Surviving Object-Oriented Projects

Alistair Cockburn
Humans and Technology
e-mail: arc@acm.org

Pre-publication version, April, 1997.
Published by Addison-Wesley, © 1998.
Preface

So you are thinking of an object-oriented project, or have started one, and want to know what you are up against? Then this book may be of use to you.

Organizations that have successfully made the transition to object technology claim significant time-to-market reduction. Developers say object orientation is a fun way to develop software. There is no shortage of literature on the subject. However, the press has made such a story of object orientation that it is hard to sort out exaggerations and selective reporting from the actual experience one should expect. Speakers rarely seem to want to name the actual costs associated with making the move to objects, perhaps for fear of scaring away future companions. There is, therefore, a lack of information on what sorts of unpleasant surprises await one on an OO project, and what to do about them. It is this lack of information that this book sets out to address.

Scope

This book is an overview of issues I have found in several dozen organizations doing object-oriented projects. From failed efforts, we learned to name specific difficulties. From successful projects, we learned how to get around the difficulties.

The early reviewers of this book unanimously volunteered that it takes several projects to apply the lessons. They said, people on their first object-oriented project are not sufficiently aware of the issues to detect them; they are not yet open to the suggestions; they cannot set aside their old habits and thinking patterns. It is my hope that you can prove them wrong, that you will succeed on your first or your second project by paying attention to the issues and lessons from other people’s experiences.

There are thousands of articles and dozens of books on object-orientation. The book is not a primer on object technology. Two good companion texts for the manager are David Taylor’s Object Technology: A Manager’s Guide and Grady Booch’s Object-Oriented Analysis and Design. The book is not a primer on object-oriented design techniques or of macro- or micro-management. Grady Booch’s Object Solutions is a good companion text in this regard. It is not a technical review of the literature, nor a cataloging of project types.

For all the thousands of articles, there is still a shortage of compiled knowledge of what happens on a project. The information in this book is based on personal project experiences - my own, those I have interviewed, the interviews I have read. The project leader will come up against numerous, specific topics for which an answer is not easily found in the literature, or the obvious answer does not work. In this book, I identify topics, point out hazards and name a workable strategy taken from a project that successfully got past the hazard. The hazards and strategies are collected in the back of the book.

Audience

This book is intended for the busy professional. Here is a reading strategy for four possible readers:

The executive scanning for impact to the organization.

Read the preface, introduction, and the first chapter. This takes you through expectations, costs and project histories. If you are interested in the next level of depth, skim the setup issues involving project selection and staffing, and the chapters on large projects. At that point, you may wish to give the book to the project manager.
The manager before starting on a project.
Read the first two parts of the book: expectations, and project setup. You may benefit by reading the chapters on incremental and iterative development, since those affect the project plan. Then look through the list of hazards and strategies at the back of the book to get a sense of the total set of issues.

The manager on a project.
The primary audience for this book is the manager on a project, working with the technical lead. Some issues are technical enough to require terms of object technology. You, the project manager, may find that the technical lead will bring these problems to your attention or will help you work through them.

Skim the entire book to get the nature and location of topics. When working on the project, look up particular topics when they appear. The chapters are organized in roughly the order topics show up and need to be dealt with. Reread the chapter on corrections before each increment.

The project's technical leader.
Use this book to help your manager understand certain issues, such as organizing teams, developing iteratively, and fighting off unproductive tools and activities. I have added technical depth in a few areas where a project hazard lurks and there is no way to carry the discussion naively. Among these are: simplistic modeling of the business (sometimes passing under the name of “analysis”), overstaffing at the beginning of a project, and false productivity measures. For some issues, it is up to you, the technical leader, to work with the arguments in the book to convince other developers and the management team to adopt a sensible direction.

I have included an extended section on C++, since it is my carefully considered opinion that C++ represents an additional hazard to the survival of the project. If you favor using C++, please read through this section and deal with the issues to assure your project’s success.

Organization
The book is written in ten parts, roughly matching the chronology of encounter with the issues.

The first chapter is the Introduction, in which I give the terms you will have to be familiar with.

The second is a reality check. What expectations are you carrying with you about object technology, and how should you adjust them? It contains stories of a dozen projects, which are referenced throughout the book.

The third deals with selecting and setting up a project. This is the best place to work on survival, even if survival means walking away from the project. It covers all the standard issues of staffing, training, tool selection, methodology, legacy systems, and the like.

The fourth deals with some basic issues in running the project: estimates, plans, milestones, measurements - and design.

The fifth deals with the inevitable corrections you will have to make. I start by citing my favorite project, which started dismally and then turned itself around. From this project we can learn a great deal about tuning. You will have much tuning to do; do not feel bad about making changes during the project.

Pretend that you just finished your project, and are giving advice to another project. What would you highlight for them? The sixth part is a reflection, where you get that hindsight in advance.
The seventh addresses organizations that have safely made it to the point (or declared themselves at the point) of committing large parts of their staff to using object technology. There are new costs and new dangers lying in wait for larger projects.

The eighth is a reality check, in which I compare the contents of the book to a project, and open the topic of organizational culture in overall software success.

The ninth is a compendium of twelve success strategies put into a medical diagnosis metaphor.

The tenth and final chapter is a summary, a pocket guide to copy, or tape to the wall as your reminder on a daily basis.

The topics cross-reference each other extensively. To keep the reading uncluttered, links to other pages are marked in the margin with a page number, like this.

Your project survival depends on your developing insights and reflexes. To give you those, the material is primarily taken from first- and second-hand experiences, my own and those of the many people I interviewed. I devote space to a few published papers which were carefully done and provide those insights. In keeping with the book's intent, keeping footnotes to a minimum, I include "Further Reading" sections after the chapters.

ACKNOWLEDGEMENTS

I thank the non-technical people around me: Kieran, Sean, Cameron, Deanna - I now know why so many authors thank their families. The people at Beans&Brew provided good coffee, good atmosphere and good conversation.

Plato said:
"Only if the various principles - names, definitions, intimations and perceptions - are laboriously tested and rubbed one against the other in a reconciliatory tone, without ill will during the discussion, only then will insight and reason radiate forth in each case, and achieve what is for man the highest possible force..." (from Plato’s 7th letter, cited in translation from Goranzon, B. and Florin, M., “Dialogue and Technology”, p. 46)

This book received much benefit from testing of principles and "rubbing together" of ideas in conciliatory tone and without ill will. I thank:
- Rebecca Wirfs-Brock
- Jeremy Raw
- Brian Henderson-Sellers
- Dave Thomas
- Cecilia Shuster
- the editors at Addison-Wesley
- the anonymous reviewers

for helping improve the book, however the final result may be. I am indebted to Sam Adams for the terms "big-M" and "little-m" methodology. Dick Antalek and Wayne Stevens taught me the most about big-M methodologies.

I thank the authors of the Eyewitness Accounts for taking time to contribute their knowledge. They are:
- Jim Coplien (Lucent, formerly AT&T)
- Ward Cunningham (Cunningham & Cunningham)
- C. D. (independent consultant).
- Sam Griffith (Interactive Web Systems)
- Luke Hohmann (SmartPatents)
- Glenn House (Mentor Graphics)
- K. L. (independent consultant)
- Jon Marshall (ParcPlace-Digitalk)
Tom Morgan (Brooklyn Union Gas)  
Jeremy Raw (independent consultant)  
K.L. and C.D. asked that I not use their names, in order not to discomfit any companies.  
All the above people were kind enough to contribute their experience to this book. They may not agree with everything I write, but we all share the wish to help you succeed on your project.
# Table of Contents

1. **INTRODUCTION**  
   Basic Concepts and Vocabulary  
2. **EXPECT**  
   Project Histories  
   Possible Benefits of Object Technology  
   "OO design, encapsulation, and system evolution" (Tom Morgan)  
   Probable Costs  
   A Checklist of Non-Object Issues  
3. **SELECT**  
   Project Suitability  
   Project Purpose  
   People  
   Technology  
   "Disciplined Use of C++" (Jeremy Raw)  
   "Using Java" (Sam Griffith)  
   Training and Advice  
   Legacy Issues  
   Review  
   "Project Setup" (C.D.)  
4. **EFFECT**  
   Methodology  
   "Building Consumer Understanding of the Design" (Ward Cunningham)  
   Estimates  
   Plans  
   "An Estimation and Planning Session" (Alistair Cockburn)  
   Milestones  
   Measurements  
   Take the Time to Design  
   "Design, and two Smalltalk Projects" (K.L.)  
5. **CORRECT**  
   A Study Project  
   Managing Precision, Accuracy and Scale  
   Increments and Iterations  
   "Burn Some Pancakes" (Luke Hohmann)  
   Your Project Increments  
   User Involvement  
   "Watching Users" (K.L.)  
   "Involve the Users " (Jon Marshall)  
   Project Teams  
   Semitechnical Issues  
6. **REFLECT**
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>EXPAND</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>Your First Big Project</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>&quot;10 Lessons The Hard Way&quot; (Glenn House)</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Training The Tidal Wave</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Productivity</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Migrating the organization</td>
<td>63</td>
</tr>
<tr>
<td>8</td>
<td>RECHECK</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>Winifred Reviewed</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>Technology is only Part of the Story</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>&quot;Organizations&quot; (Jim Coplien)</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td>9</td>
<td>COLLECTED RISK REDUCTION STRATEGIES</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>(1) Knowledge: Clear the Fog</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td></td>
<td>(2) Knowledge: Early and Regular Delivery</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td></td>
<td>(3) Knowledge: Prototype</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td></td>
<td>(4) Knowledge: Microcosm</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td></td>
<td>(5) Teaming: Holistic Diversity</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td></td>
<td>(6) Productivity: Gold Rush</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td></td>
<td>(7) Ownership: Owner per Deliverable</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td></td>
<td>(8) Ownership: Function Owners / Component Owners</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td></td>
<td>(9) Distractions: Someone Always Makes Progress</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td></td>
<td>(10) Distractions: Team per Task</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td></td>
<td>(11) Distractions: Sacrifice One Person</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td></td>
<td>(12) Training: Day Care</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td>10</td>
<td>CRIB SHEET</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Critical success factors, prime failure indicators</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Truths (from Myths)</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Expectations under control</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Selected Survival Recommendations</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>What These Successful Projects Did</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Hazardous Phrases</td>
<td>76</td>
</tr>
<tr>
<td>INDEX</td>
<td></td>
<td>Error! Bookmark not defined.</td>
</tr>
</tbody>
</table>
1. Introduction

Object technology is a technology and a mindset: a technology of software packaging and a mindset for thinking about programming. Three things about it affect you. It is a technology new to your organization. Many of the survival issues have no special dependence on object technology, but come from introducing new technology. Its practitioners expect to use modern software development techniques. Iterative development, user-based requirements, and organization for reuse, will be brought forward during this project. While object technology does not depend on them, object-oriented practitioners put them up as primary issues. Finally, it requires as well as enables increased communication between groups. Two driving forces behind objects are minimizing the impact of changes and creating a sound model of the business. The consequence of the first is increased communication needs within the development team. The benefit of the second is an increased ability of the development team to communicate with users and executives. The two forces need a different kind of software developer, one comfortable with abstractions, uncertainty and communication. This change in the developer’s personality profile becomes one of the forces on the project.

It is only fair that you know my bias as I write this. I have practiced and researched object technology in varying ways and degrees since 1980. I consider it an improvement in software technology; better in some ways though not in all. If that were not the case, I would not waste your time writing about it. It is also fun. Still, I require that it pay its own way with increased software robustness and developer productivity. I therefore have no trouble being hard on the technology where it gives difficulty.

Object orientation offers the following attractions: (1) encapsulation of design decisions, so that the effects of requirements changes can be better controlled; (2) improved communication between the programmers and the users or business executives, resulting in more effective application structure and improved developer morale; (3) additional emphasis on and mechanisms for reuse, a diabolically difficult topic.

These things come with a cost, the greatest of which is training. Current experience indicates that new OO programmers may take 9 months to fully earn their salary again. Multiply that by the hundreds or thousands of programmers that some companies will have to train, and the cost is staggering.

Some executives look at this cost and immediately say, "Not acceptable." Seeing the costs and not the benefits, they decide to wait until the object wave has passed. This book is not for them. They will read it and (rightly) claim, "I told you it was expensive and hazard-prone." Others decide to do a pilot project and watch what happens. Still others rewrite a terminally ill legacy application. A final group bets their companies on it.

If you are just going to try a pilot project, you have to make sure it is big enough to be useful, and then try to get useful information out of it. If you are rewriting a significant application, you need the project to survive to completion, as quickly as possible, with your staff getting trained en route. If you are going to bet your company on object technology, you need to know what to expect, and get some ideas for dealing with the training.

Here, then, are 4 critical success factors in creating a surviving, even thriving project.

Success in four “simple” steps:

1. Use incremental scheduling and staging.
2. Find and fix failing ideas.
3. Develop a habit of delivering.
4. Develop a good sponsor, project manager, and technical leader.
Do not confuse simple with easy. Simple often implies difficult, because simple things must be repeated daily, or require changing your work habits. Habits are a most difficult thing to change. The book reiterates those four key factors throughout.

If those are the critical success factors, I have found two key failure indicators:

**Two key failure indicators**

1. Absence of incremental development.
2. Use of C++ in a commercial IS department.

For the absence of incremental development, the argument is simple. If your project plan does not give you a place to recover from an initial error, you are unlikely to recover. Most of the projects I have visited have made mistakes, and the key to success was to have a way to recover.

The chapter on selecting technology includes C++, Smalltalk, Java and OO COBOL. All present hazards due to various misconceptions. C++, however, has been the cause of enough troubles that it merits extra attention.

I must be clear that C++ does represent an extra hazard to the survival of your project. Yes, many C++ projects do ship successfully - and we know enough now to recommend taking certain precautions. Even experts of C++ agree, notwithstanding the benefits they get from the language. I provide additional information where C++ is concerned and have been careful to check with specialized consultants.

I should like to offer a metaphor for having a project survive, even thrive. It is based upon an old Japanese story about three physician brothers. The youngest brother was court physician and widely renowned. He deprecated his fame, saying, “I am by far the worst of the three physicians, for I am forced to wait until the sickness is visible, and then must resort to puncturing veins and spreading salves. My older brother is better, for he can detect sickness at onset and make minute changes that keep it from developing. My eldest brother is best, for he sees the spirit of sickness and removes it before it takes shape. He acts so early that he is not even known as a physician.”

In the spirit of this metaphor, I distinguish the interventions needed for a project. Adequate interventions are done in time to solve the problem. Better interventions are early and small, as the spirit of sickness appears. The best are acute, advanced observations fed into a balanced risk reduction program, thus not looking at all like interventions. This presents a difficulty to the external observer and chronicler: the best-run projects show the least outward signs of course corrections. Everything just looks easy. Therefore, I have selected as a study project one that went wrong at the beginning. The lessons for us all are how they put themselves right again.

A success is better than a failure. While you can learn some lessons from failure, from success you learn how to succeed. I wish you success and the learning that comes from it.

**Basic Concepts and Vocabulary**

“Object technology” is fundamentally a technology of program packaging. It gives the designer a new set of packages from which to work: data-only, data-and-function, or function-only packages, in a full continuum. The new packaging lets designers build models containing both the data and the functions associated with the things modeled. Letting the data have behavior allows designers to work as though the software packages are really the items modeled. They can recreate the problem domain’s structure and put that structure directly onto the computer. This is good news for the development of systems.

The packaging lets the designers put those software elements together that are likely to change together. This reduces the cost and trajectory of changes. That is the best news.
If program packaging is what is fundamentally different, then only program packaging aspects of project management should fundamentally change. This is what I find on projects and what we shall see in the book. Object technology changes project management much less than it changes the thinking of the designers and the maintainability of the software. This is good news to you, the project manager.

User involvement, incremental and iterative development, risk reduction and reuse are not fundamental elements of object technology. You should already be doing those. It just happens that object-oriented people insist on them. I include those topics in this book because they are still not widely practiced and they affect your project’s success.

This book is not a primer on object technology, but we need some common vocabulary. Below are the main terms I expect you to become familiar with. I illustrate them with a simple example based on a wage-payment system:

Adam, Bill and Dave are employees. Adam works part-time, paid on an hourly basis. Bill is full-time, paid on an hourly basis. Dave is full-time, paid on a salary basis.

A "class" is the collection of subroutines to be packaged together and a description of the data to be packaged with them. It is a description of a certain class of objects, from which individual instances may be created. This combination of subroutines and data provides the continuum just referred to and is the first of the packaging improvements of object technology. In the Salary example, there might be a class describing all Hourly Employees. It would define that both the rate of pay and the hours worked need to be tracked separately for each employee, i.e., that each HourlyEmployee instance has two data slots, one for RateOfPay and one for HoursWorked. The class would contain the functions needed to compute the wages based upon the hours worked and rate of pay. A common way of drawing such a class is like this, with the class name on top, the data slots next, and the functions third.

```
HourlyEmployee
rateOfPay
hoursWorked

payment ( )
```

Figure 1.1 Sample diagram of a class.

