Automatic Code Generation using Swarm Intelligence

Submitted by:

Hina Mahmood
297-FBAS/MSSE/F09

Supervised by:

Mr. Atif Aftab Ahmed Jilani

Co-Supervised by:

Mr. Qaisar Javaid

Department of Computer Science and Software Engineering
Faculty of Basic and Applied Sciences
International Islamic University Islamabad
(June 2012)

1- Code generators ; Computer science

2- Computer software ; Development

Department of Computer Science and Software Engineering

International Islamic University Islamabad

Date: 11-06-2012

Final Approval

This is to certify that we have read the thesis submitted by **Hina Mahmood, 297-FBAS/MSSE/F09**. It is our judgment that this thesis is of sufficient standard to warrant its acceptance by International Islamic University, Islamabad for the degree of **Masters of Science in Software Engineering (MSSE).**

Committee:

External Examiner:

Dr. Arshad Ali Shahid
Professor, HOD
Department of Computer Science
National University of Computer and
Emerging Sciences (NUCES) - FAST

Internal Examiner:

Dr. Zunera Jalil
Assistant Professor
Department of Computer Science and
Software Engineering
International Islamic University, Islamabad

Supervisor

Mr. Atif Aftab Ahmed Jilani
Assistant Professor
Department of Computer Science
National University of Computer and
Emerging Sciences (NUCES) - FAST

Co-supervisor

Mr. Qaisar Javaid
Assistant Professor
Department of Computer Science and
Software Engineering
International Islamic University

Dedication...

To my family especially my mom

who is an embodiment of Diligence and Honesty.

without her Prayers and Support

this dream could have never come true.

A dissertation Submitted to

Department of Computer Science and Software Engineering,

Faculty of Basic and Applied Sciences,

International Islamic University, Islamabad

As a Partial Fulfillment of the Requirement for the Award of the

Degree of *Masters of Science in Software Engineering (MSSE).*

Declaration

I hereby declare that this thesis ***"Automatic Code Generation using Swarm Intelligence"*** neither as a whole nor as a part has been copied out from any source. It is further declared that I have done this research with the accompanied report entirely on the basis of my personal efforts, under the proficient guidance of my teachers especially my supervisor *Mr. Atif Aftab Ahmed Jilani*. If any part of the system is proved to be copied out from any source or found to be reproduction of any project from any of the training institute or educational institutions, I shall stand by the consequences.

Hina Mahmood

297-FBAS/MSSE/F09

Acknowledgement

In the name of **Allah**, the passionate, whose blessings made it possible for me to complete this complex and hard task. Its completion is a matter of great enthusiasm and pleasure for me. It is all because of Almighty Allah's guidance that made me so able.

Every mission and project has a brain behind that vivifies the theoretical raw ideas. I am fortunate enough that a masterful intellect, in the mind of my supervisor Mr. Atif Aftab Ahmed Jilani, was with me. I offer my sincerest gratitude to him, who has supported me throughout my thesis with his patience and knowledge whilst allowing me the room to work in my own way. My thesis would have been a complete fiasco in the absence of such a mastermind. I attribute the level of my Masters degree to his encouragement and effort. I have no words to thank the laborious and tiring contributions of this extraordinary personality. One simply could not wish for a better supervisor.

I wish to express my deepest gratitude to Mr. Qaisar Javaid for his worthy support and kind cooperation particularly near the completion of my thesis. I would also like to acknowledge Dr. Abdul Rauf for his genuine support, valuable advice, sincere comments and motivation throughout the project. He regularly gave his precious time to this thesis despite of his tight schedules. I thank the members of my graduate committee for their worthy comments and valuable criticism. I am also grateful to my friends and colleagues for their love and encouragement.

It will be failing in my duties if I miss to thank my beloved family. I am indebted to my parents and would like to express my deepest gratitude to them for their constant encouragement, affection and motivation. Their prayers always contribute a lot in completing difficult tasks. It is due to their unexplainable care and love that I am at this position today. I am thankful to my caring brothers especially Dr. Salman Mahmood for constantly helping me during hard times and offering me his valuable advice. Thanks are also due to my brother Imran for his support and assistance. My brother Irfan deserves my special appreciation for providing me his amusing companionship during my tight and strained schedules, after which I always feel fresh and

relaxed. Last but not least, I am particularly thankful to my sweet sister-in-law Aysha for her love, care and valuable discussions.

Every work is bound to flaws. I accept complete responsibility for all flaws in this dissertation. I shall be grateful for valuable suggestions and all positive criticism will be welcomed.

Hina Mahmood
297-FBAS/MSSE/F09

Project In Brief

Project Title:	Automatic Code Generation using Swarm Intelligence
Undertaken By:	Hina Mahmood 297-FBAS/MSSE/F09
Supervised By:	Mr. Atif Aftab Ahmed Jilani
Start Date:	December 01, 2010
Completion Date:	January 31, 2012
Tools & Technologies	Java™ SE Development Kit 7 Visual Paradigm for UML 7.0 Microsoft Office Visio 2003 Rational Rhapsody Developer V7.6
Documentation Tools	Microsoft Office Word 2007 Endnote X Volume License Edition Pdf995 Suite
Operating System:	Microsoft Windows 7, Home Premium
System Used:	HP ProBook 4530s Notebook PC

Abstract

Automatic code generation refers to the mechanical generation of implementation code from system design models by executing a set of transformation rules. Existing approaches for model-to-code (M2C) transformation assume these rules to be manually defined by the domain experts, by exploiting the source and target languages' metamodels and expressed in some model transformation language. However in reality, the definition, maintenance and evolution of a complete, correct, consistent and non-redundant transformation rule set is not an obvious task, especially in the availability of little domain knowledge. The complex nature of metamodels and transformation languages further aggravates the situation, making the code generation process complicated and time-consuming.

It has been observed that many organizations maintain a record of their past M2C transformations and feel more comfortable to show these transformation examples instead of defining a complete and consistent transformation rule set. Our work starts from these observations to view automatic code generation problem as the one to solve with fragmentary knowledge i.e. with only examples of M2C transformations. In this research, we present a novel approach for automatic code generation by utilizing the previously solved M2C transformation problems. We view M2C transformation as an optimization problem and select the best solution from all feasible solutions. The existing set of examples is used to train the system regarding automatic code generation. After the system is trained transformation blocks, that best match the constructs of the input source model to be transformed, are extracted from the transformation examples. These transformation blocks are then used to translate the source model constructs into target code. An optimal transformation solution is searched by utilizing the heuristic search technique Particle Swarm Optimization. We implemented this approach in a tool named CØde Ṣwarm.

This approach is generic and its application is not limited to any set of models. However as a proof of concept, we have applied this approach for generating Java code from class model and state model, as these two models are representatives of both the static structure and the dynamic system behavior. Experiments performed for the validation of this approach indicate

that up to 100% correct code can be generated. However, the only prerequisite of this approach is to have a set of previously solved transformation examples. Besides them, no extra information is needed. Furthermore, we can easily start with a small and non-exhaustive set of transformation examples, no special expertise are required. Moreover, the proposed approach always proposes a transformation strategy, nearest match in case if no exact match exists in the training data. This is rather impossible in the existing code generation approaches in which the absence of a rule results in a failure to perform the corresponding transformation.

Conclusively, our proposed approach does not rely on using an explicitly defined transformation rule set for performing the transformations, rather it is intelligent enough to automatically derive the rules from the existing set of transformation examples. In the absence of an explicit transformation rule set definition, automatic code generation process becomes independent of transformation languages. Moreover, by automatic extraction of transformation rules from previously solved transformation examples, our approach also becomes independent of source and target formalisms. In this way, our proposed approach makes the code generation process painless by dissociating it from explicit transformation rule set definition, its expression and metamodels' comprehension.

Table of Contents

List of Figures

List of Tables

List of Publications

1. **Hina Mahmood**, Atif Aftab Ahmed Jilani, Abdul Rauf, "A Lightweight Framework for Automated Model-to-Code Transformation", in Proc. of the 14th IEEE International Multitopic Conference (INMIC), pp. 279-283, Dec. 22-24 2011

2. **Hina Mahmood**, Atif Aftab Ahmed Jilani, Abdul Rauf, "Code Swarm: A Code Generation Tool Based on Automatic Derivation of Transformation Rule Set", accepted in the 9th International Conference on Information Technology: New Generations (ITNG), April 2012, USA

3. **Hina Mahmood**, Atif Aftab Ahmed Jilani, Abdul Rauf, "An Optimization Approach for Automatic Code Generation using Swarm Intelligence", under-review in the Journal of Systems and Software (JSS)

Acronyms and Abbreviations

ASL	Action Specific Language
CIM	Computation Independent Model
ECS	Elevator Control System
EHA	Extended Hierarchical Automata
Fujaba	From UML to Java And Back Again
FXU	Framework for eXecutable UML
GReAT	Graph Rewriting and Transformation
IE	Inference Engine
KBS	Knowledge Base System
M2C	Model-to-Code
M2M	Model-to-Model
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MOTOE	Model Transformation as Optimization by Examples
OCL	Object Constraint Language
OMG	Object Management Group
OMT	Object Modeling Technique
OO	Object Oriented
PIM	Platform Independent Model
PLC	Programmable Logic Controller
PSI	Platform Specific Implementation
PSM	Platform Specific Model
PSO	Particle Swarm Optimization
rCOS	Relational Calculus of Object Systems
SMC	Source Model Construct
STD	State Transition Diagram
TCC	Target Code Construct
UJECTOR	Uml to Java Executable Code GeneraTOR
UML	Unified Modeling Language
VUML	View-based UML
XMI	XML Metadata Interchange
xUML	Executable UML

Chapter 1

INTRODUCTION

1.1 Introduction

Within software engineering, implementation phase is considered as the core activity of developing a software system. However, the advent of Model Driven Engineering (MDE) brought a paradigm shift in the history of software engineering by changing it from a code-centric to a model-centric activity. MDE focuses on developing a software system by performing a series of model transformations to generate target code, referred to as Platform Specific Implementation (PSI).

The goal of software engineering is to produce a quality software product in a faster and cheaper way [2]. As a result, automatic model-to-code (M2C) transformation remains an actively explored research area as it makes the activity of software development more efficient, productive and less error-prone. The existing approaches and tools for M2C transformation rely on three fundamental building blocks; 1) transformation rules, 2) transformation languages and 3) source and target metamodels. Currently, program code can be automatically generated from system models by executing a set of transformation rules, which are defined on the base of source and target languages metamodels and expressed in a model transformation language.

1.2 Problem Statement

Due to various benefits offered by automatic code generation, it remains an extensively explored research area for the last few years. Consequently, literature is stuffed with approaches to automatically generate source code from system's design artifacts. By analyzing these approaches, it can be concluded that all existing approaches are based on transformation rules, which define the mapping between source and target metamodels.

As for any rule-based system, defining a set of transformation rules is not an obvious task and many unwanted limitations confine the results [3]. Some of these limitations are: 1) Some transformations cannot be easily expressed in the form of rules [4]. 2) In some cases, the availability of httle domain knowledge also hinders the way of defining a complete transformation rule set [3]. 3) Furthermore, experts may find it difficult to master both the source and target metamodels [5]. 4) Due to the availability of a wide range of transformation languages for expressing rules, it often becomes complex for experts to choose the one that best serves the

needs of their domain. All these difficulties are amplified when we consider that transformation rule set may evolve. During the evolution, addition and adaptation of transformation rules makes it difficult to ensure their consistency and correctness. Due to all these reasons, the development of a correct, complete, consistent and non-redundant transformation rule set becomes a complex and demanding activity.

More specifically, the definition of transformation rules is a human-dependent activity at the moment and expert intervention is needed during the whole process. There is no method to automatically derive or extract transformation rules without human intervention. No such approach currently exists that can perform the task of automatic code generation: 1) without explicit transformation rule set definition, 2) independent of transformation languages and 3) irrespective of source and target metamodels.

1.3 Motivation

It is recognized that experts can more easily give transformation examples instead of defining complete and consistent transformation rules [6]. In most cases, the companies have accumulated knowledge from past industrial transformation examples [4, 7]. However, currently there is no way to automatically extract transformation rules from these previously accumulated transformations and utilize them to solve new M2C transformation problems.

From these observations, our work starts to view automatic code generation problem as the one to solve with fragmentary knowledge i.e. with only examples of M2C transformations. In this case, there should be some procedure to automatically derive transformation rules from existing set of industry-based transformation examples. The automatic extraction of transformation rules will circumvent the manual definition of transformation rule set by domain experts. It will make the M2C transformation process independent of source and target formalisms. In this way, all the difficulties associated with manual definition, maintenance and expression of a complete, consistent and non-redundant transformation rule set will be eradicated.

1.4 Research Questions

This research aims to address the following three research questions:

1. How to perform the process of M2C transformation without explicitly defining transformation rules?
2. How to make the automatic code generation process independent of transformation languages?
3. How to generate code from models irrespective of source and target languages' metamodels?

1.5 Proposed Solution

Keeping in view all the problems and complexities associated with the existing code generation approaches, we aim to propose an approach that can make the task of automatic code generation simple, easy and unproblematic. The core theme of our approach is to use knowledge from previously solved industrial transformation examples to solve new M2C transformation problems.

We propose to use a Knowledge Base System (KBS) for M2C transformation. The set of existing transformation examples are used to train the system regarding automatic code generation. After the system is trained, new M2C transformations are performed. Thus, instead of providing transformation rules directly as input, our aim is just to provide an existing set of transformation examples and let the system automatically extract transformation rules from them, without any expert intervention. Besides transformation examples, no extra information is needed to utilize this approach.

In the absence of an explicit transformation rule set definition, automatic code generation process becomes independent of transformation languages. Moreover, by automatic extraction of transformation rules from previously solved transformation examples, our approach also becomes independent of source and target formalisms. In this way, our proposed approach makes the process of automatic code generation independent of manual transformation rule set

definition, transformation languages and metamodels. However, the only prerequisite of this approach is to have a set of previously solved transformation examples to be used as the training data.

1.6 Thesis Contributions

This dissertation introduces a novel approach developed for automatic code generation from system design models. Our approach makes use of the existing transformation examples, heuristic search and swarm intelligence to automate the process of code generation. To the best of our knowledge, these concepts have never been used in the context of M2C transformation before, making it significantly different from what already exists in current literature regarding M2C transformation. The core characteristics of this approach are as follows.

1. It can utihze previously solved transformation examples to solve new M2C transformation problems.
2. It can transform models into code without explicit definition of transformation rules.
3. It does not rely on the metamodels of source and target languages.
4. It is irrespective of any transformation language.

The benefits achieved by developing an approach having the above-mentioned characteristics are given below.

1. Ease the process of transformation from models to code.
2. Utilize the existing fragmentary knowledge in solving new M2C transformation problems.
3. Transform models into code without using transformation languages.
4. Generate code from models without explicitly writing transformation rules.
5. Make the transformation process independent of metamodels complexities.
6. Our approach always proposes a transformation strategy, nearest match if no exact match is found in the training data, which is rather impossible in the existing rule-based approaches.

7. Accelerate the process of M2C transformation by eradicating the need of learning complex technologies.

1.7 Dissertation Outline

Figure 1.1 illustrates the organization of this dissertation: Chapter 2 establishes the background for understanding the dissertation by providing an introductory knowledge regarding M2C transformation. Chapter 3 presents the related work in the fields of automatic code generation and swam intelligence. The issues and limitations of existing code generation approaches and research gaps are highlighted in Chapter 4. Our proposed approach for automatic M2C transformation is described in Chapter 5. Chapter 6 introduces the tool based on the implementation of our approach. Case study used for the validation of proposed approach is explained in Chapter 7. Chapter 8 discusses and evaluates the results of our research work. Finally, chapter 9 concludes this dissertation and presents the findings of our research work.

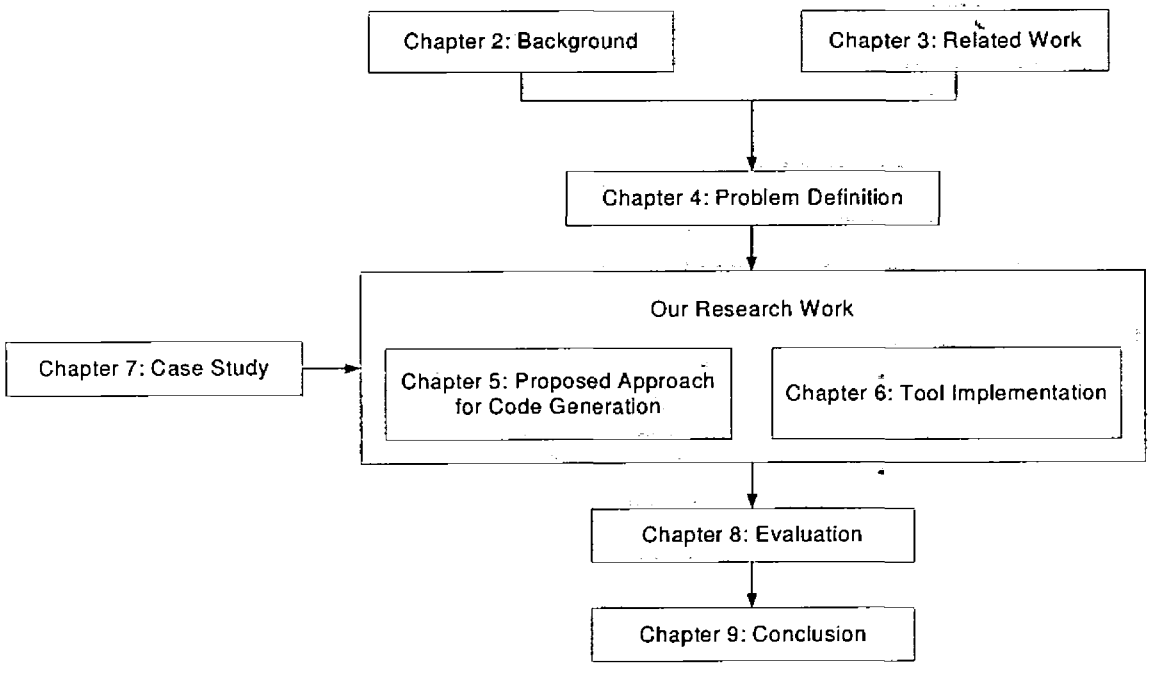


Figure 1.1 Dissertation outline

Chapter 2

BACKGROUND

2.1 Introduction

This chapter is dedicated to establish the background for understanding the dissertation. Section 2.2 presents a brief description of Model Driven Architecture. An explanation of major building blocks of code generation process is provided in Section 2.3. Section 2.4 introduces the concept of swarm intelligence. Finally, Section 2.5 covers the topic of action language, particularly Action Specification Language (ASL).

2.2 Model Driven Architecture

MDE, first proposed by Kent in [8], is a promising approach that raises the level of abstraction of program specification by using models as the major driving objects throughout the software engineering life cycle. Model Driven Architecture (MDA) defined by Object Management Group (OMG) [9, 10] in 2000, is the best realization of MDE principles. By keeping the application and implementation logic separate, MDA provides the advantage of realizing the same system model on multiple platforms [11], thus allowing the reuse of models over a software lifespan. Models and model transformation form the basis of MDA.

2.2.1 Model

MDE is a model-centric software engineering approach [12]. In MDE system, a model is composed of a complete and consistent set of formal elements describing a software system that is amenable to analysis [13]. Models representing a software system should be expressed in a well-defined modeling language. In 1997, OMG defined Unified Modeling Language (UML) [14], which quickly became the de facto industry standard for the design and specification of object-oriented (OO) software systems [15]. MDA defines three classes of models: Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM) [16].

2.2.2 Model Transformation

A central concept of MDA is model transformation [17]. According to OMG [11] , model transformation is the process of translating one model to another model of the same system. In

Model Driven Development (MDD) of software systems, CIM is transformed into a PIM and PIM is transformed into one or more PSMs. These PSMs are finally used to generate target code. In this way, the whole software development process can be seen as a series of model transformations, where one source model is transformed into one or more target models. Consequently, model transformation has been considered as the heart and soul of MDD by Sendall and Kozaczynski in [18].

At the most abstract level, model transformation can be divided into two main categories; 1) model-to-model (M2M) transformation and 2) model-to-code (M2C) transformation (automated code generation) [19, 20]. In MDA, automated transformations are performed using transformation tools. These tools work on the basis of transformation definitions. Figure 2.1 shows the process of MDA with transformation definition and transformation tool incorporated into it.

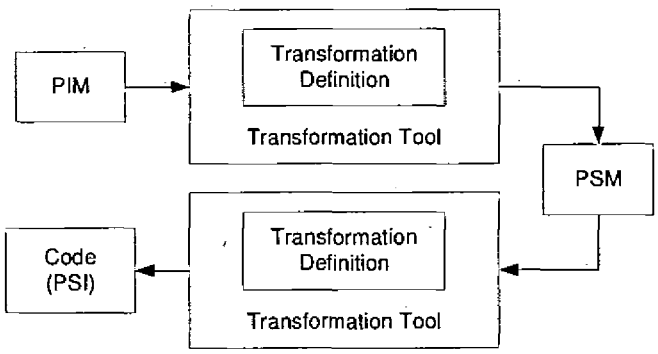


Figure 2.1 MDA process (*adopted from [10]*)

2.3 Code Generation

The most significant use of a model representing a software system is code generation [21]. Manual transformation of system models into code is a very time-consuming and tedious task. Produced code may not be fully compliant with the models due to chances of human error. Consequently, researchers have developed approaches to automate this process, resulting in increased productivity, improved efficiency and reduced errors. The existing approaches of automatic code generation are based on three fundamental building blocks; 1) metamodels of source languages and target languages, 2) transformation rules and 3) transformation languages.

2.3.1 Metamodels

Metamodel defines the semantics, structure and constraints of a language for a family of models [22]. In simple words, a metamodel is a model of the modeling language. A system model expressed in some modeling language, e.g. in UML, is said to conform to its metamodel if each and every element in UML model is a valid class in UML's metamodel. As the code generation process aims at transforming elements of modeling language's metamodel into programming language's metamodel, therefore the comprehension of these two metamodels is vital for M2C transformation.

2.3.2 Transformation Rules

The entire process of generating code from system models is based on defining and using transformation rules. These rules define the mapping between the source modeling and the target programming languages [23]. In M2C transformation, the correctness and quality of generated code depends upon the quality of the transformation rule set. For the correct transformation of system models into code, defined rule set should have the following characteristics:

1. Transformation rule set should be complete i.e. for every element of source language's metamodel, a rule should be defined for its transformation into target metamodel's element.
2. All transformation rules must be consistent with each other and there should not be any two conflicting rules in the transformation rule set.
3. Rules set should be non-redundant i.e. there should not exist two or more different rules for transforming same source metamodel's element into a different target metamodel's element, under same circumstances.
4. Rules should be syntactically correct i.e. every element of source metamodel should be transformed into a valid target metamodel's element.
5. Transformation rules should be semantically correct i.e. the transformation rules should preserve the meaning of the source model.

2.3.3 Transformation Languages

In the field of model transformation, the major contributions are concerned with the definition of transformation languages which are utilized to state the transformation rules [4]. Transformation rules can be expressed using general-purpose programming languages like Java or C#, graph-transformation languages like AGG, VIATRA, TGG, VMTS and model transformation languages like ATL (Atlas Transformation Language), Kermeta, GReAT (Graph Rewriting and Transformation), MOLA and QVT [4, 7].

2.3.4 Code Generation Process

The complete code generation process is shown in Figure 2.2. Based upon the elements of source modeling and target programming languages' metamodels, transformation rules are developed. These transformation rules, expressed in a model transformation language, are fed into the transformation tool. In the automatic code generation process, transformation tool takes system model as input and applies transformation rules to generate the target implementation code.

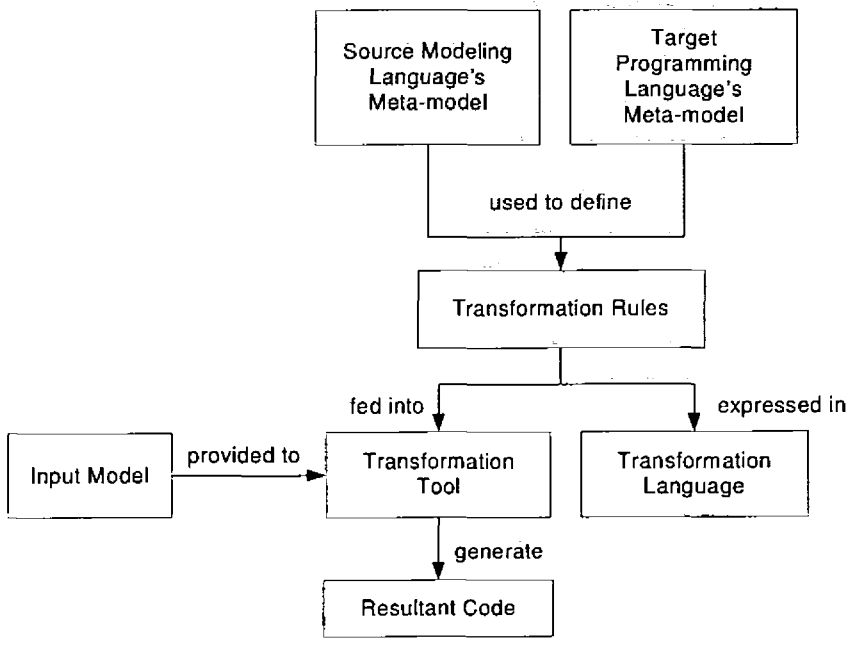


Figure 2.2 Traditional code generation process

2.4 Swarm Intelligence

Swarm intelligence refers to a kind of problem-solving ability that emerges in the interactions of simple information processing-units [24]. These units are autonomous agents, as there is no leader or global plan to follow. However, the units must interact and cooperate with each other so as to achieve a common goal collectively, as this goal is difficult to achieve for any unit individually. Several algorithms have been designed on the base of swarm intelligence including Particle Swarm Optimization (PSO), Ant Colony Optimization, Honeybees algorithm, etc. The aim of these algorithms is to search for an optimal solution in the search space of a given problem. From all algorithms utilizing swarm intelligence, our prime focus is on PSO.

2.4.1 Particle Swarm Optimization

PSO is a stochastic-based optimization technique originally proposed in 1995 [24]. The main idea of this algorithm is to iteratively improve the optimal solution with respect to a given measure of quality. PSO is inspired by the social behavior of bird flocking and fish schooling. In PSO, a population of candidate solutions is called a *swarm* and each individual solution is called a *particle*. As there are many birds in a flock, similarly the swarm is composed of multiple particles. In a flock, each single bird adjusts its movement (position) by coordinating with its flock mates. Likewise, particles fly in the swarm by updating their velocities and positions.

Table 2.1 summarizes the steps of PSO algorithm. PSO starts by initializing the swarm with randomly generated particles. These particles are placed in the search space of a problem. The fitness of particle at its current location is calculated according to a user-defined objective function. Each particle has a memory which keeps a record of the best position achieved by the particle until now in the search space. This is known as p_{best}. Overall, the swarm also remembers the best position achieved so far by any particle in the search space. This is known as g_{best}.

In every iteration, each particle updates its position and velocity by combining some aspect of the history of its own current and best locations with those of other members of the swarm [25]. The number of particles in the search space, the number of iterations, and some other parameters are problem-dependent and are set by the user. The values of these parameters

have a significant influence on the performance, efficiency and ability of PSO to search an optimal solution.

Table 2.1 Steps of PSO

1. Randomly initialize particle positions and velocities
2. While not terminate
   - A. For each particle i
     - *a.* Evaluate fitness at current position x_i
     - *b.* If fitness is better than p_{best}, update p_{best} and p_i
     - *c.* If fitness is better than g_{best}, update g_{best} and p_g
   - B. For each particle
     - *d.* Update velocity v_i and position x_i

2.5 Action Language

Executable UML (xUML) bridges the semantic gap between the UML design models and implementation [26] by the use of an action language. Action language is a high-level implementation-independent language that provides control logic and manipulation of the UML structural models. They present a way to build complete system models by incorporating logical actions at the UML level of abstraction. From among a vast variety of available action languages, this research work uses Action Specification Language (ASL).

2.5.1 Action Specification Language

ASL [27] was developed in 1993 with the aim of providing an unambiguous, concise and readable definition of the processing to be carried out by an OO system. Since then, it has been successfully applied for specifying and developing many small and large-scale software systems ranging from embedded controllers to distributed databases. It is a rich language that is capable of specifying all the processing required. Because of its simplicity and readability, any human reader can easily and quickly scan through it.

Chapter 3

RELATED WORK

3.1 Introduction

The use of models in the development of software has a rich history [28]. In the planet of research, automatic code generation remains an actively explored area by researchers for the last few years. Consequently, a plethora of approaches, techniques and tools are currently available to automatically generate executable code from system's analysis and design artifacts.

This chapter of dissertation is devoted to discuss the existing work in this field. Keeping in view the title of the thesis, our work can be associated to two areas of software engineering – automatic code generation and the application of swarm intelligence to model transformation. Section 3.2 presents the code generation approaches, techniques and research based-tools from existing literature. Some prominent commercial and open source tools for M2C transformation are discussed in Section 3.3. Section 3.4 discusses transformation of models in the light of swarm intelligence. Finally, Section 3.5 concludes the chapter with an analysis of the common traits of existing approaches and tools.

3.2 Code Generation Approaches

Initially, researchers focused on specifying mapping between individual UML models and target language. With the passage of time, they developed approaches for transforming a set of models into source code, so as to generate maximum implementation code automatically. We categorize the existing literature on the bases of models covered by these approaches and tools.

3.2.1 Class Model Transformation Approaches

Favre et al. [29] presented an approach to map static design artifacts particularly UML class diagram to OO code. Annotations in class diagram were expressed using OCL. Initially class diagram is converted into GSBL$^{\infty}$ specification, an OO algebraic language. This specification is then converted into a more complete specification using SpReIm library. Nassar et al. [30] presented a model-driven technique to generate code in VUML (View-based UML) profile. VUML metamodel extends UML with OCL rules. In the proposed approach, VUML PIM is transformed into an OO PSM, which is then used to generate OO code in Java. The transformation rules are specified and implemented using ATL.

3.2.2 State Model Transformation Approaches

Prior to MDE, Weg et al. [31] presented a method named ***CORSO***, "CASE tool support for real-time systems design" to generate PLC (Programmable Logic Controller) code from State Transition Diagram (STD) in two steps - first STDs are translated to an intermediate PLC independent textual code, which is then used to generate PLC specific code.

Ali and Tanaka [1] presented an approach to automatically convert dynamic model, represented as OMT (Object Modeling Technique), into implementation code. OMT is a predecessor of UML statechart diagram. The proposed approach focuses on generating Java code by applying transformation rules. This approach has been implemented in a tool called ***O-Code***. A few years later, this approach was extended to generate code from UML activity diagrams [32]. ***dCode*** is a tool that implements the enhanced approach. Knapp and Merz [33] presented a set of tools called ***Hugo*** to generate Java code from state machines. A generic set of Java classes provides a standard runtime component state for UML state machines.

In 2003, for the first time, design patterns were used for transforming statecharts into Java code. Niaz and Tanaka [23] proposed a novel approach based on ***State design pattern***. Their approach extends the State pattern and focuses on implementing sequential substates, concurrent substates and compound transitions in Java. They enhanced this approach in [34] to include transformations for more complex statechart elements.

Pinter and Majzik [35] provided an implementation pattern for the instantiation of UML statecharts at the source code level. In this pattern, Extended Hierarchical Automata (EHA) is used as an intermediate representation of statecharts, as UML statecharts can be automatically mapped to EHA. A prototype of this pattern was implemented in C.

3.2.3 Interaction Model Transformation Approaches

Sangal et al. [36] presented a technique to generate code from UML interaction diagrams, particularly UML sequence diagram. As sequence diagrams lack sufficient details for generating completely executable code, the concept of interaction schemata was introduced. An interaction schema is a textual description of object-interaction, represented as a list of actions. All messages of sequence diagram are translated into actions. This technique has been implemented in a Java

programming tool named ***StructureBuilder***. Engels et al. [37] presented a methodical approach for deploying UML collaboration diagrams to model functional behavior. This approach focuses on specifying transformation rules to generate Java code from UML collaboration diagrams.

3.2.4 Hybrid Approaches

Niaz and Tanaka [38, 39] proposed a behavioral approach for generating code from class and state models. This approach has been implemented in a tool named ***JCode.*** JCode takes the specifications of statecharts as input and uses transformation rules to generate Java code from these specifications. Noe and Hartrum [40] also automate the process of transforming class and state transitions diagrams into Ada code by merging two tools, ***Rational Rose 98*** and ***AFIT***. Derezinska and Pilitowski [41, 42] proposed Framework for eXecutable UML (FXU) for transforming UML class and state models into C# source code. Class and state models are given as input in the form of XMI (XML Metadata Interchange).

Thongmak and Muenchaisri [43] proposed a rule-based transformation approach for generation of Java code from UML sequence and class diagrams. This approach was implemented in a tool for automating the transformation process. Long et al. [44] presented an algorithm to generate ***rCOS*** (Relational Calculus of Object Systems) code from UML class and sequence diagrams. rCOS is an OO language and its code is quite similar to Java. The algorithm uses class diagram to generate code skeletons. Sequence diagram is traversed by the algorithm to generate method bodies.

Nickel et al. [45] developed an environment ***Fujaba*** (From UML to Java And Back Again) that supports code generation from UML collaboration, activity and statechart diagrams. Java code skeletons are generated by UML class diagrams and method bodies are generated by using behavioral diagrams of UML. Bjorklund et al. [46] also proposed an approach to generate C++ source code. UML statecharts, activity and collaboration diagrams are given as input in the form of XMI. These input models are translated into Rialto using ***SMW*** toolkit. Rialto is an intermediate formal description language used between models and code.

Usman et al. [47, 48] developed a tool named ***UJECTOR*** (UML to Java Executable Code GeneraTOR) for automatic generation of executable Java code from UML class, sequence and

activity models. Structural code is generated from class model whereas behavior is added to the code by transforming sequence and activity diagrams. Models are given as input to the tool in the form of XMI. XMI is parsed and metamodel instances are created. These metamodel instances are used to produce isolated Java code, which is then merged with code of activity and sequence diagrams.

Jakimi and Elkoutbi [49] presented a method to generate code from UML sequence diagrams. An extension of this work [50] focuses on generating Java code from UML statecharts. The proposed approach defines transformation rules for converting UML statechart elements into Java code constructs.

Doungsa-ard and Suwannasart [51] proposed an approach for automatic transformation of CafeOBJ specifications to Java template code. CafeOBJ is a formal algebraic specification language. The proposed approach describes the steps and transformation rules for generating Java Template code. This approach is implemented in a tool, ***Cafe2Java***. The aforementioned approach only generates Java code skeletons. This approach was extended to include system behavior to generate more complete code [52].

3.3 Code Generation Tools

This section is dedicated to discuss some of the renowned commercial and open-source code generation tools.

3.3.1 Rational Rose

IBM® Rational® Rose® Enterprise [53] is a commercial OO UML-based software design tool capable of generating code in several languages including Java/J2EE™, C++ and Visual Basic. It provides support for generating code from UML class, component, deployment, sequence, statechart and use case diagrams. However, it only provides facility to generate code skeletons and do not provide any support for specifying system behavior.

3.3.2 Rational Rhapsody

IBM® Rational® Rhapsody® [54] is a commercial rule-based CASE tool relying on UML modeling standard. It allows development of source code from UML class and statechart diagrams. Like IBM Rational Rose, this tool is also capable of generating code frames only.

3.3.3 StructureBuilder

StructureBuilder [55] is a tool to develop UML class and sequence diagrams and generate Java code from these. However, the core limitation of this tool is that it can generate class structures only.

3.3.4 Enterprise Architect

Sparx System Enterprise Architect [56] is a comprehensive UML design tool capable of generating source code in multiple languages including Java. This commercial tool uses template technology and follows the traditional MDA process of transforming PIM to PSM to target code. However, it does not support any action language for behavior specification.

3.3.5 Visual Paradigm

Visual Paradigm [57] is a UML modeling and design tool by OMG to generate implementation code in multiple languages from thirteen UML diagrams. Still, no support for specifying behavioral actions in any action language is currently provided.

3.3.6 AndroMDA

AndroMDA [58] is an open-source framework following MDA paradigm. It generates Java/J2EE code by using template technology. It is a rule-based tool and does not offer support for generating class and method bodies automatically.

3.3.7 MagicDraw

MagicDraw™ [59] is a commercial UML software and system modeling tool supporting template-driven transformations by following the traditional MDA process. It provides partial

support for specifying behavioral actions. Yet, behavioral actions need to be added manually to complete the implementation code.

3.3.8 Papyrus

Papyrus [60] is an open-source tool by OMG capable of transforming UML class and statechart diagrams to Java and Ada2005 code. In order to specify constraints, OCL is supported by Papyrus. Nevertheless, complete support for behavioral actions is lacking.

3.3.9 iUML

iUML [61] is a commercial product that allows code generation from UML use-case, class, sequence and statechart models. This tool supports ASL for behavioral action specification. However, it relies on rule-based approach, where rules have been explicitly specified for transforming UML models into target implementation code.

3.3.10 ArgoUML

ArgoUML [62] is an open-source modeling tool built upon standard UML 1.4 metamodel. It possesses the capability of transforming nine UML models into Java, C++, C# and PHP code. Constraints on the models can be specified using OCL but no support for any action language is currently available.

3.4 Swarm Intelligence for Model Transformation

In the context of model transformation, we can discuss the application of swarm intelligence to M2C and M2M transformation.

3.4.1 Model-to-Code Transformation

A brief review of the existing literature reveals that the concept of swarm intelligence has never been applied in the context of automatic code generation before.

3.4.2 Model-to-Model Transformation

The intelligent behavior of swarms has been recently employed in the field of M2M transformation. Kessentini et al. [4] presented an approach ***MOTOE*** (Model Transformation as Optimization by Examples) to perform M2M transformation by using PSO. In [7], MOTOE is extended to build a more sophisticated M2M transformation process. In addition to PSO, enhanced approach also uses Simulated Annealing to perform the transformation. Since it is a M2M transformation approach, it covers only the structural aspects of a model rather than focusing on both the structural and dynamic aspects of a system.

3.5 Analysis

By analyzing the aforementioned approaches and tools, we can draw following conclusions:

1. All existing approaches and tools for automatic code generation are based on three major building blocks:
   - Transformation rules
   - Formal language to express transformation rules
   - Metamodels of source and target languages

2. All presented approaches and tools are rule-based i.e. transformation rules explicitly need to be defined for transforming different models into target code. Different approaches and tools use different transformation rules.

3. All presented approaches are directly dependent on UML metamodel, which implies that these metamodels must be understood before defining a mapping between source and target languages' metamodels.

4. For automatic M2C transformation, rules must be expressed in some formal language. It can be general purpose programming language, graph transformation language or model transformation language.

5. Major focus of research-based approaches and tools is on transforming UML class and statechart diagrams into implementation code.

6. A vast majority of tools do not provide support for action languages, meaning that these tools do not generate complete class and method bodies.

Chapter 4

PROBLEM DEFINITION

4.1 Introduction

Automatic M2C transformation drastically improves the productivity and efficiency of software developers. Keeping in mind this very advantage, the process of automatic code generation has been made quite mature. However, despite the many efforts invested by software researchers and practitioners in this field, there still exist some problems and issues that can obstruct the way to smooth and easy transformation.

This chapter of dissertation explains the issues and limitations associated with the existing approaches for automatic code generation. As there are three pillars of code generation process, consequently this chapter is divided into three sections, each section explaining the concerns of each building block. Section 4.2 discusses the difficulties associated with the definition and maintenance of transformation rule set. Section 4.3 highlights the issues of transformation languages. The complexities of metamodels are elaborated in Section 4.4. Finally, Section 4.5 summarizes the problem statement and states the existing research gap.

4.2 Issues of Transformation Rules

Presently, the activity of formulating a transformation rule set is considered as the central task of automatic code generation process. However in reality, the definition of a transformation rule set is not a simple task. This is due to several reasons. First, some transformations cannot be easily expressed as rules [4]. In some situations, rule induction can become impossible or difficult to achieve [7]. Rule set needs to be correct, complete, consistent and non-redundant so as to obtain the accurate target code. These properties are especially difficult to ensure in situations where little domain knowledge is available [3].

Moreover, transformation rule set is not only difficult to define, rather it is also hard to express and maintain. With the passage of time, transformation rule sets may evolve. Adding new rules or changing existing rules makes it complex to ensure their consistency and correctness. Furthermore, the definition of a complete transformation rule set requires proficiency in high-level programming languages, knowledge of the underlying metamodels and knowledge of the semantic equivalency between the metamodels' concepts [63] , which further

aggravates the situation. These unwanted limitations make the activity of transformation rule set difficult, tricky and complex.

4.3 Concerns of Transformation Languages

So far, the contributions in model transformation have mostly relied on defining transformation languages for expressing M2C transformation rules [4, 7] . As a result, a vast variety of model transformation languages have emerged, making it difficult for experts to choose the one that best serves the needs of their domain. Learning and specializing these languages is a difficult task and lingers the process of formulating a complete transformation rule set. Moreover, some languages are not expressive enough and all transformations cannot be easily expressed as rules using these languages.

4.4 Complexity of Metamodels

The development of a complete transformation rule set requires understanding of the source and target languages' metamodels. Practically, experts may find it difficult to master both the source and target metamodels [5] . Besides understanding the individual metamodels, they also need to realize the intricate mapping between the metamodels' elements. These activities delay the process of defining transformation rule set and also make the task comphcated.

4.5 The Gap

Existing research gaps that motivated this research are listed below:

1. Up till now, there is only one way of automatic code generation i.e. to use a transformation rule set, no alternatives exist.

2. Currently, all transformation approaches require explicit formulation of a transformation rule set. No such approach exists that can automatically extract or derive transformation rules without human intervention.

3. No single approach is capable of transforming source model element in the case of a missing transformation rule.

4. All M2C transformation approaches presented in literature so far use high-level programming languages or dedicated model transformation languages to express transformation rule set, no substitutes found.

5. All approaches proposed for automatic code generation rely on source and target languages' metamodels. No approach can be found in literature that is independent of source and target formalisms.

6. A vast majority of existing approaches and tools are capable of generating code skeletons only, with little or no support for generating behavioral code from system specification.

7. Most industrial organizations maintain a record of past M2C transformations. There is no way to use this fragmentary knowledge for solving new M2C transformation problems, by skipping the manual task of defining a complete transformation rule set.

Chapter 5

PROPOSED APPROACH FOR CODE GENERATION

5.1 Introduction

In order to lessen the pains of existing code generation process and to target the research gaps highlighted in the previous chapter, we propose a novel approach for automatic code generation. Our proposed approach is significantly different from what already exists regarding code generation in current literature. This approach makes use of the existing transformation examples, heuristic search and swarm intelligence to automate the process of generating code from system models. To the best of our knowledge, these concepts have never been applied in the context of code generation before.

This chapter of dissertation is dedicated to elucidate our novel approach for automatic code generation. The chapter starts by introducing some basic terminologies related to our approach in Section 5.2. Section 5.3 gives an overview of the approach and discusses our core transformation scheme. Section 5.4 provides a detailed insight to the structure and representation of the training data. An adaptation of PSO to our problem of automatic M2C transformation is presented in Section 5.5. Finally, the entire process of generating code from system models by using this approach is explained in Section 5.6.

5.2 Preliminaries

In order to facilitate further discussion, we start by introducing some basic concepts and terminologies related to our approach.

5.2.1 Input Source Model

An input source model *M* is a system design model that needs to be transformed to generate the target implementation code. The input source model consists of one or more model constructs.

5.2.2 Model Construct

A model construct is defined as an element of a model, e.g. in UML class diagram, classes and their relationships are the model constructs. Model constructs may contain properties that describe it, e.g. names of classes, names and multiplicities of associations, etc.

A model construct can be simple or complex. A complex model construct consists of one or more sub-constructs. For example, a class construct consists of attributes and operations.

5.2.3 Mapping Block

A mapping block depicts a previously performed transformation trace by relating the subset of source model constructs to their equivalent constructs in the target implementation language. In our case, we assume that these mapping blocks are manually defined by the domain experts.

Although our mapping block shows the transformation between the constructs of the source model and target code, they are different from transformation rules. Transformation rules involve general concepts which are defined at the metamodel level. However, our mapping blocks represent specific examples involving concrete concepts instances at the model level. For example, transformation rules are defined for the general concepts "Class" and "State" while mapping block includes the concrete concept instances "*Student* class" and "*Idle* state".

TH-1009

5.2.4 Transformation Example (Training Data)

Transformation example defines the mapping of constructs from source model to the target code language. A transformation example consists of one or more mapping blocks.

5.2.5 Predicate

In our context, we define a predicate as an expression that represents the construct of a model. We propose to express the input source model and training data as predicates.

Each predicate has a name for its identification. In M2C transformation, we propose to use the model construct as the predicate name. Each predicate has some parameters. The properties of the model construct are expressed as the predicate parameters. For example in UML class diagram, the class 'ShoppingCart' can be expressed as predicate in the following way:

```
Class (ShoppingCart)
```

5.3 Approach Overview

The main theme of our approach is to use knowledge from previously solved transformation examples to solve new M2C transformation problems. The existing set of examples is used to train the system regarding automatic code generation. After the system is trained transformation blocks, that best match the constructs of the input source model to be transformed, are extracted from the training data. These transformation blocks are then used to convert the source model constructs into target code. So, instead of explicitly providing a transformation rule set as input, our aim is just to provide a set of transformation examples and let the system automatically extract transformation rules from them.

As industry is becoming knowledge-oriented and relying on expert's decision making abilities [64], our approach relies on using a Knowledge Base System (KBS) to generate code from system models. A KBS is a computer-based system that acts as an expert on demand, saves time and money and increases productivity [64] . Our KBS consists of two main components, as shown in Figure 5.1.

1. *Knowledge Base* (KB) is a repository of knowledge. In our problem of automatic code generation, existing set of training data constitutes a KB.

2. *Inference Engine* (IE) is a software program that infers knowledge available in KB [64]. In our approach, we use a heuristic search optimization technique as an IE.

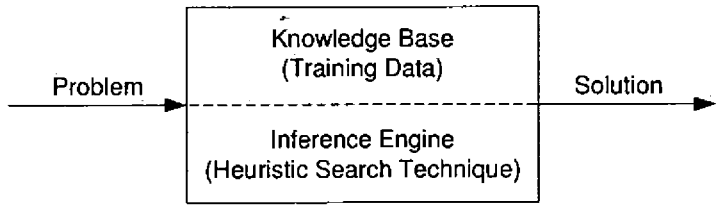


Figure 5.1 Knowledge base system

In our approach, training data is divided and represented as a set of mapping blocks. Our approach searches for the best mapping block corresponding to every construct of the input source model. During the search, all transformation examples are taken into account and not only the most similar one. For that reason, our approach actually differs from case-based reasoning

[65] in which only most similar example is selected. We take the best from all examples rather than selecting the most related example and adapting its transformation.

By selecting the best mapping block corresponding to every construct of the input source model, the final optimal solution consists of a combination of mapping blocks extracted from multiple transformation examples. If the source model constructs or the number of mapping blocks of training data is large, the number of possible combinations or solutions quickly becomes huge and exhaustive search becomes impractical. For example, if the source model contains 50 constructs and training data consists of 60 mapping blocks, the number of possible combinations raises to 60^{50}. Exploring and evaluating such a large number of possible combinations is time-consuming and inefficient.

For that reason, we view the problem of automatic code generation as an *optimization problem*. An optimal solution to this problem is found by using a heuristic search technique of *Particle Swarm Optimization* (PSO). An introduction of PSO can be found in section 2.4.1. From a wide range of available optimization techniques, we selected PSO because of the following reasons:

1. It is used to solve complex problems for which no easily implementable solution exists.
2. It is well-adapted for solving multi-modal problems.
3. As compared to other heuristic search optimization techniques, PSO consumes less computational resources.

Using PSO, solutions are represented as particles in the search space and each particle is evaluated using a user-defined objective function. Particle that produces the best fitness value for the objective function is selected as the final solution for M2C transformation.

Figure 5.2 gives an overview of our automatic code generation approach. At an abstract level, our approach divides the process of automatic code generation into four major steps, as listed below:

1. Training data represented as mapping blocks is given as input to our KBS. These transformation examples compose the KB and are used to train the system by inferring the mapping patterns of different model constructs.

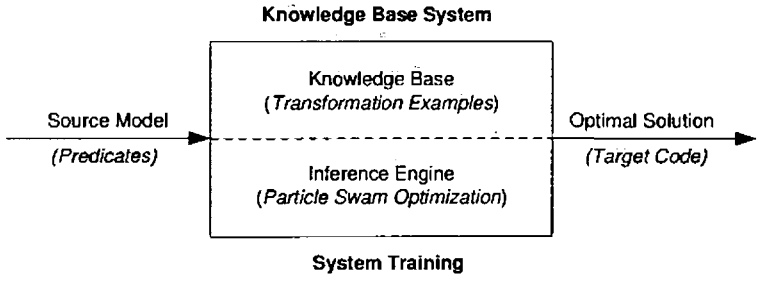


Figure 5.2 Code generation approach overview

2. Source model to be transformed is also provided as input.

3. One mapping block corresponding to each construct of input source model is selected from the set of training data by using PSO.

4. Finally, the selected mapping blocks from the training data are used to transform the constructs of the input source model into target code.

5.4 Knowledge Representation

Training data is given as input to the system as a set of mapping blocks. Each mapping block consists of a set of two building blocks:

1. Source Model Construct (SMC)

2. Target Code Construct (TCC)

SMCs represent the constructs of the input source model to be transformed and *TCCs* represent their equivalent constructs of the target code, expressed as predicates. In this way, each mapping block statement depicts that which SMC has been transformed into which TCC at model level, expressed as the following structure

```
<SMC> : <TCC>
```

A mapping block may include the mapping of SMC to TCC for more than one model construct. More specifically, the interrelated constructs or the constructs that should be transformed together are grouped into a single mapping block. Each model construct is expressed as one or more predicates.

Our approach of automatic code generation is a generic approach and can be used to transform any source model into target code. However, in order to illustrate and clarify our approach we will use class and state models as examples. Our choice of using these two models as examples is motivated by the fact that class and state models are representative of both the static structure and the dynamic system behavior. Moreover, our proposed approach cannot only be used to represent skeletons of model and code, rather behavioral actions inside the models can also be easily expressed as predicates and can be automatically transformed to generate complete implementation code.

5.4.1 Class Model Representation

Figure 5.3 shows the class diagram of a case study, an "Online Shopping System". The class model consists of 6 classes and 6 relationships (1 generalization + 5 associations). As in a class diagram, classes along with its attributes and operations are treated as a single construct and relationship as a separate construct, therefore total constructs of this class model add up to twelve.

In the class model, each relationship has a class at its both ends. This illustrates that the transformation of a relationship in a class model is dependent on these two classes. Therefore, every relationship in the class model along with its two associated classes forms a single mapping block i.e. each mapping block consists of the transformation of two classes and their relationship. This implies that the total number of mapping blocks required to represent the SMC of a class model is equal to the total number of relationships in a class model i.e. this class model can be represented by 6 mapping blocks.

Constructs of the class model are expressed as predicates and the properties of the constructs become the parameters of these predicates. For example, 'class' construct can be expressed as predicate in the following way:

```
Class(<name>)
```

For the class named 'ShoppingCart', its predicate can be expressed as follows:

```
Class(ShoppingCart)
```

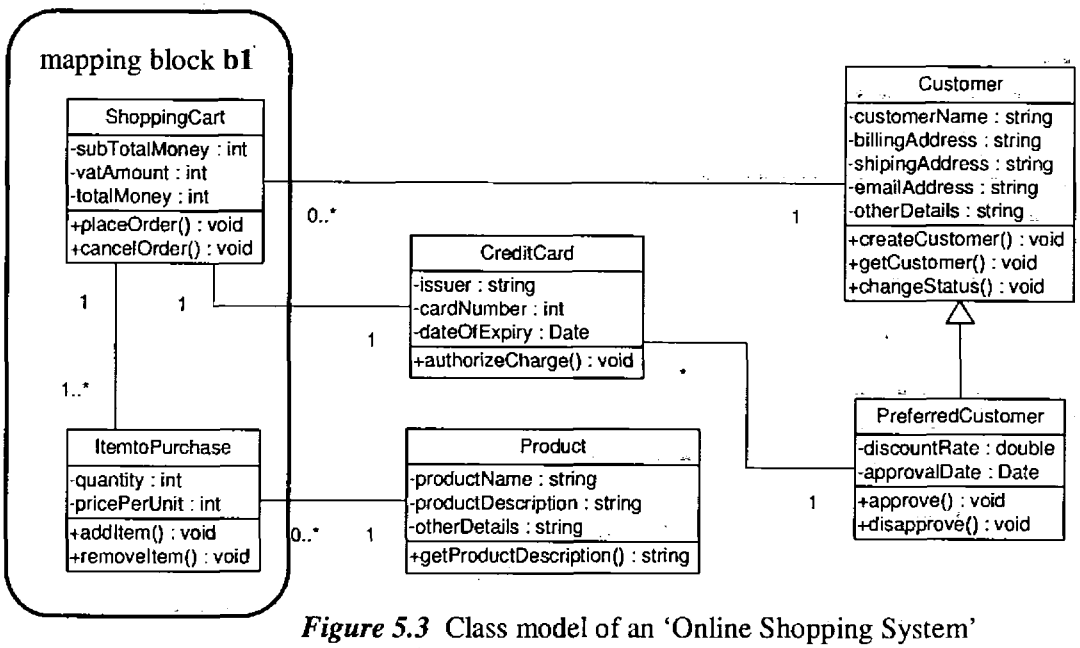


Figure 5.3 Class model of an 'Online Shopping System'

We have used only one property of class in its predicate representation. This does not imply that name is the only property of class rather it means that name is the only property of class that is required for code generation. This approach can be used with many different kinds of predicate structures, as it is a general approach. However, we have defined and used this structural scheme throughout this research project. The complete templates of our predicate structures and transformation schemes can be found in Appendix A.

