Addis Ababa University

College of Natural Sciences

Department of Computer Science

Enhanced Code Generator for Model Driven Software Development

BY

Tedi Melaku Wirtu

A Project 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, 2014

Addis Ababa

Addis Ababa University

College of Natural Sciences

Department of Computer Science

Enhanced Code Generator for Model Driven Software Development

BY

Tedi Melaku Wirtu

Name and signature of the examining board:

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

Acknowledgements

First and Foremost, I would like to express my deepest appreciation to my advisor Dr Dida Midekso for the useful comments, remarks and engagement through the process of this project work.

A special thanks to my family. Words cannot express how grateful I am to my father and my mother for your love and support.

Abstract

Software development is a process by which a new software program is created or an existing software program is enhanced. Software development passes through clear phases of engineering called Software Development Life Cycle (SDLC).

Currently, software development relies much on modeling. Models provide abstractions of a physical system that allow engineers to reason about the system by ignoring extraneous details while focusing on relevant ones. All forms of engineering rely on models to understand complex, real-world systems.

In the past two decades, different software designing tools and code generators have been introduced for model driven software development. Some of these tools have code generation capability for multiple platforms by providing their own model editor. While other code generators use model of other UML editors as an input to generate code. Henoke [6] also design and implement a code generator for model-driven software development. This code generator has a UML modeling interface for three UML diagrams. The code generator is designed to generate structural code from class diagram to a single platform which is Visual Basic .NET. It also includes the reintegration of the modification of the generated code into the model of the system being developed. Melese [7] enhanced the work of [6] by adding package diagrams to the model construction module and modified the code generator algorithm to support a package diagram.

This project work extends the works of [6, 7] by adding code generation to multiple platforms from a Single Platform Independent Model (PIM) and code generation for MVC design pattern. The model has five major components: Model Constructor, XMI Parser, Class Parser, Model Transformer, and Code Generator.

A prototype of the model is built and evaluated with users. The usability test has shown that the performance of the prototype is very good.

Keywords: Model Driven Architecture, Unified Modeling Language, Code generation, MVC

Table of Contents

List of Figures

List of Tables

Acronyms

API……..Application Program Interface

CIM……Computational-Independent Model

CWM……Common Warehouse Meta-model

DTD……Document Type Definition

EMF……Eclips Modeling Framework

HTML……HyperText Markup Language

J2EE……Java 2 Enterprise Edition

M2M……Model to Model

M2T……Model to Text

MDA……..Model Driven Architecture

MDSD……Model Driven Software Development

MOF……Meta Object Facility

MVC……Model View Controller

OA&DTF……..Object Analysis and Design Task Force

OMG……..Object Management Group

PIM……..Platform Independent Model

PSM……..Platform Specific Model

SOAP……..Simple Object Access Protocol

UI……..User Interface

UML……...Unified Modeling Language

W3C……World Wide Web Consortium

XMI……..XML Metadata Interchange

XML……..Extensible Markup Language

Chapter One: Introduction

1.1. Background

Software development is the process by which a new software program is created. This process can also be applied to an established program to create a new version of that software, though this is usually an abridged version of the process unless the new version is largely different from the previous one. Numerous steps are involved in this process, beginning with understanding what is needed from software, developing a plan for creating it, writing the code, and bug testing prior to launch.

To better understand the economics of software development, it is interesting to take a somewhat different perspective and track the life of a specific software system from its inception to its decommissioning. Normally, the term *Software Development Life Cycle (SDLC)* is used in discussions about the approach and software development method used in a specific project. SDLC model is a way of describing the planning, designing, coding, and testing of a software system, as well as the method in which these steps are implemented. A variety of life cycle models exist, but they all include the same constituent parts. All life cycle models take a project through several primary phases: a requirements-gathering phase, a design phase, a construction or implementation phase, and a testing phase

In the last two decades, software development relies much on modeling. Models provide abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on relevant ones. All forms of engineering rely on models to understand complex, real-world systems. Models are used in many ways: to predict system qualities, reason about specific properties when aspects of the system are changed, and communicate key system characteristics to various stakeholders. The models may be developed as a precursor to implementing the physical system, or they may be derived from an existing system or a system in development as an aid to understanding its behavior.

Model-driven software development aims at improving productivity and maintainability of software by raising the level of abstraction from source code in a general purpose language to

high-level domain specific models such that developers can concentrate on application logic rather than the accidental complexity of low-level implementation details. The essence of the approach is to shift the knowledge about these implementation details from the minds of programmers to the templates of code generators that automatically translate models into implementations [1]. Model-driven software can be constructed using different possible combinations of implementation technologies, and leveraging numerous design patterns in various ways, which is why the term Model-Driven Architecture (MDA) that was introduced by Object Management Group (OMG) [2]. The Object Management Group is an international, non-for-profit industrial consortium that creates and maintains software interoperability specifications. The OMG's specifications include the Unified Modeling Language (UML) modeling notation, XML Metadata Interchange (XMI), Common Object Request Broker Architecture (CORBA) middleware, and dozens of domain specific interoperability specifications in such areas as transportation, life sciences, telecommunications, and manufacturing [3].

MDA introduces an approach to system specification that separates the views on three different layers of abstraction: high level specification of what the system is expected to do (Computation Independent Model or CIM), the specification of system functionality (Platform Independent Model or PIM) and the specification of the implementation of that functionality on a specific technology platform (Platform Specific Model or PSM). CIM presents specification of the system at problem domain level and can be transformed into elements of PIM. PIM provides formal specification of the system structure and functions that abstracts from technical details, and thus presents solution aspects of the system to be developed, which enables model transformation to the platform level (PSM), named implementation domain [4].

1.2. Statement of the Problem

Advances in languages and platforms during the past two decades have raised the level of software abstractions available to developers. For example, developers today typically use more expressive object-oriented languages, such as C++, Java, or C#, rather than Fortran or C [5]. Moreover, new technologies and platforms are emerging and changing constantly, which

implies a high effort of developing software products. This situation generates different problems related to portability, integration and interoperability.

Despite these advances, several vexing problems remain. At the heart of these problems is the growth of platform complexity, which has evolved faster than the ability of general-purpose languages to mask it. A related problem is that most application and platform code is still written and maintained manually using third generation languages, which incurs excessive time and effort particularly for key integration related activities, such as system deployment and configuration.

Nowadays, it's common to use design patterns in software development. Design pattern helps developer to cleanly separate the business and presentation logic of application from its user interface (UI). Maintaining a clean separation between application logic and UI helps to address numerous development and design issues and can make application much easier to test, maintain, and evolve. It can also greatly improve code re-use opportunities and allows developers and UI designers to more easily collaborate when developing their respective parts of the application. Even if this pattern comes with a lot of advantages, there are distinct disadvantages too. Firstly, it is too complex to implement and is not suitable for smaller application; rather, implementing this pattern in such applications will have adverse effects in the application's performance and design. Moreover, having to create a new model, a new controller, and a new set of views every time you create a new database table is tedious; and copying, pasting, and editing an existing set is cumbersome.

A promising approach to address these problems is to develop a tool in the heart of MDA. This kind of tool will allow the same model to be realized on multiple platforms. There are a number of existing tools based on this architecture most of which provide powerful code generation capabilities across a wide range of languages. Henok [6] also designed and implemented MDA tool with model construction and code generation features. The model construction of his work is limited to three UML diagrams (use case, sequence and class diagram) and the code generator module generates code by parsing the contents of the code file into a set of objects which represent the different aspects of the code to a single programming language, and also

synchronize changes made in the model into the code on demand. Melese [7] enhanced the work of Henok by adding package diagrams to the model construction module and modified the code generator algorithm to support package diagram. However, both the original work [6] and the enhanced one [7] lacks generation of code to multiple platforms with consideration of design pattern, inclusion of configuration files that ease deployment and configuration activities, generation of database script and support of XMI for the purpose of interchanging model within different modeling tools.

There exist a number of tools both commercial and open source targeting at MDA driven development which may encompass the functionalities mentioned above to some level. But all lack one basic feature which is the generation of code for design pattern from class diagram.

1.3. Objective

1.3.1. General Objective

The general objective of the project is to enhance the code generator for Model-driven Software Development designed and implemented by [6] and [7].

1.3.2. Specific Objective

In order to attain the general objective, the following list of specific objectives is set:

- Conduct literature review on Unified Modeling Language (UML) and MDA specifications.
- Design and develop extended diagram editor.
- Design and implement a XMI Parser that provides a model object from imported XMI file.
- Design and implement a Class Parser that provides a model object from imported class files.
- Design and implement a code generator by adopting model transformation.
- Design and implement database script generator.
- Conduct usability test of the developed code generator.

1.4. Scope and Limitation

The scope of the project is limited to model construction, code and database script generation by adopting model transformation and the development of XMI and class parsers.

The model construction module will consider the major eight UML diagrams such as: Use case, Class, Object, Component, Deployment, Use case, Activity, State Machine and sequence). Other modeling diagrams used for Engineering and Web Ontology Language (OWL) are not included in the project.

As the OMG' suggested, the code generation module generate code to three different platform(C#, VB and Java). The outcome of this module may include source code, configuration files and other output specific to the target platform. The code generation will include transform templates for defining user transforms and will support MVC design pattern.

Finally, the reverse engineering module builds a model object from codes provided by the user with the help of the class parser. The model transformed by this module is only limited to class diagrams.

1.5. Application of the Project

The result of this project is mainly targeted to be used by system developers. Using this system, developers can easily construct UML diagrams, generate quality code for different platforms without much effort and also it helps developers to easily generate UML class diagram from source code.

1.6. Document Organization

The document is organized into Eight Chapters including the current one. In Chapter Two literature is reviewed on model driven development. Related works is presented in Chapter Three. In Chapter Four system requirements and system models are presented in detail. The System Design is presented in the Fourth Chapter followed by the Implementation which is presented in Chapter Six. Chapter Seven shows the process and result of Usability Test. Finally in Chapter Eight Conclusion and Recommendation are given.

Chapter Two: Literature Review

2.1. Overview

Software development process requires careful planning and execution to meet the goals. Especially now the existence of different platforms and architectures make it worse. To ease this challenge, many efforts are undertaken. One of these efforts is the introduction of standards and automatic code generators. In the past two decades, different software designing tools and code generators have been introduced that fit into the Model Driven Architecture (MDA) paradigm.

This Chapter presents review of literature on Model Driven Software Development (MDSD) Code generators. It presents a brief introduction to OMG and the core standards of the MDA (i.e., UML, MOF and XMI) and discusses the basics and different classifications of code generation.

2.2. Object Management Group (OMG)

The Object Management Group (OMG®) is an international, open membership, not-for-profit computer industry standards consortium. Founded in 1989, OMG standards are driven by vendors, end-users, academic institutions and government agencies. OMG Task Forces develop enterprise integration standards for a wide range of technologies and an even wider range of industries. OMG's modeling standards, including the Unified Modeling Language (UML) and Model Driven Architecture (MDA) enable powerful visual design, execution and maintenance of software and other processes. In this section, OMG's specifications which are relevant for this project will be explained.

2.2.1. Model Driven Architecture (MDA)

Nowadays, new technologies and platforms are emerging and becoming popular constantly, for example Java, HTML, XML, J2EE, UML, etc. The enterprises usually develop their information systems according to these modern technologies. For this reason, new problems have appeared in relation to the interoperability, the portability and the integration of the information systems

[8]. To overcome this problem, OMG have come up with a framework called Model Driven Architecture (MDA), which improves portability of applications by allowing the same model to be realized on multiple platforms.

MDA introduces an approach to system specification that separates the views on three different layers of abstraction: high level specification of what the system is expected to do (Computation Independent Model or CIM), the specification of system functionality (Platform Independent Model or PIM) and the specification of the implementation of that functionality on a specific technology platform (Platform Specific Model or PSM) [9].

CIM- Computation Independent Model

A *computation independent model* is a view of a system from the computation independent viewpoint. A CIM does not show details of the structure of systems. A CIM is sometimes called a domain model and a vocabulary that is familiar to the practitioners of the domain in question is used in its specification.

PIM- Platform Independent Model

PIM provides formal specification of the system structure and functions that abstracts from technical details, and thus presents solution aspects of the system to be developed, which enables model transformation to the platform level (PSM).

PSM- Platform Specific Model

A PSM is a model for a specific platform. A PIM can be transformed into one or more PSMs, which are for a specific implementation technology.

The next step is code generation. For component environments, the system will have to produce many types of code and configuration files including interface files, component definition files, program code files, component configuration files, and assembly configuration files. The more complete the application semantics and run-time behavior can be included in the platform-specific application model and the more complete the generated code can be.

In recent years, many organizations have begun to focus attention on Model Driven Architecture as an approach to software system design and implementation. This is very

possible development for several reasons. MDA encourages efficient use of system models in the software development process, it supports reuse of best practices when creating families of systems and MDA divorces implementation details from business functions. With MDA, functionality and behavior are modeled once and only once. Mapping from a PIM through a PSM to the supported MDA platforms is defined by OMG specifications which are then implemented by tools.

At the core of the MDA concept are a number of important OMG standards: The Unified Modeling Language (UML), Meta Object Facility (MOF), XML Metadata Interchange (XMI), and the Common Warehouse Meta-model (CWM). These standards define the core infrastructure of the MDA, and have greatly contributed to the current state-of-the art of systems modeling [10].

2.2.2. Unified Modeling Language (UML)

By the early 90s, many object-oriented modeling language techniques appeared. Among the more popular were OMT, Boch method and Objectory. While this brought new and improved modeling languages, it also resulted in a tower of modeling-language with no interoperability between vendors' tools. In an effort to organize, simplify, and provide a way to enable interoperability of these graphical modeling tools the OMG established an Object Analysis and Design Task Force (OA&DTF, the precursor to the ADTF), which issued a request for proposal to create what would become UML [11].

In fall 1997, UML 1.1 was adopted by Object Management Group (OMG) as an open modeling standard. Since then UML Revision Task Forces have produced several minor revisions, the most recent being the UML 2.5 specification, which was adopted in 2012 and finalized in 2013. Today, UML is used to model structure and behavior of every type of software from enterprise automation to real-time embedded systems. UML is flexible enough to be used for everything from initial concept designs through full code generation. UML continues to evolve and improve with many software vendors worldwide that provide interpretational UML software models.

UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components [12].

