Strategies for Teaching Object Oriented Concepts with Java

Miguel-Ángel Sicilia

University of Alcalá

Ctra. Barcelona, km.33.6 – 28871 Alcalá de Henares, Madrid, SPAIN

Abstract. A considerable amount of experiences in teaching object oriented concepts using the Java language have been reported to date, some of which describe language pitfalls and concrete learning difficulties. In this paper, a number of additional issues that have been experienced as difficult for students to master, along with approaches intended to overcome them, are addressed. Concretely, practical issues regarding associations, interfaces, genericity and exceptions are described. These issues suggest that more emphasis is required on presenting Java programs as derivations of conceptual models, in order to guarantee that a thorough design of the object structure actually precedes implementation issues. In addition, common student misunderstandings about the uses of interfaces and exceptions point out to the necessity of introducing both specific design philosophies and also a clear distinction between design-for-reuse and more specific implementation issues.

Introduction

A considerable number of studies and experience reports concerning the use of Java as the language of choice for introductory programming courses have appeared in the last years. Some of the existing reports emphasize the use of diagrammatic representations to enhance the comprehension of object-oriented concepts, often depicted in UML [24] or any other analogous notation — (see, for example, [3,27]). Many others propose an object-first approach for the structure of the curriculum (e.g. [7,28]). However, learning the core concepts of object-orientation and mastering the translation of conceptual models into Java programs poses significant comprehension problems for students that should be carefully approached from an instructional perspective. The classroom experiences synthesized in this paper point out that several important concepts like associations, generic containers, and the differences between interfaces and classes pose difficulties for students under some circumstances.
All these issues can be considered as concrete manifestations of the difficulties caused by the higher level of abstraction in thinking that object orientation requires

in comparison to procedural programming, as recognized by Hadjerrouit in [16]. The diverse origins of these problems are worth investigating, since they are an important part of the essential conceptual framework of Object Oriented Programming (OOP), and they form part of the everyday practice of object oriented software engineering (OOSE).

In this paper, we deal with a number of such problems, and we also propose very specific instructional practices intended to overcome them, which were put into practice by the author in diverse classroom settings. The present work is the result of experiences, analysis and classroom interventions in diverse teaching experiences of object orientation using Java in the years 1997 to 2003, with an emphasis on situating object-oriented programming as an activity that follows conceptual modelling and design considerations, eventually inside an iteration. Some of the problems found led us to consider that it would be adequate to provide students an early understanding of the concepts of analysis, design and programming as three different — but seamlessly integrated — development activities. In particular, the problems and the associated prospective solutions were obtained from the experience in the following courses using Java:

- From October 1997 to October 1999, as part of the B.Sc. programs on Computer Science of the Pontifical University of Salamanca in Madrid (UPSAM). In this setting, intermediate-level courses (according to the scheme provided in [1]) on Object-Oriented Programming and Design using Java (after an imperative-first introductory level approach) were taught. Advanced level courses that included Java for specialized uses (distributed objects programming, concurrency and object oriented databases), and Java as extra-curricular supplementary activity for students in the last stage of their degree, were also taught in this period.
- Since November 1999, as part of the B.Sc. programs in Computer Science and Telecommunication Engineering, CS1-level courses following an object-first approach were taught at the Carlos III University (UC3M), following an approach combining object concepts and algorithm design, similar to that described by Rajaravivarma and Pevac [28].
- Since October 2001, second-semester distance CS1 courses have been taught to students with previous background on procedural programming in the C language, at the Open University of Catalonia (UOC). In this case, a continuous assessment approach was used to foster the early engagement of students in practical activities, as described in [29].

In addition to the abovementioned experiences, several corporate training courses on Java programming were taught to senior programmers. These courses provided an interesting ground for comparing learning difficulties of students *versus* professional programmers.

The problems addressed in this paper are closely tied to the fabric of object orientation as a paradigm. Although those problems were especially patent, to different extents, both in the objects-first and procedural-first approaches, they also have an effect in more advanced courses. In consequence, these problems can be considered to be connected to the “conceptual knowledge” layer [16], rather than to the specifics of Java as a programming language. Nevertheless, they have been addressed in classroom settings using a combination of UML and Java as a seamless continuum from conceptual models to programming constructs. The link that crosses all the concrete issues described in this paper is that all of them are somewhat related to modelling or design activities, so that they complement lower-level issues about concrete syntactical structures or programming interfaces that are dealt with in other studies.

Over the course of the years, we have extended teaching practices of Java programming elements with a number of (micro-level) concrete analysis and design practices connected to UML diagrams. These practices are essentially oriented to foster modelling from the object to the class layer, and acquiring abilities to reason about the adequacy of using extensibility mechanisms and assigning responsibilities to parts of the software. The motivation for these practices will be detailed in the light of the experience of six years of teaching and mentoring students with diverse backgrounds in several different institutions, as described above. They have been found to be fairly effective in improving the understanding of object orientation, coming up with better object designs for Java programs. The issues addressed here somewhat complement other well-known pitfalls about this language that have been described elsewhere [4,17,15,22,10], ranging from primitive types as a weakness of Java [25] to the subtleties of Java I/O [30].

