UML can save your next project

Unified Modeling Language does more than just put a pretty face on your projects-it's the lingua franca of a good development team

by Bill Shannon, Robin Yarow, and L.J. Johnson

A competent handyman can build a simple shed without a blueprint, just as your developers can write a program without a model. But a team building a house needs a blueprint to make sure everyone has the same understanding and can work efficiently, meet the client's needs and satisfy local code requirements. And certainly no construction company would even consider building a multi-story business building without extremely detailed blueprints.

In this article

The same reasoning applies if you're heading a team developing a modern enterprise application. Instead of simple mainframe screens displaying the results of a database query, you're probably building n-tier apps running on multiple machines with multiple operating systems, communicating via COM or CORBA, perhaps through a transaction server like MTS. The need for a common vision of an enterprise application is acute.

Tools based on Unified Modeling Language (UML) can help achieve that common vision of your team's applications, provided you can overcome any programmer or upper management reluctance to visual modeling. In the pages that follow we'll examine the justifications for using UML and take a close look at some of the UML tools that can give your next software development project a headstart toward success.

Making the case for UML

Beginning a software project has never been easy, but today's graphical interfaces have made it harder than ever. Because the starting point for many projects is ill defined, it's far too easy to substitute whiteboard screen representations for hard-core requirements analysis. If you're lucky, you'll have a formal list of requirements and a clearly articulated scope. Many times, however, the only requirements you're given are an existing application or just a sketchy idea in someone's head. You need a well-formulated plan to solidify the requirements and gain concurrence with your client.

Every project should start with a list of knowns and unknowns of your project. While compiling this list, consider all aspects of the project including current business processes, available technologies, and the target user community. This brainstorming effort should result in a substantial list of issues that require resolution. These issues form the basis of your initial interviewing sessions where the vision and scope of the project are agreed upon, and the process of identifying risks is begun. Depending on the situation, you may finalize the requirements as the first step, or you may need to start modeling the application and formalize the requirements later.

We've seen several projects fail because the first step of this discovery phase focuses on screen design. The language of Windows (text boxes, buttons, and drop downs) has become the universal language for systems design. Unfortunately, this language is not well suited to either illustrate business logic or discover what is most fundamental about the system you are designing. Instead, our ideas and designs are driven by the bells and whistles of user interfaces rather than by the business requirements. Invariably, the test phase of screen-driven projects quickly winds up in crisis mode when the user realizes that something crucial has been forgotten.

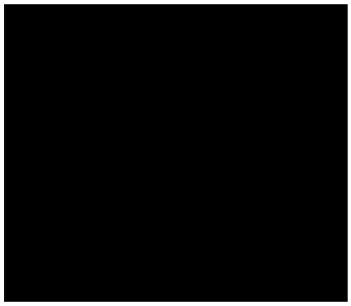
Compelling project statistics

- 56% of all bugs can be traced to errors made during the requirements stage
- 31% of all software projects are canceled before completed
- 53% of projects will cost 189% of estimates
- 9% of projects in large companies come in on time and on budget
- 16% of projects in small companies come in on time and on budget

—Standish Group survey of over 352 companies

Let's say you're building a house. You hire a builder, describe location, style (Southwestern Colonial Revival, maybe) and number of bedrooms, and the builder begins working. As the walls go up you discover that builder's concept of acceptable home construction and yours differ considerably. You wind up asking him to tear down walls, move lights, relocate plumbing and tack on rooms to suit your real needs. Cost overruns mount, project delays run into months and, not surprisingly, you don't want to live in the result. Your users probably won't want to live with software designed this way, either, but that's exactly how many Windows-based software systems evolve.

We once worked on a project to re-engineer a mainframe-based application to Windows NT. The existing application was antiquated, unwieldy, and user-unfriendly. At the first meeting, an all-out religious war erupted over the button style in the new system. The users were certain they needed buttons that read F1, F2, and so on, to mimic the existing system. These debates consumed so much time and energy that the business logic discussions (e.g. handling of batch reporting) couldn't begin for several weeks.



