Munindar P. Singh and Mona Singh

Deconstructing the "Any" Key

A popular techies' tale goes as follows. A user calls customer support with a problem: "The program says 'press any key to continue' but I can't find the any key on the keyboard."

Stupid user.

We suppose this story is apocryphal. Most people have a reasonable understanding of language even if they aren't familiar with computers. But we're not surprised if some user has had this very difficulty figuring out the "any" key.

Our point is that although an actual user may or may not be as ill-informed as the apocryphal one, virtually all real programmers are insensitive. And we don't mean personal insensitivity. We mean technical insensitivity, the kind that results from viewing a problem solely from one's own perspective.

We imagine the programmer writing the instructions to press any key wants to delay processing until the user has acknowledged reading some message. (This is called a dialogue, although it doesn't allow much in the way of conversation.) The programmer writes a simple program to achieve this effect—let's call this desired behavior "acknowledged continuation." This program might look something like the following in pseudo code:

```
<stuff done before>
display("press any key to continue");
get_character() <implied wait>
 <stuff done after>
```

This is a nice program and gives every indication of being reliable. The program displays a message and awaits an input from the user. As soon as an input, any input, arrives, the program can continue.

When the user sees the message "press any key to continue," he or she (usually) has a readily available keyboard. By definition, one would expect, a keyboard is something with keys.

But not all keys are alike. Experienced users know to press a key that works, such as the space bar or the enter key. However, this is more a matter of a user having been trained through past experience to press the right keys than a matter of the specification itself being clear.

For instance, some of the keys have no effect. Press the shift or control key and nothing happens. Press the caps lock key and it may affect what you type later, but the program still doesn't budge. On some computers, when the alt key is pressed, something weird may happen. Ditto with the function keys.

Press the power on-off key, which exists on some keyboards, and the results may be far more dramatic than desired. It is fair to say the program will not continue as promised.

Some key combinations count as one key. Press shift and a letter (say, j), and it works fine, as any programmer would expect. However, some key combinations are dangerous. In the Windows world, the control-alt- delete combination opens up a window listing running programs that the user may kill. A second control-alt-delete can reset the computer. In the Unix world, control-C can kill a program and control-Z suspends it. Control-S can suspend any output.

The gist of this is that a specification saying "press any key to continue" is far from unambiguous to the user. It may seem clear from the programmer's perspective because the program receives an input only under circumstances that the underlying oper-

ating system deems appropriate, that is, only when an acceptable key is pressed. So the right display message ought to be, "press any acceptable key to continue." However, such an instruction would be meaningless, because the term "acceptable" has no obvious interpretation.

The moral? We as programmers should take into account alternative perspectives, especially the users' perspective. To understand a communication from the perspective of its recipient is one of the lessons of deconstruction theory.

A lot of people talk of the importance of being user-centered. Here is a case where being user-centered has direct consequences on our programs. Although thinking from another perspective is rarely as easy as in the case of the "any" key, we must keep trying.

As for the matter of acknowledged continuation, just ask the user to press the space bar. C

Munindar P. Singh (singh@ncsu.edu) teaches computer science at North Carolina State University in Raleigh.
Mona Singh (mona.singh@ericsson.com) is with the Ericsson New Concepts group in Research Triangle Park, NC.

Jerzy W. Grzymala-Busse and Wojciech Ziarko

Data Mining and Rough Set Theory

This is in response to "Myths about Rough Set Theory" (Nov. 1998, p. 102) by W.W. Koczkodaj, M. Orlowski, and V.W. Marek. The authors raise some important issues and express some legitimate concerns. We are surprised they list rough set theory as the only discipline in which there are two of the cited problems—the discipline in which discretization is necessary or which deals with complex data. The third problem raised by the authors is associated with the difference between objective and subjective approaches to uncertainty.

Let us start with discretization. Many people deal with discretization unknowingly. For example, in grading student work, there are usual cut-points (90% for an "A," 80% for a "B," and so forth); original scores are replaced by intervals, coded by "A," "B," and so on. The authors are probably confused by the fact that in some applications of rough set theory, discretization is used as a preprocessing. However, discretization is required in all rule (or tree) induction systems. Such systems constitute the core of data mining (or knowledge discovery). Many such well-known systems (such as C4.5, based on conditional entropy or CART, based on Bayes rule) are equipped with their own discretization schemes. Neither C4.5 nor CART use rough set theory. Practically every machine learning system uses discretization while very few of them are based on rough set theory. To complicate matters, discretization methods used in rule induction systems based on rough set theory, such as KDD-R or LERS, are not based on rough set theory (for example, both KDD-R and LERS use statistical methods). Furthermore, these discretization methods could be used in other systems, (in C4.5 or CART). On the other hand, all existing discretization methods, based on many different approaches to uncertainty, could be used as preprocessing for KDD-R and LERS. Discretization is a technique used in many areas, including machine learning and learning in Bayesian networks, and is definitely not restricted to rule induction systems based on rough set theory.

