Booch's Ada vs. Liskov's Java: Two Approaches to Teaching Software Design

Ehud Lamm

The Open University of Israel, Max Rowe Educational Center, P.O.B 39328
Ramat-Aviv, Tel-Aviv 61392, Israel
ehudla@openu.ac.il

Abstract. We study two textbooks for teaching undergraduate software engineering, both focusing on software design and data abstraction in particular. We describe the differences in their didactic approaches. We analyze how the subject matter is influenced by the choice of programming language, one book using Ada and the other book using Java. Java is a relatively new candidate for teaching software engineering. How does it compare to Ada?
Keywords: Education and training, evaluation and comparison of languages, object-oriented programming, software development methods.

1 Background

The undergraduate software engineering course at The Open University of Israel, called *Software Engineering with Ada* is based on a Hebrew translation [2] of Booch's book of the same name [1].[1] This course has been offered since 1989. I have been involved in teaching this course since 1999. The course is elementary and focuses on classic programming techniques in Ada (e.g., designing ADTs, generic units, and tasking). Since the book deals with Ada83 the course study guide includes a detailed chapter about tagged records and inheritance in general. The course is practical and the students are expected to do a fair amount of coding in order to solve their problem sets.

We are in the process of thinking whether this course provides the best introduction to software engineering for our computer science students. Changes to the curriculum make it necessary to recheck how the course interacts with other courses (e.g., the introduction to CS course and the data structures course).

In this paper I will try to analyze the differences between the existing course and a possible replacement based on Liskov's *Program Development in Java* [10] which is used in some other schools (e.g., MIT).

2 Reasons for Change

Booch's book is example oriented (the book includes five extended examples), but lacks theoretical foundations. Teaching the course for several semesters led

[1] A third edition of *Software Engineering with Ada* coauthored by Doug Bryan was published in 1993.

me to believe that the lack of theoretical grounding makes learning harder for students (e.g., the notion of an ADT *invariant* is not defined nor is the algebraic interpretation of ADTs). Essentially, we want students to come and appreciate the notion of software abstraction, but this concept is never explained in detail or analyzed sufficiently.

A more practical reason for replacing the current course is, alas, the use of Ada. First, our introduction to CS which was previously Pascal based now has a C++ version[2] and a Java course is planned. Students arriving at the software engineering course have a hard time adjusting to the Pascal-ish syntax and terminology of Ada (e.g., the syntactic distinction between procedures and functions).

Second, students want to learn skills that are useful in industry. Their impression is that Ada is hardly ever used, which makes them question the choice of language (causing endless debates). Since the software engineering course is not mandatory, many students choose not to take it, preferring other advanced programming courses. This is problematic, since several important concepts are only taught in the software engineering course. The paradoxical result is that many students who plan to work as programmers and software designers in industry, skip the course most relevant to their needs.

3 Course Goals

In this section I discuss the *main* objectives I see for the undergraduate software engineering course.

The course should develop program design skills, teaching basic software engineering concepts needed for exploring and analyzing software design alternatives.

Students are expected to have limited programming experience, and are likely not to have been part of an industrial scale software project. Most students take the introductory software engineering course after completing only two programming courses: Introduction to Computer Science, and the Data Structures and Algorithms course. Thus, the course should provide opportunities for practical experience, in the form of exercises and – if possible – a small project. Theoretical concepts (e.g., information hiding, robustness) should be introduced and explored in the context of the programming assignments. The practical exercises should provide opportunities for software testing and include small maintenance tasks.

As this is an academic course, students are expected to be able to reason about simple design issues using accepted terminology, and to express their reasoning succinctly and precisely.

[2] C++ is used in this course as a type safe C. No OOP is taught.

3.1 Software Abstractions

The course focuses on modularity. Students should be able to design a module structure from a given problem statement, and design adequate module interfaces.

