Although OO development methods and models have been in use for several years, management techniques optimized for OO development have not kept pace. The author participated in several OO projects and, based on those experiences and the lessons learned from them, developed a management process tailored to OO development.

MANAGING OO PROJECTS BETTER

Paolo Nesi, University of Florence

Over the past decade, the object-oriented paradigm has gained a large following, and in many cases[1] has replaced traditional software development approaches with OO equivalents.[2,3] New schemes have been proposed for modeling the development life cycle as well.[4] Unfortunately, many of these approaches focus on modeling the system development around a single task or team, rather than considering how to build a specifically object-oriented system in terms of planning, team structure, and project management.

Thus, the definition of these new OO development methodologies has yet to be supported by a comparable effort in defining methods and strategies for managing OO projects, or for modeling the life cycle at both the system and the task and subtask levels. OO project management also requires integration in effort planning and process prediction. Traditional models fall short in this regard, too, because they take an almost entirely bottom-up approach and base their project plans on the mechanisms of allocating and deallocating people.

0740-7459/98/$10.00 © 1998 IEEE

Table 1
Significant Statistics For Studied Projects

Project	OS	Language	Tools and Libraries	Number of Classes	SC_{LOC}	Person-Months	People	Teams
TOOMS	Unix	C++	Lex/Yacc, CommonView	204	16568	41.5	16	6
ICOOMM	Windows NT	C++	MFC	193	10870	20	6	3
QV	Unix	C++	XLIB, Motif	65	3900	7	4	2
LIOO	Linux	C++	Lex/Yacc, XVGAlib	165	16020	30	11	5
TAC++	Unix	C, C++	Lex/Yacc, QV, XLIB	62	2300 for C, 4340 for C++	13.5	5	2

As a first step toward alleviating these shortcomings, I outline several lessons learned while deriving a more efficient model for managing OO projects. Summaries of these lessons, which represent the experiences of myself and my colleagues in managing several small- and medium-size OO projects over the last seven years, appear in italics throughout the sections that follow.

Project Profiles

Table 1 shows the profiles, in chronological order, of some of the projects from which I drew the lessons described in this article:

- TOOMS (Tool Object-Oriented Machine State), a CASE tool to specify, verify, test, and assess real-time systems;
- ICOOMM (Industrial Control Object-Oriented Milling Machines), a computerized numerical control for milling machines;
- QV (Q View), a library providing uniform OO support for Motif and X;
- LIOO (Lectern Interactive Object-Oriented), a lectern and editor for music scores; and
- TAC++ (Tool for Analysis of C++), a tool for developing and assessing C/C++ projects.

Each profile lists the project's pertinent data, including the number of system classes, SC_{LOC}, effort in person-months, number of non-project-management people involved, and number of different teams. SC_{LOC}, or OO system complexity based on lines of code, is an evolution of the LOC metric that also considers class attributes, class definition, and method interfaces.[5]

Most of these projects were carried out by heterogeneous teams that included staff members from the University of Florence, the Centro Sviluppo Tecnologicao (CESVIT) research center, and various companies. Although the project partners were in separate locations, to improve the homogeneity of results the various heterogeneous task and subtask teams assigned to a particular project worked in one place. Doing so let them use the same "quality manual," a reference document containing all guidelines for project development as prescribed by the company's general criteria. The teams performed most of the work using C++, but implemented some projects in TROL (Tempo Reale Object-oriented Language), a formal language and model similar to object-oriented SDL, ObjecTime, and others.[1]

Team Structure And Organization

A project manager coordinates the subsystem managers and directs the overall project. When the project manager adopts the well-known waterfall life cycle, a clear division is present between the phases of analysis (requirement analysis and detailed analysis), design (structural design and detailed design), coding (class implementation), test, and so on. The team organization usually replicates this division. Different groups work on the same project in different phases, and communicate via a few meetings and documents. These groups may use different notations and methods. Even if an integrated quality and methodological model unifies these notations, misunderstandings may still occur between the many people tasked with drafting and interpreting the project's documents. Thus a mis-

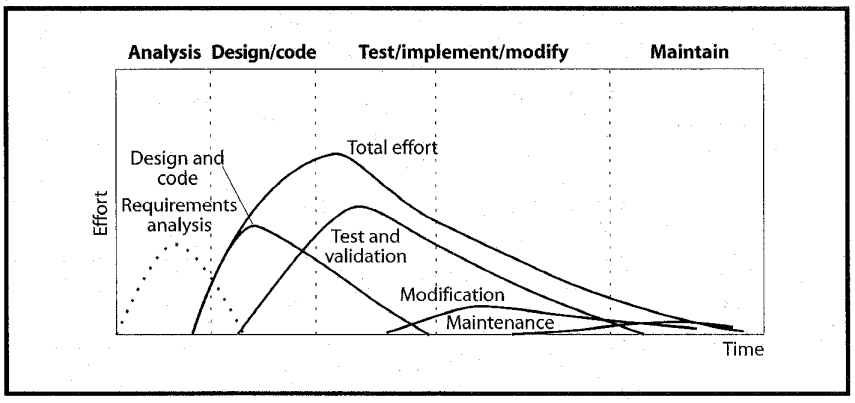

FIGURE 1. Putnam's resource allocation model, which shows project effort as a function of time.