"Object" officially means an instance of a class, but it is often also used to mean the class description itself. Thus, you might hear designers talk at one moment of the "object model", meaning the class descriptions, then at the next moment of an "account object", meaning an instance of the "account" class. The context usually makes their intentions clear. When it becomes important to separate the two, they do so. I follow that common convention in this book, since you may as well start encountering the terms here as you will on the project. A common and pleasant way of drawing an object instance is as a donut, with the functions on the outside, protecting the data on the inside. Different modeling languages have their own ways of drawing object instances, and on your project you will use whatever drawing style is given by the modeling language you select.
“Inheritance” is a programming mechanism that lets the common part of selected classes be shared. The common part is made a class in its own right, and called a “superclass” of the selected classes. Each selected class is called a “subclass” of the superclass. The selected class is linked to the superclass to indicate that the superclass is really part of the selected class’s definition. Selecting common parts to put into a superclass is one of the key decisions in creating a robust object-oriented system and is not at all a trivial decision.

Inheritance is the second packaging advance in object technology, after classes. It allows a programmer to extend an existing program by just adding a new subclass, taking advantage of all the thought and labor that went into the superclass. It allows one to program just the differences between two systems.

The following diagrams show three possible ways to organize the classes in the Salary example, using the OMT notation. None of the three designs is "right" or "wrong". They work well under different circumstances. In any real system, every box will be filled with the data and functions that make each employee type different, differences which I omit for this little example. The superclasses each define some information for its subclasses, and the subclasses therefore do not repeat that information. That is what I mean by "programming just the differences".

![Diagram](image)

**Figure 1.2** One possible way to organize the classes in the Salary example.
"Encapsulation" refers to putting into one place things related to one topic, and creating a set of functions to access and operate on that topic. Although encapsulation can be accomplished without object technology, it is an essential characteristic of objects. You can encapsulate various things within an object, data being one. The most powerful thing you can encapsulate is a "design decision". Encapsulating a design decision means putting into one place the data and functions related to that decision, creating a set of functions to access and operate on that topic. A perfect example of encapsulation and its value is given in Tom Morgan's Eyewitness Account.

The donut figure in Figure 1.2, above, is chosen to give a visual sense of encapsulation. In the three possible inheritance structures for the employee example, each class and superclass is designed to hold all the critical information, i.e., encapsulate, pertaining to that topic. Thus, the Employee class should hold whatever information, decisions, data and functions are common to all employees, no matter what their payment structure. The assumption in this example, is that all employees have a rate of pay, and get paid. The designer will select between the three designs based on where the boundaries best fit the encapsulation needed on the project. If there is heavy predicted use of the notion of HourlyEmployee, then that class is a good one to create, so that future changes to the way hourly employees are treated can all be handled by modifying this one class. Modifying only one class keeps the trajectory of change to a minimum.

"Polymorphism" refers to the ability of a function name to provide differently detailed behavior depending on the particular type of object being held at that moment. In the salary example, the function "payment( )" is polymorphic: the calculation of the payment is different for each class, even though the intention is the same. Whether you ask for Adam's payment, Bill's payment, or Dave's payment, you mean the same thing, even though the rules of calculation differ. Polymorphism is quite common in natural language, although we do not use that term. We say answer the phone, answer the door, answer the letter, and answer the question. We are not at all disturbed that answering the door is a different activity than answering a phone, letter or question. A common intention links them.

In programming, polymorphism allows continual extension of the design. As new classes are added, they might add a new behavior to an existing function name. If done well, this is another way of "programming only the difference". Polymorphism always adds to the testing effort, since every new polymorphic function must be tested against all previous uses of that name.
A “framework” is a collection of objects, classes, or subsystems that provides most or all of a needed function, and can be tailored to different situations. A framework is designed so that selected situations are particularly easy to address.

An object-oriented framework takes advantage of classes, inheritance and polymorphism. The trick in designing a framework is to guess which new situations should be most easily supported. For different proposed future needs, different classes and superclasses are best. It is the guesswork about the future needs that make framework design difficult, costly and revision-prone. However, once a suitable framework is available, development costs drop dramatically, since only simple tailoring activity is needed. As a system's architecture relies on numerous frameworks, designing frameworks for other developers' use is a primary activity of the lead designers.

In the chapter on benefits, I mention responsiveness to evolutionary changes as the greatest benefit of object technology. This responsiveness is directly due to classes, inheritance, polymorphism and framework design.

“Incremental” and “iterative” development are two distinct scheduling and staging strategies. Both involve doing analysis and design more than once during the project, and possibly even gathering requirements more than once.

“Incremental” development involves completing parts of the system at different times and different rates. It does not imply revising any part of the system already completed. It is aimed at improving the quality of the process, fixing errors in knowledge about the software development process.

“Iterative” development involves reworking a part of the system. The rework improves the quality of the system. The user interface and the infrastructure of a system are two parts of software systems that benefit the most from the improvement. The user interface is improved to better serve the users, and the infrastructure is improved to simplify the evolution of the system.

Incremental development is easier to adopt than iterative, and more crucial to the success of the project. It is easier to use because the schedule can be thought out in advance and linearized. Iterative development requires initial guesswork, constant observation and flexibility in execution. My suggestion is that you absolutely use incremental development, and in the increments learn how to manage iterative development.
2. Expect

_Bail out early or plan to survive_

Are you looking at object technology because your maintenance costs are too high? Do your competitors claim to be getting benefits from objects? Do you have a program development backlog? Or has your boss just ordered you to start using objects?

Whatever the reason, you know that the move to OO will cost you. What are you willing to pay, and what do you expect back? When do you expect it back? In this part of the book, we consider what you could be looking for, what you should really expect to receive, what you probably expect to pay, and what you may be overlooking.

Read the success and failure stories, track your responses through these chapters, and decide whether to return this book back or continue on your course.

**PROJECT HISTORIES**

This section summarizes 11 project stories. These are projects where I was able to interview a project leader, and gather project data and reflections on what went wrong or what they did right. Each offers an insight into a failure, or a strategy for success. I have seen these issues repeated on other projects. For discretion and because of some proprietary information, I cannot name each project. I do name Object Technology International, Brooklyn Union Gas, and Mentor Graphics. I gave the other projects nicknames to make them easier to refer to, and I refer to them throughout the book.

Each project story has a project profile card, indicating the project name or nickname, the key topic I associate with it, and the size and experience of the staff. I give some historic details about the evolution of the project, and some additional comments based on an overall evaluation of the project.
1. "Alfred": success with changing requirements.

<table>
<thead>
<tr>
<th>Project:</th>
<th>Project Alfred</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key topic:</td>
<td>Project setup</td>
</tr>
<tr>
<td>Project staffing:</td>
<td>Small (5-10)</td>
</tr>
<tr>
<td>Project type:</td>
<td>Investigative</td>
</tr>
<tr>
<td>Staff experience:</td>
<td>New to OO</td>
</tr>
<tr>
<td>Duration:</td>
<td>6 months</td>
</tr>
</tbody>
</table>

Project Alfred was set up to take a look at object technology. It included one super-programmer from the internal staff, 2 regular developers, a user interface developer, and a business expert with no programming experience. It was managed by an experienced project manager familiar with incremental and iterative development. The only person with any OO experience was a part-time object-oriented expert, who acted as trainer and design guide.

The first increment was set for six months. The business model was built and programmed first, prior to any user interface. The user interface was attached at the end of the increment. Just prior to the completion of the 6-month increment, the sponsoring executive told the design team that acceptance of the technology would depend on this test:

“There is a change we have wanted to make for a long time, but have not because it will take four months and 10 people to make the software and the database changes. You will be given from Friday morning until Monday morning to make that same change.”

Six people worked through the regular working hours on Friday. Four people worked until late afternoon on Saturday, at which time the change was done, including changes to the database. The team showed it to the executive on Monday, demonstrating their ability to react to change.

The primary insight I would like to draw from this project is the way the project executive set up a test bed for the technology, staffed it carefully, and then gave the technology a test. This is a good example of an investigative project (see Project Purpose).
2. Brooklyn Union Gas: success through attentiveness.

<table>
<thead>
<tr>
<th>Project</th>
<th>Brooklyn Union Gas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key topic</td>
<td>Risk management</td>
</tr>
<tr>
<td>Project staffing:</td>
<td>Large (100+ people)</td>
</tr>
<tr>
<td>Project type:</td>
<td>Legacy rewrite</td>
</tr>
<tr>
<td>Staff experience:</td>
<td>New to OO</td>
</tr>
<tr>
<td>Duration:</td>
<td>Several years</td>
</tr>
</tbody>
</table>

In 1986, B.U.G. was programming in PL/I using mainframes, IBM 3270 terminals, and a relational database. They needed to rewrite their billing system. After evaluating structured development techniques, they concluded that it would take too long to finish the system using those techniques. They decided it would be worth the added risk to develop the system in the early years of the next generation programming technology rather than in the waning years of the existing programming technology.

Not much was known about the long-term consequences of certain design decisions. They decided, in particular, that the inheritance hierarchy would be an area of risk, that they would not be able to change it easily once they put it onto their database. They therefore designed the hierarchy for simplicity and stability rather than for maximum reuse. They selected and followed simple, conservative standards. For key risky and important areas, such as the object-oriented dispatch mechanisms, they allowed for several revisions. Through these and other policies, they created an object-oriented dialect of PL/I, delivered the system and demonstrated improved system evolution over time.

They demonstrated that object orientation can be managed on any platform, in any language. Having visited dozens of projects, I also feel it shows the quality of their management and development teams.

This project contains numerous lessons for keeping a project alive:
- thoughtful planning (Estimates),
- use of increments and iterations (Increments and Iterations),
- remembering to use previous experience (Non-Object Issues),
- setting and following simple standards, and
- using the best parts of their previous methodology (Methodology).

Tom Morgan provides an Eyewitness Account of the value of encapsulation in design. A project history was published in “Object-oriented development at Brooklyn Union Gas”, by John Davis and Tom Morgan, IEEE Software, January 1993, pp. 67-74.

<table>
<thead>
<tr>
<th>Project:</th>
<th>Project Ingrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key topic:</td>
<td>Recovering from mistakes</td>
</tr>
<tr>
<td>Project staffing:</td>
<td>Medium (20-40)</td>
</tr>
<tr>
<td>Project type:</td>
<td>Production</td>
</tr>
<tr>
<td>Staff experience:</td>
<td>Beginners</td>
</tr>
<tr>
<td>Duration:</td>
<td>2 years</td>
</tr>
</tbody>
</table>

Project Ingrid was a project which decided to use C++ for the wrong but obvious reasons that they wanted to go to objects and had staff trained in C++. The 25 programmers assigned to the project had no experience with object-oriented programming. They did not use consultants or training, thinking that good programmers would be able to learn this new programming language on their own.

Recognizing the risk they were taking on, the project manager led a successful campaign to deviate from the corporate-standard, waterfall-based scheduling and staging strategy. They obtained permission to use an incremental, not iterative, strategy, in which not even the requirements would be written all at one time, but only for the next segment of the project.

At the end of the first increment they were faced with a disaster. They were beyond the schedule and not done, the design was not good, and morale was down.

The project managers replaced 24 of the 25 programmers, changed the management structure, introduced training, and then redid the first increment as part of the second increment. They delivered the first and second increments together, albeit behind the new schedule.

After additional process and management changes, they instituted an internal self-tutoring process. By this time, they were comfortable and skilled with objects, and made their third and fourth increments without difficulty. The project manager said they were able to take increasing advantage of internal similarities, getting internal reuse that allowed them to beat their scheduled delivery times.

I consider this the ideal study project (A Study Project), since it illustrates the theme of the book so well. It is ideal in that it started with some mistakes (Technology) and corrected them in time. The key insights I would like to draw from this project are their willingness to change everything in order to succeed, and their use of increments to give them the breathing space needed to make those changes (Increments and Iterations).

<table>
<thead>
<tr>
<th>Project:</th>
<th>Project Manfred</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key topic:</td>
<td>Iterations</td>
</tr>
<tr>
<td>Project staffing:</td>
<td>Medium (10-20)</td>
</tr>
<tr>
<td>Project type:</td>
<td>Production</td>
</tr>
<tr>
<td>Staff experience:</td>
<td>New to OO</td>
</tr>
<tr>
<td>Duration:</td>
<td>One year</td>
</tr>
</tbody>
</table>

Project Manfred was scheduled to use Smalltalk on a new and critical production system. It was being done by a programming division containing experienced C programmers. The programmers assigned to the project had no experience with object-oriented programming. As in Project Ingrid, they did not bring in consultants or training, thinking that good programmers would be able to learn this new programming language on their own.

The team leaders decided that since Smalltalk had such a good development environment, their design method would be to build and revise prototypes. After the final prototype was accepted, they would reprogram the entire system in production-quality code.

When they put the first prototype into the hall, they were told it was far too slow. “That is all right,” they said. “It is just a prototype. Tell us what you want.” “We want it to run faster,” was the reply.

The story was the same with the second and the third prototypes. By this time, higher-level management was getting nervous, because it appeared that the team was either not taking feedback seriously or Smalltalk would be too slow for the production system. The design team kept saying that the prototype was allowed to be slow, because it was not engineered for production.

Finally, the team was told to make the prototype run at production speed or lose the project. They were able to make some performance improvements in the time given, but not enough to meet the production speed requirements. The project was canceled, at great cost.

The first of a trio of mistakes was confusing prototype and production software. It is an oft-quoted syndrome (“The prototype looks good, when can we ship it?”) to which I have not seen a successful counter strategy, other than not to build a non-production prototype. The suggestion I pass along is to call it a “requirements model”, and to throw it away completely as soon as possible, so that a production-quality version can be started.

The second mistake was in the designers thinking that prototypes replace design (see Take the Time to Design and the Eyewitness Account by K.L. there). That mistake happens in projects of all sizes and all languages. I have seen the same issue on a 200-person project as well as a 10-person and a 1-person project.

The third mistake is called “prototypitis” in this book (Increments and Iterations), and consists of not controlling the amount and degree of iterations.

<table>
<thead>
<tr>
<th>Project:</th>
<th>Mentor Graphics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key topic:</td>
<td>Corporate conversion project</td>
</tr>
<tr>
<td>Project staffing:</td>
<td>large (ca. 600)</td>
</tr>
<tr>
<td>Project type:</td>
<td>Production</td>
</tr>
<tr>
<td>Staff experience:</td>
<td>New to OO</td>
</tr>
<tr>
<td>Duration:</td>
<td>Months</td>
</tr>
</tbody>
</table>

Mentor Graphics once dominated a segment of CAD marketplace. They were a seasoned and aggressive C programming shop. In the early 1990's they committed to converting themselves to becoming a C++ shop with release of version 8.0 of their software.

That release nearly sank the company (read the Glenn House Eyewitness Account). It was delayed beyond a reasonable marketing time, and it became clear that the hardware their customers were running on would need to be upgraded to run their new software. In a very short period of time, they lost a great deal of market share. Since then, the company recovered and has learned how to use C++. At the time, however, the executives put much of the blame on C++.

Even at this point of the story, you should notice that having C programmers does not automatically mean succeeding with C++ (Technology). The second immediate lesson is that moving an entire company to object orientation is not as simple as issuing a decree and selecting a compiler (see Expand). Glenn House, Vice President at Mentor Graphics, was involved in Release 8.0, and contributes an Eyewitness Account on their experience.

<table>
<thead>
<tr>
<th>Project:</th>
<th>Object Technology International</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key topic:</td>
<td>Productivity</td>
</tr>
<tr>
<td>Project staffing:</td>
<td>Small - Medium</td>
</tr>
<tr>
<td>Project type:</td>
<td>Production</td>
</tr>
<tr>
<td>Staff experience:</td>
<td>Expert</td>
</tr>
<tr>
<td>Duration:</td>
<td>Months</td>
</tr>
</tbody>
</table>

Object Technology International has been making its living since the late 1980’s contracting to deliver hard real-time systems in Smalltalk, "on time or you pay nothing." The company’s president, Dave Thomas, relates that at the time they started, no one would contract them unless their project was dangerously late and they were desperate. “On time or you pay nothing" was one way to sell OO services at that time.

Using experienced developers and a custom development environment, OTI met every project deadline and performance requirement over a 10 year period. Dave Thomas told me he plans on 9 months until a new hire is able to earn their salary fully. Running a small company, he recognizes both the cost and the value of education.

Regarding the methodology they follow to deliver their products, he said they hire good people and give them good tools to help them get their work done. The tools (ca. 1993) included code-generation tools and program measurement tools, but not upper-CASE tools. One of the tools, Envy/Developer, eventually was made available to the market as a versioning and configuration management system for Smalltalk.

Although this is a brief discussion of OTI, there are still a few insights to draw. The first is that they developed their own, organization-specific methodology (Methodology). It included which projects to turn down, and what tools to use and not use. They also stayed aware of their own limitations and the limitations of the current technology (Technology). Finally, they were able to demonstrate that Smalltalk could successfully be used for hard real-time systems with tight performance constraints, including waveform generation on an oscilloscope.

<table>
<thead>
<tr>
<th>Project:</th>
<th>Project Reginald</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key topic:</td>
<td>Project setup</td>
</tr>
<tr>
<td>Project staffing:</td>
<td>Small</td>
</tr>
<tr>
<td>Project type:</td>
<td>Production</td>
</tr>
<tr>
<td>Staff experience:</td>
<td>New to OO</td>
</tr>
<tr>
<td>Duration:</td>
<td>Months</td>
</tr>
</tbody>
</table>

Project Reginald started as two programmers given the assignment to investigate object technology. They should report back after 6 months as to whether it was viable for the group. They were experienced C programmers, investigating C++, and given access to training and industry consultants. The task was computer communication.

The project became more important to the division, until there were three programming teams in two countries, connected by satellite links. Over the course of a year, the scope kept changing. At the end of the year, the project manager was removed and the project folded for lack of results.

