Foundations for Feature Engineering of Software Systems

C. Reid Turner and Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

{reid,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-807-96 August 1996

ABSTRACT

The gulf between the user and the developer perspectives leads to difficulties in producing successful software systems. Users are focused on the problem domain, where the system's features are the primary concern. Developers are focused on the solution domain, where the system's lifecycle artifacts are key. Presently, there is little understanding of how to narrow this gulf. This paper presents an organizing viewpoint that we term feature engineering*. Feature engineering promotes features as first-class objects throughout the software lifecycle and across the problem and solution domains. We present a model of lifecycle artifacts and show how features can be incorporated into that model. We then discuss the application of feature engineering to a variety of lifecycle activities, examining in depth its effect on architectural design.*

This work was supported in part by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1 INTRODUCTION

A major source of difficulty in developing and delivering successful software is the gulf that exists between the user and the developer perspectives on a system. The user perspective is centered in the problem domain. Users interact with the system and are directly concerned with its functionality. The developer perspective, on the other hand, is centered in the solution domain. Developers are concerned with the creation and maintenance of lifecycle artifacts, which have no particular meaning in the problem domain. Jackson notes that developers are often quick to focus on the solution domain at the expense of a proper analysis of the problem domain [11]. This bias is understandable, since developers work primarily with solution-domain artifacts. Yet the majority of their tasks are motivated by demands emanating from the problem domain.

Looking a bit more closely at this gulf in perspectives, we see that users think of systems in terms of the *features* provided by the system. Intuitively, a feature is a coherent and identifiable bundle of system functionality that helps characterize the system from the user perspective. Users request new functionality or report defects in existing functionality in terms of features. Developers are expected to translate such feature-oriented requests and reports into a plan for operating on lifecycle artifacts. This plan ultimately should have the desired effect on the features exhibited by the system. The easier the translation process can be made, the greater the likelihood of a successful software system. The key, then, is to gain a better understanding of the notion of feature and how that notion can be carried forward from the problem domain into the solution domain.

Consider the software in a large, long-lived system such as a telephone switch. The software must provide a vast number of complex features to its users, ranging from terminal services, such as ISDN, call forwarding, and call waiting, to network services, such as call routing, load monitoring, and billing.[1] Somehow, the software that actually implements the switch must be made to exhibit these features, as well as to tolerate changes to the features in a cost-effective manner. AT&T Bell Laboratories (now called Lucent Technologies), for example, developed a design in the solution domain for its 5ESS™ switch software by following a layered architectural style [3]. The layers are depicted in Figure 1. The abstract switching machine layer was intended to provide the foundation upon which the features (called "applications") themselves would be built. This was supposed to result in a clean separation of concerns, permitting features to be more easily added and modified. An interesting and important question to ask is whether this turned out to be true and, if not, whether it was a fault of the architecture, of the process followed by the organization, or of some other factors at work.

In the research literature, the notion of feature has not been concretely defined, nor has there been much work specifically addressing its support throughout the lifecycle. Cusumano and Selby [6] describe the strong feature orientation of software development at Microsoft Corporation. That orientation, however, has more to do with project management than with product lifecycle artifacts and activities. Several researchers have studied the *feature interaction problem* [1, 2, 8, 17], which is concerned with how to prevent, identify, and resolve conflicts among a large set of features. The approaches identified in this literature do not provide insight into the role of features across the full range of lifecycle activities and the ability of features to span the problem and solution domains. There has also been some work on *requirements clustering* [10, 12], which would appear to lend

[1] Note that from the perspective of a switch builder, network services are not simply internal implementation functions, but are truly system features, since they must be made available to external organizations, such as telecommunications providers.

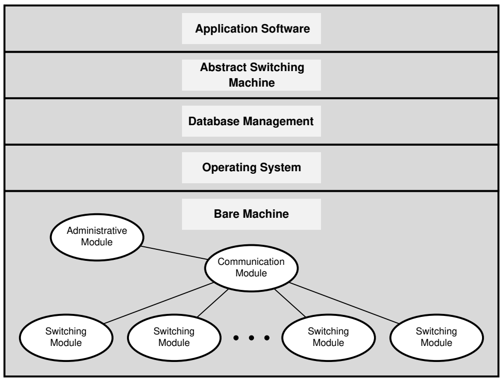

Figure 1: AT&T 5ESS$^{\text{TM}}$ Switch Architecture.

itself to the identification of features within requirements specifications, but does not address the question of how those features would be reflected in lifecycle artifacts, other than in requirements specifications and in a limited form of design prototypes.

