

Domain Engineered Configuration Control

Mark Ardis, Peter Dudak, Liz Dor, Wen-jenq Leu, Lloyd Nakatani, Bob Olsen,
Paul Pontrelli

Lucent Technologies

Key words: Domain engineering, domain analysis, application engineering, domain specific language, configuration control, FAST, InfoWiz, VFSM, Tcl/Tk

Abstract: This paper describes the experiences of a small team of domain experts in re-engineering the configuration control software for the 5ESS® switch. The project consisted of three main phases: discovery, design, and deployment. During the discovery phase the team conducted a domain analysis of configuration control software. In the design phase the team created domain-specific languages and supporting tools. The deployment phase consisted of trial use and modification in response to feedback from users. Lessons learned during each of these phases are summarized.

1. INTRODUCTION

One of the larger products of Lucent Technologies is its 5ESS® switch. As the product has evolved new software has been written to manage new and improved hardware components. The Domain Engineered Configuration Control (DECC) project was launched in 1994 to investigate opportunities for standardization of this software.

The project performed a domain analysis of the configuration control domain and determined that a reusable collection of algorithms could be designed. The algorithms would be driven by a specification of the attributes of individual hardware components, and a specification of the relationships between those components. New software would be created by writing new specifications and automatically translating the algorithms into working code. In order to accomplish this the project had to specify the reusable algorithms, define domain-specific languages for describing hardware components and their relationships, and build the translators that

would create the code. They also decided to create a graphical user interface (GUI) to simplify the writing of new specifications.

In the next section we describe the domain of interest, 5ESS configuration control, and the domain engineering process we employed. Section 3 covers the discovery phase of the project, where the team conducted a domain analysis of configuration control software. Section 4 covers the design phase, where the languages and tools were created. In section 5 we describe our experiences in deploying the new technology. Important lessons are enumerated in section 6.

2. BACKGROUND

2.1 5ESS Configuration Control

The 5ESS switch is a complex system of hardware and software that has evolved almost continuously since its introduction [Martersteck 85]. Improvements in technology and new applications of the telephone network, such as the Internet, have driven Lucent to introduce new hardware components to reduce costs and enhance capabilities. As a result, the hardware components of the 5ESS switch have evolved, and the software that controls those components has evolved, also.

Configuration control is that part of the 5ESS switch software that maintains the status of individual hardware components, called *units*, that make up a switch. Each request to change the status of a unit is first analyzed to determine if the change would degrade the performance of the switch. If not, the request is allowed, and a sequence of actions is performed to accomplish the change in status. This sequence must ensure that all units function smoothly during the transition.

For example, if a technician wants to remove a unit from service, he issues a request to that effect. Configuration control software then determines if a backup replacement is available to perform the functions of the removed unit. If so, and if other similar checks are passed successfully, configuration control initiates a sequence of actions to remove the unit from service. Some of these actions involve elevating the status of the backup unit to active, de-elevating the status of the first unit to out-of-service, and notifying other units of these changes in status.

2.2 The FAST Process

The Family-oriented Abstraction, Specification, and Translation (FAST) process for software engineering leverages the knowledge of domain experts to design software for efficient production. FAST recognizes that software products come in variations that typically have in common significant functionality and behavior. Rather than duplicate these commonalities, it is better to have only one instance. Therefore, FAST proposes that software families should be constructed by a two-stage process:

1. Domain engineering: Analyze the family of products to understand what is common among and what is variable between family members, and then build an application engineering environment that can generate a family member from a specification of only its unique characteristics.
2. Application engineering: Use the environment to specify a family member, i.e., a customized product, and automatically generate the product from its specification.

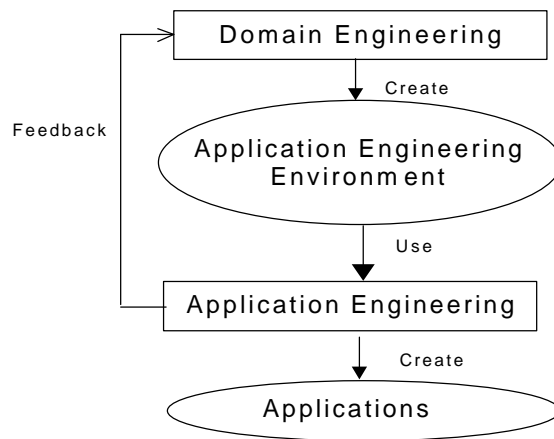


Figure 1. The FAST process of domain engineering

Figure 1 shows this two-step process and the feedback between the two. The domain engineering step of FAST results in a design of the product for efficient production. Domain engineering ensures that every instance of the product has the common characteristics of the family and preserves its design integrity. The application engineering step simplifies software development to a task of high level specification. As the domain evolves further domain engineering is performed to extend or modify the design and specification.

