

DesignIt

Componentware Runtime Environment

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

Karen Lavery

15. September 2000

Acknowledgements

I wish to thank my supervisor, Andreas Rausch who provided guidance throughout each stage of the project. In addition the efforts of the other members of the DesignIt team; Daire Kivlehan and Lucien Hoogendorn who contributed to the overall success of the project.

Financial support was provided by a European Social Fund Training Award for the duration of the year, and an additional Erasmus Socrates mobility grant administered by the International Liaison Office of The Queens University of Belfast enabled the project to be undertaken at Technische Universität München.

Finally I wish to thank my parents, Peter and Geraldine Lavery for their support throughout, both financial and otherwise.

Abstract

The modularity of a Componentware approach to software development encourages the move from large systems to modular structures, facilitating the reuse of independent components. Although the advantages of these systems are numerous, at present a development tool does not exist to aid in the production of component-based systems. The aim of this project is to develop a rapid prototyping tool, which can generate the relevant code from a series of specification documents and provide a runtime container for the code to be executed. This document focuses on the development of the Container Runtime Environment.

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
2.1.1	Hardware.....	7
2.1.2	Programming Language.....	7
2.1.3	Borland JBuilder 3.5 Foundation Edition	7
2.1.4	Concurrent Versions System CVS.....	8
2.1.5	TogetherJ	9
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 COMPONENTWARE RUNTIME ENVIRONMENT	18
3.1	User Interface.....	18
3.1.1	A simple application	18
3.1.2	BreakPlanner Sample.....	18
3.2	Overall design.....	20
3.3	System Description	20
3.3.1	Singleton Pattern.....	21
3.3.2	Proxy Pattern.....	21
3.3.3	Semaphore Pattern	21
3.3.4	Engine Processes.....	22
4	IMPLEMENTATION OF COMPONENTWARE RUNTIME ENVIRONMENT	23
4.1	Performance of Operations.....	23
4.2	Processing of Messages.....	24
4.3	Implementation Details	26
4.3.1	Java Features.....	26
4.3.2	Semaphore Implementation	26
4.4	Testing and Integration	29

5 CONCLUSIONS 30

REFERENCES 31

APPENDICES 33

A. Project Schedule.....33

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.

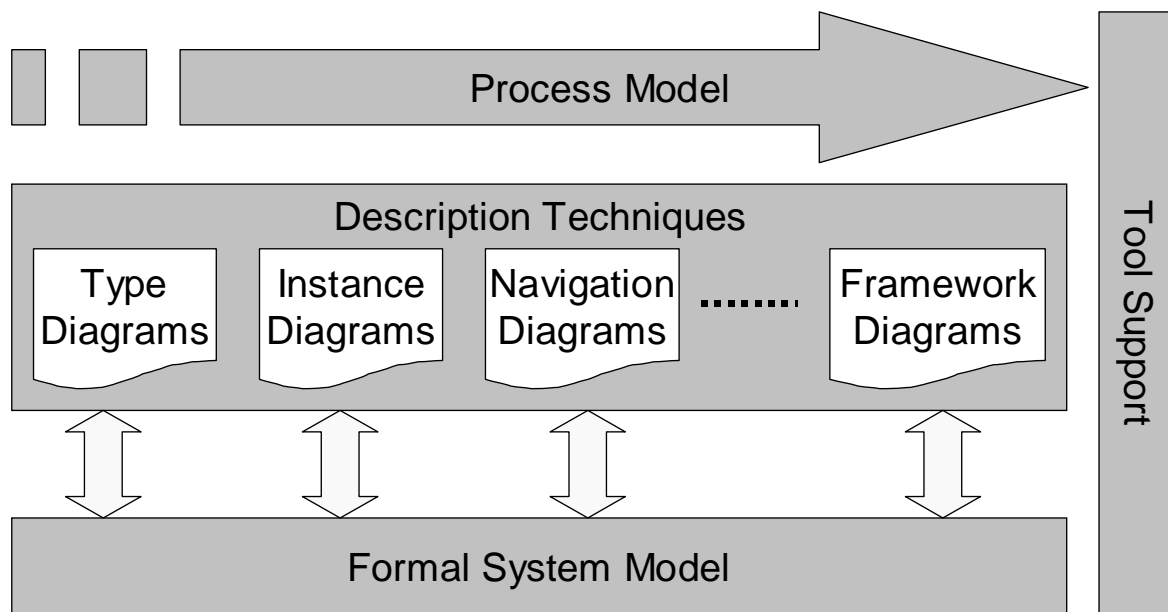


Figure 1: Componentware Methodology

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.

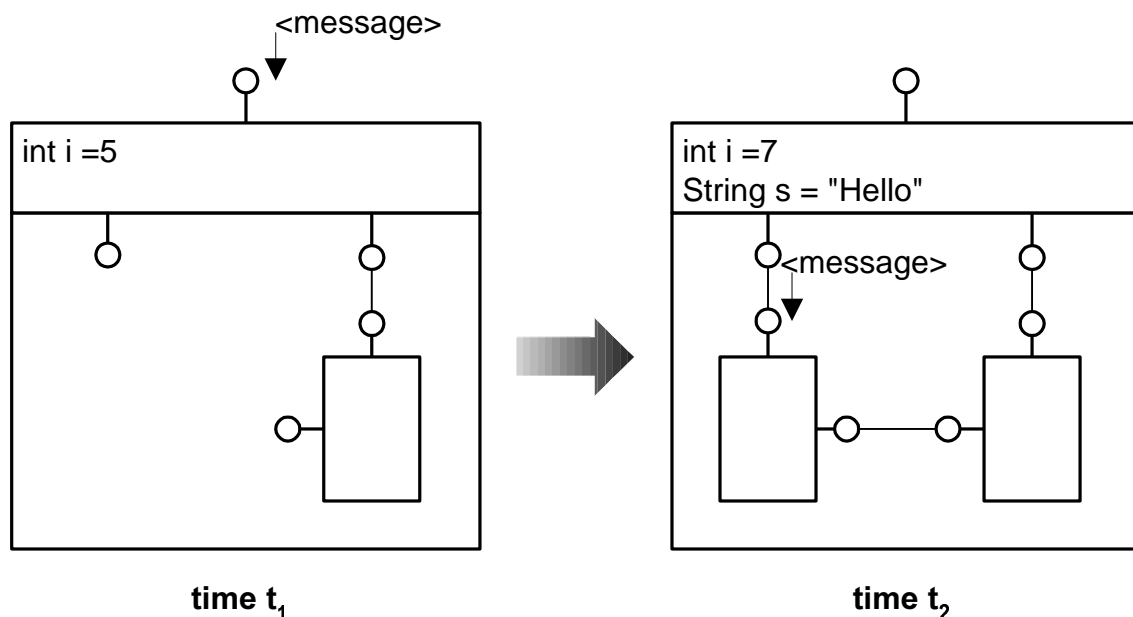


Figure 2: Basic Concepts and Elements of a Component-Oriented System

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 t_1 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.

