Migration from Structured to OO Methodologies

by Sally Shlaer and Stephen J. Mellor
Project Technology, Inc.
10940 Bigge Street
San Leandro, CA 94577-1123
(510) 567-0255
http://www.projtech.com

© Copyright 1996 by Project Technology, Inc. All rights reserved.
A version of this paper appeared in *Embedded Systems Programming* April 1996

1.0 Introduction

You've probably been using Structured Analysis and Design (SA/SD) for some years now and have been successful. There is a great deal of talk about object-oriented technology these days — especially about object-oriented analysis and design. You've heard the buzz words: "encapsulation," "inheritance" and "polymorphism" and understand the advantages they bring. You've also heard attractive things about "reuse" and "decreased time-to-market," but you've also heard some pundits describe this as hyperbole. Anyway, you know that these ideas are qualitative and hard to measure effectively.

Nonetheless, it's very clear that the software development industry is moving in this direction. You'd like to know what OOA/D implies, what's involved in migrating from SA/SD to OOA/D and if you can do it without betting both the farm and your career.

This paper describes Shlaer-Mellor object-oriented analysis and design [1, 2] by highlighting the differences between SA/SD and OOD. The Shlaer-Mellor Method was deliberately developed to take advantage of existing SA/SD tools and to make moving towards objects easier. We also describe an incremental migration path and make recommendations about how to get started.

2.0 Subject Matters

The first step in the method is to identify subject matters. In an intelligent instrument system, such as the control of a paper copier, there is an application subject matter that often has the same name as the system as a whole: Copier Control. This subject matter deals with issues that fundamentally have

© Copyright 1996 by Project Technology, Inc. All rights reserved.

to do with making copies, such as paper feed, camera synchronization and double-sided copy. There is a subject matter that controls the actual devices according to some interface protocol device control. This subject matter deals with actuators, switches, analog values such as motor speed, that just happen to control the paper grab actuator, camera control and paper-turn actuator. There is another subject matter, the User Interface, that describes which paper feed to use and whether to use single- or double-sided copy. And there are other subject matters that have to do with software architecture and organization: operating systems, languages and so on.

We can easily understand each subject matter independently of one another. We can figure out how a copier must work without knowing what protocol to use to control the actuators, or how the user tells the copier what he/she wants and which programming language is appropriate. However, some subject matters act as clients to other subject matters (servers), as shown in figure 1. In this figure, the user interface is the client of the operating system, but a server to the copier control application.

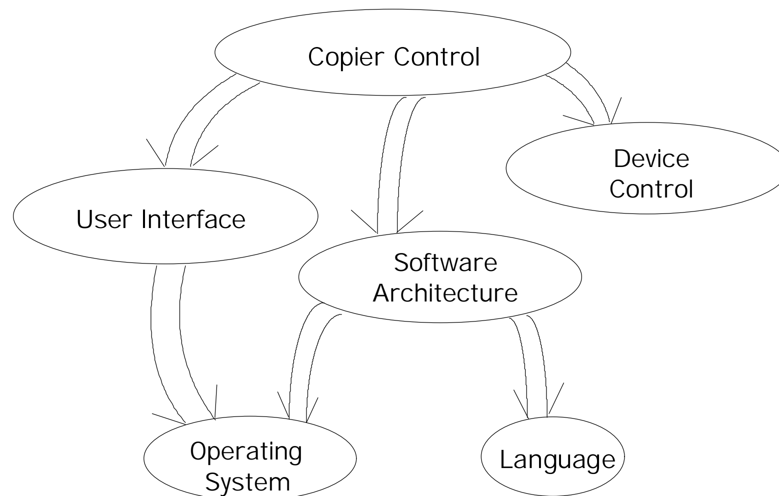

Figure 1: A Domain Chart for the Copier Control System that depicts the subject matters in the system.

Essentially, this first step defines the layers in the system as a whole: application at the top, which makes use of various interfaces, relying on some overarching software architecture, which in turn relies on the operating system and programming languages.

