# ♦ Introducing Software Reuse Technology

*Robert Lied, Lynn P. Pautler, and Patrick E. Helmers*

*Lucent Technologies' business needs for software development mandate rapid production based on careful engineering. One key method of achieving this goal is software reuse. Since 1995, the Software Engineering and Technology Group has supported a strategy of domain-specific engineering, with the underlying technology of application generators. From our successes and failures, we have learned that, to succeed, reuse—and new technology in general—needs, at the very least, grass-roots adopters, top-down management support, enabling technology, and, most importantly, integration of the new methods into the mainstream. This paper uses our experience with domain engineering to illustrate how to introduce new technology. We describe the motivations for development based on domains, our process for adopting technology, how we make domain engineering mainstream, and how the lessons we learned can be applied to help others introduce new technology.*

## Introduction

All software businesses are concerned with how their software is engineered. How can it be produced most rapidly? How can it be maintained through variations over time and differences across related products? How can businesses ensure that each software issue will be constructed with the care that yields correct behavior, good performance, and reliability? And, once a product is established, how can they help it continue to make competitive leaps forward, replacing itself, in whole or in part, with state of the art technology?

For many years, at least part of the solution to this overall problem has been known: find software that already works well and recombine its components to create new variations and even entirely new products. Supplant the components from time to time as new technology makes cheaper, better, and faster components possible. Tie the components together in an architecture—itself a kind of component that can be supplanted with improved versions.

We in the Software Engineering Technology Group have also come to this conclusion, and for the past several years have had the opportunity to bring this solution into our organization. Bell Labs' research in software engineering provided a path for creating reusable components, proof by example that it could be done, and ongoing consultation and research. Our task was to introduce software reuse technology into our organization.

This project has made us richer in at least two ways. First, we understand the processes needed on both sides of the reuse coin: how to create reusable components (software for reuse) and how to use components (software with reuse). Second, we comprehend the personal and organizational reactions that introducing technology induces and the strategies and methods required to incubate new technologies. This paper describes these two points in detail.

## Reuse in Industry

Software reuse has been studied extensively in some sectors of the software industry, in the U.S. notably as a result of the Software Technology for Adaptable, Reliable Systems (STARS) effort and its descendants,[2] and in Europe as part of the Software Evolution and Reuse Consortium and its related projects.[3] In analyzing these efforts and some resulting experience reports, [4-6] we examined the two most reliable ways to make reuse work: one is to distill the lowest common denominator components into libraries, and another is to focus on particular problem domains and solve them once and for all.

The difference can be visualized, as shown in **Figure 1**, by considering a typical program as a control structure (a graph) with computation at the nodes. Traditionally, reuse focuses on what the nodes already contain, trying to build the most general functions at the lowest common denominator. The alternative is to reuse the tree and customize its nodes.

This approach is related to the idea of patterns and frameworks in the object-oriented world. Gamma et al.[7] describe the idea this way:

> Reuse on this level leads to an inversion of control between the application and the software on which it's based. When you use ... a conventional subroutine library ..., you write the main body of the application and call the code you want to reuse. When you use a framework, you reuse the main body and write the code it calls.

This is easily said, but in reality how can we implement the framework, or the tree, in a customizable way? And what if object-oriented techniques are not an option? Application-oriented languages offer a solution. If we confine ourselves to the vocabulary of a specific domain, it becomes reasonable—even attractive—to express the domain in a computer language. The terms of the language are the vocabulary of concepts in the domain, and the expressions, parameters, and statements of the language express how the domain can be customized in various ways. (A *domain* is a set of software that has enough in common to consider constructing the whole set from a common core and a statement of the variations,[1] as shown in **Figure 2**. We use the words "domain" and "family" interchangeably.) The compiler for the language embodies the framework for the domain.

This approach was first sketched more than ten years ago, when application generators became practical. [8,9] Griss showed how the idea of building software "kits" leads to a model of software reuse that replaces the library metaphor with a manufacturing metaphor and brings reuse into the realm of business decisions.[10] Work at the Software Productivity Consortium[11] led to a coherent process, Synthesis,[12] for attacking the analysis and implementation of domains.

Our approach to reuse descends from this lineage. Within our development organization, we became acquainted with an application generator for hardware/software interfaces that automated the production of several artifacts used in diagnostic software. Developers on this project had successfully created a Smalltalk environment that generates documentation and source code. When asked about the approach they took, this team likened discovering reusable assets to family-based design.

**Panel 1. Abbreviations, Acronyms, and Terms**

| | |
|---|---|
| application engineering | a process for constructing applications based on a model and a specification language for the application |
| application engineering environment | a toolset for application engineering, with emphasis on generating systems |
| application generator | a specification-driven tool that generates artifacts of the application, especially the actual software of the application |
| commonality | an aspect of a family that is invariant in all family members |
| domain engineering | analysis of a family, or domain, and construction of an application engineering environment |
| FAST | Family-oriented Abstraction, Specification, and Translation |
| STARS | Software Technology for Adaptable, Reliable Systems |
| variability | an aspect of a family that may be optional or may take on different values in different family members |

## Family-Based Design

Families are most easily identified from our past experience. For example, after a few years of computing, most people are familiar with at least three text editors. It soon becomes apparent that all text editors share the same basic functionality: they allow the user to add, select, delete, search, and display text, and to use a handful of other features. They also vary in a few