Use cases go where the rubber meets the road

Put down your markers

So now that we've convinced you to put the marker down and stop drawing screens, what should you do? Begin by visually modeling your application, not your screens. Lucky for you, the myriad object-oriented modeling techniques that emerged in the late 80s and early 90s have been reduced to a single language that has become the de facto standard: Unified Modeling Language. This represents the collaboration between the three masters of object-oriented theory: Grady Booch, Ivar Jacobson, and James Rumbaugh. The process begins with "use case" modeling.

Several tools on the market simplify the visual modeling process while delivering real value. Unlike the previous generation of CASE tools, which required you to fill in thousands of boxes and generated reams of useless documentation, these tools guide you through a process. Designed for a multi-developer team, they produce well-documented components and processes.

Products such as Rational Software Corporation's Rose begin with use case modeling, continue by creating sequence diagrams and class diagrams, and package your classes into components which then generate shell components into Visual Basic, C, or Java code. Riverton's HOW takes this another step. This tool will allow you to export your class diagram into a database design tool such as ERwin. HOW also provides additional non-standard UML diagramming techniques, similar to

storyboard diagrams. These diagrams are not only intuitive but enable you to graphically create queries, grids and other interface components that result in the generation of serious application code.

Preparation of use case models does not require artistic inclination; anyone can draw the stick figures, circles, and lines it requires. Use case modeling begins with identifying "actors", or people and systems that will interface with the application.

Far too many seemingly well-designed systems fail because the designers didn't understand the users. On a recent trip to a drive-up ATM window, for example, we noticed that the machine had Braille instructions. We couldn't help wondering if somewhere along the line the designers hadn't thought through exactly who is involved in a "drive-through" transaction.

Actors are involved with processes or tasks, known as "use cases." At this stage, the focus is on what value or tangible results the system provides to its actors. To keep the diagrams simple and straightforward, you may group the use cases into "packages," or logical groupings of related functionality. As a rule of thumb, if you have more than six or eight cases on a single diagram, you should create a package.

Developers who are new to UML often produce extremely detailed use case models that are not unlike pseudo-code. That's really getting too detailed too quickly, and when you notice the diagrams becoming technical, you should move on to another aspect of the system.

Test case by case

Use case models are not a new form of flow charts; the order of events should not be a concern. Test each use case by asking if that case on its own delivers something to the actor. Also, document within the use case the goal within the application context, what triggers the process to occur, exceptions to the rule, required performance, frequency of occurrence and any issues or questions that need to be revisited. We're lazy by nature, so we like to put in enough documentation so that later we can hand off our use case models to the testing team to form the basis of the system test plan.

Continue with use case modeling until all aspects of the system have been explored. This is the time to think high level, and to ask probing questions about the business processes. You may discuss issues that are outside the scope of the first release, but if you know where change is coming, you'll now be better able to build in flexibility.

You can name classes before you know they exist

Once you've identified what the system needs to produce, it is time to list specific steps of complex cases or group use cases into scenarios. These may be best captured in a Word document. In scenarios, you think about a series of events that typically occur in a sequence. Start by documenting the most common "happy day" scenario, where everything goes as planned. Then begin to explore branches that depart from the usual scenario.

Understanding usage scenarios will drive you to make the most common tasks easily accessible to the user. Think again about the ATM. The "fast cash" option didn't evolve until bankers realized that people frequently withdraw cash from checking accounts in standard amounts. Now, we just have one button to press, rather than about six. Users like a system that makes frequent tasks quick and easy.

Scenarios in place, you can proceed to the next step, creating sequence diagrams. Also referred to as "swim lanes diagrams," they visually represent a scenario. You display the flow of information, typically beginning with an actor, from some visual component like a login screen, to a business component like a security manager, to a data service like a query against the user table. Since you are designing for a multi-tier environment, remember that you need to separate the front-end interface from the back-end database by implementing a set of translators, or business components, between them. This separation will make for a system that is more adaptable to change and easier to maintain.

