1999

CASCADE: Computer aided synchronization of code and documentation

Mark Roth

CASCADE: COMPUTER AIDED SYNCHRONIZATION OF CODE AND DOCUMENTATION

by

Mark Roth

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Science in Computer Science

Rochester Institute of Technology

August 12, 1999

Approved by _____

Michael J Lutz

_____

J. Fernando Naveda

_____

Peter G. Anderson

Permission From Author Required

Title of thesis:

CASCADE: Computer Aided Synchronization of Code and Documentation

I, Mark Roth, prefer to be contacted each time a request for reproduction is made. If permission is granted, any reproduction will **not** be for commercial use or profit. I prefer to be contacted via email at the following address:

markroth@bigfoot.com

Date: 8/12/1999

Signature of Author: _____

Rochester Institute of Technology

Abstract

CASCADE: COMPUTER AIDED SYNCHRONIZATION OF CODE AND DOCUMENTATION

by Mark Roth

Chairperson of the Supervisory Committee: Professor Michael J. Lutz
Department of Computer Science

Complete, accurate and up-to-date documentation is a critical factor in the development and maintenance of robust software products. Often, however, the documentation and the product diverge over time, leading to inconsistencies that are the source of confusion and faults, which can lead to failure.

This thesis addresses the inconsistency problem by describing a standard method for the synchronization of documentation with code. As part of the thesis, existing research in the field of Document Engineering is surveyed and related to the method being developed.

One goal of this approach is to create straightforward integration with existing languages, coding styles, and documentation formats. In particular, the method supports integration into existing development tools and software processes.

As a proof of concept, an implementation of the CASCADE system has been developed. The HTML documentation and the Java™ source code for the system itself is used as an example to demonstrate the capabilities of CASCADE. Hypothetical changes to the code and documentation are analyzed both with and without the use of the synchronization method. The prototype system has been found to enhance source code and documentation navigation considerably.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to Susan for her patience and support throughout the writing of this thesis. Much appreciation also goes out to Dr. Anderson for his guidance and cooperation in the final months leading up to the completion of this thesis.

Finally, the author thanks the reader in advance for their interest in reading this thesis. It is hoped that it will be of some value.

GLOSSARY

action – The result of an event, in the context of a relationship.

CASCADE – Computer Aided Synchronization of Code And DocumEntation. Also refers to any system that implements the CASCADE specification.

code-document synchronization – Ensuring that all relevant documents are synchronized with all relevant code.

documentation – In the context of this thesis, documentation usually refers to textual documents that describe the software product. However, it may occasionally refer to other related items, such as code and diagrams that supplement documents.

entity – A significant entity, subentity, or a project.

entity type – A category for a significant entity (e.g. "HTML Document" or "Java Source Code").

event – A change that has been made to one or more significant entities in the system. Standard causes for an event include a node being added, deleted, or modified, or immediate or indirect subentities being added, deleted, or modified.

flag node – A user-defined subentity whose presence indicates some condition is true. These subentities exist to make event declarations easier and more efficient and can be set directly from actions. They are not required for a minimal CASCADE implementation.

formal name – A full name that uniquely describes a node.

inter-code synchronization – Ensuring that all code and code comments are in a consistent state across all classes in the system.

inter-document synchronization – Ensuring that all documentation, not including code, is in a consistent state across all documents in the system.

intra-code synchronization – Ensuring that all code and code comments within a single source file are in a consistent state.

intra-document synchronization – Ensuring that a single document is in a consistent state.

KLOC – Short for "thousand lines of code." In this context, a LOC is a line in a source file, including comment lines, but not including blank lines.

NCKLOC – Short for "non-commented thousand lines of code." In this context, a NCLOC is a line in a source file, not including comment lines, and not including blank lines.

node – A single element that can be referred to by a formal name. This includes projects, significant entities, and subentities.

physical action – The physical results of an action. Standard physical actions include "execute system command," "display message," "email message," "set flag," and "unset flag."

project – Maintains information about significant entities and inter-relationships between nodes.

reflection – Refers to the fact that a relationship is an entity within the system, so that relationships can be constructed where the source of the event is another relationship's existence. Reflection is not required for a minimal implementation of CASCADE.

relationship – An aggregation of an event and an action, representing how two or more nodes are related.

significant entity – A single document, or a single source code project. Can also be expanded to include other entities that a user wishes to keep synchronized in some way, such as diagrams.

subentity – A significant entity is broken down in a hierarchical fashion into subentities, allowing for fine-grained referencing of portions of a document or source code.

subflag – A subentity of a flag node. If a subflag is set, its parent, by definition, must also be set.

synchronization – The adjustment of two or more entities such that they are not in conflict with each other, and that they are all up to date.

Chapter 1

INTRODUCTION AND MOTIVATION

The past few decades have brought about many new software design paradigms, CASE tools, and Integrated Development Environments, all of which are designed to make the software development process easier and less error prone. However, no advance thus far has made the need for complete and accurate documentation any less important. The field of Document Engineering attempts to define processes and methods for developing our ability to communicate ideas effectively. It is advanced by both individual researchers, and special interest groups such as ACM's SIGDOC.

There are an increasing number of issues that Document Engineering tries to address. For example, how can we take advantage of new hyperlinking technologies while still maintaining the ease of use, penetration, and robustness of paper documentation? How can we integrate multimedia content with documentation in an effective manner?

This thesis addresses the classic issue of synchronization between code and documentation. It has been shown that complete, accurate and up-to-date documentation is a critical factor in the development and maintenance of robust software products. Often, however, the documentation and the product diverge over time, leading to inconsistencies. These inconsistencies, which may occur between different documents or even within code segments, are a primary source of confusion and faults during maintenance and enhancement.

It is important to realize that that this is not a purely academic problem. Indeed, document synchronization problems have been the source of potentially serious accidents. For example, in 1994 a pilot was completing a local area maintenance test flight when, during an otherwise perfect landing, the plane nosed over and crashed. The problem was discovered to be due in part to an outdated maintenance manual that failed to emphasize particular instructions for how to release the parking brake properly [10]. In another incident, a DC-9 was substantially damaged during cruise flight due to the left engine's top thrust reverser door becoming partially detached and striking the engine's thrust reverser door repeatedly. The problem was later concluded to be due to an inconsistency between the manufacturer's parts manual and maintenance manual text. The maintenance manual text indicated that one washer should be used for a particular bolt, whereas the corresponding illustration showed two washers for the bolt [11].

The inconsistency problem is addressed in this thesis by specifying a standard method for the synchronization of documentation with code. This method is based on the use of external hierarchical XML documents to track changes to fine granularities, record dependencies, and to help automate reconciliation of code changes with support documentation and with other source code.

One goal of this approach is to create straightforward integration with existing languages, coding styles, and documentation formats. In addition, the method is not dependent on any advanced features of existing development tools and does not force any significant changes to the software development process, so it is very easily integratable within existing development teams and projects. Developers and managers can continue to use the tools and techniques they are used to while still reaping the benefits of this method.

As to ensure CASCADE can survive the test of time, existing research in the field of Document Engineering has been surveyed and applied to derive the requirements of the CASCADE system. The specification of CASCADE has been built to the highest degree possible to be compatible with both current and future trends.

As a proof of concept, an implementation of the CASCADE system has been developed in Java™. The HTML documentation and the Java™ source code for the system itself are used as an example to demonstrate the capabilities of CASCADE. A change to the code and documentation of CASCADE is attempted both with and without the use of the synchronization method. The prototype system has also been found to enhance source code and documentation navigation considerably.

Chapter 2

THE IMPORTANCE OF SYNCHRONIZED DOCUMENTATION

Documentation of a software product takes many different forms and is important to everyone that develops and uses the software product for a variety of reasons. In this chapter, various common forms of documentation related to software products are explored to help further understand which documents are in need of synchronization and for what reasons. The importance of these documents to managers, software developers, software maintainers, and end users are then analyzed to further explore the reasons synchronization is necessary.

Common Document Types

The following is a brief sampling of documents that are maintained by most software development teams that develop software on a relatively large scale. For each document, a brief description is given as to what the document usually contains, followed by an analysis of whether the document needs to be synchronized and for what reasons.

Note that some development teams may have special requirements that are not addressed by the analysis that follows. This analysis is only intended to provide a general overview, and it makes reasonable assumptions that are fitting for many development teams but not necessarily all. For example, many of these documents would be more common in waterfall life-cycle models than in other life-cycle models. Figure 1 summarizes the typical relationship between the major documents in the waterfall software development lifecycle.

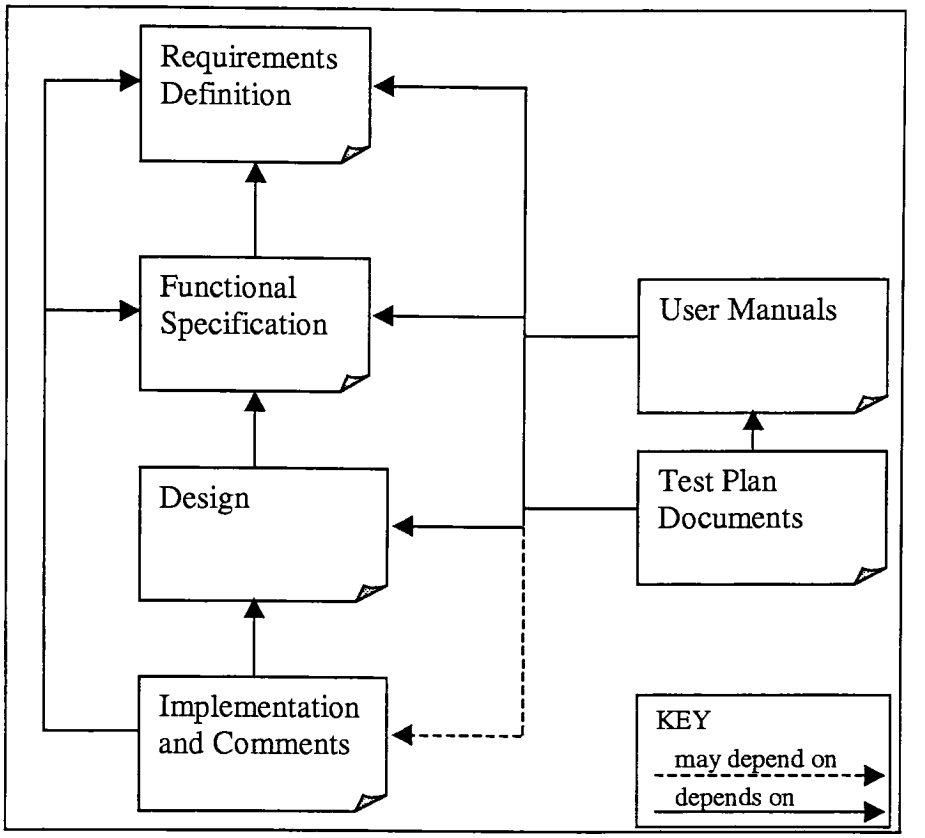


Figure 1. Typical relationship between major documents in the waterfall software development lifecycle.

Requirements Definition

The goal of the requirements definition document is to communicate an understanding as to what services the user expects a software product to provide, in a prose (non-structured) format that is easily understandable by all parties involved [7].

Almost all other documentation is derived in one way or another from this critical document. Therefore, its contents must be kept closely synchronized with the contents of other documents in the system.

This document is the primary basis for the Functional Specification and is important for the Test Plan documents and User Manuals, among other documents.

Requirements / Functional Specification

The Functional Specification is a well-structured document that goes into more detail than the Requirements Definition and uses language that is more appropriate for the technical staff [7].

This document has a special bond with the Requirements Definition, and should stay closely synchronized with it at all times. If inconsistencies arise between this document and the Requirements Definition, then the product the user thinks they are getting is not the product the developers are developing.

In addition, this document is the basis for Design documents, Test Plan documents, and User Manuals, among other documents.

Design

The Design document is an abstract system that satisfies the requirements in the Functional Specification. It forms the basis of many possible implementations [7].

The design is derived directly from the Functional Specification. Inconsistencies between the Design document and the Functional Specification can cause expectation gaps between user desires and actual software performance. This is because the software is designed to do something different from what the user wants it to do.

As mentioned, design documents form the basis for an implementation. Therefore, any implementation based off this design must stay in synchronization

with the design document. Inconsistencies between implementations and designs are often categorized as faults in either the design document or the implementation. Though logic would point towards the fault being in the implementation (since the design document came first), all too often the implementation overrides the design and the design document eventually becomes useless baggage. This destroys the beauty of a well-built design in that alternative implementations can no longer reliably be derived from it.

This document serves as the basis for the implementation and code comments, as well as the test plan, the user manuals, and various other documents.

Implementation and Code Comments

Implementation of a system is accomplished by building a working product from the design specification document. In the object-oriented paradigm, this includes writing and testing different classes in the system, and then integrating them [7].

As many have learned from aging COBOL programs with cryptic variable names, code comments are critical for human understanding of code. Indeed, even the original developer of a piece of source code will quickly forget what a variable named 'a' was originally intended for, or why a particular class or method exists without support comments to go with it. The importance of code documentation that describes "the big picture" was first taken seriously in the late 1970's and early 1980's after structured programming became popular. Donald Knuth took this concept to its extreme when he developed the WEB programming system, which treats source code as a work of literature that can also be compiled into an executable PASCAL program [5]. This system will be further examined in the chapter on Historical Solutions.

Keeping code comments synchronized with source code is a small challenge in itself, but one for which many developers do not have too much difficulty. This

may be because the code and the comments are adjacent to each other, so it does not take as much effort to keep them synchronized (the lazy programmer theory). However, such problems do occur when, for example, classes are not cohesive or rely heavily on implementation details of other classes. In cases such as these, a change to a class could make it necessary to update the code and comments for many other classes, even across project boundaries. If the code comments are not consistent with the source code, anyone working with the source code will quickly become confused. These same ideas can be applied to procedural programs, replacing "classes" with "modules."

Assuming code comments and source code are not in conflict, one still has to ensure that the code and the various documents within the system are synchronized. Most importantly, the code must be synchronized with the design. The reasons for this are explored in the previous section. In addition, often the implementation relies directly on aspects of the Functional Specification (such as user characteristics) or the original Requirements Definition (such as performance requirements) document elicited from the end-user. Inconsistencies between the Implementation and Functional Specification or Requirements Specification are faults, considering both of these documents are correct.

The implementation also serves as the basis for clear-box testing. In some cases, it may also serve as the basis for User Manuals, especially when the software product being developed is a library to be used by other software developers.

Test Plan

Test Plan documents provide a detailed outline of the testing that needs to be performed on a software product. It includes detailed test descriptions and results as well as a test schedule [7].

The Test Plan incorporates tests from every stage of the development process, and should be document-driven, meaning that tests should be derived based on existing documentation. Clearly, careful attention should be paid to ensure that once the source documents that originated the test cases change, the corresponding test cases must be changed as well. Without this synchronization between the Test Plan and other documents, testers could be wasting their time and efforts, performing incomplete or inappropriate tests.

A particularly interesting dependency is that of the Test Plan on the User Manual. The resulting user manual of a software product provides an excellent way to find tests for black-box and integration testing. Again, synchronization is critical here, as changes in the User Manual could, for example, cause certain tests to be come obsolete.

User Manuals

User Manuals are documents that are written with the end-user as the audience. In order to be successful, "manuals should not just describe the features of a system, they should help people get things done" [14]. This can be accomplished by providing task-oriented manuals, which are more difficult to write, but are much more useful to the readers.

In order to develop a successful user manual, it must incorporate aspects of the original Requirements Definition, the Functional Specification, the Design, and even sometimes the Source Code itself. For example, the Requirements Definition is where the user's initial requests came from so naturally users will be interested how to do what they initially requested. The Functional Specification, for example, outlines various specific requirements that may be of relevance to the usage of the system. The design may reference other systems that users must install or keep up to date. Especially for systems such as developer libraries, the

Source Code may have a lot to do with the user manual as well. Any loss of synchronization between the user manual and any of these documents may cause the user manual to be misleading, causing frustration and increased technical support calls among users.

Other Documents

There are various other documents that are important to keep synchronized. The following is a brief list of various other common documents:

- System Architecture
- Data Library
- Maintenance Manuals
- Configuration Management
- Revision History
- Post-Mortem Analysis
- Bug History
- Software Productivity Metrics
- Quality Assurance Plans
- Software Reviews and Audits
- Project Management Plans
- Software Lifecycle Processes

Each of these documents, if present in the development process, needs to be synchronized with other documents in some way or they will quickly become out of date. Once a document is out of date, its utility goes down and the cost to maintain it increases. In many cases, the document is simply abandoned because it causes more confusion than benefit.

Special Document Types

The majority of code and documentation that is used throughout the software development process has the following attributes:

- Typed into the computer by one or more human authors.
- Consists of text and possibly formatting.
- Can be modified on demand.

However, the forms of code and documentation for which these three attributes do not hold true must not be ignored. This section explores a number of document types that lack one or more of the above attributes. For each, its importance and its need to be kept up to date with other documentation are analyzed.

Diagrams

Diagrams are unique in that they consist of more than just text and formatting. Rather, they usually involve shapes, symbols, pictures, or other graphical elements that, when taken together, illustrate a concept. Diagrams supplement and enhance documents to provide a clearer understanding of some idea. They help to clarify, but only when they are kept up to date with other diagrams and documents. A diagram that is out of date can lead to confusion instead of clarity.

Paper Documentation

Though any electronically stored document can be printed on paper in some way, some documents can only be found on paper because their original source is lost. Retyping a document or using Optical Character Recognition are ways to "reverse-engineer" paper documentation back to an editable source. However, these methods are rarely practical and rarely yield the same results as having the original source in the first place. Nevertheless, often these documents are

important to the development process. In addition, paper documentation is popular because it is convenient, "familiar, flexible, portable, inexpensive, user modifiable [(in the on-paper sense)], and offers better readability properties than existing electronic displays" [1].

The fact that these documents are not somehow stored in electronic form does not mean they do not need to be synchronized. Dependencies may still exist between these documents and other documents in the system and therefore the same synchronization problems can still hold true for paper documents.

Read-Only Documentation

Similar to paper documentation, read-only documentation cannot be easily modified. The subtle difference is that read-only documents exist in electronic form. However, they are read-only either because of security restrictions or because the original source is not accessible and only an output file can be accessed. One of the most common forms of this phenomenon is a PostScript document [3]. Though it is possible to modify PostScript documents directly, usually this is not desirable as someone may generate a new PostScript document from the original source, overwriting any changes made to the file.

Again, just because these files cannot be modified directly doesn't mean that no dependencies exist with it, and therefore they still need to be kept synchronized.

Code Libraries

Code libraries are to source code what read-only documentation is to documents. Examples of code libraries are class files in Java, and object files or libraries in C++. These libraries are depended upon by parts of the system, and therefore, any changes to their interfaces are cause for potential synchronization problems. Note that for well-written designs that make use of encapsulation, there is no cause for concern if the library implementation changes.

Auto-Generated Code and Documentation

With the increase of popularity of tools such as Rational Rose® and Javadoc, an increasing percentage of documentation can now be automatically generated. This saves developers and authors much time, and allows them to concentrate on less tedious tasks. In addition, many modern IDEs know something about the architecture for which a developer is programming and can automatically create otherwise redundant code. These code segments are usually marked with a warning that the code has been automatically generated and may be regenerated at any time with or without warning.

The fact that this auto-generated code and documentation is regenerated frequently makes it difficult to make any permanent modifications to it. Therefore, essentially, auto-generated code and documentation can be treated the same as read-only code and read-only documentation. Note that accordingly, these documents need to be kept up to date at all times.

General Importance of Documentation

It would seem that a simple solution to the document synchronization problem would be to just eliminate the documentation. Clearly, while reducing the volume of documentation might be a reality in some instances (minimalism), eliminating it altogether is certainly not an option. There are a number of reasons that documentation is important to all parties involved in the development of a software product.

Importance to Managers

The growing complexity of software systems and the rate of employee turnover in software development teams makes complete, accurate, and up to date documentation a necessity for any manager [13]. When employees transfer to new teams or organizations, they leave behind only their code and

documentation. Therefore, without reliable documentation, the only remaining source of information is the source code. New developers must spend inordinate amounts of time reverse engineering source code into an abstract design [16].

In addition, according to Tilley and Müller, "maintenance routinely consumes from 50% to 80% of a products lifecycle and budget" [16]. Without proper documentation, a complex software project can be a maintenance nightmare, thus leading to these high maintenance costs.

Also, documentation has a significant effect on program understanding. "Software engineers and technical managers base many of their project-related decisions on their understanding of the architecture of the software systems they are responsible for" [15].

Importance to Software Developers and Maintainers

Computer programming is a field in which it is relatively easy to hack together impressive looking results without putting much thought into the design up front. This is not unlike architecture, for example, where without a solid blueprint one can put up a façade that looks attractive but is hollow on the inside. A software product that is hacked together in this fashion, if lucky, may survive an initial release, but will begin to break down very quickly when it is scaled or when new features are added.

To engineer a software system that is robust and maintainable requires up-front design effort, and in order to relay this design information to other current or future team members, this design must be well documented. Likewise, software developers require documentation of the interfaces of external systems they interface with in order to develop a software product.

In addition, as mentioned before, a complex software product can be a maintenance nightmare without the proper documentation. Software developers and maintainers can spend their energy more efficiently if they are not forced to continually reverse-engineer code to discover its original design.

Documentation can also serve as a memory refresher to developers. It is easy to forget why a particular design was chosen or why a particular requirement existed in the first place. Documents serve as a written record of concepts that were agreed upon and that have direct relevance to the software being developed.

Importance to End Users

User documents form a roadmap for users. According to Rettig, typical commercial user documentation usually includes:

- "A tutorial
- A task-oriented user guide
- An alphabetical reference to commands
- A pictorial guide to windows, icons, and tool palettes
- A reference card
- Assorted specialized guides ("getting started," "installation," or vertical market applications of a general tool)
- An on-line help system, usually sort of hypertextish" [14].

These documents provide a sense of confidence that the software they are using is well supported. Though users rarely read much of the documentation apart from reference cards and online help [14], it's the time they do need it and that it's there for them that they will appreciate it.

Chapter 3

SYNCHRONIZATION PROBLEMS

It has been established that documentation is a critical part of any software development effort. In addition, it has also been shown that practically every document in the software development lifecycle is closely related to at least one other document. Because of this, there arises a need to synchronize these documents so that when one changes, the others are kept up to date accordingly. This section starts with an analysis of synchronization, and then outlines the various problems typically encountered when trying to keep documentation synchronized, especially with code.

What is Synchronization?

In order to keep code and documentation synchronized, we must first understand what synchronization is. An informal definition of synchronization as it relates to this thesis is "the adjustment of two or more entities such that they are not in conflict with each other, and that they are all up to date."

