Position Paper

"Consuming before Producing" as a Helpful Metaphor in Teaching Object-Oriented Concepts

Christian Späh
Axel Schmolitzky

Software Engineering Group
University of Hamburg, Germany
+49.40.42883 2302
christian.spaeh@informatik.uni-hamburg.de
schmolitzky@acm.org

Abstract. We discuss the metaphor of "Consuming before Producing" that helped us structuring the contents of Computer Science 1 and 2 courses. It seems to be a feasible approach to reaching coherent compositions of object-oriented concepts. In our experience, students can more easily access topics in courses that are structured based on this consideration. Successfully applied examples at the Department of Informatics at the University of Hamburg support this view. An upcoming master's thesis will reflect on the common components of the metaphor and the pattern behind it. We give a preview of first thoughts about this generalization.

1 Introduction

Using a television for the first time is an easy task. Even if you do not know what such a thing is good for, it is probably very easy for you to find out just by using the remote control. There are direct responses to one's interaction, like changing channels or increasing volume. It is unlikely that anyone would try to discover functionality just by looking into its electronic components. There are many assemblies inside required, e.g. for receiving a signal or transforming line voltage, which are not of any interest for the TV watcher. *Usage* seems to be an effective way to gather information about functions of devices. This is what we call *consuming*: Learning by using. In some cases users might not even realize this learning process. Using or consuming something is an everyday job.

Building a television for the first time is a difficult task. If you are not a genius inventor developing it from scratch, you have to know everything about the internal workings of such a device. You have to study the functions of its electronic components, the format of the audio and video signal and so on. Besides possessing the technical skills required to fulfill this task, you have to take a closer look at implementations of similar machineries and read engineering specifications. Building something typically creates a greater understanding. This is what we call *producing*: Learning by building.

Consuming and Producing are essential parts of human learning. Developmental psychology calls it *information play* or *exploration behavior* when children and kids try to find out what they can do with objects in their environment. It is named *construction play* in childhood once they begin producing similar or completely new items [OM98]. Our metaphor focuses on these two processes and makes their perspectives explicit. Because consuming is easier than producing, things should be consumed first before being produced.

Relating this discussion to the workshop theme, we observe that Computer Science students face several programming languages with different approaches and diverse concepts in their first years at university. Therefore structuring CS 1 and 2 courses is a complex task for lecturers and tutors. They have to focus on the most relevant knowledge and present it as part of an integrated conception. Since concepts are highly interrelated, students have to master many of them early to be able to write their first programs and solve their first problems.

The intention of this position paper is mainly to present the metaphor “Consuming before Producing” that helped us structuring contents in introductory programming courses and second, to serve as an outlook for a master’s thesis that will extract the basic components of the pattern behind the metaphor.

2 Successful Examples in Teaching OO Concepts

To illustrate the use of the metaphor “Consuming before Producing,” we list several examples from our introductory programming courses taught in the first and second semester at the Department of Informatics at the University of Hamburg.

Consuming *objects* before producing them. BlueJ is an IDE which enables an *Objects First* approach [KQPR03]. It allows students to start out, in the first week of semester one, with an object “system” with just one class. They interactively use instances of this class, consuming the notion of *interacting* with an object via its interface. More knowledge about class internals is required though to produce the possibility of interaction with an object (constructing just one class). Thus this should be done after the principles of interaction with objects are well understood. We start with constructing new classes in week 3 of semester one.

Consuming *packages* before producing them. Students consume the concept of packages by importing classes or interfaces from named packages, e.g. `List` and `Date` from package `java.util`. These classes and interfaces are used very early in courses (in the middle of the first semester), and at that time students just need to know that packages bundle them. Producing packages requires knowledge about package visibility and visibility rules in general. Students get in contact with these concepts when larger systems need to be structured, which is typically the case in the second semester earliest. Thus we teach construction details of packages in the second half of semester two.