The rest of this paper is structured as follows. Section 2 describes learning issues associated with the basic object structure, i.e. classes and associations. Section 3 deals with teaching extensibility-related concepts. Section 4 deals with the adequacy of introducing a streamlined version of the “Design by Contract” philosophy at CS1. Finally, concluding remarks are provided in Section 5.

From Object Models to Java Programs

Since the early emergence of objects as a paradigm shift from previous technology, the seamless integration of object concepts throughout analysis, design and programming activities has been considered one of the main benefits of object-orientation (see, for example, the discussion of this aspect in [31]).

In essence, object-orientation is an approach to software modelling that puts its primary emphasis on structuring data around the notion of classes of objects. The approach then lefts the allocation of functionality to objects in the form of “methods” or “operation” to a subsequent step, as explained in detail by Booch [6]. Consequently, teaching object-orientation concepts should ideally start from the basic information structure of object models, consisting of objects, values and links at the “object level”, and their respective class-level counterparts: classes, attributes and associations at the “class level”. Following this basic pedagogical assumption, in this section we deal with some learning difficulties that have been identified with regards to devising basic object structures and the subsequent derivation of Java programs from them.

Classes and Objects

Commonly used informal techniques for obtaining tentative object-oriented models focus on proposing “candidate classes”, extracted in some way from the problem statement [31][2]. These techniques then provide guidelines for discarding some of them, e.g. removing redundant classes or eliminating classes that may better be considered attributes. These commonly used techniques are useful in many cases, but they have the drawback of relegating objects to a secondary place, since the emphasis is given to class structure. This results in that some students often do not turn their attention to considering examples of instances in case of doubt. Many practical difficulties in identifying classes simply vanish when approaching the model by considering “instances first” or emphasizing the value of object diagrams as a tool for class modelling, considering them in parallel to the identification of classes and class structure. For example, the following class identification criteria (extracted from chapter eight of [31]) become easier to apply when using an “instances first” approach:

- “Redundant classes”. If objects are considered first, it is unusual to come up with redundant classes, since considering classes as sets (of instances exemplified in object diagrams) will prevent putting instances holding exactly the same information in different sets.
- “Attributes”. In our experience, this is one of the main difficulties that are overcome to a large extent when considering instances first. Attribute values appear as single values, not connected to any recognizable domain entity, so that students can easily notice that they are not independent entities but values attached to other entities. This is especially the case when links between instances (that are discussed below) are routinely depicted in instance diagrams, since students easily notice, for example, that a link from a “4.3” coordinate value to a `Point` instance is nonsensical, since the value can not be interpreted in isolation.
- “Vague classes”. These ill-defined classes rarely appear when modelling instances first, since it is difficult to figure out examples of instances of concepts that are not well-defined.
- “Operations”. Instances of classes that should actually be modelled as operations typically include redundant values (or no value at all). Following the example developed throughout [31], instances of `Call` will hold the telephone numbers of the two endpoints that will appear also as values of the customers of the telephone company. Here the instance first approach helps in making evident that creating instances of the candidate class do not lead to

representing any information that should be maintained after the execution of a temporary event. Following the example, no relevant attributes can be identified in the call (provided that the problem is not dealing with the recording of the duration of the calls).

One additional argument in favour of the instances-first approach is that at the beginning of the course, students find it difficult to identify classes from problem statements, so that reasoning by drawing examples provides them with a non-intimidating point of departure to reason about classes.
In general terms, fostering thinking about instances helps in building a precise conceptual framework for object-oriented programs, and serves as a validation tool for the adequacy of concrete information modelling practices as well. Existing textbooks in the area of course explain the class-instance difference, but they actually start with classes as the fundamental construct and objects are introduced afterwards or with less emphasis [32], thus not using systematically example object diagrams to derive from them the class structure. In many cases, this leads to an abstract understanding of classes, disconnected from the instance facet, which becomes problematic at the moment of implementing associations, in which a clear mental model of the instance structure is required. Joe Bergin's site provides many resources that emphasize this view for teaching the initial object orientation concepts (e.g. the diagrams with message passing in the short article "What IS Object-Oriented Programming--Really?"), but unfortunately there is not a systematic object-first approach in the teaching resources about relationships. Consequently, we have devised and applied the following concrete pedagogical practices:

- Example instance (object) diagrams (in UML or any other similar notation) are used prior or simultaneously to class diagram drawing.
- Classes are determined by drawing circles that cover a set of instances that are believed to belong to a particular class (that is, a form of Venn-diagrams is used). This way of representing the instance-class relation is more familiar to students than the dotted-arrow element of UML, and even more easily understood than notations designed for educational purposes like OVAL [27].
- Instances are grouped in classes (sets) by clearly stating the class' *discriminator*[1], that is, the intensional definition of the set in terms of the kind of its instances values (attributes), operations and relationships to other entities. This practice is useful also to identify improper subclasses in advanced phases of the course, since for these classes it is impossible to specify such intensional definitions.
- The use of instance diagrams is motivated as a way to discuss the validity of a given class diagram from the beginning to the end of the course. This fosters thinking about instances, helping students in the assessment of their own solutions.

[1] This concept appears in the UML specification [25], although it is not very often used in practice.