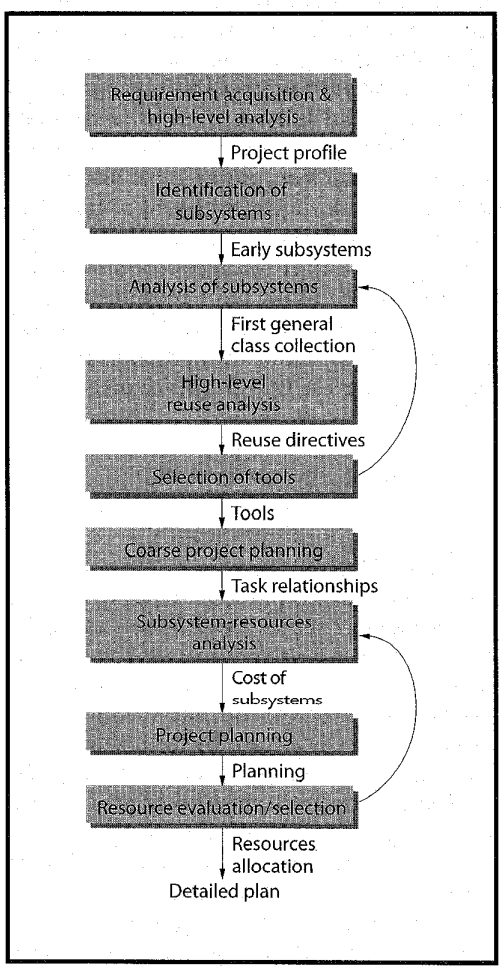

FIGURE 2. Project start-up structure with the most important products of each phase.

match between consecutive phases may develop,[6] which causes teams to work on wrong or unrequired functionalities. Although several integrated approaches to reduce these problems have been tried, their results have been limited by the persistence of different interpretations and viewpoints within groups and between groups.

In many cases, project managers adopt Larry Putnam's resource allocation model,[7] shown in Figure 1. As the project evolves, the dynamic allocation of people may lead to an increase in the number of subsystems and in the team's size. Typically, management allocates people dynamically, passing them from one phase to the next to satisfy incoming deadlines. Although training new people would avoid this reshuffling, it is often a costly and self-defeating alternative because it usually entails the support of skilled people needed elsewhere on the project.

♦ *In our experience, the hierarchical organization of a team* can *be successfully applied to OO projects. But the project plan, the development life cycle, and the roles of the project manager and subsystem managers must be revised as described in the following lessons.*

PROJECT START-UP

As Figure 2 shows, the project start-up consists of several phases typically performed sequentially or only partially overlapped.

We typically perform the requirements acquisition and high-level analysis, also called user needs analysis and feasibility study, to

♦ evaluate commercial issues such as benefits and risks;

♦ analyze the technological risks;

♦ examine the requirements with respect to industry trends and general goals;

♦ define ultimate targets in terms of end users, environments, platforms, and so on;

♦ define the quality profile of the final product; and

♦ define project timing, the most important milestones, and the critical path.

The project manager and subsystem managers identify the main subsystems according to a structural decomposition. The subsystems are, in most cases, comprised of multiple classes, many of which are used by other subsystems. This occurs because in classical OO analysis methodologies, you usually model the problem domain and not a specific system or subsystem.[2]

Typically at this stage, you strive to model exactly that part of the domain needed for the system you're building, and no more, unless reuse considerations prompt you to undertake extra work. On the other hand, during project start-up you usually have limited knowledge of the problem's scope, and thus it is best to manage as much of the problem domain as you can. A too-restrictive modeling performed during the early phases of system development typically results in hierarchies having too many complex leaf classes. The presence of too many leaf classes is frequently due to the lack of specialization. In turn, the lack of specialization is frequently due to a poor analysis of the system domain. This occurs, for example, when the application's problem domain has been neglected in favor of a limited analysis of the application under development. All applications start small and grow throughout their service life. To start with a restricted analysis frequently leads to an insufficient class hierarchy. During the analysis the focus should be on those classes used in several tasks. These are usually the most important ones for the domain under analysis or are fundamental to the system. They include repositors, model symbols, graphic components, and so on.

Lack of specialization is frequently due to poor analysis of the system domain.

In the first part of subsystem analysis, each subsystem manager must identify the most important classes of the assigned subsystems. Then, classes identified by all subsystem managers are organized in the unique domain model. In OO projects, a subsystem can be

- a subtree or tree,
- a number of independent classes, or
- a classical subsystem, identified by a structural composition-decomposition process.