Using our templates, the relationship between the classes of 'ShoppingCart' and 'ItemToPurchase' in Figure 5.3 can be represented as a complete mapping block in the following way:

```
Begin b1
Class(ShoppingCart):Class(public,ShoppingCart)
Attribute(SubTotalMoney,int,0,ShoppingCart,_):Attribute(private,
int,SubTotalMoney,0,ShoppingCart,_)
Attribute(vatAmount,int,0,ShoppingCart,_):Attribute(private,int,
vatAmount,0,ShoppingCart,_)
Attribute(totalMoney,int,0,ShoppingCart,_):Attribute(totalMoney,
int,0,ShoppingCart,_)
```

```
Operation(placeOrder,ShoppingCart,void):Method(public,void,
placeOrder,ShoppingCart)
OperationParam(-,-placeOrder,ShoppingCart,1):MethodParam(-,-,
placeOrder,ShoppingCart,1)
Operation(cancelOrder,ShoppingCart,void):Method(public,void,
cancelOrder,ShoppingCart,void)
OperationParam(-,-,cancelOrder,ShoppingCart,1):MethodParam(-,
-,cancelOrder,ShoppingCart,1)
Class(ItemToPurchase):Class(public,ItemToPurchase)
Attribute(quantity,int,0,ItemToPurchase,_):Attribute(private,int,
quantity,0,ItemToPurchase,_)
Attribute(pricePerUnit,int,0,ItemToPurchase,_):Attribute(private,
int,pricePerUnit,0,ItemToPurchase,_)
Operation(addItem,ItemToPurchase,void):Method(public,void,
addItem,ItemToPurchase)
OperationParam(-,-,addItem,ItemToPurchase,1):MethodParam(-,
-,addItem,ItemToPurchase,1)
Operation(removeItem,ItemToPurchase,void):Method(public,void,
removeItem,ItemToPurchase)
OperationParam(-,-,removeItem, ItemToPurchase,1):MethodParam(-,
-,addItem,ItemToPurchase,1)
Association(_,1,1,n,_,ShoppingCart,ItemToPurchase):Attribute
(private,ShoppingCart,shoppingcart(),ShoppingCart,ItemToPurchase)
End b1
```

This mapping block indicates that the SMCs of the classes 'ShoppingCart' and 'ItemtoPurchase' in the class model are transformed to the corresponding classes in the TCCs with the same names. Attributes and operations of source model classes become the attributes and methods of the corresponding classes in the target implementation code. The association relationship between these two classes in the source model is translated by making an object of

class 'ShoppingCart' in the class 'ItemtoPurchase' as an attribute. All this information is expressed as predicates in the aforementioned mapping block.

5.4.2 State Model Representation

In the state diagram, states along with its entry, do and exit activities form one construct while transition is taken as a separate construct. For example, consider the state model of class 'ShoppingCart' shown in Figure 5.4. It has two possible states and four transitions that change the state of an object. Since a transition is dependent on its source and target states, each mapping block consists of source state, transition and its target state. Total number of mapping blocks required to represent a state model is equal to the number of transitions that causes an object to change its state.

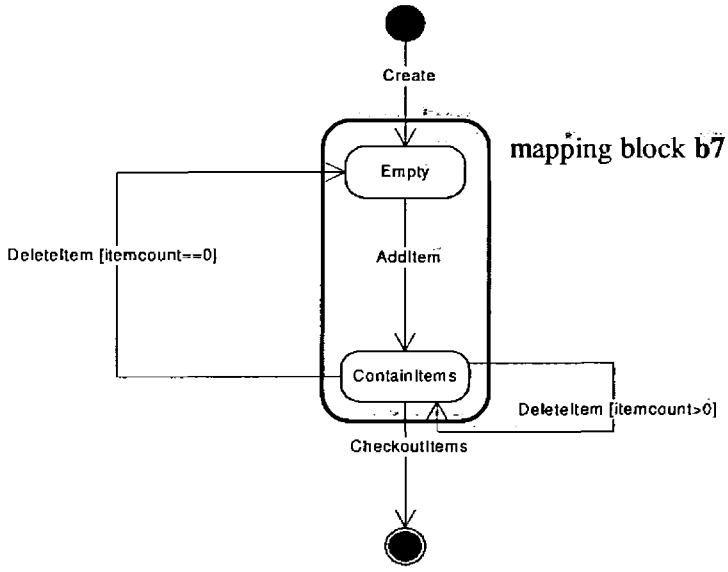


Figure 5.4 State model of the class 'ShoppingCart'

The transition from state 'Empty' to 'ContainItems' can be represented as a mapping block in the following way:

```
Begin b7
State(SCartEmpty,ShoppingCart):Class(public,SCartEmpty,
ShoppingCart)
```

```
Operation(Entry,SCartEmpty,void):Method(public,void,Entry,
SCartEmpty)
OperationParam(-,-,Entry,SCartEmpty,1):MethodParam(-,-,Entry,
SCartEmpty,1)
Operation(Exit,SCartEmpty,void):Method(public,void,Exit,
SCartEmpty)
OperationParam(-,-,Exit,SCartEmpty,1):MethodParam(-,-,Exit,
SCartEmpty,1)
Operation(doActivity,SCartEmpty,void):Method(public,void,
doActivity,SCartEmpty)
OperationParam(-,-,doActivity,SCartEmpty,1):MethodParam(-,
-,doActivity,SCartEmpty,1)
State(SCartContainItems,ShoppingCart):Class(public,
SCartContainItems,ShoppingCart)
Operation(Entry,SCartContainItems,void):Method(public,void,Entry,
SCartContainItems)
OperationParam(-,-,Entry,SCartContainItems,1):MethodParam(-,-,
Entry,SCartContainItems,1)
Operation(Exit,SCartContainItems,void):Method(public,void,Exit,
SCartContainItems)
OperationParam(-,-,Exit,SCartContainItems,1):MethodParam(-,-,
Exit,SCartContainItems,1)
Operation(doActivity,SCartContainItems,void):Method(public,void,
doActivity,SCartContainItems)
OperationParam(-,-,doActivity,SCartContainItems,1):MethodParam(-,
-,doActivity,SCartContainItems,1)
Transition(Empty,AddItem,ContainItems):Method(public,void,
AddItem,Empty)
End b7
```

In this mapping block, the SMC of state in the state model is translated into a TCC of class (i.e. each state is translated into an implementation class). For every class corresponding to a state, there are separate methods for encapsulating entry, exit and do activities of state. The construct of transition is mapped to a method in the target code. The complete predicate templates of state model can be found in Appendix A.

5.4.3 Action Specification

Generating the implementation code only in terms of class and method declarations is not sufficient. Program logic and actions is the most significant part of the software systems for execution. Our approach can not only be used to generate code skeletons, rather it is powerful and flexible enough to transform models that encapsulate dynamic actions. In our approach, each action within a model is treated as a separate construct.

Currently, several action languages are available for incorporating dynamic behavior in the system models at the design level. Logically speaking, our approach is independent of all these languages and is capable of transforming any action language into any target programming language, provided that transformation examples for that set of action and implementation languages are available. For example, if we have transformation examples for converting Object Constraint Language (OCL) model actions into C# code statements, we can perform new transformations from OCL to C# using these examples by utilizing this approach. Similarly, if the transformation examples show the translation of Action Specification Language (ASL) into Java code, our approach can also be used for this set of mapping, and so on.

To apply our approach, we only need to express the available transformation examples as predicates. The different types of actions become the predicate names and the properties of action statements become the predicate parameters. For example, consider the following statement for method call expressed in ASL.

```
[number] = op1:getPhoneNo[] on phone
```

This ASL statement indicates that the operation of 'getPhoneNo' is called on the 'phone' object and the result is stored in a variable 'number'. Its corresponding Java statement can be written like this:

```
number = phone.getPhoneNo();
```

Suppose that this function call is in the body of the operation 'GetNumber' of class 'Customer', then this example can be expressed as predicate in the following way:

```
OpBodyFn(GetNumber,Customer,number,getPhoneNo,phone,-,4) :
MethodBodyFn(GetNumber,Customer,number,phone,getPhoneNo,-
,4)
```

The predicate name (OpBodyFn) indicates that the action of 'function call' is represented by this predicate. In the 'GetNumber' operation of class 'Customer', there is an action for calling the operation of 'getPhoneNo' on the 'phone' object. Moreover, the predicate also depicts that this action is the 4th statement in the 'GetNumber' operation.

In this manner, all action statements (e.g. sequential, iteration, decision statements, etc.) can be easily expressed as predicates, no matter which action or implementation language has been used. In the context of class model, action languages can be used to specify operation body. Entry, do, exit activities and transition guards can be expressed in the state model by using the action languages. The complete predicate templates of all action statements can be found in Appendix A

5.5 PSO Adaptation for Automatic Code Generation

Our approach works by finding the most appropriate mapping for every model construct from the set of provided transformation examples. As described in section 2.4.1, PSO represents solutions as particles in the search space. Likewise, in our problem of M2C transformation these particles are the transformation blocks extracted from the available set of training data. The task of PSO is to search for the mapping block that contains the transformation of construct similar to the one in the source model. The transformation construct from the training data is considered to be similar to the source model construct if it shares the same construct name with similar properties.

To apply heuristic search techniques to a specific problem, it is necessary to specify the representation of solutions, the fitness function to evaluate the quality of the searched solution

and the operators that allow movement in the search space so as to find new solutions [7] . The next sub-sections elaborate the adaptation of these PSO elements to our problem of automatic M2C transformation.

5.5.1 Representation of Transformation Solution

Using PSO, solutions are represented as particles in the search space. These particles move in the *D*-dimensional space to find an optimal solution. In our problem, we consider the dimensions of the search space as the constructs of the input source model to be transformed. This implies that the number of dimensions in the search space is equal to the total number of input source model constructs. For example, the transformation of the class model shown in Figure 5.3 will generate a 12-dimensional search space that accounts for 6 classes and 6 relationships (1 generalization + 5 associations).

The mapping blocks in the training data will be numbered from 1 to *m*, *m* being the total number of mapping blocks. These mapping block numbers are the possible coordinates of the *D*-dimensional search space. It means that the dimensions of the search space will take discrete values from 1 to *m* [1, m]. Each of the input source model construct will be associated a discrete value from 1 to *m* that represents a transformation possibility for that construct.

This solution is implemented as a vector in the *D*-dimensional search space. The constructs of the source model are the elements of the vector whereas the mapping block numbers that show the transformation possibilities are the values in the vector elements. For example, let us consider that the training data for transforming class model of Figure 5.3 has 30 mapping blocks. Table 5.1 shows a possible transformation solution vector. This vector has 12 elements (total number of source model constructs) and each element can take values from 1 to 30 (total number of mapping blocks). First element has a value of 4, which means that 1st construct of the input source model can be transformed using mapping block number 4, 2nd construct using block number 25, 3rd construct using block number 8 and so on.

Table 5.1 Transformation solution vector

Construct number	1	2	3	4	5	6	7	8	9	10	11	12
Mapping block number	4	25	8	9	20	23	22	7	1	19	30	12

5.5.2 Evaluation of Transformation Solution

The quality of the transformation solution produced by PSO is evaluated against a user-defined objective function. In M2C transformation, fitness value of a particle indicates the appropriateness of the mapping blocks selected for the transformation of their corresponding source model constructs. For our problem of generating code from system models, we have defined the following fitness function:

$$f = \sum_{i=1}^{D} c_i * t_i \quad \text{..... (1)}$$

where D is the total number of constructs in the input source model to be transformed,

$$c_i = \begin{cases} 0, \text{ if source model construct } \textit{cannot} \text{ be transformed by the selected mapping block} \\ 1, \text{ if source model construct } \textit{can} \text{ be transformed by the selected mapping block} \end{cases}$$

$$t_i = \frac{\text{number of } \textit{key} \text{ parameters matched in the predicates of the } i^{th} \text{ construct}}{\text{total number of parameters in the predicates of } i^{th} \text{ construct}}$$

This fitness function is generic and can be used to evaluate the transformation solution of any source model. The fitness function can be divided into two parts: c_i and t_i. The value of c_i is calculated by matching the construct names and the comparison of construct parameter values determines the value of t_i. These points are further elaborated below.

Using this fitness function, first the names of the source model construct and the selected mapping block construct are matched. If the construct names are different, c_i is assigned the value of zero. In this case, there is no need to match the predicate parameters as construct names are different. If the construct names match, the value of c_i is set to 1 and the parameters are then compared to determine the value of t_i.

In calculating the value of t_i, instead of using all predicate parameters, only *key* parameters are involved. Its reason will be explained shortly after defining the term key parameters. The term *key parameters* represent those parameters of predicates which are significant in the construct and the difference of these parameters can result in a wrong transformation. In order to further clarify this term, we will use an example of the class model

shown in Figure 5.3. Consider the transformation of the following relationship construct of the class model into code construct:

```
Association(_,1,1,n,_,ShoppingCart,ItemToPurchase)
```

The target mapping block contains the following mapping:

```
Association(0,1,0,n,_,Person,Loan):Attribute(private,Person,
person,Person(),Loan)
```

In this example, the association construct has total 7 parameters - 4 multiplicities, 1 association name and 2 classes. In this particular case, 5 parameters of the input source model construct (2 multiplicities, association name and 2 classes) match with the transformation mapping block construct whereas two parameters (1st and 3rd parameter) are different. But these two parameters that do not match are *NOT* the key parameters. Whether these parameters are zero or undefined (-), they have no effect on the correctness of the transformation. SMC can be accurately transformed into its code construct, even when these two parameters are different.

The key parameters in this example are 2nd and 4th parameters because these two parameters decide that object of which class will be created in another class (object of class with multiplicity of 1 will be an attribute of class with multiplicity of *n*). In case of aggregation and composition constructs, key parameters indicate that either a single object or an array of objects will be created in the contained class. In case, if a construct has no key parameters, the value of t_i will be considered as 1 by default. For example, the generalization construct of class model has 2 predicate parameters. These two parameters are the names of the parent and child classes respectively. For example, the generalization relationship in Figure 5.3 can be represented as follows:

```
Generalization(Customer,PreferredCustomer)
```

In this case, there is no key parameter and the construct can be correctly transformed if the construct names are similar. So in the absence of key parameters, the comparison and matching of construct names is sufficient to evaluate the quality of the selected transformation solution. As we have used class and state models as examples to explain our approach, the details of key parameters of their constructs are given in Appendix A.

In case of 'Association' construct, if we use all parameters in calculating t_i, and assign equal weight to all parameters of the predicate, each parameter will be awarded a weight of 0.145 approximately (1.0/7). For the above-mentioned example of association construct where key parameters are similar, the value of t_i is 1, indicating that the selected transformation block is appropriate for the translation of the corresponding SMC. However, if we consider all predicate parameters, the value of t_i will become 0.725 (0.145*5), as two predicate parameters do not match. Involvement of all parameters in the fitness function has penalized the value of t_i although the transformation is correct.

Now let us consider another case in which the key parameters of the source model construct of 'Association' do not match with the transformation construct of the mapping block, i.e. 5 parameters of the predicates match while 2 key parameters are different. Practically speaking, if key parameters are different, the transformation should NOT be considered appropriate, as it will result in a wrong transformation. If we consider the above-mentioned fitness function, the value of t_i will be zero (as it should be) but if we consider all parameters in calculating t_i, its value will be 0.725 (0.145*5), which is a reasonably good, if not perfect, fitness value for making the transformation solution worth-considering.

If we compare the above two examples, using all parameters for calculating t_i results in the same fitness value of 0.725, although the selected transformation block is correct in the 1st case and incorrect in the 2nd. This will result in a weak fitness function as the solution with wrong transformations will also generate an acceptable fitness value. However, if we involve only key parameters, the fitness values are different, 1 in the case of accurate transformation selection and zero in the case of selecting incorrect transformation block. This means that if we consider only key parameters, our fitness function will be strong and intelligent enough to search for better solutions. This justifies our choice of using key parameters in evaluating the transformation solution, instead of involving all parameters in the fitness function.

By using the above-mentioned formula for calculating t_i, a value between 0 and 1 is obtained. So for each construct of the input source model, the values of c_i and t_i will multiply up to 1. Depending upon the number of constructs in the input source model, the fitness function shown in equation 1 will generate fitness values in different ranges. In order to make the fitness

values comparable across models having different number of constructs, a normalized fitness value in the range [0, 1] can be obtained by using the following equation.

$$f_n = \frac{f}{D} \quad \text{..... (2)}$$

where D is the total number of constructs in the input source model.

5.5.3 Deriving An Optimal Solution

Our PSO starts by initializing the swarm with randomly generated particles. This is done by assigning random block numbers to the vector elements. The block numbers are in the range [1, m], where m is the total number of mapping blocks that constitute the training data. The number of particles in the swarm is a user-defined parameter. The value of this parameter is set on the basis of the dimensions of the search space and the complexity of a problem whose optimal solution is to be searched. Typical value of this parameter ranges from 20 to 50 [25].

In the swarm, each individual particle is composed of three D-dimensional vectors, D being the dimensionality of the search space. Using these vectors, particles keep a record of their current position (x_i), velocity (v_i) and the previous best position (p_i). The current position of a particle can be considered as a set of coordinates that describe the position of that particle in the search space [25].

These randomly initialized particles are evaluated against the fitness function to determine the quality of the generated transformation solutions. The particle with the highest fitness value is stored as the global best position (g_{best}). g_{best} is the best position achieved so far by any particle in the search space. The coordinates of the g_{best} are recorded in p_g. As this is the first iteration, each particle stores its fitness value in p_{best} and its coordinates in p_i. This value is stored for comparison on later iterations. In each of the next iterations, the fitness values of the particles are compared with p_{best} and g_{best}. If the values in current iteration are better, new values are assigned to p_{best} and g_{best}.

Next iteration starts by updating the values of x_i and v_i. The current position of every particle is updated using the following formula [24]:

$$x_i = x_i + v_i \quad \dots\dots\dots \quad (3)$$

Using the two best positions p_{best} and g_{best}, the velocity of particles are updated in each iteration by applying the following formula [24]:

$$v_i = W * v_i + C1 * rand1 * (p_i - x_i) + C2 * rand2 * (p_g - x_i) \quad \dots\dots\dots \quad (4)$$

where *W* is the *inertia weight.* It is used to better control the scope of the search [25] and sets a balance between the local and global exploration abilities in the swarm [66]. Global exploration of swarm is facilitated by the large value of inertia while the small inertia value supports local exploration to fine-tune the current solution.

C1 and *C2* are the learning factors called *acceleration coefficients* [67]. These two parameters represent cognitive and social weights associated to the individual and global behavior respectively [4, 7]. *C1* is used to control the impact of particle's own history on the particle's new position whereas *C2* is used to control the impact of swarm history on the new position of the particle. These two factors are also known as the self-confidence and the swarm-confidence factors respectively [68]. Empirical studies suggest that *C1* and *C2* should not be equal to 2 all the time [69].

rand1 and *rand2* are two uniformly distributed random values between 0 and 1. These two values represent the stochastic acceleration during the attempt to pull each particle towards the p_{best} and g_{best} positions [7] .

Collectively, the first part of equation (4) represents the *previous velocity* which provides the necessary momentum for particles to roam across the search space. The second part is the *cognitive component* which represents each particle's personal thinking. It encourages the particles to move towards their own best positions found so far. The third part is the *social component.* It represents the collaborative effect of the particles in finding the global optimal solution [70] .

The algorithm iterates until the particles converge to a good transformation solution or the maximum number of iterations is completed. In our approach, we define the maximum number of iterations as the stopping criterion for our search of finding an optimal transformation solution.

5.5.4 Parameter Tuning

Parameter values play a significantly important role in PSO's ability to search for the optimal solution. We set the following parameter values in our approach to find an optimal solution:

1. First parameter is the size of the population i.e. the number of particles in the search space. In our approach, we set the number of particles to 40, as typical implementations of PSO use this swarm size [68, 70].

2. Acceleration coefficients *C1* and C2 change the amount of tension in the system. Low values allow particles to roam far from target regions before being tugged back while high values result in abrupt movement toward or past target regions [69]. We set the values of *C1* and *C2* to 1.75, to give equal importance to both the local and the global search.

3. In each iteration, inertia is calculated as follows [25, 69]:

$$W = W_{max} - ((W_{max} - W_{min}) / iter_{max}) * iter \quad \text{......} \quad (5)$$

where W_{max} is the initial value of W, W_{min} is the final value of W, $iter_{max}$ is the maximum number of iterations and *iter* is the current iteration number. We set W_{max} to 0.9 and W_{min} to 0.4, as researchers have found that best performance could be obtained between these values [25].

4. We set the maximum number of iterations i.e. $iter_{max}$ to 20.

Velocity and position of a particle must be limited to v_{max} and x_{max} respectively so that the values always remain within the specified range. v_{max} determines the resolution or fitness with which regions between present position and target position are searched. If v_{max} is too high, particles might fly past good solutions. If v_{max} is too small, particles may not explore sufficiently beyond locally good regions [69]. In M2C transformation, v_{max} should be in the range of [-*m*, *m*] and x_{max} should be in the range of [1, *m*], where *m* is the total number of mapping blocks constituting the training data.

5.6 Automatic Code Generation Process

Using this approach, the process of automatic code generation can be divided into the following four major steps, as shown in Figure 5.5.

1. Build a Knowledge Base
2. Prepare input source model
3. Search for an optimal solution
4. Transform the model using the optimal solution.

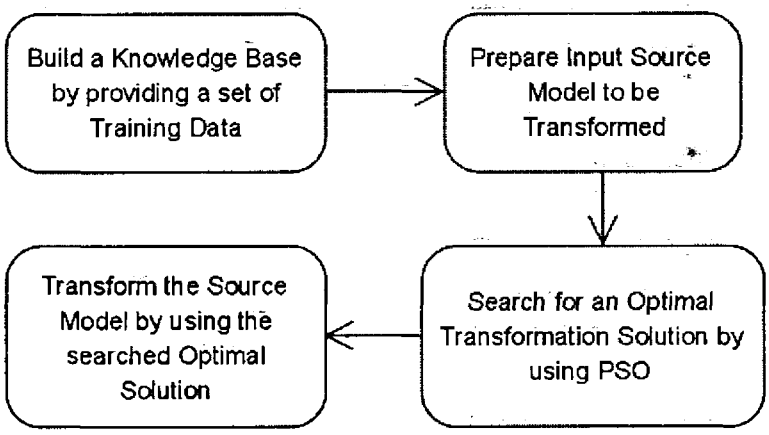


Figure 5.5 Steps of automatic code generation process

5.6.1 Build a Knowledge Base

Our proposed approach for M2C transformation relies on the existing set of transformation examples. The training data is divided and represented as a set of mapping blocks, expressed as predicates. The mapping blocks are numbered with integral values starting from 1. We assume that these mapping blocks are manually defined and represented by the domain experts. These transformation examples constitute the KB and are provided as input to the KBS. They are used to train the system regarding code generation and provide a base for automatic M2C transformation.

5.6.2 Prepare Input Source Model

This approach is generic and can be used to transform any input source model into target code. Like training data, the input source model also needs to be expressed as predicates. Currently, the task of expressing model as predicates needs to be carried out manually. After expressing the source model as predicates, it is given as input to our KBS.

5.6.3 Search for an Optimal Solution

After providing the training data and the source model to be transformed as input, an IE of the KBS then searches for the appropriate transformation corresponding to each model construct from the mapping blocks of the training data. This search space of training data is explored by using the heuristic search optimization technique of PSO. The task of PSO is to search for the mapping block that contains the transformation of the construct similar to the source model construct to be transformed. Each solution generated by PSO is evaluated against the fitness function. The solution having the best fitness value is selected as the final optimal solution. This solution is in the form of a vector in which the vector elements indicate the mapping block numbers corresponding to every source model construct.

5.6.4 Transform the Model Using Optimal Solution

Finally, an optimal solution searched by PSO is used to transform the constructs of the source model. The resultant code construct is expressed as predicates.

Chapter 6

TOOL IMPLEMENTATION

6.1 Introduction

We have successfully implemented our proposed approach in a tool named **CØde Ṣwarm**, abbreviated as **CØdṢ**. Currently, CØdṢ is capable of generating code from two system design models, class model and state model. Our motivation for selecting these two models lies in the fact that class and state models are representatives of both the static structure and the dynamic system behavior. Moreover, the method bodies of classes and other system behavior can be incorporated in these models using action language ASL. CØdṢ is capable of understanding and interpreting ASL, so that complete target code can be generated instead of just the code skeletons.

This chapter is dedicated to explain the tool CØdṢ, based on the application of our proposed approach. Section 6.2 presents the architecture and major components of CØdṢ. The implementation specific details of CØdṢ are explained in Section 6.3. Finally, section 6.4 describes the working and process flow of our tool.

6.2 CØdṢ Architecture

Figure 6.1 shows the overall architecture of our tool CØdṢ. The architecture of CØdṢ has three major components; *PredicateParser*, *SearchEngine* and *M2CTransformationEngine*. Our tool takes a set of M2C transformation examples and source models as input. This input is managed and organized by the PredicateParser. SearchEngine finds an optimal solution for the input models by using the available transformation examples. Finally, the M2CTransformationEngine uses the optimal solution produced by the SearchEngine to transform the input model constructs into target code predicates. Moreover, CØdṢ is capable of transforming these code predicates into complete Java statements. Therefore, a set of files containing Java code is also produced as the final output. An explanation of the three major components of CØdṢ follows.

6.2.1 PredicateParser

PredicateParser initiates the execution of CØdṢ. First, it takes a set of transformation examples as input and uses them to build a knowledge base. The training data is represented as a

set of mapping blocks stored in plain text files. The main responsibilities of PredicateParser are listed below:

1. To count the number of mapping blocks.
2. To divide the mapping blocks into SMCs and TCCs.
3. To organize the training data and form a knowledge base by creating separate structures for storing SMCs and TCCs of input transformation examples.

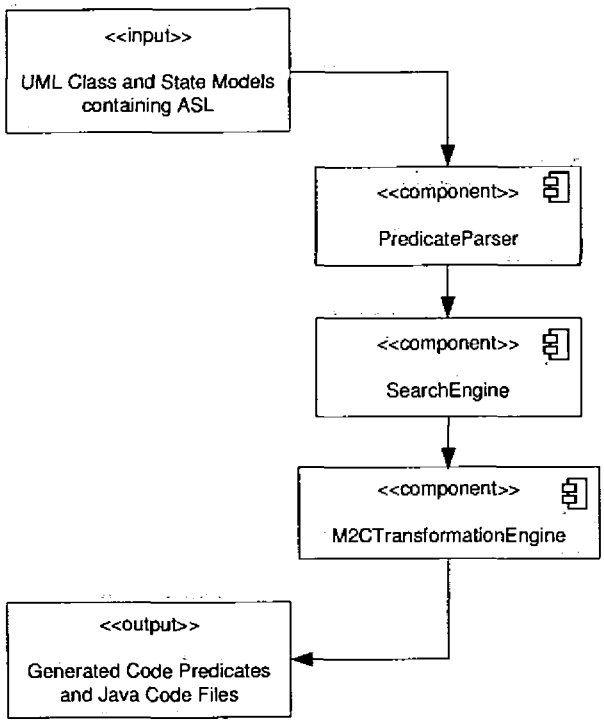


Figure 6.1 CØdŞ architecture

CØdŞ takes a set of class and state models as input stored in a text file. These models contain ASL statements to depict the dynamic system behavior, expressed as predicates. PredicateParser organizes and stores these models to be used by the later components. For the input source models, following tasks are performed by the PredicateParser.

1. To store the input source models.
2. To count the number of constructs to be transformed.

3. To divide the source models into separate constructs and maintain their record.

6.2.2 SearchEngine

SearchEngine is a vital and major component of our tool. We have used PSO algorithm as the search engine. The core responsibility of the *SearchEngine* is to search for an optimal transformation solution for the input source model constructs. The transformation solution is in the form of a set of mapping block numbers, one mapping block corresponding to every input source model construct.

SearchEngine initializes by assigning random mapping blocks for transforming the source model constructs. The quality of this random solution is assessed by using the fitness function defined in Section 5.5.2. Depending on the fitness value calculated by the objective function, the parameters of PSO are updated and more transformations solutions are generated in the next iterations. The component of the SearchEngine remains active until the total number of iterations of PSO is completed. The solution having the maximum fitness value is selected as the final optimal solution.

6.2.3 M2CTransfomationEngine

M2CTransformationEngine is a significantly important component of our tool. The principal job of *M2CTransformationEngine* is to produce the target code, both in terms of predicates and code statements, corresponding to the input source models. It does so by using the optimal transformation solution produced by the SearchEngine component.

The optimal solution generated by the SearchEngine is taken as input by the M2CTransformtaionEngine. For every input source model construct, it searches for the matching construct in the selected mapping block. The code predicate of this mapping block construct is then used to transform the corresponding input source model construct. In this way, output produced by M2CTransformationEngine is a set of code predicates produced for the input source models.

This component is also responsible for transforming these code predicates into complete Java statements. The automatic conversion of code predicates into code statements eradicates the

manual effort and time required for the conversion. The complete Java code generated by M2CTransformationEngine is organized and stored into a set of Java files.

6.3 CØdŞ Implementation

CØdŞ tool is realized using Java programming language. We used Eclipse IDE [71] with JDK 7 for the implementation of CØdŞ. From the implementation point of view, CØdŞ is organized into three main packages as shown in Figure 6.2. *AutoCodeGenerator* is the major package of our tool that implements the core logic of transforming input source models into the target code. *PSO* package encapsulates the general logic of our heuristic search optimization technique PSO. The package of *Editor* mainly deals with the GUI of CØdŞ. A brief description of the main packages of CØdŞ is given below.

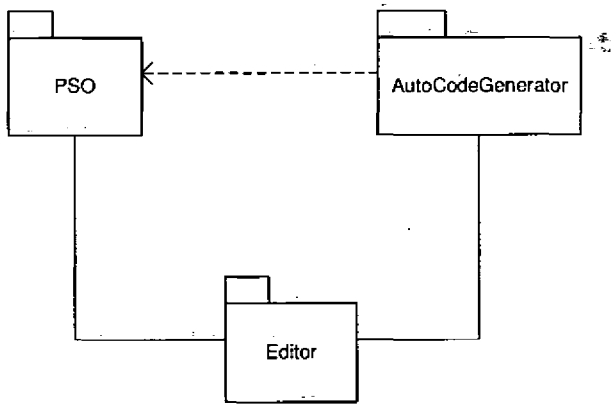


Figure 6.2 CØdŞ implementation

6.3.1 AutoCodeGenerator

The package of *AutoCodeGenerator* is mainly responsible for the transformation of input source models into code predicates and eventually into the final Java code. More specifically, following tasks are assigned to this implementation package.

1. Manage the input source models which are to be transformed.
2. Manage and organize the input training data to form the knowledge base.
3. Evaluate the quality of each transformation solution generated by PSO.

4. Select the final optimal solution.

5. Transform the input source model constructs into target code constructs by using the mapping blocks selected in the optimal solution.

6. Generate complete Java code statements from the resultant code predicates of the input source models.

AutoCodeGenerator package consists of five classes which collaborate with each other to accomplish the above-mentioned jobs. The interaction pattern of these classes is shown in Figure 6.3. These classes perform the following tasks:

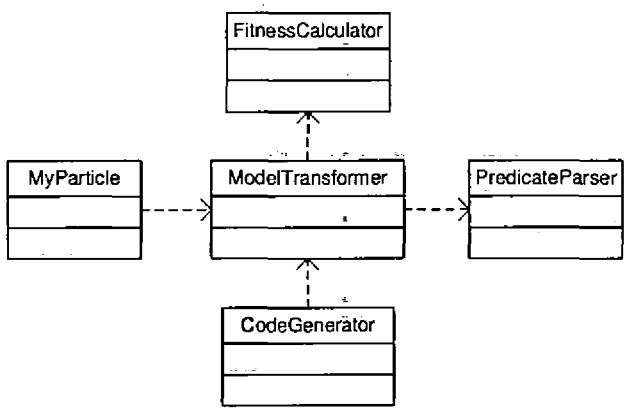


Figure 6.3 Interaction pattern within the 'AutoCodeGenerator' package

PredicateParser: This class is responsible for managing the input source models, particularly for counting the input model constructs and organizing them into proper structures. The management of the input training data to form a knowledge base is also the responsibility of this class.

FitnessCalculator: The fitness function defined in our approach is realized by this class. It takes a solution vector as input and calculates the fitness value corresponding to that transformation solution.

ModelTransformer: This class of AutoCodeGenerator package is responsible for generating target code predicates corresponding to the input source model constructs. It takes an optimal solution vector, searches for the mapping block to be used for the transformation and performs

the M2C transformation. In case if a construct similar to input SMC is not found in the mapping block, it calculates the relevance score for that input SMC. *Relevance score* tells the ratio of similarity between the input SMC and the SMC present in the mapping block. This score is calculated by using the construct name and the key parameters of the SMC.

CodeGenerator: This class performs the task of generating complete Java code statements from the TCCs produced by the 'ModelTransformer' class. It also organizes and arranges the code to be placed in multiple Java class files.

MyParticle: This class is responsible for passing the total number of input model constructs to another package.

6.3.2 PSO

The implementation logic of our search engine is encapsulated in this package named *PSO*. This package is responsible for performing the following tasks:

1. Provide a generic base class for defining specialized fitness functions.
2. Generate, update and manage multiple particles in the swarm.
3. Realize the equations defined in section 5.5.3 for updating particle velocities and positions.
4. Compare the different fitness values and keep a record of the solutions having the best fitness value.
5. Mange and evolve the swarm in accordance with the fitness values of particles.
6. Update parameters of PSO.

These responsibilities are divided among the six classes of PSO package in the following manner.

Swarm: It is the major controlling class in the PSO package and performs the task of managing the swarm of particles. It calls the methods of other classes to evaluate the swarm, update particles' positions and velocities and applies the positions and velocities constraints. It also performs the task of initializing the swarm with random particles.

FitnessFunction: This class acts as a parent class for realizing the user-defined fitness functions. It compares fitness values of different particles in the swarm and maintains a record of the best particles.

VariablesUpdate: This class is assigned the task of revising and updating parameter values of PSO. In our tool, the strategy for changing the value of inertia according to the equation 5 is implemented in this class.

ParticleUpdate: This class is a representative base class for defining the customized strategies to update the swarm particles.

ParticleUpdateSimple: Our strategies for updating the particles' velocities and positions are implemented in this class.

Particle: It is an abstract class providing a generic representation for the structure and management of particles in the swarm.

6.3.3 Editor

The implementation package of *Editor* deals with the GUI of CØdŞ. The controller of our CØdŞ application "CodS.java" initiates the execution of our tool. The five classes of this package are responsible for performing the following jobs:

1. Manage the GUI of CØdŞ.
2. Manage the CØdŞ project directory.
3. File reading and writing.
4. Managing the CØdŞ help content.

6.4 CØdŞ Process Flow

Figure 6.4 shows the main interface of our tool CØdŞ. CØdŞ manages the activity of M2C transformation in terms of projects. Separate projects are created for different

transformation processes. By default, these projects are stored in the 'D' directory under the "CodS" folder. Each project directory of CØdŞ has the following structure:

D:\ CodS\<*Project Name*> \ Input \

D:\ CodS\ <*Project Name*> \ Output \ Predicates \

D:\ CodS \ <*Project Name*> \ Output \ Java Code \

The complete information of CØdŞ tool and its user manual is detailed out in Appendix B. The process of automatic code generation implemented in CØdŞ can be divided into four major steps, as discussed below.

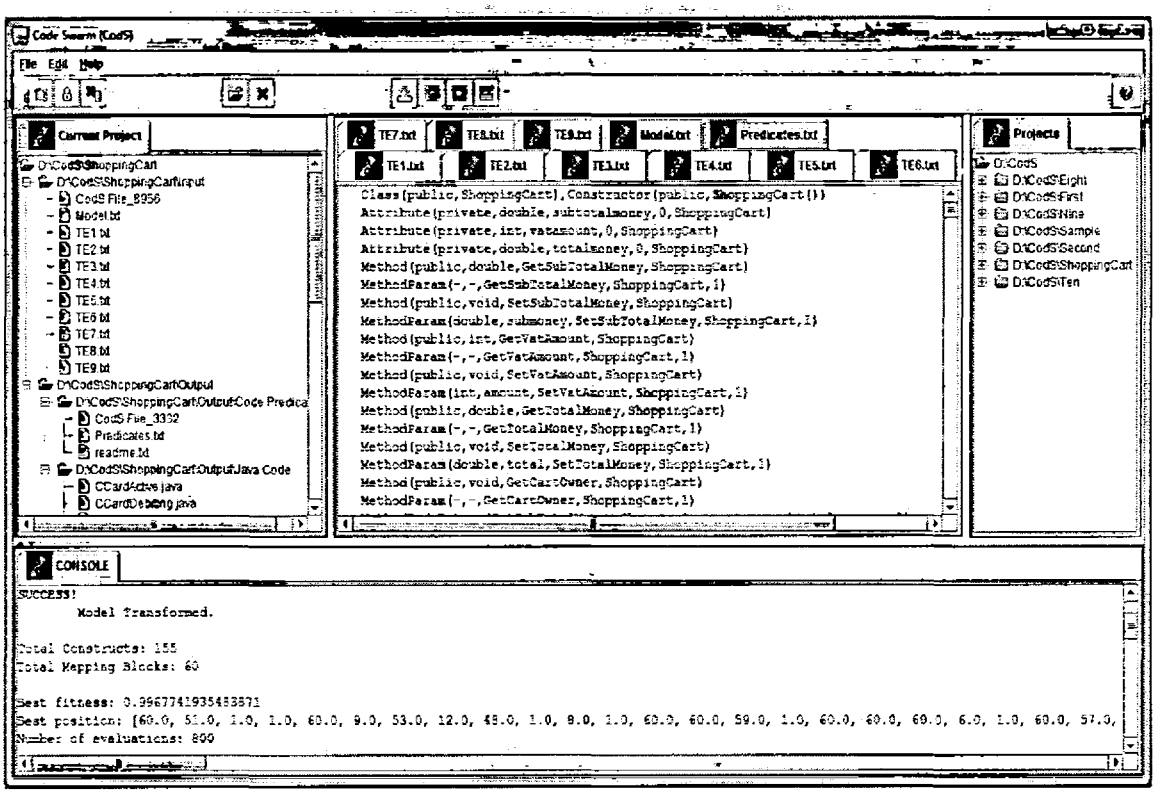


Figure 6.4 Main interface of CØdŞ

6.4.1 Import Training Data

After creating a new CØdŞ project from the CØdŞ main interface, user first needs to import the training data to build the knowledge base. This is done by using the option available

in the menu bar and toolbar of the CØdŞ interface. Training data can comprise of multiple text files containing a set of mapping blocks. When the user browses the training data files, these files are read and stored in the "Input" folder of the current CØdŞ project. Besides the classes that handle GUI of the CØdŞ, the logical class involved in building the knowledge base is "PredicateParser". A sample training data file is show in Figure 6.5.

```
Begin b1
Class(Employee):Class(public,Employee),Constructor(public,Employee())
Attribute(Empid,int,0,Employee,unique):Attribute(private,int,empid,0,Employee)
Attribute(EName,String,null,Employee,_):Attribute(private,String,ename,null,Employee)
Attribute(StartDate.Date,null,Employee,_):Attribute(private,Date,startdate,null,Employee)
Operation(GetName,Employee,String):Method(public,String,GetName,Employee)
OperationParam(-,-,GetName,Employee,1):MethodParam(-,-,GetName,Employee,1)
Operation(SetName,Employee,void):Method(public,void,SetName,Employee)
OperationParam(Name,String,SetName,Employee,1):MethodParam(String,name,SetName,Employee,1)
Operation(GetStartDate,Employee,Date):Method(public,Date,GetStartDate,Employee)
OperationParam(-,-,GetStartDate,Employee,1):MethodParam(-,-,GetStartDate,Employee,1)
Operation(SetStartDate,Employee,void):Method(public,void,SetStartDate,Employee)
OperationParam(SDate,Date,SetStartDate,Employee,1):MethodParam(Date,sdate,SetStartDate,Employee,1)
Operation(GetEmpPosition,Employee,String):Method(public,String,GetEmpPosition,Employee)
OperationParam(-,-,GetEmpPosition,Employee,1):MethodParam(-,-,GetEmpPosition,Employee,1)
Operation(SetEmpPosition,Employee,void):Method(public,void,SetEmpPosition,Employee)
OperationParam(EPosition,String,SetEmpPosition,Employee,1):MethodParam(String,eposition,SetEmpPosition,Employee,1)
OpBodyReturn(GetName,Employee,Output,Name,-,1):MethodBodyReturn(GetName,Employee,return,ename,-,1)
OpBodyAssign(SetName,Employee,EName=Name,-,1):MethodBodyAssign(SetName,Employee,ename=name,-,1)
OpBodyReturn(GetStartDate,Employee,Output,StartDate,-,1):MethodBodyReturn(GetStartDate,Employee,return,startdate,-,1)
OpBodyAssign(SetStartDate,Employee,StartDate=SDate,-,1):MethodBodyAssign(SetStartDate,Employee,startdate=sdate,-,1)
OpBodyDeclaration(GetEmpPosition,Employee,String,EmpTitle,null,-,1):MethodBodyDeclaration
(GetEmpPosition,Employee,String,emptitle,null,-,1)
OpBodyFn(GetEmpPosition,Employee,EmpTitle,GetTitle,worksat,-,2):MethodBodyFn(GetEmpPosition,Employee,emptitle,worksat,GetTitle,-,2)
OpBodyFnParam(GetEmpPosition,Employee,GetTitle,-,2,1):MethodBodyFnParam(GetEmpPosition,Employee,GetTitle,-,2,1)
OpBodyReturn(GetEmpPosition,Employee,Output,EmpTitle,-,3):MethodBodyReturn(GetEmpPosition,Employee,return,emptitle,-,3)
OpBodyFn(SetEmpPosition,Employee,-,SetTitle,worksat,-,1):MethodBodyFn(SetEmpPosition,Employee,-,worksat,SetTitle,-,1)
OpBodyFnParam(SetEmpPosition,Employee,SetTitle,EPosition,1,1):MethodBodyFnParam(SetEmpPosition,Employee,SetTitle,eposition,1,1)
State(EmpInitial,Employee):Class(public,EmpInitial,Employee)
Operation(Entry,EmpInitial,void):Method(public,void,Entry,EmpInitial)
OperationParam(-,-,Entry,EmpInitial,1):MethodParam(-,-,Entry,EmpInitial,1)
Operation(Exit,EmpInitial,void):Method(public,void,Exit,EmpInitial)
OperationParam(-,-,Exit,EmpInitial,1):MethodParam(-,-,Exit,EmpInitial,1)
Operation(doActivity,EmpInitial,void):Method(public,void,doActivity,EmpInitial)
OperationParam(-,-,doActivity,EmpInitial,1):MethodParam(-,-,doActivity,EmpInitial,1)
State(EmpWorking,Employee):Class(public,EmpWorking,Employee)
Operation(Entry,EmpWorking,void):Method(public,void,Entry,EmpWorking)
OperationParam(-,-,Entry,EmpWorking,1):MethodParam(-,-,Entry,EmpWorking,1)
Operation(Exit,EmpWorking,void):Method(public,void,Exit,EmpWorking)
OperationParam(-,-,Exit,EmpWorking,1):MethodParam(-,-,Exit,EmpWorking,1)
Operation(doActivity,EmpWorking,void):Method(public,void,doActivity,EmpWorking)
OperationParam(-,-,doActivity,EmpWorking,1):MethodParam(-,-,doActivity,EmpWorking,1)
State(EmpRetired,Employee):Class(public,EmpRetired,Employee)
```

Figure 6.5 Training data file

6.4.2 Import Source Model

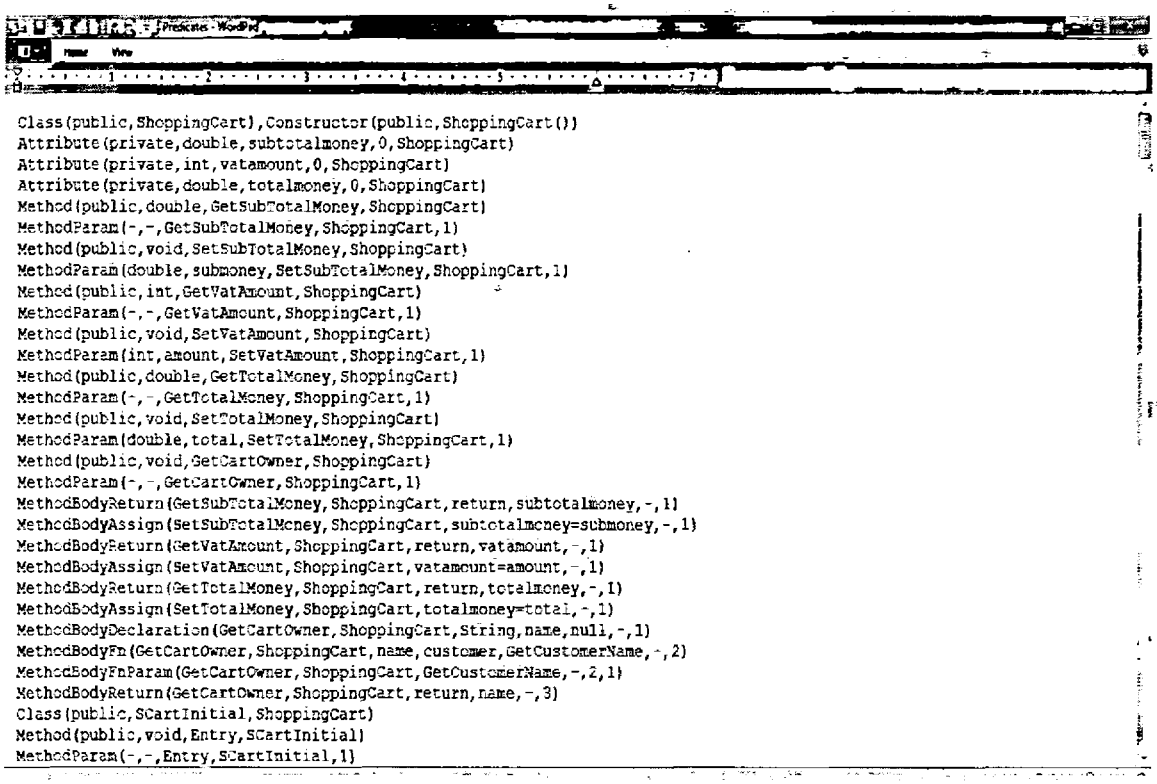
After building up the knowledge base, user selects the option for importing the source models which are to be transformed into target code. All the input source models are present in a single text file i.e. all the class and state models along with the ASL statements belonging to one application system are stored in one text file. This input file is stored in the "Input" folder of the current CØdŞ project. The class of "PredicateParser" is mainly responsible for performing the back end tasks of organizing and managing the input source model constructs. Figure 6.6 shows a file containing the input model constructs to be transformed.

Figure 6.6 Input source models file

6.4.3 Transform Model

The third option user needs to select is the option for transforming the input source model. This is the most important step in which an optimal solution is generated by using the PSO algorithm. Moreover, this optimal solution is then used to transform the SMCs of input models into TCCs. Majorly, the implementation package of PSO and classes of "FitnessCalculator" and "ModelTransformer" are involved in performing this step.

The target code predicates produced at the end of this step are stored in a text file named "Predicates". This text file is stored in the "Output\Code Predicates" folder of the current CØdŞ project. Figure 6.7 shows a sample predicate file generated for the input models of an application of an "Online Shopping System".



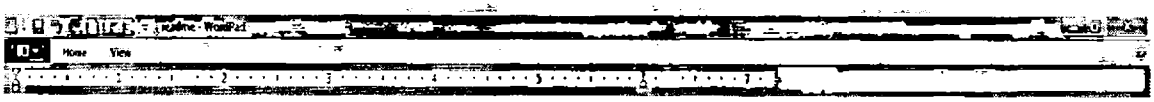
```
Class(public,ShoppingCart),Constructor(public,ShoppingCart())
Attribute(private,double,subtotalmoney,0,ShoppingCart)
Attribute(private,int,vatamount,0,ShoppingCart)
Attribute(private,double,totalmoney,0,ShoppingCart)
Method(public,double,GetSubTotalMoney,ShoppingCart)
MethodParam(-,-,GetSubTotalMoney,ShoppingCart,1)
Method(public,void,SetSubTotalMoney,ShoppingCart)
MethodParam(double,submoney,SetSubTotalMoney,ShoppingCart,1)
Method(public,int,GetVatAmount,ShoppingCart)
MethodParam(-,-,GetVatAmount,ShoppingCart,1)
Method(public,void,SetVatAmount,ShoppingCart)
MethodParam(int,amount,SetVatAmount,ShoppingCart,1)
Method(public,double,GetTotalMoney,ShoppingCart)
MethodParam(-,-,GetTotalMoney,ShoppingCart,1)
Method(public,void,SetTotalMoney,ShoppingCart)
MethodParam(double,total,SetTotalMoney,ShoppingCart,1)
Method(public,void,GetCartOwner,ShoppingCart)
MethodParam(-,-,GetCartOwner,ShoppingCart,1)
MethodBodyReturn(GetSubTotalMoney,ShoppingCart,return,subtotalmoney,-,1)
MethodBodyAssign(SetSubTotalMoney,ShoppingCart,subtotalmoney=submoney,-,1)
MethodBodyReturn(GetVatAmount,ShoppingCart,return,vatamount,-,1)
MethodBodyAssign(SetVatAmount,ShoppingCart,vatamount=amount,-,1)
MethodBodyReturn(GetTotalMoney,ShoppingCart,return,totalmoney,-,1)
MethodBodyAssign(SetTotalMoney,ShoppingCart,totalmoney=total,-,1)
MethodBodyDeclaration(GetCartOwner,ShoppingCart,String,name,null,-,1)
MethodBodyFn(GetCartOwner,ShoppingCart,name,customer,GetCustomerName,-,2)
MethodBodyFnParam(GetCartOwner,ShoppingCart,GetCustomerName,-,2,1)
MethodBodyReturn(GetCartOwner,ShoppingCart,return,name,-,3)
Class(public,SCartInitial,ShoppingCart)
Method(public,void,Entry,SCartInitial)
MethodParam(-,-,Entry,SCartInitial,1)
```

Figure 6.7 Predicates file "Predicates.txt"

Another text file named "readme" is generated in this step. A sample readme file is shown in Figure 6.8. This file contains the following information about the M2C transformation.

1. Total input model constructs.
2. Total mapping blocks in training data.
3. Best fitness value of the optimal solution.
4. Mapping block numbers selected in the optimal solution for every input model construct.
5. Number of evaluations performed by PSO.
6. Details of the source construct and mapping block, if an exact match of input model construct is not found in the mapping block.

This information is also displayed on the console of our CØdŞ application.



