AUTOMATED ADDITION OF ARCHITECTURAL SOFTWARE QUALITY ATTRIBUTES

SARTHAK K. GROVER

Bachelor of Science, Computer Engineering

University of Arizona

Dec, 2003

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

June 2007

This thesis has been approved for the

Department of **ELECTRICAL AND COMPUTER ENGINEERING**

and the College of Graduate Studies by

_____

Thesis Committee Chairperson, Dr. Nigamanth Sridhar

_____

Department/Date

_____

Dr. Chansu Yu

_____

Department/Date

_____

Dr. Yongjian Fu

_____

Department/Date

To my dear parents...

ACKNOWLEDGMENTS

Firstly, I would like to thank my thesis advisor and mentor, Dr. Nigamanth Sridhar. His encouragement and support has got me a long way from when I first started on this and helped me drive this to completion.

I would also like to thank the committee members, Dr. Chansu Yu and Dr. Yongjian Fu who have been very accommodating during the progress of my work. Many thanks to the Department of Electrical and Computer Engineering at Cleveland State University for helping me make this happen.

Many thanks to my friends and colleagues in La Crosse, who patiently watched me spend beautiful evenings sitting in my cube and working on my thesis and understanding that I couldn't join them for tennis, biking, geocaching or trips to the Goose Island.

Lastly, many thanks to my parents, sister and brother-in-law for keeping a check on me on how my thesis work was progressing. It was a strong motivation to keep them from asking about it every day.

AUTOMATED ADDITION OF ARCHITECTURAL SOFTWARE QUALITY ATTRIBUTES

SARTHAK K. GROVER

ABSTRACT

Software development is an ongoing and interactive process with the end goal of providing deliverables to stakeholders which satisfy the criteria set by them. The process begins with elicitation of high-level requirements which are then translated into specific deliverables. Based on these deliverables, the system architecture is then designed which is used to guide the system implementation. Requirements are categorized as Functional Requirements (FR's) and Non-Functional Requirements (NFR's). NFR's are not tangible in nature and are often not implemented in systems due to lack of specifications. These are often added or removed from the envisaged system during different stages of development which comes with a high cost of reanalyzing the architecture and core system redesign.

In this thesis, we present a tool which provides automated addition of code for implementing Non-Functional requirements for any generic system without redesigning the system architecture. We also provide a semantic data representation which can be reused to extend this work over various platforms and languages. Our tool provides a non-invasive and user-customizable framework for adding these quality attributes to any system without modifying any of the original core modules.

TABLE OF CONTENTS

LIST OF TABLES

Listings

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Introduction

For any system being created, it is necessary to go through certain steps to ensure that the final system is developed correctly. The process begins with requirements definition or inception as shown in Figure 1. This is followed by the elaboration stage wherein a feasibility study is done based on the requirements which leads into a preliminary system architecture design.

The software architecture of a system or program is defined as the structure or structures of the system comprising of software elements, the externally visible properties of those elements and the relationships amongst them [15].

One of the most complex processes during system development is the mapping of requirements into a system architecture. The goal of the architecture is to provide a higher level of abstraction showing the relationships between various components, applicable constraints and other system attributes. Since the architecture of a system constrains the Non-Functional Requirements (NFR's), the design decisions taken during this stage have a big

Figure 1: Software Development Process begins with requirements elicitation leading to architecture and system design.

impact on the potential for implementing NFR's.

As an example, when a system is being design to work around a core entity, it may lead to bottlenecks, consequently influencing the performance of the system. As a result, during the system architecture design, it becomes important to balance various requirements and their corresponding trade-offs. In situations like these, implementation of crucial NFR's comes at a high cost of either redesigning the system architecture or excluding the non-functional requirement and consequently bearing the cost of not having it in the system.

Current software architecture design methods do not make an explicit distinction between conventional architectural concerns that can be localized using current architectural abstractions and architectural concerns that crosscut multiple architectural components [2].

A new and developing software methodology *Aspect Oriented Software Development (AOSD)* plays an important role here. The main goal of AOSD is addressing cross-cutting concerns.

These concerns are inherent properties of conflicting/crosscutting requirements and can be classified into two types: scattering and tangling [10]. Scattering is caused when the implementation of a requirement is spread across various components of a system while tangling occurs due to implementation of multiple requirements by a single module. Figure 2 shows an example where quality attribute implementations are spread over various modules leading to code scattering and tangling.

Aspect Oriented Programming (AOP) which is a facet of AOSD [22] provides a mechanism for addressing these cross-cutting concerns and automatically incorporating them into a system. It does not act as a replacement to the core functionality of the system

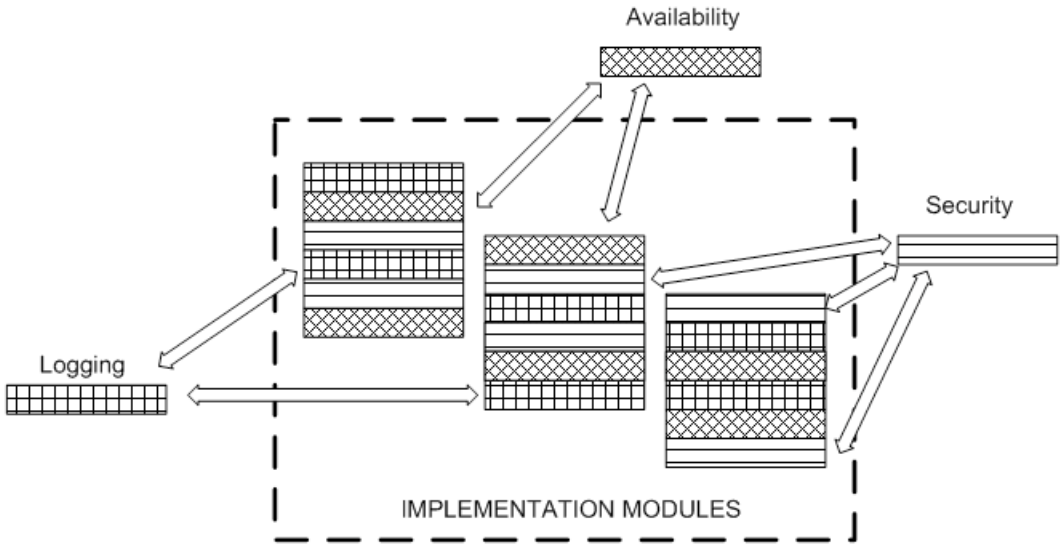

Figure 2: Code tangling occurs when modules in a system simultaneously interact with several requirements such as Logging, Availability and Security.

but enhances the core functionality without modifying the inner code. One can think of an aspect as a wrapper to the actual system which not only inherits the system functionalities but provides support from outside to enhance these functionalities.

1.2 Problem Statement

The process of software development begins with elicitation of requirements. Based on the requirements specifications and system feasibility the architecture is identified and designed, making the requirements definition one of the most important stages in software development.

Requirements can be of two types Functional Requirements (FR's) and Non-Functional Requirements (NFR's). While functional requirements can be identified and tested in tangible ways, the non-functional requirements elude the system requirements specifications. For example, in a banking scenario, the system is expected to handle account debit, credit, interest etc but the non-functional requirements such as quality, performance are often not

included in the specifications.

As a result, due to lack of specifics, NFR's are often not implemented in systems. Also, they are difficult to implement and there is no systematic way of achieving them during system development [9].

1.3 Thesis

We defend the following thesis:

An automated aspect-generation system which creates code for application in any generic system to implement specific NFR's and can be customized based on user input. Also, a semantic data structure using XML is proposed, to represent user-defined customization information for the NFR

1.4 Solution Approach

We tackle the problem of implementing NFR's once a system has been designed and implemented. Since changing the system architecture at this stage can be a very expensive option, we used AOP to implement the NFR's. As a proof-of-concept, our tool implements two quality attributes:

- *Message Logging*: Logs all actions in a system
- *Atomic Transaction*: Provides a transactional operation within the system

Our approach has been to make the process as automated as possible and at the same time leaving room for user customization. As a result of this, code-generation for each of these quality attributes (QA) requires user input via a unique Graphical User Interface (GUI) based on the QA being implemented.

The system takes the user customization input and transforms the information into an XML structure which is then used in collaboration with the back-end logic of the specific attribute to generate AspectJ code which can then directly be deployed into the system. Deliverables of this tool are:

- Automatically generated aspect code ready to be deployed into a given system, providing immediate availability of the functionality
- XML file containing information obtained from the user built into an organized structure

1.5 Contributions

Our work provides a tool for implementing NFR's in existing systems. As part of the research, NFR's were identified as non-tangible requirements which are difficult to specify and implement. Our work aims at making NFR's more approachable and feasible for systems which are already deployed. Three primary contributions in this area are:

- An automated tool which generates ready-to-use code for implementing specific NFR's in any system. This tool generates deployable modularized code which can easily be incorporated with the targeted system.
- Customizable NFR specification based on user input using a GUI. This increases usability and usefulness of the quality attribute being implemented giving more control in the hands of the user of the tool.
- Semantic data structure for representing NFR specific information which is platform/system independent and can be reused for extending behavior in different languages. This semantic structure provides guidance for future extension of this work in various environments.

1.6 Organization of the Thesis

This thesis is organized as follows. Chapter 1 introduces the software development domain, discussing the challenges in the development process and presents our thesis towards addressing these challenges. Chapter 2 provides an overview of Aspect Oriented Programming and XML, technologies which are utilized in our solution. Chapter 3 discusses research work done in this area and analyzes the work done by others. Chapter 4 gives a detailed understanding of the problem that is being tackled as well as discusses the design decisions made by us in our solution implementation. Chapter 5 is a high-level overview of our design and provides a brief overview of the quality attributes that are implemented by our tool. Chapter 6 goes into a detailed discussion of the XML structure that is used by our tool. Chapter 7 goes into the details of implementation of our core module which generates code for quality attributes and provides a complete example using an actual scenario. Finally, Chapter 8 discusses the limitations, future work and conclusion of our thesis work.

CHAPTER II

TECHNOLOGY OVERVIEW

2.1 Overview of Aspect Oriented Programming (AOP)

There are several programming problems in which neither procedural nor Object-Oriented Programming (OOP) methodologies are suffice to capture important design decisions for implementation. Implementation of such a design is spread throughout the code and leads to code-tangling which essentially makes it difficult to develop and maintain code [16].

A *concern* is defined as a consideration/requirement that must be addressed in the system being developed before it can be accepted. The core concerns are defined by the requirements but the concerns involving quality attributes such as performance, logging etc are not taken into consideration by these requirements. These concerns spread throughout the system spanning multiple modules and are referred to as *cross-cutting concerns*.

Even though OOP is often used to manage core concerns, it does not address cross-cutting concerns efficiently specially when dealing with complex applications. Due to the complexity, often the cross-cutting concerns are coupled with core modules [19]. As a result of this, anytime there is any behavior modification for a cross cutting concern or addition of new concerns, there is a dependency on the core modules which are then required to be updated. AOP complements and

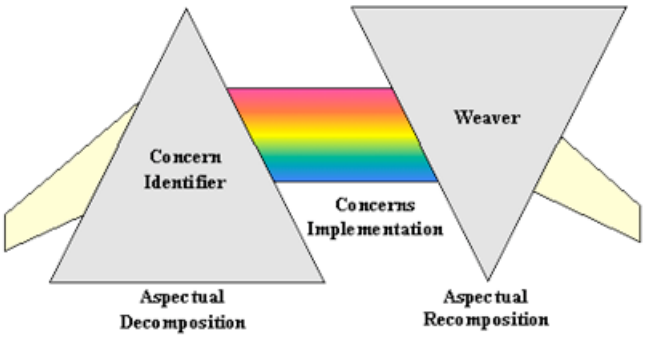

Figure 3: In the AOP methodology, process begins with identification of concerns. These concerns are then implemented and weaved with the core system. Reproduced from *AspectJ in Action* [20]

.

assists OOP methodology by addressing these crosscutting concerns as seen in Figure 3.

