Understanding and Addressing the Essential Costs of Evolving Systems

Joseph Davison, Dennis Mancl, and William F. Opdyke
{jwdavison, mancl, opdyke}@lucent.com

Abstract

A major attribute of telecommunications software systems is change. For evolving telecom systems, significant expertise is needed to handle effectively and capitalize upon these changes. This paper discusses some of the key dimensions of change that occur during telecom systems software development, the areas of expertise that software developers apply in managing these changes, and some of the means by which high performing project members have overcome the learning curves associated with these systems. We base our results on data gathered from several Bell Labs multi-year development projects and from interviews with experienced staff.

Introduction

One of the biggest challenges facing telecom software developers is the fact that any truly useful software system will inevitably evolve.

For evolving telecom systems, significant expertise is needed to handle effectively and capitalize upon these changes. Such expertise does not come cheaply; learning curves are significant for evolving telecom systems. New employees are often surprised by how much there is to learn. Even employees with 10 or more years of telecom experience often find that they have a tremendous amount to learn when they move onto a new project or even into a new area of their current project.

This paper discusses some of the key dimensions of change that occur during telecom systems software development, the areas of expertise that software developers apply in managing these changes, and some of the means by which high performing project members have overcome the learning curves associated with these systems. We base our results on data gathered from several Bell Labs multi-year development projects and from interviews with experienced staff.

It is more difficult to design for change than to design for modularity, testability, portability, performance, or ease of use. "Evolution" or "change" is a difficult thing to predict, yet it is an essential characteristic of most major software development today. (While there is, of course, software that does not need to evolve, little of it is of long-term commercial interest!) While much of the software design effort in a rapidly changing product is reactive rather than proactive, steps can be taken to anticipate design changes and introduce design flexibility in the right places if we understand the basic mechanisms behind the evolution of software systems.

Change and Co-Evolution

Co-evolution

Change, particularly in today's telecommunications industry, is inevitable.

Consider the following example:

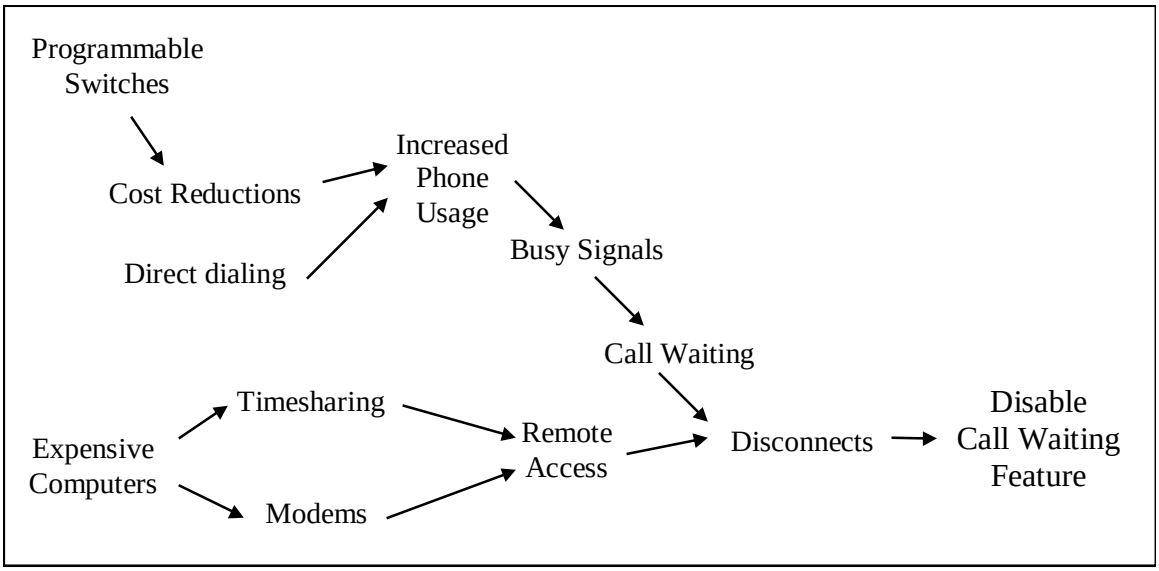

Figure 1: Communications & Computing Technologies (and User Needs) Co-Evolve

Figure 1 illustrates how telephone capabilities and user needs co-evolved through several decades of advances in computing and telephony. A product incorporates new capabilities. These new capabilities, in turn often change how customers do their business, which in turn leads to new product requirements. Rapid advances in technology accelerate this process, sometimes allowing radical new capabilities to be incorporated, sometimes inspiring customers to attempt radical changes in how they do business.