The third Chapter discusses the architecture and design of the Runtime Environment. An introduction to the core design of the system is provided with detail of the relationship between the various elements (components, interfaces, attributes, connections, operations and messages). This provides a background for the understanding of the structure of the entire system, which is also described in detail. The user interface and certain aspects of the system design are explained, including the use of several design patterns and the functioning of the engine itself.

The details of the implementation of the Runtime Environment are included in Chapter 4. The specific operations of the system are dissected and analysed. The use of Java features are also explained and justified. A brief description of testing and integration is also included.

The final Chapter includes a summary of what has been achieved in relation to the aims outlined in Chapter 1. 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 deleter 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 compliant 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 taken to ensure that the team used as many tools and project aids as was prudent. The achievement of familiarity with one's 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. 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.

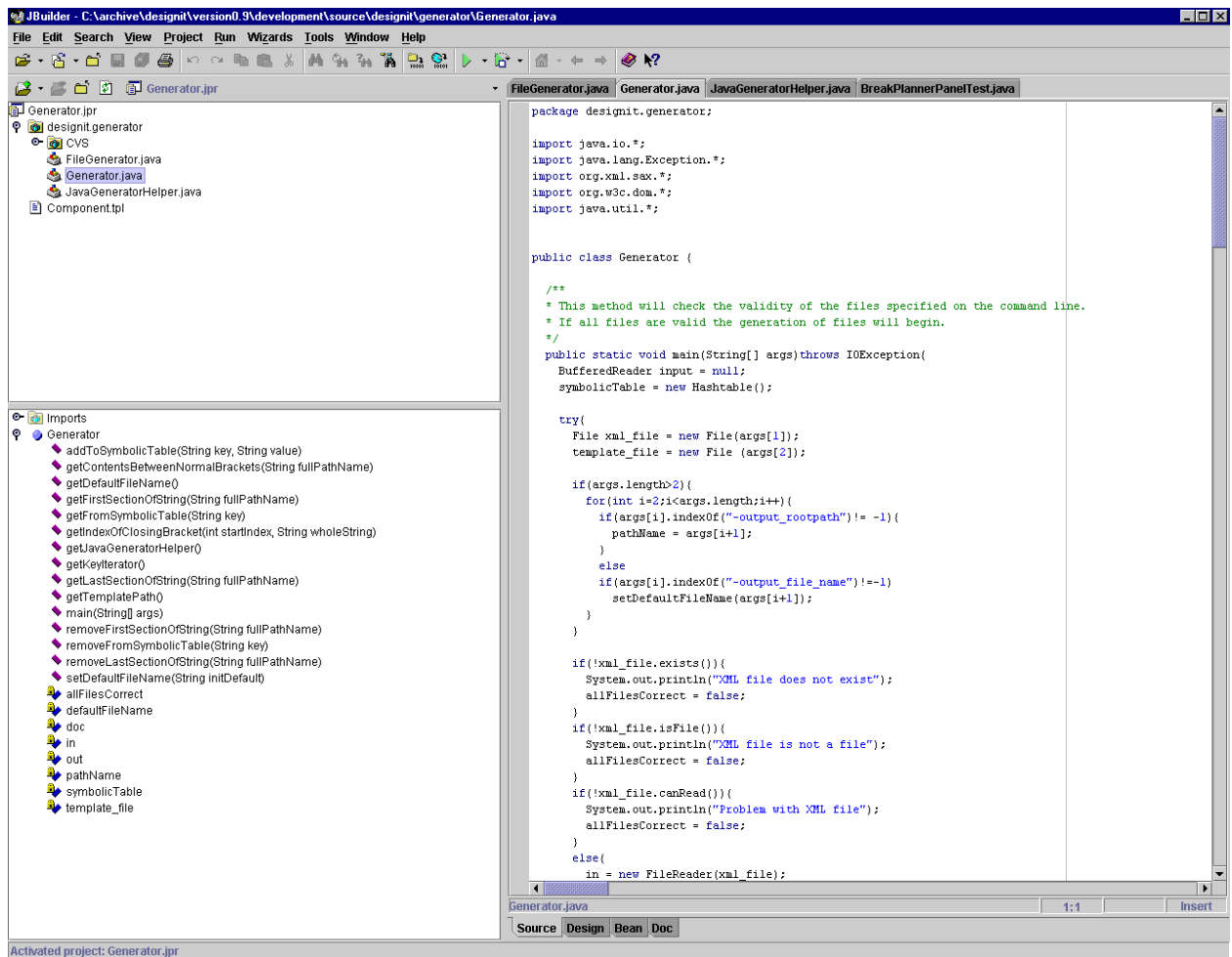


Figure 2: JBuilder Screenshot

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.

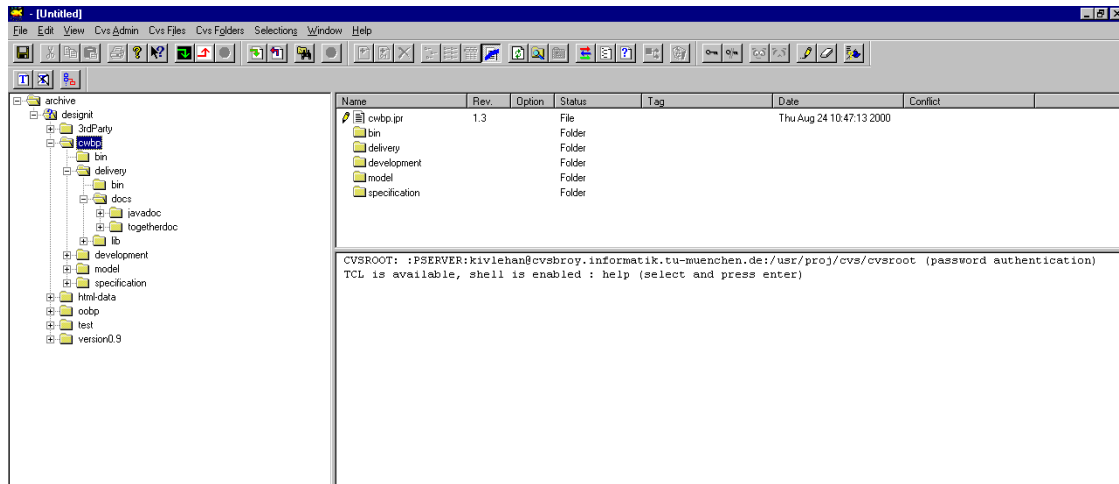


Figure 3: CVS Screenshot

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 from 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 conflict 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. Teachers 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.