AOP resolves this by means of generating another level of stand-alone modularity which is referred to as aspects. These aspects are the interface between the cross-cutting concern and the core modules. Aspects implement the cross-cutting concerns externally instead of modifying any core modules. An *aspect weaver/compiler* then weaves the aspect and core modules together, leading to a much simplified system architecture.

2.1.1 Implementation of Aspects

Aspects are stand-alone plug-ins that can be compiled with the core code of the system using an *aspect weaver* to include the behavior exhibited by the aspect into the system. Following the same principle, if some behavior needs to be removed/isolated, all it takes is to remove the aspect module and recompile the source code and the system retains its original behavior (Figure 4). This plug-and-play kind of nature of aspects saves a lot of effort in terms of complexity of the system along with the refactoring effort required to update the system with these qualities.

Here is a simple example of implementation of aspects in a system. Let us take a banking system wherein all account transactions get logged and the modules of the system can be seen in Figure 5. The way this system would be developed will include a logging statement within each of the transaction type methods such as:

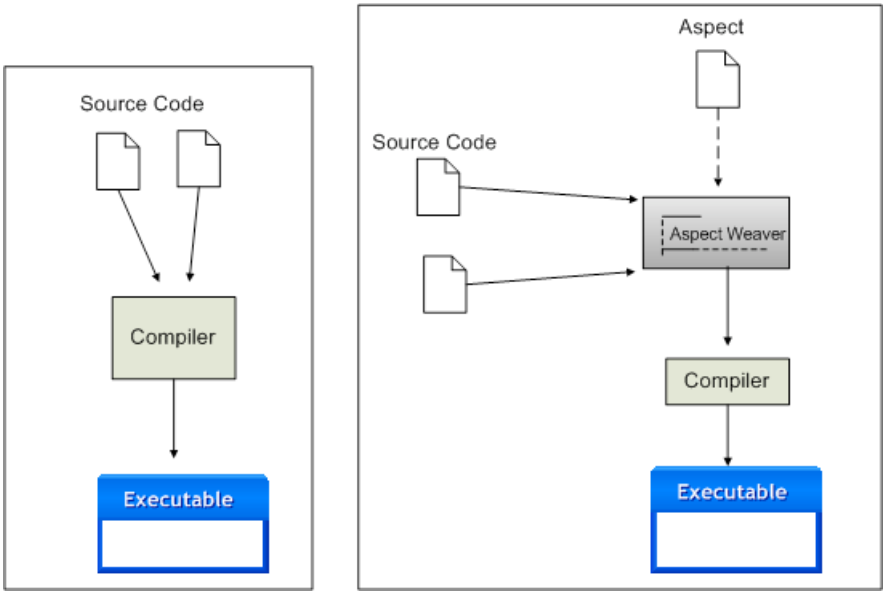

Figure 4: In the traditional OOP methodology, source code is compiled using a language specific compiler and an executable is generated, as seen on the left. When using AOP, aspects are weaved with the core code using an aspectual weaver and the meshed code is then compiled to generate an executable.

- account_checking.deposit()
- account_saving.deposit()
- account_checking.withdrawal()
- account_saving.withdrawal()

As discussed earlier, one of the issues here is refactoring of the code to implement any changes. As a result of this, anytime the logging behavior will need to be modified, each of these core modules will be required to be updated introducing potential risk of causing other bugs.

AOP would address this issue using a *logging aspect*. This aspect would be directed to capture any activity in the *deposit* and *withdrawal* methods irrespective of the account type that is calling it. Consequently, anytime there is a transaction, irrespective of the *account_type*, this aspect code would capture the state of the account and log it. Also if at anytime this behavior would need to be modified, the core modules will remain unchanged and the behavior modification would just need to be made in the aspect code.

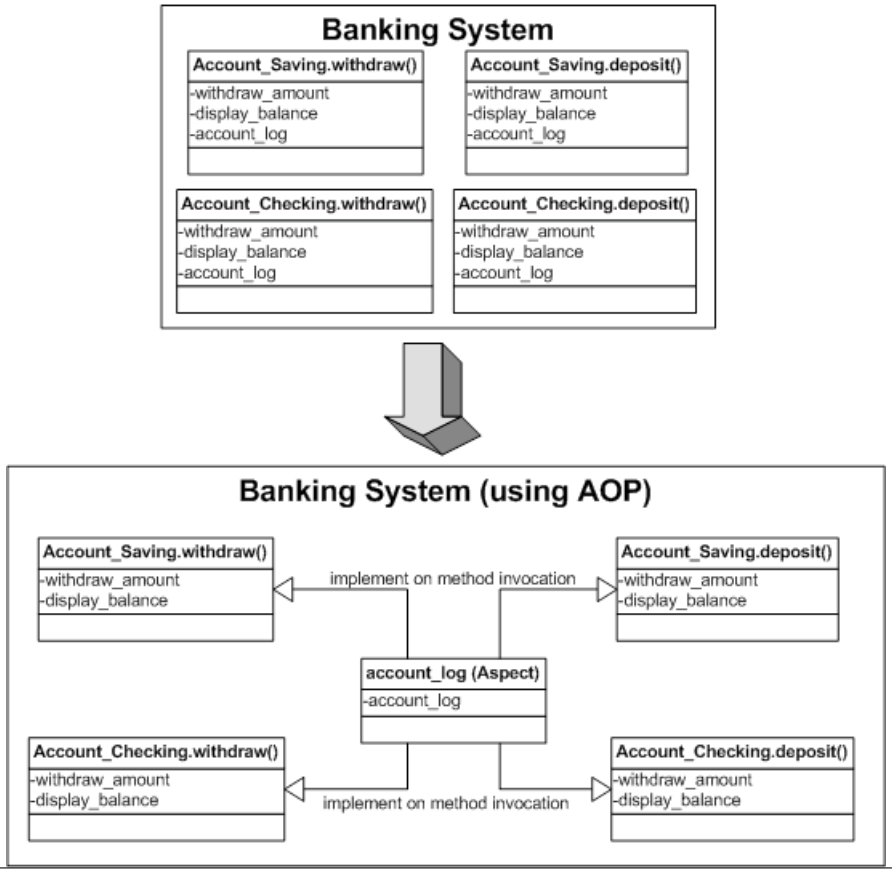

Figure 5: A comparison to show the difference in approach when implementing a characteristic such as logging in a Banking System. In most scenarios, each module contains its own logging functionality making it difficult for code refactoring. In an AOP approach, a single aspect catches calls to each of these modules and logs the information centrally.

2.1.2 AOP Terminology

The program structure of an aspect is very similar to an implementation in OOP. AOP takes on different semantics based on the flavor of implementation that is selected. For example, AspectJ program syntax would vary from AspectC syntax but the overall program structure remains the same. Table 1 compares some of the important parts of an OOP program versus its equivalent using AspectJ.

Besides the system specific syntax, general concepts of Aspect-Oriented programming consists of three components. The pattern in the core code which needs to be captured is also referred to as a *joinpoint*. A logical composition of such patterns/joinpoints is called a *pointcut*. An *advice* is the "body" of the method or the implementation that is done when a certain joinpoint is caught.

Object Oriented Structure	**Aspect Oriented Structure**
Class code unit that encapsulates methods and attributes E.g. *public class MyClass* {	Aspect code unit that encapsulates pointcuts, advice, and attributes E.g. *public aspect MyClass* {
Method signatures define the entry points for the execution of method bodies E.g. *private boolean myMethod* {	Pointcuts define the set of entry points (triggers) in which advice is executed E.g. *pointcut myInput : execution(private boolean myMethod)*
Method bodies implementations of the primary concerns E.g. int myVar = 0 ; if(myVar) return true; else return false;	Advice implementations of the cross cutting concerns E.g. before() : myInput() System.out.println(”myMethod”);
Compiler converts source code into object code	Weaver instruments code (source or object) with advice

Table I: Comparison of Object-Oriented Program Structure vs Aspect Oriented Program Structure.

To better understand how AOP works, Figure 6 shows a simple program explaining its implementation. In Line 1, the *aspect* is defined as “HelloAspectWorld”. Line 3 defines a pointcut named *introduction* which captures *execution* of any call to the *print* method. Notice the use of wildcard is allowed here to permit capturing patterns beyond specific classes and types.

The *advice* is then defined for *before* and *after* the pointcuts. When using a *before* type of advice, the program performs the operations specified within the advice before entering the *print* method. Similarly, *after* advice is executed once the system exits the *print* method. At this point, the operations specified within the advice are performed.

Taking as an example, if the same program has a *print()* method which prints out “Hello World!”, then this aspect modifies its functionality to perform other actions prior to and after execution of the original method. The output for this can be seen in Figure 6. A third type of advice, *around* permits the system to perform an *alternate* set of operations than what is specified in the actual program. This kind of by-pass of original contents of the method can be very useful specially for unit-testing

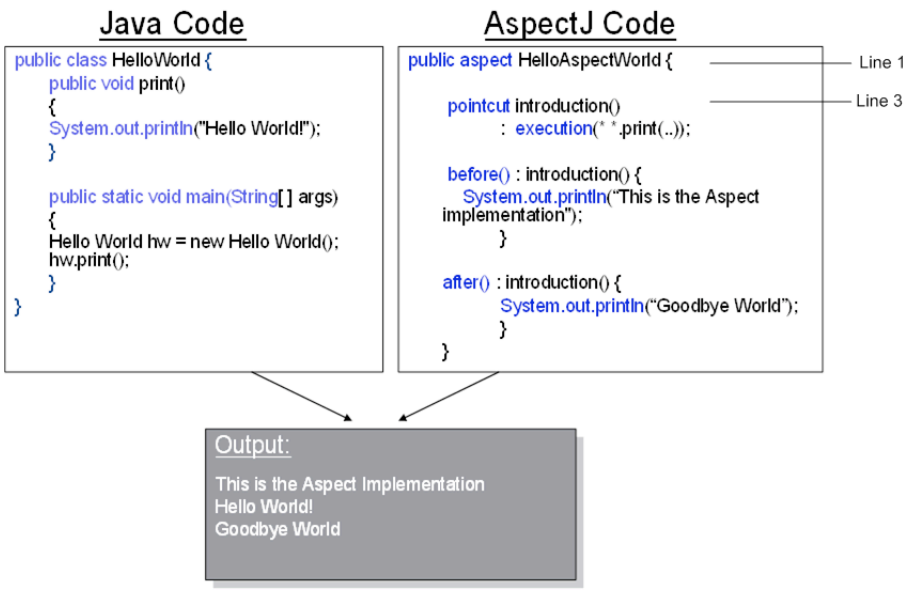

Figure 6: A simple example showing an application of aspects. The Java Code is meshed with the AspectJ Code and when compiled together, the output of the Java code includes implementations of the aspect.

2.2 Overview of eXTensible Markup Language (XML)

XML stands for e*X*tensible *M*arkup *L*anguage. It was developed by an XML Working Group (originally known as the SGML Editorial Review Board) formed under the auspices of the World Wide Web Consortium (W3C) in 1996 [7]. It is a general-purpose markup language developed to facilitate data availability across a variety of systems making it platform independent. XML is identified using a set of tags and semantic information which is identified by its *Documentation Type Definition* (DTD). A DTD is a schema of data representation that is applied to the associated XML file.

