

Using the Web for Software Annotations*

Václav Rajlich, Srikant Varadarajan
Department of Computer Science
Wayne State University
Detroit, MI 48202
rajlich@cs.wayne.edu

Keywords

Documentation, redocumentation, incremental redocumentation, hypertext, World Wide Web, partitioned annotations, legacy systems, program comprehension.

Abstract

The purpose of software redocumentation is to recover comprehension of software and to record it for future use. This paper describes Partitioned Annotations of Software (PAS), where comprehension is recorded in hypertext and browsed by web browsers. The annotations for each code component are partitioned in order to keep different explanations separate, leverage the advantages of hypertext, and better support the processes of program comprehension. The paper describes a tool that parses code and generates PAS skeletons. The paper also describes a process of incremental redocumentation where comprehension of software is recorded incrementally during normal maintenance. The experience with PAS in an industrial project is summarized.

* This research was partially supported by a grant from Ford Motor Co. and by NSF grant CCR-9803876

1. Introduction

The purpose of software reengineering is to improve a specific quality of legacy software. The activities of reengineering address different aspects of legacy system deterioration. Among them, restructuring addresses the loss of structure in the program, retargeting addresses the obsolescence of the environment in which the program operates and redocumentation addresses the loss of comprehension of the code. This paper deals with redocumentation of legacy programs.

The goal of redocumentation is to recover comprehension of software and to record it for future use. Very often the documentation of legacy software is obsolete, inaccurate, or even nonexistent. The maintenance of such software is difficult. Each change to such software starts with a long and error-prone phase of software comprehension, where programmers try to put together the available bits of information in order to gain enough understanding to be able to carry out the change. Program redocumentation is an effort to record the understanding to make future changes easier and therefore to improve software maintainability.

As the cost of electronic media decreases, documentation is more and more often stored in electronic form [10]. Among electronic media, hypertext plays a prominent role because documentation is usually voluminous and the user needs only a small fraction of it [9]. Hypertext is particularly suited to this situation and allows the user to seek out the relevant parts of the documentation while skipping over the rest. It also allows the author of the documentation to add new information wherever it is available. For this reason, specialized hypertext documentation tools were developed [1], [4], [7], [8], [22]. The field gained a new momentum with the introduction of the World Wide Web, because hypertext has become a familiar and accessible technology. Tools such as the html language, browsers, etc., are widely available and understood. The same technology can be used in the redocumentation of legacy software.

In order to leverage the advantages of hypertext, the documentation has to be properly structured so that the user can quickly find the information relevant to his/her task. This paper describes "Partitioned Annotations of Software" (PAS) which provide such structure. PAS have developed from the technique for top-down program comprehension of [16], [14]. The principles of the PAS are explained in Section 2. Section 3 describes process of incremental redocumentation with PAS. Section 4 contains a description of the HMS tool that generates the PAS skeletons. Section 5

describes our experience with the PAS. Section 6 contains an overview of the related work. Section 7 contains the conclusions of the paper. Appendix A contains an example of a skeleton documentation created by HMS and Appendix B contains a completed PAS documentation for the same class.

2. Partitioned annotations of software (PAS)

The field of software comprehension is summarized in [17], [19]. It is an interdisciplinary field, spanning both software engineering and cognitive science. A number of theories describe how programmers understand code. The bottom-up theory is based on so-called chunking, where chunks are parts of the code which have an identifiable meaning and are recognized by the programmer. Chunks in turn can be parts of bigger chunks. The programmers who are using chunking as their comprehension strategy first recognize small chunks, then the bigger ones, etc., until enough of the program is understood.

The top-down strategy of comprehension is based on hypotheses about the software and their verification [3], [20]. The programmer first accepts a set of hypotheses and then looks for evidence (beacons). If beacons are found and hypotheses are confirmed, the program is understood. If hypotheses are not confirmed, they are discarded and another set of hypotheses is adopted. The most common strategy is an opportunistic strategy where the programmer freely switches among different comprehension strategies [20].

PAS are a notebook in which a programmer can record software understanding without regard for whether it was acquired in top-down or bottom-up fashion, whether it is complete or partial, whether it is based on design decisions or on maintenance hypotheses, or even whether it is confirmed or tentative. PAS are based on the existing structure of the software and do not require any restructuring. Each component of the code (i.e. each class or function) has its own annotation that explains its meaning. This annotation is divided into several partitions, each describing the component from a different point of view. Every component is annotated by the same partitions and every partition covers all components.

A programmer may want to use a wide variety of partitions, with various levels of abstraction and various kinds of contents. Since PAS are a hypertext in the style of the World Wide Web, there is no need to limit the number of partitions or their contents. However, it is necessary to have an

understandable and consistent system of partitions, so that the programmer can always find the desired information. The structuring of the hypertext documentation into partitions was first presented in [14]. Among the partitions, domain partition plays a special role.