Synchronization problems can be classified as follows:

> **intra-code synchronization** – Ensuring that all code and code comments within a single source file are in a consistent state.
>
> **intra-document synchronization** – Ensuring that a single document is in a consistent state.

inter-code synchronization – Ensuring that all code and code comments are in a consistent state across all classes in the system.

inter-document synchronization – Ensuring that all documentation, not including code, is in a consistent state across all documents in the system.

code-document synchronization – Ensuring that all relevant documents are synchronized with all relevant code.

Novick and Juillet provide a more formal interpretation, outlined by three maxims and one corollary. These maxims are quoted as follows [9]:

"**Coherence of meaning**: Integrity of semantic relationships.

M1. Semantic relations should not be changed unless the change is intended.

Coherence of reference: Integrity of consistency and differentiation in referring to domain entities, actions and relations.

M2 References to the same thing should appear the same.

M3 References to different things should appear different.

Corollary:

C1 Changes in referential expressions should be propagated."

Armed with a formal definition, we can attempt to build a system that assists developers and technical writers in keeping coherence of meaning and coherence of reference between all code and documentation.

What are the problems?

There are a number of barriers that keep synchronization from becoming a reality in most software development teams. These include developer reluctance, read-only or non-textual documents, and heterogeneity of formats, among others.

Developer Reluctance

Software documentation is routinely put off until the last moment or even, in some cases, postponed indefinitely. Many developers fail to see any reason to write a detailed description of an algorithm or a design if that algorithm or design is likely to keep changing in the near future. Therefore, they postpone documentation until the very last moment at which point pressure builds up to get a release out on time and documentation takes a low priority [16].

Another reason documentation is put off may be that many organizations enforce template comments that often do not give enough room for accurate descriptions, or require fields that are irrelevant. These template comments can be a nuisance and a disruption to the current train of thought and therefore are often left until the last minute [16].

In addition, if documentation is already sparse and incomplete, developers are less motivated to continue to keep documents up to date, and the downward trend in documentation quality builds on itself.

Special Document Types

Special document types were described earlier as consisting of diagrams, paper documentation, read-only documentation, code libraries, and automatically

generated documentation. Documents of these types are difficult to keep synchronized for a number of reasons.

First, some documents of these types are not typed into the computer by one or more human authors. Therefore, when it becomes necessary to change them, there is less control over how they can be changed. Even automatically generated documentation may have problems in that the generator may refuse to produce output until other conditions hold true (there may be an error in an unrelated portion of the system for example). As another example, a development team's license of a document generator may have run out, so they are stuck with what is essentially read-only documentation.

Second, some documents of these types are not of textual content. For example, diagrams or scanned in reference cards are simply vector or raster graphics, which makes it more difficult to easily see what needs to be changed and how to change it. In addition, it is difficult to detect changes made to these documents other than simply looking at the timestamp and/or checksum of the file. Therefore, it is difficult to determine what has changed to a fine granularity.

Finally, some documents of these types cannot be modified on demand. When the document needs to be modified, there is no straightforward way to do so. For example, a read-only document that is published by another company can be considered read-only to the development team because there is no way for them to modify it directly. Though they can suggest the modifications to the publisher, it will take time before these modifications will take effect. These documents pose a particular problem in that because they are so difficult to change, they are often not bothered with at all and left to become chronically out of date.

Heterogeneity of Formats

In 1991, Kevin M. Cunningham was the Project Manager for a system called OLH, which was designed to provide a centralized system that users of MIT's Athena system could use to access Athena-related documents. What Cunningham discovered was that there are many problems in coordinating heterogeneous documents in a distributed network [3]. These problems are amplified when one tries to coordinate the modification of these documents.

Documents that are in incompatible formats are difficult to display in a consistent manner, and are even more difficult to link together. It is a complex task to keep track of a thread of changes to be made to a number of documents when the developer must keep switching editor environments just to view the documents to be edited.

When documents are found scattered on different networks, these editing tasks become even more complex, as the developer spends an excessive amount of time searching for the documents rather than editing them.

Effects

These complexities, all taken together, only further discourage developers from taking a moment away from their relatively enjoyable programming time to keep documents up to date. The result is a series of documents that are critical to the software lifecycle process, but are helplessly out of date, increasing maintenance costs and decreasing the quality of the software produced.

Chapter 4

HISTORICAL SOLUTIONS

Various solutions to the synchronization problem have been attempted in the past. In this chapter, a number of historical solutions are analyzed for their strengths and shortcomings. The goal of CASCADE is to incorporate the advantages of each solution while eliminating their shortcomings. Of course many of these systems are designed for a specific purpose and are best suited for those specific situations. However, CASCADE is designed to serve as a more general solution to the synchronization problem that can be applied to a number of technologies and development environments.

Automatically Generated Documentation

Some documents such as those that are composed simply of hyperlinked class hierarchies and class inter-relationships are quite tedious to maintain and require little or no thought in preparing, especially if the content comes from hints embedded in source code. For documents such as these, a program can be written that generates the documentation directly, on demand, from the source code or some other source such as database definitions. Common examples of systems that do this are Rational Rose® and Javadoc™. Programs like Rational Rose® can also automatically generate and keep diagrams such as class diagrams up to date by reverse engineering code.

Another example of such a system (perhaps the first significant one) is Donald Knuth's WEB system [5]. In this system, a source file is treated primarily as a work of literature, and secondarily as source code. A WEB file consists of a

number of named and unnamed sections, each of which describe the functionality of part of a program. The first section describes a very high level view of the system and the reader can explore each portion of it in detail by following a numbered link. The author of the WEB file creates these sections and includes both a description of what the code does and the code itself. The author can then generate a TEX file containing a readable and printable version of the code, or a PAS file, containing PASCAL source code that gets sent to a compiler (see Figure 2).

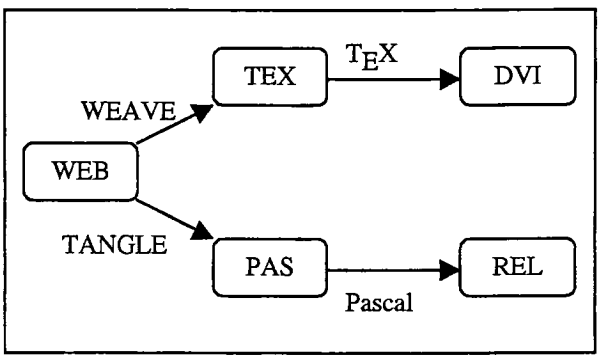


Figure 2. Dual usage of a WEB file [5].

The major advantage of the automatically generated documentation approach is that new, up to date documents can be produced on demand with little or no developer input. These documents would otherwise require constant maintenance by developers who would essentially be duplicating work (for each method added, it would have to be added to both the source code and the document). In addition, these documents are quite useful for developers and can improve their productivity by reducing the amount of time required to navigate code and other documents to find the information they need.

However, this is clearly by no means a complete solution to the synchronization problem. Not all documentation can be automatically generated (otherwise,

technical writers would be out of a job)! For example, there is no way to automatically generate a design from a requirements specification (yet?).

OO Concepts in Documentation

Sky Matthews and Carl Grove saw an increasing trend in the complexity of user documentation. This complexity was caused by an increase in the size of the document, the number of relationships among information elements, the types of information presented, and the range of audience skills and needs. In 1992, they proposed that the introduction of object oriented concepts to documentation is an abstract way to manage this complexity. "The essence of an object-oriented approach to documentation is that the structure of the documentation should mirror the structure of the original problem domain" [8].

Of particular interest is their application of the abstraction, encapsulation, and inheritance principles to documentation. These principles provide a straightforward way to design the structure of a document, making document authoring more manageable and less of an art. As an example of this technique, consider a help topic on the "File" menu of an application. Such a topic may answer the question "How do I open the File menu?" In a typical approach to documentation, each menu would have a section in the document, and each of those sections would answer the question "How do I open the *xxxx* menu?" If the procedure for opening a menu changes, the author would need to visit every section and change the corresponding instructions. Furthermore, a section on how to save a document might also answer this question.

In an object-oriented approach to documentation, the File Menu topic would be an object that inherits from the Menu topic object. The Menu topic object would know how to answer a question of the form "How do I open the *xxxx* menu?" and could produce the correct response. When the user is viewing the File Menu

topic, the answer to this question would automatically be available. A section on how to save a document could simply direct the question to the File Menu topic, which would direct the question to the Menu topic.

The advantages of this approach are many. The document becomes easier for the user (and author) to understand, it is easier to navigate, design philosophy is consistent across documents, and the task of designing these documents becomes more predictable and manageable (authors can work on different objects in the document independently) [8]. Intra-document and inter-document synchronization becomes much easier to manage, as a change to the base topic automatically propagates to all sub-topics. Additionally, because of encapsulation, there are fewer unorganized dependencies between documents, which means fewer dependencies to keep track of, thus improving coherence of reference.

Certainly, when possible, object-oriented concepts should be applied to document creation as this makes the synchronization problem much more manageable. However, the reality is that the majority of existing "legacy documents" are not structured in this manner, and synchronization must still be maintained between these legacy documents and other code and documents in the system. Changing the structure of existing documents is not a quick or easy task, and is more likely to cause more problems than solutions. Therefore, while object-oriented concepts are good rules of thumb for new documents, they are by no means a complete solution to the synchronization problem.

Rigi

Tilley, Müller and Orgun realized the value and importance of having up to date documentation, and noticed that many software development groups are lacking this. To remedy the problem, they proposed a system called Rigi that aids

software developers in reverse engineering existing source code in order to recover up to date documentation. While using Rigi, software developers reverse engineering the system can create views of the system for different parties involved [15]. For example, a manager would likely need a more high level view of a system than would a developer.

A reverse engineering system such as Rigi could prove to be an invaluable tool to a software development group that is trying to maintain existing software. Reverse Engineering can be a nightmare without the proper tools to keep the process organized. Once a system is documented with views, a manager has the information needed to make more informed decisions, and developers no longer have to spend as much time searching for the information they need.

The disadvantage of having a reverse engineering system available is that it reinforces a developer's bad habit of not keeping documentation up to date while writing code. Reverse engineering tool or not, it is still more cost effective to keep documentation up to date constantly rather than waiting until the maintenance process gets out of hand before reverse engineering the system again. Rather than requiring a pull on behalf of developers and managers, an ideal system would push changes immediately. An analogy can be made to the mechanistic approach to health vs. the preventative approach. In the mechanistic approach, when one gets sick, one goes to the doctor to get fixed and become well again. In the preventative approach, one's health is continuously maintained so that visits to the doctor are less frequent.

In addition, not all documents are maintained in synchronism. For example, though a design could be extracted, it may not be feasible to extract the original requirements document or test plans.

In summary, reverse engineering systems such as Rigi are excellent tools for repairing existing damage done to documentation, but they do not attack the cause of the problem itself.

INFO

To help solve the problem of how to keep documentation up to date, Tilley and Müller worked on another system called INFO. They realized that it is difficult, though not impossible, to keep multiple files synchronized. At the same time, they realized that storing all the documentation together with the code, as is done in WEB can cause navigation problems for the developer. Such coupling of documentation and code takes up valuable screen real estate making it more difficult for developers to find the information they need. Their INFO system provides a middle ground solution in which developers embed hypertext tags in the source code that link to external documents. Developers can easily see the hyperlinked information they need by pressing a programmable function key in an XEdit session. Developers can create new links just as easily [16].

The advantages of this system are that the program is self-documenting, valuable code real estate is preserved, required information is easily accessible, and annotations can be diagrams, multimedia or other enhanced content. Additionally, unlike Rigi, documentation can be constantly maintained rather than taking a break to reverse engineer the system every time it becomes seriously out of date.

INFO was designed to have no special hardware or software requirements rather than INFO itself. Ironically, it requires XEdit, which was a popular text editor for VM/CMS at the time, but is much less frequently used today. Therefore, one of XEdit's disadvantages is that it is incompatible with many of today's development environments. Another disadvantage of INFO is that it only keeps

code annotations up to date, and does not address keeping other documents such as requirement documents or test plans up to date. In addition, it does not address how to handle existing documentation that has not been entered using INFO in XEdit.

Chapter 5

THE CASCADE SOLUTION

We have explored the types of documents important to software development groups, the synchronization problems that exist between code and those documents, and the advantages and shortcomings of existing historical solutions to this problem. Based on these findings, a number of high level requirements for a system that solves the synchronization problem are presented.

These requirements are followed by a high-level design description of a system that satisfies these requirements called CASCADE (Computer Aided Synchronization of Code And DocumEntation). This system helps to solve the synchronization problem by providing developers with a tool that keeps track of changes and suggests where modifications are necessary. As with all solutions, CASCADE has its own advantages and shortcomings, which are discussed in the next chapter.

High Level Requirements

The following is a list of requirements that must be satisfied by systems that can claim to significantly help solve the synchronization problem discussed earlier in this thesis. These requirements are derived from an analysis of the problem and research into the advantages and shortcomings of existing solutions. The requirements assume the goal is to design a system that is general enough to meet the needs of the majority of software development groups that exist today.

The following is a list of these requirements. Following the list is a detailed description of each requirement:

- Facilitate Intra-Document and Intra-Code Synchronization
- Facilitate Inter-Document and Inter-Code Synchronization
- Facilitate Code-Document Synchronization
- Development Environment Support Requirements
    - Integrate With Programming Languages
    - Integrate With Mixed Programming Languages
    - Integrate With Document Types
    - Integrate With Programming Styles
    - Integrate With IDEs
    - Integrate With Lifecycle Processes
    - Integrate With Platforms
    - Integrate With Varying Development Team Size
- Support Real-Time vs. Interval Update Environment
- Support Read-Only Code and Documentation
- Reasonable Access Time
- Ease of Information Transfer
- Versatile Result Reporting
- Command-line Input

A detailed description of each requirement follows:

Facilitate Intra-Document and Intra-Code Synchronization

The solution shall facilitate the synchronization of information contained within a single document or within a single source file. That is, the solution shall be able

to analyze changes made to a single document or source file and recommend that changes be made elsewhere within the same document or source file. This is required because some synchronization problems happen within a single document or source file, and these problems need to be addressed.

Facilitate Inter-Document and Inter-Code Synchronization

The solution shall facilitate the synchronization of information between multiple documents or multiple source files. That is, if a document changes, the solution must be able to recommend changes to be made to other related documents. If a source file changes, the solution must be able to recommend changes to be made to other related source files. This is required because some synchronization problems occur between documents or source files and these problems need to be addressed.

Facilitate Code-Document Synchronization

The solution shall facilitate the synchronization of information between documents and source code files. That is, if a document changes, the solution must be able to recommend changes to be made to other related source code files, and vice versa. This is required because some synchronization problems occur between documents and source files, and these problems need to be addressed.

Development Environment Support Requirements

These requirements are all aimed at ensuring the solution does not depend on any particular features of a development group's choice of language, process or technologies.

Integrate With Programming Languages and Mixed Programming Languages

The solution shall be easily integratable with existing programming languages and projects that use a mixture of different programming languages. In particular, any

programming language that is structured, object-oriented, or component-based shall be supported. Other languages may be supported as well. The solution should not depend on any specific features of any particular language so that it can be used in a variety of projects.

Integrate With Document Types and Mixed Document Types

The solution shall be easily integratable with existing document types and projects that use a mixture of different document types. The solution should not depend on any specific features of any particular document type so that it can be used in projects with heterogeneous documentation formats.

Integrate With Programming Styles

The solution shall not require any changes to a developer's coding habits or programming styles, including, among others, indentation style, class headers and function headers. The option is left open to allow developers to embed hints (in the form of tags, for example) in documentation or code, but this shall not be the sole method of representing relationships between entities. This is to enable rapid deployment of the solution, and encourage its use without decreasing the productivity of developers.

Integrate With IDEs

The solution shall be easily integratable with existing IDEs. That is, it shall not depend on any specific features of any particular IDE to work properly. This is so that it can be used in a variety of development groups.

Integrate With Lifecycle Processes

The solution shall be easily integratable with existing lifecycle processes. That is, it shall not depend on any specific features of any particular lifecycle process. Examples of lifecycle processes include, but are not limited to, the Waterfall

lifecycle process, and the Spiral lifecycle process. This is so that the solution can be used in a variety of development groups.

Integrate With Platforms

The solution shall be easily integratable with existing platforms. That is, it shall not depend on any specific features of any particular platform. For example, it should not depend on the availability of symbolic links that are present on UNIX systems, but absent on Windows systems. The underlying compiled code should be the only platform-dependent feature. This is so that many development teams, each operating under different platforms, can all take advantage of the solution.

Integrate With Varying Development Team Size

The solution shall not make any assumptions as to the size of the development team working on a particular project. It shall seamlessly support development teams of one or more members. This enables the solution to be used in a wide variety of development environments.

Support Real-Time vs. Interval-Update Environment

The solution shall be able to be run in either a real-time environment or an interval-update environment, and produce the same results, though at different times. In this context, a real-time environment refers to an environment in which a change immediately invokes a response from the solution. An interval-update refers to an environment in which the solution is only updated every once in a while, and responses are buffered. This requirement is so that no changes need to be made to the development group's process.

Support Read-Only Code and Documentation

The solution shall gracefully support linking between read-only code and documentation. That is, code and documentation that is no longer, or never was,

writeable. Examples of this include source code modules that have no source (e.g. libraries), or documentation that is viewable, but not directly editable (e.g. postscript documents). The solution shall also be able to support automatically generated documentation such as that generated by products like Rational Rose®, so that these documents, too, can be kept up to date. The solution shall also provide support for linking to paper documentation, which is read-only by nature.

Reasonable Access Time

The solution shall be able to produce its suggestions within a reasonable amount of time. Reasonable is defined here as being quick enough so that the developer is not discouraged from using the system due to its slow execution. This is so that the solution can produce results without frustrating the user into not wanting to use the system.

Ease of Information Transfer

The solution shall store its information in such a way that it is straightforward to extract and import into another system, if this becomes desired or necessary. The solution shall also be able to easily import relationship information from other systems. Information in this context refers to information containing the relationship between document and code elements. This is to prevent the information from being locked up in a proprietary format in case it needs to be extracted.

Versatile Result Reporting

The solution shall be able to produce results in a flexible variety of formats. That is, the developer shall be able to gain access to the results produced by the solution in a straightforward manner by taking advantage of an API or some similar mechanism. At minimum, the system shall be able to directly or indirectly

execute a script with the solution output as a parameter to the script. This is to enhance the flexibility of the system.

Command-line Input

The solution shall be able to accept input and perform all basic functionality through a command-line interface. Additional interfaces such as a GUI are recommended but not required. The command-line requirement is to enhance the flexibility of the system and allow it to be invoked by existing IDEs.

High Level Design Description

This section provides a high-level view of the design of the CASCADE system. CASCADE is a system that fully satisfies the requirements described in the previous section, and therefore qualifies as a solution to the synchronization problem. The description relies heavily of the definitions provided in the glossary at the beginning of this thesis.

A sample implementation of this design is provided in the appendices of this thesis. The sample implementation includes a more detailed requirements definition, a functional specification, class hierarchy diagrams, a link to source code for a Java™ implementation, and a user manual.

Overview

CASCADE is broken into five separate modules, each of which has their own responsibilities. When combined together, these modules provide the developer with a coherent utility to manage code and document synchronization in their development team.

A software development team wishing to keep their code and documents synchronized can use the CASCADE system to establish relationships between significant entities. To do so, they must first setup the system with their

development environment. First, they invoke the **Project Manager** and create a new project, specifying which significant entities are important for the system to monitor. The Project Manager invokes the **Subentity Extractor**, which analyzes all significant entities and builds a tree of nodes that can be used to analyze relationships. Figure 3 illustrates the output of the Subentity Extractor. The user then uses the **Relationship Editor** to specify initial relationships between various present and future nodes in the system (these relationships can be added, deleted, or modified later). The user will integrate the **Modification Analyzer** into their favorite Development Environment, or build it in to their revision control system such that that every time a file changes, the Modification Analyzer is made aware of that change. Figure 4 illustrates the setup process. Note that new entities or relationships can be added or existing ones edited at any time.

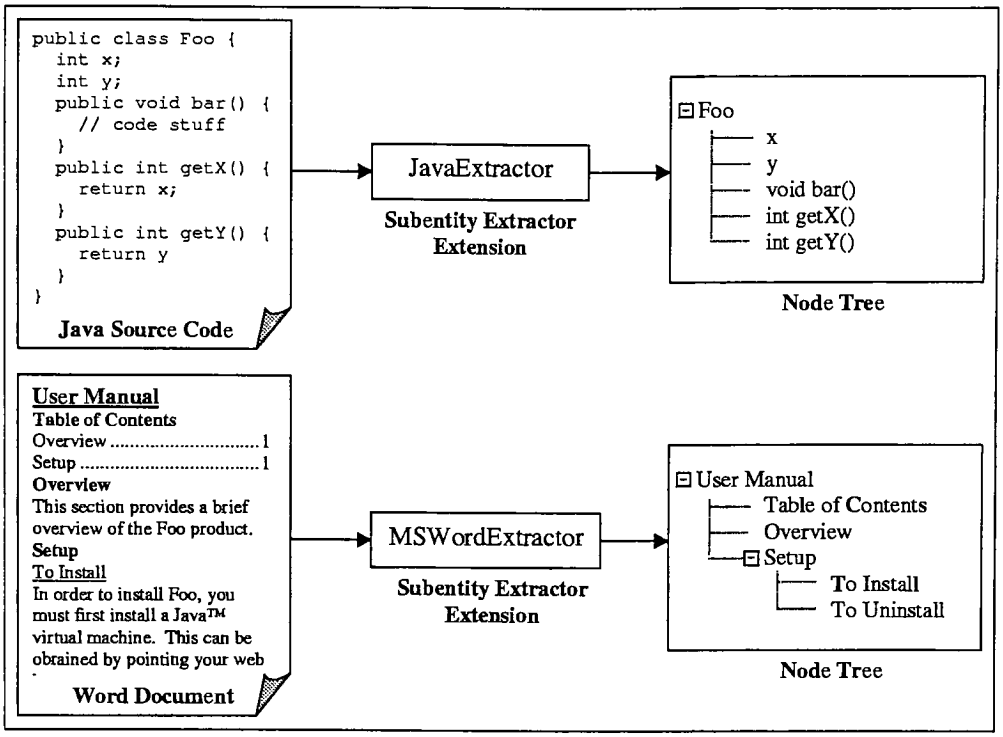


Figure 3. Input and Output of Subentity Extractor

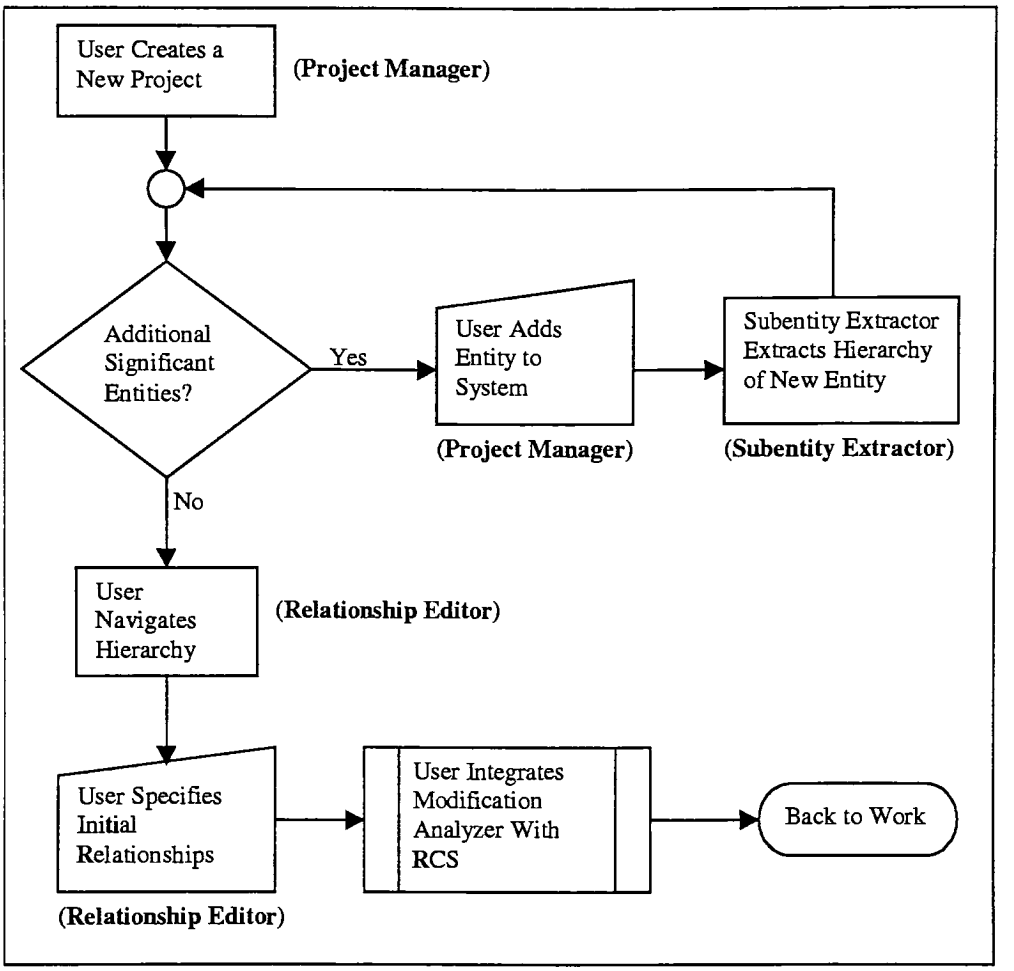


Figure 4. CASCADE Setup Process

At this point, the system is fully prepared to keep the code and documents synchronized. The developer can go about the normal software development lifecycle. Eventually, a change is made to one or more significant entities that affect other entities in the system (according to the relationships established in the Relationship Editor). At this point, the **Modification Analyzer** determines what was changed by launching the **Subentity Extractor,** and the **Relationship Analyzer** automatically determines what portions of the system are affected by that change and recommends actions to the user. Figure 5 illustrates this process.

