Studying the Evolution and Enhancement of Software Features

Idris Hsi[1], Colin Potts[2]
College of Computing
Atlanta, Georgia 30332-0280, USA
[1] +1 404 385 1101, [2] +1 404 894 5551
idris@cc.gatech.edu, potts@cc.gatech.edu

Abstract

The evolution and enhancement of features during system evolution can have significant effects on its coherence as well as its internal architecture. Studying the evolution of system features and concepts across a product line from an external or problem domain perspective can inform the process of identifying and designing future features. We show how we derive three primary views, morphological, functional, and an object view, from the user-level structures and operations of a system, using a case study of Microsoft Word's evolution. We show how these views illustrate feature evolution over three versions of Word. Lastly we discuss the lessons learned from our study of feature evolution.

1. Introduction

"Feature creep" is a phenomenon of system evolution where successive releases of a product not only grow in size and complexity, but also show a reduction in the conceptual homogeneity or intellectual coherence of the product as experienced by the user. Thus a text editor may become a page layout program, a document management system, a knowledge-based authoring tool. Just as modules or lines of code are size units for software architecture or implementation, respectively, features are the units of software function or usefulness.

Up to a point, more features are better than fewer, and it is a matter for design judgment and human-factors evaluation to decide when a product has grown too big to be useful or usable. But when this point is reached, engineering questions arise such as whether it will be possible to separate a given feature cluster (e.g. spell-checking) into a separate module so that users can plug in new versions or select among "lite", "professional" or "enterprise" editions.

More significantly, given that a major goal of modern software engineering is the assembly and directed evolution of systems from pluggable components [5], we would like to be able to anticipate these questions far in advance so that we may predict where significant feature growth is likely to occur or prove problematic in the future. Or, alternatively, if an organization is planning a new product line, it is only sensible to analyze the feature space that the product line will occupy, so that component-based assembly can be planned from the outset.

In this paper, we present a view of feature evolution that is defined exclusively in terms of *user-accessible features* and concepts. This is not to argue that software architecture is unimportant to evolution. Obviously it is. Rather, we are claiming that the terms used in questions such as "can we replace X?" should ultimately be couched in the vocabulary of the problem domain and not that of software architecture. "Checking spelling" is what it is whether it is done with a dictionary and blue pencil or an online spell-checker. It is the coherence and integrity of the activity of checking spelling, not the fact that there is a module in the design documentation or a recovered design abstraction re-engineered from the code called the "SpellChecker Module" that makes spell-checking a plausible substitution for "X" in the question above.

Given that features make sense in problem-domain or user-activity terms, we would like to be able to depict the *feature architecture* of a product independently (at least initially) of the underlying software architecture. If we want to find which modules in the architecture are implicated in spell-checking, then the very question presupposes that spell-checking is a sensible feature-oriented abstraction in the first place.

Our term "feature architecture" may sound like a "domain model." In domain analysis, application domain knowledge is modeled independently of systems to support the forward engineering (including maintenance and evolution) of product families [4,5,7]. However, the source of this application knowledge is generally domain experts or the intuitions of the designer. When existing systems or product families are the starting point for an integration or evolution project, as is more after the case, it is necessary to use the current system as a source of the 'theory' of its domain. Previous research into reverse engineering has adopted this approach [4,7] mainly from a starting point of code and code-level documentation. The forward- and reverse-engineering approaches to domain modeling differ not only in their practical aims but also in what the resulting domain model represents. In forward-engineering, a domain model is a normative, expert-generated model of what the problem domain is like. It thus constrains the software architecture by *prescribing* a view of the problem domain but does not reflect it. The reverse-engineering approach takes a domain model to be a *description* of the problem domain exhibited by the current product, not a prescription imposed from outside.