♦ *We re-extracted the main subsystems from the general collection of classes by identifying branches related to the most important classes, usually called key classes. We reassigned these new versions of subsystems to subsystem managers. The two-phase process I describe regularizes the identification of subsystems, improving efficiency by avoiding class duplication and reducing the dependencies among subsystems (and thus among subsystem teams). Each subsystem or subtask should have from 15 to 30 classes to be manageable, depending on the role these classes play in the system. A larger number usually means that team members must learn a lot about the system, while too few classes may lead to people working on the same class too frequently. A subsystem can include*

♦ *several small classes (basic objects, part of other classes);*

♦ *a number of so-called "key classes" (such as the root class for persistent objects, a class implementing the list) and,*

♦ *among these key classes, a few very important classes, called engine classes. These are more complex than the other key classes and cannot always be decomposed because their complexity derives mainly from their functional and behavioral aspects.*

Instances of engine classes usually control the most important parts of the application, such as the database manager, the state machine editor, the window manager, the event manager, or the interpreter. For such classes, the effort—defined as the hours spent executing a particular task—may be as much as five times greater than that for normal key classes. The start-up phase continues with reuse analysis, in which the project manager and the subsystem managers identify sources of existing subsystem parts, classes, or clusters that can be profitably reused in the current project. For each potential reuse source, the cost of adaptation must be evaluated before proceeding.

Next, the subsystem managers and project manager identify suitable tools for system development. These decisions may lead to reiterating from the subsystems analysis step.

The subsequent coarse-grained project plan includes

1. analyzing the overall system,
2. defining subsystems (tasks) and subtasks,
3. examining dependencies among tasks,
4. planning time-to-market, and
5. identifying milestones and other target dates.

Determining how many staff-hours are needed to develop each subsystem and its corresponding subtasks cannot be done yet because the assess-

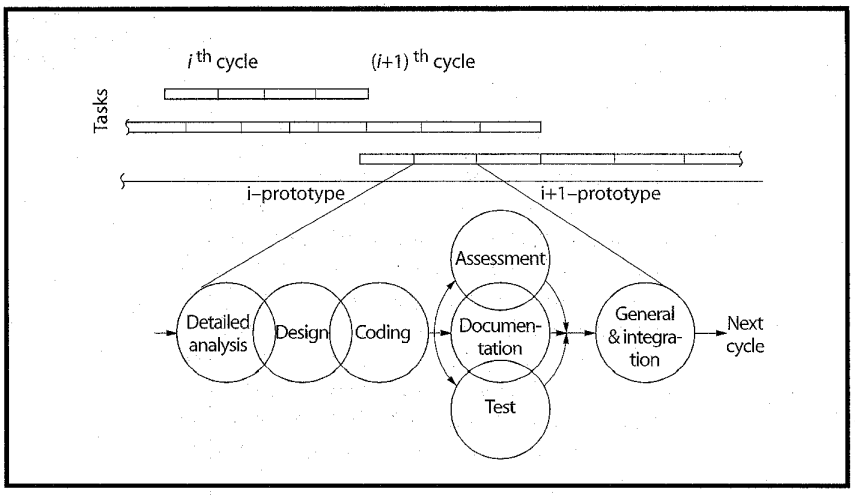

FIGURE 3. The macrocycle of the task and subtask development life cycle, and the microcycle—a simplified version of the spiral model integrated with a modified version of the fountain model.

ment effort must be performed later. During the subsystsem-resource analysis, the effort for developing each subsystem and the corresponding subtasks is evaluated in terms of human resources.

♦ *The number of identified key classes multiplied by a factor, K, gives an approximate measure of the final dimension of each system or subsystem in terms of classes. We found that K is equal to 2 for subsystems without a user interface and communication with devices, and 4 for subsystems that include the relationships with a complex user interface. We use the hypothesis for the number of system classes to predict the effort needed for each task by considering the typical person-hours needed to analyze, design, and implement a class. In the projects mentioned, this mean factor is 15 to 40 hours per class; differences depend on team efficiency and application complexity. We used the number of class attributes and methods as a more precise predictive measure of class complexity.*

You can now prepare the detailed project plan, considering at least the temporal constraints identified in the requirements acquisition and high-level analysis phases, as well as the number of people needed for each subsystem and subtask. The plan must also consider the nature of the life cycle adopted. In the case of OOP, the classic life cycles are evolutionary and typically prototype-oriented, such as the spiral or fountain models.[4,6] The fountain model leads to an unpredictable number of cycles at the system level because it is too rule-free to produce repeatable results. This makes it difficult to predict the duration of the analysis, design, implementation, and other phases.

♦ *At the system level, we used a spiral-oriented project schedule: two or three cycles of from six to eight months each that produce a milestone and prototype for progress control. Each cycle can involve several tasks and includes, at the system level, general assessment and risk analysis.*

Working from the project plan, management selects and allocates project staff, based on the skill and experience of available personnel. In our teams, the typical efficiency was from 2.2 to 4 SC_{LOC} points per hour, including analysis, design, coding, documentation, test, and assessment phases.

SYSTEM DEVELOPMENT

After project start-up, the system development phase begins by activating tasks and subtasks according to the project plan and the preanalysis performed during start-up.

