Assisted Selection of Components using Classified Identifiers

Pedro Reis dos Santos
Technical University of Lisbon
prs@digitais.ist.utl.pt

Rui Gustavo Crespo
Technical University of Lisbon
rgc@digitais.ist.utl.pt

Abstract

Developing complex software systems is a demanding task for software engineers even with the most update frameworks and tools. Since, for complex systems, automatic generation of the final product is not achievable an assisted selection of alternatives and highlighting of possible inconsistencies is needed. We propose a representation model that classifies identifiers extracted from the various software development stages. A set of tools and mechanisms assists the developer to keep track of relevant information for each stage of the software process. Possible reuse candidates can, therefor, be rejected by detecting misplaced identifiers in the representation model.

1 Introduction

Reuse of software components has been around for many years. Operating systems are one of the first examples of widespread reuse of software components. Since Fortran, languages offer input/output constructs to access storage and user in a, more or less operating system, independent form. Reuse can improve significantly the quality and productivity of software development in software projects. With the increasing size of software projects, reusable components will play a growing role in software production. Studies [11] found that the "in-house syndrome", as well as other feared factors, did not affect reuse. The programming language being used, the software engineer experience, the use of case tools or legal problems are factors that hold little influence on software reuse.

Important factors influencing reuse includes the availability of trustable and high quality assets, the existence of common software process and a reuse education. These three factors are interconnected and should be better addressed. A reuse education can be seen as driving factor, but can not on its own produce significant results if no support for reuse is given. Such support should be built around a common software process in order to guide the educated software developer. The availability of high quality assets does not depend neither on the software developer nor on the software process. However, the software process should assist the software developer in the task of identifying such high quality assets. In this perspective, the major effort should be placed on making available a common software process that allows the educated software developer to improve reuse by identifying good reusable assets [9].

The specification and representation of reusable software components requires classification techniques that allow automatic or assisted selection of such components. The selection can be automatic only if there exists an exact match between the requirements and the identified component properties. If not, we must collect the best matches and modify them or code converters in order to meet the requirements. Sometimes, if characteristics were not completely or correctly identified, matches that were less attractive, in a first approach, can prove to be the best solution. There is a need for a search engine that returns possible solution to our needs. It is not expected to provide the perfect, or even the best, match of the available alternatives. If alternative options exist, it should be possible to choose from a set of matches based on component parameters that were not a part of requirements query.

The medical equivalent is to provide a set of symptoms and get an ordered group of possible illnesses. Perhaps, the most probable answer, of the possible solutions presented, is not the solution. Some of the symptoms may not yet have been identified, and therefor omitted, leading to less accurate answers. Then, human intervention is needed actually check if the patient has the missing symptoms. Although, some of these symptoms may not allays be present. The final choice of treatment relies on the ability to select among possible candidate treatments, the best the doctor sees fit.

Our aim is to describe software components in a form that users can browse components in order to find out if they are a good match. Since these components may become very large and complex, we must provide a set of mechanisms and tools to help extracting relevant information. The user should be able to gather structural information, as well as associations and dependencies between components. In order to achieve a comprehensive description of each component, we use identifiers extracted from each stage of software development. Identifiers offer a very high level of abstraction, since they are used to materialize the idea of the developer through out the development process. By grouping related identifiers we achieve a compact description of each component. However, such a description might be too extensive for the software developer to cope with. Therefor, a set of filtering mechanisms must be available so the software developer can obtain only the data corresponding to the view that better answers its current query. The information provided by an identifier can be ambiguous in some views, so a set of attributes for each identifier may be introduced. Such attributes should help to clarify the context in which the identifier is used. Since the identifiers and attributes on their own may not be enough, and inspection of the actual data may be necessary. Each identifier or attribute may have a link to the actual data from where it was extracted.

The resulting system can be viewed as specialized naming service whose data is made of sets of identifiers extracted from the software process data.

