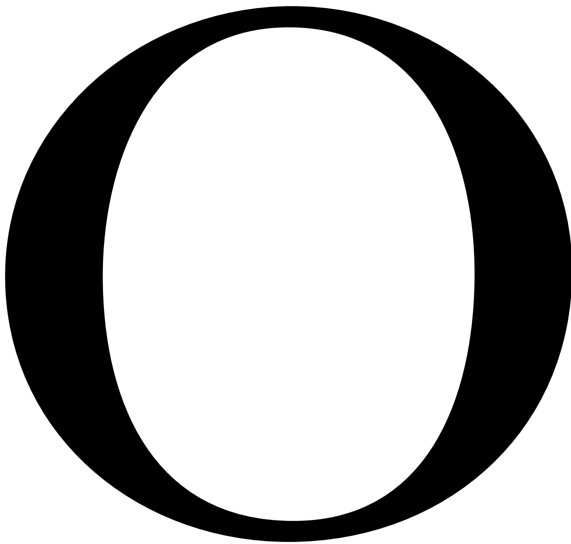
Transition
To Object-Oriented Software Development

A transition plan based on lessons learned from real-world experience is presented and several effective managerial practices are recommended.



bject-oriented software engineering (OOSE) introduces a software development model based upon the way humans think. OOSE promises remarkable benefits. It allows software components to be readily reused, potentially saving substantial development costs. Also, OOSE reduces the scope and side effects of software changes, potentially saving substantial maintenance costs.

In spite of these potential benefits, many software development groups still hesitate to use OOSE. Many of these groups are fully occupied by their current software development work and fear being overwhelmed if they introduce a new development paradigm.

Other groups, interested in OOSE, do not know where to begin. "What's the first step?" "How long will it take?" "Is it really worth it?" Six years ago members of various software development groups where one of the authors was working as a consultant asked these same questions when they decided to confront the task of developing a large software system using OOSE. They were confident that they understood the benefits of object-oriented (OO) techniques, but they did not understand the theory behind OOSE. They lacked a firm foundation from which to start and practical experience on which to build. Furthermore, many of these teams had never applied any formal or semiformal techniques.

To make matters worse, these groups had additional problems: their most recent development efforts were in most cases "inconsistent"; they were usually behind schedule—perpetually 90% complete; they were constantly finding problems late in development and feverishly applying band-aids to plug the leaks. Fixing one problem frequently created others. Also, despite having a highly centralized decision-making organization, individually developed software parts still looked radically different. Programmers or software engineers developed software by applying their own unique process. In retrospect, the overall software development process was chaotic.

The transition to OOSE is often problematic, and object-oriented approaches to software development are becoming increasingly prevalent. Even though research concerning the technical aspects of developing OO software is plentiful, many divergent opinions exist. There is little guidance for OO software development managers on how to transition to OOSE.

On average, the transition of a single software development team to OOSE takes at least a year. Throughout this transition period, they transform their organization from an ad hoc, informal, or chaotic group into an efficient OO development team. Some people may think the transition period would depend on the size of the projects, the number of people involved, the techniques used, and so on. The project size has nothing (or little) to do with getting people to change how they approach software development for problem solutions. The people can change only through the application of the process. It takes about a year of this sort of practice for the transition to be complete.

During the last seven years, we have been involved in the transitions of large software development teams from ad hoc, informal, and extremely chaotic to OOSE [11–14]. The transition became quicker and easier on each new project. The transition plan that is the topic of this article has been fine-tuned

Mohamed E. Fayad, Wei-Tek Tsai, and Milton L. Fulghum

over several subsequent OO efforts. This article presents the most significant evolutionary steps, many of which were not initially apparent.

The OO transition process encompasses three stages: the planning and pre-project stage; the technology insertion stage; and the project management stage.

Planning and Pre-Project Activities

The planning and pre-project stage encompasses two major activities: effective development planning and changing the existing culture.

Effective Development Planning

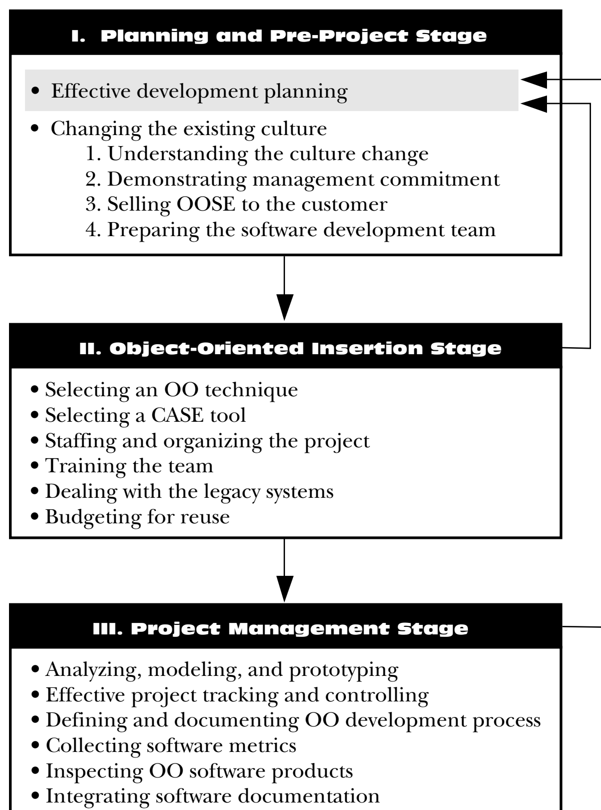
Before a project begins, we must make key decisions that can make or break the project. The current software development processes and our personnel must undergo a *calculated* change. What activities should we perform before the project begins? How do we change from ad hoc, informal, or chaotic procedures to OO techniques?

Thorough preparation and planning will greatly increase the chances of success. Software project planning is a key process area in Level 2 of the SEI Capability Maturity Model (CMM). The first project probably will not encompass all the OO software principles we desire. Comprehensive use of OO principles takes several projects to fine-tune—careful transition planning and software development will smooth the transition. Software developers who have gone through a rigorous process that uses structured analysis should be familiar with some of the management issues, but they still face some of the old problems in a new context—OO. For example, instead of worrying about the data, processes, and data flow among the processes, they need to think in terms of objects, methods, and inheritance. A transition plan and a complete software development plan are necessary.

Transition Plan. The transition plan is examined with respect to two central themes: "What makes the transition to OOSE a mission with a lot of problems?" and "How can the transition be accomplished with minimum impact on the cost and schedule?" An assessment of the software development processes is a must. This will establish some sense of where the software development team is. For a smooth transition to OOSE, an effective transition plan must be followed. Figure 1 illustrates a framework for a transition process.

Software Development Plan (SDP). The first document for any software development project should be the SDP [9]. For government contracts, this document becomes part of the contract and is the final word on what the developer is going to deliver to the customer. For both government and commercial projects, the SDP indicates who will do what and when they will do it. It describes what techniques to use, what tools to employ, and what risks are involved. For the software developer it is probably the most important document produced because it defines the scope of the project.