Task life cycle

The spiral model is too complex and complete to be applied at the subsystem level, while the fountain model is difficult to control. The adopted microcycle, shown in Figure 3, consists of a simplified version of the spiral model integrated with a modified version of the fountain model. According to the fountain model, the first three steps can be locally iterated by restarting from the first step when strictly necessary to achieve the cycle objectives. Unlike the fountain model, the last steps are performed simultaneously and only once.

Figure 3 shows that, as in the spiral model, a task or subtask is typically completed in one or more cycles, each with a duration of two to four weeks depending on fixed subgoals and milestones. Given the team members' typical productivity, you can obtain the number of cycles required by considering that a task or subtask must contain from 15 to 30 classes to be manageable. For example, it requires three cycles of two weeks each, with a two-person team—a total of 480 hours—to produce from 12 to 32 classes for C++. Less time will be needed if you are producing key classes, because fewer classes must be considered if the subsystem includes key classes. Task and subtask teams consist of two to three people, including the subsystem manager. The develop-

ment process is a sequence of partially overlapping steps. We found it advantageous to use an analysis and design methodology very similar to Grady Booch's,[2] because it meshes well with the management and lifecycle models we selected. After the design phase, the same team performs testing, documenting, and assessment simultaneously. Because consecutive microcycles partially overlap, team members have different roles in different contexts.

♦ *We observed that the effort spent in documenting and assessing depends quite linearly on the number of classes, while testing depends on time for testing classes and their relationships. The first factor is linear and the second takes a time that depends more than quadratically on the number of classes, since interactions among classes exploiting relationships of is-part-of and is-referred-by are frequently made concrete with several method calls. A small number of classes per subsystem lets you work in the first part of the cost evolution curve, where the cost of testing relationships is much lower than that for testing class relationships. This also depends on the number of relationships established among classes (another parameter that must be maintained under control along the development process). Moreover, we reduced testing, assessing, and documenting time by improving these processes.*

Table 2
Imposed Limits for Class Parameters

Parameter	Mean Values	Maximum Values
Class complexity	200	1800
Class complexity, inherited	150	1200
Class complexity, local	50	600
Number of class attributes	9, 27	15, 45
Number of inherited class attributes	6, 18	10, 30
Number of local class attributes	3, 9	5, 15
Number of class methods	36, 90	44, 144
Number of inherited class methods	24, 60	36, 96
Number of local class methods	12, 30	18, 48
Number of superclasses	2	5, 6
Number of subclasses	5	30, 90

Test activity

Typically, the test activity can last anywhere from as long as the first three phases combined to less time than it takes to complete the first phase. We reduced the time for testing by preparing test scripts of test cases and procedures directly in the analysis phases and, in some cases, by using an automatic tool for regression testing based on Capture and Playback.[8] This approach is based on two distinct phases: the capture and the playback. During capture, the system collects each computer-user and external-device interaction. The histories of these interactions are stored in sequential form in a script file. The histories are reproposed during playback to the computer interfaces, simulating the presence of real entities: users, other machines, sensors, and so on. After each simulated stimulus, the computer's responses can be tested to verify that the application answers correctly according to its predetermined behavior.

In subsequent cycles, regression testing is performed automatically for those parts that are mainly unchanged.

Documentation activity

Documentation usually takes much more time than other activities. Suitable CASE tools for analysis and design can generate a draft version of documentation, in which details related to the implementation must be added manually.

♦ *The team member who serves as main designer of the classes is the best choice for performing this work. The subsystem manager also must help prepare the documentation that describes the task status and evolution, and the decisions carried out in the task cycle. This part of the documentation helps the project manager understand and discuss the project's status at a higher level.*

Assessment activity and related mechanisms

Evolutionary development lets you produce something that can be automatically measured right from the early phases of the project, when class definitions are available. We used the class structure's attributes and method interfaces to apply predictive metrics for estimating development, maintenance, and other task effort.[5,9] We analyzed values and corresponding trends of a few metrics and indicators for each task and subtask class: complexity (as it relates to class effort, maintainability, and so on), verifiability index, reuse index, efficiency, and so on.[10,11]

As Table 2 shows, for classes we imposed specific profiles, such as number of attributes, number of methods, class complexity, inherited class complexity, and class interface complexity. When two values appear in a cell, they refer to classes not involved and involved in the GUI, respectively. Data for these met-

