A Fractal Model of the Lifecycle of Reusable Objects[1]

Brian Foote[2] -- 29 August 1991

Reusable object-oriented abstract classes, frameworks, and components have lifecycles of their own that are separate and distinct from those of the applications in which they are incubated. They are the result of an iterative, evolutionary process that unfolds simultaneously within and beyond individual applications. Because this process can be characterized as a pattern that is repeatedly replicated at all levels of a system, we call our lifecycle perspective the *Fractal Model*.

Objects, in and of themselves, have a greater potential for reuse than do conventional software components. Each of the characteristics that distinguish objects from conventional components contributes to this potential. *Polymorphism* increases the likelihood that a family of related objects will operate correctly in a variety of different contexts. *Inheritance* allows a class to spawn an entourage of related subclasses that do not undermine the integrity of the original class, and promotes the emergence of *abstract classes* and *frameworks*. *Encapsulation* insulates an evolving part of a system from the rest of the system, and promotes the emergence of *components* with identities distinct from the systems in which they are embedded.

Successful reuse requires not only a commitment to new languages and tools, but a complete change in one's outlook on the software development process as well. An organization with a commitment to reuse seeks, at the completion of a given development effort, to produce two distinct products. The first is a deliverable application. The second is an enhanced legacy of reusable abstract classes, components, and frameworks that can serve as the foundation for similar future efforts.

The Fractal Model distinguishes three distinct stages, or phases, in the evolution of object-oriented abstract classes, frameworks, and components. The first is a ***Prototype***, or ***Initial Design*** Phase. The prototype is a quick first pass that may be quite loosely structured, and make use of expedient, inheritance-based code borrowing. During the prototype phase, the designer should concentrate on the problem at hand. Reuse should be a secondary concern. Expedient, first pass designs should never be mistaken for good designs (though we all hit a hole-in-one occasionally). Note that prototypes are not produced in a vacuum, but depend instead on pre-existing libraries of reusable components.

An object that proves successful enters an ***Expansionary***, or ***Exploratory Design*** Phase. There is a distinctly Darwinian quality about this[3]. An object's demonstrated utility inspires attempts to reuse it for purposes that differ from its original purpose to varying degrees. In conventional systems, such reuse might be undertaken by *scavenging* copies of the original component, or by introducing flags and conditionals into the original code. Such activity tends to destroy the system's structure, and increase the entropy and disorder the system. There is a risk during this phase that the component may suffer from "mid-life" generality loss.

Object-oriented systems can retain the integrity of the original code by placing new code in subclasses. As a result, broad, shallow *white-box* class hierarchies can develop. The subclasses added during the exploratory phase preserve the integrity and identity of the requirements that inspired them, but are not yet truly general.

During the ***Consolidation***, or ***Design*** **Generalization** Phase, experience accrued during successive reapplications of an object is used to increase its generality and structural integrity. As is the case everywhere else in the universe, there can be no entropy reduction without an expenditure of energy. During this phase, the programmer reorganizes the class hierarchy, and abstract classes that reflect the structural regularities in the system (and the problem domain) emerge. It comes to reflect the way you would like to be able to tell people you arrived at the design. The informal, inheritance-based, white-box relationships that may be present in the system can be recast using *black-box* components.

At each level, the model moves from the prototype phase through the expansionary phase, into the consolidation phase. From there, an element may expand further, or serve as the basis for a new prototype. As objects evolve, the breadth of class hierarchies decreases as the depth increases, and discrete, black-box components emerge from ill-defined inheritance hierarchies. Both progressions enhance this system's generality and reusability.

This model has a number of technical and managerial implications. One is that design is more than a discrete phase in the development of a component, instead, it is an activity which pervades the lifecycle. This model places emphasis not so much on the generation of single applications as on the development of the software infrastructure for solving a range of application requirements. If this perspective is correct, then perhaps current programmer deployment practices are backwards. Skilled designers may be most valuable during the design consolidation phase. Hence, this model might lead to a sort of gentrification of software maintenance.

The disciplined reuse of object-oriented components, abstract classes, and frameworks stands in stark contrast to the casual reuse of conventional code and program skeletons. In order for the craftsmen to triumph over the scavengers, programmers and managers must commit themselves to designing for the long haul.

---

[1]This paper is a highly abridged of version of a longer, unpublished paper with the same title.

[2]Dept. of Computer Science, University. of Illinois at Urbana-Champaign, 1304 W. Springfield Ave.
Urbana, IL 61801 USA EMail: foote@cs.uiuc.edu Phone: (217) 333-3411 Fax: (217) 244-5876

[3]Gould has noted that natural taxonomic hierarchies exhibit maximal diversity during this early evolutionary phase.