A Conceptual Basis for Feature Engineering

C. Reid Turner,[1] Alfonso Fuggetta,[2] Luigi Lavazza,[2] and Alexander L. Wolf[1]

[1]Department of Computer Science
University of Colorado
Boulder, CO 80309 USA
{reid,alw}@cs.colorado.edu

[2]Dipartimento di Elettronica e Informazione
Politecnico di Milano
20133 Milano, Italy
{fuggetta,lavazza}@elet.polimi.it

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 life-cycle artifacts are key. Presently, there is little understanding of how to narrow this gulf.

This paper argues for establishing an organizing viewpoint that we term feature engineering. *Feature engineering promotes features as first-class objects throughout the software life cycle and across the problem and solution domains. The goal of the paper is not to propose a specific new technique or technology. Rather, it aims at laying out some basic concepts and terminology that can be used as a foundation for developing a sound and complete framework for feature engineering. The paper discusses the impact that features have on different phases of the life cycle, provides some ideas on how these phases can be improved by fully exploiting the concept of feature, and suggests topics for a research agenda in feature engineering.*

A. Fuggetta and L. Lavazza are also with CEFRIEL (http://www.cefriel.it). The work of A. Fuggetta was supported in part by CNR. The work of A.L. Wolf was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Numbers F30602-94-C-0253 and F30602-98-2-0163. 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.

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 life-cycle artifacts, which do not necessarily have a 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 [18]. 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 report defects or request new functionality in terms of features. Developers are expected to reinterpret such feature-oriented reports and requests into actions to be applied to life-cycle artifacts, such as modifying the appropriate set of implementation files. The easier the interpretation 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.

As an illustration of the central importance of features, consider the software in a large, long-lived system such as a telephone switch. This kind of system is composed of millions of lines of code, and includes many different types of components, such as real-time controllers, databases, and user interfaces. 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. Bell Laboratories, for example, developed a design in the solution domain for its 5ESS® switch software by following a layered architectural style [5]. This was supposed to result in a clean separation of concerns, permitting features to be more easily added and modified.

Despite the continuing interest in the notion of feature, to date there has been little work

[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.

specifically addressing its support throughout the life cycle. Nevertheless, one does find the notion used in several relevant, if limited, ways.

- In domain analysis and modeling, the activity of *feature analysis* has been defined to capture a customer's or an end user's understanding of the general capabilities of systems in an application domain [21, 30]. Domain analysis uses the notion of features to distinguish basic, core functionality from variant, optional functionality [14]. Although features are an explicit element of domain models, their connection to other life-cycle artifacts is effectively non-existent.

- There has been work on so-called *requirements clustering* techniques [17, 25], which would appear to lend itself to the identification of features within requirements specifications. But they do not address the question of how those features would be reflected in life-cycle artifacts other than requirements specifications and in a restricted form of design prototypes.

- Cusumano and Selby [8] describe the strong orientation of software development at Microsoft Corporation toward the use of *feature teams* and *feature-driven architectures*. That orientation, however, has more to do with project management than with product life-cycle artifacts and activities. Cusumano and Selby offer no evidence that the notion of feature has been driven throughout the development process, although doing so would seem natural in such a context.

- Several researchers have studied the *feature interaction problem*, which is concerned with how to identify, prevent, and resolve conflicts among a set of features [1, 4, 15, 23, 35]. The approaches identified in this literature do not provide insight into the role of features across the full range of life-cycle activities and the ability of features to span the problem and solution domains.

- Automatic *software generation* is based on an analysis of a domain to uncover reusable components [3, 31]. The components are grouped into subsets having the same functional interface; a complete system is created by choosing an appropriate element from each subset. The choice is based on the "features" exhibited by the elements. Here, the term feature is essentially restricted to extra-functional characteristics of a component, such as performance and reliability. Functionally equivalent systems having different extra-functional characteristics can then be automatically generated by specifying the desired features—that is, the extra-functional characteristics. Although this work represents an important element in support of features, it needs to be extended to encompass the generation of functionally dissimilar systems through selection of functional characteristics.