Each program operates in a certain domain of application, and the ontology of the application domain consists of domain concepts and their relations. The concepts are supported by the program and they are important for the understanding the program [2]. For example, if the application domain is that of a point of sale, then the concepts are "credit card", "expiration date", "customer name", etc. These concepts are implemented in specific components in the code. The domain partition explains the concepts behind the components, and therefore makes these components comprehensible to everybody who understands the domain. Comprehending the concepts and their implementation in a program is important because very often the requests for change are formulated in terms of the domain concepts.

Another important partition is the dependency partition. Components of the program are dependent on other components. The dependencies are found by code analysis, and browsers are tools used to discover and visualize them [13], [24]. However the programmer often needs more than just the dependency graph among the components, he or she needs to know the meaning of the dependencies, i.e. needs to know what particular service the supporting components provide or what particular service the dependent components require. The dependency partition of the PAS records the comprehension of the dependencies.

Besides domain and dependency partitions, the PAS may contain additional partitions based on the needs of the project and the programmers. Examples are descriptions of function arguments, explanations of the algorithms and programming techniques, recommendations and warnings about the code, comments on the quality of the code, comments on the completeness of the testing, etc.

The architecture of the PAS is in Figure 1. It contains the root annotation that is the entry to the PAS. It contains a list of all of the components of the program or a dependency graph of the components. The user can select any specific component and the browser will display its annotation. The annotation has a pointer or pointers to the code of the component. When browsing through the PAS, the user moves between the code and the annotations using both "forward" and "backward" directions. The code is a leaf in the tree of annotations and does not

need any embedded hypertext links. It also does not need any change when annotations change and hence the code update and the annotations update are independent of each other. An example of an annotation is in Appendix B.

3. Incremental redocumentation with PAS

The process of PAS creation is either a whole-scale effort, where the whole system is redocumented in one phase, or an incremental effort that is embedded into the normal software maintenance. Since incremental redocumentation is an important approach, we are going to explain it in more detail here.

During incremental redocumentation, the PAS are created opportunistically step by step. Whenever an insight into the software is gained, it is recorded in the PAS. The business case for incremental reengineering was presented in [12] and applies also to incremental redocumentation. Incremental reengineering, as opposed to whole scale reengineering, does not require any large up-front investment that is put at risk. It starts with small investment and provides an early payoff. It also offers an early feedback about the quality and effectiveness of the reengineering effort and this gives an opportunity to fine-tune or redirect the effort before major expenses are incurred. A particularly important opportunity to record software understanding arises at the end of a software change.

Process of software change [21] is in Figure 2. During the early phases of the change, the programmer gains comprehension of a particular aspect of the system that is to be changed. That comprehension is a valuable commodity and there is a risk that it will be lost at the end, when the programmer turns his or her attention to the next change that may deal with a different part of the code. Therefore, in our process, a new phase of redocumentation is added to the end, and in it the programmer records the program comprehension gained during the change. In this way, the redocumentation is a part of routine maintenance, and it is based on comprehension that the maintenance programmer has to acquire anyway. Compared to the effort needed to comprehend, the recording of comprehension is only a small fraction of the cost.

After repeated changes, a substantial amount of documentation is accumulated. The accumulated documentation is concentrated in those parts that have been visited most often in the past, and are therefore most likely to be visited in the future. Therefore incremental redocumentation directs

the attention of programmers and managers to the components with the highest payoff.

Standard World Wide Web browsers like Mosaic, Netscape, etc. are used to browse the PAS annotations, removing the need to create specialized browsing tools. The annotations can be edited by standard text editors or by html editors such as Word 97. However, a tool is needed for the creation of the PAS, in order to guarantee uniformity of the annotations. Many parts of the PAS annotations can be extracted from the code, because they reflect the structure of the components of the code. We implemented a tool called Hyperprogram Management System (HMS) that generates skeletons of PAS for programs written in C++.

4. HMS: A Tool generating PAS skeletons.

HMS is a tool that parses existing code written in C++ and generates skeletons of annotations for classes and class members [18]. The function of HMS is explained in this section as a sequence of HMS windows and programmer's actions.

The first window of HMS is in Figure 3. In it, the user selects the directory of a C++ program to be annotated. In the left part, the window gives an option to select some or all of the files in the directory. HMS parses the selected files and the right part of the window displays the classes found in the files. The user can choose which member functions or data members are to have their own annotations. The selection in Figure 3 indicates that only public and protected function members are to be annotated. (The classes of the program are always annotated and do not appear as a selection in Figure 3.)

In the next step, the user specifies which partitions should be generated. There are several partitions that are always generated. They are:

- domain and dependencies partitions for the classes
- domain, algorithm, and argument partitions for member functions

Figure 4 contains the window in which the user selects additional partitions, beyond those always generated. The only additional partition entered in Figure 4 is "Author's comments" for classes. After the user makes the selection, the tool parses all the selected files and creates skeleton PAS files written in html. The files contain blank spaces for any documentation the programmer may want to insert.

