TrueRefactor: Automated Refactoring of Legacy Software Systems to Current Best Practices

Isaac Griffith and Stephani Schielke

Montana State University - Bozeman

1 Introduction

As software engineering evolves, there is an ever-increasing desire in the software development industry to use "sound engineering" principles to design software. Even though this trend is producing excellent software solutions, there still exist legacy software artifacts which are in need of re-engineering or refactoring to meet today's best practices. Legacy software is poorly engineered when compared to currently produced software due to the lack of current standards and design principles when it was developed. Problems are encountered when current developers are required to maintain and update the legacy systems. This usually is caused by the fact that the original developers no longer available for the maintenance of the system or even for reference questions in most cases. This disconnect between the current and original engineers stems from the ever-changing and improving nature of software engineering practices. Given that such a disconnect exists and the changing nature of software engineering principles and practices, the original intention and design choices made in the legacy software systems may be unrecoverable by current engineers (in a practical amount of time). In order to shorten the amount of time that an engineer requires to understand a legacy system, we propose an automated system which could refactor a legacy system in such a way that it conforms to today's best practices.

Such a system could grant developers the ability to understand software that is not understandable without considerable time and resources devoted toward it. The majority of time spent on code in software engineering is in the maintenance portion of the life cycle. The system proposed herein aims to significantly reduce this portion of the development life cycle which would lead to substantial time and financial savings.

Within this proposal one will find the qualifications of the authors for undertaking such a project in Section 2. Previous work done on this topic and other concepts incorporated which will help to create a successful project are found in Section 3. The work schedule and development standards, Sections 4 and 5.5, are adequately detailed to assure the reader that the project is possible within the time and requirement constraints. The project requirements, methodology and design patterns used are thoroughly described in text as well as with numerous UML diagrams to convey the structure and design behind the program. They can be found in Sections 5 and 6.

2 Qualifications

Provided here are Isaac Griffith's skills and areas of focus, along with a set of relevant experiences, which given in conjunction with Stephani Schielke's skills and experiences, should be more than adequate to accomplish the goals of this research project.

2.1 Isaac Griffith

Education

Senior in Computer Science and Philosophy Departments at Montana State University. His current goal is to enter into a Master's Degree program at MSU Bozeman in the Computer Science Department.

Relevant Course Work:

- Artificial Intelligence and Machine Learning
- Advanced Algorithm Design
- Software Engineering I and II

Skills

1. Languages fluent in or have used:
    - Java
    - C/C++
    - SQL
    - Perl
    - PHP
    - Ruby
    - Clojure
2. Areas of Focus:
    - Artificial Intelligence
    - Machine Learning
    - Graph Theory
    - Software Engineering
    - Automated Software Engineering

Relevant Work Experience

Research Assistant

Organization:	Computation Ecology Research Group		Montana State University - Bozeman
Supervisor:	Clemente Izurieta, Ph.D.	**Date:**	10/2008 - Present
Responsibilities:			

- Developed a graph-based dependency tracking and concurrent iteration system.
- Helped to develop the underlying modeling framework known as NEO
- Helped to develop and maintain the UML documentation of the project.

Software Development Intern

Organization:	RightNow Technology Inc.		Bozeman, MT
Supervisor:	Ernie Turner	**Date:**	04/2010 - 10/2010
Responsibilities:			

- Developed a JavaScript based testing framework, using YUI, to test the Customer Portal platform.

Software Development Intern

Organization:	Advanced Acoustic Concepts		Bozeman, MT
Supervisor:	Ivan Van Dessel	**Date:**	05/2008 - 04/2010
Responsibilities:			

- Develop a statistical analysis package for SupDMS project using Java and the Google Web Toolkit, which was AJAX enabled.
- Prototyped and Developed GUI components in both the Swing and GWT UI toolkits.
- Developed servlet based components for the server-side back end of the SupDMS project.

Software Development Intern

Organization:	Montana Dept. of Livestock		Bozeman, MT
Supervisor:	Jim Newhall	**Date:**	01/2007 - 10/2007
Responsibilities:			

- Performed testing and development on the Laboratory Information Management System (LIMS).
- Required knowledge in Oracle PL/SQL and Oracle 9i database.
- Developed User Interfaces, and corresponding back end SQL databases to integrate with the main database.

2.2 Stephani Schielke

Education

Junior in the Computer Science Department with a Minor in Mathematics at Montana State University.

Relevant Course Work:

- Methods of Proof for Higher Mathematics
- Advanced Algorithm Design
- Software Engineering I

Skills

1. Languages fluent in or have used:
    - Java
    - C++
    - PHP
2. Areas of Focus:
    - Algorithm Design
    - Software Engineering

Relevant Work Experience

Software Development Student Associate

Organization:	SRI International		Helena, MT
Supervisor:	Lynn Voss	**Date:**	5/2010-8/2010
Responsibilities:			

- Updated and added functionality to existing software projects
- Set up web-based server for GIS data
- Recorded user instructional videos for software
- Wrote documentation for current version of software

3 Background

3.1 Epistemological Retension Refactoring System

This research project builds upon an existing project known as the Epistemological Retension Refactoring System (ERRS). The goal behind ERRS was to use artificial intelligence local search and optimization techniques to refactor legacy source code.[2] ERRS provided the work and direction towards a proof of concept, but unfortunately was due to team management issues, team communication problems, and shared media problems the overall goals where not met. The outcome of this project was a malfunctioning automated refactoring tool, which had problems located in its underlying graph data structure and input processing system. In order to make ERRS work correctly a secondary graph data structure, created from the information in the original, was required to be build for every individual of a population produced by the genetic algorithm.[2] This led to a large amount of overhead and greatly increase both the time and space requirements required for the algorithms used.

These problems left the project in a state of hacked together code and a limited solution (at least as viewed from the current design goals). In order to solve these issues the current project, now renamed TrueRefactor, has refactored the underlying Input Processing System and has defined the Code Graph data structure so that it meets the original intention and design goals of ERRS, while simultaneously meeting the current design goals of TrueRefactor. The TrueRefactor team has also refactored the Genetic Algorithm subsystem such that there no longer is a secondary graph structure, thus decreasing the complexity of having two graph structures performing the same operations.

Beyond the issues within the code, the ERRS team was itself dealing with several issues. The original intention was to setup a source and revision control for the entire project prior to performing work on the project. Unfortunately, this did not get done until very late in the project, and past the original deadline. Due to this issue maintaining a single set of current source code and project documentation was not handled well, and at in a word was a failure. The lesson learned was that prior to beginning work on the project, the TrueRefactor team has been using GoogleCode to maintain all design documentation and current source code, with a revision control scheme.

The second inter-team issue that plagued the ERRS team was the the issue of team management. This problem can be broken down into three factors. The first of which is time management. Initially the project started off well, with the inception of ideas, but the progress from inception to construction was constantly delayed by the other two factors: continual retraining of team members and reassignment of work. In remedy of all of these issues, the TrueRefactor project team has a current set of defined requirements and dates to deliver these goals.

3.2 Genetic Algorithms, Metrics, and Code Smells

3.2.1 Genetic Algorithms

Genetic algorithms provide an algorithmic means by which the solution space of a problem can be investigated by testing the viability of a large set of potential solutions at a time. When given some problem, a set of solutions, known as the population, is randomly generated. Depending on its viability, the population that is generated informs the algorithm of the area of the search space to concentrate on. Each member of the population, typically an individual or chromosome, is composed of entities known as alleles.[4] Traditionally in genetic algorithms, each allele is a binary digit and the combination of the bits, represented in a chromosome, represent an entire potential solution.[4] Each population generated during an iteration is known as a population generation. The genetic algorithm functions by selecting a pair of solutions and then combining those solutions to form a new pair of solutions. This combination is performed by randomly selecting a point within the individuals where a swap of the portion after that point, in one of the individuals, to the other is performed thus producing two new offspring.[4] After this combination procedure is applied, there is a chance that a random mutation of an allele in each of the individuals can occur, thus providing a chance to obtain potential solutions not yet found in any population.[4] This set of procedures is performed for all selected pairs until an entire new generation is produced. Each individual of this new generation is then measured to determine its fitness and whether it will remain within the population.[4] This process continues iteratively until some stopping criteria is met or until an individual(s) with maximum fitness have been produced.[4]

With this basic understanding of Genetic Algorithms understood, we can now discuss their application to refactoring as a part of the ERRS project. Within the original system the metrics provided a means to measure the improvement of the code by refactoring, or in other words it provided a means by which we measured fitness of an individual of the population produced by the GA. Each individual is itself an ordered list of refactorings, which are to be applied in sequence.[2] Each refactoring technique is then a combination of graph manipulations to be applied to the graph representing the code base of a software project.[2] In order to apply the refactorings within an individual, a clone of the graph is created, the refactorings are applied sequentially, and then the metrics are measured.[2] The fitness of the set of individuals is then a combination of these measurements. This process is repeated for each individual in the population. It was intended that only those individuals that increased the performance of the measurements would be kept, and the rest discarded.[2]

3.2.2 Metrics and Code Smells

The metrics themselves were selected from a various object-oriented design metrics sets and were intended to measure the properties of source code which have been shown to correlate to understandability, maintainability, and reusability. The metrics, as stated above, provided a means by which to inform the GA about the fitness of each individual of a population, but there was a problem of determining when to stop the GA's processing.[2] Due to size mismatches of individuals used by the GA we were not able to guarantee that convergence would occur. The ERRS project instead simply relied on reaching a specific number of iterations through the GA to be used as the stopping point. Since the stopping condition was solely dependent on empirical findings and not necessarily on a notion of overall improvement of the code base, a secondary form of determination was required.[2] In order to determine an ideal or optimized stopping point of the GA, the TrueRefactor team has decided that the use of what are known as "Code Smells" would provide better stopping heuristics.