Domain engineering is an activity performed by those who have rich domain knowledge and expertise. They are the guardians of the system's integrity. Application engineering is intended to be efficient to perform, and application engineers who are unaware of the details of the underlying machinery may carry it out. This separation of concerns serves to maintain the integrity of a design as a system evolves, and to increase the efficiency of those who produce individual products.

The FAST process has evolved from work originally done at the Naval Research Laboratory in the 1970s [Parnas 76], [Parnas 78], [Parnas 85], [Parnas 86]. David Weiss developed and introduced the FAST method in collaboration with software developers at Bell Labs [Weiss 99]. More than 30 domain engineering projects have been initiated within Lucent, many of them in switching systems.

3. DISCOVERY

The first step of the DECC project was to perform a domain analysis of configuration control software. The main task was to compose a *commonality analysis* document as a group activity. A recorder transcribed the document during the meetings, with all team members present. A moderator facilitated the process, and helped the group find appropriate representations of their knowledge. (In fact, the same person often performed the moderator and recorder roles.) This style of group writing is unusual, but it emphasizes the importance of consensus amongst experts. Each member of the team was an author of the final document and was responsible for its accuracy. Disputes between experts were resolved through further analysis and discussion.

The first problem to be solved in the domain analysis was its scope. Initially the team hoped to cover all of the software for 5ESS maintenance, including those parts that performed fault detection and recovery. The team quickly discovered that this was too large a domain, given their available resources. Instead, they decomposed the maintenance domain into several subdomains, and selected the configuration control subdomain for further analysis. The main reason for this choice was its importance to future work. The team felt that they could obtain the most benefit by re-engineering the configuration control software.

Once the scope of the analysis was decided the team proceeded to write the document. A commonality analysis document primarily consists of:

- definitions of technical terms,
- commonalities of family members,
- variabilities between family members, and
- parameters of variation, which are refinements of the variabilities.

In addition, the document describes the scope of its work and enumerates issues that were encountered during the analysis.

Definitions captured the common vocabulary of the domain, an area of surprising controversy. Experts would occasionally misunderstand one another, because they were using the same words in different ways. In fact, experts would sometimes be in “violent agreement” with one another, all the while expressing the same idea in different terms. Only after defining terms precisely did they discover their common ground. *Figure 2* contains some example definitions from the analysis. Words that have been defined this way are italicized wherever they appear in the analysis.

Child: A *unit* whose *condition* is dependent on the *condition* of its *parents*.

Primary condition: The availability of a unit: *working*, *ready*, *unready*, or *unusable*.

Unit: A set of *circuits* and a set of associated protocols that comprise a single configurable entity.

Figure 2. Example definitions from the commonality analysis

Commonalities captured what was essentially the same among all family members. Often the commonalities were obfuscated by different implementations of the software. The difficulty of domain analysis lies in abstracting the commonalities from the details of their implementations. This was often done by starting with common examples and then considering possible exceptions to the norm. Reviews performed by additional domain experts helped to ensure that all possible cases were considered. *Figure 3* contains example commonalities from the analysis.

Every *unit* has a *condition*.

Every *unit* has a name and number. *Units* with the same name provide the same functionality.

All *units* of the same name have the same set of *conditions*.

Figure 3. Example commonalities from the commonality analysis

The most important commonalities found were the algorithms for changing the status of a unit. Each algorithm divided into two aspects: a *validation* aspect that checked whether an intended request should be allowed, and a *realization* aspect that performed the appropriate sequence of steps to accomplish the request. In addition to discovering the universal applicability of these generic algorithms across different hardware units, the team discovered that the validation and realization aspects of an algorithm could be separated from one another. This simplified the resulting algorithms and reduced future maintenance. It also provided a common behavior across all hardware units, which was desired by the customers of the switch.

