Beyond the Hype:
Do Patterns and Frameworks Reduce Discovery Costs?

Panel Session

Moderator: Steven Fraser

Panelists: Kent Beck, Grady Booch, Jim Coplien, Ralph Johnson and Bill Opdyke

Abstract: Patterns and frameworks are two approaches to the development of both new and evolving software systems. An implicit hypothesis is that "discovery costs" are reduced by leveraging knowledge previously collected, analyzed, organized, and packaged. "Discovery costs" (or "getting started" costs) include both the costs of understanding the problem to be solved and the cost of understanding the tools, methods, existing software, etc. For large, multi-year development projects in industries such as defense or telecommunications, discovery costs can dominate the overall cost (and risk) of software development.

This panel will share its experience and perspectives with the audience with a discussion initiated by the following questions:

- Have patterns and frameworks really delivered on their claims for reducing discovery costs? Can current best-practices be characterized as meaningful or marginal (what are the measures)?
- What are the discovery cost factors where frameworks and patterns appear to deliver the biggest bang for the buck?
- While mature pattern languages and frameworks may reduce the learning curve, they do not eliminate it. How much of a learning curve is required to develop a sufficient shared context with the authors of a set of patterns or a framework?

Steven Fraser is manager of the Software Design Process Engineering team at Nortel's Lab in Santa Clara, California. Previous to this he spent four years at Bell-Northern Research's Computing Research Laboratory in Ottawa, Canada. In 1994 he was a Visiting Scientist at the Software Engineering Institute (SEI) collaborating with the Application of Software Models project on the development of team-based domain analysis techniques. Since joining BNR in 1987, Fraser has contributed to the ObjecTime project, an OO-based CASE-Design Tool and to the BNR BCS software development process. Fraser completed his doctoral studies at McGill University in Electrical Engineering. He holds a Master's degree from Queen's University at Kingston in applied Physics and a Bachelor's degree from McGill University in Physics and Computer Science. He is an avid operatunist and photographer.

Kent Beck

First let me restate the question to be explored by the panel: "Does experience help?" The answer is, of course, "Duh" (translation for those who didn't attend American schools, "Obviously, you idiot").

Whenever the answer is obvious, either everyone else is stupid or you are asking the wrong question. Let"s assume the latter. Two aspects complicate the question. First is the focus on patterns and frameworks as the media of expression for experience. Second is the focus on "discovery costs", as opposed to all the other costs of software development.

Our flippant answer to the original question would no longer hold if patterns and frameworks don't communicate experience, or communicate it in such a way that it costs more to learn a pattern or framework than to have and reflect on the experience in the first place. So, do patterns and frameworks really communicate experience? As a consumer of both patterns and frameworks my experience is that they certainly do. When I come into a new situation and I find patterns or frameworks in place, I begin making progress much more quickly than when I have to just start by asking a bunch of questions about a sea of code. As a writer, I have seen developers become productive much more quickly within the structure of existing patterns and frameworks.

The second challenge to the obvious answer comes when you limit the discussion to "discovery costs". Does experience help with discovery costs? Sure. "Hey, this looks like another good fit for Gemstone," is a statement that if correct is worth hundreds of thousands or millions of dollars. Can the kind of encoding of experience represented by patterns or frameworks help with discovery costs?

Here's an example of where I have used a framework to reduce discovery costs — I was working with a framework for payroll processing. We were interviewing the customer. She mentioned a new requirement. I asked the developers, "What more is this than just another station?" "We can't see anything," they replied. We explained the visible ramifications of the decision to the customer. She agreed. This was a kind of discussion I have seen take weeks. By casting it in terms of a framework proven to solve similar problems, it took all of ten minutes.

Here's an example of patterns reducing discovery costs. I have learned a handful of patterns, most from Prof. Johnson, regarding the design of counting systems, those concerned with managing transactions and accounts. Now when I talk to a client about a new systems, I can quickly discover the complexity of the architecture by asking which patterns apply. For instance, I know the Posting Date pattern, so I know to ask if the client needs to print reports as of a certain date.

For me and for the teams I coach, frameworks and, to a larger degree, patterns do reduce discovery costs. They also reduce costs throughout the software lifecycle. Most importantly, though, by reducing nasty surprises and stress, they make developing more fun.

Kent Beck is a guitarist and folk singer, father of four, and programmer. He is the author of a whole bunch of articles on software development, some of which he is still proud of, and the book "The Smalltalk Best Practice Patterns". He is currently exploring the humanistic discipline of Extreme Software while employed at CSLife in Zurich, Switzerland.