This representation of data is based on two attributes. Firstly, the data should be *well formed*. This attribute implies that the XML document cannot contain incomplete data structures. In an XML file, *structure tags* are used to encapsulate the data. So a *well formed* document must ensure that each tag that is opened has a corresponding closing tag. The second attribute that is necessary for the correctness of an XML document is its *validity*. To ensure this, the XML document must adhere to the schema defined in the DTD. The DTD identifies the tags and the relationship between them and if an XML does not follow those hierarchical definitions, the structure becomes invalid.

```
<?xml version = "1.0" encoding="UTF-8">
<!-- Begin the XML -->
<Vehicle> <!-- This is the root element -->
   <Type>Sedan</Type> <-- Child Node for Vehicle -->
   <Make>Honda</Make>
   <Model>Accord</Model> <-- Parent Node for Edition -->
       <Edition>Sports</Edition>
       <Cylinders>V8</Cylinders>
   <Year>2007</Year>
</Vehicle> <!-- This is the closing tag for the root-->
```

Figure 7: A sample XML file showing a data structure for automobiles.

There are several advantages of using XML even though it may not be most useful for specific implementations. Flexibility is one of the key strengths of XML. Applications handling large amounts of data can use XML to standardize on information organization and even then there is room for extending the structure by addition of new tags or modification of base tags. Also, XML is completely portable across various platforms making this a universal choice for data exchange. There is no dependency on special libraries or proprietary applications to parse and implement information from XML documents since these are just plain text files. This is specially advantageous when information needs to be sent across incompatible hardware or software platforms such as B2B applications, e-mail clients, mobile devices etc. XML uses human language and is not dependent on computer restricted keywords. This is beneficial since it makes it an easily readable and understandable document. It also simplifies the coding process since information relationship is explicitly defined and data is not specifically tied to the logic and language of implementation.

2.2.1 XML file structure

XML documents generally begin with an optional header which contains information on the XML such as character encoding and external dependencies. XML supports Unicode Transformation Format(UTF) encodings UTF-8 and UTF-16 but support for UTF-32 is optional [5]. Every XML document must contain a *root* tag. This tag forms the basis of a tree-like structure of the XML schema. Due to its hierarchy based structure, it is often convenient to indent the children tags to the parent tags making it visually more descriptive. Comments can be placed anywhere within the document using the *“<!- -”* tag. Figure 7 shows an example of an XML structure.

CHAPTER III

RELATED WORK

3.1 Discussion

This chapter discusses the existing work which has been done in tackling issues related to implementation of Non-Functional Requirements (NFR's). Implementing new requirements into an existing system is a common scenario but being non-tangible, NFR's become difficult to implement and are often not accounted for within the system architecture. This leads to several limitations in implementing these in a deployed system. Some tools have been developed to address issues pertaining to NFR's but the domain in which these function are based on models and design. This makes our tool both unique and practical due its direct impact and an almost instantaneous response-time for deployable code availability.

The research done in these areas primarily break into four categories. Firstly, the implementation of NFR's using Aspects which has been aimed at designing the systems based on knowledge and intent of implementing NFR's. Research done in this area pertains to implementation of NFR's at an early stage of development. Secondly, a lot of research has been done towards system architecture models and modeling frameworks for implementing NFR's. Thirdly, research has been done regarding use of *component containers* for implementation of NFR's. Lastly, the process of

automating code generation for aspects and NFR's done by other researchers will be discussed.

3.2 Implementing NFR's using Aspects

One of the approaches taken, aims at creating reliable application programs using Aspect Oriented Programming (AOP) [17]. To ensure reliability in an application, the design stage incorporates this quality attribute. The system is then developed with reliability but with this approach the developed system can have malicious effects due to development of Functional Requirements (FR's) and NFR's simultaneously. The author recommends that these requirements should be implemented separately and provides a solution to include reliability into the system effectively. In the design of this approach, the NFR (reliability) is built using AOP and then weaved with the functional requirements.

To implement this NFR, a *Checkpoint-recovery scheme* is applied and modified due to its limitations. In the *Checkpoint-recovery scheme*, it is not possible to restore the state of an application just before an unpredictable halt. In the application of this scheme, volatile memory contents are copied over to non-volatile memory periodically. But in the event of an unexpected halt, application context can only be recovered till the last checkpoint. There is a memory trade-off by increasing the number of checkpoints making that an unfeasible option.

Under the proposed scheme, the system uses not only the checkpoint information but also creates a threshold on usage of memory writing operations in order to prevent performance issues. In this scheme, every user operation is recorded to a non-volatile memory location and the memory backup process begins only when the CPU utilization goes below the specified threshold. To avoid large memory issues, the user operations in memory are deleted once the backup has been completed.

To demonstrate the approach, a recovery functionality is added to a Graphical User Interface (GUI) based application using AOP thus weaving the NFR (reliability) with the FR (GUI application source) to develop a reliable application. AspectJ was utilized to generate the aspects for introducing this functionality. In summary, the tool demonstrated two important properties of AOP. Firstly,

AOP permitted separate implementation of FR's and NFR's. Secondly, AOP is effective for adding quality attributes to a system.

Another approach [11] aims at accommodating Aspect Oriented Software Development (AOSD) in existing software development processes by adapting and extending current Object-Oriented Programming (OOP) methodology in an aspectual context. This study also aims at identifying and modeling cross-cutting concerns at an early stage in the software development life cycle. The authors lay emphasis on the requirements elicitation process being crucial since it is important to identify NFR's in the early stage.

The authors illustrate application of AOP in implementing NFR's using a use case scenario of a *conference room reservation system* applying an iterative development model. AspectJ is used to implement the application wherein the authors go through the entire process of software development starting from defining the requirements followed by the design to address cross-cutting concerns in the system. After implementing the system, the authors conclude that cross-cutting concerns are introduced in the requirements phase itself and can take different forms as the software is being developed. Identifying these concerns should happen at an early stage which can lead to good modularity as well as making the code reusable.

3.3 Modeling NFR's

A lot of research has been done related to modeling of NFR's to increase their visibility in early stages of the system design. One such tool, *NFR-Assistant*, has been built keeping the primary focus on the design of the system architecture. This Java-applet based tool allows for representation of NFR's in the system architecture, providing design trade-off analysis, design alternatives and evaluation of achievements of the NFR [26]. This Computer Aided Software Engineering (CASE) tool helps in planning of the system architecture providing developers with flexibility to incorporate NFR's into the final system.

The authors suggest a NFR Framework which provides ontology for addressing various resource constraints as well as quality attributes desired by the customer. This framework uses a

goal-oriented approach classifying goals as *softgoals*, *operationalizing goals* and *claim softgoals*. *Softgoals* are defined as generic customer requirements such as "user friendliness". The *operationalizing goals* on the other hand, are design options for achieving the soft goals. Finally, the *claim softgoals* provide rationale and explain context for the *softgoals*.

The developed tool *NFR-Assistant* brings together the semantics and heuristics of a NFR framework into a software development environment which is used by developers in analyzing NFR's. The tool provides traceability for various NFR's and at the same time gives user the control, providing an interactive evaluation environment.

Another study proposes a design notation based on standard Unified Modeling Language (UML) for separating the business logic from the design of aspect code [13]. This utility also supports automatic generation of aspect-oriented code skeletons based on the design model and another tool exploiting the system architecture. The authors believe that much focus regarding aspects has been done in implementation and there is a need to address mapping of designing models to programming models. To avoid inconsistencies amongst the design and implementation, the tool provides validation and AspectJ code generation of models.

Amongst the various requirements of this tool, the primary one was to extend UML to provide a simplified and straightforward support for AOP. The authors provide a notation based on package level decomposition using UML For Aspects (UFA) [14]. Since UFA does not provide support for AspectJ in particular, additional support was added to the UFA to include AspectJ concepts by modifying the syntax of UFA to conform with UML standards. The notation consists of three elements, the *base package* which represents the core modules, the *aspect package* representing the cross-cutting concerns and *connector* to link the *base package* and the *aspect package*.

The code generator in this tool first validates the model and then generates AspectJ code for the validated model. The limitation of this tool is that it is a one-way generation and development of modules providing roundtrip engineering is not supported.

Requirements Aspect-Oriented Modeling (RAM) [23] has been proposed to identify and tackle cross-cutting concerns/NFR's at the requirements level. This approach utilizes Aspect Oriented Modeling (AOM), to identify and trace concerns. This modeling framework first identifies

stakeholders and their associated FR's, NFR's and groups the components exhibiting similar properties. Then the cross cutting concerns are identified which cross-cut the stakeholders view and these are marked as the requirements aspect. These aspects are then associated with generic solutions based on applied domain.

This approach identifies all requirements of each of the stakeholders and then based on associations amongst them, identifies the cross-cutting concerns. To analyze how these concerns may affect the system behavior, static and dynamic models of the requirements using class and sequence diagrams are utilized. The authors provide system architects with methodology and a framework to identify, analyze and address cross-cutting concerns.

3.4 NFR's using Component Containers

Component based applications require run time support to guarantee non-functional properties [12]. In their research, the authors propose a real-time-capable and component based architecture which would allow separation of functional and non-functional concerns during software development process. This container architecture provides a run-time environment which is NFR capable (E.g. QoS) and a component based software application.

One of the key features of this model is that more than one implementation can be provided for a functional specification which allows the same functionality with a different non-functional property thus separating the functional and non-functional concerns. The architecture is split into two components - real time and non-real time. The authors make use of *JBoss* which is an advanced component technology on the non-real time side of the architecture while *Dresden Real-time OPerating System (DROPS) [8]* is used for the real-time side. The container uses information quality descriptors to instantiate and implement components to assure non-functional properties desired by the client are guaranteed.

In another approach towards container-based development [24], authors present a model of component containers based on *Service Facilities (Serfs)*. This approach provides separation of concerns in an earlier stage enabling implementations made a part of the architecture itself.

The research identifies some of the potential approaches for overcoming modularization weaknesses in developing component-based architectures. One of them, *Component Containers* provide a model of components which support clear separation of concerns between the core functionality and the implied functionality (NFR's). The designed component only deals with the core functionality while the peripheral services are encapsulated within the container. Even though the available implementation options show improvement in modularization, their weakness lies in enabling tractable reasoning.

The *Serf* approach consists of components of design patterns such as Abstract Factory, Proxy etc. These create parameterized components in languages that do not support generic programming with first-class language constructs. The authors provide the design of a tool which is used to automatically generate *Serf* wrappers which directly support binding of services to components.

3.5 Automation Tools

Due to the fairly recent entry of Aspect-Oriented Programming (AOP) and its limited use in the commercial arena, not much emphasis has been put on automation tools and thus there are very limited tools which provide automation of code generations for AOP. One such tool, Meta Aspect J (MAJ) [27] is a language tool for generating AspectJ programs using code templates. MAJ is a structured meta-programming tool which generates syntactically correct programs.

Meta-programming is essentially programming to generate other programs. MAJ extends Java with support for generating AspectJ programs. It provides a convenient syntax representing the syntactic structure of the generated program. The property of structured tools to generate syntactically correct programs allows the user to be more confident in the output than when using a lexer based tool. MAJ offers variants of code-template operators such as "[..]", #[EXPR]. This tool is based on a context-free grammar for describing AspectJ syntax. There is use of two parsers and a common lexer which recognizes tokens legal in both the meta-language (Java) as well as the object language (AspectJ).