UML defines the notation and semantics for the following domains:

- ***The User Interaction or Use Case Model*** - describes the boundary and interaction between the system and users. Corresponds in some respects to a requirements model.
- ***The Interaction or Communication Model*** - describes how objects in the system will interact with each other to get work done.
- ***The State or Dynamic Model*** - State charts describe the states or conditions that classes assume over time. Activity graphs describe the workflows the system will implement.
- ***The Logical or Class Model*** - describes the classes and objects that will make up the system.
- ***The Physical Component Model*** - describes the software (and sometimes hardware components) that make up the system.
- ***The Physical Deployment Model*** - describes the physical architecture and the deployment of components on that hardware architecture.

As the OMG specification states one of the primary goals of UML is to advance the state of the industry by enabling object visual modeling tool interoperability. However, to enable meaningful exchange of model information between tools, agreement on semantics and notation is required. UML meets the following requirements:

- A formal definition of a common MOF-based meta-model that specifies the abstract syntax of the UML. The abstract syntax defines the set of UML modeling concepts, their attributes and their relationships, as well as the rules for combining these concepts to construct partial or complete UML models.

- A detailed explanation of the semantics of each UML modeling concept. The semantics define, in a technology-in-dependent manner, how the UML concepts are to be realized by computers.
- A specification of the human-readable notation elements for representing the individual UML modeling concepts as well as rules for combining them into a variety of different diagram types corresponding to different aspects of modeled systems.
- A detailed definition of ways in which UML tools can be made compliant with this specification. This is supported (in a separate specification) with an XML-based specification of corresponding model interchange formats (XMI) that must be realized by compliant tools.

2.2.3. Meta-Object Facility (MOF)

UML, the Unified Modeling Language, allows a model to be constructed, viewed, developed, and manipulated in a standard way at analysis and design time [10]. In addition, the Meta-Object Facility (MOF) standardizes a facility for managing models in a repository. MOF provides a metadata management framework, and a set of metadata services to enable the development and interoperability of model and metadata driven systems. Examples of these systems that use MOF includes modeling and development tools, data warehouse systems, metadata repositories, etc.

MOF has contributed significantly to some of the core principles of the emerging OMG Model Driven Architecture. Building on the modeling foundation established by UML, MOF introduced the concept of formal metamodels and Platform Independent Models (PIM) of metadata as well as mappings from PIMs to PSM.

2.2.4. XML Metadata Interchange (XMI)

XML Metadata Interchange (XMI) is an OMG standard that maps the MOF to the W3C's extensible Markup Language (XML). It specifies how MOF compliant models could be represented in form of XML tags, helping in the translation of MOF based meta-models into

XML Document Type Definition (DTD) thus allowing models to be translated into XML documents that are in accordance with correspondent DTDs. This XML based XMI data provides a serialized mechanism of artifacts produced by the UML and MOF models be able to interchange between various tools and repositories [13].

It can be used for any metadata whose metamodel can be expressed in Meta-Object Facility (MOF). The most common use of XMI is as an interchange format for UML models, although it can also be used for serialization of models of other languages (metamodels).

2.3. Code Generation

Code generation is using one application to build code for another application. Automatic code generation, be it data access layers, business entity classes, or even user interfaces, can greatly increase developer productivity and code quality. This process of generation can be based on a number of inputs, such as a database, an arbitrary XML file, a UML diagram, etc.

Categorization of Code Generators

The size and scope of code generators varies from short custom made scripts or programs that run in console mode with minimal input providing predefined output, to visual enterprise tools that can generate entire systems from complex model definitions [14]. In general, there are two main categories of code generator: *passive and active*. Passive code generators build the code once, and then have nothing more to do with the code. It is up to the discretion of the user as to how to update and maintain the code. Active code generators, on the other hand, keep track of the code during its lifecycle. Active code generators are run on code multiple times during the lifecycle. With Active Code generators, there is code you can modify, and code that should only be modified by the code generator. Code generators can be further classified into code mungers, inline code expanders, partial class generators and tier generators.

Code mungers

Code mungers take existing code and generate new code. Code mungers generators do not have their own UML development environment. Instead, they use UML specifications made in

other tools as input in the form of XMI or some other interchangeable format [15]. A common code munger is JavaDoc which is a documentation generator from Oracle Corporation for generating API documentation in HTML format from Java source code. Another known one is XDoclet which, based on the principles of JavaDoc, reads the comments in Java source files and generates Enterprise Java Beans out of them.

Inline code expanders

Inline code expanders take existing code and elaborate certain sections of that code. One of the most well known code expander is *Iron Speed Designer* which expands HTML code with ASP tags and code-behind files.

A partial class generator

A partial class generator is different from the previous code generators. It takes an abstract definition as input instead of code. It then outputs code which the user extends by creating derived classes and extending methods. For example, ExpertCoder is a toolkit that can generate expert systems that convert UML models to CodeDOM objects. If UML models are required for the software development anyways, this can save a lot of time, for no effort. Turning models into code is done through a series of transformations. This kind of generators can be constructed based on MDA specifications.

Tier generators

Tier generators seem to be the most common type of code generator. Tier generators build complete output code from an abstract definition. Tools in this category are template–based and flexible. They rarely provide a graphical language for specification definition and instead they rely on database metadata and tabular metadata inputs, making them non-intuitive and awkward.

Chapter Three: Related Work

Currently, it is possible to find tools that make it possible to define both Model to Model (M2M) transformation and Model to Text (M2T) transformations. Some of these tools support only M2T transformation while others support both M2M and M2T transformation. Among these tools some of them are open source and some are commercial. Below, the most commonly used code generators are presented:

3.1. Open Source code generators

MOFScript: a model to text transformation tool, based on one of the OMG MOF Model to Text Transformation specification. It is an Eclipse plug-in based on metamodels/models in EMF.

OpenArchitectureWare: a flexible, template-based generator framework integrated with XMI.

OpenMDX: an open source MDA environment, which integrates with several tools through XMI and supports code generation towards several target platforms (J2EE, .Net).

AndroMDA: an open source template-based tool for J2EE code generation from UML/XMI. It uses VTL (Velocity Template Engine) as scripting language and Netbeans MDR as a model API.

XDoclet: an open source attribute based code generation tool for J2EE. Not really model-based, but can be combined with generation tools such as UMT to achieve good model-based value.

ArgoUML is an UML diagramming application written in Java and released under the open source Eclipse Public License. Some of its features are: XMI support, model construction (only class diagram and use-case diagrams are more or less fully implemented.), Forward engineering (code generation supports C++ and C#, Java, PHP 4, PHP 5, Python, Ruby and, with less mature modules, Ada, Delphi and SQL), Reverse engineering / JAR/class file import.

Acceleo is a code generator transforming models into code (MDA approach). It provides "off the shelf" generators (JEE, .Net, PHP…) and template editors for Eclipse. Acceleo is open source software based on the Eclipse platform.

3.2. Commercial code generators

ArcStyler: is a commercial MDA tool from Interactive Objects. It is bundled with MagicDraw UML-tool, but can also support other UML-tools through tool adaptors.

OptimalJ: a commercial product from Compuware, uses a notation of patterns to achieve PSM transformations. Has an integrated UML tool for analysis, but uses a slightly different notation (structural) for the MDA-part of the tool.

MetaEdit+: an integrated modeling and metamodeling tool for defining languages and code generators. Supports XML and SOAP/Webservice interfaces for both the metamodels and models.

iQgen 3.0: is a template-based generator that can import models in different formats, including XMI, XML and ECore, and uses templates written as JSP templates

3.3. Summary

Most of the above and other code generators do not have advance UML diagram editor they depend on UML models of other application or a database tables and almost all of them does not generate code for MVC design pattern.

Moreover, Henok [6] designed and implemented a code generator for model-driven software development. The code generator has UML modeling interface for three UML diagrams (use case, sequence and class diagram). The code generator is designed to generate structural code from a class diagram. It also includes the reintegration of the modification of the generated code into the model of the system being developed. Melese [7] enhanced the work of Henok by adding package diagrams to the model construction module and modified the code generator algorithm to support a package diagram. However, both the original work [6] and the enhanced one [7] lack code generation from single platform Independent model to multiple platforms with consideration of design pattern, generation of configuration files that ease deployment and configuration activities , generation of database script and lack the considerations of MDA specifications such as MOF and XMI.

The source code generator presented in this paper is based on UML, MOF and XMI specifications. It will not relay on an existing UML tools for delivery of UML capabilities. Instead, it will include a UML diagram editor for model design. The code generator generates code from a single platform independent model to multiple platforms by adopting model transformation (from PIM to PSM).

Having all the features of other code generators, this tool will forward code generation by adding design pattern support mainly MVC.

Chapter Four: System Analysis

System Analysis involves a detailed study of the current system to find out what features the current system has and what additional features should be included. This will lead to specifications of a new system. It includes: identifying main requirements for the system and model the identified requirements using modeling techniques and tools.

This chapter discusses the identified functional and non-functional requirements of the system and the models (use case diagram, class diagram and sequence diagram) used to present the structural and dynamic behavior of the system.

4.1. Functional Requirements

Functional requirements specify specific behavior or functions that a system or component must be able to perform. Main functional requirements of the system include:

- Provision of extended diagram editing and zooming environment.
- Printing, saving and loading diagrams designed by the user.
- XMI support (importing and exporting XMI file for model interchanging among different modeling tools).
- Transform action of the PSM into code with improved and extended domain of the code generation.
- Code generation support for MVC design pattern.
- Integration of configuration files with the generated code.
- Generate database scripts (DDL and stored procedures).
- Support of reveres engineering (from code to model).

4.2. Non-Functional Requirements

Non-functional requirements specify all the remaining requirements not covered by the functional requirements. They specify criteria that judge the operation of a system, rather than specific behaviors. The Non-Functional requirements of the system are described as follows:

- **Performance:** The system should be responsive to user commands and minimize time taken to perform operations.
- **Usability:** Tools and commands in the user interface should be recognizable and easily accessible with relatively few interactions.
- **Reliability:** The system should not be vulnerable to unintended data loss.

4.3. System Models

System models are abstract descriptions of systems whose requirements are being analysed. System models help the analyst to understand the functionality of the system and models are used to communicate with customers. In this section we present use case diagram, class diagram and sequence diagram of the system.

4.3.1. Use case Diagram

The use case diagram is used to describe the proposed functionality of the new system. It represents typical sets of scenarios that help to structure, relate and understand the essential requirements. Figure 4.1 depicts the use case diagram of the system.

Figure 4.1: Use case diagram

Actor Description

An Actor is a user of the system. This includes both human users and other computer systems. The system is expected to have one actor, as it is a single user system. All operations of the

system are initiated by this actor, named “developer”. The terms “developer” and “user” are used interchangeably to indicate the actor.

Use case Description

This section explains the use cases depicted on the above figure (Figure: 3.1) using natural language.

Use case name: Open Project

Actor: Developer

Description: Open a project from the location where it was saved before.

Precondition: A file representing the project must exist on the storage.

Flow of Events:

i. The user clicks either on the open project button or double clicks the project file.
ii. The system prompts the user for the location of the file.
iii. The user selects a file location.
iv. The system reads data from the file and creates the project and its element in memory.
v. The system displays the model and its diagrams.

Post-Condition: List of diagrams within the model is displayed to the user.

Use case name: Save Project

Actor: Developer

Description: Save the project on a specified storage location.

Precondition: A project must be created or modified.

Flow of Events:

i. The user clicks on the save button or close the main window without clicking the save button.
ii. If the user action is for the first time system prompts the user for the file location.
iii. The user selects a file location and adds a project name.

iv. The system saves the save to the specified file location.

Post-Condition: The project and all of its elements are saved.

Use case name: Add Diagram

Actor: Developer

Description: Add a new diagram to the model (like use case diagram, class diagram, …)

Precondition: 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 default diagram name (“Untitled”) under the current active model located at the model explorer.

iii. The system shows diagram elements on the palette provided.

Post-Condition: A diagram is added and its name is shown under the active model on the model explorer.

Use case name: Remove Diagram

Actor: Developer

Description: Removes a diagram and all its elements from the model.

Precondition: A diagram must be added to the model.

Flow of Events:

i. The user right clicks a diagram from the list.

ii. Select on the remove option from the context menu.

iii. The system removes the specified diagram from the model.

iv. The system updates the displayed diagram list on the model explorer window.

Post-Condition: The diagram is removed from the model.

Use case name: Display Diagram

Actor: Developer

Description: Shows the diagram elements in the diagram editor and enables diagram

editing commands.

Precondition: 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 model explorer and clicks on the diagram name.
ii. The system retrieves the diagram from the model using the selected name.
iii. The system displays (enables) diagram editing commands.

Post-Condition: The diagram is removed from the model.

Use case name: Add Element

Actor: Developer

Description: Adds an element to the active diagram and displays it in the diagram editor.

Precondition: A diagram must be active.

Flow of Events:

i. The user clicks on the palette that indicates the type of the element.
ii. The user drags the element to the appropriate place in the diagram window.
iii. The system creates a new element in the model.
iv. The system creates a new diagram element in the diagram.
v. The system shows the diagram element in the diagram editor.

Post-Condition: The new element is shown in the diagram editor.

Use case name: Remove Element

Actor: Developer

Description: Removes an element from a diagram.

Precondition: A diagram containing at least one element must be active.

Flow of Events:

i. The user selects a diagram element and right clicks on the diagram.
ii. The system displays a context menu.
iii. The user selects the remove option.

iv. The system removes the diagram element from the diagram.

v. 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.

Use case name: Modify Element Properties

Actor: Developer

Description: Allows the user to change properties of an element

Precondition: An 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.

Use case Name: Generate Code

Actor: Developer

Description: Generates source code from transformed platform specific model object and stores it in a file.

Precondition: PSM must be generated from the model.

Flow of Events:

i. The user clicks on the generate code menu.
ii. The system display option window.
iii. The user provides all the options for code generation (options includes: platform, design pattern, file location) and clicks generate button [Alternate A].
iv. The system generates the PSM.
v. The system uses the generated PSM and generates a code according to the user options.
vi. The system writes the code to files at specified location.
vii. The system displays a message dialog indicating the completion of the process

Post-Condition: The generated code saved permanently on the file location specified.

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.

Use case Name: Import Classes

Actor: Developer

Description: Read class files and generate a model object.

