Measuring the Conceptual Fitness of a Computing Application in a Computing Ecosystem

Idris Hsi
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280 USA
770 541 0321
idris@cc.gatech.edu

ABSTRACT

Developing computing applications that can match a set of evolving conceptual requirements requires an understanding of the *conceptual fitness* of these applications relative to the domains they purport to serve. We present the *computing ecosystem* framework with its associated concepts, *use niches*, *use potential*, and *activation potential*. We show how the ecosystem framework allows us to characterize the usefulness of an application through the concept of fitness. We propose a method for measuring the fitness of an application using a metric called *ontological coverage*.

We first use a technique called *ontological excavation* that identifies the user-visible concepts from applications and models them in an ontology. We then use a set of use cases to develop a *use case silhouette* on the ontology that allows us to measure the ontological coverage of an application as an initial approximation of fitness to a use niche. We present some examples from case studies showing how use case silhouettes can be used to measure the fitness of an application and conclude with some proposals for future work.

KEYWORDS

software engineering, software evolution, usefulness, computing ecosystem, ontological excavation, conceptual fitness, use niche, use potential, activation cost, activation energy

INTRODUCTION

"Perfect" computing applications are illusory. At best, they are perfect tools for a moment in time: round pegs for round holes. Applications optimized for a particular purpose tend to be poor at adapting to changing circumstances. Often the environment will change faster than the system's ability to adapt: a company may change marketing focus or adopt a new set of practices; customers that realize the potential of their new tool may decide that they need it to do more. Thus the initial requirements for the application slowly fall out of synch with the actual requirements and the round peg no longer fits because the hole has changed shape.

Software developers have attempted to solve this problem by adding functionality to newer versions of applications. While the new functionality does serve to meet the new set of requirements, work by Lehman and Belady have shown that the systems increase in complexity and become more difficult to maintain when evolved in this manner [19, 20]. If the application grows past a certain size, its users begin to have difficulties with it. These difficulties include applications having too many features, automated features that are not desired, and problems with navigating the user interfaces to find the desired features [6, 18, 23, 25, 26]. Users describe such systems as *bloated.* We formally define 'bloat' as the description applied to applications when it possesses a disproportionate number of unnecessary features that interfere with normal or desired interactions with the application.

At first it would seem that this problem is intractable. These computing applications must continually be adapted to maintain user satisfaction but increasing the functionality perturbs the use of the system. A resolution to this conundrum can be found in a biological metaphor, borrowed from Darwin's studies of evolution, that allows us to measure an application's potential success in terms of its *conceptual fitness* relative to a *use context*.

CONCEPTUAL FITNESS

Applications are engineered to solve problems in specific use contexts. A use context consists of the external physical (or virtual) environment that contains the computing application and its users, the goals that the combined computing application/user system wishes to achieve, and the various nuances (business rules, customer demand, user and system capabilities) that govern the operation and performance of both environment and goal completion. For example, the use context of a bank

Idris Hsi, "Measuring the Conceptual Fitness of a Computing Application in a Computing Ecosystem", *Proceedings of the ACM Workshop on Interdisciplinary Software Engineering Research (WISER'04) at the 12 Annual Foundations of Software Engineering (FSE-12)*, November 5, 2004, pp. 27-36.

customer database consists of the bank itself, the systems that manage and store the database, the employees charged with maintaining the stored information, and the rules and procedures established by bank management for storing and distributing the data. All use contexts exist within a *problem domain*. Arango and Prieto-Díaz state that a problem domain is a collection of items of real-world information that has “deep or comprehensive relationships among the items of information” and a community that has a stake in solving those problems [1]. Software that has been designed to function in the use context and the problem domain will possess a set of concepts and relationships that we call an *ontology* [5, 11, 12]. The ontology of a computing application can be said to be its theory of the real world. The concepts it embodies determine and structure its *features*, which we define as the *user-accessible behaviors and services implemented by the system*.

Ultimately, users evaluate software on its ability to help them to achieve their goals, whether these goals are for entertainment, productivity, scientific analysis, or industrial application. Thus, engineering concerns aside, the primary challenge for software developers has always been to ensure that their applications have a high level of *usefulness* for their customers and end-users. We define usefulness as *the extent to which an application succeeds in assisting a set of users to achieve a set of goals, relative to the amount of effort required to engage those features.* We distinguish usefulness from *usability* which is an integral but subordinate attribute of usefulness. A useful application with poor usability can still enable users to achieve their goals albeit with great difficulty. An application with little or no usefulness can be extremely usable but cannot help the users to achieve their goals.

