DesignIt

Code Generator based on
XML and Code Templates

A dissertation submitted in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in Computer Science and Applications
in the
College of Engineering
The Queen's University of Belfast

by

Daire Kivlehan

15. September 2000

Acknowledgements

I would like to thank Andreas Rausch our chief supervisor and mentor throughout the duration of the project and dissertation preparation.

The completion of this project would not have been possible without the help of the other members of the DesignIt team Karen Lavery and Lucien Hoogendoorn.

I should also mention those who provided the organisation and finance for the project to be undertaken in Munich, The International Liaison Office in The Queens University Of Belfast and the European Mobility Programme.

Finally I thank my parents for all their support an encouragement without which I may not have progressed this far.

Abstract

In this project we developed a system to enable the automatic generation of Java source code describing a Componentware system from a formal specification. This generated code was fully compliable and could be ran using a specially created Runtime Environment. Using this generator it is now possible to provide the core generation needs for the rapid prototyping of a Componentware system.

Contents

1 INTRODUCTION 1

1.1 A Briefing into the Problem 1

1.2 Basic Concepts of Componentware 3

1.3 Benefits of Componentware 4

1.4 Aims and Objectives of DesignIt 4

1.5 Readers Guide for the Paper 5

2 DESIGNIT: OVERVIEW AND BASIC CONCEPTS 6

2.1 The Development Environment 7

2.2 Our Sample Application: Breakplanner System 10

2.3 Component-Oriented Runtime Environment 12

2.4 XML and Template based Code Generator 15

2.5 The High-Level Architecture of the DesignIt Tool 17

3 ARCHITECTURE AND DESIGN OF GENERATOR 18

3.1 User Interface 18

3.2 Generator Package in Detail 21

3.3 Generator 21

3.4 File Generator 22

3.5 JavaGeneratorHelper 24

4 IMPLEMENTATION OF GENERATOR 25

4.1 System Initialisation 25

4.2 FileGenerator 25

4.3 Test Strategy 26

4.4 Task Delegation 27

4.5 BufferedStreams 27

4.6 Iterator 28

5 CONCLUSIONS 29

REFERENCES 31

APPENDICES 33

A. Project Schedule..33

B. Generator Testing 36

1 Introduction

1.1 A Briefing into the Problem

Smaller hardware and growing systems are leading to a new degree of system complexity. An additional source of complexity is introduced in distributed systems by communication. With his increasing complexity the number of errors will increase unless methods and tools to properly design those systems are provided.

On the other hand the human and financial resources become more and more limited in software engineering. For that reasons we need concepts and techniques to reduced the effort of building systems with respect to quality assurances resp. even quality improvements.

Formal based methods and tools [BJ95] enable us increase the quality of software through proving the correctness of a program with respect to a specification. Since proving is a laborious and expensive task, the specification should capture all relevant aspects for verifying safety critical properties. However, the correctness of the initial requirement specification cannot be proved. To validate the adequacy of a specification a simulation and prototyping environment is very helpful. If the simulation is based on code generation, the generated code can be used for prototyping as well or even for the final system itself. If the specification is based on formal methods, simulation, prototyping and code generation can be realized in a straightforward manner [HMR+98].

Another aspect of the possibility to generate the necessary code using formal specification techniques is, that the time taken to produce systems would be greatly reduced. But, in object-orientation, the increasing size and complexity of software systems leads to the production of a huge conglomeration of classes, which are hard to manage and understand. Moreover, they are also time consuming to produce and integrate. So using formal methods and concepts on the one side and generating code and prototyping in an object-oriented environment would be contra-productive. Obviously systems build on the concepts of formal methods require a more advanced way of structuring, describing and developing them. Componentware is a possible approach to solve these problems.

Componentware is concerned with the development of software systems by using components as the essential building blocks. It is not a revolutionary approach but incorporates successful concepts form established paradigms like object-orientation while trying to overcome some of their deficiencies. Although a variety of technical concepts and tools for component-oriented software engineering already exist, the successful model from the building industry – composing a building out of pre-fabricated parts like windows, doors, walls, etc. – was not completely transferred to software development yet. In our opinion this is partly due to the lack of a suitable Componentware methodology. Such a methodology should at least incorporate the following parts:

- A well-defined conceptual framework of Componentware is required as a reliable foundation. It consists of a mathematical *formal system model* which is used to unambiguously express the basic definitions and concepts. Corresponding informal descriptions are useful to illustrate them. The contained definitions and concepts should be as simple as possible, yet sufficiently powerful to capture the essential concepts and development techniques of existing technical component approaches.

- Based on the formal system model, *description techniques* for components are required. They correspond to the building plans of architecture and are necessary for communication with the customer and between the developers. Examples for description techniques are graphical notations like class diagrams and state transition graphs from modelling languages like UML [OMG99] as well as textual notations like interface specifications expressed in CORBA IDL [OH98] C++[Strou00] or Java [Flan99] Well-defined consistency criteria between the different description techniques allow to verify the correctness of different views onto a system with the help of specialized tools.
- Development should be organized according to a *process model* tailored to Componentware. This includes in particular the assignment of discernible development tasks to individuals or groups in different roles, for example, a software architect responsible for the overall design of a system, and component developers who produce and sell reusable components.
- The description techniques and the Componentware process model should be supported by *tools*. At least, these tools should be able to generate an implementation of the system as well as corresponding documentation. Furthermore, they could facilitate the verification of critical system properties, based on the formal system model.

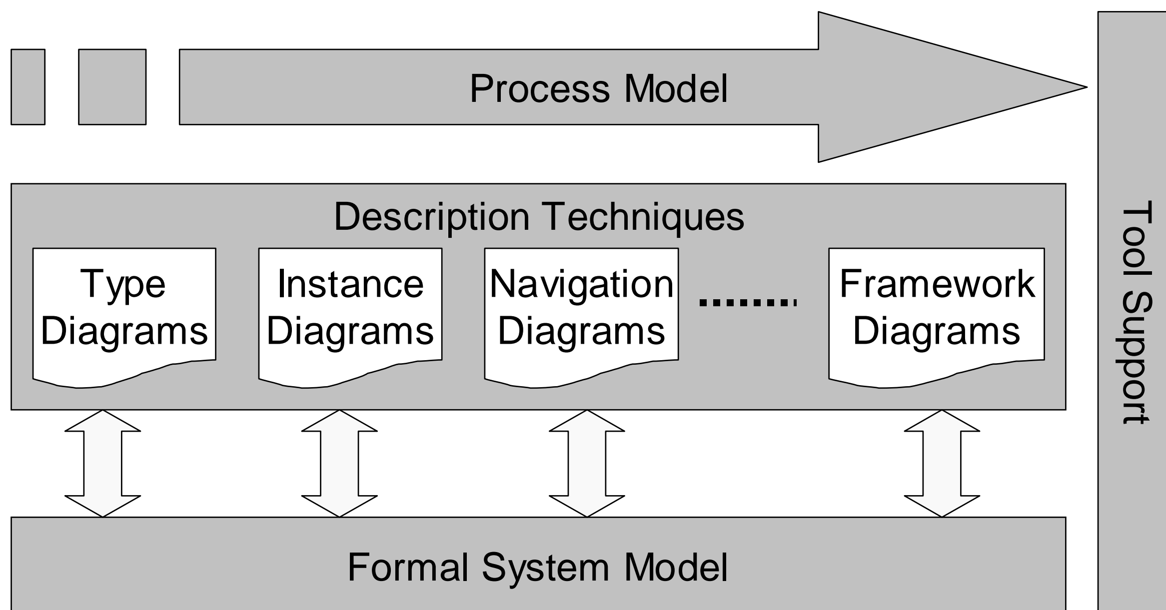

Figure 1: Componentware Methodology

These essential parts of a Componentware methodology and their relationships are outlined in Figure 1. Automate (http://automate.informatik.tu-muenchen.de/); a tool developed by the A1 subproject of the FORSOFT research group of Technische Universität München [FORS00] has attempted to address this problem. The Automate system is able to automatically generate the code of a distributed three-tier architecture system from UML class diagrams. However, the systems‘ functionality is limited, only generating the code of persistent classes. User specified methods cannot be generated. Only frame code is produced and the programmer must add additional code. These deficiencies reveal the need for a new tool based on a Componentware approach to software development.

1.2 Basic Concepts of Componentware

The main goal of Componentware is to produce a well-structured system consisting of understandable and reusable parts. Components are the main building blocks of the system, encapsulating common functionality. Components can only be accessed through clearly defined access points, called interfaces. Interfaces of different components can be connected together, thus providing the basis for building systems and enabling components to interact. Additionally interfaces have attributes, which relate to variables in object-oriented programming. They are used to hold data relating to the interface.

It is possible to create and destroy components and the various other elements of component based systems at run-time. Connections are an integral of a component-based system. All interactions and operation occur via the sending of messages across the system, which require connections between the various elements. A component cannot be directly accessed by another; communication takes place by sending messages, which consist of one-way method calls, to the interface of the desired component. For an interface of one component to interact with that of another, a connection must be established between the two interfaces. When the connection is no longer required it can be destroyed.

When a component or interface is destroyed, its connections are also deleted. Thus other element in the system will no longer have a connection to that particular element and will not attempt to send further messages.

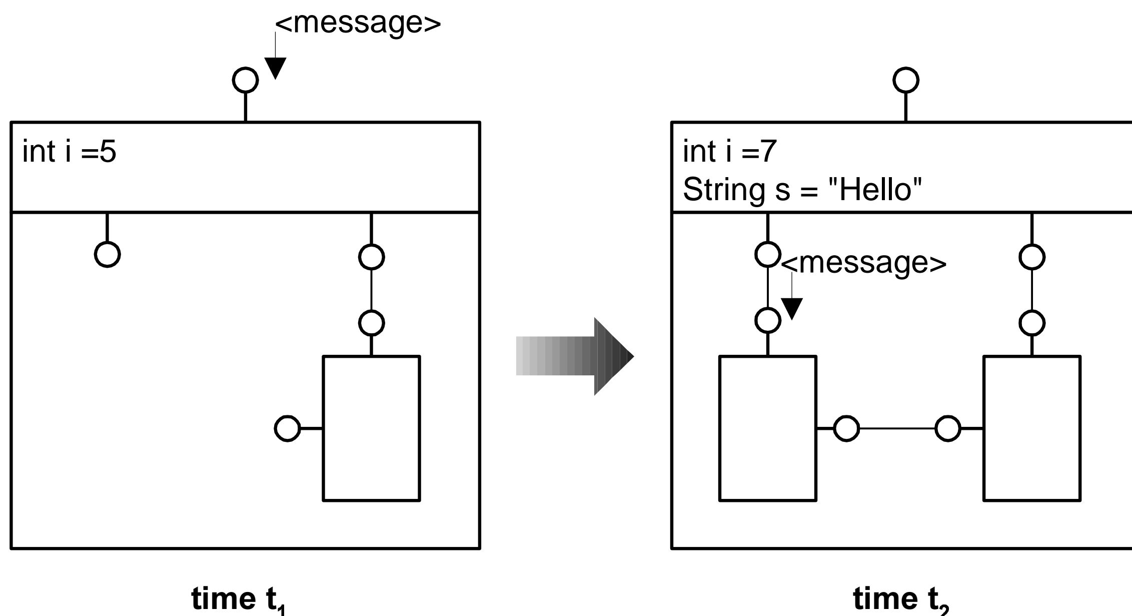

Figure 2: Basic Concepts and Elements of a Component-Oriented System

The structure of the system can be dynamically changed at runtime as indicated by Figure 2. The processing of messages can result in the creation or deletion of components and interfaces etc. Additionally the values of attributes can be altered. An attribute holds variable data relating to an interface. In the above example, the processing of the message received at t1 results in the value of the attribute i changing, and the creation of s, a new String attribute.

