Supporting Metaphorical Mappings to Facilitate End-User Programming

Simone D J Barbosa **Clarisse Sieckenius de Souza**
Departamento de Informática
Pontifícia Universidade Católica
R. Marquês de São Vicente, 225
Gávea – Rio de Janeiro – RJ 22453-900
+55 21 529-9462
[sim,clarisse]@inf.puc-rio.br

PUC-RioInf.MCC30/97 September, 1997

ABSTRACT
End-User Programming still lacks easy-to-use mechanisms that support programming from end-users' point-of-view. We follow the premise that metaphors are in the grounding of our conceptual system, and recognize that the extensions end-users want to make are usually modifications upon an application's existing functionality. We propose an environment that makes use of an application's ontology to recognize and present possible ways to generate new code. We describe mechanisms that suggest modifications upon existing code based on metaphorical mappings between the ontology elements. We believe our approach helps end-users learn how the application actually works and encourages them to make software extensions and therefore get their tasks done more efficiently.

Keywords
Human–Computer Interaction, End-User Programming Environments, Semiotic Engineering

RESUMO
Para usuários finais programarem, eles necessitam de mecanismos que suportem extensão de software. Este artigo parte da premissa de que metáforas exercem papel importante no nosso sistema conceitual. Reconhecemos que as extensões que os usuários desejam fazer geralmente são modificações na funcionalidade existente de uma aplicação, e propomos um ambiente que faz uso da ontologia da aplicação para reconhecer e apresentar aos usuários possíveis maneiras de criar novo código. Descrevemos mecanismos que sugerem modificações no código das funções existentes, com base em mapeamentos entre elementos dessa ontologia. Acreditamos que nossa abordagem ajuda usuários finais a entender melhor como uma aplicação funciona, e os incentiva a fazer extensões no software e conseqüentemente realizar suas tarefas de forma mais eficiente.

Palavras-Chave
Interação Homem–Computador, Ambientes de Programação feita por Usuários Finais, Engenharia Semiótica

Supporting Metaphorical Mappings to Facilitate End-User Programming

Simone D J Barbosa **Clarisse Sieckenius de Souza** **Carlos José Pereira de Lucena**

Departamento de Informática
Pontifícia Universidade Católica
R. Marquês de São Vicente, 225
Gávea – Rio de Janeiro – RJ 22453-900
+55 21 529-9462
[sim,clarisse,lucena]@inf.puc-rio.br

ABSTRACT

End-User Programming still lacks easy-to-use mechanisms that support programming from end-users' point-of-view. We follow the premise that metaphors are in the grounding of our conceptual system, and recognize that the extensions end-users want to make are usually modifications upon an application's existing functionality. We propose an environment that makes use of an application's ontology to recognize and present possible ways to generate new code. We describe mechanisms that suggest modifications upon existing code based on metaphorical mappings between the ontology elements. We believe our approach helps end-users learn how the application actually works and encourages them to make software extensions and therefore get their tasks done more efficiently.

Keywords

Human-Computer Interaction, End-User Programming Environments

INTRODUCTION

This paper discusses mechanisms that support some interesting cases of metaphors to make it easy for end-users to extend their applications in order to get their tasks done more efficiently. It has been argued elsewhere that users may want to program and that they are able to learn how [2, 14, 15]. Unfortunately, most current applications do not offer easy-to-use mechanisms to support end-user programming.

We follow Lakoff & Johnson's premise that metaphors are in the grounding of our conceptual systems, i.e., that we think metaphorically [12]. We draw on their results in order to design mechanisms that support some metaphorical mappings within an application's domain ontology.

Software industry is challenged by the fact that users have specific problems and tasks that cannot be anticipated by the software designers. The industry has been trying to cope with it by increasing software functionality, striving to satisfy the majority of users' interests. Unfortunately, this solution brings about two major drawbacks: cluttering in the application's interface with increasing *cognitive loads* on users; and still the same inability to implement a priori solutions to every specific user's problem.

When would users want to program? Whenever they become acquainted with an application well enough to recognize that their problems cannot be directly solved, and thoughts such as "if only it did this much more..." or "if only this were done a little differently..." come to mind.

Users find themselves doing many repetitive tasks that could and should be automated [2]. End-user programming has been trying to solve this problem by giving users the power to configure applications' parameters, record and edit macros, program by demonstrating what should be done, or write code using a script or programming language [2, 14, 15].