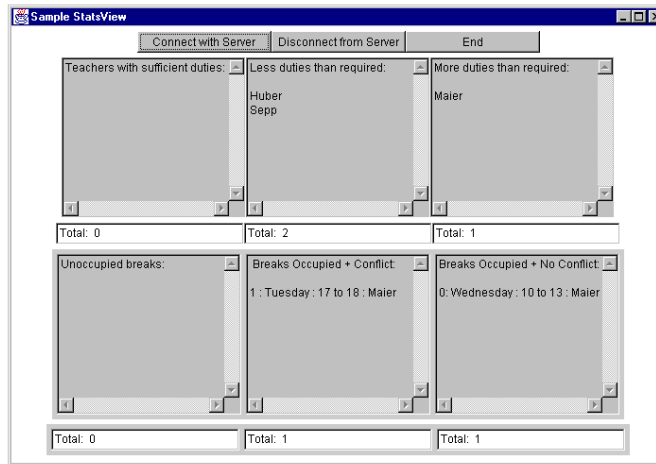


Figure 4: The graphical interface of the Statistics View

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".

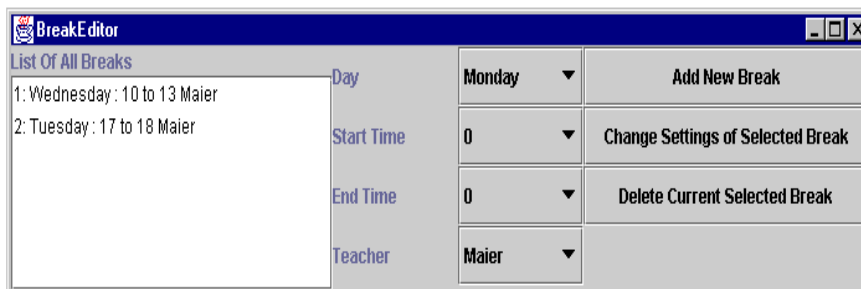


Figure 5: The graphical interface of the Break Editor

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

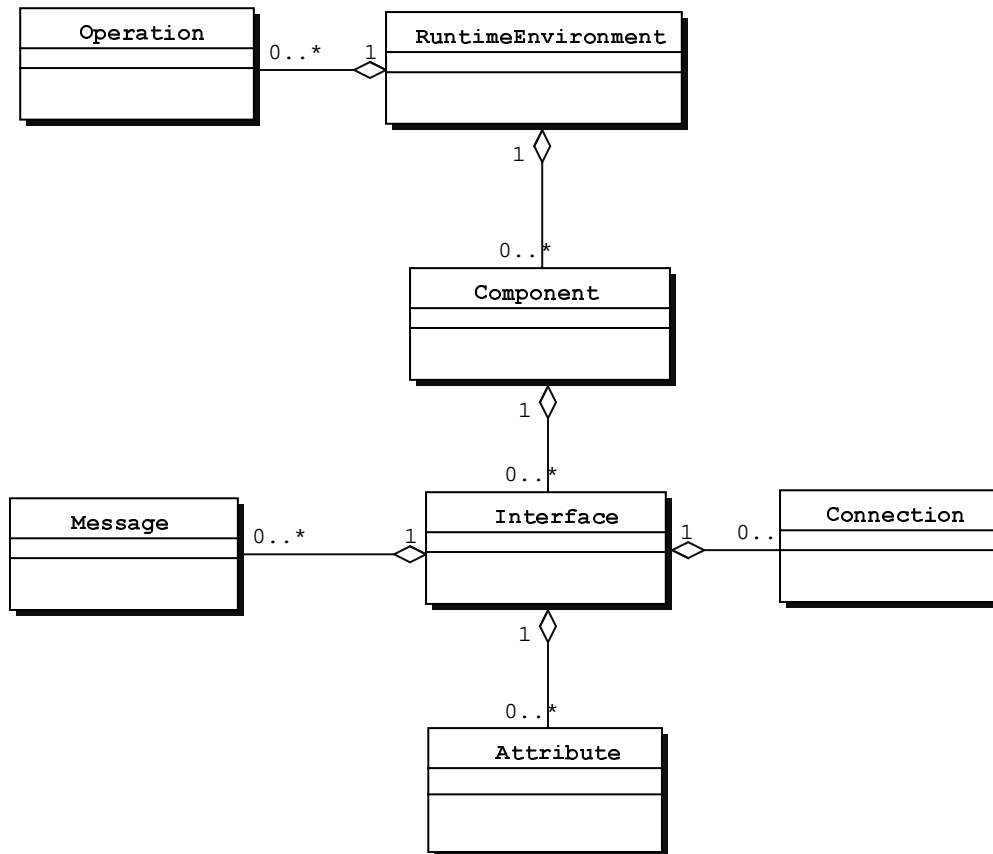


Figure 6: Core system structure

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 component's 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 from `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.

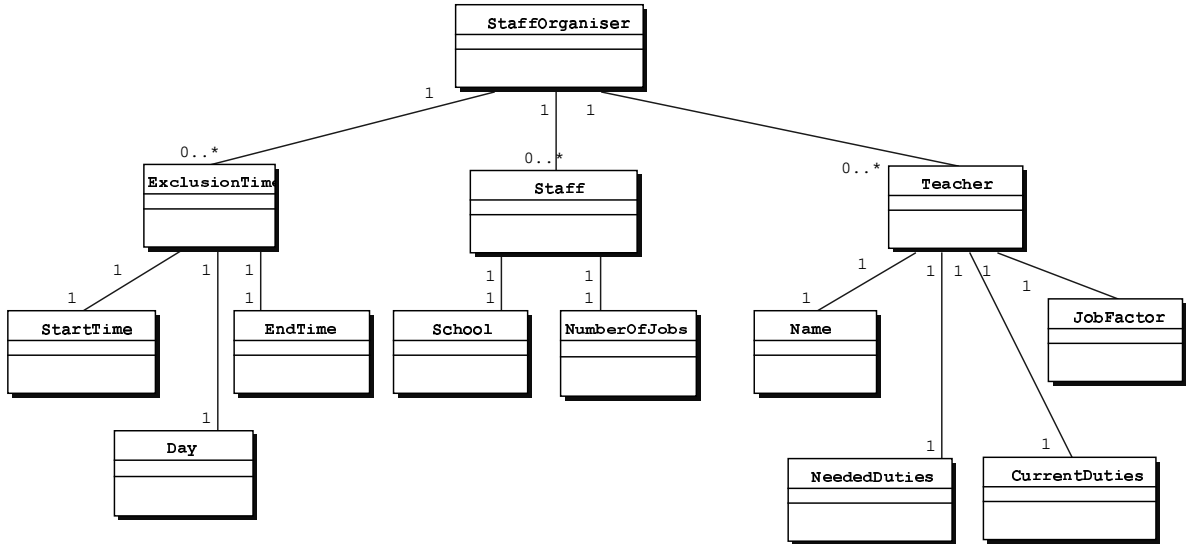


Figure 7: Structure of Staff Organiser Component

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.