2 Related Work

Software components can comprise information that ranges from requirements to code. The most simple way to describe a software component is textually, in English, usually through its reference manual. This may be good approach for high quality assets. These result, generally, from commercial libraries or frameworks. However, frequent inconsistency problems arise, even with high quality software, when documentation is out of synchronization with the software it describes. This may result from features or caveats added to new version of software and not reflected in the documentation. Sometimes, the documentation reflects what it was intended to do and not what is really does. But even when documentation is updated with the software, problems can arise from wrong interpretation of the text leading to incorrect usage of the software. Documentation problems have been around for a long time and a large effort has been made to make a steep learning curve [7]. Solutions to keep documentation updated exist, but the most reliable solutions require large human resources and reduce the final product market penetration due to the resulting high prices and time delays.

Documentation information is frequently long and difficult to browse in order to extract specific component characteristics. Classification techniques provide a simple and useful way to catalogue software components.

Simple classification schemes use enumerative techniques such as Dewey Decimal System but are, generally, too vague for efficient application because of their inability to record growing details. However, they can be useful as a first approach.

Other descriptive and reuse centered approaches have been available. Faceted classification schemes [15] uses several facets, or views, where each facet can have several terms. In order to classify an item, a term that best describes the item is chosen from each facet. Faceted classification schemes provide a better classification than enumerative schemes, since a component is no longer described by one term but many, one from each facet. More elaborate classification schemes require the use of properties or attributes.

Approaches based on software models and metrics [8] provide a much better base, since they are based on information extracted from the software process itself. This approach gives quantitative results from analysis process offering very valuable data. However, many of those quantitative values result from measures that can only be taken by what they are and do not reflect any measure of reusefulness. Since they are only hints, although very valuable, simpler and less expensive forms of achieving similar results we should investigate.

Cognition models offer a reverse engineering approach to program understanding [20]. All models use existing knowledge to build new knowledge about the model of the software that is under consideration. While cognition strategies vary, they all formulate hypotheses and them resolve, revise, or abandon them. Many of these models are based on exploratory experiments, and some of those models have been validated. While a lot of important work exists, most of it is centered around general understanding and small-scale code.

Most these methods lack high abstraction capabilities or rely on complex, non-uniform and difficult tasks to manage information. It should be possible to obtain different levels of abstraction in a uniform representation. The same representation model should be generic enough in order to cope with different types of information available in different software stages. A simple search mechanism can browse this uniform structure and highlight matches to simple queries. A further analysis of some components may use any of the above models, depending on the absence of specific information.

3 Description of components

In a first contact with a software component we must determine what is to be considered the relevant information and how to extract it. This information, should also be compact, so the effort analyzing it should be significantly smaller than analyzing the component itself. In this paper we exercise the proposed solution with a simple home finance accounting system. Current and savings accounts are available. In the currents accounts we register all transactions and savings accounts reward an interest rate.

3.1 Abstraction

Abstraction is a fundamental technique for understanding and solving problems [21, 13]. Abstracting raises the problems of software engineers being unable to determine how realistic is the abstraction and what level of detail can be considered unnecessary. Furthermore, the degree of realism or detail necessary can vary from one application to another, for the same entity. In this perspective, an abstraction can be considered good in some context and poor in another. Our solution to cope with such dependency is to record all information up to a configurable level of detail. Since this level is application dependent the degree of realism in the obtained abstraction is highly dependent on the selection process. Although abstractions to some well known applications can be easily tuned, due to previous experience, we can not foreseen other applications for the same entity. So, to describe an entity using a set of predefined identifiers [8] can be a limitation.

Our approach uses, in principle, no predefined identifiers or views. In fact, some predefined identifiers exist as entry points. The identifiers used to describe some entity are not the best identifiers from a set but, are instead the best description that could be found. The absence of stereotyped identifiers will provide greater expressiveness and realism to the abstraction but will make the selection process more difficult. Since comparing can not be performed on equal terms, is to determine the best available entity for some concrete case. The decision can no longer be based on the existence, or not, of some identifier, nor on its value. The solution found uses a classification process to store identifiers. As will be explained, identifiers performing similar tasks are supposed to be classified in a similar way. Therefor, it is expected that similar identifiers will show up in the same area. Due to their proximity a better judgment can be performed while retaining all the expressiveness of each identifier.