Our approach to feature architecture takes the reverse engineering approach, but differs in one crucial respect from the previous approaches: We reconstruct externally relevant feature objects and operations from the morphology (externally visible interfaces) of an application, and developed a reverse-engineering version of domain analysis. There is no single domain model, but rather a tripartite view of the domain/product features as follows:

- The *morphological* view is the user-visible analog for feature architecture of the source code content of a software architecture. It consists of the user-interface composition and navigation structure.
- The *functional view* is the description of what the features do. A thorough analysis of functionality would require a detailed model of interactions based on data flow or control abstractions. In this paper, we restrict ourselves to enumerating the *operations*, the activities that the system performs.
- The *object view* is a description of the subject-matter of the feature. Like an object model produced during software design or an information model for database design, the object view consists of static relationships between objects in the problem domain. In the case of the feature architecture, however, the objects are derived from user-visible phenomena, especially the user interface components from the morphological view. The objects in the feature architecture may be correlated with the objects underlying the implementation if it is object-oriented or the data structures and files if it is not, but they need not be. Again, it is the problem domain that makes the products' objects appropriate or inappropriate, not the fact that they are to be recovered from the code.

Thus these three views of feature architecture are derived without knowledge of the source code and without recourse to specialist domain expertise. Rather, the feature architecture encompasses the *working domain model* and functional repertoire of the existing product. "Spell checking" can therefore be thought of, if rather fancifully, as what a word-processing product tells us about spell-checking.

Studies of system growth or evolution from the software-maintenance perspective (e.g. [10]) address evolution as changes in the size and relational complexity of the code base of the product. To our knowledge, there have been no comparable implementation-independent studies of feature evolution, where feature architectures have been objectively defined and measured.

In addition to its potential practical value in helping us to understand the dynamics of feature evolution, we think such studies have an intrinsic interest. Significant software products affect society in numerous obvious and subtle ways, and it is appropriate for software engineering to undertake precedence studies [8] similar to those of architecture and urban planning, two professional disciplines whose products have similarly wide-ranging effects. Tracking the evolution of features in office products, for example, could tells us much about how technology drives social processes, how technology infrastructure and social phenomena affect what features grow at what epoch in a product's life history, and how the two actors in this interaction—technology and its contexts of use—co-evolve.

To this end, we are studying office productivity packages, time-management and scheduling packages, computer games, camera controls, and telephony features [2]. In this paper, we present a case study examining three versions of Microsoft Word for Windows.

This paper is a prospectus and example of this approach to studying feature architectures and their evolution and its possible value in planning future feature evolution through component assembly. Section 2 builds on this approach and describes a specific methodology for deriving the three views of feature architectures: System Morphology, System Operations, and the System Object Model. The results of applying this methodology are discussed in Section 3, in which feature evolution is documented for three versions of Microsoft Word. In Section 4, we discuss the findings of the case study. Lastly, we conclude with some benefits of the approach for forward-engineering of software in practice.

2. Three Views of Feature Architecture

2.1 System morphology

Morphology is the study of the form and structure of an organism without consideration of function. An application's morphology is the structure that organizes its features, consisting of user interface elements including menu items, user input device commands, and information displays. These provide *portals* through the external to the domain features.

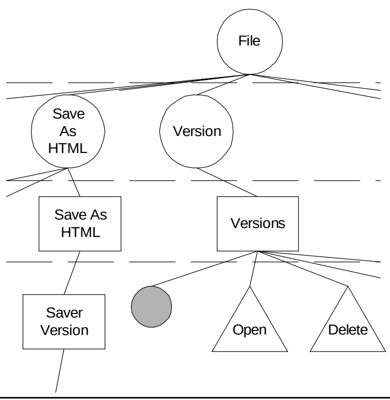

Figure 1. An example of system morphology – a portion of the Word 97 File Menu

We construct the representation without analysis of functionality or design intent by tracing paths through the interface elements and developing them into a graph representation.