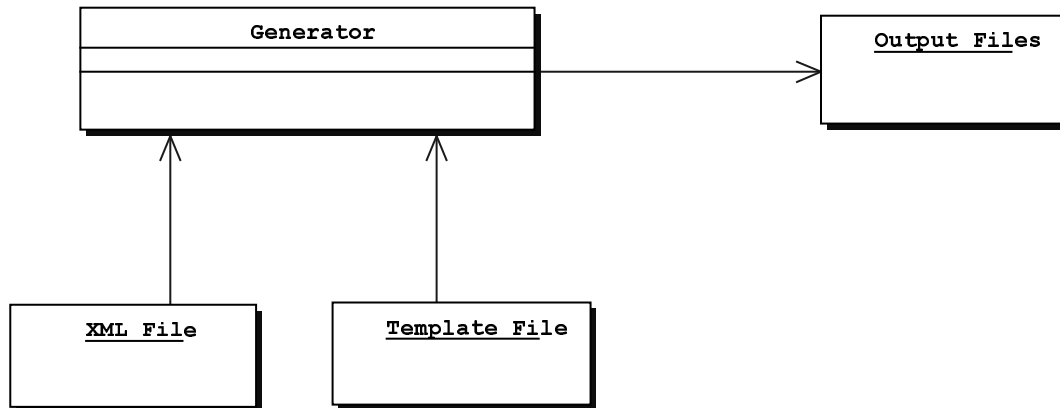


Figure 8: Overview of Generator System

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.

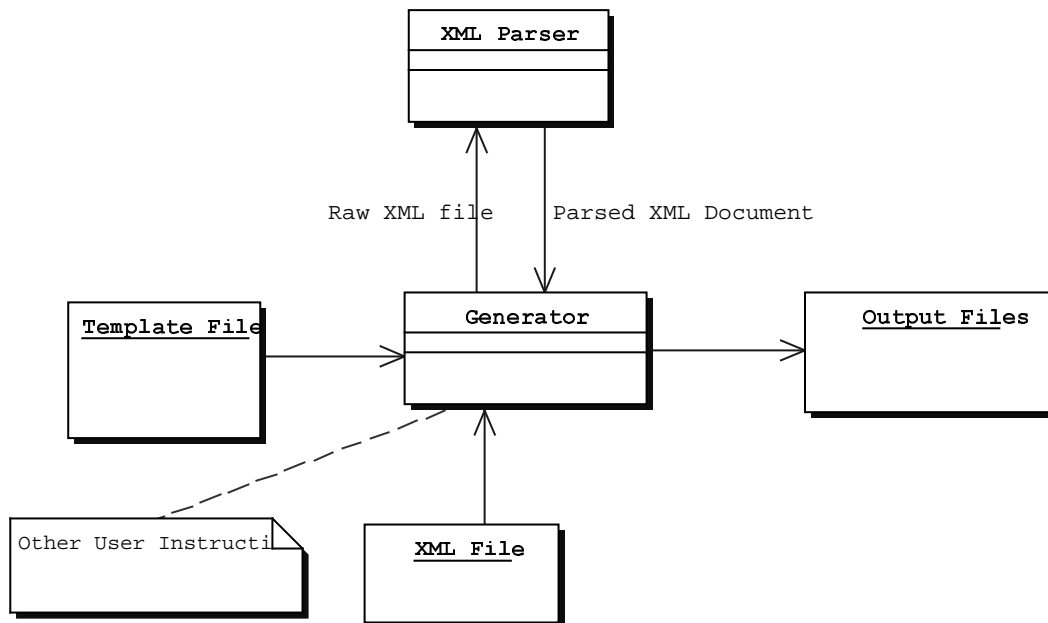


Figure 9: Interactions of Finished System

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.

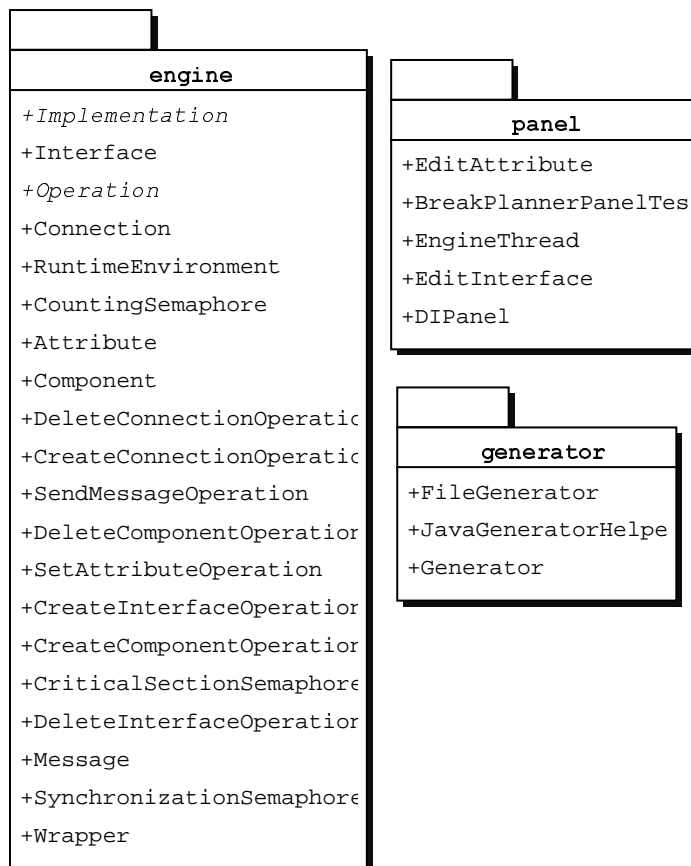


Figure 10: High Level Architecture Class Diagram

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 Componentware Runtime Environment

3.1 User Interface

The Design It Panel developed by Lucien Hoogendorn, another team member, represents the developers' access point to the system. The panel is an opportunity for the developer to view the existing state of the system.

3.1.1 A simple application

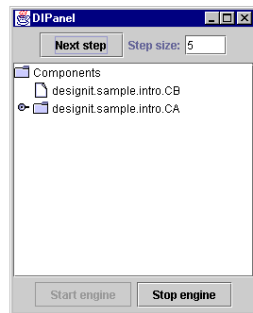


Figure 11: The Design It Panel showing the output of IntroInitializer

The initial data is established using a test program such as IntroInitializer. The results of the creation of components and interfaces can be viewed in the panel. The result of subsequent messages can also be viewed.

The present state of the panel allows the user to interact with the existing components and interfaces. The main aim of the panel is to show the developer that the creation process has been successful.

The initial state of the panel shows the components of the system. When a component is selected (by double – clicking) the connected interfaces are revealed. In the same way the features of the interfaces can be viewed.

The panel provided has been extended to enable some interaction with the engine. The Edit-Interface panel was added to enable messages to be sent to existing interfaces. For example, in the intro sample, messages (foo, foo1 etc.) can be sent to interface IA of component CA.

The message will only be executed if the name and parameter types and values are valid for the method. At present the user interface only accounts for a range of primitive data types to be sent as parameters but this could be easily extended to include components, connections etc. The panel is not a main concern of the project at this stage; it was only developed to a stage where it could be used to illustrate what was feasible in the system and to reveal if all the functions were working as intended.

3.1.2 BreakPlanner Sample

The preliminary data including the creation of components, interfaces and attributes needs to be established using a test program such as BreakPlannerInitializer. An additional feature not utilised by the intro sample is the representation attributes and connections. The addition of the EditAttribute panel enables the values of the various attributes to be manipulated.

