Interfaces First (and Foremost) With Java

Paolo A. G. Sivilotti
Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
paolo@cse.ohio-state.edu

Matthew Lang
Mathematics and Computer Science
Moravian College
Bethlehem, PA 18018
lang@cs.moravian.edu

ABSTRACT

Abstraction is a critical concept that underlies many topics in computing science. For example, in software engineering, the distinction between a component's behavior and its implementation is fundamental. Java provides two constructs that correspond to precisely this distinction: A Java interface is a client's abstract view of a component's behavior, while a class is a concrete implementation of that same component. We have developed a course that introduces Java while following a discipline of diligently decomposing *every* component into these two separate linguistic elements. In this course, interfaces are given the same prominence as classes since both are needed for a complete component. This approach is helpful to students by providing: (i) a clear manifestation of the role of abstraction in software systems, and (ii) a framework that naturally motivates many good coding practices adopted by professional programmers.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*

General Terms

Design, Languages

Keywords

abstraction, concrete state, behavioral specification

1. INTRODUCTION

Abstraction is a key concept in computing science and software engineering [3,5–7]. Students encounter it, in some form, in practically every major topic including architecture, operating systems, complexity, data structures, and programming languages. Indeed, the ACM 2001 Computing Curricula recognizes the importance of abstraction at the pedagogical core of our discipline [22]. As one of its guiding principles, the Computer Science volume states (p. 12):

> All computing science students must learn to integrate theory and practice, to recognize the importance of abstraction, and to appreciate the value of good engineering design.

Ironically, the ubiquity of abstraction may actually be a barrier to its recognition and appreciation by students. It is such a common theme that educators may use it casually, without explicitly drawing attention to it.

We have developed an introduction to Java course based on the principle of deliberate and explicit application of behavioral abstraction. Although it is a first course in Java, it is not designed as a first course in programming. Students have completed an introductory sequence (using C++) before taking this course. Beyond teaching students about abstraction, this approach has the additional benefit of motivating and explaining several important (but subtle) concepts and professional best practices in Java.

From the beginning, and throughout the course, students think about components as having two distinct parts: an abstract client-side view and one or more concrete implementations. The two views are segregated in distinct program artifacts: the client-side view is given by a Java interface while each concrete realization is given by a Java class that implements the interface.

Of course, these are exactly the intended roles of these two language constructs. What is unique about our approach is our insistence that *all* components be decomposed into these two parts. Interfaces are used early, they are used throughout, and they are used prominently. This approach does not require special frameworks or IDEs. It simply leverages existing Java constructs.

This discipline of component decomposition enforces a clear separation of concerns. It creates an explicit scaffolding, supported by Java language constructs, for distinguishing behavioral abstraction from concrete implementation. This scaffolding can then buttress and augment students' understanding of abstraction and modularity.

The rest of the paper is organized as follows. Section 2 compares our work with similar curricular structures. Section 3 describes the separation discipline and its implications on documentation and testing. Section 4 sketches some exemplar homework assignments from this course. Section 5 lists many good coding practices that follow or are related to this discipline, while Section 6 outlines some difficulties in adopting the discipline in full. Finally, Sections 7 and 8 evaluate this curricular structure and conclude.

2. RELATED WORK

The order of topics in a first Java course has been a matter of considerable debate within the SIGCSE community. One popular structure is to introduce objects early, even as soon as day 1. This technique, known as "objects first", is exemplified by the BlueJ IDE which allows students to easily interact with objects and observe object state with minimal syntactic scaffolding [1, 12]. Although such structures can help build up student intuitions about object-oriented systems, they can mingle abstract and concrete state, blurring for students the distinction between the two.

An alternative structure, termed "components-first", emphasizes the separation of client-side view and implementer's view of components. Students begin by learning how to be clients of components, or APIs, or libraries. They subsequently learn how to implement these components. Examples of this approach include [11, 16, 18]. All of these approaches are similar to ours in philosophy in that they recognize the importance of distinguishing between the two views of a component. The differences are that our design leverages standard Java constructs and that the decomposition is applied consistently throughout the course.

