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

By

Henok Abye

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, 2010

ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES
FACULTY OF INFORMATICS
DEPARTMENT OF COMPUTER SCIENCE

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

By

Henok Abye

Name and signature of the examining board:

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

Acknowledgements

I would like to express my gratitude to my advisor, Dr. Dida Midekso, for his support and encouragement through the course of this project. My efforts would not have met with success without his helpful input, insightful comments and most of all his patience.

I would also like to thank my family, who have stood by me every step of the way and never wavered in their support of my every ambition.

Finally, I would like to mention my colleagues, Beniyam Seyoum, Mulugeta Yared and Saededin Mussa who have stayed by my side in my most trying times, and my friends Addisu and Wolelaw with whom I have shared the experiences of the last two years.

Table of Contents

List of Tables

List of Figures

Acronyms

CIM ……Computational-Independent Model

IDE …….Integrated Development Environment

MDA ……..Model Driven Architecture

MOF ……...Meta Object Facility

MVC ……..Model-View-Controller

OCL ……...Object Constraint Language

OMG ……..Object Management Group

PIM ……Platform-Independent Model

PSM ……...Platform-Specific Model

SDLC …… Software Development Life Cycle

UI ……...User Interface

UML ……..Unified Modeling Language

Abstract

The field of software development has been in constant improvement since its inception. Several important milestones such as the development of high level languages, provision of better methodologies, improvements in IDEs and the advent of the object oriented approach have been part of a movement toward higher quality and better productivity in software development. The use of modeling to analyze requirements and realize software designs has been one of the significant and impactful additions to a software developer"s arsenal of tools.

UML is a modeling standard that has gained widespread use in software development. Models provide the ability to separate the core functionality of a software system from the implementation details which are platform dependent. The task of adding the implementation details and transforming a model into useable software has traditionally been a manual task. As such, the effort of maintaining both model and code has been a drain on the productivity of developers and has reduced the effectiveness of models.

Model driven development is an approach of using models to automatically generate the source code implementation of a software system. The MDA approach has been gaining ground as the model driven approach of choice because of its use of UML and standards based nature. The MDA solution is to generate all code of a system from models and concentrate on maintaining models only. However, due to shortfalls of code generation tools and the UML language itself, true MDA has not been realized. Instead, partial code generation is supported. While this is helpful, the separation between models and code has remained.

This project was proposed in the interest of solving this problem. The approach taken in this work is to allow re-integration of generated code into the model and give the developer the opportunity to work on either side of the transformation interchangeably. A prototype has been developed to this end. The prototype has shown the viability of this approach within the context of three major UML diagrams and one output language.

Keywords: Software development, Model driven, Code generation

1. Introduction

1.1. General Background

Software development is a field of human endeavor aimed at solving problems through the application of computing technology to particular problems and scenarios. The concepts behind the term „software development" can be better understood by treating each word separately. Software, in its most basic form, is a set of instructions which tell a computing device what to do [7]. The essence of developing software is the production of such a set of instructions that solve a certain problem. Of course, such a simplistic view conceals the vast complexity of modern software systems as well as the effort and resources that go into the development process.

As computing devices have become more capable, the size and complexity of software has increased to the point where it is impossible for a human mind to comprehend a system without abstraction and tools. Software projects have taken on the large scale and multi-professional nature previously observed in engineering. This led to the birth of software engineering as a discipline, with many techniques adopted from other engineering fields [11, 13, 14], including the use of well defined development methodologies with clearly separated phases. According to many of the methodologies, software development comprises a set of activities or phases known as the software development life cycle (SDLC), starting with requirement elicitation and going through analysis, design, implementation, deployment and finally maintenance [15].

The different phases of the SDLC each have a tangible output, known as a development artifact. Some of the artifacts consist of documentation, such as user requirements or support manuals. The final product of the development process is the executable artifact. There are two development artifacts which are most useful during the development process, the modeling artifacts and the source code [13]. Models are usually graphical and closer to human understanding, while code is highly structured and closer to mechanical operation. The role of models and source code in is rooted in the history of software development. The first computers were „programmed" by manipulating physical switches. As the machines improved and the complexity of tasks increased, intermediaries between the mental visualization of a system and its physical implementation were introduced [5]. Object code, assembly language, high level languages, analysis and design models are all intermediary representations of a software system linked by a series of transformations. Higher level representations are more abstract and each transformation yields a more detailed representation until the final executable solution.

The distinction between the lower level artifacts has been steadily eroded by automation. Transformation from high level source code to object code and finally to executable code is invisible to the developer since a modern compiler or IDE completely automates this process. Software developers have come to think of high level source code as the final specification of a system since it is the last stage for direct human involvement and manipulation. There is also no discernible gap in the expressive power of source code as compared to executable code. Every detail required in the final solution can be provided at the source code level, which allows the transformation to lower level artifacts to be easily automated.

Directly above the level of source code in the development artifact hierarchy are the analysis and design models. The use of models to represent software is a practice that existed even in the early days of software development. However, early models were highly informal with no standardization [1, 8, 9]. Due to the highly manual nature of software development at the time, models were intended as instruments of immediate communication rather than permanent specification. With the adoption of well defined methodologies, the importance of models increased and the nature of models became more formal. The flow charts and data flow diagrams of the structured approach are good examples of this phenomenon. The current generation of modeling is UML, which provides a standard for models in the object oriented approach.

The essence of modeling is to provide a good representation of certain aspects of a software system to enhance understanding of that system. Models tend to ignore the details of a system in order to focus on a larger picture of the system as a whole. Models also routinely exclude complex interactions in order to clearly show how certain portions of a system function. These traits are necessary to allow a level of abstraction that is conducive to human understanding [12]. The avoidance of details also serves another purpose, allowing models to be platform independent. On the other hand, this nature of models is not helpful for completely automating the transition from model to lower level artifact. There is a wide gap between the levels of detail expressed in a model and that of source code. Models allow greater freedom of expression due to their graphical syntax, but code has more precision of meaning and represents software in exact terms. Due to these facts, the application of automation to this phase of transformation has not been as quick or as successful as that of compilation.

The need for automation of the model – code transition emanates from the continuous progress of the software development field. Software solutions are being applied to many different areas and more complex tasks. Software projects are consuming more and more manpower and

resources, and this has put the focus on developer productivity. Previous efforts to increase developer productivity have focused on providing better methodologies and tools as well as automation in the lower levels of development artifacts. The next avenue for improvement is the automation of transitions between higher level artifacts, namely models and code.

The current direction taken by the software industry in automating the model-code transition is model driven development. Model driven development puts models at the center of the software development process [10]. Models are not only used to understand and analyze the system under development, but also as a final specification of how the system functions. In model driven development, the role of code as the last point of modification is taken over by the higher level models. Any changes are always made directly in the models and the code then generated automatically from the models. In the absence of modifications or additions to the generated code, this automatically avoids any problems of model-code coherence. That is, the code is guaranteed to be equivalent to the model representation of the software system.

The major proponents of this approach are the OMG, who developed the UML modeling standard and MDA. MDA is a specific form of Model driven development based on UML modeling. The problem of this approach is the lack of detail in the models. Modeling languages originally built for communication and understanding do not lend themselves to easy automated conversion into code. Ambiguities and abstractions that can easily be dealt with by a human mind become obstacles for a conversion algorithm. Therefore, current implementations of MDA have difficulties with complete generation of code. While specific areas such as embedded device development have seen success in complete code generation, more general code generators are only capable of generating part of the code. In reality, manual manipulation of code has been decreased but not removed. While the benefits of any automation are not negligible, a core problem of model – code coherence remains.

UML is an evolving standard, and improvements to the modeling language will probably lead to better expression of detail and therefore better code generation in the future. However, current needs require a solution that can mitigate the problem within the existing context. The transition of developers from code based development to model based development will take time. Until models have the capability to completely take over from code, it will be necessary to have modifiability in code as well as model. As long as this situation persists, so will the coherence problem. Therefore, it is important to solve the problem in the current context.

1.2. Statement of the Problem

The advent of object oriented development methodologies has led to better programming practices and improved developer productivity. The development of the UML standard was instrumental for the acceptance of modeling as an integral part of object oriented design. However, the translation of a design model into the source code of an implementation language is still mainly done manually. If the design model is specified with sufficient detail, this task of translation becomes somewhat mechanical and tedious syntax transformation.

In addition to consuming valuable development time and resources, the manual translation task may result in deviations from the design as well as inconsistencies between the design and the code [13]. Any modifications to the design require a fresh translation effort; therefore the adaptability of the system is reduced. Once the translation is complete, the attention of the developer is divided between the two development artifacts which exist independently. Since the final product is closer to the code than the design, developers tend to give emphasis to code rather than design. New modifications may be made only in the code, turning the design into an outdated development relic.

The MDA approach is intended to alleviate these problems. It provides strong emphasis on the design models rather than code and allows the developer to work on the design without the burden of manually converting every aspect of it to code. Since the MDA is only a standard by OMG, tools are required to be implemented based on the approach.

Given the length of time since the original availability of the MDA specifications, there are a number of existing tools based on it. Some of these tools, such as MagicDraw and ArcStyler, are proprietary, off the shelf solutions. There are also open source tools ranging from the basic Acceleo to the more advanced AndroMDA. Most of these tools provide powerful code generation capabilities across a wide range of languages, and some (MagicDraw, NClass, MetaMill) even support reverse engineering. However, all of the tools focus on one time transformation from model to code or vice versa. There is no capability for modifications to the final source code to be integrated or appended to the model based specification.

In the ideal case of MDA, where the models provide a detailed description for all aspects of the final program, such a capability would not be necessary (and may not even be desired). In practical cases, on the other hand, it is much more efficient to provide part of the source code

outside the realm of the model. For instance, many integrated development environments (IDEs) have excellent user interface (UI) designers with their own UI code generation. Developers may also have an interest to reuse a certain portion of existing code, or may require greater control over the execution sequence and details of the code. As such, it is important to have a capability of integrating the non-generated code to gain the advantages of MDA within the current practical context of software development.

1.3. Objective

1.3.1. General Objective

The general objective of the project is to design and implement a UML based code generator tool.

1.3.2. Specific Objectives

The specific objectives of the project are the following:

- Conduct a review of the UML and MDA specification.
- Develop a UML design environment.
- Develop a UML based code generator algorithm.

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 (code to models). The scope of this project must be limited given the time and resource constraints. Therefore, the project includes only the model construction and code generation aspects of MDA. These two functionalities form the essential core of an MDA tool. The model construction is again limited to the core diagrams that are essential to the design process and code generation. The following UML diagrams are expected to be supported by the tool:

- Use-case diagram
- Class diagram
- Sequence diagram

The code generation is limited to output code in a single language, although the approach should be extendable to other languages without major modification. The application areas of the tool may also be limited since it may not be practical to support applications which require a lot of low level details. Database modeling (e.g. using ER models) is beyond the scope of the project.

1.5. Document Organization

This project report contains six parts including this first introductory portion. The next part presents the literature review. In the third and fourth portions, the paper presents the analysis and design according to the object oriented methodology, after which details of the implementation are discussed. Finally the conclusion and recommendations are given.

2. Literature Review

2.1. Overview

The automated conversion of modeling artifacts into source code is the focus of this work. There is an ongoing effort in the software development community to achieve this goal under the umbrella of the model-driven development movement. Spearheading the movement is the OMG, which is consolidating some of its existing standards into a model-driven development approach known as MDA. This section deals with some of the problems that have led to MDA, the UML standard on which MDA is based and finally some core concepts of the MDA approach itself.

2.2. Significance of Automation

The advancement of software development has been measured mainly in terms of the productivity of developers. The time that is required for a development project is a crucial element of this measure. Time is indicative of the effort and resources required to complete a project. If a significant reduction in development time can be achieved, it can be translated into improvements of quality or reduction in costs. In the past, several alternatives have been suggested to improve developer productivity, ranging from improved tools to specific methodologies. Many claimed exaggerated improvements and presented one particular solution to be the end of all troubles for developers. This type of solution is known as a silver bullet, and was debunked by Fred Brooks in his article "No Silver Bullet - Essence and Accidents of Software Engineering". He correctly hypothesized that no single solution could provide an order of magnitude improvement in developer productivity. However, he also accepted that a series of step by step improvements could provide real benefits of productivity [6].

