The Selfish Class

Brian Foote
Joseph Yoder

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield
Urbana, IL 61801 USA

foote@cs.uiuc.edu (217) 328-3523
yoder@cs.uiuc.edu (217) 244-4695

Thursday, September 10, 1997

Abstract

This paper takes a *code's-eye view* of software reuse and evolution. A code-level artifact must be able to attract programmers in order to survive and flourish. The paper addresses the question of what an object might do to encourage programmers to (re-)use it, as opposed to using some other object, or building new ones. THE SELFISH CLASS pattern shows how focusing on code, rather than systems, processes, or personnel, can lead to fresh insights into software evolution and the forces that drive it.

The remaining patterns focus on more specific problems that evolving artifacts might confront. A software artifact that WORKS OUT OF THE BOX provides enough defaults to get the user up and running without needing to know anything about the artifact. An artifact that presents a LOW SURFACE-TO-VOLUME RATIO exposes its services via a relatively compact external interface, while encapsulating significant internal complexity. GENTLE LEARNING CURVE observes that artifacts that don't pose an undue learning burden on beginners can win users, while revealing additional complexity later. PROGRAMMING-BY-DIFFERENCE shows how code can adapt without mutating. FIRST ONE'S FREE suggests that giving your code away will help to make it popular. WINNING TEAM suggests that you can ride the coattails of a winning system to victory.

Introduction

Programs, and the artifacts from which they are built, have life-cycles that evolve within and beyond the applications that spawn them [Foote & Opdyke 1994]. Software is seldom built from the ground up anymore. Instead, programmers re-deploy a variety of artifacts as they confront changing requirements. Among these are function libraries, template applications, legacy code, and object-oriented abstract classes frameworks, and components. Each step of the way, programmers make choices among existing artifacts to determine which, if any, of them to (re-)use. There is something distinctly Darwinian about this process. The patterns presented herein take a *code's-eye view* of software evolution. They examine ideas drawn from evolutionary biology, to see whether they might inform our notions of how software evolves.

During the 1970's, sociobiologists proposed the notion that evolution could be best understood by focusing not on species, or even organisms, but on genes themselves as the basis for evolutionary selection. A particularly accessible treatment of these ideas was given by Richard Dawkins in *The Selfish Gene* [Dawkins 1989].

Dawkins suggested that any evolving system must be built around *replicators*. A replicator is an entity which is capable, via some process or mechanism, of creating exact (or nearly exact) copies of itself, in the presence of a suitable medium, the appropriate resources, etc. The best known replicators are of course based on the DNA molecule, and are the basis for all life as we know it.

Dawkins goes on to observe that replicators need not be based on DNA. They need not even be biological entities. Dawkins coined the term **meme** to refer to a replicator which, in effect, is an idea, which is propagated through a culture from mind to mind. While a successful gene might take many generations to predominate its gene pool, a promising meme can penetrate the meme pool at T1[1] speeds. It took nature four billion years to build a brain that could serve as a host to the array of memes that constitute human culture. It has taken only thousands of years for the meme pool to attain the richness and variety we see everywhere around us.

No matter what the nature of the replicator, its survival depends on three factors: its *longevity*, its *fecundity*, and its *fidelity*. In order to replicate, a replicator must survive long enough to make copies of itself. A replicator's fecundity is a measure of how prolific it is. Finally, a replicator's fidelity is the degree to which the copies it spawns retain a resemblance to the original replicator. Obviously, a replicator which is never around long enough to make copies of itself will not contribute to posterity. All other things being equal, a replicator should strive to leave as many copies of itself around as it can. However, as the copies become less and less faithful, the replicator's aim of preserving itself is undermined. If a replicator becomes extinct as a species evolves and adapts, then that might be good for the species, but it is bad for the replicator.

Therein lies the central thesis of the *Selfish Gene*, that replicators, will, over the course of any sustained processes of differential selection, come to behave as if their only interest was their own survival, to the exclusion of any other consideration. In particular, phenomena such as altruism, or other behavior that would appear to be exhibited for the good of a clan, or species, can be explained solely in terms of replicators looking out for number one.

THE SELFISH CLASS pattern examines how the sociobiological notion that evolving artifacts tend to behave in the interests of their own survival applies to *evolving code*. The radical shift in perspective that Dawkins proposed was that from the standpoint of a gene, the organism itself was just a convenient vehicle the gene employed to propagate itself. Our perspective is that programmers stand in just this sort of relationship to evolving code artifacts. The remaining six patterns examine specific strategies that code artifacts can employ to attract programmers.

---

[1] **T1** speed is 1,544,000 bits/sec.