1.3 Benefits of Componentware

The key concept of Componentware is the reusability of standardised units, which ensures an investment over multiple applications. It is a concept adopted by other engineering disciplines as they matured, and as the discipline of software engineering evolves it attempts to benefit from the advantages offered.

'Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.' [Szyp98] This independence and encapsulation of functions ensures robust integration, which facilitates the composition of systems lacking in the dependencies of object-oriented software.

The modularity of a component-based approach encourages the move from large systems to modular structures that offer the benefits of adaptability, and scalability. Such systems provide financial and functional benefits. Due to the ease of integration, small specialised units can be arranged to provide solutions for a wide range of applications.

Custom made software is expensive and time consuming to produce. Business requirements change rapidly and software is often completed too late to be of any use, becoming obsolete before completion. A component-based system can benefit from already existing components, which have been tested for quality and completeness. The time taken to build a system from these building blocks will be much less than building software from scratch. Existing components are standardised, while the process of assembling the system provides the opportunity for significant customisation.

The process of upgrading is simplified in a component-based system. A complete overhaul of the existing system can be replaced by an evolutionary upgrade of individual component when additional functions are required. Due to the encapsulation of functions within individual components, only the affected component needs to be altered. Additional components from different sources can also be added with little or no alteration to the code of the present system (changes may be required only to take advantage of the functions offered by the addition to the system).

1.4 Aims and Objectives of DesignIt

The goal of the DesignIt project is to create a new prototyping environment based on component based concepts. The project has two main parts to be realised:

- A container or run-time environment for components
- A generator that generates the code of the components from given specifications

The generated code should run in the container and the integrated system should provide an environment capable of aiding programmers in the development of component-based systems. The system should work for any code sample adhering to the formal specification model.

1.5 Readers Guide for the Paper

In the first chapter we provide a description of the problem and the approach to the solution. A background to Componentware is provided, outlining the advantages of the approach. The chapter ends with a statement of the Objectives of the DesignIt project.
In Chapter 2 initial stages of the project are outlined, from the setting up of the development environment to the development of the Generator and Container. A detailed project schedule is contained in Appendix A. This section provides a glimpse into the stages of development for each phase.

We look in detail on the design of the generator system in Chapter 3. We explore the classes created and the functions provided by them. Accompanying details of selected template document syntax is explained.

The implementation details of the system are treated in this Chapter 4, looking at the operation of important methods and taking an overview of the testing strategy. Some Java features exploited in the implementation of the project are discussed, with a justification for their use.

In Chapter 5 the conclusions of the project are dealt with, evaluating the success of the project, and failings in any tools and software used by the developers. Future improvements for the current system are mentioned along with developments planned to expand the system.

2 DesignIt: Overview and Basic Concepts

Throughout the project the DesignIt team members followed and maintained an online schedule for the duration of the project. The project plan was devised and placed on the project web site, providing a guide to the various stages of development. This schedule was modified and updated constantly to reflect realistic targets and deadlines. It also provided a common reference point to progress being made and steps points to be developed further. The plan as it stood at the end of the project is enclosed in the appendix, see Appendix A. Following this schedule, the project was organized in four main phases:

In the first phase, see section 2.1, it was necessary to install, configure and familiarise the team members with their tool environment. The tool environment itself consisted of the hardware and an operating system to be used through the project in alliance with the programming language and ancillary tools used in the project implementation. Familiarisation was achieved through the use of small test programming samples and common operations performed on the newly configured development environment. For instance, a small RMI sample (RMICount) was implemented. Once this was completed we were able to run the existing breakplanner application.

The next phase, see section 2.2, produced the working sample – the breakplanner system. The breakplanner had been developed in TUM as part of a previous project to illustrate using UML for modelling a distributed Java application in an object-orientated fashion [BRS97]. The system was studied to introduce the team to the principles behind a breakplanner. The existing system had been built without an implementation of a user interface. The task for this work package was to provide a graphical user interface to edit breaks and a view window to allow a view into the system at any one moment. We use the breakplanner as a sample application for the rest of the project. At the end of the project the breakplanner system should be generated and executed in the DesignIt tool.

The goal of the third phase, described in section 2.3, was to develop the core of the DesignIt tool, the Componentware Runtime Environment or so called Componentware container. This would allow moment and processing of messages being sent between interfaces be controlled every message having to be sent through it. The container also acts as the creator and deletor of all components in the system. Around the container a sample component-oriented breakplanner system was to be created as both an illustration of the capabilities of the system and also as a means of testing for the Runtime Environment.

As an addition to the container fellow student Lucien Hoogendoorn developed a control panel. This panel was developed to allow a view into the component, interface and connection hierarchy as well as the values of the attributes of the interfaces. Functionality was to be provided to allow sending of messages to interfaces and setting values of attributes. The panel was also to be used as the main means of testing the container and the breakplanner system sample.

The final phase, phase four in section 2.4, of the project aimed to create a generator capable of generating java code from a series of specification documents. The repeatability and simplicity of components and interfaces allows the relatively straightforward definition of these elements as a formal specification. A generator was to be created to allow the generation of fully formed complete compliable java code from a series of specification documents.

These formal specifications consisted of an XML document based on the definition with the corresponding data type definition files (DTD). The DTD follows the structure of the component-based system. Templates were also used to provide the outline of the code for the types of elements – components, interfaces and attributes.

The structure of the XML document is a hierarchical one and lends itself to a modelling of the structure of a Componentware system. The DTD ensures the validity of the fields and syntax of the XML documents. The templates are the final link in the generation chain providing a code skeleton upon which the XML information can be mounted.

In the following sections we will discuss these four phases and the corresponding results in detail.

2.1 The Development Environment

The efficient use of time and efforts is crucial in the success of any project. With the Design it project every care was take to ensure that the team used as many tools and project aids as was prudent. The achievement of familiarity with ones programming environment was given priority alongside the ability to personalise and retain it. With the care and time given to efficient use of time much frustration and waste was avoided.

There were a number of tools and methods used through the project to improve developer productivity and speed up development time. These some of these are outlined below.

2.1.1 Hardware

This project was being undertaken in a communal laboratory. To ensure a stable environment over the duration of the project, removable hard discs with full administrative rights were used. This allowed the developers to customise their system settings to an optimum level and retain their custom settings throughout the development.

2.1.2 Programming Language

The programming language used in the project was Java SDK 1.1.3. At the time of the project implementation, this was the most up to date version available. Java offers a wide range of utilities and ready-made collections which can aid in rapid software development. These collections negated the need for user testing of any data structures whilst also providing a series of structures optimised for operation speed. Extensive online support is available to the developer via Java documentation and online user groups.

2.1.3 Borland JBuilder 3.5 Foundation Edition

The JBuilder tool is a rapid development tool for Java code. Central to its operation is the creation of project files, within which project specific classes and variables can be fixed. Working with J Builder a developer can avail of a real-time syntax checker, displaying any errors as they are made. Compilation of entire project of many classes in different classes is made simple and requires only the use of one button on the tool panel.

Familiarisation with the tool was achieved through the use of test samples. The procedure followed by the developers involved creating a HelloWorld test sample within JBuilder. This program was compiled and run through the JBuilder Environment successfully printing the

required “HelloWorld” message to the command line. The layout of the Tool was user friendly and an example of it in action can be seen in Figure 2.