In SA/SD, there is no explicit way of showing the subject matters. Processes are processes and modules are modules. However, the layering concept is so powerful that many project teams simply show the various subject matters as processes at the highest level: a copier process, a user interface process and a device control interface. It can be difficult, though, to define precisely the flows between these processes because the point of view changes. From the point of view of the copier, a flow may correspond to the number of copies, but from the point of view of the user interface, we have button pushes from the number pad on the copier.

 © Copyright 1996 by Project Technology, Inc. All rights reserved.

Subject matters also alleviate two other common SA/SD questions:

- When does analysis turn into design?
- When do I stop decomposing these processes?

With a layered approach, each subject matter — even the operating system — is subject to analysis. We stop decomposing each subject matter when we have reached the primitive elements in that subject matter.

3.0 Subject Matter Analysis

In Shlaer-Mellor, the analysis of a subject matter comprises three closely-related parts:

- Conceptual entity identification
- Lifecycle analysis
- Functional analysis

We analyze only those subject matters that require it. For example, if we can buy the operating system, we would not analyze it; if we can reuse the device interface from another copier, then we would not analyze it. In this section, we describe the three steps involved in subject matter analysis.

3.1 Step 1: Conceptual Entity Identification

The first of the three analysis steps is to identify the conceptual entities or *objects*. If your organization uses database technology, you probably know how to do this already. You may call it Entity-Relationship (E-R) modeling, information modeling or data modeling. In Shlaer-Mellor, you do the same kind of modeling but with slightly different rules. We call the resulting model an *Object Information Model*.

An object defines the structure used to store the characteristics of each instance of an object. This is a similar concept to a table (object) having many rows (instances). (A terminology point: The C++ concepts of a class is a Shlaer-Mellor object, and the C++ concept of an object is a Shlaer-Mellor instance.)

There are two principles that may not appear in your current approach to information modeling:

- We require that you identify and describe all of the objects *in detail* because the objects you select form the foundation for the remaining steps.

- We require that the same rules, policies and characteristics must *absolutely uniformly* apply to every instance of an object.

We require these rules, in addition to the principles you are using for information modeling, because the completed analysis is a model of the conceptual entities and their behavior, not just a model of stored data.

© Copyright 1996 by Project Technology, Inc. All rights reserved.

3.2 Step 2: Lifecycle Analysis

The next step in the method is to define the lifecycles of each of the objects. We think of each object as a typical but unspecified instance. When you model the behavior of one instance, that behavior applies uniformly to all instances of that object. For example, if the copier has two paper feeds that you abstracted as the Paper Feed object, the behavior of each feed must be the same or you cannot abstract it as a legal object. To model the behavior of an object, we use a *state model.* Each instance of the state model is called a *state machine*. Each state machine operates concurrently and asynchronously with respect to all other state machines so that the state machine for one paper feed may be in one state, while the state machine for the other instance is in another state.

If you have used state models in the past, the model we use will be very familiar. As usual, we define the state model in terms of states, events, transitions and actions. There is a graphic rendition of the model called a state transition diagram and a tabular form called a state transition table, which you may know as Hatley/Pirbhai's State Event Matrix. In Shlaer-Mellor, an action is associated with the state, not with the transition, as it is in both Ward/Mellor and Hatley/Pirbhai [3 and 4].

Ward/Mellor and Hatley/Pirbhai are regarded as real-time system development techniques, and it is typical to associate the use of state models with real-time systems. However, you can use state models to model the behavior of anything. Consider, for example, a customer of a telephone company who is expected to pay a monthly fee for services, and perhaps a deposit equal to three months' service charge. The customer is expected to pay the bill in full each month. After one year of paying on time, the telephone company refunds half the deposit. If the customer moves away, he ceases to be a customer and the deposit is refunded. The states of the Customer object are: Becoming a Customer, On Probation, In Good Standing and Ending Service. In this example, it may take twenty years to progress through all the states, rather than twenty milliseconds. A state model is still a valid tool to model the behavior of the object.