Figure 1 shows a portion of the Word 97 File menu morphology. We generated the graph by traversing the File menu, identifying all the menu items listed. The menu's main items are represented as large circles. Rectangles represent dialog boxes. Small circles are simply generic terminators like OK/Cancel. They also become leaf nodes in the graph because they cause the activity to return to the top level morphology. Small triangles are actions specific to that dialog box that also act as leaf nodes. The horizontal dotted lines show how "deep" that particular path reaches. Every time an action invokes another interface structure, the path gets deeper. Other items represented by the morphological view, but not shown in the example, include mode changes, parallel dialog structures, toolbars, mouse actions, displays, and menu bars. Items not represented by this view include dialog box details such as radio buttons, selectors, dials, and so on. We chose not to represent the smaller structures within the dialog boxes to simplify the representation.

2.2 System functional view

The functional view consists of an enumeration of all the operations that the user can call through the normal operation of the system. In the absence of documentation or program specifications, we uncover these by traversing the morphological views, observing, sometimes inferring, the operations that the program performs. Fortunately, for this case study, we were able to use the lists of operations that MS Word provides to program macros and set button and keyboard shortcuts.

After we obtain the list of operations, we categorize them by whether they are old, new, or have been removed since the last release, which interface structures are used to call them, what object they affect, and a description (if needed) of the function's action. At this point, we define an object to be something that can be accessed by a user through the system's morphology or a system operation. Most of the objects can be taken directly from the operation's name and behavior. Occasionally, they have to be inferred from the morphology and action they perform. An operation called "Exit" for example, infers an Application object that you exit from.

Table 1 shows all the operations associated with the bulleted list in Word 95. From Word 2.0 to Word 95, there have been three new operations added and none removed. The fact that the older operations have no interfaces connected to them implies that they are unused in the later version and may be present for backwards compatibility or to support user-level macros created in Word 2.0.

Table 1. Functional view of Bulleted List Operations in Word 95. (Not shown are the descriptions of the operations.)

Name	Status	Menu Access	Toolbar Access	Input Device	Object
ApplyListBullet	New	None	none	Ctrl+Shift+L	Bulleted List
FormatBulletDefault	New	None	Formatting	None	Bulleted List
FormatBulletsAndNumbering	New	Format	None	Right Mouse Button	Bulleted List
ToolsBulletListDefault	Old	None	None	None	Bulleted List
RemoveBulletsNumbers	Old	None	None	None	Bulleted List

2.3 System object model

Using the objects derived from the system operations and morphology, we can build a modified entity-relationship diagram that describes how those objects interact to form the underlying domain model. There are three types of relationships that we examine because of their relevance to the work product domain.

- *has* – Object A *has* Object B if, very simply, A can physically contains B or can possess B as a sub-property or concept. This is an optional, not a mandatory relationship. The *has* relationship is also directed. Object A must be located higher than B in the morphological hierarchy for Object A to *have* Object B. The relationship is derived from a morphological connection between A and B but only the closest connection is considered in the hierarchy. For example, a page can have words and a paragraph can have words. But pages must first have paragraphs before they can have words. So in an object representation, we represent a *has* relationship between page and paragraph, and one between paragraph and word but not page and word.

- *strictly contains* – Object A *strictly contains* Object B if Object A must have Object B to exist. The *strictly contains* relationship is a subset of the *has* relationship. Fonts can optionally have a Font Underline but must have a Color. Therefore, Fonts *strictly contain* Font Color. *Strictly contains* relationships are important because they help to define tight relationships between objects. A change to this kind of relation can imply a fundamental conceptual change to the parent object.
- *type of* – Object B is a *type of* Object A if A, as the morphological parent references B from a set of equivalent objects. Fonts can have a Font Underline but in the morphology, there are 11 different *types of* Font Underlining.

Based on these relationships, we can isolate system concepts which we call *teleons*, from the Greek word *teleos* meaning goal, using the following definition.

- *teleon* – A *teleon* parent is any node that has at least one child resulting from a *has* or *type of* relationship. The teleon is then formed by tracing the graph until a node with a shared ancestor or subtypes is reached. That last node is included in the graph and the trace ends. The resulting subgraph is the complete *teleon*.