Appendix A gives an example of a skeleton generated by HMS. It contains several partitions, one selected by the user ("Author's comments") and others automatically generated for each PAS. Some partitions have blank entries while others are automatically filled. Blank entries belong to partitions that cannot be extracted from the code and have to be entered manually. Examples of such partitions are "Domain" and "Author's comments". In contrast, the graph of dependencies and the member function list are both fully extracted from the code. All links to the code of the class and the definition of the class are also fully extracted. The annotations for the class dependencies and for the function arguments are partially extracted from the code and partially entered by hand. A standard editor is used to complete the PAS skeletons. An example of a completed PAS annotation for a component is in Appendix B. The annotations can be browsed by World Wide Web browsers like Mosaic, Netscape, Explorer, etc, see Figure 5.

5. Experience with PAS

PAS has been used at Wayne State University in two different situations: In software engineering courses, it has been used to document design decisions during student projects. In research and industrial projects, it has been used as a tool for the redocumentation of legacy software. This section describes our experience with these two uses of PAS.

When PAS were used to document new student projects, the partitions of the PAS were used to record design decisions and other properties of the projects. Design decisions depend on the specific design methodology of the project, in our case OODG [15]. Over several years, approximately 100 projects of an average size of 1,000 lines of code were annotated using PAS. In these projects, we found that the authoring of PAS annotations is an art that requires training. It is related to the art of hypertext authoring, as described for example in [11]. As in any writing activity, there is a tradeoff between difficulty assumed by the writer and difficulty experienced by the reader. Disorganized annotations make reading hard, but excessive demands on the style of annotations may make them difficult to write. We observed that it was difficult for some students to separate the domain annotations from the programming terminology. They tended to pollute domain annotations by programming terms like "pointer", "list", etc. The polluted domain annotations defeat the purpose of the partitions and make reading harder, particularly when the domain annotations are used to locate concepts in the code as in [2].

In the research and industrial projects, we used PAS to redocument legacy software. The specific

projects ranged from research case studies to industrial production code and involved programs written in both C and in C++. One of the projects was maintaining and evolving the family of VIFOR browsers [16], [13]. Another was a project at a car company aimed to reengineer approximately 100,000 lines of C++ [6]. Another project was a research case study of evolution where Mosaic code was used as a test bed for location and propagation of change [5]. Finally, PAS were used to redocument approximately 1,000,000 lines of C++ in a Point-of-Sale system developed by a local software company. In all of these cases, design decisions were long forgotten and PAS were used to record the current understanding of the software. We used the incremental redocumentation strategy and recorded the understanding gained during changes to the software.

Our experience with incremental redocumentation indicates that there is a certain "citizenship" required of the programmers on the project. The fact that the programmers record their understanding at the end of the change means that they will not immediately benefit from this extra work. It is the next programmer attempting the next change who benefits. As often happens in these situations, this citizenship cannot be taken for granted and the people have to be reminded and checked to see whether they did the expected task.

In one specific industrial project, we dealt with a shrink-wrap software package for Point-of-Sale, with over one million lines of C++ code running in Windows, which was developed by a local software company. The original software was written under strong competitive pressure and as a result did not have any technical documentation. The entire software maintenance was relocated offshore to India and a totally new team of programmers was entrusted with the task of maintaining the software. As the maintenance work started, the programmers used the PAS methodology and HMS tool to redocument parts of the code. Examples in Appendix A and Appendix B are from this project. Appendix A contains skeleton of annotation for class CcardVals, and Appendix B contains a completed annotation explaining the class. The class contains the data from the credit card that was swiped at the Point of Sale terminal. The information that the maintenance programmer needs about the class is contained in the partitions.

The experience from this project supports the business case discussed in [12]. Over the last 15 months of maintenance work, almost 40% of the system had been incrementally redocumented. This was achieved during routine maintenance with an investment of approximately 15% extra time. The 15% overhead is derived from programmer's timesheets, which are done for client

billing. The timesheets include the start time and end time for each activity, the activity code (such as study, coding, review, testing, documentation, etc.), the module name being maintained, etc. The low overhead of PAS creation is explained by the fact that the programmers invest in program comprehension anyway, and only an extra 15% is needed to record the results of comprehension in the PAS form.

The benefits that the company reaped for this relatively small investment include reduction in the training time for new programmers, and increased flexibility in scheduling programmers for the individual tasks. The reduction in training time is significant because over the last 15 months, 5 people have left and 7 new people have joined the project. If a programmer leaves, the knowledge about the code is preserved in PAS.

Even more important is the flexibility in scheduling people for the individual tasks. In the past, people on the project were “associated” with specific modules. However the ideal situation is an ego-less programming [25] where anyone is able to do any change in any module. Though the project has not completely achieved this, the current accumulation of PAS allows increasing flexibility in scheduling. If the pending task deals with annotated modules, it is no longer necessary to wait for a specific person to complete his or her current task and the pending task is assigned to someone who is available.