If you have not previously used state machines, the modeling techniques are quite easy to learn as long as you maintain a selfish perspective: "I am the Paper Feed, what happens to me?" or "I am the Customer, what do I want to happen to me now?" If you maintain this view, it is easy to build the state models. If you fall into the trap of thinking about the objects from the system's point of view, you may find yourself with a complicated state model that reflects all possible transactions on the Customer data.

When you send an event from a state machine to another, you must say to which instance the event is going. For example, if you wish to send an event to the Paper Feed object to make it stop, you must say which paper feed you want to respond to the event. We do this in the method by directing the event as follows: Generate PF1: Stop Feed Now (paper feed id), where the paper feed id is the identifier of the instance to which we are sending the event. In this manner, we can synchronize the behavior of each specific state machine.

© Copyright 1996 by Project Technology, Inc. All rights reserved.

Each instance of each object operates concurrently with respect to all other instances. Because you control each instance explicitly, you can develop a fully concurrent model that enables you to simulate the behavior of the system, instance by instance. This is critical for validation and verification during systems analysis, long before you deliver any code. In turn, validation and verification enable the discovery of errors earlier in the systems development lifecycle when you have a chance to do something about it, and more cheaply than when analysis misconceptions are embedded in code.

3.3 Step 3: Functional Analysis

In the third analysis step, you identify the processes required in the application. This is achieved by drawing data flow diagrams (DFDs) to explain the details of the processing required for each action. Because we use the data flow diagram to model the action of a state, we call them *action data flow diagrams* or ADFDs.

While much of this step of the analysis is familiar — the notation is similar to that used in structured analysis — there are significant differences on how the ADFDs are constructed:

- The objects identified in step one of the analysis become the data stores on the action data flow diagram.
- A separate flat action data flow diagram is produced for each state of each object's lifecycle. There are no more hierarchies of bubbles to level and balance: Every action is entirely self-contained.
- The processes shown on the action data flow diagrams are the fundamental leaf processes required in the system.
- Each process on an action data flow diagram is assigned to the object whose data it accesses.

For example, a process that accesses the Paper Feed object's data is assigned to the Paper Feed object — even if the process appears on an action data flow diagram for a state of the Transport object.

Notice that these ADFDs represent a completely new way of organizing and thinking about processes. In this approach, a process may appear in many actions, in actions of the Paper Feed object and the Transport object. Yet we think of it as just one process and provide just one specification for it, allowing for the definition of significant reuse of processes. This is not the traditional hierarchical way of organizing functions. Nonetheless, the models depend on concepts and notations that are very familiar, so a significant amount of your previous experience still applies.

© Copyright 1996 by Project Technology, Inc. All rights reserved.

3.4 Subject Matter Analysis Framework

The overall organizational framework — a state model for each object and an action data flow diagram for each state — is shown in figure 2. The three models, taken together in this integrated manner, provide a complete and executable model of a subject matter. We now turn to design.