Figure 1. A framework for a transition process



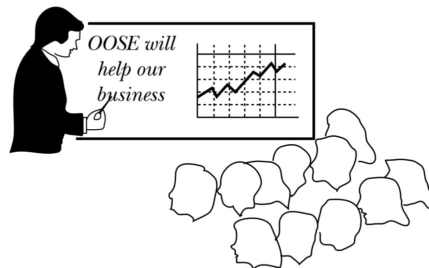
Changing the Existing Culture

Adopting OO development techniques will definitely require a change in culture in the entire project organization. This change can be made gradually and systematically. Almost every step of the software development process must be adapted to fit the new way of thinking. As a result, the following activities are recommended to help readers achieve the necessary change in their culture.

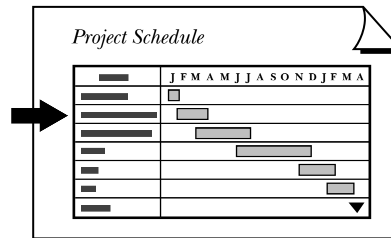
Understand the Culture Change. OOSE affects everyone participating on the project, not just the software engineers. It is, therefore, necessary to change the culture in disciplines outside as well as inside the software engineering group. Changing the culture in any organization is not an easy task. For example, during the transition to OOSE, we felt that it would be easier to convince people that the world is flat than to convince them to use OOSE.

Demonstrate Management Commitment. Perhaps the single most important factor in changing an organization's culture is demonstrating management commitment. Humphrey points out that a basic principle of software process modification is that *major changes must start at the top* [15]. The following steps, which are shown in Figure 2, can help demonstrate management commitment:

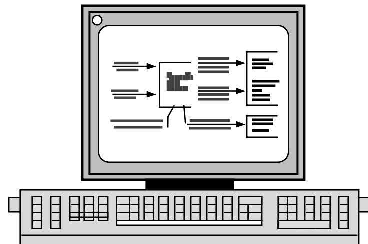
- **Demonstrate commitment to process changes.** Top-level management not only should start the changes but must also demand or reinforce the

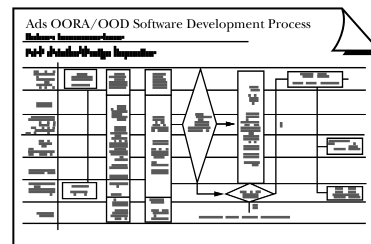


Demonstrate Commitment To Process Changes



Provide Time In Schedule To Foster Change


Provide Proper Tools



Strictly Enforce The New Process

Figure 2. Culture changes require management commitment [6, 7]

changes. Our experience confirms the top management must frequently remind people of the importance of process changes. Otherwise, people tend to slip back to their old ways of doing things due to familiarity and habit.

- **Provide time in the schedule.** The project manager or software manager should set aside at least a week or two at the beginning of the development process for training and discussion of process changes. During this time managers should expect no project-related work to be done. Team members' full concentration should be on the new processes and their definition.
- **Provide proper tools.** Although CASE tools are often expensive, we believe they are essential to the new OO process. Without proper tools, software engineers waste effort and creativity on trivial tasks instead of the analysis and design of the software system. By not committing any capital to the purchase of tools, management sends a wrong message—that it is unsure about the new paradigm shift and the success of the new OO process. Software engineers may interpret this as a lack of management commitment to the new OO process.
- **Strictly enforce the new OO process.** Almost certainly, there will come a time when a member of the team wants to bypass all the rigorous OOSE activities and develop the source code directly. This individual will undoubtedly provide numerous reasons why going directly to coding will save time and money. Managers should not back down from the process change under any circumstances. A manager who backs down, even in the most trivial of cases, will not only be setting a bad trend but will also demonstrate a lack of commitment to the newly defined process.

Selling OOSE to the Customer. In addition to clients and users, the customer is defined as the high-level management, other projects' managers, the project team next door, colleagues on the same team, and so on. In most cases we must overcome customer resistance to using OOSE, especially if the customer knows it is our first OOSE project. For an organization to succeed, it is important that the customer be involved in the transition to OOSE. Customers frequently associate new technology, such as OOSE, with high risk.

The following steps may help to make the customer a believer:

- **Provide capability briefing charts.** The customer usually has a negative attitude toward software projects with any element of risk. Many people consider an OO development high risk because they feel the tools and techniques are immature. To overcome the customer's concerns, we provide a set of briefing charts that address the perceived risk areas. The main purpose of the charts is to demonstrate that we have thought through our transition plan. Some example topics are benefits versus cost; training schedule; CASE tool strategy; OO process documentation; organizational issues; and return on investment (ROI). We have found that the more we prepare for the transition, the smaller the customer perceives the risk to be.
- **Train the customer.** If possible, the customer should attend formal training on OO topics with the development team. If the customers are unfamiliar with OO terminology, they may feel they have lost control of the project. We diminish their uneasiness by including them in training as early

and as much as possible.

- **Explain long-term payoffs.** Make sure the customers understand the long-term payoffs of OOSE. Some important benefits of an OO development, such as reusability and maintainability, directly affect the customer's pocketbook. The cost saving is not firmly established, but initial data seems to be encouraging; more data is needed to establish this.

Preparing the Software Development Team. Surprisingly, we must often overcome resistance from our software development team. There seems to be an instinctive suspicion of any deviation from the existing software development process, even if the process is chaotic. The team's first opinion of the new process is typically that it requires significantly more work for small benefit. The major purpose of the team preparation is to get people who have become accustomed to working independently to operate as team members. The principle of software process improvement is that ultimately everyone on the project must be involved [15]. The following ideas can be used to prepare the development team for the transition:

- **Work with Process Improvement Groups.** We must work with our company's process improvement groups to document the project software processes and to get the team involved in documenting the new process, taking responsibility for the process, and establishing self-ownership of the process. Do not establish a special process group on the project. Why? Because the members of the software development team are software developers, not process builders. With the help of our team, the company's process improvement group upgrades the new processes to fit the project needs. This process forces the software engineers and other members of our team to understand the development processes so they can claim ownership in them. Both the understanding and the ownership of the process are keys to successful OO development.
- **Initiate Participative Management.** In participative management, the developers help to generate schedules and make management decisions. The greatest benefit is to enlighten the development staff as to the importance and the sequence of the various software development stages of a formal development. Under participative management, development teams gain an awareness of the schedules and the products that need to be delivered at each important milestone. Participative management offers development teams an even greater feeling of buying into the new OO development process. Although participative management may seem a bit risky considering everything else we are changing, we still recommend this technique.

Object-Oriented Insertion Activities

OO insertion is the second stage of the transition process. This stage focuses on the insertion of OO technology before the initial application of the technology.

Selecting an Object-Oriented Technique

The first insertion activity, and possibly the most important task, is to select one of the many available OO software development techniques. The OO technique provides the step-by-step activities and a set of graphic notations to be used for reviews, inspections, and documentation. Therefore, this selection will have an impact on almost every step in the software development process.