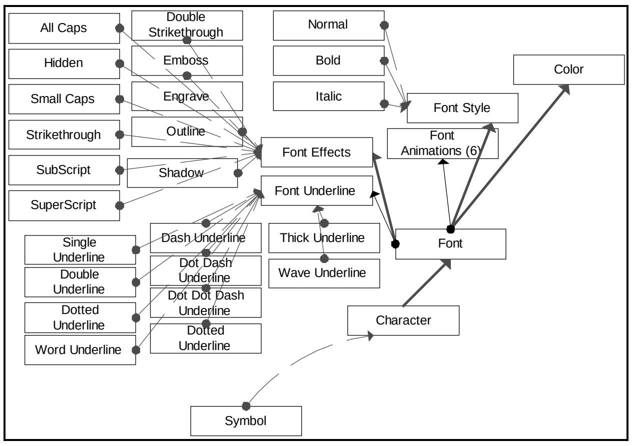

Figure 2. Object view of the character teleon in Word 97.

Figure 2, shows the Character teleon. The dashed lines represent *Type Of* relationships, the thin lines represent *has* relationships, and the thick lines represent *strictly contains* relationships. The Character teleon consists of the Symbol, Character, and Font nodes. It also contains subteleons such as Font, which consists of Font Effects, Font Underline, Font Animations, Font Style, and Color. Color, in this representation, is not a teleon because it does not have a child.

3. The evolution of MS Word

We use these system descriptions to study how an application evolves in structure and functionality over the lifetime of the product line. In addition, this approach has revealed a relationship between these three views of the system that suggests some feedback mechanisms that impact this evolution. Here we examine the evolutionary trends that we have observed in MS Word.

3.1 Morphological evolution

Evolution of system morphology has two implications. First, that there is more underlying functionality to be accessed and second, that more portals are being opened to frequently used operations. We observed two basic trends in the morphological evolution of Word: changes in the size and complexity of the overall morphological structure and to the types of primary interfaces used in the morphology.

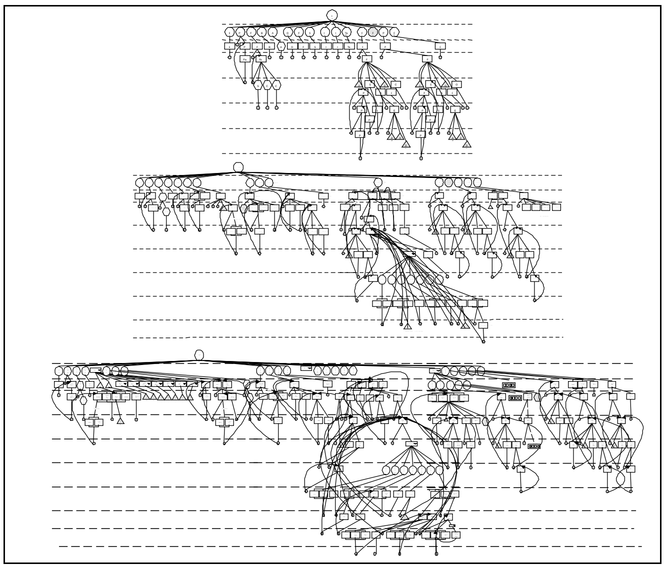

Figure 3. An overview of the graphs representing the Insert Menu morphology for Word 2.0, Word 95, and Word 97, respectively.

Figure 3 shows the evolution of the Insert menu over the three versions of MS Word. The menus (with the exception of the File menu) visibly grow in depth and breadth reflecting an increase in the types of objects that can be contained in a document. The curved lines from the bottom most nodes back to a middle layer node represent a return to a previous dialog box in the trace. So in addition to growing in overall size, there are now more loops in the graph. While the Insert menu is the most pronounced example of growth that we encountered, similar behavior can be seen across most of the other morphologies.