Precondition: A file must be a valid class file

Flow of Events:

i. The user clicks on the Import classes menu.

ii. The system prompts the user for the location of the file.

iii. The parser read the data from the file and create model object.

iv. The system displays the model and its diagram.

Post-Condition: List of diagrams within the model is displayed to the user.

Alternate A If the user enters an invalid location and clicks on import classes menu.

A.1. The system displays an error dialog.

A.2. Returns to the dialog ii.

Use case Name: Generate database scripts

Actor: Developer

Description: Generate database definition languages and stored procedures from a model.

Precondition: A model must be either created or loaded

Flow of Events:

i. The user clicks on the generate database script button.

ii. The system display database script option dialog window.

iii. The user chooses deferent options and clicks on generate button [Alternate A].

iv. The system generates the scripts and save it permanently.

Post-Condition: A file is saved containing the generated database scripts.

Alternate A If the user enters an invalid file format.

A.1. The system displays an error dialog

A.2. Returns to the dialog iii.

Use case Name:	Change setting
Actor:	Developer
Description:	Changes setting used in the code generation and integration functionality
Precondition:	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.

Use case Name:	Import XMI
Actor:	Developer
Description:	Import an XMI file created by this tool or other tool for further processing.
Precondition:	A file must be a valid XMI file
Flow of Events:	v. The user clicks on the Import XMI menu. vi. The system prompts the user for the location of the file. vii. The parser read the data from the file and create model object. viii. The system displays the model and its diagram.
Post-Condition:	List of diagrams within the model is displayed to the user.

Use case Name:	Export XMI
Actor:	Developer
Description:	Save the XMI version of the model on a specified storage location
Precondition:	A project must be either created or loaded
Flow of Events:	i. The user clicks on the Export XMI menu. ii. The user selects a file location and adds a file name.

iii. The system saves the file to the specified file location.

Post-Condition: The XMI version of the model and its elements are saved.

4.3.2. Structural Model (Class Diagram)

The class diagrams of the system are presented in Figures 4.2 – 4.5. Figure 4.2 shows the core classes of the system followed by their description.

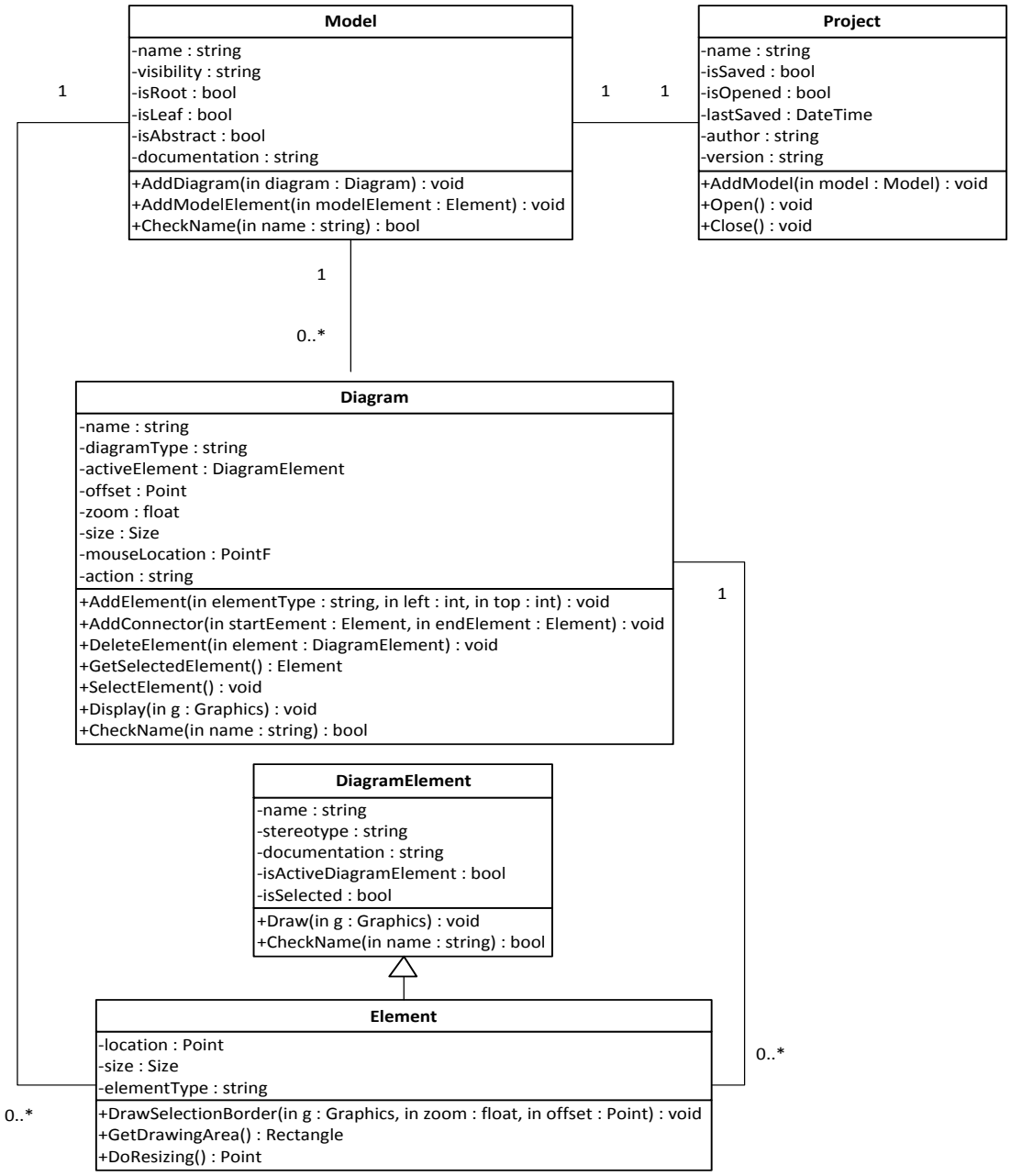


Figure 4.2: Class Diagram

Project – is a top level class which holds a UML model object. It has operation for adding a model, closing and opening a project. When a program starts one model will be add by default to the project.

Model – is the main class of the system which indirectly includes objects of all other classes. A model object includes list of diagrams and model elements. Attributes such as visibility, isRoot, isLeaf, isAbstract and documentation are added to facilitate model exchange among different UML modeling tools. Model maintains uniqueness of diagram and element names.

Diagram – represents a single UML diagram that holds diagram elements. This class monitors and controls every activities of the user on diagram element. The diagramType attribute of this class is set at creation and never changes. Diagram Type can be: Use Case, Class, Sequence, Activity and the like. Attribute zoom is added to facilitate zooming functionality on diagram elements.

DiagramElement – is a base class which includes common attributes and operations of all diagram elements. Diagram elements can be shape element or connector.

Element – this class inherits all properties and operations of DiagramElement class and also it is a base class for all the classes that represent a model element. The ElementType property of this class is set at creation and never changes. Element type can be: Actor, Use Case, Class, package and the like. An element can be added on a diagram or directly to the model.

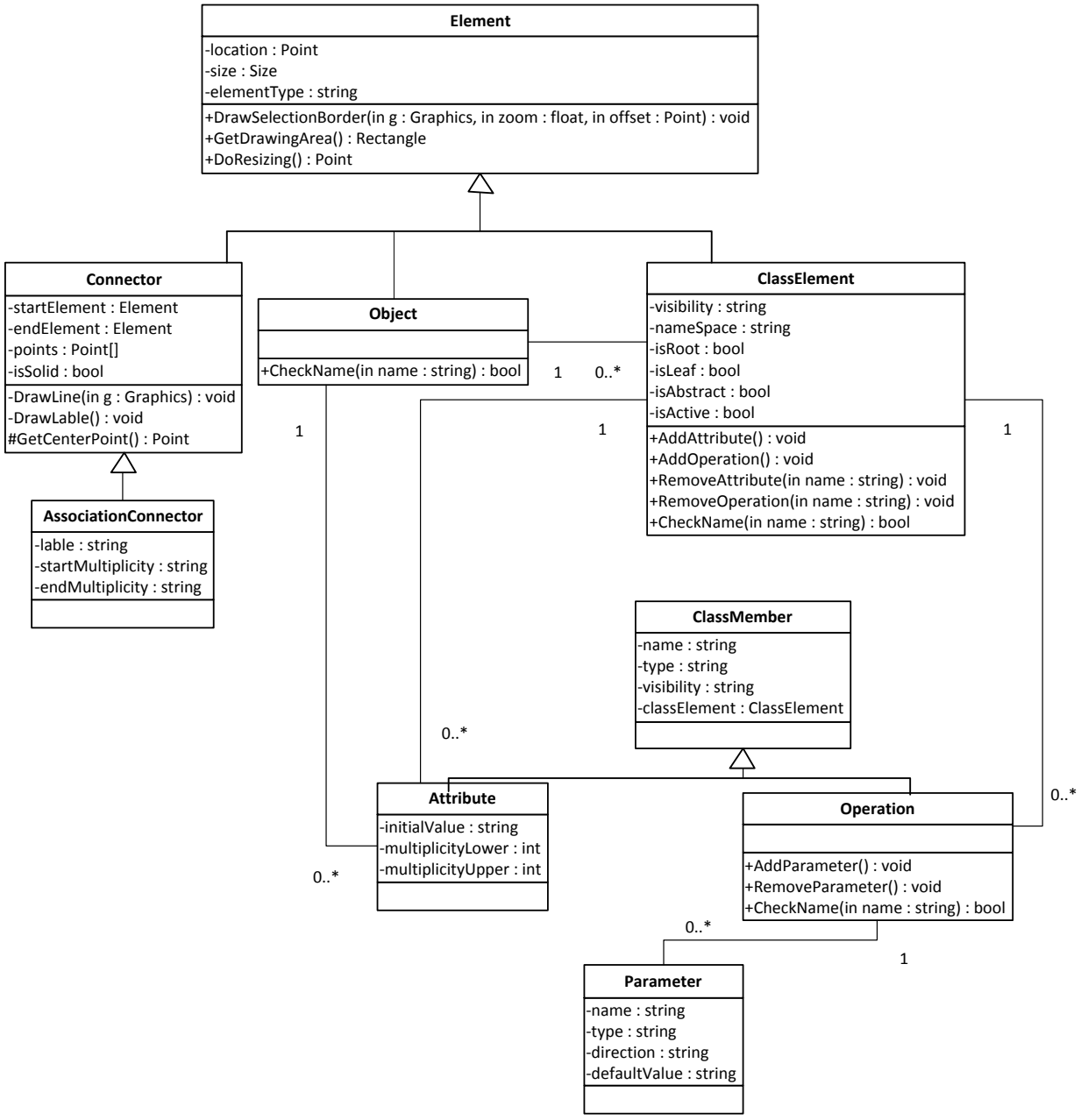


Figure 4.3: Class diagram

ClassElement – a subclass of Element class which represents a class model element. 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. The attribute called namespace is added in order to identify the package in which the class is found.

ClassMember – 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 subclass of ClassMember which represents a single attribute of a class and includes attributes for setting initial value and multiplicity.

Operation – a subclass of ClassMember which represents a single operation of a class. It adds parameter information as a list 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 default value. Direction can be: *in, out, inout* or *return*.

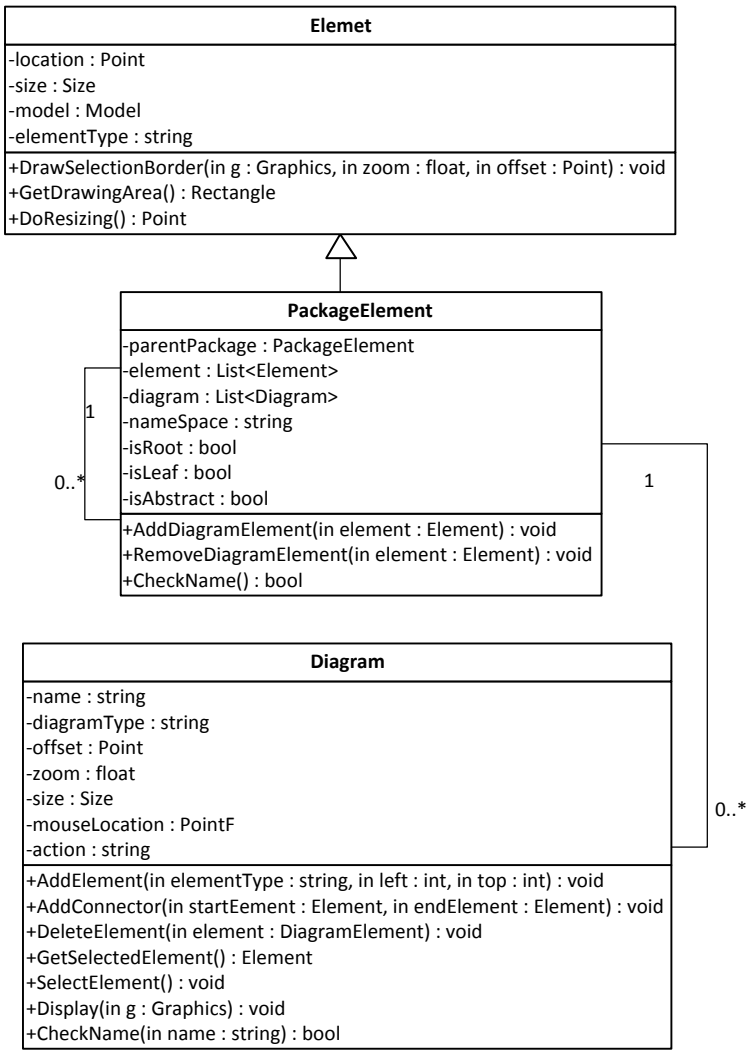


Figure 4.4: Class diagram

PackageElement – a subclass 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. This class reference itself to maintain parent chilled relationship among packages.