Thus, there is a growing recognition that features act as an important organizing concept within the problem domain and as a communication mechanism between users and developers. There has also been some limited use of the concept to aid system configuration in the solution domain. There is not, however, a common understanding of the notion of feature nor a full treatment of its use throughout the life cycle.

We have set out to develop a solid foundation for 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, and thus have features supported in a broad range of life-cycle activities. These activities 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 thus an organizational mechanism that can structure important relationships across life-cycle artifacts and activities.

This paper proposes some basic concepts for feature engineering and evaluates the potential impact of this discipline on software life-cycle activities. It is based on our experience in applying feature concepts to the modeling of several software systems, including the software of an ItalTel telephone switch, and in evaluating the support for a feature orientation offered by the leading commerical configuration management systems. This paper does not, however, attempt to report on particular solutions to problems in software engineering, but rather to articulate a framework within which solutions might be developed and assessed. Therefore, this paper should be considered a first step toward the complete and detailed definition of feature engineering and of its relationship with other domains of software engineering.

In the next section we discuss a typical entity-relationship model of life-cycle artifacts and show how features can be incorporated into that model. We then describe the application of feature engineering to a variety of life-cycle activities. In Section 4 we present a study of the Italtel telephone switch software that serves as an initial validation of some of the principal ideas developed in this paper. We conclude with our plans for future research in feature engineering.

2 The Role of Features within the Process

The term "feature" has been in common use for many years. In 1982, for instance, Davis identified features as an important organizational mechanism for requirements specifications.

> *"...for systems with a large number of internal states, it is easier, and more natural, to modularize the specification by means of features perceived by the customer." [9]*

In a recent survey on feature and service interaction in telecommunication systems, Keck and Kuehn mention a similar definition developed by Bellcore.

> *"The term* feature *is defined as a 'unit of one or more telecommunication or telecommunication management based capabilities a network provides to a user'..." [22]*

Unfortunately, despite these attempts to precisely define the notion of feature, the term is often interpreted in different and somewhat conflicting ways. Here, we present and evaluate three candidate definitions that are intended to capture the range of interpretations commonly used in the software engineering community. The first definition refers to the interpretation of the term feature as offered by most of the scientific literature on the subject, including the two examples above. The other two definitions represent other interpretations of the term feature, as used especially by practitioners. Our intent here is to emphasize the differences among these interpretations, to indicate how they are interrelated, and, therefore, how they can be eventually reconciled.

2.1 An Informal Definition

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

1. *Subset of system requirements.* Ideally, the requirements specification captures all the important behavioral characteristics of a system. A feature is a grouping 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. A feature is a subset of these modules associated with the particular functionality. This definition emphasizes the realization of a feature in the solution domain.

3. *Aggregate view across life-cycle artifacts.* A feature is a filter that highlights the life-cycle artifacts related to a specific functionality by explicitly aggregating the relevant artifacts, from requirements fragments to code modules, test cases, and documentation. This definition emphasizes connections among different artifacts.

Features are a user-centered view of a system's functionality and, therefore, originate in the problem domain, not the solution domain. The first definition captures this critical aspect. A feature is a set of individual requirements within a requirements specification for the system. The membership criterion for this set is a direct relationship to a single, identifiable functionality.

The second of these definitions has an inherent weakness; accepting it implies that a feature does not exist until it is implemented. Therefore, it defines what we refer to as a feature implementation.

There are many possible implementations for the same feature. Certainly it is possible to consider a feature independently of its realization in a particular system.

The third definition implies that a feature will change simply because an associated artifact, such as its documentation, changes. A feature should remain a feature regardless of how it is implemented, documented, or tested. In addition, the groupings of artifacts made explicit in the third definition can be inferred by using the first definition, given an appropriate model of the relationships among life-cycle artifacts (e.g., PMDB [26]).

