FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Towards Model Driven Reverse Engineering to UML Behaviors - From C# Code to fUML Models

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Martin Lackner
Matrikelnummer 0927551

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel
Mitwirkung: Dipl.-Ing. Dr.rer.soc.oec. Tanja Mayerhofer, BSc

Wien, 9. Oktober 2017

(Unterschrift Martin Lackner) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Towards Model Driven Reverse Engineering to UML Behaviors - From C# Code to fUML Models

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Martin Lackner
Registration Number 0927551

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel
Assistance: Dipl.-Ing. Dr.rer.soc.oec. Tanja Mayerhofer, BSc

Vienna, 9. Oktober 2017

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Martin Lackner
Völkendorfer Straße 8/15, 9500 Villach

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Villach, 9. Oktober 2017)	(Unterschrift Martin Lackner)

Acknowledgements

I want to thank all the people who helped and supported me in completing this thesis. Exceptionally I want to thank Ms. Tanja Mayerhofer who has supported and guided me through this work, and especially on her patience. In addition I want to thank Mrs. Kappel for supervising my thesis. Last but not least, a big thanks to Sabrina who motivated me and went along with me during the whole studies.

Thank you!

Martin Lackner

Abstract

Model Driven Engineering (MDE) aims to raise the level of abstraction in software engineering by moving from code-centric approaches to model-centric ones, which means that the main artifacts in the software development process are models. Thereby, MDE can be used for both creating new software as well as modernizing or extending existing software. The latter usage scenario of MDE requires the reverse engineering (RE) of existing software into higher-level models. The main aim of RE is to extract information of existing software and obtain a more abstract view for further analysis. Model Driven Reverse Engineering (MDRE) is the application of MDE techniques to perform RE tasks. While many MDRE approaches for reverse engineering the structure of a system already exist, there is a lack of approaches for also reverse engineering the behavior of a system, especially detailed behavior descriptions including algorithmic details of the software. This work proposes an approach for overcoming this gap by using MDE techniques to reverse engineer the detailed behavior of a system.

The goal of this work is to elaborate a mapping between code written in the general purpose programming language C# and *UML* models conformant to *fUML*, by using an MDRE approach. The OMG standard *Semantics of a Foundational Subset for Executable UML Models* or *foundational UML (fUML)* is chosen because it is possible to precisely and completely define the behavior of a software system with fUML models. Thus, fUML is a suitable candidate for serving as target language for MDRE approaches that aim to reverse engineer the detailed behavior of a software system. A prototype has been developed in this thesis, which is able to reverse engineer code written in C# to models conformant to fUML, and store them in the UML modeling environment *Enterprise Architect*.

Furthermore, an evaluation framework was developed, which uses the developed prototype to show the feasibility and practicality of the proposed MDRE approach. This framework executes fUML conformant models reverse engineered from C# code within the UML modeling environment Enterprise Architect and compares the execution result of the model with the execution result of the reverse engineered C# code to evaluate whether the model and the code behave the same. Using this evaluation framework, two case studies have been carried out comprising 108 unit tests that evaluate the elaborated C#-to-fUML mappings. The evaluation results show the general feasibility and practicality of the proposed MDRE approach. Furthermore, it identifies that maintaining the correct object flow from C# code to fUML models is the most challenging part of the C#-to-fUML mapping.

Kurzfassung

Model Driven Engineering (MDE) zielt darauf ab die Abstraktionsebene in der Softwareentwicklung durch die Verwendung von Modellen zu erhöhen. MDE Techniken können sowohl zur Erstellung neuer Softwaresysteme verwendet werden, als auch um bestehende Softwaresysteme zu modernisieren oder erweitern. Die letztere Anwendung erfordert das Reverse Engineering (RE) von bestehenden Softwaresystemen zu abstrakteren Modellen. Das Hauptziel von RE ist es, relevante Informationen über bestehende Software zu extrahieren und in abstraktere Modelle für Analysezwecke abzubilden. Model Driven Reverse Engineering (MDRE) ist die Anwendung von MDE Techniken für RE Aufgaben. Während bereits viele Ansätze existieren um strukturelle Informationen über bestehende Systemen mittels RE in Modellen abzubilden, gibt es bis heute kaum RE Ansätze, die detaillierte Verhaltensbeschreibungen von bestehenden Systemen extrahieren und abbilden. Im Zuge dieser Diplomarbeit wurde ein RE Ansatz erarbeitet, welcher detaillierte Verhaltensmodelle erstellt.

Das Ziel dieser Arbeit ist es, ein Mapping zwischen Softwaresystemen, welche in der Programmiersprache C# programmiert sind, und UML Modellen, welche dem fUML Standard entsprechen, zu erarbeiten. Um das Verhalten von bestehenden Softwaresystemen zu beschreiben, wurde der OMG Standard *Semantics of a Foundational Subset for Executable UML Models*, oder *foundational UML (fUML)* gewählt, da man mit fUML Modellen das Verhalten von Softwaresystemen präzise und komplett beschreiben kann. Somit ist fUML ein geeigneter Kandidat als Zielsprache für einen MDRE Ansatz, welcher das detailierte Verhalten von Softwaresystemen abbilden soll. Im Zuge dieser Arbeit wurde auch ein Prototyp entwickelt, welcher in der Lage ist, Softwaresysteme, die in C# programmiert wurden, zu fUML konformen Modellen zu transformieren, und diese Modelle in der UML Modellierungsumgebung Enterprise Architekt zu persistieren.

Weiters wurde ein Evaluierungs-Framework entwickelt, welche den erstellten Prototypen nutzt, um die Machbarkeit und Praktikabilität des entwickelten MDRE Ansatzes zu zeigen. Dieses Framework ist in der Lage, fUML konforme Modelle, welche aus C# Code extrahiert wurden, in der Modellierungsumgebung Enterprise Architekt auszuführen, und die Ergebnisse der ausgeführten Modelle mit jenen des ausgeführten C# Codes zu vergleichen. Damit kann festgestellt werden, ob die berechneten Modelle und der untersuchte C# Code das selbe Verhalten aufweisen. Weiters wurden Fallstudien erarbeitet die 108 Unit Tests umfassen, um den entwickelten MDRE Ansatz zu evaluieren. Die Evaluierungsergebnisse zeigen die Machbarkeit und Praktikabilität des vorgestellten MDRE Ansatzes. Weiters wurde festgestellt, dass die korrekte Abbildung von Objektflüssen in C# Code die größte Herausforderung im MDRE Prozess darstellt.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Model driven engineering (MDE) aims to raise the level of abstraction by moving from code-centric approaches to model-centric ones [5]. This is achieved by establishing models as first class citizens and using models on different abstraction levels [24]. A very popular MDE implementation is the Model Driven Architecture (MDA[1]) framework by the Object Management Group (OMG[2]). It defines standards for applying MDE including the common metameta-model called Meta-Object Facility [20], the modeling language called Unified Modeling Language [21], language specifications for defining model transformations [17]), and many more.

MDE techniques can be used for creating new software as well as for modernizing or extending existing (legacy) software. This work focuses on the later use case, in particular the reverse engineering of existing software systems. Reverse engineering (RE) is the process of computing useful higher-level representations of existing systems. Initially invented for hardware analysis it quickly extended its focus to software systems. The main goal of RE is to obtain a better understanding of an existing software system to be able to update or upgrade it, reuse parts of it, or perform reengineering tasks [10].

Model driven reverse engineering (MDRE) is the application of MDE techniques to perform RE tasks [7]. It is used in software evolution scenarios, such as modifying legacy systems, performing technical migrations from obsolete technologies to recent ones, platform migration, refactoring tasks and maintenance scenarios.

MDRE techniques are required, whenever higher-level representations of a software system are needed, but missing. This might be the case when the system has not been developed in a model-based way or when the initial design models become obsolete due to evolutions of the system that were performed independent of these models. In such a scenario, MDRE techniques can be applied to obtain models of the existing system. Doing these tasks manually is too

[1] Model Driven Architecture - http://www.omg.org/mda/

[2] Object Management Group - http://www.omg.org/

time consuming and error prone, therefor an (semi-) automated process together with modeling standards is desireable [30].

In 2011, OMG published the specification *Semantics of a Foundational Subset for Executable UML Models* or fUML in short [25]. This standard selects a subset of the UML language units *Classes*, *Actions*, *Activities*, and *Common Behaviors*, and defines a precise execution semantics for this subset. With fUML it is possible to precisely and completely define the behavior of a software system. Thus, fUML is a suitable candidate for serving as target language for MDRE approaches that aim to reverse engineer the behavior of a software system. However, currently no MDRE approach exists, that supports the reverse engineering of software systems into fUML conforming models.

1.2 Problem Statement

Most of the current existing MDRE approaches utilize UML class diagrams and UML state machine as higher-level representations, of software systems, e.g., as target formalism for representing software systems on a higher level of abstraction. The problem with these existing approaches is that they are not mapping the whole behavior of software systems to the model level. One factor that contributes to that is the lack of a complete and precise semantics of UML. With the introduction of the fUML standard, a precise execution semantics for a defined subset of UML, was introduced. When using fUML as target formalism it is possible to model the whole behavior of a software system in a MDRE scenario using fUML conformant UML activity diagrams. This enables a more accurate analysis of the behavior of a software system, e.g., the analysis of non-functional properties described in the work of Fleck et al. [12]. Exhaustive code generation in a forward engineering process is another advantage of an existing fUML model containing the complete behavior of a software system. It is possible to change technologies or programming languages without loosing information about the behavior.

Currently there are no existing MDRE approaches for object oriented programming languages which are targeting fUML. The fUML standard proposes a mapping between Java and fUML, but this mapping is incomplete. This work focuses on generating fUML conformant activity diagrams out of code written in C# to reverse engineer software systems implemented in this object oriented programming language. As development/modeling environment, the UML modeling tool Enterprise Architect from Sparx Systems[3] is chosen, because it is widely used in industry.

1.3 Aim of the Work

The goal of this work is to elaborate a mapping between code written in the general purpose programming language C# and UML models conforming to fUML for the purpose of reverse engineering software systems implemented in C# into UML models. This can be divided into the following parts.

[3] Enterprise Architect - http://www.sparxsystems.com/products/ea/index.html

1. Mapping
   As first part, a mapping between the programming concepts provided by C# and the modeling concepts provided by fUML are elaborated. In particular, it is elaborated how to map C# classes to fUML classes for reverse engineering the structure of a software system and how to map C# method bodies to fUML activities for reverse engineering the behavior of a software system. Furthermore, existing discrepancies between C# concepts and fUML concepts are identified and a solution for overcoming these discrepancies are elaborated.

2. Prototypical implementation for Enterprise Architect
   A prototypical implementation capable of reading in C# code and generating corresponding fUML compliant activities in Enterprise Architect is developed to enable the validation of the elaborated mapping between C# and fUML based on case studies. This prototypical implementation can be divided into four components.

   - Parsing
     First, the C# code to be reverse engineered has to be parsed resulting in an abstract syntax tree (AST).
   - Transformation into a C# model
     The AST from the previous step is transformed into a C# model conformant to a C# metamodel. This C# model is elaborated within this thesis because there is no existing C# metamodel which is available in the programming language C#.
   - Transformation into an fUML model
     The C# model resulting from the previous step is transformed into an fUML compliant model consisting of classes and activities.
   - Persistance
     The last part of the implementation is concerned with storing the obtained fUML model in Enterprise Architect. Therefor, an Add-In for Enterprise Architect has to be implemented.

   The separation of the implementation into several components enables the exchange of different components to be able to address additional programming languages and target UML environments in future. This possibility has been kept in mind during the implementation and is realized by adequate interfaces between the different parts.

1.4 Methodological Approach

The main focus of this thesis is the development of an MDRE approach that enables the transformation of C# source code to fUML compliant models. The methodology of the thesis is based on the design science methodology [13].

The design science methodology is a research methodology which defines several guidelines aiming to produce information system artifacts efficiently [23]. The guidelines include the awareness of a problem, the development of an artifact, the evaluation of the produced artifact, and the communication of the research [13].

According to these guidelines, this thesis will be realized in four steps as described in the following:

- Analysis
  The first step is an intensive study of current state of the art MDRE methods that exist for generating behavioral UML models from code. As the goal of this thesis is to generate UML activity diagrams, in particular fUML compliant activity diagrams, this step should also result in a better understanding of fUML. A profound analysis of the interfaces, which Enterprise Architect provides, will also be performed because of the practical part of the work. To be able to generate fUML compliant models in Enterprise Architect it must also be analyzed how C# code can be analyzed to gain the needed information about the code.

- Elaboration of a mapping from C# to fUML
  In this step, the mapping of C# classes to fUML classes, as well as the mapping of C# methods to fUML activities has to be elaborated. Potential discrepancies between fUML models and C# features should be also revealed in this step. Possible solutions for overcoming these discrepancies should be elaborated.

- Development
  This step consists of the development of a prototype integrated with Enterprise Architect that reads in C# code and generates fUML compliant activity diagrams. This includes the parsing of C# code, the creation of a C# code model, and the creation of an fUML compliant model in Enterprise Architect. The behavior of the reverse-engineered C# source code should be preserved by the transformation.

- Evaluation
  To evaluate the mapping between C# and fUML for reverse engineering software applications developed in C#, case studies are carried out. In the case studies, C# applications will be reverse engineered with the developed prototype. The resulting fUML models will be evaluated with regard to (i) completeness, i.e., all information available in the source code shall be retained in the fUML models, (ii) standard conformance, i.e., the fUML models shall conform to fUML such that they can be executed with the standardized fUML virtual machine, and (iii) behavioral equivalence, i.e., the C# code and the resulting fUML models considered in the case studies shall have the same behavior. Brunflicker [8] elaborated a prototype which is capable of executing fUML compliant models using the execution engine specified in the fUML standard, within the UML modeling environment Enterprise Architect. The evaluation uses this work, which enables the execution of the reverse engineered fUML compliant models in Enterprise Architect.

1.5 Structure of the Work

The remainder of this work is structured as follows.

Chapter 2 discusses reverse engineering and its applications, gives a brief introduction into the fUML standard of the OMG, and introduces the UML modeling environment Enterprise Architect.

Chapter 3 is concerned with the mapping between features of the object-oriented programming language C# and the concepts of fUML. It defines how to map C# classes to fUML classes for reverse engineering the structure of a software system and how to map C# method bodies to fUML activities for reverse engineering the behavior of a software system.

Chapter 4 is concerned with the implementation of a prototype which is capable of transforming C# software systems to fUML compliant models in Enterprise Architect according to the mappings introduced in Chapter 3. This includes the parsing of the source code, the transformation into a C# model, the transformation into an fUML model, and the persistance of the fUML model in Enterprise Architect.

Chapter 5 is concerned with the evaluation of the mapping between C# and fUML. In particular, it reports on case studies that were carried out using the implemented prototype and discusses the appropriateness of the elaborated C#-to-fUML mappings.

Chapter 6 discusses related work, in particular, related MDRE approaches are described and compared to this work.

Chapter 7 concludes this work and gives an outlook to future work.

CHAPTER 2

Background

2.1 Reverse Engineering

In the software development life-cycle, the maintenance phase can be seen as the most cost-intensive phase [30]. Software systems will be changed over time in terms of bringing in new functionality, bug-fixing or migrating it to other environments. This so called *software evolution* process is of high importance in software development because software systems are commonly used over many years once they are implemented. When using MDE approaches for creating new software systems there might be models available, but over time these models are often outdated due to evolutions of the software systems that were performed independent of existing models. If software systems have not been developed in a model-based way there are often almost no models available and the most accurate and reliable description of the existing software system is its source code. Reverse engineering is used to provide higher level views of existing systems to gain better understanding of it. It helps to find high level information about systems and provide indications about the impact of changes [30].

There are two different approaches for doing reverse engineering. In the first, *dynamic* analysis is used, e.g., if the source code is not or only partly available. Software systems will be executed and trace information will be collected during running the software. This includes information about objects which are manipulated and methods which are executed during the run. From this tracing information, high level models representing the analyzed system are obtained. This kind of reverse engineering can only capture parts of software systems, because not all possible execution traces can be collected during one run of a software system. The second approach is *static* analysis which statically analyzes the source code of the system and thus produces higher level models for all possible inputs and all execution traces, but needs the whole source code of a software system, or at least the part which has to be reverse engineered [30] [9]. These two approaches can also be combined, e.g., static analysis can be used to recover the structure of a system, and dynamic analysis is used to automatically search for behavioral patterns using pattern matching algorithms [29]. The prototypical implementation described in this thesis uses static analysis to reverse engineer software systems written in the object oriented programming language C#.

In MDRE, different artifacts of an existing software system are used, like source code and configuration files, to produce a set of models on a different abstraction level for the representation of such a system. These higher level models can be used for many different purposes, e.g., automated evaluation, quality assurance, further model-to-model transformations, etc. Usually there are three phases included in a MDRE process. The first phase is called *Model Discovery* which is used to create a set of initial models. These models represents the system at the same abstraction level without losing any information of the source artifacts. The second phase is called *Model Understanding* wherein different model manipulation techniques are used to obtain higher-level views of the source system. The third phase is called *Model (Re)Generation* which is used to display or generate the desired outcome of the whole reverse engineering process [6]. According to Brunelière et al. [7], a full MDRE approach must provide the characteristics of genericity, extensibility, full coverage, reusability, and automation. Genericity is achieved through the usage of technology independent standards where specific technology support can be plugged into generic ones. Extensibility refers to the decoupling of the represented information and the different steps of the reverse engineering process. Full coverage refers to supporting different abstraction levels of the source artifacts. Reusability relates to the re-usage and integration of provided components and the obtained models. Automation is achieved through relying on predefined sets of model transformations which are already available due to the usage of MDE techniques.

2.2 fUML

The current version of fUML is 1.3 [27] and was released in June 2017. fUML defines a subset of UML 2 and specifies foundational execution semantics for this subset. The fUML specification can be divided into three sections, the abstract syntax metamodel, the execution model and the foundational model library. The abstract syntax metamodel defines the fUML metamodel, which is a subset of the UML 2 Superstructure [22] metamodel with similarly named packages and introduces additional constraints on some elements. The execution model defines the semantics of fUML. The foundational model library defines primitive types and behaviors operating on these types. As this work focuses on generating fUML conformant models, the abstract syntax of fUML is described briefly in the following.

Abstract Syntax

The abstract syntax of fUML is defined as a subset of the UML metamodel, and contains UML class modeling concepts for defining the structure of a system, and activity modeling concepts for defining the behavior of a system. Furthermore, additional well-formed rules expressed as OCL constraints are defined in the fUML standard.

Structure. Figure 2.1 depicts an overview of the fUML metamodel for modeling the structure

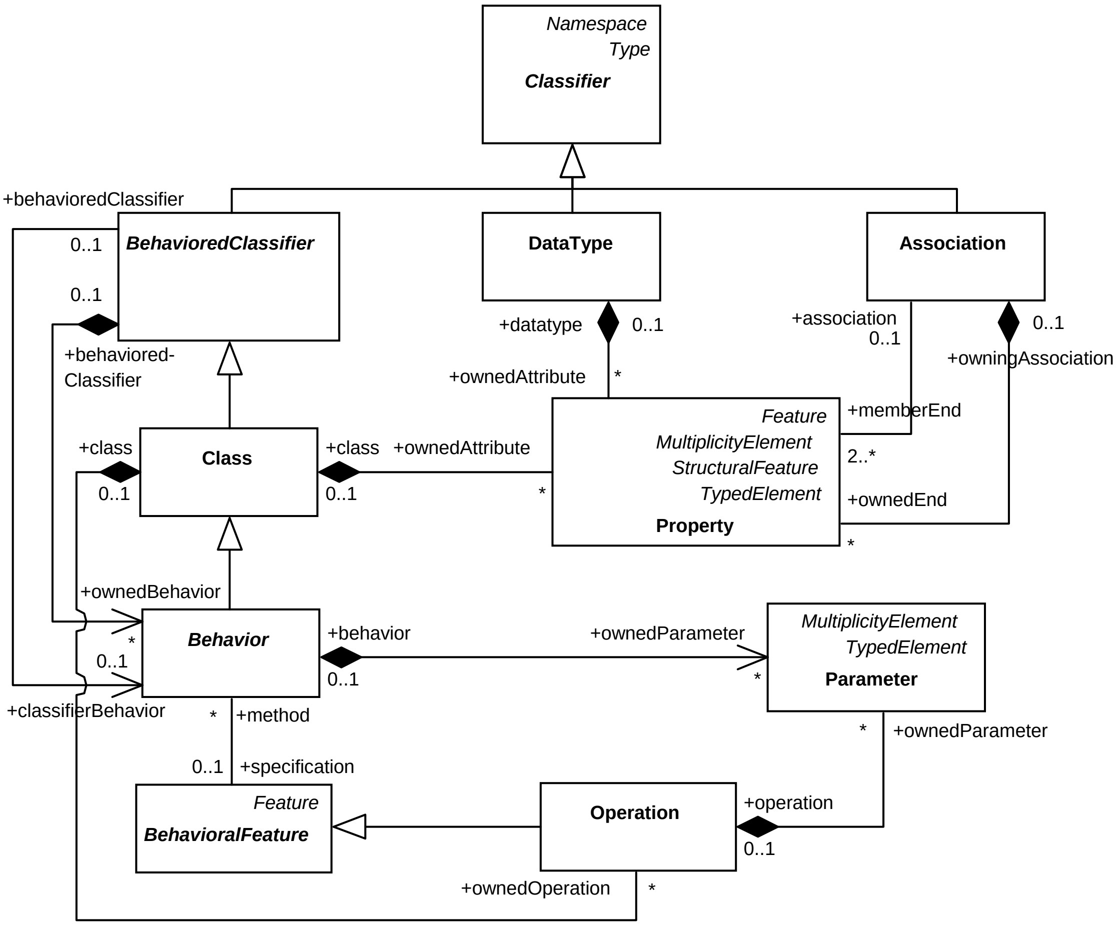


Figure 2.1: Excerpt of the fUML metamodel for modeling the structure of a system and the connections to the behavior.

of a system. Classes (class *Class*) can own attributes (class *Property*), possible links between class instances are defined by associations (class *Association*), and classes can own operations (class *Operation*), which can have parameters (class *Parameter*). A property can be associated to a structured data type (class *DataType*), which is a type whose instances are identified only by their value. In this figure, we can also see the link between the structure and the behavior of a system. In particular, behaviors (class *Behavior*) provide implementations of operations.

Behavior. Figure 2.2 depicts an overview of the fUML metamodel for modeling activities and thus the behavior of a system. Activities (class *Activity*) consists of nodes (class *ActivityNode*) and edges (class *ActivityEdge*). Edges are used to connect nodes with each other, either by control flow edges (class *ControlFlow*) for defining the control flow between the nodes, or by object flow edges (class *ObjectFlow*) for defining the flow of data between the nodes. Nodes are separated into actions (class *Action*), control nodes (class *ControlNode*), and object nodes (class *ObjectNode*). Actions are the fundamental unit of executable functionality in fUML. Control

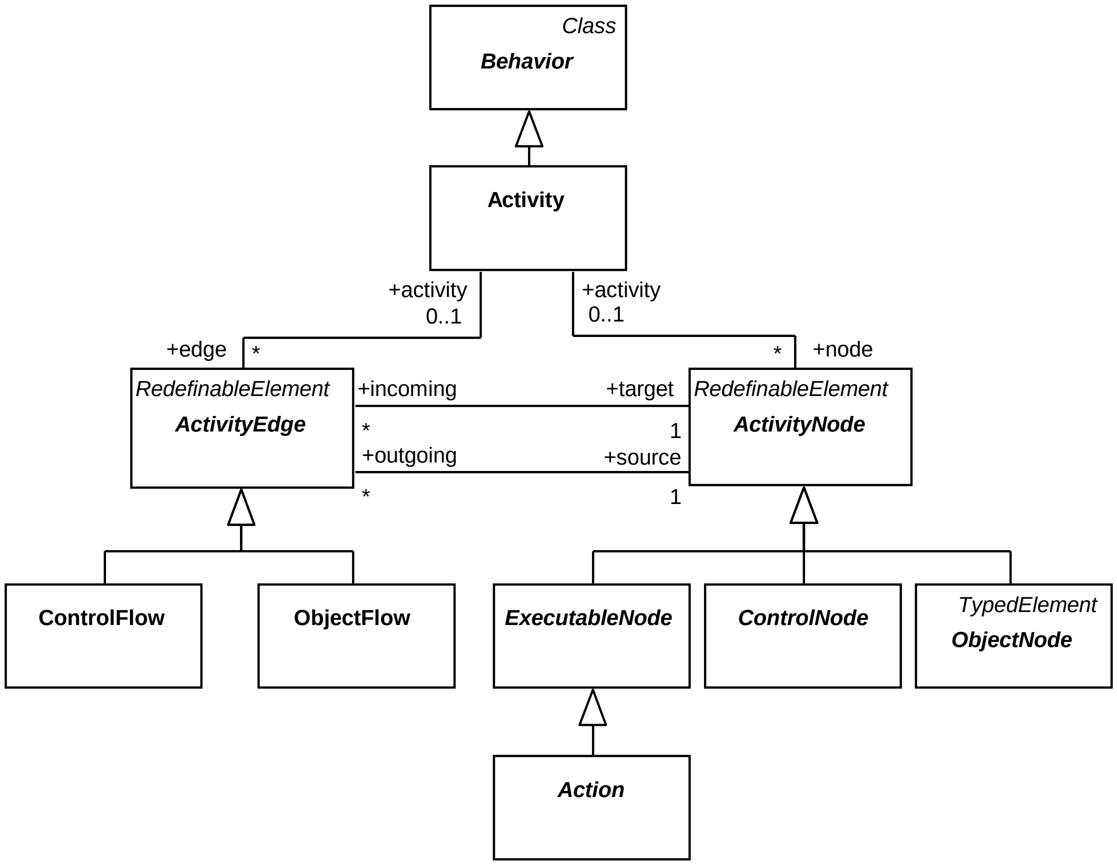


Figure 2.2: Excerpt of the fUML metamodel for modeling the behavior of a system.

nodes are used to coordinate the flows between other nodes, for defining the start and the end of an activity, and to model alternative and concurrent branches. Object nodes are used for defining the input and output of actions and activities.

Action Language

The fUML subset contains 27 types of actions for different purposes. They can be categorized into structural feature actions, object actions, link actions, communication actions, and structured activity nodes.

Structural Feature Actions

A structural feature action supports the reading and writing of structural features of objects, which are instances of classes defined in an fUML model. It uses an input pin to obtain the object whose structural feature should be accessed. Table 2.5 depicts the different structural feature actions of fUML.

Action	**Description**
AddStructuralFeatureValue	Adds a value for the defined structural feature to the provided object.
RemoveStructuralFeatureValue	Removes a value for the defined structural feature from the provided object.
ClearStructuralFeature	Removes all values for the defined structural feature from the provided object.
ReadStructuralFeature	Reads the value for the defined structural feature from the provided object.

