

CSE 681-Software Modeling and Analysis

Project #1

CODE RELATIONSHIP ANALYZER-OCD

VERSION: 1.0

Arunkumar Manikantan (899234415)

Date: 01-28-2010

Instructor-Dr. Jim Fawcett

Contents

1. Executive Summary	4
2. Introduction	5
2.1 Objective and key idea	5
2.2 Obligations	5
2.3 Organizing Principles	5
3. Uses.....	5
3.1 Actors:.....	5
3.1.1 Developer.....	6
3.1.2 Project Leader/Manager	6
3.1.2 Test team.....	6
3.1.4 Quality Assurance Team	6
3.1.5 Maintenance team	6
3.1.6 Software Architect	6
3.1.7 Customer	6
3.2 Uses	6
3.2.1 Developer – Understand Structure and complexity, reusability, impact of changes, trouble shooting, estimation, documentation, user interface:	7
3.2.2 Project leader/Manager- Understanding potential, Estimation, risk analysis, Reporting, managing complexity, user interface	8
3.2.3 Quality Assurance Team – Estimate quality, reviewing test cases and plans	8
3.2.4 Test team – Writing test plans/cases comprehensively	9
3.2.5 Maintenance team – Trouble shooting and incorporating small changes	9
3.2.6 Software Architect – Restructure existing systems, architecture for new subsystems.....	9
3.2.7 Customer - Maintenance	9
3.3 Extension or changes to the tool.....	9
3.4 Use Cases	10
4. Application Activities.....	12
4.1 Getting input	12
4.2 Identify list of files.....	12
4.3 Detect Type Definitions - First Pass:	12
4.4 Detect Type Relationships and File dependencies – Second Pass:	12
4.5 Storing the output information:	12
4.6 Storing output persistently for future reference:	13
4.6.1 Excel sheet	13
4.6.2 XML.....	13
4.7 Storing error logs	13
5. Partitions-Modules and Classes.....	15
5.1 UI Module.....	15
5.2 File Manager Module	15
5.3 Error Handler module.....	15
5.4 Parser module.....	16
5.5 Semi-Expression evaluator module.....	16
5.6 Tokenizer module	16
5.7 Type Definition Storage module	17
5.8 Type Relationship and File dependency storage module.....	17
6. Critical Issues	20
6.1 Large number of files:	20
6.1.1 Displaying of information	20
6.1.2 Time taken/Performance.....	20
6.1.3 Memory.....	20
6.2 Addition/Removal of a file during processing:	20
6.3 Improper code structure violating syntax:	21
6.4 Tool Maintenance	21
6.5 Ignoring namespaces	21

7. Views	22
7.1 Obtaining Input and Analyzing files	22
7.2 Showing Output in Tabs.....	23
8. Events.....	26
9. Appendix	27
7.1 Proposed Excel file format for storing output information:	27
7.2 Proposed XML File format for storing output information:	27
10. References	29

1. Executive Summary

Code Relationship Analyzer serves as a highly valuable tool in the development process of a large software project. It helps to identify the dependencies between the source files and the static relationships between the types used in them. These dependencies help in understanding,

- The physical structure of the project.
- The impact of latent or critical changes in the project.
- The level of cohesion and coupling between files.
- The complexity of the project.

The tool is very useful in the following contexts.

- When a new developer wants to gain insight on the physical structure and complexity of the project.
- When there are some changes to be made in the project, the tool helps in identifying the files that would be impacted and the magnitude of impact could be determined based on the number of dependencies.
- To estimate time/cost for changes to be made in the project based on the number of dependencies.
- To help making decisions when critical changes are required.

The primary users of this tool will be developers, project leads/managers, test team, Quality assurance team, Maintenance team, Software Architect and the customer. The proposed interface facilitates the user with comprehensive details in proper formats to completely identify the dependencies that exist.

Some critical issues that need to be addressed are identified and solutions are discussed.

- When the number of files in the list whose dependency information need to be identified is large, it is hard to show all the information at a time to the appropriate user as per their needs.
- The tool needs to make two passes of the list of files to identify the dependency information. When the load is heavy, the tool might take significant time in responding to the user with appropriate information which would affect performance.
- Behavior during addition/removal of files.

Some other non critical issues are tool maintenance and ignoring of namespaces.

The tool will comprise of the following modules:

- Parser module – Parse each file to identify type definition and dependencies. Makes use of Tokenizer and Semi-Expression Evaluator modules.
- Tokenizer module – Identify tokens in a file stream or string stream.
- Semi-Expression Evaluator module – Specify some rules for forming expressions from a set of tokens.
- File Processor module – To identify all the input files and send it to the parser one by one.
- ErrorHandler module – To handle errors if any during opening of files or due to invalid path input.
- UI module -> for input and output – Get the list of files from user and display the dependency information.
- Type Definition Storage module – Store all the file definitions after they are identified.

- Type Relationship and File dependency Storage module – Store all the type relationship and file dependency information after they are identified.

This tool bolsters and complements successful implementation of a large project.

2. Introduction

It is really important at times during the development process of a project, to know the dependencies and its types that exist between files. This is essential when a person wants to gain insight into the physical structure of the project or assess a large code base. In most cases, the tasks in a project are related to each other and the dependencies between them drive the schedule for the project. This tool serves to identify the dependency information.

2.1 Objective and key idea

Its main objective is to determine the type relationships and file dependencies existing in a set of source code files. The key idea is implemented by opening the source code files, searching for type-like keywords, storing the type definitions in a data structure like Hashtable and finally analyzing their use within all the files in the set. Identifying dependencies help in identifying the true “**value-generating**” aspects of the project. The tool’s implementation involves 2 passes of the entire file set.

2.2 Obligations

The tool will be developed as an efficient and user-friendly one meeting all the stated requirements. The main obligations of the tool would be

- To display all file dependencies that exist in a set of source code files
- To display all type relationships that exist in a set of source code files
- To provide an efficient and user friendly interface for the user to accept inputs and present outputs.
- To present the output to the user in an efficient way that is easier to understand.

2.3 Organizing Principles

The Classification of rules for a Parser to define grammatical constructs which in turn invokes actions is an identified organizing principle for this architecture.

3. Uses

3.1 Actors:



There is no use of developing a tool without thinking of its actors. It is for them, the tool is being developed. They interact with the tool and gain valuable information out of it. The tools make lives of the actors easier by automating certain processes. The following actors can interact with this tool and obtain valuable information.

3.1.1 Developer

A Developer is a team member responsible for development of the project. The tool helps a developer mainly in understanding the physical structure and complexity of the project based on the dependencies that exist.

3.1.2 Project Leader/Manager

A person who leads or manages a team and is responsible for scheduling the project, estimate time/cost needed for the development, monitor the progress of the project and notify customers of change in schedules and estimated time. This tool can come in handy to analyze the impact and estimate accordingly when critical changes need to be done in the project.

3.1.2 Test team

A team responsible for writing test plans and test cases to thoroughly test the functionality of various modules in the project. The tool serves in writing proper test scenarios based on existing dependencies and understanding the order of testing various modules.

3.1.4 Quality Assurance Team

A team responsible for verifying the quality of the project that needs to be delivered to the customer. They can check if the test cases cover all the scenarios (both positive and negative) and are efficient by knowing dependencies between files.

3.1.5 Maintenance team

A team that is responsible for maintaining the project after its delivery. It is also possible that a project is completely maintained by the customer itself. A major responsibility of maintenance team is to incorporate small changes and fix latent errors if any and provide support to the project. Using this tool, trouble shooting becomes easier by knowing dependencies. Also as already mentioned earlier, analyzing impact of changes is facilitated by this tool for this team.

3.1.6 Software Architect

A Software Architect is a person responsible for describing the architecture of the system and explains the concepts which bind the development team together. This tool can help a software architect in a scenario where the architect has to restructure an existing system or work on a new subsystem of an existing system.