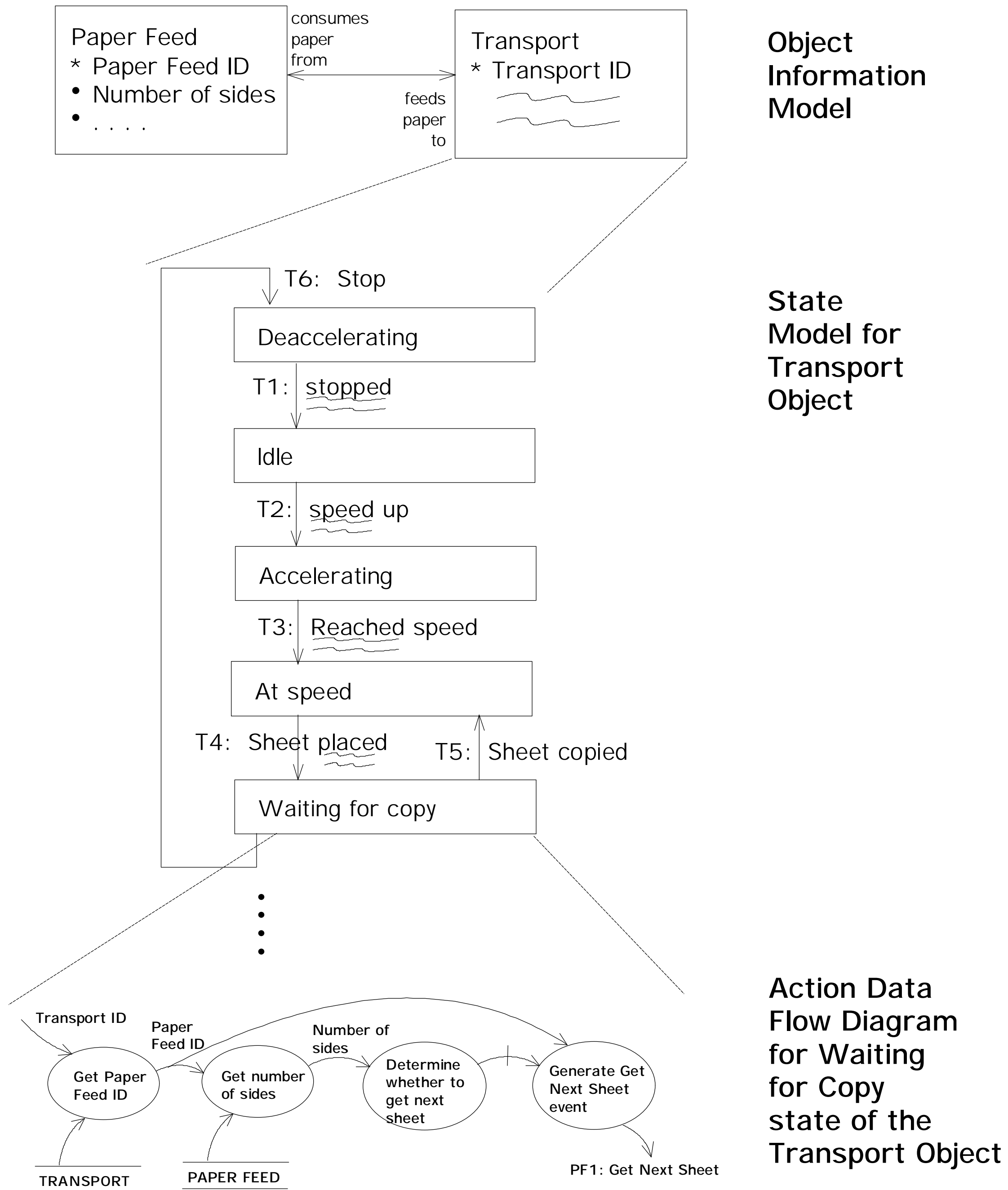

Figure 2: The three models of OOA

© Copyright 1996 by Project Technology, Inc. All rights reserved.

4.0 Design Methods

In Structured Design, you take the models constructed during analysis and transform these models into structure charts for each program. This approach has been vigorously criticized in the object-oriented community on the grounds that the notations are different (data flow diagrams vs. structure charts), and because the transition between the two phases is not smooth.

Many object-oriented methods address these issues by employing the same base notation for object-oriented analysis and object-oriented design, with additional notation for processors and tasks. The interpretation of an "object" in analysis and design is the same, avoiding discrepancies in notation. You create the design models by reorganization of the analysis models into processors and tasks, and by adding "design" information such as parameter types, synchronization types and the like. We call this type of design *elaboration* because the designer elaborates the analysis models to produce the design models.

Shlaer-Mellor believes that it is important to distinguish analysis from design for software development process and project management reasons. We also note that the designer both reorganizes the analysis models and adds design information — in effect, building another model. How can we reconcile these ideas with the object-oriented critique of SA/SD?

Shlaer-Mellor's answer is to recall the notion of separation of subject matters, and to build a separate set of design models, leaving the analysis models alone when analysis is complete. We propose two approaches to design in Shlaer-Mellor. The first approach is an elaboration approach that uses the classical techniques of copying the analysis into the design. We illustrate this approach by using an object-oriented design and a non-object-oriented design. The second technique is based on translation. However, in the translation approach, the design models *contain no information about the analysis*.