We adopt Nardi's hypothesis that people can learn how to program, since they are used to dealing with formal languages in daily tasks and hobbies [15]. However, a number of reasons make it still hard for end-users to extend their applications. First, the languages they must learn in order to make only slight modifications are usually low-level, too general, and unrelated to the task at hand. Second, applications' extension environments do not offer much support to end-users apart from statement syntax and "canned" descriptions. Third, some extension languages try to solve the problem by collecting many abstractions provided by other programming languages, which makes them even harder to learn and use. As Ghezzi & Jazayeri once pointed out: "...the proliferation of built-in abstractions, as opposed to mechanisms for defining new abstractions, has the principal effect of making the language large without exhausting all the needs that may arise in different applications" [7].

In this paper, we have tried to extract from existing research on cognitive theories of metaphor [10,11,12] lessons to be applied in interface design of end-user extension environments, so that end-user programming is made somewhat less difficult.

END-USER EXTENSIONS

The programming activity involves dealing not only with a programming language, but also with an interface to a development environment, and with documentation about both programming language and environment interface. Users must then deal with three different representation languages: interface, programming (or extension), and documentation languages. Our concern is to carefully design such languages in order to facilitate end-users' understanding of the application and the steps involved in extending it.

In the case of extending an existing application, the development environment is usually embedded in the application. Nevertheless, the application interface and the extension environment are usually independent and very different. Users must traverse a huge gap when switching between *using* and *extending* the application. Moreover, the extension environment often does not take into account the users' lack of programming experience. What is usually provided in extensible commercial applications is a text editor with nothing written on it where users can write code from scratch, and a few tracing and debugging utilities. The result is that end-users do not even know how and where to begin.

We would rather talk about extension instead of programming. The term 'extension' presupposes the existence of something to be extended, while programming may connote that users must start from scratch. Our proposal arises from the fact that users do not usually want to create anything out of nothing, but from one or more related features instead. In other words, the application elements are the substance upon which end-users rely to build their extensions.

This is where end-user programming meets metaphors, since "The essence of metaphor is understanding and experiencing one kind of thing in terms of another"[12].

METAPHORS

It is well known that we use metaphors in our everyday learning experiences, and much of our acquired knowledge consists of concepts which have no meaning except in metaphorical relation to other concepts. In this section we describe some kinds of metaphors we make use of in our environment.

Ontological Metaphors

We have an *ontological metaphor* when we understand one experience in terms of familiar entities (or objects) and substances, and thus are able to group, categorize, and quantify them. For that purpose, we must keep in mind the grounding of our conceptual system, i.e., we tend to conceptualize the *nonphysical* in terms of the *physical*. "We use ontological metaphors to comprehend events, actions, activities, and states. Events and actions are conceptualized metaphorically as objects, activities as substances, states as containers." [12]

Orientational Metaphors

Another way to organize concepts in a coherent structured whole is by means of *orientational metaphors*, where most of them have to do with spatial orientation: up-down, in-out, front-back, on-off, deep-shallow, central-peripheral. "These spatial orientations arise from the fact that we have bodies of the sort we have and that they function as they do in our physical environment. (…) Such metaphorical orientations are not arbitrary. They have a basis in our physical and cultural experience." [12]

Metonymy

It is important to make clear the distinction between metaphor and metonymy. While the main function of a metaphor is understanding, metonymy's main function is referential, allowing us "to use one entity to *stand for* another" [12]. Nevertheless, metonymies also helps provide understanding. Examples of metonymy are "the part for the whole", "the place for the event", "producer for product", and so on.

COMPUTING METAPHORS

Another discipline that may contribute to the understanding and generation of metaphors is Semiotics. Eco [5] uses literary examples to show, in a structural approach, that a metaphor can be traced back to a subjacent chain of metonymic connections. These connections constitute the framework of the code upon which is based the constitution of any semantic field, whether partial or (in theory) global. He concludes that "the mechanism of metaphor, reduced to that of metonymy, relies on the existence (or the hypothesis of existence) of partial semantic fields that permit two types of metonymic relation: (i) the *codified* metonymic relation inferable from the very structure of the semantic field; (ii) the *codifying* metonymic relation, born when the structure of a semantic field is culturally experienced as deficient and reorganizes itself in order to produce another structure." Therefore, it may be possible, for limited parts of the Global Semantic System, to construct an automaton capable of generating and understanding metaphors. Eco states that the connections among semantic fields within a universe are pre-established, and "metaphoric short-circuits" are in fact an "uninterrupted web of culturalized contiguity that our hypothetical automaton might be able to traverse through a sequence of binary choices" [5].

