# Prototyping as a Way of Life

## An Object-Oriented Lifecycle Perspective

(DRAFT)

Brian Foote

20 August 1988

## Abstract

Object-orieted systems are reputed to be effective for prototyping. They owe this reputation to their ability to allow a programmer to quickly reexploit a large budy of existing code to demonstrate a solution to a novel problem. The ability of object-oriented systems to promote code reuse, in turn, suggests an object-oriented perspective on the software lifecycle which emphasizes component and application framework evolution. This evolution can take place both within the lifecycle of individual applications or across that of a family of applications, as requirements change and experience in a given application domain accumulates. From this perspective, prototyping represents an expansionary, exploratory phase in the evolution of a part of a system #at can take place at any time during the evolution of the overall system. Prototyping episodes should be followed by design cunsulidatiun phases that restore to the system the structural integrity that can be undermined during such exploration. By allowing both exploratory and perfective design activity to pervade the lifecycle, the structural decay ~~erosion~~ that takes place during conventional application maintenance can be averted.

## Introduction

Object-oriented languages and systems have deservedly earned a reputation as being effective for prototyping. This reputation is probably due in no small part to the success seen using object-oriented systems for breadboarding user interfaces. Certainly those features which distinguish object-oriented languages from conventional ones, polymorphism, inheritance, and encapsulation ~~(BKE)~~, facilitate the rapid production of prototype applications. However, no language, no matter how powerful, can support the quick construction of elaborate applications, even at the mock-up or facade level, without the previous development of an extensive software infrastructure. The timely construction of a prototype application is simply not possible without the reexploitation of appropriate application specific and application independent components. Successful prototyping, then, is an exercise in software reuse. We believe that one of the principal strengths of object-oriented systems and languages is that they support both component and application level reuse, and that it is largely for this reason that they have proven so effective fur software prototyping.

This increased component and application level reuse potential can lead one, in turn, to a decidedly different view of the application development process and lifecycle. This greater reuse potential means that it is buth more likely that an existing component or application framework can be used to fulfill the requirements of the current project, and that code developed for a given project wii be reexploitable during subsequent projects. This reuse potential lowers the break even point fur efforts to make software components more general, thus making such efforts easier to justify. Object-oriented tools and techniques allow the lifecycle emphasis to be shifted

**from** the **design** and implementation of individual projects to the production and refinement of reusable **components and** application frameworks.

Prototyping ~~activity, given this perspective,~~ is not restricted to producing ~~a monolithic~~ rough draft ~~produced~~ at the start of a project's lifecycle. Instead, it pervades the lifecycle at all levels throughout and beyond individual product development efforts. Prototyping becomes a hallmark of an exploratory phase **in the development of both individual** software components and application frameworks. This activity is not limited to the initial design stages of a project, but may occur at *any time, even* (especially) during what is traditionally regarded as the maintenance phase of a project. Prototyping becomes a way of life rather than a dry run

It is important to distinguish the reuse capabilities found in object-oriented systems **from those present in more conventional systems. Object-oriented systems share with more traditional program development systems the ability to construct** libraries of application independent routines. However, components in an object-oriented system may, because of polymorphism, work within a wider range of contexts ~~and accept a wider variety of different arguments~~ than might components in a conventional library. **Inheritance allows library components to be specialized for specific** applications. Because of polymorphism and inheritance, the reuse potential for components in an object-oriented library is considerably greater than ~~that~~ for those **in conventional libraries**.

The availability of a rich vocabulary of library components can greatly simplify the application development process as well as the resulting applications themselves. Using conventional programming systems we can reuse application independent components with relative ease as well, but reusing the edifice that ties the components together so that they solve a problem of interest is usually possible only by physically copying the application [Foote 88]. Object-oriented systems provide an alternative to this “skeleton” program approach: object-oriented frameworks.