In this example of co-evolution, the development of programmable switches led to cost reductions which, coupled with the introduction of direct dialing, led to increased usage of the telephone. As phone usage increased, it became more likely that the person being called was busy. Seeing an opportunity to increase revenues, the "Call Waiting" feature was introduced. This feature interrupted the call briefly to insert a special circuit that could apply a tone informing the customer of a second call and allow them to switch between the two callers. Meanwhile, the introduction of computer timesharing and the development of modems led to the use of the telephone for remote access to computers. Unfortunately, if a customer with Call Waiting was using her modem when a call came in, the brief interruption caused the modem to disconnect. This led to the development of the "Disable Call Waiting" feature, whereby the customer could dial a code before placing a data call to prevent the Call Waiting feature from acting.

Similar examples of co-evolution in technologies and user needs are seen in word processing and electronic mail. Roles like manager, secretary, and typist have all changed as PCs and word processing software have been broadly introduced. Many people with technical/ professional job titles find themselves regularly performing their own word processing tasks as a secondary part of their jobs. Since word processing work is not their everyday job, they need software tools that are more "user friendly". As a result, more features have been introduced into word processing packages to support novice users. Similarly, as technology supporting electronic mail has become pervasive in organizations, the quantity and complexity of email messages have led to the need for email browsers that can rapidly sort and delete email messages, and handle a multiplicity of attachments.

Dimensions of Change

Such examples illustrate four key dimensions of change:

- *User needs and expectations.* Requirements aren't static. Even if the initial requirements are "on target", the product will enable customers to do some things more easily or more powerfully than before. This will change both their behavior and their expectations from future products
- *Market / competitive forces.* Some changes are dictated by changes in the competitive environment; telecom examples include the rise in digital switching during the 1980s, the advent of packet-based telephony in the 1990s, and the recent changes in the pricing of long distance telephone calls in the US (from a distance-sensitive model to a flat rate per minute model).
- *Interfaces (internal and external).* Each software system collaborates with other systems; interfaces with those external systems often change. External interfaces may be based on industry standards (from the International Telecommunications Union (ITU) and other sources), but these standards are often evolving in response to technology changes. Internal interfaces can also change: software systems are often built using existing commercial components such as databases and communication protocol stacks; these components may need to be upgraded to provide better performance or more features. Design changes can result from these internal interface changes.
- *Designers and developers' understanding of the application domain.* As designers and developers grow in their experience, they better understand the key design abstractions and can envision better ways to make their product work.

The useful lifetime of a telecom product sometimes spans decades. For example, the development of several of the electronic switching systems in the field today (some of which are still under active development) started 30 or more years ago. There are many factors that influence these lifetimes: for example, the difference between the cost of evolving a product to meet new market needs and the cost of building a new product "from scratch" with comparable functionality. During the lifetimes of these systems, significant changes occur in the functionality of these systems and in the technologies used. Such systems may truly be called "Evolving Systems".

Traditional Models Fail to Address Essentials of Evolving Systems

The models of the software development process, as taught in traditional software engineering texts, often ignore the essential aspects of developing and evolving telecom systems. The "traditional" (waterfall) software development process is composed of a series of stages or phases (Requirements -> Design -> Code -> Test -> Maintenance). While the focus of most computer science and software development training is on the first three phases, most of our product development costs are incurred in the final, "maintenance" phase. Calling that phase "maintenance" is really a misnomer. To use a home building metaphor, most of the software "construction" that we perform is remodeling or extending an existing structure, rather than performing repairs. People are more apt to remodel or extend software than make such changes to buildings, because of software's apparent flexibility, apparent extensibility, and ease of replication.

Some models attempt to use a modified waterfall development model, considering each release as a separate product that can be modeled by the traditional waterfall model, with the total lifecycle thus being an overlapping sequence of waterfall developments. This technique has no standard way to account for the dependency of later releases on earlier releases, and thus fails to account for some of the essential costs of developing evolving systems.