Code Smells are a set of qualitative heuristics intended to provide software engineers with an indication of possible problem areas within source code, and a prescription of possible refactorings which are known to solve such problems.[1] Given the nature of a GA which attempts to generate a large portion of a search space, and systematically, by combining the current best solutions to date, produce the optimal solution, we can then simply measure code smell across the code base. Comparing this measurement to the initial measurement would allow one to determine if the refactorings are having the intended effect of producing better and more structurally sound code, and once minimized to near zero (or some predefined value), the GA could then quit.

3.3 Output

The final area left unattended in the ERRS project was that of output. It was initially envisioned that the system would not only be able to produce the refactored version of the source code, but would also be able to produce UML diagrams which describe the system in a language most, if not all, object oriented software engineers could read.[2] In order to develop UML diagrams we need to be able to generate Object Modeling Group's (OMG) Meta-Object Facility (MOF) compliant diagrams. MOF is a means of describing meta-modeling, of which UML is a part, and defines a specific XML based file format called XMI.[5]

4 Work Schedule

The work schedule has been divided up using a work breakdown structure (WBS) where each major task has been broken down into two finer levels of granularity. At the deepest level an assignment to a member of the team, a due date for that specific element of the software, and the element's current status have been specified.

Process Activity	**Task Set**	**Task**	**Sub-Task**	**Status**	**Due Date**	**Assigned To**
Inception	Research	Philosophy	Dark Programming	Done	29-Oct	Isaac
			Abstraction	Done	29-Oct	Isaac
			CS Paradigm Shift	Done	29-Oct	Isaac
		Research	Mathematics	Metrics	Done	15-Jan
				Code Smells	Done	15-Jan
				Graph Manipulations	Done	15-Jan

Process Activity	**Task Set**	**Task**	**Sub-Task**	**Status**	**Due Date**	**Assigned To**
Inception	Analysis	Requirements	Outline Major Use Cases	Done	29-Oct	Isaac, Stephani
		Architecture	Determine Major System Components	Done	29-Oct	Isaac, Stephani
Elaboration	Research	Mathematics	Defined Metrics	In Progress	23-Nov	Isaac, Stephani
			Code Smells	In Progress	23-Nov	Isaac, Stephani
			Refactorings	In Progress	23-Nov	Isaac, Stephani
	Analysis	Requirements	Define Major Use Cases	Done	11-Nov	Isaac, Stephani
		Architecture	System Component Diagram	Done	11-Nov	Isaac, Stephani
	Requirements Analysis	Elicit Requirements	Functional Requirements	Done	11-Nov	Isaac, Stephani
			Performance Requirements	Done	11-Nov	Isaac, Stephani
			Interface Requirements	Done	11-Nov	Isaac, Stephani
			Development Standards	Done	11-Nov	Isaac, Stephani
			Expected Results	Done	11-Nov	Isaac, Stephani
	Design	Structural	UML Class Diagram	Done	29-Oct	Isaac
			UML Sequence Diagrams	Done	29-Oct	Isaac
		CLI	CLI Design	In Progress	15-Jan	Isaac, Stephani
	Testing	Regression Testing	Setup Regression Testing at NMI	Not Started	1-Jan	Isaac
		Unit Testing	Develop JUnit Test Cases	Not Started	15-Jan	Isaac, Stephani
Construction	Development	Structure	Implement UML Classes	In Progress	18-Dec	Isaac, Stephani
			Implement Data Structures	In Progress	18-Dec	Isaac
			Implement Model View Controller	In Progress	18-Dec	Isaac
			Implement Code Parsing	Done	18-Dec	Isaac

Process Activity	Task Set	Task	Sub-Task	Status	Due Date	Assigned To
Construction		Metrics	Implement Metrics Subsystem	In Progress	18-Dec	Stephani
	Design	Modeling	Develop Sequence Diagrams	In Progress	Jan-1	Isaac, Stephani
			Develop Subsystem Activity Diagrams	Not Started	Jan-1	Isaac, Stephani
	Development	Interface	Implement CLI	Not Started	Jan-15	Isaac, Stephani
			Implement Reporting	Not Started	Jan-15	Isaac, Stephani
		Output	Implement Output Subsystem	Not Started	18-Dec	Stephani
			Implement EMF Connection	Not Started	18-Dec	Isaac
Transition	Results	Methodology	Define Testing Code	Not Started	11-Nov	Isaac, Stephani
			Define Analysis Techniques	Not Started	11-Nov	Isaac, Stephani
		Generate Results	Run Against Test Code	Not Started	1-Feb	Isaac, Stephani
			Generate Results Graphs	Not Started	10-Feb	Isaac, Stephani
			Verify Results	Not Started	15-Feb	Isaac, Stephani
	Presentation	Final Poster	1st Draft	Done		Isaac, Stephani
			2nd Draft	In Progress	15-Mar	Isaac, Stephani
			Final Draft	Not Started	1-Apr	Isaac, Stephani
		Final Paper	1st Draft	Not Started	15-Feb	Isaac, Stephani
			2nd Draft	Not Started	25-Feb	Isaac, Stephani
			Final Draft	Not Started	1-Mar	Isaac, Stephani
		Final Product	Project Complete	In Progress	11-Mar	Isaac, Stephani

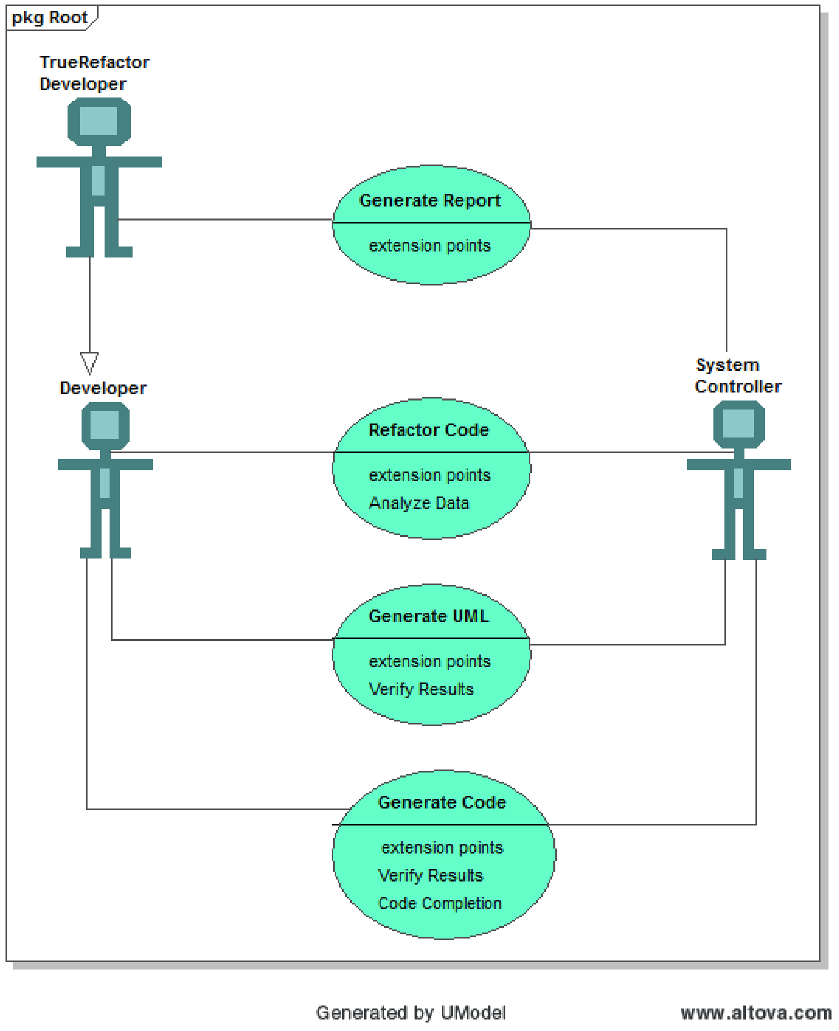
5 Proposal Statement

We are proposing a system designed to process a set of source code files and output both redesigned source code and matching UML documentation, via automated refactoring. The sections below enumerate requirements dealing with the GA, Code

Graph, and Input Processing Systems, each of which was part of the ERRS project; it should be made clear that these requirements are to define extra functionality which either was not originally incorporated or which requires a redesign. For each of these systems, maximizing the reuse of existing code is of the highest priority. The following sections: Functional Requirements, Performance Requirements, Interface Requirements, Architectural Design Requirements, and Development Standards describe exactly what is to be designed.