3.1.7 Customer

The potential buyer of the project after it is developed and tested. The customer can understand the structure and complexity of the system using this tool. This helps in maintaining the project and analyzing impact of any future changes locally.

3.2 Uses

After describing the actors, the next step is to brief out how the actors interact with the tool how it serves them, their expectations and needs out of it and the behavior of tool based on its actors. All these are described below.

3.2.1 Developer – Understand Structure and complexity, reusability, impact of changes, trouble shooting, estimation, documentation, user interface:

Understand structure and complexity:

When a new developer joins the team, he/she will find it difficult to understand the project's physical structure and complexity. The tool facilitates in understanding both by listing out the dependencies between files and types used. This tool helps in gaining an insight on the structure and proceeding with further development.

Impact of changes

When there are code changes to be made, before the actual implementation, it is very essential to analyze the impact of the change on other code parts. Dependency Resolution is essential for this and the developers usually do dependency resolution by hand. More dependency, when identified between files, degrades flexibility. This tool automates the process of dependency resolution. This enables the developers to know which files need to be focused and may need change due to the new change that needs to be incorporated. In short, this tool would be used by the developers to "restructure and re-engineer" existing software.

Estimation

It is the developer's responsibility to give an estimation of time to the manager that would be required to incorporate a new change into the project. This estimation obviously depends on the magnitude of impact that the change would make and the number of files that would probably change. Using the tool, the number of files that might need a change can be found out, using which an estimate of time can be reported to the concerned manager.

Trouble shooting

Trouble shooting is the process of finding the cause of a bug or an error that occurs in the project. It is not enough to find the immediate cause of the error but it is necessary to find the root cause of it. This is essential as there is a lot of dependency that exists in the underlying code. Hence this tool can save a lot of time during trouble shooting.

Reusability

Today there is greater focus on reusing existing components. The developers can also understand which files are reusable by knowing the dependencies. Circular file dependencies can have a negative effect on reusability. Less number of dependencies that exist between files, more reusable they are and vice-versa. The tool as a whole helps in understanding the origin of dependencies and identifying reusable components.

Documentation of Dependencies

It is the responsibility of every developer to generate documentation for each module they code which provides complete information about the functionality of the module and its technical details to a new person/developer. In such cases it is absolutely essential to document critical dependencies. If the critical dependencies are not included in the description, then there is a big risk, no matter how good the schedule has been framed.

User interface:

The developer is provided with a user-friendly GUI wherein the developer can feed in the location of files as input. After processing them, the tool displays the number of files parsed, the number of file dependencies, the number of class relationships in each file, and the details of those dependencies.

3.2.2 Project leader/Manager- Understanding potential, Estimation, risk analysis, Reporting, managing complexity, user interface

Understanding developer's potential

The Team leader or Manager has the responsibilities of monitoring the progress of the project, the capability or potential of the developer in implementing the project within the estimated time. This tool helps in their work by supplying enough dependency information using which they can know how well the developer has programmed so as to ensure that there are fewer dependencies and also there is more scope for reusable files.

Estimation and Risk analysis

Another primary responsibility of the manager is to give an estimate of time to the customer for incorporating a new change in the project. This estimate could easily be given by using the tool. Also they can cross check with the estimation provided by the developer and ensure that it doesn't have much variation. This makes things easier during critical changes in the project. Besides knowing the dependencies, it is part of their job to do a risk analysis on the change enforced. If the dependencies are complex, more the risk is and vice-versa. This tool saves their time in doing risk analysis and estimation.

Reporting to customer

The manager can use this tool to convince the customer in case a particular change takes more time than expected. He can show them the dependency information and explain the possible impact based on the number of dependencies.

Managing complexity

The manager should also ensure that the project is not so complex and is in such a way that any new person should be able to understand the system in a short span of time. This complexity depends on the dependencies that exist between files. Less number of dependencies results in a less complex system and hence this tool can facilitate in understanding the complexity and taking appropriate action in case there is more complexity than expected.

User interface

The manager is provided with a GUI which is user-friendly. He/she provides the location of source files as input and the tool processes them and displays the dependency information by default at a higher level. This is because a manager needs only higher level details most of the time and they are not interested in knowing the lowest level of detail. Even then, the tool is flexible enough so that the leads/managers can view the information at a lower level.

3.2.3 Quality Assurance Team – Estimate quality, reviewing test cases and plans

Estimate quality

The Quality Assurance Team is responsible for ensuring of the quality of the final deliverable project. This tool can come in handy for them during this process. By checking the number of dependencies, the team can have an estimate of the quality. This means when there are more dependencies, the more complex the project is and hence aggressive testing is needed to ensure quality. This makes them more cautious in reviewing the test plans and test cases and make sure that all the possible scenarios are covered.

Reviewing test cases

Another advantage provided by this tool is that for a particular new change, the team can know which files to be reviewed based on the dependency. It is enough to review the changed file and its dependent files. Without this tool, the team would have a hard time in

analyzing the impact and identifying files that would be impacted and hence needs to be reviewed.

3.2.4 Test team – Writing test plans/cases comprehensively

Test team can use this tool before writing test plans and test cases for the project or individual modules. In case dependencies are more test cases are needed to cover all possible scenarios. Extensive testing would be needed keeping in mind all the dependent files. The test data should be prepared so as to cover all cases. After looking at the dependency information, testing should be done in such a way that a file with no dependences should be tested first. Then all the files which depend on this tested code should be tested. Hence to make sure that comprehensive testing is done, this tool helps and saves a lot of time for the team by providing enough dependency information.

3.2.5 Maintenance team – Trouble shooting and incorporating small changes

This team is responsible for maintaining the project once it is delivered to the customer.

Some of the important tasks that are part of maintenance are:

- To troubleshoot in case of errors/to fix latent errors.
- To provide support for the project.
- To incorporate small changes when required.

Both troubleshooting and incorporation of small changes involve analyzing the dependency between files and hence this tool serves that purpose. Hence this tool acts as a great time-saver during these times.

3.2.6 Software Architect – Restructure existing systems, architecture for new subsystems

One of the important jobs of software architects is to restructure an existing system due to some reasons or construct the architecture of a subsystem of an existing system. For both these jobs, physical structure of the existing system needs to be well understood and this is made easier by this tool. By knowing its structure and complexity, the architect can save a lot of time in designing a new architecture.

3.2.7 Customer - Maintenance

Once receiving the project that is being delivered, the customer has to maintain it for which he should be able to understand the structure of the project. Also for making some small changes after delivery, it should be possible for the customer to know the impact of the changes. This tool would support the customer in this scenario and makes his life easier by providing all the information that he needs to analyze the impact and to identify the context of the change. Also the customer can easily get to know the files which could be reused. This information would be pretty useful for their future projects and would add more value.

3.3 Extension or changes to the tool

Firstly, the tool could be extended to show the dependency information in the form of graphs. This could be helpful for the Program managers or customers who would want to know the dependency or relationship information at a very higher level. At many points, their focus is on the number than on specific files. In such cases showing a graph with files and its dependencies facilitate their task.

Secondly, the tool can be extended to show information about function calls that take place across files. This is additional information besides type relationships that may be useful for the user in some contexts.

3.4 Use Cases

Use cases represent the interaction that occurs between the user and the system. They describe the system's behavior from the user's perspective and are a collection of events. The following use cases have been identified for this tool and have also been depicted in a use case diagram.

