Reexamining an Object-Oriented Design Heuristic

Gerald Weiss
Brooklyn College City University of New York

David Arnow
Brooklyn College City University of New York

Reexamining an Object-Oriented Design Heuristic

Gerald Weiss
David Arnow
Department of Computer and Information Science
Brooklyn College
City University of New York

Introduction

Object-oriented programming, unlike competing programming models such as functional, logic and procedural programming, does not owe its origins to mathematics or the adaptation of mathematics to the computer. Instead, its origin is simulation [2]. The purpose of simulation usually is to model the real world or a plausible system that functions in the real world. Therefore it is not surprising that one object-oriented design heuristic often presented to beginners is "model the real world". [3,4 for example].

The Pitfalls of Modeling the Real World

Unfortunately, unless one is actually carrying out a simulation of the real world, "model the real world" often turns out to be poor advice if taken literally. The reasons for this vary.

Reason 1: *real objects often exhibit little behavior*

Consider a time-card, such as one that an hourly worker might use to punch in and out. A real time-card is nothing more than a repository for information that is added incrementally over the course of a work week. Besides allowing information to be added, the information stored on a time-card can be retrieved: a collection of start times and finish times. Real time-cards do not determine the hours worked in a given day, will not identify overtime, and will not provide a weekly hourly total. The object itself is totally passive.

In cases such as this, the consequence of modeling the real world is an overly centralized design: one or more primary controlling classes incorporating all the behavior and manipulating a bevy of classes that are little more than records (structures) with get/set methods.

Reason 2: *real objects may exhibit undesirable behavior*.

Consider a real employee. How many companies have payroll systems where the employee is asked by the writer of checks for the amount of pay that is his or her due? Does that mean that we should not incorporate the calculation of payroll as a method in an Employee class?

The consequence of attempting to model the real world here is additional confusion in the design stage. In the early stages of design, assignment of responsibility and behavior to classes is problematic enough without unnecessary and confusing constraints being added.

The Heuristic Revised: Model an IDEAL Real World

Cell phones offer an interesting contrast to time-cards. Like time-cards they maintain a collection of activity intervals. Unlike time-cards, they provide additional behavior: a cell phone typically will report various activity time totals. Although not available now, one could imagine cell phones providing cost totals as well. The difference between cell phones and time-cards is that real cell phones are intelligent objects while real time-cards are not.

A reasonable design for a time-card class would include a method to query a time-card object for its total. Considering this design in the context of the above comparison, we see that we are modeling not a real time-card, but one that has a similar intelligence to that of a (real) cell phone.

The correct heuristic then is to model an ideal real world. What is an ideal real world? It is one that is functionally similar to the actual real world but which in some sense is futuristic- all objects possess an appropriate intelligence.

Imagining an Ideal Real World

A useful technique to aid in the design of such an ideal model is imagine automated, intelligent versions of the various objects. For example, we might imagine a futuristic time-card that responds to various queries concerning hourly totals, overtime, and other personnel/payroll items. Assigning some degree of "intelligence" to the time-card helps us determine the desired behavior of this object. Making this behavior the responsibility of the object relieves its users of these burdens.

Behavior is not assigned on the basis of what occurs in an analogous real world situation; it also depends on which class is best suited to be responsible for maintaining that behavior. Much of the design of an application hinges upon determining a proper distribution of responsibility among the various classes in the system. Once responsibility for some form of behavior has been delegated to a class, users of that class can rely on that behavior. The appropriate model is then not necessarily the one that most closely mirrors the real world application, but rather the one in which responsibility has been assigned so that the resulting software system is most understandable.

Some Examples

Books

Books in the real world are passive objects, again acting as no more than repositories of their content. Designing a book class that mirrors this inanimate object produces a class with no true behavior.

Now let's conduct a small thought-experiment: how would a "smart" book work? We would expect automatic page turning, searching, index creation and lookup and so on.

As we return to our book object design, our thought-experiment suggests what behavior our book class should possess.

A Board Game

The game pieces and board of a board game such as checkers are again passive, inanimate objects that are acted upon, rather than exhibiting their own behavior in real life. It is the player who moves the pieces, sets up the board, or determines whether the end of the game has been reached. Similarly, the players are the ones that are responsible for conforming to the rules, for example making legal moves and taking turns.

Again, designing a system that mirrors this situation produces a series of 'do-nothing' classes corresponding to the inanimate objects-board, pieces, dice, and so on. All complexity gets designed into the player class which becomes very tightly coupled to the other classes (as the player must be manipulating those objects at a fairly low semantic level.)

Now imagine a commercially packaged, electronic checkers game (in this case, the "future" is now). The board might consist of a grid of LCD touch-sensitive squares. The pieces are images displayed on the board's squares. Players make move by touching all the squares along the path of their move. The board verifies the legality of moves, maintains and displays the state of the game, enforces turns and announces the end of the game.

Again, our thought-experiment offers a better model for our class design, one that yields an intelligent board object, and by doing so simplifies the design of the player class.

Classroom Experience

We discovered the problem with the traditional heuristic when we started teaching a Java-based object-oriented CS1 course. We wanted to provide a "programming as modeling" view from the outset to serve as a conceptual framework for our students.

This poses several problems for beginners. First, the behavior of objects that are provided in the Java class library and of objects that instructors or introductory textbooks might introduce early on is apparently at odds with this. Consider the String class in Java. Students can be reasonably told that a String object models a sequences of characters- something which occurs in the real world all the time. Confusion immediately sets in when we start sending messages to a String to get substrings, space-trimmed strings and modified case strings from our String. What "real-world" string exhibits that behavior? For weaker students, confusion sets in. Stronger students on the other hand decide that the heuristic is fluff and become leery of design principles altogether- a very dangerous attitude.

The second problem comes later in an introductory course, as students themselves are invited to begin to design solutions to non-trivial (by CS1 standards) problems. As everyone knows, the initial forays in object-oriented design by professional non-OOP programmers typically involves a single complex controlling class along with an auxilliary set of classes that provide little behavior beyond that of data repository. Beginning students need not fall into this trap. Unfortunately, attempts to literally model the real world as it is often leads to this same problematic design. That's because most real world objects display very little behavior or intelligence- they really are a just an aggregate of data.

To summarize, the widely stated heuristic "model the real world" confuses weak students, discredits design principles in stronger students and leads beginners of all varieties astray. To avoid that, we introduce students early in the course to the concept of smart, helpful and self-responsible objects[1].

In particular, with each class design exercise, the instructor should explicitly invite the student to consider intelligent versions of passive objects discovered in the problem.

Conclusion

It is unlikely that a revised statement of the "model the real world" heuristic will have an effect on experienced object-oriented designers. Despite lip service to the heuristic their own experience and the resulting intuition as well as other heuristics ("avoid centralized designs") will lead them away from the heuristic's pitfalls. It is our hope, however, that a revision of the heuristic will help beginners in programming as well as procedural programmers embarking on the object-oriented paradigm shift. In addition, our own experience suggests that the revision eases the task of teaching object-oriented programming and in particular explaining the design of existing classes.

References

[1] Arnow, D and Weiss, G: Introduction to Programming Using Java - An Object-Oriented Approach, Addison-Wesley/Longman, 1998.

[2] Dahl, O. and Nygaard, J.: Simula- An Algol-based simulation language CACM 9(9) 671-681.

[3] Friedman, F. and Koffman, E.: Problem-Solving, Abstraction and Design Using C++. Addison-Wesley/Longman, 1996.

[4] Kamin, S., Mickunas, M. and Reingold, E.: An Introduction to Computer Science Using Java. McGraw-Hill, 1998.