Automation of the model – code transition is an important step in the direction of better software quality and quicker development. Manually implementing a software system based on a design provided in modeling diagrams is common in modern software development methodologies. The use of models has helped to separate major decisions about the architecture, structure and behavior of a software system from implementation specific details. This has in turn made the task of implementation simpler and more straight-forward. However, a significant amount of development time is spent dealing with this transition and therefore this transition is a bottleneck of developer productivity. Moreover, the technological differences in implementation platforms force a repetition and duplication of this effort whenever there is a need to change the underlying platform, even if all the decisions reflected in the model are still relevant and the software itself remains essentially the same.

2.3. Problem of model-code coherence

Manual conversion of design artifacts into code is a time consuming operation in itself. However, a more serious problem is the separation between code and model that is created by manual conversion. Once the conversion has taken place, maintaining coherence between the two artifacts takes up a lot of effort and resources away from the actual development of the software solution [2]. One option is to make changes to the design and then attempt to change only the affected parts of the code. Any change in design requires developers to look at the implemented code, find the implications of the change to other related parts of the code, make the changes and then review the design to see what the changes to code mean for other parts of the design. Another option is to make changes to the design and entirely re-implement the code according to the new design. Both options are costly and make the necessary and natural changes of requirement into serious problems for developers. Accordingly, developers can be tempted to relegate the role of models to only the initial analysis and design, or even reject the use of models entirely. This has an impact on the quality of the final product.

2.4. The OMG Solution: MDA and UML

The model-driven software development effort of the OMG is based on several standards and specifications. However, the basic MDA vision and UML as a language are the most prominent aspects of this effort and also the most relevant when it comes to the development of tools. UML represents the most widely used and mature modeling implement in the current era of software development. MDA is building on the potential of UML and taking modeling to the next level of importance within the development process. Accordingly, the following sub-sections briefly discuss UML and its core diagrams, as well as MDA and its different model types.

2.4.1. UML

The Unified Modeling Language (UML) is a family of graphical notations used for describing and designing software systems, usually using the object oriented method [1]. UML defines a standard notation for constructing models of the static and behavioral aspects of a software design. UML is an open standard which is maintained by a consortium of companies known as the Object Management Group (OMG). It is called “unified” because it was developed from three different methods: the Booch Method by Grady Booch, the Object Modeling Technique coauthored by James Rumbaugh, and Objectory by Ivar Jacobson. The original version of UML became a standard in 1997. The most recent major revision of UML was version 2.0 which was accepted in 2005, and the current version is UML version 2.2. UML is distributed as a set of four specification documents; The Diagram Interchange Specification, UML Infrastructure, UML Superstructure and the Object Constraint Language (OCL) [4].

UML is based on the meta-object facility (MOF). MOF can be described as a meta-language used to define other languages, such as UML. MOF contains the basic constructs which form the basis of UML, such as class, object and relationship.

UML 2.0 divides diagrams into two categories: structural diagrams and behavioral diagrams. Class diagrams, component diagrams and object diagrams represent a static view of the system and are considered structural diagrams. Behavioral diagrams, on the other hand, show the system in action through changes in state or different stages of control flow, and include sequence diagrams, activity diagrams and state charts [4, 8]. Three diagrams form the core of UML modeling and these are the use case diagram, class diagram and sequence diagram.

Use Case Diagram

Use case diagrams provide a way to capture system functionality and requirements. The components of a use case diagram are the boundary, use cases, actors and communication. A use case represents a piece of functionality provided by a software system. A use case is depicted by an oval shape with a descriptive name inside it. Actors are initiators of use cases, and interact with the system in the duration of the use case. Users of the system are usually actors, as well as anything which interacts externally with the system. A communication is a line showing the relationship between actor and use case. Actors and use cases are separated by the boundary, representing the delimitation between the external and internal aspects of the system.

Class Diagram

Class diagrams capture the static relationships between parts of a software system. Classes and relationships are the components of a class diagram. Classes are the basis of the object oriented approach. They are the static representation of objects, which are units of encapsulated data and operations on that data. Classes are composed of attributes and operations. A class is depicted by a square shape with a number of sections (usually three: name, attribute list and operation list). Relationships among classes can be shown by a line. A generalization (inheritance) relationship joins a derived class with a parent class via a line with a closed arrowhead toward the parent. A binary association relationship shows the multiplicities on both sides. Finally an associated class can add data to the relationship between classes.

Sequence Diagram

Sequence diagrams, also known as interaction diagrams, are representative of a set of sequential messages between objects which realize a particular functionality during the operation of the system. The components of a sequence diagram are objects, lifelines, activations, messages and actors. An object is depicted by a square shape with the object‟s name and class inside, as well as a dotted line extending downward, known as a lifeline. An activation, depicted by a narrow square on top of the lifeline, shows periods when an object is processing, sending or receiving messages. Interacting objects are shown side by side, with activations connected by open arrowhead lines pointing to the receiver of the message.

2.4.2. MDA

Model driven architecture (MDA) is a design approach which offers a higher level of abstraction for development and separates the logical design of a system from implementation, allowing developers to concentrate on high level design while automating the more mechanical aspects of translating design models into code [3]. MDA is not a single standard on its own but rather an approach integrating a number of OMG standards. MDA is supported by the Unified Modeling Language (UML), the MOF, XML Metadata Interchange (XMI), and the Common Warehouse Metamodel (CWM). MDA was originally conceived to allow middleware platform independence in enterprise software development. The problem which faced large scale enterprise software was that developers had to make use of a number of different middleware platforms such as Web Services, XML, .NET, EJB and CORBA. Re-implementing an entire enterprise system due to a change in middleware platform was a resource consuming task. The MDA solution was to separate the business logic which would be the same on any middleware from the implementation details and constraints that are peculiar to a specific middleware platform. MDA has several layers of models at different levels of abstraction.

Computational-independent model (CIM)

The CIM is the highest-level MDA model and is essentially a requirement-only model. The CIM does not make use of UML, but a domain specific language (DSL) derived from the MOF could be used instead. It describes interactions between business processes and the responsibilities of each worker, be it human or otherwise. CIM‟s can be understood easily by people who are not familiar with software modeling. It also avoids specialized knowledge of procedures internal to a computer system.

Platform-independent model (PIM)

A PIM is a representation of the functionality to be implemented by a software system. UML or a modeling language derived from UML is used in describing this model. The PIM describes the processes and structure of the system, without reference to the delivery platforms. It avoids details pertaining to operating systems, programming languages, hardware, and networking. PIM can be understood by people who know about the system under development and have at least a basic comprehension of UML. The PIM is the usually the starting point for automated MDA transformations since it contains enough detail of the system in a structured and computation related format.

Platform-specific model (PSM)

The PSM is a transformed version of the PIM for a specific platform. One PIM can be the source for multiple PSMs, and the developer must define a different PSM for each target platform. The PSM deals with a particular operating system, middleware platform and programming language and incorporates details that apply only for that specific platform. During a change from one platform or language to another a previous PSM cannot be reused, but the generation of a new PSM does not require any change of the PIM.

Code model (Source code)

The code model is the final source code, usually expressed in a high-level programming language. In an ideal MDA environment, the code model should be automatically generated from a PSM and should not require any further modification by the developer. In the more realistic case, MDA tools are not so capable and developers need to understand the technology so that they can modify and debug the application. Developers need to load the source into IDEs and edit and compile as if the code was written manually. Because less code is actually handwritten, the development time may be reduced and the code is much closer to the design in the model.

The relevant MDA models for this project are the PIM, PSM and code models. The design environment deals with the creation and modification of a PIM. The code generation and integration algorithms deal with the transformation from PIM to code and vice versa by using an intermediary model which takes the role of PSM and is very similar to the code model.

3. Analysis

This section presents the functional and non-functional requirements of the system as well as the analysis models.

3.1. Functional Requirements

The functional requirements are:

- Provide an environment to design graphical UML diagrams
    - Creation and removal of diagrams
    - Addition and removal of elements of a diagram
    - Changing the properties of elements
- Store and retrieve the models designed by the user
- Transform details of the models into code
- Integrate changes made in code to the models

3.2. Non-Functional Requirements

The non-functional requirements are:

- 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.

3.3. Use-case model

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. The use cases of the system cover most, but not all, of the interactions with the user. Slight variations of interaction, such as the use of a menu item instead of a toolbar button, are not presented independently since there is no difference in their effect to the state of the system.

Use case descriptions

The use cases of the system are save model, load model, create model, add diagram, display diagram, remove diagram, add element, remove element, modify element property, generate code, integrate code and change settings.

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 or object. An element has properties such as 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 by 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 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	iii. The user enters an invalid file location iv. The system shows an error dialog [Resume Normal Flow at step ii.]

Table 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 and the system must be running
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	iii. The user enters an invalid file location iv. The system displays an error dialog [Resume Normal Flow at step ii.]

Table 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 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 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 least 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 the diagram elements in the diagram editor iv. 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 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 least 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 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 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, then 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 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 [Alternates A, B] 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 editor display
Post condition	The element property is changed and the diagram editor display is updated
Alternate A	i. The user selects a diagram element containing a class element and clicks on edit ii. The system displays the class element properties in a class editor dialog [Resume Normal Flow at step iii.]
Alternate B	i. The user selects a diagram element containing a message element and clicks on edit ii. The system displays the message element properties in a message editor dialog [Resume Normal Flow at step iii.]

Table 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 file location and clicks on the generate button [Alternate A] iv. The system uses the information contained in the model to generate source code v. The system writes the code to a file at the 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	iii. The user enters a invalid file name or location and clicks on the generate button iv. The system displays an error dialog [Resume Normal Flow at step ii.]

Table 11: Description of the Integrate Code use case

Use case name	Integrate code
Description	Reads code from a file 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 file location iii. The user selects (enters) a file location and clicks on the integrate button [Alternate A] iv. The system reads the code from the specified file 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 modifications in the source code files are applied on the model
Alternate A	iii. The user enters a invalid file name or location and clicks on the generate button iv. The system displays an error dialog [Resume Normal Flow at step ii.]