The seven patterns in this paper are:

THE SELFISH CLASS

- WORKS OUT OF THE BOX
- LOW SURFACE-TO-VOLUME RATIO
- GENTLE LEARNING CURVE
- PROGRAMMING-BY-DIFFERENCE
- FIRST ONE'S FREE
- WINNING TEAM

THE SELFISH CLASS

also known as
SOFTWARE DARWINISM
PLUMAGE

> I want to claim almost *limitless power* for slightly inaccurate self-replicating entities, once they arise anywhere in the universe. This is because they become the basis for Darwinian selection, which, given enough generations, cumulatively builds systems of great complexity.
>
> ***Richard Dawkins*** – *The Selfish Gene*

We can think of software in terms of a pool of potentially reusable artifacts. In order for these artifacts to flourish, programmers must find them appealing. That is, programmers must elect to use these artifacts in lieu of other artifacts, and in lieu of writing new ones. A successful artifact may find its code copied into (replicated), or better yet, called from, an increasingly large number of programs.

What is the analogue to **gene** or **meme** in this tale? Is it the patterns that reside in the minds of software architects, which are expressed in individual artifacts as patterns like aisle and buttress are expressed in individual cathedrals? Or is it more appropriate to construe the artifacts themselves as the durable, evolving repositories of architectural insight? Our own belief tends towards this latter belief, that is, that *artifacts* embody *architecture*.

❖ ❖ ❖

Software artifacts that cannot attract programmers are not reused, and fade into oblivion.

Decisions regarding what objects to reuse, or whether to reuse any code at all, are subject to a host of forces. One of these forces[2] is the *Availability* of existing, potentially reusable code. *Cost* can be thought of as one dimension of *Availability*, since high cost has the effect of making an artifact less available, whereas low (or non-existent) cost increases availability. Reusable artifacts that are already part of a system are highly available, as are artifacts that are standard parts of programming environments. The enormous body of code that is available on the Internet makes it imperative that programmers scour the net to see what is available before building an artifact themselves. The marketplace itself is also becoming a more important source of reusable artifacts.

A primary consideration is the *Utility* of an artifact, or whether it in fact does what you want. The fundamental appeal of reuse is simply this: if there is something out there that already does what I need, then I'm done. A widely available artifact which solves a pervasive problem will become quite popular indeed.

A related force is the *Suitability* of an artifact to the task at hand. An artifact might be unsuitable to a particular task, even if it did what was needed, if for example, it was written for a different operating system, or tied to an incompatible GUI.

A particularly powerful force in the realm of reuse is *Comprehensibility*. If an artifact is easy to understand, programmers are more likely to use it than if it is inscrutable. Code that is easy to read is easier to modify. Comprehensibility is determined by the quality of the code itself, as well as any available examples and documentation. There may be differences among programmers in the perceived comprehensibility of a particular artifact based upon their backgrounds and experience. There are a variety of forces that drive programmers to rewrite artifacts that already exist. Vanity, and perversities in

[2] We list **forces** in *Italics* with a leading *Capital* letter.

the reward structure for reuse are certainly among them. However, artifacts that are too hard to understand remain one of the greatest obstacles to more wide-spread software reuse.

Another force is the *Reliability*, or robustness of the artifact. Code that is buggy, and hence is a source of aggravation for programmers who try to use it will (all other things being equal) be driven from the code pool. Interestingly, an artifact can protect itself by exhibiting incorrect behavior only in rare or unpredictable circumstances. These might be thought of as *non-fatal mutations*. A related force might be called *Fragility*. Fragile code is code which operates correctly out-of-the-box, but which breaks as soon as someone tries to change it.

***Therefore*, design artifacts that programmers will want to reuse. Strive to make them widely available. Make sure they reliably solve a useful problem in a direct and comprehensible fashion.**

Software artifacts that appeal to programmers will flourish. Those that do not will not. How might a potentially reusable artifact flourish? It can be an integral part of the code for a successful application. The success of such an application will guarantee that this code will remain a focus of programmer attention. However, the mere fact that thousands or millions of copies of an artifact are present in the object code of applications in the field does not help to propagate the code. Only its re-incorporation into subsequent versions of the applications in which it resides, or its incorporation into new applications, allows it to "reproduce".

In any system subject to such selection pressures, the artifacts which, for whatever reason, prove most effective at surviving these pressures will come, over time, to predominate. This is, after all, Darwinism in a nutshell. It follows then, that for a software artifact to win at this game, it must appeal to programmers. If it is able to do so it will prosper. If not, it shall not. We think this perspective is unique, in that rather than focusing on programmers or the software development process, it focuses on the code itself. This approach might be thought of as software sociobiology, since it takes the attitude that systems, users, and programmers exist merely as vehicles to abet the evolution of code. By analogy with the *selfish gene,* one might ponder the notion of the *selfish class*.