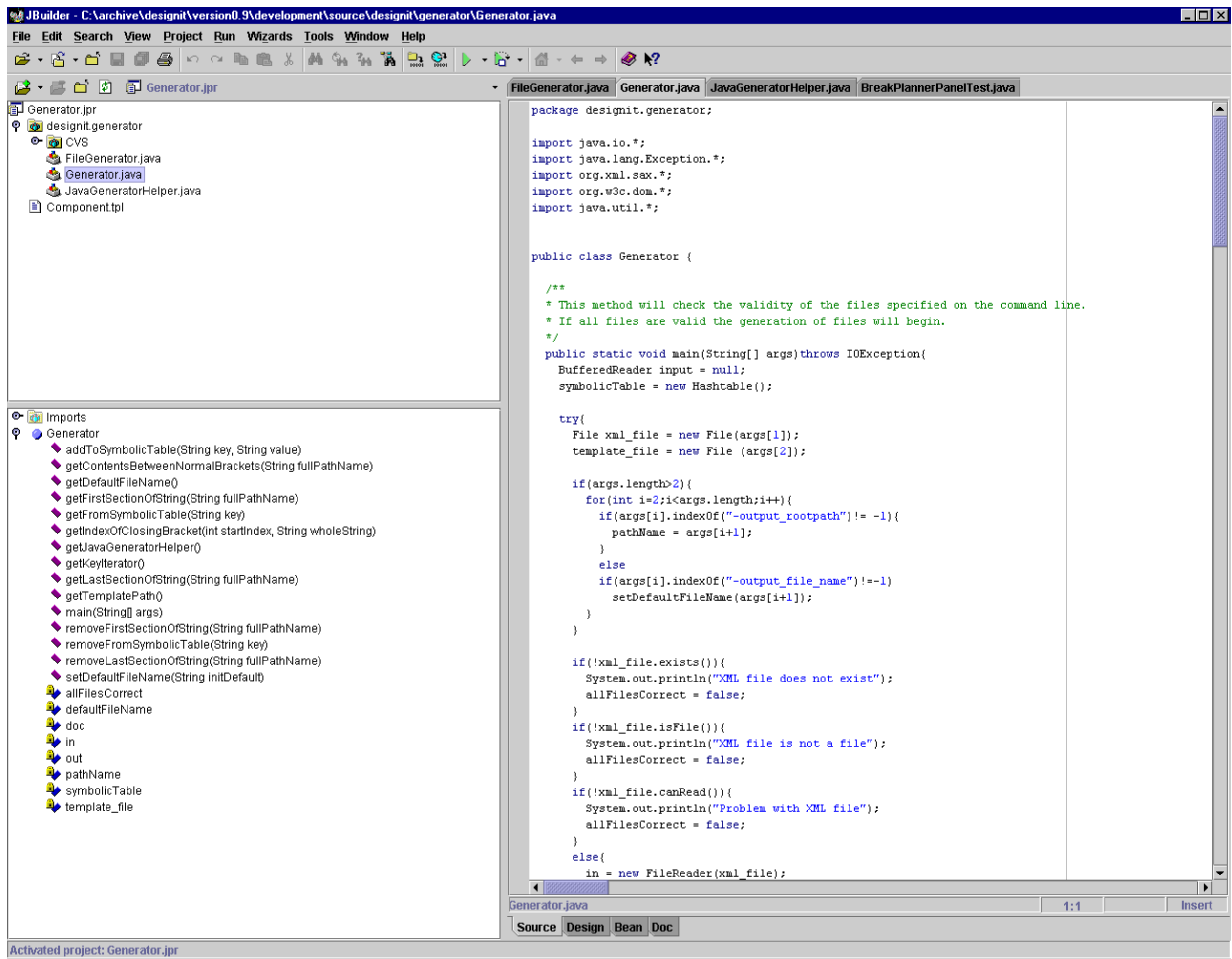

Figure 2: JBuilder Screenshot

The developers did note some limitations in its capabilities through the project. The tool often did not compile code correctly, mixing old redundant code with newly compiled code, leading to confusion and wasted time. In addition to this the tool did not permit the debugging of systems containing greater than one threads running concurrently.

2.1.4 Concurrent Versions System CVS

This communal archiving system was used throughout the project to manage the developed code being produced. CVS stores all the versions of a file in a single file in a special way that only stores the differences between versions. This means that the development team could save their work many times yet not lead to an excessive amount of disk resource being taken up by obsolete files.

The repository itself was divided into sections representative of the various stages and resources related to the project.The Layout of this tool can be seen from the screen shot below in Figure 3.

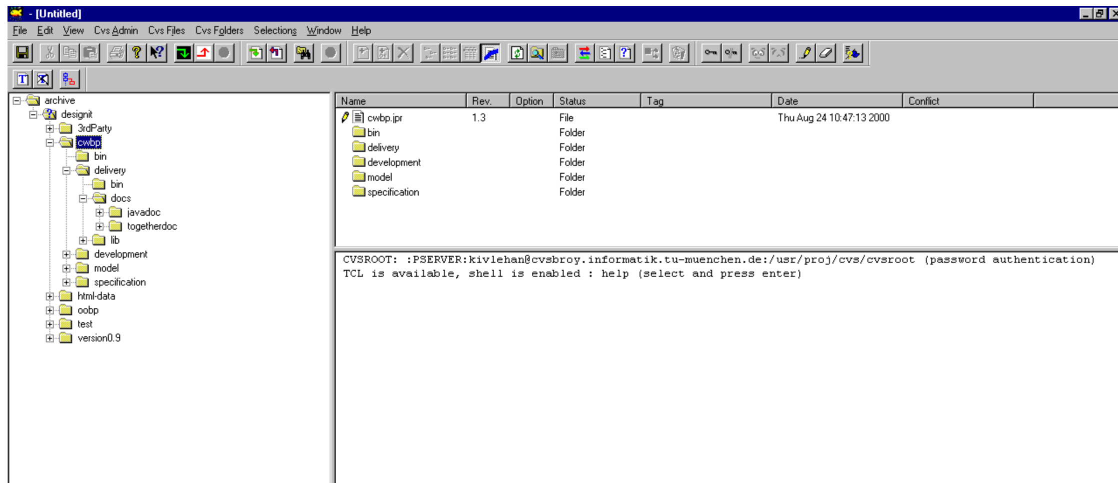

Figure 3: CVS Screenshot

To test the set-up and operation of the cvs tool a test folder was created for each developer. The developers then used these folders to save a copy of their Hello World program created in previous testing. With this done the developers could check the operation of the system by accessing each others folders. Upon making a change and saving it to the system the developers could verify the operation of the system

2.1.5 TogetherJ

TogetherJ is a Computer Aided Software Engineering tool - CASE tool - which aids in the development of object-orientated software systems. The core use of TogetherJ is that by creating a model of a project the tool can also develop the code describing this configuration. Within the visual model, links and relationships between classes can be set and modified, again with the relevant code being generated or altered automatically in the background.

Within the project great use for this tool was made, exploiting its documentation creation capability. With the minimum of effort on the developers behalf TogetherJ can provide professional looking word documentation describing a project both with the text meant for the java documentation process and diagrams from within the tool.

The diagrams within TogetherJ comply fully with the UML standard [OMG99], each element of a system such as packages and classes having a particular symbol allocated to them. Relationships between classes also follow the UML standard for linking symbols.

Testing of the tool was performed as part of the design process of the runtime environment, where the entire team undertook to define the core workings and classes for the package. A project file was set up for the package and all files to be used were stored correctly in the required place. As new files were added to the package diagram, the corresponding code appeared in the package folder.

Once again the developers did note some limitations in the operation of the tool. The modification of existing links often led to their deletion and replacement with a generic name not consistent with existing code. In effect the TogetherJ tool was liable to change perfectly good code. For the normal development of code this unpredictability forced us to turn elsewhere for our java programming environment.

Another flaw in the software was that in documentation generation there was no means of excluding unwanted folders and packages from being included in the generated word file. Removing them from the diagram did not extend to their removal from the project. The tool was then liable to generate documentation with unwanted classes which required time form the developers to manually delete them.

2.2 Our Sample Application: Breakplanner System

The goal of this phase was to establish a productive environment and familiarise the team with the settings of the various software tools, and become familiar with the sample application relevant to the remainder of the project through the creation of two additional client interfaces.

The breakplanner application was developed to organise the allocation of teachers to the supervision of pupils in various parts of a school during breaks. It was developed at TUM to illustrate the use of UML modelling for distributed Java applications. Each break must be supervised by a teacher, who are assigned to breaks depending on the time they spend teaching(i.e. their percentage of a full-time post). Teachers can provide the school with exclusion times, when they are not available for supervision duties due to other responsibilities. The system allows the creation and deletion of BreakPlans and the assignment of teachers to breaks. The tool additionally computes some statistical data indicating how many breaks a teacher still needs to be allocated and highlighting which breaks need to be supervised, or when a conflict exists. A conflicts exists when a teacher is allocated to two different breaks with overlapping times, or are allocated to a break which overlaps with an exclusion time.

The existing breakplanner is a distributed system, implemented using Java RMI for communication. The data is held by the server in the system and can be accessed by the clients, who can view and edit the information. The connection between client and server is realised via the Observer pattern [GHJV95]. The simple client provided establishes the initial data on the server. All current features are implemented using JDK1.1.

The main task of this section is to implement the new sophisticated clients: A Statistics View and a Break Editor. These features are to be developed using JDK1.3.

The Statistics View contains information regarding teachers and breaks in the system. Teachers are divided into three categories depending on the current state of their duties as shown in Figure 4. Teacher either have been allocated sufficient duties, too few duties or too many duties. The Statistics View must provide an up-to-date view of the data and is connected to the server via the Observer pattern.

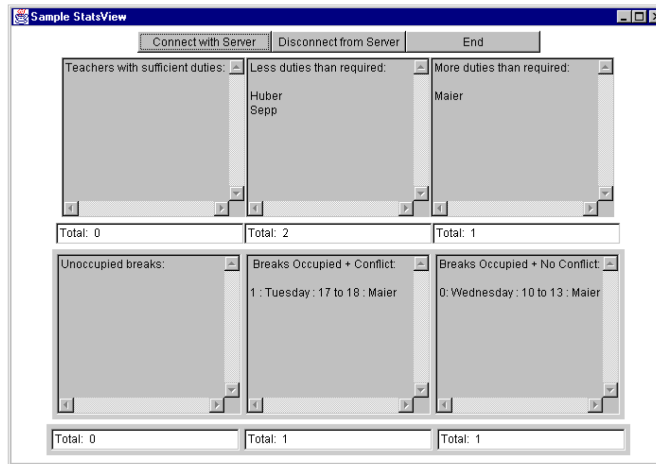

Figure 4: The graphical interface of the Statistics View

The Break Editor provides a facility to add new breaks and edit existing breaks. Each break has a day, a start hour and an end hour, that can be selected using the newly created GUI . Additionally, each break can be supervised by an existing teacher or none(i.e. an unOccupied Break). An existing break is edited by selecting the break in the list box (Figure 5), changing the values shown in the corresponding drop down boxes in the right side and confirming the changes by pressing the button "Change Settings of Selected Break". A new break is added by setting the values in the drop down boxes in the right side and pressing the button "Add New Break".

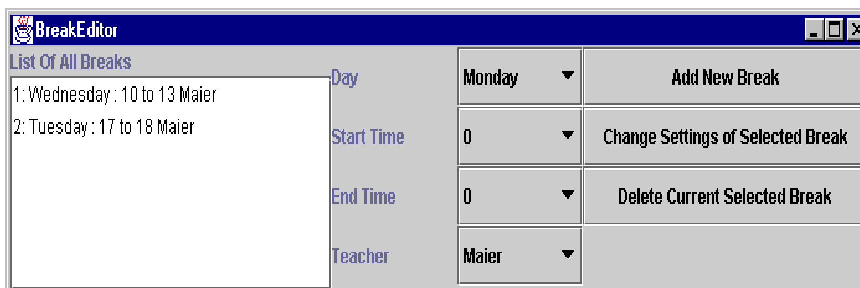

Figure 5: The graphical interface of the Break Editor

The most important feature of the new implementation is the interaction between the client programs and the RMI server. When a Break is added or altered the data on the server must be updated. Observers of the data must be notified to update their view of the data. The Statistics View will display the new Break in the relevant category and update the statistics for the Teacher assigned to the new Break. The break Editor will display the new Break in its' list of Breaks in the system.

The newly implemented interfaces were integrated with the server provided by the original application. The data on the server was initialised using the client program from the previous version and provided the statistics View and Break Editor with display information.