Table 2.1: Structural feature actions of fUML.

Object Actions

Object actions are used for handling instances of classes. Table 2.2 depicts the different object actions of fUML.

Action	**Description**
CreateObject	Creates an object (instance) of a statically specified classifier.
DestroyObject	Destroys the provided object.
ReadSelf	Provides the host object of the current behavior.
ReadIsClassifiedObject	Returns a Boolean value which indicates whether the provided object is an instance of the defined classifier.
ReclassifyObject	Adds the given classifiers to an object and removes the old classifiers from that object.
ReadExtent	Returns all instances of the defined classifier.
StartObjectBehavior	Invokes an instantiated behavior of the provided object or the classifier behavior of the provided object's type.
StartClassifiedBehavior	Invokes the classifier behavior of the provided object.

Table 2.2: Object actions of fUML.

Link Actions

Link actions are used for working on links, which are identified by the objects and ends of the links. Table 2.3 depicts the different link actions of fUML.

Action	Description
CreateLink	Creates a link between the given objects.
DestroyLink	Destroys the link between the provided objects.
ReadLink	Retrieves an object linked with the provided objects via the specified association and association end.
ClearAssociation	Destroys all links of an association that have a particular object at one end.

Table 2.3: Link actions of fUML.

Communication Actions

Communication actions are used for communication between activities, and invoking activities synchronously or asynchronously. Table 2.4 depicts the different communication actions of fUML.

Action	Description
CallBehavior	Invokes the defined behavior synchronously.
CallOperation	Invokes the defined operation of the provided target object synchronously.
SendSignal	Creates a signal and transmits it to the specified target object.
AcceptEvent	Waits for the occurrence of an event meeting specified condition defined by a trigger.

Table 2.4: Communication actions of fUML.

Structured Activity Nodes

Structured activity nodes are executable activity nodes which are representing a structured portion of an activity. It is the only activity node which can contain other activity nodes. Table 2.5 depicts the different structured activity nodes of fUML.

Uncategorized Actions

There are other actions which are not part of any previously mentioned category. Table 2.6 depicts all uncategorized actions of fUML.

fUML Reference Implementation

Model Driven Solutions developed an open source implementation of fUML acting as reference implementation[1]. This reference implementation is intended to encourage UML tool vendors

[1] http://portal.modeldriven.org/project/foundationalUML

Action	Description
StructuredActivityNode	Executes the contained activity nodes.
ConditionalNode	A conditional node represents an exclusive choice among a number of alternatives. It executes one body section whose test section evaluates to true.
LoopNode	Represents a loop with setup, test, and body sections. It executes the body section as long as the test section evaluates to true.
ExpansionRegion	Executes the contained activity nodes multiple times corresponding to elements provided through an input collection.

Table 2.5: Structured activity nodes of fUML.

Action	Description
TestIdentity	Tests whether two values are identical.
ValueSpecification	Provides the value according to the defined specification.
Reduce	Reduces a collection to a single value by applying the defined behavior.

Table 2.6: Uncategorized actions of fUML.

to implement the fUML standard and in their tools is conformant to the fUML virtual machine described in the fUML standard [25]. The reference implementation allows to load UML conformant models which are stored in an XMI file and execute UML activities defined in such models. Produced output values are provided as output of the execution.

Brunflicker [8] uses this reference implementation in the prototype elaborated within his master's thesis for executing fUML compliant models within the UML modeling environment Enterprise Architect. This work uses this prototype for the evaluation of the proposed C#-to-fUML mappings which is further described in Chapter 5.

2.3 Enterprise Architect

Enterprise Architect is a UML modeling environment from the Australian company Sparx Systems Pty Ltd. The first version 1.1.3 was released about 17 years ago. The current version 13 was released in 2016. Enterprise Architect offers functionality for requirements management, project management, test management, code engineering, simulation, and many more. It supports many standards, such as UML 2, SysML[2], BPEL[3], and WSDL[4]. It provides full life cycle modeling for business and IT systems, software and systems engineering, and real-time and embedded development. Enterprise Architect is mainly designed for Windows systems, but can

[2] Systems Modeling Language - http://www.omgsysml.org/

[3] Business Process Execution Language - http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[4] Web Service Description Language - http://www.w3.org/TR/wsdl

also be used on UNIX based operating systems with the open source software WINE[5], which is a compatibility layer capable of running Windows applications on POSIX[6]-compliant operating systems [28].

Enterprise Architect offers an automation interface, which provides access to the internals of Enterprise Architect models from any development environment which is capable of generating ActiveX COM clients. The internal storage format of Enterprise Architect is a relational database, e.g., MySQL, Oracle, SQL Server. The universal open source database Firebird[7] or an Microsoft Access[8] database can be used by default in the current version.

Enterprise Architect allows users to extend the modeling capabilities to specific domains and notations by so-called Model Driven Generation (MDG) technologies. With MDG technologies it is possible to provide further UML profiles, patterns, templates, etc., by using UML support mechanisms like Tagged Values, Stereotypes, Profiles or Design Patterns. There are MDG technologies for several existing technologies available, e.g., for SysML, Archimate or BPMN, but any user can also provide custom ones.

Due to the fact that fUML is a subset of the UML standard and Enterprise Architect is used as UML modeling environment, it is also possible to create fUML models within Enterprise Architect. There are some limitations regarding modeling fUML within Enterprise Architect, e.g., some actions are missing from the UML standard. The specific limitations which are relevant for this work are discussed in Chapter 5.

[5] WINE - https://www.winehq.org/about/

[6] Portable Operating System Interface - http://standards.ieee.org/develop/wg/POSIX.html

[7] Firebird - http://www.firebirdsql.org/

[8] Microsoft Access - https://products.office.com/en-us/access

CHAPTER 3

C# to fUML Mapping

In this chapter a mapping from C# programming concepts to fUML modeling concepts is elaborated. This includes the mapping of C# classes to fUML classes which accounts the structure of a software system, as well as the mapping of C# method bodies to fUML activities which accounts the behavior of a software system. Conceptual differences and discrepancies between C# and fUML are identified and possible solutions are elaborated in this chapter.

The mapping of the structural features are quite straightforward, because C# and fUML have a lot of similarities as both follow the object-oriented paradigm. The mapping of the behavioral features are mainly based on the proposed Java to fUML mapping described in the fUML standard. For this mapping the fUML standard v1.1 was used [26]. All mappings in this chapters are described with the help of the example which is introduced in Chapter 3.1. Chapter 3.2 describes the mapping of structural features, while Chapter 3.3describes the mapping of behavioral features of C#.

3.1 Running Example

This section introduces an example based on which the mapping is described. Figure 3.1 depicts a class diagram which shows the relevant C# classes, their properties and associations. The class *Student* consists of the properties *Age*, *FirstName*, *LastName*, *MatNr*, and *University*. It also contains a default constructor and two other constructors, as well as an association to the *University* class. The *University* class consists of a *Name* and *Students* property, and it also offers methods for adding and removing students from the *Students* property. The interface *IAdministration* declares the *CreateUniversity* method which is implemented by the *Administration* class. The *Administration* class has an association to the *University* class and offers utility operations for querying and manipulating *Student* and *University* objects. The corresponding C# program can be found in Appendix A.

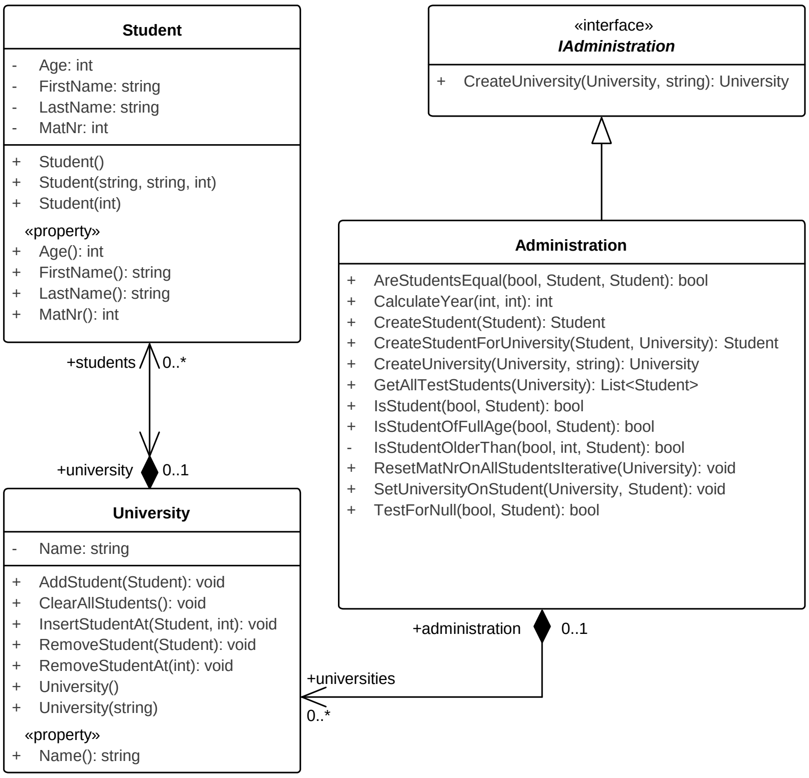

Student
- Age: int
- FirstName: string
- LastName: string
- MatNr: int
+ Student()
+ Student(string, string, int)
+ Student(int)
«property»
+ Age(): int
+ FirstName(): string
+ LastName(): string
+ MatNr(): int
«interface»
IAdministration
+ CreateUniversity(University, string): University
Administration
+ AreStudentsEqual(bool, Student, Student): bool
+ CalculateYear(int, int): int
+ CreateStudent(Student): Student
+ CreateStudentForUniversity(Student, University): Student
+ CreateUniversity(University, string): University
+ GetAllTestStudents(University): List<Student>
+ IsStudent(bool, Student): bool
+ IsStudentOfFullAge(bool, Student): bool
- IsStudentOlderThan(bool, int, Student): bool
+ ResetMatNrOnAllStudentsIterative(University): void
+ SetUniversityOnStudent(University, Student): void
+ TestForNull(bool, Student): bool
+students
0..*
+university
0..1
University
- Name: string
+ AddStudent(Student): void
+ ClearAllStudents(): void
+ InsertStudentAt(Student, int): void
+ RemoveStudent(Student): void
+ RemoveStudentAt(int): void
+ University()
+ University(string)
«property»
+ Name(): string
+administration
0..1
+universities
0..*

Figure 3.1: Class diagram of the running example.

3.2 Mapping of Structural Features

This section describes the mapping from structural C# features to fUML class modeling concepts for reverse engineering the structure of a system. Most UML modeling environments are capable of reverse engineering the structure of software systems, therefor only the most relevant mappings are described in this section. The mapping of each feature is explained by a generic C# code snippet showing the respective C# feature, a description of the feature and its mapping to fUML, an example C# code using the feature, an excerpt of the fUML metamodel relevant for mapping the C# feature to fUML, and an fUML model corresponding to the example code.

Classes

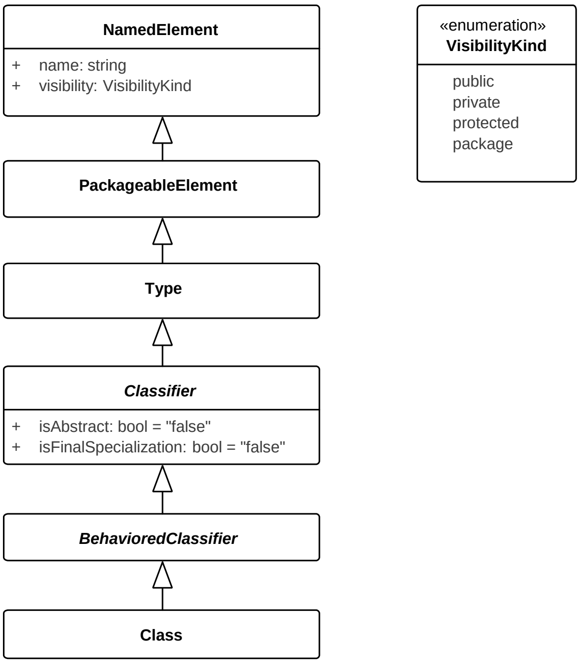
```
[public|private|protected|internal] [abstract|sealed] [partial]
class <ClassName> {...}
```

C# classes are mapped to fUML classes, which inherit the *isAbstract* and *isFinalSpecialization* property from the fUML class *Classifier*. The *isAbstract* property indicates that a class is an abstract class and the default value is *false*, and the *isFinalSpecialization* property indicates if a *Classifier* can be specialized by generalization and the default value is *false*. Abstract C# classes, indicated by the keyword *abstract*, will be mapped to fUML classes with the property *isAbstract* set to *true*. The *sealed* keyword in C# prevents a class for further specializations and this is expressed in fUML by setting the property *isFinalSpecialization* to true on fUML classes. The *name* and *visibility* properties are inherited by the fUML class *Class* from the *NamedElement* class in fUML. Using the *partial* keyword on class declarations in C# indicates that a class is split to two or more class declarations which are combined in one class with all its members at runtime. Therefor partial classes will be mapped to one fUML class containing all properties of the declared partial classes. There is no support for static classes, neither in UML nor in fUML and therefor they are omitted in this work. The class-modifier *internal* of C# is mapped to the *VisilibityKind package*.

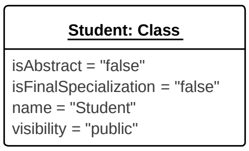
C# example:

```
public class Student {...}
```

Relevant fUML metaclasses:



fUML model:



Enumerations

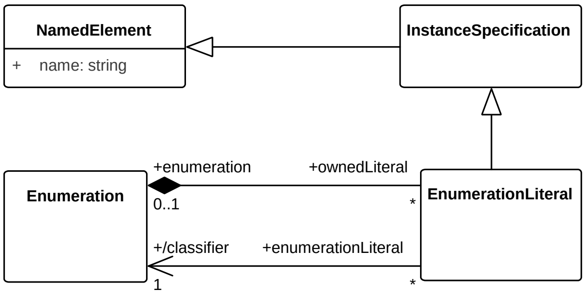
```
[public|private|protected|internal] enum <EnumerationName>
{<identifier1>, <identifier2>, ...}
```

C# enumerations are mapped to fUML enumerations which are owning an *EnumerationLiteral* collection. Each *EnumerationLiteral* is an *InstanceSpecification* which is a *NamedElement*.

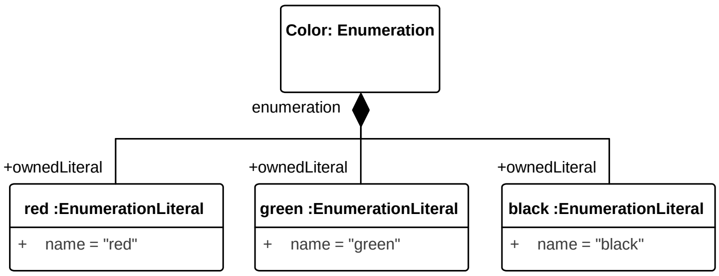
C# example:

```
public enum Color {
 red,
 green,
 black
}
```

Relevant fUML metaclasses:



fUML model:



Interfaces

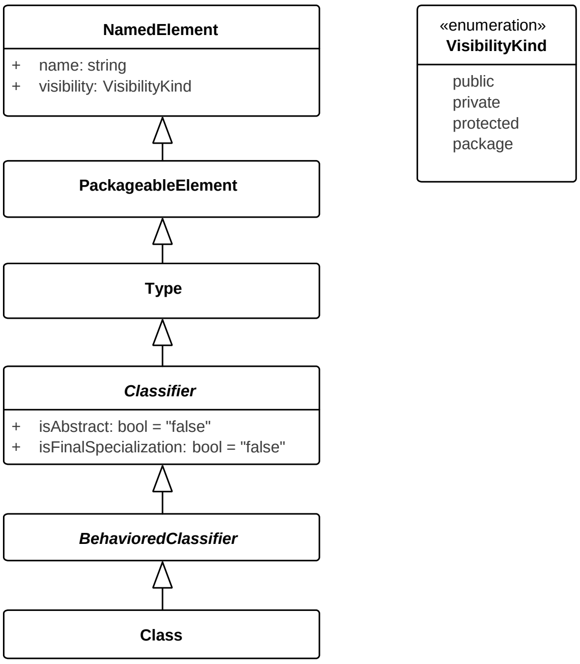
```
[public|private|protected|internal] [partial] interface
<InterfaceName> {...}
```

C# interfaces are mapped to fUML classes with the property *isAbstract* set to *true*. The fUML standard [25] proposes to achieve the effect of interfaces by using abstract classes with entirely abstract operations.

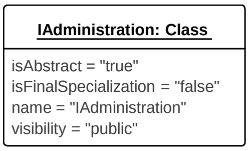
C# example:

```
public interface IAdministration {...}
```

Relevant fUML metaclasses:



fUML model:



Inheritance and interface implementation

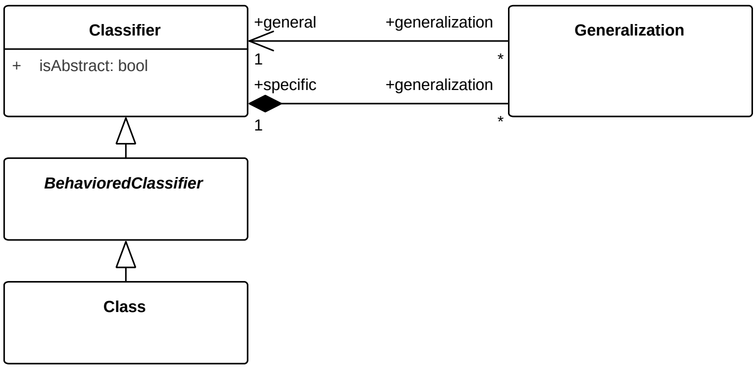
```
public class <ClassName> :  <BaseClass>, <BaseInterface1>,
<BaseInterface2> {...}
```

Inheritance is a primary concept of an object-oriented programming language. Inheritance enables extension and reusing of existing classes, as well as the modification of the behavior which is defined in other classes. An inherited class can only have one direct base class in C#. Multiple inheritance is not allowed, but a class can implement multiple interfaces. When a class implements an interface, it must provide an implementation for all members of the interface. Inheritance is mapped to the fUML *Generalization* class which is associated to one *specific* and one *general Classifier* in fUML. Remember that C# interfaces are mapped to abstract classes in fUML.

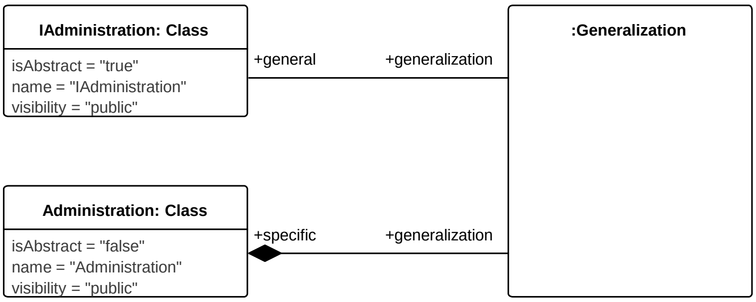
C# example:

```
public class Administration :  IAdministration {...}
```

Relevant fUML metaclasses:



fUML model:



Fields

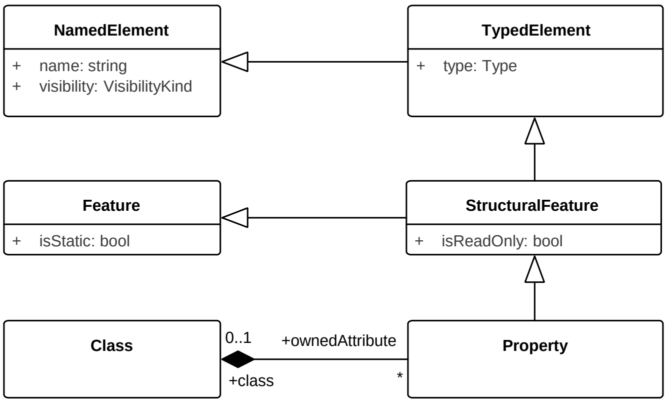
```
[public|private|protected|internal] [const] [static] <type>
<FieldName>;
```

C# fields are variables of any type declared directly in a class. Fields are mapped to fUML properties which are contained by a class. An fUML *Property* is a *StructuralFeature* which is a *Feature* itself. If a field is static, the property *isStatic* of the *Feature* is set to *true*. If a field is a constant (expressed by the *const* keyword in C#), the property *isReadOnly* of the *StructuralFeature* is set to true. Each *StructuralFeature* is also a *TypedElement* which defines a *Type* property and refers to the type of the field.

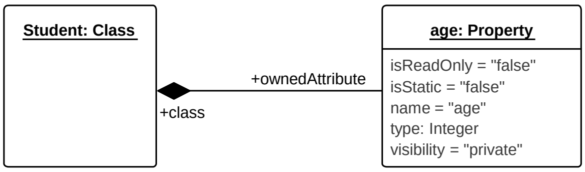
C# example:

```
public class Student {
 private int age;
}
```

Relevant fUML metaclasses:



fUML model:



Methods

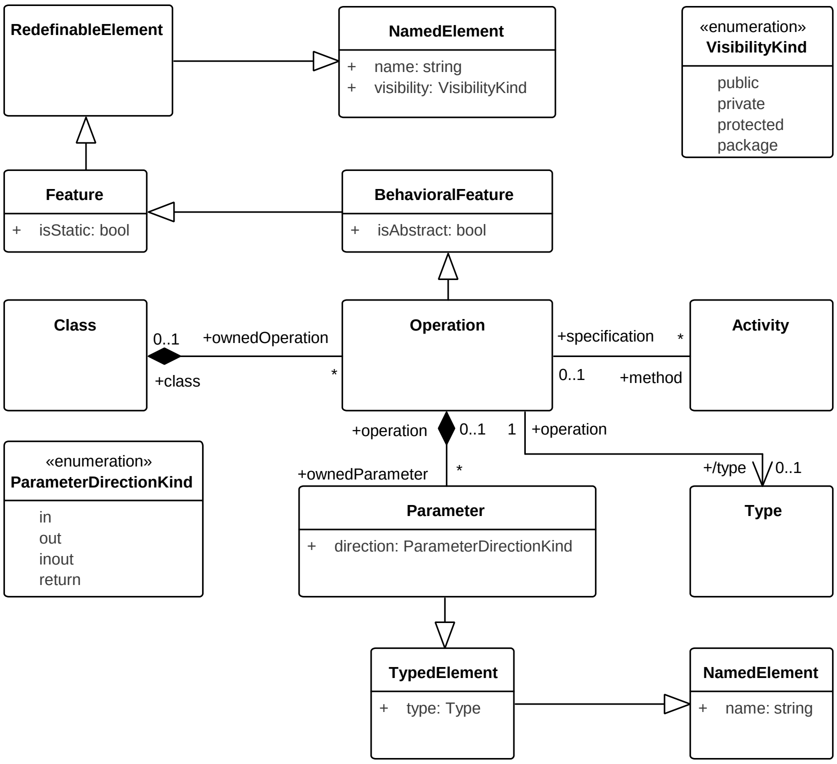
```
[public|private|protected|internal] [static|abstract]
<return-type> <MethodName> ([out|ref] <type1> <param1>,
...){...}
```

C# methods are mapped to fUML operations. Each parameter of the method becomes an fUML parameter, also the return parameter if any available. Parameters in fUML have an *ParameterDirectionKind* property which indicates the direction of the parameter. Possible directions are *in* (call-by-value), *out*, *inout* (call-by-reference), and *return* (for the return value). C# *out* parameters are mapped to fUML *inout* direction whereas C# *ref* parameters are mapped to fUML *out* direction. The return type is mapped to a parameter with direction *return*. Furthermore it can have a reference to a type, which is the type of the return parameter if it is available. Each operation is linked to an activity which defines the behavior of the method. If a method is static, the property *isStatic* of the fUML class *Feature* is set to true, and if a method is abstract, the property *isAbstract* of the fUML class *BehavioralFeature* is set to true. The mapping of the method bodies is explained in detail in Section 3.3.

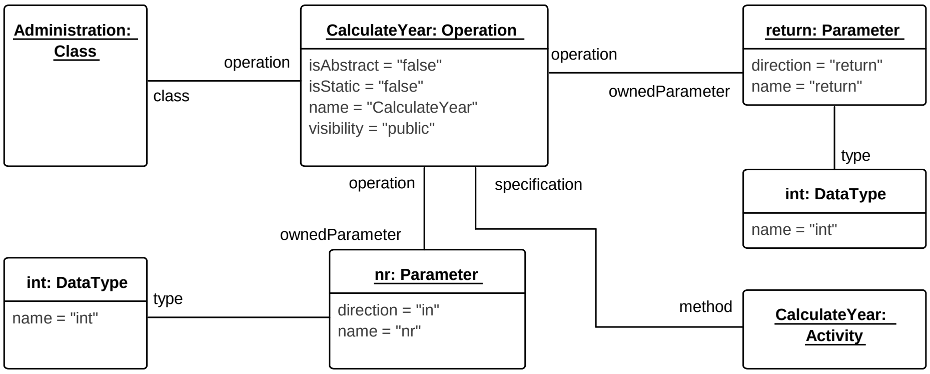
C# example:

```
public class Administration {
 public int CalculateYear(int nr) {
  ...
 }
}
```

Relevant fUML metaclasses:



fUML model:



3.3 Mapping of Behavioral Features

This section describes the mapping from C# method bodies to fUML activities for reverse engineering the behavior of a system. Each mapping is explained by a generic C# code snippet showing the respective C# feature, a description of the feature and its mapping to fUML, an example C# code using the feature, an excerpt of the fUML metamodel concerened with representing that feature in fUML, and the fUML activity diagram representing the example C# code. The mappings of behavioral features reclines on the Java-to-fUML mapping defined in Annex A of the fUML standard [26].