Boehm's "Spiral" model[1] seems better suited to evolving systems, because it explicitly depicts the dependencies. In Figure 2, each loop of the spiral represents a new product cycle – another orbit through the plan, specify, design, implement, and review activities. Using this model, one can show two different essential aspects of evolving systems (shown by the two arrows in Figure 2):

1. Following the spiraling arc from one release to another, one can see the need for "compatibility" between different releases. For example, a later release of a software product may need to be able to access persistent data created in an earlier version of the product. Users of word processing software are well aware of the difficulties new releases of the software can introduce when archived documents need to be viewed with newer releases, or new documents need to be viewed with older releases. Significant effort in telecom systems must be directed to database evolution between one release and the next – such issues often limit the ability to "skip" releases, such as moving from version 3 to version 5 (skipping version 4).

2. A second essential aspect is the "radial" dependency of the design of a new release on the older releases. The new requirements may seem to be minor to the customer, but the actual impact will depend greatly on the structure of the existing system. Furthermore, the fact that new employees joining the project will need to become familiar with the existing system is much more obvious in the spiral model. Anyone who has been involved in the development of evolving systems is well aware of this need, but when it is not obvious in the development models, there may be a tendency to think these issues are accidental.

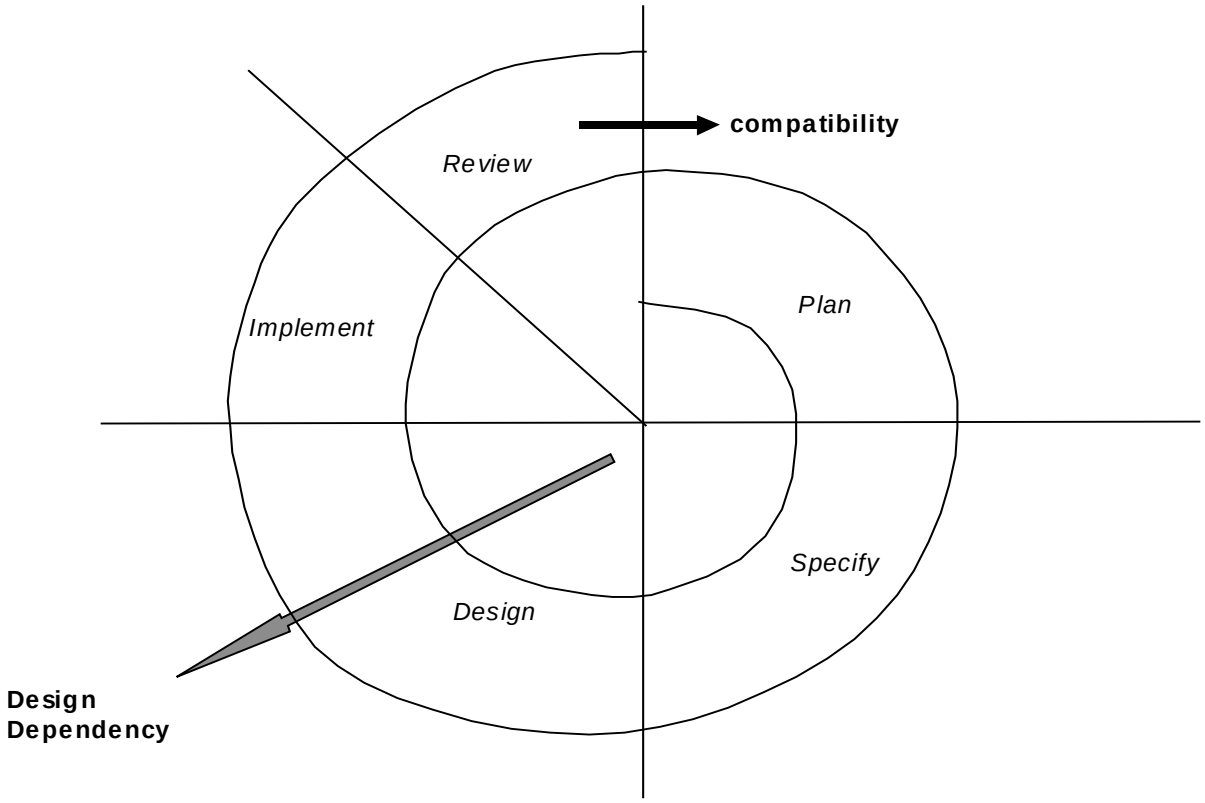

Figure 2. Spiral model of development

When a better development model is used, it is easier to maintain awareness of total lifecycle costs of development when making managerial decisions. For instance, "cleanup" is an activity difficult to justify using the waterfall variants, because there is no depiction of the benefits. The benefits are easier to see with the spiral model, because it is obvious that they are part of the foundation of later releases.