Our approach of using AOP to implement NFR's for any given system provides a solution

for existing systems and is unique such that the generated code can be integrated with the actual system without any dependencies. Another advantage of our tool is that extensive knowledge or programming effort is not required when using this tool to auto-generate the aspect. Besides, this java tool also produces an XML which can be used in extending not just the functionality of the tool but also allowing providing other development environments access to this information.

CHAPTER IV

UNDERSTANDING REQUIREMENTS

4.1 Functional and Non-Functional Requirements

Functional Requirements (FR's) are specific and define the expected behavior of the system. These requirements must be completed in order for the system being developed to be acceptable by the stakeholders. The FR's are easy to verify due to the tangible specifications defined for their acceptance and can be done in various ways such as Acceptance-Testing, Code verification etc

In contrast to this, Non-functional requirements (NFR's) are not defined specifically but are

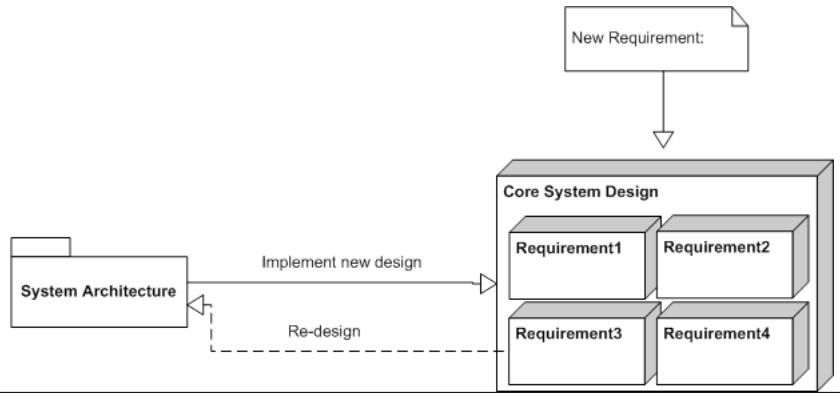

Figure 8: Implementation of new requirements using Object-Oriented Programming involves analyzing and redesigning the system architecture.

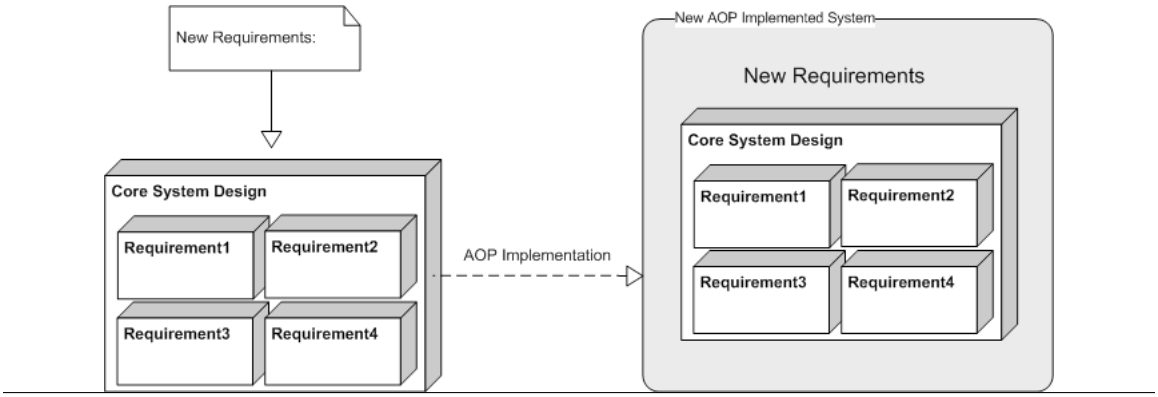

Figure 9: In contrast to OOP, implementing new requirements using AOP does not necessitate a change in the architecture since the requirements are weaved with the core system.

inherited based on the FR's of a system. NFR's can also be identified as characteristics of the system which enhance/supplement the behavior of the system being designed without affecting the system functional requirements. Figure 8 shows how addition of a new requirement to an existing system can be an expensive process. To accommodate the new requirement, the architecture of the system needs to be revisited as well as redesigned. Implementation of the new architecture will then involve more time and labor. The same situation when analyzed using Aspect Oriented Programming (AOP) methodology, would be handled differently as seen in Figure 9. Using AOP, the new requirements are implemented externally and no changes are required to either to core-module or the original system architecture.

Non-functional requirements consist of *constraints* and *qualities*. While *constraints* are restrictions which are considered the lowest priority during design-tradeoffs, *qualities* are characteristics that are desirable by the stakeholders and can influence their satisfaction with the system.

Qualities can be classified as *Run-time Qualities (RTQ's)* and *Development-Time Qualities (DTQ's)* [21]. As the name suggests, RTQ's are characteristics of the system exhibited to the end-user of the system. These qualities can vary from a system-level behavior to user-specific behavior. Some of the RTQ's consist of usability, availability, performance of the system etc. On the other hand, DTQ's are more of back-end requirements. These requirements may not influence the end-user but impact the developing organization. Some of these requirements include modifiability, support for future technologies and legacy systems, localization. Figure 10 provides a visual

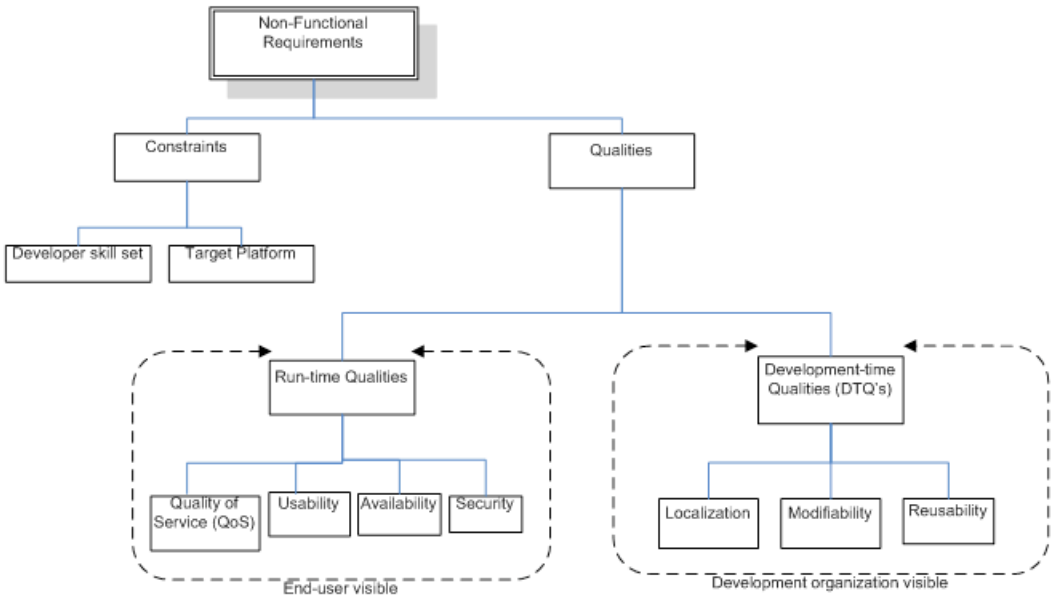

Figure 10: Non-Functional Requirements can be broken down into *constraints* and *qualities*. Our tool focuses on implementing Run-time qualities.

categorization of requirements classified as *Constraints* and *Qualities*.

4.2 Challenges with NFR's

One of the main reasons why NFR's are difficult to implement is because they do not map to tangible requirements and are often based on a set of assumptions. When the requirements can not be identified in a quantitative or measurable way it becomes difficult to decide when the requirement has been fulfilled. Another facet of this problem is that implementation may vary based on the perception of the developer. For example, a NFR like "System should log method calls" could be interpreted as a requirement to log all system calls or a requirement to log public method calls within the system. Figure 11 demonstrates the non-tangible behavior of NFR's in contrast to FR's.

The other issue in dealing with NFR's is the fact that they are difficult to test. This issue is highly coupled with the previous issue of interpreting these requirements. Based on the requirements, there are no set criterion which when tested would satisfy the system goal of the requirement being implemented correctly and completely. When a development goal can not be quantified it becomes difficult to implement and test it.

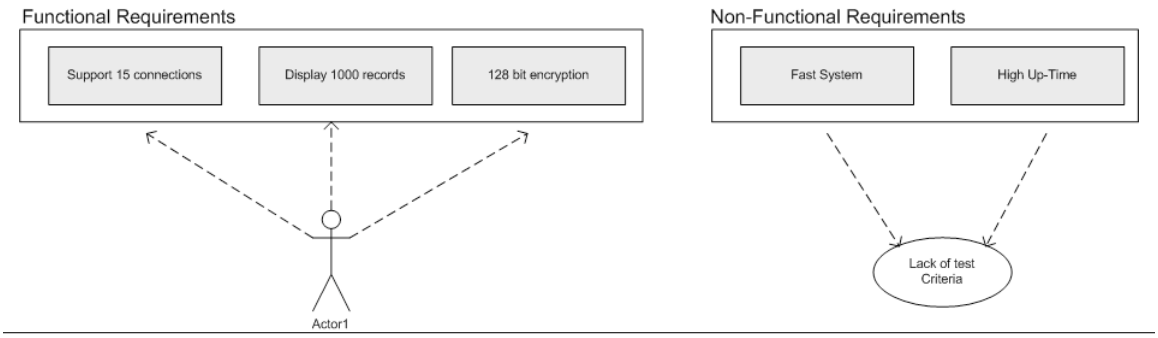

Figure 11: Functional Requirements can be tested easily using specified criteria, unlike Non-Functional Requirements.

Lastly, like any new requirement, there is risk in implementation of NFR's which can cause unwanted side-effects such as a trade-off in performance vs availability. One such scenario can be seen in Figure 12 in which, with increasing complexity of the system, the performance starts to decrease in order to maintain scalability. These kind of potential risks are hard to identify until system completion due to the nature of NFR's and once the NFR's have been implemented in a system, they get meshed with the core system so well that it becomes difficult to isolate the problem. Tackling such issues can be expensive in terms of system development and deployment time and is another reason that NFR's are not given a high priority.

NFR's, which are comprised of several quality attributes, exhibit a property such that these attributes function in the same generic way regardless of the system they are being applied to. For example, a "Logging" quality will provide functionality of capturing all information at various points in a program and making that information available in one form or another.

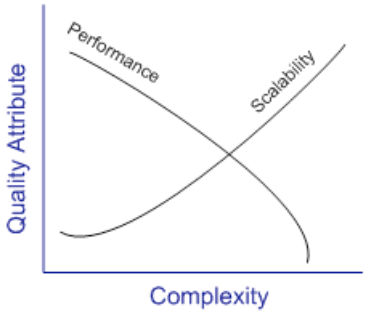

Figure 12: Increasing system complexity can lead to a potential tradeoff between quality attributes such as scalability and performance.

4.3 Our Approach

Currently there are not a lot of tools addressing the need for implementing these NFR's using a generalized approach. The use of Aspect Oriented Programming(AOP) is particularly of interest whilst considering implementation of NFR's. This is primarily because AOP permits enhancing the behavior of a system without modifying any code internally and creating an outer wrapper for additional functionality.

Our tool exploits this property of AOP to address the need for an automated tool which would generate code for any generic system based on user input. One of the key advantages of this approach is that the core system is not affected by the implementation of the NFR's as these are implemented using AOP. As discussed earlier, AOP permits adding functionality to a developed system without modifying the core components. Not only does this approach prevent increasing the complexity of the system, it also minimizes the risks involved in including these quality attributes. This is mainly because the quality attributes act like plug-ins to the system and can be taken out of the system for troubleshooting with much ease.

