

https://dspace.org/

A Software Reuse Paradigm for the Next Generation Network (NGN)

Jagot, Bilal Abdull Rahim

2006-11-17T11:19:19Z
http://hdl.handle.net/10539/1860

Downloaded from a DSpace repository. https://dspace.org/

A Software Reuse Paradigm for the Next Generation Network (NGN)

Bilal Abdull Rahim Jagot

A project report submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, in partial fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, August 2003

Declaration

I declare that this project report is my own, unaided work, except where otherwise acknowledged. It is being submitted for the degree of Master of Science in Engineering at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other university.

Signed this ___ day of _____ 20___

_____

Bilal Abdull Rahim Jagot.

Abstract

Service creation in the Next Generation Network (NGN) is focused around software creation and borrows heavily from the Software Engineering community. In the NGN, telecommunication companies demand simple, rapid and economical service creation. The key to this type of service creation is software re-use. Software re-use is a conundrum where limited, dedicated solutions exists. These solutions include amongst others Enterprise JavaBeans™ (EJBs), design patterns and object-oriented programming.

The Telecommunications Information Networking Architecture- Conformance And Testing (TINA-CAT) workgroup has done work on a functionality centric concept called RP-facets. This report proposes a redefinition of RP-facets, as *Facets*, for software re-use across the design and code level. We redefine *Facets* as functionality centric **reusable components**. A *Facet* is independent of the implementation language and the execution platform. *Facets* allow containment in a structured manner via a user defined Facet Hierarchy. *Facets* are resource, context and data agnostic. They also introduce a structured way to allow source code to be changed based on design level decisions. Also, possessing the ability to allow the simultaneous use of other reuse solutions and programming paradigms. Abstraction of detail from developers and platform migration can be achieved by using *Facets*.

Facets are composed of a Generic definition and any number of Implementation definitions. The definitions are supported by an underlying informational model called meta-π. Meta-π is a model at the M3 meta-level that focuses on describing entities. Most of the *Facet's* capabilities are enabled by the meta-π model.

An environment for developing *Facets* is created, called the Facet Development Environment (FDE). The Facet Developer (FD) role is introduced to develop and maintain *Facets*. The FD verifies programmes from programmers to be included into the catalogue of Facets via the FDE. The FD interacts with service creation teams to determine which *Facets* can be used in the service they wish to develop.

Facets prove their capability in targeted areas, yet lack in other categories. It is recommended that the underlying informational model should be revised to form a more robust and flexible entity describing model. In addition, a cataloging capability to easily find

Facets with particular functionality should be appended to the capabilities of the facet. It is proposed, for future work, that a development environment be created that encompasses a process for using *Facets* to create services.

Acknowledgements

This work was performed at the Centre for Telecommunications Access and Services (CeTAS) at the University of the Witwatersrand, Johannesburg. The centre is funded by Telkom SA Limited, Siemens Telecommunications and the Department of Trade and Industry's THRIP programme. The financial support was much appreciated.

I would like to extend my thanks to my supervisors, Prof. Hu Hanrahan and Dr. Setumo Mohapi for their guidance and assistance throughout the duration of the research project. In addition I would like to thank my colleagues at CeTAS for their criticism and valuable inputs during the research project particularly Aydin Alaylioglu. But most importantly I would like to thank my parents for their love, support and patience. It is because of them that I am who and where I am today. This work is dedicated to my mother, Farida A. Esak and my late Father Abdull R. Esak.

Contents

Declaration i

Abstract ii

Acknowledgements iv

Contents v

List of Figures x

List of Tables xiii

Acronyms xiv

Definition of Terms xvi

Set Notation Symbols xvii

1 Introduction 1

1.1 Service Creation in the NGN 2

1.2 Attempts at Software Reuse 3

1.3 Classification and Analysis of Reuse Solutions 5

1.4 Problem Objectives 6

1.5 Outline of Report 7

2 Contemporary Reuse Practices 8

2.1 Reuse in Telecommunications . . . 9
2.1.1 Intelligent Networks . . . 10
2.1.2 Telecommunications Information Networking Architecture (TINA) 11
2.1.3 Reference Model of Open Distributed Processing (RM-ODP) . . . 12
2.1.4 OSA/Parlay . . . 12
2.1.5 XML Web Services . . . 13
2.2 Reuse in the Computing Arena . . . 13
2.2.1 Model Driven Architecture (MDA) . . . 13
2.2.2 Enterprise Java Beans™ . . . 15
2.2.3 Design Patterns . . . 16
2.3 Chapter Summary . . . 17

3 Introduction to Facets **19**

3.1 TINA Reference Point-Facet . . . 19
3.1.1 Reference Point Facet Definition . . . 20
3.2 Facets Redefined . . . 21
3.2.1 Mathematical Definition of Facets . . . 22
3.3 Business Viewpoint . . . 23
3.4 Informational Viewpoint . . . 24
3.5 Chapter Summary . . . 26

4 Key Enabling Concepts **27**

4.1 Generic and Implementation Definitions . . . 27
4.2 Meta-π . . . 29
4.2.1 Definition . . . 31
4.2.2 Context . . . 33
4.2.3 Resource . . . 33

4.2.4 Data . . . 33
4.3 Placeholders . . . 34
4.4 Variable Blocks . . . 34
4.5 Facet Hierarchy . . . 35
4.6 Chapter Summary . . . 36

5 Facet Development Environment 37

5.1 Design Considerations . . . 37
5.1.1 Informational Modelling Languages . . . 37
5.1.2 Parsers . . . 38
5.1.3 Implementation Language . . . 38
5.1.4 Information Storage . . . 39
5.2 Introduction to the FDE . . . 40
5.3 Elaboration on the FDE GUI . . . 42
5.3.1 Menu Bar . . . 42
5.3.2 Tree . . . 44
5.3.3 ToolBar . . . 45
5.3.4 Editable Windows . . . 46
5.4 Design Patterns Used . . . 47
5.4.1 Façade . . . 47
5.4.2 Mediator . . . 48
5.5 Use Case Diagrams . . . 49
5.6 Class Diagrams . . . 50
5.6.1 FacetMediator class . . . 51
5.6.2 FacetFrame class . . . 51
5.6.3 FacetStatusTextCtrl class . . . 53

5.6.4 FacetToolBarManager class 53
5.6.5 FacetTree class 53
5.6.6 FacetNotebook class 55
5.6.7 Facade class 55
5.7 Message Sequence Charts 59
5.7.1 FDE Initialisation 59
5.7.2 Creating a Facet 60
5.7.3 Opening a Facet 61
5.7.4 Saving a Facet 62
5.8 Chapter Summary 62

6 Facet Examples **64**

6.1 Simple CORBA Service Facet 64
6.2 Mediator Facet 67
6.3 Web Server Tutorial Facet 68
6.4 Chapter Summary 70

7 Conclusion **72**

7.1 Discussion 72
7.2 Conclusion 74
7.3 Recommendations for future work 75

References **76**

A Meta-π DTD **79**

A.1 meta-π.dtd 79

B Facet DTD **82**

B.1 Facet.dtd 82

B.2 FacetSource.dtd 88

C Facet Hierarchy and Implementation Language **90**

C.1 Sample Facet Hierarchy XML file 90

C.2 DTD for Implementation Language Comments 91

C.3 Sample Implementation Language Comments XML file 91

D Additional UML Diagrams **92**

E Simple CORBA Service Facet **101**

F Sample meta-π xml file **106**

G CD Guide **117**

List of Figures

2.1 NGN business model 9

2.2 Meta-hierarchy used in MDA 15

3.1 Facet Developer with Stakeholders 24

3.2 Informational Context of the Facet Developer 25

4.1 Facet decomposed into Generic and Implementation 29

4.2 Meta-π compared to MOF 30

4.3 Facet Hierarchy 35

5.1 Facet Development Environment 41

5.2 Facet Explorer 43

5.3 Facet Hierarchy 43

5.4 Implementation Language Comment Editor 44

5.5 Façade Design Pattern [1] 48

5.6 Mediator Design Pattern [1] 48

5.7 FDE Use Case 49

5.8 Facet IDE 50

5.9 New Facet Wizard 52

5.10 Facet Chooser 53

5.11 Facet Explorer 54

5.12 Facet Hierarchy and Implementation Language Comment Editors 55

5.13 FacetNotebook 56

5.14 Facade 58
5.15 FDE Initialisation 59
5.16 Creating a Facet 60
5.17 Opening a Facet 61
5.18 Saving a Facet 62
6.1 Simple CORBA Service Facet: FDE Tree Structure 66
6.2 SimpleCORBAService Facet Structure 67
6.3 Mediator Facet: FDE Tree Structure 69
6.4 Mediator Facet Structure 70
6.5 Web Server Tutorial Facet: FDE Tree Structure 71
D.1 XML Explorer 92
D.2 Common Windows 93
D.3 Editor 93
D.4 Facet Data Models 94
D.5 Sub-Facet Data Models 95
D.6 Sub-File Data Models 96
D.7 Facet Editor 97
D.8 Facet Window 98
D.9 File Window 99
D.10 File Window Component Wizard 100
E.1 ORB Creation Facet: IDE structure 101
E.2 Read IOR From File Facet: IDE Structure 102
E.3 Write IOR from File Facet: IDE Structure 102
E.4 CORBA Server Facet: IDE Structure 103
E.5 CORBA Server Side Object 104

E.6 CORBA Client Facet: IDE Structure 105

G.1 Directory structure of accompanying CD 118

List of Tables

5.1 Tree Icons 45

Acronyms

API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
DPE	Distributed Processing Environment
FDE	Facet Development Environment
GII	Global Information Infrastructure
GUI	Graphical User Interface
ICA	Information Communications Architecture
IDE	Integrated Development Environment
IDL	Interface Definition Language
IN	Intelligent Networks
IOR	Interoperable Object Reference
JAIN	Java API's for Integrated Network
MDA	Model Driven Architecture
MOF	Meta Object Facility
NGN	Next-Generation Networks
OMG	Object Management Group
OO	Object Oriented
ORB	Object Request Broker
OSA	Open Software Architecture
QoS	Quality of Service
RAD	Rapid Application Development
RM-ODP	Reference Model for Open Distributed Processing
RUP	Rational Unified Process
SATINA	South African TINA Trial
SATINA-NGN	South African TINA NGN Trial
SCE	Service Creation Environment
TINA	Telecommunications Information Networking Architecture

UML Unified Modeling Language

XMI XML Model Interchange

XML eXtended Markup Language

Definition of Terms

Meta Model

A Meta Model is defined as a model that describes another. Meta Models can be arranged hierarchically to describe lower level models.

Next Generation Network

The Next Generation network is a telecommunications grade network that is built on a packet-based network. The NGN is able to support a multitude of multi-media and multi-party services.

Model Driven Architecture

The Model Driven Architecture specifies that development of any software will proceed as an evolution from a platform independent model (PIM) to a platform specific model (PSM). It is envisaged that the MDA will facilitate maintenance and portability of software.

Platform

A Platform is an architecture, framework or environment which is the context of operation for the target functionality.

Best of Breed Characteristics

Reuse solutions can be categorised into several groups or "breeds". Within a particular "breed", a reuse solution exists which qualifies to be the best because the reuse solution has the best characteristics for that "breed" of reuse solutions. Therefore, the "best of breed characteristics" are those characteristics that define the best reuse solution within a category or group of reuse solutions.