We have set out to develop a solid foundation for understanding the notion of feature and, more importantly, for carrying a feature orientation from the problem domain into the solution domain. We term this area of study *feature engineering*. The major goal behind feature engineering is to promote features as "first-class objects" within the software process. Such a promotion holds the promise of supporting features in a broad range of lifecycle activities, including requirements analysis, design, testing, and user documentation, as well as configuration management and reverse engineering.

A feature orientation to software development offers a wide range of benefits. These include identifying features in requirements specifications, evaluating designs based on their ability to incorporate new and modified features, understanding the relationship between a software architecture and feature implementation mechanisms, uncovering feature constraints and interactions, and configuring systems based on desired feature sets. Features are an organizational mechanism that can structure important relationships across lifecycle artifacts and activities.

This paper introduces feature engineering and considers its potential impact. In the next section we present a model of lifecycle artifacts and show how features can be incorporated into that model. We then briefly discuss the application of feature engineering to a variety of lifecycle activities and compare our approach to related work. This is followed by a more in-depth examination of feature engineering applied to one of those activities, architectural design. We conclude with our plans for future research in feature engineering.

2 A MODEL OF FEATURES

The term "feature" is in common use, but has no common definition. Our goal in this section is to develop a working definition that can help convey the essence of feature engineering. We present and evaluate three candidate definitions, and use one as the basis for an entity-relationship model of software lifecycle artifacts.

2.1 An Informal Definition

At the most abstract level, a feature represents a cohesive set of system functionality. Our three candidate definitions each identify this set in a different way.

1. *Subset of functional requirements.* Ideally, the requirements specification captures all the important behavioral characteristics of a system. A feature is a clustering or modularization of individual requirements within that specification. This definition emphasizes the origin of a feature in the problem domain.

2. *Subset of system implementation.* The code modules that together implement a system exhibit the functionality contributing to features. Different subsets of these modules can be associated with particular features. This definition emphasizes the realization of a feature in the solution domain.

3. *Aggregate view across lifecycle artifacts.* A feature acts as a filter that highlights all of the lifecycle artifacts related to that feature. It does this by explicitly aggregating the relevant artifacts, from requirements fragments through code modules, test cases, and documentation. This definition emphasizes historical connections among different artifacts.

At this point in our development of feature engineering, it is not altogether clear which definition is "best", although there are several good arguments in favor of the first definition. In particular, since features originate in the problem domain and not in the solution domain, the first definition appears to be more useful than the second definition. Furthermore, the groupings of artifacts made explicit in the third definition can be inferred by using the first definition together with an appropriate model of the relationships among lifecycle artifacts (e.g., PMDB [13]).

Thus, for the purposes of this paper, we employ the first definition and supplement it with an artifact model upon which we can reason about the relationship of features to other lifecycle artifacts. This model is not intended to be definitive of all lifecycle artifacts. Rather, it is intended to be suggestive of their relationships. Particular development environments may define the artifacts and relationships somewhat differently in detail, but they will nonetheless be compatible with them in spirit.

2.2 A Model of Software Lifecycle Artifacts

Figure 2 shows an entity-relationship diagram that models the role of features within a software process. The entities, depicted as rectangles, correspond to lifecycle artifacts. The relationships, depicted as diamonds, are directional and have cardinality. Despite being directional, the relationships are invertible. Again, we point out that this is just one possible model, but it is illustrative of the concepts we are exploring.

Several important points can be immediately drawn from this model. First, features serve to bridge the problem and solution domains. Second, whereas a requirements specification encompasses all the requirements, features are a means to logically modularize the requirements. Third, the documentation of a feature is a user-oriented description of the realization of that feature within the solution domain. This contrasts with, and complements, their user-oriented description as requirements within the problem domain. Fourth, the distinction between the problem and solution