Given these arguments, we employ the first definition for the purposes of this paper. We use the definition as a core concept to develop a model of the artifacts that are created during software engineering activities. This model is not intended to be definitive of all life-cycle 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. The model allows us to reason about the relationship of features to other life-cycle artifacts, and to articulate and illustrate the benefits derived from making features first class.

2.2 Features and Software Life-Cycle Artifacts

Figure 1 shows a simple entity-relationship diagram that models the role of features within a software process. The model derives from the concepts typically used in software engineering practice and commonly presented (often informally) in the literature. The entities, depicted as rectangles, correspond to life-cycle 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, and it is just meant to be illustrative of the concepts we are exploring. It is not meant to be a complete model or to constitute the novel contribution of the paper. We have derived it by studying available literature on the subject (e.g., PMDB [26]) and by analyzing our own experiences on several industrial projects, one of which is discussed in Section 4.

The model defines some of the key aspects and properties that are relevant to our understanding of the role of features in the life cycle, and are further explored in this paper.

1. Features as life-cycle entities are meant to bridge the problem and solution domains.

2. Features are a means to logically modularize the requirements.

3. 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, the user-oriented description of a feature as a set of requirements within the problem domain.

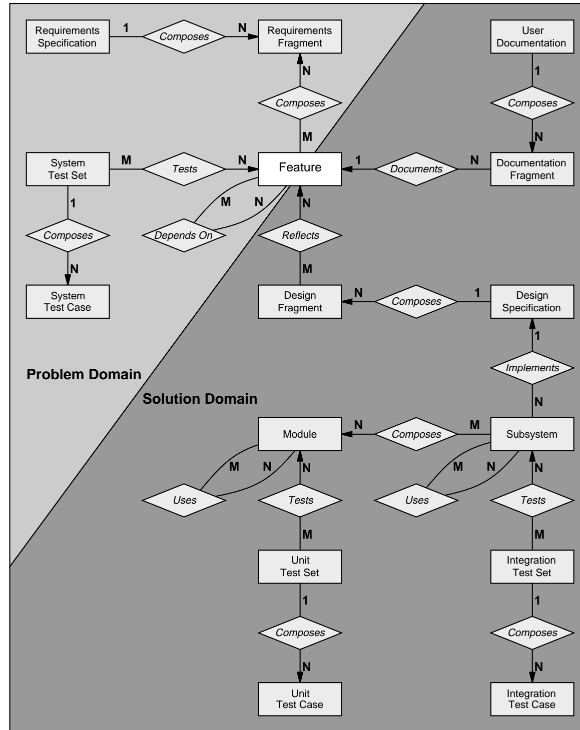

Figure 1: Common Life-Cycle Entities and Relationships.

4. The distinction between the problem and solution domains helps illuminate the fundamentally different orientations among the various testing activities in the life cycle. For example, system tests are focused on user-visible properties and are therefore conceived of, and evaluated, within the problem domain.

5. 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.

Two less immediate, but no less important, points can also be seen in 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 2 and explained below. Second, there are two distinct levels at which features interact. In

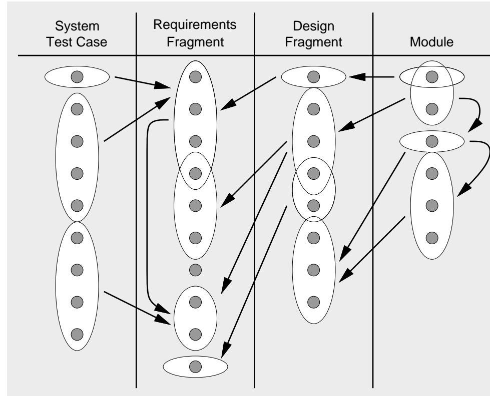

Figure 2: Instances of Entities and Relationships.

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 implementation-based feature interaction problems [15]. 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* [7]).

2.3 The Instance Level

If we populate the model of Figure 1 and examine it at the instance level, additional insights into features are revealed. Figure 2 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 1. 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 1. Recall

that they are invertible.