Implementation Language

The Programming Language used to implement a specific piece of functionality. Including Programming Languages such as C++, Java, Python, SmallTalk, etc.

Set Notation Symbols

Sxx **:** The convention to represent sets is a S followed by the description of the set. An example is: a set of Apples is SApples or SAPPLES.

$\{\}$ **:** Defines the contents of a set. $SApples = \{a, b, c\}$ tells us that the set SApples has elements a,b and c.

$\in$**:** *Element of.* The symbols is used to show that a value or variable is an element of a set. $x \in SApples$ shows us that x is an element of the set of Apples, $SApples$.

U**:** *Universal Set.* The set of all elements that exist. U_apples will refer to the universal set of all the Apples in the world.

$\emptyset$ **:** *Null Set.* A null set is an empty set and is equivalent to a zero in traditional mathematics.

$\forall$**:** *For All.* A symbol that refers to all the elements of a set.

$\cup$**:** *Union* of two sets. $\cup$ is equivalent to addition. Here we are adding two sets. Therefore, if we define a set of Apples $SAApples$ and $SBApples$. Then $SAApples \cap SBApples$ is equal to the set of all Apples in $SAApples$ and $SBApples$.

$\cap$**:** *Intersection* of two sets. $\cap$ represents the overlapping of two sets. If we define a set of Apples $SAApples$ and $SBApples$. Then $SAApples \cup SBApples$ is equal to the set of Apples that are elements of both $SAApples$ and $SBApples$.

$\subset$**:** *Subset* defines the contents of a set being elements to another set. $SAApples \subset SBApples$ shows us that all the elements of $SAApples$ are also elements of $SBApples$.

$\subseteq$**:** *Subset and Equals.* This symbol is an extension of $\subset$. $SAApples \subseteq SBApples$ means that $SAApples$ can also have exactly the same set of elements as $SBApples$.

$\supset$ **:** *Super set.* Similar to $\subset$ except that $SAApples \supset SBApples$ means that $SBApples$ is a subset of $SAApples$.

$\supseteq$ *Super set and equals.* Similar to $\subseteq$ with the characteristics of $\supset$.

$\vee$ **:** a binary or.

Chapter 1

Introduction

Telecommunication companies (telcos) have traditionally played the role of connectivity provider. Deregulation of the telecommunications marketplace forced the telco to differentiate itself from its competitors. Intelligent Networks (IN) and the proposed B-ISDN declared that provisioning of services was the key to differentiation and gaining market share in a deregulated telecommunications marketplace. Telcos were convinced and thus also took on the role of service provider. Within their new service provider role, the telco envisaged creating and deploying a multitude of services rapidly and economically. Vendors could not satisfy the telco's *vision* within the framework of legacy technologies. Yet, telcos still cling onto this *vision*.

Presently we are in the midst of an evolution of the telecommunications network. This evolution marries the computing(Internet) world to the telecommunications world resulting in a packet-based QoS-enabled network that is being globally termed the Next Generation Network (NGN). The NGN draws inspiration from the computing world to flexibly deliver services to end-users thus building on the telco's service provider role. Various initiatives proposed solutions for the NGN, or parts of it, resulting in a plethora of platforms. Web Services, Global Information Infrastructure (GII), Telecommunications Information Networking Architecture (TINA), Parlay and Java API's for Integrated Network (JAIN) are just a few examples of such initiatives. With so many initiatives, the path toward the NGN is chaotic and uncertain. Nonetheless, some Telcos are taking bold steps by employing solutions such as Parlay and Web Services. These bold steps are prompted by increasing competition the world over. Global competition is a direct result, in most cases, of deregulation of the telecommunications market [2].

Within this landscape of varying telco decisions and an uncertain path toward the NGN, how will the telecommunications world realise its *vision*? By marrying the Internet world, the telecommunications world is allowed to take advantage of its methodologies, concepts, processes and structure. Rapid Application Development (RAD) environments can help realise

the *vision* and are called Service Creation Environment (SCEs) in the telecommunications domain. RAD environments take advantage of modeling languages such as Unified Modeling Language (UML), software reusable components such as Enterprise JavaBeans™(EJBs), software lifecycle processes such as Waterfall Model and software development methodologies such as Extreme Programming in an attempt to consistently deliver software on time and within budget. Already, certain Intelligent Networks (IN) platforms have SCEs that enable *rapid* service creation. We explore a few SCEs in more detail in section 1.1.

1.1 Service Creation in the NGN

Service creation is abstract and general since there are not many detailed guidelines available on how to structure each of its phases [2]. Nonetheless, there have been many attempts to create SCEs. Many of the SCEs have a restricted scope in terms of services created and the target deployment platform, and are hence termed dedicated SCEs [3].

IN SCEs were the very first environments for creating telecommunication services. These SCEs introduced relevant concepts such as 'drag and drop' of reusable components and development based on functionality. IN SCEs used the Capability Set Specification which provided the service creation paradigm, reusable components and deployment process. The IN SCE did not incorporate modelling and reuse concepts that were being born in the Software Engineering world. Instead the IN SCEs used the IN CSS as a guide for modelling and reuse.

The TINA Consortium was the pioneer in the effort to marry the computing and telecommunications worlds. TINA is an architecture based on distributed object computing. It aims to improve interoperability, re-use of software and specifications, and flexible placement of software on computing platforms/nodes [4]. SCEs that were built around the TINA concept are worthy of investigation. These SCEs include SCREEN [5], FRIENDS [6] and TOSCA [7].

The SCREEN project stipulates five phases that complete the service creation process. In addition, cross phase activities such as quality assurance, project management and traceability are assumed. SCREEN attempts to structure the service creation process and abstract detail from the service developer.

The FRIENDS project is a model-based approach that realises an enterprise model for service creation. This enterprise model specifies a component developer and service developer. The component developer is responsible for the creation of reusable EJB-like components. The service developer creates services using the resulting components.

TOSCA successfully attempted to abstract detail from the developer. Its use of paradigms allowed less technically oriented individuals to develop services. Services were developed with the assumption that a flexible software framework exists upon which the service may execute. The reusable component existed at a high granularity introducing flexibility through properties.

We find a few points across the TINA SCEs that are worth pointing out. Firstly, all SCEs specified a reusable component at some level of granularity. The use of a reusable component signals that reuse is a driver for rapid development. This sentiment is echoed by Software communities in general. Secondly, most SCEs borrowed Software Engineering concepts such as UML modeling even though UML fails to completely describe telecommunications grade services. Thirdly, abstraction of detail from the developer is a by-product of reusable components. Abstraction is viewed as a further driver for rapid development. Lastly, all the SCEs were dedicated SCEs. Dedicated SCEs bind the environment to a platform or architecture. In all cases, services could not readily be migrated to other platforms or architectures.

We focus on the reusability aspect of service creation. We believe that reusability is a driving force for rapid economical service creation. In dedicated SCEs, the structure of reusable components are static. This does not hold true in the constantly evolving NGN environment that results in a plethora of architectures. Hence we need a reusable solution that can be applied uniformally across multiple architectures. To understand reusability, section 1.2 examines software reuse attempts.

1.2 Attempts at Software Reuse

Software methodologies such as Software Product Lines, Agile Programming, Dynamic Systems Development Method (DSDM) and Rational Unified Process (RUP) promote reuse. Implementors of software methodologies have limited options for an effective reusable software component. Their options include Microsoft Component Object Model (COM)^{TM} [8], Design Patterns [9], Sun Microsystem's Enterprise Java Beans^{TM}(EJBs) [10] and Object-Oriented Programming (OOP) [11] principles.

Microsoft's COM is a solution to implementation language dependencies. Microsoft COM allows reusability at the implementation-level. Microsoft COM is implemented as dynamically linked libraries (DLLs). DLLs can be seen as black boxes which expose their functionality as a set of interfaces. And, DLLs may use other DLLs. This makes for a single level of granularity for reuse that is very rigid and hard to change. A common problem with Microsoft COM is that experienced programmers experience difficulties when DLLs

do not provide them with the necessary functionality. Programmers using Microsoft COM objects are forced to import the entire object resulting in memory intensive applications. Memory usage can be minimised if the imported Microsoft COM objects fit the functionality requirements exactly instead of forcing the programmer to import functionality that will not be used in the programme. Hence the Microsoft COM objects can be improved by introducing a finer granularity of functionality to allow the programmer to more precisely choose the functionality that is required.

Design patterns are an attempt to describe successful solutions to common software problems. A design pattern is a proven solution to a recurring problem [9]. When related design patterns are woven together they form a "language" that provides a process for the orderly resolution of software development problems. Design pattern languages are not formal languages, rather a collection of interrelated design patterns, though they do provide a vocabulary for talking about a software problem. Both design patterns and design pattern languages help developers communicate architectural knowledge, learn a new design paradigm or architectural style, and help new developers ignore traps and pitfalls that have traditionally been learned by costly experience.

EJBs are defined by Sun Microsystems as part of the Java suite of solutions [10]. EJBs address business and enterprise aspects of software design in a distributed processing environment. A set of interfaces expose the EJB's functionality. The process of creating EJBs is supported by a number of business roles which effectively distinguish the aspect of creating the EJB from the aspect of deploying the solution. The EJB concept is developed to leverage the use of the Java platform. Although EJBs expose the business issues related to reusability, their implementation cannot be universally applied due to strong coupling to the Java platform.

As software programmes became more complex, programmers found that functional programming languages were inadequate in describing the complexities. OOP was seen as a step to move away from the chaos of functional programming [11] towards a better structured programming paradigm. OOP introduced concepts such as inheritance, containment and aggregation. Together these concepts helped in separating functionality such that they can be reused in other scenarios [12]. Although OOP helped to manage complexity, functional programming has its place within small mission-critical applications due to reduced computing overhead in functional programming languages.

Although not a reuse solution offered to the computing world, IN Service Independent Building blocks (SIBs) [13] were a major move forward in achieving rapid service development in the telecommunications domain. SIBs are script-like components that have black box characteristics and represent functionality. Service development is achieved by

stringing compatible SIBs together, effectively creating longer scripts, and thereby achieving complex functionality. The main contribution of the SIB methodology is its focus on and encapsulation of functionality.

Many reuse attempts have been successful in their own software paradigm but none can claim universal success, begging the question: *why has a reusable component not been successful in all domains?*

1.3 Classification and Analysis of Reuse Solutions

The basic premise of reusable components is:

First, do not reinvent that which has already been invented; second, construct new systems and enhance existing systems using building blocks already tested and proven [14].

Simply creating and announcing a reusable library does not work. Without a "reuse mindset", organisational support and methodical processes directed at the design and construction of **appropriate** reusable assets, reusable components become expensive pieces of archaic software [15]. Thus the proper state of mind must be achieved before reuse solutions can be employed.

Present reuse solutions can be classified into four categories [12] viz.:

1. design level: a high level description without attention to implementation;
2. inheritance level: propogation of behavioural characteristics from one reusable component to the next;
3. component level: self-sufficient component encompassing a concept such as functionality; and
4. code level: reuse of source code.

TOSCA Paradigms [7], Design Patterns [9] and Meta Object Facility (MOF) [16] can be classified as design level reuse solutions. Object-Oriented Programming (OOP) falls in line with the inheritance level, whereas, Microsoft COM™, Sun Microsystems EJBs™and IN SIBs are classified as component level reuse. Code reuse is the "cut and paste" standard that is familiar to many developers.

