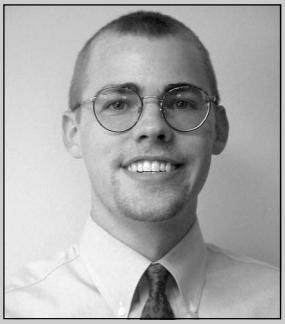
A Seven-Phase Methodology for Managing Your Next SAP Application Development Project

Jonathan Pokress



Jonathan Pokress, CPIM, is an independent SAP e-commerce consultant and president of Bluenote Consulting Group, Inc. He recently spoke on the development methodology covered in this article and other topics at the "Summit for SAP Application Developers" in March 2002, and he was a key speaker at the "Web-Enabling Your SAP Systems" seminar series in 2001.

(complete bio appears on page 22)

Nearly two decades since the advent of R/3, many custom development projects are still failing to achieve their objectives. Perhaps you've heard of companies that have spent several years and several million dollars on seemingly endless projects with dubious results. Or perhaps you work for one of those ill-fated companies whose custom code projects have all but committed them to running 3.1H well into this decade.

While most situations aren't nearly as dramatic, these examples demonstrate that planning and executing custom development projects improperly can have large short- and long-term impacts. Take a moment to review the sidebar on the next page to see if your projects are experiencing any of the top 10 symptoms of ailing development projects.

With so many pitfalls, how can you ensure the proper ROI of a custom development project? Treat it as you would any major investment, paying careful attention to potential cost, potential risk, and potential benefit. You want to identify and realize opportunities for revenue generation and at the same time minimize risk exposure and costs.

This article presents a field-tested methodology to accomplish this. Summarized in Appendixes A and B, this seven-phase methodology helps you to solidify project goals and user expectations, and tackle the project in manageable chunks. The keys to the effectiveness of this methodology are its use of a prototype as the basis for negotiating user requirements, and its well-defined deliverables for each of the seven project phases to ensure that all goals are met.

The Top 10 Symptoms of Ailing Development Projects

1. **Immeasurable success or failure:** Unless you define metrics with which to measure your project's cost, return, and risk, and take "before" and "after" snapshots, you'll be unable to quantitatively assess your project's performance. Why is this important? Knowledge of past project performance will allow you to evaluate and prioritize *future* projects, and demonstrate the effectiveness of efforts to improve the consistency and efficiency of your project teams.

2. **The never-ending project:** Most development projects should require only one or two "discovery" sessions to ascertain requirements, and one or two design walkthroughs with stakeholders to validate an application's design. If your post-construction walkthroughs tend to yield major design changes, evaluate whether your teams are spending enough time up front solidifying and validating both requirements and design.

3. **Software needs frequent repair:** Look at the number of transports you've done within two, four, and six weeks after go-live. An abundance of transports usually indicates insufficient testing. Sometimes this is due to a lack of planning, sometimes to a lack of effort, and sometimes to a lack of production-quality data with which to test.

4. **Only one or two developers "know" the code:** It's fine for only one or two developers to "know" a particular piece of code — as long as the code is well-structured and documented so that new developers will have little trouble picking up where their predecessors left off. Unfortunately, most code (including that behind many R/3 transactions) isn't well structured (or documented) and progress is threatened due to a dependence on the availability of one or two individuals.

5. **The application can't be easily reorganized, upgraded, and maintained:** Improperly structured code does not easily accommodate design changes, upgrades, or enhancements. Many changes require entire rewrites to sections of code. In practice, however, such rewrites are rare. Instead,

✓ *Tip*

To help you implement this development methodology, I've prepared a "quick reference" that is available for download from ***www.SAPpro.com****. It includes the following:*

- *Detailed descriptions of the custom development project phases*
- *The R/3 Custom Development Project Methodology Overview chart (see Appendix A on page 23)*
- *The R/3 Custom Development Project Document Overview chart (see Appendix B on page 25)*
- *An overview of the risks of R/3 development projects*
- *The top 10 symptoms of ailing development projects (see the sidebar above)*

developers usually apply "Band-Aids" to the code for one of two reasons: first, they may not fully understand (or remember) what other areas of the code do; second, they don't want to break sections of code that already work properly in production, however poor their structure.

6. **Forgotten steps:** If you get to production only to find that you've forgotten a step, like requesting that a new security profile be added to your end users' IDs, you need a methodology. Errors like these are usually easily fixed, but they can introduce significant delays and cost overruns.

7. **People are working nights or weekends:** If team members are working nights or weekends, it means either the schedule is too aggressive, the project is plagued with design or scope changes, the team is inefficient, or unexpected events have everyone struggling to meet the deadline.

8. **Red-flag phrases:** Proceed cautiously if you ever hear any of these phrases used on your project: "It's a training issue"; "[Don't worry about coding for that since] it won't happen"; or "[Let's just make it work, then] we'll clean it up later."

9. **Users aren't happy with the solution:** If your users don't feel that your solution will help make things easier, they're less likely to use it properly (or at all). End-user satisfaction is *essential* to the long-term success of your project.

10. **Too much or too little documentation:** Proper documentation of key project deliverables is essential during both the construction and support phases of a project. The problems to be solved, the decisions to be made, and the design of the software should all be documented thoughtfully, but not excessively! Too much documentation is detrimental and becomes a chore; people inevitably end up avoiding it altogether. For example, including code and pseudo-code within technical specification documents guarantees their obsolescence and rarely aids new developers. Review your documentation standards in this light to ensure that they contain only what is essential.

Phase 1: Analysis

In the analysis phase, the goal is twofold:

1. Gain a full understanding of the potential project.

2. Determine whether the potential business benefits sufficiently exceed the potential risk and cost.

Through meetings with all relevant stakeholders, an analysis team uncovers users' needs, weighs alternative approaches, identifies goals and roadblocks, assesses risks, and defines target costs and benefits. The analysis team may decide to forgo the project if a clear problem and appropriate solution cannot be identified or if an acceptable ROI cannot be achieved at a reasonable level of risk, for example.

> ✓ ***Tip***
>
> *The analysis team should reflect a cross-section of business, functional, and technical experience to ensure a practical, well-thought-out outcome. Many development teams have only the business or functional analysts solicit end-user requirements. Even at this early stage, it is important to understand the capabilities of various technologies in comparing potential solutions, so including designers and developers is critical.*

As the costs of making changes during this phase are relatively small, take care not to rush through this phase. Proceed as follows:

1. **Hold a "discovery" meeting.** Once a potential project opportunity has been identified, gather your analysis team and schedule a discovery meeting with all relevant stakeholders. During the meeting, assure the stakeholders that you are there to listen and help if you can. As they tell their stories, ask detailed follow-up questions just as a doctor would during an initial consultation: you're there to get to the problem quickly, but politely. You'll be surprised by how people open up to you (and even thank you!) for listening and trying to help make their lives easier.

   If one person is doing much of the talking, confirm what he or she is saying by asking others in the room how their experiences have been the same or different. Guide the conversation, and allow your analysis team members to ask questions. Don't allow anyone to discuss potential solutions, however, until you've heard the whole story.

   Once your team feels they understand the issue, thank the participants and adjourn the meeting; then meet with your analysis team immediately. What is the true problem resulting in the symptoms to which the users are reacting? What are some possible solutions? Is a solution likely to have measurable, tangible benefits to the *business* (i.e., in addition to the end users) that exceed its cost and risk? If so, proceed to the next step.

2. **Investigate the problem.** Hold one or two additional meetings with each group of stakeholders (e.g., end users, managers) to discover and document users' needs more thoroughly. Make sure you sufficiently pursue answers to the following questions:

   - What are the symptoms being reported? What are the underlying causes?
   - What are the dimensions of users' needs (by role/division, by country, by person)?
   - Who else is experiencing these problems or needs (i.e., other possible end users)?
   - What business processes are involved and are they contributing to the problem?
   - What functional and technical gaps currently exist?
   - How are users currently working around these gaps?

3. **Initiate an analysis document for the project.** The analysis document collects critical project information, including:

   - Project name, location, date of initiation
   - List of team members with roles and phone numbers
   - Project details such as background and sponsor
   - Project goals
   - End-user details: names, roles, locations, languages, etc.
   - Observations
   - Recommendations: process, packages, development, etc.
   - Software development needs
   - Project metrics
   - Relevant standards, procedures, quality control measures
   - Projections: costs, benefits, risks, target ROI
   - A "before" snapshot of the metrics
   - Future opportunities for enhancement or deployment

   Include any observations that your analysis team made that you believe might be useful — for example, "It was observed that the ability of customer service reps to input orders was very frequently made more difficult by missing or 'miscategorized' material master records."

