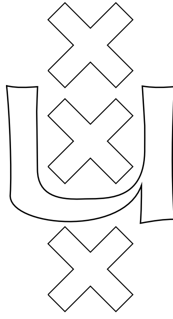

Reuse of ABN-AMRO PowerBuilder Applications

Bas Toeter
B.Toeter@uva.nl

July 2001

Abstract

This masters thesis describes a transformation process from PowerBuilder client/server applications to Java Three-Tier web applications. The transformation process operates on source code that progresses from PowerBuilder source code to Java source code in several phases. The rewriting techniques that are used include manual rewriting, lexical scanning and rewriting using XML and XSLT.

Contents

1 Introduction **3**
- 1.1 ABN-AMRO software management 3
- 1.2 ABN-AMRO Legacy PowerBuilder 4
- 1.3 Reusing PowerBuilder Software 4
- 1.4 Organization 5
- 1.5 Acknowledgements 5

2 PowerBuilder **6**
- 2.1 PowerBuilder client/server development 6
  - 2.1.1 4-GL Development Environment 6
  - 2.1.2 Object Orientation 6
  - 2.1.3 Client/Server Architecture 7
  - 2.1.4 Domain specific features 7
- 2.2 ABN-AMRO Programming method 8
  - 2.2.1 Technical Development Framework 8
- 2.3 Components of a ProDeca Application 8
  - 2.3.1 Human Interface Component 8
  - 2.3.2 Task Management Component 9
  - 2.3.3 Problem Domain Component 9
  - 2.3.4 Data Management Component 9
  - 2.3.5 System Interaction Components 9
- 2.4 PowerBuilder Implementation 9
- 2.5 Structure of a PowerBuilder application 10
  - 2.5.1 Objects 11

3 The Three-Tier model **13**
- 3.1 Intranet based Applications 13
- 3.2 Web architecture 13
  - 3.2.1 Application Servers 15
- 3.3 Requirements 15

4 PowerBuilder to Java **16**
- 4.1 Architecture 16
  - 4.1.1 Application Logic 17
  - 4.1.2 Internal State 17
  - 4.1.3 Graphical User Interface 17
- 4.2 Language Differences 18
  - 4.2.1 Object Orientation 19
  - 4.2.2 4GL vs. 3GL 19
  - 4.2.3 Conversion of Data Types 19
  - 4.2.4 Compiler and Virtual Machine 19

5 Software Transformations **22**
5.1 Transformation requirements 22
5.2 A process for Conversion 25
5.3 Obtaining Source Code 26
5.4 Transformation Strategy 26
5.4.1 Annotation 27
5.4.2 Rewriting XML with XSL 28
5.4.3 Rewriting XML annotated PowerBuilder Export files with XSL 32

6 Sample Transformation **34**
6.1 PowerBuilder to Java 34
6.1.1 Sample Application 34
6.1.2 Export Files 35
6.2 A Java implementation 38
6.2.1 PowerBuilder System Classes 40
6.2.2 Adding GUI functionality 40
6.2.3 Generating code with Stylesheets 42
6.3 Summary 44

7 Conclusions **45**
7.1 Results 45
7.1.1 Issues 45
7.1.2 Achievements 48
7.2 Future Work 48
7.3 Conclusions 48

A Sample Application PB export and Java code **51**
A.1 Application object PB Export code 51
A.2 Window PBExport code 51
A.3 mainwindow PBExport code with XML-markup 52
A.4 Java Implementation 53
A.5 Generated Java 54
A.5.1 Before Post-Procesing 54
A.5.2 After Post-Procesing 56

B Lexical Scanning Tool **57**
B.1 Source code 57

Chapter 1

Introduction

The banking industry has proved to be an early adopter when it comes to automation. Automation started in the back office where mainframe computer were first used to keep track of accounts. Later on the automation moved into the front office (tellers), the streets (ATM) and into the homes of customers (home banking).

The drawback of early adoption of automation is that at some point in time a certain software package may no longer be useful because it has reached the end of its lifecycle. All sorts of changes that can be of technological, social or political nature can be the cause of this.

This thesis reports on research we have conducted into the possibilities for the reuse of software that was about to reach the end of its technical lifecycle.

1.1 ABN-AMRO software management

In the past the ABN-AMRO has developed several branch applications for the management of services like travel insurance, currency exchange, etc. These applications are of a client/server nature in which the client resides on the office workstation and offers a user interface to a database that runs on a server elsewhere in the bank's computer network.

At present the ABN-AMRO is about to deploy a new standard for their office automation. This new standard involves an approach in which a web browser functions as the client for all office applications and information systems. So instead of many client applications the office workstation now only supports a web browser. The web browser provides the user interface for the application while in the back-end a web server is used as an interface to various databases. The combination of a browser, a web server and a database in a single application is referred to as the Three-Tier model.

By reducing the number of different software packages used in the office a cost reduction can be achieved which is one of the driving factors behind the intranet based approach. Another factor is the complexity of distributing new branch applications. This process can become so complex that it can literally be impossible to distribute an application among all the office workstations within the maintenance time frame.

A centralized approach that complies with the Three-Tier model solves both these

problems. However it also means that all client/server software that currently does not provide a web browser interface will have to be replaced.

1.2 ABN-AMRO Legacy PowerBuilder

The legacy client-server applications that this thesis describes were created using the PowerBuilder[1] client/server development environment from PowerSoft[2].

PowerBuilder allows the programmer to define objects which can inherit the methods and properties of other objects. ABN-AMRO has used this feature to create a proprietary service layer. The service layer consists of a set of objects from which a developer can create child objects that inherit the generic properties of an office application. These generic properties consist of features like the look and feel of the user interface, the database connection and support for error handling.

1.3 Reusing PowerBuilder Software

The object of the research we conducted is to determine how the PowerBuilder applications can be reused as a web application accessible through a web browser. ABN-AMRO prefers usage of a common language such as C/C++ or Java over less commonly used programming languages. We choose to use Java since it is broadly supported on web platforms.

Figure 1.1 shows the transformation process that is described in this thesis.

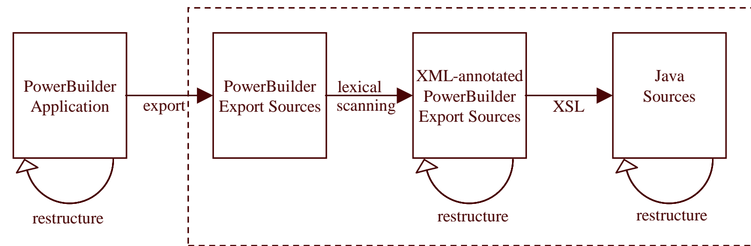

Figure 1.1: Transformation Strategy

The left side of the figure displays the PowerBuilder applications as they are before the transformation begins. In this form the applications are restructured to eliminate constructs and features that are difficult to transform to a Java web implementation.

The next step is to export source code from the PowerBuilder applications. These export files are annotated using XML-tags. Restructuring is applied to the annotated export sources to apply transformations from the PowerBuilder to the Java Syntax. Using an XSL-transformation these XML-annotated files are used to generate Java source code.

We have focused our efforts on transforming the object structure from PowerBuilder

[1] http://www.sybase.com
[2] PowerSoft has merged with Sybase in February 1995

to Java. We have not analyzed the PowerScript[3] code in the PowerBuilder applications.

1.4 Organization

The rest of this thesis is structured as follows:

Chapter 2 describes the details of PowerBuilder applications and the development method used by the ABN-AMRO.

Chapter 3 introduces Three-Tier inter/intra-net application technology.

Chapter 4 compares the PowerBuilder client/server paradigm to the Three-Tier Java web-application paradigm. A number of issues are listed that relate the transformation to architectural and language differences.

Chapter 5 elaborates on the approach sketched in Figure 1.1.

In Chapter 6 we present a sample transformation that follows the approach described in Chapter 5.

Finally we conclude this thesis in Chapter 7 which contains conclusions.

1.5 Acknowledgements

I thank my parents, sister, brother and brother in-law for their support in the past years and especially in recent times while I worked on this thesis. At the Centrum voor Wiskunde and Informatica I specifically would like to thank my supervisor Paul Klint, roommate and friend Jurgen Vinju, and Hayco de Jong. I would also like to thank Jan Boef and Joke Homan from the ABN AMRO for their support and for providing me with information and feedback. Further more I would like to thank Chris Verhoef of the Vrije Universiteit, Alan Berg of the Universiteit van Amsterdam and Remco van de Woestijne for their help and support.

[3]PowerScript is introduced in Chapter 2

Chapter 2

PowerBuilder

This chapter explains what PowerBuilder is and how the ABN AMRO has developed applications with it.

2.1 PowerBuilder client/server development

2.1.1 4-GL Development Environment

PowerBuilder provides an Object Oriented development environment for the development of client/server database applications. The programming environment consists of a graphical user interface in which most tasks can be completed using just the mouse. Many of the complex tasks that have to be performed by a client/server application are taken care of by PowerBuilder. This means that the programmer does not have to worry about establishing a connection to a database and retrieving certain records, these kind of low-level tasks are added to the application by PowerBuilder. The abstraction of low-level programming tasks is what makes PowerBuilder a Fourth-Generation Language, or rather a Fourth-Generation Programming Environment.

2.1.2 Object Orientation

Developing applications with PowerBuilder comes down to defining objects and their relations. The objects have properties, functions, procedures[1] and events. Objects can be either visible or invisible. Examples of visible objects are windows, menus, buttons and images. Examples of invisible objects are data storage objects, transaction objects and error objects.

Graphical Development

To create a user interface with PowerBuilder the programmer draws a window and adds GUI objects to it. A GUI object is a visible object that interacts with the end user. An example of such an object is a button. Like all GUI objects the button object has several properties such as: a label denoting the text on the button and a color property denoting the color of the button. Interaction can be added to the button by binding actions to events that relate to it. A button supports events like 'clicked' and 'double-clicked'. When the programmer adds an object to a window she subsequently has to fill in the properties and define actions to go along with the events. Because PowerBuilder declares the events and properties of the buttons,

[1] Functions and procedures are also referred to as *methods* in other Object Oriented Languages.

defining the properties and event code is a matter of filling in the blanks. The event code is written in a proprietary programming language called PowerScript [4].

Source code

The objects that the developer creates are stored in library files. A single application can consist of several libraries each containing one or more objects. The library files are in a binary file format. PowerBuilder features an export function that enables individual objects to be exported from the library files into plain text files. These files are quite readable and contain the object definitions.

Event Driven Programming Model

Virtually all the processing that is performed by a PowerBuilder application is triggered by an event that is generated by the user. There are however two exceptions. One exception is the timer event. This is triggered after a set amount of time has passed. The other exception is the application idle feature. This feature allows the developer to specify what code should be executed when the application has been idle for a certain amount of time. This is mostly used for activating application level security measures such as: expiring the user session, blanking the application window, or starting a password-protected screensaver.

2.1.3 Client/Server Architecture

PowerBuilder client/server applications differ from the traditional model of client/-server applications. A PowerBuilder application contains all the application logic while in traditional client/server applications the application runs on the server and the client functions mostly as a plain interface to the user. Invariably the server-side of the PowerBuilder applications under scrutiny in this thesis is a database. Although application logic may reside in the database in the form of stored procedures, its main purpose is to store data.

2.1.4 Domain specific features

PowerBuilder has many high-level features that are specific to the development of client/server database applications. These features make it easy to implement the user interface and the database connectivity.

Probably the most eminent of these high-level features is the datawindow. The datawindow automates the user/database interface. Its main role is to display data and allow the user to modify it. The datawindow object has two primary components: a data access component and a display formatting component. It uses a data source to access information. The data source can be a relational database such as Sybase, Oracle or DB/2. The datawindow offers a common front-end to these databases and deals with differences between then. Databases are not the only possible data source; it is also possible to use a text file as a data source.

The formatting of the data depends on the presentation style that is used. Build in presentation styles include: Freeform, Tabular, Graph and Cross tab. The datawindow provides a mechanism for formatting data that is retrieved from the database, or while it is being entered by the user.

2.2 ABN-AMRO Programming method

The ABN-AMRO uses their own set of methods and techniques, united in the ProDeca[2] specification, that aid the developer when building applications. The methodological part consists of the IAD-method[3] and falls outside the scope of this thesis. The technical part of ProDeca is introduced in the following paragraphs.

2.2.1 Technical Development Framework

The technical part of ProDeca is based on the products PowerBuilder and the PowerTOOL object class library. ProDeca can be applied to develop applications for two separate target platforms at the ABN AMRO.

The ProDeca application structure is based upon the PowerTOOL application structure with the addition of *Problem Domain Components*. Problem Domain Components will be explained in paragraph 2.3.3. PowerTOOL offers several templates consisting of objects that are cut out for development in the ABN AMRO infrastructure. Using these templates new objects can be created that offer application functionality to access the DB2 databases and to provide an integrated use of Problem Domain Components.

The ProDeca Service Layer is the combination of PowerTOOL features and service layer functionality. The service layer adds the following features:

- Support for ABN-AMRO GUI standards.
- Support for Problem Domain components.

2.3 Components of a ProDeca Application

An application that is being developed according to the ProDeca methods and techniques is decomposed into a default set of components:

- Human Interface Component (HIC)
- Task Management Component (TMC)
- Problem Domain Component (PDC)
- Data Management Component (DMC)
- System Interaction Component (SIC)

Implementations of these components are called objects. What follows is a description of each of the components.

2.3.1 Human Interface Component

All the interaction with the user takes place in the Human Interface Component. The interaction is shaped into 'interaction objects'. These objects usually define one or more windows. The user is able to activate one or more application *end functions* through the Human Interface Object. An application end function is defined as a function that is performed solely by the application without the need

[2]Dutch: Professionele Decentrale Applicatieontwikkeling - Professional Decentralized Application Development.

[3]Iterative Application Development.

for intervention by the user. During the execution of an application end function the control is passed to the application. An hourglass is displayed to indicate that the system is performing an operation that does not support input from the user.

2.3.2 Task Management Component

The Task Management Component(TMC) should implement all the application end functions that are supported by the application. The execution of tasks is delegated to the Problem Domain Component[4], the Data Management Component[5] and the System Interaction Component[6]. The PowerBuilder applications that fall within the scope of this thesis were based on a version of the service layer that does not support the TMC as a separate component. Due to this limitation, that is caused by the PowerTOOL architecture, the TMC has been integrated into the Human Interface Component.

2.3.3 Problem Domain Component

The Problem Domain Component contains all the 'Business Objects'. The implementation of the Problem Domain Objects(PDO) in the service layer defines the behavior of data in the application. The limitation is that one PDO can only define the behavior of one collection of data. The service layer does not support communication between PDOs. A service layer PDO can define the behavior of only one row of data at a time.

2.3.4 Data Management Component

The Data Management Component(DMC) takes care of the persistent storage of object data. This concerns storing as well as retrieval of object data to and from the Problem Domain Component. Outside of the DMC there is no object with knowledge of how this task is performed.