Thus, the course elaborates on the notion of software abstractions, and in particular data abstractions. The students are expected to feel comfortable with procedural abstractions before taking the course, but the term itself, and the connection to software abstraction in general, should be introduced. Control abstractions (e.g., non deterministic choice) are for the most part beyond the scope of the course.[3]

The course is not an exhaustive survey of software engineering concepts and methodologies, nor does it cover the full scope of software design techniques. Instead, it focuses on the fundamental notions of abstraction and modularity.

4 Didactic Approach

Booch's and Liskov's books follow a similar didactic approach. They give small code examples (snippets), and some larger scale design examples. Some of the noticeable differences:

- **Explicit vs. implicit learning.** Liskov defines several concepts that help reasoning about software abstractions. The most important ones are: adequacy ([10], sec. 5.8.3), invariant, and abstraction function.
- Throughout the book Liskov gives tips and rules for effective design (these are highlighted by using grey background). These can be seen as beneficial, but can lead students to a cookbook approach to programming. They should realize it is not enough to "follow the rules."
- Booch gives a useful list of software engineering goals and principles.
- Liskov dedicates a whole chapter to procedural abstraction ([10], chp. 3)

In our experience students find the existing course quite difficult. Some examples of tasks students find difficult:

- Designing abstraction interfaces. Not surprisingly, this is the main obstacle for students. Booch talks about this extensively, in each of the design problems. Liskov provides fewer examples, but summarizes the issues nicely in sec. 5.8, *Design Issues*, which discusses mutability, operation categories and adequacy.
- Proposing alternative designs. Students often fall in love with the first design they think of, and find it impossible to think of competing designs for solving the problem at hand.

[3] Control abstraction is often discussed in courses about programming language theory and design.

– Comparing different designs. Even when presented with two alternative designs many students find it hard to compare the designs, unless given a list of specific criteria (e.g., "How sparse does the matrix have to be, so as to justify the space overhead of using pointers?")
– Succinctly expressing their design rationale, especially doing so precisely.

5 Language Differences

The main question this paper tries to attack is the impact of the choice of Java versus Ada on the ease of teaching and on achieving the goals of an undergraduate software engineering course. We do this by comparing language features we find relevant.

Though used mainly as a vehicle for introducing language agnostic programming techniques, the programming language chosen is a major influence on the design of the course, since the order of presentation has to be consistent with relations between language constructs (e.g., in Ada, tagged types rely on packages).

It should be noted immediately that language comparisons are problematic, inflammatory, and often biased. Comparing languages by listing language features is in itself a questionable technique, seeing as there are often several possible language features that can be used to achieve any one design goal – each feature with its own advantages and disadvantages. We are not trying to compare the languages in general. We highlight language features that seem important from our experience teaching software engineering with Ada.

5.1 Problems with using Java

Since we are thinking of moving from Ada to Java, we start by noting Ada features we will miss.

The type system. Perhaps the most glaring difference is in the type system. Ada has a rich and expressive type system that structures the whole language. Ada makes it easy to understand the software engineering benefits of strong typing. The integration of the type system with generic units and inheritance is very enlightening (e.g., by showing the difference between static and dynamic polymorphism).

Java doesn't support records, which are so basic that Liskov has to define and explain them, in the chapter dealing with software abstraction ([10], sec. 5.3.4). They are, of course, a simple language feature obvious to any student with a Pascal, C, or Ada background.

Reference Semantics. Java uses reference semantics. Reference semantics are especially bothersome in the context of a course dealing with data abstraction, since they easily lead to abstraction breaking. Liskov is, of course, well aware of this problem. Section 5.6.2 deals with "exposing the rep" and gives examples of this problem. Chapter 2, which is a quick Java refresher, also covers the pertinent Java semantics.