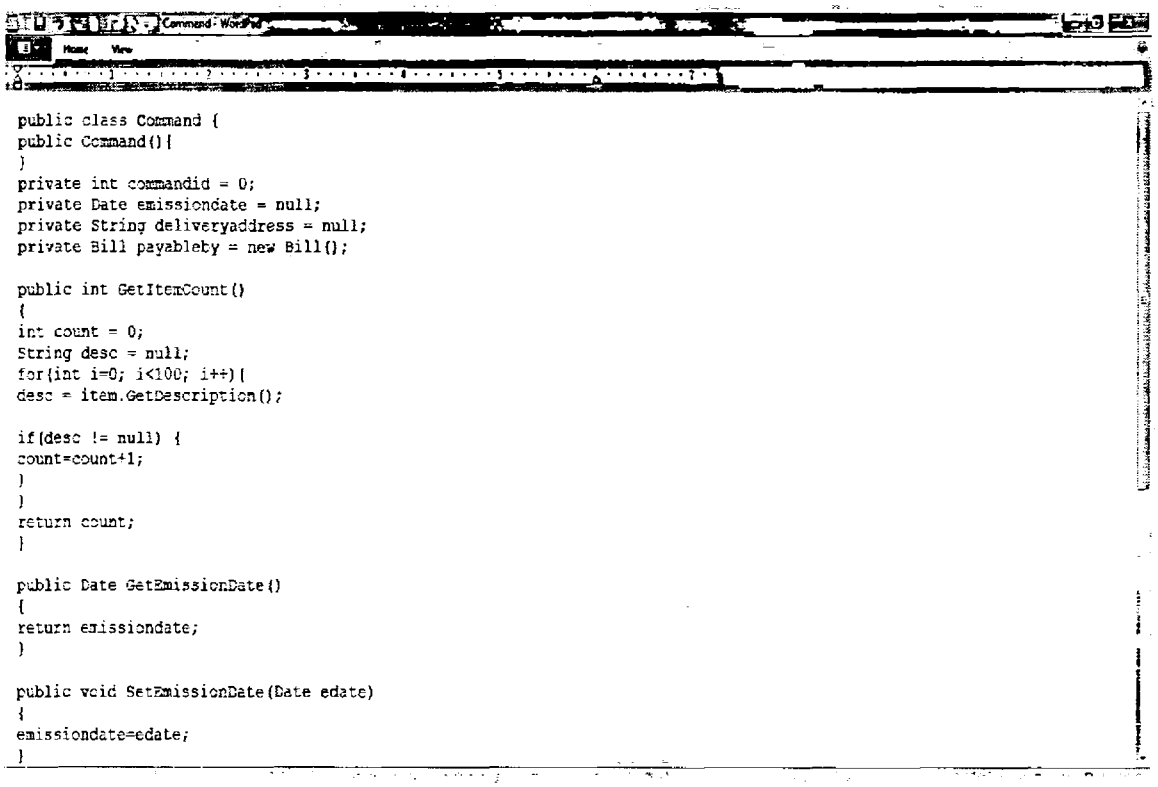
```
Total Constructs
155
Total Mapping Blocks
60
Best fitness: 0.9903225806451613
Best position: [1.0, 1.0, 60.0, 45.0, 1.0, 1.0, 1.0, 1.0, 60.0, 60.0, 1.0, 60.0, 1.0, 1.0, 60.0, 60.0, 1.0, 16.0, 1.0,
60.0, 1.0, 60.0, 1.0, 36.0, 58.0, 60.0, 60.0, 1.0, 60.0, 1.0, 1.0, 1.0, 1.0, 60.0, 1.0, 1.0, 60.0, 1.0, 60.0, 21.0, 60.0,
60.0, 60.0, 55.0, 60.0, 1.0, 60.0, 22.0, 60.0, 60.0, 60.0, 1.0, 60.0, 14.0, 8.0, 60.0, 11.0, 1.0, 23.0, 13.0, 32.0, 1.0,
60.0, 1.0, 1.0, 1.0, 60.0, 1.0, 1.0, 1.0, 60.0, 60.0, 60.0, 60.0, 29.0, 1.0, 60.0, 41.0, 1.0, 1.0, 1.0, 60.0, 60.0, 60.0,
2.0, 60.0, 15.0, 60.0, 60.0, 1.0, 1.0, 2.0, 1.0, 60.0, 39.0, 60.0, 1.0, 23.0, 52.0, 60.0, 1.0, 12.0, 60.0, 1.0, 60.0,
60.0, 60.0, 60.0, 60.0, 4.0, 1.0, 58.0, 1.0, 60.0, 1.0, 60.0, 60.0, 60.0, 1.0, 60.0, 60.0, 39.0, 1.0, 33.0, 1.0, 60.0,
60.0, 1.0, 1.0, 1.0, 60.0, 60.0, 60.0, 60.0, 60.0, 1.0, 60.0, 60.0, 60.0, 1.0, 1.0, 1.0, 60.0, 1.0, 1.0, 1.0, 55.0, 41.0,
60.0, 60.0, 60.0, 60.0, 13.0, 41.0, 60.0]
Number of evaluations: 800

Doubtful Transformations: 3
Model Construct: Association(_,1,1,n,_,ShoppingCart,ItemToPurchase)
Target Transformation: Attribute(private,ShoppingCart,shoppingcart,ShoppingCart(),ItemToPurchase)
Target Mapping Used: Association(_,1,1,n,_,ShoppingCart,ItemToPurchase)
Relevance Score: 0.67
Model Construct: Association(_,1,0,n,_,Product,ItemToPurchase)
Target Transformation: Attribute(private,Product,product,Product(),ItemToPurchase)
Target Mapping Used: Association(_,1,0,n,_,Product,ItemToPurchase)
Relevance Score: 0.67
Model Construct: Association(_,1,0,n,_,Customer,ShoppingCart)
Target Transformation: Attribute(private,Customer,customer,Customer(),ShoppingCart)
Target Mapping Used: Association(_,1,0,n,_,Customer,ShoppingCart)
Relevance Score: 0.67
```

Figure 6.8 Model transformation "readme.txt"

6.4.4 Generate Java Code

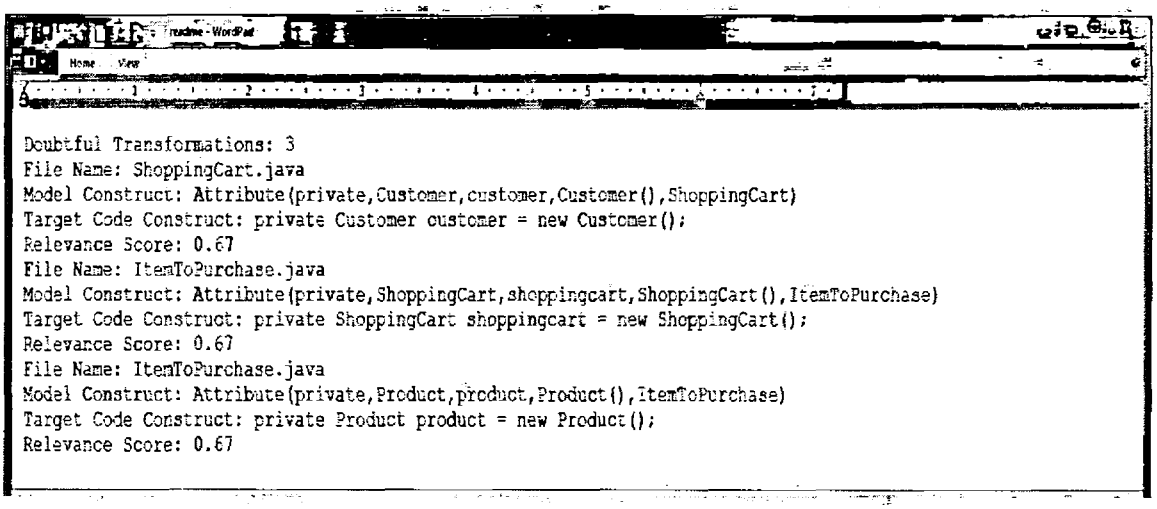
Finally, user selects the option of generating Java code corresponding to the code predicates produced in the previous step. The output produced by this step is a set of Java files containing the complete Java code statements. The class of "CodeGenerator" is mainly responsible for performing this task. Figure 6.9 shows a sample Java code file generated by CØdŞ for the class "Command". These java files are stored in the "Output\Java Code" folder of the CØdŞ project.

Moreover, a readme file is also generated in this step. This readme file contains the information about the input model constructs for which partial or no exact match was found in the mapping blocks. A sample readme file is shown in Figure 6.10.



```
public class Command {
public Command(){
}
private int commandid = 0;
private Date emissiondate = null;
private String deliveryaddress = null;
private Bill payableby = new Bill();

public int GetItemCount()
{
int count = 0;
String desc = null;
for(int i=0; i<100; i++){
desc = item.GetDescription();

if(desc != null) {
count=count+1;
}
}
return count;
}

public Date GetEmissionDate()
{
return emissiondate;
}

public void SetEmissionDate(Date edate)
{
emissiondate=edate;
}
```

Figure 6.9 Java code file "Command.java"



```
Doubtful Transformations: 3
File Name: ShoppingCart.java
Model Construct: Attribute(private,Customer,customer,Customer(),ShoppingCart)
Target Code Construct: private Customer customer = new Customer();
Relevance Score: 0.67
File Name: ItemToPurchase.java
Model Construct: Attribute(private,ShoppingCart,shoppingcart,ShoppingCart(),ItemToPurchase)
Target Code Construct: private ShoppingCart shoppingcart = new ShoppingCart();
Relevance Score: 0.67
File Name: ItemToPurchase.java
Model Construct: Attribute(private,Product,product,Product(),ItemToPurchase)
Target Code Construct: private Product product = new Product();
Relevance Score: 0.67
```

Figure 6.10 Code generation "readme.txt"

Chapter 7

CASE STUDY

7.1 Introduction

This chapter is dedicated to provide a thorough explanation of the case study used for the validation of our proposed approach. We have chosen a real-life example of an Elevator Control System (ECS) to generate the implementation code by utilizing our approach. In this chapter, we describe the ECS by considering two UML design diagrams. Particularly, the static structural design view of ECS is modeled by using the UML class diagram. As ECS is a real-time reactive system, UML state model is utilized to illustrate the dynamic view of the system. ASL is used to express the method bodies and behavioral logic of the ECS at the design level.

The rest of this chapter is structured as follows. Section 7.2 gives an overview of the ECS and states its functional requirements. The class diagram of ECS is described in Section 7.3. Section 7.4 demonstrates the state models corresponding to the ECS classes.

7.2 Elevator Control System

This section is divided into two subsections. The first subsection provides an overview of the scope of the Elevator Control System (ECS). The functional requirements of the system are detailed out in the second subsection.

7.2.1 Scope of the ECS

The task of an ECS is to control and manage the elevator of a building. The major object of the ECS is the 'Elevator', which has the basic function of moving up and down, open and close doors and picking up passengers from different floors of the building. The elevator is supposed to be used in a building having floors from 1 to *max*, where *max* is the maximum number of floors in a building. The first floor of the building is a lobby.

The elevator has the car call buttons corresponding to each floor of the building. On each floor except for the top floor and the lobby, there are two hall call buttons for the passengers to call the elevator for going up and down. At the top floor, there is only one down hall call button and in the lobby, there is only one up hall call button. When the elevator stops at a floor, the doors are opened and the car lantern indicating the current direction the elevator is going is

illuminated so that the passengers can get to know the current moving direction of the elevator. The elevator changes its speed from slow to fast while visiting the floors of the building. In order to ensure the safety of the elevator, the elevator is equipped with the emergency brakes which are triggered to force stop the elevator during unsafe conditions.

7.2.2 Functional Requirements

Process Hall Call: When the passenger requests an elevator by pressing the hall call button, the light of the hall call button is turned on. If the elevator is idle, it starts moving towards the requested floor immediately, otherwise the requested floor number is saved in the queue maintained by the elevator. When the hall call button is released, the button light is turned off.

Process Car Call: When the passenger enters the elevator, (s)he presses the car call button to express the desired destination floor. The pressed car button is illuminated, doors are closed and the desired moving direction is determined. The elevator starts moving towards the destination floor with the car lantern indicating the current moving direction of the elevator and the car position indicator showing the destination floor number.

Move/Stop the Elevator: When the elevator starts moving towards the desired floor, it moves from slow speed to a fast speed. The elevator moves with fast speed only when the source and the destination floors are more than two floors apart. When the elevator stops at a desired floor, the doors are opened, car lantern is cleared to show the next moving direction of the elevator and the car position indicator is refreshed to indicate the current floor of the building.

Open/Close the Doors: The doors of the elevator are closed before the elevator starts moving from the source floor and are opened after the elevator stops at a destination floor. However, when the doors are closing and are not fully closed, if there are passengers who want to get into the elevator, the doors are opened again.

Trigger Emergency Brakes: During unsafe conditions, the elevator controller triggers the emergency brakes to force stop the elevator at a floor. When the unsafe condition ends, the brakes are released to continue the normal operation of the elevator.

7.3 Class Model

Figure 7.1 shows the class diagram of our ECS consisting of 14 classes. The responsibilities of these classes are described below.

Building: The ECS is deployed in a building to move the passengers up and down with the help of an elevator.

ElevatorControl: ElevatorControl is the central controlling object in the ECS. It is responsible for receiving input messages from the outside world, passing these messages on for further processing by the ECS and sending response and output messages to the hardware and environmental objects of ECS.

Elevator: Elevator is the major object in the ECS. It is being controlled by the ElevatorControl to move up and down in the building at different speeds and to make stops at different floors when needed.

EmergencyBrake: In case of an emergency or an exceptional situation, EmergencyBrake of an elevator is triggered by the ElevatorControl.

Button: In our ECS, the Button class generalizes two sub-classes - HallCallButton and CarButton. The ElevatorControl communicates with the Button objects, gets the information whether a button is pressed and in turn controls the illumination of the button lights.

HallCallButton: HallCallButton exists in pair at each floor, except for the top floor and lobby. ElevatorControl commands the elevator in response to the HallCallButton press and gives feedback to HallCallButton lights.

CarButton: For each floor in the building, there exists a CarButton in the Elevator. The ElevatorControl moves the elevator according to the press of the CarButton and is in charge of turning the CarButton lights on and off.

Floor: The ECS is supposed to be used in a building having floors from 1 to *max*, max being the total number of floors in a building. Each floor has a pair of HallCallButtons for calling the elevator, except for the first floor and lobby, which have a single HallCallButton.

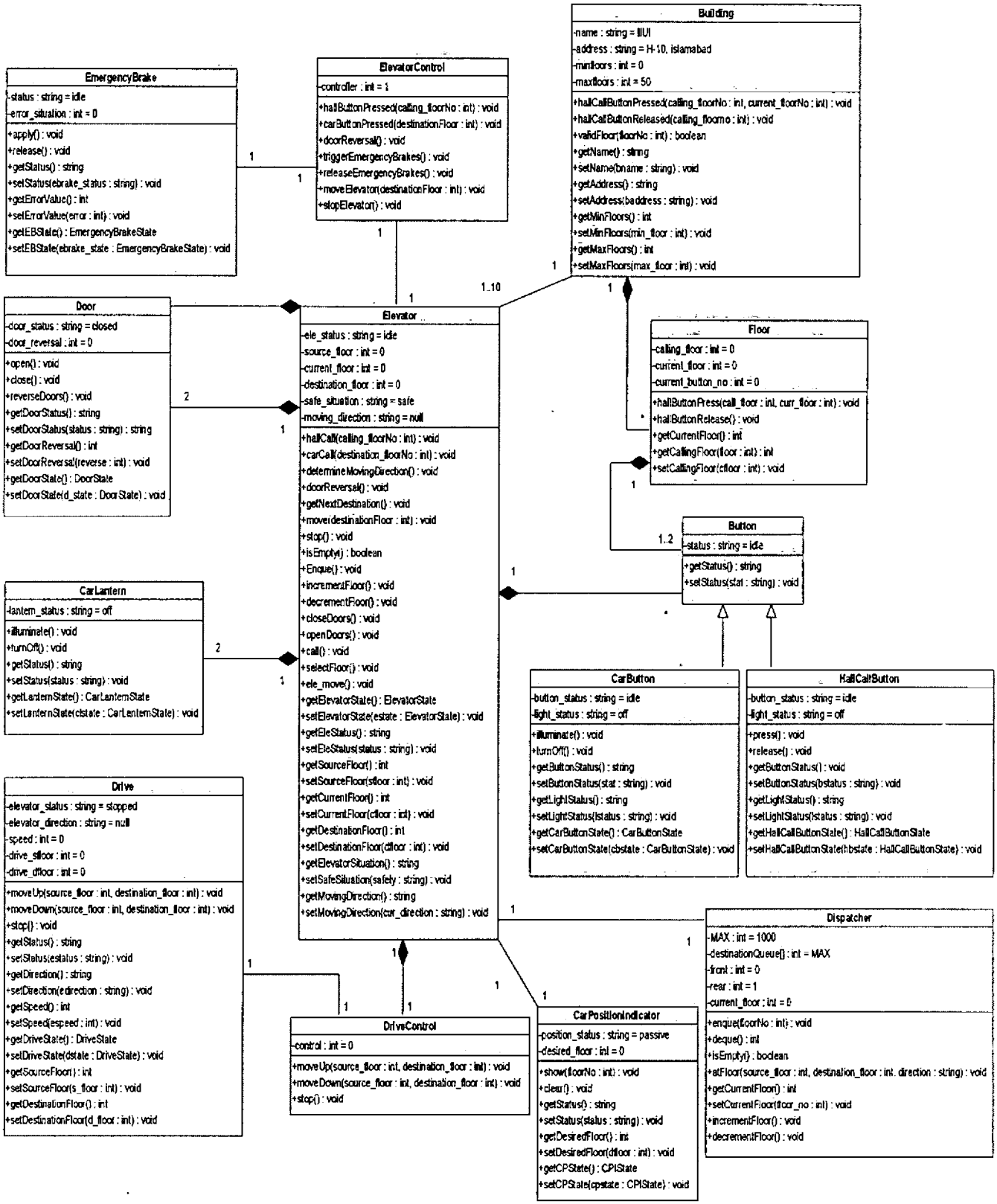


Figure 7.1 Class diagram of the 'Elevator Control System'

CarPositionIndicator: CarPositionIndicator is used to inform the passengers about the current position of the Elevator. When the Elevator is at rest, it indicates the current floor whereas the desired floor is shown to the passengers when the elevator is moving.

CarLantern: CarLantern are two in number, for indicating the up/down moving direction of the elevator to the passengers. One of the two car lanterns is illuminated according to the current moving direction of the elevator.

Door: There are two doors in the system. The ElevatorControl commands the Door object to open, close or make a door reversal according to the situation.

DriveControl: DriveControl is responsible for controlling the elevator Drive.

Drive: Drive controls the movement of the elevator. It moves the elevator up and down and makes stops at different floors.

Dispatcher: Dispatcher is an important component of the software system although it does not control the actual elevator components. The main function of the dispatcher is to calculate the target moving direction and destination for the elevator.

The method bodies of these classes are expressed using ASL. As an example, the body of the method 'getNextDestination' in the 'Elevator' class is expressed in ASL below. The keywords of ASL are expressed with the bold font style.

```
Boolean queue_empty = FALSE

[queue_empty] = op9:isEmpty[] on dispatcher

if queue_empty == FALSE then

   Integer next_floor = -1

   [next_floor] = op3:deque[] on dispatcher

   Op5:carCall[next_floor] on this

endif

else
```

```
        op4:stop[] on drivecontrol
        ele_status = "idle"
        for i in {1,2}
           op2:close[] on door[i]
           op2:turnoff[] on carlantern[i]
        endfor
        source_floor = current_floor
        destination_floor = -1
        op2:turnoff[] on carbutton[current_floor]
        op2:clear[] on caprpositionindicator
        op1:show[current_floor] on carpositionindicator
    endif
```

In this method, the next destination of the elevator is determined. If the waiting queue of the elevator is non-empty, the elevator takes its next destination floor from the queue and moves towards it. However, if the waiting queue is empty i.e. there are no passengers waiting for the elevator on any floor of the building, the elevator is stopped at the current floor, doors are closed, car lantern is turned off and the car position indicator is set to show the current floor of the elevator.

7.4 State Model

This section illustrates the state diagrams corresponding to the classes of the ECS to show the behavior of the reactive objects.

7.4.1 State model of Elevator

Figure 7.2 shows the state model of the major reactive object in the ECS, the Elevator. The state model consists of 13 states and 29 transitions, including 3 self transitions. Within the

state, the 'entry/Activity' and 'do/Activity' illustrate that there are activities in the entry and do methods of the state respectively. These activities are expressed in ASL at the design level.

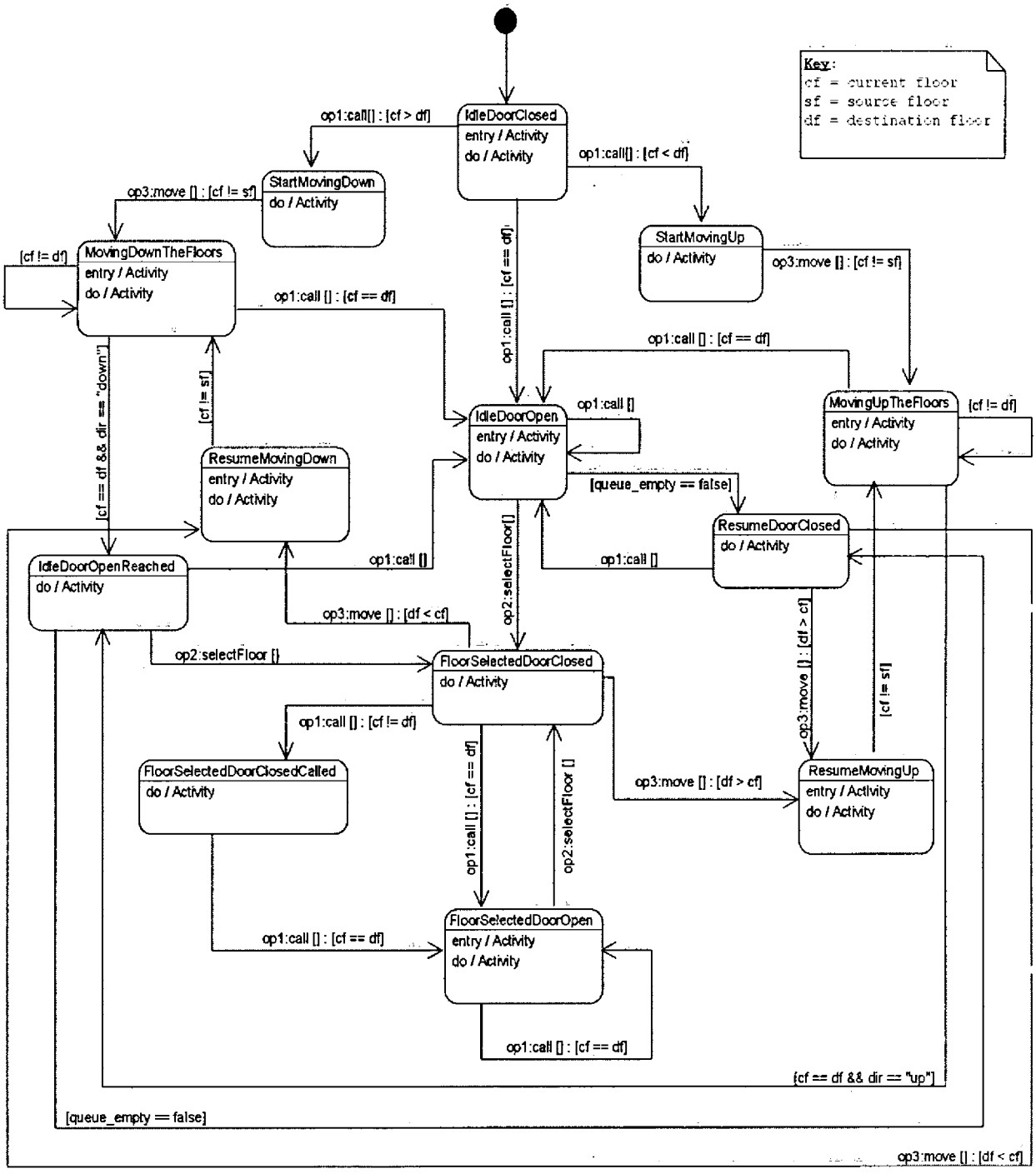


Figure 7.2 State model of the 'Elevator'

7.4.2 State Model of EmergencyBrake

Figure 7.3 demonstrates the state model of the EmergencyBrake of the ECS.

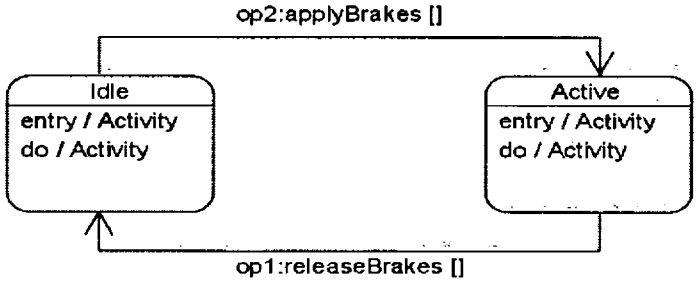


Figure 7.3 State model of the 'EmergencyBrake'

7.4.3 State Model of HallCallButton

The state model of the class HallCallButton is shown in Figure 7.4.

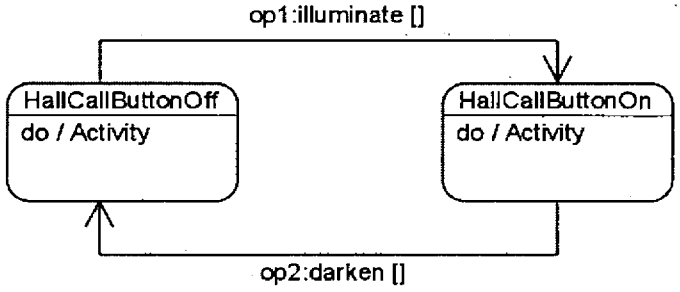


Figure 7.4 State model of the 'HallCallButton'

7.4.4 State Model of CarButton

Figure 7.5 shows the state model of the CarButton class.

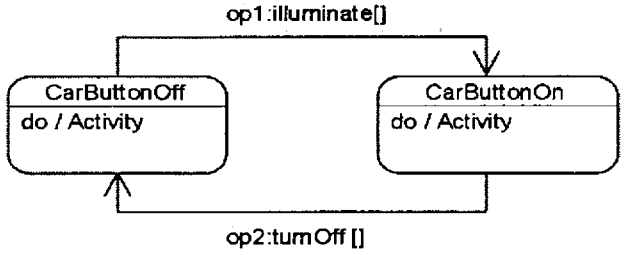


Figure 7.5 State model of the 'CarButton'

7.4.5 State Model of CarPositionIndicator

Figure 7.6 shows the behavior of the CarPositionIndicator class in the ECS.

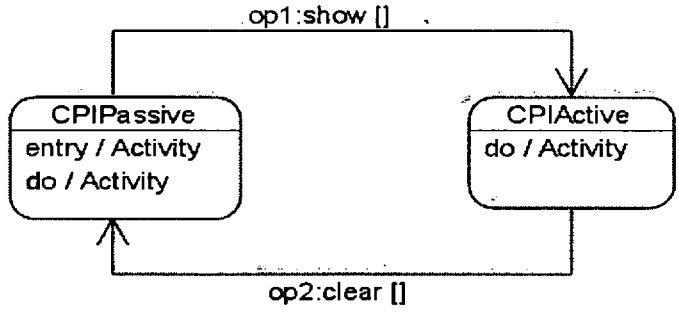


Figure 7.6 State model of the 'CarPositionIndicator'

7.4.6 State Model of CarLantern

The state model of the car lantern is shown in Figure 7.7.

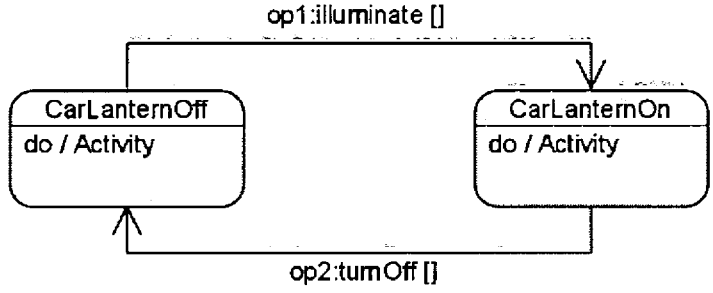


Figure 7.7 State model of the 'CarLantern'

7.4.7 State Model of Door

Figure 7.8 shows the dynamic behavior of the Door class of the ECS.

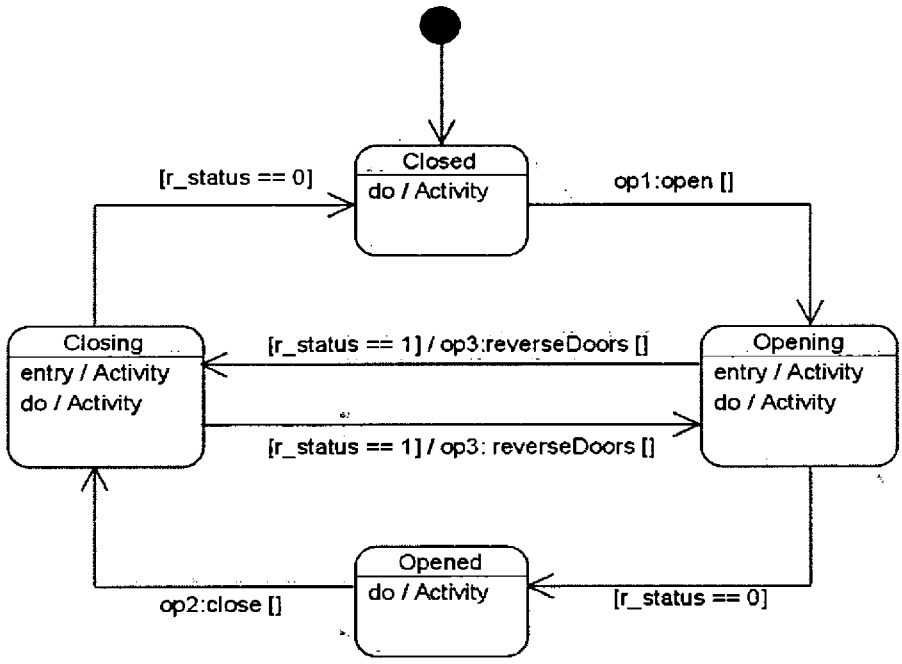


Figure 7.8 State model of the 'Door'

7.4.8 State Model of Drive

The state model of Drive is illustrated in Figure 7.9.

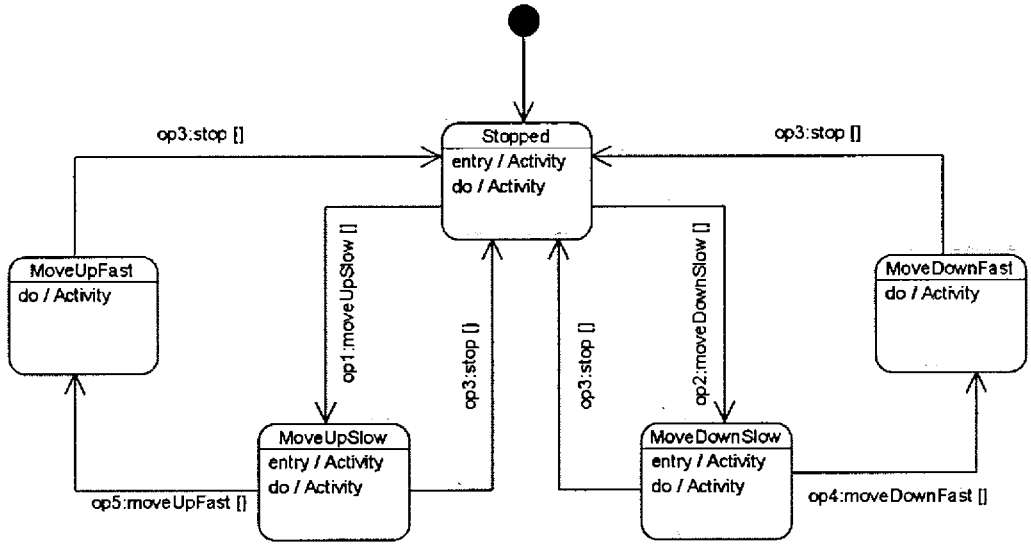


Figure 7.9 State model of the 'Drive'

Chapter 8

EVALUATION

8.1 Introduction

This chapter is devoted to explain, discuss and evaluate the results of the experiment performed to validate our proposed approach for automatic code generation. Section 8.2 presents and discusses the results of the application of our approach on the comprehensive models of the Elevator Control System described in Chapter 7. The outcome of 10-fold cross validation is discussed in Section 8.3. Section 8.4 presents the comparison of our approach with the existing code generation approaches and tools. Finally, the overall assessment of our approach in terms of benefits and hmitations is given in Section 8.5.

8.2 Elevator Control System

In this section, we describe the experimental setting and present and discuss the results for the application of our approach to the class and state models of the ECS.

8.2.1 Experimental Setting

The steps of our experiment are summarized below.

1. In the first step, we express the training data as predicates. The class and state models of 9 different software systems are used as the training data. The complete models of the training data can be found in Appendix C. For ECS, the models M2-M10 constitute our training data. We stored the predicates of our training data in 9 text files i.e. one text file contains the class and state model predicates of one software system. This training data is given as input to our CØdŞ tool. Table 8.1 shows the details of the model constructs and mapping blocks of the training data.

Table 8.1 Training data

Model	M2	M3	M4	M5	M6	M7	M8	M9	M10
Model constructs	106	96	135	145	249	155	202	135	200
Total mapping blocks	60								

2. During the second step, we express the constructs of the input source models to be transformed as predicates. These predicates are stored in a single text file, which is given as input to the CØdŞ. The input source models are demonstrated in Chapter 7.

3. The input source models are transformed to generate code in terms of predicates through CØdŞ tool.

4. The code predicates generated by CØdŞ are transformed to generate complete Java code files.

In order to check the correctness of the solution generated by CØdŞ, we used the following formula to calculate the percentage of the correctly produced code.

$$\text{Correctness (\%)} = \frac{\text{Number of correctly transformed constructs}}{\text{Total number of model constructs}} * 100 \quad \ldots\ldots\ldots\ldots (6)$$

As the proposed approach uses a stochastic algorithm, for which two different executions may produce different results for the same model, we choose the best result for each input source model from five executions.

8.2.2 Results and Discussion

Figure 8.1 shows the screenshot of the result generated by CØdŞ. According to the result, from the total of 703 input model constructs, there are 27 doubtful transformations which constitute 3.84% of the total input model constructs. By *doubtful transformation* we mean that for 27 input model constructs, exact match was not present in the optimized solution searched by our approach, either because their matching constructs do not exist in the training data or our approach is unable to find them. Because of these doubtful transformations, the fitness value is penalized, as their exact match is not present in the optimal solution. Our tool highlights these doubtful transformations separately so that users can analyze their correctness manually in order to avoid the generation of incorrect code. Moreover for the doubtful transformations, CØdŞ also calculates and shows the relevance between the input model constructs and the selected transformation from the training data in terms of the relevance score to facilitate its intended users.

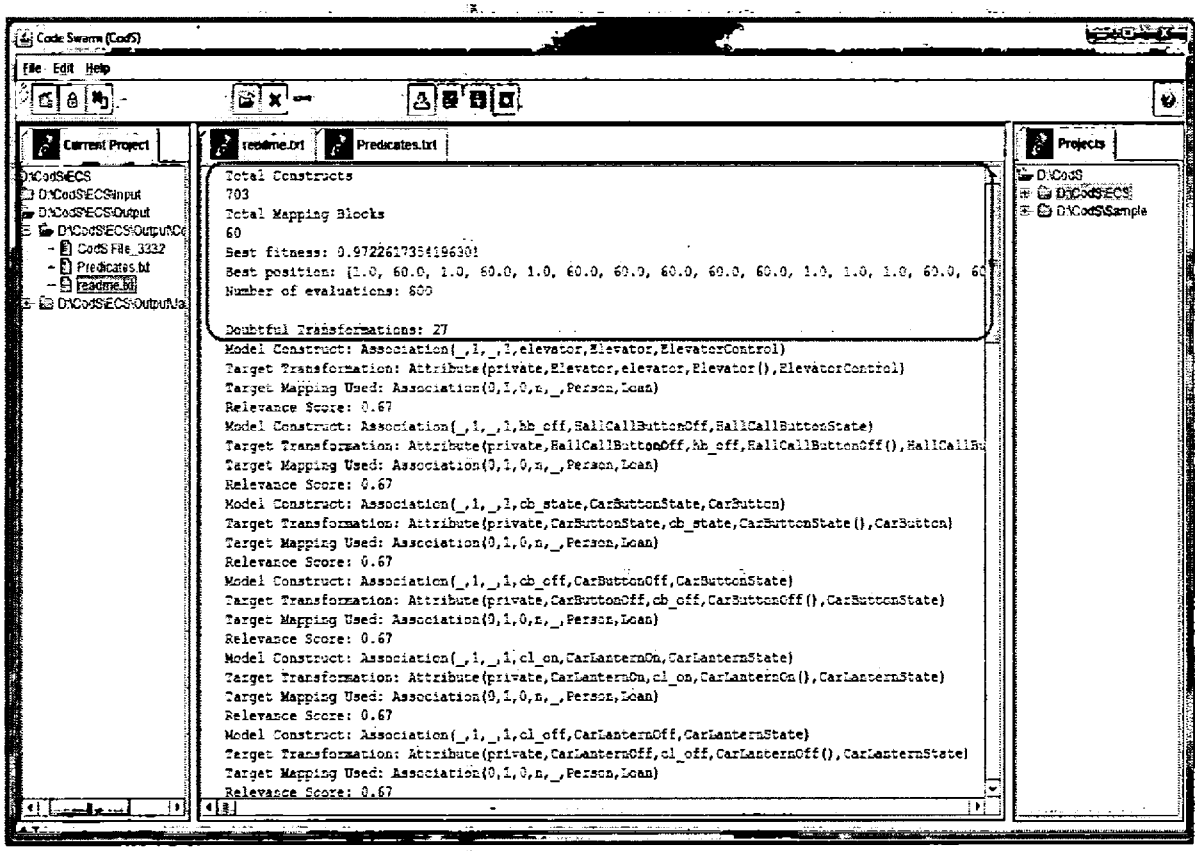


Figure 8.1 CØdŞ screenshot highlighting *doubtful transformations*

The complete code generated by CØdŞ can be found in Appendix D. When we analyzed the generated code corresponding to these doubtful transformations manually, we found that the code generated for 22 of these doubtful transformations is correct. This is due to the reason that our fitness function is intelligent enough to look for the nearest match, in case if no exact match is found in the training data.

For the remaining 5 input model constructs, their matching constructs are not even present in the training data. For this reason, PSO could not find their corresponding correct matching transformations. However, our approach does not leave out these constructs and maps these input model constructs to their nearest match from the training data. For example, the following input model construct is not present in the training data.

```
Composition(_,10,_,1,carbutton,CarButton,Elevator)
```

However, PSO chooses its nearest match and proposes the following transformation.

```
Composition(_,1,_,1,_,Address,Customer)
```

This transformation is very close to the desired transformation, as our approach is able to find the transformation from the training data with the similar model construct (predicate name). Again, this supports our choice of the good fitness function. But since the key parameters (2nd parameter) of the two constructs are different, we considered this transformation as incorrect. The chosen transformation creates only one instance of the 'CarButton' in the 'Elevator' class instead of the desired object array of size 10.

However besides these doubtful transformations, there are 3 input model constructs for which our approach suggests incorrect transformation. Although their matching constructs are present in the training data but our approach is unable to find them. This is because of the reason that our approach uses PSO which utilizes stochastic search instead of the exhaustive search while looking out an optimal solution. Our tool CØdŞ is also able to keep track of these input model constructs and highlights them separately so that the intended users can come to know about the missing code statements. Figure 8.2 shows the screenshot of the CØdŞ tool highlighting these missing code constructs.

Table 8.2 and Table 8.3 summarize the execution and the post-analysis execution results obtained for the ECS respectively. From Table 8.2 it can be inferred that our approach finds the exact transformation for 673 input model constructs from the training data. However in Table 8.3, it can be seen that the number of correctly transformed constructs rises to 695. This implies that our approach proposes 22 correct transformations. This is another sign of our strong and sharp fitness function.

The best fitness value obtained for an optimal solution is 0.9722 whereas the correctness of the generated code is 0.9886 (98.86%). Here we can observe that the fitness value of the optimal solution is less than the code correctness. This is because of the reason that for 22 correct doubtful transformations, the correctness is 1.0 (100%) but the individual fitness values of these constructs are less than 1.0. The dissimilar names (predicate name) and the properties (predicate parameter) of the input model constructs and the training data constructs penalizes the overall fitness function, resulting in the low fitness value.

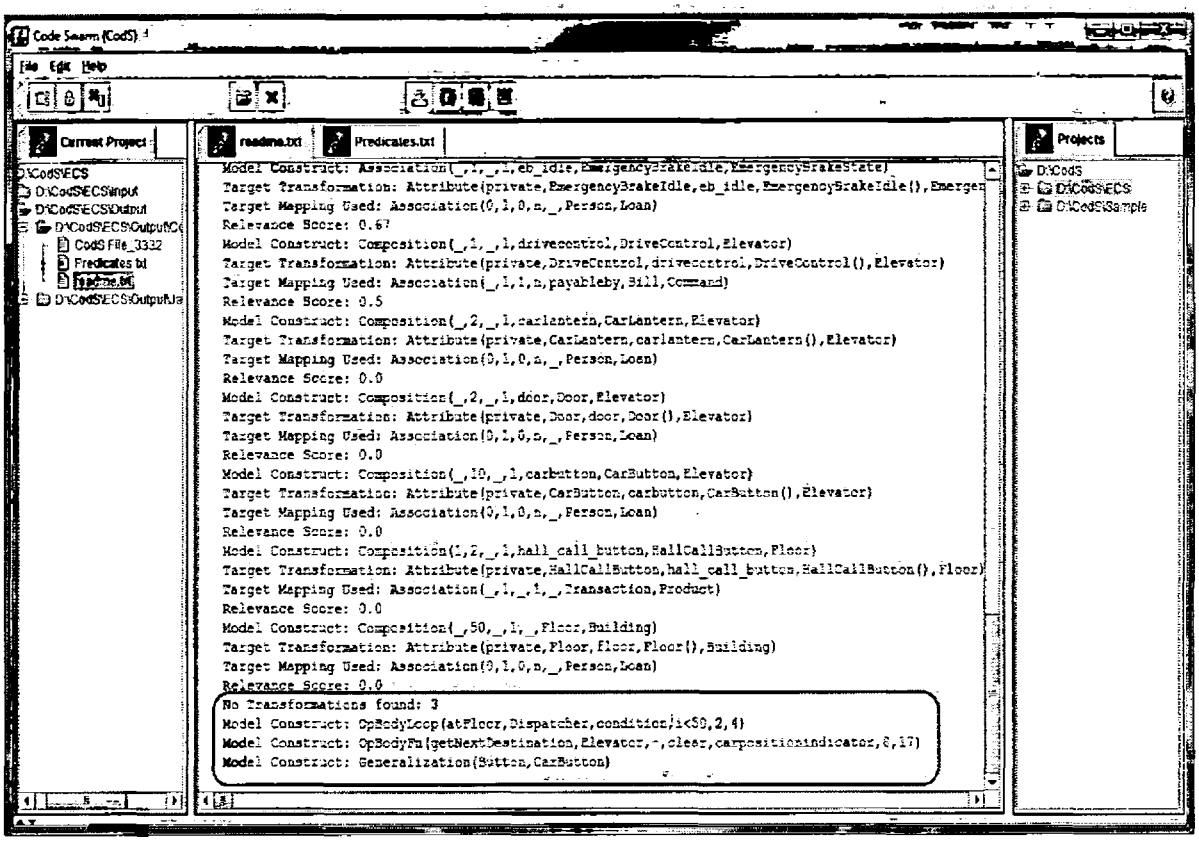


Figure 8.2 CØdŞ screenshot highlighting *missing transformations*

Table 8.2 Execution results for ECS

Input model constructs	Exactly matched constructs	Doubtful transformations		Incorrect/Missing transformations	Best fitness
		27 (3.84%)			
703	673 (95.73%)	*Correct* 22 (81.5%)	*Incorrect* 5 (18.5%)	3 (0.42%)	0.9722

Table 8.3 Post-analysis execution results for ECS

Model	Mapping blocks	Input model constructs	No. of correctly transformed constructs	Best fitness	Correctness (%)	Minimum execution time (sec.)
ECS	60	703	695	0.9722	98.86	165

8.3 10-fold Cross Validation

This section is divided into two sub-sections. The first subsection describes the experimental setting followed by a discussion of the obtained results in the second subsection.

8.3.1 Experimental Setting

The class and state models of 10 different software systems, given in Appendix C, are used for performing the 10-fold cross validation of our proposed approach. For each fold, the class and state models of one software system is transformed by using the remaining 9 software models as the transformations examples. This implies that the 9 software models are used to create the initial swarm to find an optimal transformation solution for the 10^{th} model.

The experimental steps described in Section 8.2.1 are performed to carry out the 10-fold cross validation. The generated code is analyzed with respect to the correctness by using the formula given in Equation 6 (Section 8.2.1). The total number of input source model constructs and the mapping blocks used in each execution are summarized in Table 8.4.

Table 8.4 Number of input model constructs and mapping blocks

Model	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
Constructs	187	106	96	135	145	249	155	202	135	200
Mapping blocks	60	62	63	62	60	57	60	59	53	58

8.3.2 Results and Discussion

Table 8.5 presents the execution results obtained by performing the 10-fold cross validation of our proposed approach. Highest fitness value obtained for models of 3 software systems (M2, M3 and M4) is 1.0 and lowest fitness value found during execution is 0.9758 for M5. Fitness value of 1.0 indicates that the solution searched by our approach is 100% correct. This imphes that models of 3 software systems are perfectly transformed and 100% correct code is generated for the corresponding input model constructs.

However, when we manually analyzed the code in detail, we found that there are 6 software systems for which 100% correct code has been generated, as shown in Table 8.6. It

Table 8.5 Execution results for 10-fold cross validation

Model	Input model constructs	Exactly matched constructs	Doubtful transformations		Incorrect/Missing transformations	Best fitness
			Correct	*Incorrect*		
M1	187	186 (99.47%)	1 (0.53%)		0	0.9973
			1 (100%)	0		
M2	106	106	0		0	1.0
			---	---		
M3	96	96	0		0	1.0
			---	---		
M4	135	135	0		0	1.0
			---	---		
M5	145	140 (96.55%)	3 (2.07%)		2 (1.38%)	0.9758
			3 (100%)	0		
M6	249	246 (98.80%)	3 (1.20%)		0	0.9899
			0	3 (100%)		
M7	155	154 (99.35%)	1 (0.65%)		0	0.9967
			1 (100%)	0		
M8	202	201 (99.50%)	1 (0.5%)		0	0.9975
			1 (100%)	0		
M9	135	134 (99.26%)	0		1 (0.74%)	0.9925
			---	---		
M10	200	197 (98.5%)	2 (1.0%)		1 (0.5%)	0.9899
			---	2 (100%)		

indicates that although no exact match was found in the transformation examples, still some constructs of their models are correctly transformed. This is due to the intelligence of our fitness function which searches for the nearest matching transformation, in case no exact mapping is found in the training data. For example, for M1 the fitness value is 0.9973, still 100% correct code is generated. For one input model construct (one-to-many association) our approach could not find the exact mapping and selects one-to-one association, which is the nearest match of one-to-many association. Since, the final solution searched by the PSO does not contain exact transformation of one construct, it penalizes the fitness value. Figure 8.3 shows a comparison between the correctly matched constructs and the correctly transformed constructs. Out of 7 system models with missing/incorrect transformations, the code correctness of 4 system models

(M1, M5, M7 and M8) is increased by using the correct transformation proposed by our approach.

Table 8.6 Post-analysis execution results for 10-fold cross validation

Model	No. of mapping blocks	Total no. of constructs	No. of correctly transformed constructs	Best fitness	Correctness (%)	Minimum execution time (sec)
M1	60	187	187	0.9973	100	41
M2	62	106	106	1.0	100	25
M3	63	96	96	1.0	100	22
M4	62	135	135	1.0	100	31
M5	60	145	143	0.9758	98.62	33
M6	57	249	246	0.9899	98.79	52
M7	60	155	155	0.9967	100	35
M8	59	202	202	0.9975	100	45
M9	53	135	134	0.9925	99.26	30
M10	58	200	197	0.9899	98.50	42
Average	59.4	160.9	160	0.99396	99.516	35.6

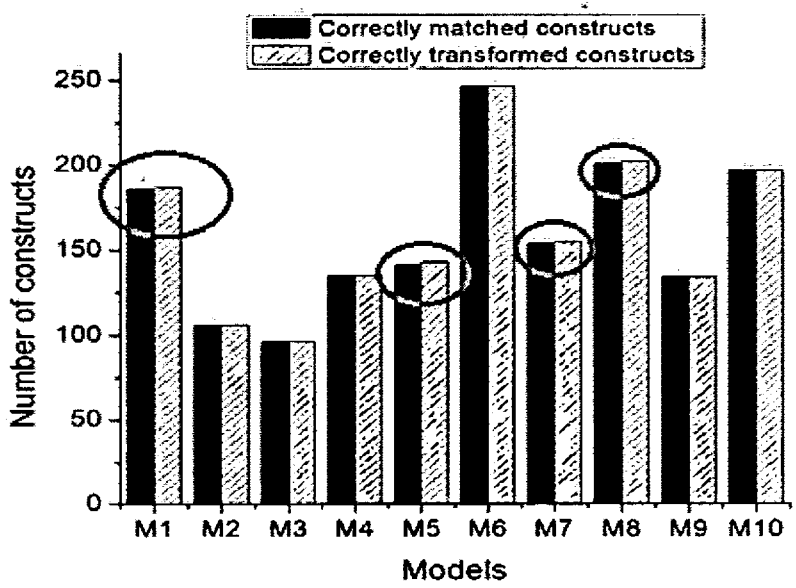


Figure 8.3 Correctly mapped constructs vs. Correctly transformed constructs

The lowest fitness value obtained during execution is 0.9758 for the model M5. This low fitness value can be attributed to the fact that there are 5 constructs in the input source model for which no exact transformation is present in the training data, as we have taken a non-exhaustive set of transformation examples. Still due to the intelligence of our fitness function, there are only 2 constructs for which no or incorrect transformation is found. Again, this increases our confidence on the appropriateness of the selected fitness function.

In most of the cases, the best fitness value is an indicator of the correctness of the code generated i.e. the more the fitness value, the better the percentage of the code correctness. However, in some cases, we can see that although the fitness value is high, the percentage of code correctness is less. For example, we can observe that although the fitness value of M10 is better than M5, still the percentage of code correctness of M5 is high, as shown in Figure 8.4. This is due to the fact that the percentage of code correctness depends upon the number of constructs in the input source model to be transformed, whereas the fitness value is independent of this count.

During the experiment, we observed that our approach always proposes a transformation, even in the absence of an exact construct match in the transformation examples. It is advantageous as this is rather impossible in the existing rule-based code generation approaches, in which the absence of a rule results in a failure to perform the transformation. The transformation rule set needs to be exhaustive and complete to ensure that it proposes a transformation for every construct of the input source model. This implies that we can employ this approach, even if we have a small and non-exhaustive set of transformation examples available. The use of the transformation examples also eliminates the need of understanding transformation languages and complex metamodels. Moreover, besides the existing transformation examples, no other information or expertise are required to perform the experiment.

From the performance point of view, these experiments were performed on a laptop with 1GB RAM and 1.86 GHz processor. As we can see in Figure 8.5, the larger the model, the more the time required to generate the code. However, in our experiments it took less than one minute to generate the corresponding code using the tool based on this approach. Hence, we can say that if system models consist of less than 250 constructs, its corresponding code can be generated in

less than a minute. However, time taken for the execution also depends on the swarm size and number of iterations. In our experiment, we limit the swarm size to 40 and the number of iterations to 20.

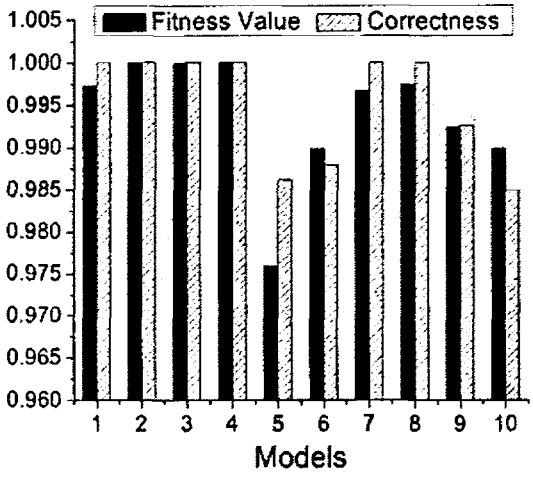


Figure 8.4 Best fitness vs. Code correctness

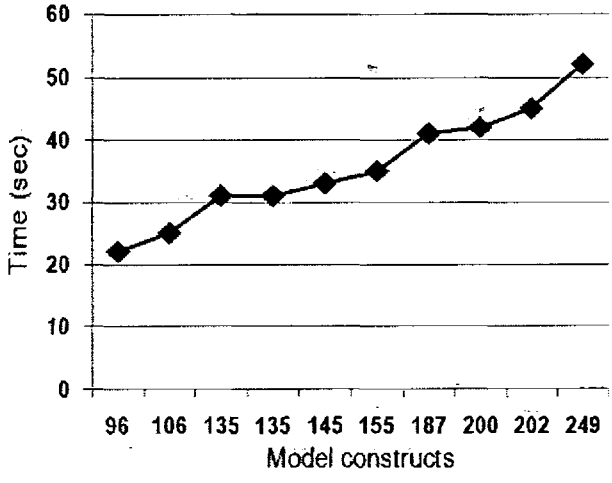


Figure 8.5 Model constructs vs. Execution time

8.4 Comparison

Currently, all the approaches and tools that are in existence are capable of generating correct and consistent code by utilizing their own transformation rules. Our proposed approach and tool also does so. However, the major difference lies in the way this code is generated and the process that is being followed to generate the code. Therefore, instead of focusing solely on the characteristics of the produced output (generated code), we have based our comparison on the entire process of generating code from system models. This section is dedicated to present a description of our comparison.

8.4.1 Code Generation Approaches

The parameters of comparison and their details are given below.

Essence of Approach: All the existing code generation approaches focus on defining a set of transformation rules to perform the transformation process. These approaches identify that which source model construct should be transformed into which target code element in the transformation process. Keeping this in view, it can be said that the existing code generation

approaches basically consist of a set of transformation rules. However, our code generation approach is significantly different from what already exists regarding code generation in current literature. Our approach is a generic approach which can transform any source model into target code without the need of formulating a transformation rule set. Therefore, all complexities associated with the formulation, maintenance and evolution of the transformation rule set are circumvented by this approach.

Foundation of Approach: All existing code generation approaches are based on the metamodels of source modeling and target code languages. However, our approach is independent of the metamodel complexities and does not exploit metamodels as its foundation.

Level of Ease: Since our approach is independent of the complexities of metamodels and transformation rules, it is easy to comprehend and implement.

Degree of Generality: The existing code generation approaches are specific as new transformation rules need to be defined for generating code from different design models. However, our approach is generic and is able to transform any input design model into target code without the need to modify the fitness function.

8.4.2 Code Generation Tools

Among the plethora of tools that support automatic code generation, we have focused on three commercial tools 1) Rhapsody [54], 2) Enterprise Architect [56] and 3) Visual Paradigm [57], and four research-based tools 1) UJECTOR [47, 48], 2) JCode [39], 3) dCode [32] and 4) OCode [1]. A description of the comparison follows.

