Identifier Based Representation and Management of Software Components

Pedro Reis dos Santos
Technical University of Lisbon
Av. Rovisco Pais, 1096 Lisboa Codex, Portugal
E-mail: prs@digitais.ist.utl.pt

Abstract— **As the problems to be solved by computers become larger, software development becomes more complex and difficult to grasp by software engineers. We propose a representation model that uses identifiers extracted from the various software development stages and interconnects them in a hierarchical naming tree. A set of mechanisms is then used to manipulate and gather relevant information for each particular task of software processes.**

I. Introduction

Computers are being used to store, manage and solve increasingly more complex problems. This complexity is reflected on every stage of computer application development [14]. Problems begin as we need to decompose the problem and build up to the final test and acceptance stages. The main problem is that humans are limited in their ability to deal with large and complex systems [12]. Abstraction becomes the most common mechanism to enable humans to grasp and solve large and complex problems. From these abstractions we are able to model the problem, identify the requirements and produce a computer solution. Finally, the software solution is even more complex than the problem that we started with [16]. Much of the software complexity comes from the problem to solve but, up to some extent it results from the solution found. Some of the additional complexity results from the model chosen to solve the problem. Models based on simple and extendible concepts using a small set of entities and operations have proven successful [11]. They result in a series of manageable abstractions composed or decomposed in hierarchies, that humans can deal more easily. Examples include Petri net processes, entity-relationship databases, object-oriented models.

Engineering disciplines use sets of representations for modeling artifacts. In software engineering we have entity-relationship models, dataflow and state transition diagrams. However, they address a limited set of stages in the software development leading to the well known inconsistencies between the final product and the original problem to be solved. These problems become more acute as the problems to be solved and the resulting software representation scheme become more complex. We advocate the needed for a place, that we can use a reference, where we can represent the initial problem and then enrich it as we get closer to the solution.

Producing several separate documents leads invariably to inconsistencies so the model used must ensure continuity. It would be too complex to induce every aspect of the problem and every aspect of the solution into one model. It better to integrate different representation schemes, each one focusing on a particular aspect. Since different participants have different views, of the problem and solution, they need a model that helps them to cope with possible inconsistencies.

The existence of a single model follows the need to track each system functionality from definition to the final implementation. This model includes a mesh of links to relevant areas of each aspect from the initial problem representation to the final solution. The abstraction that humans use to refer functionalities within a system are names. So names are good candidates to model a wide range of software process stages. Our model is based on a rich structure of names that allows the representation of each different stage of software processes. It provides flexible cross-referencing mechanisms that enables consistency checking as the development flows. In addition, it offers a set of management and manipulation mechanisms to navigate through the names and their associated values.

We aim at achieving a full representation of the software system just by using names to identify the abstraction used when sketching the problem and adding references from those identifiers to the several stages of development process. These identifiers will have a meaning to the project member and can be used as a reference as the project evolves. A set of mechanisms will allow search operation based on those identifiers. The resulting system is a name service where different types of data can be registered and searched in a flexible and intuitive way.

II. Separation of Concerns

In order to have a clean model we impose the requirements uniformity and economy of concepts. The first requirement makes life easier by giving less to remember, no special cases. The later requirement aims to get the greatest power with the smallest number of concepts [15]. By looking closely into the way we represent information we find out that at the higher abstraction level resort to names [7]. Our objective is to concentrate on names bound to some piece of information, called identifiers, to represent a system. If we extract all useful and meaningful identifiers from objects to the model we end up with raw data, called entities.

The first problem we come up with is mobility of entities. That is when entities change place we end up with dangling links, a common problem when dealing with symbolic links in file systems and URL in WEB services. We need an intermediate level to make identifiers immune to entity changes. This reference level is present in many systems, such as file system i-nodes, object's capabilities and internet addresses. The reference level is also a naming level where names are represented in computer form, instead of a user easily readable form.

The identifier, no matter how meaningful its name can be, may not be enough to provide an immediate distinction of the entities it refers to. This does not mean that it can be used to refer two entities or that an entity may be referred by other identifiers, it simply means that the name used may not be enough to identify the entity. The solution is the use of attributes to qualify the entity making the distinction clearer. However attributes should be treated as a name decomposition mechanism, not as different kind of object in the model.