## Frameworks

A framework **is a** collection of cooperating **classes** ~~that~~ together define **a generic** or **template solution to a family of domain specific** problems ~~requirements~~. The best known frameworks, **such as MVC and** MacApp, define generic user interfaces. However, frameworks are by no means limited to user interface construction. For instance, the Battery Simulation [Foote 88) defines a framework for constructing realtime data-acquisition and experimental control applications. As object-oriented techniques are applied ~~to~~ in other application domains, frameworks for these domains can be expected to appear as well.

**Frameworks are often** characterized by an inversion of control in which the framework plays the role of a main program in coordinating and sequencing application activity. The user of a framework supplies methods that override specific **framework** behaviors to tailor it for a specific application. Frameworks can hence serve as dynamically extensible skeletons.

Changes to a superclass are inherited by its subclasses. However, there is n
way to propagate
[illegible]
change in a skeleton

Frameworks ~~are unlike~~ differ from skeletons~~, though~~, in that ~~their cores~~ they are dynamically shared ~~among~~ by all applications derived from them. (Object-oriented inheritance can be thought of as **having a super-Lamarkian flavor. Traits acquired by parents even after** the production of offspring are inherited.) This allows a framework to serve as the nucleus of a family of related applications as evolving requirements cause its members to diverge.

Does not follow; This is not a second point.

Deutsch [Deutsch 83] has pointed out that frameworks allow designs to be reused at different levels of abstraction. A framework can embody an abstract design that can become increasingly more concrete as one moves towards the **leaves of** the framework's inheritance hierarchy. (An **abstract framework** is like an **abstract** superclass in this respect.) The more abstract levels of the framework can come to resemble a high-level specification, while the lower levels fill in the implementation detail. A single **abstract design can serve as the basis for a number** of related concrete realizations of that design.

A framework's application specific behavior is usually defined by adding methods to subclasses of one or more of its classes. Each method must abide by the internal conventions of its superclasses. We call these *white-box* frameworks because the internal implementation details of the framework are visible to the application specific methods, and must be understood if the framework is to be successfully used [Johnson & Foote 88].

The relationships among the elements of a white-box framework tend to be rather informal. As a framework evolves, the relationships among its elements tend to become better defined. Portions of the framework frequently emerge as distinct components. Communication among components is then performed in conformity with the component's external protocol. The white-box elements become black-box components. We call such frameworks *black-box* frameworks. A black-box framework is easier to reuse than a white-box framework, since only the external protocol of the framework components need be understood, and since any component conforming to that protocol may be substituted for any other.

**White-box frameworks can play an important role during the early, exploratory phase of a system's evolution.** They encapsulate an informally organized part of a system while its structure is stii ~~the subject of experimentation~~ under development. As the system and problem domain become better understood, distinct black-box components begin to emerge.

Prototype implementations frequently exhibit white-box characteristics, since extensive opportunistic code-borrowing is frequently employed. It is almost invariably necessary to subsequently reorganize the class hierarchy to better reflect the structural demands that the new system component is making on the overall system.

We need examples from a real software project

Designing and implementing a system that meets its specifications is a challenging task in itself. Adding to this the requirement that the components of such a system anticipate and accommodate future requirements is a much more daunting task Designing a system from a fixed specification is a deductive process, whereas designing reusable classes and frameworks is an inductive one. Most often, a designer will know how to produce a general solution to a problem only after having seen several related specific solutions to it. A prototyping pass may cm occasion provide such experience, but ~~it will more typically accrue during the preventative maintenance phase of the product's lifecycle, or~~ that experience is more often gained during the implementation of successive members of a family of related products.

Say much more about this and give an example.

As a result, design itself can be seen as a process that pervades an object-oriented product's lifecycle. Indeed, some of the must valuable design effort, that involving the identification, generalization and refinement of framework components, will take place during the maintenance phase of the project. An interesting implication of this observation is that existing programmer deployment practices that place the most skilled designers on new projects and delegate maintenance to fledgling programmers may be less than optimal.

Designing reusable classes and frameworks is a difficult task that requires experience, judgement and skill. Even the best designers will seldom be able to divine optimal abstractions of a first attempt. Only experience within a given application domain can lead to the insights needed to product general components far it.