Once completed, sequence diagrams form the basis for class diagrams. This process is not unlike the more familiar process of designing a database. We're accustomed to coming up with a list of entities, identifying their attributes, walking through the rules of normalization, and thereby designing a database. Designing classes is very similar, although some classes will not be permanently stored in a database. In addition, you need to think about the functions and methods that the classes will perform. In the case of data classes, the methods may be implemented as stored procedures.

During this process, it will be difficult to resist drawing screens. You may want to consider doing prototypes for aspects of the system that are technical challenges or that have high usability risks. The important thing to remember about a prototype is that, in general, it should be done quickly and then thrown away.

The popular Rapid Application Development (RAD) and Joint Application Development (JAD) methodologies were great in that they ended the days of analysis-paralysis that resulted from waterfall methodologies. Instead of staying locked away for months doing analysis, we started coding right away and in front of the users. Many of these prototypes evolved into full-blown applications. But because they were built piecemeal, the code was not maintainable or flexible.

Requirements gathering, like most elements of project management, needs to be an iterative process. There's no beginning or end, really. You may prioritize features for later, but that's okay. Every time the users see it, right up through testing, they'll think of new ideas and new features. But more on change control later.

Plan for the worst

Managers embarking on a new software project must be the most optimistic group of people we've ever met. They rarely start a project by thinking about what could go wrong, but that's the surest means to guarantee that they will.

Expecting and planning for risks helps ensure a good outcome. A good analogy is that of a parent and a teenager. If you find out the teen has done poorly on a quiz or test, you can take steps to improve their likelihood of passing the class. If you don't know until the class has been failed, it's too late. You'll never receive a list of unknown problems up front, but if you force yourself to really think about the elements of risk, you may get closer than you think.

It is important to start keeping a list of carefully documented potential risks from the start of the project. First, define the risk. Next, describe the implication of the risk: What is the worst thing that will happen if the risk occurs? Give the risk a rating for how likely you think it is that it will occur, and use that rating to prioritize the management of all of your risks. Keep a status on the risk, so you'll be able to quickly see which risks have been avoided and which are still concerns. Most importantly, list specific actions that can be taken to mitigate the risk, and an owner for each action.

We often separate our risk list into internal and external concerns. Internal risks we know can be handled without alarming the user or client, although they might demand an additional hire or resource training in order to fill a skills gap on the team. External risks must be communicated to the

client. More important, we must solicit the client's support for executing the mitigation plan. You need to work in partnership with your client or user so that you're both working to identify and mitigate risks in a proactive fashion.

Risk management begins with thinking about everything on the project that could go wrong. Any technology that is new to you is a risk. Think about the environment you've been developing in, and what new challenges Windows NT is going to bring. If you're more familiar with the Unix or mainframe environment, you're going to need to understand a new architecture. You need to understand how memory is managed and how you can best structure your components for performance. You need to understand the security features of NT and how you can leverage these if security is needed in your application. If NT authentication is not sufficient, you will need to consider possibly using cookies or certificates. If these concepts are new, these represent risks which, if dealt with early on, can easily be avoided.

There is always a risk in what is most critical to the project. Most projects are driven by either a strict deadline, a tight budget, or the completion of a total set of deliverables. Knowing which of these is driving the process, and knowing that no project can maximize all three, will help you better manage the project. For example, knowing that cost is the most important aspect of a given project would lead you to postpone features that are not completely critical to a future release.

Anticipate Risk
A source of risk we frequently forget about is political or cultural risk. What is the political nature of the environment? How much of the client's time do you require? If you find that it takes two weeks to schedule the briefest of meetings with the client, consider how it's going to affect the project in the long run. State assumptions about how long you expect turnaround time on decisions to take. Ask for alternate resources that may be able to provide faster turnaround on minor issues.