Developing useful software requires that developers understand what their users are trying to do in a specific use context and encode that knowledge into the design. Yet an application must possess enough features to be useful to its users but without becoming too complex: a design tradeoff between functional power and conciseness. The features are accessed by the users of the system through its user interface or *morphology,* which is the external presentation of the software. Thus what the software is, how it is presented to the users, and how it functions must ultimately be determined by its ontology. If its ontology does not match the user’s understanding of the problem domain then the application will fail. If the ontology has been modeled incorrectly, relative to the problem domain, then the most advanced techniques in program design, development, and testing will not be able to produce a useful computing application. In other words, *the usefulness of a computing application is determined by the conceptual fitness of its ontology to the domain of the user.*

In biology, an organism’s fitness, as a function of both its survival and reproductive capabilities, depends greatly on the environment or *ecosystem* that it inhabits. Thus, conceptual fitness must also be measured against the environment that express the domains embodied by the computing application. We now propose a framework that we call a *computing ecosystem.*

THE COMPUTING ECOSYSTEM

In biology, ecosystems describe a defined envelope of physical, chemical, and biological processes within a space and time [21]. More generally, an ecosystem is “a system of interacting species in a particular environment” [17]. We formally define a computing ecosystem as *a set of use contexts that use computing to fulfill goals, contained within an environment of interest.* A computing ecosystem can be a single person and a handheld PDA or a multi-national company of database management specialists. In a computing ecosystem, the organisms are the computing functions that users apply towards achieving their goals.

The *biological fitness* of an organism is described as “the ability of an individual to produce viable offspring and contribute to future generations” [21]. In principle, the fitness of an individual organism takes secondary importance to overall genetic fitness of the species, as measured by its population size and, in evolutionary biology, by how many years that species managed to survive over the lifetime of the Earth [9, 10]. But this genetic fitness is really a property that emerges from the individual organism’s abilities to survive and procreate, summed over the entire population of the species. Thus, one cannot understand biological fitness solely by studying the genetic code. One must examine the resulting phenotypic expressions – the *physiological features* of an organism encoded in the gene. Analyzing the fitness of a species requires studying not only the individual organism’s physiological attributes that enable it to reproduce, the most direct contribution to fitness, but how its features enable it to interact successfully with its environment and its fellow organisms in activities like its ability to gather and consume nutrients, escape predation, and maintain homeostasis across climatic variations.

Likewise, for computing applications, code and architecture do not reveal anything about their fitness in a computing ecosystem. The correct unit of study has to be the phenotypic expressions of the software or its features. If an application lacks the features that would make it useful to its users, then it lacks sufficient fitness for it to exist in the computing ecosystem. We now need a more precise characterization of the relationship between an

application's features and concepts and the computing ecosystem.

THE USE NICHE AND FEATURE FITNESS

Biologists tend to think of the habitats of particular organisms in a more narrow context – that of the *ecological niche*. An ecological niche is a physical environment that supplies the food and space required for the survival of a set of species [21]. Species that occupy the same ecological niche will compete for these resources. Over time, only the species that have evolved or adapted a sufficient level of fitness will survive in these niches.

Features in computing applications inhabit *use niches*, a bounded space in the computing ecosystem that contain subsets of the ecosystem's resources and goals. A use niche may be as broad as "document writing" and as narrow as "database sorting". The fitness of a feature in a use niche is primarily determined by that feature's ability to fulfill the requirements of that niche. However, a feature can possess the necessary concepts and functions to occupy a niche but still be unfit due to poor usability, which we characterize as a conflict between the available energy in the ecosystem and the usage cost of the feature.

USE POTENTIALS AND ACTIVATION COST

In a use niche, the resources that allow features to exist can be collectively abstracted to what we call *use potentials*. A use potential is the amount of available energy or effort that the users are willing to expend to activate the features of the application. On the application side, features have an *activation cost* that represents the corresponding amount of energy or effort required to engage those services. For example, a GOMS (Goals, Operations, Methods, Selection) formulation from work in human-computer interaction would measure this activation cost in terms of the number of user interface item selections [7]. If a feature can achieve the goals of a niche and its activation cost is less than the use potential of the niche it will occupy, then it has a high potential fitness.

For example, you could write a document, such as a memo, in a spreadsheet or a word processor. The word processor has a low activation cost for its text processing functions because it has been designed that way. The spreadsheet application has some of the same concepts related to text but has a higher activation cost because its ontology has been designed around the management and analysis of numerical data. For a normal memo, it would make sense to choose the word processor. However, if you wanted to write a memo with charts and graphs displaying financial information, it may be easier to use the spreadsheet application as the word processor will have a higher activation cost for those features.

We now present some work that we have done to characterize use niches, the activation costs of features, and the fitness of an application relative to a use niche.

CASE STUDIES

We have argued that the fitness of a computing application can be described in terms of an application's conceptual fitness relative to the domain of an ontology. While a complete measurement of fitness requires the teleological examination of an application's actual functions to verify that it could achieve the goals in that domain, we felt that conceptual fitness was a sufficient first approximation for this work since the functions themselves have to be derived from the concepts.

We studied three applications: Windows 95/98 CD Player, Microsoft Notepad, and the Protocol Calculator/Calendar device. The Windows 95/98 CD player plays CDs and helps the user manage playlists. The Microsoft Notepad is the default text editor for the Windows operating system. The Calculator / Calendar made by Protocol is a handheld device that implements an alarm clock, calendar, calculator, currency exchange calculator, and countdown timer. The clock also allows its users to view times in sixteen different time zones. All three were chosen for their relatively small feature sets and simplicity.