With contrast to the previous object-oriented breakplanner system, instead of an advanced graphical representation of the status of breaks etc the various issues involved are shown in a different manner. The panel was not intended for use by an administrator in a school and therefore does not require the same ease of use. The panel is more of a development tool for the testing stage. It requires knowledge of the system before the user can find the information they are looking for. An example is illustrated below.

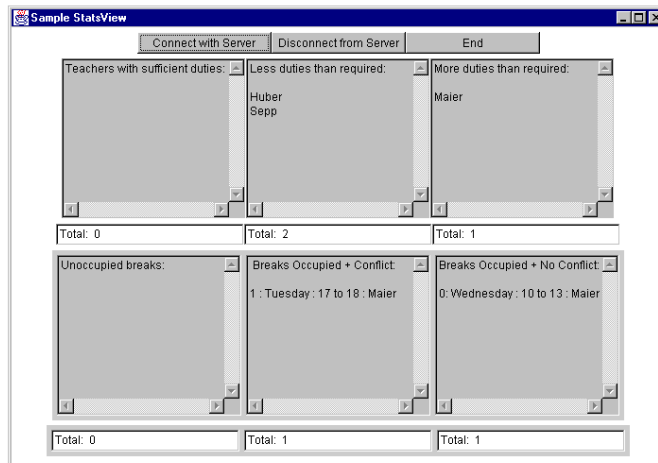


Figure 12: The graphical user interface of the object-oriented breakplanner

The old system clearly shows when a break has a conflict or is unoccupied.

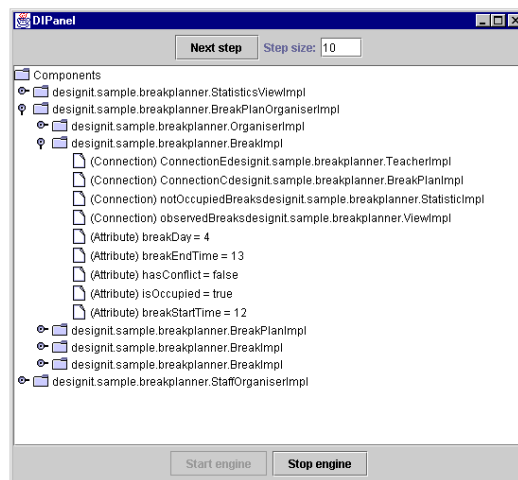


Figure 13: The representation of data in the component-based system

In contrast the new system contains the relevant information, but here it is displayed in a subtler manner, as a connection between the break and the StatisticsImpl interfaces, with the label revealing the nature of the relationship.

3.2 Overall design

The overall engine design can be viewed in the class diagram below:

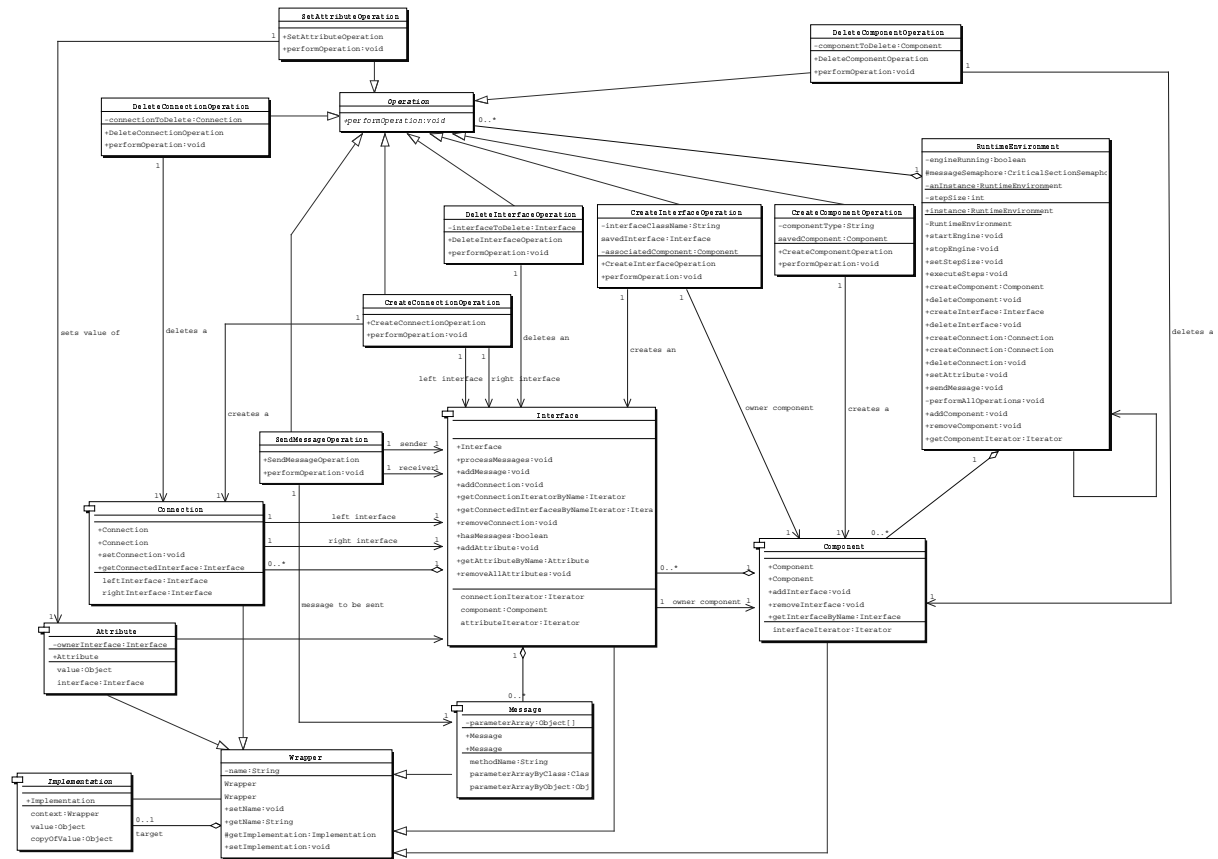


Figure 14: Structure of the engine package

3.3 System Description

The main building blocks of the system surrounding the Runtime Environment are components. Each component has one or more interfaces, which act as access points. Each interface can have one or more attributes, which hold an item of data concerning the interface. In the implementation of the sample system it was necessary to create different classes for each attribute of each interface. However as they are all attributes via inheritance they can be created and processed in a uniform manner. The structure builds on the outline illustrated in Section 2.

Interfaces can be connected to other interfaces via a connection, which may or may not indicate the purpose of the connection. For example, a connection between a teacher interface and a `StatisticImpl` interface indicates the current status of the teachers' duties. Similar connections exist between the `ViewImpl` and the break interfaces to indicate the status of the break.

Interfaces also have a Set of messages. Each message contains the name of the method to be called on the interface and the parameters necessary for the execution of the method. When `processAllMessages` is executed on the Runtime Environment it iterates through the Set of components. For each component the Set of interfaces contained are iterated through to check if they contain any messages.