And budget the amount of time you plan to spend in client meetings. One of our clients would commonly schedule three or four conference calls per week to discuss the project. As it was a large international company, there would commonly be 15-20 people in attendance at these meetings and the calls could take three or four hours apiece. We quickly found our actual project management time draining away, and were forced to do most of our work after regular business hours.

When you begin constructing the project plan, revisit your mitigation plans. Schedule time and accountable resources for mitigating risks. For critical ones, schedule this time early in the project. Hopefully, most risks will be dealt with early on. But you'll continue to identify risks throughout construction, in testing, and possibly upon release.

Building a successful multi-tier application on Windows NT requires the assembly and organization of a strong project team with the right skill sets. While many project managers minimize the importance of this task, it is quickly becoming a critical issue to tackle early in your development cycle. The justification for this early attention lies in the distributed, services-based nature of Windows NT.

Developers have enjoyed the benefit of an object-oriented approach in the Windows environment. Windows NT facilitates this approach by supporting the distributed component object model (DCOM). This architecture enables the development and deployment of discrete application components across several machines. These components then "communicate" with one another to provide a complete and seamless application.

The benefits of this architecture are enormous; several developers can each work on separate chunks of code, which are then deployed as specific components to comprise a complete

application. The notion of developers "tripping" over one another by working on the same source files is virtually eliminated.

This approach sounds great, right? Yes, with one caveat: It is imperative that the development team have a clearly defined methodology for communicating in this environment. Without a process to send, receive, and share information, your developers will find themselves working in isolation rather than in concert as a unified team.

Killing with kindness

We were brought in to manage a team that suffered greatly from these communication problems. The group was previously led by a manager who embraced the Microsoft Solutions Framework (MSF). Although it proved helpful in identifying the team, process, and service models, MSF failed to provide a strict methodology for coordinating team communications, and each team member was starved for information. They'd been assigned various components to develop, but no one was aware of the overall goal of the project, or its application architecture. The code that was generated often duplicated the efforts of another team member. To compensate, the developers sought information by spending more time attending meetings than they did writing code.

To avert this situation, a successful manager will establish communication pathways early in the project life cycle. By effectively providing the development team with all pertinent project information (vision/scope, design, milestones), the developers spend more time generating code and less time figuring out what they need to do.

The services-based nature of the Windows NT environment also creates a challenge for the inexperienced project manager. Today, many development efforts require the tight integration of several NT services (Exchange Server, SQL Server, Transaction Server, Message Queue) to form a complete application.

This integration of services results in project teams comprised of several team members; each member contributes to a specific, but limited, part of the overall project. Project management in this environment is overwhelming. Many project managers exhaust valuable budget resources identifying and meeting critical dependencies while ensuring that the application's distinct components are united in a single application.

One recent project's proposed application incorporated Site Server, Exchange Server, Internet Information Server, Message Queue, and custom Visual Basic components to provide a complete solution. When completely formed, the project team consisted of 22 developers. Each member specialized in a different facet of the development effort. Management of this project was an incredible challenge. Developers joined the team, completed their assigned tasks, and moved onto their next project.

Classes call for cases-and vice versa

This issue cannot be avoided. Windows NT applications will continue to employ a services-based architecture. The best course of action is to ensure that everyone on the team understands his or her contribution and is aware of the project's relative dependencies on that contribution. While this may be difficult with a large development team, it is imperative to ensure project success.

"Little change" means "big headache"

Throughout our consulting experience, we've been subjected to clients who use phrases like "minor enhancement," "little change," and "slight clarification" to xcommunicate a change of requirements

during the project's construction phase. Translated, these phrases all mean that your client has changed the requirements and wants you to accommodate those changes without impacting the schedule or budget.