For example, let us consider the following fragment of a problem statement where the nouns (and noun phrases) have been underlined:

> "The Virtual University is planning the automation of its Package Delivery Department (PDD). [...]. Packages are of three types: normal, special and urgent, and the system must be able to produce and schedule for the packages that must be sent to the students of the coming semester. The package for each course must be scheduled for each of the students enrolled in that course, and each package will consist of either a CD or a hardcopy (depending on the course). The application must also schedule the package (plus an instructor guide, sent as urgent) to the tutors of each course. Students and tutors will be able to see the status of their deliveries through the Virtual Campus, by using their user identifier and password."

In most cases, some of the underlined nouns in the text are easily discarded by students. For example, "delivery" is easily recognized as a synonym of "package", while "application" and "system" are typical examples of things not related to the problem space. Conversely, many others are difficult for students to discard if they just think about their "relevance" to the problem. Figure 1 provides the identified classes provided as a solution by a student that took the approach of identifying classes by taking nouns and noun phrases from the text (actually, what is shown is a synthesis of the solutions proposed by several students in an experience that was repeated three times). It becomes evident that more precise criteria are required to come up with reasonable information models.

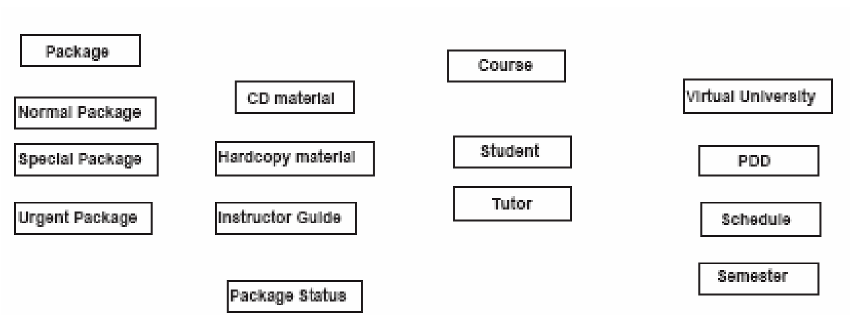

Figure 1: Typical result following a class-first approach to modelling

Figure 2 provides a synthesis of results that can be obtained from students when using an instances-first approach, showing some of the possible links among objects. In this case, the need for a discriminator to describe each set of objects lead most students to narrow the number of classes. For example, drawing objects for each of the "types" of packages shows that instances do not have different values, operations or links from the viewpoint of the scheduling application. In addition, dubious classes like "Semester" or "PDD" manifest themselves as unnecessary for the given setting for two reasons: it is difficult to imagine more than one instance of these classes, and they do not provide useful information for the rest of the objects.

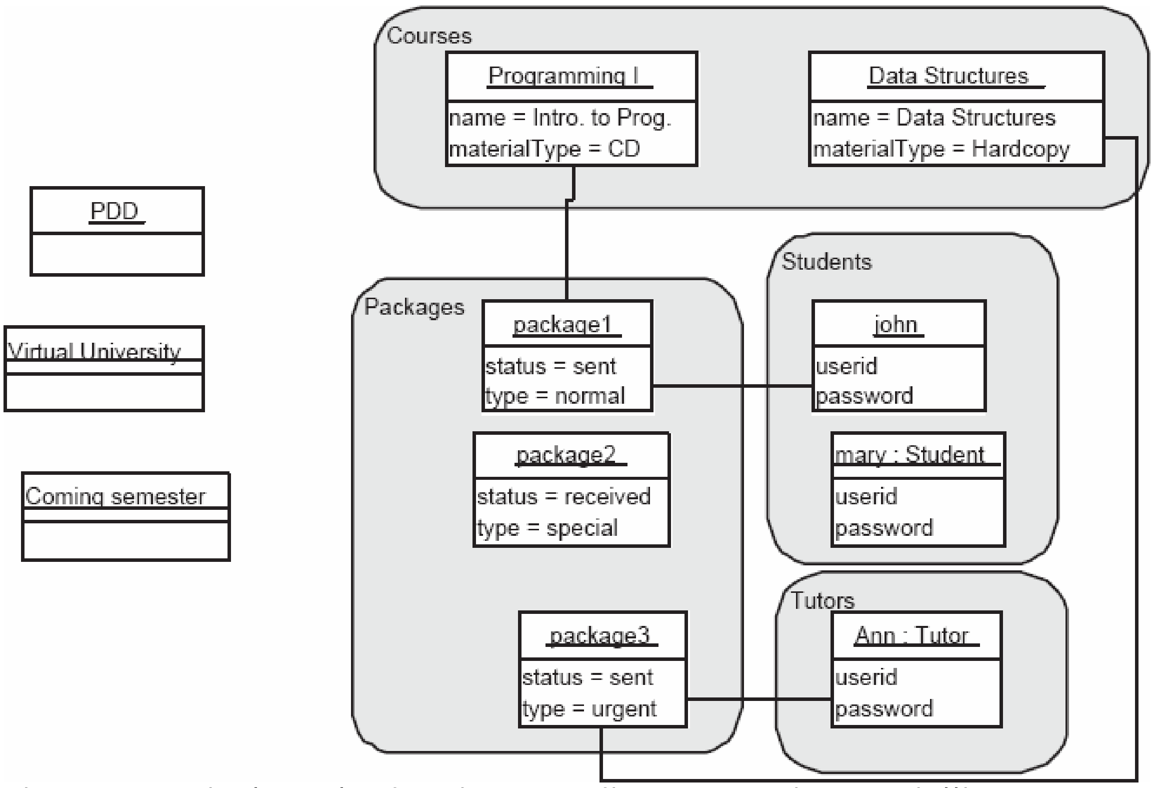

Figure 2: Typical result of an instance-first approach to modelling

Figure 2 shows a more appropriate solution to the described problem statement, since many of the classes in Figure 1 will be attribute-less when implemented in Java, leading to confusion[2].