An instance diagram forms the basis for reasoning about relationships across the life cycle. 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 requirements specifications. A shared requirement from the switch example could be both the call-forwarding and call-screening features signaling completion with an aural tone. These relationships lead to a deeper set of questions regarding the role of features in a particular system. Answering the questions that are posed here implies the existence of a number of many-to-many relationships. Researchers have investigated some those relationships [10, 27] and proposed solutions that would answer some of the questions. Since features are a natural stucturing of the requirements specification, organizing the relationships around features holds promise for making such efforts more valuable across life-cycle activities.

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 may potentially 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 directly reflected in the requirements. A rather obvious core is evident in the example instance diagram of Figure 2. 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 [6] 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 to 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.

3 Features and Life-Cycle Activities

The artifacts and relationships discussed in the previous section are created and maintained through various life-cycle 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 related to

> *"...identification and documentation of customer and user needs, creation of a doc-*

ument 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." [16]

Research in requirements engineering is primarily focused on formulating improved notations and analyses, and on developing methods and mechanisms for elicitation, rationale capture, and traceability. Requirements engineering is the starting point for feature engineering, since it is concerned with the creation and maintenance of the raw material from which features are composed. Requirements analysis must include the identification of the set of requirements fragments that comprise each feature, as well as the various dependencies that might exist among the features. Indeed, several requirements methods have been proposed that are potentially useful in the development of feature identification techniques.

Domain analysis is a method for understanding requirements in a particular problem domain. The product of domain analysis is a domain model, which captures the essential entities in a domain and the relationships among those entities. Research in the area of domain analysis is focussed on the development of better methods for eliciting and representing domain models. In addition, for stable domains, automated software generation techniques are being sought that can exploit the domain models. These techniques would provide reuse of components that implement the entities defined in the domain model.

Several domain analysis methods, including FODA [21, 30], use the term feature to refer to the capabilities of systems in a domain. They typically seek to distinguish the features that represent basic, core functionality from those that represent variant, optional functionality. A good example of this approach is the domain modeling method and environment of Gomaa et al. [14]. The environment is used to generate object specifications for target systems based on a domain model. The object specifications are therefore artifacts in the solution domain. Clearly, because the object specifications are generated, their relationship to features can be easily maintained, although this notion of feature is not well developed.

Domain analysis plays a role in the software generation work of Batory and O'Malley [3] and of Sitaraman [31], where they analyze a domain to uncover reusable components. The components are grouped into subsets (*realms*, in the terminology of Batory and O'Malley) having the same functional interface; a complete system is created by choosing an appropriate element from each set. The choice is based on the "features" exhibited by the elements. Here, the term feature is essentially restricted to extra-functional characteristics of a component, such as performance and reliability. A system can then be automatically generated by specifying the desired extra-functional

characteristics. Although this work represents an important step toward feature engineering, it requires a fuller treatment of the concept of feature. In particular, if feature specifications also represented functional differences, then the generation process would allow for the creation of functionally different systems.

In addition to domain analysis techniques, there are also a number of other requirements methods that have been developed to represent and structure requirements. Representation and structuring of requirements are key elements of a feature orientation, but the methods discussed below would need to be enhanced in order to be appropriate for feature engineering.

Use cases [13, 20] are a method for representing requirements that have become quite popular within the object-oriented analysis and design community. Use cases are similar to features to the extent that they represent requirements and some relationships among those requirements, such as "uses" and "extends". However, features support additional relationships, such as "conflicts", "competes", and "constrains", not currently represented by use cases. Moreover, features capture non-functional requirements also not currently expressable through use cases.

Quality Function Deployment (QFD) [11] is a requirements and design process aimed at identifying customer desires and related technical requirements. QFD exploits some notion of feature, but does not offer any specific aid to support the representation and management of features during the requirement engineering phase, nor throughout the software development process.

