Where Do Reusable Objects Come From?

Brian Foote

16 June 1995

On Objects

Diane Crawford (the Executive Editor of CACM) states:
Our most recent survey results indicate more than half the
Communications *readers rate* ***object-oriented programming*** *and*
related technologies their ***Number One*** *area of technical interest*

Fred Brooks *(from No Silver Bullet)*
Many students of the art hold out more hope for
object-oriented programming
than for any of the other technical fads of the day.
I am among them.

Why are Objects Popular?

Reuse

Productivity

Where Do Reusable Objects Come From?

Out of Boxes

From Programmer's Fingers

What Can be Reused?

Applications
Libraries

Abstract Classes
Components
Frameworks

Object and Reuse

Objects, in and of themselves, have a **greater reuse potential** than do conventional software components

Polymorphism

Allow a family of related objects to work correctly in a variety of different contexts

Inheritance

Allows a class to spawn an entourage of related subclasses
Allows core classes to be **shared unscathed**
Promotes the **emergence** of
Abstract Classes and Object-Oriented **Frameworks**

Encapsulation

Insulates evolving parts of a system
Maintains the integrity of **black-box** components and **Frameworks**

Object-Oriented Frameworks

A ***framework*** is
a collection of **abstract**, **cooperating classes** that
together comprise a **generic solution** to a
family of **domain specific requirements**

A framework can be thought of as the <u>embodiment</u> of an
abstract design
for solutions to problems drawn
from such a domain

Frameworks are often characterized by
an **inversion of control**
don't call us, we'll call you

Notables on Frameworks

Interface design and functional factoring
constitute the key intellectual content of software
and are far more difficult to create or recreate than code
L. Peter Deutsch

A framework is
the design for an applcation or subsystem
A set of abstract classes and the way objects
in those classes collaborate
Ralph E. Johnson

Framework Examples

Smalltalk-80 Model/View/Controller (MVC)
MacApp
InterViews
ET++
OWL
MFC

Taligent Frameworks
Choices
Battery Simulation
Accounts
Persistence/Distribution (e.g. SOM)
Languages

Object-Oriented Frameworks have made the GUI Revolution Possible

MVC
begat
Lisa Toolkit
begat
Macintosh Toolbox
begat
MacApp
begat
Interviews/ET++
begat
OWL/MFC

The analysis and evolution underlying MVC
now permeates the industry

White-Box vs. Black-Box Frameworks

White-Box Frameworks

Are extended primarily via inheritance
Each method must abide by
the internal conventions of its superclasses
Are informally organized internally, yet encapsulated
"Permissive" scope rules allow structure to undergo experimentation
Are characteristic of the **early**, **exploratory** phases of a framework's evolution

Black-Box Frameworks

As a framework evolves
portions of the framework emerge as distinct **components**
Communication is in conformity with the components external protocol

Are indicative of a better, **more mature** understanding
of the structureof a system
Are a goal toward which framework designers should aspire

Black-box frameworks are **easier** to **reuse** than white-box frameworks, since only the **external** component **protocols** need be understood, and any components conforming to these protocols may be substituted for an equivalent one

Where do Frameworks Come From?

Just because you wrote it
in an *object-oriented language*,
that doesn't *mean* its
reusable
or a **framework**

A first-pass design is seldom
truely resuable

The Fractal Model
of the Lifecycle of Resuable Objects

Resuable object-oriented abstract classes, components
and frameworks have **lifecyles** of their own
that are **distinct** from those of the applications
that incubate them

Objects **evolve** within and beyond the **applications**
that spawned them

Structure emerges as **objects evolve**

Because the **pattern** in which they evolve is **similar**
at each level, the overall pattern can be thought
of as a **fractal curve**

Iteration

Consider these two statements:

Iteration considered harmful

"Iteration" considered harmful

"Iteration" inspires visions of **wheels spinning**
or **dogs chasing** their **tails**,
or of **money** spinning **down** the **drain**...