Table 12: Description of the Change Settings use case

Use case name	Change Settings
Description	Changes settings 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 settings button ii. The system displays a list of settings in the settings dialog iii. The user enters changes to the desired settings iv. The system updates the model with the changed settings v. The system updates the display with visible changes (e.g. model name)
Post condition	The settings are changed

Use case diagram

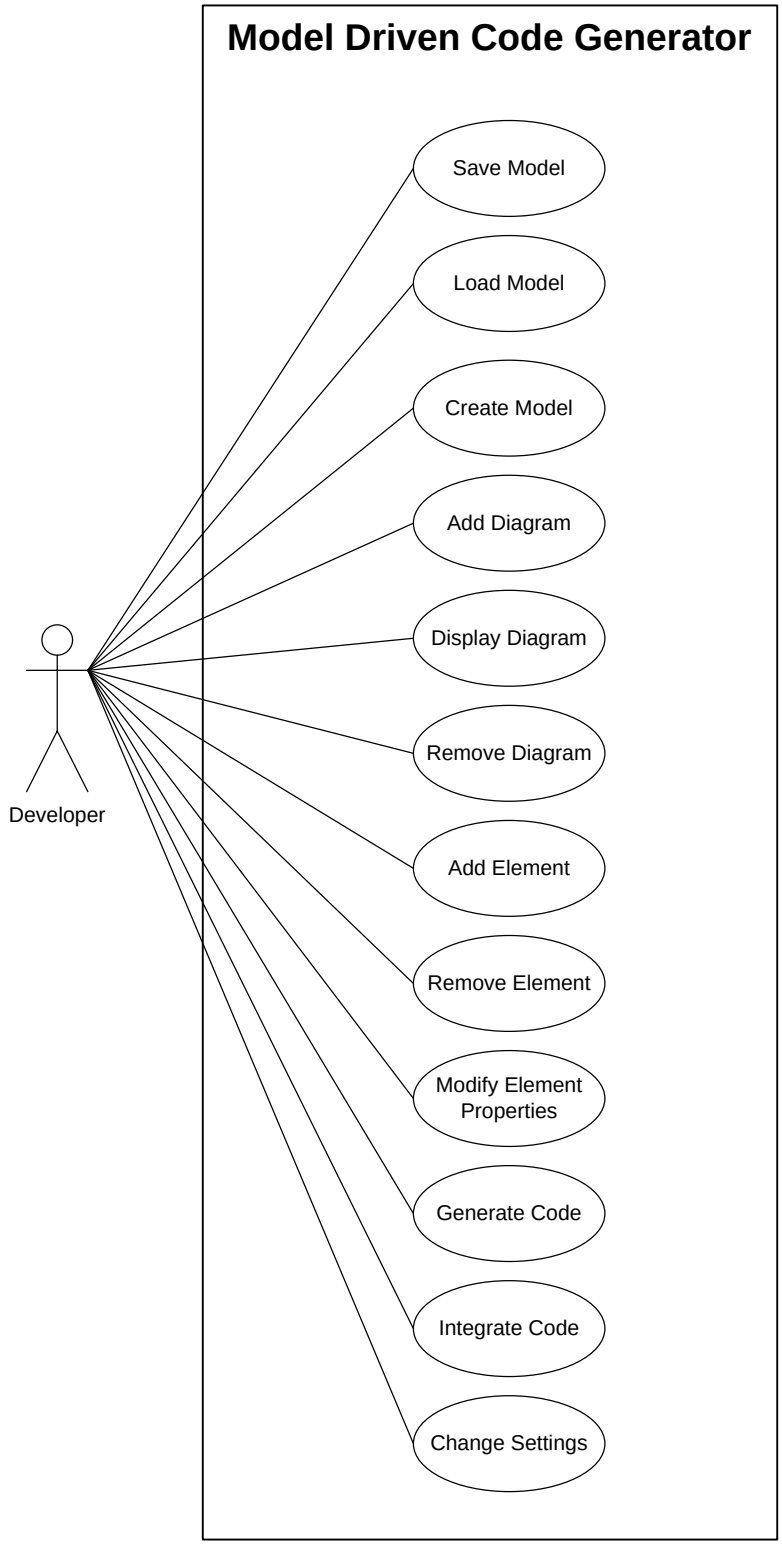

Figure 1: Use case diagram of the system

3.4. Structural model (Class diagram)

The following diagrams depict the major classes of the system. The three diagrams are presented separately for readability with highly related classes shown in the same diagram.

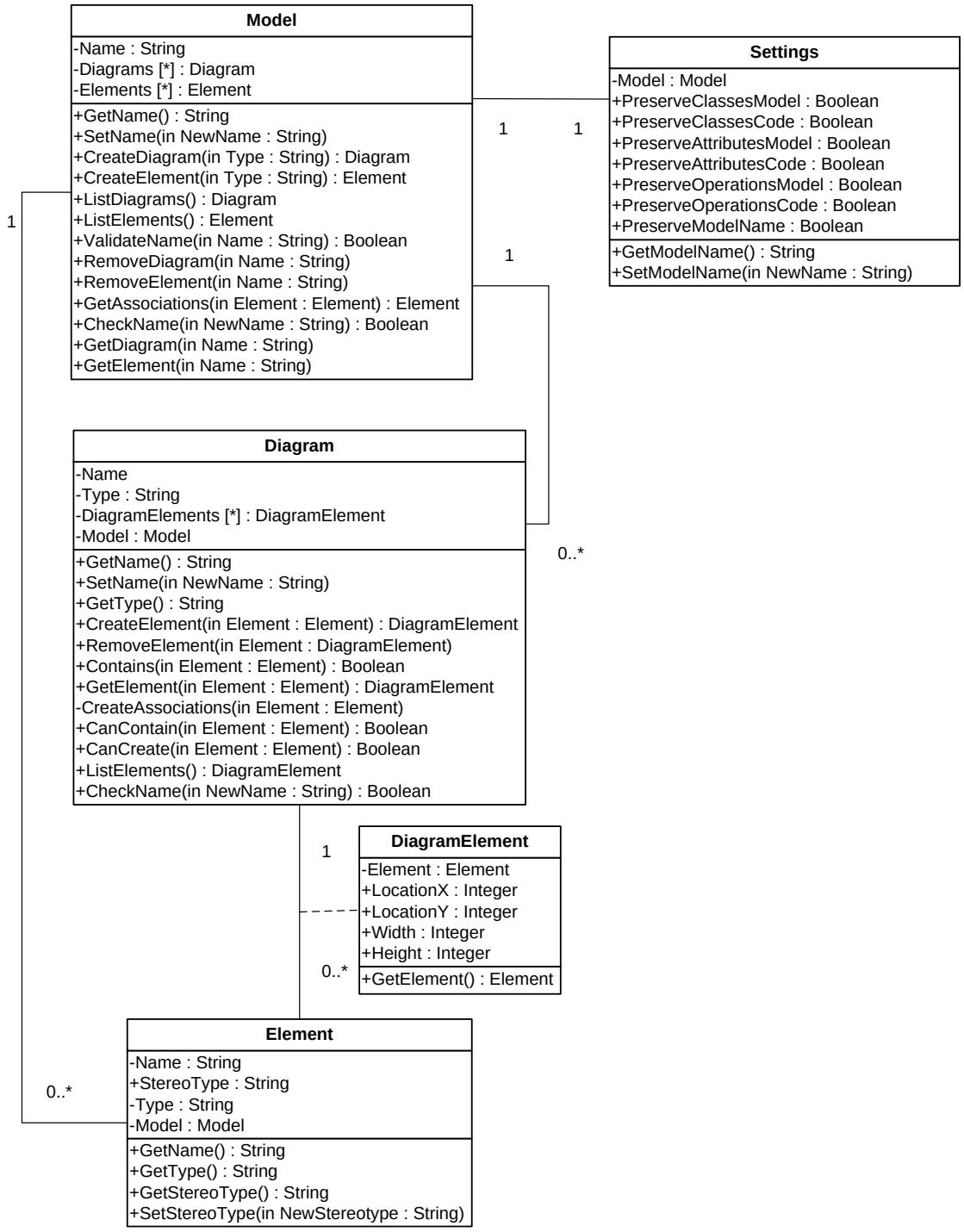

Figure 2: Class Diagram (part 1)

Figure 2 shows the basic structure of a *Model*, with a *Model* containing its *Settings*, as well as a set each of *Diagrams* and *Elements*. A *Diagram* then contains a *DiagramElement*, which contains an *Element*. The indirect inclusion of an *Element* in a *Diagram* allows elements to be shared by multiple diagrams while preserving the location and relationships it has in each diagram separately.

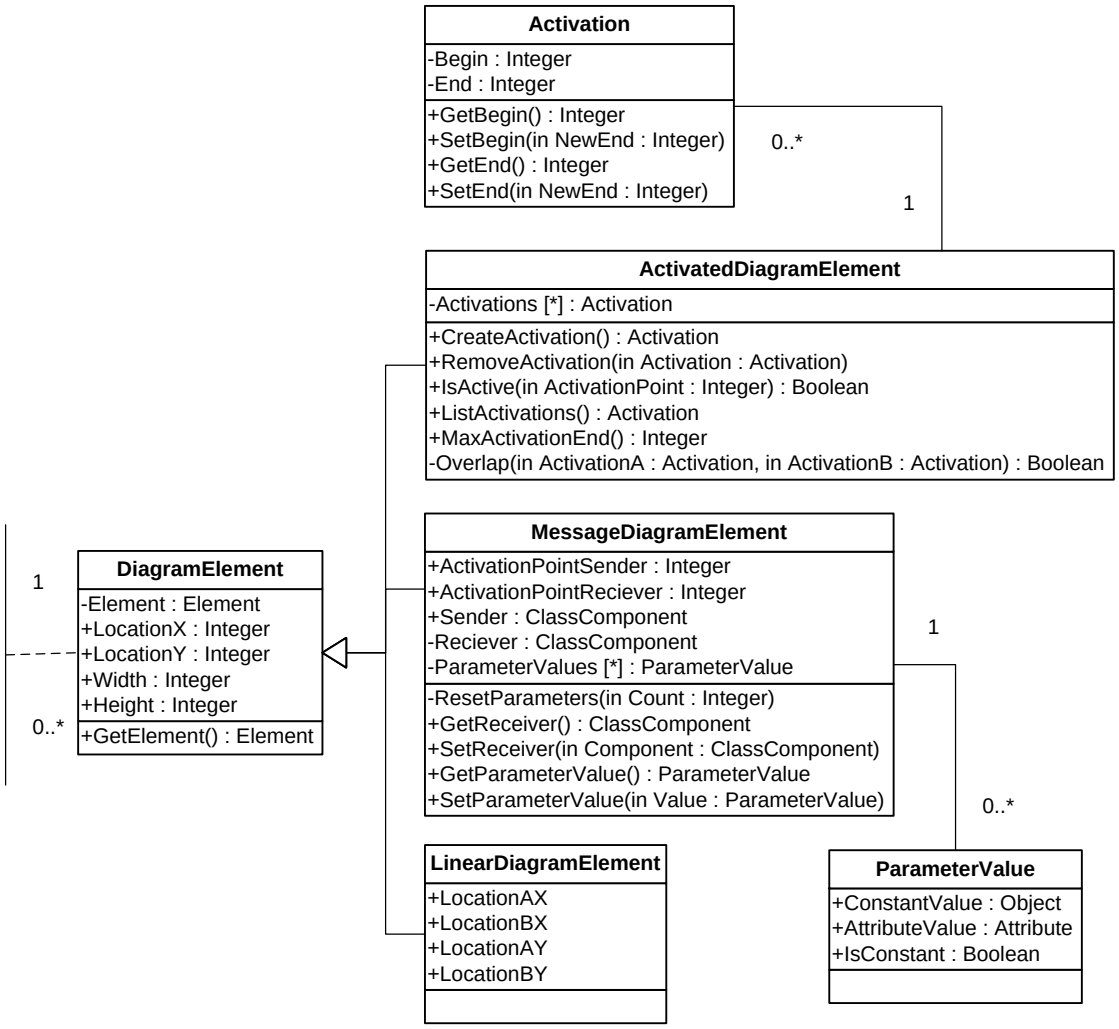

Figure 3: Class Diagram (part 2)

Figure 3 shows the different subtypes of *DiagramElement* and their helper classes. Each type of *DiagramElement* is designed to include a different type of element.

Figure 4 shows the different subtypes of *Element* and their helper classes. The *ClassElement* is the most complex, containing a set each of *attributes* and *operations*, which both inherit from *ClassComponent*. An *Operation* contains a set of *Parameters*. *ObjectElement*, *ClassComponent* and *Parameter* all contain an *ObjectType*, which can possibly contain a *ClassElement*.