Method declaration

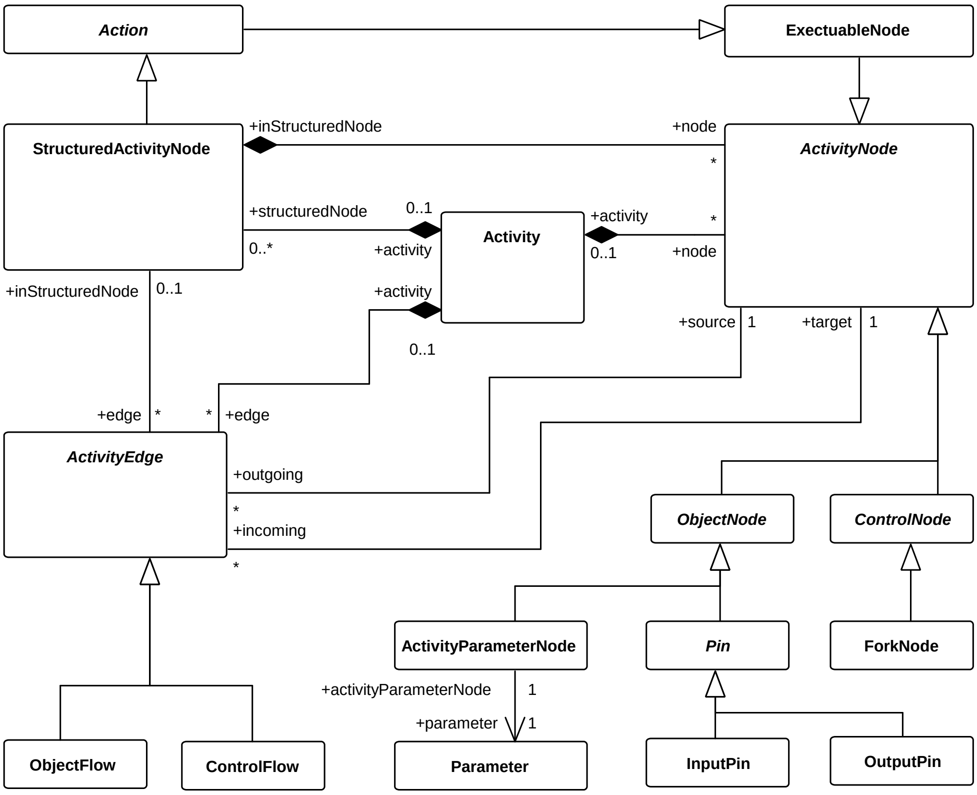
Each method body is mapped to an activity with the corresponding operation set as its specification. Input parameters are mapped to input activity parameters and corresponding activity parameter nodes. The body of the method is mapped to a structured activity node that contains the mappings of each statement of the body. If the method has a non-void return type there must be at least one return statement with an expression. The result of the mapping of the return expression is connected by an object flow to the return activity parameter node.

A fork node for each method parameter is introduced because of the possibility of multiple usage of method parameters within the method body. If a method parameter is used within the body it is connected by an object flow from the corresponding fork node to its usage in the body. If a method parameter is only used once inside the method body, the fork node can be omitted.

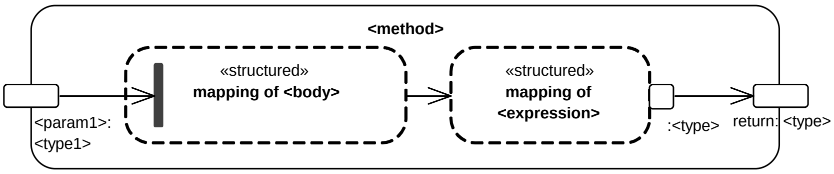
C# code:

```
public <type> <method>(<type1> <param1>) {
 <body>
 return <expression>;
}
```

Relevant fUML metaclasses:



Resulting fUML activity:



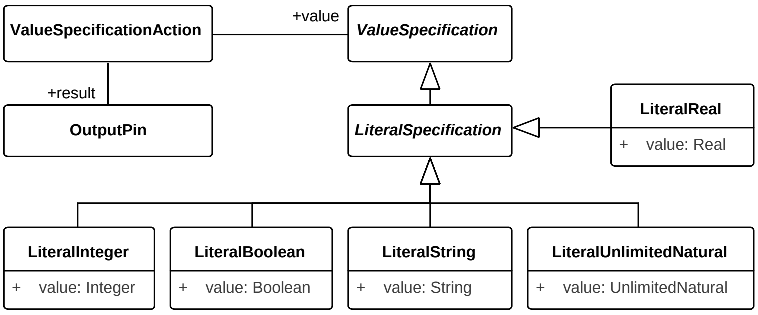
Literals

A literal is mapped to a value specification action with the corresponding literal value, and must be of type Integer, Real, Boolean, String, or UnlimitedNatural. The value specification action provides the value via its result output pin with the corresponding type of the literal value.

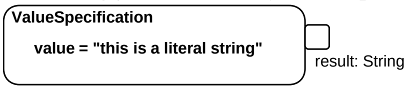
C# example:

```
„this is a literal string‟
```

Relevant fUML metaclasses:



Resulting fUML activity (excerpt):



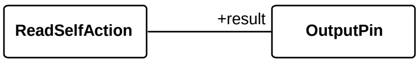
This

A use of *this* maps to a read self action which provides the context object of the containing activity on its result output pin. The context object which corresponds to the current class instance on which the activity is executed. The type of the output pin depends on the class of the current object. Also an implicit use of *this*, e.g., if the keyword *this* is omitted, maps to a read self action.

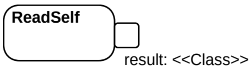
C# code:

```
this
```

Relevant fUML metaclasses:



Resulting activity part(s):



Local variable declaration

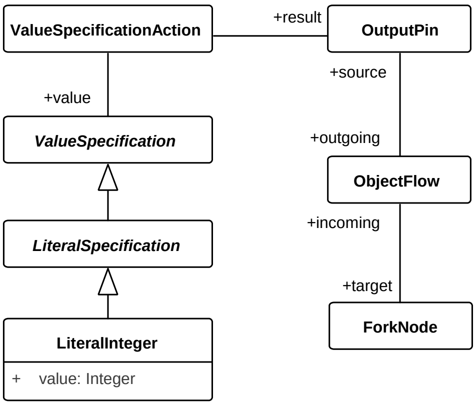
```
<type> <variableName> [= <initializationExpression>];
```

A local variable declaration statement maps to a fork node to be able to read the value multiple times. The proposed *Java to UML Activity Mapping* of the fUML standard [25] requires an initialization expression on local variable declaration in the source code. Therefor, the fork node receives an object flow from the result of the mapping of the initialization expression. To overcome this limitation, the fork node is introduced on the first variable assignment if there is a local variable declaration without initialization expression. In the following example the initialization expression maps to a value specification action. The result output pin is connected to a fork node which represents the local variable. Whenever this local variable is used later, the value is received by an object flow from this fork node.

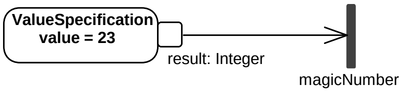
C# example:

```
int magicNumber = 23;
```

Relevant fUML metaclasses:



Resulting fUML activity (excerpt):



Instance variable assignment

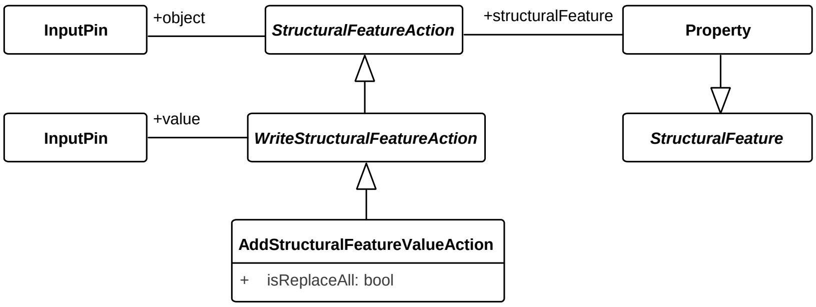
```
<object>.<field> = <value>;
```

An instance variable assignment maps to an add structural feature value action with the property *isReplaceAll* set to *true*. The *isReplaceAll* property indicates if existing values of the structural feature of the object should be removed before adding the new value. The value to be assigned must be provided to the value input pin, and the object which contains the instance variable must be provided to the object input pin of the add structural feature value action. In the following example the structural feature of the add structural feature value action is connected to the structural feature *University* of the *Student* class.

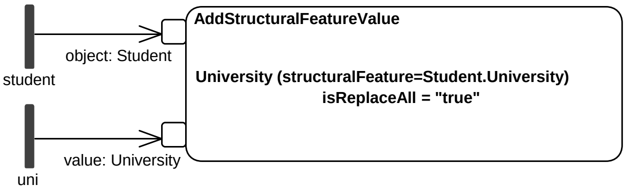
C# example:

```
student.University = uni;
```

Relevant fUML metaclasses:



Resulting fUML activity (excerpt):


AddStructuralFeatureValue
object: Student
student
University (structuralFeature=Student.University)
isReplaceAll = "true"
uni
value: University

Instance variable access

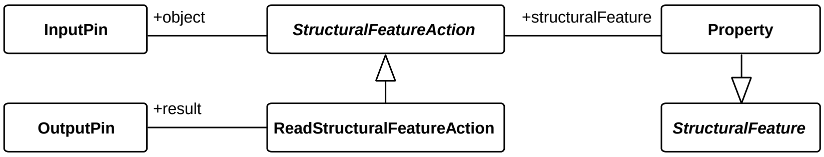
```
<object>.<field>
```

An access of an instance variable maps to a read structural feature action for the structural feature of the corresponding instance variable. The object which contains the desired instance variable must be provided to the object input pin of the read structural feature action, and the value of the structural feature will be provided on the result output pin.

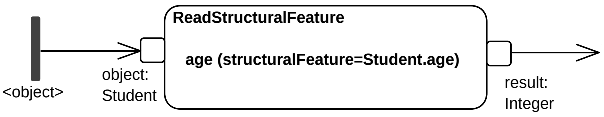
C# example:

```
this.age
```

Relevant fUML metaclasses:



Resulting fUML activity (excerpt):



Auto-implemented properties

Since version 3.0 of C#, auto-implemented properties can be declared when no additional logic is required for the property accessors. In such cases, the compiler creates a private anonymous backing field that can only be accessed through the property's get and set access operations. A property for the private field and an activity for each accessor is created in fUML for auto-implemented properties. In the example below, the activity set_Name is created for the set accessor which references the set_Name operation by its specification property, and the activity get_Name is created for the get accessor which references the get_Name operation by its specification property. The Name property of the class is referenced by the structural feature of the add structural feature value action and the read structural feature action in the activity set_Name and get_Name, respectively.

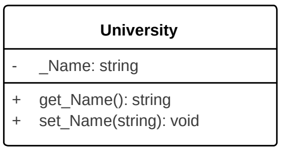
C# example:

```
public class University {
 public string Name{ get; set; }
}
```

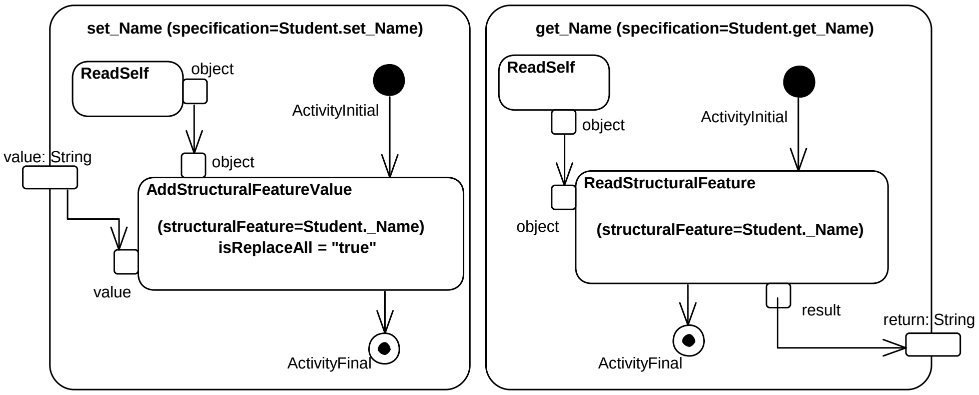
Equivalent code which will be generated by C# internally:

```
public class University {
 private string _Name;
 public string get_Name { return this._Name; }
 public void set_Name { this._Name=value; }
}
```

Resulting fUML class:



Resulting fUML activities:



Method call

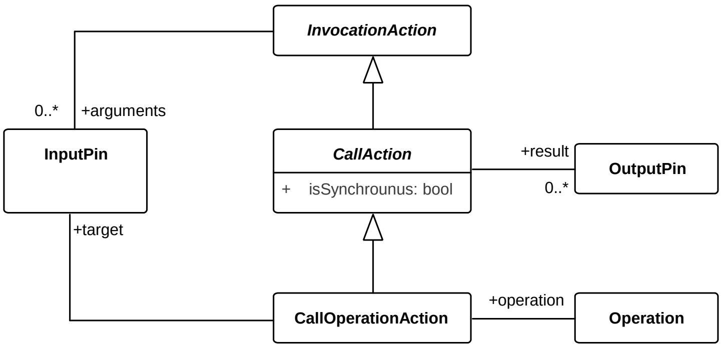
```
[retvar =] <object>.<operation>(<param1>, <param2>,...);
```

A method call maps to a call operation action which references the called operation. For each argument of the called method an input pin is created. If the method returns a value, the call operation action defines also one output pin. The property *isSynchronous* of the call action determines if the operation has to be executed synchronously or asynchronously and is set to *true* on method calls. The example method IsStudentOlderThan of the class Administration demands a Student object and an integer value as input, and returns a value of type bool. Another important point is that accessing properties by getter/setter operations which looks like an instance variable use/assignment (e.g. `student.FirstName`), is also mapped to call operation actions. This includes auto-implemented properties as well as user defined properties.

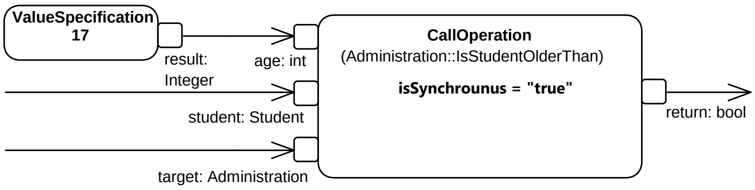
C# example:

```
IsStudentOlderThan(student, 17);
```

Relevant fUML metaclasses:



Resulting fUML activity (excerpt):



Object creation

```
new <ClassName>(<param1>);
```

An object creation maps to a create object action which provides the created object on its *result* output pin. fUML does not allow class properties to have default values. The default values will be mapped in the so-called classified behavior of a class to bypass this limitation. Each mapped class gets an activity *_ClassifiedBehavior* which handles the mapping for default values of fields. The mapped class and the activity are connected by a reference. A start object behavior action which executes the classified behavior will be called after each create object action. Each class in fUML may have at most one classified behavior. The *isSynchrounus* property must be set to false due to a constraint in fUML. To overcome another limitation of fUML, which defines that a class can only have a default constructor without parameters, for each available constructor an operation is created and this operation is called after the start object behavior action is executed if an other constructor than the default one is used. Control flows must be used in this example to obtain the right order of the execution.

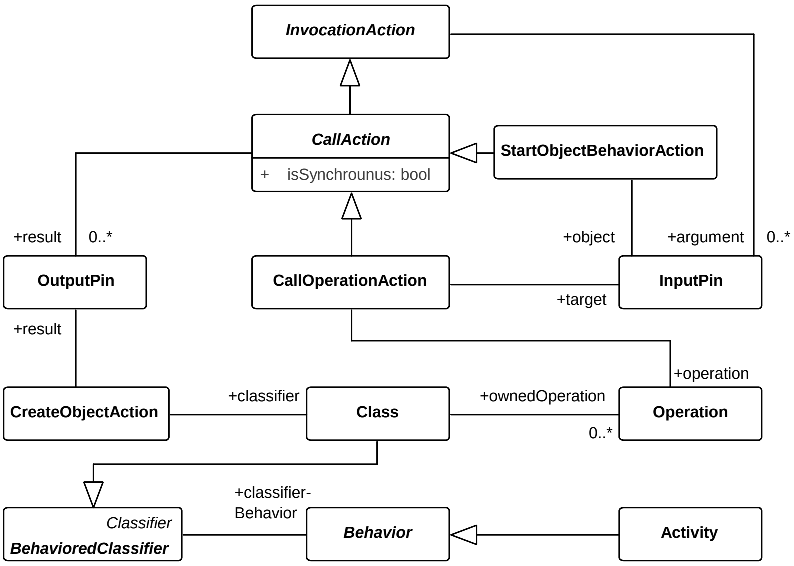
C# example:

```
public Student CreateStudent() {
 return new Student(123456);
}
```

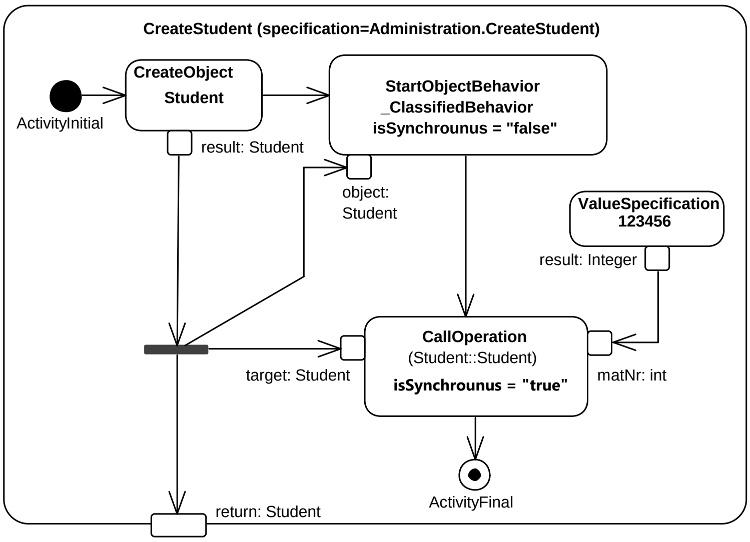
Relevant C# code in the Student class:

```
public class Student {
 ...
 public int MatNr { get; set; }
 private int age = 18;
 ...
 public Student(int matNr)
 {
  MatNr = matNr;
 }
 ...
}
```

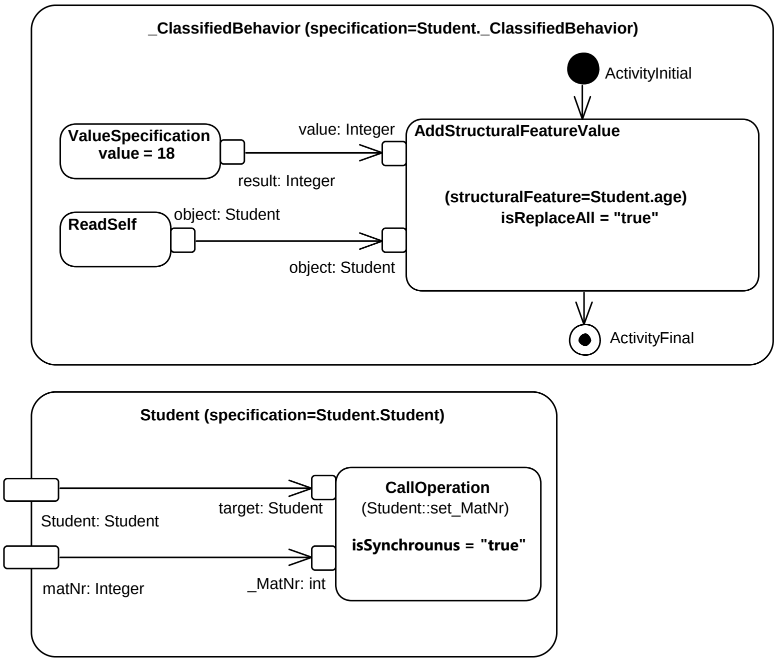
Relevant fUML metaclasses:



Resulting fUML activity:



Relevant fUML activities of the Student class:



Equality comparison

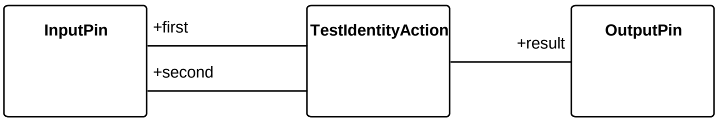
```
<arg1> == <arg2>
```

It can be distinguish between two different types of equality comparisons in C#. The first one is value equality known as equivalence, which compares two values contained by a variable. The second one is reference equality or identity, which determines where two variables refer to the same underlying object in memory. In fUML both types are mapped to a test identity action which gets two arguments on its input pins and provides a result output pin of type Boolean. fUML as well as C# handles the two different types of equality comparison in the same way. On primitive types it will be compared for value equivalence and on reference types for reference equality. Testing for inequality (in C# with the operator !=) is also mapped to a test identity action, but the result has to be negated (by using a call behavior action on the *boolNeg* primitive behavior which is described in the end of this section).

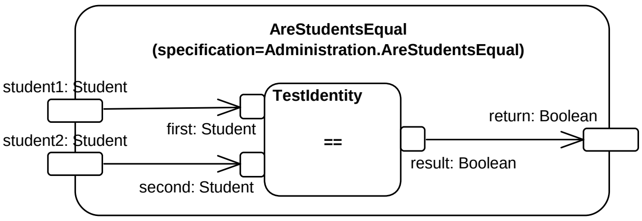
C# example:

```
public bool AreStudentsEqual(Student student1, Student
student2) {
 return student1 == student2;
}
```

Relevant fUML metaclasses:



Resulting fUML activity:



Testing for null

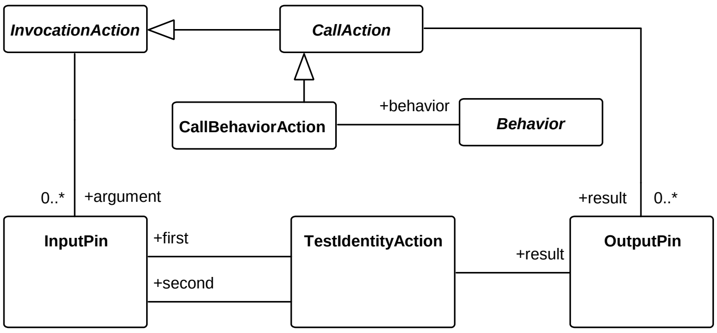
```
<object> == null
```

Testing for null is mapped to a test whether the argument of the test has a list size of zero, because the concept of *null* is not present in fUML. The list length of the argument is obtained by a call behavior action on the *ListSize* behavior of the primitive behaviors described in the end of this section.

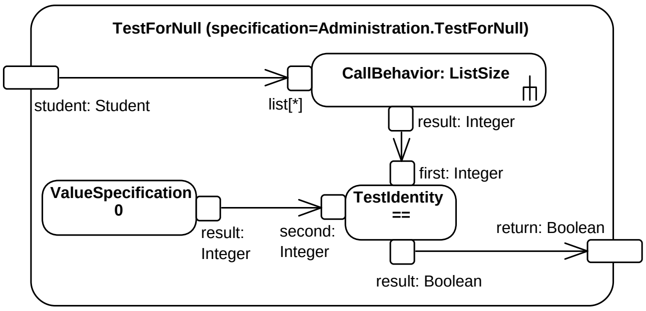
C# example:

```
public bool TestForNull(Student student) {
 return student == null;
}
```

Relevant fUML metaclasses:



Resulting fUML activity:



Operator expressions

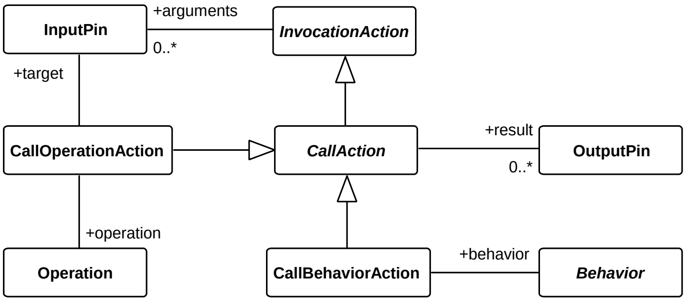
```
<arg1> <operator> <arg2>
```

Operator expressions are mapped to call behavior actions calling the primitive behavior corresponding to the operator and the type of the arguments. Unary operators (e.g., negation) are mapped similarly, except that they have only one argument and the call behavior action has only one input pin. The following example also demonstrates an operation call for accessing the Age property of the student (the getter of the property is called).

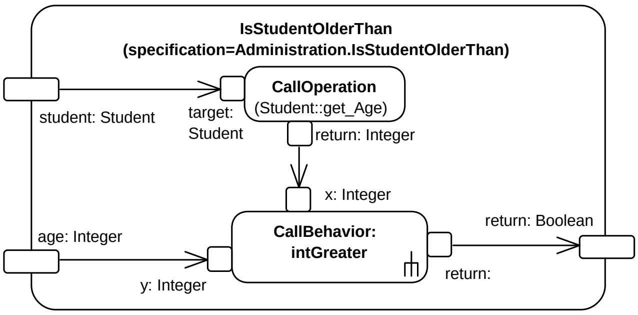
C# example:

```
public bool IsStudentOlderThan(Student student, int age) {
 return student.Age > age;
}
```

Relevant fUML metaclasses:



Resulting fUML activity:



Type checking

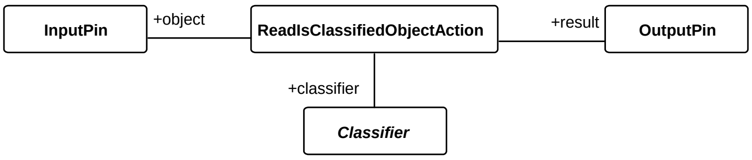
```
<object> is <Class>
```

The test whether an object is an instance of a certain class by using the C# operator *is*, is mapped to a read is classified object action. The object to be tested is passed to the object input pin, and the class is set as classifier of the action.