In the debriefing, we tried to find out just what went wrong. It appeared that the language was not to blame. We blamed the deliverables-heavy methodology used, the distances separating the groups, and the changes in the project’s scope.

The project’s original goal was to investigate whether object technology would be viable for the organization (contrast this with Project Alfred). The failure of the project was largely due to the way that the ground rules changed, and the project was not restarted, reoriented to its new goal. The two original, lead designers thought they were working on a low-key, small, investigative project, just to find out what C++ could be used for. It suddenly became an important production project, and they became team leaders. They never made the shift (Project Purpose).
8. "Stanley": too much cutting edge.

<table>
<thead>
<tr>
<th>Project:</th>
<th>Project Stanley</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key topic:</td>
<td>Cutting edge technology</td>
</tr>
<tr>
<td>Project staffing:</td>
<td>Large (100-200)</td>
</tr>
<tr>
<td>Project type:</td>
<td>Production</td>
</tr>
<tr>
<td>Staff experience:</td>
<td>New to OO</td>
</tr>
<tr>
<td>Duration:</td>
<td>Years</td>
</tr>
</tbody>
</table>

The company running project Stanley was not in the computer industry, and did not have an experienced software team. The project sponsor simply fell in love with object technology, and declared they were going to make a large project, a modern project, “another Brooklyn Union Gas”. The requirements document was a collection of all the latest ideas they had recently heard, and included newer notions every few months. Over time, the proposed system evolved to one requiring numerous, fully decentralized, peer-to-peer servers running distributed Smalltalk. At the time, there were no Smalltalk distribution frameworks and CORBA was not yet developed. The project restarted several times over several years. At the time of this writing, it has still not deployed.

Several lessons are visible. One is not to change requirements so frequently, to which I hardly need add comment. I also wish to extract the lesson not to plan on using new technology for a time-critical project if the technology is not available on the market, in your company’s expertise, or your group’s experience (Technology). Although it appears rather obvious in this lopsided story, overambitious technology plans cause many project failures.
2. Expect


<table>
<thead>
<tr>
<th>Project:</th>
<th>Project Tracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key topic:</td>
<td>User involvement, Methodology</td>
</tr>
<tr>
<td>Project staffing:</td>
<td>Medium (10-20)</td>
</tr>
<tr>
<td>Project type:</td>
<td>Production</td>
</tr>
<tr>
<td>Staff experience:</td>
<td>New to OO</td>
</tr>
<tr>
<td>Duration:</td>
<td>Months to Years</td>
</tr>
</tbody>
</table>

Project Tracy was run by an experienced software development group, with experienced developers learning object technology. They read the literature and came away with two impressions: get users involved and model the real world.

They started with the assumption that object technology was “just another programming language”, which could be learned by their programmers. This meant, in particular, that they did not get special training for the team.

They tried to get users involved in the system specification and design review, but got only lukewarm support. They were given access to whichever users happened to be free at any moment. They therefore had a continually shifting user group, with varying views, opinions and skills. There was no continuity in the discussions with the users, and the more experienced users never found the time to come to the meetings.

They were using a new modeling technique, one that they were told was good at “modeling the world”. They were under the impression that if they just modeled the world, they could directly turn the model into their classes. In the postmortem, they decided their modeling was naive, and that even a decent model of the world might not turn into a decent software design.

The mainframe and workstation groups were physically separated, with the mainframe group being the larger of the two. The two groups did not communicate well, making it hard to alter the overall architecture. Each group claimed it was doing its job properly, but the total system did not function. They eventually changed the team structure, so that people were assigned to total, cross-platform function as well as to subsystems.

The insights I wish to draw are: do get real user involvement (see Jon Marshall’s Eyewitness Account), do not be naive about business modeling (The Domain Model), assign a person to every form of deliverable, use cases and classes in particular (Methodology).

<table>
<thead>
<tr>
<th>Project:</th>
<th>Project Udall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key topic:</td>
<td>Attitude, Architecture</td>
</tr>
<tr>
<td>Project staffing:</td>
<td>Medium</td>
</tr>
<tr>
<td>Project type:</td>
<td>Production</td>
</tr>
<tr>
<td>Staff experience:</td>
<td>Mixed</td>
</tr>
<tr>
<td>Duration:</td>
<td>2 years</td>
</tr>
</tbody>
</table>

Project Udall was a medium-sized project that found itself in trouble. Developers were milling around, without a cohesive architecture for them to work around.

A few of them convinced their management that the only way to make progress was to start over, with a small team. They put the dozen or so developers on hold until they could be useful. A small team worked over several months to create an architecture. They defined subsystems for further development. When a good team leader could be found and brought up to speed, a few developers were invited to work with the team leader on the subsystem. Those not considered valuable by the team leaders were simply not invited onto any team. The developers put on hold did what they had been doing anyway up to that time: doing nothing or learning.

The system eventually shipped and was considered a success.

The person telling this story said that success hinged around being able to put the project plan into reverse, to admit that they had overstaffed and proceeded along a false direction. Fortunately, system delivery was considered more important than pride in the initial project plan, and more important than optimizing use of salaries (a reference to developers sitting "idle"). The other insight from the project is the value of getting the architecture done first, by a small group of people, and evolving to development of subsystems by small teams (Your Project Increments).
11. "Winifred": inattentive but persistent.

<table>
<thead>
<tr>
<th>Project:</th>
<th>Project Winifred</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key topic:</td>
<td>Incremental Development</td>
</tr>
<tr>
<td>Project staffing:</td>
<td>Medium (20-40 people)</td>
</tr>
<tr>
<td>Project type:</td>
<td>Legacy system rewrite</td>
</tr>
<tr>
<td>Staff experience:</td>
<td>New to OO with mentors</td>
</tr>
<tr>
<td>Duration:</td>
<td>2 years</td>
</tr>
</tbody>
</table>

Project Winifred involved rewriting a legacy mainframe system, replacing it with a three-tier, client-server-mainframe system. It used Smalltalk on the workstation, C on a relational database server, COBOL on the mainframe. The project started off with an OO-experienced executive, an OO-experienced lead designer for the workstation portions, an experienced mainframe designer for the mainframe portions and an experienced project manager. They had access to OO experts, training and tools. They had funding, management support, and an incremental development strategy. The project had two goals: deliver the new system, and train a dozen newcomers to object technology.

The project immediately encountered morale problems. Communication within and across the teams did not work, and the OO consultants lectured instead of listening. The requirements phase was done all at one time, taking nearly 4 months. During this time, no architecture or design activity took place, and the group began to tire. Job assignments were not clear, and technical leaders, each of whom had their own style, were replaced frequently. These morale and communications problems lasted for half of the project.

Most of the programming was done by novices working under experts. The experts were overwhelmed by the task of programming and teaching concurrently.

Initially, the staff was teamed according to delivered function. With no ownership of the classes’ design, the classes became collect-alls of different people’s ideas, and no one dared touch them.

The first increment barely shipped, and did so with a weak internal architecture and minimal functionality. On the second increment, the expert staff was changed, new teams were set up, and a greater emphasis was given to the internal architecture. The second increment shipped successfully and with the first coherent architecture. Over the increments, the architecture evolved and the team became confident about delivering the system on time and with function.

The requirements shifted continually. To deal with this, an iterative process was set up within each increment, requiring two fully functional viewings by the user community to settle final requirements.

The project manager, when asked what she thought of the claims of maintenance and productivity made for object technology, answered that the Smalltalk group consistently made changes faster than she was accustomed to seeing, that there was no way the mainframe COBOL group or the short-staffed database group could keep up with the iteration cycle being used on the workstation. As a result, they changed the rules for iterations, so that the Smalltalk group had three iterations per increment, and the others each received a draft specification followed by a final specification.

At all times, management support was good, money was available for equipment, and there was an accessible and fairly stable user group.

The first insight to draw from project Winifred is not to forget the simple lessons of previous experience: job assignments, lines of communication, ownership of deliverables (Legacy Issues). The second is that the unwavering support of upper management provided a constant boost and freedom to maneuver during periods of low morale. The third is to use incremental development (Increments and Iterations). It gave them a chance to find their weak areas, reorganize teams and develop the architecture. It gave the team increased confidence and morale. By the third iteration...
they knew they could and they knew how to produce a system every three months, which gave them freedom to experiment a bit, even under tight time pressures.

The fourth insight was that they managed to keep their users involved consistently (User Involvement). The final comment for now is that a number of the suggestions in this book were tried on that project, and helped it become more stable.

Project Winifred is examined more closely in Your Project Increments and is analyzed in Project Winifred Reviewed.

How many of the successes and failures just described were due to object technology and how many to (lack of) normal software engineering practices? The first contribution of this book is to remind you to apply the principles you are familiar with from your non-OO projects. For some readers, the shift to objects will provide a first opportunity to put these suggestions to use.

One Line Summary of Project Histories:

Learn how some cleared the chasm and became productive

POSSIBLE BENEFITS OF OBJECT TECHNOLOGY

You may have heard of some possible benefits from object technology. The following list is my ordering of these possible benefits based on interviews with project leaders, discussions with consultants and experts, and reading project reports. I list them in their order of likely contribution based on that experience, with the most likely and most rewarding ones first and the least likely and least clear last.

1. Responsiveness to variations on a theme.

Object-oriented software excels when your next program differs only mildly from your previous. OO design orients itself toward this situation. Properly designed classes and frameworks let you add new features just by describing the differences between the new and old systems.

If your company competes on time to produce variations on themes (as with sales promotions, financial instruments, insurance policies, and so on), then object technology is naturally suited to your situation.

2. Responsiveness to change.

Large and unpleasant changes can often be made relatively easily to an object-oriented system, due to encapsulation (Introduction). We wish to localize design changes in software, and encapsulating design decisions into classes helps us do that. Part of the OO design procedure is to identify probable changes and localize them. The Eyewitness Account by Tom Morgan describes encapsulation in action.

Consistent attention to encapsulation of design decisions pays off when unexpected and large requirements changes are encountered. These might be changing the hardware platform, changing a server or its function, or changing the nature of the user interface. I separate such significant requirements changes from maintenance and evolution because of the difference in frequency, effort and savings.

3. Time-to-market.

If one thing is pushing you toward object technology, it is probably time-to-market. Most systems contain internal similarities, parts of the system being similar to each other rather than to other products. Using incremental development, the development team can capitalize on the first two benefits of object technology, encapsulation and variation-on-themes, to take advantage of the internal similarities. The first release of the incremental development staging corresponds to
an initial product, in which the basic classes and frameworks are created. Subsequent releases evolve and extend the classes and frameworks, taking advantage of the internal similarity.

Your time-to-market benefit will vary depending on the amount of internal and external similarity you can exploit.

4. Communication between developers and their users and executives.

Object technology lets developers work in users' and executives' vocabulary.

Normally there are two translations involved in creating a program: when the developer translates a user's or executive's statements into the programming language, and when the developer tries to convert the program structure back to something the user can use, at the user interface. Each translation is error prone and costly. Simplifying the translations reduces cost and effort, both of communicating ideas and developing systems.

It is the packaging technology that lets the developer put the user's vocabulary directly onto the computer. Thus, instead of arguing about data structures on the disk, developers, as Tom Morgan said about the Brooklyn Union Gas experience, argued "about whether the fault is in the gas meter or the gas line." Although that may sound rather odd as a benefit, organizations that have made the transition to objects successfully repeatedly cite improved communications as one of the most significant benefits (see the Eyewitness Account by Ward Cunningham).

5. Maintainability.

You may be in the situation that your current code is old, weakly structured and difficult to modify, and your hope is that the new OO code will be so well structured it will be easy to maintain. The good news is the ability to encapsulate design decisions, possible benefit #2, above. Use of encapsulation will improve maintainability.

There are, however, new opportunities for creating poor software structure. If many classes are used to carry out each system function, it becomes difficult to understand how the system works. Each new person will have a hard time learning how the system works and may make mistakes in correctly placing changes.

Thus, object-oriented design does not automatically confer maintainability - it only provides the capability for it.

6. Reuse.

Perhaps you expect to reap massive gains on subsequent projects through reuse. Reuse in object technology is an unpleasant repetition of "business as usual" with a bit of good news.

The good news about reuse is that object orientation provides additional reuse mechanisms: classes, inheritance, polymorphism, and frameworks. They allow you to reuse both smaller and larger parts of the program, and to develop by "programming the difference" from what you already have.

The bad news is that reuse really depends upon severe and human issues. These are, first, the willingness of your developers to suppress their egos long enough to look for and use someone else's solution, and secondly, the willingness of your managers to allocate time and money to finding reusable components. Reuse within a project is fairly easy, across projects is fairly hard, and across an organization is exceedingly difficult. This is not new news; it is really business as usual for software development teams. Object technology on its own cannot affect the human issues.

There is one last piece of good news. Infrastructure frameworks, which are expensive to develop, do not confer much competitive advantage to a business system. As a result, there is a growing market and a growing number of providers for them. These frameworks provide cost-effective reuse for difficult parts of the system.
7. Productivity.
Perhaps you have a productivity crisis and are hoping that an OO system will increase everyone's productivity so much that you can work through your backlog. While I would happily put my money on a set of experienced OO programmers against any other programmers in the world, that sentiment contains two important notions: experience and programming. Only a fraction of total development time is spent in programming. Requirements gathering, requirements analysis, system test, rollout, installation and training times are roughly the same. The programming time will only be reduced if you are using experienced programmers, with over 12 months of active OO programming behind them. Odds are that people will be using the project to learn, so they will be less productive. Therefore, let your anticipation of productivity gains be guarded.

8. Window-based user interfaces.
Object-oriented programming confers no benefit to the user's perception of quality while looking at the user interface. The same interface can be programmed in any of many languages, and the user will not care. In fact, the object-oriented, direct-manipulation interfaces up through the mid-1990's were almost all written in non-OO languages! Do not feel motivated to move to object-oriented programming just to get an "object-oriented"-looking user interface.
There may be some benefit toward programming the user interface using objects. In the early 1990's, development of window-based software was so labor intensive and variable that object-oriented programming was one of the few ways to manage it. User interface generators made simple interfaces easier to create and lowered the attraction of learning OO programming. "Drag-and-drop" interfaces made them harder to create again, and are more naturally suited to structuring the code with objects. This see-saw between making user interface programming harder, then coming out with tools to simplify it, may be expected to continue. The justification for introducing object technology just to simplify user interface development must be made in the context of available tools.

It is unlikely that you will introduce objects just to increase the morale of your developers. You might consider it a pleasant side-benefit. In this you are likely to be right. Most programmers find the programming environment so advanced compared to what they are used to that the task of programming will become more enjoyable. They will be encouraged further because they are using the newest, latest technology, which is important to people working in a technology-sensitive field. As is covered in a later chapter on staffing, a few of them will not enjoy it. There are two groups of people likely to have difficulty with the new technology: those who are expert in another language and cannot handle the stress of becoming beginners again, and those who have developed a dependence on strict waterfall, "give me the requirements, go away and let me program", process. Although most of the staff will enjoy the move, some will not.

10. Automated code generation.
Some tool vendors advertise that your programs will be automatically generated from the domain model, since objects are the same medium of description in both analysis and programming. On a recent project we estimated the amount of code amenable to automatic generation to be a disappointing 5-15%. This code comes from structural aspects of the business domain classes and from the user interface classes. The behavioral parts of those classes still had to be written by hand, as did all of the linkage between the user interface and all of the infrastructure. These comprised the bulk of the code and the bulk of the difficult software design.
So do not look to regain your initial costs by simply pushing a button and having systems generated from analysis models.
11. Software process.

Many organizations have difficulty getting different groups of developers to work together in a socially responsible and predictable way. They are still personality driven. Object technology will not fix that. If there were one place I would not recommend injecting object technology, it is into an organization late on a project, suffering from software process problems.

The reason is that object development requires more, rather than less, communication. One project described their experience as “living out of each others’ pockets”. Object technology emphasizes minimizing the impact of change, even if that means increasing personal and interprogram communication.

Therefore, do not look to object technology to fix your software process. Use it perhaps as an incentive to rebuild or build a more controlled software process.

One Line Summary of Possible Benefits

*Work toward responsiveness to variation-on-theme*

...
4. **The difficulty of achieving reuse among your developers.** Scale down your expectations, scale up your education program, and put checks into place. "Reuse" is too simple a word for the concept—it should be unpronounceably difficult, to give the sense of how hard it is to achieve.

5. **The difficulty of getting developers to follow a process.** Putting a process into place fails for two reasons—the process is probably wrong, and the programmers won't be able to (or just won't) follow it. Expect detours to your process.

6. **The difference between business modeling and software design.** Despite much wishing, it is still not the case that an "accurate" business model makes a "good" software design. Converting a clear business model to a good software design requires thought and creativity. The software design must address the additional constraints of modifiability and performance.

   Recognize that once you have an initial, clear, business model, you have still to undertake the creation of a good software design. If your designers work well, the final design will still be an accurate model of the business, but will also run fast enough and localize changes. It will quite possibly look different from the original business model.

7. **The usefulness of CASE modeling tools.** Modeling the domain is only a small portion of the total design effort (see possible benefit #10, Automated code generation). CASE tool vendors and researchers are very optimistic about the capabilities of their pet tools, but history is not on their side. Plan on getting only minor code generation from an object-modeling tool.

Here are some fairly realistic costs to expect:

1. **Training.** Spend 3-6 weeks of classroom training per developer.

2. **Experience.** Spend 6-9 months waiting for each developer to start earning their salary again, as they get the hang of thinking in OO tradeoffs.