Our approach creates a balance between automation and customization. Even though quality attributes have specific behavior and nature, they can not be generalized in a single pattern for all systems. With an increase in generalization of these attributes, their usefulness decreases proportionately. Due to this, our approach incorporates not just creating a back-end logic for these attributes but also giving control to the user to customize the implementation and generated code to best suit the need of their system. This approach permits us to add a behavior to a system without having knowledge of its code and handing over control to the user to best-fit the behavior for their system making this tool flexible and easily deployable.

The goal of this tool is not just to automatically create code for a system but to provide a framework which can be used in various ways to extend the capabilities of the NFR's in different environments.

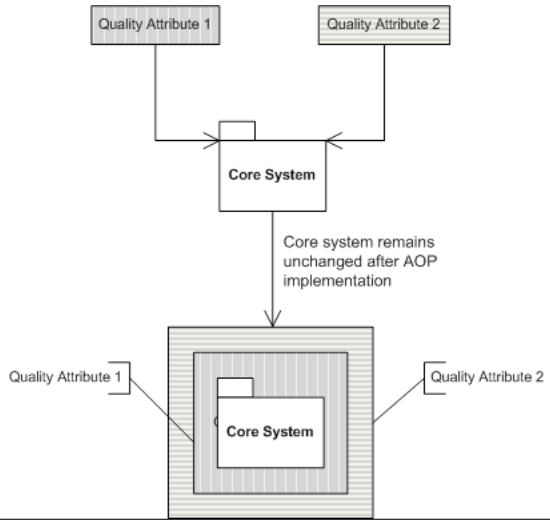

Figure 13: Aspects exhibit a plug-and-play behavior in terms of their deployment. They can simply be added or removed from the source directory to add/ remove their behavior after recompiling.

4.4 Design decisions

4.4.1 AOP

Using AOP decreases the risk involved in system deployment since the code written for implementing the Functional Requirements is never modified. Due to the nature of AOP methodology, instead of modifying the core code, the behavior is implemented externally greatly reducing any risk of modifying the actual system behavior

Another reason for selecting AOP approach was the plug-and-play characteristic of aspects. Figure 13 shows an example of adding quality attributes to a system which leaves the core system untouched and implements the attributes from outside. To implement any of these quality attributes, the actual system just needs to be compiled along with the generated source code and to remove these attributes, the system can be compiled after removing the source file for the particular attribute. This makes the quality attributes easy to deploy.

As discussed earlier, AOP prevents code tangling and it makes isolation of any issues easier due to their externally modifying behavior. Aspects can as easily be unloaded/removed from a system to help troubleshoot any potential issues while retaining the others. There is also availability

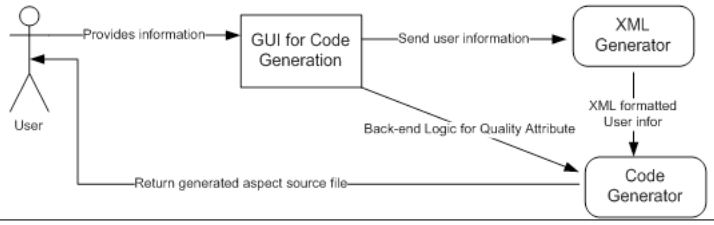

Figure 14: Our tool is designed based on three core functionalities, GUI, XML Generation and Aspect Generation. For a high level overview, the interactions between these modules are shown here.

of open source IDE (Integrated Development Environment) tools which support aspect weaving. It was decided that we would use these tools (Eclipse, NetBeans) for development since they best suited our requirement of developing aspects in AspectJ and also providing support for Graphical User Interface(GUI) development.

4.4.2 Automated NFR Code Generation

The tool simplifies the process of generating code for implementing NFR's in a system. Writing this code independently would consume a lot of time and will be catered to that instance of the system. If there were any changes made to the system, the code would have to be rewritten each time to support the new system. This tool makes it easy to automatically generate the code for quality attributes.

Figure 14 shows the complete sequence of steps that are followed to generate code for the quality attribute implementation aspect. The tool takes a two prong approach in generating the aspect code. The first step required is to obtain information from the user for gaining knowledge of the system and provide the options for customization. In the next step, the tool then weaves this information with the generic back-end logic of the specific quality attribute to generate a complete aspect code for implementing this quality attribute in the user defined system. It is important to have flexibility to mesh the user information with the back-end logic since this setup also provides room for extending this tool to various environments and platforms. If the back-end logic was made too rigid, then it would provide very limited functionality in the system.

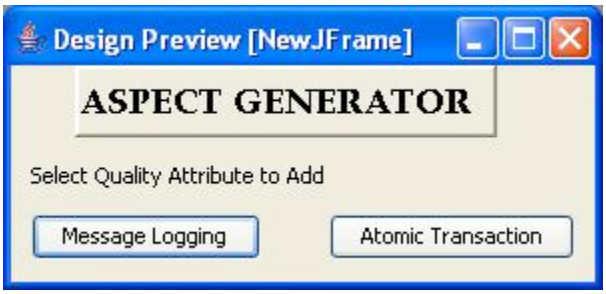

Figure 15: When starting the tool, the user must first select the quality attribute which they want to implement.

4.4.3 GUI for User Input

The GUI was developed in Java using NetBeans. The goal of the GUI is to obtain the required information from the user which is then meshed with the back-end logic to generate the aspect. We decided to give the user as much customization options as feasible so they can maximize the use of the tool and cater the generated code to their needs. The user is given an option (Figure 15) to select which Quality Attribute they would like to add to their system bringing up a different GUI screen based on the selected attribute.

4.4.4 Coding

To make this tool widely usable, the option to include custom code was added. This option was included so the user could write their own custom code besides the generated aspect and it gets included in the generated code. Having the option for the user to enter the custom code gives them more freedom in specifying the information they would want to include in the quality attribute. Since this tools aims on generating code for any system, it does not have sufficient knowledge to specify each and every attribute that could be customized in a system. Thus having this option gives the user the flexibility of adding more information besides what is provided by the tool.

Java was used in this project for researching aspect generation as well as coding the tool. Java is an Object Oriented programming language making it one of the potential choices besides C++. Besides this, there are extensions available in Java (AspectJ [1]) which are supported by open source tools such as Eclipse. This was specially important since the IDE contained an AspectJ

weaver which would compile the aspect code with the core program and give immediate output.

The first step towards creating this tool involved understanding how these quality attributes work and implementing actual aspects on some systems. Having researched aspect generation for the system, it was easier to identify the modules that would go into making this tool. Primarily the tool has three core modules. These are the *GUI*, *XML Generator* and *Aspect Generator*.

The *GUI* has been developed using Java NetBeans which provided a comprehensive view of the interface and ease-of-use for adding custom functionality in it. The *Aspect Generator* module has been coded purely in Java. For generating and parsing the XML structure, *Java Document Object Model (JDOM)* object was utilized. JDOM is an open source java based document object model for XML designed for the Java platform and integrates *Document Object Model (DOM)* as well as *Simple API for XML (SAX)*. More details regarding its implementation in our tool is discussed in Chapter VII.

CHAPTER V

HIGH LEVEL OVERVIEW OF THE TOOL

5.1 Functioning of the Tool

This section provides a high level overview of how the tool collects the required information and generates the code for the quality attribute. The sequence diagram shown in Figure 16 shows the interaction between the user and the Graphical User Interface (GUI) as well as the other steps that take place in the back-end of the tool.

For creating a tool which would generate aspects for specific quality attributes it was essential to understand the behavior and characteristics of each of the implemented attributes. The approach taken to develop this tool started with reverse engineering an implementation of one of the quality attributes designed for a specific system. The aspect coded at this stage provided the superset of information that would be required for generating this code. It was important to separate the concerns at this stage since this information included a mix of system specific as well as quality attribute specific characteristics.

The goal at this point was to extract the quality attribute specifics which would provide the back end of the system. At the same time it was equally important to identify the system-specific information that was required to be generalized for any system and obtained from the user using a

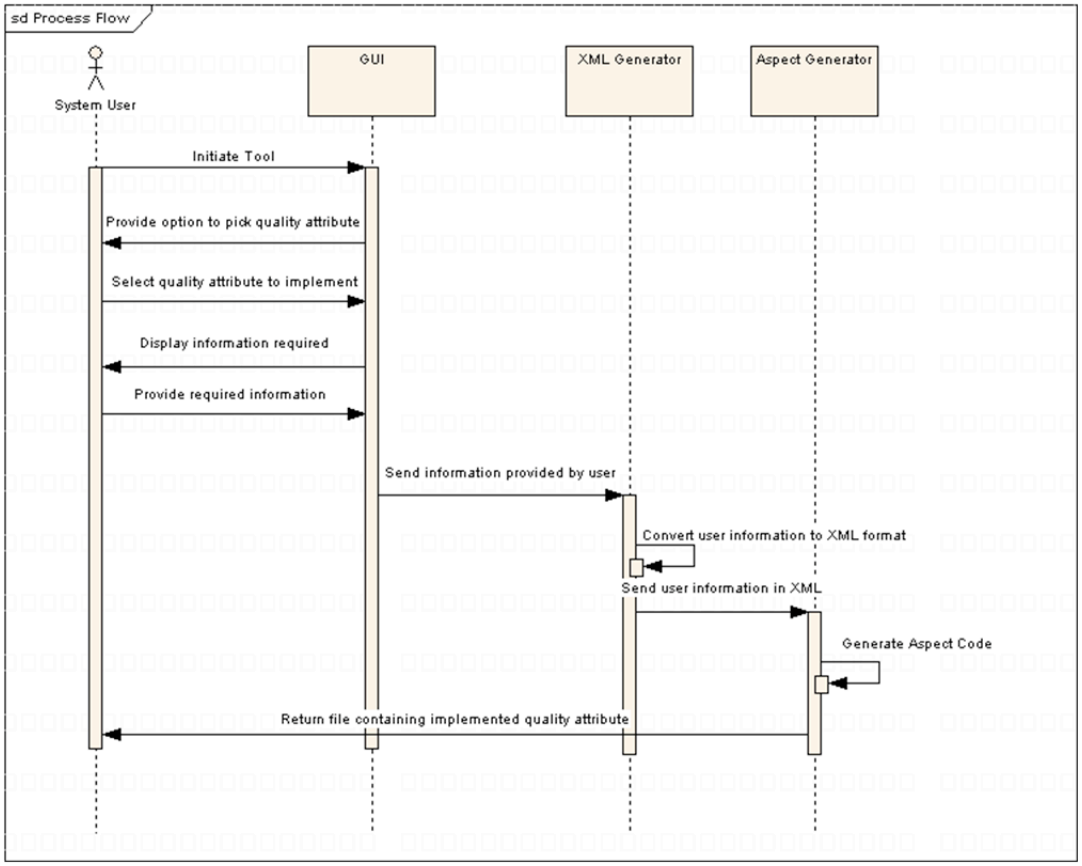

Figure 16: Sequence Diagram for this tool which starts with user providing information and ends with a generated aspect code returned to the user.

GUI on the front-end. This stage not only helped identify the back-end logic that would be required but also helped mould the GUI interface since required information was also identified here.

Once these sets of information were extracted and separated, it was essential to modularize data representation for the data obtained from the GUI. For this purpose we decided to use XML to organize the user input which would essentially be merged with the back-end logic to generate the aspect.

5.2 Data Representation

While selecting a data structure to represent information being obtained from the user, characteristics such as portability and extendibility were considered important. An XML structure provided both of these features and more.

