ADDIS ABABA UNIVERSITY SCHOOL OF GRADUATE STUDIES

COLLEGE OF NATURAL SCIENCES

FUCULTY OF COMPUTER AND MATHEMATICAL SCIENCES

DEPARTMENT OF COMPUTER SCIENCE

Enhanced Design and Implementation of Code Generator for Model-driven Software Development

By

Melese Erku

A Project paper submitted to the School of Graduate Studies of Addis Ababa University in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science

June, 2011

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

FUCULTY OF COMPUTER AND MATHEMATICAL SCIENCES

DEPARTMENT OF COMPUTER SCIENCE

Enhanced Design and Implementation of Code Generator for Model-driven Software Development

By

Melese Erku

Name and signature of the examining board:

1. Dr. Dida Midekso (Advisor) _____
2. _____ _____
3. _____ _____

Acknowledgements

First and foremost, I would like to express my deepest sense of gratitude to my advisor Dr. Dida Midekso for his guidance, encouragement and advice throughout the project.

I am also very thankful to my friends Haftom G/egziabher, Mekdes Kenito and Muhdin who have been with me through their fruitful and valuable suggestions and support.

Finally, I take this opportunity to express my profound gratefulness to my families for their patience during my study. Specially, I thank my sister Aregash Erku for her constant moral and advice throughout my life.

Table of Contents

List of Tables

List of Figures

Acronyms

SDLC.....Software Development Life Cycle
MDA.....Model Driven Architecture
MDD.....Model Driven Development
UML.....Unified Modeling Language
OMG.....Object Management Group
MOF.....Meta Object Facility
XMI.....XML Metadata Interchange
CWM.....Common Warehouse Metamodel
CIM.....Computational-Independent Model
IDE.....Integrated Development Environment
MVC.....Model View Controller
PIM.....Platform Independent Model
PSM.....Platform Specific Model

Abstract

Software development passes through clear phases of engineering called Software Development Life Cycle (SDLC). In this development process, transformation between consecutive phases should be performed in order to reach the last output, the executable artifact. However, manual transformation is very time and resource intensive task.

An automated transformation between models based on an increased reuse of models with the use of MDA costs low, increases productivity and makes maintenance cheap. A project entitled "Design and Implementation of Code Generator for Model-driven Software Development" was conducted in 2010 that aims at automated transformation of model (Class Diagram) into a source code and the integration of the changes made to the source code into the model so that the two artifacts are consistent[5].

The extension of the work in [5] is needed to make it more helpful in model-driven software development. Including additional UML diagrams that have important role in modeling and code generation has valuable contribution in modeling a software system to make more reasoning on it through the addition of relevant details. Accordingly a major UML diagram is added.

Keywords: MDA, Code Generator, Model

CHAPTER ONE: INTRODUCTION

1.1. Background

The notion of software engineering was proposed in 1968 at a conference held to discuss the difficulties and pitfalls of developing complex software systems [1]. It is a discipline that deals with the theories, methods and tools which are needed to develop high quality software systems. Software development passes through clear phases of engineering called Software Development Life Cycle (SDLC). The phases of SDLC in the order they are performed are requirement elicitation, requirement analysis, system design, implementation, deployment and maintenance.

Each of the SDLC phases has a concrete development output, called development artifact. The artifacts may be documentation, modeling or an executable program depending on the phase in which the artifact is gained. Requirement elicitation phase is characterized by its documentation artifact whereas the implementation phase of the SDLC has documentation and executable artifacts, the user manual and the final output of the development process that is the executable artifact. Even if all the artifacts are important and necessary, the modeling artifact and the source code artifact are more important in the development process [6]. In this software development process, transformation between two consecutive phases is made using the artifact of the previous phase as input for the next phase.

There is a practice of representing software using models, existed even in the early days of software development, however, the early days of using models were highly informal, not standardized and manual [7, 8]. The Unified Modeling Language (UML), which is the current generation of software modeling, is a standard tool in object-oriented software development approach.

The demand for software continues to rise. Moreover, the complexity of software development increases as the requirements of software systems rise with the advent of high performance computing devices. So that software development becomes time and resource intensive task. Thus, high amount of cost is spent by organizations for software they need to have.

Based on an increased reuse of models costs become lower, productivity becomes higher, and maintenance becomes cheap with the use of Model Driven Architecture (MDA) in software development [2]. The core concept of MDA is creating different models at different levels of abstraction and then linking them together to form an implementation. Some of these models will exist independent of platforms, while others will be specific to particular platforms. Each model will be expressed using a combination of text and multiple complementary and interrelated diagrams. Software development approach based on models and transformations between models is known as model-driven development (MDD) [3].

Models are a set of elements that provide abstractions of a physical system that allow software developers to reason about that system by ignoring details of not interest while focusing on the relevant ones. Once the models are constructed completely, they have greater longevity than code because they evolve independently of other models, that is, they become assets. So, they can be reused when developing new software systems and make maintenance of the existing system easier.

In object-oriented software development the primary modeling notation used is the UML which is a standard by Object Management Group (OMG). UML as a modeling language allows capturing a variety of important characteristics of a system with its thirteen diagrams, each with different semantics and syntax.

Code generation is about transforming a higher level model (the system design model) into a target model that is the actual implementation code. Transforming the design model into the implementation code manually consumes high amount of development time and resource, moreover, it introduces inconsistency between the model and the source code. Rather tools can be used to manage traceability and dependency relationships among model elements according to best practice guidance on maintaining synchronized models as part of a large-scale development effort [3]. There are commercial and open source tools that are used in model-driven software development. To list some, MagicDraw and ArcStyler are proprietary while AndroMDA is an open source model-driven software development tool.

A design and implementation of code generator for model-driven software development was conducted taking into consideration model construction and code generation [5]. This code

generator uses class diagrams of a system to generate the structural code. Moreover, the code generator was designed in such a way that reintegration of modifications made to the generated source code into the design model is possible; as a result the model and the source code become consistent.

1.2. Statement of the Problem

Some of the models of a software development have a tendency of growing, for example the use case model and the class diagram model (which is the main victim of being very large). This may lead to unmanageable and very complex models to understand during the development of huge and complex software systems. The UML has a solution through one of its modeling constructs which is the package diagram.

Better modeling can be achieved by utilizing the package diagram in order for the class diagrams to be organized and grouped. Package diagrams are relevant in visualizing the large-scale organization of the software in modeling which makes a good logical road map of the system and the compile time grouping of the classes and the relationships between the groups [4]. The grouping of the class diagrams may be based on whether the classes in a package should need changing for similar reasons or should all be reused together. Both way of grouping classes are very important in developing software systems to reduce complexity and increase reusability.

The extension of the work in [5] is needed to make it more helpful in the model-driven software development. Including additional UML diagrams that have important role in modeling and code generation has valuable contribution in modeling a software system to make more reasoning on it through the addition of relevant details.

Extension of the design of the code generator is also important in such a way that additional modeling diagrams can be used to increase the realm of code generation even if it is difficult to generate code from behavioral diagrams and some are completely not related to the code generation. This decreases the manual transformation of the code from the model.

1.3. Objective of the Project

1.3.1. General Objective

The general objective of the project is to enhance and extend the UML based design and implementation of the code generator for model-driven software development.

1.3.2. Specific Objectives

The specific objectives of the project are:

- To conduct a review of the UML and MDA specifications
- To develop an extended and improved UML design environment
- To develop an extended and enhanced UML based code generator algorithm
- To develop a prototype of the system

1.4. Scope and Limitation

The functionalities of a general Model Driven Architecture tool may encompass model construction, model verification, model execution, transformation to intermediary models, code generation and reverse transformation [5]. Even if all these are necessary, due to the time and resource constraints the scope of the project is limited to the model construction and code generation.

Some of the diagrams are not exactly known how well they will work out in practice and some are very complex in code generation since they have no predefined and restricted rule of design [4]. Due to the time and resource constraints, and the relevance of the diagrams, the model construction is limited to only enabling to use package diagrams in addition to the three core diagrams considered in the previous project work. Database modeling is also excluded from the project.

The code generator is again limited to the structural model diagrams, class diagram and package diagram. And it is also limited to output code to a single programming language.

1.5. Application of the Project

The result of this project is used in software development process. This makes the developer beneficiary of designing the UML model of a system and generating the structural code of the system from the UML model.

1.6. Document Organization

This document has six chapters including this chapter. The literature review is presented in chapter two. In the third and fourth Chapters, the analysis and design according to the object-oriented methodology are presented respectively. The implementation details are discussed in the fifth Chapter. Finally conclusion and recommendations are given in the last Chapter.

CHAPTER TWO: LITERATURE REVIEW

2.1. Significance of Model Automation

With the increase in the software demand and the rise of the requirements of software systems with the advent of high performance computing devices, the complexity of the software development increases. Developing these complex systems leads to the consumption of high amount of time or late delivery of the software product. Moreover, the quality of the software may be low.

Implementing a software system manually based on a design provided in modeling diagrams helps to separate major decisions about the architecture, structure and behavior of a software system from implementation specific details. Though, the transition made from the design model to the implementation becomes a bottleneck since high amount of development time is spent dealing with this transition [5]. It also introduces inconsistencies between the model and the implementation code. Moreover, it is difficult to reuse development artifacts when there is a platform difference. This shows that the manual transformation between models use the development artifacts as documentation only. As a result, an automated implementation of a software system is very important in reducing development time, improving the quality of the software system, easing the reusability of the modeling artifacts, and keeping the models consistent with each other.

2.2. The OMG Solution: MDA and UML

2.2.1. The UML

There have been different graphical modeling languages in the software industry for a long time. The rationale towards these languages is that programming languages are not high level enough to ease the communication among the developers and between the developer and the clients. In object oriented software development, the UML is used to model software systems which is a collection of graphical notations and specified by the OMG. The UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software system. It offers a standard way to write a system's blueprints, including conceptual things such as business

processes and system functions as well as concrete stuffs such as programming language statements, database schemas, and reusable software components. It is used by developers to facilitate the communication of some aspects of a system and more importantly in forward-engineering and reverse-engineering [4]. There are different versions of the UML. This work focuses on the UML 2.0 which is a stable specification and since the previous work is based on it.

UML 2.0 has 13 diagrams that are grouped into structure diagram and behavior diagram. The structure diagram includes class diagram, component diagram, composite structure diagram, deployment diagram, object diagram and package diagram. On the other hand, activity diagram, use case diagram, state machine diagram, sequence diagram, communication diagram, interaction overview diagram and timing diagram are grouped as behavior diagrams. Among these diagrams, three diagrams; class diagram, use case diagram and sequence diagram build the core of the UML. The package diagram is very important in grouping the class diagram.

Package Diagram

Package Diagrams are used to reflect the organization of packages and their elements. When used to represent class elements, package diagrams are used to provide a visualization of the namespaces. The most common use for package diagrams is to organize use case diagrams and class diagrams, although the use of package diagrams is not limited to these UML elements. Elements contained in a package share the same namespace. This sharing of namespace requires the elements contained in a specific namespace to have unique names. Packages can be built to represent either physical or logical relationships. When choosing to include classes to specific packages, it is useful to be based on the following cases:

- To assign classes with the same inheritance hierarchy to the same package
- To assign classes that belong to the same functionality to the same package
- To assign classes that have the aggregation / composition relationship with each other to the same package

2.2.2. The MDA

MDA is a style of software development by which generic rules are defined for automating many of the steps needed to transform one model representation to another for tracing between model elements, and for analyzing important characteristics of the models. It is being considered in the software industry today as a way of increasing the quality, efficiency, and predictability of large-scale software development [3]. It is a standard specified by the OMG. There are a number of standards that are also specified by the OMG to support the model driven architecture. These are UML, MOF, XMI, and CWM. The current project focuses on the UML.