5.1 Functional Requirements

The functional requirements for TrueRefactor have been developed using UML Use Case analysis. The Use Case diagram can be found below, and the textual representations describing each use case and its encapsulation of the functional requirements can be found in sections 5.1.1, 5.1.2, 5.1.3 and 5.1.4.



5.1.1 Use Case 1: Refactor Code

Description: The Developer uses the TrueRefactor tool to refactor legacy source code to increase its understandability and to make it more structurally sound according to today's best practices (in accordance with refactoring techniques). The user sets the directory where the legacy software's code base can be found, the directory where to output the generated code to, the directory where to output the generated UML diagrams to, and the name of the report file to be generated.

Primary Actor: Developer, TrueRefactor Developer

Secondary Actor: System Controller

Pre-conditions:

- A legacy software code base exists and can be found in the directory specified.
- The Developer has access rights to this directory and the files contained within it.
- The user has specified the new generated code and UML documentation directories.
- The user has specified the name of the report file to be generated.
- The legacy system is using an object-oriented language that the TrueRefactor system has a current parser for.

Post-conditions:

Success End Condition: The best individual set of refactorings, which optimizes the measured values for the metrics and minimizes the overall code smell measurement, has been found.

Minimal Guarantee: A Code Graph representing the legacy code base is built and modified using refactoring techniques as directed by the Genetic Algorithm. The effectiveness of an individual set of refactorings is measured by the system's provided metrics. The overall effectiveness of the best individual set of refactorings is measured by the system's set of quantitative Code Smells.

Triggers: A legacy software system is available but not well understood, and requires refactoring to meet today's best practices.

Main Success Scenario:

1. Create a Code Graph for the entire code base of the legacy system.
2. Processing each file one at a time.
3. Each file is processed by a parsing code generated by JavaCC.
4. Measure the initial metrics and code smell for the Code Graph, providing a base line set of values.
5. Invoke the Genetic Algorithm to begin the refactoring process.
6. For each individual generated during an iteration of the GA generate a clone of the Code Graph and apply that individual's refactorings to the Code Graph.
7. Apply all known metrics to the generated refactored Code Graph to generate the fitness value for that individual represented.
8. For each iteration of the GA select the individual with the best fitness in that generation.
9. Generated a clone of the Code Graph and apply that individual's refactorings to the Code Graph.
10. Take this Code Graph and apply all known Code Smells to obtain an overall value of code smell.
11. Use the combined values measured for each Code Smell to determine whether to continue to refactor or not.
12. Report the best individual generated by the GA.

Variations:

2. Create a Code Graph for the entire code base of the legacy system, processing files concurrently.
   1. If a file cannot be read report this and quit.

Extensions:

1. Analyze Data
   1. After metrics and code smells have been measured store this information (indexed by iteration of the GA).
   2. The System Controller uses Generate Report to produce a report containing these measurements.
   3. TrueRefactor Developers use this report to analyze performance of the Refactoring System.

5.1.2 Use Case 2: Generate Report

Description: After the program has finished refactoring the code base, a report should be generated, if requested by the user, detailing the measured values for all metrics and code smells during the refactoring process. The report should also include information about the number of generations and size of the population per generation during the execution of the genetic algorithm.

Primary Actor: System Controller

Secondary Actor: TrueRefactor Developer

Pre-conditions:

- The user must have requested the extra output of a system report.
- The system has finished refactoring and has generated and applied the best individual set of refactorings.

Post-conditions:

Success End Condition: A text file containing the required information has been produced and written to the filesystem.

Minimal Guarantee: The reported information must contain the following:

- Size of population of a generation of individuals created by the Genetic Algorithm and indexed by iteration number of the GA.
- Values for each measured metric of the best individual indexed by iteration of the GA.
- Values for Code Smells measured for the best individual and indexed by iteration of the GA.

Triggers: TrueRefactor Developer requires a report of activity from the refactoring system.

Main Success Scenario:

1. The System Controller gathers the necessary information into a sufficient data structure.
2. This data structure is passed to the Output Subsystem by the System Controller.
3. The Output Subsystem builds the report file and saves it to the location specified by the user.

Variations:

3. The user did not specify a report file name to be used.
    1. Use a system default report file output to the same current working directory.

5.1.3 Use Case 3: Generate UML

Description: After the program has finished refactoring the code base the program should produce UML Class Diagrams documenting the new structure of the source code.

Primary Actor: System Controller

Secondary Actor: Developer

Pre-conditions: User has provided the System Controller with a output directory name, within a directory where the user has write-access rights.

Post-conditions:

Success End Condition: UML Class Diagrams, which are consistent with UML version 2.0, representing the structure of the refactored code base are produced and written to the file system in XMI file format.

Minimal Guarantee:

- The UML Class Diagrams for the refactored code base are constructed and produced using the Eclipse Modeling Framework (EMF) libraries.
- The UML 2.0 is the minimum standard for the generated UML.
- The XMI file format, as defined by OMG, is the format for the files to be written.

Triggers: The refactoring system has completed execution and returned a refactored Code Graph to the System Controller.

Main Success Scenario:

1. For each node of the Code Graph process the type and properties of the node to generate a corresponding EMF representation of that entity.
2. For each edge of the Code Graph process the type and properties of the edge to generate a corresponding EMF representation of that edge.
3. Once the entire Code Graph has been processed, produce the corresponding Class Diagrams using the EMF library.
4. From these diagrams using the EMF library produce an XMI file.
5. The XMI file should be written to the user specified output file name.

Extensions:

1. Verify Results:
   (a) The developer inspects the produce XMI file using an XMI compliant UML editing tool.
   (b) The developer can then ensure that the UML generated meets their needs.

5.1.4 Use Case 4: Generate Code

Description: After the program has finished refactoring the code base it should generate the newly refactored code into a user specified output directory.

Primary Actor: System Controller

Pre-conditions:

- User has specified the directory into which the generated source code should be written.
- The user and program should have write-access to the directory where the output directory is to be created (if it doesn't already exist), or have write-access in the output directory (if it already exists).
- The UML EMF structure has been created.

Post-conditions:

Success End Condition: Source code files have been generated by the EMF code generation facilities, and the files have been written to the filesystem in the location specified by the user.

Minimal Guarantee:

- The program will output source code that represents only the information contained within a UML Class Diagram (in other words stubbed classes and empty methods)
- The program will generate this source code using the EMF code generation facilities.

Triggers:

Main Success Scenario:

1. The System Controller is notified that the EMF representation of the final Code Graph has been created.
2. The System Controller then directs the Output Subsystem to generate the source code files in the user supplied output directory.

3. The Output Subsystem utilizes the EMF code generation facilities to generate the source code.

Variations:

2. The System Controller may wait until the Output Subsystem has completed generating the XMI files. (See Use Case 3: Generate UML)
    1. Once the XMI files have been generated it will continue to step 2. of the Main Success Scenario

Extensions:

1. Verify Results:
    (a) After the files have been completely generated the Developer can then verify the results.
    (b) This verification process starts the process of learning the code to develop enhancements or to simply continue maintenance of the source code.
    (c) Verification can also include the compilation of the code to ensure that it was correctly generated.
2. Code Completion:
    (a) The System Controller can read each file of the newly generated code
    (b) The System Controller then has this code parsed and using the final code graph generated from the refactoring process, it then inserts the code from the code graph back into the newly generated code.
    (c) Once the file is processed and the code correctly inserted, it is then written back out to the same file.

5.1.5 Textual Representation of Functional Requirements

The provided diagrammatic and textual representations of the main Use Case above provide enough documentation to derive the functional requirements for the system. These requirements are broken into four major subsections: 1) Output Subsystem, 2) Input Processing Subsystem, 3) Refactoring Subsystem, and 4) Code Graph Data Structure. Each of these subsystems will also be used to describe the associated performance requirements in section 5.2. Each requirement has been assigned an unique number for use later if tracking requirements will be required.

1. Output Subsystem

    5.1.5.1.1 The Output Subsystem shall generate UML Class diagrams for the refactored code base.

    5.1.5.1.2 The Output Subsystem shall generate Stub versions of the newly refactored classes, containing all the attributes and methods of each class.

    5.1.5.1.3 The Output Subsystem may generate the internal code for each method of the classes.

    5.1.5.1.4 The Output Subsystem should generate classes that are compilable using a compiler for that specific language.

2. Input Processing Subsystem

    5.1.5.2.1 The Input Processing System shall verify that every file to process exists. If a file does not exist or is not readable, the system shall throw an exception with a message that informs the user of the files causing the problem. After the exception has been thrown the system shall exit and return control to the operating system.

    5.1.5.2.2 The Input Processing System shall read each source file of the provided code base and parse the files. The parsing will be done using code generated by JavaCC parser generator.

    5.1.5.2.3 The Input Processing System should be able to allow multiple parsers generated by JavaCC to be implemented and used.