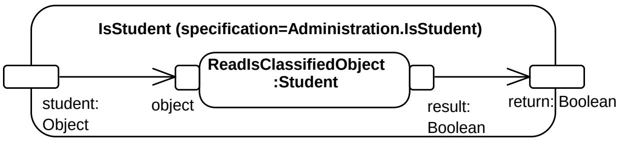
C# example:

```
public bool IsStudent(object student) {
 return student is Student;
}
```

Relevant fUML metaclasses:



Resulting fUML activity:



If-else statements

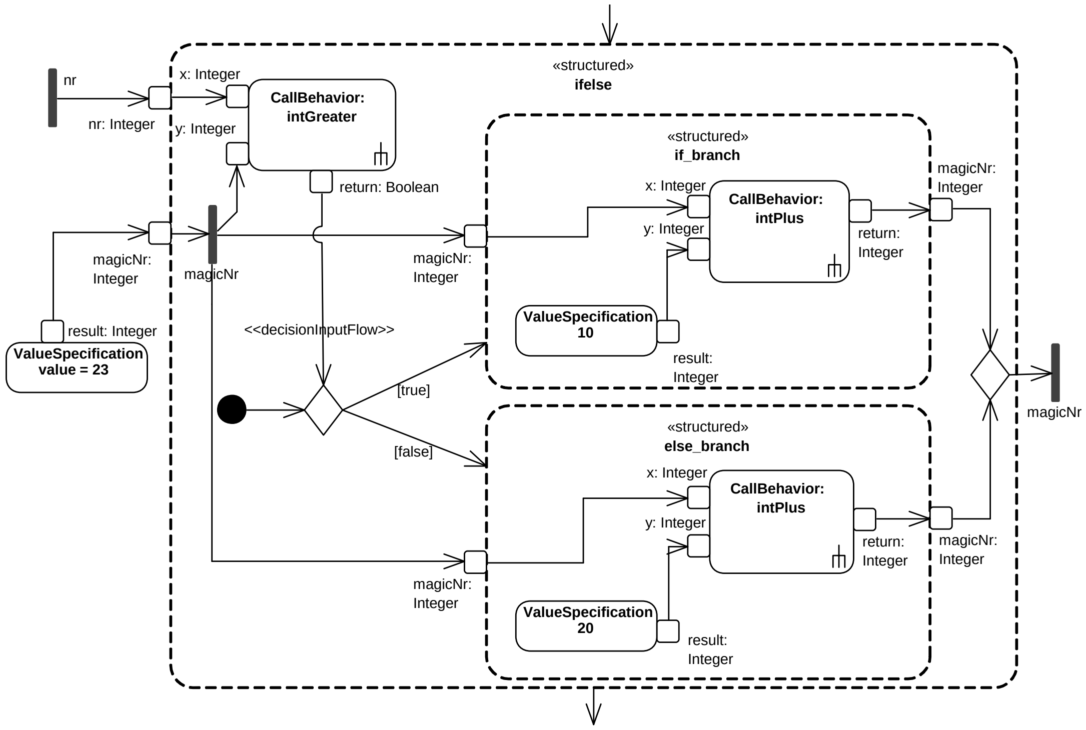
```
if (<testexpression>){
 <body1>
}
else {
 <body2>
}
```

An if-else statement is mapped to a structured activity node with an initial node connected to a decision node by a control flow. The decision node gets a decision input flow from the mapping of the test expression and has two outgoing control flows with guards. One control flow is connected to a structured activity node for the if branch with *true* as guard, the other control flow is connected to a structured activity node for the else branch with *false* as guard. The statements within each branch are mapped separately within each structured activity node. The structured activity for each branch has input and output pins for each variable access and assignment done in the branch. The output pins for each variable assignment in each branch are connected to a merge node, which has an object flow to a fork node, which acts as the source for all variable accesses subsequently defined after the if-else statement. If a variable is only assigned in one branch, the structured activity node representing the other branch must also provide an input and output pin for the variable which are directly connected by an object flow.
If there is no else branch, and a variable is assigned in the if branch then there must also be an structured activity node for the else branch with all input pins connected to output pins (and of course the merge node to provide the variable for subsequent statements after the if statement). The structured activity for the else branch can be replaced by an activity final node if there are no variable assignments within the if branch. A nested if-else-if statement leads to nested structured activity nodes, e.g., the mapping of the second if statement is nested within the structured activity node from the first else mapping, etc.

C# code:

```
int magicNumber = 23;
if (nr > magicNumber)
{
 magicNumber = magicNumber + 10;
}
else
{
 magicNumber = magicNumber + 20;
}
```

Resulting fUML activity (excerpt):



Do-while loops

```
do {
 <body>
} while (<testexpression>)
```

Do-while loops are mapped to a structured activity node with looping control structure outside which is realized by decision and merge nodes. A merge node (on the left side in the example) serves as entry point. It is connected by a control flow to the structured activity node. A decision output pin of the structured activity node, which receives its value from the test expression, serves as decision input flow for a decision node, which decides whether the loop has to be repeated (on the right bottom side in the example).

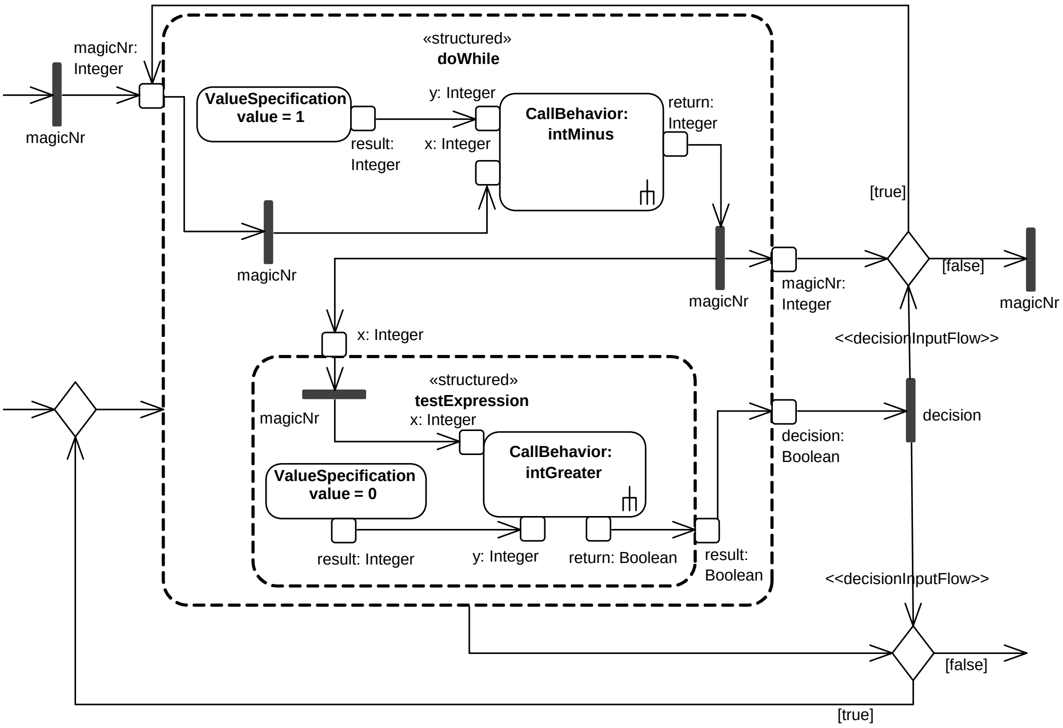
Each statement within the loop is mapped separately within the structured activity node. The structured activity node has input and output pins for each referenced variable within the loop. The input pins for each variable is connected by an object flow from the mapping for the variable from before the loop. Each input pin is connected by an object flow to a fork node, which is used as source for all other mappings inside the loop. If a variable is changed inside the loop the new value is provided to the corresponding output pin by an object flow from the fork node which represents the variable.

The test expression is mapped to a structured activity node which has input pins for each used variable in the test expression. The input pins are connected from fork nodes, which represents the variable. The result of the test expression is provided by an object flow to an output pin of the outer structured activity node. The output pin for each variable used within the loop is connected to a decision node by an object flow which serves as input if the loop has to be repeated, or it provides the new value as fork node for subsequent use after the loop (the right upper decision node in the example controls the object flow for the variable).

C# code:

```
do
{
 magicNumber = magicNumber - 1;
} while (magicNumber > 0)
```

Resulting fUML activity (excerpt):



While loops

```
while(<testexpression>) {
 <body>
}
```

While loops are mapped to do-while loops which are nested in an if statement.

C# code:

```
while (magicNumber > 0)
{
 magicNumber = magicNumber - 1;
}
```

Equivalent code in C# that is mapped to fUML:

```
if (magicNumber > 0)
{
 do
 {
  magicNumber = magicNumber - 1;
 } while (magicNumber > 0)
}
```

For loops

The fUML standard proposes to map for loops to expansion regions. Expansion regions are however only working when iterating over lists. To overcome this limitation, for loops are treated like while loops, also if the loop is used for iterating over list items.

C# code:

```
for (int i = 0; i < uni.Students.Count; i++)
{
 uni.Students[i].MatNr = 0;
}
```

Equivalent code in C# that is mapped to fUML:

```
int i = 0;
while (i < uni.Students.Count)
{
 uni.Students[i].MatNr = 0;
 i = i + 1;
}
```

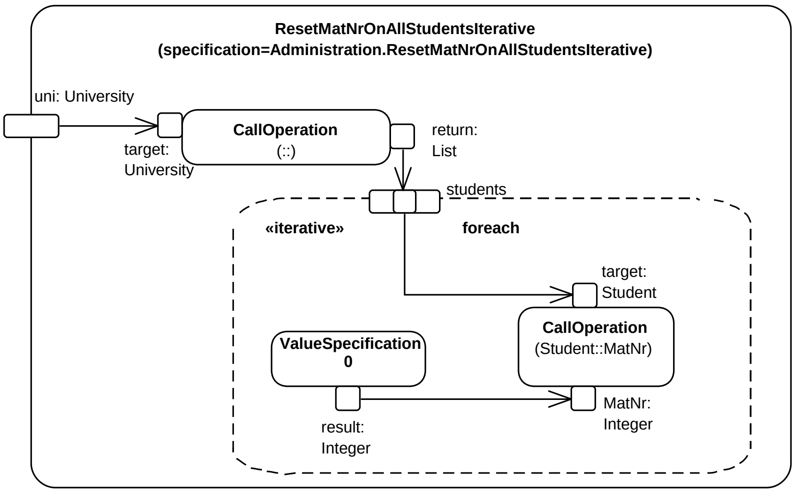
Foreach loops

Foreach loops are mapped to expansion regions. The list of iterated values is passed by an object flow to an input expansion node, which serves as input node within the loop body. Each statement within the loop body is mapped seperately. If any other variables are used inside the loop, then they are provided into the expansion region via input pins.

C# code:

```
public void ResetMatNrOnAllStudentsIterative(University uni)
{
 foreach (Student student in uni.Students)
 {
  student.MatNr = 0;
 }
}
```

Resulting fUML activity:



Switch-case statements

Switch-case statements are mapped to nested if-else statements. For each case a (nested) if statement is created, the last else branch behaves as the default case (if any available). In C# it is not allowed to fall through cases. This means each case must have a *break* at the end. Therefor it is possible to map switch-case statements to nested if-else statements.

C# code:

```
switch (nr)
{
 case 1:
  magicNumber = 14;
  break;
 case 10:
  magicNumber = 24;
  break;
 default:
  magicNumber = magicNumber + 1;
  break;
}
```

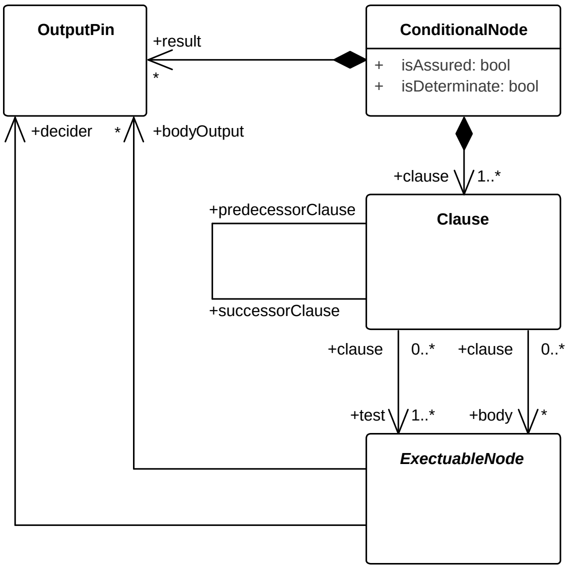
Equivalent code in C# that is mapped to fUML:

```
if (nr == 1)
```

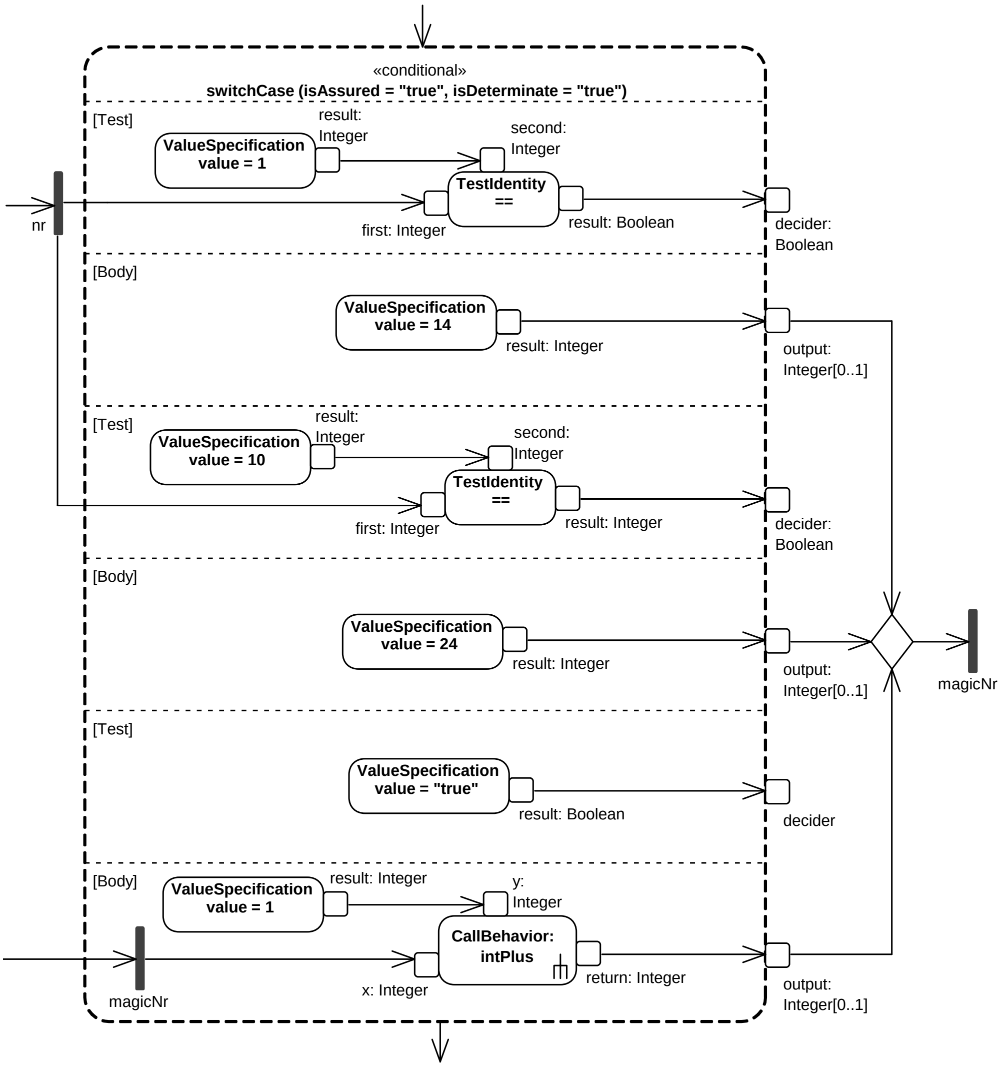
```
{
 magicNumber = 14;
}
else
{
 if (nr == 10)
 {
  magicNumber = 24;
 }
 else
 {
  magicNumber = magicNumber + 1;
 }
}
```

Alternative mapping:
Switch-case statements can also be mapped to conditional nodes alternatively. A conditional node consists of one or more clauses wherein each clause can have successor and predecessor clauses. Each clause consists of a test section and a body section. All test sections are evaluated and if one test section evaluates to true (the value *true* is assigned to the decider output pin of the test section), the corresponding body section is executed. The *isAssured* property of the conditional node asserts that at least one test section evaluates to *true*, the property *isDeterminate* asserts that at most one test section evaluates to *true*. Because it is not allowed to fall through cases in C#, the property *isDeterminate* is always set to true. It is possible to model a default clause by setting this clause as last clause and returning always true in its test section. Any output value which is created in the body section of a clause and provided by an output pin has to be provided on each clause to prevent an undefined value which could be used outside the conditional node.

Relevant fUML metaclasses:



Resulting fUML activity (excerpt):



Primitive Datatypes and Behaviors

The Foundational Model Library for fUML contains user-level model elements which can be referenced in an fUML model. It defines the primitive types *Boolean*, *Integer*, *Real*, *String*, and *UnlimitedNatural*, as well as a set of primitive behaviors which are operating on the primitive data types. It also defines two behaviors for List types. These primitive datatypes and behaviors have to be provided by an fUML modeling and execution environment to make them available at design-time and execution/run-time of fUML models, because they can be referenced from any fUML model.

Table 3.1, 3.2, 3.3, 3.4, and 3.5 depict all functions available for the primitive types *Boolean*, *Integer*, *Real*, *String*, and *UnlimitedNatural*, respectively. They also show the C# equivalent of each function, except for *UnlimitedNatural* because there is no adequate datatype available in C#.

An activity is created for each function of the different primitive datatypes, because Enterprise Architect does not allow to define primitive behaviors. Such activity serves as placeholder to be able to reference these primitive behaviors in the mappings of the behavioral features described above by using a call behavior action. An fUML execution environment must take this into account to be able to map the call of the call behavior action to the right primitive behavior defined in the fUML execution environment.

Table 3.1: fUML Boolean functions.

Function Signature	**Description**	**C# equivalent**
Or(x: Boolean, y: Boolean): Boolean	Returns *true* if either x or y is true.	z = x || y; (z, x, and y of type *bool*)
Xor(x: Boolean, y: Boolean): Boolean	Returns *true* if either x or y is true, but not both.	z = x ˆ y; (z, x, and y of type *bool*))
And(x: Boolean, y: Boolean): Boolean	Returns *true* if x and y are true.	z = x && y; (z, x, and y of type *bool*)
Not(x: Boolean): Boolean	Returns *true* if x is false and vice versa.	z = !x; (z, and x of type *bool*)
Implies(x: Boolean, y: Boolean): Boolean	Returns true if x is false, or if x and y is true.	z = !x || y; (z, x, and y of type *bool*)
ToBoolean(x: String): Boolean[0..1]	Converts x to a *Boolean* if possible.	z = bool.Parse(x); *or* z = Convert.ToBoolean(x); (x of type *string*, z of type *bool*)
ToString(x: Boolean): String	Converts x to a *String* value.	x.ToString(); (x of type *bool*)

Table 3.2: fUML Integer functions.

Function Signature	**Description**	**C# equivalent**
Neg(x: Integer): Integer	Returns the negative value of x.	x = -y; (x and y of type *int*)
+(x: Integer, y: Integer): Integer	Returns the value of the addition of x and y.	z = x + y; (z, x, and y of type *int*)
-(x: Integer, y: Integer): Integer	Returns the value of the subtraction of x and y.	z = x - y; (z, x, and y of type *int*)
*(x: Integer, y: Integer): Integer	Returns the value of the multiplication of x and y.	z = x * y; (z, x, and y of type *int*)
/(x: Integer, y: Integer): Real[0..1]	Returns the value of the division of x by y.	z = (*double*)x / y; (z of type double, x and y of type int, x is casted to *double* to get decimal value).

Abs(x: Integer): Integer	Returns the absolute value of x.	z = Math.Abs(x); (z and x of type int, usage of static *Abs* method on static *Math* class.
Div(x: Integer, y: Integer): Integer[0..1]	Returns the number of times that y fits completely within x.	z = x / y; (z, x, and y of type *int*)
Mod(x: Integer, y: Integer): Integer	Returns the result of x modulo y.	z = x % y; (z, x, and y of type *int*)
Max(x: Integer, y: Integer): Integer	Returns the maximum of x and y.	z = Math.Max(x, y); (z, x, and y of type *int*, usage of static *Max* method on static *Math* class.)
Min(x: Integer, y: Integer): Integer	Returns the minimum of x and y.	z = Math.Min(x, y); (z, x, and y of type *int*, usage of static *Min* method on static *Math* class.)
<(x: Integer, y: Integer): Boolean	Returns *true* if x is less than y.	z = x < y; (z, x, and y of type *int*)
>(x: Integer, y: Integer): Boolean	Returns *true* if x is greather than y.	z = x > y; (z, x, and y of type *int*)
<=(x: Integer, y: Integer): Boolean	Returns *true* if x is less than or equal to y.	z = x <= y; (z, x, and y of type *int*)
>=(x: Integer, y: Integer): Boolean	Returns *true* if x is greather than or equal to y.	z = x >= y; (z, x, and y of type *int*)
ToString(x: Integer): String	Converts x to a *String* value.	z = x.ToString(); (z of type *string*, x of type *int*)
ToUnlimitedNatural(x: Integer): UnlimitedNatural[0..1]	Converts x to an *UnlimitedNatural* value.	No C# equivalent available.
ToInteger(x: String): Integer[0..1]	Converts x to an *Integer* value if possible.	z = int.Parse(x); or z = Convert.ToInt32(x); (z of type *int*, x of type *string*)

Table 3.3: fUML Real functions.

Function Signature	**Description**	**C# equivalent**
Neg(x: Real): Real	Returns the negative value of x.	x = -y; (x and y of type *double*)
+(x: Real, y: Real): Real	Returns the value of the addition of x and y.	z = x + y; (z, x, and y of type *double*)
-(x: Real, y: Real): Real	Returns the value of the subtraction of x and y.	z = x - y; (z, x, and y of type *double*)

*(x: Real, y: Real): Real	Returns the value of the multiplication of x and y.	z = x * y; (z, x, and y of type *double*)
/(x: Real, y: Real): Real[0..1]	Returns the value of the division of x by y.	z = x / y; (z, x, and y of type *double*)
Abs(x: Real): Real	Returns the absolute value of x.	z = Math.Abs(x); (z and x of type *double*)
Floor(x: Real): Integer[0..1]	Returns the largest integer that is less than or equal to x.	z = (int)Math.Floor(x); (z of type *int*, x of type *double*, usage of the static *Floor* method on static *Math* class)
Round(x: Real): Integer[0..1]	Returns the (larger) integer that is closest to x.	z = (int)Math.Round(x); (z of type *int*, x of type *double*), usage of the static *Round* method on static *Math* class.
Max(x: Real, y: Real): Real	Returns the maximum of x and y.	z = Math.Max(x, y); (z, x and y of type *double*, usage of the static *Max* method on static *Math* class)
Min(x: Real, y: Real): Real	Returns the minimum of x and y.	z = Math.Min(x, y); (z, x and y of type *double*, usage of the static *Min* method on static *Math* class)
<(x: Real, y: Real): Boolean	Returns *true* if x is less than y.	z = x < y; (z, x, and y of type *double*)
>(x: Real, y: Real): Boolean	Returns *true* if x is greather than y.	z = x > y; (z, x, and y of type *double*)
<=(x: Real, y: Real): Boolean	Returns *true* if x is less than or equal to y.	z = x <= y; (z, x, and y of type *double*)
>=(x: Real, y: Real): Boolean	Returns *true* if x is greather than or equal to y.	z = x >= y; (z, x, and y of type *double*
ToString(x: Real): String	Converts x to a *String* value.	z = x.ToString(); (z of type *string*, x of type *double*)
ToReal(x: String): Real[0..1]	Converts x to an *Real* value if possible.	z = double.Parse(x); or z = Convert.ToDouble(x); (z of type *double*, x of type *string*)
ToInteger(x: Real): Integer	Converts x to an *Integer* value.	z = Convert.ToInt32(x); (z of type *int*, x of type *double*)

Table 3.4: fUML String functions.

Function Signature	**Description**	**C# equivalent**

Concat(x: String, y: String): String	Returns the concatenation of x and y.	z = x + y; (z, x, and y of type *string*)
Size(x: String): Integer	Returns the number of characters in x.	z = x.Length; (z of type *int*, x of type *string*, usage of the *Length* property on the string)
Substring(x: String, lower: Integer, upper: Integer): String[0..1]	Returns the substring of x starting at character number lower, up to and including character number upper.	z = x.Substring(lower-1, upper-lower); (z and x of type *string*, lower and upper of type *int*)

Table 3.5: fUML UnlimitedNatural functions.

Function Signature	**Description**
Max(x: UnlimitedNatural, y: UnlimitedNatural): UnlimitedNatural	Returns the maximum of x and y.
Min(x: UnlimitedNatural, y: UnlimitedNatural): UnlimitedNatural	Returns the minimum of x and y.
<(x: UnlimitedNatural, y: UnlimitedNatural): Boolean	Returns *true* if x is less than y.
>(x: UnlimitedNatural, y: UnlimitedNatural): Boolean	Returns *true* if x is greather than y.
<=(x: UnlimitedNatural, y: UnlimitedNatural): Boolean	Returns *true* if x is less than or equal to y.
>=(x: UnlimitedNatural, y: UnlimitedNatural): Boolean	Returns *true* if x is greather than or equal to y.
ToString(x: UnlimitedNatural): String	Converts x to a *String* value.
ToUnlimitedNatural(x: String): UnlimitedNatural[0..1]	Converts x to an *UnlimitedNatural* value if possible.
ToInteger(x: UnlimitedNatural): Integer[0..1]	Converts x to an *Integer* value.

Mapping of List Types

Lists declared in C# in the form *List<base type>* are representing a list of values of the type <base type>. These types are mapped e.g., to <base type> properties or parameters with multiplicity [*]. fUML restricts calls to *Clear* (clear structural feature action), *Add* (add structural feature value action) and *Remove* (remove structural feature action) methods to be used on structural features only. With this restriction it is not be possible to define a local list within a method and add or remove list items. A class *List* is introduced to overcome this limitation, which holds the list of values, and offers operations for manipulating these items. Figure 3.2 shows the class *List* with its operations.

Table 3.6 depicts the mapping from C# operations which can be called on list types to the *List* class operations shown in Figure 3.2.

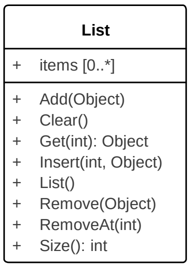


Figure 3.2: The class *List*.

Table 3.6: Mapping between C# list functions and operations defined by the fUML class *List*.

C# list function	**fUML *List* class operation**
Add(item)	Add(item)
Clear()	Clear()
list[int]	Get(Integer)
GetAt(int)	Get(Integer)
Insert(int, object)	Insert(int, object)
Remove(object)	Remove(object)
RemoveAt(int)	RemoveAt(int)
Count	Size(int)

In the following, the behavior of the operations is described:

- Adding items to a list
  Figure 3.3 shows the activity for the *Add* operation of the introduced fUML class *List*. It maps the C# list add method to an add structural feature value action with the property *isReplaceAll* set to *false*. A read self action puts the list object to the *object* input pin of the add structural feature value action and the *value* input pin of the add structural feature value action gets the value from the input activity parameter node *item*. The value specification action provides the unlimited natural value *-1* to the *insertAt* input pin of the add structural feature value action to add the item at the end of the list.

- Clear all items of a list
  Clearing all items of a list maps to a clear structural feature action on the *List* object. The object input pin of the clear structural feature action is provided with the list object by the read self action, which is shown in Figure 3.4.

- Retrieving an item of a list
  A call to the C# *List Get* operation maps to an fUML call behavior action calling the primitive behavior *ListGet* of the fUML foundational model library. The result pin is

connected to the result pin of the activity. The list input pin of the call behavior action gets its input from a read structural feature action which returns the structural feature *items* of the *List* object. Figure 3.5 depicts the resulting activity for getting an item of a list.

- Inserting an item into a list
  Inserting an item into a list is similar to adding an item to a list, but instead of specifying a value specification action for the insertAt input pin of the add structural feature value action, the input value is provided through the activity parameter node *index*. It has to be considered, that the index of the C# *Insert* operation is a zero-based index. This means, that the first element of a C# list is at index 0, as against the first element of an fUML list starts at index 1. Hence, the value of the index is increased by one. The activity for inserting an item to a list can be seen in Figure 3.6.

