Shlaer-Mellor Object-Oriented Development: A Manager's Practical Guide

Michael M. Lee
January 31, 1996

Project Technology, Inc.
10940 Bigge Street
San Leandro, California 94577-1123
510 567-0255
http://www.projtech.com

Table of Contents

1. Introduction

Theory and Practice

An intellectual understanding of downhill skiing techniques, no matter how thorough, is not sufficient to ensure the success of a new skier as they peer down the side of a mountain for the first time. Theory is not enough. The new skier must *experience* the phenomena of hurtling down the mountain's side and *practice* applying the theory before success can be achieved. The same is true of mastering a new software development method. While an intellectual understanding of the method is a necessary first step, only through experience and practice will an individual, a team, and an organization develop the capability to apply a new method successfully. This article describes experiences gained introducing the Shlaer-Mellor Method of object-oriented analysis (OOA) and Recursive Design (RD) to a variety of projects and presents an approach, evolved over time, for succeeding with this task.

Background

The observations and recommendations in this paper are the result of first hand experiences over the past 12 years on several (10+) projects using Shlaer-Mellor (S-M) for the first time. They are also the result of many long and spirited discussions with colleagues too numerous to mention, though without whom, this synthesis would not have been possible.

The projects involved medium to large, real-time systems including applications in manufacturing, communications, medical instrumentation, and computer peripherals. The organizations typically had little or no prior experience with formal software engineering methods. The motivations for using a more formal, object-oriented approach included efforts to:

- reuse all or significant parts of the system,
- manage the size and complexity of the system,
- develop a long-term, product platform, and
- improve the software development process.

The projects had varying degrees of success, and in one case there was a failure. The amount I learned from each project seemed to be in direct proportion to the amount of difficulty encountered. Almost without exception, the difficulties were not technical, (i.e., did not relate to the size or complexity of the problem, the underlying implementation technology, or the method) and they generally fell into one of the following categories:

- inability to gain project-wide commitment to change,
- no experience with, or an inability to establish, an engineering process, and
- poor management of the engineering process.

Committing to and managing change is one of the more difficult tasks one can undertake. When successful, however, it can have stunning results, so let's proceed.

Shlaer-Mellor Object-Oriented Development

Shlaer-Mellor Object-Oriented Analysis uses an integrated set of models to identify the objects in a system, their dynamic behavior, and the required processing. The motivations for using this approach to analysis include:

- explicit attention to real-time issues,
- capability for scaling to large and small projects,
- ability to verify the analysis by simulating the OOA models,
- well-defined transition between analysis and design, and
- availability of textbooks, training and technical support.

Shlaer-Mellor Recursive Design is based on a translation of the OOA models, as specified by the *software architecture*, that can be automated and can produce 100% of the code required for the capabilities expressed in the models. The motivations for using this approach to design and implementation include:

- independent reuse of both application OOA models and software architectures,
- reduced time to market provides by concurrently working on application and software architecture,
- reduced time to market provided by the automated translation, and
- reduced costs provided by the automated translation.

For a more thorough understanding of the Shlaer-Mellor Method, the reader is encouraged to refer to the books on this topic [1,2]. A diagram of the major process steps of the method is shown in Figure 1.

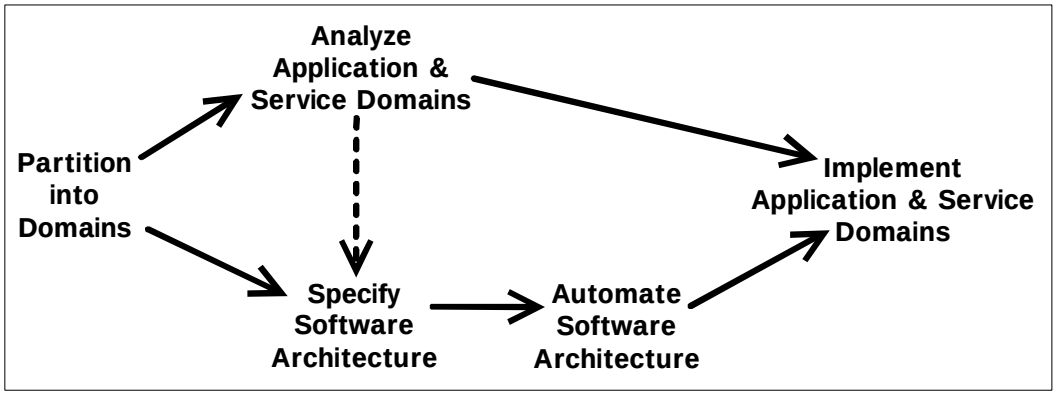

Figure 1: Activities of the Shlaer-Mellor Method

2. The Real World

Practice, as opposed to theory, is typically neither pretty nor clean. In this section, those aspects of projects which *are* sometimes ugly and dirty are considered. The issue here is not to point a self-righteous finger at these areas, but to recognize that they do exist and will have a negative impact on the project if not effectively addressed.

Requirements

The problems that arise when requirements for a software system are poorly understood are widely recognized and well documented in a number of books and articles [4, 5]. Still, the problems persist, and, in my estimation, will never totally disappear. Here are a few reasons.

- *Competitive Pressures*. It is a fact of life that competitive pressures constantly push for more functionality in less time. Under such pressures, product requirements are frequently less than complete for various reasons. Competitive pressure also cause requirements to change with time.