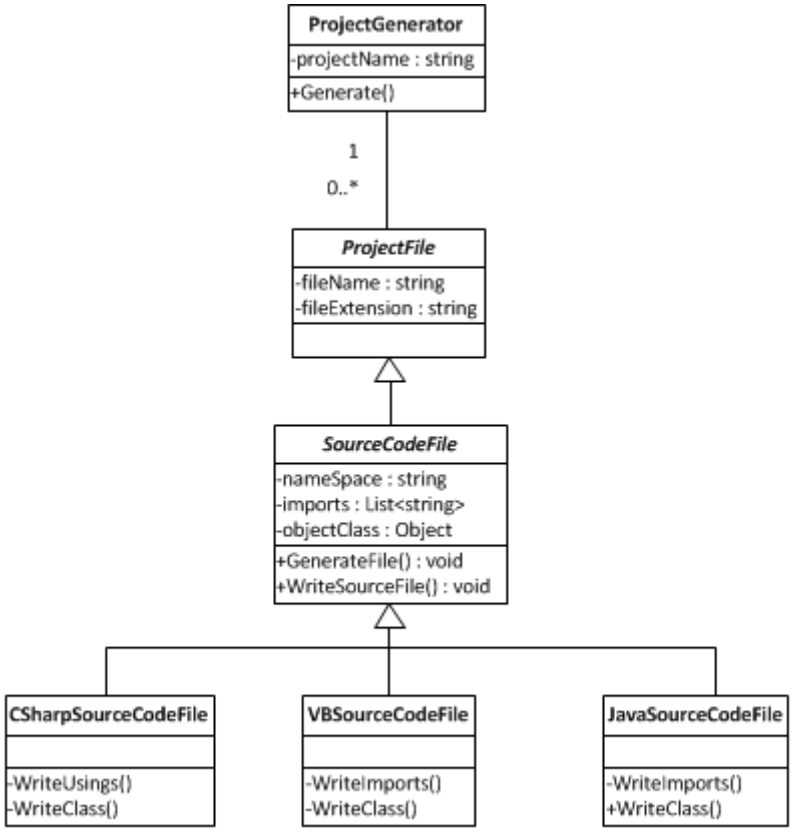


Figure 4.5: Class diagram

ProjectFile: is an abstract class that represents all files of a project. A file can be source code file or other file used by platform. A project file has file name and file extension.

SourceCodeFile: is a super class which represents a source code file. A source code file includes namespace, lists of imports and the class in which the file represent.

4.3.3. Behavioral Model (Sequence Diagram)

Sequence diagrams are used to show how objects interact in a given situation. Sequence diagrams of some use cases are presented in Annex 1-3.

Chapter Five: System Design

5.1. Design Goal

To make the system acceptable and usable by the target users we set the following three major design goals:

Usability: The tool must be easy to use, and the code it generates should be ready-to-use; that is, the user should have to make a minimum of changes to make the generated code work in his application.

Response time: The system should respond quickly to each user's commands during designing and code generation.

Extensibility: The system is designed to enable new functionalities like supporting additional programming languages and design patterns without affecting its current functionality. A minor change in functionality should incur a correspondingly small implementation effort. But even major changes should be well-contained:

5.2. Software Architecture

In this section, we describe the general software architecture of the system presented in Figure 5.1. Each components of the system architecture are described as follows:

Model Constructor: The model constructor has the task to construct a model object when a user designs a UML Model.

XMI Parser: The XMI Parser extract the XMI file generated by other UML modeling tool and create a model object which is used by the code generator.

Class Parser: The class parser extracts class files or DLL/EXE and creates a model object which is used by the code generator.

Model Transformer: The model transformer has the task to convert the model object into another platform specific model object targeting the platform in which the user is selecting for code or script generation.

XML: XML (Extensible Markup Language) is used to store model objects and system settings permanently. For writing XML data to a file, we adopt the in-memory XML processing technique because this technique is a good choice to perform operations on XLM content like searching, transforming and validating it.

XMI: XML Metadata Interchange XMI is an interchange standard for UML models from OMG. The main objective of the XMI is to facilitate the interchange of meta-data between UML and MOF-based modeling tools.

Generator: The generator task is to generate code or database script based on the platform a user select. The code generation is carried after a successful mapping of the model to a particular platform. The target code, such as Java, C#, VB can be generated by the development tool including the appropriate library files.

Store: The store converts the model object and project settings into XML file and save the file permanently on the storage this file also includes the shape properties of each model element. The store is also responsible for permanently saving a model object into XMI file.

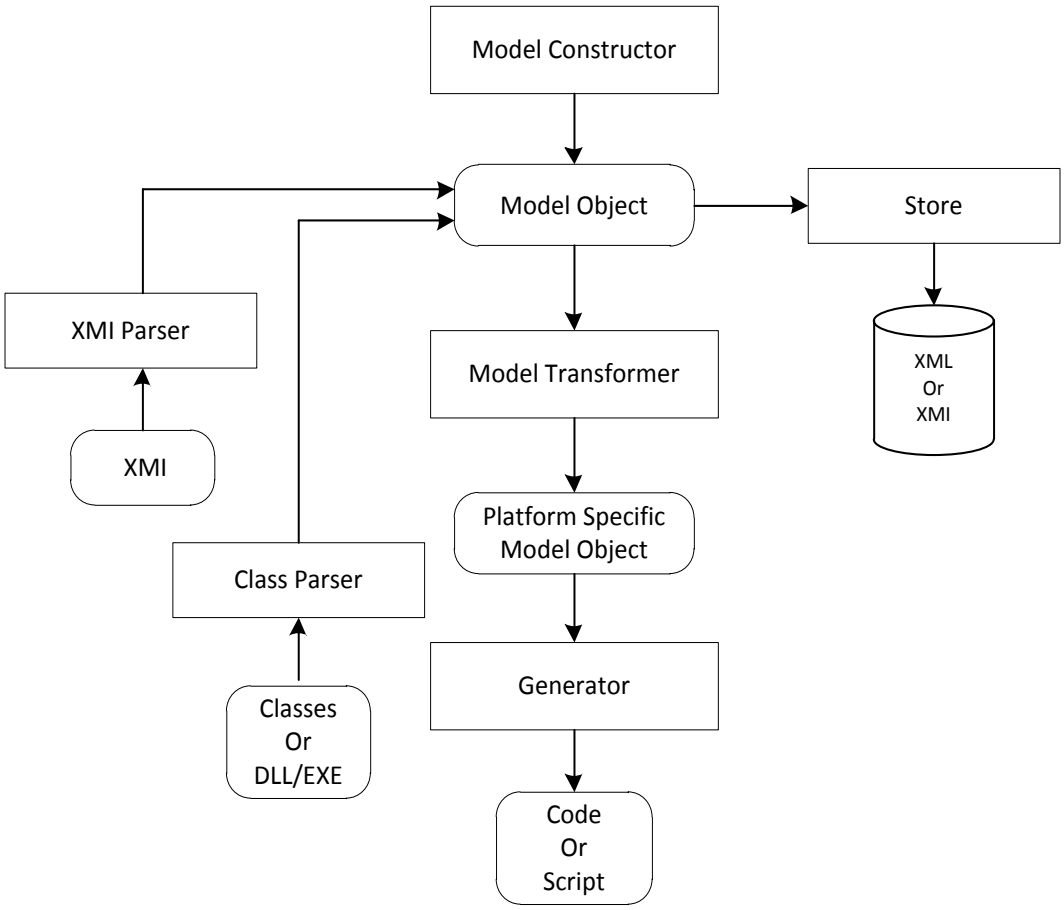


Figure 5.1: System Architecture

5.3. Subsystem Decomposition

The system is divided into seven subsystems. Each subsystem provides services to other subsystems. A service is a set of related operations that share a common purpose. The subsystems and their inter-dependencies are depicted in Figure 5.2.

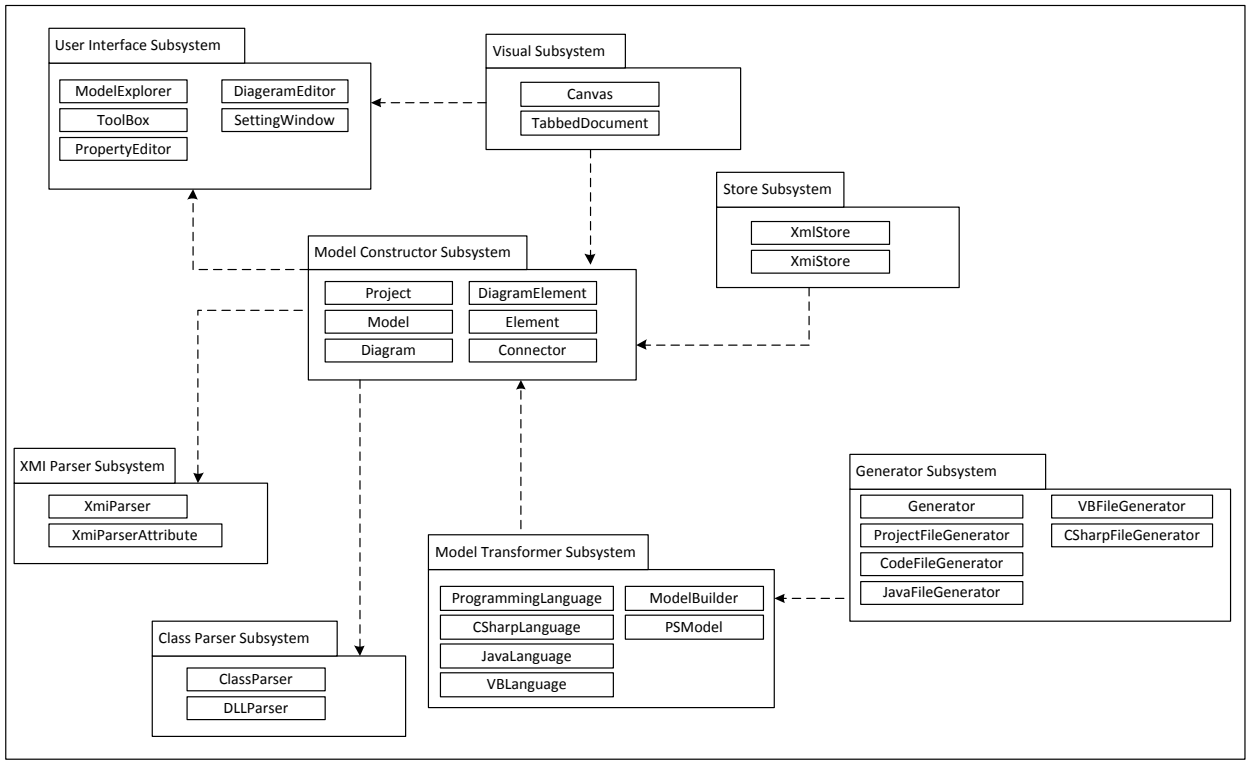


Figure 5.2: Subsystem Decomposition

User Interface Subsystem

The user interface subsystem provides UI components for displaying and editing diagram elements. Most of the UI components will be custom controls (controls that are created by the developer) for the purpose of flexibility and reusability. The UI subsystem components are:

ModelExplorer: which allows user to edit and view project items (Model, Diagram, and Diagram elements).

Toolbox: is used to show diagram elements such as Use case, Actor, Package and the like. To easily be accessible by the user we will classify them based on the UML diagram they belong.

PropertyEditor: is used to set properties of diagram elements.

DiagramEditor: is a UI component which is used to draw a UML diagram. The DiagramEditor has both zooming and scrolling functionality.

SettingWindow: is a UI component which is used for setting diagram properties such as: grids, diagram background colors, and other settings.

Visual Subsystem

The visual subsystem is responsible for creating a tabbed document interface that allows developers to draw and edit UML diagrams. The major components of this subsystem are Canvas and TabbedDocument.

Canvas: Canvas or graphical editor is a component containing different types of graphical objects responsible for viewing and editing.

TabbedDocument: is a major component of the visual subsystem, which binds the canvas with diagram and manages the tabbed document.

Model Constructor Subsystem

This is the core subsystem of the application which includes different classes that are responsible to create a project, model, diagram and diagram elements. Some of the classes of this subsystem are explained on Chapter Four on structure model (Class Diagram) section.

Storage Subsystem

This subsystem is responsible for storing a model object and project setting to XML file. The major two classes of this subsystem are XmlStore and XmiStore.

XmlStore: is responsible for writing model object to XML file.

XmiStore: is responsible for writing model object to XML file for XMI store.

XMI Parser Subsystem

This subsystem is responsible for creating a model object from XMI file. We use .NET Framework's Reflection API to dynamically create an instance of a model object from the XMI file. The two major classes of this subsystem are XmiParser and XmiParserAttribute. Due to time constraint the parser is capable only to parse class diagram elements (class and interface).

XmiParser: is a main class of XMI parser subsystem that parses a class and interface elements of a model from XMI file.

XmiParserAttribute: this class will register the name of an element and its properties that will be parsed by the parser.

Class Parser Subsystem

This subsystem is responsible for creating a model object from Class files. The two major classes of this subsystem are ClassParser and DLLParser.

ClassParser: is a main class of the system that parses class elements from a class file and return a class element object.

DLLParser: this class will extract classes and interfaces from DLL/EXE files and pass the object to ClassParser class.

Model Transformer

This subsystem is responsible for transforming platform independent model object to platform specific model object by binding language element to the input model object. The transformed model object will be an input for code generator. Some of the classes of this subsystem are:

ProgrammingLanguage: this class is used to set common properties that are shared by all the supported programming languages.

CSharpLanguage: is a sub class of ProgrammingLanguage class. This class will maintain the properties and operations of C# language.

JavaLanguage: is a sub class of ProgrammingLanguage class. This class will maintain the properties and operations of Java language.

VBLanguage: is a sub class of ProgrammingLanguage class. This class will maintain the properties and operations of Visual Basic language.

ModelBuilder: is a main class of this subsystem in which all the model transformation activities instantiated.

Generator subsystem

This subsystem generates source code from the model to different platforms and integrates different project libraries with the generated source code. The code generator subsystem also considers the MVC design pattern. It generates and integrates Model, View and Controller classes separately. This subsystem is also responsible for generating a database script that includes stored procedures and tables from class diagram by adopting model transformation. A stored procedure is a subroutine available to applications that access a relational database system. This subsystem generates stored procedure scripts for the four essential database operations: **C**reate, **R**ead, **U**pdate and **D**elete or sometimes referred as CRUD.

5.4. Persistent Data Management

The persistent data management of this system is based on the pervious projects [6, 7]. The system has four kinds of persistent output; the model, XML files, database script and code. The model construction subsystem classes hold the in-memory data of a model while the serialization operation makes it possible to save the information in XML file format.

The script will be generated to a folder with a single SQL server script file format that holds the SQL statements for creating stored procedures and tables. The generated code will be stored as text file format of a programming language the developer chooses during model construction. The system will generate a Visual Studio project and link the necessary library files in the project. The model, package and pattern hierarchy will also be considered during the process.

