MANAGING A DESIGNER/2000 PROJECT

Leslie M. Tierstein,
W R Systems, Ltd.

INTRODUCTION

Congratulations! W R Systems has just been awarded a multi-million dollar contract to do full life cycle development of a mission-critical system for a major government agency. And you've been named the technical project manager. The new system will replace numerous "stove-pipe" application systems as well as the aging, obsolete hardware that hosts the applications at not one, but three, divisions of the agency. And, oh, by the way, although the divisions will have one management and management structure very soon, right now, they still have different ways of doing business which somehow have to be accommodated by one, totally integrated application -- the one you're responsible for building.

After the initial joy of being awarded the contract, reality swiftly set in. We were faced with many management and technical decisions to be made in a relatively short period of time, followed by more than a year of specifying the hardware and developing the software following the methodology we'd said we would use. This paper explains some of the decisions we made, and the methodology we ended up using, and, hopefully, offers some pointers for people embarking on similar projects.

RE-ENGINEER OR RE-IMPLEMENT?

The decision whether to re-engineer business processes or re-implement existing procedures was, in principle, made for us by the contract: our mandate was to re-implement existing systems, so we could move the users off of hardware which was no longer being manufactured, for which spare parts and maintenance were almost impossible to get, and which would run out of disk space within a year.

Although this mandate limited the scope of re-engineering, it didn't eliminate it. Three organizations, with divergent processes, were being merged into one. Some divergent processes would still be needed, to support different businesses at the organizations; however, in most cases we needed to come up with one common practice with one computer implementation.

Re-engineering was also required by virtue of re-implementing in a different hardware-software environment. We were going from a batch-system with 24-hour turnaround on transaction processing, almost no on-line editing and verification, and reports as the primary way to view data, to an on-line system. This *de facto* re-engineering will have a profound effect on the way the users do business, and had a profound effect on some of the analysis and design decisions we had to make and methodologies we had to use during development.

SELECTING ORACLE

Like the decision to re-implement in general and re-engineer when required, the requirement to use the Oracle design and development toolset was specified in the contract. As part of a previous contract, W. R. Systems had studied the commercially available products which supported doing integrated CASE development. The candidate product set had to support full life cycle development, from requirements gathering, through analysis, design, implementation, deployment, and maintenance, in a client/server development, but with deployment on both character-based terminals and GUI workstations. The Oracle product suite was a clear winner, both in terms of functionality and cost-benefit analysis, although it was far from perfect. We recognized that we would have to supplement the Oracle-provided tools with others, either developed in-house, or purchased from another vendor and integrated, via procedures or custom software, to the Oracle tools.

So, although we didn't 100 percent agree with the Oracle methodology, we bought it. In the version we benchmarked (CASE 5.1), the user interface scored low, but it was the best methodological framework -- and robust, working toolset that went with that framework -- for our mission-critical, database-intensive software project.

LIFE CYCLE DEVELOPMENT METHODOLOGY

One of the most attractive features of the Oracle design and development tools was that they provide a framework which lets implementors combine top-down analysis and design with bottom-up rapid prototyping. I like to call this a "middle-out" approach. A certain amount of top-down analysis and design is required. But some of the traditional top-level development phases can be worked in parallel with more detailed design and implementation. So, roughly in chronological order, but with some overlap

between the tasks, our development effort consisted of the phases discussed below.

Plan, Plan, Plan

This being a government contract, we had lots of plans to write before we could get to designing and developing software. These included a Software Development Plan, Software Test Plan, Configuration Management Plan, Quality Assurance Plan, and more. Although there was some groaning during this phase, especially when people got their plans back from editorial and management review, in retrospect, I'd say that preparing the plans was worthwhile, because:

?? The Software Engineering Institute likes them.
?? Your users and sponsors like them.
?? The developers can even grow to, if not like them, at least rely on them.