2.3 Component-Oriented Runtime Environment

The first stage of development used the TogetherJ tool, which provided a basic code outline, based on the representation of the system as a class diagram. The formal specification of the component-based system was finalised at this point

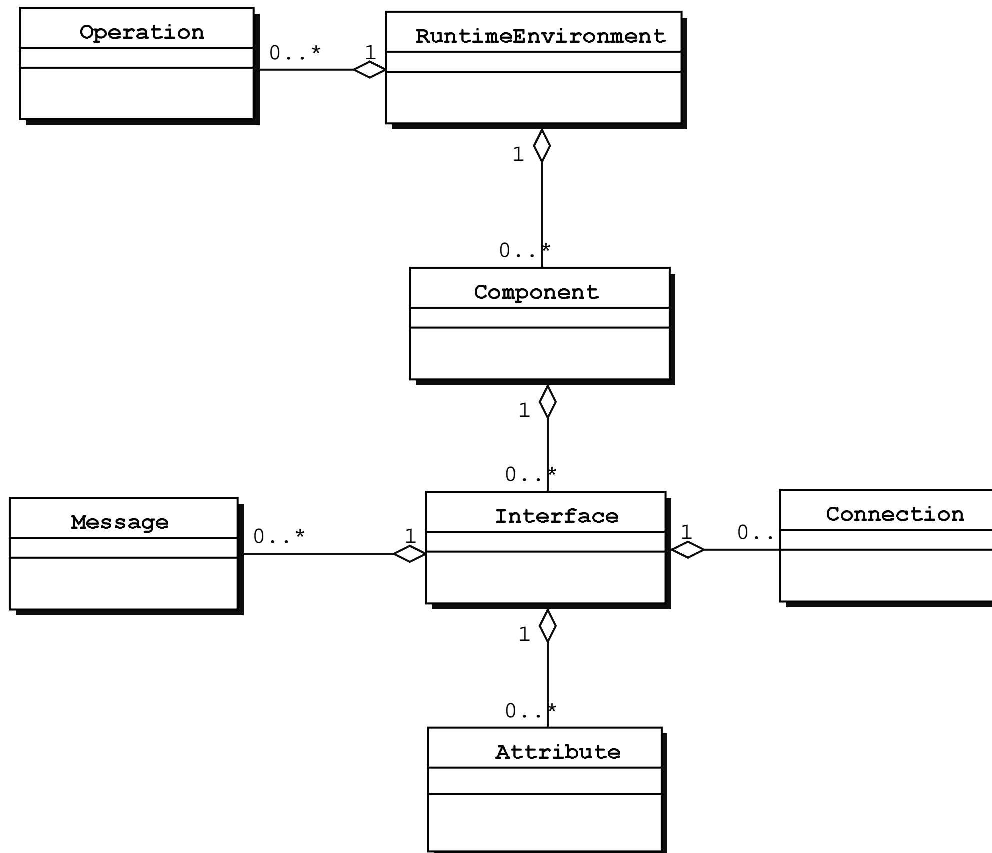

Figure 6: Core system structure

The system consists of a series of components, interfaces, attributes, connections, messages and operations(Figure 6). The Runtime Environment has a Set of components and a Set of operations. Each component has a Set of interfaces. Each interface has a Set of attributes, connections and messages.

To implement the simple sample the structure of the engine and the necessary classes to be included were developed. In addition to the Runtime Environment itself, classes had to be devised for each element in the system, representing the common functionality of similar elements.

A standard format for components, interfaces and attributes was devised. For some elements the format was simple, while others proved to be more complicated. For example, an Attribute in the system needs only to hold a value belonging to an interface. Therefore the main functions of an attribute are getValue and setValue. A components main function was to maintain a Set of interfaces, so the functions of a component involve manipulation of this Set. In contrast the varied operations involved in interfaces required a greater range of functionality. The interfaces require the ability to process messages, so a method to perform this function was devised. In addition an interface has a Set of connections and a Set of attributes. .

Several methods were required to deal with the various aspects of these relationships.

Similar consideration was given to the format of connections, operations and messages. Messages had to be devised to enable parameters to be added to the message. A standard way of accessing the parameter types and values was included. Connections merely contain information on the two interfaces they are linking; in addition a connection can be allocated a name to indicate the relationship between the interfaces.

Operations proved a little more difficult to devise. Due to the different sets of parameters required to create different elements in the system (for example, the creation of a component requires only the component class to be specified, whereas the creation of an interface requires the interface class and a component for the interface to be added) it was decided to create separate classes to deal with different objects to be created. These classes are all designed to inherit form operation, which specifies that a performOperation method must be declared. This was designed to enable all operations to be dealt with by the Runtime Environment in a simplified manner.

To familiarise the team with component based techniques a small sample was used to aid development of the Runtime Environment. This required the creation of components in the system and the addition of simple interfaces, containing methods.

The simple classes for the components and interfaces were provided. Components were created using the classes provided and interfaces were created and added to these components. The system could be viewed in the panel. Various methods provided in one of the interfaces gave the opportunity to test the ability of the Runtime Environment to receive and process messages correctly. The messages created in the sample were at first very basic, only printing out some text to indicate they had been processed. Tasks to be performed increased in difficulty to creating new interfaces and calling methods on the newly created interfaces. This provided us with a shell of the Runtime Environment, which could handle simple requests.

The next stage of development was to implement the BreakPlanner sample using the Runtime Environment. The various objects from the old system had to be coded based on the standard formats devised for components, interfaces and attributes. A small sample of the system is illustrated in Figure 7 The elements of each type will have all the functions as defined in the engine, but additionally require specific features.

Aspects of the previous system that constituted variable data were mapped to attributes in the new system. These attributes are capable of holding a value. Each attribute of each interface required a separate class due to differences between the attributes. Some merely have their values set by the user while others such as neededDuties in the teacher interface are calculated based on data held in other parts of the system, i.e. this value depends on the teachers' job share and the overall number of breaks in the system.

The owners of the attribute data, the main focus of the previous system, such as teachers and breaks equate to interfaces in the component-based system. These are the aspects of the system who require connections to be made and can be manipulated via the transmission of messages. Those elements of the previous system that represented the intended user interfaces of the entire system are coded as components in the new system.

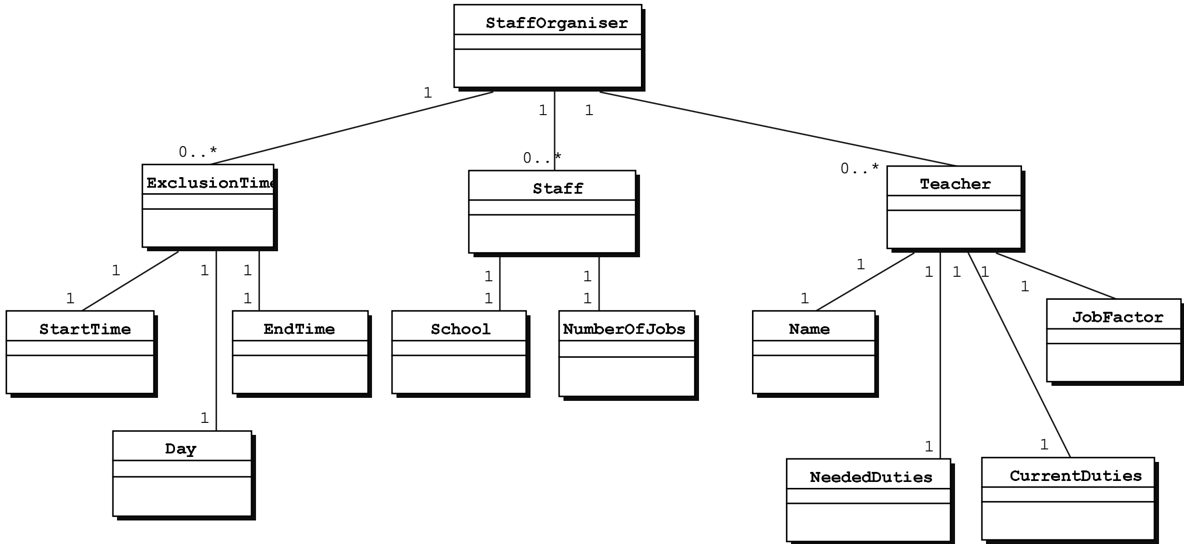

Figure 7: Structure of Staff Organiser Component

2.4 XML and Template based Code Generator

The aim of this stage was to achieve an implementation of a Generator system, able to create java source code from a formal specification. This specification consisted of an XML file representing the configuration of a Componentware system alongside a series of code template files.

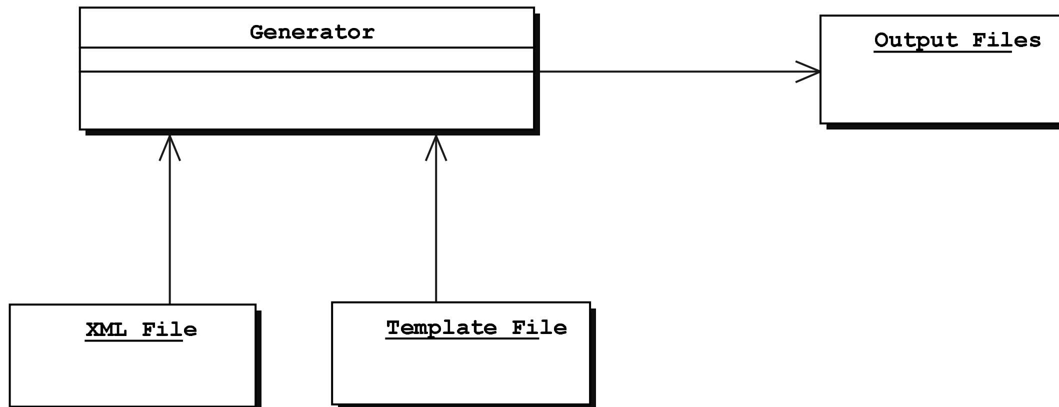

Figure 8: Overview of Generator System

The inputs to the system were the XML file and the Template files with the output being Java source files. An overview to the inputs and output of the generator system can be seen in Figure 8.

The approach taken in the achievement of this work package was to make all development in steps, an incremental development approach. Starting with a basic core of a file input-output system functionality was added and tested as the model was built upon. In a similar way the functional level of syntax used in the template documents was continuously increased as a means of testing the capabilities system. With every build of the system bugs and deficiencies in implementation were exposed and solved. With a full developed template developing a satisfactory out put, the work package was deemed complete.

The system achieved proved successful in the generation of java source code from the inputs given, as was the aim as stated in the package goal. The interactions within the final system are outlined in Figure 9.