Consuming *inheritance* before producing it. Students consume inheritance (code reuse) by using the `assert`-methods in a test class derived from the JUnit class `TestCase` or when they call the method `equals` on objects of their own classes. They implicitly reuse code from the class `Object` that compares references. Both unit testing and redefining `equals` can be practiced in semester 1. Producing inheritance, for example refactoring commonalities of two classes into an abstract superclass, requires more knowledge about inheritance and can be taught later. We introduce inheritance in detail in the first half of semester 2.

Consuming *genericity* before producing it. Students consume genericity (parametric polymorphism) by using lists and maps from the Java Collections Framework (JCF). They parameterize variables of types `List` or `Map` and instances of `ArrayList` or `HashMap` with their respective element type and learn that this type is checked at compile time. The need to declare the element type of a collection is fairly easy to understand, thus we teach consuming collections in the second half of semester 1. Producing genericity is far more challenging. Producing genericity means writing a class with a formal type parameter and understanding the limitations implied by Java’s type erasure. Students need to be well-versed in several concepts (actual and formal parameters, typing issues with type casts, etc.) before they fully understand implementing parametric classes. Thus we teach genericity in detail only in semester 2.

Sometimes, producing a concept can even be left out. For example, Barnes and Kölling teach consuming genericity in their text book [BK06], but they do not cover producing genericity.

3 Extracting the Pattern

While talking about consuming in our examples, we implicitly refer to *interfaces* of things. For example, watching TV means using the television device according to its interface specification. Producing, as the second part of the metaphor, always means to *implement* something. Thus interface and implementation are essential parts of our metaphor. Distinguishing these two scopes of an element helps

to discover fragments that can be coherently ordered. We illustrate a possible order for an extended version of our last example of section 2.

Consuming *collections* before producing them. Students consume the concept of collections by using lists and maps from the JCF, implicitly also consuming genericity and the concept of an *iterator*. As `Collection`, `List` and `Map` are interfaces in the Java Collection Framework they have to consume the concept of a *named interface*, too. These dependencies induce a structure illustrated in figure 1.

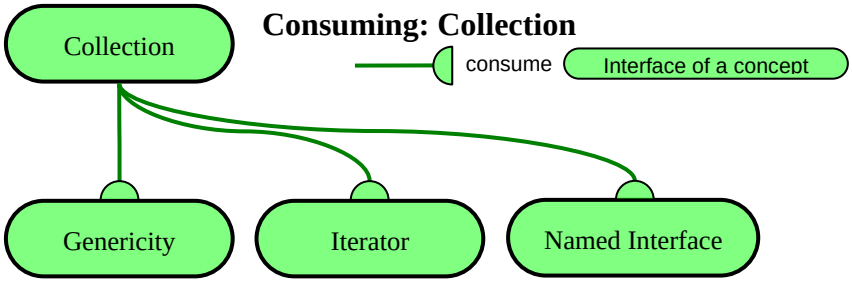

Figure 1: Example of dependencies for consuming a collection

When consuming collections, the concept of a collection is interrelated with *genericity*, *iterator* and *named interface* through a consume relation. There are several other concepts that collections depend on, such as parameters, references, equality and identity. They are left out for the clarity of this example.

Once students are well-versed in using collections, they can begin to actively produce collection implementations themselves (as a good training for working with references). They then move on to producing genericity (the question how to implement a generic class) and to producing iterators. Figure 2 shows the drawing in this case.

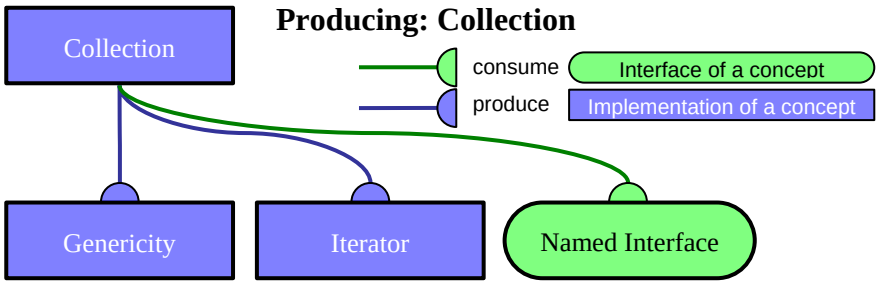

Figure 2: Example of dependencies for producing a collection

When producing collections the reference between this concept and genericity and iterator switches to produce. Implementing a collection means to program to the interface `Collection` or one of its sub-interfaces. The concept of a named interface still has to be consumed then. We observed that producing a concept interrelates with interfaces and implementations of other concepts, while consuming a concept interrelates with interfaces only. But this has to be verified.