    5.1.5.2.4 The Input Processing System shall take the parsed files and generate a graph which represents enough metadata to perform refactorings.

    5.1.5.2.5 Parsing of any single Class file must be parsed to the line of code, but does not need to go any further.

    5.1.5.2.6 Each parsed line of code should generate a link to object types within the scope of the Code Graph.

3. Refactoring Subsystem

5.1.5.3.1 The Refactoring Subsystem shall provide a Genetic Algorithm (GA) to generate the best possible set of refactorings to be applied to the Code Graph where each population is comprised of individuals which are themselves simply a ordered list of refactorings.

5.1.5.3.2 The Refactoring Subsystem shall provide a set of metrics. Each metric must provide a means to measure some aspect of the project which can be related to a behavioral aspect of understanding the project. The process of measurement is to take the Code Graph as input and through graph traversals derive the measurement. The combination of these measurements form the fitness of the set of refactorings and thus inform the GA whether or not to keep that set of refactorings in the population.

5.1.5.3.3 The Refactoring Subsystem shall provide a set of heuristics to inform the GA when it is time to stop. The heuristics should be a set of quantitative measures based on the qualitative notions of Code Smells. These heuristics, like the metrics, should measure across the Code Graph to determine overall effect of the refactorings applied to date. Unlike the metrics, these heuristics are not used to inform the GA about a single set of refactorings, but instead inform the GA when it is done.

5.1.5.3.4 The Refactoring Subsystem shall accurately represent a set of refactoring techniques, which require no user interaction.

5.1.5.3.5 The Refactoring Subsystem shall provide a set of metrics which can be measured using only the information within the graph data structure

5.1.5.3.6 The Refactoring Subsystem shall complete refactoring when the amount of overall code smells as measured across the Code Graph has decreased below a specified minimum value.

4. Code Graph Data Structure

5.1.5.4.1 The Code Graph Data Structure shall be able to contain all classes, attributes of classes, methods of classes, and provide the connections between them. The classes, packages (name spaces), attributes, methods, and event statements of classes are represented as nodes. The connections between these types of nodes are contained as edges. This data structure stores enough information to generate UML documentation.

5.1.5.4.2 The Code Graph Data Structure shall ensure that each node will maintain the original code representing the distinguishing characteristics of the item it represents. This requirement is necessary in order to ensure that we can maintain the underlying logic of the original code as it is transformed.

5.1.5.4.3 Each edge of the data structure must maintain the correct connection between nodes, while maintaining the necessary information such as cardinality of either source or destination node, labels associated with the nodes, type of edge, and directionality of the edge. This is required to maintain enough information to engineer structural diagrams of the modified code.

5.2 Performance Requirements

1. Input Processing Subsystem

5.2.1.1 The Input Processing Subsystem shall collect information on attributes (and their types), cardinality of attributes, operations and their return types, uses between classes, generalization and realization between classes, in order to generate the code graph.

2. Output Subsystem

5.2.2.1 The Output Subsystem shall output UML diagrams for UML 2.3.

5.2.2.2 The Output Subsystem shall output UML diagrams in the XMI file format so that the file can be read by nearly all UML editing tools.

5.2.2.2 The Output Subsystem shall generate source code that must be able to be compiled without errors using a compiler for the specific version of the language of the output code.

3. Refactoring Subsystem

5.2.3.1 The Refactoring Subsystem shall calculate the minimum value of code smells to be less than the initial measured value of the code smells, but greater than or equal to 0.

5.2.3.2 The Refactoring Subsystem shall be able to compute each of the metric measurements and code smells measures in less than linear time (per calculation).

4. Code Graph Data Structure

5.2.4.1 The Code Graph Data Structure shall include in each of the nodes, representing classes, an aggregate of nodes representing their methods and attributes. This is required in order to reduce processing time.

5.2.4.2 The Code Graph Data Structure shall include in each of the nodes, representing methods, an aggregate of nodes representing the statements contained in the methods. This is required in order to reduce processing time.

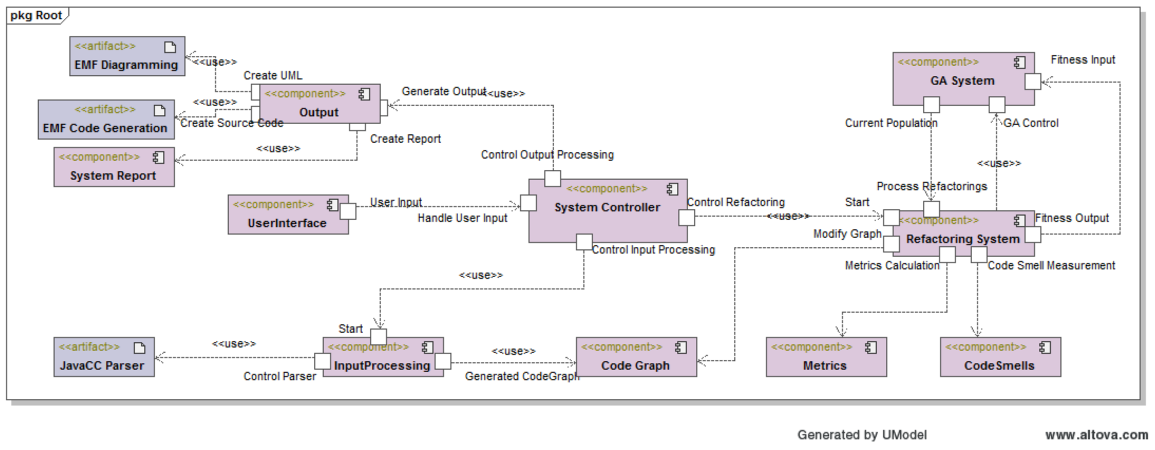
5.2.4.3 The Code Graph Data Structure shall include in each of the nodes, representing packages, an aggregate of nodes representing the classes contained in the packages. This is required in order to reduce processing time.

5.3 Interface Requirements

Name of Item	Description of Purpose	Source of Input/Destination of Output	Relationships to other inputs/outputs	Data Formats	Command Formats
5.3.1 UML Output Directory Name	Defines the directory to output all UML diagrams to	Command Line Option		String	–uml-out-dir name
5.3.2 Code Output Directory Name	Defines the directory to serve as a base directory to hold the generated code	Command Line Option		String	–code-out-dir name
5.3.3 Original Code Directory	Defines the directory of the original code base	Command Line Option		String	–code-in-dir name
5.3.4 Language	Sets the language to interpret the files in the original code directory as, and which file types to look for	Command Line Option		String	-l name, –lang name
5.3.5 Files	The actual files to be parsed to create the code graph	File System	Location of the files is provided by 5.3.3	Text	
5.3.6 Log File Name	File name to store reported information gathered during the programs operation	Command Line Option		String	–log-file file-name, -g file-name
5.3.7 Log File	File which actually stores reported data including refactoring employed, files written, initial and final metrics, and the initial and final code smells measurements	File System	Location of file provided by 5.3.6	Text	

5.4 Architectural Design Requirements

The architecture of the system is broken into the following components: Output, Refactoring System, Genetic Algorithm System, System Controller, Input Processing, and the User Interface. Each of these components and their interconnections will be described in this section. The overall System Architectural diagram can be seen below.



5.5 Development Standards

In this project we are following a modified form of the Unified Process. This form of the Unified Process is known as the Basic Unified Process (BUP) or OpenUP/Basic.[6] BUP is itself a part of the larger set of practices known as OpenUP, developed by IBM and later transitioned to the Eclipse Process Framework.[6] Within BUP there are four main phases: Inception, Elaboration, Construction and Transition each of which is an iterative process, in addition to the entire process also being iterative.[6] This Unified Process was selected due to its nature of combining the structure and iterative nature of prescriptive processes with the flexibility of agile processes. This specific form of the Unified Process was also selected due to its typical use by small teams working on short duration projects.

Each phase of this project is broken down into several distinct activities: Research, Requirements Analysis, Analysis and Design, Development, Testing, and Deployment. Depending on the specific phase more or less work for a particular activity may be required. Below each phase of this project is described including the details of the activities, the tools, libraries or techniques to be used, and the deliverables to be achieved.

5.5.1 Inception Phase

In the Inception Phase the main activities comprising the majority of the work are Research and the initial concepts of Requirements Analysis being performed. Near the end of this phase the remainder of the main activities should be started including Analysis and Design, Implementation, Testing, and Deployment. The transition criteria from the inception phase is the complete definition of the problem we are attempting to solve and the identification of the major use cases. Along with completing this requirement, we also require the tools to be used, the language for the project, and the testing platform also be defined. Once these requirements are met we can transition to the Elaboration Phase.