Use case 1: Specify the location of files as input

Use case 2: Run the analyzer

Use case 3: View error messages/warning messages

Use case 4: View File Dependency Information

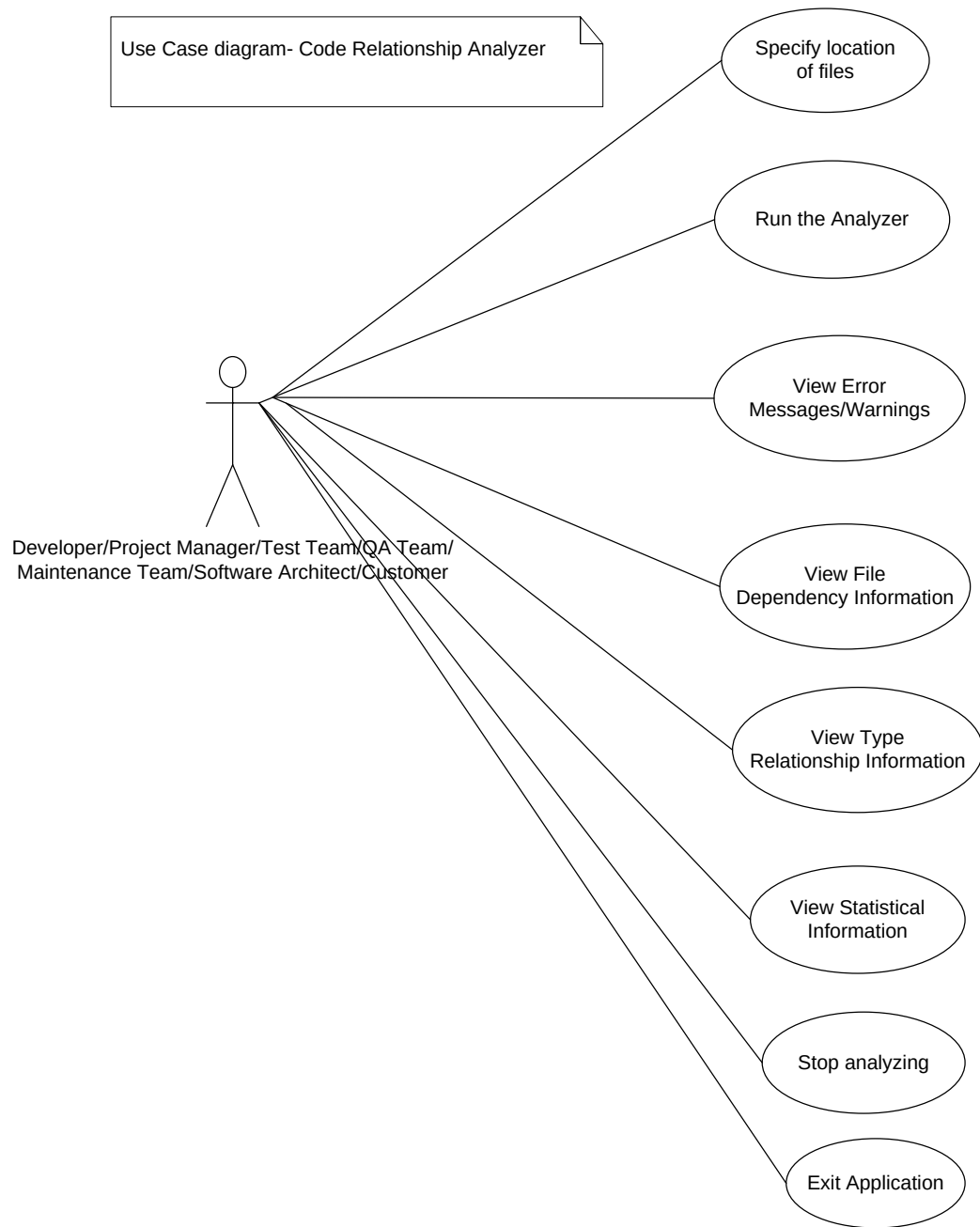
Use case 5: View Type dependency Information

Use case 6: View Statistical Information

Use case 7: Stop Analyzing

Use case 8: Exit application

Each of the above use cases can be considered as events that take place while the user interacts with the tool. The sequence of events is better represented in an Event-trace diagram/Sequence diagram in the Events section.



The Use case diagram above describes the actors involved in interacting with the system and the events that occur during the interaction. Each event triggers some action that involves some processing and then responds to the user with appropriate information. From this diagram, the primary actors and the functionality of the system/tool as a whole are conveyed in the form of possible events that occur.

The Views of the proposed User interface are discussed in the “Views” section for the different use cases.

4. Application Activities

The tool requires 2 passes over the set of source files to identify the dependency information. The major tasks that this tool would perform would be in a sequence as described below.

4.1 Getting input

The input is obtained from the user in an interactive way. The input will be a location of the set of source files to be analyzed.

4.2 Identify list of files

The input location is then accessed and the list of files of the pattern "*.cs" is identified by recursively searching through all the subdirectories.

4.3 Detect Type Definitions - First Pass:

Now in the first pass, each file is parsed in order to identify the type definitions inside each file. The possible candidates for types are "class, struct, interface, enum and delegate". There is a possibility that a same type name is defined in different namespaces and the namespaces could be nested. This tool "ignores namespaces" for the moment. This may lead to some false positives but that is a rare occurrence. The type definitions are identified by opening the source files, searching for type-like keyword by identifying individual tokens in the file stream. Once the types are identified they are stored in a structure like Hashtable where the key will be the type name and the value will be an Object containing the two information-actual type (class/interface/struct etc.,) and the file name in which it is defined, as attributes.

Ex., class Action {...} defined in File Action.cs

The Hashtable contains an entry as follows:

Key=>Action

Value=> object with (type=class and filename=Action.cs) as attributes.

4.4 Detect Type Relationships and File dependencies – Second Pass:

Now the second step is to identify the type relationship and file dependency information. In the second pass the list of files are parsed again to analyze if the files are using other types defined in other files. For this operation, the Hashtable is made use of and the type dependencies and in turn the dependent files are stored in another data structure convenient for display. This step also involves in identifying how the types are related-inheritance, composition, and aggregation or uses relationship. Some possible data structures that could be used are discussed in the next section.

4.5 Storing the output information:

The output dependency information is stored in the above discussed data structures which can then be retrieved to be displayed to the user.

The output information must contain the following:

- 1) Number of files that are analyzed
- 2) Number of file dependencies
- 3) Number of inheritance relationships
- 4) Number of Composition relationships
- 5) Number of Aggregation relationships
- 6) Number of Using relationships

- 7) All the relationships associated with each type.
- 8) File level dependencies.

The number of relationships is essential to be stored so as to analyze the magnitude of impact when changes are to be made.

Some possible data structures that could be used to store and retrieve data are as follows:

- 1) An ArrayList of File Dependency objects for each source file which contain the information like owner file name and the list of dependent file names as attributes – This stores File dependency information.

- 2) An ArrayList of Type Dependency objects for each type which contain information like type name, list of inheriting type names, list of composed type names, list of aggregated type names and list of used-class type names as attributes – This stores Type Dependency information.

Besides these, the number of files that are analyzed, number of file dependencies and number of each relationship type are maintained somewhere in memory.

Using these two lists and the above statistical data, all the needed information could be retrieved and displayed as appropriate to the user.

4.6 Storing output persistently for future reference:

The output information should be stored in the file system persistently so that the users can view the generated dependency information at a later point of time without the need to run the tool again. This could save a lot of time and can also be used for any other purpose. The reports can be sent to customer along with status reports and other documentation to make the customer feel comfortable with the project. This could add a lot of value to the project.

The output information can be stored in various formats but two formats are suggested here in this document. These reports can be tailored to focus on specific metrics that the user cares about.