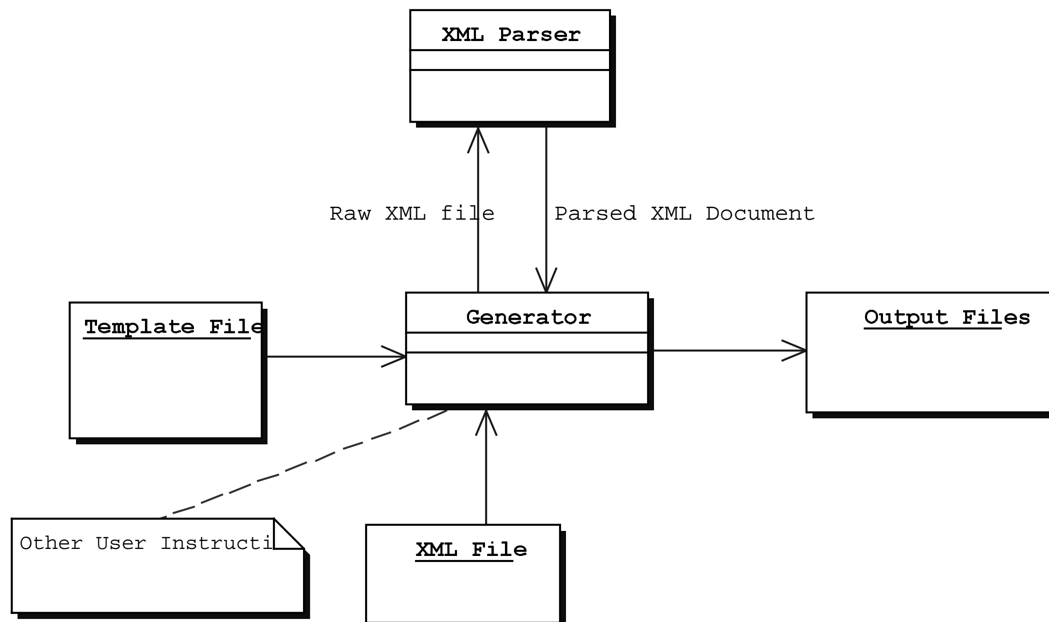

Figure 9:Interactions of Finished System

It is possible in the system to generate many files from just one input setting, with the generator spawning sub generators to cope with whatever amount, large or small, of components and interfaces required by the input documents. The complexity of the generation is shielded from the user.

A feature of the developed system was the ability to specify output paths by the user. Taking advantage of this facility it was possible to have all files created by the generator diverted to any valid file destination path. This gives the user extra control over the operation of the system.

An XML file was required for the generator to operate. This file contained all the specifics of each component in a Componentware system along with their interfaces and attributes. The processing of the file into a format navigable by the system was done externally using an XML parser. The resulting document could then be traversed like any tree like data structure.

The other main input to the system was in the form of a code template file. This included special statements contained within “#” symbols, allowing the system to take instructions during the generation process. The syntax attached to the use of these # statements allowed different actions to be performed depending on the number of #’s used. Use of the different statements enabled actions like retrieval of values from the XML document, invoking of methods or the repeated processing of an external template be performed.

Inbuilt to the developed system was a series of validity checks and error detection code used mainly in the initialisation of the system. Should a user enter incorrect instructions or invalid input files, the system would exit gracefully providing a meaningful message on the output path.

2.5 The High-Level Architecture of the DesignIt Tool

The Generator system belongs to a greater overall system which enables the running of the generated code and the modification of the system using a user friendly Graphical Panel. There are three packages in the system, an engine a generator which generates code from a specification, the engine upon which the generated code can be ran, and a panel with which the user can see the current state of the running system. A package diagram of this complete system is provided in Figure 10.

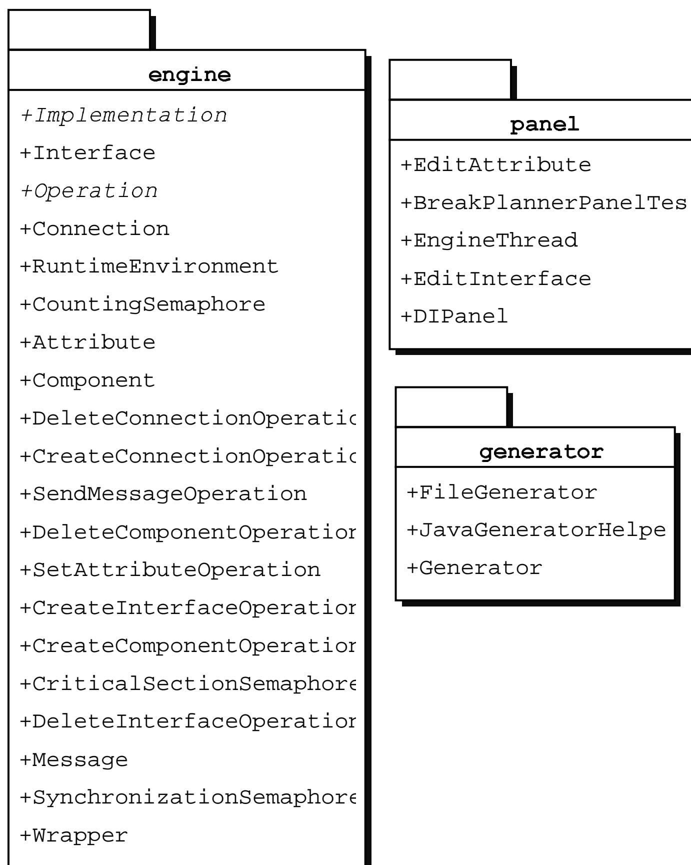

Figure 10: High Level Architecture Class Diagram

The Engine provided the Runtime Environment with which the compiled code could be mounted. It is a component based system which operated on the component ware principles outlined in Section 1.2.

This system allowed the creation and deletion of a component are system consisting of components interfaces and attributes. Messages could be sent to interfaces allowing methods to be invoked. Connections could be made between interfaces denoting a relationship between these interfaces. The Runtime Environment also enabled the setting of attributes to any valid value.

The panel provided the user interface for the system, which ran using the Runtime Environment. It allowed an insight into the current state of the system and the ability to make modifications. In a simple visual manner messages could be sent to interfaces allowing methods to be called. Attributes could also be modified and changed to valid values.

The Generator package contained the classes used in the process of the generation of code from an input of a formal specification. These will be dealt with fully throughout this chapter.

3 Architecture and Design of Generator

3.1 User Interface

The user interface for the system was a command line entry in the form of :

java designit.generator.Generator -generate <XML-file> <TPL-file> [-output_rootpath <outputrootpath>] [-output_file_name <outputfilename>]

The arguments can be explained as follows:

-generate	compulsory argument used here for completeness. In the future different commands may be possible in its place.
<XML-file>	compulsory argument being a string representation of a path to an xml file
<TPL-file>	compulsory being a string representation of a path to a template file
[-output_rootpath <outputrootpath>]	optional argument being a path to a folder where is is wished that all the output from the generator will be dumped. This folder is to be created if not already on existence
[-output_file_name <outputfilename>]	optional argument allowing a standard general output file to be named. This file will be saved at the default root path location if it has been set

If an output file name location was set in a template file it was this location that was used, overriding the command line default file name.

By not specifying an output file name location in either the command line or the template, a runtime error would occur and the generate would exit.

Related to the user interface were the files supplied by the user, in the form of an XML file and a template file. To explain these files and their format, it would be useful to present a small sample of input files and generated output. This name given to this small sample was the intro sample.

Code was generated by using the combination of the standard template files, which described the code of interfaces, components and attributes, alongside an XML file describing the specifics of the system to be produced. Firstly an example of an XML file is shown below in Text Extract 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v3.0 NT (http://www.xmlspy.com) by Andreas Rausch
              (4Soft GmbH) -->
<!-- edited by Andreas Rausch -->
<!DOCTYPE COMPONENT_SPECIFICATION_DOCUMENT SYSTEM
              ".\..\..\generator\DesignItSpecification.dtd">
<COMPONENT_SPECIFICATION_DOCUMENT>
   <COMPONENT_SPECIFICATION NAME="designit.sample.intro.CB"
              MIN_CARDINALITY="1" MAX_CARDINALITY="1">
      <INTERFACE_SPECIFICATION NAME="IB" MIN_CARDINALITY="1"
              MAX_CARDINALITY="1">
         <CONNECTION_SPECIFICATION NAME="IA_2_IB">
            <CONNECTION_END NAME="FROM_IA_2_IB" TYPE="Set(IB)"
              MIN_CARDINALITY="1" MAX_CARDINALITY="*"/>
```

```
        </CONNECTION_SPECIFICATION>
        <MESSAGE_SPECIFICATION NAME="bar">
          <BEHAVIOR_SPECIFICATION BEHAV-
               IOR_SPECIFICATION_LINE="System.out.println(Thread.currentThre
               ad().getName()+ &quot;: &quot; + &quot;method bar called on
               interface IB:&quot; + getContext().getName());"/>
        </MESSAGE_SPECIFICATION>
      </INTERFACE_SPECIFICATION>
    </COMPONENT_SPECIFICATION>
</COMPONENT_SPECIFICATION_DOCUMENT>
```

Text Extract 1 Example of XML File-Used In Intro Sample

The XML file represented perfectly the hierarchical structure of the system to be generated, with components on top, components having child interfaces and then interfaces having child attributes and connections. The specifics of the code relate to the template file, an example of which is given in Text Extract 2. These specifics are beyond the scope of this report.

```
##FileGenerator.setOutputFile( ##JavaGeneratorHel-
             per.getFileOutputPathFromComponentName(
             #COMPONENT_SPECIFICATION_DOCUMENT.COMPONENT_SPECIFICATION.getA
             ttributeNAME#)##)##
/*******************************************************************
*  this file is generated by the DesignIt code generator            *
*  do not edit it!                                                  *
*  in case of re-generation changes may be lost!                    *
********************************************************************/
package ##JavaGeneratorHelper.getPackageNameFromFullName (
             #COMPONENT_SPECIFICATION_DOCUMENT.COMPONENT_SPECIFICATION.getA
             ttributeNAME#)##;

import designit.engine.*;

public class ##JavaGeneratorHelper.getClassNameFromFullName(
             #COMPONENT_SPECIFICATION_DOCUMENT.COMPONENT_SPECIFICATION.getA
             ttributeNAME#)## extends Implementation
{
  public ##JavaGeneratorHelper.getClassNameFromFullName(
             #COMPONENT_SPECIFICATION_DOCUMENT.COMPONENT_SPECIFICATION.getA
             ttributeNAME#)##(Wrapper myWrapper)
  {
    super(myWrapper);
  }

  /* method setValue should never be called on a component. */
  public void setValue(Object value)
  {
    throw new java.lang.RuntimeException("method setValue should never be
             called on a component!");
  }

  /* method getCopyOfValue should never be called on a component. */
  public Object getCopyOfValue()
  {
    throw new java.lang.RuntimeException("Method clone should never be
             called on a Component!");
  }
}
```

Text Extract 2 Example of Template File-Used In Intro Sample

It can be seen that the template file closely resembled an example of a normal Java source file. The template file can be conceptualised as the “frame” onto which the substance contained in the XML file was put. It was a mixture of regular straight Java and special statements.

During the generation process the template file was read through, line by line, by the generator. For normal text the generator dumped the line straight into an output file. In the case of special statements being read the generator must evaluate the output itself. These special statements, contained between #|## markers, allowed the generator to call methods or make references to the XML document and then use the resulting information to determine what should be written to the output file. The resulting generated component CB can be seen below in Text Extract 3.