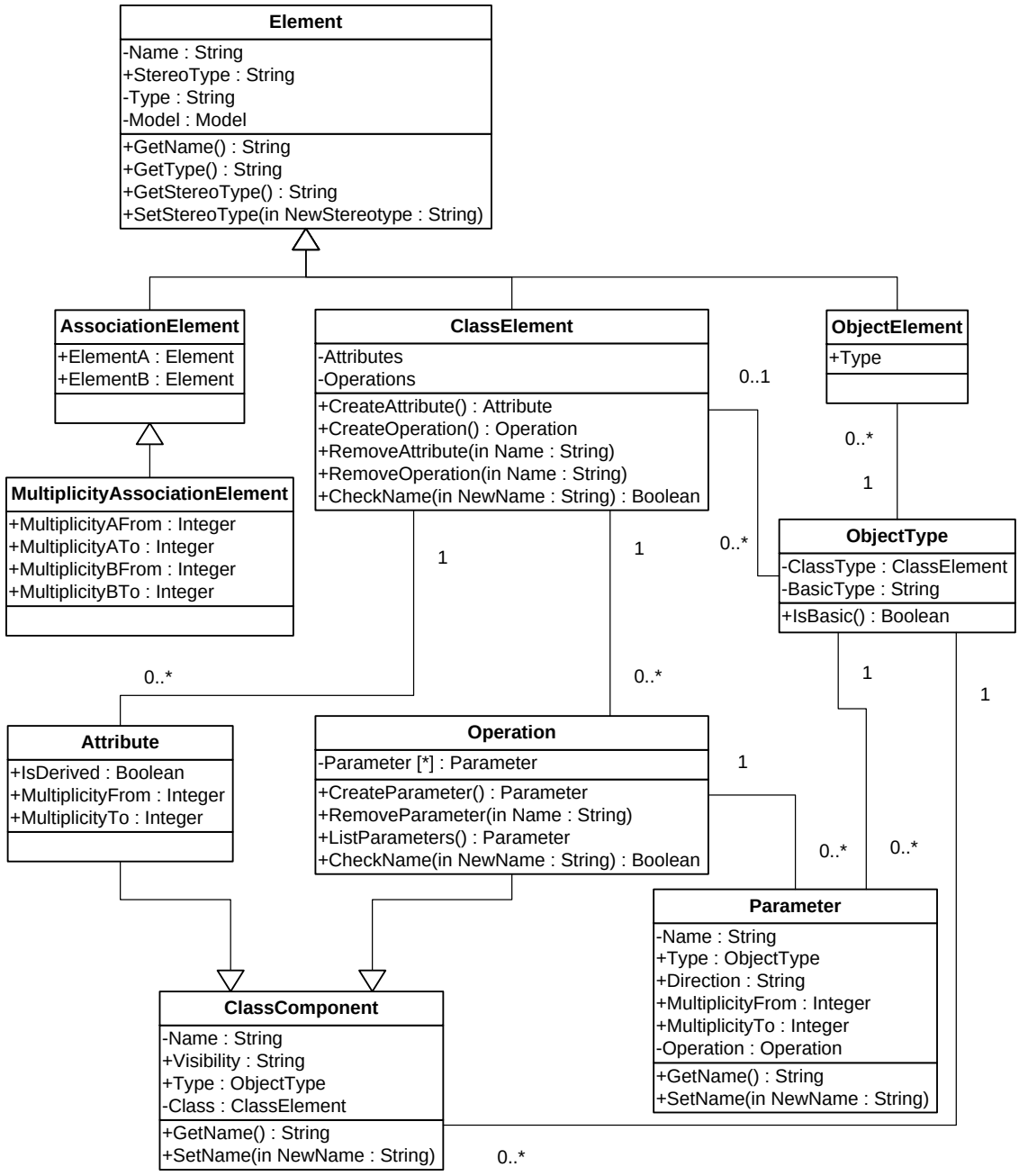

Figure 4: Class Diagram (part 3)

Class Descriptions

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.

Settings – is a helper class for model which stores all the settings used during code generation and integration operations on the model. It also allows modification of the model name.

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

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 an Element is set at creation and never changes. Type can be: *Actor, Use Case, Communication, Generalization, Class, Object, Message, Call,* or *Return.*

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

MultiplicityAssociationElement – is a subtype of AssociationElement which adds multiplicity information about each side of the relationship. Multiplicity is represented by two integers (negative integer indicates the many value – „*").

ObjectElement – is a subtype of Element which represents an object which can participate in a sequence diagram. It adds the type attribute which indirectly stores the ClassElement to which the object belongs (through the ObjectType class).

ClassElement – is a subtype of Element which represents a class. A class 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.

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

Operation - is 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 Parameter, 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*.

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

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

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

MessageDiagramElement – is 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 – is 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.

Additional Classes

In addition to the above classes, the system also has three classes dealing with storage and code generation. The Serializer class is responsible for converting the data contained in a model object to and from a storage format (dealt with in the persistent data management section of the next chapter). The Generator and Integrator classes transform model data to and from code respectively, and also handle saving and loading of text files. Another set of classes are responsible for the visualization and display of diagram elements as well as user manipulation of the elements in drag-and-drop manner, the most important of which are the Canvas and Visual classes. Canvas represents the entire drawing area while Visual represents each individual diagram element. The system also has control classes such as DiagramEditor and ModelExplorer as well as specialized interface dialogs for editing elements and settings.

3.5. Behavioral model (Sequence diagrams)

The sequence diagrams present the series of messages among objects during the operation of the system. Each diagram shows the interactions of objects which were described by a single use case in the previous section.