4. **Draft a step-by-step flowchart.** To understand a business process and its decision points, it is usually helpful to create a flowchart. It takes very little time in PowerPoint or Visio, and it often helps you to uncover details that might have otherwise been missed. See page 27 in Appendix B for an example of an "as-is" flowchart.

5. **Evaluate solutions.** With your analysis team, brainstorm possible solutions at a high level. In particular, research available R/3 functionality and relevant current and future vendor offerings. Weigh the costs, potential benefits, and risks (both short and long term) of each solution. Use the following questions as a guide to finding and evaluating solutions:

   - Can a full or partial solution be achieved through process changes?
   - Does standard R/3 functionality exist to address the software needs?
   - Is there a packaged solution or R/3 enhancement in the works for this problem?
   - What technologies and approaches could be used in developing a custom solution?
   - Could any existing assets be reused in this custom solution to reduce its cost?
   - What are the costs, risks, and potential benefits of possible solutions?

6. **Formulate recommendations.** Add your team's recommendations to your analysis document. To expedite your design walkthroughs, your recommendations should be directly (and obviously) tied to your observations, process flowcharts, and other documented findings.

   For example, your team might recommend that specific procedures be implemented to ensure that the team responsible for material master data keeps their data more up-to-date. This is directly tied to the observation that the delay in entering material master changes is a primary cause of a symptom reported by customer service reps (our earlier example).

   Following this guideline ensures the appropriateness of your recommendations and that external influences (such as politics) on the course of action are minimized. For each of the possible solutions that have been identified, answer the following questions:

   - Should the issue be left open due to concerns about ROI or risk, or in anticipation of a packaged solution?
   - What processes should be changed?
   - What R/3 functional changes need to be made or new functionality implemented?
   - How will software gaps be addressed (e.g., implementation of a package solution or development of one or more custom applications)?

   As a team, assemble a solution consisting of one or more of these components (i.e., process changes, packaged solutions, custom development). If any business processes will be improved, make sure to draft a "to-be" flowchart for each, and compare it to the "as-is" version previously drafted. Finally, briefly document the solutions considered in your analysis document and how the final approach was decided upon.

7. **Define what will constitute success.** If you have not already done so, draft a list of specific goals for the project. Is the goal of the project to streamline user interaction with the system? To improve data quality? What's the target ROI, cost savings, or other measure?

   In a separate section of the analysis document, identify and forecast the metrics that will be used to gauge the project's success. In addition to things like cost, risk incurred, and financial benefit achieved, include metrics for user satisfaction or for the skills and/or experience that should be gained by inexperienced project members.

8. **Validate your understanding of the project and get stakeholder buy-in.** With your analysis document nearly complete, hold one or two walkthroughs with all stakeholders to review your observations, resolve any questions, and discuss your recommendations. Your team has succeeded if by the end of the meeting the following things have happened:
   - End users and their managers seem confident that a true solution is on the way.
   - Those funding the project are confident of the costs, risks, and potential benefits of the investment.
   - Your team is energized by customers' reactions and anxious to take the next step. (This sounds idealistic, but I have seen this happen on several projects.)

9. **Take a "before" snapshot.** Take a "before" snapshot for each of the project's metrics, and add this final element to your analysis document.

Analysis Phase Pitfalls

Watch out for the following pitfalls during this phase:

☑ **Forgetting the "investment" mindset:** Perhaps the most dangerous pitfall is viewing your development project as anything other than an investment on which return is sought. For all projects, regardless of size, you should anticipate costs, assess risks, identify metrics, and take "before-and-after" pictures. Remember that, as an IT organization, *your primary customer is the company; your secondary customer is the end user*. That is, projects should aim to satisfy end users but should more importantly have a demonstrable, positive impact on the business' profitability. Many a project has forgotten this.

☑ **Solution looking for a problem:** New technologies, servers, and even versions bring exciting new functionality, bug fixes, and other opportunities to make life easier and more profitable. But with new opportunities come costs and risks as well. Make sure to consider the costs, risks, and potential ROI of each upgrade or new endeavor — both in isolation and with respect to the project as a whole — to ensure there is a problem or need (with business impact) that would justify its implementation.

☑ **Rushing to build:** It's tempting to judge the progress of a development project by examining how much code has been written or how quickly a solution appears. Developers, in particular, want to build almost immediately, designing as they go rather than in advance on paper. Rushing to build in either of these ways is almost always fatal to a project's ROI. Changes are inevitably made where they are most costly — during construction or testing instead of during analysis and design.

> ✓ ***Note!***
>
> *This article's methodology forces you to solidify your design before writing a line of code. Approximately 40% of your project should be complete before ever visiting the ABAP Workbench. This may seem to slow projects down at first, but recognize that the design work performed up front nearly eliminates the never-ending syndrome of recoding in response to design changes and drastically reduces testing and support issues. As a developer, I've experienced this in practice!*

☑ **Forgetting to establish metrics:** Without metrics for comparing the "before-and-after" pictures of your solution, you will rarely be able to demonstrate its effectiveness or calculate its ROI.

☑ **Lack of knowledge about R/3 functionality:** Many a development project has sought to redevelop standard R/3 functionality simply because the team members were unaware it existed. You can reduce the likelihood of this happening by taking three steps before developing anything:

1. Ask other team members if they're aware of standard functionality or tools to accomplish the task.

2. Search R/3 help for your R/3 versions (as well as subsequent ones!) using **http://help.sap.com**.

3. Support cross-functional and cross-technical awareness within your teams through knowledge-sharing sessions and training.

Phase 2: Design

Having exercised due diligence in researching your project, the design team can now begin designing any custom applications called for in your recommendations.

Using the findings of the analysis phase, the design team fleshes out a prototype design to review with end users and other stakeholders. Current end-user needs must be carefully balanced with the need for a flexible, expandable design. The prototype is usually a visual representation of the actual product, recognizing that users typically don't know what they want until they see it. Stakeholders with distinct needs and interests can negotiate openly about the design until a construction-ready version is solidified.

> ✓ ***Tip***
>
> *It is important that the prototype usually model the application as closely as possible. Good prototypes help users spot issues they would otherwise miss if they had to imagine the product.*

Before construction begins, designers and developers together prepare technical specifications as a battle plan, and as a means to expose and resolve design flaws. Because the costs of changes will rise significantly once construction begins, it's important to avoid design flaws by carefully following these steps in the design phase:

1. **Create a prototype document.** Begin by creating a starter prototype (i.e., a realistic visual representation of how each application screen will look). Include all images, fields, buttons, drop-down value lists, and static text — just as they would appear on the final application screen. Depending on how realistic your prototype looks, you may need to explain to users that the prototype merely consists of "pretty pictures" and that no coding has actually begun.

Why Create a Prototype?

Creating a prototype is time well spent, for all of the following reasons:

☑ **Users usually don't know what they need until they see it.** I used to spend a large amount of time on change requests *after* construction, despite intense efforts to verbally clarify users' needs beforehand. Once I began prototyping, post-development changes seemed to be reduced by over 50%. There's something about seeing the product that helps users communicate their true needs.

☑ **Design tradeoffs are exposed and resolved early.** Prototyping forces you to resolve common design "gotchas" and lets you try out different ideas without commitment.

☑ **You are forced to fabricate site content and images early.** Graphics, icons, and static content should be created early in the design phase — especially for graphic-intensive applications such as web applications. Waiting until construction to wrestle with these content issues disrupts the entire development flow. A prototype document allows end users to see the "final" product before it's built, and it allows development to proceed uninterrupted.

> ✓ ***Tip***
>
> *I usually create a prototype document in PowerPoint. To depict fields, buttons, checkboxes, or other controls, I cut actual pictures of the objects from screenshots of web pages or R/3 screens as needed. I paste the screenshot into Microsoft Paint, lasso the desired object, and then paste it into my prototype document. I also add these graphics to a central bitmap file for future use.*

2. **Negotiate the design with stakeholders.** Hold one or two meetings to review your design with stakeholders. Guide these meetings by walking through the screens one by one and by visibly recording users' comments in a notebook. When users see that their comments matter, they share their opinions and ideas more openly.

3. **Develop technical specifications.** With the design solidified, your designers and developers are ready to create technical specifications. Create one technical specification document per application (e.g., module pool), beginning with a high-level description of the component and how it fits into the overall solution.

   Include enough detail for the specifications to be useful, but not so much that they become a chore to maintain. In particular, *do not* copy code into the specifications, as it almost instantly becomes obsolete. (I am also wary of some flavors of "pseudo-code." High-level everyday language describing the processing in steps usually suffices, without too many details of the "ifs" and "elses.")