- *Pushing the Technology Envelope*. When systems are expanding into areas of new technology, the "requirements" are frequently being discovered as development proceeds.

- *Building Product Platforms.* When it is the objective of a project to develop a platform to support a family of products over an extended period of time, seldom are all the members of this family defined yet. Many are little more than a glimmer in some marketing VP's eye.

In situations like this, requirements are never well understood at the beginning, nor do they remain static over time. What is needed in an analysis method to manage this is the ability for the engineer to *quantify* the tradeoffs between different or changing sets of requirements. What is needed in the design and implementation method is the ability to isolate application (read: feature) changes from design and implementation (read: software architecture) changes.

Schedules

Producing relevant schedules for software development is at least as problematic as doing the development itself. This is especially true for the initial analysis phase when we don't know what we don't know yet. Add to that a learning curve of uncertain slope for a new software engineering method and it truly becomes a Herculean task. No method, it would seem, can offer the clairvoyant assistance needed here. What we can ask of a good method, however, is that it provide relevant feedback for adjusting and calibrating the plan against the schedule as well as include techniques for actively reducing development time wherever possible through automation of the development steps.

Size and Complexity

Of issue here is not that systems are large or complex, but that their true size and complexity are often poorly understood. As long as this is the case, schedule projections suffer and a mild sense of being out of control persists. It is in the analysis phase that the size/complexity question must be answered, and the analysis method should have two important characteristics to deal with this. First is the ability to scale up for large and complex problems if necessary. Second is the ability to factor and reduce the size and complexity to their smallest possible dimensions.

Engineering Skills

Successful software engineering requires a number of very different technical skills. These include the ability to do analysis, system design, program design, coding, integration, and testing. Very few individuals are equally talented in all of these areas. My informal sampling of software engineers produces a distribution that peaks in the middle with program design and coding, and falls off in both directions with good analysts and testers being the most rare. This creates two challenges during the analysis phase. One, good analysts must be identified and assigned judiciously. Two, productive work must be found for members with other skills. A third challenge is created during the (concurrent) specification and construction of the software architecture phase, and it is to identify designers with good computer science, as opposed to application, expertise.

Silver Bullet Syndrome

Perhaps the most insidious real-world problem when introducing new software methods is the over zealous and unrealistic expectation that the next new method (object-oriented anything, these days) will prove to be the long awaited "silver bullet" with which the dreaded software development dragon can be slain. It is insidious because it carries enthusiasm that is short lived and a commitment that is only slightly longer lived. Addressing this problem is outside the scope of *any* particular method, though it must be done if a method is to be assessed on its true merits. The best policy here is complete frankness on the benefits and liabilities of a method, and an ability to adjust for these.

3. Practical Approach to Applying the Method

Halfway through one particularly challenging project, when some of the difficulties began to feel familiar, an attempt was made to identify and categorize these difficulties [3]. The strongest pattern that arose was that of time and sequence. Different problems came up at different times in the project, and different problems were dealt with more or less effectively depending on the sequence in which they were resolved. For example, it was important to gain both management and technical support for using this new development method *before* getting too far down the path of applying it on a project. Otherwise, one was constantly fighting the battle of "Do we really want to do this?" in the middle of attempting to do it!

What evolved from this was a set of strategies (we'll call them "key success factors") for addressing each problem and an identification of when in the project life cycle it's important to apply these strategies. These key success factors are shown in the following diagram relative to the phase in which they should be applied. The following chapters address the issues and proposed approach for each of these strategies.