Variabilities described the possible differences between family members. For configuration control that meant different attributes of hardware units and different relationships between those units. Discovering the variability of hardware attributes was relatively straightforward, but occasionally tedious. The main job was to pare down all of the possible attributes to those that affected configuration control. In the final analysis the units were distinguishable on about forty attributes. *Figure 4* contains example variabilities from the analysis.

- Some *unit* names have *inhibit conditions*.
- Some *unit* names have routine maintenance.
- The *units* to which a *unit* is related vary from *unit* to *unit*.

Figure 4. Example variabilities from the commonality analysis

The team discovered that a small set of relationship types sufficed to specify a configuration. Some of these relationships were well known, while others were described inconsistently by various implementations. That is, common relationships were obscured by various implementations. The abstraction of several of the relationships was a major breakthrough for the team.

Once the variabilities had been described, the team refined them into *parameters of variation*. These were precise specifications of the ranges of variabilities, binding times of values, and default values. Creation of the parameters took only a few sessions, but it provided a review of the variabilities for internal consistency. The resulting precise specifications were especially valuable to the following phases of domain engineering. *Table 1* contains example parameters of variation.

Table 1. Example parameters of variation from the commonality analysis

Parameter	Range	Binding Time	Default Value
Inhibit condition	Boolean	Specification	False
Routine maintenance	Boolean	Specification	False
Parent units	List of units	Specification	Empty list

Throughout the analysis effort issues would occasionally arise that prevented the team from making progress. Whenever this occurred the team would write the issue as a question to be resolved and assign an owner to the issue. It was the owner's responsibility to research the issue and propose a set of alternative solutions. The team would consider the solutions and try to reach consensus. If they could not come to consensus they would leave the issue unresolved and seek more research. Most issues were resolved after one round of research, but a few were revisited several times.

The team met once or twice a week over six months to complete the document, with each meeting lasting about 3 hours. Although this may seem like a lot of work, the total effort for this phase was only about one staff-year. Team membership varied over the duration of the project. Some experts came for a few meetings and then dropped out. The core of the team consisted of four experts who were present for almost all meetings.

Other domain experts reviewed draft versions of the commonality analysis document. This provided two main benefits. First, it helped ensure the accuracy of the analysis by employing the knowledge of more experts. Second, it began the socialization of the ideas in the analysis throughout the community of eventual users of the resulting technology. Domain experts outside the team were also consulted on issues identified for research. These informal contacts helped to educate the outside experts in the mission and direction of the DECC team. This process also made those experts better reviewers, as they had a better understanding of the purpose of the analysis.

4. DESIGN

Once the domain analysis had been completed, the team moved on to the design phase of the project. Two of the four domain experts left the team at this point, leaving two experts and the moderator. Their objective was to design and implement the application engineering environment consisting of the following components:

- language for specifying the generic algorithms for configuration control,
- language for specifying the unique attributes of units,

- GUI tool (described below)
- other technology needed to integrate these components.

The team decided to proceed with a spiral development model [Boehm 88], delivering components with increasing functionality on each cycle.

One of the first design issues was legacy software. It was not feasible to rewrite all of the existing software for configuration control, so the new software had to coexist with some residual legacy software. It was desirable to create an interface between the new software and the legacy software. For example, requests that the new software made for unit status information could be made through this interface.

Fortunately, domain experts in software maintenance were already using a technology for interfacing with legacy software. A finite state machine language, called Virtual Finite State Machines (VFSM) [Flora-Holmquist 97], had been developed a few years earlier at Lucent, and was already familiar to several domain experts in software maintenance. VFSM provided a layer of abstraction for separating the stimuli and side effects of state transitions from the descriptions of those states and transitions. The DECC team chose to translate the generic algorithms into VFSM to take advantage of its interface mechanisms to the legacy software.

Another design issue was the choice of technology for defining languages and translators. Although the domain-specific languages were small, none of the team members were experts in language definition or translator writing, so they wanted language development tools that would be easy to learn and use. Furthermore, it was likely that there would be considerable churn in the definitions of the languages.

The team chose InfoWiz [Nakatani 97], a technology invented at Bell Labs for making special-purpose languages quickly and easily. With InfoWiz a language is made simply by specifying a set of expressions in a prescribed syntax, and defining the semantics of each expression by writing actions in a high-level general-purpose programming language. A library of Application Programming Interface functions enables the actions to interface with a ready-made interpreter. Using InfoWiz, domain experts who were not language experts were able to make their own languages. Models of artifacts in the domain were specified in the specialized languages made with InfoWiz, and the models processed to automatically generate other artifacts, such as run-time code.