While one may think that Eco's hypothetical automaton for creating metaphors is an unrealistic far-fetched dream, some of the research in the field of Artificial Intelligence already shows interesting results toward the realization of that dream, following metonymic chains in order to create analogies in limited domains [8].

Artificial Intelligence and Analogies

Hofstadter and his colleagues at the Fluid Analogies Research Group have used microdomains in order to study human mechanisms of "fluid analogy-making" [8].

Aristotle's metaphor by analogy or by proportion can be defined as A/B=X/Y (In [4]). The goal of Hofstadter's applications is precisely to answer questions of the kind "What is the A of Y?". By "X is the A of Y", we "refer implicitly to a role that A plays relative to some unmentioned but presumably obvious entity B".

In the Tabletop application, for instance, they have a very rich list of "abstract Platonic categories". The application deals with mental pressures, or *interacting pressures*. These pressures are what makes us "tweak" an object's category membership, and thus make analogies. They claim that, "for a human, the process of perceptually scanning a (tabletop) situation involves focusing briefly on one area and then on another, and having one's attention gradually drawn in more and more to specific areas". Some of these interacting pressures are enumerated, for instance: "local or locational, having to do with category membership, physical location, grouping, relative salience". In order to defend the scalability of their solution, they emphasize that these types of pressures are "universal, or nearly so, in the sense of existing in virtually any domain in which analogies can be made".

The concepts underlying the Tabletop application are presented in [6]. French expands Hofstadter's *slippages* [9], and defines three kinds of slippages: export, transport, and import.

An export slippage is a generalization or variabilization, within a situation or framework, resulting in an abstraction usually at the basic level of categorization (basic level as defined in [11]). A transport slippage involves mapping this abstraction to another situation or framework, by means of rebinding the free variables obtained in the process of variabilization. The last slippage, the import or conceptual slippage, consists of the replacement of one or more concepts in the original schema by suitable concepts that apply to the target situation. He argues that true analogy-making involves all the kinds of slippages. His work is an attempt to provide a "unified framework in which categorization, recognition, and analogy-making can be seen as being, if not precisely the same, related in a very deep and essential way" [6]. His analogy-making is in fact analogy-recognition, in that he feeds the computer with all the categorizations and information from which the analogies will be extracted. No analogies will be truly generated.

French's research also reveals the importance of categorization, not only of objects and their attributes, but especially of possible relations among these objects, such as proximity, external position in a group of objects, groups versus simple objects, previous relations established, and types of composition of objects, such as repetition, common superordinate category, and size. These relations among objects are what makes the objects more or less "salient", and therefore more or less suitable for analogy-making.

USING METAPHORS TO FACILITATE END-USER'S SOFTWARE EXTENSIONS

In order to verify our assumptions, we have built a toy application inspired in Pattis' robot Karel [17]. We have adopted a frame-like representation of Karel's world [13]. Our frames consists of properties, which can be both attributes and operations. Some slots can contain daemons which will provide a new way to traverse the frames network, and thus make use of new metaphorical mappings.

A Microworld

Our world consists of 10 streets intersecting 10 avenues, i.e., a 10x10 grid, making up 100 corners labeled (1,1) to (10,10). There may be robots, walls, boxes, toys, and beepers in it (Figure 1). Robots assemble toys from spare parts and connectors. A robot has a bag in which it may carry any kind of transportable element, such as toys, beepers, spare parts, and screws. Every element has a location, either in a corner (walls and robots) or inside a container — transportable elements may also be inside boxes or a robot's bag. A box can contain zero or more boxes. Toys, beepers, and boxes have colors.

Our robot has a limited vocabulary, and can only understand the orders move, turn left, pick object (X), and put object (X). It can also answer a few questions such as "front is clear?", "next to an object X?", "facing orientation Y?", "how many objects are there in bag?", and their negative forms, such as "front is blocked?" and so on.

End-users must program additional actions or questions in order to get their jobs done. Our goal is to make it easy for novice programmers to teach the robot how to do tricks and become smarter.