The separation of abstract and concrete states for specification purposes is a well-known technique from abstract data types. Serious Java-based specification notations all support this separation. For example, Liskov & Guttag's book [14], JML [13], and the "Laboratory in Software Engineering" course at MIT, number 6.170 [17], all use specification fields in the documentation of a class to define object state and method behavior. These strategies rely on a documentation discipline of declaring specification variables, then writing pre and postconditions in terms of these variables, rather than fields in the implementation. Such disciplines, however, are difficult to strictly enforce and students may see the declaration of specification variables that closely match fields in the concrete implementation as an unnecessary extra step.

Perhaps the closest work to our own is [20], where the case is made for teaching interfaces *before* inheritance. This ordering is reflected in one popular introductory textbook [9] where interfaces and polymorphism come before inheritance. We agree with this ordering, but this is just one example of placing interfaces earlier in the curriculum and giving them greater prominence. Our approach advocates consistently decomposing components into interfaces and classes. For example, we present the Collections Framework entirely in terms of interfaces (List, Queue, Deque, Set, SortedSet). Iterators (and ListIterators) over these collections are interfaces as well, so the presentation can be quite sophisticated before the implementing classes are even discussed.

3. SEPARATION OF CONCERNS

One way to motivate the interface construct is simply as a mechanism for overcoming Java's single-inheritance restriction. Another option is to present the interface construct as simply a variant of the class construct, much like an abstract class, but with further restrictions (*e.g.*, it can contain only public members, no constructors, and no static methods). Such characterizations, as exemplified in [19], are common in introductory Java courses and serve to relegate the interface construct to secondary standing. In [19], interfaces are not even mentioned until p. 694, where they are covered in half a chapter. This treatment is similar to other popular introductory Java textbooks. While this treatment may reflect the use of Java interfaces in practice, it misses an opportunity to accomplish a larger pedagogical goal: teaching students about abstraction.

Our curricular approach, on the other hand, sets interfaces and classes on equal footing. Every software component is decomposed into two distinct artifacts: (i) a Java interface which is the (abstract) client-side view of the component, and (ii) a Java class which is the (concrete) implementation. Thus, the interface construct is present early and often. Both an interface and a class, together, are required for a complete component.

The notion of separating abstract and concrete state is a classic idea of abstract data types. Specification notations such as JML permit exactly this separation (through model or specification fields). What characterizes our "interfaces first and foremost" approach is the *requirement* of creating a Java interface; separation of abstraction from realization follows as a consequence. Since the interface is a distinct lexical scope from any implementing classes, students see the need for a clear and thorough description of the behavioral cover story in an implementation-neutral manner.

3.1 Documentation with Javadoc

In addition to all the standard Javadoc tags, interfaces and classes are documented in accordance with the distinct roles they serve in defining a component. Custom tags are used to further structure this information.

For interfaces, the documentation defines both the abstract state of the component and the behavior of each (public) method in terms of its effect on that abstract state. That is, the interface documentation must present a cover story that is understandable to a client with absolutely no knowledge of the component implementation (*i.e.*, class).

The cover story can be given with various degrees of formal rigor. We use a collection of custom tags as hooks for telling the client-side cover story:

`@mathmodel`: abstract fields whose types are mathematical concepts such as integers, sets, and strings
`@mathdef`: derived abstract fields that serve as convenient shorthands
`@constraint`: invariants on state
`@initially`: guarantees on the initial state
`@requires, @alters, @ensures`: classic behavioral descriptions of each method

The tag arguments can be written as formal mathematical expressions or informal prose. Either way, the cover story is isolated in a construct with no lexical connection to any underlying implementation. Thus, invariants, preconditions, and postconditions must perforce be written in terms of abstract state. This explicit separation reinforces for students the mental model of distinguishing the abstraction from the realization.

Classes are also carefully documented with Javadoc. In this case, however, the documentation is written in terms of concrete state (*i.e.*, private fields). Custom tags for class documentation include:

`@convention`: invariants on (concrete) state
`@correspondence`: the abstraction relation [21] for mapping (concrete) state to abstract fields

Behavior descriptions (requires and ensures clauses) of public methods are *not* written because this (client-side)

```
public abstract class BigNaturalTest {
  private BigNatural b;
  protected abstract
    BigNatural create(int value);

  @Test
  public void smallInitialization() {
    b = create(34);
    assertEquals("Two-digit initialization",
                 "34", b.toString());
  }

  //more test cases
}
```

Listing 1: Abstract test class with test cases

```
public class SlowBigNaturalTest
  extends BigNaturalTest {

  @Override
  protected BigNatural create(int value) {
      return new SlowBigNatural(value);
    }
}
```

Listing 2: Derived test class provides factory

documentation is already in the interface. On the other hand, private (helper) methods, which exist only in the class, are documented with `@requires`, `@alters`, `@ensures` tags as above. An important difference is that these descriptions are written in terms of concrete state. Again, this practice is easy for students to adopt because of the lexical scoping provided by the Java interface and class constructs.