4.6.1 Excel sheet

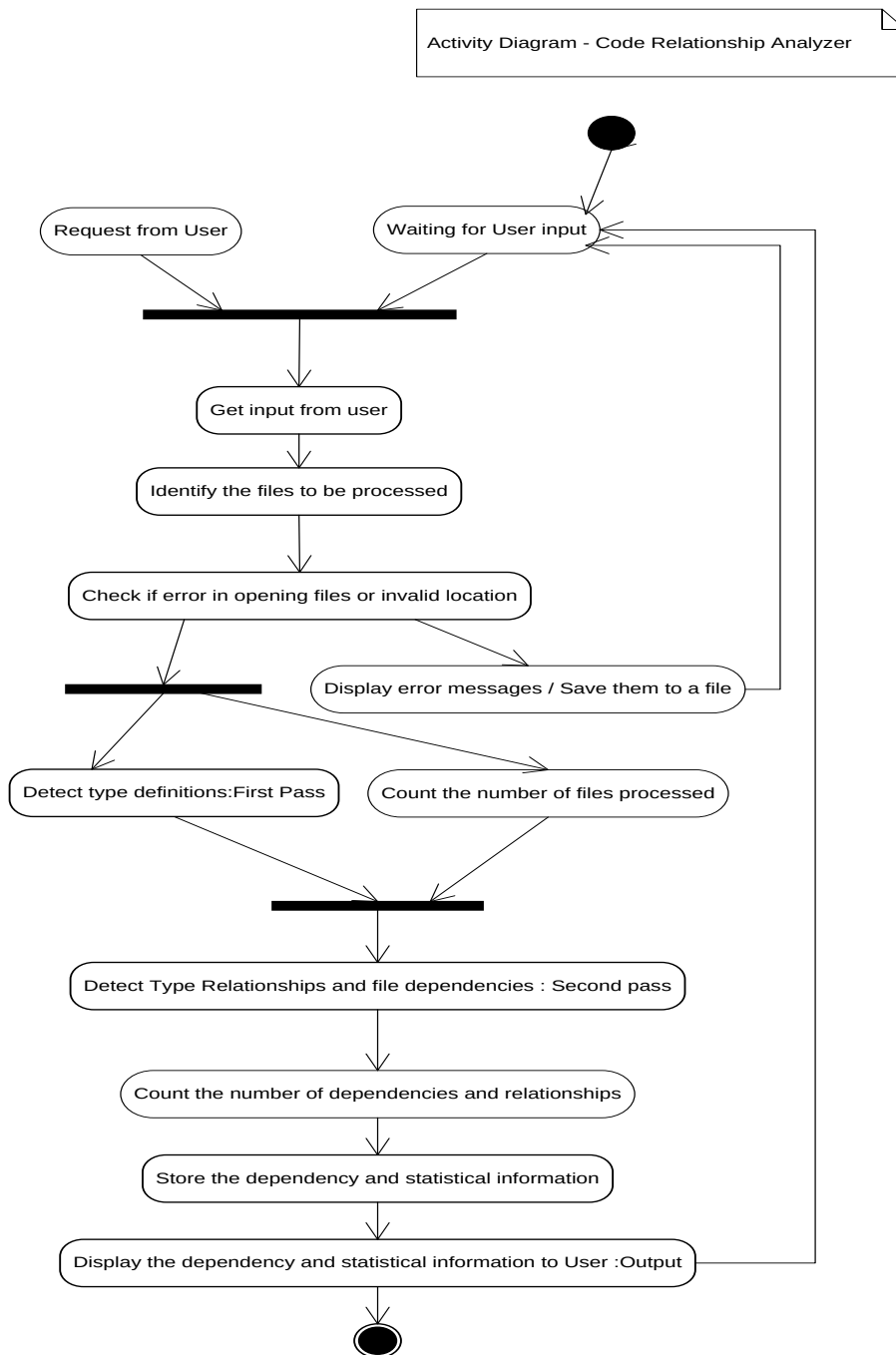
The output dependency information can be stored in an Excel sheet which is one of the most widely used documents for storing and generating reports across organizations. The excel file should contain the dependency information for each combination of files and for all types. A suggested format for storing the information is explained in the Appendix section.

4.6.2 XML

Another most widely used format for storing information is XML. This format can contain dependency information for each class and can also contain the statistical information of the dependencies. Even though Excel is a more convenient way of looking at information, XML has lot of other uses in a way that there is a lot of XML parsing tools to extract individual information from the whole lot of them.

4.7 Storing error logs

In case of errors during the opening or processing of files, the generated error logs could be saved into a file and an appropriate message could be displayed to the user specifying the location of the error log.



The activity diagram above conveys the activities that are part of this tool. This diagram's purpose is to represent activities because they indicate the functional details and provide the lower level information that group into tasks which again groups into modules at a higher level.

5. Partitions-Modules and Classes

Bases on the tasks and the activities within each task, the tool is subdivided into 8 modules as described below:

- 1) UI Module
- 2) File Manager Module
- 3) ErrorHandler Module
- 4) Parser Module
- 5) Semi-Expression Evaluator module
- 6) Tokenizer module
- 7) Type Definition Storage Module
- 8) Type Relationship Storage Module

The responsibilities of each module, dependencies between them and the contained classes are described in detail below:

5.1 UI Module

This module is the User Interface module responsible for accepting input from the User and displaying output to the user. The user is prompted to enter the location of source files to process. The input location is then sent to the File Manager module. The final output is displayed in a convenient format for various users.

Class and interfaces used:

- 1) CRAInput- To get the input path from the user and pass it to File Manager module.
- 2) CRAOutput – To get all the stored relationship and dependency information from the used data structure and display it on the screen.

5.2 File Manager Module

This module is responsible for managing the list of source files that needs to be processed. The files are identified from the specified location and maintained in a list. This module talks to the Error Handler module in case it is unable to open or process some files or the entered location is invalid. It ensures that the file streams are properly closed after they are processed. This module interacts with Parser module and sends each file in the list to it one by one to be parsed.

Classes and interfaces used:

- 1) CFileProcessor – To process each of the files in the list and send it to parser.
- 2) IFileProcessor – Contract for the services of the CFileProcessor.

5.3 Error Handler module

This module is responsible for handling file related errors. This module shows error or warning messages to the user. For example, a file can't be opened or location doesn't exist or doesn't have access. In these cases appropriate messages are shown to the user and are also logged in a file for the user to get complete error information.

Classes and interfaces used:

- 1) CFileErrorHandler – To handle file related errors. It also handles errors in case the user input path points to an invalid location or there are no "*.cs" files in that location.
- 2) IFileErrorHandler – Contract for the services of CFileErrorHandler.

5.4 Parser module

The Parser module is a container of rules each of which is a detector for some grammatical construction like class, struct, interface, enum, functions etc., This is written by Dr.Fawcett. The rules contain a set of actions which is executed when a particular rule is detected. For example when a class or interface definition rule is found out, appropriate action is taken. The parser uses the services of Tokenizer module and Semi-Expression evaluator module for its operation. Besides the existing classes and interfaces, some additional classes are needed to achieve the required functionality.

Classes and interfaces used:

- 1) Parser – Main class used for parsing. Contains set of rules.
- 2) CSemiExp – Class that holds a semi-expression
- 3) CToker – Class that splits a file stream into tokens
- 4) IRule – An interface contract for grammatical rules.
- 5) ARule – An abstract class to define rules for some common operations.
- 6) IAction – An interface contract to specify actions for rule.
- 7) AAction – an abstract class to define actions for rules.
- 8) DetectNameSpace – rule for detecting namespaces
- 9) DetectClass – rule for detecting classes, interfaces, structs and enums
- 10) DetectFunction – rule for detecting functions
- 11) DetectScopeChange – rule for detecting scope changes

New classes and interfaces to be used:

- 12) DetectInheritance – rule for detecting inheritance relationships
- 13) DetectComposition - rule for detecting composition relationships
- 14) DetectAggregation - rule for detecting aggregation relationships
- 15) DetectUses - rule for detecting uses relationships

5.5 Semi-Expression evaluator module

This module contains the C# semiexpression detection class written by Dr. Fawcett. Semiexpression is a partial C# expression, and is a sequence of tokens that have meaning for code analysis. It is a minimal sequence of tokens that ends with the characters '{', '}', ';' or '\n' if the line in which the newline appears starts with the char '#'. This module uses Tokenizer module for its operation.

Classes used:

- 1) CSemiExp – Form a partial expression from a set of tokens.

Instances of class CSemiExp class provide facilities for constructing, reading, analyzing and editing semi-expression token sequences.