The OMG has defined four layers and transformations that provide a conceptual framework and vocabulary for supporting MDA: Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) described by a Platform Model (PM), and Implementation Specific Model (ISM).

2.3. The Previous Work

The previous work [5] was carried out taking into consideration the three UML diagrams; class, use case and sequence diagram. The code generation is designed through which structural code from class diagram is generated. Moreover, it has the ability of the reintegration of the modification of the generated code into the design model of the system being developed. However it is limited to the three UML elements in model construction and to class diagram only in code generation.

The UML is a visual language for specifying, constructing, and documenting the development artifacts of a software system [9]. This implies that the different UML diagrams are there for the purpose of describing various facets of the model of a software system. For the code generation to cover more structural code, the package diagram is relevant. As a result, the extension of the model construction and the enhancement of the code generation and reintegration are done by including the package diagram in this project.

CHAPTER THREE: ANALYSIS

The functional requirements of the system are presented in this section. The analysis models are also shown in detail.

3.1. Functional Requirements

The following are the functional requirements of the system.

- Provide an extended and enhanced environment to design graphical UML diagrams
- Store and retrieve the models designed by the user
- Transform the details of the models into code with improved and extended domain of the code generation
- Integrate changes made in code to the models

3.2. Use-case Model

There is only one actor named “Developer” that interacts with the system. The use case model of the improved and extended system is the same as that of the previous system, but with some changes on the flow of events of some use cases. Figure 3.1 shows the use case model of the system.

Use Case Diagram

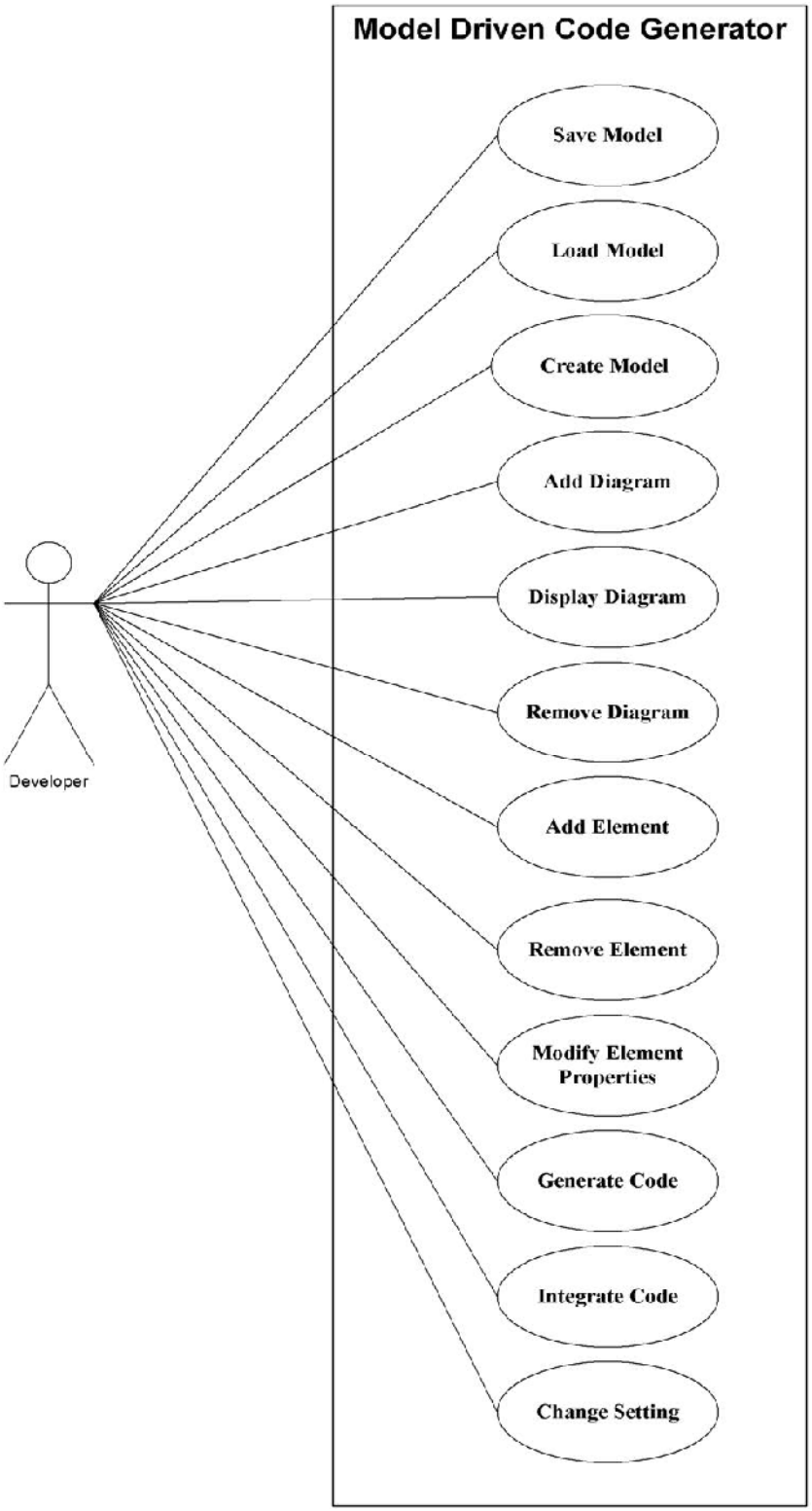

Figure 3.1: Use case diagram of the system

Use Case Descriptions

The use cases of the system are save model, create model, load model, add diagram, display diagram, remove diagram, add element, remove element, modify element property, generate code, integrate code and change setting [Figure 3.1]. The description of the use cases are presented in tables 3.1 – 3.12. The use case descriptions include the following terms:

- **Element** – a representation of a single part of a software system such as a class, use case, actor, object or package. An element may have properties such as location, name, related element… etc. An element belongs to the entire model, and can be included in multiple diagrams through an intermediate diagram element for each diagram.
- **Diagram** – a set of diagram elements and relationships among these elements that describes some aspect of the software system.
- **Model** – the entire set of diagrams and elements which describe the software system. Not more than one model is active in the system at any given time.
- **Diagram list** – a visible list of diagrams contained in a single model.
- **Diagram Editor** – a visual environment for displaying and modifying diagram elements, a diagram is active when it is displayed in the diagram editor.

Table 3.1: Description of the Save Model use case

Use case name	Save Model
Description	Stores the model in permanent storage
Actor	Developer
Pre-condition	A model must be either created or loaded
Flow of Events	i. The user clicks on the save button ii. The system prompts the user for the file location iii. The user selects(enters) a file location [Alternate A] iv. The system writes the model data to a

	file at the specified location
Post-condition	A file representing the model is stored
Alternate A	If the user enters an invalid file location **A.1** The system shows an error dialog **A.2** The system resumes at step iii.

Table 3.2: Description of the Load Model use case

Use case name	Load Model
Description	Loads a model from permanent storage
Actor	Developer
Pre-condition	A file representing the model must exist in local storage
Flow of Events	i. The user clicks on the load button ii. The system prompts the user for the location of the file iii. The user selects(enters) a file location [Alternate A] iv. The system reads data from the file and creates the model in memory v. The system displays the diagram list of the model
Post-condition	A list of diagrams within the model is displayed to the user
Alternate A	If the user enters an invalid file location **A.1** The system displays an error dialog **A.2** The system resumes at step iii.

Table 3.3: Description of the Create Model use case

Use case name	Create Model
Description	Allows the user to start editing a new model
Actor	Developer
Pre-condition	The system must be running
Flow of Events	i. The user clicks on the new model button ii. The system clears the diagram list and the diagram editor iii. The system enables model editing commands
Post-condition	Editing commands are available

Table 3.4: Description of the Add Diagram use case

Use case name	Add Diagram
Description	Adds a new diagram to the model
Actor	Developer
Pre-condition	A model must be either created or loaded
Flow of Events	i. The user clicks on a button indicating the type of diagram ii. The system adds a new diagram to the model iii. The system displays the diagram name in the diagram list
Post-condition	A diagram is added and its name is shown in the diagram list

Table 3.5: Description of the Display Diagram use case

Use case name	Display Diagram
Description	Shows the diagram elements in the diagram editor and enables diagram editing commands
Actor	Developer
Pre-condition	A model(containing at list one diagram) must be either created or loaded
Flow of Events	i. The user selects a diagram name from the list and clicks on the display button ii. The system retrieves the diagram from the model using the selected name iii. The system displays (enables) diagram editing commands
Post-condition	The elements of the diagram are shown in the diagram editor and the diagram editing commands are available

Table 3.6: Description of the Remove Diagram use case

Use case name	Remove Diagram
Description	Removes a diagram (and all its elements) from the model
Actor	Developer
Pre-condition	A model must be either created or loaded and the diagram list must be displayed with at list one diagram name
Flow of Events	i. The user selects a diagram name from the list and clicks on the remove button ii. The system removes the specified diagram from the model iii. The system updates the displayed diagram list iv. If the diagram is active the diagram editor is cleared
Post-condition	The diagram is removed from the model and the diagram list

Table 3.7: Description of the Add Element use case

Use case name	Add Element
Description	Adds an element to the active diagram and displays it in the diagram editor
Actor	Developer
Pre-condition	A diagram must be active
Flow of Events	i. The user clicks on a toolbar button indicating the type of element ii. The system creates a new element in the model iii. The system creates a new diagram element in the diagram iv. The system shows the diagram element in the diagram editor
Post-condition	The new element is shown in the diagram editor

Table 3.8: Description of the Remove Element use case

Use case name	Remove Element
Description	Removes an element from a diagram
Actor	Developer
Pre-condition	A diagram(containing at least one element) must be active
Flow of Events	i. The user selects a diagram element and clicks on the remove button ii. The system removes the diagram element from the diagram iii. The system removes the diagram element from the diagram editor
Post-condition	The element is removed from the diagram and no longer displayed in the diagram editor

Table 3.9: Description of the Modify Element Properties use case

Use case name	Modify Element Properties
Description	Changes the properties of an element
Actor	Developer
Pre-condition	A diagram (containing at least one element) must be active
Flow of Events	i. The user selects a diagram element [Alternate A,B,C] ii. The system displays the diagram element properties in a property editor iii. The user enters a new value for the desired property iv. The system changes the element's property v. The system updates the diagram display
Post-condition	The element property is changed and the diagram editor display is updated
Alternate A	If the user selects a diagram element containing a class element and clicks on edit **A.1.** The system displays the class element properties in a class editor dialog **A.2.** Resume Normal Flow at step iii.
Alternate B	If the user selects a diagram element containing a message element and clicks on edit **B.1**. The system displays the message element properties in a message editor dialog **B.2.** Resume Normal Flow at step iii.
Alternate C	If the user selects a diagram element containing a package element and clicks on edit **C.1**. The system displays the package element properties in a message editor dialog **C.2.** Resume Normal Flow at step iii.

Table 3.10: Description of the Generate Code use case

Use case name	Generate Code
Description	Generates source code from the model and stores it in a file
Actor	Developer
Pre-condition	A model must be either created or loaded
Flow of Events	i. The user clicks on the generate code button ii. The system displays a dialog prompting the user for a file location iii. The user selects (enters) a folder location and clicks on generate code button [Alternate A] iv. The system uses the information contained in the model to generate source code v. The system writes the code to files at specified location vi. The system displays a message dialog indicating the completion of the process
Post-condition	A file is saved containing the generated code
Alternate A	If the user enters an existing folder name or location and clicks on generate button **A.1.** The system displays an error dialog **A.2.** Returns to the dialog ii.

