

**DISSAG: A SOFTWARE DEVELOPMENT ENVIRONMENT
FOR DISTRIBUTED INFORMATION SYSTEMS**

NIANPING ZHU

**DISSAG: A Software Development Environment
for Distributed Information Systems**

Nianping Zhu

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science in Computer Science
The University of Auckland

July 2001

© 2001 Nianping Zhu

Abstract

Distributed information systems (DISs) are increasing in usage rapidly. But the software development life cycle for DISs has changed very little. In such a cycle, a modeling tool is used in the analysis and design phase to help developers to build UML analysis and design models; and then in the implementation and testing phase, an IDE (Iterative Development Environment) is used for textual programming and program testing. The main problem with this cycle is the modeling cannot provide any direct help in the obtaining of the source code. Thus it is very inefficient.

The feasible solution to thoroughly solve this problem consists three activities: 1) defining a new model that is based on the UML model but capable of supporting modeling DIS in more details for the purpose of generating code; 2) defining a feasible way to build such a model for a given DIS; 3) defining a code generation logic that can access the model and generate source code.

DISSAG is the result of above activities. It is a component-based software development environment combining a modeling tool, a code generator and an IDE together seamlessly and, thus, supporting most DIS software development activities and automatically generating reasonably detailed source code based on the system specification. It has actually changed the traditional DIS software development cycle and enables its users to develop DIS software with an effective, efficient and reliable method.

In this thesis we address all issues with the actual development life cycle of DISSAG.

To Cathy and Fiona

Acknowledgements

First of all, I would like to express my sincere thanks to my supervisor, Dr. John Grundy, who have provided me invaluable guidance and support throughout the past one year. His insight, understanding, experience, and knowledge benefit me in the scope far beyond this thesis.

I also would like to thank those people who have spent so much time and energy on evaluating DISSAG and providing valuable feedback.

Finally, my immense thanks go to my wife, Qun. You have given me whatever a wife can give and more, and more.

Table of Contents

Chapter 1: Introduction.....	1
1.1 Motivation.....	2
1.2 Objectives	5
1.3 Methodology	5
1.4 Thesis Overview	6
Chapter 2: Background and related tools	7
2.1 Introduction.....	7
2.2 Background 1: DIS and its implementations	7
2.2.1 About DIS	8
2.2.2 About client/server	9
2.2.3 2-Tier architecture	10
2.2.4 3-Tier architecture	12
2.2.5 N-tier architecture	14
2.2.6 About middleware.....	14
2.2.7 JDBC	15
2.2.8 CORBA	17
2.2.9 RMI	19
2.2.10 DCOM.....	20
2.2.11 Comparison of CORBA, RMI and DCOM.....	21
2.3 Background 2: some concepts of software engineering	22
2.3.1 Software development life cycle.....	22
2.3.2 The Unified Modeling Language	27
2.3.3 CASE tool and MetaCASE tool.....	30
2.4 Related tools.....	31
2.4.1 Rational Rose	31

2.4.2	Argo/UML.....	34
2.4.3	JComposer.....	37
2.4.4	JBuilder.....	38
2.5	Summary.....	40
Chapter 3: DISSAG: requirements and specifications		41
3.1	Introduction.....	41
3.2	DISSAG overview	41
3.3	Functional requirements.....	43
3.3.1	DISSAG	43
3.3.2	Diagram Editor.....	45
3.3.3	Code Generator	48
3.3.4	Text Editor	49
3.3.5	Test Bed	50
3.4	Non-functional requirements	51
3.5	OOA Specification.....	52
3.5.1	DISSAG	52
3.5.2	Diagram Editor.....	53
3.5.3	Code Generator	55
3.5.4	Text Editor	57
3.5.5	Test Bed	59
3.6	Summary.....	60
Chapter 4: DISSAG: design		61
4.1	Introduction.....	61
4.2	What is the task.....	62
4.3	Rough design	62
4.4	DISSAG model design.....	64
4.5	Diagram Editor architecture design	65
4.6	DTD design.....	69

4.7	Code Generator architecture design.....	71
4.8	Code generation logic design.....	73
4.9	OOD class diagrams Summary.....	74
4.9.1	OOD class for Diagram Editor.....	74
4.9.2	OOD class diagram for Code Generator.....	79
4.9.3	OOD class diagram for Text Editor.....	79
4.10	Summary.....	80
Chapter 5: DISSAG: implementation		81
5.1	Introduction.....	81
5.2	Using JAVA as our implementation language.....	82
5.3	Using Java JFC/Swing to implement DISSAG interface.....	85
5.4	Using JComposer to implement Diagram Editor architecture.....	88
5.5	Using Sun JAXP 1.0 to parse XML file.....	91
5.6	To realize multiple threads by implementing the interface <i>Runnable</i>	92
5.7	Implementing test bed.....	94
5.8	Summary.....	96
Chapter 6: Video library: an example.....		97
6.1	Introduction.....	97
6.2	Scenario.....	97
6.3	Requirements.....	98
6.4	The steps to implement video library.....	98
6.4.1	Preparation.....	99
6.4.2	Specification phase.....	99
6.4.3	Code Generation phase.....	103
6.4.4	Further implementation phase.....	106
6.4.5	Test phase.....	107
6.5	Summary.....	108
Chapter 7: Usability evaluation and future work		109

7.1	Introduction.....	109
7.2	Usability Evaluation.....	109
7.2.1	About usability evaluation	109
7.2.2	The evaluation method – survey	110
7.2.3	Issues with evaluating DISSAG.....	111
7.2.4	Evaluation Results.....	112
7.3	Further Work.....	115
7.4	Summary	119
Chapter 8: Conclusion		120
8.1	Introduction.....	120
8.2	What we have done	120
8.3	What are the main contributions of this research.....	122
Bibliography.....		125
Appendix A: Model.dtd.....		A1
Appendix B: DISSAG metamodel.....		B1

List of Figures

Figure 1.1:	A typical software development cycle and tools	2
Figure 1.2:	Basic design of DISSAG.....	4
Figure 1.3:	DIS software development cycle using DISSAG.....	4
Figure 2.1:	2-tier architecture.....	10
Figure 2.2:	JDBC 2-tier architecture	11
Figure 2.3:	3-tier DIS architecture	12
Figure 2.4:	3-tier architecture using middleware.....	14
Figure 2.5:	JDBC architecture.....	16
Figure 2.6:	The basic CORBA architecture	18
Figure 2.7:	Relationships among RMI objects and classes	19
Figure 2.8:	DCOM architecture.....	21
Figure 2.9:	Waterfall Development Model	23
Figure 2.10:	Iterative development cycles.....	24
Figure 2.11:	Software development life cycle	24
Figure 2.12:	The top-level packages of the UML metamodel for class diagram.....	28
Figure 2.13:	Some examples of UML notations.....	29
Figure 2.14:	Rational Rose interface	32
Figure 2.15:	Rational Rose class specification windows.....	33
Figure 2.16:	Argo/UML interface.....	35
Figure 2.17:	Some features of Argo/UML.....	36
Figure 2.18:	JComposer interface	37
Figure 2.19:	JBuilder interface	39
Figure 3.1:	DIS Software development using DISSAG	42
Figure 3.2:	DISSAG OOA class diagram.....	52
Figure 3.3:	Diagram Editor OOA class diagram	53

Figure 3.4:	Code Generator OOA class diagram	56
Figure 3.5:	Text Editor OOA class diagram.....	58
Figure 3.6:	Test Bed OOA class diagram.....	59
Figure 4.1:	DISSAG conceptual model	62
Figure 4.2:	Rough design	63
Figure 4.3:	DISSAG metamodel	64
Figure 4.4:	DISSAG metamodel classes	65
Figure 4.5:	DISSAG notations	67
Figure 4.6:	Diagram Editor architecture	68
Figure 4.7:	Model.dtd	70
Figure 4.8:	Code Generator architecture.....	73
Figure 4.9:	OOD class diagram for Diagram Editor framework	75
Figure 4.10:	Some basic components of JViews	76
Figure 4.11:	Some BBW components	77
Figure 4.12:	Code Generator OOD class diagram	79
Figure 4.13:	Text Editor OOD class diagram.....	80
Figure 5.1:	Java code execution in JVM	83
Figure 5.2:	DISSAG interface layout	86
Figure 5.3:	Classes related to editor area.....	87
Figure 5.4:	Classes related to manager area.....	87
Figure 5.5:	DISSAG interface	88
Figure 5.6:	Creating base components using JComposer	89
Figure 5.7:	Creating view components using JComposer	90
Figure 5.8:	Creating view to base mapping using JComposer.....	91
Figure 5.9:	A thread example.....	93
Figure 5.10:	Test Bed interface	94
Figure 5.11:	Classes to implement Test Bed interface	95
Figure 5.12:	Test Bed working Mechanism	96

Figure 6.1:	DISSAG configuration dialog.....	98
Figure 6.2:	Diagram Editor in working with the example	99
Figure 6.3:	Inter-classes relations of Video Library	100
Figure 6.4:	Intra-class relationships of Rental class	101
Figure 6.5:	Base layer viewer	102
Figure 6.6:	Properties sheets for class, attribute, method and parameter.....	103
Figure 6.7:	DISSG interface when using Code Generator and Text Editor.....	104
Figure 6.8:	VideoLibrary.xml showed in a Text Editor window	105
Figure 6.9:	A generated file: runRental.bat.....	106
Figure 6.10:	RentalUserInterface.java	106
Figure 6.11:	RentalUserInterfaceG.java.....	107
Figure 6.12:	Using Test Bed tool.....	108
Figure 7.1:	The generated Rental user interface.....	113
Figure 7.2:	GUI composer as a view.....	115

Chapter 1

Introduction

For myself, I found that I was fitted for nothing so well as for the study of truth; as having a mind nimble and versatile enough to catch the resemblances of things (which is the chief point) and at the same time steady enough to fix and distinguish their subtler differences; as being gifted by nature with desire to seek, patience to doubt, fondness to meditate, slowness to assert, readiness to consider, carefulness to dispose and set in order; and as being a man that neither affects what is new nor admires what is old, and that hates every kind of imposture. So I thought my nature had a kind of familiarity and relationship with truth.

---Francis Bacon, "Of the Interpretation of Nature"

Suffering from hard working of hand writing java code, I have been always wondering if there is a way to generate source code automatically. So I asked myself: "Why not try it now and do it your way!"...

*---From a sent mail in which I answered my friend
why I chose code generator as my research topic*

DISSAG is the abbreviation for *distributed information system (DIS) specification and generation*, the name of a complete exploratory prototype of component-based software development environment for distributed information systems. We developed DISSAG for the purpose of solving some practical issues with developing a new type of tools which integrates a modeling tool and a IDE (Interactive Development Environment)

[Hamilton 1999] together and thus can provide a visual environment supporting almost all activities in each software development phase and can generate detailed source code for certain applications in a particular domain. This thesis will introduce the development process of DISSAG and concentrate on what DISSAG has contributed to the current software engineering research.

1.1 Motivation

A typical DIS software development life cycle can be roughly divided into five separate but associated phases: Requirements, Analysis, Design, Implementation and Testing [Larman 1997] [Hamilton 1999] (See Chapter 2 for detail). There are various CASE (Computer Aided Software Engineering) [Fuggetta 1993] tools available for each phase of the cycle. Typically, software developers use UML [UML 2000] modeling tools, e.g. Rational Rose [Rose Documents], in Analysis and Design phases, and IDEs (Interactive Development Environments), e.g. JBuilder [JBuilder 2001] in Implementation and Testing phase. Fig 1.1 shows the cycle after the Requirements phase and the tools used in each phases.

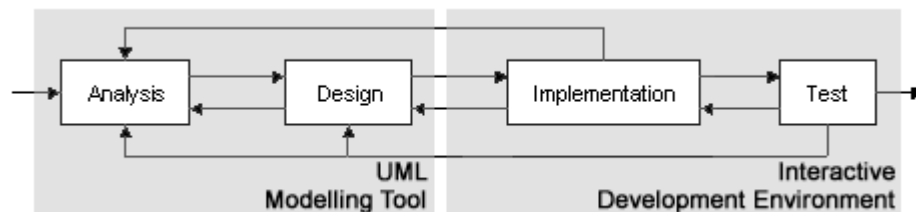


Figure 1.1: A typical software development life cycle and tools used

There are obviously some demerits with the cycle:

- The result of analysis and design does not provide any immediate help in writing source code. That means even developers has finished design of the architecture and solved other detail design problems, in implementation phase, they have to use textual programming tools to write the implementation code sentence by sentence even though often much of the code can be actually automatically generated.
- As a particular domain, DIS has been studied and practically implemented by a very large number of

researchers and developers for many years. Many of the architectures used have been well established and implementation methods have been well developed. So it is not always necessary to repeat some common parts of the design phase for each concrete DIS case.

- Textual programming is not only very inefficient, developers have to think about what is the right syntax to transfer the implement idea to the real code in a certain language and then input code into textual format files by hands, but also prone to errors, nobody can avoid typing and semantic errors due to the complicated syntax and behaviors.
- Developers have to be experienced and have to know DIS implement technologies such as CORBA, RMI, JDBC or DCom in detail so that they can derive OOD classes from OOA classes in design phase and write source code according to OOD class in implementation phase.
- Different tools are used in different phases. So it is hard to get information from the other phases. I.e. in implementation phase, if a developer wants to have a look at the OOA class diagram, he has to run another application to show the diagram. And it is impossible to communicate with each other in a transparent manner. I.e. if a developer has made a change in class relationship, the modification will not automatically appear in the source code. Moreover, extra effort has to make to learn how to use those different tools and there is extra cost to buy those tools.

Our motivation for this project comes from trying to solve some of above problems. That is to develop a tool that has these significant features:

- Providing a software development tool which combines a CASE tool and an IDE seamlessly, and using which the developers can perform most of the DIS software development tasks, including analysis, design, implementation and test, in a more convenient and more efficient way.
- Capable of automatically generating the common, mechanical implementation source code for DIS based on the system specification.

Fig 1.2 shows our basic design of DISSAG. The modeling tool here will be an UML-like diagram editor that supports DIS modeling and specification. Code Generator will be a tool that can take the specified DIS model and generate implementation source code. To achieve higher efficiency, the IDE we will integrate in DISSAG

can be a third party tool such as JBuilder. But we will provide our own IDE that consists of a Text Editor and a Test Bed.

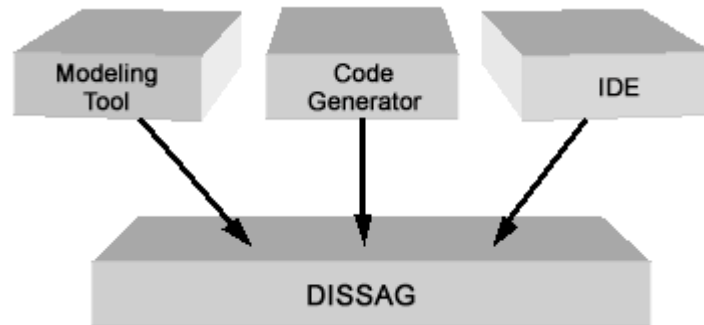


Figure 1.2: Basic design of DISSAG

By deploying DISSAG, the development life cycle (after Requirements phase) for DIS software will consist of four new phases: Specification, Generation, Further Implementation and Testing. There will be a sub-tool in DISSAG for each of these phases correspondingly: Diagram Editor, Code Generator, Text Editor and Test Bed.

Fig 1.3 shows the new development cycle for DIS software development using DISSAG.

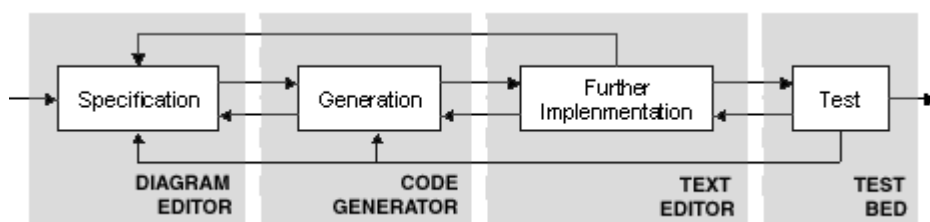


Figure 1.3: DIS software development cycle using DISSAG

The development cycle has at least these following advantages over its previous:

- *Very efficient.* Much of design task will be automatically performed and much of the implementation code will be generated automatically.
- *Very reliable.* The generated code is extracted from the previous solutions to DIS and is

recommended by experienced developers and has been proven to be possibly the best solution.

- *Very convenient.* All sub-tools are integrated into a single environment. Users can finish the whole development cycle with them. Changing sub-tools, or in other words, changing working phases, is quite easy.

1.2 Objectives

Our objective is to develop a complete exploratory prototype of software development environment for DISs. More accurately, we will try to change the old DIS software development life cycle into an efficient, reliable and convenient one by providing a powerful CASE tool, DISSAG, which will bring its users a DIS software development environment supporting most development activities and featuring automatically performing design tasks and automatically generating detailed implementation source code.

To achieve this goal, a range of software engineering topics will be involved:

- How to visually specify a distributed information system?
- How to store the information collected from the system specification and pass it to its potential user?
- How to generate implementation source code based on the system specification?
- How to make the generated code extendable?
- How to integrating associated tools into a visual environment.
- How to develop a code generator applied to a certain domain.
- How to handle component-based software architectures?

Although DISSAG is just an exploratory prototype, it is aimed to solving above questions or at least to provide a feasible optional or partial solution to them. The experience gained from developing DISSAG will be helpful for the study in the same field, and thus, undoubtedly, contribute to the current software engineering research.

1.3 Methodology

Initially we have thoroughly studied the DIS concentrating on questions such as what are the architectures of many common DISs, what are the middleware solutions used for DIS implementation, what are the similarities and dissimilarities of the middleware solutions, what information needed to be specified for the purpose of generate code, and what code pieces can be generated etc. Then we have researched the related tools comprehensively with the questions such as what are their architectures, interfaces and underlying working mechanisms, what are their merits and demerits etc. The results of the study and research have been outlined in the next chapters. Then we derived the requirements for DISSAG and performed very detailed OOA&D. We then have actually implemented DISSAG based on the design. At last, we have evaluated DISAG and established the potential future work for DISSAG.

1.4 Thesis Overview

The rest of this thesis is organized as follows:

- Chapter 2: Background and the related tools: – provides background information related to two main fields, DIS and software development life cycle, and also introduces some of the related tools.
- Chapter 3: DISSAG: requirements and specifications – introduces requirements and OOA specifications of DISSAG.
- Chapter 4: DISSAG: design – introduces all activities involved in the design phase, including the design of DISSAG model, the architecture design and OO design of DISSAG and its sub-tools.
- Chapter 5: DISSAG: implementation – discusses issues with implementing DISSAG.
- Chapter 6: Video library: an example – presents an example to implement a video library system using DISSAG.
- Chapter 7: Evaluation and future work – evaluates DISSAG and indicates the avenues of future research.
- Chapter 8: Conclusion – summarizes the contributions made by this thesis to the field of research.

Chapter 2

Background and related tools

2.1 Introduction

DISSAG is a software development environment supporting most DIS software development activities and featuring automatically performed design tasks and automatically generated detailed implementation source code for DIS. Building DISSAG thus involves knowledge in two main fields, one relating to software engineering, mainly software development life cycle, and UML, another one relating to the DIS, which is the working domain of DISSAG. In this chapter we will cover background material related to these two fields, which is very important and helpful in understanding DISSAG's goals, design and implementation.

As a CASE tool consisting of a modeling tool, a code generator and an IDE, DISSAG has many related tools. In this chapter we provide a section to briefly introduce four typical related tools and we will also explain what are the differences between DISSAG and them.

2.2 Background 1: DIS and its implementations

In this section, we will briefly introduce the distributed information system (DIS) and its implementations. At first, we will describe what "DIS" means. Then we will introduce some common DIS software architectures such as 2-tier, 3-tier etc and discuss the advantages and disadvantages of them. After that, we will give a brief introduction to JDBC and then three common kinds of middleware, CORBA, RMI and DCOM. We will also compare CORBA, RMI and DCOM side-by-side at the end of this section.

2.2.1 About DIS

DIS is the abbreviation for distributed information system. Information is the data which has been presented in a way that is meaningful to the beholder. Information system is a type of system which generates, manipulates and communicates information [McDERMID, 1990]. Distributed information system thus is an information system of relatively independent subsystems which are, however, tied together within the organizational framework by communication interfaces.

The best way to understand DIS is comparing it to its opposite, the integrated system. Physically, a integrated system [Pritchard *et al* 1999] runs on a single machine, thus has only one data processing unit and one database. Users can obtain information from this single machine only. It cannot communicate with the other integrated systems therefore it is impossible to share data or share data processing unit with them. It obviously cannot satisfy the need of the modern information society. The only solution is to connect these individual systems together using various networks into a big system. The result is DIS. Those individual systems then became the subsystems of the DIS. They can still keep certain data store and data processing unit, but to take the advantages of the network, all data store and data process units will be redesigned and rearranged in the scope the whole DIS. DIS makes it possible for its subsystems to share data and share data processing facilities properly thus the whole system works more flexible and more efficient. The mature network technologies and middleware technologies make physical locations and machine types less important and enable computers (called as clients) to utilize as many as other computers (called servers) as they want.

DIS thus has these features:

- Multiple servers and clients. There can be not only multiple clients but also multiple servers in a DIS.
- Location independent. The clients and the servers can be anywhere if they are connected by a network.
- Machine independent. The clients and the servers can use different types of computers.
- Using networks to physically connect all participating computers and using middleware to handle the communication over networks.
- Using, usually, relational databases.

2.2.2 About client/server

As mentioned above, a DIS is actually a system that enables its users to get information from a certain location. The first important thing that we should pay attention here is that the information is totally different from the raw data. It is the key to help us to understand most important concepts in DIS.

In most cases, each DIS has at least one database that will be used to store raw data. The raw data are normally stored in the database in a simple form as a number, a string or a Boolean, even though they can be as complicated as articles in text files or html files, or some other forms [Rob *et al* 1997]. They can be stored to or retrieved from database by proper means.

A piece of information can be the straight presentation of a piece of raw data or in most cases, can be a processing results based on one piece or more pieces of retrieved raw data. For example, assume we have a very simple table called “student” in a database, and there are fields called “ID”, “name”, and “average mark”, if we want to get information about “Who is the name for the student whose ID is 2334135”, we may immediately get the answer by submitting a query and the raw data “Nianping Zhu” returned from the query is the information we want thus there is not any processing necessary with the raw data. But if the information we want is “Is the student whose ID is 2334135 eligible to enroll in a Master thesis?”, then there will be no query can return yes or no directly. Thus we need an intermediate logic that will do the following:

- Take the request from the user.
- Create a query that will get the “average mark” for ID 2334135 and submit the query to the database.
- Compare the returned raw data with “A-“, return “yes” if it is A- or above, “No” otherwise.
- Pass the result to the user.

This example not only gives us the idea that how information is different from raw data, but also helps us to understand the key concepts in DIS.

The machines where the database servers nest are called servers. The machines that users use to get information are called clients. Of course, to let users properly input their request and get the result, there will be a user interface running on each client machine. The intermediate logics are called business logics.

In most cases, the databases in DIS are relational databases now. Each server runs a corresponding DBMS to manager the database. Structured Query language (SQL) will be used to create queries that will be submitted to the database. The SQL is a nonprocedural language and so it is impossible to embed any data processing logic into its syntax. Therefore the data processing must be done in somewhere else. This is why the business logic is needed [Orfali *et al* 1994] [Orfali *et al* 1994].

In a simple client/server system, business logic can appear in client side, server side or the both. When business logic appears in the client side, it can be easily nested in the client side applications. When it appears in the server side, it will be stored into database in the form of ‘stored procedure’ which can be triggered to process related raw data and return the result to certain queries.

2.2.3 2-Tier architecture

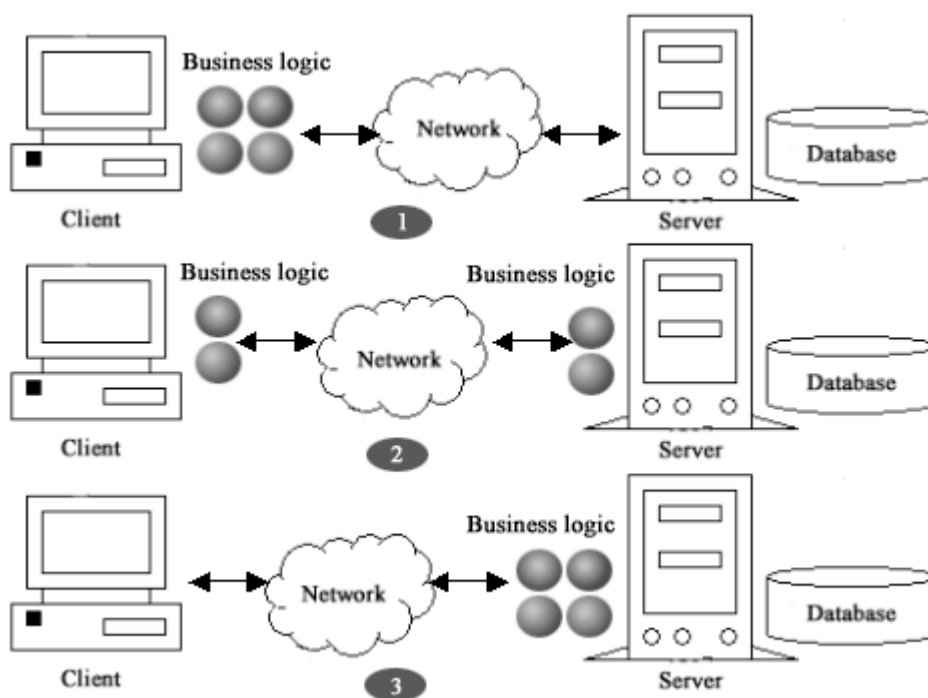


Figure 2.1: 2-tier architecture

The client/server system we introduced in 2.2.2 is actually a DIS with 2-tier architecture. Fig 2.1 shows the 2-tier architecture. To best understand this architecture, we will continue the topic of the business logic here.

When most, if not whole, of the business logic runs on the client machine, the client then is called a “fat client” (See Fig 2.1 (1)). The architecture under this situation is called a pure 2-tier architecture. In this case, the server has very limited ability to process data, thus most raw data have to be transferred to the client side to be processed. Network workload therefore is heavy and so is the client workload.

So a better way to distribute workload is to move part of business logic to server side. The architecture then becomes a less pure 2-tier architecture (See Fig 2.1 (2)). In this case, the server will be able to process data to certain extent by the means of executing the “stored procedure” as we talked about in 2.3.

The extreme situation is occurred when most of the business logic runs on the server side. The client is called “thin client” now (See Fig 2.1 (3)). As it cannot process user’s queries, all queries will have to be sent to server, so the network is also busy.

To implement 2-tier DIS architecture in JAVA, JDBC is the sole option to be used (We will introduce JDBC in details later). Fig 2.2 shows the JDBC 2-tier architecture [Orfali *et al* 2000].

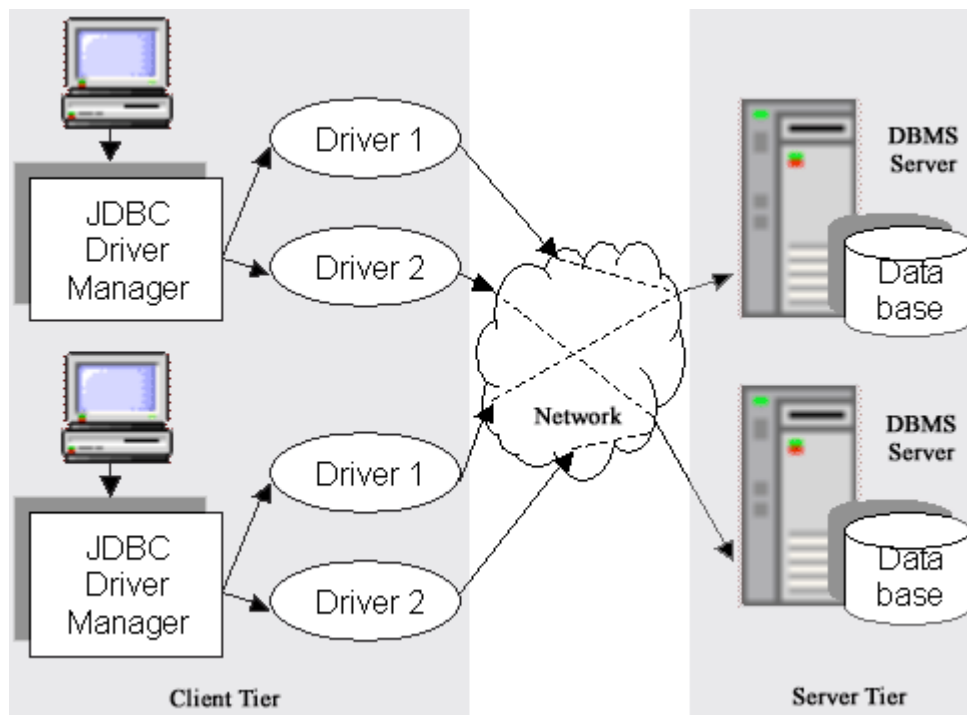


Figure 2.2: JDBC 2-tier architecture

There will be also two forms of the 2-tier architecture: pure and less pure. In the pure 2-tier architecture, the client is a “fat” client and it not only maintains JDBC drivers for different database types but also runs the business logic. In a less pure 2-tier architecture, part of the business logic will be moved to the server side.

One problem should be mentioned here is that the stored procedure is not standard like SQL. Different database vendors provide different methods to realize it. So if the stored procedure is used, then the system will be limited to a certain database.

2.2.4 3-Tier architecture

We have mentioned in 2.3 that when business logic appears on the client side, it is nested in the client’s applications. It is possible to separate it from user’s applications and put it on the third machine. This brings out a new tier, the middle tier. And the architecture now is so-called 3-tier architecture. Fig 2.3 shows a typical 3-tier DIS architecture.

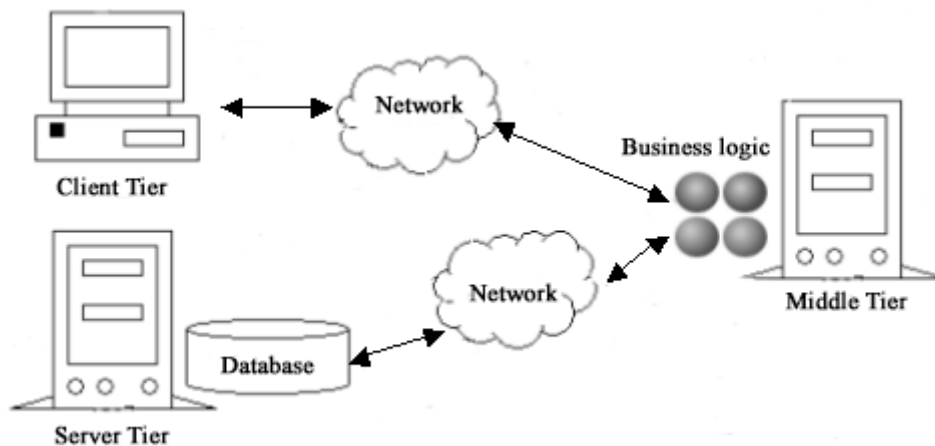


Figure 2.3: 3-tier DIS architecture

From this figure, we can see the new tier, the middle tier, appearing between the server tier and the client tier.

The functions of each tier can then be outlined as the following:

1. Client tier: providing a user interface that visually presents information asked by users.
2. Middle tier: running the business logic to processing data.
3. Server tier (Data tier): running a DBMS for persistent data management.

Some advantages can be seen with the 3-tier architecture:

1. By proper design, the middle tier can be close to the server tier. That means the business logic can be performed in a location close to database while as the client can be located anywhere. The direct benefit is that network workload will be reduced.
2. All clients that ask for the same business logic to processing data can share the same middle-server; therefore it will be easy to maintain or upgrade the business logic.
3. Client-tier applications now will communicate with middle-tier servers only; therefore can be developed independent of database in the server tier.
4. New client requests can be easily handled by adding a corresponding new middle-server. Therefore the 3-tier systems are more flexible than the 2-tier ones.

The communication between the middle tier and the server (data) tier in 3-tier systems is nearly the same as the communication between the client tier and the server tier in 2-tier ones and is relatively simple because the database drivers have abstracted the various network protocols. But the communication between the middle tier and the client tier is much complicated. Middleware (We will talk about it in next section) was used to help solving the related problems.

There are three common middlewares used to implement the 3-tier DIS architecture, CORBA, RMI and DCOM. Fig 2.4 shows a typical 3-tier architecture using middleware [Orfali *et al* 2000].

From the figure, we may see that the communication between client tier and the middle tier is handled by distributed objects provided by the middleware. Just like those database drivers that abstracted all network protocols, the distributed objects will enable the client tier and the middle tier to communicate with each other regardless their locations.