   I draft technical specifications in PowerPoint and include the following information:

   - Application description
   - Features
   - How the application will be invoked
   - Future direction
   - Application flow diagram
   - List of components (e.g., programs, screens, exact names)
   - Module flow diagram
   - Screen elements (e.g., fields, buttons)
   - Processing (e.g., initialization, defaults, validation, events)

   Finally, add screen details to the document. Since I create both my technical specification and prototype documents in PowerPoint, I am able to copy each screenshot slide directly into the technical specification without modification.

How Design Flaws Impact Code Quality

Despite diligent efforts in the analysis and design phases, some issues will result in design changes or additional analysis work to determine a course of action. Besides being costly, these design changes may require portions completed in other areas to be reworked, or worse, discarded entirely.

In the interest of minimizing project delays and additional costs, this "rework" often seeks to preserve what's been done and patches are applied or a component is modified to do something other than its original function. The effects of these "Band-Aids" become evident only after a period of time, through increased failures and maintenance costs. Enhancements are also made more difficult and the compromised components are often retired or replaced instead of reused.

The Value of Technical Specifications

Technical specifications are not very exciting to write, but are essential for properly structuring applications. As a result of writing specifications, you gain the following benefits:

- ☑ Designers and developers become aware of design inconsistencies and they have an opportunity to consider the tradeoffs of different approaches.
- ☑ Designers uncover design flaws or have the opportunity to make design changes to ease development.
- ☑ Developers gain a deep understanding of the designer's intent.
- ☑ Because the specification serves as a contract between the designer and developer, it relieves the developer from having to remember design details.
- ☑ The specification documents the essential software design details for those maintaining or enhancing the code in the future.

I then detail the purpose, elements, attributes, events, and processing of the screen in one or two slides positioned behind each prototype screenshot.

Design Phase Pitfalls

In the usual rush to build, it is often all too easy to treat the design phase as optional. I have seen many teams skip this phase almost entirely and instead design *while writing code*. In most cases, the result is a mediocre solution at best, with code scarred by major design changes during final walkthroughs. With that said, beware of the following pitfalls:

☑ **Not using a prototype:** Prototypes let your users see what they'll get in advance and help ensure that what gets built is what was designed. Use a prototype as the basis for negotiating a design with users.

☑ **Failing to design an application that works the way your end users do:** At one time or another, we have all complained that a system we have used doesn't work "the way we do." Usually this is because the designers (or developers) were either working from an incorrect mental model of how we might use the system, or they arrogantly decided that we should work the way the system does, not the other way around. The former is common. Typically, it is recognized and rectified during usability testing (if it is done). The latter (unilaterally deciding that users should work the way the system does) almost inevitably assures the solution's failure.

☑ **Forgetting to include future stakeholders:** Good solutions inevitably draw attention, and users on other teams or in different locations may want to adopt them. Anticipate this trend and contrast the needs of potential end users with those of the initial group. Design sufficient flexibility into the application to enable changes that might need to be made to accommodate other types of users and make conscious tradeoffs about what functionality will be initially in and out of scope.

☑ **Letting politics reign:** During the analysis and design phases, evaluate opportunities for redesigning

processes and technologies on the basis of costs, risks, and ROI. In particular, avoid making decisions based on politics. Beware if you hear phrases like, "So and so won't be happy if we do this" or "Let's go this way because so and so owns this area."

Phase 3: Construction

With user expectations set and technical specifications in hand, development can now begin. During construction, teams consisting of individuals of various backgrounds use design-phase outputs (i.e., the prototype and technical specifications) to build the product. The construction task itself is approached in stages, as each layer is built, reviewed, and unit-tested (i.e., tested to determine that the component works in isolation).

The construction phase should pretty much run smoothly "in the background" without end-user involvement. Questions and/or problems almost always arise, however, and developers will occasionally need to consult with designers or end users to clarify design points and/or resolve issues. While minor issues should be expected, major issues may prompt you to review your analysis or prototype documents. As always, the sooner that issues are caught, the less costly they are to resolve.

Follow these steps:

1. **Build the code … correctly.** I can't stress enough the need to code defensively (i.e., modularly, cleanly, and generically, in anticipation of changes and future reuse). Failure here is costly and is rarely remedied afterward!

2. **Prepare a test plan.** During development, the application's designers should draft a baseline functional test plan by going through each prototype screen to identify *test scenarios*. For example, be sure to test all expected input validations (e.g., "user enters bad data"); functional scenarios (e.g., "multi-line sales order missing batches is entered"); and processing paths (e.g., "user hits cancel").

> ✓ ***Tip***
>
> *Initially, developers should write their code based on the technical specifications, with an eye toward getting it to work. I define an application as "working" when it satisfies the functional test plan to which it will be subjected.*
>
> *So when an application is close to completion (e.g., approximately 80%), it is useful to switch to the test plan (discussed in "Phase 4: Testing") as the basis for deriving what still needs to be done. This will avoid the inevitable slowdown that occurs when developers lose precise track of what they have and have not coded and unit-tested in the application.*

The test plan document identifies test steps to be performed while logged on with test user IDs. Scenarios can be as high-level as "multi-line sales order" and as technical as "bad data," in which the tester is instructed to place characters in date fields, for example. The plan should consist of:

- Test steps per module, scenario, or role
- Test data
- Expected results
- Test problems and assignment (if tracking manually)

In particular, don't forget to include:

- Role-based and/or personalization features
- A note, where appropriate, that date and times are handled in *local* time, not *system* time
- Internationalization features like date formatting and multi-language support

3. **Secure stakeholder approval.** As soon as possible after the application is acceptably functional, hold one or two walkthroughs with your stakeholders to gather feedback. *Don't* wait until the application is fully tested, because a short list of changes is expected at this point and retesting will be required anyway.

4. **Request test user IDs.** As soon as end-user security requirements can be enumerated, request appropriate profile or authority group changes from your security team. Have them set up one or more test user IDs for each role in your development *and* test systems. You will use these to execute your functional test plan in both systems.

5. **Fix your test plan until it executes cleanly in the development system.** Before transporting *any* code to the test system, ensure that the test plan executes without any errors in your development system. This will minimize the number of transports and downtime waiting for fixes to move to the test system.

Construction Phase Pitfalls

For developers, construction is the most exciting phase, which is why it's especially important to carefully follow all of the principles (outlined above) that help ensure quality and proper design implementation. Beware of the following common pitfalls:

☑ **Not having good data to develop with:** I cannot emphasize this point enough. The quality of the data on which a solution is built and tested *is one of the most significant determinants* of the number and severity of testing and production issues. To minimize the cost of correcting design flaws and programming bugs, it is essential to test in the *development* system using realistic data. This is usually difficult since development systems typically lack production-quality master or configuration data.

☑ **Improper coding technique:** Unless code is developed in a clean, consistent, modular way, immediate and long-term maintenance and reuse will be difficult. Beware of developers who claim they'll "clean it up later" — time rarely permits. To ensure consistency and proper technique, have one or more experienced coders with a reputation for detail conduct reviews of the team's code. It is also useful to develop and enforce coding standards for your organization for things like function, program, and variable naming.

☑ **Not keeping an eye on the critical path:** Delays often result when steps are skipped or activities are done out of sequence. Keep an eye on the critical path of your project and use a methodology such as the one described in this article to avoid such mishaps.

☑ **Not having construction specifications:** Developers and designers should work together to create technical specifications *before* developing a single line of code, as described in the earlier section on design (Phase 2).

☑ **Forgetting to document on-the-fly changes:** Design changes are inevitably made during construction and after walkthroughs with end users. Make sure to update all analysis, design, and construction documents to reflect these changes.

Phase 4: Testing

Testing is an opportunity to minimize your go-live headaches and ensure user satisfaction. In this phase, code is first exposed to real-world data to test its operation, usability testing guarantees its usability, and performance (or stress) testing ensures its robustness and scalability.

Beyond testing each component of the product in isolation, a system-wide test of the entire solution is generally executed. The test is usually guided by a

test plan that identifies the major components and scenarios to be tested. Assembly of the plan starts in the design phase, when user requirements and project goals are most clearly understood. It is finalized during construction as the product materializes and unanticipated issues are resolved.

Testing is perhaps the most underrated of the project phases. Most technologists will tell you that the purpose of testing is to ensure code works as it was designed to. But testing serves a much more valuable purpose — it validates the design of the software and exposes missing or incorrect assumptions! Given that changes of any type become more costly as projects progress, it follows that design issues need to be caught and resolved as early as possible rather than waiting until testing or production to uncover problems.