The model is composed of names that refer references and references that refer entities. So far there is nothing really new, any system with a complete naming scheme will provide such concepts. For example, filing systems use pathnames, i-nodes and files. WEB services use URN, URL and HTML files. Programming languages use variable names, variable addresses and variable values. DNS uses host and domain names, internet addresses and hardware addresses. However, in all these approaches there is a distinction between user names, machine names and values.

III. Splitting Objects

Abstraction is the most effective way to manage complexity. A hierarchy is a structured organization where different abstractions can be handled at different levels. The problem is divided into ordered levels or increasing detail [1]. The detail does not go away, you just push it into a lower level. Most important is the fact that hierarchies come natural to most people. So, when you describe a problem whether in top-down or bottom-up approach you end up with the solution at the root of the hierarchy and set of simple problems at every leaf. Functional languages claim must of their expressively by exploring this concept as opposed to languages based on side effects of sequences of independent instructions. However, each intermediate level has a role to play in the process and, in many cases, refers to important milestones. So, when describing a problem there is a need to associate data with each node and each leaf. The root node will start to contain the core of the problem and finish up with the final solution, while each leaf will start from a simple and atomic problem that is added with one or more solutions (figure 1).

The characterization of each node or leaf will start from the name and be enriched with a series of properties that represent their values. The properties will also have a name describing their function that will be inserted as sub-names of the node being characterized. The objects being modeled will be broken down into values and their names, and these names will be regrouped as a hierarchical tree and pointing to the respective values. These object nodes [3], after being characterized, must relate to other objects in the system. The links [8] are, in a second stage, a way to express weaker connections between objects in the system. They form the basis for sharing among objects and definition of objects as extension of others [13]. Links allow the designation of other identifiers while identifiers refer to entities. The way a link operates is given by the name of the link. For instance, links named *delegate*, *class* or *inherit* will operate differently on the target object. Because, there are no predefined names a dictionary must be supplied, if the behavior can not be unambiguously extracted from the name of the link. This dictionary is a set of identifier to entity associations and can also be a part of the global hierarchy.

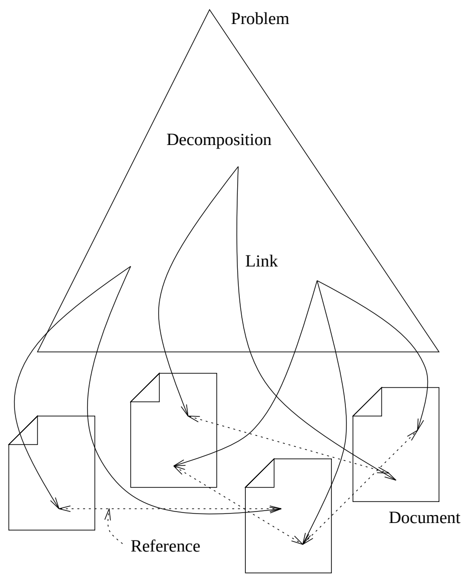

Fig. 1. A naming tree links project documents.

IV. Management Issues

Since we are representing potentially large and complex systems in a single monolithic hierarchy, we must tackle the managerial problem. We divide the system naming hierarchy down into small interconnected hierarchies. The smaller hierarchies may reflect different views over the same system or may reflect some administrative or geographical organization. The interconnection of the hierarchies into the global hierarchy can be done by special links. These links can reside on leaf of the hierarchy if there is a need to refer another hierarchy or, the other way around, the link can be inserted at the root of the hierarchy stating where this hierarchy should inserted into the global system.

As the system is described in very simple and repetitive way the mechanisms necessary will be fairly simple. The tools that implement them, however, will time consuming since they must iterate through thousands of equal structures differing only the names and values of each entry.

A. Locating and Browsing

To keep track of the relevant information in a certain stage of the development process, two operations are fundamental in helping the user to locate and relate relevant information: locating and browsing.

