

Operational Concept for a Code Relationships Analyzer

**CSE681 SWMAA
Spring 2010**

**Bradley Randolph
SUID: 10053 - 5908**

Executive Summary

As computer applications increase in complexity the need to control dependencies between the composing parts also increases. Software developers, project managers and software testers can benefit from a system that analyzes the dependency relationships between software components. The static code relationships analyzer will identify the dependencies between a set of source code files, identifying when data used in one file may have its definition specified in another file. The output from this analyzer will be graphical, summarizing the analysis to assist in project planning and decision making as well as a detailed textual report to assist in making changes and improvements to the files.

Introduction

In sociology the relationship and interaction among two or more people is the basis of analysis for social scientists. Enquiries into the nature of social relations help us to understand the nature of communities and societies. In the field of mathematics there is the somewhat humorous idea of a mathematician's "Erdős number," which is similar to the popularly used concept of "degrees of separation." In the area of software development the relationship between files within a group of files can also be discovered.

There is an insidious problem with any computer application that is well liked and is extended to meet additional requirements. In his book UML Distilled, Martin Fowler describes the problem this way:

As computer systems grow, you have to worry more and more about controlling dependencies. If dependencies get out of control, each change to the system has a wide ripple effect as more and more things have to change. The bigger the ripple, the harder it is to change anything. [Fowler 47]

There is certainly a need to understand file dependency in order to control it.

In software engineering there are testing and maintenance concerns for source code. Integrating individual source code files with other files to create new components creates difficulties when testing. Localizing the file (or files) where an error occurs requires additional analysis of the interaction between the files in the newly formed component. It can be very insightful to see the relationship of one file to other files within the system. In addition, when a change is required it is very useful to know the extent of the impact the correction has on the system. Also, after the system is delivered, when repairs, additions or improvements are needed it is again useful to know how much of the system will need to be verified.

This document describes a computer program architecture for quantifying and analyzing the static relationships of source code files in terms of the data types used and the dependencies between files. This kind of static analysis is a fundamental step towards understanding the interdependency of the files and the extent of the data types within a sub-system or a complete system.

This software system will be responsible to read source code files, understand enough of the syntax of the coding language to detect the data types as it reads each file and build a

metadata representation of the data types found in the file. The software will also be responsible for analyzing the collection of metadata in order to represent the relationships in a simple human-readable form.

Uses

The static code relationships analyzer will read a set of source code files and deliver descriptions of the data type and name space dependencies between all files in the file set. The analyzer will also provide an optional description of all the data types used within a file but not defined elsewhere in the file set.

This analyzer may be used by software developers, testers or project managers. The diagram in Figure 1 shows the usage context and the users of the analyzer system.