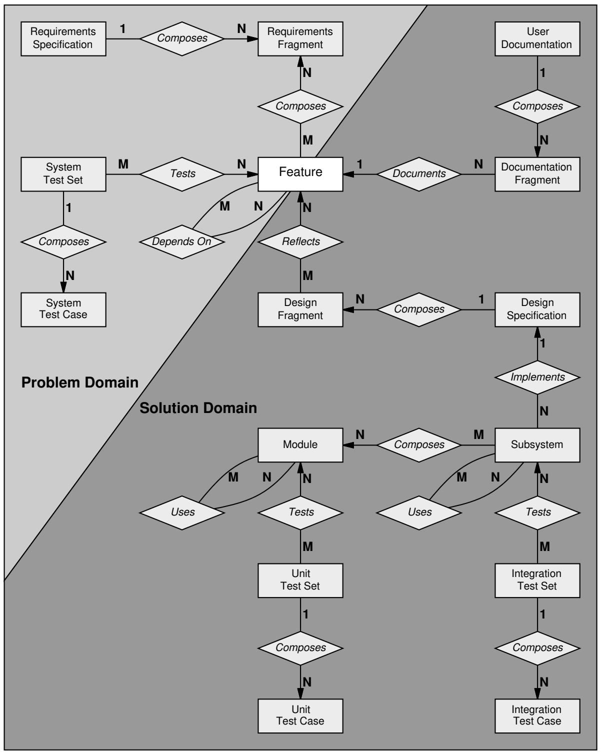

Figure 2: Lifecycle Entities and Relationships.

domains helps illuminate the fundamentally different orientations among the various testing activities in the lifecycle. For example, "system tests" are focused on user-visible properties and are therefore conceived of, and evaluated, within the problem domain. Fifth, the connection between requirements and architectural design is difficult, if not impossible, to formalize beyond the notion that designs reflect the requirements that drive them. However, if those drivers are features, then there is hope for a better tie between the problem and solution domains. This connection is explored more fully in the following two sections.

Two less immediate, but no less important, points can also be drawn from the model. First, while design artifacts are directly related to features, the relationships between features and the deeper implementation artifacts are implicit. For example, a developer might want to obtain all modules associated with a particular feature to make a change in the implementation of that feature. Satisfying such a request would require some form of reasoning applied to the relevant artifacts and relationships. In general, this reasoning would occur at the instance level, as illustrated in Figure 3 and explained below. Second, there are two distinct levels at which features interact. In the problem domain, features interact by sharing requirements or by simply depending on each other for particular services. Similarly, features can interact in the solution domain through shared subsystems and modules or through use dependencies. Although similar in nature, they are quite different in their ramifications. The absence of an interaction in the problem domain does not imply the absence of an interaction in the solution domain, which gives rise to the feature interaction

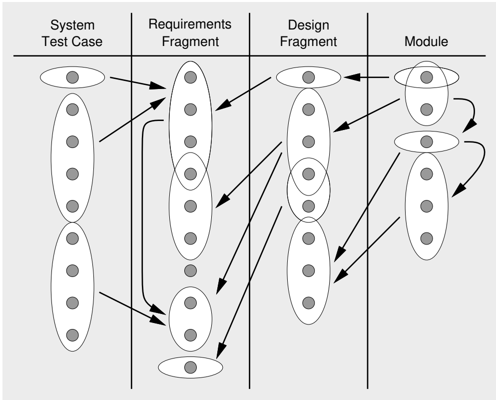

Figure 3: Instances of Entities and Relationships.

"problem" [8]. The reverse is also true, but less obvious, since it arises from the duplicate-then-modify style of code update. Such a style results in a proliferation of similar code fragments that are falsely independent (so-called *self-similar code* [5]).

2.3 The Instance Level

If we take the model of Figure 2 and examine it at the instance level, additional insights into features are revealed. Figure 3 depicts this level for the instances of entities and relationships of a hypothetical system. The figure is simplified somewhat by only considering a subset of the entities. The shaded dots represent individual instances of the entity types named in the columns. The unlabeled ovals within a column represent instances of aggregate entity types, which are defined through the *Composes* relationship in Figure 2. In particular, the ovals represent, from left to right, test sets, features, design specifications, and subsystems. For example, there are ten requirements fragments and four features depicted in the figure. Notice that aggregate artifacts are not necessarily disjoint. So, for example, the top two features share the fourth requirement fragment. The semantics of the arrows are given by the relationships defined in Figure 2. Recall that they are invertible.

An instance diagram forms the basis for reasoning about relationships across the lifecycle. There has been a significant amount of work in developing, maintaining, and even discovering the data for such representations, but none has involved the use of features as a central element. We advocate a representation that allows one to ask questions that include the following.

- *Which features were affected by this change to a requirement?*
- *Which modules should a developer check out to make a change to this feature?*
- *Which features were affected by this change to a module?*
- *Which test cases will exercise this feature?*
- *Which modules are needed to configure the system for these two features?*

For instance, it is clear that different features are able to share requirement specifications. A shared requirement from the switch example could be both the call-forwarding and call-screening features signaling completion with a beep tone. These relationships lead to a deeper set of questions regarding the role of features in a particular system.