3.3.1 Singleton Pattern

The Runtime Environment is the central feature of the engine. An important feature of the system is that only one instance of the Runtime Environment should exist at any one time. Several reasons made this a necessary feature. The engine, the actual processing mechanism within the Runtime Environment, uses a lot of system resources and several instances would be a strain on the system. Several systems are not required to process the messages and operations when one is sufficient and desirable. In addition the Runtime Environment needs a complete picture of all components in the system for the process of sending messages etc. If several instances existed, each would only have a partial view of the system, making them unable to take full advantage of all the features offered by the system. The Runtime Environment acts as a gateway to all existing components, able to navigate from the Set of components to their interfaces and attributes. In this way the Runtime Environment is responsible for creating and deleting all elements and maintaining a consistent view of the overall system. This ensures the integrity of the data and prevents operations and elements being duplicated.

All operations are executed via the Runtime Environment. An operation is any one of several creation or deletion functions of the system or the transfer of messages. The various different types extend the operation class, which merely states that a method to perform all operations, the method that does the actual work of the operation, should be included. This is to facilitate the processing of the operations in the engine. As all can be declared of type operation, the Runtime Environment can use this generality to perform operations in a uniform manner, thus simplifying the task.

3.3.2 Proxy Pattern

All elements, which are created or deleted at run-time, extend the wrapper class. This is an implementation of the Proxy Pattern [GHJV95]. The wrapper of each newly created object provides a placeholder for the object, which cannot be, accessed itself because it has not been computed yet. This feature is necessary due to the deferred processing of operations in the system. When an operation is called on the Runtime Environment it is not executed immediately, the Runtime Environment merely creates an operation of the specific type requested by the user. At this point only the data necessary to perform the operation is saved within the newly created operation, which is added to the List of operations contained within the Runtime Environment. These operations will only be processed when the user of the panel, or a test program starts the engine.

3.3.3 Semaphore Pattern

A primitive Semaphore pattern [GHJV95] was used to provide synchronization in the system and control access to the List of operations contained within the Runtime Environment. To ensure that each operation is only executed once, when the Runtime Environment enters the critical section of processing the List of operations, the List is frozen to prevent other processes from altering the list. If another part of the system attempts to access the List during this phase, for example to add a new operation, they will be forced to wait until the critical section releases the hold on the List. In this case a local copy of the List is made, the original List is emptied, and returned to the system for use. The copy of the List is then used to process the operations, thus making the original list unavailable for only a short period of time. This needed only to be implemented for SendMessageOperation, as this is the only type of operation created at run-time. All other operations are created at the initial setting up of the

system (e.g. creation of components etc) and will not be added to the List of operations once the engine is running. If the system was improved to allow creation and deletion of these elements at run-time, the Critical Section Semaphore needs to be implemented at the point where the List of operations is accessed.

3.3.4 Engine Processes

It was initially thought that the engine itself, the processing part of the Runtime Environment would consist of a continual while loop, constantly performing operations and processing messages. This was a sufficient solution for early sample program where all operations etc were defined in the test program and no further commands were sent to the Runtime Environment when the engine was running. However, as the system was developed and further examples added it was realised that the engine continued to run even when there were no further tasks to complete. In addition, this continual processing prevented the system from accepting additional commands at run-time. It proved difficult to develop a method of passing control back to the user when the engine had no further operations to perform. A member of the team devised the concept of enabling the user to specify a step-size before starting the engine, which specifies the number of times the while loop is executed, processing messages and performing operations. It was not a sufficient solution merely to alternate processing power between the engines processes and the addition of new operations as this depended on the number of actions the engine has to execute, as specified by the user.

4 Implementation of Componentware Runtime Environment

The implementation of the project was entirely through Java SDK 1.3 edition, with JBuilder used as the main development tool. The following sections discuss the operation of the Runtime Environment in detail; focusing on how operations are performed and messages are processed. The implementation of the Semaphore Pattern briefly mentioned in Section 3.3.3 is discussed in more detail in addition to a justification of the use of several Java features.

4.1 Performance of Operations

As stated in the specification all operations and communication occur via the Runtime Environment, whose central role in the system ensures an up-to-date view of the components and interfaces in existence. All operations are stored until the engine (within the Runtime Environment) is started. The user of the Design It Panel starts the engine. An initial configuration of the system should be established in a test program such as BreakplannerInitializer which creates the initial components and interfaces and can also set some initial attribute values. The necessary connections are also created at this stage.

The panel user sets the step-size thus indicating the number of times the while loop is executed. When the engine has finished processing the user is able to carry out manipulations on the newly created objects resulting from the operations.

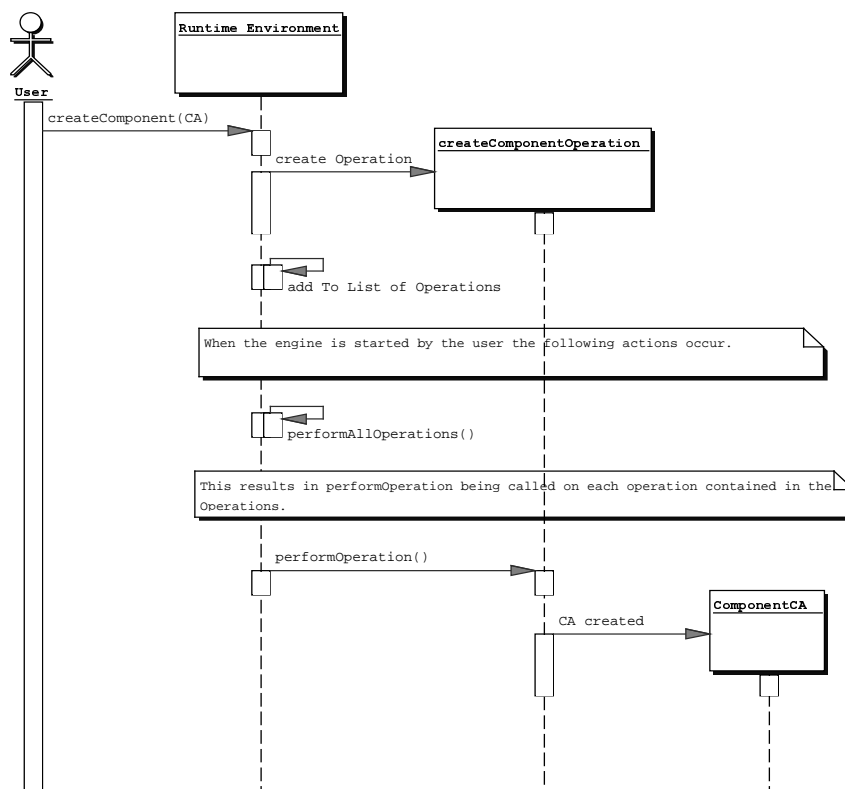


Figure 15: Performance of Creation Operations

When the user initiates a creation operation, for example to create a component, a method, createComponent, is called on the Runtime Environment (Figure 15). At this stage an empty component and a wrapper are created. A CreateComponentOperation is defined and added to

the List of operations, which are to be executed when the engine is running. At this stage the component specified has not been created, thus making the existence of the wrapper class a necessity.

