Becoming More Predictable

Reprinted from *The Smalltalk Report*
By: Rebecca J. Wirfs-Brock

Smalltalk provides an excellent platform for incremental development. Prototypes can be built to clarify poorly understood requirements. Design can complement prototyping efforts to produce production quality code. The benefits of incremental, iterative software development are enormous. Large, complex systems have a good chance of meeting customer requirements when they ship. Concepts can be validated before major resources are committed and schedules finalized. Functionality can be routinely added to an existing application base without things grinding to a halt. Object-oriented designs and programming languages are a real aid to incremental, iterative development. The modularity, flexibility and encapsulation that objects provide makes incremental development practical.

Managing an incremental development can be extremely challenging. This is particularly true when the development team is new to both object technology and an incremental development process. I know of no magic formula that guarantees success. But life can be a lot easier if some checks and balances are applied. It is possible to design (and re-design) in an iterative, incrementally developed object-oriented application. Predictability (and taking action to become more predictable over time) is the key to making it work.

Life in the Fast Lane

Iterative, incremental development projects are typically started for several valid reasons:

1. Requirements for parts of the software are unclear. The plan is to develop prototype code, get feedback, assess what needs to be done, and then do it.
2. New hardware or complex processes need validation. Given long lead times, it may be necessary to write code that exercises new system functionality long before a detailed design is finished.
3. In even moderately complex applications, it is difficult to complete all subsystems at the same time. In fact, planning for a single massive integration phase is risky business. Confidence can often be gained if the system is brought alive in planned phases. With phased development, temporary functionality needs to be provided to make the things work. These interim parts are replaced over time (if things go according to plan) with well-designed, production quality code.

These are sound reasons. There are also situations where projects slide into an iterative whirlpool due to lack of leadership and planning, neglect, or inexperience. On danger to avoid

(regardless of how you embarked on an iterative development) is wandering for too long without making significant progress, jeopardizing the entire project. Another risk to avoid is continual backtracking to fix things up when premature decisions didn't pan out.

Odds can be improved by committing to and sticking with a process that encourages open communications. People need to consider the consequences of their actions or inaction. Even if you have to tinker and adjust the process over the course of the project, it's easier to put the appropriate force fields in place early rather than inject change into an organization that has been lumbering along for a while. The objective is to encourage communication and thoughtful behavior, making the entire team function more smoothly.

Some Typical Scenarios

In the remainder of this article I want to discuss several tasks that are part of most incremental developments, and offer advice on how to do them effectively. Although the tasks undertaken by most object-oriented development teams are roughly equivalent, outcomes vary widely. What's startling to consider, however, is just how big an influence *anticipating* and *preparing* for change can have on the final outcome. It's crucial to think things through before reacting, and to provide enough information to allow others that same opportunity. I've learned some strategies for improving the outcome through direct personal experience. Observing the habits of people and organizations that meet or exceed their objectives has been another rich source of inspiration.

Making Progress. Knowing precisely what's left to do and how long it will take is difficult to ascertain in a phased development. It's important to be as open and honest as possible when assessing status. Trust and teamwork play a big part. On one project, we lived and breathed the creed of incremental, iterative development. Not every project team will be so dynamic, have such exciting chemistry, or be so committed to the project. But I learned a lot from that project that I've found extremely useful on many other occasions.

We recognized our plans were estimates. They needed to be living, changing documents. We knew we couldn't do it otherwise. Team members didn't feel persecuted when they were behind on their initial estimates. If someone honestly didn't know where they were, the last thing they did was hide the fact. We described our designs and implementations as either being throw away, experimental, temporary, works well enough, or of finished quality. We had guidelines for communicating project status that stated both our progress and confidence in our decisions and achievements.

When things didn't go as planned, we brainstormed about what it might take to get back on track. We set a new date for achieving measurable results, generated a list of action items, and kept moving. We acknowledged our situation, and actively sought help and advice from others when necessary. We were team players. It was important to do assigned tasks, but making the team succeed was the primary objective. We freely debated the impacts of change (and our progress) not only with the design and development team, but with marketing, manufacturing and project management.

Management made it very clear that it was OK to say, "I don't know." You were expect to take action to find answers, but you didn't have to bear the burden alone. The team helped develop alternatives. And most importantly, a messenger of unexpected news was *never* punished.

Deciding what to Prototype. Determining what you want to prototype before starting is important. Sloppiness and unpredictability on the part of some prototyping efforts has gained prototyping an undeservedly bad reputation. If you have a clear set of objectives and a plan of attack it will be much easier to get management buy-in. It also improves your chances for developing a meaningful prototype. It's vital to explore options before committing to any serious prototyping. What's serious prototyping? Spending more than a couple of weeks.