First of all, we didn't have to prepare the plans from scratch. Existence of such plans is required if your organization's development methodology is going to rank anywhere above level 1 (chaotic) on the Software Engineering Institute's Capability Maturity Model. W R Systems is a software development contractor; in order to get contracts, we have to go through Software Capability Evaluations (SCE's), where a team of independent auditors reviews your procedures and methodologies. So, for this contract, we mostly reviewed and enhanced existing documents, to incorporate the new methodologies and/or standards embodied by the Oracle development tools.

Second, these plans are the first things your users and sponsors can see that makes your methodology explicit. As such, they start the process of user education. And, you'll need lots of user education, starting with the way the system will be developed, and the role you expect them to play in this process, going through them learning how to use the software.

As for the sponsors, if your sponsor is the U. S. government, the plan should give concrete examples of the sorts of specification documents that will be produced. MILSTD specifications, while evolving, have not kept pace with evolution of CASE tools. We had to make sure that the sponsor understood that we would not be producing specifications which exactly matched MILSTD, but the nearest thing that the development tools could produce automatically.

The plans are also the first step in developer education. Developers can get an ideal of what will be expected from them at each stage of the project. These plans didn't go into excruciating detail -- we got to that soon enough, in our standards and procedures manuals. But, they did formulate methodological guidelines and establish expectations.

Principles Applicable to all Development Phases

Some principles of our methodology apply to all phases of the development life cycle. I'd like to thank Dai Clegg, of the Designer/2000 development (marketing) team for talking about the concept of the "Time-box".

A project is defined by the three dimensions of the box:

??	Time	How much time do you have to complete development and deployment?
??	Functions	What functions does the system have to include?
??	Money	How much money can you spend -- on staff and other resources -- to get the job done?

Determine which of these can be adjusted if the project runs into difficulty. Have a risk management plan in place, for when you do run into difficulty. You will never have flexibility in all three dimensions -- most managers and users won't put up with it, and the uncertainty can be unnerving to project teams, too. Having no flexibility at all is a recipe for disaster, especially if this is your first project using a particular methodology and/or toolset. At a minimum, it will result in burn-out; I don't even want to suggest the maximums, but all scenarios include varying degrees of mayhem and employment instability.

Another useful technique suggested by Dai Clegg was "MoSCoW". It stands for the four categories - "must have", "should have", "could have", and "won't have" -- that you can use to prioritize features and functions which are candidates for inclusion in the new application.

Must have	An supply management system absolutely MUST have an inventory replenishment algorithm.
Should have	The provisioning screen SHOULD look and act like this, for maximum efficiency; but, a slightly different user interface could do in a pinch.
Could have	Well, if we have time, we COULD make the report accessible from both screens, as well as from the menu.
Won't have	Sorry, that's not in the requirements, so you WON'T have it.

The MoSCoW categories and criteria are easy to apply, and easily understood by users. Get the users to help you classify their own requirements and, later in the project cycle, their requests for changes and enhancements.

Requirements Gathering

Requirements gathering is one high-level task that has to be performed before you get too deep into other analysis and design tasks. You need well written, understandable requirements, that all prospective users have agreed on. These requirements have to be understandable to functional users, so they can see that their system is taken care of; to implementors, who have to write the code implementing the requirements, and, don't forget -- to testers, who have to test the software and make sure it meets the requirements.

Our project refined the requirements which had previously been gathered through extensive user interviews and workshops. Gathering and distributing the requirements is essential for user buy-in. And it's a vital weapon for analysts in stopping "creeping featurism": "Sorry, it's not on the requirements list." Of course, in conjunction with the requirements list, we developed a means to change the requirements list, and, also, a way to track the requirements, and to trace them to functions, and later, to modules. Unfortunately, requirements traceability is one of the incomplete pieces in the current Oracle toolset.