Kent Beck has "pursued" patterns for almost 10 years and is a founder and director of the Hillside Group. In 1991, Kent founded his company First Class software which produces applications and development tools in Smalltalk and provides consulting services. From 1989 to 1992 Kent was the programming tools project manager at MasPar Computer for a massively parallel supercomputer written in C++ and Smalltalk. From 1987 to 1989, while Apple Computer, Kent worked with a group under Alan Kay designing, implementing, and testing Playground, an object language for children. Prior to this Kent worked at Tektronix TekLabs to research and develop an advanced programming environment in Smalltalk. Kent holds a BS and MS in Computer Science from the University of Oregon.

Grady Booch

There's considerable anecdotal evidence and some empirical evidence that points to the benefits of patterns and frameworks in reducing discovery costs. From a methodological perspective, their use has two places in the software development life cycle: as a factor in avoiding new development via the reuse of preexisting components, and as a risk mitigator in the sense of providing components that are already proven in practice. In the first case, theoretically, this should reduce discovery costs; and in the second cast, this should reduce software scrap and rework. However, software development is not a frictionless environment, so it's reasonable to ask the question: what are costs associated with patterns and frameworks? The short answer is that there are some costs, and depending upon the culture of the development organization, these costs can be prohibitive. In high ceremony organizations, the use of patterns becomes so bureaucratic that any opportunity for benefit is crushed. In low ceremony organizations, the use of patterns becomes so informal that their impact often never reaches beyond the individual developer. In the optimal case, the use of patterns are institutionalized, but in a manner that balances the need for control and the need for creativity.

Grady Booch has been with Rational Software Corporation as chief scientist since shortly after its founding in 1980. He was instrumental in developing two major concepts: an iterative model of software development and the importance of software architecture. He developed Ada-language reusable components and the Booch C++ source-code components, which helped to popularize software reuse and make it economically feasible. Grady's methods and ideas are being used to develop complex and demanding software systems, ranging from air traffic-control systems to commercial aircraft avionics to financial trading systems to telecommunication switching networks to defense systems. Grady has served as an architectural mentor for a number of large software projects and was also one of the original developers of the Rational Environment, the company's original software-engineering environment, and its compiler technology. Grady was the original architect for the Rational Rose object-oriented analysis and design tool and has used a variety of object-oriented and object-based languages, including Ada, C++, and Smalltalk.

Grady is the author of five books, his fifth and latest book, "Best of Booch: Designing Strategies for Object Technology", was released in October 1996. Grady has also published more than 100 technical articles on object- oriented technology and software engineering. Grady has consulted and lectured on these topics throughout the world. Grady is a Distinguished Graduate of the United States Air Force Academy, where he received his B.S. in computer science in 1977. He received an M.S.E.E. in computer engineering from the University of California at Santa Barbara in 1979. Grady is a member of the American Association for the Advancement of Science, the Association for Computing Machinery, the Institute of Electrical and Electronic Engineers, Computer Professionals for Social Responsibility, the Association for Software Design, and the Airlie Software Council. Grady is also an ACM Fellow.

Jim Coplien

Too many projects look for the "home run" in reusable platforms and frameworks. Frameworks work well only if they can predict well: to predict what will change, and what will not. This is a difficult enough problem for individual objects or modules, let alone for extensible application skeletons. Small frameworks like MVC work, but few large frameworks enjoy success. The main problems are rooted in the socioeconomic systems that tie together the vendor and customers of a framework. In common practice, each customer approaches the vendor for customizations. Even internal implementation customizations, or addition of new sites of parameterization, can't be kept invisible from other customers. It's nice in theory, which lights up the eyes of the academics who are my fellow pattern members. It looks good on paper, which delights the project management and system engineers on the panel who don't write code. But practical experience with CORBA, pSOS, and dozens of supposedly standard internal frameworks have just not borne out the theoretical success of this lucrative idea. To the contrary, they have increased overhead and development costs in the names of reuse, standards, and NIH.

Frameworks work to the extent they are small, or to the extent that their sites of parameterization can all be dealt with at run time. Patterns help solve this by engaging human intellect. Coding the result of a transmogrified pattern is easy. Give me the code to start with, and it just gets in my way.

Jim Coplien is a member of the Software Production Research Department in Bell Laboratories. He holds a BS in Electrical and Computer Engineering, and an MS in Computer Science, both from the University of Wisconsin at Madison. His early career work includes applied research in software development environments, version and configuration management models, and in object-oriented design and programming.

He is currently studying organization communication patterns to help guide process evolution. This research has already created a generative pattern language that has successfully been used for business process engineering in corporations worldwide. His other research areas include multi-paradigm design and architectural patterns of telecommunication software.