2.3.5 System Interaction Components

All interaction with other systems, such as: peripherals, remote computers, printers, etc. is handled by the System Interaction Component(SIC). By isolating the interaction with other systems in the SIC it is easier to adapt to changes in the interfaces of the external systems.

2.4 PowerBuilder Implementation

Some of the components described above have been implemented in PowerBuilder and form a basis from which developers can create applications. This basis is the ProDeca Service layer as mentioned in Section 1.2. It contains the Human Interface Component, Task Management Component, Problem Domain Component and System Interaction Component. Due to the event driven nature of PowerBuilder applications the TMC has been integrated into the HIC.

During the initial implementation of the ProDeca components an attempt was made to create a separate PowerBuilder object for each of the components. This resulted

[4]Introduced in Section 2.3.3
[5]Introduced in Section 2.3.4
[6]Introduced in Section 2.3.5

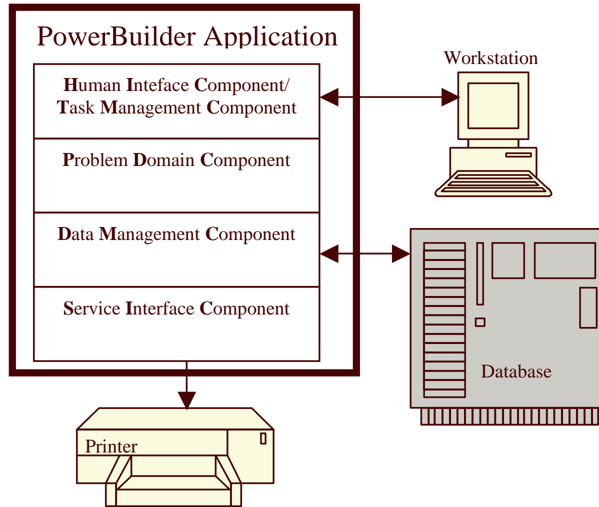

Figure 2.1: PowerBuilder Architecture

in the HIC and TMC being implemented in a PowerBuilder Window Object. The PDC and SIC have each been implemented in Non Visual PowerBuilder Objects. Figure 2.1 shows the individual components of a PowerBuilder application with their interfaces to other systems.

2.5 Structure of a PowerBuilder application

A PowerBuilder application is a collection of windows that perform related activities, such as order entry, accounting, etc. The application object is the entry point into the windows that perform these activities. It is stored in a PowerBuilder library just like any window-, menu- or function-objects that may be part of the application. Figure 2.2 displays a PowerBuilder library that defines the components of an application.

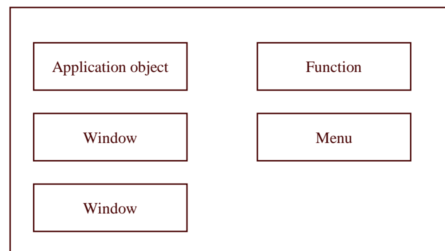

Figure 2.2: PowerBuilder library

The application object defines application-level behavior, and information about the application such as which libraries contain the objects that are used in the application, which text fonts are used by default, and what processing should occur when an application is started or terminated. When a user runs the application the PowerScript that has been written for the `open` event initiates the activity in

the application. A common practice is to use the `open` event to open a database connection and display the first application window.

2.5.1 Objects

During runtime execution, before an object can be referenced, it needs to be instantiated. The `open(window)` function instantiates the `window` that is being opened and any controls, graphical user objects or menus attached to that window.

Certain objects are defined and instantiated automatically at runtime. These are:

- SQLCA - SQL Communications Area.
- SQLSA - SQL Dynamic Staging Area.
- SQLDA - SQL Dynamic Description Area
- Message object
- Error object

System Object Classes

PowerBuilder maintains a set of system object classes from which user-defined classes are derived. The system object classes add functionality of a high level to PowerBuilder applications. Support for this functionality is built into the PowerBuilder Virtual Machine(PVM). A sample of such a feature is a datawindow. A datawindow is capable of running a database query and displaying the result set on the screen. The actual SQL query is generated by the PVM. It has knowledge of the database system and is able to generate queries for updating, retrieving and deleting data. So a lot of the functionality that makes PowerBuilder a fourth generation development platform is built into the virtual machine. Details of how they are implemented are hidden from the developer.

User objects and classes

A developer can create his own PowerBuilder classes and objects. By defining a new class it is possible to group a number of system objects into a single class. User objects are instantiations of user classes or system objects classes. Figure 2.3 displays the hierarchy of classes and objects that make up a PowerBuilder application.

Inheritance

PowerBuilder features single inheritance for a limited set of objects. Single inheritance implies that an object can only inherit properties from one parent object. However, the parent object may inherit properties from yet another parent object. Inheritance can be used with three PowerBuilder objects: windows, menus, and user objects.

Overloading

PowerBuilder offers the capability to create *overloaded functions*. An overloaded function is one that has several implementations under the same name. It is a concept that is very closely related to polymorphism. But instead of having functions in different objects with the same name, there are several implementations in the same object with the same name. At runtime, PowerBuilder figures out which

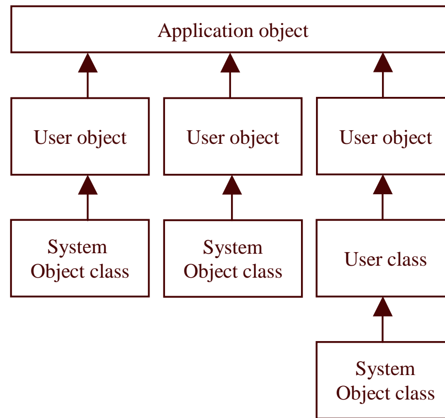

Figure 2.3: PowerBuilder class/object hierarchy

one to call by the number and type of parameters. Each implementation has to have a different set of parameters. They can have the same number of parameters but different types. They cannot, however, have the same parameters and types and a different return value. A return value alone is not enough to distinguish the function. See [4], chapter 22.

Reflection

Reflection, also called *introspection*, makes it possible to discover the fields, methods, and constructors of loaded classes at runtime. Besides this it is also possible to dynamically manipulate the fields, methods and constructors of a loaded class. Many Object-Oriented Programming Languages provide reflective features. However, PowerBuilder has no support for reflection.

Chapter 3

The Three-Tier model

As mentioned in the introduction the ABN-AMRO aims at reducing the variety of client applications that run in the office. They plan to remedy this diversity by deploying the web browser as the user interface to all information systems. The web browser has become a very common application that is found on almost every office workstation.

3.1 Intranet based Applications

Using a browser to view web content means that somewhere else in the network there must exist a web server for the browser to connect to. In a confined office environment this implies an intranet. An intranet implements Internet technologies and standards on a local network that is not connected to the Internet. Figure 3.1 displays a possible intranet configuration with support for shared storage, e-mail, shared printing and a web-server that interfaces with a database.

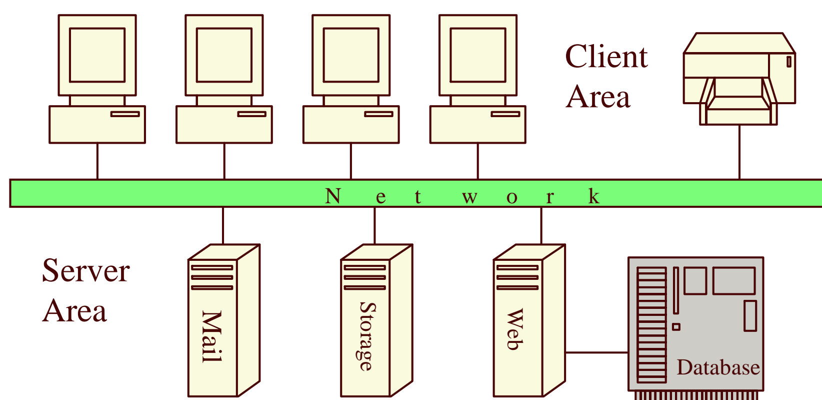

Figure 3.1: Intranet

3.2 Web architecture

The basic architecture behind the World Wide Web is the combination of a web browser, a communications network and a web server. The web server is idle until a client (web browser) requests an object from the server. The server inspects the type of the requested object, and if appropriate, sends it back to the client. All

communication is initiated by the client; the web server is stateless but keeps a log of clients and the objects they have requested.

Three-Tier model

To allow the web server to communicate with a database an application server is used as an interface between both systems. The combination of a web browser with a web server/application server and a database server is called a Three-Tier architecture. Figure 3.2 displays a model of the Three-Tier architecture.

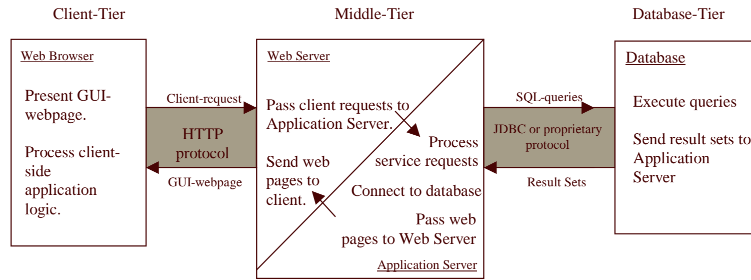

Figure 3.2: Three-Tier terminology

Next we introduce some terminology to discuss the Three-Tier model.

Middle-tier processing

This is the execution of application logic on the middle-tier. Middle-tier processing always results as a response to a client-request. The client request is directed at the middle-tier web server that passes it on to the application server. The processing may include communication with the database in which case a record set with information retrieved from the database could be processed. The middle-tier performs this processing and generates a GUI web page that is sent back to the client through the web server.

GUI web page

This is an information unit that can be visualized in the browser on the client-tier. It includes layout expressed in HTML and application logic expressed in a scripting language such as JavaScript[9]. Another common use for scripting languages in web pages is to create advanced graphical features that are not supported by HTML alone.

Client-tier processing

This is processing that is performed by the client tier. The processing is two-fold. It involves rendering of the layout-part of the GUI web page and the processing of the application-logic-part of the GUI web page.

Client-request

This is a request for processing by the middle-tier from the client-tier. The request complies with the HTTP protocol [8]. It contains the network address of the middle-tier web server and the name of the application server object that should service the request. It also contains variable name and value pairs that the browser specifies. These data pairs contain data that was entered into the web page by the user.

Database-request

This is a request from the application server to the database. The request is usually stated as an SQL-query[1].

Database-processing

This is the execution of queries that are received as database-requests. The result of a query is a record-set that is sent back to the application server.

3.2.1 Application Servers

A wide range of application servers exists. A typical application server can host several applications and has features that make it possible to develop advanced web applications. These features involve the integration of different backend database systems, transaction management, session management, load balancing and security.

One type of application server allows the developer to add server side scripts to the HTML-code. Examples of this technology are Active Server Pages®[2], ColdFusion®[3] and PHP3[4]. Database queries and result set formatting rules are specified in script elements that are embedded in the HTML-code. The application server processes these elements and replaces them with HTML-code. A high level of integration between layout and application is realized this way.

Another type of application server holds executable application objects that produce HTML-code. When a client requests an application page the related object is executed by the application server. The object generates an HTML-page, which is passed back to the browser via the web server.

Yet another type of application server functions merely as a gateway to stored procedures in the database. These database procedures produce HTML-code that is passed back to the browser through the application server and the web server.

The type of application server suitable for this project is one that supports Java as the programming language. We leave the choice of application server outside the scope of this thesis in order to focus on the transformation from PowerBuilder to Java.

3.3 Requirements

The ABN AMRO has two application language related requirements that are to be met by a Three-Tier system that is to replace their PowerBuilder applications. The first requirement, which was mentioned in the introduction, is that the language used for the applications is a mainstream language such as C/C++ or Java. The second requirement is that in the HTML-pages that are part of the user interface no dynamic components such as Java Applets[5] are used. Applets are considered insecure and therefore are not allowed in branch applications.

[1]Structured Query Language.
[2]Microsoft, http://msdn.microsoft.com/
[3]Macromedia, http://www.macromedia.com/
[4]Open source, http://www.php.net/
[5]A Java Applet is an embedded Java application.

Chapter 4

PowerBuilder to Java

There is more than one possible strategy to reuse the existing PowerBuilder Applications in a Java web application environment. A strategy for reuse may involve manual rewriting, automatic rewriting or a combination of both. A good strategy for ABN AMRO is one that leads to an acceptable solution at minimal cost.

This chapter explains some basic ideas behind transformations and focuses on the differences between PowerBuilder client/server applications and Three-Tier Java web applications. These differences lead to issues that have to be dealt with when transforming from PowerBuilder to Java and from the client/server paradigm to the Three-Tier paradigm. The issues identified in this chapter are discussed again in chapter 7 and are numbered to allow easy referencing.

The differences between PowerBuilder and Java Web Applications have been divided into two categories:

- architectural based differences
- and language based differences.

The next two sections describe these differences in detail.

4.1 Architecture

PowerBuilder Applications are of a client/server architecture. In this case the server is a database that contains data and the client is an application that contains presentation and application logic. In the web model these functions are separated and divided among the Three-Tiers of the architectural model. On the third tier, the database level, there are no changes because the same database is used in the web application as is used by the PowerBuilder application. However in the other tiers there are changes that have a major impact on the way the application functions. In the web model the application is implemented in application objects that reside in the application server on the middle tier. The client is responsible for the presentation of the user interface. Bear in mind that the presentation is stored in the middle-tier and is executed on the client tier after it has been downloaded into the web browser. The following paragraphs deal with architectural differences on a level of application logic, internal state and graphical user interface.

4.1.1 Application Logic

The application logic defines the data processing that goes on inside the application. It deals with data entered by the user and data retrieved from the database. In the web model this processing takes place on the client-tier and on the middle-tier. We are not interested in data processing inside the database since the database is considered to be a constant factor in the transformation from PowerBuilder client/server to Java web applications. In a PowerBuilder application all the application logic is built into the client. This needs to be taken into account when transforming a PowerBuilder application into a web application.

The difficulty here lies in the distance between the GUI and the application logic in the web model. The GUI is presented in the client-tier while the application is active in the middle-tier. This may lead to an overhead in communication between the client and the middle-tier. For instance; the user may have to enter several fields in a form that triggers an event after each form field changes. These events are passed back to the middle-tier for event handling. The result would be that the entire form is submitted to the server, processed, and reloaded into the browser many times before the user has completed the form. This is not only a nuisance to the user, but also leads to an unscaleable application. A solution could be to handle events in the browser whenever possible. A scripting language for executing scripts on the client side, such as JavaScript, can be used for this purpose. The difficult task remains of recognizing what parts of the application logic should be built into the client, and what parts should be implemented on the middle-tier.

Issue 1: The application logic is split across the client-tier and middle-tier

4.1.2 Internal State

A PowerBuilder application has an internal state that controls the processing that takes place. The internal state of an application consists of all object states and holds variables such as the name of the person currently using the application, or the exchange rate of a foreign currency. The internal state of the PowerBuilder application remains intact during the entire user-session.