The different results observed in Figures 1 and 2, when examined from a cognitive viewpoint, evidence that class diagrams are very often used as conceptual maps by students, while the use of instance diagrams forces a strict information modelling approach. The advantage of the latter is that it is better connected to the activity of developing object oriented programs, since conceptual maps are many times closer to lexical information than to an attempt to provide a concrete representational structure.

The described practices help students to think in terms of linked instances, which eventually results in better run-time Java program comprehension. For example, acquiring the mental model of a Java program as a network of instances enhances, simultaneously their comprehension of the garbage collection mechanism and their conceptualization of the fact that several objects can “share” another instance by having separate links referring to it.

[2] Some of these classes may eventually be useful when extending the scope of the problem statement, but extensibility is not a design issue for the moment, since students still lack knowledge about inheritance or interfaces.

In addition to facilitating class identification, fostering thinking about example instances is also useful when deciding whether a given generalization-specialization (gen-spec) relationship is appropriate for a given problem or not. Considering classes as sets of instances that have different discriminators helps in eliminating subclasses that do not provide any additional attribute or operation to their superclasses, since it is easy to identify identical discriminants when drawing sets in an instance diagram.

Associations and Links

Despite the potential uses of UML as a concept mapping tool [11], UML object models are not equivalent to concept maps. Concept maps are a pedagogical tool to approach inquiry about the mental models of students, while the UML, as applied to software modelling, is essentially an activity of information modelling that must come up with representations precise and flexible enough to serve as a basis for the construction of programs.
A major difficulty for many students taking their first steps in class diagramming is their inability to determine the adequacy of the class diagrams they create to the information requirements of a given situation. Let us see a typical example to illustrate it:

> “The University Library is in the process of developing an application to manage book loans and simple queries (by title, library register number and author as the only query criteria). Library users are responsible for damages in the books they borrow, so that a complete historical record of items borrowed and borrowing dates is required to identify the copies of a book that have been damaged.“

Figure 3 shows a solution commonly obtained from students at the beginning of the course.

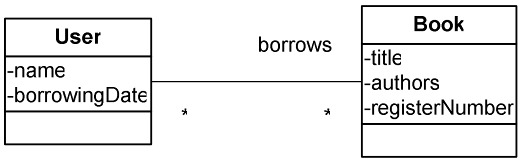

Figure 3. Wrong model for a borrowing history

The main problem with Figure 3 is that it actually appears to be correct since, in some way, it conveys all the information required. However, it can be definitely considered a non-acceptable solution for the given problem statement, due to the simple fact that it is not possible to record more than one borrowing date for each user. In addition, prior to enable the tracking of concrete book copies, a programmer should understand class “Book” as “copy of book”, thus introducing redundant information in the form of repeated strings for “title” and “authors”. Even simple cases like this are difficult to assimilate by students taking their first steps with classes and associations. Alternatively, drawing instances and links provide a good aid both to understand mistakes and to learn to avoid them. Following our previous example, if students try to draw a concrete historical record, they will be forced to include an entity to store the different borrowing dates for the same user, resulting in a model like the one depicted in Figure 4 (Book authors have been omitted for simplicity).

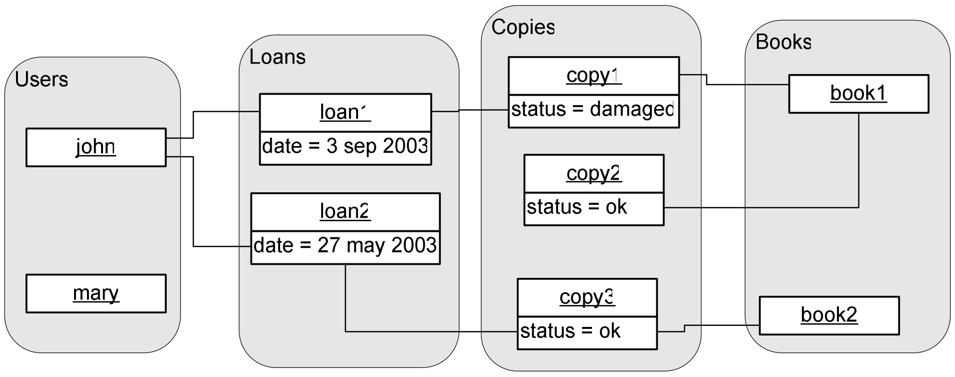

Figure 4. Using instances to derive a model for a borrowing history

The model in Figure 3 has obviously the additional problem of producing redundant information about book authors, and a subsequent decrease in efficiency in searches that use “author” as a criterion. This consideration can also be introduced and explained to students through instance diagrams, since these diagrams make evident such redundancy and the associated growth in the number of instances to inspect for a given query.