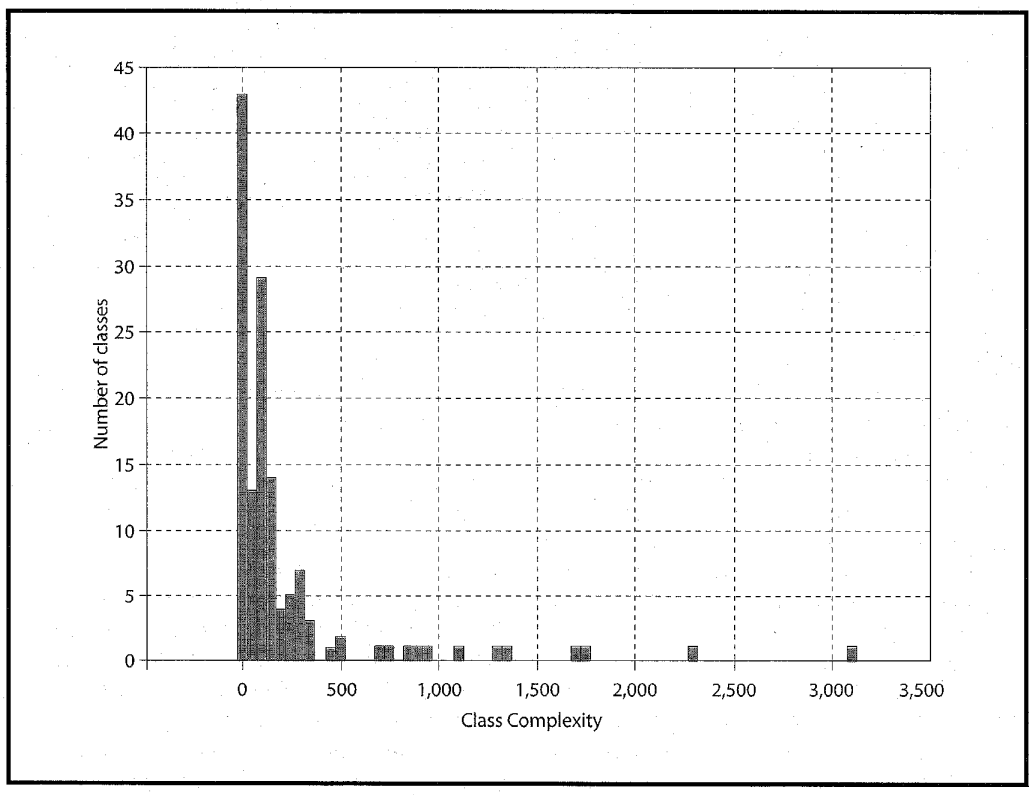

FIGURE 4. Histogram of the class complexity attribute for project LIOO near the midpoint of the process's development. The typical histogram must present an exponential trend, while in this case a couple of classes exhibited a class complexity greater than 1800.

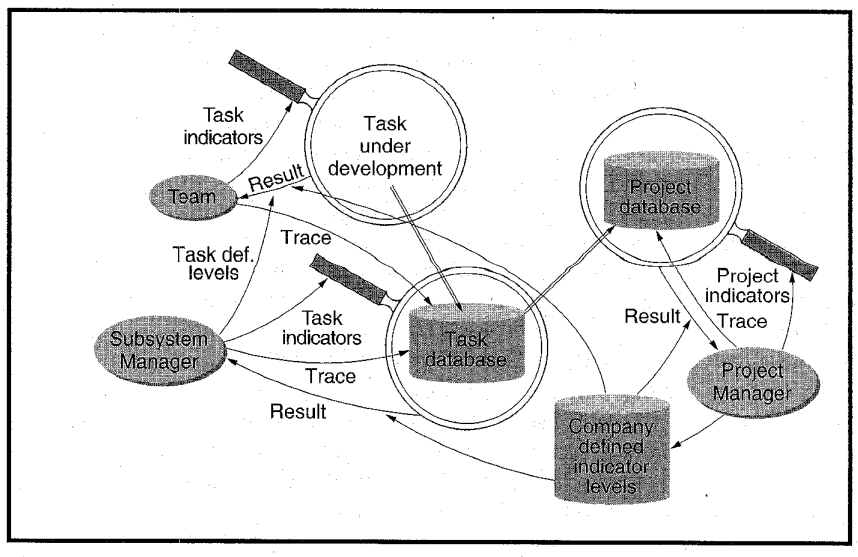

FIGURE 5. Task and project levels in a sample assessment. Only one task is reported.

rics has been evaluated using a definition developed in my work with colleagues.[5,11]

We define these profiles using the typical histograms obtained for that metric in other projects, as shown in Figure 4. When a class grows beyond its imposed limits it must be carefully analyzed and, if possible, corrected. We accomplish this by, for example, splitting the class or moving the code closer to its parents.[11] Doing so maintains the class's quality and ensures control of the staff effort devoted to the class.

Assessment, the shortest activity of those performed in parallel, is done by a team member skilled in OO analysis, with the