3. **Tools.** Ignore your current investment in CASE tools for software development (consider ignoring upper-CASE tools altogether and save some money). Forcing a tool to do things it was not built for will eventually turn it into just a drawing tool at very high cost. Buy the right tool.

4. **Consultants.** Good consultants are expensive and fully booked. For just that reason, there is a market for "experts" with only 6-9 months of experience. Nonetheless, you need access to experience. That will cost you, either in making the mistakes and learning about them yourself, or hiring someone who can steer you away from them or help correct them. You spend the same amount of money either way, but the latter takes less time.

   There. The bad news is all out on the table, so you can assess it. How did you do? Are you perhaps in such pain that you have to try this anyway? Are you game to develop an organization that makes steady gains over several years?

**One Line Summary of Probable Costs**

*Budget $200,000, plus $6,000 and 6 months time per person*

**A Checklist of Non-Object Issues**
Actual project management is largely independent of object technology. Gathering and understanding requirements does not depend on the technology. Testing, documentation and deployment still need to be done. Twenty or more people working together need a project framework with a communications infrastructure.

Use your current way of working as a baseline. I provide a short review list of items that do not depend on object technology, dividing it into two parts. The first part consists of tasks which should always appear on your standard project list, and which you still have to do. The second part mentions tools and processes.

Allocate time for:
1. interviewing users and project sponsors to determine project requirements,
2. researching and documenting the existing system, listing all the interfaces that must be matched in the new system,
3. training the developers and managers,
4. analyzing the user wishes and requirements,
5. people to argue with each other about what they meant, or thought the other was supposed to do.
6. creating technical infrastructures for the operating system, the network, error messages, the user interface,
7. deploying a prototype and reevaluating requirements after the feedback session,
8. creating test cases,
9. integrating the work of different groups,
10. documentation,
11. making last-minute changes, just before delivery, as all the pieces are put together and last-minute insights about the system become apparent,
12. releasing an “alpha”, and a “beta” version of the product, and correcting each of those,
13. preparing training materials.

Other standard considerations:
1. Process needs: the form of deliverables exchanges, the form of interaction between the groups.
2. Social needs: team leaders, documenters, testers, team interrelationships.
3. Technical needs: versioning, configuration management tools, e-mail, LAN connections, and the like.
4. Diversity of skills: COBOL programmers, C programmers, network experts, database designers.

Those two lists should give you confidence in your ability to manage an object-oriented project, building on your non-OO background. Some few differences have to be taken into account, particularly team structures, as we shall see (Project Teams). However, much of what makes OO appear hard is bringing reuse, increments and iterations to the fore, plus the mythology that has grown around object orientation.

Object systems still have talk to legacy, non-OO mainframe system. This is not something that should break your project. It is a matter of designing an interface. Make sure you have a system architect who is familiar with both workstations and mainframes. If you do not have such an architect, or you have many, disagreeing architects, then you do have a survival hazard to fix, quickly.

You should consider restructuring your COBOL system prior to replacing it with an object-oriented system. Very powerful COBOL analysis and structure tools and services are available, making the task practical. As a result of the restructuring, you will find it easier to document the
interfaces to the COBOL system, and easier to separate modules for replacement. You may even reduce the size of the object-oriented revision, which will save you money.

Your architect may recommend that the mainframe non-OO code be developed differently than before. Several mainframe experts have told me that an effective server design is "object-centered". The server is organized around the structure of the data requests coming from the workstation. The data access requests mirror the structure of the classes, because the classes are generating the requests. Interestingly, this design recommendation did not originate with the OO developers, but the mainframe experts.

Mainframe and workstation developers do not speak the same language and are unlikely to negotiate well with each other. The architect or technical lead must be able to sensibly settle disputes over the placement of responsibilities.

**One Line Summary of Non-OO Issues:**

*Use your previous experience to avoid the most common traps*
3. Select

*Select for time-to-market.*

Your first project: not too big, not too small, not too important, not too unimportant. What does this mean?

The project should either deliver value directly or show you how the company will react to object technology. It must fit the history, temperament and capabilities of the company.

Consider two extremes, each faulty. On one extreme is a small, test project with just two people. That project will be fine for those two people; it will improve their resumes. It is unlikely to show whether the company ought to use object technology. It does not characterize your situation, so people are unlikely to pay attention to the results. On the other extreme is a company-critical project as a first project. It can be done, but the risks are high. You must take special precautions.

The project should be selected to reveal how the technology will be used, what problems you will encounter. It should be important enough to be noticed, but not so critical that the risks are deadly.

As all rules have exceptions, here are four complementary rules:

1. A project designed to get as much done as possible quickly should consist of 2 people, to avoid the communications difficulties that accompany OO projects.
2. A project to investigate the technology should have at least 4 people on it to reveal some of the communications issues that accompany OO projects.
3. A project to decide whether to shift the entire company to object technology must be serious enough that completion, or failure to complete, is a significant event to the company. Otherwise, its success or failure will be inconclusive.
4. A project whose failure would be significant needs special precautions and backup measures. It should not be allowed to fail because of neglect, poor management or insufficient funds. That would be a triple blow: the failure of the project, the failure of the management involved, and a failure to master the technology.

The best first intervention is to set up the project well. Therefore, I next go over topics you should address well before the project gets underway: selection of the project itself, selection of staff, language, tools, and methodology.
4. Effect

Plan the project by milestones

By now you have made your selection. To effect your project, you must create methodology, estimate, plan, set milestones, and start design.

METHODOLOGY

Methodology is, "how your organization repeatedly produces and delivers systems". It is

- who you hire, what you hire them for,
- what they expect from co-workers and pass them,
- what conventions they follow, and even
- what sorts of jobs you agree to do.

No one person should need to "read the methodology", because your methodology is the collected job descriptions of everyone on the project, what they do and how (see Figure 4-1). As a person learns their job, they stop referring to the methodology, except to check the project standards for the deliverables they produce. The rest is simply how they get their job done.

![Figure 4-1. "Big-M" methodology. Addresses all job descriptions.](image)

Major development houses use the term 'methodology' in that way, and it is a powerful way to think. I call it "big-M methodology". This chapter outlines a base "big-M" methodology for you, from which you can evolve your own, tailored version. Figure 4-1 and the following sections illustrate what I mean by a "big-M" methodology.

A second and weaker use of "methodology" is, "which author and drawing notation (UML / OMT / Fusion / Booch / Shlaer-Mellor) do you follow?" I refer to this as "little-m methodology" because each author covers only a part of what you need in your "big-M" methodology. Do not spend too much time worrying over your choice of author or notation. Your "little-m" methodology will not be a critical factor in your project's success or failure. Whatever you choose, you still have to decide who is going to do what activity to produce it, who will read the result, and what their interaction is. The "little-m" methodology, illustrated in Figure 4-2, is a fragment of the "big-M" methodology.

In this book, methodology always refers to "big-M" methodology unless explicitly noted.

---

1 Miriam-Webster defines "Methodology" as: 1) a series of related methods or techniques; 2) the study of methods. The Oxford English dictionary has only the second definition. Hence American and English speakers disagree on whether to say "method" or "methodology". I use the American form.
Your Big-M Methodology

Developing a big-M methodology is crucial to the success of your organization. The Eyewitness Account by Glenn House conveys as a undercurrent how their company evolved a methodology slowly, expensively, painfully. Exactly as I promise here, they did finally develop one, which is serving them well.

Your big-M methodology consists of the following elements, at least. Compare them to Figure 4-1.

1. **Roles** - the job descriptions you put into the ads when you need to hire more staff: JAD facilitator, project manager, requirements gatherer, business class designer, tester, program designer, programmer, writer.

2. **Skills** - the skills you expect those people to have when they answer the ads: facilitating, planning, programming, OO design, familiarity with tools.

3. **Techniques** - the techniques you expect your staff to use in their work: JAD (Joint Application Design), session facilitation, CRC (Class-Responsibility-Collaborator) card exercising, Smalltalk or C++ programming, domain modeling, use case modeling.

4. **Tools** - what tools the people use in their jobs, either within a technique or to produce a deliverable according to the standard.

5. **Job assignments** - how you assign people to multiple roles. In the methodology outlined below, one job assignment is called "Business Analyst-Designer", and consist of the roles: requirements gatherer, requirements analyst, business class designer, business class documenter. Another, "Designer-Programmer", consists of business class designer, programmer, software documenter, unit tester.

6. **Teams** - how you group the people. In the methodology outlined below, each "Business Design Team" has one business analyst-designer, 2 - 3 business designer-programmers, one database designer, one mainframe developer, and one system documenter. An "Infrastructure Design Team" has one lead designer plus 2 - 4 designer-programmers.

7. **Deliverables** - what you want each person or team to hand off to another person or team: use cases, class-, screen-, test- specifications, framework documentation, class diagrams, interface diagrams.

8. **Standards** - what is permitted or not permitted in the design and in the deliverables. These include notation, project conventions and quality concerns.

The notation legislates things like selection of:

- "Use cases" for functional requirements,
- UML or Fusion, or Shlaer-Mellor, or whichever notation for the object model,
- The programming language, its allowed variants and compilers.

Project conventions legislate things like:
4. Effect

? "Each use case has its name as the first line",
? "No many-to-many relationships on entity-relation models",
? "No use of 'friend' classes in C++",
? "Every program method or function has a public header description".

Quality concerns may legislate or simply raise awareness about things like:

? Completeness with respect to the requirements, modeling, documentation, and pro-
programming templates,
? Use of published design patterns in the design,
? Patterns of interobject communications in the design.

9. Activities - the activities and milestones of the different teams, how they fit together over time to produce the final deliverables. Aspects of activities are covered over several chapters in this book.

Do not look to the popular OO books for your big-M methodology. Quite rightly, a mass-
market book can only scratch the surface for what has to become your organization's personal way of working. Most OO book authors focus on one role, such as the designer, and then discuss notation for part of that person's deliverables, giving a few preferred technique fragments. The different authors focus on slightly different roles and techniques, leading developers to combine little-m methodologies on a project.

Most of the popular books are not intended as project management texts, and so include only a simplistic process model, along the lines of "1. Gather requirements. 2. Find the objects. 3. Draw a model. 4. Describe the behavior. 5. Code. 6. Evaluate and repeat." This is as useful as the one-page instruction sheet I was given for building a dune buggy:

"1. Take off old body.
2. Put on wiring harness.
3. Put on new body.
4. Attach controls.
5. Attach bumper cages.
6. Attach roof.
7. Drive away and enjoy your new dune buggy!"

Your software development process deserves considerably more attention.
A methodology is organization-personal. Each organization and company develops its own successful way of working. Different factors drive the development along different lines. Does the group do big projects or small projects, aggressively incorporate new technology or play it safe, have rapid or slow staff turnover? These drive whether you have a heavy or light methodology, with many job descriptions or few. These two likely methodology summaries for a small and a large organization illustrate the difference.

**Small-shop methodology.** "We never take on a project that needs more than 10 people. We put 2-3 people on it first, until the architecture has been tried and is stable. We add people only after each partition in the system is defined and a lead technical person is comfortable with its overall design or needs help. Each team is 2-4 people. There are no deliverables besides interface definitions and the final system, properly commented. Each team iterates at its own comfort level within a timebox. Risk/payoff lists are used to select each next activity."

**Big-house methodology.** "We use a project manager and assistant, JAD facilitators, requirements analysts, business experts, an architect, technology designers (network, database, server, OO, etc.), technical writers, testers, trainers. There are templates for each deliverable: use cases, business rules, domain class diagrams, interface charts, class and method/function specifications, test specs, test case results, etc. The following tools are used ... . Development teams
follow an incremental process with 4-month increments, with at least two full iterations per increment.

Which methodology is correct? Neither, or both. Each work under different circumstances.

**A base methodology for you to tailor**

A full methodology fills 300 to 1,000 pages, detailing roles, activities, techniques and standards. There are small organizations with small projects, and also large organizations with very large projects. There are S.W.A.T., investigation, production and full-commit projects (Project Purpose). There are as many methodology variants.

I present the following methodology outline for a medium-sized production project in an industrial setting. The characteristic of such are project are:

- 10 to 40 people total.
- 1 to 2 years duration.
- Time-to-market is important.
- There is a need to communicate with present and future staff, and a need to keep time and costs down.
- It is not a life-critical system.

It is a common sort of project, requiring trade-offs between complete, extensive deliverables and rapid change in requirements and design. I have kept the number of deliverables low, to reduce the cost of maintaining them, yet included enough to keep the teams communicating. I tailored job assignments and teams to allow the fluidity usually needed on this kind of project. Many other sorts of projects also need provisions for fluidity and can take advantage of this methodology.

**Discussion of the Methodology**

This methodology is designed to let the technical development team deliver quickly. It is targeted to speed the communication from requirements through programming, and back, without removing deliverables needed for future system evolution.

Some projects need more deliverables. A life-critical system needs to preserve more visibility into the decisions. A very long-running project needs to preserve intermediate information, as developers might never get to ask each other questions. Each document is expensive to create and maintain, which is the reason this methodology minimizes them.

Small organizations will want fewer job assignments, each containing more roles. Small organizations have extreme cost concerns, and rely on more face-to-face explanations between developers. Having fewer job assignments can sometimes mean fewer deliverables, which lowers costs more.

What you use on the first project will not be your final methodology. Rather, your first project will show you what you really need in the methodology, how you must amend your first one to suit your organization.

If you choose to separate the requirements from the design team, or the design from the programming team, you reduce the amount of parallelism and flux your organization supports. I dwell on this because team structure, creation of deliverables, and parallelism are items of contention between project management and software developers. Developers typically prefer minimal structure, few deliverables, maximum parallelism and flux. This reduces their work load and lets them get their work done faster. Managers typically prefer the reverse, which makes the project easier to monitor, plan and understand. It is not that one is correct and the other wrong. The team, monitoring, and deliverables structures determine the amount of parallelism and flux the project can support, or else are determined by the amount of parallelism and flux the project must support. When you choose one set, the other is no longer open to free choice.
The following quote from the Fusion methodology illustrates a common misconception and pitfall regarding team structure, parallelism and flux. I cite this passage because it illustrates so well a typical breakdown that causes difficulties on projects (see *Sentences You Hope Never to Hear*).

"The [Fusion] method distinguishes three stages:

? **Analysis** - which produces a declarative specification of what the system does

? **Design** - which produces an abstract object-oriented model of how the system realizes the required behavior

? **Implementation** - which encodes the design in a programming language

Fusion starts with an informal requirements document. Analysis transforms the information in the requirements document into a set of models that more precisely characterize the way the software system interacts with its environment. These analysis models are then input to the design process, which produces another set of models describing how the system is structured and how the system's behavior is realized in terms of that structure... Finally, the classes and methods identified during design are implemented in an object-oriented programming language."


The quote makes the development process sound much tidier than it actually happens. Taking it at face value adds a hazard to your project, that of thinking that requirements, analysis, design and implementation are easily separated.

Many projects find there a great deal of interaction between requirements, analysis, design, and programming, with fluidity in both the requirements and the design. Requirements shift while the designers design, often because of the designers' insights and questions. Further, most OO design modules consists of very few lines of code (often only 3-10), so that detailed class design is almost programming. This means that there is a shift in the understanding of the project on the scale of days and weeks. Iterative development by itself will not keep pace with these changes, as iterative development operates on a time scale of weeks and months.

The methodology outlined in this chapter supports the maximum flux by arranging only two people between requirements and code. These people share information on a daily basis, with no written deliverables required between them. The one spends time in meetings, collecting and examining information for easy transmission to the other. The other keep the detailed software design in his or her head. They negotiate over the business class definitions, one set of eyes focused on the business, the other focused on the software.

Finally, the methodology contains the incremental delivery strategy as a cornerstone, since it lets you make mid-course adjustments to everything else. At the end of each increment and at the end of the project, evaluate what you learned: your priorities, what you would like to keep, what change. That becomes your organization's personal methodology.

Here is the methodology.

**Roles, skills, techniques**

**Business expert.** An experienced business person. Knows how the business operates and can answer questions about how things are done, what is stable, what is changing, what is essentially attached to each concept, habit and term, and what is only conveniently associated with each.

Skills: Knowledgeable in the operation of the business and the business plans.

Techniques: none related to designing the new system.

**Usage expert.** An experienced user. Knows how the business operates from the point of view of the person who will be using the new software. Can say which system services are more
Important than others, which features can be dropped, what operational shortcuts can be taken, what alternatives can be created for procedures. Must know the user's job intimately. Is not an IS person who happens to have talked to a number of users, a manager who gives the software to the users, or the purchasing agent. There may be several usage experts for the several user groups targeted.

Skills: The users' job skills, ability to articulate their actions and priorities.

Technical facilitator. Knows how to conduct a group discussion session, whether requirements gathering, joint design or design review.

Skills: Working with groups.

Techniques: JAD (Joint Application Design) or Group CRC card design, design review facilitation. (CRC card design is described more in the Eyewitness Account below).

Requirements analyst. Knows enough of the business to examine the users' requirements statements and enough of the technology to ask for alternative requirements when the solution looks too hard. Examines and catalogs interfaces to the new system, both current and planned.

Skills: communication skills, thoroughness.

Techniques: common sense, domain object modeling (find the nouns, reduce them, check business rules, apply cardinality rules).

Architect. Knows how to design the system as a whole. Responsible for major subdivisions and interfaces within the system, performance targets, ensuring cleanliness and functioning of the overall system. Reuse advocate across teams. Works with the project manager to prioritize and organize the project plan. Does high-level design.

Skills: ability to evaluate the entire system in her or his charge, ability to make global and also detailed technical decisions.