We first identify the concepts in these applications using a method that we call *ontological excavation*. Using analysis techniques from graph and social network theory, we identified the *core concepts* in these applications. We then obtained a set of *use cases* from the help files or instructions of these applications to characterize a potential use niche of the application. We then use a technique called *use case silhouetting* to measure their *ontological coverage* of the application.

ONTOLOGICAL EXCAVATION

In previous work, we developed this method for excavating the concepts and relationships of a computer application [14, 15]. Ontological excavation uses black-box techniques; the ontology is reverse engineered from the user interface of the application rather than the source code. Black box reverse engineering allows us to identify just the concepts visible to the user rather than the concepts relevant to the application's implementation. We use the external presentation or *morphology* of the application as our unit of analysis.

The steps are:

1. Model the user interface in a *morphological map* of the application's interactors, displays, and containers.
2. Generate a list of morphological elements.
3. For each element, identify the concepts (entity types and attributes) that it invokes.

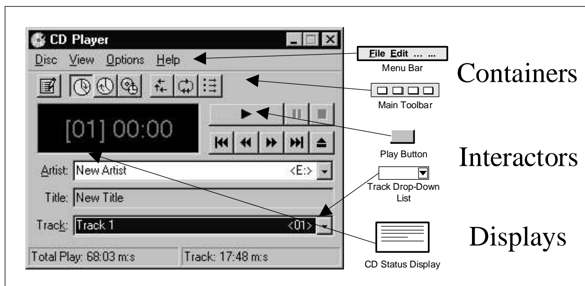
Idris Hsi, "Measuring the Conceptual Fitness of a Computing Application in a Computing Ecosystem", *Proceedings of the ACM Workshop on Interdisciplinary Software Engineering Research (WISER'04) at the 12 Annual Foundations of Software Engineering (FSE-12)*, November 5, 2004, pp. 27-36.

4. Through dynamic interaction with the application, identify the relationships between the concepts.
5. Model the concepts and relationships into a semantic network representing the application's ontology.

In the following sections, we summarize the major steps in excavating the ontology from the application.

3.1. THE MORPHOLOGICAL MAP

Figure 1 – Examples of containers, interactors, and displays from the Windows 95/98 CD Player

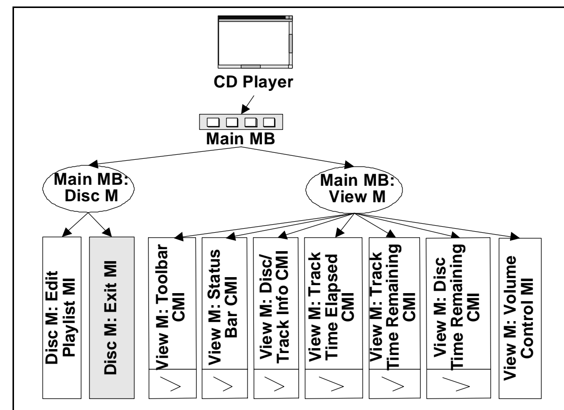


We model the user interface in a morphological map. This map consists of the interface's interactive elements or *interactors* (e.g. buttons, text fields, check boxes), *containers*, a morphological element that contains and structures interactors (e.g. windows, dialog boxes, toolbars), and *displays*, morphological elements that present both static and dynamic data about the computing application's states to the user. Figure 1 shows the Windows 95/98 CD Player application along with examples of containers, interactors and displays.

We build this map by traversing and activating all the user interface elements in a systematic, depth-first fashion. Each element is represented by a visual icon and given information corresponding to its label in the user interface. These visual icons are linked using arrows to show either their container (e.g. a toolbar containing buttons) or their point of activation (e.g. a menu item opening a dialogue box). A portion of the Windows 95/98 CD Player morphology is shown in Figure 2.

Currently, this process is performed manually and uses Microsoft Visio to store the representation. While methods do exist for automatically reverse engineering the user interface structure [22], we have not yet integrated them into our work.

Figure 2 – Part of the menu bar in the Windows 95/98 CD Player Morphology.



IDENTIFYING THE CONCEPTS

Using the morphological map as an information source, we first identify the concepts indicated by the labels attached to those elements, looking for noun phrases and the indirect objects implied by verbs, a process borrowed from object-oriented analysis methods [2, 27]. For example, a "File Menu" implies that there is a concept of "File". A "Font" dialog box informs the concept "Font Size". In cases where a noun does not exist in the label, concept identification requires interaction with the system. For example, "Play" on a CD Player plays a "Track" on a "Disc". Once we identify a concept, we determine whether it is an entity type, attribute, or instance.

- An *entity* is a thing that can be distinctly identified [8]. A set of entities that share a set of attributes is an *entity type* [11]. Example: In Figure 3, Disc and Track are entity types.
- An *attribute* is an intrinsic property of a thing in the real world [29]. Basically, it is a concept that lacks independent existence except as a property of an entity type. Example: In Figure 3, Track Name and Track Number are attributes of Track.
- An *instance* is a concrete manifestation of an entity type [3].

