Engineering Group at the University of Hamburg that prepared and conducted the lab class for our undergraduate course with me.

7. REFERENCES