

# Software Product Lines: a Case Study

Mark Ardis \*    Nigel Daley<sup>†</sup>    Daniel Hoffman<sup>‡</sup>    Harvey Siy<sup>§</sup>  
David Weiss<sup>¶</sup>

January 28, 2000

## Abstract

A software product line is a family of products that share common features to meet the needs of a market area. Systematic processes have been developed to reduce dramatically the cost of a product line. Such product line engineering processes have proven practical and effective in industrial use, but are not widely understood. The Family-Oriented Abstraction, Specification and Translation (FAST) process has been used successfully at Lucent Technologies in over 25 domains, providing productivity improvements of as much as four to one. In this paper, we show how to use FAST to document precisely the key abstractions in a domain, exploit design patterns in a generic product line architecture, generate documentation and Java code, and automate testing to reduce costs. The paper is based on a detailed case study covering all aspects from domain analysis through testing.

**Keywords:** domain engineering, software product line, commonality, variability, FAST

## 1 Introduction

A software product line is a family of products that share common features to meet the needs of a market area. The creators of the product line use a systematic process that exploits the commonality across the family, while allowing variation among the members. The Family-Oriented Abstraction, Specification and Translation (FAST) process is an example of such a product line development process [1]. With FAST, the common and variable characteristics of a product family are described using a process called *commonality analysis*. Based on the commonality analysis, an *application engineering environment* is developed, to produce family members as quickly and cheaply as possible. Architectural frameworks and domain-specific languages are often employed in an application engineering environment.

We have successfully employed the FAST process on several projects at Lucent [2]. The main benefits were reduction in development interval, greater consistency of behavior

---

\*Bell Laboratories, Lucent Technologies

<sup>†</sup>University of Victoria, Department of Computer Science

<sup>‡</sup>University of Victoria, Department of Computer Science

<sup>§</sup>Bell Laboratories, Lucent Technologies

<sup>¶</sup>Bell Laboratories, Lucent Technologies

			Points:	10	30	[ ]	[ ]
			Percentage:	5.0	10.0	[ ]	[ ]
			Assignment name:	quiz	Program	[ ]	[ ]
Last name	First name	Number					
Hoffman	Dan	33024	7.0	22.5	[ ]	[ ]	
Strooper	Paul	11576	9.0	27.0	[ ]	[ ]	
[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]

Figure 1: Sample Header Table application

across multiple products, reduced maintenance cost, and improved training of new employees within the domains where it was used. Many projects have experienced an interval reduction of four to one, with less investment than would be required to produce three family members by other means.

This paper presents a case study initiated at Bell Labs to gain a better understanding of the relationships between FAST and other technologies, including object oriented programming, design patterns, and automated testing. The case study is based on *Header Table*, an example product line chosen to be simple enough for full implementation but complex enough to raise significant design and implementation issues. The Header Table product line provides a family of editors for tabular data. Every family member provides access to three rectangular matrices: column header, row header, and core. For example, the *Grades* family member in Figure 1 maintains student grades. The *column header* matrix has three rows and four columns. Each column contains the points, percentage, and name for the assignment whose scores appear directly below. The *row header* matrix has five rows and three columns. Each row contains the last name, first name, and number for the student whose scores appear immediately to the right. The *core* matrix contains the scores, one for each student/assignment pair. Every family member provides windowing, with vertical and horizontal scrolling. When the window is scrolled vertically, the row header and core matrices are affected while the column header matrix is unchanged. Similarly, when the window is scrolled horizontally, the column header and core matrices are affected while the row header matrix is unchanged. Commands are provided to insert and remove table rows and columns. As with scrolling, the three matrices are linked in pairs. For example, inserting a table row generates a new row in the row header and core matrices, while leaving the column header matrix unchanged.

The Header Table product line was developed as a vehicle for research rather than as a commercial enterprise. Nonetheless, a complete implementation was built including

documentation, code generation, and automated testing. The Grades family member is in regular use at the University of Victoria. Further, the three-matrix scheme described above is supported by commercial products such as the `JTable` class in the Java libraries and the Excel spreadsheet product (in "freeze panes" mode).

## 2 Related work

Our work in domain engineering focuses on documenting the *commonalities* and *variabilities* in a software product line, and on methods to use them for fast generation of individual products [1, 3]. These methods are based on earlier work. Parnas's information hiding principle encodes commonalities as a module's interface and variabilities as a module's secret [4, 5]. Dijkstra recognized a key relationship between design decisions and program families; each alternative corresponds to a new family member [6].

Concepts of commonality and variability are now appearing widely in the domain engineering literature, although sometimes implicitly. Schmid [7] presents the design of a family of software controllers for automated machining. The design focuses on "hot spots" (variabilities) and makes use of design patterns to encapsulate them. Lam [8] describes a process for variability analysis based on variability templates and a variability hierarchy. This process was used to develop an initial architecture for a family of software controllers for jet engines. Abstract interfaces encapsulated the variabilities, just as abstract interfaces to printer drivers are commonly used to encapsulate the differences between printers. Meekel [9] describes the design of the FLEX kernel, a real-time operating system for a wireless-pager product family. Variabilities were classified as feature, device, and performance variabilities, and techniques were developed to handle each type. Work by Batory in domain-specific architectures [10] and by Tracz [11] in software development processes explicitly uses commonalities and variabilities. Keepence and Manion [12] show how to use design patterns to model *discriminants*, a kind of variability resulting from the presence or absence of features. Gomaa's Evolutionary Domain Life Cycle (EDLC) work [13] shares some of the same objectives of managing software development around a family of products. EDLC emphasizes the selection and analysis of features and feature-sets, which we capture in variabilities. Similarly, Kang's Feature Oriented Domain Analysis (FODA) method [14] emphasizes the selection and analysis of features in order to create an instance of a product family.

While domain engineering can significantly reduce the cost of generating family members, testing remains a significant problem. This paper describes a testing approach based on modeling each test case with an integer tuple and then generating large numbers of tuples to test many interesting combinations of input values. Another approach to tuple testing generates tuples with variants of cartesian product that focus on boundary values [15]; another uses ideas from experimental design to select subsets with all pairs (or all triples, etc.) of values [16]. We use a safety invariant as a test oracle, similar to the well-known technique of embedding assertions in the code under test [17]. Voas describes other approaches to testing for safety properties [18].