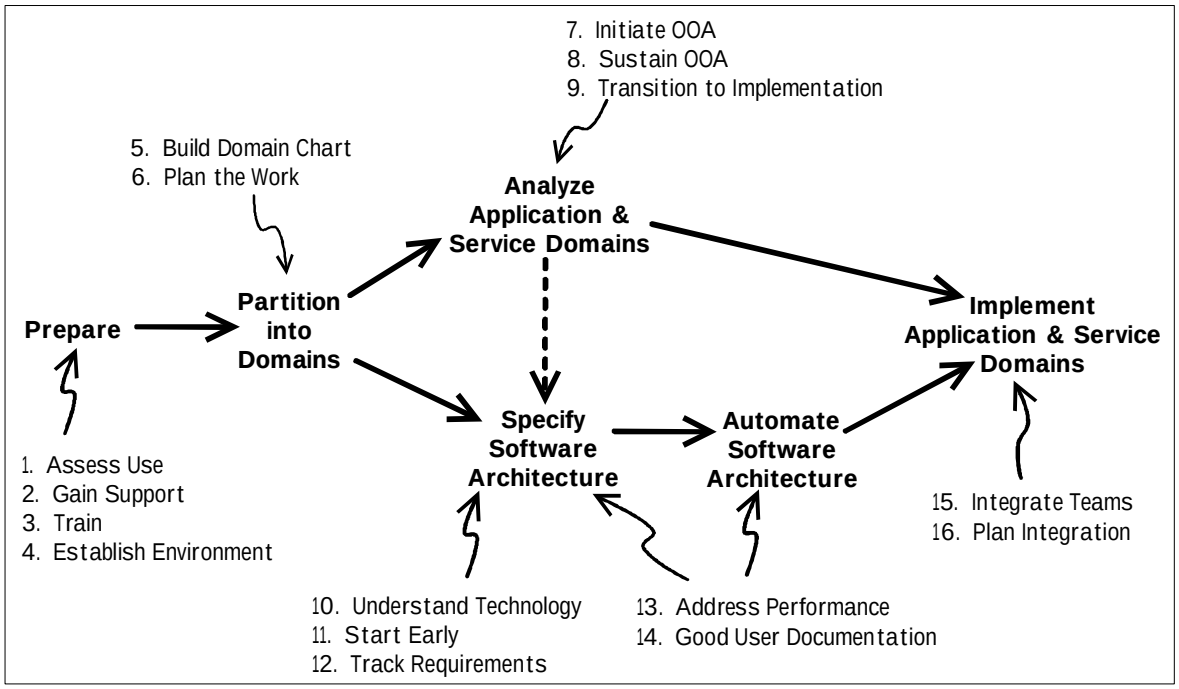

Figure 2: Key Success Factors in Shlaer-Mellor Process

4. Preparation Activity

Our main objectives in this activity are to gain support for the method and get the software team prepared to apply the method on their project.

Key Success Factor 1: Assess Applicability of Shlaer-Mellor

As basic as this step may seem, it is seldom done in as thorough a manner as it should be before the other steps are started. This can cause difficulties in the later steps when it continues to be assessed, multiple times, by different groups, possibly with different results! The basic assessment which must be made is the *match between the projects objectives and the method's capabilities.* There are three basic areas which need to be considered in making this assessment

- *Technical Considerations.* Is S-M the right tool for the job? Does it address the difficult issues which the project expects to encounter? Does it help meet the project's technical objectives?

- *Management Considerations.* Can the project lifecycle be "front loaded"? Will there be support for training? Is there a perceived need for this technology? Does it help meet the project's business objectives?

- *Staff Considerations.* Is there an existing analysis method? Does it work? Is there a perceived need for this technology? Does it help the staff meet their professional objectives?

When assessing these issues, one needs to be aware of project objectives that may and may not be addressed by S-M. Examples of project characteristics that can be addressed extremely well by using S-M include analyzing large and/or complex systems; developing configurable, data-driven systems; preparing for object-oriented design and programming; designing for reuse and maintenance; and automating the implementation.

Examples of project characteristics that may not be addressed particularly well by using S-M include the "n-th iteration" on a well understood problem; a project which is already absorbing a number of other new and different technologies; projects with extremely aggressive schedules; and systems where "quick and dirty" is an acceptable approach. Meilir Page-Jones has recently compiled an interesting list of motivations for taking, and not taking, an object-oriented approach which expands on this topic [7].

One of the best ways to assess applicability is to use S-M on a pilot project. This allows the development team to gain first-hand experience and substantiate their assessment with real-world results. In one project, for example, a pilot effort not only resulted in a positive assessment of the method, but also gained significant insights into critical performance issues. These results made the next step, gaining management support, much easier.

Key Success Factor 2: Gain Management Support

Having determined that there is a good match between the project's objectives and the capabilities of S-M, the next step is to gain management support for use of this new analysis method. The support that will be needed will include funding for such things as training and CASE tools, commitment to a schedule that will spend more time in analysis and design and correspondingly less in implementation, organizational changes to accommodate the different skill profiles required for this work, and cultural changes in the way work gets documented and reviewed.

It's important to gain this support based on *realistic expectations* of how the work will proceed. It will do more harm than good to gain support based on unrealistic expectations like : "it's object-oriented, so we'll see a 10 fold increase in productivity immediately", or "the learning curve impact is just the training time". Such support is very likely to dissipate quickly as these unrealistic expectations are not met, and the project will suffer accordingly.

At the heart of gaining management support is convincing the appropriate managers that there is a good match between the project's objectives and S-M's capabilities. The following tasks are key for doing this.

- *Present Assessment of S-M's Applicability.* Having established that there is a good match between the project's objectives and S-M's capabilities, this must be communicated to the management team. They should have an opportunity to probe the assessment, have their questions answered and hopefully come to the same conclusion.

- *Demonstrate Benefits of Good Match.* It's important at this point to recognize that you are involved in making a *sale*. The best way to succeed with this is to go beyond just establishing the good match and elaborate on other benefits to be gained on future projects when S-M, and hopefully a significant amounts of code, can be reused.

- *Establish Realistic Expectations.* It's extremely important at this point, for reasons cited earlier, to ensure that the above benefits are presented in a realistic light, and that the costs are acknowledged. Typical costs are training, learning curve, CASE tools, and possibly technical support. Typical changes in "business as usual" are some organizational shuffling, new documentation and review procedures, and more time spent at the front end of the project.

- *Present a Plan.* A good management team will not commit to a process without at least a preliminary plan of execution. Be prepared to provide this. Base it on the prescribed approach of constructing a Domain Chart, Subsystem Diagrams, and Project Matrix [1].

Key Success Factor 3: Train Staff in the Methodology

A series of courses exist for training in Shlaer-Mellor Object-Oriented Analysis/Recursive Design. The best thing to do is train the team as a group in OOA immediately before starting the analysis. After some familiarity has been gained with OOA, again train the team as a group in RD. Send both technical staff and their managers: the technical staff for obvious reasons, the managers so they understand the process they'll be attempting to schedule and control. An important point to understand about the training is what it can and cannot accomplish. One, it will not create analysts out of people without these skills. Two, it will only impart knowledge, not experience. Three, it's folly not to do it.

