

BUILD.NET: A GRAPHICAL APPLICATION GENERATOR FOR
OBJECT-ORIENTED SOFTWARE AND SAMPLE APPLICATIONS

by

Mehmet Gönen

B.S., Industrial Engineering, Boğaziçi University, 2003

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2005

ACKNOWLEDGEMENTS

First of all, I am grateful to my thesis supervisor Prof. H. Levent Akin and my thesis co-supervisor Assoc. Prof. Ümit Bilge for privileging me to work with them. Their guidance and support throughout this study made things easier for me.

I would like to express my gratitude to Prof. Ethem Alpaydın, Prof. Fikret Gürgen and Assoc. Prof. Ali Tamer Ünal for taking part in my thesis jury.

I would like to thank my colleagues Salim, Erinç, Yavuz and Erhan for the delightful times we have spent together.

Finally, I want to thank my family for their continuous support and encouragement. This could not be true without them.

The work reported in this thesis is supported by Boğaziçi University Research Fund under Grant No.s: 01S107 and 05A301, and by State Planning Organization under Grant No: 01K120310.

ABSTRACT

BUILD.NET: A GRAPHICAL APPLICATION GENERATOR FOR OBJECT-ORIENTED SOFTWARE AND SAMPLE APPLICATIONS

Existing application and code generators are designed for domain- specific tasks. They are used in the intermediate steps of software development process and are not intended to develop full scale applications. BUILD.NET introduces a new graphical application generation framework for object-oriented software implementation. It utilizes a language-neutral representation of source code through dialog forms and flow diagrams. This representation method allows non-programmers to develop object-oriented software easily. The designed framework can convert this graphical representation into source code in four different programming languages. Generated source code can be compiled into executable directly without using any external compiler or interpreter.

Proposed application generation framework is tested on a complex software development project. An object-oriented discrete-event simulation package for flexible manufacturing systems, FMS.NET, is implemented by using BUILD.NET. This field study shows that this framework can easily be utilized on real life scenarios.

ÖZET

BUILD.NET: NESNE TABANLI YAZILIMLAR İÇİN ÇİZGESEL UYGULAMA ÜRETİCİSİ VE ÖRNEK UYGULAMALARI

Mevcut uygulama ve kod üreticileri özel görevler için tasarlanmışlardır. Yazılım geliştirme süreçlerinin bazı adımlarında kullanılırlar ve bir yazılımın bütününe geliştirilmesi için yeterli değildir. BUILD.NET nesne tabanlı yazılımlar için yeni bir çizgesel uygulama üretme çerçevesi ortaya koymaktadır. Diyalog formları ve akış şemaları kullanarak kaynak kodunu programlara diline bağlı olmadan temsil edebilmektedir. Bu yolla programlama dili bilmeyenlerde nesne tabanlı yazılımlar geliştirebilmektedir. Geliştirilen çerçeve bahsedilen çizgesel gösterimi dört farklı programlama dilinde kaynak koduna çevirebilmektedir. Oluşturulan kaynak kodu başka bir derleyici veya yorumlayıcı kullanılmadan uygulamaya çevrilebilir.

Önerilen uygulama üretme çerçevesi karmaşık bir yazılım geliştirme projesinde test edilmiştir. BUILD.NET kullanılarak esnek imalat sistemleri için nesne tabanlı bir kesik olay benzetim paketi geliştirilmiştir. Bu saha çalışması geliştirilen çerçevenin gerçek hayatta uygulanabilirliğini göstermiştir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
ÖZET	v
LIST OF FIGURES	ix
LIST OF TABLES	xv
LIST OF ABBREVIATIONS	xvii
1. INTRODUCTION	1
2. APPLICATION GENERATION	3
2.1. Motivation	3
2.2. Application Generator Design	6
2.2.1. Requirements	6
2.2.2. Choosing the Right Dimension	6
2.2.3. Different Approaches for Code Generation	7
2.2.3.1. Templates with Filtering	8
2.2.3.2. Templates with Metamodel	8
2.2.3.3. Frame Processing	9
2.2.3.4. API-Based Generators	9
2.2.3.5. Inline Code Generation	10
2.2.3.6. Code Attributes	10
2.2.3.7. Code Weaving	10
2.2.4. Existing Work	11
2.2.5. Advantages and Disadvantages of Using Code Generators	13
2.3. Code Document Object Model	13
3. BUILD.NET: THE PROPOSED APPLICATION GENERATOR	19
3.1. Requirements	19
3.2. Target Users and Application Areas	20
3.3. Design Issues	21
3.3.1. Structure Definition Classes	22
3.3.1.1. ModelDefinition Class	24

3.3.1.2.	NamespaceDefinition Class	24
3.3.1.3.	DelegateDefinition Class	24
3.3.1.4.	ClassDefinition Class	26
3.3.1.5.	EventDefinition Class	26
3.3.1.6.	FieldDefinition Class	28
3.3.1.7.	PropertyDefinition Class	28
3.3.1.8.	MethodDefinition Class	29
3.3.1.9.	ParameterDefinition Class	30
3.3.2.	Operation Definition Classes	31
3.3.2.1.	Statement Class	31
3.3.2.2.	StatementLinkPoint Class	31
3.3.2.3.	StatementLink Class	33
3.3.2.4.	Expression Class	33
3.4.	Sample Source Code to Object Graph Conversions	35
3.5.	Implementation Details	37
3.5.1.	Logical Layer	37
3.5.2.	GUI Layer	39
3.5.3.	Experimentation	41
4.	AN APPLICATION: A NEW FMS SIMULATOR (FMS.NET)	42
4.1.	Flexible Manufacturing Systems Simulation	42
4.1.1.	Problem Definition	44
4.1.1.1.	Static Model Data	44
4.1.1.2.	Dynamic Model Data	46
4.1.1.3.	Operational Decisions	46
4.2.	Design of FMS.NET	48
4.2.1.	FMS.NET.Random Namespace	49
4.2.2.	FMS.NET.Namespace	50
4.2.3.	FMS.NET.Layout Namespace	52
4.2.4.	FMS.NET.Decision Namespace	54
4.2.4.1.	Blockage Solving Algorithms	55
4.2.4.2.	Matching Algorithms	56
4.2.4.3.	Dispatching Algorithms	56

4.2.4.4.	Routing Algorithms	56
4.2.4.5.	Traffic Management Algorithms	56
4.2.4.6.	Operation Decision Algorithms	56
4.2.4.7.	Process Order Algorithms	57
4.2.5.	FMS.NET.Operation Namespace	57
4.3.	Implementation of FMS.NET	63
4.4.	Experimentation	64
4.4.1.	Experiment#1	67
4.4.2.	Experiment#2	68
4.4.3.	Experiment#3	68
5.	CONCLUSIONS AND FUTURE STUDIES	71
	APPENDIX A: UML DIAGRAMS OF BUILD.NET CLASSES	74
	APPENDIX B: SCREENSHOTS OF BUILD.NET	87
	APPENDIX C: UML DIAGRAMS OF FMS.NET CLASSES	93
	APPENDIX D: SIMULATION RESULTS	104
	REFERENCES	106
	REFERENCES NOT CITED	110

LIST OF FIGURES

Figure 2.1.	Application generation methodology	5
Figure 2.2.	Application generation framework	5
Figure 2.3.	Trade off between expressive power for design and expressive power for implementation	7
Figure 2.4.	An optimal tool design	7
Figure 2.5.	Code generation patterns categorization	8
Figure 2.6.	Templates with filtering code generation pattern	8
Figure 2.7.	Templates with metamodel code generation pattern	9
Figure 2.8.	Frame processing code generation pattern	9
Figure 2.9.	API-based code generation pattern	10
Figure 2.10.	Inline code generation pattern	10
Figure 2.11.	Code attributes code generation pattern	10
Figure 2.12.	Code weaving code generation pattern	11
Figure 2.13.	Example table for DOM	14
Figure 2.14.	Graphical representation of the DOM of the example table	15

Figure 2.15. CodeDOM hierarchy	16
Figure 3.1. BUILD.NET framework	19
Figure 3.2. UML diagram of structure definition classes	23
Figure 3.3. C# code of a namespace declaration	24
Figure 3.4. XML document of a namespace declaration	25
Figure 3.5. C# code of a delegate declaration	25
Figure 3.6. XML document of a delegate declaration	25
Figure 3.7. C# code of a class declaration	26
Figure 3.8. XML document of a class declaration	27
Figure 3.9. C# code of an event declaration	27
Figure 3.10. XML document of an event declaration	27
Figure 3.11. C# code of a field declaration	28
Figure 3.12. XML document of a field declaration	28
Figure 3.13. C# code of a property declaration	29
Figure 3.14. XML document of a property declaration	29
Figure 3.15. C# code of a method declaration	30

Figure 3.16. XML document of a method declaration	30
Figure 3.17. UML diagram of operation definition classes	33
Figure 3.18. Sample statement decomposition for an assignment statement . . .	35
Figure 3.19. Sample statement decomposition for a variable declaration statement	35
Figure 3.20. Sample source code for object graph conversion	36
Figure 3.21. Object graph corresponds to sample source code	36
Figure 3.22. Method explorer	40
Figure 4.1. Sample flow path layout	45
Figure 4.2. Alternative operation routes of a job type	47
Figure 4.3. FMS.NET framework	49
Figure 4.4. System workflow in an FMS	62
Figure 4.5. Simulation framework information flow	64
Figure 4.6. FMS layout used in experiments	65
Figure 4.7. Completed job count with changing AGV fleet size	67
Figure 4.8. Average flow time with changing AGV fleet size	68
Figure 4.9. Completed job count with changing AGV fleet size after doubling AGV capacities	69

Figure 4.10. Average flow time with changing AGV fleet size after doubling AGV capacities	69
Figure A.1. BUILD.NET class diagrams - I	75
Figure A.2. BUILD.NET class diagrams - II	76
Figure A.3. BUILD.NET class diagrams - III	77
Figure A.4. BUILD.NET class diagrams - IV	78
Figure A.5. BUILD.NET class diagrams - V	79
Figure A.6. BUILD.NET class diagrams - VI	80
Figure A.7. BUILD.NET class diagrams - VII	81
Figure A.8. BUILD.NET class diagrams - VIII	82
Figure A.9. BUILD.NET class diagrams - IX	83
Figure A.10. BUILD.NET class diagrams - X	84
Figure A.11. BUILD.NET class diagrams - XI	85
Figure A.12. BUILD.NET class diagrams - XII	86
Figure B.1. Model definition wizard	87
Figure B.2. Namespace definition wizard	87
Figure B.3. Delegate definition wizard	88

Figure B.4.	Class definition wizard	88
Figure B.5.	Enumeration definition wizard	89
Figure B.6.	Interface definition wizard	89
Figure B.7.	Struct definition wizard	90
Figure B.8.	Event definition wizard	90
Figure B.9.	Field definition wizard	91
Figure B.10.	Property definition wizard	91
Figure B.11.	Method definition wizard	92
Figure C.1.	FMS.NET class diagrams - I	94
Figure C.2.	FMS.NET class diagrams - II	95
Figure C.3.	FMS.NET class diagrams - III	96
Figure C.4.	FMS.NET class diagrams - IV	97
Figure C.5.	FMS.NET class diagrams - V	98
Figure C.6.	FMS.NET class diagrams - VI	99
Figure C.7.	FMS.NET class diagrams - VII	100
Figure C.8.	FMS.NET class diagrams - VIII	101

Figure C.9. FMS.NET class diagrams - IX	102
---	-----

Figure C.10. FMS.NET class diagrams - X	103
---	-----

LIST OF TABLES

Table 3.1.	Structure definition classes	22
Table 3.2.	Operation definition classes	31
Table 3.3.	Operation definition classes derived from Statement class and sample programming language statements	32
Table 3.4.	Operation definition classes derived from Expression class and their tasks	34
Table 4.1.	FMS.NET namespaces	49
Table 4.2.	FMS.NET.Random namespace classes	51
Table 4.3.	FMS.NET namespace classes	51
Table 4.4.	FMS.NET.Layout namespace classes	53
Table 4.5.	Algorithms in FMS.NET.Decision namespace	55
Table 4.6.	Job-related classes in FMS.NET.Operation namespace	57
Table 4.7.	Event-related classes in FMS.NET.Operation namespace	59
Table 4.8.	Management-related classes in FMS.NET.Operation namespace	61
Table 4.9.	Operations used in experiments and their processing times	66
Table 4.10.	Job set used in experiments	66

Table 4.11.	Decision algorithms used in experiments	66
Table 4.12.	Completed job count before and after increasing flexibility	70
Table 4.13.	Average flow time before and after increasing flexibility	70
Table D.1.	Completed job count results for Experiment#1	104
Table D.2.	Average flow time results for Experiment#1	104
Table D.3.	Completed job count results for Experiment#2	105
Table D.4.	Average flow time results for Experiment#2	105
Table D.5.	Completed job count and average flow time results for Experiment#3	105

LIST OF ABBREVIATIONS

AGV	Automated Guided Vehicle
API	Application Programming Interface
BNF	Backus-Naur Form
BUFAIM	Boğaziçi University Flexible Automation and Intelligent Manufacturing Laboratory
CASE	Computer-Aided Software Engineering
CNC	Computer Numerical Control
CodeDOM	Code Document Object Model
DOM	Document Object Model
EJB	Enterprise JavaBeans
FMS	Flexible Manufacturing Systems
GDI	Graphics Device Interface
GUI	Graphical User Interface
HTML	Hyper-Text Markup Language
MSDN	Microsoft Developer Network
OCL	Object Constraint Language
UML	Unified Modeling Language
W3C	World Wide Web Consortium
XHTML	Extensible Hyper-Text Markup Language
XLST	XML Stylesheet Language Transformation
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XQuery	XML Query Language

1. INTRODUCTION

Software development cycle starts with collecting requirements from the potential users. These requirements are converted into formal specifications. Software engineers take an active part in the process after this step. They convert the formal specifications into source code by utilizing a programming language. This programming step requires a good knowledge of the programming language to perform the conversion operation adequately. The potential users of the developed software typically can not participate the implementation phase actively due to lack of technical ability. Possible flaws in the requirements modeling phase can lead up to differences between the requirements and the resulting software. Incorporating the potential users into development process can solve this communication problem.

Programming languages have evolved over decades due to changing requirements and technological achievements. High-level programming languages are designed to overcome difficulties in programming with machine language or assembly language. After this breakthrough, application-oriented languages drew attention. Application-oriented languages are typically utilized in database management systems, user interface generators and report generators. They are also used in developing domain-specific application generators which target especially embedded and real-time systems. Domain-specific application generators had great success and are used extensively as support tools in software development (Rockstrom and Saracco, 1982; Lewis, 1990; Guerrieri, 1994; Batory *et al.*, 1998; Harada and Mizuno, 1999; Fertalj *et al.*, 2002; Helman and Fertalj, 2004).

Instead of concentrating on a specific domain, developing a general-purpose application generator could be a different approach. This application generator can be used by non-programmers to develop, modify and extend software without using any programming language. A smart input collection mechanism should be provided to enable the users to give their specifications to application generator. A graphical modeling approach is suitable to give and interpret the entered specifications easily.

In this thesis, a general-purpose application generation framework, BUILD.NET, for object-oriented software is proposed. In this framework, specifications are collected in a language-neutral manner from the users. Source code generation in four different programming languages and application generation - compilation - are performed automatically without user intervention. The designed framework can be easily extended to support other programming languages and used as an infrastructure for domain-specific application generators. Graphical representation forms such as dialog forms and flow diagrams are selected to gather information from the users in order to simplify input collection process and give more interpretability to the framework.

After developing the application generator, it was tested on the implementation of a complex software. Target application area was selected as developing an object-oriented discrete-event simulation package for flexible manufacturing systems. Mentioned simulation package was completely implemented by using BUILD.NET.

In the next chapter, application generation methodology and patterns are analyzed in detail. Example code and application generators from different domains are listed. The particular infrastructure used in the implementation of the proposed application generator is explained. Then, in Chapter 3, the design and implementation details of BUILD.NET are described. Language-neutral representation method is illustrated by examples in this chapter. Chapter 4 gives information about the simulation package developed by using BUILD.NET. This chapter also introduces flexible manufacturing systems briefly to the reader. Finally, conclusions drawn from this study and future research directions are explained in Chapter 5.

2. APPLICATION GENERATION

2.1. Motivation

The first generation of languages used to program a computer, is called machine language or machine code. It is actually the only language a computer really works with, a sequence of zeros and ones that the computer's controls interpret as instructions, electrically. The main advantage of using a first generation programming language is that the code can run very fast and efficiently since it is directly executed by the processor. Maintenance difficulty is a major problem for machine language, for example, if new instructions are added to memory at some location, all the instructions that come after new instructions should be moved down to make room for new instructions. Another problem is portability, transferring written code to a different computer is not always possible due to changes in architecture and machine language.

The second generation of languages is called assembly language. Assembly language turns the sequences of zeros and ones into mnemonics like *add*, *load* and *save*. Unlike first generation programming languages, the code can be read and written fairly easily by a human. Assembly language is always translated back into machine code by programs called assemblers. The language is specific to and dependent on a particular processor family and environment. Since it is the native language of a processor it has significant speed advantages, but it requires more programming effort and is difficult to use effectively for large applications due to following reasons:

- The programmer has to have a knowledge of the processor architecture and the instruction set.
- Many instructions are required to achieve small tasks.
- Source programs tend to be large and difficult to follow.
- Programs are machine dependent, requiring complete rewrites if the hardware is changed.

The third generation of languages, is called high-level languages, which are closer to natural languages and syntax (like words in a sentence). In order for the computer to understand any high-level language, a compiler or an interpreter translates the high-level language into either assembly language or machine code. All software programming languages need to be eventually translated into machine code for a computer to use the instructions they contain.

High-level programming languages like Ada, C (Kernighan and Ritchie, 1988) and C++ (Stroustrup, 1997) are designed to overcome the difficulties in designing complex software systems by using assembly language. It is inefficient to code and analyze such large systems in conventional methods. Similar problems emerge now with high-level programming languages as the applications are no longer small enough to perform detailed analysis. For this reason, fourth generation of programming languages and automatic application generators are proposed to deal with these problems and to decrease development efforts.

Software engineers typically need Computer-Aided Software Design (CASE) tools to speed up the development process. Sommerville (2000) defines CASE tools as tools that are used to support software process activities such as requirements engineering, design, program development and testing. CASE tools therefore include code editors, compilers, interpreters, user interface generators, debuggers, code generators and so on.

The fourth generation of languages, called application-oriented languages, which take specifications that describe the problem or task to be done by the program. These specifications are taken in the form of interactive dialog forms, graphical form or written in a fourth generation language. Query languages and report writers are fourth generation languages. Any computer language with English-like commands that does not require traditional input-process-output logic falls into this category. Many fourth generation language functions are built into graphical interfaces and activated by clicking and dragging.

Application generators get specifications from the user and convert them into executable applications. An application generator, operating much like a language compiler, translates the high-level information into a low level implementation. To change or modify the final program, it is sufficient to change the input specifications and return them through the generator. Figure 2.1 and Figure 2.2 show the application generation methodology and the basic processes that are needed to develop a program using an application generator.

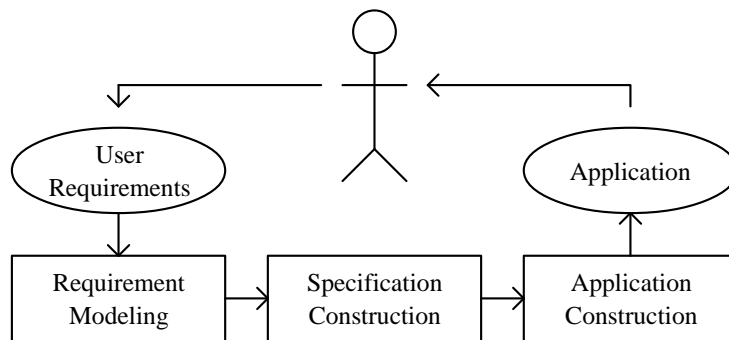


Figure 2.1. Application generation methodology

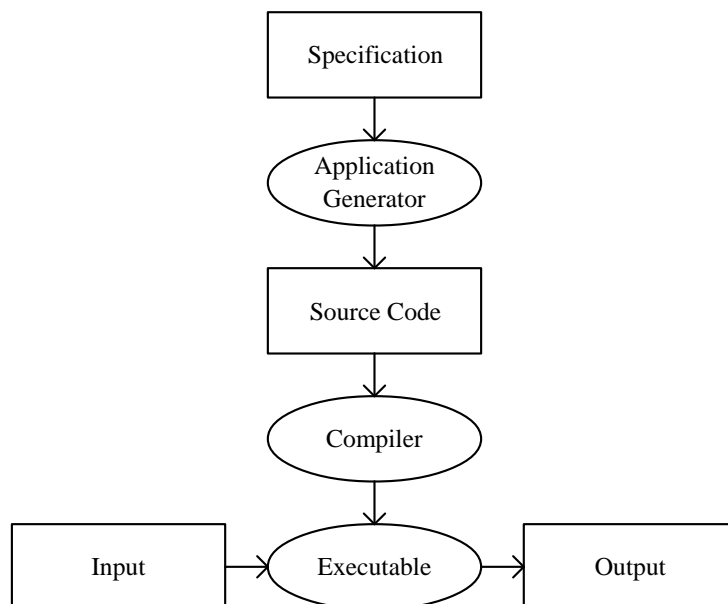


Figure 2.2. Application generation framework

2.2. Application Generator Design

2.2.1. Requirements

The goal of code generation from high-level formal requirements models is to increase productivity and quality by directly deriving production code from the formal model. Unfortunately, having a fully formal and correct requirements model does not guarantee the success of a development approach based on automated code generation; the translation and the translation tool must also be correct. Whalen and Heimdahl (1999) established a benchmark set of requirements for high integrity code generation:

- The source and target languages must have formally well defined syntax and semantics.
- The translation between a specification expressed in a source language and a program expressed in a target language must be formal and proven to maintain the meaning of the specification.
- Rigorous arguments must be provided to validate the translator and/or the generated code.
- The implementation of the translator must be rigorously tested and treated as high-assurance software.
- The generated code must be well structured, well documented, and easily traceable to the original specification.

2.2.2. Choosing the Right Dimension

Audsley *et al.* (2003) defined a trade-off relation between expressive power for design and expressive power for implementation. Figure 2.3 highlights four common types of CASE tools currently exist. Attempts to maximize both powers often give a tool that is not only complex but also inflexible. It is unrealistic to expect high expressive power on both axes from a single design tool. It is reasonable to split the process into different subprocesses, namely designing and generating code. Figure 2.4 illustrates the optimum and most achievable path from high expressive power for design

to high expressive power for implementation. Breaking the process up in this way simplifies the problem and reduces it to two different small problems. An application generator has two different tools: design tool and generator. Design tool is responsible for collecting specifications from user and storing them in a fourth generation language format. Generator can convert these specifications into source code or executable application after reading the specifications by using the same language design tool uses.

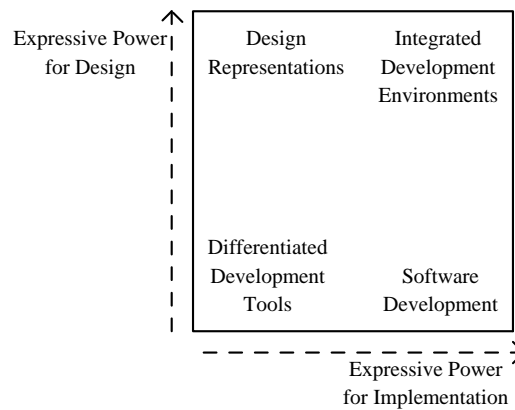


Figure 2.3. Trade off between expressive power for design and expressive power for implementation (Audsley *et al.*, 2003)

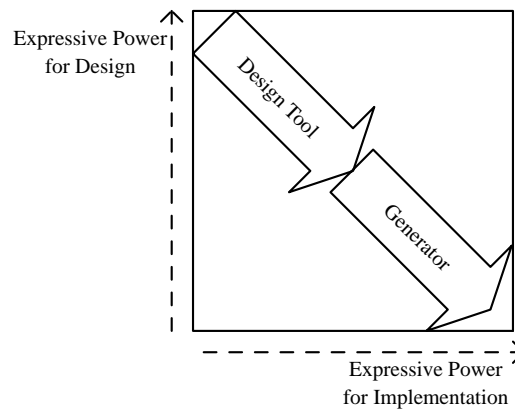


Figure 2.4. An optimal tool design (Audsley *et al.*, 2003)

2.2.3. Different Approaches for Code Generation

In this section, seven basic design patterns for code generation are listed and explained (Voelter, 2003). Each pattern is not analyzed in detail here but a quick preview and sample applications are presented to give necessary insights.

These seven patterns can be grouped into two main categories: API (Application Programming Interface) based approaches and template-based approaches. API-based generators are the most fundamental pattern in API-based approaches and other patterns can use it as a subcomponent. Templates with metamodel pattern is an extension to templates with filtering pattern in template-based approaches. This categorization is represented in Figure 2.5.

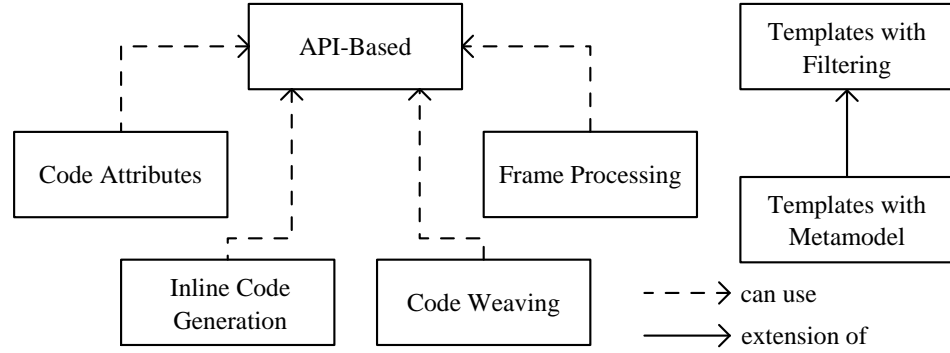


Figure 2.5. Code generation patterns categorization (Voelter, 2003)

2.2.3.1. Templates with Filtering. This is basically generating source code, simple class skeletons, from UML (Unified Modeling Language, 2005) models. UML models are generally formed by using XMI (XML Metadata Interchange, 2005) standard. XMI files are large files and contain a lot of information that are not useful in code generation. Before generating source code, these parts should be filtered out and the remaining part is used in generation process. This conversion can be made easily with the use of XSLT (XML Stylesheet Language Transformation, 2005) and XQuery (XML Query Language, 2005) but only class skeletons are obtained as final product.

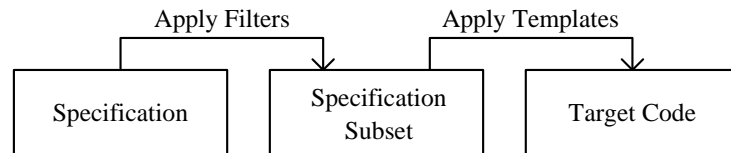


Figure 2.6. Templates with filtering code generation pattern

2.2.3.2. Templates with Metamodel. This pattern is suitable for domain specific application generators. Architectural building blocks are defined and the application is modeled by using these blocks. Blocks are mapped to implementation platform. The user is only concerned about domain specific issues but not implementation details.

UML models can be made more domain specific by using OCL (Object Constraint Language, 2005). EJB (Enterprise JavaBeans, 2005) bean types could be examples of this pattern. *Activities*, *transitions* and *processes* in a workflow system are modeled as building blocks and code generation is performed on these building blocks.

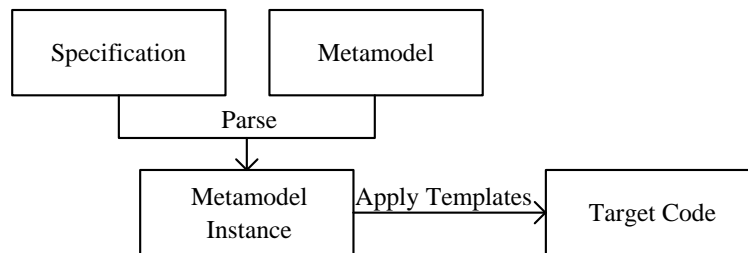


Figure 2.7. Templates with metamodel code generation pattern

2.2.3.3. Frame Processing. This pattern uses frames which are functions that generate code as the result of their evaluation. These frames can accept primitive data types or other frames as input parameters. Frame processing can be applied in two different ways: script-based frame processors whose frames are parameterized by filling slot values and injection-based frame processors whose frames are parameterized by inserting code directly. This method is not as efficient as the first two methods due to its imperative nature.

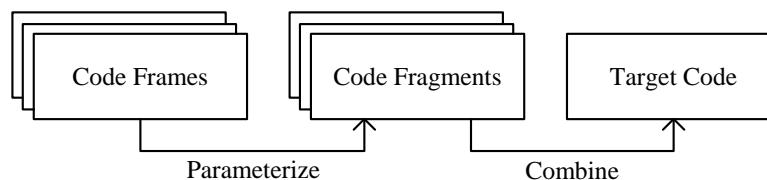


Figure 2.8. Frame processing code generation pattern

2.2.3.4. API-Based Generators. API-based systems depend on the syntax of the target language. The processor provides an API in terms of the syntax and abstractions. There are no templates or models to create code, instead a manually written program creates the code. Microsoft.NET CodeDOM (Code Document Object Model) (Microsoft Developer Network Library, 2004) technology is the most recent and advanced application of this pattern.

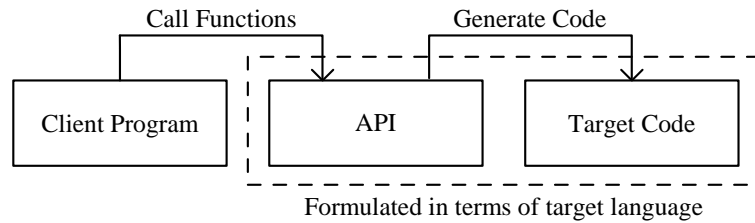


Figure 2.9. API-based code generation pattern

2.2.3.5. Inline Code Generation. Code is generated at compile time by means of a precompiler. Precompilation translates the original program into target program, then target program is compiled to generate final output. C++ template and preprocessing mechanisms are examples of inline code generation.

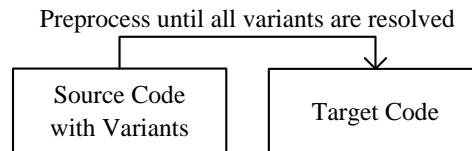


Figure 2.10. Inline code generation pattern

2.2.3.6. Code Attributes. Source code is composed not only of programming languages statements. Additional information is supplied with another representation other than the programming language. The code generator parses the code and uses this extra information to generate additional source code. XML configuration files and attributes in Microsoft.NET Framework are the most well-known applications of this approach. Microsoft.NET Framework compilers check these attributes and generate necessary source code and embed it to the written source code at compile time. Detailed information about code attributes in Microsoft.NET Framework can be found in Microsoft Developer Network Library (2004).

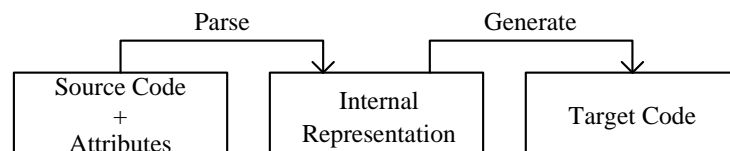


Figure 2.11. Code attributes code generation pattern

2.2.3.7. Code Weaving. Source code is written as different modules and connection specifications between these modules are specified to system. A code-weaver will then

join the different modules according to these specifications. The important step in this process is defining relationships between modules clearly. The most famous member of this pattern is AspectJ (AspectJ Project, 2005), a code-weaver for aspect-oriented programming in Java. IBM has also developed another tool named Hyper/J (Hyper/J Project, 2005) which allows composition and extraction of code artifacts for Java.