Architectural Strength Lost & Development Costs Increase

Over time, as they evolve, software systems often lose architectural strength due to factors that some have described as maintenance degradation[2] or software entropy.[3] A design that starts out "clean" often gets compromised as a system evolves. To complicate matters, while a system evolves and grows more complex, the organization responsible for maintaining that system also evolves and inevitably loses project members who possess valuable expertise about this system.

People are more apt to remodel or extend software than make such changes to buildings, because of software's apparent flexibility, apparent extensibility, and ease of replication. Increased complexity, combined with loss of expertise, can lead to escalating development costs. What can an organization do?

The Challenges in Developing Evolving Systems

Discovery Costs

Imagine, for a moment, that you have just been assigned as project manager for a development that is critical to the success of your business.

1. The project *must* be completed within 15 months.
2. Both you and your boss are given, from a trusted source, an estimate of 100 technical head-count years (THCY) for building a solution from scratch.
3. Your boss wants you to build your system by using/ extending a legacy system that "provides 80% of the required functionality."
4. Your job is on the line – you will live (or die) by your staffing estimate.

What is your staffing estimate?

- A simple-minded estimate: development using the legacy system ought to only require 20 THCY – 20% of the cost of building the system from scratch.
- A more informed estimate: the project using the legacy system could require 40 THCY, 50 THCY, or maybe even more than 100 THCY (the cost of building from scratch).

Why is this the case? While there are many factors, the predominant factor that arose in our interviews and analysis was what we call *discovery* costs.

Discovery is the process of learning what you need to know in order to do your job. Discovery costs are the costs associated with that discovery process. We classify the kinds of knowledge that are needed to do significant development work on a particular evolving system:

- **Application domain** – A developer must learn to understand the application domain of the product they develop. If the application domain is not widely known, as was long the case in telephony, it is almost certain that this will be learned on the job.
- **User requirements** – The overall requirements for such products may be considered part of the application domain, but the specific requirements for the version of the particular product being developed will be incurred each time a version is developed.
- **Relevant design paradigm(s)** – The design of evolving systems is a non-trivial task; people need to be trained to do it. Today many employees arrive with a working knowledge of software techniques such as object oriented technology, but may never have been exposed to finite state machines, which are ubiquitous in telephony. Furthermore, in evolving systems it may be necessary to learn a paradigm that was used in the original development but is seldom used today.

- **Development languages –** It is necessary to understand the syntax and semantics of the languages used for development. This is generally much easier than learning the appropriate paradigm(s).
- **Software development environment** – Clearly it is necessary to know how to use the tools. Tools evolve as well as products; significant advances in tool capabilities almost certainly requires significant discovery costs in retraining developers.
- **Library of available components**. – One of the major areas that requires significant discovery is learning the library of components available. This includes not only frameworks and libraries specifically intended for (re-)use but also the "legacy software" – and that can take considerable effort.

A simple model of these costs can be useful in examining the impact of these costs on productivity.

We represent an individual's total effort by **f**, and the effort they put into discovery as **d**. In interviewing people, it's often easier to get an estimate of **D = d/f**, the fraction of a developer's time spent in discovery. We also represent the "useful" work by **w**, and assume that all effort is either discovery or useful work (admittedly an oversimplification). That is, we assume **f = w + d**, or **w = f – d**.

We can then define the efficiency of a developer as the fraction of useful work they are capable of: **E = w/f = (f – d)/f = 1 – d/f = 1 – D**.

Our own experience and the results of interviews with other experienced developers suggests these costs are very significant. Figure 3 plots estimates of an individual's efficiency for three telecom software projects at Bell Labs. The data shows an individual's efficiency (**E = 1 – D**), by development cycle for that developer. These projects collectively spanned nearly two decades. The projects varied in size from 100 developers to more than 1000 developers. One of the systems used a centralized processor architecture; two used a distributed architecture. Several different programming languages were used across these projects.

The estimates in Figure 3 were gathered from dozens of developers, collectively spanning several projects. These developers estimated their discovery costs by development phase. Such costs vary from individual to individual, depending largely on their previous experience. They differ as well by phase of the development cycle and how the project assigns people – for instance, in systems where developers do their own testing in a laboratory setting, the costs will often be significantly higher during testing than in coding. This is so because the development language is generally rather stable, while the lab changes as the product evolves – thus each release the tester comes to an environment that differs from what they learned last time.