At present all creation and deletion operations are created through test programs, although the initial sample (intro) reveals that it is possible for such operations to be created at run-time. When the user starts the panel and sets the step-size to a sufficient level to allow all the necessary processing to take place, the elements can be viewed in the panel window.

When the engine is started for each loop `performAllOperations` is executed. A local copy of the List of operations is made and the initial List is cleared and released for further additions. For each operation in the List, `performOperation` is executed. This results in the `performOperation` method specific to the type of operation being performed. For example, if the operation is to create a component, the `performOperation` method in `CreateComponentOperation` is called. The use of inheritance with regard to all operations inheriting from class `Operation` enables the system to dynamically type the operation at run-time and execute the appropriate method.

In `CreateComponentOperation` the data necessary for further processing of the operation had been established when the operation was created in the Runtime Environment. When `performAllOperations` is called, the `performOperation` method in `CreateComponentOperation` is executed to perform the actual work of creating the component from the component name, referring to the Java file, which reflects the type of component to be created.

An empty component of no specified type is created. The class for the specific component is retrieved using the `Class.forName(interface type)` function provided by Java. The process then attempts to get the appropriate constructor for the class using the parameter types supplied.

The next phase in processing is to get the Implementation for the component. A new instance of the specific component class is created using the saved component from the earlier operations. The implementation is used in the final phase of the component creation.

4.2 Processing of Messages

The next stage is for all messages on interfaces to be processed. If any `SendMessageOperations` were contained in the List of operations processed by the Runtime Environment in the step above the appropriate message will have been added to the Set of messages on the specified interface. Before committing processing power to the task of processing messages the Runtime Environment checks if any interfaces have messages to be processed. If messages exist the Runtime Environment proceeds by calling the `process` function on each message.

When `processMessages` is called on each interface the following actions occur. If the interface contains messages to be processed, for each message the system attempt to execute the method of that name. The `java.lang.reflect` package was used to achieve this. This provides classes and interfaces for obtaining reflective information about classes and objects. Reflection allows programmatic access to information about the fields, methods and constructors of loaded classes, and the use reflected fields, methods, and constructors to operate on their underlying counterparts on objects.

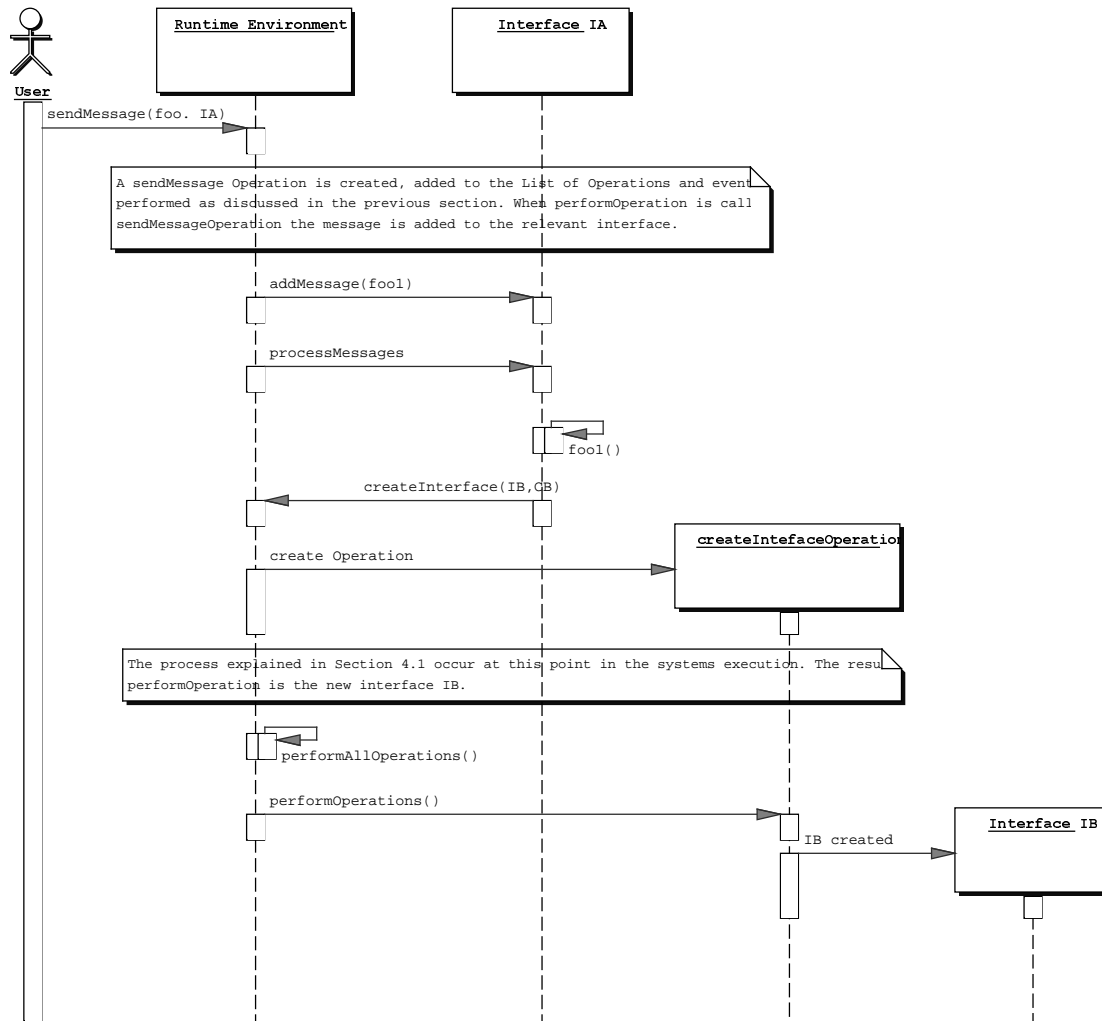


Figure 16: Message processing in the Runtime Environment

The user selects interface IA in the Design It Panel and sends message foo1 without parameters (Figure 16). A sendMessage command is sent to the Runtime Environment, which creates a sendMessageOperation. When the engine is started by the user performAllOperations is called. This results in the message being added to the Set of messages on interface IA.

The Runtime Environment checks the interfaces of all components in the system for the existence of messages. It then calls processMessages on each message. When foo1 is processed a createInterface command is sent to the Runtime Environment. On the next loop of the engine processAllOperations results in the creation of interface IB on component CB. One of the strengths of Componentware is this ability to dynamically alter the system structure at runtime through the creation of additional items.

4.3 Implementation Details

Several features of Java were exploited to provide functionality in the system, while a simple implementation of the Semaphore Pattern was used to provide control over access to the features.

4.3.1 Java Features

The Runtime Environment was linked to the system structure through a Set of components. This was implemented using a HashSet. This ensured that no duplicate elements existed in the collection.