Server side web applications are most often stateless. In practice this means that a server side application object is executed upon a request from the client, during execution it generates HTML and then terminates. By terminating it loses its internal state. The application server usually offers a means of storing state-information that is related to a specific user session.

Issue 2: Persistently storing and restoring the application state

4.1.3 Graphical User Interface

PowerBuilder offers a variety of user interface objects that can easily be deployed in an application. In the web-application model HTML defines the user interface. Often the HTML-code is enriched with JavaScript to create a more advanced interface.

Most of the PowerBuilder user interface objects can be rendered in HTML in a straightforward fashion. For instance the equivalent of a PowerBuilder command button in HTML is the form submit button. In Section 6.2.2 we present a Java implementation of a commandbutton that produces HTML output to enable a browser to render a button.

For other objects, such as datawindows, more work needs to be done to render them in HTML. One possibility is to use Java Applets to have full control over the rendering possibilities of the object. We did not look into Applets because in the requirements (see Section 3.3) it is stated that Applets are not to be used.

Issue 3: From a PowerBuilder based GUI to a web based GUI

4.2 Language Differences

Differences between the PowerBuilder language and the Java language have to be dealt with when transforming from one to the other.

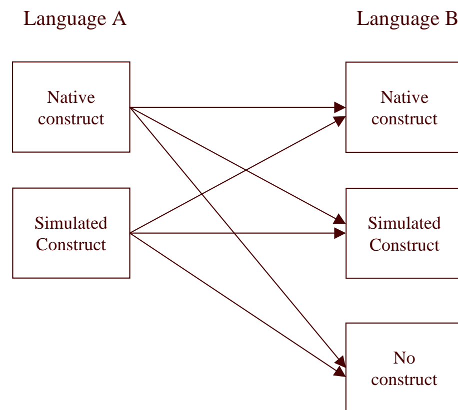

Figure 4.1: Feature mapping from one language to another

Figure 4.1, taken from [3], shows the possible mappings for features of one language to another. The figure shows the relationship between native and simulated constructs. In this context a construct is a programming language construct that has a specific function. E.g. when a language offers a function to sort lists we say that there is a native construct to sort lists in that specific language. When we translate from that language to another language that does not have a native construct to sort lists we need to simulate that functionality. Hence the term simulated construct. In the worst case we have to deal with a native or simulated construct in language `A` that has no equivalent translation in language `B`. This is expressed in the figure by the mappings into the "no construct" category.

The language differences between PowerBuilder and Java have been categorized as follows:

- Differences in Object Orientation
- Differences between 4GL and 3GL
- Differences between data types
- Differences between compiler and virtual machine

The following sections describe these differences in detail.

4.2.1 Object Orientation

Although we are dealing with a transformation from one Object Oriented language to another there still are some major differences between them that make a transformation difficult. This section lists these problems.

Inheritance

PowerBuilder, like most Object Oriented languages, offers the possibility to extend a method from a parent class. When a method is created in a sub-class, that is also defined in the parent class, the programmer has to choose between using method-extending or method-overriding. When a method is called that is the extension of a parent method the parent-class method is first executed and then the sub-class method. This is a feature that is not supported by Java.

Issue 4: Extending inheritance is not supported in Java

Explicit Calling

In PowerBuilder it is possible to explicitly call a method from an ancestor class even when it has been overridden in a sub-class. This is achieved by prefixing the method name with the name of the class it should be called from. Java does not offer this functionality.

Issue 5: Explicit calling not supported in Java

4.2.2 4GL vs. 3GL

As mentioned earlier PowerBuilder offers a broad selection of high level functionality that specializes in database access and the representation of data in the user interface. All PowerBuilder applications depend heavily on these high level functions. Java lacks built in support for almost all of these functions. This means that many of the native PowerBuilder functions have to be implemented in Java.

Issue 6: No native support for 4GL PowerBuilder functions in Java

4.2.3 Conversion of Data Types

Another language related issue is the conversion of data types. The author of [2] states that the conversion of PowerBuilder data types to Java is fairly easy. However he also states that the conversion of real and decimal values can be tricky if precision is an issue. He suggests researching and testing these data types in both languages to make sure the right result is obtained.

The authors of [3] state that even for simple datatype conversions a sophisticated data type analysis is mandatory. We conclude that the author of [2] made a good suggestion about the testing of data types in both languages.

Issue 7: Mapping problems between PowerBuilder and Java data types

4.2.4 Compiler and Virtual Machine

In PowerBuilder as well as in Java environments source code is compiled into a format that can be interpreted by a virtual machine. The PowerBuilder virtual

machine can be compiled into the application to create a standalone application. Java uses a separate virtual machine.

Lazy vs. eager evaluation

While Java uses a lazy approach for the evaluation of expressions PowerBuilder employs an eager evaluation scheme. Whether this is a difference on the level of the virtual machine or of the compiler is of no interest since a transformation is performed at the source code level.

Lazy evaluation is a means of speeding up execution time by partially evaluating an expression without violating the laws of logic. Consider the following example:

```
01  public class LazyEvaluation {
02
03    public static void main(String[] args) {
04
05      System.out.print("Evaluating: Bool_1 OR Bool_2 \n");
06      System.out.println("EvaluatesTo: " + ( Bool_1() ||  Bool_2() ) + "\n");
07
08      System.out.print("Evaluating: Bool_1 AND Bool_2 \n");
09      System.out.println("EvaluatesTo: " + ( Bool_1() &&  Bool_2() ) + "\n");
10    }
11
12    private static boolean Bool_1() {
13      System.out.println("Evaluating: Bool_1");
14      return true;
15    }
16
17    private static boolean Bool_2() {
18      System.out.println("Evaluating: Bool_2");
19      return true;
20    }
21 }
```

This piece of Java code generates the following output:

```
Evaluating: Bool_1 OR Bool_2
Evaluating: Bool_1
EvaluatesTo: true

Evaluating: Bool_1 AND Bool_2
Evaluating: Bool_1
Evaluating: Bool_2
EvaluatesTo: true
```

This sample shows that when evaluating `Bool_1 OR Bool_2` the right part of the expression is not evaluated if the left part is true. This becomes an issue when one of the operands is a function that has a side effect. In Java the function may not be called and subsequently the desired side effect may not occur.

Issue 8: Lazy vs. eager evaluation

Object Binding

When an object is referenced the virtual machine looks up that object in the repository of available objects. In PowerBuilder this is done by searching the library file path until the first occurrence of the object is found. This means that an object may exist more than once in a given set of libraries that make up an application. The ABN-AMRO has used this property of PowerBuilder for the distribution of bug fixes, this is documented in [1]. They do this by including a dummy library at the beginning of the search path. When it has been determined that one of the objects in another library contains a bug, that object is repaired and inserted into the dummy library. When the object is referenced the repaired version is found first and is used by the application.

Issue 9 Duplicate and unreached objects

Chapter 5

Software Transformations

In Chapter 4 we stated that we are dealing with both an architectural transformation and a language transformation. The goal is to make it possible to use the existing PowerBuilder application functionality in a web-application. In this chapter we suggest an approach for the transformation process.

5.1 Transformation requirements

In [3] Terekhov and Verhoef present a list of requirements that they think are necessary for realistic development of source-to-source converters. Their list contains:

- Inventory of native and simulated constructs
- Conversion strategy for each construct mapping
- Functional equivalence
- Test set migration
- Maximal automation
- Conversion Time
- Maintainability of the converted system
- Efficiency of the converted system
- Size of the converted system

In the following sections we elaborate on each of these requirements.

Inventory of native and simulated constructs

This relates to Figure 4.1 that displays the mapping possibilities of native and simulated constructs in the transformation. It is necessary to gain insight into the PowerBuilder constructs that have no native support in Java.

Constructs that can be simulated in Java have to be identified so we can implement such a simulation and a transformation to apply it.

Optionally we can also research simulated constructs in the PowerBuilder code that have native support in Java. We suggest this is optional because although applying native constructs where possible improves the readability of the source code, a transformation of one simulated construct to another simulated construct will still work.

Conversion strategy for each construct mapping

Based on the inventory of the construct mappings that are relevant in the transformation a rewriting strategy can be devised in order to make the mappings work. Mappings into the 'unsupported' category should be avoided by removing the constructs to which they apply from the PowerBuilder applications. Mappings from native to simulated should be expressed in the rewriting that is to take place on the annotated export files. The optional rewriting of simulated constructs to native constructs can take place on the resulting Java.

Functional equivalence

With functional equivalence the authors of [3] mean that the transformed system should be functionally equivalent to the original system. So a bug in the original system should lead to a bug in the transformed system. It is their experience that a customer seeking a transformation will often request that the transformation modifies the original in such a way that unsafe code and faults are removed. They suggest specifying how to deal with a 'requirements creep' in the requirements specification of the transformation.

As explained in Section 4.2.4 the PowerBuilder applications may contain duplicate objects. As a requirement of our transformation we should state that no duplicate objects may exist in the PowerBuilder applications. They should be removed before the transformation takes place. Since the duplicate objects all reside in a specific library this is not so hard to do. The benefit for the conversion process is the reduction of the amount of code that has to be transformed.

Another aspect of functional equivalence is the prevalence of application level bugs during the transformation. We suggest to take a step back when a bug is discovered and fix it in the PowerBuilder Integrated Development Environment (PB-IDE). Using the PB-IDE for this purpose is preferred because it is known how to perform maintenance and development tasks in that environment, and existing testing procedures can be used to evaluate bug-fixed-versions of the PowerBuilder applications.

The same should be done for unsafe code. For example; it may be the case that PowerBuilder has no array boundary checking. In unsafe code an off-by-one error could access an out of bounds array element. At runtime this may remain undetected. However, when transforming this unsafe code to Java without taking the different approach to boundary checking into account, side-effects could occur. The compiler may issue a warning, or the application may display unexpected behavior.

We suggest to deal with unsafe code issues in the PB-IDE. We note that code that is considered unsafe in Java, may be considered safe in PowerBuilder. This is due to differences in the implementation of the PowerBuilder runtime environment and the Java Virtual Machine. Instead of using the term unsafe code it can also be said that certain constructs map onto the 'no construct' category of Java.

Test set migration

> *"It should be stated whether the test sets belonging to the original system also have to be converted. During such automated efforts, also errors in the tests are exposed. It should be clearly stated what is the policy towards modification of the test sets."*([3] page 5.)

The test sets should show that the application is capable of performing the functions it is built for. For each change to the PowerBuilder application it should be determined whether or not the test sets have to be updated so they remain valid. Test sets should be transformed with the applications to enable testing of the transformed applications. The transformation of the test sets is outside the scope of this thesis. We leave this to future work because it is something that should be done after the transformation of the PowerBuilder applications has been achieved.

Maximal automation

Limiting the amount of human intervention that is necessary for the transformation also limits the amount of human errors that can be made. However since a lot of the 4-GL functions in PowerBuilder are not supported in Java there is a considerable amount of work to be done to re-implement these. Since we do not have access to source code of the PowerBuilder implementation of these 4-GL functions this is to be performed by hand. It only needs to be done once for the entire collection of PowerBuilder applications that are transformed.

Conversion Time

It is stated that conversion time can be an issue if the converter is needed many times instead of just once. Since we are doing a one time transformation this is not such an issue. As long as the transformation can be done in less time than a manual re-implementation we are on the safe side.

Maintainability of the converted system

The converted system should be maintainable. One way of maintaining the converted system is by applying the maintenance to the PowerBuilder application and performing a transformation after each round of maintenance. This would only work if the conversion is fully automated, and does not take too much time. It would limit the use of native Java constructs that are not available in PowerBuilder, and also not supported in the transformation. Preferably the maintenance is performed on the converted system itself especially since the ABN AMRO wants to get rid of PowerBuilder as a whole.

A common observation[3] is that when the maintenance engineers of the original system are to maintain the converted system they would benefit from similarities between the two. In our transformation this is mostly limited to the maintenance of the PowerScript, since the programmer is used to the graphical development environment, and has little need for knowledge about the exported source code. A suggestion is to implement a Java interpreter for PowerScript. On the one hand this means that there would be no need to transform the PowerScript in the PowerBuilder applications. The PowerBuilder developer is familiar with the PowerScript code and could thus benefit from this. On the other hand, for engineers with no prior experience with PowerScript this approach would not improve maintainability but would decrease it instead. Besides, interpreting PowerScript defeats the purpose of the ABN AMRO to use standard languages such as C/C++ and Java.

Efficiency of the converted system

The converted system should be acceptable in both compilation and execution time. Because in our Three-Tier model most of the processing is done in the middle-tier there is a need for a powerful server platform. Using an application server adds a single point of failure to the system, clustering the middle-tier over several servers

can reduce the risk of failure and improve performance. Most application servers provide possibilities for clustering.

Reduced performance of the transformed system should be dealt with by clustering and up scaling the application server. There should be no more degradation in efficiency than can be relieved by the application server cluster.

Size of the converted system

The converted system is most probably going to expand in size due to the differences between 4-GL languages and 3-GL languages. The extra code for the support of the 4-GL functionality will be shared among the different applications that are transformed from PowerBuilder. Another source of shared code is the Service Layer, described in Section 1.2.

The shared code should be regarded as two separate libraries, one with Service Layer functions and one with support for 4-GL functions, that are used by the transformed applications. Maintenance of the applications should also be separated from maintenance of the libraries. The extra code does not amount to the amount of code that is generated for each transformed application and does not influence the size of the converted system.

5.2 A process for Conversion

According to [3] any sensible language conversion begins with extensive restructuring of the source language programs. The authors of [3] state that because transformed source code has to be well-formed, the original source code has to be well formed as well. Figure 5.1 depicts the transformation process.

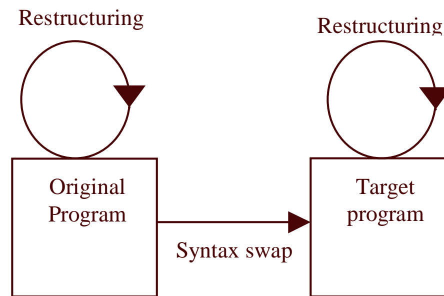

Figure 5.1: Process for language conversions

The restructuring of the original program should remove as many hard to translate language constructs as possible. Another purpose is to remove dead code from the original. Dead code is source code that is never executed but that is present in an application. In ABN-AMRO PowerBuilder applications this specifically applies to duplicate objects that are part of the bug-fixing scheme described in Section 4.2.4.

5.3 Obtaining Source Code

As explained in Section 2.1.2 the PowerBuilder Integrated Development Environment (PB-IDE) offers a graphical interface for the development of applications. Application code is stored in objects, and these objects may be exported into a text format. If these export files contain a true representation of the PowerBuilder object they originate from then we can use them to analyze the PowerBuilder objects.