## Prototyping and the Software Lifecycle

The ability to quickly demonstrate the basic design ideas behind a system to a client is one of prototyping's greatest virtues. Object-oriented encapsulation and polymorphism can allow the substitution of alternate implementations, such as rough drafts, mock-ups, or simulations for the final components of an object-oriented system. This permits final implementation decisions to be deferred, and vital early experience to be gained.

Much of the motivation fur a prototyping pass can be found in Brook's classic admonition: *"Plan to throw one away; you will, anyhow"* [Brooks 75]. A prototype is frequently treated as a rough draft, or as vehicle for demonstrating the soundness of first level design concepts. Considerations such as efficiency, elegance, thoroughness and completeness are often treated as secondary during the construction of a prototype. During such an effort, the ability to co-opt existing code to the purposes of the prototype application can be quite valuable. Object-oriented inheritance allows one to casually borrow existing code with realtively much ~~minimal~~ effort. Existing frameworks and classes can be a ~~veritable~~ treasure trove of code and ideas ~~waiting to be subverted to the will of the prototyper~~.

Such opportunistic code borrowing, but this should never be mistaken for good design. Applications constructed in this fashion will usually have an ad hoc, haphazzard structure. The extensions made to existing classes may undermine their conceptual integrity as well. A prototypingpass should be seen as a prelude to a good design, and nut a substitue for it.

It is ironic that the very experience that can lead to the production of truely generic applications is largely squandered during the maintenance phase of the conventional software lifecycle. Consider the following quote from the Mythical Man-Month [Brooks 75]:

> Lehman and Belady have studied the history of successive releases in a large operating system. They find that the total number of modules increases linearly with release number, but that the number of modules affected increases exponentially with release number. All repairs tend to destroy the structure, to increase the entropy and disorder the system. Less and less effort is spend on fixing original design flaws; more and more is spent on fixing flaws introduced by earlier fixes. As time passes, the system becomes less and less well ordered....
>
> ...Systems program building is an entropy-decreasing process, hence inherently metastable. Program maintenance is an entropy increasing process, and even its most skillful execution only delays the subsidence of the system into unfixable obsolescence.

Maintanance, it would seem, is like fixing holes in a failing dyke. Eventually, ~~it~~ the dyke fails, and must be rebuilt. Only then are the lessons learned during it's tenure exploited.

We believe that a well mangaged team using object-oriented tools and techniques can stay this tide, by employing an incremental refinement strategy that distributes design exploration and consolidation across the entire lifecycle, and across both low level components and high-level application frameworks. Such a strategy should be flexible and opportunistic It would treat the production of an individual application as an opportunity not only to solve the problem at hand, but to lay the groundwork for related future efforts. Indeed, many software houses operate in just this fashion, treating the first application effort in a given domain as an effort to gain experience that will make subsequent efforts less painful Certainly attempts to do just this are not unique to the object-oriented world. However, the greater reuse potential of object-oriented components would seem to make them more likely to succeed.

Design, given this perspective, is an activity that pervades the software lifecycle. The encapsulation capabilities of object-oriented systems allow a system as a whole to be indifferent to localized design evolution and consolidation. By the same token, the

What does this mean?

constituent parts of a system can be made relatively immune to global changes in the system's structure.

This perspective resembles Boehm's spiral lifecycle model in a number of respects [Boehm 88].

There is a Darwinian quality to component reuse, in that a successful component will produce a lot of offspring (subclasses). This very success, can, in some cases, conspire to make a component less general as it evolves. This is because a previously general component can become unnecessarily constrained by code that addresses some specific new requirement in such a way as to undermine the components previous generality. This midlife generality loss can be mitigated in object-oriented systems via subclassing. The proliferation of requirements made of a successful component becomes represented in a white-box inheritance hierarchy or framework instead. Much is made in discussion of object-oriented design techniques of the ability of object-oriented architectures to model the underlying structure of the application domain. The ability of object-oriented architectures to reflect the structure of a system with evolving, diverging requirements in such as way as to make their evolution more managable is perhaps one of their greatest strengths.