Techniques: systems analysis, performance modeling.

Project manager. Knows how to gather and integrate information from all stakeholders in the project (executive sponsors, architects, developers, testers, etc.) and put it together into a workable plan. Knows how to fend off feature creep and other hazards of running the project. Responsible for the process, with input from the other roles.

Skills: motivation, observation, communication, planning.

Techniques: project estimation, management-by-walking-around, team building.

Lead designer-programmer. Knows how to create frameworks, knows the difference between strong and weak designs, can monitor and coach other programmers without demoralizing them. Reasonably good communication skills. Designs subsystems, applying design techniques to the requirements. Typically but not always the best programmer on the team.

Skills: framework design, class design, communicating with newer designers, programming.

Techniques: domain modeling, framework design, design by responsibility allocation (CRC cards), design by client interface, design by theory building, or even design by intuition.

Designer-programmer. Knows how to design a complete and programmed set of classes from requirements and a sketched design. May know how to design frameworks. Could be an expert designer with little interest in communicating and teaching.

Skills and Techniques: same as lead designer-programmer.

Design mentor. Similar skill set as lead designer, but with stronger communication skills. Responsible for improving the level of design and programming in the group, coaching novices.

Skills and Techniques: as for lead designer, plus ability to articulate design considerations, coach novice and middle-rank designers.

Reuse point. Either the architect or a designated designer-programmer with good programming skills. A part-time role. Identifies commercial classes to buy and classes being designed which overlap and should be reconciled.

Skills and techniques: same as lead designer-programmer.
**Writer.** Creates the external documentation, such as the class, screen and test specifications, and first-draft user manual.

Skills and Techniques: Good at writing, familiar with OO topics.

**Tester.** Knows how to create and run test cases, given either the requirements document or the class and screen specifications. Creates system test suites that can be run repeatably (regression tests).

Skills: creating test suites.
+ Techniques: white box / black box test creation, disturbance tests and test recovery.

**User interface designer.** Knows how to create easy-to-use user interfaces. Knows or can learn the UI standards. Enforces simplicity and consistency in the UI design. May have special training in gathering feedback from users on the interface.

Skills: able to learn the users' work needs and habits, able to evaluate and test the user interface design.
+ Techniques: user-centered design, low-fidelity UI modeling, videotaping- and questionnaire-based testing.

Other roles might be present, such as test manager, training and rollout specialist, help desk operator, database designer, network designer.

The above roles are standard, and fit to people one-to-one. There is one, less-than-obvious job assignment I have found to work well. Combine part of the designer-programmer role with the requirements analyst to makes the "business analyst-designer", or "analyst-designer" for short. Having a business analyst-designer simplifies discussion and speeds feedback from design to requirements, as you will see.

**Business analyst-designer:** Responsible for talking with the users, eliciting requirements, cataloging interfaces, as the requirements analyst above. Creates the initial design of business classes, and negotiates and reviews final design of business classes (Skills and Techniques: class design, CRC cards). May help create frameworks for the business classes. Is responsible for seeing that the class diagrams match the final code upon system acceptance.

**Tools**

Each role needs tools. Include at least these sorts of tools:

- ? versioning
- ? project tracking
- ? communications
- ? drawing
- ? programming
- ? performance measuring
- ? testing
- ? screen drivers for repeatable GUI tests

Work out the details of these tools and their templates as appropriate for your specific product and working environment.

**Teams**

The assignment of roles to people depends on the skills your staff already have, the communication strengths and weaknesses particular to your organization, and your ability to alter the existing organization. Teams are created both along and across system decomposition lines. Here are those based along system decomposition lines:

**Function Teams:** A function team identifies and delivers user-based system functionality. Each consists of one or more usage experts, part of a technical facilitator, one business analyst-designer, one to three designer-programmers, a UI designer, a tester, and half or one writer. A database designer is included if there is a significant database to develop. Multiple function teams share the frameworks for the business classes.

The business analyst-designer and designer-programmers on a function team work as though one person in several bodies. To keep communications fast and allow fluidity in the re-
requirements, design and program, no documents are required to pass between them. The analyst-designer is responsible for researching requirements and interfaces, passing on new information to the designer-programmers, perhaps several times a day. The analyst-designer will create the initial draft of the business classes, reflecting her or his domain knowledge. The designer-programmer starts work as soon as the functional requirements are "good enough to start". The designer-programmers alters the base design in negotiation with the designer-analyst to suit the evolving needs of the software. The analyst-designer reviews the final design, to reestablish validity with the user needs and business domain.

**Infrastructure teams:** The infrastructure teams create the subsystems that support the business classes: network, database, and user-interface support systems, for example. Each team contains a lead designer-programmer, several designer-programmers, a tester and half or one writer. The structure team is usually made from people having a strong computer science background, since severe, and often, subtle, technical issues will have to be addressed. An infrastructure team may be responsible for simulating and evaluating the overall architecture.

**External Test Team:** The testers create usage-based regression tests for the system. They may have close interaction with the function and technical teams, or they may work entirely from the technical specifications and documentation. Each way of working has its proponents.

Here are the teams that go across system decomposition lines:

**System Planning Team:** Executive sponsor, project manager, architect and one or more users. They identify and prioritize the work.

**Project Monitoring:** Project manager and each team lead. They check where the next risks are coming from and how to reduce them.

**Architecture Team:** Architect and lead designers. They ensure the overall system design stays coherent.

**Technology Teams:** One per specialty involved, consisting of the experts in each technology. Include the UI Design and the Programming Team, at least. Most projects will also have a Database Design Team, a Writing Team, an Internal Test Team. The purpose of each technology team is to set conventions and standards for their members, who are scattered across the Function and Infrastructure teams. For example, the UI Design Team makes sure that screens are consistent across Function teams in terms of the metaphors used, layout and behavior. The Programming Team makes sure that design, coding, and documentation conventions are set and followed.

These teams imply a lot of communication. That is normal. The OO teams will see additional communication, because of the frameworks being developed. A framework typically affects people in several teams.

**Ownership**

Ensure that each deliverable has an owner. This means that each function to be delivered to the end user has a dedicated owner, and each class also has a dedicated owner (see **Project Teams**, and **Risk Reduction Strategies**). The two may overlap, but there must be someone to answer for the ability of the system to deliver any given function, and some one person to answer for the integrity of each class.

Ownership may be matrixed, with the same person owning both a function and a class. This is illustrated in Figure 4-3, below.
4. Effect

Deliverables

Although each team has deliverables, I shall only describe here the ones for the Function Teams.

Function Team:

? The requirements document. This includes the system purpose, the use cases, the business rules and relationships that must be preserved in the design, usability and performance requirements, definitions of needed interfaces to other systems or subsystems. This document is produced by the business analyst-designer and the writer. The purpose of this document is to communicate with the executive sponsors, the user community, the External Test Team, and the Function Team over time.

? The user interface design document. This initially includes a description of the user metaphor, each screen's purpose, the navigation between the screens. Over time, the details of each screen are added, and details of error conditions. This is produced by the UI designer, analyst-designer and writer.

? The object model document. This initially includes the major partitioning of the system and of the subsystem, each with its purpose, responsibilities and interfaces. Over time, the purposes, definitions and examples of use of each framework and class are added, along with a checklist of which major components are used for each use case. This is produced by the analyst-designer, designer-programmers and writer.

? Specifications for other teams. These include specifications for mainframe or other special programs that need to be written, or specifications from the Function Teams to the Infrastructure teams. These are produced by the designer-programmers and the writer.

? Usage manual. This describes how the users will use this part of the system. It is produced by the entire team, and is reviewed by the users, and used for external test and also to prepare training material.

? Code. This comprises the source code and the compiled, bound code for delivery.

? Test cases. These are the regression tests applied to every class, subsystem and total system.

It is possible for all deliverables to be started and undergoing change at the same time. The requirements document may still be fluctuating while business design is underway (see "Monitoring", below). How much parallelism and simultaneous flux you can tolerate in your deliverables will depend on your team structure and the interteam communications (see "Activities", below).
Standards

Notation standards
? Use Smalltalk, using the proposed ANSI subsets where possible, for all of the business logic on the workstation. Use C on the Unix server. Use COBOL on the mainframe. (This is just an example. Another company would say, use the ANSI-proposed C++, or Java, or Objective-C, or whichever language they preferred.)
? Use OMT until the forthcoming Unified Method Language is stabilized, then switch to that.

Design conventions
? The development team agrees upon and adopts design conventions for their deliverables (see, the C++ program design conventions in Jeremy Raw’s Eyewitness Account, and the modeling conventions of the Brooklyn Union Gas project).
? Other design conventions may require or preclude selected modeling techniques, programming language facilities, user interfaces styles and writing styles.
? The book Design Patterns (Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1994) might be required as a reference for common design problems. Where possible, those design solutions are preferred.

Formatting standards
? A standard template is created for the elements of each deliverable: use cases, business rules, screen navigations, interfaces, class definitions, function definitions, etc.
? Programming standards describe what comments are needed with every function and class.
? New templates may be created on the first increment of the project, by joint agreement of the project teams. Old templates may be taken from existing standards.

Quality standards
I include no quality standards here, but suggest you declare a small set based on your past experience. Your team will discover more. Keep the set small (see the Eyewitness Account by Glenn House). Avoid gratuitous requirements along the lines of “every method must have 10 lines of code or less; every class must have 20 methods or less.” These only push code elsewhere, they do not improve the quality of your system.

Activities

Staging: Plan for a user-useable release every three or four months. Each release is one “increment” of the entire system. Each increment contains whatever each team feels they can deliver in the period, as orchestrated with the System Planning Team. Each increment may include time for several revisions (see “revisions”, below). Incremental development is described in more detail in the chapter, Incremental Development.

Revision and review: Each increment contains several iterations, and each iteration consists of construction, demonstration and review. A shallow iteration is construction, demonstration and review of selected aspects of the team’s assignment, perhaps just the screen design, just the business behavior, or just the database access. A deep iteration is construction, demonstration and review of a slice of the team’s assignment, working from end to end. A deep iteration for a Function Team has user function running from the user interface through to the database and back.

Use three deep iterations within each increment. Any number of shallow iterations are permitted. The purpose of the review at the end of the first deep iteration is to let users re-check their assumptions, and react to the basic design. At the end of the second iteration they get to double check the design and add fine-tuning. They may remember additional assumptions, but are discouraged from making major changes. The third iteration results in the subsystem going to test and delivery. The scheduling of each iteration is left to each individual team.
4. Effect

**Monitoring**: Monitor each team's deliverables with respect to both progress and stability. Progress is measured in milestones, which are sequential; the stability states are not necessarily sequential.

The progress milestones are:
1. Started
2. Review 1
3. Review 2
4. To Test
5. Delivered

The stability states are:
1. Wildly fluctuating
2. Fluctuating
3. Stable Enough To Review

A common sequence of stability states might be: 1-2-3-2-3-2-3. A deliverable rated as gone "To test" may get labeled as "Fluctuating" if some unexpected problem is encountered which questions the design or the requirements.

**Parallelism and flux**: Any dependent task may start as soon as all the predecessor deliverables are Stable Enough To Review. The team structure given above is tuned to permit this sort of parallel flux. The designers may start working out their basic needs, but should not start serious design, while their requirements are Wildly Fluctuating. As soon as the requirements for the structure or function reach Stable Enough For Review, designers may begin parallel design. As soon as the design reaches that state, serious programming may begin (investigative programming may have been done earlier, to assist in creating the design). As soon as the business object model is Stable Enough for Review, database design may start. And so on. This maximizes the parallelism in the production of a release, while permitting flux in the deliverables.

The function and infrastructure teams work in parallel. The project monitoring and architecture teams review their work plans, stability, and synchronization points.

One Line Summary of Methodology

Learn how your organization has learned to produce systems

**Estimates**

The best way to estimate your project is to ask someone who has done it before. I assume you would have done that already if you knew such a person. Usually you find yourself in a new situation, at least regarding team and tools. This chapter contains some estimating and planing tips that have helped other projects.

... 

One Line Summary

The work estimate and the staffing plan are connected.

**Plans**

Not having a project plan is embarassingly common.

...
4. Effect

One Line Summary

*Have the project show working results every 3-4 months.*

Here is an Eyewitness Account of a planning session.

... MILESTONES

As you work, there is an awkward sense that everything depends on everything. It seems that requirements cannot be completed until the second deep iteration, that UI design and component design are not complete until programming is complete and the whole thing goes to test. This hardly makes for a satisfactory project plan.

... One Line Summary

*Plan by milestones, sufficient results to let the next work start.*

MEASUREMENTS

"the researchers have not yet been able to find measures that are practical, scientifically sound (according to measurement theory principle), and cost-effective to capture and analyze."


... If you are interested in lowering the complexity of the system:  

*Hire a better requirements analyst to find simpler requirements,  
hire a better designer to design a simpler solution.*

TAKE THE TIME TO DESIGN

Team leaders complain about absence of design in all languages. Take the time to design - what

Note the presence of other key success factors on the first project: user involvement, architecture, incremental development, expert teams.

...
5. Correct

Learn what you don't know you don't know

The ideal project does nothing wrong, it simply delivers the system. However, that kind of project does not show you how to get past your own, unique difficulties. For that reason, I have chosen for a study project one that went wrong in worse ways than your project will, yet turned itself around to deliver a useful system with skilled staff. This is Project Ingrid. It is ideal in the sense that you can learn the most from it, and model your own recovery after it. I have myself applied the lessons from this project, to good effect, and think you can, too. I use it as a starting point, to let you permit yourself mistakes, and to motivate you to adjust your project according to those mistakes. It is by correcting mistakes that you survive.

... OneLineSummary

Work in increments, focus after each, rebuild with agility

MANAGING PRECISION, ACCURACY AND SCALE

Oddly enough, your team can get into trouble from being too precise, just as from being too vague. Define levels of precision, accuracy and scale, even imprecisely, to improve the ability of your group to keep discussions on target and produce the information needed at a particular moment in time. Keep relevance in mind.

Let me clarify what I mean by precision, accuracy and scale. Relevance is what items you decide to include in the discussion. Precision is how much you care to say about a particular item. Accuracy is how correct you are when you say that much about the item. Scale is how many actual items you collect into one term. I use the acronym PARTS to remember these: precision, accuracy, relevance, tolerance and scale.

We are familiar with these terms from everyday life. When we lay out the plan of an office, we choose to talk about the size of the furniture, rather than weight or color. We have selected for relevance. In software design, we might choose the domain classes and omit the others for discussion.

For the purpose of a conversation, describing a bookshelf to the nearest foot is precise enough. So we refer to "the six-foot bookshelf", when the bookshelf is actually six feet, one and a half inches long. If we had said, "the four-foot bookshelf", we would have been inaccurate, describing to the nearest foot. Working on the initial design of a house, it is useful to work to the precision of the nearest foot. When cutting the lumber, we need to be accurate to greater precision, perhaps the nearest inch, half-inch, or eighth of an inch. Accuracy is meaningful with respect to the precision needed at the time. There is no point in worrying whether the second decimal place of the bookshelf size is .04 or .05 if we have not yet decided if it is to be 4 feet or 6 feet long.

Scale is changing how much we include in one thing, in order to cover more material in the same space. We don't have space on the paper to draw the house at a 1:1 scale. To fit the house onto the paper, we use a small box to mean a large space. Scale interacts with relevance and precision. Once we select to draw the whole house on a single piece of paper, we become limited in the precision we can use, and in the space we have. So we omit from the drawing everything whose overall size is below our level of precision. Thus, paperclips would not show up in the drawing, even if we wanted them. Typically, at this level of scale, paperclips are not relevant. We are interested in other relationships.

On a software development project, these terms have analogous meanings and analogous uses. Every deliverable, from requirements to project plan to hardware and software design, can be described at various scales, to varying precision, with varying accuracy. Working at the inap-
propriate one means working at with inappropriate information. Let us look at these more closely.

Precision is the most significant of the three. It defines the amount you care to say about a particular item, so it controls how you connect work on the project. Most deliverables have three or four levels of precision.

- For the project plan, the **first level** of precision is to state the releases, what basic functionality is in each, and what the development dependencies are between them. The **second level** of precision is to state clearly which functions are delivered in each release, which team is developing each, and to name the specific dependencies between the teams. Delivery dates for the next several releases fits into this level. The **third level** of precision state the milestones and their dates of each deliverable involved in the release.

- For the functional requirements, the first level of precision may include just the name of the function or use case being delivered, the user group expecting it, the anticipated value, urgency and development cost. The second level of precision may include details about the basic usage scenario, the business events and business rules involved, frequency and performance. The third level may include all other details about the function's requirement.

- For the object design, the first level of precision is to state at least the name and main responsibility of each class. The second level is to state detailed responsibilities and collaborations used. The third level is to state attributes, relations, function signatures. The fourth level is the code.

It is especially important to note that when someone draws (or shows) a class diagram describing relations, functions and attributes, they are operating at the third level of precision. There are times when trying to create that diagram, or showing that diagram, is inappropriate and less precision should be used. These occur both at the beginning of the project and when presenting it in summary form.

For a given amount of precision, there is a desired amount of accuracy, and the question, "when is it good enough?"

- During initial development, each deliverable needs only to be accurate enough at a given level of precision to permit work to start on the next level. Striving for too much accuracy early can waste time, since your work at higher precision will turn up new information, which may alter the previous description: Working through the various failure scenarios and error conditions of a use case may turn up new use cases. Working through interteam deliverables may change the development priorities. Working through a design may change the originally planned collaboration structure. The earlier, low precision, work needs to be done, and it needs to be done just accurately enough to permit the next work to be estimated and coordinated.