3.2 Testing with JUnit

The use of JUnit to test software components can further leverage and reinforce the explicit separation of client-side view and implementation.

Black-box test cases are written in a standard JUnit test class using *only* the interface of the component under test. The test class includes one or more abstract factory methods that serve to generate instances for test cases to exercise. Each test case begins by calling these factory methods. Thus, the test class is completely independent of any particular implementation. Listing 1 illustrates part of a test class written in this manner, where `BigNatural` is an interface type.

The test class is an abstract class since it includes an abstract method. In order to execute tests, this class is extended and an implementation for the factory provided (see Listing 2). The derived test class does not provide any new test cases.[1] Of course, this style of test case coding is not new. The point is for students to observe and carefully respect the division achieved by separating abstraction from realization.

In summary, each component consists of an interface/class pair related by implementation. To test this component, a pair of test classes related by inheritance is used.

[1]Implementation-specific test cases could be provide in the derived test class, but this is the exception rather than the rule.

4. CANONICAL SAMPLE ASSIGNMENTS

A good early assignment is to develop an unbounded natural number component. The requirements are simple: A natural number can be initialized, incremented without bound, decremented when it is positive, and its value displayed as a string. The abstraction is clean. The implementation, however, is more complicated since it must account for an unbounded growth. Students quickly see that many design choices for implementing the concrete state exist, including an array of bytes and a string of characters. Furthermore, these design choices involve trade-offs in performance and complexity for implementing the small set of required behaviors.

A simple component such as this one is then refined over the subsequent assignments to illustrate concepts as covered in lectures, including documentation, testing, exceptions, comparability, and immutability. Comparing the (abstract) view of an unbounded natural number with Java's BigInteger is also a nice hook for introducing the subtleties of behavioral subtyping.

For an assignment related to subtyping, we have used a set of three components: Person, Student, and Faculty. All three can contribute to a university's scholarship fund and all three can enter a lottery for football tickets. Only the Java interface is given for each of these components. The components differ in how much prior contribution is required for eligibility in the ticket lottery and in the quality of seats the nondeterministic lottery might yield. Students are asked to identify subtyping relationships and to modify interface descriptions so new subtyping relationships exist. Again, the discipline of decomposing components into both an interface and a class clarifies for students the distinction between subtyping and inheritance [4].

5. GOOD CODING PRACTICES FOR JAVA

Beyond language syntax and structures, students should also learn effective idioms and strategies that support writing good code. There are many such strategies [2], some of which are quite subtle. Using an "interfaces first and foremost" approach clarifies the motivation and key concepts behind many of these strategies, making them easier for students to understand, remember, and appreciate.

Code to the interface.

Recommended practice is to prefer the use of interface types (over class types) for all declared types (*i.e.*, local variables, fields, parameters, and return types). The advantage of this practice is the resulting generality and loose-coupling of the code.

This good practice follows directly from our decomposition discipline. Clients, as far as possible, work only with interfaces.

Document the contract.

Recommended practice is to document method *behaviors* rather than implementations with Javadoc. This practice is sensible given the role Javadoc plays as client-side documentation for a class. Unfortunately, an uncomfortable tension exists between this ideal and the pragmatic observation that Javadoc, as a universally understood documentation notation, can also effectively be used to describe matters of interest to the implementer and future maintainers of an implementation. Indeed, a single command-line flag instructs Javadoc to produce documentation for all private members of a class too.

By decomposing every component into an interface and a class, students do not encounter this tension. They use Javadoc to produce all possible documentation for the interface and all possible documentation for the class, including private members. The former is for the client's consumption and the latter is for the implementer and maintainer.

Design getters/setters properly.

Using public methods to read and write private fields is certainly better than making the fields themselves public. Tools such as Eclipse can even generate these methods automatically for a class with private fields. The problem, however, with this style is that the concrete state drives the abstract behavior instead of the other way around [8].