The Hashset class offered basic operations (add, remove, iterator) useful for our purposes. As it provided an Iterator of the Set this meant we did not have to implement the function ourselves. It makes no guarantees as to the iteration order of the set in particular, it does not guarantee that the order will remain constant over time although this was not an important consideration for the purpose of holding components, interfaces and attributes.

The Runtime Environment contained a Set of components. These components contained a Set of interfaces which in turn possessed a Set of attributes, connections and messages. The features of the HashSet made it easy to traverse through this structure.

A List was used to store the operations to be processed by the Runtime Environment. It was necessary to use a List as the order in which the operations are processed needed to be controlled. Operations must be processed in the order they have been specified. This is because the result of an operation may be required to process an operation specified later in the List. For example, an operation to create a component must be executed before an operation to create an interface on that particular component can be completed successfully. The List gives the advantage that new operations are added to the end of the List while in the processing phase, operations are processed starting with those at the top of the List.

The main work of the engine is done by a simple while loop, controlled by the step-size(as explained in the previous section). Due to the way in which operations and messages are stored and processed only simple commands are necessary to do the actual processing work. The specific functions are held in the individual operation classes, and can all be accessed in a common way.

4.3.2 Semaphore Implementation

As has already been discussed in Section 3.3.3, the Semaphore pattern was required to control access to the List of operations in the Runtime Environment and the set of messages on each interface. It was necessary to implement this feature to ensure the integrity of the data within the system.

The simple implementation CriticalSectionSemaphore is merely a system whereby a certain block of code can be accessed if the value of the variable count is equal to one. The system is initialised in the Runtime Environment with a value of 1.

The system checks the status of the control variable 'count' by calling the method P. The system only allows the process to continue if the value of count is 1. When the value is 1 and a process can access the data, the value of count is adjusted to zero preventing unauthorised access. If another process attempts to access the section the count is zero and the new process is forced to wait until the process currently underway relinquishes hold on the section. On

leaving the critical section, V is called increasing the value of count to 1 and notifying those waiting that the data can be accessed.

```

public class CriticalSectionSemaphore {
private int count = 0;

    public CriticalSectionSemaphore(int initialCount) {
        count_ = initialCount;
    }
    public synchronized void P(){
        while(count <= 0)
            try{wait();}catch (InterruptedException ex){}
        --count;
    }
    public synchronized void V(){
        ++count;
        notify();
    }
}

```

Figure 17 : Critical Section Semaphore

Within the Runtime Environment the messageSemaphore is used to control access to the List of Operations. To prevent the scenario where an operation is added while the current list is being processed a critical section is included.

```

// enter critical section
messageSemaphore.P();
List cloneLnkOperation = new LinkedList(lnkOperation);
lnkOperation.clear();
// leave critical section

messageSemaphore.V();
for (Iterator it = cloneLnkOperation.iterator(); it.hasNext(); )
{
    ((Operation)it.next()).performOperation();
}

```

Figure 18 : Implementation of Critical Section

The current status of the messageSemaphore is assessed using method P. If the process can continue(i.e. the value of count is 1 as no other process is accessing the List), a clone of the List is created. The original List is cleared and is released via the V method. The original List can then be accessed by other processes and the operations can be performed using the temporary List. Other processes wishing to access the List of operations, to add a new operation have similar critical sections.

```
// enter critical section
messageSemaphore.P();
InkOperation.add(newSendMessageOperation);
// leave critical section
messageSemaphore.V();
```

Figure 19 : Attempts to access the List are monitored

The implementation provides a simple way to ensure that data can only be accessed by one process at a time. It is used to control access to the List of operations on the Runtime Environment and the Set of messages on each interface.

4.4 Testing and Integration

The generator and Runtime Environment were developed separately. The Runtime Environment was developed first and was extensively tested at each stage of development, when new functions were added. Each new function was tested using the panel and various test programs used to configure the initial data contained in the system.

Many of the features of the system evolved as a result of discoveries made during these various tests. For example, the decision to use a List to hold the operations instead of a Set was due to panel testing. It was discovered that operations needed to be executed in a specific order. In addition, the step-size feature was developed after a discovery made during development and testing.

The generator was subjected to the same process of constant testing throughout development. The breakplanner code implemented in the component-based system provided the skeleton of the templates to be used in the generation of the code. Therefore the result of the generation function could be compared to the desired output.

The integration of the two parts of the system was not a major concern once it had been established that both individual aspects functioned well independently. As the generator code produced was based on the breakplanner sample previously implemented with the Runtime Environment, the final stage proved to be the least troublesome.

5 Conclusions

The aim of the project was to create a prototyping environment consisting of a Runtime Environment or container in which component-based code can be executed and a Generator which generates the required component-based Java code from a series of template files and specification documents.

Both parts of the system have been realised and function as intended.

A Runtime Environment exists at the centre of a component-based system, enabling sample programs adhering to the specified system structure to create and delete various elements in the system.

A Generator, which is able to generate sections of Java code based on specifications held in XML documents, has also been developed. The code generated is compilable and can be run using the Runtime Environment.

The Runtime Environment can create and delete the various elements of the system using the code generated. The panel enables the user to interact with the system. In the sample implemented only sending messages to interfaces and changing the values of attribute data are possible. However, the Design It Panel could be easily expanded to account for the creation and deletion of components etc at runtime. The Runtime Environment is capable of such actions (as indicated in intro) and only a small addition to the user interface would make this possible.

The generator generates the code as specified. Although the system work sufficiently for the sample used it would be possible to extend the range of circumstances the system can deal with, enabling it to be used for a wider range of data. For example, the various combinations of hashes are restricted in number, yet an improved evaluation technique may make an extension possible.

The generator is not capable of producing the code necessary to carry out some of the calculations on the attributes in the system. At present this code is added manually as XML is not powerful enough to provide an adequate specification. In the future the calculations will be specified using OCL. An OCL interpreter is already available and will be added to the system to provide the code necessary to create the calculations.

Eventually the system will be distributed enabling components existing on different hosts to form part of the one system. The Runtime Environment will continue to play a central role, responsible for the performance of all operations.

Another possibility is the addition of a simulation environment to provide the system with the same ease of use as the original breakplanner system. Existing GUIs could be integrated to facilitate interaction with the system.

References

- [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 Develop-

- ment 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. Componentware. 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 Re-engineering, 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 Documentation 5. Installation, configuration, 6. Java-Files in the development directory 7. TogetherJ-File in the development 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.				finished
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, Karen, Lucien	Design & Implement the Control-Pannel, the Engine and a small Sample with two Components:	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, Karen, Lucien	Design & Implement the BreakPlanner with the new Engine	finished
MS 2.3	CW-Breakplanner is running	21.7.	all	all	TogetherJ Model File java files, class files, documentation, install documentation, a more detailed proiect 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-Generator	7.8.	Andreas	all	XML DTD for the Specification, Sample Specification, Sample Output of the Code-Generator, 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	Attribute-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 DesignIt 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