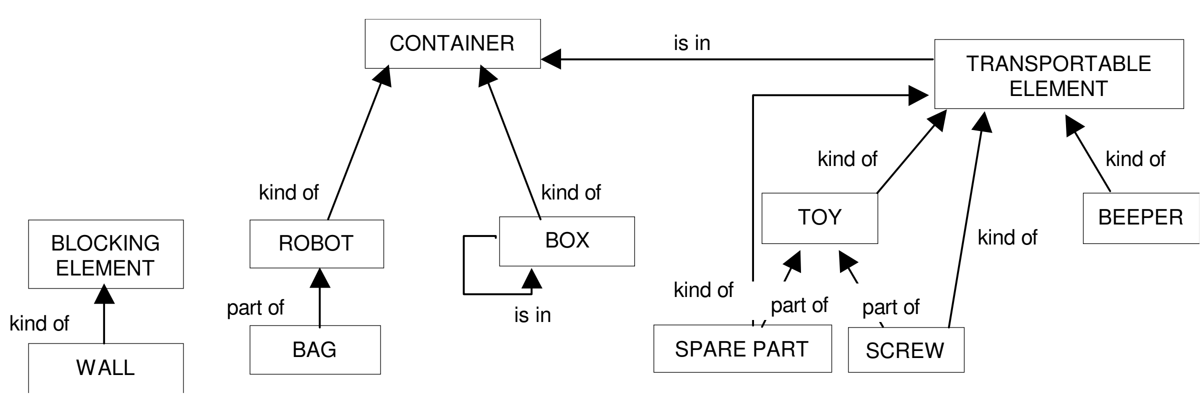

Figure 1 — Objects hierarchy.

As we were building the application's ontology, we noticed the "transportable element—is in—container" relation, and asked ourselves, where these containers could be. Moreover, where were robots, walls, and boxes? Still in the design phase, we —the application's designers— felt the need to create some element that looked like a container, but had no volume. Hence we extended the objects hierarchy by creating the CORNER frame, by removing the CONTAINER's volume property (Figure 2). We now describe how that was accomplished, because it hinges on interesting metaphorical mappings.

Mechanisms to Support Metaphorical Mappings in Creating New Elements

There are three general mechanisms for creating new frames: we can add, remove, or modify properties. The mechanism for adding one or more properties is an inheritance mechanism: a new node is created with the same properties as the parent node, and new properties may then be added. In order to remove a given property, a virtual super-node is created, transporting to it only the properties that are to be maintained in the new node. Then another node is created inheriting from the supernode. In the third mechanism, to modify a given property, a virtual supernode is created, the new node inherits from the first, and the slots corresponding to the modified properties are left for the user to fill interactively. This would seem equal to removing and then adding properties, but has two distinctive features: users may modify only the range of possible values that property may assume, or substitute another property for that one, changing its name and semantics. Therefore, when modifying a given property, users need to verify the conceptual validity of the operations that involve such property. The environment keeps track of the modified properties, and for every operation in the newly created frame that involves one of these properties, it asks users if it should substitute the property for the new one, or treat it like a property that has been removed.

