Managing Object-Oriented Software Development

Project managers should not use the novelty of the object-oriented approach as an excuse to let software developers manage themselves. Even though project managers may be unfamiliar with the latest technical advances, they must stay involved in planning and controlling software development. Nonetheless, in many organizations developers have usurped management planning and control by convincing managers that OO is so different from other technologies that they can’t understand or control it. The long-term results of this “hyper-empowerment” are almost always unfortunate, ultimately threatening the business value of the OO approach.

The solution is not authoritarian control. Managers must control the project in a way that doesn’t destroy the qualities they want to develop. For example, if they want an application to be flexible in a certain way, they must allocate resources to engineer that flexibility. However, developers must not be given a blank check to add flexibility wherever they want. It should be added only where it will bring the most business value. Marketing and management issues determine this, not technical ones.

The fundamental aspects of the traditional software development approach are woven into the fabric of the organization’s culture. Yet the analogous fundamentals for the OO software development approach are new to many, and this novelty often cuts across the grain of the established cultural norms. Changing the cultural aspects is often much more difficult than helping software developers understand OO technology. Because of the need to adapt the culture to the approach, a wise manager participates in some form of OO training or education.

This issue provides part of that OO education for both managers and developers. We have distilled management practices and techniques from our experiences in more than 100 OO projects and explored new practices. We cover the many ways that OO complicates the project manager’s role in such areas as planning, staffing, training, scheduling, cost estimation, documentation, development, legacy systems, and metrics, as shown in Figure 1.

**TRANSITION PLAN**

As in most endeavors, thorough preparation and planning greatly increase the chances of success. Software project planning is a key process area in level 2 of the SEI’s Capability Maturity Model. The goal is to smooth the transition from chaotic development to an organized, rational approach to OO software development. A transition plan is a critical ingredient. Transitioning to OO development addresses several managerial challenges that are critical to success. The OO transition process encompasses four phases:
1. **Planning and Pre-Project**, which includes activities to condition the development environment, such as changing the culture.

2. **Technology Insertion**, which includes activities to identify and plan for the required OO resources, such as selecting the right OO technique and tools, staffing, training the team, budgeting for reuse, and dealing with legacy systems.

3. **Project Management**, which includes activities to monitor and direct the project, such as effective project tracking and controlling, collecting software metrics, defining and documenting the OO software-development process, inspections, and documentation.

4. **Post-Project and Maintenance**, which includes activities to provide maintenance and product support for the software, such as software configuration management.

This transition plan has been used effectively on new and existing OO development projects.

### Expectation management

Managing expectations is critical to applying OO technology effectively. Adopting a new paradigm creates the fear of "getting it all wrong." People are also uncertain about which process to use. Organizations often begin the transition to OO without any software engineering process to use as a foundation.

Because the transition will affect other activities, senior managers need to be aware of the expected impact so they can understand what is happening. It's especially important that the software manager inform senior management of the schedule and budgetary aspects of the transition.

Expectation management is also critical for the development team, since the transition process is going to cause some changes that will affect each team member, such as learning new OO techniques and tools, applying new OO development processes, dealing with legacy systems, and budgeting for reuse, among others. Because of this, team members should have a part in developing the transition plan.

### Written guidance

There is little written guidance for project managers on how to transition to OO (see "Toward a risk management catalog" sidebar). They must find their way through a maze of techniques, tools, and environments with little or no guidance from published sources. Available advice is often slanted toward the product or service being promoted by the consulting firm giving the advice.

### Selecting the first project

Select the first OO project very carefully. The ideal first project should:

- have clean interfaces. Clean interfaces have very little legacy code to violate the project's API. New projects automatically have clean interfaces, which is ideal. If the project is a rewrite of a subsystem in a legacy application, the APIs of that interface must be respected by the other subsystems. For example, if a significant amount of legacy code violates the APIs of the subsystem to be replaced, the design and implementation of the replaced subsystem become overly constrained, adding considerable difficulty to the OO project. This significantly increases the chance of failure.
- be large enough to be meaningful but not larger than the organization is used to handling. The development organization won't learn anything from tiny projects. Also, a small project won't convince others that the transition to OO is worth the trouble. On the other hand, if the project is larger than the organization is used to dealing with, many new problems having nothing to do with OO will increase the chance of failure.
- be supported by senior management. Projects without sufficient support may get killed for reasons that have nothing to do with OO.