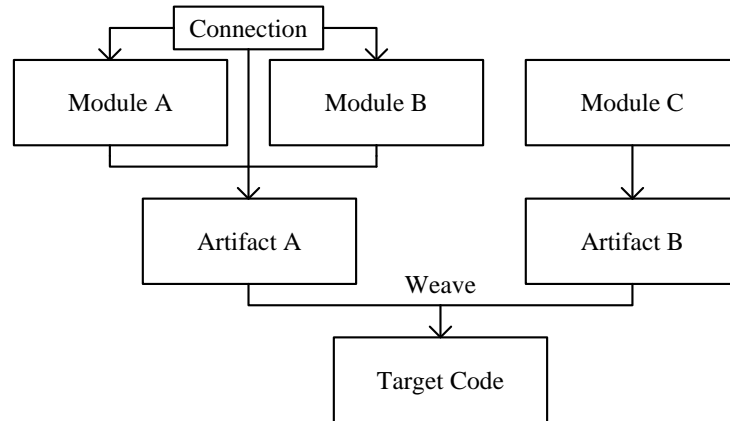


Figure 2.12. Code weaving code generation pattern

2.2.4. Existing Work

Automatic code and application generation has been extensively studied and a lot of work has been reported. In this section, some selected examples and their application areas are listed.

As an early attempt, Rockstrom and Saracco (1982) designed an application generator for telephony applications. They aimed to develop a representation form, called *Specification and Description Language* which enables construction of a telephony application by means of a graphical representation. They use different types of blocks namely, *state*, *input*, *task*, *output*, *decision* and *save*, to describe the functionality of the desired program. The user constructs the graph by using these blocks, connects them appropriately and the generator converts this flow diagram into an application.

Application generation was applied initially also to embedded and information systems. Lewis (1990) presents four different generators from different domains. The first one, called as *Tags*, is from the real-time control systems domain. *Microstep* is a

similar kind of tool for data-processing applications. *Escort* is another generator for telephony applications. These three generators use flow diagrams as input method. The last generator is *Programming System Generator* which generates programming environments from formal-language definitions.

A different usage is reported from relational database systems domain (Guerrieri, 1994). Digital Equipment Corporation developed *DECAdmire*, an application generator that provides code generation for database management systems, and saved 80 to 95 percent from development time through reuse.

Batory *et al.* (1998) used a component-based approach in code generation. Their code generator uses basic data structures to design complex data structures. Harada and Mizuno (1999) tried to build a code generator that generates C++ code from object-oriented design diagrams and decision tables in a domain-independent manner. Class skeletons are derived from object-oriented design diagrams and method implementations are implemented by using decision rules given in a formal language. Park and Kim (2001) proposed an XML specification format to store and describe source code generator input to avoid different storage representations of distinct UML tools. McLaren and Wicks (2001) also proposed an XML-based generation methodology and showed that XSLT can be effectively applied to transformation from specifications to source code.

Fertalj *et al.* (2002) proposed a template-based code generator for managing databases. Complex statements are generated from the high level specifications. A similar approach is also used to generate source code in different programming languages by using same templates (Helman and Fertalj, 2004).

Another example of domain-specific application generators comes from scheduling and optimization field. *ICRON* (Icron Technologies, 2005) is designed to perform planning and scheduling activities of enterprises. It allows to construct and execute scheduling algorithms by using its graphical modeling tool. Knowledgeable planners can easily perform their tasks without programming.

2.2.5. Advantages and Disadvantages of Using Code Generators

There are numerous advantages of using code generators in software development. Some of the advantages that are reported in the literature are:

- Customizing and reusing general software design easily.
- Increasing productivity during development and maintenance.
- Easier to read, write and change the input.
- Reducing programming errors, letting the designer concentrate on specification errors.
- Generating and maintaining application by non-programmers.
- Ease of prototyping and testing alternative specifications.
- Ease of standards implementation by using the generator to automatically create a standard interface or output format. It is especially important for concurrency, replication, security, availability and persistence issues.

There are also disadvantages of using code generators. Main disadvantages are:

- A single application generator can be used effectively only in a few situation.
- Application generators are difficult to build. They require carefully designed specification languages and user interfaces. In fact, building an application generator requires knowledge and skill in building parsers and language translators.
- Recognizing where an application generator can be used is difficult and often occurs late in life cycle, when there is less motivation to redo development.

2.3. Code Document Object Model

DOM (Document Object Model) is a specification provided by the W3C (World Wide Web Consortium). The W3C DOM is a platform and language-neutral interface that permits scripts to access and update the content, structure, and style of a document (DOM Level 3 Core Specification, 2004). The W3C DOM includes a model for how a standard set of objects representing HTML (Hyper-Text Markup Language)

and XML (Extensible Markup Language) documents are combined, and an interface for accessing and manipulating them.

Programmers can create documents, explore their structure, and insert, change, or remove elements and content via DOM. Any data found in an HTML or XML document can be accessed, changed, deleted, or added by using DOM. W3C aims to provide a standard programming interface that can be used in a wide variety of environments and applications while designing DOM. Basically, DOM is a programming API for document management. It is based on an object structure that closely resembles the structure of the documents it models. For instance, consider this table, taken from an XHTML (Extensible Hyper-Text Markup Language) document:

```

<table>
  <tbody>
    <tr>
      <td>Computer Engineering< /td>
      <td>CMPE< /td>
    < /tr>
    <tr>
      <td>Industrial Engineering< /td>
      <td>IE< /td>
    < /tr>
  < /tbody>
< /table>

```

Figure 2.13. Example table for DOM

Figure 2.14 shows the graphical representation of the DOM of the above table. Documents have a logical structure very much like a “forest” which can have more than one tree. Documents are modeled using objects, and the model includes not only the structure of a document, but also the behavior of a document and the objects of which it is composed. In other words, the nodes in the figure do not represent a data structure, they represent objects, which have functions and identity. As an object model, the DOM identifies:

- the interfaces and objects used to model a document
- the semantics of these interfaces and objects
- the relationships between these interfaces and objects

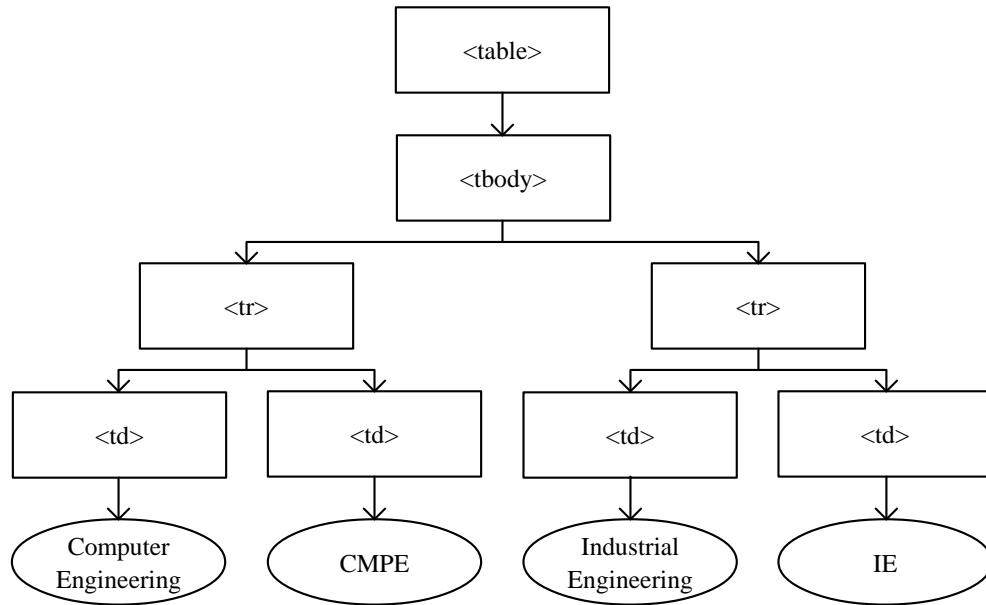


Figure 2.14. Graphical representation of the DOM of the example table

CodeDOM, a concept similar to DOM, is a language-neutral programmatic representation of source code. The basic intuition behind CodeDOM is to express the source code as an object graph as in DOM and use this graph as the basis in code generation. This object graph can be converted easily into any programming language that supports CodeDOM. Formal description of CodeDOM grammar is specified by using BNF (Backus-Naur Form) representation (CodeDOM Grammar, 2005). Figure 2.15 shows the hierarchical structure of basic CodeDOM elements.

Microsoft.NET Framework has two different namespaces to support CodeDOM. The first one is *System.CodeDOM* and it consists of classes that are required to build object graphs. The other one is *System.CodeDOM.Compiler* and it consists of classes that are used to convert an object graph into source code or executable. Important classes of these namespaces are explained in the following paragraphs. A detailed discussion about *System.CodeDOM* and *System.CodeDOM.Compiler* namespaces can be found in MSDN (Microsoft Developer Network Library, (2004)).

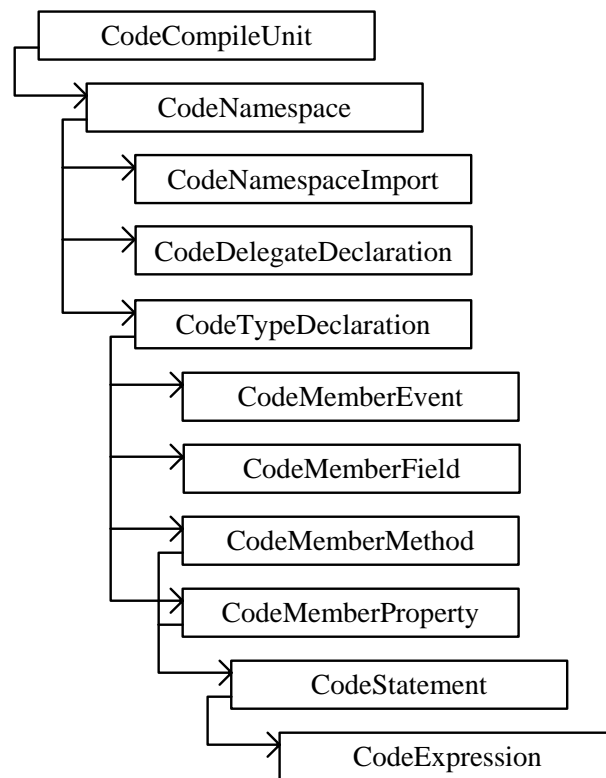


Figure 2.15. CodeDOM hierarchy

CodeCompileUnit is at the highest level of CodeDOM hierarchy because it is the complete object graph of a program and only it can be compiled by a CodeDOM compiler. *CodeCompileUnit* holds the global information about the represented program such as referenced assemblies¹ and project attributes.

Namespaces are used to provide fully qualified names to types and variables to avoid potential naming ambiguities in large software projects. *CodeNamespace* holds a namespace declaration and its imported namespaces, delegate declarations and type declarations.

CodeNamespaceImport holds the name of an imported namespace. Importing namespace can reference types in the imported namespace directly without using fully qualified names.

¹is primary unit of a .NET application and a self-describing collection of code, resources, and metadata.

CodeTypeDelegate is used to represent a delegate² definition in source code. It contains information about return type and parameters of underlying delegate.

CodeTypeDeclaration is used to define a class, structure, interface or enumeration in source code. It also stores the members of the declared type such as fields, properties, methods and events. Object-oriented concepts such as inheritance and polymorphism can be applied to type declarations.

CodeMemberEvent, *CodeMemberField*, *CodeMemberMethod* and *CodeMemberProperty* are used to add events, fields, methods and properties, respectively, to a given type. These members defines static and dynamic structure of a type. *CodeMemberMethod* and *CodeMemberProperty* require code statements to be meaningful.

CodeStatement is the base class for all different types of code statements. *CodeAssignStatement*, *CodeConditionStatement* and *CodeGotoStatement* are some types of code statements that are supported in CodeDOM. *CodeSnippetStatement* is also introduced to CodeDOM to represent statements not presently supported. This enables the user to enter regular source code into anywhere of a CodeDOM object graph.

CodeExpression is the base class for all different types of expressions that are used in building complex statements. *CodeCastExpression*, *CodeObjectCreateExpression* and *CodePrimitiveExpression* are some types of code expressions that are supported in CodeDOM. *CodeSnippetExpression* is also added due to same reasoning in *CodeSnippetStatement*.

CodeDOM also has other objects to support creating, compiling and running programs. Above mentioned objects are responsible for modeling the program as an object graph but there must be some other mechanism to convert them to source code and compile the obtained source code.

²is a reference type that can be used to encapsulate a method with a specific signature. Delegates are roughly similar to function pointers in C++; however, delegates are type-safe and secure.

CodeDomProvider can be used to create and retrieve instances of code generators and code compilers. Code generators can be used to generate source code for a particular language, and code compilers can be used to compile the source code into assemblies. *CodeGenerator* is capable of rendering source code in a specific language according to the structure of a CodeDOM object graph. *CodeCompiler* is like a regular compiler but it compiles a CodeDOM object graph not a text file. It gives compilation results such as compilation errors and warnings to provide information.

3. BUILD.NET: THE PROPOSED APPLICATION GENERATOR

In this study, a graphical application generation framework, called BUILD.NET, is designed and implemented. The designed application generator is capable of converting formal requirements that are collected via dialog forms and diagrams into source code (namely C++, C#, J# and VB.NET) and compiling the generated source code. This is achieved by the help of CodeDOM libraries of Microsoft.NET Framework. BUILD.NET represents source code as an object graph and stores it in an XML file. Figure 3.1 shows the basic elements of BUILD.NET framework.

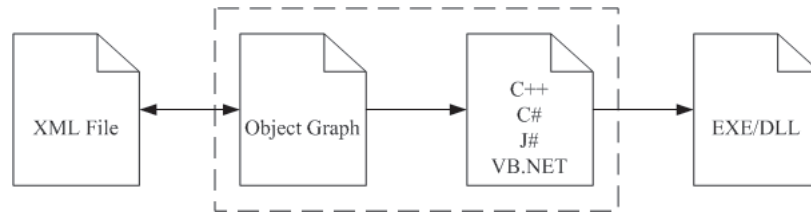


Figure 3.1. BUILD.NET framework

3.1. Requirements

Commercial application and code generators are designed to achieve domain-specific tasks such as database management, report generation and user interface generation. They help users to develop programs but are not capable of generating a program completely from scratch. A different approach is constructing a general-purpose application generator which does not specialize on a task.

The first concern should be representing a programming language completely in an application generator to develop any kind of application. This is possible by modeling all different statement (*assignments, loops, branches, etc.*) and expression types (*casting expressions, method invocation, primitive expressions, etc.*) that can be found in programming languages. The user can form any statement easily by using what the application generator provides.

The second concern is the input collection mechanism: the application generator should have necessary tools to collect input from the user. This can be achieved in different ways, but flow diagram representation is a good candidate for this task due to its easy interpretability. The user can be a non-programmer and the application generator should not expect a deep programming language knowledge from the user. The application generator must provide a high level abstraction of the target programming language.

Instead of concentrating on a domain-specific task, a general approach is selected and an application generator for object-oriented software is proposed. Proposed application generator will enable users to develop any kind of application without considering its domain properties.

3.2. Target Users and Application Areas

Application generators are designed for helping software engineers in their tasks but BUILD.NET has a different user profile. It is designed for non-programmers to develop applications without using any programming language. They give their algorithms with a high level abstract representation to the application generator and the application generator will generate source code or executable for them. BUILD.NET can also be used as a learning tool to teach programming languages. Users generate their algorithms in the form of flow diagrams and see the corresponding source code.

Extensibility is a major goal for scientific software packages such as simulation and optimization programs. This is achieved by providing APIs. Users can write their programs by using API function calls. This method requires programming language knowledge and user effort to learn API functions. Instead of providing APIs, a software package can be extended by using BUILD.NET and additional algorithms implemented by the user can be integrated into a commercial software product. This extension method helps non-programmers to develop their algorithms in a high level representation and combine them with already implemented algorithms by the vendor.

Another application area is updating released software packages. Vendors supply setup programs for customers to update software packages. This is generally achieved by DLL (Dynamically Linked Library) updates and this can cause some problems such version conflicts. Update progress can be performed by replacing an XML file if BUILD.NET is used to develop the software. Source code can be secured by utilizing encryption on XML files.

3.3. Design Issues

CodeDOM constitutes the infrastructure of designed application generator. As mentioned in Section 2.3, CodeDOM has the necessary classes to build an application generator. BUILD.NET needs additional classes to fill the gap between requirements for a graphical application generator and what the CodeDOM provides.

BUILD.NET can accept two forms of input from the user: dialog-based and diagram-based. Dialog-based input is used to describe the structure definition of the program such as namespace declarations, class declarations and method declarations. Diagram-based input is used to enable the user to enter operation definitions of declared properties and methods.

BUILD.NET is based on a combination of four different code generation patterns that are explained in Section 2.2.3. *Frame Processing* pattern is utilized in collection of structure definitions. The users should fill necessary slot values to define namespaces, delegates, classes, events, fields, properties and methods. Implementation details of methods and properties are collected via *Templates with Metamodel* pattern. Building blocks for basic programming language statements are defined and each block is responsible of creating its corresponding source code. BUILD.NET saves the project file as an XML file and this file contains information about implemented algorithms and the graphical layout of algorithm blocks. Necessary parts of the project file is filtered in code generation process by using *Templates with Filtering* pattern. Code generation process is achieved by using *API-Based Generators* pattern. Building blocks of algorithms are converted into source code by using API function calls provided by

Microsoft.NET Framework. Source code in different programming languages that are not supported by Microsoft.NET Framework can also be generated after implementing required conversion functions for these programming languages.

3.3.1. Structure Definition Classes

Structure definitions in Microsoft.NET Framework include namespace declarations, delegate declarations, class declarations, event declarations, field declarations, property declarations and method declarations. Structure definition classes are needed to model these declarations in BUILD.NET. Table 3.1 shows the structure definition classes and their roles. Figure 3.2 demonstrates the relationships between structure definition classes.

Table 3.1. Structure definition classes

Class Name	Explanation
ModelDefinition	Stores project related information and namespaces
NamespaceDefinition	Stores namespace information such as imported namespaces, delegate definitions and class definitions
DelegateDefinition	Stores delegate information such as return type and parameter definitions
ClassDefinition	Stores class information such as member definitions
EventDefinition	Stores event information such as type and name
FieldDefinition	Stores field information such as type and access mode
PropertyDefinition	Stores property information such as return type
MethodDefinition	Stores method information such as return type and parameter definitions
ParameterDefinition	Stores parameter information such as type and direction

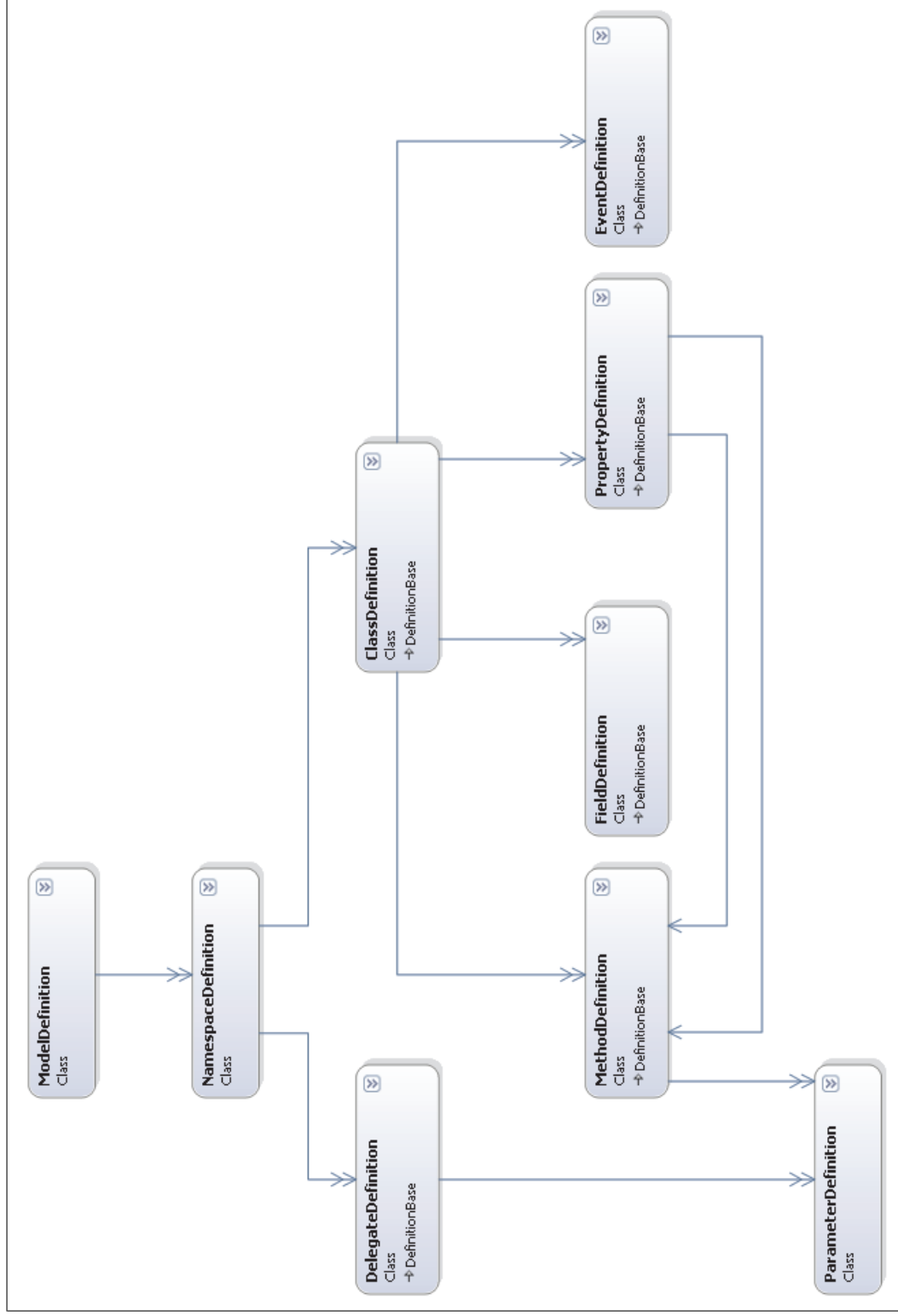


Figure 3.2. UML diagram of structure definition classes

3.3.1.1. ModelDefinition Class. *ModelDefinition* is the main class that stores the entire project and has the necessary methods to provide CodeDOM functionality. It manages the namespaces that form the underlying program in a list and UML diagrams that are generated by the user. Source code generation and compilation tasks are achieved by *ModelDefinition*.

3.3.1.2. NamespaceDefinition Class. *NamespaceDefinition* class represents namespace declarations that are defined in the project. It stores delegates and types that reside in namespace which it represents and names of imported namespaces. *NamespaceDefinition* class is converted into a *CodeNamespace* object to properly join to a *CodeCompileUnit* while source code generation or compilation. An example namespace declaration and corresponding XML document that is generated by BUILD.NET are shown in Figure 3.3 and Figure 3.4, respectively.

```
namespace FMS.NET {
    using System;
    using System.Collections;
    using System.IO;
    using System.Xml.Serialization;
}
```

Figure 3.3. C# code of a namespace declaration

3.3.1.3. DelegateDefinition Class. *DelegateDefinition* class is responsible for storing delegate related information such as name, return type, access modifiers and parameters. A *CodeTypeDelegate* object is created from delegate information to use it in a CodeDOM object graph. An example delegate declaration and corresponding XML document that is created by BUILD.NET are shown in Figure 3.5 and Figure 3.6, respectively.

A delegate can reference a method only if the signature of the method exactly matches the signature specified by the delegate type. When a delegate references an


```

<Namespace>
  <ID>52de3654-27ef-4ff5-916a-d6795d42a0e6< /ID>;
  <Name>FMS.NET< /Name>
  <Comment/ >
  <Imports>
    <Import>System< /Import>
    <Import>System.Collections< /Import>
    <Import>System.IO< /Import>
    <Import>System.XML.Serialization< /Import>
  <Imports/ >
  <Delegates/ >
  <Classes/ >
< /Namespace>

```

Figure 3.4. XML document of a namespace declaration

```
public delegate void MatchingDelegate();
```

Figure 3.5. C# code of a delegate declaration

```

<Delegate>
  <ID>d06f73c6-91b7-48b8-a3ac-e390e7b55a9f< /ID>;
  <Comment/ >
  <Attributes/ >
  <Access>public< /Access>
  <Type>System.Void< /Type>
  <Name>MatchingDelegate< /Name>
  <Parameters/ >
< /Delegate>

```

Figure 3.6. XML document of a delegate declaration

instance method, the delegate stores a reference to the method's entry point and a reference to an object, called the target, which is the class instance that the method is invoked on. The target of an instance method cannot be a null reference. When a delegate references a static method, the delegate stores a reference to the method's entry point. The target of a static method is a null reference.

3.3.1.4. ClassDefinition Class. *ClassDefinition* class is responsible for storing class related information such as name, access modifiers and managing its members. Events, fields, properties and methods are included in corresponding lists. Different type definitions, *classes*, *interfaces*, *structures* and *enumerations* can be declared by using *ClassDefinition* class. A *CodeTypeDeclaration* object is created from delegate information to use it in a CodeDOM object graph. An example class declaration and corresponding XML document that is generated by BUILD.NET are shown in Figure 3.7 and Figure 3.8, respectively.

```
public class Agv : MovableObject {
}
```

Figure 3.7. C# code of a class declaration

3.3.1.5. EventDefinition Class. *EventDefinition* class represents events that are declared in a class. It simply stores the name and type of the event. Event information is converted into a *CodeMemberEvent* object to use it in a *CodeCompileUnit*. An example event declaration and corresponding XML document that is created by BUILD.NET are shown in Figure 3.9 and Figure 3.10, respectively.

An event is a way for a class to provide notifications to clients of that class when some interesting thing happens to an object. The most familiar use for events is in graphical user interfaces; typically, the classes that represent controls in the interface have events that are notified when the user does something to the control. Events provide a generally useful way for objects to signal state changes that may be useful to clients of that object.

```

<Class>
  <ID>b30112d7-5f52-4922-bd1c-5e45a68f6703< /ID>;
  <Comment/ >
  <Attributes/ >
  <Access>public< /Access>
  <Name>Agv< /Name>
  <Modifier/ >
  <Base>MovableObject< /Base>
  <IsClass>true< /IsClass>
  <IsEnum>false< /IsEnum>
  <IsInterface>false< /IsInterface>
  <IsStruct>false< /IsStruct>
  <Events/ >
  <Fields/ >
  <Methods/ >
  <Properties/ >
< /Class>

```

Figure 3.8. XML document of a class declaration

```
private event System.EventHandler TestEvent;
```

Figure 3.9. C# code of an event declaration

```

<Event>
  <ID>920eb953-68bd-4f45-ac4a-4722784148c0< /ID>;
  <Comment/ >
  <Attributes/ >
  <Access>private< /Access>
  <Type>System.EventHandler< /Type>
  <Name>TestEvent< /Name>
< /Event>

```

Figure 3.10. XML document of an event declaration

Events are declared using delegates. An event is a way for a class to allow clients to give it delegates to methods that should be called when the event occurs. When the event occurs, the delegate(s) given to it by its clients are invoked.

3.3.1.6. FieldDefinition Class. *FieldDefinition* class is responsible for storing information that is required to declare a field properly. This includes field name, field access modifiers, field type and initialization value if it has. *FieldDefinition* class is converted into a *CodeMemberField* object to properly join to a *CodeCompileUnit* while source code generation or compilation. An example field declaration and corresponding XML document that is generated by BUILD.NET are shown in Figure 3.11 and Figure 3.12, respectively.

```
private const int capacity;
```

Figure 3.11. C# code of a field declaration

```
<Field>
  <ID>39f6bc7e-f301-4e62-9596-c499f2ba4455< /ID>;
  <Comment/ >
  <Attributes/ >
  <Access>private< /Access>
  <Type>System.Int32< /Type>
  <Name>capacity< /Name>
  <Modifier>const< /Modifier>
  <Value/ >
< /Field>
```

Figure 3.12. XML document of a field declaration

3.3.1.7. PropertyDefinition Class. *PropertyDefinition* class is used to model property declarations in classes. It has property name, return type, access modifiers and get/set method implementations if they are defined. A *CodeMemberProperty* object is created from property information that are contained in *PropertyDefinition* class. An example

property declaration and corresponding XML document that is created by BUILD.NET are shown in Figure 3.13 and Figure 3.14, respectively.

```
public int Speed {
    get { ... }
    set { ... }
}
```

Figure 3.13. C# code of a property declaration

```
<Property>
  <ID>45c2afb3-a405-47a2-8c39-2f497c008d62< /ID>;
  <Comment/ >
  <Attributes/ >
  <Access>public< /Access>
  <Type>System.Int32< /Type>
  <Name>Speed< /Name>
  <Accessor>get-set< /Accessor>
  <Modifier/ >
  <GetMethod/ >
  <SetMethod/ >
< /Property>
```

Figure 3.14. XML document of a property declaration

A property is a member that provides access to a characteristic of an object or a class. Properties do not denote storage locations unlike fields. Instead, properties have accessors that specify the statements to be executed when their values are read or written. The *get* accessor is called when the property's value is read; the *set* accessor is called when the property's value is written.

3.3.1.8. MethodDefinition Class. *MethodDefinition* class represents method declarations and statements that determine execution context. It has method name, return type, access modifiers, parameters and statements. *MethodDefinition* class is converted into a *CodeMemberMethod* object and *CodeStatement* objects that correspond to state-

ments in that method definition are added to *CodeMemberMethod*. An example method declaration and corresponding XML document that is generated by BUILD.NET are shown in Figure 3.15 and Figure 3.16, respectively.

```
public void ReadLayoutFromXML(in string layoutPath);
```

Figure 3.15. C# code of a method declaration

```
<Method>
  <ID>824d2392-27a4-4682-a23e-d1565fa7314d< /ID>;
  <Comment/ >
  <Attributes/ >
  <Access>public< /Access>
  <Type>System.Void< /Type>
  <Name>ReadLayoutFromXML< /Name>
  <Modifier/ >
  <Parameters>
    <Parameter>
      <ID>367e5258-bae9-4112-8c09-748d5a3e3871< /ID>;
      <Modifier>in< /Modifier>;
      <Type>System.String< /Type>;
      <Name>layoutPath< /Name>;
    < /Parameter>
  < /Parameters>
  <Statements/ >
< /Method>
```

Figure 3.16. XML document of a method declaration

3.3.1.9. ParameterDefinition Class. *ParameterDefinition* class is used to create the parameters of *DelegateDefinition* and *MethodDefinition* classes. It includes parameter name, parameter type and parameter direction. *CodeParameterDeclarationExpression* object is created and added to corresponding *CodeTypeDelegate* or *CodeMemberMethod* object when necessary.

3.3.2. Operation Definition Classes

The program becomes meaningful with the way it implements its operation definitions. Methods and properties consists of statements to perform desired actions. These include variable declarations statements, assignment statements, conditional branches, loops, etc. BUILD.NET models method and property implementations as graphs which are composed of statement nodes and links between nodes represent execution flow. Operation definition classes are introduced to BUILD.NET to construct this kind of execution graphs. Table 3.2 shows the operation definition classes and their roles. Figure 3.17 demonstrates the relationships between operation definition classes.