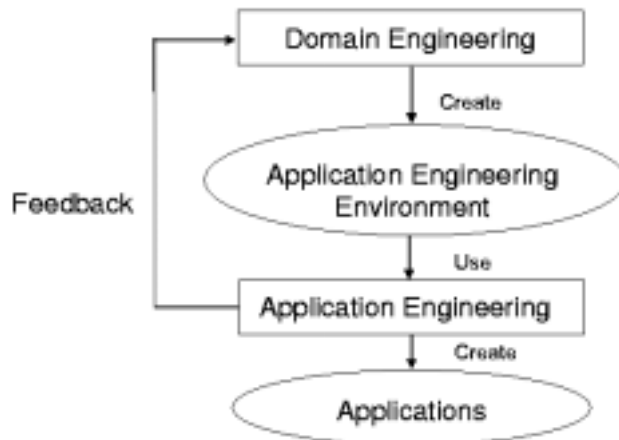


Figure 2: FAST process of domain engineering

### 3 The FAST process

The Family-Oriented Abstraction, Specification, and Translation (FAST) process [1, 3] is an alternative to the traditional software development process. It is applicable wherever an organization creates multiple versions of a product that share significant common attributes, such as common behavior, common interfaces, or common code. Rather than maintain duplicate versions of those commonalities, FAST assumes it is better to have only one instance.

The FAST process is split into two phases: domain engineering and application engineering. (See Figure 2.) The domain engineering step represents an investment, where effort is spent understanding how a family of products share common attributes but remain distinct. This understanding is translated into technology, such as a common set of subroutines or a domain-specific language. We call this technology an *Application Engineering Environment*. Application engineering uses that environment to produce individual members of the product family. The investment in the environment results in an efficient process, so that many family members may be produced quickly and cheaply. Feedback from use of the environment suggests modifications to that environment, which are made after considering the impact on the original domain analysis effort.

The economic justification for domain engineering is straight-forward: If the marginal cost to produce new family members is smaller with domain engineering than without it, then the original investment will be recouped after producing a few new family members. (See Figure 3.) In our experience the reduction in marginal cost is on the order of four to one, and the initial investment is usually less than the cost of producing three new family members.

The first stage of domain engineering is domain analysis, where experts collect and document their knowledge of the product family. A preferred analysis method of FAST is called *commonality analysis*. It consists of a structured social process to enumerate

### Cumulative Cost of Producing Family Members

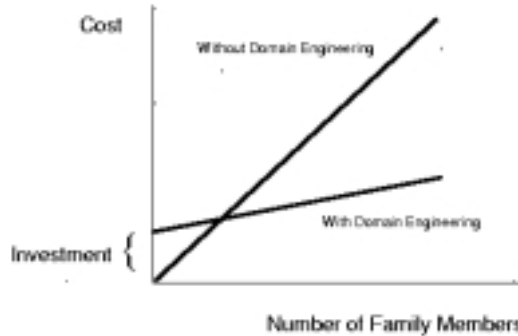


Figure 3: Economic model for domain engineering

the commonalities and variabilities of the product family in a primarily natural language document. Formalisms are often used for key areas of the analysis, but the overall style and tone of the document is narrative text. The document also records the terminology of the domain, the scope of the analysis effort, and issues (with their resolutions) that occurred during the analysis.

During commonality analysis it is common practice to collect and use example scenarios to explore differences between family members. For this project we distinguished between *usability scenarios*, which describe the actions required to perform common user operations, and *variability scenarios*, which emphasize the differences between individual products. These two classes of scenarios are similar to *direct scenarios*, and *indirect scenarios*, as described in [19].

The second stage of domain engineering is to translate the knowledge gained from the analysis stage into useful technology. There are several ways to do this, but most of our experience has been with two methods: creation of domain-specific languages and creation of composable components. In the first method a small language is created that allows the user to specify the values of the variabilities between family members. Specifications in that language are then translated into code. In the second method a library of components is constructed. An overall framework that uses those components is designed, and a composer is constructed so that individual products can be created by composing appropriate components within the framework.

The switching systems area of Lucent uses domain engineering to improve efficiency and maintain high levels of reliability in their products. Large systems, such as the 5ESS(RM), are developed from many smaller subsystems that each offer their own opportunities for optimization. Domain engineering has been applied to subsystems that produce customized forms for entering and changing data, that produce billing records for customers, that maintain the status of individual components of the switch, and that monitor and adjust performance characteristics of the switch. Many of these projects have created domain-

**Overview:** briefly describes the domain and its relation to other domains

**Definitions:** provides a standard set of technical terms

**Commonalities:** a structured list of assumptions that are true of every member of the family

**Variabilities:** a structured list of assumptions about how family members differ

**Parameters of Variation:** a list of parameters that refine the variabilities, adding a value range and binding time for each

**Issues:** record of important decisions and alternatives

**Scenarios:** examples used in describing commonalities and variabilities

Figure 4: Major sections of commonality analysis document

specific languages, usually with a graphical interface for specification of products. Productivity of the groups that have used domain engineering has increased by a factor of 3 to 5, with quality of the resulting code exceeding previous versions.

## 4 Commonality analysis

Figure 4 shows the structure of the commonality analysis document produced by this project, which consists of 7 major sections. The early sections sketch out the domain, identifying major issues and design concerns. Later sections delve into more detail as they describe differences among individual family members. The document was produced in a series of moderated meetings among the domain experts.

The Overview section describes the scope of the effort. It gives a high-level view of the product family and its relationship to other domains. This is similar to a context diagram in structured analysis.

The Definitions section records technical terms and their meanings. Whenever these terms are used in the rest of the document they are italicized, to remind the reader to use the special technical definition. For example, an important term for this family is *header table*, which is defined as follows:

A *header table* stores 3 closely-linked matrices: *core matrix*, *column-header matrix*, *row-header matrix*. The cells in each matrix are addressed relative to (0,0), with the origin in the upper-left corner. The first coordinate is the horizontal displacement and the second is the vertical displacement within the matrix.

A *window table* is then defined as a special case of a *header table*.

A *window table* is a *header table* containing the data that are currently visible.

Thus, a *window table* inherits all the properties of a *header table*, but it may be restricted in size.

The Commonalities section describes those assumptions that are true for all members of the family. This section is often subdivided into subsections that address different classes of attributes. In our case the subsections are: Static Presentation, Cursor/Window Repositioning, Cell Validation Policy, and Computation/Reporting. An example commonality from the Cursor/Window Repositioning subsection is:

The *column-header matrix* of the *window table* always has the same horizontal displacement as its *core matrix*.