In our home finance accounting system it is expected that keywords like *account*, *current*, *savings* or *interest* will be major entry points into our model. Typical entry points include data types, functions or variables, since they provide the first level of abstraction. However, these entry points must be treated uniformly, since different approaches might model each concept by a different resource. In our example, the interest rate can be fixed for all savings account (constant), can be fixed for some groups of savings accounts (function), or can have different values from one account to another (variable).

3.2 Classification

The first stage, in order to solve our problem, consists of determining what is to be considered the relevant information and how to extract it.

By looking closely into the way we represent information we find out that at the higher abstraction level resort to names [6]. These name are used to describe views, objects, variables, functions, types, and so forth. More important is the fact that they retain their name from the requirement stage, down to the code being executed. The fact that new names came up or disappear as the development process is carried out can give useful information about deviations from the original requirements or analysis stage. These can be seen as the signature of the entity being focused.

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. By extracting we do not mean to remove identifiers from wherever they are but to retain a reference in model pointing to what the identifier refers to. In this process we end-up with two separate domains: an identifier domain and an entity domain (figure 1). The identifier domain should capture the essential information about the object, allowing most subsequent operations to be perform without the need to access the object itself. The entity domain will retain the object in the previous form, although the identifiers that might exist are ignored, since a duplicate exists in the identifier domain.

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

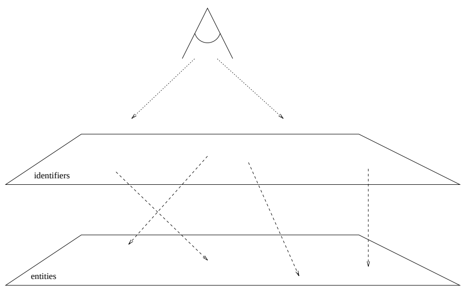

Figure 1: Extracted identifiers reference their entities.

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. Attributes can be manipulated as identifiers although they might not point to some object in the entity domain.

Our model relies on uniformity and economy of concepts for its expressiveness. The first makes life easier by giving less to remember, no special cases. The later aims to get the greatest power with the smallest number of concepts [18, 17].

4 Representation of components

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, it just gets pushed it into a lower level. Most important is the fact that hierarchies come natural to most people. So, when describing a problem whether in top-down or bottom-up approach, the solution comes up at the root of the hierarchy and set of simple problems at every leaf. Functional languages claim must of their expressiveness 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 2).

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 [10] 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 [16]. 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. 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.

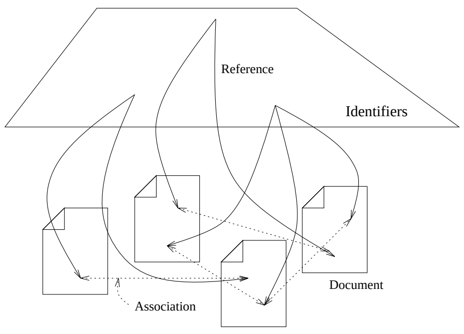

Figure 2: A naming tree links project documents.

In our accounting example, we expect *interest* to be a qualifier of *savings*. The classification *savings/interest* will represent *interest* as a component of *savings*, in all development stages, no matter how the system is modeled (constant, function or data variable). Other classification may include inheritance relations, such as *account/savings*, or composition descriptions, like *banking/account*.

4.1 Locating and Browsing

From the previous section we obtain a description of a system by representing all named entities in an hierarchical naming tree. Very large systems, when described in a detail, generate large name trees. Since the objective was to break down the complexity, powerful mechanisms are needed to extract information.

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.

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 [22]. In this model, however, both breadth and depth algorithms can be large enough, if many details are added. 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 to test the existance and location of some name or entity on the system. If a graphical representation is used,the problem is even more serious. Highlighting relevant information in large representation might be difficult for user's naked eye. A better approach is to generate an auxiliary model containing only relevant information, more or less the same way select operations are performed on tables of a relational database and end up with new table containing the results. In this case, a smaller naming tree containing only the nodes leading to the selected leafs is obtained. Then, browsing operations can be performed on this subset tree as in the original one but without being distracted by lots of data irrelevant for this stage of development.