In this role, testing is most valuable when done *during* development, instead of *after* coding is complete. While writing code and unit-testing its components, developers become intimately familiar with the assumptions made in the design. For example, when writing a data retrieval routine, by examining a table's key fields a developer will know whether the data on hand is sufficient to return a unique record of data, or if multiple records might be returned. The designers may have assumed only one record would be returned and important design decisions, such as a screen's layout, could have been based on this assumption. Design flaws such as this can be hard to catch and can be very costly if not addressed early in the project.

Gotcha!

Essential to this effort is use of "production-quality" data to represent the spectrum of what will be encountered in production. But this ideal is rarely achieved. Testing is generally viewed as a chore and creation of real-world test data can be difficult, especially in development systems with incomplete configuration and master data.

Four-Point Plan for Getting the Most Out of Testing

1. **Sell the concept of "early testing."** Convince your team that the time spent testing properly will actually help avoid major changes (e.g., "evening and weekend" catch-up work) during testing or go-live. Explain how catching critical design or coding bugs early (during construction!) will really keep costs down.

2. **Have developers perform unit-testing.** Developers usually do this to some degree, running their creation a few times to see if it works. But to truly weed out first-round bugs and design flaws, you should provide a spectrum of realistic test data to your developers and have them try to "break" the code with unexpected entries (e.g., filling a quantity field with 9s or leaving all fields empty) before declaring the component "done."

3. **Build data around functional scenarios.** Brainstorm possible test scenarios with your team and document them in a test plan. Assign one or more people to create realistic test data in the development and test systems. This step *cannot* be skipped since your developers will need to test each unit as they build it.

4. **Execute your test plan in development, before transporting.** Before transporting an application to test, run through your complete test plan and resolve any bugs or design issues. You'll be surprised at how many issues you catch. Resist the urge to wait until transporting to the test system to do your first few test-plan runthroughs; do them in the development system instead.

Types of Testing

Three types of application testing are critical:

- **Functional testing:** Your functional test plan defines what it means to say, "It works!" To be more specific, it ensures that your application functions as designed by validating the *code* portion of your application and at the same time determining that all of the functionality specified in the design has been included in the final product. "System testing" describes functional testing in which testing is conducted across application boundaries, simulating the actual business processes in which the application(s) will be used.

- **Usability testing:** The purpose of usability testing is to validate the user interface design of the application and to ensure that the application is "easy to use." You will observe end users using your application(s) to identify stumbling blocks, training issues, etc. This testing can usually be considered "user acceptance testing" as well; while you'll be evaluating how they use the application, they'll be commenting on how well the final product meets their needs and expectations.

- **Performance (or stress) testing:** Your application's robustness can be demonstrated in several ways. One approach is to perform a full stress test using a tool such as SAP's Computer Aided Test Tool (transaction CATT). Stress tests simulate the activity of a large number of users, often taking different paths through an application. Bottlenecks can be identified on a macroscopic level by observing such things as server loads and application or database processing time.

  Alternatively, you can perform a runtime linearity test, wherein you essentially run your application repeatedly against small, medium, and large data sets to ensure that execution time increases *at most* linearly. Exponential growth in processing time can indicate trouble, especially since the volumes of data processed in production are generally far greater than those available in a "test" system.

  Several R/3 tools are available to help you identify bottlenecks and improve performance, including the following:

  - **SM51 (Work Process Inspection):** Shows a summary of the time spent in different operations, such as database reads and inserts for a running application session.

  - **SE30 (Runtime Analysis):** Allows you to quickly identify the most expensive database and internal processing sections within your code. By viewing detailed benchmarks in a sorted list, you can identify and correct expensive internal table operations and database queries.

  - **ST05 (Performance Trace):** Uncovers expensive database operations such as nested or redundant SELECT statements within your code. You can also use it to review the code execution plan of your queries (e.g., which table index was used).

Testing How-To

Assuming that you got your test plan to execute cleanly in development (step 5 of the construction phase), follow the steps below to complete the testing phase:

1. **Set up testing:** Transport your ABAP code and other application objects to your test system. If you have not already done so, create sufficient test data to reflect the diversity of data to which your code will be exposed in production.

   By this point, your security team should have already set up in this system one or more test user IDs with which to conduct your tests. These user IDs should model your end users' IDs as closely as possible (i.e., with respect to security profile

values, time zone, and parameter defaults that are set up in transaction SU3 or SU52, depending on your system). Using these IDs for testing is essential to detecting hard-to-find security-related issues or those relating to users' geographic locations.

2. **Perform testing:** For maximum efficiency, perform all of your testing in cycles. Prepare your data. Gather your testers. Execute the test items, recording issues as you go. It is important to not spend too much time investigating or discussing problems at this point. Simply log issues in sufficient detail and move on.

   Once your testers have completed the plan, or have encountered enough "showstoppers" to warrant stopping early, collect the list and assign a severity to each issue. I typically use gray (low), yellow (moderate), and red (severe). I don't use green for low severity since it might be mistaken to mean that the issue has been resolved.

   Usability testing is best done with a group of untrained end users in an environment that closely resembles the productive environment (i.e., don't perform tests in a room with 18-inch screens if your end users have 12-inch screens, or in a room with normal lighting if the productive environment is a dimly lit warehouse).

   Observe and document users' questions, points of difficulty or misunderstanding, and the speed with which untrained users move through the application. In particular, observe how quickly and easily users seem to be able to input or access data in frequently used areas of the application. Afterward, discuss the test's results and users' opinions with your team and make prudent design changes.

3. **Work through your prioritized list of issues:**

   a. **Delegate issues as a team.** As a team, agree on who will assume responsibility for each issue. If the list of issues is large, it is usually best to simply assign a group of issues of a particular type (or for a particular piece of the application) instead of assigning each issue individually.

   b. **Clarify issues as a team.** Before adjourning your team, allow each person to review his or her list of issues and to ask questions. It is important to do this as a group, for two reasons: (1) some team members may have witnessed a bug that is not clearly explained on the issues list; and (2) developers can propose solutions on the spot for issues that may have an impact on areas assigned to other team members.

   c. **Resolve the issues.** This is a two-part step. For each issue, do the following:

      - First, unless an application behavior that has been flagged as an issue to be addressed is obviously a bug, developers should check to see whether or not that behavior is actually called for in the design. For example, an issue like "Cancel button takes users back to the first transaction screen instead of the main menu" may be exactly what the technical specification calls for. If this design decision now seems undesirable, developers should flag the issue to be resolved with designers or stakeholders at a later time.
      - Then, developers should reproduce the issue in the development system. After identifying the problem and making any necessary code changes, the functionality should be unit-tested with the same data used to demonstrate its malfunction — *yet another reason why it's important to have real-world test data in the development system*.

   d. **Execute a new test cycle.** When all issues have been addressed, it is usually best, if the

list was large, to re-execute your entire test plan. As the issues become smaller in number and scope, only incremental retests need to be performed. When retesting, be sure to test all areas that could potentially be affected from a technical, not functional, perspective. In other words, retest all five areas a piece of code is invoked from, even if from a business standpoint the change relates to only one area specifically.

Repeat this process (step 3) until the test plan executes cleanly, users are satisfied, and performance is acceptable.

4. **Update project documentation.** Once testing is complete and the design is solidified, perform a final review of the analysis, prototype, and technical specification documents. Compare the final product to the plans and update them as necessary.

Testing Phase Pitfalls

Beware of the following common pitfalls:

☑ **Not tracking test issues:** It may seem obvious that issues discovered during testing should be tracked, but some teams don't formally track issues until after go-live. In my experience, this inevitably leads to missed, misunderstood, or incompletely verified issues. See the sidebar on the next page for ideas on how to track test issues.

☑ **Forgetting to reconcile the product with the design:** Often what gets built isn't what was designed. By the time construction is complete, the decisions of initial design meetings are usually forgotten. But users expect the product to match the design they agreed to. Deviations can leave users frustrated, or worse, angry that their input was ignored or changed. Before conducting walkthroughs of the finished product, reconcile the product with the design and research, and document reasons for any deviations that remain.

☑ **Not preparing a test plan with scenarios and roles:** When thinking about testing, put yourself in the end users' shoes. Imagine users' needs by role, division, or geography, and consider whether what has been built will meet those needs. Walk through each of the application's input and output screens, imagining how the application will be used, what the user may do wrong, or what failures can occur. For example, leave input fields blank and click the "Save" button. Place characters in date fields. Fill quantity fields with 9s to test overflows. Users will do all that you can imagine and some things you can't.

☑ **Not testing on realistic data:** As mentioned earlier, having production-quality data with which to test is essential to catching issues and bugs before they become costly. This is not easy to accomplish, but it is critical. Build your data around the scenarios identified in your test plan to avoid missing anything.