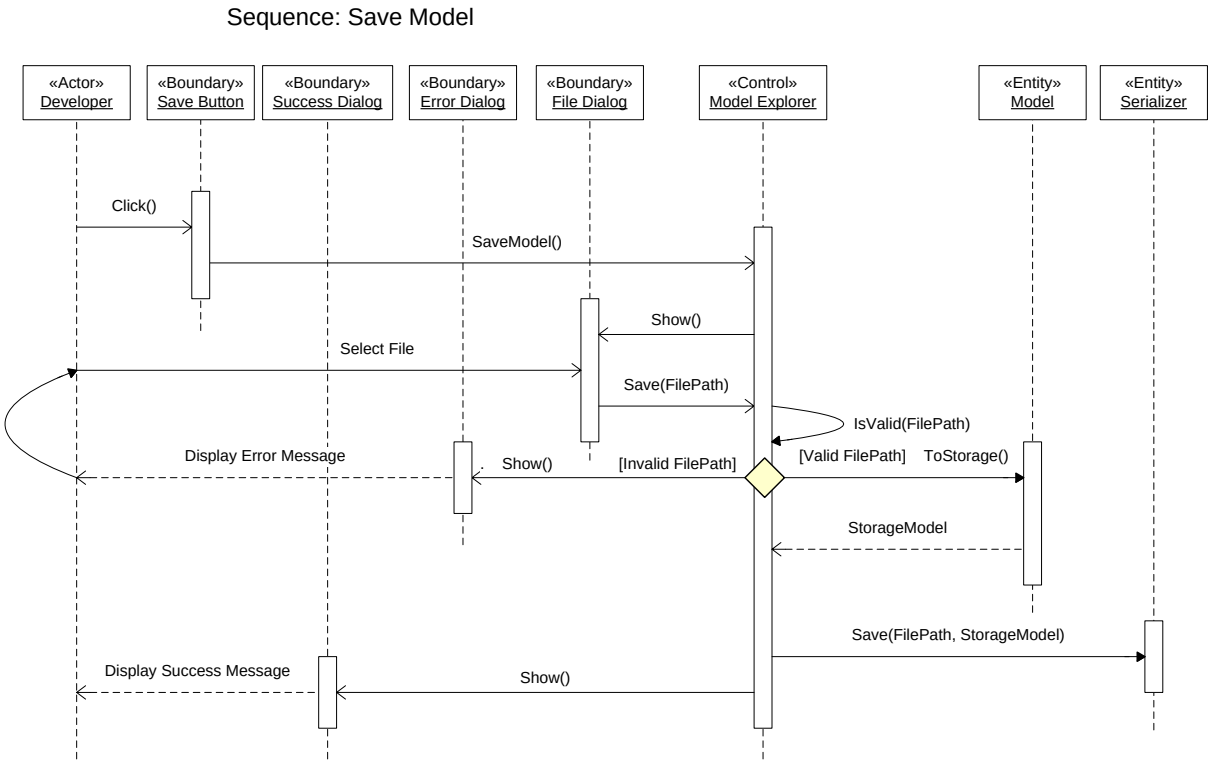

Figure 5: Sequence Diagram for Save Model

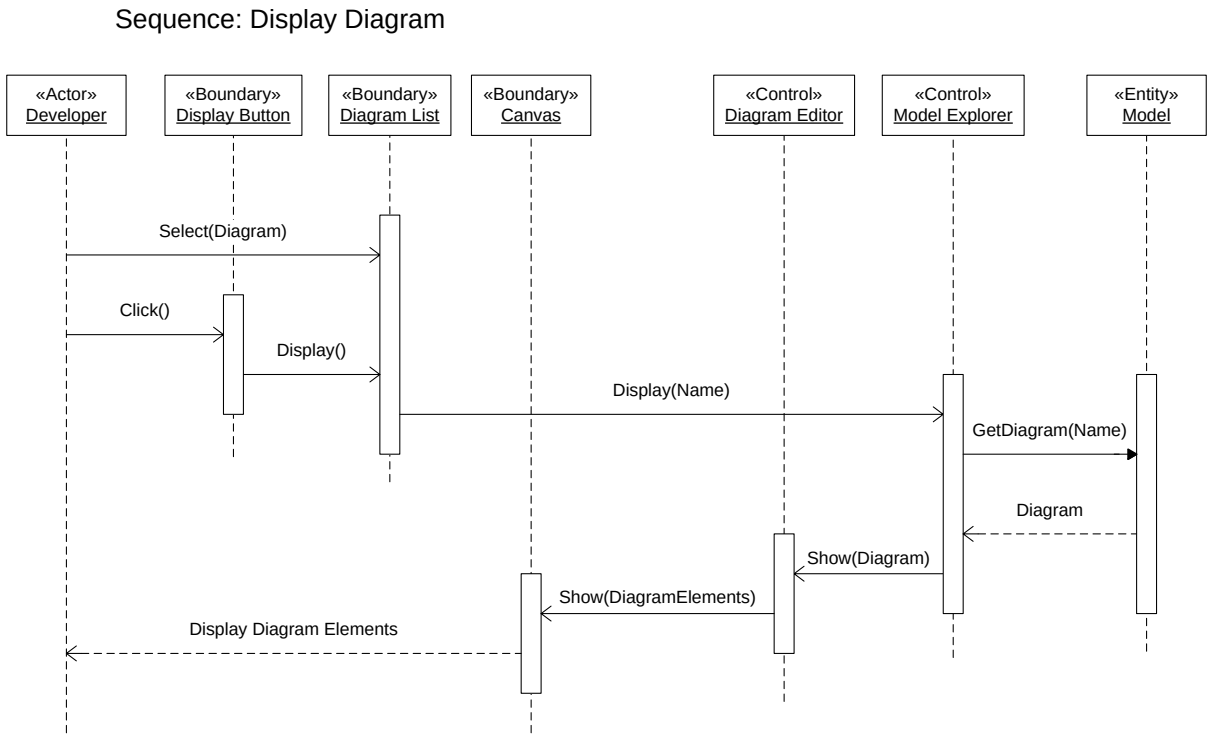

Figure 6: Sequence Diagram for Display Diagram

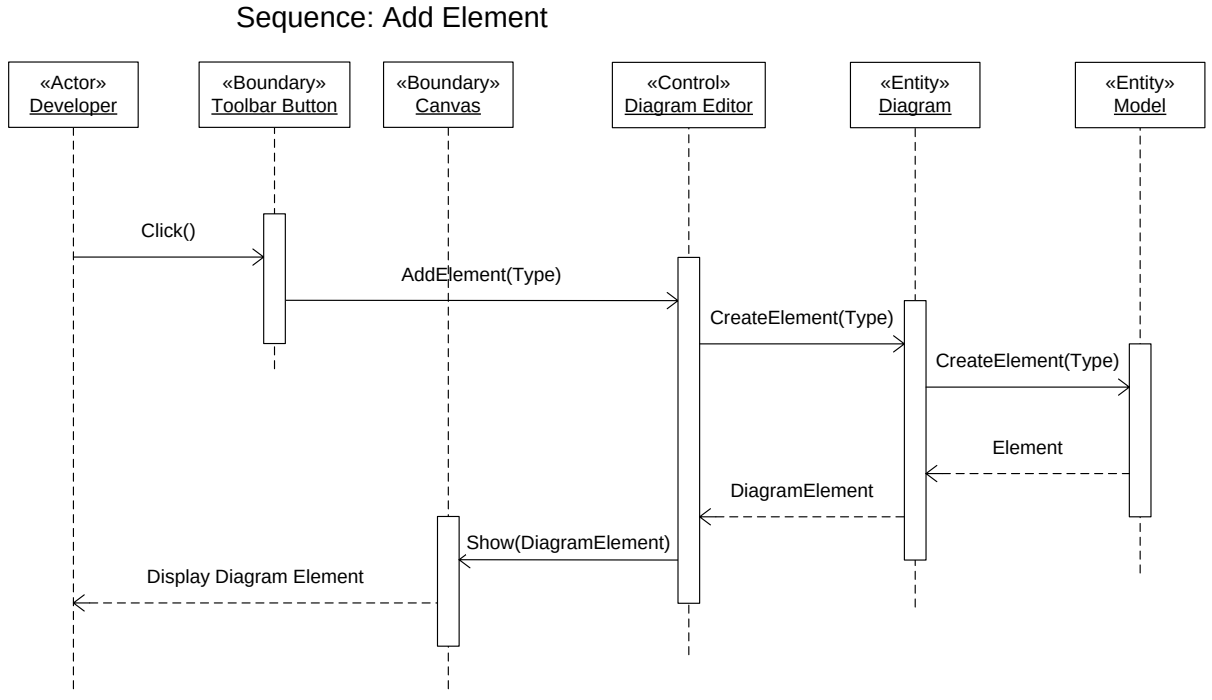

Figure 7: Sequence Diagram for Add Element

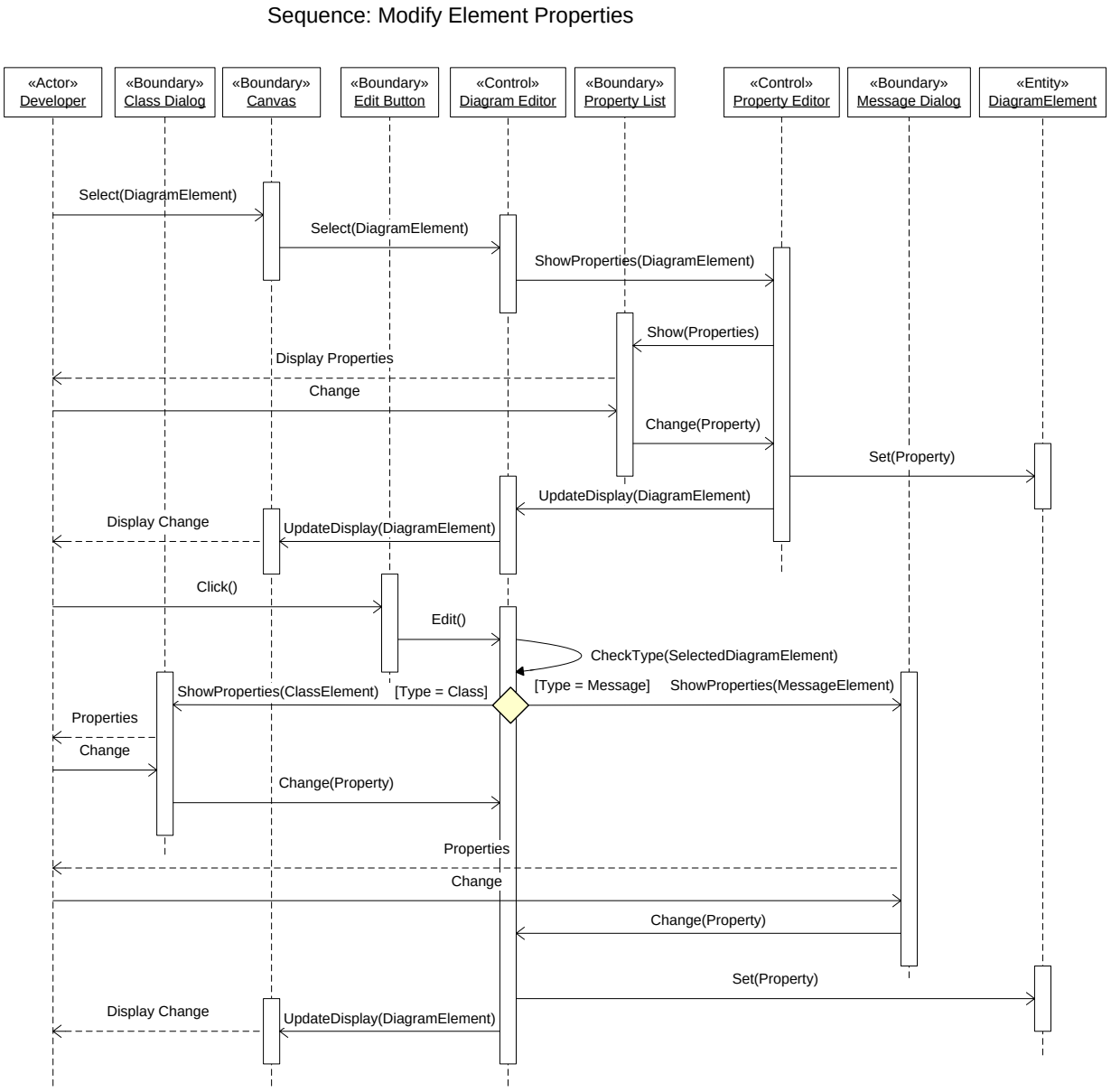

Figure 8: Sequence Diagram for Modify Element Properties

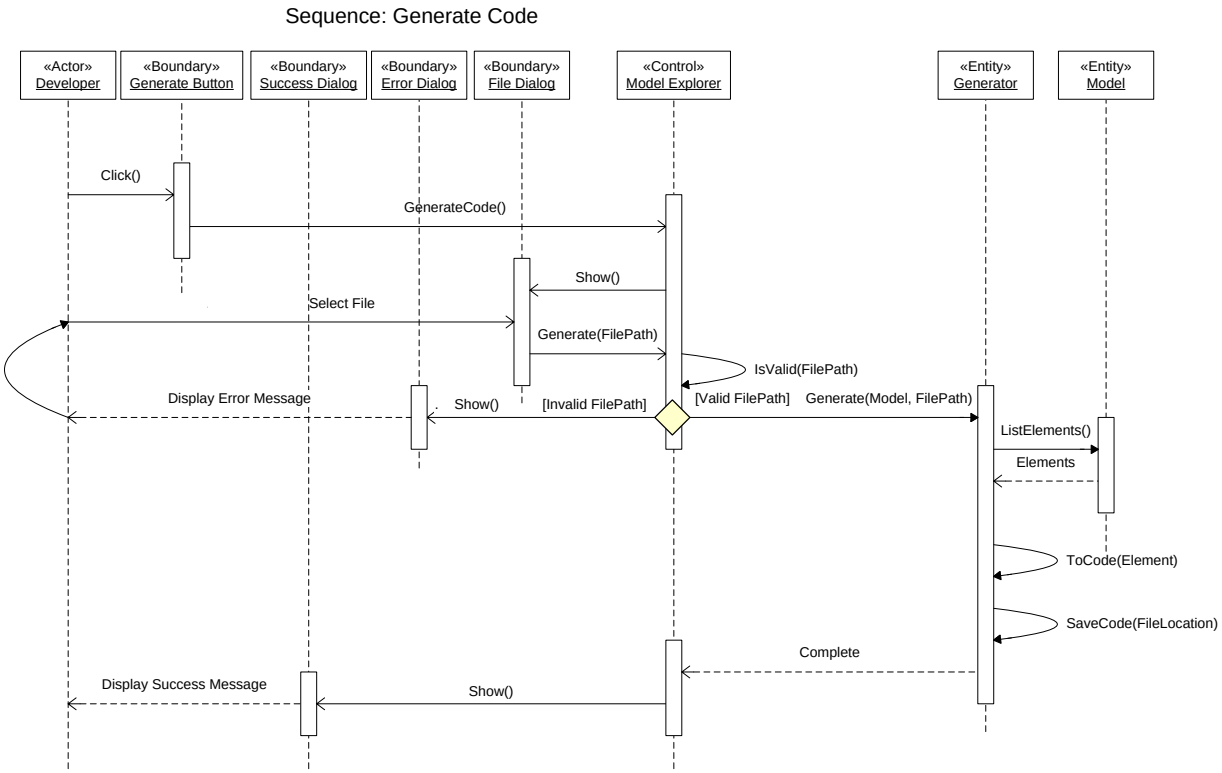

Figure 9: Sequence Diagram for Generate Code

4. Design

4.1. Design Goals

Performance Criteria

Response Time: The purpose of the system is to improve the productivity of developers, and time is an important aspect of productivity. Any time wasted while the user waits for a response adds to the total delay of the user‟s work and also causes unwelcome interruption to the workflow and concentration of the user. Therefore, response lag should be kept to a minimum. The system should be responsive to user commands in the design environment without significant visible lag (it should to respond to modeling commands immediately). The response time during saving and loading models should also be quick, but not necessarily as fast as modeling commands because these tasks are not performed at a high frequency.

Dependability Criteria

Reliability: The most important part of this system is the data. The models which are modified during the operation of the system represent a great amount of time and effort invested by the user and therefore it is critical to prevent any unintended loss of data. The system should not fail in such a way that data is lost or modified in an irreversible manner.

Robustness: Another aspect of data safety is consistency. Incorrect user input may lead to inconsistent state of data and unpredictable output. In order to avoid this, the design of the system should minimize incorrect input from the user. Whenever the user has the opportunity to provide incorrect input, it should be identified and reported to the user.

End User Criteria

Usability: The usability of a system is also a factor in the productivity of the user. The user should be able to concentrate on the work rather than how to operate the system. Time should not be wasted searching for tools or commands, and the number of interactions required to access a command should be minimized. Therefore, user interface elements should be clearly labeled and all tools and commands used within the design environment should be available within a few interactions (mouse clicks).

Maintenance Criteria

Extensibility: The system under development is restricted in a number of ways. One of these is the output programming language. In order to be useful to a wider audience of developers, it may be important in future to remove this restriction. In order to make this possible, the design of the system should attempt to minimize the language specific aspects of the system and limit the use of language specific algorithms to only one subsystem so that other subsystems can be reused without modification in case a different output language is required.

4.2. Software Architecture

The software architecture applied in the development of the system is a variant of the model-view-controller (MVC) architecture. The MVC design pattern divides the system into three parts. A model is the representation of the domain data which the system is operating on. A view is the part of the system responsible for the presentation of the data and a controller is the part of the system accepting input and translating modifications to the model.

MVC is traditionally applied in large scale web applications to separate client side pages from server side application logic and storage. MVC is especially important for web applications since the separation of the system into parts makes it easier to support multiple views on the same data. In the case of this system, the „model" in the architecture represents an actual UML model containing elements which may be viewed and modified using different aspects of the user interface. For instance, modification of the location of an element can be made in the property editor and then reflected back to the diagram editor as a change in the visual position of an element. This kind of interaction is typical of the MVC architecture and therefore the software architecture applied for this system can be categorized as MVC.

4.3. Subsystem Decomposition

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, but most subsystems depend on one or more of the other subsystems for their operation. The subsystems and their inter-dependencies are shown in Figure 10 below, followed by a description of each subsystem.