5.6 Tokenizer module

This module contains the C# tokenizer class written by Dr. Fawcett. This is a simple tokenizer that does not combine multi-character operators (such as "<<", "+=", "++") into single tokens. It supports reading words, called tokens, from a string or a file stream. Transitions from alphanumeric to whitespace or punctuator characters and back again, are treated as token boundaries. Quoted strings and comments are returned as single tokens.

Classes used:

- 1) CToker – to separate tokens from a file stream or string stream.

5.7 Type Definition Storage module

This module is responsible for holding the type definition information present in all files after they are parsed in the first pass. This information is stored in a Hashtable where the key is the type name and the value is an object containing actual type and the file name which owns it. This module is used by the Type Relationship and File dependency storage module.

Classes and interfaces used:

- 1) CTypeInfo – A class to store the actual type and the containing file name for each type name identified.
- 2) StoreTypeDef – A class that maintains a list of CTypeInfo objects along with the type names in a Hashtable.

5.8 Type Relationship and File dependency storage module

This module is responsible for holding all the file dependency and type relationship information present in the source files after they are parsed in the second pass. The file dependency is identified by checking each token in the file against the Hashtable and comparing the file names. If the file name differs then it means a type from some other file has been used in the currently parsed file which conveys a dependency. The type relationships are then analyzed using new parser rules. This information is stored in some of the data structures discussed earlier so as to ensure proper display of information in appropriate formats. This module interacts with Type Definition Storage module.

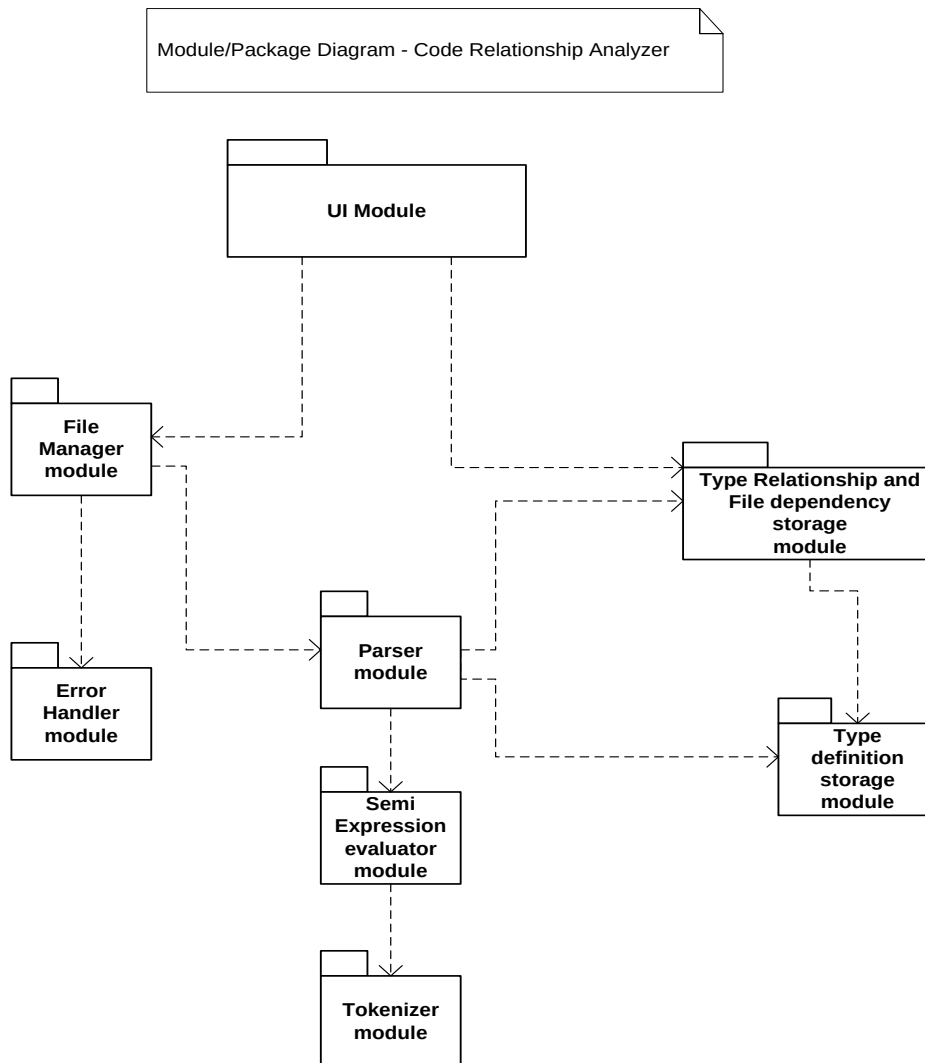
Classes and interfaces used:

- 1) CFileDependency – A class to hold file dependency information for each source file.
- 2) CTypeDependency – A class to hold type relationship information for each type identified.
- 3) ADependency – An abstract class that defines common operations for dependencies.

The above mentioned modules and classes are represented in the following package diagram (Modules are synonymous with packages) and class diagram respectively.

PACKAGE DIAGRAM

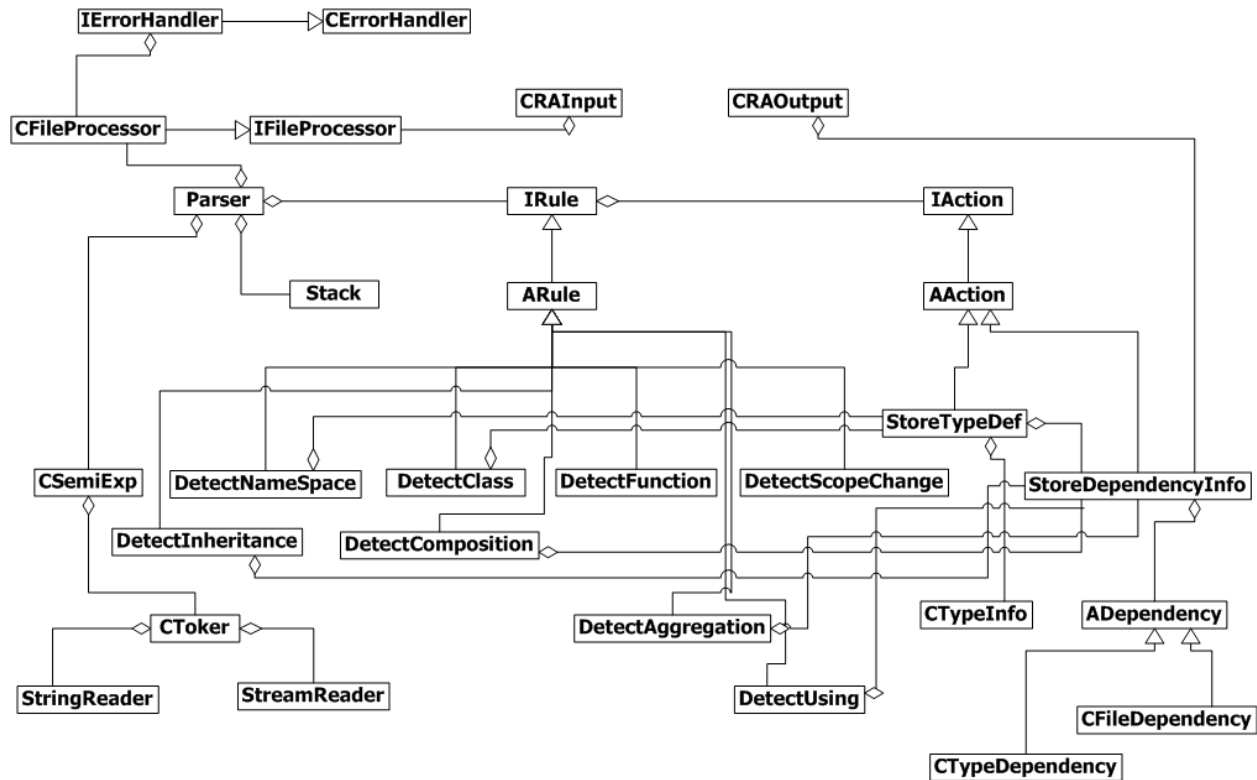
The below package diagram gives a high level view of packages and their dependencies:



The package diagram helps in visualizing the system at a high level. Through this diagram, the complexity of the tool could be guessed based on the number of modules and how they interact between them. Always the first step in understanding a new system or tool would be to analyze and understand the package diagrams, which is why this diagram is important and is here. The modules also give a picture of functionality of the entire system or tool.