The other basic trend that we observed was the changes to the types of primary interfaces. For example, Word 95 and 97 employ more mode shifts and toolbars to accomplish tasks. Also, Word 97 departs from redundant accessibility, where a function could be reached from menu and toolbar. Instead, it employs unique accessibility, or specialized portals, where a function can only be reached from a particular toolbar that can be accessed during a particular mode.

3.2 Functional evolution

The number of operations provided by each word processor significantly increased over versions. This seems to be a reasonable result given the changes to the morphology: more portals implies more operations on average.

Table 2. Function Growth in MS Word

Version	# New Operations	# Kept from last version	# Removed from Last	% Growth	Total # of Operations
2.0	311				311
95	362	253	58	97%	614
97	383	572	42	56%	955

Table 2 shows a brief quantitative analysis of how the numbers of operations evolved. The removed operations were actually renamed, consolidated, or relocated to other parts of the operating environment. For example, Word 2.0 used to have file management capabilities and Word 97 uses Visual Basic to manage its macros.

The numbers imply that Word experiences a steady, calculable growth in functionality. However, further examination reveals that almost half of the new operations in Word 97 are related to graphics teleons, specifically 3D drawing objects, 125 new drawing objects, and Word Art. Some of these operations also support the management of these drawing objects. Other objects, such as Tables or Bulleted lists, see a few new operations that extend their capabilities but not significantly. Our general finding is that functional evolution in the MS Word product line is not evenly distributed, as one might see in an application that experiences monotonic, conservative growth.

3.3 Object evolution

To help constrain our analysis for this study, we chose to limit our object model to the electronic and paper document that Word produces. We did not look at the window and application mechanisms or the supporting operations, such as spelling and grammar checking.

After deriving the object representations, we noticed some general tendencies in the object model. With the exception of the character/font teleons from Word 2.0 to Word 95, older teleons rarely changed their existing subgraph. Teleons changed by either increasing their potential space or by increasing the number of different types associated with it.

For example, Paragraph is a very important teleon. Word 2.0 has 9 nodes in the Paragraph teleon. Word 95 has 18. Word 97 has 21. In general, Word changed significantly from Word 2.0 to Word 95 but the extensions to the Paragraph teleon in Word 95 were almost all new teleons that could now be contained in a paragraph, such as a Cross Reference. Word 97 simply adds three more items, such as Hyperlink, to this list. What this implies is that Paragraph is becoming a more stable teleon in definition and is growing in capability.

The other behavior, increasing the number of types, can be seen in Word 97's Drawing Objects (added 115 objects). The functional growth described earlier is partly the result of adding over a hundred drawing objects. Each object needs a minimum of one function to be used in a document. Other things that developed more types included Field, Font Effects, Document, and Links.

We also examined the conceptual evolution of the Word document. Table 3 shows that the growth of new teleons over the versions. If we removed "sub-teleons", such as 3D Lighting (a sub-teleon of 3D object), we're left with an evolutionary model that indicates conservative growth – adding a small number of teleons to the document per release.

Table 3. Conceptual Evolution of the Document in MS Word

Word 2.0 Teleons	Word 95 – New Teleons	Word 97 - New Teleons
Annotation	**Caption**	**3D Direction**
Border	**Cross-Reference**	**3D Lighting**
Character	**Database**	**3D Object**
Column	**Drawing**	**3D Surface**
Document	**Drawing Object**	**Comment**
Envelope	**Font**	**Font Animation**
Field	**Font Effects**	**HTML Document**
Font Style	**Font Underline**	**OCX Object**
Footer	**Form Field**	
Footnote	**Heading**	
Frame	**List**	
Header	**Note**	
Index	**Numbering**	
Line Numbering	**Revisions**	
Object	**Table of Authorities**	
Page	**Table of Figures**	
Paragraph		
Picture		
Section		
Shading		
Style		
Summary Info		
Tab Alignment		
Table		
Table Cell		
Table of Contents		
Tabs		
Word		

4. What changed and why?

From the data, we know that none of the teleons vanish from the new domain model. In fact, they become more entrenched, growing more connections to different objects and morphologies over time. Intuitively, one can say with some confidence that the teleons outlined in the Word 2.0 represent a set of teleons that are core to a document produced in the MS Word family. In fact, the objects introduced in Word 97, with the exception of HTML Document, seem to have only peripheral relevance to what you might expect to find in a typical document.