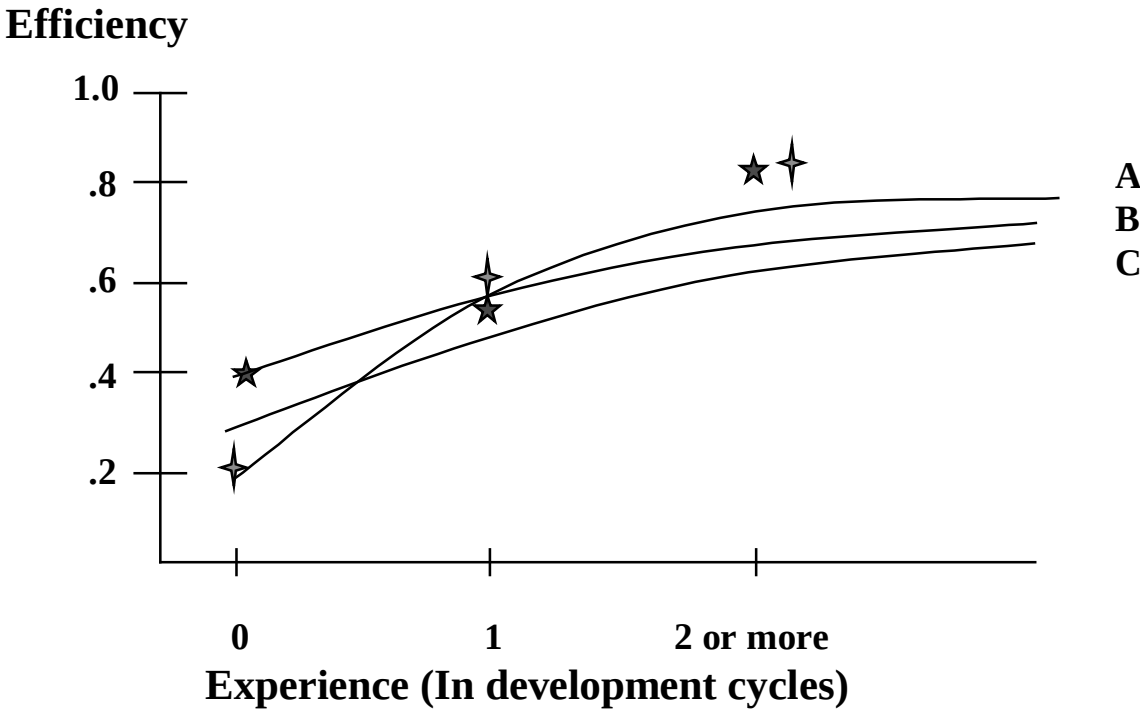

Figure 3: Estimated Efficiency vs. Experience for three development projects

Subsequent discussions with other experienced staff, and estimates derived by other experienced developers during estimation exercises such as described at the start of this section, suggest that these results roughly apply across a broader range of product developments.

Several key insights can be gained:

- For individual developers, discovery costs were seen as the dominant costs for the first two development cycles. Whenever the majority of developers are inexperienced, discovery costs may well be the dominant cost of development.
- The efficiency of experienced developers (i.e. those who had worked in the same area/sub-system of the same product for at least two development cycles) was two to four times that of staff who were new to that part/ sub-system of the product. Whereas new project members spent 60-80% of their time doing discovery tasks and 20-40% of their time doing "useful work", experienced developers spent approximately 20% of their time in discovery and 80% doing "useful work".
- There was a "core" discovery component (approximately 20%) that continued, even for experienced staff. This is an essential cost of evolution – as long as the systems continues to change, this cost will continue, disappearing only once the system becomes stable and stops evolving.

These insights reinforce the intuitions that experienced staff (i.e. those who have worked through two or more prior releases) can work much more productively than newer staff on large, legacy system projects, and that learning/ discovery is a part of everyone's job. What was surprising (at least to some) was how significant/ time consuming the discovery process can be for newer employees.