InfoWiz offered several advantages. First, it was simple to learn and use. Each of the team members became proficient in InfoWiz in only a few weeks. Second, InfoWiz facilitated language design through example. That is, each new language was created by experimenting with common examples. Third, InfoWiz made it easy to process a model to produce multiple related artifacts. For example, the generic algorithms were

translated into two different representations for review, run in a simulator, and translated into run-time code.

During the domain analysis the team experimented with several different representations of the generic algorithms. The style that most reviewers preferred, a simple sequence of actions, worked fairly well for the validation aspect of the algorithms but was inadequate for the realization aspect. This was due partly to the recursive nature of the algorithms, and partly to the need for branching. After several trials the team chose a simple language with three control structures: a conditional statement to choose between sequences, a simple loop based on relationships, and a break statement to skip unneeded steps. This compromise retained most of the flavor of a sequence of actions, but provided the power to describe more complicated structures. *Figure 5* shows the switch algorithm in its textual review format.

```
Tell CLIENT that the switch realization is about to
  begin
Set acknowledgment result to success, and its reason
  to null
Invoke( Elevate spare mate unit )
Did the invocation succeed?
Invoke( De-elevate self unit )
Did the invocation abort?
Set acknowledgment result to aborted, and its reason
  to spare mate elevate
Skip forward to [endreal]
Did the invocation fail?
Set acknowledgment result to failure, and its reason
  to re-validate
----[  endreal  ]-----
Acknowledge input request
Tell CLIENT that the switch realization is about to
  end
End
```

Figure 5. Example algorithm in textual review format

The design of the unit attribute language was rather straightforward. Each parameter of variation from the domain analysis was mapped to an expression. A few additional expressions were needed to complete the language. Each of these expressions was then translated into a C data structure.

Most of the unit attribute information could be represented in a graph, which was better drawn than specified in a textual language. Hence, a GUI tool was designed to draw the graph. The team first tried to develop the tool using a technology invented at Bell Labs [Koutsofios 93], but found it difficult to learn and use. They switched to Tcl/Tk [Ousterhout 94], which was well supported by tutorials, books, and examples. Within a few months they had a GUI tool that allowed users to draw unit attribute graphs. The graph was translated into the textual language for subsequent translation into C.

Figure 6 shows some fragments of the unit attribute language for a simple example. Two attributes of the `Unit1` type are shown: the type has no inhibit condition, and no routine maintenance is performed. A parent-child link between units `Unit1-0` and `Unit2-0` is also shown. The notation "`(Unit1 | 0)`" represents the name "`Unit1-0`". Although the parent-child link is described twice in this textual language, only one line is drawn between the two graphical objects when using the GUI tool.

```

;unit.class(Unit1)
    ;inhibit.cond[no]
    ;routine.maint[no]
;unit(Unit1 | 0)
    ;children
        ;child(Unit2 | 0)
;unit(Unit2 | 0)
    ;parents
        ;parent(Unit1 | 0)

```

Figure 6. Fragments from unit attribute language

The design and implementation of the application engineering environment took two years. (This includes the time spent in verification and validation, which was done incrementally.) Three members of the team who stayed on from the domain analysis phase did the work. In addition, several other domain experts and consultants with other skills helped for short periods. For each of the members of the team this was a part-time activity, though a high-priority one. Total effort expended was about two staff-years.

5. DEPLOYMENT

Even if you build it, they may not come. That is, the existence of an innovation is no guarantee that it will be adopted. For the DECC team the challenge was to convince other engineers in their community to use the technology the team had developed. In fact, technology transfer was a concern of the DECC team throughout the project. As described in [Ardis 98], adoption of domain engineering at Lucent often follows the innovation diffusion model of Everett Rogers [Rogers 95]. For example, minimizing the complexity of an innovation seems to increase the likelihood of adoption.

The DECC team took several steps to minimize the complexity of their technology and to ensure its compatibility with the user population. They created an easy-to-use GUI tool for specifying unit attributes, they reused languages (such as VFSM) that were already accepted by the user community, and they gave several demonstrations and tutorials. All of these efforts helped to make the DECC technology attractive and familiar to new users, but such efforts were still not enough to convince a new project to commit to their use.