Behavioral Action Specification: From the available set of automatic code generation tools, only some tools support the transformation of system's dynamic behavior. For different tools, behavioral actions need to be specified using different programming or action languages. For example, in UJECTOR, the actions need to be specified in the UML superstructure. The use of UML superstructure actions raises the level of complexity as it is difficult and time-consuming to specify and understand these actions. On the other hand, CodS relies on a light-weight action language ASL for specifying the behavioral actions, which is simple, readable and easy to learn

and comprehend [27]. Currently there is only one tool that can transform ASL actions into target code language, but it is a rule-based commercial tool.

Explicit Transformation Rules: Existing commercial and research-based tools generate the implementation code by creating a mapping between the source modeling and the target programming languages. All these tools rely on the explicit specification of the transformation rule set. However, our tool does not take a set of transformation rules as input. Rather it is intelligent enough to automatically derive transformation rules from the existing set of transformation examples. Besides the training data, no extra information is needed.

Exhaustive Rule Set: All the existing automatic code generation tools utilize an exhaustive set of transformation rules to correctly generate the target code. These tools will fail to perform the transformation if a rule does not exist for any source model construct. However, our tool is smart enough to assist the user by proposing a nearest transformation, if no exact transformation is found in the training data for the input model construct.

Underlying Approach: All the contemporary commercial and research-based tools are based on the approaches that are model-specific. However, CodS uses a generic approach for automatic code generation i.e. this approach can be used to generate code for any set of source models in any target programming language, provided that the transformation examples exist.

8.5 Assessment

This section is dedicated to precisely present the potential benefits and limitations of our proposed approach.

8.5.1 Benefits

Our proposed approach offers many benefits over the existing code generation approaches. These benefits are given below.

Automatic Extraction of Transformation Rules: Our approach does not rely on the domain experts to manually and explicitly define a set of transformation rules for automatic code

generation. Rather, our aim is just to provide a set of transformation examples and let the system automatically extract transformation rules from them without human intervention.

Irrespective of Metamodels: Our approach is not based on the source and target languages' metamodels, making the M2C transformation process independent of the metamodel complexities.

Independence from Transformation Languages: This approach is capable of transforming models into code without the need to learn and comprehend the complex transformation languages.

Intelligence of Transformation Proposition: The proposed approach always proposes a transformation strategy, nearest match in case if no exact match exists in the training data. This is rather impossible in existing code generation approaches in which the absence of a rule results in a failure to perform the corresponding transformation.

Ease of Transformation Process: This approach makes the automatic code generation process effortless and unproblematic by eradicating the need to learn complex technologies and minimizing human intervention. Besides transformation examples, no extra information is needed and no special expertise is required.

Utilization of Existing Knowledge: Our approach utilizes the existing fragmentary knowledge to perform the automatic code generation process. It uses knowledge from previously solved transformation examples to solve new M2C transformation problems.

Generic Approach: Our proposed approach is a generic approach which is capable of transforming any source model into target code.

Action Specification: This approach is not only capable of generating structural code rather it also provides full support for transforming dynamic actions into the target implementation code. Currently, we have used ASL for specifying the behavioral actions in system models but this approach is not specific to any action language.

Acceleration of Transformation Process: By circumventing the need to manually formulate a transformation rule set and learn complex technologies, the transformation process becomes quick and fast.

8.5.2 Limitations

Availability of Transformation Examples: This approach uses knowledge from previously solved transformation examples to solve new M2C transformation problems. Therefore, the availability of training data is a pre-requisite for the application of this approach.

Increasing Time: Our experimental results revealed that code generation may become time-consuming with the increasing size of models i.e. the larger the models, the more the time required to generate code. However, still it has an acceptable execution time and is many times less than the time required to manually define the transformation rule set and follow the traditional code generation process.

Different Execution Results: As this approach relies on the heuristic search optimization technique, therefore multiple executions for the same input source models may lead to different results. The generation of best solution is not guaranteed in every execution.

Quality of Transformation: As this approach utilizes existing M2C transformation examples to perform the automatic code generation process, therefore the quality of the resulting code is entirely dependent on the correctness of the transformation examples.

Chapter 9

CONCLUSION

9.1 Introduction

This chapter is devoted to present the significant findings from this dissertation. A comprehensive summary of the general conclusions is given in Section 9.2. Finally, Section 9.3 concludes this dissertation by summarizing some future research directions.

9.2 Conclusion

This work can be considered as a contribution to the study of model transformation particularly M2C transformation that have continued to be an area of intense research. For the last few years, a plethora of code generation approaches and tools have been contributed to this field, both by the software researchers and practitioners. Due to these various efforts, the process of automatic code generation has become quite mature and expert intervention is extensively required to carry out the entire code generation process.

Existing code generation approaches rely on the domain experts to manually formulate a transformation rule set, based on the source and target langauges' metamodels and expressed in some model transformation language. In reality, the definition, maintenance and evolution of a complete, consistent, correct and non-redundant transformation rule set is a complex and hard task and many unwanted limitations confine the results. This task is further complicated by the scarcity and paucity of domain knowledge, complexity of metamodels and obscurity of transformation languages.

A comprehensive survey of the existing literature reveals that currently there is only one way to generate code from system models i.e. to manually formulate a transformation rule set. None of the existing approach offers a replacement for manual transformation rule set definition. On the other hand, it can be observed that many organizations keep a memory of their past M2C transformations and feel more comfortable to show these transformation examples instead of defining a complete and consistent transformation rule set. Our work starts from these observations to view automatic code generation as the one to solve with fragmentary knowledge i.e. with only examples of M2C transformations.

In this thesis, we have presented a novel approach for automatic code generation by utilizing the previously solved M2C transformation problems. The available set of transformation examples are used to train the system regarding automatic code generation. After the system is trained, the input models to be transformed are provided. The search space of the transformation examples is explored by using the heuristic search algorithm PSO. The task of PSO is to search for the matching transformation block from the training data corresponding to every construct of the input source model. Every solution searched by PSO is evaluated against an objective function that we have tailored for the M2C transformation problem. The fitness value produced by the fitness function indicates the appropriateness of the transformation block selected by PSO for the transformation of the corresponding input model construct. The transformation solution with the best fitness value is selected as the final optimal solution. The optimal solution searched by PSO is then utilized to transform the input source models into target code. So instead of explicitly providing a transformation rule set as input, our aim is just to provide a set of transformation examples and let the system automatically extract transformation rules from them.

We implemented this approach in a tool named CØde Ṣwarm, abbreviated as CØdṢ. This approach is generic and its application is not limited to any set of models. However as a proof of concept, we validated this approach by generating Java code from class model and state model of software systems, as these two models are representatives of both the static structure and the dynamic system behavior. Our experimental results indicate that up to 100% correct code can be generated by this approach. Moreover, the only prerequisite of this approach is to have a set of previously solved transformation examples. Besides these transformation examples, no extra information is needed to perform the M2C transformation process.

A comprehensive analysis of our experimental results reveals that our fitness function is intelligent enough to look for the nearest match of constructs, in case if no exact match is found in the transformation examples. This implies that we can easily start with a small and non-exhaustive set of transformation examples. This is rather impossible in existing rule-based approaches where an exhaustive set of transformation rules is required to ensure the correct and complete M2C transformation. Our evaluation shows that this approach is not only effective in generating code for small sized models, rather it is also capable of generating quality code for

models containing hundreds of constructs. Furthermore, no special expertise is required for the application of this approach. This approach makes the M2C transformation process painless by dissociating it from transformation rule set definition, transformation languages and source and target languages' metamodels.

However, this approach also has some limitations. The availability of the transformation examples is a prerequisite of this approach. As the size of input source models increases, the task of generating code may become time-consuming. Since we use heuristic search for finding an optimal solution, the generation of best solution is not guaranteed in every execution.

9.3 Future Work

In this section, we present some guidelines for the potential future work that would be interesting to investigate further.

9.3.1 Improve Code Correctness

Although the use of PSO as heuristic search technique yields good results, the use of other evolutionary algorithms to further improve code correctness can be interesting.

9.3.2 Reduce Execution Time

Although the target code is generated in an acceptable execution time at the moment, this time should be further reduced in order to speed-up the automatic code generation process. One of its possible solutions is to improve the efficiency of PSO. This can be achieved either by using a guided search for PSO or improve the stopping criterion of the heuristic search.

9.3.3 Application to Large-Scale Models

We have validated this approach by applying it to small and medium sized system design models. However, in future the effectiveness of this approach should be investigated by the application of this approach to large-scale structural and dynamic software design models.

9.3.4 Application to Multiple System Design Models

Currently, this approach has been applied to generate code from class and state models of the software system. In future, this approach should be validated by utilizing it to generate code for other system design models e.g. UML sequence diagram, UML activity diagram etc.

9.3.5 Automate Expression of Predicates

The task of representing transformation examples and input source model constructs as predicates is performed manually at the moment. An algorithm should be designed and implemented to perform this task automatically.

9.3.6 Automation of Transformation Examples Representation

Currently, the development of mapping blocks from the available transformation examples is a human-dependent activity. Automation of this task can further facilitate the code generation process.

9.3.7 Enhance CØdŞ Tool

Presently, CØdŞ is capable of generating Java code from UML class and state models, with the behavioral actions specified in ASL. This tool should be enhanced to generate code in multiple programming languages from various system design models. Moreover, the support for other action languages should also be incorporated in CØdŞ.

REFERENCES

[1] J. Ali and J. Tanaka, "*An Object-Oriented Approach to Generate Executable Code from OMT-based Dynamic Model*" Journal of Integrated Design and Process Design, Vol. 2, No. 4, pp. 65-77, 1998.

[2] B. Pierre, R. Dupuis, A. Abran, J. Moore, and L. Tripp, "*The Guide to the Software Engineering Body of Knowledge*" IEEE Software, Vol. 16, Issue 6, ISSN: 0740-7459, pp. 35-44, 1999.

[3] U. Behrens, M. Flasinski, L. Hagge, J. Jurek, and K. Ohrenberg, "*Recent Developments of the ZEUS Expert System ZEX*" IEEE Transactions on Nuclear Science, Vol. 43, Issue 1, ISSN: 0018-9499, p. 65, 1996.

[4] M. Kessentini, H. Sahraoui, and M. Boukadoum, "*Model Transformation as an Optimization Problem*" Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, Vol. 5301, pp. 159-173, Springer Berlin / Heidelberg, 2008.

[5] D. Varro, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, "*Model Transformation by Example*" Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, Vol. 4199, pp. 410-424, Springer Berlin / Heidelberg, 2006.

[6] E. Alexander, "*Automated Abstraction of Class Diagrams*" Journal of ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 11, Issue 4, pp. 449-491, New York USA, 2002.

[7] M. Kessentini, H. Sahraoui, M. Boukadoum, and O. Omar, "*Search-based Model Transformation by Example*" Software and Systems Modeling, pp. 1-18, 2010

[8] K. Stuart, "*Model Driven Engineering*" in Proc. of the 3rd International Conference on Integrated Formal Methods (IFM), Springer-Verlag, London UK, 2002.

[9] J. Bezivin, R. Limmel, J. o. Saraiva, and J. Visser, "*Model Driven Engineering: An Emerging Technical Space*" Generative and Transformational Techniques in Software Engineering, Lecture Notes in Computer Science, Vol. 4143, pp. 36-64, Springer Berlin / Heidelberg, 2006.

[10] A. Kleppe and J. B. Warmer, W., "*MDA Explained: The Practice and Promise of Model Driven Architecture*" Addison-Wesley, 2003.

[11] J. Miller and J. Mukerji, "*MDA Guide*" Version 1.0.1, 2003.

[12] H. Liu, "*A Template-Based Model Transformation Approach using a Simplified Hierarchical Metamodel*" PhD. Dissertation, College of Computing and Digital Media, DePaul University, 2010.

[13] S. Ed, "*What Models Mean*" IEEE Software, Vol. 20, Issue 5, ISSN: 0740-7459, pp. 26-32, 2003.

[14] "*OMG Unified Modeling Language Specification*" Version 1.3.1, 1st Edition, 2000.

[15] "*OMG Unified Modeling Language (OMG UML) Infrastructure*" Vol. 2.3, Doc. no. formal/2010-05-03, May 2010.

[16] D. D. Ruscio, "*Specification of Model Transformation and Weaving in Model Driven Engineering*" PhD. Thesis, Universita di L'Aquilla, 2007.

[17] R. Runde and S. K., "*What is Model Driven Architecture?*" Research Report 304, University of Oslo, 2003.

[18] S. Sendall and W. Kozaczynski, "*Model Transformation: The Heart and Soul of Model-Driven Software Development*" IEEE Software, Vol. 20, Issue 5, ISSN: 0740-7459, pp. 42-45, 2003.

[19] K. Czarnecki and S. Helsen, "*Classification of Model Transformation Approaches*" in 2nd OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture, Oct. 2003.

[20] T. Mens and P. Gorp, "*A Taxonomy of Model Transformation*" in Proc. of the International Workshop on Graph and Model Transformation (GraMoT), Vol. 152, pp. 125-142, March 2006

[21] M. Piefel, "*A Common Metamodel for Code Generation*" in Proc. of the 3rd International Conference on Cybernetics and Information Technologies, Systems and Applications (IIIS), 2006.

[22] S. Mellor, K. Scott, A. Uhl, and D. Weise, "*MDA Distilled: Principles of Model Driven Architecture*" Addison-Wesley, 2004.

[23] I. Niaz and J. Tanaka, "*Code Generation from UML Statecharts*" in 7th International Conference on Software Engineering and Applications (SEA), pp. 315-321, 2003.

[24] J. Kennedy and R. Eberhart, "*Particle Swarm Optimization*" in Proc. of the IEEE International Conference on Neural Networks, Vol. 4, pp. 1942-1948, Perth Australia, 1995.

[25] R. Poli and J. Kennedy, "*Particle Swarm Optimization: An Overview*" Swarm Intelligence, Vol. 1, pp. 33-57, 2007.

[26] J. Ke, Z. Lei, and S. Miyake, "*An Executable UML with OCL-based Action Semantics Language*" in 14th Asia-Pacific Software Engineering Conference (APSEC), ISSN: 1530-1362, pp. 302-309, Dec. 2007.

[27] K. Carter, "*UML ASL Reference Guide for ASL Language Level 2.5*" Manual Version D, 2003.

[28] A. W. Brown, S. Iyengar, and S. Johnston, "*A Rational Approach to Model-Driven Development*" IBM Systems Journal, Vol. 45, Issue 3, ISSN:0018-8670, pp. 463-480, 2006.

[29] L. Favre, L. Martinez, and C. Pereira, "*Transforming UML Static Models into Object-Oriented Code*" in Proc. of the 37th International Conference on Technology of Object-Oriented Languages and Systems TOOLS-Pacific, pp. 170-181, 2000.

[30] M. Nassar, A. Anwar, S. Ebersold, B. Elasri, B. Coulette, and A. Kriouile, "*Code Generation in VUML Profile: A Model Driven Approach*" IEEE/ACS International Conference on Computer Systems and Applications (AICCSA), pp. 412-419, May 2009.

[31] R. van de Weg, R. Engmann, R. van de Hoef, and V. ten Thij, "*An Environment for Object-Oriented Real-Time Systems Design*" in 8th Conference on Software Engineering Environments, pp. 23-33, Apr. 1997.

[32] J. Ali and J. Tanaka, "*Implementing the Dynamic Behavior Represented as Multiple State Diagrams and Activity Diagrams*" ACIS International Journal of Computer and Information Science, Vol. 2, Issue 1, Mar. 2001.

[33] A. Knapp and S. Merz, "*Model Checking and Code Generation for UML State Machines and Collaborations*" in Proc. of the 5th Workshop on Tools for System Design and Verification, Technical Report, Vol. 11, pp. 59-64, 2002.

[34] I. Niaz and J. Tanaka, "*Mapping UML Statecharts to Java Code*" in Proc. IASTED International Conference on Software Engineering, pp. 111-116, 2004.

[35] G. Pinter and I. Majzik, "*Automatic Code Generation Based on Formally Analyzed UML Statechart Models*" in Proc. of the Workshop on Formal Methods for Railway Operation and Control Systems, pp. 45-52, 2003.

[36] S. Neeraj, F. Edward, L. Karl, and L. David, "*Interaction Schemata: Compiling Interactions to Code*" in Proc. of the Technology of Object-Oriented Languages and Systems (TOOLS) 30, pp. 268-277, Aug. 1999.

[37] G. Engels, R. H¼cking, S. Sauer, A. Wagner, R. France, and B. Rumpe, "*UML Collaboration Diagrams and Their Transformation to Java*" in UML'99 The Unified Modeling Language, Lecture Notes in Computer Science, Vol. 1723, Springer Berlin / Heidelberg, 1999.

[38] I. Niaz, "*Automatic Code Generation from UML Class and Statechart Diagrams*" PhD. Dissertation, University of Tsukuba, Japan, 2005.

[39] I. Niaz and J. Tanaka, "*An Object-Oriented Approach to Generate Java Code from UML Statecharts*" International Journal of Computer and Information Science, Vol. 6, No. 2, June 2005.

[40] P. A. Noe and T. C. Hartrum, "*Extending the Notation of Rational Rose 98 for use with Formal Methods*" in Proc. of the IEEE National Aerospace and Electronics Conference NAECON, pp. 43-50, 2000.

[41] A. Derezinska and R. Pilitowski, "*Correctness Issues of UML Class and State Machine Models in the C# Code Generation and Execution Framework*" International Multiconference on Computer Science and Information Technology (IMCSIT), pp. 517-524, Oct. 2008.

[42] A. Derezinska and R. Pilitowski, "*Realization of UML Class and State Machine Models in the C# Code Generation and Execution Framework*" Informatica, Vol. 33, pp. 431-440, 2009.

[43] M. Thongmak and P. Muenchaisri, "*Design of Rules for transforming UML Sequence Diagrams into Java Code*" in 9th Asia-Pacific Software Engineering Conference, ISSN: 1530-1362, pp. 485-494, 2002.

[44] L. Quan, L. Zhiming, L. Xiaoshan, and J. He, "*Consistent Code Generation from UML Models*" in Proc. of the Software Engineering Conference, ISSN: 1530-0803, pp. 23-30, Apr. 2005.

[45] N. Ulrich, J. Niere, and A. Zundorf, "*The FUJABA Environment*" in Proc. of the 22nd International Conference on Software Engineering, pp. 742-745, Limerick, Ireland, June 2000.

[46] D. Bjorklund, J. Lilius, and I. Porres, "*A Unified Approach to Code Generation from Behavioral Diagrams*" Languages for System Specification, pp. 20-34, 2004.

[47] M. Usman and A. Nadeem, "*Automatic Generation for Java Code from UML Diagrams using UJECTOR*" International Journal of Software Engineering and its Applications, Vol. 3, No. 2, Apr. 2009.

[48] M. Usman, A. Nadeem, and K. Tai-hoon, "*UJECTOR: A Tool for Executable Code Generation from UML Models*" in Advanced Software Engineering and Its Applications ASEA, pp. 165-170, Dec. 2008.

[49] A. Jakimi and M. Elkoutbi, "*An Object-Oriented Approach to UML Scenarios Engineering and Code Generation*" International Journal of Computer Theory and Engineering (IJCTE), Vol. 1, No. 1, pp. 35-41, Apr. 2009.

[50] A. Jakimi and M. Elkoutbi, "*Automatic Code Generation from UML Statechart*" International Journal of Engineering and Technology, Vol. 1, No. 2, pp. 165-168, June 2009.

[51] D. C. and S. T., "*An Automatic Approach to Transform CafeOBJ Specifications to Java Template Code*" in Proceedings of the International Conference on Software Engineering Research and Practice (SERP), pp. 171-176, 2003.

[52] C. Doungsa-ard and T. Suwannasart, "*A Semantic Part Generated Java Statement from a CafeOBJ Specification*" in IEEE International Conference on Electro/information Technology, pp. 388-393, May 2006.

[53] "*IBM® Rational® Rose® Enterprise*", http://www-01.ibm.com/software/awdtools/developer/rose/enterprise/.

[54] "*IBM® Rational® Rhapsody®*", http://www-01.ibm.com/software/rational/products/rhapsody/swarchitect/.

[55] N. Sangal and K. Lieberherr, "*StructureBuilder Tendeil Software Inc.*" in OOPSLA, 1998.

[56] "*Sparx Systems - Enterprise Architect*" http://www.sparxsystems.com/.

[57] "*Visual Paradigm (VP-UML)*" http://www.visual-paradigm.com/product/vpuml/.

[58] "*AndroMDA*" http://www.andromda.org/docs/index.html, 2011.

[59] "*MagicDraw*" https://www.magicdraw.com/, 2011.

[60] "*Papyrus UML, Open Source Tool for Graphical UML2 Modeling*" http://www.papyrusuml.org, http://www.eclipse.org/modeling/mdt/papyrus/.

[61] K. Carter, "*Abstract Solutions - iUML*" http://www.kc.com.

[62] "*AgroUML, Open Source Software Engineering Tool*" http://argouml.tigris.org/.

[63] P. Gorp, "*Model-Driven Development of Model Transformations*" PhD. Thesis, University of Antwerp, 2008.

[64] R. Akerkar and P. Sajja, "*Knowledge-Based Systems*" 2010.

[65] A. Agnar and P. Enric, "*Case-based reasoning: foundational issues, methodological variations, and system approaches*" Journal of AI Communications, Vol. 7, No. 1, pp. 39-59, IOS Press Amsterdam, Netherlands, Mar. 1994.

[66] W. Elloumi, N. Rokbani, and A. M. Alimi, "*Ant Supervised by PSO*" in 4th International Symposium on Computational Intelligence and Intelligent Informatics, (ISCIII), pp. 161-166, Oct. 2009.

[67] Z. LanLan, W. Ling, W. Xiuting, and H. Ziyuan, "*A Novel PSO-Inspired Probability-based Binary Optimization Algorithm*" in 8th International Symposium on Information Science and Engineering (ISISE), Vol. 2, pp. 248-251, Dec. 2008.

[68] B. Soudan and M. Saad, "*An Evolutionary Dynamic Population Size PSO Implementation*" in 3rd International Conference on Information and Communication Technologies: From Theory to Applications, pp. 1-5, Apr. 2008.

[69] P. N. Suganthan, "*Particle Swarm Optimiser with Neighbourhood* Operator" in Proceedings of the 1999 Conference on Evolutionary Computation (CEC), Vol. 3, 1999.

[70] A. Ratnaweera, S. K. Halgamuge, and H. C. Watson, "*Self-Organizing Hierarchical Particle Swarm Optimizer with Time-Varying Acceleration Coefficients*" IEEE Transactions on Evolutionary Computation, Vol. 8, Issue 3, ISSN: 1089-778X, pp. 240-255, June 2004.

[71] Eclipse, "*Eclipse IDE for Java Developers*" http://www.eclipse.org/downloads/moreinfo/java.php.

Appendix A

PREDICATE STRUCTURE TEMPLATES

A.1 Class Model Templates

This section presents the templates of predicates structure of class diagram constructs defined and used in this research project.

A.1.1 Class Construct

The construct of 'Class' along with its attributes and operations is treated as a single construct.

Source Model Construct

```
Class(<name>)
Attribute(<name>,<datatype>,<initialvalue>,<classname>,<unique/notunique>)
Operation(<name>,<classname>,<returntype>)
OperationParam(<paramname>,<paramtype>,<operationname>,<classname>,
<paramnumber>)
```

Target Code Construct

```
Class(public,<classname>)
Attribute(private,<datatype>,<name>,<initialvalue>,<classname>)
Method(public,<returntype>,<name>,<classname>)
MethodParam(<paramtype>,<paramname>,<methodname>,<classname>,<paramnumber>)
```

Mapping

```
Class(<name>):Class(public,<classname>)

Attribute(<name>,<datatype>,<initialvalue>,<classname>,
<unique/notunique>):Attribute(private,<datatype>,<name>,<initialvalue>,
<classname>)

Operation(<name>,<classname>,<returntype>):Method(public,<returntype>,<name>,
<classname>)

OperationParam(<paramname>,<paramtype>,<operationname>,<classname>,<paramnumb
er>):MethodParam(<paramtype>,<paramname>,<methodname>,<classname>,
<paramnumber>)
```

Key Parameters

None

A.1.2 Generalization Construct

The generalization relationship of class diagram is transformed into an inheritance relationship in OO programming languages.

Source Model Construct

```
Generalization(<parentclassname>;<childclassname>)
```

Target Code Construct

```
Class(public,<childclassname>,extends,<parentclassname>)
```

Mapping

```
Generalization(<parentclassname>,<childclassname>):Class(public,
<childclassname>,extends,<parentclassname>)
```

Key Parameters

None

A.1.3 Association Construct

Associations of class model are transformed to the corresponding class attributes.

Source Model Construct

```
Association(<multiplicity>,<multiplicity>,<multiplicity>,<multiplicity>,
<relationshipname>,<classname>,<classname>)
```

Target Code Construct

```
Attribute(private,<objecttype>,<objectname>,<objecttype>,
<containingclassname>)
```

Mapping

```
Association(<multiplicity>,<multiplicity>,<multiplicity>,<multiplicity>,
<relationshipname>,<classname>,<classname>):Attribute(private,<objecttype>,
<objectname>,<objecttype>,<containingclassname>)
```

Key Parameters

1st and 3rd parameters

A.1.4 Composition Construct

Composition construct in class model is transformed to the corresponding class attribute.

Source Model Construct

```
Composition(<multiplicity>,<multiplicity>,<multiplicity>,<multiplicity>,
<relationshipname>,<classname>,<classname>)
```

Target Code Construct

```
Attribute(private,<objecttype>,<objectname>,<objectcount>,
<containingclassname>)
```

Mapping

```
Composition(<multiplicity>,<multiplicity>,<multiplicity>,<multiplicity>,
<relationshipname>,<classname>,<classname>):Attribute(private,<objecttype>,
<objectname>,<objectcount>,
<containingclassname>)
```

Key Parameters

1st parameter

A.1.5 Aggregation Construct

Aggregation construct in class model is transformed to the corresponding class attribute.

Source Model Construct

```
Aggregation(<multiplicity>,<multiplicity>,<multiplicity>,<multiplicity>,
<relationshipname>,<classname>,<classname>)
```

Target Code Construct

```
Attribute(private,<objecttype>,<objectname>,<objectcount>,
<containingclassname>)
```

Mapping

```
Aggregation(<multiplicity>,<multiplicity>,<multiplicity>,<multiplicity>,
<relationshipname>,<classname>,<classname>):Attribute(private,<objecttype>,
<objectname>,<objectcount>,<containingclassname>)
```

Key Parameters

1st parameter

A.2 State Model Templates

The state model constructs are represented as predicates using the following templates.

A.2.1 State Construct

State along with its entry, do and exit activities is treated as a single construct.

Source Model Construct

```
State(<statename>,<classname>)
Operation(Entry,<statename>,<returntype>)
OperationParam(<paramname>,<paramtype>,Entry,<statename>,<paramnumber>)
Operation(Exit,<statename>,<returntype>)
OperationParam(<paramname>,<paramtype>,Exit,<statename>,<paramnumber>)
Operation(doActivity,<statename>,<returntype>)
OperationParam(<paramname>,<paramtype>,doActivity,<statename>,<paramnumber>)
```

Target Code Construct

```
Class(public,<statename>,<classname>)
Method(public,<returntype>,Entry,<statename>)
MethodParam(<paramtype>,<paramname>,Entry,<statename>,<paramnumber>)
Method(public,<returntype>,Exit,<statename>)
MethodParam(<paramtype>,<paramname>,Exit,<statename>,<paramnumber>)
Method(public,<returntype>,doActivity,<statename>)
MethodParam(<paramtype>,<paramname>,doActivity,<statename>,<paramnumber>)
```

Mapping

```
State(<statename>,<classname>):Class(public,<statename>,<classname>)

Operation(Entry,<statename>,<returntype>):Method(public,<returntype>,Entry,
<statename>)

OperationParam(<paramname>,<paramtype>,Entry,<statename>,<paramnumber>):
MethodParam(<paramtype>,<paramname>,Entry,<statename>,<paramnumber>)
Operation(Exit,<statename>,<returntype>):Method(public,<returntype>,Exit,
<statename>)

OperationParam(<paramname>,<paramtype>,Exit,<statename>,<paramnumber>):
MethodParam(<paramtype>,<paramname>,Exit,<statename>,<paramnumber>)
```

```
Operation(doActivity,<statename>,<returntype>):Method(public,<returntype>,
doActivity,<statename>)

OperationParam(<paramname>,<paramtype>,doActivity,<statename>,<paramnumber>):
MethodParam(<paramtype>,<paramname>,doActivity,<statename>,<paramnumber>)
```

Key Parameters

None

A.2.2 Transition Construct

Transitions are transformed to the methods in the corresponding state.

Source Model Construct

```
Transition(<sourcestate>,<transitionname>,<targetstate>)

TransitionParam(<transitionparamname>,<transitionparamtype>,<transitionname>,
<sourcestate>,<transitionparamnumber>)
```

Target Model Construct

```
Method(public,void,<transitionname>,<sourcestate>)

MethodParam(<transitionparamtype>,<transitionparamname>,<transitionname>,
<sourcestate>,<transitionparamnumber>)
```

Mapping

```
Transition(<sourcestate>,<transitionname>,<targetstate>):Method(public,void,
<transitionname>,<sourcestate>)

TransitionParam(<transitionparamname>,<transitionparamtype>,<transitionname>,
<sourcestate>,<transitionparamnumber>):MethodParam(<transitionparamtype>,
<transitionparamname>,<transitionname>,<sourcestate>,<transitionparamnumber>)
```

Key Parameters

None

A.3 Action Specification

Generating code only in terms of class and method declarations is not enough. Program logic is the most significant part of software systems for execution. Our proposed approach

cannot only be used to represent skeletons of model and code, rather the behavioral actions inside the models can also be easily expressed as predicates. In the context of class model, dynamic actions are used to specify operation body. Within a state model, entry, do and exit activities and transition guards are specified to represent the dynamic system behavior.

A.3.1 Variable Declaration

The predicate structure template for declaring a variable is given below.

Source Model Construct

```
OpBodyDeclaration(<methodname>,<classname>,<datatype>,<variablename>,
<initialvalue>,<iterationnumber>,<statementnumber>)
```

Target Code Construct

```
MethodBodyDeclaration(<methodname>,<classname>,<datatype>,<variablename>,
<initialvalue>,<iterationnumber>,<statementnumber>)
```

Mapping

```
OpBodyDeclaration(<methodname>,<classname>,<datatype>,<variablename>,
<initialvalue>,<iterationnumber>,<statementnumber>):MethodBodyDeclaration
(<methodname>,<classname>,<datatype>,<variablename>,<initialvalue>,
<iterationnumber>,<statementnumber>)
```

Key Parameters

None

A.3.2 Assignment Statement

Assignment statements have the following structural template.

Source Model Construct

```
OpBodyAssign(<methodname>,<classname>,<assignment>,<iterationnumber>,
<statementnumber>)
```

Target Code Construct

```
MethodBodyAssign(<methodname>,<classname>,<assignment>,
<iterationnumber>,<statementnumber>)
```

Mapping

```
OpBodyAssign(<methodname>,<classname>,<assignment>,<iterationnumber>,
<statementnumber>):MethodBodyAssign(<methodname>,<classname>,<assignment>,
<iterationnumber>,<statementnumber>)
```

Key Parameters

None

A.3.3 Return Statement

Return statements are expressed using the following predicate template.

Source Model Construct

```
OpBodyReturn(<methodname>,<classname>,Output,<variablename>,
<iterationnumber>,<statementnumber>)
```

Target Code Construct

```
MethodBodyReturn(<methodname>,<classname>,return,<variablename>,
<iterationnumber>,<statementnumber>)
```

Mapping

```
OpBodyReturn(<methodname>,<classname>,Output,<variablename>,<iterationnumber>
,<statementnumber>):MethodBodyReturn(<methodname>,<classname>,return,
<variablename>,<iterationnumber>,<statementnumber>)
```

Key Parameters

None

A.3.4 Iteration/Loop

Loops have the following predicate template.

Source Model Construct

```
OpBodyLoop(<methodname>,<classname>,condition,<conditionstatement>,
<iterationnumber>,<statementnumber>)
```

Target Code Construct

```
MethodBodyLoop(<methodname>,<classname>,for,int,<loopinitialization>,
```

```
<conditionstatement>,<loopincrement/decrement>,<iterationnumber>,
<statementnumber>)
```

Mapping

```
OpBodyLoop(<methodname>,<classname>,condition,<conditionstatement>,
<iterationnumber>,<statementnumber>):MethodBodyLoop(<methodname>,<classname>,
for,int,<loopinitialization>,<conditionstatement>,<loopincrement/decrement>,
<iterationnumber>,<statementnumber>)
```

Key Parameters

None

A.3.5 Decision Statement

We defined the following template for decision statements' predicates.

Source Model Construct

```
OpBodyCondition(<methodname>,<classname>,if,<conditionvariable>,
<conditionsymbol>,<conditionvalue>,<iterationnumber>,<statementnumber>)
```

Target Code Construct

```
MethodBodyCondition(<methodname>,<classname>,if,<conditionvariable>,
<conditionsymbol>,<conditionvalue>,<iterationnumber>,<statementnumber>)
```

Mapping

```
OpBodyCondition(<methodname>,<classname>,if,<conditionvariable>,
<conditionsymbol>,<conditionvalue>,<iterationnumber>,<statementnumber>):
MethodBodyCondition(<methodname>,<classname>,if,<conditionvariable>,
<conditionsymbol>,<conditionvalue>,<iterationnumber>,<statementnumber>)
```

Key Parameters

None

A.3.6 Function Call

The function call along with its parameters is treated as a single construct.

Source Model Construct

```
OpBodyFn(<callermethodname>,<callerclassname>,<storagevariable>,
<calledmethodname>,<calledclass/object>,<iterationnumber>,<statementnumber>)
OpBodyFnParam(<callermethodname>,<callerclassname>,<calledmethodname>,
<paramname>,<statementnumber>,<paramnumber>)
```

Target Code Construct

```
MethodBodyFn(<callermethodname>,<calledclassname>,<storagevariable>,
<calledclass/object>,<calledmethodname>,<iterationnumber>,<statementnumber>)
MethodBodyFnParam(<callermethodname>,<callerclassname>,<calledmethodname>,
<paramname>,<statementnumber>,<paramnumber>)
```

Mapping

```
OpBodyFn(<callermethodname>,<callerclassname>,<storagevariable>,
<calledmethodname>,<calledclass/object>,<iterationnumber>,<statementnumber>):
MethodBodyFn(<callermethodname>,<calledclassname>,<storagevariable>,
<calledclass/object>,<calledmethodname>,<iterationnumber>,<statementnumber>)

OpBodyFnParam(<callermethodname>,<callerclassname>,<calledmethodname>,
<paramname>,<statementnumber>,<paramnumber>):MethodBodyFnParam(
<callermethodname>,<callerclassname>,<calledmethodname>,<paramname>,
<statementnumber>,<paramnumber>)
```

Key Parameters

None

Appendix B

USER MANUAL

B.1 Main Interface of CØdŞ

The main interface of CØdŞ can be divided into five sections, as show in Figure B.1.

1. Left Panel
2. Right Panel
3. Center Panel
4. Console
5. Menu Bar and Toolbar

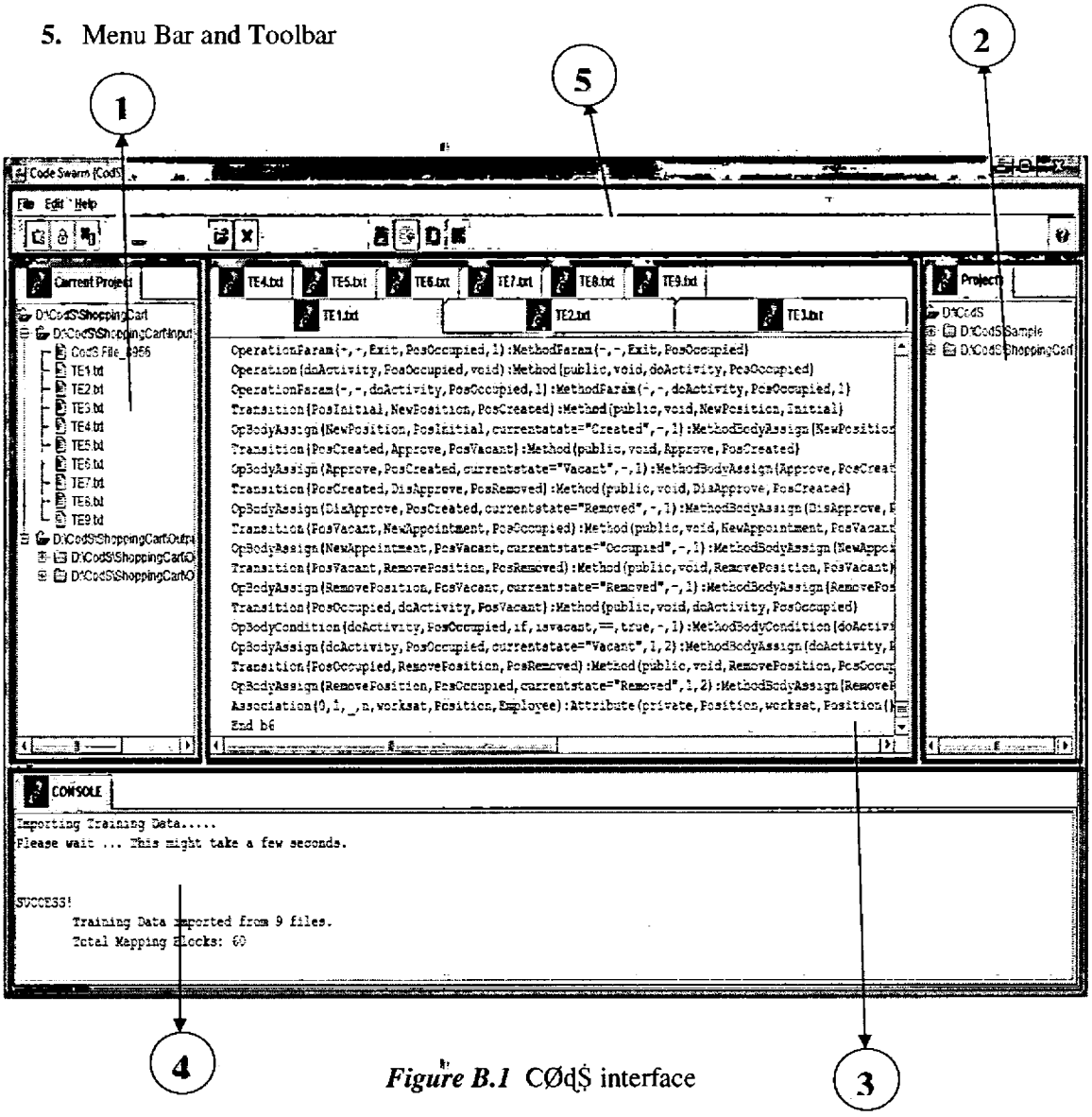


Figure B.1 CØdŞ interface

B.1.1 Left Panel

The left panel of the CØdŞ editor displays the currently opened project. At a time, only one project can be opened for working.

B.1.2 Right Panel

The right panel of the interface displays a list of all existing CØdŞ projects. These projects are present in the "D:\CodS" folder.

B.1.3 Center Panel

The center panel of the editor is used for displaying text files. Multiple files can be opened.

B.1.4 Console

The bottom panel of the editor is used as a console for displaying informative and error messages to the end-user.

B.1.5 Menu Bar and Toolbar

The top of CØdŞ interface contains a menu bar and a toolbar. All options of the menu bar are also available in the toolbar for ease of use and quick selection.

B.2 File Menu

The File menu consists of five options, as shown in Figure B.2.

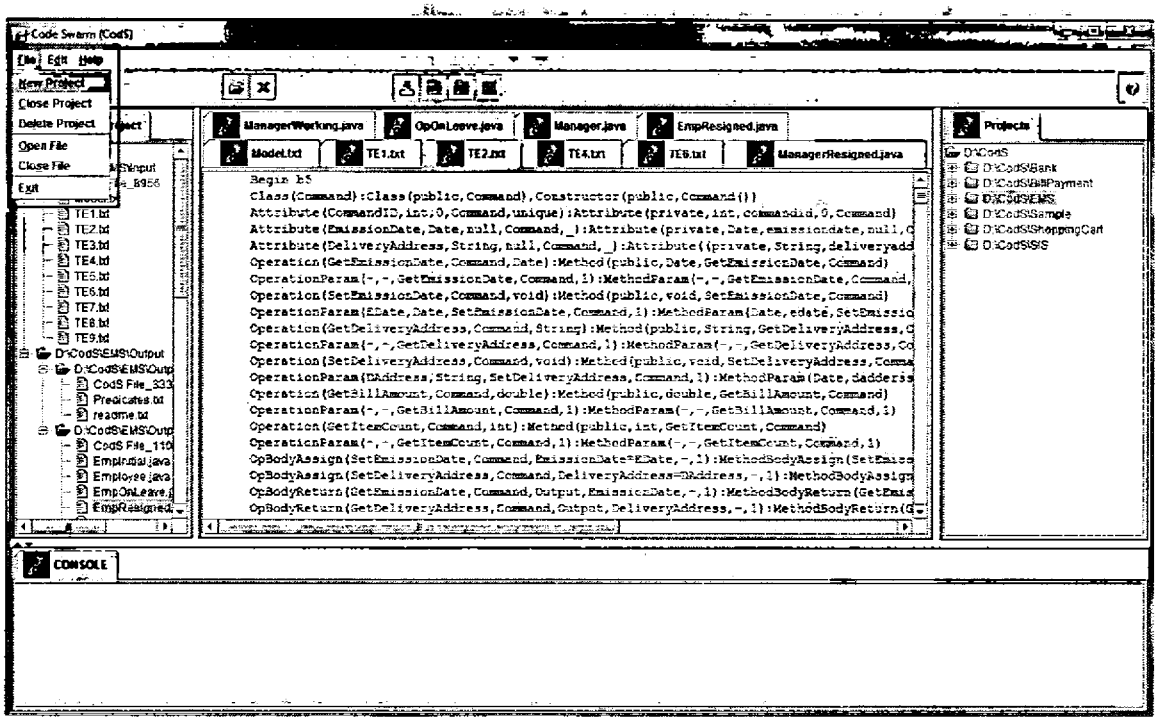


Figure B.2 File menu

B.2.1. New Project

This option enables the user to create new project. This new project is created in D:\CodS folder and is displayed in the left panel. Project name is provided by the user, as shown in Figure B.3. New project is created with the following structure:

D:\ CodS\<Project Name> \ Input

D:\ CodS\ <Project Name> \ Output \ Predicates

D:\ CodS \ <Project Name> \ Output \ Java Code

If another project is already opened (i.e. displayed in the left panel), user will be asked to close the currently opened project, as only one project can be opened at a time. This scenario is shown in Figure B.4.

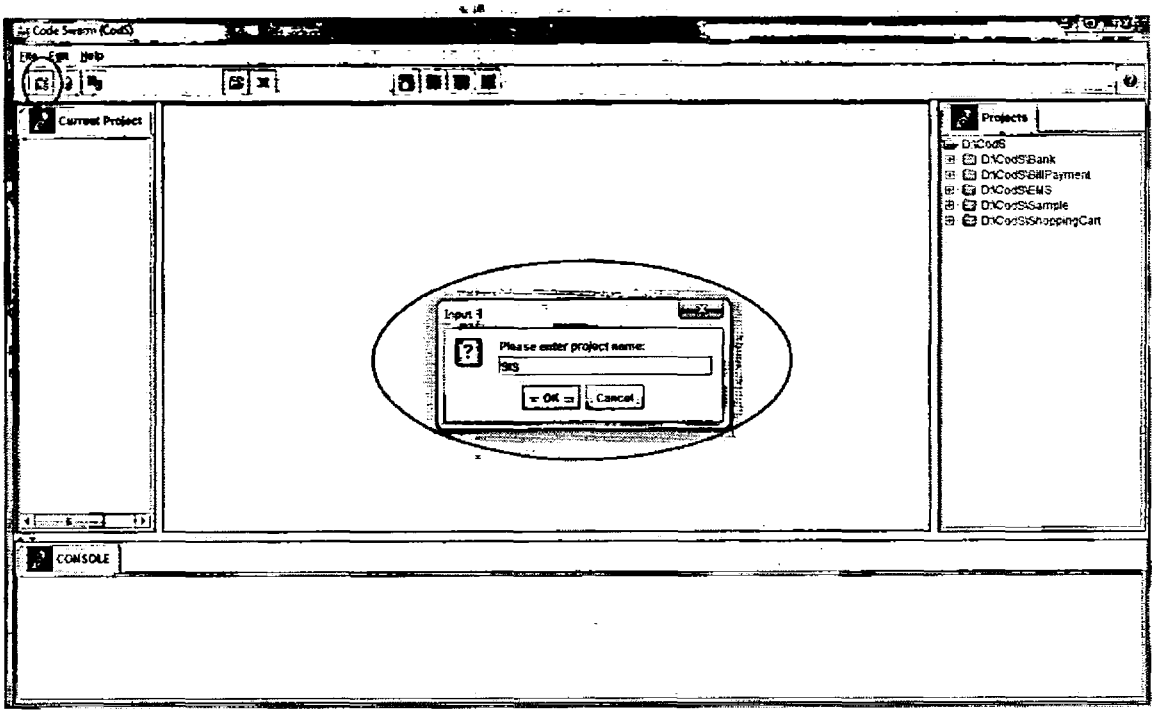


Figure B.3 New project

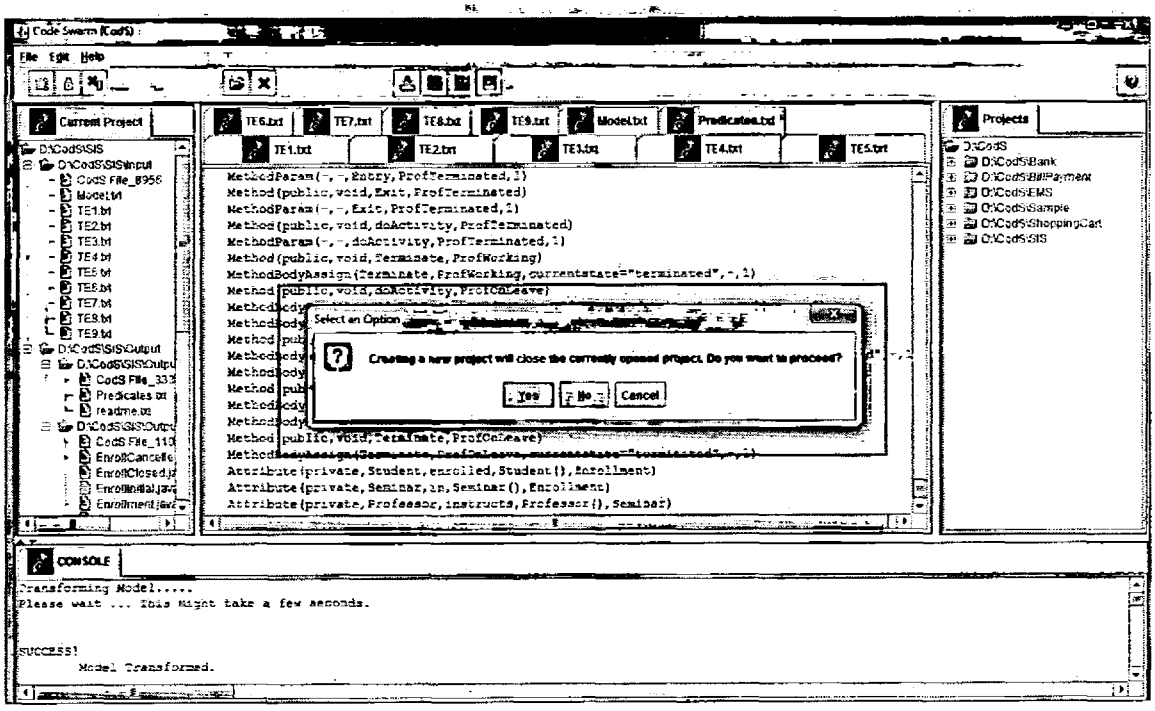


Figure B.4 New project confirmation screen

B.2.2. Close Project

This option allows the user to close the currently opened project. The user is provided with a confirmation dialog to confirm the selection for closing the project (Figure B.5).

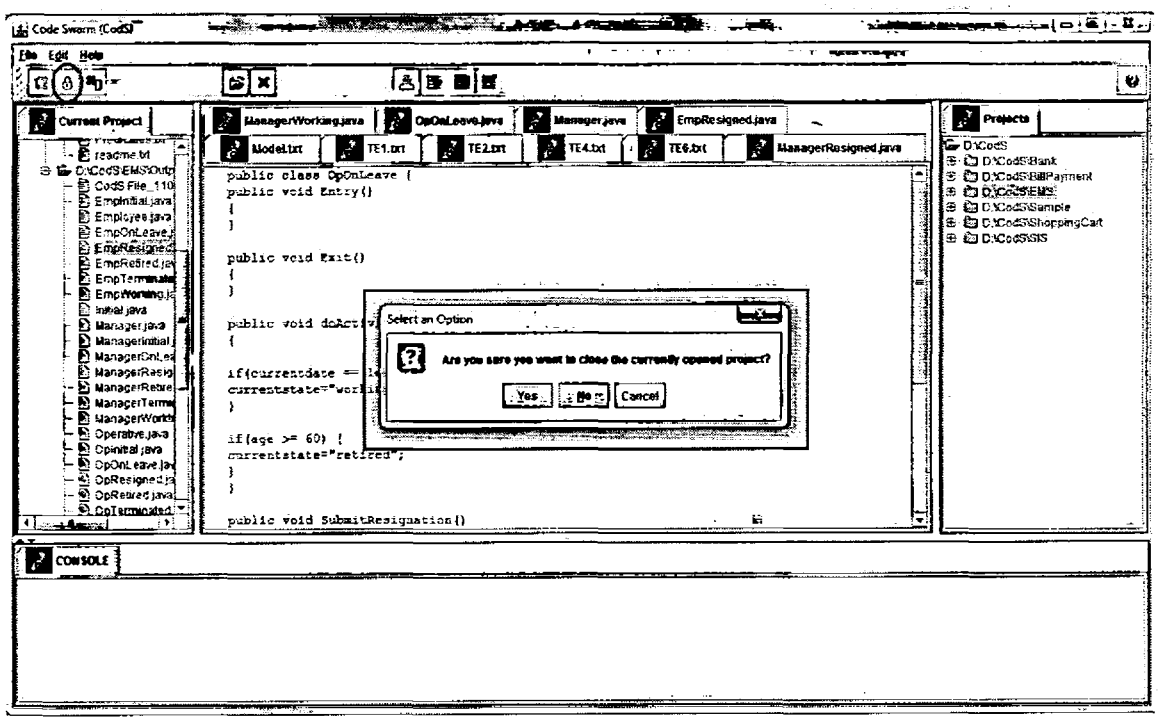


Figure B.5 Close project

B.2.3. Delete Project

Selecting this option will enable the user to delete the currently opened project. When the user selects this option, he/she is asked to confirm the decision for deleting the project (Figure B.6, B.7).

B.2.4. Open File

A text file can be opened by selecting this option. When the user selects this option, a dialog box is opened to browse the file (Figure B.8). Picture formats are not supported by this CØdŞ application. The opened file is displayed in the center panel.

B.2.5. Close File

This option permits the user to close the file which is currently in focus. The user can also close the file by making right-click on the tab and selecting the appropriate option (Figure B.9).

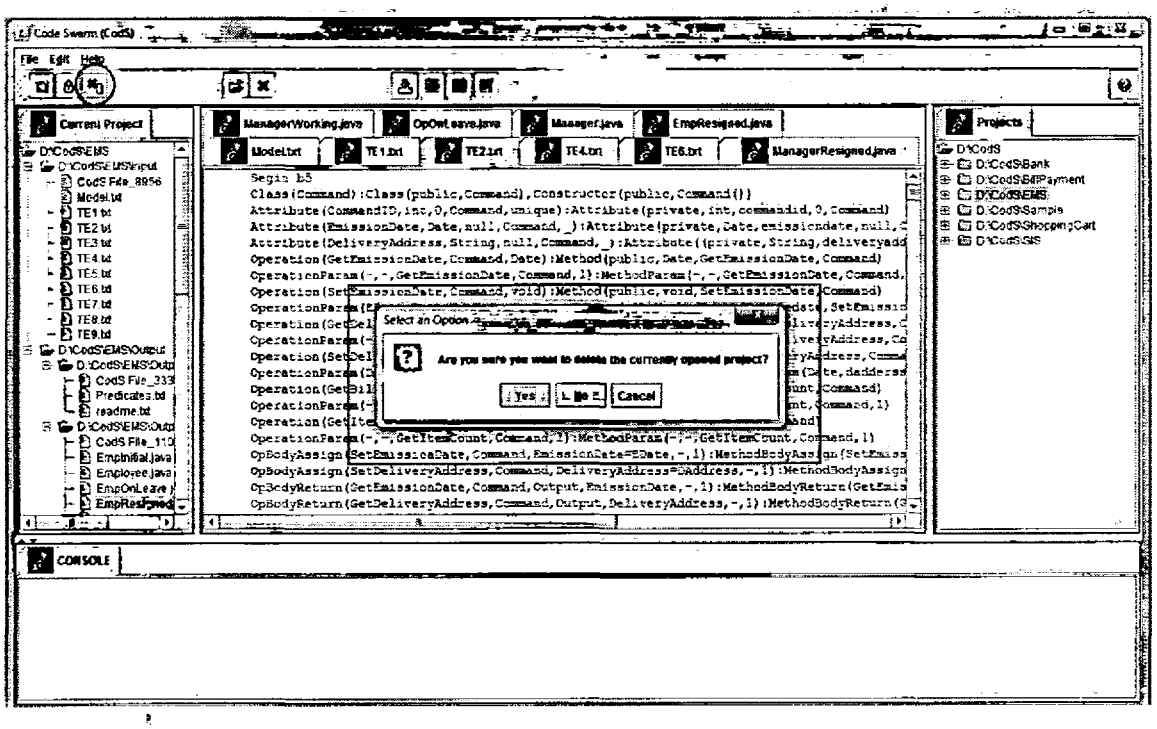


Figure B.6. Delete project confirmation

B.2.6. Exit

This option is used to close the CØdŞ application. Upon selection of this option, user is asked to confirm the decision (Figure B.10, B.11).

