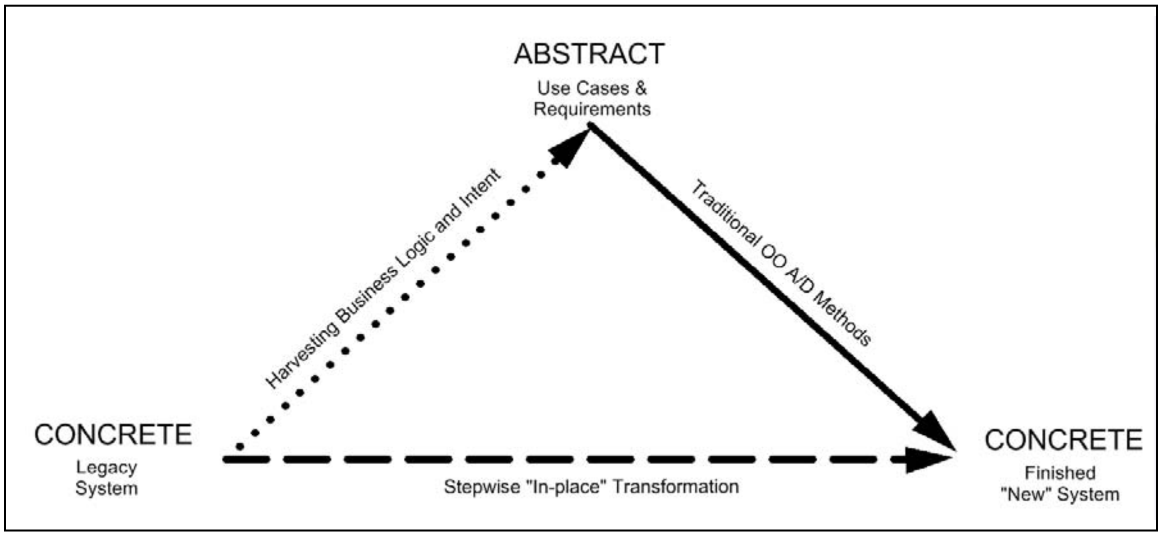
COMPUTING INTEGRITY
INCORPORATED
60 Belvedere Avenue
Point Richmond, CA 94801-4023
510.233.5400 Sales
510-233.5444 Support
510.233.5446 Facsimile

PROGRESS Application Partner

The Transformation Triangle

24 June 2008

Thomas Mercer-Hursh


ABSTRACT
Use Cases & Requirements
Harvesting Business Logic and Intent
Traditional OO A/D Methods
CONCRETE
Legacy System
Stepwise "In-place" Transformation
CONCRETE
Finished "New" System

Development that follows traditional Object-Oriented Analysis & Design (OOAD) methodologies begins with tools such as Use Cases and Requirements specifications, which are very abstracted from the target software. Through a series of steps, some of which may be automated transformations, these abstract elements are gradually converted into more and more concrete model forms, which are closer and closer to the target software that is the focus of the development effort. In particular, using the techniques of Model-Driven Architecture (MDA), models are transformed from Computationally Independent Models (CIM) to Platform Independent Models (PIM) to Platform Specific Models (PSM). For example, the PIM might be a model which fully specifies the target software, but without a User Interface, and the PSM resulting from one stage of the transformation would then have a specific User Interface applied to that model. The transition of PIM→PSM can occur multiple times, with the PSM from one step providing the PIM of the next step. Thus, the notions of Independent and Specific are relative to the context.

While this general approach can be highly successful for developing new systems, when considering the transformation of an existing legacy system one has a somewhat different problem. In this case the concrete, “finished” system already exists, but something about it is “wrong”. It might use an outdated technology, an old user interface technology or style, or have a monolithic architecture, which is not as well suited as service-oriented architectures for nimble response to changing business conditions. Whatever its flaw or flaws, the desire for transformation indicates that there is a need for substantial change in the application, far beyond the day to day maintenance, modifications, and small new features which are typical of mature applications.