In the summer of 1996 the DECC team found a project that would cooperate in the trial use of the DECC technology. The project was already too far along to switch over to the DECC technology, but they were willing to share their requirements and tests for a side-by-side comparison. The team used the DECC technology to generate software equivalent in functionality to software developed by the cooperating project. They then performed a static analysis and dynamic analysis to show that the two versions had comparable run-time performances.

This was an important demonstration of the feasibility and practicality of the DECC technology. Additionally, it provided useful feedback to the domain engineering cycle. After conducting the pilot study the DECC team revised the commonality analysis document and updated the tools. The total effort spent in performing the pilot project, revising the analysis, and updating the tools was about one staff-year.

The first project to adopt DECC appeared a few months later. Again, the DECC team revised the commonality analysis document in light of feedback from this application, and made further modifications to the tools. The successful completion of this project has helped to convince other projects to adopt the DECC technology.

6. LESSONS LEARNED

We have learned several valuable lessons from this effort that may have broad applicability. Domain engineering occurred in three phases: discovery, design, and deployment. For each phase we describe the lessons learned.

6.1 Discovery Lessons

Most of the discovery phase was spent on investigating and documenting the behavior of a legacy system. This involved domain experts both on and outside the team. The important lessons of the discovery phase are:

- Re-engineering should be done neither too quickly nor too slowly.
- Different domain knowledge requires different representations.
- Collect a good set of examples to spur and test abstractions.

Re-engineering is best done as a part-time activity. This allows the team members to maintain their contact with their organizations, and it provides time between meetings to investigate issues off-line. When possible, use these opportunities to approach domain experts outside the team. This brings more expertise to bear to the problem, and it also facilitates later adoption of the resulting solutions. On the other hand, re-engineering should not be done too slowly. If too much time elapses some discoveries and decisions may be forgotten, leading to their re-investigation and re-resolution. Finally, there is a long list of potential risks to projects that take too long: reorganization, personnel changes, mission changes, and budget changes all undermine projects that last more than a few months.

When performing domain analysis use different representations, including multiple languages, to model different aspects of the domain. That is, use whatever is most natural to the domain experts to express the knowledge. This is just another manifestation of the divide-and-conquer strategy for dealing with complexity. In this case we divided a complex domain into subdomains, made a different language for each subdomain, modeled each subdomain in its language, and then modeled the entire domain by composing the subdomain models. For the configuration control domain the two major subdomains were the unique attributes of units and the algorithms for changing configurations. The interoperability of languages created with InfoWiz made it possible to compose models represented in different languages. We found this strategy of making languages to fit the domain to be more effective than forcing models of different domains to fit one single modeling language.

Although the main purpose of domain analysis is to identify and express abstractions that apply to a family of products, it is important to have good representative examples from the family to test the abstractions. Both the writers and the readers need the examples for that purpose. By reusing the same examples throughout the analysis the team increases consistency and simplifies the work of the reviewers. The DECC team received more reviewer comments about the examples in the commonality analysis document than about the abstractions.

6.2 Design Lessons

During the design phase the team designed and implemented the application engineering environment. This was an exploratory activity where the team experimented with several technologies for making languages and GUI tools. The most important lessons of the design phase are:

- Use technology that domain experts can master quickly.
- Use composable technology whenever possible.

During the design phase the team needed to mitigate the risk of adopting new technology to implement the application engineering environment. Technology that is difficult to learn increases risk. Hard-to-learn technology also aggravates maintenance issues. Domains change continuously in response to market needs, which means that the environment must also change continuously. Hence, a technology that makes domain-specific languages easy for domain experts to make and maintain is crucial.

The team also realized early on that their tools would need to compose with one another. Composability of the languages was crucial. Since the data structures created from the unit attributes would drive the generic algorithms, it was important that these pieces could be combined easily. To accomplish this the team first made sure that the expressions of the languages were independent of one another. Next, they minimized the number of global structures that would need to be shared by using traditional information hiding techniques. Finally, they used the common concrete syntax of InfoWiz for all languages. This facilitated the independent development of the different languages, while also guaranteeing that the models written in those languages would compose easily.

An interesting angle on these two lessons was that the DECC team had to reconsider them from an opposite point of view when creating the DECC technology. That is, the potential users of the technology would be more successful in their initial trials if these lessons were followed. The DECC technology, therefore, strives to be easy to use and composable.