Genericity. Another crucial feature is Ada's support for generic units. Indeed, Liskov dedicates a whole chapter to the software design notion of *polymorphic abstractions* ([10], chp. 8), whereas Booch's related chapter is focused on the Ada feature generic units ([1], chp. 12). It is important to note that Booch deals with Ada83, and thus does not talk about using inheritance for polymorphism. This omission can be fixed with supplemental material, which indeed we provide for our students. Java's lack of genericity, however, is harder to overcome. If we move to Java, and use Liskov's book, it is likely that when genericity is added to Java [9], we will have to provide supplemental material covering genericity, until the book is revised or a new book is chosen.

Teaching genericity (i.e., *parameterized polymorphism*) before introducing inheritance seems to ease the understanding of both topics. The compile-time nature of Ada generic units makes them an ideal stepping stone to the more complicated mixture of dynamic and static properties that exists in the presence of inheritance (e.g., class wide programming and dispatching).

The importance of polymorphism, from a software engineering point of view, is apparent. It is an important design technique, that enhances reuse, clarity and flexibility. The question is whether polymorphic abstractions based on inheritance as provided by Java, are good enough for the purpose of our software engineering course, or is a templating mechanism as provided by Ada generic units essential.

Genericity is a tool for building parameterized units. Parameterization is an important abstraction method. Ada generic units, indeed the Ada syntax, emphasize this fact.[4]

Coupled with tagged types for dynamic polymorphism (i.e., dispatching), the Ada model which allows for nesting of generic and non-generic program units is very appealing, see figure 1 (a different scenario, which is handled with exclusive use of inheritance can be seen in the exercise shown in figure 2. Students are expected to be able to understand the differences between the two designs.). However, it seems to me that the treatment of polymorphic abstractions in Liskov's book is acceptable. Much as I like Ada's generic units, I think we can achieve the course goals without them.[5]

Concurrency. Ada provides higher level concurrency constructs than Java. Booch takes advantage of this and dedicates two chapters to tasking (a technical chapter and a detailed design problem). Liskov does not cover concurrency.

By teaching Ada's tasking constructs we are able to explore interesting kinds of software abstractions. For example, we show active objects (e.g., a self sorting array for fast lookup). Ada tasks are also helpful for showing simple parallel algorithms.

---

[4] Visible discriminants of private types, are another useful way to introduce students to parameterization.

[5] Java's interfaces will, of course, come in handy.

In this exercise students implement a *Set* ADT.

1. Write a generic package, exporting an abstract tagged type Set. The type should provide the standard Set interface. The type of the elements in the set will be provided as a generic parameter.
2. Write a child package, in which you inherit from the abstract Set type, and implement the set type as a sorted linked list. *Note: The < parameter is only required by the child package.*
3. Compare the uses of inheritance and genericity in this exercise.

Students are often asked to use an ADT they wrote previously (e.g., a TRIE) to implement the abstract type (layered design).

Fig. 1. Genericity and Inheritance

In this exercise students implement a polymorphic *Stack* ADT.

1. Write a package exporting an abstract tagged type Stack. The type should provide the standard Stack interface. The Stack should be polymorphic: it should be possible to store values of different types in the same Stack *Note: The exercise explains how to achieve this by defining a parent type for all items. Genericity is used to wrap any type, in a type derived from this parent type.*
2. Write two child packages, in which you inherit from the abstract Stack type. One package will implement the Stack using a linked list. The second package will implement the Stack using an array.
3. Inheritance is used for two reasons (and to achieve two different goals) in this exercise. What are these uses, and how does inheritance help achieve them?

Fig. 2. Interface inheritance, and heterogeneous collections

Combined with inheritance and other Ada language features, Ada tasks make it easy to show simple patterns for concurrent programming (e.g., thread pools, synchronization objects [3] etc.)

It is possible to do similar things in Java, but Ada task and protected types make implementing them much easier.

