Process and Predictability in an Age of Reuse

OOPSLA ’95 Workshop
on
Object-Oriented Process and Metrics
for Effort Estimation

Brian Foote

15 October 1995
Sunday

foote@cs.uiuc.edu
http://www.cpl.uiuc.edu/~bfoote

The Impact of Reuse on
Process and *Predictability*

Reuse Changes Everything
What you do
Where you start from

<u>Reusable</u> objects
are the result of an
iterative evolutionary *process*
that unfolds <u>within</u> and <u>beyond</u> applications

The **best process**
is <u>not</u> necessarily the
most *predictable* process

What Can be (Re)used?

Applications
Libraries

Abstract Classes
Components
Frameworks

Builders/Tools

Designs
Patterns
Experience
Architecture

The Fractal Model
of the Lifecycle of Reusable Objects

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

Reusable object-oriented abstract classes, components
and frameworks have **distinct lifecyles** of their own

Objects **evolve** <u>within</u> and <u>beyond</u> 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**

The Lifecycle of an Object-Oriented Entity

The Fractal Model distinguishes three stages, or phases

Initial Design (Prototype) Phase

Exploratory (Expansionary) Phase

Design Consolidation (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 Consolidation (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 system'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 available,
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
Components emerge as framworks evovle
Components are suited to **builders**

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

A Fractal Method

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

Triage and the Tyranny of the Timetable

Why are Programmer's Estimates so Bad?

Parkinson's Law
Unexpected Complexity
Future is the Hardest Thing to Predict
Macho Programmer Syndrome
Labor Pain Principle
Mushy Completion Criteria
Montgomery Scott Gambit

hours, days, weeks, months…

What Can We Predict?

Intercranial Communication Penalty: **5 x**

Effort devoted to Refactoring: **25%**

Framework Leverage: **80%-95%**

Virtuoso Penalty: **1.0-2.5 x**

Experience Multiplier: **1.0-10.0-? X**

White-box framework learning time:
proportional to ***volume***

Black-box framework learning time:
proportional to ***surface area***

To Win Big

(Re)use Component-based Frameworks

Build Domain-Specific Frameworks

Quality Talent

Deploy Along the Grain

Employ Fractal Process/Method

Leave the Predicting to Jean Dixon