☑ **Not testing software for usability:** Many teams forget that how an application will be *used* is as important as what it *does*, and that users' impressions of a product's quality is directly tied to how easy it is to use. The best way to identify areas for improvement is to observe users in a pseudo-productive environment.

☑ **Not testing with actual user IDs:** Remember to execute your test plan while logged on with an end user's ID. End users typically have greatly restricted access in production; testing with developers' user IDs will not expose these problems. Request sample user IDs to be set up in your development and test systems with the same profile(s) you will request for your users.

☑ **Developers testing their own code:** There are two main dangers in having developers execute anything other than unit tests (i.e., testing their own code in development to ensure basic operation):

- Developers tend to test a piece of code with the same set of actions and the same set of data

repeatedly. Forcing people other than the author to test is more likely to expose failure points.

- Developers tend to be able to identify only technical bugs like short dumps or missing output data. Only functional experts and end users will be able to identify missing functionality, or indicate how well the solution will meet their needs. So while developers should ensure operation of their code in development, they should not be relied upon as the sole source of testing.

Phase 5: Deployment

In the deployment phase, end users are trained, the product is released for general use, support procedures are established, support teams are on heightened alert, and major issues are resolved. Frequently, an initial pre-deployment (or "pilot") allows major issues to be caught before general release.

Think about some issues you've had during past go-lives. Could they have been prevented by more thorough analysis or design work? Through more thorough testing? Perhaps experience has taught you as well that care in soliciting requirements, validating the design, properly structuring code, and thorough testing with realistic data are the keys to trouble-free rollouts.

Here's what to do during this phase:

1. Register the application with the support desk.
2. Transport code and other objects to production.
3. Have production user IDs linked to new profiles.
4. Train users.

Some Tips for Managing Documents and Tracking Issues

Here are a few final issues to keep in mind throughout the project…

Creating Project Documents

You may notice from Appendix B (see page 25) that each of the methodology's documents has been drafted using a Microsoft Office application like PowerPoint, Word, and Excel. While other, specialized tools could have been used, using these applications ensures the documents' portability to PCs that might not have specialty software like Microsoft Project or Visio. Also, drafting the analysis and prototype documents with PowerPoint enables their immediate use in walkthroughs using a projector.

Managing Project Documents (Important!)

It is very important to manage project (and organizational) documents effectively, but few teams do. Most teams simply place documents somewhere on a shared drive, don't track or coordinate changes, and rely on shared-drive backups as their strategy for archiving. Furthermore, team members often make local copies of documents and email copies to others. This leaves everyone wondering whether his or her version is the latest available.

All of this is easily avoided using a document management system (DMS), which can be downloaded and implemented within a day in many cases. DMS package prices range from free to $50,000 or more, depending on your needs and the skill of the salesperson you end up talking to, so shop carefully.

5. Resolve issues found during training.
6. Perform pilot rollout if desired.
7. Roll out the application for general use.

Deployment Phase Pitfalls

Beware of two commonly forgotten steps:

☑ **Forgetting to register the application well in advance with the support desk:** For obvious reasons, it is in everyone's best interests for the support desk to be well prepared before going live.

☑ **Forgetting to request security updates for your end users' IDs:** This task can be easily overlooked in the rush of deployment. You need to provide the security team with early notice of the list of end user IDs to update with new profiles (activity groups). Test all new activity groups first, using test user IDs in your test system.

Phase 6: Support

After the number of support issues per week falls to a reasonably low number, team members usually move on to other projects and transition their responsibilities to support personnel (or they become the support personnel). Beyond responding to production issues, the support phase is an opportunity to reflect and measure. The stress of go-live has subsided and the application has been in place long enough for users to gain experience.

Take this time to assess your project's success against the metrics you defined in the analysis phase document by taking an "after" snapshot and adding it

All DMSs use a library concept, allowing team members to check in and check out documents from a central repository, record notes about what changes were made, and retrieve previous versions instantly. Many allow you to keep local read-only copies of the documents and advise you when your versions are out of date. Using a DMS, everyone always knows *where* project documents can be found and *what* the latest version is!

Many teams use Lotus Notes as a DMS, but Notes does not excel in this role, in my opinion. DMSs are often integrated into the desktop, allowing you to check in and check out documents right from Windows Explorer. Plus, DMSs are more efficient as they save only *changes* to files to disk. Notes attachments are complete copies of the newly revised document.

> ✓ ***Tip***
>
> *For personal document management, I have had great success with CS-RCS, available from Component Software (**www.componentsoftware.com**). And no, I don't get a kickback for saying so.*

Tracking Project Progress and Issues

While tools abound for managing project phases, tasks, and issues, I find a well-thought-out "issues" document drafted in Microsoft Word to be sufficient in most cases.

Such a document tracks:

- Project tasks and milestones
- Overall project status (percentage complete and target date next to each)

(continued on next page)

to your analysis document. Through comparison with the "before" snapshot, compute your current ROI and put a plan in place to periodically recalculate the ROI.

Support Phase Pitfalls

Beware of the following common pitfalls:

☑ **Failing to adequately track production support issues:** Production support issues must be tracked with a bit more diligence than issues that arise during the testing phase. There are specialized packages designed for call-center use and many automatically determine the appropriate support contact by application, degree of difficulty, time of day, and other criteria. Regardless of the package used, instruct your support desk to solicit problem information similar to that recorded during testing.

☑ **Forgetting to take an "after" snapshot:** In the rush to move on to new projects, it's easy to forget to periodically take an "after" snapshot to evaluate your project's success (it can also be scary at times!). If you've truly succeeded and can demonstrate your success, it's a great feeling and exceptional leverage for negotiating a raise!

Phase 7: Enhancements

During the final phase, enhancements collected throughout the project are considered (e.g., features that were placed out of scope during analysis). Each one should be evaluated on the same basis as the initial product with regard to potential benefits, costs, risks, etc.

(continued from previous page)

- Open issues, questions, and to-dos (and who is assigned to each)
- Resolved issues

Which tool you choose will ultimately depend on your familiarity with the tool and the need for specific functionality. But take care not to "overdo" task scheduling lest it require an unjustifiable amount of time to maintain.

Tracking Test Issues

It may seem obvious that issues discovered during testing should be tracked, but some teams don't formally track issues until after go-live. In my experience, this inevitably leads to missed, misunderstood, or incompletely tested issues.

But how should issues be tracked? Some teams use a Lotus Notes database, some use a proprietary software package, and others use an Excel spreadsheet. There is no "right" way, but ideally the approach should involve as little "paperwork" as possible, and focus on capturing the essential details of the issue and providing a list of outstanding and resolved issues.

When working with companies with small development teams, the easiest way has been a "manual method" — have testers print out a screenshot demonstrating the issue and write on it the following:

1. R/3 system
2. Logon user ID

Here are the recommended steps for collecting and evaluating enhancements:

1. Collect potential enhancements for consideration throughout the project:
   - Ask current end users for suggestions. They are the ones who will be using them!
   - Review project documents for notes on functionality that was earmarked "to do later" or initially placed out of scope.
2. Evaluate proposed improvements on the basis of the impact they will have on the project's overall ROI.
3. Thoroughly test (for functionality, usability, and performance) all code changes before transporting them to production.

Enhancements Phase Pitfalls

The most common pitfalls of this phase are:

☑ **Only recording enhancement ideas at the end of the project:** The ideas for future enhancements that arise in the course of a project are seldom remembered afterward — unless they are recorded. This is especially true of enhancements thought of during construction and walkthroughs with end users. Record these ideas in a document or repository that is accessible to everyone, and encourage its use.

☑ **Not evaluating each enhancement on its own merits:** When considering enhancements or design changes, consider how each one will affect the solution's ROI. Some will involve more risks or costs than benefits and could threaten an otherwise successful project.

3. Application or transaction involved
4. Input data and actions developers can take to reproduce the issue

When I or another developer receives the page, we open the test plan document and record a brief summary of the issue on an issues worksheet. A tracking number is then assigned and written on the page, which is placed in a central location. Developers then each open up the test plan and take ownership of issues by putting their initials next to the issue. Once the issue is resolved, the entry is marked "complete" in the spreadsheet (and on the printout) and both are retained until the fix is validated during the next test cycle.

This "manual" method breaks down for mid-sized to large development teams. For these clients, I usually recommend a specialized software tool like TestTrack Pro from **www.seapine.com** (again, unfortunately no kickback). Unlike a generic product like Lotus Notes, specialized software usually offers much more useful functionality. For example, TestTrack includes a small program that you can put on your test PCs with which users can record their bug (with screenshots) in the "to-do" database with one or two clicks! No more manual entry.