Key Success Factor 4: Establish Productive Environment

The software profession has made great strides over the past few decades in improving the implementation environment. The programmers' workbench, coding standards, configuration management systems, and interactive debugging environments are a few good examples. By comparison, analysis environments are usually primitive. Though the tools exist, it seems that this phase receives little attention. However, if you plan to spend a significant part of your schedule in this phase, it behooves you to give it more consideration. The essentials for a productive environment include the following.

- *Document Library*. Establish a central, public repository for project documentation. This is *the* deliverable during the analysis phase. It's important to manage it and ensure its common and convenient accessibility. This is most conveniently done "on line", though a manual system can also work.

- *Document Standards.* It's a small thing, keep it simple, but do it! Designate a document and drawing tool. Ensure that title, author, version, and date information are uniformly included on every document.

- *Meeting Guidelines.* Unproductive meetings place unnecessarily large overheads on the analysis process. For productive meetings, clarify objectives, limit allocated time, and stick to agendas. Distinguish between the following three types of meetings. A working meeting for getting technical work done. A presentation meeting for informal communication of the results of a piece of work. A review meeting for formally reviewing a work product.

- *CASE Tool.* OOA generates a set of integrated models which all have graphic representations. RD translates the information associated with these models into code. This is not something that can productively be done manually or with a simple drawing tool. What is needed is a CASE tool which fully supports the formal nature of OOA and provides thorough support for RD.

5. Domain Partitioning Activity

Our main objectives in this activity are to create a technical framework for the software engineering activities on the project based on the domain chart, and construct a project plan based on that framework.

Key Success Factor 5: Build a Good First Domain Chart

The first step in the Shlaer-Mellor Method is to build a domain chart. The domain chart represents a system wide view of all the material that must be specified and built to realize the end product. It is organized into *domains* that represent distinct subject matters that can be analyzed and implemented separately. This system wide partitioning based on subject matter is one of the main strengths of the Shlaer-Mellor Method because it provides a complete view of the whole system, it forms the framework for all the technical work, it allows different parts of the system to be worked on independently and still be successfully integrated, and it provides for reuse at a much higher level than single objects.

Given these characteristics, the domain chart is a high leverage factor on the success of the rest of the project work. Get it right and things go smoothly, get it wrong and things are forever snarled. Thus, there is a big return on time invested in building as good a domain chart as possible when you start your project. The following considerations help you succeed at this.

- *Partition on Subject Matter*. Most engineers are more familiar with partitioning software on criteria other than subject matter. You need to watch out for this, and keep the effort focused on the right partitioning criteria: a complete and coherent body of knowledge about a particular subject matter that can exist independently of other subject matters. Popular, but incorrect, criteria to avoid include:
    - functional partitioning, identifying the functions not the knowledge base,
    - task partitioning, jumping forward to implementation packaging, and
    - hardware partitioning, allocating according to the hardware design.
- *Use the "Object Blitz"*. This is an informal technique taught in the training courses that helps to get started and overcome the inhibitions that a blank sheet of paper (the yet-to-be-built domain chart) frequently raises. It should be utilized to test the many theoretical arguments that will be raised in considering different partitioning.
- *Engineer the Solution*. There are sometimes trade offs to be made when constructing a domain chart -- there can be more than one way to conceptually layer something. Consider these trade offs from an engineering, effort/benefit perspective and not some theoretical, abstract, or worse still, methodologically "religious" perspective.

- *Stop When Additional Effort Produces Diminishing Returns*. As important as the domain chart is, you need to recognize that one seldom gets it perfect on the first pass. As the team proceeds with the analysis you learn more and sometimes will want to alter the domain chart to take advantage of these new insights. Thus it is important to recognize when you're attempting to answer questions that will only be answered by doing some real analysis work. At this point, stop working on the domain chart and get on with the analysis. This is most easily recognized by iterations on the domain chart that consume significant time and produce trivial or circular changes.

- *Recognize the Need to Maintain the Domain Chart*. Do not consider your first effort at the domain chart etched in stone. As pointed out in the previous item, it is frequently to your advantage to update the domain chart as you understand more about your system. Remain mentally supple enough to take advantage of these changes when they present themselves.

Key Success Factor 6: Create an Executable Plan

The plan created in the previous phase of gaining management support was really a strategic statement outlining a general approach. Now a tactical statement is needed to clarify the week-to-week activities and focus the effort. When planning this work, especially consider what you want to achieve with your initial analysis efforts. The best guidance that can be offered is to initially target those areas of the application which are expected to provide additional insights. This may mean different things on different projects.

- *Well Understood Area*. Sometimes doing a well-understood area to gain experience with OOA method helps to build confidence in the method.

- *Difficult Area*. Sometimes doing a particularly difficult area to demonstrate the capability of the method helps. This works best when you have experienced analysts.

- *Natural Starting Point*. Sometimes the structure of the problem suggests a natural starting point based on dependencies on this area. Focus your efforts there first.

The important point is to target your efforts for a *defined objective* as a means of providing a focus for your activities. Taking an undirected, breadth first approach to the problem is very likely to stall out without producing anything of significance.