Species are subject, it is said to the law of the jungle. The jungle that anoints the winners and losers in the software domain is the marketplace. An inferior artifact may flourish if it is hosted by an application, that, for whatever reason, succeeds in the marketplace, which in turn makes the source code for the application containing the artifact the subject of additional development efforts. Life in the jungle can be merciless. For example, there is little to prevent a mass extinction of Macintosh software should the marketplace pull its platform out from under these applications.

❖ ❖ ❖

We present six additional patterns that help to complete THE SELFISH CLASS. A software artifact that WORKS OUT OF THE BOX is immediately able to exhibit useful behavior with minimal arguments or configuration. Enough defaults are provided to get the user up and running without needing to know anything about the system. An artifact that presents a LOW SURFACE-TO-VOLUME RATIO is easier to understand, and provides greater leverage than an artifact that presents a broader cross-section. GENTLE LEARNING CURVE admonishes designers to build artifacts that reveal their complexity and power gradually. PROGRAMMING-BY-DIFFERENCE shows how code can evolve without jeopardizing its identity. FIRST ONE'S FREE and WINNING TEAM contrast two strategies an artifact may employ to solve the problem of finding a broad audience.

WORKS OUT OF THE BOX

also known as
BATTERIES INCLUDED
WORKING EXAMPLE
GOOD FIRST IMPRESSION

When things we make work out-of-the-box, we not only provide immediate satisfaction, we also establish confidence that lays a foundation for long-term trust.

❖ ❖ ❖

If it is too much trouble to reuse an artifact, programmers may not bother.

There was a time when a programmer's reuse options were limited to a handful of standard library routines. Today, programmers are faced with a rich but daunting range of potential reuse opportunities. Simply evaluating the relative merits of each possibility can be an overwhelming task. Designers find that they don't have the time to carefully study each new, potentially useful artifact they come across. Instead, they often just try them out, and see what they can do.

Designers are more likely to reuse an object if it is easy to try it out and see how it works. A *good initial impression* can motivate the designer to spend the additional time to develop a detailed sense of an object's reuse potential. When the designer can actually see that an object works, he or she develops the confidence that a more detailed exploration will be time well spent. Conversely, if an artifact, such as a class, framework, component, or application, can't be made to work at all, or requires elaborate preparation in order to work, the designer may become discouraged, and look to other options.

***Therefore*, design objects so that they will exhibit reasonable behavior with default arguments. Provide everything a programmer needs to try out these objects. Make it as easy as possible for designers to see a working example.**

Reuse is an act of *Trust*. The designer must be confident not only that an object will merely conform to its public interface, but that the semantics associated with this interface are consistent with his or her needs. In other words, the designer must be able to understand how the objects works.

Of course, other factors, such as an artifact's *Heritage* influence whether programmers will trust it as well. A programmer may (or may not) regard code written by Microsoft, for example, as being more reliable, dependable, or polished that that from a less well know supplier.

When a designer first encounters a class or framework, he or she may not have the time to develop a full comprehension of the power and possibilities implicit in the these public interfaces. Hence, designers of such objects should strive the identify a minimal subset of this interface necessary to get a working version of their objects on-the-air.

Classes should be equipped with constructors that supply reasonable, working defaults for as many parameters as possible. Arcane, inscrutable mandatory parameters can be as annoying to a test driver as finding the brakes and clutch reversed. A successful test drive may encourage a longer look under the hood.

Abstract classes and frameworks should be bundled with at least one fully functional set of working, concrete subclasses or components (in other words, a working example).

Such example objects, classes, and frameworks should come with fully functional, working test programs, and these programs should be accompanied by sample input and output objects or files, where relevant. It's almost always not enough to merely document an artifact's interface. Providing working examples of how interfaces are actually used helps to resolve ambiguity and uncertainty and fosters confidence. Instructions that describe how run these examples should be included too. These minimal working examples are particularly important when the user is called upon to master complex interfaces. Users should not be left sitting frustrated on Christmas Day because the batteries were <u>not</u> included.

One way to learn how a new artifact works is to <u>methodically</u> <u>study</u> its code and documentation. Another is to dive in cold and <u>experiment</u> with the artifact, and thereby get a sense of what it can do. Initial success with such experiments will give programmers the confidence they need to delve more deeply into these object. Working examples can serve as test beds for exploratory experimentation. Such exploration can permit programmers to incrementally learn how to use an artifact, while keeping the growing example working. Programmers can progress from tinkering with this working examples to verify that they can rebuild them correctly, to a point where they feel that their command of the interface of the objects they are using is sufficient to justify embedding these objects in their own applications. At the beginning of this process, when every aspect of such a program will be new, and probably opaque to the programmer, it is particularly important that getting the artifact to work be as painless as possible.