We model attributes as nodes in our network rather than collapse them into the entity types as one would do in an object model. This is similar to the methods used in NIAM (Natural language Information Analysis Method) [28] and ORM (Object Role Modeling) [13, 24]. For example, a "Disc" in the CD Player has an "Artist" and a "Title" as seen in Figure 3. In our observations of Microsoft Word's evolution, we noticed several times that concepts that might have been modeled as attributes in one version would become full fledged entity types in the next. Modeling attributes in this manner allows us to make better comparisons of growth and complexity across application versions and examples.

Idris Hsi, "Measuring the Conceptual Fitness of a Computing Application in a Computing Ecosystem", *Proceedings of the ACM Workshop on Interdisciplinary Software Engineering Research (WISER'04) at the 12 Annual Foundations of Software Engineering (FSE-12)*, November 5, 2004, pp. 27-36.

IDENTIFYING RELATIONSHIPS

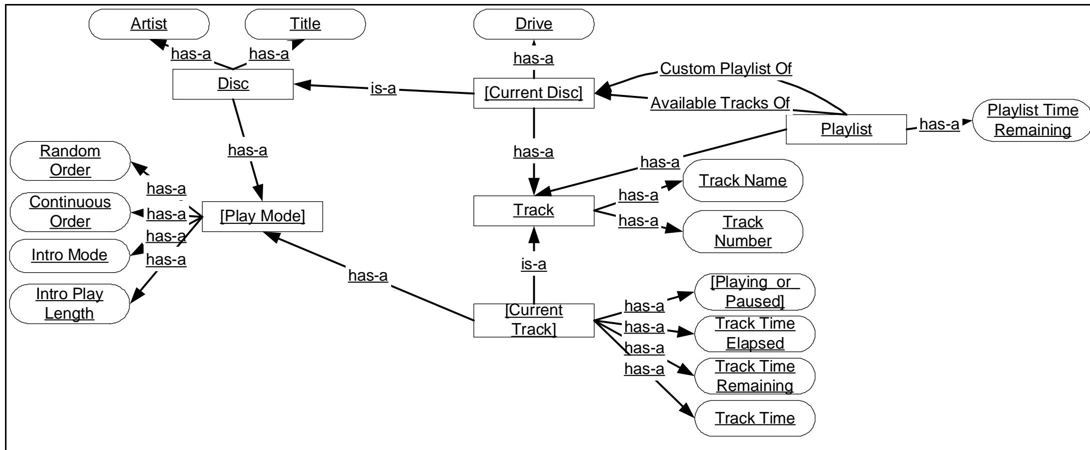
After identifying the concepts from the morphology, we identify the relationships between them by interacting with the system and by reconstructing them from observations of both static information and dynamic behavior. For constructing a semantic network, we use the basic relationships from object modeling: associations, generalization (*is-a*), and aggregation (*has-a*) [3]. We do not model constraints such as cardinality or dependencies. An example of the CD Player ontology can be found in Figure 3.

ONTOLOGICAL ANALYSIS AND RESULTS

To analyze the ontologies, we identified techniques from graph theory, specifically those used in social network analysis [30] to identify the *core concepts* and *peripheral concepts*. A core concept is essential to the application's ontology while a peripheral one is not. We identify these using *node betweenness centrality* [30] which measures a node's structural importance by the number of times it appears on a shortest path between all pairs of nodes.

We wrote a Visual Basic macro for Visio that refines a graph into an adjacency matrix that could be read by an application called UCINET, a tool for social network analysis [4]. The social and behavioral science communities model relationships between social entity types as social networks. Social network analysis methods apply algorithms from graph theory to identify both patterns and variables in the structural relationships of these networks [30].

Figure 3 – The Windows 95/98 CD Player Ontology



CORE CONCEPTS FROM THE CASE STUDIES

We identified the following core concepts from the applications along with a number of peripheral concepts that are not listed here to save space.

Table 1 – Core Concepts found in the case studies. Concepts are listed in order of their betweenness centrality values

Application	Candidate Core Concepts
CD Player	[Current Track], [Play Mode], Track, Disc, [Current Disc], Playlist
Palm Pilot Scheduler	Event, Date, To Do Item, Hot Synch, Day, Month, Time, Alarm, Repetition, Note, Every
Notepad	Page Setup, Font [Setting], Paper, Text, Paper Size, Font, Script, Header, Footer, [Configuration], [Header/Footer Code], Margins, Alignment, Font Style
Protocol Calculator / Calendar	[Time Zone], Time, Home Time

The Protocol Calculator / Calendar was found to have multiple components (isolated subgraphs) in its ontology. We performed a separate analysis on each subgraph and obtained the following core concepts per subgraph:

Table 2 – Core concepts found in Protocol Calculator / Calendar. Note: Subgraph 4 only has 2 nodes.

