Abstract Better and Enjoy Life

by Edward Colbert
Absolute Software Co., Inc., 1444 Sapphire Dr., Carlsbad, CA 92009-1200
Voice (760) 929-0612, *FAX* (760) 929-0236, *E–mail* colbert@abssw.com

Since software by its very nature is abstract, i.e. “conceived apart from concrete realities” [1], software developers should naturally be adept abstractionists. Yet many developers have difficulty abstracting.

Software developers struggle with the object–oriented paradigm, and as a result often fail to meet project goals. Applying the paradigm to requirements analysis and design, which are problematic enough in themselves, seems to worsen the struggle. My teaching and consulting experience suggests that these difficulties result largely from poor abstraction.

Most discussions of the object–oriented paradigm use *abstraction* to mean development of class hierarchies. However, it has much broader importance. Whenever we describe a “real object” in software, whether using classes or not, we are abstracting. We do not state all we know about the object; instead we only include enough detail for purposes of the system we are building. For example, *a car* described for a freeway simulation would look very different from the abstraction of the same vehicle in a dealer’s inventory control system. If I create a single description of a car object that would be fully interchangeable in both systems (possibly defined by a hierarchical or networked set of classes), I can succeed only by raising the overhead in each system.

A developer can bog down building elaborate, “correct” class hierarchies that are unnecessary to solve a problem. This attempt to maximize re–use can have the opposite effect. If one does not limit classes and relations to those actually in the system, the result will be reduced development, maintenance, and execution efficiencies for the current system, and often a class hierarchy that is ineffective for re–use as well [2].

Developers are often confused by the way the object–oriented paradigm represents interactions. For example, *Fred hits the ball* is typically represented as *the object Fred requests that the object Ball perform its Hit operation*. In the “real” world a ball is not normally viewed as having a *hit* operation which it performs, but we use this representation to localize the operation *hit* and the structure (“data”) of the *ball* in a module we can easily maintain or enhance. Equally object–oriented would be *the object Fred requests that the object Ball perform its React To Applied Force operation*. This also localizes the operation and data, can be re–used (e.g. *Fred throws the ball*), and is nearer the way we “really” think, but is less specific to what we originally stated, and may make the system harder to maintain. The best abstraction is the one that meets the goals of our project.

Choosing among representations can be worrisome. For example, a typical cruise–control problem says *the driver engages the system by pressing the engage button, which writes a value to the line connected to the cruise control*. Which of the following is right ?

Published in the *Jornal of Object–Oriented Programming*, 1994, March/April Issue, SIGS Publications, New York.

the Driver object requests that the Cruise Control object perform its Engage operation;

the Driver object requests that the Button object perform its Push operation, and the Button requests that the Cruise Control perform its Engage operation; or

the Driver object requests that the Button object perform its Push operation, the Button requests that the Line object perform its Write operation with a Value parameter, the Cruise Control requests that the Line perform its Read operation returning a Value, and the Cruise Control will perform its Engage operation if the line value is "Engage".

This is a typical problem of relating analysis to design. The earlier of these representations are "implemented by" the later ones. Each might be documented at a different phase of development, so that the thought process could be traced, and the impact of a change could be easly localized (e.g. replacing the *line* with a *network* or replacing the *button* with a *voice activation*). The choice is easier when representations like these are recognized as only different levels of abstraction.

Relax, developers! Ease your troubles by applying a skill you already have — the ability to abstract. The object–oriented paradigm is itself only a kind of abstraction. We could instead be function–oriented, or data–oriented, or flow–oriented, and indeed such paradigms have been used. We came to prefer the object–oriented paradigm because it is more convenient for the features we need in software today, e.g. maintenance, enhancement, and re–use. On this scale too we choose an abstraction based on what we want it for.

References

1. *Webster's Encyclopedic Unabridged Dictionary of the English Language*. 1989, Portland House:
2. Gabriel, R.P., "Abstraction descant, part 1", *Journal of Object–Oriented Programming,* 1993. Vol. 6, No. 1 (March/April): p. 10–14.