Programs should exhibit reasonable behavior in the absence of arguments or pre-existing data. Designers should strive to ensure that a user's first encounter with their creations does not require elaborate preparation.

You should strive to make installation effortless, and minimize user configurations. A user will never be so ignorant of the ramifications of configuration decisions as upon the first time he or she confronts a new application. Hence, programs delivered in source form should build in response to `make`, or something like it. Installation programs should provide a default configuration button, and allow customization later on.

A design, and the objects that embody it, survive and evolve only if they are used. To be reused, they must attract programmers. To do this, they must make a good first impression. Like peacock feathers, an initial, painless presentation of impressive functionality can enhance an object's appeal. Objects which do this can flourish, and those which cannot, even if they are technically superior in other ways, may not.

The following box gives several examples of things that don't work out-of-the-box, and of things that do:

Bad

- Window systems that require elaborate event handling and detailed interaction with arcane API's to get a single, simple window on the air.
- Objects that can be built only using complex constructors, requiring too many unfamiliar arguments.
- Objects that are loosely coupled to others, that must themselves be prepared in fashions that are not carefully explained.

Good

- Frameworks that subsume the event loop.
- `Window new open.`
- Objects that are self-contained.
- `make all.`
- Supplying working grammars with `lex/yacc/flex/bison`.

One consequence of this pattern is that some abstract classes like `Window` will not be *pure* abstract classes. Instead, they can beget instances that exhibit simple, default behavior. C++-style pure abstract classes usually define interfaces. Abstract classes that work out-of-the-box will provide default behavior as well. It is, of course, possible to partition these functions between two classes: a superclass that defines an interface, and a subclass that provides the default behavior.

❖ ❖ ❖

There are two patterns that might be gleaned from WORKS OUT-OF-THE-BOX and presented as distinct patterns: DEFAULT ARGUMENTS, and WORKING EXAMPLES.

LOW SURFACE-TO-VOLUME RATIO

also known as
HIGH VOLUME-TO-SURFACE RATIO
SIMPLE INTERFACE

A sphere has the lowest surface-to-volume ratio of any three dimensional object

Objects that allow a user to control a large volume of complex machinery with a small, simple interface are more likely to flourish than those that don't. See THE SELFISH CLASS. An object with a simple interface relative to its internal complexity may be more likely to WORK OUT OF THE BOX.

❖ ❖ ❖

Objects with complex interfaces that conceal few of their internals are hard to understand and reuse.

Objects that marshal the resources to perform complex tasks in response to simple external protocols provide the programmer with a high degree of leverage and power. To be effective though, it must be easy to understand how these objects work. The interfaces must be easy to *Comprehend*. This comprehension must be based on the clarity of the interface itself, and of its accompanying documentation and examples, and should not depend on a thorough understanding of the object's internals.

Several forces can conspire to prevent the emergence of these self-contained artifacts. Complex legacy objects may be *Fragile* and difficult to comprehend. The lack of domain experience that inevitably accompanies the start of a project can lead to expedient, first-pass prototype code that properly defers structural improvement in the name of getting the system on the air. And of course, some problems are just inherently complicated, and will demand that complex artifacts be produced in their image.

***Therefore*, Design objects with low surface to volume ratios, that is, objects with small external interfaces, or surface areas, that encapsulate a large volume of internal complexity.**

In other words, programmers are attracted to good abstractions, with clear separations between the public interfaces and the system's internals. A good abstraction becomes an element of the vocabulary of a domain-specific language. These abstractions must be well encapsulated, in order to make it easy to use them in multiple contexts. Encapsulation is not enough. An artifact may be self-contained, but still expose all its internals to the public. Such an artifact will, of course, have a surface to volume ration of one. Abstraction is not enough. An artifact might present a tidy interface to the world, but draw in with it a tangled, gangling collection of global variables and resources.

If a programmer needs a thorough understanding of an object's internal workings, the programmer will often find it easier to write his or her own code. When it comes to being certain about how an object works, there is simply no substitute for having written it yourself. If an artifact encapsulates too little functionality, programmers may find it preferable to write the code themselves as well. For example, since no collection classes were supplied with C++ up until the recent development of the Standard Template Library, C++ programmers have historically written their own collection classes. It was not the case that no collection code existed. Instead libraries like the NIH package [Gorlen 1986] were not universally available, and were not suitable to a number of platforms. However, an overriding factor was the fact that, in the simple cases, a programmer could be certain of the architecture of his or her own abstractions if he or she built his or her own. Since the simple cases are by far the most common ones, it can be some time before these home-brew artifacts evolve into mature, reusable artifacts.

This was not the case with Smalltalk-80 [Goldberg & Robson 1983]. Smalltalk's `Collection` classes provide a convenient, comprehensible, and powerful vocabulary for manipulating aggregates that most programmers have come to regard as an integral part of the language.