As discussed earlier, XML data structures are highly portable and have great flexibility to add/remove information. Being highly portable ensures that the data structure generated by the tool can be reused on other platforms for not just generating quality attributes for systems but for extending the work as well. Also, XML is not restricted to any proprietary software base and further enhances the user experience due to its accessibility to open-source platforms.

XML also provides hierarchy for data structures which is a very useful characteristic specially when dealing with large data sets. XML also provides scalability since new data formats can be easily added to the structure without breaking backward compatibility.

As part of the research, it was extremely important to represent user-fed information in a logical manner. Having gone through the initial process of identifying the required user information, the next step involved generating the actual structure for this information. This is discussed in further detail in Chapter VI.

5.3 Quality Attributes implemented

5.3.1 Message Logging

Logging is a technique that is used to analyze and understand information. In its most basic form, logging involves capturing various states of the code during its execution much like a debugger. Logging is usually a more reasonable choice for debugging distributed systems [18].

Current implementations of logging involve adding logging statements throughout the operation's core logic making this a cross cutting concern. Furthermore, any changes to the logging necessitate change in associated modules resulting in increased effort and turn-around time.

The logging service adds to the behavior of target methods by writing details of their calls but does not cause any change in their behavior [25]. In their handling of the logger, *ILog* captures information from the caller of the method to target. Method values are then added to the *ILog* and in our case is done via Reflections (discussed in Section 7.1.5). *WriteReturnValues* method logs method values obtained from *ILog*.

Using AOP we can modularize the logging process without affecting the core logic. It also

helps create a centralized control such that changes can easily be accommodated from a single module instead of having to go through several different modules of the code.

5.3.2 Atomic Transaction

A transaction defines a unit of work that ensures the system remains in a consistent state before and after its execution. There are four properties of a transaction, namely - Atomicity, Consistency, Isolation, and Durability (ACID) [19].

Transaction management is also a cross-cutting concern and the operations under transaction control span across multiple modules. Transactions are essential for ensuring data integrity and a classic example of that is a transferring money from one bank account to another. Both, the *debit* and *credit* actions need to be executed systematically in a transaction to prevent any inconsistencies. These transactions have been are also referred to as *deferred executions* [25].

Transactions are broken into three components. First, a mapping of transaction requirement for each method, identifying the dependencies for transaction support. Secondly, a string of methods that are associated with the current transaction. The third component is a trace which collects a method call on execution, not when it is invoked. This component ensures that the methods in the trace are executed and as a result, methods associated with a transaction are immediately added to this trace when the transaction is committed.

Transaction management supports two functionalities in this domain, *commit* and *rollback*. The *commit* method is called during a live transaction and the method commits the transaction, as a result, all the associated methods in the trace are executed. *Rollback* on the other hand is called when a transaction fails or is not committed and essentially throws away the methods in the trace.

In our approach to transaction management, we utilize Atomicity which ensures that either all operations are successful within a transaction, or if there is a failure in a part of the transaction then the entire transaction is aborted and rolled-back.

CHAPTER VI

XML DESIGN AND IMPLEMENTATION

Part of the effort in this research was to identify a semantic structure which can be used to represent the information in a modularized manner so it can be interpreted and used efficiently. This section discusses the design and implementation of our approach for the XML structure.

6.1 Granularity of information

Three levels of granularity were considered initially - High, Medium and Low. Based on characteristics of each, Medium level was selected since it was the most appropriate solution for our purpose. Following reasoning justified the use of Medium granularity:

1. *High*: High granularity meant information was broken down into the most atomic form and that implied a lot of information (some even redundant) would need to be processed making it inefficient and not easy to interpret. For example, a Bank Account at this level would contain information such as :

    - Account Holders First Name
    - Account Holders Last Name

- Account Holders City of Residence
- Account Holders State of Residence
- Account Balance
- Account Last Deposit

2. *Low*: This level of granularity was not useful for our tool since at this level the data structure loses its value as a data profiler. For example, a Bank Account at this level would contain information such as:

   - Account Record Type, Balance, Holder Personal Information

3. *Medium*: This level of granularity blends the *High* and *Low* levels to provide a structure which has the information presented in just enough details. For example, a Bank Account at this level would contain information such as:

   - Account Holder Full Name
   - Account Holder Address
   - Account Transactions

6.2 XML structure design

The XML structure which was defined based on each of the quality attributes which were being implemented. This is because for each of the attributes being implemented, the information being obtained from the user was different depending on the nature of the attribute.

Message Logging requires a high degree of customization since the behavior of logging is to capture information which could be done in several different ways. As a result it was important to give the user maximum flexibility in determining the kind of information that needed to be logged and the format of saving extracted information. Due to the volume of information that was required to be retrieved, the XML structure was designed so that user feedback could be organized without creating clutter

In contrast to *Message Logging*, *Atomic Transaction* required a much lesser degree of customization. The nature of *Atomic Transaction* is to capture a set of actions and make sure they all get executed or aborted as a group. In this attribute less emphasis is made on information organization and more on the functional behavior. Because of this, the XML structure was much simpler due to limited user feedback required to implement it.

6.3 XML implementation

Some of the options for generating and parsing XML files included a hard-coded File Reader/Writer for generating an XML and reading the same using String buffers. In terms of feasibility this was not an optimum approach due to overhead, inefficiency and lack of scalability.

We decided to use Java Document Object Model (DOM) for generating the XML structure as well as parsing it for code generation. DOM is a platform and language independent object model for representing Hypertext Markup Language (HTML) or XML formats [6]. Some of the advantages of using DOM included *Navigability* and *Scalability*. DOM supports navigation in any direction (e.g. parent and previous sibling). Along with this DOM makes the information easier to access and use. Also, DOM is highly scalable and does not require the overhead of hard coding new modules. This helps maintain consistency, data integrity and optimizes data insertion and retrieval

We decided to generate this XML file and retrieve information from the XML instead of using the objects internally within the code. The reason is that this provides room for expansion since the same XML file can be taken to a different environment and the semantics can be reused for generating code. Also, this XML file can be used in the same environment to drive code generation for a different aspect thus extending its usefulness.

CHAPTER VII

QUALITY ATTRIBUTES IMPLEMENTATION

7.1 Message Logging

7.1.1 Behavior

The main idea behind *Message Logging* is the ability to capture various attributes of a system at certain specified points during its execution. The attributes being captured can vary greatly depending on the purpose of the logging for the given system. For instance, in a banking system, it may be more important to log events such as account transactions while on a web server the focus of logging may be more on calls to a specific module which loads the web pages.

In our implementation we have taken one such instance wherein the user can create a log of methods based on specific properties of the method. For example, the user may want to log any method call made to a *private* method which returns a *string* data type. This may be to ensure integrity of the system or to track activity which may otherwise be restricted. For making this quality attribute useful for a variety of systems, the Graphical User Interface (GUI) provides a wide range of options where the user can customize logging of multiple scenarios.

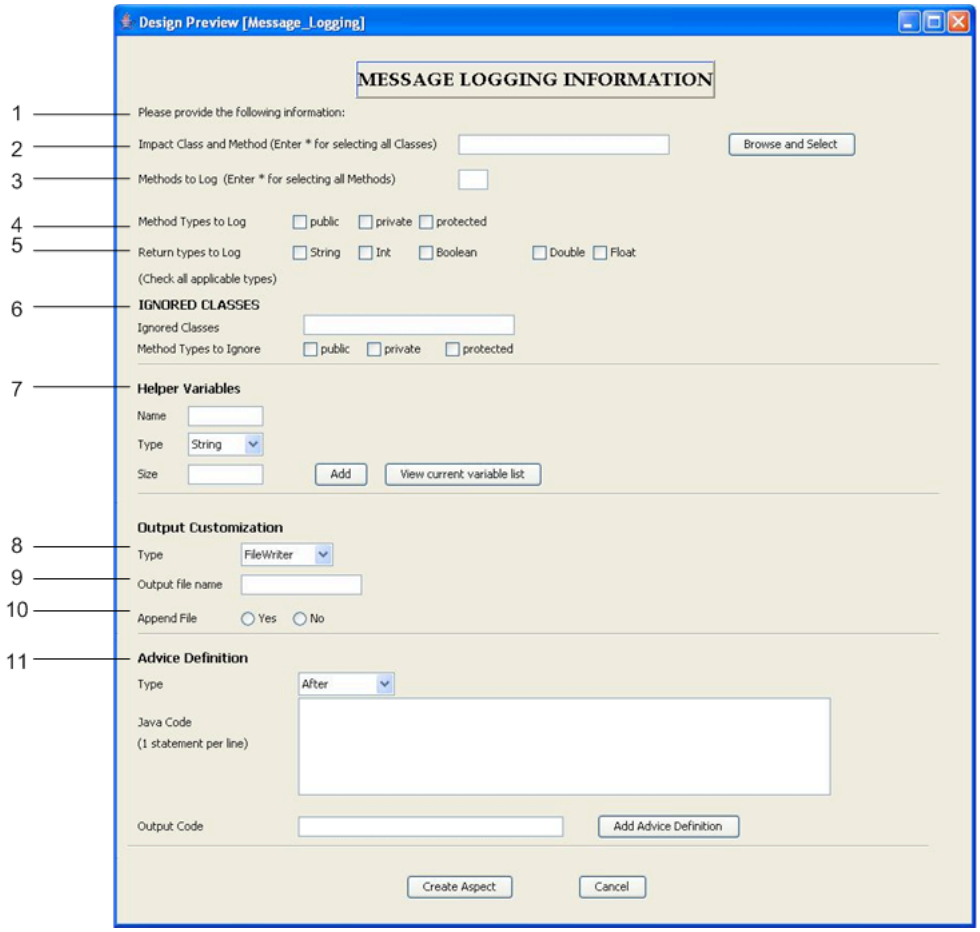

Figure 17: Message Logging requires detailed information and accordingly its GUI has been designed to provide high customization options to the user.

7.1.2 GUI design

The core piece of information required to create this aspect is the list of classes and methods which the user would like to log. In Figure 17, (1) is used to obtain class/method information from the user. For specifying this a class browser is included which allows the user to specify the classes and the methods that should be logged.

If the user has a more general need than specifying specific classes/methods, they can always select the generic options which permits them to log all methods of various types (such as Public, Private, Protected) implemented by (3) and (4) based on users need. The user can even specify if they want to capture methods which return a specific type of value using option (5) (such as String, Int, Boolean, Double, Float). In the process of creating a generic scenario like capturing all the

public methods, there may be instances where the user may want to exclude some specific methods. To cover such scenarios, the tool provides the option of specifying *ignored classes and methods* using (6). This option permits the user to specify classes and methods which will be excluded from the specified logging criteria. Like the earlier option of making this as generic as possible, the ignored options also give the user to exclude specific method types which they can ignore besides specifying some classes.

When logging the information specified by the user, there can be different ways in which the user may want to save the logged information. For this reason, this tool provides the option of specifying the output file type (for example PrintWriter, Console output and others available in Java) and the output file name using (8), (9). There is also an option of creating a new output log file for each instance or appending the previously generated file and this is available in (10). The tool has a utility for advanced users to add helper variables which can be used in custom code and permits the user to view the list of variables that have been generated to avoid potential conflicts when the program is compiled.

Lastly, the GUI also provides room for allowing the user to enter custom Java code which is automatically included in the generated aspect code as seen in (11). This can be very beneficial for power users who not only want to make use of auto-generated aspects but also like to have more control and provide their environment specific information. The tool reads in code written by the user and adds it to the Aspect Code. This is an optional feature and is not required to generate the aspect. It was built in to make this tool flexible and productive for advanced users.