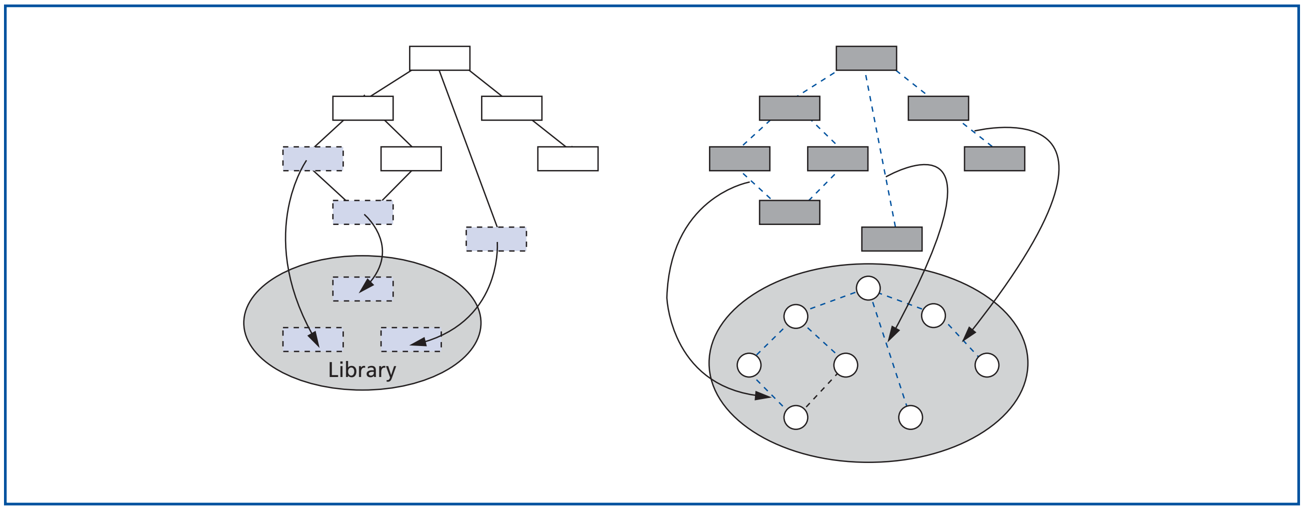

***Figure 1.***
***The library approach versus the domain approach.***

predictable ways: moded versus modeless, keyboard command set versus mouse and windows, and the level to which features such as searching engender baroque options. Clearly, however, every text editor has a core.

Other examples from consumer-level software are easy to name: e-mail readers, calendar schedulers, directory and file managers, and personal finance programs. Families also occur in more arcane specialties, such as missile guidance systems, real-time microcontrollers, telephone call billing, and protocol stacks.

Not surprisingly (to us, anyway), families appear in droves in telecommunication software, although often in arcane specialties. Lucent Technologies has more than 80 areas of switching software that we have identified as probable families or domains, and we believe that, in reality, twice that many exist.

In applying family-based design, we have two ways of thinking about the software that implements domains, as shown in **Figure 3**. The first, or intuitive, view (presented in Figure 3a) is one in which a set of applications is produced, with each application targeted for a customer or platform. In this view, we consider the set of variations in the domain to be fixed and imagine a set of applications that covers these variations.

Another approach is to consider just one application, but to think about the complete set of things it must do, as shown in Figure 3b. We could hold the application constant and generate within it the complete set of outputs in the domain. The first approach works well across product lines and platforms; the second works well within a product line. Either will give us a strategy for planning reuse.

Even within an application, we might constantly be generating new variations on the output. The totality of the outputs can represent a family. For example, in the telephone business, each call has an associated billing record that lists, among other things, the call's duration, origination and termination, and perhaps other attributes such as a third party to whom the cost should be charged. The format and content of these records vary somewhat, depending on the telephone company and local standards; they vary significantly in different countries. We can consider the family, or domain, to be the set of possible billing records.

## FAST as a Reengineering Framework

We knew that taking a family-oriented approach to software design, with application generators as an underlying technology, was a good way to proceed. But what process would enable us to develop software this way? The Software Productivity Research Department of Bell Labs provided us with a solution, a descendant of Synthesis called Family-oriented Abstraction, Specification, and Translation (FAST).[1]

FAST incorporates a number of elements, including:

- A development model based on separating

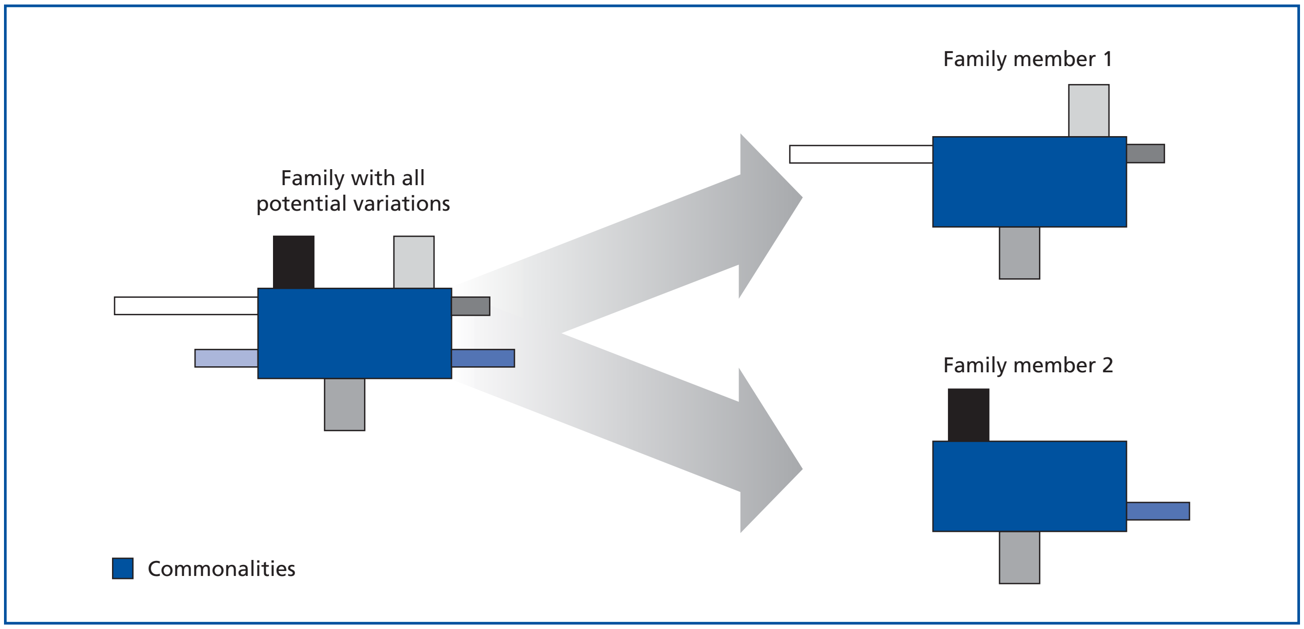

***Figure 2.***
***Software families, with a common abstract core and a known set of variations.***

domain engineering from application engineering,

- A succinct method of capturing the commonalities and variabilities of the family members in a domain,
- A method leading to the expression of commonalities and variabilities in the form of an application-oriented language, and
- A life-cycle process model for maintenance and evolution of a domain.

The FAST production environment makes it easy to specify and produce family members, as shown in **Figure 4**. This application engineering environment satisfies new customer requirements by providing rapid production with careful engineering. We envision the application engineering environment as a set of configuration files or as one of the installation "wizards" that are increasingly common in personal computer software. Instead of using it just once to install some software, however, the application engineer uses the application engineering environment repeatedly to specify different family members for the domain.

One domain of switching in which an application engineering environment has been developed is that of maintaining equipment status. Equipment is related in various ways, with some units functioning as parents of others, some as spares, and some as fully redundant duplex units. These relationships govern how equipment can change state. If we need to replace a parent circuit, first all children of that parent must be out of service; if a unit that has spares goes out of service, one spare must go into service.

Each time we develop a new piece of equipment or improve an existing one, a developer must specify all the relationships among the equipment and then implement software that carries out state changes within the constraints of these relationships. The developers responsible for this work used the FAST process to develop an application engineering environment in which they could specify relationships in a graphical notation and then generate a program that sequenced equipment states correctly.

Of course, to reap the benefits of using an application engineering environment, first someone must build it. The FAST process calls this domain engineering. FAST's systematic procedure for doing this is probably its greatest strength. In the past, the equivalent of application engineering environments appeared from time to time, but they were always the inspirational acts of highly motivated individuals. The prospect of

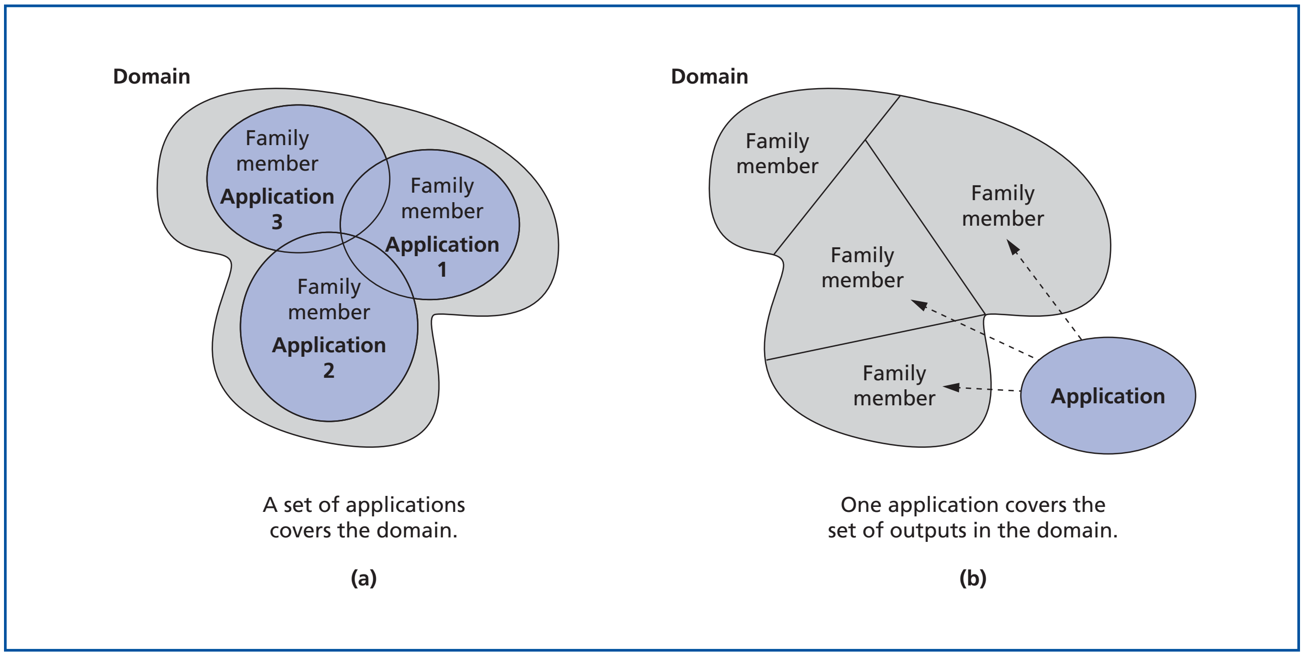
Domain
Family member Application 3
Family member Application 1
Family member Application 2
A set of applications covers the domain.
(a)
Domain
Family member
Family member
Family member
Family member
Application
One application covers the set of outputs in the domain.
(b)