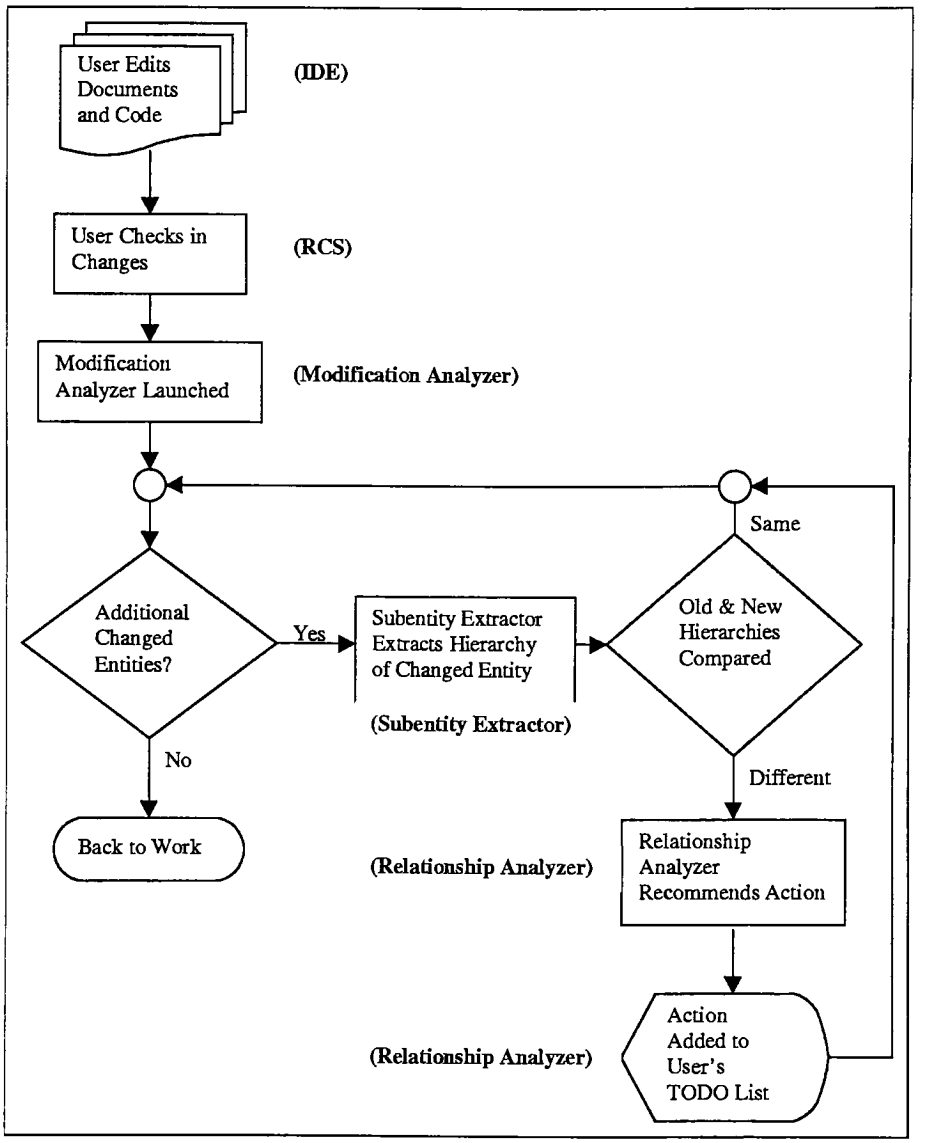


Figure 5. Normal Operation of CASCADE

Project Manager

The Project Manager is the primary user interface to the CASCADE system. Any action performed through the CASCADE system takes place within the context of a project. The Project Manager provides a way to view existing

projects and create new ones. It also provides the user access to the Subentity Extractor and the Relationship Editor. The Project Manager must be capable of handling simultaneous access by one or more members of a development team. Multiple users can view or read a project at the same time, but only one user can edit a single project at a time.

The responsibilities of the Project Manager include:

- Project Creation – The user can create new projects
- Significant Entity Registration – The user can register new significant entities.
- Subentity Extraction – The Subentity Extractor is automatically launched when a significant entity is added or its properties are changed.
- Access to Relationship Editor – User can launch the relationship editor.
- Significant Entity Type Registration – The user can register new significant entity types.
- Edit Project Options – The user can edit various project options such as the project search path and the log file path, among others.
- Project Integrity Detection – Upon startup, the Project Manager analyzes the integrity of the project and alerts the user as to any detected problems.

Subentity Extractor

The Subentity Extractor's main purpose is to increase the granularity of a significant entity by breaking it down into subentities. It does so by turning any significant entity into an XML document (using the CASCADE_node DTD found in the appendices) that represents its contents in a format that is readable

to CASCADE and other interested systems. This XML document also provides an easy way for the Modification Analyzer to compare two versions of the same significant entity and determine what, if anything, has changed. Note that this translation process makes for graceful handling of read-only code and documentation. Because it is difficult, if not impossible, to come up with a program that translates any file format in any language into XML, the Subentity Extractor relies on extensions that do this job. An API is provided so that new extensions can be developed, thus allowing a development team to have CASCADE analyze their existing code and documentation.

The responsibilities of the Subentity Extractor include:

- Aid in entity type recognition – Be able to determine the entity type by analyzing the file extension, header, and various other information.
- Node Tree Creation – The subentity extractor must be able to turn a generic significant entity into a hierarchical tree of nodes.
- Hint Extraction – Significant entities may have embedded hints (such as suggested relationships or search keywords) that should be extracted and relayed to the rest of the system.
- Significant Entity Type Recognition – The Subentity Extractor recognizes the significant entity types established by the Project Manager.

Relationship Editor

The Relationship Editor is the primary user interface for navigating the node tree of a project and establishing relationships between nodes. By decoupling the relationships from the actual files, CASCADE can record relationships between multiple entities without modifying the files themselves, thus providing graceful support for read-only code and documentation and eliminating code clutter. At

the same time, by allowing the Subentity Extractor to extract hints from code and establish relationships, users are free to embed hard-coded links within documents where they choose. The Relationship Editor will be able to gracefully handle access to a project by multiple users at a time.

The responsibilities of the Relationship Editor include:

- Navigation – The user will be able to navigate a project's node tree and view all available information about each node in the node tree. This includes the ability to visit hyperlinks in the user's default web browser. The following notes apply:
  - Launching – The navigation component can be launched, on demand, from the command line or from other modules.
  - Circular Dependencies – Can properly handle circular relationship dependencies properly.
  - Navigation only mode – The relationship editor can be launched in navigation only mode, in which case no editing of relationships is allowed.
  - Searching – The user can find a node by searching for keywords associated with various nodes in the system.
- Event Registration and Maintenance – Allows users to register new events for later detection, as well as modify and delete existing events.
- Relationship Construction and Maintenance – Allow users to create new relationships by associating an event with one or more actions. In addition, existing relationships can be modified or deleted.
- Command line support – Users will have the ability to perform relationship construction and maintenance from the command line.

Modification Analyzer

The purpose of the Modification Analyzer is to notice when a significant entity has changed, and provide information about what subentities have changed. The user launches the modification analyzer on a regular basis, either ad-hoc, or after every save from an editor, or at every check-in of a version-controlled file, or at regular timed intervals. It relies on the services of the Subentity Extractor to produce an XML document representing the contents of the significant entity. It stores the current version of each XML document along with the project, and compares the latest XML document with the stored version. The analyzer reports whether any differences are detected, and if there are differences, the analyzer determines where they are based on the structure of the XML document, and replaces the stored document with the latest XML document. The information on the nature of the changes is passed along to the Relationship Analyzer so that it can determine what these changes mean in the context of the relationships that have been established.

The Modification Analyzer must support arbitrary team sizes by being able to analyze changes made by different developers, and by being able to be run simultaneously by multiple users.

The responsibilities for the Modification Analyzer include:

- Current Version Archival – Must be able to keep the most recent version of all significant entities archived in XML format so that it has a previous version with which to compare.
- XML Comparison – Must be able to compare two XML documents in the CASCADE DTD, and create a list of nodes that have changed, along with causes for the change.

- Node Tree Structure Comparison – Must be able to detect changes in the structure of the node tree and create a list of nodes that have changed and the causes for the change.
- IDE Integration – Must be easily integratable with existing IDEs by providing the ability to run the analyzer and access all of its functionality from the command line.
- Change Log – Keep a log of changes noticed, recording when they were noticed as well.
- Invocation of Relationship Analyzer – When a change is detected, the Relationship Analyzer must be launched.

Relationship Analyzer

The purpose of the Relationship Analyzer is to determine which events are fired, given a list of nodes that have changed and the cause of the changes from the Modification Analyzer. After this is determined, a list of actions to be performed is generated and those actions are performed.

The responsibilities for the Relationship Analyzer include:

- Event Recognition – Given a significant entity, a list of subentities that have changed, and the nature of the changes, be able to determine which events are fired for each relationship established with this significant entity as the source.
- Action Execution – Given a list of events that have been fired, determine which actions should be executed, and execute them.
- Flexible Output – Be able to send output to a file, to standard output, or via e-mail to any member on the development team.

Phase-In

To support development teams that are already in synchronization trouble, the following process is recommended:

1. Initial Recovery – Install CASCADE, or other such system and establish links between significant entities in the system.
2. Gather New Requirements – Gather new requirements from users for next version of software. Update the requirements document.
3. Update Documentation – Follow CASCADE's recommendations for updating documentation to reflect these new changes.
4. Update Code – Follow CASCADE's recommendations for updating code to reflect these new changes.
5. Repeat steps 3 and 4 until CASCADE's checklist is clear.

As developers go along, they will realize inconsistencies in the documentation. Developers should be encouraged to fix the inconsistencies and fix any propagated inconsistencies. The documentation quality will begin to improve as a result of this.

Chapter 6

CASCADE ADVANTAGES AND DISADVANTAGES

Just as with any other solution to the synchronization problem, CASCADE has its own advantages and disadvantages. These advantages and disadvantages are described in this chapter.

Advantages

The CASCADE system has many advantages. It solves all of the synchronization problems stated earlier in this thesis, it satisfies all of the stated requirements, and it incorporates the advantages of the historical solutions studied while overcoming their shortcomings. This section describes the advantages of CASCADE in detail.

Solves all Stated Synchronization Problems

CASCADE solves all of the synchronization problems stated in earlier chapters.

Addresses Coherency Issues

CASCADE addresses the coherency issues brought up by Novick and Juillet [9]. Coherence of meaning is preserved in that no semantic relations are changed without developer intervention. So long as the developer takes care to realize what changes he or she is making, this will not be a problem. Coherence of reference is maintained by establishing relationships between entities. These relationships encapsulate the way two or more objects are related in an unvarying fashion. The corollary that changes in referential expressions should be propagated is handled in that when a change is suggested by CASCADE and then physically made to an entity, other relationships are automatically invoked, thus

propagating the change. Changes can continue to propagate after every successive execution of the Modification Analyzer.

Solves Developer Reluctance

Humans, especially developers, are unpredictable creatures by nature, so it is impossible to correctly claim that any system solves developer reluctance. However, CASCADE provides the tools and techniques to make keeping code and documentation synchronized as you go along a much easier and well-defined task. Tilley and Müller claim that "programmers tend to provide detailed design and implementation decisions more often if they have as much (or as little) room as they desire in which to do so, and are allowed to document these in a free-format fashion" [16]. CASCADE certainly provides this flexibility. An additional advantage is that CASCADE can be phased in at any speed, to turn around the declining trend of not keeping documentation up to date.

Handles all special document types

CASCADE handles all special document types in a well-defined manner. This is accomplished by creating a document "proxy" that links the system to the document. By decoupling the relationships from the actual files, CASCADE can record relationships between multiple entities without modifying the files themselves, thus providing graceful support for these special document types.

For diagrams, a document proxy can be created that describes all the elements in the diagram and describes what the diagram is trying to convey. When the diagram changes, CASCADE can notify the developer to change the document proxy. Relationships can be constructed with the document proxy instead of the diagram. When other parts of the system change that are related to the document proxy, the developer is notified to change the document proxy and therefore to change the diagram itself.

For paper documents, the document proxy will contain the same structure as the physical paper document (e.g. chapter and section headings), without all of the details filled in. For each section, a simple timestamp can be maintained. When a new copy of the paper document is received, the document proxy can be updated to reflect which sections of the document were changed by modifying the time stamps. This process can be automated with OCR and some scripts if this will improve productivity. The modifications to the proxy can trigger other changes in the system. Likewise, links can be made from other documents in the system to the document proxy. When the paper document needs to be updated, the developer is notified which section should be updated, and any standard process can be followed to suggest changes to the document.

Read-only documentation can have a document proxy that mirrors the read-only copy but is writeable. Instead of modifying the writeable copy, CASCADE can suggest changes to be made to the writeable copy and any standard process can be followed to update the read-only document. If the read-only document is simply unmodifiable (such as with a PostScript document), annotations can be maintained in the write-only version.

In the case of code libraries, a script can generate a document proxy that represents the structure of the classes, methods, etc. Again, links are made to the document proxy, and when changes are suggested, any standard process can be followed to contact the owner of the library to make the changes.

In the case of automatically generated documentation, the document proxy can be the automatically generated documentation itself. When changes are suggested, they are made to the source of the automatically generated document instead of the generated document itself.

Supports heterogeneity of formats

CASCADE supports heterogeneity of formats through extensions to the Subentity Extractor. When a new document format is used within the development environment, an extractor extension is developed that can understand that format and transform it into an XML document containing a tree of nodes.

Satisfies All Stated Requirements

CASCADE satisfies all of the requirements that were stated in the previous chapter.

Facilitates All Five Types of Synchronization

Intra-document, and intra-code synchronization is accomplished by establishing relationships between subentities within the same entity. For example, a relationship can be constructed from one method to another method just by linking their nodes. Inter-document and inter-code synchronization is accomplished by linking nodes from one document to another document, or from one source file to another source file. Code-document synchronization is accomplished by linking nodes from a significant entity representing a source file to a significant entity representing a document.

Development Environment Support Requirements

CASCADE is flexible enough to support virtually any development environment. It can integrate with different programming languages by providing Subentity Extractor extensions that can interpret that programming language. Mixed programming languages can be supported in the same way, as can mixed document types. CASCADE does not intrude upon the developer's programming style in that relationships can be made without cluttering code with tags and links if so desired, or alternatively hints can be embedded in code.

CASCADE can integrate with existing IDEs because it does not require any special IDE features. CASCADE integrates smoothly with lifecycle processes because it does not put any requirements on when development teams are to stop and synchronize the documentation with the code. This synchronization can be performed in either a real-time fashion or an interval-update fashion. Finally, CASCADE integrates smoothly with development teams of any size in that it supports concurrent users.

Reasonable Access Time

CASCADE is a simple solution, and as such does not require unreasonable time to perform its operations. The only limiting factor is in the speed of the Subentity Extractor extensions, so as long as they are optimized, CASCADE will be able to respond in a reasonable amount of time.

Ease of Information Transfer

CASCADE stores all of its information in XML documents for which the DTDs are given. It is quite simple to write programs that access and manipulate this data, and therefore it is easy to import and export relationship information from and to CASCADE.

Versatile Result Reporting

The Relationship Analyzer can report results in any number of ways. The sample implementation that accompanies this thesis, for example, can send results via e-mail to technical writers, developers or other members of the team, and can alternatively display messages to the console or execute shell scripts that perform whatever operations are required.

Command-line Input

Each of the five CASCADE modules can have all of their functionality accessed via command-line which makes it flexible to interface with.

Incorporates Historical Solutions Findings

CASCADE incorporates all of the advantages of the historical solutions while overcoming their shortcomings.

CASCADE allows the flexibility to take advantage of systems such as WEB or Javadoc™ by linking directly to the source (the WEB file in WEB or the Java™ source files in Javadoc™). Alternatively, the final output files (DVI or PAS files in WEB, or HTML or class files in Javadoc™) can be treated as special read-only document types. Output files can still be produced, maintaining synchronization between the code and documentation. In addition, CASCADE allows all other documentation to be synchronized as well, such as requirements documents and test plans, which WEB and Javadoc™ style systems do not address.

CASCADE leaves the flexibility to apply Object Oriented concepts to documentation, while still supporting legacy documentation that is not already authored in this formation.

CASCADE is compatible with reverse engineering systems, such as Rigi, while not relying on reverse engineering as the sole solution to the synchronization problem. This means that developers are encouraged to get into the good habit of keeping all documentation synchronized, now that they have the proper tools to do so.

CASCADE provides the flexibility to preserve code real-estate since links are stored external to files, while still providing the flexibility to embed relationship hints in source files and documents. At the same time, by allowing the Subentity Extractor to extract hints from code and establish relationships, users are free to embed hard-coded links within documents where they choose.

Disadvantages

Of course, any system is not without its disadvantages. The major disadvantage of CASCADE is that it is not completely automatic, and it still relies on developer motivation. CASCADE is not a magical solution and still relies on developer motivation. Though developers are encouraged to fix documentation inconsistencies as they go along, it is still up to them to do so diligently. Therefore, it is still possible for inconsistencies to leak through. It is hoped, however, that with proper attention, CASCADE will, on average, increase documentation quality rather than letting it decrease by not instating any system of documentation maintenance at all.

Another disadvantage of CASCADE is that the number of links can become overwhelming if not managed properly. It is suggested that relationships are established sparingly, where synchronization is of utmost importance. Otherwise, branching factors get out of hand, and developers may begin to ignore CASCADE's suggestions.

Chapter 7

EXAMPLE USAGE

To determine whether CASCADE works in practice, a sample implementation has been created, using Java™ as an implementation language. This implementation, along with its associated documents, can be found in the appendices of this thesis. The code and documentation for the CASCADE system itself is used in this chapter to demonstrate the capabilities of CASCADE. To do this demonstration, a feature is added to the source code, and the results of this change are analyzed with and without the use of CASCADE.

Overview

During the development of CASCADE, HTML was chosen as a standard documentation language because of its easy integration with web browsers and straightforward hyperlinking abilities. In addition, many existing software development groups use HTML documentation because it makes it easy to publish documents online. Java™ was chosen as the implementation language because of its overall simplicity and because of its seamless integration with XML, expression parsers, its advanced GUI creation abilities, and its facilitation of Javadoc™ as well as many other reasons.

The implementation is a complete minimal implementation of CASCADE (with a few non-essential features missing), though it is not in a production-ready state. There are still a number of desired features to be implemented such as flag nodes, and more advanced link management. These features can hopefully be implemented by future students seeking a master's project. The current

implementation can be found in Appendices A, B, and C of this thesis. A discussion of what is and is not implemented can be found in Appendix C. Appendix D presents a User's Guide to the current implementation.

To facilitate synchronization between code and documentation, links have been embedded in the source code during the development process. These links take the form of Javadoc™ tags which are passed to a Javadoc Doclet which generates an XML document that is parsed by the Java Extractor, an extension to the Subentity Extractor. These tags specify a bi-directional link between the tagged method, class or field name and the corresponding section of documentation (or code). Taking the time to add these links to the Javadoc™ code upon creation of the stubs proved to be beneficial, even before the implementation of CASCADE was complete, in that it pointed out various requirements that had to be taken into consideration upon implementation of those stubs.

The inter-document relationships were constructed manually, after the implementation was completed. This was done to explore how reverse engineering is facilitated by CASCADE. In all, it took approximately 2 hours to get the documentation satisfactorily linked, most likely a worthwhile investment. The time could have been improved had the current implementation of CASCADE provided more automated relationship construction features.

Hypothesis

In a typical development environment, document and code quality will continue to degrade over time and inconsistencies will arise between the two. The hypothesis is that without CASCADE, documentation and code consistency and quality will be degraded slightly after this change is made, whereas with using CASCADE, the documentation and code consistency and quality will actually slightly improve after the change is made.

Experiment

To analyze how well CASCADE works, a new feature was added the current system, and an analysis was performed both with and without the use of CASCADE.

Before the experiment took place, flag nodes (see the glossary definition in Appendix A) were not supported. During the experiment, flag node support was implemented using a text file to store a list of flags that are currently set, and creating a FlagsExtractor extension to the subentity extractor that can parse this text file.