The browsing operation on trees is performed in two different ways: breadth and depth [17]. In this model, however, both breadth and depth algorithms can be large enough, if you add many details. To make browsing lighter, we extract a subset of tree nodes that match a certain criteria using a find operation and then browse the result.

Finding is necessary when you know what you are looking for but you forgot the location on the model. If you have a graphical representation the problem is even more serious since you will end up with huge sheets and highlighting relevant information might be difficult for the system to identify them and for the user see them in the naked eye. A better approach is to generate an auxiliary model containing only these relevant information, more or less the same way you perform select operations on tables of a relational database and end up with new table containing the results. In this case you will get a smaller naming tree containing only the nodes leading to the selected leafs. Then you can perform browsing operations on this subset tree as in the original one but without being distracted by lots of data irrelevant for this stage of development.

B. Name Spaces

The hierarchical naming tree can be broken into small hierarchies interconnected, but independent from each other, the named spaces (figure 2). These name spaces form the basis to express the necessary change in the support characteristics as we deeper into the hierarchy. Different name spaces can reside on distant geographical locations or hold different access protections. Name spaces can also perform different types of persistence or even, during a test and try phase, provide no persistence.

Since we have dissociated identifiers from entities, we need to have different operations to operate on both. If you move an identifier only its name is moved, the entity will remain where it is. On the other hand, when you move an entity, the identifier or identifiers that refer to it will remain where they where. The usual operation of moving an identifier and carrying the associated entity can also be done by a composition of the previous operations.

C. Manipulation Language

In order to perform more complex operations on the structure obtained when representing the system we need a manipulation language. This language, following the same philosophy of the representation model, is composed only of names and values. This means that there are very simple syntax rules, since any sequence of names and values are allowed. The language interpreter performs the lexical identification of the names and values and then, as it evaluates

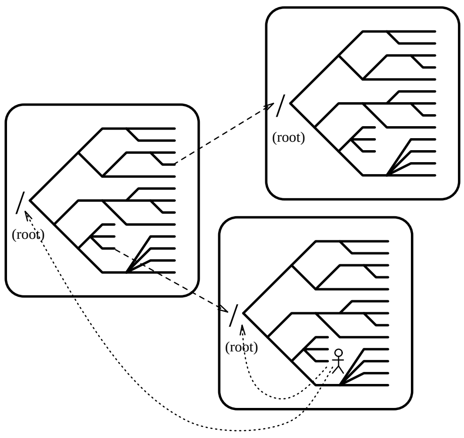

Fig. 2. Connecting project name spaces.

the code, it replaces the names by the associated values. If the values are associated with data, the later are pushed into the stack of the interpreter; if it is associated with a compiled or interpreted function, the function is called performing the operations directly on the stack of the interpreter.

The integration into the representation model is done by allowing the variable names to be entries in the hierarchical naming tree. These variables can then be accessed by relative or absolute pathnames. By given a certain behavior to some names, like *inherit* or *delegate*, simulations of the representation model can be performed.

Although the final solution to the problem being modeled can be solved using the language, it is intended as a model manipulation language. This means that end-user friendliness is not one of its strongest points [9]. Nevertheless, it can be used as a symbolic assembler for some higher level language.

D. Policy Free Environment

Many models give you a large set of constructs to built the solution for a problem. This offers you a model ready to use but, generally limited to a certain area of problem solving where those constructs apply better [2]. By giving no specific constructs, we enable each project member to establish particular constructs or restrictions to each particular area. On the other hand, if some rules must be followed there is a need to first develop a meta-model using this model and then model the problem in hands in those rules. However, if no other constructs are defined the model can still be used to develop solutions that have no restriction other than a hierarchical problem decomposition. In the later case every project member can still enjoy the cross-reference and browsing capabilities. This is specially useful when modeling problems that do not seam to match any existing model.

The purpose of this model is to force the user to a minimal set of constructs that constitute the most general form of design: hierarchical abstractions using names.