In our example, the location of some identifiers provides important contextual information. The precise location of some identifier is not important at this example and is represented by *...* . In an object-oriented perspective, locations like *.../account/transactions* or *.../account/interest* suggests that the base class *account* will play some roles reserved for its derived classes *current* and *savings*. On the other hand, the same identifiers can appear in different locations, such as *.../current/transactions* and *.../savings/interest*, suggesting a more specialized approach. These identifiers' locations can be reached by browsing directly the actual code or browsing the design description. However, if many modules and different description languages are available, it might require specialized knowledge and additional time to hop from one place to another.

4.2 Hierarchical Modularity

The hierarchical naming tree can be broken into small hierarchies interconnected, but independent from each other, the named spaces (figure 3). 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.

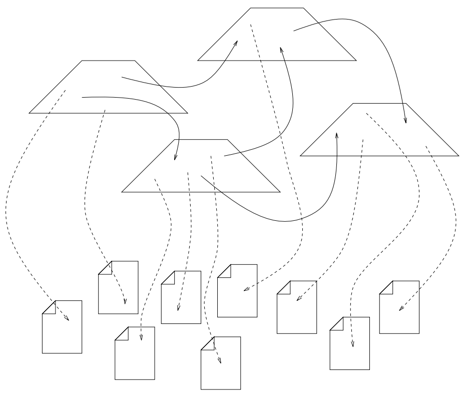

Figure 3: Connecting project name spaces.

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

The capacity to interconnect independent naming hierarchies allows the composition of multiple views. These views can represent different stages of development and different alternatives from the same stage of development. It can also be used to ensure a uniform naming space when data is gathered from different sources. This uniformity makes a comparison between code and design stages easier, since the same form of presentation is used, although information is collected into independent hierarchies.

5 Framework Environment

We began by determining what is to be considered the relevant information and how to extract it. Then a model to represent it was presented and operations to operate on it defined. Now we must explore ways to obtain conclusive results that can significantly contribute to improve the quality of the final product. It would be expected that a better rejection of entities, poor candidates in the selection of reusable components, as well as better choice when alternatives are available.

Many models give large set of constructs to built the solution for a problem. These models 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. This approach separates the model and its set of base operations from the high level operations needed by the software developer. The major advantage is not to restrict the software developer to fixed set of choices, making exploratory queries possible and the definition of new procedures easy to integrate.

However, the complete absence of operating procedures may inhibit the software developer of using all the capacities available. As a starting point three operations are available: location, selection and extraction. A location operation will return the context where on or more identifier are used. Since the same identifier may be present in different contexts many location can be returned. The set of location will generate a new hierarchical tree where the identifier being located is the same for all leaf nodes. The branches of this new tree will represent different views where the same identifier is being used. These views can represent different stages of development and different alternatives from the same stage of development.

Since the identifiers used do not come from a fixed set the search may have to based on their attributes. It is expected that some of its attributes will be same if two different identifiers perform similar tasks. The returned context consists on hierarchical positioning of the identifier in the tree. An analogy with the UNIX `find` command is possible since it may search name or attributes, although the attributes are limited to the file `i-node` description.

The selection operation will return the list of attributes of an identifier, while the extraction operation return the subtree starting at an identifier. These operations would return, for a complex system, too much information and require a location operation to follow. An extraction operation is used to limit the analysis to a limited set of views, by eliminating all others. A selection operation reduces the search to a certain level of detail, limiting subsequent searches to a certain granularity.

5.1 The Selection Process

The system we described so far must now be used to provide some useful results. We do not hope to make available all the information available in the software process documents in the identifier domain. The identifier domain can only be interpreted as a signature of the software component being described. However it is expected that it will reflect many of the characteristics, static and dynamic, of the system it models.

The first result can be interpreted as a more or less accurate measure of the complexity of the system. A very complex software component should need a large naming tree to describe it. So it can be used a simple metric that, nevertheless, can be reliable and portable between different stages of development; unlike lines of code that can only be used in the final stages and can give misleading results.

Simple context analysis can be done by extracting the context of a given identifier from a specific view. If the identifier is not previously known, the search must based one of its well known attributes. Context information will show the surrounding identifiers and give a glance of the environment where it is being used. Hopefully, this identifier information will be complete enough in order not to be necessary to look at the actual code or describing document.