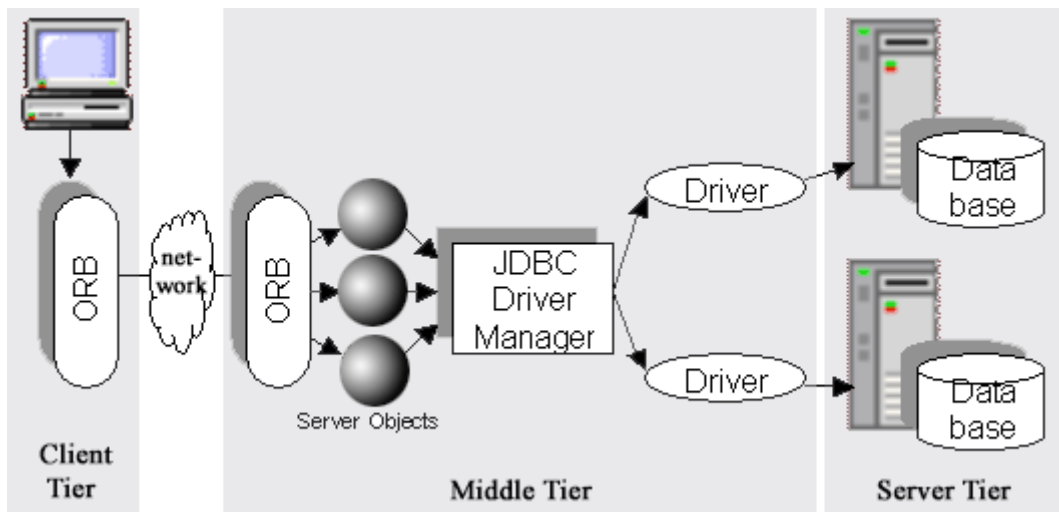


Figure 2.4: 3-tier architecture using middleware (reproduced from [Orfali et al 2000])

2.2.5 N-tier architecture

From previous section, we have seen that separating business logic from the client results in a new tier, the middle tier. In fact, the business logic in the middle tier still can be further divided into a number of tiers. Each tier will provide a certain service by running the corresponding logic. Then the 3-tier architecture becomes a N-tier architecture. According to this definition, each tier of a N-tier system simply represents a division of work across multiple computers.

The N-tier architecture is useful when the business logic is very complicated. The fine tiers make it easy to maintain or upgrade the business logic units and allow for greater security, reliability, and scalability [Pritchard *et al* 1999].

2.2.6 About middleware

Physically, any DIS consists a set of computers connecting together by certain networks. The communication between deferent tiers is actually, in most cases, communication between deferent computers that rely on some networking protocol such as TCP/IP. If there is no middleware, the implementation of DIS (when 3-tier

architecture is used) would definitely be very complicate as all communication between tiers would have to be implemented at network level.

In the last section, we have investigated the DIS architectures and mentioned that the implementation of 3-tier architecture is much complicated than that of 2-tier. In 2-tier systems, the relational database vendors provided drivers to access the database that abstracted the various network protocols. Client applications can be developed independent of the actual location of the server tier database and communicate with the database either statically or dynamically.

But in 3-tier systems, there are two kinds of communications involved. One is the communication between middle tier and server tier. This is handled by database drivers the same as what happened in 2-tier system. Another one is the communication between client application and the middle-tier services. Any database drivers, of course, cannot do this. Therefore there is a need for a new mechanism. Middleware is born to meet this request.

Middleware is used to connect different layers in a DIS, hide the complexity of the communication over the network, and provide some extra functions such as locating remote objects etc.

There are many kinds of middleware existed. Some of them are in procedure style like RPC (Remote Procedure Call). As object-oriented programming languages such JAVA become more and more popular, distributed middleware becomes the dominant force of this domain. The three outstanding examples are CORBA, RMI and DCOM.

2.2.7 JDBC

When we implement DIS with JAVA, we face lots of problems related to the database. In previous sections, we have mentioned that in DIS, most databases are relational databases, or in other words, SQL databases. In real life, they are very complicated: 1) they are supplied by different vendors; 2) they can be anywhere in a

DIS; 3) the machines they are in can be any type and can be running on any operation platform. So there are many problems to be solved for the client applications. How can they find the right database? How can they treat each kind of databases individually in a proper way? How can they handle different machine types and different operation systems? And how can they communicate with the databases over the networks? All these questions can be included into one, how to make remote databases transparency to the client applications.

The JAVA Database Connectivity (JDBC) [Java API] [Heller 1999] takes solving above problems as one of its objectives. JDBC provides a driver interface that all database vendors must adapt to their particular database products. Each database vendor then provides a special driver, which can be used by JDBC manager, to talk to the corresponding database. The JDBC manager can automatically load the right driver for any given database. The drives abstract the various network protocols and so enable the JAVA applications to access databases of any type and in anywhere (See Fig 2.5).

But there are still some other problems. A SQL database normally accepts queries in SQL syntax only and returns results in certain forms. How can the JAVA applications connect or disconnect to a database, submit a query and get the result back properly?

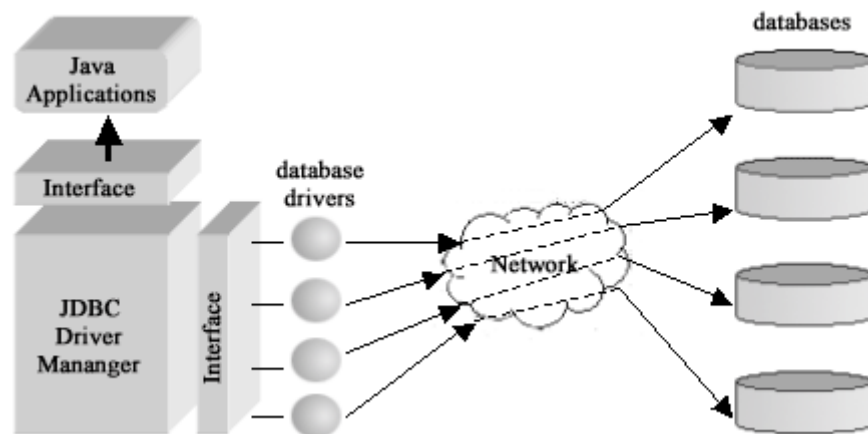


Figure 2.5: JDBC architecture

JDBC can solve this problem by provide a JAVA SQL “wrapper”. That means all the operations to the

database are still performed by SQL instructions. The wrapper is actually an application interface that lets JAVA application access SQL services. For example, JDBC provides a Connection interface that can be used to open or close a connection to a database, a Statement interface for executing SQL statements on a given connection, a ResultSet interface for encapsulating the rows returned by the database after queries were executed (See Fig 2.5).

Although the underlying mechanism is complicated, the function of JDBC can be looked as very simple, that is to let JAVA applications access a given database. And the actual programming code is very simple as well. The high-level abstraction releases DIS programmers from the sophisticated details. JDBC thus become the inevitable part in all JAVA implementations of DIS.

2.2.8 CORBA

When the business logics appear in the client tier, it normally nests in the client applications. Therefore it is very easy for the client applications to get services from certain business logics. When the business logics are moved to a new machine and thus form a new middle tier, lots of problems occur.

The client applications still need getting services from the business logics. But how the services will be provided, how to find the right services and how to handle the communication over the network? The Common Object Request Broker Architecture (CORBA) provides good answers to all these questions.

The services will be provided by CORBA distributed objects. CORBA distributed objects are living in the form of binary components. So they can be anywhere over a network and can be accessed by any clients. That means the client applications can be in any programming language and running on any machine with any operational system. When the objects are instantiated, they are registered as the run-time objects with a registry. So the client applications can find the right CORBA objects via the registry and thus get the right services.

The communications between the client applications and the CORBA distributed objects are handled by a unique mechanism of CORBA. CORBA has its own Interface Definition Language (IDL). The IDL can be used by objects to tell their potential clients what operations are available and how they should be invoked. All CORBA distributed objects have an interface defined with IDL. Compiling the IDL files using a suitable compiler will produce stubs on the client side and skeletons on the server side. The stubs provide static interfaces to their corresponding remote object's services and the skeletons provide the same interface but on the server side. Just like those database drivers in 2-tiers architecture systems hide the network protocols from the application programmers, the stubs and the skeletons do the same thing and make it easy for the application programmers to handle the communications between the client applications and the CORBA distributed objects [Vogel 1998] [Mowbray *et al* 1995].

A typical CORBA architecture is showing in Figure 2.6. As we can see, CORBA supports both static and dynamic remote method invocation. The Dynamic Invocation Interface (DII) and the Dynamic Skeleton interface (DSI) are used when a dynamic invocation is performing. The other components include: the Interface Repository containing all IDL-defined interfaces, the Implementation Repository containing information of all server-side classes, objects and the implementation of ORBs, the Object Adapter providing the run-time environment for instantiating server objects, passing requests to them, and assign them object Ids, the ORB providing a mechanism for the client applications to transparently communicate with server object s.

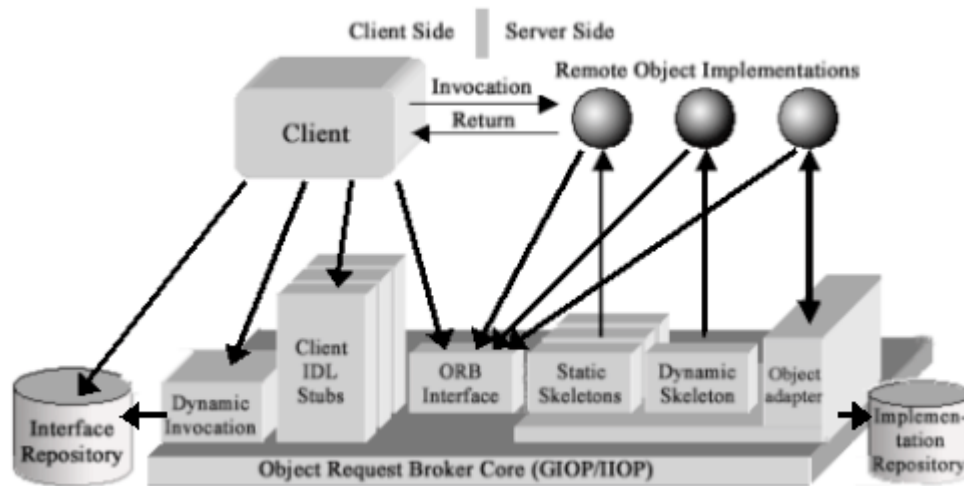


Figure 2.6: The basic CORBA architecture

2.2.9 RMI

Remote Method Invocation (RMI) is a JAVA package that supports remote method invocations on objects running in different JAVA virtual machines, and has been built into the core JAVA API. It can be used for JAVA client applications to interact with remote objects across the network [Flanagan *et al* 1998].

Unlike CORBA, RMI has not deployed any IDL. Instead, it defines a remote interface using a JAVA interface that extends `java.rmi.Remote` interface. The server classes can be created by implementing the remote interface and extending `java.rmi.UnicastRemoteObject` class. All of these JAVA files can then be compiled by any JAVA compiler to generate `.class` files. After that, all stubs for those `.class` files will be generated by a stub compiler. These stubs run on both the client side and the server side (called skeleton also) and provide the interface that client application and server objects use to interact with each other. Just like those stubs and skeletons in CORBA architecture, the stubs and skeletons abstract all network protocols and make client applications and server objects transparent to each other.

Fig 2.7 shows the relationships among remote object, stub, and skeleton classes. It also shows how these

objects are obtained.

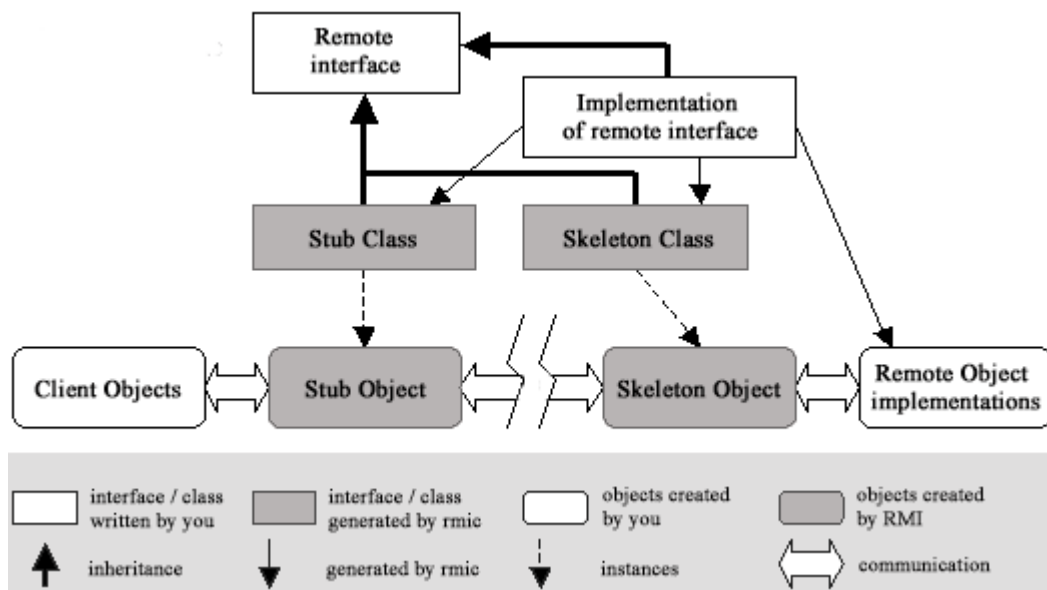


Figure 2.7: Relationships among RMI objects and classes (reproduced from [Flanagan et al 1998])

When client applications need to get service from a certain remote object, one important thing to do is to find the right remote object. RMI has a mechanism to solve this problem. RMI has a RMI registry application that can run on any server machine. When an instance of the remote object is created, it is registered with the RMI registry using the `java.rmi.Naming` class. The client applications thus can also use `java.rmi.Naming` class to locate a remote server object.

2.2.10 DCOM

While CORBA and RMI have many similarities in their architectures and services, Microsoft's Component Object Model (COM) provides a different way for software components to communicate with each other.

With CORBA and RMI, the server objects are completely transparent to the client applications. They provide mechanisms to guarantee the client applications access the right remote objects no matter where they are. DCOM explicitly indicates the locations of the server objects and divides them into three different types according to their running locations [Rock-Evans 1998] [Rosen et al 1998] [Rubin et al 1999].

- In-process server. DCOM implements an in-process server as a dynamic linked library (DLL) that can be loaded into client's application process and can be executed within the same process space as the client application. In this case, the server and the client are mixed into a single application, so it is very easy to get desired service from the server. Thus high performance is achieved.
- Local server. DCOM implements a local server in a Microsoft's own executable (.EXE) file that can be executed alone in a separate process space on the same machine as the client application. Communication between an application and a local server is accomplished by the DCOM runtime system using a high-speed interprocess communication protocol, called Lightweight Remote Procedure Call (LRPC).
- Remote server. DCOM implements a remote server in the same executable (.EXE) file as that for local server but it will be executed alone in a remote machine. Communication between a client application and a remote server is accomplished by the DCOM runtime system using Remote Procedure Call (RPC) protocol.

Fig 2.8 shows the DCOM architecture when all these three servers are being using. From this figure we will see that the communication between the client and the server (both local and remote) are handled by Proxy and Stub objects. Their function is similar as the function of the stub and skeleton objects in CORBA and RMI in the view of abstracting the network protocols.

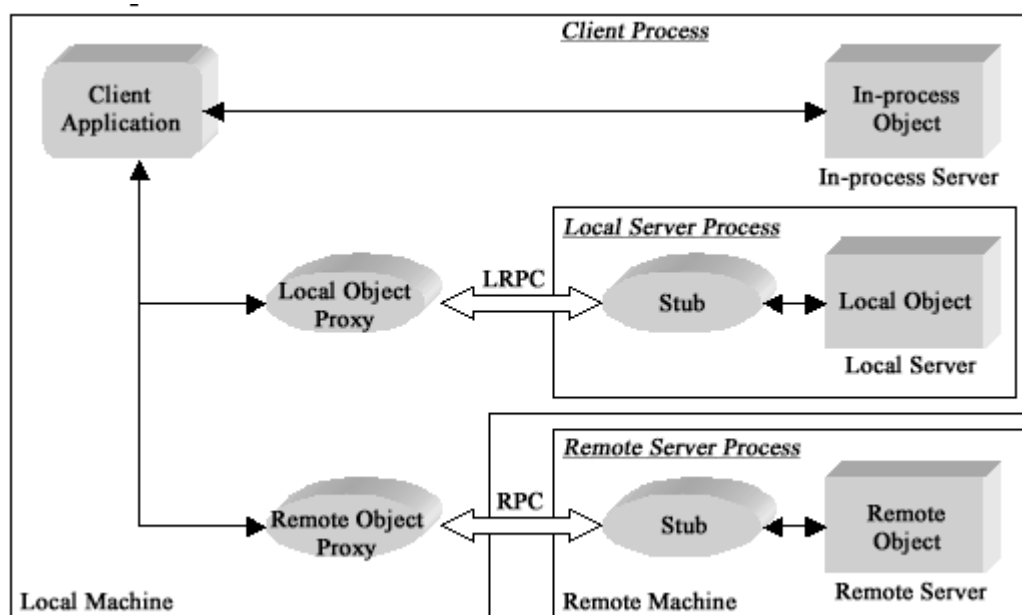


Figure 2.8: DCOM architecture

2.2.11 Comparison of CORBA, RMI and DCOM

CORBA, RMI and DCOM are the dominant technologies to construct DIS applications. All of them provide high-level abstraction and thus enable developers to concentrate on their business logic by hiding all low-level details. Here we provide brief comparisons of them [Pritchard *et al* 1999] [Orfali *et al* 1998].

CORBA, RMI and DCOM all have stub/skeleton (proxy/stub in DCOM) interfaces that enable client applications and server objects communicate with each other transparently. They all separate the object interface from its implementation. CORBA and DCOM require that all interfaces be declared using an Interface Definition language (IDL). But CORBA IDL and DCOM IDL are different and are not compatible with each other. RMI declares object interfaces with java interface extending java.rmi.Remote interface.

CORBA and DCOM support dynamic invocations, and dynamically discovering a remote object's interface while RMI support static invocation only. CORBA and RMI have persistent objects that also have persistent object references. So client applications can reconnect to exactly the same object instance with the same state at a later time. DCOM objects are stateless. In DCOM, client applications are given a pointer to access the functions in an interface; this pointer is not related to state information. Thus DCOM is not good at implementing applications that heavily rely on state information.

CORBA is really platform-independent and thus runs on all major operation systems; RMI runs on all OS platform with JAVA virtual machine (JVM) as RMI objects can live in a JVM only. DCOM has best performance only when it runs on Windows OS even though it can run on other OS platform such as UNIX.

CORBA is language-independent. CORBA objects are in binary form and can be created with any language. DCOM is normally written in C/C++, Microsoft provides JAVA adapters but its Java implementation can run in its own JAVA VM on windows OS. The other platform will not support this kind of Java VM. RMI is

totally JAVA-based and not supported by any other languages.

2.3 Background 2: Some concepts of software engineering

In this section we will first discuss the concept of software development life cycle and list major activities in each phases of this cycle. Then we will outline the UML as it is frequently referenced in this thesis and in the DISSAG development process. At last, we will give a very short introduction to CASE tools and metaCASE tools.

2.3.1 Software development life cycle

It is very hard to accurately describe how a software life cycle looks like as it is affected by lots of varying factors. We can put them into a long list with little effort. Some obvious examples are the size and the organizational way of the development team, the size and the working domain of the software, the development language and the development tools etc.

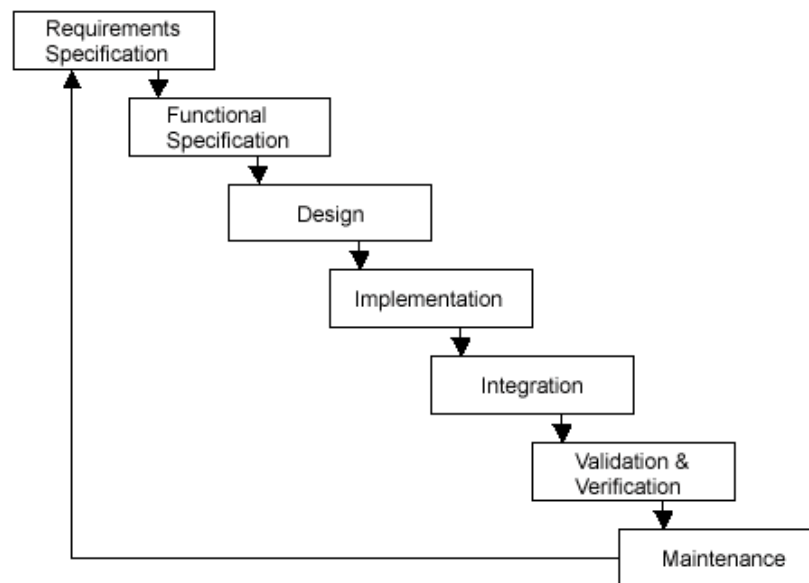


Figure 2.9: Waterfall Development Model (reproduced from [Hamilton 1999])

In his book “Software Development—Building Reliable Systems”, Marc Hamilton introduced two typical models for software development life cycle. The first one is the Waterfall Development model that was introduced in 1970 by Winston Royce in his paper “managing the development of large software systems”. In this model, software life cycle is divided into seven stages, requirements specification, functional specification, design, implementation, integration, validation and verification, and maintenance (See Fig 2.9).

The second one is called Spiral Development Model, which was developed by Barry Boehm in his 1998 IEEE Computer article, “A Spiral Model of Software Development and Enhancement”. This model divided software development life cycle into four main activities, planning, Risk analysis, engineering, and Customer evaluation.

Craig Larman introduced an iterative software development life cycle in his book “Apply UML and Patterns” (1998). He included the development activities into different levels and indicated that the development process is actually an iterative process (see Fig 2.10). In the high level, activities include Plan and Elaborate, Build, and Deploy. Then Build is divided into iterative development cycles that contain low-level activities.

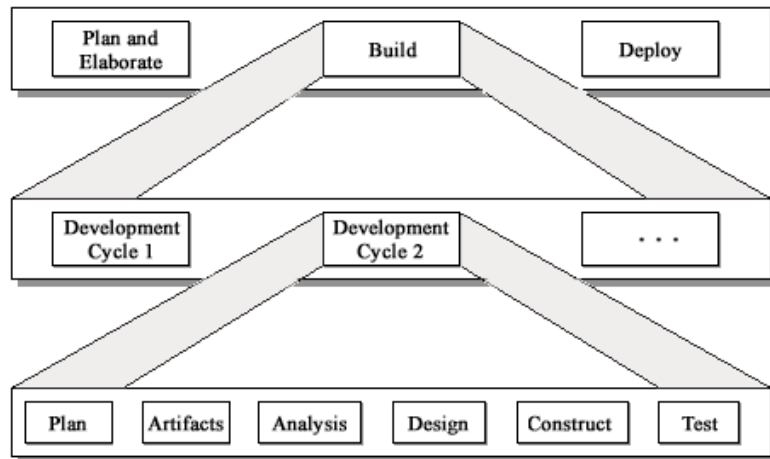


Figure 2.10: Iterative development cycles (reproduced from [Larman 1998])

There are some other views else to the software development cycle. Although all of them are different in appearance, there are still many common points.

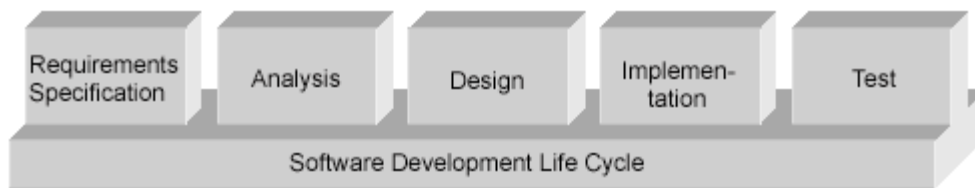


Figure 2.11: Software development life cycle

By our understanding, the software development life cycle consists of five phases. Fig 2.11 shows them as separate “components” connected by a “plate”, which implies that although the development generally follows the direction from the left to the right, each phase may associate with any others. That says the development cycle can be iterative. Each phases involves some specific activities.

1. Requirements specification

Requirements explicitly define what the software is supposed to do. They are included into two groups, the functional requirements and the non-functional requirements. The formal strictly specify the functions the software should at least provide. The later specify the request for the software's performance. Both can be looked as the objectives of the development cycle and serves as the basis of the analysis, design and implementation.

2. Analysis

In one word, analysis is to answer the question, "what is the problem". That means the objective of analysis phase is trying to understand the requirements. In object-oriented analysis (OOA), the activities involved in this phase includes:

- Defining use cases. This is to convert the requirements into a set of concrete use cases. This helps to figure out how the potential user will use the software and what functions (at the programming level) the software should provide. User cases can either be written to a document in some specific formats, or be represented using a UML user case diagram, or both.
- Defining conceptual model with class diagrams. OO takes each entity appeared in the requirements and user cases as an object. Finding out all these objects and finding out the relationships among them is the most important step in OO analysis. The result is the so-called conceptual model. In this model, each object or relationship is identified with a name that is the best match to its real life role. Each object can be further specified with its state (attributes) and behaviors (functions). This job is normally done with a tool supporting UML class diagram.
- Defining collaboration diagrams. In user case stage, the requirements have been converted into a serial of fine use cases. How will they be satisfied by the software? As the conceptual model has given out all possible objects, the question is then to be how these objects will collaborate to satisfy users' requests in each use case. The logical relationships will be worked out in the form of messages flow between objects. This job can be done with a tool supporting UML collaboration diagram.

3. Design

While analysis emphasizes what is the problem, design pays more attentions to how the problem will be solved. Software can be created by different designs. A good design plus good implementation will bring out good software. A poor design will absolutely bring out software in poor quality. Thus design is crucial in the software development life cycle. By using object-oriented design (OOD), the main activities in this phase are:

- Defining software architecture. Software can normally divided into certain functional units. Each functional unit again consists of some fine units. And there are some units that are hidden from the users (don't provide directly service to the users) but useful to enable those functional units to make sense. Thus questions like how to arrange them into a specific structure to achieve best performance and how to connect them together into a whole appear and the answers to these questions can be obtained from the architecture design. The best way to present software architecture is to use diagrams. But the diagram varies in types. UML class diagrams, package diagrams can be used to describe architectures. But in most cases the architectures are presented in some other suitable diagrams.
- Defining design class diagrams. The design diagram will be based on the architecture design and the results of analysis phase, mainly the collaboration diagram and the analysis class diagram. As the purpose of the analysis class diagram is to visually present what the problem really looks like, some objects that are definitely not a part of the software, such as the user of the software may be included into the class diagram, some objects appear in analysis class diagram with the form which is not suitable to be implemented thus need to be refined, some new functions or new attributes resulted from the collaboration diagram need to be added, and some old functions need to be splitted to make sure they are feasible. All these problems will be resolved in this phase. The resulted diagrams will contain all objects to be implemented and all attributes, methods that are necessary (not those helper methods that makes coding easier). This job will be done using a tool which supports UML diagram.

4. Implementation

The design class diagrams serve as the “blueprints” of the implementation. In this phase, programmers convert all designs into actual programming code. Theoretically any text editor can be used to perform this task. But in fact, a syntax-sensitive textual programming tool is really needed because it will help programmers in many ways (we cover this later). Moreover, the implementation is an iterative process, which means the implementation cannot be done all in one time. It needs a test tool to test the code at any stage of the implementation phase. Therefore an Iterative Developing Environment (IDE) is required in this phase.

5. Testing

In fact, testing is always being performed during the implementation stage because programmers should always test the code to check if the specific function has been properly implemented when they think the implementation to this function is finished. There are many testing methods. Two popularly used methods are Black-Box Testing and White-Box Testing. The former focuses on testing functional requirements. The later focus on using the internal knowledge of the software to design the test data and analysis the test results. Both can be done inside an IDE.

2.3.2 The Unified Modeling Language

In the real world, a system is usually very complicated. Even described in a well-presented document, it is hard to be understood even by experienced programmers. Starting programming directly on the basis of such a document seems impossible. So it is very important to find a way to accurately represent a system so that programmers or even machines can understand it quickly. Using a model to describe and abstract the essential aspects of the system is such a way.

To model a real car, we may use terms such as “maker”, “model”, “manufacture date”, “capacity” etc. To model a system, we may also need certain terms. Such terms are defined in a metamodel. Metamodel, by its definition, is a model that defines other models. So, is there any metamodel for modeling a real world system? Moreover, the model that represents the real world system must be presented in a way that its potential users

can see it and then can understand and use it. So how will the model be presented? In a text file, in a XML file or in some kinds of diagrams?

The Unified Modeling Language (UML) gives the best answers. UML is the leading modeling language for object-oriented analysis and design. Although it was the work of Grady Booch, Ivar Jacobson and James Rumbaugh, it was taken over by the OMG and became a standard in 1997. “Any software architect today who is designing large, complex systems must use a modeling language like UML” [Hamilton 1999]. Two major contributions of the UML are:

- Defining a common object analysis and design metamodel. The metamodel formally and accurately defines what objects and what relationships between objects can exist in an object model. For example, the metamodel defines Class, Method, Attribute etc can be existed in the UML model, therefore, we may use Class, Method, Attribute etc to abstract our real world system and change it to a UML model.

As now UML is an industry standard for describing objects model and it supports use case diagram, class diagram, object diagram, sequence diagram, collaboration diagram, statechart diagram, activity diagram, component diagram and deployment diagram, the metamodel is very large. It contains over 130 classes that grouped into 11 packages, each with descriptions, constrains and semantics fully defined.

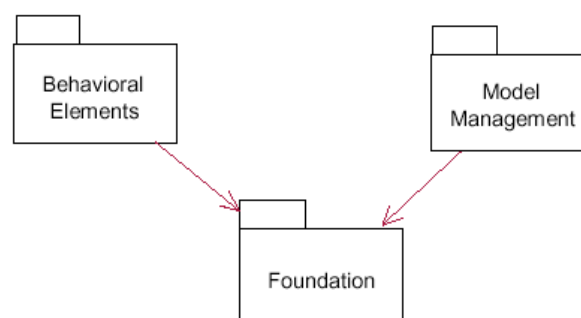


Fig 2.12: The top-level packages of the UML metamodel for class diagram

The UML metamodel itself can be presented by UML diagrams. As an example, Fig 2.12 shows the

Top-Level Packages of the UML metamodel for class diagram that is defined in a UML package diagram. There are some packages under those top-level packages. All low level objects are defined in UML diagrams.

- Defining the corresponding notations. To visually display the UML model objects and consistently express the UML's semantics, UML defines a graphic syntax that supports all UML diagrams. Each diagram can be looked as a view of the UML model. Some typical notations for class diagram are showing in Fig 2.13.

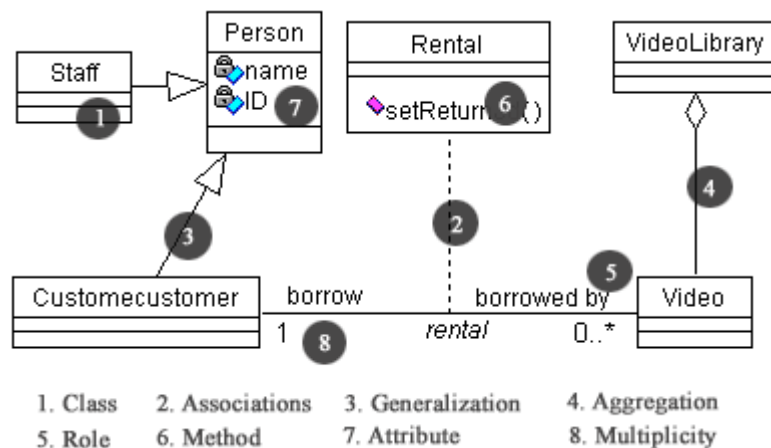


Figure 2.13: Some examples of UML notations

A model is a representation of your problem domain and software system. A model contains classes, logical packages, objects, operations, component packages, modules, processors, devices, and the relationships between them. Each of these model components possesses properties that identify, and characterize them.

A model also contains diagrams and specifications, which provide a means of visualizing and manipulating the model's components and their properties. Since diagrams are used to illustrate multiple views of a model, icons representing a model component can appear in none, one, or several of a model's diagrams. Rose therefore enables you to control which components and properties appear on each diagram.

2.3.3 CASE tool and MetaCASE tool

CASE stands for Computer-Aided Software Engineering. CASE tool has no clear definition. By our understanding, CASE tool contains at least two meanings: 1) it is a piece of software, which can be used as a tool in certain software engineering activities; 2) when being used, it can help its user in some way and to some extent.

By this definition, only piece of software that supports any software engineering activities can be looked as a CASE tool. Thus modeling tools, IDEs, code generators, compilers, debuggers, code editors, testing tools etc are all CASE tools [Gray et al 2000].

CASE tools can be a single tool providing single function, like a code editor, or a compound of some sub-tools to provide a serial of functions. The later is normally called software engineering environments [Mynatt 1990]. DISSAG is CASE tool, and also a software development environment.

CASE tool itself is a piece of software. Therefore there must be a corresponding software development cycle to build it. Some CASE tools are similar in certain functions and it is possible to find a common way to build them. That is using a metaCASE tool.

Simply, metaCASE tool is nothing but a tool, which can be used to build other tools. There are two ways for metaCASE tools to builder other tools [Gray et al 2000]:

- Provide a well defined metamodel, and provide a good way to set up the model, and then generate code from the model. Using this way, a framework for the new tool normally can be obtained then the tool builder can extend the framework and get the whole solution. JComposer [Grundy *et al* 2000] [Grundy *et al* 1998a] provides a good example for this kind of metaCASE tools. The framework of our modeling tool (the Diagram Editor in DISSG) is actually generated by JComposer.
- Develop some reusable components, make them robust and reliable in the role they act and then assemble them together properly to achieve some new functions. This method helps tool builder in

such a way that directly provide some well-implemented functional components. JComposer [Grundy *et al* 2000] [Grundy *et al* 1998a] again provides a good example for this kind of metaCASE tools. JComposer itself is a component-based software. Some components, like JViews will be directly integrated into the new tool

Most of the existed metaCASE tools are used to build modeling tools. MetaEdit+ [metaEdit+], and metaMoose [Ferguson *et al* 2000] are two of good examples.