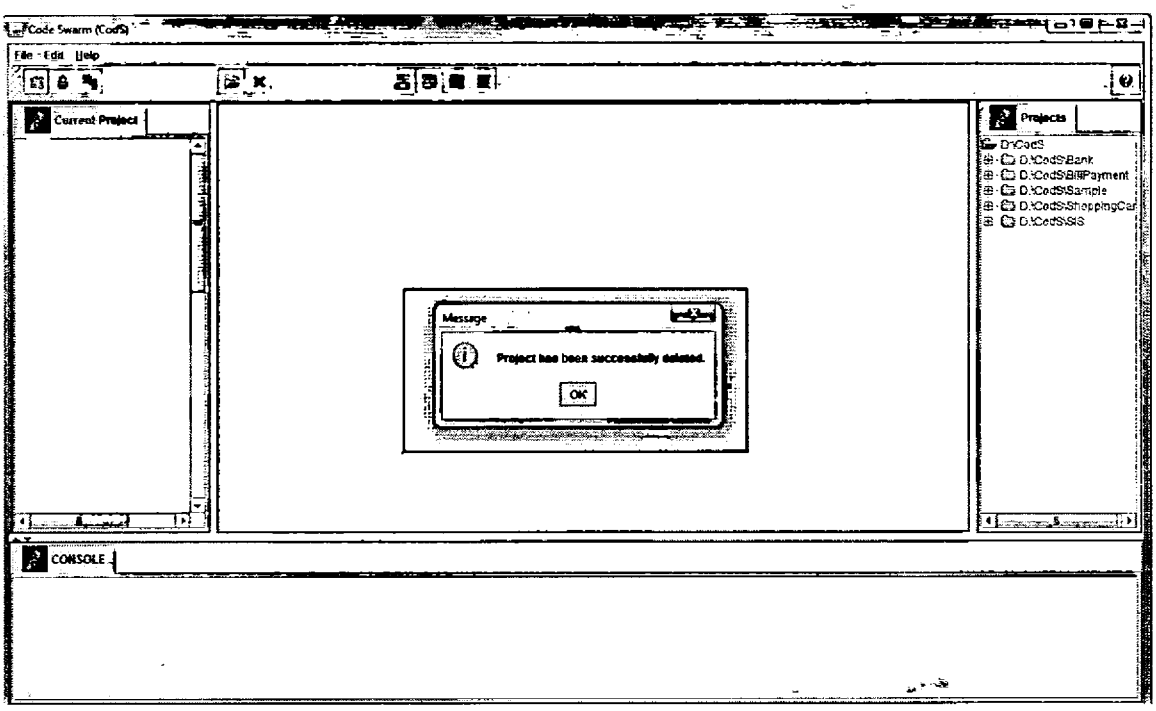


Figure B.7 Delete project confirmed

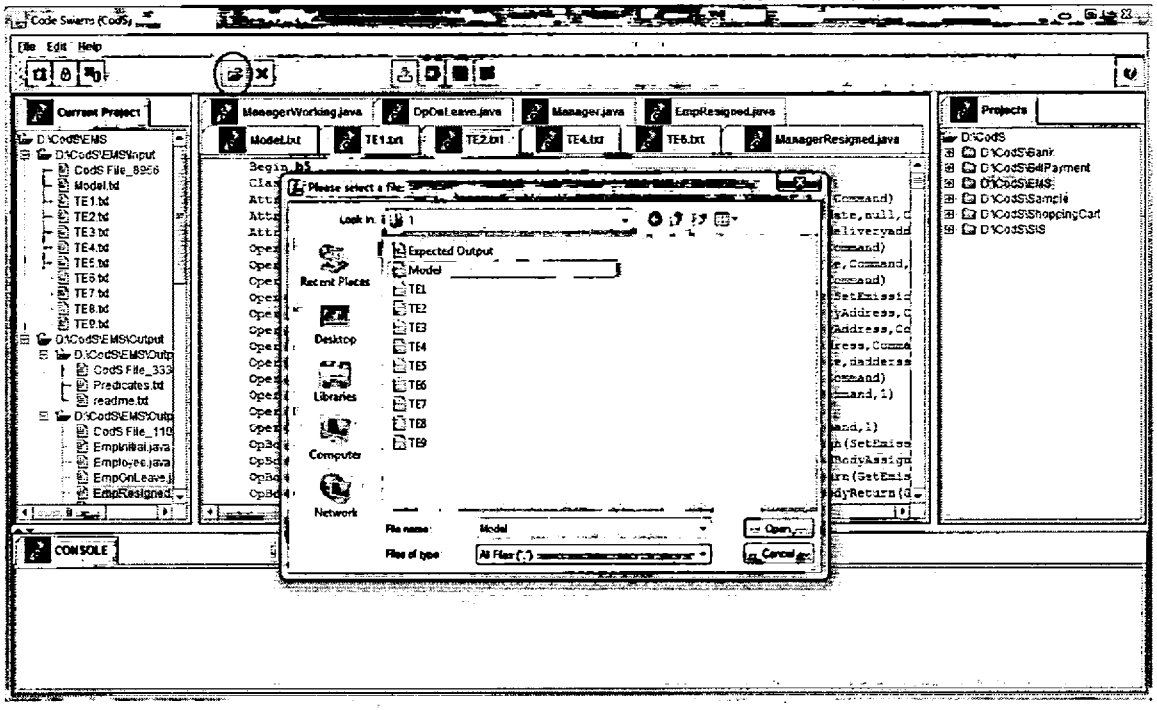


Figure B.8 Open file

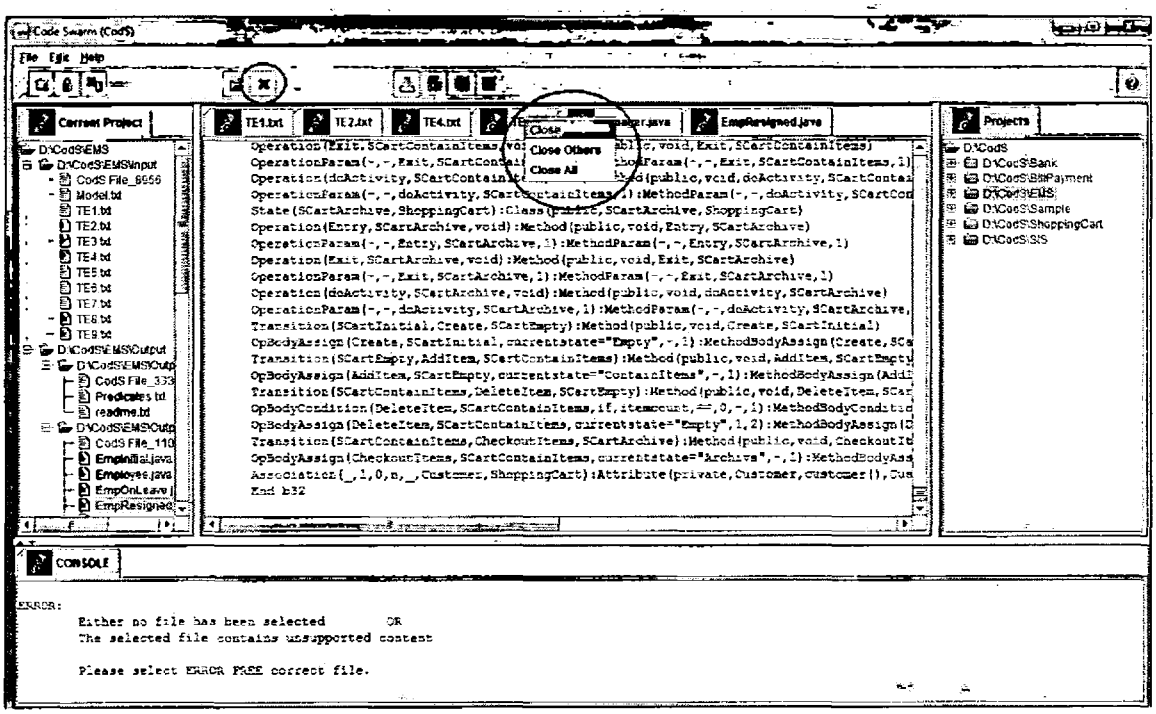


Figure B.9 Close file

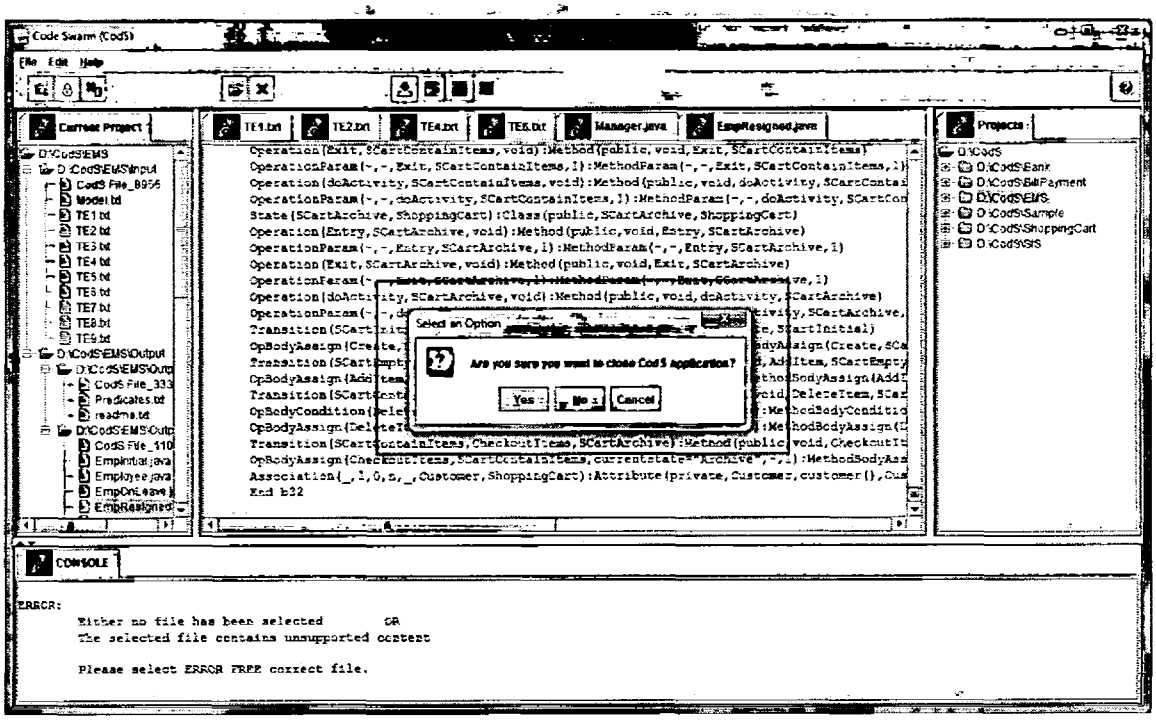


Figure B.10 Exit

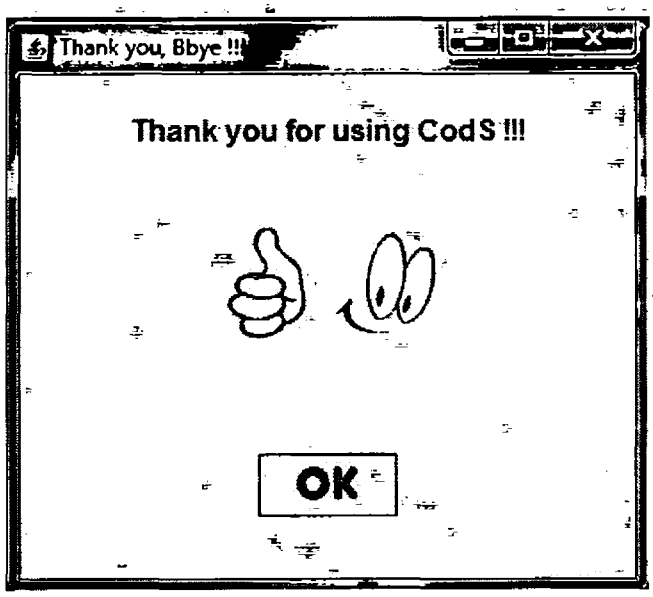


Figure B.11 Thank you

B.3 Edit Menu

This menu provides the options for transforming system models into code predicates and complete Java code statements (Figure B.12).

B.3.1 Import Training Data

This option allows the user to import files containing previously solved transformation examples represented in the form of mapping blocks. The content of these files are displayed in the center panel. The imported files are automatically made a part of the currently opened project. All files of training data must be imported at once (Figure B.13, B.14).

B.3.2 Import Input Model

This option is used to import the models from which code is to be generated. All the models must be stored in a single text file (Figure B.15, B.16).

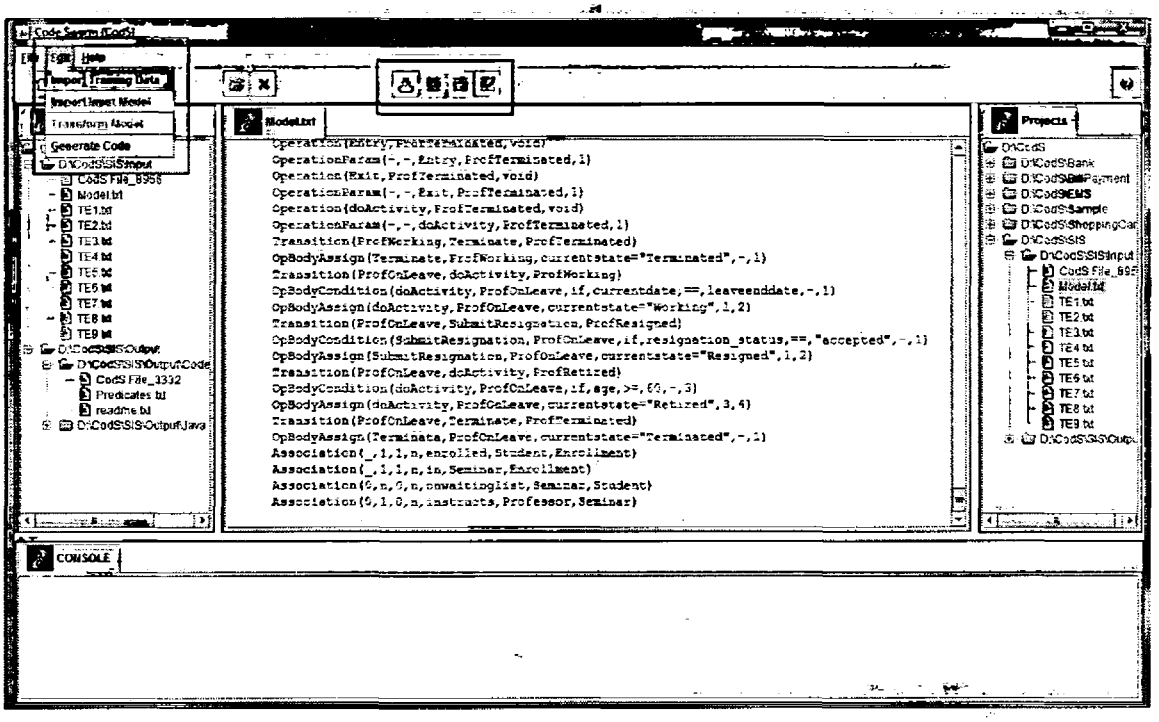


Figure B.12 Edit menu

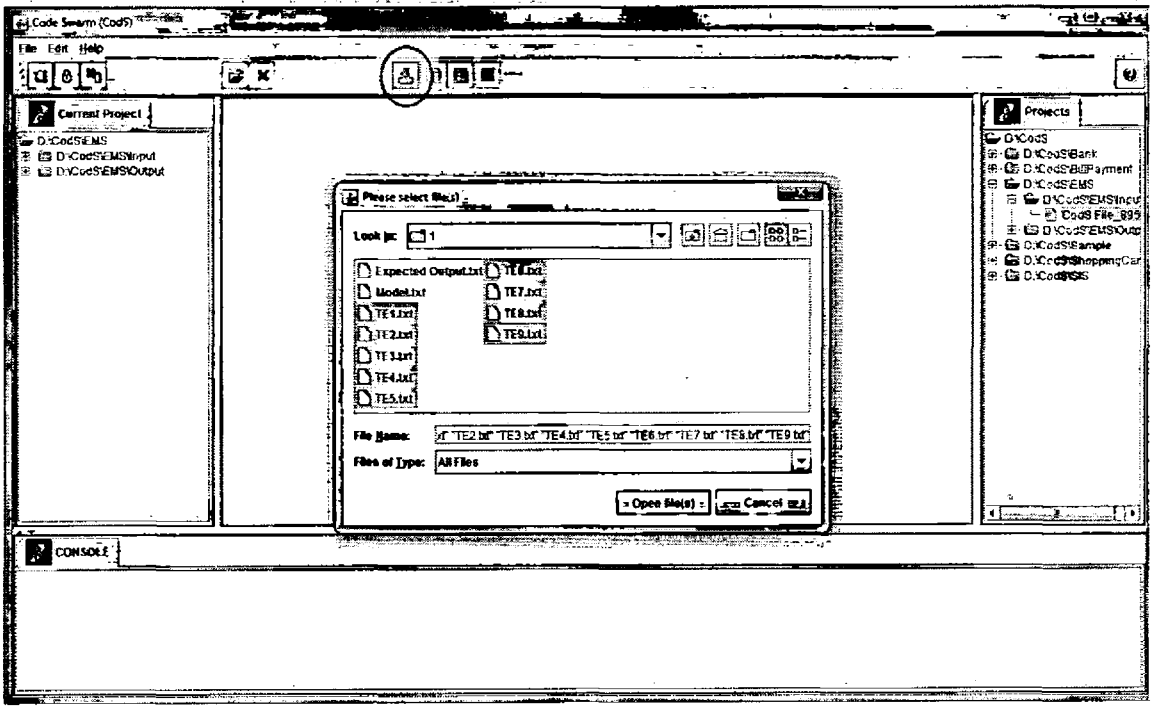


Figure B.13 Import training data - browsing dialog

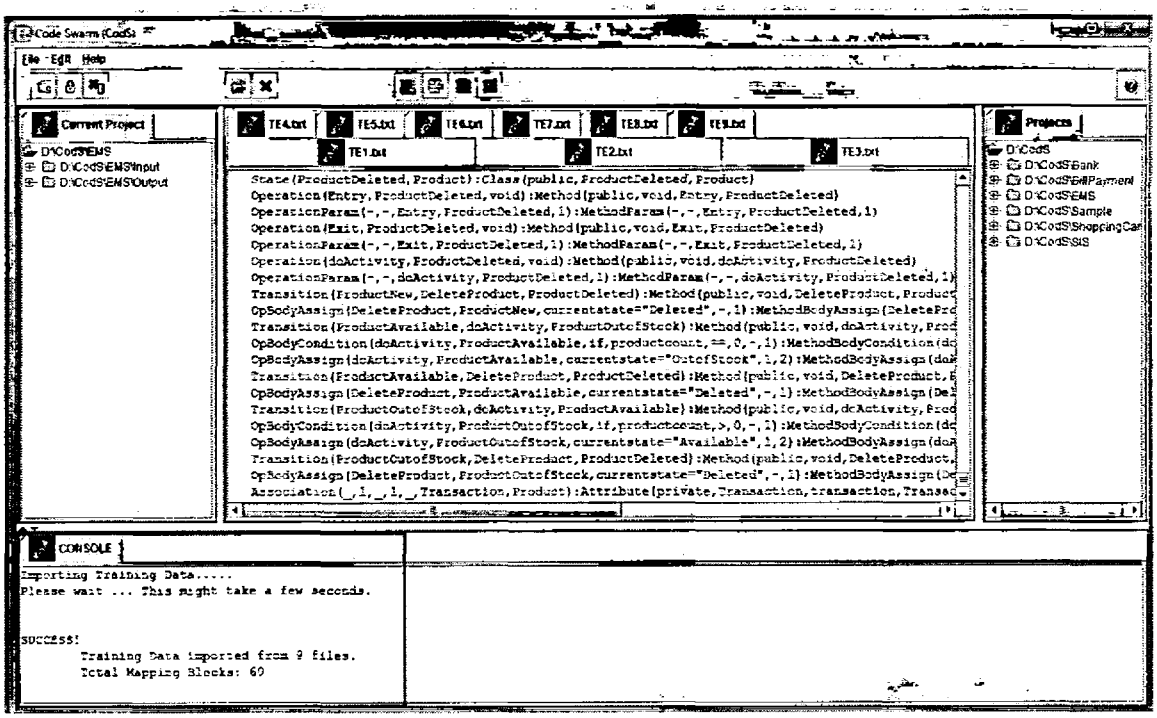


Figure B.14 Import training data - console

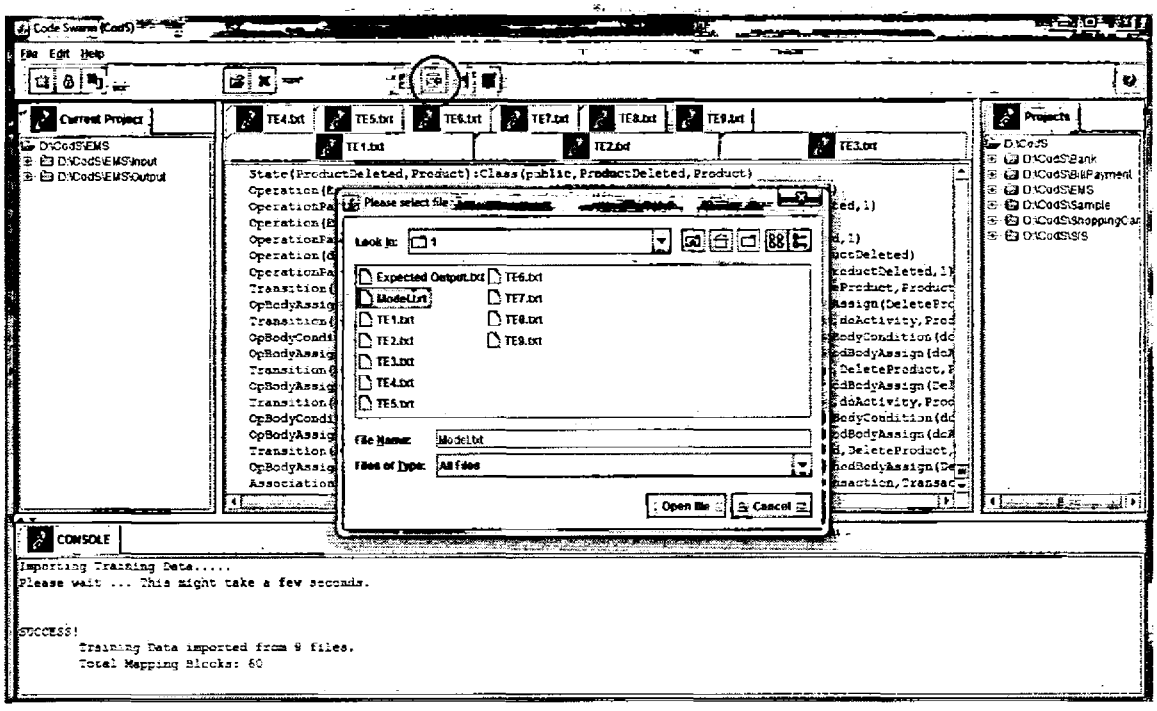


Figure B.15 Import input model - browsing dialog

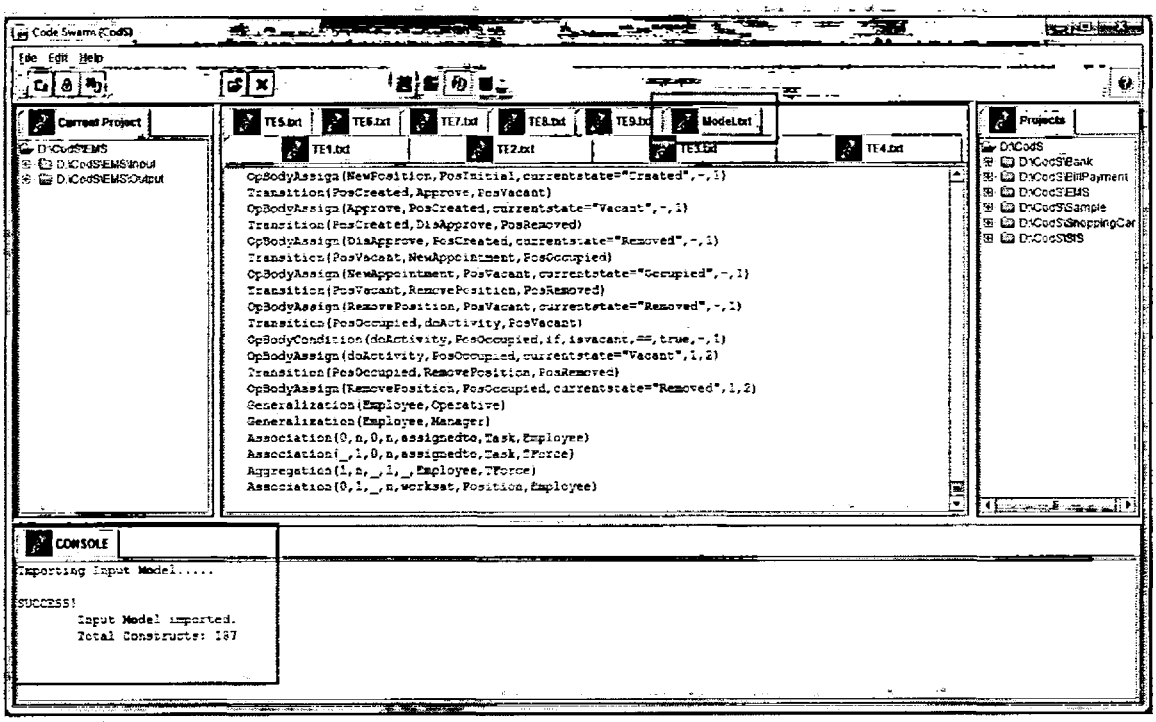


Figure B.16 Import input model - console

B.3.3 Transform Model

This option is used to transform input class and state models containing ASL into code predicates. The resultant predicates are stored in the file "Predicates.txt". The information about the transformation process is displayed on the console and stored in a file named "readme.txt" (Figure B.17, B.18).

B.3.4 Generate Code

This option is used to generate the complete code statements and Java code files corresponding to the previously produced code predicates (Figure B.19, B.20).

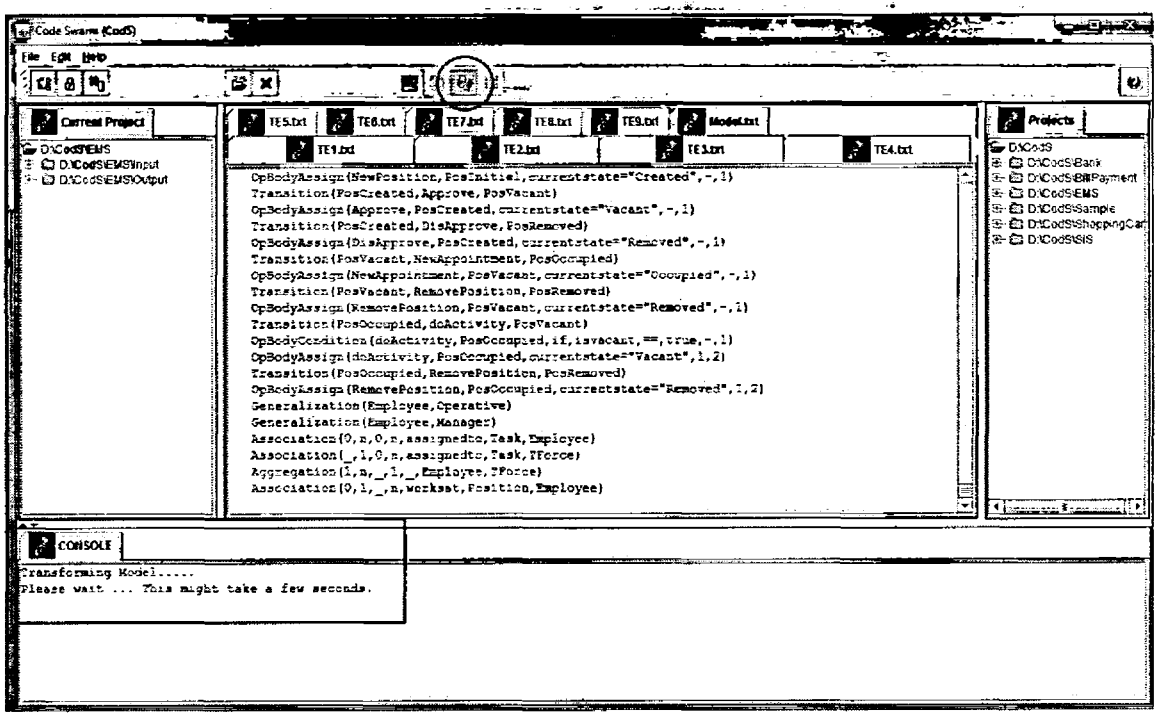


Figure B.17 Transform model - in progress

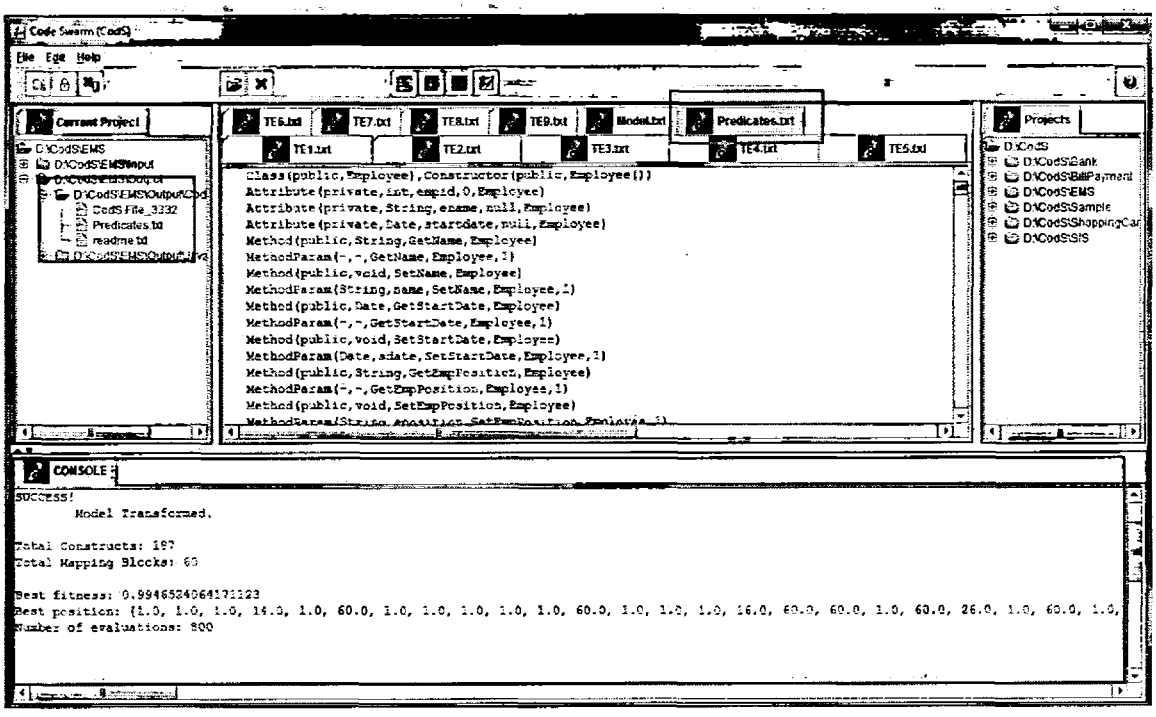


Figure B.18 Transform model - process completed

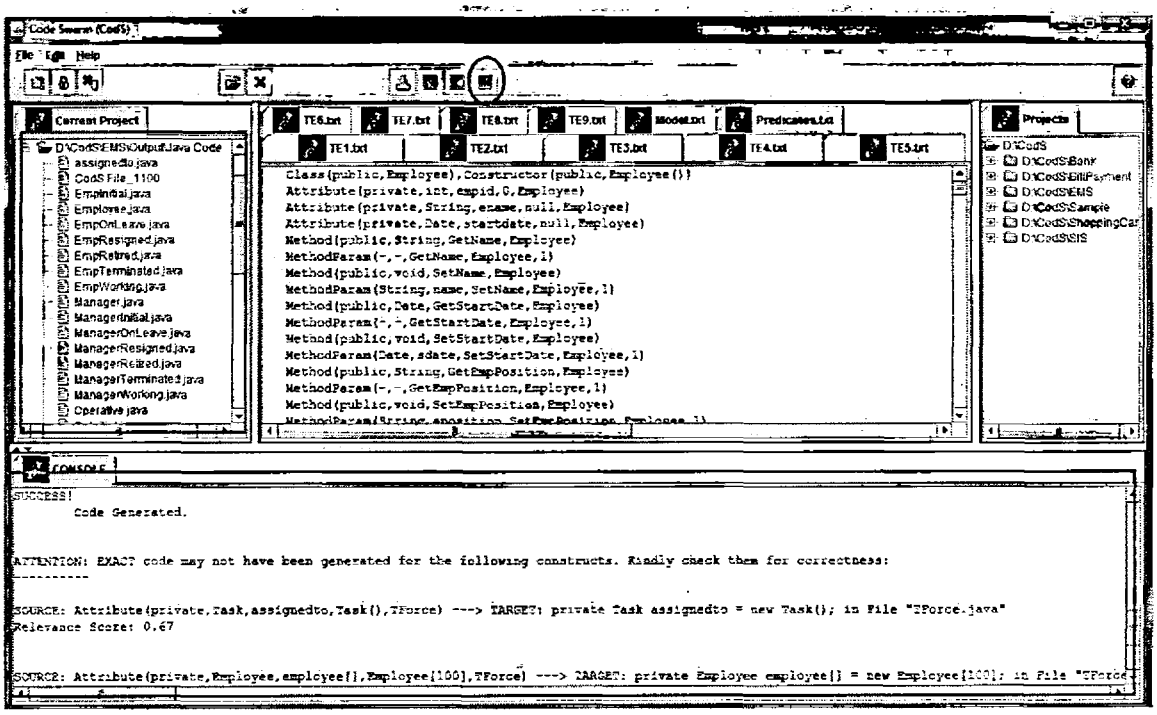


Figure B.19 Generate code - in progress

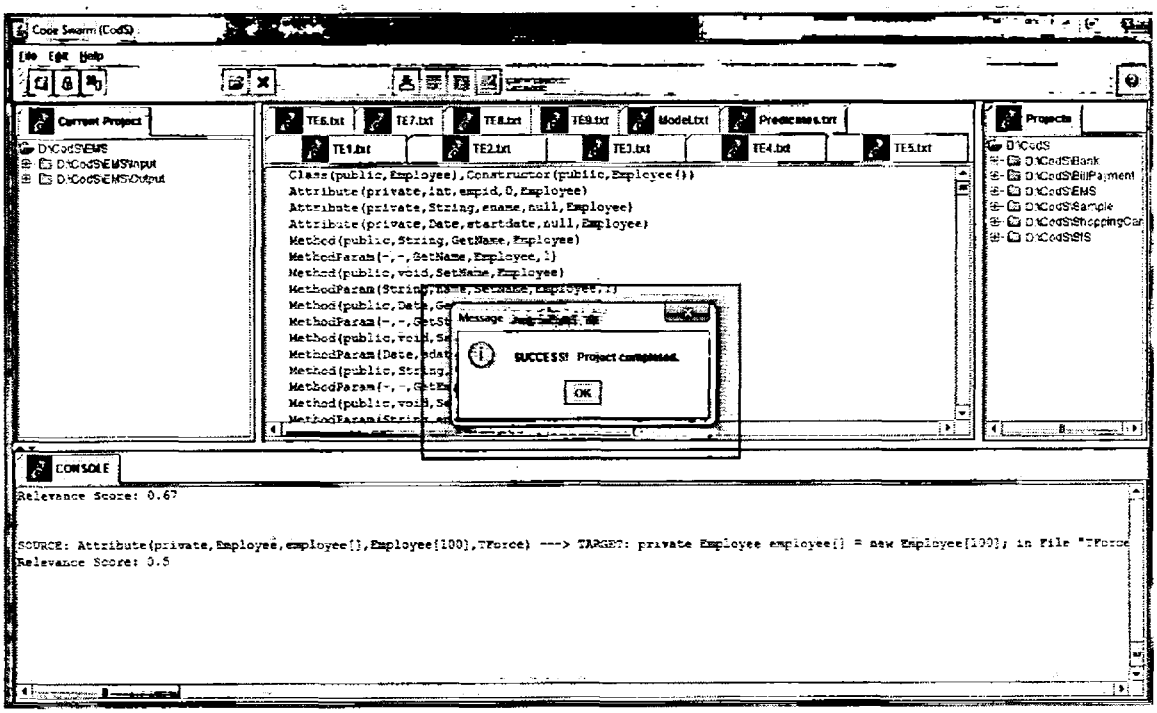


Figure B.20 Generate code - process completed

B.4 Help Menu

This option opens the frame for displaying help content to the user. The following figures show the different views of help content.

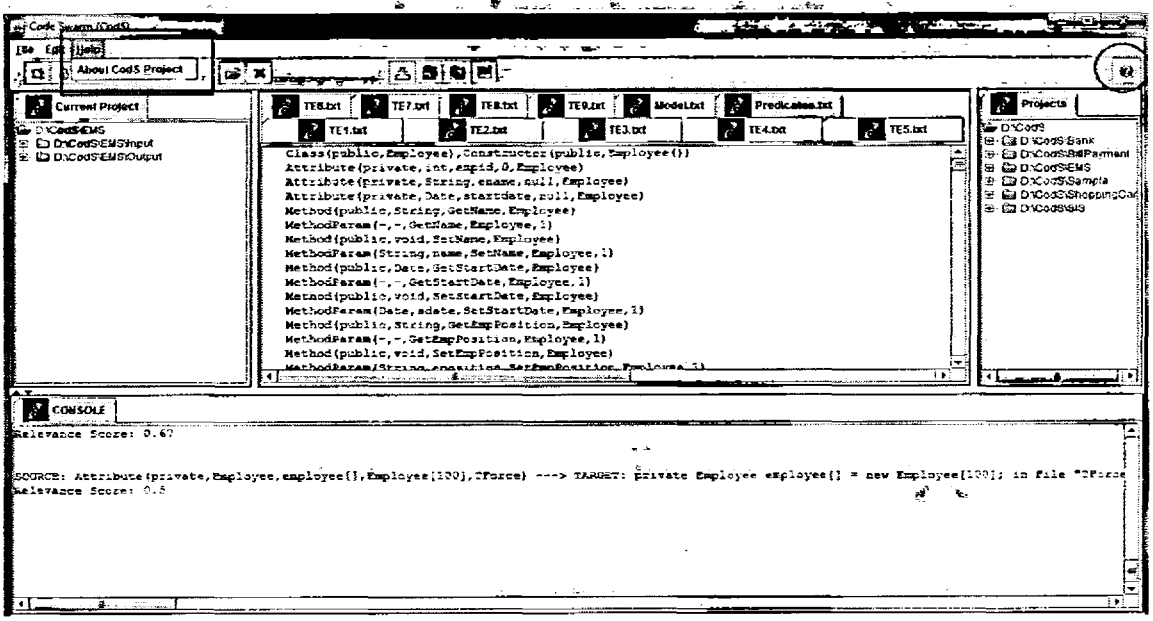


Figure B.21 Help menu

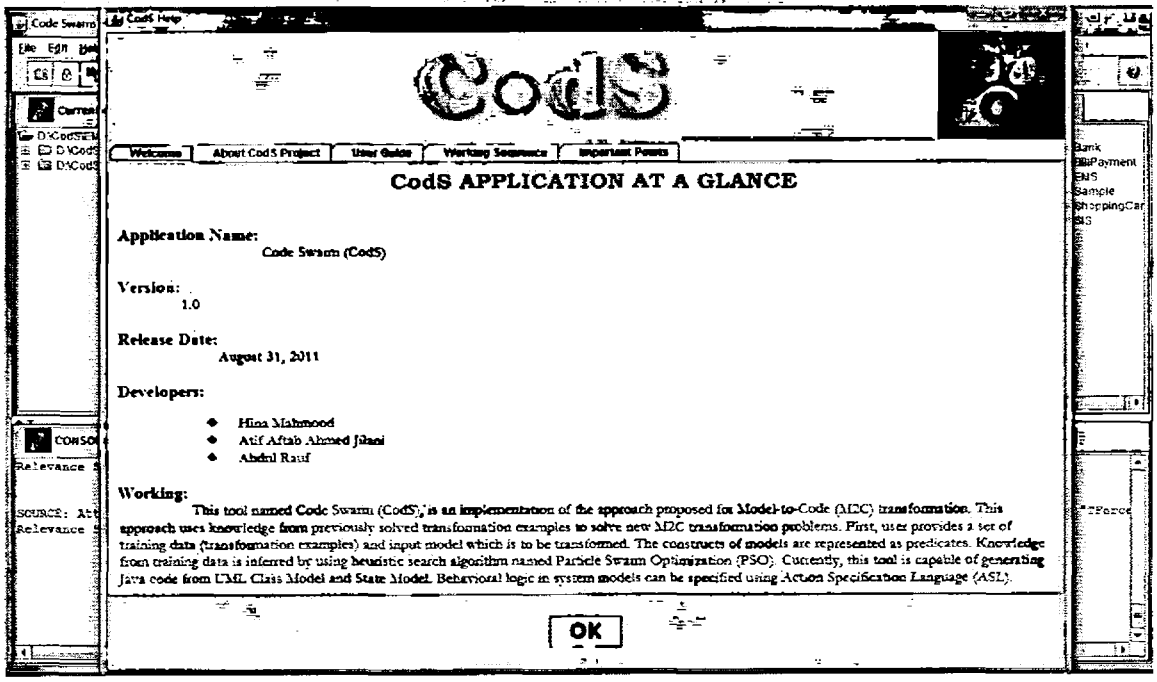


Figure B.22 Help menu - welcome screen

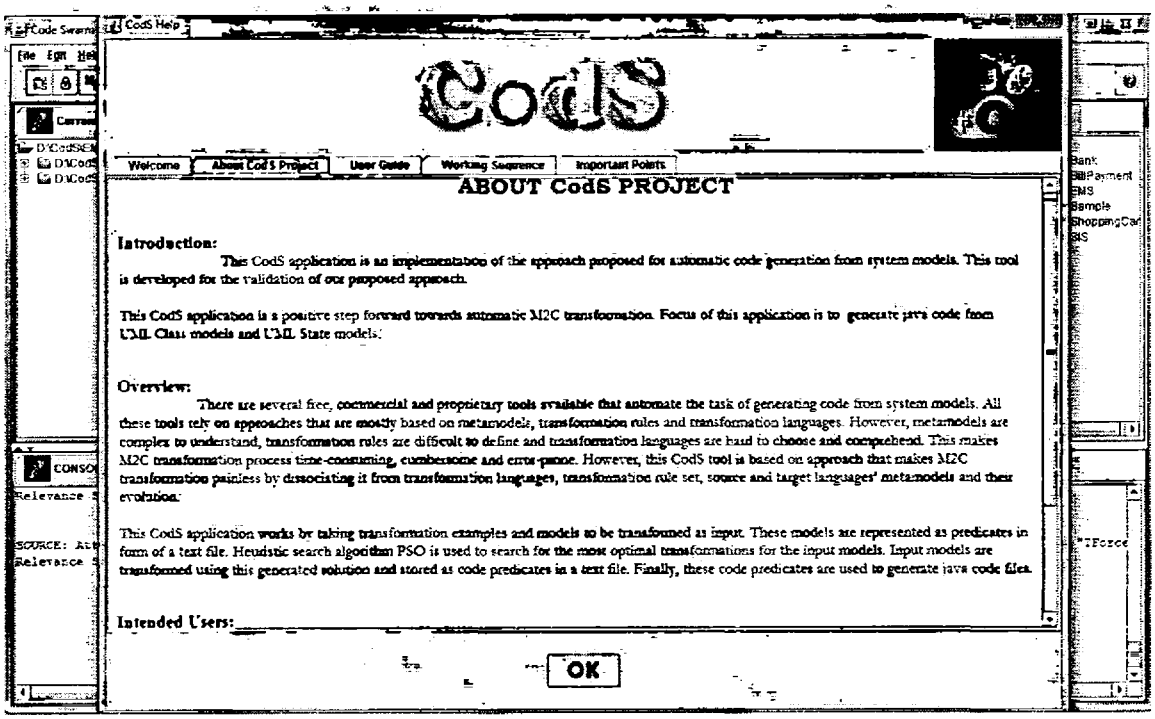


Figure B.23 Help menu - about CØdŞ project

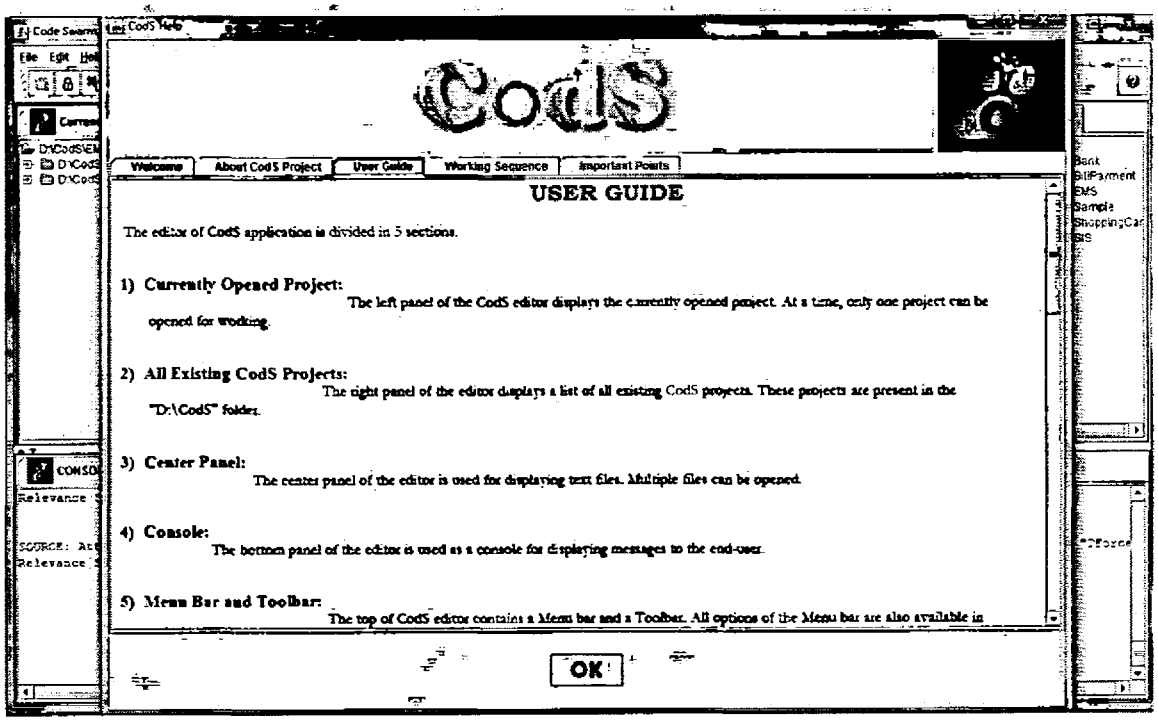


Figure B.24 Help menu - user guide

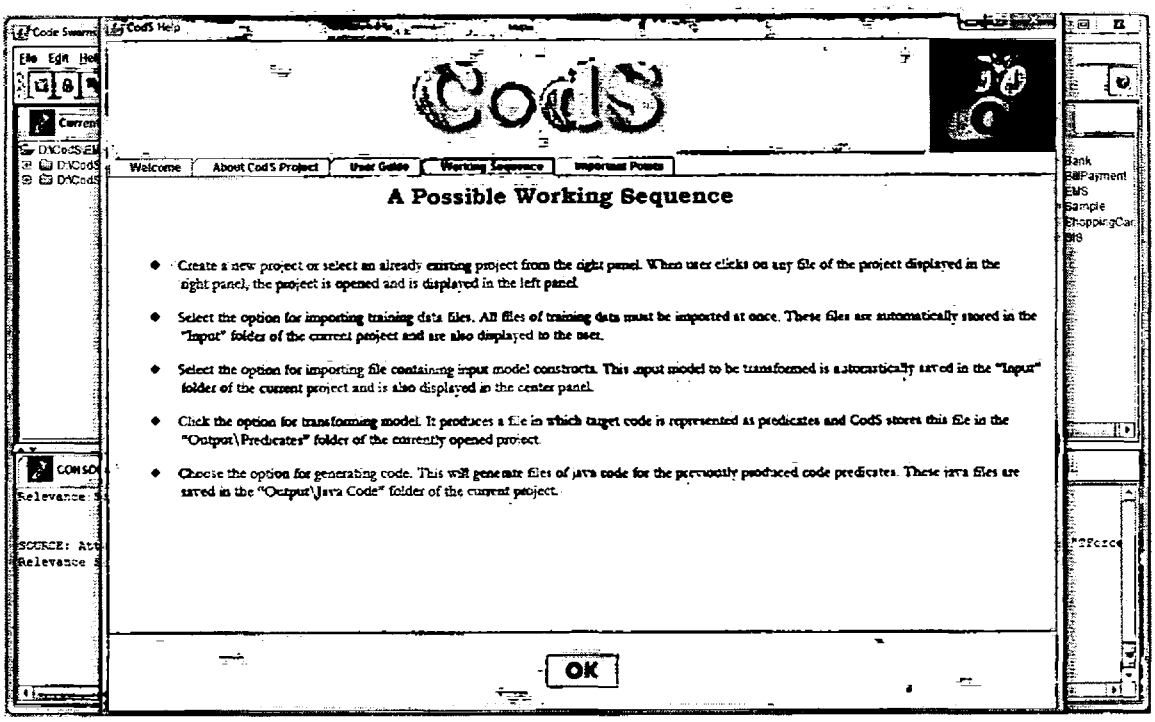


Figure B.25 Help menu - working sequence

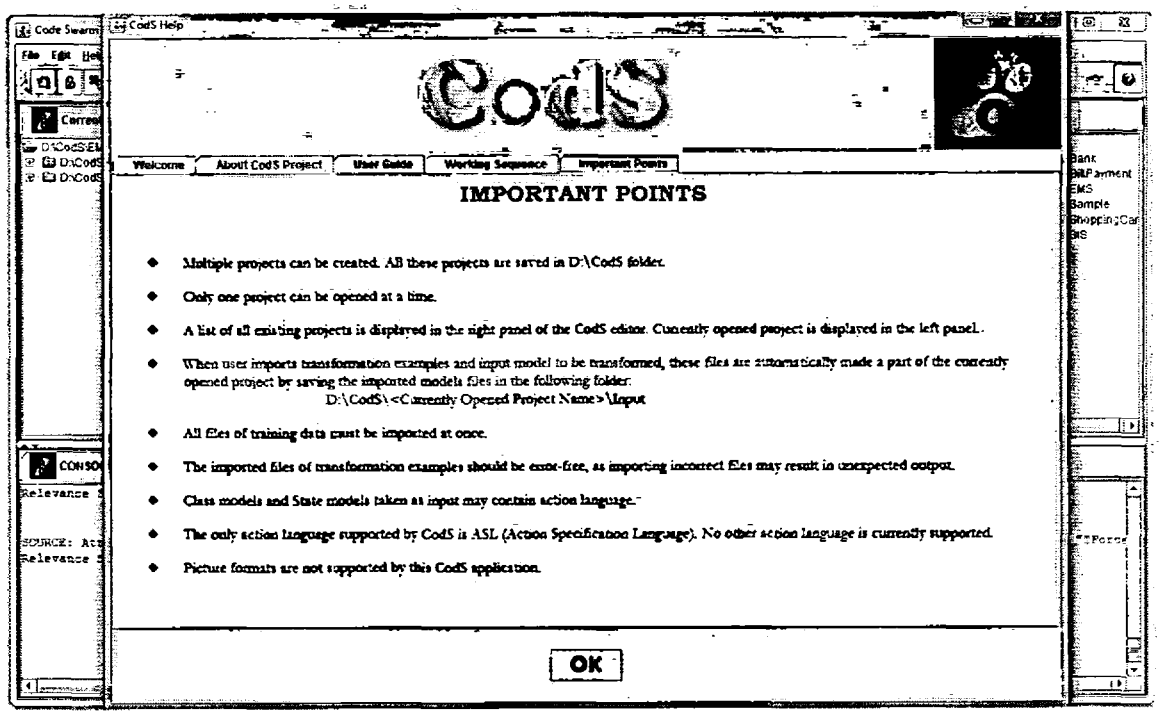


Figure B.26 Help menu - important points

Appendix C

TRAINING DATA

C.1 Model 1 (M1)

This section illustrates the class and state diagrams for the 'Task Management System'.