Hsia and Gupta [17] 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. However, their goal of delivering ADT-based prototypes transcends requirements analysis and forces what amount to design choices.

Palmer and Liang [25] have described a somewhat different requirements clustering technique. They define the problem as an effort to "aggregate a set of N requirements into a set of M requirement[s] clusters where $M \ll N$". This is a precise statement of the goal of identifying features. Their motivation, however, is to detect errors and inconsistencies within requirements clusters, and therefore the organizing principle behind their clusters is *similarity* of the requirements within a cluster. In other words, they seek to find sets of redundant requirements in order to analyze the elements of the set for consistency. For feature engineering purposes, we instead advocate that the organizing principle of a cluster should be *relevance* of the constituent requirements to the desired properties of the feature; the issue of redundancy and consistency is orthogonal, and so a clustering for that purpose, while important, is also orthogonal.

To conclude this discussion of requirements engineering, let us return to the feature interaction problem in telecommunications applications mentioned in Section 1. In telephone switch software, features such as call waiting and call forwarding both relate to the treatment of incoming calls to a busy subscriber line [2], and thus exhibit overlapping requirements fragments. The identification of such feature interactions at the requirements phase can help eliminate unanticipated interaction problems during later phases in the software life cycle. The most common research approach to this problem is the application of formal verification techniques to system specifications, with the goal of detecting all undesired feature interactions. The critical part of this activity is the system specification—that is, the definition and application of a specification technique that actually captures the relevant properties of the system. Jackson and Zave propose DFC [19], a virtual architecture for representing features that can be dynamically composed to form a configuration suitable to provide a specific service. From our point of view, features can be represented and handled in several different ways. In particular, features in DFC are treated as first class, and expected to drive the subsequent model checking activities and the design of the concrete system architecture. This clearly conforms to our idea of a feature-centric development process.

In general, 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." [16]*

Feature engineering holds the promise to make the requirements effort more useful by carrying the results of this effort forward to the other life-cycle activities in a disciplined way.

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 implementation. 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 two mutually supportive approaches: feature tracing and feature-oriented design methods.

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 extra-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.

In the telephone switch, for example, the call forwarding, abbreviated dialing, and direct connection features all require the association of directory numbers with a subscriber line [2]. Each so-called Switching Module in the architecture (i.e., the module responsible for routing calls) includes a special database to store such information. Thus, the database, as a design element, is driven by a specific and identifiable set of features. Maintaining this relationship is critical to understanding how to properly evolve this element without violating some constraint imposed by a feature.

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. Moreover, 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 [8] 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 teams assigned responsibility for testing particular features.

Ossher and Harrison [24] discuss a method of extending existing class hierarchies by applying "extension hierarchies", which would appear to bear some relation to feature engineering at the implementation level. Goals for this work include reducing modification of existing code and separating different extensions. Much like change sets in configuration management (see Section 3.5), these extensions can be used as a conceptual mechanism to add functionality to existing object-oriented systems. Unfortunately, this research describes extensions only at the granularity of object methods, which seems inappropriate for dealing with complex features such as the call-forwarding feature of a telephone switch. In addition, the semantic compatibility of combined extensions are not well understood in this technique, which is a critical need for feature engineering.

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, including the system's intended functionality and performance. System testing is oriented toward the problem domain, while unit and integration testing are oriented toward the solution domain (see Figure 1).

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. The telephone switch software, for example, supports a vast number of features that need to be tested for every release. Having requirements for each feature provides a basis for testing each feature in isolation. 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 2, point out sets of features that should be tested in combination. This would be useful, for example, in guiding the testing of modifications to the database component of the telephone switch's Switching Module, which is shared by several features [2]. Such feature-combination tests might detect 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 (and, possibly, any other features that depend upon 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, the developers of the telephone switch software should be able to populate a workspace through a request for a specific version of all the artifacts associated with a particular feature, such as call waiting, by simply identifying the feature, not each of the individual relevant artifacts. It should also be possible to issue a request to construct a system where that request can be parameterized by a given set of features. For example, it might be useful to construct a "compact" release of the telephone switch software that has basic call processing features but no call waiting or call forwarding 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 [12, 33], 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 any available artifacts. Typically, such artifacts are limited to those associated with the implementation, such as source files. The activities in reverse engineering center on 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 2. For example, reverse engineering could be used to discover the interactions between call waiting and call forwarding, or to discover the features that are dependent on the database component of the Switching Module.