Miscellaneous. Two of the mini-projects I give involve building program analysis utilities. In one, the students have to traverse an Ada source code tree, and collect interesting metrics (e.g., number of procedures, number of with statements etc.). In the second project, the students have to display the package dependency graph. Some of the students were shown how to use ASIS (*Ada Semantic Information Specification*). This reduced the effort needed to complete the project, and learning the ASIS API was a useful learning experience. Though similar tools for Java do exist, ASIS is standard Ada, and the Gnat implementation is relatively easy to use.

5.2 Problems with using Ada

Perhaps the most problematic thing about using Ada is the relative lack of freely available tools, as compared to C++ and Java. Ideally, students learning about software engineering should come in contact with various kinds of CASE tools. Because of the nature of the course, I am thinking specifically about IDEs, testing tools and frameworks, measurement utilities, diagraming and code generation tools, and pretty printers.

There are tools of these kinds for Ada, of course. It would be nicer if there was a larger choice, so we could have students compare tools etc. The main problem, however, is to package a *stable* set of quality tools.

Tools we currently use are: Gnat and AdaGide (our basic setup). Students occasionally use GRASP for pretty printing.

Reusable Code. Available Ada libraries are not always easy to find and use. I would like to encourage students to use reusable code as much as possible. However, of the available libraries some are pretty hard to use, and require knowing advanced Ada techniques, which students learn towards the end of the semester. Perhaps the best example is using the Booch Components (the Ada95 version), which require knowing about tagged types and nested generic instantiations. This is a shame, since using standard data structures is a classic reuse scenario, which comes naturally when learning about data abstraction. For example, a standard exercise we give is building a priority queue, which is based on an array of linked lists of elements. Naturally, standard linked lists can be reused, thus saving the students time, and teaching them about the process of reusing publicly available code. Indeed, I want students to compare several reuse scenarios: The priority queue can be seen as an array (indexed by priority) of queues, rather than as an array of lists. It can also be seen as a map from priorities to queues. Having a standard collection library, as part of the Ada

standard library in Ada200X, would make giving such exercises a bit easier, and could also help improve Ada textbooks.

Students are encouraged to explore the Internet for reusable code. In my experience this often leads to better educational results than giving students code to reuse. The main advantage with this approach is that it teaches about some of the problems with achieving reuse in real life (e.g., finding the appropriate package is not easy, not all kinds of documentation help clients use a package, packages you find may interfere with each other etc.)

Liskov's book makes use of the extensive Java library as a source of examples (e.g., [10], sec. 7.10).

I currently recommend to students wanting to build GUIs, to use JEWL [6] (some students have used GWindows). A simplified unit testing framework was recently made available to students.

I have plans for using AWS ("Ada Web Server") and XML/Ada [6] for extended exercises (mini-projects) which will involve building web based applications (e.g., a browser based user interface, and an RSS news aggregator[7]).

Interfaces. One of the important objectives of the course it to teach students to write interface oriented code. By that we mean polymorphic code that works for a variety of abstractions that supports a common interface. A simple example may be printing the contents of a container (e.g., tree) using an active iterator. The print procedure should be agnostic to the details of the container, and be compatible with any container that provides an iterator with the required set of operations.

Ada provides quite a few ways to specify the interface such a routine relies on. At times generic formal types are enough (e.g., when the routine works for any *array*). Another technique is to pass a private formal, and specify additional formal subprogram parameters. The interface can be encapsulated as a signature package (and passed as a formal package parameter). Another approach is to represent the interface as an abstract tagged type, in which case the routine may at times be written as a class wide routine (however, the lack of multiple inheritance makes this approach problematic).

All these techniques have their uses, of course.[8] But the fact that the basic and essential concept of "interface" can be represented programmatically in so many ways can be confusing to students. It is not possible to restrict our attention to only one of the relevant Ada constructs, since their uses are quite different. Specifically, package specifications must obviously be introduced, but this interface definition is not appropriate for parameterization purposes. Likewise, signature packages cannot be the only interface definition discussed, because packages in Ada are not first class and this prevents useful design techniques [7]. The usefulness of abstract tagged types as interfaces is marred by the lack of multiple inheritance.