All software development projects are susceptible to mid-project requirement changes, but the Windows environment tends to increase the probability of those change requests. This is mainly due to the graphical presentation of the operating system and those applications designed to run in that that environment. Hopefully, your clients understand that there is a multitude of interface options when developing a Windows application. If they don't, then it's to your advantage to educate them . . . quickly.

A client's pressing need to know what the application will look like in advance often forces consultants to use a RAD methodology. RAD focuses on a joint development approach in which the developers present iterative releases to the client for review and sign-off. Unfortunately, these review sessions rarely focus on anything but the user interface. Often, the business owner, end user, and other decision-makers wind up debating the presentation layer of your application, making frequent and usually inconsistent requests to change the interface. The impact to the project can be devastating.

Although changes to the user interface are rarely significant, making them again and again can greatly impact the budget and schedule. More importantly, accommodation of these changes tacitly expresses to your client that mid-project changes are easy to incorporate. Once this precedent has been set, the client will expect all changes to be implemented quickly and with minimal impact. That's virtually impossible with changes to the business services layer.

In one project, a client was not the actual business owner and, uncertain of the real owner's wishes, insisted on frequent and immediate changes to the application under construction. The inexperienced project manager believed that he was best serving the client by acquiescing to every requirement change. The result was disastrous: the project was over budget, beyond schedule, and failed to meet the client's expectations.

Implementing change order control is your only defense against this common problem. Change order control is the process of minimizing change requests and closely managing those changes that are critical to the project. Project managers handle change orders in a variety of ways. The best advice we can give is to establish change control early in the development process. In addition, make sure to clearly define your process and gain client acceptance before development begins. This will minimize those requests for trivial interface changes and assist in keeping your project on schedule.

As in all projects, your comprehension and definition of the requirements is a crucial, first step to solving the puzzle. It is then important to follow through with the continuous identification and mitigation of project risks. The volume and power of the available NT toolset requires us to select the right team, clearly delegate roles and responsibilities, and effectively manage communication.

Finally, expecting but managing change is the best way to avoid feature creep and ensure successful delivery. While these issues should be addressed in the earliest phase of your project, it is important that you continue focus on each throughout the life of the project. We call this managing the three P's of a project: the People, the Process, and the Product. Remembering to focus on these critical areas before and during your next Windows NT project will help to keep your project on time and within budget. Keep these tips handy, and good luck on your next project!

—Bill Shannon and Robin Yarow

A closer look at Rational Rose

On my first OO project, I was eager and excited, but (like many programmers) not fully convinced of the value of the newfangled modeling tool we were using. I spent weeks beating my head against the wall of understanding. But after a couple of months, magic happened. I could see the flow of the application. Moreover, I could talk with the other team members in a precise language with a minimum of misunderstanding. It was common to see two or more team members walking over to where they could talk and point at the class diagrams on the wall, since experience had proven that this was the easiest way to communicate their ideas.

I'm going to be using Rational Rose 98, applied to one particular language, Visual Basic 5.0, to demonstrate the value of UML. But these lessons are universal and many similar products can do the same sort of job. Many, including Rose 98, support multiple languages.

Finally, a common language

Winning over management can be as simple as mentioning how modeling tools can help you minimize the negative impact of employee turnover. I once left a project without the benefit of a visual model and had to turn over my part of the project to another programmer. The code in question was written to run as a server process, but it was relatively straightforward and exhibited good structured programming techniques (every procedure did just one thing, related procedures were grouped in their own file, low coupling, high cohesion, the works). When I was trying to explain the flow of the program, much of the time was spent saying things like "Well, you jump to this function in module XYZ, which calls two functions in the utility module." I tried diagramming the program structure on a yellow pad, but the lack of a common set of language constructs and notational symbols hindered our progress.

Yes, the other programmer finally "got it," but not without a fair amount of struggle and wasted motion.