2.4 Related Tools

DISSAG is a development environment specially designed for DIS. Strictly say, there is no similar tool exist at the moment. But there are lots of related tools. In this section, we will introduce four tools. Rational Rose is chosen for it is not only a famous modeling tool but also now a whole development environment (It is much like DISSAG in this sense). Argo/UML is chosen as it provides a very good direction for building “clever” modeling tools. JComposer is chosen as it has showed us how much a code generator can do with a well designed metamodel, a good model specification way and a fine code generation logic. JBuilder is chosen as a good example of powerful IDEs.

2.4.1 Rational Rose

In previous sections, we have mentioned that the UML was originally works of Grady Booch, Ivar Jacobson and James Rumbaugh who worked for Rational Software. So it is reasonable to assume that Rational Software have a tool to support UML diagrams. This assumption is absolutely right. The Rational Software has a powerful modeling tool named Rational Rose to image, which supports all UML diagrams and is so far one of the best tools for object-oriented analysis, design and implementation. Fig 2.14 shows its main interface.

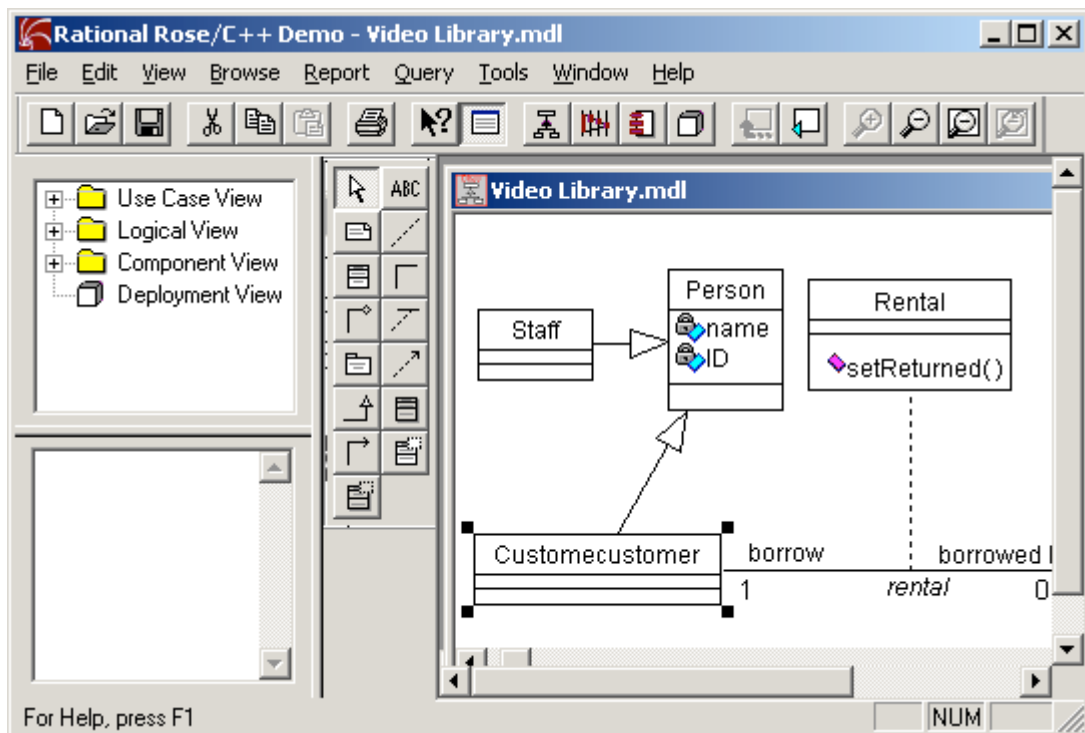


Figure 2.14: Rational Rose interface

When we say that it is one of the best tools for analysis, design and implementation, we actually point out the fact that Rational Rose is not only a modeling tool, but also an IDE. Some of its exciting features are:

- Supporting all kinds of UML diagrams. Therefore it enables users to set up conceptual models in the analysis stage and design models in design stage.
- Supporting specifying components in detail. It provides two specification schemes. One for specifying features that will visually appear in the diagram, such as multiplicity. One for specifying underlying model for the purpose of generating code (See Fig 2.15).
- Supporting code generation. It provides two ways to generate code, manually or automatically. In manual mood, users can choose any time to generate code for the current model. In automatic mood, any change to the diagram will be synchronously reflected in the generated code.
- Supporting controllable synchronization. That means any changes to the code will be reflected in the diagram if necessary. Therefore the developers can work on both diagram and code.
- Supporting reverse engineering. This makes it possible to integrate the existed components or systems. That means if the user adds an existed components or existed systems to the current model,

the change can be automatically reflected in the diagram.

- Supporting editing code. As pointed out above, users can work with the generated code and any change will be reflected in the diagram.

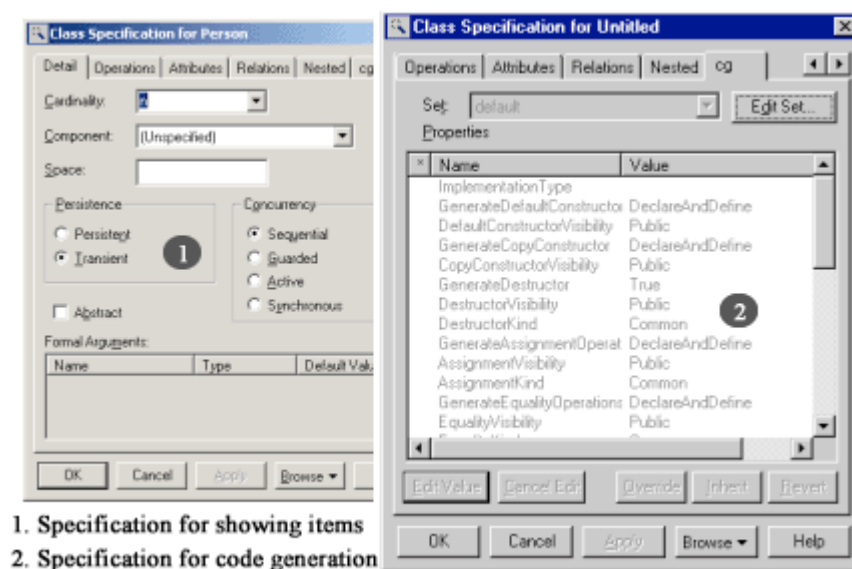


Figure 2.15: Rational Rose class specification windows

- Supporting system testing. As Rational Rose now has functions of IDE, it supports iterative development. That means users may test the system implementation at any time, to debug code, to find functional errors etc.
- Supporting component quality prediction. A large system normally divided into certain numbers of components to develop individually. The quality of the components is difficult to be checked before they are integrated into the whole system. It would be too late if problem occurs in the final test. Rational Rose provides means to test the components before they are integrated into the large system.

But Rational Rose is not flawless. Some demerits we think are:

- Code generation. Rational Rose code generation strongly rely on the modeling. The more you model, the more code you will get. It is good at accurately transferring the model information into actual

code. In this sense, it more or less likes a “model-code translator”. But it really does little to help users to set up a framework or architecture for a certain system. This is due to it is designed to work for a general domain not for any specific one. This gives us chance to design code generators for specific domain. If there are stable solutions for implementing systems in this domain, the code generate can automatically generate the framework of these specific solutions. Then users need only to specify the system in a relative small scope. It will hide most code details from the users, thus enable them work in a high abstract level. In this sense, the code generator would be very smart and would dramatically reduce the workload of the developers in this domain. This is one of our motivations to develop DISSAG, which is supposed to be such a code generator in DIS domain.

- Rational Rose generates code directly from its underlying model. It is good as it is very efficient without any intermediate. But it is bad in that the Rose model information cannot be used by the third party tools. More clearly, if somebody, like us, don't like its code generator and so want to develop our own code generator with its own generation logic, it would be hard for the new code generator to access Rose model and get information from it. So it would be desirable if the Rose model can export its information in a format that the other code generator can take. Our tool, DISSAG will be capable of export all information in XML format from its underlying model, thus any other code generator can use the information properly.

2.4.2 Argo/UML

One advantage of the UML diagrams is that they can visually represent real world systems and thus allow system developers to view these systems from different points and in different abstraction levels. These diagrams can help them to understand the systems and provide means for them to clearly represent their design ideas. If the function of a modeling tool were limited in drawing UML diagrams only, the modeling tool would be useless. Because using a pencil and a paper to draw those diagrams would be more efficient and more convenient although the quality of the diagrams would be worse. So what are the advantages of modeling tools over the traditional “pencil plus paper” tool?

There are at least two advantages. Firstly, a good modeling tool will help you to model a system. It will not only provide a tool to draw diagrams but also help you “thinking”, in other words, it will be a smart agent to guide you through the modeling process. Secondly, a good modeling tool will help you to properly use the result of the modeling, in other words, to convert it into implementation source code.

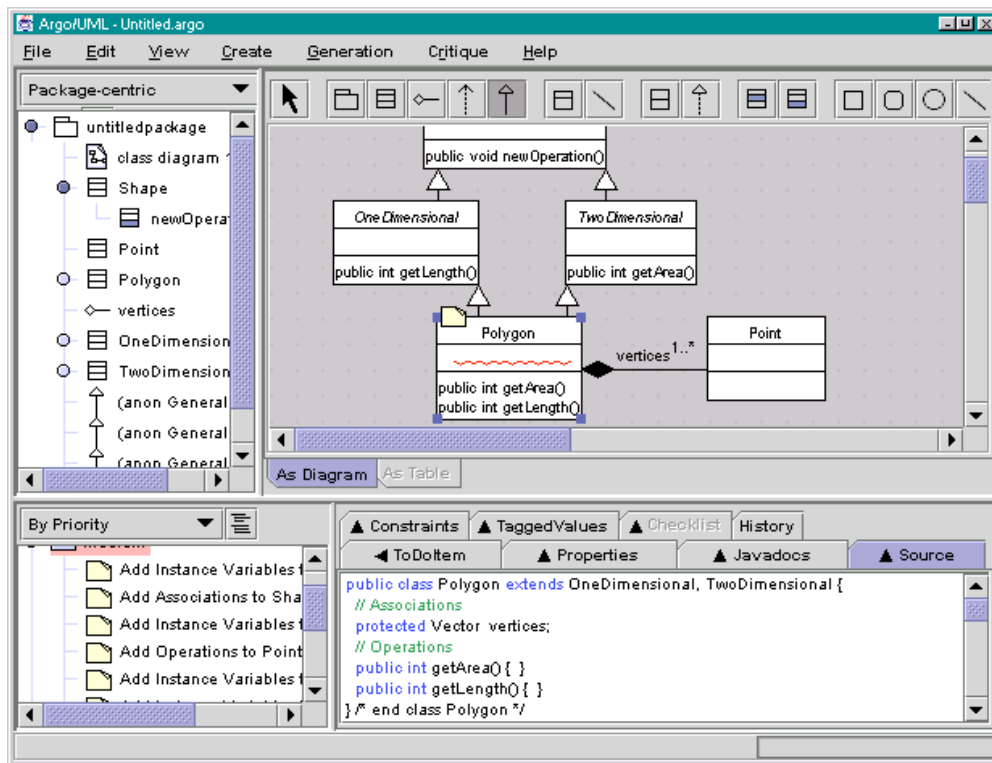


Figure 2.16: Argo/UML interface

Argo/UML is such a modeling tool that has at least above two advantages. It can generate source code from a UML model. But the most important feature that actually makes it different from the other modeling tools is that it provides multiple facilities to help its users during the modeling process. Fig 2.16 shows its main interface.

Argo/UML has the following features:

- Providing multiple design critics that continuously check the design for potential errors, stylistic violations, and incomplete sections. Tasks are performed automatically by critics in separate threads

so that users do not need to do anything to make them work. Each critic performs its own task independently and generates a feedback to the user providing some possible options for users to handle its specific information (see Fig 2.17 (2)). Users therefore can properly finish their modeling tasks. Argo/UML also has a Criticism control mechanisms that can select proper critics and even configure them.

- Dynamically generating “To do” list. All feedbacks from the critics are included into a “To do” list and grouped by category. So at any time, the users are aware of what tasks are waiting them to do. Argo/UML even allow users to add a new item to the list as s personal reminder (See Fig 2.17 (3)).
- Generating checklist to remind designers of the common problems that may be exist in the modeling.
- Providing wizards that lead users to solve some particular problems with the common solutions.
- Supporting UML class diagrams, state diagrams, use case diagrams, activity diagrams and collaboration diagrams, and supporting generating code from UML class diagram.

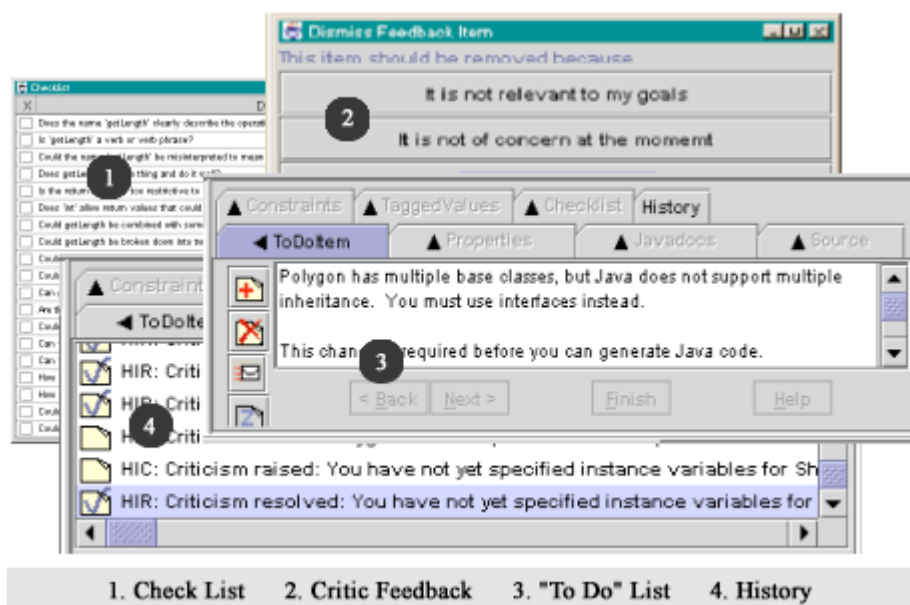


Figure 2.17: Some features of Argo/UML

Generally say, Argo/UML is good modeling tool. That means it can be used only in OOA&D phases. Another tool like an IDE is needed to continue its job to finish the whole development life cycle. Although it is capable of generating java code from class diagram, it is not very good at this. The code it generates is a just a skeleton which is the direct reflect of the model information.

2.4.3 JComposer

Tools that can be used to model a system should at least satisfy two basic requirements, one is that they must have a metamodel defining the model they will use to model the real world systems, the other one is that they must have defined the corresponding notations for the metamodel and have actually implemented them into visual components that can be used in diagrams.

Most modeling tools now are using UML metamodel as their own metamodel. And the visual notations they use now are very alike. JComposer is a special one that has its own metamodel and notations. Fig 2.18 shows the interfaces of JComposer.

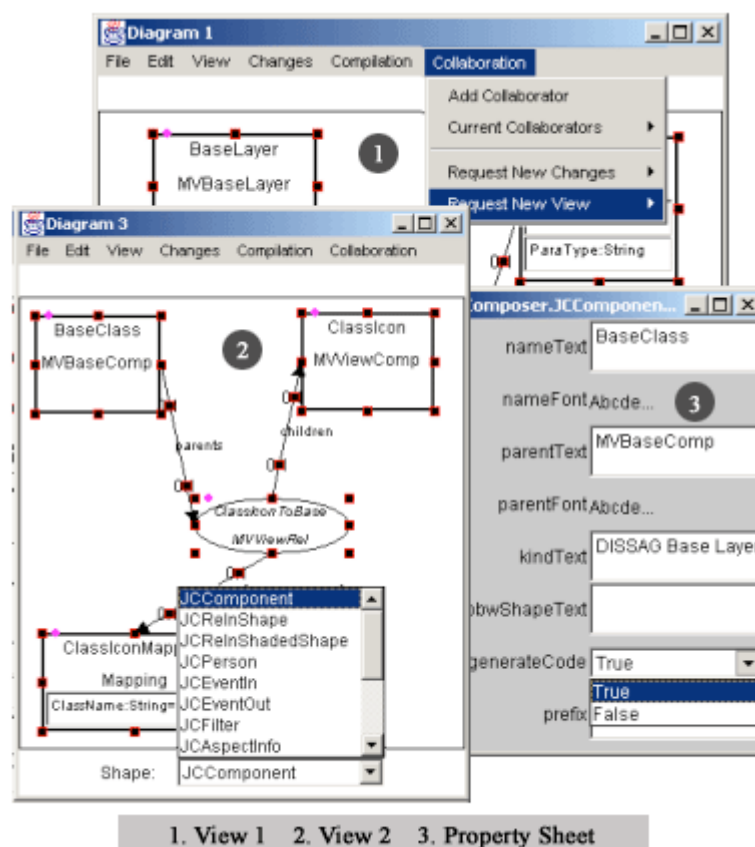


Figure 2.18: JComposer interface

JComposer is a metaCASE tool that can be used to build JViews-based modeling tools. JComposer itself is a seamless integration of a code generator and a modeling tool, which is also JViews-based.

JViews is a component-based software and Java class framework, which is especially good at setting up the architecture for a modeling tool and bringing it some excellent features. It supports 3-layer (representation, view, and repository) abstraction and allows tool-builders to define their own base layer, view layer and representation layer components. It also supports multiple views and persistency management for base layer or view layer components. It even supports collaboration.

As a metaCASE tool, JComposer can be used for tool-builders to define their own metamodel in a visual environment. The definition process is actually a modeling process. As we have pointed out above, the metamodel and notations the JComposer use are different from that of UML. And JComposer provides a way for tool-builders to notate the diagram components, which in fact is way to specify its underlying model.

JComposer's sister tool—Builder By Wire (BBW), a metaCASE also, can be used to define the visual notations for the metamodel in another visual environment. The code generator of JComposer can generate the whole solution in java source code.

JComposer now is still more like a prototype. Its interface and some details have not yet been implemented very well. But its performance is excellent. What are better are its underlying ideas for building a good CASE tool. We have not only used JComposer to generate the very original source code for our modeling tool, but also adopted some design ideas from JComposer such as defining our own metamodel and our own notations.

2.4.4 JBuilder

JBuilder [JBuilder 2000] is a PC-based iterative development environment (IDE). Like any other good IDEs, It provides a full range of code editing, source code debugging and test tools and thus enables its users to freely move through the edit-compile-debug cycle. Fig 2.19 shows its main interface.

JBuilder has many attractive features. Three of them should we pay more attention to. The first one is its well-designed user interface. It provides us a good example to construct DISSAG user interface. The second

one is its GUI composer for creating user interface. “What you see is what you get” is a good idea that we can adopt to improve DISSAG. The third one is its context sensitive code editor. When a piece of code has to be write by hands then such a code editor is very much helpful.

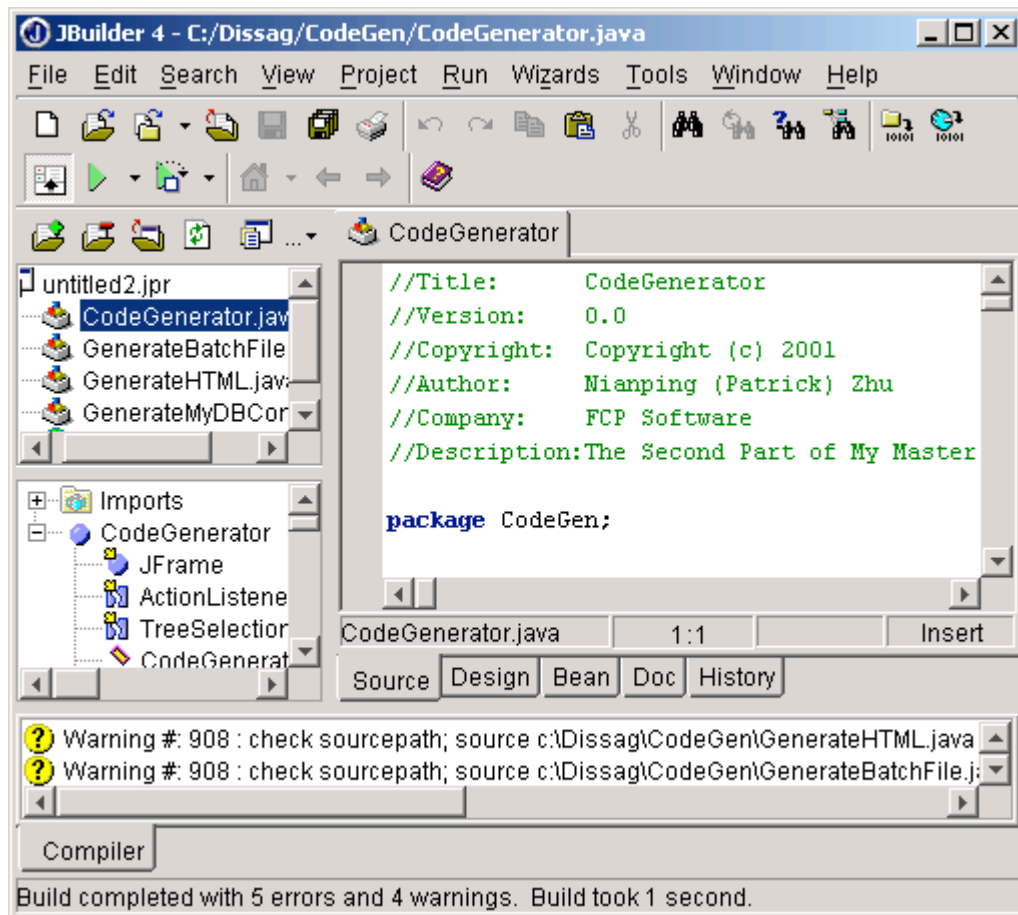


Figure 2.19: JBuilder interface

But like all other IDEs, JBuilder has no means to modeling a system, thus cannot be used to perform any OO analysis and design. It has not any underlying UML or UML-like model, its ability to obtain system information is much limited. Therefore it cannot generate source code based on system specification. But as we have mentioned above, it can generate source code for GUI. But the generate GUI code contains no underlying logic behind those visual elements. DISSAG can provide some key functions that JBuilder provides, such as text editing and code testing. And DISSAG’s main function will be modeling DIS and generating code, both are what the JBuilder lack.

2.5 Summary

DIS is an information system that consists of relatively independent subsystems, which are connected by certain networks. DIS is used to generate, manipulate and communicate information. As DIS involves storing data to and retrieving it from relational databases, JDBC is widely used when DIS is implemented in java. Most of DIS implementation has a 2-tier or 3-tier architecture. Middleware is added to handle communications between different tiers. CORBA, RMI, and DCOM are three of the most popular middleware. Each of them has stable coding pattern. The actual implementation code therefore is possible to be generated. This is why DISSAG as a code generator for DIS is feasible.

Software development life cycle can be divided into five phases, requirements, OOA, OOD, implementation, and testing. There are many kinds of CASE tools can be used in this cycle. Modeling tools and IDEs are the most important tools. Nowadays, more and more modeling tools and IDEs provide code generation function. Most modeling tools using UML metamodel to model systems and generate source code from the model. Thus the code is just the direct translation of model information. Most IDEs have a GUI composer thus can generate code for GUI. Some IDEs provide templates or provide wizard to generate solution for some particular domain. But the ability of code generation is limited. DISSAG will provide a new way to generate code to efficiently model DIS and generate code.

Chapter 3

DISSAG: requirements and specifications

3.1 Introduction

This chapter will focus on the requirements and specifications for DISSAG. At first, an overview of DISSAG will be given. Then, a detailed description of the functional and non-functional requirements will be presented. Finally, we shall provide and explain some object-oriented analysis class diagrams for DISSAG and its sub-tools.

3.2 DISSAG Overview

As a complete exploratory prototype, DISSAG can provide a visual programming environment in which the users (DIS software developers) can finish almost all developing tasks. It provides a handy, flexible user interface to enable the users to model the distributed information systems; It automatically perform the OO design task based on the input information; It automatically generate detailed source code; It also provides a text editor allowing users to view the generated code and do the further implementation; It even provides a test bed allowing the users to compile and run the generated and further implemented code. It is easy to use, easy to integrate third party tools and can be extended by users to add more user-desired functions.

Fig 3.1 shows the whole DIS software development life cycle using DISSAG. From which we can see the whole cycle started from the DIS case description and finished with the tested source code. We can also see how DISSAG is used and what activities are involved.

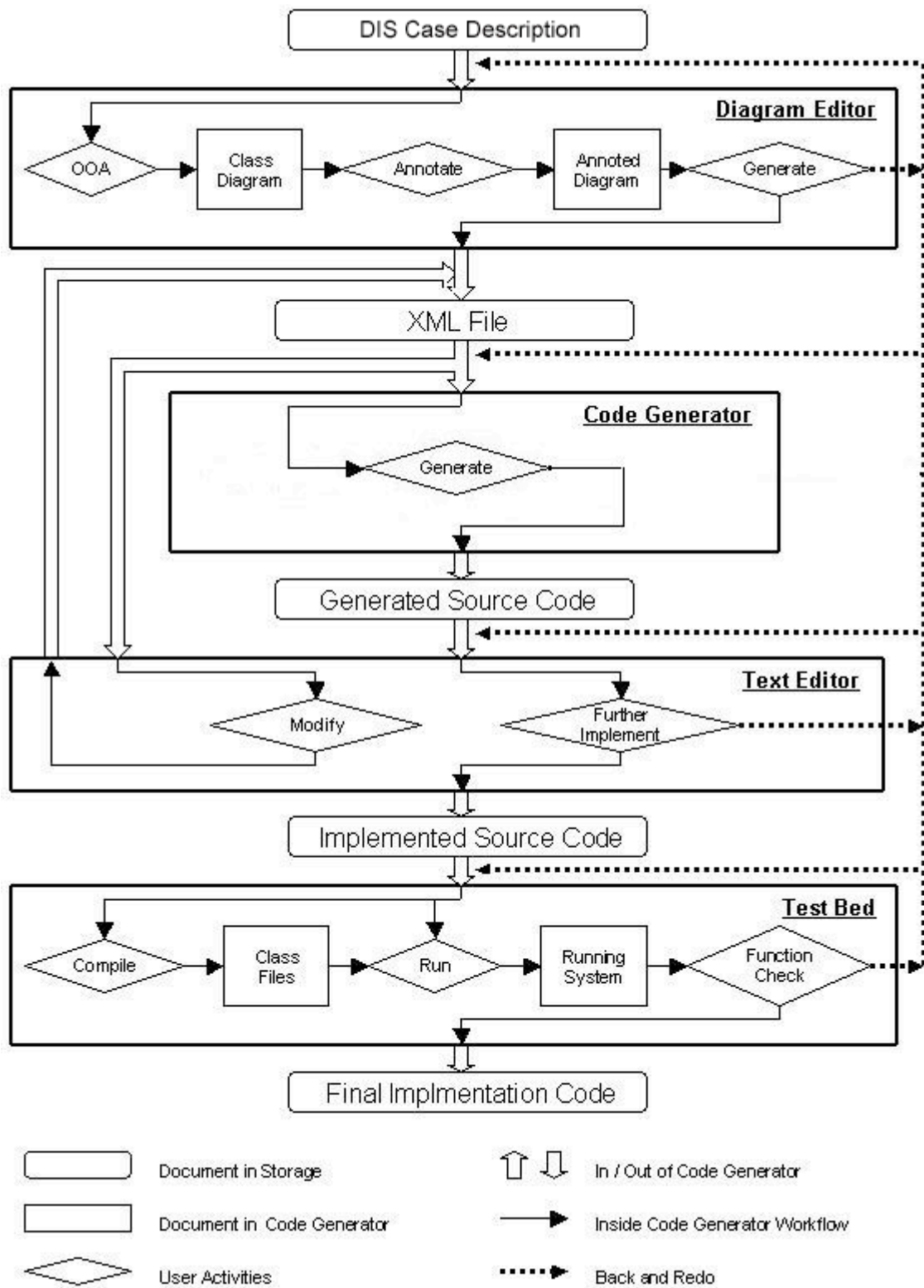


Figure 3.1: DIS Software development using DISSAG

DISSAG consists of four tools:

- **Diagram Editor:** a specification tool that can draw OOA class diagram, annotate the components in the diagram with information about DIS characteristics and generate XML files. This is the modeling tool of DISSAG and is used as the input device for DISSAG Code Generator.
- **Code generator:** a tool that takes XML files generated by diagram editor as input and generates detailed source java code for DIS. It is the core part of DISSAG.
- **Text Editor:** a normal text editor with a tree panel that organizes the generated code. It is a tool with which to view generated classes and do further implementation. It is an output device for DISSAG.
- **Test bed:** a handy mini tool to compile and run the generated and then further implemented code.

3.3 Functional Requirements

3.3.1 DISSAG

1. *Capable of providing an environment in which the users can perform most of possible tasks related the development of distributed information system software.* That means in this single environment, users can finish analysis, design, implementation and even testing of software applied to the DIS.
2. *Capable of drawing OOA class diagram.* OOA class diagram is one of the powerful means to do OO analysis. DISSAG is to provide users the function to analyze a DIS, so it must be able to let users draw OOA classes. In other words, a diagram editor is required.
3. *Each of the OOA class diagram icons can be annotated.* By finishing OOA class diagram, the user has input the most important information into DISSAG. But there are still some more information needed for the purpose of generating much detailed DIS design and code. For example, if there is attribute of “name” in a certain class, in the diagram, the user has to not only name a attribute icon as “name”, but also to annotate it by filling in some entries in its property dialog, such as “Type = String” and “Initial Value = none”. Each kind of icons has a corresponding property sheet that can be

modified at any time. By filling some or all of the entries of the property sheet, the user can annotate the diagram.

4. *Capable of generating DIS design and implementation code from annotated OOA class diagram.* After annotate the OOA diagram, users have already input the required information into DISSAG. That means all characters of the DIS have been obtained. DISAG thus can generate code based on them. In another view, DISSAG will enable the users to jump over design phase. That means DISSAG will automatically split OOA classes into OOD classes based on all information obtained from the OOA diagram and its annotations.
5. *Capable of generating much detailed source code.* As discussed in Chapter 3, software applied to DIS has relative stable architectures and has well developed middleware technologies to support its implementation. This provides the feasibility of generating much detailed implementation source code. For example, if the user choose CORBA as the middleware to implement a DIS, and given that the system has been well specified (OOA class diagram has been finished and annotated), DISSAG should be able to generate IDL files, implement classes, remote manager classes and user interface classes and most of them will be ready to be compiled and run.
6. *The generated code must be readable, understandable and user extensible.* Although generating ready-to-run code with full user expected functions is the ultimate target, it is very hard to actually achieve this. As a result, users have to take part in the further implementation. This requires the generated code must be easy for users to read and understand and must be able to be extended by users.
7. *Support textual programming.* As talked above, the generated code has to be further implemented by users. As a developing environment, DISSAG should provide means to allow users to perform their further implementing task. Thus a text editor is needed to be integrated into DISSAG, which can view and edit the generated code.

8. *Capable of testing generated code before or after further implementation.* It is very important for users to test the code to find out what requirements have been satisfied or not. It will guide users to find out where to be further implemented and even how to do that. To test here means to compile and to run.
9. *Should appear as a single application.* That means although DISSAG is in fact an assembly of four associated tools, it appears as a single tool. All sub-tools are sharing an interface, and are integrated into a single application.
10. *Capable of switching to and working with each sub-tool at any time and switching as often as needed.* For example, it should be easy for user to switch to diagram editor and begin to work with an OOA class diagram while he is working at further implementation phase using text editor.

3.3.2 Diagram Editor

As a specification tool of DISSAG, diagram editor must be capable of drawing OOA class diagram and the diagram must be able to be annotated. That has been looked as a very important requirement of DISSAG and has been included in 3.3.1. In this section, more requirements to the diagram editor will be presented:

1. *Can be used both as a normal class diagram editor and as a special input device for DIS code generate.* That means it can be used to draw class diagrams and help users to perform normal analysis tasks or design tasks. But it will be added more special functions so that the diagram can be annotated thus it can be used as an input device of DISSAG.
2. *Capable of modeling multiply projects simultaneously and switching to any project at any time.*
3. *Supporting multi-views and capable of keeping views consistent.* Multi-views here means users may open two or more diagram windows to model the same project. Each window, or a view here,

represents a part of the diagram. This is useful when the project is large and there are lots of classes. Because in this case, even using a scrollable canvas to draw diagram, it is still not convenient to draw all classes, their relationships and their sub-components together in a single canvas. This is also useful to keep the class diagram neat. For example, users can use one or more views to represent inter-classes relationships and use one view to represent intra-class relationships for each class. So Multi-views is an important requirement for diagram editor. This leads to another requirements, keeping views consistent. As each view represents a part of the diagram, it is quite normal that a component appear in this view will appear there. Thus effort must be made to keep them consistent. That means any modification made to the component in a view must be automatically and instantly reflected by the same component in the other views.