One possible technique would be to broaden the scope of program slicing, following Sloane and Holdsworth [32], 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 Case Study: The Software of an Industrial Telephone Software

We recently performed a study of Italtel's telephone switch software in order to validate some of the ideas presented in this paper against a large and complex software system. Our intent was to identify features in the system and to evaluate the support that the development process provides for managing features during the system's evolution.

The software implementing the switch consists of millions of lines of code and thousands of files. Each release of the switch incorporates added functionality, while the basic software architecture remains stable; only small changes are relatively frequent. This is consistent with our view of software architecture as defining the system core (see Section 2.4).

Figure 3 gives a UML [28, 29] model of the software and documentation artifacts of the system. We developed the model by examining project documents and interviewing project personnel. Space does not permit us to explain the entire model in detail. Instead, we highlight some relevant portions of the model.

Requirements for releases are separated into *user requirements* and *release requirements*; the latter dictate concerns related to project management of the release process, such as the schedule and the assignment of organizational responsibilities. User-visible services are referred to as *services*. Services are assigned to project managers who oversee their design, implementation, documentation, and testing.

Each service is in a one-to-one correspondence to a *service requirements* document. Service requirements modularize the functional and project management requirements, documenting the user-visible service to be added or modified in the release. Thus, the concept of service neatly corresponds to our notion of a feature, incorporating both the solution-domain orientation and requirements clustering aspects of our definition (see Section 2.1).

Having confirmed the central role that features play in structuring evolutionary changes to the software and the assignment of work activities to perform those changes, we next sought to understand the extent to which the development process exploits the feature-oriented nature of the software product. It turned out that the development process is organized in a traditional way, focusing on the artifacts that are produced in every phase of a waterfall-like life cycle. Requirements documents are written by means of simple word processors and are stored in directories whose