Resuable objects are the
result of a highly **iterative** process

The **Fractal Model** tries to more **precisely characterize**
the **nature** of this **iterative** activity
and why **mature**, **resuable** objects result from it

Software Process Models

Boehm identified four, and added a fifth:

Code-and-Fix Model

Write some code, fix it, repeat

Stagewise and Waterfall Models

feasability -> requirements -> product design ->
detailed design -> coding -> integration -> unit testing ->
implementation -> operations and maintenance
Waterall approaches add a single level of feedback between
a given stage and the preceeding one

Salmon Ladders

A very durable "dead horse"

Evolutionary Development Model

Arose in response to the rigidity of the waterfall approach

Stages consist of successive, incremental, increasingly realistic prototypes

Transform Models

Specifications are directly translated into executable code

The specifications can be thought of as a very high level language for a given domain

Have only succeeded in well understood domains such as 4GLs and speadsheets

Spiral Model

An attempt to reconcile previous approaches,
particularly the **Waterfall** and **Evolutionary** Approaches

An **iterative** model
Each cycle unfolds as follows:
Objectives ->Constraints ->
Alternatives ->Risks ->
Risk Resoution ->Results ->
Plan for next phase ->
Commitment

Risk Driven

Kick the **nearest wolf** away from your door **first**

Radial dimension of the spiral is a function of
cumulative cost

Each cycle begins with the hypothesis that a given operational requirement might be met via a software development effort
When this **test fails**, the **spiral** is **broken**

What's Wrong with this Picture?

Various observers have noted that the ratio of
effort between **development** and **maintenance** is
40/60, **30/70**, and even **10/90**

Yet, **existing** process **models** emphasize
a **front-loaded** deployment of **resources** and **talent**

Maintenance comes in several flavors
Corrective 21% Adaptive 25%
Perfective 50% Others 4%

Where the **money** goes:
Adaptive and **Perfective** maintenance together
account for 75% of all maintenance costs
Clearly, **applications continue** to **evolve** beyond
the initial development phases of their lifecycles

From the Mythical Man-Month

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 spent fixing original design flaws;
more and more is spent on fixing flaws introduced in 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 obsolecence...

Maintenance, it would seem, is like fixing holes in a failing dyke. Eventually it fails, and must be rebuilt. Only then are lessons learned during its tenure exploited

The Lifecycle of an Object-Oriented Entity

The Fractal Model distinguishes three stages, or phases

Initial Design (Prototype) Phase
Exploratory (Expansionary) Phase
Design Consolodation (Generalization) Phase

Initial Design (Prototype) Phase

Usually a prototype (of sorts)
Informally structured
Employs expedient "code borrowing"
Constitutes a first pass at design
General design opportunities should be anticipated, where possible
Expedient design should never be mistaken for good design

Exploratory (Expansionary) Phase

Occurs when a design is successful

Frequently occurs during the "perfective" maintenance phase

Characterized by the spawning of a number of specialized variations on the original theme

Broad, shallow class hierarchies may develop

There is a risk of "mid-life" generality loss

These may be somewhat informally organized

A "white-box" phase

Design Consolodation (Generalization) Phase

Where **entropy reduction** gets done
Class hierarchy is reorganized and refactored
Abstract classes that reflect
structural regularities in the system emerge
Hierarchy comes to reflect the way you'd like to tell people
you got it there

Refactoring at this point in a systems's evolution allows the
designer to exploit the insights available from having
specialized a design to suit a number of applications
Hindsight, which is now availabe,
can be brought to bear on the redesign
A "black-box" phase

These phases **unfold** independently
at all levels of the system
At each level, objects move from the **prototyping** phase,
through the **exploratory** phase, and then into **consolidation**

Components, Abstract Classes, and **Frameworks**
all **evolve** in this fashion

Both exploration and consolidation
should be undertaken in an
"opportunistic" fashion
as opposed to the "risk-based" criterion of the Spiral Model

Refactoring

Refactoring is the engine of consolidation

Refactorings are **program transformations** that preserve program semantics, while improving structure

Refactoring has traditionally been done by hand, but **tools** are starting to **emerge**

Languages differ <u>significantly</u> in the degree to which they support refactoring