Such legacy applications are also often poorly documented. Even if there were design documents created at the time the software was originally written, it is typical for these to be so badly out of date as to be of limited use as documentation for the current system. The original intent and purpose which drove the current design is reflected only in the details of the existing software, where it can be difficult to discern. Thus, it is very difficult to imagine a transformation process which would reverse the normal OOAD process by creating abstract model forms from concrete ones. Even when one is working in a normal OOAD context where the software has been created by moving from abstract to concrete forms, if one modifies the code rather than the model and wants to bring the model back into line with the revised code, only the simplest and most straightforward transformations can be reversed and the more abstract levels can only be corrected manually.

If there were some “magic” tool which would create the abstract components corresponding to an existing legacy application, this would be enormously attractive. Then, one could perform some reorganization of those elements to better suit modern architectural practice and use well understood OOAD processes to create a fresh application with a new architecture and a new user interface. This fresh application could be just as modern in its technology and principles as one wanted. However, no such “magic” tool exists.

The difficulty of this problem can be appreciated by considering a simple example from a typical legacy application. Consider a simple file maintenance program. This program or group of programs might easily be hundreds of lines of code. Exactly what those hundreds of lines would look like are likely to vary significantly from one application to another based on both the state of technology at the time it was written and the “style” of the shop where it was created. All or most file maintenance programs from the same application are likely to be similar in structure, unless there was a shift in style during the life of the application, but the programs from one application are likely to be quite different in details than those from another application. This typical pattern of how such programs are written in a given shop constitutes a kind of intrinsic framework.

If one was working in the context of such an intrinsic framework and were given the schema for the table to be maintained and any special business logic which applied, one would have little difficulty in copying the framework pattern and applying it to these new elements. But, when we are trying to create an abstract model, we want to extract just the schema and business rules because the rest of the framework will be discarded in favor of the framework to be used for the new architecture. Seen from this perspective, the intrinsic framework of the legacy application is “noise” which is getting in the way of us extracting the “signal” of the key information.

To date, there is no software that can do this kind of “noise removal” so moving from concrete to abstract is a largely manual process. Indeed, in order to understand what one reads in the code, it is likely that one has to also interview the users in order to determine why the code behaves in a certain way, so the effort required may not be substantially less than simply designing the system from scratch. In fact, the biggest benefit in this context of having the legacy code as a starting point may be that one has a very concrete documentation of what has been considered correct behavior, which is not always easily determined from user interviews. If there were software that could automate or semi-automate large parts of this process, it would be perhaps the most desirable approach to transformation since we would end up with a completely fresh application using all of the most modern technology and architectural principles. But, without such software, a complete transformation along this path can be extremely expensive and time consuming.

In any transformation effort there may be selected parts of the application that are appropriate for this kind of complete re-invention. This might be, for example, a module which was known to need substantial redesign anyway, so that taking the approach of doing a fresh implementation from core principles can be less time-consuming than doing an in-place redesign. Such selective re-invention tends to work only when there is a well-identified and well-separated module that is appropriate for more intensive upgrade.

While such re-invention is obviously desirable, since the resulting application is fully modernized, many companies cannot afford the massive effort required. The alternative is to perform some kind of stepwise transformation, making incremental improvements over a period of time. It should be noted that applying a stepwise approach is likely to require more total effort and a greater length of time than direct re-invention, if one carries it through to the same end point, but most companies utilizing a stepwise approach only make partial transformations, thus moderating the cost and effort.

One of the forms of stepwise transformation which has become popular in recent years is a progressive migration to a service-oriented architecture (SOA). In this approach, one typically begins by implementing an Enterprise Service Bus (ESB) to serve as the messaging backbone and then focuses on a small series of key projects to convert them from the legacy form to a service. One classic example of this kind of re-engineering might be an application which had some complex body of code which was more or less replicated in multiple modules or systems, each differing slightly in functionality. Since changes in one version need to be manually replicated in the other systems, such situations are error prone and labor intensive. By centralizing all versions of this code in a common service, which is then called by all of the consumers, future changes can be implemented with consistency and lack of disruption. The loose coupling characteristic of SOAs means that the system is more resilient to change, typically allowing individual services to be upgraded as needed without changes to other services.