The instance diagram also provides useful information for evaluating the structure of the system. For example, we can see that the two features represented by the two topmost ovals in the second column share a requirement, which means that a change to that requirement will impact both features. Further, we can see that despite this shared requirement, the feature represented by the topmost oval is reflected in a single design fragment, which is in turn implemented in a single module. This implies a significant separation of concerns that might make it easier to modify the feature. We can also see that the features represented by the two bottommost ovals do not interact at the requirements level, but do in fact interact at the subsystem level. Finally, we can see that there are two subsystems forming part of the system whose designs are not related to any particular feature. This last observation deserves further discussion.

2.4 The System Core

If a system's functionality is viewed, as we advocate, as a set of features then it is natural to ask the following question: *"Is a system completely composed of the set of features it provides?"* It is clear that systems include underlying componentry to support their features. This underlying componentry, which we call the *core*, arises solely in the solution domain to aid development of features. Users are generally not concerned with the core, and therefore it is not reflected in the requirements. A rather obvious core is evident in the example instance diagram of Figure 3. At the module level, the core is composed of the bottom two subsystems, which have no tie back to any feature at the requirements level, other than in their use by subsystems that do have such a tie.

Chen, Rosenblum, and Vo [4] make a similar observation about the existence of feature components and core components, but their definition is based on test coverage. In particular, core components are those that are exercised by all test cases, whereas feature components are those exercised by only a subset of the test cases.

In a sense, then, the concept of feature is helping us define the concept of core—the core is what remains of the system in the absence of any particular feature. Given that we would like maximum flexibility in both modifying features and in selecting the set of features in any specific configuration of a system, then this definition identifies something quite significant. In fact, what it provides is the conceptual foundation for the role of *software architecture* in software system development. An architecture provides the core functionality of the system within the solution domain that supports the functionality of the system desired in the problem domain. Of course, an architecture must embody assumptions about the features it is intended to support, and the degree to which it correctly anticipates the needs of those features will determine the quality of that architecture. For example, the layers up to and including the abstract switching machine shown in Figure 1 can be considered to form the core of the system and the essence of its architecture. The test of that architecture comes as features are added and modified.

We examine this notion of core more fully below when discussing the effect of feature engineering on software architecture.

3 LIFECYCLE ACTIVITIES

The artifacts and relationships discussed in the previous section are created and maintained through various lifecycle activities. In this section we present a brief and high-level survey of what we see as the impact that feature engineering can have on several of those many activities. Our intention is to suggest some of the broad ramifications of feature engineering, rather than to attempt a complete coverage of the topic.

3.1 Requirements Engineering

Requirements engineering includes activities relating to the

> *"...identification and documentation of customer and user needs, creation of a document that describes the external behavior and associated constraints that will satisfy those needs, analysis and validation of the requirements document to ensure consistency, completeness, and feasibility, and evolution of needs." [9]*

Research in requirements engineering is primarily focused on formulating improved requirements notations and analyses, and on developing methods and mechanisms for requirements elicitation, rationale capture, and traceability.

Requirements engineering is the starting point for feature engineering because it is concerned with the creation and maintenance of the raw material from which features are composed. But requirements analysis must be expanded to include the identification of the set of requirements fragments that compose each feature, as well as the various dependencies that might exist among the features. The fundamental difficulty with requirements engineering in practice today is identified by Hsia et al.:

> *"For the most part, the state of the practice is that requirements engineering produces one large document, written in a natural language, that few people bother to read." [9]*

Feature engineering will make the requirements effort more useful by carrying the results of this effort forward to the other lifecycle activities in a disciplined way.

Looking at requirements engineering research, we are able to identify two efforts that are oriented toward the development of feature identification techniques. Hsia and Gupta[10] have proposed an automated technique for grouping requirements specifications. Their purpose is to support incremental delivery of system functionality through progressive prototypes. The cohesive structures that Hsia and Gupta seek to identify are abstract data types (ADTs) for objects in the problem domain. It is not clear, however, that ADTs are the most suitable representation for features. In addition, their goal of delivering ADT-based prototypes transcends requirements analysis and forces what amount to design choices.

The second effort that has potential for identifying features within system requirements is the requirement clustering technique described by Palmer and Liang. They state

> *"A major requirement of this process is to aggregate a set of N requirements into a set of M requirement clusters where $M \ll N$."[12]*

This is a precise statement of the goal of identifying features. Their motivation, however, is to detect errors and inconsistencies within requirement clusters, and therefore the organizing principle behind their clusters is *similarity* of the requirements within a cluster. On the other hand, for a cluster of requirements representing a feature, the organizing principle should be *relevance* of the constituent requirements to the desired properties of the feature.