Accuracy feeds the iterative process. The purpose of an iteration is create the best deliverable possible at that time, i.e., making it as accurate as possible. Luke Hohmann, in his Eyewitness Account, uses the metaphor of cooking pancakes. When he says he "just wants to see a burnt pancake", he is saying that accuracy is not needed at this time, movement through the process is. When his team asks him, "How edible do you want it to be?", they are asking for clarification about accuracy. Edibility is a good measure for accuracy, because people have an intuitive understanding of its meaning and consequences.

Scale is a matter of getting more of the system into view at one time. It has three aspects:

- Using lower precision. Just as a 1:30 or 1:100 scale drawing of your office will not show the keys on your keyboard or support boards on your bookshelf, so the 1:30 or 1:100 scale drawing of your software design will not show all the attributes of each class. This way, each item shown can be shown smaller.
6. Reflect

? Selecting items for relevance. To get more of the domain model into view, you choose to omit the non-domain classes entirely. This is similar to a cartographer omitting bus lines and houses on a city map.

? Bundling multiple items together. Numerous component bundling schemes have been proposed in the literature, Booch's class categories, Shlae and Mellor's domains being two of the better known. Other suggestions are ensembles, subsystems, and fat objects. There is no consensus yet as to what is the best way to bundle classes. Each of those ways appears to work adequately. In project management, a release is useful unit bundling many functions and deliverables, and use cases may be clustered to gain an overview of the function sets.

Once the project has more use cases, dependencies or classes than comfortably fit on a few pages, developing a larger scale of view becomes necessary. Unlike precision and accuracy, it is not necessary to the development process per se, but it is necessary to understanding the overall shape of the project. As the project becomes larger, it becomes more necessary.

Managing work according to precision and accuracy

There is no point in trying to get the milestones right until the release dependencies are documented. Similarly, it is counterproductive to argue over the creation behavior of a link between objects before subsystems have been carved out or basic responsibilities assigned. How you sequence your work affects the progress of the team.

The value of monitoring precision was not clear to me until I worked with groups of people who insisted on working at high precision too early. The domain modelers were interviewing business experts, who had been gathered at great expense and under time pressure. The domain modelers had decided to capture the information at very high precision. If they drew a line one way, it would have one meaning, drawing it another way would give a slightly different meaning. While the expert was talking, the modelers could not draw any line at all, because they did not have enough information to decide between the two styles of line! As they pursued the information they needed to the precision they needed, the conversation got sidetracked and the attendees lost the overall conversation. At the end of the allotted time, the business experts vanished, and they had not gotten the critical information they needed.

These modelers' priority should have been to capture that information which they could only get from these experts gathered in this room together, and which they needed to proceed. They could have and should have worked at a lower level of precision. It would have allowed them to cover all the territory they needed to cover, and mark places where they needed to come back to.

Other than modelers like these, it is common for developers to program or draw class diagrams immediately, before having worked out a basic design. In their haste, they forget that the less precise design can be reviewed much more easily, sooner, and changed more easily, if worked out only to the level of name, responsibility and collaborators.

It would be soothing to say at this point that one should always work at the lower level of precision before working more precisely. That is not the case, however. History has not been kind to those who developed all the deliverables at large scale and low precision before touching the next scale and precision. Too many surprises lie in store, and it is to catch those surprises that we use prototypes, iterations, and increments. Therefore, managing the level of precision employed at any moment takes some effort.

Here is a sample ordering of work illustrating managing the level of precision. It fits within the first increment. It shows how low-precision deliverables can free teams to work in parallel, optimizing the use of their time.

1. Collect the functional requirements to the first level of precision: user goal and user group. Estimate urgency, value and cost of each (note that this information is medium-accurate and low precision), as well as technical dependencies. This gives you the system spec at the first precision level.
2. Estimate how you will release the system, ordering the releases by technical dependency, priority, and ease of construction. This gives you the project plan at first precision level, medium accuracy ("edible"), and scale of perhaps 1:100.

3. At this point you can create teams by function and release, and let them work in parallel.

4. Let the teams gather requirements to the third level of precision, capturing main and failure scenarios for the functions selected for the first release or first two. Capture, also, external systems for which interface drivers will have to be developed.

5. The teams can now be split into management, user interface, domain, and infrastructure, and may work in parallel. The management team will create dependencies and milestones. The user interface group, domain modeling group, and technical infrastructure teams all have enough information to start their work. Each team moves its deliverables sets to the next level of precision.

6. If some aspect of the project needs a quick investigation, this should show up about now. Some people like to always do a quick prototype about here to discover what sorts of surprises are waiting ahead.

   Decide how much precision is appropriate for each discussion. Move from lower to higher precision over days and months generally to master the intellectual complexity of your task. Dive briefly into high precision, as in a prototype or detailed examination, to evaluate risks and uncover unknowns. Use the acronym PARTS to remember the issues.

   One Line Summary

   Decide when to work at low precision, when high precision.

INCREMENTS AND ITERATIONS

Increments are so important that we should become clear as to what they are, and are not. Here are three questions to consider.

? What is the difference between "increments" and "iterations"?
? What is the purpose of doing increments?
? What is the purpose of doing iterations?

The short answer is:

? Increments let you fix or improve the development process.
? Iterations let you fix or improve the quality of the system.

Although both are necessary to produce a good system, increments are usually necessary just to survive.

... 

Increments and V-W Staging

Incremental development is a staging strategy in which portions of the system are developed at different times or rates, and integrated as they are completed. It does not imply, require or preclude iterative development or waterfall development, which are rework strategies. The alternative to incremental development is to develop the entire system as one lump deliverable.

... 

aOne Line Summary

Increments let you improve your development process
6. Reflect

**Iterations**

Iterative development is a **rework scheduling strategy** in which time is set aside to revise and improve parts of the system. It does not presuppose incremental development, but works very well with it.

One Line Summary

*Iterations let you improve the quality of your product*

**Combining Increments and Iterations**

Increments and iterations do not require each other, but they work well together. A useful strategy is to create, for the end-user functionality, 4-month increments, each with 3 iterations (see Figure 5-8). The infrastructure has its own increments, but it will have iterations that span the function increments, because feedback to the design of the infrastructure comes from delivering functionality. Here is how it works:

One Line Summary

*Use iterations within increments to reduce risk*

**YOUR PROJECT INCREMENTS**

This section shows a Production project as a first increment, a second increment, and a succession of next increments. The first two are different, the rest should look fairly much the same. Each increment is approximately four months long.

**Increment 1**

The purpose of the first increment is to:

1. Establish an architecture for the software,
2. Deliver feedback to the project sponsors and users,
3. Develop good patterns of programming and designing,
4. Find the boogieman and learn WYDKYDK.

One Line Summary

*Establish the architecture, deliver one function, find the Boogieman*

**Increment 2**

The purpose of the second increment is to:

1. Repair the major mistakes discovered in the first increment,
2. Establish or tune the software architecture,
3. Create a process and teaming that works for you,
4. Settle on good development habits.
5. Deliver more function to the users.

...
One Line Summary

*Pause & learn, revise & persist*

**Increment N**

. . .

One Line Summary on Increment N

*Always recall how you became successful*

**User Involvement**

Your project will survive without direct user involvement, but chances are that it will not be considered a success without direct user involvement. The best way to discuss this is to review the paper by Mark Keil and Erran Carmel, "Customer-Developer Links" (Communications of the ACM, May, 1995, pp. 33-44).

. . .

One Line Summary of User Involvement

*Get the real users involved*

Here is the Eyewitness Account

. . .

**Project Teams**

One Line Summary

*Ensure each deliverable has an owner*

**Semitechnical Issues**

**The Domain Model**

Prepare for heated discussions about "modeling". Experts differ at the moment as to whether enterprise modeling, entity-relationship (E-R) modeling, domain modeling, or even business process reengineering are essentially the same, differing only because of old development habits, or are fundamentally different. If you are not careful, you can sink a great deal of time and money into one or more models without getting the benefit back. Here are four hazardous situations you may find yourself in:

- The object and database modelers become enemies. The object modelers claim that their model is fundamentally different from a data model, while the data modelers claim that the two are fundamentally the same but that the object modelers create a poor model. The result is that they cease to communicate. My experience is that the structural part of the object model is fundamentally the same as the conceptual data model, and must be so. This is a position I retest each year, because of the strong reactions it provokes. It appears that the biggest difference is between data modelers who model the business in conceptual entities, and data modelers who model data on the disk. The domain object model most strongly resembles the conceptual data model.
The programmers ignore the domain model. While your business experts are carefully modeling the objects in the business, your programmers create whatever they want in the software. In the end, the programmers win, since their code runs your business. If the program code does not match your business, your system is basically flawed. Create an "audit" task in your development process, to resolve differences between the two.

You are defining the entire business model at one time. This is like trying to boil the ocean. History has not been kind to organizations who have tried to model the enterprise in one go, for two reasons:

- There is no feedback to the enterprise modelers from the programmers as to how well they are doing. Of the many possible, valid models of the enterprise, some transfer well to software, and some do not. Since the program code and the domain model must match, it is important to get feedback to the modelers continually.
- The enterprise keeps changing. By the time the full enterprise model is complete and ready to hand to the program designers, the rules have changed. Since change is inevitable, create a model/design/build process that gracefully accepts shifts in the business.

You try to convert the domain model directly to a software design. There are many valid models of the domain. Quite easily, the first domain model you create will not make a good software design. On the other hand, whatever final design you create must be a legitimate domain model. It is straightforward to convert a good design model to a readable business model, but not straightforward to convert just any readable business model to a good design model. Beware any advisors who advocate the latter path.

How can you create a reasonable domain model? Incrementally, with audits, and using a common domain model.

**The common domain model**

The program code must represent a valid business model that is also a good software design. That fundamental requirement drives the rest of the discussion.

First, create a high-level model of your business. This should take no longer than two to four weeks. Use a skilled object modeler and three to six business experts. The business experts do not need to know anything about objects to begin with, although they will become skilled by the end of the work period. Using use cases, CRC cards, semantic modeling, or some other techniques, create a model of the business using not more than a dozen really key classes and several dozen supporting classes. Identify the most important four, six or eight classes. This small model serves as a guide as you evolve and detail the model. The main 4-8 classes are the key abstractions that will show up over and over. Refer back to this initial model as time goes on.

Second, decide what your first release will contain. Model just enough of the domain to develop that release. This first model may be built by your business experts with an OO-skilled technical facilitator, or by your OO analysts, depending on your teaming structures and skill mix. The result of this work is conventionally called the analysis model. I refer to it, or any similarly created analysis model, as version 1 of the common domain model.

Third, let your OO software designers and database designers create version 2 of the common domain model. The requirements for version 2 and every subsequent version of the common domain model are that:

- It must be a good software design;
- It must be a valid and understandable domain model.

The relationship between the later versions of the common domain model and your programmers' design model is that the design model may contain extra lines, extra data or methods,
extra classes to clarify details of the software design. However, the common domain model should be obtainable by erasing the non-business terms, collapsing communication routes that run through non-business objects.

Version 2 of the common domain model does not have to look like version 1. It does, however, have to pass acceptance of the business experts. From this point on, the business experts serve as advisors on the common domain model.

Fourth, as the design and programming progress, late-breaking design issues will cause the OO designers to change the common domain model. At every point, it must be a good software design and a valid, understandable domain model. The business experts must concur that the model does not violate the rules of the business.

Fifth, audit the program code periodically. Programming almost certainly will change the model. Most of the time, you will update the common domain model to reflect the program code. Sometimes you will find that the program code is simply incorrect as a model of the domain. Change the program, get it back to being a valid domain model.

Repeat this process for every release. The second time through, the business modelers can take advantage of knowing what changes were made in the first release.

When you put your system into production, the program code will match the common domain model.

**Why are there multiple valid domain models?**

There are many ways to describe the world. It is not that some are “wrong” while the others are “right”, it is just that there are many ways to characterize information. I am often asked to give an example of a domain with multiple valid domain models, whose designs differ significantly. In the interest of stimulating argument and comparison, here are three I have encountered:

1. Checkbook entries. Perhaps you are not a bank, but your system needs the concept of a checkbook, keeping track of entries. You need a dozen different kinds of deposits and about as many kinds of withdrawals. Your users must be able to void out a checkbook entry after creating it.

   - Does your domain model include “deposit” and “withdrawal” as kinds of objects?
     - Probably so.
   - How about “checkbook entry”?
     - Also probably so.
   - Is there a different model that avoids checkbook entries altogether?
     - Yes, you could model the system with transactions and logs of transactions, instead of using a checkbook.
   - Do you model a void of a deposit as a deposit or as a withdrawal?
     - I have found that the answer to this varies considerably with the person being asked, and there are many more than just two answers. I was not able even to understand some of the answers, so I cannot list them here. Try this out on your friends and colleagues, and see what you come up with.
   - How would you place a void of a void of a deposit?
   - Is a “voided entry” a subclass of entry, or is it so different that it is simply a separate kind of object?
   - Do you display a voided deposit in the “deposits” or “withdrawals” column?
   - What will you do when the users change their mind about whether a voided deposit should be considered a deposit or a withdrawal?

When we first encountered this problem on a project, we created the class, “CheckbookEntry”. We gave it two subclasses, “Deposit” and “Withdrawal”. There were about a dozen classes listed under Deposit, and a similar number under Withdrawal (see Figure 5-9).
6. Reflect

There was no debate about this initial design. Everyone considered the model obvious and true. Then we encountered the voids. Each week, as we interviewed more people, we got new and different answers to the fourth question, to the point that we felt we could not trust the stability of any answer. It was, however, clear that the users wanted to see a voided deposit in the deposits column.

We decided in the end that while we could trust that the two dozen kinds of checkbook entries would continue to exist, there was nothing intrinsic or stable about the classes Deposit and Withdrawal. Each entry had to be able to answer whether it was a deposit or withdrawal, and we were clearly going to change our minds quite often as to how a voided deposit would answer that question.

Eventually, we created about two dozen subclasses under the CheckbookEntry class, one for each entry type, including voided entry (see Figure 5-10). Rather than locking the answer to “Are you a deposit?” into the class hierarchy, we gave each entry type a method to answer whether it was a deposit or withdrawal. The reason for moving the question from the class structure to a method is that changing the class structure is a serious change to the system, while changing a method's answer from “true” to “false” is minor.

Once we created this design, we stopped worrying about the answer to the question, “is a voided deposit a deposit or a withdrawal?”. That could change every week without holding up the project. To this day I do not know what the final answer came out to be, nor do I worry. The software design was adequate, timely, stable, efficient and a reasonable model of the business.

Figure 5-9. Initial class structure for checkbook entries
2. Organization structures. An organization consists of suborganizations, down to a department, which contains people. There are many similar kinds of organizations: legal organizations, sales organizations, corporate organizations, etc. Is it better to model the organization as a recursive structure or to model the levels of the organization explicitly?

Both are valid models of the domain. Some analysts prefer the recursive structure, because it gives a simple and general model. The experienced designer-programmers I know prefer the explicit structure, because it gives an efficient software design. They say that by the time you encode the allowed behaviors of the different organizational levels, you have done all the work of writing it explicitly, but hidden your solution so that it is harder to see and change.

On the project, I really wanted to use the recursive solution - because it is universal, beautiful, and so cool. Hearing from the developers that it would also be harder to understand and maintain, we did a careful impact analysis. In nearly all cases, the explicit solution showed itself easier and simpler than the recursive structure, both to program and to modify. The only case in which the recursive solution worked better was the least likely and least frequent change situation. Reluctantly, I decided not to use the recursive solution. Since then, I have learned to prefer the explicit solution over the recursive one, for reasons of design and maintenance. Both are valid domain models, so I shall have no trouble revisiting the decision on a future design.

3. Flavors of soda. A restaurant serves many kinds of food and drink. These can be modeled as classes and subclasses. There is, for example, the Pizza class, with possible subclasses for special products. There is also a class, Soda. Does Soda have subclasses for Coke, Pepsi, Sprite, 7-Up, Root Beer, and diet of each? Or does the class Soda just call a name to differentiate the particular sodas? After you decide your answer, consider whether the nature of your business makes any difference in your answer.

One Line Summary of the Common Domain Model

Integrate domain models while delivering increments

PolyBloodyHardReuse

Why did I pick this term PolyBloodyHardReuse - why not just say "reuse"? Because there is an ingrained tendency to relate the difficulty of a topic to how long its word is. The topic of reuse is so diabolically difficult that the short word "reuse" actually gets in the way of dealing with it. If we cringe at "multiple inheritance", "genericity" and "polymorphism", we should cringe more at polyBloodyHardReuse.
... 

One Line Summary of PolyBloodyHardReuse

Improve the attitude; if it hasn’t been reused, it cannot claim to be reusable
6. Reflect

Learn what you wish you had known

Pretend for a moment that you have now completed your OO project, and look back at what you just experienced. What do you wish you had known about before? What would you say are the actual costs and benefits? What advice would you give yourself, if you could travel back in time to the start of the project?

...
7. Expand

*There are new traps for bigger projects*

**YOUR FIRST BIG PROJECT**

... 

**One-line summary of Your First Big Project**

*A big, OO project is more a big project than an object project.*
Here is the Eyewitness Account by Glenn House, recounting their experience in moving a large system to C++.

"10 Lessons The Hard Way" (Glenn House)
Glenn House, Vice President, Mentor Graphics

TRAINING THE TIDAL WAVE

We are about to revisit the bleak subject of training.