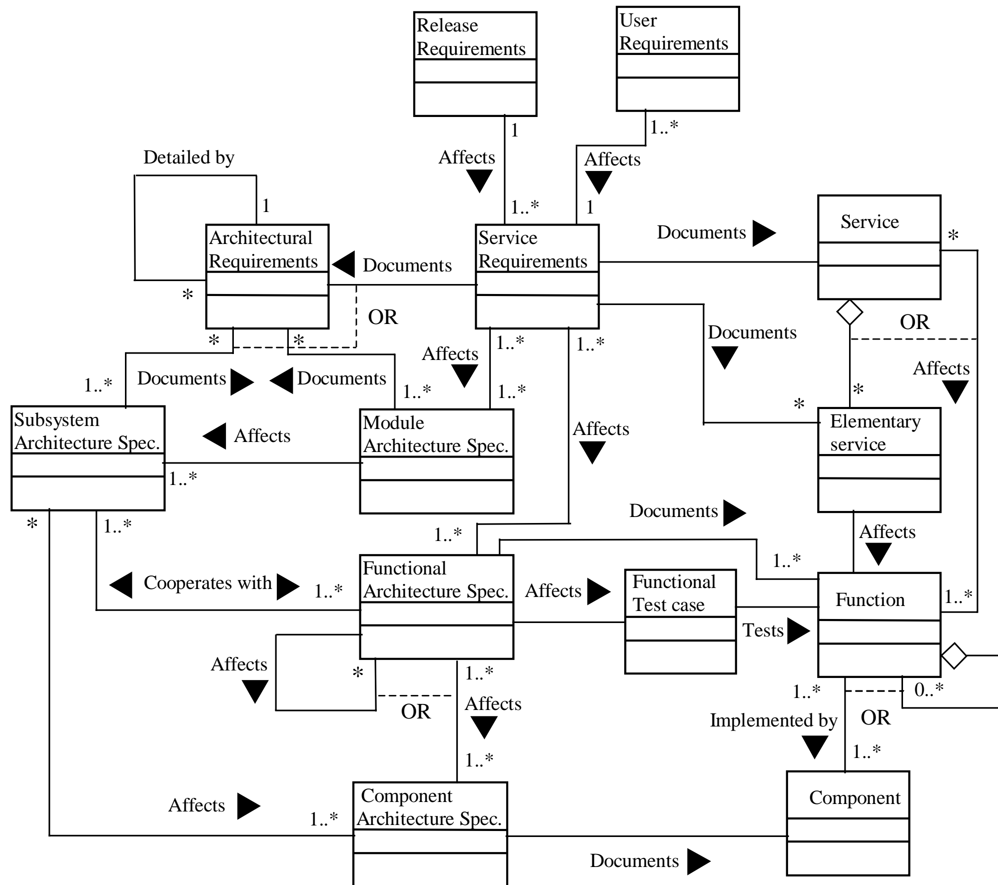

Figure 3: UML Model of Italtel's Telephone Switch Software Artifacts.

structure reproduces the hierarchical structure of the contents. Relationships can be traced in a top-down fashion by navigating explicit references to the names of the lower-level documents. Only in a limited number of cases are references represented bidirectionally. As a consequence, it is very hard to determine to which performances a given function or component contributes.

The tools employed in the project are traditional and general purpose, having no particular tie to the feature context in which they are employed. We thus set about to investigate how one such tool, the configuration management system, could be better integrated into the context. We first defined a set of requirements for configuration management support of a feature-oriented development process. These requirements address four basic goals of configuration management.

- **Identification**: Identify and classify the system artifacts and the relationships among them.
- **Control/Process**: Control access to system artifacts and ensure that all changes adhere to

a desired software process.

- **Construction**: Determine and build valid product configurations.
- **Status**: Record significant events within a development process and provide information to track the system's evolution.

For example, the configuration managment system must identify the set of features currently available, the set of versions for each feature, the set of modules that implement each feature, and the set of test cases associated with each feature. The developer must be able to retrieve all the artifacts associated with each feature and to guarantee some consistency constraints when changes are made to those artifacts.

We next developed an evaluation framework that could be used to indicate the effort involved in realizing a feature orientation within a given configuration management system. In particular, for each activity and structure identified in a requirement, we characterized the support provided by the configuration management system as follows.

- **Native**: Feature semantics are built into the system.
- **Direct**: Feature semantics can be supported by configuration or interpretation of an existing system facility or facilities.
- **Indirect**: Feature semantics can be supported by scripts or programs that use the facilities within the system and that can guarantee the preservation of system constraints.
- **Inadequate**: Feature semantics must be supported by scripts or programs that cannot be prevented from violating system constraints or that require duplicating managed information outside of the system.

We evaluated six commercial systems and found that while none of them provided native support for any of the required capabilities, a few did provide direct support for several of the required capabilities. Those systems therefore allowed, with some moderate tailoring, a higher degree of integration of feature orientation with the current Italtel development process than the system currently used by the development organization. The full results of this study are reported elsewhere [34].

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 life-cycle artifacts and activities. We argue that there is a growing need for feature engineering

methods and techniques able to support a disciplined projection of 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, the notion of feature engineering has the potential to improve system understanding by raising the level of abstraction consistently across the life cycle.

This paper is a first step towards the development of feature engineering. We have presented a provisional model for features in terms of their relationships to other artifacts. Moreover, we have explored how feature engineering affects life-cycle activities, including requirements engineering, testing, configuration management, and reverse engineering. This has been accomplished by analysing the state of the art in the field and by evaluating our own experiences in real industrial projects. In particular, we have briefly discussed the results of a study we carried out to assess the maturity and effectiveness of a large software house developing telecommunication software. As a conclusion, we argue that the concepts involved in feature engineering cut across the entire software life cycle, 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.