Chapter Six: Implementation

6.1. Tools

Because of the additional features the system is re-built again. For the system implementation we chose the C# programming language, Window form application, Visual Studio 2012 IDE, and .NET framework 4.0.

6.2. Prototype

In this section we present all the system functionalities and how the user interacts with the system by showing some of the interaction results with screen shots. In this presentation, the default names given by the system for each object are used to show the design environment and code generation result.

6.2.1. The Application Environment

The application starts with the start page window that displays lists of recent projects. At this stage most of the application menus are disabled. From this window a user can create new project or open an existing project by using File menu or simply clicking on the links.

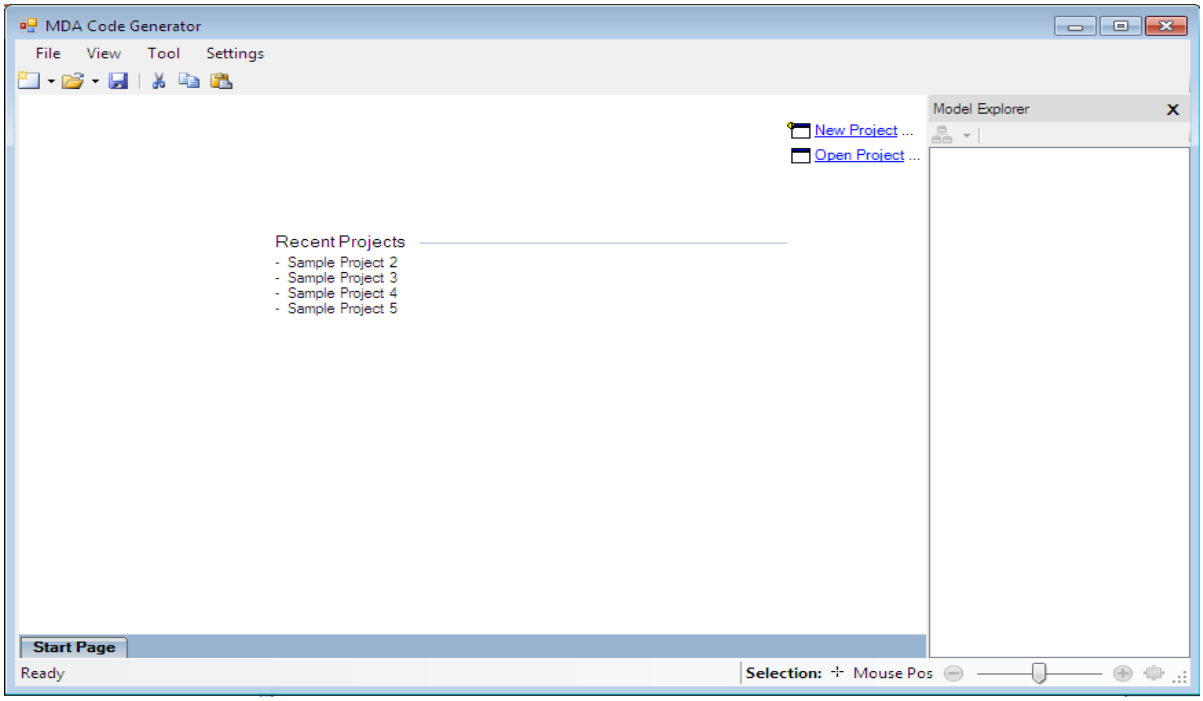


Figure 6.1: Start Page Window.

Main Window

The main window user interface allows a user to do all the activities of the system. This window is composed of menu, status bar, toolbox, model explorer and diagram editor. Figure 6.2 shows the main window of the system.

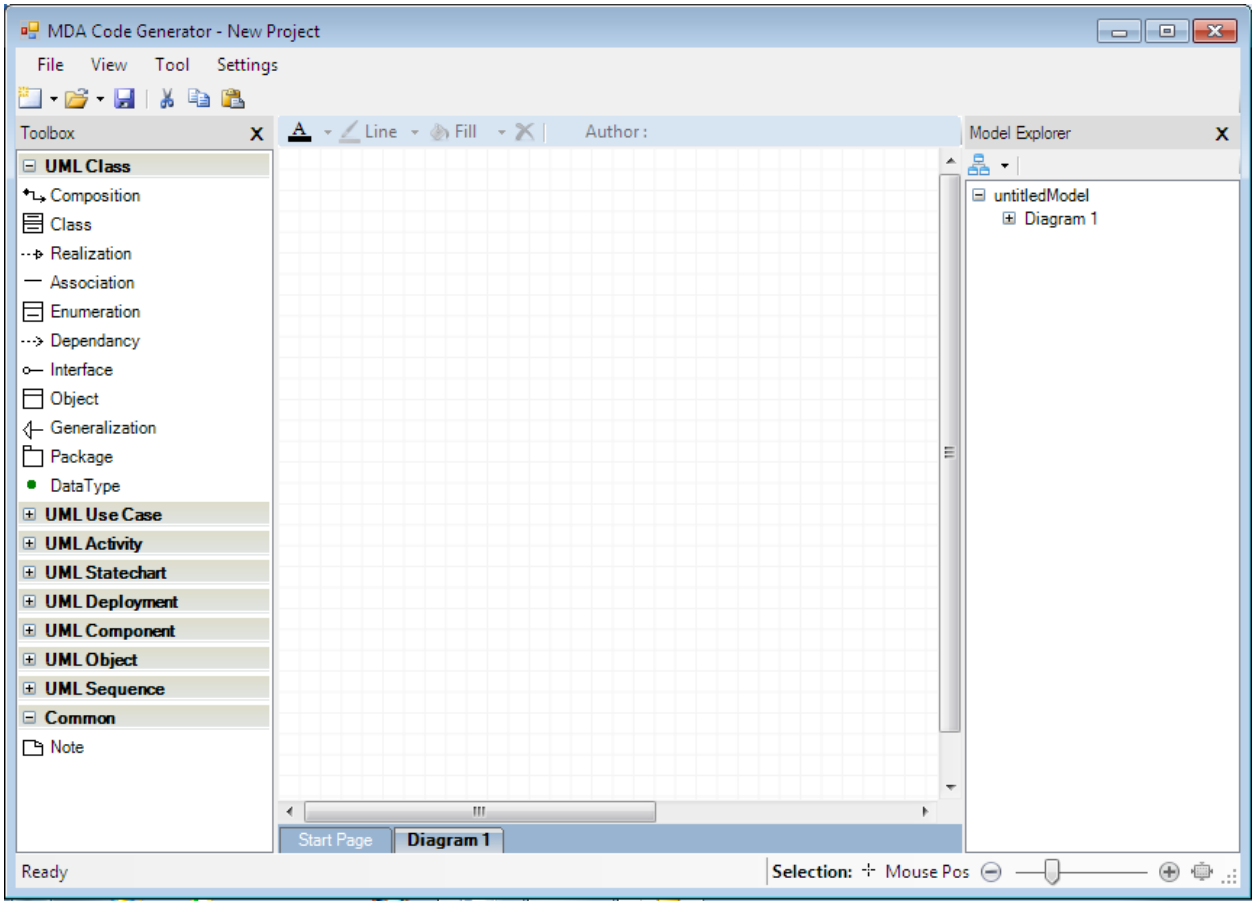


Figure 6.2: Main Window.

Menus

The menu bar at the top of the window contains the Fill, View, Generation, and Setting menus. Each of these menus and their sub menu items are discussed bellow.

File: this menu contains sub menus items which are used for manipulation of project at run time.

View: is used to show or hide Model Explorer and Toolbox windows.

Tool: this menu includes sub menu items for code and script generations. The tool menu includes Generate Code, Generate Script, and Reverse Engineering menu items.

Setting: includes sub menu items that are used for changing different settings of the application.

Model Explorer

Model Explorer is a window that is attached on the right hand side of the main window. This window is used to display model and its elements hierarchically. Initially this window is located at the right hand side of the main window but a user can change the location by dragging this window around the four corners of the main window.

Toolbox

Toolbox is used to list diagram elements of a model. Elements on toolbox are classified under the diagram they belong. The user can drag and drop an element from the toolbox to the diagram editor to draw a diagram element. This window is only displayed when the user execute create UML diagram command. The default location of this window is at the left hand side of the main window but a user can change the location by dragging this window on the four corners of the main window.

6.2.2. Diagrams

The system is designed and implemented to support eight UML diagrams. The prototype of some of these diagrams will be presented in this section.

Use Case Diagram

A use case diagram of the system contains Actor, Use case, Include, Extend and Association elements. There is also a boundary element in which all the use cases are contained. Figure 6.3 shows the state of the system in editing a use case diagram.

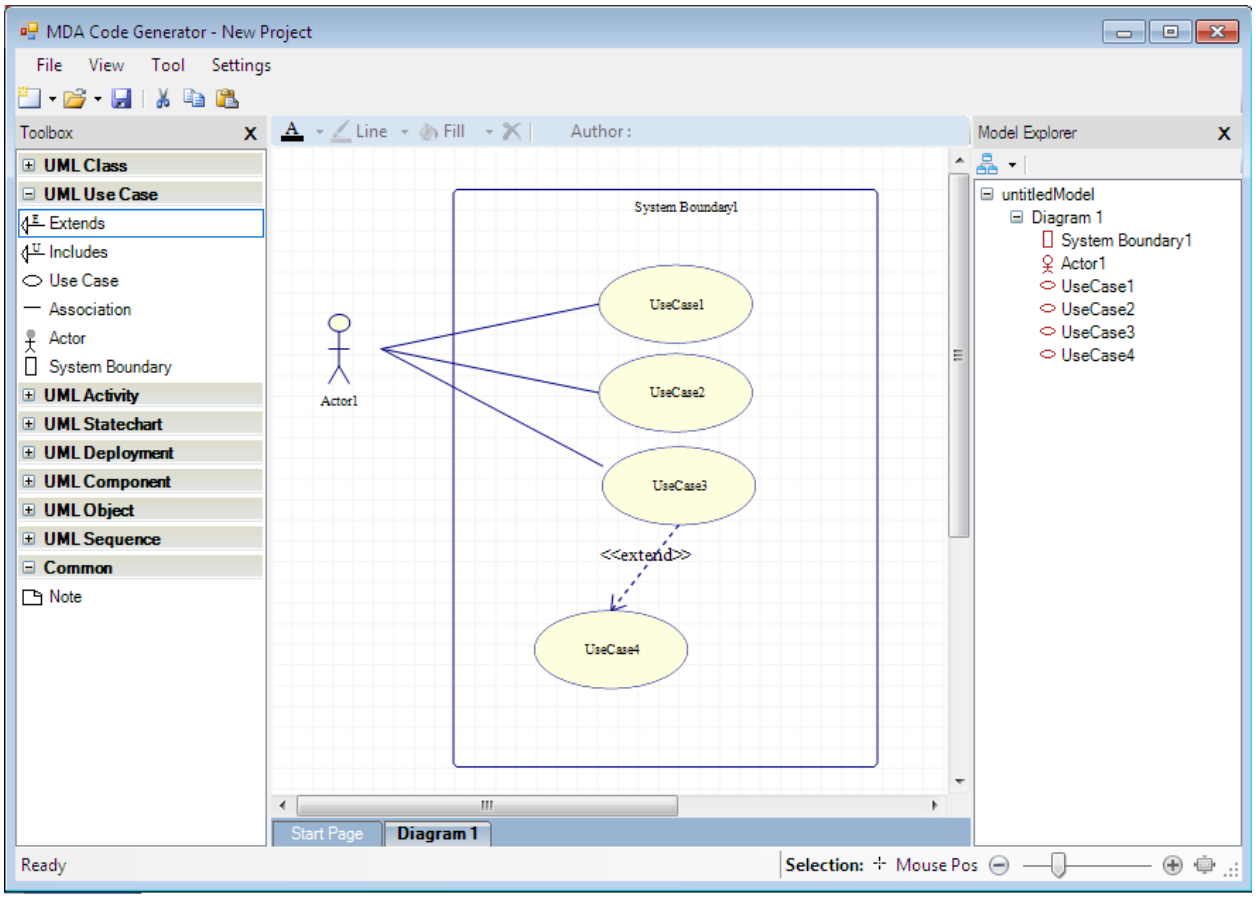


Figure 6.3: Use Case Diagram.

Sequence Diagram

A sequence diagram of the system includes basic sequence diagram elements such as: Object Lifeline, Activation, Lifeline, Message (call) and Message (return). Figure 6.4 shows the state of the system in editing a sequence diagram.

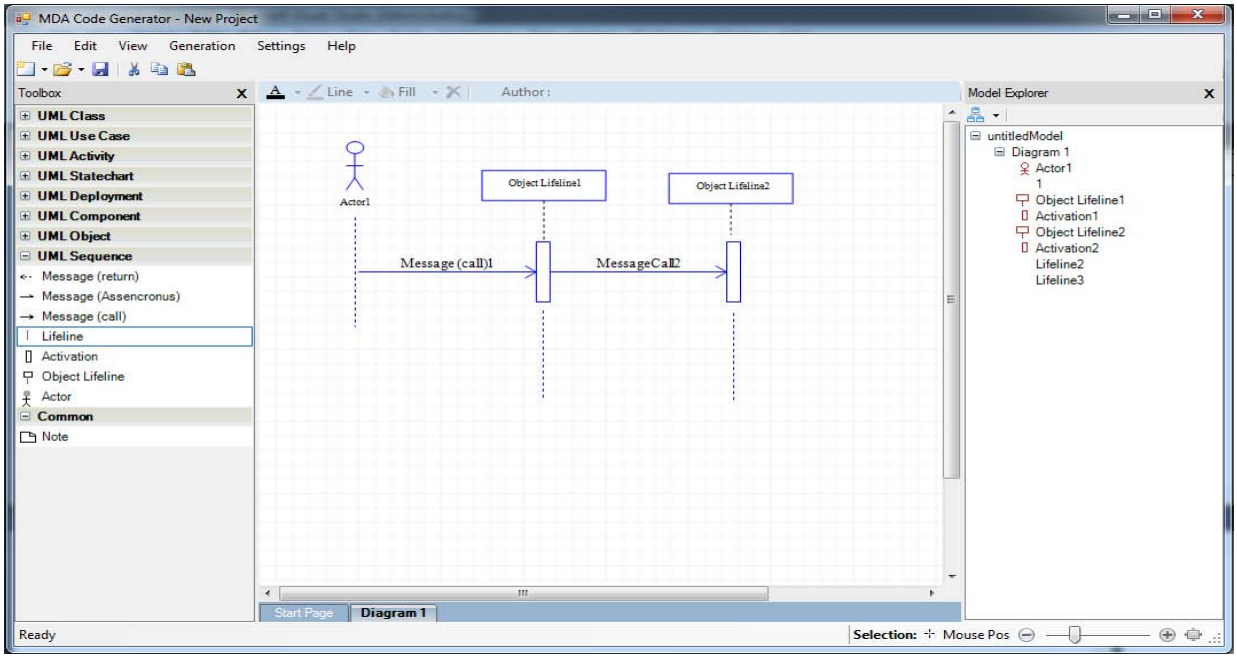


Figure 6.4: Sequence Diagram.