So it's clear we need a common, very explicit language and a precise notation to describe our vision. But why UML? Well, you can find good reasons in a number of papers available at www.rational.com/uml/resources.html if you are interested in looking them up. But the primary reason is simple: UML is the both the practical and de facto standard. While Booch, Rumbaugh, and Jacobson (all now working at Rational) contributed their individual methodologies, many other OO gurus contributed major sections of the specification, and vendors converged on the standard. Support for UML is now part of corporate checklists and vendor support is widespread and deep. For example, Visual Modeler 1.1 is available free for Microsoft Visual Basic 4.0/5.0 (see "OOA&D for free"), and Visual Modeler 2.0 comes in the box with Visual Studio 6.0 with support for both VB and C++.

An important disclaimer is in order here. UML is only a notation, and tools like Rational Rose are only ways to automate using that notation. Neither UML or Rational Rose include a modeling process.

There are a large number of books and training CDs available on using various processes with UML, including Rational's own Objectory process (available as a Web-enabled interactive guide on CD directly from Rational at www.rational.com/products/o_process).

To software designers, redundancy isn't always a dirty word

Diagram every aspect

Class diagrams, along with the component and deployment diagrams, are the heart and soul of visual modeling. Class diagrams generate the code framework. These are the diagrams that the programmers and architects live with, argue about, and put many fingerprints on during the construction phase of the project. But if you don't go beyond class diagrams, you are missing out on some of the most valuable portions of the tool. Just as the construction phase is only one part of the software development project, class diagrams are only one part of the OOA&D process.

In fact, the first diagram you are likely to use is the use case diagram. A use case describes a series of transactions performed by the system in order to deliver a measurable result for a particular actor. Actors are not part of the system, but interact with the system in some way. The exact nature and number of use cases needed for a system is a hotly debated item (on the Rose newsgroups this angel-density/pinhead controversy is positively inflammatory), but their value is clear. Just don't take use cases to the extreme and fall into use-case paralysis.

Use cases do not generate code, and are generally not tied directly to the class structure. Instead, they document the behavior of the system from the actor's (which, in many cases, is the user's) point of view. In other words, use cases depict the behaviors the system will exhibit to the people actually using the system from their perspective. They are the dynamic models, while the class models are the static models. Use cases document how the system should respond without worrying about how we will implement that classes that handle those responses. Use cases are generally created during the analysis phase, expanded during the design phase, and then used as touchstones for completeness during the construction and testing phases.

UML's sequence diagrams are specific instances of a use case-the objects, classes, and interactions needed to fulfill the behavior described by the use case. Each use case normally has one primary sequence (or scenario) and many secondary scenarios. It is very easy to get carried away with this part of the project. Do not try to fill in all of the primary scenarios during the initial portion of analysis, and restrict secondary scenarios to only a few exceptional cases. Sequence diagrams can ease the communication between the analysts and the end users, and can be used directly by the architect and programmers to verify that the objects in the system are the objects needed by the system.

Collaboration diagrams provide an alternate way to present scenarios. Collaboration diagrams show how objects interact with each other, but not in a time-based order as with sequence diagrams. Since collaboration diagrams show less detail, they allow you to look at the system at a higher level of abstraction.

Discover classes

Discovering (notice that I did not say "creating") the classes needed by an application to fulfill the functional requirements of the system is hard, and well beyond the scope of this article. Why is it hard? Because it is intellectually difficult and totally dependent upon the reason for classification. Ask ten reasonably observant people how two children resemble their parents, and you will get a multitude of differing, but reasonable, answers. Some will compare facial features ("He looks more like his dad") while others will emphasize behavioral or personality features ("She's very outgoing,

like her mom"). Neither is "wrong," but the first answer might be more relevant if the problem is to determine physical characteristics based on genetics.

Another take on the difficulty comes from Booch, who estimates that 70% of classes are discovered almost immediately, 25% are found during design and implementation and 5% are not discovered until the maintenance phase.