One-line summary of The Tidal Wave

Be creative to manage the next 200,000 trainees.

PRODUCTIVITY

This chapter contains some productivity models that can help you select between alternative project staffing strategies. I separate the discussion of productivity from that of estimation (see Estimates). This is because two projects with identical people will produce quite different results based on their team structure, their communications habits, their morale, when and how they come onto the project, the project staging strategy, and even the system design itself. As Jim Coplien describes in his Eyewitness Account, those factors interact with each other to push the productivity curves around. Typically, just two or three factors will dominate the productivity you get. I have picked out some of the dominant ones. You will find it useful to detect which ones drive your project.

Size

Functioning properly, a large team can do more than a small team. In The Mythical Man-Month, Fred Brooks gives a simple calculation that shows that sometimes a project requires a very large team. Here is that calculation recapitulated, and then generalized.

You might be able to get just the very best people to work on a small project. As soon as you need more people, the odds of your being able to staff with only the best people diminishes. The actual difference between people is marked, and you might notice a difference of 5, 10 or even 20 between people. This decreasing of average skill as the project grows I refer to as skill dilution. Fred Brooks used an example of 1,000 people on a project, with a skill dilution factor of 7 (that is, the average contribution per person on his postulated 1,000-person project is only 1/7 of that on a small, top-notch team).

The curve describing skill dilution should drop steeply at first and then flatten out. This captures the notion that the very best people are both good and rare, and that the group will slowly tend toward an average. I modeled with various curves, which gave similar results. I use here the curve $\text{Average Skill} = 1/\log(n)$, where $n$ is the number of people on the project. This is a nice, simple function which captures the notions just described. I omit adding the various constants and factors that are there for any one project, since there is so much variation between projects.

The other important factor is communication overhead. As the project grows in size, people will have to communicate with more people. Brooks again used a productivity drop to 1/7 in each his 1,000 person project. I use the curve $\text{Productivity} = 1/\log(n)$ again, based upon a hierarchical communication structure. Again, I omit constants and factors.
The two factors combine to make total project productivity = \( n / (\log(n) \times \log(n)) \), which is pictured in Figure 7-1. While the curve is obviously a rough estimate, it illustrates two things that are critically important:

- as you add people, at some point the total productivity goes down;
- as you add people, eventually the total productivity goes up.

The first point, which matches the dip in the curve, I discuss in the next section, Staff Skill Mix. For the moment, I want to notice that the total productivity eventually goes up. This is because the number of people contributing adds up faster than their contribution getting smaller. So Brooks is right. When you have a large enough project, you need a really large number of people; as you add people, eventually the total productivity goes up - as long as you manage them adequately.

The main lesson of figure 7-1 is that even with radically diminishing productivity per person, a properly functioning large team can get more done than a properly functioning small team. The operational phrase is "properly functioning". Many of us have seen an improperly functioning large team manage to get less done than a decently functioning small team.

Now for that dip at the beginning of the curve...

**Staff skill mix.**

A very good developer may produce function at several times the rate of a medium developer, who may produce at several times the rate of a beginning or poor developer. Let us say that a beginner works at one arbitrary unit of rate of production of function, an average developer works at 3 units, and a very good developer at 9. These numbers are rough but adequate for our purposes.

Six very good developers therefore work at a productivity rate of 54 units, which is the same as 18 average developers, not subtracting for communications overhead. It is the same as 27 mixed trainees and experts, again not taking communications into account. Which would you prefer, a project with six very good developers, 18 average developers, or 27 mixed trainees and experts?
This is the dip leading the curve of Figure 7-1. The model is too rough to be precise about just when the curve goes down and then back up again. It indicates that for small and medium projects, skill dilution alone hurts you more than the extra developers can make up for!

And now for the bad news - you may not be able to get your hands on six very good OO developers. To make matters worse, you are very likely to be staffing your project with trainees.

Project Winifred was in this situation. There were 18 object developers, in the expert/novice mix. We estimated the project could have moved 50% faster if it consisted of only six experts. However, at the time they started, they could not find six experts to hire, and, they were using the project as a training project, so they were required to have at least six novices.

Let us look at a team structure for such a mix of people...

**Team structure**

How you pair people affects their productivity. An expert gets better working with another expert. Managing a trainee drains most of the productivity out of an expert. This was the basis for the Day Care strategy (Risk Reduction Strategies).

Let us apply the formula from Day Care to a project with four experts and six trainees and two team structures.

**Homogeneous Teams.** According to a common staffing strategy, there are 3 teams, each with one expert and 2 trainees, under the lead of the senior expert. Within each team, the expert works at a 1/3 their full rate, or 3, and the trainees each at a rate of 1, for a team total of 5. The leader may be lucky to do 1 unit of work while coordinating the teams. The combined rate for the project is 16.

**Progress Team / Training Team.** Splitting into a training team and progress team was advocated in Day Care, one expert is assigned to all six trainees, while the other five experts produce the system. The training expert produces no function, and the six trainees produce 1 each, for a team total of 6. The progress team, consisting of 3 experts, works at a rate of 27. The combined rate for the project is 33, more than double the homogeneous team strategy. There are other factors that add in, such as that the three experts may work at higher rates by working together, that there are only two teams to coordinate instead of three, and that the trainees may learn more by getting more, dedicated expert time.

To get an idea about the overall effect of teams on a project, read the eyewitness account by Jim Coplien (Recheck), who has examined organizations for efficiency characteristics.

**Productivity changes over time**

It is not the case that even best programmers develop at a constant rate over the course of the project. They must spend some time learning their way around the problem, and not much is produced. For a while after this, they can produce at some peak level, until the complexity of the system finally starts to tax their recall, and they slow to a slower, sustainable pace. Figure 7-2 illustrates.
During the learning phase, the total lines of code / classes / function points produced starts off slowly. It increases as the developer internalizes the material and starts programming. For a while, a function is produced at a great rate, until the system reaches a size and complexity that fills the developer's head. At this point, which I call "saturating", the rate of newly produced function starts to decrease, because of the time required to refresh or relearn things about the system. After a while, productivity stabilizes at a value lower than the peak productivity.

As with all these sorts of curves, there is a great deal of variation from person to person. As before, we can extract some value from even a rough estimate. I use numbers that match my personal experience: a 2-month learning phase, a 4-month peak phase, a 2-month saturating phase, and a saturated production rate of 1/3 - 1/2 the peak production. The time periods correspond to about when your first and second increments should be coming out!

I used this curve to estimate project Winifred at its start. That calculation indicated that six good developers, operating at just twice an average productivity, could deliver the system in 15 months, where a team of 12 average programmers would need almost two years. At the end of the project, we decided that both the curve and the prediction were right.

I need to reiterate that this curve, the productivity curve, and the next section on lines of code per month, vary across people. Nonetheless, they will give you some insight into the effects of time, quantity and quality of people on the project. They help you understand what is happening, or going to happen on your project.

**Lines of code per month.**

It is a common hearsay observation, also mentioned in *The Mythical Man-Month*, that the lines of code produced per person per month is relatively independent of language. I have seen good programmers produce on the order of 3,000 lines per month at their peak and 1,000 lines per month in saturation. That has been the case in PL/AS, FORTRAN, C, Prolog, and Smalltalk. You get the industry standard rule of thumb of 10-20 delivered lines of code per work-day if you amortize these numbers over a fully staffed project, including non-programming staff.

An object-oriented version of this rule of thumb is "2 weeks per class for single project classes, 4 weeks per class for commercially reusable classes." This is a fairly commonly used figure. It is variable with the writing style of the developer, since some developers like to make lots of small classes, while others prefer fewer, larger classes. Using published averages, though, an "average" class might be expected have 15 methods or functions, each with 10 lines of code. At 2 weeks to develop, that also works out to 15 lines of code per day (for more discussion on these numbers, see *Estimates*).

So is there validity to the measurements, "count the lines of code" or "count the classes"? I shall say, "Yes", and immediately follow it with a riddle: you only know how many classes and lines of code there are in the program when you are done! How do you plan to get them earlier? In the middle of the project, they are fluctuating. The expert I mentioned earlier said he hits his final line count 1/3 of the way through the project.

The counts serve your purpose at the very beginning of a project, when you are trying to get order-of-magnitude estimates about the upcoming costs, from the results from a similar project. I used it in the preceeding section to estimate the progression in Project Winifred. There, I did not care what the actual number of classes were, but used them as a convenient measure of delivered function. Any units could have been used, because they vanished in the final result.

There are two twists in the entire discussion of lines of code. One I have already mentioned: a good designer will slowly reduce the size of the design over time. This makes it hard to know what line or class count to use. It is not that they stop typing. They replace lines as they write. I suspect their change log would show a steady production of lines of code, even while the total system line count shrinks.

The other twist is "frameworks".
Frameworks

One-line summary of Productivity in Large Teams

*Two or three factors will drive the project’s productivity*

*Size: A small team can do an amazing amount, but a properly functioning large team can do more*

*Skill Mix: Try for a few good people*

*Structure: Team people for synergy*

*Lines of Code per Month: How are you going to use it?*

Migrating the Organization

Do you really need to migrate the whole company at once? I argue, probably not.

One-line summary of Migrating the Organization

*Try migrating 10 people per year*
8. Recheck

Technology is only part of the story

...  

WINIFRED REVIEWED

The project review is given in four parts: summary, history, analysis, and relation to the book topics. You may find this format useful for examining your own project.

Summary: Delayed unnecessarily by overconfidence
System Type: Business system, OO client to relational server  
Project Duration: 1.5 years
Staffing: Peak staff 45, about 20 OO developers, 16 novice, 4 expert
End Result: Delivered system on time, learned object technology
Quality: Never high, but always sufficient
Success Factors: Increments, determination, a few key players
In Brief: Successful, but delayed through overconfidence, lack of attentiveness

History:

Stage 1 - Good start, with experience & support
The organization was very supportive. The sponsoring executives allocated ample money for training, consultants, upgraded workstations, and new software tools. They arranged for knowledgeable users to be available up to a full day per week, as needed. The contact company providing the OO expertise has an OO-savvy executive, the project technical lead was an experienced OO designer, the mainframe technical lead was also experienced. The department had done client-server systems before, using various non-OO tools. They were comfortable with prototyping, and iterative, incremental development. They had installed and knew how to use group communications tools, in this case Lotus Notes. They established a plan with quarterly deliveries. The first delivery was to have only a tiny amount of function, but contain the key architecture and most of the infrastructure. Every sub-system was to deliver something each quarter.

Some things did not start well. The consultants had poor communication skills. They argued with each other constantly, differing on every point of concern, and lectured at, instead of listening to, the business experts. The two organizations, the client and the contract company, were each intent on training as many novices as they could, with the result that there were 12 novices to the one OO expert. Other OO experts were brought in periodically to comment. As they disagreed with each other, this resulted in confusion.

The result was low staff morale within two months, and a history of poor communications that persisted for the first year of the project.

Stage 2 - No architecture, almost no first delivery
The first increment did deliver, but it almost did not.

The OO designers and the relational database designers disagreed violently as to the basic entities to model and came to an impasse. The OO designers wanted a highly abstract, recursive design; the relational database designers wanted a straightforward design. In a two-day meeting, we resolved to use a straightforward approach, and immediately found that there was almost no difference between the OO- and the relational designs at the conceptual level. It became apparent that the differences were due to technology, particularly the use of inheritance and many-to-many relations in the OO design, or to small differences in assumptions, which were easily ironed out.
Although an infrastructure was built, there was no coherency of design—no architecture. Each programmer designed and wrote as he saw fit. Since most of the programmers were novice and the rest disagree with each other, the result was that the design and code were inefficient, hard to read, hard to change.

There was no ownership of classes. Ownership was by use case. The result was that classes became bloated with arbitrary responsibilities. No one person could clean up the classes.

On the good side, two excellent OO programmers were hired, who almost singlehandedly put out the first release, through their arduous work.

The users were heavily involved in the creation and review of the use cases, and reviewed the intermediate results. They were patient and provided good information. The executive sponsors never backed away from their commitment to objects and a working system. The project management, although frustrated at the OO contractors, understood the nature of a first increment in a new technology.

Due to lack of progress, scope was cut on the delivery. The first delivery, promised to be small, would be even smaller.

The result was what I call a "bubblegum" release. It worked, but its construction was not robust. The release went out, and was well received by the user community.

Stage 3 - New increment, add teams, mentors, architecture

Following the first release, it became apparent the project needed architecture-level decisions. Throughout the project, no one person was ever found to be the “system architect” or even the “workstation architect”. However, the various team leaders banded together to act as architect’s surrogate. The result was, there were never any mandates about architecture, but the teams managed to avoid making big mistakes. A sense of minor architecture evolved through the team leaders.

Standards and reviews were instituted. Reviews were not held regularly, but the use cases and the business classes were published, reviewed, and errors were found. The coding standards were ignored by the programmers.

There was no staff turnover among the novices, but significant staff turnover elsewhere. The project manager, development manager, OO technical lead positions changed twice each. The OO consultants changed constantly, as did the OO experts. In stage 3, a final, relative stable group of managers and experts were put in. There were still only 3 OO experts. One was assigned to business class design, one to Smalltalk programming, and one to the infrastructure.

In response to the chaotic development of the first increment, one of the managers instituted a waterfall, “throw it over the wall” approach, with requirements signoff, handover to a second team for analysis and design, with handover to a third team for programming. We were able to terminate this process after a short time. It was clear that the programmers were ignoring the designers during this period.

The end result was delivery of a second increment, containing the beginnings of an architecture. The release was again well received by the user community.

Stage 4 - Owner per deliverable, new infrastructure

Ownership per deliverable was instituted. Each use case, subsystem, class, and document to be produced received an owner, even though several people might actually edit any one deliverable. This provided accountability and ease of evolution of the products.

The infrastructure expert decided the infrastructure had to be rebuilt, with a careful and robust design. This process took over six months, but he was able to hire two experienced OO designers and support the two releases that went out in that period.

Morale, although not high, was stable. The team knew they could produce a release per quarter, they had come to know the users, the technology, and their own weaknesses.

The primary hazard was that the requirements were unstable. The users and executives frequently changed details of use cases and screens, even just before deployment. The development
team adopted a 3-pass approach to requirements (three “deep” iterations). The first, base on ini-
tial requirements, resulted in a first user viewing of full functionality. The users changed what
they wanted based on that viewing and any new thoughts. Upon second user viewing, they were
asked only to name errors on the screen and in the function. After that, no changes were accepted
and the system went out.

The result was a project that delivered regularly to content users. The architecture, design,
and code, while never beautiful, was adequate to the task.

Two questions

What caused the early problems?

? Forgetting lessons of previous decades, on project planning, staff stability, responsibilities
and communication channels.
? Consultant ego.
? Absence of architecture.
? Missing expert staff, in architecture, UI, design, and code.

Why was it able to succeed at all?

? Key individuals at key moments.
? Increments, used to fix the process.
? They developed a habit of delivering.
? Unwavering support from the executive sponsors.

Project checklist

1. Charter. The project fit and stayed within the charter. It was a production project, replacing a
legacy application. The charter included training of 12 novices, and outsourcing of most of
the design and programming.
2. Suitability. Not “suited” to object technology in any particular way, which meant that the
learning curve was relatively high for the benefits expected. It was too hard a system to build
with the application generators available at the time, which is part of why Smalltalk was se-
lected. The project benefited more from the rapid development environment of Smalltalk
than from variation-on-theme. The project sponsor said she could see that the OO team could
respond to changes in the requirements faster than her other groups, even though they were
largely novices.
Very mixed temperaments in the staff, some did well moving to objects, a few had a difficult
time with the abstractness and with the iterations. Few experts for many beginners. Ego
problems with the OO consultants.
4. Technology. Object technology and Smalltalk were selected by the contracted company, not
by the client company. Nonetheless, the client felt it was worth exploring. Three-tier archi-
tecture had never been tried before. They bought the latest upper-CASE tool, spent large
amounts of money customizing it and arranging for code generation, but people only relu-
ctantly used the tool, and only to draw class diagrams. They made organization-wide use of
Lotus Notes to capture use cases, meeting minutes, issues and decisions, and bug-reports.
They used Envy/Developer for Smalltalk versioning and configuration control. They bought
but never used a simulation tool to simulate overall system performance under load. The 21"monitors they for every programmer were greatly appreciated.
5. Methodology. Deliberately minimalistic, defined on the fly. At each new stage of development
they first worked through part of the topic, then defined a first-cut standard for the current
and future use. They revised the standard based on experience when they encountered it on
the next increment.
Roles. There were six OO-based roles:
(1) Business analyst: got requirements, initial business class desig, UI design.
(2) Designer/programmer: did domain and UI design and programming.
(3) Design mentor: taught and coached on OO design.
(4) Smalltalk lead: designed and programmed frameworks
(5) Reuse point: programmed utility classes, bought classes for examination.
(6) Infrastructure designer and programmer.

There were the standard non-OO roles:
manager, tester, administrative assistant, database designer, mainframe programmer, etc.

Key techniques:
(1) JAD-style joint requirements gathering
(2) Use cases for capturing functional requirements
(3) Class diagrams for the business object model, some CRC-card use, some responsibility-based design without CRC cards, a fair amount of standing around the whiteboard drawing instance diagrams and messages or standing around a screen pointing at classes.
(4) E/R diagrams for the data model
(5) Standard COBOL design
(6) Black box testing by an external test group
(7) Milestone-based project management

Key deliverables:
(1) Use cases
(2) Class diagrams
(3) Statements of abstraction and responsibility for each domain class
(4) Matrix showing class categories used by each use case cluster
(5) Screen flow diagrams, screen snapshots
(6) Commented source code

Key standards:
(1) Only the externally visible UI standard was adhered to
(2) Coding standards were ignored.