3.2 Software Architecture and High-level Design

Ideally, a requirements specification is a precise statement of a problem to be solved; it should structure the problem domain as features to be exhibited by an implemented system. The software architecture, on the other hand, is the blueprint for a solution to a problem, structuring the solution domain as components and their connectors. Researchers in software architecture are focusing attention on languages for architectural design, analysis techniques at the architecture level, and commonly useful styles or paradigms for software architectures.

Feature engineering has significant implications for software architecture. One is in relating the problem-domain structure of features to the solution-domain structure of components and connectors. Rarely is this mapping trivial. Another implication is that, from the perspective of the user, features are the elements of system configuration and modification. A high-level design that seeks to highlight and isolate features is likely to better accommodate user configuration and modification requests. Within this context, then, we see at least three mutually supportive approaches: feature tracing, feature-oriented design methods, and feature-based style evaluation. We briefly discuss the first two approaches here and provide a more in-depth discussion of the third approach in the following section.

The tracing of requirements to designs has been an area of investigation for many years. The basic problem is that it is essentially a manual task whose results are difficult to keep up-to-date and are prone to errors. One way to mitigate this problem is to raise tracing's level of granularity from individual requirements fragments to sensible groupings of such fragments—features. We conjecture that tracing at the feature level is more tractable and, in the end, more useful than traditional methods.

A somewhat different approach to high-level design than traditional functional decomposition or object-oriented design methods arises from a focus on features. The starting point for a feature-oriented design method is an analysis of the intended feature set to gain an understanding of the features, both individually and in combination. Of particular importance is understanding the requirements-derived dependencies among the features. If one takes feature prominence as a design goal, then the top-level decomposition of the architecture should match the decomposition of the requirements specification into features. At each successive level of architectural decomposition, the goal should continue to be feature isolation. Points of interaction among features naturally arise from shared requirements, as well as from the need to satisfy non-functional requirements, such as performance. The criteria for creating new components should be to capture explicitly some shared functionality among some subset of features. In this way, the feature interactions are caused to distill out into identifiable components.

Combining tracing at the feature level with a design method that leads to modules representing features and feature interactions should help to illuminate the traditionally obscure relationship between specific features and the design elements supporting them. When a request for a feature change is presented to a developer, that feature can be traced immediately to a design element associated with the feature. Any potentially problematic interactions with other features become visible through their capture in shared modules representing that interaction.

3.3 Low-level Design and Implementation

Low-level design and implementation are the activities that realize the modules and subsystems identified in architectural design. While we could postulate a need for feature-oriented implementation languages, our experience with feature engineering has not lead to the discovery of any compelling arguments in their favor. The effect that feature engineering has on these activities is more likely felt indirectly through the effects on the high-level design and testing activities and through the contribution of a tool set that makes the relationships across artifacts visible to the developer.

Nevertheless, a feature orientation frequently exists during implementation. Cusumano and Selby [6] report that development efforts for many Microsoft product groups are organized by the features of their product. Small teams of developers are assigned responsibility for one or more of those features. Especially complicated features are assigned to stronger teams that include developers with more experience. This organizational structure built around features extends to testing teams that are assigned responsibility for testing particular features.

A primary benefit to be gained from concentrating on features as a bridge from the problem domain to the solution domain is a reduction of the intellectual burden placed on developers when interacting with the implementation of a system's features. Developers will be able to work with a feature implementation without having to recreate the mapping from problem-domain artifacts to solution-domain artifacts, and vice versa.

3.4 Testing

Testing is an approach to software verification that involves experimenting with a system's behavior to determine if it meets expectations. In practice, there are three levels of testing. Unit testing is used to test the behavior of each module in isolation. Integration testing is used to detect defects in the interactions among modules at their interfaces. System testing is focused on testing the complete system as a whole for compliance with the requirements set out by the users. System testing is oriented toward the problem domain, while unit and integration testing are oriented toward the solution domain (see Figure 2).

Feature engineering can have an impact on testing activities by suggesting a somewhat different organization of test sets than is traditionally encountered. In particular, test sets would be organized around the feature or features they are intended to test. Taking all the feature tests together, we get the equivalent of a system test set. Where a feature implementation is confined to a single module, tests for that feature amount to unit tests for the module. Of course, feature implementations frequently involve more than one module. In this case, feature tests are a mechanism for evaluating module integration. The connections between features that are highlighted by instance diagrams, such as Figure 3, point out sets of features that should be tested in combination. Such feature-combination tests could prove useful for detecting unwanted feature interactions.