The high level specification in the design level reusable component decouples the solution from the implementation language. The decoupling introduces a top-down approach to reuse allowing the solution to be implemented in a number of implementation languages.

The top-down approach does not promise that an implementation of the high level description will exist in the implementation language of choice.

By contrast, code level reuse provides the implementation but creates an explosion of detail for the developer. An influx of detail can overwhelm the developer and prevent meaningful work from being done. The only advantage is the reuse of actual lines of source code.

Reusability at the component level is useful as it encompasses a concept such as functionality or business process. Complex concepts are created by stringing components together, just as IN SIBs are strung together to form more complex functionality. The distinct disadvantage of component level reusable components, as well as inheritance and implementation level components, is their tight coupling to an implementation language.

An inability to allow multiple reuse solutions to co-exist is a general shortcoming across reuse solutions. In addition, many reuse solutions tend to present unstable source code. The exception is design patterns where only proven solutions to existing problems qualify to become design patterns.

In summary, a reuse solution is required that embodies a top-down approach to reuse together with reuse at lower levels. The reuse solution should hide detail from the developer without hindering the developer's need to access and modify source code. The simultaneous co-existence of multiple reuse solutions should be possible. And, there must be loose coupling to the implementation language. The reuse solution should only present stable, tested source code.

1.4 Problem Objectives

The objective of the project is to design and implement a reuse solution that meets the requirements for a reuse solution targeted at the telecommunications domain. The reusable solution will strive to achieve the follow characteristics:

1. functionality centric: the reusable software is described by the functionality it offers;
2. multi-granular: achieved by structured containment of one reusable component by another;
3. platform-independent: independent of the deployment and/or execution environment;
4. implementation language-independent: independent of the language that the functionality is implemented in;
5. simultaneous co-existence of other reuse solutions (where possible);

6. abstraction of detail from the developer; and

7. agnostic of resources, data and context.

The reusable component is validated in a development environment. Later, we term this particular reuse solution a **Facet**.

1.5 Outline of Report

Chapter 2 examines proposed NGN solutions with respect to reusability. Architectures and platforms such as TINA, Parlay, JAIN and Web Services are investigated. Reuse in the computing arena is also examined. Chapter 3 introduces the proposed reuse solution as a redefinition of the concept developed by the TINA-CAT workgroup. The key aspects are introduced in this chapter but are elaborated in later chapters. Chapter 4 details the reuse solution in terms of its composition and structure. Chapter 5 introduces the development environment, called the Facet Development Environment (FDE), that enables creation and modification of the reusable components. This chapter also explains the design considerations in developing the FDE. The FDE is introduced functionally and technically with UML diagrams and User Interface snapshots. Chapter 6 solidifies the facet concept with a few examples that are played out using the development environment from chapter 5. The examples illustrate the facet concept, a facet's ability to allow the simultaneous co-existence of other reuse solutions and a facet's software paradigm independence. The conclusions drawn from this work together with discussions and proposals for further work is presented in chapter 7

Chapter 2

Contemporary Reuse Practices

Reuse is seen as a means to increase overall productivity of software and improve time to market resulting in increased revenue. Hence the driver for reuse is increased revenue or return on investment. To increase return on investment a company will embark on a reuse initiative from different organisational levels which, needs staff and processes for creating and encouraging the use of reuse solutions.

At the corporate level, corporate management allocates resources to the reuse initiative expecting to reap benefits in terms of improved product quality, increased productivity and shorter time to market. At the reuse department level, the reuse department will invest man hours into developing reusable components expecting to reap benefits by selling the reusable components to development teams. The development teams can be internal or external to the company. At the development team level, the risk of using reuse components is taken expecting benefits of increased productivity, quality and timeliness of projects [17].

Once all levels in a company are committed to reuse, decisions about reuse solutions to use must be made. Reuse solutions can be software design methodologies, programming paradigms, software lifecycles or source code sharing mechanisms. In addition, the reuse solutions can be categorised into one of the four categories listed in section 1.3. This chapter reviews reuse from the telecommunications and computing viewpoints. Section 2.1 explores telecommunications architectures such as Intelligent Networks (IN), Telecommunications Information Networking Architecture (TINA), Parlay and Web Services for reuse solutions that are offered. Section 2.2 discusses the more widely accepted reuse solutions of the computing domain such as Model Driven Architecture (MDA), Enterprise JavaBeans™(EJBs) and Design Patterns.

2.1 Reuse in Telecommunications

Traditionally, the telecommunications company (Telco) and the Vendor were the only stakeholders in the telecommunications domain. IN exists in the Telco-Vendor business model. The Vendor sells the IN platform to the Telco. The Telco offers value added services to the Consumer thus generating revenue.

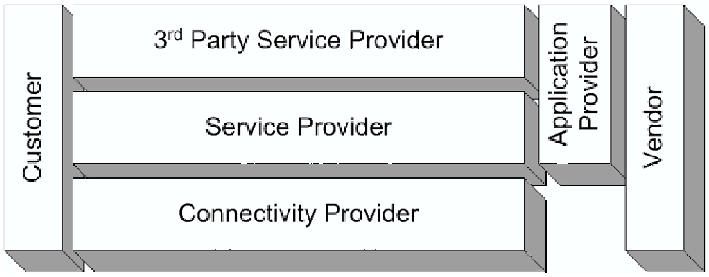


Figure 2.1: NGN business model

Introduction of the NGN ushers in a dynamic business model that expands on the Telco-Vendor business model. Figure 2.1 illustrates a general NGN business model. Separation of services from connectivity results in the Service Provider and the Connectivity Provider roles. Services can be offered by the Service Provider or the 3^{rd} Party Service Provider. The 3^{rd} Party Service Provider is subscribed to the Service Provider to provide applications and services to the Service Provider's consumers. The Application Provider develops and provides the 3^{rd} Party Service Provider or Service Provider with the applications or services they require. The Application Provider is effectively a sub-contractor employed by the 3^{rd} Party Service Provider or Service Provider to develop a particular service for deployment on a particular target platform. The Vendor keeps the traditional role of providing hardware and software that the parties want. The Application Provider and Vendor overlap such that the Application Provider can be considered as a Vendor that specialises in software.

Reuse in the pre-NGN business domain required innovative mechanisms. IN implements SIBs as a solution towards reuse in sub-section 2.1.1. TINA proposes a component level reuse in subsection 2.1.2. Reference Model of Open Distributed Processing (RM-ODP) is used by TINA. RM-ODP is discussed in subsection 2.1.3. Parlay provides an interface that is supported by underlying generic functionality. The generic functionality is used to build the complex logic of multiple services. Parlay is discussed in sub-section 2.1.4. Sub-section 2.1.5 discusses XML Web Services approach to reuse.

2.1.1 Intelligent Networks

IN was the pioneer in separating call control from service control. IN moved service control out of the switches and into an intelligent service provisioning platform. IN is described in the Capability Set specifications i.e. CS-1, CS-2 and CS-3. The Capability Set specification defines the IN Conceptual Model (INCM) to deal with the complexities of service creation. The INCM is a framework that consists of four planes. The four planes separate the concerns of roleplayers and creates a transition from abstract definition to detailed definition and deployment [13]. The four planes are:

1. Service Plane (SP): the SP depicts the services. Multiple services may exist on this plane. Multiple services may use the same functionality. Resulting in an intersection of functionality that is called a service feature. Service features can be used in multiple services and are a form of high-level reuse.

2. Global Functional Plane (GFP): the GFP consists of a Basic Call Process (BCP) and a number of Service Independent Building Blocks (SIBs). The BCP represents the call process that executes in the switch. The BCP has a number of trigger points or points of initiation (POI). A POI is the point when the switch hands over service control to the IN platform. The IN platform returns service control to the switch by BCP points of return (POR). SIBs are the core reusable components that IN developers use to implement a service. The SIB is described in terms of functionality offered with a well-defined stable interface. A SIB is tailored to a particular service by service specific data. SIBs are strung together to form service logic. Services and Service Features consist of SIBs.

3. Distributed Functional Plane (DFP): the DFP is a distributed definition of the functional entities and the actions performed at functional entities that underly SIB functionality.

4. Physical Plane (PP): the PP represents the physical entities and the protocols that make up the IN platform.

SIBs are described from an external view. A SIB's external view consists of a logical start, multiple logical ends, service specific data, input call instance data and output call instance data. The external viewpoint of a SIB abstracts detail from the developer when creating services. IN Services are constructed by associating a SIB's logical end with another SIB's logical start.

The SIB methodology works well for IN to introduce reuse and improve the efficiency of service development. The SIB is an example of component-level reuse that is confined to a vendor specific IN platform.

2.1.2 Telecommunications Information Networking Architecture (TINA)

TINA is an open telecommunications and information architecture that hides the heterogeneity of the underlying network from the service developer. A Distributed Processing Environment (DPE) is used to hide the heterogeneity and to create perceived co-location of physically separated computational objects. A DPE also achieves implementation language independence by focusing on informational flows.

TINA has been instrumental in introducing concepts such as sessions and separation of concerns. Separation of concerns include the separation of applications from the environment upon which they run and separation of an application into service specific part and a generic management and control part.

The complexity of the domain that TINA addresses, is broken down into four main sub-architectures [4]:

1. Service Architecture: defines a set of concepts and principles that apply to telecommunication services.
2. Network Architecture: defines a set of concepts and principles that apply to transport networks.
3. Management Architecture: defines a set of concepts and principles that apply to software systems that are used to manage services, resources, software, and underlying technology.
4. Computing Architecture: defines a set of concepts and principles for designing and building distributed software and the software support environment.

The Computing Architecture addresses reuse and is used by the other three sub-architectures. The Computing Architecture achieve reuse by using the RM-ODP and specifying the computational modelling concepts (TINA-CMC) [18]. RM-ODP is discussed in sub-section 2.1.3. TINA-CMC gives rise to reusability by defining distributed software consisting of a group of computational objects. Each computational object offers one or more interfaces for interaction and to allow other computational objects to access its capabilities. In addition, the computational objects are implementation language independent and loosely coupled to each other. TINA's approach to reuse is classified as a combination of component reuse and high-level reuse (RM-ODP).

2.1.3 Reference Model of Open Distributed Processing (RM-ODP)

RM-ODP is a framework that assists in developing open distributed architectures. RM-ODP aims to enable the building of distributed architectures or systems that are open, flexible, and modular amongst other characteristics [19].

Information regarding a non-trivial system can grow exponentially. Trying to use the information becomes difficult if there is a lack of structure and organisation to the information. RM-ODP attempts to organise the information by abstracting the information to five viewpoints. The five viewpoints are [20]:

Enterprise viewpoint: is concerned with the business environment of the system and its role in the business environment.

Information viewpoint: is concerned with the flow of information within the system in terms of interfaces. The necessary processing of the information is part of this viewpoint.

Computational viewpoint: is concerned with the description of the system as interworking distributed objects. Objects interact with interfaces and can be sources or sinks of information.

Engineering viewpoint: is concerned with the mechanisms supporting system distribution.

Technology viewpoint: is concerned with the hardware and software details that make up the system.

RM-ODP introduces reuse at the Information viewpoint with later viewpoints focusing on the specifics of the solution. The five viewpoints can be used to describe the reusable component exhaustively and are a useful ingredient in the makeup of a reuse solution. RM-ODP is classified as a software design methodology at high-level reuse.