Subgraph	Core Concepts
1	Date, Month, Year, Calendar
2	[Time Zone], Time, Home Time, [Time Display Mode], Alarm Time, Alarm
3	[Mathematical Operation]
4	Currency Exchange [Calculator], Exchange Rate *

Through this subgraph analysis, we identified four independent subgroups in the ontology and only three core concepts. Within those subgroups, our analysis revealed core concepts that define each of them respectively.

USE CASES AND USE NICHES

Use cases come from the Unified Software Process where they are used to express requirements and guide developers in the design, construction, and testing of the system [16].

> *"A use case specifies a sequence of actions, including variants, that the system can perform and that yields an observable result of value to a particular actor." [16]*

Idris Hsi, "Measuring the Conceptual Fitness of a Computing Application in a Computing Ecosystem", *Proceedings of the ACM Workshop on Interdisciplinary Software Engineering Research (WISER'04) at the 12 Annual Foundations of Software Engineering (FSE-12)*, November 5, 2004, pp. 27-36.

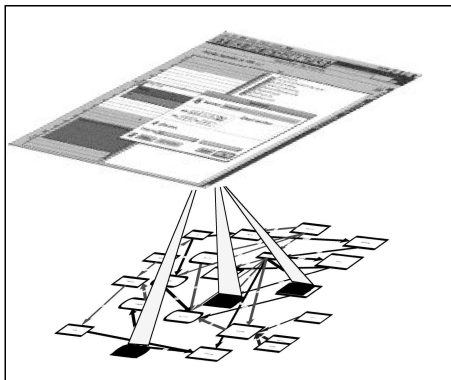
In software development, specifically in the requirements phase, the developer will gather narratives from users and structure them into these use cases.

A set of use cases can be selected to describe a set of procedures required to achieve a specific goal, a set of related goals, or to represent the all activities that will take place in a use context. Thus, use cases are ideally suited to represent both the purposes contained in a use niche and the activation cost necessary to realize the fulfillment of those purposes. To examine the relationship between conceptual fitness and use niches, we apply a technique called *use case silhouetting* that takes a set of use cases and measures the amount of *ontological coverage* by those use cases [14].

THE USE CASE SILHOUETTE

In engaging the services of a computing application through its morphology, users use or invoke concepts in its ontology. Ontological coverage measures the percentage of the ontology covered by those concepts for the desired unit of analysis. For example, one could measure the amount of ontological coverage for a given user's actions, a scenario, a goal, a specific task, an organization, and so on. We can also measure the importance of these individual concepts by examining the frequency by which they are activated. If we find that the unit of analysis has a high ontological coverage, we could infer that the application has a high usefulness insofar as its conceptual correspondence is concerned. We refer to the process of collecting data on concept frequency as *silhouetting*.

Figure 4 – The Silhouette Metaphor



The application's morphology (e.g. the user interface) provides affordances that permit access to the services. Viewed another way, these morphological elements provide portals in the 'skin' of the application through which the underlying conceptual model can be seen. Activating particular elements in the morphology casts a 'silhouette' on the concepts below where only specific concepts are highlighted as we show in Figure 4. Using *use case silhouettes*, we can measure the ontological coverage for a proposed system for a set of use cases that reflect a specific scenario, user, or set of requirements.

A use case silhouette is developed by recording the number of times a concept is referenced in a particular use case. This can be done in a number of ways. For high level use cases, where the interface is not mentioned, we can simply examine the concepts activated at each step of the use case. For example, a use case action that says "The customer requests a transaction slip from the system." tells us that the 'customer', 'transaction', and 'transaction slip' concepts have been activated by the as-yet nonexistent user interface. For low level use cases that explicitly describe how the user interface is activated, we simply account for each morphological element mentioned in the sequence of actions and trace the concepts that they invoke. For example, a use case action that says "To change the size of a character, on the Formatting toolbar, click a point size in the Font Size box." invokes the concepts 'character', 'font', and 'point size' through the morphological elements Formatting Toolbar and Font Size Drop Down Box.

Here is sample text from one of the use cases of the Windows 95/98 CD Player (CDs: storing track titles):

To store the track titles of your CD

1. Make sure your CD is in the drive.
2. On the **Disc** menu, click **Edit Play List**.
3. In **Artist,** type the artist's name.
4. In **Title**, type the title of your CD.
5. In **Available Tracks**, click the track whose name you want to store.

From this use case, we identified the morphological elements Disc Menu, Edit Play List Menu Item, Disc Settings Dialog Box, Artist Text Field, Title Text Field, and Available Tracks List. From these elements we identified the concepts Disc, Artist, Title, Track (2 times), Track Number, and Playlist (3 times).

We can learn the following from use case silhouettes:

- *The total amount of ontological coverage provided by a set of use cases.* – Assuming that the use case set provides a complete set of usages by a user or a specific use context, what is the percent of ontological coverage reached? If the coverage is low, then the application may not have high conceptual fitness for this set of use cases.
- *The parts of an ontology that are covered by those use cases and to what degree.* – A set of use cases may emphasize certain parts of an ontology over others. Even though all concepts may eventually be engaged, some concepts may see more silhouetting than others. These may correspond to the core concepts of the application or indicate concepts that are important only to that set of use cases.
- *The amount of ontological coverage by a particular use case.* – An individual use case may have low or high engagement with the application's ontology, measured by the number of concepts, especially core concepts, that

it activates. A frequently used use case with high engagement with an application must be considered carefully during design because the concepts that it uses could affect the overall ontology even if those concepts lack importance by the structural measures of centrality.