CLASS DIAGRAM

The below class diagram gives a picture of various classes that constitutes each package.



This further explains modules at a lower level by depicting the various classes that form the whole tool and their relationships. The attributes and operations are suppressed for now as this diagram is just to understand the structure of each module at a higher level. The finer inner level details would be a part of design and this would be very helpful for the design team to get a nice picture before actually defining attributes and member functions for each class.

6. Critical Issues

Any tool that is being developed may have some critical issues that need to be addressed. This tool has some issues which need to be peeked at and are described below.

6.1 Large number of files:

When the number of files whose dependency information needs to be identified is large, there are two potential problems.

6.1.1 Displaying of information

The tool's responsibility is to show all the file dependencies as well as static type dependencies. When the number of files exceed certain threshold, it is really hard to make all the information fit into a single window so that the user can view all of them at a time.

Solution:

The tool can display information in such a way that the user sees only a small amount of information at a point of time in a list and can see the whole information by scrolling down the list. This cannot be avoided during this situation and is not a very critical issue. An output format like graph may show all the information at a single point of time but at a higher level.

6.1.2 Time taken/Performance

Time taken to analyze dependencies and show them to the user is critical as that decides the performance of the tool. It is not effective when the tool takes a long time to respond back to the user with all the necessary information.

Solution:

This problem is difficult to address as a large project could involve a large code base and large number of files. The least operation that should be ensured is to focus on performance of the tool during processing time as this may reduce a considerable amount of time taken than expected and would compensate to some extent, the actual problem.

6.1.3 Memory

The memory consumption would be considerably high when there is large number of files. But this is not a potential problem as nowadays, memory capacities have increased and this should not be a threat to performance.

6.2 Addition/Removal of a file during processing:

Addition of a new source file or removal of a file to the project during processing may cause some errors or warnings to be shown. The tool might not take into consideration, the newly added file or it may stop processing due to deletion of file when it is being processed.

Solution:

- The newly added file is ignored
- When a file is deleted which results in an error, an appropriate message is logged and the tool proceeds to the next file.

6.3 Improper code structure violating syntax:

When a large number of files to be analyzed are fed to the tool, there is a possibility that some files have build errors and doesn't have proper code structure. In such cases the tool may not provide complete or correct information.

Solution:

The user must ensure that the source files to be analyzed contain proper code and are free from build errors. This helps in obtaining comprehensive and correct information from the tool.

6.4 Tool Maintenance

It is necessary to maintain this tool once it is developed and tested and people start using it. There is a possibility that the tool has problems while it is in production or users face some problems in interacting with the tool. In these situations, it is essential for the person who looks into the problem to have sheer knowledge on the tool and its functioning.

Solution

In parallel with the development of this tool, proper documentation about the functioning of each module is essential. Even the lowest level of detail should be documented to support the tool after it has gone live.

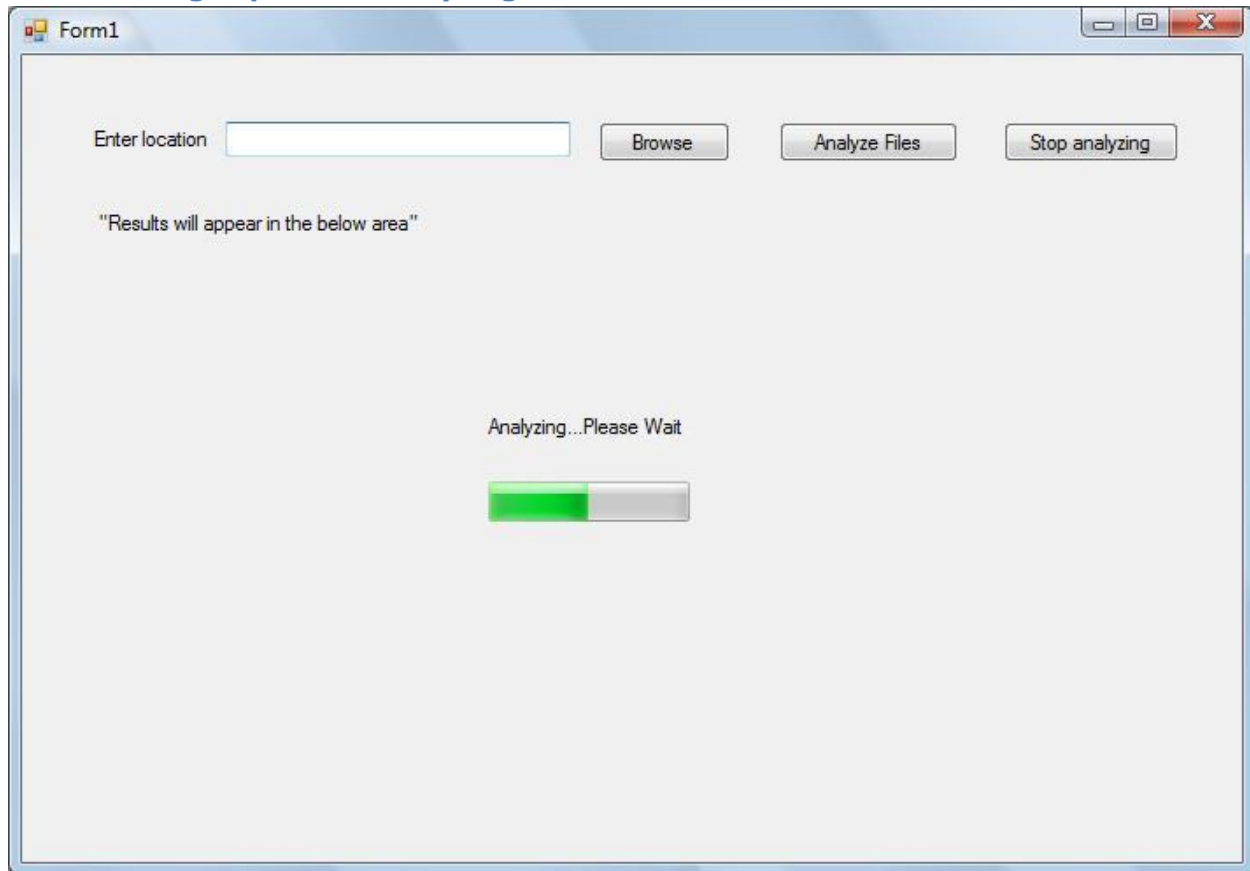
6.5 Ignoring namespaces

Ignoring namespaces may lead to occurrences of false positives. But this occurrence happens rarely and hence is not a critical issue.