Table 3.11: Description of the Integrate Code use case

Use case name	Integrate Code
Description	Reads code from files and integrates it into the model
Actor	Developer
Pre-condition	A model must be either created or loaded
Flow of Events	i. The user clicks on the integrate code button ii. The system displays a dialog prompting the user for a folder

	location iii. The user selects (enters) a folder location and clicks on integrate code button [Alternate A] iv. The system reads the code from the file in the selected folder v. The system uses the information contained in the source code to update the model vi. The system displays a message dialog indicating the completion of the process
Post-condition	The modification in the source code file is applied on the model
Alternate A	If the user enters an invalid location and clicks on integrate button **A.1.** The system displays an error dialog **A.2.** Returns to the dialog ii.

Table 3.12: Description of the Change Setting use case

Use case name	Change Setting
Description	Changes setting used in the code generation and integration functionality
Actor	Developer
Pre-condition	A model must be either created or loaded
Flow of Events	i. The user clicks on the setting button ii. The system displays a list of settings in the setting dialog iii. The user enters changes to the desired settings iv. The system updates the model with the changed setting v. The system updates the display with visible changes (e.g. model name)
Post-condition	The settings are changed

3.3. Structural Model (Class Diagram)

The class diagrams of the system with the additional classes (PackageElement and DepenedencyElement) are presented in Figures 3.2 – 3.5. The attributes in uppercase are the newly added attributes. Figure 3.2 shows the basic structure of a model followed by the description of the classes.

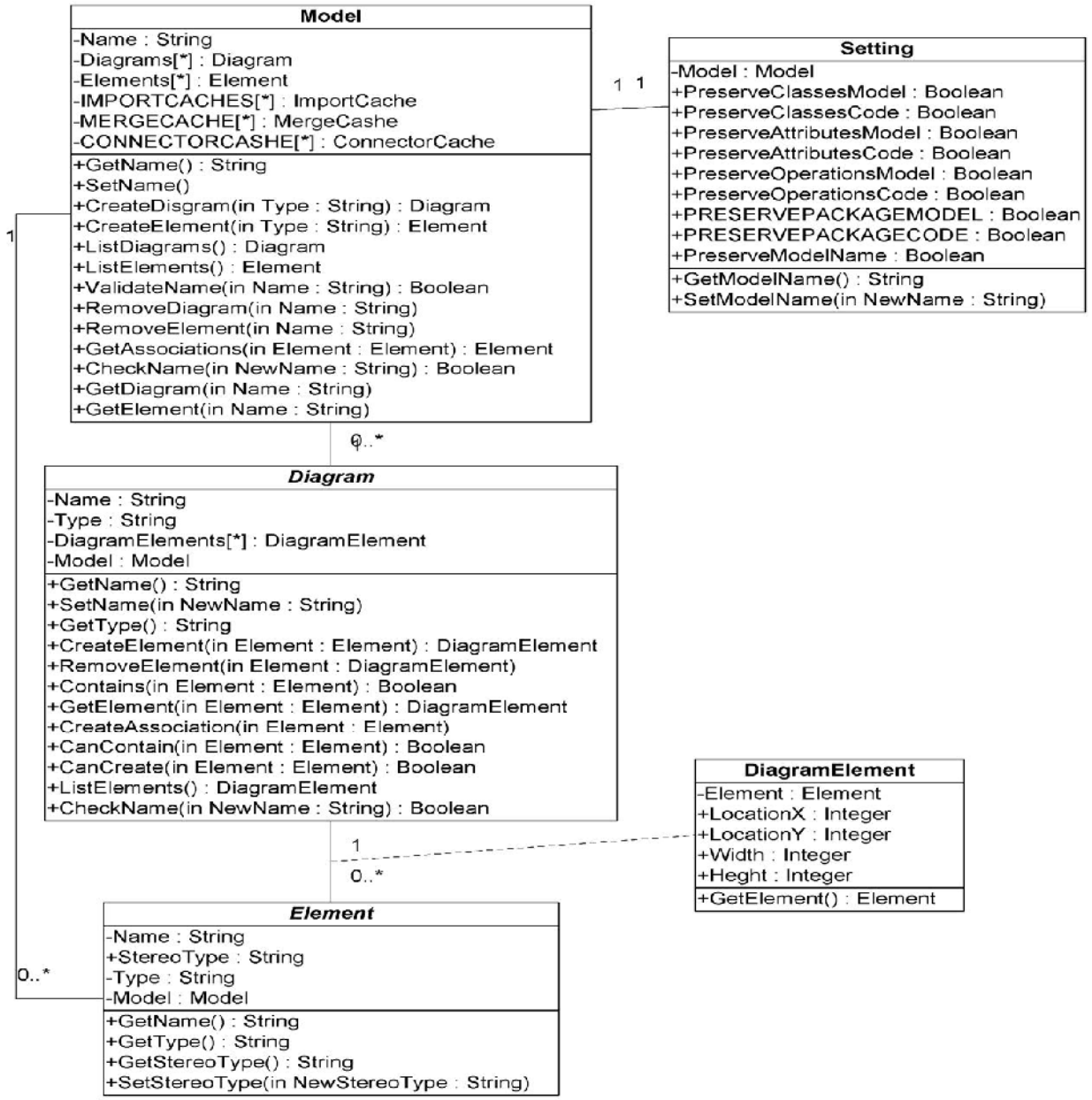

Figure 3.2: Class Diagram (Part 1)

Model – is the main class of the system which indirectly includes objects of all other classes. A model object includes one settings object and arrays of diagrams and elements. Model maintains uniqueness of diagram and element names. A diagram or element object can only be added to a model at creation by using the CreateDiagram and CreateElement operations of the model. In this class ImportCaches, MergeCaches and ConnectorCaches attributes are added. The methods UpdateCach, RemoveRelations, CreateElement, CreateDiagram and AddElement are modified so that the Package diagram can be created and the package elements can be created and added.

Settings – stores all the settings of a model used during code generation and integration operations on a model. This class is modified so that it also helps to incorporate model settings related to package diagrams. As a result attributes PreservePackagesCode and PreservePackagesModel are added.

Diagram – represents a UML diagram and contains an array of DiagramElements and indirectly contains objects of Element type. The Type property of this class set at creation and never changes. Type can be: Use Case, Class, Sequence or Package.

Element – represents the elements that make up a diagram. An element can be independent or represent a relationship between other elements. The Type property of this class is set at creation and never changes. Type can be: Actor, Use Case, Class, Communication, Generalization, Object, Message, Call, Return, Package or Dependency.

DiagramElement – represents an Element within a diagram, and stores information about its location and relations to other elements in the context of one diagram.

Figure 3.3 shows the different subtypes of DiagramElement and their helper classes followed by the description of each of the classes.

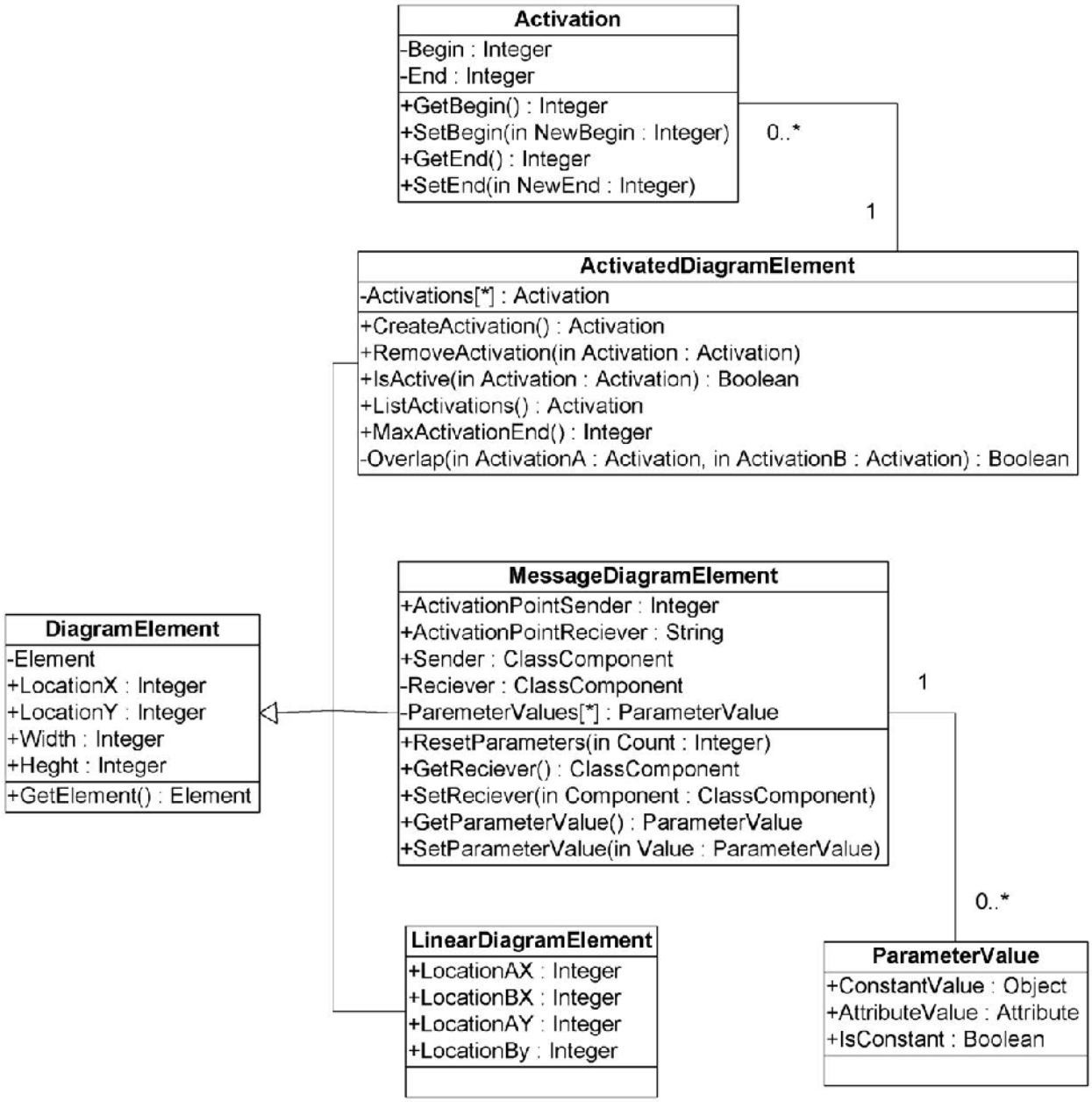

Figure 3.3: Class Diagram (Part 2)

LinearDiagramElement – a subtype of DiagramElement which adds information about two locations, typically of a line-type AssociationElement like communication or generalization.

ActivatedDiagramElement – a subtype of DiagramElement which adds information about the activation sections of elements in a sequence diagram. It also ensures there is no overlap between activation sections.

Activation – a helper class which stores the start and end of activation section. It also ensures the start is always before the end of an activation section.

MessageDiagramElement – a subtype of DiagramElement representing a message between objects in a sequence diagram. It adds information about the sender and receiver of a message and contains an array of parameter values if the message is an invocation of an operation.

ParameterValue – a helper class which stores values of a single parameter when its parent operation is the receiver of a message. It can contain either a constant value or a reference to an attribute of the message sender. Constant values must belong to one of the basic types represented by ObjectType.

Figure 3.4 shows the different subtypes of Element and their helper classes followed by the description of each of the classes.