Further analysis of data collected by Davison, Modica, and Baumann showed that, while discovery costs for an individual vary by experience, they also vary by the phase in the development process. For experienced staff, discovery costs remain larger for the early (requirements/ design) and the latter (system testing) phases for the coding phase. Requirements and Design discovery costs remain high in the early phases because the system is evolving – at least in large systems it is likely that new requirements will involve a part of the system that is unfamiliar. Testing discovery costs remain high for developers largely because of evolution in the labs and test equipment; also, the testing process is often less standardized and less automated than the development process.

In summary, discovery costs are pervasive – related to evolving systems, customers and markets, and regarding new/ disruptive technologies and trends.

Problem Understanding

As noted above, requirements and design discovery costs remain relatively high, even for experienced staff. The only "constant" is change. Unforeseen changes in customer needs and marketplace preferences will occur, sometimes in ways that do not neatly align with the current structure of a system.

Problem understanding is hard – and, frankly, for most engineers it is much less satisfying to understand a problem than to implement a solution. In addition, customers may be hard to identify or hard to reach, and may be unwilling to describe their needs in terms other than "I'll know it when I see it!"

Not surprisingly, a significant percentage of the areas of concern that arise during architecture reviews at Bell Labs/ Lucent have related to problem understanding.

Disruptive Technologies and Blindsiders

There is yet another key area of discovery for telecom projects – blindsiders and disruptive technologies.

Understanding your customers and incrementally evolving your current products might be sufficient for business success – if your marketplace was stable and technology advances were modest.

As Bower and Christensen note,[4] our dynamic marketplace and rapid technology advances result in an environment where established companies may find themselves driven out of markets and perhaps driven out of business by new entrants whose products leverage

disruptive technologies. Rational business planning dictates focusing on the needs of current customers and high margin products – typically these products are based on established, "sustaining" technologies. Newer, disruptive technologies are often immature at first – they have lower margins, but many of these technologies gain maturity very fast. When disruptive technologies mature to the point that they satisfy customer needs, they begin a gradual (or not-so-gradual) displacement of the established technologies and products. Failing to anticipate early and capitalize upon the newer, disruptive technologies can be disastrous.

The Challenges – How Do We Address Them?

How Do Staff Discover?

When we interviewed experienced architects and developers across several business units at Bell Labs/ Lucent Technologies,[5] these are the ways that they told us they do discovery:

- Interacting with other staff (colleagues and/or experts)
- Reading code
- Experimenting and debugging in lab or simulator
- Participating in reviews/ Testing
- Taking training courses
- Reading documentation
- Building something (preferably, a simplified example)
- Developing a behavioral simulation.

While all tasks are valuable, their relative use (i.e. the time spent in each area) varied among technical staff. This variation is likely due both to the product/ sub-system and to the background and personality of the developer/ architect. For example, extroverts may spend more time talking with other staff, while introverts spend more time reading.

Assigning experienced technical staff as mentors for new members of a project can be an effective technique, both for reducing the discovery costs for the new employees, and for reducing the impact of discovery costs on the broader organization.

Further details on this analysis of discovery costs at Lucent (and elsewhere) are contained in the panel position statements for an OOPSLA '97 panel session.[5]

Tools & Techniques That Aid in Discovery

Which software tools and techniques can aid in avoiding or reducing discovery costs?

There is no single tool that can make all discovery costs disappear. However, lack of tools can significantly increase discovery costs. This is less of a problem with modern development environments, but experience has shown that the lack of cross reference tools, in particular, can significantly increase the cost of development. The inclusion of

good browsers, debuggers, and simulators in modern development environments reflect their value in reducing discovery costs.

The choice of software technology can also have a significant impact on the resulting discovery costs. Some technologies that seem to help are:

- Using an existing software framework
- Creating a new software framework based on a generalized view of the current problem
- Applying some standard design patterns
- Creating a high-level model of a system with use cases

Software frameworks can help to reduce discovery effort. A framework is "a collection of objects, classes, or subsystems that provides most or all of a needed function and can be tailored to different situations."[6] Frameworks have been widely promoted as a good source of design-level and code-level reuse.

A mature, well-documented framework provides simple mechanisms to redefine some of the framework's standard behavior and to extend the framework's behavior with new functionality. A developer doesn't need to be an expert on the internal workings of the framework – a developer only needs to check that the new extensions fit into the design rules of the framework. Some frameworks include supplementary tools to help make the process of extending a framework more automatic, such as the "wizards" in Microsoft Visual C++ development environment and similar automatic class generation tools in Java Beans tools.