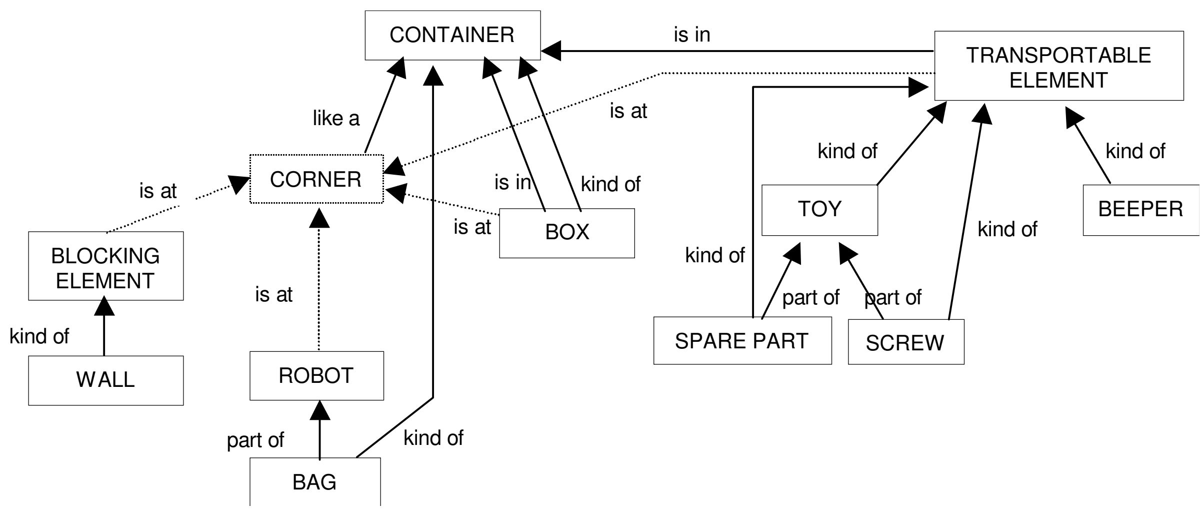

Figure 2 — Extended objects hierarchy. The extensions are represented in dashed lines.

Vanishing Properties

As stated earlier, properties may be attributes or operations. Attributes, as the name suggests, are stated qualities within a range of possible values, whereas operations are qualities whose values are calculated by fragments of code, and the calculus is made upon properties themselves.

When removing a property, we run the risk of invalidating some of the inherited operations. In that case, users must specify what happens when they encounter a property that does not exist in the newly created frame.

When creating a new frame by removing a feature, the environment will check for the validity of the operations inside that new frame. Whenever a property that has been removed appears, users are prompted to specify what to do. This would include removing conditionals where such property is involved, changing the property to a different one, or assuming a constant value.

Dangling Links

After a frame F; has been created based on a frame A, the environment asks users what links are preserved, i.e., "if A–R–F, does A–R–F' ?" If not, "is there a new relation R' , metaphorically related to R so that A–R–F' ?"

In our example, we have decided to create a new relation "is at", and specified that every frame connected to CONTAINER through the "is in" relation is now also connected to CORNER through the "is at" relation, such as TRANSPORTABLE ELEMENT and BOX. Moreover, we now link other frames to CORNER through the "is at" relation, as is the case of ROBOT and BLOCKING ELEMENT.

Users' Perspective

The result of applying one of the above mechanisms is a "like a" relation between the source frame and the newly created one. This "like a" marks a slackening of some of the object features. In this case, corner has the feature "may contain things" just like a container, but it does not have a "volume" attribute.

This is equivalent to having a super-node CONTAINER0, from which we derive the CONTAINER node which has a volume attribute, and a CORNER node, which does not have a volume attribute. Despite the equivalence, end-users may have a hard time understanding virtual classes, i.e., classes that have no instances and act only as a template for other subclasses. But, it seems very simple to add or remove attributes of a given class whenever necessary, in order to create a new one, and this is the approach we follow.

Every element in our world has certain attributes. The hierarchy of some of these attributes is shown in Figure 3:

Reusing Code Through Metaphorical Mappings

Our environment allows users to reuse existing code and modify only what is necessary to generate their extensions. The existing code can either be the code corresponding to a sequence of interactions, or to a single interaction, which can be unfolded into lower level constructs. Every users' action is recorded in a history list, and they can choose a sequence of actions from that history to use in their extension code. They would then enter the programming mode, in which the environment offers support for modifications based on metaphorical mappings within the application domain ontology.

Whenever a user wants to reuse and modify a piece of code, s/he must indicate what are the elements that are to be changed. It may involve changing an object or a property.

Changing an Object

When a user chooses a different object O' to replace an existing object O within an extension, the environment first searches the object hierarchy for the relations in which O is involved, and verifies interactively if they are maintained for O'. Then it replaces every occurrence of O for O', and verifies the corresponding attributes. If O' does not possess one of the specified attributes of O, the attributes hierarchy is then searched for related attributes in order to present the user with choices of substitute attributes and comparisons.

Changing a Property

When a user chooses to modify a given object's property, a list of the object's properties is generated. In addition, other options are presented depending on the kind of property: attribute or operation.

Attributes

When a user chooses to modify a given object's attribute, the attributes hierarchy is searched for related properties and some options are generated.

Operations

When a user chooses to modify a given object's operation, the search space depends on the type of value the operation returns: if it is an object, the objects hierarchy is searched; if it is an attribute, the attributes hierarchy is searched.