Rules for Finding Standard Protocols

Recursion Introduction
Eliminate Case Analysis
Reduce the Number of Arguments
Reduce the Size of Methods

Rules for Finding Abstract Classes

Class Hierarchies should be Deep and Narrow
The Top of a Class Hierarchy should be Abstract
Minimize Access to Variables
Subclasses should be Specializations

Rules for Finding Frameworks

Split large classes
Factor implementation differences into subcomponents
Separate methods that do not communicate
Send messages to components instead of to self
Reduce implicit parameter passing

Builders

One consequence of evolution of frameworks
from **white-box** to **black-box** is
that black-box components are
more amenable to manipulation
by program building tools than are white-box frameworks

Examples: VisualWorks, MFC, OWL

This will take on increasing importance as
dynamic components come into their own

Frameworks vs. Conventional Generalization Approaches

Dealing with volatile requirements

Conventional programming environments cope with volatile, evolving requirements in relatively haphazzard ways

Object-oriented approaches can <u>subsume</u> these, and allow applications themselves, as well as their constituent components, to be reused

Conventional	**Object Oriented**
Conditionalization	Variants as Subclasses
Parameterization	Prototype Objects
Metastisization	Frameworks

From Gould, Gilinsky, and German

Asymmetry of Lineages and the Direction of Evolutionary Time

Evolutionary groups generally concentrate diversity
during their early histories
Any taxonomic group builds to maximum diversity,
and declines to extinction

They suggest that their documented temporal asymettry
may display a *fractal character of self similarity at all scales*

[Early] asymmetry arises when an initial emptiness permits
*unusual **opportunity** for diversification*

Such patterns have been noted in the evolution
of Egyptian pottery designs, New England headstones,
and 19th century gas lanterns

Why Build Frameworks?

The embody, the invariant, hard-to-write
parts of an application

Maintenance burden is reduced, since all applications
share a common core
Lavishing effort on this core becomes cost effective
Tools become practical

New applications can leverage the effort
and design insight embodied in the framework

Coding is minimized, even eliminated

Framework construction <u>is</u> domain analysis

Risks and Pitfalls

Over-design: Proving Parkinson's Law by exhaustion
Outside application domains that are well understood,
the paths to generality may not be obvious

Attempts to exploit **generality** that's **not** really **there**
Or, time may be spent solving problems that will never arise

The construction of **Swiss Army Knives**
Like blank check ICs, components with
complex interfaces are hard to learn and reuse

The **hiding** of **power** in the name of generality
Designers may reject good domain specific solutions

A Fractal Methodology

Framework Cultivation
Framework Farming
Selective Breeding
Framework Eugenics

The idea: **Exploit** the **evolutionary characteristic** of object oriented frameworks by "**seeding**" the framework with a broad range of **representative requirements** drawn from the domain of interest

Building frameworks for **unfamiliar domains** is hard to do **up-front**
Building a domain-specific framework is a good way to **explore** a domain's **design space**

One program is easier to maintain than five.

Implications of this Lifecycle Perspective

Design pervades the lifecycle

Emphasis is not so much on single applications
as on developing the
software infrastructure for solving a range of application requirements

If **hindsight** is so **valuable**, the perhaps current **programmer deployment practices are backwards**
Skilled designers may be most valuable during the
design consolidation phase

Perhaps this perspective can lead to a
gentrification of maintenance

Greater reuse potential can make lavishing greater care,
resources, and attention on components pay

Patterns

Focus on **design ideas** at a more **abstract level** than frameworks

Is about the architectural insights that underlie
the code you have to write to
put together your abstract classes, components and frameworks

A movement whose goal is to get programmers
to write down and disseminate this information

A **literary** rather than **technical** form

Examples: The Gang of Four book
Strategy, **Composite**, **Command, Factory**

Summary

Objects have succeeded because they
really do encourage increased **productivity**
through **reuse**

Building systems out of objects
encourages the emergence of reusable
abstract classes, **components** and **frameworks**

Truely resuable objects can not be
designed through superior insight and sheer force of will
They are the instead the result of an
iterative evolutionary process
that unfolds at all levels in a system,
within and beyond the applications that spawn these objects.