4. *Capable of switching to any view at anytime.* As users may work on different parts of a diagram or different diagrams, there is no wonder for this requirement.
5. *Having separate icons for class, attribute, constructor, method and parameter and for each kind of inter-class relationships.* Most of UML tools have only icons for classes and their relationships. They take attributes, constructors, methods and parameters as properties of class components. They have a shortcoming: it is hard to specify attributes, methods and parameters in detail. Taking attribute, constructor, method and parameter as separate components will make things easier to specify system more efficient.
6. *Each icon has a name that will be showed inside icon.* The different shapes of icons makes a kind of objects different from the others, such as a class icon is obviously different from an attribute icon. But to distinguish a class component from the other class components, it is necessary to name each icon and let the icon name appear in the icon shape.
7. *Each icon can be annotated. See 3.3.1.*

8. *Each icon can be moved and resized individually or as a group.* Move and Resize are two important tool in diagram editor. By using them, the users can arrange icons or adjust icon sizes so as to achieve a better visual effect.
9. *Each icon can be selected and deleted.* The users are likely to add an icon by mistake or to change their mind at any stage so want to remove a previously added icon, then select and delete tools will be useful.
10. *Capable of viewing components and its relationships as a component tree at any time.* As there are multi-views for a class diagram, sometimes when the number of classes getting big, it is hard for users to grasp how many component icons they have added to the diagram and what are the relationships among them. So it would be much helpful if there were a means to enable users to view the whole project at any time with an efficient way. This requirement is good compensation to the weakness of multi-views.
11. *Capable of generating XML files that contain all specified system information from the class diagram.* The purpose of drawing OOA classes and annotating the diagram is inputting all known information into DISSAG so that the code generator tool in DISSAG can read the information and generate source code based on it But for some reasons (See 3.4 and 4.2), the diagram editor tool is totally separated from the code generator tool. So there must be an intermediate between them to pass the information from diagram editor to code generator. XML is the best option to serve as this role because XML is able to code rich information into text-flavor file and extract the information back by the means of parsing XML. So generating XML files means to collect all information, which is input by the users from drawing OOA diagram and annotating the diagram, and code it into XML file which can be easily read by code generator.
12. *Capable of saving and loading OOA class diagram and its underlying OOA model.* There are lots of reasons to save and load an OOA class diagram and its underlying OOA model. It is impossible to

finish all modeling in a short time so several middle versions needed to be saved; System may be crashed unexpectedly so it is wise to save the modeling periodically; A new modeling may be based on an previous one; a finished model may need to be updated etc.

3.3.3 Code Generator

Code generator is the core part of DISSAG. As talked in 3.3.1, code generator must be capable of generating much detailed and user extensible source code from annotated OOA class diagram. Here are some more requirements for code generator:

1. *Capable of accessing XML files.* The input to the code generator is XML file, which is generated by diagram editor and contains all specified system information on which the code generation logic will be based. To extract the information, code generator must be able to access XML files.
2. *Capable of splitting OOA classes to OOD classes according to the specification, generating code using CORBA, RMI and JDBC as middleware and generating code for user interface (See 3.3.1).*
3. *Capable of generating html file to nest applet that is a kind of user interface.* If the generated user interface is a java applet, it is necessary to generate a html file in order to test the applet.
4. *Capable of generating deploying files that can be run as processes to compile source code into class files or to execute the compiled class files.* In order to check the generated code, it is often needed to compile and run the generated source code again and again. To help users easily perform this task, the deploying files to compile java source code and to run the compiled class files should be generated as well.
5. *Capable of generating a tree view that tracks the generated files.* For each project, there would be a set of generated files. To help users grasp what files have been generated, there should be a function

to display all generated file names as a tree.

6. *The generated files can be viewed easily and immediately.* As talked above, there would be a tree to show all generated file names. The same tree would be able to serve as a index by clicking whose node a text editor window would be open and the file the node represents would be viewed in the window.

3.3.4 Text Editor

1. *Capable of viewing and editing all kinds of text files including java files, html files and XML files etc.*
2. *Capable of opening generated files by clicking the nodes of the tree that is generated by code generator (See 3.3.3).*
3. *Capable of creating a new file or opening an existed file.*
4. *There should be a separate editor window for each file.*
5. *Capable of working with multiple files simultaneously and switching to any file at any time.* As talked above, there would be a separate window for each file, thus there would be multiple windows for editing or viewing multiple files. And users can work on any file at any time by making the window, in which the file is viewed or edited, active.
6. *Showing names of all opened files.* For users to track all opened files, there should be a means to show the names of all opened files clearly.
7. *The opening editor windows can be closed separately or all together at any time.*

8. *The files being viewing and editing can be saved or “saved as” separately or together at any time.*
9. *Providing “copy”, “cut”, “paste” and “select all” function.* The text editor should be capable of perform these functions not only inside DISSAG but also with other applications, that means to copy to or paste from the system clipboard.
10. *Providing “find”, “go to” and “replace” function and these functions can be customized.* For example “find” has options “Forward from beginning”, “Backward from the end”, “Forward from current caret position” and “Backward from current caret position”. “Replace” has the same options as “find” but with one more option “Replace All”.
11. *Providing “undo”, “redo” function.* This enables the users to undo or redo a latest performed task.
12. *Dynamically show the caret position.* The line number will be showed somewhere whenever and wherever the users click on the editor window. This would be helpful for debugging in test phase.
13. *Warning when closing a changed file.* This will stop losing important unsaved contents from a careless action to shut an editor window.

3.3.5 Test Bed

Test bed is integrated into DISSAG so that the users can debug their further implemented code or run the code to check if any requirement has been satisfied (See 3.3.1).

1. *Capable of executing instructions that are saved in files or input directly.* Code generator has generated most of those files that contains instructions to compile or run the generated and further implemented code. Test bed should be able to execute those files or be able to accept instructions

input by users and then execute them.

2. *Capable of browsing for files to be executed.* Instead of typing in the full path of the saved files, there should be a easier way to browse and select those files.
3. *Capable of selecting command from command history.* For those commands that are typed in directly, there should be a command history to record them and there should be a way to select a command from the command history to speed the inputting and reducing errors.

3.4 Non-functional Requirements

Non-functional requirements here mean those requirements that do not relate directly to the functions or operations to be performed. Any important information with DISSAG that cannot conveniently be represented in the functional requirement part will appear in non-functional part. There could be desired features or characteristics, quality guidelines, and any restrictions that may apply to the system. For DISSAG, the non-functional requirements could include:

1. *The four sub-tools should be totally separated so that each of the four sub-tools can be maintained and extended separated.*
2. *Third party tools can be integrated to replace any of the four sub-tools of DISSAG.* For example, any other CASE tool can replace the diagram editor as long as the new tool is capable of accepting information and change it to XML format. Any similar tools can replace the text editor tool as long as that the new tool can be used to view or edit text files.
3. *Third party tools can be integrated to add functions to DISSAG.* For example, JDK 1.3 will be integrated into DISSAG so that the test bed can be used to compile and run java source code. Another example is that the Internet Explorer can be integrated into DISSAG so that the users can view help files or tutorial on line.

3.5 OOA Specification

3.5.1 DISSAG

Fig 3.2 shows the high-level OOA diagram for DISSAG. There are five objects identified as “DISSAG”, “DiagramEditor”, “CodeGenerator”, “TextEditor” and “TestBed”. “DISSAG” represents this distributed information system software development environment. The others represent the four sub-tools, Diagram Editor, Code Generator, Text Editor and Test Bed. Each of them is “a part” of DISSAG so that there is an “aggregation” relationship between “DISSAG” object and the object of each of these four sub-tools. Each DISSAG object at least contains “one” but only “one” object of each sub-tool. Each sub-tool object belongs to at least one and only one DISSAG object.

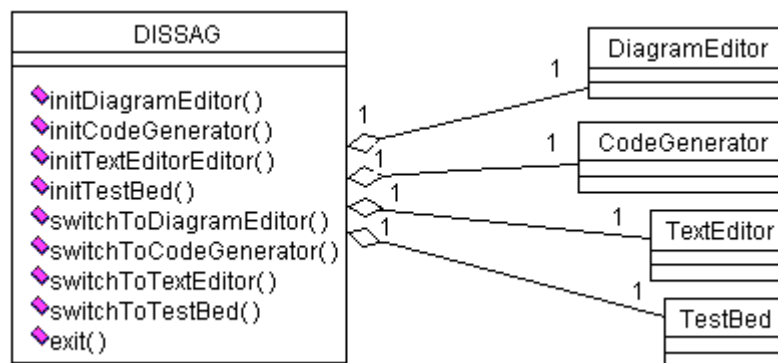


Figure 3.2: DISSAG OOA class diagram

To meet the requirements set up for DISSAG, the DISSAG object must at least has the following functions:

- `initDiagramEditor()`, `initCodeGenerator()`, `initTextEdiotr()` and `initTestBed()`: To init those four sub-tools separately when there is a need to use them.
- `switchToDiagramEditor()`, `switchToDiagramEditor()`, `switchTo- DiagramEditor()` and `switchToDiagramEditor()`: To enable DISSAG users to select a tool to use at anytime.
- `exit()`: To stop using DISSAG and all of the four sub-tools.

The details to the four sub-tool objects will be specified in the following sections.

3.5.2 Diagram Editor

Fig 3.3 shows the high-level OOA diagram for Diagram Editor. There are eight objects in this diagram.

“DISSAG” has been discussed in previous section.

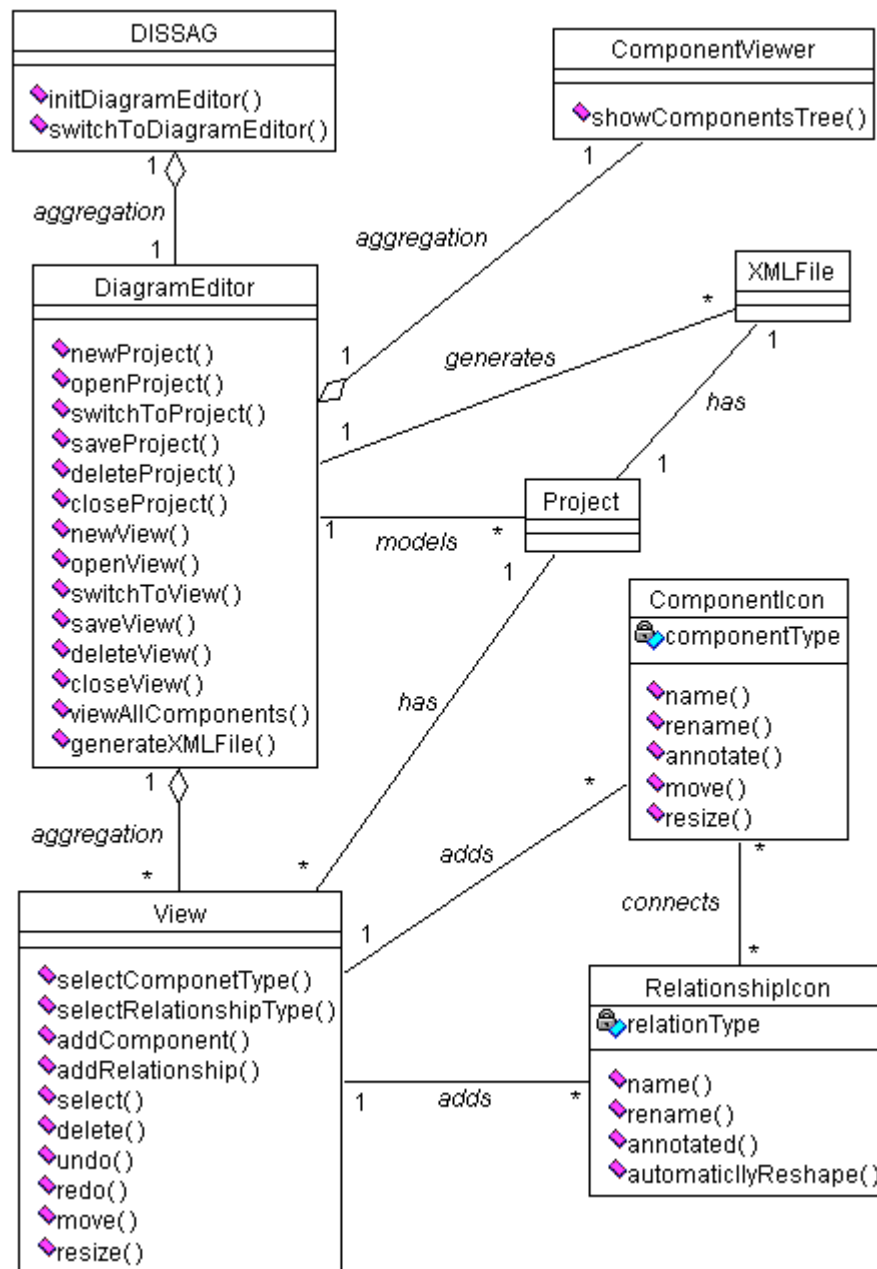


Figure 3.3: Diagram Editor OOA Class Diagram

“DiagramEditor” is the main component of “DISSAG”. It has two components: “View” and “ComponentViewer”. It can “models” multiple “project” simultaneously and can “generates” corresponding XML file for each “project”. It has the following grouped functions:

- *newProject(), openProject(), saveProject(), deleteProject(), closeProject() and switchToProject()*. All these functions are related with the operations to “Project” and enables users to create a new project, to open an existed project, to save a project at anytime, to delete a project at anytime, to close a project and, if there are more than one projects, to switch to a project to work with.
- *newView(), openView(), saveView(), deleteView(), closeView() and switchToView()*. All these functions are related with the operations to “View” and enables users to create a new view, to open an existed view, to save a view at anytime, to delete a view at anytime, to close a view and, if there are more than one views, to switch to a view to work with.
- *ViewAllComponents()*. This function enables users to view all information about a project by calling “ComponentViewer” to “show ComponentsTree”.
- *GenerateXMLFile()*. This is one of the most important functions. It changes all information specified by the user, through drawing OOA diagram and annotating the diagram, into a XML file that will be accepted by the successor, Code Generator.

“ComponentViewer” is a component of “DiagramEditor”. Its function, “showComponentsTree()”, enables users to view the information for a project, all components and the relationships among them. It will present all the information in a tree.

“View” is another component of the “Code Generator”. Each view provides a canvas and a set of tools to enable users to drawing OOA diagram by “adding” component icons and relationship icons. It has the following grouped functions:

- *selectComponentType(), addComponent()*. These functions provide users means to model project objects. *SelectComponentType()* enables users to select an icon type (shape) for the object to model. *addComponent()* enables users to add an icon of the selected type to the canvas.
- *selectRelationshipType(), addRelationship()*. These functions provide users means to model project

objects. *SelectRelationshipType()* enables users to select an icon type (shape) for the relationship. *addRelationship()* enables users to add an icon of the selected type to the canvas so as to connect two component icons.

- *select()*, *move()*, *resize()* and *delete()*. “*select()*” enables the user to select an added icon or select several added icons as a group. “*move()*”, “*resize()*” and “*delete()*” will provide the means to move, resize or delete the selected icon or icon group.
- *undo()* and *redo()*. These two functions will enable users to undo redo the last operation to this view.

“ComponentIcon” will be “added” to “View”. It has a attribute named “componentType” to specify the types of the actual icon. It has following functions:

- *name()* and *rename()*. These functions will enable users to name the icon or to modify the name.
- *annotate()*. This function will enable users to specify component features.
- *Move()* and *resize()*. This functions will change the position or size of the icon.

“RelationshipIcon” will be “added” to “View”. Like “ComponentIcon”, it has “name(), rename() and annotate() functions. But instead of having functions of “move” and “resize”, it has another function:

- *automaticallyReshape()*. This function will ensure that the relationship icon will automatically change its position and shape according to the changes made to the position and size of the two components connected.

“Project” object represents the project to be modeled by “Diagram Editor”. “XMLFile” object represents the XML files generated by Code Generator for each project modeled.

3.5.3 Code Generator

Fig 3.4 shows the high-level OOA diagram for Code Generator. There are seven objects in this diagram.

“DISSAG” has been discussed in 3.5.1, “XMLFiles” has been discussed in 3.5.2.

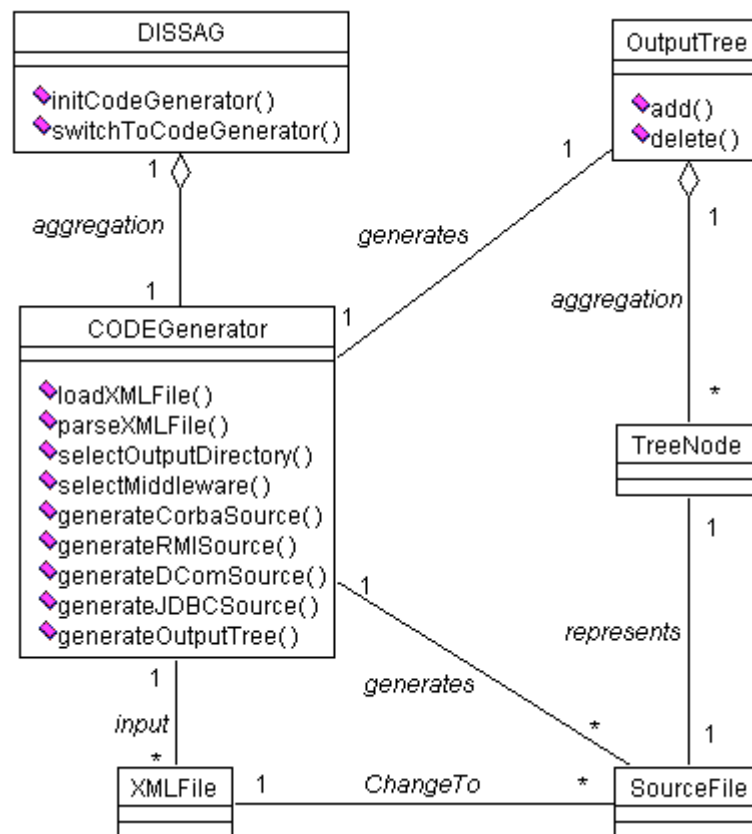


Figure 3.4: Code Generator OOA Class Diagram

“Code Generator” is the main component of DISSAG. It accepts “XMLFile” as “input”, “parses”, “generates” “SourceFile” as output and also “generates” “OutputTree” that can be used as an index by Code Generator’s successor Text Editor. It has the following functions:

- *loadXMLFile()*. To load a XML file to Code Generator.
- *selectOutputDirectory()*. To specify a directory to which the Code Generator will generate source code.
- *selectMiddleware()*. To choose a middleware for implementation.
- *generateCorbaSource()*, *generateRMISource()*, *generateDComSource()* and *generateJDBCSource()*. These functions are the main functions of DISSAG. They will be used to generate implementation source code for both client side and server side using one of the four middlewares.
- *generateOutputTree()*. Not only will Code Generator generate source code and output the code to the

specified directory, but also it will provide a convenient way for users to view and edit the generated code. That is generating an output tree, which presents all names of the generated files. This tree will be used as an index by the successor, Text Editor, to manage files to be edited.

“XMLFile” here is the input to Code Generator. It has also been discussed in 3.5.2.

“OutputTree” is a tree with the names of generated files as its nodes. It is generated by Code Generator along with generating source code. Each Code Generator has only one instance of “OutputTree”. It will be used by Text Editor to manage the viewing and further implementation of generated source code. It has two functions:

- *add()*. To add a new node to the tree. It was called in most cases by Code Generator itself.
- *delete()*. To delete a node from the output tree.

“TreeNode” is a part of “OutputTree”. Each “TreeNode” “represents” a generated “SourceFile” by taking the file name as its user object.

“SourceFile” here represents any generated files indexed by the “TreeNode” of the “OutputTree”. For each project, there will be a set of source files generated.

3.5.4 Text Editor

Fig 3.5 shows the high-level OOA diagram for Text Editor. There are six objects in this diagram. “DISSAG”, “OutputTree” and “TreeNode” have been discussed in previous sections.

“TextEditor” is a part of “DISSAG” and has “EditorView” as its component. It also “uses” “OutputTree” to manage the files in the editor views if the files are generated by Code Generator. It has the following functions:

- *open()* and *new()*. To open a existed file or create a new file.
- *close()*, *saveAll()* and *closeAll()*. To close a editor view, to save all files or to close all editor views.
- *SwitchToEditorView()*. To select an editor view as the current view .

“EditorView” is a part of the “TextEditor”. Each “EditorView” “contains” a “textFile” to be edited. It has the following functions:

- *Copy(), cut(), paste()* and *selectAll()*.
- *undo()* and *redo()*.
- *goto(), replace()* and *find()*.
- *ShowCaretPosition()*. To show the current caret position when the canvas of the view is clicked.
- *save()* and *saveAs()*.
- *CloseWarning()*. A warning will be given out before an opened view with modified file close.

“TextFile” represents the file to be edited. Each “TextFile” has a “EditorView” associated with it. If “TextFile” is a “SourceFile”, then there will be a “TreeNode” that contains the file name to associate with the file.

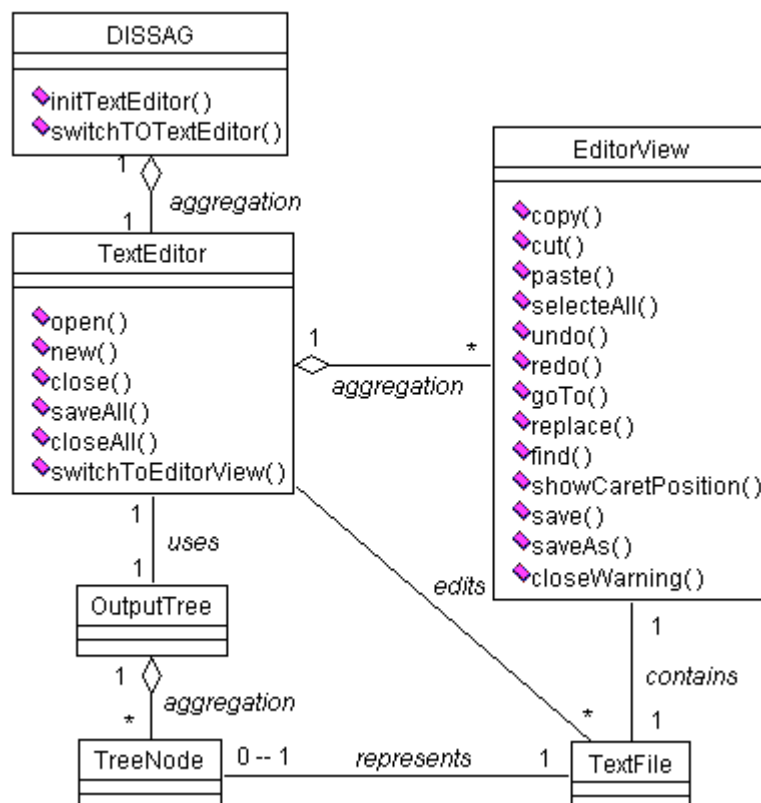


Figure 3.5: Text Editor OOA Class Diagram

3.5.5 Test Bed

Fig 3.6 shows the high-level OOA diagram for Text Editor. There are six objects appeared in this diagram. “DISSAG” has been discussed in 3.5.1.

“TestBed” is a part of “DISSAG”. It accepts “CommandFile” or “Command” as “input” and “generates” “RunningProcess”. It has a component named “CommandHistory”. It has the following functions:

- *inputCommand()* and *getCommandFromHistory()*. To enable users to type in commands directly or select a command from the command history.
- *InputFilePath()* and *browseForFilePath()*. To enable users to type in the path of the file to be executed directly or to browse the system storage to get the file path.

“CommandHistory” is a part of “TestBed” and consists of “Command”. It has functions: *add()*, *delete()* and *getCommand()* to add, delete a command or select a command from the Command history.

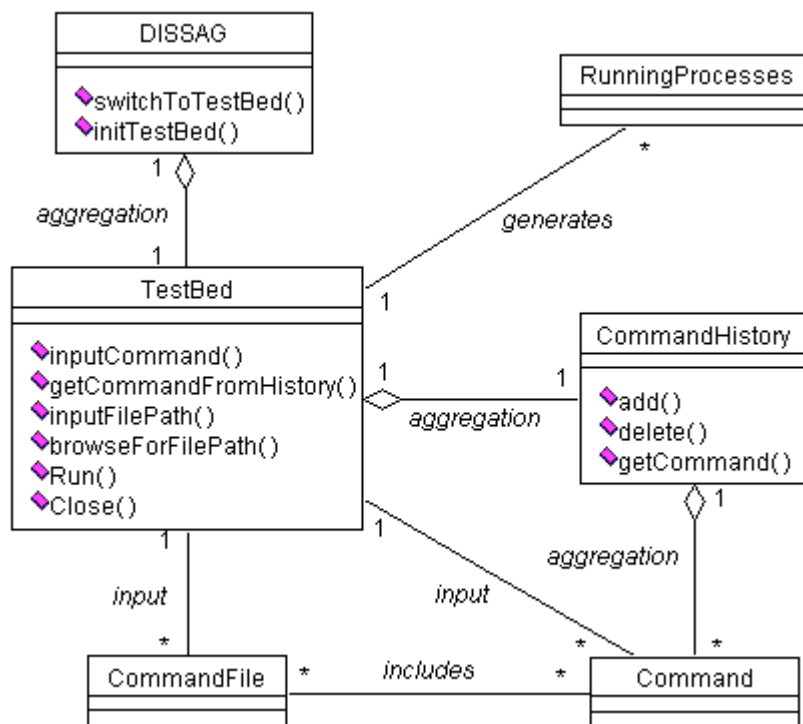


Figure 3.6: Test Bed OOA Class Diagram

3.6 Summary

Generally say, DISSAG should be able to be used by its users to model a given DIS, to generate detailed source code from the results of the modeling, to extend the generated code to get the whole solution source code, and to test the solution. To satisfy this key requirement, a set of fine requirements must be satisfied first. In this chapter, we have outlined all these requirements for DISSAG and its sub-tools, and also have made a brief comment for each of them.

Based on these requirements we have performed thorough OO analysis. The results of analysis are the OO class diagrams for DISSAG and its sub-tools, which have actually built up the concept model for DISSAG.

Chapter 4

DISSAG: design

4.1 Introduction

In Chapter 3, we have described in detail requirements for DISSAG and its sub-tools. Also we have performed thorough OO analysis based on them. In this chapter, our concerns will focus on the issues with the architecture and design of DISSAG and its sub-tools.

The quality of a software product derives from the quality of its design. This bestows particular importance on the design phase in the software development cycle. Therefore we have paid much attention to the design and have actually done it from different views and different levels.

DISSAG is a large tool and the design is obviously complicated. Some design is hard to be clearly described in limited space and also is hard for readers, who have not enough preparation, to grasp the ideas. Thus we carefully “designed” an introduction tour to help our readers to understand our design step by step. This tour will start from the low level rough design, and will be continued in the order of our actual design activities. The descriptions of architectures of our two most important tools, Diagram Editor and Code Generator will be inserted into this tour.

All designs have a common objective, to form a good implementation basis. Therefore at the end of the tour, we will present the OOD class diagrams for each of the DISSAG sub-tools. These OOD class diagrams have actually integrated various previous designs. Our implementation is directly based on them.

4.2 What is the task?

From the pure functional view, DISSAG can be abstracted into a conceptual model showed in figure 4.1.

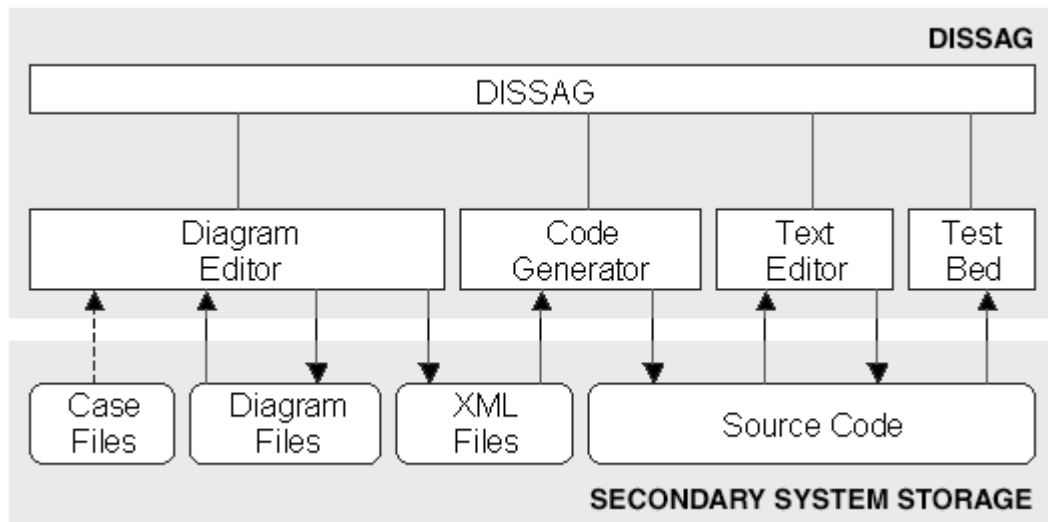


Figure 4.1: DISSAG conceptual model

Its main functions are: 1) Supporting modeling system by drawing and annotating UML-like diagram. 2) Supporting generating XML files encoding all modeling information. 3) Supporting generating source code from XML files. 4) Supporting further implementation by extending the generated code. 5) Supporting debugging and testing the source code.

Our task in design phase is then finding out proper and feasible solutions to satisfy above key functional requirements.

4.3 Rough design

The code generation for a given DIS is totally based on the DIS information. The more detailed, more accurate the information the code generation logic obtained is, the more proper code it will generate. Thus the most important thing is how to enable the code generation logic to obtain the DIS information.

One possible solution is using a wizard to guide users to input required data step by step. This works if the input data is not too much and relations among them are very simple. For a real DIS system, the information the code generator logic needs to know is far beyond a wizard's ability. So this is not a proper solution..

The second thought comes with the UML. The UML defines the UML model that can hold most system information in a well-designed structure. So if we could implement the UML model ourselves and let our code generation logic access the model, hopefully, the problem would be solved.

But it is not so easy. The key problem is that the UML model is designed for modeling all kinds of systems. That means there will be no place in UML model for some special information of a certain DIS. Therefore it is necessary to design our own model—DISSAG model which will be used to hold all DIS information and allow the code generation logic to access it.

The successive question is that even if we have created a DISSAG model, how can we properly input DIS information into the model. We design two ways to do this: 1) drawing the UML-like class diagram which will actually input information about all DIS classes and their relationships; 2) annotating each of the component in the diagram thus inputting more detailed information into the model.

Above thoughts form our rough design showing in Fig 4.2.

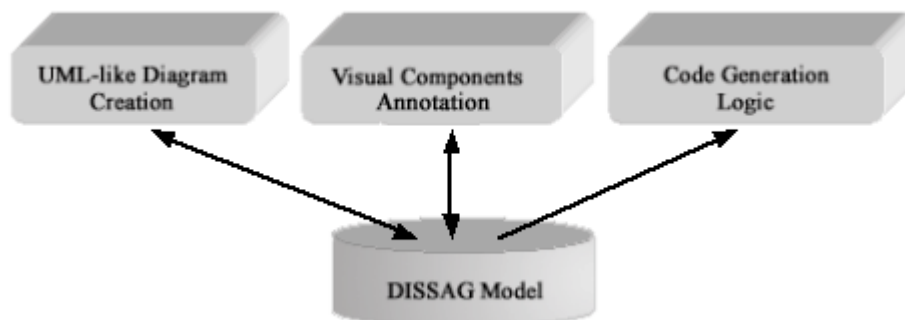


Figure 4.2: Rough design

4.4 DISSAG model design

From above analysis, the DISSAG model is a very important role in DISSAG. For each DIS, a DISSAG model is built by drawing UML-like class diagram and annotating its visual components, then the code generation logic can use the information from the model to generate code. Therefore the design of the DISSAG model becomes our first design task.

A model is normally defined by its metamodel. UML is a very good example. UML metamodel has more than 130 classes to define UML model [Booch *et al* 1998]. DISSAG presently supports UML class diagrams only; therefore DISSAG matamodel adopts the classes related to UML class diagram from the UML metamodel. On the other hand, to allow users to specify a DIS effectively so as to generate proper source code, DISSAG metamodel is enlarged with a set of classes that do not belong to UML metamodel. These new classes are useful in enabling the DISSAG model to support specifying DIS in details.

Fig 4.3 shows the relationship between the UML metamodel and the DISSG metamodel.

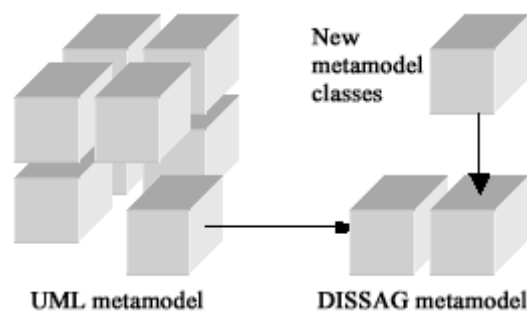


Figure 4.3: DISSAG metamodel

In DISSAG, all classes in DISSAG metamodel will be implemented with DISSAG base layer classes or attributes in those base layer classes (we cover this very soon). Thus the DISSAG model will be “living” in these base layer classes and their attributes. “Drawing class diagram” is actually to create the instances of the base layer classes, and “annotation” is actually to set the attribute values of those created instances. The process of “modeling” a DIS will in fact build up the DISSAG model for the DIS.