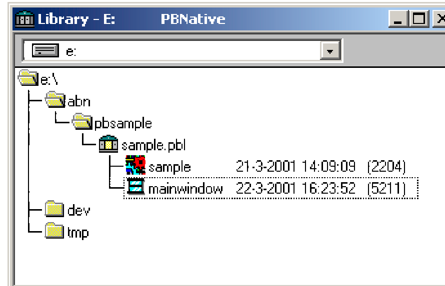

Figure 5.2: PowerBuilder Library Tool

Figure 5.2 displays a screenshot of the PB-IDE library tool. The library under scrutiny here is called `sample.pbl`. It contains an application object named `sample` and a window object called `mainwindow`. Both these objects may be exported to a text file. Appendix A shows the exported texts.

We ran a simple test to check if there is object code that is not exported when we use the built in export function. This was done by exporting the `mainwindow` object from the above sample library, then deleting it from the library file using the delete function in the library browser. After this we were unable to run the sample application. The PB-IDE generated an error message stating that the `mainwindow` object, referenced in the application object, could not be found. After importing the `mainwindow` export file back into the sample library, with the import function, this problem was solved and the application functioned like before.

From this we conclude that the export files contain a complete representation of the PowerBuilder objects inside the PB-IDE.

5.4 Transformation Strategy

In this section we describe the transformation strategy that we used for the transformation of PowerBuilder objects into Java objects. The PowerBuilder export files contain all the information that is necessary to define an application. This information can be divided into three categories:

- Comments
- Information about objects
- PowerScript

Comments contain information such as the version of PowerBuilder and the name of the object that the export file belongs to. Information about objects consists of object definitions, member variables and the relation between the objects, etc. PowerScript is mostly defined by the developer to implement event handling code, functions and procedures.

5.4.1 Annotation

When transforming to Java we want to be able to treat each category in a different way. We therefore split the information in the export files into logical blocks using tag-based markup. A logical block of export file code is defined as a series of lines that serve a common purpose such as defining an object or an event. The idea is to enforce a hierarchical structure on the export files so that we are able to select a logical subset of an export file and research the transformation of it to Java. In this way an incremental approach to the transformation can be realized.

XML-markup

The Extensible Markup Language (XML) [5] is a restricted form of SGML, the Standard Generalized Markup Language [7]. It provides a method for putting structured data in a text file. In this case we will use it to explicitly add structure to a text file. XML uses begin and end tags to create elements. These elements may contain text as well as other elements. The following sample is an XML definition of a library containing two books:

```
01 <?xml version="1.0"
02  <library type="Scientific Books">
03   <book isbn="0-201-10194-7">
04    <title>Compilers</title>
05    <subtitle>Principles, Techniques and Tools</subtitle>
06    <shelf number="1"/>
07    <author>
08     <name>Alfred V. Aho</name>
09    </author>
10    <author>
11     <name>Ravi Sethi</name>
12    </author>
13    <author>
14     <name>Jeffrey D. Ullman</name>
15    </author>
16  </book>
17  <book isbn="0-201-11954-4">
18   <title>Discrete and combinatoral mathematics</title>
19   <subtitle>An applied introduction</subtitle>
20   <shelf number="2"/>
21   <author>
22    <name>Ralph P. Grimaldi</name>
23   </author>
24  </book>
25 </library>
```

An XML document starts with the XML declaration in line 1. It should contain only one root element, in the sample this is the `library` element. All begin tags must have matching end tags except for the empty element. The empty element is a self-closing tag, an example can be found in line 6 of the above sample. The begin tag of an element can contain attribute pairs (`name=value`) as is shown in line 3 where the `book` tag contains an `isbn` attribute.

A schema for an XML-document can be defined in the form of a document type definition(DTD). The document type definition defines a grammar for a family of XML-documents. Using a DTD is optional, for many of the XML-documents that are in use there is no DTD specified. An XML processor can validate an XML-document to see if it complies with the schema if it has been specified which DTD it uses.

The reason we choose XML as the format for the annotation is the availability of the XSL language. This language is described in the next section. We will use it for the rewriting of XML-annotated export files.

5.4.2 Rewriting XML with XSL

XML is a language for structuring data. The eXtensible Stylesheet Language (XSL) [6] is a language for processing XML-documents. The World Wide Web Consortium[1] divides the language into three parts:

- XSL Transformations (XSLT): a language for transforming XML documents;
- The XML Path Language (XPath): an expression language used by XSLT to access or refer to parts of an XML document;
- An XML vocabulary for specifying formatting semantics (XSL Formatting Objects).

We have used only the XSLT part of the extensible style sheet language, the other two parts are outside the scope of this thesis.

XSLT

XSL transformations address some common needs in XML:

- Enabling display: The XSL transformation language enables display of XML by transforming XML into a language that is suitable for display, such as HTML.
- Schema translations: The transformation process is independent of any particular output grammar and can be used to transform XML data from one schema to another.

A transformation expressed in XSLT describes rules for transforming a source tree into a result tree. The transformation is achieved by associating patterns with templates. A pattern is matched against elements in the source tree. A template is instantiated to create part of the result tree. The result tree is separate from the source tree. The structure of the result tree can be completely different from the structure of the source tree. In constructing the result tree, elements from the source tree can be filtered and reordered, and arbitrary structure can be added.

XSL templates are defined using a set of XML elements. Listed below are the elements from this set that we have used:

- `xsl:value-of`
- `xsl:for-each`
- `xsl:template`
- `xsl:apply-templates`
- `xsl:copy`

xsl:value-of

The `xsl:value-of` element is used to insert the text of a node from the source XML document into the target document.

Syntax:

[1] http://www.w3c.org/

```
<xsl:value-of select="pattern">
</xsl:value-of>
```

The `select` attribute contains a string expression that is to be matched against the current context. The current context is defined as the path to the node in the source tree that is the basis for this XSL element. Appending the pattern to the current context results in the path to the node from which text is to be inserted.

xsl:for-each

The `xsl:for-each` element is used to apply a template repeatedly to a set of nodes.

Syntax:

```
<xsl:for-each
  order-by="sort-criteria-list"
  select="expression">
</xsl:for-each>
```

The required `select` attribute contains a string expression that is evaluated on the current context to determine the set of nodes to iterate over. The `order-by` attribute is optional and can be used to force iteration over the nodes in a certain order. When XSL-elements are processed that are inside an `xsl:for-each` element their context is set to the node that was matched by the `select` parameter of the `xsl:for-each` element.

xsl:template

Defines an output-template for nodes of a particular type and context.

Syntax:

```
<xsl:template
  match="pattern">
</xsl:template>
```

The `match` attribute sets the context for which the template should be executed. This attribute can be used to change the context of the source document, and provides a convenient way to navigate down into the document tree.

xsl:apply-templates

Directs the XSL processor to find the appropriate template to apply based on the type and context of each selected node.

Syntax:

```
<xsl:apply-templates
  order-by="sort-criteria-list"
  select="expression">
</xsl:apply-templates>
```

The `select` attribute defines the context for which the template should be executed. A list of sort criteria may be assigned to the `order-by` attribute.

xsl:copy

Copies the current node from the source to the output.

Syntax:

```
<xsl:copy>
</xsl:copy>
```

The `xsl:copy` element creates a node in the output with the same name, namespace, and type as the current node. Attributes and children are not copied automatically. This element makes it possible to perform identity transformations.

Using XSL two modes of rewriting are possible: template driven rewriting and data driven rewriting. The following sections give an example of these rewriting techniques.

Template driven rewriting

In the template-driven model the XSL template defines the form of the result document. Elements from the XML document that is being formatted can be included using tags that belong to the XSL namespace. A sample of a template driven XSL style sheet that rewrites the XML sample from Section 5.4.1 is given below:

```
01 <?xml version='1.0'?>
02 <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
03 <xsl:template match="/">
04
05 <html>
06  <head>
07   <title><xsl:value-of select="/library/@type"/></title>
08  </head>
09  <body>
10
11  <h1>Library of <xsl:value-of select="/library/@type"/> </h1><br/>
12  <p>
13  <table border="1">
14   <tr>
15    <td>title</td>
16    <td>author(s)</td>
17    <td>shelf #</td>
18   </tr>
19   <xsl:for-each select="/library/book">
20    <tr>
21     <td><xsl:value-of select="title"/></td>
22     <td><xsl:value-of select="author/name"/></td>
23     <td><xsl:value-of select="shelf/@number"/></td>
24    </tr>
25   </xsl:for-each>
26  </table>
27  </p>
28  </body>
29 </html>
30
31 </xsl:template>
32 </xsl:stylesheet>
```

This sample XSL style sheet rewrites the library sample XML to an HTML page that displays a table containing the title, the name of the first author and the shelf number for each `book` element. The XSL-style sheets contains two types of XSL tags: `value-of` and `for-each`. The `value-of` tag is used to select the contents of a single element of the XML document specified in the `select` attribute of the `value-of` tag. Attribute values are specified using the `@` sign. The `for-each` tag is used to iterate over each `library/book` element of the XML-document.

Figure 5.3 shows the output of the template driven XSL-style sheet when it is

applied to the XML with the Microsoft Internet Explorer. We use the Internet Explorer because it is capable of processing the XML-document according to the XSL-style sheet and displaying the result.

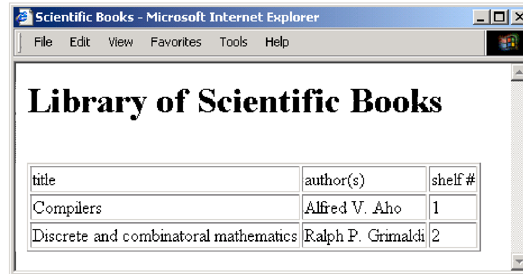

Figure 5.3: Template Driven XSL sample

Data driven rewriting

Data driven stylesheets allow an XML-tree to be traversed recursively. Consider the following example:

```
01 <?xml version='1.0'?>
02 <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
03
04 <xsl:template match="/">
05   <xsl:apply-templates/>
06 </xsl:template>
07
08 <xsl:template match="/library">
09   <html>
10    <head>
11     <title><xsl:value-of select="@type"/></title>
12    </head>
13    <body>
14    <h1>Library of <xsl:value-of select="/library/@type"/> </h1><br/>
15    <p>
16    <table border="1">
17     <tr>
18      <td>title</td>
19      <td>shelf #</td>
20      <td>author(s)</td>
21     </tr>
22      <xsl:apply-templates/>
23    </table>
24    </p>
25    </body>
26   </html>
27 </xsl:template>
28
29 <xsl:template match="book">
30   <tr>
31     <xsl:apply-templates/>
32   </tr>
33 </xsl:template>
34
35 <xsl:template match="title">
36   <td>
37     <xsl:value-of select="."/>
38   </td>
39 </xsl:template>
40
41 <xsl:template match="author[0]">
42   <td>
43     <xsl:value-of select="."/>
44   </td>
45 </xsl:template>
46
47 <xsl:template match="shelf">
48   <td>
```

```
49     <xsl:value-of select="@number"/>
50   </td>
51 </xsl:template>
52 </xsl:stylesheet>
```

As opposed to the template driven style sheet this style sheet contains multiple `template` tags. The `template` tag in line 4 matches the root of the XML document. The contents of the element is an XSL-directive that tells the XSL processor to evaluate all the `template` elements that match at this level, the root level, of the XML tree. The only element that matches at the root level is the `library` element which is matched in line 8. Instead of selecting the `title`, `author`, and `shelf/@number` we added an `apply-templates` tag. At this level of the tree the `book` elements are matched. They are processed by placing `<tr>` and `</tr>` tags around another `apply-templates` tag. Notice also that in line 41 we match `author[0]` so only the first occurrence of an `author` element is matched.

Figure 5.4 shows the output of the XML-sample using the data driven XSL style sheet.

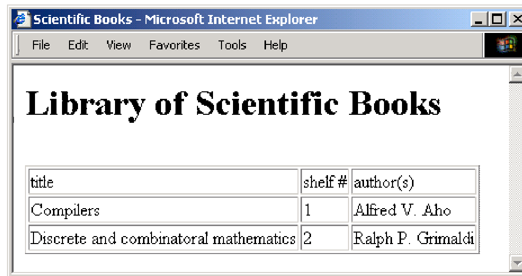

Figure 5.4: Data Driven XSL sample

5.4.3 Rewriting XML annotated PowerBuilder Export files with XSL

The transformation strategy we used while researching the transformation of PowerBuilder objects to Java involves the rewriting of XML annotated Export files using XSL. The following diagram visualizes how we are going to use this method.

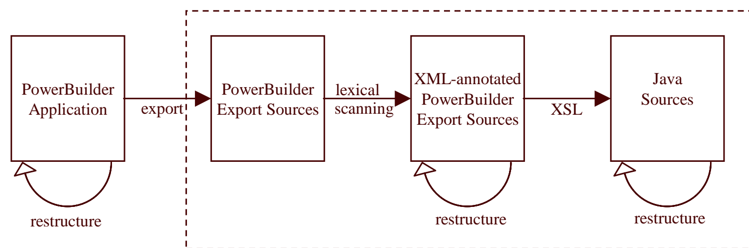

Figure 5.5: Transformation Strategy

The software that we are transforming is in one of the four states of figure 5.5:

- PowerBuilder Application
- Export Sources

- XML-annotated Export Sources
- Java Sources

We describe each of these stages:

PowerBuilder Application

These are the initial applications that are to be transformed. They are restructured using the PowerBuilder development environment. During the restructuring problematic constructs are replaced by alternative constructs that can be translated. This should be done without altering the functional aspects of the application.

Export Sources

The export sources are obtained by exporting objects from the PowerBuilder application to text files.

XML-annotated Export Sources

The Export Sources are annotated using XML-tags. At first we did this by hand. Inserting XML tags is a boring and error prone job that can be performed by a lexical scanning tool. We have implemented such a tool and have described it in Appendix B.

Rewriting is performed on the XML-annotated export sources using XSL. The rewriting process takes an XML document and rewrites it to another XML document. Section 6.2.3 contains a sample of rewriting at this level that deals with part of issue 7, the mapping problems between PowerBuilder and Java data types that was discussed in 4.

Java Sources

XSL-stylesheets are used to transform the XML-annotated export files to Java. This can be done in an incremental fashion. For instance, in an early transformation one may want to transform only the object structure of the PowerBuilder application. Using the stylesheets this is fairly easy as is shown in Section 6.2.3.

On the resulting Java sources rewriting may applied as well. We suggest using existing code beautifiers for Java, if available, for this purpose. This is outside the scope of this thesis.

Chapter 6

Sample Transformation

In this chapter we present the transformation of a sample PowerBuilder application to Java using the method described in the previous chapter. The application contains some very basic PowerBuilder features and serves to provide a sample Java implementation and a sample of the transformation process.

6.1 PowerBuilder to Java

Our approach is to examine the format of the PowerBuilder export files. We use the Extensible Mark-up Language [5] to mark parts of the code we can identify as a block having certain functionality. We then manually implement a Java application that has the same object structure as the PowerBuilder application. Finally we present an XSL-style sheet that automates the rewriting of the annotated export file to Java.