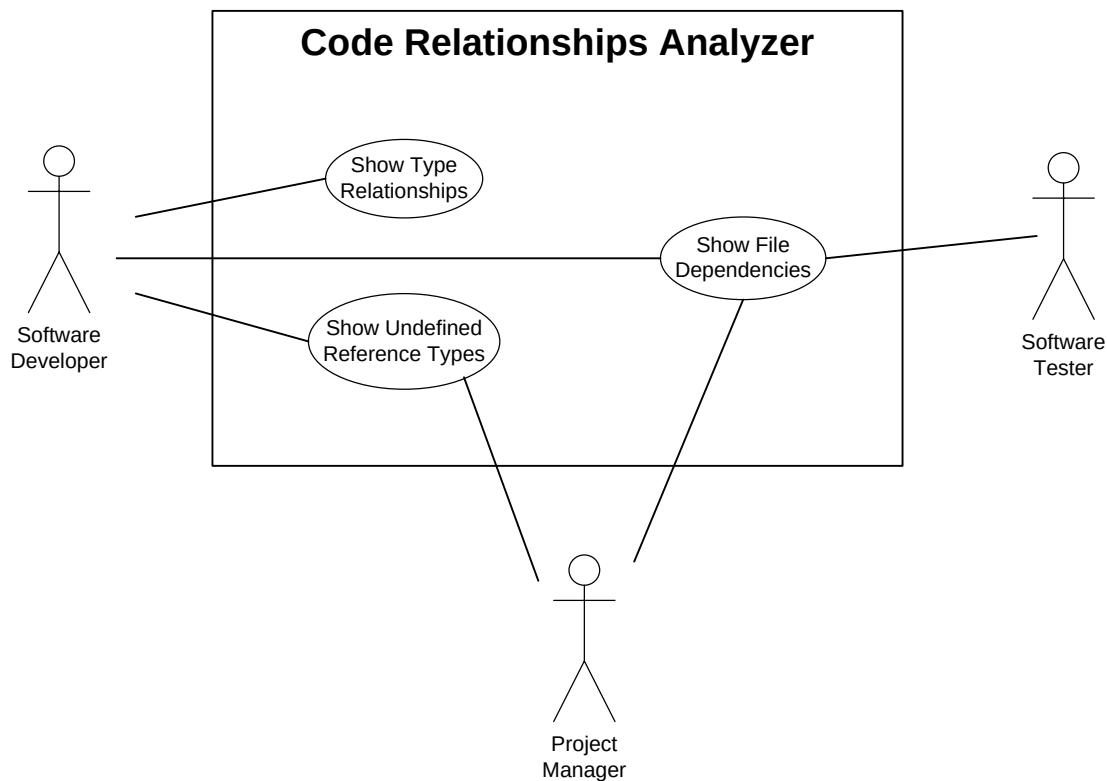


Figure 1 - Users and Uses of the Analyzer System

The software developer may use the analyzer for optimizing the implementation of their design, discovering dependencies that may lead to problems in the future. The developer may also use the analyzer for planning the unit testing of their implementation by discovering other files which depend on the one file they are testing. The optional undefined references output can help the developer to discover data types left to be implemented.

The software tester may use the analyzer for test planning to discover the extent a changed file has on the remaining set of files. This information can be useful to establish the scope or boundaries of the system testing.

The project manager may use the analyzer for project planning to discover the extent a proposed change has on the set of files and may help to determine the level of effort to implement the change. The manager may also use the optional undefined references output to determine how much of the system needs to be re-implemented.

In all cases the users will need the output to be easily interpreted. The software developer may require greater detail in the output than a tester or manger. Also, the developer may require the analyzer to be easily integrated into their development environment. The software tester will require an output artifact that can be incorporated into a test document. The project manager will likely require a summarization of file dependencies or a summary of the optional undefined references.

The analyzer may be a stand-alone application with a unique graphical user interface. This output may provide vertices showing the relationships between the files.

The analyzer may also be a plug-in module or an add-on tool for integrated development environments, like Visual Studio™ or Eclipse.

The analyzer may also have a command-line console interface to make it possible to incorporate into an automated build system. This would allow it to become part of the build process and providing formatted output that can be archived with a specific build.

It may be desirable to implement the design in several DLLs, where one DLL will implement the analyzer's processing rules and other DLLs will provide the required services within the design. The front-end parser could be a DLL that only understands the syntax of one source code language, while another language is implemented within another DLL.

Activities

The code relationships analyzer could be designed as a two pass system.

The first pass will compose a list of fully formed path names from the file and path specification provided by the user. This file name list is used by the file reader to gain access to the contents of each file in order to search for the data types, base types, dynamically allocated object types and name spaces within each file. As the artifacts are found they are placed into a cross reference table with the name of the file where it is defined.

In the second pass, after the cross references have been completed, the type analyses occur to find the dependencies between the files. This pass includes identifying unfound references.

After the files have been analyzed the output renderer interprets the data from the analyzers to create either a graphical or formatted text report describing the file dependencies.

Partitions

The activities of the code relationships analyzer are described in this section in the context of the packages in Figure 2. In addition to the activities already discussed (like the file reader, the code analyzers and the output renderer) there is also a need for a controlling executive and user interface packages. These are described fully in the next section.