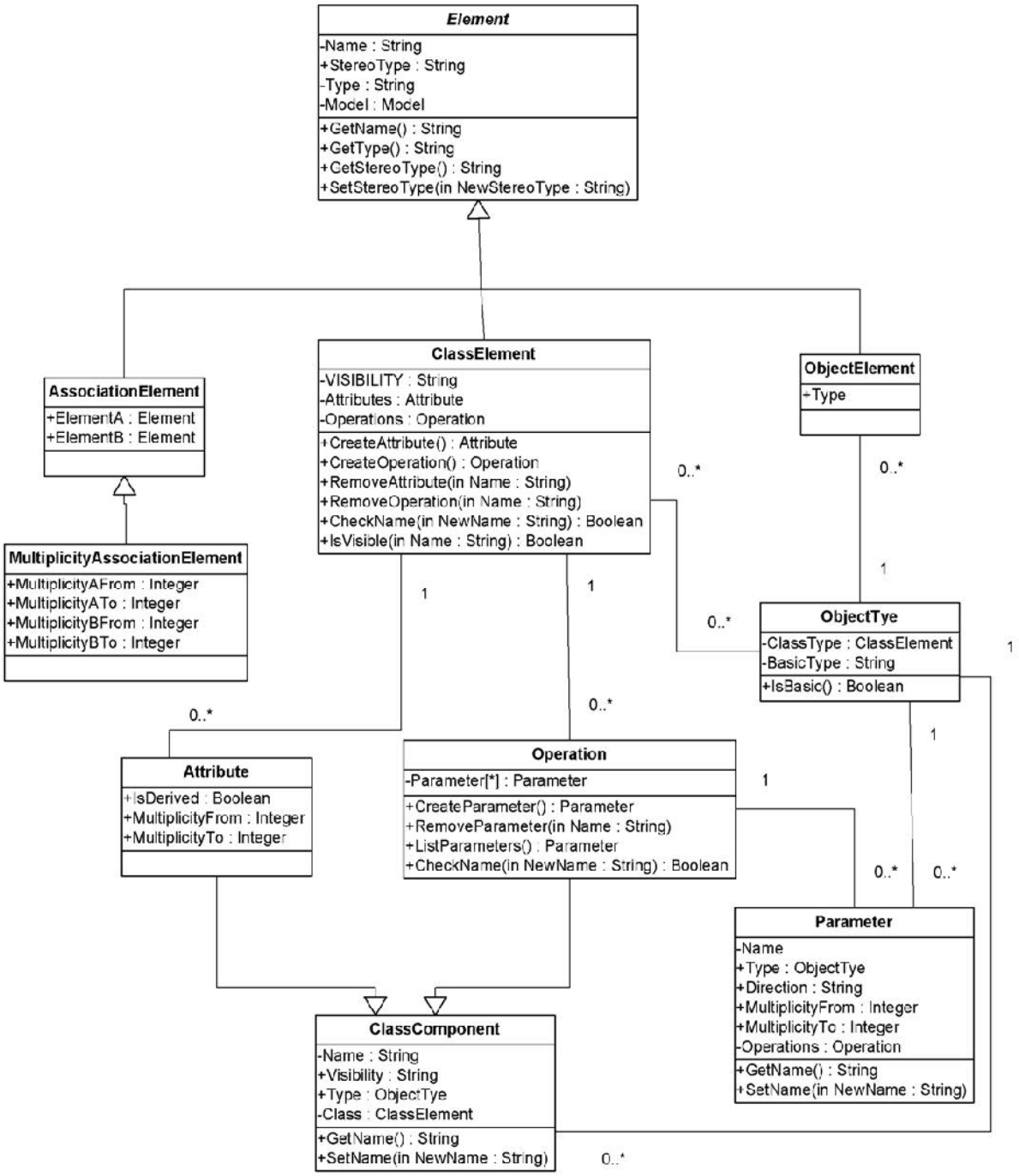

Figure 3.4: Class Diagram (Part 3)

AssociationElement – is a subtype of Element which represents a relationship between two independent elements which is relevant across multiple diagrams.

MultiplicityAssociationElement – a subtype of AssociationElement which adds multiplicity information about each side of the relationship.

ObjectElement – a subtype of Element which represents an object which can participate in a sequence diagram.

ClassElement – a subtype of Element which represents a class. It is part of a class diagram and is also used for defining ObjectElement. It also ensures name uniqueness among its sets of attributes and operations. This class is modified so that its visibility is used in the package diagram in order to control the access rights of a class. As a result an attribute called Visibility is added.

ClassComponent – represents a part of a class, either an attribute or an operation. It stores the name, type and visibility information which is common to both Attribute and Operation.

Attribute – a subtype of ClassComponent which represents a single attribute of a class and adds information about multiplicity and whether an attribute is derived.

Operation – a subtype of ClassComponent which represents a single operation of a class. It adds parameter information as an array containing parameters of the operation. It is responsible for ensuring name uniqueness among its parameters.

Parameter – represents an operation parameter and stores direction, type and multiplicity information. Direction can be: *in*, *out*, *inout* or *return*.

ObjectType – represents information about the type of a Parameмter, ClassComponent or ObjectElement. It can be either a basic type or an instance of ClassElement. Basic types can be: int, Boolean, long, double, float, Date, String, char or void.

Figure 3.5 shows the new classes, PackageElement and DependencyElement which are also subtypes of the Element class, and the relationship with the Diagram class.

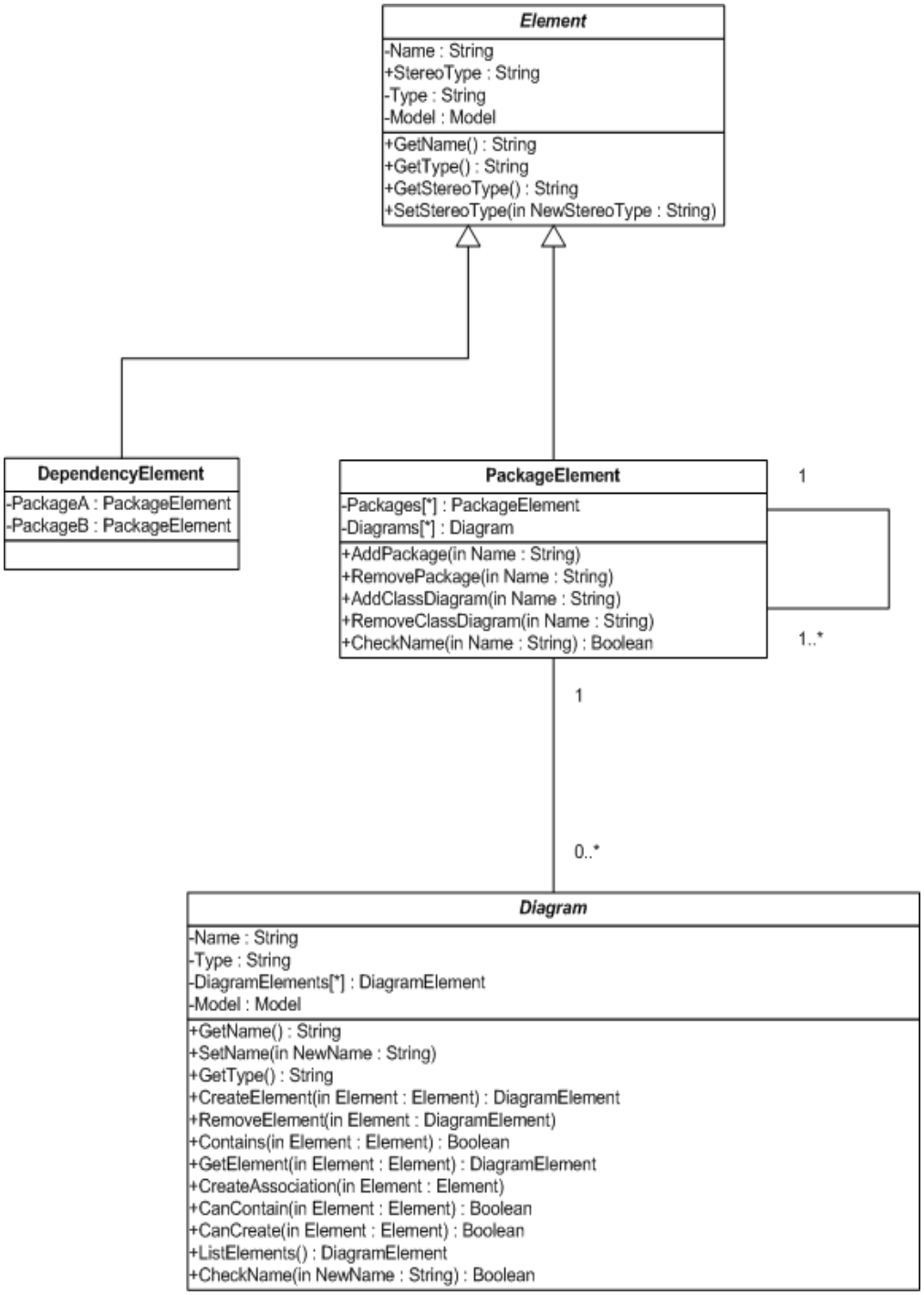

Figure 3.5: Class Diagram (part 4)

PackageElement – a subtype of the Element class which represents a package. It also ensures name uniqueness among the classes in a package and among packages in a package.

DepenedencyElement – a subtype of Element class which represents the relationship between two packages.

3.4. Behavioral Model (Sequence Diagrams)

The sequence of messages among objects during operation of the system is presented using sequence diagrams. Sequence diagrams of some use cases are presented in figures 3.6 – 3.9.