As OOSE has matured, the number of projects applying OO techniques has multiplied. Choosing some of the more popular methods and trying to determine which one best fits the application is a difficult area to master.

Consider the following factors, shown in Figure 3, when selecting an OO technique:

Select a fully object-oriented technique. We highly recommend choosing an OO technique that covers as much of the software life cycle system as possible. Most OO techniques are focused only on requirements analysis, software design, and OO programming languages, resulting in a potpourri of representations and models. Unfortunately, testing and verification and validation (V&V) are completely ignored in most OO techniques. Using OO techniques on a single activity (i.e., only analysis or only design) is risky. For example, some projects have used a functional analysis approach followed by an OO design approach. Paradigm shifts in the middle of a software project have generally ended in failure.

Similarly, taking an OO analysis technique and completing it with an OO design technique can produce unpredictable results. Mixing of techniques should be left to teams experienced in OO development and not to an organization in transition. Some OO techniques are available that take a software organization from requirements analysis through at least the detailed design activity. These are the most appropriate.

Domain considerations. The application domain is usually supported by a particular class of OO techniques. Partitioning the available techniques into unambiguous classes is impossible. However, one simple guideline we use is a distinction between a top-down abstract object approach and an information modeling approach.

The top-down abstract object approach uses a successive breakdown of objects. The analyst successively decomposes high-level objects into lower-level objects until the lowest-level objects are all easily understood. Computation-intensive systems in which the data is not initially well understood (e.g., digital signal processing, pattern recognition) map better to a top-down, abstract approach. Examples of OO techniques using a top-down approach are Colbert's OO software development (OOSD) technique [4, 7, 12, 13], Fayad's object-engineering technique (O-ET) [10], and Selic's real-time OO modeling (ROOM) [8, 18].

An information modeling approach uses data objects as the starting point and builds up from there. Information management systems (e.g., payroll systems, scheduling systems, and insurance applications)

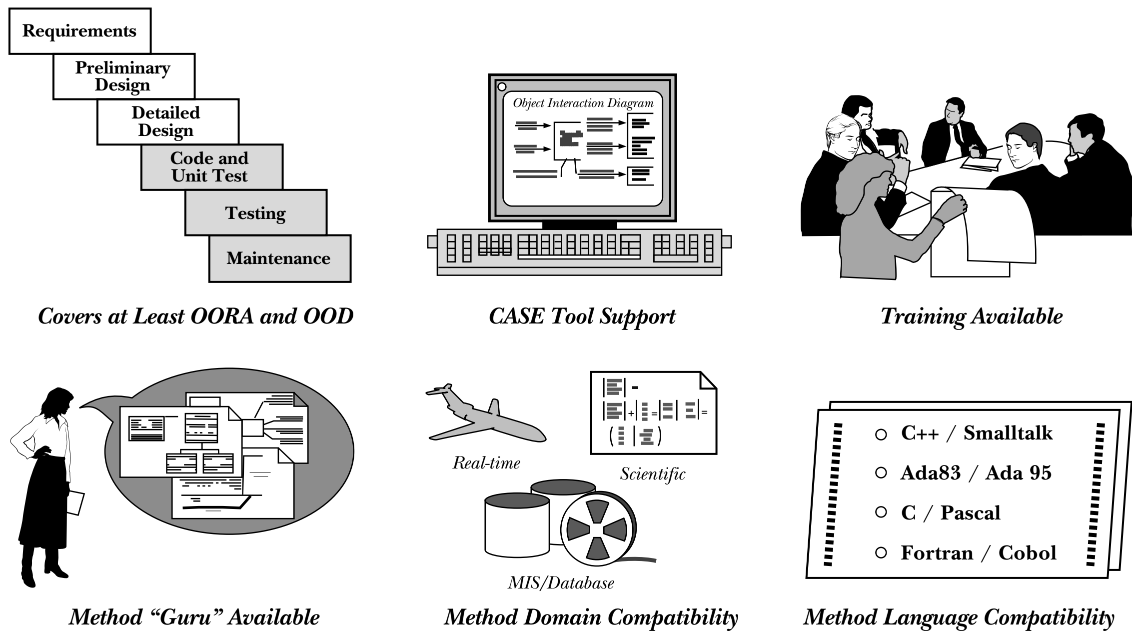


Figure 3. Six major factors for selecting an OO technique [6, 7]

map better to an information modeling approach. Since the analyst fully understands much of the data early in the project, how to build objects from the data is reasonably clear-cut. Examples of OO techniques using an information modeling approach are Object-Modeling Technique (OMT) [17], Fayad's O-ET [10], and Shlaer and Mellor's OOA [6, 7, 13, 19].

Target language considerations. Some OO techniques conform to the features of a particular target language. For a first OO development it is best to use an OO technique that maps easily to the target language.

For example, if the target language is Smalltalk or C++, choose a technique that fully uses the features of these languages. OO techniques like Fayad's O-ET [10] and OMT [17] use inheritance as a major feature of the technique. Other techniques may not use inheritance at all. Since Smalltalk and C++ feature inheritance as an integral part of the language, we want to use a technique that has inheritance with them.

Availability of formal training. Typically, each software development team member will learn each step of the technique and then apply it to a small example. Obviously, each member will not be an expert immediately after formal training. But at least each member will get to see how an experienced user creates objects using the specific technique. This type of knowledge is often impossible to derive from a textbook describing the technique. One major warning: Make sure formal training is available from an experienced user before selecting a particular technique.

CASE tool support. For a first OO development, it is essential to have a CASE tool that can adequately implement the chosen technique. The CASE tool lets the developers concentrate on the analysis and design of the system and not on the mechanics and diagramming of the OO technique.

Consider a consultant. A good consultant can save a great deal of time by helping the development team with OO technique evaluation and selection processes. In addition, new techniques are frequently being published and old techniques are constantly being updated. This is a difficult area to track on a part-time basis. A good consultant could be well worth the cost.

Selecting a CASE Tool

Modern CASE tools are the subject of controversy, having been credited and blamed for substantial increases and decreases in productivity. Few would argue that the right tools in the right hands can effectively increase software quality and reduce costs. The key is identifying the *right* tools.

We found the use of a CASE tool essential to a successful transition to OOSE. However, be aware that the CASE tool will not eliminate all the obstacles associated with first-time OO development. There is still no substitute for qualified software engineers who are open to change. The following key factors are important in selecting a CASE tool.

Implement the technique correctly. Numerous CASE tool vendors (and even the OO authors) have provided a "mapping" of a particular OO technique to CASE tools that implemented structured analysis and design. Although this is undoubtedly possible with careful engineering, we found it intuitively difficult. Additionally, people familiar with the functional graphics often have trouble making a complete switch to OO thinking. Avoid using a CASE tool that requires a "mapping" of the graphical representations to a particular technique.