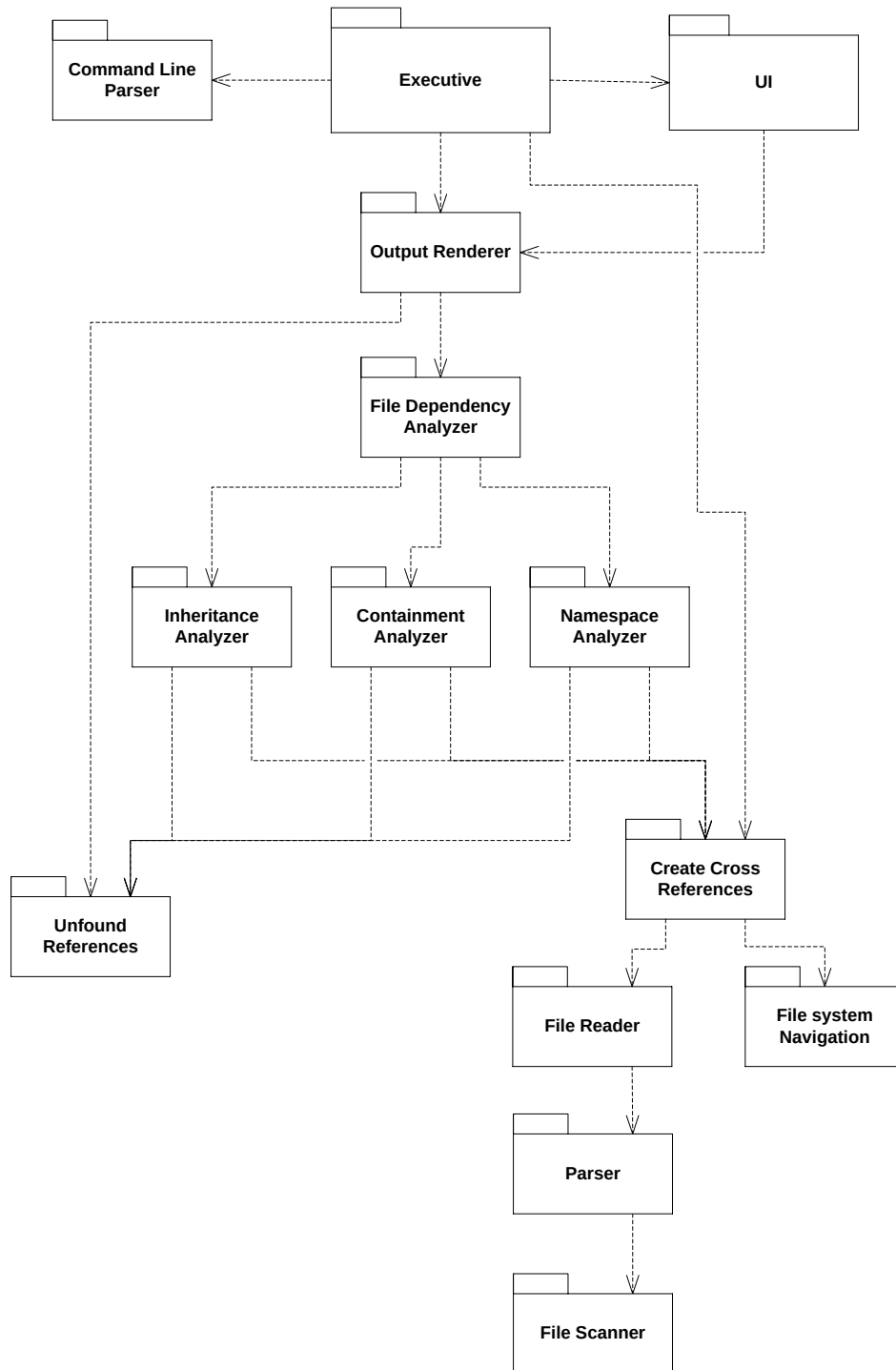


Figure 2 - Code Analyzer Package Diagram

Package Descriptions

This section describes each package in the code relationships analyzer which includes a statement of their responsibilities and their interaction with other packages.

Executive

The executive package is the main controller of the system, handling object creation and implementing the business rules of the system. The executive initiates collecting the list of fully qualified file names to scan, requests the Cross References package to scan the source code files and then initiates the processing by the analyzers. When the analyzers have completed their work, the executive initiates the output rendering and requests the UI to display the results.

Command Line Parser

The command line parser receives a pathname and filename specification string and divides it up into a folder specification and a filename specification. The executive then sends these to the Cross References package to specify all the folders to traverse and all the files to access.

User Interface (UI)

The user interface (UI) package is the graphical or console command line means of communicating with the user of the analyzer. The UI provides a means to collect the file and path name patterns and collect the option to analyze for unfound references. When all the files have been analyzed the UI will provide a means to display the relationships between the files in the set as represented by the output renderer.

Output Renderer

The output renderer is responsible for representing the interpretation performed by the File Dependency analyzer and the Unfound References package. The output may be graphical charts or formatted text. The optional unfound references can be a simple list or chart of the undefined types used in the files.