In order to analyze this evolution, it is important to be able to separate the changes that are a result of technological advances in hardware or implementation from those that represent fundamental changes to the concepts embodied in the software. The former type of change tends to be primarily morphological in nature: better graphics, new widgets, and new interactive devices. We consider these changes to be superficial in nature. They alter the outward appearance of the application and sometimes give the illusion of significant enhancement. We are more concerned with the evolution of the software's concepts.

4.1 "True" conceptual evolution

Concepts that evolved through the different versions of Word did so almost exclusively by the addition of new subtypes of the object or the addition of new contained objects. However, these new objects were almost never strictly contained and the concept structure grew monotonically without restructuring. This suggests an operational method to deliver stable and core features: they grow but do not have to be restructured; and they accrete new parts, but not in a way that necessarily affects what already exists.

Studies of conceptual evolution in other areas reveal a similar phenomenon. For example, Thagard [11] shows that the conceptual schemas of science, before and after major paradigm shifts are structured differently, with different classification and containment relations holding between concepts. However, normal science proceeds more routinely by the addition of specialized and component concepts.

This suggests, by analogy, that occasional changes to core feature concepts are likely to have radical effects, either immediately or in subsequent releases. An example of this may be present in the inclusion in Word 97 of the 'HTML document' – which is not merely a new kind of document but is likely to affect the concepts of document sections, pages, etc. It is, of course, more common for changes to be made for functional reasons than because the domain concepts have changed [1,6,14]. Such shifts do, however, occur occasionally, and it is vital to identify the objects most vulnerable to radical change.

Older features, representing the "core" of the application experience less change and evolution over the lifetime of the product line, stabilizing with each version. Newer features tend to be added to the periphery, either as small extensions of existing concepts or as large "clumps" of functionality that expand the overall domain.

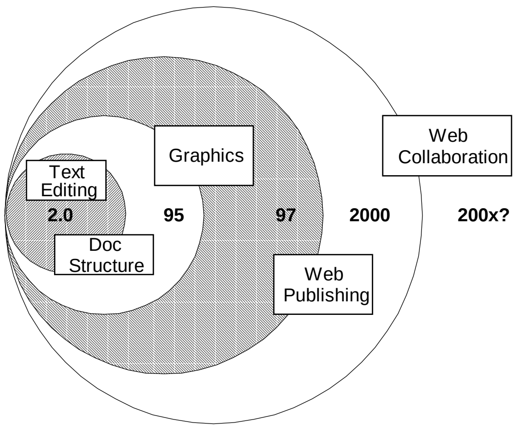

Figure 4. Feature "clumps" in successive versions of MS Word

Figure 4 shows how Word 2.0's text editing capabilities were extended by changes to document structure management in Word 95. Then Word 97 added graphics capabilities. Our initial work with Word 2000 shows additional Web publishing features as its large clump. In each case, the clumps grew from existing concepts in previous versions. Using this heuristic, we can hypothesize that the next version of Word will have more Internet collaboration capabilities, extending the existing email and web publishing capabilities.

4.2 Morphological and conceptual changes

While morphological changes have important implications for the efficiency and usability of these applications, they do not tend to alter what we consider to be the core teleology of a product. Changes to teleons naturally affect the morphology, reflecting the intuition that the "deep structure" of the application affects the "surface structure." We would therefore expect changes in the morphological scale and complexity of the product to reflect the underlying functional and object-oriented complexity. However, the MS Word evolutionary record shows that its morphological changes far outstripped any underlying changes.