C.1.1 Class Model

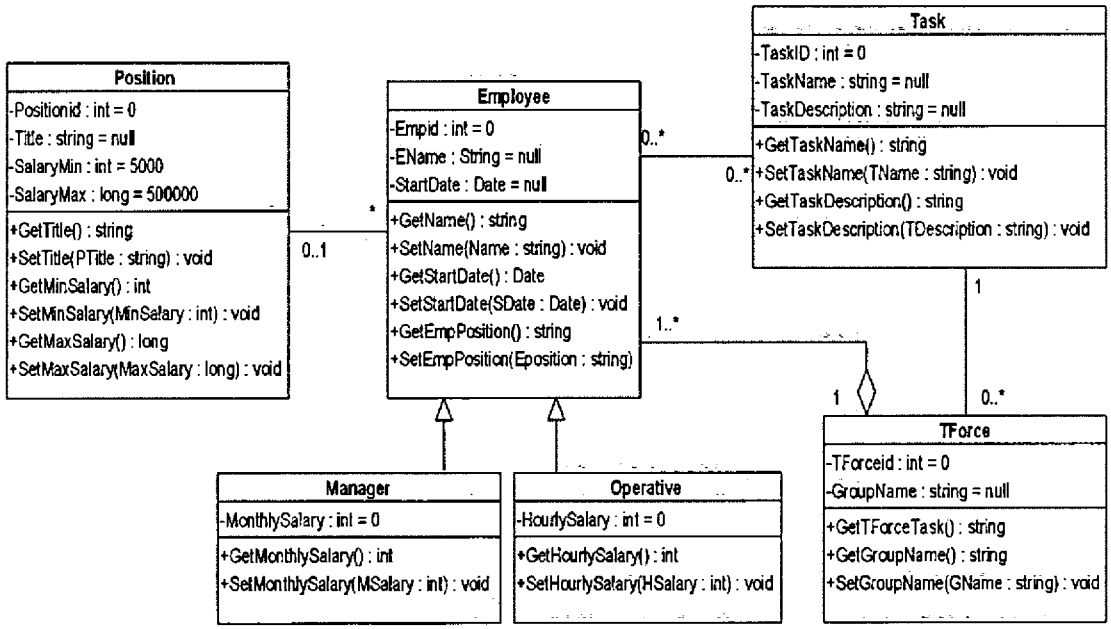


Figure C.1 Class model of 'Task Management System'

C.1.2 State Model

This section shows the state models corresponding to classes in the class model of the Task Management System.

State Model of 'Employee'

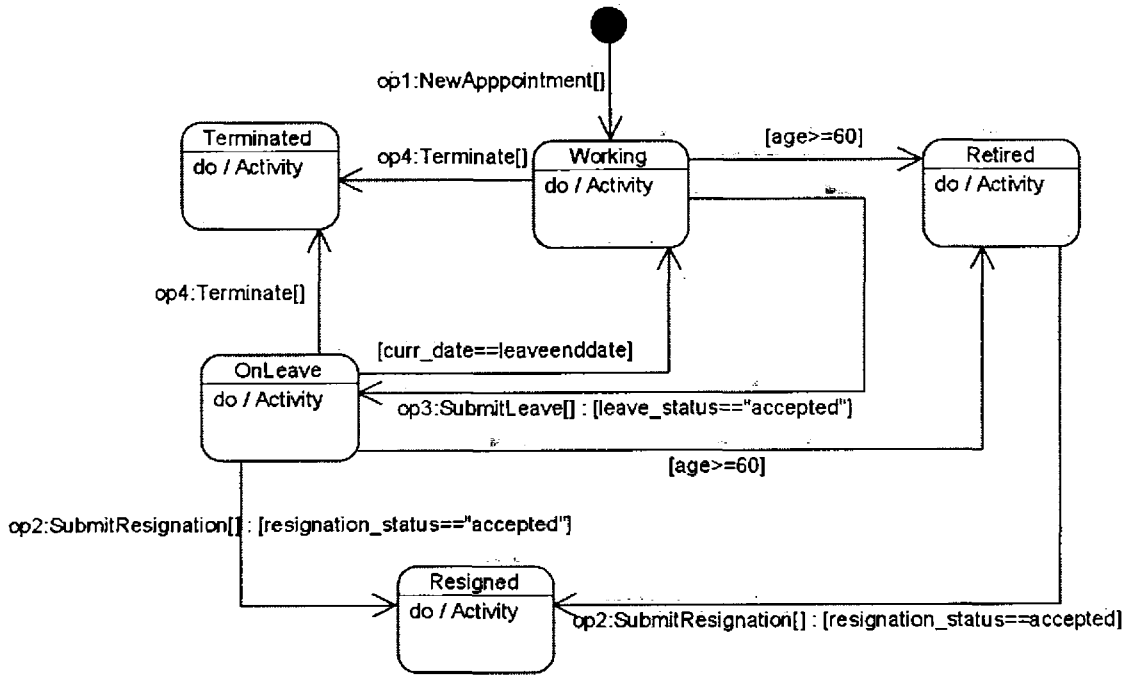


Figure C.2 State model of 'Employee'

State Model of 'Task'

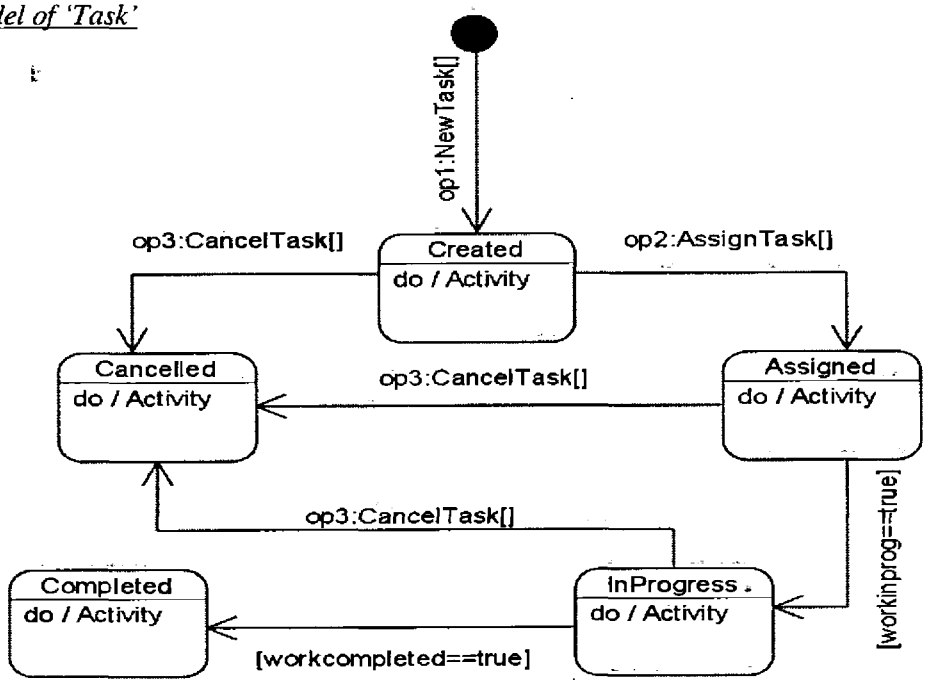


Figure C.3 State model of 'Task'

State Model of 'TForce'

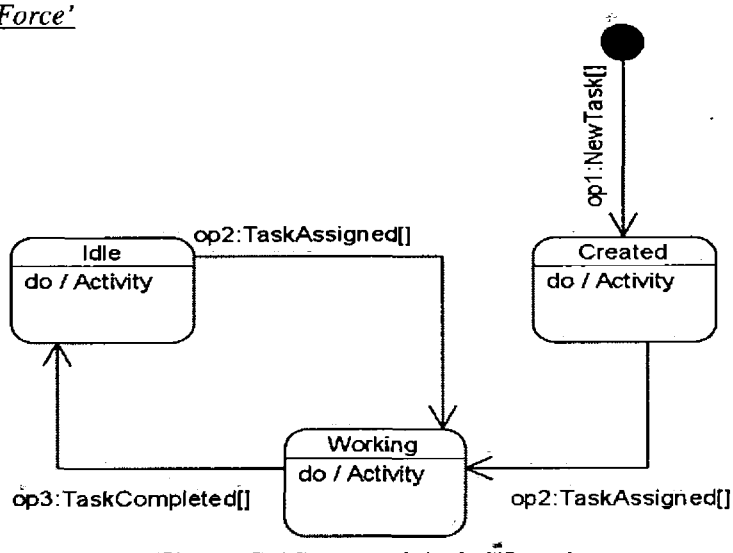


Figure C.4 State model of 'TForce'

State Model of 'Position'

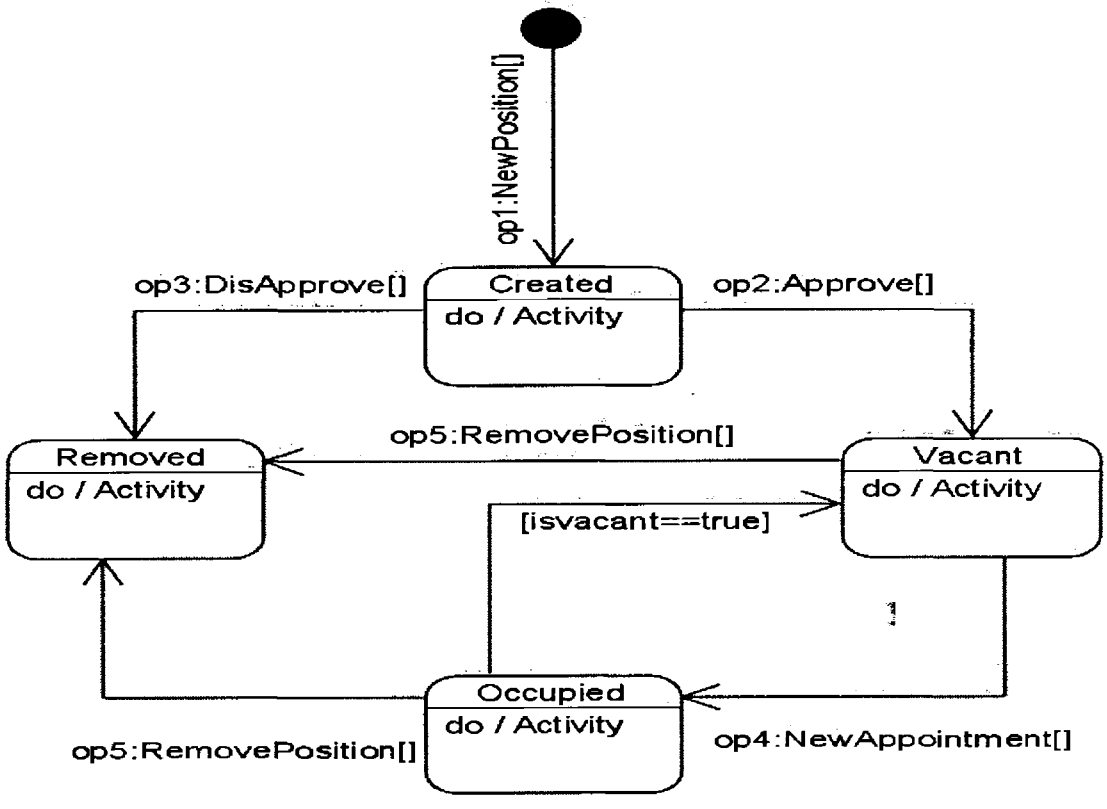


Figure C.5 State model of 'Position'

C.2 Model 2 (M2)

This section shows the class and state models for the application of 'Book Bank'.

C.2.1 Class Model

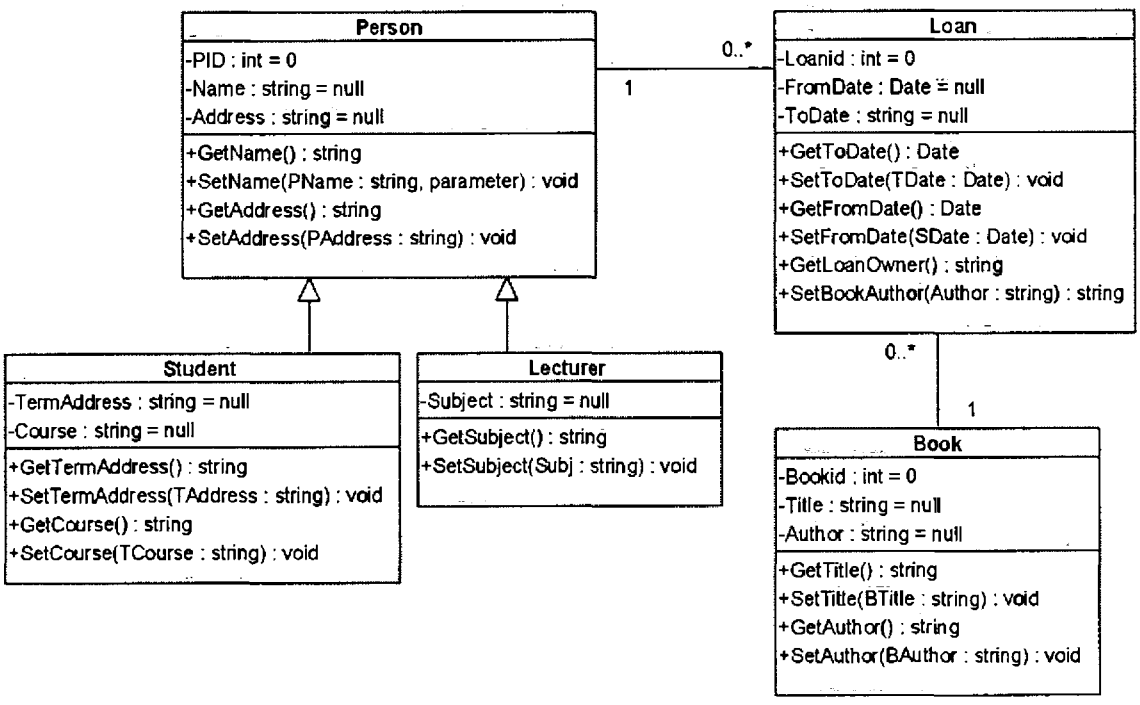


Figure C.6 Class model of 'Book Bank'

C.2.2 State Model

The state models of the Book Bank application are presented in this section.

State Model of 'Person'

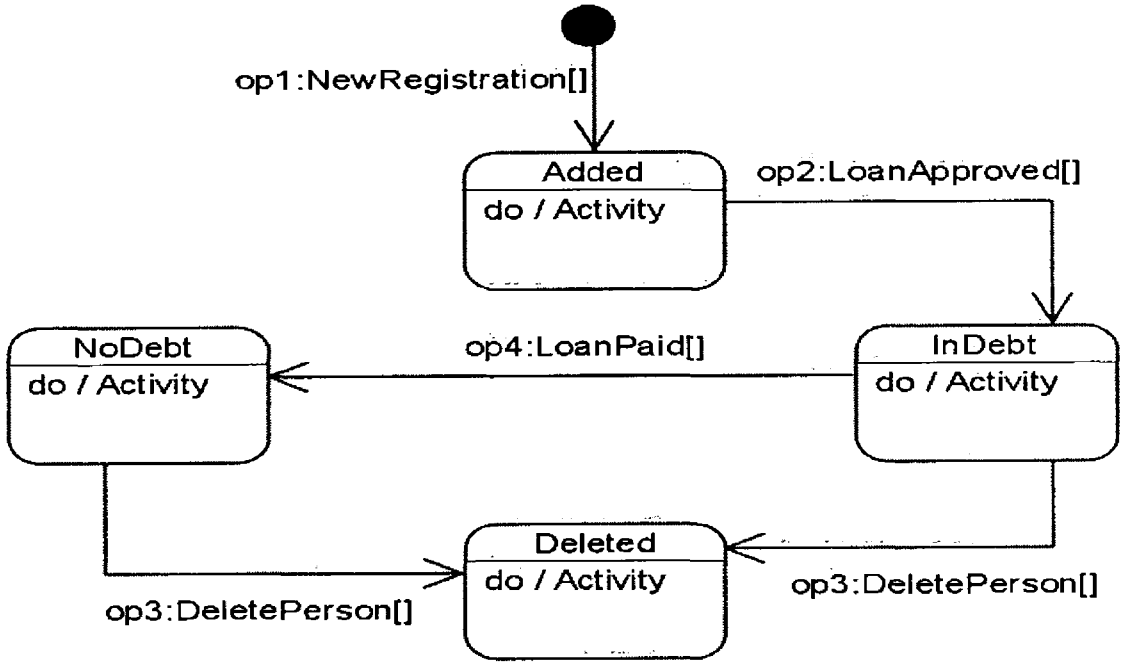


Figure C.7 State model of 'Person'

State Model of 'Loan'

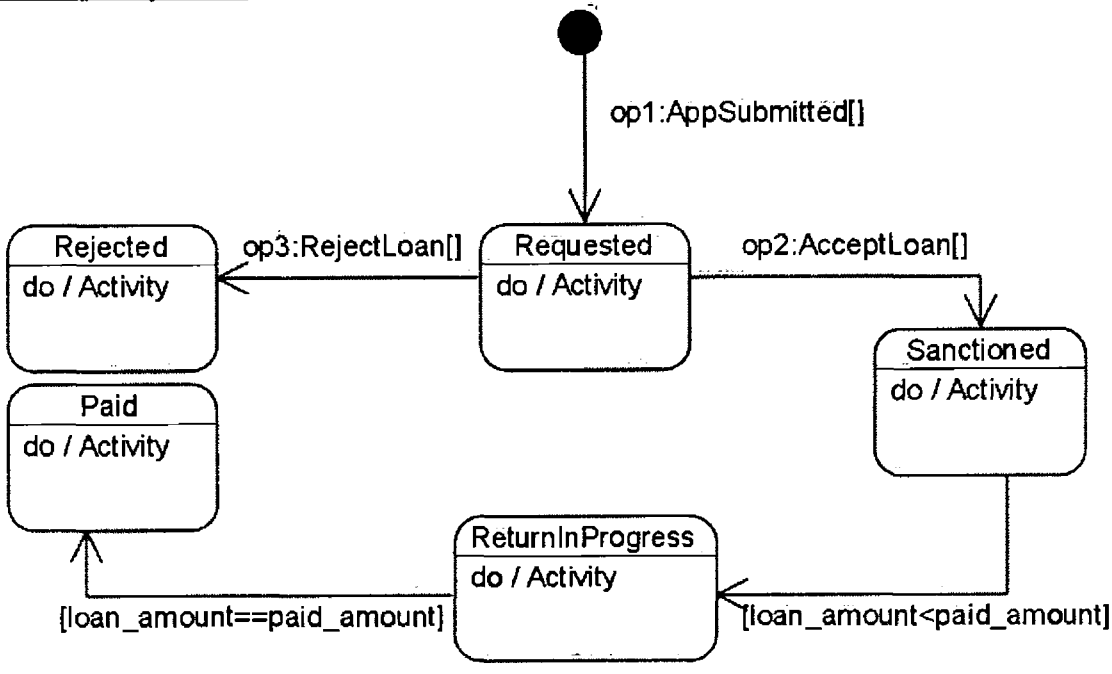


Figure C.8 State model of 'Loan'

State Model of 'Book'

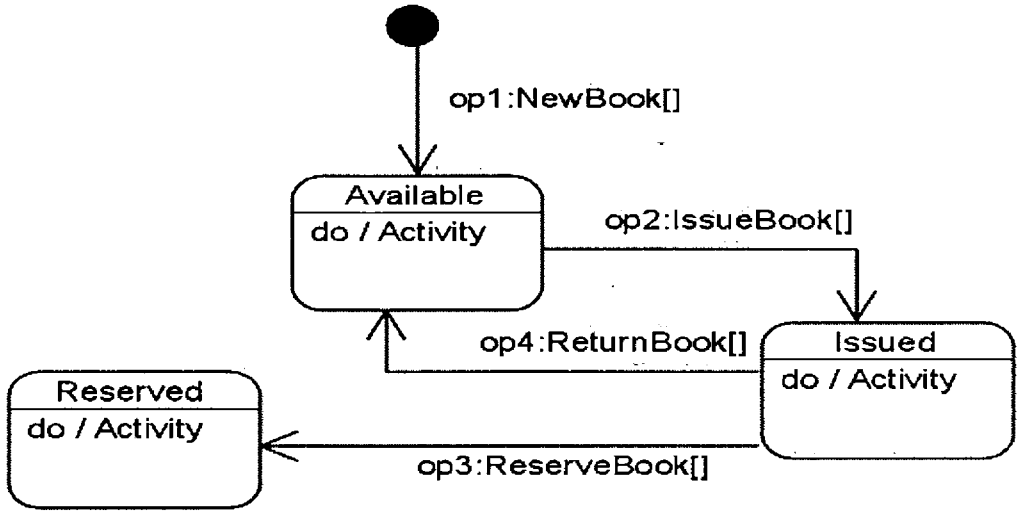


Figure C.9 State model of 'Book'

C.3 Model 3 (M3)

This section illustrates the class and state models for the 'Bill Payment System'.

C.3.1 Class Model

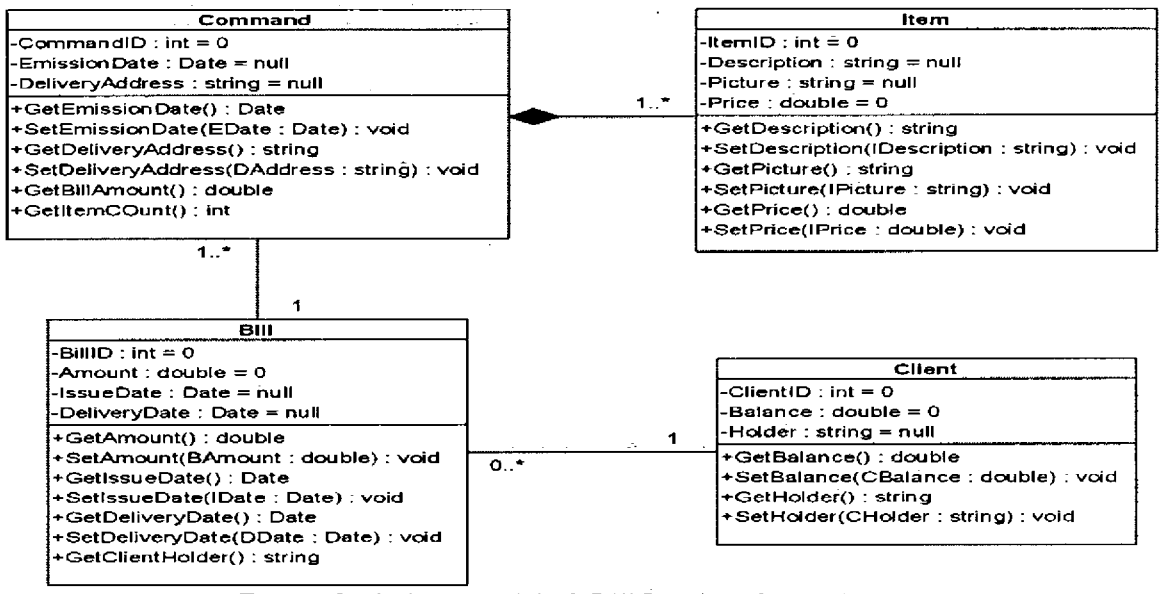


Figure C.10 Class model of 'Bill Payment System'

C.3.2 State Model

This section demonstrates the state models corresponding to the classes of the Bill Payment System.

State Model of 'Command'

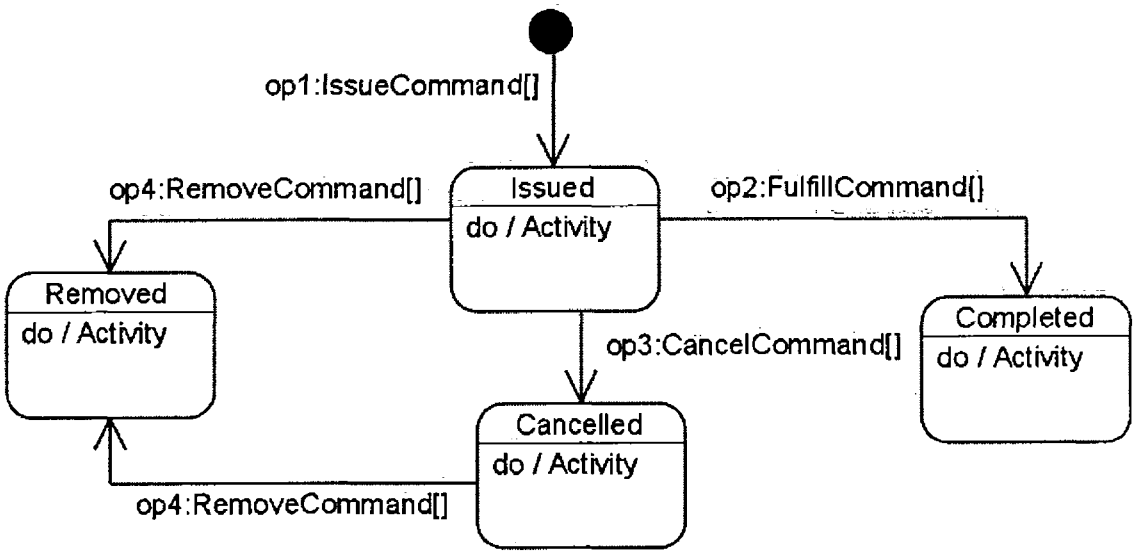


Figure C.11 State model of 'Command'

State Model of 'Bill'

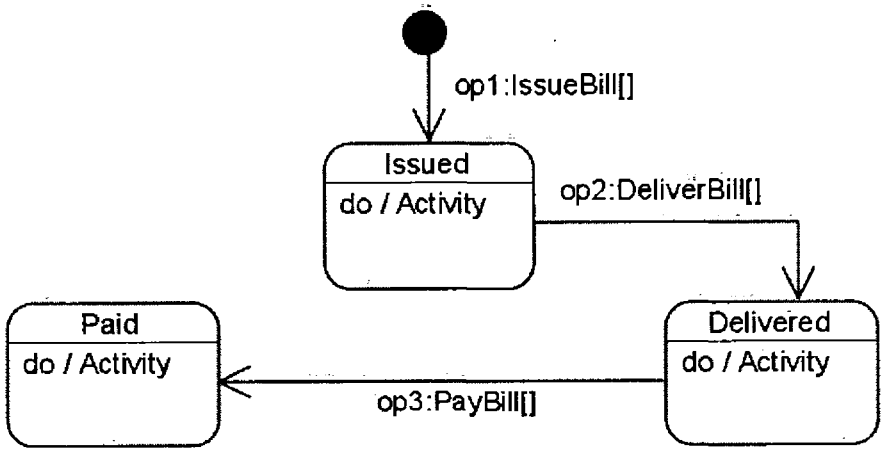


Figure C.12 State model of 'Bill'

State Model of 'Item'

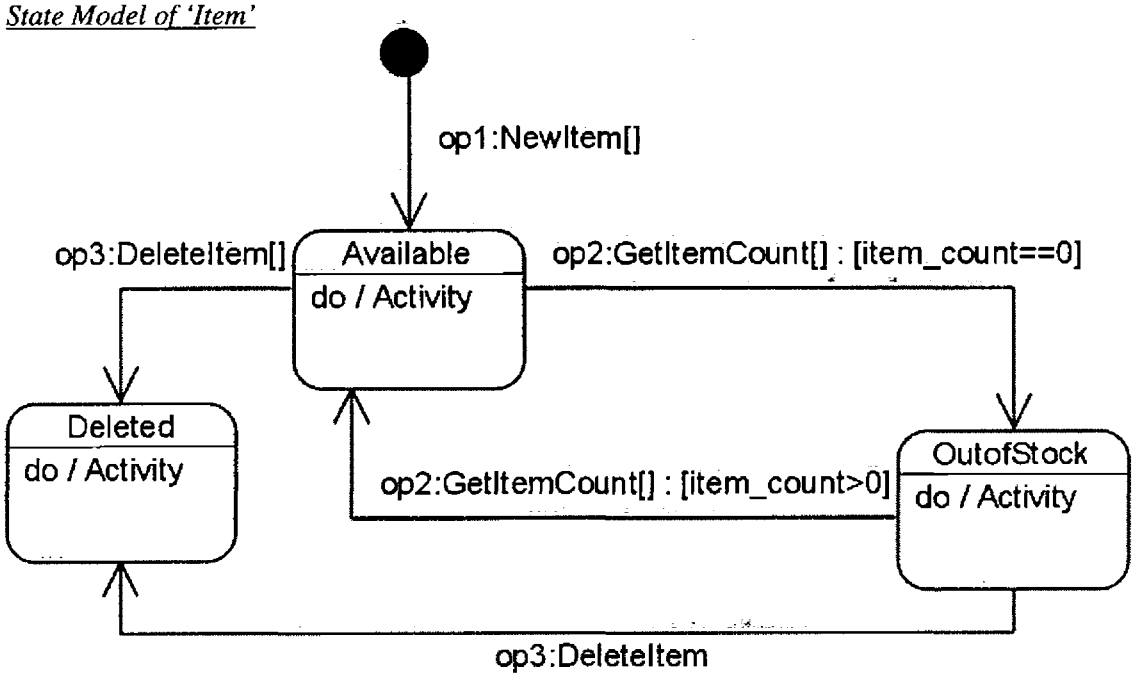


Figure C.13 State model of 'Item'

State Model of 'Client'

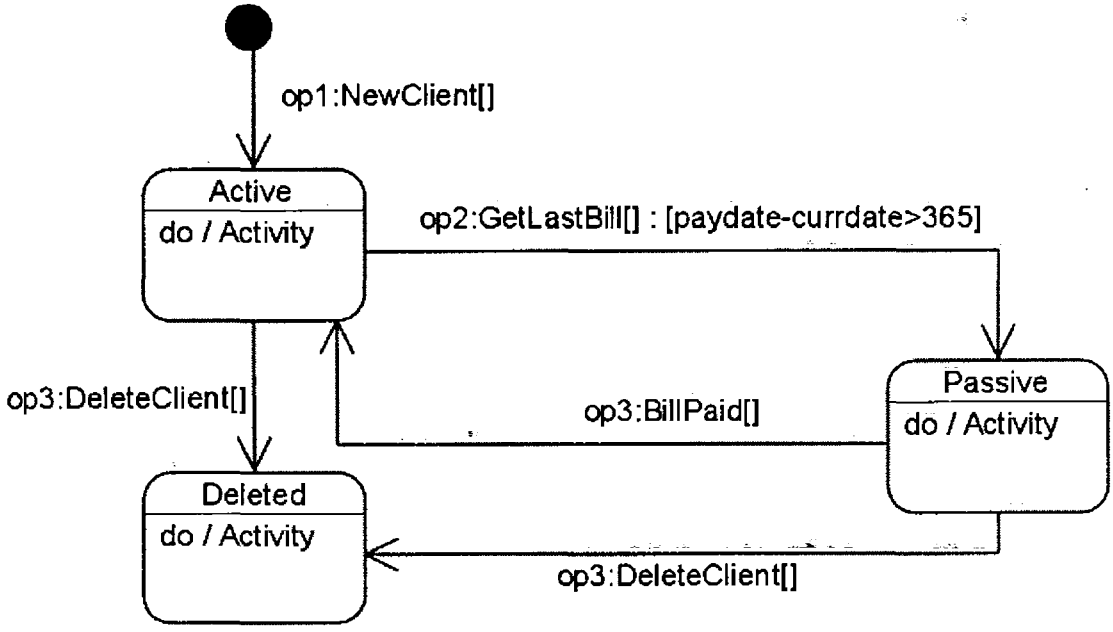


Figure C.14 State model of 'Client'

C.4 Model 4 (M4)

This section illustrates the class and state diagrams for the 'Student Enrollment System'.

C.4.1 Class Model

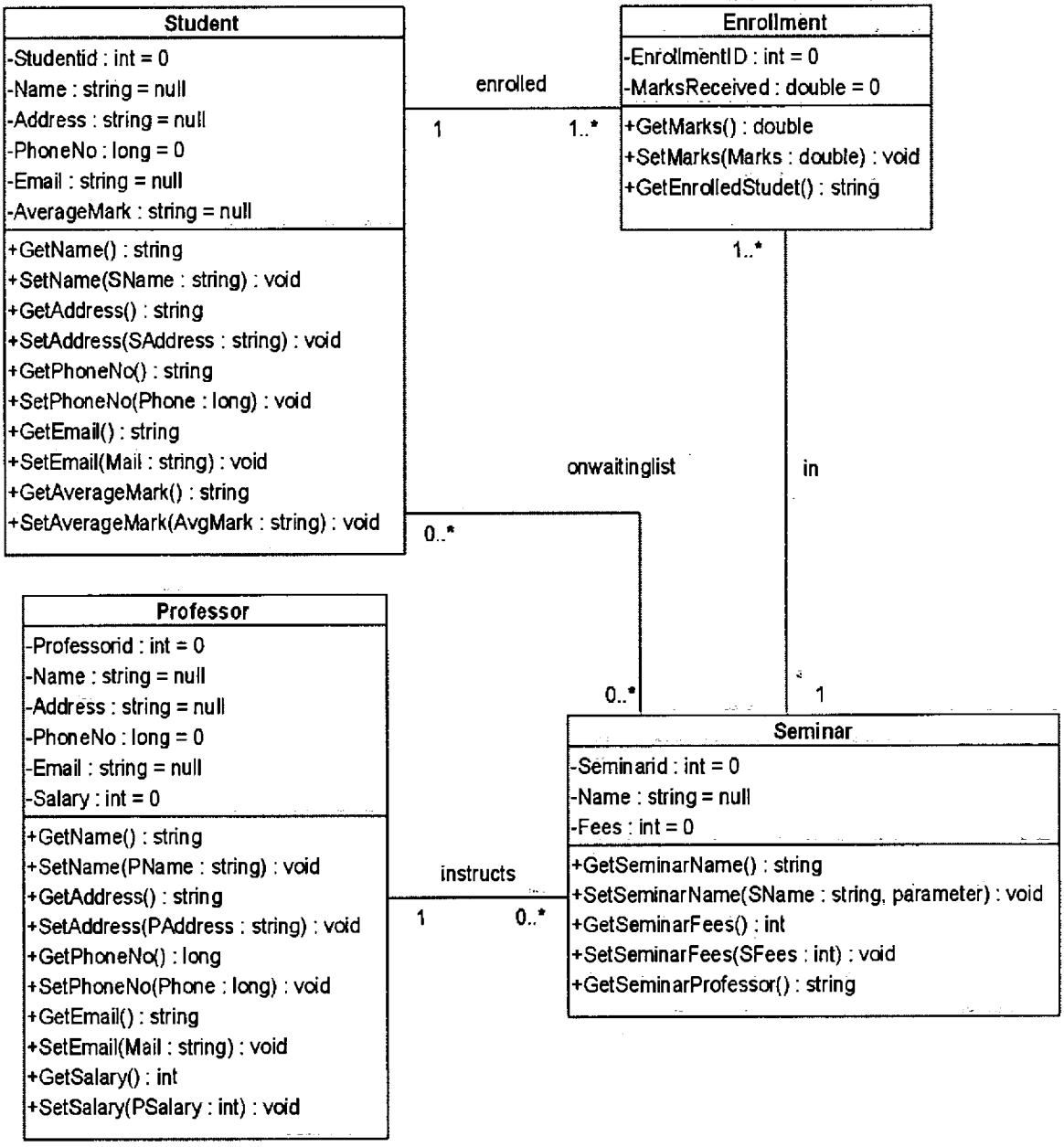


Figure C.15 Class model of 'Student Enrollment System'

C.4.2 State Model

The state models corresponding to the classes of the Student Enrollment System are presented in this section.

State Model of 'Student'

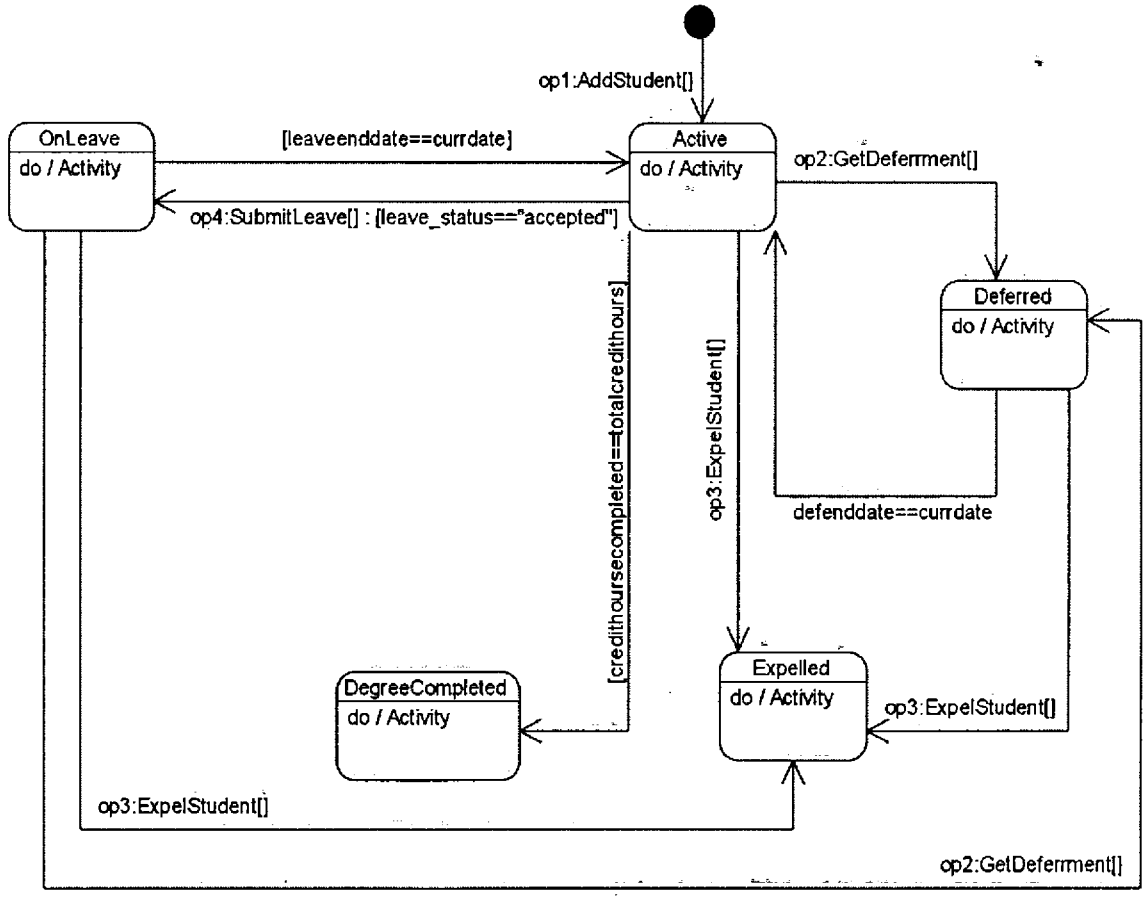


Figure C.16 State model of 'Student'

State Model of 'Enrollment'

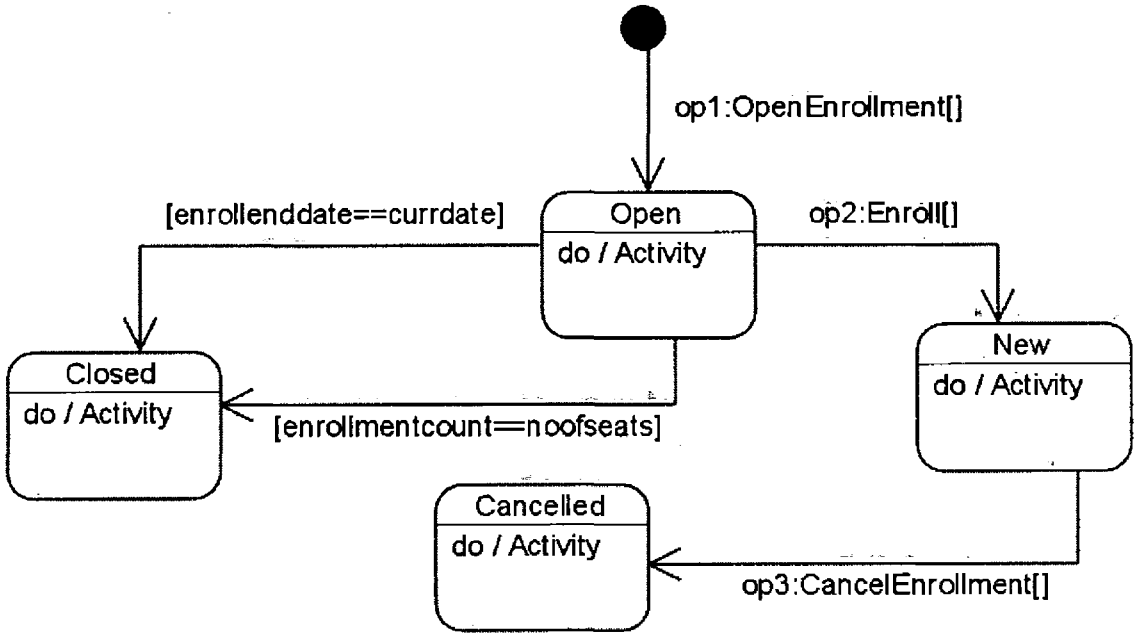


Figure C.17 State model of 'Enrollment'

State Model of 'Seminar'

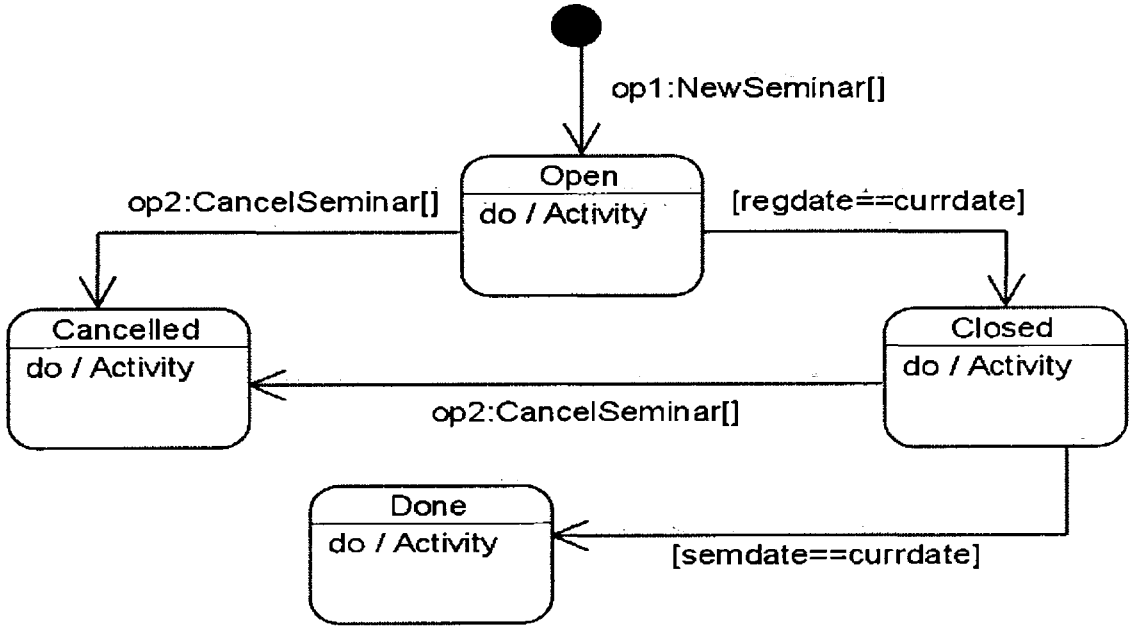


Figure C.18 State model of 'Seminar'

State Model of 'Professor'

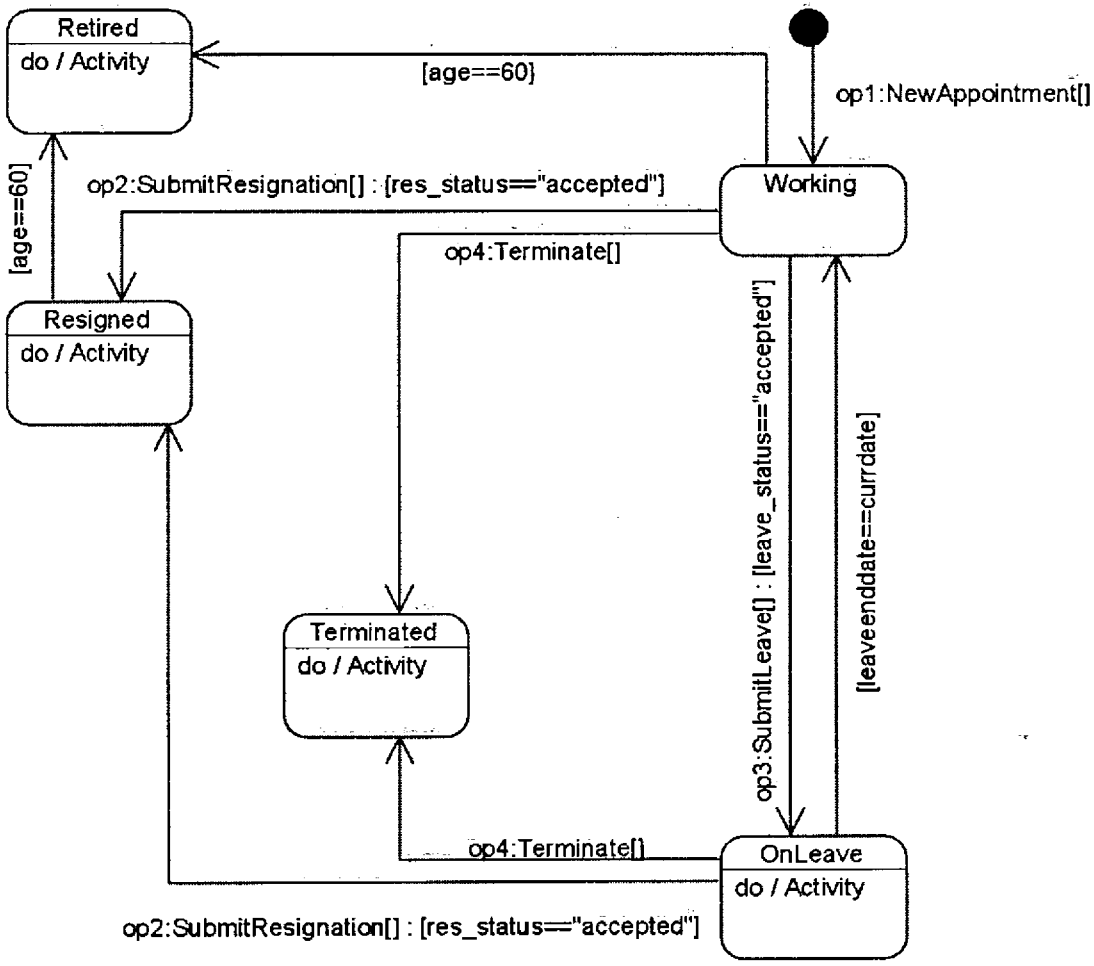


Figure C.19 State model of 'Professor'

C.5 Model 5 (M5)

This section illustrates the class and state diagrams for the 'Purchase Order' application.

C.5.1 Class Model

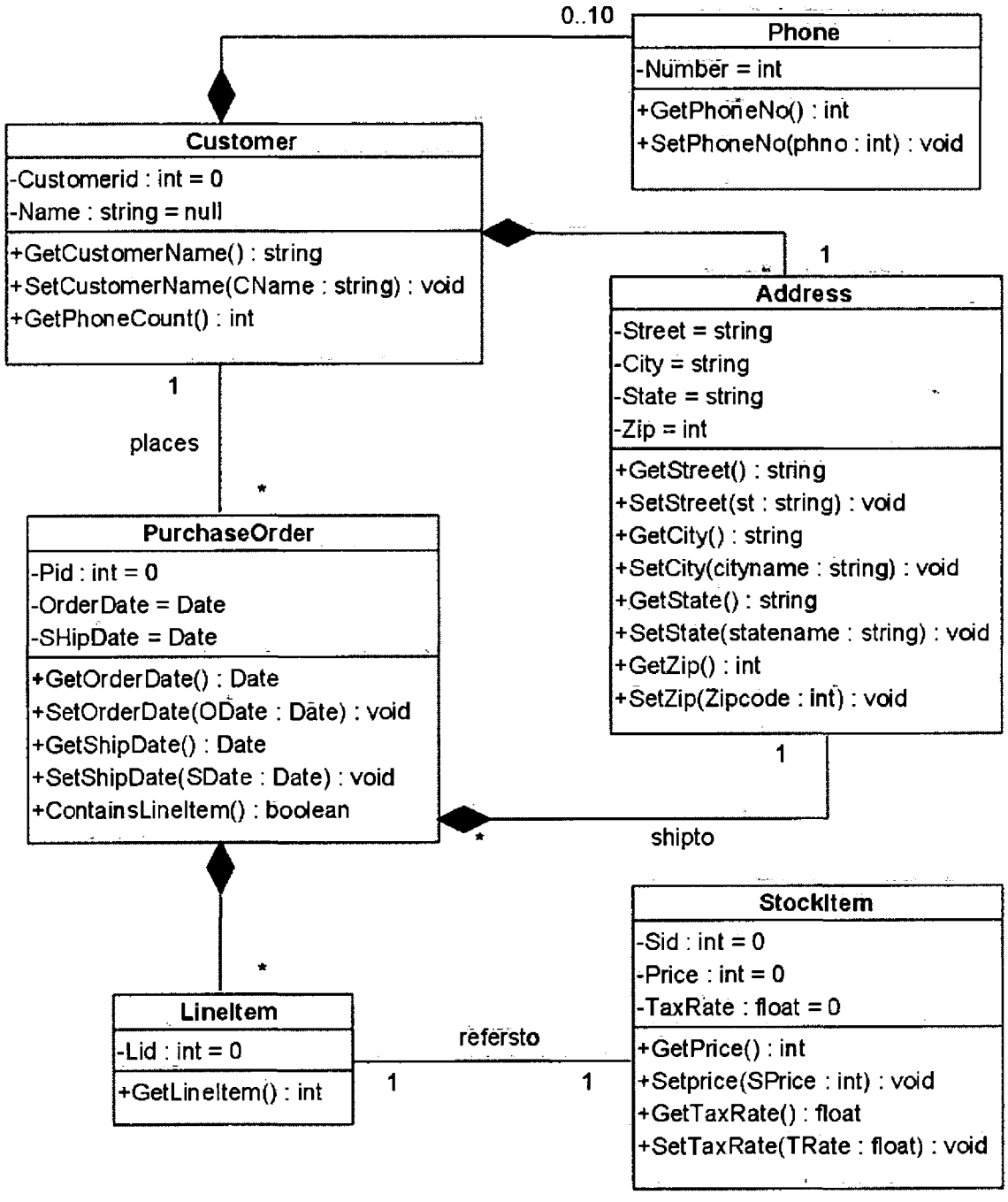


Figure C.20 Class model of 'Purchase Order' application

C.5.2 State Model

This section shows the state models corresponding to the classes in the class model of the Purchase Order application.