Fig 4.4 shows all the DISSAG metamodel classes that have been implemented with base layer classes. To make the diagram more elegant and easier to be understood, the other DISSAG metamodel classes that are implemented as attributes of those base layer classes are not showed. But as an example we list all those classes that implemented as attributes for class Class and show it in the form of attributes under the class name Class. In fact, there are lots of metamodel classes implemented as attributes to each base layer classes. All these DISSAG metamodel classes together define the DISSAG model.

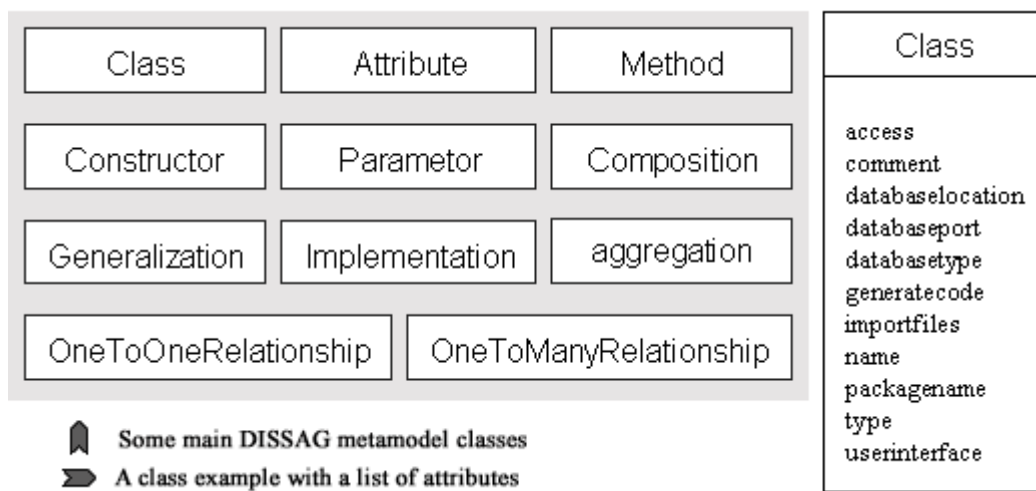


Figure 4.4: DISSAG metamodel classes

4.5 Diagram Editor architecture design

Finishing designing the DISSAG model, we encounter two new questions. One is how to implement the DISSAG metamodel; another is how to build DISSAG model for an actual DIS. In 4.2, we partially answered these questions, but not in detail.

Our basic solution comes from using JViews and BBW [Grundy *et al* 2000] [Grundy *et al* 1998a] [Grundy *et al* 1998b]. As we have introduced in the related tools section of chapter 2, JViews is a component-based architecture and java framework that is specially designed for implementing modeling tools. BBW itself is a metaCASE tool, but it can be used directly as a framework that provides necessary supports for constructing a

visual environments. While BBW components work visually and provide a presentation layer, JViews components work invisibly and provide underlying sophisticated supports to all those visual activities. Both of them can be adopted directly by our tool, Diagram Editor.

But we have to define at least four groups of classes and combine them with JViews and BBW to make our tool Diagram Editor working.

- 1) Implementation of the DISSAG metamodel classes. Some metamodel classes, such as Class, Attribute, Method, OneToOneRelationship etc, will be implemented as base layer classes such as DISSAGBaseClass, DISSAGBaseOneToOneRelationship etc. The other metamodel classes, like Name, Type etc will be implemented as attributes in the related base layer classes. Thus the DISSAG metamodel has been actually implemented with the base layer classes in this group.
- 2) Defining the visual notations of the DISSAG metamodel classes. This involves two steps, defining their visual shapes and defining the java classes that will create these shapes.

Fig 4.5 shows the visual notations we designed for DISSAG metamodel classes.

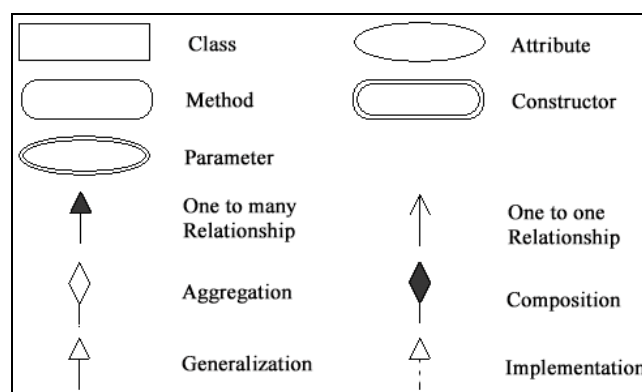


Figure 4.5: DISSAG notations

The major difference between the UML notations and DISSAG notations is that DISSAG separate attribute, constructor, method, and parameter from the class notation and let them have their own

notations. The purpose for doing so is to enable annotation to go under the class level and thus make annotation more effective and more efficient.

We design to implement the DISSAG notations with JavaBean [JavaBean Document]. Each JavaBean component can be configured at runtime with its associated property sheet, therefore we can embed the specification information such as name, type for Class etc into its property sheet and have chances to specify the DIS by specifying diagram components at any time.

- 3) After 1) and 2), we have actually implemented the metamodel and defined the notations. As a simple tool, if we add some suitable components to connect the base layer objects and its corresponding visual components properly, the Diagram Editor will work. But JViews provides us a chance to make a powerful tool. What we need to do is to define view layer classes, such as `ClassIcon` or `OneToOneRelationshipGlue` etc, according to the base layer classes we have defined. These classes have not only attributes what their corresponding base layer classes have, but also attributes related to the positions and sizes of their corresponding visual components. It is the view layer that actually makes the JViews-based tools to support multi-views, collaborative view editing and visioning.
- 4) The last step is to defining the relationship between the view layer components and the base layer components. The result is a group of classes, such as `OneToOneRelationshipGlueToBase`, `ClassIconToBase` etc, that will map changes between classes in the view layer and the base layer.

All these four steps look awesome. But JComposer [Grundy et al 2000], as we introduced in related tools section, can help us build all these classes effectively and efficiently. We will cover this in Chapter 5.

Fig 4.6 shows some key JViews components during an actual modeling process, from which we can clearly see the three layers architecture of the Diagram Editor using JViews. The key to understand this figure is that each component is actually an instance of a class. Such as the component, `Class Icon`, in view layer is an instance of `ClassIcon` class, and the component, `Base Class`, in base layer is an instance of `DISSAGBaseClass`.

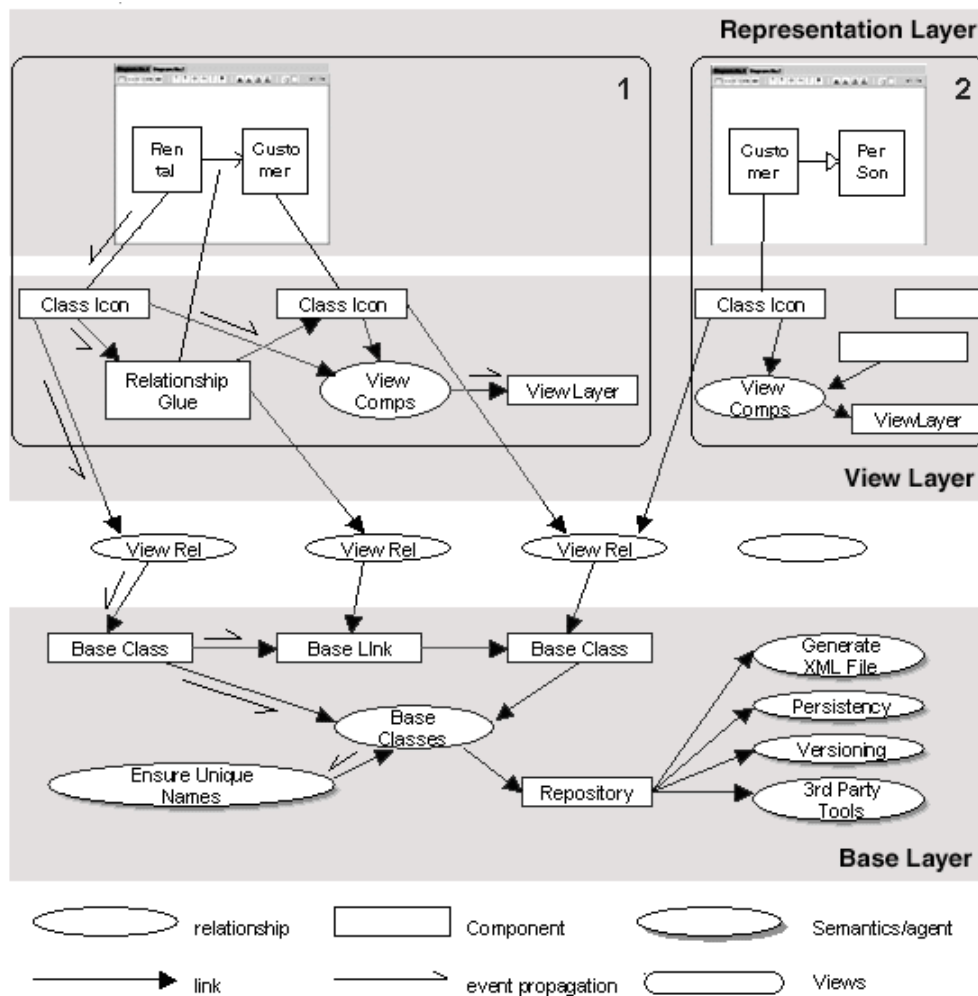


Figure 4.6: Diagram Editor architecture

With this figure, we can see how the Diagram Editor tool satisfies our expectations for a modeling tool that supporting drawing class diagrams and annotating each component in detail.

When a visual component, such as a class shape, is added, the property sheet for this component will be appear. And if we enter the name, e.g. Person, and tell the tool we have finished by hitting the return key, an event will be generated and as the response to the event, an instance of view class, ClassIcon, will be created and its attribute for name will be set to “Person”. As the same, an instance for base layer class, DISSAGBaseClass, will be created and have the name attribute set to “Person”, a relationship component, an instance of ClassIconToBase, connecting these two components is also created. So in this way, adding a new visual component or deleting a visual component will result the corresponding changes in the base layer.

The annotation to each visual component works the same way. The difference is that there will be no component adding or deleting in view or base layer, only the attributes of the existed components will be set according to the information obtained from the property sheet changes.

By deploying JViews, some features such as supporting multi-views will be automatically added to the tool. Fig 4.6 shows two views coexist. Each view has its own view layer components thus the same visual component can appear in different views with different size and position. But they share a base layer—only one DISSAG model.

4.6 DTD design

In previous sections, we have outlined our designs related to DISSAG model. We have talked about what is DISSAG model and how to build a DISSAG model for a given DIS. Now we move to how to use the modeling results.

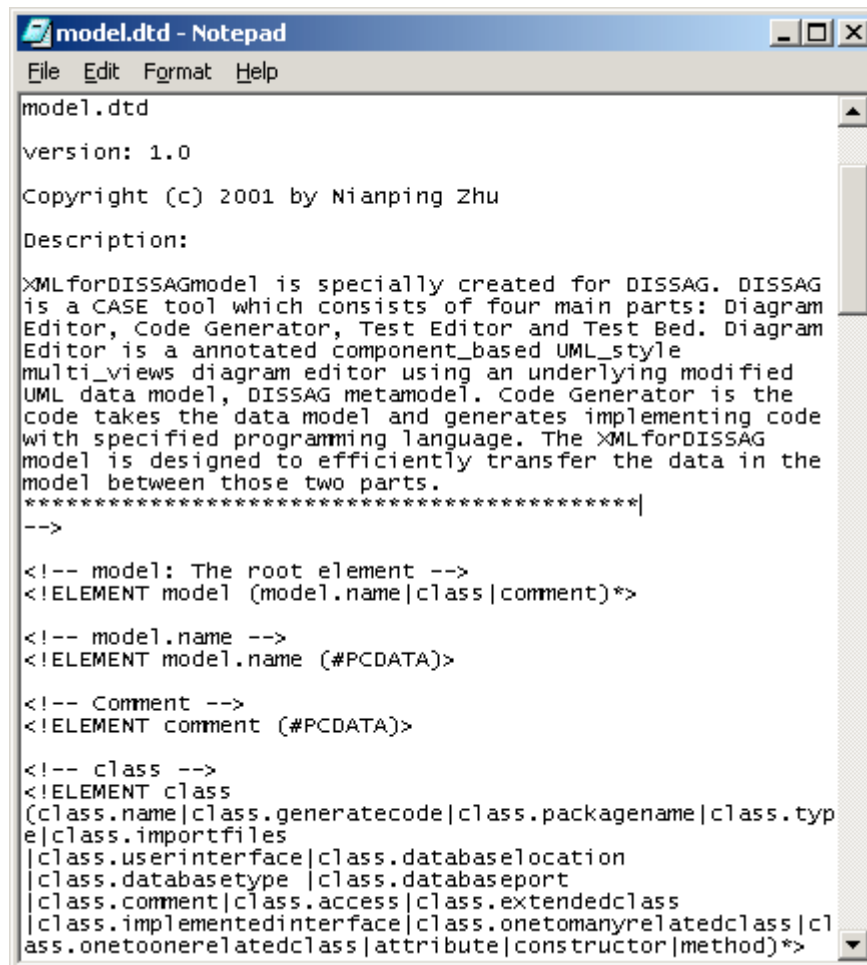
Using UML diagrams to model the real world systems can help users to abstract the practical problems, and find out the possible solutions. The diagrams can also be used as a media for developers to communicate each other. But if these are the final objectives, DISSAG model will be useless. Because the UML model is enough, maybe better. DISSAG model can go further than UML model as they can be used to generate more accurate, more detailed source code.

Theoretically, code generation logic can access the model directly and this might be the fastest way. But we add one ring, XML files between these two. We will first generate XML files that contain all useful information from the DISSAG model and then pass the XML files to the code generator. We do so for the following purposes:

- Separating code generation logic from modeling logic will enable both to be developed, maintenance and upgraded individually.
- Separating these two brings up a chance to integrate a third party tool. We may export the DISSAG

model to another code generator, or our code generator may be used to generate code based on the results of the other modeling tools.

- If a remote user needs the DISSAG model information, the XML files would be much easier to be passed over networks.



```

model.dtd
version: 1.0
Copyright (c) 2001 by Nianping Zhu
Description:
XMLforDISSAGmodel is specially created for DISSAG. DISSAG
is a CASE tool which consists of four main parts: Diagram
Editor, Code Generator, Test Editor and Test Bed. Diagram
Editor is a annotated component_based UML_style
multi_views diagram editor using an underlying modified
UML data model, DISSAG metamodel. Code Generator is the
code takes the data model and generates implementing code
with specified programming language. The XMLforDISSAG
model is designed to efficiently transfer the data in the
model between those two parts.
*****|
-->
<!-- model: The root element -->
<!ELEMENT model (model.name|class|comment)*>
<!-- model.name -->
<!ELEMENT model.name (#PCDATA)>
<!-- Comment -->
<!ELEMENT comment (#PCDATA)>
<!-- class -->
<!ELEMENT class
(class.name|class.generatecode|class.packagename|class.type
|class.importfiles
|class.userinterface|class.databaselocation
|class.databasetype |class.databaseport
|class.comment|class.access|class.extendedclass
|class.implementedinterface|class.onetomanyrelatedclass|cl
ass.onetoonerelatedclass|attribute|constructor|method)*>

```

Figure 4.7: Model.dtd

The contents of XML files will be defined by a DTD (Document Type Definition) [McLaughlin 2000] file, which contains meta-information about valid elements or attributes names and values, and how elements can nest in each other.

The DTD file we used for DISSAG is directly derived from the DISSAG metamodel. But there are two important differences between these two. Firstly there is no element for any relationships in DTD. That means

the XML generator will be assigned a task to “pre-process” the information from the DISSAG model. Clearly, the XML generator will dissolve those relationships by adding proper attributes that represent the relationships to related elements. Secondly, all attributes, methods etc appear inside the classes they belong to. The main merit for doing so is to enable the code generation logic easily get all information related to a single class and thus to improve the efficient of the code generation logic.

The DTD file we designed to be used in DISSAG is partially shows in Fig 4.7. The Whole DTD file will be attached as an appendix of this thesis.

4.7 Code Generator architecture design

Code Generator is the core part of DISSAG. As mentioned in 4.6, the code generation logic will not directly access DISSAG model. Instead, it will generate code from the XML file that contains all DISSAG model information. Therefore one of the most important components of Code Generator will be XML parser that converts XML files into a form so that the code generation logic can directly obtain information they needed.

Currently, two major API specifications define how XML parsers work: SAX and DOM. The DOM specification defines a tree-based approach to navigating an XML document. In other words, a DOM parser processes XML data and creates an object-oriented hierarchical representation of the document that you can navigate at run-time [Gulbrandsen 2000].

The SAX specification defines an event-based approach whereby parsers scan through XML data, calling handler functions whenever certain parts of the document (e.g., text nodes or processing instructions) are found [Simon 2001].

The SAX implementation is generally faster and requires fewer resources but SAX code is frequently complex, and lack of a document representation.

We decide to use DOM API to parse our XML files. The main reasons are:

- Our XML files are small in size. The information included in the XML files is just DIS data. So the size of XML file won't be larger than that of the text file that describes the same DIS using normal English syntax. Thus when it is parsed into DOM and stored in the memory, it won't take up much space.
- Once an XML file is parsed into DOM, it will stay in memory and we can access it as many times as we want. This is very useful for our Code Generator. Often we need to generate source code for the same DIS using different middleware. By using DOM, we do not need to load and parse the same XML file again and again.
- By using DOM, we can separate XML parsing logic from its successor, code generation logic. Therefore it will be much easier to implement, maintain and upgrade Code Generator. If using SAX, the "parser" will be integrated into code generation logic. The effect will be reversed.
- DOM API provides means to access each node (information pieces) very easily. It is very convenient for code generation logic to go through the tree and get the information wanted.
- DOM API is comparatively easier than SAX API. So we can develop Code Generator with lower cost (reducing time to learn SAX API).

Using DOM API to parse XML files actually divided Code Generator into two parts: the XML parser part and the code generator part. The code generator part again will be divided into four parts, one for each generator with a kind of middleware, to satisfy the requirement, generating source code using CORBA, RMI, DCOM and JDBC as middleware.

Code Generator architecture is formed based on above description. Fig 4.8 shows the architecture by showing those functional parts and their relations, and putting them together in a typical workflow chart.

"XML File Parser" converts XML files to their DOM notation. The four generators will generate source files with the corresponding middleware. Under each of these four code generator components, there will be a number of fine components with dedicated functions; some of them will be shared by the four generators.

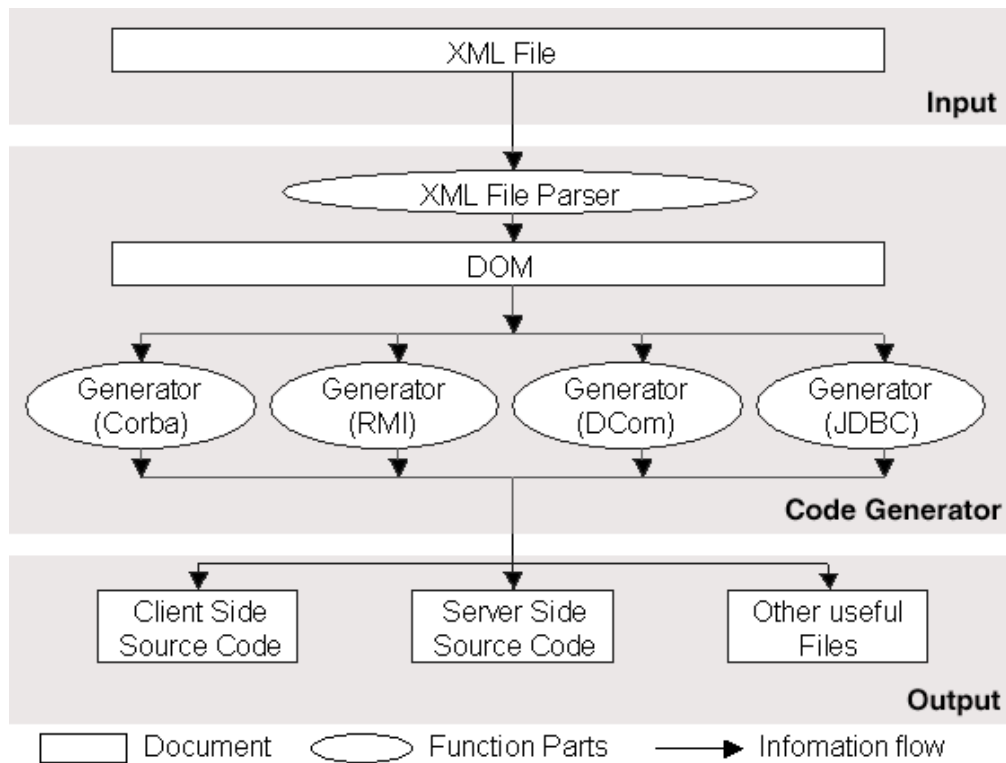


Figure 4.8: Code Generator architecture

4.8 Code generation logic design

We have mentioned in last section that we will use DOM API to parse XML files. As a result, our code generation logic can access DOM at any time because it will stay in the memory. The COM API provides methods to get the values of any elements or any attributes inside elements. So when we design code generation logic, we can simply assume that all necessary data are ready to be used. Then we can concentrate on how to use them to generate code.

Our basic design idea for the code generation logic can be described like the following sequence:

- Get the output directory from the user for the generated files.
- Get the middleware type from the user.
- Pass them to the corresponding sub code generator together with the DOM. For example, if a user chooses CORBA as middleware, those arguments will be passed to a sub code generator called CORBA code generator.

- Get all Class elements and put them into a Vector.
- For each class in the vector, do the following things.
- Get the value of the attribute “generateCode”, if it is “false”, return; else, do next.
- Pass the node of the class to its fine code generators. For example, for the CORBA generator, there are fine code generators like IDL file generator, remote object implementation generator, remote server generator, local application generator and deploying file generator etc.
- Each fine code generator will generate code based on the information included in the Class node. The generated code will be written out to the specified output directory. Normally each code generator will generate classes in pair. One is the class with all of the generated code and it is named with a “G” as appendix to its normal name. Another one is an almost empty class with the normal class name. The later extends the former. Users will add more pieces of codes into the later to extend the generated code.

To save space, the details of the fine code generators will not be discussed here. Some of them are very simply, like the IDL file generator, the code generation logic will simply get the information for each attribute and method then write it out in IDL in syntax. Some will be complicated, like the local application generator, the code generation logic will have to read more data and consider more factors.

4.9 OOD class diagrams

In last chapter, we have performed OO analysis for DISSAG and its sub-tools. We have drawn the OOA class diagram for it. The OOD class diagrams base on the OOA class diagrams and integrate the result of the architecture design and some other designs. To avoid repeat, we will introduce the OOD class diagrams for Diagram Editor, Code Generator, and Text Editor. In chapter 5, we will have a section talking about how to merge all of the four DISSAG sub-tools into a single visual environment.

4.9.1 OOD class for Diagram Editor

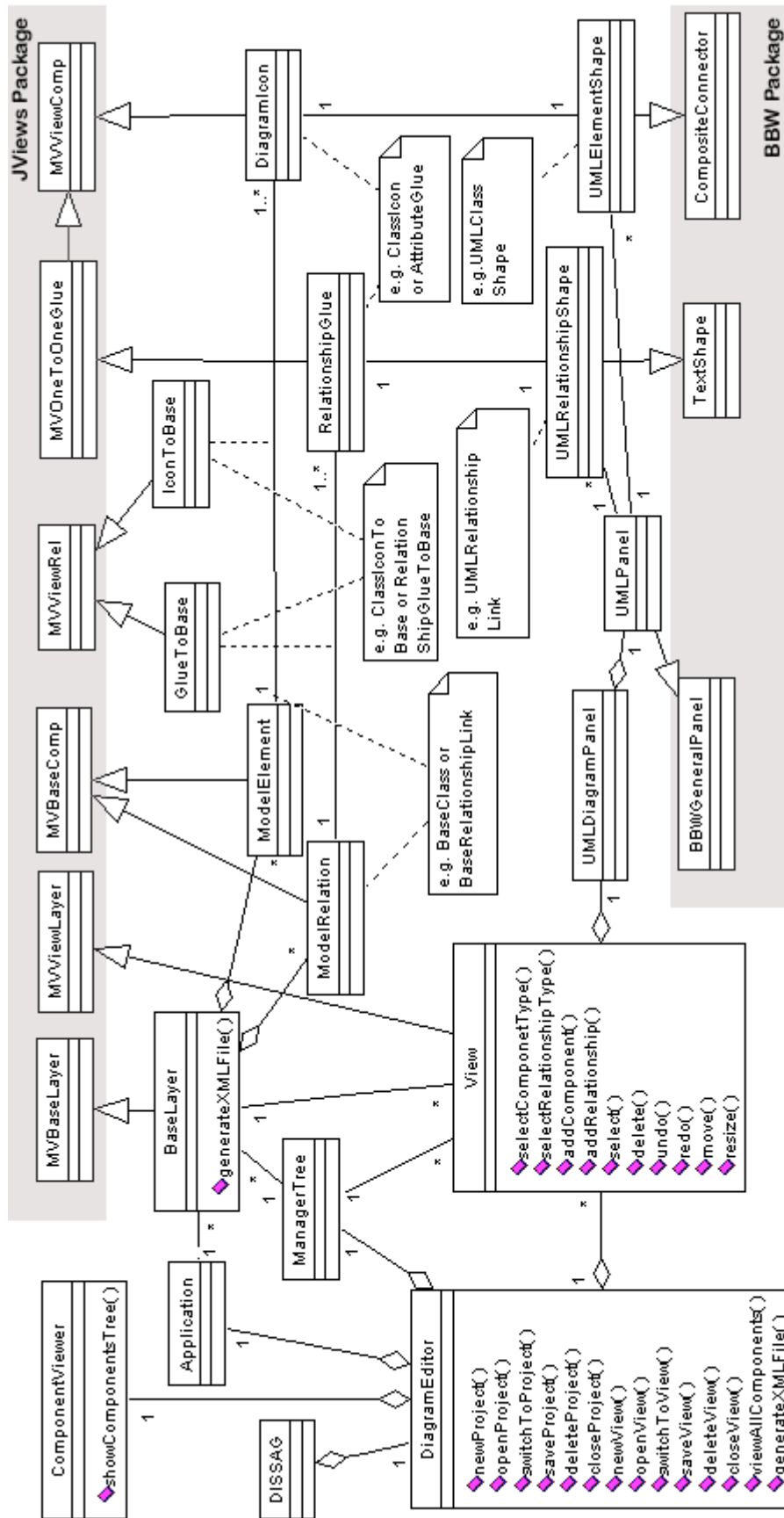


Figure 4.9: OOD class diagram for Diagram Editor

Fig 4.9 shows the whole OOD class diagram for Diagram Editor. There are many classes in this diagram. Some classes belong to other packages, such as JViews and BBW. To understand this diagram, we group these classes into the following groups.

1) Classes extending JViews framework

Fig 4.10 shows some main JViews classes that form the three-layer architecture. But it does not show those classes related to generating, passing and responding events, which in fact enable this architecture to work properly.



Figure 4.10: Some classes of JViews framework

The following classes in Diagram Editor extend JViews classes:

- The base layer classes that implement the DISSAG metamodel. E.g. BaseClass, BaseAttribute and

BaseRelationship etc. All of them extend BaseComp.

- The view layer classes. There will be two kinds of extensions: entity extension and relationship extension. The former, e.g. ClassIcon and AttributeIcon, extends MviewComp; the latter, e.g. RelationshipGlue, extends MVOneToOneGlue that is a class in JViews and extends MVViewComp.
- The view relationship classes. For those view classes representing entities, such as ClassIcon, there will be IconToBase classes, such as ClassIconToBase and AttributeIconToBase. For the rest, there will be GlueToBase classes, such as RelationshipGlueToBase. All these classes extend MVBaseRel.
- The BaseLayer class that extends MVBaselayer class. New functions such as “generateXMLFile” will be added.
- The View class that extends MVViewLayer.

2) Classes extending BBW frame work

BBW framework is used to construct visual components in presentation layer. Like JViews, many BBW components are reusable and extensible. Fig 4.11 shows some BBW classes, which we are interested in. Some of them have been directly extended by Diagram Editor classes in practice.

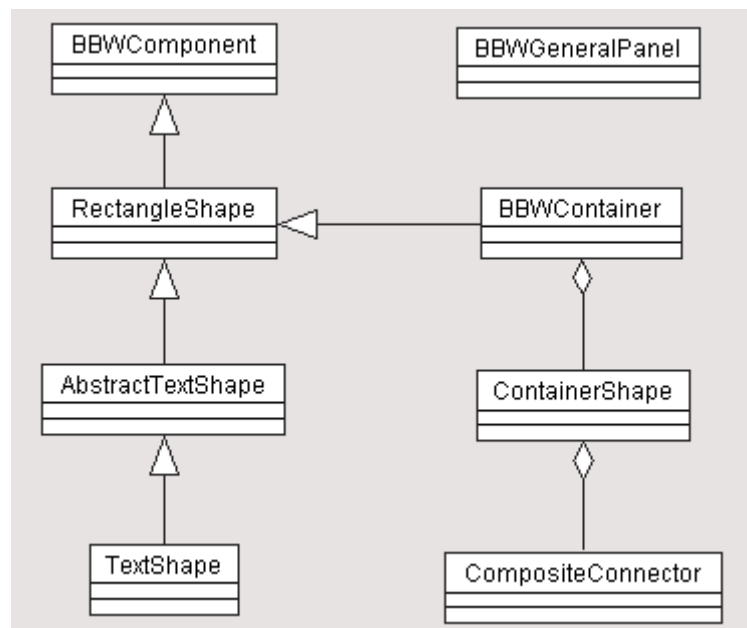


Figure 4.11: Some BBW components

The following classes extend BBW classes:

- UMLPanel that extends BBWGeneralPanel. A BBWGeneralPanel provides some basic panel capability for BBW-based tools. It handles BBWTransactions, which wrap a transaction around a user's action (such as dragging a shape) so that JViews can reduce the details. UMLPanel will be integrated into UMLDiagramPanel together with a tool panel, which provides various tools related to UML diagram. UMLDiagram then will be used by a View component to display visual component.
- All classes that present icons for entities in presentation layer. They extend TextShape. For example, UMLClassShape extends TextShape and will provide an iconic representation for UML class component.
- All classes that present icons for relationships in presentation layer. They extend CompositeConnector. For example, UMLRelationshipLink extends CompositeConnector and will provide an iconic representation for UML relationship component.

3) ManagerTree class

The architecture of Diagram Editor satisfies many of its requirements. But there are still some minor concerns. One of them is how to organize multiple projects and multiple views. We propose a manager tree to solve this problem. This tree has project names as its nodes, which in turn has names of the views, which belong to the project it represents, as its child nodes. Each node of the tree will be used as an anchor to link its associated project or view. With its visual appearance, the manager tree benefits us with:

- Diagram Editor can model multiple projects simultaneously now and can switch to any project that is being viewing or modeling with a handy manner.
- Diagram Editor now can manager multiple views much easier and can switch to any view that is currently opened.

4) Other classes

The other classes in the OOD class diagram have been appeared in the OOA diagram. We will not make any more comments about them.

4.9.2 OOD class diagram for Code Generator

Fig 4.12 shows the part of the OOD class diagram of the Code Generator. Comparing it with the OOA class diagram, we may see that there are two different parts. One is the XMLParser class that will parse XML files using SUN JAXP API. Another one is that the CodeGenerator now consists of some sub-generators which can be used to generate code with given middleware type. For example, CorbaGenerator now has the function to generate code using CORBA as middleware. Each sub-generator again consists of some fine generators that can be used to generate particular code pieces or classes to be a part of the whole solution. For example, there is a class called GenerateIDLFiles that can be used specially to generate IDL files CORBA remote objects. In Fig 4.12 we only show those fine generators under the CORBA generator.

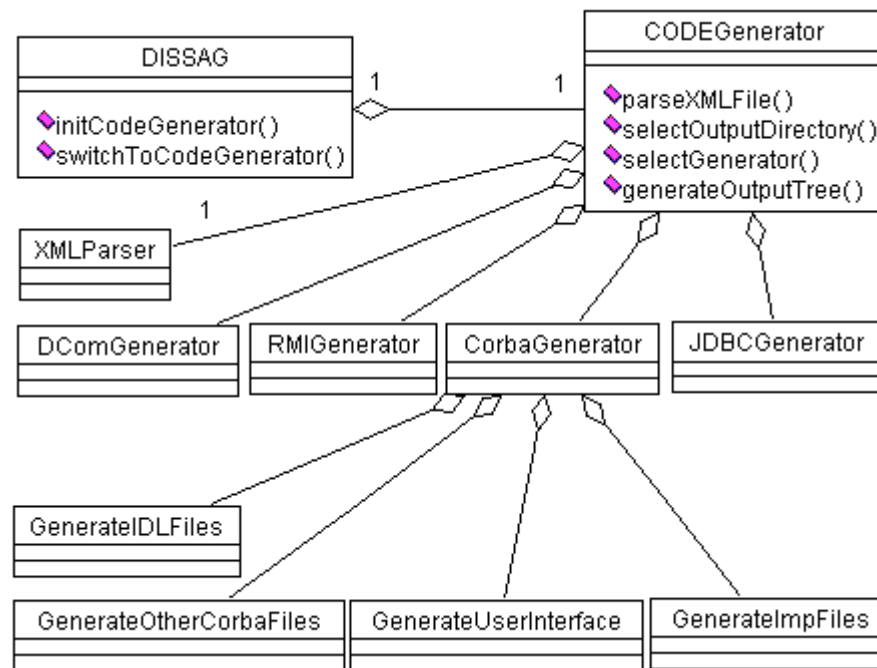


Figure 4.12: Code Generator OOD class diagram

4.9.3 OOD class diagram for Text Editor

There is not too much difference between the OOA class diagram and the OOD class diagram of the Text Editor. Fig 4.13 shows the OOD class diagram. There can be multiple text editor views working at the same

time, and there will be a tree to manage them. Some dialog classes are necessary to help editors to work properly. There will be some classes adding in in actual implementation phase; but in the design phase, we stop at this level.