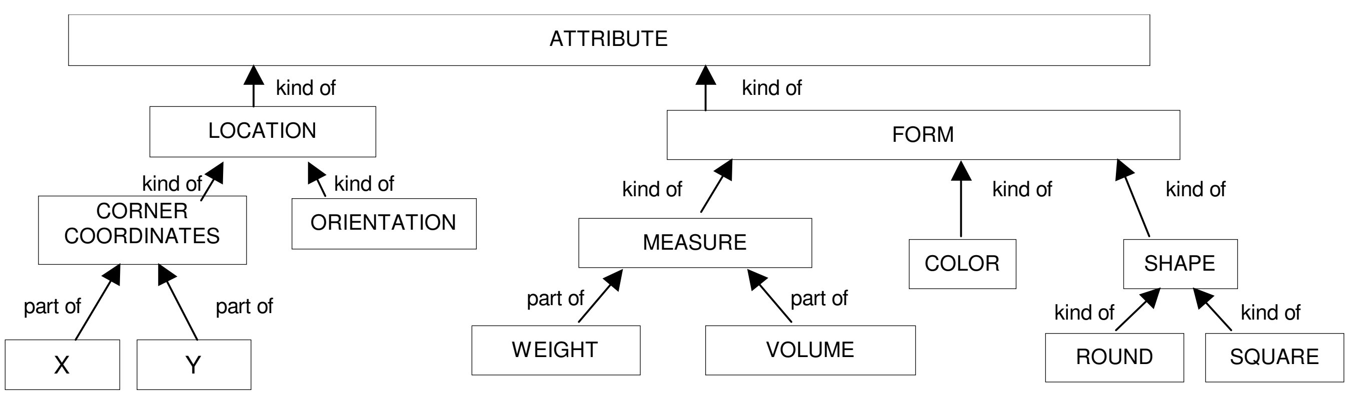

Figure 3 — Attributes hierarchy.

Relations

Part_of relations indicate a content–container relation, and this can be used for generating possible modifications to existing code, as will be seen in a few examples below.

Example

One of the primitive properties of a robot's bag, **count_beepers**, counts the number of beepers in the bag. When reusing this property, users may choose to unfold this primitive into lower level constructs, as in[1]:

attach property **count_beeepers** to **bag**.

[1] The scripting language used in this paper is the result of ongoing research, and is forthcoming by Silva, de Souza and Ierusalimschy.

```
count_beeepers is an operation with default value 0, and has
steps:
for each beeper in me,
        let count_beepers now be count_beepers + 1.
```

Users may want to modify that operation code to count only red beepers, which would yield the following code:

```
attach property count_red_beeepers to bag.
count_red_beeepers is an operation with default value 0, and
has steps:
for each beeper in me,
        if beeper.color = red then
                let count_red_beepers now be
count_red_beepers + 1.
```

It would be very easy to substitute bag with box, and now count the number of red beepers inside a box:

```
attach property count to box.
count is an operation with default value 0, and has steps:
for each beeper in me,
        if beeper.color = red then
                let count now be count + 1.
```

In the next few paragraphs we describe some mechanisms involved in helping users change elements in the code.

Searching objects and attributes hierarchies

In order to reuse a piece of code, the user should point out what are the changing elements.

If users want to count the screws inside a box, they may click on **beeper** and choose to replace it with **screw**. Our environment first checks what is the existing relationship between **beeper** (the modified element) and **box** (referred by the **me** keyword). The relation **located in** is found. Then it verifies if the same relation holds between **screw** and **box**. It does. Next, it searches every step verifying if each beeper property holds for screws. When it comes to screw.color, it verifies that screws do not have the color attribute. We could simply eliminate the conditional, and always increment our counter. Nevertheless, we continue our search.

It would be pretentious to think we could provide heuristics to come up always with the right answer. It would be like a guessing game, where sometimes the environment would guess the right answer, and sometimes not. In any case the user would not have gained much: if the guess is right, the end-user would not care for understanding the resulting code, and if it were wrong, he would not know what to do to get it right. Our approach tends to provide a few different possibilities of modification and, even if users want some other thing, they can investigate the resulting code for each presented option and then acquire a more coherent mental model of the underlying ontology and programming language. Then they would choose the closest match and make the necessary modifications to it.

Back to our example, the attribute hierarchy is then searched, in order to find any similar attributes that may be used and replaced. Since color is a *form* attribute, other *form* attributes are presented to users so they would make the appropriate choice.