State Model of 'Customer'

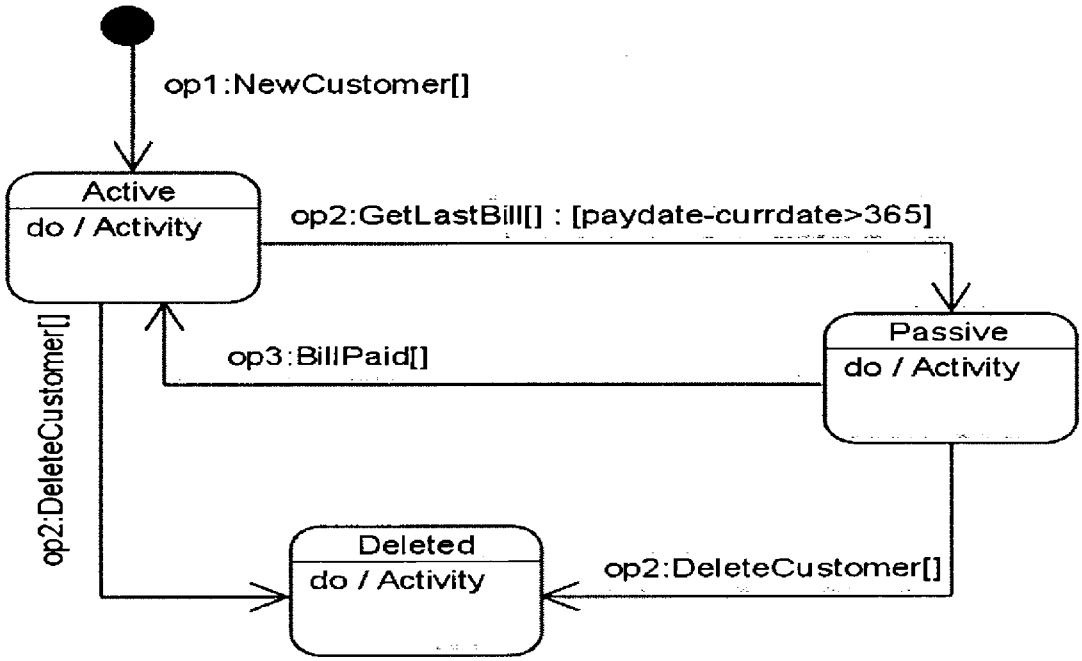


Figure C.21 State model of 'Customer'

State Model of 'Phone'

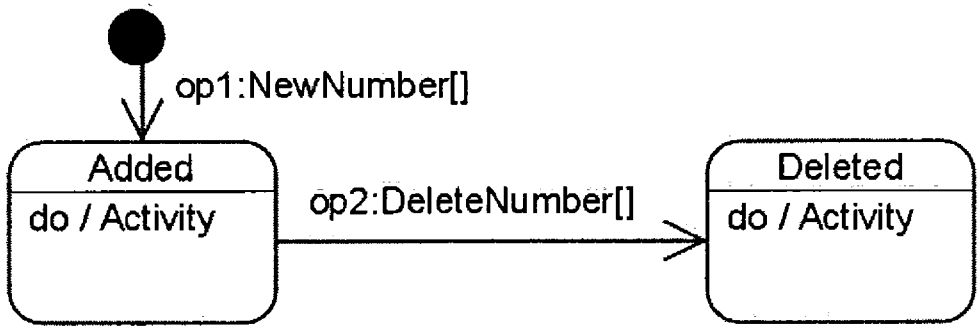


Figure C.22 State model of 'Phone'

State Model of 'PurchaseOrder'

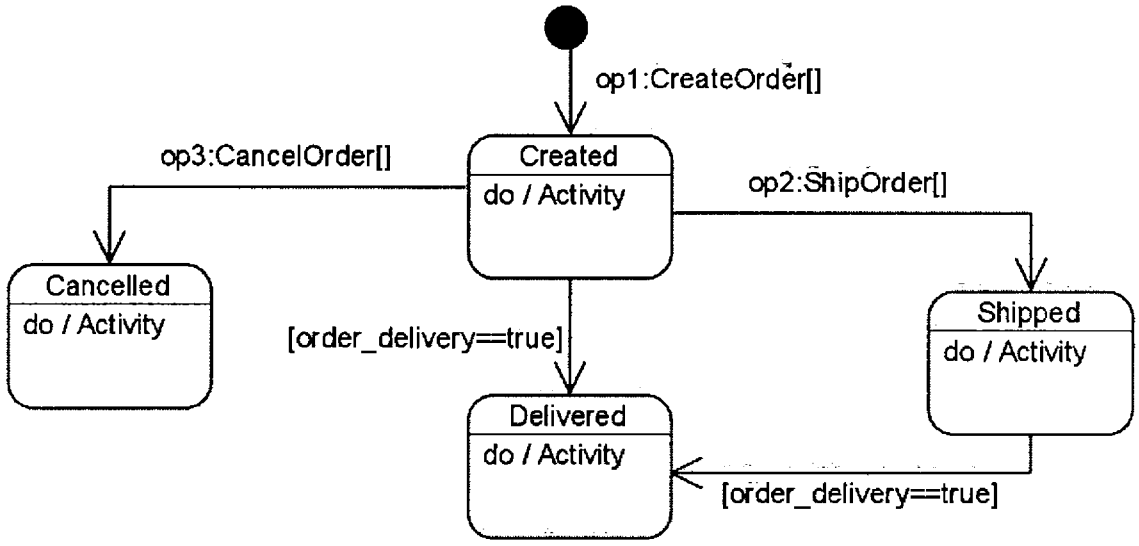


Figure C.23 State model of 'PurchaseOrder'

State Model of 'LineItem'

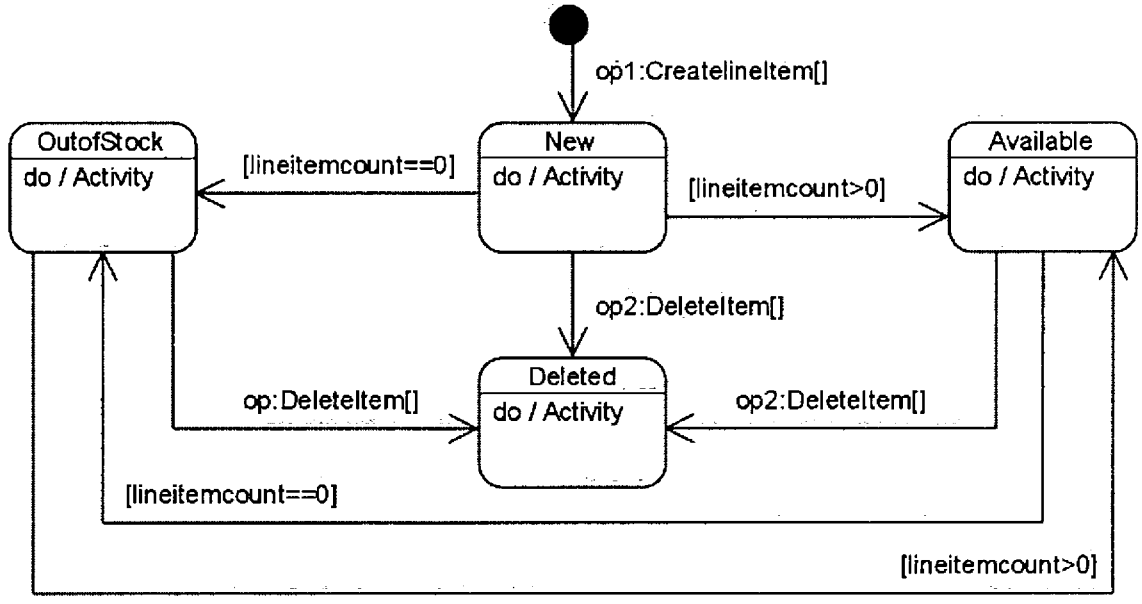


Figure C.24 State model of 'LineItem'

State Model of 'Address'

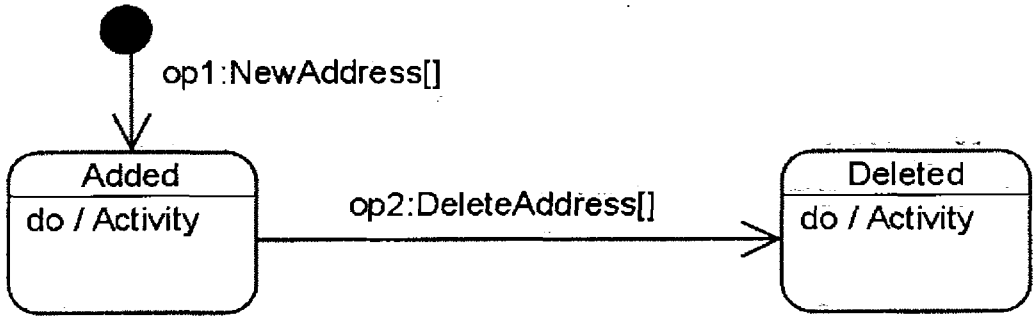


Figure C.25 State model of 'Address'

State Model of 'StockItem'

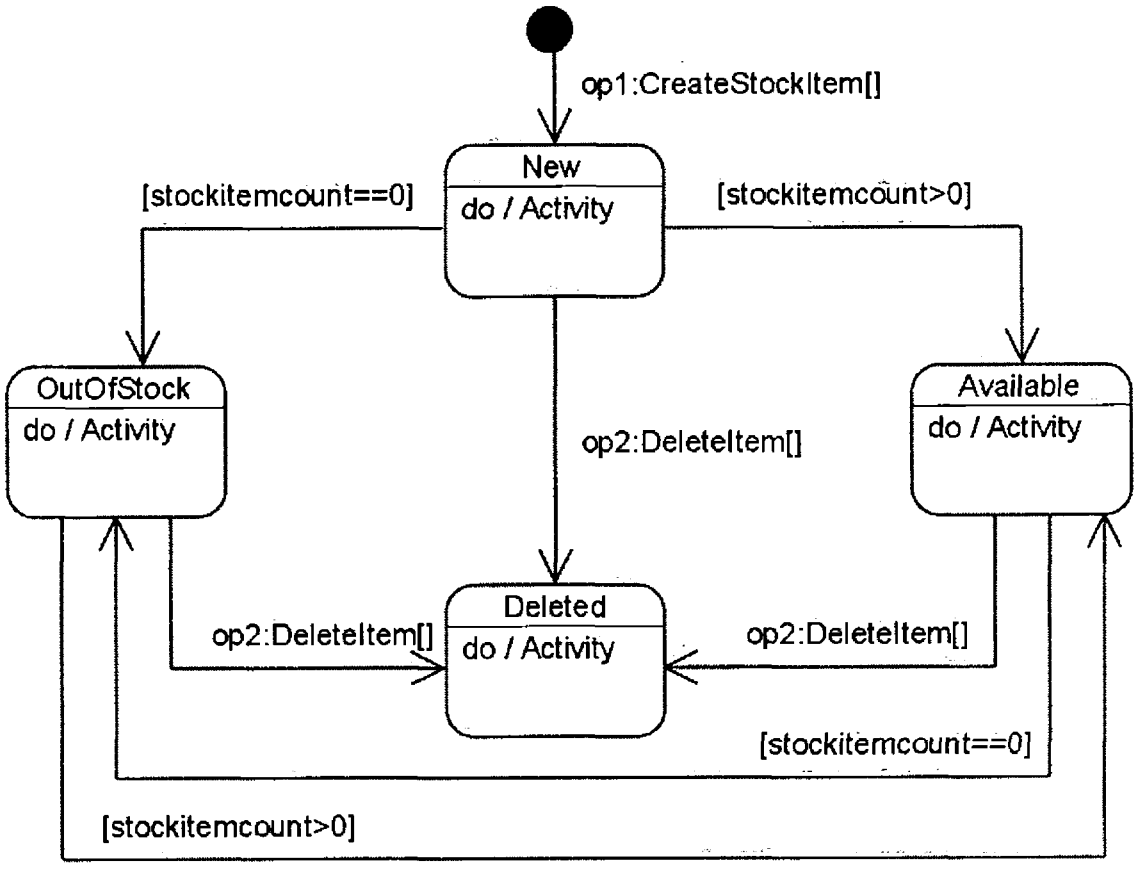


Figure C.26 State model of 'StockItem'

C.6 Model 6 (M6)

This section illustrates the class and state diagrams for the 'Library Management System'.

C.6.1 Class Model

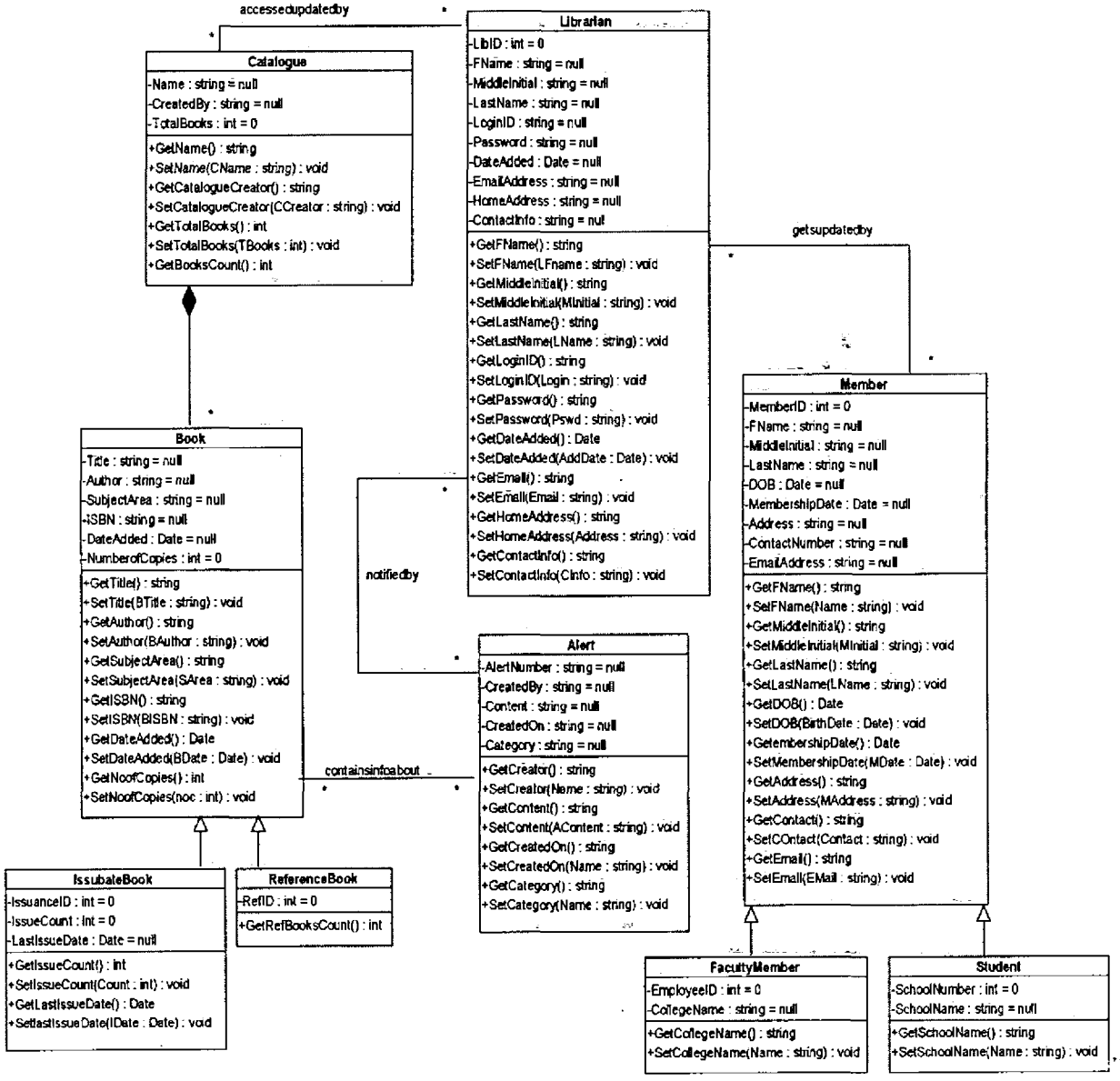


Figure C.27 Class model of 'Library Management System'

C.6.2 State Model

This section shows the state models corresponding to the classes in the class model of the Library Management System.

State Model of 'Catalogue'

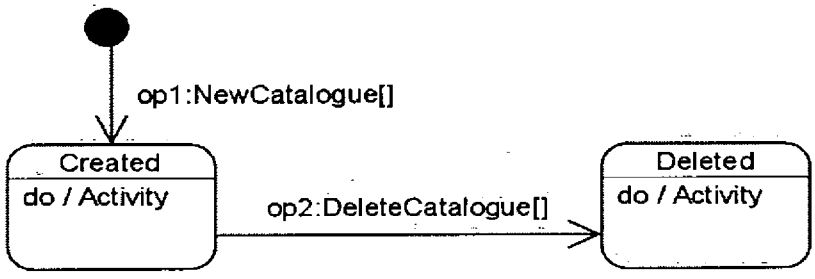


Figure C.28 State model of 'Catalogue'

State Model of 'Book'

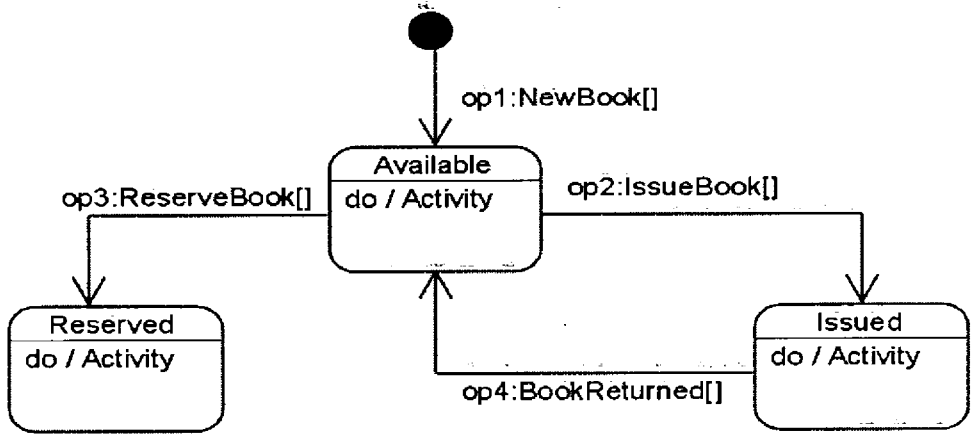


Figure C.29 State model of 'Book'

State Model of 'Alert'

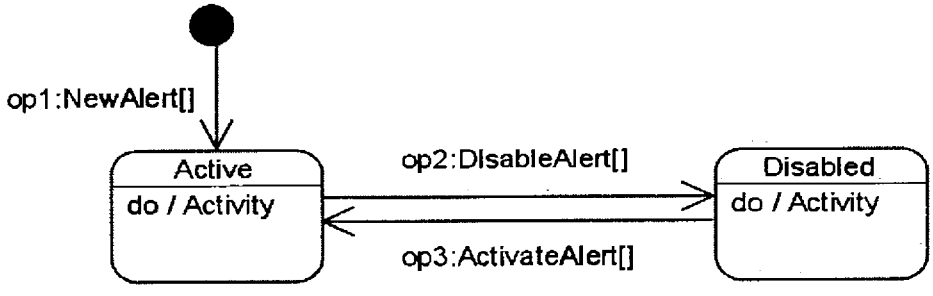


Figure C.30 State model of 'Alert'

State Model of 'Librarian'

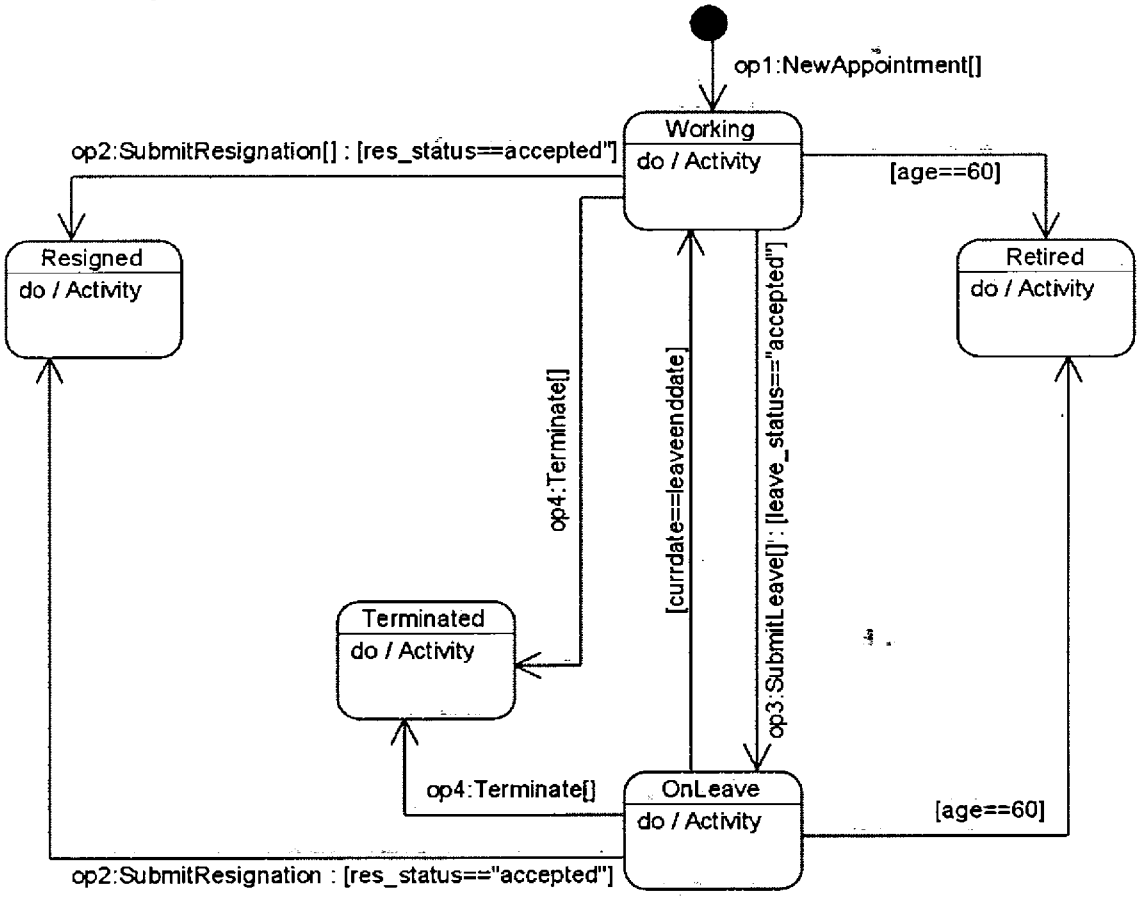


Figure C.31 State model of 'Librarian'

State Model of 'Member'

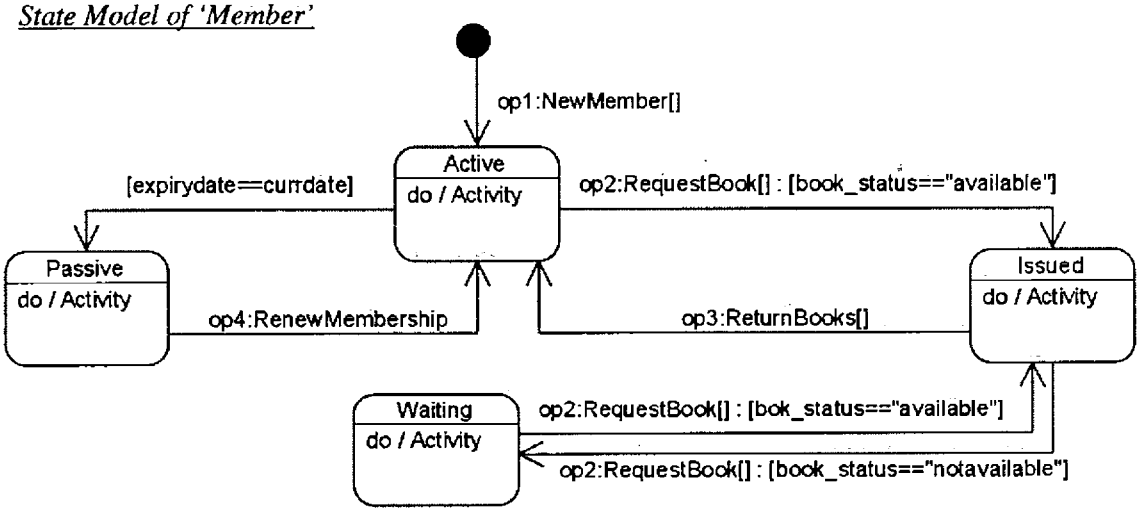


Figure C.32 State model of 'Member'

C.7 Model 7 (M7)

This section illustrates the class and state diagrams for the 'Online Shopping System'.

C.7.1 Class Model

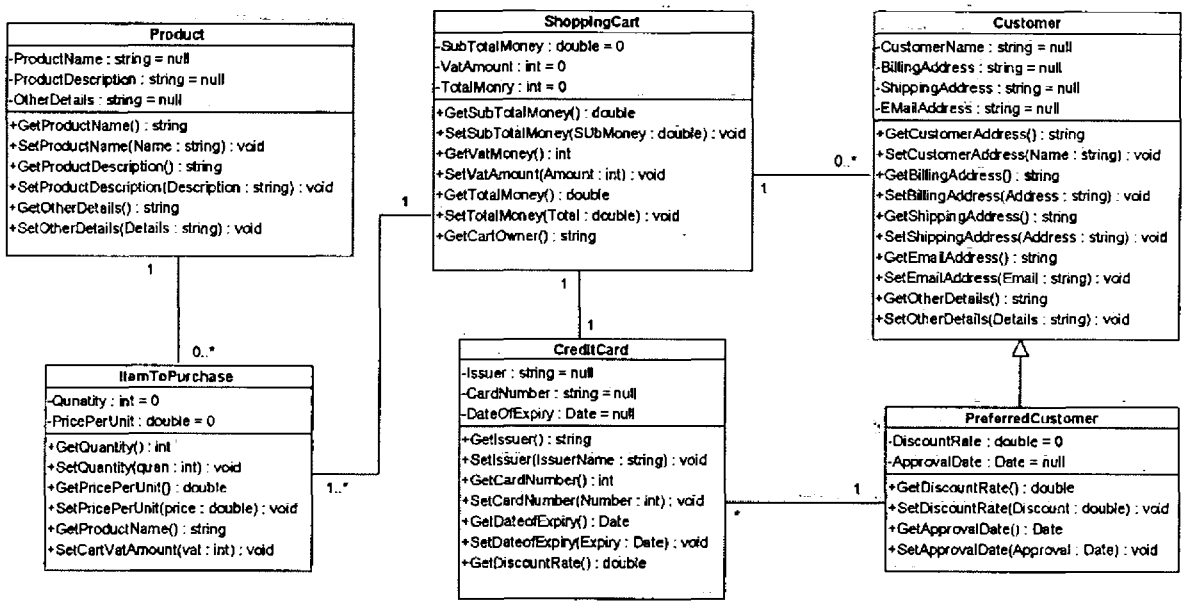


Figure C.33 Class model of 'Online Shopping System'

C.7.2 State Model

This section shows the state models for the classes of Online Shopping System.

State Model of 'ItemtoPurchase'

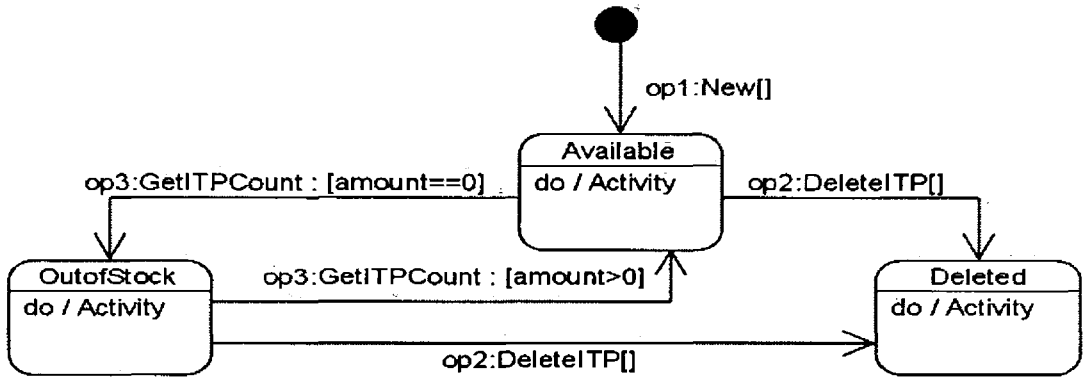


Figure C.34 State model of 'ItemtoPurchase'

State Model of 'ShoppingCart'

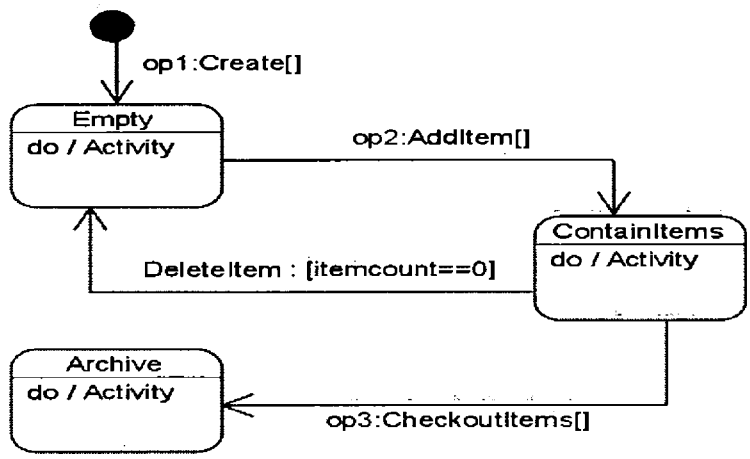


Figure C.35 State model of 'ShoppingCart'

State Model of 'Product'

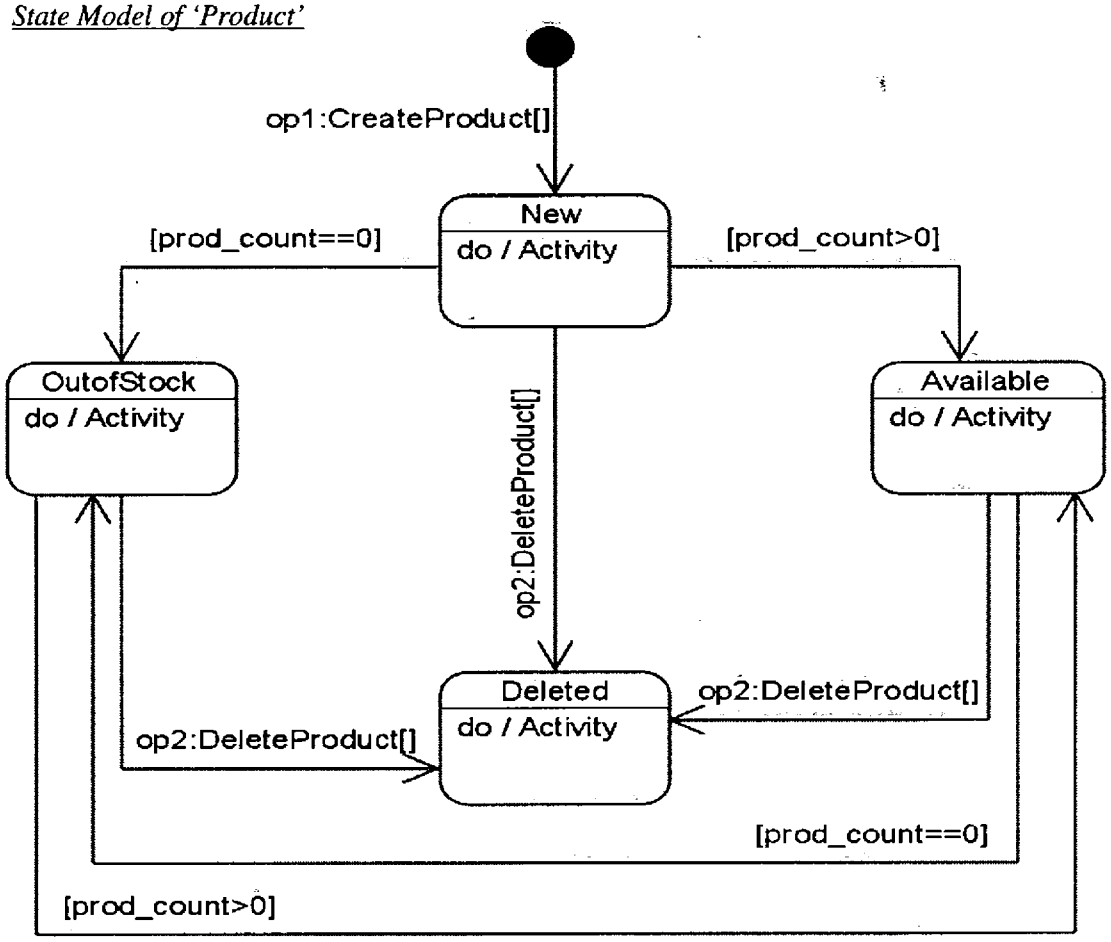


Figure C.36 State model of 'Product'

State Model of 'CreditCard'

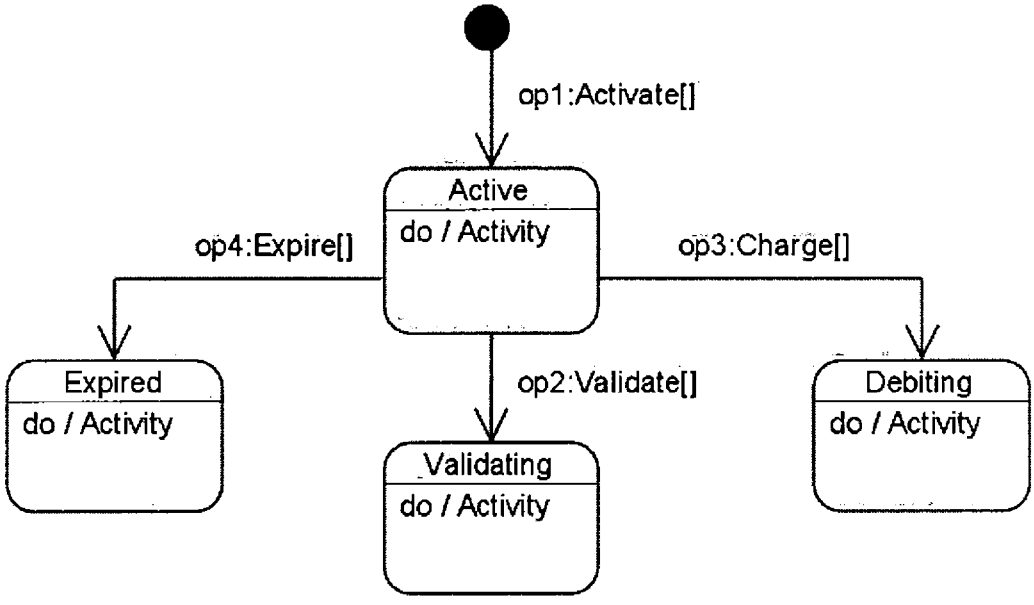


Figure C.37 State model of 'CreditCard'

State Model of 'Customer'

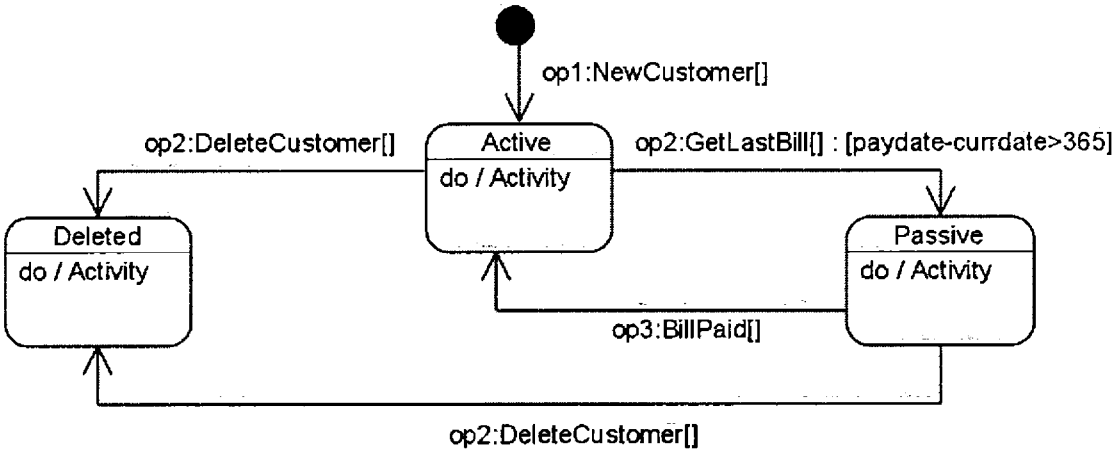


Figure C.38 State model of 'Customer'

C.8 Model 8 (M8)

This section presents the class and state diagrams for the classes in the 'Purchase Management System'.

C.8.1 Class Model

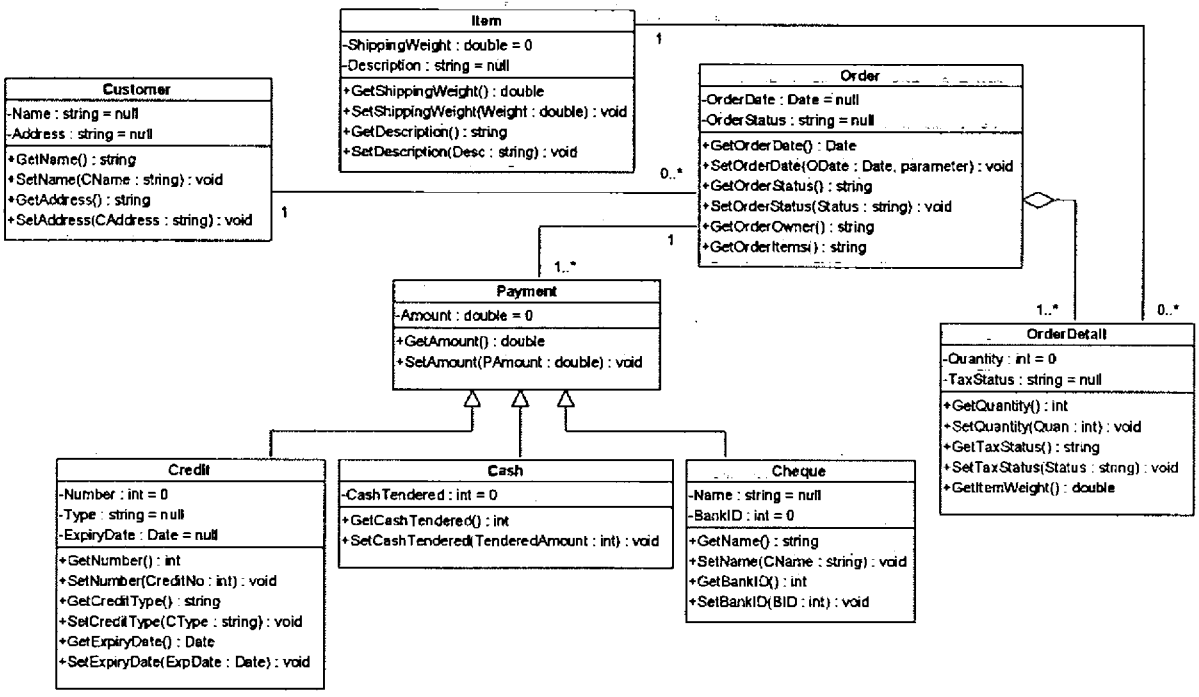


Figure C.39 Class model of 'Purchase Management System'

C.8.2 State Model

This section shows the state models for the classes of Purchase Management System.

State Model of 'Customer'

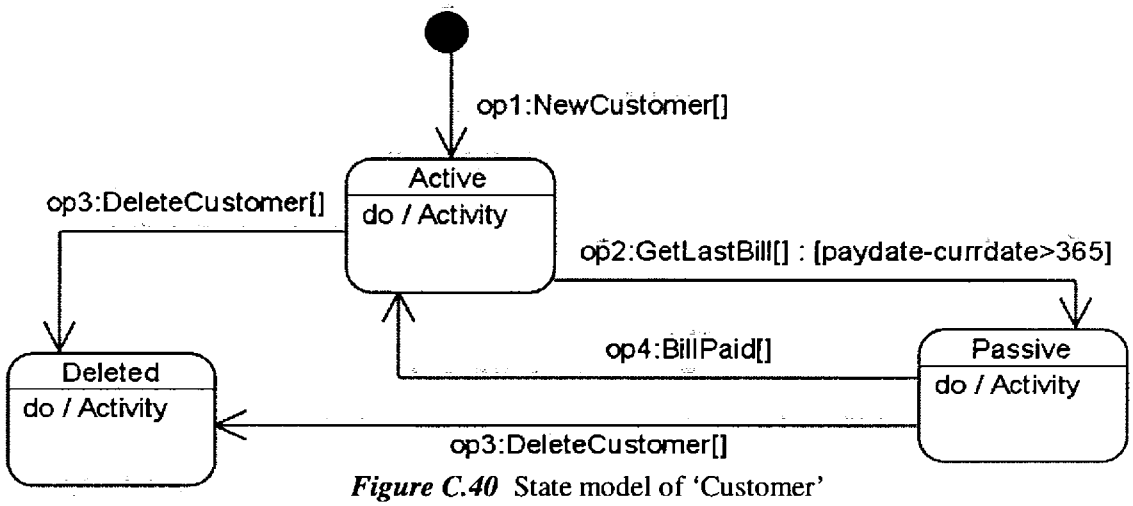


Figure C.40 State model of 'Customer'

State Model of 'Order'

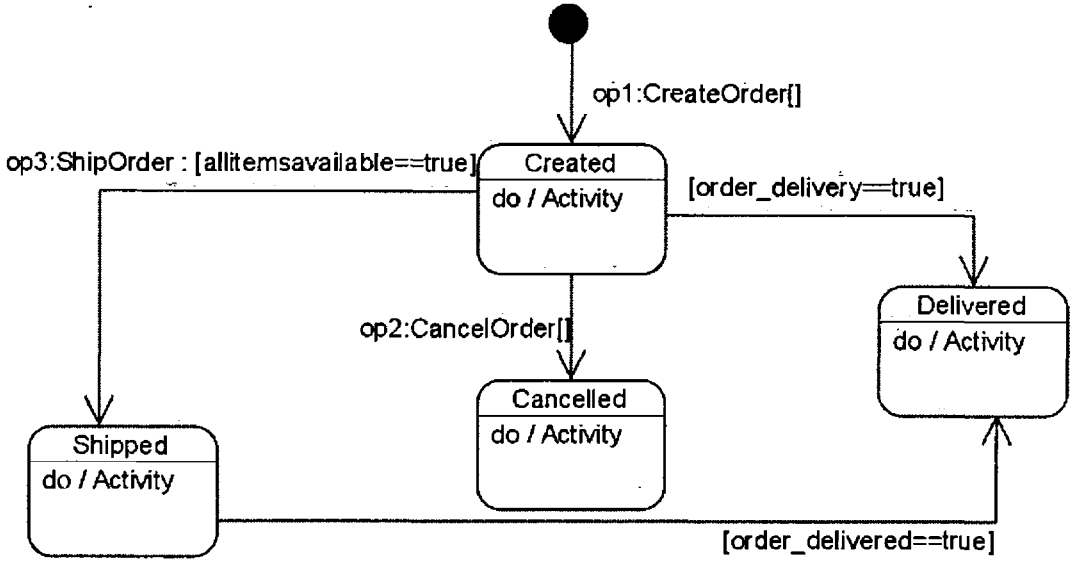


Figure C.41 State model of 'Order'

State Model of 'Payment'

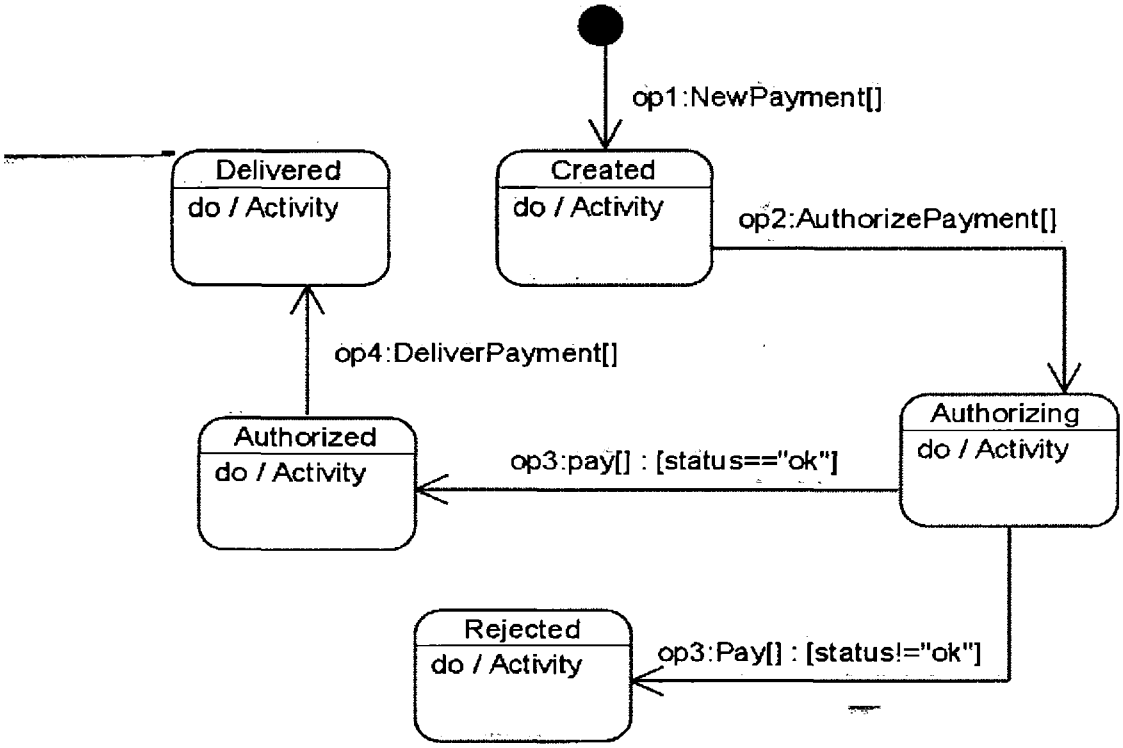


Figure C.42 State model of 'Payment'

State Model of 'Item'

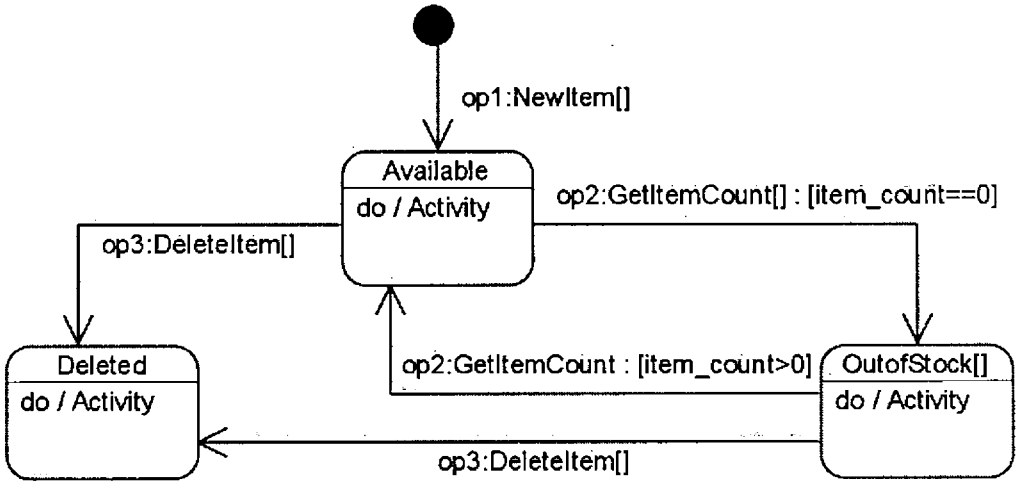


Figure C.43 State model of 'Item'

State Model of 'OrderDetail'

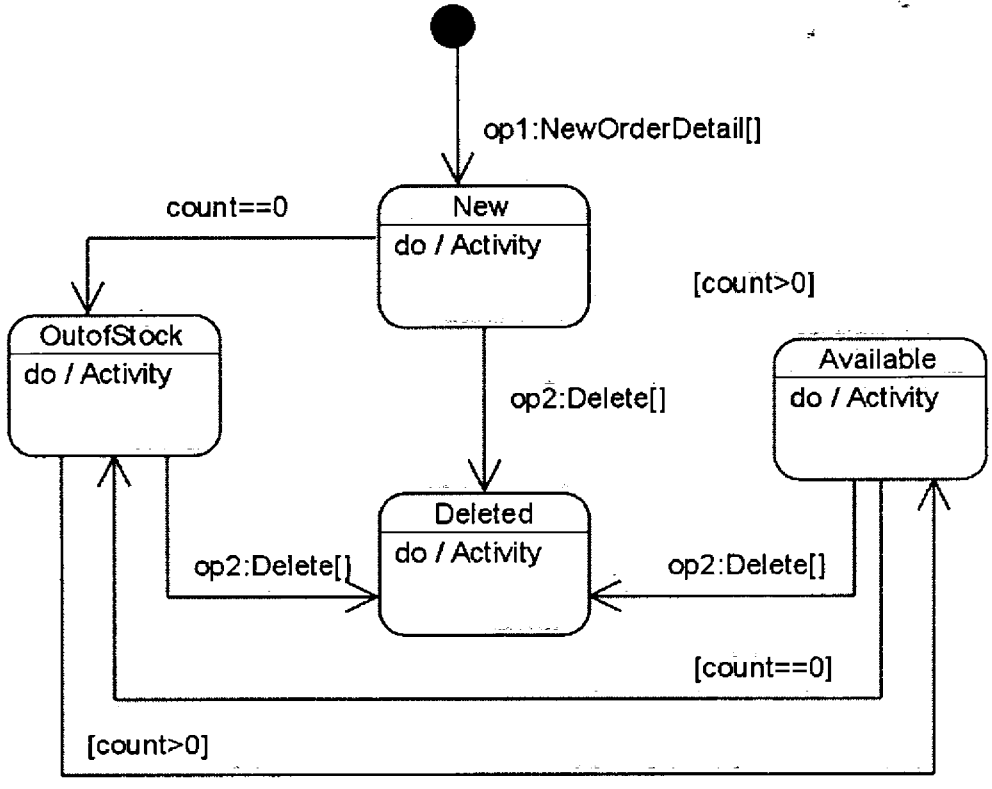


Figure C.44 State model of 'OrderDetail'

C.9 Model 9 (M9)

This section illustrates the class and state diagrams for the 'Drawing Application.

C.9.1 Class Model

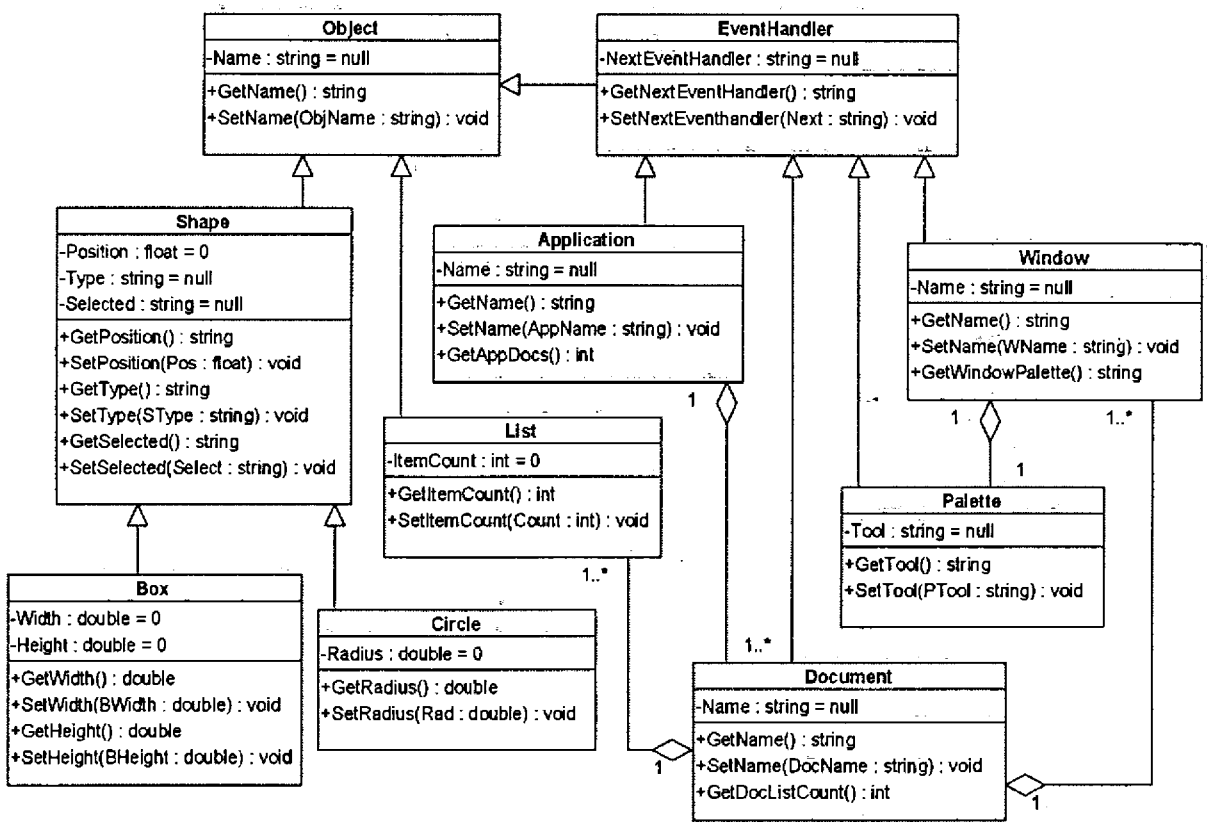


Figure C.45 Class model of 'Drawing Application'

C.9.2 State Model

This section shows the state models corresponding to the classes in the class model of the Drawing Application.