The Variabilities section describes how individual family members differ. This section is subdivided into subsections in the same way as the commonalities. An example variability from the Cursor/Window Repositioning subsection is:

When repositioning the cursor and the window after an operation the amount of movement (the *scroll delta*) may vary. For example, they might be moved as little as possible, as much as possible, or halfway between these extremes.

The Parameters of Variation section further refines the variabilities, specifying legal values, possible default values, and binding time for each. Binding time is the phase of development when a parameter's value is fixed. During the analysis the team determines the earliest opportunity to bind each parameter. Common values for binding time are: domain analysis time, application engineering time, compile time, or run time. An example parameter is:

Window cursor position:  $\langle m, x, y \rangle$  where  $m$  is one of  $\{\text{null, core, row header, column header}\}$ ,  $x$  and  $y$  are the offset into the window table. The binding time is run time and the default is  $\langle \text{null}, 0, 0 \rangle$ .

The Issues section records significant problems that were addressed by the team, alternative resolutions and discussion of those alternatives, and the final resolution. This section provides an historical window into the commonality analysis process itself, as it records disagreements as well as final solutions. An example issue is:

Issue: Should cell validation policy *granularity* and *error action* be allowed to vary

- a) cell by cell,
- b) bound to the *type*, or
- c) uniform across all cells?

Resolution: b) because it seems most adequate—a) is too much freedom and c) too restrictive.

The Scenarios section collects common examples to use in exploring commonalities and variabilities. As mentioned earlier, we collected two types of scenarios: usability scenarios that demonstrate intended use of the products, and variability scenarios that distinguish

between individual products. Within the variability scenarios we collected scenarios for two kinds of behavior: cursor window repositioning and cell validation policies.

An example usability scenario is:

Start a new term—add several new rows (students)

This scenario describes the sequence of actions that a user might perform at the beginning of a new academic term when one needs to add a list of students for a course, but there are no grades yet. This sequence should be easy to perform, and the behavior of the product should be consistent over the sequence. For example, the steps required to add each row should be the same no matter what row is being added.

An example variability scenario is:

Scenario to illustrate scroll delta: Normal case

- State: 4 rows, 1 column, 2 row headers, 3 column headers, window starts at first column and last row, cursor in the core table
- Parameter values: scroll delta can be any of {min, half, max}
- Event: remove row
- Behavior:
  - When the scroll delta is min, then the window is only scrolled one line backward.
  - When the scroll delta is half, then the window is scrolled backward by half a window’s height. I.e., the new last line will be halfway down the window.
  - When the scroll delta is max, then the window is scrolled backward by a full window’s height. I.e., the new last line will be at the bottom of the window.

Note that this example describes several possible behaviors. Each one corresponds to a different choice of scroll delta value. This scenario describes how different scroll delta values yield different behavior in the product. It is useful in exploring the consequences of choosing a particular cursor/window repositioning policy.

Three people participated in the commonality analysis for this case study. They held 15 meetings, most of two hours in duration, over a 3 month period. The total effort expended in meetings was 90 staff-hours.

Figure 5 shows the progress of the commonality analysis by plotting the size of each section as a function of the number of meetings held. For example, after 7 meetings 20 commonalities had been identified. The bulk of the work was finished by the end of the 11th meeting. The pattern of incremental progress across all of the sections of the commonality analysis is typical, though it is unusual to see such an early start on parameters of variation. Most projects wait until they are confident that the variabilities are stable before starting on parameters. In most other ways this project was about half the size of “typical” industrial domain engineering projects. That is, most industrial projects

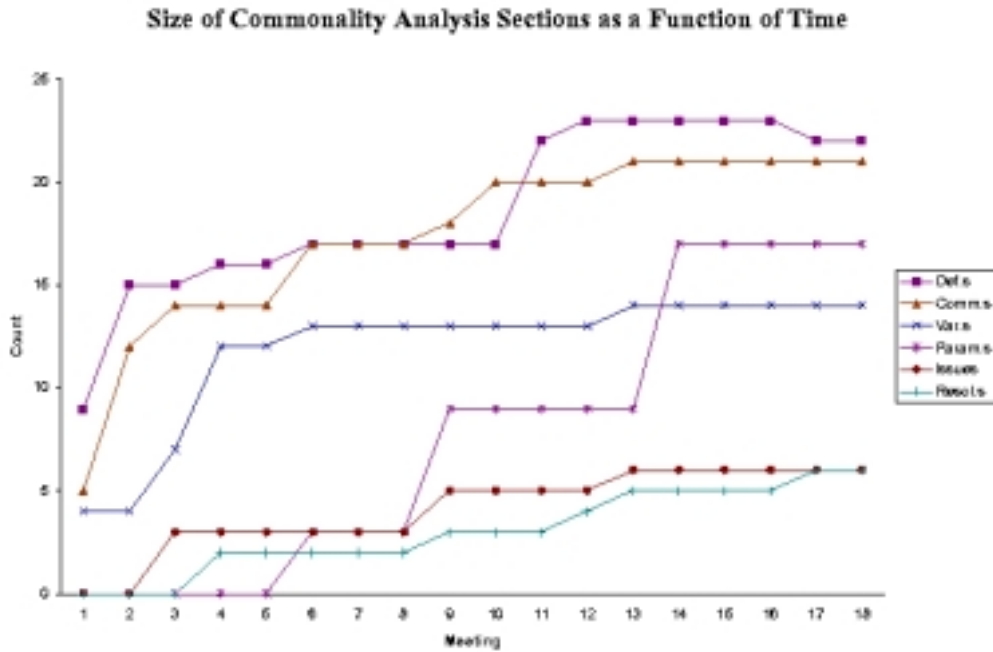


Figure 5: Progress of commonality analysis

take twice as long, with twice as many people. The resulting documents are about twice as long, with twice as many terms, commonalities, variabilities, etc.

The scope of the project was chosen to maximize research benefits, rather than to increase potential sales of family members. An industrial project would probably spend more time wrestling with issues of scope, which would also prolong the commonality analysis.

Another difference between this project and those we have observed at Lucent is in the cost of creating the application engineering environment. It was relatively easy to construct a generator of members of the header table family, so the expended effort was only about one eighth of a typical project. Industrial projects often encounter legacy issues, training issues, and other “real world” problems that we chose to ignore.

## 5 Prototyping cursor/window repositioning

Some topics are difficult to discuss without graphical aids. Most commonality analysis documents include diagrams or pictures. One of the more difficult concepts to describe in this domain is the sequence of events that change the locations of the windows and the cursor. For example, when the user deletes a row the cursor must move to a new row, and the windows may need to move. We called this behavior *cursor/window repositioning*. At first we drew pictures by hand to describe different possible sequences. It quickly became apparent that a prototype tool would be a better way to explore this topic. So, we built a simple prototype that demonstrates the possible alternative sequences.