In this case, they may want to:

- exclude the conditional, and always execute the let count... statement
- modify beeper.color = red to screw.measure.volume = ?
- modify beeper.color = red to screw.measure.weight = ?
- modify beeper.color = red to screw.shape = ?

But that is not all. Searching the object hierarchy, it is also verified that screws can be part of toys, and toys have a color attribute. In this case, the iteration itself can be modified, in order to search all toys in a box, and check if the toy's color is red and then iterate within each screw of that toy. The option would be presented like:

- count every screw within every red toy inside the box.

This would generate the following code:

```
attach property count to box.
count is an operation with default value 0, and has steps:
for each toy in me,
        if toy.color = red then
                for each screw in toy,
                        let count now be count + 1.
```

Content–container relations

If the user now wants to count beepers in a certain corner of the world, they may click on **box** and choose **corner** instead. The environment verifies the beeper–located in–corner relation, and then proceeds to the verification of the operation steps. This time, the **beeper.color** property still holds. However, the statement **for each beeper in corner(X,Y)** poses an additional problem: there may be boxes in a corner, and beepers inside these boxes. The environment detects this possible recursion, and presents the following choices:

- search only beepers in **corner** (X,Y) which are outside any other kind of **container**
- search inside every **container** in **corner**(X,Y) recursively

Notice these two options differ on the kind of access to the objects. In the first option, we have immediate access to the desired objects, while in the second option we have an indirect access to an object which can be inside another, and so on recursively. Instead of guessing which is the preferred choice, we present both options for the user to choose.

Unmatched attributes

The user may want to make multiple transformations to the operation definition. For instance, he may want to change **beepers** to **walls** and **box** to **world**. But walls do not have any color, or any kind of form attribute, only a location attribute, specified by giving an (X,Y) corner coordinate. This attribute is already accounted for in the iteration, since the relationship beepers–located in–box is now mapped to walls–located in–corner–part of–world, and any iteration on the corners of the world is in fact an iteration throughout every valid (X,Y) coordinate. Therefore, the environment would present the only option, such as:

- count every **wall** in the **world**

One might argue that, since we had only one option, we could present it directly within the code, as in:

```
attach property count to world.
count is an operation with default value 0, and has steps:
for each wall in me,
        let count now be count + 1.
```

Nevertheless, even if the environment only finds one option, that does not necessarily mean it is the only possible option. Therefore, we present it in the same way we present every other option, in order to ensure that users consider it exactly what it is: an option, possibly among others. Moreover, if the code was automatically presented, maybe users would not understand it, and run the risk of having an extension doing the *wrong* thing, i.e., something that was not intended by them.

A FRAMEWORK

In order to support the extension mechanisms based on metaphorical mappings described throughout this paper, we could make use of framework development approach based on viewpoints described in [1], where application objects (Abstract Design Objects or ADOs) are kept separate from their "views" (Abstract Design Views or ADVs). Their research emphasizes the usage of their approach within traditional software development process of OO systems, mostly at the design level. We draw from their results in order to build user-extensible applications.

ADOs provide the core functionality and data of the system, and ADVs are objects that perform interface roles in the software model, such as handling of user input and presentation of information to users. The relation between a view and its corresponding problem-domain object is called "views_a". ADVs are interfaces that know the identity of the ADOs they are viewing, whereas ADOs have no knowledge of the identity or existence of their views. A viewpoint is a grouping of classes and their relationships. The final system is derived from a composition of the original system with its viewpoints. The matching components in different viewpoints will also be connected by a "views_a" relationship.

In such a framework, our environment would provide multiple viewpoints, from which end-users may choose the ones which are adequate for the task at hand. Furthermore, end-users would be able to create their own viewpoints and fit them into the framework. This framework approach allows for unanticipated extensions to existing systems. By following the proposed application design, extensibility emerges naturally from the existing application. Moreover, by using viewpoints we preserve existing applications and eliminate the risk that end-users might accidentally corrupt them.

What our research adds to the framework development approach based on viewpoints is the ability to dynamically generate viewpoints based on the application's domain ontology. Therefore, while some viewpoints would be hardwired and provided directly by application designers, others would be created during application extension, according to users' choices that define the focus of the modification or extension.

CONCLUSION

We have drawn from Cognitive Science, Semiotics, Artificial Intelligence, and End-User Programming research results that may provide users with a better understanding of the applications they are dealing with, and hence the ability to extend them to more efficiently carry on their own tasks.