State Model of 'Object'

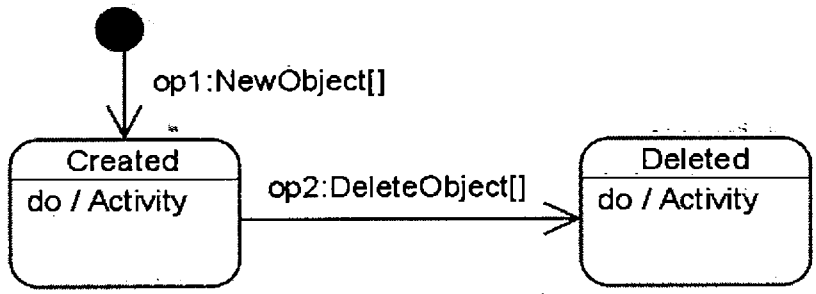


Figure C.46 State model of 'Object'

State Model of 'Shape'

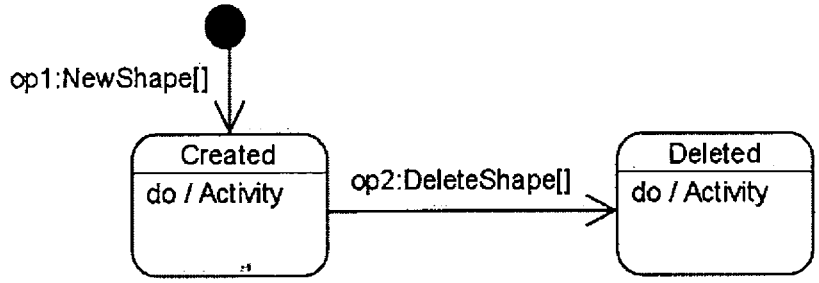


Figure C.47 State model of 'Shape'

State Model of 'List'

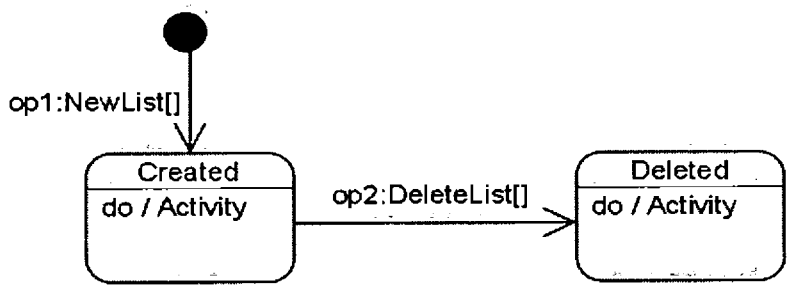


Figure C.48 State model of 'List'

State Model of 'EventHandler'

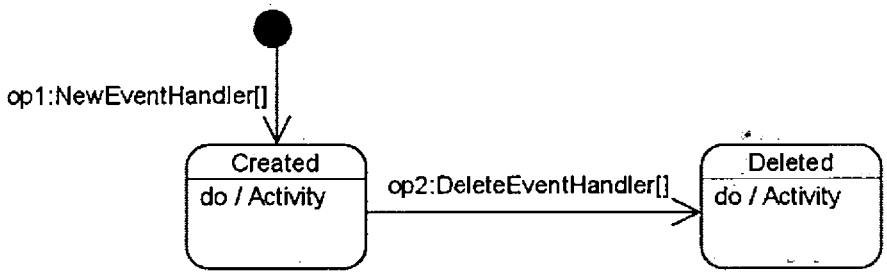


Figure C.49 State model of 'Eventhandler'

State Model of 'Application'

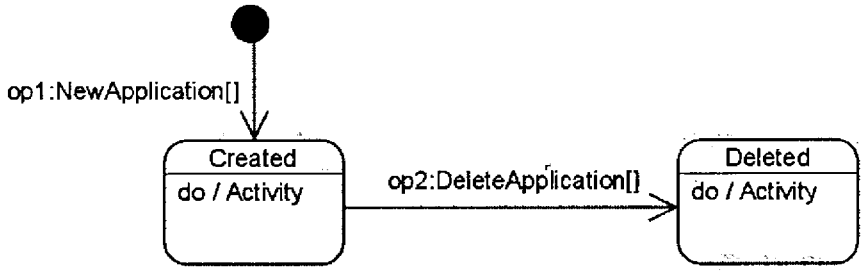


Figure C.50 State model of 'Application'

State Model of 'Box'

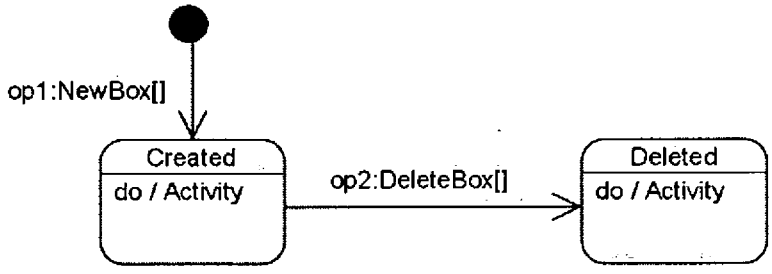


Figure C.51 State model of 'Box'

State Model of 'Circle'

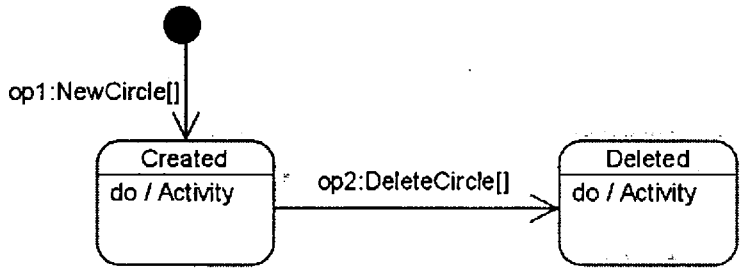


Figure C.52 State model of 'Circle'

State Model of 'Document'

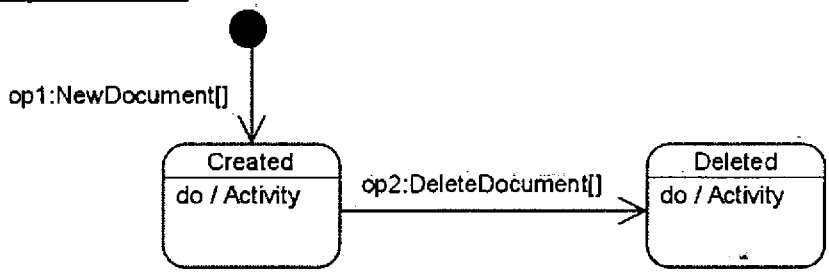


Figure C.53 State model of 'Document'

State Model of 'Palette'

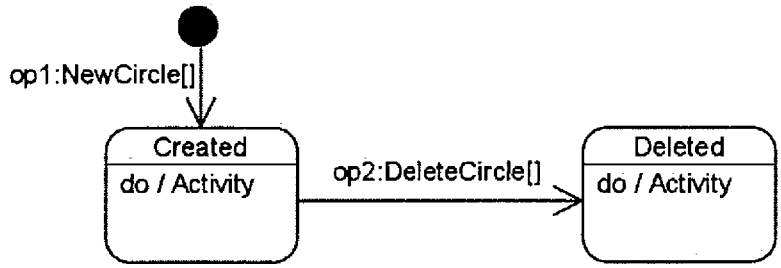


Figure C.54 State model of 'Palette'

State Model of 'Window'

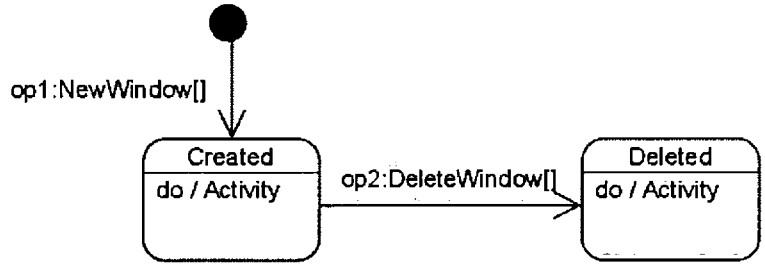


Figure C.55 State model of 'Window'

C.10 Model 10 (M10)

This section illustrates the class and state diagrams for the classes corresponding to the 'Account Management System'.

C.10.1 Class Model

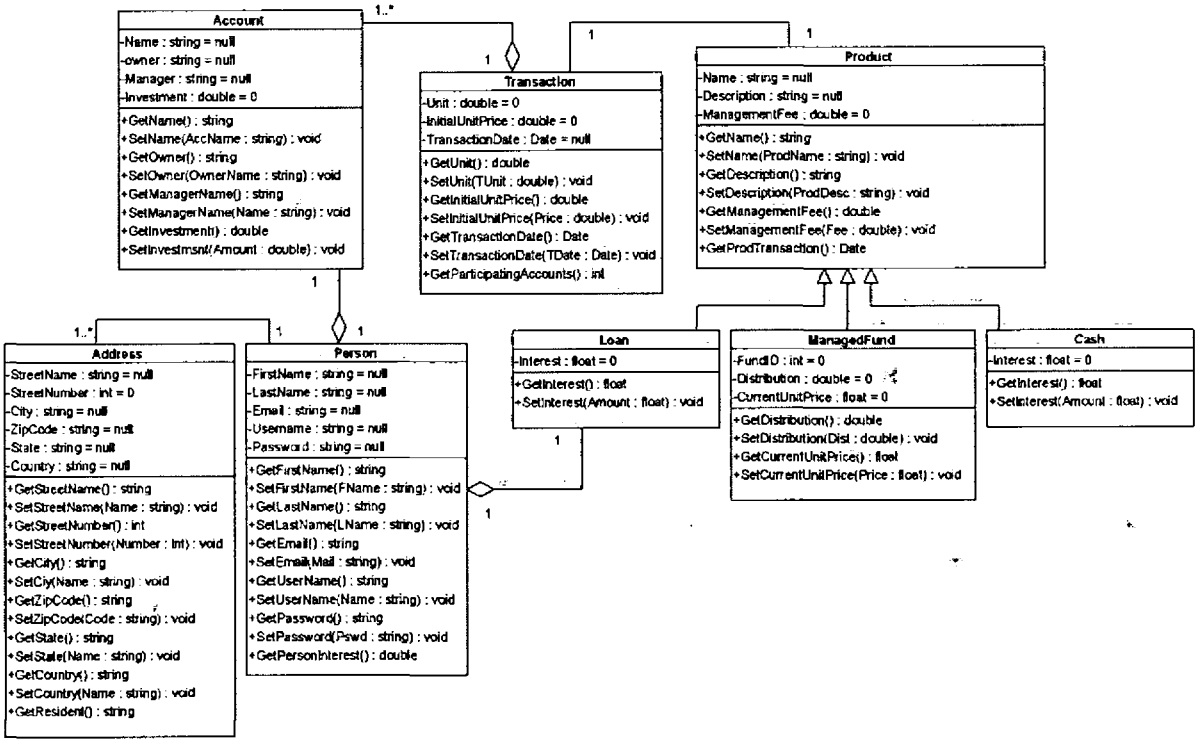


Figure C.56 Class model of 'Account Management System'

C.10.2 State Model

The state models corresponding to the classes of the Account Management System are illustrated in this section.

State Model of 'Product'

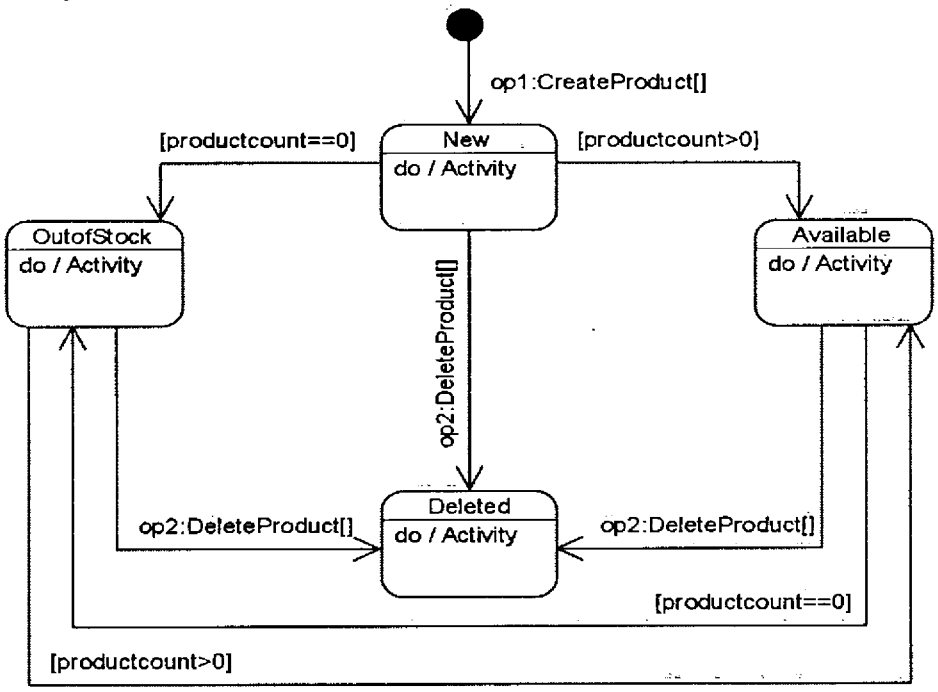


Figure C.57 State model of 'Product'

State Model of 'Cash'

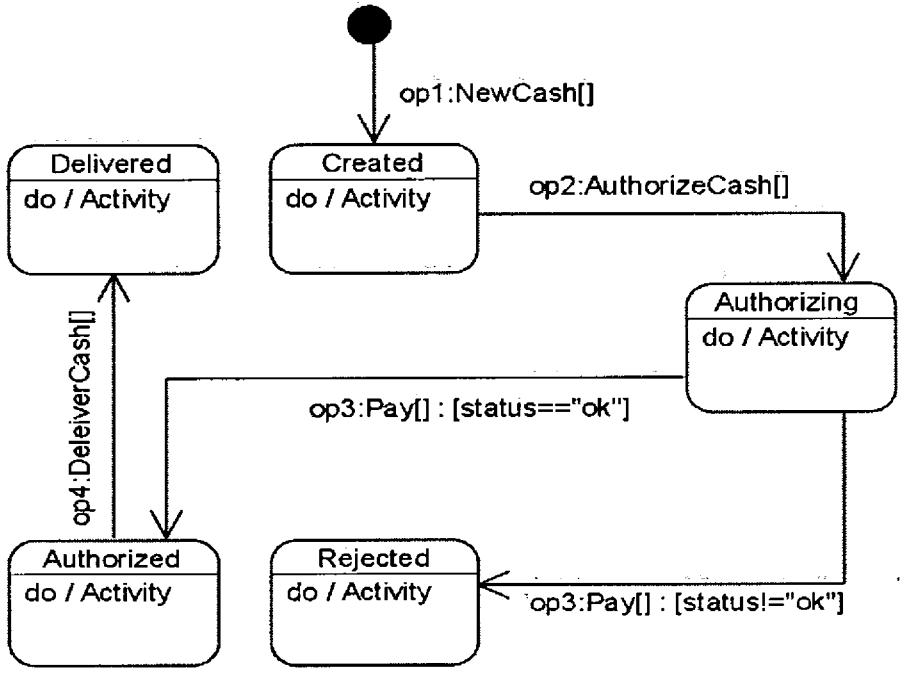


Figure C.58 State model of 'Cash'

State Model of 'ManagedFund'

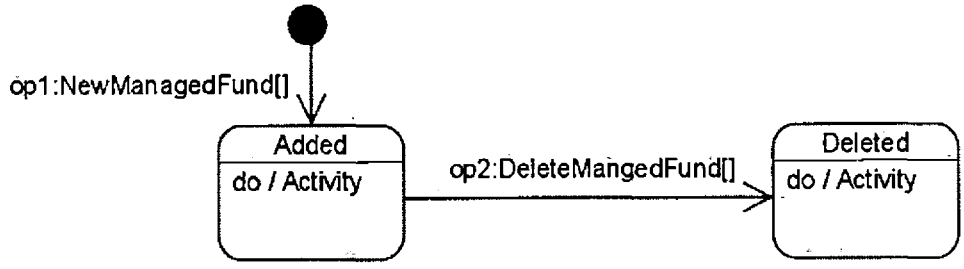


Figure C.59 State model of 'ManagedFund'

State Model of 'Loan'

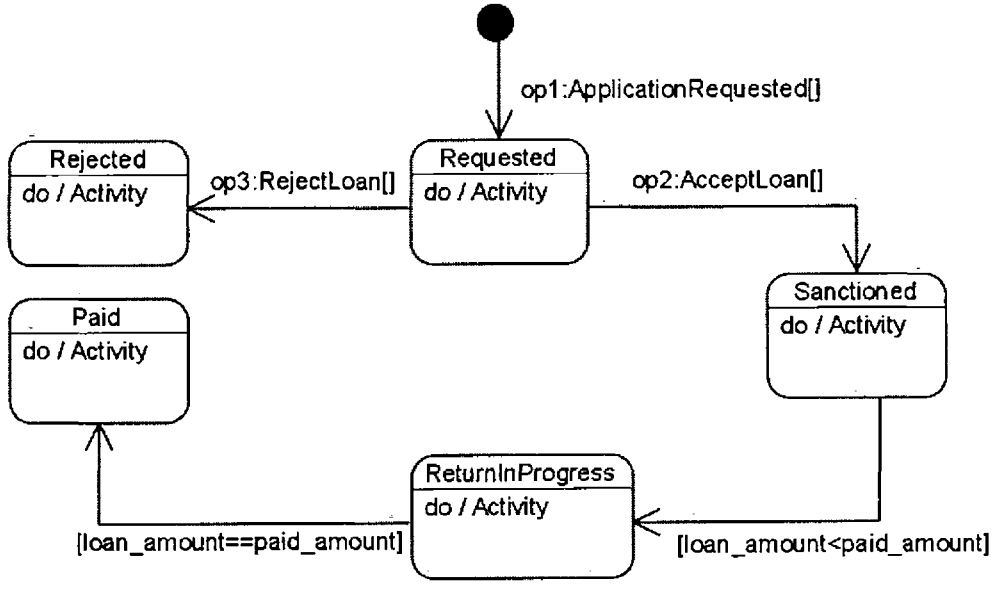


Figure C.60 State model of 'Loan'

State Model of 'Address'

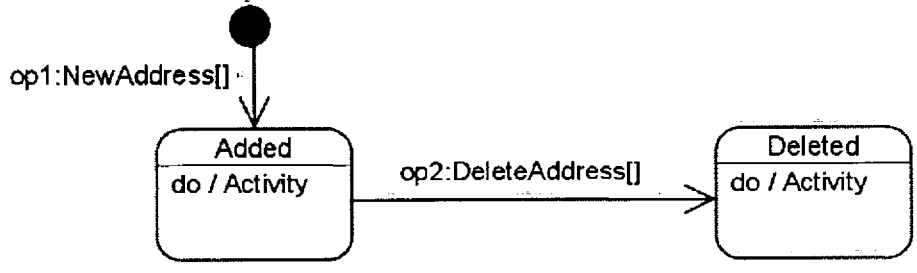


Figure C.61 State model of 'Address'

State Model of 'Person'

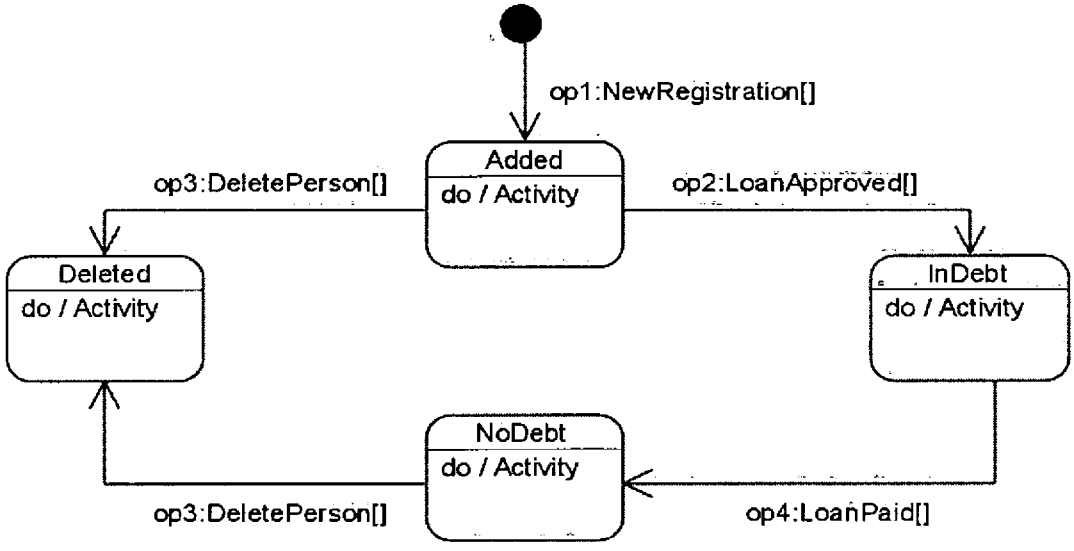


Figure C.62 State model of 'Person'

State Model of 'Transaction'

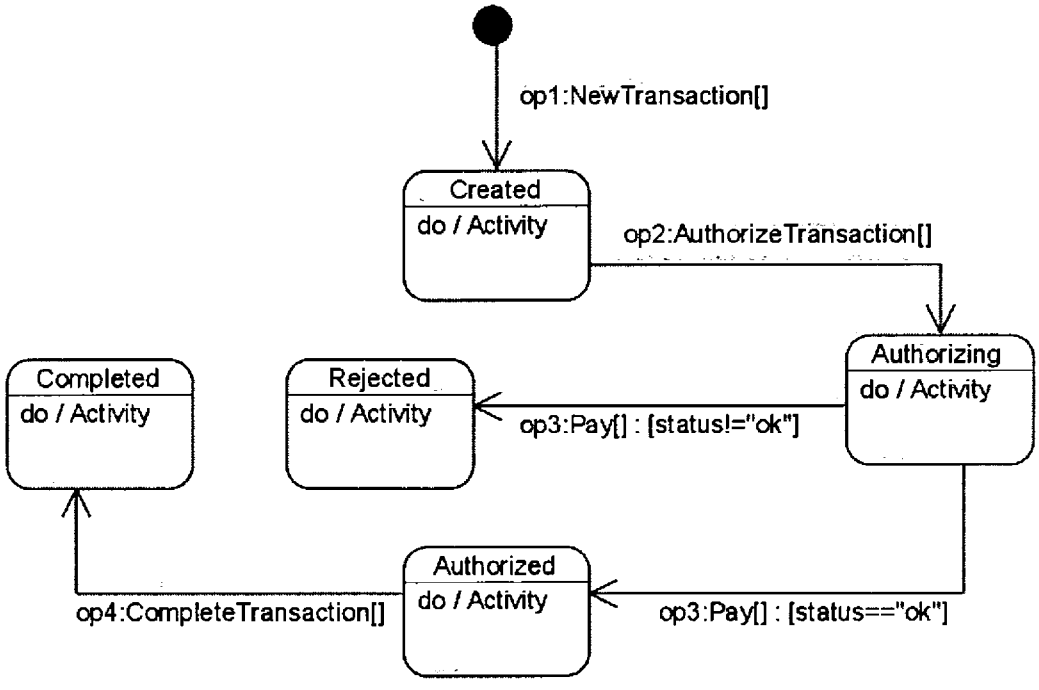


Figure C.63 State model of 'Transaction'

State Model of 'Account'

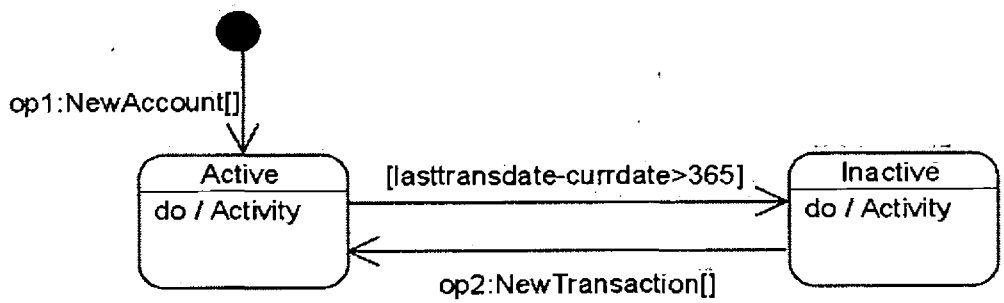


Figure C.64 State model of 'Account'

Appendix D

GENERATED CODE

D.1 Elevator Control System

This appendix is dedicated to illustrate the code generated by CØdŞ corresponding to the Elevator Control System (ECS).

Building.java

```
public class Building {
public Building(){
}
private String name = "IIUI";
private String address = "H-10 - Islamabad";
private int minfloors = 0;
private int maxfloors = 50;
private Floor floor = new Floor();
public void hallCallButtonPressed(int calling_floorno,int current_floorno)
{
floor[calling_floorno].hallButtonPress(calling_floorno, current_floorno);
}

public void hallCallButtonReleased(int calling_floorno)
{
floor[calling_floorno].hallButtonRelease();
}

public boolean validFloor(int floorno)
{
boolean valid = false;

if(floorno >= minfloors) {

if(floorno <= maxfloors) {
valid=true;
}
}
return valid;
}

public String getName()
{
return name;
}

public void setName(String bname)
{
name=bname;
}

public String getAddress()
{
return address;
}
```

```
public void setAddress(String baddress)
{
address=baddress;
}

public int getMinFloors()
{
return minfloors;
}

public void setMinFloors(int min_floor)
{
minfloors=min_floor;
}

public int getMaxFloors()
{
return maxfloors;
}

public void setMaxFloors(int max_floor)
{
maxfloors=max_floor;
}
}
```

Button.java

```
public class Button {
public Button(){
}
private String status = "idle";
public String getStatus()
{
return status;
}

public void setStatus(String stat)
{
status=stat;
}
}
```

CarButton.java

```
public class CarButton {
public CarButton(){
}
private String button_status = "idle";
private String light_status = "off";
private CarButtonState cb_state = new CarButtonState();
public void illuminate()
{
cb_state.illuminate();
```

```
}

public void turnOff()
{
cb_state.turnOff();
}

public String getButtonStatus()
{
return button_status;
}

public void setButtonStatus(String stat)
{
button_status=stat;
}

public String getLightStatus()
{
return light_status;
}

public void setLightStatus(String lstatus)
{
light_status=lstatus;
}

public CarButtonState getCarButtonState()
{
return cb_state;
}

public void setCarButtonState(CarButtonState cbstate)
{
cb_state=cbstate;
}
}
```

CarButtonoff.java

```
public class CarButtonOff {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setButtonStatus("idle");
this.setLightStatus("off");
}
public void illuminate()
```

```
{
this.setCarButtonState(cb_on);
getCarButtonState().Entry();
}
}
```

CarButtonOn.java

```
public class CarButtonOn {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setButtonStatus("pressed");
this.setLightStatus("on");
}

public void turnOff()
{
this.setCarButtonState(cb_off);
getCarButtonState().Entry();
}
}
```

CarButtonState.java

```
public class CarButtonState {
private CarButtonOn cb_on = new CarButtonOn();
private CarButtonOff cb_off = new CarButtonOff();
public void Entry()
{
}

public void Exit()
{
}

public void doActivity()
{
}

public void illuminate()
{
}

public void turnOff()
{
```

```
}
}
```

CarLantern.java

```
public class CarLantern {
public CarLantern(){
}
private String lantern_status = "off";
private CarLanternState cl_state = new CarLanternState();
public void illuminate()
{
cl_state.illuminate();
}

public void turnOff()
{
cl_state.turnOff();
}

public String getStatus()
{
return lantern_status;
}

public void setStatus(String status)
{
lantern_status=status;
}

public CarLanternState getLanternState()
{
return cl_state;
}

public void setLanternState(CarLanternState clstate)
{
cl_state=clstate;
}
}
```

CarLanternOff.java

```
public class CarLanternOff {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
```

```
{
this.setStatus("off");
}

public void illuminate()
{
this.setLanternState(cl_on);
getLanternState().Entry();
}
}
```

CarLanternOn.java

```
public class CarLanternOn {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setStatus("on");
}

public void turnOff()
{
this.setLanternState(cl_off);
getLanternState().Entry();
}
}
```

CarLanternState.java

```
public class CarLanternState {
private CarLanternOn cl_on = new CarLanternOn();
private CarLanternOff cl_off = new CarLanternOff();
public void Entry()
{
}

public void Exit()
{
}

public void doActivity()
{
}

public void illuminate()
{
```

```
}

public void turnOff()
{
}
}
```

CarPositionIndicator.java

```
public class CarPositionIndicator {
public CarPositionIndicator(){
}
private String position_status = "passive";
private int desired_floor = 0;
private CPIState cpi_state = new CPIState();
public void show(int floorno)
{
cpi_state.show(floorno);
}

public void clear()
{
cpi_state.clear();
}

public String getStatus()
{
return position_status;
}

public void setStatus(String status)
{
position_status=status;
}

public int getDesiredFloor()
{
return desired_floor;
}

public void setDesiredFloor(int dfloor)
{
desired_floor=dfloor;
}

public CPIState getCPState()
{
return cpi_state;
}

public void setCPState(CPIState cpstate)
{
cpi_state=cpstate;
}
}
```

CPIActive.java

```
public class CPIActive {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setStatus("active");
}

public void clear()
{
this.setCPState(cpi_passive);
getCPState().Entry();
}
}
```

CPIPassive.java

```
public class CPIPassive {
public void Entry()
{
this.setDesiredFloor(-1);
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setStatus("passive");
}

public void show(int floorno)
{
this.setDesiredFloor(floorno);
this.setCPState(cpi_active);
getCPState().Entry();
}
}
```

CPIState.java

```
public class CPIState {
private CPIActive cpi_active = new CPIActive();
```

```
private CPIPassive cpi_passive = new CPIPassive();
public void Entry()
{
}

public void Exit()
{
}

public void doActivity()
{
}

public void show(int floorno)
{
}

public void clear()
{
}
}
```

Dispatcher.java

```
public class Dispatcher {
public Dispatcher(){
}
private int max = 1000;
private int destinationQueue[] = new int[max];
private int front = 0;
private int rear = 1;
private int current_floor = 0;
public int getCurrentFloor()
{
return current_floor;
}

public void setCurrentFloor(int floor_no)
{
current_floor=floor_no;
}

public void enque(int floorno)
{

if(front != rear) {
destinationQueue[rear]=floorNo;
rear=rear+1;

if(rear == max) {
rear=0;
}
}
}

public int deque()
```

```
{
int temp = -1;

if(front == rear-1) {
return temp;
}

if(front == max-1) {

if(rear == 0) {
return temp;
}
}
front=front+1;

if(front == max) {
front=0;
}
temp=destinationQueue[front];
return temp;
}

public boolean isEmpty()
{
boolean empty = false;

if(front == rear-1) {
empty=true;
}

if(front == max-1) {

if(rear == 0) {
empty=true;
}
}
return empty;
}

public void atFloor(int source_floor,int destination_floor,String direction)
{
current_front=source_floor;

if(direction == "down") {
this.decrementFloor();
}

if(current_floor > destination_floor) {
}

if(current_floor > destination_floor) {
this.decrementFloor();
}

if(direction == "up") {
this.incrementFloor();
for(int i=0; i<50; i++){
```

```
if(current_floor < destination_floor) {
}
}
}
}

public void decrementFloor()
{
current_floor=current_floor-1;
}

public void incrementFloor()
{
current_floor=current_floor+1;
}
}
```

Door.java

```
public class Door {
public Door(){
}
private String door_status = "closed";
private int door_reversal = 0;
private DoorState doorstate = new DoorState();
public void close()
{
doorstate.close();
}

public void open()
{
doorstate.open();
}

public void reverseDoors()
{
door_reversal=1;
doorstate.reverseDoors();
}

public String getDoorStatus()
{
return door_status;
}

public void setDoorStatus(String status)
{
door_status=status;
}

public int getReversalStatus()
{
return door_reversal;
}
```

```
public void setReversalStatus(String reverse)
{
door_reversal=reverse;

if(door_reversal == 1) {
doorstate.reverseDoors();
}
}

public DoorState getDoorState()
{
return doorstate;
}

public void setDoorState(DoorState d_state)
{
doorstate=d_state;
}
}
```

DoorClosed.java

```
public class DoorClosed {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setDoorStatus("closed");
}

public void close()
{
this.Entry();
}

public void open()
{
this.setDoorState(doorOpening);
getDoorState().Entry();
}
}
```

DoorClosing.java

```
public class DoorClosing {
public void Entry()
```

```
{
this.setDoorStatus("closing");
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
int r_status = 0;
r_status = this.getReversalStatus();

if(r_status == 0) {
this.setDoorState(doorClosed);
getDoorState().Entry();
}

if(r_status == 1) {
this.setReversalStatus(0);
this.setDoorState(doorOpening);
getDoorState().Entry();
}
}

public void reverseDoors()
{
this.setReversalStatus(0);
this.setDoorState(doorOpening);
getDoorState().Entry();
}
}
```

DoorOpened.java

```
public class DoorOpened {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setDoorStatus("opened");
}

public void open()
{
this.Entry();
}
```

```
public void close()
{
this.setDoorState(doorClosing);
getDoorState().Entry();
}
}
```

DoorOpening.java

```
public class DoorOpening {
public void Entry()
{
this.setDoorStatus("opening");
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
int r_status = 0;
r_status = this.getReversalStatus();

if(r_status == 0) {
this.setDoorState(doorOpened);
getDoorState().Entry();
}

if(r_status == 1) {
this.setReversalStatus(0);
this.setDoorState(doorClosing);
getDoorState().Entry();
}
}

public void reverseDoors()
{
this.setReversalStatus(0);
this.setDoorState(doorClosing);
getDoorState().Entry();
}
}
```

DoorState.java

```
public class DoorState {
private DoorOpened doorOpened = new DoorOpened();
private DoorClosed doorClosed = new DoorClosed();
private DoorOpening doorOpening = new DoorOpening();
private DoorClosing doorClosing = new DoorClosing();
public void Entry()
{
```

```
}
public void Exit()
{
}

public void doActivity()
{
}

public void close()
{
}

public void open()
{
}

public void reverseDoors()
{
}
}
```

Drive.java

```
public class Drive {
public Drive(){
}
private String elevator_status = "stopped";
private String elevator_direction = null;
private int speed = 0;
private int drive_sfloor = 0;
private int drive_dfloor = 0;
private DriveState drive_state = new DriveState();
public void moveUp(int source_floor,int destination_floor)
{
drive_sfloor=source_floor;
drive_dfloor=destination_floor;
drive_state.moveUpSlow();
}

public void moveDown(int source_floor,int destination_floor)
{
drive_sfloor=source_floor;
drive_dfloor=destination_floor;
drive_state.moveDownSlow();
}

public void stop()
{
drive_state.stop();
}

public String getStatus()
{
return elevator_status;
}
```

```
public void setStatus(String estatus)
{
elevator_status=estatus;
}

public String getDirection()
{
return elevator_direction;
}

public void setDirection(String edirection)
{
elevator_direction=edirection;
}

public int getSpeed()
{
return speed;
}

public void setSpeed(int espeed)
{
speed=espeed;
}

public DriveState getDriveState()
{
return drive_state;
}

public void setDriveState(DriveState dstate)
{
drive_state=dstate;
}

public int getSourceFloor()
{
return drive_sfloor;
}

public void setSourceFloor(int s_floor)
{
drive_sfloor=s_floor;
}

public int getDestinationFloor()
{
return drive_dfloor;
}

public void setDestinationFloor(int d_floor)
{
drive_dfloor=d_floor;
}
}
```

DriveControl.java

```
public class DriveControl {
public DriveControl(){
}
private int control = 0;
private Drive drive = new Drive();
public void moveUp(int source_floor,int destination_floor)
{
drive.moveUp(source_floor, destination_floor);
}

public void moveDown(int source_floor,int destination_floor)
{
drive.moveDown(source_floor, destination_floor);
}

public void stop()
{
drive.stop();
}
}
```

DriveMoveDownFast.java

```
public class DriveMoveDownFast {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setSpeed(2);
}

public void stop()
{
this.setDriveState(d_stopped);
getDriveState().Entry();
}
}
```

DriveMoveDownSlow.java

```
public class DriveMoveDownSlow {
public void Entry()
{
this.setDirection("down");
this.doActivity();
```

```
}

public void Exit()
{
}

public void doActivity()
{
this.setStatus("moving");
this.setSpeed(1);
this.moveDownFast();
}

public void moveDownFast()
{
int sfloor = 0;
int dfloor = 0;
sfloor = this.getSourceFloor();
dfloor = this.getDestinationFloor();

if(dfloor < sfloor-2) {
this.setDriveState(dmove_downfast);
getDriveState().Entry();
}
}

public void stop()
{
this.setDriveState(d_stopped);
getDriveState().Entry();
}
}
```

DriveMoveUpFast.java

```
public class DriveMoveUpFast {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setSpeed(2);
}

public void stop()
{
this.setDriveState(d_stopped);
getDriveState().Entry();
}
}
```

DriveMoveUpSlow.java

```
public class DriveMoveUpSlow {
public void Entry()
{
this.setDirection("up");
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setStatus("moving");
this.setSpeed(1);
this.moveUpFast();
}

public void moveUpFast()
{
int sfloor = 0;
int dfloor = 0;
sfloor = this.getSourceFloor();
dfloor = this.getDestinationFloor();

if(dfloor > sfloor+2) {
this.setDriveState(dmove_upfast);
getDriveState().Entry();
}
}

public void stop()
{
this.setDriveState(d_stopped);
getDriveState().Entry();
}
}
```

DriveState.java

```
public class DriveState {
private DriveStopped d_stopped = new DriveStopped();
private DriveMoveUpSlow dmove_upslow = new DriveMoveUpSlow();
private DriveMoveUpFast dmove_upfast = new DriveMoveUpFast();
private DriveMoveDownSlow dmove_downslow = new DriveMoveDownSlow();
private DriveMoveDownFast dmove_downfast = new DriveMoveDownFast();
public void Entry()
{
}

public void Exit()
{
}
```

```
public void doActivity()
{
}

public void moveUpSlow()
{
}

public void moveUpFast()
{
}

public void moveDownSlow()
{
}

public void moveDownFast()
{
}

public void stop()
{
}
}
```

DriveStopped.java

```
public class DriveStopped {
public void Entry()
{
this.setSpeed(0);
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setStatus("stopped");
this.setDirection(null);
}

public void moveUpSlow()
{
this.setDriveState(dmove_upslow);
getDriveState().Entry();
}

public void moveDownSlow()
{
this.setDriveState(dmove_downslow);
getDriveState().Entry();
}
}
```

Elevator.java

```
public class Elevator {
public Elevator(){
}
private String ele_status = idle;
private int source_floor = 0;
private int current_floor = 0;
private int destination_floor = 0;
private String safe_situation = "safe";
private String moving_direction = null;
private ElevatorState elevator_state = new ElevatorState();
private CarPositionIndicator carpositionindicator = new
CarPositionIndicator();
private Dispatcher dispatcher = new Dispatcher();
private Building building = new Building();
private DriveControl drivecontrol = new DriveControl();
private CarLantern carlantern = new CarLantern();
private Door door = new Door();
private CarButton carbutton = new CarButton();
public ElevatorState getElevatorState()
{
return elevator_state;
}

public void setElevatorState(ElevatorState estate)
{
elevator_state=estate;
}

public String getEleStatus()
{
return ele_status;
}

public void setEleStatus(String status)
{
ele_status=status;
}

public int getSourceFloor()
{
return source_floor;
return cf;
}

public void setSourceFloor(int sfloor)
{
source_floor=sfloor;
current_floor=cfloor;
}

public int getCurrentFloor()
{
int cf = -1;
cf = dispatcher.getCurrentFloor();
```

```
}

public void setCurrentFloor(int cfloor)
{
}

public int getDestinationFloor()
{
return destination_floor;
}

public void setDestinationFloor(int dfloor)
{
destination_floor=dfloor;
}

public String getElevatorSituation()
{
return safe_situation;
}

public void setSafeSituation(String safety)
{
safe_situation=safety;
}

public String getMovingDirection()
{
return moving_direction;
}

public void setMovingDirection(String cur_direction)
{
moving_direction=cur_direction;
}

public void hallCall(int calling_floorno)
{
building.hallCallButtonPressed(calling_floorNo, current_floorNo);
dispatcher.enque(calling_floorNo);
building.hallCallButtonReleased(calling_floorNo);

if(ele_status == "idle") {
this.getNextDestination();
}
}

public void carCall(int destination_floorno)
{
boolean isValid = false;
isValid = building.validFloor(destination_floorNo);

if(isValid == true) {
ele_status="active";
int carlantern_no = 0;
carbutton[destination_floor].illuminate();
destination_floor=destination_floorNo;
```

```
current_floor=source_floor;
this.determineMovingDirection();
for(int i=0; i<2; i++){
door[i].close();
}

if(moving_direction == "up") {
carlantern_no=1;
}
carlantern[carlantern_no].illuminate();
carpositionindicator.show(destination_floor);

if(moving_direction == "up") {
drivecontrol.moveUp(source_floor, destination_floor);
}

if(moving_direction == "down") {
drivecontrol.moveDown(source_floor, destination_floor);
}
dispatcher.atFloor(source_floor, destination_floor, moving_direction);
drivecontrol.stop();
carlantern[carlantern_no].turnOff();
carbutton[destination_floor].turnOff();
current_floor=destination_floor;
source_floor=current_floor;
for(int i=0; i<2; i++){
door[i].open();
}
carpositionindicator.clear();
carpositionindicator.show(current_floor);
}
}

public void determineMovingDirection()
{

if(source_floor < destination_floor) {
this.setMovingDirection("up");
}

if(source_floor > destination_floor) {
this.setMovingDirection("down");
}
}

public void doorReversal()
{
for(int i=0; i<2; i++){
door[i].reverseDoors();
}
}

public void setSafety(int safety)
{

if(safety == 0) {
safe_situation="unsafe";
```

```
if(safety == 1) {
safe_situation="safe";
}
}

public void getNextDestination()
{
boolean queue_empty = false;
queue_empty = dispatcher.isEmpty();

if(queue_empty == false) {
ele_status="active";
int next_floor = -01;
next_floor = dispatcher.deque();
this.carCall(next_floor);
}

if(queue_empty == true) {
drivecontrol.stop();
ele_status="idle";
for(int i=0; i<2; i++){
door[i].close();
carlantern[i].turnOff();
}
source_floor=current_floor;
destination_floor=-1;
carbutton[current_floor].turnOff();
carpositionindicator.show(current_floor);
}
}

public void move(int destinationfloor)
{
boolean isValid = false;
isValid = building.validFloor(destinationFloor);

if(isValid == true) {
this.carCall();
}

if(isValid == false) {
this.getNextDestination();
}
}

public void stop()
{
drivecontrol.stop();
for(int i=0; i<2; i++){
carlantern[i].turnOff();
}
int curr_floor = -1;
curr_floor = dispatcher.getCurrentFloor();
current_floor=curr_floor;
source_floor=curr_floor;
```

```
destination_floor=-1;
for(int i=0; i<2; i++){
door[i].open();
}
carpositionindicator.clear();
carpositionindicator.show(current_floor);
}

public boolean isEmpty()
{
boolean queue_status = false;
queue_status = queue.isEmpty();
return queue_status;
}

public void incrementFloor()
{
dispatcher.incrementFloor();
}

public void decrementFloor()
{
dispatcher.decrementFloor();
}

public void closeDoors()
{
for(int i=0; i<2; i++){
door[i].close();
}
}

public void openDoors()
{
for(int i=0; i<2; i++){
door[i].open();
}
}

public void call()
{
elevator_state.call();
}

public void selectFloor()
{
elevator_state.selectFloor();
}

public void ele_move()
{
}
}
```

ElevatorControl.java

```
public class ElevatorControl {
public ElevatorControl(){
}
private int controller = 1;
private EmergencyBrake emergencybrake = new EmergencyBrake();
private Elevator elevator = new Elevator();
public void hallButtonPressed(int calling_floorno)
{
elevator.hallCall(calling_floorno);
}

public void carButtonPressed(int destinationfloor)
{
elevator.carCall(destinationfloor);
}

public void doorReversal()
{
elevator.doorReversal();
}

public void triggerEmergencyBrakes()
{
emergencybrake.apply();
elevator.setSafety(0);
}

public void releaseEmergencyBrakes()
{
emergencybrake.release();
elevator.setSafety(1);
}

public void moveElevator(int destinationfloor)
{
elevator.move(destinationFloor);
}

public void stopElevator()
{
elevator.stop();
}
}
```

ElevatorState.java

```
public class ElevatorState {
private IdleDoorClosed idleDoorClosed = new IdleDoorClosed();
private StartMovingUp startMovingUp = new StartMovingUp();
private StartMovingDown startMovingDown = new StartMovingDown();
private MovingDownTheFloors movingDownTheFloors = new MovingDownTheFloors();
private MovingUpTheFloors movingUpTheFloors = new MovingUpTheFloors();
private ResumeMovingUp resumeMovingUp = new ResumeMovingUp();
```

```
private ResumeMovingDown resumeMovingDown = new ResumeMovingDown();
private ResumeDoorClosed resumeDoorClosed = new ResumeDoorClosed();
private IdleDoorOpen idleDoorOpen = new IdleDoorOpen();
private FloorSelectedDoorClosed floorSelectedDoorClosed = new
FloorSelectedDoorClosed();
private IdleDoorOpenReached idleDoorOpenReached = new IdleDoorOpenReached();
private FloorSelectedDoorClosedCalled floorSelectedDoorClosedCalled = new
FloorSelectedDoorClosedCalled();
private FloorSelectedDoorOpen floorSelectedDoorOpen = new
FloorSelectedDoorOpen();
public void Entry()
{
}

public void Exit()
{
}

public void doActivity()
{
}

public void call()
{
}

public void move()
{
}

public void selectFloor()
{
}
}
```

EmergencyBrake.java

```
public class EmergencyBrake {
public EmergencyBrake(){
}
private String status = "idle";
private int error_situation = 0;
private EmergencyBrakeState eb_state = new EmergencyBrakeState();
public String getStatus()
{
return status;
}

public void setStatus(String ebrake_status)
{
status=ebrake_status;
}

public int getErrorValue()
{
return error_situation;
```

```
}

public void setErrorValue(int error)
{
error_situation=error;
}

public EmergencyBrakeState getEBState()
{
return eb_state;
}

public void setEBState(EmergencyBrakeState ebrake_state)
{
eb_state=ebrake_state;
}

public void apply()
{
eb_state.applyBrakes();
}

public void release()
{
eb_state.releaseBrakes();
}
}
```

EmergencyBrakeActive.java

```
public class EmergencyBrakeActive {
public void Entry()
{
this.setErrorValue(01);
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setStatus("active");
}

public void releaseBrakes()
{
this.setEBState(eb_idle);
getEBState().Entry();
}
}
```

EmergencyBrakeIdle.java

```
public class EmergencyBrakeIdle {
public void Entry()
{
this.setErrorValue(0);
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setStatus("idle");
}

public void applyBrakes()
{
this.setEBState(eb_active);
getEBState().Entry();
}
}
```

EmergencyBrakeState.java

```
public class EmergencyBrakeState {
private EmergencyBrakeActive eb_active = new EmergencyBrakeActive();
private EmergencyBrakeIdle eb_idle = new EmergencyBrakeIdle();
public void Entry()
{
}

public void Exit()
{
}

public void doActivity()
{
}

public void applyBrakes()
{
}

public void releaseBrakes()
{
}
}
```

Floor.java

```
public class Floor {
```

```
public Floor(){
}
private int calling_floor = 0;
private int current_floor = 0;
private int current_button_no = 0;
private HallCallButton hall_call_button = new HallCallButton();
public void hallButtonPress(int call_floor,int curr_floor)
{
calling_floor=call_floor;
current_floor=curr_floor;

if(calling_floor < current_floor) {
current_button_no=0;
}

if(calling_floor > current_floor) {
current_button_no=1;
}
hall_call_button[current_button_no].press();
}

public void hallButtonRelease()
{
hall_call_button[current_button_no].release();
}

public int getCurrentFloor()
{
return current_floor;
}

public void setCurrentFloor(int floor)
{
current_floor=floor;
}

public int getCallingFloor()
{
return calling_floor;
}

public void setCallingFloor(int cfloor)
{
calling_floor=cfloor;
}
}
```

FloorSelectedDoorClosed.java

```
public class FloorSelectedDoorClosed {
public void Entry()
{
this.doActivity();
}

public void Exit()
```

```
{
}

public void doActivity()
{
this.closeDoors();
}

public void call()
{
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(curr_floor == destination_floor) {
this.setElevatorState(floorSelectedDoorOpen);
getElevatorState().Entry();
}

if(curr_floor == destination_floor) {
this.setElevatorState(floorSelectedDoorClosedCalled);
getElevatorState().Entry();
}
}

public void move()
{
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(destination_floor < curr_floor) {
this.setElevatorState(resumeMovingDown);
getElevatorState().Entry();
}

if(destination_floor > curr_floor) {
this.setElevatorState(resumeMovingUp);
getElevatorState().Entry();
}
}
}
```

FloorSelectedDoorClosedCalled.java

```
public class FloorSelectedDoorClosedCalled {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.Enque();
```

```
}

public void call()
{
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(curr_floor == destination_floor) {
this.setElevatorState(floorSelectedDoorOpen);
getElevatorState().Entry();
}
}
}
```

FloorSelectedDoorOpen.java

```
public class FloorSelectedDoorOpen {
public void Entry()
{
this.closeDoors();
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
}

public void call()
{
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(curr_floor == destination_floor) {
this.setElevatorState(floorSelectedDoorOpen);
getElevatorState().Entry();
}
}

public void selectFloor()
{
this.setElevatorState(floorSelectedDoorOpen);
getElevatorState().Entry();
}
}
```

HallCallButton.java

```
public class HallCallButton extends Button {
public HallCallButton() {
super() ;
```

```
}
private String button_status = "idle";
private String light_status = "off";
private HallCallButtonState hb_state = new HallCallButtonState();
public String getButtonStatus()
{
return button_status;
}

public void setButtonStatus(String bstatus)
{
button_status=bstatus;
}

public String getLightStatus()
{
return light_status;
}

public void setLightStatus(String lstatus)
{
light_status=lstatus;
}

public HallCallButtonState getHallCallButtonState()
{
return hb_state;
}

public void setHallCallButtonState(HallCallButtonState hbstate)
{
hb_state=hbstate;
}

public void press()
{
hb_state.illuminate();
}

public void release()
{
hb_state.darken();
}
}
```

HallCallButtonOff.java

```
public class HallCallButtonOff {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}
```

```
public void doActivity()
{
this.setButtonStatus("idle");
this.setLightStatus("off");
}

public void illuminate()
{
this.setHallCallButtonState(hb_on);
getHallCallButtonState().Entry();
}
}
```

HallCallButtonOn.java

```
public class HallCallButtonOn {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setButtonStatus("pressed");
this.setLightStatus("on");
}

public void darken()
{
this.setHallCallButtonState(hb_off);
getHallCallButtonState().Entry();
}
}
```

HallCallButtonState.java

```
public class HallCallButtonState {
private HallCallButtonOn hb_on = new HallCallButtonOn();
private HallCallButtonOff hb_off = new HallCallButtonOff();
public void Entry()
{
}

public void Exit()
{
}

public void doActivity()
{
}
```

```
public void illuminate()
{
}

public void darken()
{
}
}
```

IdleDoorClosed.java

```
public class IdleDoorClosed {
public void Entry()
{
this.closeDoors();
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
super.setEleStatus("idle");
source_floor=0;
current_floor=0;
destination_floor=0;
}

public void call()
{

if(current_floor < destination_floor) {
this.setElevatorState(startMovingUp);
getElevatorState().Entry();
}

if(current_floor > destination_floor) {
this.setElevatorState(startMovingDown);
getElevatorState().Entry();
}

if(current_floor == destination_floor) {
this.setElevatorState(idleDoorOpen);
getElevatorState().Entry();
}
}
}
```

IdleDoorOpen.java

```
public class IdleDoorOpen {
```

```
public void Entry()
{
this.openDoors();
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
boolean queue_empty = false;
queue_empty = this.isEmpty();

if(queue_empty == false) {
this.setElevatorState(resumeDoorClosed);
getElevatorState().Entry();
}
}

public void call()
{
this.Entry();
}

public void selectFloor()
{
this.setElevatorState(floorSelectedDoorClosed);
getElevatorState.Entry();
}
}
```

IdleDoorOpenReached.java

```
public class IdleDoorOpenReached {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
boolean queue_empty = false;
queue_empty = this.isEmpty();

if(queue_empty == false) {
this.setElevatorState(resumeDoorClosed);
getElevatorState().Entry();
}
}
```

```
public void call()
{
this.setElevatorState(idleDoorOpen);
getElevatorState().Entry();
}

public void selectFloor()
{
this.setElevatorState(floorSelectedDoorClosed);
getElevatorState().Entry();
}
}
```

MovingDownTheFloors.java

```
public class MovingDownTheFloors {
public void Entry()
{
this.setEleStatus("moving");
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.decrementFloor();
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(curr_floor == destination_floor) {
String dir = null;
dir = this.getMovingDirection();

if(dir == "down") {
this.setElevatorState(idleDoorOpenReached);
getElevatorState().Entry();
}
}

if(curr_floor != destination_floor) {
this.doActivity();
}
}

public void call()
{
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(curr_floor == destination_floor) {
this.setElevatorState(idleDoorOpen);
getElevatorState().Entry();
}
```

MovingUpTheFloors.java

```
public class MovingUpTheFloors {
public void Entry()
{
this.setEleStatus("moving");
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.incrementFloor();
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(curr_floor == destination_floor) {
String dir = null;
dir = this.getMovingDirection();

if(dir == "up") {
this.setElevatorState(idleDoorOpenReached);
getElevatorState().Entry();
}
}

if(curr_floor != destination_floor) {
this.doActivity();
}
}

public void call()
{
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(curr_floor == destination_floor) {
this.setElevatorState(idleDoorOpen);
getElevatorState().Entry();
}
}
}
```

ResumeDoorClosed.java

```
public class ResumeDoorClosed {
public void Entry()
{
```

```
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.closeDoors();
}

public void call()
{
this.setElevatorState(idleDoorOpen);
getElevatorState().Entry();
}

public void move()
{
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(destination_floor > curr_floor) {
this.setElevatorState(resumeMovingUp);
getElevatorState().Entry();
}

if(destination_floor < curr_floor) {
this.setElevatorState(resumeMovingDown);
getElevatorState().Entry();
}
}
}
```

ResumeMovingDown.java

```
public class ResumeMovingDown {
public void Entry()
{
this.setEleStatus("moving");
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(curr_floor != source_floor) {
this.setElevatorState(movingDownTheFloors);
getElevatorState().Entry();
```

```
}
}
}
```

ResumeMovingUp.java

```
public class ResumeMovingUp {
public void Entry()
{
this.setEleStatus("moving");
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(curr_floor != source_floor) {
this.setElevatorState(movingUpTheFloors);
getElevatorState().Entry();
}
}
}
```

StartMovingDown.java

```
public class StartMovingDown {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setEleStatus("moving");
this.setMovingDirection("down");
}

public void move()
{
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(curr_floor != source_floor) {
this.setElevatorState(movingDownTheFloors);
```

```
getElevatorState().Entry();
}
}
}
```

StartMovingUp.java

```
public class StartMovingUp {
public void Entry()
{
this.doActivity();
}

public void Exit()
{
}

public void doActivity()
{
this.setEleStatus("moving");
this.setMovingDirection("up");
}

public void move()
{
int curr_floor = -01;
curr_floor = this.getCurrentFloor();

if(curr_floor != source_floor) {
this.setElevatorState(movingUpTheFloors);
getElevatorState().Entry();
}
}
}
```