Table 3.2. Operation definition classes

Class Name	Explanation
Statement	Stores statement information such as expressions
StatementLinkPoint	Connection points of statement nodes
StatementLink	Connects statements, defines execution flow
Expression	Stores expression information such as variable names

3.3.2.1. Statement Class. *Statement* class is the basic building block for operation definitions. As mentioned, method and property implementations are modeled by graphs and *Statement* class represents nodes in the graphs. Different classes are derived for each type of programming language statements. Each statement class is responsible for creating corresponding CodeDOM object that represents itself. Table 3.3 shows statement classes that are defined in BUILD.NET and sample programming language statements for each statement type. UML diagrams of statement classes can be found in Appendix A.

3.3.2.2. StatementLinkPoint Class. In the execution flow, each node has two main connection points which are connected to predecessor node and successor node. Some statements like conditional branches and loops need additional connection points. Con-

Table 3.3. Operation definition classes derived from Statement class and sample programming language statements

Class Name	Sample Statements
AssignStatement	counter = 10; counter = counter + 1;
AttachEventStatement	this.TestEvent += new System.EventHandler(this. TestMethod);
CommentStatement	// This method calculates travel time
ConditionStatement	if(test == true){ } else{ }
ExpressionStatement	Console.WriteLine("Program terminated");
GotoStatement	goto testLabel;
IterationStatement	for(int i = 0; i < count; i++) { }
LabeledStatement	testLabel:
MethodReturnStatement	return result; return;
RemoveEventStatement	this.TestEvent -= new System.EventHandler(this. TestMethod);
SnippetStatement	<i>Any statement</i>
ThrowExceptionStatement	throw new System.Exception();
TryCatchFinallyStatement	try{ } catch(System.Exception ex){ } finally{ }
VariableDeclarationStatement	int counter; Point point = new Point(50, 50);

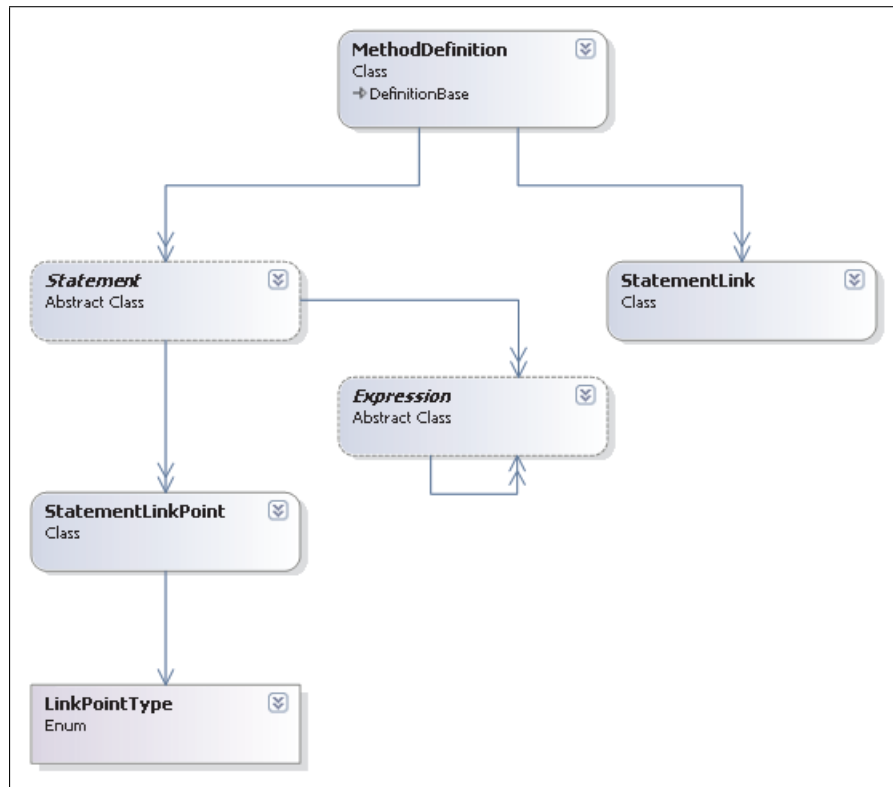


Figure 3.17. UML diagram of operation definition classes

ditional branches need two additional connections: one for true branch and one for false branch. *StatementLinkPoint* class models different connection types for nodes.

3.3.2.3. StatementLink Class. *StatementLink* class is the connection between two statement nodes. *StatementLinkPoint* and *StatementLink* classes do not produce any CodeDOM objects but they determine the structure of object graph. A node with zero in-degree is selected as the starting statement of method or property. All nodes are traversed starting from this node and object graph is constructed properly.

3.3.2.4. Expression Class. A programming language statement is composed of one or more expressions. Nodes in BUILD.NET represent statements, so nodes should be decomposed into smaller elements. *Expression* class models the logical partitions of statements such as array indexing, object creation and casting. Same reasoning, applied to statements, is also applied to expressions and different expression classes are derived from *Expression* class. Each expression class is responsible for creating corresponding CodeDOM object that represents itself. Table 3.4 shows expression classes that are

defined in BUILD.NET and their tasks. UML diagrams of expression classes can be found in Appendix A.

Table 3.4. Operation definition classes derived from Expression class and their tasks

Class Name	Task
ArgumentReferenceExpression	Referencing a parameter of a method
ArrayCreateExpression	Creating an array
ArrayIndexerExpression	Referencing an index of an array
BaseReferenceExpression	Referencing the base class
BinaryOperatorExpression	Connecting expressions with an operator
CastExpression	Casting an expression to target type
DelegateCreateExpression	Creating a delegate
DelegateInvokeExpression	Invoking a delegate
DirectionExpression	Representing direction of a parameter
EventReferenceExpression	Referencing an event
FieldReferenceExpression	Referencing a field
IndexerExpression	Referencing an index of a indexer property
MethodInvokeExpression	Invoking a method
MethodReferenceExpression	Referencing a method
ObjectCreateExpression	Creating a new instance of a type
ParameterDeclarationExpression	Declaring a parameter for a method
PrimitiveExpression	Representing a primitive data type value
PropertyReferenceExpression	Referencing a property
PropertySetValueReferenceExpression	Representing the value argument
SnippetExpression	Representing a literal expression
ThisReferenceExpression	Referencing current local class instance
TypeOfExpression	Returning type of an object
TypeReferenceExpression	Referencing a type
VariableReferenceExpression	Referencing a local variable

3.4. Sample Source Code to Object Graph Conversions

This section provides different examples which show conversion process between source code and object graph. Figure 3.18 and Figure 3.19 show two sample statements and how BUILD.NET decomposes them into expressions. Figure 3.20 shows a complete source code of a program and Figure 3.21 illustrates the object graph that corresponds to this source code.

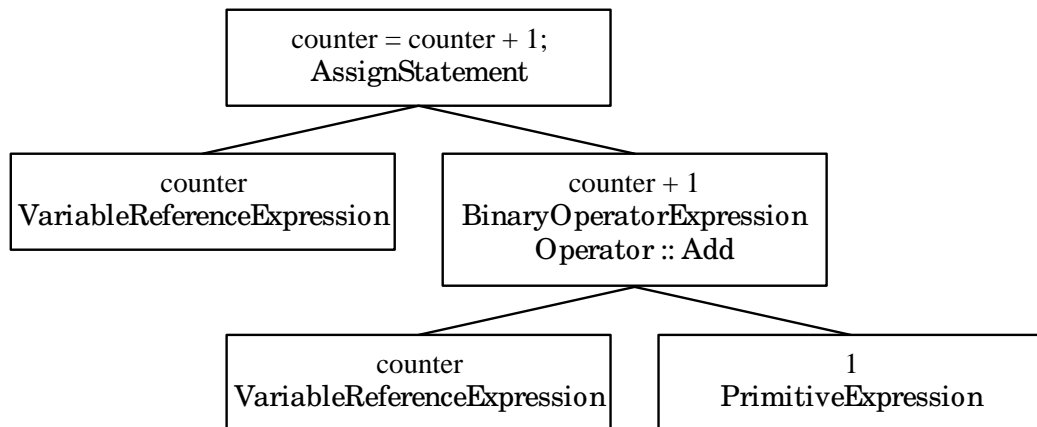


Figure 3.18. Sample statement decomposition for an assignment statement

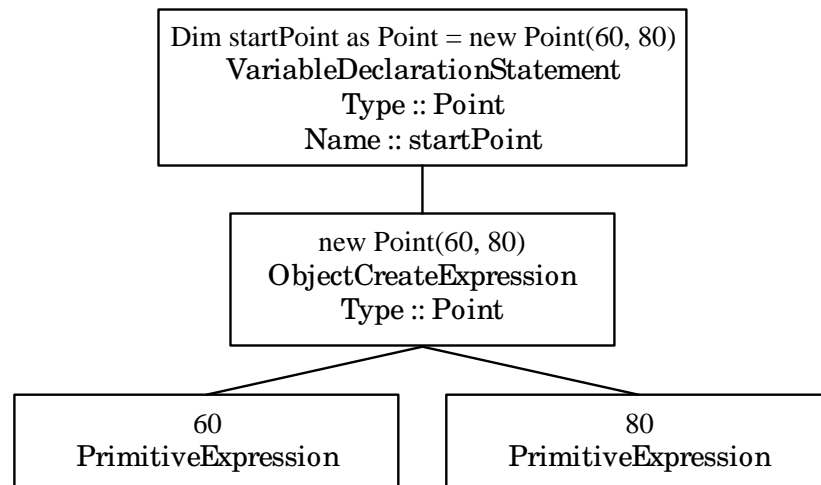


Figure 3.19. Sample statement decomposition for a variable declaration statement

```

namespace Math{
    public class Addition{
        private int a;
        private int b;
        public Addition(int aIn, int bIn){
            this.a = aIn;
            this.b = bIn;
        }
        public int Add(){
            return a+b;
        }
    }
}

```

Figure 3.20. Sample source code for object graph conversion

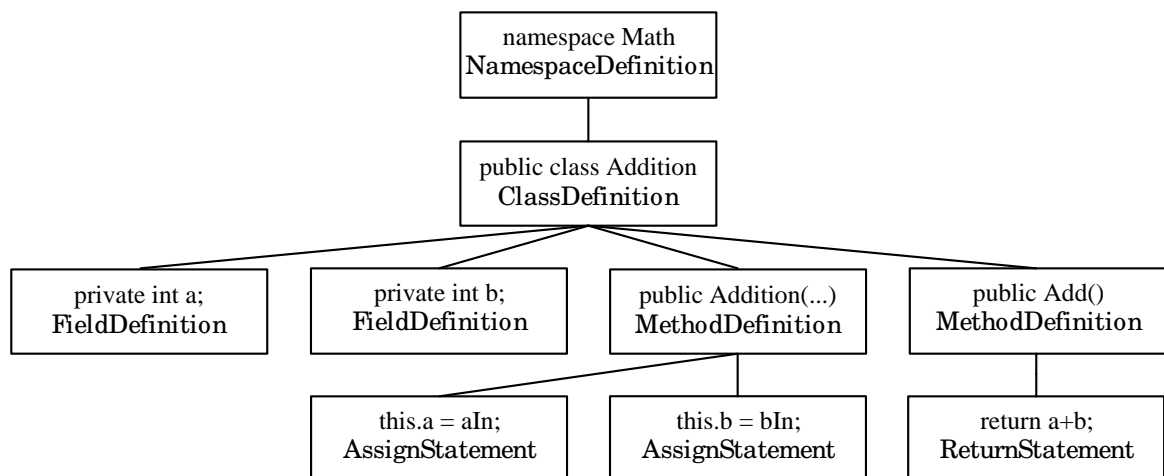


Figure 3.21. Object graph corresponds to sample source code

3.5. Implementation Details

Designed application generator is implemented in C# by using Microsoft.NET Framework. BUILD.NET is decomposed into two main parts: logical layer and graphical user interface (GUI) layer. Logical layer performs the following tasks:

- Saving and loading project files
- Managing objects
- Managing relationships between objects
- Generating source code from object graph
- Generating executable from object graph
- Generating UML class diagrams from class definitions
- Generating documentation from source code

GUI layer performs the following tasks:

- Collecting information from user
- Displaying operation definition diagrams of methods and properties
- Displaying class diagrams of class definitions
- Displaying compilation errors to user

3.5.1. Logical Layer

Logical layer of BUILD.NET is implemented as a ClassLibrary³ in Microsoft Visual Studio.NET. This layer is an independent assembly so that it can be also used in an another program easily. Logical layer consists of classes that are described in Section 3.3 and additional classes derived from *Statement* and *Expression* classes to model CodeDOM objects appropriately.

Logical layer classes are implemented such that they can be serialized and de-serialized. Serialization is the process by which objects or values are converted into

³is used to create reusable classes and components that can be shared with other projects.

a format that can be persisted or transported. Serialization can also be used to save the state of objects from memory to a storage medium such as files or to transport objects and values across the network. To read the state of the objects or values that are persisted or transported using serialization, deserialization, complementary to serialization, is used. The Microsoft.NET Framework supports two types of serialization, binary serialization and XML serialization. By using serialization and deserialization, BUILD.NET can easily save objects to an XML file or load objects from an XML file.

One other task, performed by logical layer, is managing objects in a project and relate these objects properly. Delegates and classes are contained by namespaces, members (events, fields, methods and properties) are contained by classes, statements are contained by properties or methods, statements are connected by statement links that determine the execution flow and finally expressions are contained by statements.

Code generation functionality is distributed to objects. Each object has a method named *GenerateCodeDOM* and creates the corresponding CodeDOM object by itself. The object at the root starts the code generation process and this request is sent through the hierarchy to lower levels. Each object generates CodeDOM object as a response to this request. Formed CodeDOM objects percolate up through the hierarchy and at the topmost level of hierarchy, collection of these object becomes a *CodeCompileUnit*. *ModelDefinition* class has functionality to convert this *CodeCompileUnit* to source code.

Executable generation functionality is achieved by a similar approach. Source code is generated by using same procedure from object graph and compiled. If compilation fails, the compilation errors and warnings are passed to GUI layer to report them to the user.

Logical layer has the class information of project and by using these information, it can create UML class diagrams and passes these diagrams to GUI layer to show them to the user. Logical layer also generates automatic documentation for the written program from source code and comments that are signed as documentation notes.

3.5.2. GUI Layer

GUI layer enables the user to interact with the logical layer objects and consists of wizard forms and some GDI (Graphics Device Interface) related functions that help to draw statement nodes, expression nodes and UML class diagrams. Adding, removing and modifying objects are performed through GUI layer. Namespaces, delegates, classes, fields, properties and methods are created and edited by using wizards. Screenshots of these wizards can be found in Appendix B.

Statement nodes and expression nodes are also managed via wizards. These wizards contain the necessary information to build a statement or an expression from scratch. Input validation is also performed at the time statements and expressions are created to prevent the model having an invalid state. For example, identifier names are checked by using regular expressions to avoid possible syntax errors in generated source code.

GUI layer has another important component, named *MethodExplorer*, which helps to construct operation definitions of methods and properties. *MethodExplorer* can contain statement nodes, statement links and statement link points. The user can describe operational definition of any method by dragging and dropping statement nodes and connecting statement nodes via statement links. *MethodExplorer* allows only valid execution flows to be entered to the system. An invalid connection between two statement nodes can not be inserted. An example BUILD.NET graph which has four different types of statement nodes and connections between them can be seen in Figure 3.22. Each statement node has at least two connection points, one for incoming connection and one for outgoing connection, and some statement nodes such as *ConditionStatement* in Figure 3.22 has additional connection points. Active connection points become black to differentiate them from inactive connection points.

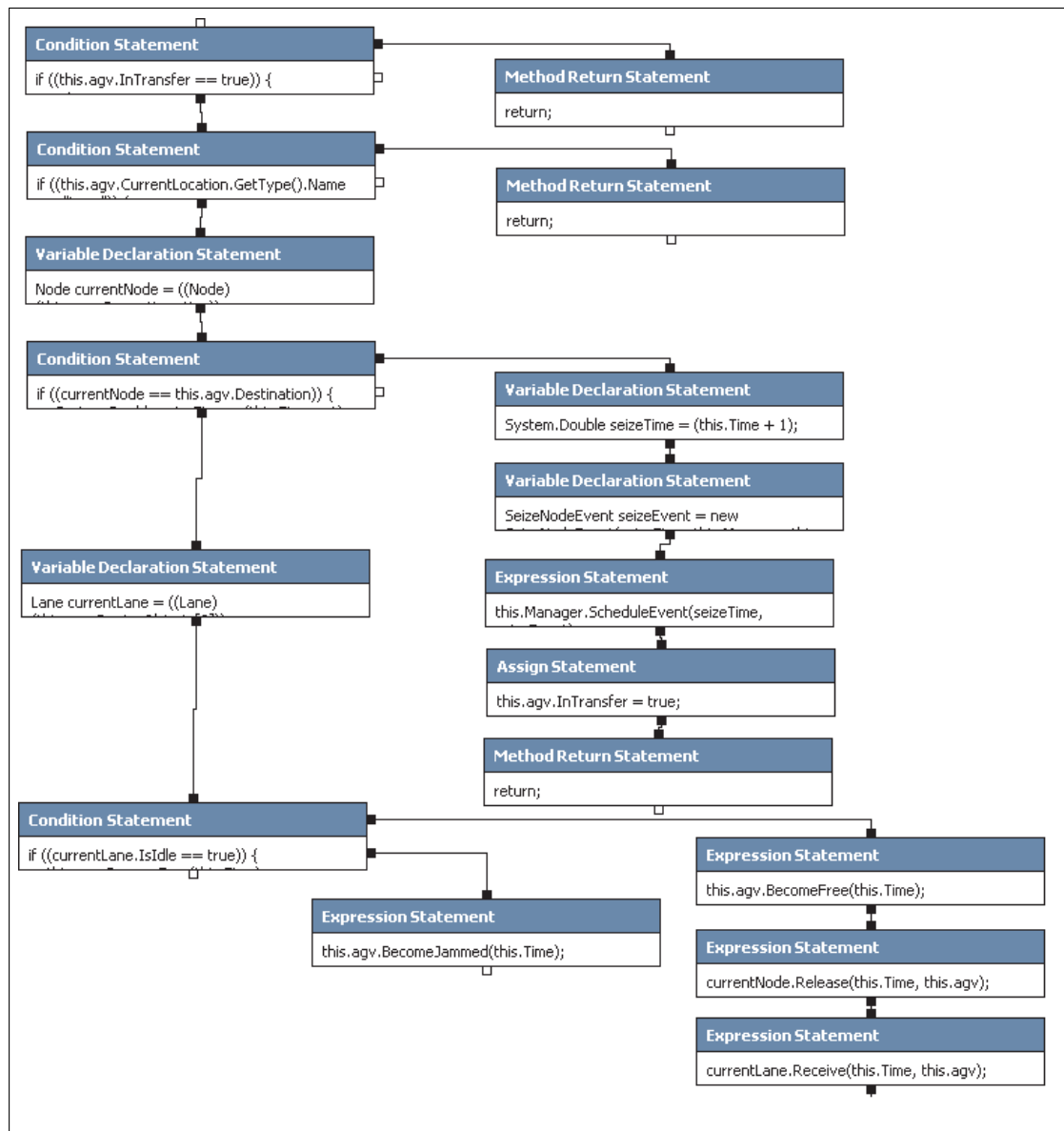


Figure 3.22. Method explorer

3.5.3. Experimentation

After developing a prototype of GUI layer, the program was tested on five different subjects who are not very familiar with programming languages and their opinions about input collection and code generation mechanisms were collected. The subjects were asked to write small programs that implement graph algorithms such as topological sort and shortest path problems.

The main concern of the subjects was seeing generated code right away after adding a statement or an expression. *MethodExplorer* was modified so that corresponding code statements for selected language is displayed on the screen without generating the whole source code.

Another concern of two of the subjects was transferring some part of the project into another project. For example, only a class definition is wanted to be transferred to another project. GUI layer was modified to save a specific structure definition in an XML file and load a structure definition from an XML file to achieve this transfer mechanism.

One of the subjects also proposed a different way to differentiate statements that cause errors after compilation process. After the user selects an error from the list, the statement node that cause this selected error is animated for helping the user to distinguish the cause of the error.

Source code automatically generated by BUILD.NET was compared with manually written source code. There are small differences between them:

- Generated source code has some extra parentheses to guarantee operator precedence.
- Generated source code is well-structured in terms of standard coding style.
- Generated source could not have some programming language constructs such as *while* and *switch* statements due to lack of CodeDOM support.

4. AN APPLICATION: A NEW FMS SIMULATOR (FMS.NET)

Flexible manufacturing systems (FMS) are integrated systems that can help companies achieve the goal of increasing profitability through the increase of productivity. An FMS is composed of an automated material handling system, CNC (Computer Numerical Control) machines and a computer-controlled network that coordinates the processing stations and the material handling system. FMS aim to produce a medium variety of product types at medium volumes. Different product types can be produced simultaneously in an FMS. The flexibility of an FMS, is primarily due to its capability of performing different operations at the same processing station, *machine flexibility*. Another source of the flexibility, *routing flexibility*, is the possibility of producing the same product by alternative sequences of operations and performing a given operation of a product on alternative machines. Although FMS can provide many benefits, its implementation requires a high investment capital. Extensive analysis and careful design must be conducted before implementation to avoid costly mistakes (Li *et al.*, 1998). Also continuous monitoring and intervention is required to operate the existing FMS efficiently to respond changing production and market conditions.

4.1. Flexible Manufacturing Systems Simulation

Mathematical models are not adequate to study the dynamic nature of FMS operation and to capture the many different situations arising from FMS operation. In fact, mathematical models can not capture the congestion effect due to changing conditions in material handling availability and ignore the effects of scheduling and control policies. Conducting simulation studies is more reasonable to capture the stochastic nature of an FMS. Computer simulation is a powerful tool for FMS planning, design and operation. There is a general consensus in the literature that discrete-event simulation is the most popular and appropriate approach for modeling FMS (Borenstein, 2000).

There is a large number of commercially available simulators to evaluate the design and operational decisions of FMS. There are two different approaches for designing FMS simulators: using general programming languages (C++, Fortran) and using simulation languages such as SIMAN (Arena Simulation, 2005) and SIMSCRIPT (SIMSCRIPT II.5, 2005). In addition, commercial simulation packages such as ARENA (Arena Simulation, 2005) and ProModel (ProModel Simulation, 2005) are specifically designed for modeling manufacturing systems.

These simulation languages or packages are suitable for developing FMS models small to moderate size. They have default entities to model FMS and perform statistical analysis. However, major changes in the system model cannot easily be handled. Also the user should rely on vendor description of the algorithms and procedures implemented in the package. Implementing new decision algorithms and integrating them with the simulation package requires vendor support through APIs and user effort. These APIs provided by the vendor may not be sufficient to implement any decision algorithm due to initial system design of the vendor.

First approach, using general programming languages, is preferred to gain more flexibility in designed simulator. In this approach, any future extension to initial design can be performed easily by directly changing source code. Increasing system flexibility causes system complexity to increase. Each entity in manufacturing environment should be modeled in detail as opposed to default entities of general simulation packages.

Doğan (2001) developed an object-oriented discrete-event FMS simulator, called *FLEXIM*, for BUFAIM (Boğaziçi University Flexible Automation and Intelligent Manufacturing Laboratory). *FLEXIM* is used in the research projects of BUFAIM and several undergraduate course projects. *FLEXIM* has been modified and extended to add different decision types, algorithms and system entities by graduate and undergraduate students throughout the years. During this process, it has been observed that achieving continuous improvement, maintaining, documenting and teaching an FMS simulator is a very challenging task and can best be achieved by using an application

generator. An application generator will allow students to concentrate on the system design instead of implementation details. Thus, the development of the system by different teams becomes easier. For this reason, designing an FMS simulator is selected as target application area for the proposed application generator.

4.1.1. Problem Definition

The FMS model can be decomposed into three subcomponents: static model data, dynamic model data and operational decisions.

4.1.1.1. Static Model Data. The static model data describe the operational environment of FMS and consists of the following entities:

- Material Handling System
 - Automated Guided Vehicle (AGV)
 - Lane
 - Node
- Machining System
 - Workcenter
 - Queue
 - Operation

AGV is a battery powered driverless cart with programming capabilities for path selection and positioning. The following properties of the vehicles are important for modeling the system properly:

- Capacity
- Speed
- Parking position
- Breakdown and repair distribution
- Load/unload transfer time

Vehicles in the system travel on an actual or virtual flow path. Flow path layout is composed of lanes and nodes. Nodes define starting and ending positions of lanes and transfer positions. Nodes are classified as path nodes, park nodes and transfer nodes. Path nodes are the intersection points of lanes. Park nodes are the parking positions of empty AGVs. Transfer nodes are the points where transfers between material handling system and machining system - pickups and deliveries - occur. Lanes and nodes are used in zone control mechanisms to prevent collisions. A lane or a node can be used by at most one vehicle at a time. Figure 4.1 shows a sample flow path layout.

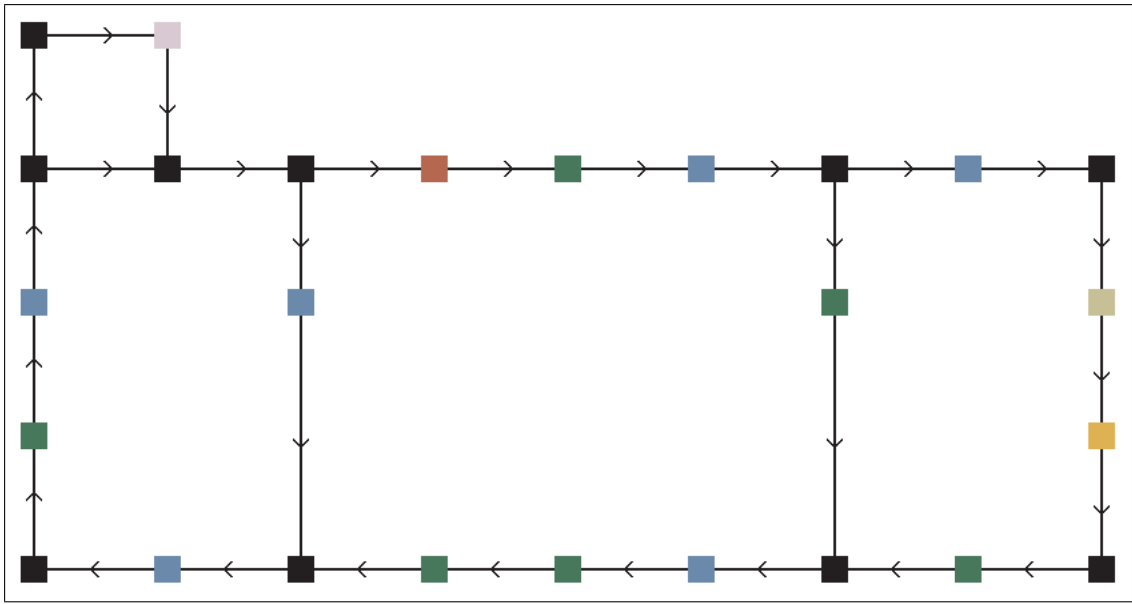


Figure 4.1. Sample flow path layout

Workcenters are machining centers with identical processors. Workcenters can be grouped into four categories: receival, shipment, processing and central buffer. Receival and shipment workcenters are locations where orders enter to and leave from the system, respectively. System can have more than one receival and shipment workcenters. Processing workcenters perform operations on orders. Central buffer is a temporary storage for loads on vehicles if these loads can not be transferred to their destinations because of blockage. The following properties of the workcenters are parameters:

- Number of processors
- Workcenter type
- Input and output queue

- Breakdown and repair distributions of processors
- Transfer time from/to queue

Queues are the storage and transfer positions of workcenters. Material handling devices deliver/pickup loads to/from queues. These locations are also used as storage location for finished and waiting loads.

Operations are the basic processes performed in workcenters. An operation can be performed in different workcenters with different processing times. Processing time of an operation is the same for all processors of a workcenter.

4.1.1.2. Dynamic Model Data. Dynamic model data describe product mix that is produced in an FMS and consists of job definitions. Job definitions give information about each job type that the FMS is capable of producing. Following parameters form a job definition:

- Alternative operation routes
- Interarrival time distribution
- Batch size distribution

Jobs can be produced by using different operation sequences. Directed graphs can be used to represent alternative operation routes of a job type. Each directed path from reception operation to shipment operation in the directed graph corresponds to a unique operation route for the job type. Figure 4.2 shows a directed graph representation of alternative operation routes of a job type and the linearized routes from this graph.

Interarrival time distribution can be specified by using standard statistical distributions or order arrivals taken as input data from high level planning agents.

4.1.1.3. Operational Decisions. Operational decisions are rules that define how the system reacts to events. There are seven basic operational decision types:

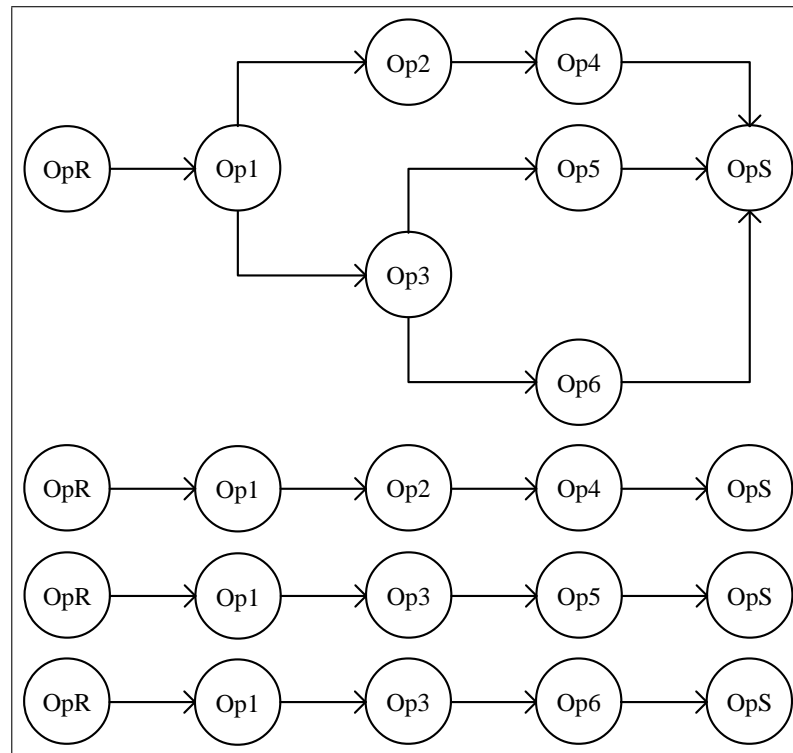


Figure 4.2. Alternative operation routes of a job type

- Blockage Solving
- Matching
- Dispatching
- Routing
- Traffic Management
- Operation Decision
- Process Order

Blockage Solving is used to avoid blockages in the system. If an AGV arrives to its destination to deliver a unitload and the input queue of the workcenter is full, AGV becomes blocked. If no action is taken, this may result in system deadlock depending on the layout.

Matching is the process of pairing AGVs and unitloads that are required to be transported. A unitload is transferred to output queue when its operation is completed in current workcenter and this unitload should be transferred to another workcenter. This unitload is assigned to one of the AGVs which have empty position. If there is

no eligible AGV, this unitload is not assigned to any AGV and waits for an AGV to become eligible.

Dispatching is the process of deciding the destination of an AGV. After an AGV arrives to its destination and completes delivery or pick-up tasks, its new destination should be provided. This is also required when a unitload is assigned to an AGV.

Routing is the process of selecting a route between an AGV's current location and its destination.

Traffic Management is the process of selecting an AGV from jammed AGVs when a node becomes free. AGVs may become jammed if they want to enter an occupied location. If more than one AGVs are jammed and they want to enter the same node, a selection must be made to decide which AGV will move into the node when the node becomes empty.

Operation Decision is the process of deciding next operation for a unitload. When an operation is completed, next operation for this unitload is selected by using alternate operation routes of the unitload.

Process Order is the process of deciding which unitload is processed first. When a processor becomes empty, one of the unitloads that are in input queue and wait for the processor is selected and transferred to processor.