In the first part of the experiment, the modifications were made without using CASCADE. In the second part of the experiment, the modifications performed in the first part were used as a base, and CASCADE was told to analyze what had changed. The advice it gave was then applied to improve the quality of the documentation. CASCADE was then told to analyze these new changes and so on until the documentation and code seemed to be up to date.

Results

The results of the experiment, both without the use of CASCADE and with the use of CASCADE are summarized here.

Without CASCADE

Without the use of CASCADE, a class called FlagsExtractor was added to the system. The source code for this class can be found in Appendix E. This class took a total of 43 minutes to design, implement, and test. It adds the ability to create flag nodes by specifying a file called `flags.txt` containing a listing of flag names, each of which is to be considered as set. The extractor creates a tree of subentities based on this text file.

Next, the documentation was updated (without the aid of CASCADE). It was difficult to determine exactly where to start in updating the documentation. This difficulty made the idea of updating the documentation at all unattractive, as it was easier to pick a more well-defined item off a todo list and implement that instead.

The easiest way to get started was to update the Javadoc documentation. This was accomplished by simply running a batch file that kicked off Javadoc. To rebuild the entire Javadoc HTML documentation took a total of 2 minutes.

To determine which document to update next, a list of documents that would possibly need changes were made. This list was compiled by simply looking in the various project directories and jotting down the name of any document found that seemed like it may need updating. The following list resulted, after 2 minutes of analysis:

- Functional Specification
- Requirements Specification
- Source Code Comments

Each one of these documents was analyzed individually for the possible necessity of changes. The Functional Specification was analyzed first. Each section was scanned briefly, and the only section that appeared to need modification was the Standard Extensions subsection of the Subentity Extractor section. This was modified by adding the FlagsExtension as a standard extension, along with a brief description. Overall, it took 8 minutes to scan this document and make the required changes (approximately 4 minutes searching and 4 minutes in modification).

The next document to be analyzed was the Requirements Specification. After three minutes of analysis, the only relevant section seemed to be requirement HB.19., which directly addresses flag nodes. This requirement did not seem to be in need of updating, and was left alone. Therefore, the Requirements Specification was not updated.

The final document to analyze was the source code comments themselves. Of the three, this was the most massive amount of documentation to analyze. CASCADE has almost 12,000 lines of code and about 4,000 lines of comments (measured 12 KLOC 8 NCKLOC) and is spread over 50 source files. This makes it far too tedious to manually page through all source code after every change, making it difficult to determine which files to look at and which ones to skip. Javadoc came in useful in determining which files to look at, because it presented an overview of all classes in an organized format, along with all documentation.

Even after browsing through Javadoc documentation for three minutes, only one class stood out enough to justify attention. This class was called FlagNode, and turns out to be a remnant from the original design phase in which Flag Nodes were going to be treated as a special type of node. This outdated class was examined carefully for any loss of data, then deleted, and the system was recompiled, taking a total of three minutes.

At this point, all changes were deemed complete, the FlagExtractor had successfully been implemented and all necessary documentation modified. Note that more care than usual was taken to update documentation in this first experiment. Realistically, even the blatantly obvious changes would rarely be made because developers are reluctant to modify documentation, especially if

they know it will eventually get out of date anyway. Figure 6 summarizes the amount of time each phase of the experiment took.

Task	Time
Implement FlagsExtractor	43 minutes
Build Javadoc documentation	2 minutes
Determine document list	2 minutes
Search Functional Specification	4 minutes
Update Functional Specification	4 minutes
Search Requirements Specification	3 minutes
Browse Source Code	3 minutes
Delete FlagNode class and recompile.	3 minutes
Total:	**64 minutes.**

Figure 6. Task times for system modification without the use of CASCADE.

As is evident by Figure 6, 43 minutes were spent in actual development, and 21 minutes were spent in code and document maintenance. In other words, about 33% of the total development time for this fix was spent in document and code maintenance. The result was the updating of Javadoc documentation, the addition of a small section to the Functional Specification, and the removal of a single source file. This being a relatively simple and small change, it is clear how larger changes could easily require a much larger time investment. The quality of the update is questionable, as it was impractical to spend too much more time looking over code and documents to determine what else needed to be updated.

Though it was easy to miss something, all reasonable attempts were made at fixing any obvious document and code inconsistencies due to this change.

With CASCADE

Because hindsight would have interfered with a timing study, instead of duplicating the work already done, all of the changes mentioned in the previous section (without CASCADE) were used as a base for this experiment. CASCADE was used to analyze the changes just made to the system as to discover what documents were left not updated.

First, CASCADE's modification analyzer was run and the results were recorded. The modification analyzer and the relationship analyzer took a little over one minute to run and produced the summarized output found in Figure 7.

```
{Flags}
{Requirements Specification}
{Functional Specification}
  FlagsExtractor added.
{Source Code}
  FlagNode() deleted.
  void FlagNode(edu.rit.cs.cascade.common.Node,
    java.lang.String) deleted.
  String getContents() deleted.
  void setContents(java.lang.String) deleted.
  FlagNode deleted.
  FlagsExtractor() added.
  void extract() added.
  void processFile(java.io.File) added.
  boolean isCompatible(java.io.File) added.
  FlagsExtractor added.
{Master Index}
{Users Guide}
```

Figure 7. Output of Modification Analyzer (emphasis added).

The Modification Analyzer correctly detected all changes made to the significant entities in the system in the previous experiment. The log file (see Figure 8)

reflected a much more detailed (though more difficult to read) analysis. Note that programs can be written that analyze the log file and invoke their own events when certain significant entities change.

Subentity	Status
{./Functional Specification/Functional_Specification.html/Functional Specification/Subentity Extractor/Standard Extensions/FlagsExtractor}	added
{./Functional Specification/Functional_Specification.html/Functional Specification/Subentity Extractor/Standard Extensions}	subentityAdded indirectSubentityAdded
{./Functional Specification/Functional_Specification.html/Functional Specification/Subentity Extractor}	indirectSubentityAdded
{./Functional Specification/Functional_Specification.html/Functional Specification}	indirectSubentityAdded
{./Functional Specification/Functional_Specification.html}	indirectSubentityAdded
{./Functional Specification}	indirectSubentityAdded
{./Source Code/edu/rit/cs/cascade/common/FlagNode/FlagNode()}	deleted
{./Source Code/edu/rit/cs/cascade/common/FlagNode/void FlagNode(edu.rit.cs.cascade.common.Node, java.lang.String)}	deleted
{./Source Code/edu/rit/cs/cascade/common/FlagNode/String getContents()}	deleted
{./Source Code/edu/rit/cs/cascade/common/FlagNode/void setContents(java.lang.String)}	deleted
{./Source Code/edu/rit/cs/cascade/common/FlagNode}	deleted subentityDeleted indirectSubentityDeleted
{./Source Code/edu/rit/cs/cascade/common}	subentityDeleted indirectSubentityDeleted
{./Source Code/edu/rit/cs/cascade/extractor/extension/FlagsExtractor/FlagsExtractor()}	added
{./Source Code/edu/rit/cs/cascade/extractor/extension/FlagsExtractor/void extract()}	added
{./Source Code/edu/rit/cs/cascade/extractor/extension/FlagsExtractor/void processFile(java.io.File)}	added
{./Source Code/edu/rit/cs/cascade/extractor/extension/FlagsExtractor/boolean isCompatible(java.io.File)}	added
{./Source Code/edu/rit/cs/cascade/extractor/extension/FlagsExtractor}	added subentityAdded indirectSubentityAdded
{./Source Code/edu/rit/cs/cascade/extractor/extension}	subentityAdded indirectSubentityAdded
{./Source Code/edu/rit/cs/cascade/extractor}	indirectSubentityAdded
{./Source Code/edu/rit/cs/cascade}	indirectSubentityAdded indirectSubentityDeleted
{./Source Code/edu/rit/cs}	indirectSubentityAdded indirectSubentityDeleted
{./Source Code/edu/rit}	indirectSubentityAdded indirectSubentityDeleted
{./Source Code/edu}	indirectSubentityAdded indirectSubentityDeleted
{./Source Code}	indirectSubentityAdded

```
                                                    indirectSubentityDeleted
```

Figure 8. CASCADE log file after making original changes (timestamp removed and formatting added to enhance readability).

Also note the accuracy to which the modification analyzer has detected changes. This is a side benefit in that CASCADE keeps an accurate log of what parts of the system have changed on what days. This can be helpful for quickly determining exactly what has changed between two different releases, for example, which is difficult to do otherwise. Whereas most systems that do provide for this functionality only describe the differences between two source code builds, CASCADE provides the differences in documentation and any other entities as well.

After the modification analyzer detected these changes, it automatically launched the relationship analyzer, which determined which relationships are relevant given these changes. It then generated ten email messages with suggestions as to what to look at next. The suggested changes can be summarized as follows (this summary is also output to the display):

1. extension package → Standard Extensions in Functional Specification
2. Standard Extensions in Functional Specification → extension package
3. Functional Specification → Last Updated
4. Functional Specification → Revision History
5. Class FlagNode → Requirement HB.19.
6. Class FlagNode → Glossary definition of flag node in Requirements
7. Class FlagNode → CASCADE DTD in Functional Specification
8. Class FlagNode → Significant Entity DTD in Functional Specification

9. Subentity Extractor in Functional Specification → Class Extension

10. Subentity Extractor in Functional Specification → Class Extractor

Suggestions 1 and 2 indicate the relationship between the Standard Extensions section in the Functional Specification, and the extensions package. It indicates that if anything is added, deleted or modified from either, they both need to be kept up to date. The need for this change was discovered in the first experiment manually by searching through the documentation. Had CASCADE been used instead, the four minutes spent finding this particular change would have come free. No additional changes are necessary now.

Though somewhat minor, suggestions 3 and 4 were completely missed in the first experiment. In these two suggestions, CASCADE is indicating that since the functional specification changed, so should the Revision History and the Last Updated sections of that document. These changes were promptly made in an elapsed time of about one minute. Note that most of the times referred to in this section are rounded up to the nearest minute.

Suggestions 5 and 6 exist because the FlagNode class has been deleted. They indicate that in the requirements document, requirement HB.19. and the glossary definition of "flag node" should be checked for inconsistencies. These two sections were analyzed for about one minute as well. Requirement HB.19. is relevant in that it specifies that flag nodes are a requirement in the system. Naturally, CASCADE was worried that the class originally meant to handle these flag nodes was deleted. Though the new implementation of flag nodes does not require a change to HB.19. or the glossary definition, it served as a reminder to create relationships from this new class to that requirement and glossary entry since they are all related. These relationships were established by creating two

javadoc tags in the FlagsExtractor class. These tags were added to the FlagsExtractor class in one additional minute:

```
@cascade {requirement}/HB/HB.19
@cascade {glossary}/flag node
```

In addition, while looking at the glossary definition for "flag node," it was noticed that it does not mention anything about subflags. This edit also took approximately one minute.

Suggestions 7 and 8 also exist because the FlagNode class was deleted. They indicate that in the Functional Specification document, the CASCADE DTD and the Significant Entity DTD sections should be checked. The checks were performed in under a minute. These links turned out to be historical in that the DTD used to contain specific tags for flag nodes, but they no longer do.

Because the FlagNode class has been deleted, all relationships related to that class had to be destroyed manually (future versions of CASCADE will do this automatically). This was done by editing the CASCADE.xml file directly (the relationship editor only currently supports removing relationships one at a time), and took an additional minute.

Finally, suggestions 9 and 10 indicated that since the Subentity Extractor section of the Functional Specification was changed, the Extension and Extractor classes should be checked. The Extension class was recommended because it is the base class of all subentity extractor extensions. The Extractor class was recommended because it controls all of the extension classes. In an additional minute, these classes were checked for any necessary changes and none were found.

Now that all ten suggestions were addressed, the files were checked in and the analyzer was run again to check for cascading effects. Again, the analyzer took 1:20 to produce its results. The output after the second run is illustrated in Figure 9.

```
{Flags}
{Requirements Specification}
  flag node modified.
{Functional Specification}
  Revision History modified.
  Last Updated modified.
{Source Code}
  FlagsExtractor modified.
{Master Index}
{Users Guide}
```

Figure 9. Output of Second Run of Modification Analyzer (emphasis added).

Again, the relationship analyzer was launched, and this time, five email messages were generated, as follows:

1. Requirements Specification → Last Updated
2. Requirements Specification → Revision History
3. Class FlagsExtractor → Glossary for flag node in Requirements
4. Glossary for flag node in Requirements → Class FlagsExtractor
5. Class FlagsExtractor → Requirement HB.19.

Recommendations 1 and 2 were due to careless mistakes of not updating the Last Updated and Revision History sections after updating the definition for flag node. These were taken care of in one minute.

Recommendations 3, 4 and 5 are due to the new relationships that associate a change in the glossary definition of flag node or requirement HB.19. with the FlagsExtractor class (this used to be associated with the FlagNode class that was

deleted). These recommendations can be ignored because the change made to the definition was only a clarification and the change to the class was only to add the links to the relationships.

After running the modification analyzer one last time, CASCADE had no more suggestions, so the process was complete. Figure 10 summarizes the time it took to make all changes.

Task	Time
Experiment 1 Base	64 minutes
First Run of Modification Analyzer	1:20 minutes
Suggestions 3, 4	1 minute
Suggestions 5, 6	3 minutes
Suggestions 7, 8	2 minutes
Relationship Maintenance	1 minute
Second Run of Modification Analyzer	1:20 minutes
Suggestions 1, 2	1 minute
Third Run of Modification Analyzer	1:20 minutes
Total:	**76 minutes.**

Figure 10. Task times for modifications to system with the use of CASCADE.

The new changes made can be summarized as follows:

- The Revision History and Last Updated sections of the Functional Specification were updated.
- The FlagsExtractor was updated to include links to requirement HB.19. and the glossary definition of flag node.
- The glossary definition of flag node was updated to mention subflags.
- All relationships dealing with the deleted FlagNode class were deleted.
- The Revision History and Last Updated sections of the Requirements Specification were updated.

All of these changes were enacted in an additional 12 minutes. This upgrades the percentage of development time spent on document and code maintenance from 33% to 44%. At the same time, the number of updates made to the documentation and code increased considerably. However, looking back at Figure 6 it becomes clear that using CASCADE completely eliminates the need for many of the activities performed in the first experiment.

First, CASCADE eliminates the need to determine which documents to search through (saving 2 minutes in this case) because it does this work and reports specific entities to visit via email. CASCADE also eliminates the need to search through the Functional Specification (saving 4 minutes) for the same reason. In fact, the very first suggestion it gave provided the exact section of functional specification to visit. The 3 minutes spent searching through the Requirements Specification and the 3 minutes spent browsing through source code, looking for things to change, can be saved as well, for an additional 6 minutes. Overall, therefore, approximately 12 minutes could have been saved if CASCADE were used from the start.

This would have reduced the total time from 76 minutes to 64 minutes. Therefore, in the same amount of time it took to make the changes in the first experiment without the use of CASCADE, all of the changes in the second experiment could have been made as well, if CASCADE had been used from the beginning.

Analysis

Of course, this is an isolated example, and a more formal study must be conducted before CASCADE can truly claim to improve productivity on any scale, but the point is well made. CASCADE eliminates a good portion of the tedium of searching through documentation to determine what needs to be changed. Instead, this time can be better spent maintaining relationships between entities and having CASCADE help guide the developer to keep the code and documentation up to date.

Indeed, as expected, the quality of the documentation after the first experiment (without using CASCADE) was questionable. This is because it relied on developer motivation, intuition, and tedious navigation through documentation to come up with the changes needed after even as simple a change as this. Clearly, a more involved modification to the source code, or a change in a major requirement would have complicated matters greatly. In addition, it is rare to find a developer that has the time or motivation to spend 33% of his or her time following random processes to keep the code and documentation up to date.

Though it is difficult to measure the quality of documentation, even with the very relevant changes made to the documentation in the first experiment, arguably the overall quality of the documentation degraded somewhat by the end of the experiment. Whereas before the change the "Revision History" and "Last Updated" portions of the Functional Specification were up to date, for example,.

after the experiment they were not. In addition, whereas before the experiment the FlagNode class was fully linked to its defining requirement and glossary definition, after the experiment, the class was left without any links to such documentation.

With the use of CASCADE, the process of keeping the documentation up to date was far more well defined, and allowed the developer to concentrate on how to change the entities, rather than on finding which ones to change. All of the changes made in the first experiment were automatically suggested by CASCADE, and new changes were picked up as well. The process was as simple as running the Modification Analyzer, following CASCADE's suggestions, and then restarting the process until everything is up to date. This simple, well-defined process encourages developers to keep documentation and code up to date, and motivates them to do so by recommending what to do next rather than expecting them to spontaneously take action on their own. By feeding these suggestions directly to the appropriate developer via email, the developer has no choice but to read it and take care of it, before deleting it for filing it away.

Again, documentation quality is difficult to measure, but after the second experiment (with the use of CASCADE), the quality of the documentation seems to be in better condition than it was at first. The Revision History and Last Updated sections were kept up to date, even after the developer forgot to update them on two separate occasions. Furthermore, the glossary definition of flag node was actually improved by updating it to mention subflags, and the FlagsExtractor class was linked to its requirement and glossary definition. Even the relationships themselves were kept up to date.

Additional Findings

After these experiments were concluded, a brief analysis of where the developer would go next in the implementation of flag nodes was performed. Without the use of CASCADE, the developer would have to scan through the documentation and determine what the next problem to tackle would be. However, with the use of CASCADE's built-in cross-entity searching ability, a simple 5 minute search on the term "flag node" turned up nine separate documentation and code results. These results served as a roadmap for what still needed to be done to complete the implementation of flag nodes. Another facility that came in very useful is the ability to graph the relationships between many nodes. For example, viewing a graph for requirement HB.19. reveals all nodes that the "flag node" requirement is related to in some way. See Appendix D for screen shots of these various features.

It was found that tagging source files with relationship information does not take much time or energy, and is extremely useful in tracing code segments back to original requirements and specifications, thus improving the quality of the code and documentation. For CASCADE, which is a 12 KLOC program, only 121 lines (approximately 1%) of the source code was taken up by lines with @cascade tags.

A negative finding included the fact that having too many links can be a new source of confusion. Some classes and document sections are heavily linked, and any small change can quickly multiply into a hundred emails sent by CASCADE to the developer's account. In addition, these relationship links must be continuously maintained.

Chapter 8

CONCLUSION AND FURTHER RESEARCH

It has been established that documentation is of critical importance to managers, software developers and maintainers, end users, and all other parties involved with a software product. Documents found to be of particular importance include Requirements Definitions, Functional Specifications, Design documents, User Manuals, Test Plan documents, and code comments.

Each of these critical documents is closely related to at least one other document in the system, therefore implying that a change to any document requires close inspection of related documents to ensure synchronism. To fully explore the issue of synchronization of code and documentation, a formal definition of synchronization was established, placing synchronization problems into five categories, namely intra-code, intra-document, inter-code, inter-document, and code-document synchronization. In addition, three maxims and a corollary have been borrowed from Novick and Juillet, dealing with coherence of meaning, coherence of reference and propagation of changes. Problems related to maintaining synchronization were found to include developer reluctance, special document types, and heterogeneity of formats, among others. These problems result in documents that are critical to the development process, but that are helplessly out of date.

Existing systems that attempt to tackle part of the synchronization problem have been analyzed. These solutions include automatically generated documentation, the application of object-oriented concepts to documentation, a reverse-

engineering tool called Rigi, and a hyperlinking system called INFO. None of these systems presents a comprehensive solution, though each system successfully attacks part of the problem. Each system has been analyzed for its advantages and shortcomings.

Based on the formal problem definition and the advantages and shortcomings of existing systems, a set of requirements has been carefully drawn out for a system that can claim to provide a comprehensive solution. A high level design for CASCADE, a system that satisfies these requirements has been presented, and then analyzed for its own advantages and shortcomings. The system successfully addresses all issues raised in this thesis.

Finally, in order to demonstrate its effectiveness, an implementation of CASCADE has been achieved, using the Java™ programming language and various related technologies. The implementation converts all code and documents into XML documents of a uniform format that could be easily be compared and linked to. The code and documentation of CASCADE itself has been used to demonstrate its own effectiveness. During this demonstration, a number of additional findings took place. These findings included the fact that too many links can cause confusion, and that an unexpected benefit of the system was a considerable improvement of navigation. In particular, the ability to easily search all documentation at once, and the ability to view relationship graphs came in handy.

Future Research

The results of this thesis are encouraging, and there are a number of future research topics that can be spawned from these results. The following list includes suggestions for future Master's projects or research studies.

- Improve implementation to support a multiple user environment. This can probably be best accomplished by creating a two or three-tiered system consisting of a CASCADE server that accepts requests from clients. This can also have performance benefits. The current implementation does not correctly handle multiple users accessing the system at the same time.
- It may be possible to have CASCADE automatically suggest intra-document, intra-code, inter-document, inter-code, or code-document links within a project. These suggestions can come from natural language parsing and extraction of keywords.
- The issue of link management needs to be visited. Currently, the number of links in a project can grow out of hand and the only facilitation to resolve this is by deleting or modifying them one at a time. Advanced link management needs to be integrated with CASCADE for it to be a successful product. This can be partially accomplished by implementing relationship reflection and having relationships update themselves when entities in the system are added or deleted.
- Design principles for unambiguous and concise documentation would complement this work nicely. For example, the issues of object-oriented documentation and the minimalist approach to documentation were touched on briefly, but not fully explored. It would be interesting to develop a set of design patterns for documentation, just as Gamma et al. have done for code [4].
- Security can be introduced to CASCADE as well. Currently, all developers have access to all code and documentation in the relationship editor. It would be useful to be able to set access restrictions on certain information while still having the system function properly.

- Currently, CASCADE is software-centric. However, based on the two National Transportation and Safety Board dockets presented in the introduction, it is clear that a system like this would have applications in other industries as well. How can CASCADE be generalized to be more useful to other industries?

Contact

Mark Roth, the author of this thesis, encourages any constructive feedback on this thesis, including but not limited to comments and suggestions, discussion, questions, other future research areas, and desire to supplement this work. To contact Mark Roth, please send email to markroth@bigfoot.com. Your mail will be forwarded to the correct address.

Appendix A

REQUIREMENTS DEFINITION

This document is reformatted to fit this thesis and it is available in its original online format on the accompanying CD. The online version is fully hyperlinked to glossary entries. Note that this requirements definition was developed before the high-level requirements list that appears in this thesis in earlier chapters, and therefore may be missing some requirements. Unfortunately, CASCADE was not used to keep this Thesis up to date with the documents in the appendix.

Description

This document enumerates the requirements for the CASCADE system. Each requirement is in a requirements group, and is uniquely numbered. This unique number can be used to refer to a requirement in other documentation or code. As such, the number of a requirement should never change. The last portion of this document is a glossary of key terms. This glossary is crucial to the understanding of these requirements.

Formatting

The format for a requirement is as follows:

<req-number> [<VOID>] <date> <requirement-name> <CR>
<requirement-description>

Where:

<req-number> is the requirement number, composed of the section number followed by a period, followed by a unique number, followed by another period.