See the "The two prime reasons OO projects fail" sidebar for more discussion.

### Software Development Plan

The Software Development Plan is a document that covers all management processes, including:

- risk management,
- the scheduling and activity network,
- corrective action, and
- software development.

The software development processes include development techniques, testing, evaluation, and management configuration. The SDP also describes organizational structures and personnel. For the software developer it's probably the most important document, because it defines the scope of the project and provides a live contract between management and developers. The following lessons are valuable experiences about the transition phase and SDP.

#### The first document

Everyone knows that a business must have a business plan before anyone talks to investors, yet we've seen millions of dollars allocated for software development projects that have no planning whatsoever. The SDP defines each major task and estimates the time
and resources required. It also provides a framework for management review and control. The organization should have a policy requiring every major software development project to have an SDP before funding is approved. This will ensure that project managers perform effective planning by developing and maintaining an SDP for the projects under their supervision. This policy is normally required in a US Department of Defense environment, since an SDP draft must be provided by the software development contractor and included as part of a prospective contractor's proposal.

In a commercial environment, the SDP should be prepared and approved by management; in a military environment, by the program manager. On commercial projects, the SDP is the contract with the project manager’s management or company. On military projects, the SDP is the vehicle by which the development contractor tells the program manager how the contractor will do what was promised. The SDP is also the contract between the project manager and all software development teams.

**Staying current.** A complete SDP doesn’t guarantee success. Even a good one is sometimes ignored by the development team. The SDP may have been written by a consulting firm or by another group who doesn't participate in the management or development activities. Key development team members may not even know it exists.

In commercial settings, managers must enforce the SDP—which may be defined under ISO 9000 or any of the IEEE standards—and make certain product managers do what the SDP promised. In military settings, the program manager must make sure the contractor delivers the promises in the SDP—which must be defined according to military standards—and then verifies contractor compliance.

An update cycle (say, quarterly) must be established to make sure that a “living” or timely plan always exists. The vital processes of the original plan must not be eliminated while updating. In a commercial setting, this is the responsibility of management; in a military setting, it’s up to the program manager. To track changes to the SDP, it may be helpful to place it under software configuration management after initial management approval.

**Development process**

The development process must be documented. The ultimate goal is to map the abstract theories of the OO technique into concrete and repeatable actions that have been customized for the organization.

We recommend that the development process provide more specific detail than usually included in published processes. It may be helpful to use a team experienced in the project process. The process is detailed enough when it tells the team members what's next at each stage.

A well-defined process is essential. It's especially important when OO is still new to the organization. Software development processes ensure that the software products have consistently high quality; using an unfamiliar development technique makes these development processes

---

**Toward a risk management catalog**

_Alistair Cockburn, _Practices and Technology_

Consistently successful project leaders have not communicated how they succeeded in a way that has helped others. They say, "Identify the key risks and address them in order of potential damage. In the years since risk management was named, no body of literature for risk management catalogs—has grown to transfer knowledge. Why not, when many good suggestions await suitable packaging?"

A risk management catalog requires two elements of content—diagnosis and treatment—and two elements of presentation—format and indexing. Words, diagnosis, and “treatment” suggest a medical metaphor. Indexing could therefore be based on symptoms and treatments, quite appropriate for a manual in a project format. A catalog might contain patterns, an example of documenting design patterns (see Programming). The catalog has been started. Each entry has a name and is placed within a major heading (Productivity, Communication, Other Issues, and so on). An entry has six parts: the heading, name, variations, and symptoms. For example, if you are pushing a person in a particular direction, treatment is what you do. The catalog lists what causes you to stop applying a treatment and proposes what to do next.

The following extractor entries come from an agent-oriented project. They clearly apply to all projects. Note that "Sacrifice One Person" refers to another entry: "Team Next Task."

**Diagnosis:** Sacrifice One Person

**Other Names:** Solitaire Variation, Sacrificial Lamb

**Symptoms:** We are getting distracted by our primary purpose.

**Effects:** A single interruption is taking time from all the team members. This is a special case of Team Per Task in which the secondary task is some form of interruption that can be handled by one person.

**Issues:** You cannot drop the distracting task as it is very important. On the other hand, it is keeping the team from moving on its primary task.