2.1.4 OSA/Parlay

The Parlay architecture defines an open yet secure framework for multi-media multi-party services. The key aspect of this framework is the API that enables a 3^{rd} Party Service Provider access to network capabilities in the Service Provider domain. The network capabilities of the Service Provider are bundled into Service Capability Features (SCFs). The SCF is an object that implements interfaces which is used in creating a service. Many services can use a particular SCF, thus promoting reuse. The use of SCFs is yet another

component level reuse solution in the telecommunications domain. The problem with Parlay is that the SCFs are just objects that implement interfaces. The glue for creating services is the Application domain where services are created and where reuse is most needed.

2.1.5 XML Web Services

XML Web Services allow programmes to exchange information using an XML-based interface. Each programme is registered with and located via a Web Service registry. Web Services expose an interface for data exchange and publish information on services offered. XML Web services do not tightly couple interacting programmes together [21].

XML Web Services use Simple Object Access Protocol (SOAP) [22] over Hypertext Transfer Protocol (HTTP) as the one-way communication mechanism. The SOAP message is placed within a HTTP Post or Get request. The SOAP message consists of a SOAP Header and Body. The SOAP Header can possess multiple headers that target intermediaries. Each intermediary is a Web Service programme. Thus multiple programmes can be targeted with a single invocation, that is SOAP has decentralized extensibility.

XML Web Services are implemented on Web Servers using server-side web application languages such as ASP, JSP, Perl and Python. Web Services do not present any reusability. Instead, Web Services rely on the reusability capabilities present in the underlying implementation language such as ASP, JSP, Perl or Python. In most cases, the only reusability that exists is code-level reuse.

2.2 Reuse in the Computing Arena

Reuse initiatives in the computing arena are active. This section focuses on some of the more prominent reuse initiatives with a view to exposing the advantages of each. We investigate Model Driven Architecture (MDA), Enterprise JavaBeans^{TM}(EJBs) and Design Patterns.

2.2.1 Model Driven Architecture (MDA)

The Model Driven Architecture (MDA) [23] is an evolutionary progression of the set of modelling standards developed and maintained by the Object Management Group (OMG). The MDA seeks to allow an implementation independent specification of system functionality to be implemented on different platforms through defined mappings. The MDA defines its standard modelling and interchange constructs in the Meta Object Facility (MOF) [16],

uses a standard data warehousing model (CWM) [24] and provides for standard exchange between various tools, repositories and middleware through XML Meta-data Interchange(XMI) specifications [25]. Together these specifications facilitate the realisation of the MDA objectives, listed below:

1. Definition of a solution in a manner that is detached from the desired target platform, that is, a Platform Independent Model (PIM);
2. Migrating the solution, at an appropriate stage, from the PIM to a Platform Specific Model (PSM) tailored for the desired target platform;
3. Existence of a shared meta-data infrastructure. The infrastructure allows interchange of models. The shared meta-data infrastructure is also called a meta-hierarchy; and
4. Adherence to an Object-Oriented Paradigm.

The types of target platform envisaged in the MDA documents include databases, networking nodes, programming languages and telecommunications platforms. Telecommunications platforms are often more complex and heterogeneous, requiring greater flexibility in service creation support.

The OMG has remained faithful to its initial specification, Unified Modelling Language (UML), by defining MOF using UML [26]. Consequently, Object Oriented Programming (OOP) is the only paradigm for system specification and design.

Recent telecommunications software architectures are object oriented and, provided that their specification is expressed in or translated into UML, appear amenable to the MDA approach [27]. However, functional and scripting paradigms remain effective for particular applications. An important legacy script-like methodology is IN SIB-based service creation.

The inability of MDA to easily accommodate other programming paradigms is seen as a problem in telecommunications service creation. Specifically, the MDA lacks the ability to effectively and explicitly define interactions between heterogeneous entities in telecommunications, such as interactions between a back-end object-oriented system and a script-based front-end web server application. A further shortcoming of UML-based system definition is the incomplete specification of object behaviour. Telecommunications standards, by contrast, make extensive use of SDL for detailed specification of behaviour.

MDA utilizes a meta-hierarchy to catalogue the parts of the MDA structure based on what each part describes. Figure 2.2 shows the conventional meta-hierarchy layers, described as follows [16]:

M0: The user object layer is comprised of the information that the user wishes to describe.

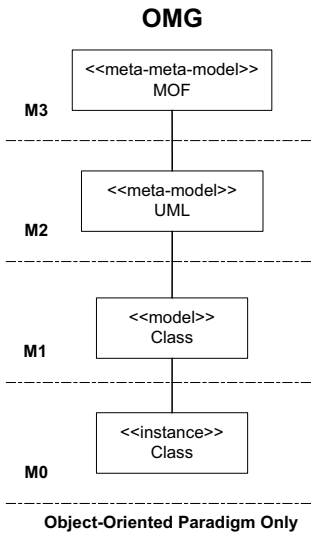


Figure 2.2: Meta-hierarchy used in MDA

This information takes the form of concrete instances of data;

M1: The model layer is comprised of meta-data that describes information the user wishes to represent. The aggregation of meta-data is a model;

M2: The meta-model layer defines the structure and semantics of meta-data used in the M1 layer: a language for describing different kinds of data; and

M3: The meta-meta-model layer contains the description of the structure and semantics of meta-meta-data used in layer M2.

The UML/MOF approach to reusability focuses on a generic object-oriented design. Reusability comes into effect once the structural description of the solution is general enough. At that point the solution can be moulded into a specific implementation. Although skeleton code (for a specific implementation) can be generated from UML models, the UML solution does not alleviate the problem that detailed source code will have to be written. In summary, the OMG approach via UML and MOF to reusability is a top-down approach where the developer is at a loss to incorporate already defined parts of the source code. UML and MOF are adequate for high level software design but fail to capture detail.

2.2.2 Enterprise Java Beans™

EJBs are the standard component architecture for building distributed object-oriented business applications in the Java programming language [10]. EJBs are created to abstract the

underlying complexity from that developer and allow EJBs to contain each other in an unstructured manner.

The EJB Specification applies a mindset which realises that software exists within a business domain. Six roles are defined as part of the EJB's business domain. The six roles are:

1. Bean Provider: creates EJBs from the client's perspective;
2. Application Assembler: assembles many beans into an application for use in a container;
3. EJB Container Provider: creates a container for the EJBs that provides non-system level tasks such as security;
4. EJB Server Provider: provides low-level requirements required by the container such as thread management and distributed object management;
5. Deployer: deploys the EJB, Application, Container and Server within an operational context; and
6. System Administrator: carries out post-deployment maintenance.

EJBs are an example of component level reuse with a containment capability. EJBs are tightly coupled to their implementation language but provide an interface for other non-Java objects to interact with it. This interface is defined using CORBA IDL.

2.2.3 Design Patterns

Design patterns capture and describe successful solutions to recurring software problems. Pattern languages provide a well-defined way to describe and catalogue design patterns. Most pattern languages have the following broad descriptive headings:

- Problem: describes the recurring software problem with as much detail as possible;
- Context: describes the pre and post context of the pattern;
- Solution: describes the logical steps required to implement the solution albeit from a high-level perspective;
- Intent: describes the intention of the proposed solution;
- Forces: describes the reasons for the approach used;
- Applicability: describes the areas or domains where the solution is applicable;

- Participants: describes the entities that participate in this solution; and
- Known Uses: describes the situations where the solution has been successfully applied.

Design patterns, most often, do not provide implementations of the solutions they present. The onus is on the developer to understand the solution and implement it. Nonetheless, design patterns are successful because of its strong binding to relevant software problems. Whereas with MDA the solutions are sometimes too abstract to grasp. Extensive information describing the design pattern helps the developer in using the design pattern correctly. Design patterns are an example of high-level reuse.

2.3 Chapter Summary

This chapter has examined contemporary software reuse solutions within the telecommunications and computing environment. There seems to be a general consensus that component reuse is an adequate solution to reuse issues.

IN SIBs emphasized a function-centric reusable component that possesses a black box description. The IN SIB black box description facilitates rapid development by aligning logical ends of SIBs.

TINA-CMC introduces a reusable component that is loosely coupled to other reusable components. Also, TINA-CMC emphasizes implementation language independence.

RM-ODP is a software design methodology that assists in structuring the influx of information about a reusable in such a manner that assists the development of a reusable component. RM-ODP creates awareness of five important aspects through which all reusable components should be described.

MDA introduces a high-level specification characterised with meta-levels. The MDA illustrates its capability to describe a number of implementations using one model. Meta-levels enable this capability. A disadvantage of the MDA approach is an inability to furnish sufficient detail.

EJBs focus on the business aspect of a reusable component. EJBs can contain other EJBs and interface with other implementations by means of an interface defined with CORBA IDL. A disadvantage is EJB's close coupling to its implementation language.

Design patterns address relevant software problems with a proven solution. Other reuse solutions do not force reusable components to be well tested or a recurring solution. Design

pattern languages describe reusable components extensively. Extensive descriptions prevent a misinterpretation of the design pattern.

The reuse solutions described here have diverse strengths and weakness. No single method provides the multi-granularity, platform-independence and implementation language-independence that is called for in section 1.4.

We assume that by amalgamating the strengths of the discussed reuse solutions we can approach a universally applicable reuse solution. Chapter 3 introduces a proposed universally applicable solution called **Facets**.

Chapter 3

Introduction to Facets

This chapter introduces a universally applicable reuse solution called a Facet. The Facet concept has, as its foundation, the redefinition of the TINA Reference Point-facet (RP-facet). The Facet concept expands on this foundation by borrowing *"best of breed"* characteristics from the reuse solutions mentioned in chapter 2. Section 3.1 begins to explain the TINA RP-facet as a basis for understanding the Facet, which we define in section 3.2. The definition of Facets is done mathematically using set notation. Section 3.3 illustrates how the Facet concept exists within a business context. The Facet is viewed from an Informational viewpoint in section 3.4.

3.1 TINA Reference Point-Facet

TINA is a complex architecture that is simplified by Reference Points (RPs). TINA defines RPs as a mechanism to separate concerns and to express the TINA architecture in terms of objective requirements. TINA RPs tend to be large, non-incremental and unstructured without measures to ensure interoperability. In a multi-vendor telecommunications environment interoperability between various TINA products is key. In an effort to add structure and reduce interoperability problems the TINA Conformance And Testing (CAT) workgroup was formed. The TINA CAT workgroup was tasked with proposing a framework for the conformance and testing of TINA RPs. The resultant framework is structured around a key concept called Reference Point facets (RP-facets). The framework gives a mathematical definition of RP-facets; defines a specification template for the definition of RP-facets and derives a conformance test method for TINA RP validation [28].

3.1.1 Reference Point Facet Definition

RP-facets are the functional building blocks of the TINA Reference Points. Each RP-facet can be defined as a meaningful self standing portion of functionality [29]. There can be a number of RP-facets in a TINA Reference Point, which are self-contained in terms of functionality and support the incremental specification of a TINA Reference Point.

A TINA Reference Point is made up of many interfaces each containing its own set of operations. Each interface or operation can be classified as either mandatory or optional. A mandatory interface or operation is one that is essential for the operation of the TINA Reference Point. The set of mandatory interfaces and operations, together with their dependencies, make up a core-based RP-facet. A RP-facet, created by a vendor, must contain the core-based RP-facet to pass conformance. The vendor RP-facet can implement any other functionality over and above the core-based RP-facet [30]. The core-based RP-facet provides a way for TINA Reference Points to be implemented partially thus greatly improving the extendability and ease of use of the TINA Reference Point.