5.0 Two Elaborative Designs

This section describes two elaborate design approaches: one using an object-oriented design and the other using a non-object-oriented design.

5.1 An Object-Oriented Design

For an object-oriented design, the package of design models shows the definition of each class in the design, the inheritance hierarchy of the classes and the internal design of each class. We describe how to build an object-oriented design from an OOA.

5.1.1 Class definition

To define the application classes, make a class for each object that was identified during the analysis. The attributes of the object (as shown on the information model) become the instance data of the corresponding class. The methods of the class are defined by collecting all the data-accessing processes assigned to the object on the ADFDs — each one becomes a method. And we define a method

© Copyright 1996 by Project Technology, Inc. All rights reserved.

for each event that can affect the object; these methods cause the object to move through the states of its lifecycle as required by the analysis.

5.1.2 Inheritance hierarchy

The inheritance hierarchy for the application classes is then derived from the “is-a” relationships on the information model of the application. In addition, the application classes are made to inherit from container classes of a standard class library as required to implement the desired internal data structures. This step (like class definition) is probably quite new and unfamiliar, but it is an orderly one and not very difficult.

5.1.3 Internal design

Finally, to develop the internal design of your class, build a Class Structure Chart. The diagram and concepts are similar in nature to a structure chart in structured design but with these differences:

- There is one structure chart for each class (instead of one for each program).
- The class structure chart has as many “boss” modules as there are methods in the class, instead of a single boss module for the entire program.
- Data types are specified for each of the couples.

This final step in the design is an easy one for people experienced in structured design since it draws on concepts and notations that are well understood.

5.2 A Non-Object-Oriented Design

What if you’re not using an object-oriented language? Or you want to integrate a new function with some legacy code? For example, consider the implementation for the Copier Control System. It is to be written in C, has multiple tasks and must use existing code for the device control. In addition, let’s stipulate that the Transport, Paper Feed and all other objects that access the hardware are in one task, and that all other objects, including those that interact with the user, exist in another task. This system partitioning requires no data access between the two tasks, though the tasks must still synchronize their behavior.

5.2.1 Task Definition

To define each task, select all the objects from the analysis that belong in the task under construction. For the “hardware access” task, this includes the Paper Feed and Transport objects. This task must also compile (or link) the existing device control code. Each object in the application analysis becomes a struct. We define a set of functions (in the same file) that access this struct by finding all the data accessing processes assigned to that object on the ADFDs. We define another function for each event that the object can receive and a function for each action on the State Model.

5.2.2 Function Definition

When we define the functions for the actions, we write code for each process on the ADFD for that function. These are quite straightforward except for functions that send events and communicate with the device control code.

 © Copyright 1996 by Project Technology, Inc. All rights reserved.

There are two cases when sending an event: an event-send to an object assigned to the same task and an event-send to an object in another task. In this system, the allocation of objects to tasks is static, so we can simply code the two cases differently. For events in the same task, we simply invoke the event-receiving function for the destination object. For a different task, we package up an event inside an operating system message. (If this approach makes you nervous, you could instead write a single interface and look up the destination of the event at run time.)

For functions that access the devices, simply write code that invokes the existing functions in the legacy code. If necessary, build a layer of function calls (or macros) to make this job easier.

5.3 Notational Issues

We did not describe any notation for representing either of these two designs since the needs of the two designs are different. In the object-oriented design, you must have a way to denote inheritance, but in the non-object-oriented design you don't need one. In the object-oriented design, you use the structure chart for the internal design of each class. For the non-object-oriented design, you would use the structure chart to show the design of each task.

The point is that any given design style requires its own notation. There is no need for a notation to represent generics if your design (often determined by your programming language) doesn't have the concept. Similarly, you don't need a way to denote exception handling, templates, types or whatever unless your design incorporates the concept. A design notation containing all possible concepts is impossible to remember and harder to use. A design notation that doesn't contain all possible concepts won't work for certain types of design. We defined a very simple notation for a simple set of object-oriented designs called OODLE: object-oriented design language (the E is silent). If this works for you, fine. If you wish to use the notations of SD, that will work too. But there is a better way.