7.1.3 Generated XML Structure

Listing 7.1: ”A complete XML file generated from a test run of the tool on a simple Banking System”

```
<?xml version="1.0" encoding="UTF-8"?>
  <QualityAttribute>
    <XMLName>
      <ImpactClass>
        <Name>AccountHandler</Name>
```

```
6         <Method>public void AccountHandler.work(BankAccount,BankAccount[],int)</Method>
7       </ImpactClass>
8       <ImpactClass>
9         <Name>BankAccount</Name>
10        <Method>public void AccountHandler.work(BankAccount,BankAccount[],int)</Method>
11        <Method>public java.lang.String BankAccount.toString()</Method>
12        <Method>public java.lang.String BankAccount.getID()</Method>
13      </ImpactClass>
14      <ReturnTypeLog>
15        <Type>String</Type>
16        <Type>Int</Type>
17      </ReturnTypeLog>
18      <Influence>
19        <HelperVariable>
20          <Name>ILog</Name>
21          <Type>String</Type>
22          <Size/>
23        </HelperVariable>
24        <Ignored>
25          <Name>myIgnoreClass1</Name>
26        </Ignored>
27        <Ignored>
28          <Name>myIgnoreClass2</Name>
29        </Ignored>
30        <Output>
31          <Type>PrintWriter</Type>
32          <FileName>myOutputFile.txt</FileName>
33          <Append>True</Append>
34        </Output>
35      </Influence>
36    </XMLName>
37  </QualityAttribute>
```

For each of the quality attributes implemented, the user customization information is organized in terms of an XML structure. For message logging, there is a high degree of user customization and requires a detailed XML structure.

The scenario used to illustrate application of this tool is a *Banking System* which has been

developed in Java. As mentioned earlier, the key component of this quality attribute are the classes and methods which need to be logged. This information is logged using the $< ImpactClass >, < Name >, < Method >, < Type >$ attributes as seen in lines (4) through (7) in Listing 7.1. The classes and methods which are ignored use similar tags with the exception that they are classified under *Ignored* category and are shown in lines (24) through (29). A separate XML tag of $< Output >$ is used to specify output file information such as $< Type >, < FileName >, < Append >$ This option lets the user select the format in which they want to save the log.

7.1.4 Generated Code

Based on user input, we can see the XML output which is generated. Correspondingly, the following listing demonstrates the aspect which is generated based on the input provided by the user. Listing 7.2 shows the generated aspect which the user receives from the tool. Since the tool generates this aspect based on various pointcuts, they need to be named dynamically depending on the input classes and methods. This makes the utility flexible and scalable instead of being restricted by a hard-coded variable set.

Looking at the generated XML in Listing 7.1, lines (5), (6) identify a specific method "Work" in the class "AccountHandler" which needs to be logged. Along with this, lines (10) through (12) identify other methods in the "BankAccount" class which are also required to be logged. In the event that the user wants to invert a selection, they have the option of ignoring some classes which may fall under one of the specified pointcuts. These are seen in lines (24) through (29) of the XML listing.

Lines (14) through (17) of the XML (Listing 7.1) represent *return types* which need to be logged, meaning any method which returns a *String* or *Int* value, will also need to be logged. The user can also customize the format of output log which the aspect will generate. This is seen in lines (30) through (34) where the output type is specified to be used as "PrintWriter" and the output log has been named "myOutputFile.txt". The *append* option in line (33) gives the user an option to start

a fresh log with each run or continue adding log to the same file.

Now looking at the generated aspect code (Listing 7.2), for each of the user specified method type, a pointcut has been generated which can be seen in lines (10), (42), (74). Each of these pointcuts are followed by the advice implementing them. One thing to notice in the pointcuts that have been defined is the use of *!execution* parameter. These parameters specify the classes which were selected by the user to be ignored and these get automatically appended to each of the pointcuts to ensure that they do not get logged.

Lines (138), (145) are pointcuts identifying methods which return specific type of variables such as *String (138)*, *Int (145)*. Also, a helper method is added to the code in line (153) which is essentially the *log writer*. This method also uses the customization details provided by the user in the XML for the output format.

This code does not have any dependency on any supporting files and can be compiled directly along with the example Banking System using an *Aspect weaver* to add the logging capability. Essentially, instead of having to go through the system design and identifying modules which would need to be updated to include this, the utility simplifies the task of adding a quality attribute to the system. Another big advantage of using the tool generated aspect is that any changes required to the logging operation are based on this single module. As a result, if at some point the user wants to extend these logging capabilities, they can just make changes to the aspect instead of going to each core module and updating it, thus solving the code scattering and tangling issues.

Listing 7.2: ”Complete AspectJ Code generated from a test run of the tool on a simple Banking System and based on the XML generated in Listing 7.1”

```
1  import org.aspectj.lang.reflect.*;
2  import java.io.*;
3  import org.aspectj.lang.*;
4
5
6  public aspect MessageLoggerAspect
7  {
8          String ILog = "";
9
10         pointcut methodToLog00(): execution(public void AccountHandler.work(BankAccount,
```

```
        BankAccount[],int)) && !execution(* myIgnoreClass1.*(..)) && !execution(*
        myIgnoreClass2.*(..));

    before(): methodToLog00()
    {
            Signature sigToLog = thisJoinPoint.getSignature();

            try {

             FileWriter outFile = new FileWriter("myOutputFile.txt",True);
             PrintWriter out = new PrintWriter(outFile);

             Class[] parameter_Type = ((CodeSignature)sigToLog).getParameterTypes();
             String[] parameter_name = ((CodeSignature)sigToLog).getParameterNames();
             if(thisJoinPoint.getSignature().getName() == null) ;

             else
             ILog = "\nEntering␣Method:␣" + thisJoinPoint.getSignature().getName() + "
                 ()\n";

            for(int j = 0; j<paramter_type.length; j++)
            {
                    methodArguments[j] = ("Method␣Parameter␣" + "[" + j + "]␣=␣"+ "␣
                        Type:␣" + (String)parameter_type[j].getSimpleName() +
                        ", Value: " + parameter_name[j] + "\n");
                    out.write(methodArguments[j]);
            }
            out.close();
            }

            catch(IOException ioe)
            {

            }
    }

    pointcut methodToLog10(): execution(public void AccountHandler.work(BankAccount,
        BankAccount[],int)) && !execution(* myIgnoreClass1.*(..)) && !execution(*
        myIgnoreClass2.*(..));

```

```
44          before(): methodToLog10()
45          {
46                  Signature sigToLog = thisJoinPoint.getSignature();
47
48                  try {
49
50                   FileWriter outFile = new FileWriter("myOutputFile.txt",True);
51                   PrintWriter out = new PrintWriter(outFile);
52
53                   Class[] parameter_Type = ((CodeSignature)sigToLog).getParameterTypes();
54                   String[] parameter_name = ((CodeSignature)sigToLog).getParameterNames();
55                   if(thisJoinPoint.getSignature().getName() == null) ;
56
57                   else
58                   ILog = "\nEntering␣Method:␣" + thisJoinPoint.getSignature().getName() + "
                         ()\n";
59
60                  for(int j = 0; j<paramter_type.length; j++)
61                  {
62                          methodArguments[j] = ("Method␣Parameter␣" + "[" + j + "]␣=␣"+ "␣
                                Type:␣" + (String)parameter_type[j].getSimpleName() +
                                ", Value: " + parameter_name[j] + "\n");
63                          out.write(methodArguments[j]);
64                  }
65                  out.close();
66                  }
67
68                  catch(IOException ioe)
69                  {
70
71                  }
72          }
73
74          pointcut methodToLog11(): execution(public java.lang.String BankAccount.toString()
                ) && !execution(* myIgnoreClass1.*(..)) && !execution(* myIgnoreClass2.*(..));
75
76          before(): methodToLog11()
77          {
78                  Signature sigToLog = thisJoinPoint.getSignature();
79
```

```
80                  try {
81
82                   FileWriter outFile = new FileWriter("myOutputFile.txt",True);
83                   PrintWriter out = new PrintWriter(outFile);
84
85                   Class[] parameter_Type = ((CodeSignature)sigToLog).getParameterTypes();
86                   String[] parameter_name = ((CodeSignature)sigToLog).getParameterNames();
87                   if(thisJoinPoint.getSignature().getName() == null) ;
88
89                   else
90                   ILog = "\nEntering␣Method:␣" + thisJoinPoint.getSignature().getName() + "
                         ()\n";
91
92                  for(int j = 0; j<paramter_type.length; j++)
93                  {
94                          methodArguments[j] = ("Method␣Parameter␣" + "[" + j + "]␣=␣"+ "␣
                                Type:␣" + (String)parameter_type[j].getSimpleName() +
                                ", Value: " + parameter_name[j] + "\n");
95                          out.write(methodArguments[j]);
96                  }
97                  out.close();
98                  }
99
100                 catch(IOException ioe)
101                 {
102
103                 }
104         }
105
106         pointcut methodToLog12(): execution(public java.lang.String BankAccount.getID())
                && !execution(* myIgnoreClass1.*(..)) && !execution(* myIgnoreClass2.*(..));
107
108         before(): methodToLog12()
109         {
110                 Signature sigToLog = thisJoinPoint.getSignature();
111
112                 try {
113
114                  FileWriter outFile = new FileWriter("myOutputFile.txt",True);
115                  PrintWriter out = new PrintWriter(outFile);
```

```
116
117                 Class[] parameter_Type = ((CodeSignature)sigToLog).getParameterTypes();
118                 String[] parameter_name = ((CodeSignature)sigToLog).getParameterNames();
119                 if(thisJoinPoint.getSignature().getName() == null) ;
120
121                 else
122                 ILog = "\nEntering␣Method:␣" + thisJoinPoint.getSignature().getName() + "
                        ()\n";
123
124                for(int j = 0; j<paramter_type.length; j++)
125                {
126                        methodArguments[j] = ("Method␣Parameter␣" + "[" + j + "]␣=␣"+ "␣
                            Type:␣" + (String)parameter_type[j].getSimpleName() +
                            ", Value: " + parameter_name[j] + "\n");
127                        out.write(methodArguments[j]);
128                }
129                out.close();
130                }
131
132                catch(IOException ioe)
133                {
134
135                }
136         }
137
138         after() returning(string methodReturnValue): execution(public string *..*(..)) &&
                !execution(* myIgnoreClass1.*(..)) && !execution(* myIgnoreClass2.*(..))
139         {
140                 Signature mySig = thisJoinPoint.getSignature();
141                 String toWrite = "Returning␣from:␣" + mySig.getDeclaringTypeName() + "."+
                        mySig.getName()+"()␣with␣value␣" + methodReturnValue + "\n";
142                 WriteReturnValues(toWrite);
143         }
144
145         after() returning(int methodReturnValue): execution(public int *..*(..)) && !
                execution(* myIgnoreClass1.*(..)) && !execution(* myIgnoreClass2.*(..))
146         {
147                 Signature mySig = thisJoinPoint.getSignature();
148                 String toWrite = "Returning␣from:␣" + mySig.getDeclaringTypeName() + "."+
                        mySig.getName()+"()␣with␣value␣" + methodReturnValue + "\n";
```

```
                WriteReturnValues(toWrite);
        }


public void WriteReturnValues(String x)
{
        try {
                FileWriter outFile = new FileWriter("myOutputFile.txt",True);
                 PrintWriter out = new PrintWriter(outFile);
                 out.write(x);
                 out.close()}
        catch(IOException ioe)
                {
                }
}
}
```

7.1.5 Generated Log

A major part of implementation of the logging mechanism has been created using *Java Reflection*. Reflection is a powerful method to examine or modify runtime behavior of a system [4]. Reflection is used to create instances of objects during system execution using their fully-qualified names to ascertain their characteristics. This tool utilizes this API to obtain information such as *method name*, *method arguments*, *argument values*.