In such progressive SOA re-engineering, there is often a very high level of harvesting of code since the approach tends to be more one of putting a wrapper on existing code than it is one of creating entirely new code. At some point, the "insides" of a service may get rewritten if this is indicated, but in many cases one simply proceeds with the old code wrapped in a new interface and the architectural shift is in the move from a tightly coupled monolithic system to a loosely coupled collection of services. This is very different from the architectural shift which results from re-invention. While re-invention might include a shift to SOA, it also produces changes to all the details of the application as well. E.g., in the ABL world, it would not be surprising to find the code inside a service developed by the progressive approach continuing to use shared variables, while these would never appear in a re-invented application.

The two most common goals of transformation are a move to SOA and separation of user interface (UI) from business logic (BL), usually accompanied by a shift in the technology for the UI, e.g., to a web or other non-ABL client. A stepwise approach is easily applied to a move to SOA since all unmodified parts of the application continue to work as they did before and the changes are primarily "behind the scenes". When all functions in a given area have been converted to services, it becomes quite easy to provide a new UI for those functions because the desired separation of UI and BL has been achieved by moving the BL into services. But, this tends to defer the UI shift for a significant period while the conversion to services takes place and tends to only allow the conversion of specific clusters of functions rather than an application wide shift. This can meet the needs of some businesses, e.g., where the interest in a new UI is concentrated on a limited number of functions such as web access for customers for creating orders. But, it does not provide for an overall shift in the UI for an application.

If the goal of the transformation is dominated by the desire for a new UI, this is less readily achieved by a stepwise approach since converting individual functions tends to result in users require a mix of old and new UI required to complete their work, which can be confusing and awkward. At a minimum, one tends to have to select whole modules for conversion at each step. Since most legacy applications are monolithic in architecture, with UI, BL, and data access all mixed together, such a transformation can closely resemble a complete re-write, at least of the selected modules.

In terms of moving toward compliance with the layered structure of the OpenEdge® Reference Architecture, it is likely that one can replace direct access to the database with calls to new data access objects in a stepwise fashion, but separating the UI from the business logic has the same difficulties as it does in a transformation motivated by the UI alone.

Thus, we have two quite different strategies for moving from a legacy application to one of a more modern technology and architecture. Traversing the upper two legs of the triangle we begin with the extremely difficult task of extracting abstract model information from the concrete legacy application, which is currently not susceptible to much automation. However, once we arrive at the same kind of starting model which might have arisen from standard OOAD processes, then we can take advantage of MDA to help write the new application and the revised application can be as modern in technology and architecture as we like. Alternatively, traversing the lower leg of the triangle, we can have good success with gradual stepwise transformation if our primary goal is moving to a service-oriented architecture, but we are less likely to find an easy stepwise path to replacing the user interface.

There are some transformation projects which can be thought of as taking something of a middle ground. These are typified by projects in which one of the primary goals is replacing the user interface and the goal is to implement this change across the entire application. Some of these projects make some use of MDA to generate at least parts of the new application, but many are almost entirely manual. It is often the case that those following this path are disappointed by the amount of code which can be harvested, but if one considers the discussion about intrinsic framework above and the dramatic change in the fundamental architecture which is being undertaken, it should perhaps not be surprising that the amount of harvestable code is modest at best.

The productivity gains possible through the use of generator technologies such as MDA are dramatic. There is an up-front, one-time investment required to develop the framework and transforms, but once developed these can be applied to multiple projects with only small incremental investments to accommodate new requirements. The big obstacle to the efficient use of such technologies for transforming legacy applications is the limited toolset available for the automated extraction of abstract model information from the legacy code. If improvements can be made in that area, then it is almost certain that a full re-invention of the application can be accomplished by this process less expensively and more rapidly than through the sort of mass re-write process typical of UI replacement projects today. As an alternative, one can achieve significant restructuring of an application by the stepwise conversion to SOA approach as long as one is cautious or patient about replacing the user interface.