4.2. Design of FMS.NET

FMS.NET needs three different information, namely layout definition, job mix and simulation parameters, about FMS to simulate it properly. Layout definition consists of properties of material handling and machining systems. Job mix specifies job definitions that are produced in this FMS. Simulation parameters are composed of selected decision algorithm names and run parameters that will be used in a given simulation experiment such as end time, seed and warmup time. Figure 4.3 shows

the basic input and output entities in the system and illustrates the operation of the simulation system.

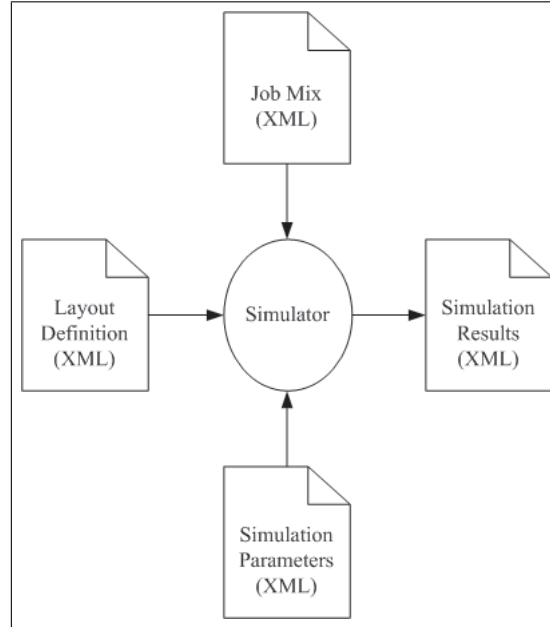


Figure 4.3. FMS.NET framework

Designed FMS simulator should have at least three basic layers to model FMS adequately as described in problem definition. FMS.NET layers are designed as separate namespaces and each namespace is analyzed in detail.

Table 4.1. FMS.NET namespaces

Namespace Name	Explanation
FMS.NET	Contains basic object definitions
FMS.NET.Decision	Contains operation decision implementations
FMS.NET.Layout	Contains static model data objects
FMS.NET.Operation	Contains dynamic model data objects and simulation related objects
FMS.NET.Random	Contains random variate generators

4.2.1. FMS.NET.Random Namespace

L'Ecuyer (2001) stated that the availability of multiple independent streams of random numbers is a must in a general-purpose discrete-event simulation environment. Multiple streams simplify the application of certain variance reduction techniques and

are also useful for simulation on parallel processors. One way of implementing such multiple streams is to compute seeds that are random numbers generated by using one common seed and to use these generated seeds as starting seeds in different streams. However, the independence of these streams is not guaranteed. The other and reasonable way of implementing multiple streams is to compute seeds that are spaced far apart in the random number stream and to use the random number stream subsequences starting at these seeds as if they were independent streams. These streams are considered as distinct independent random number streams.

L’Ecuyer (2001) constructed a new random number generator package with multiple streams by using the second approach and this implementation is now used in ARENA and AutoMod simulation environments. Random variate generators which use L’Ecuyer’s random number generator reside in this namespace. Following random variate generators are included in *FMS.NET.Random* namespace:

- Exponential (λ)
- Fixed (a)
- Normal (μ, σ)
- Triangular (a, b, c)
- Uniform (a, b)
- Weibull (α, β)

Classes that are included in *FMS.NET.Random* namespace are listed in Table 4.2. UML class diagrams of random variate generator objects can be found in Appendix C.

4.2.2. FMS.NET Namespace

This namespace consists of basic object definitions that are required for FMS simulation. These are *Statistics*, *FMSObject*, *StaticObject* and *MovableObject*. Classes that are included in *FMS.NET* namespace are listed in Table 4.3. UML class diagrams of these four classes can be found in Appendix C.

Table 4.2. FMS.NET.Random namespace classes

Class Name	Base Class	Explanation
RVGenerator	-	Base class for random variate generators
ExponentialRVGenerator	RVGenerator	Exponential random variate generator
FixedRVGenerator	RVGenerator	Fixed random variate generator
NormalRVGenerator	RVGenerator	Normal random variate generator
TriangularRVGenerator	RVGenerator	Triangular random variate generator
UniformRVGenerator	RVGenerator	Uniform random variate generator
WeibullRVGenerator	RVGenerator	Weibull random variate generator

Table 4.3. FMS.NET namespace classes

Class Name	Base Class	Explanation
FMSObject	-	Base class for all simulation objects
MovableObject	FMSObject	Moving entities in FMS
StaticObject	FMSObject	Static entities in FMS
Statistics	-	Statistics object for performance measures

Statistics object is used to collect information about system performance and to calculate statistics about collected performance measures such as standard deviation, minimum, maximum and mean values. *Statistics* object can hold three different types of information: count based statistics (submitted job count, completed job count, etc.), average based statistics (average flow time, average travel time, etc.) and time-weighted average based statistics (average queue length, average utilization of workcenters, etc.).

FMSObject is the base class for all objects in simulation package. It has common properties such as *ID*, *Name*, *Statistics* dictionary and *Parent*. *ID* is used to differentiate an object from others and it is a 128-bit globally unique identifier. *Statistics* dictionary stores key-value pairs where key holds statistics name and value holds corresponding statistics object. *FMSObject* can access to its statistics objects by their names and update them. *Parent* property is used to build a hierarchy between different objects.

StaticObject is used to represent static objects in FMS such as nodes, lanes, queues and workcenters. It has three important properties: *Capacity*, *Reserved* and *Content*. *Capacity* is the number of total slots in the object and *Reserved* is the number of reserved slots. *Content* property is a linked-list that stores the objects that are currently in the used slots.

MovableObject is used to model moving objects in FMS such as AGVs and unit-loads. *CurrentLocation* and *EntryTime* are two properties that are used in simulation. *CurrentLocation* gives current location object of the moving object and *EntryTime* gives time value at which moving object entered to *CurrentLocation*.

4.2.3. FMS.NET.Layout Namespace

Static model data of FMS are modeled by using objects in this namespace. The classes defined here hold two types of information: static data and dynamic data. Static data define FMS configuration and it is given as an initial parameter to the simulator. Dynamic data are formed at run-time to simulate the dynamic nature of FMS. For example, AGVs and workcenters are empty at the start of the simulation but their contents are changing throughout the simulation. Static data of these classes are signed as serializable to store and retrieve them as XML documents. Classes that are included in *FMS.NET.Layout* namespace are listed in Table 4.4. UML class diagrams of these classes can be found in Appendix C.

Layout is the main class that represents the whole FMS layout. It has linked-lists to store AGVs, lanes, nodes, operations and workcenters. Layout class has utility functions to connect the layout objects appropriately. Nodes and lanes are connected to determine AGV paths, nodes and queues are connected to determine transfer points, AGVs are initialized at their park nodes. Layout class is also responsible for calculating all routes in the system at the start of the simulation and store them in corresponding route tables.

Table 4.4. FMS.NET.Layout namespace classes

Class Name	Base Class
Agv	MovableObject
Lane	StaticObject
Layout	FMSObject
Node	StaticObject
Operation	FMSObjet
Queue	StaticObject
Route	FMSObject
Workcenter	StaticObject

AGV class has properties of AGVs that are listed in problem definition and some dynamic properties such as *Assigned*, *Content*, *Destination*, *Route*. *Assigned* linked-list contains unitloads that are matched to AGV but not picked up yet. *Content* linked-list stores unitloads that are currently on AGV. *Destination* is the next delivery or pickup position of AGV and *Route* stores the path that is selected to travel to *Destination*. AGV class also traces breakdown and repair operations and calculates next breakdown time as the simulation progress. AGV class has some important utility functions: *GetDelivery*, *GetPickup*, *MakeMatch*, *Receive* and *Release*. *GetDelivery* method returns the unitloads that are destined to *CurrentLocation*. *GetPickup* methods retrieves the unitloads that are assigned to AGV and can be picked up from *CurrentLocation*. *MakeMatch* method inserts a unitload to *Assigned* list. *Receive* method inserts a unitload to *Content* list and removes it from *Assigned* List. *Release* method removes a unitload from *Content* list. *MakeMatch*, *Receive* and *Release* methods also update AGV statistics.

Lane class represents flow paths in the FMS layout and stores lane length, starting and ending nodes. *Begin* and *End* properties gives starting and ending nodes of the lane. *Lane* class also has *Receive* and *Release* methods. These methods are used to model AGV entrances and AGV exits.

Node class represents intersection points of lanes. A node may be ending node or starting node of more than one lanes. So, *Ending* and *Starting* linked-lists store lanes ending at node and lanes starting from node, respectively. Transfer nodes are the transfer points between AGVs and queues, so, *Node* class has *InQueue* and *OutQueue* properties to have this connection. Routes in flow path layout are also stored in node classes. The reason is that routing algorithms need table lookups to find routes between two locations. If these routes are stored in one big table, look up time is substantially increased. In order to decrease look up time, routes are distributed to nodes. Each node stores all routes that start from itself in a dictionary like data structure. *Node* class has *Receive* and *Release* methods that are similar to *Receive* and *Release* methods of *Lane* class.

Operation class has alternate workcenters list and processing time distribution of operation on each alternative workcenter. *GetProcessTime* and *GetExpectedTime* methods retrieves the processing time and expected processing time for an operation at a given workcenter. Processing time is used as the simulation progress and calculated by using corresponding random variate generator. Expected processing time is used in process order and operation decision algorithms to get an estimated finishing time.

Queue class is basically a list of waiting unitloads. It has *Receive* and *Release* methods to model unitload entrances and unitload exits.

Workcenter class has properties of workcenters that are listed in problem definition and some dynamic properties such as *Processors* and their contents. *Processors* is a linked-list structure to represent identical processors. Working processors have unitloads in their slots but free processors have nothing as their contents. *Receive* and *Release* methods are similar to *Receive* and *Release* methods of *Queue* class.

4.2.4. FMS.NET.Decision Namespace

This namespace has seven classes and seven delegates for operational decisions. Each of classes and delegates corresponds to one of the operational decisions. Each

class has static methods that represent the algorithms that are provided in simulation package. Algorithms that are included in *FMS.NET.Decision* namespace are listed in Table 4.5. UML class diagrams of delegates and classes in this namespace can be found in Appendix C.

Table 4.5. Algorithms in FMS.NET.Decision namespace

Decision Class	Algorithm Names
BlockageSolving	No Action Output Buffer Availability
Matching	Nearest Pickup Minimum Remaining Output Queue Space/Nearest Pickup Random
Dispatching	Nearest Active Station Random
Routing	Shortest Path Shortest Path Without Blockage Random
TrafficManagement	First Come First Served Random
OperationDecision	Earliest Expected Finish Time Smallest Queue Workload Random
ProcessOrder	First Come First Served Shortest Process Time Random

4.2.4.1. Blockage Solving Algorithms. *No Action* algorithm, as the name implies, does not take any action as a response to AGV blockage. *Output Buffer Availability* algorithm checks the output queue of the corresponding workcenter. If the output queue has available space, no action is taken and the blocked AGV waits at the current node for the input queue to become available. If the output queue has no available space, the blocked AGV is dispatched to another destination.

4.2.4.2. Matching Algorithms. *Nearest Pickup* algorithm selects a unitload from the output queue that is nearest to the eligible AGV. *Minimum Remaining Output Queue Space/Nearest Pickup* algorithm assigns a unitload from the output queue that has minimum number of free slots to the nearest eligible AGV. *Random* algorithm selects a unitload from all output queues that have waiting unitloads randomly to the eligible AGV.

4.2.4.3. Dispatching Algorithms. *Nearest Active Station* algorithm considers the delivery stations of unitloads that are on the AGV and the pickup stations of unitloads that are assigned and selects the nearest station as the destination. *Random* algorithm selects a random destination from all possible candidate workcenters.

4.2.4.4. Routing Algorithms. *Shortest Path* algorithm selects the shortest route between the current location and destination node. *Shortest Path Without Blockage* algorithm selects the shortest route which currently has no blocked lanes or nodes between the current location and destination. *Random* algorithm selects a random path from all possible routes.

4.2.4.5. Traffic Management Algorithms. *First Come First Served* algorithm moves the AGV which arrived earliest to clear off a jammed junction. *Random* algorithm randomly selects one of the jammed AGVs to move.

4.2.4.6. Operation Decision Algorithms. *Earliest Expected Finish Time* algorithm calculates the expected finish time for all possible alternative operations and alternative operation with the smallest expected finish time is selected as next operation. *Smallest Queue Workload* algorithm compares the workload of candidate input queues of alternative operations and alternative operation with the smallest input queue workload is selected as next operation. *Random* algorithm assigns a random operation from alternative operations.

4.2.4.7. Process Order Algorithms. *First Come First Served* algorithm takes the first arrived unitload from input queue to the released processor. *Shortest Process Time* algorithm assigns the unitload with smallest expected processing time to the released processor. *Random* algorithm picks a unitload from input queue randomly.

Doğan (2001) explains the mentioned operational decision algorithms and some other in detail except *Operation Decision Algorithms*. Bilge and Albey (2004) perform a comparison between *Operation Decision Algorithms* and give insights about *Earliest Expected Finish Time* algorithm.

4.2.5. FMS.NET.Operation Namespace

This namespace contains the classes that are required for simulation. There are three different categories of classes in this namespace: management-related classes, event-related classes and job-related classes. The most of these classes, event-related and job-related ones, are created at run time when they are needed and destroyed when they are no longer needed. UML class diagrams of these classes can be found in Appendix C.

Table 4.6. Job-related classes in FMS.NET.Operation namespace

Class Name	Base Class
Job	FMSObject
JobDefinition	FMSObject
JobMix	FMSObject
JobRoute	FMSObject
Unitload	MovableObject

Job class is used to model orders that are submitted to simulated FMS. It has *ArrivalTime*, *Completed*, *Unitloads*, *BatchSize* and *JobDefinition* properties. *ArrivalTime* is used to calculate flow time of job when it is departing. *Completed* and *Unitloads* linked-lists are responsible for manipulating finished unitloads and in-progress unitloads, respectively. *BatchSize* is the number of unitloads that belong to job.

JobDefinition classes are the types of jobs that are defined in job mix. It has properties of job definitions that are defined in problem definition. It has also two utility functions to trace job arrivals and job departures: *CreateJob* and *DisposeJob*. *CreateJob* method creates a *Job* object with given parameters and update submitted statistics. *DisposeJob* method calculates job-related statistics and updates modified *Statistics* objects.

JobMix class is a collection of job types that can be produced by the simulated FMS. *JobDefinitions* linked-list holds these job types as *JobDefinition* classes.

JobRoute class defines an alternate operation route for a job type. *Operations* linked-list stores an alternate operation route from receipt operation to shipment operation.

Unitload class represents single operational entities of jobs. *Unitload* class holds valid alternate operation routes of its job type in *Alternates* linked-list when it is created. Invalid operation routes are removed from *Alternates* linked-list by *Complete-Operation* method when one operation is completed. Completed operations are stored in *Completed* linked-list.

Discrete-event simulators have an event-calendar whose events are sorted in increasing time order. Simulator executes the event with the minimum time, this event can cause new events to be created and simulator add these newly created events to its event calendar. Execution goes on like that till simulation end time is reached. The events that are used in FMS.NET are listed in Table 4.7.

Event class is the base class for all events and has two important properties: *Manager* and *Time*. *Manager* represents simulation manager and *Time* represents execution time of event. It has two abstract methods: *TraceEvent* and *Execute*. Each event defines its execution context in *Execute* method. *TraceEvent* method is used to record information about each event in system trace file.

Table 4.7. Event-related classes in FMS.NET.Operation namespace

Class Name	Base Class	Main Actor Class
Event	-	-
ArrivalEvent	Event	JobDefinition
DepartureEvent	Event	Job
EndLoadEvent	Event	Unitload
EndMoveEvent	Event	Agv
EndProcessEvent	Event	Unitload
EndSimulationEvent	Event	-
EndUnloadEvent	Event	Unitload
EndWarmupEvent	Event	-
ReleaseProcessorEvent	Event	Unitload
SeizeNodeEvent	Event	Agv
StartMoveEvent	Event	Agv
StartProcessEvent	Event	Unitload

ArrivalEvent class is used to model job arrivals to FMS. Its execution context is simply creating a *Job* object and inserting this new job and its unitloads to corresponding data structures. *ArrivalEvent* also creates a new *ArrivalEvent* object by using the interarrival time distribution of *JobDefinition* of the arrived job.

DepartureEvent class is responsible for job departures. It removes departing job and its unitloads from corresponding data structures.

EndLoadEvent class models the end of loading activity of an AGV. Target unitload is removed from output queue and placed on AGV. If there is another loading or unloading activity at this transfer node, an *EndLoadEvent* or an *EndUnloadEvent* is scheduled. Otherwise a *StartMoveEvent* is scheduled. If workcenter of output queue has blocked processor(s), a *ReleaseProcessorEvent* is scheduled.

EndMoveEvent class performs the movement of an AGV from a lane to a node according to *Route* object of AGV. Target AGV is removed from lane and inserted

into node. After the *EndMoveEvent* execution, a *SeizeNodeEvent* is scheduled if AGV arrived to its destination. Otherwise a *StartMoveEvent* is scheduled. If there is an AGV waiting to enter to emptied lane, a *StartMoveEvent* is scheduled.

EndProcessEvent class ends the current operation of target unitload. If next operation can not be performed on current workcenter, a *ReleaseProcessorEvent* is scheduled. If current workcenter is also selected for next operation, a *StartProcessEvent* is scheduled.

EndSimulationEvent class clears the event-calendar and forces the simulator to stop. This event also performs final operations such as converting statistics in a suitable format before the results are reported.

EndUnloadEvent class models the end of unloading activity of an AGV. Target unitload is removed from AGV and placed in input queue. If there is another loading or unloading activity at this transfer node, an *EndLoadEvent* or an *EndUnloadEvent* is scheduled. Otherwise a *StartMoveEvent* is scheduled.

EndWarmupEvent class clears the statistics after the transient period to apply steady state analysis.

ReleaseProcessorEvent class takes target unitload from processor and places it to output queue. If input queue of released processor has waiting unitload(s), a *StartProcessEvent* is scheduled. A *DepartureEvent* is scheduled, if one of the operation routes of unitload is completed.

SeizeNodeEvent class models the docking activity of an AGV. An AGV docks to a node if AGV has a transfer activity at this node or AGV is jammed at this node due to traffic. If there is a transfer activity at docked node, an *EndLoadEvent* or an *EndUnloadEvent* is scheduled.

StartMoveEvent class performs the movement of an AGV from a node to a lane according to *Route* object of AGV. Target AGV is removed from node and inserted into lane. An *EndMoveEvent* is scheduled for target AGV. If there are AGV(s) waiting to enter to emptied node, an *EndMoveEvent* is scheduled.

StartProcessEvent class starts the current operation of target unitload. An *EndProcessEvent* is scheduled for target unitload. If there is an AGV waiting to deliver a unitload to input queue of starting processor, an *EndUnloadEvent* is scheduled.

Figure 4.4 shows the relationships between events and operational decision algorithms. All possible sources for event creation and algorithm execution can be traced from the figure. Rectangles and diamonds represent event execution and algorithm execution, respectively.

Table 4.8. Management-related classes in FMS.NET.Operation namespace

Class Name	Base Class
JobManager	FMSObject
LayoutManager	FMSObject
SimulationManager	FMSObject
SimulationParameter	-

JobManager class is responsible of managing job mix that is given as input parameter to simulator. It has a utility function, named as *ReadJobDefinitionsFromXML*, which deserializes and prepares a job mix for simulation. *JobManager* is also managing all jobs and all unitloads that are currently in the system by using *Jobs* and *Unitloads* linked-lists.

LayoutManager class is managing FMS layout and performs initialization tasks for *SimulationManager*. This includes deserialization of layout information by calling *ReadLayoutFromXML* method and preparing layout object for simulation by calling *PrepareLayout* method.

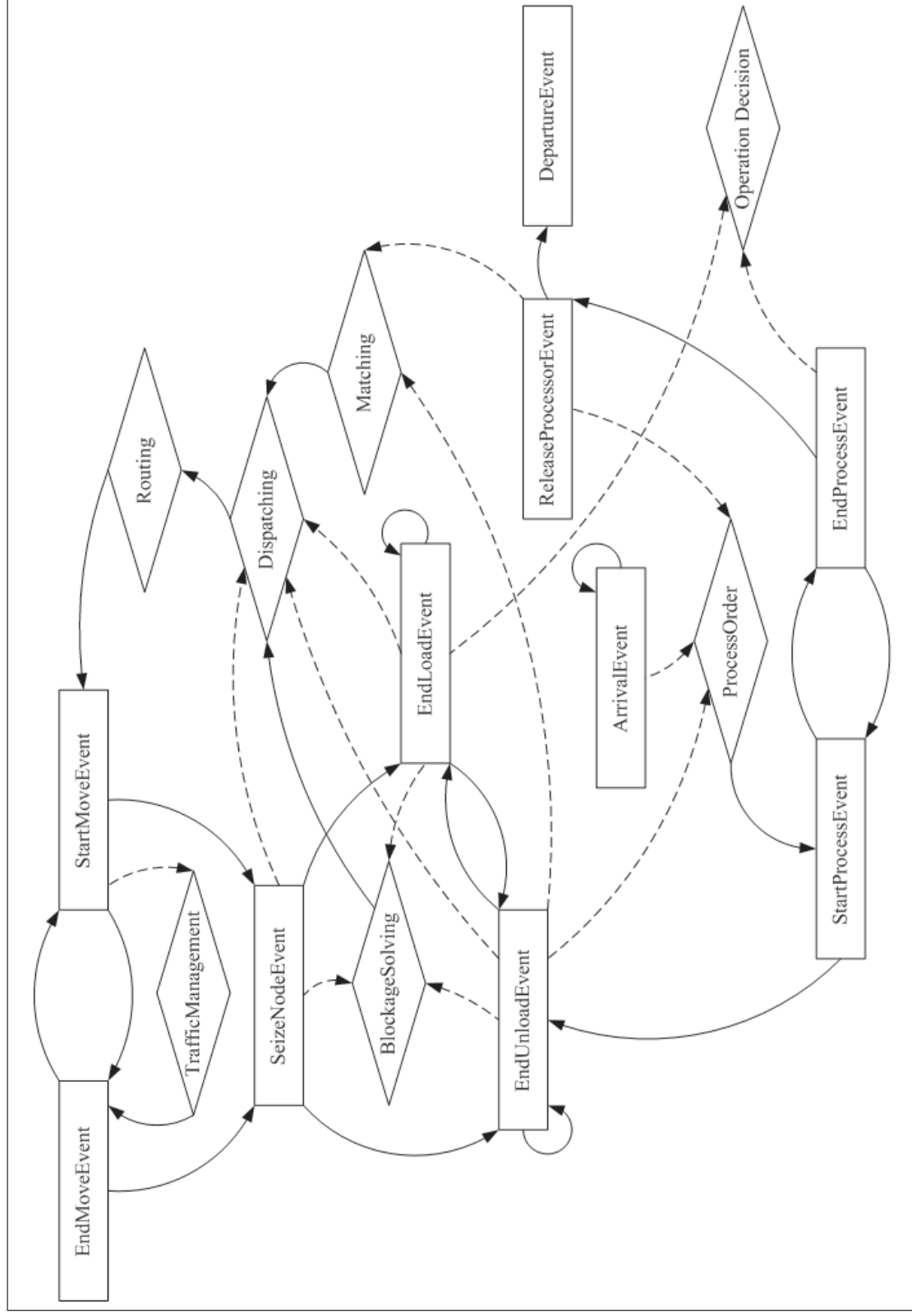


Figure 4.4. System workflow in an FMS

SimulationManager class is the main executing unit of proposed simulator. It contains *EventCalendar*, *JobManager*, *LayoutManager*, delegates for operational decision algorithms and simulation parameters. Simulation parameters are supplied to constructor of *SimulationManager* as a *SimulationParameter* object. *SimulationManager* initializes *LayoutManager* from XML file of layout definition and *JobManager* from XML file of job mix. After this step, operational decision algorithms are connected to *SimulationManager* through delegates and simulation starts.

SimulationParameter class is used as an configuration object. It includes file paths of layout definition and job mix, selected operational decision algorithm names, warmup time, final arrival time, simulation end time and seed.

4.3. Implementation of FMS.NET

Simulation framework is implemented as three separate programs: simulation package, editor program and experimentation program. Figure 4.5 illustrates the information flow between these three programs. Designed FMS simulator, FMS.NET, was implemented by using BUILD.NET. All necessary namespaces, delegates, classes, fields, properties and methods are declared by using definition wizards of BUILD.NET. Property and method implementations are constructed by using drag-and-drop mechanism. Simulation package is compiled and converted into a DLL.

Operational decision algorithms are connected to simulation objects through delegates. Each user can implement new decision algorithms and integrate them into FMS.NET easily by using BUILD.NET. New algorithms should be created in corresponding classes, if so, they become eligible for simulation package automatically.

Editor program which enables users to draw an FMS layout and to describe properties of system elements is implemented separately. The editor collects flow path layout information such as node positions and lanes in the system from the user. The user can specify machining system information after describing material handling system. Workcenters, queues and operations are added to the defined system to describe

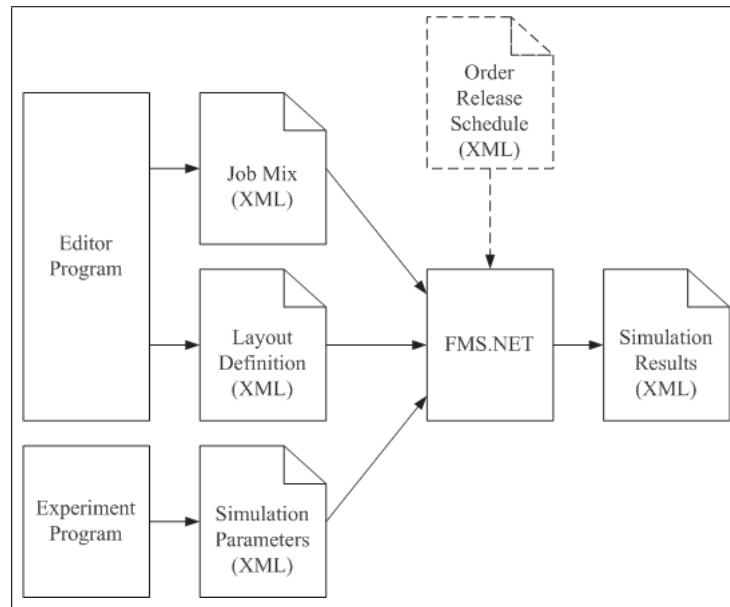


Figure 4.5. Simulation framework information flow

processing environment. Editor is also responsible of collecting job mix information. The user can create a new job mix by using operations that are defined in the FMS layout. Editor converts these information into two different XML files. One of them is layout definition XML, the other one is job mix XML. Experimentation program that collects simulation parameters and animates the simulation progress is also developed separately. This program also stores simulation parameters as an XML file. Simulator can accept an order release schedule XML to generate arrivals instead of using interarrival distributions. Real life scenarios can be experimented to see the performance of the designed FMS.

4.4. Experimentation

After the implementation, FMS.NET was tested to validate the design and the implementation. Simple experiments were conducted and they showed that FMS.NET reproduces the general results that are reported in the literature.

A hypothetical FMS environment whose flow path layout and workcenters are shown in Figure 4.6 was used in the experimentation of FMS.NET. There are six processing workcenters each with certain capacity of one, input queue and output queue

capacities of three. There are one receival workcenter and one shipment workcenter where jobs arrive to and leave the system, respectively. One central buffer workcenter is used to store blocked jobs temporarily.

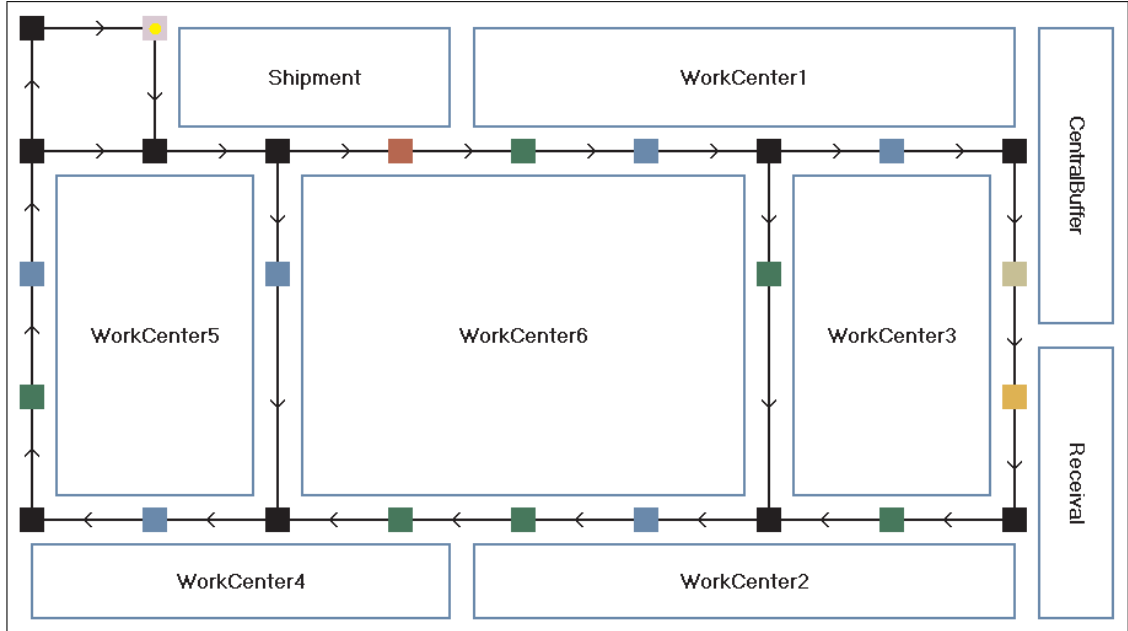


Figure 4.6. FMS layout used in experiments

Possible workcenters for each operation along with corresponding processing times are given in Table 4.9 and the processing times shown in parenthesis belong to their secondary machines. The job set that is produced in the described FMS is listed in Table 4.10.

Each simulation experiment has a duration of 21 hours and warmup time is specified as five hours. The statistical results belong to the last 16 hours after five hours transient period. Each run is replicated five times with different initial seeds (46, 93, 281, 1758 and 2064). Table 4.11 lists decision algorithms used in experiments. The performance measures evaluated are the number of outputs and the average flow time of the jobs.

Table 4.9. Operations used in experiments and their processing times

Operation Name	WC1	WC2	WC3	WC4	WC5	WC6
Operation1	240	-	-	-	-	-
Operation2	-	-	180	-	-	(200)
Operation3	(160)	-	150	-	-	-
Operation4	-	200	-	-	-	-
Operation5	-	-	-	300	-	-
Operation6	(310)	-	-	-	280	-
Operation7	-	-	-	-	-	140
Operation8	-	(440)	400	-	-	-
Operation9	-	-	400	-	-	-
Operation10	-	-	-	(420)	380	-
Operation11	-	-	-	-	-	340

Table 4.10. Job set used in experiments

Job Name	Operation Routes
JobA	Operation2 - Operation6 - Operation7
JobB	Operation4 - Operation10
JobC	Operation1 - Operation8
JobD	Operation3 - Operation4 - Operation6 - Operation5
JobE	Operation9 - Operation5 - Operation11

Table 4.11. Decision algorithms used in experiments

Decision Class	Algorithm Name
Blockage Solving	Output Buffer Availability
Matching	Nearest Pickup
Dispatching	Nearest Active Station
Routing	Shortest Path
Traffic Management	First Come First Served
Operation Decision	Earliest Expected Finish Time
Process Order	First Come First Served

4.4.1. Experiment#1

Determining AGV fleet size is one of the important decisions of FMS design. There are numerous mathematical models in then literature to estimate AGV requirement. These models assume a deterministic demand structure and do not take dynamic behaviour of FMS such as breakdowns and blockages into account. Simulation can be used to estimate AGV fleet size more accurately.