Enforcement of OO technique. The CASE tool should

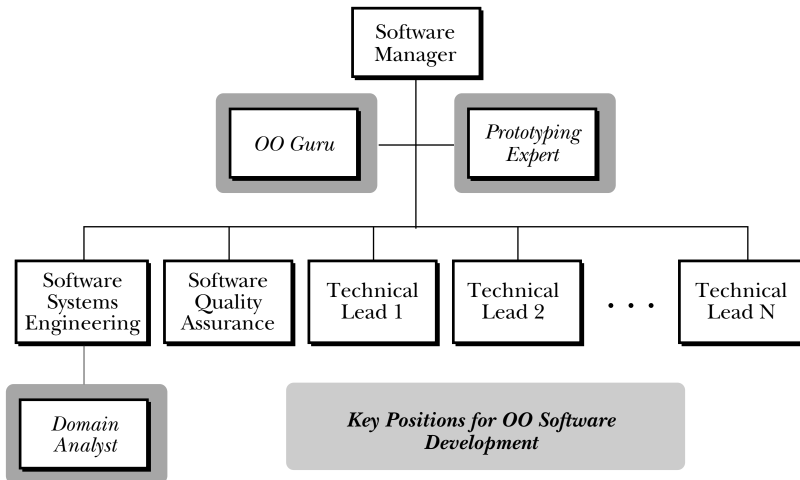


Figure 4. Key positions for a successful OO software development

enforce the rules of the OO technique as much as possible. The best time to find mistakes is while the developers are making them. If the CASE tool can catch these faults before reviews or inspections, then a great deal of time will be saved. This is especially important when the technique is new. The continual enforcement of the technique by the CASE tool serves as a good training mechanism for the developers.

Watch out for low pay-off "features." CASE tool vendors are constantly trying to add more features to outdo the competition. Features such as "reverse engineering" or "automated code generation" are generally not the most important reasons to buy a CASE tool. We found these items to be marginally useful. The most important parts of the CASE tools were graphical editors that enforce consistent application of the graphical rules. Consistency checking between diagrams and a data dictionary are useful.

Also be aware that the CASE tool will not work entirely as the vendor promised. For example, after several tries we eventually gave up on one feature of a CASE tool: "automated documentation generation." The poor quality and high overhead of this feature convinced us we were better off using a commercial word processor.

A CASE tool should also support several OO techniques and their development processes, and must be flexible for generating and updating these development processes.

Staffing and Organizing the Project

Staffing for a first OO software development requires special consideration. We found that the standard software engineering organization did not support the needs of OO software development. As teams proceded through the development process, they made several improvements to their software organization. Key positions, such as OO guru, domain analyst, and prototyping expert ("super hacker"), are essential to successful OO software development (Figure 4). The people in the new positions have direct responsibility for eliminating several problems that development teams have encountered early in development.

Staffing of the new positions within the organization requires careful consideration. Software engineers who have experience in the application of complete OO development techniques are difficult to find. How should we organize and assign our staff to minimize the risk involved in introducing OO techniques?

OO Guru. An OO guru is required in the organization to refine the use of the OO technique. Even though each team member receives formal training in a specific OO analysis and design technique, there are still many technique-related questions. Numerous clarifications and extensions of the technique are necessary to accommodate the unique elements of the project, such as reviews, documentation, particular CASE tool, target language, etc. Poor decisions in technique-related areas can necessitate extensive rework efforts. Therefore, appropriate staffing of the OO guru position must be a high priority. Figure 5 shows some of the OO guru's responsibilities.

Domain Analyst. A primary goal of OOSE is to lay a firm foundation for software reuse. The project needs a domain analyst to specifically address the reuse problem. The domain analyst is also

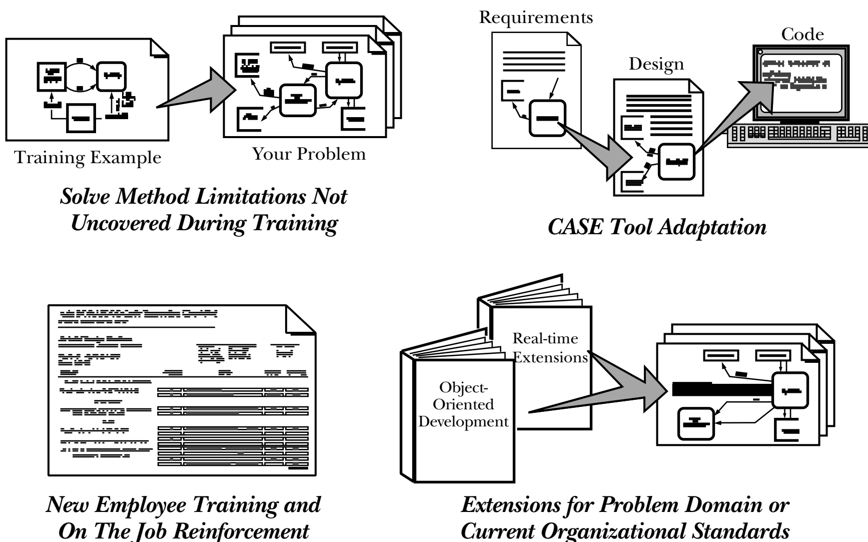
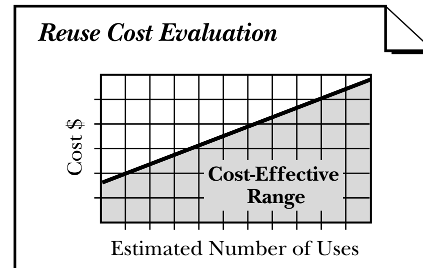
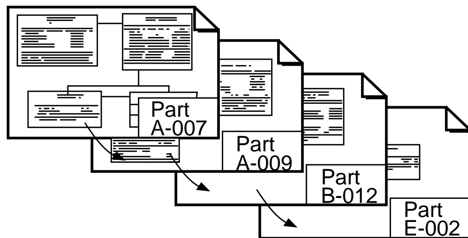


Figure 5. OO guru's responsibilities [6, 7]

Coordinate With Domain Experts On Different Projects



Decide When It Is Cost-Effective To Design For Reuse



Develop Specifications For Reusable Software Parts

Class Library

Class	Part No.	Description
Sensor	A-001	Sensor Monitors
Quick Sort	B-001	Efficient Sort
Signal	A-002	Alarm Signals
GPS Navigation	C-001	Nav with GPS update
Air Data Unit	A-001	Pressure/Temp
Inertial Navigation	C-002	Basic Nav Routines

Create/Maintain A Class Library

Figure 6. Domain analyst's responsibilities [6, 7]

responsible for understanding multiple software projects within the domain, extracting the common elements, and designing practical, reusable software parts and components. Figure 6 shows the domain analyst's responsibilities.

The domain analyst must be an expert in the application domain. Without a thorough knowledge of the application area, it will be difficult to create common parts. The domain analyst must also have a firm understanding of "classes" and "objects." Encapsulation of the details within a class or object is important. If too many internal details are visible, designers will be less likely to use the part.