6.1.1 Sample Application

The sample application consists of a window object named `mainwindow` that has two member objects: a commandbutton `cb_1` and a single line edit `sle_1`.

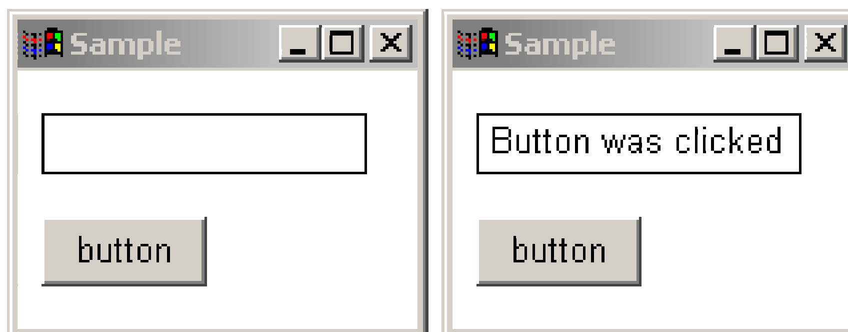

Figure 6.1: Two states of the Sample application

Figure 6.1 displays screen shots of two states that the application can be in. On the left the initial application state is shown, on the right its state after the button has been clicked.

In the PowerBuilder development environment this application consists of two objects: an application object and a window object. The approach is to export the objects and to use the export files for further analyses. Appendices A.1 and A.2 show the export code of the two objects.

6.1.2 Export Files

Since there is no specification of the PowerBuilder export file format we basically do not know what to expect inside the export files. The export files are generated by PowerBuilder, so we expect them to be of some structure that is constant for all export files. The export files however do contain source code that is not machine generated. Event handling code for instance. This code is written in PowerScript by the developer and can be found in the export files verbatim. This is also the case for any other blocks of PowerScript that have been hand written by the developer. The idea is to research the structure of the export files and the structure of the object definitions and then try to translate this to an equal and valid object structure in Java. The translation of the pieces of PowerScript are left out of the transformation at this time.

What follows is the step by step processing of the export file to add the XML markup. We begin with the first line of the export file:

```
01 $PBExportHeader$mainwindow.srw
```

This appears to be a header that PowerBuilder uses to keep track of the contents of export files. The header contains the name of the object and its type. The name of the object defined in this export file is `mainwindow`, its type is `srw`, this probably means that the file contains a window definition. Because we assume that the first line is a header we place it into a container named `PBExportHeader`. This results in:

```
01 <?xml version="1.0"?>
02
03 <PBExportFile>
04   <PBExportFileHeader>$PBExportHeader$mainwindow.srw</PBExportFileHeader>
```

The first line of our XML-document contains a tag from the XML-namespace denoting the version of this XML-document. We create a root element that is called `PBExportFile`.

The header is followed by a `forward` block that lists the types that are defined in this file:

```
02 forward
03 global type mainwindow from Window
04 end type
05 type cb_1 from commandbutton within mainwindow
06 end type
07 type sle_1 from singlelineedit within mainwindow
08 end type
09 end forward
```

For each type it is stated what its parent type is. For example; `mainwindow` inherits from `Window` and `cb_1` inherits from `commandbutton`. The type `cb_1` is a member type of the `mainwindow` which is expressed using the `within` keyword. We capture all this information by marking up these lines as follows:

```
05   <forwardDeclarations>
06     <typeDeclaration access="global" identifier="mainwindow" from="Window" within=""/>
07     <typeDeclaration access="" identifier="cb_1" from="commandbutton" within="mainwindow"/>
08     <typeDeclaration access="" identifier="sle_1" from="singlelineedit" within="mainwindow"/>
09   </forwardDeclarations>
```

Note the fact that we completely replaced the `forward` keyword with the XML-tag `<forwardDeclaration>`. We continue with the next part of the export file:

```
11 global type mainwindow from Window
12 int X=1075
13 int Y=485
14 int Width=723
15 int Height=497
16 boolean TitleBar=true
17 string Title="Sample"
18 boolean ControlMenu=true
19 boolean MinBox=true
20 boolean MaxBox=true
21 boolean Resizable=true
22 cb_1 cb_1
23 sle_1 sle_1
24 end type
```

In this part of the export file we find the declaration of the `mainwindow` type that inherits from `Window`. This part of the declaration contains the initialization of member variables and member objects. On line 17 we find the title that we gave the window when we created it in PowerBuilder. On lines 22 and 23 we find the objects that are members of this window. We apply XML-markup and arrive at the following:

```
10   <typeDefinition access="global" identifier="mainwindow" from="Window" within="">
11     <memberDeclarations>
12       <variable type="int" identifier="X">1075</variable>
13       <variable type="int" identifier="Y">485</variable>
14       <variable type="int" identifier="Width">723</variable>
15       <variable type="int" identifier="Height">497</variable>
16       <variable type="boolean" identifier="TitleBar">true</variable>
17       <variable type="string" identifier="Title">"Sample"</variable>
18       <variable type="boolean" identifier="ControlMenu">true</variable>
19       <variable type="boolean" identifier="MinBox">true</variable>
20       <variable type="boolean" identifier="MaxBox">true</variable>
21       <variable type="boolean" identifier="Resizable">true</variable>
22       <object type="cb_1" identifier="cb_1"/>
23       <object type="sle_1" identifier="sle_1"/>
24     </memberDeclarations>
25   </typeDefinition>
```

The member variables that are found inside this type are all defined in the parent type `Windows`. We know this from the PowerBuilder help files which contain a section that lists all methods, properties and functions for all PowerBuilder system classes.

The next part of the source code is a line that contains the instantiation of a global object named `mainwindow` that is of type `mainwindow`.

```
25 global mainwindow mainwindow
```

We add XML-markup and arrive at the following:

```
26   <globalObjectInstantiation type="mainwindow" identifier="mainwindow"/>
```

The next piece of the export file makes things a little more interesting:

```
27 on mainwindow.create
28 this.cb_1=create cb_1
29 this.sle_1=create sle_1
30 this.Control[]={ this.cb_1, this.sle_1}
31 end on
```

Inside the `on` block there are some PowerScript statements that are to be executed when the mainwindow is created. Line 28 instantiates the `cb_1` object which is referenced using `this.cb_1`. That makes us believe that we are dealing with the constructor of the mainwindow object. Line 30 adds a control array that contains references to the member objects `cb_1` and `sle_1`. We add XML-markup:

```
27   <onDeclaration typeIdentifier="mainwindow" event="create">
28     this.cb_1=create cb_1
29     this.sle_1=create sle_1
30     this.Control[]={ this.cb_1,this.sle_1}
31   </onDeclaration>
```

We have left the handling code for this on-event unchanged and stored it in an `onDeclaration` element. In this case the event handling code is generated PowerScript. The export file continues:

```
33 on mainwindow.destroy
34 destroy(this.cb_1)
35 destroy(this.sle_1)
36 end on
37
38 type cb_1 from commandbutton within mainwindow
39 int X=42
40 int Y=221
41 int Width=284
42 int Height=105
43 int TabOrder=20
44 string Text="button"
45 int TextSize=-10
46 int Weight=400
47 string FaceName="Arial"
48 FontFamily FontFamily=Swiss!
49 FontPitch FontPitch=Variable!
50 end type
```

What we see in the above code fragment is the definition of another `on` event, namely the destructor of the mainwindow object. Lines 38 through 50 contain the declaration of the member object `cb_1`. We added XML-markup to this fragment and arrive at the following:

```
32   <onDeclaration typeIdentifier="mainwindow" event="destroy">
33     destroy(this.cb_1)
34     destroy(this.sle_1)
35   </onDeclaration>
36   <typeDefinition access="" identifier="cb_1" from="commandbutton" within="mainwindow">
37     <memberDeclarations>
38       <variable type="int" identifier="X">42</variable>
39       <variable type="int" identifier="Y">221</variable>
40       <variable type="int" identifier="Width">284</variable>
41       <variable type="int" identifier="Height">105</variable>
42       <variable type="int" identifier="TabOrder">20</variable>
43       <variable type="string" identifier="Text">"button"</variable>
44       <variable type="int" identifier="TextSize">-10</variable>
45       <variable type="int" identifier="Weight">400</variable>
46       <variable type="string" identifier="FaceName">"Arial"</variable>
47       <variable type="FontFamily" identifier="FontFamily">Swiss!</variable>
48       <variable type="FontPitch" identifier="FontPitch">Variable!</variable>
49     </memberDeclarations>
50   </typeDefinition>
```

In the next piece of the export file we find something new:

```
52 event clicked;sle_1.text="Button was clicked"
53 end event
```

These lines contain the definition of an event. Since we defined this event in PowerBuilder ourselves we know it belongs to the previously defined commandbutton `cb_1`. So the position of the event definition in the export file tells us to which object it belongs. We add this knowledge to the XML-markup as follows:

```
51   <event typeIdentifier="cb_1" action="clicked">
52     sle_1.text="Button was clicked"
53   </event>
```

The type of the object the event belongs to is stored in the parameter `typeIdentifier`. The actual event-handling code is the content of the `event` element. The rest of the export file contains the definition of the `singlelineedit` object. Appendix A.3 shows the full text of the annotated export file.

6.2 A Java implementation

The next thing we are going to do is try to implement the mainwindow object in Java. In this process we use the XML-tags as a guideline. The tags give us information about the export code they enclose.

We start at the top of the annotated file:

```
01 <?xml version="1.0"?>
02
03 <PBExportFile>
04   <PBExportFileHeader>$PBExportHeader$mainwindow.srw</PBExportFileHeader>
```

We add the `PBExportFileHeader` as a comment to the Java source so we can always recognize it as the Java implementation of the `mainwindow` object. So the first lines of our Java Source looks like this:

```
01 /*
02    This is the java implementation of
03    the PowerBuilder window object:
04
05    $PBExportHeader$mainwindow.srw
06
07 */
08
```

In Java there is no need to include a forward block with object declarations, so we skip that part of the annotated file and continue with the `globalTypeDefinition`:

```
10   <typeDefinition access="global" identifier="mainwindow" from="Window" within="">
11     <memberDeclarations>
12       <variable type="int" identifier="X">1075</variable>
13       <variable type="int" identifier="Y">485</variable>
14       <variable type="int" identifier="Width">723</variable>
15       <variable type="int" identifier="Height">497</variable>
16       <variable type="boolean" identifier="TitleBar">true</variable>
17       <variable type="string" identifier="Title">"Sample"</variable>
18       <variable type="boolean" identifier="ControlMenu">true</variable>
19       <variable type="boolean" identifier="MinBox">true</variable>
20       <variable type="boolean" identifier="MaxBox">true</variable>
21       <variable type="boolean" identifier="Resizable">true</variable>
22       <object type="cb_1" identifier="cb_1"/>
23       <object type="sle_1" identifier="sle_1"/>
24     </memberDeclarations>
25   </typeDefinition>
```

We are now going to implement this `typeDefinition` as a `public class` definition in Java:

```
09 public class Mainwindow extends Window {
10
11   cb_1 cb_1 = new cb_1();
12   sle_1 sle_1= new sle_1();
13
14   Mainwindow() {
15     X = 1075;
16     Y = 485;
17     Width = 723;
18     Height = 497;
19     TitleBar = true;
20     Title = "Sample";
21     ControlMenu = true;
22     MinBox =  true;
23     MaxBox  = true;
24     Resizable = true;
```

We know from the annotated source that the `Mainwindow` object is a child of the `Window` type. We express this in Java using the `extends` keyword. Notice also that the member variables are initialized inside the constructor function of the `Mainwindow` class while the member objects are instantiated inside the class definition. We continue with the next part of the annotated export file:

```
26   <globalObjectInstantiation type="mainwindow" identifier="mainwindow"/>
```

This line contains the instantiation of a global object `mainwindow`. For now we skip it and continue with the rest of the annotated export file.

```
27   <on typeIdentifier="mainwindow" event="create">
28     this.cb_1=create cb_1
29     this.sle_1=create sle_1
30     this.Control[]={ this.cb_1,this.sle_1}
31   </on>
```

The `on` block belongs to the `mainwindow` type and contains code that is to be executed when the `mainwindow` object is created. It contains member object instantiations for all the member objects of the `mainwindow` type and assigns a control array that contains a reference to each of this types member objects. There is no need to implement the member object instantiations in the Java implementation because we already instantiated the member objects in lines 11 and 12.

We do implement the control array in Java. At this time we do not know its purpose, especially since we don not recognize it from the PowerBuilder development environment. Our java implementation of it looks like this:

```
025     Control = new GuiObject[] {this.cb_1 , this.sle_1};
```

The next part of the annotated export file contains another `on` block. This one belongs to the `destroy` event of the `mainwindow` object. We assume PowerBuilder uses it to destroy the member objects `cb_1` and `sle_1`. There is no need to transform these destructor functions to Java because the Java virtual machine will handle that for us.

The next element in the annotated export file is a `typeDeclaration`. This is translated to Java in the same way the `globalTypeDefinition` was translated and results in the following Java code:

```
38 class cb_1 extends commandbutton{
39   public cb_1() {
40    X=42;
41    Y=221;
42    Width=284;
43    Height=105;
44    TabOrder=20;
45    Text="Button";
46    TextSize=-10;
47    Weight=400;
48    FaceName="Arial";
49    //FontFamily=Swiss!;
50    //FontPitch=Variable!;
51   }
52 }
```

We have commented out the code in lines 49 and 50. These contain enumerated PowerBuilder types `fontFamily` and `FaceName` that are not native to Java. Since they only deal with layout and are not vital for the application we ignore them for now. The next element we have to transform is an `event`:

```
51   <event type="cb_1" action="clicked">
52     sle_1.text="Button was clicked"
53   </event>
```

For the time being we implement an `eventHandler` method inside the mainwindow class to deal with events. Our event handler takes a string argument `event` that is used to pass events to the event handler. Since events are bound to objects we use dot notation to express the relation between objects and events. In our notation the event in the annotated source code above would be `cb_1.clicked`. This is the Java code for the event handler:

```
28   public void eventHandler(String event){
29
30     System.out.println("eventHandler("+event+")");
31
32     if (event.compareTo("cb_1.clicked")==0) {
33       sle_1.text="The 'go' button was clicked";
34     }
35   }
36 }
```

Line 33 contains the event handle code for the event `cb_1.clicked`. In this specific case the line of PowerScript happens to be legal Java code that performs the same function. So in this case we don't have to convert the PowerScript to Java but simply copy it verbatim.

The rest of the annotated file contains the declaration of the single line edit. This is transformed to Java in a similar manner as the commandbutton.

6.2.1 PowerBuilder System Classes

The `mainwindow`, `cb_1` and `sle_1` objects inherit from `Window`, `commandbutton` and `singlelineedit` respectively. However, these parent objects are not native to Java and should be re-implemented in order to enable the application objects to inherit their properties and methods. We implement only those features that are used in the sample application.