FMS.NET is used to simulate the hypothetical FMS described above. Secondary machine alternatives of operations are not used and five different randomly generated shift orders are submitted to the system. Each AGV has one capacity which means it can carry or travel to pick up only one unitload at a time. Figure 4.7 and Figure 4.8 show the results of the simulation runs. Completed job count is increasing with the increasing number of AGVs. Average flow time exhibits an opposite behaviour. AGV fleet with four, five and six AGVs achieve very close results. Cost factor comes into play in deciding among these alternatives.

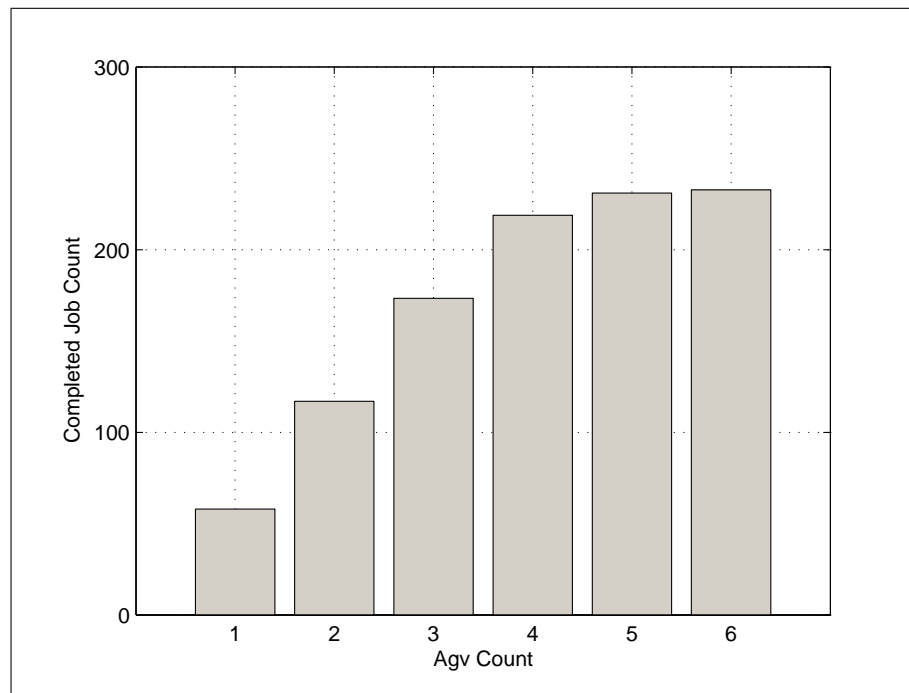


Figure 4.7. Completed job count with changing AGV fleet size

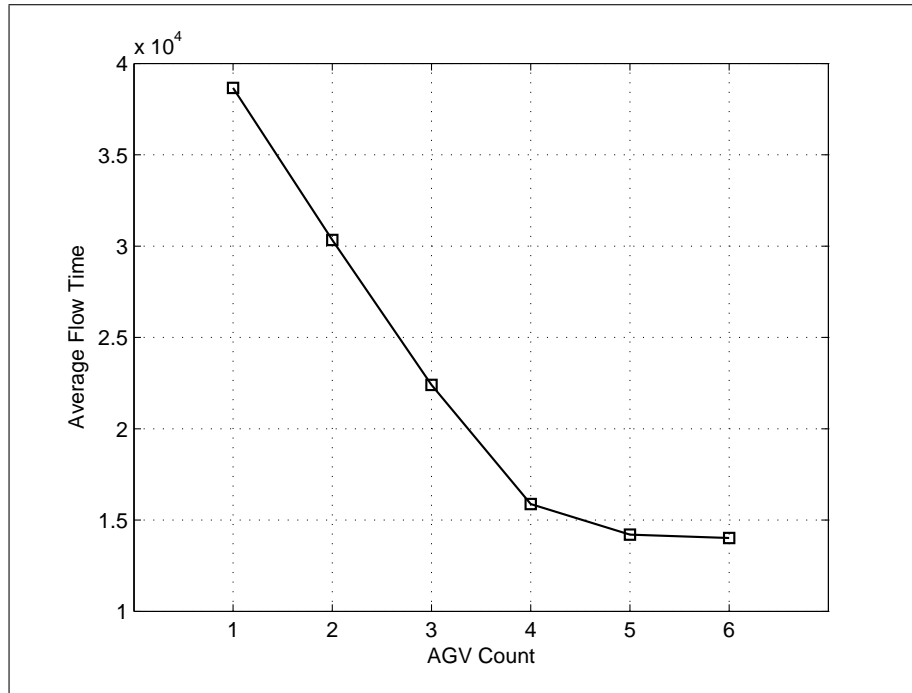


Figure 4.8. Average flow time with changing AGV fleet size

4.4.2. Experiment#2

Instead of using single load AGVs, AGV capacities are increased to two. Figure 4.9 and Figure 4.10 show the results of simulation runs. AGV fleet with three and four double capacity AGVs achieve similar results as AGV fleet with five single capacity AGVs. This configuration can be installed with less cost because it requires smaller number of AGVs than initial configuration in Experiment#1.

4.4.3. Experiment#3

Secondary machine alternatives of operations are utilized to see the effect of flexibility on system performance. AGV fleet is composed of five single capacity AGVs. Table 4.12 and Table 4.13 gives simulation results for the base configuration and the modified configuration after introducing more flexibility. Giving secondary machine alternatives to operations significantly increases the performance of the system.

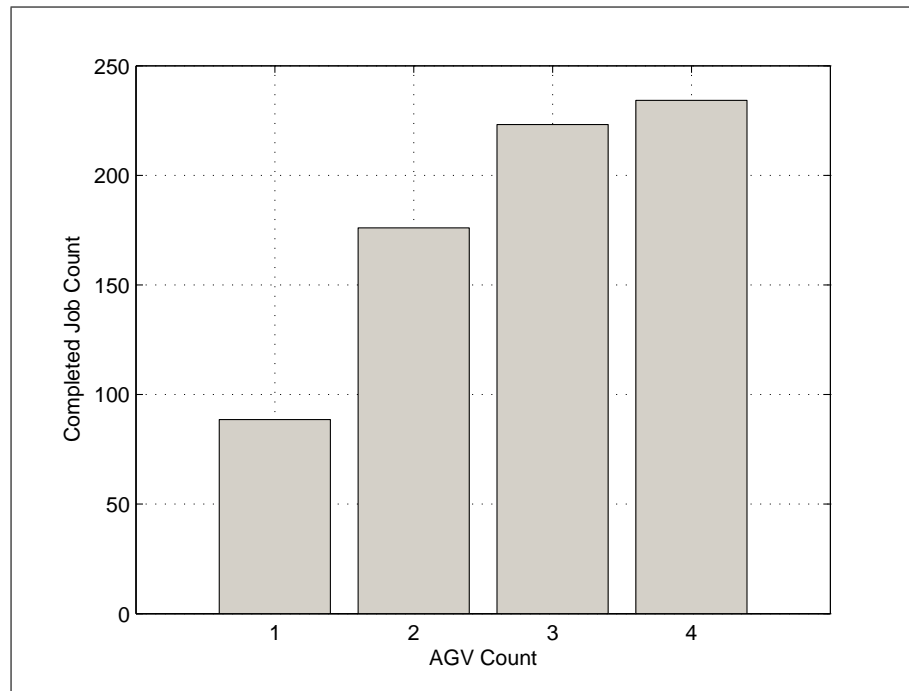


Figure 4.9. Completed job count with changing AGV fleet size after doubling AGV capacities

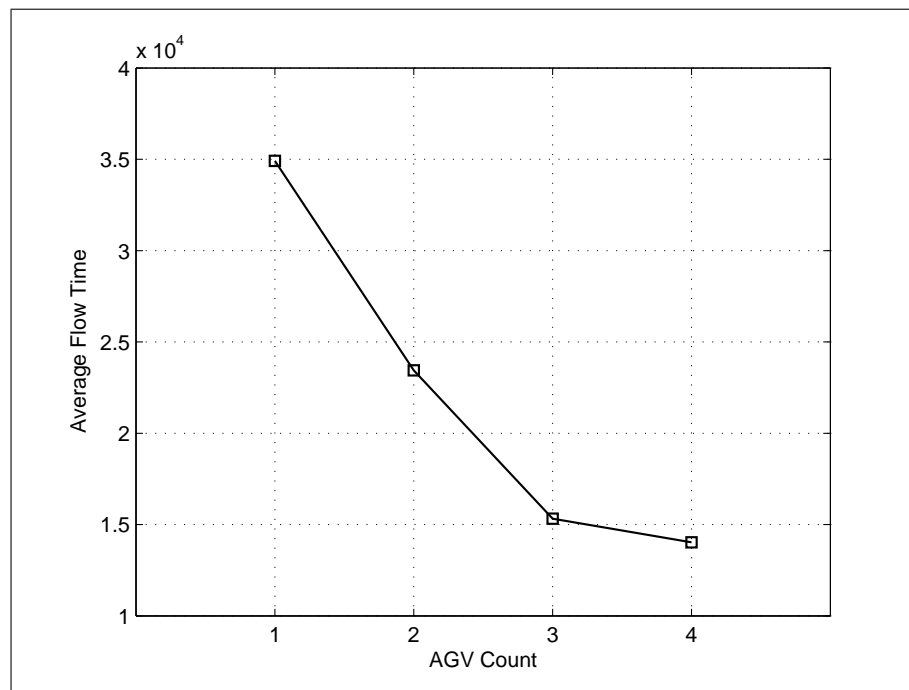


Figure 4.10. Average flow time with changing AGV fleet size after doubling AGV capacities

Table 4.12. Completed job count before and after increasing flexibility

	Base Case		Routing Flexibility Case	
	EEFT	MROQS	EEFT	MROQS
Average	231	231	307.8	294.6
Std. Dev.	7.58	7.58	3.77	5.13

Table 4.13. Average flow time before and after increasing flexibility

	Base Case		Routing Flexibility Case	
	EEFT	MROQS	EEFT	MROQS
Average	14203.86	14203.86	4314.21	5724.86
Std. Dev.	788.87	788.87	485.07	644.76

5. CONCLUSIONS AND FUTURE STUDIES

This thesis introduces a new application generation framework for object-oriented software. Proposed framework is based on graphical input collection through dialog forms and flow diagrams. The users give specifications through this graphical input collection mechanism and the application generator converts this graphical representation into four different programming languages (C++, C#, J# and VB.NET) and compiles the generated source code into an assembly.

The main contribution of proposed application generator is providing a general-purpose application generation mechanism as opposed to domain-specific application generators. Commercial application generators are designed for helping software engineers and specialized in a sub-problem of software development such as database management, user interface design and report generation. These generators requires programming language knowledge at least to some intermediate level.

Instead of concentrating to some specific domain, the proposed generator focuses on common properties of programming languages. Basic programming language constructs, statements and expressions are modeled in the proposed generator as appropriately. Programming language constructs such as namespace, classes, fields, etc. are taken from user through dialog forms. Programming language statements and expressions are collected via flow diagram representation. This general approach allows non-programmers to use this generator for building applications without using any programming language knowledge. This tool can also be used as a learning tool to teach programming languages that are supported in the generator. Novice users reported that the usability of BUILD.NET is satisfactory for them.

The basic limitation of the proposed framework is that it only supports statements and expressions that are defined in CodeDOM standard document. Statements and expressions that are not supported by CodeDOM should be given in a programming language syntax or converted into their correspondent statements or expressions.

Providing extensibility is another important contribution of the proposed generator. Scientific software packages such as optimization and simulation software are basic candidates for this feature. The users want to devise their own algorithms into commercial products. This is achieved via a vendor-generated script language or coding a completely different application by using a programming language and calling API functions of target software. These methods do not guarantee an acceptable performance after implementation and require user effort. APIs that are provided by software vendor may not be adequate to implement intended algorithm. Software vendors may provide a tool like proposed application generator to extend their software and added algorithms are compiled directly into their software. This method is better than using an external program or a script language in terms of performance measures.

An FMS simulator, FMS.NET, is designed and implemented in BUILD.NET to show extensibility feature in a real-life scenario. Operational decision algorithms that are rules that manage the manufacturing environment can be extended by the users. New decision algorithms can be added to the system and these algorithms become available in simulator program after compilation. FMS.NET simplifies algorithm development and does not require implementation time and effort as much as coding algorithm directly with a programming language. Also learning is much easier with this new simulator, students can easily gain information about the system structure and already implemented decision algorithms. FMS.NET will be used as a test-bed in research projects of BUFAIM and undergraduate course projects.

The proposed framework can also be used for updating software packages with a different approach. The compilation logic of proposed framework is integrated into the released software. Required updates and fixes can be applied by sending an XML file to customers and the software is rebuild at the client's computer. This method provides an easy method for updating the software without any configuration errors such as DLL version problems in updating software by utilizing DLL replacements. Confidentiality of source code can be achieved by encrypting XML file that contains updates.

Beside these features, this application generator can be used as infrastructure for a domain-specific application generator. Entity blocks that are required for domain-specific application generators can be derived from basic building blocks of the proposed framework. This extension requires a new user interface and the source code is generated still by the basic building blocks behind the scene.

The proposed framework can be extended to convert source code of a program from the programming language in which it was implemented to another. Current version is capable of only generating source code from object graph that is generated in memory. If source code can be parsed and converted into object graph representation, constructed object graph can easily be converted into source code in a different programming language. This extension requires a carefully implemented parser for the target programming language.

APPENDIX A: UML DIAGRAMS OF BUILD.NET CLASSES

The design process of BUILD.NET is documented by using UML diagrams. This chapter includes UML class diagrams of the classes that reside in BUILD.NET.