In addition to the just described problems regarding expressiveness and redundancy, students that possess previous knowledge in programming and relational databases tend to substitute associations by “primary and foreign keys” explicitly stated in the classes as special attributes. For example, in the above simple User-Book model, this kind of students tend to put a sort of foreign key on Book that references the identifier of the user that borrows the book. Once again, this kind of common misunderstanding can be avoided to some extent by providing an instance-and-link-first approach to modelling. Among all the experiences in the classroom synthesized in this paper, this is the only element that can be considered as a negative impact of previous knowledge of students, which in overall terms has been found to be consistent with positive correlations as reported by Sharp and Griffyth [33].

An additional problem with associations is that Java --- as well as many other OO languages like C++ or C# --- lack an explicit declarative construct providing a default implementation of associations (an example of such declarative elements can be found in the C++ mapping of the ODMG database standard [9]). One-to-one association implementations are relatively easy to master, but one-to-many or many-to-many associations require a deeper understanding of object structures. We have found that for this case, guided common examples (e.g. student-to-subject or book-to-author associations) are convenient for students to get used to association implementation as a coding pattern. Probably the simplest illustrative pattern is that of a one-to-many unordered, unidirectional association, but it is preferable to provide students a complete set of association variants to work out. We found that introducing Java collections early is counterproductive in a objects-first approach, since Java Collection classes introduce a considerable number of concepts like hashing, iterators and the like that interfere with learning basic concepts. This undesired effect, which does not occur in procedural-first approaches, also hampers the acquisition of array programming abilities that are common to all programming languages, regardless of whether they are OO or not.

The additional concrete teaching practices that follow from the above considerations are the following:

- Associations are identified by grouping links with the same purpose and connected entities in example instance diagrams.
- Association cardinality should also be determined through examples, trying for each association to draw a many-to-many example as the more general case.
- Instance diagrams are used for the evaluation of design decisions regarding associations.
- Introduce association implementation through examples using single references or arrays for n-ary association ends.

Examples like those provided in this section are patterns that appear recurrently in many typical problems that can be used at introductory stages. In consequence, using an instance-first approach is, in our view, a very important point to help students develop their object modelling activities.
For students in intermediate or advanced courses, the use of some form of object-pseudocode to sketch operations resulted to be a very positive experience. In particular, the use of the pseudocode together with an object navigation notation syntax similar to that described in [5] with students at UPSAM resulted in a significant improvement of the consistency and overall quality of OO student projects. A simple example of pseudocode fragment that follows our previous example could be:

```
User::hasBorrowed(String name): integer {
    var count =0;
    for each (book in borrowed) do
    begin
        if book.getAuthor().getName()=name then
             count:=count+1;
    end
    return count;
}
```

Such pseudocode fragments help students in noticing the patterns of association navigation before they start to implement the model.

In addition, representations of Java memory models like the one of Gries & Gries [14] can be connected to this instance-focused view to go into details of a program's memory structure. This is particularly useful when teaching recursion by making use of drawings of the program-execution stack.

Teaching Extensibility

Once students master the basic structuring principles of object oriented programming, the concepts related to inheritance and interfaces can be introduced as extensibility mechanisms. In addition, polymorphism allows the provision of a form of genericity that is used to alleviate the lack of built-in genericity constructs in Java. These two related object design issues pose significant learning challenges.

Inheritance as Specialization and Interfaces

According to our experience, and contrary to some instructors' usual expectations, polymorphism and inheritance are not perceived as especially difficult topics by students (which is also consistent with Hyland and Clynch studies [18]). Nevertheless, the subsequent introduction of the concept of *interface* often confuses students and leads to significant comprehension problems. This fact could be explained as a difficulty in discerning the subtle differences between interfaces and abstract classes that only make sense in the context of design as a situated activity and not in the mere translation of a UML diagram to a Java program. In other words, interfaces, as pointed out by Gamma et al. [12], are essentially design-time artefacts, and an understanding of the activity of design is required to properly understand their role.
Even though the decision of introducing interfaces in a first-semester CS1 course may be an arguable decision, it is important to consider that they are required to fully understand the separation of interface and implementation, a basic topic in OO Design[3]. It is difficult to provide novice students with a full understanding of the role of a professional designer, but it is at least possible to describe design situations that emphasize producing design structures with certain quality characteristics, like reusability or minimal coupling. This is the approach taken in the description that follows.

The concept of "interface" in Java plays the role of the concept of "pure abstract classes" that were common practice in C++, but they provide a clear syntactical distinction between the two concepts, i.e. Java introduces an explicit `interface` declaration. Understanding the different purposes of interfaces and abstract classes requires a careful explanation and motivation to realize that design and implementation as two distinct (although closely related) activities. From a design

[3] As mentioned in the "breakdown of topics for software design" of the SWEBOK Guide, http://www.swebok.org/

perspective, interfaces are design artefacts, while classes are related to implementation, which can be summarized in the well-known maxim “design to interfaces” [12]. This way, classes are artefacts involving implementation aspects, while interfaces simply describe required sets of operations, enabling a higher degree of “plug-ability” for classes conforming to those interfaces. The concepts of inheritance and interface extension should also be differentiated since the former entails code reuse, while the latter simply adds new operations to a previous contract.