Notice that even if the notion of feature and our own experience originate from concepts and techniques defined within the context of telecommunication software, the validity of the observations presented in this paper are not limited to that domain. Indeed, the advent of component-based software development makes the notion of feature engineering of paramount importance in all the application domains where software is used. Component-based development means development by integration of different chunks of functionality. Thus, feature engineering is *de facto* a crucial methodological constituent of any component-based development method.

Certainly, this paper does not provide final solutions. It aims at laying the foundations to address the problem effectively. In this respect, the framework presented in this paper can be further improved. For instance, features are treated as undifferentiated from each other. In complex software systems, features will exist within hierarchies organized by 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. 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. We are carrying out some experimentation on this topic, as part of our ongoing work to explore and address the issues and problems of feature engineering.

Acknowledgments

This work benefited from discussions with David Rosenblum of the University of California at Irvine. The information concerning Italtel's software process and products was kindly provided by Giorgio Comparin.

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] AT&T Network Systems. *5ESS® Switch Global Technical Description*, September 1991. Issue 3.

[3] D. Batory and S O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] A.M. Davis. The Design of a Family of Application-Oriented Requirements Languages. *Computer*, 15(5):21–28, May 1982.

[10] C.G. Davis and C.R. Vick. The Sofware Development System. *IEEE Transactions on Software Engineering*, SE-3(1):69–84, January 1977.

[11] R.G. Day. *Quality Function Deployment*. ASQC Quality Press, Milwaukee, Wisconsin, 1993.

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

[13] M. Fowler. *UML Distilled.* Addison-Wesley, Reading, Massachusetts, 1997.

[14] H.V. Gomaa, H.V. Sugumaran, C. Bosch, and I. Tavakoli. A Prototype Domain Modeling Environment for Reusable Software Architectures. In *Proceedings of the Third International Conference on the Software Reuse*, pages 74–83. IEEE Computer Society, November 1994.

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

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

[17] 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.

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

[19] M. Jackson and P.Zave. Distributed Feature Composition: A Virtual Architecture for Telecommunications Services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.

[20] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Sofware Engineering A Use Case Driven Approach.* Addison-Wesley, Reading, Massachusetts, 1997.

[21] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, Pennsylvania, 1990.

[22] D.O. Keck and P.J. Kuehn. The Feature and Service Interaction Problem in Telecommunications Software Systems: A Survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, October 1998.

[23] Y.-J. Lin and M. Jazayeri. Guest Editoral: Introduction to the Special Section on Managing Feature Interactions in Telecommunications Software Systems. *IEEE Transactions on Software Engineering*, 24(10):777–778, October 1998.

[24] H. Ossher and W. Harrison. Combination of Inheritance Hierarchies. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 25–40. Association for Computer Machinery, October 1992.

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

[26] 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.

[27] W.N. Robinson and S. Pawlowski. Surfacing Root Requirements Interactions from Inquiry Cycle Requirements Documents. In *3rd International Conference on Requirements Engineering*, pages 82–89. IEEE Computer Society, April 1998.

[28] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1998.

[29] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, 1998.

[30] R.W. Krut, Jr. Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology. Technical Report CMU/SEI-93-TR-01, Software Engineering Institute, Pittsburgh, Pennsylvania, July 1993.

[31] M. Sitaraman. Performance Parameterized Reusable Software Components. *International Journal of Software Engineering and Knowledge Engineering*, 2(4):567–587, October 1992.

[32] 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.

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

[34] C.R. Turner, A. Fuggetta, L. Lavazza, and A.L. Wolf. Evaluating Support for Feature-Based Development in Configuration Management Systems. Technical Report CU-CS-875-98, Department of Computer Science, University of Colorado, Boulder, Colorado, November 1998.

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