[<VOID>] is optional, and specifies that this requirement is no longer valid.

<date> is the date that the requirement was instantiated, in the format M/D/CCYY

<requirement-name> is a single, short (up to 60 characters) phrase used to identify the requirement.

<CR> is a carriage return. In HTML, either a <HR> or a
.

<requirement-description> is a description of the requirement. It should answer the questions what and why, but not how.

High Level Requirements

HA. Development Environment Requirements

HA.1. 3/5/1999 Programming Language Integration

CASCADE shall be easily integratable with existing programming languages. In particular, any programming language that is either structured, object-oriented, or component-based shall be supported. Other languages may be supported as well. CASCADE should not depend on any specific features of any particular language. This is so that CASCADE can be used in a wide variety of projects.

HA.2. 3/5/1999 Programming Style Integration

CASCADE shall not require any changes to developer's coding habits or programming styles, including, among others, indentation style, class headers and function headers. CASCADE may allow developers to embed additional "hints," but shall not require any changes to be made to code that is already in place. This is to enable rapid deployment of CASCADE, and encourage its use without decreasing the productivity of developers.

HA.3. 3/5/1999 Mixed Programming Language Integration

CASCADE shall be usable within projects that use mixed programming languages. This will increase the scope of projects that can use this system.

HA.4. 3/5/1999 IDE Integration

CASCADE shall be easily integratable with existing IDEs. That is, it shall not depend on any specific features of any particular IDE to work properly. This is so that it can be used in a variety of development groups.

HA.5. 3/5/1999 Lifecycle Process Integration

CASCADE shall be easily integratable with existing lifecycle processes. That is, it shall not depend on any specific features of any particular lifecycle process. Examples of lifecycle processes include, but are not limited to, the Waterfall lifecycle process, and the Spiral lifecycle process.

HA.6. 3/5/1999 Platform Integration

CASCADE shall be easily integratable with existing platforms. That is, it shall not depend on any specific features of any particular platform. For example, it should not depend on the availability of symbolic links that are present on UNIX systems, but absent on Windows systems. The underlying compiled code may be platform-dependent. This is so that many development teams, each operating under different platforms, can all take advantage of the CASCADE system.

HA.7. 3/5/1999 Development Team Size

CASCADE shall not make any assumptions as to the size of the development team working on a particular project. It shall seamlessly support development teams of one or more members. This enables CASCADE to be used in a wide variety of development environments.

HA.8. 3/6/1999 Real-Time vs. Interval-Update Environment

CASCADE shall be able to be run in either a real-time environment or a interval-update environment, and produce exactly the same results, though at different times. In this context, a real-time environment refers to an environment in which a change immediately invokes a response from CASCADE, whereas interval-update refers to an environment in which CASCADE is only updated every once in a while, and responses are buffered.

HA.9. 3/28/1999 Read-Only Code and Documentation

CASCADE shall gracefully support linking between read-only code and documentation. That is, code and documentation that is no longer, or never was, writeable. Examples of this include source code modules that have no source (e.g. libraries), or documentation that is viewable, but not directly editable (e.g. postscript documents). CASCADE shall also be able to support auto-generated documentation such as that generated by products like Rational Rose, so that these documents, too, can be kept up to date. CASCADE shall also provide support for linking to paper documentation, which is read-only by nature.

HB. Functional Requirements

HB.1. 3/5/1999 Project Creation

CASCADE shall allow a user to create a project. A project will maintain information about significant entities, and inter-relationships between

subentities. All operations will be performed in the context of a particular project. This requirement is necessary because CASCADE will behave differently depending on which significant entities are selected for analysis. CASCADEs behavior also changes when the specified relationships between subentities changes.

HB.2. 3/5/1999 Significant Entity Registration

CASCADE shall enable users to identify those entities that are significant. The list of significant entities will be maintained from session to session as to eliminate the need for a user to specify these relationships each time CASCADE is started. Each significant entity will have a description stored along with it.

HB.3. 3/5/1999 Entity Type Recognition

CASCADE shall be able to recognize various types of significant entities and treat them according to their entity type. Entity types are required because different significant entities are composed of different types of information. For example, a source file may contain functions or class declarations, whereas a requirements document may contain requirements.

HB.4. 3/5/1999 Subentity Extraction

CASCADE shall be able to analyze a significant entity and extract the important subentities that lie within. This extraction will be dependent upon the entity type. Since subentities can lie within other subentities, CASCADE must be able to construct a hierarchy of subentities that lie within a significant entity. Subentity extraction is required so that users have a way of referring to a particular portion of a significant entity. Without it, granularity would be extremely course and it would be difficult to maintain synchronization between portions of significant entities.

HB.5. 3/5/1999 Node Recognition

CASCADE shall be able to recognize and find a node given its formal name. See the glossary definition of node. This is required to allow for a naming scheme that addresses either projects, significant entities, or subentities.

HB.6. 3/28/1999 Event Registration and Recognition

CASCADE shall allow the user to specify significant events that are caused by changes made to entities within the system. The system shall be able to recognize when these events occur or have occurred. See the glossary definition of event.

HB.7. 3/28/1999 Action Registration and Execution

CASCADE shall allow the user to specify actions that can be performed when events are fired. The system shall be able to execute these actions, and use the information provided to hypothesize what events will be fired on other nodes as a result of executing this action. This allows a detailed list of the effects of a change to be built. See the glossary definition of action.

HB.8. 3/28/1999 Relationship Construction and Maintenance

CASCADE shall allow a user to create and store relationships between nodes. Relationships are represented by associating events with actions. See the glossary definition of relationship. CASCADE shall be able to maintain relationships between nodes by recognizing when events should be fired and executing the resulting actions when appropriate. These relationships are at the core of the CASCADE concept. They indicate that when one portion of the system is modified in some way that some other portion of the system is affected in another way.

HB.9. 3/28/1999 Change Log

CASCADE shall have the ability to keep a log of changes it has observed after each run, on a project basis. Each log entry will store the date and time of the run, the nature of the change (change to the list of events, actions, or relationships, or changes to specific entities), and a description of the change. This allows users to keep track of when various changes to the system have occurred, which provides historical information to aid in tracking defects.

HB.10. 3/28/1999 Searching

CASCADE shall provide the ability to search through descriptions of significant entities and subentities within the system, for keywords. This allows users to locate entities and subentities without knowing their exact name.

HB.11. 3/28/1999 Navigation

CASCADE shall provide a user the ability to navigate between related documents and code. For example, a user can find a node and then get a listing of all nodes that are related to that node in some way. This improves documentation and code navigation considerably by making it more structured than typical search engine results.

HB.12. 3/28/1999 Linking

CASCADE shall allow the user to link nodes to other resources that are related, but not through a specified formal relationship. For example, a user may wish to relate a requirement to a web site, just for informational purposes. These links can be specified as a standard URL. This will improve navigation and understandability of documentation and code.

HB.13. 4/3/1999 Relationship Modification and Deletion

CASCADE shall allow the user to modify and delete relationships that have previously been specified. This makes relationships non-permanent, to allow the user to correct those relationships that are currently incorrect.

HB.14. 4/3/1999 Event Modification and Deletion

CASCADE shall allow the user to modify and delete events that have previously been specified. This makes events non-permanent, to allow the user to correct those events that are currently incorrect. Upon modification, CASCADE will alert the user as to what relationships are dependent on the event. Upon deletion, CASCADE will delete all relationships that are dependent on the event, with the user's permission. An event cannot be deleted without first deleting all relationships dependent upon the event. CASCADE can automatically delete these relationships for the user.

HB.15. 4/3/1999 Action Modification and Deletion

CASCADE shall allow the user to modify and delete actions that have previously been specified. This makes actions non-permanent, to allow the user to correct those actions that are currently incorrect. Upon modification, CASCADE will alert the user as to what relationships invoke this action. Upon deletion, CASCADE will modify all relationships that make use of this action to no longer invoke this action, upon the user's permission (if the user disagrees, the action will not be deleted). Any other actions registered with the relationship will remain consistent.

HB.16. 4/3/1999 Significant Entity Type Registration

CASCADE shall allow the user to register new significant entity types with the system. The user must supply a method for CASCADE to be able to extract subentities from any significant entity with the new significant entity type.

HB.17. 4/3/1999 Project Search Path

CASCADE shall allow the user to specify a project search path, which will contain a list of directories (or repositories) to look for other projects within. This is so that a user can work with projects that are rooted in different parts of the system.

HB.18. 4/3/1999 Wildcard Formal Names

CASCADE shall support a limited wildcard capability when it comes to formal names. An asterisk (*) may be used in place of either a project name, a significant entity name, or a subentity name, within a formal name in certain contexts. For example, "CASCADE/Requirements Document/*", "*/Requirements Document/Glossary" or "*/*/Glossary" (note that "*/Glossary" is not the same as "*/*/Glossary" since the asterisk only replaces a single portion of a formal name). These wildcards are only valid in the context of defining the source of an event or nodes affected by an action. When defining the source of an event, a wildcard indicates that all nodes that match the given wildcard should be considered when deciding whether the event has occurred, and if at least one of them satisfies the event cause, the event is considered as having occurred. When defining the nodes affected by an action, the wildcard indicates that all nodes that currently exist that match the given wildcard will be affected by the action. Unless otherwise specified by another requirement, wildcards are not valid in any other context, especially in defining the name of a node. The allowance of wildcards makes it much easier to specify events in a much more compact form, helping to reduce the number of event declarations.

HB.19. 4/3/1999 Flag Nodes

CASCADE shall allow the creation of flag nodes. CASCADE shall recognize a special significant entity called "Flags," under which all flags for a project are created. If a flag is set, it will exist as a subentity of the Flags significant entity. If the flag is not set, no node will appear with the formal name of the flag. Flags can have subentities, which are subflags for a particular flag. If a subflag is set, the flag itself is automatically set. These subflags are flags in themselves, so they, too, can have subflags. These flags allow for simpler definition of events, and reuse of certain complex event source criteria.

HC. User Interface Requirements

HC.1. 3/28/1999 User Access to Functional Requirements

CASCADE shall provide a user interface that is robust enough to provide users access to all the functionality of the system that is specified in the HB (Functional Requirements) section of this document.

HD. Error Handling Requirements

HD.1. 3/28/1999 Corrupt Projects

CASCADE is required to recognize corrupt projects, and warn the user of them, but is not required to recover from them. The recognition is required so that users are not misled into believing they are working from a valid project when in reality they may not be.

HD.2. 3/28/1999 Missing Entities

If CASCADE encounters a missing entity that it requires for operation, it shall inform the user, and assume the entity has maintained its previous state (i.e. that it has not been added, modified, deleted, and that its existence has not changed). Subentities within that entity are assumed to remain the same as well. This is so that the remainder of the system can be used until the problem is resolved.

HD.3. 3/28/1999 Corrupt Entities

If CASCADE encounters a corrupt entity, it is treated as a missing entity (see requirement HD.2). A corrupt entity is one that cannot be analyzed for content, or one that subentities cannot be extracted from. This is so that the remainder of the system can be used until the problem is resolved.

HD.4. 4/3/1999 Circular Dependency Detection

If CASCADE encounters a circular dependency when resolving the effects of actions on nodes, it should be able to detect these dependencies and only visit the node once. For example, if a modification of A causes B to need modification, and if a modification of B causes A to need modification, CASCADE will trace no further since A is already assumed to have been modified. The same behavior should be observed if a circular dependency is detected when determining which relationships to delete when an event is deleted. This enables the creation and handling of relationships that imply circular dependencies, which often occur in real life.

HE. Advanced Requirements

These requirements are not necessary for a minimal implementation of CASCADE, but are very handy features that make CASCADE more usable.

HE.1. 3/28/1999 Auto-Suggested Links

An advanced implementation of CASCADE may support natural language parsing capabilities to suggest relationships between documents, or auto-link them. For example, it may notice a particular keyword is used frequently in a certain section of a document and in the comments of a certain section of code and suggest that the user create a relationship between the two. This improves integration of CASCADE into existing projects that would otherwise be difficult to create relationships from. It may also catch some relationships that users miss.

HE.2. 4/3/1999 Event Detection and Action API

An advanced implementation of CASCADE may support an API that allows developers to detect events that do not fall within the standard set of events, as outlined in the glossary. The API would also allow for more advanced actions to be performed when events occur, that can not be performed using the existing set of actions as outlined in the glossary.

HF. Relationship Reflection Requirements

HF.1. 4/3/1999 Project as Significant Entity

CASCADE shall consider the project itself as a significant entity in the system. Other projects, too, shall be visible as significant entities, with the entity type of "CASCADE Project." The formal name of a project significant entity will simply be the project's formal name. This will allow relationships to be established across multiple projects.

HF.2. 4/3/1999 "Relationships" as a subentity

CASCADE shall recognize a special node called "Relationships" as a subentity of any project. This allows relationships to be categorized under this subentity instead of directly under the project itself.

HF.3. 4/3/1999 Relationships as a subentity of "Relationships"

CASCADE shall consider all currently registered relationships as subentities of the "Relationships" node, as specified in requirement HF.2. The formal name for a relationship node will, therefore, be "<project-name>/Relationships/<relationship-name>" where <project-name> is the

formal name of the project, and <relationship-name> is the name of the relationship. This is done so that relationships can be sources of events, and therefore actions can be performed when relationships are added, removed, or modified.

HF.4. 4/3/1999 "Events" as a subentity

CASCADE shall recognize a special node called "Events" as a subentity of any project. This allows events to be categorized under this subentity instead of directly under the project itself.

HF.5. 4/3/1999 Events as a subentity of "Events"

CASCADE shall consider all currently registered events as subentities of the "Events" node, as specified in requirement HF.4. The formal name for an event node will, therefore, be "<project-name>/Events/<event-name>" where <project-name> is the formal name of the project, and <event-name> is the name of the event. This is done so that the existence of events in the system can be a source of events, and therefore, actions can be performed when events are added, removed, or modified.

HF.6. 4/3/1999 "Actions" as a subentity

CASCADE shall recognize a special node called "Actions" as a subentity of any project. This allows actions to be categorized under this subentity instead of directly under the project itself.

HF.7. 4/3/1999 Actions as a subentity of "Actions"

CASCADE shall consider all currently registered actions as subentities of the "Actions" node, as specified in requirement HF.6. The formal name for an action node will, therefore, be "<project-name>/Actions/<event-name>" where <event-name> is the formal name of the project, and <action-name> is the name of the action. This is done so that existence of actions in the system can be a source of events, and therefore, actions can be performed when actions are added, removed, or modified.

Glossary

action A change that should be made to the system, usually as a result of an event in the context of a relationship. At minimum, a CASCADE implementation must allow the user to specify a name for the action, a description, and the effects of the action. The name must be short (up to 60 characters) and unique to the project. The effects of the action are to affect nodes in the system (which may eventually invoke events), and to perform

some physical action. Note that though it can be specified in which way nodes in the system should be affected, the tree is not permanently changed until the actions are performed. This is due to the fact that, in reality, the exact changes to a system cannot be fully predetermined. The nodes affected can be specified by the name of the nodes affected, along with the names of the events to invoke for each node (so that complete list of estimated changes can be developed). The physical action allows some of the actions to be automated. The minimal list of physical actions that any CASCADE implementation must support includes:

compound action Chains a number of actions together

execute system command - Executes a command in the native OS.

display message - Displays a message to the user or adds a message to the user's todo list.

Email message – Emails a message to a user or adds a message to the user's todo list.

set flag - Sets the given flag in this project if it does not already exist.

unset flag - Unsets the given flag in this project if it is currently set.

Theoretically, an API could be provided (see requirement HE.2) to the user to be able to execute powerful actions based on events. This is not required for a minimal CASCADE implementation.

CASCADE - Computer Aided Synchronization of Code And DocumEntation. Also refers to any system that implements CASCADE. See Mark Roth's Masters Thesis for more information.

entity - A significant entity, subentity, or a project. Because of reflection, a relationship is also considered an entity.

entity type - A category for a significant entity. Some sample entity types are "requirements document" and "source code module."

event - A change that has been made to one or more significant entities in the system. At minimum, a CASCADE implementation must allow the user to specify a name for the event, a description, a source, and a cause of the event. The name must be short (up to 60 characters) and unique to the project. The source is a formal name of an entity within the system. The cause can be any boolean combination of the following, using AND, OR, and

NOT:

added - Indicates that the given node did not previously exist, and now exists.

deleted Indicates that the given node used to exist in the system, but now no longer exists.

exists - Indicates that the given node may or may not have existed before, but does currently exist.

modified - Indicates that changes have been made to the content of the entity represented by the given node.

immediate subentity added - A subentity has been added directly below this node.

immediate subentity deleted - A subentity has been deleted from directly below this node.

immediate subentity modified - The contents of a subentity directly below this node has been modified.

indirect subentity added - A subentity has been added one or more levels beneath this node.

indirect subentity deleted - A subentity has been deleted one or more levels beneath this node.

indirect subentity modified - The contents of a subentity one or more levels beneath this node has been modified.

Theoretically, an API could be provided that allows the user to use a more powerful language to specify when an event should be fired. This is not required for minimal CASCADE implementation.

flag node - A user-defined subentity whose presence indicates some condition is true. These subentities exist to make event declarations cleaner and more efficient, and can be set directly from actions. The formal name for a flag node is "<project-name>/Flags/<flag-name>" where <project-name> is the name of the project this flag is in, and <flag-name> is the name of the flag. The name of the flag must be short (up to 60 characters) and unique to the system.

formal name - A full name that uniquely describes a node.

node - A single element that can be referred to by a formal name. This includes projects, significant entities, and subentities.

project - Maintains information about significant entities and inter-relationships between subentities. The same significant entities and subentities can be referenced in multiple projects. A project's formal name is simply the name of the project, which may not be the same name as the file the project is stored in. A project's name may contain spaces. An example of a project formal name is "CASCADE." The formal name "." is reserved to mean the current project, and may be used in place of the project name anywhere a formal name appears.

relationship - A representation of how two or more nodes are related. A relationship consists of a name, a description, an event, and one or more actions to be performed when that event occurs. The name of the relationship should be short (up to 60 characters), and unique to the project.

relationship reflection - Refers to the fact that a relationship is an entity within the system, so that relationships can be constructed in which the source of the event is another relationship's existence.

significant entity - A single document, or a single piece of source code. Usually separate files containing these entities, but possibly as portions of a repository containing these entities. A single significant entity can be found in more than one project. The name of a significant entity may be different than the name of the file or repository entry that contains it. The name of a significant entity may contain spaces. Therefore, a document or piece of source code can be referred to using different names in different projects, or even within the same project. The formal name of a significant entity is "<project-name>/<significant-entity-name>" where <project-name> is the formal name of the project this significant entity is defined in, and <significant-entity-name> is the name of the significant entity. An example of a significant entity formal name is "CASCADE/Requirements Document"

subentity - Refers to a sub-portion of a significant entity. For example, if a source file were a significant entity, a subentity of that source file could be a function or a class declaration. Subentities can lie within other subentities, to form a hierarchy of subentities. The name of a subentity may contain spaces. All subentities lie within one and only one significant entity. A subentity is formally named using the format "<significant-entity-formal-name>/<subentity0-name>[/<subentity1-name>[/<subentity2-name>[...]]]" where <significant-entity-formal-name> is the formal name of the significant entity this subentity is defined in, and subentity*n*-name is the name of the

node at the *n*th level of the hierarchy tree of subentities. At least one subentity name must be represented, and the subentity might not refer to a leaf node. An example of a subentity formal name is "CASCADE/Requirements Document/Glossary/subentity" Subentities have descriptions associated with them as well. These descriptions are often extracted from code comments or documentation snippets.

subflag - A flag that is considered under the category of another flag. A subflag exists as a subentity under an existing flag. If a subflag is set, its parent, by definition, must also be set.

Appendix B

FUNCTIONAL SPECIFICATION

This functional specification document is reformatted to fit this thesis and it is available in its original online format on the accompanying CD. The online version is fully hyperlinked to the requirements definition and the glossary entries.

Description

This document describes the functional specification for the CASCADE system. It is derived from the Requirements Specification document and uses vocabulary from its glossary. This document should be used in conjunction with the Requirements Specification, as many details are left out so as to not duplicate information. in particular, links to specific requirements and glossary definitions should be paid close attention.

Formatting

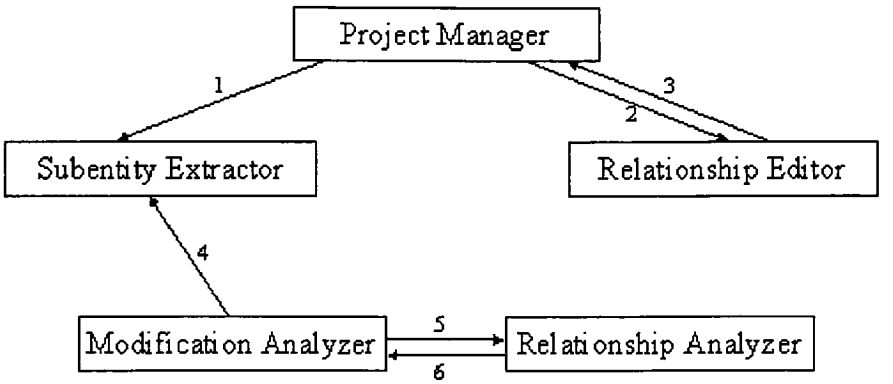
This document is divided into a number of sections, each describing a portion of the system. It starts with an Overview section, which describes how all of the portions of the system are used together. Each section that describes a portion of the system has, minimally, a description section that gives an overview of the component, a dependencies section that summarizes its relation to other components, and a responsibilities section that enumerates its responsibilities.

Overview

A software development group wishing to keep their code and documents synchronized can use the CASCADE system to establish relationships between significant entities. To do so, they must first invoke the Project Manager and create a new project, specifying which significant entities are important for the system to monitor. The Project Manager invokes the Subentity Extractor which analyzes all significant entities and builds a tree of nodes that can be used to analyze relationships. The user then uses the Relationship Editor to specify relationships between various present and future nodes in the system. At this point, the system is fully prepared to keep

the code and documents synchronized. The user will integrate the Modification Analyzer into their favorite Development Environment, or build it in to their revision control system such that that every time a file changes, the Modification Analyzer is made aware of that change. When a change is made to a significant entity that affects other entities in the system (according to the relationships established in the Relationship Editor), the Relationship Analyzer determines what portions of the system are affected by that change and recommends an action to the user.

The following diagram illustrates the nature of the relationships between the five modules:



The project manager requires the services of the subentity extractor when the user adds another significant entity to the project. This is necessary so that a detailed node tree can be built for the new significant entity.

When the user wishes to navigate the node tree or edit the relationships between significant entities, the project manager launches the relationship editor.

The relationship editor relies on the project manager to detect corrupt projects.

The modification analyzer relies on the services of the subentity extractor to produce a new XML document that it can compare to the original XML document.

When the modification analyzer detects a change in a significant entity, it launches the relationship analyzer to determine what to do next.