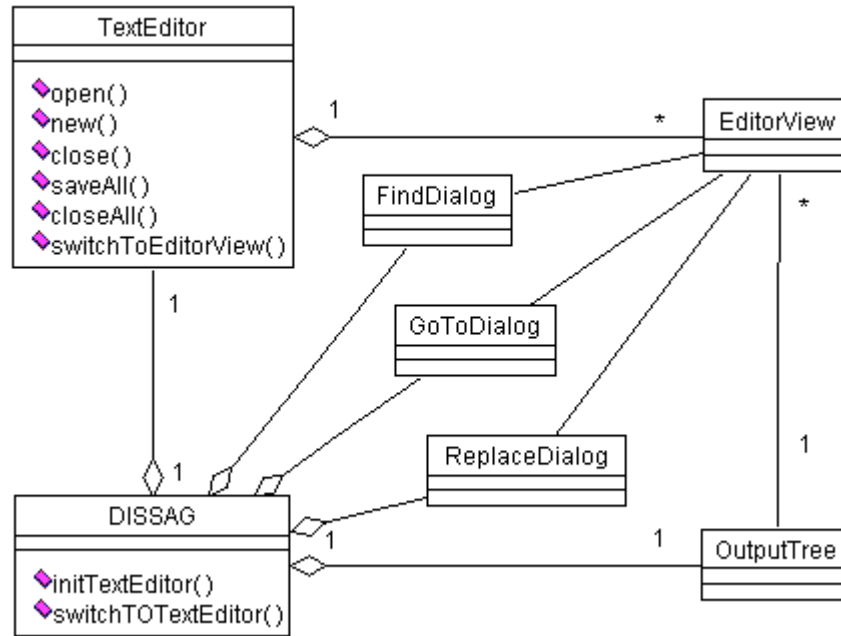


Figure 4.13: Text Editor OOD class diagram

4.10 Summary

In this chapter, we have outlined all designs we performed in the design phase of DISSAG development life cycle. We have covered all important parts in DISSAG development, such as defining the DISSAG metamodel, defining notations for metamodel, defining the DTD file which defines the XML file, designing the architecture for Diagram Editor and Code Generator, designing the code generation logic etc. At last, we have introduced the OOD class diagrams for DISSAG sub tools, which are the base of our implementation.

Chapter 5

DISSAG: implementation

5.1 Introduction

In previous chapters, we have described the requirements architectures and some design details of DISSAG. In this chapter, we will discuss some practical issues related to its implementation.

By the term *implementation* of a software system we mean the transformation (coding) of the design results into programs using a certain language. The major product of the implementation phase of the software development cycle is the source code for the system. The essential and most important issue with coding is deciding what programming language will be used. Often the very success or failure of software can depend on the choice of languages [Mynatt, 1990]. We choose JAVA as the language to implement DISSAG.

Some other issues with coding include how to get the code and how to solve some particular problems with the chosen language. Basically, we have three ways to get desired code: the easiest way to get proper code is to directly use or to extend the existed code pieces or complete applications that have the same desired functions; another easy way is to deploy some CASE tools or metaCASE tools to automatically generate the ideal source code (DISSAG itself is a kind of this tool); even both of them are feasible, the third way can not be avoided, that is writing code. To implement DISSAG, we used all of them. For example, we used SUN JAXP 1.0 directly as our XML parser; we used JComposer, which is a metaCASE tool, to generate code for building the architecture of our Diagram Editor tool and of course we did write code ourselves.

For a certain function, different language has different way to implement it. So for some particular problems, we will describe our JAVA solutions. E.g. we will describe in detail how we implement the DISSAG user interface in JAVA; how we use multiple threads, one of the merits of JAVA, to achieve better performance; and how to run another application from DISSAG, the underlying mechanism of our Test Bed tool.

5.2 Using JAVA as our implementation language

The question of which is the “right” programming language has always been a favorite topic of discussion in programming cycles. Actually, the choice of a programming language for the implementation of a project frequently plays an important role. In ideal case, the design should be carried out without any knowledge of the later language so that it can be implemented in any language [Pomberger et al 1996]. Unfortunately, in practice this is seldom feasible because certain constraints require a certain programming language.

Our design for DISSAG implies that our implementation language should be object-oriented, but does not specify a particular one. In implementation stage, we decided to use JAVA as our implementing language. The decision based on the following reasons:

1. Java is an object-oriented language.

In previous phases of developing DISSAG cycle, we have always used object-oriented analysis and design skills, so it is no doubt in implementation phase we will continue to use OO technologies by using an object-oriented language. Actually, DISSAG and some of its sub-tools have component-based architectures. To implement them, an object-oriented language is essential.

2. Java is platform independent.

The potential users of DISSAG are software developers specially those working in DIS domain. It is not hard to image that they are working with various platforms. To enable all of them to use DISSAG on their

platforms, DISSAG itself must be platform independent. To achieve this point, we have to choose a platform independent language to implement DISSAG.

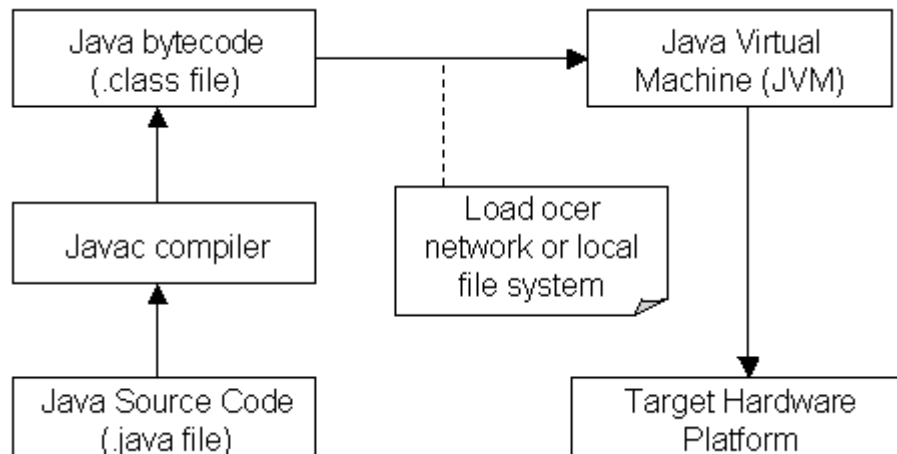


Figure 5.1: Java code execution in JVM [Hamilton 1999]

Java meets this requirement. It accomplishes platform independent through an intermediate computation stage. See Fig 5.1, Java source code is compiled into a byte code, which is machine-independent. Interpretation will take place before the byte code runs on the native instruction set of the target processor. The interpretation is done through a Java virtual machine (JVM), a software layer that forms a bridge between the byte code and actual hardware. The JVM takes system calls and interprets them into corresponding machine instructions. Thus Java source code can run on any processor having a JVM. [Hamilton, 1999]

3. Java is multi-threaded.

DISSAG is a software development environment using which users can perform most of the developing tasks. Often users want to perform some tasks at the same time. For example, a user is testing his source code using Test Bed tool, when he sees something wrong, such as a button name has been misspelled, he may want to change the source code immediately, but not to stop the running test. Then multiple executing threads are needed. Therefore we should choose a language that supports multiple threads.

Java is one of the first languages to be designed explicitly for the possibility of multiple threads of execution running in one program [Budd 1998]. A single JAVA application can have many threads executing independently, continuously and concurrently. Again JAVA satisfies the requirement for the implementing language for DISSAG.

4. Java has very sophisticated support for graphical user interfaces.

DISSAG is a visual environment. So it is different from other applications in having more strictly requests for the user interface. And to realize the proposed powerful functions, its interface will be definitely very complicated.

JAVA is good at building sophisticated graphical user interfaces. Java Foundation Classes provides pure Java-based user interface facilities, which can be tailored by programmers, to help them to build more realistic and portable graphical user interfaces (GUIs).

5. Some existed components we adopted in DISSAG are developed in JAVA.

No matter from the point of its functions or from that of its source code, DISSAG is a huge software. Even there are still lots of new features to be integrated; it currently possesses around 200 classes and 20.000 lines. So it is impossible to finish such a big job in several months without the aid of some useful tools or without adopting some existed software components. Many of the components we used are developed in JAVA, e.g. JViews, BBW and JAXP 1.0. We choose Java in order to integrate them more conveniently and more smoothly.

6. The source code generated by DISSAG as the solutions to DIS is Java code.

For many reasons, Java has been chosen by many software developers to implement DIS. As a result, DISSAG will generate Java code for DIS. Theoretically we still can implement DISSAG with other

languages. But with Java, we save the time to learn a second language and to think with two languages when actually implementing DISSAG.

7. The most important fact is that using Java, we feel no difficult to transform our design idea into actual Java code which meets any functional and non-functional requirements of DISSAG. That said Java satisfied all our expectation for a programming language to implement DISSAG.

5.3 Using Java JFC/Swing to implement DISSAG interface

From the requirements and the design of DISSAG, we know that there are many issues related to the DISSAG interface. We can restate some of them here:

- DISSAG is a visual software development environment.
- DISSAG consists of four sub-tools and is able to switch to any one of them at any time.
- Diagram Editor has a manager tree.
- Diagram Editors can be used to model multiple projects and there will be multiple views with each project model.
- Code Generator has an output tree to visually index its generated files.
- Text Editor can be used to edit multiple files simultaneously.

Many implementations can satisfy all above requirements related to the user interface. As we have chosen JAVA as our implementation language, we use JFC/Swing to implement DISSAG user interface.

JFC is short for Java Foundation Classes, which encompass a group of features to help people build graphical user interfaces (GUIs), such as containing Swing components, pluggable Look and Feel support, accessibility API, Java 2D API and Drag & Drop support. Swing was the code name of the project that developed the new components. Although it's an unofficial name, it's frequently used to refer to the new components and related API. It's immortalized in the package names for the Swing API, which begin with javax.swing. We use Swing components in javax.swing package to build our tool interface.

The base component we used is JFrame. Its content pane is divided into five main areas. See Fig 5.2. The manager area and editor area are the left and right components of a JSplitPane. All common tool icons are contained in a JToolBar. A JLabel and a JTextField, both used to display dynamic information, are contained in a JPanel. A GridbagLayoutManager arranges them properly into their planned position.



Figure 5.2: DISSAG interface layout

There are two interesting area to be further described:

1. *Editor Area.* This is the main working area. All diagram editor views and text editor views will appear here. To achieve this, we used JTabbedPane. Each diagram editor view and each text editor view will be included into JTabbedPane as a separate pane. So we implemented View class (in DiagramEditor) to let it extend JPanel, then it can be treated as a pane. The EditorView class (in Text Editor) extended JEditorPane and can be treated as a pane also. The tab names of the JTabbedPane are good index and by clicking on the tab users can switched to any views easily.

The switching among the views can also be associated with their manager trees on the left side. That means by clicking a node of the tree, the corresponding pane will be showed. Adding or deleting nodes of the tree also add or delete the corresponding panes. Fig 5.3 shows the classes related to building the part of the DISSAG interface.

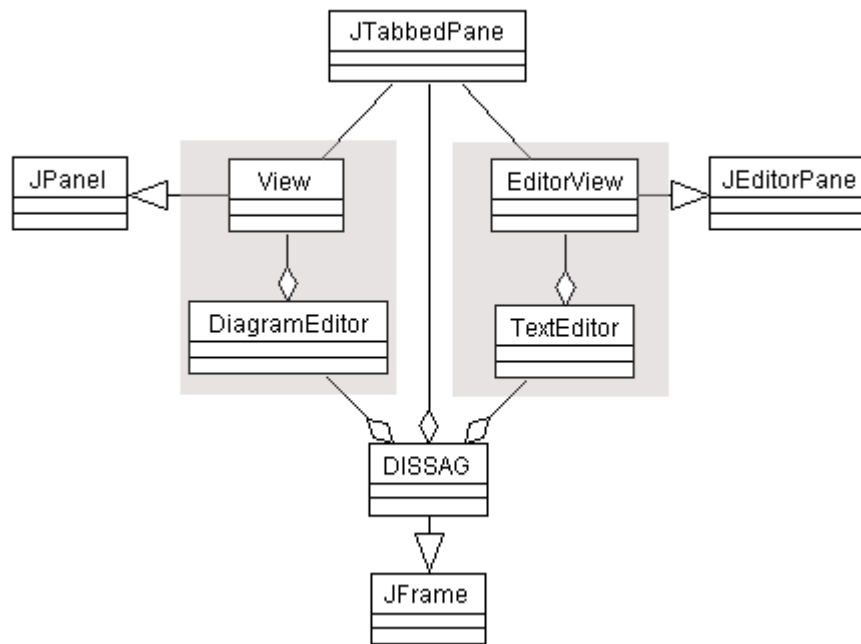


Figure 5.3: Classes related to editor area

2. *Manager Area*: There are two trees in this area. Each tree is wrapped in a JScrollPane and then included into a JTabbedPane as a separate pane. Each Pane contains a set of tool icons as well. Switching between the panes is actually equivalent to switching between the tools. Fig 5.4 shows classes related to building this part of the DISSAG interface.

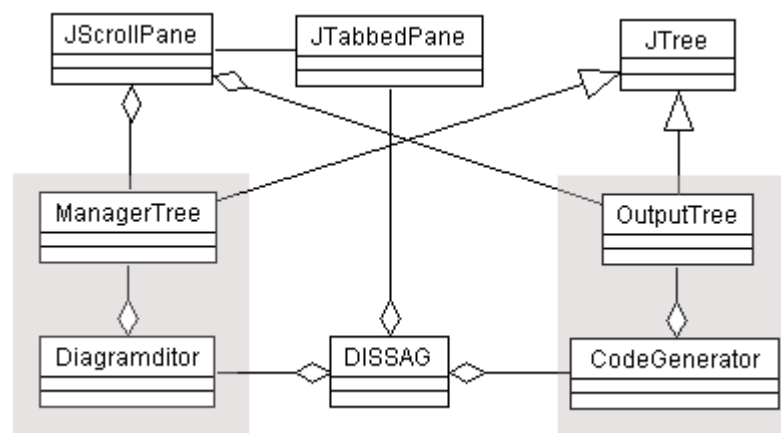


Figure 5.4: Classes related to manager area

Fig 5.5 shows the final interface. It not only satisfied all requirements related to the user interface of DISSAG and its sub-tools, but also got some extra features:

1. Tool icons plus tool tips strengthen the feeling of “visual”.
2. Working area is resizable so that users may use their working space more effectively.
3. Similar to popular IDE interfaces reduces users learning curve.

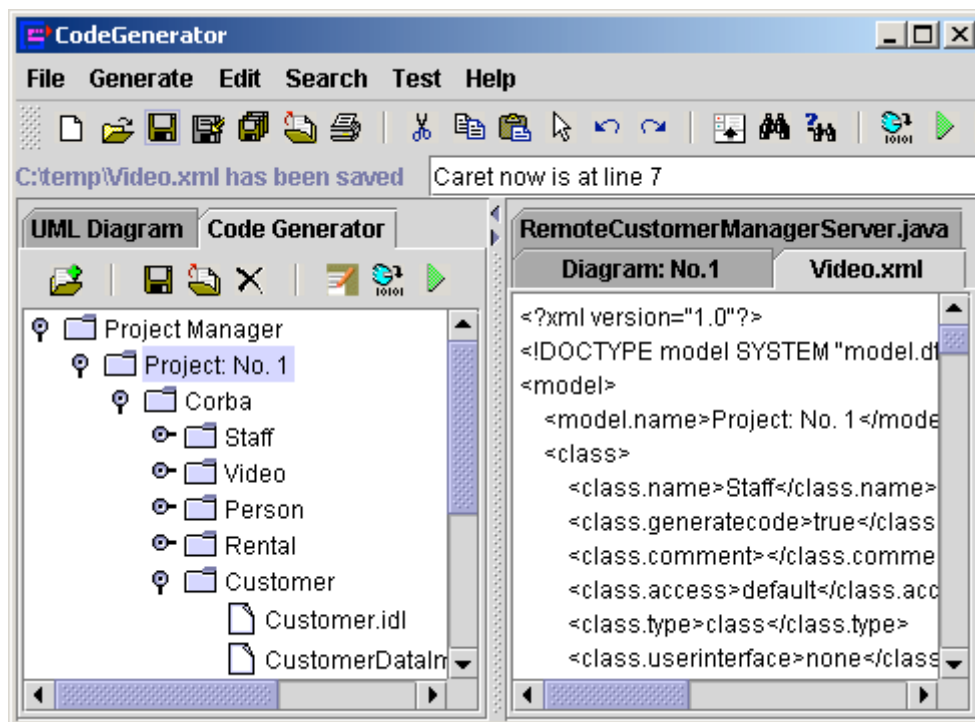


Figure 5.5: DISSAG interface

5.4 Using JComposer to implement Diagram Editor architecture

In chapter 4, we have described that we will use JViews framework as the basic architecture of our Diagram Editor tool. We have also discussed how to extend JViews and BBW to obtain our desired functions. It should be feasible if we change these design ideas into real code and write it out “by hands”. But there is no doubt that although JViews is available, the extension job would be an awful time-consuming job.

Fortunately there is a very useful metaCASE tool, JComposer [Grundy et al 1998] that we can use to

automatically generate the code we wanted from a proper visual specification. In 4.5, we have already introduced how to extend JViews to actually build the tool. Now we have a glance at how to use JComposer to finish most of the tough job.

The main steps using JComposer to generate our Diagram Editor source code are summarized below.

1. *Metamodel Implementation.* The key in this step is visually specifying what base components will be in the base layer. These base components are all of the DISSAG metamodel classes that have their own visual notations. The other metamodel classes that have no visual notions will be specified as attributes of the components. Fig 5.6 shows part of the whole visual specification.

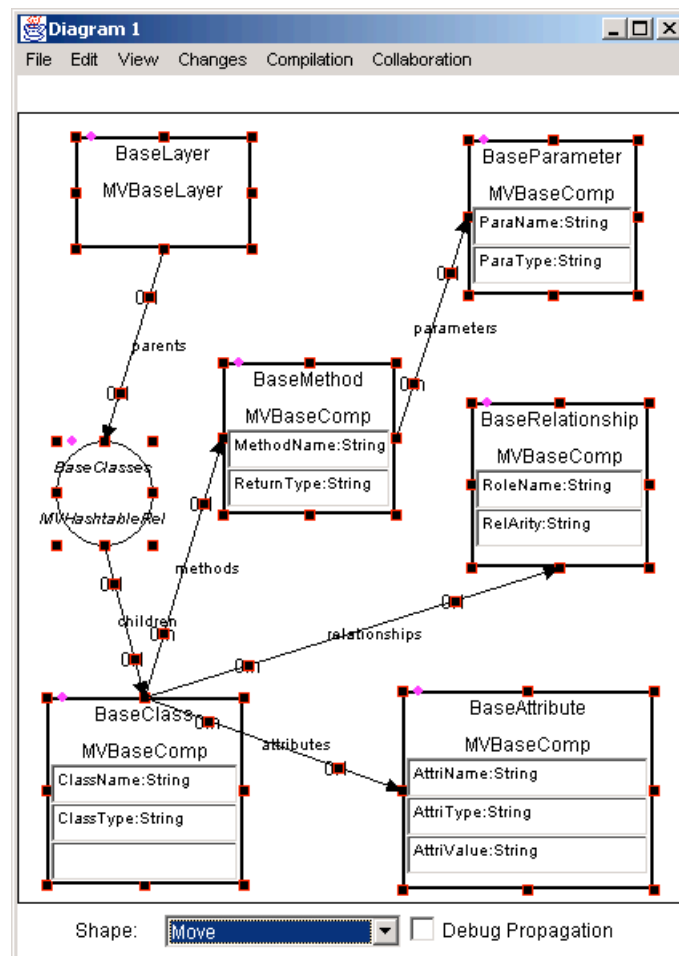


Figure 5.6: Creating base components using JComposer

2. *Defining view layer components.* View layer work as a bridge to connects presentation layer and base layer. Normally every base component has a corresponding view layer components. In this step, what we need to do is to specify the view layer components and its attributes. Fig 5.7 shows part of the specification.

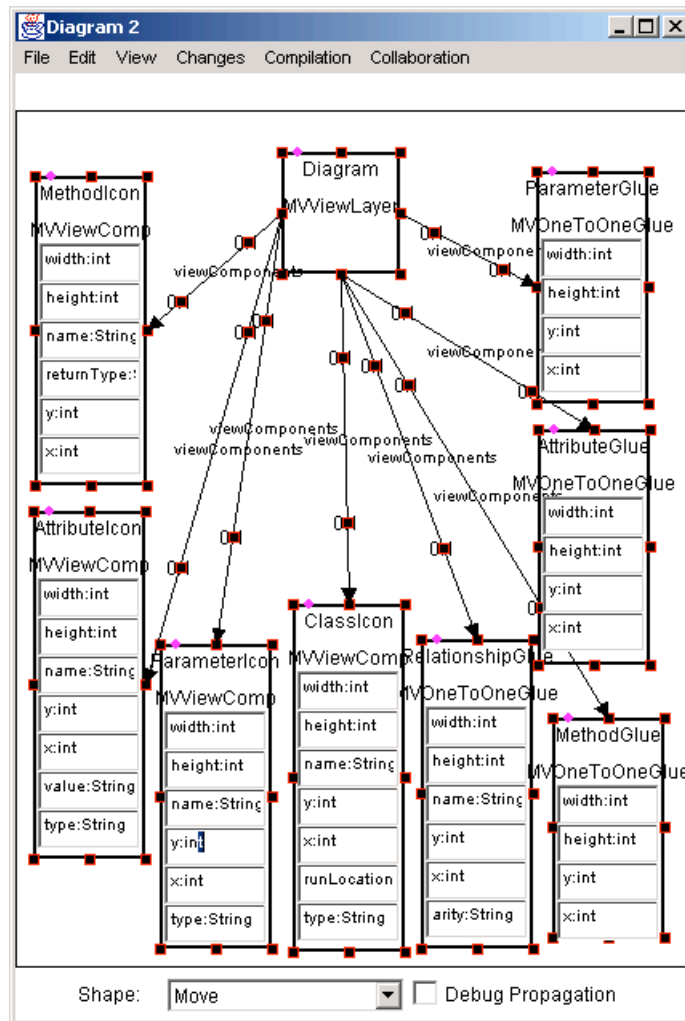


Figure 5.7: Creating view components using JComposer

3. *Defining view/base component mappings.* This step involves defining a group of components pairs. That means to define which view layer component is related to a certain base layer component. Once they are connected, there will be a means to guarantee them to call and response each other together. Fig 5.8 shows some views to define these mappings.

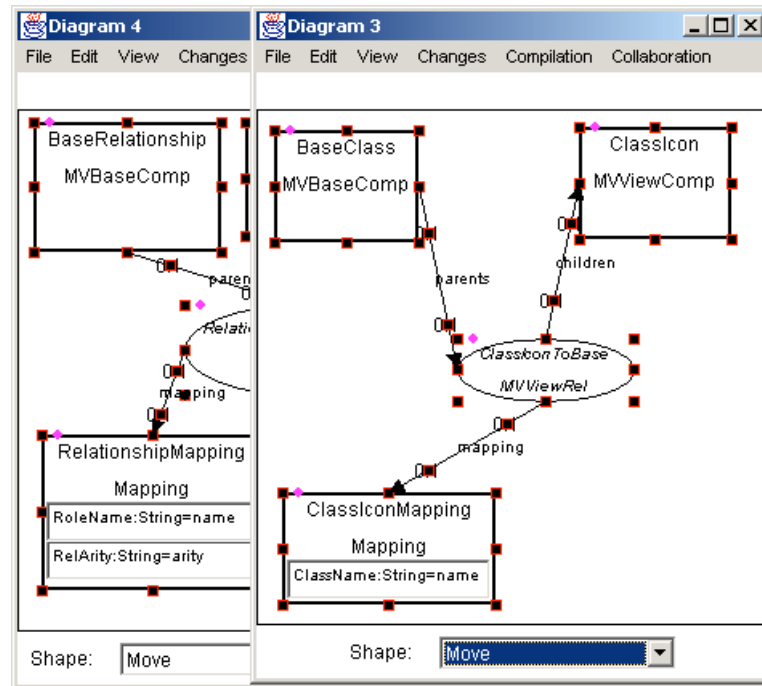


Figure 5.8: Creating view to base mapping using JComposer

Once all these specification properly performed, JComposer would automatically generate source code that establishes the architecture of DISSAG Diagram Editor.

5.5 Using Sun JAXP 1.0 to parse XML file

The Code Generator can take XML files and generate code based on the information retrieved from them. So there must be a component in Code Generator that can parse XML files (See 4.7). It is nearly impossible for us to implement such a parser “by hands” due to the same reason we have mentioned many times before: lack of time. Again we have to search any existed software that can meet our request.

As XML is becoming increasingly popular in the developer community as a tool for passing, manipulating, storing, and organizing information, and as XML is a standardized rather than proprietary format, there are lots of XML parsers. We chose Sun JAXP 1.0 [JAXP 1.0 API] as our XML parser for the following reasons:

1. It is implemented in pure Java, which is the language we chosen for this project, and is supplied by

SUN Microsystem. This means: firstly we will not choose a parser developed in other languages, such as C++, in order to get consistent performance, e.g. platform-independent; secondly, even there are still many other XML parsers developed in JAVA language, but as JAVA itself is a trademark of SUN Microsystem, if there is no sufficient proof, such as benchmark reports, to show that there are much better parsers, we will choose intuitively JAXP of SUN Microsystem as our XML parser.

2. It supports both DOM and SAX. Some parsers, e.g. XP, support SAX only. In chapter 5, we have described why we decided to choose COM API and how our Code Generator tool architecture looks like based on the decision. So supporting DOM becomes the essential standard for us to choose XML parser.
3. It is free. Many other parsers are not.
4. DISSAG is a prototype rather than commercial software. So when we choose XML parser for our Code Generator tool, the first question we care is “Can it satisfy our requirements”, but not “Is it the best one that satisfies our requirements”. JAXP meets our request very well, so we choose it even there might be some better parsers. Moreover, our Code Generator tool has a component-based architecture. So once we get a proven better parser, we can use it to replace JAXP any time with little effort.

5.6 To realize multiple threads by implementing the interface *Runnable*

A Java application can execute multiple tasks at the same time. Each of these computing tasks is then called a thread. Unlike separate computer applications, multiple threads exist in the same executing environment, or in other words, they have the same context. This means they share the same data variables, access the same procedures, and so on [Budd 1998].

Multiple threads actually make it possible for us to integrate our four sub-tools into DISSAG, a single

application and also makes it possible for a single tool to perform several tasks at the same time.

In Java, threads are created using the class *Thread*. There are two common ways that this class is used. One way is to subclass from *Thread*, and override the method *run()*, which is the method that is executed to perform the task assigned to a thread. But there is a shortage: if the thread class is already inheriting from some other class, it cannot be inheriting from *Thread*, because in Java a class cannot extend from two classes [Budd 1998].

To overcome this barrier, Java provides an alternative technique. Instead of inheriting from *Thread*, the action portion of a thread can be declared as implementing the interface *Runnable*.

When developing DISSAG, we use threads in many functions, such as generating code from DOM, compiling code, testing code, some text edit functions and getting help etc. We create all these threads by implementing the interface *Runnable*.

```
package CodeGen;
import java.io.*;
//get on line help information when using DISSAG
public class GetOnlineHelp implements Runnable {
    //an empty constructor
    public GetOnlineHelp {}
    //to create a process that launches IE browser
    //and view help files from my homepage
    public void run() {
        try{
            Runtime runtime=Runtime.getRuntime();
            Process p = runtime.exec("C:\\Program Files
            \\Internet Explorer\\IEXPLORE.EXE http://
            www.cs.auckland.ac.nz/~nzhu002");
            InputStream in = p.getInputStream();
            int k;
            while((k = in.read()) != -1){
                system.out.print((char)k);
            }
            p.waitFor();
        }
        catch(IOException e){
            e.printStackTrace();
            system.err.println("IOException:"
            +e.getMessage());
        }
        catch(InterruptedException ee){
            system.err.println("InterruptedException"
            +ee.getMessage());
        }
    }
}
```

Figure 5.9: A thread example

Fig 5.9 shows a typical class to use a thread. When a user using DISSAG wants to get more help information, he may click on the “help” button and if he does so, a GetOnLineHelp thread will be created and as the result of executing run() method, a IE browser will be launched and more help information will be presented by showing DISSAG’s homepage.

5.7 implementing test bed

Test Bed is integrated into DISSAG as a tool to debug source code or to compile and run it. Now we take two steps to describe the implementation of Test bed.

1. Implementing Test Bed interface

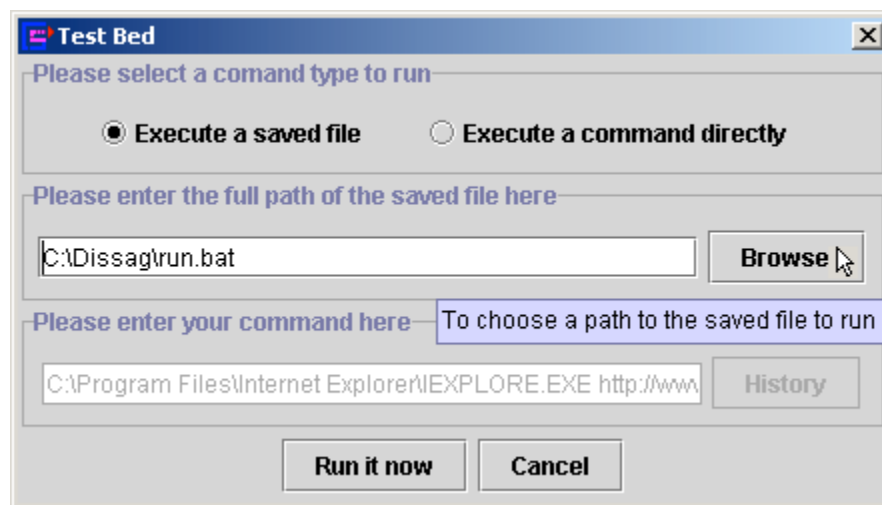


Figure 5.10: Test Bed interface

Fig 5.10 shows the Test Bed interface we developed by Java. The main concern here is how to handle its relationship with DISSAG interface. To make it a part of DISSAG interface, which is implemented by extending JFrame, we implemented it by extending JDialog. It is initiated with DISSAG and is normally invisible. DISSAG users can “switch” to it, or to use it by clicking a button on DISSAG interface and thus making it visible. Another thing we should mention here is that Test Bed itself is bonding with another dialog, CommandHistory, which contains a JList used to hold previous input commands.

Like DISSAG interface, the Test Bed interface is implemented by JFC/SWING. Fig 5.11 shows the main classes related to it.

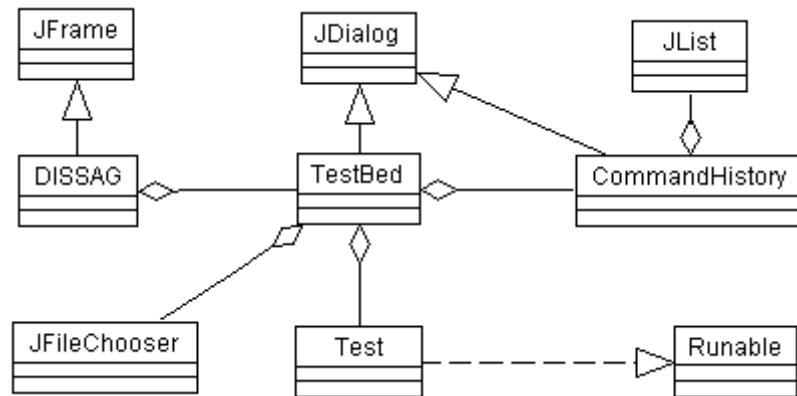


Figure 5.11 Classes to implement Test Bed interface

2. Implementing Test Bed underlying mechanism

When The “RunItNow” button is clicked, the command in the command text field will be executed. That means a new thread will be generated to execute that command (Se Fig 5.9). In prevouse section of this chapter, we have described how threads are used in DISSAG implementation. Now our interest is how to execute those commands.

Test Bed is mainly used to execute those JDK commands, such as javac, java, javadoc and appletviewer etc. That means DISAG (mainly Test Bed) will be able to activate another application from DISSAG. Java enables us to do that. Every Java application has a single instance of class *Runtime* that allows the application to interface with the environment in which the application is running. The current *runtime* can be obtained from the *getRuntime* method. The *Runtime.exec()* methods create a native process and return an instance of a subclass of *Process* that can be used to control the process and obtain information about it. The class *Process* provides methods for performing input from the process, performing output to the process, waiting for the process to complete, checking the exit status of the process, and destroying (killing) the process.