Activity Diagram

Activity diagram of the system includes basic activity diagram elements such as: Activity, Transition, Decision, Initial state, Final State, Horizontal fork/join, Vertical fork/join. Figure 6.5 shows the state of the system in editing activity diagram.

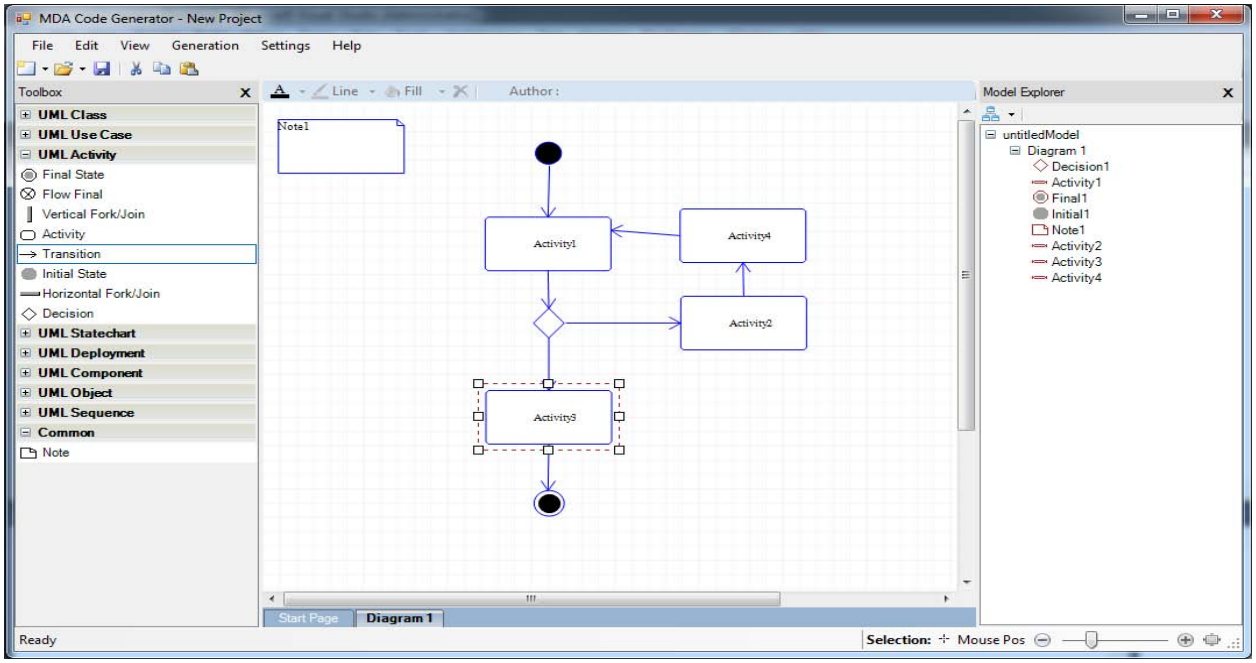


Figure 6.5: Activity Diagram.

Deployment Diagram

Deployment diagram of the system includes basic deployment diagram elements such as: Node, Component, Artifact, and dependency. Figure 6.6 shows the state of the system in editing deployment diagram.

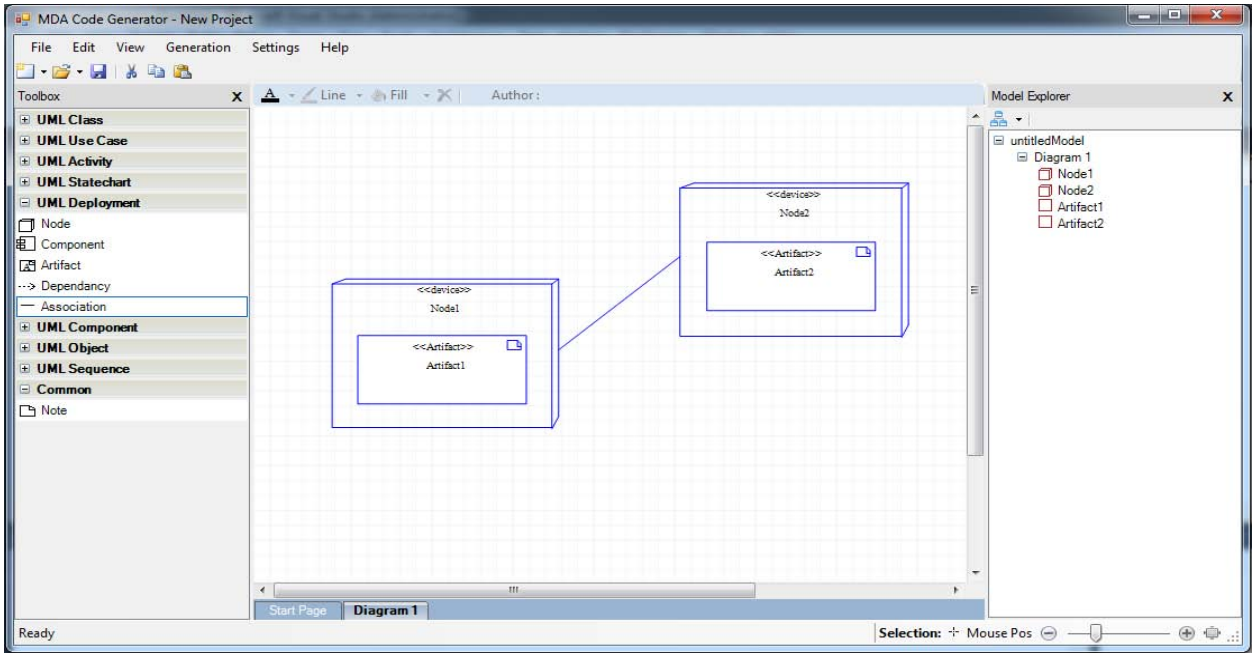


Figure 6.6: Deployment Diagram.

Class Diagram

A class diagram includes Class, Interface, Object, Package, Enumeration and other elements. Figure 6.7 shows the state of the system in editing class diagram.

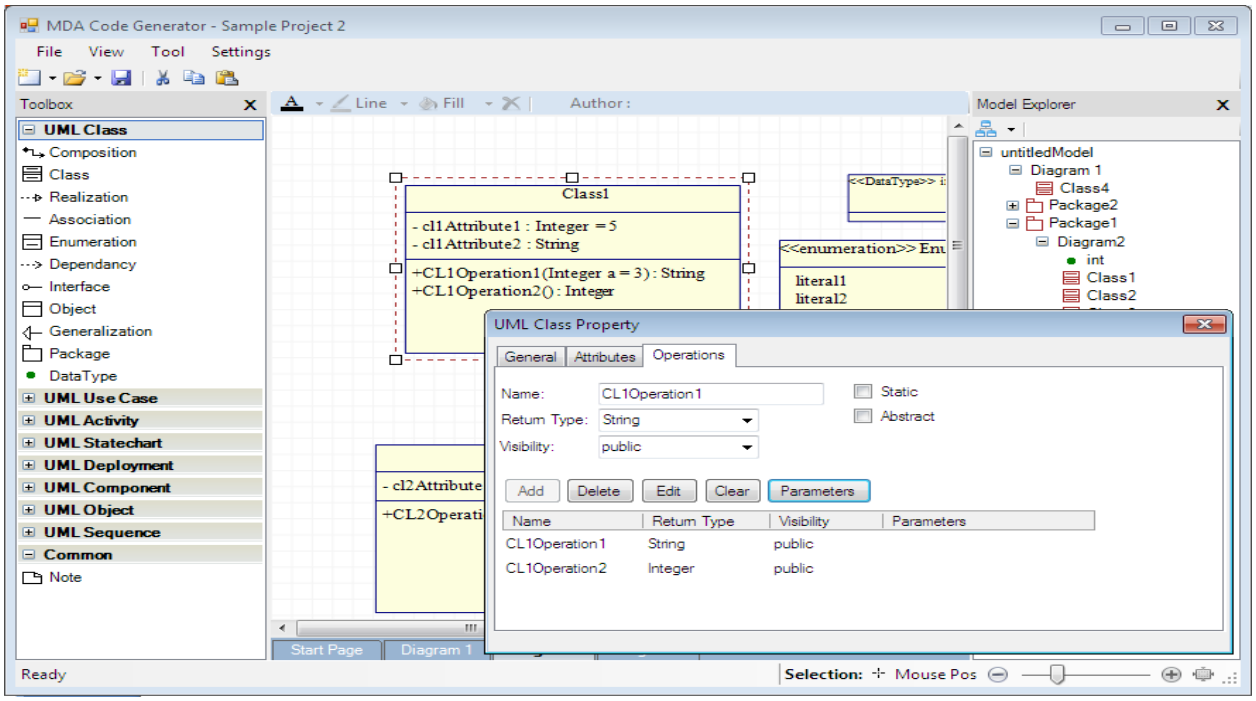


Figure 6.7: Class Diagram.

Package Element

Package is one of a common diagram element of a model. A package can contain Classes, Interfaces and even another package. Figure 6.8 shows a snapshot of package element.

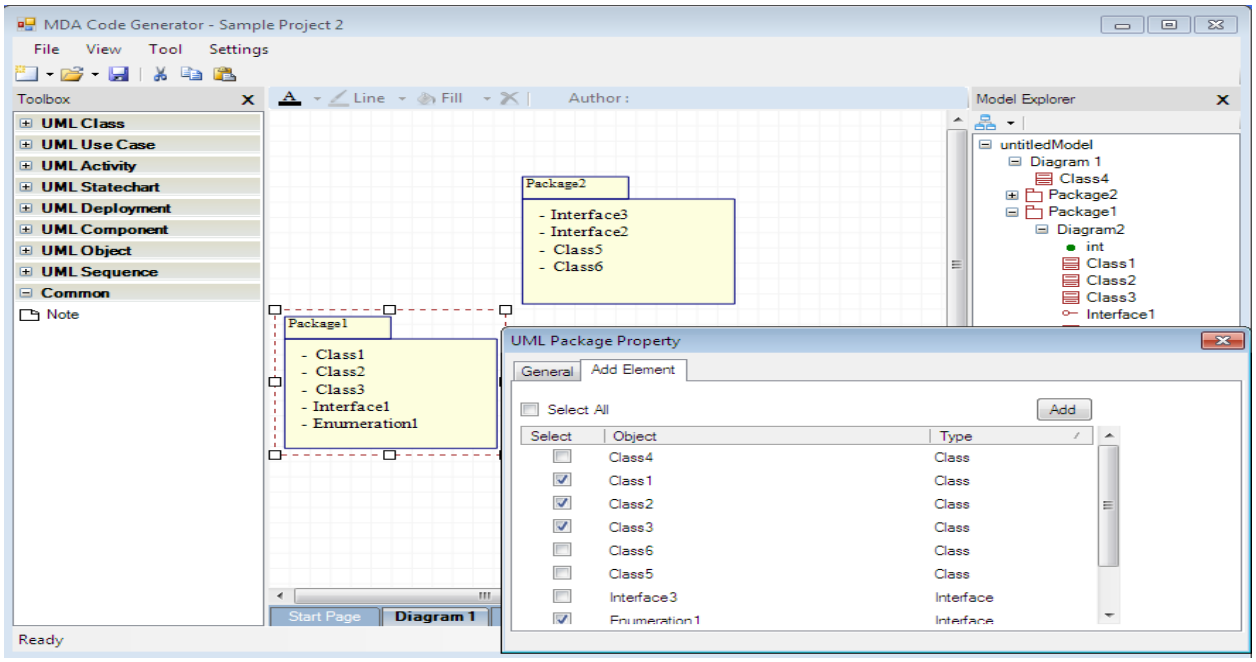


Figure 6.8: Package element.

6.2.3. XMI Support

Model designed by this tool can be easily accessible by other modeling tools that support XMI for model interchange and also a model designed by other modeling tool can also accessible by this tool.

Exporting a Model

A user can export a model by clicking on Export XMI menu item. The system will generate the XMI file which includes only model information. Table: 6.1. Shows sample XMI file generated by the system.

Table 6.1: Sample XMI file generated by the system.