The feature-oriented organization of test sets can also help to minimize regression testing. This harks back to the theme of users posing requests in terms of features. If a developer can make a change to the system with respect to some particular feature, then only the tests related to that feature need to be run.

3.5 Configuration Management

Configuration management is the discipline of coordinating and managing evolution during the lifetime of a software system. Traditionally, configuration management is concerned with maintaining versions of artifacts, creating derived objects, coordinating parallel development efforts, and constructing system configurations.

The vocabulary of existing configuration management systems is oriented toward solution-domain artifacts, such as files, modules, and subsystems. Many of the accepted configuration management techniques such as version management and derived-object creation should be directly applied at the feature level. For example, it should be possible to request a version of the artifacts associated with a particular feature. It should also be possible to construct a system configured by a given set of features. Another useful capability would be the delineation of parallel workspaces based on features. For features to become first class, they will have to exist in the models of the systems that are built. This has the potential for raising the level of abstraction at which developers work from files to features.

Realizing this expanded role for configuration management will require feature implementations to be separately encapsulated and versioned. Bare files do not appear to be the right abstraction for this purpose. Change sets [7, 16], on the other hand, show promise as a potentially useful storage base. In addition, information about feature dependencies at both the specification and implementation levels will be needed for assembling consistent configurations.

3.6 Reverse Engineering

Reverse engineering is the process of discovering the structure and organization of an existing system from whatever artifacts may be available. Typically, such artifacts are limited to those associated with the implementation, such as source files. The activities in reverse engineering center around the application of various analyses techniques to the artifacts in order to reveal internal structure, as well as to reveal static and dynamic dependencies.

The primary influence of feature engineering on reverse engineering is to focus the analyses toward discovering connections to features. In essence, this means recreating the (lost) relationships in Figure 3. One possible technique would be to broaden the scope of program slicing, following Sloane and Holdsworth [15], to create a feature slice through the implementation artifacts. A feature slice would include all of the fragments that contribute to a feature's implementation. Working in the other direction, if a feature test set existed, then observations of test case executions could reveal the portions of the implementation that were involved in implementing the feature.

4 FEATURE-BASED STYLE EVALUATION

Change is endemic to nearly all successful software systems, so features are a natural increment to system functionality. The demand for change is frequently couched in terms of the additional features that the system should provide and the defects in existing features that should be corrected. Understanding the difficulty involved in adding and modifying features is a firm analytical basis upon which to evaluate architectural styles. Results from such an analysis provide design rationale that support informed choices among competing styles.

Here we begin to examine this particular interaction between feature engineering and software architecture as a way to illustrate the general utility of a feature orientation in lifecycle activities.

It is important to note that this study is only an initial attempt at using feature engineering to gain insight into architectures, and that much more work remains to be done in this area.

4.1 Mechanisms for Change

By separately viewing features and the architectural core, we can define a range of mechanisms for adding or modifying a feature. These mechanisms ultimately result in changes to the set of components comprising the system. We list them in order of increasing impact on the development effort.

1. changes to components, excluding core components;
2. changes to components, including individual core components; and
3. changes to components that involve a modification of the architectural style embodied in the core.

The first mechanism is the one intended for the switch software whose architecture is depicted in Figure 1. Features were to be added or modified as largely separate components built on top of the abstract switching machine layer, which served as the interface to the core. A greater level of difficulty is encountered through the second mechanism. For example, the addition of ISDN features to the switch required changes at all levels of the architecture, since the core was not defined in such a way as to anticipate those features. The third mechanism is, of course, the most radical. In essence, the architectural style does not feasibly support the required functionality, and so must be replaced. Generally, such a change is resisted by an organization well beyond when it is appropriate. But this is understandable because of the severe cost implied.

As noted above, the core of a system captures the essence of its architecture. Each canonical architectural style should therefore present a distinct core that highlights its differences from other styles. We can gain insight into the nature of an architectural style by considering what it means to apply the change mechanisms to systems that follow the style. Here we look at two styles, *pipe and filter* and *event-action*. We restrict ourselves to considering only the first two change mechanisms. In addition, we only consider the effects of adding a feature, since they are similar to the effects of modifying a feature.

4.2 Pipe and Filter Style

The core of a pipe and filter style consists of the following four elements: the standardized input and output channels provided by each component, the pipes that carry the information, the structure of the information that flows within pipes, and the controller that connects filters together into a pipeline. A good example of a system that follows this style is an HTTP (Hyper Text Transfer Protocol) proxy server. It is designed to allow the addition of new features into the server, such as data compression, by simple insertion or modification of filters.