Analysis

Somewhere towards the end of the analysis phase, I came across a book by Richard A. Moran: *Cancel the Meetings, Keep the Doughnuts: And Other New Morsels of Business Wisdom*. It's a collection of pithy sayings. Almost immediately, one of them became applicable:

> *The fastest way to turn the aircraft carrier around in the proverbial lagoon is to blow it up and reassemble it facing in the right direction.*

In other words, once problems with a design become apparent, examine them, and their implications, carefully. Most analysts will probably try to make first one band-aid fix, then another. They might need outside help -- in the form of a formal technical or QA review -- to realize that they should redesign immediately. Yes, they'll have to throw out part of their beautiful ERD or function descriptions, but it will be less work in the long run than making multiple band-aid fixes, and finding out, eventually, that you have to redesign anyway.

This principle is also true if the "mistake" is not the result of a bad design, but, say, learning more about the toolset, and/or figuring out a better way to do something. Here's where two of Tierstein's rules for system development come into play:

?? Let the system do the work for you.
?? Don't fight the system.

The Oracle toolset, like most modern tools, is built on a processing paradigm that may be more or less explicit. Working with the paradigm is almost always more productive than not taking advantage of it or even actively fighting it. Have you ever seen PL/SQL code that was written as if it was COBOL? It's not a pretty sight.

Design

Taking advantage of the Oracle tools' methodology, we started design and implementation (code generation) before analysis was 100 percent complete. However, standards and guidelines were already formulated: The project teams never developed a screen -- or showed it to the users -- before the corresponding requirement and function had been approved. And, changes were not applied to the screens before "Action Items" could be written up, to document what changes the users had requested, when, and why. In doing this, we avoided one of the worst dangers in doing Rapid Application Development (RAD). If you do too much instant coding, without design infra-structure or development standards in place, you risk ending up with what Dai Clegg called "instant legacy code" - the code only took one day to develop, but it's as unmaintainable as code that has been patched for many years.

When you are going from a primarily batch to on-line requirement, you have to expect lots of design changes

in the format of output provided to the users. Your requirements probably include alot of printed output, including proof and reconciliation reports, which are required for batch systems, but not at all applicable to on-line systems with field-level data and referential integrity checks built in. However, until the users see your screens -- with the query ability automatically provided, by virtue of the Oracle Forms paradigm, and additional query-only forms, it may be hard to convince them that they don't need printed reports. Persevere. If you design the forms so they provide all or most of the output previously only available in printed form, you may not have to provide all the reports previously specified.

STANDARDS

There's an old saying that goes, "If you have four rabbis in a room, you'll have five opinions." The same principle applies to project teams and standards: if you fail to document a standard for a particular design or implementation issue, and you have four project teams, you will end up with five different designs or implementations.

We tried to develop standards for all aspects of design and development. In fact, the project has three books (not "white papers", but full-size books, with many chapters in each) on standards.

Design and Development Standards

The Oracle Development Standards cover every applicable option in the Designer/2000 toolset. For every tab or property sheet to be filled in, the standards specify which fields are required, which are optional, and which are not used, and give instructions for supplying valid field values.

At a more conceptual level, the standards cover topics such as:

- ?? naming data elements;
- ?? specifying the appearance of screens, via the use of templates, groups, and other capabilities;
- ?? how an entity relationship diagram should be drawn;
- ?? which preferences are set at the application level, which can be modified on a module-by-module basis, under what circumstances they can be modified, and acceptable variations.

Development Standards and Procedures.

The Development Standards and Procedures give instructions on how to do everything not directly tied to using the design and development tools. These include guidelines for writing procedural code (in our case, PL/SQL); for writing SQL*Loader control files; for preparing a module for presentation at a prototype; and reporting on the outcome of that prototype and tracking resultant specification and code changes. These standards also include several checklists, lists of all the activities that must be completed, on a module-by-module or table-by-table basis, for the object to be considered complete.