Practical use of RP-facets for testing and validation is enabled by formal definitions of the RP-facets. Static and dynamic models together with behavioural characteristics of the RP-facets are captured using Formal Definition Languages(FDLs). At the technical level, a RP-facet is defined in terms of [29, p.5]:

- its interface specification;
- the roles between which there is interaction;
- a protocol (or object interactions and state changes) that the interface is intended to support;
- the other facets this RP-facet depends on; and
- typical usage of the RP-facet.

We mould the RP-facet to focus on its characteristics that assist reuse. The key RP-facet characteristics that assist reuse are, firstly, a meaningful self-standing portion of functionality, secondly, self-containment in terms of functionality, thirdly, incremental specifications, and lastly, the use of Formal Definition Languages (FDLs) to capture exact descriptions. The resulting moulded RP-facet is termed a Facet and is detailed in section 3.2.

3.2 Facets Redefined

The RP-facet was defined specifically for conformance testing of TINA Reference Points. The focus of our work is software reuse applicable to the NGN. Therefore, RP-Facets are extended and redefined as reusable components called Facets. The Facet's effectiveness is achieved (in the general Software Engineering domain and NGN Service Creation domain) by a number of *"best of breed"* characteristics borrowed from other reuse solutions such as MDA, EJBs, Design Patterns and IN SIBs.

Migration from a Platform-Independent Model (PIM) to Platform-Dependent Model (PDM) is a key concept embodied in Facets borrowed from the MDA. Each Facet must realise a functional description that is independent of a particular platform, in the MDA sense. Although the MDA focuses on top-down reuse, effective reuse is only possible when accompanied by a bottom-up approach. EJBs address this bottom-up approach.

EJBs are closely coupled to their implementation language, Java. Close coupling implies a bottom-up approach to reuse. The bottom-up approach works in conjunction with the top-down approach to achieve effective reuse. EJBs possess other desirable properties. EJBs address the need of commercial software projects by considering the business viewpoint of the reusable component [10]. Also, EJBs are accompanied by deployment descriptions that facilitate easier and faster deployment of EJBs. The deployment descriptor provides both structural and application assembly information. A business viewpoint, deployment descriptions and a bottom-up approach is borrowed from the EJBs.

Design Patterns present structural information as well as the purpose of the design pattern/reusable component [31]. Being aware of the reusable component's purpose helps the developer decide which is the best reusable component for the job. A pivotal concept of design patterns is the acceptance of proven solutions which reduces the existence of poor reusable components. Also, design patterns are independent of the software engineering methodology, software paradigm, implementation language and platform. Facets borrow the description capabilities of design patterns and the concept of only using proven solutions to recurring problems.

IN SIBs propose reuse as functional components-echoing a key concept of the RP-facets. IN SIBs are specified as black-boxes to help service developers create services without worrying about the underlying detail. Hence, service creation is reduced to matching outputs to inputs of various succeeding SIBs. SIBs are dedicated reusable components in a single technology environment. The black-box approach encounters difficulties in a multi-technology environment. Difficulties encountered are due to interoperability issues. Knowing detailed information about the reusable component assists developers in choosing appropriate reusable components in a multi-technology environment. A detailed description

is termed a white-box definition. We draw the idea of a functional component, black-box definitions and white-box definitions from the IN SIB.

Processing these *"best of breed"* characteristics results in a description of the Facet concept as defined mathematically in sub-section 3.2.1. Page xvii provides brief explanations of the set notation used.

3.2.1 Mathematical Definition of Facets

Definition 1: Facet F has a functionality description F_F such that

$$F_F \in U_{functionality} \tag{3.1}$$

In addition Facet F has a set of Technology descriptors F_T; has a set of Implementation Languages F_{Impl} and has a set of Platforms $F_{Platforms}$, for which implementations exist.

Definition 2: U_{SI} is the universal set of Implementation Languages. A Facet is Implementation Language independent iff

$$F_{Impl} \subseteq U_{SI} \text{ and } F_{Impl} \neq \emptyset \tag{3.2}$$

Definition 3: U_{SP} is the universal set of telecommunications platforms. A Facet is Platform independent iff

$$F_{Platform} \subseteq U_{SP} \text{ and } F_{Platform} \neq \emptyset \tag{3.3}$$

Definition 4: Facet Hierarchy (FH)[1] can be described as a set of discrete categories describing functionality called SFH. The elements of SFH, called sfh_i are ordered such that:

$$sfh_i > sfh_{i+1} \; for \;\; i = 1 \ldots n \tag{3.4}$$

Definition 5: F_F has a FH corresponding to an element of SFH at F_H. F_H is also called the Facet Hierarchy of the Facet.

$$F_H \in SFH \tag{3.5}$$

Definition 6: Facet F has a set of facets $SFacets$ which obey the same rules as F. In essence $SFacets = \wp(F)$. The Facet Hierarchy of $SFacets_i$ is $SFacets_i^H$ which is an element of SFH. The functionality of $SFacets_i$ is denoted by $SFacets_i^F$.

$$SFacets_i^H \in SFH \tag{3.6}$$

[1]The Facet Hierarchy is a user-defined hierarchy of discrete levels that Facets are associated with. Refer to section 4.5 for more information

$$SFacets_i^F = SFacets^F_{i\ mandatory} \cup SFacets^F_{i\ optional} \tag{3.7}$$

$$SFacets^F_{i\ mandatory} \neq \emptyset \tag{3.8}$$

Rules of Containment: This rule explains how Facets may contain other Facets.

$$\bigcup_{i=1..n} SFacets_i^F \subseteq F_F \tag{3.9}$$

$$SFacets_i^F \cap SFacets_j^F = \emptyset, \quad i \neq j \tag{3.10}$$

$$SFacets_i^T \subseteq F_T \text{ and } SFacets_i^T \supseteq F_T \tag{3.11}$$

$$SFacets_i^H \leq F_H \tag{3.12}$$

Cumulativeness Functionality If a set of Facets $SFacets$ are to be added together, the Facet Hierarchy of the cumulative functionality in relation to the Facet Hierarchy of individual functionalities is:

$$Facet^H_{Cumulative} \leq \max(\cup_{i=1...n} SFacets_i^H) \tag{3.13}$$

Lemma 1 Equation 3.4 introduces a structure for categorisation of Facet Functionality. Definition 5 and 6 link Facets to this structure such that a Facet may be categorised into *one* category of the Facet Hierarchy. By definition of 3.4 the structure for categorisation is multi-level. Therefore the Facet concept is multi-granular.

Lemma 2 Equations 3.7 and 3.8 show that Facet F is implemented with a mandatory and optional set of component Facets. This implies that optional component Facets may be excluded from the set of component Facets without violating the functionality requirements of Facet F. Therefore, Facet F can be implemented partially with the mandatory set of component Facets, where necessary.

Lemma 3 Definition 6 defines $SFacets$ as a $\wp(F)$. This definition can be recursively applied to $SFacets$ leading to a realisation that the functionality of F is expanded by $SFacets$. Reversing this process we find that functionality is being continuously abstracted until the the functionality of F which is presented to the developer.

With a clear understanding of Facets and their characteristics, we describe Facets from different viewpoints in sections 3.3 and 3.4 to complete the facet picture.

3.3 Business Viewpoint

The stakeholders that have a vested interest in the utilisation of reusable components for service creation are the Application Provider, Service Provider, 3^{rd} Party Provider and vendor. These stakeholders have a relationship as discussed in section 2.1 which is consistent with the Facet concept.

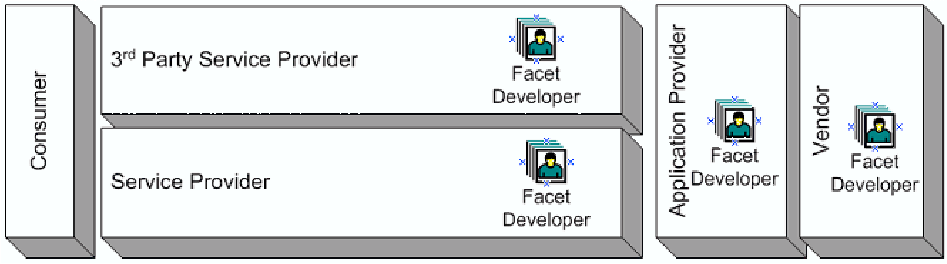


Figure 3.1: Facet Developer with Stakeholders

We introduce the Facet Developer (FD) role which is responsible for creating, maintaining and advising on the use of facets. Figure 3.1 shows a FD as a job description of an individual who can be a member of any of the stakeholders. The FD has a responsibility to the stakeholder to maintain a database of Facets such that the majority of Facets are actively utilised in service creation projects. The FD must also ensure that every Facet in the database abides by the rules of a Facet defined in sub-section 3.2.1.

The FD interacts with Programmer(s) and Service Creation Teams. The reader should be aware that the Programmer and FD could be the same individual although each job description has its own peculiarities. The FD is not responsible for the way that service creation teams use Facets.

In a Vendor environment that provides a single platform, the FD can only emphasize the implementation language independence (see Equation 3.2) of a Facet. FDs in the other stakeholder environments, in addition, emphasize platform independence (see Equation 3.3).

3.4 Informational Viewpoint

In creating Facets, the FD must exercise discretion to ensure that only reusable components with a proven track record are admitted into the database of reusable components. Also, each Facet must possess sufficient information to ensure effective utilisation. The FD is encouraged to formulate Facets from reusable components provided by multiple vendors for various service execution platforms. If a Facet is composed of multiple implementation from differing vendors, the platform independence characteristic of Facets is solidified.

If a service is composed of a number of platform independent Facets (see Equation 3.3) then platform migration is implied. Platform migration means moving a service from platform x to platform y which is *only possible if* all constituent Facets have the capability to migrate to platform y. Platform migration greatly improves service development time for the stakeholder who envisages the use of multiple service execution environments and where

required services already exist for non-target platforms.

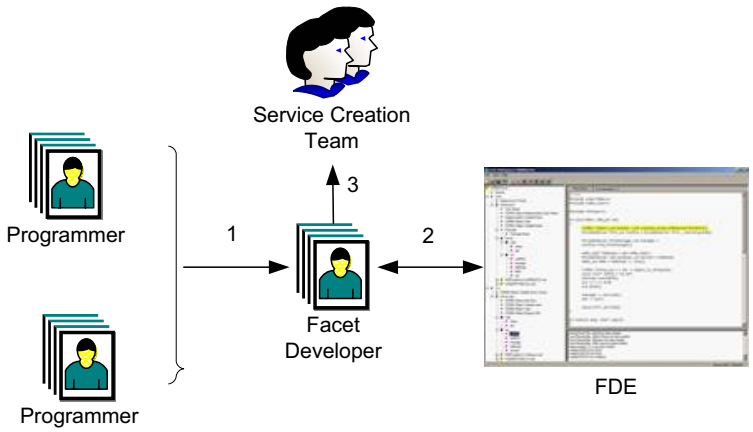


Figure 3.2: Informational Context of the Facet Developer

Figure 3.2 shows the informational relationship the FD has with other teams. The programmers include the high level system designers and code developers that create implemented functionality. The service creation teams are responsible for getting services to market in a cost effective manner. A set of programmers will create an incarnation of functionality in a programme. These programmes are submitted to the FD for verification through interaction 1 in figure 3.2. The FD determines if the proposed programme meets the requirements of a Facet. If the programme is adequate, it is added to the catalogue of Facets else the programme is returned to the programmers with reasons that explain the invalid aspects of the submission.