He is author of "C++ Programming Styles and Idioms," the foremost high-end C++ book in the industry, and co-editor of two volumes of "Pattern Languages of Program Design." He writes a patterns column for the C++ Report. He sits on the

board of the Hillside Generative Patterns Group, a small consortium of industry leaders developing pattern languages as an adjunct to object-oriented approaches. He was program chair of ACM OOPSLA'96.

Ralph Johnson

Asking whether patterns and frameworks reduce discovery costs is like asking whether someone who knows something about billing is going to have an easier time making a billing system than someone who doesn't. Of course! The problem is whether we have the right patterns and frameworks to reduce discovery costs. If not, how can we get them?

Most of the current patterns are design patterns, and design patterns might help you reverse-engineer a system, but they will not help you analyze a problem and learn the application domain. I think this is what you mean by "discovery cost". The kind of patterns we'll need for that are like those in Martin Fowler's book. We don't yet have the right patterns for decreasing discovery costs.

Frameworks are similar. Most of the available frameworks are for technical problems like user interface or persistence. These won't help reduce discovery costs very much, except that they make it easier to build prototypes, which is a popular discovery technique. But domain-specific frameworks can have an enormous impact on discovery costs.

I've been working with a company that has built a telecommunications billing framework. They have developed a method for analyzing telecommunications companies' billing systems in terms of their frameworks. They then build a customized billing application. The can analyze a customer, design a billing application, implement and install it, in about three months. The first two months are analysis and design, the implementation and installation take only a month.

As you might expect, one of their problems is teaching their framework to new employees. We are uncovering the patterns that they use, documenting them, and including them in a training program so that new employees can learn them faster.

Ralph Johnson is on the faculty at the University of Illinois. He is a co-author of "Design Patterns" and has been to every OOPSLA.

Bill Opdyke

For much of the past several years I have been researching, consulting and involved in product evolution activities related to frameworks. Part of my job has involved selling the benefits of frameworks to product groups.

Frankly, I feel that developers experienced in *using* frameworks are in a better position than many of us consultants, researchers and academics in assessing the real discovery cost implications. So, I asked several of them to share their experiences with me. Our discussions focused on discovery costs and frameworks, but not upon patterns. Personally, I feel that patterns are important in reducing discovery costs, but I'll leave it to my fellow panelists to discuss that connection.

Discovery costs are significant for evolving systems. For all but the first release, the system itself is a major (perhaps *the* major) constraint on the design. Discovery costs are greatest for new staff — but are still significant for other, more experienced staff.

People new to a project need to gain an understanding of the paradigm, application domain, language, library of available components and the development environment. They discover how a system works by reading documentation, dabbling, interacting with other staff and building something (preferably a simplified example). Mature frameworks help ease this discovery process.

Developers who have worked with a mature (or nearly mature) framework felt that such a framework reduced their overall development times in half, reduced their discovery costs by more than half and in some cases provides as much as 90% of the functionality that they needed. The greatest benefit of mature frameworks came in the design phases. Framework documentation helped reduce discovery costs; the paradlgm, domain abstractions and other design information embodied in a mature framework provided a shared language that facilitated communication among team members. The object-oriented framework implementation technology supported rapid prototyping and iterative development, facilitating discovery through building simplified examples.

Unfortunately, *mature* frameworks seem to be the exception rather than the norm. In industry, frameworks are often viewed as the *means* while products are the (revenue generating) *ends*. Framework maturity takes time and involves churn; both can be disruptive to application/product development. Tools and process issues can also inhibit the development of a mature framework.

Domain-specific frameworks, to maintain their funding/survival, are often "married" to an early application/development project. These applications are motivated to make expedient specializations and other framework changes that compromise the generality of the framework. Similarly, unless there are mechanisms for sustaining *domain-independent* pieces outside of a particular application, new applications may end up re-developing these pieces. Discipline and executive commitment are need to mature a framework and to maintain framework identity in the midst of these influences.

In conclusion, mature frameworks can and have made a difference in reducing discovery costs and reducing development intervals. The path to a mature framework can be rocky and requires expert navigators. As with software reuse and software developments in general, the success of a framework is often more tied to organizational and cultural issues than to the technical concerns.

Many thanks to Don Brown, Ian Bruce, Joe Davison, Larry Mayka, Warren Montgomery, Ralph Straubs and Tom Williams for contributing their insights.

Bill Opdyke is currently involved in advanced network services architecture and platform planning projects at Bell Labs—Lucent Technologies. He consults with platform and application development projects on topics such as reusable assets and distributed object computing. Bill previously focused on intelligent network and decision support applications of object-oriented technology. Bill's doctoral research at the University of Illinois focused on refactoring C++-based object-oriented application frameworks.