The experience with PAS has been so positive that the company now employs two trainee programmers whose only job is to create, maintain and fine tune the PAS documents. They follow the following process: The programmer who has made the change provides an informal ‘Release Note’ that describes the change. The trainee programmers start with this document, read the source code, and write the documentation. They interview the maintenance programmer in case of uncertainty, but since the maintenance programmer is a more valuable resource, the management places certain limits on this activity. The work on PAS prepares the trainees for their next position as programmers.

6. Related work

Hypertext annotations of software have been studied in [1], [4], [7], [8], [22]. The PAS differ from that work by the fact that they are partitioned, i.e. each annotation for each software component consists of several partitions, each describing the component from a different point of

view. The programmer visits only those partitions he or she needs. In contrast, the monolithic annotations of [1], [4], [7], [8], [22] provide one large annotation for each component, and the programmer has to search for the desired information within it. This search complicates process of program comprehension. Since monolithic annotations will be repeatedly read in their entirety, programmers are reluctant to enter unconfirmed or subjective information. However in some situations, this may be the only available information and its loss lessens the maintainability of software. PAS were developed within the context of program comprehension research [14].

Hypercode of [23] is a different approach to using the web for software documentation. In hypercode, the code and documentation are crossreferenced by hypertext links, which point also from code to documentation and between the components of the code. Hence compared to PAS, hypercode provides greater support for the browsing, at the expense of more difficult updates of the links when a change is made. Although partitioned annotations have not been explored in the context of hypercode, they are fully compatible with it and therefore these two concepts are complementary.

The generator of annotations HMS is based on program analysis that extracts software components and their dependencies from the code. There has been a substantial amount of work done in program analysis. The closest work deals with the browsers that extract components and dependencies from the code, store them in a database, and allow the programmer to query the database. Often they display the results graphically, and they allow browsing from one component to neighboring ones, see for example [16], [24].

7. Conclusions and future work

In this paper, we presented Partitioned Annotations for Software (PAS). PAS divide annotations for individual software components into partitions of a specialized nature so that the programmer can easily find the specific information he or she needs.

As mentioned earlier, one of the difficulties when using PAS is the fact that they require a certain citizenship from the programmers creating them. The idea to be explored in this context is to control the quality of annotations by walk-throughs done by peers. They are the prospective users of the annotations and hence they can accurately judge their quality and future usefulness. Since

annotations are easy to change and their change does not require changes to the code or recompilation, we speculate that the annotations can be improved during walk-throughs.

We also observed that annotating the program and maintaining it require two different sets of skills. One of the options we explored is to have specialized annotators. They are experts in annotation authoring and are able to write understandable annotations based on information provided by the maintenance programmers. The annotations are their main contribution to the project, therefore they are likely to invest their best effort in PAS. This practice is analogous to the one already employed by writers of user manuals and user help systems. Their expertise centers on communication with the human users and PAS require similar expertise. As mentioned in Section 5, specialized writers of annotations were employed with the Point-of-Sale system and produced effective annotations of approximately 40% of the software.

Another idea to be explored is the extraction of existing comments from legacy software and filing them into the PAS. This extraction is strongly dependent on coding conventions and may differ widely between projects, therefore it was not pursued in the current version of PAS. The comments particularly suited for this purpose are header comments that give a history of the file and its updates.

In conclusion, legacy systems have many problems, one of them being the loss of system comprehension. PAS allow programmers to record their understanding of the system, and they support doing so incrementally. PAS are particularly effective in teams with high turnover, because the new programmers frequently visit unfamiliar code and PAS greatly facilitate the understanding of that code. For the same reason they are also very effective with understaffed teams, where each programmer is responsible for more code than what he or she can remember. They also facilitate ego-less programming practice where any programmer can make any change to any component. In all these cases, PAS help preserve program comprehension. This in turn helps preserve or improve the maintainability of these systems.

References.

- [1] Bigelow, J. "Hypertext and CASE." IEEE Software, Mar.1988, 23-27
- [2] Biggerstaff, T.J., B.G. Mitbender, D.E. Webster, Program Understanding and the Concept Assignment Problem, Communications of ACM, May 1994, 72-8
- [3] Brooks, R., "Towards a theory of the comprehension of computer programs." International

Journal of Man-Machine Studies 18, 1983, 543-554.

[4] Brown, P., "Integrated Hypertext and Program Understanding Tools," IBM Systems Journal, Vol. 30, No. 3, 1991

[5] Chen, K., Rajlich, V., Program Comprehension during Change Planning, unpublished report.

[6] Fanta, R., Rajlich, V., Reengineering Object-Oriented Code, Proc. 1998 IEEE International Conference on Software Maintenance, 238-246.

[7] Fletton, N., Munroe, M., Redocumenting software systems using Hypertext Technologies, Proc. 1988 IEEE Conference on Software Maintenance, 54-59

[8] Horowitz, E. and Williamson, R.C., SODOS: A Software Documentation Support Environment, IEEE Transactions on Software Engineering 12 (11) (1986) 1076-1087.