Fig 5.12 shows the mechanism that test bed working. The actual Java code piece is simple and can be seen under run() method inside the code of getOnLineHelp, which is showed in Fig 5.7.

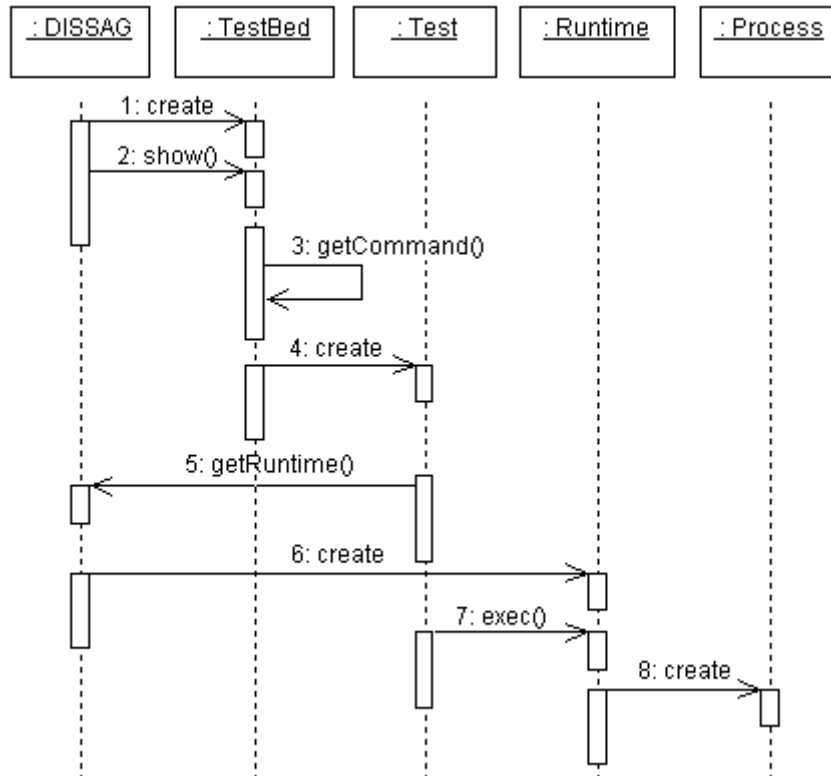


Figure 5.12: Test Bed working Mechanism

5.8 Summary

In this chapter we have discussed some practical issues with the DISSAG implementation. We have introduced why we chose JAVA as the programming language to build DISSAG and how JAVA is used to implement DISSAG interface, to implement multithreads etc. We have also introduced why and how to use the metaCASE tool JCompose to generate the original code for our tool Diagram Editor and to use SUN JAXP as the XML parser in our tool Code Generator.

Chapter 6

Video library: an example

6.1 Introduction

In previous chapters, we have described the desired functions for DISSAG, and have introduced how we analyze, design and implement DISSAG to meet those function requirements. In this chapter, we will illustrate the utility and the power of DISSAG by an example—implementing a Video Library system.

Of course, there will be much more activities involved in the actual implementation, but we will concentrate on the programming portion only.

6.2 Scenario

A video library possesses a certain number of videos. Each video has its own ID and name. Each video has also been assigned a category code and a rating grad. The customer of the video library, who has given the information of his/her name, age, address and phone number when he/she registered, is assigned an ID by the Library and thus can rent the videos home. The staff of the library, who have also been assigned an ID, will handle the rentals. Each rental will be recorded with a customer ID, a video ID, an issuing date and can be specified as returned or not.

To make the library convenient for customers, there will be facilities inside or outside library to allow customers to check the video list and their own rental records. The library has means to maintain customers,

including finding existed customer, adding new customer, updating customer information, deleting a customer who will no longer keep a business relationship with the library. Of course, the library also has the means to maintain staff and videos just like that for customers.

6.3 Requirements

The key requirements are listed below:

- A database to store information with all staff, customers, videos and rentals.
- An interface for customer to get video information.
- An interface for customer to get rental information of his or her own.
- An interface to maintain customers.
- An interface to maintain staff.
- An interface to maintain videos.
- An interface to maintain rentals.

6.4 The steps to implement video library

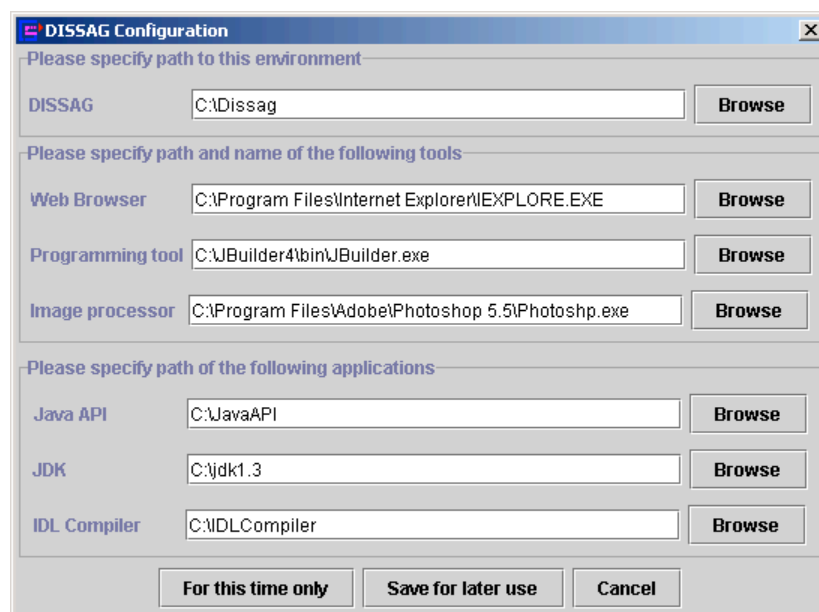


Figure 6.1: DISSAG configuration dialog

6.4.1 Preparation

Preparation includes running DISSAG and configuring it. When DISSAG is running, click the “Configure DISSAG” menu item under menu “Tools”, a configuration dialog will appear (See Fig 6.1). All items except the first one in this dialog are used to integrate the third part tools into DISSAG to enable its related functions. “Web Browser” is used to get information such as online help from the web. “Programming tool” such as JBuilder (using it as a textual programming tool only) will be integrated into DISSAG as an alternative to the “Text Editor” tool in DISSAG. “Image processor” is added when there is a need to processing any images that will be used in a user interface. “Java API” is a very important reference when there is lots of coding job involved in further implementation phase. “JDK” and “IDL Compiler” are integrated so that DISSAG can be used to debug, compile and run the implementation code.

6.4.2 Specification phase

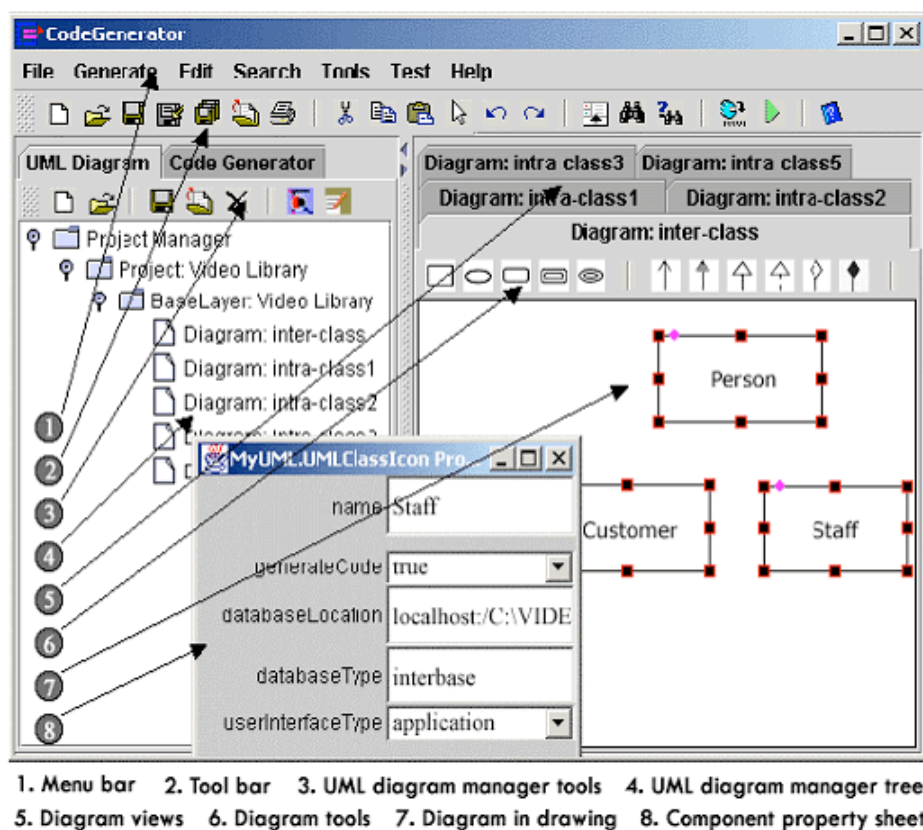


Figure 6.2: Diagram Editor in working with the example

Clicking on the “UML Diagram” tab of the tabbed pane on the left side of the DISSAG interface will start the Diagram Editor tool. Fig 6.2 shows the Diagram Editor in working with this example.

Under the “UML Diagram” tab there is a tool bar containing icons for tools maintaining the project manager tree. Using them, project names, base layer names and diagram names can be added into the manager tree as tree nodes which can be added and deleted easily. The diagram nodes are associated with the diagram editor windows on the right side which have been organized into tabbed panes. Each of the windows will be used to model a part of the system and will be a view to the underlying model.

In Video Library example, we create a project “Video Library”, a base layer “Video Library” under the “Video Library” project and then a set of views called “inter-class x” or “intra-class x”. In the view named “inter-classes x”, we modeled the inter-classes relations. Each class icon can be notated with a properties sheet. In Fig 6.2, we have showed a typical properties sheet for class icons. Fig 6.3 shows the interclass relationships.

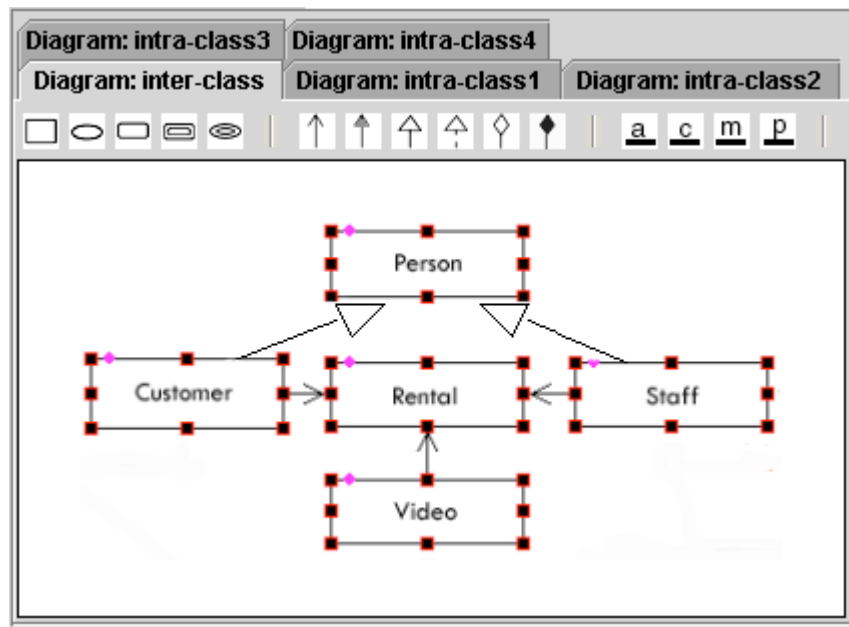


Figure 6.3: Inter-classes relations of Video Library

In this figure, we only showed the most important part of the inter-classes relationships. We have Rental as an association class that related to Customer, Video and Staff. But we neglected the relations such as that

between customer and video because these relations are not necessary for the purpose of generating code for user-desired functions.

One of the merits of multi-views is that we can use as many as views to model our system. As a result, each view can be used to simply model a small portion of the whole model. We use a separated view to model intra-class relations inside each of the classes. This allows each attribute or method to have its own icon and its own properties sheet so that it can be specified more accurately and in more detail.

Fig 6.4 shows the intra-class relationship of the Rental class. We didn't show the attributes for customer ID, video ID and staff ID because those will be automatically added from the inter-classes relationships. We didn't show the parameters for method *newRental* and *findRental* because in DISSAG there will be a mechanism to handle methods named with "new", "find" and "delete", normally, if there is no parameters specified in the diagram for these methods, the premier keys of the class entity will be added in as parameters.

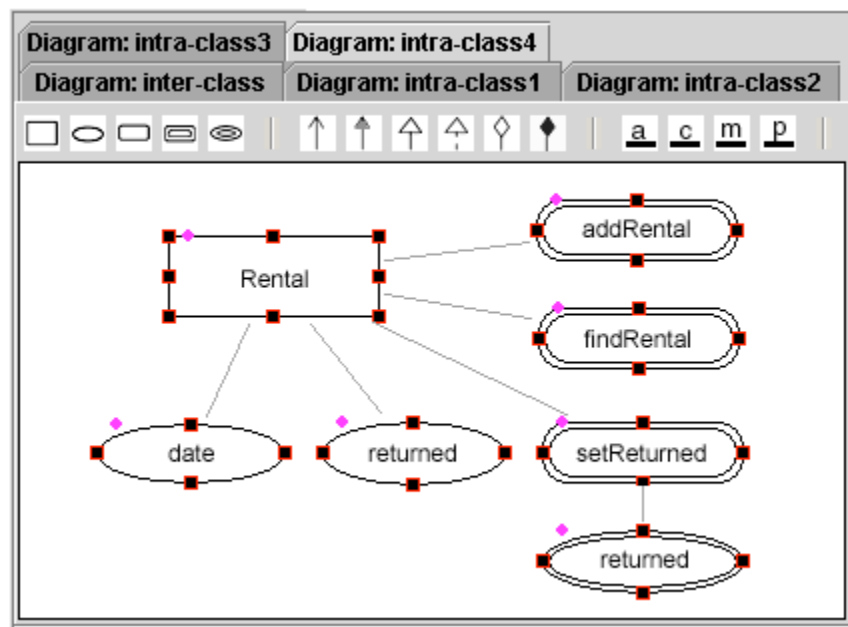


Figure 6.4: Intra-class relationships of Rental class

As we have mentioned in the requirements of DISSAG, multi-views has a shortage that when the number of views getting big, the user may lose track of the components they have added. So as a complementation, there is a tool called Base Layer Viewer that allows the user quickly check all components they have modeled.

Fig 6.5 shows the tool in modeling Video Library. From which, we can see all information the base layer holds. Or in other words, we can see the model information that will be included into a XML file to generate source code. If there is anything wrong or missing, we can correct it or add it through a view.

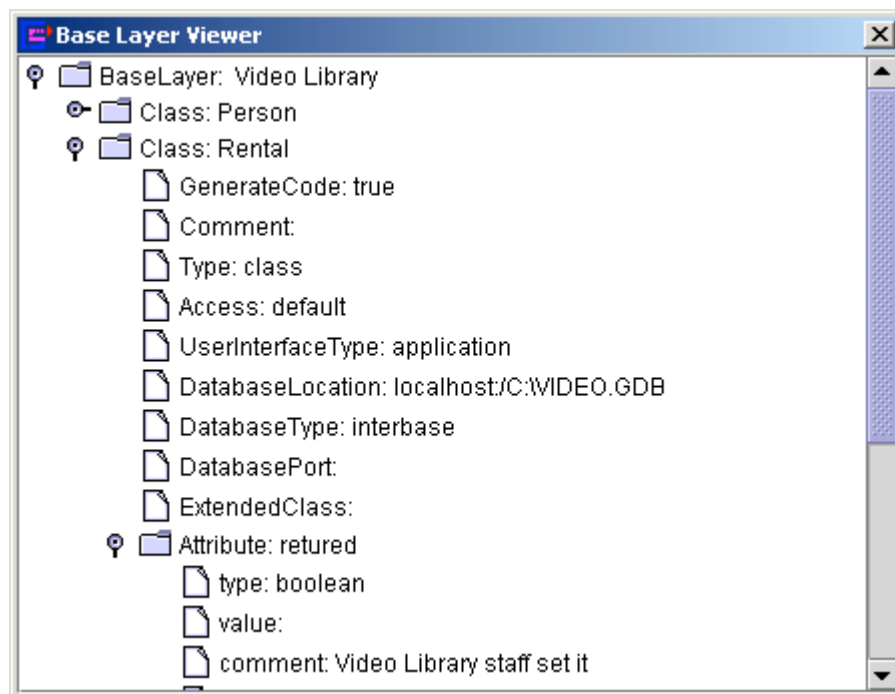


Figure 6.5: Base layer viewer

The annotation to the modeling components can be done at any time during the modeling process. The properties sheets for classes, attributes, methods and parameters are showed in figure 6.6.

When the modeling and notation are ready, Clicking on the XML generation button on the tool bar will generate XML file into the directory specified. In this example, we have generate a XML file called VideoLibrary.xml and stored it in the folder C:\temp.

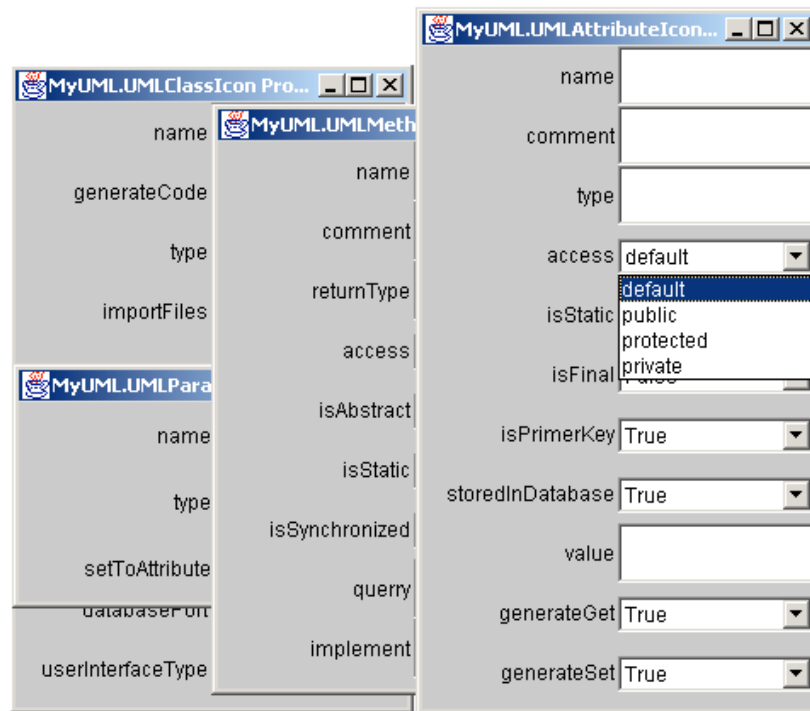


Figure 6.6: Properties sheets for class, attribute, method and parameter

6.4.3 Code Generation phase

Clicking on the “Code generation” tab of the tabbed pane on the left side of the DISSAG interface will start the Code Generation tool.

Fig 6.7 shows the DISSAG interface when the Code Generator tool and Text Editor tool are being used. On the left side there is a code generator manager tree with those related tree maintaining tools. On the right side, there is the area where text editor windows will be showing.

The first step in code generation phase is to load an XML file. With a file chooser of DISSAG a XML file can be chosen and then loaded. The loading will result in parsing the selected XML file into its DOM notation, which can be used later by the code generation logic. In this example, C:\temp\VideoLibrary.xml will be chosen and will be loaded into DISSAG.

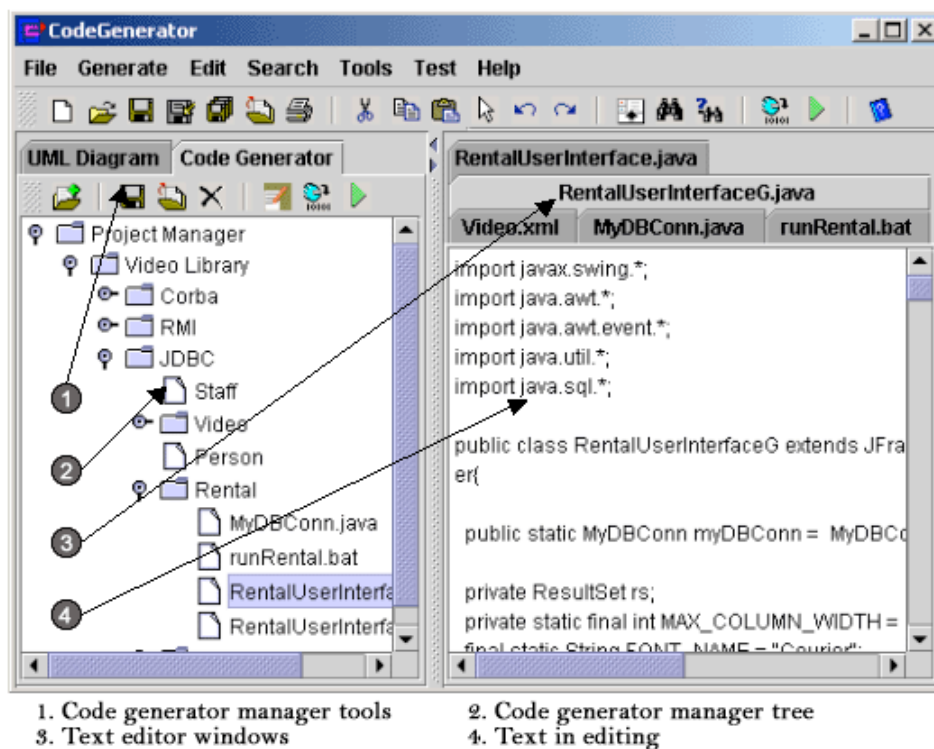
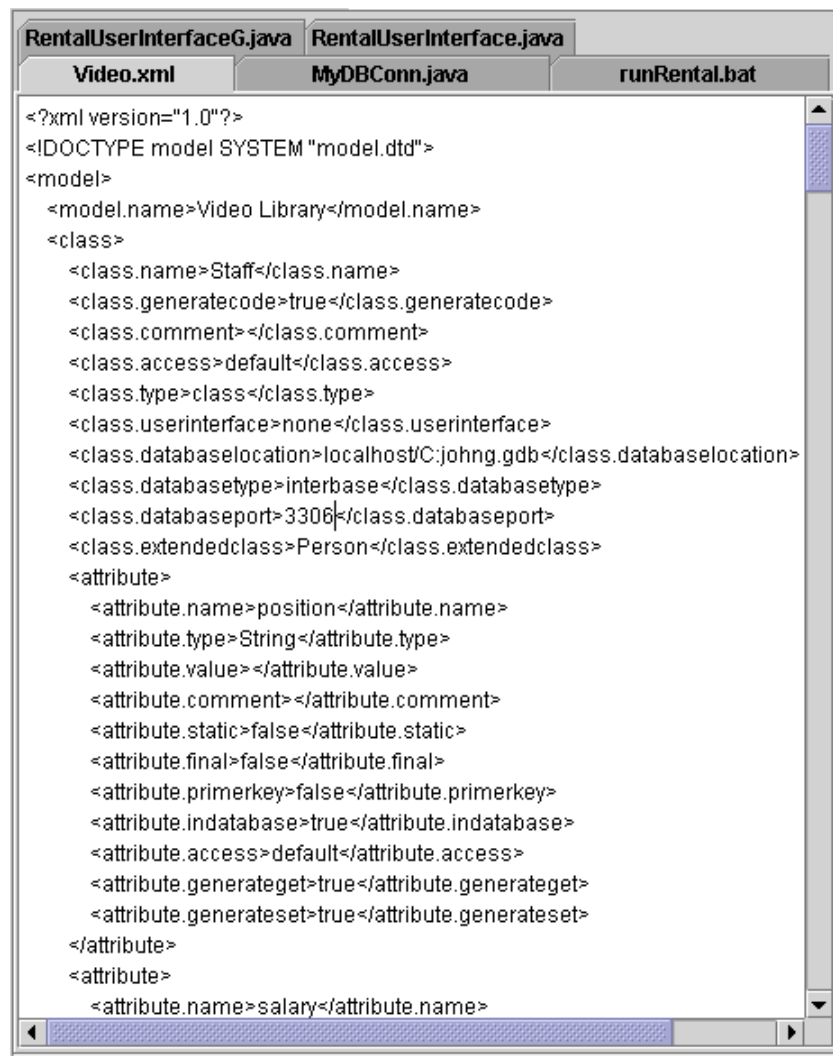


Figure 6.7: DISSG interface when using Code Generator and Text Editor

If the loading and parsing are successful, there will be a code generate manager tree appearing in the manager pane. The project name, Video Library in this case, will be added as a sub-node into the tree under the root node. Clicking this node will add a text editor window to the editors' area showing the XML file related to the project. Fig 6.8 shows a part of VideoLibrary.xml file.

The next step is to select a code style. Currently DISSAG supports generating JDBC-implemented 2-tier system, and CORBA- or RMI-implemented 3-tier system. Clicking a menu item under the "Generation" menu will start a code generation process corresponding to the clicked menu item. In this case, we generate CORBA, RMI and JDBC code in turn.

Each generation will result in adding the method name such as JDBC, CORBA as a sub-node to the manager tree. These sub-nodes will have all class names in this project as its sub-nodes. Under these class name nodes there will be all generated file names as its sub-nodes. Fig 6.7 shows the manager tree.



```

RentalUserInterfaceG.java RentalUserInterface.java
Video.xml MyDBConn.java runRental.bat
<?xml version="1.0"?>
<!DOCTYPE model SYSTEM "model.dtd">
<model>
  <model.name>Video Library</model.name>
  <class>
    <class.name>Staff</class.name>
    <class.generatecode>true</class.generatecode>
    <class.comment></class.comment>
    <class.access>default</class.access>
    <class.type>class</class.type>
    <class.userinterface>none</class.userinterface>
    <class.databaselocation>localhost\C:johng.gdb</class.databaselocation>
    <class.databasetype>interbase</class.databasetype>
    <class.databaseport>3306</class.databaseport>
    <class.extendedclass>Person</class.extendedclass>
    <attribute>
      <attribute.name>position</attribute.name>
      <attribute.type>String</attribute.type>
      <attribute.value></attribute.value>
      <attribute.comment></attribute.comment>
      <attribute.static>>false</attribute.static>
      <attribute.final>>false</attribute.final>
      <attribute.primerkey>>false</attribute.primerkey>
      <attribute.indatabase>>true</attribute.indatabase>
      <attribute.access>default</attribute.access>
      <attribute.generateget>>true</attribute.generateget>
      <attribute.generateset>>true</attribute.generateset>
    </attribute>
    <attribute>
      <attribute.name>salary</attribute.name>


```

Figure 6.8: VideoLibrary.xml showed in a Text Editor window

In this phase, Text Editor tool is used to view the XML file and all those generated files. Clicking the generated file name node will result in adding an editor window to the editors' area inside which the generated file will be showing.

The generated files include not only the JAVA files and the IDL files, but also those files for debugging, compiling and testing the generated code.

Fig 6.9 shows a .bat file used to compile and run the JDBC implementation code for Rental User Interface.



```
RentalUserInterfaceG.java RentalUserInterface.java
Video.xml MyDBConn.java runRental.bat
set path=C:\jdk1.3\bin;%path%;
set classpath=.;\interclient-core.jar;%classpath%;
javac *.java
java RentalUserInterface
pause
```

Figure 6.9: A generated file: runRental.bat

6.4.4 Further implementation phase

DISSAG generated codes are of course not the final implementations for a given DIS. To allow users further implement the given system, specially with those business logics, DISSAG names most of its generated java files with an appendix “G”, and at the same time generate another class extending this generated java class. Then the users can add as more functions as they like into the extending class.



```
RentalUserInterfaceG.java RentalUserInterface.java
Video.xml MyDBConn.java runRental.bat
public class RentalUserInterface extends RentalUserInterfaceG{

    public RentalUserInterface(){

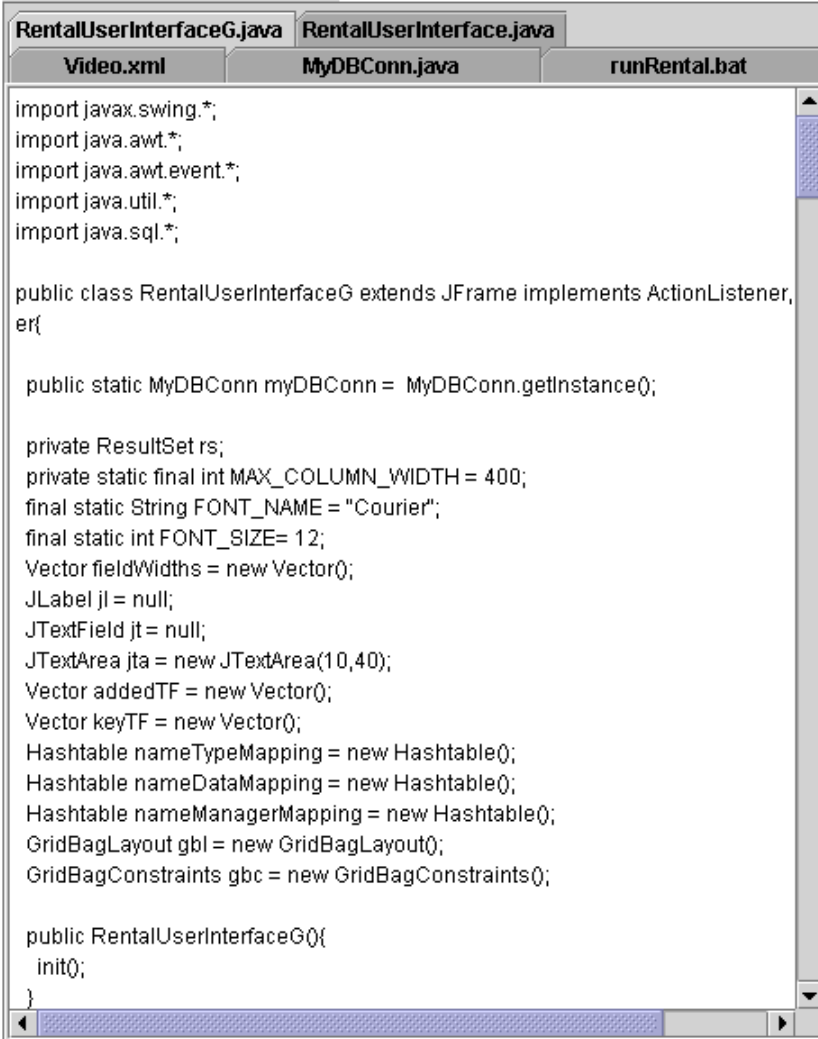
    }

    public static void main (String[] args){
        new RentalUserInterface();
    }

}
```

Figure 6.10: RentalUserInterface.java

Fig 6.10 shows the RentalInterfaceG.java and Fig 6.11 shows the RentalInterface.java file which extending RentalInterfaceG.java.



```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.sql.*;

public class RentalUserInterfaceG extends JFrame implements ActionListener,
er{

    public static MyDBConn myDBConn = MyDBConn.getInstance();

    private ResultSet rs;
    private static final int MAX_COLUMN_WIDTH = 400;
    final static String FONT_NAME = "Courier";
    final static int FONT_SIZE = 12;
    Vector fieldWidths = new Vector();
    JLabel jl = null;
    JTextField jt = null;
    JTextArea jta = new JTextArea(10,40);
    Vector addedTF = new Vector();
    Vector keyTF = new Vector();
    Hashtable nameTypeMapping = new Hashtable();
    Hashtable nameDataMapping = new Hashtable();
    Hashtable nameManagerMapping = new Hashtable();
    GridBagLayout gbl = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();

    public RentalUserInterfaceG(){
        init();
    }
}
```

Figure 6.11: RentalUserInterfaceG.java

In this phase, the tool will be used is the Text Editor tool. For some of the classes involving much coding work, the integrated tool, JBuilder or a tool with the same function will be used.

6.4.5 Test phase

Like an IDE, DISSAG can be used to debug, compile and run the implementation code. Two steps have been

taken to realize this function. The first step has been done with the code generation, which is generating .bat files that will be used to debug, compile and run implementation code. The second step will be using the Test Bed tool to executing the .bat code.

Test Bed tool can also be used as a command prompt to execute commands directly and provide a command history to speed the operation. Fig 6.12 shows the Test Bed tool being used to test part of the generated JDBC code

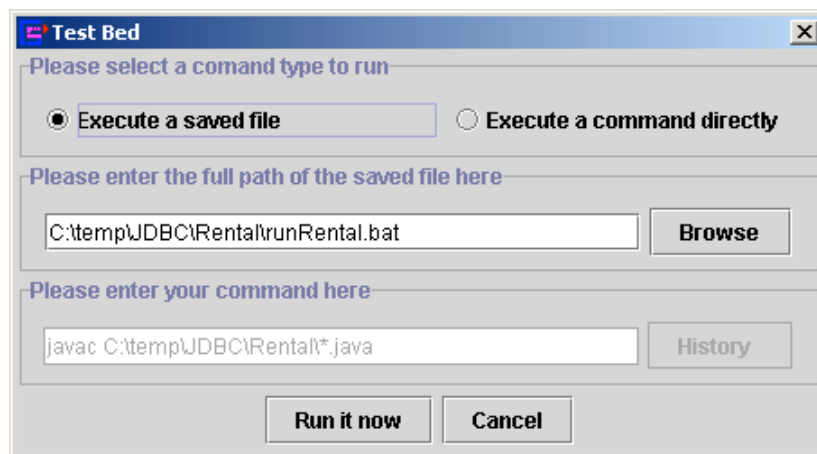


Figure 6.12: Using Test Bed tool

6.5 Summary

Using Video Library as an example, we have showed the whole developing cycle for a DIS with the use of DISSAG. This cycle includes specification, code generation, further implementation and testing phases. DISSAG has sub tools for each of these phases. DISSAG can also integrate some third part tools and as a result, it is getting more powerful. The success of implementing Video Library system proves that DISSAG is a very attractive tool.