Currently, the project has met the transition requirements for this phase. The problem statement as defined in Section 1 of this document and the Use Cases in Section 5 are the main deliverables. We have also selected the Java 2 Standard Edition version 6 as our development language and the Eclipse IDE as our development platform. The selection of language and platform was due to the need for a language and IDE with which both team members are familiar with. We also selected this platform due to the availability on both Linux and Windows operating systems and for its rich set of plugins and inherent connection to the Eclipse Modeling Framework. As for testing, we have defined the need to create JUnit tests for each class created, and have selected the NMI Build and Test Laboratory to support our regression testing. Finally we currently have set up Subversion revision control for the project through GoogleCode, but have recently noticed certain connection issues that have led to a change to GitHub using Git as our revision and source control. We have selected UML as our modeling language for this project, and the tool we are using to create the models for the project is Altova UModel 2010.

5.5.2 Elaboration Phase

In Elaboration Phase the main activities to be performed are Research, Requirements Analysis and Analysis and Design. In this phase the Research activity should complete the majority of its work and begin to sharply drop off, opening up more time for Implementation. The same type of tapering off should also occur for the Requirements Analysis activity. It is also in this phase that the Analysis and Design and Implementation activities begin to gain the largest share of the focus. The transition for this phase requires several requirements to be fulfilled. The first is the requirements from the Research activity.

In order to transition from Elaboration to Construction all of the major Research activity's work must be completed. The first item to be completed is the mathematical and algorithmic description and the justification behind selected metrics to be used. The next item is the selection and quantification of code smell heuristics. Finally, the selection of refactoring techniques and the graph manipulations to be used must be fully described.

The next activity to be completed in the phase is the Requirements Analysis activity. The major work items that must be completed for this activity is the identification and elaboration of all major requirements. In order to produce these the major use cases identified in the inception phase should be completed as well as textually and diagrammatically defined. From these use cases the functional, performance, interface, and architectural requirements should be completely identified and described. It is also within the purview of this activity that the work break down structure for the project should be defined and the schedule of work and work assignments should be set.

As the Elaboration Phase completes, the Analysis and Design and Construction activities should be gaining considerable focus. Prior to transitioning from this phase the Analysis and Design activity should produce UML Class diagrams describing the structure of the system, and this activity should also define any design patterns to be used during the construction phase. The Construction activities to be started are the generation of stub classes as defined in the UML Class diagrams. Along with these stub classes the minimal stub JUnit test files should be generated as well.

Currently we are in the process of completing the transition criteria and deliverables for this phase. We have completed the UML Class Diagrams and have begun implementing the initial classes for the system. We are still in the process of completing the Research activity, but we have generated the textual and diagrammatic representation of the major use cases. We have also finished the elaboration of the functional, performance, interface, and architectural requirements.

5.5.3 Construction Phase

The Construction Phase is a phase in which multiple micro-iterations routinely occur, where each iteration produces a deliverable version the program. The separate iterations have a milestone to be associated with a deliverable set of features for that release of the software. The main activities of this phase include Implementation and Testing. This phase also includes the Requirements Analysis and Research activities, but the primary utilization of these activities is to both continually maintain documentation entities that are within the purview of these activities and to adjust and modify the deliverables of these activities as changes to the design occur.

The Implementation activity intends to deliver the operational program at the transition point of this program. In order to meet this requirement each iteration of this phase requires the development of milestones for the iteration. In order to track and maintain these milestones we keep our documentation within revision control. Each milestone must be related to the uniquely numbered requirement that the milestone is to implement. As the end of an iteration is reached the Testing activity tends to gain more focus.

The Testing activity during construction requires that the project members have developed JUnit test cases for each class they implement. This activity also focuses on getting regression testing implemented and running each test case. This latter requirement should be implemented near the end of the first iteration of the Construction phase and should continually run during the rest of the project life cycle.

The transition criteria for leaving the Construction phase and entering the Transition phase is the following. The first is the complete implementation of all defined system requirements. Due to the nature of software engineering, requirements change, and this is covered by the continually active Research, Requirements, and Analysis and Design activities. During this phase the Analysis and Design activity will decrease in focus, but this activity is required to deliver completed UML Sequence diagrams for all major sequences in the system. It also requires that specifically for the underlying subsystems UML Activity Diagrams should be produced as well.

5.5.4 Transition Phase

The final phase is the Transition Phase in which the main activity is normally Deployment. This is also the phase in which all activities complete and all deliverables are due in completion prior to the next iteration of the entire life cycle. For this project there is the extra activity of developing the final paper and poster for the project. The Analysis and Design activity should wrap up as this phase starts, thus finalizing the documentation of the system, and simultaneously the final source code modification should be completed in the Implementation activity. The final steps of the Requirements Analysis should be to finalize and document any requirements that have remained only in informal communications. The last requirement is generation of the deploy-able build for this iteration of the software life cycle and prepare for the next iteration. The transition conditions and deliverables for this projection in the Transition phase are the completely developed system, the final documentation comprised of a final paper and poster, and a finalization of all requirements and UML documentation.

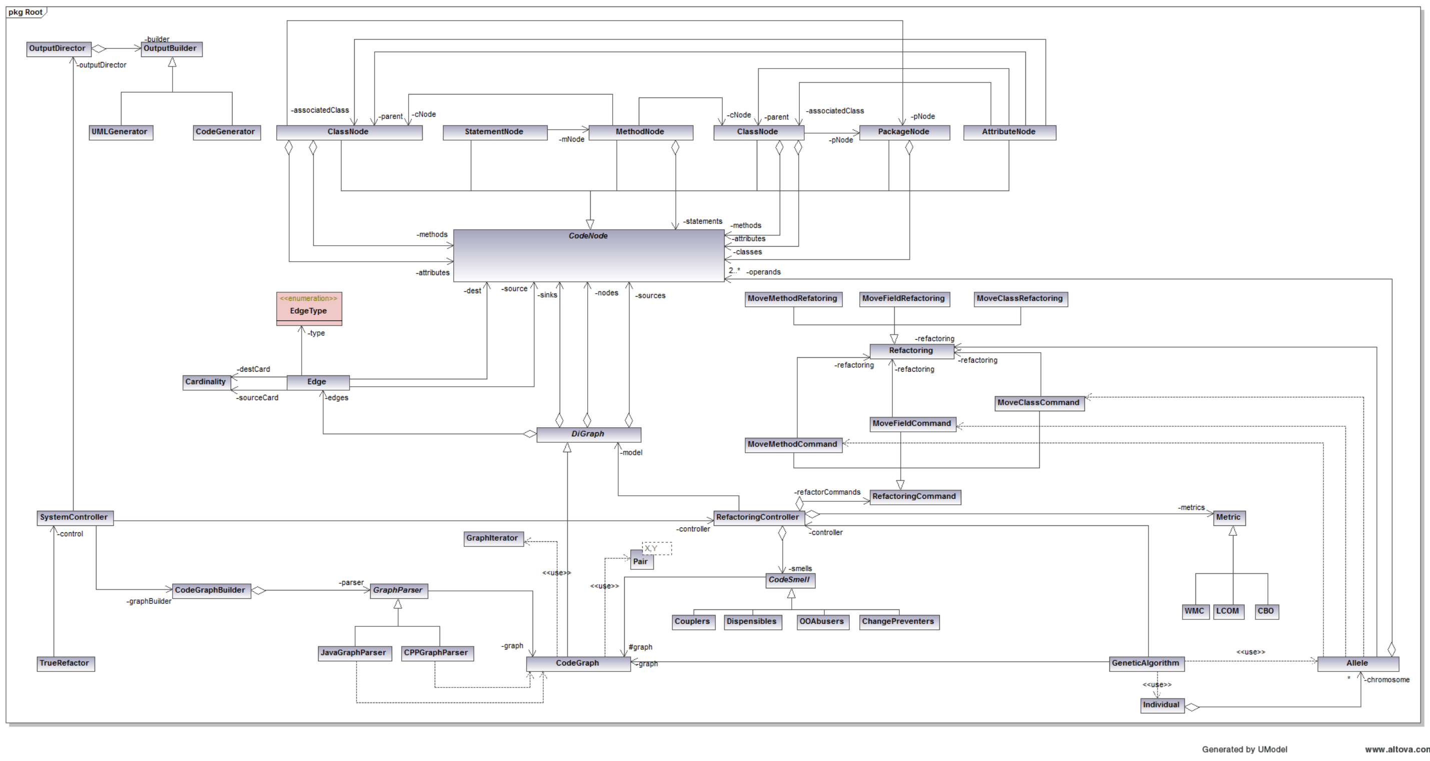
6 Methodology

This project is working within a pre-existing source code base. Given the problems discussed above with the original project, ERRS, and following the tenets of BUP, we have begun to reform the original process in order to support the additional features and goals of the current project. This new process will use the already existing framework which parses a provided code base into a graph data structure, and which provides the genetic algorithm to select and apply the refactorings to this code graph. We have decided the focus should be on the correct application of a set of metrics that have been designed to measure the understandability, reusability, and maintainability of the software. Given these metrics have been put in place, the goal is then to provide the program with a code base, measure the initial values and attempt to optimize the metrics. Once the genetic algorithm has converged (or reached a point where the amount of measured code smells is minimized), the code graph that is developed will reflect the completely refactored code and the program will be ready to produce a set of output documents. The first of this documentation will be a set of UML diagrams. Currently UML is a standard means of conveying not only the structure of a program but also a means of conveying the operation of the program. Coupling this information with the refactored program, which has been converted to optimize the structure of the code and reduce problem areas (as defined by code smells), we are then able to output the modified code. The final document to be output is a report of the refactorings that occurred and the values measured for each metric during each iteration of the genetic algorithm.