UML helps with object discovery by presenting a standard nomen-clature (use cases, sequence diagrams and collaboration diagrams). You can also use other techniques such as CRC (Class-Responsibility-Collaboration) cards to discover the objects needed by the system. Yet the task remains difficult, and definitely nonlinear. You'll find that the most talented programmers are not necessarily the best at discovering classes. When you find a programmer/analyst who is good at the discovery process, make her responsible for the integrity of those classes that she helped find-she won't let the architecture be bastardized.

Rose 98 is a big help during the initial class discovery phase as well as later in the life cycle when the design begins to drift. Rose 98 automates the documenting of classes as you initially discover them as well as helping you bring revisions to the code back into the design loop.

State diagrams are the other major diagrams included with Rose 98. They are needed only for the classes within a system exhibiting very dynamic behavior and therefore in need of careful analysis for possible errors. Real-time systems immediately come to mind, but classes in ordinary enterprise applications may also need this specialized analysis.

Flexible means efficient

You don't have to use all the diagrams available with Rose 98. In fact, one of greatest dangers for neophyte users of a tool like Rational Rose is its size, flexibility and richness-it can be overwhelming at times. You don't need to use every single construct available via UML on your project! A good article on the tradeoffs of using the various constructs available is "UML Applied: Nine Tips to Incorporating UML into Your Project" by Doug Rosenberg (available at www.sd-magazine.com/uml).

Each diagram presents a unique view of the system you are creating, and using multiple diagram types (views) allows you to survey a rich and synergistic overview of the application that you are creating. Going back to the architectural analogy, a large office building goes beyond the basic set of blueprints (in the apdev case, the class diagrams) to include many views of the structure, from all angles, together with scale-size physical models, cut-aways, detail views, computer walk-throughs, room decoration proposals, and so on. Each diagram shows the system from a different perspective.

For those of us in the trenches, these various diagrams allow all members of the team to communicate the shared vision more effectively and with more precision. The use cases are useful not only in the original analysis, but in certifying that the final product actually accomplishes the goals of the system. The testers can use them when they construct their test plans. Sequence diagrams can be used by the programming team to communicate with the customer as well as with each other, since the 1-2-3 ordering of the program flow is generally familiar to the customer. The class diagrams then force the programmers to focus on the interfaces of the classes, not the internal implementations.

Rose 98 itself isn't wedded to one particular language, but to be able to effectively use the tool, you must understand your target language, its capabilities and idiosyncrasies. The choice of Visual Basic, Java, C++, or some other language will effect some of the design decisions you make while designing the classes themselves. For example, if you design a system that is heavily dependent upon traditional inheritance and then decide to implement it in Visual Basic, you will end up doing a

fair amount of redesign. If you knew the language choice was VB, you would probably use delegation instead of inheritance. You might even work around the problem Rose 98 seems to have in handling VB's Type/End Type structure.

All this translates directly to less wasted time, a more productive team, and faster development. And that, my dear Watson, is how good, and occasionally even great, software is produced. Producing software is ultimately a people issue, not a technological issue. Tools like Rose 98 are indeed worth the pain of acquisition and training because they help teams of people overcome the technology issues.

Bill Shannon is a Director at Rubicon Technologies, Inc. where he heads one of the firm's analysis and design teams. Bill is a leader in business process implementation, object-oriented design, and project management. He may be reached at bshannon@rubicontechnologies.com.

Robin Yarow is the Project Practices Manager at Spectrum Technology Group where she sets policy and performs internal training on UML, engagement management and project management practices. She may be reached at ryarow@spectrumtech.com.

L.J. Johnson is the owner of Slightly Tilted Software in Dallas, Texas. He has been programming in Visual Basic since 1.0 and is a section leader on The Development Exchange at www.devx.com. He is also the Ask The NT Pro at www.inquiry.com. Reach him by e-mail at ljjohnsn@flash.net and visit his Web site at www.flash.net/~ljjohnsn.