An artifact with a low surface-to-volume ratio presents a compact vocabulary, with a small number of powerful nouns and verbs. The verbs are the artifact's operations, and the nouns are its arguments. To keep the surface area low, the designer must strive to find a concise set of nouns and verbs that can be expressively combined.

For instance, lengthy parameter lists can be consolidated into parameter objects [Johnson & Foote 1988]. These objects, in turn, can be constructed with default fields that spare the user the burden of making decisions about them one at a time.

When mature frameworks and components are available, much of a programmer's time will be spent gluing these elements together. When there is a good fit between the operations and arguments of different elements, less glue is needed to get them to work together. One the other hand, if getting elements to work together requires cumbersome argument packing and unpacking, and awkward protocol translation, programmers may become discouraged, and turn elsewhere.

Another technique that can be used to minimize the effective surface area of an object is to provide simple template methods [Gamma et. al. 1995] that call other methods that fill in the details. These methods, in turn, can be defined by subclasses of an original class, thereby permitting customization and tailoring.

Ironically, *Comprehensibility* can work against an artifact's preservation, by causing it to mutate more rapidly than artifacts that are harder to understand. Such objects may proliferate, but quickly become unrecognizable. This is often the fate of simple template applications and boilerplate code. An object with a clear interface and hard to understand internals may remain relatively intact. This is a good example of the phenomenon Dawkins called *selfishness*, whereby the needs of the individual object seem at odds with those of the system as a whole.

Indeed, it's fair to ask what role this phenomenon plays in the proliferation of mediocre code. Is it this case that highly comprehensible code, by virtue of being easy to modify, will inevitably be supplanted by increasing less elegant code until some equilibrium is achieved between *Comprehensibility* and *Fragility*? Perhaps *simple on the outside/fragile on the inside* can be an effective survival strategy for evolving artifacts.

Of course, from the perspective of someone on the outside looking in, allowing good code to evolve into bad code is obviously a bad thing. We don't advocate making good code bad. Instead, we hope that by making programmers aware of how mediocre code might protect itself, we might encourage them to counteract this tendency by refactoring their code so that it is both general <u>and</u> readable.

❖ ❖ ❖

A system in the early, PROTOTYPE PHASE of its evolution is often haphazardly organized [Foote & Opdyke 1994]. They may be thought of as white boxes, because so many internal details are left exposed. As systems are re-deployed, during the EXPLORATORY PHASE, encapsulation often erodes even further, as first-pass guesses as to what needed to be hidden are proven wrong by experience. It is only as an artifact matures, after being reused in a variety of contexts, that the interfaces that have really proven valuable can be discerned. It is for this reason that a CONSOLIDATION PHASE emphasizing refactoring, late in the life-cycle, is so valuable. It is during this phase that good abstractions emerge, and white-box, inheritance-based artifacts give way to black-box components. [Roberts & Johnson 1996] presents these notions as the WHITE-BOX FRAMEWORK and BLACK-BOX FRAMEWORK patterns.

Hence, a high surface-to-volume ratio may be seen as a sign of immaturity, or of evolutionary flux. A high surface-to-volume ratio might also be a sign that an object needs to be factored into several distinct objects. In other words, there may be distinct objects trapped inside, crying to get out.

The GENTLE LEARNING CURVE pattern states, in essence, that an object should strive to give the appearance of having a lower surface area than it really does to new users. The full interface is gradually exposed as the user learns his or her way around the object.

GENTLE LEARNING CURVE

also known as
GRADUAL DISCLOSURE
INCREMENTAL REVELATION
FOLD-OUT INTERFACES

❖ ❖ ❖

Complex interfaces can overwhelm beginning users.

In order to be *Flexible* and *Adaptable*, artifacts may provide a variety of *Customizable* and *Tailorable* interfaces. The variety and complexity of these interfaces can be confusing and intimidating to users who are unfamiliar with an artifact. These users are precisely the ones an artifact must attract if it is to broaden its mind-share. Artifacts that exhibit high *Utility* without imposing an up-front learning burden on the programmer have an advantage over those that do not.

An important force here is *Time*. Given unlimited time, a programmer might elect to learn a complex but powerful artifact as an investment in his or her skills. However, it is now far more likely that such a programmer, faced with the overwhelming array of choices that the marketplace now, will opt for the easier to *Comprehend* artifact every time.

***Therefore*, design artifacts that allow users to start with a simple subset of their capabilities, and permit them to gradually master more complex capabilities as they go along.**

An artifact should require the mastery of a simple, minimal set of capabilities at first. One way to do this is to provide defaults. Default constructors and arguments not only allow an artifact to WORK OUT OF THE BOX. They also provide direct, simple templates and examples that show how an artifact can be used to perform likely tasks. A GENTLE LEARNING CURVE provides many of the same benefits that WORKS OUT OF THE BOX does early on, while rewarding more advanced programmers with more intricate and powerful capabilities later on.