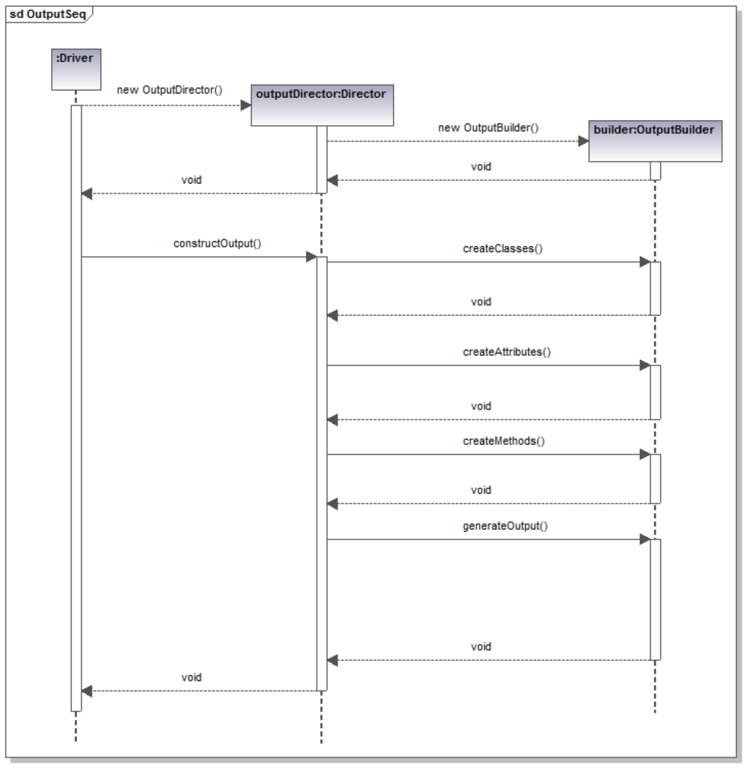
Using the describe process generated during the Inception Phase and the requirements generated during the Elaboration phase of the BUP we are following, we have developed the following UML diagrams (found in section 6.1). The first diagram provides a simplified view of the overall system structure. All subsequent class diagrams display all of the attributes and methods of each class. Additionally, where appropriate, sequence diagrams have been provided to clarify the underlying operation and interactions between objects of the system. After the Overall Structure Class Diagram the system is broken into 4 major portions. The first is the Output Subsystem (6.1.2), followed by the Input Processing Subsystem (6.1.3), the Refactoring Subsystem (6.1.4) and finally the Code Graph Data Structure (6.1.5).