Interaction 2 shows the FD appending the Facet to the Facet catalogue via a Facet Development Environment (FDE). The FDE is used to manage the Facet catalogue and the individual Facets.

Interaction 3 shows the FD interacting with the service creation team. The service creation team has a mandate to create a service that encompasses a set of functionality. The FD helps the service creation team to decompose the service into fundamental functionality. Once the fundamental functionality is realised, the FD can propose possible Facets that encompass the required functionality. Upon requests from the service creation team, the FD will facilitate the creation of new Facets to meet functionality requirements. The task of creating new Facets can be forwarded to the Programmers or carried out by the FD as a composition of existing Facets.

3.5 Chapter Summary

This chapter explains the origin of Facets and defines Facets mathematically. A Facet is an implementation-language independent, platform-independent, multi-granular reusable component that is defined in terms of its functionality. A Facet's functionality is categorised into a level in the Facet Hierarchy. Facets may contain other facets.

This chapter also presents Facets from a business and informational viewpoint. The business viewpoint focuses on the stakeholders and their contractual interactions. The Facet Developer (FD) role is introduced as a job description that may exist in the Application Provider, Service Provider, 3^{rd} Party Service Provider or Vendor stakeholder domains. The FD interacts with programmers and service creation teams. The FD could also be a programmer.

The informational viewpoint describes the informational interactions between the FD and the programmer(s) or service creation team(s). The Facet Development Environment (FDE) is the application that the FD uses to manage the Facet catalogue. The concept of platform migration is stated as an advantage of using Facets.

The concepts, definitions and viewpoints discussed in this chapter paint the landscape for describing the key enablers that support the concepts of the Facet. Chapter 4 discusses the key enablers of Facets in more detail.

Chapter 4

Key Enabling Concepts

Chapter 3 introduces the Facet in terms of its goals and characteristics. This chapter explains the implementable concepts that realise the goals and characteristics of Facets.

Section 4.1 discusses the Generic and Implementation definitions which help to create a MDA approach to the solution. Generic and Implementation definitions are supported by the underlying informational model called meta-π (section 4.2). Meta-π satisfies most of the goals of the Facet, such as:

1. Five Viewpoints of RM-ODP;
2. Deployment descriptions;
3. Design pattern-like detailed descriptors;
4. Black-box and white-box approaches; and
5. Encapsulation of functionality.

Meta-π also contains two concepts which are vital for achieving a code-level reuse, Placeholders(section 4.3) and Variable Blocks (section 4.4). They are used for user-defined variances in the source code. Section 4.5 describes the Facet Hierarchy which is used by meta-π, and hence the Facet, to illustrate the level of functionality that is being encapsulated. The Facet Hierarchy introduces structure to Facets thereby easing the management of contained Facets and allowing partial implementations.

4.1 Generic and Implementation Definitions

Equation 3.2 and 3.3 specify implementation-language and platform independence, respectively. If solutions are independent of the implementation-language and platform then we

reason that the solution can be re-aligned to a platform other than its present one. This process is called platform migration. Platform migration is a complex concept to implement due to the difficulties involved in "translating" code from one language or platform to the next. Platform migration issues are eased by decoupling of a reusability description from its implementation and making decoupled implementations aware of their data, context and resources. Facets accomplish decoupling by *Generic* and *Implementation* Definitions. Data, context and resource awareness is achieved by the underlying model, meta-π, discussed in section 4.2.

The relationship of Generic and Implementation definitions to each other and a particular Facet is demonstrated mathematically in Equations 4.1, 4.2 and 4.3. Figure 4.1 illustrates this relationship graphically.

Definition 7: If there exists a Facet F; a Generic Definition $generic$ and a set of Implementation Definitions $SImpl$ then

$$F \rightarrow \{generic\} \text{ and } generic \notin SImpl$$
$$F = \{generic\} \cup SImpl \tag{4.1}$$

GI Rule: If $generic$ has a set of arbitrary descriptions $SDGeneric$ and an arbitrary element in $SImpl$ at k has a set of arbitrary descriptions $SDImpl_k$ then

$$SDGeneric \subset SDImpl_k \tag{4.2}$$

Definition 8: If there exists a set of source code SC then

$$srcfile_j = \sum_{i=1\ldots n} (iff\ SC_i \in SImpl_j \rightarrow SC_i)\ j = 1 \ldots m$$
$$SImpl_j \rightarrow \{srcfile_j\} \tag{4.3}$$

Figure 4.1 shows that a Facet is made up of a single Generic Definition from which multiple Implementation definitions inherit information. The Implementation definitions build on the body of information in the Generic definition. All source code that is necessary for a particular Implementation definition is placed in a corresponding *.src* file.

Facets use other facets in the Implementation Definition. Therefore, containing and contained Facets must have common technology descriptions, which echoes the *Rule of Containment* [Equation 3.12]. Both the definitions and $.src$ files use XML syntax to represent information and their tag structure is described by a document type definition (DTD). Appendix B contains the DTD for a definition (i.e Facet.dtd) and $.src$ file (i.e. FacetSource.dtd).

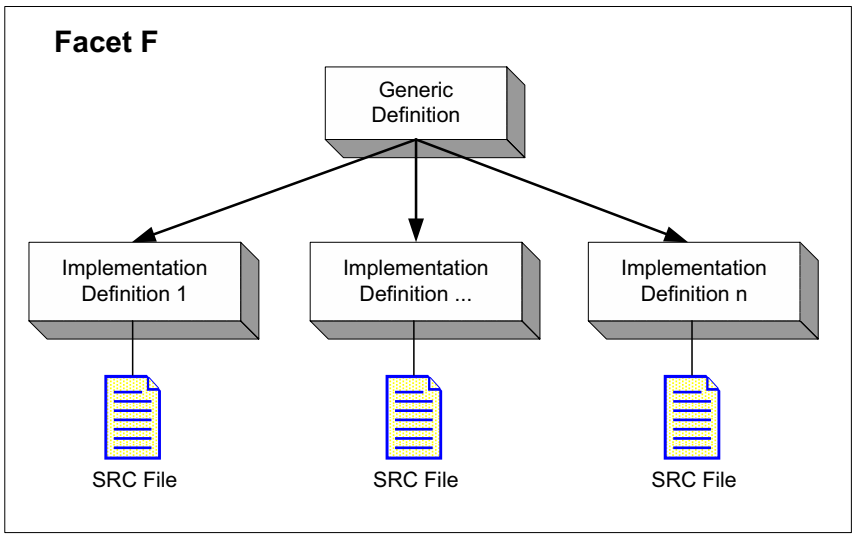


Figure 4.1: Facet decomposed into Generic and Implementation

4.2 Meta-π

Meta-π is the underlying informational model supporting the Generic and Implementation definitions that introduces data, resource and context agnostic features into the reusable component.

Each implementation language and platform has a structured way of describing implementations. Structured descriptions ensure that relevant information exists for the reusable component to be understood and used effectively. By detaching the Facet concept from implementation languages and platforms, the structured descriptions is lost. Also the domain to be described is broadened. Hence an open-minded highly descriptive and structured approach is required. A starting point is the following statement:

Everything is Information!

Not concerned with atomic pieces of information, the statement's applicability is limited to entities. An entity can be defined as a structured self-standing piece of information, within a context, which processes input data into output data using resources. The meta-level concept, developed by OMG and discussed in section 2.2.1, is employed to describe entities. Meta-π is the meta-meta-model (M3) used to describe entities. Equation 4.4 shows the relationship between the meta-π model and the Generic or Implementation definitions (section 4.1) as a one-to-one relationship.

Definition 9: If there exists a set of Meta-π M3 models called $SMeta$ then

$$\{generic\} \vee SImpl_j \rightarrow SMeta_i;\ \ j = 1 \ldots n \text{ and } i = 1 \ldots n + 1 \tag{4.4}$$

To aid an understanding of meta-π, a comparison to the MOF structure is made. Figure 4.2 shows the meta-π model on the same level as the MOF model. Although meta-π contains UML/MOF defined structural definitions, meta-π is unable to precisely describe UML or MOF. Hence, meta-π is placed at the same meta-level as MOF. A distinction between the two approaches is MOF's dedicated Object-Oriented Paradigm whereas meta-π allows the description of any programming paradigm.

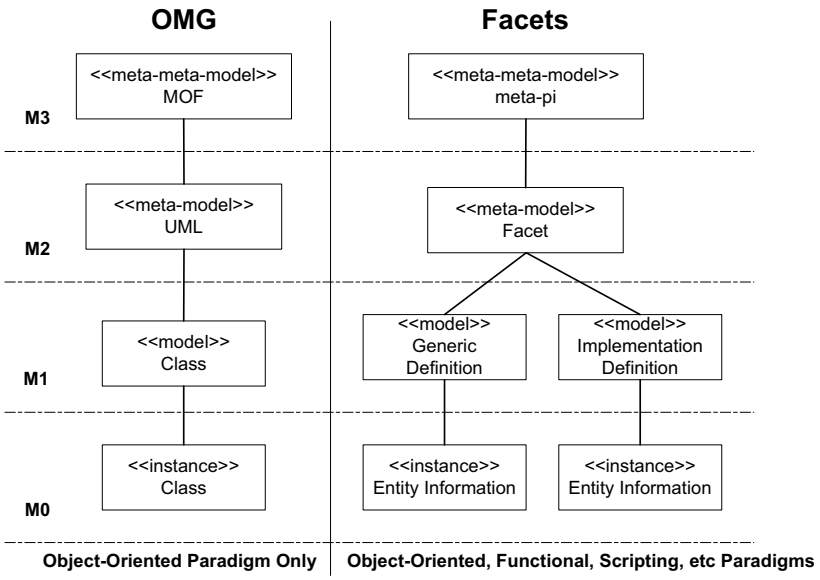


Figure 4.2: Meta-π compared to MOF

Meta-π describes the Facet conceptually. The M2 level is transparent in the implementation of Facets. Using Equation 4.1 level M1 is derived from level M2. The definitions of M1 describe the underlying Entity information at the M0 level.

An entity and Facet require an underlying model to be data, context and resource agnostic. Meta-π meets the requirements by creating six broad informational parts:

1. Definition: describes the entity in terms of its appearance, structure, behaviour, relationships, constraints, rules and objectives. Subsection 4.2.1 elaborates on this informational part further.

2. Context: describes an entity's environment from various viewpoints. A context-aware entity is able to "understand" the constraints placed on it by its environment. Subsection 4.2.2 elaborates on this informational part further.

3. Resource: describes the resources the entity relies upon for its correct functioning. A resource can be any other entity. Sub-section 4.2.3 elaborates on this informational part further.

4. Data: describes the informational characteristics of an entity in terms of the entity's input, output and internal workings. Sub-section 4.2.4 elaborates on this informational part further.

5. Documentation: describes unstructured information to ensure that other human beings will correctly interpret the model.