7. Views

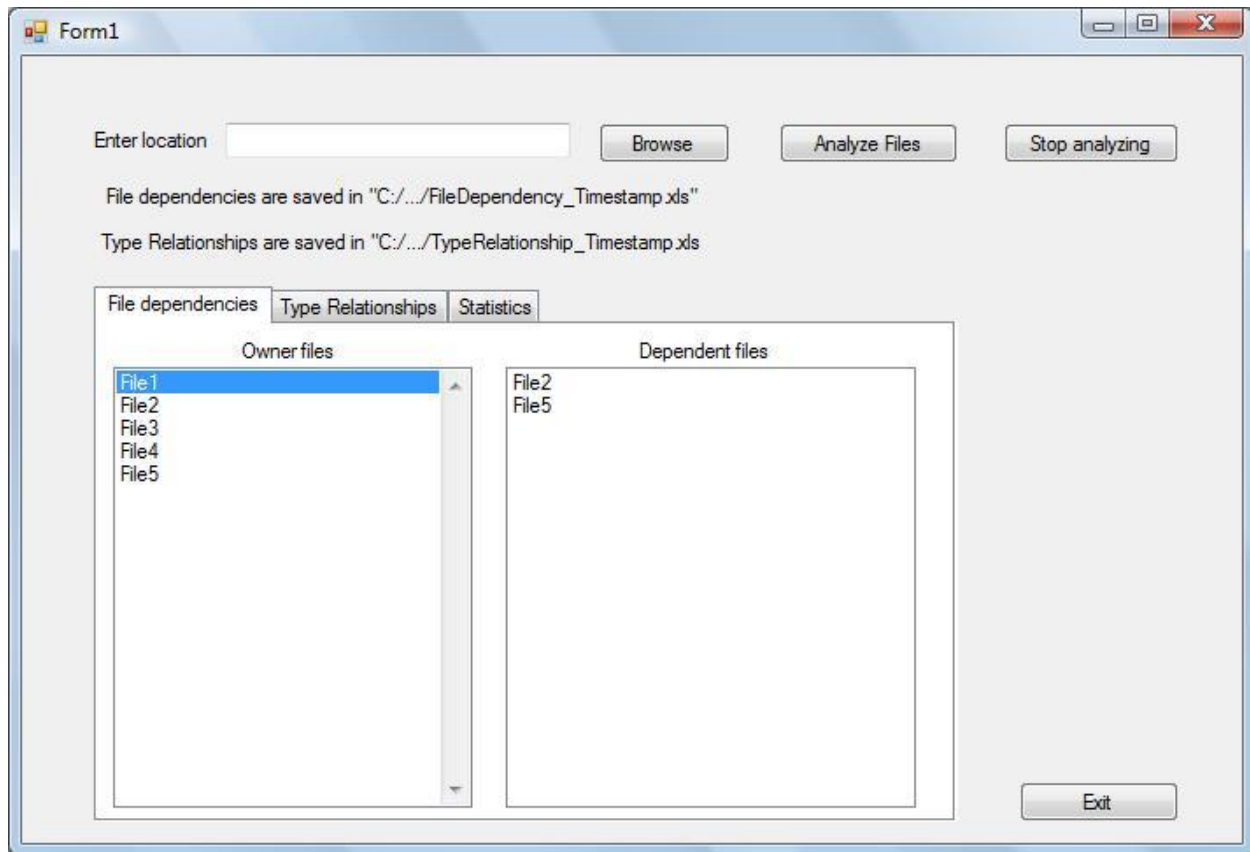
This section describes the appearance of the system to its users and designers. The proposed user interface is depicted in the following screenshots below:

7.1 Obtaining Input and Analyzing files



- 1) When the User interface is loaded by the GUI module, the user is prompted to enter the input location files. The input location of files can be browsed using the Browse button.
- 2) Once the location is fed, the user has to click the button "Analyze Files".
- 3) This event triggers the tool to start analyzing all the ".cs" files for dependencies and relationships.
- 4) The user can select the "Stop analyzing button" anytime during the processing of the tool. This causes the tool to exit from this current operation.
- 5) A progress bar indicates the progress of the tool and the amount of task that has been completed.
- 6) The label designated as "Results will appear in the below area" conveys to the user where he would see the results after the tool processes the source files.
- 7) In case of any errors a message box could be displayed to the user conveying about the error or the location of the error log that has been generated and then prompt the user again for input. This error log enables the user to troubleshoot what went wrong and restart his/her process.

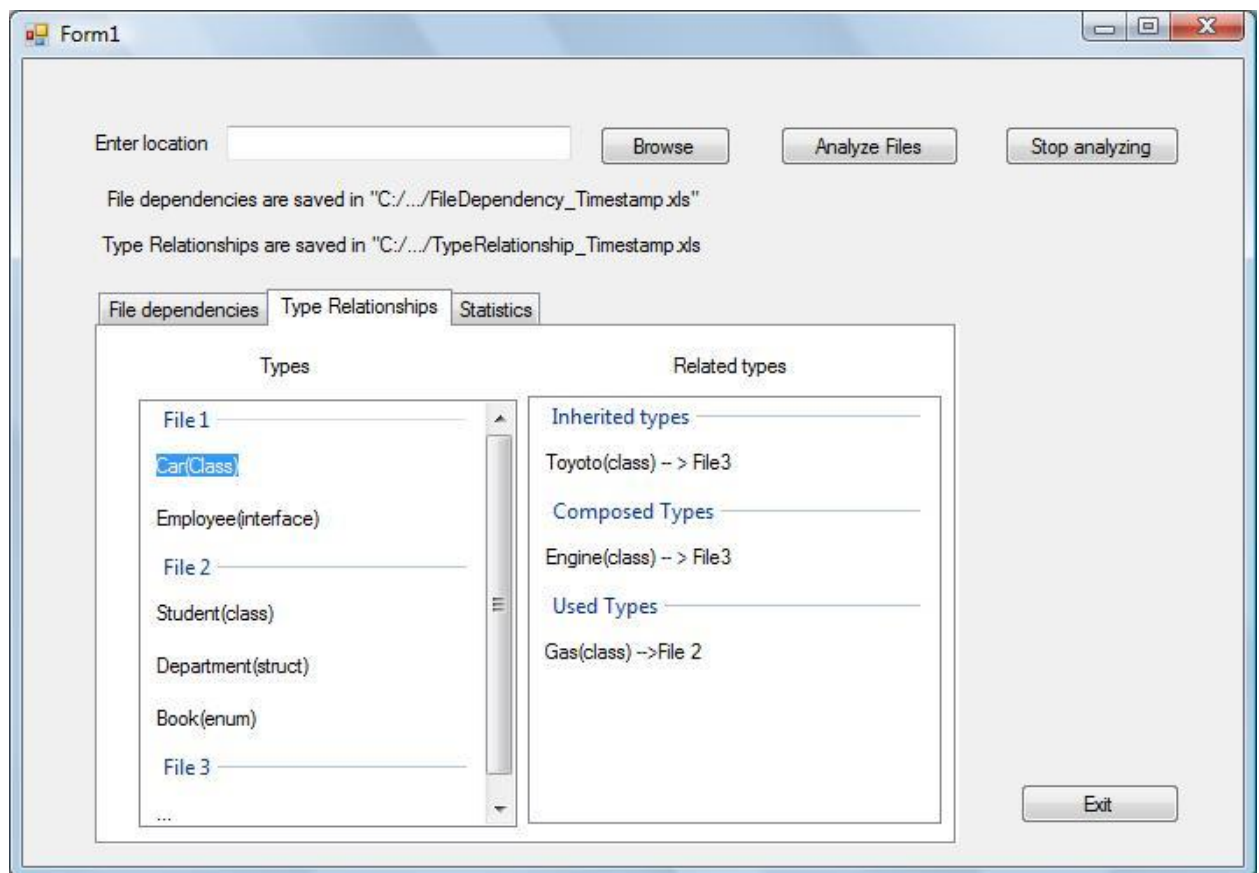
7.2 Showing Output in Tabs



- 1) The output information is presented to the user in tabs. There are 3 tabs in the output information.
 - a. First tab- shows File dependencies
 - b. Second tab – shows Type Relationships
 - c. Third tab – shows Statistics
- 2) At any point of time only one tab is active.
- 3) Tab sheets are suitable for output because we have less categories of information to be shown which could be suppressed inside tabs. All the information to be shown under a particular category could be placed under a single tab. It is also easier for the user to navigate across tabs or switch tabs to see the desired information he/she needs.
- 4) By default, when the result is loaded the File dependencies tab is active.
- 5) As already mentioned earlier, the output information can stored persistently into the file system in any one of the two formats-excel or XML file. The above screenshot displays information that is shown to the user regarding the same.
"File dependencies are saved in "C:/.../FileDependency_Timestamp.xls" and Type relationships are saved in "C:/.../TypeRelationship_Timestamp.xls– Here the files could also be XML files instead of excel. Timestamp contains the date and time during which the file is being created.