V. Existing Approaches

Filing systems support hierarchical naming trees with a limited set of links. However, the major limitation results from the fact that, while splitting different phases of the development process into different directories is common, detail are stored in files. Establishing associations between the same abstraction in different approaches and development stages is difficult. The use of hypertext is opening new perspectives for the first stages of the development process but integration with the remaining process is difficult to achieve.

Entity-relationship databases offer a more systematic approach. Tool exist to model design and to produce in automatic way a structure and basic procedures for each problem. A software engineering database should be able to express several levels of abstraction. While tables are a powerful and organized way to store data at some level they lack scalability. Object-oriented databases [10] have additional expressively but it is still difficult to begin with an incomplete specification and add detail in a simple way. Expressing entities on earlier analysis and design stages is difficult, due to typing and semantic restrictions that will only be available on later stages.

Integrated project support environments help reducing complexity of tools manipulation and organize large sets of items by structuring them. These environments provide their own tools or integrate existing tool into it. The use of databases is the most common way to support all data related to the project. PCTE [6] is the basis for many IPSEs. PCTE and its based IPSEs provide global schemas that must be followed closely and are generally monitored and kept consistent. The PCTE approach proves to be too restrictive since each implementation must follow a set of rules resulting from the scheme definition. The entity-relationship approach adapts poorly to a composition and decomposition of abstractions needed in the software development process. It is successful in representing each layer in structured way but that is not the only problem for a software engineer.

Object-oriented modeling languages do not support the later stages of the development process. The Unified Modeling Language [4], [5] is a very complete and expressive analysis and design method borrowing ideas from the most known works. Its diagrams offer very visual and intuitive descriptions of the system. However, cross-referencing between diagrams is a time consuming job. UML is a solution for problems that can be solved by automatic generation of the final code from the final design, if an analysis and design is available. Large projects tend to require direct intervention at low level languages in order to optimize pieces of code or ensure interoperability with other systems.

References

[1] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, April 1996.

[2] Sergio C. Bandinelli, Alfonso Fuggetta, and Carlo Ghezzi. Software process model evolution in the spade envrionment. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, December 1993.

[3] Daniel Bardou and Christophe Dony. Split objects: a disciplined use of delegation within objects. In *Object-Oriented Programming Systems and Applications*, 1996.

[4] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language for Object-Oriented Development.* Rational Software Coporation, 0.91 edition, September 1996.

[5] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language Sematics.* Rational Software Coporation, 1.0 edition, January 1997.

[6] Gerald Boudier, Ferdinando Gallo, Regis Minot, and Ian M. Thomas. An overview of PCTE and PCTE+. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, November 1988.

[7] David Boundy. A taxonomy of programmers. *Software Engineering Notes*, 16(4):23–30, October 1991.

[8] Link Architecture for a Global Information Infrastructure. *Jeffrey R. Van Dyke.* PhD thesis, Massachusetts Institute of Technology, June 1995.

[9] James Howatt. A project-based approach to programming language evaluation. *ACM SIGPLAN Notices*, 30(7):37–40, July 1995.

[10] Won Kim. *Modern Database Systems.* Addison-Wesley, Reading, MA, USA, 1995.

[11] Steve McConnell. Keep it simple. *IEEE Software*, 13(11), November 1996.

[12] Michael C. McFarland. The social implications of computarization: Making the technology more humane. In *26th ACM/IEEE Design Automation Conference*, pages 129–134, 1989.

[13] Hernán Astudillo R. Reorganizing split objects. In *Object-Oriented Programming Systems and Applications*, 1996.

[14] Jerzy W. Rozenblit and Sanjaya Kumar. Toward synergistic engineering of computer systems. *IEEE Computer*, 30(2):126–127, February 1997.

[15] António Rito Silva, Pedro Sousa, and José Alves Marques. Development of distributed applications with separation of concerns. In *Asia-Pacific Software Engineering Conference*, Digital Equipment Corporation 1995.

[16] Anthony I. Wasserman. Toward a discipline of software engineering. *IEEE Software*, 13(11), November 1996.

[17] M. Wein, Wm Cowan, and W. M. Gentleman. Visual support for version management. In *Symposium on Applied Computing ACM/SIGAPP*, pages 1217–1233, March 1992.