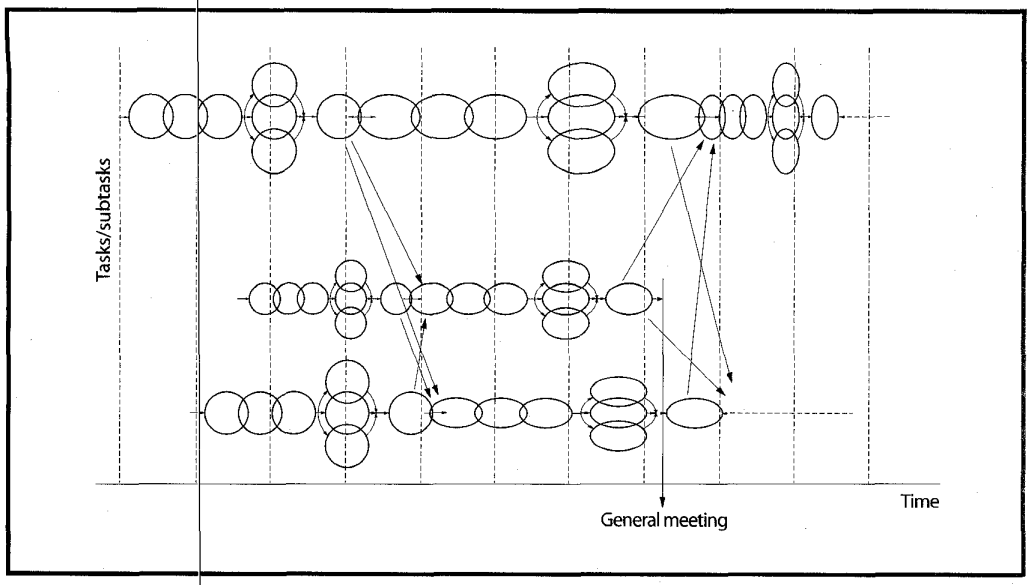

FIGURE 6. Relationships and integrations among project tasks and subtasks. The dotted lines indicate the intervals for periodic meetings to discuss integration among classes and clusters by different teams.

help of the subsystem manager. As Figure 5 shows, the assessment checks that the process satisfies company metrics and indicators for controlled software development. The subsystem managers must include in the documentation and the project database any strategies for correcting deviations from the plan.[5] For our assessments we used a suitable tool for measuring selected metrics and comparing the imposed profiles with the current ones.

♦ *Continuous metrication must be associated with a continuous revalidation of the indicators adopted to adjust weights and threshold values, and for tuning the organization's metric suite. We collected non-automatically measurable data by filling in an electronic questionnaire daily with, for example, the effort of each class (for detailed analysis, design, coding, and other tasks), the modification of class hierarchy, the effort for the other activities, and a brief description of the work. Real and measured values let us identify the model.*

Generalization and integration activity

This phase partially overlaps the next microcycle. To accomplish generalization and integration, the subsystem manager must first identify the detailed goals of the next microcycle. These goals may be best defined once the subsystem manager finishes the current phase and begins full-time work on the next microcycle.

♦ *During this phase each subsystem manager may meet with other subsystem managers and the project manager, depending on task relationships, to*

1. identify new detailed requirements for generalizing classes and clusters developed so that they can be used in the whole system;

2. provide other teams with the current version of software developed in the corresponding task or subtask, along with its documentation, test, and assessment reports;

3. discuss with the project manager how to correct problems identified by the assessment activity.

As regards point 2, the other task and subtask teams will use the results produced starting from their next detailed analysis phase, as shown by the arrows in Figure 6.

Task relationships

The dotted lines in Figure 6 show that, about every two to four weeks, depending on the length of the microcycle, project task and subtask teams hold periodic meetings to exchange information. In these meetings, only the choices made in the analysis and design phases are discussed, with the intent of improving integration among classes and clusters implemented by different teams. Thus, the periodic meetings address the general aspects of analysis and design, while restricted meetings between the team and its subsystem manager address

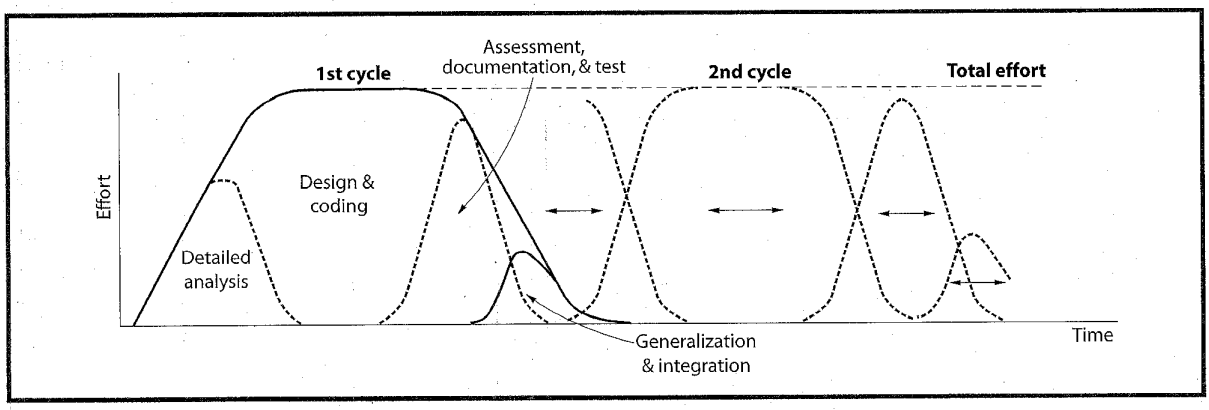

FIGURE 7. The model adopted for task and subtask effort as a function of time. The dashed line represents general effort.

more detailed and technical problems.