**Treatment:** Assign one person to the distraction. Let the rest of the team move forward at full speed.

**Counterforce:** If this keeps happening, you will have no one performing the primary task you want to examine why you have so many distractions in the first place.

**Example:** The schedule was out of date. Project leaders thought it fair to let each person estimate their own work. This would spread the load, experience, and discomfort of work estimation. Progress stopped for weeks, and the designers lost track of what they had been working on.

One team applied "Sacrifice One Person." They fingered one person to do all the estimation, while the others proceeded with design. At the end of several weeks of estimation, that team had moved forward while the others were at a standstill. Thereafter, every team applied it.
even more important. Adapting to OO will undoubtedly change the way developers do their jobs, and a documented development process helps them understand what is expected of them during each development phase. It's also important that team members buy into the new development technique. The manager must ensure that they understand the technique and how it has been adapted to their specific project.

**Staffing.** Staffing the first OO project requires special considerations. First-time OO project managers will find that their previous methods don't work with OO activities. Some team roles may need redefinition.

**Attitude.** Developers' attitudes can make or break the project. Selected team members should have two personality traits: They must be aggressive enough to push forward with new ideas, but willing to learn with a service-oriented demeanor. An uncontrolled ego will prevent even the most talented developer from making the transition to OO. This is especially true for OO because it promotes teamwork; developers cannot work in isolation.

**Mentoring.** Development of a large software system may require an OO mentor to ensure that the methods and tools are rigorously followed, as well as to provide follow-up training to the development team when needed. This allows OO to be "caught rather than just taught." 

**Writing Code.** It's difficult for people to understand OO analysis and design if they don't understand the underpinnings of OO programming. We have found it helpful when all involved, even architects, spend three months programming. This can be emotionally difficult for some, but those who feel that programming is beneath them generally have an ego problem anyway.

**Specialized software disciplines.** After someone has fully made the transition to OO, it may be helpful to have them perform a specialized task, such as OO requirements analysis, design, design patterns, programming, or testing. Specialization leads to quality work, promotes teamwork, improves processes, and leads to the discovery of problems early in the development cycle. Rather than having one person work with a given object through all phases of the development process, give one group of developers the responsibility for OO requirements analysis and handing off the requirements product (for example, providing object interaction and state transition diagrams to other groups for design or programming).

**Early involvement.** The entire team should be involved as early as possible. Ideally most of the developers will be involved in the requirements analysis sessions. In addition to helping everyone thoroughly understand the system being built, bringing everyone together early helps forge the team and creates buy-in for the development process.

---

**Principles:** The principle is to protect the main task, where constant progress is paramount. In the situation just described, it was not that the design teams were complaining of the loss of flow during design, as is the case with Team per Task. They simply were not doing design.

**Training**

Training Team/Progress Team

**Other names:** Day Care

**Sensation:** We are losing precious expertise.

**Symptoms:** The experts are not proceeding at the rate you or they would expect. The people they are supposed to train are draining their energy, time, and concentration.

**forces:** You would like to give newcomers close proximity to an expert. On the other hand, novices distract and drain experts.

**Treatement:** Create an experts-only, progress group and a training group under the tutelage of an expert. Let the progress team design most of the system; let the training team focus on quality training. Transfer people to the progress team as they become able to contribute.

**Counterforce:** If there are only one or two experts, they may both have to work on training until there are at least two people to constitute the progress team and one expert for the training team.

**Example:** When our experts worked together, they made great progress. Then we made an expert responsible for novices, and the expert got little done and the novices got little training. We split the group into experts and mentored novices. The mentor did almost no "active" work, but the novices got good training and the experts got on with their job.

**Principles:** The principle is balancing synergy and distraction. Say an expert produces (E), a novice produces (N), and there are (x) experts and (n) novices. An expert answering novices' questions works at a fraction of their standard, something like (E / n), without raising the productivity of the novice very much. They produce ((E * x / n) + N * n) when mixed together in a group, and (E * (x - 1) + n * N) when one expert is dedicated to training.

The nature of the training does not matter. When adding people to a project, the new people drain productivity from the experts (R. Brooks, The Mythical Man-Month, Addison-Wesley, 1995). Training Team/Progress Team also applies to this situation: put the newcomers in a separate team to learn the system. Move them to the progress team as soon as they are up to speed.