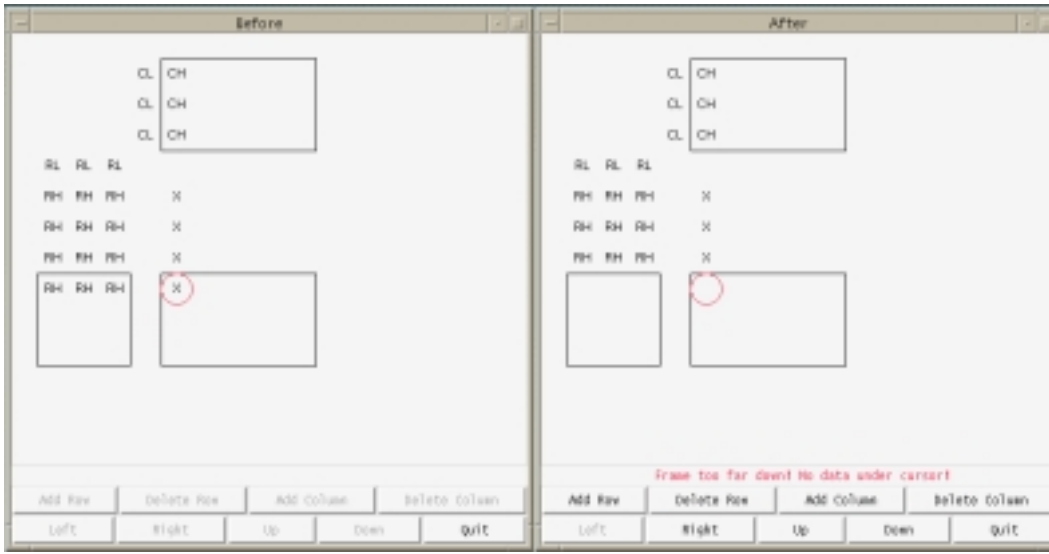


Figure 6: Before and after views of the “delete-row” operation as demonstrated by the cursor/window position simulator.

When the prototype is started it displays the underlying root windows with dummy values provided. Rectangles represent the displayed area of the root windows, and a circle denotes the cursor position. (See Figure 6.) The user may reposition the cursor by clicking with the mouse. Adding and deleting of rows and columns are accomplished by clicking on the buttons at the bottom of the window. Similarly, moving the display windows up, down, left, or right is accomplished by clicking on buttons.

If the result of an operation leaves the cursor outside of the displayed area, an error message is displayed. Similarly, an error is reported if the displayed area moves outside of the data. The prototype provides a “before” and “after” picture of each operation, so that the domain engineer can see how an operation might behave. The left side of Figure 6 shows the state of the system when there are 4 rows and 1 column of data, the displayed windows are as far down as they can go, and the cursor is on the only data visible in the core matrix of the base table. The right side of Figure 6 shows the result of removing the last row of data. Two errors are reported: the displayed windows are outside of the data, and the cursor is not positioned over any data.

A member of this product family needs to implement a cursor/window repositioning policy to establish the correct positions for different situations. As seen earlier in the example variability about scroll delta, one possible policy for movement is to move as little as possible, or one row up from the deleted row. Another policy is to move as far as possible, so that the last row of the data is the last row in the window. Another policy is to move halfway between these extremes. Since the prototype does not implement any of these policies, it can be used to experiment with any of them (manually). Such experimentation often reveals interesting special cases, such as where to put the cursor when the last row in the row header matrix is deleted.

The prototype is a simplified version of a member of this product family. It does not

allow editing of table values, but only allows manipulation of the shape of the configuration. It does not implement any cursor/window repositioning policy, but allows manual simulation of any of those policies. These restrictions made it easier for us to experiment with policies, and they ensured that the prototype would not need updates as our understanding of the policies evolved. Using the prototype, we were able to explore several interesting scenarios that exposed differences between members of the product family. The investment in creating the prototype was minimal: one person spent about 20 hours developing it in Tcl/Tk [20] over a period of a week. The resulting program has 19 functions and about 550 non-commentary source lines.

## 6 Generation from a domain-specific language

In the Header Table family, the cursor/window repositioning (CWR) policies are handled by first modeling the commonalities and then creating a domain-specific language to express the variabilities.

### 6.1 Modeling cursor/window repositioning policies

We model a CWR policy as a pair of functions: one for window repositioning and one for cursor repositioning. The function prototypes are defined in terms of the following fields:

1. *Table shape.* For CWR purposes, the base and window table contents are irrelevant, but the table shape is central. The table shape is a four-tuple:

$\langle \text{numColHdrs}, \text{numRowHdrs}, \text{numCols}, \text{numRows} \rangle$

The values for numCols and numRows vary at runtime, as rows and columns are inserted and deleted. The numColHdrs and numRowHdrs values, however, are fixed for each family member. In Figure 1, for example, the table shape is  $\langle 3, 3, 2, 2 \rangle$ . (Note: We ignore the empty cells shown in the figure. The shape is determined by non-empty cells.)

2. *Window position.* The window position is of the form  $\langle x, y \rangle$  where x and y are the horizontal and vertical displacements of the window table from the upper left corner of the underlying table.
3. *Cursor position.* The cursor position is of the form  $\langle \text{type}, x, y \rangle$ . Here type indicates the matrix—COLHDR, ROWHDR, CORE, or NULL—with NULL used for the special case where the base table is empty, and where x and y are the horizontal and vertical displacements of the cursor from the upper left corner of the window table matrix indicated by type.
4. *User events.* These include all of the mouse, menu, and keystroke events that can impact the cursor or window position, such as row insertion, row deletion, and window scrolling.

The function prototypes are

`nextCursorPosition`:  $S \times W \times C \times E \rightarrow C$   
`nextWindowPosition`:  $S \times W \times C \times E \rightarrow W$

where  $S$ ,  $W$ ,  $C$  are the sets of all legal table shapes, window positions, and cursor positions, and  $E$  is the set of all user events.