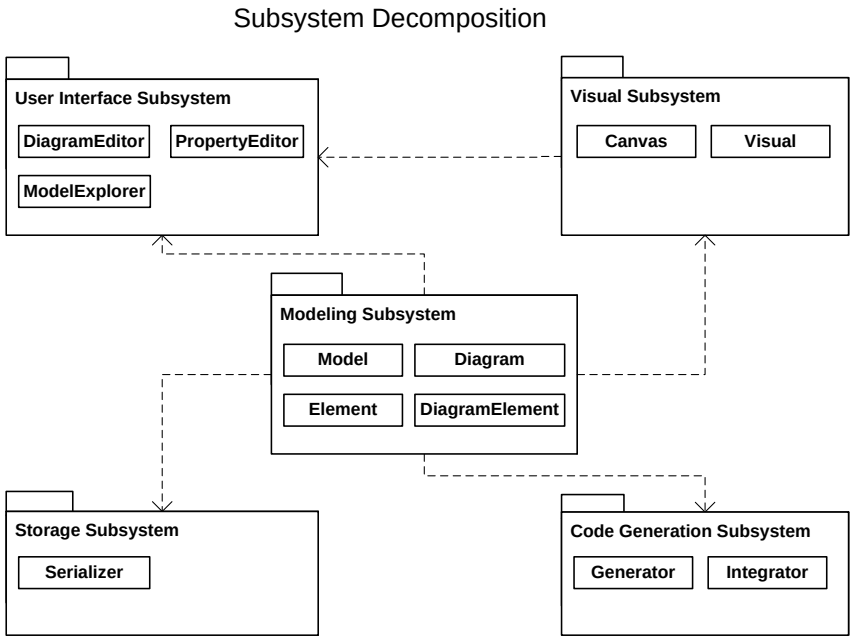

Figure 10: Subsystem Decomposition of the System

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 relationships 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.

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 Serializer.

Code Subsystem

This subsystem is responsible for generating programming language 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.

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 each element and the relationship amongst 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.

User Interface Subsystem

This subsystem contains the classes for each of the major building blocks of the user interface of the system. The classes in the user interface subsystem each encapsulate a single part of the UI functionality of the system 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.

4.4. Persistent Data Management

The system has two kinds of output which are persistent. The first is the model itself, including the various diagrams and the locations and relationships of elements on each diagram, which must be stored and retrieved from a file. The other is the code, which must be generated and then stored as a set of files in a format that is understood by the IDE and compiler. The code must also be retrieved for integration after any changes.

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

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 transmission. 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.

There are two major problems that had to be overcome for XML serialization to be used as the storage format for the model. The first issue is that XML serialization provided on the chosen platform has a limitation of storing only attributes with public visibility. As the system is an object oriented system, it is not acceptable to make the visibility of all attributes of the model public only for the purpose of storage. Therefore the solution was to create sister classes which replicate the attributes of each class but with public visibility and without the operations, and add conversion methods to and from the storage classes, as shown below in Figure 11.

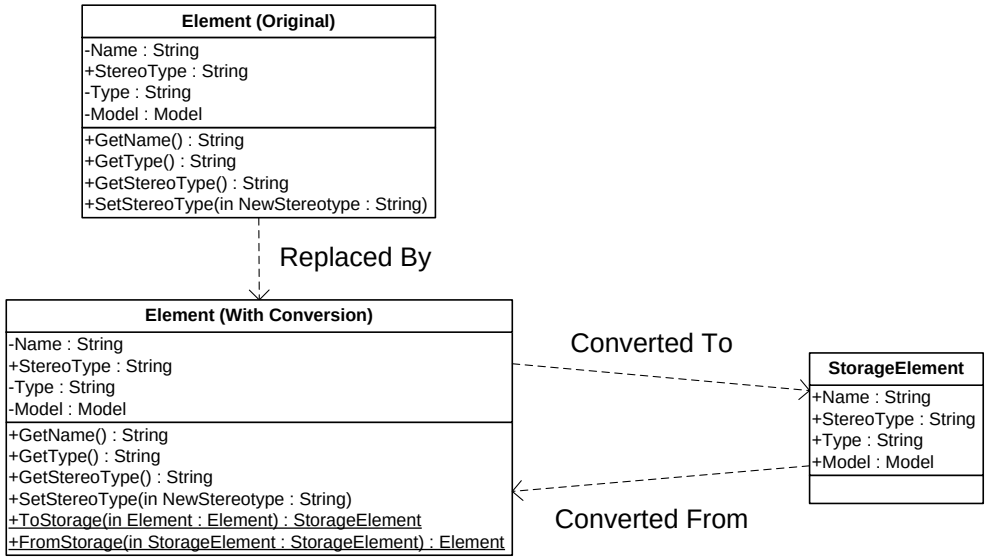

Figure 11: Sample of class mapping for XML serialization

The second problem encountered was that XML output is repeated for every reference to an object. That is, if the same object is referenced by two other objects, all its attributes will be present in the XML nesting structure of both objects. This is the nature of the XML specification rather than a bug, but it is still a problem for storage. Additionally, while at first this problem seems to be a minor issue of XML document size, it also presents complications during saving and loading the document. In serializing an object, the platform serializer is unable to process circular references. If two objects reference each other, even indirectly, serialization would become infinite so this case is detected and stopped. Since the model includes many cases of cross referencing, this was a serious problem. The other complication, which occurs during loading, is that the duplicate representations are instantiated separately and we end up with multiple objects instead of a single object, which renders useless all the references that must be preserved.

The solution for this problem was to add a conversion step before serialization where all objects are given a unique identifier, with the identifier taking the place of the attributes at each reference of the object (Figure 12). All objects are then stored in a linear (or flat) XML representation with the relationship hierarchy restored upon loading by using the identifiers to recreate references correctly.

Figure 12: Sample of reference attribute replacement

5. Implementation

5.1. Tools

The development platform used for this system is the .Net framework. The Microsoft Visual Studio 2008 IDE, which contains framework version 3.5, was chosen tool for the programming of the system. Specifically, the VB.net programming language was used. Visual studio has a rich set of built-in libraries, and the XML serialization as well as 2D drawing and rendering functionality depends on two of the built-in libraries. The property grid control was also reused from the available control libraries of visual studio.

5.2. Prototype

The prototype of the system is a graphical windows application. In this subsection, the major user interface elements of the main window will be discussed briefly, followed by editing of the three diagram types. Finally, the code generation and XML storage functionality of the system will be shown. The sample models and code in this section are shown with default names like „actor1" for the purpose of the illustrations, and would assume actual names when the system is used to develop a practical project. The initial window which appears when running the system only provides two functionalities, open model and new model, as seen in Figure 13. Once the user has created a new model or opened an existing model using a standard file dialog, further commands become available and the user interface looks more like that of Figure 14.

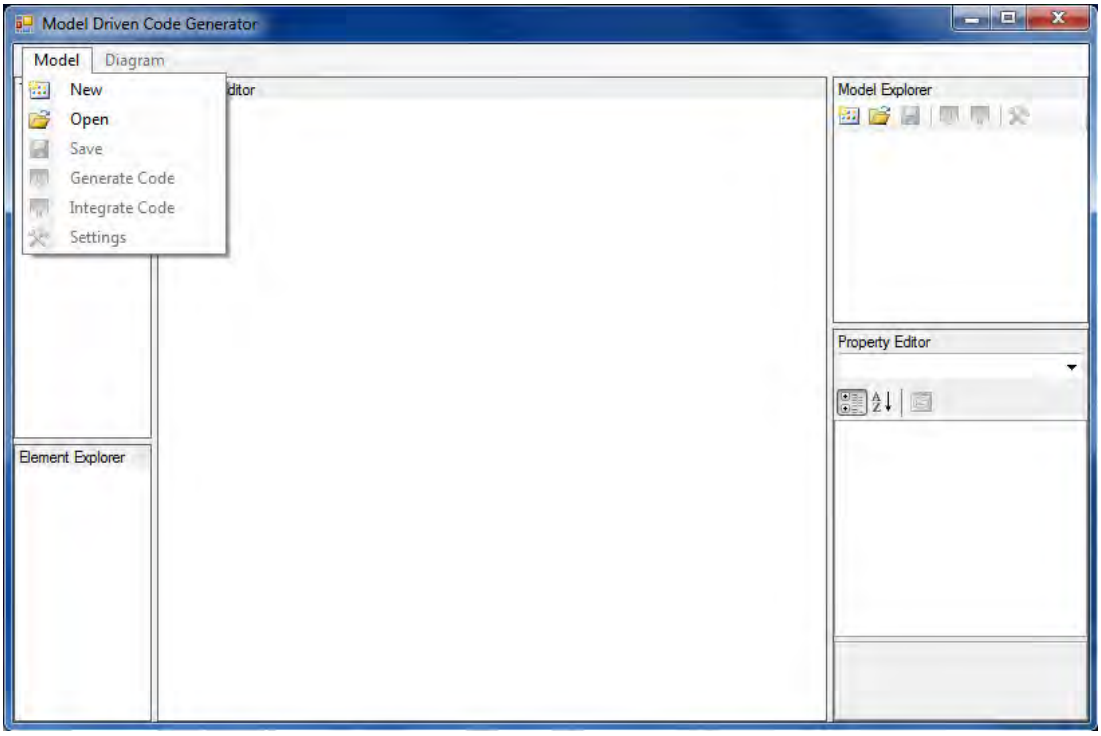

Figure 13: Screenshot of the initial state of the system

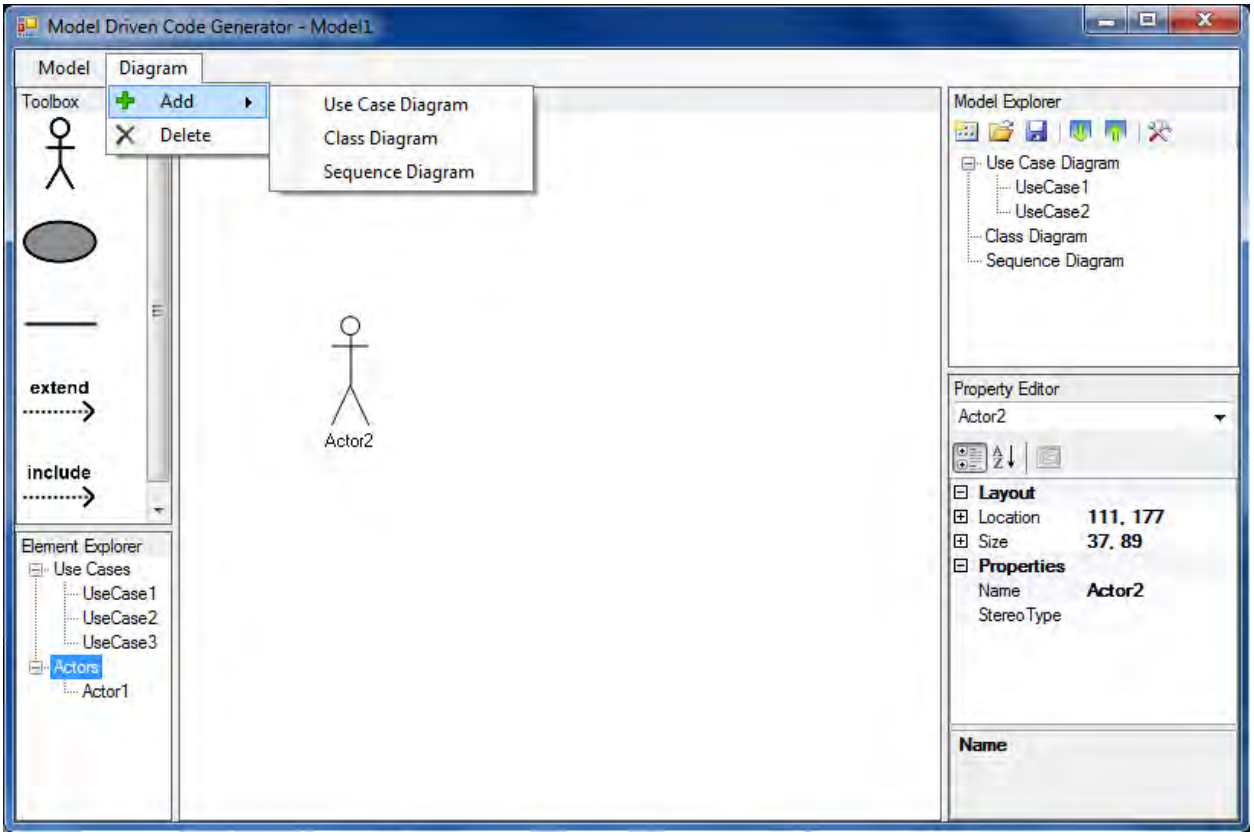

Figure 14: Screenshot of the typical working state of the system

Menus

The menu bar at the top contains the model and diagram drop down menus. The model menu contains commands for creating, opening and saving a model as well as items to generate and integrate code and finally change settings. The diagram menu has a delete command and a submenu for adding new instances of the different diagram types to the model.