In advanced implementations of the relationship analyzer, an action may encompass modifying a significant entity. If this is the case, the relationship analyzer launches the modification analyzer.

The bulk of the user access to requirements (see HC.1.) is provided through the Project Manager and the Relationship Editor.

Project Manager

Description

The Project Manager is the primary user interface to the CASCADE system (see HC.1.). Any action performed through the CASCADE system takes place within the context of a project. The Project Manager provides a way to view existing projects and create new ones. It also provides the user access to the Subentity Extractor and the Relationship Editor. The Project Manager must be capable of handling simultaneous access by one or more members of a development team. Multiple users can view or read a project at the same time, but only one user can edit a single project at a time (see HA.7.).

Dependencies

The Project Manager provides services to the Relationship Editor. It relies on the services of the Subentity Extractor and the Relationship Editor.

Responsibilities

The Project Manager is responsible for the following:

Project Creation (see HB.1) -- The user can create new projects.

Significant Entity Registration (see HB.2) -- The user can register new significant entities. The Project Manager will make an educated guess (aided by the Subentity Extractor) as to the type of entity, but the user must be allowed to manually specify the entity type (see HB.3.).

Subentity Extraction (through the Subentity Extractor) -- A significant entity within a project can be searched for subentities by launching the Subentity Extractor.

Navigation (see HB.11.) (through the Relationship Editor) -- The user can navigate an existing project by browsing through its nodes and reading descriptions of entities.

Relationship Establishment (through the Relationship Editor) -- The user can launch the Relationship Editor and build relationships while navigating the node tree.

Action Registration and Maintenance (see HB.7.) -- The user can register new customized actions with the system. The user can also modify or delete existing actions (see HB.15. for details as to how this is to be handled).

Significant Entity Type Registration (see HB.16.) -- The user can register new significant entity types. For each significant entity type, an extension to the Subentity Extractor can be specified. If no subentity extractor is specified, no subentities will be in existence for entities of this type. If an old significant entity type is replaced with a new one, subentities for all existing significant entities of this type will be re-extracted.

Project Options, including:

Project Search Path (see HB.17.) -- Allows projects to be rooted in different parts of the system. This affects the list of projects that appear under the Project node.

Log file path (see HB.9.) – Allows user to change the location of the log file recorded by the Modification Analyzer.

Alias filename – Allows user to specify shortcuts for full formal significant entity names. This is done by mapping a nickname to an expanded name. This text file contains one line per nickname, and contains the nickname (which must be one word), followed by a space, followed by the significant entity name, enclosed in braces ('{' and '}').

Email server – The name or IP address of the SMTP-compatible email server to contact when sending mail messages to team members.

Email sender – The sender name to use when sending these email messages.

Email recipient – The email address of the email recipients, separated by commas.

Project Integrity -- Upon startup, the Project Manager should analyze the integrity of the project, including:

Detection and graceful handling of corrupt projects (see HD.1.) -- Should notify user if project is corrupt, but there is no need to recover.

Detection and graceful handling of missing entities (see HD.2.) -- Should notify user which entities are missing upon startup.

Detection and graceful handling of corrupt entities (see HD.3.) -- Should treat corrupt entities as if they are missing.

Project Composition

A CASCADE project is composed of the following. See the CASCADE DTD section for more details.

A list of significant entities. Each significant entity can be of any form and in any language, programming or otherwise (see HA.1. and HA.3.). Note that more than one formal name can possibly map to the same significant entity because of the existence of nodes.

A tree of nodes, representing the structure of the significant entities, various flag nodes (see HB.19.), and special "Relationship Reflection" nodes, including:

Projects (see HF.1.)

Relationships (see HF.2. and HF.3.)

Events (see HF.4. and HF.5.)

Actions (see HF.6. and HF.7.)

A list of possible actions.

A list of possible events.

A list of established relationships.

Node Composition

A node is composed of the following. See the CASCADE DTD section for more details.

Formal Name -- Composed of a series of names, each of which is up to 60 characters and may contain spaces. The names are separated by the forward slash ('/') character.

Keywords -- Keywords to be associated with this node, so that it can be located with a search.

Links (see HB.12.) -- A list of informational links and their descriptions, allowing a user to find out more information about this node or related concepts.

Physical Filename -- If appropriate, the physical filename or URL of this significant entity.

Significant Entity Composition

A significant entity appears as a node within a project, and its subentities appear as child nodes. A significant entity is composed of the following. See the CASCADE DTD section for more details.

Name -- Up to 60 characters, and may contain spaces, but not slashes ('/'). The formal name of the node representing this significant entity is indicated in the glossary entry for significant entity.

Entity Type -- The type of entity; used to choose which subentity extractor is appropriate.

Keywords -- Keywords to be associated with this significant entity, so that it can be located with a search. The node representing this significant entity will share the same keywords.

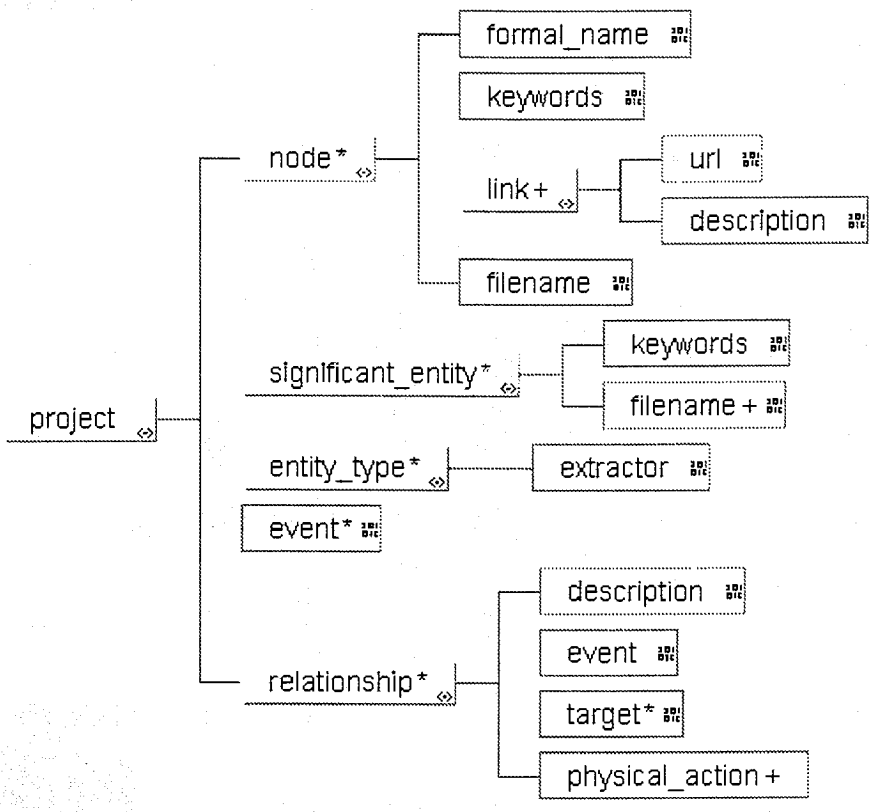
Subentities -- A list of subentities within this significant entity, created by the subentity extractor. The formal name of the node representing this subentity is indicated in the glossary entry for subentity. Each subentity is composed of:

Name -- Up to 60 characters, and may contain spaces, but not slashes ('/').

Keywords -- Keywords to be associated with this subentity, so that it can be located with a search. The node representing this subentity will share the same keywords.

CASCADE DTD

Projects are stored in XML format. The following is the XML DTD for CASCADE projects.



This diagram was generated using XML Authority™. In the diagram, diagonal lines mean a choice, whereas straight lines mean a sequence. A name followed by a plus indicates that zero or more can appear. The following is the DTD source:

```
<!ENTITY % cascade_name "">
~

~
<!ELEMENT project  (node* , significant_entity* , entity_type* ,
event* , relationship* )>
~
<!ATTLIST project  name                CDATA  #REQUIRED
~
                   project_search_path CDATA  #IMPLIED
                   change_log_path     CDATA  #IMPLIED
                   alias_path          CDATA  #IMPLIED
                   mail_server         CDATA  #IMPLIED
                   mail_sender         CDATA  #IMPLIED
                   mail_recipients     CDATA  #IMPLIED >
```

```
<!ELEMENT node  (formal_name , keywords , link+ , filename )>

<!ELEMENT formal_name  (#PCDATA )>
<!ATTLIST formal_name  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT keywords  (#PCDATA )>
<!ATTLIST keywords  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT link  (url , description )>

<!ELEMENT filename  (#PCDATA )>
<!ATTLIST filename  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT url  (#PCDATA )>
<!ATTLIST url  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT description  (#PCDATA )>
<!ATTLIST description  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT significant_entity  (keywords , filename+ )>
<!ATTLIST significant_entity  name CDATA  #REQUIRED
                              type CDATA  #REQUIRED >
<!ELEMENT entity_type  (extractor )>
<!ATTLIST entity_type  name CDATA  #REQUIRED >
<!ELEMENT physical_action EMPTY>
<!ATTLIST physical_action  type      CDATA  #REQUIRED
                           parameter CDATA  #REQUIRED >
<!ELEMENT type  (#PCDATA )>
<!ATTLIST type  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT parameter  (#PCDATA )>
<!ATTLIST parameter  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT event  (#PCDATA )>
<!ATTLIST event  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT relationship  (description , event , target* ,
physical_action+ )>
<!ATTLIST relationship  name CDATA  #REQUIRED >
<!ELEMENT extractor  (#PCDATA )>
<!ATTLIST extractor  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT entity_type_name  (#PCDATA )>
<!ATTLIST entity_type_name  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT target  (#PCDATA )>
<!ATTLIST target  e-dtype NMTOKEN  #FIXED 'string' >
```

Subentity Extractor

Description

The Subentity Extractor's main purpose is to increase the granularity of a significant entity by breaking it down into subentities. It does so by turning any significant entity into an XML document (using the CASCADE node) that represents its contents in a format that is readable to CASCADE. This XML document also provides an easy way for the Modification Analyzer to compare two versions of the same significant entity and determine what, if anything, has changed. Note that this translation process makes for graceful handling of read-only code and documentation (see HA.9.). Because it is difficult, if not impossible, to come up with a program that translates any file

format in any language (see HA.1.) into XML, the Subentity Extractor relies on extensions that do this job. An API is provided so that new extensions can be developed, thus allowing a development team to have CASCADE analyze their existing code and documentation.

Dependencies

Note that the subentity extractor will never require user intervention once it is set up. It is designed to run behind the scenes to provide uniform access to all significant entity types. The Subentity Extractor provides services to the Project Manager and the Modification Analyzer. It does not rely on the services of any other module.

Responsibilities

The Subentity Extractor and its comprising extensions are responsible for the following:

Aid in entity type recognition (see HB.3.) -- The Project Manager will call on the Subentity Extractor to help it determine the entity type for a given significant entity. The user can override this initial guess, so that when the actual extraction takes place, the Subentity Extractor is to use the entity type specified by the user.

Node Creation -- The Subentity Extractor must be able to turn a generic significant entity into a tree of nodes with formal node names that are recognizable (see HB.5.) to the rest of the system.

Hint extraction -- Significant entities may have embedded hints (see HA.2.) that should be extracted and relayed to the rest of the system. These hints may take the form of any of the following:

Suggested relationships with other entities

Search keywords (see HB.10.)

URL links (see HB.12.)

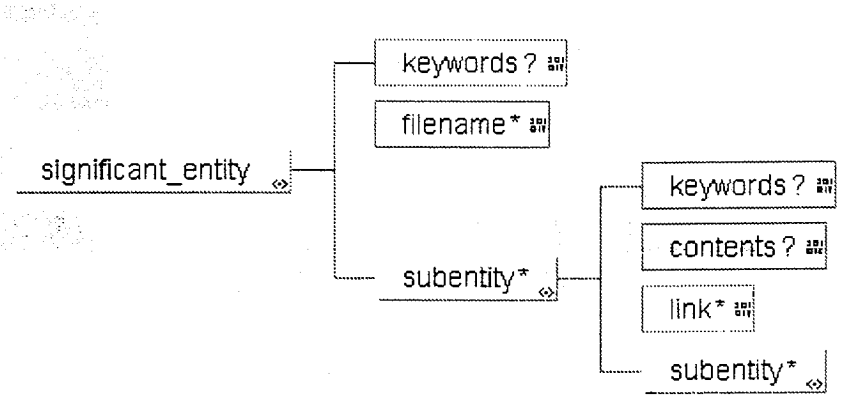
Creation, modification, and deletion of Flag Nodes (see HB.19.)

Significant Entity Type Registration (see HB.16.) -- The user will register new entity types through the Project Manager. The Subentity Extractor will recognize this registration and be able to extract subentities from these significant entities on the user's demand.

Advanced implementations of the Subentity Extractor will be able to automatically suggest relationships by analyzing the similarity of keywords and content between nodes. This is not a requirement of all CASCADE implementations, but will increase the power and ease of use of the CASCADE system (see HE.1.).

CASCADE Node XML DTD

This section describes the standard CASCADE XML DTD for subentities that have been extracted for content.



The diagram is in the same format as the standard CASCADE DTD diagram, but is not the same DTD. The following is the DTD source:

```
<!ELEMENT significant_entity  (keywords? , filename* , subentity* )>
<!ATTLIST significant_entity  name CDATA  #REQUIRED
                              type CDATA  #REQUIRED >
<!ELEMENT keywords  (#PCDATA )>
<!ATTLIST keywords  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT subentity  (keywords? , contents? , link* , subentity* )>
<!ATTLIST subentity  name     CDATA  #REQUIRED
                     checksum CDATA  #REQUIRED
                     type     CDATA  #REQUIRED >
<!ELEMENT contents  (#PCDATA )>
<!ATTLIST contents  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT filename  (#PCDATA )>
<!ATTLIST filename  e-dtype NMTOKEN  #FIXED 'string' >
<!ELEMENT link  (#PCDATA )>
<!ATTLIST link  url         CDATA    #REQUIRED
```

```
description CDATA    #REQUIRED
e-dtype     NMTOKEN  #FIXED 'string' >
```

API Specification

The Subentity Extractor provides a programmer's API that allows extensions to be built to allow translation of new significant entity types into XML (see HB.16. and HE.2.). This section specifies the functionality of this API.

The API provides the following services:

Obtain the name of the file or files associated with the significant entity.

Operations for the creation of a node tree, with the significant entity as the root of the tree.

Read access to the entire node tree for the project.

Ability to establish relationships between nodes for this significant entity and subentities and any other node in the project.

Ability to create new events.

Standard Extensions

The following extensions to the Subentity Extractor are standard:

NullExtractor

Does no extraction - entities using this extension will have no subentities and no extraction will take place.

HTMLExtractor

Processes standard HTML files. The HTML file will be broken down into a hierarchical subentity tree based first on filenames and then on "h" tags. For example, the following HTML file:

```
Stuff

<H1>Section 1</H1>

Some text

<H2>Section 1.1</H1>

Some more text
```

```
<H2>Section 1.2</H1>
Even more text
```

would be treated as follows:

```
content: Stuff
```

--subentity: Section 1

--content: Some text

----subentity: Section 1.1

----content: Some more text

----subentity: Section 1.2

----content: Even more text

JavaExtractor

Processes standard .java files in a project. The java files must first be processed by the CASCADEDoclet Javadoc doclet, which will scan the Java files and create an XML document that represents the files' contents. To do this preprocessing, execute the following command:

```
  javadoc -doclet
edu.rit.cs.cascade.extractor.extension.CASCADEDoclet
```

files

The root of the tree is the significant entity. Each package is a subentity, and package names are processed such that each dot starts a new subentity. The class name stats a new subentity under the full package name. Inner classes and methods or attributes start new subentites under a class name.

Relationship Editor

Description

The Relationship Editor is the primary user interface for navigating the node

tree of a project and establishing relationships between nodes. By decoupling the relationships from the actual files, CASCADE can record relationships between multiple entities without modifying the files themselves, thus providing graceful support for read-only code and documentation (see HA.9.) and eliminating code clutter. At the same time, by allowing the Subentity Extractor to extract hints from code and establish relationships, users are free to embed hard-coded links within documents where they choose. The Relationship Editor will be able to gracefully handle access to a project by multiple users at a time (see HA.7.).

Dependencies

The Relationship Editor provides services to the Project Manager. It also relies on the services of the Project Manager.

Responsibilities

The Relationship Editor has the following responsibilities:

Navigation (see HB.11.) -- The user will be able to navigate a project's node tree, and view all available information about each node (see Node Composition). This includes the ability to visit associated links through the user's web browser. The following notes apply:

Launching -- Navigation is a frequently used component that will be accessible to the Project Manager and the Relationship Analyzer and can be launched from within an IDE or from the command line.

Circular dependencies -- During navigation, the Relationship Editor should be aware of the possibility of circular dependencies between nodes (see HD.4.).

Special nodes -- Navigation should be aware of various special nodes including the following. These nodes should be continuously maintained.

Flag Nodes (see HB.19.)

Relationships (see HF.3.)

Events (see HF.5.)

Actions (see HF.7.)

Navigation only mode -- The Relationship Editor can be launched in "Navigation Only" mode, in which case users cannot register new events or

create or maintain relationships or flag nodes.

Searching (see HB.10.) -- The user can find a node by searching for keywords associated with various nodes in the system.

Flag Node Creation and Maintenance (see HB.19.) -- Navigation should treat flag nodes as it does any other node. In addition, users can create, delete, and modify the properties of flag nodes.

Event Registration and Maintenance (see HB.6.) -- Allow users to register events for later detection (see the glossary definition of event). Users shall also be able to modify and delete events (see HB.14. for details on how this is to be handled).

Relationship Construction and Maintenance (see HB.8.) -- Allow users to create new relationships by associating an event with one or more actions (see glossary definition of event, action, and relationship). Users shall also be allowed to modify and delete or disable existing relationships, including those automatically suggested by CASCADE (see HB.13.). The following notes apply:

Wildcards -- When defining the source of an event or the nodes affected by an action, wildcards can be used (see HB.18. for details on how this is to be handled).

Projects -- Under relationship reflection (see HF.1.), all projects in the project search path are considered significant entities as well, and therefore, relationships can be established across projects.

Command line support -- Users will have the ability to create new relationships from the command line.

Error Detection and Handling (see HD.1., HD.2., HD.3.) -- The Relationship Editor shall be able to gracefully handle corrupt projects, missing entities, and corrupt entities as specified. It uses the Project Manager to detect these errors.

Modification Analyzer

Description

The purpose of the Modification Analyzer is to notice when a significant entity has changed, and provide information about what subentities have changed. The user launches the modification analyzer on a regular basis, either ad-hoc, or after every save from an editor, or at every check-in of a

version-controlled file, or at regular timed intervals. It relies on the services of the Subentity Extractor to produce an XML document representing the contents of the significant entity. It stores the current version of each XML document along with the project, and compares the latest XML document with the stored version. The analyzer reports whether any differences are detected, and if there are differences, the analyzer determines where they are based on the structure of the XML document, and replaces the stored document with the latest XML document. The information on the nature of the changes is passed along to the Relationship Analyzer so that it can determine what these changes mean in the context of the relationships that have been established.

The Modification Analyzer must support arbitrary team sizes by being able to analyze changes made by different developers, and by being able to be run simultaneously by multiple users (see HA.7.).

Dependencies

The Modification Analyzer provides services to the user or the user's IDE through the command line. It relies on the services of the Subentity Extractor and the Relationship Analyzer.

Responsibilities

The Modification Analyzer is responsible for the following:

Current Version Archival -- Keeping the most recent version of all significant entities archived in XML format so that it has a previous version to compare against. This archival is handled internally and does not require revision control or versioning to be available from the current platform (see HA.6.). In order to support lifecycles that include version branches (see HA.5.), the ability to take a snapshot of the system in its current state should be provided.

XML Comparison -- Must be able to compare two XML documents in the CASCADE DTD, and create a list of nodes that have changed and their formal node names (see HB.5.), and causes for the change (see the glossary definition of event for a list of causes).

Node Tree Structure Comparison -- Must be able to detect changes in the structure of the node tree and create a list of nodes that have changed, and causes for the change (see the glossary definition of event for a list of causes). This includes:

Flag nodes (see HB.19.)

Project nodes (see HF.1.).

Relationships (see HF.3.)

Events (see HF.5.)

Actions (see HF.7.)

IDE Integration (see HA.4.) -- Will be easily integratable with existing IDEs by providing the ability to run the analyzer and access all of its functionality from the command line. This provides the flexibility to either be run every time a significant entity is saved from within an IDE, or every time a file is checked in, or every time a user feels it is necessary to run the analyzer (see HA.8.).

Change Log -- Keeping a log of changes noticed and when they were noticed (see HB.9. for details as to what will be stored in the log).

Invocation of Relationship Analyzer -- When a change is detected, the Relationship Analyzer is launched, and is passed the list of all changed nodes, along with the nature of the changes.

Advanced versions of the Modification Analyzer will provide an Event Detection API (see HE.2.) with which programmers can provide an extension to the Modification Analyzer to allow it to more accurately detect changes and associate them with the correct subentities and causes. This API will provide programmers access to the node tree, to the original XML document, and to the new XML document. It will also provide access to the original file (if relevant) so that more hints can be extracted. The API will also allow the programmer to add to the list of nodes that have changed.

Relationship Analyzer

Description

The purpose of the Relationship Analyzer is to determine which events are fired, given a list of nodes that have changed and the cause of the changes from the Modification Analyzer. After this is determined, a list of actions to be performed is generated and those actions are performed.

Dependencies

The Relationship Analyzer provides services to the Modification Analyzer.

Advanced implementations will rely on the Modification Analyzer.

Responsibilities

The Relationship Analyzer has the following responsibilities:

Event Recognition (see HB.6.) -- Given the formal node name of a significant entity, a list of subentities that have changed, and the nature of the changes, be able to determine which events are fired for each relationship established with this significant entity as the source.

Action Execution (see HB.7.) -- Given a list of events that have been fired, determine which actions should be executed, and execute them. Note that CASCADE does not modify code itself. Instead, it recommends actions to the user, or runs scripts that make modifications for the user (see the glossary entry for action for more details as to what actions can be performed). This provides support for read-only code and documentation, while still being flexible enough to modify the code and documentation automatically if that is what the user wants (see HA.9.).

Flexible Output -- Be able to send output to a file, to standard out, or via e-mail to any member on the development team (see HA.7.). This allows for easy integration with existing IDEs (see HA.4.)

Wildcards (see HB.18.) -- The Relationship Analyzer should be able to handle wildcards that specify the nodes affected by an action.

Flag Nodes (see HB.19.) -- The Relationship Analyzer should be able to recognize relationships with flag nodes.

An advanced implementation of the Relationship Analyzer will provide an event action API (see HE.2.). This API will allow for the possibility of advanced actions to be performed. The API should provide the programmer with the ability to perform the standard actions (see glossary entry for action) as well as establish new relationships, update the node tree, and modify significant entities, causing the Modification Analyzer to be launched. In the latter case circular dependencies should be handled gracefully, as to prevent infinite recursion.