It would be an exaggeration to say that the user interface of Word has become extremely rich, whereas the product has not evolved substantively; our analysis of MS Word does show a large growth in morphology and only a small growth in the number of teleons. Some changes may therefore be imposed by interface efficiencies. In the case of user interfaces, this could lead to user opinions that a product had become complex and "bloated" far beyond its actual functional and conceptual growth.

Features are composed of these teleons and objects. In order to understand the differences between the small changes to the teleons and the dramatic changes to the morphology, we need to examine how introducing objects to create features or enhance existing ones can affect operations and morphology. Consider the simple example of a single object, with one function, accessed by a single morphological port shown in Figure 5.

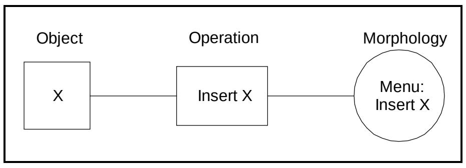

Figure 5. 1:1:1 correspondence

Many objects in an application tend to have attributes, options, and capabilities, each of which requires a function to use it properly. If the user wants to be able to change a Font Style from Normal to Bold, an extra function is needed. This situation is better portrayed by Figure 6 than the simple correspondences of Figure 5.

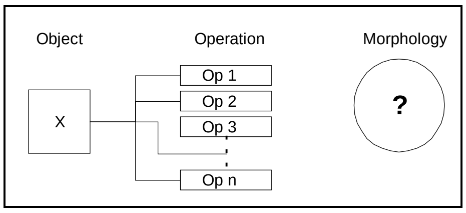

Figure 6. 1:n correspondence between object and operations.

But in order for these operations to be useful, they require some form of accessibility from the system morphology. Important or frequently used operations may also require multiple portals to increase accessibility. Figure 7 shows how the final morphology grows from adding a new object.

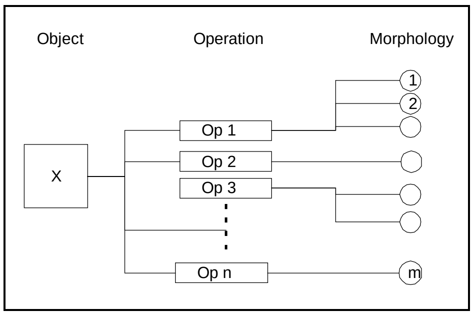

Figure 7. 1:n:m correspondences with object in system

This illustration shows how introducing or extending a features can have tremendous impacts on the overall morphological complexity of the system. The rapid structural changes in the morphology of Word compared with the relative stability of its core features reinforces the standard architecture guideline to decouple user interface code from application features.

5. What is in a feature?

In proposing a feature architecture independent of implementation architecture, we have assumed a traditional function/data split in describing the deep structure of the application. Our description of features consists of two representations: a modified object model showing the structure of a teleon as a network of related or contained objects, and a list of operations that create, access, update, or delete such objects.

Object structuring, particularly ownership and containment, is an appropriate organizing principle for a word processor's feature architecture, because word processing is a "work piece" problem frame [9]. That is, the software features are responsible for creating an artifact that can subsequently be inspected and manipulated but which does not exist independently in the world outside the software or change independently of it. A different set of basic categories would probably be more appropriate for control, information management or transform applications (which correspond broadly to Jackson's control, information system and JSP problem frames).

For example, control features, such as setting reminders in real time or controlling the operation of a device like a camera, have the achievement of goals as a primary category for these application features. These have to be modeled as the achievement of event-recognizing and phenomenon-affecting goals [2,12].

6. Feature Coherence

The application itself is a source not only of the teleons and operations represented in its features but also their relative centrality and connectivity. Earlier we argued that 'core' and peripheral' teleons evolve differently, but did not define these terms independently of their age ("core" teleons being the earliest).

We are investigating graph-theoretic and statistical clustering techniques for quantifying and presenting teleon structure independently of their evolution, an approach complementary to Waters's use of lattice-theoretic techniques ("concept analysis") for reverse engineering architectural concepts [13].