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmi.version="1.2" xmlns:UML="org.omg.xmi.namespace.UML" timestamp="1.2">
 <XMI.header>
  <XMI.documentation>
   <XMI.exporter>MDA Code Generator</XMI.exporter>
   <XMI.exporterVersion>0.0.1</XMI.exporterVersion>
  </XMI.documentation>
  <XMI.metamodel xmi.name="UML" xmi.version="1.4" />
 </XMI.header>
 <XMI.content>
  <UML:Model xmi.id=".:-mda-code-generator-1" name="untitledModel" xmlns:UML="Model">
   <UML:Namespace.ownedElement xmlns:UML="Namespace">
    <UML:Class xmi.id=".:-mda-code-generator-2" name="Class1" visibility="public"
isSpecification="False" isRoot="False" isLeaf="False" isAbstract="False" isActive="Class1"
xmlns:UML="Class">
     <UML:Classifier.feature xmlns:UML="Classifier">
      <UML:Attribute xmi.id=".:-mda-code-generator-3" name="attrib1" visibility="private"
xmlns:UML="Attribute">
```

```
          <UML:StructuralFeature.multiplicity xmlns:UML="StructuralFeature">
            <UML:Multiplicity xmi.id=".:-mda-code-generator-4" xmlns:UML="Multiplicity">
              <UML:Multiplicity.range>
                <UML:MultiplicityRange xmi.id=".:-mda-code-generator-5" lower="1" upper="1"
xmlns:UML="MultiplicityRange" />
              </UML:Multiplicity.range>
            </UML:Multiplicity>
          </UML:StructuralFeature.multiplicity>
        </UML:Attribute>
        <UML:Attribute xmi.id=".:-mda-code-generator-6" name="attrib2" visibility="private"
xmlns:UML="Attribute">
          <UML:StructuralFeature.multiplicity xmlns:UML="StructuralFeature">
            <UML:Multiplicity xmi.id=".:-mda-code-generator-7" xmlns:UML="Multiplicity">
              <UML:Multiplicity.range>
                <UML:MultiplicityRange xmi.id=".:-mda-code-generator-8" lower="1" upper="1"
xmlns:UML="MultiplicityRange" />
              </UML:Multiplicity.range>
            </UML:Multiplicity>
          </UML:StructuralFeature.multiplicity>
        </UML:Attribute>
        <UML:Operation xmi.id=".:-mda-code-generator-9" name="Operation1" visibility="public"
xmlns:UML="Operation" />
      </UML:Classifier.feature>
    </UML:Class>
    <UML:Class xmi.id=".:-mda-code-generator-10" name="Class2" visibility="public"
isSpecification="False" isRoot="False" isLeaf="False" isAbstract="False" isActive="Class2"
xmlns:UML="Class">
      <UML:Classifier.feature xmlns:UML="Classifier">
        <UML:Attribute xmi.id=".:-mda-code-generator-11" name="attrib3" visibility="private"
```

```
xmlns:UML="Attribute">
          <UML:StructuralFeature.multiplicity xmlns:UML="StructuralFeature">
           <UML:Multiplicity xmi.id=".:-mda-code-generator-12" xmlns:UML="Multiplicity">
            <UML:Multiplicity.range>
             <UML:MultiplicityRange xmi.id=".:-mda-code-generator-13" lower="1" upper="1"
xmlns:UML="MultiplicityRange" />
            </UML:Multiplicity.range>
           </UML:Multiplicity>
          </UML:StructuralFeature.multiplicity>
         </UML:Attribute>
         <UML:Operation xmi.id=".:-mda-code-generator-14" name="Operation2" visibility="public"
xmlns:UML="Operation" />
        </UML:Classifier.feature>
       </UML:Class>
       <UML:Generalization xmi.id=".:-mda-code-generator-15" isSpecification="False"
xmlns:UML="Generalization">
        <UML:Generalization.child>
         <UML:Class xmi.idref=".:-mda-code-generator-10" xmlns:UML="Class" />
        </UML:Generalization.child>
        <UML:Generalization.parent>
         <UML:Class xmi.idref=".:-mda-code-generator-2" xmlns:UML="Class" />
        </UML:Generalization.parent>
       </UML:Generalization>
     </UML:Namespace.ownedElement>
   </UML:Model>
  </XMI.content>
</XMI>
```

Importing a Model

An XMI parser is designed and implemented to read and load XMI file generated by other model editor tool. Due to time constraint we limit the XMI parser scope to parse only class diagram elements.

6.2.4. Code Generator

To generate code from class diagram the user needs to click the Generation menu and click Generate Code sub-menu item. The system displays code generation option window. The user must specify the path where the codes and different project files will be saved. The user can specify the langue in which to generate the code. C#, VB and Java are the supported language. The default language is c#. The user can also specify design pattern optionally. Finally the user can select classes of a model that will be included in the code generation process. During the code generation process, the package class hierarchy is considered. Sample code generation results are presented in Table 6.2 – Table 6.3.

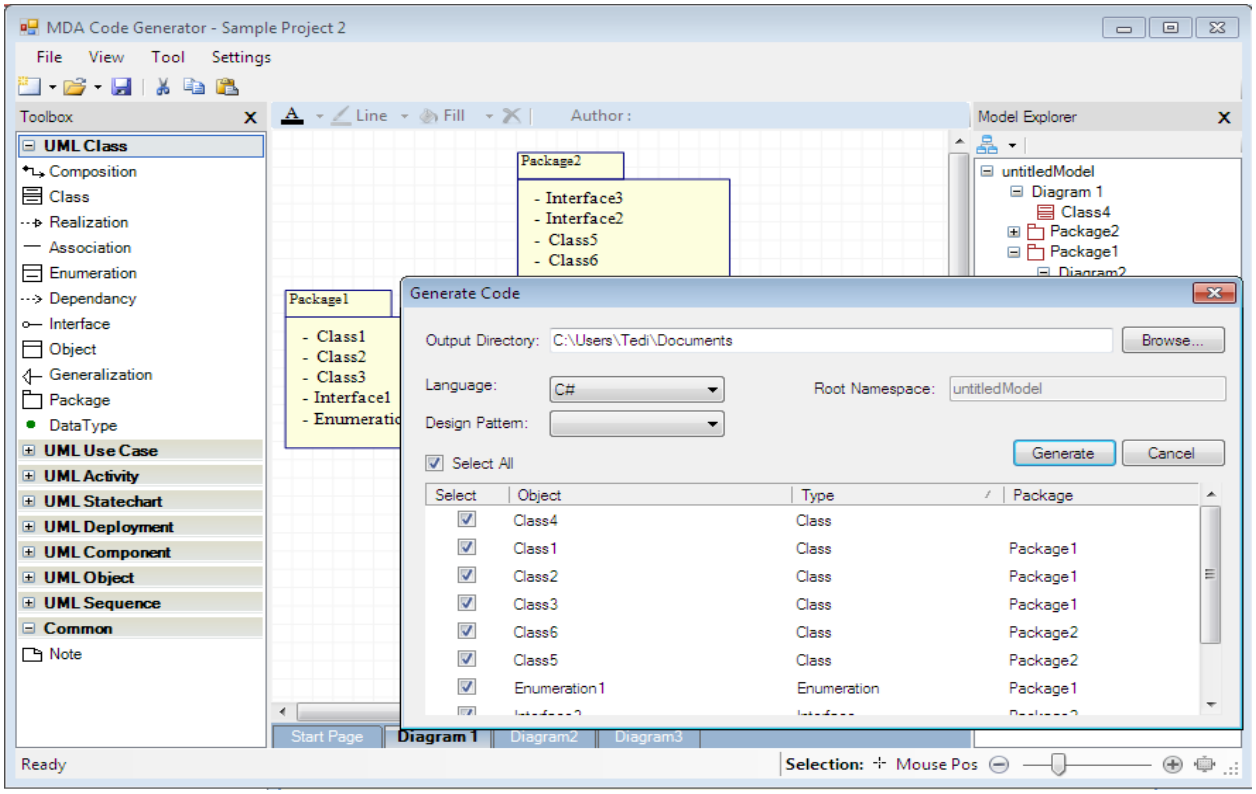


Figure 6.9: Code generation option Window.

Table 6.2: Sample code generation for C# language.

Class1.cs

```
using System;
using System.Collections.Generic;

namespace untitledModel.Package1
{
	public abstract class Class1
	{
		private int cl1Attribute1=5;
		private static string cl1Attribute2;

		public int Cl1Attribute1
		{
			set {cl1Attribute1 = value; }
			get { return cl1Attribute1; }
		}

		public string Cl1Attribute2
		{
			set {cl1Attribute2 = value; }
			get { return cl1Attribute2; }
		}

		public string CL1Operation1(int  a = 3)
		{
		}

		public int CL1Operation2()
		{
		}
	}
}
```

Class2.cs

```
using System;
using System.Collections.Generic;

namespace untitledModel.Package1
{
	public sealed class Class2
		 : Class1 , Interface1
	{
		private int cl2Attribute1;

		public int Cl2Attribute1
		{
			set {cl2Attribute1 = value; }
			get { return cl2Attribute1; }
		}
		public Class2()
		{
		}
```

```
            ~Class2()
            {
            }

            public static string CL2Operation1(int  a , bool  b)
            {
            }

            public string Int1Operation1(int  a , string  b)
            {
            }
      }
}
```

Interface1.cs

```
using System;
using System.Collections.Generic;

namespace untitledModel.Package1
{
      public interface Interface1
      {
            string Int1Operation1(int  a , string  b);
      }
}
```

Table 6.3: Sample code generation for Visual Basic .NET language.

Class1.vb

```
Imports System
Imports System.Collections.Generic

Namespace untitledModel.Package1

      public MustInherit Class Class1

            private  cl1Attribute1 As Integer
            private static cl1Attribute2 As String

            public Function CL1Operation1(Optional ByVal a AS  Integer = 3) AS
String


            End Function

            public Function CL1Operation2() AS Integer

            End Function

      End Class
End Namespace
```

Class2.vb

```
Imports System
Imports System.Collections.Generic

Namespace untitledModel.Package1

	public NotInheritable Class Class2
		Inherits Class1
		Implements Interface1

		private  cl2Attribute1 As int


		public Sub New()

		End Sub

		public static Function CL2Operation1(a AS  Integer , b AS  Boolean) AS
		String

		End Function

		public Function Int1Operation1 (a AS  Integer , b AS  Boolean) AS
		String

		End Function

	End Class
End Namespace
```

Interface1.vb

```
Imports System
Imports System.Collections.Generic

Namespace untitledModel.Package1

	public Interface Interface1

		Function Int1Operation1(a AS  Integer , b AS  String) As String
	End Interface
End Namespace
```

Table 6.4: Sample code generation for Java language.

Class1.java

```
package Package1;

import java.util.*;

import java.io.*;

public abstract class Class1

{
```

```
	private int cl1Attribute1=5;
	private static String cl1Attribute2;

	public void Setcl1Attribute1(int  value )
	{
		cl1Attribute1 = value;
	}
	public int Getcl1Attribute1()
	{
		return cl1Attribute1;
	}
	public void Setcl1Attribute2(String  value )
	{
		cl1Attribute2 = value;
	}
	public String Getcl1Attribute2()
	{
		return cl1Attribute2;
	}
	public String CL1Operation1(int  a)
	{
	}
	public int CL1Operation2()
	{
	}
}
```

Class2.java

```
package Package1;
import java.util.*;
import java.io.*;

public final class Class2  extends Class1   implements Interface1
{
	private int cl2Attribute1;

	public void Setcl2Attribute1(int  value )
```

```
	{
		cl2Attribute1 = value;
	}
	public int Getcl2Attribute1()
	{
		return cl2Attribute1;
	}
	public Class2()
	{
	}
	protected void finalize ()
	{
	}
	public static String CL2Operation1(int  a , boolean  b)
	{
	}
	public String Int1Operation1(int  a , String  b)
	{
	}
}
```

Interface1.java

```
package Package1;
import java.util.*;
import java.io.*;

public interface Interface1
{
	String Int1Operation1(int  a , String  b);
}
```

Before code generation a new model is constructed by the model transformer. This model includes all the necessary language elements of the selected platform. Using this model the code generator will generate a code that can be easily usable by the developers. The pseudo code of code generation process is presented in Table 6.4.

Table 6.5: Code generator pseudo code.

```
Function GenerateCode(location) // code generation entry point function
    location= combine(location and SolutionName) // top level directory.
    if(directory exist(location))
      display warning message
    else
      create directory (location)
    call GenerateSourceCodeFiles(location);
end
Function GenerateSourceCodeFiles(location)
     rootDirectory= location
     if(language is Csharp)
        location = combine(location and ProjectName) //  project directory under top level directory
        if(directory exist(location))
           display warning message
        else
          create directory (location)
     For each(class type elements in CsharpModel.OwendElements) //class or interface not in any package
        create source code file object
        initialize class
        initialize file extension
        initialize file name
        add import list
       add source code file to project file list
       call Generatefile (location)
     end
     For each(packages in CsharpModel.OwendElements )
        If (package is root package)
           location = combine(location and root package name)
           For each(class type elements in root package)
              create source code file object
              initialize class
              initialize file extension
              initialize file name
              add import list
              add source code file to project file list
              call Generatefile (location)
           end
           For each(packages in root package) // child packages of root package
              Call PackageTraversal(child package, location)
```