```
/**************************************************************************
*****
*  this file is generated by the DesignIt code generator
*
*  do not edit it!
*
*  in case of re-generation changes may be lost!
*
**************************************************************************
*****/
package designit.sample.intro;

import designit.engine.*;

public class CB extends Implementation
{

  public CB(Wrapper myWrapper)
  {
    super(myWrapper);
  }

  /* method setValue should never be called on a component. */
  public void setValue(Object value)
  {
    throw new java.lang.RuntimeException("method setValue should never be
called on a component!");
  }

  /* method getCopyOfValue should never be called on a component. */
  public Object getCopyOfValue()
  {
    throw new java.lang.RuntimeException("Method clone should never be
called on a Component!");
  }
}
```

Text Extract 3 Example of Generated Code-CA.java from Intro Sample

The generated code is fully compileable and fully functional. It can be seen that the generator has removed all the special statements and replaced them with the correct text for this particular component, whilst ignoring the regular text. Under normal circumstances this is last interaction that the user will have with the generation system.

3.2 Generator Package in Detail

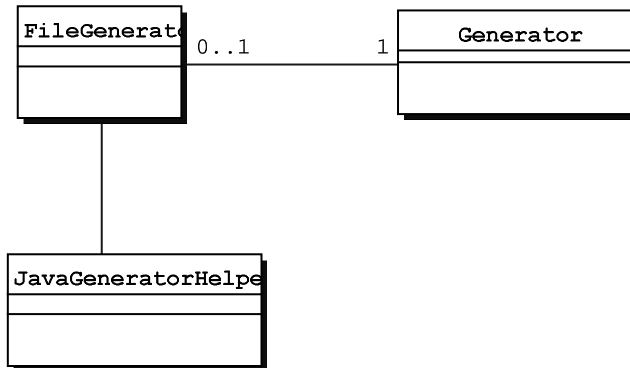

Figure 11:Classes in the generator package.

These three classes comprised the generator package. The overall entry by the user was made through the Generator class itself. This was the class that provided all the initialisation checks and operations. With all being present and correct the generator would allow the actual detailed generation process to begin. An instance of FileGenerator was created and a method to begin processing the first template called. The FileGenerator could then take control, spawning new FileGenerators as required to divide up its workload of generation tasks. The JavaGeneratorHelper class contained a number of helper methods which were accessed by templates through the FileGenerator. These methods performed operations only relevant to templates being used to generate code for Java Systems.

3.3 Generator

In the Generator class we placed a main method, the only occurrence of a main method in the entire generator package. The initiation of the generation process was triggered from here, before which a series of file validity checks etc. were performed.

The user inputs to the system were made through this class, being provided as a series of parameters as specified in section 3.1. This was the only interface with which the user could interact with the system.

All methods in this class were static. This choice was a deliberate one, allowing the methods of this class to be accessed without either having a new instance of the object or having a back pointer to an already created object. Finally Generator was also the storage place for any generation wide variables and data structures.

3.4 File Generator

File Generator was the central processing area of the entire system. It was here that all input streams were processed and output streams written to. In the FileGenerator an input was read (initially direct from a template) line by line. This line of text can contain the following 4 types of information.

- a statement enclosed by a # statement
- a statement enclosed by a ## statement
- a statement enclosed by a ### statement
- either plain text or a combination plain text and one of the statements above.

Using the information contained in the special statements the FileGenerator decides on an action to be performed, for the # a method call is made, for the ## a lookup is made to the XML file and for ### a high level execution decision is made. A full description of the statements follows in the next sections.

3.4.1 File Generation Hierarchy of Order of Importance

The hierarchy for hash recognition in the File Generator input reader was Triple hash then Double hash then Single hash. In a line containing multiple instances of special statements, the resulting text or operation is determined in the above order.

3.4.2 # (one hash) statement

A # statement indicated a reference is being made to the underlying XML document.

There can be more than one # statement in a line, each statement being evaluated one after the next.

A # statement was the last statement in a line to be processed, the other statements having precedence over it.

Shortcuts could be created which would allow certain text in a # statement be replaced by a corresponding value if it existed in the Generator hashTable.

The syntax for a single hash statement was as follows:

single hash statement = *path_to_base_element* + "." + |"getAttribute"||"getElements"| + *key_to_sub_search*

path_to_base_element = Names of a series of nodes (dot separated) which defined a path to an element. This path started at the top of a parsed XML document and ended at the desired element.

"getElements" = statement which lead the evaluator to return a series of cloned documents. Each cloned document was a clone of the chief xml document, modified to only have one instance of a child of the base element of name contained in key_to_sub_search.

"getAttribute" = statement which will lead the evaluator to return a value contained by an attribute. This attribute will be an attribute of the base element of the name contained in key_to_sub_search.

key_to_sub_search = A string which will be used by getElements/getAttribute as an evaluation key.

3.4.3 ##(two hash) statement

A ## statement indicates that a method call is to be made through the Java Runtime Environment. The method call between the hashes is made in the normal way as per normal in Java.

For operational reasons all methods must rest in the designit.generator package.

The system at the moment is configured to recognise nested ## and will not recognise consecutive ## statements.

The rules for a double hash statement are as follows:

It is assumed that there is only one double hash statement per line with no other single or triple hash statements following it.

The parameters for the method can consist of

-any number of parameters

-any parameter in the form of a # statement.

-any parameter in the form of a ## statement.

-any parameter in the form of normal text / normal text + # statement / ## statement.

3.4.4 ### (three hash) statement

The purpose of the triple hash statement in this system is to allow the systematic processing of a number of similar elements.

By using the "EXECUTE_WITH" keyword another template will be processed in association with an array of modified documents evaluated through the getElements.

This template is specified by the text immediately proceeding the statement.

Effectively for one use of a triple hash statement you can generate any number of components on the same level of a document.

A feature of the triple hash statement is the ability to create navigation shortcuts. The shortcut is represented by a key and value in the added to the system wide hashtable stored in the Generator class.

The key to be added is the string contained between the first single hashes in the opening for statement.

The value added is the string contained in the second single hash statement, minus the "getElements" command statement.

Wherever the key is encountered in the execution of a generation it is replaced by its accompanying value.

At the closing triple hash the key / value are removed from the hashtable and are not available for future use as a shortcut.

It is also allowed that within a definition of a key or value that an already existing shortcut can be used.

The syntax for a triple hash statement is as follows:

triple hash statement = *opening_triple_hash* + |*execution_statement*| + *closing_triple_hash*

opening_triple_hash = "###FOR #" + *shortcut_key* + "# IN #"+*shortcut_value* + "####"

shortcut_key = string to be added to hashtable as the key.

shortcut_value = Dual purpose statement representing text to be added as value to hash table and also defining a single hash statement to be evaluated for an array of modified documents.

*execu-
tion_statement*="###EXECUTE_GENERATION_WITH"+*template_for_generation*+"###".

template_for_generation = String representing the file location of a template file to be used in the creation of source code - one file being created for each modified document returned by opening_triple_hash.

closing_triple_hash = "###ENDFOR #"+*shortcut_key*+"#".

3.5 JavaGeneratorHelper

The JavaGeneratorHelper class existed to provide methods for use in the generation of Java Source Code. These methods were accesed by text contained in a template, and called through the template processing system in the FileGenerator. Only templates used in the generation of Java code would reference any method in this class.

4 Implementation of Generator

The implementation of the project was entirely through Java SDK 1..3 edition. Development was preformed mainly using JBuilder rapid development tool. The operating system used through out the project was 'Windows NT 4.0'. Elaboration and justification of the processes used in the project is presented in detail in the schedule section.

4.1 System Initialisation

Entry into the generation is through a command line argumentised format as described earlier. The main purpose of the generator is the validity checking of the user input and subsequent trigging of the generation process. A master input buffered Reader for the initial template file is created from user argument. The master parsed XML document is created using the XercesJ Parser [AP00].

With all files prepared and in a valid condition a master instance of File Generator is created. The method process telling is called and is passed the master documents created at initiation.

4.2 FileGenerator

The File Generator class is the central core of the generator system. At the hub of this class is the process Templates Method. Taking only a Buffer Reader input and a pursed document this quite short method reads line by line through the input.

This method is the decision engine of the system. Delegation to the case specific methods of the evaluate Triple/Double/Single hash is performed at the stage. These methods can sub delegate their evaluation amongst each other as required. Eventually an evaluated string is returned and written to the current output file.

4.2.1 Determination of choice of Output File

The current output file that is used by a process template call is determined by whether an output file has been created by the template being processed. If the firs line of a template is not a file output statement the default system output file is used.

4.2.2 Overview of Triple Hash

Evaluate Triple Hash is the most powerful of the methods used in the system. Within the Triple Hash statement it is possible to have one or more Triple Hash statements each requiring an a recursive call for processing. Each statement is represented by a block of the input file defined by the shortcut name used in this section.