[9] Lakhota, A., Understanding Someone Else's Code: An Analysis of Experience, J. Systems and Software, 1993, 269-275.

[10] Lehner, F., Software - Dokumentation und Messung der Dokumentationqualität, Carl Hanser Verlag, Munchen, 1994, ISBN 3-446-17657-8.

[11] Martin, J., Hyperdocuments and How to Create Them, Prentice Hall, Englewood Cliffs, NJ, 1990.

[12] Olsem, M.R., An Incremental Approach to Software Systems Re-engineering, Software Maintenance: Research and Practice 10, 1998, 181-202.

[13] Rajlich, V., N. Damaskinos, W. Khorshid, P. Linos, J. Silva, Visual Support for Programming-in-the-large, Proc. IEEE Conference on Software Maintenance, 1988, 92-99.

[14] Rajlich, V., R. Gudla, J. Doran, Layered Explanations of Software: A Methodology for Program Comprehension, Proc. 1994 IEEE Workshop on Program Comprehension, 46-52.

[15] Rajlich, V., Decomposition/Generalization Methodology for Object-Oriented Programming, J. Systems Software, 181-186, 1994.

[16] Rajlich, V., Adnapaly, S.R., VIFOR 2: A Tool for Browsing and Documentation, Proc. 1996 IEEE International Conf. on Software Maintenance, 296 - 300.

[17] Robson, D.J., Bennett, K.H., Cornelius, B.J., and Munro, M. "Approaches to Program Comprehension." Journal of Systems and Software 14 (1991), 79-84.

[18] Srikant, V., Hyperprogram Management System, M.S. thesis, in progress.

[19] Storey, M.A.D, Fracchia, F.D., Muller, H.A. Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization, Proc. 1997 IEEE International Workshop on Program Comprehension, 17-28.

- [20] von Mayrhauser, A., Vans, A.M., On the Role of Hypotheses During Opportunistic Understanding While Porting Large Scale Software, Proc. 1996 IEEE Workshop on Program Comprehension, 68-77.
- [21] Yau, S.S., Collofello, J.S., Some Stability Measures for Software Maintenance, IEEE Trans. On Software Engineering, SE-6, No. 6, Nov. 1980, 545-552.
- [22] Younger, E.J. and K.H. Bennett, Model-Based tools to Record Program Understanding, Proc. 1993 IEEE Workshop on Program Comprehension, 87-95.
- [23] Kaiser, G.E., Dossick, S.E., Jiang, W., Yang, J.J., An Architecture for WWW-based hypercode Environments, Proc. International Conf. On Software Engineering, 1997, 3-13.
- [24] Muller, H.A., Klashinsky, K., Rigi - a system for programming-in-the-large, Proc. International Conf. On Software Engineering, 1988, 80-86.
- [24], Schach, S., Classical and Object Oriented Software Engineering, 4th edition, WCB/McGraw Hill, 1999, 92-93.

Appendix A: Example of skeleton annotation generated by HMS

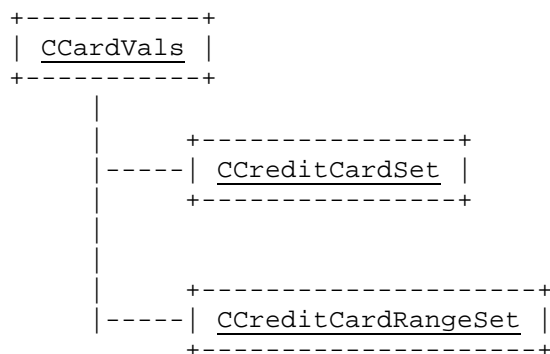
Class CCardVals

- Definition of the class
- Code of the class
- Domain annotation for the class
- Dependencies of the class
- Dependency Annotation
- Author's Comments
- Member functions of the class

Domain annotation for the class

// Empty

Dependencies of the class



Dependency Annotation

CCreditCardSet

// Empty

CCreditCardRangeSet

// Empty

Author's Comments

// Empty

Member functions of the class

- BOOL InitializeValues()
- BOOL IsValidCard(CString strCardNo,CString strCardExpDate,CARDVALS_STRUCT *retCardStruct=NULL)
- BOOL ParseAndGetInfo(CARDINFO_STRUCT& cardInfoStruct)
- BOOL IsValidExpDate(CString strCardExpDate,CString& strMsg)

BOOL InitializeValues()

- Domain annotation
- Algorithm

Domain annotation

// Empty

Algorithm

// Empty

BOOL IsValidCard(CString strCardNo,CString strCardExpDate,CARDVALS_STRUCT *retCardStruct=NULL)

- Domain annotation
- Algorithm
- Representation annotation for each argument

Domain annotation

// Empty

Algorithm

// Empty

Representation annotation for each argument.