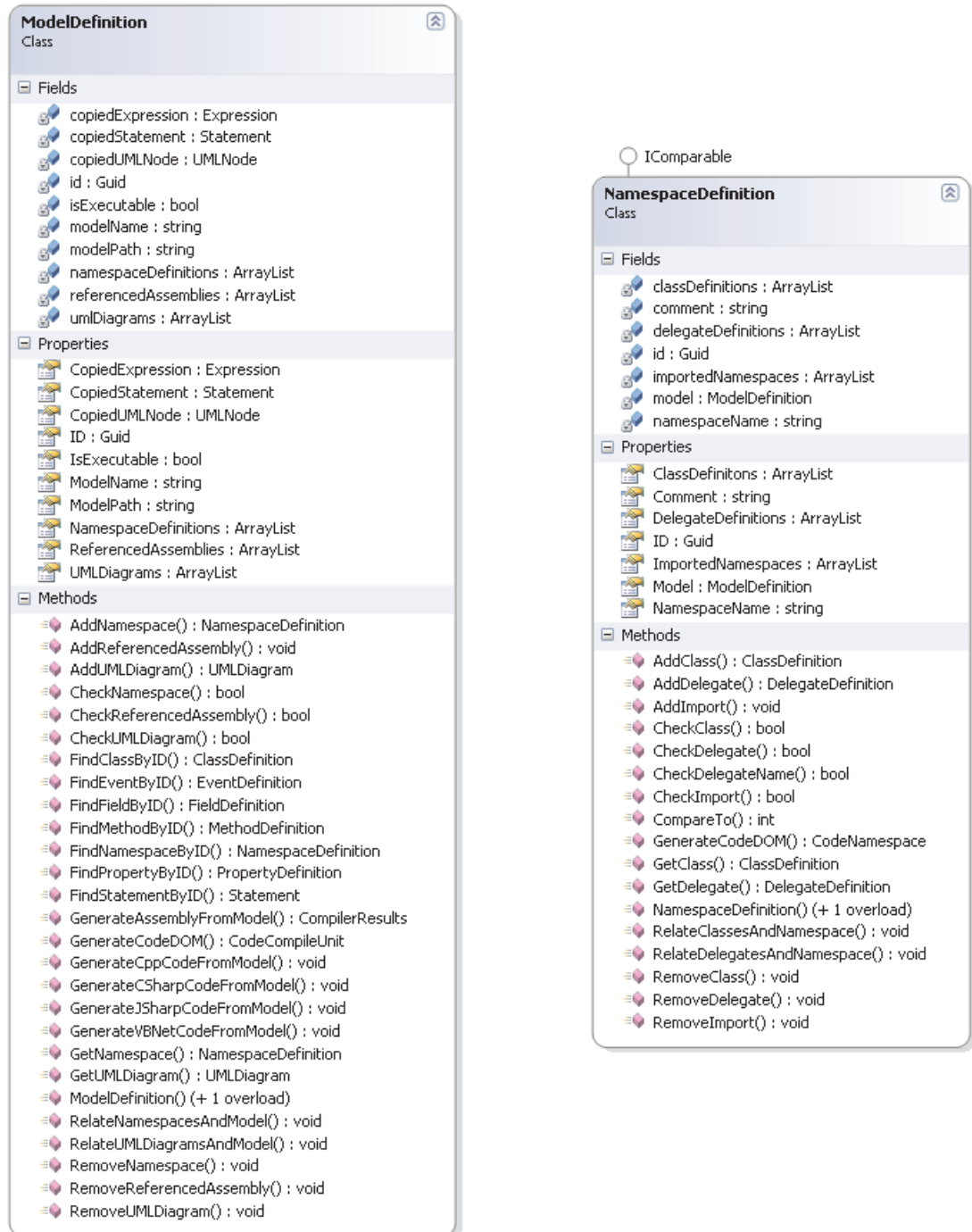


Figure A.1. BUILD.NET class diagrams - I

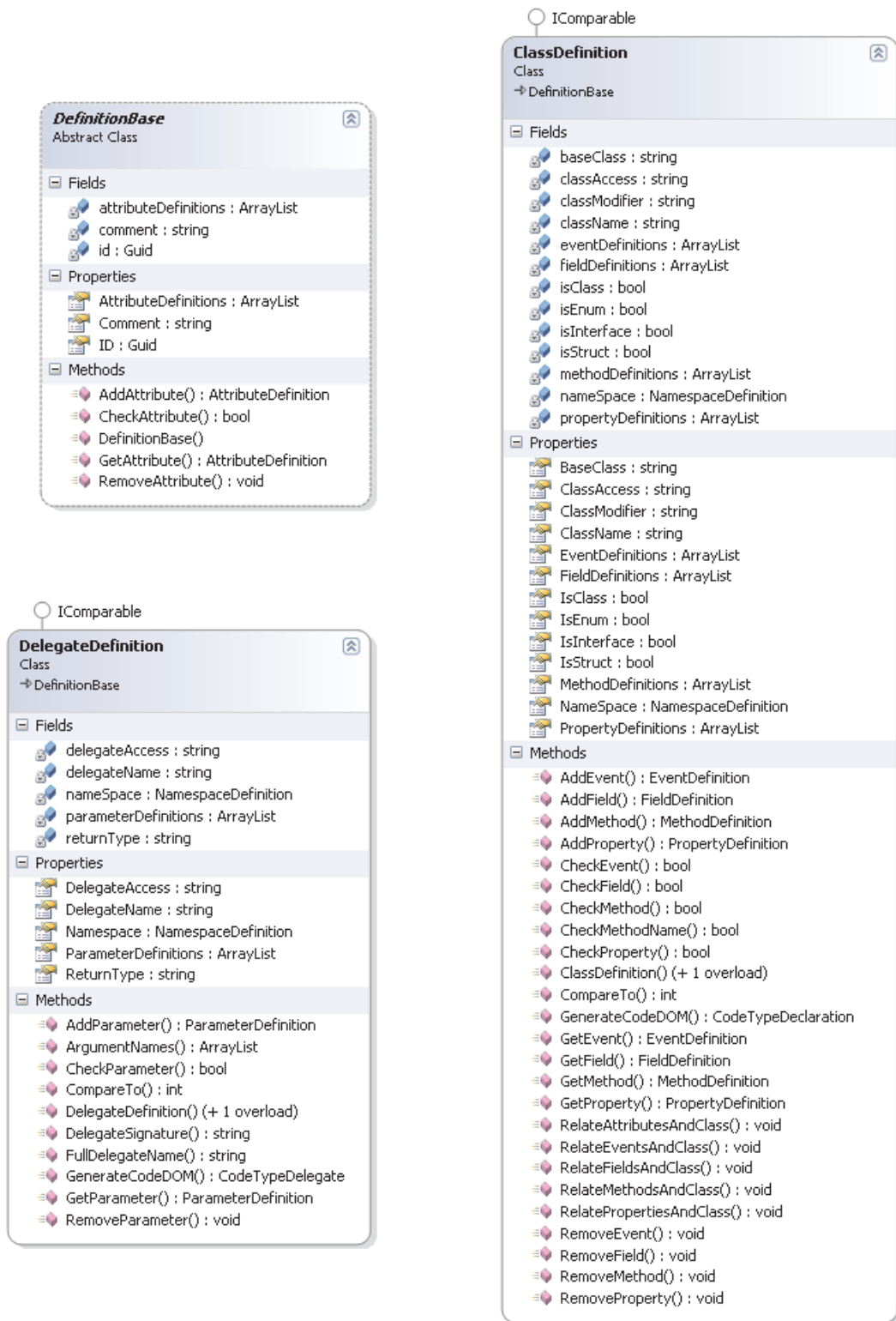


Figure A.2. BUILD.NET class diagrams - II

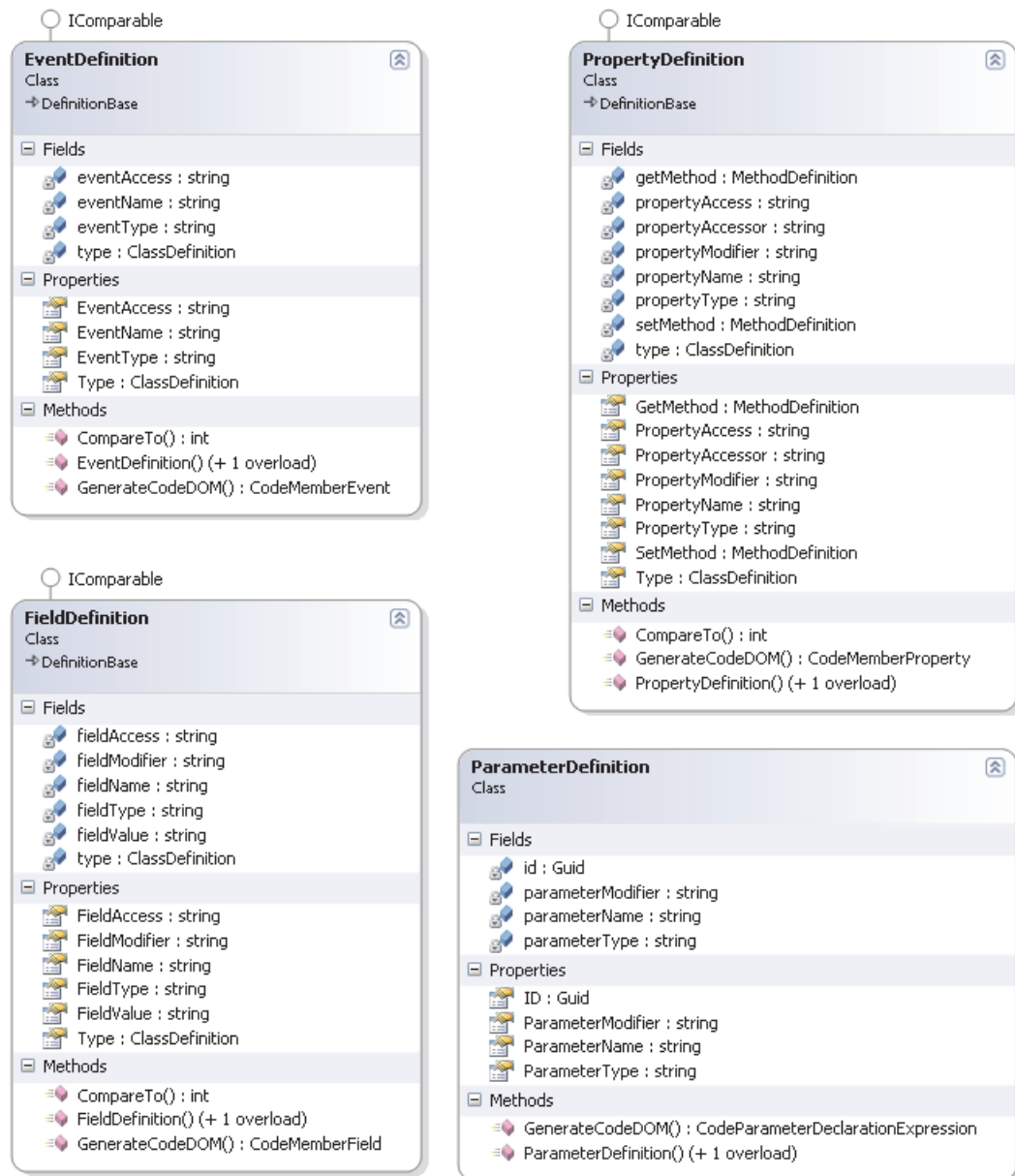


Figure A.3. BUILD.NET class diagrams - III

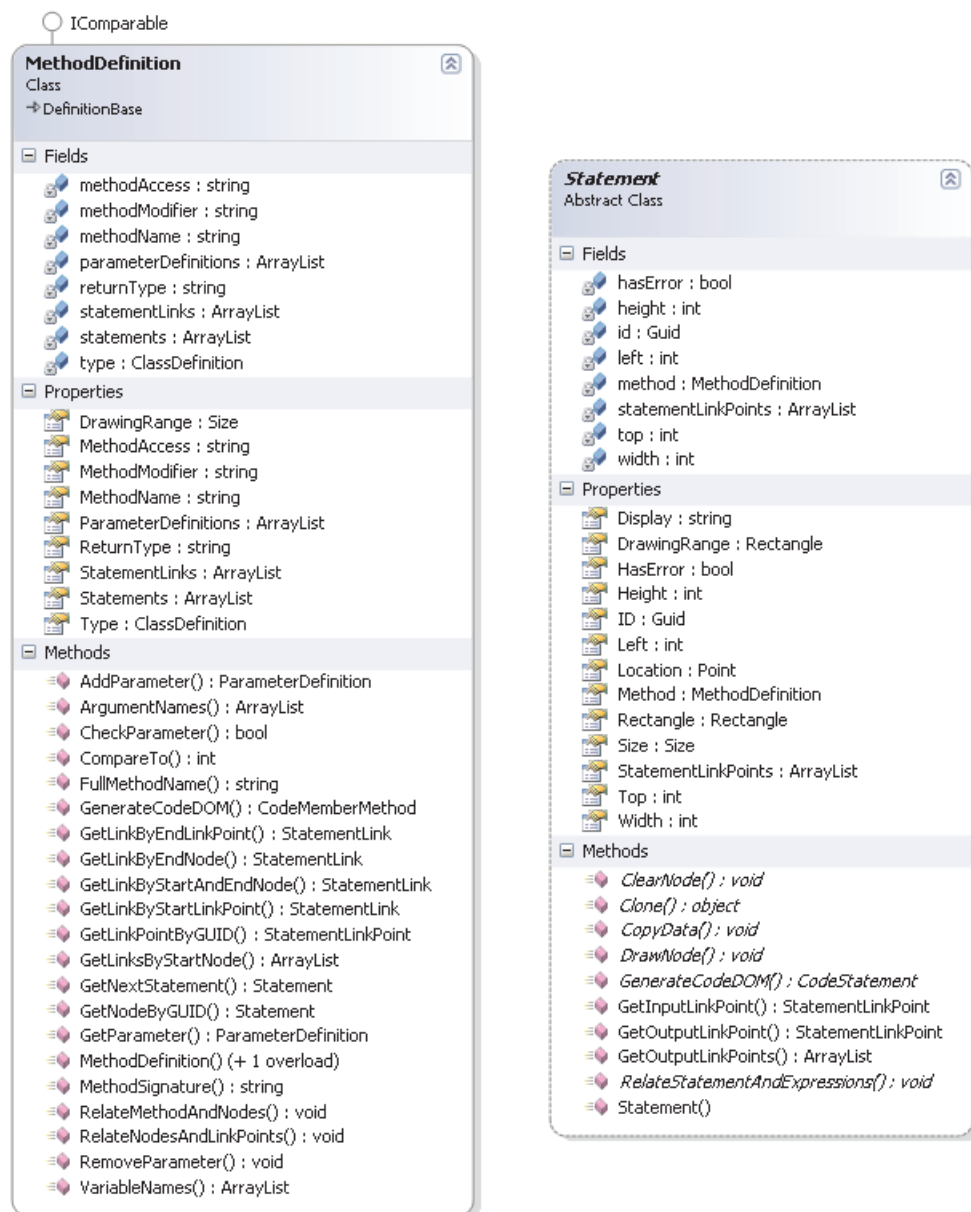


Figure A.4. BUILD.NET class diagrams - IV

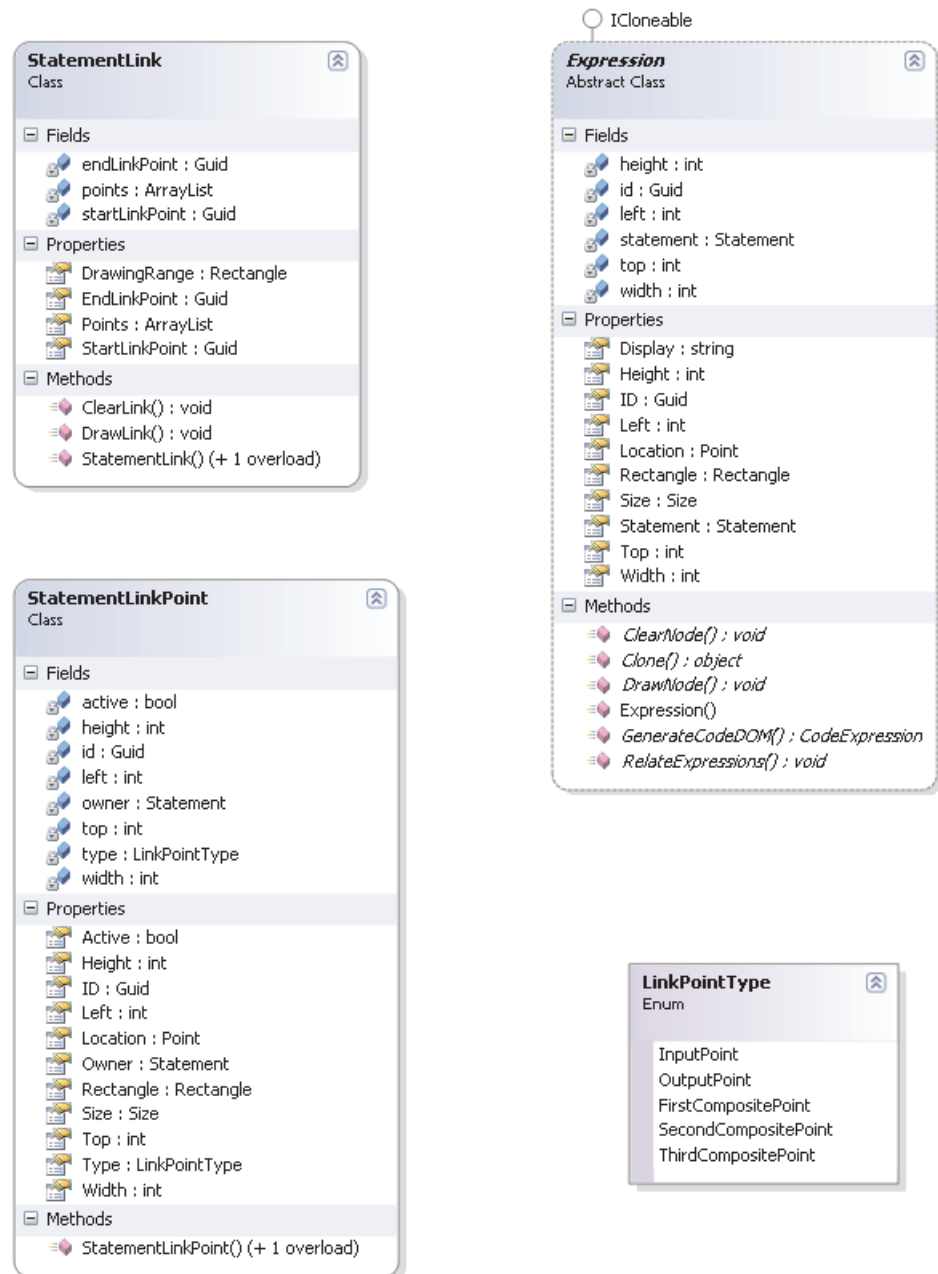


Figure A.5. BUILD.NET class diagrams - V

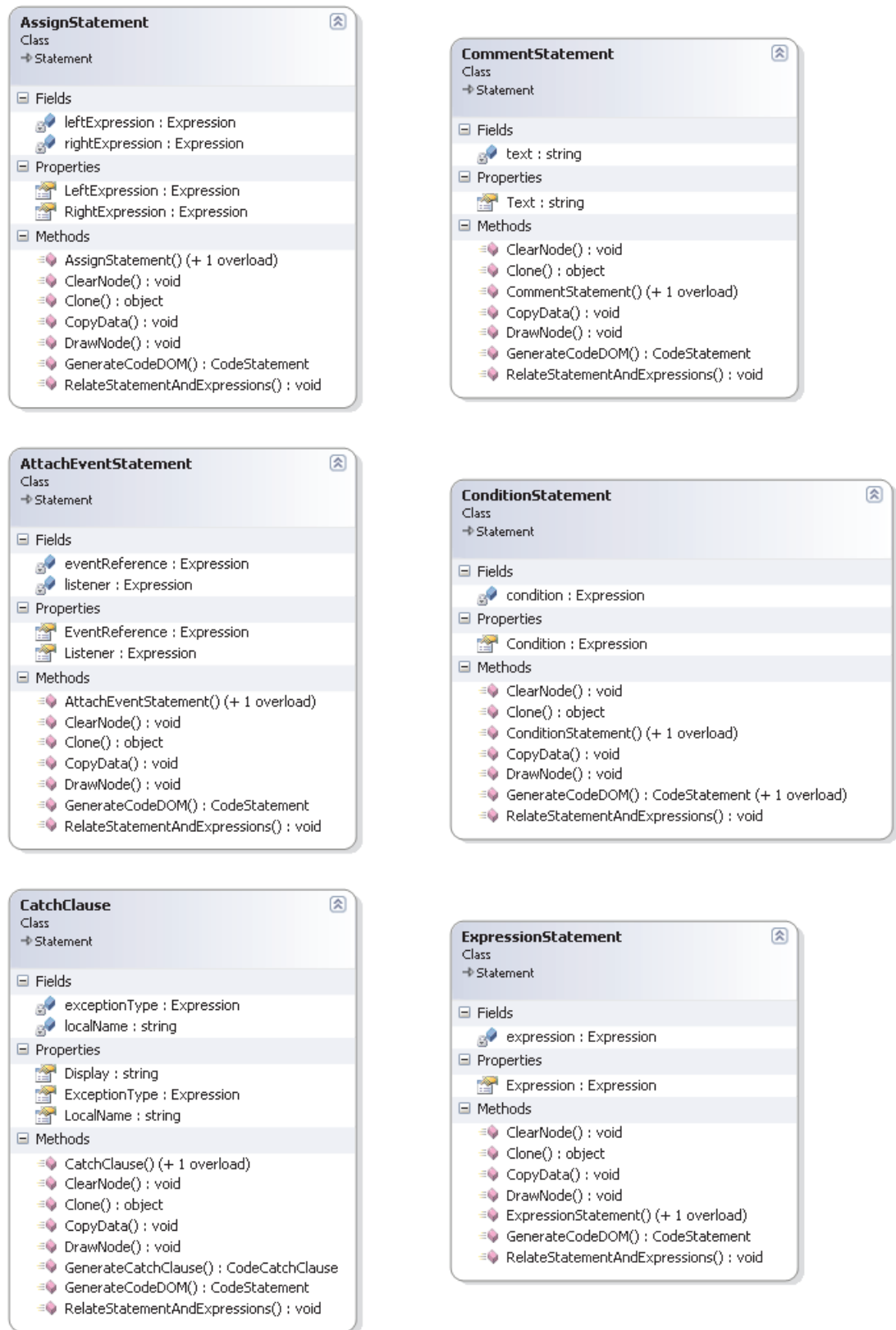


Figure A.6. BUILD.NET class diagrams - VI



Figure A.7. BUILD.NET class diagrams - VII

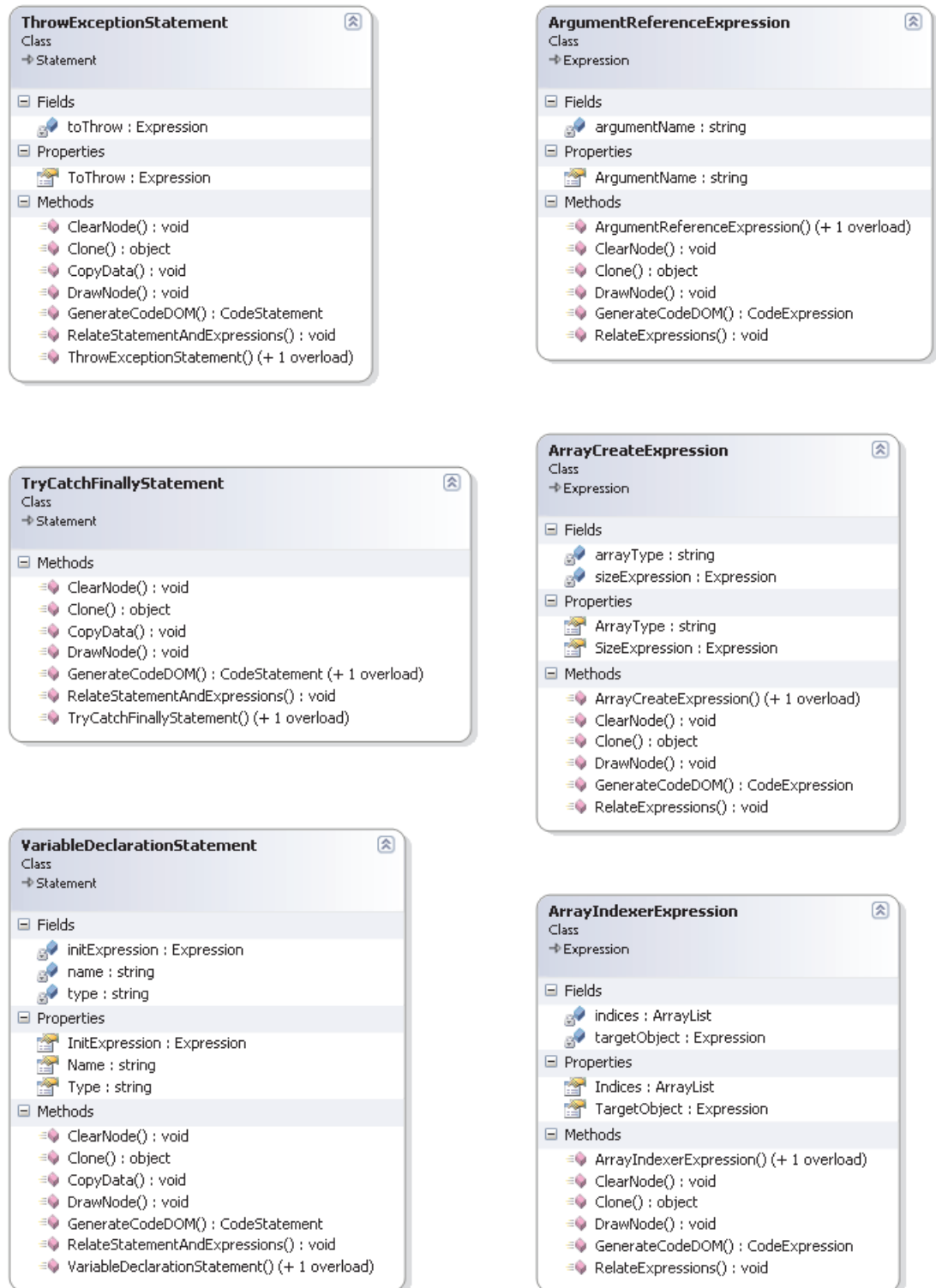


Figure A.8. BUILD.NET class diagrams - VIII

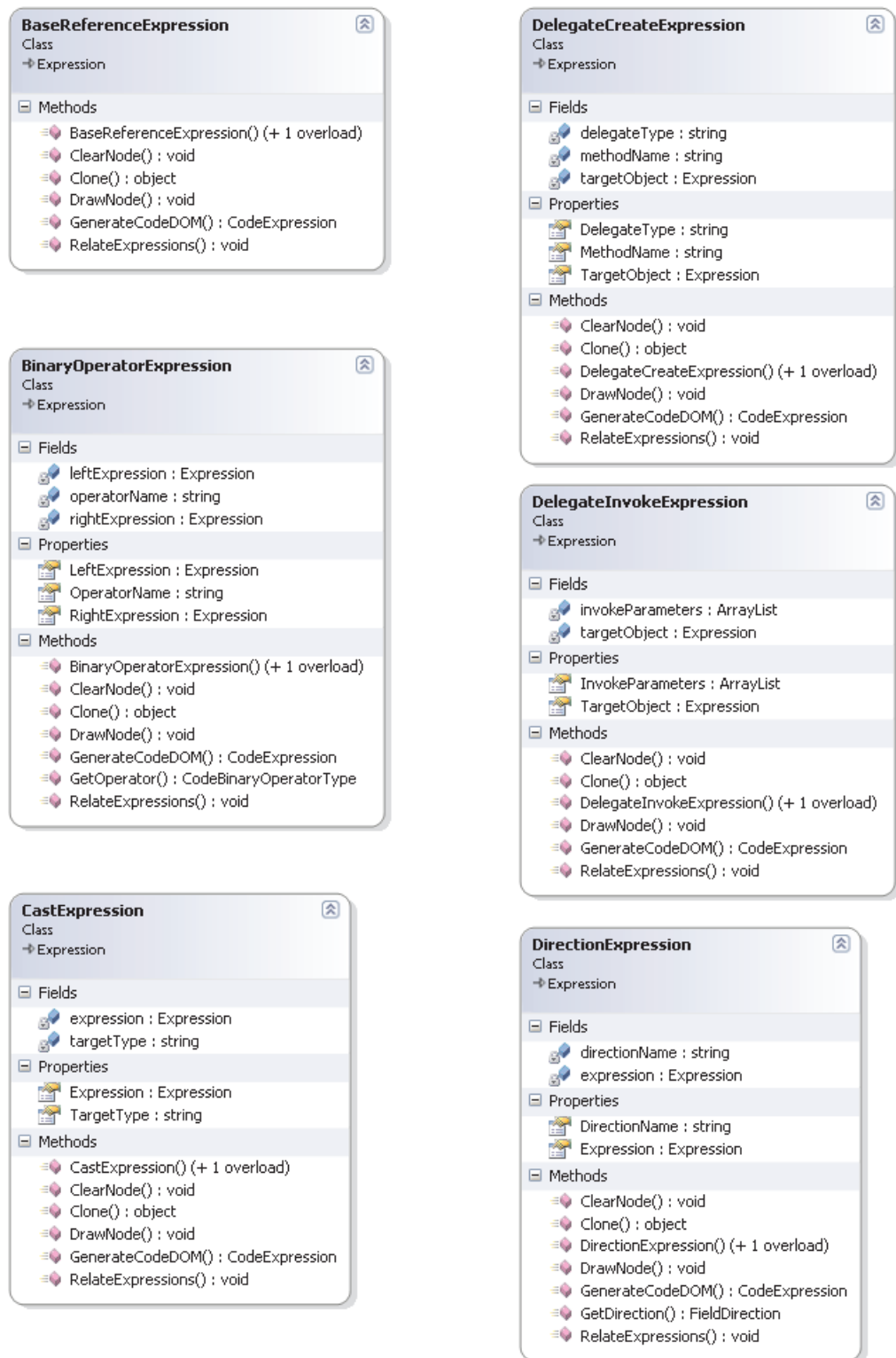


Figure A.9. BUILD.NET class diagrams - IX

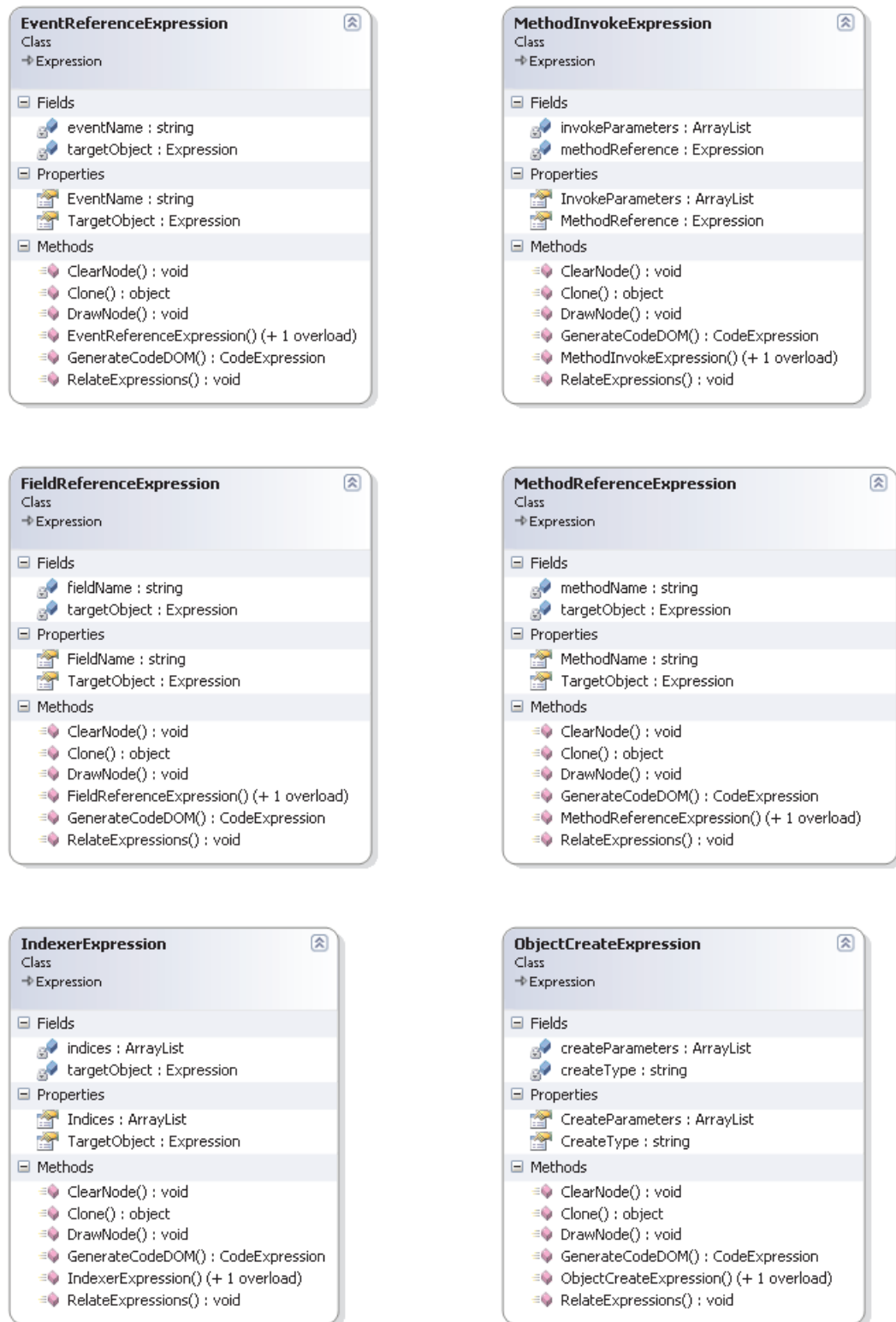


Figure A.10. BUILD.NET class diagrams - X

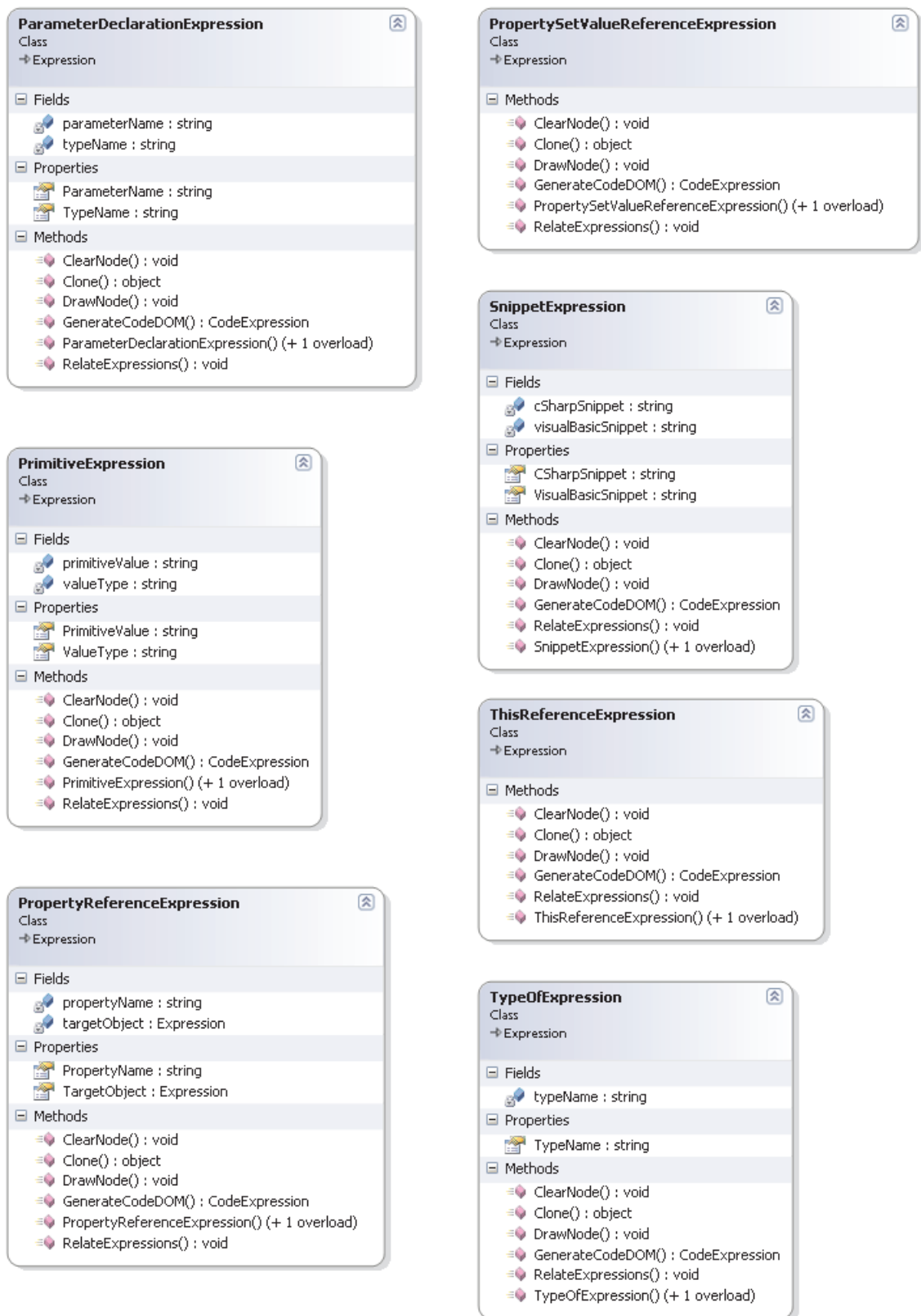


Figure A.11. BUILD.NET class diagrams - XI

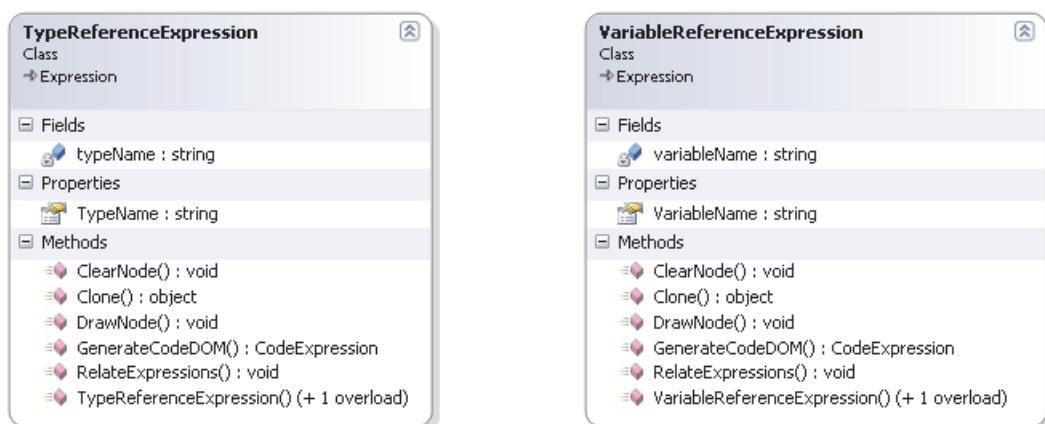
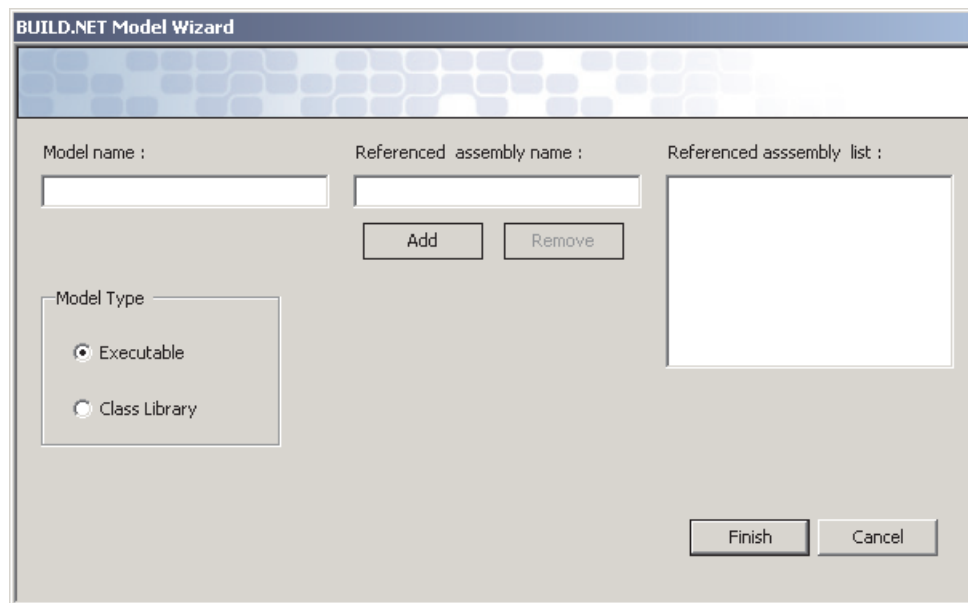


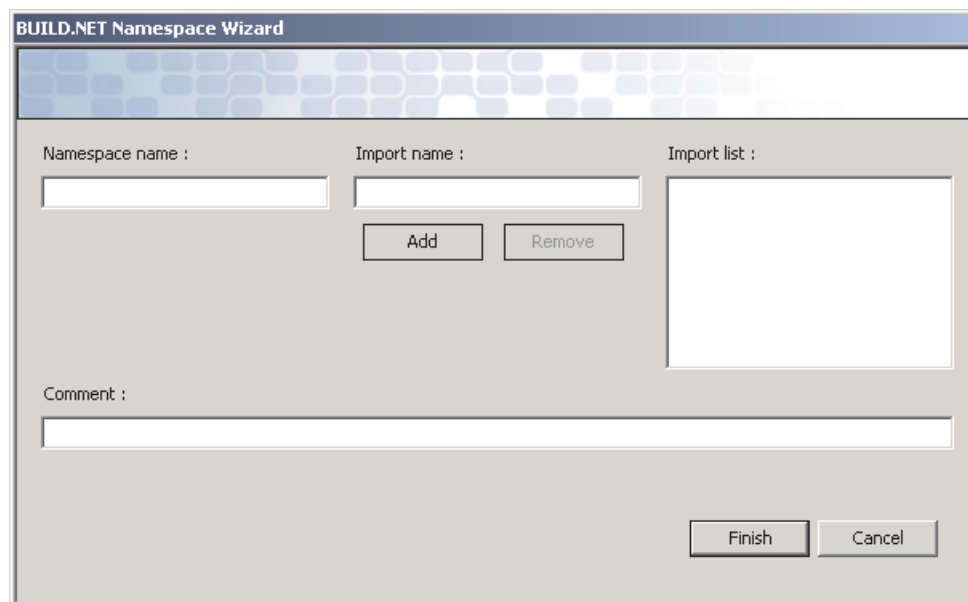
Figure A.12. BUILD.NET class diagrams - XII

APPENDIX B: SCREENSHOTS OF BUILD.NET



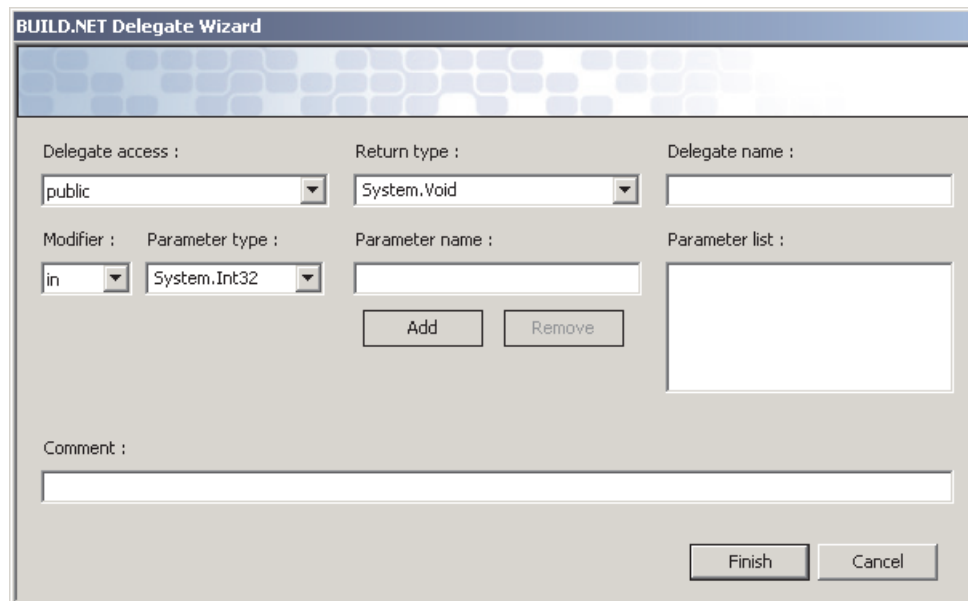
The screenshot shows the 'BUILD.NET Model Wizard' dialog box. It has a title bar with the text 'BUILD.NET Model Wizard'. Below the title bar is a decorative header with a blue and white pattern. The main area contains three input fields: 'Model name :', 'Referenced assembly name :', and 'Referenced assembly list :'. The 'Model name' and 'Referenced assembly name' fields are text boxes. The 'Referenced assembly list' is a list box. Below the 'Referenced assembly name' field are 'Add' and 'Remove' buttons. To the left of the 'Referenced assembly list' is a 'Model Type' section with two radio buttons: 'Executable' (selected) and 'Class Library'. At the bottom right are 'Finish' and 'Cancel' buttons.

Figure B.1. Model definition wizard



The screenshot shows the 'BUILD.NET Namespace Wizard' dialog box. It has a title bar with the text 'BUILD.NET Namespace Wizard'. Below the title bar is a decorative header with a blue and white pattern. The main area contains three input fields: 'Namespace name :', 'Import name :', and 'Import list :'. The 'Namespace name' and 'Import name' fields are text boxes. The 'Import list' is a list box. Below the 'Import name' field are 'Add' and 'Remove' buttons. At the bottom left is a 'Comment :' text box. At the bottom right are 'Finish' and 'Cancel' buttons.

Figure B.2. Namespace definition wizard

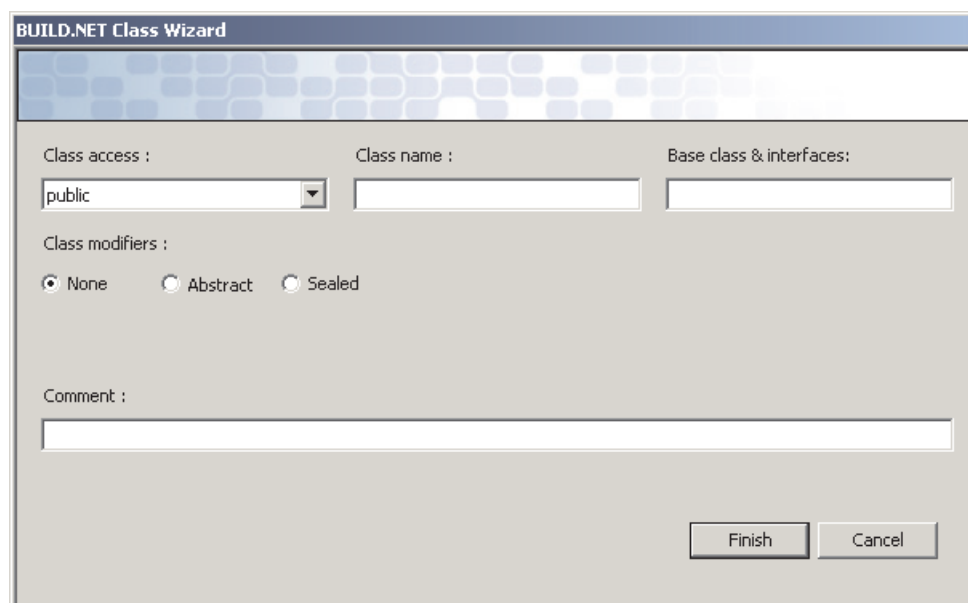


The BUILD.NET Delegate Wizard dialog box is used for defining a delegate. It features a title bar with the text "BUILD.NET Delegate Wizard". Below the title bar is a decorative header with a blue and white grid pattern. The main area contains several input fields and buttons. The "Delegate access" field is a dropdown menu set to "public". The "Return type" field is a dropdown menu set to "System.Void". The "Delegate name" field is a text box. The "Modifier" field is a dropdown menu set to "in". The "Parameter type" field is a dropdown menu set to "System.Int32". The "Parameter name" field is a text box. The "Parameter list" field is a large text box. There are "Add" and "Remove" buttons between the "Parameter name" and "Parameter list" fields. A "Comment" field is a text box at the bottom. At the bottom right are "Finish" and "Cancel" buttons.