6.1 UML Diagrams

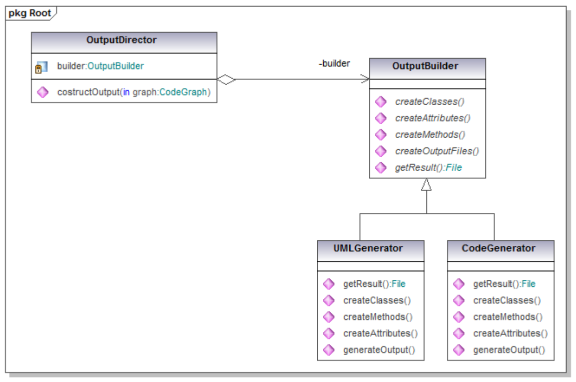
6.1.1 Overall Structure



6.1.2 Output Subsystem

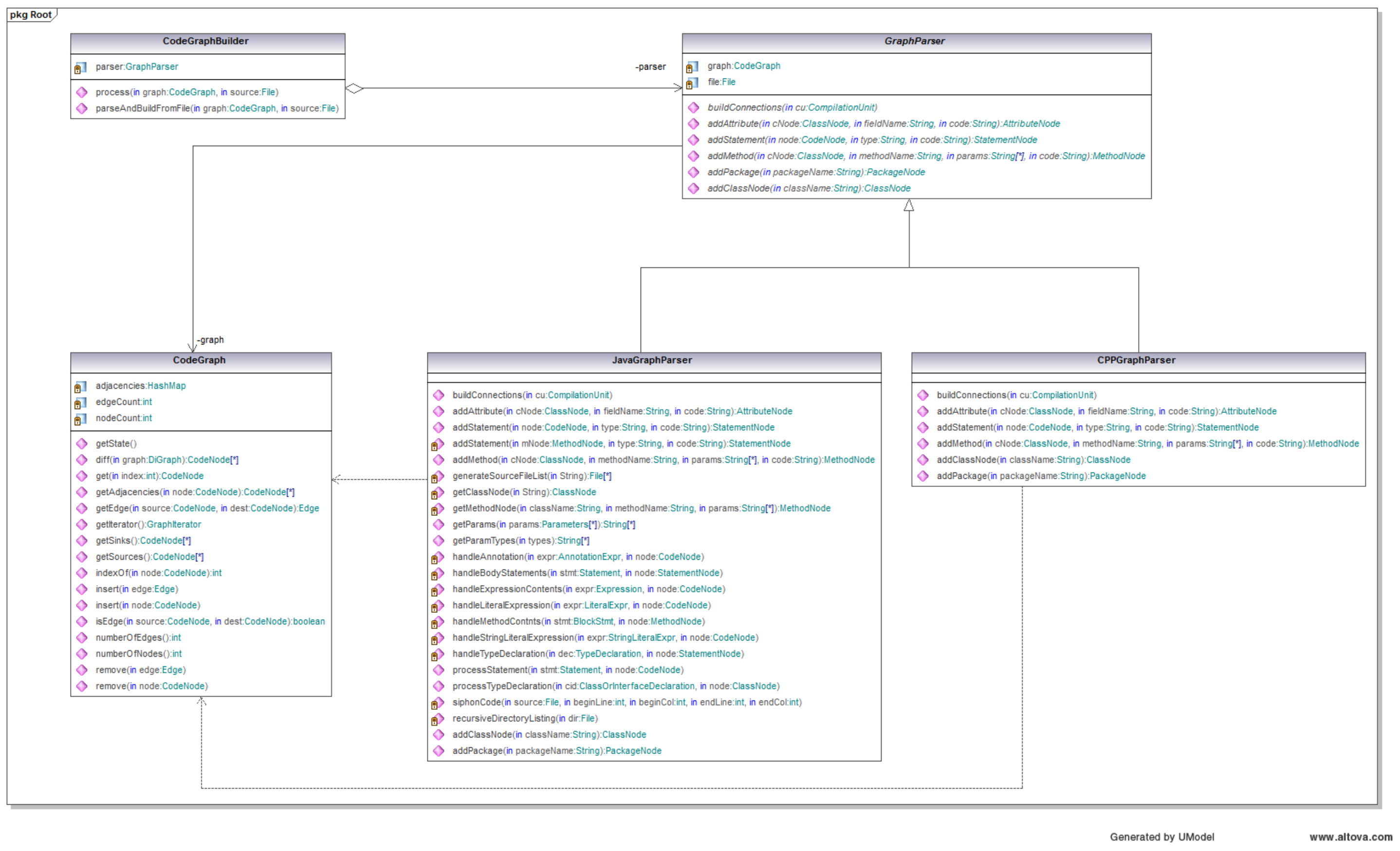


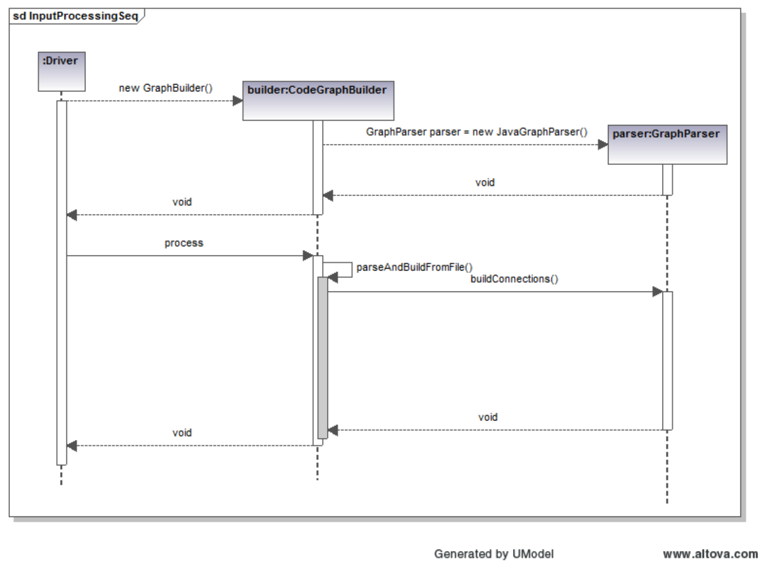
Generated by UModel www.altova.com



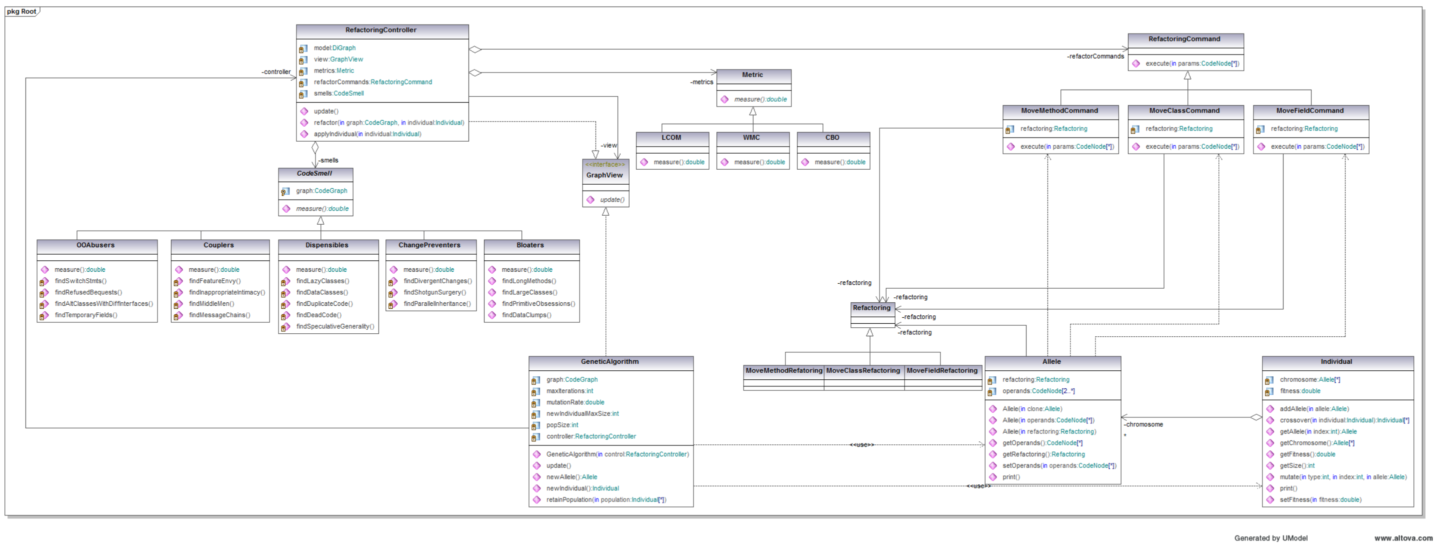
Generated by UModel www.altova.com

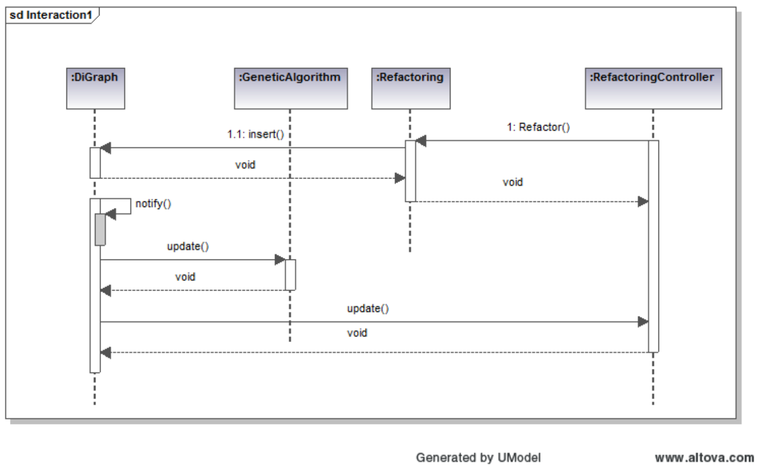
6.1.3 Input Processing Subsystem


pkg Root
CodeGraphBuilder
parser:GraphParser
process(in graph:CodeGraph, in source:File)
parseAndBuildFromFile(in graph:CodeGraph, in source:File)
-parser
GraphParser
graph:CodeGraph
file:File
buildConnections(in cu:CompilationUnit)
addAttribute(in cNode:ClassNode, in fieldName:String, in code:String):AttributeNode
addStatement(in node:CodeNode, in type:String, in code:String):StatementNode
addMethod(in cNode:ClassNode, in methodName:String, in params:String[*], in code:String):MethodNode
addPackage(in packageName:String):PackageNode
addClassNode(in className:String):ClassNode
-graph
CodeGraph
adjacencies:HashMap
edgeCount:int
nodeCount:int
getState()
diff(in graph:DiGraph):CodeNode[*]
get(in index:int):CodeNode
getAdjacencies(in node:CodeNode):CodeNode[*]
getEdge(in source:CodeNode, in dest:CodeNode):Edge
getIterator():GraphIterator
getSinks():CodeNode[*]
getSources():CodeNode[*]
indexOf(in node:CodeNode):int
insert(in edge:Edge)
insert(in node:CodeNode)
isEdge(in source:CodeNode, in dest:CodeNode):boolean
numberOfEdges():int
numberOfNodes():int
remove(in edge:Edge)
remove(in node:CodeNode)
JavaGraphParser
buildConnections(in cu:CompilationUnit)
addAttribute(in cNode:ClassNode, in fieldName:String, in code:String):AttributeNode
addStatement(in node:CodeNode, in type:String, in code:String):StatementNode
addStatement(in mNode:MethodNode, in type:String, in code:String):StatementNode
addMethod(in cNode:ClassNode, in methodName:String, in params:String[*], in code:String):MethodNode
generateSourceFileList(in String):File[*]
getClassNode(in String):ClassNode
getMethodNode(in className:String, in methodName:String, in params:String[*]):MethodNode
getParams(in params:Parameters[*]):String[*]
getParamTypes(in types):String[*]
handleAnnotation(in expr:AnnotationExpr, in node:CodeNode)
handleBodyStatements(in stmt:Statement, in node:StatementNode)
handleExpressionContents(in expr:Expression, in node:CodeNode)
handleLiteralExpression(in expr:LiteralExpr, in node:CodeNode)
handleMethodContnts(in stmt:BlockStmt, in node:MethodNode)
handleStringLiteralExpression(in expr:StringLiteralExpr, in node:CodeNode)
handleTypeDeclaration(in dec:TypeDeclaration, in node:StatementNode)
processStatement(in stmt:Statement, in node:CodeNode)
processTypeDeclaration(in cid:ClassOrInterfaceDeclaration, in node:ClassNode)
siphonCode(in source:File, in beginLine:int, in beginCol:int, in endLine:int, in endCol:int)
recursiveDirectoryListing(in dir:File)
addClassNode(in className:String):ClassNode
addPackage(in packageName:String):PackageNode
CPPGraphParser
buildConnections(in cu:CompilationUnit)
addAttribute(in cNode:ClassNode, in fieldName:String, in code:String):AttributeNode
addStatement(in node:CodeNode, in type:String, in code:String):StatementNode
addMethod(in cNode:ClassNode, in methodName:String, in params:String[*], in code:String):MethodNode
addClassNode(in className:String):ClassNode
addPackage(in packageName:String):PackageNode
Generated by UModel
www.altova.com