6. Extensions: describes information that should be structured but cannot be readily bundled under any of the other informational parts. Extensions also serve as a mechanism to extend the meta-π model. Examples of how to use a Facet is placed in this informational part.

Meta-π is implemented as an XML file, whose DTD can be studied in appendix A. The DTD distinctly illustrates the six informational parts.

4.2.1 Definition

The Definition of an entity or Facet is sub-divided into four categories:

1. Description;

2. Behaviour;

3. Interactions and

4. Logic.

The *Description* describes the appearance, structure and objectives of the entity or Facet. This category borrows the description field used for design patterns. Allowing for a high-level yet useful description of the Facet, from a functional perspective. Sub-section 2.2.3 introduces the Design Pattern description fields, which keep their original meaning within meta-π and are re-iterated for clarity:

- Problem: the problem that is being solved;

- Intent: the intention of the proposed solution;

- Forces: the reasons for the approach used;

- Applicability: the areas or domains where the solution is applicable;
- Participants: the entities that participate in this solution; and
- Known Uses: the situations where the solution has been successfully applied.

The *Description* also furnishes the information described in Equation 3.1, namely the set of Technologies and the Facet Hierarchy associated with a Facet. The Facet Hierarchy is responsible for the logical structuring of functionality that enables Facet containment. The Facet Hierarchy is discussed in section 4.5. Unambiguous structural descriptions are contained under a *Description* sub-field called *Structure*. The structure of the solution is in terms of its classes or modules, methods or functions, attributes, and internal interactions or associations. To this list of structural elements we append placeholders, global placeholders, variable blocks and global variable blocks as facet-specific structural descriptors.

The entity description mechanism allows the use of any reuse solution above the code-level. By disregarding atomic information, code-level reuse is excluded from entity descriptions and thus excluded from a complete meta-π model. Code-level reuse is important and is enabled by the facet-specific structural components that are bound to the source code. Placeholders, global placeholder, variable blocks and global variable blocks are the structural components that implement code-level reuse. Section 4.3 explains placeholders and section 4.4 explains variable blocks.

The *Behaviour* part is concerned primarily with the Facet's interaction with its surroundings. Behaviour of a Facet, at the technical level, is determined by the Logic which is embodied in the source code. The *Behaviour* we refer to extends beyond the technical issues and encompasses behavioural characteristics such as regulatory limitations. The behaviour of a Facet can be described by *Rules* that the Facet must abide by; the *Policies* the Facet must enforce and the *Limitations* that have been placed on the Facet.

The *Interactions* describes the external interactions that the Facet has with other entities. External interactions can be categorised as *Relations* or *Collaborations*. Relations describe a usage scenario whereas Collaborations describe a scenario where n number of Facets operate in conjunction to achieve a particular functionality.

The *Logic* has a primary focus on describing the internal workings of a Facet in terms of the logical steps necessary for implementing the Facet's functionality. Logic is usually used by the Generic Definition to specify the broad logical structure of the implementations. The logical structure of the Implementation definition is preferred within the Data informational part, discussed in sub-section 4.2.4.

4.2.2 Context

An entity requires a context to function or else its definition is meaningless. By employing the entity and separation of implementation from functionality concepts, the key environment distinguisher of any source code is removed. Thus, there is a need to explicitly define the context of the Facet and its corresponding implementations to realise a meaningful Facet and implementation. Not only does the *Context* informational part provide base source code contexts but can be extended to specify other context information. All the context information is categorised as being part of one of the following context categories:

1. Business Context: defines the business environment in terms of the stakeholders and their relationships;
2. Design Context: defines the context of the Facet during the design of the functionality;
3. Deployment Context: defines the required context to deploy the Facet; and
4. Operational Context: defines the required context for the Facet to be operational.

The context categories attempt to educate the user of the Facet's context at most points during its software lifecycle. Knowing the context of the Facet will further strengthen the case for the correct and effective use of the Facet. By realising that all software is part of a business process, the Business context has been included in-line with the EJB and RM-ODP concepts

4.2.3 Resource

At some point, a Facet's functionality will draw on resources that are either internal, on the boundary or external to the system boundary (the system boundary is described by the Context). Particular resources have higher priority than others thus resources can be prioritised in order of which is the most prefered resource. Resources can be physical, software, logical or informational.

4.2.4 Data

The Facet is characterised from an informational viewpoint using the Data informational part. The Data informational part is split up into Input, Algorithm and Output.

The Input part allows the user to define the information that flows into the Facet. Similarly, the Output part defines the information that flows out of the Facet. By viewing the Facet

with only an Output and an Input, the Facet's characteristics is simplified. This view of the Facet is called a black-box. The Input and Output parts also allow the user to specify pre- and post conditions, respectively, necessary for the operation of a Facet.

The Algorithm part expands the black-box view by exposing the internal logical processes of the Facet. Viewing the expanded internals of the Facet is termed a white-box view.

4.3 Placeholders

The *placeholder* concept is part of the suite of concepts that enable code-level reuse. Essentially a placeholder is any text in any file that can be replaced by any other text. Placeholders manage points in the source code that can vary. As an ancillary, this characteristic of placeholders allow templates to be created for creating classes, functions and attributes.

In larger Facets with multiple files per implementation, a situation may arise where an *object type* needs to be synchronised across multiple files. A placeholder is created within each file to allow the *object type* to be changed. These placeholders are linked using a global placeholder. Specifying a text value for the global placeholder is the same as specifying a text value for each placeholder thus synchronising the spatially differentiated *object types*.

4.4 Variable Blocks

The *variable block* is the second concept that is part of the suite of concepts that enable code-level reuse. The variable block represents any text within a source file that is optional to the functionality of the Facet. Variable blocks are either included into the source code or excluded by being commented out. Commenting the variable blocks leaves the user to modify the source code at some future date outside the boundaries of the Facet environment.

Variable blocks can contain placeholders but placeholders cannot contain variable blocks. Variable blocks are useful for creating software with additional "optimisation" code which would afford the user the **option** of using the "optimisation" code.

Global variable blocks are used in a similar manner as with global placeholders. The difference is synchronisation of multiple variable blocks across multiple source code files.

4.5 Facet Hierarchy

Facet Hierarchy (FH) introduces structure to Facets by producing rules for the containment of Facets. Since Facets can use other Facets, allowing partial implementations, the FH is developed to manage this usage or containment.

The FH is a user-defined multi-level hierarchical structure. Each level of the Facet Hierarchy represents an encapsulation of functionality. Facets are associated with the FH by defining a path to the FH level that is representative of the overall type of functionality the Facet encompasses. Any level can be configured to be an optional level, meaning that the optional level can be omitted when describing the path to a level specific to a Facet. The primary rule of the FH applicable to Facets is: A level in the FH can contain and use the same or lower levels of the FH but not higher ones.

Figure 4.3 shows "user-defined" example of a Facet Hierarchy, which is used in later chapters. The blocks with dashed outlines represent the optional levels in the FH.

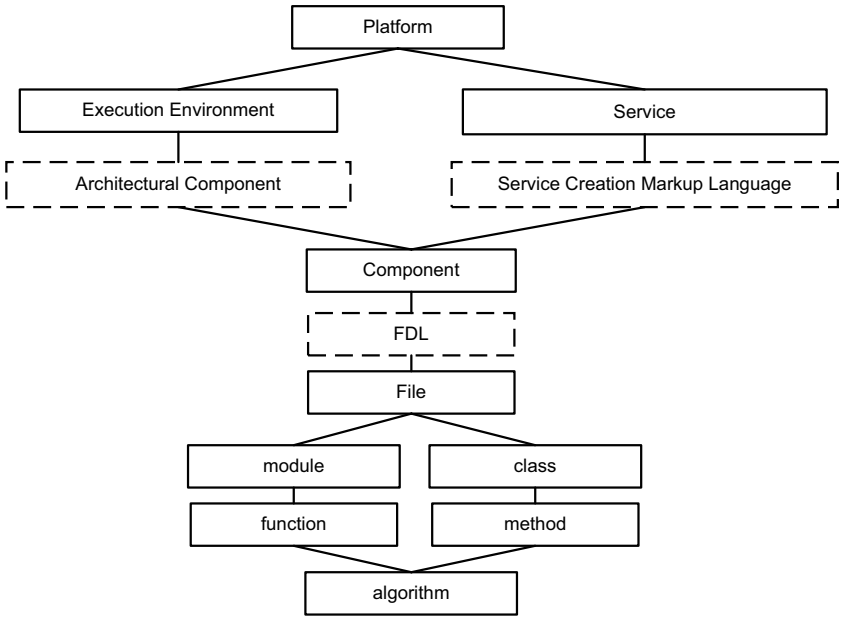


Figure 4.3: Facet Hierarchy

A path to a FH level is defined by starting at the top of the FH and moving down toward the desired FH level. If necessary, an optional level can be omitted from the path. An example of a path is: *Platform.Service.Component.FDL*. With the example path we see that the definition starts at the top (i.e. Platform), traverses down the right while skipping the optional level *Service Creation Markup Language* and then proceeding to the target level (i.e. FDL). The path definition separates the levels from each other using a full stop ('.').

If the example path (*Platform.Service.Component.FDL*) is associated with a Facet then the Facet can use or contain any Facet at the *FDL* level or lower FH levels. Higher FH levels cannot be used by the Facet.

The Facet Hierarchy is represented as an XML file. A sample Facet Hierarchy XML file is found in appendix C.1.

4.6 Chapter Summary

This chapter appends the implementable concepts to the description of Facets given in Chapter 3. The Facet consists of a Generic and multiple Implementation definitions. The Implementation definition extends the descriptions of the Generic Definition. Each Implementation definition has a $.SRC$ file associated with it.

The definitions are described by an underlying model called meta-π. Meta-π is described by six informational parts that encompass the design pattern and RM-ODP philosophies.

Placeholders and Variable blocks are introduced as facet-specific structural descriptors. They create a pivotal point where changes to the reusable component can occur.

The FH introduces structure to Facet containment. Each FH level represents a functionality encapsulation. FH levels can be optional or mandatory. The FH specifies what encapsulations of functionality a Facet can use or contain.

The GI-Rule allows high-level reuse to be implemented, at the same time binding it to an implementation, where available. The entity mechanism allows inheritance and component level reuse to be effectively described. While placeholders and variable blocks enable code-level reuse.

With an understanding of meta-π and the mechanics of Facets, the application for creating and manipulating facets is presented at in Chapter 5. The application for Facet creation and manipulation is called the Facet Development Environment (FDE).

Chapter 5

Facet Development Environment

Chapter 3 introduced Facets conceptually. Chapter 4 defined the implementable aspects of the Facet concept. This chapter presents the realisation of the Facet concept as an application that is used by Facet Developers and is called the Facet Development Environment (FDE). Section 5.1 investigates the technological considerations and trade-offs that are made in developing the FDE resulting in a set of supporting technologies. Thereafter the FDE is discussed in terms of its Graphical User Interface (GUI) and internal source code structure.

5.1 Design Considerations

Considerations when implementing the solution include the modelling languages, the parsers, the programming language, information storage solutions and Graphical User Interface (GUI) used.

5.1.1 Informational Modelling Languages

The underlying meta-π model requires a meta-descriptive modelling language to allow the meta-π model to be described at the M3 level. Standard Generalized Markup Language (SGML) [32] and eXstensible Markup Language (XML) [33] are the candidate meta-descriptive languages. SGML is regarded as the parent of XML. XML is chosen as the modelling language to be used.