One way to do this is to use what might be called *a fold-out interface*. The name “fold-out interface” was inspired by the PKZIP help system, which displays a list of rudimentary switches when the user asks for help, and reveals more complex options incrementally in the manner of a fold-out programmer’s reference card. The simple options are those which provide more or less out-of-the-box functionality while providing the appearance of a LOW SURFACE TO VOLUME RATIO. Then as the user becomes more proficient and comfortable, ways to get to additional features of the interface are exposed

As users become more sophisticated, they can gradually delve more deeply into more powerful and exotic aspects of the artifacts capabilities. For instance, Unix and DOS commands are usually designed to perform a common useful tasks with no arguments at all. In other cases, the no-argument form provides a simple help message. More complicated forms can be learned as the user gains confidence with these commands. Manuals and online documentation can then provide examples of a command’s more powerful but arcane variants.

VisualWorks provides a couple of good working examples of GENTLE LEARNING CURVE. One example can be seen in the `ApplicationModel` class. A method called open is provided for opening an applications on a default canvas. As the user becomes more proficient, they can develop different canvases and use the `openInterface:` method to dynamically choose which canvas to open. The user can also add new widgets, enable/disable widgets, etc by using the `postBuild:` and `postOpen:` methods.

Also, when new developers learn Smalltalk, they are taught the basics of the collection hierarchy by learning simple methods such as `do:` and `add:`. After becoming comfortable with using the basics of collections, developers are pointed to using more advanced methods such as `collect:`, `select:`, `inject:into:`, and `contains:` for doing the same work that they can do with the basic methods.

❖ ❖ ❖

An artifact which presents a GENTLE LEARNING CURVE can appeal to programmers because it places a minimal burden on prospective users at precisely the time they are choosing from among it and its alternatives. There are, of course, artifacts that require programmers to immerse themselves in all an artifact's details before they can be used. In other words, they present *steep learning curves*. Usually these are artifacts that are essentially in monopoly positions. Programmers must learn them, because there are no simpler alternatives. When this is the case, programmers will dutifully, but reluctantly, bite the bullet and master these artifacts.

PROGRAMMING-BY-DIFFERENCE

also known as
WRAP IT, DON'T BAG IT

> The true value of object-oriented techniques as opposed to conventional programming techniques is not that they can do things conventional techniques can't, but that they can often extend behavior by adding new code in cases where conventional techniques would require editing existing code instead. Objects are good because they allow new concepts to be added to a system without modifying previously existing code. Methods are good because they permit adding functionality to a system without modifying previous existing code. Classes are good because they enable using the behavior of one object as part of the behavior of another without modifying previously existing code
>
> The Treaty of Orlando [Stein et. al. 1988]

❖ ❖ ❖

You want to adapt an artifact to address new requirements while maintaining the artifact's integrity.

There is a tension between the forces of change and the forces that encourage stability. One force that leads to wrapping is a lack of *Availability*. If the code for an artifact is unavailable, it may be difficult (to say the least) to change it. For example, the artifact may reside in a legacy system, and provide a service which the developer is compelled to use. One can use these existing services by writing wrappers around these services. Even in those instances were rewriting the code might be a better long term strategy, expediency might dictate that simple wrappers be used with legacy code to get something running quickly.

More commonly, artifacts are *Available*, but may be *Fragile*, or difficult to *Comprehend*. Under these circumstances, changes to them might undermine not just the new application, but existing code that shares the artifact. Isolating changes in an entourage of distinct subclasses or components that leave the original artifact untouched avoids these pitfalls. As such, this strategy is often the ideal way to balance new requirements against a reluctance to jeopardize existing, working artifacts.

***Therefore*, use translators, subclasses, and/or wrappers to supply new state or behavior while leaving the original artifact intact.**

A translator is an adapter that takes results from some existing artifacts and converts them to some other format that may be more usable by other artifacts. A translator can take existing output, say from a legacy system, and do some work on it and put it in some new artifact, say a relational database, which is more easily accessible. Sometimes this is done by using the INTERPRETER pattern [Gamma et. al. 1995].

Sub-classing preserves behavior of the original code while making it easy to extend and/or specialize. Ralph Johnson calls this style of programming PROGRAMMING-BY-DIFFERENCE [Johnson & Foote 1988]. Composition allows one to use two or more existing artifacts to get the desired results. Of course, these approaches can be mixed.

There is one quality of reusable artifacts that does not have a direct analog in biology. That is, when a reusable artifact is shared and called, rather than copied, every program that refers to it immediately benefits from any changes made to the artifact. One might say there is a Super-Lamarkian[3] character to this kind of sharing, in that descendants can benefit immediately and retroactively from changes made to their parents.