Extending existing features or adding new ones requires developing new associations with the current features. Older features tend to be more entangled with associations and will therefore require more effort to modify in later releases. New features with effective conceptual relationships to existing features may also require many associations with them. This difficulty may account for the tendency to supply new features that only loosely associate with old features and are thus peripheral to the core teleology of the application.

A major practical consideration for developers is how to manage the design and architecture of a version to allow for the coherent evolution of its features. Developers planning to evolve systems need to design and structure architectures to support such coherent growth. We have argued that a more principled definition of feature architecture as a combination of morphology, functional model, and object model is a viable way to describe a product's feature set independently of its code architecture and that such planning of feature evolution could be framed in these terms.

7. Acknowledgements

We thank the reviewers for their helpful comments.

8. References

[1] Abowd, G., Ertmann-Christiansen, C., Goel, A., et al., "MORALE: Mission Oriented Architectural Legacy Evolution", *Proc. International Conference on Software Maintenance'97* (Bari Italy, 1997), IEEE Computer Society Press, 1997, pp. 150-159.

[2] Antón, A. I. and C. Potts, "The Use of Goals to Surface Requirements for Evolving Systems", *Proc. 20th International Conference on Software Engineering (ICSE '98)*, (Kyoto Japan, 1998), IEEE Computer Society Press, 1998, pp. 157-166.

[3] Anton, A. I and C. Potts, "Requirements Engineering in the Long Term: Fifty Years of Telephony Evolution", Accepted to *International Workshop on Feedback and Evolution in Software and Business Processes (FEAST 2000)*, (London UK, 2000).

[4] Arango, G., "Domain Analysis: From Art Form to Engineering Discipline", *Proc. International Workshop on Software Specification and Design*, (Pittsburgh PA, 1989), IEEE Computer Society Press, 1989, pp. 152-159.

[5] DeBaud, J.-M. and K. Schmid., "A Systematic Approach to Derive the Scope of Software Product Lines", *Proc. International Conference on Software Engineering*, (Los Angeles CA, 1999), IEEE Computer Society Press, 1999, pp. 34-43.

[6] Easterbrook, S. and B. Nuseibeh. "Managing Inconsistencies in an Evolving Specification", *Proc. RE'95: Second IEEE International Symposium on Requirements Engineering*, (York UK, March 1995) IEEE

Computer Society Press, 1995, pp. 48-55.

[7] Fischer, G., "Seeding, Evolutionary Growth and Reseeding: Constructing, Capturing and Evolving Knowledge in Domain-Oriented Design Environments", *Automated Software Engineering*, Boston, MA, Kluwer Academic Publishers, **5**(4), 1998, pp. 447-464.

[8] Hillier, B., Space is the Machine: A configurational theory of architecture. Cambridge, UK, Cambridge University Press, 1996.

[9] Jackson, M.A., Software Requirements and Specification, Reading, MA, Addison-Wesley, 1995.

[10] Lehman, M. and L. Belady, Program Evolution, New York, NY, Academic Press, 1985.

[11] Thagard, P. Conceptual Revolutions. Princeton, New Jersey, Princeton University Press, 1992.

[12] van Lamsweerde, A. and E. Letier, "Integrating Obstacles in Goal-Driven Requirements Engineering", *Proc. 20th International Conference on Software Engineering* (Kyoto Japan, 1995) IEEE Computer Society Press, 1995, pp. 53-63.

[13] Waters, R. and G. Abowd, "Architectural Synthesis: Integrating Multiple Architectural Perspectives", *Proc. Sixth Working Conference on Reverse Engineering* (Atlanta GA, 1999). IEEE Computer Society Press, 1999, pp. 10-15.

[14] Zowghi, D. and R. Offen, "A Logical Framework for Modeling and Reasoning about the Evolution of Requirements", *RE'97: Third IEEE International Symposium on Requirements Engineering*, Annapolis, Maryland, January 6-10, IEEE Computer Society Press, 1997, pp. 247-2.