- strCardNo: // Empty
- strCardExpDate: // Empty
- retCardStruct: // Empty

BOOL ParseAndGetInfo(CARDINFO_STRUCT& cardInfoStruct)

- Domain annotation
- Algorithm
- Representation annotation for each argument

Domain annotation

// Empty

|-----| CCreditCardRangeSet |
+-----+

Dependency Annotation

CCreditCardSet

This class accesses the CREDIT_CARD table to fetch a list of all the credit cards accepted by the system. For e.g. the system determines if American Express card is accepted by the store.

CCreditCardRangeSet

This class accesses the CREDIT_CARD_RANGE table to fetch the valid range of credit card numbers. Each card, like VISA, MasterCard, American Express etc., has a predefined starting numbers. VISA starts with a 4, MasterCard starts with a 5 and American Express starts with a 3. By looking at the credit card number, this class tells the system which card it is.

Author's Comments

- The system currently doesnt enforce a 4 digit year entry because the credit card processors do not require the year in 4 digit format.
- This entire operaton must be made into a dynamic library in order to use it with other modules.

Member functions of the class

- **BOOL** InitializeValues()
- **BOOL** IsValidCard(CString strCardNo,CString strCardExpDate,CARDVALS_STRUCT *retCardStruct=NULL)
- **BOOL** ParseAndGetInfo(CARDINFO_STRUCT& cardInfoStruct)
- **BOOL** IsValidExpDate(CString strCardExpDate,CString& strMsg)

BOOL InitializeValues()

- Domain annotation
- Algorithm

Domain annotation

This function reads up a list of all the valid credit cards and ranges from the database.

Algorithm

Open the CREDIT_CARD, CREDIT_CARD_RANGE table.

Read all the values into CREDIT_CARD_STRUCT and add it to m_CreditCardVals linked list.

BOOL IsValidCard(CString strCardNo,CString strCardExpDate,CARDVALS_STRUCT *retCardStruct=NULL)

- Domain annotation
- Algorithm
- Representation annotation for each argument

Domain annotation

This function takes the swiped credit card number and expiration date and validates it and returns the name of the credit card

Algorithm

Check if the expiration date is valid.

Check if the starting range of the number entered falls within the range of any of the cards in m_CreditCardVals list.

If found, return true and fill retCardStruct with appropriate values, such as the credit card name.

if not found return false.

Representation annotation for each argument.

- strCardNo: This is of the type CString. The credit card number is passed the the function through this variable.
- strCardExpDate: This is of the type CString. The credit card expiration date is passed the the function through this variable.
- retCardStruct: This is of the type CARDVALS_STRUCT* . The function returns the credit card name through this variable.

BOOL ParseAndGetInfo(CARDINFO_STRUCT& cardInfoStruct)

- Domain annotation
- Algorithm
- Representation annotation for each argument

Domain annotation

When swiped, the credit card information is a long string of numbers and characters. This function parses the string and extracts the credit card number, expiration date and the customer name from this string.

Algorithm

```
%B1234 567890 12345^ABCDEFGHJK/L ^9912123456789?
```

The swiped information is in the above format.

Between %B and the ^, is the credit card number.

Between the two ^ is the customer last name followed by the customer first name.

The 4 characters after the second ^ is the expiration date in YYMM format

Representation annotation for each argument.

- cardInfoStruct: This is of the type CARDINFO_STRUCT&. The credit card number, expiration date and customer name is returned in this structure.

BOOL IsValidExpDate(CString strCardExpDate,CString& strMsg)

- Domain annotation
- Algorithm
- Representation annotation for each argument

Domain annotation

This function validates the expiration date.

Algorithm

Check if the string is 4 characters long.

Check if month is between 1 and 12.

Representation annotation for each argument.

- strCardExpDate: This is of the type CString. 4 digits of yymm.
- strMsg: This is of the type CString&. If the validation was not successful, this contains the err message.

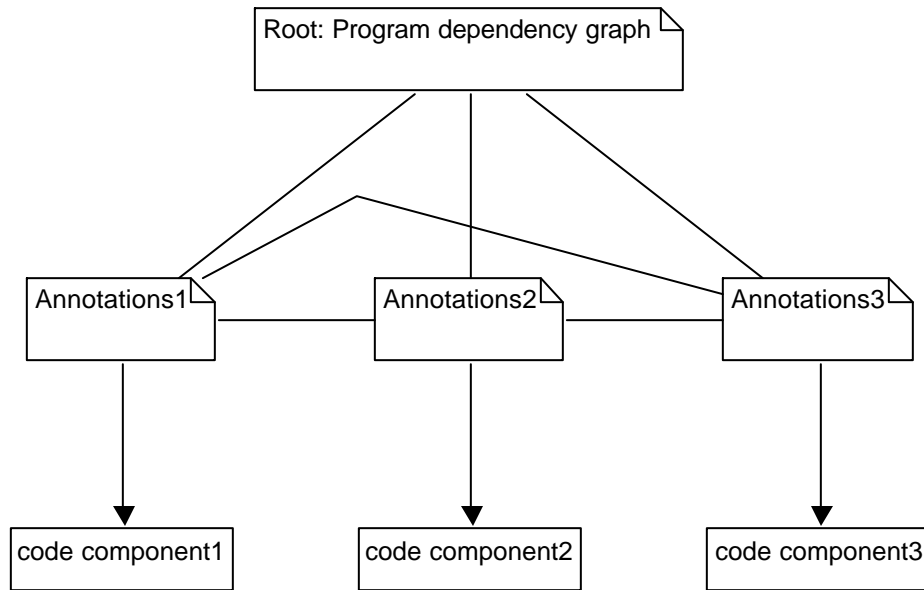


Figure 1.