- Removing an item from a list
  Removing an item from a list by passing the item to be removed as parameter, maps to an fUML remove structural feature value action which receives the item to be removed (input pin value) from the activity parameter node item. Figure 3.7 shows the resulting activity for removing an item of a list.

- Removing an item from a list by an index
  Removing an item from a list by using an index maps also to a remove structural feature value action. Thereby, the item to be removed first has to be retrieved from the list using a call behavior action for the primitive behavior *ListGet* providing the index as input. The item to be removed is sent to the value input pin of the remove structural feature value action. The removeAt input pin gets its value from the activity parameter node index. The activity for removing an item of a list by an index is shown in Figure 3.8.

- Retrieving the size of a list
  An access to the list count property in C# maps to a call behavior action for the primitive behavior *ListSize* of the foundational model library described in Section 3.3. The return pin is connected to the return pin of the activity. The list input pin gets its value from the read structural feature value action retrieving the list items. The activity for counting the items of a list is depicted in Figure 3.9.

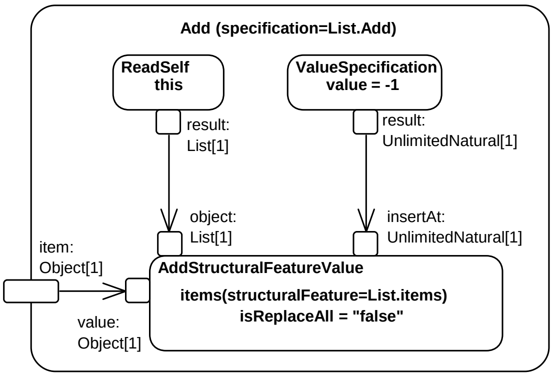


Figure 3.3: fUML activity for adding an item to a list.

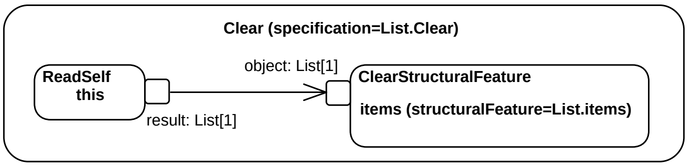

Clear (specification=List.Clear)
ReadSelf
this
object: List[1]
result: List[1]
ClearStructuralFeature
items (structuralFeature=List.items)

Figure 3.4: fUML activity for clearing all items of a list.

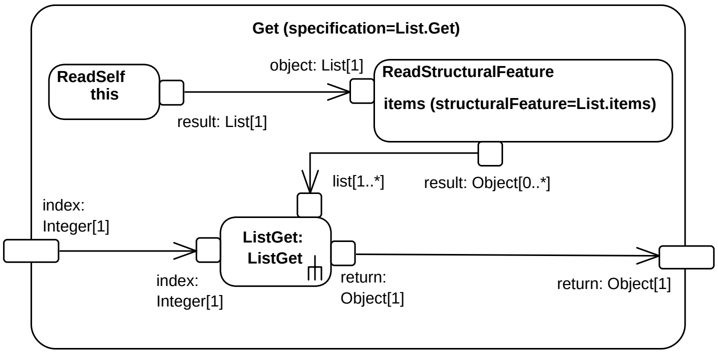


Figure 3.5: fUML activity for retrieving an item from a list.

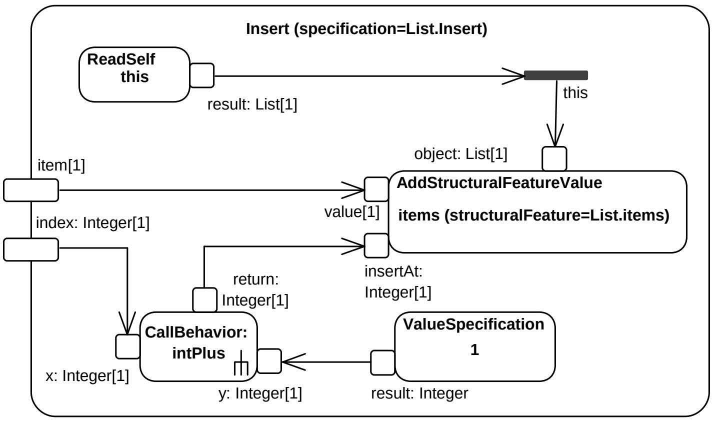


Figure 3.6: fUML activity for inserting an item into a list.

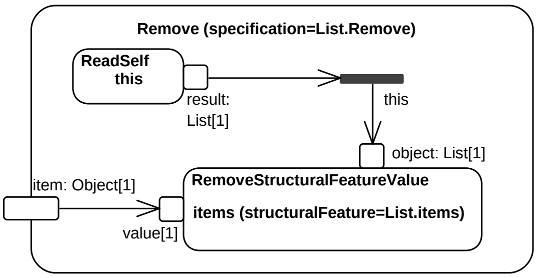

Remove (specification=List.Remove)
ReadSelf
this
result:
List[1]
this
object: List[1]
item: Object[1]
RemoveStructuralFeatureValue
items (structuralFeature=List.items)
value[1]

Figure 3.7: fUML activity for removing an item from a list.

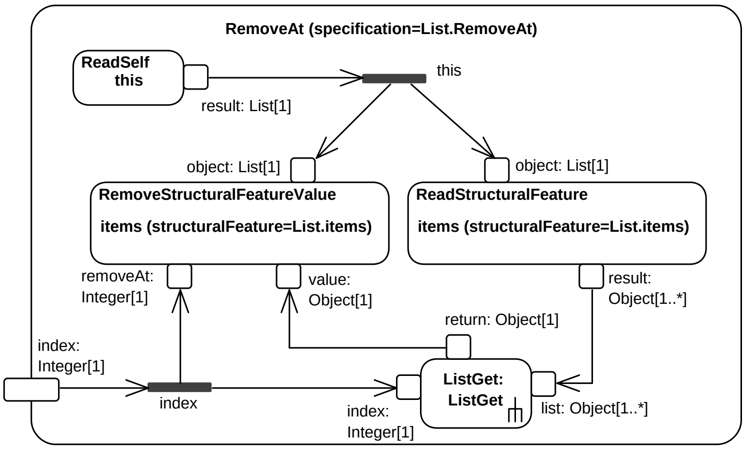


Figure 3.8: fUML activity for removing an item from a list by an index.

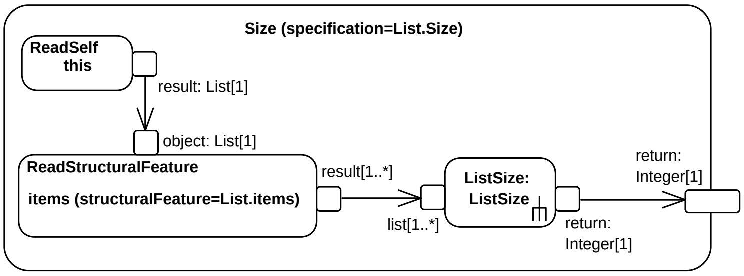


Figure 3.9: fUML activity for retrieving the size of a list.

Unused fUML Actions

Not all available fUML action types are needed in the proposed mapping. Table 3.7 depicts and describes all action types which are not used.

Unconsidered C# Features

The following listing describes some common C# features which are not considered within this work and outlines a possible solution for the mapping of each feature if it is possible.

- Break statement
  If a break statement in C# is used within a loop, the loop ends. This could be realized by mapping this break statements to *FlowFinalNodes* within the fUML activities.

- Continue statement
  If a continue statement in C# is used within a loop, the loop starts with the next iteration. This could only be partially solved. If a continue statement is used within a do-while, while, or for loop, the mapping could be done by using correct control flows. But there is no accurate mapping possible if a continue statement is used within a foreach loop.

Table 3.7: Unused fUML action types.

fUML action type	**Description**
destroy object action	Destroying an object in C# is not done manually but done automatically by the garbage collector after all references to the object have been set to *null*.
reclassify object action	There is no need for this action because C# does not support multiple inheritance.
read extent action	Receiving all objects from a class is not possible in C#.
start classified behavior action	The start object behavior action can be used instead of the start classified behavior action for calling created constructors.
link actions	Links and associations are not explicit features of C# and therefor not needed.
send signal action	Send signal actions could be used for the mapping of delegates and events, but they are not covered in this work.
loop node	Loops can also be realized by looping control structure which is realized by decision and merge nodes, as described above. But of course loop nodes could also be used for realizing loops.

- Usage of external libraries
  It is an essential feature of C# to use existing libraries in each project. The .NET framework provides a lot of different classes for different purposes. A possible approach for the mapping would be to reverse engineer all used libraries within a C# project if the source code is available.

- Reflection
  Reflection allows to read the metadata of assemblies as well as information about types and modules at runtime. With reflection it is possible to create dynamically instances of a type, invoke methods or access fields and properties of an existing object. It is a very powerful mechanism which is provided by the .NET framework. fUML does not support the concept of reflection.

- Generics
  `public class Example<T> { ... }`
  Generic type parameters make it possible to design classes and methods in a generic way, so that the specification of one or more types is deferred until the class or method is declared and instantiated by client code. fUML does not support the concept of generics, but UML supports them by the usage of so-called *Templates*. A possible solution for the mapping of generics would be to add Templates to the fUML metamodel and enhance an fUML execution environment to support them.

- Exception handling

  ```
  try {
  ...
  } catch (Exception ex) {
  ...
  }
  ```

  Errors in C# programs at runtime are propagated by using the mechanism of exceptions. Exceptions can be thrown by the .NET framework common language runtime (CLR) or by the program code. Whenever an error occurs within a *try* block, the system propagates the execution of the *catch* block. An error handling can be provided within the *catch* block. fUML does not support the concept of exceptions, but UML has support mechanisms for error handling. A possible solution for the mapping of exception would be to add the *RaiseExceptionAction* and the *ExceptionHandler* to the fUML metamodel and enhance an fUML execution environment to support them.

CHAPTER 4

Implementation of a C# to fUML Mapper

To validate the elaborated mapping between C# and fUML, we have developed a prototype capable of reverse engineering C# programs to fUML models. The prototype is realized as an Add-In for Enterprise Architect version 12 and implemented in C# within Visual Studio 2013 using .Net 4.0. Figure 4.1 gives an overview of the implemented prototype. The prototype consists of the following components:

- Enterprise Architect Add-In
  This component represents the Enterprise Architect Add-In which puts all needed components together. It enables the selection of a C# project and serves as entry point for the whole reverse engineering process. The basics of Enterprise Architect Add-In development is described in Section 4.1.

- Parser
  The parser is used to read C# code files, generate an AST of the code, and transform the AST into a C# model. The functionality of this component is described in Section 4.2.

- C#-to-fUML Transformation
  The C#-to-fUML transformation is responsible for the transformation from a C# model to an fUML model. The functionality of this component is described in Section 4.3.

- Serializer
  This component is responsible for the persistence of the fUML model. It uses the COM interface of Enterprise Architect for storing model elements in the internal storage of Enterprise Architect. This component of the prototype is discussed in Section 4.4.

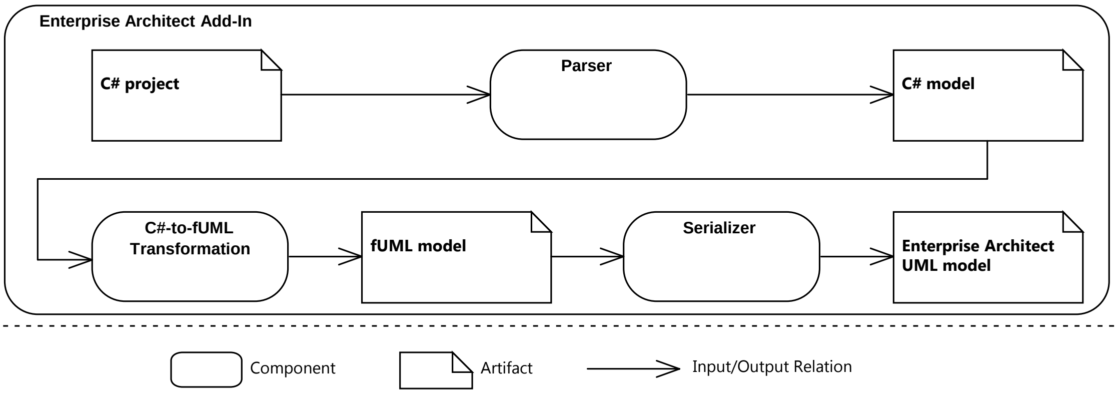

Enterprise Architect Add-In
C# project
Parser
C# model
C#-to-fUML Transformation
fUML model
Serializer
Enterprise Architect UML model
Component
Artifact
Input/Output Relation

Figure 4.1: Overview of the prototype implementation.

4.1 Enterprise Architect Add-In

The implemented Enterprise Architect Add-In provides a file selection dialog allowing the user to select the C# project that should be reverse engineered to fUML. After selecting a C# project file, the Parser component of the Add-In transforms the C# code of the selected project into a C# model as described in Section 4.2. Thereafter the C# model is transformed into an fUML model by the C#-to-fUML Transformation component which is described in Section 4.3. Afterwards the Serializer component stores the fUML model in Enterprise Architect, and the generated model can be accessed within the project browser of Enterprise Architect. The set of functions which Enterprise Architect provides for accessing the internal model and adding new elements to a model is described in Section 4.4.

Figure 4.2 shows the user interface of Enterprise Architect. The toolbox on the left side can be used to add new model elements to an existing model. Diagrams are displayed in the middle of Enterprise Architect, and the project browser showing the complete content of the model is on the right side. The project browser displays the model in a tree view. To start the conversion from existing C# code to fUML models a package has to be selected in the project browser, and the *Extensions/CSharp2fUML/Convert* menu entry of the context menu of the selected package has to be called. The reverse engineered fUML model will be stored in the selected package.

Overview of the Prototype Implementation

The prototype consists of several components which can be seen in Figure 4.1. Figure 4.3 depicts the main classes of the implementation of these components. The class *Enterprise_Architect_Add_In* is the main entry point of the whole conversion process. It has to implement the *Add_In_Interface* defined by Enterprise Architect to be recognized as an Enterprise Architect Add-In. The *StartConverting* method is called whenever a user clicks on the *Convert* menu entry which can be seen in Figure 4.2. It calls the *Parse* method of the *Parser* class, which parses the selected C# project. There are several methods available within the parser, which are parsing different parts of the C# project, e.g., the *ParseClass* method parses the class declarations, the *ParseMethods* operation parses the method declarations, etc. The functionality of the parser is described in

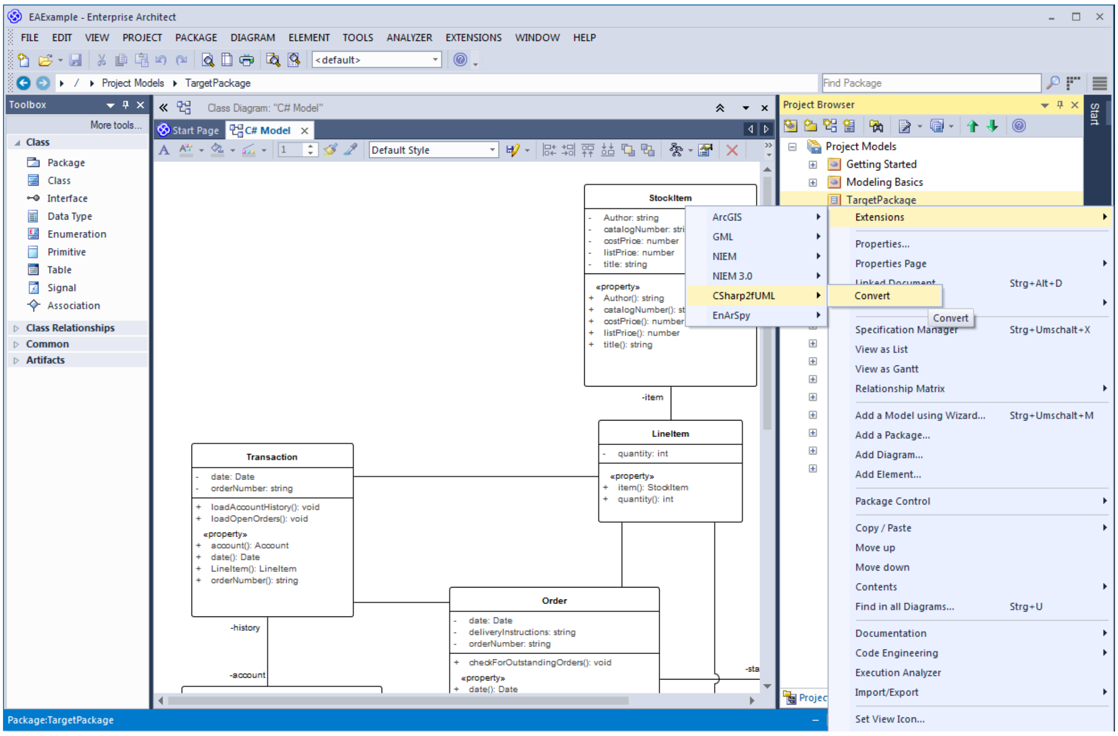


Figure 4.2: User Interface of Enterprise Architect.

Section 4.2. The Parser returns a C# model which serves as Input for the *CreateFumlModel* method of the *FUmlFactory* class, which does the C#-to-fUML transformations. There are again several methods available which take care of the transformation for different code element types. The transformation into an fUML model is described in Section 4.3. At the last step, the *FUmlFactory* returns an fUML model, which serves as input for the *WriteModel* method of the *ModelWriter* class, which takes care of the serialization of the fUML model in Enterprise Architect.

Implementation of an Enterprise Architect Add-In

To be able to access the internals of Enterprise Architect and also to store new model elements, an Add-In has to be implemented. For this implementation several steps have to be done. The following enumeration describes the steps which are needed when implementing an Add-In for Enterprise Architect using C#.

- Set up a ClassLibrary project
  First a C# class library project has to be setup.

- Make library COM interoperable
  Enterprise Architect uses the COM interface for communication with an Add-In, and therefor the class library project has to be COM interoperable. This can be done by ticking

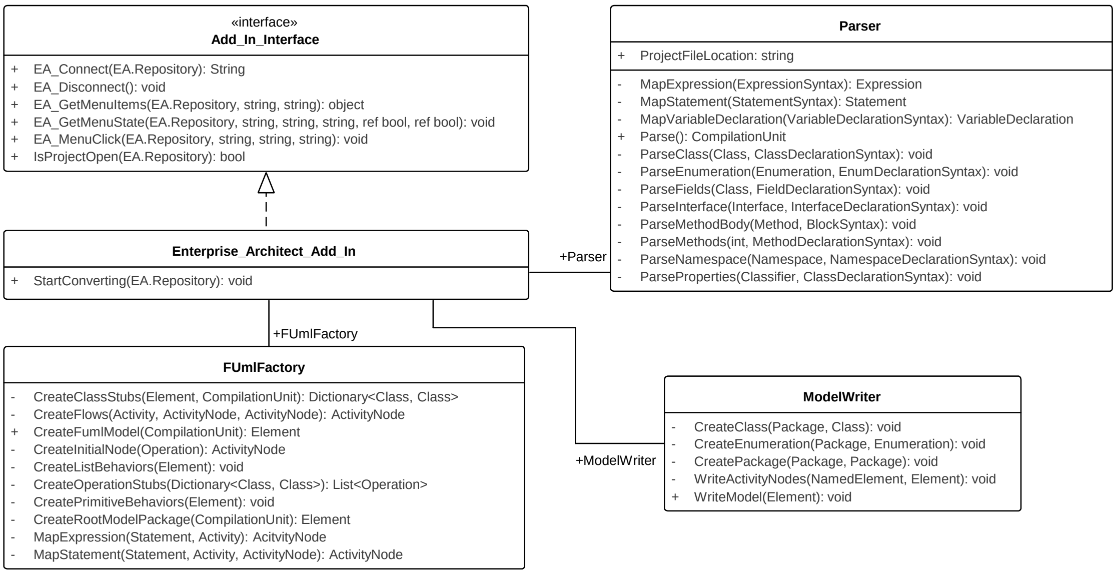

«interface»
Add_In_Interface
+ EA_Connect(EA.Repository): String
+ EA_Disconnect(): void
+ EA_GetMenuItems(EA.Repository, string, string): object
+ EA_GetMenuState(EA.Repository, string, string, string, ref bool, ref bool): void
+ EA_MenuClick(EA.Repository, string, string, string): void
+ IsProjectOpen(EA.Repository): bool
Parser
+ ProjectFileLocation: string
- MapExpression(ExpressionSyntax): Expression
- MapStatement(StatementSyntax): Statement
- MapVariableDeclaration(VariableDeclarationSyntax): VariableDeclaration
+ Parse(): CompilationUnit
- ParseClass(Class, ClassDeclarationSyntax): void
- ParseEnumeration(Enumeration, EnumDeclarationSyntax): void
- ParseFields(Class, FieldDeclarationSyntax): void
- ParseInterface(Interface, InterfaceDeclarationSyntax): void
- ParseMethodBody(Method, BlockSyntax): void
- ParseMethods(int, MethodDeclarationSyntax): void
- ParseNamespace(Namespace, NamespaceDeclarationSyntax): void
- ParseProperties(Classifier, ClassDeclarationSyntax): void
Enterprise_Architect_Add_In
+ StartConverting(EA.Repository): void
+Parser
+FUmlFactory
FUmlFactory
- CreateClassStubs(Element, CompilationUnit): Dictionary<Class, Class>
- CreateFlows(Activity, ActivityNode, ActivityNode): ActivityNode
+ CreateFumlModel(CompilationUnit): Element
- CreateInitialNode(Operation): ActivityNode
- CreateListBehaviors(Element): void
- CreateOperationStubs(Dictionary<Class, Class>): List<Operation>
- CreatePrimitiveBehaviors(Element): void
- CreateRootModelPackage(CompilationUnit): Element
- MapExpression(Statement, Activity): AcitvityNode
- MapStatement(Statement, Activity, ActivityNode): ActivityNode
+ModelWriter
ModelWriter
- CreateClass(Package, Class): void
- CreateEnumeration(Package, Enumeration): void
- CreatePackage(Package, Package): void
- WriteActivityNodes(NamedElement, Element): void
+ WriteModel(Element): void

Figure 4.3: Main classes and methods of the prototype.

the *Register for COM interop* setting on the project settings and the *ComVisible* attribute in the *AssemblyInfo.cs* has to be set to *true*.

- Reference Interop.EA.dll
  The *Interop.EA.dll* library comes with Enterprise Architect and is needed to access and communicate with Enterprise Architect. It can be found in the installation directory of Enterprise Architect.

- Set registry key for Enterprise Architect
  A registry key must be set for Enterprise Architect, so that Enterprise Architect is able to load the Add-In at startup. The registry key has to be set at *HKEY_CURRENT_USER\Software\Sparx Systems\EAAddins* and consists of the full qualified class name of the main class in the created class library project.

- Implement needed methods
  Several methods must be implemented to communicate with Enterprise Architect. The following list describes the needed methods and shows the concrete implementation used for the prototype:

  - EA_Connect
    This method is executed when Enterprise Architect starts, to check if an Add-In exists. This method needs to exist for the Add-In to work but nothing has to be done here.
    ```
    public String EA_Connect(EA.Repository Repository)
    {
    ```

```
 return "";
}
```

- EA_Disconnect
  Enterprise Architect executes this method when it exits. It can be used to do some cleanup work.

```
public void EA_Disconnect()
{
 GC.WaitForPendingFinalizers();
}
```

- IsProjectOpen
  Determines if a model is currently opened within Enterprise Architect. This is done by trying to access the models within Enterprise Architect.

```
private bool IsProjectOpen(EA.Repository Repository)
{
 try
 {
  EA.Collection c = Repository.Models;
  return true;
 }
 catch
 {
  return false;
 }
}
```

- EA_GetMenuItems
  Enterprise Architect executes this method whenever a user clicks on the Add-In menu item within Enterprise Architect. In this method, it is possible to define which menu items are available for the Add-In. The prototype defines a top level menu option „CSharp2fUML“ and a submenu called „Convert“.

```
public object EA_GetMenuItems(EA.Repository Repository,
   string Location, string MenuName)
{
 switch (MenuName)
 {
  // defines the top level menu option
  case "":
   return "-&CSharp2fUML";
  // defines the submenu options
  case "-&CSharp2fUML":
   string[] subMenus = { "&Convert" };
   return subMenus;
 }
 return "";
}
```

- EA_GetMenuState
  Enterprise Architect executes this method when the top level menu has been opened, to see which menu items should be activated. In this example we are activating the „Convert“ submenu entry.

```
public void EA_GetMenuState(EA.Repository Repository,
   string Location, string MenuName, string ItemName,
   ref bool IsEnabled, ref bool IsChecked)
{
 if (IsProjectOpen(Repository))
 {
  switch (ItemName)
  {
   // enable Convert submenu
   case "&Convert":
    IsEnabled = true;
    break;
   // default state is disabled
   default:
    IsEnabled = false;
    break;
  }
 }
 else
 {
  // If no open project, disable all menu options
  IsEnabled = false;
 }
}
```

- EA_MenuClick
  Enterprise Architect executes this method whenever a user makes a selection in the menu. This is the main entry point where the desired functionality of the implemented Add-In is called. In this example the method *StartConverting* is called if the user clicks on the *Convert* submenu entry.

```
public void EA_MenuClick(EA.Repository Repository,
   string Location, string MenuName, string ItemName)
{
 switch (ItemName)
 {
  // user has clicked the Convert menu option
  case "&Convert":
   // start the mapping process
```

```
      StartConverting(Repository);
      break;
    }
  }
```

4.2 Parsing C# Source Code

The .Net Compiler Platform *Roslyn* [18] was chosen for parsing C# source code. It is an open source compiler from Microsoft, which provides APIs for performing code analysis, refactoring, compilation, etc., on C# and Visual Basic projects. This work is based on the April's End User Preview of Roslyn from 2014, which installs on top of Visual Studio 2013, because the final version of Roslyn was not released when this work started.

Listing 4.1: Example of how to retrieve a syntax tree and its root element with Roslyn.