Based on the specification that was provided by the user, the generated log contains method information for target methods such as their arguments, values, call and execution in the system as seen in Listing 7.3.

Listing 7.3: ”Sample log which is generated by the Message Logging quality attribute based on information provided by the user”

```
Entering Method: authenticateUser()
Method Parameter [0] =  Type: BankAccount[], Value: account
Method Parameter [1] =  Type: AccountHandler, Value: my_AccountHandler
Method Parameter [2] =  Type: String, Value: idCheck
Method Parameter [3] =  Type: int, Value: pinCheck
Entering Method: displayTransac()
Returning from: BankAccount.displayTransac() with value Transactions
Thu Apr 19 20:35:53 CDT 2007  MONEY WITHDRAWN :   $1.0 Balance $99.0
```

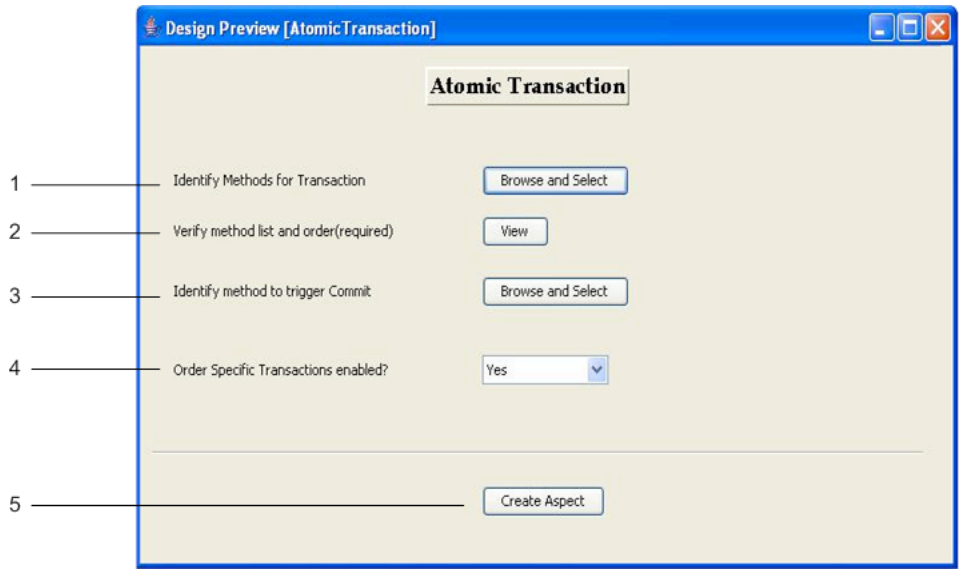

Figure 18: Atomic Transaction GUI which permits the user to specify the trigger method and related transaction methods.

7.2 Atomic Transaction

7.2.1 Behavior

Amongst the set of properties which ensure that a transaction in a system has been processed reliably is the property of *Atomicity*. An Atomic Transaction is defined as a series of operations which either *all* occur or *all* do not occur. This is to ensure that there are not any partial updates which can lead to bigger synchronous issues than rejecting the entire series.

There can be several ways of implementing this property in a system such as order-specific or order-independent. When a set of operations that have been identified which need to occur, the user may want to ensure that they occur in a particular order or may want it to be order-independent.

7.2.2 GUI design

This quality attribute required limited information from the user since the goal is to ensure that a set of specified operations are completed or rolled back together and does not require a high degree of customization. In Figure 18, the primary information obtained using the GUI is the list of methods (1) which need to be executed before a *trigger* method (3) is permitted to execute. If any

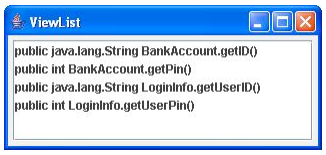

Figure 19: The methods associated with a transaction can be reordered in this GUI to allow the user to implement order-specific Atomic Transaction.

one of the specified methods do not get executed, the trigger method is not allowed to be executed either.

To make this quality attribute more flexible, the user is provided a way to view the methods selected (using (2)) and reorder them to modify the transaction order. This option (as seen in Figure 19) is a simple drag-and-drop interface to move the modules within the window. Changes made in the window are automatically absorbed in the generated code.

The GUI also provides the user the option to specify if the order of execution of these methods is required or not (4). The user can then generate the aspect by clicking the "Create Aspect" button (5)

7.2.3 Generated XML Structure

In Atomic Transactions, the level of user customization is low and as a result, the XML data structure to represent this information is much simpler. Listing 7.4 shows a simple XML generated for implementing this attribute.

The primary tags used here are the $< TransactionMethod >$, $< TriggerMethod >$ and $< OrderEnabled >$. *TransactionMethod* refers to the set of operations which will be need to be executed before the *TriggerMethod* is permitted to run. The importance of order of execution of these operations is identified by the *OrderEnabled* tag.

7.2.4 Generated Code

Based on the XML structure above, the aspect is generated (Listing 7.5). Following the same principle regarding dynamic creation of variables (as Message Logging attribute), variables are generated based on selected methods instead of being hard-coded. The code captures calls

Listing 7.4: ”Sample XML generated for Atomic Transaction quality attribute showing method information captured from the user”

```
      <TransactionMethod>
        <Name>public java.lang.String BankAccount.toString()</Name>
        <Name>public java.lang.String BankAccount.getID()</Name>
      </TransactionMethod>
      <TriggerMethod>
        <Name>public static void  Test.main(java.lang.String[])</Name>
      </TriggerMethod>
      <OrderEnabled>No</OrderEnabled>
```

Listing 7.5: ”Based on the generated XML - code for Atomic Transaction is generated”

```
public aspect AtomicTransaction
{
        ArrayList stopMethod = new ArrayList();
        ArrayList compareWithMethod = new ArrayList();
        Boolean myFlag = false ;
    pointcut myPointCut0(): execution(public double BankAccount.getBalance());
        String around(): myPointCut0()
        {
                try {
                myObject0 = thisJoinPoint.getThis();
                stopMethod.add(myObject0);
                try {
                  synchronized(myObject0)
                  myObject0.wait();
                   }
                }
        catch(InterruptedException e) {}
        catch(IllegalMonitorStateException e) {}
        ...
```

to methods which are specified by the user and once the criteria has been fulfilled, allows their execution to continue.

CHAPTER VIII

LIMITATIONS, FUTURE WORK AND CONCLUSION

8.1 Limitations of the Tool

This tool was developed using Java and generated code in AspectJ. As a result, the syntax of this utility restricts it to be used in a Java environment. The generated code is written in AspectJ and requires an aspect weaver for this language to weave it with the Java source code. However this does not limit the capabilities of the tool to be used in other languages due to the provision of XML file.

This tool is limited to the implementation of two quality attributes *Message Logging* and *Atomic Transaction*. To extend capabilities for users to be able to browse through source code and directly select methods using GUI (Graphical User Interface). Also, being a proof-of-concept implementation, this tool has not been subject to thorough testing for edge cases and regular GUI tests.

8.2 Future Work

There are two key areas for extending this work. First is the domain extension. This tool has been built in Java and functions best for systems have been developed in Java itself. Currently support is available for languages such as AspectC/AspectC++/AspectC# [3] and this would be one of the areas that the tool can be extended and utilized. Doing so will increase the viability of the design over different languages overcoming its current limitation of just Java.

The other key area for extension is Quality Attributes. Currently, the tool provides automation of code-generation for two quality attributes. There are several other quality attributes that can be researched such as *availability* and *performance*. The benefit of doing so is that it will increase the usefulness of the tool making it a more comprehensive one-stop utility giving the user more options for enhancing their system.

8.3 Conclusion

Much research has been focused on theoretical models for analyzing Non-Functional Requirements (NFR's) and linking Aspects with them at an earlier stage. Putting these models into practice involves a greater effort. We provide a solution for systems which have been developed and need to be enhanced with various quality attributes such as logging, atomic transactions. This tool addresses the need for automated generation of code for NFR's and provides a comprehensive solution for implementing these without modifying the core code. We also suggest a semantic model for structuring information required to implement these attributes which can be useful for extending this work as well.

BIBLIOGRAPHY

[1] Ajdt: Aspectj development tools. Website. http://www.eclipse.org/ajdt/.

[2] The aspect-oriented software architecture design portal. Website. http://trese.cs.utwente.nl/taosad/.

[3] The home of aspectc++. Website. http://aspectc.org/.

[4] Trail the reflection api. Website. http://java.sun.com/docs/books/tutorial/reflect/index.html/.

[5] Xml. Website. http://en.wikipedia.org/wiki/XML.

[6] Document object model (dom). Website, 2005. http://www.w3.org/DOM/.

[7] Extensible markup language (xml) 1.0 (fourth edition), w3c recommendation. Website, 2006. http://www.w3.org/TR/xml/.

[8] R. Baumgartl, M. Borriss, H. Härtig, C. Hamann, M. Hohmuth, L. Reuther, S. Schönberg, and J. Wolter. Dresden realtime operating system. In *Proceedings of the First Workshop on System Design Automation (SDA'98)*, pages 205–212, Dresden, Mar. 1998.

[9] L. Chung, B. A. Nixon, and E. Yu. Using quality requirements to systematically development quality software. In *Proc. 4th Int. Conf Software Quality*, 3–5 1994.

[10] S. Clarke and R. J. Walker. Separating crosscutting concerns across the lifecycle: From composition patterns to aspectJ and hyper/J. Technical Report TR-2001-05, 08 2001.

[11] C. Constantinides and T. Skotiniotis. Providing multidimensional decomposition in object-oriented analysis and design. In *Software Engineering*. Acta Press, 2004.

[12] S. Gbel, C. Pohl, R. Aigner, M. Pohlack, S. Rttger, and S. Zschaler. The comquad component container architecture and contract negotiation.

[13] I. Groher and S. Schulze. Generating aspect code from UML models.

[14] S. Herrmann. Composable designs with UFA. 2002. Workshop on Aspect-Oriented Modeling with UML at 1st Intl. Conference on Aspect Oriented Software Development.

[15] R. Kazman, P. Clements, and L. Bass. *Software Architecture in Practise*. Addison-Wesley, 2003.

[16] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[17] S. Kuninobu. Making programs reliable using aop. In *Fifth International Conference on Aspect-Oriented Software Development*, 2006.

[18] R. Laddad. Simplify your logging with aspectj. Website. `http://www.developer.com/java/other/article.php/3109831/`.

[19] R. Laddad. *AspectJ in Action, Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[20] R. Ladded. I want my aop part 1. Website, 2002. `http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html?page=3`.

[21] R. Malan and D. Bredemeyer. Defining non-functional requirements. Technical report, 08 2001.

[22] G. Pollice. A look at aspect-oriented programming. Website. `http://www.ibm.com/developerworks/rational/library/2782.html`.

[23] Y. R. Reddy. An aspect oriented approach to early software development. 2004. 7th International Conference on the Unified Modeling Language.

[24] N. Sridhar and J. Hallstrom. Generating configurable containers for component-based software. 2003.

[25] N. Sridhar and J. Hallstrom. A behavioral model for software containers. pages 139–154, 2006.

[26] Q. Tran and L. Chung. Nfr-assistant: Tool support for achieving quality.

[27] D. Zook, S. S. Huang, and Y. Smaragdakis. Generating aspectj programs with meta-aspectj. 2004.