Testing Standards and Procedures

Unit and unit integration testing are largely covered by the standards promulgated for module development, since this is, after all, a CASE environment. The Testing Standards and Procedures apply to subsystem and system integration testing, as well as all phases of system and acceptance testing. We needed to supplement the Oracle toolset with testing tools, so usage of those tools is explained here, too.

Using the Standards

All these standards were not in place by the start of the project. They were enhanced as the proj??ect proceeded and we discovered new things about the tools. It was particularly gratifying to the technical director when team leaders, and even team members, suggested additions or enhancements to the standards. I think we've been living up to another of Moran's axioms:

> *"Having both standards and flexibility can be done. It's just not easy."*

USER EDUCATION

User education must be an on-going activity. It starts as soon as the contract is awarded (or management gives the project the go-ahead), and continues throughout the project life cycle. The trade presses have paid some, but not alot of, attention, to the re-training of end-users and their management. It should be receiving just as much attention as re-training COBOL developers to do client-server development, since it's at least as critical to the project's success. The customers for the system have to be aware not only of the ways in which the end-product will differ from their current systems, but also of the differences in the way the product will be developed, and what their roles will be in the development cycle.

For example, our users were accustomed to large-scale COBOL developments. They were initially scared that, after the design reviews, we would vanish into

our offices for 6 months and emerge with finished code, which would be near impossible to change. We pointed to our plans, which said we would have periodic design reviews. We showed them our Guidelines, which said we would be doing prototypes, keeping track of action items, and showing them revised modules at the next convenient opportunity. We gave the users CASE-generated reports (most of them customized to include only the information the users were most interested in), and instructions on how to read the reports. With every prototype, we distributed preliminary versions of the user documentation -- it consisted mostly of a screen print and the module description text, cut and pasted from the CASE repository -- but it was enough to help them visualize their new processing paradigm.

DATA CLEANSING AND CONVERSION

Conversion of legacy data into the new system is not sexy -- programmers generally don't get to use high-tech GUI tools. But it has to be started early: to convert data from the old format to the new, someone has to know the structure and semantics of the old data. With legacy systems, some of which have been in place for twenty years, the original designers have probably long since vanished. Be prepared for some arduous work.

A good place to start is with an "attribute cross-walk", showing the mapping between the old data and the new. Not only will this form the basis of your data conversion design, it also let's you verify the database design, to ensure you're capturing all the data you need to. Plus, it reassures the users that none of their data will be lost. In order to do our attribute cross-walks (one for each of five subsystems), we had to extend the development repository to include the definition of the old files and fields, and their relationships to the new tables and columns.

Users will probably have to do some work - so let them get it on their schedules. Their work will be both in reviewing your mappings, and in data cleansing.

You'll have to start work early in order to do data cleansing. Our data cleansing reports, based on samples of the legacy data, indicated all the "dirty" data we found -- from invalid dates, to violated check and domain constraints, through missing data items and missing records. We also suggested fixes, and who had to do the fixing -- the developers, by writing more sophisticated conversion software than originally envisioned, or the users, who had to research what the missing data was, and provide it to us.

An interesting side effect of involving the users in the data cleansing process was getting good feedback from them on other aspects of the proposed system. We'd been distributing reports on domains and their values since the analysis phase, with some, but, limited comments. When the users reviewed the data cleansing reports, they realized, "Oops, if the value's not in the domain, I'll lose my data." We got accurate feedback on required domain values in a hurry.

STAFFING

The number of people assigned to any development project will, of course, depend on the size of the project. However, the skill sets of each of these people, and their levels of expertise, remain fairly constant for any project developed using Oracle's development tools and RDBMS.

Project Management

The **project manager** does the usual stuff - reports to management; acts as the official liaison with management of the user organization; and keeps track of financials and project costs. The project manager also protects his technical staff from too many non-project related interruptions.