BUILD.NET Delegate Wizard

Delegate access : public
Return type : System.Void
Delegate name :
Modifier : in
Parameter type : System.Int32
Parameter name :
Parameter list :
Add Remove
Comment :
Finish Cancel

Figure B.3. Delegate definition wizard

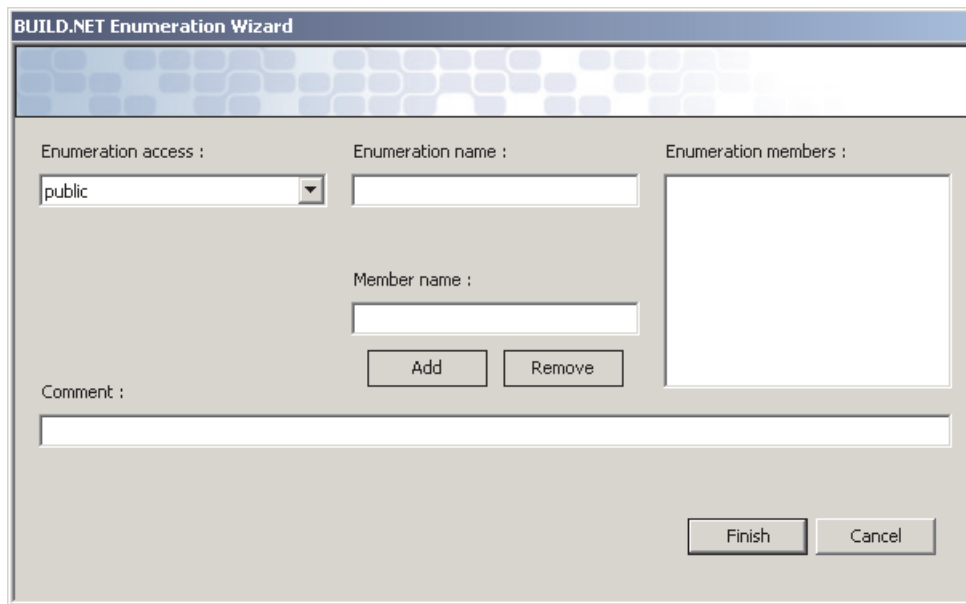


The BUILD.NET Class Wizard dialog box is used for defining a class. It features a title bar with the text "BUILD.NET Class Wizard". Below the title bar is a decorative header with a blue and white grid pattern. The main area contains several input fields and buttons. The "Class access" field is a dropdown menu set to "public". The "Class name" field is a text box. The "Base class & interfaces" field is a text box. The "Class modifiers" section has three radio buttons: "None" (selected), "Abstract", and "Sealed". A "Comment" field is a text box at the bottom. At the bottom right are "Finish" and "Cancel" buttons.

BUILD.NET Class Wizard

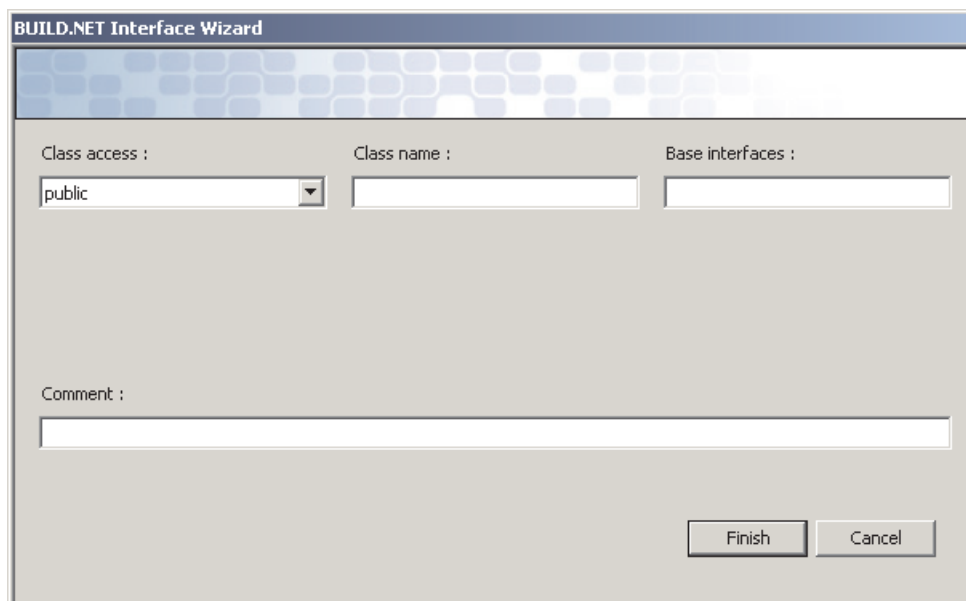
Class access : public
Class name :
Base class & interfaces :
Class modifiers :
☒ None ☐ Abstract ☐ Sealed
Comment :
Finish Cancel

Figure B.4. Class definition wizard



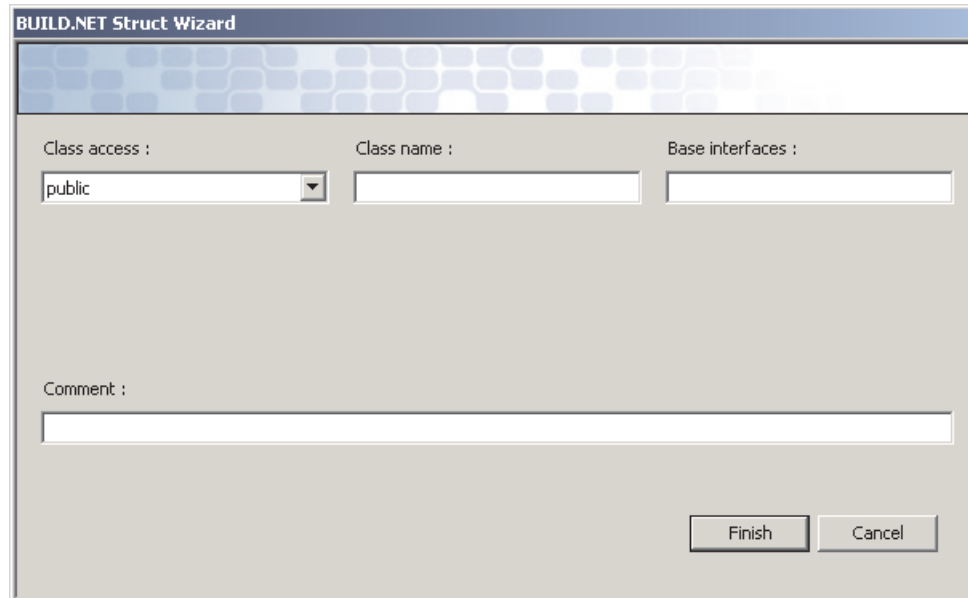
The BUILD.NET Enumeration Wizard dialog box features a title bar with the text "BUILD.NET Enumeration Wizard". Below the title bar is a decorative header with a blue and white grid pattern. The main area contains three input fields: "Enumeration access :" with a dropdown menu showing "public", "Enumeration name :" with a text box, and "Enumeration members :" with a large empty list box. Below these is a "Member name :" text box and two buttons, "Add" and "Remove". A "Comment :" text box is located at the bottom left. At the bottom right are "Finish" and "Cancel" buttons.

Figure B.5. Enumeration definition wizard



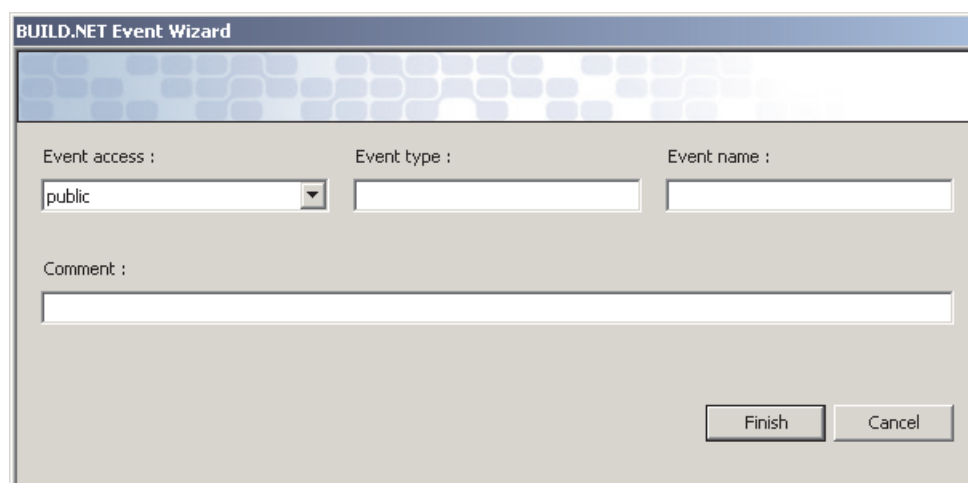
The BUILD.NET Interface Wizard dialog box features a title bar with the text "BUILD.NET Interface Wizard". Below the title bar is a decorative header with a blue and white grid pattern. The main area contains three input fields: "Class access :" with a dropdown menu showing "public", "Class name :" with a text box, and "Base interfaces :" with a text box. A "Comment :" text box is located at the bottom left. At the bottom right are "Finish" and "Cancel" buttons.

Figure B.6. Interface definition wizard



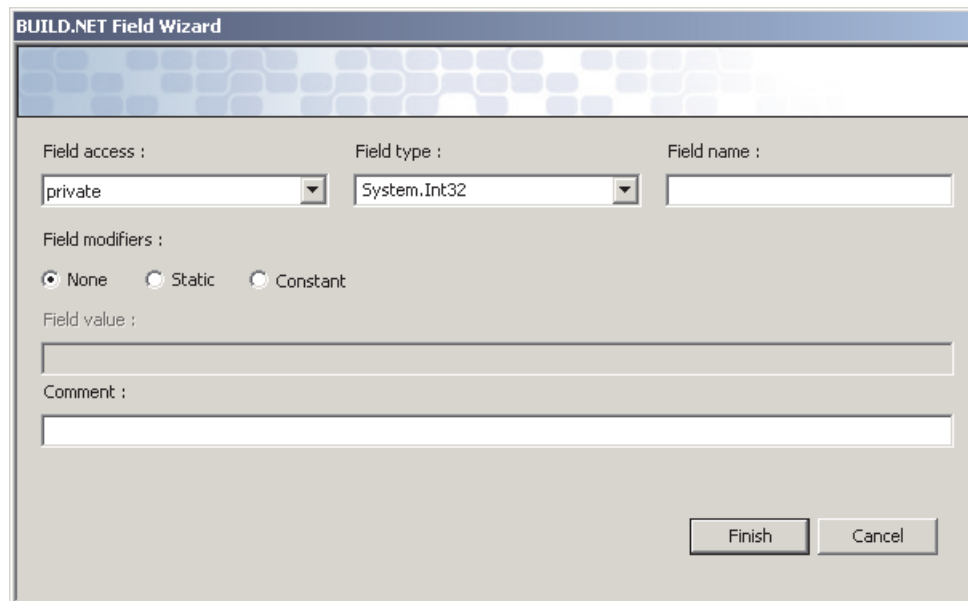
The BUILD.NET Struct Wizard dialog box features a title bar with the text "BUILD.NET Struct Wizard". Below the title bar is a decorative header with a blue and white grid pattern. The main area contains three input fields: "Class access :" with a dropdown menu showing "public", "Class name :" with an empty text box, and "Base interfaces :" with an empty text box. Below these is a "Comment :" label followed by a large empty text area. At the bottom right are "Finish" and "Cancel" buttons.

Figure B.7. Struct definition wizard



The BUILD.NET Event Wizard dialog box features a title bar with the text "BUILD.NET Event Wizard". Below the title bar is a decorative header with a blue and white grid pattern. The main area contains three input fields: "Event access :" with a dropdown menu showing "public", "Event type :" with an empty text box, and "Event name :" with an empty text box. Below these is a "Comment :" label followed by a large empty text area. At the bottom right are "Finish" and "Cancel" buttons.

Figure B.8. Event definition wizard



The BUILD.NET Field Wizard dialog box is used for defining a new field. It features a title bar with the text "BUILD.NET Field Wizard". Below the title bar is a decorative header with a blue and white grid pattern. The main area contains several input fields and a group of radio buttons. The "Field access" dropdown is set to "private". The "Field type" dropdown is set to "System.Int32". The "Field name" field is empty. The "Field modifiers" section has three radio buttons: "None" (selected), "Static", and "Constant". The "Field value" field is empty. The "Comment" field is empty. At the bottom right are "Finish" and "Cancel" buttons.

BUILD.NET Field Wizard

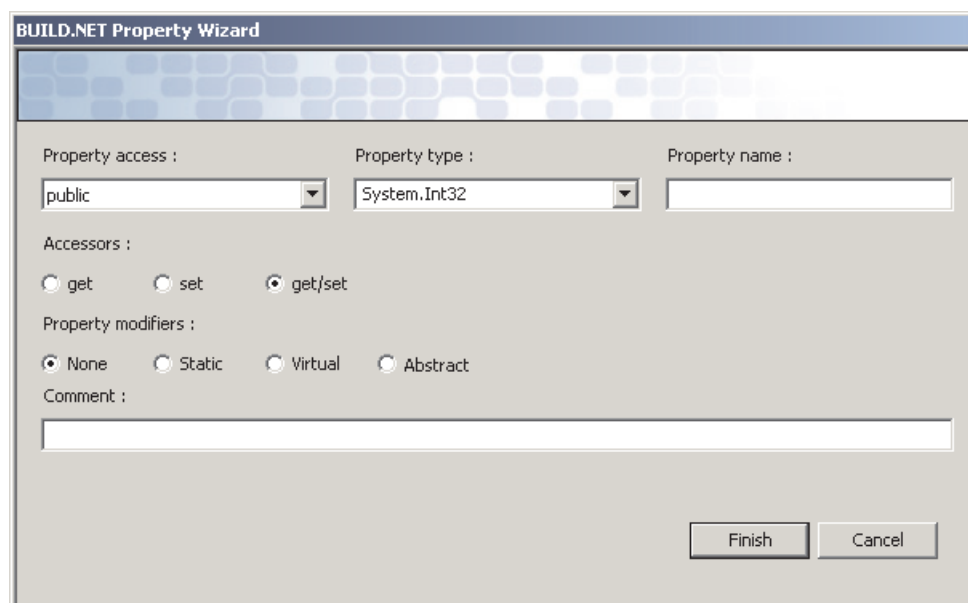
Field access : Field type : Field name :

Field modifiers :
☒ None ☐ Static ☐ Constant

Field value :

Comment :

Figure B.9. Field definition wizard



The BUILD.NET Property Wizard dialog box is used for defining a new property. It features a title bar with the text "BUILD.NET Property Wizard". Below the title bar is a decorative header with a blue and white grid pattern. The main area contains several input fields and a group of radio buttons. The "Property access" dropdown is set to "public". The "Property type" dropdown is set to "System.Int32". The "Property name" field is empty. The "Accessors" section has three radio buttons: "get" (selected), "set", and "get/set". The "Property modifiers" section has four radio buttons: "None" (selected), "Static", "Virtual", and "Abstract". The "Comment" field is empty. At the bottom right are "Finish" and "Cancel" buttons.

BUILD.NET Property Wizard

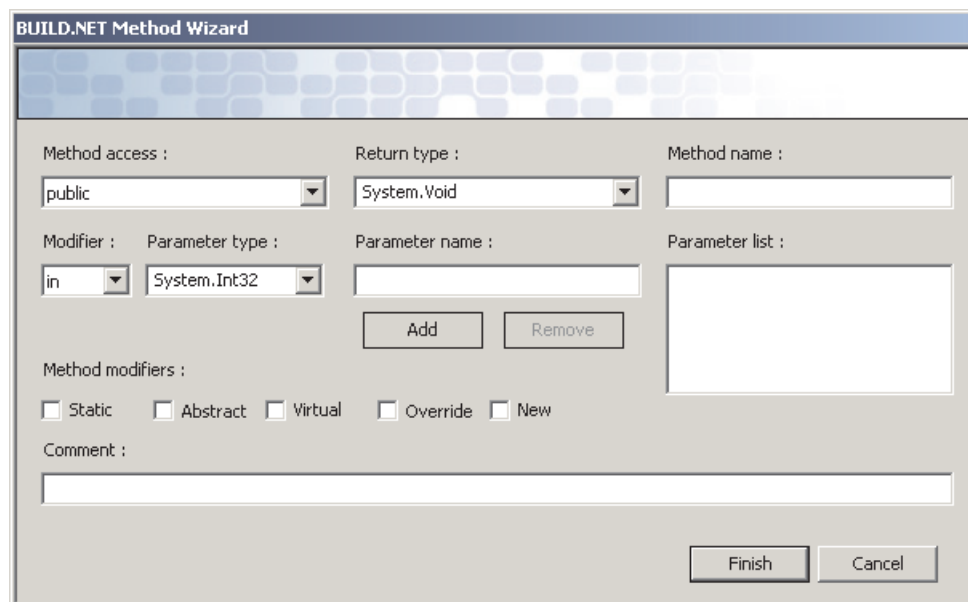
Property access : Property type : Property name :

Accessors :
☒ get ☐ set ☐ get/set

Property modifiers :
☒ None ☐ Static ☐ Virtual ☐ Abstract

Comment :

Figure B.10. Property definition wizard



The image shows a 'BUILD.NET Method Wizard' dialog box. It has a title bar with the text 'BUILD.NET Method Wizard'. Below the title bar is a decorative header with a blue and white pattern. The main area contains several fields and controls:

- Method access :** A dropdown menu with 'public' selected.
- Return type :** A dropdown menu with 'System.Void' selected.
- Method name :** An empty text box.
- Modifier :** A dropdown menu with 'in' selected.
- Parameter type :** A dropdown menu with 'System.Int32' selected.
- Parameter name :** An empty text box.
- Parameter list :** A large empty text box.
- Method modifiers :** A row of checkboxes: ☐ Static, ☐ Abstract, ☐ Virtual, ☐ Override, and ☐ New.
- Comment :** A large empty text box.
- Buttons:** 'Add' and 'Remove' buttons are located below the 'Parameter name' field. 'Finish' and 'Cancel' buttons are at the bottom right.

Figure B.11. Method definition wizard

APPENDIX C: UML DIAGRAMS OF FMS.NET CLASSES

The design process of FMS.NET is documented by using UML diagrams. This chapter includes UML class diagrams of the classes that reside in FMS.NET.