Model Explorer

This is the main point of interaction with the model, where diagrams can be added, removed, renamed and shown via contextual commands. A small toolbar also replicates the model menu functionalities for quick access. All newly created diagrams will be named sequentially, with a number appended to the type of the diagram.

Toolbox

Once a diagram is shown (active), the toolbox is populated with the types of elements that can participate in the active diagram. The user can drag the icon of an element into the diagram editor to create a new element within the 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 add them using the same drag-and-drop action.

Diagram Editor

The diagram editor is the main point of interaction with the diagram and elements of the model. Once the user has added an element to the diagram editor area, it can be moved around by dragging it to the appropriate location. Some elements provide resizing capability or can be dragged onto other elements to create relationships. The diagram editor area can expand to show large diagrams with vertical and horizontal scrolling. An element on the diagram editor can be removed through a context menu command.

Property Editor

The property editor shows the properties of the element selected in the diagram editor. Location, size and name are common to all elements, but other properties appear depending on the selected element. All elements can be renamed using the property editor; however illegal characters (space and all special characters) or identical names are rejected with an error dialog.

Use case diagrams

A use case diagram can contain two types of elements, „actor“ and „use case“, along with three types of relationship elements and a boundary element. The „include“ and „extend“ relationship elements are used between use cases, while the „communication“ element relates a use case with an actor. An actor is represented by a stick figure and a use case is represented by an oval shape. A system boundary element is automatically created and resized to contain all use cases of the diagram as they are added, moved and deleted. The element explorer lists all actors and use cases of the model when a use case diagram is active. Any relationship which is created between elements of a use case diagram applies across all diagrams. When two related elements are present in a diagram, the relationship is automatically added to the diagram. When a relationship is removed from one diagram, all diagrams are affected. New relationships are created by dragging the endpoint of a relationship element on top of another element which must be of the appropriate type. Dragging a communication which is already related to a use case succeeds only if the element it is dropped on is an actor. Relationships are automatically repositioned to the shortest distance between anchor points on the two related elements whenever either of the elements is moved. Figure 15 shows the state of the system when editing a use case diagram.

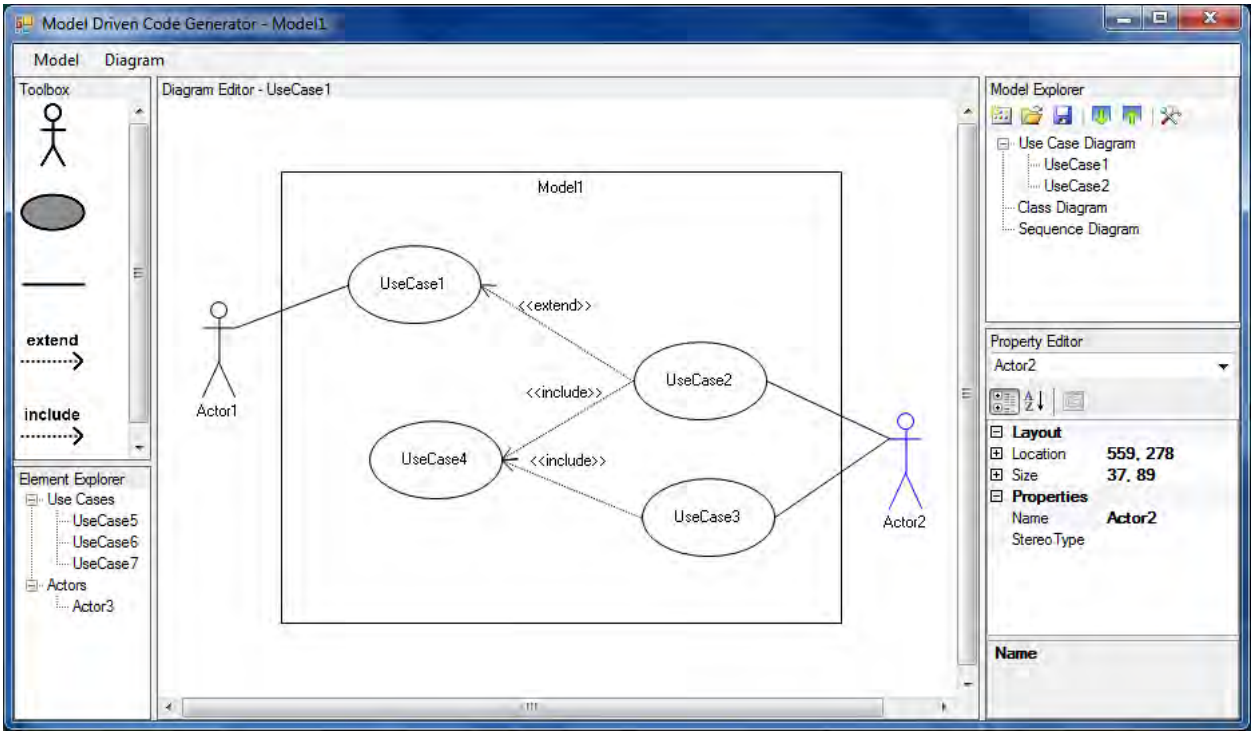

Figure 15: Screenshot of use case diagram editing

Class diagrams

A class diagram can contain a class element and two relationship elements, generalization and binary association. Generalization is shown by a line with a closed arrow at one end. A binary association is also shown as a line, with multiplicity values for both ends that can be changed in the property editor. A class is depicted as a rectangle with three sections, with the name of the class shown in the first section. The attributes of the class are shown in the next section and operations are shown in the final section. Attributes and operations are represented by text which indicates their properties according to UML rules. A class element can be resized to show all its attributes and operations.

In addition to changing the name of a class using the property editor, the attributes and operations of a class can be edited using a class editor dialog. The dialog is show when the edit context menu item is clicked or when a class element is double-clicked. The dialog allows the user to add and remove attributes and operations of the class. Operation parameters are also edited in the dialog after selecting one of the operations of the class. Attributes, operations and parameters can be reordered according to the user‟s requirement, and the order is maintained when the class is drawn in the diagram editor. The state of the system during editing of a class diagram can be seen on Figure 16.

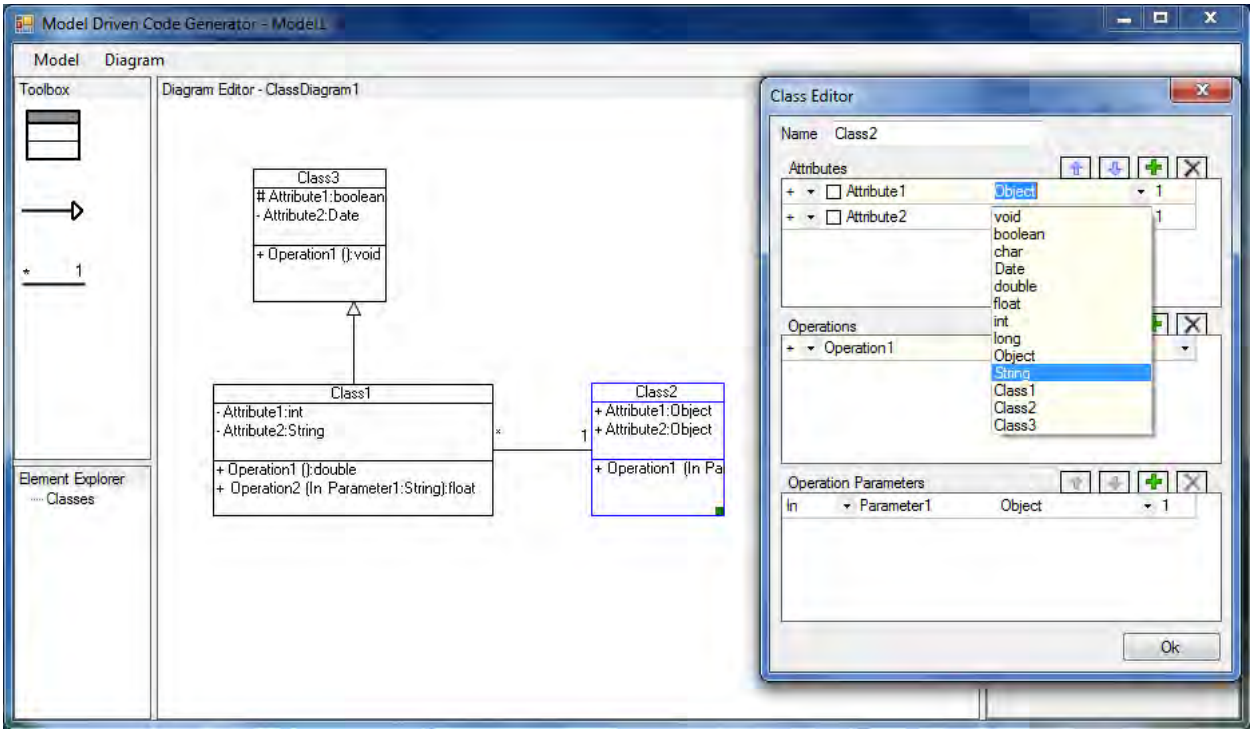

Figure 16: Screenshot of class diagram editing

Sequence diagrams

A sequence diagram can contain actor and object elements as well as three types of message elements. An actor in a sequence diagram is depicted by a slightly smaller stick figure with a name and a lifeline below it. An object is represented by a rectangle with its name, the stereotype and the name of the class it belongs to inside, and a lifeline below it. An asynchronous message element is shown by a solid line with an open arrow, a synchronous call with a solid line and a black closed arrow and a return message with an open arrow and a dotted line. Figure 17 shows the state of the system when a sequence diagram is active.

An existing actor from a use case diagram can be reused via the element explorer, as can an object element created in another sequence diagram. Classes existing in the model are also listed, but dragging a class element will create a new object element belonging to the class. A message between two objects only exists in the diagram where it was created and is not automatically recreated when both sides are present in another diagram.

Message elements have additional properties to indicate which operation of the receiver object is invoked. Double clicking a message or selecting the edit item on the context menu brings up the message editor dialog. This dialog allows the user to choose which operation is invoked on the receiver, which sender attributes or constant values are sent as parameters and which sender attribute accepts the return value of the operation. The user can also choose to hide one or both sides of this information from being displayed above the message arrow in the diagram.

Life lines in sequence diagrams can also contain activation areas, which are shown as rectangular strips running down a section of the lifeline. Activations can be resized and moved in increments of several pixels at a time, but can never overlap. A message endpoint can be attached to anchors in an empty section of the lifeline or on one side of an activation block.