It is important therefore that a new input reader is created for the recursively evaluated block with only the appropriate code for the block being sent. When a evaluate triple hash is called control of the parent input reader is taken. A temporary local copy is made for the entire section of the FOR block using a buffered String writer treating any inner FOR loops as plain text. An array of modified documents, each containing only one instance of the node described by the text after 'IN', is created by a call to evaluate single hash.

With an individual buffer reader of the block of FOR block text along with one of the modified documents from the array, process templates is called.

Process templates will then scan through the input making recursive calls to produce triple hash on any inner FOR blocks that may be present.

When Evaluate Triple Hash is called with the first line of the input reader containing "EXCUTE_GENERATION_WITH" a new generator sequence is triggered.

A new file input reader is created for the file named template file along with a new instance of FileGenerator. The process template method is called being passed the new template reader and the current pursed XML document.

This starts the whole processing procedure as new upon the template file.

4.3 Test Strategy

At all stages through the development process was it possible to verify the current code using client supplied templates and documents. Apart from some minor changes made by the developers, these samples were a solid base for judging success.

Templates supplied varied in complexity to allow the creation at first of a simple generator, and then to incrementally improve and build in extra functionally into the code.

Of great help to the developers was the facility within J-Builder to monitor and track changes to variables through a debug session, with a watch placed on a variable or object the developers could determine at which points bugs were occurring and make the requited modifications needed.

As the system moved further towards completion, and the command line interface was being used more, extra reliance was placed upon the ability to print to the standard output. This proved confusing at times, but on the whole was seen as an important strategy in the detection of bugs.

At the completion of the system, a full test suite of samples was provided by the client. These tests comprised of three groups:

4.3.1 GROUP 1

Simple tests ranging from templates with only plain text to those with 1 to 2 hashes. Also varied was the setting of the output path for allowable file descriptions.

4.3.2 GROUP 2

These comprehensive tests were more akin to the type of operation likely to be encountered in generation. They ranged from ct1 which had a generation of a simple nature, to ct4 which was the generator of a full system with nested FOR statements and the use of all features included in the system.

4.3.3 GROUP 3

These tests were error tests checking for bad syntax, incorrect file names, missing data type dictionaries , etc. To pass these tests it was required to modify the existing code to allow more meaningful output to the user and to warn of possible invalid in any outputs already created.

A full detailed description of the tests are included in the appendix.

4.4 Task Delegation

Within the implementation, every attempt was made to delegate all evaluation, navigation operations on the parsed document/lines of text to external helper methods.

These methods on the whole have to be given names descriptive of the operations that they perform; i.e. 'get index of closing brackets' will get the index position of closing bracket, i.e. "(" when given a string and the index position of a corresponding closing bracket.

- Helper methods to be used across the system are stored as static methods in the Generator class.
- Methods used in the generation of code in Java from Java specific templates are stored as static methods in the JavaGenerator Helper Class.
- Methods used exclusively in FileGenerator are stored as instances methods in the FileGenerator.

4.5 BufferedStreams

Used extensively in the implementation of the Generator system were various forms of input and out put streams provided by the java.io package. I/O libraries often use the abstraction of a stream, which represents any data source or sink as an object capable of producing or receiving pieces of data. The stream hides the details of what happens to the data inside the actual I/O device. This allows us to treat each buffered reader / writer in exactly the same fashion. We can switch between the various types of streams with ease as essentially everything is being treated as a stream of characters.

4.6 Iterator

Used in a limited extent in the implementation of the Generator system (but widely used in the Runtime Environment) is the java.util.Iterator class. The built in java collections package is very comprehensive and quite confusing to deal with at first (see diagram below).

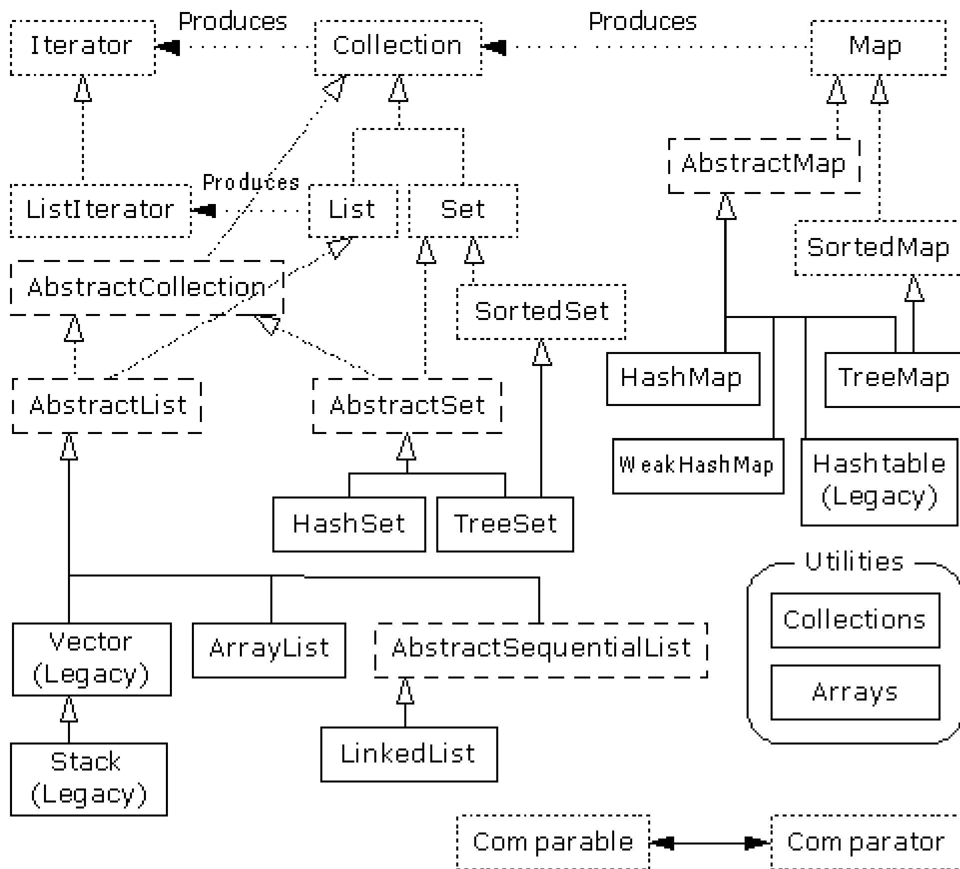

Figure 12:Java collections (Courtesy Bruce Eikel)

To allow a standard method of processing there is the Iterator class. All Collections can produce an Iterator via their Iterator( ) method thus allowing the traversal of any collection without regard to type. This allows the programmer to take advantage of the benefits of different types of collections and process them in a universal manner.

5 Conclusions

This project started with the end goal being a generator system being created for the output of fully working Java compatible code from an input of a series of specification documents. It is clearly the case that this target has been reached and the end goal achieved.

The integration of the full generation system complete with runtime environment and a sample Componentware system was also achieved successfully.

It is possible for a developer to use this rapid development tool by following the allowable syntax for templates to create templates for code generation which along side the accompanying XML parseable document gives a system compatible for use with the supplied runtime generator.

The user can run this code through the Runtime. Environment. and send messages calling methods, create and delete interfaces/components and manipulate and view the system using the viewing panel available in the designate panel package.

Throughout the project many difficulties and obstacles were faced by the developers regarding hardware/software concerns.

In the initial stages the problem of writing in a laboratory used by many users was over come by the developers own personal hard-discs. This strategy provided a success and provided a stabile working environment for the duration of the project.

The choice of Java as the project programming language can also be seen as a success. Whilst the API documents at times provided a sketchy and incomplete help menu, associated user groups and related books came to the developers rescue at times.

Java 1.3 has a wide range of collections and utilities. Such as the variety of buffered income streams and output streams available which when understood fully can provide a tailored solution to a problem with only a limit amount of developer coding required. This depth of ready made classes was probably one of the reasons the project could have been completed in the limited time frame available.

An evaluation of JBuilder and WinCVS is a little less straight forward than the previous case. Whilst both packages had many advantages (as outlined earlier in the report) these were balanced by the negative problems the developers had with bugs and inconsistencies in both these pieces of software.

In the case of JBuilder one feature of the software was is a built in intelligence that speeds up compiling by merging old code with new. In the latter stages when wholesale rearrangement of the system structure was being undertaken JBuilder coped dismally with the changes, outputting complied code which bore no relation to the current state of the source code.

The inconsistency caused by JBuilder for a time cast doubt over the success of the system with the test cases and in the final stages of the project this was more than small cause for concern.

In the future I would not use JBuilder unless I was assured that this problem has been veritably an completely solved.

In the case of WinCVS problems were encountered with committing and retrieving mainly it appeared due to out of date or missing reposition data files on the local disc. Whilst these

problems had an apparent reason, actions made by the developers inconsistent to the principles of normal CVS operation, there were no features for recovery provided by WinCVS with a small addition of a partial module checkout operation, option or other similar feature, CVS would be greatly improved and this developer could then use the system in confidence.

The system could be improved by:

- Improving the syntax allowable in template files and extending the allowable combinations.
- Improving the double hash evaluator to use a variety of packages.
- Renaming a number of methods and classes to a more uniform meaningful standard.
- Further testing to prove the stability of the developed code.

It is planned that the system will be developed further by another group of students after the completion of this phase. These developments will advance the while generator model to a new level, adding user aids and extra system functionality. Some of the developments possible include changing the system to a distributed model to be implemented through java RMI and allowing the potential for components to reside over many machines. In addition to this it is planned to add an OCL interpreter module to generate the calculated attributes. This in conjunction with a CASE tool can provide a total visual solution to the generator, allowing transfer direct from a user friendly GUI describing the specification to a fully working Java system.

References

[AP00] Apache XML Project. www.apache.org

[BBR+00] Klaus Bergner, Manfred Broy, Andreas Rausch, Marc Sihling, Alexander Vilbig. A Formal Model for Componentware. In Foundations of Component-Based Systems, Cambridge University Press. 2000.

[BDD+92] Manfred Broy, Franz Dederichs, Claus Dendorfer, Max Fuchs, Thomas Gritzner, Rainer Weber. The Design of Distributed Systems – an Introduction to FOCUS. Technical Report TUM-I9203, Technische Universität München. 1992.

[BGR+99] Klaus Bergner, Radu Grosu, Andreas Rausch, Alexander Schmidt, Peter Scholz, Manfred Broy. Focusing on Mobility. In Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences (HICSS 32), IEEE Computer Society. 1999.

[BHRS97] Klaus Bergner, Franz Huber, Andreas Rausch, Marc Sihling. Component-Oriented Redesign of the CASE-Tool AutoFocus. Technical Report TUM-I9752, Technische Universität München. 1997.

[BJ95] M.Broy and S.Jähnlichen, editors. KORSO: Methods, Languages and Tools for the Construction of Correct Software –final Report, volume 1009 of LCNS. Springer, Heidelberg, Nov 1995.