[6] Both available from *http://libre.act-europe.fr*

[7] See *http://radio.userland.com/whatIsANewsAggregator*

[8] We, in fact, try to teach students to choose the most appropriate technique in each case.

Java, unlike Ada, provides a special language construct for interfaces.

Constructors and Destructors. Data abstractions often require special set up and cleaning code. This functionality can be implemented by defining constructors and destructors, which are automatically called by the language. In Ada this is done using `Ada.Finalization`. This requires knowledge of tagged types. Combining controlled types with genericity is awkward, since the generic units must then by instantiated at library level. The rationale for this technical aspect of the language is lost on students, and they find the compiler error messages hard to understand.[9]

When students first learn about building ADTs, they often want to overload assignment. Instead of doing so they are told to write a `Copy` routine, since at that point they haven't yet learned about tagged types. When they are taught about `Ada.Finalization` it turns out that changing their packages to use controlled types requires a fair amount of work.

There are situations where manual cleanup is impossible or hard to do correctly, and the use of finalization is especially important (e.g., multitasking programs).

Coupled with the fact that Ada doesn't mandate garbage collection[10] the issue of manual memory management must be addressed. In a sense, this is a good thing, since memory management is something programmers are expected to understand. However, the effort to do it correctly can be substantial.[11]

Visibility. It has been argued that the methods of controlling visibility of methods and fields to subclasses (i.e., types extended via inheritance) are less than ideal. Other languages, including Java, support designating properties as either public, private or protected. The `protected` designator granting visibility to subclasses.

In Ada, you achieve this sort of visibility either by using child packages (that have visibility over the parent's private part) or by including the derived type in the parent package.[12] Students may find these techniques confusing and awkward. Moreover, they are taught to include as few declarations as possible in the `private` part of package specifications (to avoid recompilations) and put helper routines that are implementation specific in the body of the package. When hierarchical units are introduced, they are told that in order to provide for easy extendibility some routines are better declared in the private part, so as to allow child unit visibility. Another classic example is the state variables of an abstract state machine, which are defined in the body, but must be moved to the

[9] Recall that the course deals with software engineering principles, and we do not have enough time to explore subtle language specific issues.

[10] Students are often confused by this, failing to grasp the difference between what the language specifies, and implementation particulars.

[11] Garbage collection is known to help with modularity, due to the complexity of manually managing memory between module boundaries (e.g., because of aliasing). Discussing this issue is beyond the scope of this paper.

[12] A more complicated scenario involves genericity.

`private` part, if the package is to be extended. These situations are confusing, because when you see a declaration in the private part, there is no way to tell if it is there by mistake, or whether it was placed there for a reason.

The appropriate uses of the explicit `protected` definitions in Java are discussed in [10] (sidebar 7.5, for example). Personally, I find the Ada model more flexible, however this issue was raised by instructors, and I did encounter students that were confused by it.

Perhaps hierarchical units should be introduced immediately following the chapter on packages (as done in [5]), instead of being introduced after tagged types, the way we do today. This may help overcome some of the problems discussed above. However, Ada's lack of *explicit* `protected` definitions is a language expressiveness and readability concern.

Miscellaneous. Ada95 was developed from Ada83 and is mostly backwards compatible. This causes redundancy. We teach implementing ADTs as encapsulated private types exported from packages, making no use of inheritance. Students are then shown how to implement ADTs that can be derived from, leaving them wondering when to use non-tagged ADTs.

Ada's inheritance is not class based. This has important advantages (e.g, no need for special treatment of friend classes, binary methods are more readable etc.) However, since the students are likely to move to a class based object oriented programming language (either C++ or Java), this can be seen as a liability. However, from a didactic point of view, this difference is a bonus, since it makes students review their implicit assumptions, especially those who have prior OOP experience.