Figure 7(a) shows a tabular specification for the remove row event for one CWR policy. There are three logical conditions in the middle column of the table. Consider the case where the first condition is true, the second false, and the third true. In this case, the cursor is on the last row of the base table (condition one true), the last remaining row of the base table is not being deleted (condition two false), and the cursor is on the first (0th) row of the window (condition three true). Because the event removes the last visible row in the window, the result, shown in column three, moves the window up one row. The Java code in Figure 7(b) implements the policy shown in Figure 7(a). We design and review policies using the form shown in Figure 7(a) but also need the Java code. Because it is difficult to maintain these two descriptions consistently, we use a domain-specific language to generate both descriptions from a single source.

## 6.2 Generating CWR documentation and code

The domain-specific language illustrated in Figure 7(c) uses the InfoWiz language generation system [21]. InfoWiz provides generic language support for *jargons*: languages based on terms of the form `;term(argument | argument | ... )`. In a jargon, indentation is significant, as the principal way to indicate grouping. In Figure 7, the `if`, `else`, and `result` terms are specific to the jargon and are used as labels in the abstract syntax tree generated by the InfoWiz system. The concrete syntax is crude, but it allows useful jargons to be created quickly. We have written a generator that produces two files. From a file containing a CWR specification in the form shown in Figure 7(c), we generate the tabular documentation in Figure 7(a) and the Java code in Figure 7(b). For the CWR policy discussed in this section, the complete specification is 224 lines in length; the generated implementation is 259 lines of Java code and the documentation is 271 lines of HTML. To date, we have developed four CWR policies, each of roughly the same size.

## 6.3 Discussion

Once the decision to generate is made, lots of small improvements are available. Effective tabular documentation must be compact. While abbreviations are often essential in compact documentation, in executable code long names are preferred and class name prefixes are often required. It is easy to make the generator perform code expansions, e.g., replacing `removeRow` in Figure 7(a) with `Event.removeRow` in Figure 7(b). The expansions are trivial, but can be essential to having usable documentation and compilable code generated from the same source.

This section illustrates two key design decisions:

1. *Have a single point of truth, i.e., generate documentation and code from a single representation.* Generation makes it easy to keep the documentation and code consistent, and to make the documentation compact and readable. Even a simple language and generator can provide significant benefits: a little formalization goes a long way.

Event	Condition	nextWindowPosition(s,w,c)
removeRow	if $w.y + c.y = s.numRows - 1$ if $s.numRows = 1$ else if $c.y = 0$ else else	$\langle w.x, 0 \rangle$ $\langle w.x, w.y - 1 \rangle$ $\langle w.x, w.y \rangle$ $\langle w.x, w.y \rangle$

(a) Tabular specification

```

public static WindowPosition nextWindowPosition
    (int event, TableShape s, WindowPosition w, FieldId c) {
switch (event) {
case Event.removeRow:
    if (w.y + c.y == s.numRows - 1) {
        if (s.numRows == 1) {
            return new WindowPosition(w.x,0);
        } else {
            if (c.y = 0) {
                return new WindowPosition(w.x,w.y-1);
            } else {
                return new WindowPosition(w.x,w.y);
            }
        }
    } else {
        return new WindowPosition(w.x,w.y);
    }
}
// ...

```

(b) Java implementation

```

;event(removeRow)
;if(w.y + c.y = numRows-1)
;if(s.numRows = 1)
;result(w.x | 0)
;else
;if(c.y = 0)
;result(w.x | w.y-1)
;else
;result(w.x | w.y)
;else
;result(w.x | w.y)

```

(c) Domain-specific language specification

Figure 7: Documentation, implementation, and DSL for remove row event

2. *Implement the CWR policy as a pair of functions.* We put all the CWR code in one module, rather than scattering the code across commands, as is usually done. This approach provides big advantages for documentation, generation, ease of change, and testing, as shown in the section on automated testing.

## 7 Application engineering environment

In creating and using the Application Engineering Environment (AEE), three roles become important:

1. the *Domain Analyst* writes the commonality analysis document
2. the *Domain Implementor* develops the Application Engineering Environment based on the commonality analysis document
3. the *Application Engineer* uses the Application Engineering Environment to generate new family members

The Domain Implementor must make a number of important decisions before creating the Application Engineering Environment. First, he addresses the broad issue of *how* the Application Engineer will generate a new family member. This requires envisioning the concrete tasks the Application Engineer will perform during the generation process.

For each parameter of variation the Domain Implementor must decide whether or not to provide the Application Engineer with automated, e.g., “point-and-click,” support for selecting the parameter value. For the parameter selections that are not automated, the Application Engineer will have to modify the implementation manually. For different levels of automation, configuration files, forms or “wizards” can aid the Application Engineer in selecting parameter values, e.g., by offering only legal values and combinations of values.

In implementing these automation decisions, the Domain Implementor has available the classic approaches of polymorphism, automated code replacement, command line parameters, function parameters, and configuration file changes. Carefully structured commonality analysis allows these standard programming techniques to be effective in providing automated support for code generation from parameter of variation values. Even so, there are still important tradeoffs to consider before supporting automated parameter selection. Factors such as the cost of implementing the Application Engineering Environment, the capabilities of the Application Engineer, and the expected number of family members all play vital roles in the decision to support automated parameter selection:

- Parameter point-and-click support has significant initial costs in building the form or wizard, with potential further costs in later modifying it to incorporate additions or deletions in the parameter value space. These high costs, however, can be offset by the relatively low costs of generating new family members and the small amount of system knowledge required by the Application Engineer. This approach becomes more profitable when the relative cost per family member is low. One such situation is when there are many family members.

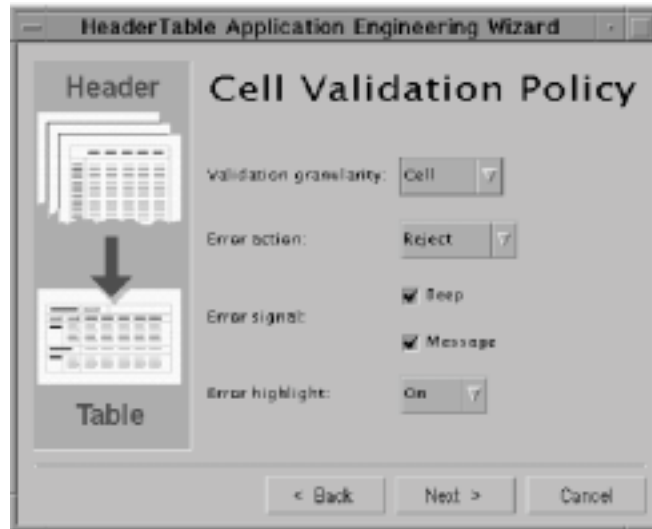


Figure 8: Cell Validation Policy selection pane

- Support for configuration file parameter changes incurs lower initial implementation costs but are harder to use, especially for non-technical Application Engineers. Like point-and-click support, configuration files can reduce the cost of family member generation.
- Manual coding changes have no built-in Application Engineering Environment implementation costs but result in significantly higher generation costs and require the Application Engineer to have special programming knowledge. Nevertheless, with the system structured as described in the next section, manual changes can still be made easily, provided the Application Engineer has the necessary skills.