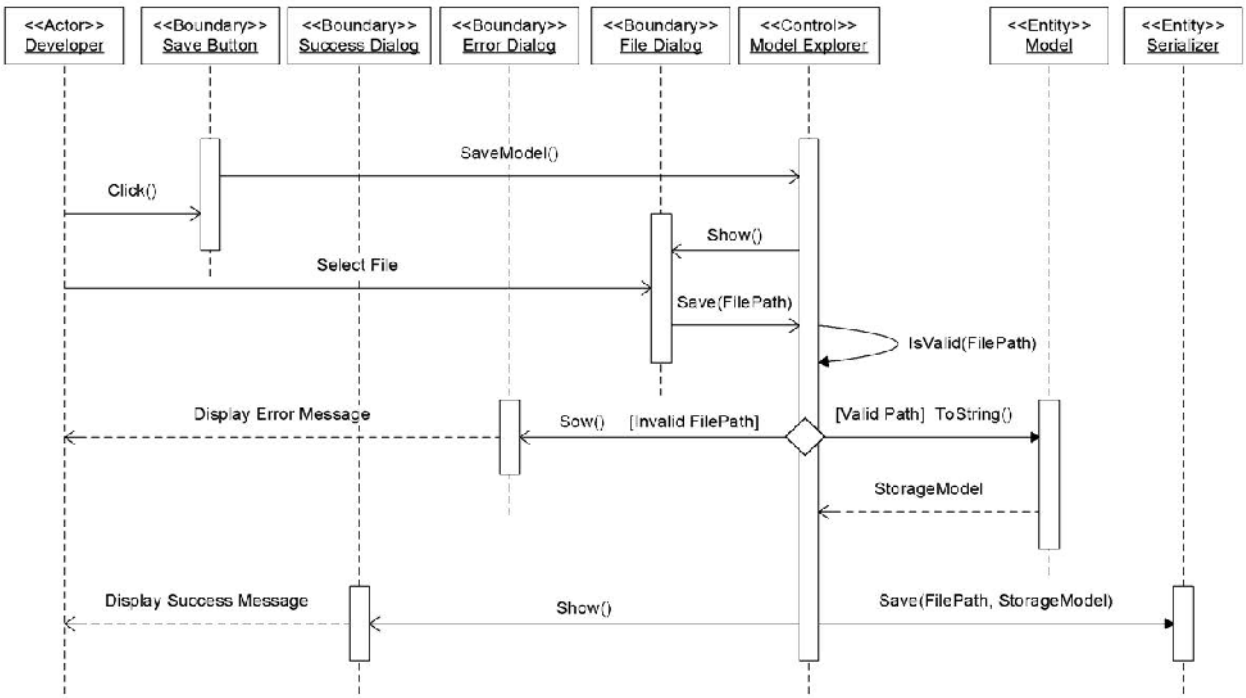

Figure 3.6: Sequence Diagram for Save Model use case

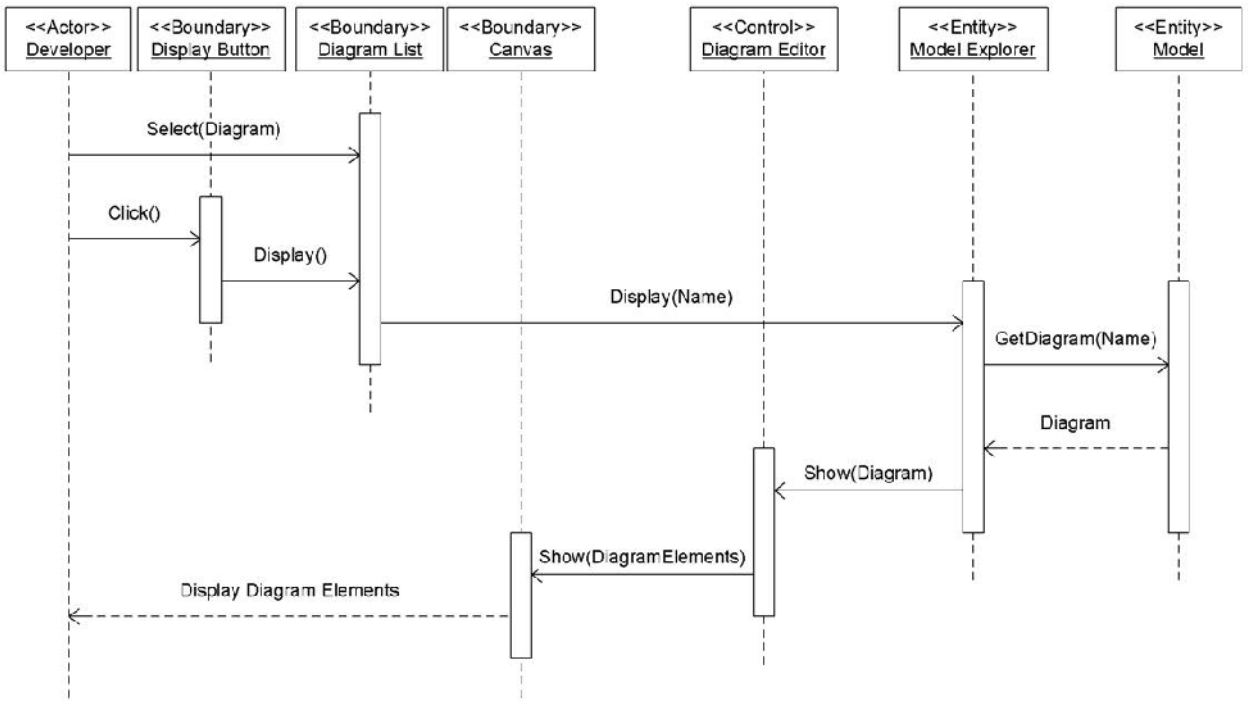

Figure 3.7: Sequence Diagram for the Display Diagram use case

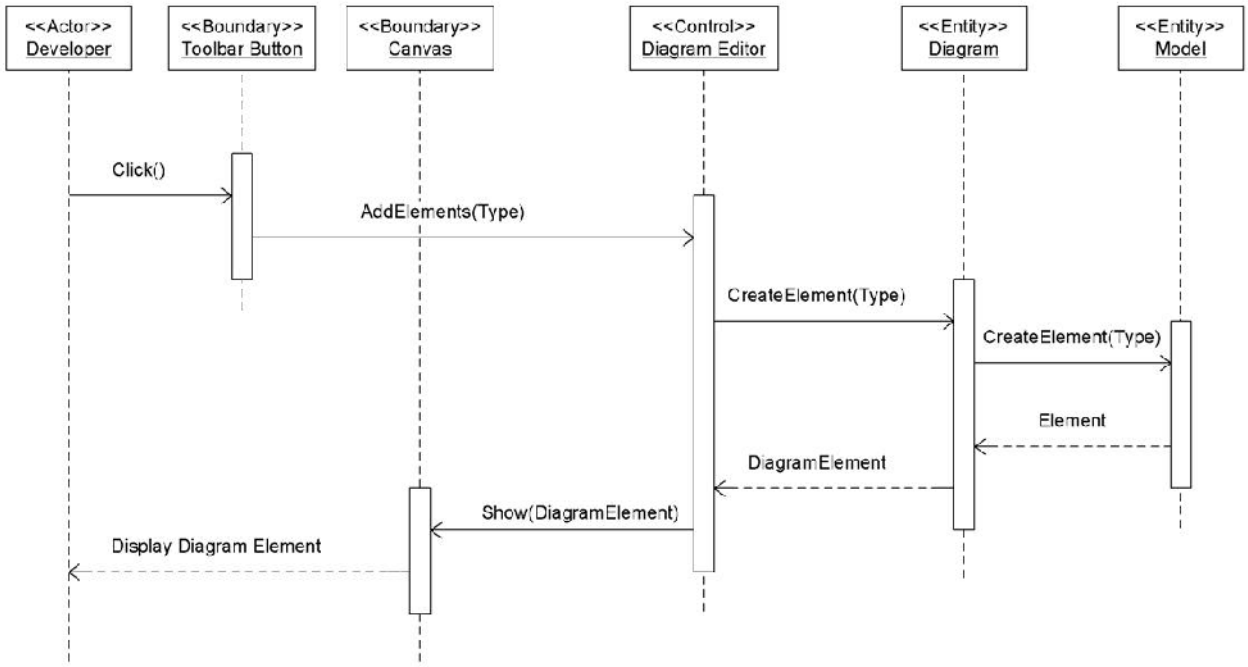

Figure 3.8: Sequence Diagram for Add Element use case

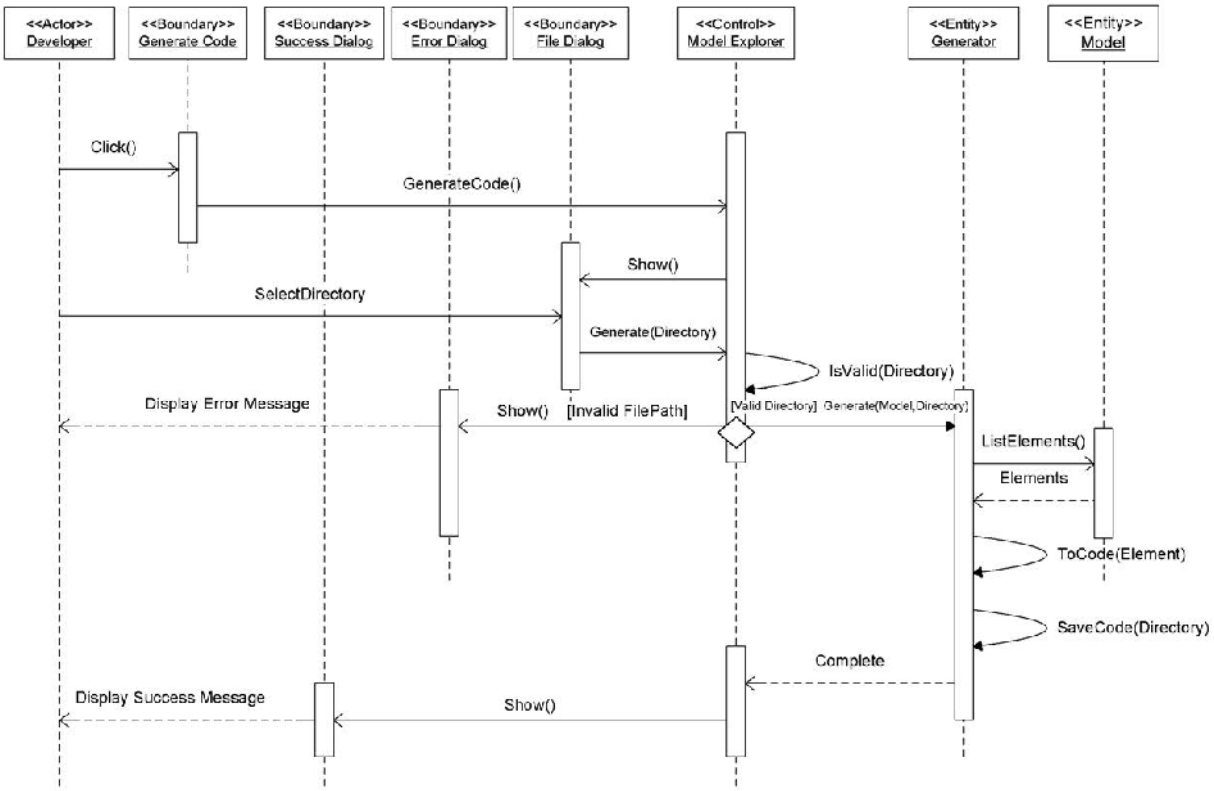

Figure 3.9: Sequence Diagram for Generate Code use case

CHAPTER FOUR: DESIGN

4.1. Software Architecture

The development of this system uses similar software architecture used in the previous system that is a variant of the model view-controller (MVC) architecture. A system is divided into three parts in the MVC design pattern. These are: the model which is the representation of the domain data which the system is operating on, the view which is the part of the system responsible for the presentation of the data, and the controller of the system which accepts input and translates modifications to the model.

4.2. Subsystem Decomposition

The decomposition of the system is based on the previously conducted work [5] so that the system is decomposed into five major subsystems according to the coherence and similarity of the classes making up each subsystem. Each subsystem provides a separate part of the functionality of the system however; most subsystems depend on one or more of the other subsystems for their proper operation. The subsystems and their inter-dependencies are shown in Figure 4.1 followed by the description of each subsystem.

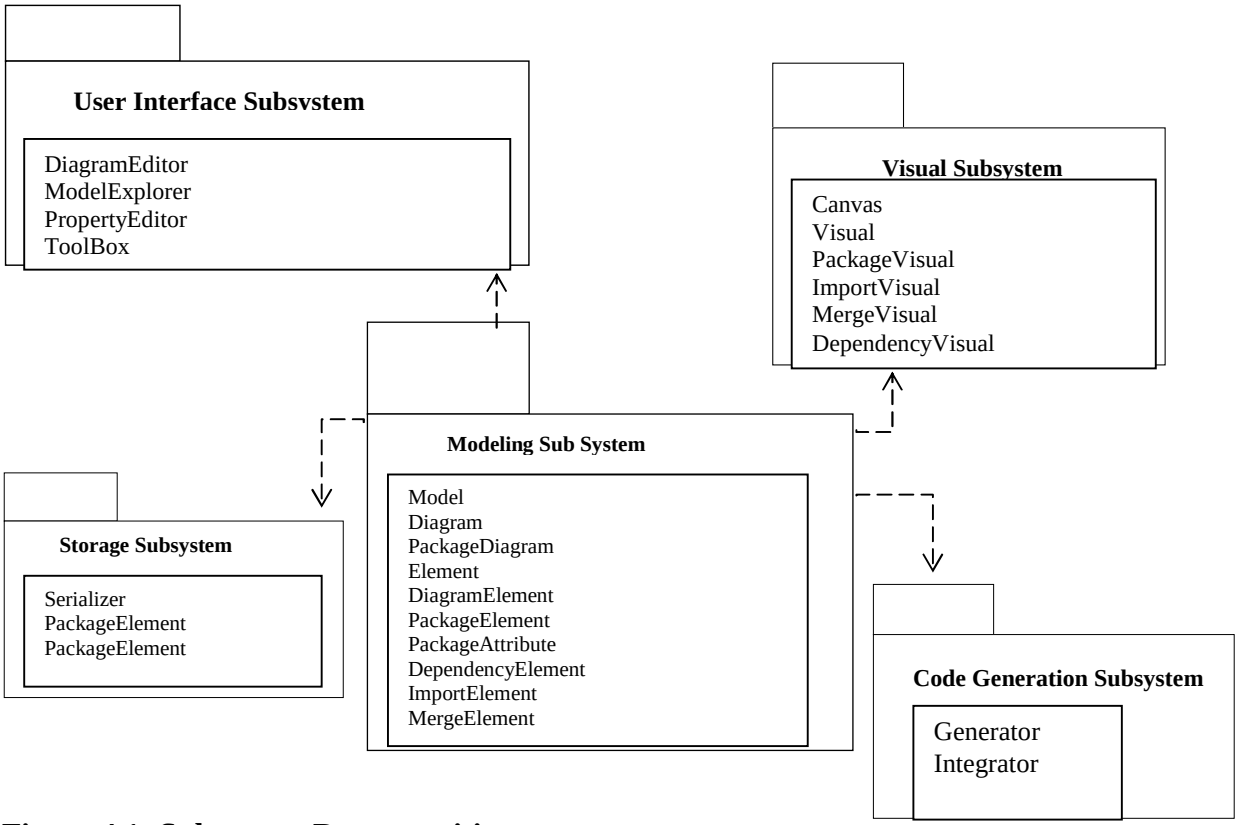

Figure 4.1: Subsystem Decomposition

Modeling Subsystem

This subsystem forms the base of the entire system. It contains the classes representing the UML model, including the diagrams and diagram elements as well as the relationship among the elements. It is responsible for maintaining an in memory representation of a model which serves as a data source that is used and modified by the other subsystems. The major classes in this subsystem are: Model, Diagram, Element, and DiagramElement. The newly added classes in this subsystem are PackageDiagram, PackageElement, PackageAttribute DependencyElement, ImportElement and MergeElement, ConnectorElement.