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(code);
var root = (CompilationUnitSyntax)tree.GetRoot();
```

Listing 4.1 depicts an example of how it is possible to retrieve a C# syntax tree with Roslyn and how to access the root element of the syntax tree. The *ParseText* method uses a string parameter which represents the C# code.

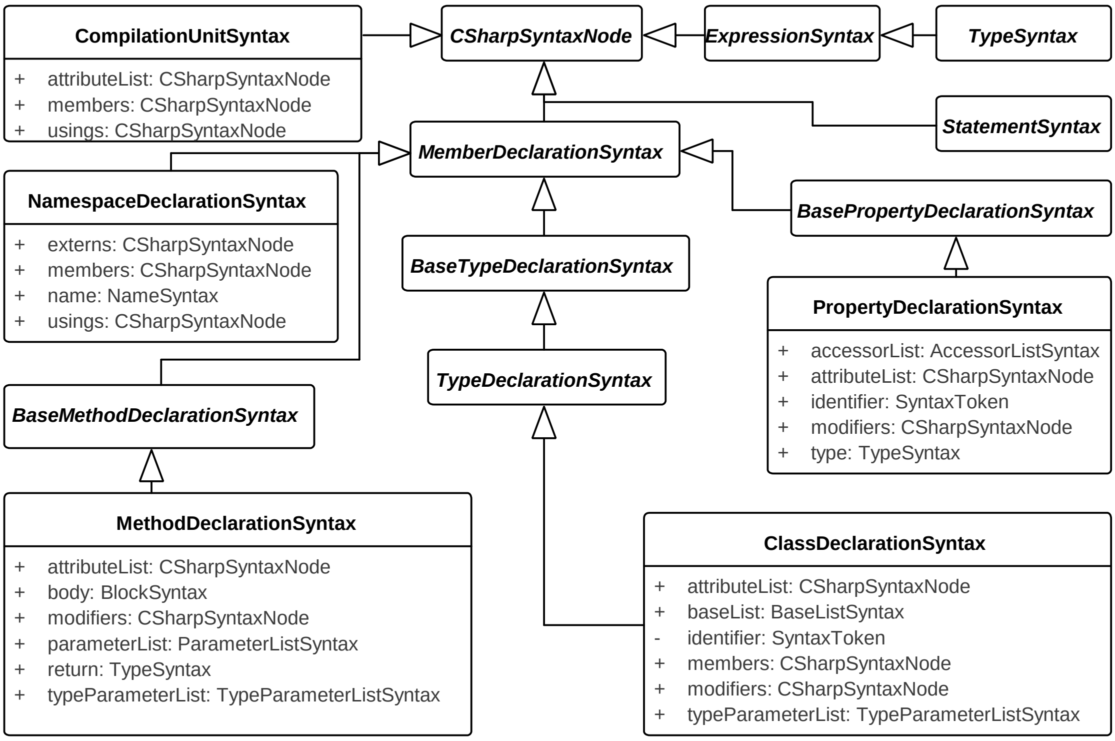

CompilationUnitSyntax
+ attributeList: CSharpSyntaxNode
+ members: CSharpSyntaxNode
+ usings: CSharpSyntaxNode
CSharpSyntaxNode
ExpressionSyntax
TypeSyntax
StatementSyntax
MemberDeclarationSyntax
NamespaceDeclarationSyntax
+ externs: CSharpSyntaxNode
+ members: CSharpSyntaxNode
+ name: NameSyntax
+ usings: CSharpSyntaxNode
BasePropertyDeclarationSyntax
BaseTypeDeclarationSyntax
PropertyDeclarationSyntax
+ accessorList: AccessorListSyntax
+ attributeList: CSharpSyntaxNode
+ identifier: SyntaxToken
+ modifiers: CSharpSyntaxNode
+ type: TypeSyntax
TypeDeclarationSyntax
BaseMethodDeclarationSyntax
MethodDeclarationSyntax
+ attributeList: CSharpSyntaxNode
+ body: BlockSyntax
+ modifiers: CSharpSyntaxNode
+ parameterList: ParameterListSyntax
+ return: TypeSyntax
+ typeParameterList: TypeParameterListSyntax
ClassDeclarationSyntax
+ attributeList: CSharpSyntaxNode
+ baseList: BaseListSyntax
- identifier: SyntaxToken
+ members: CSharpSyntaxNode
+ modifiers: CSharpSyntaxNode
+ typeParameterList: TypeParameterListSyntax

Figure 4.4: Excerpt of the Roslyn C# syntax tree.

Figure 4.4 depicts an excerpt of the C# syntax tree used by Roslyn. The base element is a *CSharpSyntaxNode*. The root element of each C# syntax tree is the *CompilationUnitSyntax* element. *ClassDeclarationSyntax* elements are used for class declarations, *MethodDeclarationSyntax* elements are used for method declarations within classes, *PropertyDeclarationSyntax* elements are used for property declarations, and *NamespaceDeclarationSyntax* elements are used for namespace declarations. There are a lot of specializations of *StatementSyntax* elements, e.g. for local declaration statements, block statements or expression statements. The same applies to *ExpressionSyntax* elements, e.g. for binary expressions, unary expressions, member access expressions, etc.

Listing 4.2 depicts a C# program containing a namespace declaration *Example* followed by a class declaration with the name *Test*, which has an auto-implemented property *Name* with a set and a get accessor.

Listing 4.2: Example of a simple C# class.

```
namespace Example {
  public class Test {
    public string Name { get; set; }
  }
}
```

Figure 4.5 depicts an instance of the C# syntax tree after parsing the example from Listing 4.2 with Roslyn. The root element is the *CompilationUnitSyntax* element which has a *NamespaceDeclarationSyntax* element representing the namespace *Example* as referenced member. This *NamespaceDeclarationSyntax* element references a *ClassDeclarationSyntax* element representing the declaration of the class *Test* as member. The *ClassDeclarationSyntax* element in turn references a *PropertyDeclarationSyntax* element representing the declaration of the property *Name* as member. This *PropertyDeclarationSyntax* element references an *AcessorListSyntax* element, which contains an *AccessorDeclarationSyntax* element for each of the accessor defined for the property *Name* in the example. The names of the different elements are expressed by *SimpleNameSyntax* elements and *SyntaxToken* elements.

Transformation into a C# Model

The C# syntax tree, which is generated by the parser, is in the next step transformed into an instance of a C# model which was implemented within this work. The transformation to a C# model was done to be able to use another parser in future, or to be able to use a newer version of Roslyn without influencing the transformation to an fUML model. The C# model was elaborated with the help of the C# Language Specification[1] (in particular with the lexical grammar given in the specification) and the Roslyn syntax tree model. This elaborated C# model is not complete, but only the most common language features have been considered which are required to support the C#-to-fUML mapping described in Chapter 3.

[1] C# Language Specification - http://www.microsoft.com/en-us/download/details.aspx?id=7029

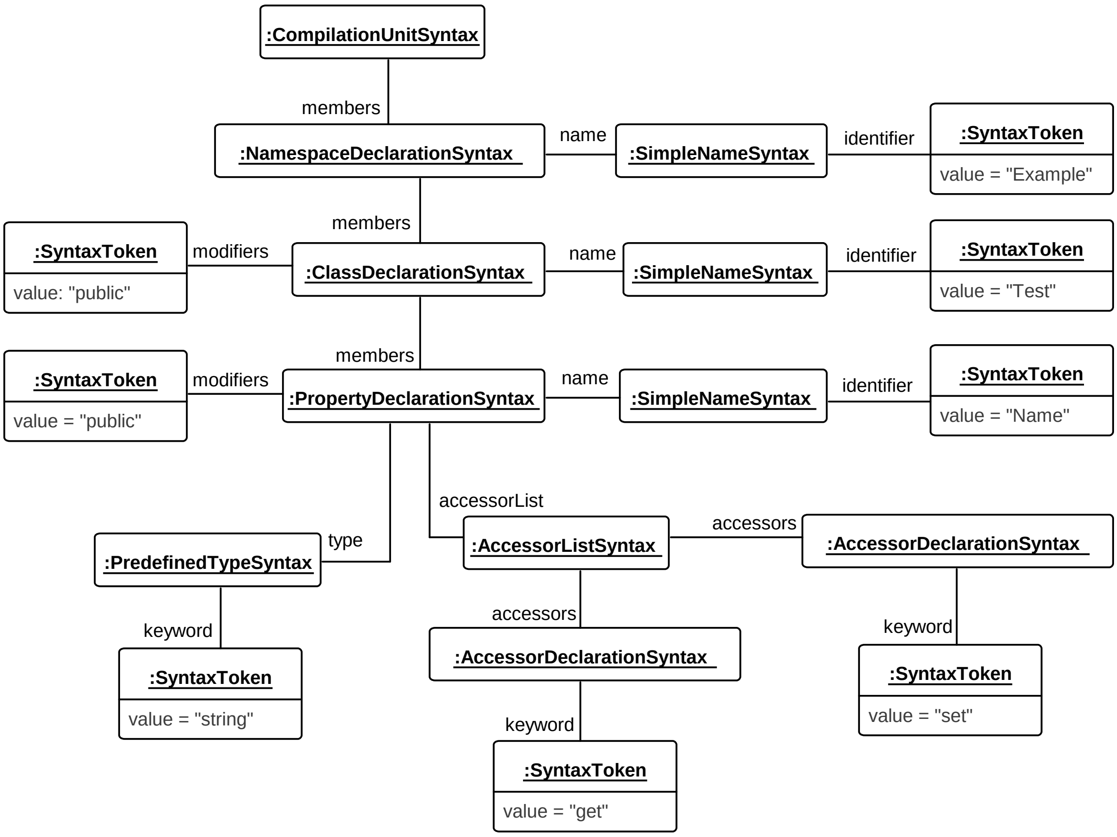


Figure 4.5: C# syntax tree obtained from Roslyn representing the example from Listing 4.2.

C# Metamodel

Figure 4.6 shows the main metaclasses defined in the elaborated C# metamodel. The most general metaclass is *CodeElement*, which represent each C# code element. It defines a reference to the metaclass *CodeComment*, which is used for representing comments in C# code, because each C# code element can be associated with a comment. The root element of each C# project is an assembly or application, which is represented by an instance of the *CompilationUnit* metaclass that contains the representations of all other code elements. The *Block* metaclass represents all code blocks which is represented by using curly brackets in C#. All code elements within a block are represented by code elements which are contained in an *items* collection (represented by an association between *Block* and *CodeElement*). Furthermore, the *Block* metaclass offers also operations to *Add* and *Insert* code elements to its items collection. The *TypedElement* metaclass represents a code element which has a type, which is represented by the *Type* metaclass (assocation from *TypedElement* to *Type*). Each case of a switch-case statement is represented by the Case metaclass, which contains a block as body (assocation from Case to Block), and contains a condition represented by an instance of the *Expression* metaclass. The *Statement* element serves as base class for all possible statements in C#, e.g. return statements, loops, switch statements, etc., which is shown in Figure 4.7.

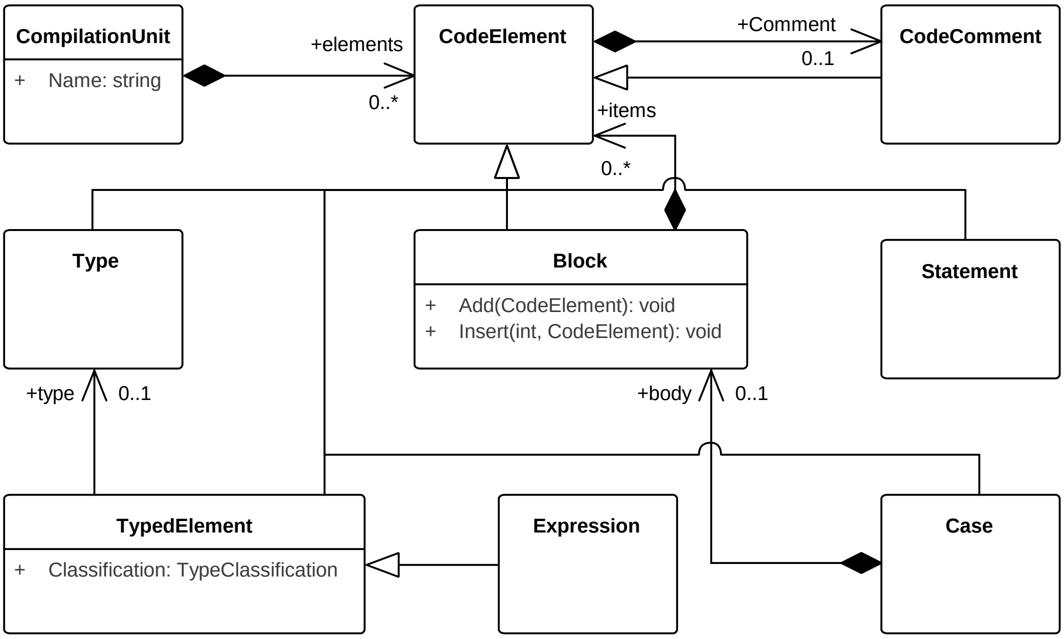


Figure 4.6: Main metaclasses of the C# metamodel.

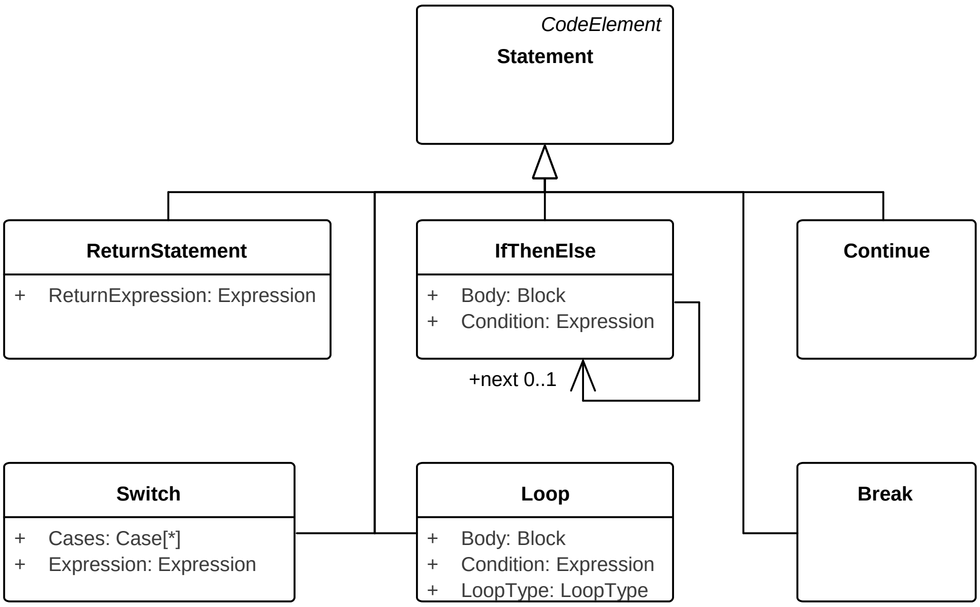


Figure 4.7: Metaclasses for representing C# statements.

Figure 4.8 shows that the metaclass *TypedElement* serves as base class for C# elements which have a type, e.g. methods (metaclass *Method*), fields (metaclass *Field*), parameters (metaclass

Parameter), expressions (metaclass *Expression*), etc. Except for parameters and expressions, typed elements also consist of a declaration modifier, e.g. *private*, *public*, etc., which is represented by the *DeclarationModifier* metaclass. C# expressions are represented by *Expression* elements, which are an specialization of the metaclass *TypedElement*. Different types of expressions exists, e.g. method calls, binary and unary expressions, null expressions, etc., which are shown in Figure 4.9.

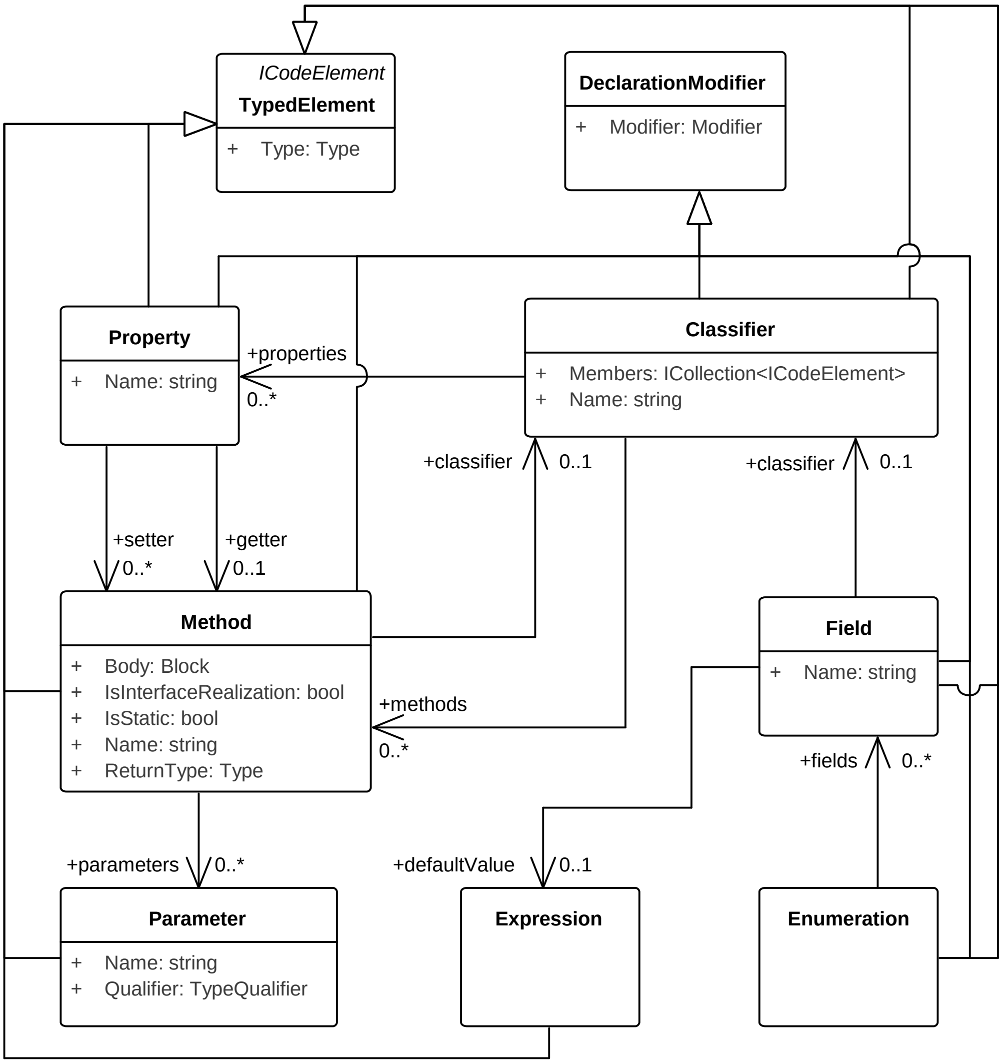


Figure 4.8: Metaclasses for representing C# typed elements.

Figure 4.10 depicts the possible specializations of *Type* and *Classifier* metaclasses. All enumerations which are used in the C# metamodel are summarized in Figure 4.11.

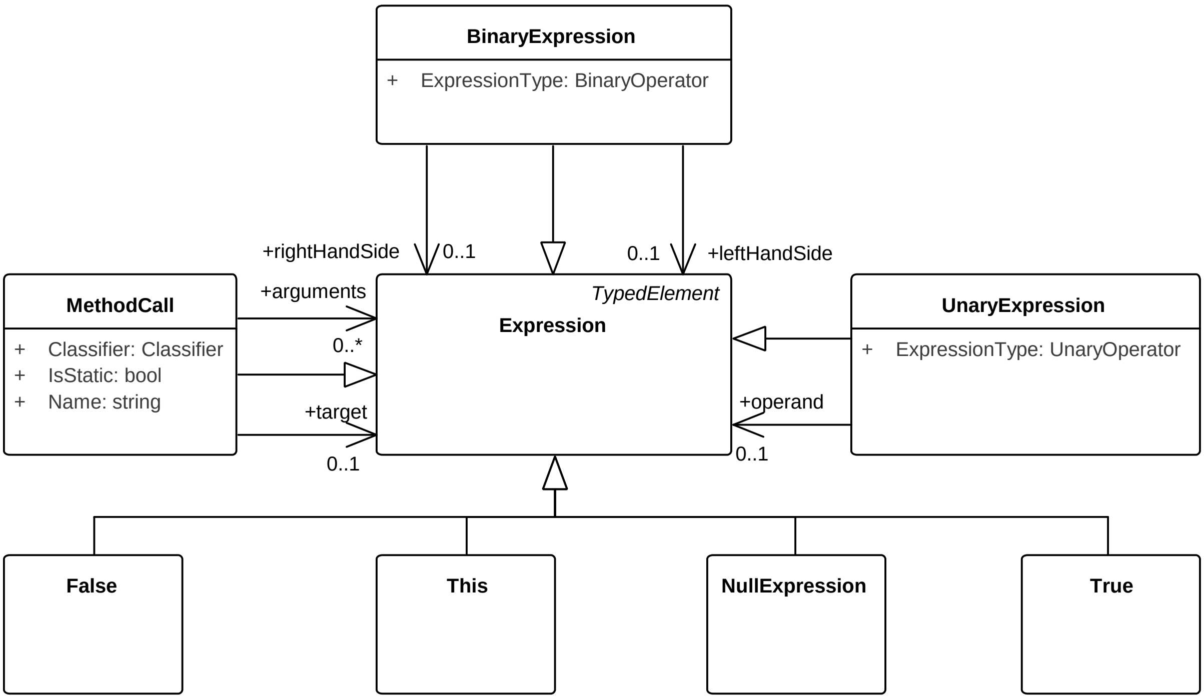


Figure 4.9: Metaclasses for representing C# expressions.

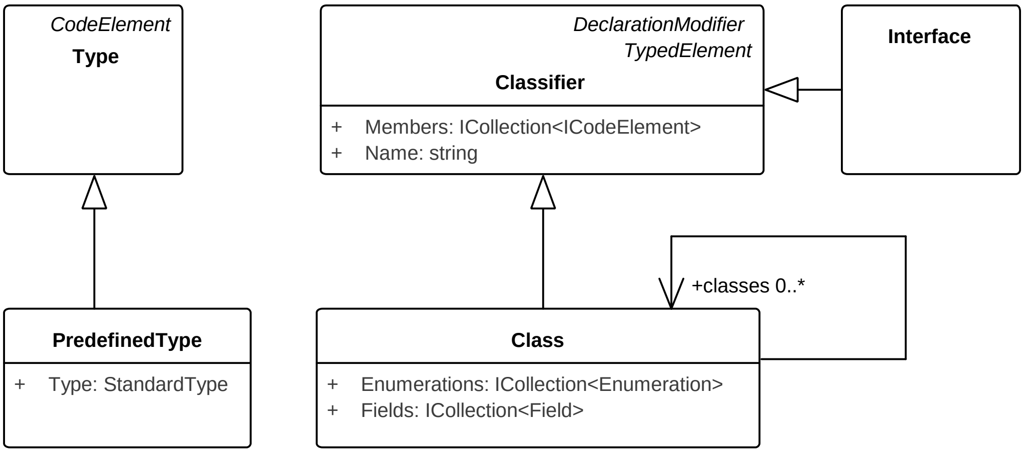


Figure 4.10: Metaclasses for representing C# types and classifiers.

Mapping the Roslyn Syntax Tree to a C# Model

The mapping from the C# syntax tree to the C# metamodel is straight-forward because of the similarities of the elaborated C# metamodel and the C# syntax tree used by Roslyn. Table 4.1 depicts the mapping between the C# syntax tree and the elaborated C# metamodel.

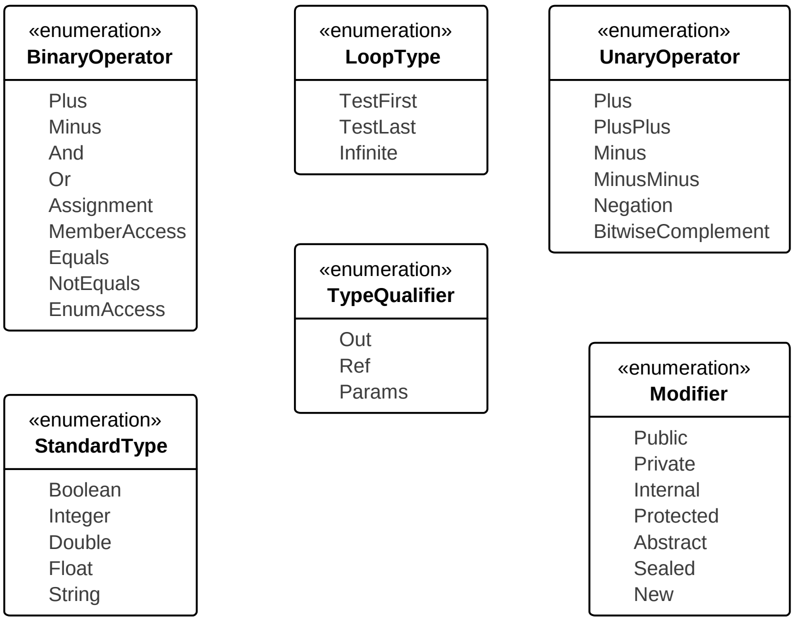


Figure 4.11: Enumeration elements used in the C# metamodel.

Table 4.1: Mapping between C# syntax tree and C# metamodel.

Syntax tree element	**C# metamodel element**
CompilationUnitSyntax	ICompilationUnit
UsingDirectiveSyntax	IUsing
NamespaceDeclarationSyntax	INamespace
InterfaceDeclarationSyntax	IInterface
EnumDeclarationSyntax	IEnumeration
EnumMemberDeclarationSyntax	IEnumerationMember
ClassDeclarationSyntax	IClass
PropertyDeclarationSyntax	IProperty / IAutoProperty
MethodDeclarationSyntax	IMethod
ConstructorDeclarationSyntax	IMethod
ConstructorInitializerSyntax	IConstructorInitializer
FieldDeclarationSyntax	IField
ParameterSyntax	IParameter
StatementSyntax	IStatement
BlockSyntax	IBlock
ReturnStatementSyntax	IReturnStatement
LocalDeclarationSyntax	ILocalDeclarationSyntax
ExpressionStatementSyntax	IExpressionStatement
IfStatementSyntax	IIfStatement / IElseClause

SwitchStatementSyntax	ISwitch
SwitchSectionSyntax	ICase
BreakStatementSyntax	IBreak
WhileStatementSyntax	IWhile
DoStatementSyntax	IWhile
ForStatementSyntax	IForLoop
ForEachStatementSyntax	IForEachStatement
VariableDeclarationSyntax	IVariableDeclaration
VariableDeclaratorSyntax	IVariableDeclarator
MemberAccessExpressionSyntax	IMemberAccessExpression
ThisExpressionSyntax	IThis
LiteralExpressionSyntax	ILiteralExpression
BinaryExpressionSyntax	IBinaryExpression
IdentifierNameSyntax	IIdentifierName
InvocationExpressionSyntax	InvocationExpression
UnaryExpressionSyntax	IUnaryExpresssion
ObjectCreationExpressionSyntax	IObjectCreationExpression
InitializerExpressionSyntax	IInitializerExpression
PredefinedTypeSyntax	IPredefinedType
GenericNameSyntax	IGenericName
ArgumentSyntax	IArgument
BaseListSyntax	IBaseList

Listing 4.12 depicts the C# model obtained from the example shown in Listing 4.2. The *CompilationUnit* references the *Class* „Student“, which contains the *Property* „Name“. The property references a *Method* for the getter and a *Method* for setter, and the setter method references the *Parameter* „value“.

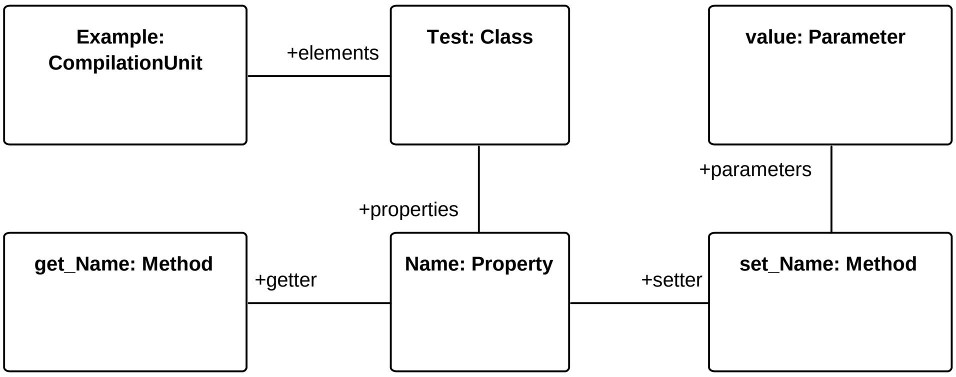


Figure 4.12: C# model of the example given in Listing 4.2.