Tracking Production Support Issues

Production support issues must usually be tracked with a bit more diligence. Specialized packages have been designed for call-center use and many automatically determine the appropriate support contact by application, degree of difficulty, time of day, and other criteria. Regardless of the package used, instruct your support desk to solicit problem information similar to that recorded during testing.

Conclusion

Your success with this or any methodology will inevitably depend on how completely and thoughtfully you apply it. Let the following principles guide you in this pursuit:

☑ **Solve the true problem.** When users speak negatively of a system, it is a symptom of one or more problems. Is the user interface needlessly complex or cryptic? Does the system allow the users to input data easily? Can users access the information they need quickly enough? Take the time to investigate by getting in their shoes.

☑ **Use the right hammer for the job.** Before implementing new technologies or packages, or developing new applications, take the time to understand the tradeoffs between various approaches. When there is a choice, favor the solution that is strategic in the long term (5+ years).

☑ **Focus on deliverables.** Keep your projects moving forward by focusing on the deliverables for each phase. By passing the output for each phase to the next in a clear, documented way, you avoid forgotten decisions, solidify project goals, set users' expectations, and tackle projects in manageable chunks, instead of all at once.

☑ **Build it right, the first time.** I cannot overemphasize the importance and cost implications of proper coding techniques. Develop your applications modularly, and use function modules, classes, subroutines, and includes to encapsulate reusable sections of code. Build components generically for reuse and develop BAPI-style function modules that encapsulate "reads" and "writes" from and to R/3 (e.g., BAPI_MATERIAL_GETLIST to read a material record). Avoid hard-coding and comment succinctly as you code. Use consistent naming conventions for all objects and variables. We developers learn each and every one of these principles in school, but frequently relax them in practice.

☑ **View teammates and end users as customers.** The vendor-customer concept is very powerful. Somehow, when end users and teammates are viewed as customers instead of coworkers, work products tend to be of better quality and are on time more often. Harness this phenomenon in your development projects. View the user of your work as a customer, even if the user is yourself.

By reconciling the points made in this article with your own experience, hopefully you can determine *if* your projects have succeeded and *why* they did or did not. For those of you who can conclude that you are participating in "healthy" projects, congratulations! For those on projects that are ailing badly, I hope this article will empower you and your management to revisit past (and current!) development efforts with a fresh eye and renewed vigor to set things right. Good luck!

Acknowledgements

I would like to especially thank Troy Bayne, Butch McNally, Richard Smith, and Ankur Tiku of Eastman Chemical Company for their ongoing support and participation in bringing this methodology to life.

Jonathan Pokress, CPIM, is an independent SAP e-commerce consultant and president of Charlotte-based Bluenote Consulting Group, Inc. (www.bluenoteonline.com). He recently spoke on the development methodology covered in this article and other topics at the "Summit for SAP Application Developers" in March 2002, and he was a key speaker at the "Web-Enabling Your SAP Systems" seminar series in 2001. Bluenote assists clients in the areas of R/3 e-business strategy, custom development, infrastructure, quality assurance, and mentoring. You can reach Jonathan at jpokress@bluenoteonline.com.

Appendix A: R/3 Custom Development Project Methodology Overview

Phase	Activities	Deliverables
1 | Analysis	• 1-2 process discovery meetings • Draft "as-is" (and "to-be") process documents as needed • Draft project analysis document • Hold 1-2 analysis walkthroughs • Take "before" snapshot and add to analysis document	• "As-is" and "to-be" process documents as needed • Project analysis document • Stakeholder approval of these documents
	Questions: • What are the symptoms being reported? What are the underlying causes? • What are the dimensions of users' needs (by role/division/country/person)? • Who else is experiencing these problems or needs (i.e., other possible end users)? • What business processes are involved and are they contributing to the problem? • What functional and technical gaps currently exist? • How are users currently working around these gaps? • Can a full or partial solution be achieved through process changes? • Does standard R/3 functionality exist to address the software needs? • Is there a packaged solution or R/3 enhancement in the works for this problem? • What technologies and approaches could be used in developing a custom solution? • Could any existing assets be reused in this custom solution to reduce its cost? • What are the costs, risks, and potential benefits of possible solutions? • Should the issue be left "open" due to concerns about ROI or risk, in anticipation of a packaged solution? • What are your team's recommendations (e.g., process changes, staff reorganizations, package implementation, development of custom software)? • What standards, methodologies, etc. will be used to control quality? • What are the specific goals of the project? • What are the metrics upon which success will be measured?	
2 | Design	• Design starter prototype (prototype document) • Conduct walkthroughs with stakeholders and refine the prototype • Author technical specification documents	• Prototype completed • Stakeholders' approval of prototype • Technical specifications developed

(continued on next page)

(continued from previous page)

Phase	Activities	Deliverables
3 | Construction	• Prepare sample data in development (DEV) system • Develop code: function modules, module pools, HTML templates, etc. • Request updates to security profiles and new test user IDs in DEV and test systems • Document assumptions and validate • Product walkthrough 1 with stakeholders • Implement stakeholder feedback • (Product walkthrough 2 with stakeholders) • Implement final stakeholder feedback • Prepare test plan identifying key scenarios • Execute test plan in DEV system and resolve issues*	• Test data prepared in DEV system • Code developed and unit-tested • Security profiles created/changed as necessary and test user IDs created in DEV and test systems • Test plan prepared • Stakeholder approval of product • Test plan executes “cleanly” in DEV system
4 | Testing	• Prepare test data in test system • Transport code and other objects to test system (e.g., Business HTML templates) • Execute test plan in test system and resolve issues* • Usability test and resolve issues • Performance test and resolve issues • Update design and technical specification documents	• Test data prepared in test system • Test plan executes cleanly in test system • Usability issues identified and resolved • Product stress-tested • Analysis and technical specifications up-to-date
5 | Deployment	• Register application with support desk • Transport code and other objects to production system • Link production user IDs to new profiles • Train users • Resolve issues found during training • Perform pilot rollout if desired • Roll out application for general use	• Support process defined and registered • Users trained and training documents distributed (e.g., via PDFs) • Application rolled out for pilot and/or general availability
6 | Support	• Resolve issues as they arise • Take “after” snapshot and assess project success • Establish plan to periodically monitor ROI	• Issues resolved in a timely manner • Project “success” analyzed • Plan for ROI monitoring in place
7 | Enhancements	• Collect and assess risks, costs, and potential benefits of enhancements	• Enhancement requests and ideas collected, considered, and scheduled

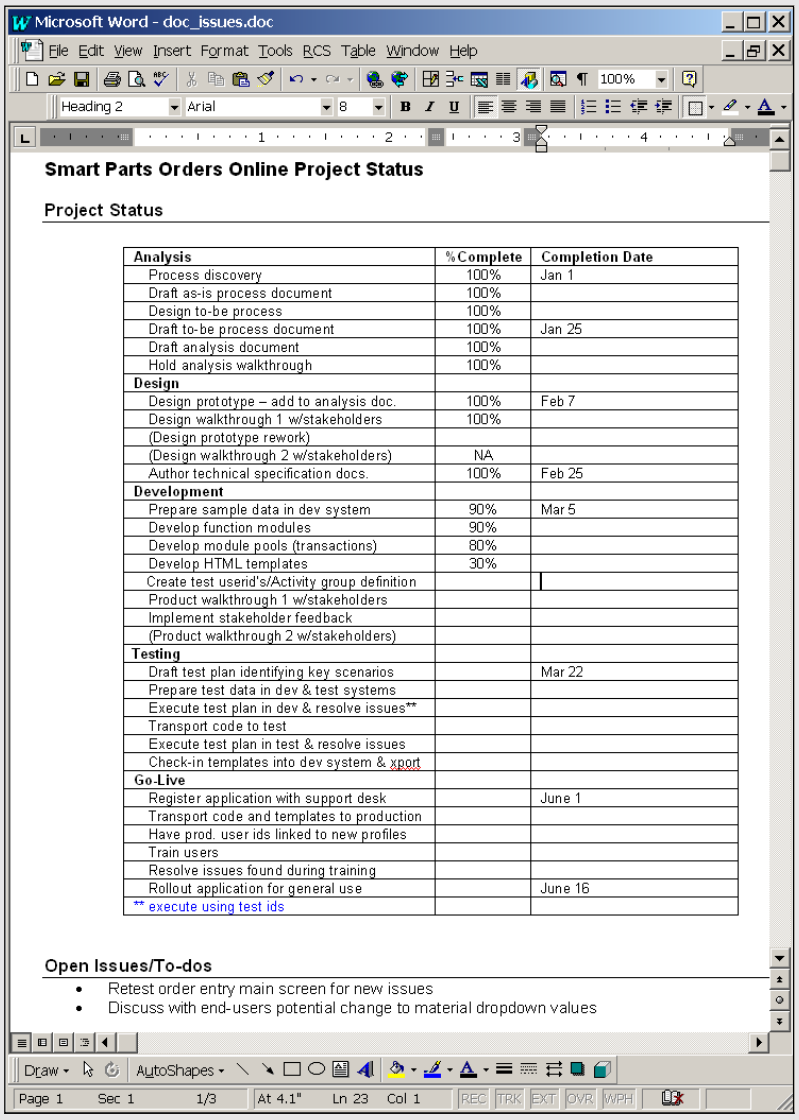
* Don’t forget to perform all “full” tests while logged on with test user IDs!