---

[3] **Lamarkism** is the belief that characteristics acquired during an organism's lifetime can be inherited by that organism's offspring.

❖ ❖ ❖

The ADAPTER, COMPOSITE, DECORATOR, FACADE, and INTERPRETER patterns [Gamma et. al. 1995] all provide ways by which an existing artifact might be wrapped to provide extended functionality.

Approaches like this one that preserve the integrity of existing artifacts are particularly valuable during the PROTOTYPE and EXPANSIONARY PHASES of the life-cycle, since they allow objects to be re-deployed in new contexts without the potential disruption to existing applications that might insue if they were subject to invasive modification.

An artifact with a LOW SURFACE-TO-VOLUME RATIO will be easier to wrap than one with a more complex external interface. An artifact that is widely available, and that WORKS OUT OF THE BOX, is more likely to be incorporated in new work than one that is less accommodating.

FIRST ONE'S FREE

also known as
NETSCAPE NOW
DOOM
COPYLEFT
FREEWARE/SHAREWARE

This pattern helps to solve the problem presented in SELFISH CLASS by encouraging the wide dissemination of an artifact.

❖ ❖ ❖

In order to survive, an artifact must become widely *Available*.

No matter how good an artifact is, it will have no chance of proliferating if other programmers never see it. In order to survive, an artifact must gain a wide audience. One of the forces that may limit an artifact's *Availability* is it's *Cost*. A countervailing force is *Bankruptcy*. With this strategy, there is always the risk of giving away the store.

***Therefore*, give the artifact away.**

One scheme that artifacts use to gain wide dissemination is the FIRST ONE'S FREE strategy. Urban legend has it that drug peddlers would give away introductory samples of their otherwise expensive wares as a way of recruiting new addicts. Once a potential customer was addicted, this 100% discount no longer applied.

All other things being equal, an artifact which is less expensive, and easier to acquire, will proliferate faster and further than one that is not. When *Cost* drops to next to nothing, its usual effect on an artifact's *Availability* is minimized. In the software world, these discounts often reach 100% and beyond, as in the case of promotions like American OnLine's or Compuserve's, that include postage and floppy disks. If only a small number of customers becomes dependent on a steady diet of services from these suppliers, this strategy can prove profitable in the long run.

The World Wide Web browser wars provide a striking example of this dissemination strategy in action. To gain market share, Netscape and Microsoft simply give away their Navigator and Explorer products.

The Free Software Foundation's GNU project has long given dozens of complex applications, complete with source code. Their novel *copyleft* license requires that new applications derived from their code be distributed under the same terms as the original source code. The focus on source code distinguishes this approach from the freeware/shareware and first one's free variants.

❖ ❖ ❖

WINNING TEAM

also known as
PIGGYBACKING
REMORA[4] STRATEGY
WHATEVER FOR WINDOWS
HITCH YOUR WAGON TO A STAR

❖ ❖ ❖

In order to survive, an artifact must become widely available.

No matter how good an artifact is, it will have no chance of proliferating if other programmers never see it. In order to survive, an artifact must gain a wide audience. One way an artifact can become widely *Available* is to hitch a ride on a popular platform. The way for an artifact to win big is to have its code universally included with every copy of a system that ships. One drawback to this strategy is that the artifact loses its *Autonomy*. It's fate becomes tied to that of it's platform.

***Therefore*, strive to become bundled with a popular platform.**

Interestingly, every successful commercial object-oriented framework has been distributed with full source code. In the case of the GUI frameworks, such as Model-View-Controller (MVC), MacApp, Object Windows Library (OWL), and Microsoft Foundation Classes (MFC), the framework, with its source and documentation, is bundled with the development tools for a particular language/platform pair.

It's interesting that giving away the store in this fashion doesn't seem to undermine the product. The phenomenon seems to work something like this: give away a thousand lines of code and ruin your business, give away a million lines of code and ruin the other guy's business. While it may be possible to steal key ideas from a small body of code, overtly copying a million line product without being obvious is more difficult. More importantly, few organizations are willing to assume the comprehension burden and maintenance responsibility for a large body of someone else's code.

A drawback to this strategy is the possibility of mass extinction: If the platform itself looses favor, all the artifacts that depend on it, regardless of their individual strengths or weaknesses, will perish as well.

The Java [Gosling et. al. 1996] programming language has employed both the FIRST ONE'S FREE and WINNING TEAM strategies. The runtime environment necessary to support Java applets hitches a ride inside the popular World Wide Web browsers, which in turn are given away. Hence, Java gleans the benefits of both, without incurring the risk associated with tying itself to a specific platform.

❖ ❖ ❖

[4] The **remora** is a parasitic marine fish of the family *Echeneidae*.