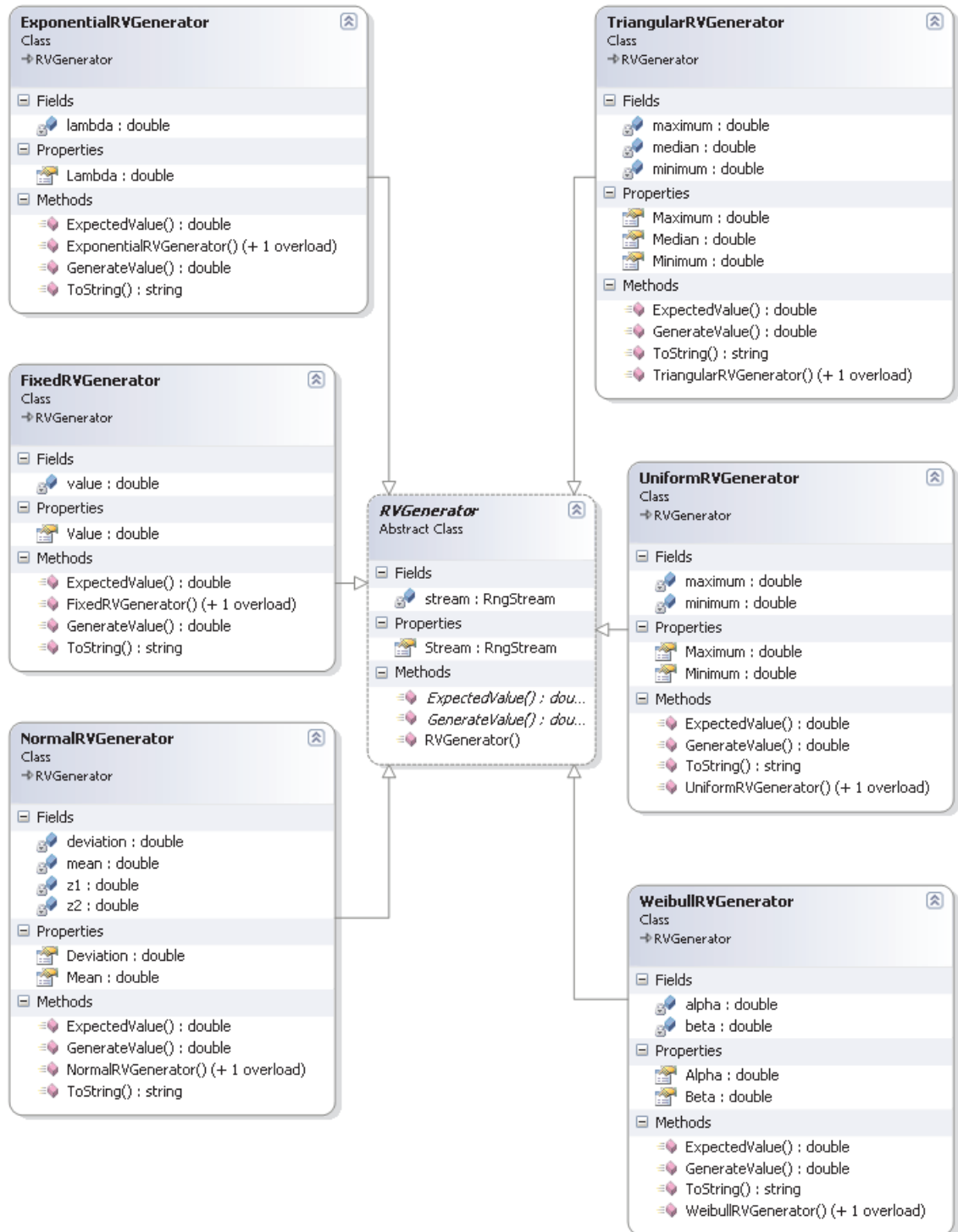


Figure C.1. FMS.NET class diagrams - I

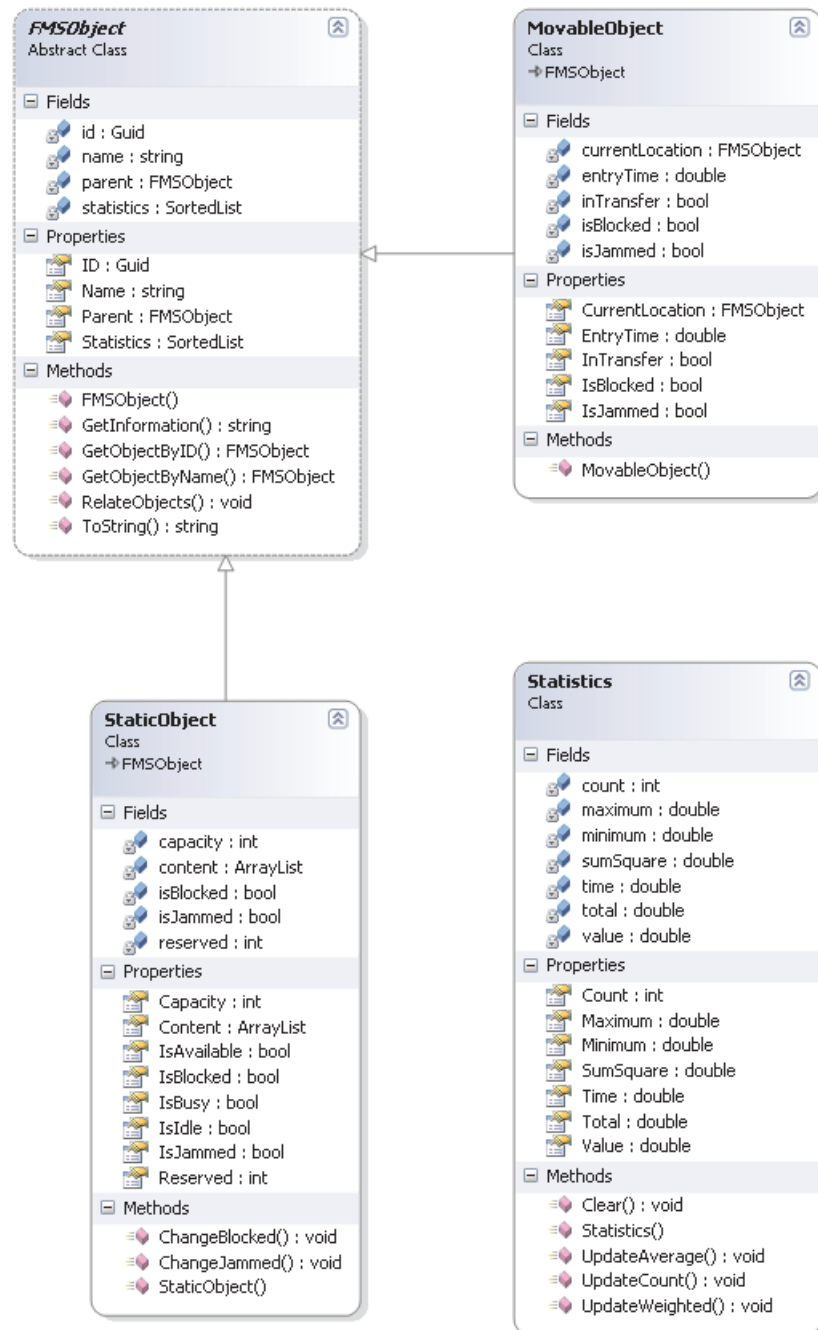


Figure C.2. FMS.NET class diagrams - II

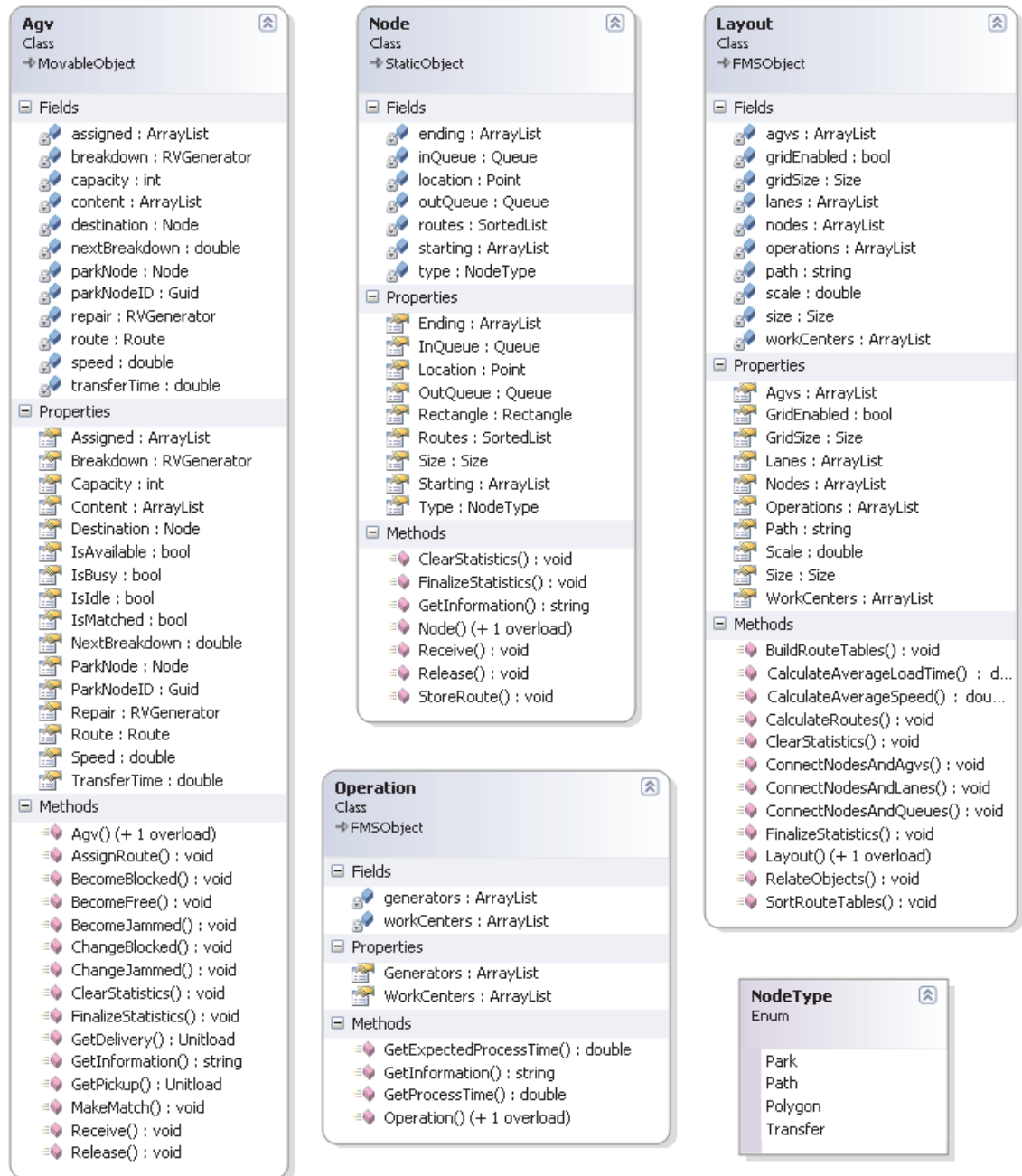


Figure C.3. FMS.NET class diagrams - III

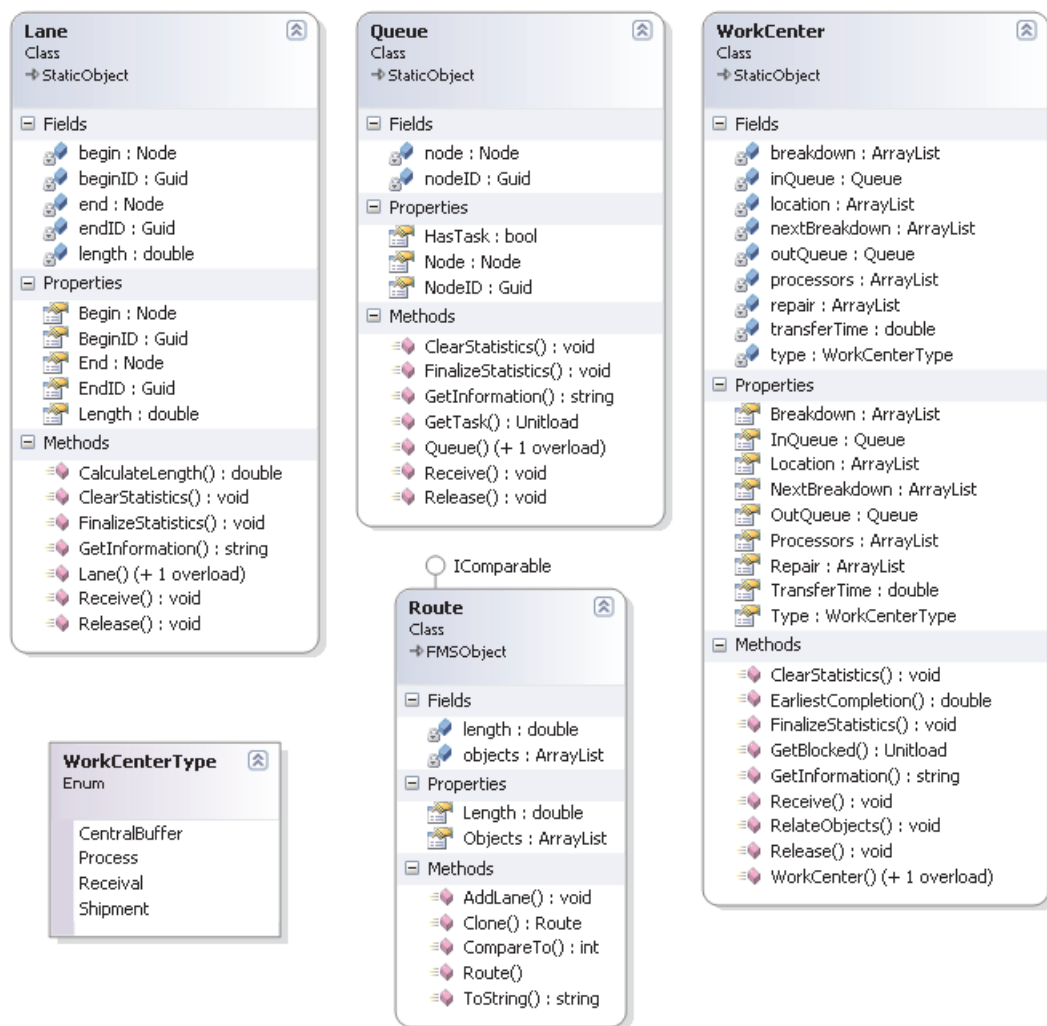


Figure C.4. FMS.NET class diagrams - IV

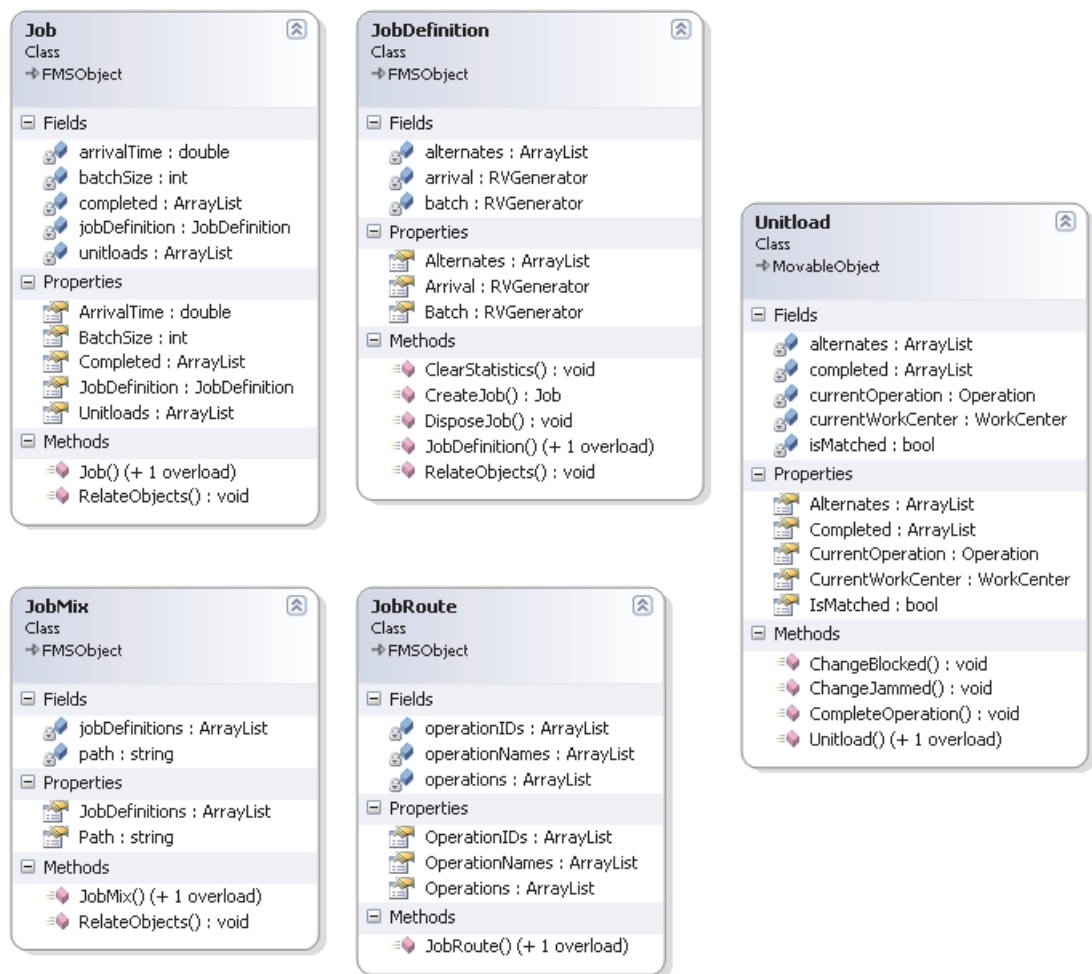


Figure C.5. FMS.NET class diagrams - V

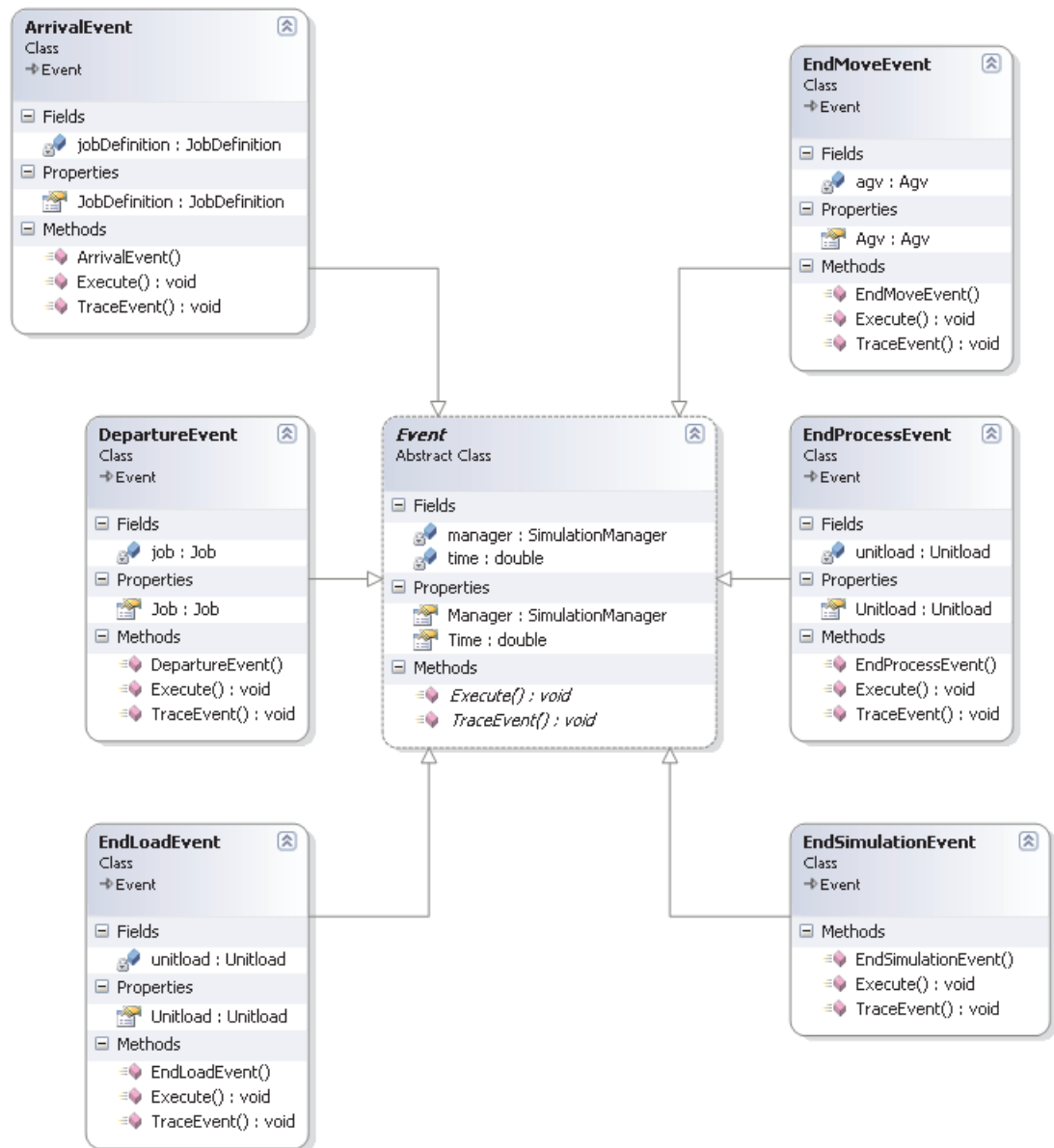


Figure C.6. FMS.NET class diagrams - VI

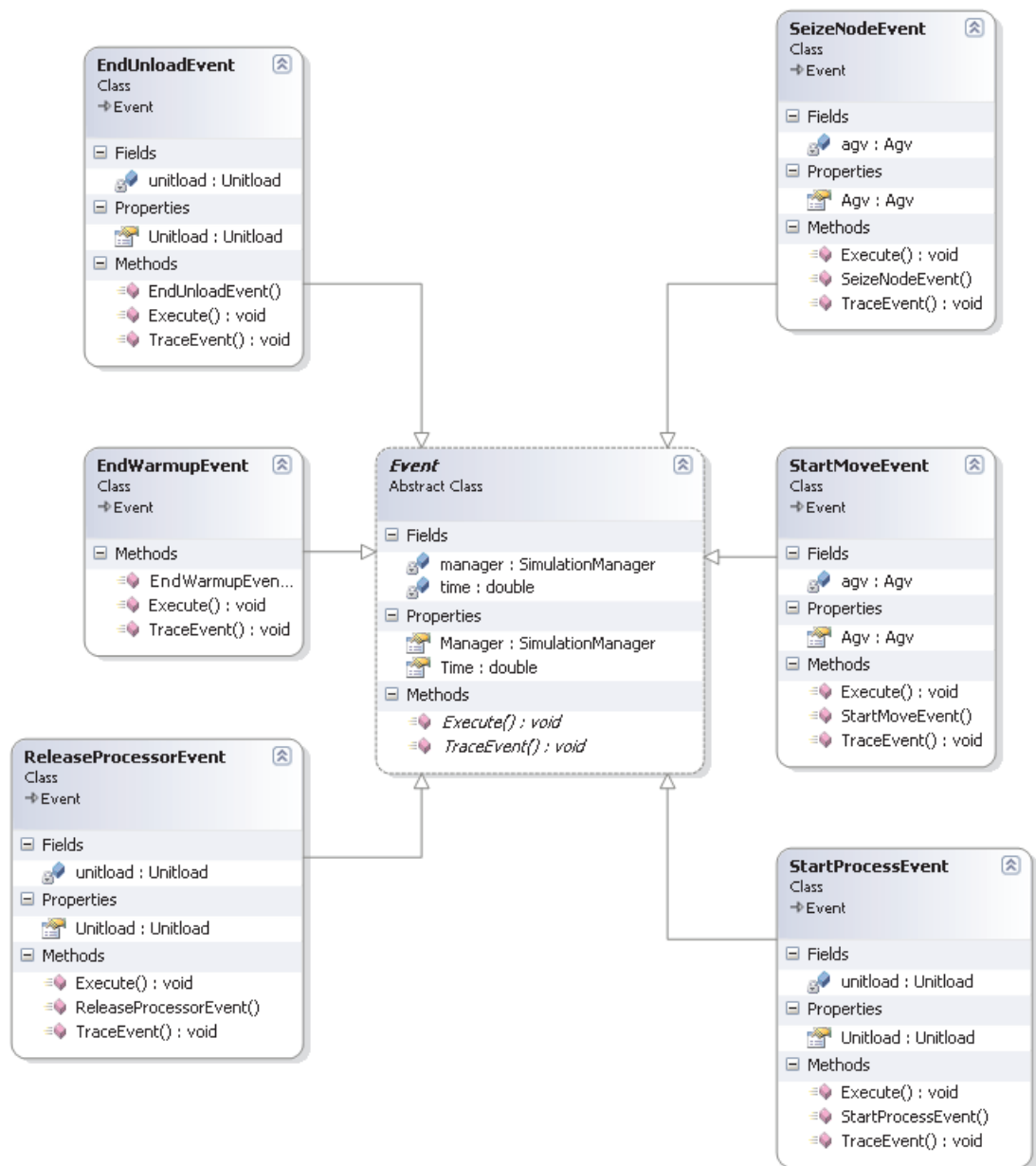


Figure C.7. FMS.NET class diagrams - VII

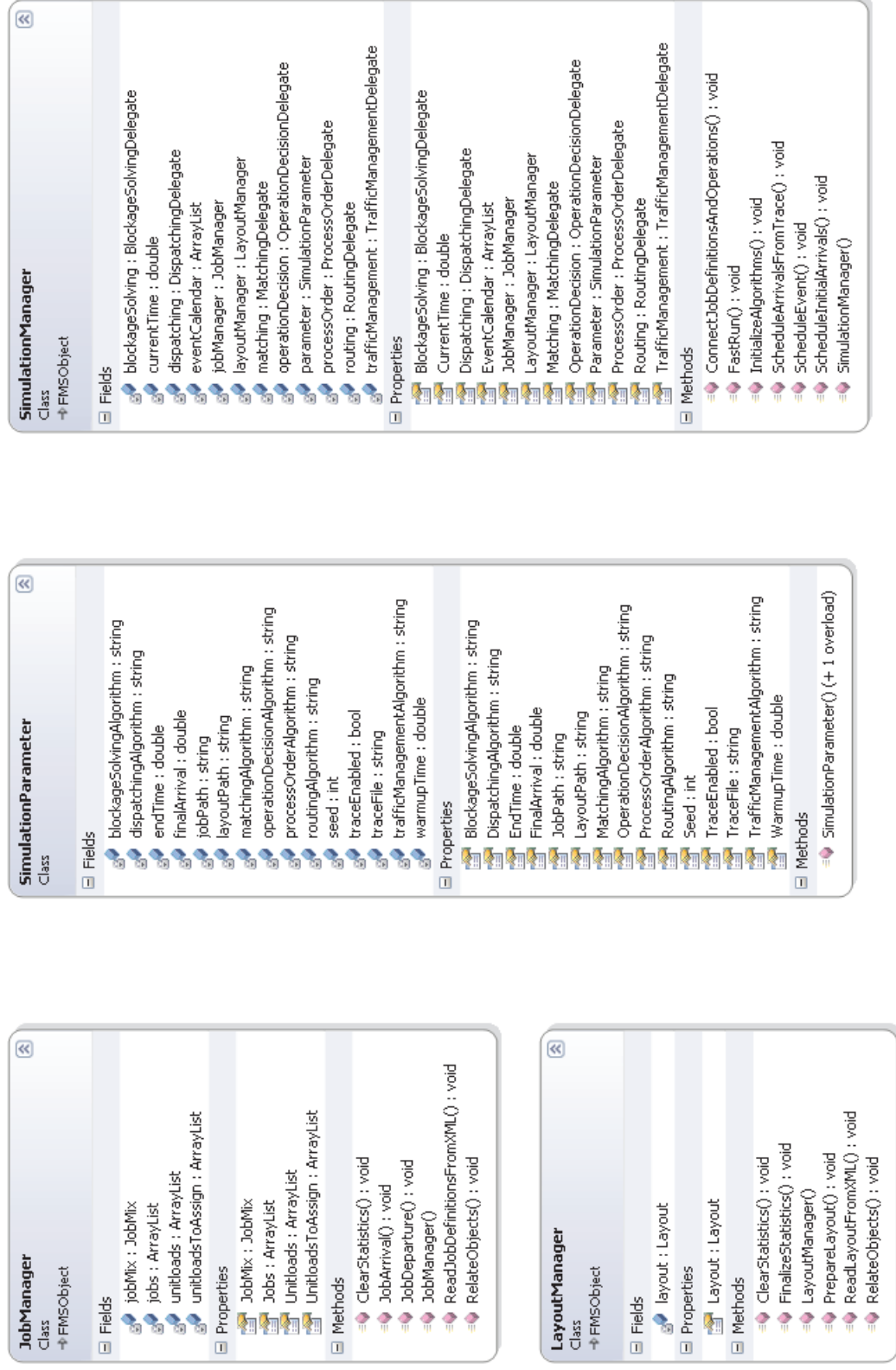


Figure C.8. FMS.NET class diagrams - VIII

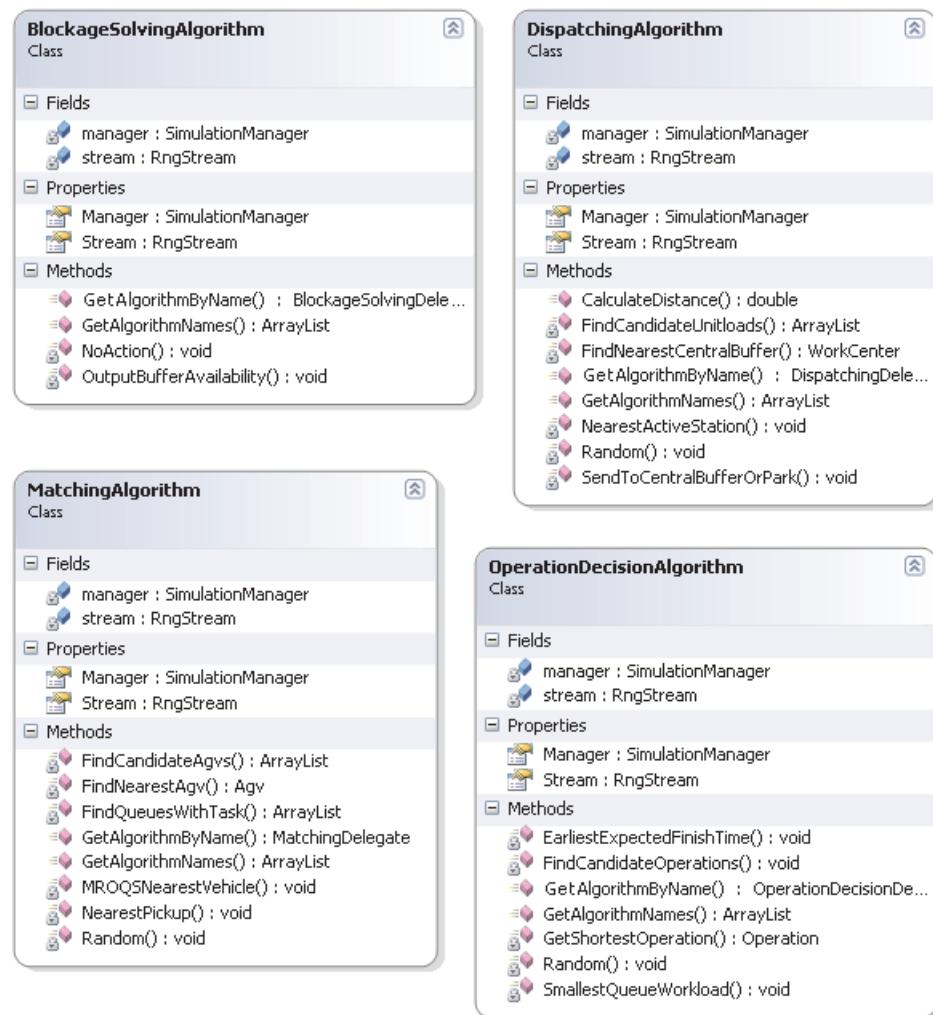


Figure C.9. FMS.NET class diagrams - IX

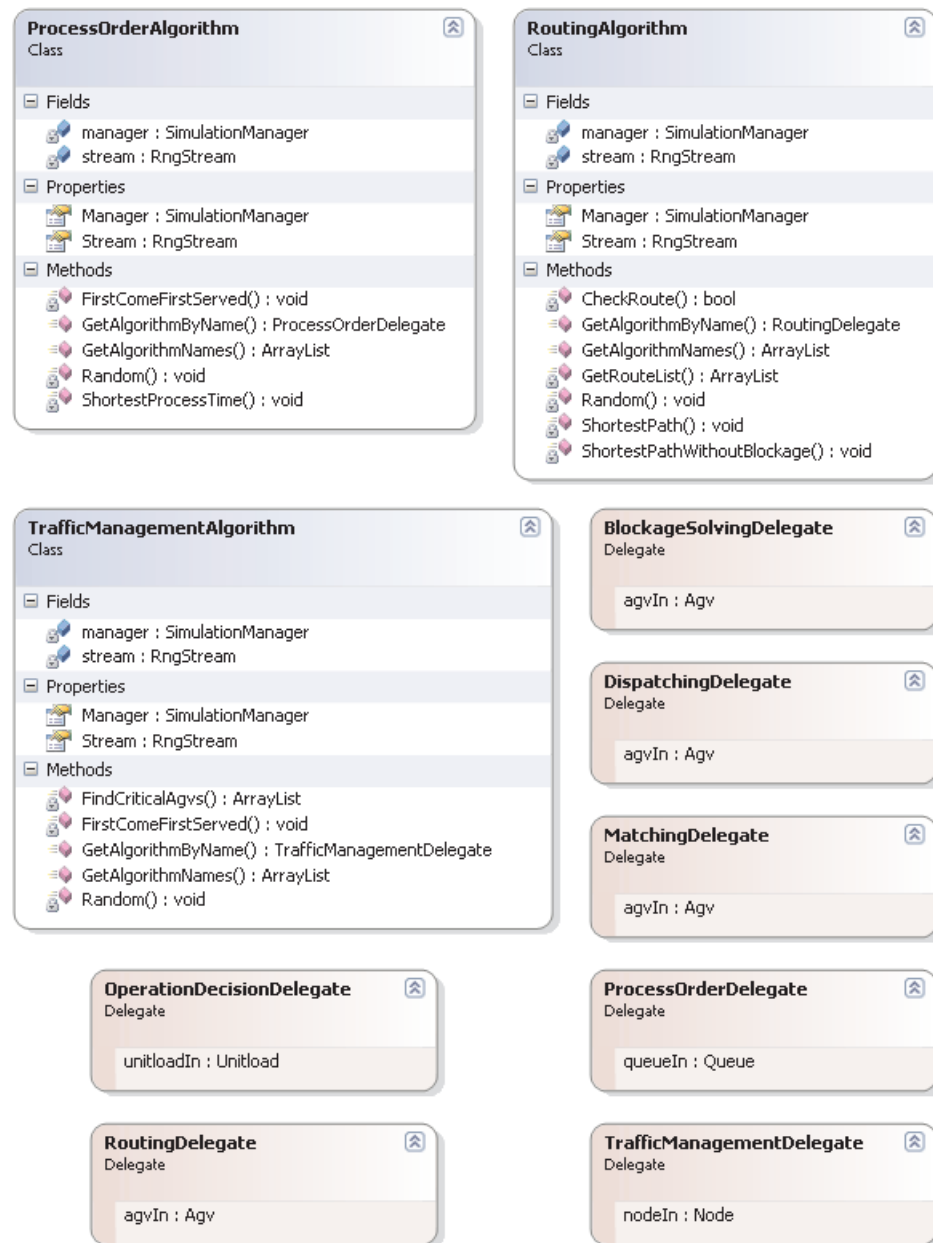


Figure C.10. FMS.NET class diagrams - X

APPENDIX D: SIMULATION RESULTS

Table D.1. Completed job count results for Experiment#1

Agv Count	1	2	3	4	5	6
Run#1	57	117	176	227	236	235
Run#2	60	119	174	221	239	240
Run#3	54	111	170	213	233	235
Run#4	59	123	178	219	227	229
Run#5	60	115	169	214	220	225
Average	58	117	173.4	218.8	231	232.8
Std. Dev.	2.55	4.47	3.85	5.67	7.58	5.85

Table D.2. Average flow time results for Experiment#1

Agv Count	1	2	3	4	5	6
Run#1	38734.49	30481.41	21941.49	15352.72	13678.55	13622.04
Run#2	39608.85	30055.56	22442.14	15979.02	14906.97	14415.52
Run#3	38637.39	30670.77	23546.32	16192.34	13107.84	13433.57
Run#4	38239.39	29972.69	21313.60	15720.43	14559.29	14329.14
Run#5	38370.65	30379.53	22889.50	16186.12	14827.54	14318.17
Average	38665.68	30333.69	22402.83	15872.64	14203.86	14015.42
Std. Dev.	536.21	292.77	857.67	355.09	788.37	459.10

Table D.3. Completed job count results for Experiment#2

Agv Count	1	2	3	4
Run#1	106	174	233	238
Run#2	85	174	230	243
Run#3	81	166	220	234
Run#4	88	182	218	231
Run#5	83	184	215	225
Average	88.6	176	223.2	234.2
Std. Dev.	10.06	7.21	7.85	6.83

Table D.4. Average flow time results for Experiment#2

Agv Count	1	2	3	4
Run#1	34019.90	22921.25	14607.60	13827.68
Run#2	35106.16	23533.00	15323.57	14131.64
Run#3	34810.53	24492.04	15022.25	12865.16
Run#4	34564.94	22314.46	15638.55	14935.14
Run#5	34291.29	24351.13	16068.85	14013.26
Average	34913.01	23438.84	15320.31	14025.66
Std. Dev.	425.73	928.39	560.74	741.49

Table D.5. Completed job count and average flow time results for Experiment#3

Operation Decision Algorithm	Job Count		Flow Time	
	EEFT	SQW	EEFT	SQW
Run#1	310	295	4045.18	5216.21
Run#2	312	291	4256.25	6437.08
Run#3	307	299	5113.78	6255.43
Run#4	308	300	3839.64	4997.27
Run#5	302	288	4347.77	6054.39
Average	307.8	294.6	4314.21	5724.86
Std. Dev.	3.77	5.13	485.07	644.76

REFERENCES

- Arena Simulation*, <http://www.arenasimulation.com>, 2005.
- AspectJ Project*, <http://www.parc.com/research/projects/aspectj>, 2005.
- Audsley, N., I. Bate and S. Crook-Dawkins, 2003, “Automatic Code Generation for Airborne Systems”, *Proceedings of the IEEE Aerospace Conference 2003*, Vol. 6, pp. 2863-2873.
- Batory, D., G. Chen, E. Robertson and T. Wang, 1998, “Design Wizards and Visual Programming Environments for Generators”, *Proceedings of the 5th International Conference on Software Reuse*, pp. 255-267.
- Bilge, Ü. and E. Albey, 2004, *Real Time Shop Floor Control With Process Plan Flexibility*, Boğaziçi University Technical Report FBE-IE-05/2004-08.
- Borenstein, D., 2000, “Implementation of an Object-Oriented Tool for the Simulation of Manufacturing Systems and Its Application to Study the Effects of Flexibility”, *International Journal of Production Research*, Vol. 38, No. 9, pp. 2125-2142.
- CodeDOM Grammar*, <http://www.mondrian-script.org/codedom>, 2005.
- Document Object Model Level 3 Core Specification*, <http://www.w3c.org/TR/DOM-Level-3-Core>, 2004.
- Doğan, D., 2001, *An Object Oriented Test-bed for Design and Real-time Control of AGV Systems*, M. S. Thesis, Boğaziçi University.
- Enterprise JavaBeans*, <http://java.sun.com/products/ejb>, 2005.

- Fertalj, K., D. Kalpic and V. Mornar, 2002, "Source Code Generator Based on a Proprietary Specification Language", *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, Big Island, Hawaii.
- Guerrieri, E., 1994, "Case Study: Digital's Application Generator", *IEEE Software*, Vol. 11, No. 5, pp. 95-96.
- Harada, M. and T. Mizuno, 1999, "Executable C++ Program Generation from the Structured Object-Oriented Design Diagrams", *Proceedings of 6th Asia-Pacific Software Engineering Conference*, pp. 630-636.
- Helman, T. and K. Fertalj, 2004, "Application Generator Based on Parameterized Templates", *Proceedings of the 26th International Conference on Information Technology Interfaces*, Cavtat, Croatia, pp. 151-157.
- Hyper/J Project*, <http://www.research.ibm.com/hyperspace/HyperJ>, 2005.
- Icron Technologies*, <http://www.icrontech.com>, 2005.
- Kernighan, B. and D. M. Ritchie, 1988, *The C Programming Language*, 2nd ed., Prentice Hall, Englewood Cliffs, New Jersey.
- Lewis, T., 1990, "Code Generators", *IEEE Software*, Vol. 7, No. 3, pp. 67-70.
- L'Ecuyer, P., 2001, "Software for Uniform Random Number Generation: Distinguishing the Good and the Bad", in *Proceedings of the 33th Conference on Winter Simulation*, pp. 95-105.
- Li, X. N., H. B. Yuan, X. Y. Huang and E. H. M. Cheung, 1998, "A New FMS Simulator with Object-Oriented Programming Techniques", *Journal of Materials Processing Technology*, Vol. 76, pp. 238-245.

McLaren, I. and T. Wicks, 2001, “Developing Generative Frameworks Using XML”, *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pp. 368-372.

Microsoft Developer Network Library, <http://msdn.microsoft.com>, 2004.

Object Constraint Language, <http://www.omg.org/docs/ptc/03-10-14.pdf>, 2005.

Park, D. H. and S. D. Kim, 2001, “XML Rule Based Source Code Generator for UML CASE Tool”, *Proceedings of 8th Asia-Pacific Software Engineering Conference*, pp. 53-60.

ProModel Simulation, <http://www.promodel.com>, 2005.

Rockstrom, A. and R. Saracco, 1982, “SDL-CCITT Specification and Description Language”, *IEEE Transactions on Software Engineering*, Vol. 30, No. 6, pp. 1310-1318.

SIMSCRIPT II.5 Simulation Language, <http://www.simprocess.com/products/simsript.cfm>, 2005.

Sommerville, I., 2000, *Software Engineering*, 7th ed., Addison-Wesley, Massachusetts.

Stroustrup, B., 1997, *The C++ Programming Language*, 3rd ed., Addison-Wesley, Massachusetts.

Unified Modeling Language, <http://www.uml.org>, 2005.

Voelter, M., 2003, A Catalog of Patterns for Program Generation”, *Technical Report*, Heidenheim, Germany.

XML Metadata Interchange, <http://www.omg.org/technology/documents/formal/xmi.htm>, 2005.

XML Query Language, <http://www.w3.org/XML/Query>, 2005.

XML Stylesheet Language Transformation, <http://www.w3.org/TR/xslt>, 2005.

Whalen, M. W. and M. P. E. Heimdahl, 1999, “On the Requirements of High-Integrity Code Generation”, *Proceedings of the 4th IEEE International Symposium on High Assurance Systems Engineering*, Washington DC, pp. 217-226.

REFERENCES NOT CITED

- Bruccoli M., S. N. L. Diega and G. Perrone, "An Object-Oriented Approach for Flexible Manufacturing Control Systems Analysis and Design Using the Unified Modeling Language", *The International Journal of Flexible Manufacturing Systems*, pp. 195-216, 2003.
- Cleaveland, J. C., "Building Application Generators", *IEEE Software*, Vol. 5, No. 4, pp. 25-33, 1988.
- Gullander P., S. Andreasson and A. Adlemo, "Database Design for Flexible Manufacturing Cells", *Control Engineering Practice*, pp. 1411-1420, 1998.
- Park J., J. Park and J. Kim, "A Generic Event Control Framework for Modular Flexible Manufacturing Systems", *Computers & Industrial Engineering*, pp. 107-123, 2000.
- Pelechano V., O. Pastor and E. Infran, "Automated Code Generation of Dynamic Specializations: An Approach Based on Design Patterns and Formal Techniques", *Data & Knowledge Engineering*, Vol. 40, pp. 315-353, 2002.
- Ray, W. J. and A. Farrar, "Object Model Driven Code Generation for the Enterprise", *Proceedings of the 12th International Workshop on Rapid System Prototyping*, pp. 84-89, 2001.
- Sabuncuoğlu, İ. and M. Lahmar, "An Evaluative Study of Operation Grouping Policies in an FMS", *The International Journal of Flexible Manufacturing Systems*, pp. 217-239, 2003.
- Shah M., L. Lin, R. Nagi and E. H. M. Cheung, "A Production Order-Driven AGV Control Model with Object-Oriented Implementation", *Computer Integrated Manufacturing Systems*, pp. 35-48, 1997.

- Shlaer, S. and S. Mellor, “Recursive Design of an Application Independent Architecture”, *IEEE Software*, Vol. 14, No. 1, pp. 61-72, 1997.
- Thibault, S. and C. Consel, “A Framework for Application Generator Design”, *Proceedings of the 1997 Conference on Software Reusability*, Boston, MA, pp. 131-135, 1997.