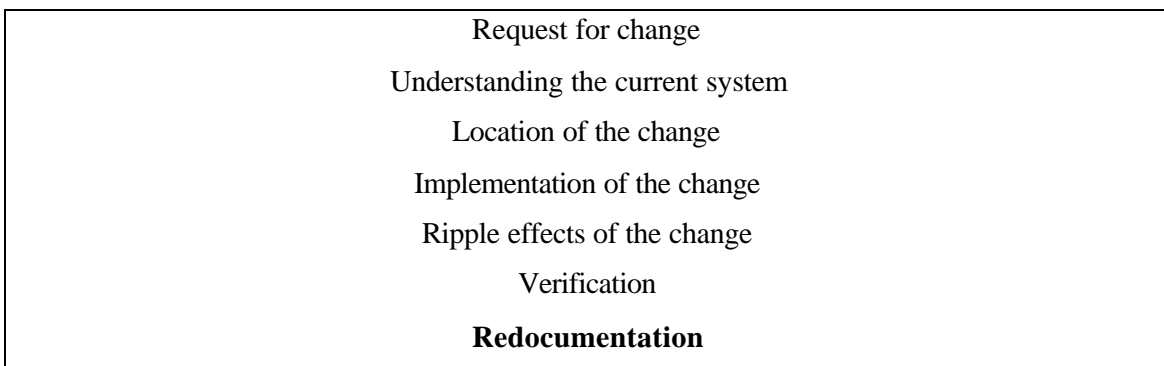


Figure 2.

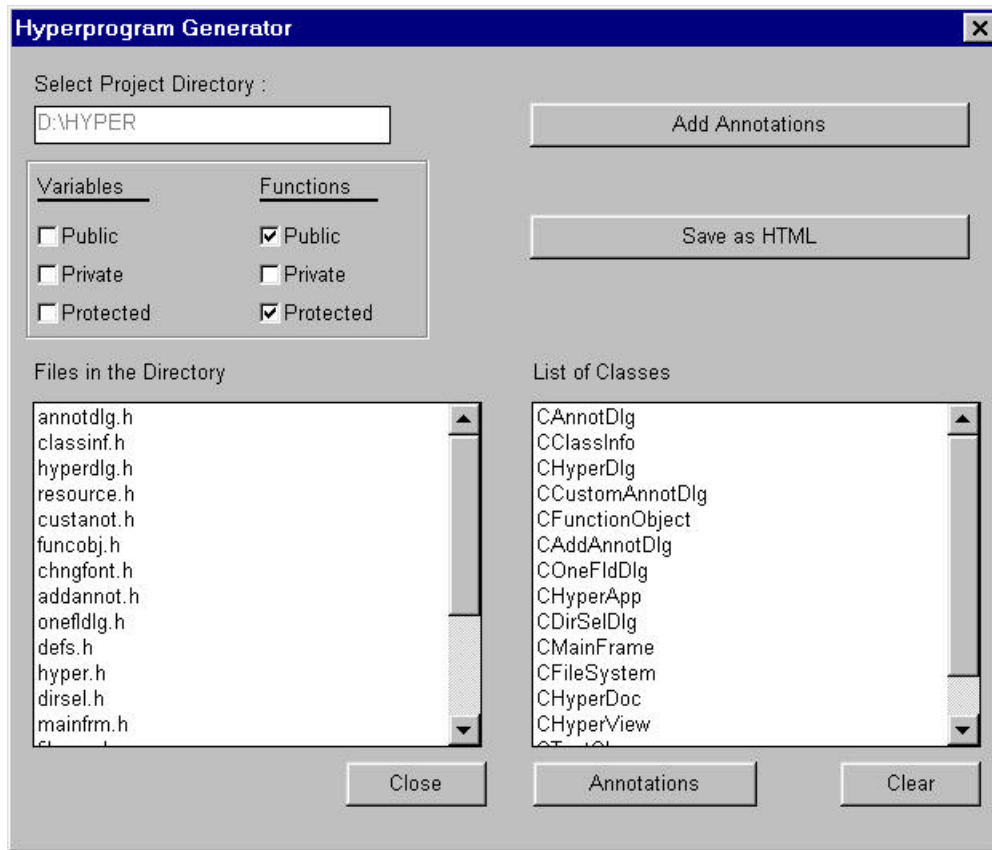


Figure 3.