The **technical manager** manages technical aspects of project development. The technical manager needs to be able to communicate (both verbally and in written form) effectively; you have to lead your team, write the project standards documents, and write numerous reports to users - both regularly scheduled status reports, as well as impromptu position papers, explaining yet another aspect of the development process. For example, I've written papers on data cleansing methodology and prototyping techniques and status reports.

It goes without saying, the technical manager needs expertise in the development and database methodologies you're using -- maybe not enough to do the most complex implementation work, but enough to pitch in if needed, and to double check designs and code and verify that design and implementation decisions are correct. In another development methodology, the technical manager might be thought of as the product architect.

Administration

The **repository administrator** is a relatively new position, absolutely vital for repository-based developed. This position really entails two sets of skills. On the one hand, the repository administrator needs to know how to add new users, extract

applications, and manage the CASE environment. In addition, the repository administrator is the person who knows the dictionary views and APIs inside out. You'll probably need someone who can provide customized reports; extend the repository to incorporate additional information, such as requirements tracking and attribute cross-walks; and maybe write an interface to a non-Oracle product which provides missing functionality.

The position of **database administrator** is probably not full time, even in a large project. But high-level expertise is required to set up the repository database. And, throughout the proj??ect, the administrator must be available to offer advice on the structure of user database, and do performance tuning for the development environment. The database administrator and the repository administrator have to like each other very much, or finger-pointing will occur when things do not run smoothly.

Both database and repository administrators have to talk to the **system administrator**, who is in charge of installing software and, in general, making sure you have a reliable system to develop your software on, with all the required support tools.

The Designers and Developers

Team leaders have to be both people-people and technical people. They have to be able to manage people; coordinate designs and schedules with fellow team leaders, especially in complex, multiple application development efforts; and follow the methodological guidelines of the project. Plus, they need technical skills to run their project team. Don't skimp on team leaders.

Most of the **analysts** will be working with the design and development tools - writing functional specifications, designing the database, generating code based on the module and database definitions, and then enhancing the generated code. You'll need a mix of senior and junior people doing these tasks. Since we're no longer using a waterfall model of development, you can't have senior "designers" start to design a module, then throw it over the waterfall to junior "programmers" to finish. Everyone is a designer; you just might hire the senior designers before you find the more junior staff members.

Additional analysts will need a different skill set, in order to use the tools to write the data conversion software. You'll need SQL*Loader expertise, but, probably more important, SQL and PL/SQL hard-hitters, if the data requires any type of more complex massaging than SQL*Loader can handle. And since we were going from non-normalized ISAM and other sources, extensive massaging was in order.

Quality Assurance

The SEI model prescribes an independent **quality assurance** organization - or, at least, individual. Our QA person ("Q") monitors compliance with standards, and performs periodic audits on designs and code, as well as regularly scheduled reviews. QA can take some of the heat off the technical manager, by monitoring standards compliance, so the technical manager is not perceived as the "Standards Czar".

Support Personnel

The **documentation specialist** and **testing coordinator** also reported to the QA organization. For documentation, this was a convenience, since there was only one documentation person. But it's critical for testing, where an independent evaluator is required.

The **configuration manager** reported directly to the project manager. Configuration management and related disciplines, such as version control, have consistently been, and continue to be a problem, given the nature of repository-, as opposed to ASCII code-based source. We're still on the lookout for a satisfactory solution.

Finding the People

What people did we have to work on the proj??ect? Unfortunately, not enough, and not senior enough. As we probably all know, experienced Oracle developers and in big demand, and hard to find. Start hiring early. I would rather hire smart people with limited, but applicable experience, but the right mindset ("Wow, this generated-code is great!") than someone with experience but a BAAAD attitude ("It would be faster to write it in C++ -- or pro-C.") Just be prepared for a longer learning curve. (Increase the time dimension in that time-box.)

CONCLUSION

This is a work in progress, so there's no conclusion, yet. The project, which WILL finish on time and on budget, has another four months to go. I'd like another Oracle development project, so I can use everything I've learned. But not too soon.