Further reading about patterns

The Risk Management Catalog, A. Cockburn, ed., http://members.aol.com/acockburn. (This is evolving the catalog, with commentary from readers.)


Alistair Cockburn is Consulting Fellow at Humas and Technology in Salt Lake City. He was the focal point for OO in the IBM Consulting Group and formerly a research staff member at IBM's Zurich Research Laboratory. His current specialties are human and social factors in software development, and scaling up OO development projects. Contact him at arc@acm.org.

---

September 1996
Training considerations

Training is essential for a team making the transition to OO development. Just as organizations are wise to buy tools rather than build their own from scratch, expecting developers to build their own training regime through self-directed study is naive and often vastly more expensive than the alternatives.

Periodic retraining should also be administered to help experienced OO developers keep their edge. Like spring training in major league baseball, this lets the team concentrate on the fundamentals in a neutral setting.

Curriculum. The curriculum shouldn't focus on OO analysis or design alone. A view of the entire life cycle should be presented, demonstrating how OO analysis, design, programming, and testing all support subsequent

---

The two prime reasons OO projects fail

Norman L. Kert

It's time to talk about the dark side of objects. The truth is there's a high probability that your first OO project will fail for one of these reasons:

- Technically poor decisions, often made because of lack of mastery of the new technology;
- Ineffective OO tools, components, and methods that overpromise and underdeliver;
- Entrenched cultural norms that negate the advantages of OO thinking; and
- Lack of sufficient budget, time, or experienced people to reach critical mass.

No doubt there are many other reasons, but this message is about a manager's contribution to an OO failure. In leading his or her first OO project, a leader often makes one of these prime mistakes:

1. Expect OO technology to solve a preexisting problem within a group's development process or culture; or

2. Attempt a shift to OO technologies without understanding how to change his/her organization accordingly.

There is no solid data about how many first-time OO projects fail, because in our field it's rare for us to acknowledge and discuss our failures. Nevertheless, a conservative estimate suggests that a first-time OO project's probability of failure is more than twice that of a comparable procedural programming project. As a result, managers need to put a great deal of care into leading their first OO effort.

Often when managers decide to lead a group toward objects, it's with the hope that the technology will help them improve their software development process—increased productivity, lower maintenance costs, reuse, and so on. However, if the organization's cultural attitude toward software development is immature, that attitude won't be fixed by adopting an OO approach. Many things these preexisting problems are even more pronounced.

There is a simple explanation: When software developers work in a new and unfamiliar technology, they use much of many well-practiced skills developed for use in the old technology. The developers are thus working at a reduced capability until they master the technology. With this reduced capability, and the lack of a mature process that shows developers are merely working through their learning curve, processes and procedures that previously had been marginal now become ineffective. My best advice is that a team shouldn't attempt a transition to OO unless they are a level 3 or pattern organization.1,2

The second prime reason for first-time OO failure is that even skilled project managers usually haven't learned how to manage change in their organization.3,4 They are good at leading technical projects, but a change to object technology is a people project. The transition requires clear goals and limits; it needs to be measured, monitored, and funded. Often the project of mastering objects is confused with the project of deploying the technology—they are two separate efforts often running concurrently. Ignoring the people mastery aspect assures that object thinking will be overshadowed by the importance of meeting the schedule. At this point an OO failure is likely.

References


Norm Kerth is a consultant working with companies interested in ensuring that they make successful transitions to OO methodologies and project management. His company, Fajad, specializes in the transformation of programming and design activities into object-oriented processes and processes improvement, project management, and building effective teams. He is particularly involved in building large, distributed object-oriented systems and software project architectures. Contact him at nkerth.com; http://www.togaf.com - Kerth
stages. The training should emphasize how the require-
mments are traced through the entire cycle, to validate and
verify the user requirements and assure that you develop
the right system.

A case study of similar size and complexity as the target
system is helpful for spotting early indications of problems.
Customers should also be included in the training if they
will be participating in the development process by evalu-
ating documents (such as the Software Requirements
Specification) or participating in reviews (such as the

**SPREADING IT OUT.** Training can either be crammed into
a 40-hour week, or it can be spread out over several weeks.
For example, a requirement for 40 hours of training may be
fulfilled in one week if it's totally dedicated to the training
activity. The same training may be spread across two weeks
on a half-time basis. Training may also be spread across 10
weeks by adding a part-time consultant.