With the help of the metaphor “Consuming before Producing,” we can split interrelated contents into parts that are potentially easier to order. These parts reduce the amount of topics students have to cover before solving a problem. Fragments unnecessary at the moment can be taught later.

In our opinion the two parts – consuming an interface and producing an implementation – can be found in every learning context. We therefore think that “Consuming before Producing” can form the base of a pedagogical pattern. Its application can follow this line:

1. Identify the concept to teach (and the concepts it depends upon).

2. Identify the interface and implementation of each concept.
3. For any interface and implementation identified, define whether it consumes or produces concepts it depends upon. Draw consuming connections to interfaces and producing connections to implementations.
4. Teach consuming (parts) before producing (parts).

This pattern results in the topics of a course being more refined and teachers get a detailed plan of the contents' order.

There are similarities to Bergin's Spiral pattern [PPP]. He recommends breaking down complex topics into fragments and introducing these fragments "in an order that facilitates student problem solving". Early fragments give just enough detail to form a basic understanding. "Additional cycles contain reinforcing fragments that go into more detail on the topic." But Bergin leaves it to the reader to decide which parts come first. Our pattern may give more constructive advice here. While it is essential for the Spiral to reinforce topics through additional cycles, this is not an important point for the "Consuming before Producing" pattern. Our perspective primarily aims at showing how to structure course contents.

4 Authors' Biographies

Christian Späh (christian.spaeh@informatik.uni-hamburg.de) has been a student of information systems at the University of Hamburg, Germany, since 2002. His interests include organizational theory and psychology, software engineering and didactics of teachings. He has been working as a tutor at CS 1 and 2 courses and currently helps to prepare and accomplish a seminar on concepts of object-oriented programming. During his studies he designed several animations in SVG for academic teaching. He co-authored a paper on this topic that got accepted at the ED-MEDIA conference in 2004. In 2006 he was working part-time as a software engineer for the LAssi-Project, Hamburg. This paper will contribute to his upcoming master's thesis.

Axel Schmolitzky (schmolitzky@informatik.uni-hamburg.de) has been a research assistant at the University of Hamburg, Germany, since 2001. He graduated in Computer Science at the University of Bremen, Germany, and holds a Ph.D. in Computer Science from the University of Ulm, Germany. He spent one year in the BlueJ-Group as a post-doc fellow of the DAAD (German Academics Exchange Service) at Monash University in Melbourne, Australia. From 2001 to 2003 he was also working part-time as a consultant and trainer for an Informatics Department spin-off company in Hamburg where he gained experience in industry projects and trainings. He has been actively engaged in restructuring the first year programming education at Hamburg University. His main research interests are in the areas of object-oriented software construction, agile process models, programming language design, and educational issues in computer science.

5 References

[BK06] Barnes, D. and Kölling, M.: *Objects First with Java - A Practical Introduction Using BlueJ (3rd Edition)*, Pearson Education, UK, 2006.

[KQPR03] Kölling, M., Quig, B., Patterson, A., Rosenberg, J.: "The BlueJ system and its pedagogy," *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13:4, pp. 249-268, 2003.

[PPP] Bergin, J.: *The Pedagogical Pattern Project*, http://www.pedagogicalpatterns.org (last visited on May 11, 2007.

[OM98] Oerter, R, Montada, L.: *Entwicklungspsychologie*, 4th ed., Psychologie Verlags Union, Weinheim, pp. 250-267, 1998.