*Figure 3.*
***Ways of thinking of a family, or domain, as (a) a set of applications, or (b) one application with a set of outputs.***

being able to systematically automate the production of family members in a domain—any domain—is exciting news for anyone concerned with productivity and intervals.

## Analysis and Automation Experiences

The domain engineering process begins with an analysis phase called *commonality analysis*, during which a handful of domain experts hold a moderated, facilitated discussion to specify the family members, what they all share in common, and how the variations can be quantified. The results of this discussion are recorded in real time. The analysis is documented, usually in plain prose, and divided into a dictionary of terms and three separate lists—commonalities, variabilities, and quantifications of the variabilities.

We have found commonality analysis to be a particularly fertile technique. The two unique aspects of the analysis process—real-time recording of a moderated discussion among experts and the commonality/variability viewpoint—are very effective for illuminating a domain. Invariably, even the participants already recognized as experts find the dissection of commonalities and variabilities to be enlightening. As a result, we have seen commonality analyses used in unexpected ways. Some groups have used this type of analysis as a problem-solving method in its own right. Others have used it as the basis for training new employees; and some, to spearhead a reengineering program to recombine divergent software versions.

Most, however, continue in their quest for domain automation and use the commonality analysis as the basis for designing the application engineering environment. The environment consists of two major parts: the tools that generate the software, and the documentation of a process for using the tools. In brief, the commonality analysis leads to an application generator by using:

- Terms in the dictionary as the basis for key words in the language,
- Commonalities as the basis for the architecture of the application generator, and
- Variabilities as the basis for expressions and relationships in the language.

An application engineering environment can also be implemented by using the commonality analysis as the basis for frameworks and classes in an object-oriented analysis and design.

This is, of course, a rather glib description of the process, because language design is still a bit of an art.

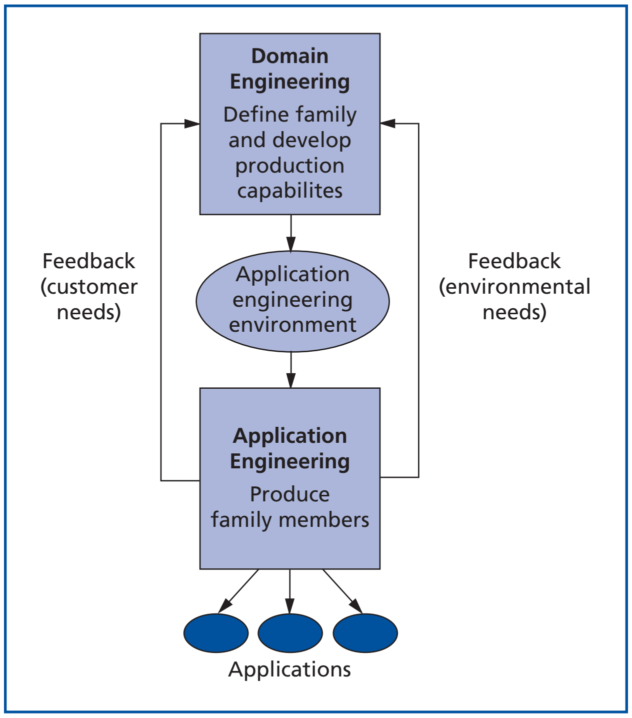
Domain Engineering
Define family and develop production capabilites
Feedback (customer needs)
Application engineering environment
Feedback (environmental needs)
Application Engineering
Produce family members
Applications

*Figure 4.*
*A FAST process.*

Even with the help of a commonality analysis, designing a syntax and building a parser is difficult; adding such niceties as a debugger or validator is daunting indeed. In this area, however, we are encouraged by the work on jargons and the infocentric paradigm by Lloyd Nakatani of Bell Labs.[13] Some of our domains use his InfoWiz information processing system as an application generator.

### Switch Domain Engineering Projects to Date

We have direct experience with many teams that are applying domain analysis or domain implementation or using the equivalent of application engineering environments. Several of them started as collaborations with the Bell Labs Research area. Our involvement ranges from educating, facilitating, and consulting to budget management and even implementing some modules. We have experienced several outstanding successes, a few utter failures, and on the whole, a generally positive result. Projects use domain engineering to build application engineering environments in about the same time it would take to implement a single application. The resulting application engineering environments can reduce production of future applications by a factor of three to one. We have received positive feedback from experienced developers, who find it gratifying that their knowledge is critical to domain engineering. Novices have also reported learning more quickly using commonality analysis and the application engineering environment than they would have by studying the internal intricacies of the domain.