6. Application Analysis Activity

Our main objective in this activity is to successfully develop Shlaer-Mellor OOA models for all the capabilities required of the software for the end product.

Key Success Factor 7: Get Analysis Work Moving

Two things are important as you begin the analysis activities. One is organizing the analysis teams, and the other is recognizing that the analysis process is frequently akin to prospecting. When organizing the analysis team, both size and skill are critical. Small teams of two to four people per sub-system seem to work best. At least two people are needed to provide second opinions, and more than four tend to either leave some members out of the process or slow down as the diversity of opinions increases. The skill issue is both obvious and yet difficult to address. The obvious part is ensuring that each team has at least one skilled analyst. The difficult part is identifying the good analysts, and then keeping them evenly distributed among the teams.

When doing the analysis, a productive mind set is that of an oil prospector. It's impossible to accurately predict where the oil will be found. Rather than agonize over exactly where to drill, or continue to drill in areas which have not yielded anything, the best thing to do is make an educated guess, proceed with sinking some holes, and use the results to guide your future work. From an OOA perspective, this means identifying potentially fruitful areas (as described in the previous subsection) and proceeding to build the object information models for these areas. The results of this analysis will help guide the direction of future work.

Key Success Factor 8: Keep Analysis Work on Track

As the different analysis teams begin to develop and review their OOA models, new issues can arise which affect the ability to successfully sustain the analysis effort. These include confusion with details of the OOA formalisms, OOA models that are only understood by their developers, models which can never be completed, confusion with what to model and what not to model, difficulties with integrating the models from different teams, obsolescence of the analysis plans, and wavering management support as schedule pressures increase. A number of things can be done to address, and in many cases prevent, these problems.

- Augment the formal models with informal notes and diagrams.
- Review the models.
- Leverage available analysis experience.
- Distinguish between analysis and design issues.
- Maintain the domain chart and subsystem interfaces documents.
- Demonstrate tangible progress on a regular basis.
- Use results to calibrate and maintain the plan.

Augment Formal Models

The OOA formalisms (object information model, state model, and process model) are an excellent means to precisely model the conceptual entities and desired behavior of a system. They are not always sufficient, however, for understanding the underlying abstractions and engineering strategies upon which the models are based. What we want to capture are the white board pictures and explanations that initially produced the "Yes, that's the way to think about this!" insights.

For example, in a material handling system being analyzed, it was important to treat conveyors, automatically guided vehicles, and human carriers as sub-types of the same "delivery resource" super-type in order to provide system flexibility as delivery resources were changed. What works best in situations like this is to produce an informal document that explains the strategies and underlying abstractions behind the models as a *prelude* to doing the formal models. Using diagrams and pictures in this type of document greatly enhances its ability to communicate complex ideas.

With these informal documents (commonly called "technical notes"), it is possible to capture and communicate some of the most important work that goes on during the analysis phase. It provides a good foundation for developing the OOA models. It makes them more comprehensible after they have been developed. It allows conceptual approaches to be assessed prior to investing time in modeling them. And, it provides a convenient mini-step which aids in tracking progress.

Review Analysis Results

A discipline of reviewing analysis models when they reach an acceptable level of maturity must be established on the project. This monitors quality, measures progress, provides interactions between different analysis teams, guards against unnecessary "polishing", and produces a sense of progress as tasks are completed. Each model should be reviewed as it is completed. For each subsystem, this means a review for the object information model, state model set, object communication model, and process model set. For the system this means reviews for the domain chart and the subsystem information, communication and access models. Reviews can also be helpful prior to "shelving" a piece of work for some time, and when a piece of work is "stuck". These reviews should be formal assessments of the work product, documented with meeting minutes, and generally follow good software inspection procedures.

Leverage Available Analysis Experience

This experience typically come from two sources. First, from other projects within the organization that have used OOA, and second from within the project, as some team members pick up the analysis technique more quickly than others. These individuals need to be identified and effectively assigned to the important analysis problems to leverage their skills. This requires a certain degree of flexibility in being able to assign and reassign team members to tasks. This flexibility can be problematic in certain organizations and with particular individuals. It helps if the dynamic nature of assignments during the analysis phase can be established early.

If the availability of in-house experience is insufficient, outside help from experienced OOA practitioners should be sought to mentor the project team. This will significantly enhance the speed of the object-oriented technology transfer and ensure that things get off to a good start. Remember, bringing in expertise up front provides much greater leverage than bringing it in later to clean things up.

Distinguish Between Analysis and Design

Analysis should focus on the application. Design should focus on the implementation. When implementation issues begin to creep into the analysis, the job gets bigger and the issues get cloudier. The result is that the analysis process can bog down. For example, on one of the projects, the application analysts began worrying about the fact that they would require more instances of objects at run-time than their processor memory could handle. Their next step was to add instance caching capabilities to their initial application models which greatly complicated their work. The correct thing to have done here was to recognize that instance caching is an implementation issue and can be handled separately. One approach, for example, is to provide an architecture base class from which interested application classes can inherit the ability to be (transparently) cached. One needs to be on constant guard against implementation issues creeping into the application. The OOA courses are very explicit about this distinction, and the message needs to be reinforced throughout the analysis phase. The Recursive Design course can also help by clarifying how implementation is done through a translation of the analysis models.