As a result of our experiences, we have devised and applied the following concrete pedagogical practices:

- Introduce a clear separation of concerns among analysis, design and implementation as three differentiated –but seamlessly integrated– OOSE activities. Analysis is concerned with language-independent and technology-independent information modelling, while design departs from analysis models to come up with efficient and feasible designs in a concrete language. Implementation thus comprises the final coding and testing activities that bring to reality the selected design model.
- Provide examples of different design models obtained from the same analysis model with the aim of fostering critical thinking about structuring OO systems.
- Describe the motivation of classes and abstract classes as design-implementation constructs, and the motivation of interfaces as pure extensibility-oriented design constructs. The use of interfaces to supplement the lack of multiple inheritance mechanisms in Java is in this view a secondary (although important) use of interfaces.

A good explanation and interesting examples of the differences between extending interfaces and extending implementation through inheritance can be found, for instance, in Carroll & Ellis' C++ book [8].
Pedagogical examples of interfaces are difficult to find, and taking examples from the Java library is not always appropriate. For example, illustrating interfaces with `java.lang.Runnable` is confusing in CS1 settings, since it not only requires a minimum background on concurrency, but it is also related to a concrete execution aspect that has little to do with information modelling.
Over the years, we have developed a number of case studies in which interfaces can be introduced to fit an extension of a model previously worked out in detail. One of these examples is that of an inventory system for a company that keeps track of a number of taxonomies of computer and furniture pieces as computers, printers, tables and the like. Once the resulting disjoint class hierarchies have been modelled and programmed, an extension of the problem statement requires the computation of a monetary net value for all the inventory items (for financial accountability purposes, not concerned with the type of the items). Since the type of calculation required to obtain the value of each type of item differs widely, it becomes necessary to redefine a single method, say `getNetValue()`, for each of the classes. Here, the use of an abstract class is not justified since it would

artificially subsume elements that do not possess any kind of common behaviour or structure. Instead, interfaces can be used to provide an alternative, simplified and uniform view of the items without interfering with the inheritance hierarchy. This way, the methods that calculate the global net value can both rely on an interface, say `NetValuableItem`, and take a collection of objects of that type (e.g. `calculateGlobalNetValue(NetValuableItem[] items)`) regardless of the class they belong. Such a model is also prepared to accommodate future types of items. These examples are easier for students to understand, since they demonstrate the use of a design feature within a realistic and common context of use.

Implementing Polymorphism-Based Generic Classes

The Java language did not provide support for genericity (i.e. parameterized classes[4]) until the 1.4 version. Even though the recent 1.5 version provides such support, the use of genericity based on the use of `Object`-typed references still remains an interesting element as it demonstrates the use of polymorphism in object collections.
The lack of genericity can be substituted by some common use idioms to declare parameterized classes. One example of such an idiom is the following:

```
class Stack {
  void push(Object o) {...if (t.isInstance(o)) ...)}
  Object pop() {...}
  ...
  public Stack(Class t) {this.t = t;}
  private Class t;
}
// stack of Strings
String s ="Hello";
Stack st = new Stack(s.getClass());
st.push(s);
...
String s = (String)st.pop();
```

This example requires a previous understanding of both the Java “cosmic class” `Object` and the explicit run-time representation of Java class system, which is organized around a set of classes including `Class`, as shown in the example.
In cases where class parameters are user-defined classes, interfaces provide a much better approach, as illustrated in the following example:

```
interface Priority {
```

[4] Although there exists some extensions like `Gj` (http://www.cis.unisa.edu.au/$\sim$pizza/gj/) that provide genericity, we have discarded its use for now, since there is no guarantee that they will eventually become part of the Java language.

```
   int getPriority();
}
class Student implements Priority {
   public int getPriority() {...}
}
class PriorityQueue {
  Priority queue[];
   void insert(Priority e) {
      if(e.getPriority() < queue[i].getPriority()) {...}
     ...
   }
}
...
p.insert(new Student());
```

Our experience suggest that explaining such kind of programming idioms is important due to the fact that they are extensively used in Java libraries and also pervade Java design practice. For example, Java Graphical User Interface (GUI) programming requires an understanding of such approach to generics since they lie at the hearth of the container and layout manager structures. For that reason, approaches to teaching Java using GUI libraries [26] should beforehand address, at least to some extent, these notions. Nonetheless, these concepts are not straightforward for students even at the fall of the semester, despite being (arguably) easier to understand than C++ templates as pointed out by Adams and Frens [2]. The following strategy can be used in an attempt to overcome the inherent complexity of these concepts:

- Introduce the existence and purpose of the `Object` class as early as possible (e.g. at the time of introducing inheritance). Complement it with foundations about the run-time type information system, e.g. the use and purpose of the class `Class`.
- Afterwards, the implementation of a generic simple container class, similar to `Vector`, can be introduced as an exercise, what is also an example of the use of an `Object`-based approach to genericity.
- Finally, interface-based genericity can be introduced as a form of “bounded genericity” enabling compile-time checking of required interfaces.

These practices help students to acquire the basic toolset to confront the understanding of many aspects of the design of Java class libraries like GUI containers, collections and even the internals of `Object`'s `hashCode` and `equals`.

Design by Contract and Exceptions

Once students are equipped with the fundamental object and algorithm design concepts, more complex assignments will be likely scheduled, raising the issue of *correctness* as an important summative assessment criterion, i.e. the correct operation of the program with regards to the specification should become a central assignment assessment criteria. In this context, the mechanism of Java exceptions is usually taught as an approach to structured handling of error conditions. Unfortunately, this method often leads to misusing exceptions in situations in which they are not appropriate, what in turn leads to a form of (often redundant) *defensive programming* that hampers code readability and imposes students significant and unnecessary work overloads.