Ada exceptions, that are used as part of an abstraction interface, are declared in the package specification. However, there is no way to specify in Ada which exceptions are raised by each routine. We found that requiring students to specify this information in comments helps them think about using exceptions as part of an interface. It would be better if this information could be specified in the code and checked by the compiler.[13] This would also improve integration between exceptions and signature packages.

5.3 Features missing from both languages

Both languages are missing some features that would have enriched the course. First and foremost is *Design by Contract* (DbC) [8] which is especially important for a course focused on data abstraction. There are several DbC tools for Java, but contracts are not part of the language. Indeed, Liskov consistently provides preconditions and postconditions, in the form of source code comments (Booch doesn't do this).

Other language features that would have been helpful: anonymous and first class functions, and laziness. These features help support a style of programming

[13] Java has checked and unchecked exceptions, which give the programmer finer control on the way exception consistency is checked ([10], sec. 4.4.2).

closer to functional programming. For example, they help create high order functions, which are useful and interesting abstraction and reuse mechanisms. It is possible to program in this style in both Ada and Java, but in both languages this style of programming is quite awkward. Laziness provides an alternative and often useful solution to the problem of iterating over data structures (a common data abstraction issue, for which Liskov dedicates an entire chapter, [10] chp. 6). Another classic use is for defining infinite data structures (e.g., streams), which can be used to implement powerful abstractions, and which are quite hard to implement otherwise [4].

6 Conclusions

The books discussed [1, 10] are very different in outlook. Liskov's book is much more suited for academic use, as it provides tools for reasoning about data abstraction, and software abstraction in general. However, this difference is not the result of the choice of programming language.

The problems encountered with Ada are mainly the popularity of the language, and library and packaging issues. The problems with Java are with essential properties of the language (e.g., reference semantics). Though Java is a new player in the field, Ada still seems at least as good for teaching software engineering.

The decision whether to make the move to Java has not been made yet. If it will be made, it will not be the result of shortcomings of the Ada language.

Acknowledgments. I had valuable discussions with instructors teaching Open University course 20271, *Software Engineering with Ada*.

References

[1] Grady Booch. *Software Engineering with Ada*. The Benjamin Cummings Publishing Company, 2nd edition, 1987.

[2] Grady Booch. *Software Engineering with Ada (Hebrew)*. The Open University of Israel, 1989. Translated from the second edition (1987).

[3] Alan Burns and Andy Wellings. *Concurrency in Ada*. Cambridge University Press, 2nd edition, 1998.

[4] Arthur G. Duncan. Reusable Ada libraries supporting infinite data structures. In *Proceedings of the annual ACM SIGAda international conference on Ada*, pages 89–103. ACM Press, 1998.

[5] John English. *Ada 95: The Craft of Object-Oriented Programming*. Prentice Hall, 1996.

[6] John English. JEWL: A GUI library for educational use. In Dirk Craeynest and Alfred Strohmeier, editors, *Reliable Software Technologies – Ada-Europe 2001*, volume 2043 of *Lecture Notes in Computer Science*, pages 266–277. Springer-Verlag, May 2001.

[7] Ehud Lamm. Component libraries and language features. In Dirk Craeynest and Alfred Strohmeier, editors, *Reliable Software Technologies – Ada-Europe 2001*, volume 2043 of *Lecture Notes in Computer Science*, pages 215–228. Springer-Verlag, May 2001.

[8] Ehud Lamm. Adding design by contract to the Ada language. In Johann Blieberger and Alfred Strohmeier, editors, *Reliable Software Technologies – Ada-Europe 2002*, volume 2361 of *Lecture Notes in Computer Science*, pages 205–218. Springer-Verlag, June 2002.

[9] Sun Microsystems. *JSR 14 - Add Generic Types To The Java Programming Language*. http://jcp.org/jsr/detail/14.jsp.

[10] Barbara Liskov with John Guttag. *Program Development in Java. Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.