Given the application's ontology, our approach is to infer, through metaphorical or structural mappings, some possible modifications users may want to make upon existing code. By providing the options for users to choose, instead of inferring a single solution, we let users be in control of their code, and hope that they understand the underlying application by inspecting different and possibly unforeseen options.

We have shown how metaphorical mappings can be made upon the application ontology by *slipping* some features of certain objects or attributes in order to reach a metaphorically related object or attribute that can henceforth be used in place of the previous one.

Our approach differs from some traditional ways of reusing code that advocate the development of reusable components that are to be used integrally, and by means of inheritance mechanisms. Instead, we provide mechanisms for creating new code by modifying existing code, based on the application's ontology.

This paper presents part of ongoing research on the design of an interconnected environment and of the languages involved in end-user programming — interface, programming, and documentation languages —, in order to bridge the gap users must traverse when extending their software. A prototype environment proposed in this paper is currently being implemented for the Microsoft® Windows® 95 environment.

Besides finishing the implementation of the achieved architecture, scalability measurements are involved in future work called upon to verify the validity of our approach in real applications, such as word processors, spreadsheets, calendars, electronic mail, and so on. We must yet investigate the cost/benefit ratio of building such applications with our extension mechanisms in mind, and whether such applications will really improve users' performance in carrying out their tasks by extending the software as to fulfill their specific needs.

ACKNOWLEDGMENTS

Simone D J Barbosa would like to thank CNPq for providing her doctoral scholarship. Clarisse Sieckenius de Souza is grateful to CNPq, the agency which has given her a grant to do research about Semiotic Engineering. The authors would also like to thank Prof. Dr. José Lucas Mourão Rangel Netto and the Semiotic Engineering Research Group at PUC-Rio for their contributions to ideas presented in this paper.

REFERENCES

1. Alencar, P.; Cowan, D.D.; Fontoura, M.; Lucena, C.J.P. A Framework Development Approach Based on Viewpoints. To be published in the *ACM Book on Object-Oriented Application Frameworks* (two volumes), 1997
2. Cypher, A. *Watch What I Do: Programming by Demonstration*. The MIT Press. Cambridge MA, 1993.
3. Draper, S.W. (1986) Display managers as the basis for user machine communication. In Norman and Draper (eds.) *User Centered System Design*. Hillsdale. Lawrence Erlbaum and Associates.
4. Eco, U. *Semiotics and the Philosophy of Language*. Indiana University Press. Bloomington IN, 1986.
5. Eco, U. The Semantics of Metaphor. In Eco, U. *The Role of the Reader*. Indiana University Press. Bloomington IN, 1979. pp.67-89.
6. French, R. *The Subtlety of Sameness*. The MIT Press. Cambridge MA, 1995.
7. Ghezzi,C.; Jazayeri, M. *Programming Language Concepts, 2nd Edition*. John Wiley & Sons, Inc. 1987
8. Hofstadter, D. et al. *Fluid Concepts and Creative Analogies*. Basic Books, A Division of HarperCollins Publishers, Inc. New York NY, 1995.
9. Hofstadter, D.R. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books. New York NY, 1979.
10. Lakoff, G. The contemporary theory of metaphor, in Ortony, A. (ed.) *Metaphor and Thought, 2nd edition*. Cambridge University Press. Cambridge MA, 1993, 202-251.
11. Lakoff, G. *Women, Fire, and Dangerous Things.* The University of Chicago Press. Chicago, 1987.
12. Lakoff, G. and Johnson, M. *Metaphors We Live By.* The University of Chicago Press. Chicago, 1980.
13. Minsky, M. A Framework for Representing Knowledge. In Winston, P.H. (ed.) *The Psychology of Computer Vision*. McGraw-Hill Book Company. New York NY, 1975.
14. Myers, B.A. (ed.) (1992). *Languages for Developing User Interfaces*. Jones and Bartlett Publications. Boston.
15. Nardi, B. *A Small Matter of Programming*. The MIT Press. Cambridge MA, 1993.
16. Norman, D.A. Cognitive Engineering. In Norman,D.A. and Draper, S. (eds.) *User-Centered Systems Design*. Lawrence Erlbaum and Associates. Hillsdale, NJ. 1986.
17. Pattis, R. *Karel, the Robot*. Second Edition. John Wiley and Sons. New York NY, 1995.