Prototyping Expert. Even in the middle of semi-formal OO development there is still a need for a super hacker—someone who can develop a software product rapidly and correctly without necessarily following any specific techniques. The activities of the prototypers must be carefully separated from those of the rest of the software developers. This is especially true when we are using OO techniques. The software product will not be consistent if software designers incorporate prototypes directly into the formal development. Figure 7 shows the prototyping expert's responsibilities.

Once the prototypes are complete, the prototyping staff must communicate the requirements to the development personnel, who are then responsible for OO development. Therefore, an important qualification for the prototyping position is the ability to explicitly communicate the prototypes to the development staff.

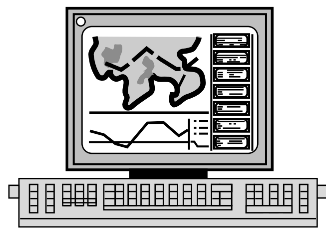
Other personnel. It is important that all software personnel understand the OO concepts quickly. We have had the best success with people experienced in an OO programming language. Even though these people had not used an OO technique, they could relate the concepts to the OO language constructs. Relating OO principles to language constructs is a good way to understand and explain the more abstract concepts of OO development.

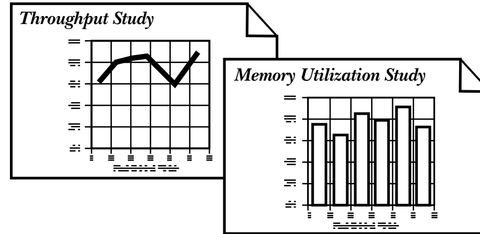
Training the Team

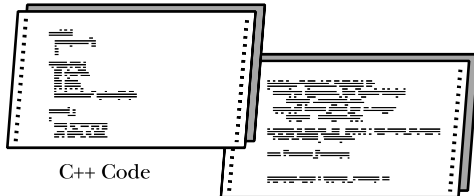
Training is essential for new OO development teams. Even experienced software developers benefit from a formal introduction to OO tenets. An early training investment pays large dividends in increased quality and productivity throughout the development process. Additionally, just as with reusing OO software systems, the benefits of reusing an experienced, highly trained work force increase dramatically as we apply their knowledge to the next project.

After selecting an appropriate OO technique, we arrange for in-house training. For a first-time OO development, we recommend contracting with an experienced training organization. Such an organization can provide experience and valuable insight into application of OO techniques and tools. This approach is helpful for the development team, in that it forces the software developers to clearly explain the project to the trainer, who has little domain knowledge.

A requirement for 40 hours of training may be fulfilled in one week if that week is totally dedicated to the training activity. The same training may be spread across two weeks if employees are available only half-time. The training may also be spread across 10 weeks with the addition of a part-time consultant. The







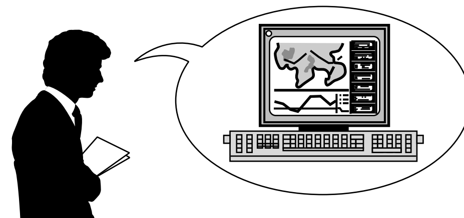


Figure 7. Prototyping expert's responsibilities [6, 7]

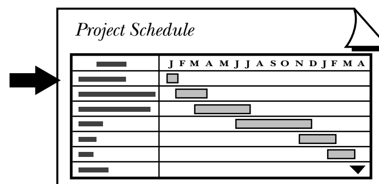
authors favor the last approach, for several reasons. Compressing the training tends to overwhelm the students. They are exposed to too much, too fast. Spreading the training across a longer period allows the students time to reflect on and practice each significant detail. Adding a consultant provides the opportunity for much more one-on-one learning.

Software engineers are not the only people who benefit from training. Each member of the development team, including managers, contract engineers, cost estimators, and system engineers, needs to understand his or her role in an OO development. A training program should be tailored to complement the entire development team. Figure 8 shows the five W's for training the project team.

3. Who

Customers
Software Engineers
Managers
Quality Assurance
Subcontractors
IV & V
Testing

Everyone Involved In Project



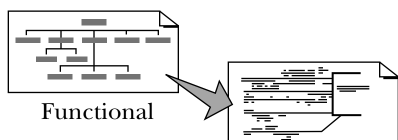


Figure 8. The Five W's for training the project team

Dealing With Legacy Systems

The preexisting non-object-oriented software system is called a legacy system. This issue is of great concern to management and will come up often. The ways of dealing with legacy systems can be categorized as follows: (1) use as is; (2) reverse-engineer them; or (3) modify them to fit with OO systems. Each of these approaches has its own set of problems and issues that we must deal with. We used the third approach, using the preexisting software modules by creating an OO shell around them, as

shown in Figure 9. The shell approach provides a cleaner interface, and the OO software will interact with these modules as existing objects or classes.

Budgeting for Reuse

Reusable software costs extra time and money to develop. If there are software parts that must be reused, be sure to budget the time and resources to make them reusable. Determining how much to spend on reuse is a strategic decision. We must determine how much extra money we want to spend right now to save money in the future. It should be remembered that money spent now will save money in the future.

We should not expect huge gains from our first OO project. Many people still associate OO design with automatic reusability. We found that just because we used an OO technique, that did not make the code reusable. The software organization must learn how to develop reusable software. Therefore, we recommend budgeting at least a small portion of the effort to developing reusable software.

Object-Oriented Project Management Activities

This is the third stage of the transition framework. Up to now, we have focused on the preparation—laying a foundation for a significant cultural and paradigm shift. Once the project begins, additional activities are required.

Analyzing, Modeling, and Prototyping

Because OO techniques optimize the life-cycle costs, reusability, and supportability of software development, penalties often arise in performance and memory utilization. This can be especially true with inexperienced OO development teams, who may over-apply some OO tenets. For example, we found that many new development teams tend to ambitiously create several layers of abstracted objects. The resultant increase in system overhead is usually not noticed until system integration. It is essential that software parts considered critical to project success be fully evaluated early in the project life cycle to reduce development risks.

The first step is to establish a computer resource budget. This budget should be based on a full examination of the capacity of the target machine, including memory, CPU performance, and I/O bandwidth. This budget must now be compared to the specific requirements of the application program.

This is where the plan gets interesting. Detailed resource requirements are difficult to establish, especially for OO developments. Designers must understand not only the resource requirements of the algorithms but also the overhead associated with their construction.

OO designs may introduce more overhead than traditional developments. This overhead ranges from implementations of polymorphism and inheritance to dynamic binding. Also, as designs maximize object encapsulation and privatization, dedicated methods and messages must be created to gain access to hidden data. The resource impacts of all of the OO trade-offs must be understood before the software design can safely mature.

Software prototypes, on the other hand, are functional approximations of some aspect of the actual target system. These approximations can be generated to evaluate alternative software designs. Eventually, these prototypes evolve into "rules of thumb" to help guide future design decisions.

It is important not to over-design models or prototypes. These tools should be used to estimate resource utilization or iron out design issues. Once this is done, the models and prototypes should be considered disposable.