♦ *Periodic meetings avoid class duplication and facilitate the adoption of uniform notations in the project database's quality manual for the selection of class, method, and attribute names; for compiling documentation; for preparing test scripts; and so on. To improve control and uniformity, plan general meetings with all team members and subsystem managers*

♦ *when an important task is completed,*

♦ *when an important change to the project structure and management is needed, or*

♦ *for discussing prototypes of the whole system or time-consuming milestones and deliverables.*

At general meetings, everyone explains what they are doing and are going to do, then comments on issues raised by the others. Such meetings improve the code's uniformity and quality because they let all points of view be taken into account. They also have a strongly positive affect on morale and motivation. General meetings can even help reduce the fuzzy thinking of programmers who sometimes get distracted by a trivial task that could be safely neglected, even if doing so caused inconsequential errors in the finished product.

Team people

Traditional project and subsystem managers typically lack the training and expertise to manage OO projects. This shortcoming becomes more pronounced when using our approach, which requires management participation in several meetings that address technical problems.

♦ *As shown, the roles of the project manager and subsystem managers are quite different than in traditional projects where, during the development, they only supervise the work of all groups, defining documents' structure, instruments, and planning. According to our model, the project manager can profitably manage the project only if he or she directly knows how the problem domain has been covered in terms of classes and class relationships. Moreover, the subsystem manager must participate actively in the task and subtask development by analyzing details and designing and implementing specific parts, or parts related to other tasks.*

Effort planning

Models such as the Putnam Resource Allocation model, Jensen's model, and COCOMO are mainly suitable for single-team projects.[12] Although some versions of these models can be applied to multiteam projects, they entail considerable dynamic allocation of human resources during the design and coding phases at the task level, as Figure 1 shows. At the subsystem level, this means it is difficult to predict costs, especially if you use traditional development models. With OO methodologies, on the other hand, effort is shifted from the design phase to that of analysis. System development starts by using a bottom-up approach and then reverts to a top-down one.[4] Doing so moves the peak of the effort curve to the analysis phase, but the problems related to dynamic allocation persist because the traditional effort allocation method remains. This problem becomes more severe when you use the Putnam model for each cycle of the spiral development life cycle, because it calls for the allocation and deallocation phases to be performed at each cycle, with a consequent increase in overhead costs.

Using the alternative approach shown in Figure 7, we report the effort for a task or subtask as a func-

tion of time. I constructed this idealized diagram by observing what happens when our micro life cycle is applied across several OO projects. The graph includes the first cycle's detailed analysis phase, when human resources are allocated. The subsystem analysis starts with the allocation of the subsystem manager and continues until all other team members have been assigned. The design and coding phases follow the analysis and are performed by the same team, with a constant number of people. Each cycle overlaps the next, with team members constantly allocated to the task (represented by the dashed line in Figure 7), but performing different duties depending on the cycle.

♦ *Because no clear separation exists between the life-cycle phases of OO methodologies, allocating a constant number of people to the task team is consistent with the OO approach, letting the same people who perform the analysis work on all other phases. This leads to a reduction of effort and less risk of misunderstanding. The lack of clear separation stems mainly from the impossibility, in many cases, of separating the development phases—for example, whether to include or exclude object and class specialization and relationship identification from the analysis phase. All these relationships have a strong impact on cluster identification and on the general domain analysis. Some methodologies are more flexible while others are much too rigid in this regard. Further, during each single cycle there are periods in which one team member still works on the analysis while the others proceed to the other phases. The project manager must try to distribute the project staff's efforts uniformly throughout the project or at least along the single tasks. Uniformity of effort also guarantees consistent quality and efficiency, improves controllability and effort predictability, and avoids the extra training required by dynamic allocation.*

When the team needs more effort to perform a cycle, project management lengthens the phases as depicted in the second microcycle of Figure 7. This does not conflict with the schedule if the *total* effort needed to develop a task (represented as the area under the curve) remains the same. The effort predicted for each task, and that of the whole project, can be adjusted after each task cycle in relation to the trend of normalized indicators,[5,9] such as

♦ the increment of task classes per time unit,

♦ the increment of class complexity per time unit,

♦ the increment of task complexity per time unit,

♦ the relative difference between the number of classes and the predicted number, and

♦ the ratio between internal and external class complexity.

Development teams of more than four people result in decreasing productivity.

If project management predicts insufficient effort for the planned trend, more effort is allocated by, for example, increasing the next cycle's duration. If this is infeasible because of task deadlines or other factors, project management divides the task into two subtasks under a single subsystem manager. In this case, dynamic allocation takes place in the analysis phase.

Task division can, however, generate mismatches. To reduce such problems, project management can perform the real division into subtasks after the detailed analysis. In some cases, task division is infeasible or too expensive because it would affect too many related classes. In such cases, the team can be increased to at most four people, some minor classes can be reassigned to related subsystems, or both. Larger teams would result in decreasing productivity and increasing cohesion among subsystems.