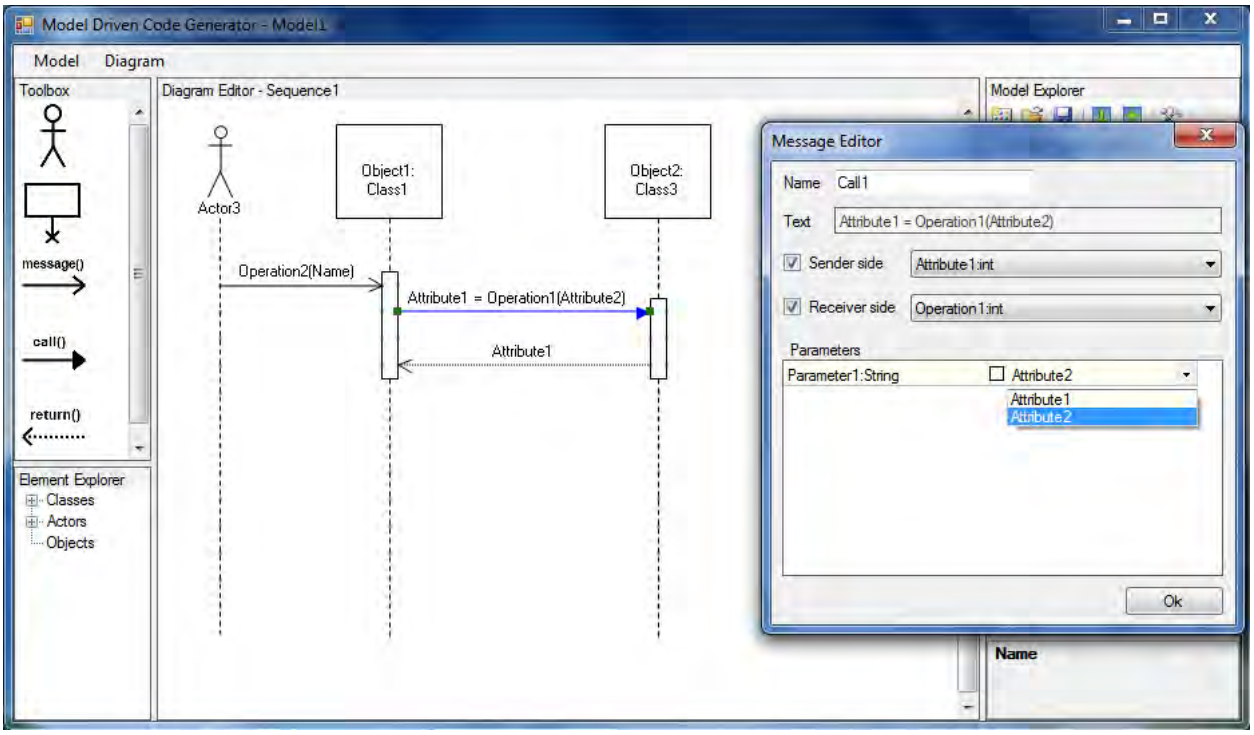

Figure 17: Screenshot of sequence diagram editing

Code generation and integration

The code generation and integration functionalities require almost identical interaction from the user so only code generation will be discussed from that perspective. Code generation can be activated by using the model menu or the toolbar button in the model explorer. The code generation dialog (Figure 18) is then shown, and allows the user to change code generation settings like whether the system should remove classes found in the code that do not exist in the model, or whether to rename the code module if the name of the model has changed. Once the user is satisfied with the settings, the file to be used for generation must be selected.

The user can choose to create a new file; in which case the folder path must be selected using a standard windows folder dialog and the file name is automatically set to the model name with a number appended if the name is already used. The user can also overwrite an existing file, by selecting it via a standard file dialog. Once a file is selected, the user clicks on the generate button to start the process. If a valid file name has been given, the system will convert elements of the model into code and write the code to the file, finally displaying a success dialog. If the file name is invalid, an error dialog is shown after which the user can change the file selection.

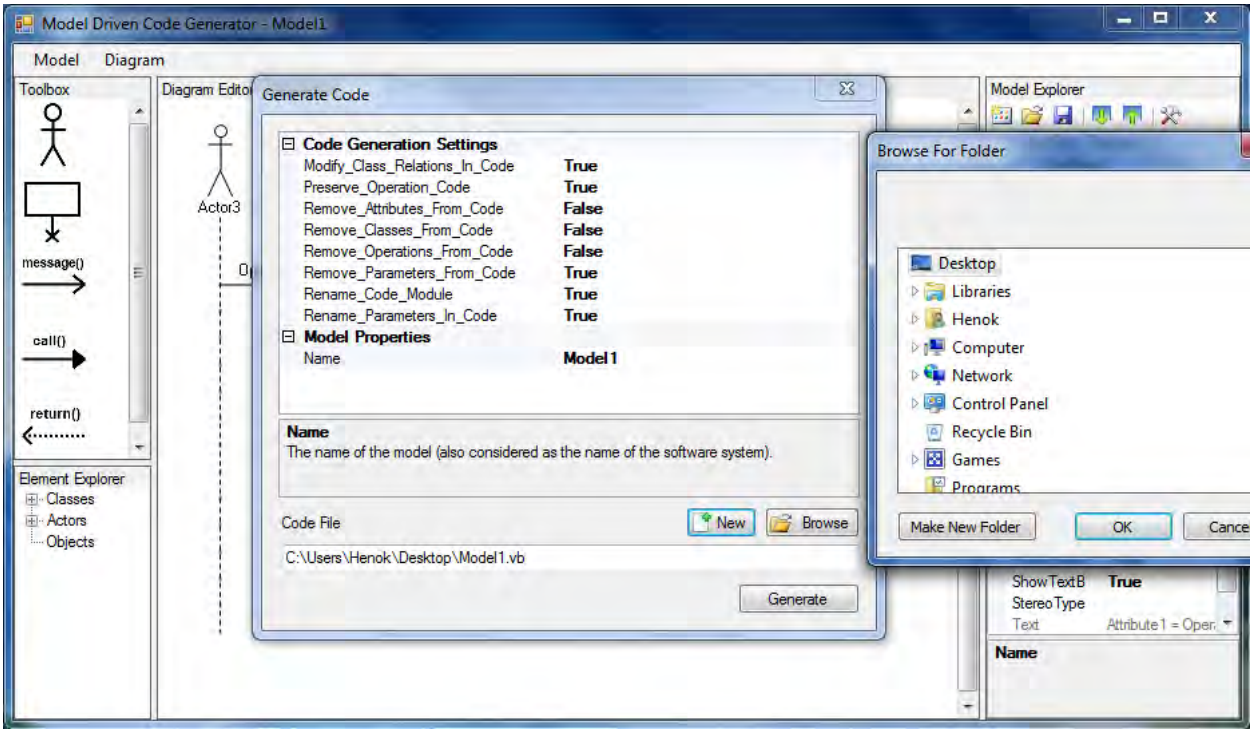

Figure 18: Screenshot of code generation dialog

The code generation algorithm works by parsing the contents of the code file into a set of objects which represent the different aspects of the code. The model elements are then iterated, and each element is matched to an existing element in the code objects. If a match is not found, a new code object is created to represent the model element. Otherwise, the parts of the object and the element are compared and new parts from the model are created as necessary. The generation settings are applied during this series of transformations. Once the code objects represent the current version of the model, the entire set of code objects is then parsed back into code text and written to the file. If the original file is new or empty, the generation process is restricted to the creation of code objects from model elements. If the file has existing code, the matching process is carried out and differences are resolved according to the generation settings. Table 13 shows a sample of code generated by the system. Differences are resolved in the opposite direction during integration, with changes in code added or included into the model.

Table 13: Sample of generated code

```
Public Class Class3

    Protected Attribute1 As Boolean
    Private Attribute2 As Date

    Public Sub Operation1()

    End Sub

End Class
```

XML storage

To save or open a model, the user can select the corresponding menu item or the toolbar button. A standard file dialog is shown and the user can select a file location. During a save operation, the system converts the model into XML format and stores it in the desired location. During a load operation, the XML data is reconstituted into the model object. The details of the XML storage and retrieval technique has already been discussed in the persistent data management section of the previous chapter. The „HandleReference" type shown in Table 14 below actually contains the id used to recreate references.

Table 14: Sample of XML storage format

```
<?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>53</Id>
          </StorageHandle>
...
    - <StorageHandle xsi:type="ActorElement">
        <Id>2</Id>
        <Name>Actor1</Name>
      </StorageHandle>
    - <StorageHandle xsi:type="UseCaseElement">
        <Id>3</Id>
        <Name>UseCase1</Name>
      </StorageHandle>
...
    </StorageHandle>
  </StorageHandles>
</StorageHandle>
```

6. Conclusion and Recommendations

The development of this system encompassed dealing with a variety of problems, including finding an effective representation of a UML model, developing an editing environment for display and interaction with UML elements, and storage of a model. The main purpose of the system is to enhance productivity by allowing developers to rely on automation for keeping models and code consistent. In this respect, the developed prototype represents a significant step by showing the viability of repeated generation from and integration of code into a UML model.

One of the goals during development was to reduce the affinity of the solution to the single language selected for output. The conceptual approach can be applied to any output language, and it was desirable to reflect this portability in the solution as well. The developed prototype has achieved this goal by separating the language specific elements into one subsystem.

Other goals such as performance and usability have been addressed in a satisfactory manner. Code generation and model storage are very fast operations and the design environment also performs without any visible lag. Most of the commands of the system are presented in the main window and drag-and-drop interaction ensures easy interaction for the user.

In conclusion, the solution presented here should be viewed not as a perfected end product but instead as a stepping stone in the direction of better tools for developers and greater usage of models in the software development process.

Recommendations

Due to time constraints and the limitations of scope, certain aspects of the system have been left out and there is room for improvement in the following ways:

- A number of UML diagram types have not been included in this work; addition of these diagram types may increase the application areas and usefulness of the tool.
- The decision element has not been incorporated into the sequence diagram, and reflexive (self referencing) relationships have not been included in either sequence or class diagrams. The effect of these elements may be relevant for better code output.
- The system mostly generates structural code. Improvements to the code generation algorithm to include more dynamic aspects of code would greatly enhance the value of this tool in improving developer productivity.
- Testing was only conducted at a basic level; additional testing with real and applicable software projects would help to reveal weaknesses and possible improvements.

7. References

[1] Martin Fowler, *UML distilled: a brief guide to the standard object modeling language*, Addison-Wesley, 2004

[2] Stephen J. Mellor, Kendall Scott and Dirk Weise, *MDA Distilled, Principles of Model Driven Architecture*, Addison-Wesley, 2004

[3] Andrew Watson, *Visual Modelling: past, present and future*, Whitepaper, retrieved from http://www.uml.org/Visual_Modeling.pdf on February 10, 2010

[4] Dan Pilone and Neil Pitman, *UML 2.0 in a nutshell*, O'Reilly, 2005

[5] Chris Raistrick and Paul Francis, *Model driven architecture with executable UML*, Cambridge University Press, 2004

[6] Fred Brooks, *No Silver Bullet — Essence and Accident in Software Engineering*, Proceedings of the IFIP Tenth World Computing Conference, 1986

[7] Carl Reynolds and Paul T. Tymann, *Schaum's Outline of Principles of Computer Science*, McGraw-Hill Professional, 2008

[8] B.S.Ainapure, *Object Oriented Modeling And Design*, Technical Publications, 2010

[9] Russell Miles and Kim Hamilton, *Learning UML 2.0*, O'Reilly, 2006

[10] Markus Voelter, Christian Salzmann and Michael Kircher, *Model Driven Software Development in the Context of Embedded Component Infrastructures*, Lecture notes in computer science Vol. 3778, pp 141-163, Birkhäuser, 2005

[11] Shari Lawrence Pfleeger, Joanne M. Atlee, *Software Engineering: Theory and Practice*, Prentice Hall, 2009

[12] Benjamin A. Lieberman, *The art of software modeling*, CRC Press, 2006

[13] Matthias Biehl and Welf Lowe, *Automated Architecture Consistency Checking for Model Driven Software Development*, Lecture Notes in Computer Science, Vol. 5581, pp 36-51, Springer, 2009

[14] Frank F. Tsui, *Managing software projects*, Jones & Bartlett Learning, 2004

[15] B. B. Agarwal, S. P. Tayal, M. Gupta, *Software Engineering & Testing: An Introduction*, Jones & Bartlett Learning, 2009

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: Henok Abye

Signature: _____

Date: _____

Confirmed by advisor:

Name: Dr. Dida Midekso

Signature: _____

Date: _____

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