As experience with a number of specific requirements sets is gained, the structure of a general solution to a range of application problems can reveal itself to the system designer. As the structure of such a solution becomes more obvious, the system will tend to evolve away from a rather casual white-box structure into a black-box structure. Not all Components will complete, or even begin, such an evolutionary journey. However, the greater reuse potential of the resulting ~~these~~ components ~~that do~~ can make a decision to lavish the resources necesssary to achieve such a result on them easier to justify.

## Conclusion

At the simplest level, the motivation for prototyping a system first can be characterized as an attempt to gain hindsight, that is, to answer the question: If hindsight is so valuable, how do we get it?

(Out of time...)

# References

[Apple 84]
Apple Computer, Inc.
*Lisa Toolkit 3.0*
Apple Computer, Cupertino, CA

[Boehm 88]
Barry W. Boehm
A Spiral Model of Software Development and Enhancement
Computer, May 1988, Volume 21, Number 5

[Brooks 75]
Frederick P. Brooks
*The Mythical Man-Month: Essays on Software Engineering*
Addison-Wesley, Reading MA, 1975

[Brooks 87]
Frederick P. Brooks
No Silver Bullet: Essence and Accidents of Software Engineering
IEEE Computer, Vol. 20, No. 4, April 1987

[Cox 86]
Brad Cox
*Object-Oriented Programming: An Evolutionary Approach*
Addison-Wesley, 1986

[ Deutsch 83]
L Peter Deutsch
Reusability in the Smalltalk- Programming System
pp 72-76 ITT Proceedings of the Workshop on Reusability in Programming
(reprinted in Tutorial on Software Reusability,
IEEE Computer Society Press, 1987)

[Foote 88]
Brian Foote
*Designing to Facilitate Change with Object-Oriented Frameworks*
Master's Thesis, University of Illinois at Urbana-Champaign, 1988

[Goldberg 83]
Adele Goldberg and David Robson
*Smalltalk-80: The Language and its Implementation*
Addison-Wesley, Reading MA, 1983

[Johnson & Kaplan 86 ]
Ralph E. Johnson and Simon Kaplan
Towards Reusable **Software** Designs and Implementations
Proceedings **of** the **Workshop** on
Future **Directions** in Computer Architecture and Software
May **5-7, 1986, Seabrook** island, **Charlston,** SC
Sponsored **by** the Army **Research Office (ARO)**

[Johnson **& Foote 88]**
Ralph E. **Johnson** and Brian Foote
Designing Reusable **Classes**
Journal **of** Object-Oriented Programming
Volume **1,** Number **2,** June/ July 1988

[Lehman 80)
M. M. Lehman
Programs, life cycles and the laws of software evolution
**Proc.** IEEE **68(9), 1060-76, 1980**

[McCracken 82)
Daniel D. McCracken and Michael A **Jackson**
Life Cycle **Concept** Considered **Harmful**
**Software** Engineering Notes, Volume **7,** Number 2,
April 1982, pages 29-32

**[Parikh 84]**
Girish **Parikh**
What is **Sofware** Maintenance Really? What is in a Name?
**Software** Engineering Notes, Volume 9, Number 2,
April 1984, pages **114-116**

**[Parikh 85]**
Girish **Parikh**
SOFTWARE MAINTENANCE NEWS
Software Engineering Notes, Volume 10, Number **2,**
April 1985, pages **58-60**

**[Schmucker 86]**
Kurt **J. Schmucker**
***Object-Oriented Programming for the Macintosh***
Hayden, **Hasbrouck** Heights, NJ, **1986**

**[Sheil 83]**
B. Sheil
Environments for Exploratory **Programming**
Datamaticm, February 1983

**[Stefik 86]**
Mark Stefik and Daniel G. Bobrow
Object-Oriented Programming Themes and Variations
AI Magazine 6(4): 40-62, Winter, 1986