For example, FileDependency_10122010_1340.xls". Timestamp is important because, the tool could be run many times by any number of users, and the output information/report for each run needs to be saved. When the numbers of reports exceed a certain limit, it is the responsibility of the user to backup the different versions that are created.

- 6) The File dependencies tab shows two lists. The list on the left contains owner files and the list on the right contains dependent files. Dependent files are shown only when the user selects a particular file on the left. This tab displays for each file on the left, a set of all its dependent files on the right.
- 7) The Type Relationships tab displays all the type relationships that exist in the source code. The output is displayed in two panels. The panel on the left contains all the type definitions and the actual type (class or interface etc.,) grouped together for each file and for each selected type on the left, the right panel displays the type relationships existing in other files. The right panel shows all the inherited types, composed types, aggregated types and used types along with their containing file names. This grouping of types using individual files helps the user understand the type relationships more clearly at file level which in turn reveals the dependencies.



9) The Statistics tab shows statistical information based on the overall number of file-level dependencies and type relationships. This statistical information may be very useful for the managers or customers as they would be more interested in numbers that lower-level details.

Form1

Enter location

File dependencies are saved in "C:/.../FileDependency_Timestamp.xls"

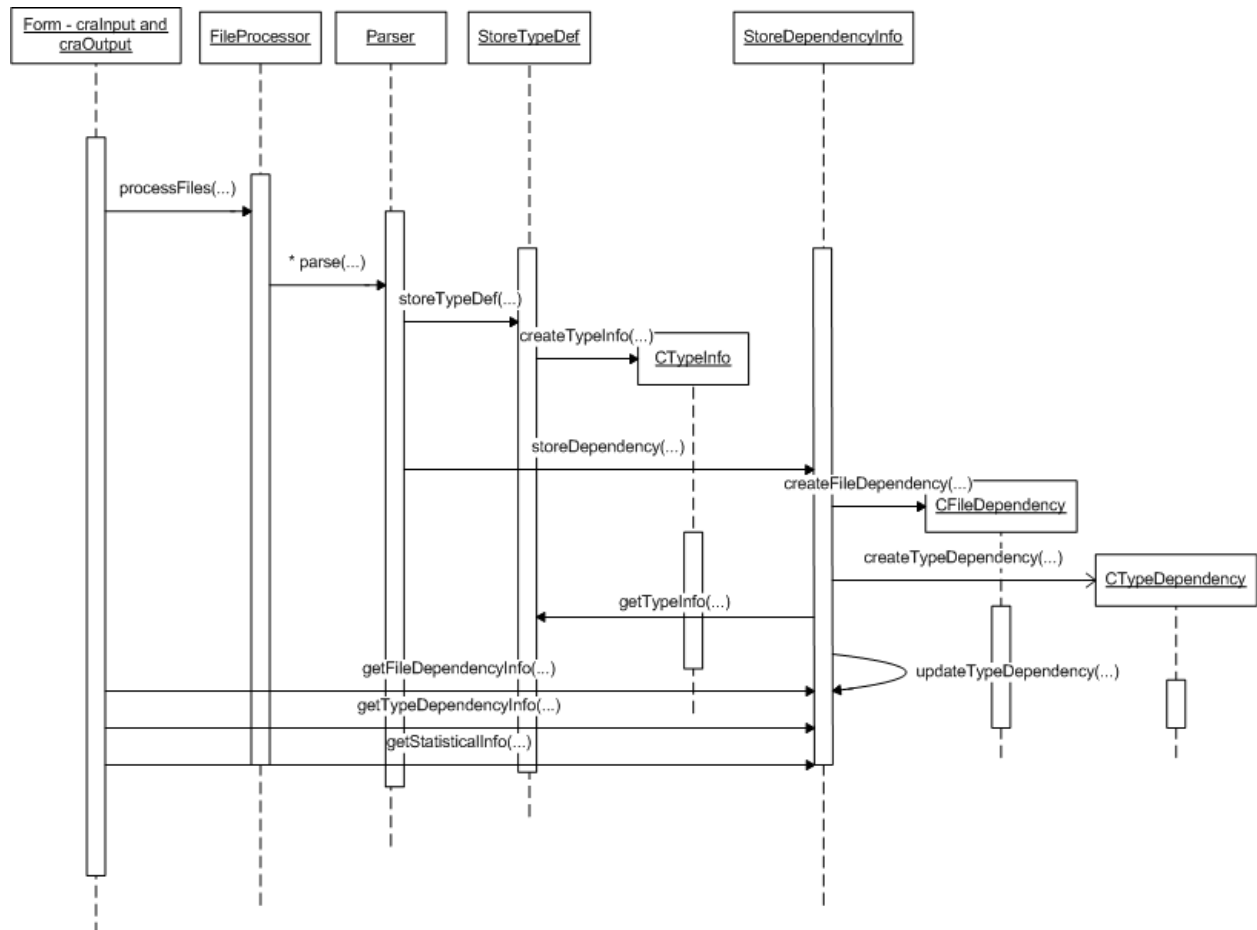
Type Relationships are saved in "C:/.../TypeRelationship_Timestamp.xls"

Statistical Information

Number of files analyzed	:	5
Number of File dependencies	:	10
Number of inheritance relationships	:	7
Number of composition relationships	:	9
Number of aggregation relationships	:	2
Number of Using relationships	:	3

8. Events

The Event trace diagram or Sequence diagram below illustrates the events that occur in the system and better explains the interactions between the objects involved in a detailed fashion.



The class diagram that was illustrated earlier showed the various classes that would be used, which depicts the structure of each module. This diagram is a detailed portrayal of how different classes or its objects interact by using method calls. Method calls occur as a result of an external event. When a user presents an input path to the input module window, an event occurs and then as a second step, the input path is passed to the CFileProcessor class. The sequence of events that follow is represented in the sequence diagram. Some objects are being created as part of an event. In the above diagram CTypeInfo, CFileDependency and CTypeDependency are created to store the dependency information. Finally the UI module contacts the CStoreDependency class to get all the stored information and displays them to the user.

The diagrams purpose is to better understand the lower level activities that take place within the tool as a result of an external request from the user.

9. Appendix

7.1 Proposed Excel file format for storing output information:

File dependencies

Files	File1	File2	File3	File4
File1		Uses	Uses, inherits	composes
File2	aggregates			uses
File3				
File4				

The intersection of each row and column specifies the dependencies between both the files. The cell value represents, by what relationship, the files are dependent. All combinations of files should be considered. When there are no dependencies, the cell is left blank.

Type relationships

Types	File name	Inherited types and File name	Composed types and file name	Using types and file names	Aggregated types and File names
Car	File1	Toyoto - File2	Engine - File3	Gas - File4	
Employee	File2				
...	...	...	...	...	

Both file dependencies and type relationships are represented in different ways. They could be shown in two different sheets in a single excel file. The above table gives for each type, its containing file name, and all its related types together with their containing file names.

7.2 Proposed XML File format for storing output information:

```
<class-relationship-analyzer>
<inputloc>"path"</inputloc>
  <class name="Car">
    <subclass name="Toyoto"></subclass>
    <Composedclass name="Engine"></composedclass>
    <usedclass name="Gas"></usedclass>
  </class>
...
<statistical-data>
  <number-of-files>100</number-of-files>
  <number-of-dependencies>200</number-of-dependencies>
  <number-of-inheritance>100</number-of-inheritance>
  <number-of-composition>75</number-of-composition>
  <number-of-aggregation>75</number-of-aggregation>
```

```
        <number-of-using>50</number-of-using>  
<statistical-data>  
</class-relationship-analyzer>
```

This XML file can be parsed using a DOM parser or a SAX parser whenever needed for any further analysis or reference.

10. References

- 1) Course handouts for CSE681 given by Dr.Fawcett
- 2) <http://www.ecs.syr.edu/faculty/fawcett/handouts/cse681/code/>
- 3) <http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/StudyGuideOCD.htm>