Effective Project Tracking and Controlling

Large software projects require accurate and detailed tracking systems to provide insight into development progress. Tracking systems become even more indispensable for large developments, because several software parts, at various life cycle stages, typically coexist. Using the tracking system, managers can always ascertain current performance against schedule and project personnel always have access to the latest development data.

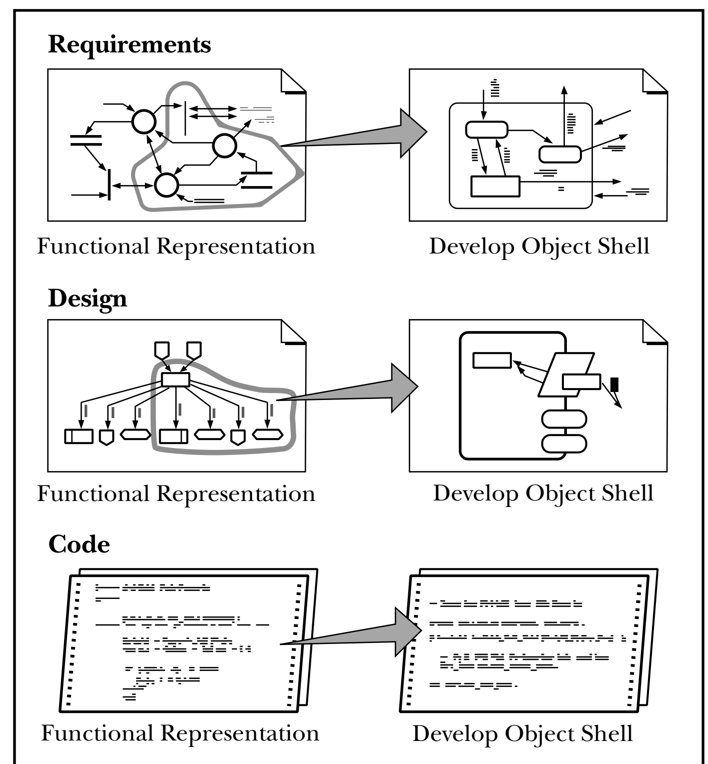


Figure 9. Using the shell approach to create objects from legacy systems

Deciding what and how much information to track depends upon how the system will be used. A system used solely to chart progress will contain less information than one used to plan future work. A system used for management overview may contain significantly less data that one used for future cost estima-

tion. It is more efficient to decide what we want out of a system before we decide what to put in.

A tracking system identifies, estimates, allocates, and tracks the software objects, or "work-packages." In the development sense, these work-packages are "objects" and "classes" in most OO techniques. In addition to object and classes, the work-packages are also called "blocks" in Jacobson [16], and "modules" in OMT [17]. A work-package can consist of one or many work-packages. We think of work-packages as individual orders for which we contract and measure work. Work-packages exist for requirements, design, code, and testing/integration. We identify sufficient information with each assignment to allow an external contractor, knowledgeable about the specific application and processes, to execute the assignment.

The typical work-package allocates about one person-week of work. We found that this level of detail is sufficient to adequately estimate the work without being unmanageable.

After the system's engineering organization creates and estimates the initial system work-packages. These typically identify several subsystems' requirements to define and prototypes to implement. This initial set of work-packages recursively spawns additional work-packages to complete the system. For example, design, code, test, and integration work-packages are created for sufficiently decomposed software objects.

Additionally, we estimate the required effort and resource utilization for each new work-package. Required effort is usually estimated in the form of person-hours or lines of code. Resource utilization estimates the required access to development facilities, such as unique test stations. The tracking system maintains these estimates to help establish and maintain realistic system development and integration schedules.

The work-packages must be closed after satisfactory completion. The process mandates the authenticating agent, such as SQA or system engineering, to close the work-package. Closure typically involves a formal review of the work, documentation of actual resources used or created (such as lines of code), and collection of appropriate quality metrics. The closure of work-packages earns (accumulates) value for the project.

The developers, managers, and customers use the system to quickly view project status. Developers can identify what new work is available and how their completed pieces fit into the big picture. Managers can review the recently completed work, compare progress to schedule, and compare actual to estimated metrics. The customer, with limited access to the system, can review the up-to-date schedule and quality metrics.

Consider the following ideas before implementing the tracking system.

- Incorporate tracking system updates directly into the existing software development processes. The software development processes should specify what, when, and how to update. Complete compliance is imperative for tracking system usefulness.
- Make the tracking system readily available and easy to use. People naturally resist tedious systems, and resistance is the enemy of compliance. It is worth spending time up front to ensure that the tracking system is easy and helpful. Automation of key features, including extensive use of default or boilerplate data, and template data is always appreciated. Time invested in tracking system simplification is always well spent.
- Establish trust in the tracking system. The faith that is placed in the system must be visually obvious and unwavering. Set high quality standards and expect no less. The development team must clearly understand the importance of the tracking system and the potential results of neglect. This attitude will become contagious and spread through the group. Ensure the system works—and trust it.

Defining and Documenting the Development Process

Defining and documenting the development process is an important step. The processes that tailor the OO technique to the project must be defined. The processes transform the theories of the textbook into the real world—our world. Documented processes enable the development team to consistently apply and benefit from the new OO technique.

The customer may want progress reviews, extensive documentation, and bidirectional traceability from requirements models to code. The OO techniques we used did not address these topics. To address such topics, we need a detailed, documented process that can consistently apply the features of the OO technique to the target system and satisfy the customer's demands.

We recommend appointing a team member to interact with the company's process improvement group to document exactly how to apply the OO technique to the target system. The team member should attend the process group meeting on a regular basis and work with them to determine what, how, and when to document. Forming a project's process team must be avoided because of process paralysis. Process paralysis, as defined by Yourdon, is what happens when the project team becomes thoroughly overwhelmed by the new technology and gradually ends up spending all of its time (1) trying to understand the new technology, (2) arguing about its merits, or (3) trying to make it work [18]. We must watch for this paralysis and get our project team on track with the project objectives and goals by using a specialized software process improvement team outside the project.

The process' definition concentrates on what and when software products are required. For example, what level of documentation is required for a preliminary design review? Which graphics are incorporated into our requirements specification? What are the exit criteria for object-level testing? We found that by precisely defining the software products required for each development phase, each developer can maximize individual contributions and still maintain system consistency.

The software process must be defined with sufficient detail that any competent developer outside the scope

of the current project could correctly answer the question, “What’s next?” Specifically, the process defines the intent, techniques, entry and exit criteria, and appropriate quality standards for each step of development. Figure 10 shows appropriate process detail.

There is a common misconception that we cannot document techniques, such as software development, that involve creativity. We would argue, as may our customer, that without defined processes the development team cannot consistently apply any development approach. Indeed, the Software Engineering Institute strongly advocates this position [15]. Without the process, developers freely apply their version of software development. This approach becomes especially risky when implementing a new OO technique. Core development processes should exist before starting a project, and should be continuously tuned as the program matures.

Collecting Software Metrics

While documented processes provide the framework for improvements, metrics provide the ruler that measures success or failure. Carefully selecting measures indicating application of the process will help to prove a growing proficiency. The collection of valuable software metrics should be permanently ingrained in each software process.