Storage Subsystem

This subsystem is responsible for converting and storing the model in the XML file format and also retrieving and restoring the in-memory structure of the model from a stored file. The major class in this subsystem is the Serializer which is modified by adding the PackageElement, PackageDiagram, ImprortElement, MergeElement serializable attributes so that the elements

related to packages are stored and retrieved. Other classes such as PackaageElement, PackageDiagram are also added.

Code Generation Subsystem

This subsystem is responsible for generating source code from the model and integrating changes made to the code into the model. The code subsystem also deals with the storage and retrieval of source code files for the generation and integration purpose. All language specific aspects of the system are limited to this subsystem. The major classes in this subsystem are Generator and Integrator which are modified in order for the code generation and integration for classes in package diagrams to namespaces in the code is possible. The BasePackages, ClassesInPackage, ConnectedPackage attributes are added in the Generator class. Moreover, the CreateGenerator, Generate, and ProcessClasses methods are customized to support code generation from packages. ProcessPackages method is added. Likewise, in the integrator class BasePackage, ClassesInPackage, ConnectedPackages are added and a new method ProcessPackage is added. CreateIntegrator, Integrate, ProcessClasses are modified.

Visual Subsystem

The visual subsystem contains all the classes responsible for the visualization and graphical representation of diagram elements. It deals specifically with the visual presentation of element and the relationship among elements onto the screen, accepting modifications to the visual representation and translating the modifications back to the elements in the model. The major classes in this subsystem are Canvas and Visual. The Visual class is modified in such a way that the package element and the dependency elements are contained in the canvas. Some classes that supports the visualization of package are added; the PackageVisual, MergeVisual, ImportVisual and the DependencyVisual.

User Interface Subsystem

This subsystem contains the classes for each of the major building blocks of the user interface of the system. Each class in the user interface subsystem encapsulates a single part of the user interface functionality and form the entire user experience through their interaction. They also serve as control classes which drive the functionality of the system by instantiating, modifying

and destroying objects based on user input. The major classes in this subsystem are: ModelExplorer, DiagramEditor, PropertyEditor and ToolBox. All these classes are modified so that they support the package diagrams. The modifications are described as follows:

- ModelExplorer – additional treenode is used for the package diagram.
- DiagramEditor – the AddElement, DiagramCanvas_DoubleClick, EditToolStripMenuItem_Click are customized.
- PropertEditor – is created that is used to edit PackageElement property.
- ToolBox – is modified so that it can hold the package diagram elements that can be dragged and droped onto the DiagramEditor.

4.3. Persistent Data Management

The persistent data management of this system is based on the previous project. The system has two kinds of persistent output; the model itself and the code. The model is stored as an XML file which replicates the structure of the model in memory. The process of storing the model is a form of object serialization in which attributes of an object are transformed into textual XML format for storage or transformation. This form of storage allows the restoration of the state of the in-memory objects directly. The speed of storage and retrieval is quite good with a slight trade-off in the size overhead of the file.

The generated code will be stored in VB format which is a plain text file containing the code structured according to the syntax of the visual basic.net language. The location and structure of the files in relation to the packages and the model elements from which they are generated will be stored as part of the model in order to facilitate the integration of changes.

CHAPTER FIVE: IMPLEMENTATION

5.1. Tools

The development platform, the IDE, and the programming language used are similar to that of the previous project work in order for the project to be finished on time. The development platform is the .NET framework. The Visual Basic.Net programming language, the Visual Studio 2008 IDE, and .NET framework 3.5 are chosen to implement the system.

5.2. Prototype

5.2.1. The Application Environment

A graphical windows application is used for the prototype of the system as used by the former project conducted. In this section, the interactions with the system and how the system operates are presented. In this presentation, the default names given by the system for each object are used to show the design environment and the generated code. The initial window which appears when running the system is shown in Figure 5.1.

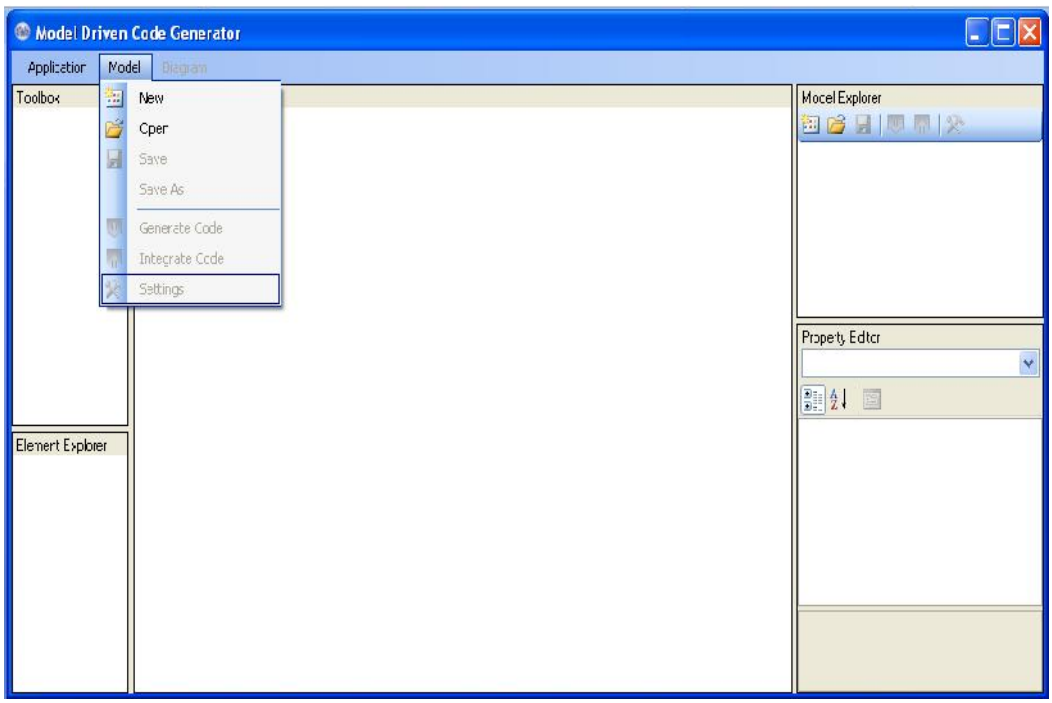

Figure 5.1: Screenshot of the initial state of the system

As shown in the above figure only the New and Open submenus of the Model menu are enabled so that opening new model or stored models. After opening new or saved model other relevant commands are enabled when appropriate to avoid incorrect input from the user. Figure 5.2 shows what the system looks like after creating a new model or opening saved model.

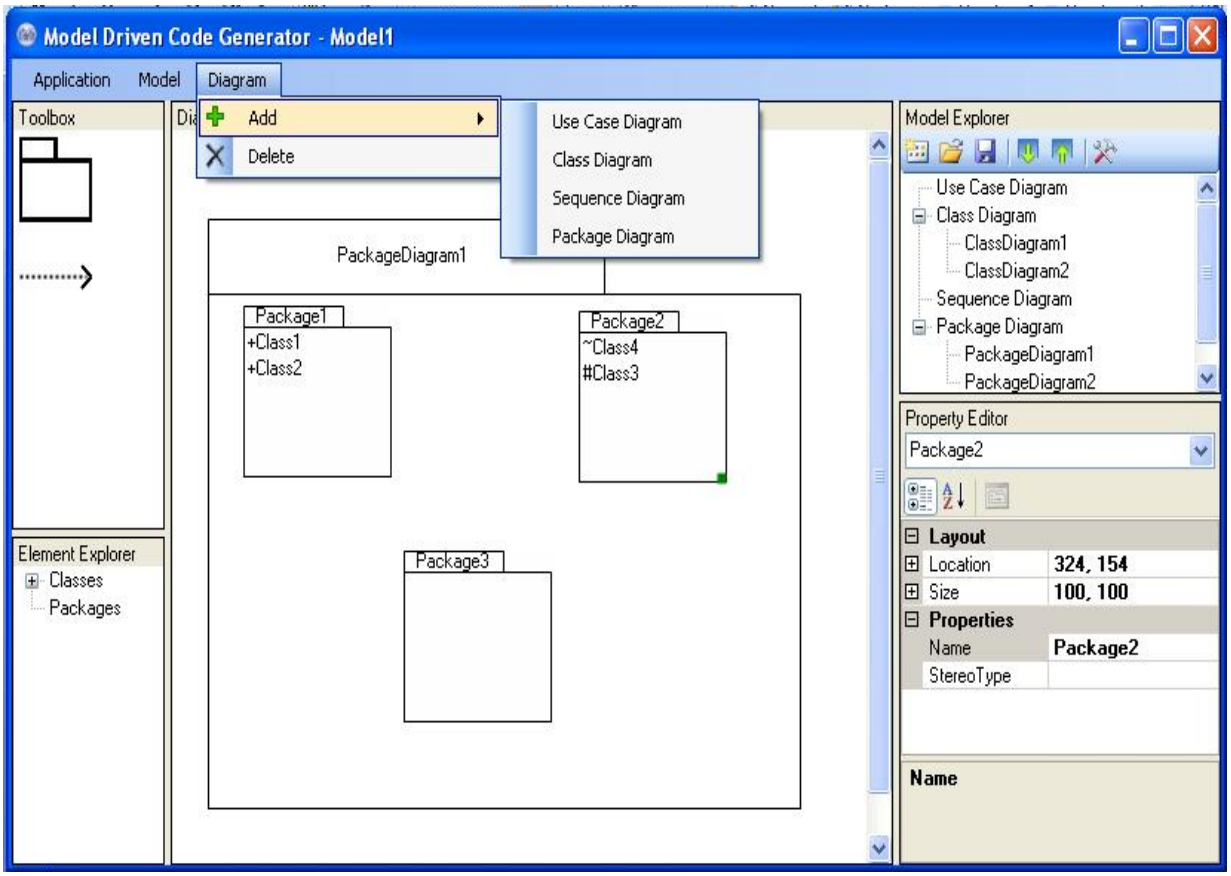

Figure 5.2: Screenshot of a typical working state of the system

Menus

The menu bar at the top of the window contains the model, diagram and application drop down menus. The model menu contains commands for creating, opening, saving a model as well as items for generating and integrating. It also contains an item for changing setting. For adding different types of new diagrams and deleting a diagram, the menu items are found in the diagram menu. The Application menu contains commands for application environment setting such as remembering the previous windows state and to close the system.