When students work with an interface and class-based decomposition, however, they recognize the role of getters/setters as readers and writers of abstract state. Facilitating the implementation of these methods is just one factor in the design of a class's concrete state.

Make defensive copies.

Given the ubiquity of references in Java, it is easy for dangerous aliases to a class's concrete state to exist. For example, if a constructor assigns a private field `x` to an argument `y`, both the caller of the constructor and the object itself have references (through `y` and `x` respectively) to the same thing. This is dangerous since the caller of the constructor can make changes to the concrete state of the constructed object directly without going through its public interface.

The separation of interface and implementation does not, itself, mitigate the dangers of aliasing in Java. It does, however, simplify the presentation of these dangers. If students are comfortable with simultaneously considering both the abstract and concrete state, and with maintaining the correspondence between the two, the dangers of aliasing are easily illustrated and quickly appreciated.

Use exceptions properly.

The proper use of exceptions is a matter of much debate, even amongst seasoned Java developers. The choice of whether exceptions should be checked or unchecked, and what kind of exception should be used is often a subtle design choice involving many trade-offs.

Some aspects of exception design, however, follow directly from the disciplined decomposition of our approach. For example, the need to catch an exception and then re-throw it as a *new* exception, possibly of a different type, is clear. The method signatures, including checked exceptions, that appear in the interface must make sense to the client in terms of the abstract cover story. Students recognize when an exception reveals aspects specific to a particular implementation.

As for when to use exceptions, advice is often generic and even circular, for example: Use exceptions for exceptional situations. A better guide is to clearly characterize situations where exceptions are useful. For example, if the client can not unilaterally guarantee the precondition of a method, exceptions are appropriate. A classic example is the existence of a file. Because the code runs concurrently with a real file system in which files may be created and deleted, a client can not know whether the file exists when the method it calls actually starts to execute. Appreciating this lesson is easier if the students are comfortable with behavioral specifications.

Respect behavioral subtyping.

Behavioral subtypes can be dynamically substituted for their supertypes without affecting the correctness of client code [15]. This substitution is sound only when the subtype's invariant and ensures clauses are covariant, while its requirements are contravariant. Since class inheritance involves coupling of concrete implementations, subtyping is best modeled in Java as a relationship between interfaces. A discipline of always declaring Java interfaces is therefore helpful in creating a context for presenting subtyping and its implications.

6. CHALLENGES

In Java, the interface construct corresponds most closely to a purely abstract, client-side view of a component. A Java interface, however, can not include a constructor. That is, an interface is actually the client-side view of an *instance* of a component, and the creation of (other) instances is not generally part of that behavior. On the other hand, some clients do need to create instances. Ideally, such clients would only need the name of the implementing class, nothing else. Unfortunately, without a constructor in the interface, the implementing class must be consulted to confirm the existence of a constructor with the proper signature.

There are several ways to circumvent this difficulty. One is to use a creation pattern in which a separate component serves as a factory. The interface of that factory component defines the valid signatures for instantiating the original component. Apart from leaving a bootstrapping problem (how does the client know how to create a factory?), this solution is somewhat cumbersome since it requires components to consist of 4 artifacts: an interface/class pair for the core component and an interface/class pair for the factory.

Another approach is to require all classes to have a zero-argument constructor. This strategy, however, is limited when it comes to immutable types (which typically do not have zero-argument constructors). Our compromise is to document in the interface the signatures of constructors that implementing classes are expected to have.

7. EVALUATION

The "interfaces first and foremost" approach described here has been the basis for a course that has been offered 5 times, with a total enrollment of 156 students. Survey results indicate the course was well-received by students. The course has averaged 4.7 for "overall rating" (on a Likert-type response scale with 5 being the highest) and 4.2 for "relative ranking compared to other courses in computer science".

Beyond student reaction, however, a better measure for the success of such an approach is the degree to which it transforms students' thinking and instills sound principles of the discipline. To this end, we have followed the cohort of students from our early pilot offerings that subsequently enrolled in the "programming in the large" software engineering course (an existing course that entails significant software design, development, testing, and documentation all done as part of a team). In that second course, students are free to use whatever language they prefer and most choose C++ (the language used in the introductory sequence).