An example of a basic implementation of the `Window` class:

```
01 public class Window {
02   int       X,Y,Width,Height;
03   boolean   TitleBar,ControlMenu,MinBox,MaxBox,Resizable;
04   String    Title ="";
05 }
```

6.2.2 Adding GUI functionality

If we want to render the user interface in a web browser, then we need to generate HTML at some point. We are going to add functionality to the Java implementation of the PowerBuilder system classes for this purpose. The following shows the Java implementation of the `commandbutton` class:

```
01 public class commandbutton extends GuiObject{
02   int X;
03   int Y;
04   int Width;
05   int Height;
06   int TabOrder;
07   int TextSize;
08   int Weight;
09   String Text;
10   String FaceName;
11   //FontFamily FontFamily=Swiss!
12   //FontPitch FontPitch=Variable!
13
14   public String toHTML() {
15     return("<div style=\"position:absolute;" +
16            "left:" + X +
17            ";top:" + Y +
18            ";height:" + Height +
19            ";width:" + Width +
20            "\">\n" + "<input type=\"submit\" name=\"" +
21            getClass().getName()+"\" value=\"" + Text + "\">" +
22            "</div>\n")
23   }
24 }
```

We added a method `toHTML()` that returns a string containing an HTML representation of the `commandbutton`.

We also added the `toHTML()` method to the `Window` class, the Java implementation of the PowerBuilder system type `Window`. When calling the `toHTML()` method for a `Window` object one would want to get the HTML for the entire window, including its member objects, in return. This is where we can use the `Control` array that we found in the PowerBuilder export code we dealt with in Section 6.1.2. The `toHTML()` method of the `Window` object could make a call to the `toHTML()` method of each of the members of the `control` array. In order for this to work the `control` array must hold objects of a class that support the `toHTML()` method. We therefore created a parent class `GuiObject` that implements this method:

```
01 public class GuiObject {
02   public String toHTML() {
03     return("");
04   }
05 }
```

All classes that actually implement a `toHTML()` method should now extend the `GuiObject` class. This allows us to call this method for all the objects in an array of `Guiobject` objects very easily, as can be seen in line 17 of the `toHTML()` method of the `window` class:

```
01 public class Window {
02
03   int      X,Y,Width,Height;
04   boolean  TitleBar,ControlMenu,MinBox,MaxBox,Resizable;
05   String   Title;
06   GuiObject[] Control;
07
08   public String toHTML() {
09     String HTMLheader = new String();
10     String HTMLfooter = new String();
11     String memberObjectHTML = new String();
12
13     HTMLheader = "<html>\n<head>\n<title>"+Title+"</title>\n</head><body>\n<form>\n";
14     HTMLfooter = "</body>/n</form>\n</html>";
15
16     for (int index=0 ; index < Control.length; index++)
17        memberObjectHTML += Control[index].toHTML();
18
19     return( HTMLheader + memberObjectHTML + HTMLfooter );
20   }
21 }
```

We have added an array of `GuiObjects` to the window class. This array is instantiated with two objects of the type `GuiObject`. The constructor of the `mainwindow` class should assign the member objects `cb_1` and `sle_1` to the array. The above code sample shows how the control array is used to include HTML representations of all the member objects in the HTML representation of the window.

Figure 6.2 shows the output of the Java implementation of the sample application. The Following Java program was used to retrieve the HTML from the Java implementation of the window:

```
01 public class Run {
02  Public static void main(String[] args){
03
04  Mainwindow window = new Mainwindow();
05  window.eventHandler("cb_1.clicked");
06  System.out.println(window.toHTML());
07  }
08 }
```

In line 4 we instantiate an object identified by `window` of the class `Mainwindow`. This is the instantiation that we skipped in Section 6.2.

Figure 6.2: Browser with generated HTML from Java implementation

6.2.3 Generating code with Stylesheets

In Section 5.4.2 it was shown how XSL can be used to rewrite an XML document. What we will do is create an XSL-stylesheet that generates the Java implementation of Section 6.2.

XML to Java

```
01 <?xml version='1.0'?>
02 <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" language="VBScript">
03 <xsl:template match="/">
04 <html><pre>
05
06 /*
07    This is the java implementation of
08    the PowerBuilder window object:
09
10    <xsl:value-of select="/PBExportFile/PBExportFileHeader"/>
11
12 */
13
14 public class <xsl:value-of select="/PBExportFile/typeDefinition/@identifier"/>
  extends <xsl:value-of select="/PBExportFile/typeDefinition/@from"/> {
15
16   <xsl:for-each select="/PBExportFile/typeDefinition[@access='global']/
  memberDeclarations/object">
17   <xsl:value-of select="@type"/> <xsl:value-of select="@identifier"/>  = new
     <xsl:value-of select="@type"/>();
18   </xsl:for-each>
19
20   <xsl:value-of select="/PBExportFile/typeDefinition/@identifier"/>(){
21     <xsl:for-each select="/PBExportFile/typeDefinition[@access='global']/
  memberDeclarations/variable">
22     <xsl:value-of select="@type"/> <xsl:value-of select="@identifier"/>  =
  <xsl:value-of select="."/>;
23     </xsl:for-each>
24
25     Control = new GuiObject[] {<xsl:for-each select="/PBExportFile/
  typeDefinition[@access='global']/memberDeclarations/object"> <xsl:value-of
  select="@identifier"/><xsl:if test="context()[not(end())]">, </xsl:if>
  </xsl:for-each>};
26
27   }
28 }
29
30
31
32 <xsl:for-each select="/PBExportFile/typeDefinition[@access='']">
33 class <xsl:value-of select="@identifier"/> extends <xsl:value-of
  select="@from"/>{
34
35   <xsl:value-of select="@identifier"/>() {
36     <xsl:for-each select="memberDeclarations/variable">
37     <xsl:value-of select="@type"/> <xsl:value-of select="@identifier"/> =
  <xsl:value-of select="."/>;
```

```
38     </xsl:for-each>
39    }
40 }
41 </xsl:for-each>
42 </pre></html>
43 </xsl:template>
44 </xsl:stylesheet>
```

This template-driven XSL-stylesheet rewrites the XML-annotated PowerBuilder export files to Java. We enclose the generated Java in HTML-tags as can be seen in lines 4 and 42. We used the Microsoft[1] Internet Explorer version 5 to apply the stylesheet to the XML document. The `<pre>` tags instruct the browser to render the content as text with no markup. Due to some limitations of the implementation of XSL in Internet Explorer we had to slightly tweak the generated Java by removing some linefeeds. Appendix A.5.1 shows the generated Java with the extra linefeeds The version with the linefeeds removed can be found in Appendix A.5.2. The interested reader is invited to compare that with the manual Java implementation in Appendix A.4. In future releases of the XSL processor that is used in Microsoft products like the Internet Explorer there will be more support for the control of white space.

XML to XML

What we have shown here is how to rewrite the annotated export files to Java. The following sample employs XSL rewriting that generated XML from XML. With this sample we solve a small problem in the generated Java of the previous sample. In the PowerBuilder export files the string data type is denoted as `string`. In Java this is equivalent to the `String` object. We will use an XSL-translation to capitalize the `string` keyword in the annotated export files. We consider this difference in case to be part of issue 7, mapping problems between PowerBuilder and Java data types.

We will be using the XSL identity transformation to perform a transformation that leaves the XML-document intact. This transformation effectively creates a copy of the input XML-document without altering it. We will add a matching rule that matches the `string` keyword in variable declarations and definitions and replaces it with `String`.

```
01 <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
02  <xsl:template match="/">
03   <xsl:apply-templates select="*"/>
04  </xsl:template>
05
06  <xsl:template match="*|@*|text()|cdata()|comment()|pi()">
07   <xsl:copy>
08    <xsl:apply-templates select="*|@*|text()|cdata()|comment()|pi()"/></xsl:copy>
09  </xsl:template>
10
11   <xsl:template match="variable/@type[.='string']">
12   <xsl:copy>String</xsl:copy>
13  </xsl:template>
14 </xsl:stylesheet>
```

In line 3 this data driven template applies the template matches it contains on all the elements of the XML document. The template match in line 6 matches any XML element and uses `xsl:copy` to reproduce it. Any nested templates are copied using the `apply-templates` statement in line 8. Line 11 matches `variable` elements that have a `string` parameter. The keyword `String` is copied into its place.

[1]Tools from other vendors are available, see http://www.w3.org/Style/XSL

Pre-Processing

XML-namespace elements as well as XML-tags start with a "<" symbol and end with a ">" symbol. In order to generate well-formed XML these symbols have to be escaped before the XML-tags are added to the export files. Another preprocessing measure that we took is to remove the PowerBuilder line-continuation character "&". When a single line of PowerScript is to be split up in multiple lines they should be separated by the line-continuation character. We removed these characters along with the following new-line character. This simplifies the lexical scanning tool that is described in Appendix B.

6.3 Summary

What we have shown is how to transform the object structure of a PowerBuilder application to Java. With this approach we can take a set of PowerBuilder objects and transform them to Java. The resulting Java objects have the same inheritance structure as the PowerBuilder objects.

We transformed objects and member objects without the member functions and procedures. We also have not transformed the PowerScript that holds most, if not all, of the application logic.

In [12] an approach is described for system renovation that is related to our work. The authors of [12] introduce an *Annotated Term Format (ATF)*. They show how to use the ATF to represent parse trees. [12] Also states minimal properties for a query algebra to inspect and modify the ATF parse trees. Instead of an ATF we have used XML. XSL is what we have used to query the XML-annotated export files. [12] states:

> *"Often combinations of extracted information yield important new information, therefore it is natural to have the possibility of combining queries. This implies the existence of basic queries as well as operations to combine them, in other words an algebra of queries."*

XSL offers no proper support for combining queries other than nesting two or more `xsl:for-each` statements. Another drawback to using XML and XSL for transforming PowerScript to Java is the irregular nature of the PowerScript in the PowerBuilder applications. Due to this we need something more advanced than lexical scanning to annotate, or parse, the PowerScript.

We suggest that for the transformation of PowerScript to Java a different rewriting technique should be used. A possible candidate for this is the ASF-SDF Meta Environment[2] [13][14] which features a powerful parser generator and extensive rewriting possibilities.

[2] Available at http://www.cwi.nl/projects/MetaEnv/

Chapter 7

Conclusions

In this thesis we have described a strategy for the reuse of PowerBuilder applications. We presented a transformation strategy that transforms PowerBuilder objects to Java objects in three steps as is shown in Figure 7.1.

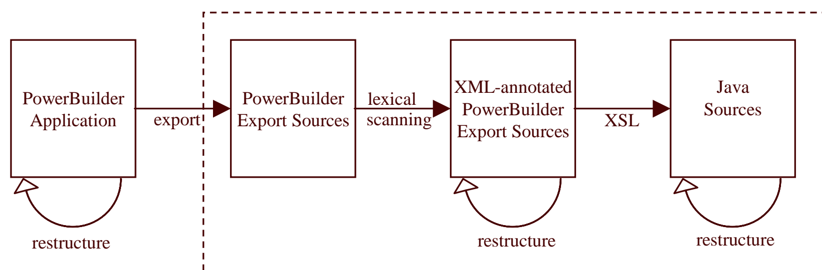

Figure 7.1: Transformation Strategy

What we have contributed to answer the question how PowerBuilder applications can be reused as Java web applications is a transformation strategy that solves some of the issues related to the problem. Some of the issues that are related to the transformation can be dealt with at some stage of the transformation. Any construct that is too difficult to transform automatically has to be dealt with before the transformation takes place.

In this chapter we state the results of our research, what future work there is left and our conclusions.

7.1 Results

In this section we state the results of our research. We do this in two parts. The first part deals with the issues that are described in Chapter 4. The second part states the achievements of our transformation strategy.

7.1.1 Issues

In Chapter 4 we listed 9 issues that are related to the transformation from PowerBuilder client/server applications to Java web applications. We list these issues here and discuss how our transformation strategy can help to solve them.

- Issue 1, The application logic is split across the client-tier and middle-tier

- Issue 2, Persistently storing and restoring the application state
- Issue 3, From a PowerBuilder based GUI to a web based GUI
- Issue 4, Extending inheritance not supported in Java
- Issue 5, Explicit calling not supported in Java
- Issue 6, No native support for 4GL PowerBuilder functions in Java
- Issue 7, Mapping problems between PowerBuilder and Java data types
- Issue 8, Lazy vs. eager evaluation
- Issue 9, Duplicate and unreached objects

Issue 1, The application logic is split across the client-tier and middle-tier

This is one of the differences between PowerBuilder applications and Java web applications that is caused by the difference in architecture. We have described this in Section 4.1.1. In Section 6.2.2 we described how to implement the HTML GUI in the Java web application objects. The application logic for the client-tier should be added to the HTML that is sent to the client. For each GUI object that needs client-tier application logic JavaScript can be added to the `toHTML()` method.

Our transformation strategy does not specify how to decide what processing can take place on the client and what processing should take place in the middle-tier. Theoretically all the processing may take place on the middle tier. As explained in section 4.1.1 performance will improve if some of the processing is moved to the clients tier.

Issue 2, Persistently storing and restoring the application state

Application servers usually offer a mechanism for storing and restoring application state information that is related to a specific user session. In practice this means that there is a set of variables for each client whose values are restored when the client requests a service.

The suggested transformation strategy does not deal with this issue. How this should be implemented depends on the type of application server that is to be used. A possible solution is to leave the Java web application running throughout the user session, and use the application server as an interface between the browser and the Java web application.

Issue 3, From a PowerBuilder based GUI to a web based GUI

In Section 6.2.2 we described how to implement the HTML GUI in the Java Web application objects. We suggest implementing a `toHTML()` method for each visible object that is capable of generating HTML to display the object. The drawback to this method is that the layout is hardcode into the application.

Another approach would be to replace the `toHTML()` function with a `toXML()` function that generates an XML definition of the object. A separate XSL-style sheet can then be used to generate the HTML that is now generated by the `toHTML()` method. The benefit of this is that the layout is separated from the application logic. This makes the layout easier to maintain because a requirement change in the layout can be handled by adapting the XSL-style sheet. In the current solution the `toHTML()`

would have to be changed which will have a bigger impact on the application than a change in the XSL-style sheet. A change to the application code would require extensive testing of the application. Obviously separation of application code and layout is preferred here.

Issue 4, Extending inheritance not supported in Java

Extending inheritance is a PowerBuilder feature that can be turned on and off with a checkbox in the function/procedure design interface of the PB-IDE. When this feature is used PowerBuilder simply inserts a statement into the PowerScript that calls the function or procedure in the parent object. This can be done in Java in the same way that is done by PowerBuilder using a call to the method in the parent object using `super().MethodName()`. This issue will automatically be resolved when a transformation for the PowerScript is available.

Issue 5, Explicit calling not supported in Java

Besides calling methods in the parent object PowerBuilder allows calling a method in e.g. a parent of a parent. In Java this is possible using a construction with a call to the parent of the parent using `super().super().Method`. The difference in PowerBuilder is that the class name of the ancestor type may be used to reference its methods.