These tradeoffs can be analyzed by constructing an economic model of the sort shown in Figure 3. The Header Table AEE provides a “wizard” for entry of parameter of variation values. The wizard comprises eight panes that collect values for twenty parameters of variation. The Application Engineer selects some parameter values from list boxes and check boxes. Others are entered as free form text, which then are validated by the wizard. For example, Figure 8 shows the cell validation policy pane of the wizard, which allows the Application Engineer to specify values for the following parameters: the validation granularity, the action to take when an invalid cell entry occurs, the type of signal to warn the user of the invalid entry, and whether or not to highlight the invalid cell. For each parameter, the wizard offers only the legal values. Some value combinations, however, are prohibited. For example, if error highlighting and both error signals are turned off, then the error action cannot be “reject”. With that choice, an erroneous user action would be rejected with no feedback whatsoever.

Manual coding changes are necessary if the Application Engineer needs to modify variabilities not specified in the commonality analysis document. Examples of this type of variability in the Header Table domain include the background color of the application

and the width of each cell.

The final pane of the wizard displays a summary of the chosen values and allows the Application Engineer to generate the new family member. The generation process involves copying the base directory (code common to all family members) to the target directory. Blocks of code are then added to source files to set the requested cell validation policy and various Header Table parameters. Finally, the appropriate implementation for the requested cursor/window repositioning policy is copied to the target directory.

## 8 A generic architecture

The previous section described the Application Engineering Environment from the application engineer's perspective. This section presents the underlying architecture, focusing on three aspects: module structure, internal scenarios, and design patterns.

### 8.1 Module structure

Following Parnas [22], a *module* is a programming work assignment, and the focus of efforts for design for change. The work assignment is to design and implement an encapsulation of a design decision, hiding the implementation of the decision while making available to others the services the module provides. While modules and classes are closely related, it is important not to insist that each module be implemented as a single class and vice versa [23]. Most modules are implemented as a few closely related classes; sometimes a class contains code from more than one module.

We specify the module structure in a document called a *Module Guide*. The Module Guide serves two main purposes:

1. *Provide an annotated inventory of the source code.* The Module Guide is typically short and easy to maintain, and very useful when planning system changes. For systems with many modules, the Module Guide is usually hierarchically structured [24].
2. *Tell the Application Engineer which source code files to change when handling a variability not supported automatically by the AEE.* For example, the Header Table AEE supports changes in the number of row headers and the row header labels automatically: the Application Engineer enters values and the AEE generates the code. Changing the row label font or color, however, requires a manual change to the source code. The Module Guide says that the Window Format module is the place to make the change.

The Header Table Module Guide contains two sections. The module summary section lists the names of the modules and the names of the public classes associated with each module. The second section contains two entries for each module:

1. *Service* : a brief description of the functionality supported by the module.
2. *Secret* : the expected change, i.e., the design decision encapsulated by the module [4].

- *Classes.* HeaderTable, FieldId, TableShape, HeaderTableEnumeration
- *Service.* The Header Table module offers basic header table services. The HeaderTable class provides a header table with element type Object. Rows and columns may be inserted and deleted; each table element is accessible using an index of type FieldId. The TableShape class maintains an integer four-tuple, storing the number of rows, columns, row headers, and column headers in the table. For a given TableShape, the HeaderTableEnumeration class provides sequential access to the set of all FieldIds in a table with that shape.
- *Secret.* The algorithms and data structures used to implement the module, especially the HeaderTable class, and the ordering of the FieldIds returned by the HeaderTableEnumeration class.

Figure 9: Module Guide entry for the Header Table module

The Header Table system has 12 modules and 36 public classes, with 3,769 lines of code in total. The modules have from one to eight public classes each; no class contains code from two modules. Thus, the module structure partitions the source code files: every file is contained in exactly one module. Figure 9 shows the Module Guide information for the Header Table module.

## 8.2 Internal scenarios

As described earlier, the Commonality Analysis document describes “external” scenarios: sequences of user events, such as keystrokes and mouse clicks, in a context. For a given architecture, each external scenario gives rise to an internal scenario: the internal events generated in response to the user events in the external scenario. These internal scenarios are invaluable for design walkthroughs. The main strength of a generic architecture is abstractness: the same architecture can be applied to many applications. Abstract architectures can be hard to reason about, however; even glaring mistakes are often overlooked. Internal scenarios provide concrete examples to exercise abstract generic architectures and frequently reveal errors saving costly debugging and rework. We illustrate these ideas with two cell-validation scenarios. As shown in Figure 8, the Cell Validation Policy allows variabilities in four areas:

1. *Validation granularity: cell or form.* With cell granularity, validation occurs whenever a cell loses the input focus. With form granularity, validation occurs only when the current table is saved to disk.
2. *Error action: accept or reject.* When validation fails, accept or reject the action that triggered the validation.
3. *Error signal: message box (on or off) and beep (on or off).* When validation fails, issue a message box, a beep, or neither.
4. *Error highlighting: on or off.* When validation fails, highlight the invalid cell or not.

<b>External events</b>	<b>Internal events</b>
user clicks in target cell	source cell lost focus handler invoked source cell validated: pass target cell got focus handler invoked set cursorId to target cell
target cell receives input focus	

(a) cell validation on lost focus: validation passes

<b>External events</b>	<b>Internal events</b>
user clicks in target cell	source cell lost focus handler invoked source cell validated: fail request focus issued: source cell message box creation requested target cell got focus handler invoked target cell lost focus handler invoked source cell got focus handler invoked source cell lost focus handler invoked
message box appears	message box got focus handler invoked
user clicks 'ok' in message box	message box lost focus handler invoked source cell got focus handler invoked
source cell retains focus	

(b) cell validation on lost focus: validation fails

Table 1: Internal and external scenarios for cell validation

Suppose that the validation granularity is cell, the error action is reject, the error signal is message box on/beep off, and the error highlighting is off. Table 1 shows what happens when the user changes the focus from a source to a target cell. In the first scenario, the source cell is valid and the internal events are straightforward. In the second scenario, the source cell is invalid, and focus events are numerous. Debugging the cell validation code was quite difficult because of the many variability combinations (32 in all) and because focus handling behavior differs across JDK versions and target platforms. While the external scenarios in Table 1 are simple, they were quite useful. They are sufficient to distinguish all 32 combinations and became the focus of the lengthy debugging sessions required to get the the cell validation code working properly. In each session, the discussion inevitably focussed on the same question: “What are the internal events generated by the external events in Table 1?”