Mature frameworks can significantly reduce discovery costs and development intervals.[5] A mature, well documented framework makes explicit the domain abstractions, paradigms, and other design information that staff members need to get "up to speed", as well as providing a vocabulary that project members can use to communication with each other. Object oriented frameworks usually support rapid iteration and prototyping that facilitates exploration and learning.

On the other hand, overuse or inappropriate uses of software frameworks can actually increase discovery costs. Some frameworks may have deeply-embedded design features that cause performance problems in some applications. These problems might require a lot of knowledge of the internal structure of the framework in order to make an accurate diagnosis of the problem. Using multiple frameworks in a single application can also cause integration problems: this happens when different frameworks make incompatible assumptions about their environment.

The positive and negative effects of the use of tools that rely heavily on frameworks have been experienced in Lucent software development projects. For example, Corba (the Common Object Request Broker Architecture) and the use of Corba-based software environments has been both a help and a hinderance. The use of commercial Corba packages has simplified the creation of small portable software systems based on a

distributed architecture. On the other hand, Corba has been tricky to use in multi-threaded environments and in performance-critical applications. Reliance on an evolving framework adds extra dependencies in the software development process: the success of some applications has become more dependent on the delivery and update cycles of external tools and frameworks vendors.

Creating a new framework is a great way to leverage the knowledge of domain experts. A framework may be a good thing to build for a software organization that plans to develop a series of applications with similar capabilities. The core capabilities can be "factored" into a common set of implementations in the framework, and new applications can be built on top of the common base. Some techniques such as Domain Engineering have been proven to be useful in the design of frameworks for well-understood problem domains.[7]

There are some significant pitfalls in the creation of new frameworks. A framework needs to have good documentation. This documentation should include example programs illustrating the typical ways to extend the framework plus clear documentation of the rules and constraints of the framework. A poorly documented framework can cause the average software developer to have to discover a lot of these rules and constraints without much help.

Many of the standard design patterns (such as the object oriented design patterns described in the *Design Patterns* book[8]) document good design practices. These design patterns contribute to lower discovery costs in two ways. First, the patterns are a useful vocabulary for common design techniques – a designer can explain some of the important mechanisms in a design document by referring to the names of the patterns that are used in the design. A newcomer to a project can get a better idea of the design intent of a section of the design if it is described using pattern vocabulary. Second, some of the design patterns (such as the Facade, Bridge, and Proxy patterns) are techniques that promote clean separation between different sections of a design. In these kinds of designs, some of the internal changes that are required to support new features or to improve system performance can be achieved using "refactoring" techniques.[9]

Finally, more software development teams are writing some of their system-level requirements documentation in terms of use cases, which provides a good end-to-end view of system operations for developers who are new on a development team.[10,11] Use cases describe the externally-visible behavior of the system under design by documenting groups of related high-level scenarios. Use cases can be used to give requirements writers and customers a method of describing the requirements that are most likely to change in a system, so the system designers can be sure to keep certain parts of the design flexible.

Why do use cases have an impact on discovery costs? One of the biggest problems in making design changes is "understanding the big picture". Each design change has a chance of breaking existing features in addition to adding new features. A developer must understand more than just the localized structure of the code that is being modified: the designer has to understand the context of the change. By referring to a high-level use case model of the system, the developer can quickly grasp which scenarios are likely to be

affected by a specific design change. The use cases act as a simple behavioral roadmap to a large software system.

Addressing Problem Understanding

Problem understanding is a critical issue – and one which is addressed in Lucent's architecture reviews and related training.

Architecture reviews and associated training are an important part of our corporate culture, and have paid dividends for our product development teams. Hundreds of architecture reviews have been held in recent years – where project members present the goals and state of their project to a review team of experts from other business units. The review team is often able to uncover blind spots that a product team may have missed – in time to correct them and avoid costly rework downstream.

Training courses focus on developing effective problem statements and formulating a balanced product architecture that meets form, function, economy, and time requirements as well as balancing technology, design, and management concerns. Checklists have been defined to aid in evaluating architectures. Students also learn and apply a range of creativity techniques especially useful in the up-front stages of product definition.

While these resources are valuable, problem understanding remains a challenging problem.