To illustrate the complexities involved in data analysis, the authors refer to an example of a table with 10 attributes, each with 20 values, which is likely to lead to a large "number of possible instances." However, one could cite this kind of example to illustrate potential problems occurring in all disciplines dealing with data, starting from statistics, through database management, and ending with machine learning (for the sake of correctness, it is not clear what the authors mean by "the number of possible instances."

Most likely they refer to the number of possible different cases (rows) of the table. They are mistaken. The correct number is 2010^10 = 1.024*10^13). In all of these areas we may deal with big data sets and with potentially large number of different data sets. Again, the problem is common to all of these disciplines and by no means occurs just in rough set theory. Fortunately, rough set theory offers algorithms with polynomial time complexity and space complexity with respect to the number of attributes and the number of cases.

Finally, regarding the authors' comments about objectivity of rough set theory, we are puzzled why they assume objectivity means superiority. They confuse daily life with science. In real life we would like to have objective managers, for example, giving salary raises based on merit. But this is not how the subjective or objective approach is understood in science. Probability theory is a good example. It is an example of well-established calculus of uncertainty. For a long time there was a dispute (and still is) between the objective approach to the definition of probability (based on experiments and relative frequencies of outcomes) and the subjective approach (based on experts' opinions). For example, an individual may observe a game based on a random process and evaluate probabilities. This is the objective approach. Or, the individual may ask a gambler how he or she will bet his or her own money. This is the subjective approach. Currently, subjectivism prevails in probability theory. The proponents of the subjective approach do not show any inferiority complex. The problem is definitely not which approach is superior. The Dempster-Shafer theory uses the subjective approach to uncertainty because its fundamental tool, a belief function, should be estimated by experts. There are close relations between the Dempster-Shafer theory and the rough set theory. Both theories describe the same phenomena. However, in rough set theory the basic tools are sets: lower and upper approximations of the concept. These sets are well defined and are computed directly from the input data. Thus, rough set theory is objective, but it does not mean that it is superior (or inferior). For example, if input data were preprocessed and numerical attributes were discretized by an expert, the resulting data might be subjective. But again, this preprocessing is not a part of rough set theory, as we explained previously. Input data must be given to initiate rough set theory procedures, and, when rough set theory comes into the picture, its methods are objective with respect to given data. C

Jerzy W. Grzymala-Busse (jerzy@eecs.ukans.edu) is a professor in the Department of Electrical Engineering and Computer Science at the University of Kansas. **Wojciech Ziarko** (ziarko@cs.uregina.ca) is a professor in the Computer Science department at the University of Regina, Canada.

Friedrich Steimann

Abstract Class Hierarchies, Factories, and Stable Designs

Much of the debate about the general aptness of class hierarchies is rooted in the different objectives taxonomists and implementers are thought to pursue. Designers of conceptual hierarchies tend to embrace Aristotle's principle of *genus et differentiae* leading to a taxonomic hierarchy of categories or types [7], while those with implementation in mind focus on the reuse of class definitions and polymorphism as made possible by subclassing and inheritance. This has led to an extensive discussion (see [1, 4, 5, 8]) as to whether Square should be a subclass of Rectangle or vice versa, a dilemma that is, of course, precedential in character.

Despite the different perspectives there appears to be a broad consensus that, in principle at least,

- both a conceptual type and a class (as a programming construct) are intensions the extensions of which are sets of instances; and
- the extensions of subtypes are subsets of the extensions of

their supertypes, so that the instances of a subtype can occur wherever instances of its supertypes are expected (principle of substitutability).

Depending on the programming language, the latter may not be the case for class hierarchies so that conceptual type hierarchies and class hierarchies are not generally isomorphic to each other [6]. However, as Grosberg rightfully observed [4], the discrepancies can easily be resolved by adhering to one simple rule: by requiring that only the leaf classes can have instances.

Abstract Class Hierarchies

This rule is not as arbitrary as it may seem. In fact, it only paraphrases a common constraint on subtyping, namely that the extension of a supertype is totally covered by the extensions of its subtypes, thus rendering the supertype a mere abstraction. For example, applied to the class hierarchy of Figure 1a, it is implied that all instances of type Person must either be an instance of class Female or of class Male.