Model Explorer

This is the main point of the interactions to the system. Some of the commands important to add, remove and rename a diagram are replicated in this element through a small toolbar.

Toolbox

After a diagram is opened or created and shown different types of elements are populated and shown in the toolbox, that are related to the specifically shown diagram. The user can drag and drop an element into the diagram editor to create a new element within the shown diagram.

Element Explorer

The element explorer shows a list of elements which have been created in other diagrams but not added to the active diagram. The user can drag and drop them to add them to the active diagram.

Diagram Editor

The user interacts with the system mostly through this part of the system since most of the commands such as resizing, moving, deleting and editing of diagram elements are possible in this part. It has a vertical and horizontal scrolling to contain all the diagram elements when the visible area is not enough to display all the elements.

Property Editor

The property editor shows the name, location, size of a selected element within a diagram. The selected element can be renamed through the property editor.

5.2.2. Diagrams

Use case diagrams

A use case diagram can contain the ‘actor’ and the ‘use case’ main elements and three relationship elements; ‘include’, ‘extend’ and ‘communication’. There is also a boundary element in which all the use cases are contained. There is no any modification made to the previous project implementation of the use case diagram. Figure 5.3 shows the state of the system in editing a use case diagram.

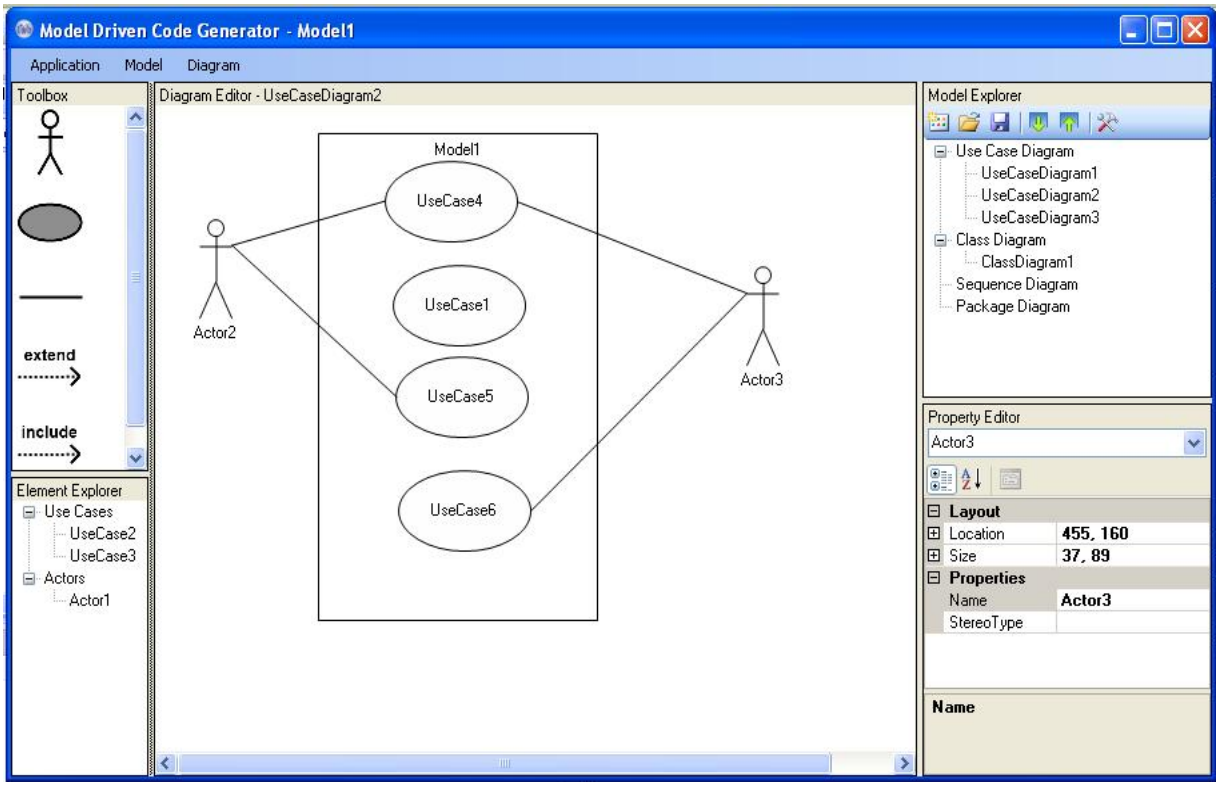

Figure 5.3: Screenshot of use case diagram editing

Class Diagrams

The elements that can be contained in a class diagram are the two relationship elements generalization and binary association elements. The class itself is represented using a class element which is a rectangle with three sections; name, attribute, and operation section. There is a slight modification made to the previous implementation of the class diagram. This modification is the addition of an attribute, Visibility, so that the visibility of classes in a package is specified. The state of the system when editing class diagrams is shown in Figure 5.4.

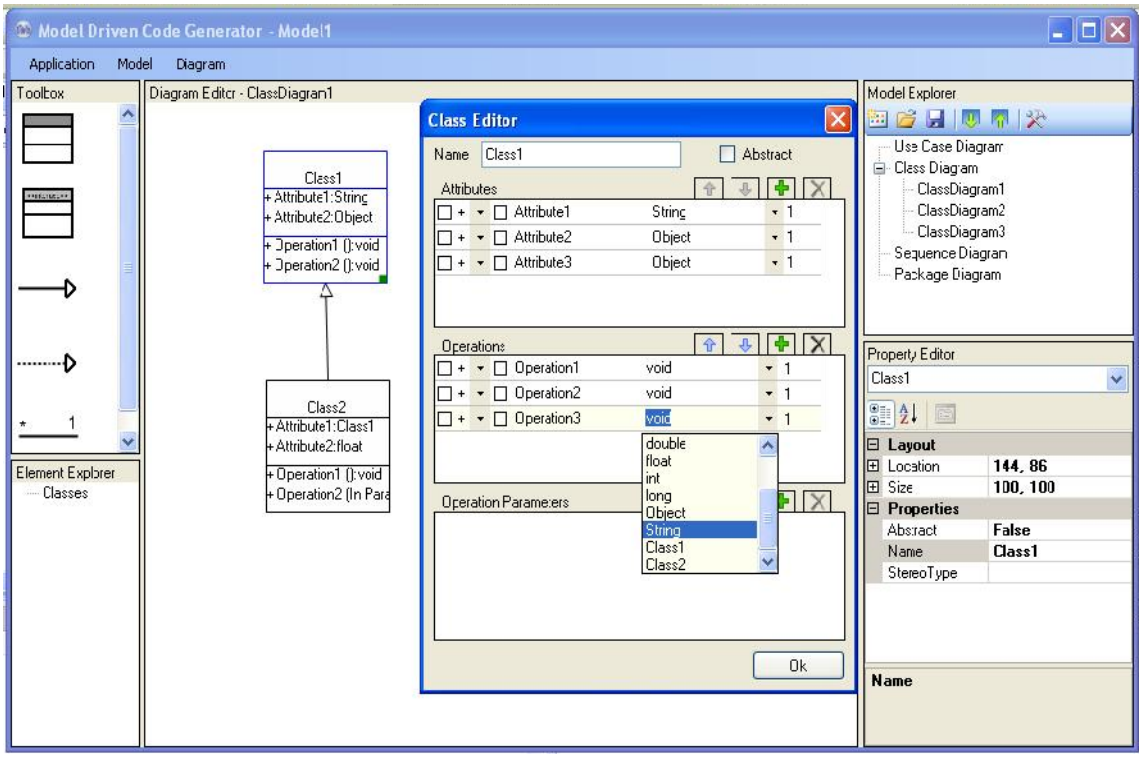

Figure 5.4: Screenshot of class diagram editing

Sequence Diagrams

Actor and object elements and three types of message elements can be contained in a sequence diagram. No modifications are made to the previous implementation. Figure 5.5 shows the system in a state of editing a sequence diagram.

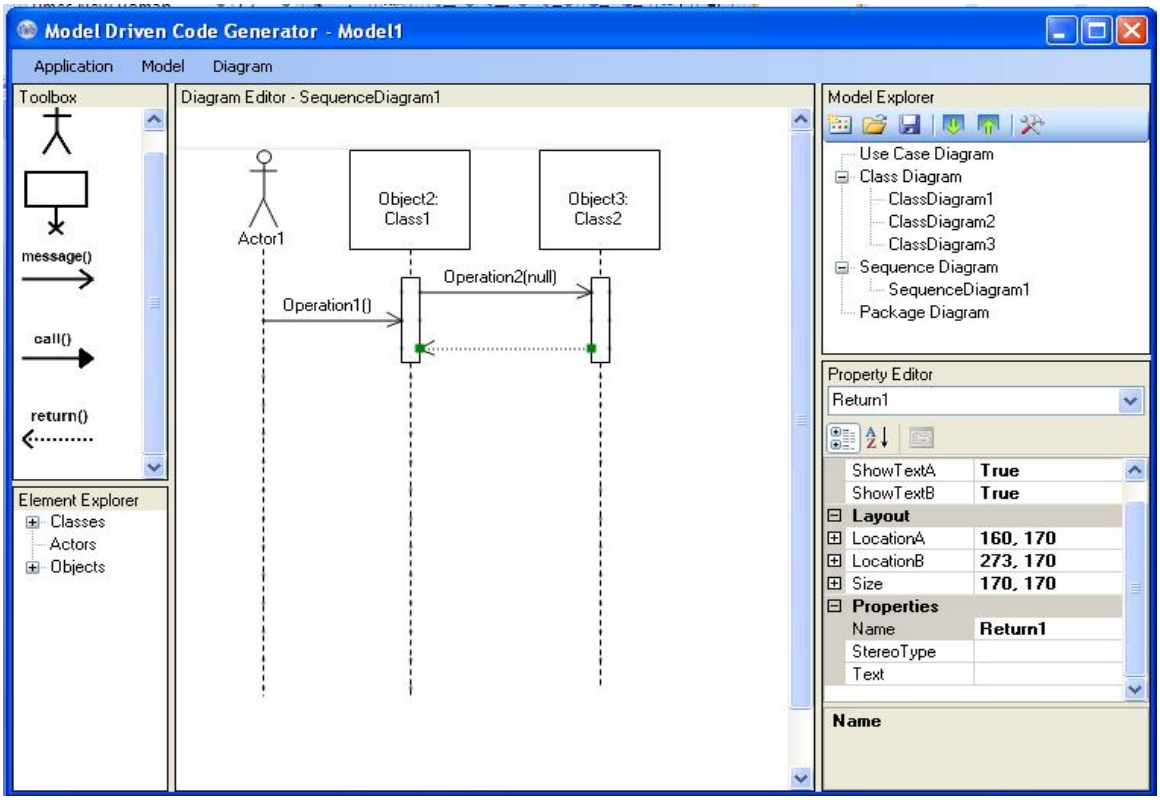

Figure 5.5: Screenshot of Sequence diagram editing

Package Diagrams

A package diagram can contain three dependency elements, the connector, the merge and the import dependencies. The merge dependency represents that the dependent package merges the classes from the other package. This happens where the source and the target packages contains elements of the same name. The import dependency element represents that the dependent side package import and use the classes from the other side package of the import dependency. The connector element is used to show nested packages.

The package itself is represented using a folder like element where the name of the package is displayed on the top left rectangle. It contains attributes which are classes contained in a package with their visibility relative to the containing package and the other package. For example, a class with a private visibility cannot be imported or merged in the other package whereas a public class is accessible to all other packages. The package name can be renamed in the

property editor. The attributes of a package can be edited through an edit package dialog. Figure 5.6 depicts the state of the system when editing package diagrams.