- *The importance of a particular concept relative to a set of use cases.* – A concept frequently invoked by the use cases may or may not have structural importance in the ontology. In either case, its design will impact the performance of those use cases.

Each use case describes a goal that the user wants to achieve and the sequence of actions performed on the morphological elements of an application required to achieve this goal. Because ontological excavation links each morphological element to a set of concepts, we can count the concepts activated across all the use cases to collect statistical data of activation frequency. This data allows us to measure both general and specific ontological coverage. General ontological coverage looks at how many concepts in the ontology were activated by a set of use cases. Specific ontological coverage examines how often each concept was activated by a use case to determine a concept's relative importance in the ontology for that given set of use cases.

USE CASE SILHOUETTE STUDIES

For each application, we identified use cases from the help or instruction sets. Because these were low level use cases, they described each of the morphological elements that would be triggered during the use case. We used the relationships between the morphological elements and the concepts to identify which the concepts being silhouetted by the operations in the use case. For each application, we present tables showing the ontological coverage of the set of use cases, some of the use cases and their metrics, and a list of the most frequently accessed concepts and their occurrence percentage relative to the total number of concepts invoked (including duplicates).

THE WINDOWS 95/98 CD PLAYER

The Windows CD Player allows the user to play CDs, to manage information about that CD, which has to be entered manually by the user, and to manage custom playlists.

Table 3 – CD Player Use Case Silhouette Statistics

Source	Help file associated with application
# of use cases:	23
# concepts invoked:	16
Total # concepts	20
Ontological coverage:	80%

Table 4 – CD Player Sample Use Cases and their Ontological Coverage

Use Case Name	**# Unique Concepts**	**% Coverage**
Adding Tracks to Play Lists	5	24 %
Deleting Tracks from Play Lists	3	14%
Moving Between Tracks	2	10 %
Options	6	29 %
Stopping a CD	1	5 %

Table 5 – CD Player Frequency of Concept appearance in use case set. Core concepts are italicized.

Name	**# Times Accessed in Use Case Set**	**% of Total # of concepts invoked**
Playlist	18	26 %
[Current Track]	10	14 %
[Current Disc]	8	11 %
Track	8	11 %
[Play Mode]	7	6 %
Artist	4	6 %
Title	3	4 %

By its ontological coverage, we can claim that the CD Player displays relatively high conceptual fitness to its use cases. One possible discrepancy with regard to potential actual use of the application is the prominence of the Playlist concept relative to the concepts of Current Track and Current Disc. Because the use cases were derived from help files, more complicated features, like managing playlists, required more steps to describe, producing a larger silhouette on the ontology. The concepts not covered in the use cases concerned different play modes of the CD player, such as Continuous Order.

MICROSOFT NOTEPAD

MS Notepad is a text editor that comes with the Windows operating systems. Notepad accepts a variety of types of text files in different encodings and displays and prints a document using application settings that are applied to every text file read by Notepad. These display and print settings do not get saved with the program. It also support a Log which is a code entered on the first line of a text file so that the current day and time get printed with the document.

Table 6 – MS Notepad Use Case Silhouette Statistics

Source	Help files associated with application
# of use cases:	32
# concepts invoked:	66
Total # concepts	82
Ontological coverage:	80%

Table 7 – Notepad Sample Use Cases and their Ontological Coverage

Use Case Name	**# Unique Concepts**	**% Coverage**
Adding a Log	7	9 %
Change Page Setup	11	13 %
Changing Fonts	10	12 %
Creating Headers and Footers	25	30 %
Editing Text	2	2 %
Print Document	2	2 %

Table 8 – Notepad Frequency of Concept appearance in use case set. Core concepts are italicized.

Name	# Times Accessed in Use Case Set	% of Total # of concepts invoked
Document	16	10%
Text	15	9%
Current File	13	8%
Page Setup	12	7%
[Configuration]	4	2%
Case	4	2%
Orientation	4	2%
[Header/Footer Code]	4	2%

The use cases for Notepad also showed Notepad to have a high conceptual fitness based on its ontological coverage. The concepts not accessed by Notepad included types of paper, the Character concept, and the File concept, presumably because they were too low of a level to articulate in a use case. A surprising result of the Notepad Use Case silhouette was the prominence of printing features, such as setting Header and Footer parameters, and display features, such as changing the font. Both of these features seem extremely important in the silhouette but, like Playlists in the CD player, are known to be subordinate to the main functions of Notepad which concern text editing.

PROTOCOL CALCULATOR / CALENDAR

The Protocol Calendar / Calculator is a device that implements an alarm clock, calendar, calculator, currency exchange calculator, and countdown timer. The clock also allows its users to view times in sixteen different time zones.