[BJR98] Grady Booch, Ivar Jacobson, James Rumbaugh. The Unified Modeling Language User Guide. Addison Wesley Publishing Company. 1998.

[BKR98] Klaus Bergner, Karsten Kuhla, Andreas Rausch. Schnelle Schichten: Transparenter Zugriff auf ODBMS über CORBA. iX No. 11. 1998.

[BNR+96] Frank Buschmann, Regine Neunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern-Oriented Software Architecture, A System of Patterns. John Wiley & Sons.. 1996.

[Broy97] Manfred Broy. Towards a Mathematical Concept of a Component and its Use. In Software-Concepts and Tools 18, pp. 137-148. 1997.

[BRS+99] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig, Manfred Broy. A Formal Model for Componentware. In Proceedings des 9. GI/ITG-Fachgespräch Formale Beschreibungstechniken für verteilte Systeme, FBT `99, Herbert Utz Verlag. 1999.

[BRS97] Klaus Bergner, Andreas Rausch, Marc Sihling. Using UML for Modeling a Distributed Java Application. Technical Report TUM-I9735, Technische Universität München. 1997.

[BRS98a] Klaus Bergner, Andreas Rausch, Marc Sihling. Componentware – The Big Picture. In Proceedings of the International Workshop on Component-Based Software Engineering. 1998.

[BRS98b] Klaus Bergner, Andreas Rausch, Marc Sihling. A Component-Oriented Architecture for the CASE-Tool AutoFocus. In Proceedings of the IASTED Conference on Software Engineering, ACTA Press. 1998.

[BRS99] Klaus Bergner, Andreas Rausch, Marc Sihling. AutoFocus – ein CASE-Tool für verteilte System. In Erfahrungen mit Java: Projekte aus Industrie und Hochschule, dpunkt Verlag. 1999.

[BRSV00] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig. Putting the Parts Together – Concepts, Description Techniques, and Development Process for Componentware. Proceedings of the 33th Annual Hawaii International Conference on System Sciences, IEEE Computer Society. 2000.

[BRSV99c] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig. An Integrated

View On Componentware – Concepts, Description Techniques, and Development Process. In Proceedings of IASTED Conference on Software Engineering, ACTA Press. 1998.

[Flan99] David Flanagan.Java in a Nutshell : A Desktop Quick Reference (Java Series). O'Reilly & Associates, 1999.

[FORS00] FORSOFT: Bayerische Forschungsverbund Software-Engineering (FORSOFT). http://www.forsoft.de/. 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison Wesley Publishing Company. 1995.

[Grif98] Frank Griffel. Componenteware. dpunkt Verlag. 1998.

[HMR+98] Franz Huber, Sascha Molterer, Andreas Rausch, Bernhard Schätz, Marc Sihling, Oscar Slotosch. Tool supported Specification and Simulation of Distributed Systems. Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems, IEEE Computer Society. 1998.

[HS99] Peter Herzum, Oliver Sims. Business Component Factory. John Wiley & Sons. 1999.

[OH98] R. Orfali and D. Harkney, Client/Server Programming with Java and CORBA, John Wiley&Sons, 2nd ed.,1998.

[OMG97] Object Management Group (OMG). OMG Unified Modeling Language Specification (Version 1.1). http://www.omg.org, document number: 97-08-05.pdf. 1997.

[OMG99] Object Management Group (OMG). OMG Unified Modeling Language Specification (Version 1.3). http://www.omg.org, document number: 99-06-08.pdf. 1999.

[OOSE00] oose.de Dienstleistungen für innovative Informatik GmbH. Object Engineering Process (OEP). http://www.oose.de/oep/. 2000.

[Raus00a] Andreas Rausch. A Proposal for Software Evolution in Componentware. In Proceedings of the 4th European Conference on Software Maintenance and Reengineering, IEEE Computer Society. 2000.

[Raus00b] Andreas Rausch. Software Evolution in Componentware – A Practical Approach. In Proceedings of the 2000 Australian Software Engineering Conference, IEEE Computer Society. 2000.

[Raus00c] Andreas Rausch. Software Evolution in Componentware Using Requirements/Assurances Contracts. In Proceedings of the 22th International Conference on Software Engineering. 2000.

[Raus99] Andreas Rausch. Executive Summary: Software Evolution in Componentware – A Practical Approach. Proceedings of the International Workshop on Software Change and Evolution. 2000.

[Strou00] Bjarne Stroustrup, The C++ Programming Language, Special Edition, Addison-Wesley Pub Co; 2000.

[Szyp98] Clemens Szyperski. Component Software. Addison Wesley Publishing Company. 1998.

[WK98] Jos B. Warmer, Anneke G. Kleppe. The Object Constraint Language: Precise Modeling With UML. Addison Wesley Publishing Company. 1998.

Appendices

A. Project Schedule

Number	Description	Scheduled	Responsible	Workers	Deliverables	Status
WP 1	**Setting up the Project Environment**	- 12.6.				**finished**
WP 1.1	Getting Login, Configuration of standard System: EMail, WWW, Office, etc.	- 28.5.	all	all		finished
WP 1.2	Setting up the Project WWW Page	- 28.5.	Andreas	all	first initial web pages in repository and on the web. Each of the web pages has found someone who is responsible for it	finished
WP 1.3	Download, Install & First Contact with WinCVS & CVS Checkout DesignIt Repository and make your own sub directory under ~designit/test/<YourName>	- 30.5.	Daire, Karen	Daire, Karen	create your own personal folder	finished
WP 1.4	Download, Install & First Contact with J2SE: Realize the Hello World Program with J2SE. Let it run and check it into the repository under your name	- 30.5.	Daire, Karen	Daire, Karen	Code in personal folder in repository	finished
WP 1.4	Download, Install & First Contact with JBuilder: Set up a Project file and let the Hello World run there	- 31.5.	Daire, Karen, Lucien	Daire, Karen, Lucien	Code + Project File in personal folder in Repository	finished
WP 1.5	Download, Install & First Contact with J2EE & RMI. Each should implement the small RMI sample: RMI Count: See Orfali	- 2.6.	Daire, Karen, Lucien	Daire, Karen, Lucien	Code + Project File in personal folder in Repository	finished
WP 1.6	Run the Breakplanner Application: Using a new common directory in the DesignIt Repository. Run the Breakplanner in this directory.	- 6.6	Daire, Karen, Lucien	Daire, Karen, Lucien	Code + Project file in common folder in repository	finished
WP 1.6a	Create The GUI's required in the specification. Link to specification file	- 11.6	Daire, Karen, Lucien	Daire, Karen, Lucien	Gui and Batch files in Delivery Folder	finished
WP 1.7	Download, Install & First Contact with TogetherJ: Re-engineering of the Breakplanner Code. Add-	- 26.6.	Daire, Karen, Lucien	Daire, Karen, Lucien	TogetherJ Model File	finished

	ing a new simple statistic view.					
MS 1.1	Project Environment is built up.	26.6.	all	all	**1.** Running BreakPlanner application **2.** BreakPlanner JAR Files **3.** Java-Doc Files **4.** TogetherJ generated Word Documenation **5.** Installation, configuration, **6.** Java-Files in the development directory **7.** TogetherJ-File in the develop-ment directory Structure of the Project WWW Pages is fixed	reached: 27.6.
WP 2	**Design & Implementation of the Breakplanner in the new Componentware Fashion**	27.6. - 21.7.				**finshed**
WP 2.1	Initial Specification of the new CW-Breakplanner	- 27.6.	Andreas	Andreas		finished
MS 2.1	Presentation of the CW-Breakplanner Specification	27.6.	Andreas	all	Documentation of the CW-Breakplanner Specification, Presentation of the Specification	reached: 27.6.
WP 2.2	Design and Implementation of a small dummy sample	27.6. - 7.7	Daire, Karen, Lucien	Daire, Ka-ren, Lucien	Design & Im-plement the Con-trol-Pannel, the Engine and a small Sample with two Com-ponents:	finished
MS 2.2	Dummy Sample runs or is finished	7.7.	all	all	TogetherJ Model File + Code	finished
WP 2.3	Desing & Implementation of the new Breakplanner	8.7. - 21.7.	Daire, Karen, Lucien	Daire, Ka-ren, Lucien	Design & Im-plement the BreakPlanner with the new Engine	finished
MS 2.3	CW-Breakplanner is run-ning	21.7.	all	all	TogetherJ Model File java files, class files, documenta-tion, install documentation, a more detailed project plan of	reached: 1.8.

					WP3, Updated content of WWW pages	
WP 3	**Design & Implementation of the DesignIt Tool**	7.8. - 25.8.				**finished**
MS 3.1	Presentation of the Requirements of the Code-Genrator	7.8.	Andreas	all	XML DTD for the Specification, Sample Specification, Sample Output of the Code-Gernerator, Sample Configuration-File of the Generator	finished
WP 3.1	Implementation of the Code-Generator for Attribute-Classes	7.8. - 11.8.	all	all		finished
MS 3.2	Attrubute-Classes can be generated	11.8.	all	all		finished
WP 3.2	Implementation of the full Code-Generator	14.8. - 23.8.	all	all		finished
MS 3.3	The whole Code can be generated out of the specification	23.8.	all	all		finished
WP 3.3	Integration of the Code-Generator with the DesignIt-Runtime Environment	24.8. - 25.8.	all	all		finished
MS 3.3	The whole DsignIt Tool is running and ready for delivery	25.8.	all	all		finished
WP 4	**Test & Documentation of the DesignIt Tool**	28.8. - 1.9.				finished
MS 4,1	Outline of Project Dissertation	1.9	all	all		finished
MS 4.2	Draft of MainSection	4.9	all	all		finished
Ms 4.3	Completed Main Section	8.9	all	all		finished
MS 4	All done and delivered	8.9.	all	all	Delivery of the DesignIt Tool, including: java files, class files, documentation, install documentation, updated WWW-Site	reached

B. Generator Testing

Simple Test

Test number	Test Description	Status
ST1	Generate one file without any #-tags, directory exists, but file does not exist	Passed
ST2	Generate one file without any #-tags, directory and file do not exist	Passed
ST3	Generate one file without any #-tags, directory and file al-ready exist	Passed
ST4	Generate one file with #-tags	Passed
ST5	Generate one file with #-tags, more than one #-tags in one line	Passed
ST6	Generate one file with ##-tags	Passed
ST7	Generate one file with ##-tags, more than one ##-tags in one line	Passed

Complex and Full Test

Test number	Test Description	Status
CT1	Generate components for breakplanner	Passed
CT2	Generate interfaces for breakplanner	Passed
CT3	Generate attributes for breakplanner	Passed
CT4	Generate whole breakplanner	Passed

Error Test

Test number	Test Description	Status
ET1	Calling the generator with wrong arguments: XML-file does not exist	Passed
ET2	Calling the generator with wrong arguments: TPL-file does not exist	Passed
ET3	Generate one file without any #-tags, directory and file already exist	Passed
ET4	Calling the generator with wrong arguments: TPL-file and XML-file do not exist	Passed
ET5	Calling the generator with wrong arguments: XML-file is not valid against DTD	Passed
ET6	Calling the generator with wrong arguments: TPL-file is buggy	Passed
ET7	Calling the generator with wrong arguments: XML-file is not valid and TPL-file is buggy	Passed
ET8	Calling the generator with wrong arguments: TPL-file is buggy	Passed
ET9	Calling the generator with wrong arguments: TPL-file is buggy	Passed
ET10	Calling the generator with wrong arguments: TPL-file is buggy	Passed
ET11	Calling the generator with wrong arguments: TPL-file is buggy	Passed
ET12	Calling the generator with wrong arguments: TPL-file is buggy	Passed