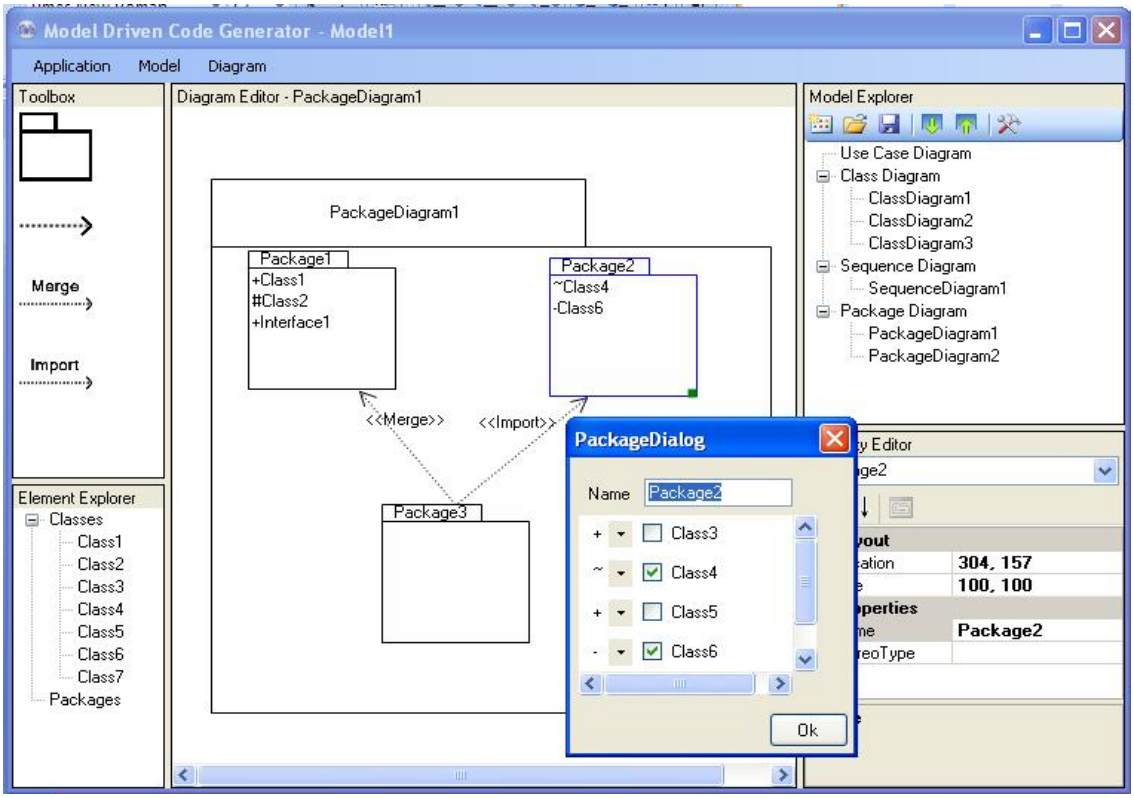

Figure 5.6: Screenshot of Package Diagram editing

Code Generation and Integration

From the model menu or the generate toolbar item on the model explorer can be used to generate code from the model. The user can specify the path where the model folder is created in which the package folders and files are created for each package. In each package a VB file with the name of the package diagram is created where all the code related to this package is generated and saved. Table 5.1 and Table 5.2 show the previous [5] algorithm and the modified algorithm of code generation respectively.

Table 5.1: The Previous Code Generation Algorithm

- Accepts the model of a system (the currently opened class Diagram model)
- Accepts the generation settings
  - Rename_code_module
  - Remove_classes_from_code
  - Preserve_operation_code
- The user enters new or existing file where the generated code is going to be saved
- If existing file is given and if the Rename_code_module is true, the name of the existing module is renamed with the model name.
- Then, the code generator iterates through all class and association elements of the model and a code object is created for each classes

  If the given file exists and if the Remove_classes_from_code is true, the classes existed in the code that are not existed in the model are removed. If Remove_classes_from_code is false, the classes that are in the code but not in the model are preserved to exist.

  - If a class has operations, the code generator iterates through all the operations and creates code object for each operations

    If the given file is existing and an operation is found that already exists in the code and if the Preserve_operation_Code is true, the code in the operations are left undeleted.

Table 5.2: The New Code Generation Algorithm

- Accepts the model of a system (the currently opened package diagram model)
- Accepts the generation settings
  - Preserve_operations_code
  - Remove_classes_from_code
  - Remove_packages_from_code
- The user enters new or existing directory where the generated code is going to be saved
- Then, the code generator iterates through all elements of the model and a code object is created for each packages

  If existing directory is given and there exists a package that is not in the model and the Remove_packages_from_code is true, the package that is not in the model is deleted.
  - For each of the classes in a package, class code object is created and contained in the package's code object

    If the given directory exists and if the Remove_classes_from_code is true, the classes existed in the code that are not existed in the model are removed. If Remove_classes_from_code is false, the classes that are in the code but not in the model are preserved to exist.

    If a class has operations, the code generator iterates through all the operations and creates code object for each operations

    If the given directory is existing and an operation is found that already exists in the code and if the Preserve_operation_Code is true, the code in the operations are left undeleted.

The algorithm works somewhat in a similar way to the previous implementation however it is modified and improved so that code generation from package diagrams is supported. It first gathers all information related to each package diagrams, then it collects all the data related to each packages in each package diagrams, and finally it generates a file for each package elements. Each file contains all the classes implementation contained in a package, the nested packages implementation code and the dependency code based on the visual basic.net programming language syntax. Table 5.3 shows the sample code generated.

Table 5.3: Sample Generated Code

```
Imports Package2
Namespace Package1

    Public Class Class1
        Inherits Class4

        Public Attribute1 As String
        Public Attribute2 As Object
        Public Attribute3 As Object

        Public Sub Operation1()

        End Sub

        Public Sub Operation2()

        End Sub

        Public Sub Operation3()

        End Sub

    End Class

    Public Class Class2
        Inherits Class1

        Public Attribute1 As Class1
        Public Attribute2 As Float

        Public Sub Operation1()

        End Sub

        Public Function Operation2(ByVal Parameter1 As Long) As Date

        End Function

    End Class

    Public Interface Interface1


    End Interface

End Namespace
```

The integration accepts the folder where the model code is generated and reads the code and based on the namespace and class information it reintegrates the modifications made to the code into the model class diagrams and package diagrams currently opened. The model currently opened should be the model in which the code to be integrated is generated from.

XML Storage

The user can select the corresponding menu item from the model menu or the model explorer toolbar in order to save or open a model. A file dialog appears that a user can select a file where the model is saved or from where the model is opened. The model is converted into XML format storage. Table 5.4 shows a sample XML file in which a model is saved.

Table 5.4: Sample XML format storage of a model

```
<?xml version="1.0" encoding="utf-8"?>
<StorageHandle
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xsi:type="HandleSet">
  <Id>-3</Id>
  <StorageHandles>
    <StorageHandle xsi:type="Model">
      <Id>0</Id>
      <Name>Model1</Name>
      <Settings xsi:type="HandleReference">
        <Id>1</Id>
      </Settings>
      <Diagrams>
        <StorageHandle xsi:type="HandleReference">
          <Id>7</Id>
        </StorageHandle>
        <StorageHandle xsi:type="HandleReference">
          <Id>11</Id>
        </StorageHandle>
      </Diagrams>
...
<StorageHandle xsi:type="ClassElement">
      <Id>2</Id>
      <Name>Class1</Name>
      <IsAbstract>false</IsAbstract>
      <Attributes />
      <Operations />
    </StorageHandle>
    <StorageHandle xsi:type="PackageAttribute">
      <Id>3</Id>
      <Visibility>1</Visibility>
      <PAttribute xsi:type="HandleReference">
        <Id>2</Id>
```

```
    </PAttribute>
  </StorageHandle>
  <StorageHandle xsi:type="PackageElement">
    <Id>4</Id>
    <Name>Package1</Name>
    <Attributes>
      <StorageHandle xsi:type="HandleReference">
        <Id>3</Id>
      </StorageHandle>
    </Attributes>
  </StorageHandle>
   <StorageHandle xsi:type="DependencyElement">
    <Id>6</Id>
    <Name>Dependency1</Name>
    <ElementA xsi:type="HandleReference">
      <Id>4</Id>
    </ElementA>
    <ElementB xsi:type="HandleReference">
      <Id>5</Id>
    </ElementB>
  </StorageHandle>
  <StorageHandle xsi:type="PackageDiagram">
    <Id>7</Id>
    <Name>PackageDiagram1</Name>
    <Elements>
      <StorageHandle xsi:type="HandleReference">
        <Id>8</Id>
      </StorageHandle>
       </Elements>
  </StorageHandle>
...
<StorageHandle xsi:type="ClassDiagramElement">
    <Id>12</Id>
    <SizeW>100</SizeW>
    <SizeH>100</SizeH>
    <LocationX>145</LocationX>
    <LocationY>107</LocationY>
    <Element xsi:type="HandleReference">
      <Id>2</Id>
    </Element>
  </StorageHandle>
 </StorageHandles>
</StorageHandle>
```

CHAPTER SIX: CONCLUSION AND RECOMMENDATION

6.1. Conclusion

The automation of the transitions between the phases of software development reduces the resources and time spent on the development of software. The most basic transition is the conversion of the system model into the executable artifact that is the source code. Between these models, only the conversion is not enough, the two artifacts should also be consistent at any given time in the development process.

A project work conducted by Henock Abye [5] in 2010, was aimed at construction of a UML design environment for the three core UML constructs (use case, class and sequence diagrams) and generating code from a class diagram and integrating changes made to the code into the class diagram. The purpose of this project is also similar to that of the previous project. However, in this project the domain of using the UML constructs is increased so that package diagrams can be used. At this point, this system supports generating code from a model based on packages not classes only and integrating the code to the model too.

6.2. Recommendations

Due to time and resource constraints the scope of this project was limited. As a result, many works are left undone.

- Still, a number of UML diagrams have been excluded, inclusion of these diagrams may improve the application.
- The code generation has not considered behavioral diagrams; behavioral diagrams increase the domain of code generation more.
- The decision element in the sequence diagrams has not been included; addition of this element may help for good sequence diagram.

References

1. Niklaus Wirth, *A brief history of software engineering*, IEEE Computer Society, 2008
2. Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise, *MDA Distilled-Principles of Model-driven Architecture*, Addison Wesley, 2004.
3. Sami Beydeda, Matthias Book, Volker Gruhn (Eds.), *Model-driven Software Development*, Springer-Verlag Berlin Heidelberg, 2005.
4. Martin Fowler, *UML Distilled a brief guide to the standard Object Modeling Language*, Third Edition.
5. Henock Abye, "*Design and Implementation of Code Generator for Model-driven Software Development*", MSc Project, Addis Ababa University, 2010.
6. Matthias Biehl and Welf Lowe, *Automated Architecture Consistency Checking for Model Driven Software Development,* Lecture Notes in Computer Science, Vol. 5581, pp 36-65, Springer, 2009.
7. B.S. Ainapure, *Object Oriented Modeling and Design,* Technical Publications, 2010.
8. Russel Miles and Kim Hamilton, *Learning UML 2.0,* O'Reilly, 2006.
9. *Unified Modeling Language*: *Infrastructure* version 2.0 formal/05-07-05.

Declaration

I, the undersigned, declare that this project is my original work and has not been presented for degree in any other university, and that all sources of materials used for the project have been acknowledged.

Declared by:

Name: Melese Erku

Signature: _____

Date: _____

Confirmed by advisor:

Name: Dr. Dida Midekso

Signature: _____

Date: _____

Place and date of submission: Addis Ababa, June 2010.