Event Conditions

To determine whether an event has occurred, the event condition for each established relationship is parsed and checked to see whether it returns true or false. The following is the grammar for which event conditions are written. This grammar file is compatible with the ANTLR system, available

at http://www.ANTLR.org.

```
/**
 * Grammar for event conditions, to be parsed using the ANTLR system,
 * available in the public domain from www.ANTLR.org
 *
 * File:   $Id$
 * Author: Mark Roth
 * Date:   7/12/1999
 *
 * Revision History:
 *   $Log$
 *
 */

////////////////////////////////////////////////////////////////////////////
// Parser
////////////////////////////////////////////////////////////////////////////

class EventConditionParser extends Parser;
options {
 buildAST = true; // uses CommonAST by default
 k = 5;    // Five token lookahead
}

tokens {
 ADDED; DELETED; EXISTS; MODIFIED; SUBENTITY_ADDED; SUBENTITY_DELETED;
 SUBENTITY_MODIFIED; INDIRECT_SUBENTITY_ADDED;
 INDIRECT_SUBENTITY_DELETED; INDIRECT_SUBENTITY_MODIFIED;
 SEMI; OP; NOT; LPAREN; RPAREN;
 ENTITY;
}

expr
 : mcond SEMI
 ;

mcond
 : cond (OP^ cond)*
 ;

cond
 : func ENTITY^
 | NOT^ cond
 | LPAREN! mcond RPAREN!
 ;

func
 : ADDED
 | DELETED
 | EXISTS
 | MODIFIED
 | SUBENTITY_ADDED
 | SUBENTITY_DELETED
 | SUBENTITY_MODIFIED
 | INDIRECT_SUBENTITY_ADDED
 | INDIRECT_SUBENTITY_DELETED
 | INDIRECT_SUBENTITY_MODIFIED
 ;
```

```
////////////////////////////////////////////////////////////////////////////
// Lexical Analyzer
////////////////////////////////////////////////////////////////////////////
class EventConditionLexer extends Lexer;

options {
 k=20;
 testLiterals=false;
}

WS
  : ( ' '
    | '\t'
    | '\r'
    | '\n'
    )
    { _ttype = Token.SKIP; }
  ;

SEMI
  : ';'
  ;

LPAREN
  : '('
  ;

RPAREN
  : ')'
  ;

OP : "or" | "OR" | "and" | "AND" ;

NOT
  : "not"
  | "NOT"
  ;

ADDED
  : "added"
  | "ADDED"
  ;

DELETED
  : "deleted"
  | "DELETED"
  ;

EXISTS
  : "exists"
  | "EXISTS"
  ;

MODIFIED
  : "modified"
  | "MODIFIED"
  ;

SUBENTITY_ADDED
  : "subentity_added"
  | "SUBENTITY_ADDED"
  ;
```

```
SUBENTITY_DELETED
 :"subentity_deleted"
 | "SUBENTITY_DELETED"
 ;

SUBENTITY_MODIFIED
 :"subentity_modified"
 | "SUBENTITY_MODIFIED"
 ;

INDIRECT_SUBENTITY_ADDED
 :"indirect_subentity_added"
 |"INDIRECT_SUBENTITY_ADDED"
 ;

INDIRECT_SUBENTITY_DELETED
 :"indirect_subentity_deleted"
 | "INDIRECT_SUBENTITY_DELETED"
 ;

INDIRECT_SUBENTITY_MODIFIED
 :"indirect_subentity_modified"
 | "INDIRECT_SUBENTITY_MODIFIED"
 ;

//
// Any characters are valid for an identifier except for { and }
// (' ' = 32, 'z'   122, '{' = 123, '|' = 124, '}' = 125, '~' = 126)
//
ENTITY
 options{ testLiterals=true; }
 : '{' (' '..'z'|'|'|'~')* '}'
 ;

////////////////////////////////////////////////////////////////////////////
// END
////////////////////////////////////////////////////////////////////////////
```

Appendix C

SYSTEM IMPLEMENTATION

Rather than attaching endless pages of printed Java™ source code, the source distribution can be found in CD-ROM format bundled with this thesis. An online version of Javadoc™ for the system is also on the CD.

Current Implementation Limitations

CASCADE can be continuously expanded to provide new functionality. The current implementation included with this thesis is a minimal implementation, though still complete enough to prove that it works. The following features have yet to be implemented:

- Extensions need to be written to support C++, C, BASIC, and other common languages. Currently, only the Java language is directly supported.
- Extensions need to be written to support Microsoft Word, LaTeX, and other common document formats. Currently, only HTML documentation is supported.
- The current implementation of CASCADE has not yet been tested on any platform other than PC. All tools and libraries are available for UNIX and other Java-supporting platforms, but no testing has been done as of yet (minor tweaking may be necessary).

- The current implementation of CASCADE does not currently smoothly support large development teams. It is easy for people to step on each other's toes. Only one person should use this implementation at a time.
- There is currently no cross-project support implemented. Each project is in its own separate environment.
- Wildcards for project names are not currently supported.
- There is limited support for flag nodes at this time. The basis of flag node implementation was covered in Chapter 7 of this thesis, but there are still other things that need to be done for flag nodes to be complete, such as actions that modify flag nodes.
- None of the advanced requirements are currently implemented, including relationship reflection.
- Some of the advanced APIs still need an implementation, including the ability to extend the relationship analyzer and the modification analyzer.

Implementation Technologies

The following technologies were used in implementing CASCADE. In order for CASCADE to run, each of these components must first be installed.

Java JDK 1.2

The software was written to be compatible with the Java JDK 1.2, and its VM. Specific features used in JDK 1.2 include:

- The Java Foundation Classes (JFC), for GUI design. In particular, the Swing classes and the HTML editor components.
- Reflection API, for automatic detection of extensions. To add an extension, simply create a class than inherits from the Extension class and place it in the corresponding package. Reflection allows CASCADE to

automatically detect the presence of new extensions at runtime and query the extensions for information.

- Collections Framework. Various basic data structures were used from the collections framework.

More information on this technology can be obtained at http://java.sun.com/products/jdk/1.2/index.html.

Java Project X, Technology Release 2

Though not yet standard, this is a very powerful library for Java that provides an XML parser and encoder. It is quite fast, and can make use of any standard XML DTD.

In CASCADE, this technology is used to save and load project and relationship information in a standard format that other tools can get to if need be. In addition, the Subentity Extractor converts any document (e.g. HTML or Java source code) into a tree of subentities. This tree of subentities is stored in XML format, giving it structure and making it much easier to compare two distinct hierarchies.

More information on this technology can be obtained at http://java.sun.com/xml/.

ANTLR 2.6.1

ANTLR stands for ANother Tool for Language Recognition. It is a parser and translator generator tool, similar to lex and yacc for UNIX. ANTLR generates Java™ source code that can parse an expression, using a user-defined grammar. It is a very powerful tool and supports some interesting concepts such as grammar inheritance.

In CASCADE, conditions for events are written using a condition definition language. ANTLR was used to generate a parser that can interpret strings written in this language. The classes that ANTLR generated are found in the edu.rit.cs.cascade.common package, and are called EventConditionLexer, EventConditionParser, and EventConditionTokenTypes. The class that launches the parser is called EventDetector.

More information on this technology can be obtained by visiting http://www.ANTLR.org/.

JavaMail 1.1

JavaMail is a Java Platform standard extension that provides a framework for sending, receiving, and filing email messages. It is expandable to support any email interface, and comes standard with the ability to deliver SMTP e-mail messages.

CASCADE uses this to send email to development team members when changes are detected in various documents and action needs to be taken. This is done in the relationship analyzer, automatically launched from the modification analyzer. Each action can be configured to either output a string to the screen, or send a detailed email message to one or more users.

More information on JavaMail can be obtained by visiting http://java.sun.com/products/javamail/index.html.

JavaBeans Activation Framework, Version 1.0a

This framework was not used directly, but it is a required component for the JavaMail library.

More information about this technology can be obtained by visiting http://java.sun.com/beans/glasgow/jaf.html.

Appendix D

USER'S GUIDE

This section presents an introductory User's Guide for the current implementation of CASCADE described in Appendix C. This is a User's guide written in the minimalist spirit, and it is sufficient to get a development team started with using CASCADE, but does not serve as a full reference guide. Recall that the current implementation of CASCADE is **not** production-quality and is not yet recommended for actual use other than for testing purposes. This document is reformatted to fit this thesis and it is also available on the accompanying CD.

Description

This document describes how to use the CASCADE system in its current implementation.

Installation

It should be noted that installation has not been attempted on any system other than Windows NT and Windows 95 / 98. However, theoretically everything should work properly under UNIX as well. There is a strong possibility that minor tweaks may be needed to enable CASCADE to work on other systems.
Before installing the CASCADE source distribution itself, a number of external Java libraries need to be installed first. These libraries are all available free of charge and can be obtained by following the specified links:

Java JDK 1.2 -- See http://java.sun.com/products/jdk/1.2/index.html.

Java Project X TR 2 -- See http://java.sun.com/xml/.

ANTLR 2.6.1 -- See http://www.ANTLR.org.

JavaMail 1.1 -- See http://java.sun.com/products/javamail/index.html.

JavaBeans Activation Framework 1.0a -- See
http://java.sun.com/beans/glasgow/jaf.html.

Once these libraries are successfully installed and set up, CASCADE can be installed as follows:

Download the source distribution.

Unzip the distribution to any available directory (in this example we will use C:\CASCADE\).

Add "C:\CASCADE\src\" to your CLASSPATH environment variable.

Compile

To compile cascade, a batch file has been provided that works on PC's. This batch file assumes the CLASSPATH has been properly set and it compiles all java files from within the src directory. To compile the system, make C:\CASCADE\src\ your current directory, and type:

```
makeall
```

To remove all class files, type:

```
clean
```

To generate Javadoc documentation, type:

```
makejavadoc
```

These are relatively simple batch files, and equivalent UNIX shell scripts can be written fairly easily.

Test Run

To ensure CASCADE is working, it is recommended you try a test run. An example project is included in the ...\projects\example\ directory under the installation. Make C:\CASCADE\Projects\example\ your current working directory, and execute the 'run' batch file to start CASCADE. The following batch files are available:

`run` -- Starts the CASCADE Project Manager.

`runeditor` -- Starts the CASCADE Relationship Editor directly, without invoking the Project Manager.

`runanalyzer` -- Starts the CASCADE Modification Analyzer, which also

invokes the Relationship Analyzer if appropriate.

`makecode` -- This should be run whenever a significant change has been made to the source code. CASCADE will not detect changes to source code without first running this batch file.

Note that the example is set up assuming you have installed everything using the C:\CASCADE directory. If you installed everything under a different directory, you will have to manually edit all .xml files and do a global search and replace on C:\CASCADE, replacing it with the directory you installed everything in.

Template Directory

To start a new project, you can make a copy of the provided Template directory (C:\CASCADE\Projects\Template). This will provide you with the dtd files you need to have in your working directory in order for CASCADE to work. Additionally, you can copy over the batch files in the example directory, replacing CASCADE.xml with the name of your project file, once it is created in the project manager.

Usage

This section provides an overview of how the system is used. More details can be found by reading the Functional Specification, or the thesis itself.

Overview

The CASCADE system is designed to keep code and documentation up to date. It does so by observing any important significant entities for changes, and when changes are detected, it uses the relationship information provided by the user to determine what actions should be taken to keep the documentation and code up to date. The steps in using CASCADE are outlined as followed, and described in detail in this document:

> Project Creation -- Create a new project and register significant entities Relationship Construction -- Inform CASCADE of the relationship of entities and subentities within the system. Modification Analysis -- Instruct CASCADE to analyze recent changes and recommend actions and interpret the results produced by CASCADE.

Project Creation

Before CASCADE can do anything, it needs to be set up. This can be

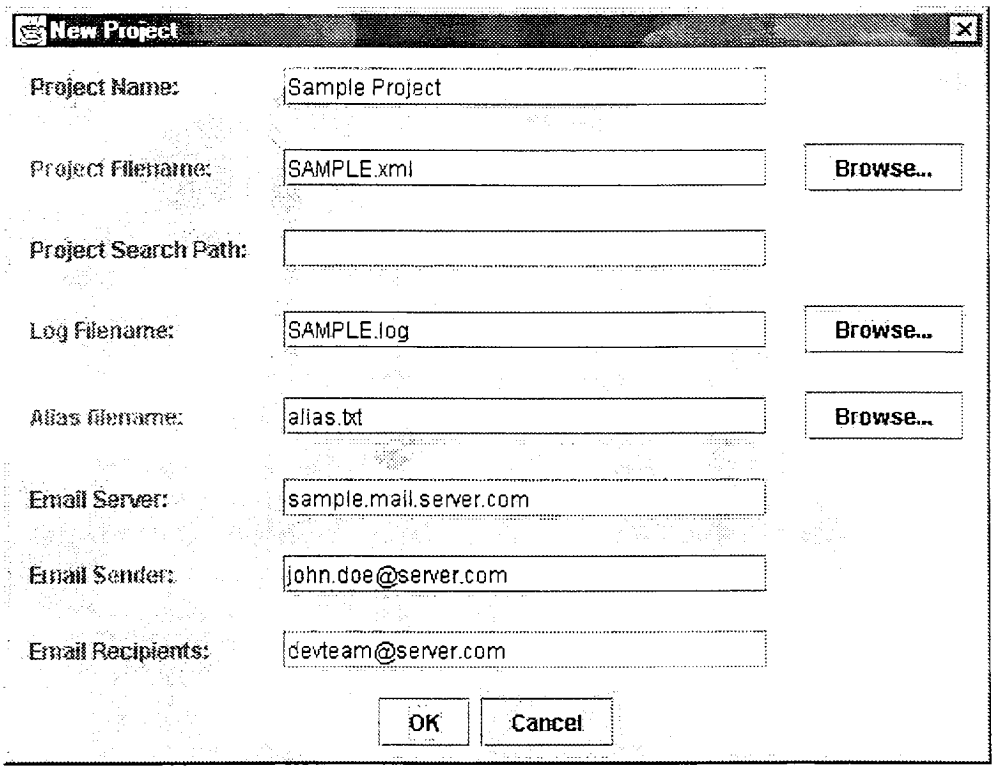
achieved by creating a new project. To create a project, follow these steps:

Copy the C:\CASCADE\template\ directory over to a new directory. This directory gives you the basic files you need to start a new project. **NOTE: Currently, all projects MUST be stored in the C:\CASCADE\Projects\ directory. Otherwise, CASCADE will not be able to find the resources it needs.**

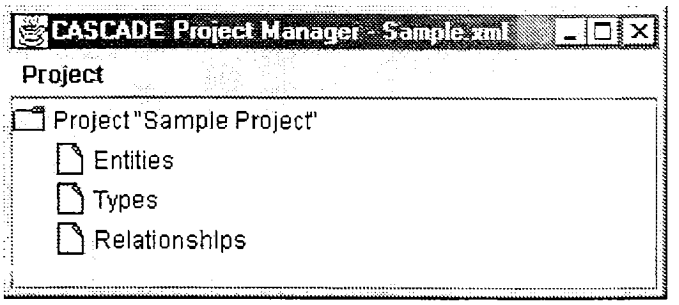
Start the Project Manager by typing:

```
java edu.rit.cs.cascade.manager.ProjectManager
```

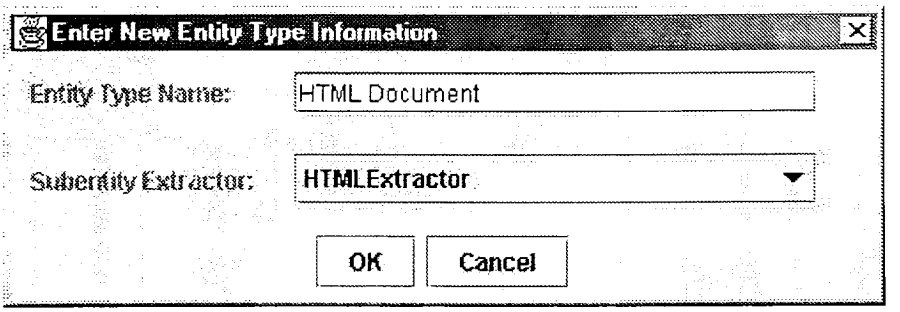
Create a new project by selecting New from the Project menu. A New Project options dialog appears. Fill in the required information. Project Search Path is reserved for future use and can be left blank. Example:



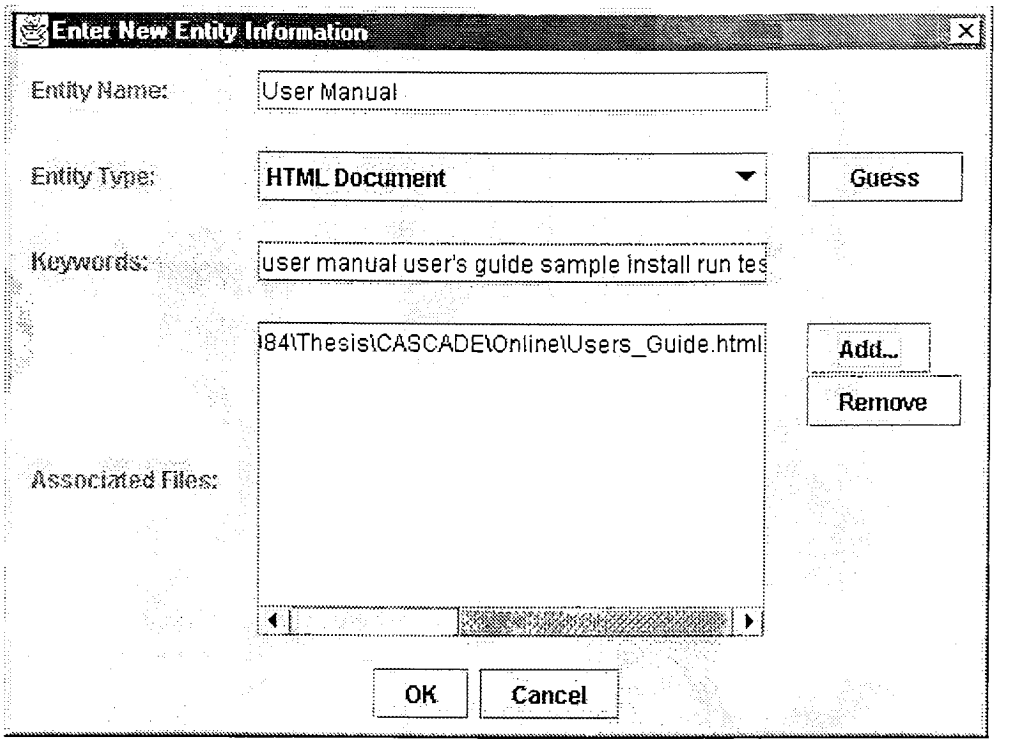
Once the project is created, you will get a window that looks like this:



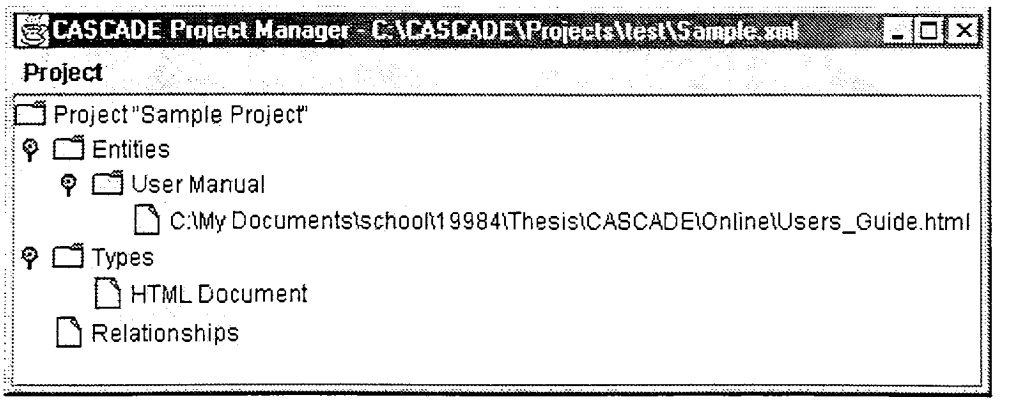
Eventually, under entities will appear all significant entities that are important for CASCADE to analyze for changes. Types will contain a list of all possible entity types (such as "Java Source Code" or "HTML Requirements Document"). Finally, under Relationships will be a list of all relationships between entities. Before we continue, we must define entity types. This can be done by right-clicking on Types and selecting "New Type..." which will bring up the following dialog:



Enter the entity type and choose from a list of available extractor extensions (if you add extensions to the edu\rit\cs\cascade\extractor\extension directory, this list will dynamically change). Then, click OK. Your Project window should now contain the entity type you just entered. Now you are ready to add an entity to the system. To do this, right-click on Entities and select "New Entity..." You will be prompted with the following dialog box:



Specify an entity name (will be used to refer to this entity), type in some keywords to describe the entity, and select which files the entity consists of. You can click Guess for CASCADE to automatically guess which entity type to use, or you can specify one yourself. Click OK when you are done. You should now have a project screen that looks something like this:



Immediately after you click OK, subentities are extracted from the files you specified. This allows CASCADE to remember what a files looks like

before you changed it. You will notice an XML file in your directory for each significant entity you specify.

Continue to add new entity types and entities until all entities in your system have been specified. At this point, you have finished creating your project. Select Exit from the Project menu to save your project. The project will be saved in an XML file using the CASCADE DTD.

Relationship Construction

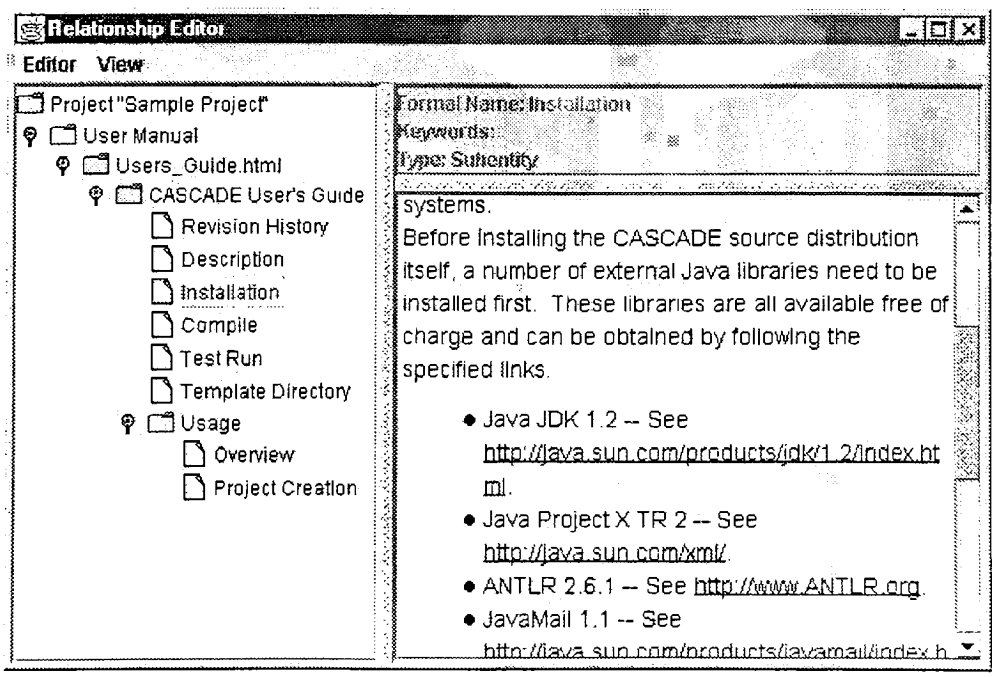
Now that your project is created and you have specified entities, you can begin editing relationships. To do so, bring up the Project Manager and follow these steps:

To bring up the project manager, type:

```
java edu.rit.cs.cascade.manager.ProjectManager Sample.xml
```

(substitute Sample.xml with your project filename).