Take enough time to collect your thoughts and set objectives. One way to ensure that you've done enough homework is to write things down. If you have trouble summarizing objectives, you aren't ready to launch into prototyping. You need to be able to state both what you hope to accomplish and how you intend to do so. Summarize any burning issues or questions. Communicate what you know and what you want to find out.

To clarify objectives, discuss the prototype with people whose perspectives differ from your own. It is fair to state a preferred course of action, but be willing to listen to other ideas. It's best if you can bounce ideas off someone who is both receptive to your ideas and a good critic. Brainstorm alternatives. Listen carefully to others' comments and criticisms. Don't shoot down new ideas. Mull things over.

Do a paper design of the parts of the parts you think you understand. This preliminary design probably won't be worked out in much detail. But be prepared to discuss key objects and architectural strategies. Talk to those who are reviewing your ideas in their own language. If your audience understands objects, talk about objects. If they don't, you can either spend time educating them, or speak to them in their own terms. I have had several meaningful discussions with people with an electronic engineering background. I found it useful when presenting my prototyping ideas to draw analogies with phased hardware development.

Setting objectives doesn't require a lot of time. What's appropriate obviously depends on the duration of the prototype. It's perfectly reasonable to spend a few days setting goals before a month long prototyping effort. A six month effort might require a few weeks to set clear objectives. Without such forethought, it's difficult to know if you are even working on the right problem.

Building a Successful Prototype. The most effective prototyping effort I have seen was pulled off by a team that felt certain that they'd require several prototyping cycles before finalizing their design. Knowing that, they planned for incremental, iterative success. I wouldn't characterize their efforts as random prototyping. They didn't just build something and keep tinkering with it until they had a final product. They documented what they expected to

accomplish (and what issues they wouldn't address). They made sure others bought into their concept of prototyping. They made it clear to management that they wanted to design and implement prototypes in order to gain understanding. They didn't hope or expect a final result would 'pop out' if they were lucky. They set milestones, and measured results along the way. They spent as much time assessing their prototypes as they did building them. They actively solicited advice and expertise when they felt uncertain. They spent a lot of time analyzing whether their design would be flexible enough and whether it would scale to accommodate future system requirements.

They went through several prototyping cycles on a major part of a large, complex system without ever being on the critical path. Were these people more brilliant or harder working than their teammates? They were skilled and had over ten years of experience. But they weren't the only bright stars on the team. Instead, I'd characterize them as being in the habit of thinking before doing. They also believed very strongly in working smarter, not harder. They weren't constantly programming. Coding was a by-product of designing and reasoning about the problem, and not its only manifestation, even during prototyping.

Solidifying Subsystem Interfaces. In incremental development, the objective is to accommodate change without leaving interfaces too ill-defined or soft and squishy. Working out interfaces is naturally an iterative process. Expect to refine them to match both class users and class developers' needs. They key is to agree upon initial interfaces, and agree to *renegotiate* change. Changes should be made in context of their impacts on the overall system. Although responsibility for making change ultimately rests with the developers, system concerns need to be injected into the process of deciding what (and how) to change.

As subsystem designers work out the details of how services provided by their subsystem will be supported, they collect ideas for changes. Others using their subsystem will also find room for improvement. Evolving an interface requires teamwork. One way to foster teamwork is to publish *proposed* changes rather than to notify others after the fact. If no one responds to the proposed changes, don't assume they agree by default. Things go more smoothly if people are given a chance to understand and comment on proposed changes. Make sure people have enough time to absorb the impacts of a proposed change. What may seem minor to subsystem implementors can cause major repercussions elsewhere. Expect debate on alternatives before making any major change.

Smalltalk team programming environments make it easier to propose changes. Developers can pass back and forth workable alternatives without affecting others. They aren't a substitute for thinking things through. Effectively evolving interfaces requires adopting and promoting 'systems think'. On larger projects, one way to promote system thinking is to designate a system architect. The architect's initial role is to determine the initial structure and organization of the application, including subsystem interfaces. Throughout the project, the architect keeps on top of proposed changes, and actively works to mediate needs of subsystem developers and users of those subsystems.

Avoid the two ends of the spectrum: standardizing too early, or being so flexible that nothing can ever be agreed upon. Both extremes cause problems. If an interface is frozen, other parts of the application contort to fit. It isn't always appropriate that the first one 'done' defines what is expected of others. On the other hand, if nothing is ever agreed upon, people are constantly in a reactionary mode, consuming lots of time adjusting and readjusting to shifting interfaces.

An Alternative to Just Living with the Consequences

There are many factors that go into a project's success. There's no substitute for persistence, intelligence and commitment. Iterative and incremental development require extra attention to planning, designing and careful consideration of consequences. Individuals can make a difference by planning for change, rather than just letting it happen. Things won't work well if individuals go off and 'do their own thing' ignoring the rest of the project. Successful iteration is fostered by teamwork and a willingness to accept and solicit constructive criticism. Improving predictability is a constant, ongoing process.