File Dependency Analyzer

The file dependency analyzer is responsible for interpreting the available data from all the analyzers. The interpretation will show “inheritance”, “containment” and “using” relationships between types defined in the files. The interpretation will also show dependency relationships between the files.

Inheritance Analyzer

The inheritance analyzer will scan the cross reference data for class types that may be defined in other files. For each file in the set, as a base class in each file is discovered, the location of the definition for that class is searched for in the cross references. If a definition is found in another file a dependency link is recorded for these files.

Containment Analyzer

The containment analyzer will scan the cross reference data for dynamically allocated objects whose type may be defined in other files. For each file in the set, as a dynamically allocated object is discovered, the location of the definition for that data type is searched for in the cross references. If a definition is found in another file a dependency link is recorded for these files.

Namespace Analyzer

The namespace analyzer will scan the cross reference data for name space references to obtain the name space paths which may be defined in other files. For each file in the set, as a referenced namespace is discovered, the location of namespace definition is searched for in the cross references. If a namespace definition is found in another file a dependency link is recorded for these files.

Create Cross References

The cross references package composes a list of fully formed pathnames according to the file and path specifications. This list is used to access all the files for developing the cross reference data. As this package gains access to each file it will filter the expressions from the parser, collecting the data type names defined in each file. It will also filter for and collect all base class type names referred to in each file. It will also filter for and collect the data types of all dynamically allocated objects in each file. It will also filter for and collect all name spaces referred to in each file. This package will provide one or more tables that can be searched by the analyzers. Implementing hash tables should be considered to improve searching efficiency.

Unfound References

The unfound references package will collect the unfound data type references from the analyzers. This should be a simple mechanism for depositing the definitions in each file that cannot be located in the cross references package. The data will be held for use by the output renderer to optionally report the unfound references.

File Reader

The file reader package opens the specified file for reading and sends the contents of the file to the parser. The output from the parser is held in a queuing mechanism waiting to be read from the cross reference package. When the end of the file is reached a notification is raised indicating that the processing of the file has completed.

File system Navigation

The file system navigation package is used to compose the fully formed path name to the file being analyzed. This package will report when a file does not exist.

Parser

The parser package is the detection mechanism for finding the data types, base types, dynamically allocated object types and name spaces within a file. The detected items are

searched for in the semi-expressions provided by the file scanner package. This package is reused code from "Parser/Tokenizer". [Fawcett]

File Scanner

The file scanner package extracts token sequences from the file being read to provide a string called a semi-expression. A semi-expression is composed of the right amount of source code in order to make a decision about a language construct. This package is reused code from "Parser/SemiExpression". [Fawcett]

Critical Issues

There are a few areas that need to be addressed when designing and implementing the code relationships analyzer.

The first area of concern is file reading performance. As the number of analyzed files increases the capability of the file storage medium (whether a physical disk or a network drive) will determine how quickly the set of files can be analyzed. Modern operating systems typically provide data caching of the attached physical storage while a network drive may not. Implementing a read-ahead buffer is not likely to improve this performance concern.

A related area of concern when reading a large number of files is the memory size of the cross reference tables. For memory constrained systems, a large cross reference table will hamper performance as memory pages are swapped with virtual storage. This also may be unavoidable when compared to increasing program complexity.

The next area of concern is detecting false positive dependencies. When finding data type definitions in a perfect system the type would be fully qualified with a namespace. In order to keep program complexity low a data type may not need to be fully qualified. This would lead to the possibility of detecting a type in a file where it is not truly defined, which may be an acceptable side-effect.

Bibliography

- "Social relation" - Wikipedia, the free encyclopedia. Web. 25 Jan. 2010.
<http://en.wikipedia.org/wiki/Human_social_interaction>.
- "Information about the Erdős Number Project" Home - Oakland University. Web. 24 Jan. 2010. <<http://www.oakland.edu/enp/readme/>>.
- "Six degrees of separation" - Wikipedia, the free encyclopedia. Web. 24 Jan. 2010.
<http://en.wikipedia.org/wiki/Six_degrees_of_separation>.
- Sommerville, Ian. *Software Engineering*. 6th ed. Essex, England: Pearson Education Ltd, 2001. Print.
- Fowler, Martin. *UML Distilled*. 3rd ed. A Brief Guide to the Standard Object Modeling Language. Boston, MA: Addison-Wesley, 2004. Print.
- Fawcett, Jim. *Parser*. Computer software. Web.
<<http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/Parser/>>