The *Design by Contract* (DBC) philosophy, described by Meyer in [23], provides a comprehensive conceptual framework to avoid such misunderstandings. DBC emphasizes the fact that software can be considered a formal agreement between classes and their clients (users), expressing each party's rights and obligations. According to the DBC design approach, assertions are used for every method to determine both the outcomes of the corresponding messages (i.e. postconditions) and the requirements on the input values that should be met in any method invocation (i.e. the preconditions) to guarantee the specified outcomes. In addition, invariants can be used to characterize the possible valid states of the objects of a given class. DBC provides a clear separation between correctness and robustness, so that correctness is relative to the specification (through assertions) of the software, while robustness is concerned with the behaviour of the software when an unexpected error condition occurs. Exceptions, as proposed by Meyer, are a mechanism to deal with robustness, not to check correctness. This has important implications for teaching, since correctness is tied to student's competence in designing and programming algorithms, while robustness can be considered as a quality factor that is supplementary to correctness. This way, a clear separation is drawn between defects resulting from incorrect decisions of the student (programmer) and errors as unexpected run-time events that should be gracefully handled.

However, introducing formal assertions in a CS1 course imposes a significant burden in the curriculum and its nuances are not pedagogically appropriate (as pointed out in [7]). This especially due to the introduction of new constructions (like the `old` element in Eiffel) and the related conceptual notions of Hoare triples. These elements introduce pre- and post-conditions as a replacement of defensive programming as described by Meyer [23], which reason on the states of objects and the changes operated in them by method calls. Due to the schedule limitations of one-semester courses, a simplified account of DBC should be used. The main objective is to avoid the use of `Java` extensions that provide the typical elements of DBC like `iContract` [21][5] (in contrast to the desires expressed by Hosch [17]) since they introduce many new elements that are not trivial to master for students. Among these elements, expressions on the previous state of the object and existential and universal quantifiers can be found.

[5] An `assert` sentence has been recently added to Java syntax, providing a limited way to check conditions anywhere in the code of a class. After an initial attempt, we discontinued its use since it led students to use it as a control flow statement, producing confusion with basic structured programming practices.

The practical setting for introducing essential DBC concepts into CS1 OOP courses can be summarized in the following classroom tactics:

- When introducing algorithms, the practice of specifying separate comments informally describing pre- and post-conditions can be proposed. To do so, fictitious *javadoc* tags `@pre` and `@post` can be introduced just to write down natural language comments.
- Instructor-led discussions about the correctness of code fragments must emphasize the role of specifications as the contract used to evaluate defects, both from the viewpoint of a given class and from its clients' viewpoint. Students can take the role of either the class viewpoint or its clients, making clear the different perspectives that take place in real-life programming situations. For example, from the viewpoint of a class writer, preconditions should be as strong as possible (as illustrated in [23]).
- The separation of input and validation in helper classes is introduced as a basic design technique (as described in [20]), so that these helper classes are used as filters for user input that prepare input data to fit the preconditions of the processing classes.
- Once students begin to be assigned relatively complex programs involving several classes, and provided that they possibly work in small teams, the concept of unit testing is addressed in connection with pre- and postconditions. This way, students become used to write small unit testing modules aimed at checking them. To do so, either simplified unit testing frameworks like `JUnit` (see for example [12]) or an informal approach can be used.

Kernighan and Pike's book [19] contains some examples regarding this informal way of approaching contracts. The benefits of the outlined approach to DBC became evident in the last two semesters of our classes at UC3M as a substantial reduction in the length of methods, mainly caused by the reduction in defensive input parameter-checking code. Additionally, the revision of student's assignments took less time thanks to the improved readability and more explicit intentions of the methods. The *javadoc* tool and its widespread use by practitioners and students provides a valuable tool in introducing code commenting practices as those described above, despite being syntactically separated from the language.

It has been observed also that explaining the rationale for the above tactics to the students could not be enough to foster their classroom use in some situations [13], since they're often viewed as time-consuming activities. In consequence, they must be taken into account in grading the students' programs, so that they are perceived as an integral part of the software process.

Final Remarks

A number of problematic issues regarding the teaching of object-orientation concepts with Java in CS1 courses have been raised in this paper, complementing known pitfalls about language elements reported elsewhere. They have been identified as critical elements in the connection of two forms of knowledge –object concepts and object languages– that require building strong links between them from a constructivist perspective, as described in [15]. These issues lead also to a consideration of the adequacy of introducing an early distinction of design and implementation practices, so that design approaches like DBC and concepts like interfaces can be properly put into the context of real programming practice.

Lessons learned from teaching Java have led us to emphasize the role of instance diagrams for the initial comprehension and reasoning about OO models and programs. Once the basic structure of object programs has been learned, the focus on extensibility --- as the main advantage of OO software with regards to previous approaches --- requires a clear distinction between design and implementation issued, manifested in the different uses of classes (abstract or concrete) and interfaces. Finally, approaches to correctness (arguably *conditio sine qua non* assessment criteria require some kind of rationale for unit testing and checking code. A streamlined version of the DBC approach has demonstrated effective to that end, helping also in the clarification of the role of exceptions as a way to deal with failures, and not with defects.