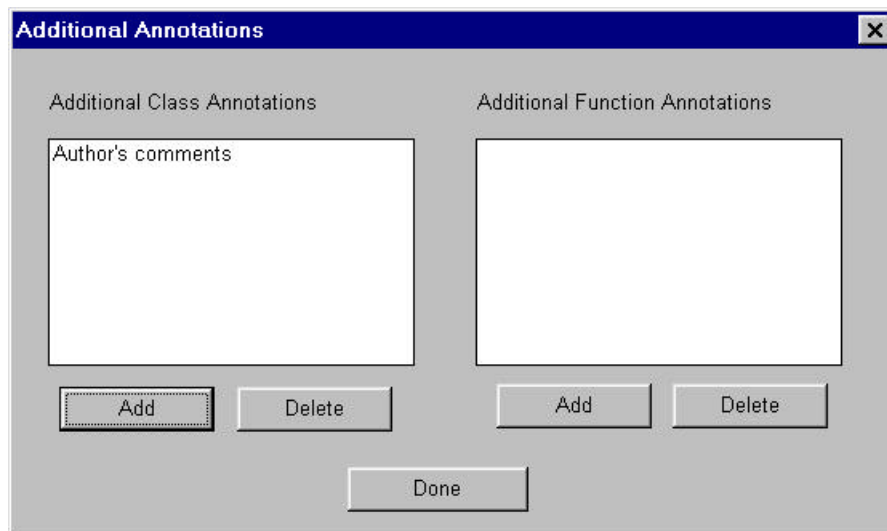


Figure 4.

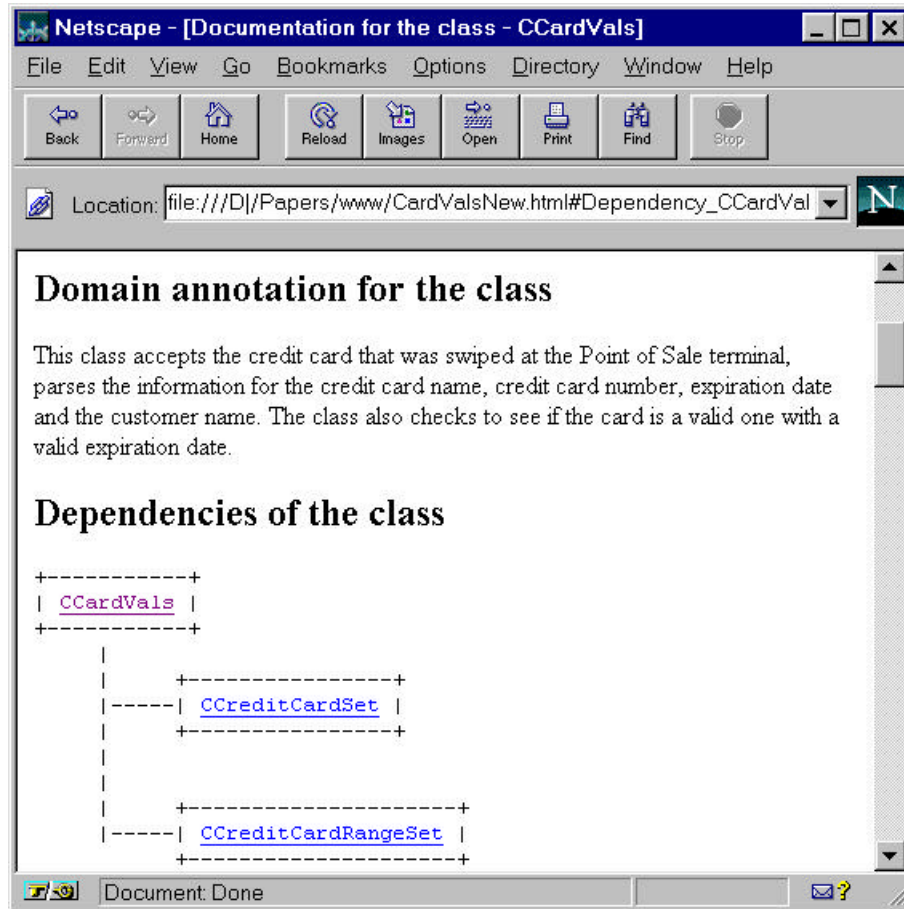


Figure 5.

Václav Rajlich is a professor and former chair of the Department of Computer Science at Wayne State University. Before that, he was an associate professor of Computer and Communication Science at University of Michigan in Ann Arbor, and software manager at the Research Institute for Mathematical Machines in Prague, Czech Republic. He was a founder and general chair of IEEE International Workshop on Program Comprehension, general chair of IEEE International Conference on Software Maintenance, and executive vice-president of Technical Council on Software Engineering of IEEE Computer Society. His current research interests include software maintenance, evolution, and comprehension.

Srikant Varadarajan received a B.E. in Electrical Engineering from the University of Roorkee, Roorkee, India in 1990 and currently works for Access Computers as a lead developer. He is a M.S. student at the Wayne State University, specializing in software engineering. His areas of interests include software design, maintenance and documentation.