Collecting software metrics is easy; collecting useful software metrics requires effort. What makes them meaningful? Consider the following suggestions: Gather metrics only in support of specific, predetermined objectives, and evaluate the measures in that context. Figure 11 illustrates software metrics collection principles. Measures of planned versus actual class or object reuse, or of cost of designing for reuse, help guide follow-on projects. Measures of object quality, such as coupling or cohesion between classes or objects, provide insight on how well developers are applying OO technique concepts.

Before measuring any process, rank the metrics based on the potential value they add. The act of measuring human processes is personal and disruptive. The risks range from offending the developers to erroneously invalidating long-standing values. Also, the act of measuring some processes alters what we are measuring. With all of this at stake, we make sure that we collect metrics only for high-value, repeatable processes. Collect metrics on processes, not people. Figure 12 shows some useful software management metrics that were collected on several projects.

We conclude that collecting measurements is important for software development tracking and control. However, there are no universally accepted measurements for OO. Jacobson [16], Chen and Lu [2], and Chidamber and Kemerer [3] suggest well-known and accepted measurements. They could be used as the starting point.

Jacobson [16], treats metrics in a very general fashion

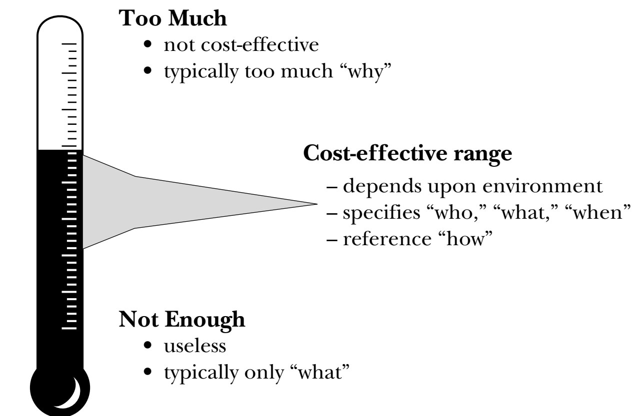


Figure 10. Appropriate process detail

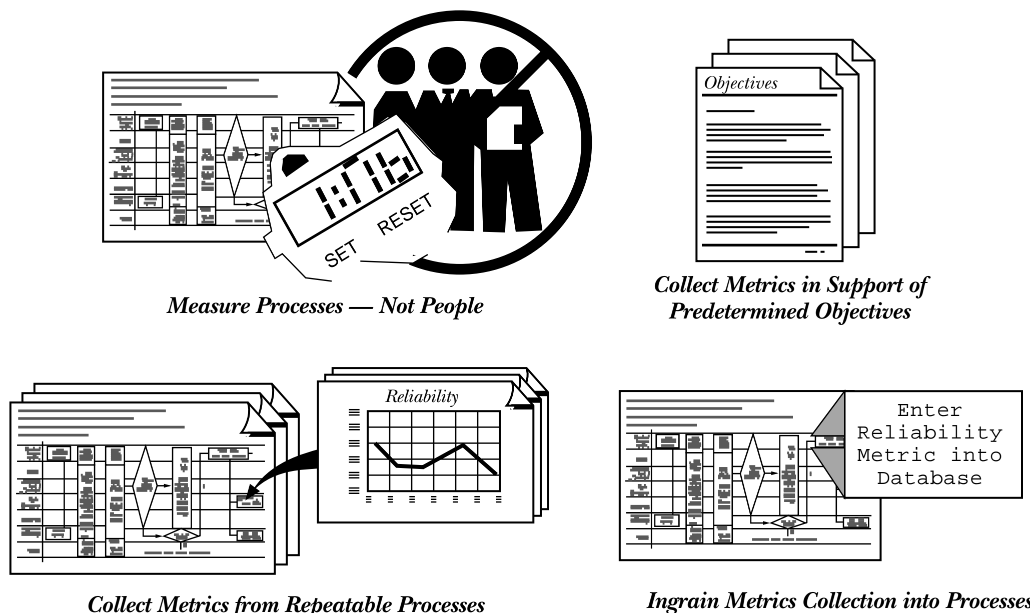
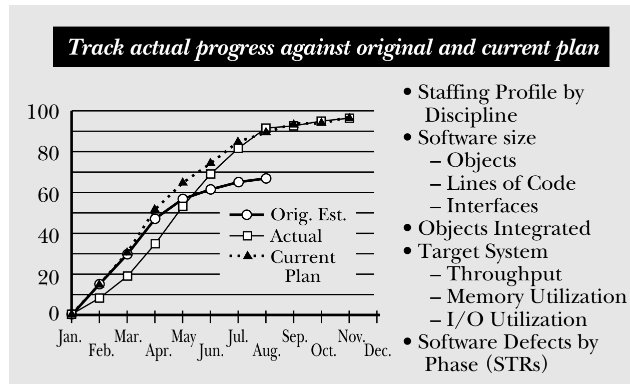


Figure 11. Software metrics collection principles [7]

Figure 12. Useful software management metrics [7]



and divides them into two categories: (1) process-related metrics, such as total development time, development time in each process and subprocess, and time spent modifying models from previous process, and (2) product-related metrics, such as total number of classes, of width and height of inheritance hierarchies, average number of operations in a class, and length of operations (in statement). Chen and Lu [2] present eight metrics for OO design. Chidamber and Kemerer's work [3] is most frequently quoted; they propose a set of six metrics: Weighted methods per class, depth of inheritance tree, number of children, coupling between objects, response of a class, and lack of cohesion in methods.

Inspecting Object-Oriented Software Products

Software inspections substantially improve product quality and help to keep new OO development on track by reducing defects and costs [5]. As with anything new, the application of specific concepts may not be fully understood. Even concepts that are well understood may allow several implementations, all equally correct.

Documentation is a good example. Those who have attempted to integrate several developers' writing into a coherent document are well aware of the difficulties. Without formal interim reviews, the final product will resemble a United Nations meeting—several seemingly unrelated individual elements trying to work together.

Approach software inspections as a rigorous form of peer review or walkthrough. As such, inspections should formally compare developed software to established standards. Standards vary with the project, but inspections will ensure that the development team follows the appropriate standard.

Thoroughly reviewing all software on a large project is time-consuming. Because of this, inspection processes should be optimized by defining specific, complementary roles for individual reviewers. For example, one reviewer may ensure all requirements are sufficiently allocated in the design, while another may verify a unit's testability. The number and specific responsibilities of the reviewers must be formally documented for each type of material reviewed. We have found that inspection checklists are a good way to ensure that several independent reviewers cover all the bases.

Some software teams feel they can not afford inspections. We contend that inspections are the most cost-effective technique for ensuring software quality. Removing defects at their point of insertion significantly reduces the cost of their repair. Efficient inspections are always a wise investment.