Chapter 7

Usability evaluation and future work

7.1 Introduction

This chapter consists of two parts. The first part presents a usability evaluation of DISSAG. In this part, we will introduce our way to evaluate the usability of DISSAG, and then make comments on DISSAG by integrating results of the survey we took during the evaluation and our own analysis. The second part outlines the areas that, we think, are the most potential for future work to continue this research. At the end of this chapter, there is a summary to both of these two parts.

7.2 Usability evaluation

7.2.1 About usability evaluation

Usability means the quality of a system that enables it to be easily understood and conveniently employed by a user. Technically, usability can be defined as the extent to which people can use a system to achieve their goals with effectiveness, efficiency and satisfaction in a specified context and environment [Lindgaard, 1994].

The goal of DISSAG is to provide its users a tool which integrated modeling tool, code generator and IDE together. That means users can perform almost all DIS software development tasks using DISSAG only, and moreover, those tasks will be much simpler because the main job will be done automatically by DISSAG.

Therefore, the evaluation to the usability of DISSAG can be specified with the following concrete questions:

- Does DISSAG really support all DIS developing activities?
- Is the development cycle using DISSAG really feasible?
- Is the DISSAG meta-model appropriate for the purpose of modeling DIS and generating DIS code?
- Is the way DISSAG is used to model and specify DIS appropriate and efficient?
- Is the code generated by DISSAG correct, readable, easy to understand, and easy to extend?
- Has the generated code really set up the structure of the solution and formed a sound base that further implementation can be rely on.
- Is DISSAG reliable, consistent, user-friendly and easy to maintain and upgrade?

All these questions are listed in a questionnaire and the evaluation to DISSAG is taken in the form of a survey during which the questionnaires are performed.

7.2.2 The evaluation method -- survey

The development process of DISSAG is an interactive process. During the development cycle, the evaluation has been taken many times in casual ways. The feedbacks of those evaluations have been directly adopted into the design of DISSAG. The current profile and functions of DISSAG are much different from those appearing in the original design of DISSAG. Therefore evaluation is very useful and necessary.

But the formal evaluation for DISSAG has been taken just once. Because the evaluation is actually a very time-consuming job. As DISSAG is a really big tool, it takes time to learn how to use it properly; and to evaluate DISSAG properly, the best way is using it to develop a real DIS, which is undoubtedly a hard job.

The best targets of the survey should be the potential users of DISSAG. Therefore we invited five people to take part in our formal survey. Three of them are students in Auckland University who studied the paper 415.335. The other two have commercial programming experience.

They were asked to individually implement part of the Video Library system, which we have introduced in last chapter. To save their time, we accompanied them and provided them technical support to the extent that a help file could provide. We discussed the issues we concerned face to face during the evaluation and they briefly wrote their main comments and answered the questions on the survey question sheets.

7.2.3 Issues with evaluating DISSAG

Though we have made our best efforts to add as many useful functions to DISSAG as we could in the given limited time for this thesis, DISSAG, in fact, just as this thesis title states, is a prototype rather than mature business software. Therefore the evaluation of DISSAG should be concentrated on the key functions rather than sophisticated details. In other words, to properly evaluate DISSAG, we should pay more attention to questions like “what functions are essential for DISSAG and has DISSAG really provided them or is it really feasible to realize these functions under the current architecture and design” etc but not to those like “how perfect DISSAG has provided these functions?” etc.

For example, the diagram editor of DISSAG uses *java.awt.Graphic* to draw those modeling components in a canvas. As the limitation of the Graphic class, the figures look not so good as expected. But this does not affect using the diagram editor to properly model a DIS. And we know that the much better visual effect can be achieved under the current design and architecture by using Java2D classes to replace those corresponding classes in *java.awt* package providing we have enough time and it is necessary to do so.

Another example is related to the test bed. By our design, the test bed works in this way: along with the generation of DIS implementation code, the files to debug, compile and run the implementation code are generated also. In testing phase, the Test Bed tool can allow users to browse and run those files so as to test the code. It seems that the test bed is somewhat awkward. But in fact there is not any technical problem to change the test bed into its business-software-like form. We may hide the building and deploying files from the users and replace the test bed interface by adding a menu to DISSAG main interface with dynamic changing menu items by clicking which the corresponding file or a piece of corresponding instruction will be executed.

These sort of examples can be seen everywhere in DISSAG. They add difficulties to the evaluation of DISSAG as a prototype. Two things thus had to be done: 1) to carefully design the questions to ensure the answers of the survey to focus on what we want, 2) even if we had done 1) properly, there were still some answers comparing DISSAG to those business tool such as Rational Rose and JBuilder, so we have to carefully analysis the answer and extricate the useful part.

7.2.4 Evaluation Results

All of our survey participants have used DISSAG themselves. As we always accompanied them during the whole evaluation, we observed that there was not any problem for them to use DISSAG. Almost all of them felt no difficulty in finding the right tools to use and no one got lost in any phase.

They were asked to implement part of the Video Library using DISSAG. All of them have carefully done the modeling and code generating. They then checked generated code very carefully and compared the code with the code example. To extend the generated code they have done further implementation using the text Editor tool and testing tool we provided in DISSAG.

Their general comments on DISSAG were very good. All of them admitted that they could finish the whole development cycle inside DISSAG, and it is very convenient to use and the development is very much faster. Than that using tools like Delphi, JBuilder etc.

As to the generated code, they had comments like:

- The generated code had actually set up a good solution skeleton.
- The generated code had provided some detailed code pieces related to the deployed middleware.
- The code is understandable and easy to extended

Of course, there were lots of comments pointed out the weaknesses of DISSAG. In fact, most of them thought their roles as “mistakes-pickers” and immersed in their work. Although some comments obviously based on

treating DISSAG as business software, the others were very helpful. Some of the most valuable feedbacks are described below and we have made comments on them.

1. Lack of user interface composer.

Many survey participants noticed this problem as soon as they understood the components of DISSAG and their functions because almost all of them were familiar with at least one IDE with a GUI composer.

In fact, we knew this problem at the beginning of our project to develop DISSAG, and we actually took it as an advantage of DISSAG. Because we believed that by well designing the modeling tool and the code generation logic, DISSAG would be able to automatically generate GUI code.

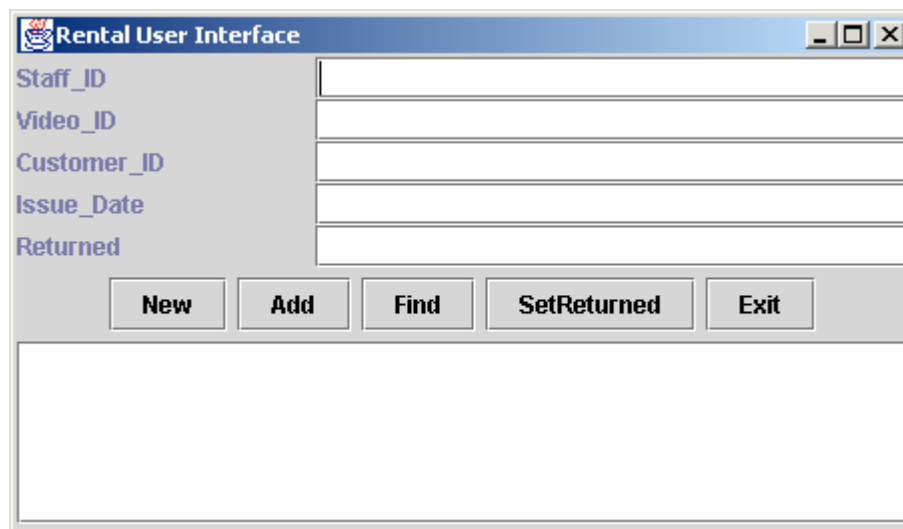


Figure 7.1: The generated Rental user interface

Fig 7.1 shows the interface generated by DISSAG without any further implementation on the code for GUI. DISSAG can generate GUI code for accepting inputs (normally using `JTextField`), for triggering methods (normally using `JButton`) and for showing outputs (normally using `JTextArea` and also those `JTextField` used for input). “But users always like to build interfaces with their own flavor”. So our GUI generating mechanism is not a perfect solution to the practical needs.

We include redesign of GUI generation mechanisms into our future work.

2. No server tier implementation

Some of attentive survey participants found from the DISSAG-generated code that our generated solutions were based on a very simple server tier. In other words, in the generated system, the server tier is nothing but a pure data store.

In chapter 2, we have mentioned that some business logic can be nested in the server side to achieve certain desired effects such as decreasing network traffic. DISSAG now neglects the important step in implementing DIS, to distribute workload among tiers. As a result, all business logics are nested in the client tier in 2-tier architecture and in middle tier in 3-tier architectures.

Solving this problem is included into our future work.

3. Supporting only one kind of UML-like diagram.

Although there can be as many views we can create to model a system as we want, there is only one kind of diagram supported by DISSAG, the class diagram. To model a system more accurate and more efficient, “we need more kinds of diagrams”.

For DISSAG to support more kinds of UML-like diagrams, more kinds of elements will be added into DISSAG model, a set of corresponding notations will be created for each kind of the new element, and how the operations to the diagrams will be reflected in the underlying model will be defined. The mechanism remains the same as that for class diagrams. The actual implementation will be just the same as introduced for build the current Diagram Editor.

We include this into our future work.

7.3 Further Work

1. *Creating a new sub-tool called GUI Composer.*

DISSAG now can automatically generate simple user interface and DISSAG users (DIS software developers) can modify the code to achieve more beautiful, more professional out-looking. But it is not convenient. So a Graphical User Interface (GUI) Composer is really necessary. Using this tool, the DIS software developers can easily compose more complicated user interfaces with their own flavor. But this tool will have a significant feature that makes it different from those traditional GUI development tools, such as that one in JBuilder.

It will not generate any code; instead, it will be used as a special “view” to specify the system. That means, just like Diagram Editor that transfers all specified information from its “views” (class diagrams or some other diagrams) to the underlying model, it will pass all GUI information to the same model (The DISSAG metamodel thus needs corresponding modification to meet this new requirement). Therefore in the code generation stage, the Code Generator will be able to generate the desired user interface code.

The traditional graphical development tools can generate exactly user-designed interfaces, but can do nothing with the logic behind them. With the GUI Composer, DISSAG will be able to generate automatically not only the GUIs, but also the whole lot of logic behind them.

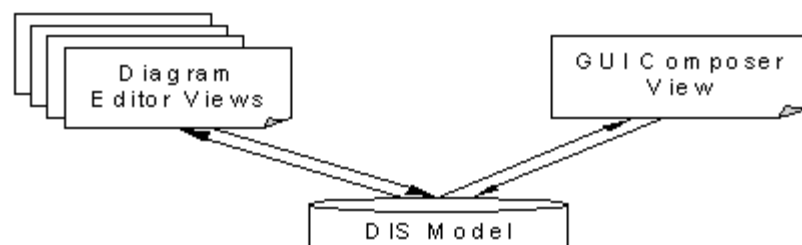


Figure 7.2: GUI composer as a view

2. *Extending DISSAG metamodel to support more UML diagrams, such as deploying diagram and sequence diagram etc.*

Now DISSAG metamodel supports class diagram only. When modeling DIS, sometimes, it is not convenient, or even not sufficient. A different diagram will provide a different kind of views to model DIS and may be good at modeling certain particular features of DIS. Thus when multiple kinds of UML diagrams are used, they will complement each other and provide us more suitable and more powerful means to specify DIS. As a result, the modeling would be more accurate and more efficient and the generated code would be more in detail.

3. *Improving code generation logic to support generating code which distributes workload among tiers.*

We have covered this topic in last section. We know that the actual DIS often distributes their business logic into different tiers so as to distribute runtime workload. Therefore DISSAG's code generation logic should support generating code which separates and distributes business logic. A possible solution will be extending diagram editor to support deploying diagram and allow users specify function units running on a machine.

4. *Generating higher quality implementation source code*

Once the generated code can satisfy the basic functional requirements to implement a DIS, our vision will immediately shift to how to generate high quality code, in other words, how to satisfy those non-functional requirements. One of them is speed. For a particular DIS, there are many factors contributing to the speed. In programming scope, the significant factor that can affect the speed is using cache at client and/or server side. A proper cache can dramatically reduce the network loads and the frequency of using database thus short the response time for a client query. Another one is security. Needless to say how important the security is to the DIS. Therefore for generating higher quality implementation code, DISSAG should be capable of generating those cache or security code pieces.

5. *Extending DISSAG into an environment using multiple languages*

Currently DISSAG can only be used to develop DIS software using JAVA. In fact, DISSAG sub-tools like Diagram Editor, Text Editor and Test Bed are suitable for any language. The bottleneck is Code Generator tool as it can only generate JAVA code now. Therefore Extending DISSAG into an environment using multiple languages is actually extending Code Generator to enable it to generate source code with multiple languages, such as C++. DISSAG would be more flexible and powerful after the extension.

6. *Extending DISSAG into a global-accessible groupware*

One of the objectives to develop DISSAG is to enable DIS software developers to perform most of the developing tasks within a single environment or using a single tool. But in real life, the development of software is usually divided into some stages such as design, implement and test, and the tasks from each stage are allocated to different people. Even the use of DISSAG would actually change the traditional development cycle and reduce people involved, it would not be strange that there would still be more than one person, who may be in different companies, different cities or even different nations, working together for a DIS project. Therefore it is necessary to extend DISSAG into a global-accessible groupware.

7. *Generating code using other middleware technologies.*

As a prototype, DISSAG now can generate source code for DIS using CORBA, RMI and JDBC. It can be easily extended to generate source code using more middleware technologies such as DCOM.

8. *Adding an information panel to Diagram Editor.*

This panel will be used to show all information, related to the current working project and the current

selected component, in separate tabbed panes. Each pane again contains some tabbed panels.

In the pane that shows currently working project information, there will be the following tabbed panes:

- “To Do” list: shows what is the possible next task to be performed in this project at present.
- “Undo” list: shows what operations can be undone at present.
- “Redo” list: shows what operations can be performed again at present.
- Checklist: shows the normal activities for modeling a DIS with DISSAG.
- Component viewer: shows all components and their specified features at present in a tree.

In the pane that shows currently selected component information, there will be the following tabbed panes:

- Property sheet: shows the properties both for the visual presentation and for the underlying model. All properties can be set or modified. Thus it will be the right place not only to view the component information, but also to specify the component.
- Related components: shows the components that have direct relationships with the selected component and the relationships. This would be helpful when the selected component appeared in multiple views.

9. *Using Java 2D to achieve better visual effect*

As we have introduced in evaluation section, some users dislike the appearance of the class diagram DISSAG drawn. The visual components in Graphical Editor are drawn with Java AWT Graphic package. Some graphical elements such as broken lines and oblique lines appear very ugly. Adopting Java 2D to draw those visual components will get better visual effects.

10. *Re-implement some of BBW components.*

Some of DISAG visual components are extending BBW components. As BBW is currently implemented using JAVA AWT package, some components cannot get along with Java Swing components very well. So it is necessary to re-implement some of BBW components with JAVA Swing.

7.4 Summary

The evaluation of DISSAG shows that it is an effective, efficient, and flexible tool. It combines a modeling tool, a code generator and an IDE together seamlessly and provides users with an easy, fast and reliable way to develop DIS software.

However, it is just a prototype. Therefore, though it appears very attractive with its powerful functions, it has currently many weaknesses. Future work will be concentrated on how to overcome these weaknesses and how to make DISSAG perfect in the view of functions. It includes adding a new tool—an interface composer, extending the modeling tool to support more UML-style diagrams, improving code generation logic to generate more accurate, more reasonable code, turning it into a groupware and supporting multiple languages.

Chapter 8

Conclusion

8.1 Introduction

There are two parts in this concluding chapter. In the first part, we will briefly review the work we have done as part of this research. Then in the second part, we will present a summary of the main contributions of this thesis to the related research field.

8.2 What we have done?

Our objective is to develop a DIS software development environment supporting all development activities and featuring automatically performing design tasks and automatically generating much detailed implementation source code.

Therefore the research we have done has two parallel threads went to the area of the software development environments or software developing tools, and to DIS, distributed information system applications.

The former has shown the following facts:

- There is no single tool can be used as a DIS software development environment which can perform the main developing tasks such as analysis, design, implement and test.
- Existing CASE tools or software development environments cannot be simply combined together seamlessly as an environment to support all DIS software developing activities.

- There are no existing tools that can automatically perform some basic design tasks and automatically generate code for DIS.

The latter has shown the following facts:

- DIS has relatively stable architectures and has mature middleware technologies to support their implementation.
- The design and implementation of significant parts of DIS software is possible to do automatically.

All of above facts have actually indicated the necessity and feasibility of the tool DISSAG. They have also implied the two main requirements of DISSAG. One is providing a DIS software development environment that supports most of the DIS software developing tasks; another is automatically performing design task and generating implementation source code.

To satisfy the former, DISSAG should consist of tools that support typical DIS software development activities. Thus UML diagram editor for analysis and design, textual programming tool for implementation and test tool for software test are inevitable parts of DISSAG.

To satisfy the latter, DISSAG should have a code generation tool that can automatically generate code based on certain information. This tool must have well designed input and output so as to combine the other tools properly.

All other requirements that help achieving above goals are added to form a set of detailed requirements that in fact indicate all basic functions of DISSAG.

The design and implementation of DISSAG are the most important parts of our research. We have designed a component-based architecture for DISSAG and its sub-tools. We have designed DISSAG meta-model that can be used to effectively and efficiently model and specify a DIS. We have also designed input and output for all sub-tools, for example the output of Diagram Editor is XML files that contains all information from the DIS

model and can be taken as input by Code Generator, so that they can be contained in a single environment and be used as a single tool. This also makes it possible to integrate the third part tools whenever needed.

We use JAVA as our implementation language. Its abilities to support multi-threads and to invoke another application at runtime make it possible to easily integrate the third part tools and perform multiply tasks within DISSAG environment. Its JFC/Swing packages enable us to construct the complicated user interface of DISSAG. We adopt JComposer to establish the JViews component-based architecture for our Diagram Editor tool and adopt Sun JAXP 1.0 as our XML parser in our Code Generator tool to parse XML file into DOM from which the source code will be generated.

Finished as a prototype, DISSAG source code contains over 200 classes. Nearly all of the requirements have been satisfied. As a DIS development environment, it provides all necessary tools to enable its users to develop high quality DIS software with an effective, efficient and convenient way.

8.3 What are the main contributions of this research?

1. Suggesting a new approach to develop software, especially DIS software.

Traditionally, a software development cycle has four man phases: specification, design, implementation and test. We found that for some particular domain, such as DIS, the software development cycle can be: specification, code generation, further implementation and test. But there are some constraints for this new cycle:

- In this particular domain, software has relatively stable architectures and there are common source codes to implement those architectures and the codes have been proven to be good solutions.
- The architectures and some source code details can be clearly specified with some predefined system attributes. That means once the attributes have been given, the software architectures and some code details can be generated.
- All attributes can be visually set by means of various diagrams and annotations.

2. *Introducing a new-concept software development environment.*

As a new tool, it has features of the software environments: it supports textual programming and can be used to debug, compile and run the finished source code; it has features of the modeling tool: it supports UML diagrams; it also has its own features: it can automatically generate much detailed source code. All these features make it different from the existed tools and also make it a good reference for the new generation software environments.

3. *Suggesting a good approach to specify DIS.*

The precondition for a code generator tool to generate a detailed and desired source code is to specify the target system well. In DISSAG, we provide a very good approach to specify DIS. It includes:

- Creating DISSAG metamodel. It adopts parts of UML metamodel so that it supports modeling DIS by UML-like class diagram; it has its own notations so that it supports specify the components in class diagram.
- Adopting JViews component-base architecture. This architecture supports visually modeling and specifying DIS.
- Output modeling results with XML files. This makes it possible to separate code generation logic from specification logic.

4. *Handling multiply tools within a single environment*

DISSAG itself contains four sub-tools and some suitable third party tools can be integrated for certain purposes. We provide a feasible way to make them work together as a single tool for the aim of developing DIS software.

5. *Suggesting an approach to generate DIS source code*

In DISSAG, we have designed an algorithm to generate DIS source code from the DOM that is parsed from the XML file containing DIS information.

6. *Handling multiply tasks in a single environment at the same time*

There is no doubt that a good development environment should be capable of performing multiply tasks at the same time. In DISSAG, we bring forward solutions to perform multiply tasks.

Appendix A

Model.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!-- author: Nianping Zhu -->
```

```
<!-- email: nZhu002@ec.auckland.ac.nz -->
```

```
<!-- This DTD file is specially created for DISSAG -->
```

```
<!--
```

```
*****
```

```
model.dtd
```

```
version: 1.0
```

```
Copyright (c) 2001 by Nianping Zhu
```

```
Description:
```

XMLforDISSAGmodel is specially created for DISSAG. DISSAG is a CASE tool which consists of four main parts: Diagram Editor, Code Generator, Test Editor and Test Bed. Diagram Editor is a annotated component_based UML_style multi_views diagram editor using an underlying modified UML data model, DISSAG metamodel. Code Generator is the code takes the data model and generates implementing code with

specified programming language. The XMLforDISSAG model is designed to efficiently transfer the data in the model between those two parts.

-->

<!-- model: The root element -->

<!ELEMENT model (model.name|class|comment)*>

<!-- model.name -->

<!ELEMENT model.name (#PCDATA)>

<!-- Comment -->

<!ELEMENT comment (#PCDATA)>

<!-- class -->

<!ELEMENT class (class.name|class.generatecode|class.packagename|class.type|class.importfiles|class.userinterface|class.databaselocation|class.databasetype|class.databaseport|class.comment|class.access|class.extendedclass|class.implementedinterface|class.onetomanyrelatedclass|class.onetoonerelatedclass|attribute|constructor|method)*>

<!-- class.name -->

<!ELEMENT class.name (#PCDATA)>

<!-- class.comment -->

<!ELEMENT class.comment (#PCDATA)>

<!-- class.access -->

<!ELEMENT class.access (#PCDATA)>

<!-- class.generatecode -->

<!ELEMENT class.generatecode (#PCDATA)>

<!-- class.packagename -->

<!ELEMENT class.packagename (#PCDATA)>

<!-- class.type -->

<!ELEMENT class.type (#PCDATA)>

<!-- class.importfiles -->

<!ELEMENT class.importfiles (#PCDATA)>

<!-- class.userinterface -->

<!ELEMENT class.userinterface (#PCDATA)>

<!-- class.databaselocation -->

<!ELEMENT class.databaselocation (#PCDATA)>

<!-- class.databasetype -->

<!ELEMENT class.databasetype (#PCDATA)>

<!-- class.databaseport -->

<!ELEMENT class.databaseport (#PCDATA)>

<!-- class.extendedclass -->

<!ELEMENT class.extendedclass (#PCDATA)>

```
<!-- class.implementedinterface -->
```

```
<!ELEMENT class.implementedinterface (#PCDATA)>
```

```
<!-- class.onetoonerelatedclass -->
```

```
<!ELEMENT class.onetoonerelatedclass (#PCDATA)>
```

```
<!-- class.onetomanyrelatedclass -->
```

```
<!ELEMENT class.onetomanyrelatedclass (#PCDATA)>
```

```
<!-- attribute -->
```

```
<!ELEMENT attribute (attribute.name|attribute.type|attribute.value|attribute.comment|attribute.static|attribute.final|attribute.indatabase|attribute.primerkey|attribute.access|attribute.generateget|attribute.generateset)*>
```

```
<!-- constructor -->
```

```
<!ELEMENT constructor (constructor.name|constructor.access|constructor.comment|constructor.implement|parameter)*>
```

```
<!-- method -->
```

```
<!ELEMENT method (method.name|method.comment|method.returntype|method.access|method.synchronized|method.static|method.abstract|method.query|method.implement|parameter)*>
```

```
<!-- parameter -->
```

```
<!ELEMENT parameter (parameter.name|parameter.type|parameter.settoattribute)*>
```

```
<!-- parameter.name -->
```

```
<!ELEMENT parameter.name (#PCDATA)>
```

<!-- parameter.type -->

<!ELEMENT parameter.type (#PCDATA)>

<!-- parameter.settoattribute -->

<!ELEMENT parameter.settoattribute (#PCDATA)>

<!-- attribute.name -->

<!ELEMENT attribute.name (#PCDATA)>

<!-- attribute.type -->

<!ELEMENT attribute.type (#PCDATA)>

<!-- attribute.value -->

<!ELEMENT attribute.value (#PCDATA)>

<!-- attribute.comment -->

<!ELEMENT attribute.comment (#PCDATA)>

<!-- attribute.static -->

<!ELEMENT attribute.static (#PCDATA)>

<!-- attribute.final -->

<!ELEMENT attribute.final (#PCDATA)>

<!-- attribute.indatabase -->

<!ELEMENT attribute.indatabase (#PCDATA)>

<!-- attribute.primerkey -->

<!ELEMENT attribute.primerkey (#PCDATA)>

<!-- attribute.access -->

<!ELEMENT attribute.access (#PCDATA)>

<!-- attribute.generateget -->

<!ELEMENT attribute.generateget (#PCDATA)>

<!-- attribute.generateset -->

<!ELEMENT attribute.generateset (#PCDATA)>

<!-- method.name -->

<!ELEMENT method.name (#PCDATA)>

<!-- method.comment -->

<!ELEMENT method.comment (#PCDATA)>

<!-- method.returntype -->

<!ELEMENT method.returntype (#PCDATA)>

<!-- method.access -->

<!ELEMENT method.access (#PCDATA)>

<!-- method.synchronized -->

<!ELEMENT method.synchronized (#PCDATA)>

<!-- method.static -->

<!ELEMENT method.static (#PCDATA)>

<!-- method.abstract -->

<!ELEMENT method.abstract (#PCDATA)>

<!-- method.query -->

<!ELEMENT method.query (#PCDATA)>

<!-- method.implement -->

<!ELEMENT method.implement (#PCDATA)>

<!-- constructor.name -->

<!ELEMENT constructor.name (#PCDATA)>

<!-- constructor.comment -->

<!ELEMENT constructor.comment (#PCDATA)>

<!-- constructor.implement -->

<!ELEMENT constructor.implement (#PCDATA)>

<!-- constructor.access -->

<!ELEMENT constructor.access (#PCDATA)>

<!-- End of model.dtd -->

Appendix B

DISSAG metamodel

1. Classes for components



2. Classes for relationships



Bibliography

- [Balen 1999] Henry Balen, *Distributed object architectures with CORBA*, Cambridge University Press (1999).
- [Bell *et al* 1992] Doug Bell, Ian Morrey, John Pugh, *Software engineering: a programming approach*, Prentice Hall (1992).
- [Booch *et al* 1998] Grady Booch, James Rumbaugh and Ivar Jacobson, *The unified modeling language user guide*, Addison-Wesley (1998).
- [Booch 1996] Grady Booch, *Object solutions: managing the object-oriented project*, Addison-Wesley Pub. Co. (1996).
- [Booch 1994] Grady Booch, *Object Oriented Analysis and Design With Applications*, Benjamin/Cummings (1994).
- [Booch 1991] Grady Booch, *Object oriented design with applications*, Benjamin/Cummings Pub. Co. (1991).
- [Budd 1998] Timothy A. Budd, *Understanding object-oriented programming with JAVA*, Addison Wesley Longman, Inc (1998).
- [Ferguson *et al* 2000] R.I.Ferguson, N.F.Parrington, P.Dunne, C.Hardy, J.M.Archibald, J.B.Thompson, "MetaMoose -- an object-oriented framework for the construction of CASE tools", *Information and Software Technology* 42 (2), pg 115-128, (2000).
- [Fowler 2000] Martin Fowler, Kendall Scott, *UML distilled: a brief guide to the standard object modeling language*, Addison Wesley (2000).
- [Fuggetta 1993] A. Fuggetta, "A classification of CASE technology", *IEEE Computer* 26 (12), pg 25-38, (1993).
- [Gray *et al* 2000] J.P.Gray, A.Liu, L.Scott, "Issues in software engineering tool construction", *Information and Software Technology* 42 (2), pg 73-77, (2000).

- [Grundy *et al* 2000] J.Grundy, W.Mugridge, J.Hosking, “Constructing component-based software engineering environments: issues and experiences”, *Information and Software Technology* 42 (2), pg 102-114, (2000).
- [Grundy *et al* 1998a] J.C.Grundy, J.G.Hosking, “Serendipity: integrated environment support for process modeling, enactment and work coordination”, *Automated Software Engineering* 5 (1), pg 27-60, (1998).
- [Grundy *et al* 1998b] J.C.Grundy, J.G.Hosking, W.B.Mugridge “Inconsistency management for multiple-view software development environments”, *IEEE Transactions on Software Engineering* 24 (11), pg 960-981, (1998).
- [Gulbransen 2000] David Gulbransen, *The complete idiot's guide to XML*, Prentice Hall (2000).
- [Hamilton 1999] Marc Hamilton, *Software Development: Building Reliable System*, Prentice Hall PTR (1999).
- [Heller 1999] Philip Heller, Simon Roberts, *Java 1.2 developer's handbook*, SYBEX (1999).
- [Java API] JAVA 2 Platform Standard Edition, v1.2.2, API Specification, available online at, <http://java.sun.com/products/jdk/1.2/docs/api/index.html>.
- [JAXP API] Java API for XML Parsing *Release: 1.0*, available online at, <http://java.sun.com/xml/jaxp-1.0.1/docs/api>.
- [JBuilder 2001] All latest information with JBuilder is available from <http://www.borland.com/jbuilder>.
- [Jepson 1997] Brian Jepson, *Java database programming*, Wiley Computer Pub, (1997).
- [Larman 1998] Craig Larman, *Applying UML and patterns: an introduction to object-oriented analysis and design*, Prentice Hall PTR (1998).
- [McDERMID 1990] Donald C. McDermid, *Software Engineering For Information Systems*, Blackwell Scientific Publications (1990).
- [McLaughlin 2000] Brett McLaughlin, *Java and XML*, O'Reilly (2000);
- [MetaEdit+] MetaEditor+, available from <http://ww.metacase.com>.
- [Ferguson *et al* 2000] R.I.Ferguson, N.F.Parrington, P.Dunne, C.Hardy, J.M.Archibald, J.B.Thompson “MetaMoose—an Object-oriented framework for the construction of CASE tools”,

- Information and Software Technology 42 (2), pg 102-114, (2000).
- [Flanagan *et al* 1998] David Flanagan, Jim Farley, William Crawford, Kris Magnusson, *JAVA enterprise in a nutshell*, O'REILLY (1998).
- [Mingers 1997] Edited by John Mingers and Frank Stowell, *Information systems: an emerging discipline*, McGraw-Hill (1997).
- [Mowbray *et al* 1995] Thomas J. Mowbray, Ron Zahavi, *The essential CORBA: systems integration using distributed objects*, John Wiley & Sons, Inc (1995).
- [Mynatt 1990] Barbee Teasley Mynatt, *Software engineering with student project guidance*, Prentice Hall (1990).
- [Orfali *et al* 1998] Robert Orfali, Dan Harkey, *Client/server programming with Java and CORBA*, Wiley Computer Pub (1998).
- [Orfali *et al* 1994] Robert Orfali, Dan Harkey, Jeri Edwards, *Essential client/server survival guide*, Van Nostrand Reinhold (1994).
- [Orfali *et al* 1996] Robert Orfali, Dan Harkey, Jeri Edwards, *Essential distributed objects survival guide*, Wiley Computer Pub (1996).
- [Pomberger *et al* 1996] Gustav Pomberger and Günther Blaschek, translation by Robert Bach, *Object orientation and prototyping in software engineering*, Prentice Hall (1996).
- [Rob *et al* 1997] Peter Rob, Carlos Coronel, *Database System: Design, Implementation, and Management*, Course Technology (1997).
- [Rock-Evans 1998] Rosemary Rock-Evans, *DCOM explained*, Digital Press (1998).
- [Rose Documents] Available from <http://www.rational.com/products/rose/> .
- [Rosen *et al* 1998] Michael Rosen, David Curtis, *Integrating CORBA and COM applications*, John Wiley & Sons (1998).
- [Rubin *et al* 1999] William Rubin, Marshall Brain, *Understanding DCOM*, Prentice Hall (1999).
- [Sigfried 1996] Stefan Sigfried, *Understanding object-oriented software engineering*, IEEE Press, IEEE Computer Society Press (1996).
- [Simon 2001] Solomon H. Simon, *XML*, McGraw-Hill (2001).
- [Slama *et al* 1999] Dirk Slama, Jason Garbis, Perry Russell, *Enterprise CORBA*, Prentice Hall PTR

- (1999).
- [UML 2000] UML Specification v1.3, Object Management Group Document formal/00-03-01, 2000, available from <http://www.omg.org>.
- [Vogel 1998] Andreas Vogel, *JAVA Programming with CORBA*, Wiley Computer Publishing (1998).