Conclusion

Software that does not evolve will die [Foote & Yoder 1995]. Of course, an anonymous chunk of code trapped in a successful application may persist inside it for decades, dutifully doing what ever it is it does. However, if this application is so inscrutable and fragile that programmers never dare touch it, our chunk of code is a sterile as a mule, and will never have the opportunity to propagate. Artifacts flourish only by attracting programmers. This, in turn, ultimately leads to specialization. One of the major benefits of reuse is that responsibility for the maintenance, improvement, and evolution of a widely reused artifact that retains its distinct identity can remain in one place.

We think an interesting and unexpected result of our *code's eye* perspective on reuse is that phenomena like selfishness, which have been observed by zoologists in their own studies of evolution, may arise as software evolves as well. Software, it would seem, plays by the same rules as everyone else when it comes to evolution. Of course, just as biology is not necessarily destiny, we need not give in to entropic evolutionary tendencies in our code. Judicious refactoring, good documentation, and an eye for architectural integrity can overcome the impulse on the part of our artifacts to just look out for themselves.

The programmer with a commitment to reuse, while focusing on the problem at hand, keeps one eye towards the future, and the potential that the artifact he or she is building will find a wider audience within or beyond the applications it currently inhabits. This programmer is not content that his or her craftsmanship be known only to God, like the artisans that carved gargoyles that could be viewed only from perspectives that would never be seen by mortals.

Acknowledgments

We are grateful to the members of the University of Illinois Patterns Group: Jeff Barcalow, John Brant, Ian Chai, Charles Herring, Ralph Johnson, Mark Kendrat, Donald Roberts, and Dmitry Zelenko who soldiered through a particularly rough earlier draft of this paper, and provided a variety of commentary and advice, a good deal of which was genuinely useful.

We would also like to thank Desmond D'Souza who shepherded our next draft.

Doug Lea provided a number of valuable insights after reviewing a late version of these patterns.

We'd also like to gratefully acknowledge Tom Lee and Vic Alcott of General Electric, who helped us to gain GE's permission to use the vintage air conditioner advertisement depicted in the WORKS OUT OF THE BOX pattern.

Finally, we'd like to express our gratitude, to the PLoP '96 writers workshops participants, who, with their candid and constructive criticism, helped us shape and polish the current incarnations of these patterns.

References

[Alexander 1979]
Christopher Alexander
The Timeless Way of Building
Oxford University Press, Oxford, UK, 1979

[Alexander et. al 1977]
C Alexander, S. Ishikawa, and M. Silverstein
A Pattern Language
Oxford University Press, Oxford, UK, 1977

[Dawkins 1989]
Richard Dawkins
The Selfish Gene
Oxford University Press, Oxford, UK.
Second Edition, 1989

[Foote & Opdyke 1994]
Brian Foote and William F. Opdyke
Life-cycle and Refactoring Patterns that Support Evolution and Reuse
First Conference on Pattern Languages of Programs (PLoP '94)
Monticello, Illinois, August 1994

Pattern Languages of Program Design
edited by James O. Coplien and Douglas C. Schmidt
Addison-Wesley, 1995

[Foote & Yoder 1995]
Brian Foote and Joseph Yoder
Architecture, Evolution, and Metamorphosis
Second Conference on Pattern
Languages of Programs (PLoP '95)
Monticello, Illinois, September 1995
Pattern Languages of Program Design 2
edited by John Vlissides, James O. Coplein,
and Norman L. Kerth.
Addison-Wesley, 1996

[Gamma et. al 1995]
Eric Gamma, Richard Helm, Ralph Johnson,
and John Vlissides
Design Patterns:
Elements of Reusable Object-Oriented Software
Addison-Wesley, Reading, MA, 1995

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

[Gorlen 1986]
Keith Gorlen
Object-Oriented Program Support (OOPS)
Reference Manual
National Institutes of Health, May 1986

[Gosling et. al. 1996]
James Gosling, Bill Joy, and Guy Steele
The Java™ Language Specification
Addison-Wesley, Reading, MA, 1983

[Johnson & Foote 1988]
Ralph E. Johnson and Brian Foote
Designing Reusable Classes
Journal of Object-Oriented Programming
Volume 1, Number 2, June/July 1988
pages 22-35

[Roberts & Johnson 1996]
Don Roberts and Ralph E. Johnson
Evolve Frameworks into Domain-Specific
Languages
PLoP '96 submission

[Stein et. al. 1988]
Lynn Andrea Stein, Henry Lieberman,
and David Ungar
A Shared View of Sharing: The Treaty of Orlando
Object-Oriented Concepts, Databases, and
Applications
edited by Won Kim and Frederick H. Lochovsky
ACM Press, New York, New York, 1989