Appendix B: R/3 Custom Development Project Document Overview

1 | Issues Document

Tracks overall project status, to-dos, outstanding questions, and open and resolved issues. Can be quickly drafted in Microsoft Word (see the screenshot below). Contents include:

- Project status (percentage complete and target date next to each)
- Summary of resolved issues
- Open issues, questions, and to-dos (and who is assigned to each)



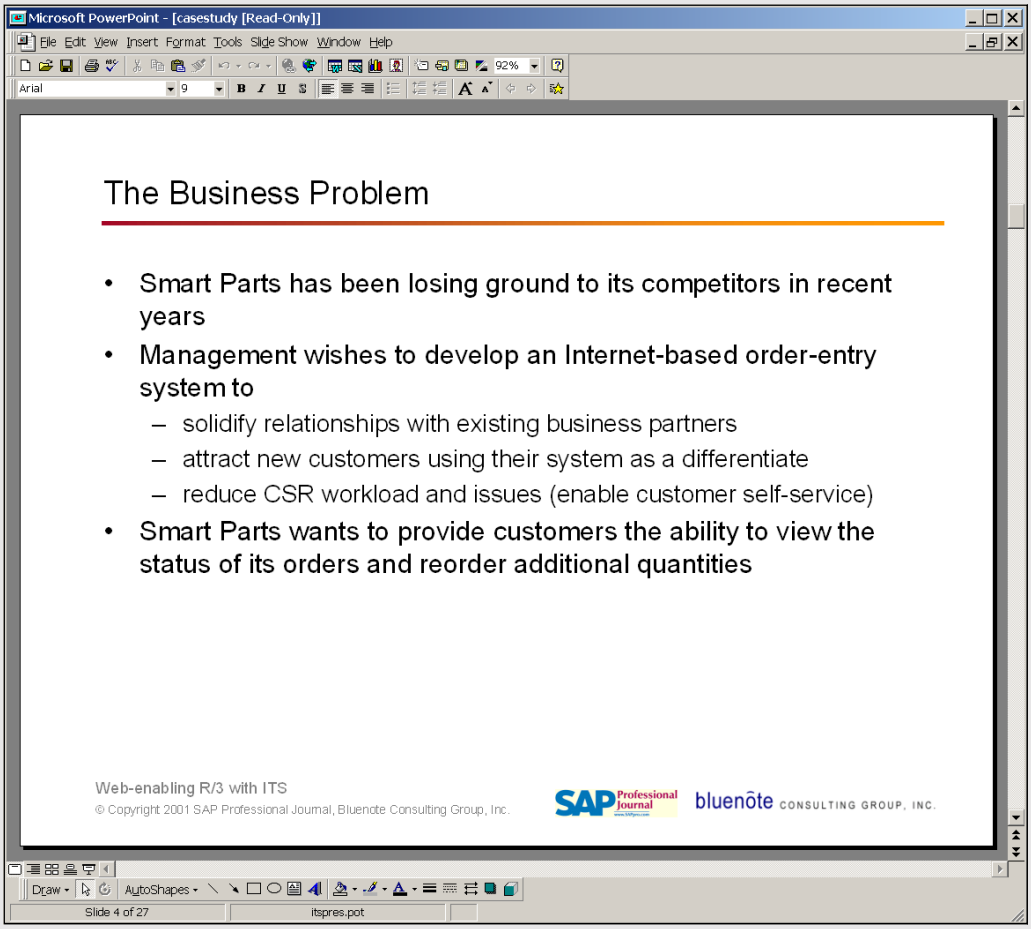
(continued on next page)

(continued from previous page)

2 | Analysis Document

Collects critical project information from a list of team members through project metrics. Drafting it in Microsoft PowerPoint (see the screenshot below) forces brevity, and information is instantly presentable during walkthroughs. Contents include:

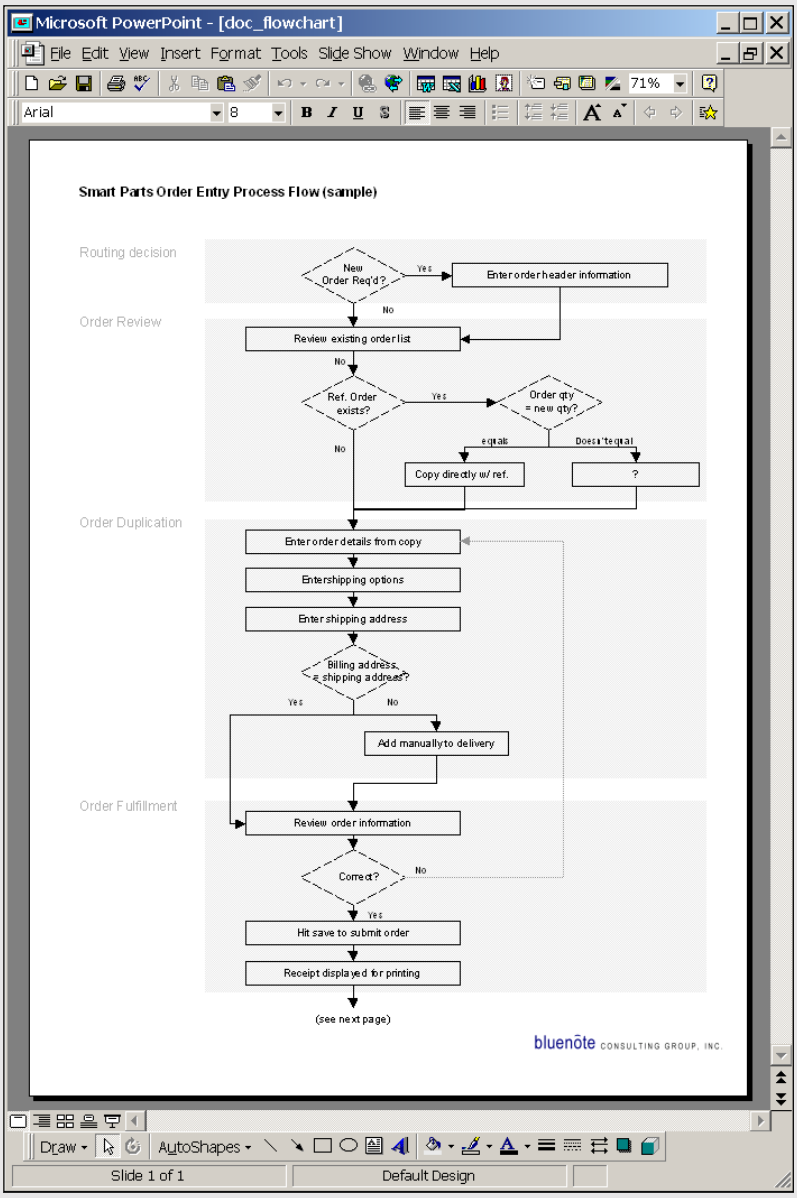
- Project name, location, date of initiation
- List team members with roles and phone numbers
- Project details such as background and sponsor
- Project goals
- End-user details: who, roles, facilities, languages, etc.
- Recommendations: process, packages, development, etc.
- Enumeration of software development needs
- Project metrics
- Relevant standards, procedures, quality control measures
- Projections: costs, benefits, risks, target ROI
- "Before" snapshot of project metric values
- Future opportunities for enhancement or deployment
- Observations



3 | "As-Is" and "To-Be" Process Flowcharts

Solidify understanding of business process and decision points, and are effective tools for negotiating and demonstrating process improvements. Consider drafting in PowerPoint for portability (see the screenshot below), but Visio can be used as well. Contents include:

- Step-by-step process steps
- Decision points



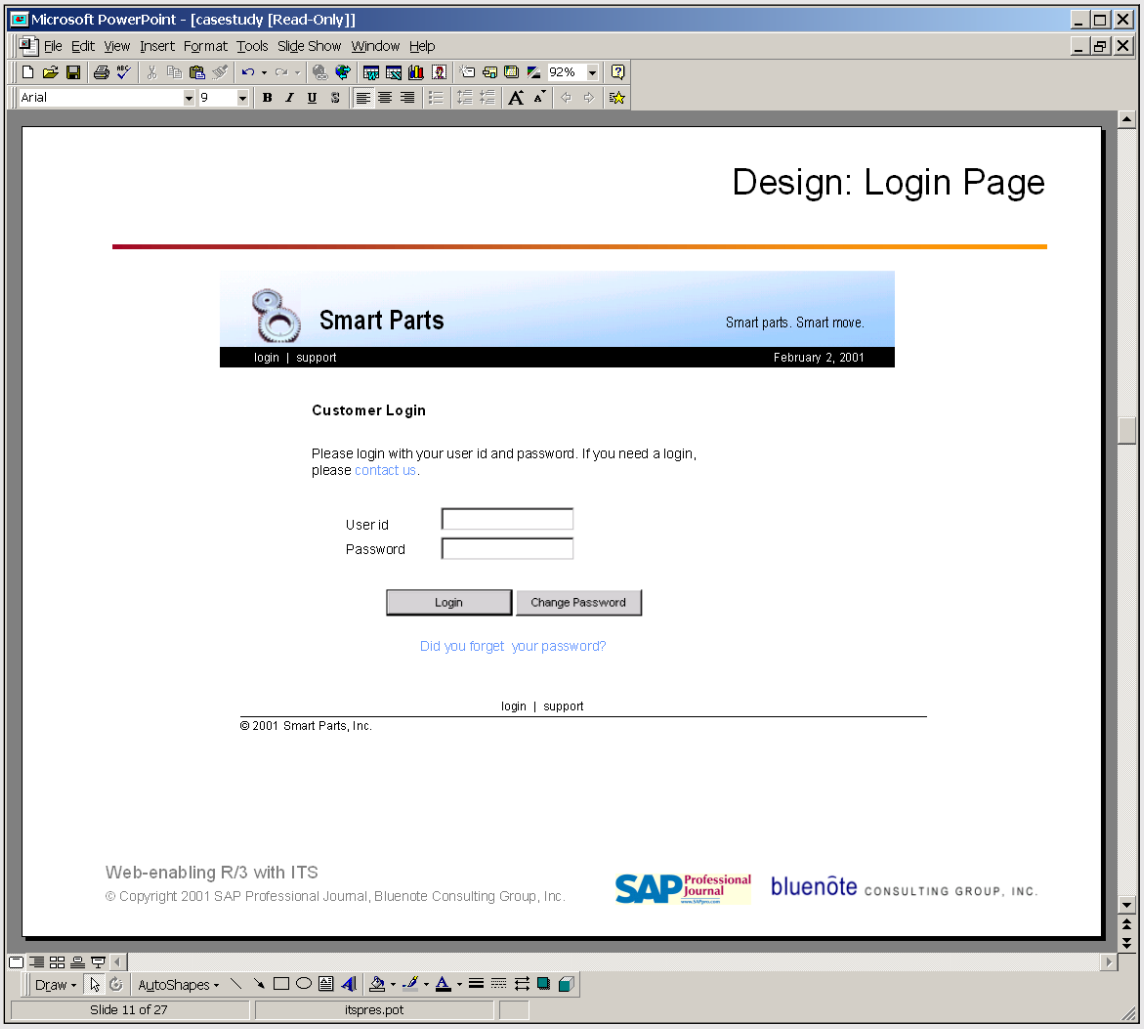
(continued on next page)

(continued from previous page)

4 | The Prototype

Essential to negotiating an application's design. Only by seeing the entire solution can users accurately communicate their needs. Can be easily drafted in PowerPoint (see the screenshot below). Contents include:

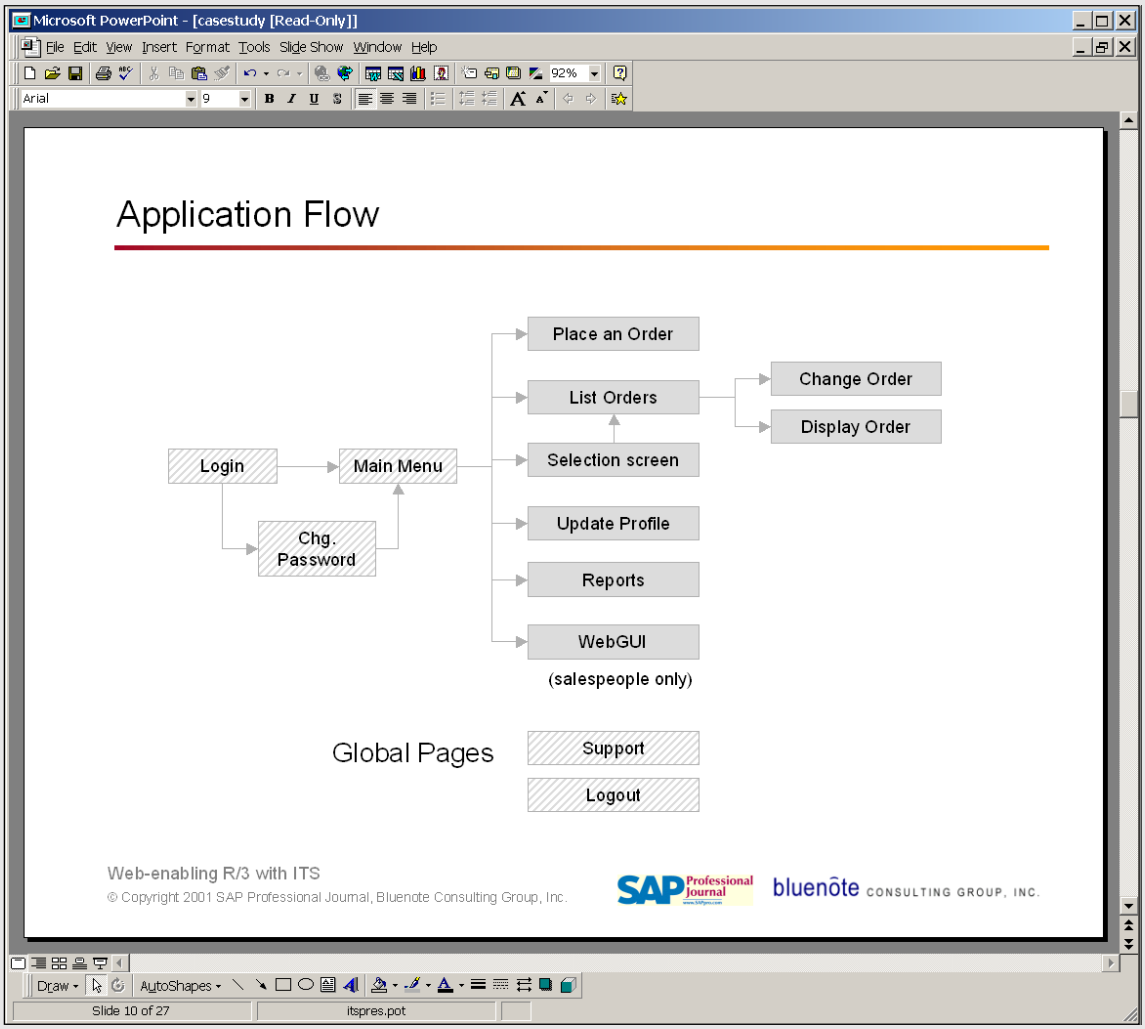
- Realistic screen-by-screen illustrations
- Notes about functional subtleties (e.g., list sorting)



5 | Technical Specifications

Not only document an application's design, but also serve as a contract between the designers and developers. Include enough detail for the specifications to be useful, but not so much that they become a chore to maintain. In particular, do not copy code into the specifications. Can be easily drafted in PowerPoint (see the screenshot below). Contents include:

- Application description, features, future direction
- How it gets invoked
- Application flow diagram
- List of components (e.g., programs, screens, transaction names)
- Module flow diagram
- Screen elements (e.g., fields, buttons)
- Processing (e.g., initialization, defaults, validation, events)



(continued on next page)

(continued from previous page)

6 | Test Plan

Identifies test steps to be performed while logged on with test user IDs. Scenarios can be as high-level as "multi-line sales order" and as technical as "bad field data," in which the tester is instructed to place characters in date fields, for example. Can be quickly assembled in Microsoft Excel (see the screenshot below). Contents include:

- Test steps per module, scenario, or role
- Test data
- Expected results
- Test problems and assignment (if tracking manually)