We make some final suggestions about inspections:

- **Review products for specific qualities of object orientation.** These qualities range from consistent abstraction levels for generic objects and methods to appropriate utilization of inheritance. Additionally, potential for reuse may be scored and documented. These reviews will help reinforce OO concepts by revealing several applications in the project domain. Applying these theories is the best continuing education available.
- **Develop quantifiable metrics to ensure adherence to specific technique.** These measurements should concentrate on the specifics of the application of the technique itself. They may include definition of an object, correct utilization of graphics, and other documentation.
- **Inspect the quality of documentation, including graphics.** Standards should be developed to define documentation quality. These standards are often overlooked in software development, especially when CASE tools are introduced as documentation aids. We've found that while good graphics increase maintainability, poor graphics detract. Remember that the customer's main perception of system development quality is through the documentation.

Integrating Software Documentation

Documentation may be the most visible sign of software quality. It exposes not only the construction but the logic behind the physical connections. The documentation is usually the only link maintenance personnel have to the original designers. It is imperative that this link be strong.

Mapping an OO technique to documentation standards can be difficult. The problem is compounded with extensive use of graphics. This difficulty is usually traced to understanding the term "standard" as implying "applicable to all developments except mine." What is worse, OO developments were probably uncommon when the documentation standard originated. Undoubtedly, the documentation standard will need to be revised for compatibility with an OO development technique.

We have found that the best approach is to tailor the documentation standard to the specific technique before the development begins. This includes translating all the terms given to the development phases, concepts, and deliverable pieces. It is important to define and to maintain the focus on the real purpose of the documentation. Poor documentation is worse than none at all.

Technique-specific graphics (possibly generated by

CASE tools) should be mapped into specific documentation sections. This step may prove surprisingly difficult to achieve. The mapping requires several revisions; however, the results are well worth the effort.

If customer approval is required for documentation decisions, include the customer in the early discussions. The customer, who will often be the ultimate system maintenance agent, may have strong feelings on specific documentation issues. Simple agreements early in a program may avert many later headaches.

One final note about documentation. In many cases, documentation revisions are more common than software revisions. Without software configuration management, documentation integrity is impossible.

Using Software Configuration Management

Configuration management is the process of identifying and defining the configuration items in a system (e.g., class diagram, design document, object test case), controlling the release and change of these items throughout the system life cycle, recording and reporting the status of configuration items and change requests, and verifying the completeness and correctness of configuration items [1]. A large number of configuration items (e.g., design artifacts and products) are produced by an OO technique. These configuration items, such as requirements, class diagrams, object-interaction diagrams, object diagrams, object-hierarchy diagrams, process diagrams, object/class design, code, and test cases must be placed under software configuration management. During development, updating, and reuse, these products are revised several times and must be placed under version control. This intensifies the need for a powerful yet simple software configuration management and version control system.

Conclusions

We have shown that a successful transition to OO software development techniques requires the use of management processes designed for the paradigm. The project must be preceded by an effective software development planning process that includes OO technique considerations—A critical part of these early activities is changing the culture.

We have concluded that the OO insertion stage of the transition defines the development environment and allocates the required resources. The critical activities during this stage include defining techniques and tools and selecting and training the development team.

We recommend that managers make every process and tool used during development their own. This requires that processes be customized for the current project and that tools be used in the most effective manner for the current project. Managers should not be frightened away from good concepts because some documented implementation seems expensive. They should instead study what the concept means in the context of their project and adapt it to their own purposes. Good concepts and techniques do not have to be forced on people who want to maximize effectiveness. If concepts and techniques are truly good, developers will fight to be able to use them.

Management of software projects can proceed effectively when good management techniques are applied. We have shown that the best of software management techniques can be effectively applied to OO development projects.

References

1. Bersoff, E.H. Elements of software configuration management. *IEEE Trans. Softw. Eng. SE-10*, 1 (Jan. 1984), 79–87.
2. Chen, J.Y. and Lu J.F. A new metric for object-oriented design. *Information and Software Technology* (Apr. 1993), 232–240.
3. Chidamber, S.R. and Kemerer, C.F. Toward a metrics suite for object-oriented design. In *Proceedings of OOPSLA '91* (1991).
4. Colbert, E. The object-oriented software development method: A practical approach to object-oriented development. In *Proceedings of ACM TRI-Ada '89* (Oct. 1989), pp. 400–415.
5. Fagan, M.E. Advances in software inspections. *IEEE Trans. Softw. Eng. SE-12*, 7 (July 1986), 744–751.
6. Fayad, M. E. *Managing Object-Oriented SoftwareDevelopment Projects.* Three-day seminar, Technology Training Corporation (TTC), Toronto/Washington DC, April/May 1993.
7. Fayad, M.E. *Managing Object-Oriented Software Development Projects.* Full-day tutorial, ECOOP '93. July 1993, Kaiserslauter, Germany
8. Fayad, M.E. Object-Oriented Experiences. In *Proceedings of the TOOLS USA Conference '93.* (Santa Barbara, Calif., Aug. 1993).
9. Fayad, M.E. *Software Development Plan for Object-Oriented Projects.* IEEE Computer Society Press, March 1996.
10. Fayad, M.E. Object-oriented software engineering: Problems and perspectives. Ph.D. dissertation, June 1994.
11. Fayad, M.E., et al. Using the Shlaer-Mellor object-oriented method. *IEEE Softw.* (Mar. 1993).
12. Fayad, M.E., et al. Adapting an object-oriented development method. *IEEE Softw.* (May 1994).
13. Fayad, M.E., et al. Object Modeling Technique (OMT): Experience report. *J. OO Programming (JOOP)* (Nov.–Dec. 1994).
14. Fayad, M.E. and Fulghum, M. *Object-Oriented Experiences.* SIGS Books, New York, 1996.
15. Humphrey, W. *Managing the Software Process.* Addison-Wesley, Reading, Mass., 1989.
16. Jacobson, I., et al. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Reading, Mass., 1992.
17. Rumbaugh J., et al. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, N.J., 1991.
18. Selic, B., et al. *Real-Time Object-Oriented Modeling.* Wiley, New York, 1994.
19. Shlaer, S., and Mellor, S.J. *Object Lifecycles: Modeling the World in States.* Yourdon Press, Englewood Cliffs, N.J., 1992.
20. Yourdon E. *A Game Plan for Technology Transfer. Tutorial: Software Engineering Project Management.* R.H. Thayer, Ed. Computer Society Press, 1987, 214–217.

About the Authors:

MOHAMED E. FAYAD is an associate professor of computer science at the University of Nevada, Reno. **Author's Present Address:** Computer Science/171, University of Nevada, Reno, NV 89557; email: fayad@cs.unr.edu

WEI-TEK TSAI is a professor of computer science at the University of Minnesota. **Author's Present Address:** Computer Science Department, University of Minnesota, Minneapolis, MN 55455; email: tsai@cs.umn.edu

MILTON L. FULGHUM is a senior staff engineer at FlightSafety International in St. Louis, MO. **Author's Present Address:** Flight Safety International, St. Louis, MO 63042; email: fulghum@vss.fsi.com

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© ACM 0002-0782/96/0200 $3.50