6.0 Design by Translation

Many Shlaer-Mellor projects have built systems using an elaborative approach such as that followed by one of the examples previously described. In both these elaborative examples, we build a design model that *incorporates* the analysis model. We also described the design by telling you how to build the design model by copying and changing the analysis models in very specific ways.

To produce the design models this way, we don't need to do much work after figuring out what the system looks like. If we decide to use C++, this necessarily implies classes. If we decide on C, this means structs and functions. If we decide on two tasks and an allocation scheme (device-driving objects in one task, the rest in another), then this determines the message traffic between the two tasks for a given set of scenarios.

We can reason about the design by thinking explicitly about the strategy, and we don't need to know any details about the subject matter of the application. For example, the non-object-oriented design relies on the absence of data accesses between the two tasks. We only need to study the application to ensure that the data accesses are sufficiently localized to take advantage of the proposed scheme.

© Copyright 1996 by Project Technology, Inc. All rights reserved.

To build a system using design-by-translation, we first determine the organization of the software: C or C++, one task or many and how to handle events, etc. We call this subject matter (the organization of the software), the *software architecture.*

We then define the translation rules: object to struct, event to event-receiving function invocation, event to message send to another task and so on. This is called the *bridge* and it corresponds to the client-server relationships among the subject matters. Note that we state some bridges in terms of interfaces to existing packages. For example, the device control package is an implementation of a domain. We state the bridge between the application and this domain by stating how to invoke already-implemented functions.

Finally, we allocate the elements of the analysis model to the elements in the software. We call this *coloring* because we visualize the process as taking several highlighter pens and coloring some elements of the OOA with one color, and other parts with another color. We allocate all elements of one color to one task, and all elements of the other color to another. In the previous example, we chose to determine the allocation first for ease of explication. However, note that once you have the software and the translation rules for a multitasking system, you can select a different allocation scheme simply by choosing to color the objects differently.

What notation do we use to understand the software architecture? Because the software is a subject matter like any other, we can use OOA.

A moment's reflection shows that the size of the design work is no longer proportional to the size of the application analysis: a very significant benefit even in small projects.

7.0 An Incremental Migration Path

You can bite off one piece at a time. But to get the benefits, you have to swallow each piece!

Consider separate subject matters, for instance. This notion is very simple and intuitive. You've surely used the concept in your previous systems. You could even use the domain chart (see figure 1) tomorrow in the context of SA/SD. That then requires you to build separate analysis models and data dictionaries for each domain that must be analyzed.

You can use an Object Information Model to help identify the conceptual entities in your problem. This will work best if you have a domain chart and you keep your analysis separate, or if you have a single domain to analyze. If you do not partition your system into domains, you may have a problem drawing relationships. Our favorite example of this problem is a system that required certain critical events to be recorded along with the time and date. The analysts built a Logging object, and then discovered that 50% of their objects needed to participate in a relationship HAS CRITICAL EVENTS LOGGED BY! Of course, the Logging object belonged in another domain.

You can use state modeling to model the dynamics of objects on the object information model. Please don't try to use state models with a weak object information model or one that does not

© Copyright 1996 by Project Technology, Inc. All rights reserved.

conform to our rules. If you do, you will find that your state models are incredibly complex and messy, and may find yourself wishing for hierarchies of states. This indicates that your Object Information Model contains incorrectly abstracted objects.

Build an ADFD for each state of each object's lifecycle. Any other organization would be closer to a variation on structured analysis. In practice, some projects find process modeling with ADFDs onerous and instead choose to use an action description language as provided by the CASE tool, or even to use a programming language, such as C. All these approaches work. But using a programming language, such as C, is dangerous. The problem is that a programming language is very powerful and flexible — so it is easy to flout the rules of OOA and to start coding, making some assumptions about what the design will look like. If you later choose to change the design to use different assumptions, then it is difficult to change the programming language to generate different code (that would mean changing the compiler). If you used an action description language designed for the method, then the logic can be translated into many different forms and languages.