As our systems evolve, some changes which may look easy to our customers prove to be difficult to implement. Customers' views of our products abstract away many of the complexities of these products, some of which account for much of the development complexities. Some of the issues that are glossed over in the customers' views of our products are resource contention among multiple simultaneous calls/ connections, handling asynchronous events, error handling, and feature interactions. If we want to reduce the disconnect between customer expectations and development realities, we need to foster effective customer engagement and apply relevant expertise (often a team of experts) to handle the tough problems. Acquiring the relevant expertise comes through the discovery activities described previously (interacting with other staff, reading code, experimenting, etc.).

Addressing Disruptive Technologies

There are many examples (documented by Christensen and others) of companies that failed to recognize and capitalize upon emerging/ disruptive technologies. Fortunately there are means by which established players in an industry can leverage emerging technologies, redefine themselves as a business, and achieve significant growth.

Bell Labs has long been committed to both basic research and to nearer term, more applied research, and has aggressively moved to capitalize on new technologies within our business units.

One of the ongoing challenges that Lucent and other high technology companies face is supporting our product development staff in understanding new technologies and trends, while maintaining the necessary focus on bringing near term products to market. One of the means by which development staff become familiar with emerging and potentially disruptive technologies and trends is through an ongoing series of half-day Blindsiders seminars, held several times each year. These seminars are routinely attended by several hundred staff at each of several Lucent sites. In addition to the seminars, a wide range of short courses are offered on emerging topics, and annual forums such as the Lucent Software Symposium bring together staff from across our business units.

It is very challenging to effectively train our diverse audience, which consists of recent college graduates, market hires, and staff with considerable "in house" work experience. The training must contain a sufficient level of detail in order to achieve credibility with this audience. But the technical detail in the seminars should be balanced with discussions of market related and financial topics. Many of our product development staff want "news they can use" – much of their learning comes from seeing experience reports, interacting with experts from inside and outside Lucent, and applying newly learned concepts in student exercises. For example, in one of our recently introduced two-day short courses on evolving systems, we include a mixture of in-company product team experiences, current and emerging industry best practices, a discussion of discovery costs issues, and seven team-based exercises where students take on design, project management, and analyst roles. A final team exercise brings together students with common business unit and/or project interests to discuss how to apply what they have learned "on the job" – including a discussion of challenges and how to overcome them.

Trends in the telecom industry change quickly; members of our seminar planning teams and curriculum development teams (many of whom have development responsibilities within our product groups) actively monitor the literature and the competitive landscape. Effective training is hard work – but we see benefits both for our business units (as they compete in changing markets) and our individual staff members (as they manage their careers).

Summary

Much of the complexity in building telecom systems comes from the need to adapt to and capitalize upon change. Discovery costs are significant for evolving systems. Problem understanding and disruptive technologies are important areas to understand in planning and developing telecom products. This paper quantifies the costs of discovery, as well as describing techniques that assist our telecom development projects in developing the understanding needed to succeed.

References

1. Barry Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, May 1988, pp. 61-72.
2. Grady Booch, *Object-Oriented Analysis and Design*, second edition, Benjamin Cummings, Redwood City, California, 1994, p. 263.
3. Luke Hohmann, *Journey of the Software Professional*, Prentice-Hall, Upper Saddle River, New Jersey, 1997, p. 349.
4. J. L. Bower and C. M. Christensen, "Disruptive Technologies: Catching the Wave," *Harvard Business Review*, Jan.-Feb. 1995, pp. 43-53.
5. Steven Fraser, Kent Beck, Grady Booch, Jim Coplien, Ralph Johnson, and Bill Opdyke, "Beyond the Hype: Do Patterns and Frameworks Reduce Discovery Costs?" *Proceedings of OOPSLA '97*, Atlanta, Georgia, Oct. 5-9, 1997, pp. 342-344.
6. Alistair Cockburn, *Surviving Object-Oriented Projects*, Addison-Wesley, Reading, Massachusetts, 1998, p. 9.
7. Robert Lied, Lynn P. Paulter, and Patrick E. Helmers, "Introducing Software Reuse Technology," *Bell Labs Technical Journal*, Winter 1997, pp. 188-199.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley, Reading, Massachusetts, 1995.
9. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Reading, Massachusetts, 1999.
10. Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard, *Object-Oriented Software Engineering: a Use Case Driven Approach*, Addison-Wesley, Reading, Massachusetts, 1992, pp. 442-450.
11. Geri Schneider and Jason P. Winters, *Applying Use Cases*, Addison-Wesley, Reading, Massachusetts, 1998, pp. 133-135.