We are confident that we are well on our way to creating reusable software assets. We can now tell our developers how to create reusable software, and our project planners and managers how to incorporate reuse into their development strategies. Along the way, we learned not only about the reuse technology itself, but how to get people to try it. The developers were the easiest to win over, but we also had to persuade the people who run organizations and projects. What started for us as a journey to create systematic reuse has become an adventure in technology transfer.

## The Problem of Technology Transfer

The task of technology transfer is to match technical solutions to business problems and to establish the practice of the technical solution. Often, the inability to introduce a new technology successfully stems from insufficient attention to the process of adopting technology transfer. Thus, even when a trial looks promising, widespread use may never develop. Whether technology transfer succeeds is, for the most part, driven by people issues. Technology implies change, and change challenges the day-to-day activities of each employee. A human being's natural inclination is to fear change and become entrenched in the status quo. However, we believe the opportunity for professional growth and the promise of finer products and services far outweigh the risks.

### The Adoption Model

**Figure 5** shows the technology transfer adoption model that we use to promote domain engineering within our development organization. This model evolved from our experiences with structured methods, our collaborations with Bell Labs Research, the industry's experience with object-oriented methods, and our instincts to create an adoption model for technology. We have also used this

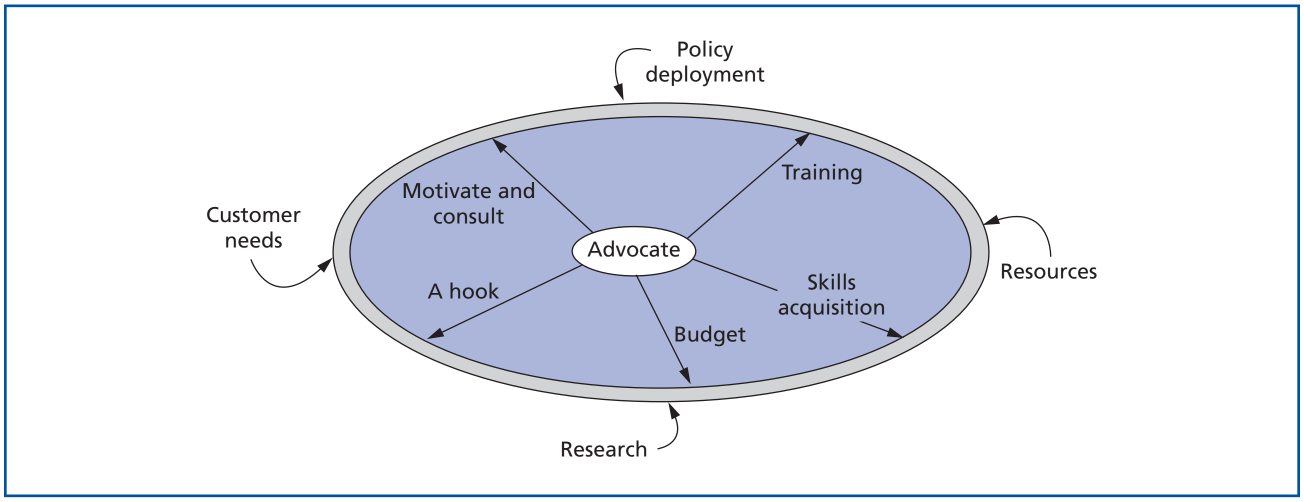

***Figure 5.***
***How technology transfer forces act on an organization.***

model to facilitate other projects within our development organization.

We view technology transfer as an organizational problem. As such, we must bring to bear organizational forces that will push the technology into use. A set of things—such as business requirements, management's statements of policy, advances in technology, and the resources to work with—can be imposed from the outside. Along with these, the new technology must quickly grab users' attention. This "hook," as we call it, needs to be relatively inexpensive and easy to apply; in addition, it must provide real benefits quickly. The hook in the FAST process was the commonality analysis. It could be completed in a few staff weeks and was invariably seen as a valuable exercise in itself. Focusing on this part of the total process brought quick gratification that directed future effort and drew development teams into the full process.

The hook gets individuals and small teams to risk trials. To go beyond that to sustained use and adoption, an organization must establish advocacy within its own boundaries and grant the advocate real power in the ways that matter—time and money. The advocate must be first to use the new technology and to embark on training and consultation. The advocate also has budgetary discretion, to help startup projects plant their seeds and to maintain their own viability as trainers and consultants. Much of the technology transfer process is conducted by the advocacy team.

Once the organizational pieces, motivations, and technologies are available, the development team needs the following fundamental aspects to guide them through technology transfer:

- A vision,
- A champion for the cause,
- The right problem domain,
- A motivated team,
- Expertise in domain and technology,
- Adequate education,
- A reasonable formal schedule, and
- Tools of the technology.

Although this looks like a laundry list for success, the successful adoption of a technology depends on addressing each item. These issues do not resolve themselves; ignoring any of them invites failure. The rest of this section describes each item and the role it plays in the process of technology transfer.