6.3 Deployment Lessons

During the deployment phase the team sought to convince other projects to use their technology. This phase was dominated by a concern with technology transfer issues. The most important lessons of the deployment phase are:

- Give demos often.
- Talk to customers early and often.
- Use shadow projects to demonstrate value and compatibility.

Give demos often. These help to get and keep management support, which is needed to keep resources for the project. These also help convince peers that the technology should be adopted. Put more strongly, demos are concrete indicators of progress, and a project should be managed to produce demos at each milestone. Plan for demos; don't regard them as a distraction.

Customer focus is essential to technology transfer. This should start during analysis and continue through design and deployment. Note that learning takes place in both directions. As the team learns more about the customers' needs they become better able to anticipate the customers' reactions to the technology. At the same time, the customers learn more about the value of the new technology, and so become better able to judge its strengths and weaknesses.

If possible, conduct a shadow project to demonstrate value and collect data. This is very important for selling the technology to potential customers. No one wants to be the first user. It may not be as hard to conduct a shadow project as it first appears. On our project we needed access to requirements and tests. These are normal byproducts of our software development process, so they did not impose an additional burden on the other project. We did need to spend some time discussing issues with the other team, but we also contributed to their effort by providing review support.

7. CONCLUSIONS

The DECC project re-engineered an important software component of a large legacy system. To do this it employed several new technologies, including domain engineering, domain-specific languages, and graphical user interfaces (which were new to the team). Fortunately, the DECC team had the support of other pioneers within Lucent in adopting these new paradigms. But, the most important contribution to the success of the project was the rich domain knowledge of the software development community.

Domain engineering projects may fail for many reasons, but they can never succeed without domain knowledge.

ACKNOWLEDGMENTS

We thank the many domain experts who contributed to the success of this project. In particular, we thank Paul Iverson and Andy Kranenborg for their help during the domain analysis phase. We thank David Weiss for creating and supporting the FAST process. We thank Mehry Moukhtar, Dan Johnson, Michelle Homer, Tom Denton, Lynn Pautler, and Carl Amport for their support and encouragement. Finally, we thank the reviewers for their help in improving the presentation of this material.

REFERENCES

- [Ardis 98] M.A. Ardis and J.A. Green, " Successful Introduction of Domain Engineering into Software Development", *Bell Labs Technical Journal* v 3 n 3, July-September 1998.
- [Boehm 88] B.W. Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Computer* v 21 n 5, May 1988, pp. 61-72.
- [Flora-Holmquist 97] A.R. Flora-Holmquist, E. Morton, J.D. O'Grady, and M.G. Staskauskas, "The Virtual Finite-State Machine Design and Implementation Paradigm", *Bell Labs Technical Journal* v 2 n 1, Winter 1997, pp. 96-113.
- [Koutsoufios 93] E. Koutsoufios, "Editing Graphs with dotty", AT&T Bell Laboratories Technical report, July 1994.
- [Martersteck 85] Martersteck, K.E. and Spencer, A.E., "Introduction to the 5ESS® Switching System", *AT&T Technical Journal* v 64 n 6, pp. 1305-1314, July-August 1985.
- [Nakatani 97] Nakatani, L.H., Jones, M.A., "Jargons and Infocentrism", *Proc. ACM SIGPLAN Workshop on Domain-Specific Languages*, January 1997.
- [Ousterhout 94] J.K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
- [Parnas 76] Parnas, D.L., "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering SE-2*, pp. 1-9, March 1976.
- [Parnas 78] Parnas, D.L., "Designing Software for Ease of Extension and Contraction", *Proceedings of 3rd International Conference on Software Engineering*, May 1978.
- [Parnas 85] Parnas, D.L., Clements, P.C., Weiss, D.M., "The Modular Structure of Complex Systems", *IEEE Transactions on Software Engineering SE-11*, pp. 259-266, March 1985.
- [Parnas 86] Parnas, D.L., Clements, P.C., "A Rational Design Process: How and Why to Fake It", *IEEE Transactions on Software Engineering SE-12*, February 1986.
- [Rogers 95] Rogers, E.M., *Diffusion of Innovations*, Fourth Edition, The Free Press, New York, NY, 1995.
- [Weiss 99] Weiss, D.M., Lai, C.T.R., *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison Wesley, 1999.