There are two main methods for adding a new feature using the first change mechanism. One method is to replace an existing filter with a version that provides the additional feature. For example, we may want to add data compression to the HTTP proxy server. We can do this by adding compression to the last filter in the pipeline and decompression to the first filter. While keeping the core intact, this method confounds the implementations of features. The other method involves creating a new filter and adding it into the pipeline. This would require the use of an additional pipe resource and a configuration change to the controller so that the controller can establish connections with the new filter. But these are not considered changes to the core, since they make use exclusively of primitive features provided by the core for exactly that purpose. Although this method has the advantage of keeping the feature implementations separate, there is still a potential feature interaction, since the output from one filter is expected to be the input to another.

The second change mechanism involves modifications to the core as part of adding a feature. Potentially, each of the four core elements of the pipe and filter style could be altered to support a new feature. Let us consider one such modification. Suppose we wish to have features that run on different hosts on the internet, perhaps to take advantage of some remote processing power or to help secure sensitive data. This would mean creating pipelines that span the internet, rather than simply confining themselves to a single machine. Such a change could be carried out by modifying different core elements. For instance, changing the standardized input and output channels and the nature of the pipes, one could support connections that span network links. But this change involves modifying each of the components that takes part in the new feature.

Changes to the core that require modifications to all the filters is potentially an onerous task. Often pipeline systems employ individual components that are not under the control of the system developer. An alternative approach to adding the distribution feature is to modify the controller that connects the components. The control mechanism can be responsible for managing the flow of information across the network links by inserting hybrid components that support the old standardized connections on one end and the network-based connections on the other end. This change to the core will have an impact on the existing filters due to the decreased reliability of, and delays inherent in, network channels. Nevertheless, the basic architecture of the system could remain the same.

4.3 Event-action Style

In an event-action style, the core consists of a message substrate and the protocols that are used to send and receive messages. The message substrate is a shared medium that is responsible for message delivery, while the protocols provide for anonymous communication. A good example of a system using the event-action style is the Field environment [14]. This system provides a collection of software development tools that are integrated using the message substrate. The message substrate essentially directs messages generated by one tool to all other tools that have registered their interest in the message. New features can be added to the environment by adding or modifying tools and by changing the set of generated messages.

Adding a feature to an event-action system using the first change mechanism is conceptually similar to adding a feature to the pipe and filter style, in that there are two distinct methods, one of which confounds features and the other of which keeps them separate. In particular, there is a choice when adding a feature whether to modify an existing component or to add a new component. For example, one might want to add a feature to list the source code surrounding the current line being executed by a debugger. This can be done either by modifying the debugger itself to include a lister, or by adding a new listing tool. The listing tool would be added by having it register for current line number messages, presumably generated by the debugger. The message substrate acts as a decoupling mechanism that allows features to exist in isolation. The ability to extend the system features by adding independent components that react to messages is a defining characteristic of the event-action style. Some reconfiguration of the message substrate is required for a new component to participate. But, as in the pipe and filter style, this is not considered a change to the core, since it makes use exclusively of primitive features provided by the core for exactly that purpose.

Making a modification to the core using the second change mechanism is another way to add a feature to an event-action system. An example would be to add security and authentication to the system. For authentication, the message protocol would be enhanced to require an exchange of password each time a tool connects to the message substrate. Another measure of security would be to encrypt the data that is sent across the message substrate. This feature would prevent untrusted components from participating in the communication. These changes to the core would require retrofitting the participating components with the new protocols. Clearly, such a change to the core involves a significant impact on the entire system.

4.4 Style Evaluation Summary

Both the pipe and filter style and the event-action architectural style enable the addition of features to a system without having an impact on the architectural core. In fact, the inherent flexibility in those styles supports feature additions that can be contained to a small set of components. When the strategy for introducing a new feature involves a change to the core, however, the impact of the change can be seen to be much greater. The maintainer of a system must be careful to understand the tradeoffs in using these different approaches.

To date, there has been little support available for the analysis of architectural designs beyond basic functional correctness. Cohesion and coupling of modules remain the sole generally accepted criteria for such analysis. We have demonstrated how to begin to employ features as a tool for evaluating architectural styles. Feature engineering thus provides a new dimension along which architectures can be evaluated.