Process:
One increment per quarter, 3 deep iterations per increment. Key reviews and milestones were identified, but no other ordering was required of the developing software. Each next stage was started when the deliverable was judged "stable enough to proceed", knowing that there would be requirements flux up through the last day. The milestones were of the nature: “ready to start design”, “shown to users”, “design reviewed”, “ready for DBA”.

Training. They arranged one week of Smalltalk programming, one week of use case and OO design training per developer. They used project mentoring as on-going training, which was insufficient, given the number of novices and the few experts. Although they did not create self-study groups, a few programmers started teaching themselves after a year.

Consultants. A single major contractor and other outside contractors were used for most of the programming and all of the OO expertise. The consultants and contractors had a high turnover rate, which caused lost communications and breaks in the continuity of design. The in-house staff and the testing staff were stable over the period of the project, which gave the project coherence over time. Even though it was estimated that the entire programming could have been done with 6-10 expert Smalltalk programmers, they were simply unable to find that many available experts.

Mainframe and relational database. After the initial mismatch, the OO and database designs became common. Friction between the relational and object designers persisted throughout the project, each team dominating during different periods of the project. The project spent
about 18 expert work months in “the persistence black hole”, making a fast and stable object persistence mechanism to join the OO programs in the workstation to the relational database. The database designers and COBOL programmers could not keep up the rapid change cycles the Smalltalk programmers could, so they followed single pass design cycles, starting work only after the main design was declared relatively stable.

9. **Domain modeling.** This was a point of debate at the beginning. It became apparent within a few months of modeling that there would always be several legitimate analysis models and that not all of them worked equally well as software designs. The notion of “analysis model” was therefore given up, in favor of “initial design”. Each subsequent design was required to be a valid business model.

10. **Cutting edge.** The managers stayed as conservative as they could, given the number of new technology elements they were using. They had used client-server designs before. Only the object technology elements were new. They avoided an object database in order to reduce new, fragile technology. They avoided discussion of distributed objects entirely.

11. **Estimating.** Estimating was initially done from use cases, and guesses as to the number of business classes to be developed. From those, multipliers were used to guess screens, total class count and framework count. These were divided by the skilled people needed and available. The result was matched against common sense. During an increment, a second estimate was made once a reasonable use case-, screen-, and class-count could be made. The multipliers were rechecked after each increment for the next increment.

12. **Planning.** A person was dedicated to creating and maintaining the project plan over a 4-month period. This person initially tried to create a plan of the form: requirements phase triggers analysis phase, which triggers design phase, and so on. He had to give up, since requirements were found to be unstable until the very end. He settled for tracking the key milestones of each deliverable, which worked well and simplified his work.

13. **Increments, iterations.** Macro increments, micro iterations, as described in the process part of methodology. There ended up being 5 increments over 18 months. The incremental development was key to the success. There was one shallow followed by three deep iterations on the functionality: initial screen design, functional draft, functions corrected, deliver. The boogeyman was found to be in the changing requirements, the unstable infrastructure design, and a long test-and-fix QA cycle.

14. **Risk reduction.** Twice a month a risk assessment was taken. Some of the risks found and addressed were instability in requirements, code quality, a requirements “signoff” was not meaningful, users needed to see a fully functioning version in order to decide what they really wanted, novices getting insufficient attention, training and control, a long QA cycle, absence of an architect. The project used common sense, plus most of the strategies in Risk Reduction Strategies to reduce these risks. Sacrifice One Person and Gold Rush were the most appreciated. The middle tier of the three-tier system was put in partly to specifically reduce the risk that the object team would damage the mainframe's database with its new and possibly radical business model.

15. **Teams.** After trying various teaming strategies, they used the idea of “function teams”, closely knit specialists jointly responsible for delivering the subsystems. Each team was carefully composed to insure proper skill coverage and non-conflicting personalities. Team members specialized in different ways. The best communication rate was achieved using one requirements analyst with two or three designer/programmers.

16. **Reuse.** One person was given half-time to seek, read, evaluate and order class libraries from industry sources. Even within the project, there was very little inclination to use other people’s solutions or purchased solutions. The project was small enough that everyone knew everyone else, which allowed team leads to persuade each other or individuals.

17. **Productivity.** A key factor that drove productivity was use of novices in an expert-novice mix. The experts were soon reduced to a fraction of their usual productivity. A second key
factor was the use of a handoff-based, requirements/design/programming approach early in the project. This slowed productivity significantly. Productivity improved again when the function teams were employed.

This concludes the review of Project Winifred.

**TECHNOLOGY IS ONLY PART OF THE STORY**

To put the entire issue of technology into perspective, I include this last Eyewitness Account, by Jim Coplien

...
9. Collected Risk Reduction Strategies

Where most of the material in this book is written in an anecdotal style, suited to light reading, this chapter contains twelve, selected strategies that have been tested on projects, collected into a reference format.

To reduce project risk, we tend to apply some particular strategy, often a staging strategy: incremental, iterative, spiral, eddy, or fountain. Each gives us new information early, to enable some mid-course adjustment. You can invent your own strategies, creating whatever sequence of development you need, if you keep in mind the fact that each action should reduce risk of non-delivery. Here the basic risk reduction strategy, good advice, hard to apply:

- Look carefully all around the project.
- Detect the risks.
- List the risks in order.
- Work on them in order of danger.

Not really much use, was that? The top risks are those that will keep you from delivering the system you need. Unless they are addressed, the rest of your work does not matter. The top risks do not arrive in waterfall sequence; they arrive in any sequence, popping up at any time. You will know a project based on risk management, because it will become increasingly clear over time that the system will ship successfully.

At some point, the top risk is absence of good requirements, at another point it is shortage of staff, then perhaps training, lack of knowledge about an algorithm, questions about a vendor. Risk reduction would be the favorite strategy among project managers, if only we could figure out how to describe it. Very few people can do that.

In this chapter, I put forward a format for capturing risk reduction strategies. The software industry is in the position to produce a several hundred page risk-reduction catalog listing typical project risks and known ways to address them. Every successful project manager has a set of pet strategies she or he uses in a pinch. Once that catalog is made available, it can be used by new project managers to leverage previous experience.

The first fragments of such a catalog were published in IEEE Computer journal, October 1996, and reviewed at the Pattern Languages of Program Design conference, 1996. All of these strategies have been applied and recommended. Just as in medicine, where a prescription is not a guaranteed cure, so these strategies can only be starting points for your experience. That is why each says, "try this", instead of "do this''.

Here, first, are thumbnail sketches of the 11 strategies, clustered by major issue.

1. Knowledge: Clear the Fog
You don't know the issues well enough to put together a sound plan, so...
Try to deliver something (almost anything); this tell you what the real issues are.
(General strategy with specializations:)
- Early and Regular Delivery
- Prototype
- Microcosm

2. Knowledge: Early and Regular Delivery
You don't know what problems you will encounter during development, so...
Deliver something early - discover what you don't know you don't know.
Deliver regularly - improve each time.

3. Knowledge: Prototype
You don't know how some design decision will work out, so...
Build an isolated solution - discover how it really works.

4. Knowledge: Microcosm
You have to create a real plan, but have never done this sort of project, so...
Run an 8-12 week instrumented pilot to get productivity and throughput data for your plan.

5. Teaming: Holistic Diversity
Development of a subsystem needs many skills, but people specialize, so...
Create a single team from multiple specialties.

6. Productivity: Gold Rush
You don't have time to wait for requirements to settle, so...
Start design and programming immediately, adjust their requirements weekly.
(Presupposes Holistic Diversity)

7. Ownership: Owner per Deliverable
Sometimes many people are working on it, sometimes nobody, so...
Make sure every deliverable has exactly one owner.
(General strategy with specializations:)
  Function Owners / Component Owners
  Team per Task

8. Ownership: Function Owners / Component Owners
If you organize teams by components, functions suffer, and vice versa, so...
Make sure every function has an owner, every component has an owner.

9. Distractions: Someone Always Makes Progress
Distractions constantly interrupt your team's progress, so...
Whatever happens, ensure someone keeps moving toward your primary goal.
(General strategy with specializations:)
  Team per Task
  Sacrifice One Person
  Day Care

10. Distractions: Team per Task
A big diversion hits your team, so...
Let a subteam handle the diversion, the main team keeps going.

11. Distractions: Sacrifice One Person
A smaller diversion hits your team, so...
Assign just one person to it until it gets handled.

12. Training: Day Care
Your experts are spending all their time mentoring novices, so...
Put one expert in charge of all the novices, let the others develop the system.

Project management is a balancing act. One strategy remedies several situations, each situation resulting from some force getting out of balance. The strategies in this chapter list the forces they balance. When any one of the forces is excessive, you may find yourself in a situation needing the recommended action. Similarly, one situation has several possible remedies, depending on small variations in the situation, or your personal management style.
Each entry is written as though it were a medical diagnosis. It starts from subjective information ("indications"), how you feel, or complaints you hear in the hall. It continues with the forces that you are trying to balance. Then comes the recommended action. It is followed by an overdose detection: if you apply the strategy too heavily, you will find yourself in a different difficulty. The "resulting context" is the situation you are likely to find yourself in next, perhaps with a different problem to solve. There is a section for the principles involved. These are included for future researchers to examine, studying the underlying principles across many strategies to form more solid theories of project management. Each strategy is accompanied by a number of typical situations you may find yourself in, or a previous project was in.

...
10. Crib Sheet

CRITICAL SUCCESS FACTORS, PRIME FAILURE INDICATORS

CSF 1. Incremental scheduling and staging.
CSF 2. Willingness to find and fix failing ideas.
CSF 3. A habit of delivering.
CSF 4. Executive sponsor, project manager, technical leader.

PFI 1. Absence of a delivery after 8 months.
PFI 2. Use of C++ in an I/S shop.

TRUTHS (FROM MYTHS)

<table>
<thead>
<tr>
<th>TRUTH:</th>
<th>Myth:</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1. Managing object development is</td>
<td>Object development is completely different.</td>
</tr>
<tr>
<td>mostly the same.</td>
<td></td>
</tr>
<tr>
<td>T2. Any new technology is hard.</td>
<td>Objects are easy to learn.</td>
</tr>
<tr>
<td>T3. Retraining in object-think swamps all other costs.</td>
<td>We should use C++ because we know C.</td>
</tr>
<tr>
<td>T4. Iterative development still has handoffs and milestones.</td>
<td>Iterative development means no &quot;handoffs&quot;.</td>
</tr>
<tr>
<td>T5. An expert produces the system quickly, a mentor communicates well.</td>
<td>Any expert programmer is a good mentor.</td>
</tr>
<tr>
<td>T6. Smalltalk is large.</td>
<td>Smalltalk is small.</td>
</tr>
<tr>
<td>T7. Object COBOL is OO ... or else just COBOL.</td>
<td>Object COBOL is like COBOL.</td>
</tr>
<tr>
<td>T8. C++ is much harder than C.</td>
<td>C++ is like C.</td>
</tr>
<tr>
<td>T9. Methodology is learning what to use, when, how to communicate the result.</td>
<td>The project will succeed if we use XYZ methodology.</td>
</tr>
<tr>
<td>T10. Every job involves design, everyone is responsible for finding bugs.</td>
<td>Requirements analysts only gather, designers only design what analysts say to, programmers only program what designers design.</td>
</tr>
<tr>
<td>T11. Specialties develop; different tasks use different mindsets.</td>
<td>The OO programmer should do everything: interview users, model the business, design and program.</td>
</tr>
</tbody>
</table>

EXPECTATIONS UNDER CONTROL

E 1. Understand first where your organization is in need.
E 2. Objects change the project manager's life relatively little, while changing the developers' lives enormously.
E 3. Requirements gathering, requirements analysis, system test, rollout, installation and training times are roughly the same as for non-OO.
E 4. Programming time will only be reduced if you are using experienced OO programmers.
E 5. Object development requires more, rather than less, communication.
E 6. Where in normal life the boogieman is apocryphal, on a software development project it is real -- it is What You Don't Know You Don't Know.
E 7. Plan on 9 months until a new hire is able to re-earn her/his salary fully.
E 8. Although most of the staff will enjoy the move, some will not.

E 9. Whichever deliverables notation you choose, you must still decide who does what activity to produce it, who reads it, how people interact.

E 10. Late-breaking design issues will be found during programming and will change the model.

E 11. Your first project will show what you really need in the methodology.

E 12. Your time-to-market benefit will vary depending on the amount of internal and external similarity you can exploit.

**SELECTED SURVIVAL RECOMMENDATIONS**

S 1. Choose a technology that minimizes time-to-delivery, minimize the constraints on that choice.

S 2. Assume knowing current language and tool set does not reduce training time.

S 3. Plan and measure by delivery milestones.

S 4. Use your previous experience to avoid the most common traps.

S 5. Put a business specialist on the development team.

S 6. If it is not your specialty, buy it; until you can buy it, assume it can't be built.

S 7. Find someone who has worked on a project this big.

S 8. Run 2 weeks language training,
   1 week training on OO design thinking,
   a 1/2 day internal class on coding conventions,
   internal training on the contribution of each role.
   Add training as you encounter the need, continuously.

S 9. Get versioning and configuration management, team communication, and performance monitoring tools.

S 10. Exercise extreme sobriety in judging upper-CASE tools.

S 11. Set standards for system design, language-based design, and coding. Encourage and enforce them.

S 12. Assign one person to have the time and task to find and import class libraries. To improve reuse, improve people's attitude toward its economic feasibility.

S 13. Manage precision and accuracy to keep discussions on target and produce needed information. Dive briefly into high precision to evaluate a risk or get needed information.

S 14. Analyze just enough to build and ship the next increment. (Note: this, of all the recommendations, is tuned to surviving and delivering; a small project with expert staff may work differently).

S 15. Design and program properly from the start.

S 16. Provide developers time and encouragement to write truly object-oriented C++/ COBOL/ Ada.

S 17. Assign an owner to each deliverable, including use cases and classes.

S 18. Let specialties flourish, let each team contain different specialties.

S 19. Separate a training team from the progress team.

S 20. Make sure that someone always makes progress.

S 21. "Seed" the organization: migrate some (10) people at a time to expert level, use them to seed the next team.

S 22. Rebuild the project schedule only after each increment is delivered.

S 23. Change techniques, drop deliverables, drop function, improve staff skills, in preference to adding staff.

S 24. Find the boogieman as early as possible.

S 25. Be willing to change anything to deliver.

S 26. Show working results every 3-4 months.
S 27. After each increment, pause and learn.
S 28. Terminate the current project if its charter changes, create a new project with a new charter.
What These Successful Projects Did

P 1. The unwavering support of upper management provided a constant boost and freedom
to maneuver during periods of low morale.

P 2. They built the running domain model first, prior to any user interface.

P 3. They selected and followed simple, conservative standards.

P 4. They allowed for several revisions for key, risky, important areas.

P 5. They started over, with a small team.

P 6. Their small team worked to create an architecture.

P 7. They used variation-on-themes to take advantage of the internal similarities.

P 8. They instituted an internal self-tutoring process.

P 9. They hired good people and gave them good tools to help them get their work done.

P 10. They got real users involved.

P 11. They made total-ownership teams, responsible for everything from requirements to code,
test and documentation.

P 12. They established a weekly self-teaching meeting.

Hazardous Phrases

Add one risk point for each phrase that applies to your project, or that you believe is a safe
recommendation. Since I don't want you to mistake these for the constructive advice in the book, I
put each line in parentheses.

H 1. ("Object technology fixes your software process").
H 2. ("Object technology gives you reuse").
H 3. ("Never mind previous experience, OO is completely different").
H 4. ("Define the entire business model at one time").
H 5. ("Prototypes replace design").
H 6. ("Prototypes become production software").
H 7. ("Iterations control themselves").
H 8. ("An 'accurate' business model makes a 'good' software design").
H 9. ("Requirements, analysis, design and implementation are easily separated").
H 10. ("We should use C++ because we already know C").
H 11. ("We have someone who knows part of C++ already").
H 12. ("We should use this CASE tool because we already have it in house").
H 13. ("Moving an entire company to object orientation is as simple as issuing a decree and select-
ing a compiler").
H 14. ("Use a technical architect only 2-3 days per week, or one who appears to be a really bright
person but has poor communication skills").
H 15. ("A successful smaller project ensures a successful big project").
H 16. (Inventing productivity gains that object technology will bring, in your efforts to gain ex-
ecutive sponsorship).
H 17. (Separated analysts, designers, programmers).
H 18. (Using new technology for a time-critical project).
H 19. (Contractors learning the technology on your project).
H 20. (A deliverables-heavy methodology).
H 21. (Changing project scope).
H 22. (Continually shifting user group).
H 23. (The experienced users do not find time to come to the meetings).
H 24. (Experts teaching and programming concurrently).
H 25. (No or shared ownership of the classes’ design.)
H 26. (Mastery of language syntax, but not OO thinking.)
H 27. (Trust that the tools will somehow guide your people in their thinking; working through all the diagrams it supports, hoping that the software will pop out.)
H 28. (Budgeting one week of language training for novice OO developers who are about to build a $2,000,000 system.)
H 29. (Producing huge amounts of code per day using copy-paste techniques.)
H 30. (Relying on indirect links to the users.)
H 31. (Community alteration of classes (no ownership).)