### 8.3 Design patterns

In object-oriented system development, design patterns play a central role. According to Coplien [25], most design patterns focus on variability, making them especially important in product line architectures. The Header Table architecture is based on seven classic design patterns [26]: iterator, adapter, model/view/controller, command, template, strategy, and state machine.

In the Header Table architecture, design patterns are used primarily to factor out commonality and to encapsulate variability. Three examples show this was done:

1. *Adapter.* Each Header Table family member contains two header tables, to store the window and base tables. The Header Table module provides only the services needed by both the window and base tables. Then the adapter pattern is used twice to supply the special services needed by each of the tables. For example, the base table has element type String and has a facility for determining if the table has changed since the last save to disk. The window table has element type TextField and supports a setHighlight method for flagging invalid fields.
2. *Template.* The Command class has abstract methods getId, isLegal, and execute. Each concrete command is a subclass of Command and must implement the three methods. The Command class uses the Template pattern in the implementation of the dispatch method, containing polymorphic calls to getId, isLegal, and execute. The dispatch method is complex and error-prone. By using the Template pattern, it can be implemented once, rather than once per concrete command.
3. *State machine.* As shown in Table 1, even a single mouse click can generate numerous focus events. The sequences of got focus and lost focus events differs, depending on the state of the system when the mouse click occurs. Ad hoc solutions using flags to indicate context are error prone. Much better control is achieved with a state machine.

## 9 Automated testing

Testing a product line presents three significant challenges:

1. There are many special cases, requiring many tests.
2. Because the test budget for each family member will be small the test effort for each family member must be minimized.
3. Graphical user interface code is usually tested using capture/playback tools. Such tools are widely available, allowing a tester to capture keystrokes and mouse clicks, and to replay them at will. The resulting scripts are, however, long and notoriously hard to maintain.

For the Header Table family, we have developed three test strategies suitable for general use in product line testing. All three strategies depend on design for testability, requiring an architecture that minimizes the cost of both the generation and the testing of each family member.

## 9.1 Test common code thoroughly

For code that is shared by all family members, it pays to develop thorough, automated test drivers. In our case study, the Header Table module in Figure 9 falls into this category.

We developed a driver in which each test case is based on a tuple of the form

$$\langle numColHdrs, numRowHdrs, numCols, numRows \rangle$$

Each tuple represents one Header Table shape. Thousands of these tuples were generated. For each tuple, the driver:

1. generates a table with that shape and with unique values in each cell,
2. checks that the table contents are correct,
3. inserts a column before and after every existing column, checking that the table contents are correct after each insertion,
4. removes every existing column, checking that the table contents are correct after each removal, and
5. performs the tests analogous to (3) and (4) for each table row.

All failures found in the Header Table module were revealed with this driver. Tool support made the driver compact and inexpensive to develop [15].

## 9.2 Exploit common aspects of variable code

In a given family member, some code will be specific to that member but still may have characteristics shared across the product line. Such shared characteristics can be used to reduce test costs.

In the Header Table product line, the cursor/window repositioning (CWR) policy varies widely across family members. By focusing on the common aspects of the CWR policies, a single driver can be used to test all CWR implementations. The common aspects are the function prototypes (see "Generation from a domain-specific language") which determine the form of the test inputs, and the invariant described in the next subsection, which serves as test oracle.

The driver generates tuples of the form  $\langle S, W, C \rangle$  where  $S$  is the table shape,  $W$  is the window position, and  $C$  is the cursor position. Thus each tuple has nine fields in all: four for table shape, two for window position, and three for cursor position. The driver implements the pseudocode shown in Figure 10. A large number of tuples are generated by calculating the cartesian product of integer ranges of various sizes for each tuple field. Then, each tuple is checked for validity using the following invariant. All the generated table shapes are valid. The window position must be "in range," e.g., the vertical displacement must be in  $[0..s.r - 1]$  where  $s.r$  is the number of rows in the shape. Similarly, the cursor must be in range. A separate case is required, depending on the cursor type field. For example, if the cursor is in a row header, then its vertical displacement must be in  $[0..(s.r - w.r - 1)]$  where  $s.r$  is the number of rows in the shape and  $w.r$  is the vertical displacement of the view. For each valid tuple, and for each user event:

```

for each tuple  $T = \langle S, W, C \rangle$ 
  if  $T$  is valid
    for each command  $c$ 
      calculate  $S'$  to reflect  $c$ 
      call CWR methods to calculate  $C'$  and  $W'$ 
      check that  $T' = \langle S', C', W' \rangle$  is valid

```

Figure 10: Pseudocode for cursor/window repositioning test driver

1. the shape is adjusted to account for the command, e.g., for a remove row command, the number of rows in the shape tuple is decremented,
2. the CWR code is invoked to calculate a new window and cursor position, and
3. the new tuple is checked for validity.

This driver focuses on safety: ensuring that the CWR code does not put the application in an inconsistent state. Such testing is important because the failures revealed by this driver frequently cause the application to crash.

### 9.3 Utilize scenarios from the commonality analysis

For code in one family member that has little in common with other family members, some custom testing is unavoidable. As in all testing, it pays to be systematic. In the Header Table family, we derive test cases from the scenarios in the commonality analysis document. There are five usability scenarios and 14 variability scenarios; each scenario becomes a test case. The variability scenarios are designed to illustrate the differences between family members. Thus, the test cases will exercise the code specifically written for the family member.

## 10 Discussion

Based on our experience, we have formulated the following principles for product line engineering:

- *Commonality analysis is central.* Successful domain engineering depends on making the right decisions regarding commonalities and variabilities in the product line. The commonality analysis document is a powerful tool for communication between marketing, senior management, and developers. It records the essential facts of the product line vision in a compact form understandable to all parties.
- *Bound the variabilities carefully.* Because variabilities are the main focus of automated software generation, bounding the variabilities is essential. Both the values and the binding times are important. Later binding can be extremely beneficial by giving the user control over product variability. In the extreme case, all parameters

of variation are bound at run time and the shipped product is actually an application engineering environment. Such late binding can make design, coding and testing expensive, can affect performance, and can also make the product difficult to use. Commercial word processors, for example, have so much run-time variability that it has become a burden to most users.