**A vision.** A vision generated by the team has the greatest chance of being successfully adopted. Generally considered a difficult hurdle, this process requires introspection, which in turn depends on an unbiased perspective best attained in a facilitated meeting. The first task is to have the team create a list of wants. Because this list often lacks focus, it is then useful to work from a needs perspective and enumerate complaints that may hamper efficiency, productivity, and satisfaction. Once generated, these complaints—windows to areas of possible improvement—are ranked in order of importance, from most to least critical. Each attribute on the list is inverted to create a companion list of wants. The inverted list contains, in order of preference, a set of attributes that will be used to frame a vision in the team's own words. For example, **Table I** shows a list of complaints juxtaposed with a list of wants.

The list of wants is then used to craft a vision that describes the goals and objectives of the project and to guide technological choices.

**Champion the cause.** Earlier, we implied that large organizations are full of inertia and work hard to ensure the status quo. To offset this force, a champion is needed to rally the troops and guide them toward the vision. The role of champion is best played by a manager, but well-respected engineers have successfully championed some domain engineering projects. Because the champion is ultimately responsible for the project's success, it is best if he or she is the domain owner. Knowledge about the technology will serve the champion well, but organizational savvy is even more important. The champion must have people skills and the ability to generate excitement and trust. He or she must be committed to the cause, able to express the vision, and also able to continually guide the team around organizational barriers.

**The right problem—cost/benefit analysis.** Technology transfer is only successful if the real business concerns of a domain are addressed. Because of its investment nature, technology transfer is best limited to areas where the business will obtain the most return on its investment. These areas often have high maintenance costs and many upcoming extensions, or they are about to introduce a new product with clear market demand. It helps if the new work comprises a large new chunk of code or if the existing software has clear interfaces.

Market need must be balanced against the cost of deploying a new technology. Reasonable estimates can only be made when the organizational structure is clearly understood. Details regarding organizational analysis will be discussed later, but for now we stress that a cost/benefit analysis must be performed early to ensure that the team is working on the right problem. Finally, the team must possess a true willingness to reengineer aging design and a clear ownership of the domain.

**Table I. Complaints versus wants.**

| Complaints | Wants |
|---|---|
| Data population difficult | Data population easy |
| Logic tightly coupled | Separation of concerns |
| Overtime necessary | Little overtime |
| Testing found bugs hard to fix | No bugs |
| Aging interface | Graphical user interface |
| Architectural rules broken | Architecture honored |

**Teamwork.** Teamwork occurs when individuals bond together to pool their unique and complementary talents for some greater cause. A team cannot be formed if the individuals do not value the project and assume the vision as their own. Teams with little more than hope and courage have developed remarkable technological systems. It is important, however, for all members to speak the same language. Common notations, graphs, and terms are necessary for clear communication. Ownership of the completed product is also important, for it is the only way team members will truly be accountable for a project's success or failure.

A motivated team is not the only factor needed to ensure a successful domain engineering effort, however. The team must also be populated with the right mix of people. Complex telephone systems cannot be created by amateurs. The domain knowledge needed to create these products resides in our engineers. Thus, successful technology transfer depends on having domain experts on the team.

We also recognize that new paradigms are difficult to accept. Occasionally, teams may need personal tutoring regarding the nuances of a new technology. In these situations, consultants should be used to validate architectures, review designs, audit estimates, and facilitate brainstorming. Someone with an unbiased view has a better chance of seeing past organizational denial and distinguishing destructive behavior. We use in-house consultants whenever possible and draw on the industry when necessary.

**Education.** We have found that few engineers have the tenacity to learn a technology through self-study. Our organization has developed classes in object-oriented analysis, object-oriented design, commonality analysis, and technology transfer. The classroom creates a space for engineers to consider and assimilate new technologies and processes. We have found that just-in-time training is rarely successful for paradigm shifts. Continued exposure to a new technology is the best way to ensure complete assimilation.

Education is not the place to cut costs. Engineers must be allowed to purchase texts and periodicals, and to attend conferences. It is folly to send the best engineer to class and have that person return to mentor the others. Often the best engineer lacks the skills or time to "core dump" all that he or she has heard and experienced. Teams communicate better when they are exposed to common modeling languages such as class, responsibility, collaborator (CRC) cards, entity relationship diagrams, finite state machine modeling, use cases, object-oriented programming, a visual programming environment, domain engineering, commonality analysis, design patterns, and the fundamentals of computer networking.

**Schedule.** The successful adoption of new technology requires a reasonable and formal schedule. Reengineering and technology transfer work best when they are not challenged by a tight market window. When schedules are tight, the first offering should be treated as a prototype (using some older technology); in the next release, the prototype should be reengineered using the new technology. When time for investment is available, a formal schedule with externally visible benchmarks should be published every 2 to 4 weeks; artifacts such as documents, prototypes, and presentations should appear every 8 to 10 weeks to show progress and maintain momentum. Schedules should be strictly adhered to, and if dates are missed, the integrity of the plans should be reevaluated. Uncertainty occurs when an unfamiliar technology is being used, and engineers tend to be perfectionists, fidgeting with molehills while ignoring the mountains. Schedules ensure continued progress toward the vision. If a schedule is not working, a new one should be created. But the importance of a schedule must be stressed, because it measures one of the fundamental metrics of success, time.