5 CONCLUSION

In current practice, the notion of feature is an ill-defined yet widely used concept. It becomes meaningful when seen as a way to modularize system requirements and when related to the range of lifecycle artifacts and activities. The goal of feature engineering is to provide disciplined methods for projecting features from the problem domain into the solution domain. Doing so will bridge the gulf between the user and developer perspectives of a system. In addition, feature engineering has the potential to improve system understanding by raising the level of abstraction consistently across the lifecycle.

This paper has laid the foundation for feature engineering by presenting a model for features in terms of their relationships to other artifacts. Further, we have explored how feature engineering affects lifecycle activities, including requirements engineering, testing, configuration management, and reverse engineering. We have also taken a deeper look at its affect on the emerging area of software architecture. We can conclude that the concepts involved in feature engineering cut across the entire software lifecycle, and that research efforts in a number of software engineering disciplines are relevant to feature engineering. These efforts should be leveraged to help bring features to the fore.

In this paper, the features we describe are undifferentiated from each other. In complex software systems, features will exist within hierarchies organized by interesting properties such as dependence, importance, and complexity. Understanding such hierarchies, and particularly the nature of feature dependencies, should produce additional insights and benefits to software development.

In addition to refining basic concepts of feature engineering, there is a need for the development of tools to support the integration of features into the solution domain. Using a feature orientation to make explicit the relationships among development artifacts should increase a developer's ability to comprehend complex systems. This will only come about, however, if tools exist to capture, organize, and present the structure that features offer. For example, we can easily imagine the development of new-generation configuration management tools that are cognizant of features. These tools would, for instance, provide primitives for controlling access to artifacts in terms of features, as well as support the configuration of systems based on feature sets.

ACKNOWLEDGMENTS

This work benefited greatly from discussions with Alfonso Fuggetta of the Politecnico di Milano and David Rosenblum of the University of California at Irvine. This work was supported in part by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

REFERENCES

[1] A.V. Aho and N. Griffeth. Feature Interaction in the Global Information Infrastructure. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 2–5. ACM SIGSOFT, October 1995.

[2] E.J. Cameron and H. Velthuuijsen. Feature Interactions in Telecommunications Systems. *IEEE Communications Magazine*, 31:18–23, August 1993.

[3] D.L. Carney, J.I. Cochrane, L.J. Gitten, E.M. Prell, and R. Staehler. Architectural Overview. *AT&T Technical Journal*, 64(6):1339–1356, 1985.

[4] Y.-F. Chen, D.S. Rosenblum, and K.-P. Vo. TestTube: A System for Selective Regression Testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–220. IEEE Computer Society, May 1994.

[5] K.W. Church and J.I. Helfman. Dotplot: A Program for Exploring Self-Similarity in Millions of Lines for Text and Code. *Journal of Computational and Graphical Statistics*, 2(2):153–174, June 1993.

[6] M.A. Cusumano and R.W. Selby. *Microsoft Secrets*. The Free Press, New York, 1995.

[7] P.H. Feiler. Configuration Management Models in Commercial Environments. Technical Report SEI-91-TR-07, Software Engineering Institute, Pittsburgh, Pennsylvania, April 1991.

[8] N.D. Griffeth and Y. Lin. Extending Telecommunications Systems: The Feature-Interaction Problem. *IEEE Computer*, 26(8):14–18, August 1993.

[9] P. Hsia, A.M. Davis, and D.C. Kung. Status Report: Requirements Engineering. *IEEE Software*, 10(6):75–79, November 1993.

[10] P. Hsia and A. Gupta. Incremental Delivery Using Abstract Data Types and Requirements Clustering. In *Proceedings of the Second International Conference on Systems Integration*, pages 137–150. IEEE Computer Society, June 1992.

[11] M. Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices*. Addison-Wesley, Reading, Massachusetts, 1995.

[12] J.D. Palmer and Y. Liang. Indexing and Clustering of Software Requirements Specifications. *Information and Decision Technologies*, 18(4):283–299, 1992.

[13] M.H. Penedo and E.D. Stuckle. PMDB—A Project Master Database for Software Engineering Environments. In *Proceedings of the 8th International Conference on Software Engineering*, pages 150–157. IEEE Computer Society, August 1985.

[14] S.P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57–66, July 1990.

[15] A.M. Sloane and J. Holdsworth. Beyond Traditional Program Slicing. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA '96)*, pages 180–186. ACM SIGSOFT, January 1996.

[16] Software Maintenance & Development Systems, Inc, Concord, Massachusetts. *Aide de Camp Product Overview*, September 1994.

[17] P. Zave. Feature Interactions and Formal Specifications in Telecommunications. *IEEE Computer*, 26(8):20–29, August 1993.