Bibliography

[1] ACM/IEEE (2001).*Computer Science Computing Curricula 2001 -— Final Report* —-, December 15, 2001

[2] Adams, J., Frens, J. (2003). Object centered design for Java: teaching OOD in CS-1. *ACM SIGCSE Bulletin 35(*1*)*: 273--277.

[3] Alphonce, C. and Ventura, P. (2002). Object Orientation in CS1-CS2 by Design. *ACM SIGCSE Bulletin 34*(3): 70--74.

[4] Bidle, R., Tempero, E. (1998). Java Pitfalls for Beginners. *ACM SIGCSE Bulletin 30(*2*):* 48--52.

[5] Blaha, M., Premerlani, W. (1997*). Object-Oriented Modeling and Design for Database Applications*. Prentice Hall.

[6] Booch, G. (1994). *Object-oriented Design with Applications*. Benjamin Cummings.

[7] Buck, D. and Stucki, D. (2000). Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on levels of Cognitive Development. *ACM SIGCSE Bulletin 32*(1): 75--79.

[8] Carroll, M.D., Ellis, M.A. (1005) *Designing and Coding Reusable C++*. Addison-Wesley.

[9] Cattell, R.G.G., Barry, D.K., Berler, M., Eastman, J., Jordan, D., Russel, C., Shadow, O, Stanienda, T. and Velez, F. (2000) *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers.

[10] Clark, D., MacNish, C., and Royle, G.F. (1998). Java as a teaching language --- opportunities, pitfalls and solutions. *ACM SIGCSE Bulletin XX*(X): 173--179.

[11] Ferguson, E. (2003). Object-oriented concept mapping using UML class diagrams. The *Journal of Computing in Small Colleges 18*(4): 344--354.

[12] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design Patterns: elements of reusable object-oriented software,* Addison-Wesley.

[13] García, E., Sicilia, M.A., Aedo, I., Díaz, P. (2002). An experience in automated unit testing practices in an introductory programming course. *ACM SIGCSE Bulletin 34*(4): 125--128.

[14] Gries, P., Gries, D. (2002). Frames and folders: a teachable memory model for Java. The *Journal of Computing in Small Colleges 17*(6): 182--196.

[15] Hadjerrouit, S. (1998). Java as a first programming language: A critical evaluation. *ACM SIGCSE Bulletin 30*(2):43--47.

[16] Hadjerrouit, S. (1999). A constructivist approach to object-oriented design and programming. *ACM SIGCSE Bulletin 30*(3): 171–-174.

[17] Hosch, F. (1996). Java as a First Language: An Evaluation. *ACM SIGCSE Bulletin 28*(3):45--50.

[18] Hyland, E. and Clynch, G. (2002). Teaching with Java: Initial experiences gained and initiatives employed in the teaching of Java programming in the Institute of Technology Tallaght. *Proceedings of the Conference on the Principles and Practice of programming*, 101--106.

[19] Kernighan, B.W. and Pike, R. (1999). *The Practice of Programming*. Addison Wesley.

[20] Koffman, E., and Woltz, U. (1999). CS1 Using Java Language Features Gently. *ACM SIGCSE Bulletin 31*(3):40--44.

[21] Kramer, R. (1998). iContract -- The Java Design by Contract Tool. *Proceedings of the 26th Uonference on Technology of Object-Oriented Systems* (TOOLS--USA).

[22] Martin, P. (1998). Java, the good, the bad and the ugly. *ACM SIGPLAN Notices 33*(4): 34--39.

[23] Meyer, B. (1997). *Object Oriented Software Construction*. Prentice Hall.

[24] Object Management Group (OMG). (2003). *Unified Modeling Language Specification. Version 1.5 March 2003*, doc. number formal/03-03-01.

[25] Ourosoff, N. (2002). Primitive Types in Java Considered Harmful. *Communications of the ACM 45*(8): 105--106.

[26] Raab, J., Rasala, R., Proulx, V.K. (2000). Pedagogical power tools for teaching Java.*ACM SIGCSE Bulletin* archive *32*(3): 156--159.

[27] Raner, R. (2000). Teaching object orientation with the Object Visualization and Annotation Language (OVAL). *ACM SIGCSE Bulletin 32*(3):45--48.

[28] Rajaravivarma, R., and Pevac, I. (2003). When to introduce objects in teaching Java. *Proceedings of the 35th Southeastern Symposium on System Theory*.

[29] Rodríguez, E., Sicilia, M.A., Gallisa, E., Dodero, J.M. and Alvarez, J. (2003). Continuous Assessment in Online-Teaching: the Case of an Object-Oriented Programming Course. *Proceedings of the 2nd International Conference on Multimedia and Information & Communication Technologies in Education* (m-ICTE 2003).

[30] Rosenberg, J. and Kölling, M. (1997). I/O considered harmful (at least for the first few weeks). *Proceedings of the second Australasian conference on Computer science education*, 216--223.

[31] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. (1991). *Object-Oriented Modeling and Design*, Prentice Hall.

[32] Winder & Roberts. (1998). Developing Java Software, Wiley.

[33] Sharp, H.C. and Griffyth, J. (1999) The Effect of Previous Software Development Experience on Understanding the Object-Oriented Paradigm, *Journal of Computers in Mathematics and Science Teaching*, 18(3), 245-265.