**Tools.** A foundation environment must be created to support the technologies to be explored. Possible supplemental tools to be deployed on the computing environment include tools to create graphs, such as computer-aided software engineering; computer-aided design; word processing; visual programming environments; object-oriented compilers; class libraries; database managers; change control systems; real-time operating systems; networking software; development kits for graphical user interfaces; World Wide Web access; and a current set of industry catalogs. Tools for prototyping and hands-on learning must be available and plentiful.

**Integration—anchor analysis.** Successfully integrating an application generator into an existing production environment requires an appreciation of the architecture and domain-specific software development processes. The team should have a strong understanding of the domain's general components, the relationships among them, and the architectural rules used to enhance them. Although these rules are sometimes not honored, they need to be rediscovered. Knowledge of them is extremely important when decisions are being made about which functionalities should be moved into the application generator, which functionalities should remain in the process, and how the remaining functionalities should be reorganized for reuse.

The team should also have a complete understanding of the development processes currently used to introduce new functionality into the domain. Repetitive and mundane steps are prime candidates for automation. Eliminating mechanical steps performed by engineers can greatly increase efficiency and productivity. Only when armed with the knowledge of the underlying domain architecture and the existing processes used to deploy features can a team use technology to successfully automate the production of new products.

Most organizations prefer to remain anchored in a stable state and actively toil to ensure that the status quo prevails. Because each domain has a unique history, deploying a new technology within that domain demands a unique plan of attack, constructed with a keen awareness of the domain's organizational structure. Consider a domain as being held in place by a set of anchors. We can think of these anchors as forces, events, and practices that support the current state. Anchors keep things the way they are despite new forces and changes exacted on the organization. Anchors are neither good nor bad—they make no moral statement; they just are. Some common examples of organizational anchors include:

- Existing software and production methods;
- Schedules driven by individual features;
- A single customer perspective for requirements;
- Quarter-to-quarter decision making;
- Total quality management process documents;
- Project management procedures;
- Current organizational structure;
- Educational requirements, or lack thereof;
- Existing desk-top environment; and
- The culture regarding team work.

Adding new forces while ignoring the anchors is a common cause of failure. Anchors can be found by revisiting the list of complaints generated as the vision was being constructed. Once the domain's anchors are recognized, they can be changed or destroyed. In some instances, the anchors—for example, the old domain structure—need to be cut free, while other anchors—such as demanding education for competency—should be reworked and used to help create change. New anchors—new processes and tools—may even be created to halt sliding back into the old paradigms. Again, each domain's set of anchors is unique, and the task of the team is to enumerate the anchors and understand their place and power over the organization.

## The Technology Transfer Process

Technology transfer can be systematically deployed, even on systems that have an existing production environment. First, the team must define the scope of the problem, enumerate the various tradeoffs for each solution, and describe the aspirations of the business. Then they must analyze the organizational anchors, those things that hold the culture in place. Using this analysis, they can create, review, and deploy a plan of adoption until success is imminent or failure demands reevaluation.

**Defining the scope of the problem.** The team initially generates a hypothesis regarding the use of some new technology. Further discussions occur regarding whether that technology is appropriate for reengineering a domain for reuse. If the technology is deemed worthy, the team can create a vision to guide all future work and a set of general metrics for cost, performance, and schedule. The team selects a champion among its members and is commissioned to start the project.

**Understanding the business.** In this step, the team performs two analyses, the first on factors that shape the business market and the second on existing software development processes. If the market demands more products from a domain, the domain is most likely a good candidate for reengineering. Otherwise, it is best to skip domains that have little future business. If a domain seems to have a strong, clear future, the next step is to analyze the processes currently used to produce its products. One by one, the team should enumerate steps unique to the domain that are used to describe, build, and deploy products.

At this point in the process, the anchors of the domain should be distinguished and listed in order of priority. To address those anchors, the team creates a general plan that contains a description of which technologies will be used to reengineer the domain, along with approximate cost estimates. Given this knowledge of cost and the expected benefits, the team can decide if and when the project should proceed. The importance of a cost/benefit analysis cannot be overemphasized, because it is easy to get caught up in the allure of a new technology and promote that technology for technology's sake.

**Creating and reviewing the plan.** During this phase, the team creates a project-based process. This process contains externally visible events that can be measured against a schedule. The process should include educational concerns, domain analysis, design, modeling concerns, tool acquisition, role enumeration, coding, testing, and team considerations. The domain team reviews the plan and then uses it as a guide for the reengineering effort.

**Deploying the plan.** As the plan is deployed, team dynamics should be emphasized. Adherence to the schedule is a good sign of progress, but to fully judge the quality of the work, the team must evaluate the artifacts and prototypes. If the process defined by the plan is not working, the team must address the barriers immediately and recommit to a process that will work.

**Evaluating success.** After the project is completed, its initial metrics are used to evaluate its success. Whether the project succeeds or fails, the team should write a postmortem document to catalog the wisdom it has collected during the project. This postmortem is shared with the organization so all members can benefit from the past and integrate those successes into future projects.

## Lessons Learned

The task of technology transfer is to successfully match technical solutions to business problems and then to introduce them into common usage. We use a formal technology transfer process in which appropriate technologies are systematically deployed. The responsibility for success lies with the domain experts who are commissioned to apply the process for technology transfer. These early adopters must be committed to deploying technologies that serve the needs of their company. We recognize the limitations of technology and believe that successful deployment depends on addressing a domain's concerns and understanding the anchors that hold the domain in place.