Table 9 – Calendar / Calculator Use Case Silhouette Statistics

Source	Instructions included with device
# of use cases:	11
# concepts invoked:	48
Total # concepts	48
Ontological coverage:	100%

Table 10 – Calculator / Calendar Sample Use Cases and their Ontological Coverage

Use Case Name	# Unique Concepts	% Coverage
Setting the Calendar	5	10 %
Set Count-Down Timer	4	8 %
Set Alarm	5	10 %
Calculator	11	23 %
Set Keytone On / Off	1	2 %

Table 11 – Calculator / Calendar Frequency of Concept appearance in use case set. Core concepts are italicized.

Name	# Times Accessed	% of Total # of concepts invoked
[Time Zone]	16	13%
Count Down Timer	9	8%
Hour	7	6%
Minute	7	6%
Second	7	6%
[Mathematical Operation]	6	5%

The Calendar / Calculator, by ontological coverage measures showed the highest conceptual fitness to its use cases. From the list of concepts we can see that the device's primary function seems to be timekeeping. Even so, the calculator use case has a large silhouette on the ontology but mainly because it encapsulates all the basic mathematical operations that one would expect to find on a basic calculator.

DISCUSSION

We have shown how use case silhouetting can provide a reasonable first approximation of an application's conceptual fitness by measuring the ontological coverage of those use cases. This analysis is clearly sensitive to use case selection and, to a lesser extent, the fidelity of the ontology excavated from the application. In the examples above, the use cases were obtained from the help files of the applications, the ontological coverage was likely to be very complete as they have to assume that all the functions will be used. This critique does not invalidate the potential benefits of this analysis. A complete analysis would simply determine use case frequency to modify the results – the number of times that a user would invoke the use case within a specified time frame. For the CD Player, one would expect the basic use cases such as Play Track to be used more often than Playlist management use cases.

ENGINEERING FITNESS INTO APPLICATIONS

In the introduction, we proposed the problem of evolving computing applications so that their fitness improves over time without a corresponding increase in complexity. We now argue, using a thought experiment, how the computing ecosystem framework and its subordinate concepts – use niches, use potentials, and activation cost – can characterize both system fitness and how that fitness can decrease or increase over time.

A thought experiment for understanding use niches is to imagine a suite of desktop productivity tools (e.g. Microsoft Office), not as a collection of programs, but as a collection of functions unbundled from their arbitrary boundaries. Instead of a word processor, a database tool, a spreadsheet application, and so on, there are simply a collection of functions for processing text, managing graphics, saving files, copying objects, sorting data, and so on. Now imagine some environment that uses these tools, like an accounting office or academic department. Over a year, we could collect data on the use of these functions and eventually we would have a characteristic profile that

could represent the fitness of all of the productivity suite's features relative to that computing ecosystem.

With a detailed analysis of the computing ecosystem, one might discover unused use potentials and untapped niches. For example, prior to Microsoft Powerpoint, people would develop presentations by handwriting them on slides or printing slides using some word processor. This showed that a use niche existed for applying computing to the problem of presentation development with a high use potential but few existing features that could take advantage of this. When Powerpoint was released, it supplied features that displayed a high fitness for the use niche of presentation creation that the word processor's features could no longer occupy that niche.

Thus, with a hypothetical profile of an idealized application with high conceptual fitness coupled with knowledge of potential features that will be required in the future, one could imagine reengineering applications to contain only those features that have some fitness in the ecosystem to reduce perceptions of bloat and to reduce unnecessary complexity in the system. We can imagine that if someone wanted their product to remain competitive against similar applications, they would want to engineer the architecture to support features that may be required in the future or to reduce the activation cost of frequently used features to improve that application's fitness.

While we have not shown in this work that such an endeavor actually guarantees a system with a high conceptual fitness or that this fitness translates to usefulness for the users, the biological metaphor does afford some understanding of how the usefulness of a computing application can be characterized in terms of its conceptual fitness to a use context.

CONCLUSION

We have described the problem of engineering useful computing applications in an evolving environment. To address this problem, we have presented a number of ideas in this paper:

- Usefulness is a desirable property of computing applications and is the extent to which an application succeeds in assisting a set of users to achieve a set of goals, relative to the amount of effort required to engage those features.
- Computing applications are designed to solve problems for specific problem domains. Thus, the usefulness of a computing application is a function of its conceptual fitness to a domain.
- A use context consists of the external physical (or virtual) environment that contains the computing application and its users, the goals that the combined computing application/user system wishes to achieve, and the various nuances that govern the operation and performance of both environment and goal completion.
- A computing ecosystem is a set of use contexts that use computing to fulfill goals, contained within an environment of interest.
- A use niche is a bounded space in a computing ecosystem that contains a subset of the ecosystem's resources, expressed as use potentials, and goals.
- A use potential is the amount of available energy or effort for activating the services of a feature.
- Activation cost is the amount of effort required to activate a feature
- Fitness within a use niche requires that the activation cost of a feature be less than the use potential.
- Use cases offer a method for characterizing the requirements of a use niche or a computing ecosystem.
- Use case silhouetting and ontological coverage can be used to measure the conceptual fitness of an application relative to a use niche or computing ecosystem.