This can dealt with at two places in the transformation process. One possibility is to replace explicit method calling while restructuring the original PowerBuilder applications. The calls can be replaced by calls to `super().super()`. The other possibility is to replace the explicit method calls where they occur in the PowerScript. Translating the PowerScript to Java is left outside of the scope of this thesis.

Issue 6, No native support for 4GL PowerBuilder functions in Java

A lot of the typical 4-GL functions that are used in PowerBuilder applications have to be implemented in Java. This can only be done manually because the source code of their PowerBuilder implementations is not available. This is a general drawback when transforming to a less rich language.

Issue 7, Mapping problems between PowerBuilder and Java data types

Not all of the PowerBuilder data types can be mapped onto Java data types in a straightforward manner. Some PowerBuilder data types may have to be implemented in Java just like the 4-GL functions in issue 6.

Issue 8, Lazy vs. eager evaluation

One way of dealing with this issue is to make sure that the expressions in the PowerBuilder application are safe for lazy as well as for eager evaluation. This can be done manually during the restructuring of the PowerBuilder applications. Another way of dealing with it is during the transformation of the PowerScript to Java. We left the transformation of the PowerScript code to Java outside the scope of this thesis.

Issue 9, Duplicate and unreached objects

In our transformation strategy duplicate and/or unreached objects are removed from the application using the PowerBuilder Integrated Development Environment.

The benefit of removing these objects in the first stage of the transformation is that existing testing procedures can be used. This way we can be sure that we are transforming an application that implements the functionality it is meant to implement, provided that this can be ascertained using the existing testing procedures.

7.1.2 Achievements

Using a relatively simple lexical scanner proved to be a quick and effective way to add markup to the export files. An other approach is to specify a grammar for the export files and use a grammar based tool to generate Java code. Due to the regular nature of the generated PowerBuilder export files a lexical scanner was sufficient.

In chapter 6 we demonstrated that by following the strategy that is described in chapter 5 a simple PowerBuilder application can be transformed to a Java application. The limitation of what we have achieved is that we do not transform the PowerScript code that contains most of the business logic of the PowerBuilder applications. However, we were able to transform the object structure of a PowerBuilder application to a valid object structure in Java.

Our solution for implementing a user interface in the Java objects using a `toHTML()` of a `toXML()` method is solution to issue 3.

7.2 Future Work

The solution to issues 5, 7 and 8 is uncertain because they rely on a transformation of PowerScript to Java. Research into that transformation is necessary to decide on its feasibility. There are about 90.000 lines of PowerScript in the ABN-AMRO applications, transforming that to Java manually is probably more work than a manual re-implementation of the applications.

The manual restructuring of the PowerBuilder applications involves a lot of overhead caused by necessary testing. Future research could be conducted to determine if this restructuring can be done automatically, and how much restructuring there is to be done.

7.3 Conclusions

We conclude that the transformation method using a lexical scanner, XML and XSL provides a fast way of transforming the object structure from PowerBuilder to Java. However, without the business logic that is captured in the PowerScript these object structures are useless for the ABN-AMRO. Automated transformation of the PowerScript is a requirement for the success of our transformation strategy. Besides that, the amount of manual restructuring of the PowerBuilder applications is yet unknown. We expect that these two aspects will require to much manual work to be able to effectively use our strategy.

Bibliography

[1] *Gebruikershandleiding ProDeca* ABN AMRO, September 1997

[2] Robert Breidecker *Converting PowerBuilder to Java.* PowerBuilder Developers Journal, November 2000, http://www.PowerBuilderJournal.com

[3] A.A. Terekhov and C. Verhoef *The Realities of Language Conversions* IEEE Software November/December 2000

[4] Bill Hatfield *Developing PowerBuilder 5 Applications, Fourth Edition.* Macmillan Computer Publishing, 1996

[5] *Extensible Markup Language (XML) 1.0 (Second Edition)* http://www.w3.org/TR/2000/REC-xml-20001006 Copyright 06-10-2000, World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/"W3C Recommendation, 6 October 2000

[6] *XSL Transformations (XSLT) Version 1.0* http://www.w3.org/TR/1999/REC-xslt-19991116 Copyright 16-11-1999, World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/" W3C Recommendation, 16 November 1999

[7] *Standard Generalized Markup Language (SGML)* ISO (International Organization for Standardization) 8879:1986(E). Information processing - Text and Office Systems - Standard Generalized Markup Language (SGML). First edition - 1986-10-15. [Geneva]: International Organization for Standardization, 1986

[8] *RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1.* R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. June 1999. Status: DRAFT STANDARD

[9] *Client-Side JavaScript Guide, version 1.3* Netscape Communications Corporation, 501 East Middlefield Road, Mountain View, CA 94043

[10] *Programming Perl 3rd edition* Larry Wall, Tom Christiansen & Jon Orwant, July 2000, O'Reilly & Associates, Inc.

[11] *Levaraging XML in Automatic Conversion of Your Client/Server Applications to Internet* Victor Rasputnis and Anatole Tartakovsky, XML-Journal Volume 2 Issue 3, April 2001 page 30, http://www.sys-con.com/xml/

[12] *Core Technologies for System Renovation* M. van den brand, P. Klint, C. Verhoef, Proceedings of the XXIII-rd Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM 1996

[13] *A meta-environment for generating programming environments*, P. Klint, ACM Transactions on Software Engineering and Methodology, pages 176-201, 1993

[14] *The ASF+SDF Meta-Environment: a Component-Based Language Development Environment*, M.G.J. van den Brand and J. Heering and H. de Jong and M. de Jonge and T. Kuipers and P. Klint and L. Moonen and P. Oliver and J. Scheerder and J. Vinju and E. Visser and J. Visser, Proceeding of Compiler Construction 2001 (CC 2001), Springer-Verlag, 2001

Appendix A

Sample Application PB export and Java code

A.1 Application object PB Export code

sample.sra

```
01 $PBExportHeader$sample.sra
02 forward
03 global transaction sqlca
04 global dynamicdescriptionarea sqlda
05 global dynamicstagingarea sqlsa
06 global error error
07 global message message
08 end forward
09
10 global type sample from application
11  end type
12 global sample sample
13
14 on sample.create
15 appname = "sample"
16 message = create message
17 sqlca = create transaction
18 sqlda = create dynamicdescriptionarea
19 sqlsa = create dynamicstagingarea
20 error = create error
21 end on
22
23 on sample.destroy
24 destroy( sqlca )
25 destroy( sqlda )
26 destroy( sqlsa )
27 destroy( error )
28 destroy( message )
29 end on
30
31 event open;open(mainwindow)
32 end event
```

A.2 Window PBExport code

mainwindow.sra

```
01 $PBExportHeader$mainwindow.srw
02 forward
03 global type mainwindow from Window
04 end type
05 type cb_1 from commandbutton within mainwindow
06 end type
07 type sle_1 from singlelineedit within mainwindow
08 end type
09 end forward
```

```
10
11 global type mainwindow from Window
12 int X=1075
13 int Y=485
14 int Width=723
15 int Height=497
16 boolean TitleBar=true
17 string Title="Sample"
18 boolean ControlMenu=true
19 boolean MinBox=true
20 boolean MaxBox=true
21 boolean Resizable=true
22 cb_1 cb_1
23 sle_1 sle_1
24 end type
25 global mainwindow mainwindow
26
27 on mainwindow.create
28 this.cb_1=create cb_1
29 this.sle_1=create sle_1
30 this.Control[]={ this.cb_1,&
31 this.sle_1}
32 end on
33
34 on mainwindow.destroy
35 destroy(this.cb_1)
36 destroy(this.sle_1)
37 end on
38
39 type cb_1 from commandbutton within mainwindow
40 int X=42
41 int Y=221
42 int Width=284
43 int Height=105
44 int TabOrder=20
45 string Text="button"
46 int TextSize=-10
47 int Weight=400
48 string FaceName="Arial"
49 FontFamily FontFamily=Swiss!
50 FontPitch FontPitch=Variable!
51 end type
52
53 event clicked;sle_1.text="Button was clicked"
54 end event
55
56 type sle_1 from singlelineedit within mainwindow
57 int X=42
58 int Y=65
59 int Width=558
60 int Height=93
61 int TabOrder=10
62 boolean AutoHScroll=false
63 long BackColor=16777215
64 int TextSize=-10
65 int Weight=400
66 string FaceName="Arial"
67 FontFamily FontFamily=Swiss!
68 FontPitch FontPitch=Variable!
69 end type
```

A.3 mainwindow PBExport code with XML-markup

```
01 <?xml version="1.0"?>
02
03 <PBExportFile>
04   <PBExportFileHeader>$PBExportHeader$mainwindow.srw</PBExportFileHeader>
05   <forwardDeclarations>
06     <typeDeclaration access="global" identifier="mainwindow" from="Window" within=""/>
07     <typeDeclaration access="" identifier="cb_1" from="commandbutton" within="mainwindow"/>
08     <typeDeclaration access="" identifier="sle_1" from="singlelineedit" within="mainwindow"/>
09   </forwardDeclarations>
10   <typeDefinition access="global" identifier="mainwindow" from="Window" within="">
11     <memberDeclarations>
12       <variable type="int" identifier="X">1075</variable>
13       <variable type="int" identifier="Y">485</variable>
14       <variable type="int" identifier="Width">723</variable>
```

```
15       <variable type="int" identifier="Height">497</variable>
16       <variable type="boolean" identifier="TitleBar">true</variable>
17       <variable type="string" identifier="Title">"Sample"</variable>
18       <variable type="boolean" identifier="ControlMenu">true</variable>
19       <variable type="boolean" identifier="MinBox">true</variable>
20       <variable type="boolean" identifier="MaxBox">true</variable>
21       <variable type="boolean" identifier="Resizable">true</variable>
22       <object type="cb_1" identifier="cb_1"/>
23       <object type="sle_1" identifier="sle_1"/>
24     </memberDeclarations>
25   </typeDefinition>
26   <globalObjectInstantiation type="mainwindow" identifier="mainwindow"/>
27   <on typeIdentifier="mainwindow" event="create">
28     this.cb_1=create cb_1
29     this.sle_1=create sle_1
30     this.Control[]={ this.cb_1,this.sle_1}
31   </on>
32   <on typeIdentifier="mainwindow" event="destroy">
33     destroy(this.cb_1)
34     destroy(this.sle_1)
35   </on>
36   <typeDefinition access="" identifier="cb_1" from="commandbutton" within="mainwindow">
37     <memberDeclarations>
38       <variable type="int" identifier="X">42</variable>
39       <variable type="int" identifier="Y">221</variable>
40       <variable type="int" identifier="Width">284</variable>
41       <variable type="int" identifier="Height">105</variable>
42       <variable type="int" identifier="TabOrder">20</variable>
43       <variable type="string" identifier="Text">"button"</variable>
44       <variable type="int" identifier="TextSize">-10</variable>
45       <variable type="int" identifier="Weight">400</variable>
46       <variable type="string" identifier="FaceName">"Arial"</variable>
47       <variable type="FontFamily" identifier="FontFamily">Swiss!</variable>
48       <variable type="FontPitch" identifier="FontPitch">Variable!</variable>
49     </memberDeclarations>
50   </typeDefinition>
51   <event type="cb_1" action="clicked">
52     sle_1.text="Button was clicked"
53   </event>
54   <typeDefinition access="" identifier="sle_1" from="singlelineedit" within="mainwindow">
55     <memberDeclarations>
56       <variable type="int" identifier="X">42</variable>
57       <variable type="int" identifier="Y">65</variable>
58       <variable type="int" identifier="Width">558</variable>
59       <variable type="int" identifier="Height">93</variable>
60       <variable type="int" identifier="TabOrder">10</variable>
61       <variable type="boolean" identifier="AutoHScroll">false</variable>
62       <variable type="long" identifier="BackColor">16777215</variable>
63       <variable type="int" identifier="TextSize">-10</variable>
64       <variable type="int" identifier="Weight">400</variable>
65       <variable type="string" identifier="FaceName">"Arial"</variable>
66       <variable type="FontFamily" identifier="FontFamily">Swiss!</variable>
67       <variable type="FontPitch" identifier="FontPitch">Variable!</variable>
68     </memberDeclarations>
69   </typeDefinition>
70 </PBExportFile>
```

A.4 Java Implementation

```
01 /*
02    This is the java implementation of
03    the PowerBuilder window object:
04
05    $PBExportHeader$mainwindow.srw
06
07  */
08
09 public class Mainwindow extends Window {
10
11   cb_1 cb_1 = new cb_1();
12   sle_1 sle_1= new sle_1();
13
14   Mainwindow() {
15     X = 1075;
16     Y = 485;
17     Width = 723;
18     Height = 497;
19     TitleBar = true;
```

```
20     Title = "Sample";
21     ControlMenu = true;
22     MinBox =  true;
23     MaxBox  = true;
24     Resizable = true;
025     Control = new PBObject[] {this.cb_1 , this.sle_1};
26  }
27
28   public void eventHandler(String event){
29
30     System.out.println("eventHandler("+event+")");
31
32     if (event.compareTo("cb_1.clicked")==0) {
33       sle_1.text="The 'go' button was clicked";
34     }
35   }
36 }
37
38 class cb_1 extends commandbutton{
39   public cb_1() {
40    X=42;
41    Y=221;
42    Width=284;
43    Height=105;
44    TabOrder=20;
45    Text="Button";
46    TextSize=-10;
47    Weight=400;
48    FaceName="Arial";
49    //FontFamily=Swiss!;
50    //FontPitch=Variable!;
51   }
52 }
53
54 class sle_1 extends singlelineedit {
55   public sle_1() {
56     X=42;
57     Y=65;
58     Width=558;
59     Height=93;
60     TabOrder=10;
61     AutoHScroll=false;
62     BackColor=16777215;
63     TextSize=-10;
64     Weight=400;
65     FaceName="Arial";
66     //FontFamily=Swiss!;
67     //FontPitch=Variable!;
68   }
69 }
```