Maintain Domain Chart

On the one hand, the partitioning of the system into domains and subsystems is essential. On the other hand, it cannot effectively be done in one pass. Not enough is known at the start of the project to do this correctly. Given this dilemma, how should the project proceed? A best guess, based on good engineering judgment, must be made at the start of the project. Thereafter, it is important to revisit this choice periodically and make adjustments as more is learned through the analysis. The process then becomes one of refining the system vision as more is learned about the individual components. One wants to avoid wasting time at the start of the project agonizing over the perfect partitioning. One also wants to ensure that the insights gained through analysis get factored back into the partitioning and an integrated system vision.

Demonstrate Tangible Progress

This should be done on a regular basis for two important reasons. First, it encourages continued management support for the project and its team. Very few management processes accept the Big Bang (results will suddenly appear overnight) theory of development and hence want to see incremental progress. Second, it has a positive impact on the morale of the team to recognize their progress. It's easy for them to lose sight of this progress when the daily focus is on the work that remains to be done.

There are a number of analysis activities that can be used to demonstrate tangible progress: a revised domain chart which reflects a better understanding of the different subject matters in the system; new conceptual insights documented in technical notes; completion of reviews; or successful subsystem partitioning of a domain allowing parallel work to proceed. The analysts

should take the initiative on documenting and communicating these achievements. The best defense is a good offense!

Calibrate and Maintain Plan

As the analysis effort proceeds, more is learned about the scope of the work and the rate at which it can be done. This information needs to be factored back into the analysis plan on a regular basis to understand its impact and allow for timely adjustments. The situation here is very analogous to creating and maintaining the domain chart. Good engineering judgment is used to initialize the plan and adjustments should be made as new information is available.

Adjustments to scope should track changes to the domain chart. Adjustments to efficiency can be made as soon as the team gains experience with the analysis techniques. For example, on one project, we used the actual analysis time per object in the application domain to estimate similar work in the service domains. In such a manner, a project can generate and use metrics as it proceeds.

Key Success Factor 8: Manage the Transition to Implementation

Just as you were advised not to take an indiscriminate, breadth first approach to initiating the analysis, you also should not plan to transition into implementation in this manner. Given that implementation with the Shlaer-Mellor Method means a translation (automated, we hope) of the analysis models, and that analysis and design (software architecture) occur concurrently, the most important consideration you must make is when you'll need which domains and ensure that you target completion of critical areas first. Domains can be critical for two reasons: they're required early in the implementation and system integration or they may be high risk. Early work on high risk domains reduces the possibility of unpleasant surprises late in the development lifecycle when they are difficult to recover from.

7. Specify Software Architecture Activity

Our main objectives in this activity are to understand and specify the characteristics of a software architecture that will meet the performance and size constraints of the application and can be realized on the underlying implementation technologies (e.g. processors, operating systems, compilers, CASE tool).

Key Success Factor 10: Understand the Technologies

There are two different types of technology that the architecture team will need to master to develop an appropriate software architecture. These are a thorough understanding of Shlaer-Mellor, both the OOA modeling and execution rules and the basic principles of Recursive Design, and a thorough understanding of the implementation technologies: processor, operating system, compilers, CASE tool, communication protocols, and third party software packages (e.g. a database). Things that can be done to address this need include the following.

- *Train Architects in Complete Method*. Don't just give your architects RD training thinking that they won't be doing analysis and therefore don't need OOA training. They need both.

- *Recognize the Need for Computer Science Expertise*. Computer science expertise is more valuable on the architecture team than application or product expertise. Put the "system software" experts on this problem, not the application experts.

- *Build a Prototype Architecture*. If time and resources allow, build a prototype architecture with less functionality first to gain experience with the RD approach and the automation tools.

- *Purchase a Prototype or Final Architecture*. If you don't have the time and/or resources to build your own prototype or final architecture, purchase one. This will get you up and running with the technology immediately and offer a concrete example from which your team can learn.

- *Utilize Experienced Architects*. As with any new endeavor, having someone on the team who has done this before can help avoid confusion and false starts. Consider the trade offs between obtaining this internally (hire) or externally (consultant).

Key Success Factor 11: Start Architecture Work Early

This simple point cannot be emphasized enough. Not starting early on the software architecture is by far the most frequent cause for projects having difficulties with Recursive Design and slipping their schedules accordingly. The architecture team gets behind and is forever trying to catch up as they are on the critical path for the final delivery of the system. It's not a pretty picture!
Remember, with Shlaer-Mellor, when the analysis of the application is complete, the software architecture work must be complete also. This is not an elaborative approach where the next step

is to begin adding implementation details to the analysis models. The next step in Shlaer-Mellor is to use the automation provided by the software architecture to begin generating code for integration and testing of the system. If the software architecture isn't ready when the analysis is complete, the analysts are very likely to begin hand implementing the analysis models and chaos will soon reign on the project. Things that need to be done to address these issues include the following.

- *Create a Separate Architecture Team from the Start*. At the project planning step, you should be establishing the organizational structure for your development team. Create a software architecture team as a separate entity at the beginning of the project. Do not wait, there are many tasks they can be productively working on right from the start. If you anticipate a large architectural effort, you can phase the building of the team over time, but start with at least the team leader on day one and build as the work load demands.