```
input  : $compilationUnit$: The C# model
output: $rootModelPackage$: The root package of the fUML model
1  $rootModelPackage \leftarrow$ CreateRootModelPackage($compilationUnit$);
2  CreatePrimitiveBehaviors($rootModelPackage$);
3  CreateListBehaviors($rootModelPackage$);
4  classes $\leftarrow$ CreateClassStubs($rootModelPackage, compilationUnit$);
5  methods $\leftarrow$ CreateOperationStubs(classes, $compilationUnit$);
6  foreach method in methods do
7      lastnode $\leftarrow$ CreateInitialNode(method);
8      foreach statement in method.Statements do
9          lastnode $\leftarrow$ MapStatement(statement, method.Activity, lastnode);
10     end
11     final $\leftarrow$ CreateFinalNode(method.Activity);
12     CreateFlows(method.Activity, lastnode, final);
13 end
```

Algorithm 4.1: Main algorithm for mapping a C# model to an fUML model.

4.3 Transforming C# Models into fUML Models

The *fUMLFactory* component traverses the generated C# model and transforms its elements into fUML elements according to the mappings presented in Chapter 3. For *INamespace* elements Packages are created.

Algorithm 4.1 shows a simplified version of the algorithm which generates the fUML model. An *ICompilationUnit* (compilationUnit) representing the root element of the C# model is provided as input, and the root package of the mapped fUML model is returned (rootModelPackage). First, the root model package is created (line 1). Note that the primitive behaviors, as well as the predefined list behaviors, which are described in Section 3.3, are created for each run of the whole mapping process (line 2- 3), i.e., the primitive behaviors and predefined list behaviors are added to each fUML model reverse engineered from a C# compilation unit.

Creating the structural parts of the model (lines 4- 5). First, all packages, classes, enumerations, interfaces, properties, and fields are created, and stored in a seperate data structure which is returned (line 4). Afterwards all operations, defined in the C# compilation unit are mapped into corresponding fUML operations and parameters. This mapping includes the creation of one activity per operation containing acticity parameter nodes corresponding to the operation's parameters. The created operations are stored in a variable (line 5).

Creating the behavioral parts of the model (lines 6- 13). The mapping of the behavioral part of each operation is done for each operation seperately. First an inital node is created (line 7) for each method. Afterwards each statement within the method is mapped seperately by the MapStatement method (line 9) which is described in Algorithm 4.2. Subsequently the final node of the activity is created (line 11), and last a control flow edge is created between the activity final node and the activity node representing the last statement of the method (line 12).

Algorithm 4.2 shows the algorithm which is responsible for the mapping of a single statement defined within a method. The inputs of the method are the current statement which has to

```
input  : statement: The current C# statement, activity: The activity representing the
         method, lastnode: The last activity node of the previous mapping.
output: lastnode: The last activity node of the mapping of this statement
1 if statement is of type IBlock then
2     foreach substatement in statement do
3         lastnode ← MapStatement (substatement, activity, lastnode) ;
4     end
5 end
6 lastexpressionnode ← MapStatementWithExpression (type, statement,
  activity) ;
7 CreateFlows (activity, lastnode, lastexpressionnode) ;
8 lastnode ← lastexpressionnode;
```

Algorithm 4.2: Algorithm for mapping C# statements to fUML (MapStatement).

be mapped (statement), the activity which is representing the current method (activity), and the last activity node (lastnode) of the previous statement mapping (or the initial node if it is the first statement). The variable lastnode is needed to be able to set the correct object and control flows within the activities.

Mapping a block (line 1- 5). If the current statement is a block of other statements (represented by an IBlock), each sub-statement is mapped seperately by calling the same method recursively.

Mapping a statement with its expression (line 6- 8). For all statements which are not of type IBlock, the statement and its containing expression are mapped by a seperate method MapStatementWithExpression which is an implementation of the mapping described in Section 3.3 (line 6). The mapping of the statement and its expression returns the last created activity node (lastexpressionnode) which is needed to be able to create the object and control flows correctly (line 7). Finally, the last node created for the mapped expression is returned as last node of the statement (line 8).

Algorithm 4.3 shows as an example for mapping statements the algorithm for mapping an invocation expression statement which is a method call in C#. It is one of 12 implemented statement mappings. First the operation is determined which has to be called (line 1). Then a call operation action has to be created with pins for all operation parameters and object flows have to be set correctly (line 2- 3). If the operation has a return parameter, a return pin has to be created (line 4-line 5). The call operation action is returned to the caller of the operation.

The created root model package of the mapped fUML model is returned to the caller of the mapping process and serves as input for storing the fUML model within Enterprise Architect, which is described in the following Section 4.4.

4.4 Storing fUML Models in Enterprise Architect

Enterprise Architect provides an object of type *EA.Repository* which is the main element to access a model within Enterprise Architect. It is a container of all model structures and can be used for accessing the opened model. Listing 4.3 depicts an example of how new packages can

```
input  : $expressionStatement$: The C# expression statement, $activity$: The activity
         representing the method.
output: $lastnode$: The last activity node of the mapping of the expression statement
operationToCall ← GetOperation($expressionStatement, activity$);
callOperationAction ← CreateOperationAction(operationToCall, $activity$);
MapPossibleParametersOfOperation(operationToCall, callOperationAction,
$activity$);
if operationToCall has a return type then
    CreateReturnPin(operationToCall, callOperationAction, $activity$);
end
$lastnode$ ← callOperationAction;
```

Algorithm 4.3: Algorithm for mapping a C# invocation expression statement to fUML.

be created. First the *Package*, which is currently selected in the project browser, is accessed. In the next step the *Packages* collection is accessed from the current package which contains all sub packages. The *AddNew* method of this collection adds a new sub package to the current package. The first parameter of the *AddNew* method is the name of the package which should be created and the second parameter is the type of the element which should be created. After the creation it is possible to set other properties of the newly created element. In this example some further description of the element is set. As last step, we have to call the *Update* method on the newly created element to make sure that the changes will be stored within the model.

Listing 4.3: Example of new Package element creation in Enterprise Architect.

```
EA.Package package = repository.GetTreeSelectedPackage();
var newPackage = (EA.Package)package.Packages.AddNew(
    "ExamplePackage", "Package");
newPackage.Notes = "Further␣Description.";
newPackage.Update();
```

There are several different types of elements in Enterprise Architect. The EA.Package type represents UML packages, the EA.Element type represents a broad range of UML elements like Classes, Activities, Actions, Components, etc., and the EA.Connector type represents any type of a connector between elements. Further description of the available types and features can be found within the Enterprise Architect Object Model User Guide[2].

Storing the fUML Model in Enterprise Architect

Storing the mapped fUML model in Enterprise Architect is straightforward. First the fUML model is traversed and all structural parts, like packages, classes, interfaces, enumerations, attributes, operations and its parameters are created. After the creation of the structural parts, the activities created for the operations of classes are added to the model. First, each activity node

[2] Enterprise Architect Object Model User Guide - http://www.sparxsystems.com.au/resources/user-guides/automation/enterprise-architect-object-model.pdf

of the activity is retrieved and the corresponding element is created in the Enterprise Architect model and initialized in accordance with the properties of the activity node. Afterwards each activity edge of the activity is retrieved and the corresponding connector is created in the Enterprise Architect model. This way, the resulting model in Enterprise Architect corresponds to the reverse engineered fUML model.

CHAPTER 5

Evaluation

In Chapter 3 the conceptual mappings of C# code to fUML models were described. Based on these mappings, a prototypical implementation was developed as described in Chapter 4. This prototype was used to evaluate the appropriateness of the elaborated mappings. The conduced evaluation is presented in this chapter. In particular, Section 5.1 describes the evaluation framework which was developed to evaluate the mappings, and Section 5.2 describes the different case studies which have been studied in. Finally, Section 5.3 discusses the results of the evaluation.

5.1 Evaluation Framework

An own evaluation framework was developed for the evaluation of the developed fUML reverse engineering approach. It follows a test-driven approach which is based on unit tests. Unit tests are executing the code which is reverse engineered as well as the fUML models corresponding to that code, and compares the results of botch executions. If the results are equal, it can be concluded that the reverse engineered fUML model is complete, standard conform, and behavioral equivalent to the code for the given input. For the investigated case studies, it was ensured that all possible paths through the reverse engineered C# code are tested.

Execution of an fUML Model Stored in Enterprise Architect

Brunflicker [8] elaborated in this master's thesis a prototype, which is capable of executing fUML compliant models using the execution engine specified in the fUML standard, within the UML modeling environment Enterprise Architect. It uses the MOLIZ fUML Debug API [16] for executing the models by converting the entire API into DLLs by utilizing the IKVM[1] toolkit. A so-called Bridge converts the Enterprise Architect models into fUML models and passes them to the fUML execution engine. This prototype was extended corresponding to the needs of this work to support a broader range of the fUML abstract syntax, which were needed to gain a higher

[1] IKVM - http://www.ikvm.net/

test coverage for the elaborated C#-to-fUML mapping. Table 5.1 depicts the enhancements which have been made. Unfortunately the prototype is not capable of executing more complex activities which are needed e.g., for the proposed mapping of loops, nested structured activities with pins and decision nodes, etc. described in Section 3.3. Those tests of the case studies have been evaluated manually, by examining the model and comparing it against the proposed mapping.

Table 5.1: Changes on the prototype of Brunflicker [8].

fUML type	**Changes**
add structural feature action	Add support for *insertAt* Pin. Handle the *isReplaceAll* property as tagged value on the action because Enterprise Architect does not support this property.
activity parameter node	Add proper support for the direction of activity parameter nodes. Out parameters have to be returned after the execution of an activity.
structured activity node	Add support for structured activity nodes.
call operation action	Add support for call operation actions.
decision node	Add support for decision input flow.
value specification action	Add support for the primitive type *UnlimitedNatural*.
test identity action	Add support for test identity actions.
clear structural feature value action	Add support for clear structural feature value actions.
start object behavior action	Add support for start object behavior actions.
primitive types	Add support for primitive type *Integer*. Set proper default values for all primitive types.
primitive behaviors	Add support for primitive behaviors, which are defined in Section 3.3.

Unit Tests

Enterprise Architect identifies all model elements by using global unique identifiers (GUIDs). Therefor, a unit test class stub was manually developed for each test class. It includes everything needed for the tests, except the GUIDs for accessing the model elements. Placeholders are used instead of the GUIDs in the stubs. After the reverse engineering of a C# project to an fUML model is done with the developed Enterprise Architect Add-In, the placeholders are replaced by the GUIDs of the model elements in a post-processing step. Afterwards, the unit tests can be executed. This semi-automatic approach was used during the test-driven development of the prototype.

Each unit test, tests one activity. For executing an activity, a context object is needed. Therefor a context object is created, which is an instance of the class that contains the activity. Input parameters are provided by using a dictionary, which contains for each input parameter a value. To execute the activity, the execution method of the Enterprise Architect fUML execution engine is called with the GUID identifying the activity, and the reference to the context object. If there is any output parameter, the dictionary for the input parameters contains a „return“ parameter with the return value. Assertions are used to compare the output values of the activity with those of the executed C# method.

Listing 5.1 depicts an example unit test. First, an object of the reverse engineered C# class *ActionsExample* is created, and the return value of the method *ValueSpecificationActionString* is remembered in the variable *retVal*. Afterwards, the context object *actionsExampleContext* is created, which references the fUML class *ActionsExample*. Next, a dictionary is created and the *return* parameter is added which is set by the execution engine after the activity has been executed. The *Execute* method of the execution engine is called, and the GUID of the fUML activity to be executed, as well as the context object and the dictionary is provided. Last but not least, the return value of the C# method is asserted against the return value of the executed fUML activity. If the return value of the executed C# method and the return value of the executed activity are not equal, the failure message „Return value not equal.“ is shown when executing the unit test.

Listing 5.1: Example of a unit test.