If we extend the previous search to several views, a set of contexts, one for each view, will be gathered. The comparison of such contexts will highlight important information from these views. The views can represent different alternatives for the same stage of development or a development case. Deviations from initial requirements can be identified as different context paths or even the absence from some views. Very long paths relate, usually, to the add of excessive detail can result from very general components; while short paths can be associated with specialized software components.

When context information is insufficient to get precise conclusions, a select operation can provide additional data. This data can include other attributes or related identifiers that will enhance the function and environment where a given identifier is being used. Larger amounts of data can be obtained through an extraction operation. However, such extraction will provide, generally, too much information and it must be then be treated by a select operation.

This system provide no quantitative results and it requires a good understanding of the major software engineering concepts. The approach is, therefor, human-centered [14] since it relies entirely on the experience and interpretation capacity of the software developer. On the other hand, it provides lots of useful information in a compact form that otherwise can only be obtained through direct inspection of the code or higher level description.

5.2 Validation and Evaluation

A simple validation test was carried out using student projects. The same specification was given to all student groups. Each group delivered an UML [4, 5] graphical presentation of the analysis and design stages, and a C++ [12, 19] implementation. The test consisted of two part: a consistency checking and an interoperation test. The consistency checking should highlight inconsistencies with the given specification while the interoperation test should detect parts that could easily interchanged between different approaches. From the students perspective would measure the abstractions and their modularity. In the test case we used a complex version of the UNIX `grep` command that gave the context where an identifier was found. A simple example could include a structural search: class/instance/method/variable. The extraction of information from the analysis and design stages was performed manually into the hierarchical naming tree. The C++ extraction used some rudimentary tools and extracted only a limited set of identifiers.

The first result was given by direct analysis of the resulting tree. Missing information was evident on size of the resulting tree as well as some case of over-coding, by the huge tree size. Location of specific identifiers, representing what were expected to be key evaluation points, was a bit difficult. Although most groups used meaningful names, some others used difficult to track names. The major resulted in the fact that the software process and the resulting application was correct, the chosen names were just awkward. This made detection of inconsistencies error prone, since many false errors where found. Only later discussion with group members clarified the situation.

The approach proved very useful for well behaved cases, but misleading in some others. The existence of at least some predefined identifiers proved essential in order to keep references. This can, however, be difficult to achieve with outsourced components, unless manual classification is performed.

6 Conclusion

The use of identifiers proved to be is a good choice, it gives a good overview of the system. Some time there is the need to look at the actual data but it was expected that no abstraction can replace completely the object.

Hierarchy is good because is keep good notion of where we are. It must be specially attractive for software developer that are familiar with large hierarchical filing systems with deep nesting of directories, as The need to keep reference, absolute or relative, is specially useful in our approach.

The use of non-standard identifiers may be a drawback in location operations. The use of synonymous and different naming schemes make the search process difficult and error prone. Therefor, a set of base identifiers is need to catalogue key information within the system. However, the ability to add non predefined identifiers is an important feature in improving the expressiveness and enriching the descriptive power of the model.

The proposed operation allowed the extraction of conclusive results quickly. However, the amount of information based on which decisions had to be made may be excessive for larger projects. We hope that as we gain experience more targeted operations can be designed. Nevertheless, we fear that very targeted searched might exclude important information, leading to incorrect decisions.

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

[7] Greg Butler and Pierr Denommée. Documenting frameworks. In *8th Annual Workshop on Software Reuse*, March 1997.

[8] Gianluigi Caldiera and Victor R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24(2):61–70, February 1991.

[9] Pedro Reis dos Santos. Identifier based representation and management of software components. In *ECOOP'97 workshop on Modeling Software Processes and Artifacts*, pages 33–36, 1997.

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

[11] William B. Frakes and Christopher J. Fox. Sixteen questions about software reuse. *Communications of the ACM*, 38(6):75–87, June 1995.

[12] Stanley B. Lippman. *C++ Primer*. Addison-Wesley, Reading, MA, USA, second edition, 1991.

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

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

[15] Rubén Prieto-Diáz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):89–97, May 1991.

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

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

[18] 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.

[19] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, USA, second edition, 1991.

[20] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, August 1995.

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

[22] 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.