Technology transfer is a difficult task. Although it bears great responsibilities and demands courage, we believe that the numerous opportunities it affords—to make breakthroughs; to increase the satisfaction of our customers; and to spark creativity, passionate commitment, loyalty, and a feeling of ownership in software development teams—justify this burden.

From our experiences introducing the domain engineering technology, we have learned that the problem is not so much one of searching for the right technology, but of giving developers a way to choose a new technology and then facilitating access to it. Technology alone leads to sporadic success. A few extraordinary people will always succeed, but expanding that success requires more than technology—it calls for organizational and procedural support.

To foster culture change, an advocate is needed inside the organization. More pragmatically, the advocate needs to oversee resources, benchmarks, and metrics—that is, to manage and market the change. Managing the change includes modifying developers' day-to-day activities so they see the new technology as a choice and consider it routinely as they report estimates, plans, and benchmarks.

Organizations must plan toward a goal in which integration is the final hurdle. Any new technology is going to change the development process for the individual and the organization. As the technology diffuses to each part of the development environment—design documents, source control, configuration management, compile time, and run time—it must remain realistic to integrate that technology with existing interfaces and procedures.

### Acknowledgments

This paper summarizes the experiences of several teams within Lucent Technologies that have applied the domain engineering techniques. We are indebted to the members of the Software Production Research Department of Bell Labs, champions of the FAST process. The early adopters of the methods deserve praise for educating themselves and us, and for giving demonstrations long past the point of fun. We especially appreciate the efforts of the Hardware–Software Interface (HSI), Asset Information Manager (AIM), and Domain-Engineered Configuration Control (DECC) teams. We also acknowledge the management team that supported this effort with their dollars and policies. We also thank Dr. David Weiss, Dave Cuka, Janel Green, Bob Olsen, Dick Braatz, Doug Stoneman, Mark Ardis, Neeraj Gupta, Mark Bradac, Mike Moy, Lloyd Nakatani, Bob Nawa, Gary Eldredge, and Jan Sharpless for their contributions.

### References

1. D. M. Weiss, "Family-Oriented Abstraction Specification and Translation: the FAST Process," *Proceedings of the 11th Annual Conference on Computer Asssurance*, Gaithersburg, Maryland, IEEE Press, Piscataway, New Jersey, 1996, pp. 14–22.
2. R. G. Creps, M. A. Simos, and R. Prieto–Diaz, "The STARS Conceptual Framework for Reuse Processes,"*Software Technology for Adaptable, Reliable Systems (STARS) '92*, Nov. 1992. http://www.asset.com/stars/
3. E. Karlsson, *Software Reuse: A Holistic Approach*,

John Wiley & Sons, New York, 1995.
4. T. J. Biggerstaff and A. J. Perlis, *Software Reusability, Volume 1, Concepts and Models*, Addison-Wesley, Reading, Massachusetts, 1989.
5. R. R. Macala, L. D. Stuckey, and D. C. Gross, "Managing Domain-Specific Product-Line Development," *IEEE Software*, Vol. 12, No. 12, May 1996, pp. 57–67.
6. J. O'Connor, C. Mansour, J. Turner-Harris, and G. H. Campbell, "Reuse in Command-and-Control Systems," *IEEE Software*, Vol. 11, No. 5, Sept. 1994, pp. 70–79.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995.
8. J. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, Sept. 1984, pp. 564–573.
9. J. C. Cleaveland, "Building Application Generators," *IEEE Software*, Vol. 5, No. 5, July 1988, pp. 25–33.
10. M. L. Griss, "Software reuse: From library to factory," *IBM Systems Journal*, Vol. 32, No. 4, Oct. 1993, pp. 548–566.
11. *Reuse Adoption Guidebook*, SPC-92051-CMC, Software Productivity Consortium, Herndon, Virginia, Nov. 1993.
12. *Reuse-Driven Software Processes Guidebook*, SPC-92019-CMC, Software Productivity Consortium, Herndon, Virginia, Nov. 1993.
13. Lloyd Nakatani, "Jargons and Infocentrism," *Conference Record of POPL '97: The 24th ACM SIG-PLAN–SIGACT Symposium on Principles of Programming Languages, Proceedings of DSL '97*, Paris, ACM Press, New York, 1997, pp. 59–74.

***(Manuscript approved March 1997)***

*ROBERT LIED is a member of technical staff in the Call Processing Features Development Department at Network Systems in Naperville, Illinois, where he works on processes and techniques for domain-specific reusable software. Mr. Lied earned a B.S. from Syracuse University, Syracuse, New York, and an M.S. from Purdue University, West Lafayette, Indiana, both in electrical engineering.*

*LYNN P. PAUTLER, director of System Management and Quality Architecture at Network Systems in Naperville, Illinois, received a B.S. in biology and an M.S. in entomology from the University of Illinois in Champaign, and an M.S. in computer engineering from the University of Michigan in Ann Arbor. Her interests are in how the evolution of natural systems relates to software, particularly the integration of new technology into existing systems.*

*PATRICK E. HELMERS, a technical manager in the System Management and Quality Architecture Department at Network Systems in Naperville, Illinois, is responsible for promoting object-oriented analysis and design in the context of domain engineering. He holds a B.S. from Southern Illinois University in Carbondale and an M.S. from the Illinois Institute of Technology in Chicago, both in computer science.* ◆