♦ *Using our microcycle approach requires more effort from the project manager than that needed to manage traditional projects: we obtained a value of nearly 210 hours per year for a project of four person-years. This value must be scaled for larger projects, which contain a higher number of subtasks, and thus require additional meetings and greater technical involvement by the project manager.*

I have found traditional methods inadequate for managing OO projects, for several reasons:

♦ There is a sizeable gap between OO software development methodologies and diffuse managing approaches.

♦ The life cycles usually adopted focus too much on single-task projects and structured or functional methodologies.

♦ Managers lack prior experience in the adoption of OO indicators for controlling system development at both project and task levels.

♦ Organizations share a deeply ingrained tradi-

tion of allocating and deallocating human resources among different projects.

♦ Project and subsystem managers lack the technical expertise necessary to profitably manage OO projects. Thus, to be effective, the OO approach I've proposed must be introduced throughout the whole organization.

My project experiences and those of my colleagues have enabled us to create and fine-tune a stable OO management and development model that addresses these shortcomings. Our method is now being used by several organizations that manage internal and multipartner Esprit projects. On these projects, the method has predicted final effort with errors lower than 10 percent, satisfying the project organizations' early-defined quality and company requirements. We obtained these results by facilitating strong collaboration among team members and increasing gratification and motivation according to the lessons and guidelines I've described. Other, longer-term benefits have resulted from our approach as well. For example, some team members have exhibited a growing capability to manage projects after they participated in a project using our method, which distributes management tasks more evenly across the team hierarchy. ❖

Acknowledgments

I thank the following project managers and subsystem managers. For project TOOMS: U. Paternostro of the Department of Systems and Informatics; M. Traversi and M. Campanai of CESVIT; F. Fioravanti, M. Bruno, and C. Guidoccio of DSI. For project TAC++: A. Borri of CESVIT and T. Querci of DSI; S. Perlini. For project ICOOMM: M. Perfetti and F. Butera of ELEXA. For the QV/MOOVI project: T. Querci; L. Masini, M. Caciolli, and L. Fabiani. For project LIOO: F. Bellini, N. Baldini, S. Macchi, A. Mengoni, and A. Bennati. For the projects of series MICROTelephone: M. Traversi, G. Conedera, D. Angeli, C. Rogai, and M. Riformetti of OTE S.r.l. For project INDEX-DSP: M. Montanelli and P. Ticciati of SED S.r.l. A warm thanks to the many developers who have worked and are working on these and other projects with me.

References

1. G. Bucci, M. Campanai, and P. Nesi, "Tools for Specifying Real-Time Systems," *J. Real-Time Systems*, Mar. 1995, pp. 117-172.
2. G. Booch, *Object-Oriented Design with Applications*, Addison Wesley Longman, Reading, Mass., 1994.
3. R.J. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object Oriented Software*, Prentice Hall, Upper Saddle River, N.J., 1990.
4. B. Henderson-Sellers and J. M. Edwards, "The Object Oriented Systems Life Cycle," *Comm. ACM*, Sept. 1990, pp. 143-159.
5. P. Nesi and T. Querci, "Effort Estimation and Prediction of Object-Oriented Systems," *J. Systems and Software*, to appear, 1998.
6. B.W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Software*, Sept. 1988, pp. 61-72.
7. L. H. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimation Problem," *IEEE Trans. Software Eng.*, July 1978, pp. 345-361.
8. P. Nesi and A. Serra, "A Non-Invasive Object-Oriented Tool for Software Testing," *Software Quality J.*, Vol. 4, No. 3, 1995, pp. 155-174.
9. W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *J. Systems Software*, Vol. 23, No. 2, 1993, pp. 111-122.
10. P. Nesi and M. Campanai, "Metric Framework for Object-Oriented Real-Time Systems Specification Languages," *J. Systems and Software*, Vol. 34, No. 1, 1996, pp. 43-65.
11. F. Fioravanti, P. Nesi, and S. Perlini, "Assessment of System Evolution Through Characterization," *Proc. IEEE Int'l. Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., Apr. 1998, pp. 456-459.
12. S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings, Menlo Park, Calif., 1986.

About the Author

Paolo Nesi is an assistant professor of information technology and a researcher with the Department of Systems and Informatics of the University of Florence. His research interests include OO technology, real-time systems, quality, testing, formal languages, physical models, and parallel architectures. He holds the scientific responsibility for high-performance computer networking at CESVIT, a high-tech agency for technology transfer. He also belongs to the editorial board of the *Journal of Real-Time Imaging*.

Nesi received a Laurea in electronic engineering from the University of Florence and a PhD in electronics and informatics from the University of Padoa, Italy. He is a member of IEEE, the International Association for Pattern Recognition, Taboo (the Italian association for promoting object technologies), and AIIA (the Italian association on artificial intelligence).

Address questions about this article to Nesi at Department of Systems and Informatics, Faculty of Engineering, University of Florence, Via S. Marta 3, 50139 Florence, Italy; nesi@dsi.unifi.it; http://www.dsi.unifi.it.