- *Make Timely Decisions on Implementation Technologies*. A common cause for delay in getting the architecture work started is that implementation technologies have not been selected yet. "What's the rush?", you'll hear the decision makers say, "We've got months of analysis ahead of us before we'll be ready to code." Yes, but in the meantime, you have to specify and build a software architecture based on these unmade decisions -- an impossible task. Recognize the dependency here and make timely decisions.

- *Phase Features*. If all else fails, and you still find yourself behind schedule on the software architecture, consider phasing features so you can make less fully featured architecture available to the project earlier.

Key Success Factor 12: Track Application Requirements

A software architecture that doesn't meet the needs of the application, either performance, size, or ease of use, fails. To avoid this failure requires the attention of both the application teams and the architecture team. The following steps can be taken to reduce the risk of this failure.

- *Application Team Reviews Architecture Work*. Make sure the application team understands the manner in which their models will be translated.

- *Architecture Team Reviews Application Work*. Make sure the architecture team understands the architecture requirements implicit in the analysis models.

- *Perform Hand Translation Early*. Demystify the translation process by walking through hand translation of the OOA models as soon as the software architecture is specified.

8. Automate Software Architecture Activity

Our main objectives in this activity are to design, build, and test the automated translation of the OOA models into code.

Key Success Factor 13: Address Performance

A common failure mode for the architecture work is to not meet the speed, size, or flexibility requirements of the system. This frequently happens because the architects mistake the execution rules of the OOA models to be the "one true architecture". That is, they formulate a one-to-one mapping from the OOA models to constructs in the software architecture. This can result in such inefficiencies as an event queue per instance, relationships implemented only with referential attributes, and so forth. Successfully addressing performance in this and the previous phase as well, the following things are recommended.

- *Balance Ease of Translation with Performance*. The reason the one-to-one mapping is a popular mistake is that it's easy to do. Recognize this trade off and make sure you can make it. If not, it's best to recognize it and begin the investment in the additional complexity to provide the necessary performance.

- *Benchmark Key Implementation Technologies Early*. Measure the performance of your key implementation technologies early to see what "headroom" they leave the architecture. For example, how fast can your operating system perform a context switch? How fast do you get data on a cache hit in your database? On a cache miss?

- *Benchmark Key Architecture Components Early*. Measure the performance of key architecture components early, when there still time to make changes. What's the overhead on commonly used items like generate an event, dispatch an event, get an event between tasks, between processors? How much memory does an object incur, an instance of an object, a one-to-many relationship?

- *Evaluate Key Threads of Control*. Using the above benchmark data, calculate the time required for key threads of control in the application.

Key Success Factor 14: Develop Good User Documentation

The analysts are typically the "users" of the software architecture. They use it to translate their analysis models into code so they can test that code and integrate it with the rest of the system. If good user documentation does not exist, they will become frustrated with the architecture and can subvert the success of a perfectly good architecture. To avoid this, the architecture team needs to produce good user documentation, a fact that is often overlooked in the rush to automatic code generation. Another thing that can further help, or augment poor documentation, is to provide user training on the architecture for the analysts.

9. Implementation Activity

Our main objective in this activity is to translate the OOA models into code and integrate and test the final system. This is usually done incrementally, one domain or subsystem at a time.

Key Success Factor 15: Integrate People, Not Just the System

The implementation of the application and service domains is an integration of people as well as an integration of subject matter and code. Teams that have been working separately will now need to work together. A number of things can be dome to facilitate this.

- *Cross Review and Train.* This has been mentioned earlier and is bought forward again to point out that it serves multiple needs.

- *Use Architects as Implementation Mentors*. Architects whose work has been completed by this phase can be reassigned to application teams as implementation mentors. Given their experience with the architecture and the implementation technologies, they are well suited to assist the application teams in the implementation phase.

- *Use Architects as System Integrators*. For many of the same reasons architects make good implementation mentors, they can also make good system integrators and relatively impartial testers as well.

Key Success Factor 16: Plan System Integration and Test

This point is almost universally valid for any effort using any method. The sequence with which one integrates and tests the system should be the result of a conscious strategy that addresses specific needs of the project. When working with custom hardware, for example, getting diagnostic code ready first is a high priority. Or when exhibiting a preliminary version of the system at a trade show is necessary, identifying, modeling and integrating the components required for the show is important. For a Shlaer-Mellor project, the following things can help.

- *Integrate and Test Architecture First*. The software architecture can be integrated and tested totally separate from the application models and code. This should be done to ensure that you don't end up testing two things at the same time: the architecture and the application.

- *Validate Critical Application Models with Verifier*. Parts of the analysis models that are known to be important of difficult should be validated, prior to translation into code, in the CASE tool verifier or simulator. Again, let's test one thing at a time.

10. Activity Independent Factors

There are some success factors that involve many or all activities of the development effort. These are addressed here.

Key Success Factor 17: Staff Properly

Assigning the correct number of properly skilled staff to each phase of the project ensures that work proceeds in the most effective and efficient manner possible. Too many people at the wrong time actually slows things down. Too few, stresses those that are attempting to get the work done. And mismatches between an engineer's skills and the assigned task can result in either great inefficiencies or failure. Our experience in this area has resulted in the following recommendations.