```
[Test]
public void ValueSpecificationActionStringTest() {
 ActionsExample ae = new ActionsExample();
 string retVal = ae.ValueSpecificationActionString();
 ActionsExampleContext actionsExampleContext =
    new ActionsExampleContext(class_ActionsExample);
 Dictionary<string, object> dictionary =
    new Dictionary<string, object>();
 dictionary.Add("return", "");
 Execute(activity_valueSpecificationActionString.Guid,
    actionsExampleContext, dictionary);
 Assert.AreEqual(retVal, dictionary["return"],
    "Return␣value␣not␣equal.");
}
```

5.2 Case Studies

For evaluating the developed fUML reverse engineering approach, two case studies have been conducted. First, a simple example has been setup, which covers the different C# concepts, each in a separate method. It consists of 222 lines of code. Table 5.2 depicts an overview of the C# concepts, and the corresponding fUML concepts considered in the first case study.

Nr.	C# concept	fUML concept	Additional tests
1.1	literals	value specification action	Different primitive data types used.

1.2	this	read self action	Implicit usage of this as well as explicit access.
1.3	local variable declaration	value specification action if there is a default value, fork node	Primitive types as well as complex types used.
1.4	instance variable access	read structural feature action	Primitive types as well as complex types and collections used.
1.5	(auto) properties	add/read structural feature action	Primitive types as well as complex types used.
1.6	method call	call operation action	With and without parameters, different parameter directions, with and without return parameters.
1.7	object creation	create object action, start object behavior, call operation action	Class with complex types and simple types as properties used, with and without default values.
1.8	equality comparison	test identity action	Primitive types as well as complex types used, checking for null, inequality test.
1.9	operator expression	call behavior on primitive behaviors	Different operators used.
1.10	type checking	read is classified object action	
1.11	if-else statement	structured activity node with decision and control nodes using sub structured activity nodes for if and else block	With and without else clause, return within if/else, and nested if-else considered.
1.12	do-while loop	structured activity node with looping control structure using decision and merge nodes	Break within loop and return within loop considered.
1.13	while loop	combination of if-else statement concepts combined with do-while loop concepts	Break on decision inside loop and return inside loop considered.
1.14	for loop	treated like while loop (cf. 1.13)	Treated like while loop (cf. 1.13)
1.15	foreach loop	expansion region	Continue inside loop and return inside loop considered.
1.16	removing list items	remove structural feature value action	Remove by index considered.

1.17	clearing a list	clear structural feature action	
1.18	adding list items	add structural feature value action	Primitive types as well as complex types used.
1.19	set new value on property	add structural feature value action	
1.20	set property to null	clear structural feature value action	
1.21	switch-case statement	treated like nested if-else structure	Return inside case, default case, and combined cases considered.
1.22	working with variables	forks	Using different variables and calculations on them.

Table 5.2: C# concepts tested in first case study.

The second case study is based on the .NET PetShop MVC[2] example. It consists of 306 lines of code, and mainly consideres the data model of PetShop and the methods operating on this model. The database related code was completely removed because this work does not consider the usage of external libraries. This case study comprises 15 classes with 15 corresponding interfaces and one enumeration. There is a total number of 21 methods, and 31 properties which are not auto-properties. Table 5.3 depicts the methods considered in this case study where each method corresponds to one test case and gives a short description of the methods' behavior.

Nr.	Method	Description
2.1	Cart.SetQuantity(itemId, qty)	Iterates over the cart items and updates the quantity for the given itemId.
2.2	Cart.Add(itemId)	Adds an item with the given id to the cart items. If the cart already contains the item it increases the quantity, otherwise a new cart item is added to the cart.
2.3	Cart.Add(item)	Adds the given item to the cart. If the cart already contains the item, the quantity is increased.
2.4	Cart.Remove(itemId)	Removes the item with the given itemId from the cart.
2.5	Cart.Clear()	Clears the cart items.
2.6	Category.GetCategories()	Returns the categorieInfos list.
2.7	Category.GetCategory (categoryId)	Returns the category with the given categoryId. If the category is not contained in the categorieInfos list, null is returned.
2.8	Category.Add (categoryInfo)	Adds the given categoryInfo to the categorieInfos list.
2.9	Inventory. CurrentQtyInStock(itemId)	Returns the total quantity of items with the given itemId contained in the inventory.

[2] .NET Pet Shop MVC - https://petshopmvc.codeplex.com/

2.10	Inventory. TakeStock(lineItems)	Reduces the quantity of each item in the inventory for the given lineItems.
2.11	Inventory.AddStock (itemId)	Increases the quantity of items in the inventory for the given itemId.
2.12	Item.Add(itemInfo)	Adds the itemInfo to the itemInfos list.
2.13	Item.GetItemsByProduct (productId)	Returns all itemInfos which are matching the given productId.
2.14	Item.GetItem(itemId)	Returns the itemInfo for the given itemId. If the item is not contained in the itemInfos list, null is returned.
2.15	Order.Insert(order)	Calls the ProcessCreditCard method and adds the given order to the orderInfos list.
2.16	Order.ProcessCreditCard (order)	A dummy method which would be used to check the given credit card information contained in the order.
2.17	Order.GetOrder(orderId)	Returns the orderInfo for the given orderId if it is contained in the orderInfos list, otherwise null is returned.
2.18	OrderInfo.GetOrderTotal()	Sums up the totals of each line item, calculates a discount depending on the credit card type and returns the order total.
2.19	Product.Add(productInfo)	Adds the productInfo to the productInfos list.
2.20	Product. GetProductsByCategory (category)	Returns all products from the productInfos list which are matching the given category.
2.21	Product. GetProduct(productId)	Returns the productInfo from the productInfos list for the given productId. Null is returned if the product is not contained in the list.

Table 5.3: PetShop methods considered in the second case study.

5.3 Results

This section presents the results of the evaluation of the above mentioned case studies and discusses identified issues. Possible solutions to overcome the identified issues are provided, and changes which have been made manually to get more unit tests running are described.

Case Study 1

For the first case study, 62 unit tests have been developed. Of these unit tests, 29 succeeded without the need for manual adaptions. Table 5.4 depicts the C# concepts considered in the first case study and provides for each the C# concept, the number of tests, the number of tests that succeeded without problems, the number of tests that succeed after manually applying changes on the retrieved fUML model, the number of tests that fail even after considering manual changes, and categorizes the failing tests. Please note that some test cases are covering more than one C# concept.

Nr.	C# concepts	UnitTests	Succeeding without changes	Succeeding with changes	Failing	Category
1.1	literals	2	2	0	0	
1.2	this	2	2	0	0	
1.3	local variable declaration	3	3	0	0	
1.4	instance variable access	3	2	0	1	A
1.5	(auto) properties	10	8	0	2	A
1.6	method call	3	3	0	0	
1.7	object creation	2	1	1	0	B
1.8	equality comparison	5	3	0	2	D, E
1.9	operator expression	4	4	0	0	
1.10	type checking	1	1	0	0	
1.11	if-else statement	5	0	1	4	C, G, E
1.12	do-while loop	3	0	1	2	C, G, F
1.13	while loop	3	0	1	2	C, G, F
1.14	for loop	3	0	0	3	E
1.15	foreach loop	3	0	0	3	A, E
1.16	removing list items	2	0	0	2	A
1.17	clearing a list	1	0	0	1	A
1.18	adding list items	2	0	0	2	A
1.19	set new value on property	6	3	0	3	A
1.20	set property to null	2	1	0	1	A
1.21	switch-case statement	3	0	0	3	E
1.22	working with variables	2	2	0	0	

Table 5.4: Result of the unit tests of the first case study.

After manual changes there were 35 unit tests succeeding out of 62. Possible solutions for the remaining failing tests are discussed in the end of this section. Manual changes were either correcting the object flow or the control flow. Whenever the target or source of a flow was wrong, or missing, it has been corrected manually.

Case Study 2

The second case study comprises 46 unit tests. Of these 46 test cases, only nine succeeded without problems and manual adaptions. Table 5.5 depicts the C# concepts considered in the second case study and provides for each the C# concepts, the number of tests, the number of tests that succeeded without problems, the number of tests that succeed after manually applying changes on the retrieved fUML model, the number of tests that fail even after considering manual changes, and categorizes the failing tests. Please note that some test cases are covering more than one C# concept.

Nr.	**C# concepts**	**UnitTests**	**Succeeding without changes**	**Succeeding with changes**	**Failing**	**Category**
2.1	foreach loop, if-else statement, equality comparison, property assignment	2	0	0	2	A
2.2	foreach loop, if-else statement, equality comparison, property assignment, operator expression, object creation, method call, adding items to list	2	0	0	2	A
2.3	local variable declaration, foreach loop, if-else statement, equality comparison, variable assignment, operator expression, if-else statement, adding items to a list	3	0	0	3	A

2.4	local variable declaration, foreach loop, if-else statement, equality comparison, variable assignment, removing items from a list	2	0	0	2	A
2.5	clearing a list	1	0	0	1	A
2.6	property access, return statement	2	2	0	0	
2.7	foreach loop, if-else statement, equality comparison, return statement	2	0	0	2	A
2.8	property access, adding items to a list	1	0	0	1	A
2.9	if-else statement, equality comparison, local variable declaration, foreach loop, variable assignment, operator expression, method call, return statement	3	1	0	2	A
2.10	foreach loop, if-else statement, equality comparison, method call, property access, operator expression	3	0	0	3	A

2.11	foreach loop, if-else statement, equality comparison, method call, operator expression	2	0	0	2	A
2.12	adding items to a list	1	0	0	1	A
2.13	local variable declaration, if-else statement, equality comparison, foreach loop, adding items to a list, return statement	3	1	0	2	A
2.14	if-else statement, equality comparison, foreach loop, property access, return statement	3	1	0	2	A
2.15	method call, adding items to a list	1	0	0	1	A
2.16	if-else statement, equality comparison, return statement	2	2	0	0	
2.17	if-else statement, for loop, accessing items in a list, equality comparison, return statement	4	1	0	3	E

2.18	local variable declaration, variable assignment, foreach loop, operator expression, switch-case statement, return statement	3	0	0	3	A, E
2.19	adding items to a list	1	0	0	1	A
2.20	local variable declaration, if-else statement, equality comparison, adding items to a list, return statement	3	1	0	2	A
2.21	foreach loop, if-else statement, equality comparison, return statement	2	0	0	2	A

Table 5.5: Result of the unit tests of the second case study.

Most of the test cases which are failing are due to the lack of the propper support of C# collection types. The identified issues are discussed in more detail in the following. Manual changes have not been possible for the failing test cases.

Categorization of Identified Issues

The issues identified in the two conducted case study have been classified into five categories A-E (cf. Table 5.4 and Table 5.5). In the following, these categories of issues are discussed.

- A
  Category A contains test cases investigating C# methods that make use of collection types. Due to the special conversion of collection types, described in Section 3.3, the prototypical Enterprise Architect fUML execution engine developed by Brunflicker [8] is not capable of handling the obtained fUML models correctly in the mapping from Enterprise Architect to fUML. fUML models containing operations on collections have been evaluated manually, by examining the reverse engineered fUML model and comparing it against the proposed mapping.

- B
  Category B contains test cases, where the control flow of the obtained fUML model is not completely correct. These issues have been resolved manually by adding missing control flows and adjusting control flows with a wrong source or target activity node. All test cases of this category succeeded after manual corrections.

- C
  Category C contains test cases, where the object flow of the obtained fUML model is not completely correct. These issues have been resolved manually by adding missing object flows and adjusting object flows with a wrong source or target activity node. All test cases of this category succeeded after manual corrections.

- D
  Category D contains test cases, where the execution engine was not capable of handling all used activity nodes. Therefore, the only possibility would have been to further enhance the execution engine.

- E
  Category E contains test cases, where C# concepts have been used which are not considered in this work (e.g., break statements, continue statements), or which are not implemented by the prototype (e.g., switch-case statements, for loops).

- F
  Category F contains test cases, where the execution engine was also not able to deal with some special object flows. E.g., the execution engine does not allow a decision input flow from a fork node (only decision input flows from output pins are allowed). Those restriction e.g., has been resolved by changing the source of the object flow from the fork node to the right output pin and removing the fork node from the model. All test cases of this category succeeded after manual adaptions.

- G
  Category G contains test cases, where the execution engine was not capable of converting the model from Enterprise Architect to fUML. This conversion is very error prone if the activity to convert contains many nested activity nodes with pins. The actual order of the conversion of the single activity nodes and edges is inefficient. It was tried to resolve some issues of this category manually but without success.

In summary there are three main problems within the implemented prototype. First, there are problems with the reverse engineering of the control flow (Category B). Second, there are problems with the reverse engineering of the object flow (Category C). Third, there are problems with unsupported C# features (Category E). The other problems are related to unsupported features of the execution engine (Category A, D, F, and G). The reverse engineered models for the failing tests have been evaluated manually. The only alternative possibility would have been to extend the prototype and the execution engine, which was however out of the scope of this thesis.

Discussion of the Results

The biggest challenge in realizing the prototype was the resolving of the correct object flow (c.f. failing test cases of Category C). Currently, whenever a variable is used within some block of code, the implementation tries to get the latest activity node which represents this variable (e.g., pins, forks, etc.). This kind of back-tracking is not always working. A better approach would be a forward-tracking algorithm, which explores all used variables within one block of code at the beginning, and provides the corresponding model elements when a variable access is processed in the reverse engineering. Then it would be easier to always identify the correct source or target of an object flow.

Another major issue was the resolving of the correct control flow, especially when a lot of nested activity nodes are used, or, whenever one statement in the code results in more than one activity node. Therefor, most of the mappings omit control flows and rely only on the object flow. However, this does not always lead to the correct execution order of reverse engineered fUML activities as shown by the failing test cases of Category B.

In the current implementation of the prototype, fUML loop nodes are not supported. Instead of using loop nodes, C# loops are reverse engineered into structured activity nodes with looping control structure around it using decision and merge nodes. A loop node would decrease the complexity when converting loops which would make it more robust. Another possible implementation could be to try to convert the different kinds of loops to one defined kind of loop (e.g., while loop) when transforming the C# syntax into a C# model. This would also reduce the complexity of the C#-to-fUML transformation.

Possible solutions to overcome the issues found within the evaluation are given in the following along the defined categories:

- A
  The prototypical Enterprise Architect fUML execution engine developed by Burnflicker [8] has to be extended with support for the generic List class, which is used for mapping C# collection types.

- B
  To overcome issues of this type, the prototype must be enhanced to correctly produce control flows for all mappings.

- C
  The handling of the object flows must be reworked completely within the proposed prototype. Instead of searching for the correct source and target nodes of a object flow in a back-tracking manner, a promising approach would be to first compute the pins and forks representing variables and using them in the mapping of C# method bodies.

- D
  The prototypical Enterprise Architect fUML execution engine has to be extended to support fUML completely.

- E
  Support of the missing C# concepts like for-loops, switch-case statements, break statements, and continue statements must be implemented in the prototype.

- F
  The prototypical Enterprise Architect fUML execution engine has to be extended to support fUML completely.

- G
  The prototypical Enterprise Architect fUML execution engine has to be extended to support fUML completely.

CHAPTER 6

Related Work

Although the fUML standard was released some years ago, there are no mature or established tools that support the reverse engineering of object oriented programs to fUML models. The work of Bergmayr et al. [2] described in Section 6.1 is the most related work regarding this master thesis. This work is also concerned with reverse engineering an object oriented programming language, namely Java, to fUML models.

Brunelière et al. [7] have elaborated a framework which supports the development of MDRE tools, which could be extended to support the reverse engineering of object oriented programming languages to fUML. The work is briefly described in Section 6.2.

Section 6.3 briefly discusses other related work regarding the reverse engineering of programming languages. But no one of those are attempting to reverse engineer code written in C# and targeting fUML as target language.

6.1 fREX

Bergmayr et al. [2] established an open framework called fREX for reverse engineering executable behaviors for dynamic software analysis. It aims to reverse engineer Java code to fUML models by using a static approach. The model discovery phase uses MoDisco [7], which generates a Java code model from Java source code. The obtained Java model conforms to a metamodel of the Java programming language which precisely describes the terminology and structure of Java. This Java model is transformed to fUML by using the ATL model-to-model transformation language and tooling by using a defined Java-to-fUML mapping, which is, like in this work, inspired from the initial mapping defined within the standard fUML specification [26].

fREX is capable of automatically generating and executing fUML models from existing Java source code. It uses the fUML virtual machine, which has been extended to provide execution traces [16]. fUML models are statically discovered from Java source code in the model discovery phase. In the model understanding phase, these fUML models are executed by using the fUML virtual machine which produce traces. This generated runtime model, together with the fUML model can be used for model-based analysis techniques, like model refinement, slicing, etc. Instead of using the whole range of the Java language, fREX is targeting MiniJava [1] as source language, which is a very small subset of Java.

The validation of this approach is done by applying a test-driven approach. Like the validation of this work, fREX uses unit tests for asserting that the derived fUML models, and the source Java code are behaving the same. This is also done by input-output comparison. For a given input, the result of the executed model is compared with the result of the executed code corresponding to the model.

The work of Bergmayr et al. pursues quite the same strategy for reverse engineering existing code to fUML like described in this work. But they are targeting Java (or rather MiniJava) instead of targeting C# as source language. The described work in this master's thesis provides also conversions for C# specific language features, as well as mechanisms to overcome some of the limitations of the fUML standard. The target UML modeling environment Enterprise Architect is also specific to the work presented in this thesis. Bergmayr et al. rely on the Eclipse platform.

6.2 MoDisco

Brunelière et al. [7] describe in their work a generic, extensible, and customizable framework which is called MoDisco (Model Discoverer) and supports the development of MDRE tools. It supports use cases of software modernization which are based on MDRE approaches. They define MDRE as *Model Discovery* plus *Model Understanding*. Model Discovery is the phase of representing legacy systems by a set of models without the loss of any information. The Model Understanding phase takes the models from the Model Discovery phase and generates the desired output models based on model transformations. MoDisco is an open source project, which is hosted under the Eclipse Foundation and it is also quoted by the OMG as providing the reference implementations and tooling of several industry standards promoted by the Architecture-Driven Modernization (ADM[2]) task force industry standards. ADM is an initiative of the OMG, which builds and promotes standards dealing with the modernization of legacy systems. Brunelière et al. describe several concrete use cases for MDRE approaches using MoDisco like refactoring of existing Java applications or code quality evaluations.

Brunelière et al. identified four main challenges for an MDRE solution to overcome. First, any loss of information of the heterogeneity of legacy systems must be avoided. As much information as possible must be retrieved. Second, comprehension of the legacy systems must be improved to be able to derive higher abstract views with the most relevant information. Third, the solution must be scalable. MDRE techniques must be able to load, query and transform very large models which are usually involved when dealing with legacy systems. And fourth, MDRE solutions must be generic for different needs. They should not be technology or scenario dependent, or support only one legacy system or technology. A full MDRE approach must have the characteristics of genericity, extensibility, full coverage, reusability, and automation, to adress the mentioned challenges. Therefor it is necessary to switch as far as possible to the world of

[1] MiniJava - http://www.cambridge.org/us/features/052182060X/

[2] Architecture-Driven Modernization - http://adm.omg.org/

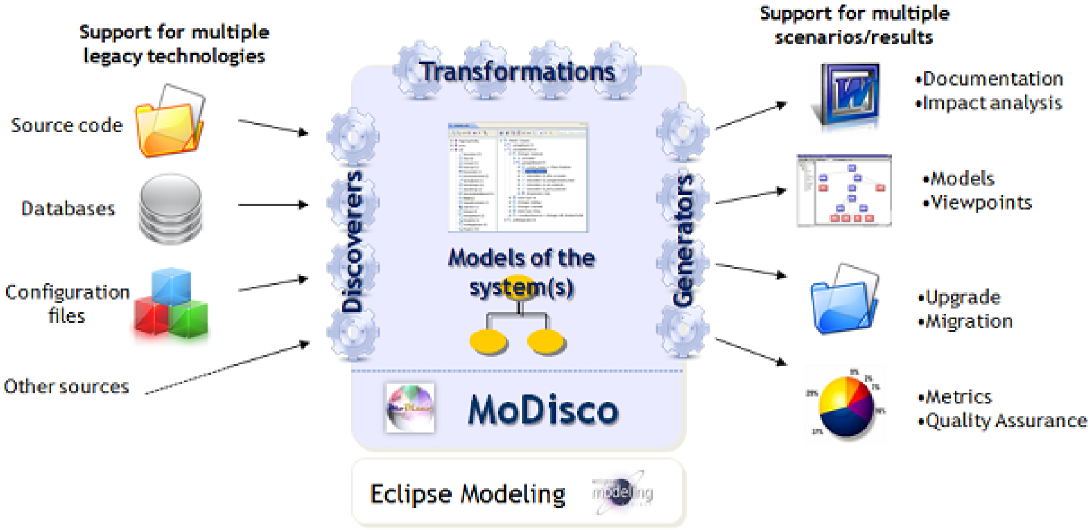


Figure 6.1: Overview of the Eclipse-MDT MoDisco project [7].

models to benefit of MDE and the available technologies and techniques around it. The prototype described within this master's thesis has also been designed with these characteristics in mind. Genericity, extensibility, and re-usability are fostered by the structure of the prototype. Interfaces ensure that each part of the prototype can be exchange. E.g., the C# metamodel is not specific for the reverse engineering of C# code to fUML models and could be used to generate different views on different abstraction levels. Also the resulting fUML model could be used for other purposes like dynamic or static analysis.

To structure the two phases *Model Discovery* and *Model Understanding*, they propose a global architecture of a MDRE framework which consists of the layers *Infrastructure*, *Technologies*, and *Use Cases*. The infrastructure layer is independent of legacy systems and provides generic metamodels and model transformations, e.g., via specific APIs. The *Technologies* layer which is build on top of the *Infrastructure* layer covers some legacy technologies. This could be technology specific metamodels and the corresponding model discoverers, and transformation operations on these models. Examples are model discoverer for Java programs, C# programs, etc., which are representing the legacy system without loosing any information on a very low abstraction level and close to the source technology. This is often realized by an abstract syntax tree. Last but not least, the Use Cases layer provides some simple demonstrations or integration examples, e.g., perform some refactoring on Java code.

Figure 6.1 shows the overview of the Eclipse-MDT MoDisco project. It offers discoverers, transformations, and generators, based on the Eclipse Modeling Framework (EMF)[3]. It supports multiple legacy technologies as input (e.g., source code, databases, etc.), and targets different views and artifacts from the legacy system as output, depending on the expected reverse engineering tasks, by using the different transformations. Due to its extensibility it can be enriched to support specific custom needs. To gain such an extensibility, the Infrastructure layer provides

[3] Eclipse Modeling Framework - https://eclipse.org/modeling/emf/

a set of generic components and supports three concrete OMG ADM standard metamodels, namely Knowledge Discovery Meta-Model (KDM)[4], Structured Metrics Meta-Model (SMM)[5] and Generic Abstract Syntax Tree Meta-Model (GASTM)[6]. KDM deals with legacy systems and their artifacts, and represents the system and its entities in a generic way on a higher level of abstraction. With the use of SMM, it is possible to specify different metrics and measurements on the source legacy systems. GASTM, a subdomain of the Abstract Syntax Tree Meta-Model (ASTM)[7], is a low level abstract syntax tree independent of the source language, which uses UML classes for representing the models. MoDisco has a comprehensive support of KDM, e.g., predefined transformations for obtaining UML class diagrams, which can be used whenever a model discoverer obtains a KDM model.

Although there have been made some experiments about the migration of C# programs within the context of the ARTIST FP7-ICT European project [1], there is no solution for code written in the programing language C# available yet in MoDisco.

6.3 Other Related Work

Bergmayr et al. [3] describe the project *Advanced Software-based Service Provisioning and Migration of Legacy Software (ARTIST)*, which is funded by the European Commission. The goal of the project is a comprehensive software modernization approach based on MDE to migrate legacy software to the cloud. This includes a reverse engineering step consisting of a semi-automatic model discovery as well as a forward engineering step to generate source code deployable in the cloud from the discovered models after having applied transformations to the models incorporating cloud specific features. The fREX framework mentioned above originated from the ARTIST project. Also the *JUMP* framework [4] has been developed within the ARTIST project. JUMP is a model based reverse engineering approach which is capable of reverse engineering Java annotations to UML profiles.

Martinez et al. [15] describe a framework for reverse engineering UML activities. Their approach is based on static and dynamic analysis, metamodeling, and formal specification to be able to recover activity diagrams from Java code. They are using compiler techniques for parsing the source code and extracting an AST. This AST is transformed to a model conforming to a metamodel defined with MOF which represents a simplified abstract Java model. This model is transformed to the algebraic language NEREUS which has a formal semantics. From this model, UML activities are generated using model transformations. The Eclipse Modeling Framework (EMF) is used for the various transformations. They are recovering *action states*, which are deduced from method calls, *swimlanes*, which are inferred from the classes containing the called method, *transitions*, which represent the call flow, and *objects*, which are representing local variables of parameters of an operation. While action states, swimlanes, and objects are obtained by static analysis, transitions are determined by dynamic analysis of execution traces

[4] Knowledge Discovery Meta-Model - http://www.omg.org/spec/KDM/

[5] Structured Metrics Meta-Model - http://www.omg.org/spec/SMM/

[6] Generic Abstract Syntax Tree Meta-Model - http://www.omg.org/spec/ASTM/

[7] Abstract Syntax Tree Meta-Model - http://www.omg.org/spec/ASTM/

of defined test cases. This master's thesis also focuses on UML activities but additionally relies on fUML. Also, instead of Java, C# is considered. Martinez et al. [15] are reverse engineering Java code to UML activity diagram models on a high level of abstraction. Compared to the work within this master thesis, detailed behavior is not represented in the target models. They do not consider the UML action language, hence the behavior of the resulting models is incomplete.

A very similar approach is described in the work of Favre [11] with the focus on UML state machine diagrams. In this work, formal metamodeling techniques are described for maintaining consistency in the reverse engineering process mainly using the Object Constraint Language (OCL). First, a parser generates an AST of the source code according to the grammar of the source language. A data flow graph is built and tracer tools are used to capture system states through dynamic analysis. The focus of the work of Favre [11] is the maintaining of consistency in a reverse engineering process. This master's thesis is using a similar approach as described by Favre [11] but it focuses on UML activity diagrams instead of UML state machine diagrams.

Izquierdo et al. [14] describe a domain-specific language called Grammar To Model Transformation Language (Gra2MoL), which is capable of extracting models from code written in a general purpose language. It uses a program which conforms to a grammar, the grammar itself, the metamodel of the desired reverse engineered output model, and mappings between grammar elements and metamodel elements as input. In the first step they build a syntax tree (either concrete or abstract) from the source code by using ANTLR[8]. The second step is about transforming the syntax tree into the desired output model by using the Gra2MoL textual definition, which defines the mapping between the grammar and the metamodel. The approach also provides a query language inspired by XPath for navigating the transformed model. An example is also given for reverse engineering source code, which is written in the programming language Delphi. Gra2MoL neither targets a specific source language, nor does it define a specific target model formalism. It could be used within an own reverse engineering approach. The work described within this master's thesis targets a specific source language and a specific target formalism, namely C# and fUML. However, Gra2Mol could be used as an alternative approach to define the mapping between C# and fUML.

[8] ANother Tool for Language Recognition - http://www.antlr.org/

CHAPTER 7

Conclusion and Future Work

This chapter summarizes and concludes this work, and gives an outlook to future work.

7.1 Conclusion

Model Driven Engineering (MDE) techniques using models as a central part of the software development process are commonly used today. But not all software systems have been developed by using MDE techniques. Whenever there is the need for software modernization tasks like refactoring, upgrading, reusing parts, etc., reverse engineering (RE) techniques can be used to obtain model-based representations of existing software systems. Such RE techniques may also be required whenever a higher-level representation of a software system is needed but missing. Doing this reverse engineering tasks manually can become cumbersome and error prone. Therefor, model driven reverse engineering (MDRE) techniques have been introduced that automate the reverse engineering process using MDE techniques. However, MDRE techniques for reverse engineering the detailed behavior of software including its algorithmic details are currently still missing.

In 2011, OMG published the specification Semantics of a Foundational Subset for Executable UML Models or fUML in short, which selects a subset of the UML language. fUML defines a precise and complete execution semantics for this subset and therefor serves as a suitable candidate to reverse engineer the complete behavior of an existing software system. When using fUML as target formalism it is possible to model the whole behavior of a software system in a MDRE scenario. However, currently there are no MDRE approaches available which aims to reverse engineer the complete behavior of existing software systems to fUML.

The aim of this thesis was to develop an MDRE approach for reverse engineering software written in C# to fUML. To achieve this, a basic mapping between C# programming concepts and fUML modeling concepts was elaborated. This includes the mapping from C# classes to fUML classes which is needed for reverse engineering the structure of a software system, as well as the mapping from C# method bodies to fUML activities which is needed for reverse engineering the behavior of a software system. The mapping of the behavior is based on the proposed Java to fUML mapping described in the fUML standard. Some limitations of fUML for completely representing the behavior of C# programs have been discovered, such as calls to Clear (clear structural feature action), Add (add structural feature value action), and Remove (remove structural feature action) methods to be used on structural features only, which prevents the definition of lists within a method as well as the addition and removal of list items. Another limitation of fUML is, that it does not allow class properties to have default values. Solutions to overcome these limitations have been proposed in this thesis. A prototypical implementation was elaborated to show the feasibility of the proposed C#-to-fUML mapping. The prototype is able to reverse engineer basic concepts of C# programs to fUML conformant models, which are persisted in the UML modeling environment Enterprise Architect. Furthermore, the architecture of the prototype was designed to be extendable and reusable, to be able to target different source languages or different target UML modeling environments with little effort.

The described approach shows that it is possible to define a basic mapping between C# and fUML. Furthermore, it shows that an MDRE approach is feasible to reverse engineer existing software systems which are written in the general purpose language C# to fUML compliant models. It has also shown that fUML is a suitable candidate as target formalism due to its precise and complete semantics.

7.2 Future Work

In order to provide comprehensive support for reverse engineering code written in the general purpose language C# to fUML conformant models the prototypical implementation that was built within the work of this thesis has to be extended. The following extensions may be the most interesting ones:

- Support further C# concepts
  Currently, the mapping is only defined for the main C# concepts. The mapping has to be defined for further concepts like continue statements, break statements, and exception handling.

- Support of the .NET framework
  In order to be able to fully reverse engineer software systems written in C#, the prototype must be extended to be able to make use of the most common concepts used within the .NET framework. This regards various different data types defined in the .NET framework like different collections and dictionaries, as well as built in methods which are commonly used within C# programs, such as methods operating on primitive types which are not covered by the primitive behaviors defined in the fUML standard and preserve the behavior of the reverse engineered C# program.

- Correct reverse engineering of object flow and control flow
  Besides the lack of support for further C# concepts, the current implementation is not able to set the correct object flows and control flows in every case. This has to be reworked, so that the resulting models are conformant to the fUML standard.

- Support of external libraries
  Currently no external libraries are supported by the elaborated prototype. In order to generate complete fUML models it would be necessary to either reverse engineer all used external libraries if the code is available, or to extend the foundational model library of fUML with such libraries. The latter could be achieved by applying the approach proposed by Neubauer et al. [19], who show how existing software libraries could be integrated with fUML compliant UML models at design time, as well as at runtime during model execution.

- Support of other source languages
  Another interesting future work would be to support other programming languages like Java as source language. Due to the structure of the prototype by using components with dedicated interfaces it would be a feasible and interesting task.

APPENDIX A

Example Code

The listings in this section show the complete code of the running example introduced in Section 3.1.

Listing A.1: Code of the Student class

```
public class Student
{
 public string FirstName { get; set; }
 public string LastName { get; set; }
 public int MatNr { get; set; }
 public University University;
 private int age = 18;

 public int Age
 {
  get { return this.age; }
  set { this.age = value; }
 }

 public Student()
 {
 }

 public Student(int matNr)
 {
  MatNr = matNr;
 }

 public Student(string firstName, string lastName, int matNr)
 {
  FirstName = firstName;
  LastName = lastName;
  MatNr = matNr;
 }
}
```

Listing A.2: Code of the University class

```
public class University
{
 public string Name { get; set; }
 private List<Student> students = new List<Student>();

 public List<Student> Students
 {
  get { return students; }
  set { students = value; }
 }

 public University()
 {
 }

 public University(string name)
 {
  this.Name = name;
 }

 // Adds a new student
 public void AddStudent(Student student)
 {
  Students.Add(student);
 }

 // Inserts a new student on the given position
 public void InsertStudentAt(Student student, int position)
 {
  Students.Insert(position, student);
 }

 // Removes a student
 public void RemoveStudent(Student student)
 {
  Students.Remove(student);
 }

 // Removes a student on the given position
 public void RemoveStudentAt(int position)
 {
  Students.RemoveAt(position);
 }

 // Clears all students
 public void ClearAllStudents()
 {
  Students.Clear();
 }
}
```

Listing A.3: Code of the IAdministration interface

```
public interface IAdministration
{
 University CreateUniversity(string name);
}
```

Listing A.4: Code of the Administration class

```
public class Administration : IAdministration
{
 public List<University> universities = new List<University>();

 // Creates a new University object
 public University CreateUniversity(string name)
 {
  University uni = new University();
  uni.Name = name;
  universities.Add(uni);
  return uni;
 }

 // Creates a new Student object
 public Student CreateStudent()
 {
  return new Student(123456);
 }

 // Creates a new Student object for a university
```

```
public Student CreateStudentForUniversity(University uni)
{
 Student student = new Student(123456);
 uni.Students.Add(student);
 return student;
}

// Checks if the parameter is a Student
public bool IsStudent(object student)
{
 return student is Student;
}

// Checks if two students are equal
public bool AreStudentsEqual(Student student1, Student student2)
{
 return student1 == student2;
}

// Checks if the student is null
public bool TestForNull(Student student)
{
 return student == null;
}

// Checks if a student is full aged
public bool IsStudentOfFullAge(Student student)
{
 return IsStudentOlderThan(student, 17);
}

// Compares if the students age is greater than the
// given age
private bool IsStudentOlderThan(Student student, int age)
{
 return student.Age > age;
}

// Sets the university on the student
public void SetUniversityOnStudent(Student student, University uni)
{
 student.University = uni;
}
```

```
// Does some calculation to show if-else statement
// while loop and switch-case statement
public int CalculateYear(int nr)
{
 int magicNumber = 23;
 if (nr > magicNumber)
 {
  magicNumber = magicNumber + 10;
 }
 else
 {
  magicNumber = magicNumber + 20;
 }
 while (magicNumber > 0)
 {
  magicNumber = magicNumber - 1;
 }
 switch (nr)
 {
  case 1:
   magicNumber = 14;
   break;
  case 10:
   magicNumber = 24;
   break;
  default:
   magicNumber = magicNumber + 1;
   break;
 }
 return magicNumber;
}

// Sets the MatNr to zero on all students on the
// given university
public void ResetMatNrOnAllStudents(University uni)
{
 for (int i = 0; i < uni.Students.Count; i++)
 {
  uni.Students[i].MatNr = 0;
 }
}
```

```
 // Sets the MatNr to zero on all students on the
 // given university using foreach
 public void ResetMatNrOnAllStudentsIterative(University uni)
 {
  foreach (Student student in uni.Students)
  {
   student.MatNr = 0;
  }
 }
}
```

Bibliography

[1] ARTIST European Project. Online available at: http://www.artist-project.eu/, Accessed: 2017-03-07.

[2] Alexander Bergmayr, Hugo Brunelière, Jordi Cabot, Jokin García, Tanja Mayerhofer, and Manuel Wimmer. fREX: fUML-based reverse engineering of executable behavior for software dynamic analysis. In *2016 IEEE/ACM 8th International Workshop on Modeling in Software Engineering (MiSE'16)*, pages 20–26. IEEE, 2016.

[3] Alexander Bergmayr, Hugo Brunelière, JL Cánovas Izquierdo, Jesus Gorronogoitia, George Kousiouris, Dimosthenis Kyriazis, Philip Langer, Andreas Menychtas, Leire Orue-Echevarria, Clara Pezuela, and Manuel Wimmer. Migrating Legacy Software to the Cloud with ARTIST. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13)*, pages 465–468. IEEE, 2013.

[4] Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. JUMP - From Java Annotations to UML Profiles. In *Proceedings of the 17th International Conference on Model-Driven Engineering Languages and Systems (MODELS'14)*, volume 8767 of *Lecture Notes in Computer Science*, pages 552–568. Springer, 2014.

[5] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[6] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[7] Hugo Brunelière, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. MoDisco: a Model Driven Reverse Engineering Framework. *Information and Software Technology*, 56(8):1012–1032, 2014.

[8] Uwe Brunflicker. Integrating fUML into Enterprise Architect. Master's thesis, Vienna University of Technology, 2015. Online available at: http://resolver.obvsg.at/urn:nbn:at:at-ubtuw:1-80420.

[9] Gerardo Canfora and Massimiliano Di Penta. New Frontiers of Reverse Engineering. In *Proceedings of the 2007 Workshop on the Future of Software Engineering (FOSE'07)*, FOSE '07, pages 326–341. IEEE Computer Society, 2007.

[10] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and Challenges in Software Reverse Engineering. *Communications of the ACM*, 54(4):142–151, 2011.

[11] Liliana Favre. Formalizing MDA-based reverse engineering processes. In *Proceedings of the 6th International Conference on Software Engineering Research, Management and Applications (SERA'08)*, pages 153–160. IEEE, 2008.

[12] Martin Fleck, Luca Berardinelli, Philip Langer, Tanja Mayerhofer, and Vittorio Cortellessa. Resource Contention Analysis of Service-Based Systems through fUML-Driven Model Execution. *Proceedings of the 5th International Workshop Non-functional Properties in Modeling (NiM-ALP'13)*, page 6, 2013.

[13] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quaterly*, 28(1):75–105, 2004.

[14] Javier Luis Cánovas Izquierdo and Jesús García Molina. Extracting Models from Source Code in Software Modernization. *Software & Systems Modeling*, 13(2):713–734, 2012.

[15] Liliana Martinez, Claudia Pereira, and Liliana Favre. Reverse Engineering Activity Diagrams from Object Oriented Code: An MDA-Based Approach. *Computer Technology & Application*, 2(11):969–978, 2011.

[16] Tanja Mayerhofer, Philip Langer, and Gerti Kappel. A runtime model for fUML. In *Proceedings of the 7th Workshop on Models@run.time*, pages 53–58. ACM, 2012.

[17] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.2, 2015-02-01. Online available at: `http://www.omg.org/spec/QVT/1.2/PDF/`.

[18] Microsoft. .NET Compiler Platform ("Roslyn"). Online available at: `https://roslyn.codeplex.com/`, Accessed: 2014-11-06.

[19] Patrick Neubauer, Tanja Mayerhofer, and Gerti Kappel. Towards Integrating Modeling and Programming Languages: The Case of UML and Java. In *Proceedings of the 2nd International Workshop on The Globalization of Modeling Languages*, pages 23–32, Vol-1236, 2014. CEUR.

[20] Object Management Group. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, 2013-06-01. Online available at: `http://www.omg.org/spec/MOF/2.4.1/PDF/`.

[21] Object Management Group. OMG Unified Modeling Language^{TM} (OMG UML), Infrastructure, Version 2.4.1, 2011-08-05. Online available at: `http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/`.

[22] Object Management Group. OMG Unified Modeling Language^{TM} (OMG UML), Superstructure, Version 2.4.1, 2011-08-06. Online available at: `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/`.

[23] Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, 2007.

[24] D.C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[25] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.0, 2011-02-01. Online available at: `http://www.omg.org/spec/FUML/1.0/PDF/`.

[26] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.1, 2013-08-06. Online available at: `http://www.omg.org/spec/FUML/1.1/PDF/`.

[27] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.3 Beta, 2017-02-01. Online available at: `http://www.omg.org/spec/FUML/1.3/PDF/`.

[28] Sparx Systems Pty Ltd. Enterprise Architect. Online available at: `http://www.sparxsystems.com.au/products/ea/index.html`, Accessed: 2014-11-18.

[29] Tarja Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, University of Tampere, 2000. Online available at: `http://urn.fi/urn:isbn:951-44-4811-1`.

[30] Paolo Tonella and Alessandra Potrich. *Reverse Engineering of Object Oriented Code*. Springer-Verlag New York, 1st edition, 2005.