We have both used all these combinations and have
found the trainees' depth of understanding and retention
rates to increase significantly when a course is spread over
a longer time period. Compressing the training tends to
overwhelm the trainees.

Although retaining a trainer over several weeks does
cost more, this overhead can be eliminated by some cre-
ative scheduling. For example, if two 40-hour courses are
planned anyway, rather than being taught in two consec-
tutive weeks, they can be taught simultaneously over the
same two-week period. That is, one course is taught in the
mornings and the other in the afternoons. The instructor
cost is the same as it would have been if each course were
taught in a single 40-hour workweek, but the trainees have
the material spread over a two-week period.

**LABORATORY PARTICIPATION.** Because OO techniques
must be practiced, laboratory participation is mandatory.2
Participation should be checked by the management rep-
resentative responsible for the training budget.

**IN THIS ISSUE**

This issue explores several managerial issues and high-
lights the experiences and lessons learned from transi-
tioning and managing OO software development projects.

- **Software metrics.** Metrics are the ruler with which to
  measure progress or failure. Useful software metrics
  measure process, not people. Simon Moser and Oscar
  Nierstrasz report on their study of 30+ projects,
  which shows that using OO frameworks has a sub-
  stantial effect on productivity.
- **Traceability.** Jean-Pierre Corriveau sketches a com-
  prehensive OO software-development process that
  attempts to maximize software evolvability while
  minimizing problems caused by late integration. For
  Corriveau, traceability extends far beyond referenc-
  ing back to requirements—it is central to the entire
  development process.
- **Processes and techniques.** The other four articles
discuss OO management and processes and techniques,
  describing experiences and lessons learned.

**MANAGING SOFTWARE PROJECTS can proceed effectively
when good project management techniques are effectively
applied to OO software development.**

**Acknowledgments**

We thank all the reviewers for helping to select the arti-
cles for this issue. We also thank all those who contributed
articles or sidebars.

**References**

1. B. Berg, M. Cline, and M. Girou, "Lessons Learned from the
OS/400 OO Project," *Comm. ACM*, Oct. 95, pp. 54-64.
2. M. Paulik et al., "Capability Maturity Model for Software, Ver-
Institute, Pittsburgh, 1993.
3. M.E. Fayad et al., "Transition to Object-Oriented Software
4. M.E. Fayad et al., Software Development Planning for OO Pro-

**Mohamed Fayad** is an associate professor at the Univer-
sity of Nevada, Reno. He has been actively involved in over
600 projects in several companies using such methods as
Shlaer-Mellor, Colbert, GOMS, use case approach, unified
approach, design patterns, frameworks, and distributed
computing. His research topic is object-oriented software
engineering: problems and perspectives.

Fayad is serving on several conference program commit-
tees, such as TOOLS USA '96 and Hong Kong OOSD '96. He is
also an associate editor of the Communications of the ACM
and an editor-in-chief of the IEEE Computer Society Press—
Computer Science and Engineering Practice Press.

Fayad received an MS and a PhD in computer science from
the University of Minnesota at Minneapolis. He is the author
of Software Development Planning for OO Projects and
OO Modeling and Applications, IEEE CS Press, 1996. He is
a senior member of the IEEE and the IEEE Computer Society,
and a member of ACM.

**Marshall Cline** is president of Paradigm Shift, Inc., a tech-
nology consulting firm in New York and Dallas. His clients
include Bull, HP, and IBM. He specializes in solving business
problems by exploiting object-oriented software, intranets,
and the World Wide Web. He is experienced in large OO pro-
jects (hundred million dollar), and has led large organisa-
tions (hundreds of people) in their transition to OO.

Cline received an MS and a PhD in electrical and computer
engineering at Clarkson University, Potsdam, New York. His
research interests include formal reasoning, distributed com-
puting, and OO technology. He is coauthor, with Mike Girou,
of the BOLT approach to business, which improves market-
ing and management control over software specification and
development. His writings include several technical articles
and a book on OO.

Cline is a member of the IEEE, IEEE Computer Society, and
ACM.

Contact Fayad at fayad@cs.unr.edu and Cline at cline@
parashift.com.

September 1996