A.5 Generated Java

A.5.1 Before Post-Procesing

```
001 /*
002    This is the java implementation of
003    the PowerBuilder window object:
004
005    $PBExportHeader$mainwindow.srw
006
007 */
008
009 public class mainwindow extends Window {
010
011   cb_1
012 cb_1  = new cb_1();
013   sle_1
014 sle_1  = new sle_1();
015   mainwindow(){
016     int
017 X  = 1075;
018     int
019 Y  = 485;
020     int
021 Width  = 723;
022     int
```

```
023 Height  = 497;
024     boolean
025 TitleBar  = true;
026     string
027 Title  = "Sample";
028     boolean
029 ControlMenu  = true;
030     boolean
031 MinBox  = true;
032     boolean
033 MaxBox  = true;
034     boolean
035 Resizable  = true;
036
037
038     Control = new PBObject[] {cb_1, sle_1};
039
040   }
041 }
042
043
044
045
046 class cb_1 extends commandbutton{
047
048   cb_1() {
049     int
050 X = 42;
051     int
052 Y = 221;
053     int
054 Width = 284;
055     int
056 Height = 105;
057     int
058 TabOrder = 20;
059     string
060 Text = "button";
061     int
062 TextSize = -10;
063     int
064 Weight = 400;
065     string
066 FaceName = "Arial";
067     FontFamily
068 FontFamily = Swiss!;
069     FontPitch
070 FontPitch = Variable!;
071
072    }
073 }
074
075 class sle_1 extends singlelineedit{
076
077   sle_1() {
078     int
079 X = 42;
080     int
081 Y = 65;
082     int
083 Width = 558;
084     int
085 Height = 93;
086     int
087 TabOrder = 10;
088     boolean
089 AutoHScroll = false;
090     long
091 BackColor = 16777215;
092     int
093 TextSize = -10;
094     int
095 Weight = 400;
096     string
097 FaceName = "Arial";
098     FontFamily
099 FontFamily = Swiss!;
100     FontPitch
101 FontPitch = Variable!;
102
```

```
103    }
104 }
```

A.5.2 After Post-Procesing

```
01 /*
02    This is the java implementation of
03    the PowerBuilder window object:
04
05    $PBExportHeader$mainwindow.srw
06
07 */
08
09 public class mainwindow extends Window {
10
11   cb_1 cb_1  = new cb_1();
12   sle_1 sle_1  = new sle_1();
13   mainwindow(){
14     int X  = 1075;
15     int Y  = 485;
16     int Width  = 723;
17     int Height  = 497;
18     boolean TitleBar  = true;
19     string Title  = "Sample";
20     boolean ControlMenu  = true;
21     boolean MinBox  = true;
22     boolean MaxBox  = true;
23     boolean Resizable  = true;
24
25
26     Control = new PBObject[] {cb_1, sle_1};
27
28   }
29 }
30
31
32
33
34 class cb_1 extends commandbutton{
35
36   cb_1() {
37     int X = 42;
38     int Y = 221;
39     int Width = 284;
40     int Height = 105;
41     int TabOrder = 20;
42     string Text = "button";
43     int TextSize = -10;
44     int Weight = 400;
45     string FaceName = "Arial";
46     FontFamily FontFamily = Swiss!;
47     FontPitch FontPitch = Variable!;
48    }
49 }
50
51 class sle_1 extends singlelineedit{
52
53   sle_1() {
54     int X = 42;
55     int Y = 65;
56     int Width = 558;
57     int Height = 93;
58     int TabOrder = 10;
59     boolean AutoHScroll = false;
60     long BackColor = 16777215;
61     int TextSize = -10;
62     int Weight = 400;
63     string FaceName = "Arial";
64     FontFamily FontFamily = Swiss!;
65     FontPitch FontPitch = Variable!;
66    }
67 }
```

Appendix B

Lexical Scanning Tool

For adding XML-tags to PowerBuilder export files we implemented a lexical scanning tool. This tool was written in the Practical Extraction and Reporting Language, Perl[10]. It scans the export file and inserts XML-tags in the same way we inserted them manually in Section 6.1.2.

The source code of the scanner is given in the following section. The scanner maintains a scan state variable named `$scanState`. This way it can keep track of what it should be scanning for in the current state. For instance after matching the beginning of a type definition in line 145 the `$scanState` variable is set to `"typeDefinition"`. With this scan state the scanner will only match an `end type` construct in line 64 or in an `event`, `variable` or `object` construct in lines 94-108. The matching of export file constructs is done using regular expressions. See [10] for details about regular expressions in Perl.

When unexpected input is encountered a warning is printed out in line 238. This way we get a warning message when a line is scanned that is not recognized by the scanner. During the creation of the lexical scanner this was a handy feature. We built the scanner around a specific export file. When we tried scanning other export files the warning message would indicate that the scanner did not recognize a construct in the export file. We then added code to the scanner so it could process the newly found construct. This process was repeated until the scanner was able to process all the export files that are part of one PowerBuilder application, with the exception of datawindows. Datawindows are specified in an entirely different format that requires a separate lexical scanner. In [11] it is shown how datawindows can be transformed to HTML using XML, XSL and JavaScript.

The drawback of using a lexical scanner written in Perl is that the script is hard to maintain. This is due to the fact that it is easier to write a regular expression in Perl than it is to read and understand it at a later time.

B.1 Source code

```
001 # Applies XML annotation to PoweBuilder export files
002 # Date:   13-04-2001
003 # Author: Bas Toeter
004 ##########################################
005 #
006 # pb2xml.pl <filepattern>
007 #
008
009 if ( ((scalar @ARGV < 1) | (scalar @ARGV > 3) )){
010   print "USAGE: # pb2xml.pl <filepattern> \n";
```

```
011 }
012
013 $pattern = $ARGV[0];
014 open(FL,"dir /b $pattern |");           # get file that match the pattern
015 while(<FL>){
016   chop();
017   $source = $_;
018   $source =~ /.*\.(...)/;               # get the extension
019   $exportType=$1;                       # store the extension
020   $target = "$source.xml";
021
022   open(PB,"<$source") or die "can't open input file $source";
023   open(OUT,">$target") or die "can't open output file $target"; ;
024
025   $scanState="root";                    # initialize scan state variable
026   $iCount=0;                            # counts number of indents
027   $iString="";                          # indentation string
028
029   &Iprint("<?xml version=\"1.0\"?>\n");
030   &Iprint("<?xml:stylesheet type=\"text/xsl\" href=\"$exportType.xsl\" ?>\n");
031   &Iprint("<PBExportFile>\n");
032   &Indent(+2);
033
034   while(<PB>){
035     chop();
036     if ( (/^\s*$/) || (/^\/\//) ) {
037       #just skip empty lines and lines with comments
038     }
039     elsif    (/(^\$PBExportHeader\$.*\.(...))/){
040       &Iprint("<PBExportFileHeader>$1</PBExportFileHeader>\n");
041     }
042     elsif (/(^\$PBExportComments\$.*)/) {
043       Iprint("<PBExportComments>CDATA[$1]</PBExportComments>\n");
044     }
045     elsif (/^end subroutine/) {
046       &Iprint("</powerScript>\n");
047       &Indent(-2);
048       &Iprint("</subroutineDefinition>\n");
049       $scanState="root";
050     }
051     elsif (/^end function/) {
052       &Iprint("</powerScript>\n");
053       &Indent(-2);
054       &Iprint("</functionDefinition>\n");
055       $scanState="root";
056     }
057     elsif ( (/^end on/) && ($scanState eq "onEvent")) {
058       &Indent(-2);
059       &Iprint("</onDeclaration>\n");
060       $scanState="root";
061     }
062     elsif ( (/end type/) && ($scanState eq "typeDefinition")) {
063       &Indent(-2);
064       &Iprint("</memberDeclarations>\n");
065       &Indent(-2);
066       &Iprint("</typeDefinition>\n");
067       $scanState="root";
068     }
069 # onDeclaration
070     elsif ($scanState eq "onDeclaration") {
071       if    (/^end on/)   {
072         &Indent(-2);
073         &Iprint("</onDeclaration>\n");
074         $scanState="root";
075       }
076     }
077 #  forwardDeclarations
078     elsif ($scanState eq "forwardDeclarations") {
079       if (/^end forward/) {
080         &Indent(-2);
081         &Iprint("</forwardDeclarations>\n");
082         $scanState="root";
083       }
084       elsif ((/^end type/) && ($scanState eq "forwardDeclarations")) {
085         # ignore 'end type'
086       }
087       elsif (/^(global)*\s*type\s*([^ ]*)\s*from\s*([^ ]*)\s*(within)*\s*(.*)/)  {
088         &Iprint("<typeDeclaration access=\"$1\" identifier=\"$2\" from=\"$3\" within=\"$5\"\/> \n");
089       }
090     }
```

```
091 #  typeDefinition
092     elsif ($scanState eq "typeDefinition") {
093       if (/^event ([^ ]*) \( (.*) \)/) {
094         &Iprint("<eventDeclaration identifier=\"$1\">\n");
095         &Iprint("  <parameters>$2</parameters>\n");
096         &Iprint("</eventDeclaration>\n");
097       }
098       elsif (/^event (.*) (.*)/){
099         &Iprint("<eventDeclaration identifier=\"$1\">\n");
100         &Iprint("  <parameters>$2</parameters>\n");
101         &Iprint("</eventDeclaration>\n");
102       }
103       elsif (/(.*) (.*)=(.*)/) {
104         &Iprint("<variable type=\"$1\" identifier=\"$2\">$3</variable>\n");
105       }
106      elsif (/(.*) (.*)/) {
107         &Iprint("<object type=\"$1\" identifier=\"$2\"/>\n");
108       }
109     }
110 #
111     elsif (/^on ([^\.]*)\.create/) {
112       &Iprint("<onDeclaration ownerIdentifier=\"$typeLastDefined\" event=\"create\">\n");
113       &Indent(+2);
114       &Iprint("//PowerScript\n");
115       $scanState="onDeclaration";
116     }
117     elsif (/^on ([^\.]*)\.destroy/) {
118       &Iprint("<onDeclaration ownedByType=\"$typeLastDefined\" event=\"destroy\">\n");
119       &Indent(+2);
120       &Iprint("//PowerScript\n");
121       $scanState="onDeclaration";
122     }
123
124     elsif (/^event (.*)\;/) {
125       &Iprint("<eventHandlingCode type=\"$typeLastDefined\" action=\"$1\">\n");
126       &Indent(+2);
127       &Iprint("//PowerScript event handling code for $typeLastDefined.$1 \n");
128       $scanState="event";
129     }
130     elsif (/^type prototypes/) {
131       &Iprint("<typePrototypes>\n");
132       &Indent(2);
133       $scanState="typePrototypes";
134     }
135     elsif (/^forward$/) {
136       &Iprint("<forwardDeclarations>\n");
137       $scanState="forwardDeclarations";
138       &Indent(+2);
139     }
140     elsif (/^(global)*\s*type\s*([^ ]*)\s*from\s*([^ ]*)\s*(within)*\s*(.*)/)  {
141       $scanState="typeDefinition";
142       $typeLastDefined=$2;
143       &Iprint("<typeDefinition access=\"$1\" identifier=\"$2\" from=\"$3\" within=\"$5\"\> \n");
144       &Indent(+2);
145       &Iprint("<memberDeclarations>\n");
146       &Indent(+2);
147       }
148     elsif(/^global ([^ ]*) ([^ ]*)/) {
149       &Iprint("<globalObjectInstantiation type=\"$1\" identifier=\"$2\"\/>\n");
150     }
151     elsif(/^type variables/) {
152       &Iprint("<typeVariables>\n");
153       &Indent(+2);
154       $scanState="typeVariables";
155     }
156 #  typeVariables
157     elsif  ($scanState eq "typeVariables") {
158       if(/^end variables/) {
159         &Indent(-2);
160         &Iprint("</typeVariables>\n");
161
162         $scanState="root";
163       }
164       elsif (/^PROTECTED:/) {
165         $accessQualifier="protected";
166       }
167       elsif (/([^\s]*)\s*([^\s]*) = ([^ ]*)/) {
168         &Iprint("<variable access=\"$accessQualifier\" type=\"$1\" identifier=\"$2\">$3</variable>\n");
169       }
170       elsif(/([^\s]*)\s*([^\s]*)/){
```

```
171         &Iprint("<variable access=\"$accessQualifier\" type=\"$1\" identifier=\"$2\"></variable>\n");
172       }
173     }
174     elsif(/^forward prototypes/) {
175       $scanState="forwardPrototypes";
176       &Iprint("<forwardPrototypes>\n");
177       &Indent(2);
178     }
179 #  forwardPrototypes
180     elsif ($scanState  eq "forwardPrototypes") {
181       if (/^end prototypes/) {
182         &Indent(-2);
183         &Iprint("</forwardPrototypes>\n");
184         $scanState="root"
185       }
186       elsif (/([^ ]*) subroutine ([^ ]*) \((.*)\)\;*/) {
187         &Iprint("<subroutineDeclaration access=\"$1\" identifier=\"$3\">\n");
188         &Iprint("  <arguments>$4</arguments>\n");
189         &Iprint("</subroutineDeclaration>\n");
190       }
191       elsif (/([^ ]*) function ([^ ]*) ([^ ]*) \((.*)\)\;*/) {
192         &Iprint("<functionDeclaration access=\"$1\" identifier=\"$3\" type=\"$2\">\n");
193         &Iprint("  <arguments>$4</arguments>\n");
194         &Iprint("</functionDeclaration>\n");
195       }
196     }
197     elsif( (/^end prototypes/) && ($scanState  eq "typePrototypes")){
198       &Indent(-2);
199       &Iprint("</typePrototypes>\n");
200     }
201     elsif ( (/^end event/) && ($scanState eq "event")) {
202       &Indent(-2);
203       &Iprint("</eventHandlingCode>\n");
204       $scanState="root";
205     }
206     elsif (($scanState eq "functionDefinition") ||
207            ($scanState eq "subroutineDefinition") ||
208            ($scanState eq "event") ||
209            ($scanState eq "onEvent") ){
210       #&Iprint("$_\n");
211     }
212
213     elsif (/([^ ]*) subroutine ([^ ]*) \((.*)\)\;*/) {
214       $scanState="subroutineDefinition";
215       &Iprint("<subroutineDefinition access=\"$1\" identifier=\"$3\">\n");
216       &Iprint("  <arguments>$4</arguments>\n");
217       &Iprint("  <powerScript>\n");
218       &Iprint("    //handling Code $3($4) \n");
219       &Indent(+2);
220     }
221     elsif (/([^ ]*) function ([^ ]*) ([^ ]*) \((.*)\)\;*/) {
222       $scanState="functionDefinition";
223       &Iprint("<functionDefinition access=\"$1\" identifier=\"$3\" type=\"$2\">\n");
224       &Iprint("  <arguments>$4</arguments>\n");
225       &Iprint("  <powerScript>\n");
226       &Iprint("    //handling Code $3($4) \n");
227       &Indent(+2);
228     }
229       elsif (/^on ([^ ]*)\;/) {
230       &Iprint("<onDeclaration ownerIdentifier=\"$typeLastDefined\" event=\"$1\">\n");
231       &Indent(+2);
232       $scanState="onEvent";
233     }
234     else {
235      print "UNPROCESSED in $source: $_\n";
236     }
237   }
238   &Indent(-2);
239   &Iprint("</PBExportFile>");
240   close(PB);
241   close(OUT);
242   print "Wrote file: $target \n";
243 }
244
245 sub Indent {
246   my $change = shift;
247   my $index;
248   $iString="";
249   $iCount += $change;
250   for ($index=0;$index<$iCount;$index++){
```

```
251       $iString .=" ";
252     }
253 }
254 sub Iprint {
255   my $text = shift;
256   print OUT "$iString$text";
257 }
```