Task Profile for Staffing

- *Building the Domain Chart*. This should be done by a core technical team of three to five people, with expertise in the application and strong system engineering and conceptual skills.

- *Project Planning*. This should be done by the core management team, typically the project manager and team leaders. Familiarity with the method and the skill sets of the staff are essential.

- *Application Domain Analysis*. This should be done with a small (two to four person) team per subsystem. Members should be experienced with the application and have strong analytical, fact-finding, and method skills. The fact-finding skills help them to determine the true application requirements when they are poorly defined.

- *Service Domain Analysis*. This should also be done with a small (two to four person) team per subsystem. Members expertise may be domain specific, and they should have strong analytical, synthesis, and method skills. The synthesis skills help them to create structure and requirements in the service domains that are not always specified in the product requirements.

- *Software Architecture Domain*. The size of this team should reflect the complexity of the architecture, not the size of the system. We have seen teams as small as two and as large as twelve. Members should have excellent design and programming skills and be experienced with hardware architectures, operating systems, programming languages, third party software packages, and system software. They must also have a detailed knowledge of the OOA models and their execution rules.

Time Profile for Staffing

- *Do Not Over Staff at Start*. A common mistake is to have twenty people ready to start analysis on day one of the project, before the core technical team has partitioned the system and the management team has planned the work. Great spinning of wheels and wasting of time usually follows. Parceling the work out to this many people also dilutes the conceptual integrity of the system and slows progress.

- *Staff Architecture Domain as Early as Possible*. This was brought up earlier and it is repeated hear to emphasize its importance. There is almost always work to be done in this area starting day one of the project.

- *Plan for Staff Increases*. These will take place when the analysis fans out into the service domains and when the architecture transitions from specification to implementation.

- *Plan for Decreases*. These will take place when the analysis is completed, when subsystems are successfully unit tested, and when subsystems are successfully integrated.

Key Success Factor 18: Use Your Experience

A common error made by managers when introducing object-orient technology into their organization is to assume that their years of experience with planning and managing software projects are no longer valid in the context of this new object-oriented approach. Yes, some things do change, but the fundamental issues involved with managing a team of software engineers and adhering to a disciplined process do not change and should not be compromised.

- *Define Process Around the Method*. The Shlaer-Mellor Method prescribes specific engineering techniques for the tasks one encounters when developing software and it structures these techniques into a sequence shown in Figure 1: The Shlaer-Mellor Method. Notice that these techniques focus on the technical problems associated with constructing software, and do not address the more general management and process issues associated with the effort. How is reviewing and reporting done? How is configuration management done?. How is quality measured? Improved? These are all important questions to consider and address on your project if you are to succeed. While Shlaer-Mellor will provide an excellent engineering method upon which to build a disciplined process, it does not do this on its own. You must define a process around the method.

  Various process models and standards exist for the software development process including:

  - SEI's Software Maturity Model [9],
  - ISO's 9000 series of standards for software development,
  - DOD's 498 standard for software development, and
  - IEEE's 982 standards for software engineering.

Build on the model and standards with which you are familiar when defining your project-specific or organization-specific process for using the Shlaer-Mellor Method.

11. Summary

There is a lot more to succeeding with Shlaer-Mellor than mastering the modeling techniques and recursive design. Some of it is just good project management practices. Some of it is just good analysis practices. Some of it is just good design practices. Some of it has to do with the process of introducing a new engineering discipline into an organization. And some of it has to do with the general, object-oriented approach of Shlaer-Mellor. If the issues in all of these areas are actively tracked and addressed, the chances of success are greatly enhanced. The following checklist summarizes critical activities for project success.

Key Success Factors for Your Project

- ✓ 1 Assess Applicability of Shlaer-Mellor Method
- ✓ 2 Gain Management Support
- ✓ 3 Train Staff in Shlaer-Mellor
- ✓ 4 Establish Productive Environment
- ✓ 5 Build a Good Initial Domain Chart
- ✓ 6 Create an Executable Plan
- ✓ 7 Get Analysis Work Moving
- ✓ 8 Keep Analysis Work on Track
- ✓ 9 Manage Transition to Implementation
- ✓10 Understand the Technologies
- ✓ 11 Start Architecture Work Early
- ✓ 12 Track Applications Requirements
- ✓ 13 Address Performance
- ✓ 14 Develop Good Software Architecture User's Guide
- ✓ 15 Integrate People, Not Just The System
- ✓ 16 Plan System Integration and Test
- ✓ 17 Staff Properly
- ✓ 18 Use Your Experience

References

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

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

[3] Private conversations with Leon Starr, 1991.

[4] Tom DeMarco, *Structured Analysis and System Specification*, Yourdon Press, 1978.

[5] T.E. Bell, T.A. Thayer, *Software Requirements: Are They Really a Problem?*, Proceedings, 2nd International Conference on Software Engineering, 1976.

[6] Harlan D. Mills, *Software Engineering*, IEEE Transactions on Software Engineering, Volume SE-2, No. 4, December 1976.

[7] Sally Shlaer, Stephen J. Mellor, *Recursive Design*, Computer Language, March, 1990.

[8] Meilir Page-Jones, *Object-Orientation: The importance of being earnest*, Object Magazine, July-August, 1992.

[9] Sanjiv Gossain, *Using the Shlaer-Mellor Method for SEI Level-4*, Project Technology White paper, 1993.