Right-click on the Relationships folder and click Launch Relationship Editor (alternatively you can use the Project menu to do the same thing). This will bring up the relationship editor, which looks like this:



On the left panel is a hierarchical view of all significant entities registered in the system. By clicking on a subentity, an HTML representation of the subentity selected will appear in the right frame. If a Java class had been selected, for example, you may see the Javadoc comments in HTML format on the right pane. What is visible in the right pane depends on the subentity extractor extension used to extract the information. In this case, the subentity is this document, and the default HTML extractor extension retrieves hierarchy information by looking at the Heading tags.

To create a relationship, select two or more subentities in the left pane and bring up the Editor menu. You can choose one of three options (for these explanations, assume you have selected A, B, C, D, and E, in that order):

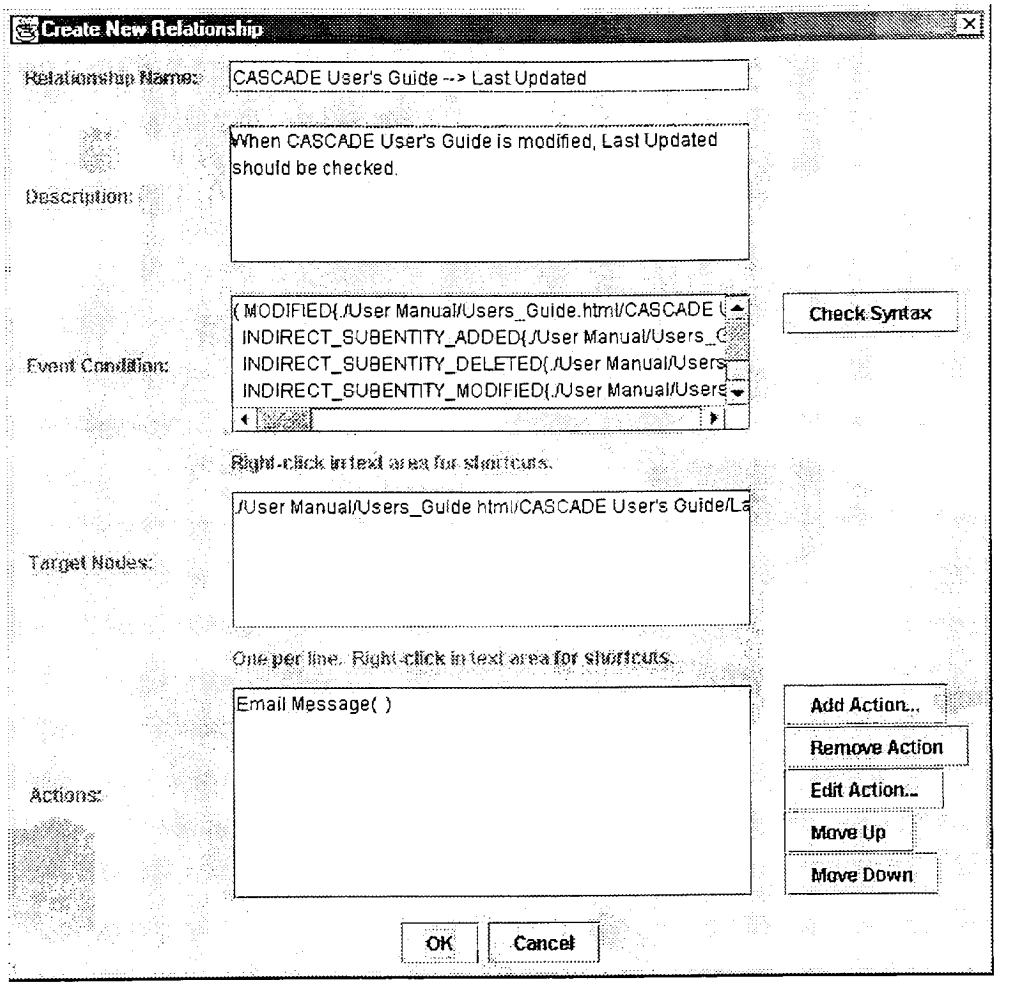
Create Relationship -- This creates a single, uni-directional relationship using the first selected node as the source of the event, and using the second, third, fourth, etc. nodes selected as the targets of the relationship. This would yield the relationship "A -- B, C, D, E".

Create Bi-directional Relationship -- This creates all possible bi-directional relationships using each selected node as a source node exactly once, with all other selected nodes as target nodes. This would yield the relationships "A -- B, C, D, E", "B -- A, C, D, E", "C -- A, B, D, E", "D -- A, B, C, E", and "E -- A, B, C, D."

Create From Source to Each Destination and Back -- This creates all possible bi-directional relationships consisting of only one source and one destination. The first node you selected will always be either a source or a destination. This would yield the relationships "A -- B", "B -- A", "A -- C", "C -- A", "A -- D", "D -- A", "A -- E", "E -- A."

Note that these options are only conveniences. You can create any type of relationship you want, but these help reduce the tedium.

Next, you will be prompted by one or more dialogs. If you selected either "Create Bi-directional Relationship" or "Create From Source to Each Destination and Back," then CASCADE will ask you whether you want to be prompted for each relationship. Answer Yes if you wish to customize any of the default options, or No if you wish CASCADE to do all the work without asking you anything. If you wish to customize a relationship, you will be prompted by the following (rather large) dialog:



First, choose a name for the relationship. Next, enter a detailed description of the relationship. This is the description users will see when they receive a message from CASCADE recommending an action to take. Enter as much detail as possible. Next, enter the conditions that fire this event. Right-clicking in this text area will give you a list of available commands for constructing this boolean statement. Any boolean statement can be constructed using parenthesis, AND, OR, and NOT. Items inside braces are full formal names (based on the hierarchy) of significant entities. The following commands are available:

ADDED -- The given node did not previously exist, and now exists.

DELETED -- The given node used to exist, but now no longer exists.

EXISTS -- The given node may or may not have previously existed, but does currently exist.

MODIFIED -- Changes have been made to the content of the entity represented by the given node.

SUBENTITY_ADDED -- A subentity has been added directly below this node.

SUBENTITY_DELETED -- A subentity has been deleted directly below this node.

SUBENTITY_MODIFIED -- The contents of a subentity directly below this node have been modified.

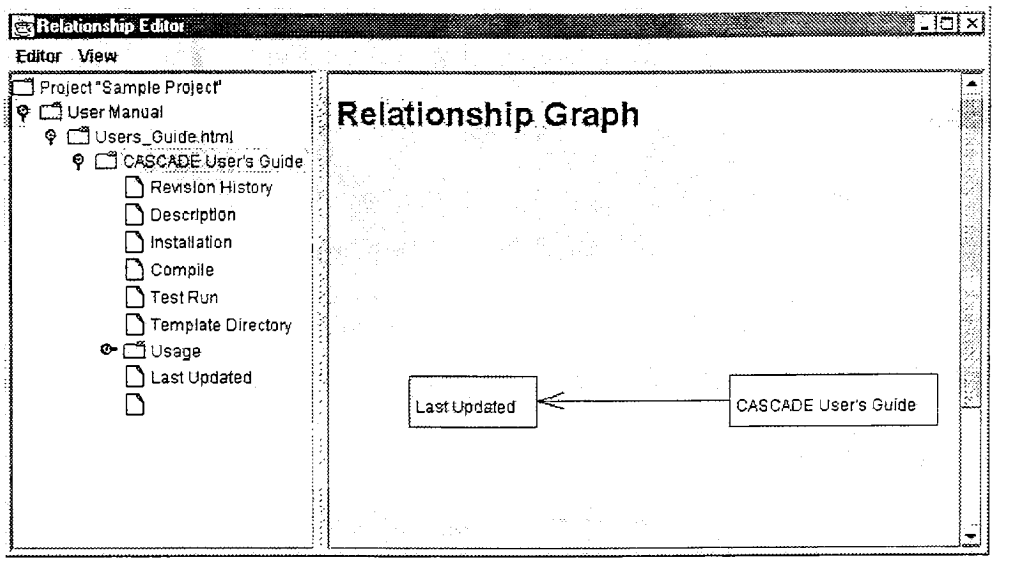
INDIRECT_SUBENTITY_ADDED -- A subentity has been added one or more levels beneath this node.

INDIRECT_SUBENTITY_DELETED -- A subentity has been deleted one or more levels beneath this node.

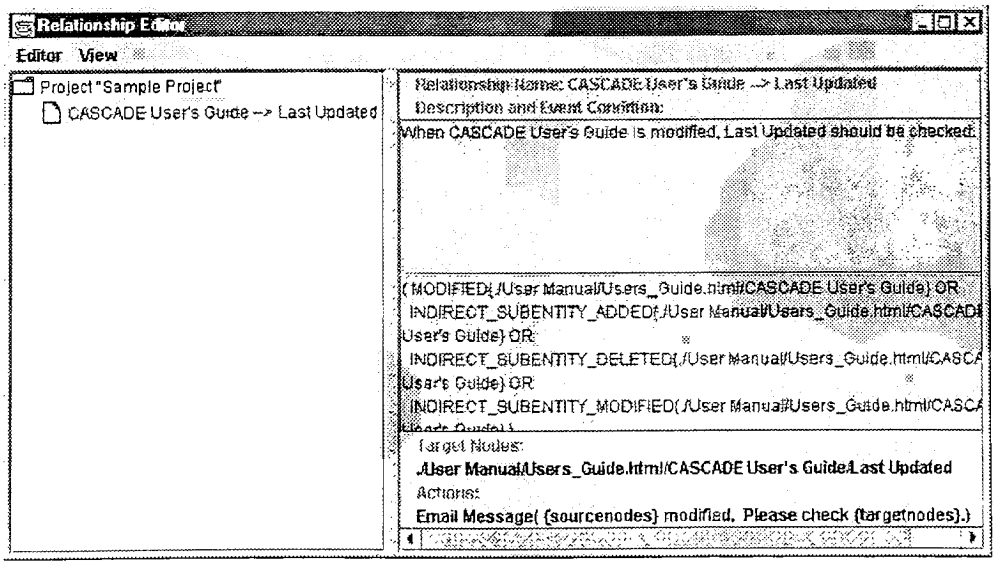
INDIRECT_SUBENTITY_MODIFIED -- The contents of a subentity one or more levels beneath this node have been modified.

You can use the Check Syntax button to check if the event condition string you entered is syntactically valid. For target nodes, specify all target nodes that are affected by this relationship. These are listed so that they can appear in email messages and so that relationship graphs (described later) are drawn correctly. Right-click in this field for shortcuts. Finally, specify any number of actions to be performed in any order when this event holds true. By default a single action is performed, which is to send email to all developers with the given text. However, additional actions can be performed as well, such as displaying a message to the screen, sending additional email messages, or executing a system command.

When you click OK, the relationship will be created. You can verify that the relationship is as expected by clicking on the target node and then selecting "Graph" from the View menu. This will turn the right pane into graph mode, and CASCADE will display a graph of all currently selected nodes and the entities to which they are related. Currently selected nodes will appear in white, and target nodes that are not currently selected will appear in gray. Example:



If you wish to edit or delete existing relationships, simply select Relationships from the View menu and right-click on the desired relationship and click either "Edit Relationship..." or "Remove Relationship..." Unfortunately, there is no simple way to remove more than one relationship at a time. However, being that all relationships are stored in the project XML file, it would not be difficult to do this using a text editor or through a special utility created by the user. You can also view information about each relationship in this view. For example:



In the future, when new relationships come up, just launch the relationship editor and add, modify, or delete relationships.

Subentity Extractor extensions have the ability to create, modify, and delete relationships as well. For example, the Java Extractor that comes standard with CASCADE automatically creates relationships between entities by looking at special `@cascade` Javadoc tags. To use these tags, follow these instructions:

Create nicknames in your alias file (usually `alias.txt`) to make it shorter to type the full formal names of nodes in the system. For example, an alias file for this project may contain the line:

```
userguide {./User Manual/Users_Guide.html/CASCADE User's
Guide}
```

Next, embed a `@cascade` tag whenever you want to link the current class, attribute or method to the specified entity. For example:

```
/**
 * Automatically installs the proper libraries for CASCADE.
 *
 * @cascade {userguide}/Installation
 */
public void installFiles() {
  // ...
}
```

In this particular example, the string `"{userguide}/Installation"` expands to `"./User Manual/Users_Guide.html/CASCADE User's Guide/Installation"`. For more examples of this, see the source code for CASCADE itself.

Finally, run the modification analyzer (discussed in the next section). The new relationships will automatically be constructed. These relationships will be bi-directional in nature. That is, if the code is modified, CASCADE will recommend the documentation be checked, and if the documentation is modified, CASCADE will recommend the code will be modified.

Modification Analysis

Once the project has been created and all relationships have been established, the modification analyzer must be integrated into your software development lifecycle. To do this, simply ensure that whenever all changes to a given file

have been made that the modification analyzer is launched, using the significant entity name as a parameter. This is most easily done by integrating it with the revision control checkin process (a must-have for any serious development team). If this is not present, the team can simply learn to run the modification analyzer manually at periodic intervals (perhaps a system script can remind developers every so often). Alternatively, a program can be written that automatically launches the analyzer whenever the timestamp on a file is updated. In any event, there are many ways to work the modification analyzer into the development lifecycle. To actually launch the modification analyzer, simply execute:

```
java
edu.rit.cs.cascade.modification.ModificationAnalyzer
```

This will provide you with a list of options. In general, you should at least specify the name of the project. Optionally, you can specify the names of the entities to analyze. If none are specified, all entities are analyzed. Once the analysis is complete, email will be sent to developers recommending changes that need to be done. See the next section for how to interpret these emails. A sample output session would look as follows:

```
C:\CASCADE\Projects\testjava
edu.rit.cs.cascade.modification.ModificationAnalyzer
Sample.xml
{User Manual}
Modification Analysis modified.
Sending Mail (CASCADE User's Guide -- Last Updated) from
john.doe@server.com to devteam@server.com.
```

This indicates that while the Modification Analyzer was analyzing the significant entity called {User Manual}, the subentity called "Modification Analysis" was modified. The relationship analyzer searched through all relationships in the system and determined that "CASCADE User's Guide -- Last Updated" should be fired. Therefore, it followed the only registered action, which was to send email from john.doe@server.com to devteam@server.com with the details of the relationship. The email received by devteam@server.com would look as follows:

```
Subject:
        CASCADE: CASCADE User's Guide -- Last Updated
   Date:
        Sun, 08 Aug 1999 01:31:20 -0400 (EDT)
   From:
        john.doe@server.com
     To:
        devteam@server.com
```

```
Relationship "CASCADE User's Guide -- Last Updated"
detected at Sun Aug 08 01:31:19 EDT 1999.

When CASCADE User's Guide is modified, Last Updated
should be checked.
-------------------------------------------------
Event Condition:
( MODIFIED{./User Manual/Users_Guide.html/CASCADE User's
Guide} OR
  INDIRECT_SUBENTITY_ADDED{./User
Manual/Users_Guide.html/CASCADE User's Guide} OR
  INDIRECT_SUBENTITY_DELETED{./User
Manual/Users_Guide.html/CASCADE User's Guide} OR
  INDIRECT_SUBENTITY_MODIFIED{./User
Manual/Users_Guide.html/CASCADE User's Guide}
)
AND NOT MODIFIED{./User Manual/Users_Guide.html/CASCADE
User's Guide/Last Updated}
-------------------------------------------------
Source Nodes:
  ./User Manual/Users_Guide.html/CASCADE User's Guide
  ./User Manual/Users_Guide.html/CASCADE User's
Guide/Last Updated
-------------------------------------------------
Target Nodes:
  ./User Manual/Users_Guide.html/CASCADE User's
Guide/Last Updated
-------------------------------------------------
```

In addition, a log file is written containing the exact details of what was modified. For this sample, the log file looked as follows:

```
Sun Aug 08 01:31:17 EDT 1999 {./User
Manual/Users_Guide.html/CASCADE User's
Guide/Modification Analysis} modified
Sun Aug 08 01:31:18 EDT 1999 {./User
Manual/Users_Guide.html/CASCADE User's Guide}
subentityModified indirectSubentityModified
Sun Aug 08 01:31:18 EDT 1999 {./User
Manual/Users_Guide.html} indirectSubentityModified
Sun Aug 08 01:31:18 EDT 1999 {./User Manual}
indirectSubentityModified
```

As can be seen, this is enough information to write any number of utilities to perform automated tasks to aid in the synchronization process if need be.

Appendix E

SOURCE CODE FOR FLAGSEXTRACTOR

This appendix contains the source code to the FlagsExtcator class, a sample extension to the Subentity Extractor, created for the experiment in Chapter 7.

```
package edu.rit.cs.cascade.extractor.extension;

import edu.rit.cs.cascade.common.*;
import java.io.*;
import java.util.*;

/**
 * Subentity Extractor extension that processes a text file containing
 * a list of flags to be set.
 *
 * The root of the tree is the significant entity.  Each flag is a
 * subentity, and subflags are subentities of those subentities.
 * The existence of a subflag indicates that it is set, and all parent
 * flags are set as well.
 *
 * @verson $Id$
 * @author Mark Roth
 *
 * Revision History
 *    $Log$
 */
public class FlagsExtractor extends edu.rit.cs.cascade.extractor.Extension {

  /**
   * Performs the extraction
   */
  public void extract() {
    SignificantEntity se = getSignificantEntity();
    List files = se.getFiles();
    int i;

    // Read in all relevant files.
    for( i = 0; i < files.size(); i++ ) {
      File file = new File( (String)files.get( i ) );
      try {
        processFile( file );
      }
      catch( IOException e ) {
        // error reading this file.  Ignore it, but let the user know.
        System.err.println( "Error processing " + file + ": " +
          e.getMessage() );
      }
    }
  }
```

```
  /**
   * Processes a single file full of flags
   *
   * @param file The file containing the list of flags
   * @exception IOException Thrown if there was an error reading the file.
   */
  private void processFile( File file ) throws IOException {
    BufferedReader in = new BufferedReader( new FileReader( file ) );
    String flagName;

    Node root = getTreeRoot();

    while( (flagName = in.readLine()) != null ) {
      // Get next flag from file
      StringTokenizer st = new StringTokenizer( flagName, "/" );

      Node currentNode   root, nextNode = null;

      // Break down name using / as separator.
      while( st.hasMoreTokens() ) {
        String nodeName = st.nextToken();
        nextNode = currentNode.getNode( nodeName );

        // If the node was not found, create it:
        if( nextNode == null ) {
          SubentityNode seNode = new SubentityNode(
            root, nodeName, "", "true" );
          currentNode.addNode( seNode );
          nextNode = seNode;
        }
        currentNode = nextNode;
      }
    }

    in.close();
  }


  /**
   * Test compatibility   Only files by the name of flags.txt are valid.
   */
  public boolean isCompatible( File file ) {
    boolean result = true;

    if( !file.getAbsolutePath().endsWith( "flags.txt" ) ) {
      result = false;
    }

    return result;
  }
}
```

BIBLIOGRAPHY

[1] Arai, T., Aust, D., Hudson, S. E.; PaperLink: a technique for hyperlinking from real paper to electronic content. In: CHI'97. Conference proceedings on Human factors in computing systems, 1997; pp. 327-334.

[2] Bra, P. D., Houben, G. Kornatzky, Y.; An extensible data model for hyperdocuments. In: ECHT '92. Proceedings of the ACM conference on Hypertext, 1992; pp. 222-231.

[3] Cunningham, K. M.; OLH: an on-line help facility for managing multiple document types in their native formats in a distributed environment. In: SIGDOC '91. Proceedings of the conference on 1991 ACM ninth annual international conference on systems documentation, 1991; pp.12-20.

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J.; *Design Patterns: Elements of Reusable Object-Oriented Software*; Reading, Mass: Addison-Wesley, 1995.

[5] Knuth, D. E.; *Literate Programming*, CSLI, 1992.

[6] Laurent, S.; *XML: A primer*, Foster City, CA: MIS Press, 1998.

[7] Lutz, M. J.; Rochester Institute of Technology, Dept. of Computer Science: Lecture notes from Software Engineering 1, 1995.

[8] Matthews, S., Grove, C.; Applying object-oriented concepts to documentation. In: SIGDOC '92. Proceedings of the 10th annual international conference on Systems documentation, 1992; pp. 265-271.

[9] Novick, D. G., Juillet, J.; Documentation Integrity for Safety-Critical Applications: The COHERE Project. In: SIGDOC '98. Proceedings on the sixteenth annual international conference on Computer documentation, 1998; pp. 51-57.

[10] NTSB; Docket CHI96LA011; 1996.

[11] NTSB; Docket LAX94LA137; 1994.

[12] Pesch, R. H., GNU info: a decade of hypertext experience. SIGDOC '93. Proceedings of the 11th annual international conference on Systems documentation, 1993; pp.233-240.

[13] Rawal, D.; E-mail to the author. Landmark Graphics Corporation; 27 Jul 1999.

[14] Rettig, M.; Nobody reads documentation. In: Communications of the ACM, Jul 1991, *34*(7); pp.19-24.

[15] Tilley, S. R., Müller, H. A., Orgun, M. A.; Documenting software systems with views. In: SIGDOC '92. Proceedings of the 10th annual international conference on Systems documentation, 1992; pp. 211-219.

[16] Tilley, S., Müller, H. A., INFO: a simple document annotation facility. In: SIGDOC '91. Proceedings on the conference on 1991 ACM ninth annual international conference on systems documentation, 1992; pp. 30-36.

Online Resources

Java JDK 1.2: http://java.sun.com/products/jdk/1.2/index.html. Visited: 7 Aug 1999.

Java Project X, Technology Release 2. http://java.sun.com/xml. Visited: 7 Aug 1999.

ANTLR 2.6.1. http://www.ANTLR.org/. Visited: 7 Aug 1999.

JavaMail 1.1. http://java.sun.com/products/javamail/index.html. Visited: 7 Aug 1999.

JavaBeans Activation Framework, 1.0a. http://java.sun.com/beans/glasgow/jaf.html. Visited: 7 Aug 1999.

Trademarks and Copyrights

INDEX