```
            end
          end

      end
         Call GenerateProjectFiles(location)
      end // end if
end // end function
Function PackageTraversal(node package, location)
        location = combine(location and node package name)
        If(node is leaf)    // base case
         For each(class type elements in node package)
             create source code file object
              initialize class
              initialize file extension
              initialize file name
              add import list
              add source code file to project file list
              call Generatefile (location)
        else
            for each (child packages in node package)
               call PackageTraversal(child package, location)
            end
        end
end
Function GenerateProjectFiles(location)
      If (language is Csharp )
          fileName = project Name + ".csproj";
     else if (language is Visual basic)
          fileName = projectName + ".vbproj";
       projectfile = combine(location and node filename)
       template Directory = combine( application startup path + Project template)
       file reader = template directory
       while (reader is not end of file)
          line = read a line
          if (line contains ${Sourcefile})
              for each (file in project file list)
                  new line= replace line containing ${Sourcefile}) with file name combines with extension
                  write new line
              end
          end
       end
end
```

Chapter Seven: Usability Test

7.1. Overview

Usability tests identify areas where user struggle with a software and help to make recommendations for improvement. The aim is to better understand how real users interact with the software and to improve it based on the results. In this Chapter, we present the usability test process and result to verify how much the system is usable by the target users.

7.2. Target Audience

The system target users are software developers. A company, institutions, or even an individual can use this application to develop software. To conduct the usability test of the prototype we chose CNET Software Development Company, Sunshine Construction PLC and Addis Ababa University.

7.3. Evaluation Criteria

The concept of usability consists of three dimensions: effectiveness, efficiency and satisfaction in a specified context of use. ISO standard 9241 on usability recognizes each of them. We use this three attributes to evaluate the usability of the system.

7.4. Methodology

A six individual usability test sessions with representatives of the target users is prepared. The session takes approximately about 20-to-30 minutes and held at user location. We use questionnaire and observation to conduct the usability test result. The application is installed on the target user machine and each participant asked to complete key tasks of the system. During the course of their exploration of the system, we observe and take notes on participants' reactions, impressions, intentions, and experiences.

Questionnaire was designed in such a way that we can get quantitative results from it. The questions in the questionnaire are related to the criteria that were selected such as effectiveness, usefulness, user reaction, architectural and visual clarity, and functionality. As common users don't understand the criteria of evaluations, we designed questionnaire

according to Likert scale. We designed easy and understandable questions for all criteria so that common users can answer it without difficulty. The questionnaire contains 10 questions all of which were close-ended, and the respondents have to check only one option for each question. The designed questionnaire can be found in Annex 4.

7.5. Evaluation Result

After the usability testing, the designed questionnaires were distributed to the participants. The questionnaire was handed out to all the six participants in the form of hard copies.

After getting response from users through the questionnaire, we calculated each scale of questionnaire. The scale of these questionnaires is based on strongly agree**,** agree**,** neutral**,** disagree, strongly disagree. For each question the total and average values of the scale is calculated. Evaluation result of the survey is shown in table 7.1. From the evaluation analysis, the overall average is 4.35 out of 5 which shows that the prototype met its objective.

Table 7.1: Evaluation Result

No.	Questions	Score					Total	Avg
		1	2	3	4	5		
1.	It is simple to use the system.				2	4	28	4.67
2.	The user interface of the system is visually pleasing.				1	5	26	4.33
3.	The menu items are well organized and functions are easy to find.				1	5	21	3.5
4.	The system provides all of its functionalities.				3	3	27	4.5
5.	The system has a clear purpose.				2	4	28	4.67
6.	The system helps to save development time.					6	30	5
7.	The system is response is quick.					6	30	5
8.	The system gives error messages that clearly tells me how to fix the problem.				3	3	27	4.5
9.	Mistakes are easy to correct.			3	1	2	23	4
10	Whenever I make a mistake using the system, I could recover easily and quickly.		1	2	3		20	3.3
Overall average								**4.35**

Chapter Eight: Conclusion and Future Work

8.1. Conclusion

Model-Driven Architecture (MDA) is about separating the specifications of the system from the platform the system is running on. This is done by presenting the system with one or more UML models that may present different levels of abstraction. The change from one model to another is called model transformation.

A tool that implements the MDA concept would allow developers to produce models of the application and business logic and generate code for a target platform by means of transformations. Instead of writing platform-specific code in some high-level language, developers focus on developing models that are specific to the application domain but independent of the platform. In this way, MDA raises the level of abstraction in software development.

In this project, we presented a tool for Model –Driven Architecture. This tool extends the work of [6, 7] by adding additional features like: XMI support, code generation for multiple platforms, code generation for MVC design pattern, database script generation and reverse engineering (from code to model). The model has five major components: Model Constructor, XMI Parser, Class Parser, Model Transformer and Code Generator. A prototype of the model is developed and evaluated with users. The evaluation result shows that the performance of the prototype is very good.

8.2. Future Work

To improve the quality of the system, the following point should be considered.

- Due to time constraint the system only supports MVC design pattern. It is better to add additional design patterns.
- The system generates code for three languages C#, Java and Visual Basic. Additional language support will make this work strong.

- The database script generation considers only SQL server script. It is recommended that the system can include additional platforms like Oracle and MySQL.

References

[1] ZefHemel, Lennart C. L. Kats and EelcoVisser, *Code Generation by Model Transformation A Case Study in Transformation Modularity,* IEEE Computer Security, *2005.*

[2] Jorn Bettin, *Model-Driven Software Development Activities*, The Process View of an MDSD Project version 0.1, SoftMetaWare Ltd, May 2004.

[3] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami, *Model-Driven Development,* IEEE Computer Security, 2003.

[4] Krzysztof Czarnecki and Simon Helsen, *Classification of Model Transformation Approaches,* OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[5] Douglas C. Schmidt, *Model-Driven Engineering*, IEEE Computer Security, 2006.

[6] Henock Abye, *Design and Implementation of Code Generator for Model-driven Software Development*, MSc Project, Addis Ababa University, 2010.

[7] Melese Erku, *Enhanced Design and Implementation of Code Generator for Model-driven Software Development***,** MSc Project, Addis Ababa University, 2011.

[8] Paloma Cáceres, Esperanza Marcos, Belén Vela, *A MDA-Based Approach for Web Information System Development,* Kybele Research Group, Rey Juan Carlos University Madrid (Spain), 2002.

[9] O. Nikiforova, U. Sukovskis, V. Nikulsins, *Principles of Model Driven Architecture for the task of study program development,* Faculty of Computer Science and Information Technology, Riga Technical University, LV-1658 Riga, Latvia, 2008.

[10] John D. Poole, *Model-Driven Architecture: Vision, Standards and Emerging Technologies, Hyperion Solutions Corporation,* Workshop on Metamodeling and Adaptive Object Models, April 2001.

[11] OMG, *OMG's Unified Modeling Language (UML) Celebrates 15th Anniversary, URL: www.**omg**.org/news/releases/pr2012/08-01-12-a.htm*, August 2012.

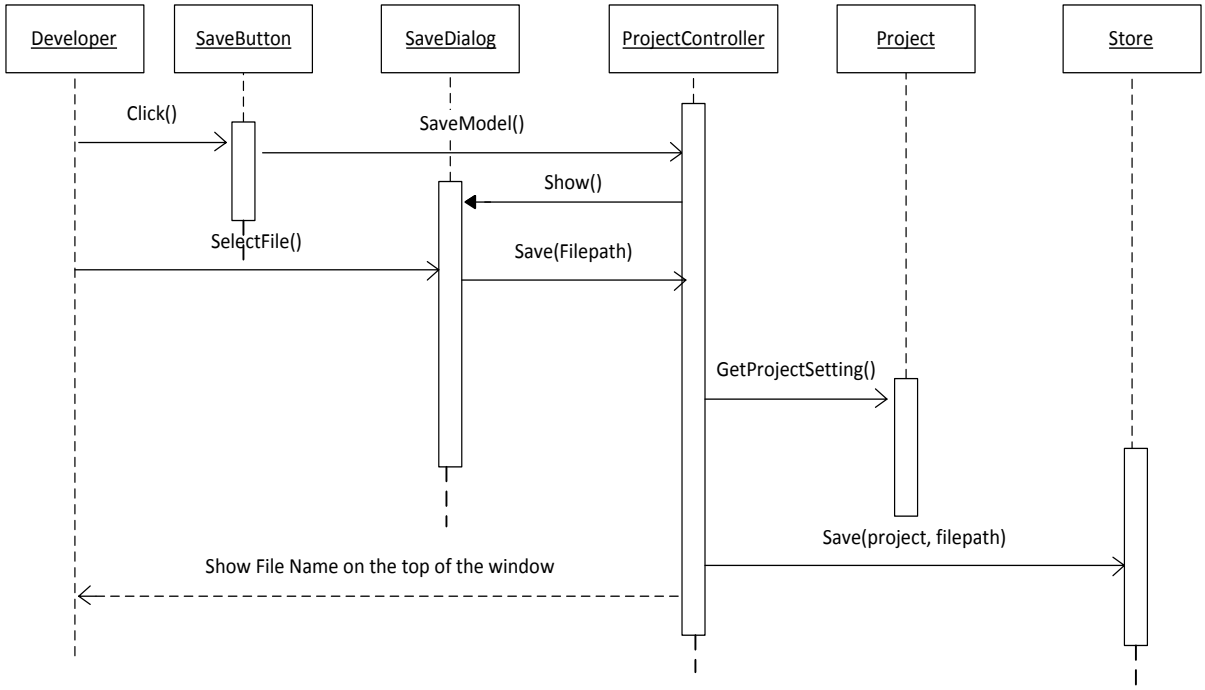
[12] OMG, *OMG Unified Modeling Language Specification*, OMG-Unified Modeling Language, V1.4, September 2001.

[13] Igor Sacevski, Jadranka Veseli, *Introduction to Model Driven Architecture (MDA)*, Seminar Paper, June 2007.

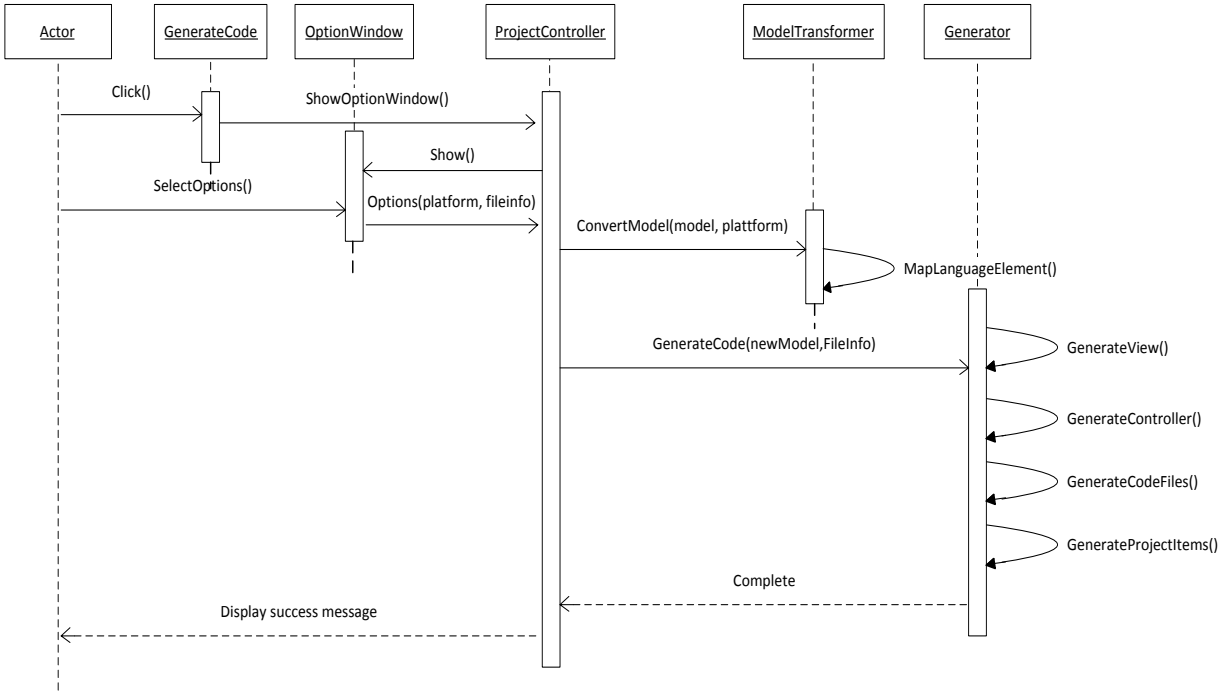
[14] Dimitrios S. Kolovos, *Extensible Code Generation Framework An agile code-oriented model driven generative approach,* MSc Thesis, Department of Computer Science, The University of York, 2004.

[15] Richard Soley, The OMG staff strategy Group, Model Driven Architecture, Object Management Group White Paper Draft 3.2, 2000

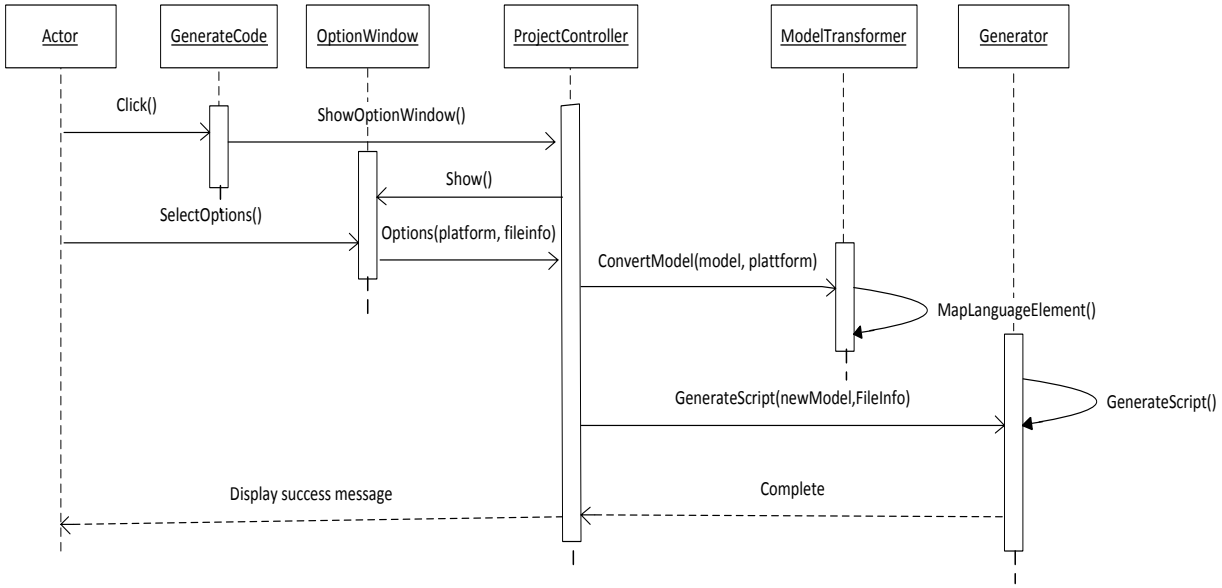
Annex 1: Sequence diagram for save model use case



Annex 2: Sequence diagram for generate code use case



Annex 3: Sequence diagram for generate script use case



Annex 4: Questionnaire

Instruction

For each questions, please use (✓) a number to indicate the level to which you agree with each statement on a scale of 1 to 5. (Strongly agree=5, agree=4**,** neutral=3**,** disagree=2, strongly disagree=1).

No.	**Questions**	**Score**				
		1	**2**	**3**	**4**	**5**
1.	It is simple to use the system.					
2.	The user interface of the system is visually pleasing.					
3.	The menu items are well organized and functions are easy to find.					
4.	The system provides all of its functionalities.					
5.	The system has a clear purpose.					
6.	The system helps to save development time.					
7.	The system is response is quick.					
8.	The system gives error messages that clearly tells me how to fix the problem.					
9.	Mistakes are easy to correct.					
10	Whenever I make a mistake using the system, I could recover easily and quickly.					

Declaration

I certify that this work contains no material which has been accepted for the award of other degree in any university or institute.

Furthermore, I took reasonable care to ensure that the work is original and to the best of my knowledge, does not breach copyright law, and has not been taken from other sources except where such work has been cited and acknowledged within the text.

This work was done under the guidance of my advisor Dr. Dida Midekso, at Addis Ababa University.

Declared by:

Name: Tedi Melaku

Signature: _____

Date: _____

Confirmed by advisor:

Name: Dr. Dida Midekso

Signature: _____

Date: _____

Place and date of submission: Addis Ababa, June 2014