Following this principle, the Rectangle/Square dilemma is resolved as shown in Figure 1b), where OtherRectangle denotes the set of rectangles that are not squares. Surely, this is going to affront many system modelers and most implementers: why waste the name Rectangle for an abstract class which cannot have instances, and why introduce an additional class oddly named OtherRectangle which will create most of the instances of Rectangle? First, not having class OtherRectangle is a bit like having classes Person and Female, but not Male. Second, OtherRectangle could, of course, just as well be named NonSquareRectangle—the point here is there is always a sibling class that holds the remainder otherwise assigned to the superclass. And third, when creating a particular rectangle, its clients need not see or know about (unless they desire to) the distinction between Square and OtherRectangle—they simply resort to a factory.

Figure 1. (subclasses are indented, leaf classes underlined).
a) Female and Male are the only subclasses of Person
b) rectangles are either Squares or OtherRectangles

```
a)Person   b)Shape
    Female      Rectangle
    Male          Square
                  OtherRectangle
                ...
```

Factories

A factory is an object-oriented programming construct providing for the creation of instances without specifying their concrete classes. Factories come in many different guises, the most common of which have been stereotyped in the form of design patterns [2, 3]. Here we think of a factory as an abstract class whose creator methods (called factory methods) return instances of its concrete subclasses. In the geometrical shape example, squares and (nonsquare) rectangles might be created by calls to factory methods of class Shape as shown in Figure 2.

The clients of the hierarchy, cognizant only of class Shape, will not know or need not care about the actual type of the instance they get, but nevertheless (through dynamic binding) receive all the benefits of the different, possibly optimized implementations of methods for classes Square and OtherRectangle, such as the calculation of the area.

One may object: what if I stretch a square in one dimension? Does that not imply instance migration? Well, what if I shear a rectangle? Indeed, stretching and shearing should be viewed and implemented as what they actually are: mathematical operations that return new instances. In this light, every operator is a small factory method returning a new instance of a class determined solely by the operands (including the implementor) and the operator itself. The same principle naturally applies to hierarchies of numbers, collections, and so forth, with plenty of opportunities to exploit the efficiency and

Figure 2. Calls to a factory class creating instances of the appropriate type.

```
Shape.rectangle(120,60) // (width, height)
   // creates a new instance of class OtherRectangle
Shape.square(80)
   // creates a new instance of class Square
Shape.rectangle(80,80)
// also creates a new instance of class Square
```

maintainability gains offered by a clean partitioning of the problem domain. For instance, depending on its operands, the division of two integers may return an integer or a (noninteger) fraction.

Stable Designs

While the practical benefits of conceptually sound class hierarchies are still arguable, there is another, very pragmatic reason to enforce the rule of letting only leaf classes have instances: it protects the rest of the class hierarchy from ad hoc alterations made to individual class definitions. Given that most of the many changes that become necessary in the course of system evolution pertain to the behavior of instances of individual classes, the propagation of these changes (through inheritance) to other classes is not generally desired. However, especially if class hierarchies are big and used by many clients, the existence of and consequences for descendant classes are not immediately realized, making inheritance a mixed blessing. By designing the class hierarchy as a hierarchy of abstract classes and by letting its clients manipulate only the concrete classes attached as leaves, the effect of modifications needing to be made by the clients is always confined to the instances of single classes. The need for a (partial) redesign of the class hierarchy because of practical requirements is thus greatly reduced. C

FREIDRICH STEIMANN (steimann@acm.org) is a research assistant at the Universität Hannover, Germany.

REFERENCES

1. Baclawski, K. and Indurkhya, B. The notion of inheritance in object-oriented programming. *Commun ACM 37*, 9 (Sept. 1994), 118–119.
2. Cooper, J.W. Using design patterns. *Commun ACM 41*, 6 (Jun. 1998), 65–68.
3. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass. 1995.
4. Grosberg, J.A. Comment on considering 'class' harmful. *Commun. ACM 36*, 1 (Jan. 1993), 113–114.
5. Halbert, DC. O'Brien, P.D. Using types and inheritance in object-oriented programming. *IEEE Softw. 4*, 5 (May 1987), 71–79.
6. LaLonde, WR and Pugh, JR. Subclassing π subtyping π is-a. J. *Object-Oriented Program.* (Jan. 1991), 57–62.
7. Sowa, J.F. *Conceptual Structures: Information Processing in Mind an Machine*. Addison-Wesley, Reading, Mass., 1984.
8. Winkler, J.F.H. Objectivism: 'class' considered harmful. *Commun. ACM 35, 8* (Aug. 1992), 128–130.