- *Design strongly influences specification.* While marketing concerns usually drive the definition of a product family, implementation concerns play an important role as well. Other things being equal, family members that are easy to generate are preferred. Thus, architectural concerns often influence the family definition, giving rise to Faulk's law: "domain engineering is the process of promoting selected aspects of the design into the requirements" [27].
- *Family member generation: automated or manual?* While the main benefit of an AEE is a result of the automated generation of new family members, it rarely pays to automate the generation process for every variability. Obviously, with automation generation is faster, shrinking labour costs and time to market. Also, an automated AEE may permit the generation to be performed by less skilled staff, e.g., a technician rather than a programmer. Automation, however, makes the AEE more expensive, and inevitably constrains the product family. For example, in the Header Table family, generation for the cell validation policy is completely automated. Generation is fast and can be done without any knowledge of Java. However, the generator screens and underlying code were expensive to develop, and many plausible cell validation policies can not be generated.
- *Product line engineering is more demanding.* Product line engineering forces you to be much more systematic and to produce generic models where an ad hoc solution would have been sufficient in a one-shot application. Because there will never be time to debug (or even build) every possible variant, well designed architectures and components that can be composed with confidence are essential. Otherwise, you will be plagued by "composition bugs": components that work well in one family member but fail in others. For example, in the Header Table family, new GUI paradigms had to be invented for the cell validation and cursor/window repositioning policies. While every feature in these policies is present in well-known commercial products, the notion of explicit policies appears to be new. Such explicit policies were needed to support automated generation.
- *Generate the documentation too.* When the number of family members is large, development and maintenance of documentation can be prohibitively expensive. Common parts of the documentation can be written once. As Section 6 shows, domain specific languages can help reduce the cost of generating the variable portions of the documentation.
- *Plan the testing carefully.* While the AEE can make family member generation inexpensive, every member must still be tested. Careful planning is required lest testing and debugging consume the savings realized by generation. Hardware designers have long known this and take design for testability seriously, knowing they must minimize

“socket time.” Test costs can be reduced by testing shared components extremely thoroughly. Both input generation and output checking must be considered.

Software product line development requires more effort than development of a single product. However, the extra costs are recouped in later versions of that product, either through maintenance or the production of new versions. Lucent has successfully applied the methods we described in this paper in varying degrees on more than 25 domains, with productivity improvements of as much as four to one.

## 11 Conclusions

This case study, combined with our other experiences with software product lines at Lucent, has convinced us that a product line approach is worth taking in any situation where we expect to produce a number of variations on a software product. Of course, we are not the first to discover the value of automation; the history of software engineering is replete with examples where automation of the development of some or all of a product is worthwhile. What distinguishes our product lines is our conscious attempt to follow an explicit, systematic process for creating product lines, i.e., the FAST process. Having such a process gives us a structure and a set of principles on which to base our thoughts about the economics of a product line, the scope of a product line, its evolution, its underlying architecture, and the tools and processes needed to implement, validate, and verify members of the product line. Put another way, the process and the successes we have had with it give us greater confidence that we can create more product lines in the future with higher probability of success than by simply relying on ad hoc means.

## Acknowledgments

Thanks to David Cuka and Lloyd Nakatani for their guidance on domain-specific languages and to the referees for many helpful suggestions.

## References

- [1] David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison Wesley, 1999.
- [2] Mark Ardis and Janel Green. Successful introduction of domain engineering into software development. *Bell Labs Technical Journal*, 3(3):10–20, July–September 1998.
- [3] J. O. Coplien, D. M. Hoffman, and D. M. Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, 1998.
- [4] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.
- [5] D. L. Parnas. On the design and development of program families. *IEEE Trans. Soft. Eng.*, pages 1–9, March 1976.

- [6] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Academic Press, 1972.
- [7] H.A. Schmid. Creating applications from components: a manufacturing framework design. *IEEE Software*, pages 66–75, November 1996.
- [8] W. Lam. Creating reusable architectures: an experience report. *ACM Soft. Eng. Notes*, 22(4):39–43, July 1997.
- [9] J. Meekel, T.B. Horton, and C. Mellone. Architecting for domain variability. In *Proceedings Second Intl. Workshop on Development and Evaluation of Software Architectures for Product Families*, pages 205–213. Springer Verlag, 1998.
- [10] D. Batory and S. O’Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Soft. Eng. and Methodology*, pages 355–398, October 1992.
- [11] W. Tracz. LILIANNA: a parameterized programming language. In *Proceedings 2nd Intl. Conference on Software Reliability*, pages 66–78, 1993.
- [12] B. Keepence and M. Mannion. Using patterns to model variability in product families. *IEEE Software*, 16(4):102–108, 1999.
- [13] H. Gomaa, R. Fairley, and L. Kerschberg. Towards an evolutionary domain life cycle model. In *Workshop on Domain Modeling for Software Engineering*, 1989.
- [14] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.
- [15] D. M. Hoffman, P. A. Strooper, and L. White. Boundary values and automated component testing. *Journal of Software Testing, Verification, and Review*, 9(1), 1999.
- [16] S.R. Dalal et al. Model-based testing in practice. In *Intl. Conf. Software Engineering*, May 1999.
- [17] D.S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Soft. Eng.*, SE-21(1):19–31, January 1995.
- [18] J.M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. Wiley, 1998.
- [19] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.
- [20] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.
- [21] L. Nakatani, M.A. Ardis, R. Olsen, and P. Pontrelli. Jargons for domain engineering. In *Second Conference on Domain-Specific Languages*, pages 15–24, October 1999.

- [22] D. L. Parnas and P. C. Clements. A rational design process: how and why to fake it. *IEEE Trans. Soft. Eng.*, SE-12(2):251–257, February 1986.
- [23] D.L. Parnas. On a ‘buzzword’: Hierarchical structure. In *Proc. IFIP Congress 1974*. North Holland Publishing Co., 1974.
- [24] D.L. Parnas, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. In *Intl. Conf. Software Engineering*, May 1985.
- [25] J.O. Coplien. *Multi-paradigm Design for C++*. Addison-Wesley, 1999.
- [26] E. Gamma, R. Helm, R. Johnson, and J. Vlissedes. *Design Patterns*. Addison-Wesley, 1994.
- [27] Stuart Faulk. Personal communication, Bell Laboratories, Naperville, IL; Jan. 12, 1999.