The followed cohort consisted of 85 CS majors. Their work was qualitatively different than that of their peers, according to the instructors for that subsequent course. The recognition and clear application of separation between abstract behavior and concrete representation was present in all of their work (and not their peers). This separation was manifested in their design, documentation, and testing. On surveys, they reported feeling better prepared for a significant software development project than their peers.

8. CONCLUSIONS

In the "real world", Java programmers do not define an interface for every class. While the benefit of encapsulation and information hiding offered by OO are widely recognized, the effort of defining two separate structures for each type is usually too onerous in a deadline-driven environment. Thus, professional programmers often work with just a class and intermingle the realization with its abstraction. As a mental model, the two are hopefully kept somewhat distinct, but this distinction is usually not directly reflected in the code (beyond visibility modifiers such as private and public).

For students of computing science, however, this intermingling should be avoided. Not only does a clear separation help to motivate a wide variety of good coding practices, it also provides an exemplar for the general notion of abstraction, which plays such a fundamental and cross-cutting role in our discipline.

We have used this strategy in the development of a new class that follows an introductory course sequence. Students come to the class knowing imperative programming but not Java. We are optimistic that this strategy could also be adopted for the intro level, especially given the success reported from other efforts in that direction [10,11,18,20]. For example, the materials for this course have been adopted at Clemson and initial reaction there has been positive.

9. ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation under Grant No. 0931669.

10. REFERENCES

[1] D. J. Barnes and M. Kölling. *Objects First with Java: A Practical Introduction Using BlueJ*. Prentice Hall, 2002.

[2] J. Bloch. *Effective Java*. Prentice Hall, 2nd edition, 2008.

[3] T. Colburn and G. Shute. Abstraction in computer science. *Minds Mach.*, 17(2):169–184, 2007.

[4] W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, New York, NY, USA, 1990. ACM.

[5] D. Gries. A principled approach to teaching OO first. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 31–35, New York, NY, USA, 2008. ACM.

[6] O. Hazzan and J. Kramer. The role of abstraction in software engineering. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 1045–1046, New York, NY, USA, 2008. ACM.

[7] P. B. Henderson, D. Baldwin, V. Dasigi, M. Dupras, J. Fritz, D. Ginat, D. Goelman, J. Hamer, L. Hitchner, W. Lloyd, J. Bill Marion, C. Riedesel, and H. Walker. Striving for mathematical thinking. *SIGCSE Bull.*, 33(4):114–124, 2001.

[8] A. Holub. Why getter and setter methods are evil. http://www.javaworld.com/javaworld/jw-09-2003/jw-0905-toolbox.html, September 2003.

[9] C. Horstmann. *Big Java*. John Wiley & Sons, 3rd edition, 2008.

[10] E. Howe, M. Thornton, and B. W. Weide. Components-first approaches to CS1/CS2: Principles and practice. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 291–295, New York, NY, 2004. ACM.

[11] A. Koenig and B. E. Moo. *Accelerated C++: Practical Programming by Example*. C++ In-Depth Series. Addison-Wesley, 2000.

[12] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13(4):249–268, December 2003.

[13] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. Technical Report TR #00-15, Iowa State University, August 2000.

[14] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[16] T. J. Long, B. W. Weide, P. Bucci, D. S. Gibson, J. Hollingsworth, M. Sitaraman, and S. Edwards. Providing intellectual focus to CS1/CS2. *SIGCSE Bull.*, 30(1):252–256, 1998.

[17] Massachusetts Institute of Technology. 6.170: Lab in software engineering. Course notes on web, Fall 2007. http://www.mit.edu/~6.170/.

[18] H. Roumani. Practice what you preach: Full separation of concerns in CS1/CS2. *SIGCSE Bull.*, 38(1):491–494, 2006.

[19] W. Savitch. *Absolute Java*. Pearson Education, 3rd edition, 2008.

[20] A. Schmolitzky. "Objects first, interfaces next" or interfaces before inheritance. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 64–67, New York, NY, USA, 2004. ACM.

[21] M. Sitaraman, B. W. Weide, and W. F. Ogden. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Trans. Softw. Eng.*, 23(3):157–170, 1997.

[22] The Joint Task Force on Computing Curricula. Computing curricula 2001. *Journal on Educational Resources in Computing (JERIC)*, 1(3es):1–240, 2001.