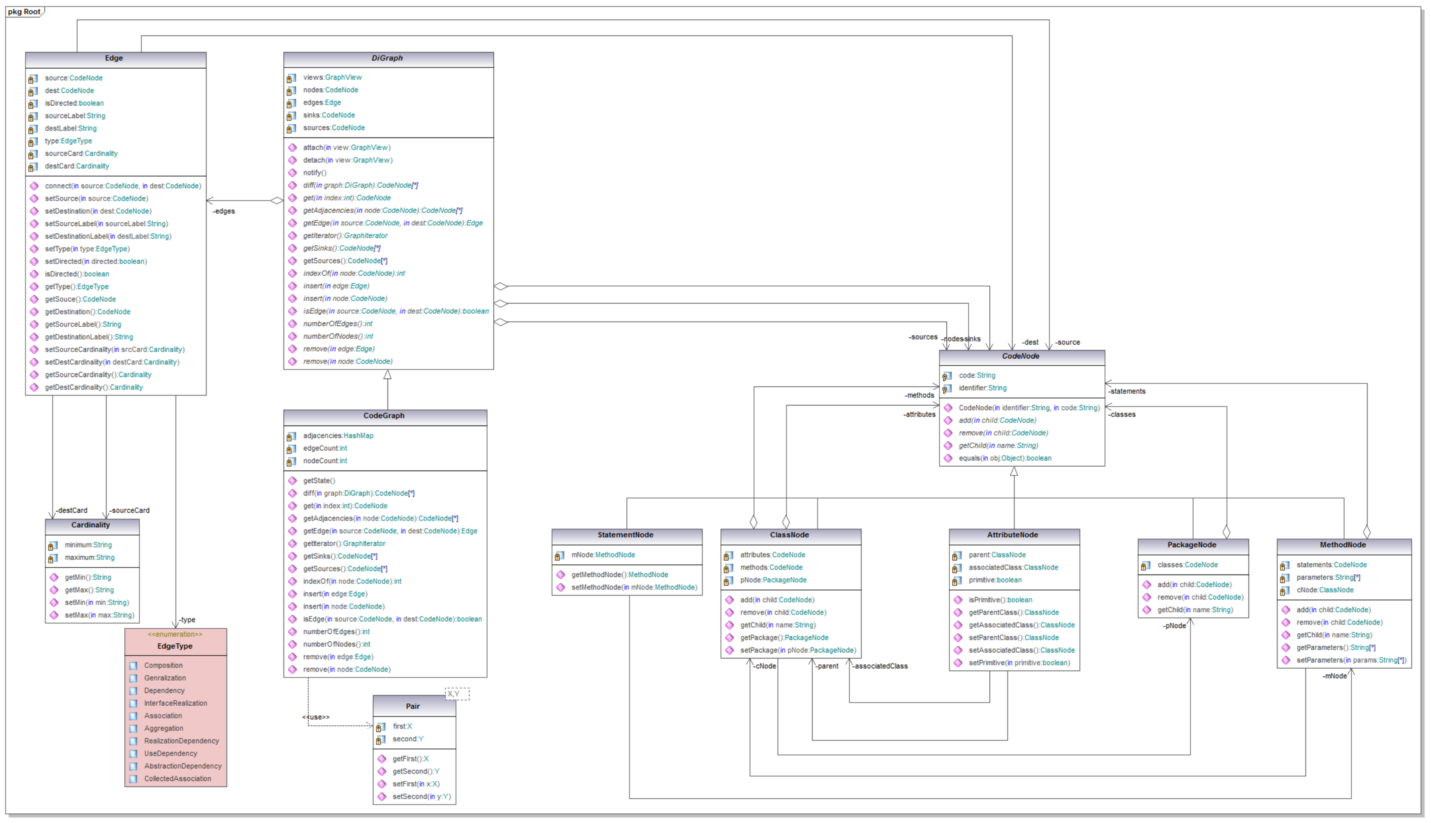


6.1.4 Refactoring Subsystem





6.1.5 Code Graph Data Structure



6.2 Design Patterns

In the design and analysis of the proposed system we have identified several design patterns to be used. The following section provides each pattern identified to be used, the subsystem in which the pattern is to be used, and the rationale for using the design pattern. For some of the identified patterns they are to be combined with other patterns to form the entire subsystem. In such cases both patterns are referenced and the connections between patterns is explained.

Composite Pattern: Used to define the Node inheritance structure for the Code Graph data structure. This specific pattern was selected due to the ability to treat nodes as both a leaf and a container of other nodes. The rationale behind this decision was that it would provide the required performance links necessary to minimize traversals of the graph structure attempting to find different types of nodes. It also provided a means to differentiate between types of nodes without requiring an extra attribute which would need to be queried every time we needed such information. Finally this structure provides an almost direct correlation to UML class diagram notation and will allow easy mapping into the EMF model hierarchy.

Builder Pattern: Used in both the input subsystem, to build the code graph, and in the output subsystem, to build either UML, generate Code, Generate a Metrics Report, or any combination thereof. The choice of this pattern for Input Processing System was used as a means to provided a common interface to the classes which interface with the JavaCC generated code parsing libraries. Unfortunately there is no way to know if the standard form the classes generated by JavaCC will be created, and so having a standard interface to the TrueRefactor system with specific known methods to generate a Code Graph was necessary. This also was selected in order to provide extensibility to the system and allow new languages to be added at a later time. This pattern was also selected for the Output System for similar reasons. In the output system we need to create various text files based on events that have occurred during the operation of TrueRefactor. Given that this information can be gathered from the System Controller we define a specific interface and extension point to produce a variety of output types using the same set of methods.

Iterator Pattern: Used with the Code Graph ADT in order to facilitate a means by which we can iterate over the nodes of the graph. The Iterator pattern was selected in order to provide a well defined Iterator interface to a set of traversal and iteration classes for the Code Graph, given that various types of traversals will be needed during input processing, output generation, refactoring, metric calculations, and measuring code smells.

Model-View-Controller (MVC) Pattern: Used in conjunction with both an Observer Pattern and a Strategy Pattern to define the Processing Subsystem which restructures the Code Graph. The MVC Pattern was selected in order to provide defined separation between the refactoring and genetic algorithm logic, the Code Graph itself, and the controlling logic for the interactions between them.

Observer Pattern: Used in conjunction with the MVC Pattern above in the Processing System. In the processing system the Observer Pattern provides the connection between the Model (the Code Graph) and both the Views (Processing Algorithm) and the Controller (Refactoring Control). This pattern was selected in order ensure that all of the view components and the controlling component of the MVC pattern are correctly updated when the Code Graph is modified, or replaced by a refactored clone.

Strategy Pattern: Used in both the Metrics and Code Smells sections of the design. This pattern was selected to provide an interchangeable and extensible means to measure the fitness of a set of refactorings. The same rationale applies for the Code Smells sections as well.

The choice of the above design patterns comes with many consequences. Yet, given the desire for extensibility, maintainability, reliability, etc. we have decided currently that any adverse consequences of the patterns present manageable risk. As design progresses we are maintaining enough flexibility to change or incorporate different design patterns as they are needed.

6.3 Design Tradeoffs

We have acknowledged two major design tradeoffs that will be present during this project. The first is the distinction between engineering the solution well and the time spent modifying and reusing already existing code. The second tradeoff is the distinction between time spent in analysis and design and the time spent during development. Each of these design tradeoffs and the solutions we have incorporated into the project are discussed below.

Time Spent on New Feature Engineering VS Time Spent on Modification and Reuse of Existing Code

Given that this research project is an extension of an existing system, we will eventually run into a design tradeoff between engineering additional functionality or modifying and reusing existing functionality. In order to circumvent this tradeoff and maximize our time, we have incorporated the refactoring and reuse of existing code as a part of engineering the new features. This is accomplished by consistently modifying existing classes and features while adding new classes and features. Attacking this tradeoff in such a way has managed to reduce the number of iterations in the initial design portion of the Elaboration phase.

Analysis and Design Time VS Development Time

In order to ensure that a working solution is developed and a fair balance between design and analysis was maintained, we selected a specific software development process that not only fit our small team size, but also fit the type of project we are working on. This process has helped us develop, in the elaboration phase, the above work breakdown structure, which will help to ensure that we meet our deadlines. This helps to remain flexible enough to allow adequate time for analysis and design while still providing the time required to produce a working solution.

7 Expected Results

Once development is complete we expect to have a tool which can be used to convert legacy code to today's standards. We expect that this tool will be able to accurately generate an optimal solution which increases the maintainability and reusability of legacy software. Simultaneously this tool should also increase the understandability of this legacy software project's code base for developers who are new to the project or have been tasked with adding new features. This expectation is achieved through the use of refactoring techniques which have been selectively applied via a Genetic Algorithm. We also expect that the metrics developed in this project, which measure characteristics of the source code which directly correlate to the aforementioned behavioral aspects of understandability, reusability, and maintainability, will preserve a clear distinction between the structural complexity of the source code and the psychological complexity from a human perspective. In addition to the metrics we also expect to have a set of quantitative heuristics developed to measure code smells. These quantitative heuristics will provide a tool which will aid developers in both understanding the concept of code smell as a measurable characteristic of source code, and they will provide a specific range of a measure prescribing when to apply the refactorings associated with particular code smells. Quantifying code smells will provide a new path in the world of automated software engineering, and this could provide a base from which development of new tools can be formed. With the combination of both the metrics and the quantitative code smell heuristics we should have the ability to start providing a generalized mathematical framework to define the properties which characterize the concepts used in these measurements. Finally, this project also provides an approach to automate the application of refactoring techniques, when to apply this automation, and a general framework for implementing and extending the refactoring presented. Thus, it provides an example and a base for developing a specific library of refactorings to be used in later developed tools.

References

[1] Fowler, Martin. *Refactoring: Improving the Design of Existing Code.* New York: Addison Wesley Longman, Inc, 1999. 1-407. Print

[2] Griffith, Isaac, Scott Wahl, and Ben Darling. "Epistemic Retension Refactoring System (ERRS)." (2010): 1-6. Print.

[3] Janlert, Lars-Erik. "Dark Programming and the Case for the Rationality of Programs." *Journal of Applied Logic* 6 (2008): 545-52. Print.

[4] Russel, Stuart, and Peter Norvig. *Artificial Intelligence A Modern Approach.* 3rd ed. Upper Saddle River, NJ: Pearson Education Inc., 2010. 120-33. Print.

[5] "Unified Modeling Language." *Wikipedia.* WikiMedia Foundation, 5 Nov. 2010. Web. 9 Nov. 2010. <http://en.wikipedia.org/wiki/Unified_Modeling_Language>.

[6] "OpenUP/Basic." *Wikipedia.* WikiMedia Foundation, 9 Mar. 2010. Web. 9 Nov. 2010. <http://en.wikipedia.org/wiki/Basic_Unified_Process>.