We proposed two forms of design, one elaborative and one based on translation. Elaboration is technically closer to what you do today with SA/SD. It is also closer in project management terms because many project management paradigms assume that the sizes of the downstream phases are roughly proportional to the size of the application. If your project is pressed for time, you might choose to think of the detailed application analysis work as "design," reducing any management conflict induced by a front-loaded process.

Formal translation is the better choice. However, you must have a complete and detailed analysis for formal translation to work. For maximum effectiveness, invest in complete tooling, including translation technology.

8.0 Getting Started

To get started, charter a pilot project that is small-to-medium in size and not critical to your organization. Put your best people on the project with the condition that each person is committed to trying something different, and is capable of becoming a mentor once the pilot project is complete. Assign a separate person to measure progress, making sure to compare the cost of doing something for the first time versus subsequent efforts. Make clear to the pilot project personnel that their job is to get the project done, not to evaluate or modify the approach — that's the job of the metrics team. The reasoning here is simple: as soon as a project's personnel get the idea that the project exists to evaluate a new approach, then they will evaluate the new approach. After all, that's the purpose of the project. The all too common result is that the project personnel spend time debating the fine points of the approach and do not get the work done. By all means include project personnel in the post mortem.

The approach described in this paper is fundamentally different from SA/SD even though every effort was made to minimize unnecessary differences. To think in the new way, you will certainly need professional training. It is important to train the entire team, not send one or two people to training and then use them to teach the rest of the team. The time spent learning, as opposed to working on the project, is the real cost, and you'll have to spend that one way or another with each

© Copyright 1996 by Project Technology, Inc. All rights reserved.

individual. In addition, your people need and deserve the information first hand, along with first-hand experience.

It is critical that you have a mentor for the technical work and the project manager. The mentor should have seen the entire process (or at least those parts you have chosen to use), and may come from another part of your organization or from outside. Remember that all object-oriented methods are not equal, and that an object-oriented programmer is not necessarily a methods or project management expert.

References

[1] Sally Shlaer and Stephen J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice Hall, Englewood Cliffs, NJ 1988.

[2] Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice Hall, Englewood Cliffs, NJ 1991.

[3] Paul T. Ward and Stephen J. Mellor, *Structured Development for Real-Time Systems*, Prentice Hall, Englewood Cliffs, NJ 1985 (three volumes)

[4] Derek Hatley and Imtiaz Pirbhai, *Strategies for Real-Time System Development*, Dorset House, New York, New York. 1988.

© Copyright 1996 by Project Technology, Inc. All rights reserved.

Sally Shlaer and Stephen J. Mellor

Sally Shlaer and Stephen J. Mellor are the co-founders of Project Technology, Inc. They created the Shlaer-Mellor Method in 1979 while consulting on a large rapid transit system. Together they have written two books on software development: *Object-Oriented Analysis: Modeling the World in Data*, and *Object Lifecycles: Modeling the World in States*, both of which are available from Prentice Hall. They are currently working on a third book in the series entitled *The Shlaer-Mellor Method: Recursive Design.*

Project Technology, Inc.

Project Technology's mission is to improve the productivity of real-time software development. As developers of the Shlaer-Mellor Method for software analysis and design and the BridgePoint automation tool suite, the company has been providing training, consulting, tooling and hands-on implementation services for over 10 years. Applications developed using the Shlaer-Mellor Method can be found in defense, telecommunications, financial services, real-time control, instrumentation, manufacturing, transportation and utilities sectors. Customers include AT&T, BNR, Abbott Laboratories, Delco, EDS, General Electric, IBM, Lockheed-Martin, Motorola and Westinghouse. Project Technology has offices in several U.S. locations, including Texas, Arizona and California. They also have a U.K. office in Glasgow, Scotland and distributors worldwide.

To learn more about the Shlaer-Mellor Method or Project Technology, please visit their web site: **http://www.projtech.com**

© Copyright 1996 by Project Technology, Inc. All rights reserved.