Our future work will include the development of methods for characterizing computing ecosystems and their use niches from a combination of requirements engineering and ethnographic techniques, studies to correlate application fitness to actual use, and the development of a ontology-driven, component architecture that will allow ontological grafting and pruning to improve application fitness to an evolving computing ecosystem.

ACKNOWLEDGMENTS

We thank Colin Potts for his invaluable guidance and support on the formulation of this work.

REFERENCES

1. Arango, G. and Prieto-Díaz, R. Domain Analysis Concepts and Research directions. in Priéto-Diaz, R. and Arango, G. eds. *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, Los Alamitos, CA, 1991, 9-26.
2. Booch, G. *Object-Oriented Analysis and Design*. The Benjamin/Cummings Publishing Company, Inc., Reading, MA, 1994.
3. Booch, G., Rumbaugh, J. and Jacobson, I. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999.
4. Borgatti, S.P., Everett, M.G. and Freeman, L.C. *UCINET 5.0 Version 1.00*. Analytic Technologies, 1999.
5. Brachman, R.J., McGuinness, D.L., Patel-Schneider, P.F. and Resnick, L.A. Living with CLASSIC: When and How to Use a KL-ONE-Like Language. in Sowa, J. ed. *Principles of Semantic Networks*, Morgan Kaufmann Publishers, 1990.

6. Brooks, F. *The Mythical Man-Month*. Addison-Wesley, Reading, MA, 1995.
7. Card, S.K., Moran, T.P. and Newell, A. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.
8. Chen, P.P. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, *1* (1). 9-36.
9. Dawkins, R. *The Blind Watchmaker*. W.W. Norton and Company, New York, 1987.
10. Dawkins, R. *The Selfish Gene*. Oxford University Press, New York, 1989.
11. Elmasri, R. and Navathe, S.B. *Fundamentals of Database Systems*. Addison-Wesley, New York, NY, 1994.
12. Falbo, R.d.A., Guizzardi, G. and Duarte, K.C., An Ontological Approach to Domain Engineering. in *International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, (Ischia, Italy, 2002), ACM Press, 351-358.
13. Halpin, T. *Conceptual Schema and Relational Database Design*. Prentice Hall, Sydney, AUS, 1995.
14. Hsi, I. *Analyzing the Conceptual Coherence of Computing Applications Through Ontological Excavation*, Thesis Proposal, College of Computing, Georgia Institute of Technology, Atlanta, 2004.
15. Hsi, I., Potts, C. and Moore, M., Ontological Excavation: Unearthing the core concepts of the application. in *Working Conference on Reverse Engineering*, (Victoria, Canada, 2003), IEEE Computer Society, 354-352.
16. Jacobson, I., Booch, G. and Rumbaugh, J. *The Unified Software Development Process*. Addison-Wesley, Reading, MA, 1999.
17. Kohl, H. *From Archetype to Zetigeist*. Little, Brown and Company, Boston, MA, 1992.
18. Laurel, B. (ed.), *The Art of Human-Computer Interface Design*. Addison-Wesley, Reading, MA, 1990.
19. Lehman, M., Laws of software evolution revisited. in *5th European Workshop on Software Process Technology*, (Nancy, France, 1996), 108-124.
20. Lehman, M. and Belady, L. *Program Evolution: Processes of Software Change*. Academic Press,inc., Orlando, 1985.
21. Mackenzie, A., Ball, A.S. and Virdee, S.R. *Instant Notes in Ecology*. BIOS Scientific Publishers Ltd, Oxford, UK, 1998.
22. Memon, A., Banerjee, I. and Nagarajan, A., GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. in *Tenth Working Conference on Reverse Engineering*, (Victoria, BC Canada, 2003), IEEE Computer Society, 260-269.
23. Nielsen, J. *Usability Engineering*. Academic Press, Cambridge, MA, 1993.
24. Nijssen, G.M. and Halpin, T.A. *Conceptual Schema and Relational Database Design*. Prentice Hall, New York, 1989.
25. Norman, D. *The Invisible Computer*. MIT Press, Cambridge, MA, 1998.
26. Norman, D.A. *The Design of Everyday Things*. Doubleday, New York, NY, 1988.
27. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. *Object-oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
28. Verheijen, G.M.A. and Van Bekkum, J. NIAM: An Information Analysis Method. in Olle, T.W., Sol, H.G. and Verrijn-Stuart, A.A. eds. *Information Systems Design Methodologies*, North-Holland Publishing Company, Amsterdam, 1982.
29. Wand, Y., Storey, V.C. and Weber, R. An Ontological Analysis of the Relationship Construct in Conceptual Modeling. *ACM Transactions on Database Systems*, *24* (4). 494-528.
30. Wasserman, S. and Faust, K. *Social Network Analysis*. Cambridge University Press, Cambridge, 1994.