XML is a restricted subset of SGML which is widely accepted by industry for its simplicity and ease of use. OMG uses XML as part of a standard to facilitate model interchange, called XML Model Interchange (XMI). By using XML it is easier to render meta-π models as UML classes for use in UML applications.

5.1.2 Parsers

Manipulation of XML is done via a *parser*. Two standards for XML parsers exists i.e. Document Object Model (DOM) [34] and Simple API for XML (SAX) [35]. The DOM standard represents the XML file as a hierarchy of parent and child nodes. Each node can be interrogated for data. Due to the hierarchical structure, moving between nodes is very easy. Also, the DOM model supports XPath and XQuery which are standards that allow searches to be performed on the XML document. DOM requires the entire XML file to be read into memory. Within the FDE, there will typically be thousands of models, each represented by a DOM residing in memory leading to a memory intensive solution. The memory intensive solution minimizes the computers capability to process information. Depending on the magnitude of a Facet, memory utilisation becomes critical.

SAX, on the other hand is not memory intensive. The SAX parser is a stream-based event-driven XML parser. SAX reads each character, one after the other, and generates events to signify the beginning and end of tags and data. By using SAX, the performance of an application can only be improved if the structure of the XML document is known. A disadvantage of SAX is its inability to search within the XML document effectively.

The FDE requires the XML structure to be easily manipulated and searched. Therefore the DOM is chosen as the parser.

5.1.3 Implementation Language

To implement the FDE timeously, a rapid prototyping interpreted language with an efficient Graphical User Interface (GUI) class set is required. The implementation must be able to execute on any platform with a minimum of modifications between platforms. Java, Perl and Python are the options.

The Java programming language is memory and processor intensive due to poor garbage collection. Thus, with larger applications the Java Runtime Environment tends to dominate CPU processing time and, in extreme cases, crash the Java application. From past experience, it was found that the Swing GUI classes are not easy to manipulate. Java is well accepted by industry with developers around the world creating many reusable source code components. Java is not a *glue language*, meaning that Java cannot easily incorporate functionality that has been written in an implementation language other than Java. By using a *glue language*, the programmer is able to pick the best tool for particular parts of the application to optimise the application's overall performance. Java is not chosen because it is not considered a rapid prototyping language, the GUI class is not efficient and Java is not a *glue language*.

Perl is referred to as a *glue language*. Perl is used in Web Server Applications, mission critical application and to interface applications written with different languages. Perl is implemented on most operating systems including Windows, Linux, Unix and HP-UX. Perl is a free-form language whose syntax is very similar to C++. Perl code tends to be written in a very unstructured non-user friendly manner. Perl uses the Tk\Tcl as its default GUI class set. Perl is considered to be a rapid prototyping language. Although the GUI class is efficient, it is felt that the GUI class is lacking in terms of ease of use and key functionality requirements.

Python falls into the family of *glue languages*. Python is similar to Perl in performance. Perl tends to be more powerful at regular expressions. Python is an interpreted language that is usually used by programmers for rapid prototype development. Python supports both OOP and functional programming. Python uses runtime type casting thus variables do not have to be explicitly created and managed. Python tends to out-perform Java at most operations except Object creation. One of the best GUIs that Python has to offer is the wxPython class set that is based on the wxWindows GUI class set. wxWindows is truly uniform in its presentation across operating systems. Python is chosen as the implementation language for its ease of use, portability, GUI class set and supporting packages such as pyXML for XML handling.

5.1.4 Information Storage

Most information, relating to the FDE, is stored in XML format. The options available for storage of XML information are:

- File System;
- Relational Database; and
- XQuery Database.

The File System is the simplest storage mechanism that can be used. Each XML document is stored as a flat file. XML is a bloated language with a lot of whitespace. The XML File footprint can be reduced by compression algorithms. The disadvantage of storing XML Files on the File System is the slow read and write processes combined with little or no efficient search capabilities. The advantage of using flat files is that no special software needs to be installed to store the information.

Relational databases such as MySQL and PostgreSQL are open source databases used in mission critical applications. When storing XML documents within relational databases, the database can be designed to be data-centric or document-centric [36]. Data-centric

means that the data contained within the XML document is saved and the structure of the XML document is lost. A data-centric database design allows for complex searches to execute efficiently. A document-centric database design maintains the structure of the XML document for quick reproduction of the XML file. The disadvantage of the document-centric approach is that data searching becomes complex and inefficient.

XQuery databases are designed for the storage of XML content in a manner that allows for efficient data searching with easy reproduction of the original XML document. XQuery databases are specifically designed to meet the requirements for storing XML files. At the time of developing the FDE, the XQuery specification was just released and no appropriate implementations were available.

For the purposes of rapid development the File System was chosen to store the information in XML format.

5.2 Introduction to the FDE

The purpose of the Facet Development Environment (FDE) is to help Facet Developers(FD) create and manipulate Facets; edit the Facet Hierarchy and edit the Implementation Language comments. The Facet Hierarchy is explained in section 4.5. The Implementation Language comments define the in-line text character(s) used in an implementation language for comments. The FDE uses these comments for variable block manipulation.

The FDE allows the FD to describe the Generic and Implementation parts of a Facet. Files, Sub-Facets, Placeholders, Variable blocks and other structural components can be added to Facets using the FDE. Generally, we refer to all the contained items of a Facet as an object. An object includes the Generic Definition, Implementation Definition, Files, Facets, Placeholders, Variable blocks, Global Placeholders, Global Variable blocks, Sub-Facets and other structural components.

Figure 5.1 shows an FDE as having six GUI components:

1. Menu Bar: The Menu bar has limited functionality that allows Facets to be opened, saved and closed. The Menu Bar also launches editors for the Facet Hierarchy and Implementation Language Hierarchy. Finally, the Menu Bar has the capability to launch a Facet Explorer.

2. Toolbar: A set of dynamically loaded object-specific tools. The tools are loaded when a particular object is selected in the Tree.

3. Tree: Facet structure is shown in the Tree as a set of hierarchically linked objects.

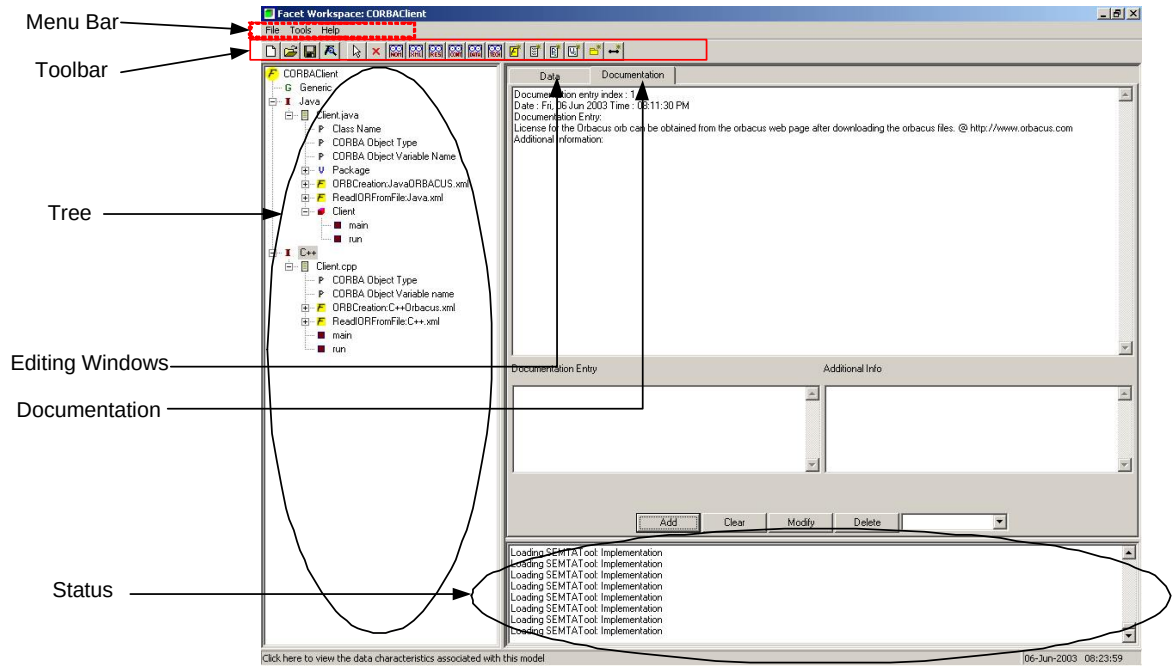


Figure 5.1: Facet Development Environment

The Tree allows easy navigation of a Facet and its components. Selection of objects in the Tree triggers specific tools to be loaded in the ToolBar.

4. Documentation tab: Documentation regarding a particular Facet can be added, retrieved, and manipulated. Miscellaneous information is documented.

5. Editing Windows tab: Object characteristics are edited from a particular viewpoint using an Editing window.

6. Status: Status is a non-interactive mechanism used to inform the FD about background operations.

Section 5.3 explains the FDE User Interface further. Section 5.4 describes the usage of design patterns in implementing the FDE. Section 5.5, 5.6 and 5.7 describe the use cases, class structure and message sequences of the FDE, respectively.

5.3 Elaboration on the FDE GUI

This section examines the FDE in greater depth. Use of the FDE requires an initial installation. Installation of the FDE prompts the FD for his/her name, email address, organisation and a *data* directory. The *data* directory is used to store Facets, templates for creating Facets, Facet Hierarchy definitions and Implementation Language comment definitions. The sub-directory, *Facets*, within the *data* directory, is used to store *all* Facets. Each Facet is described within its own directory. The Facet is described by a Generic.xml file and a multitude of implementation xml files with corresponding implementation src files. The name of the implementation xml file is derived from the implementation name with a .xml suffix. The name of the src file is similarly derived with a .src suffix.

The Menu Bar is discussed in sub-section 5.3.1 followed by a discussion of the Tree in sub-section 5.3.2. Sub-section 5.3.3 discusses the Toolbar followed by a discussion of the Editable windows in sub-section 5.3.4.

5.3.1 Menu Bar

The FDE Menu Bar consists of a File, Tools and Help menu.

The File menu is concerned with creating, loading, saving and closing of Facets. Creation of Facets launches a wizard that captures the basic description of the new Facet. The File menu also keeps a list of the last five Facets that were loaded. The FDE can be closed by selecting the Exit menu item under the File menu.

Facet querying and viewing, Facet Hierarchy manipulation and Implementation Language comment manipulation are done via menu items under the Tools Menu. Facet Viewing is done using the Facet Explorer shown in figure 5.2. The Facet Explorer allows the FD to explore the Facets within the *data* directory. The Facet Explorer consists of *Selector*, *Viewer* and *CXML* tabs. The *Selector* displays the Facets for the FD to choose from. The *Viewer* allows the FD to browse the generic and implementation definition of Facets. When an Implementation is selected, the FD can also display the source code associated with the Implementation. The viewer only displays the key descriptions of the Generic and Implementation Definitions. Viewing further information can be done by browsing the XML file of the definition using the CXML tab. Querying of Facets is supported by Facet name only.

Figure 5.3 shows the editor for the Facet Hierarchy (FH). FH is expressed in XML. The FH editor allows the FD to manipulate the levels of the FH. A FH level can be set to “optional” by specifying a “Yes” value for the optional attribute of the FH level. Specifying “No” for

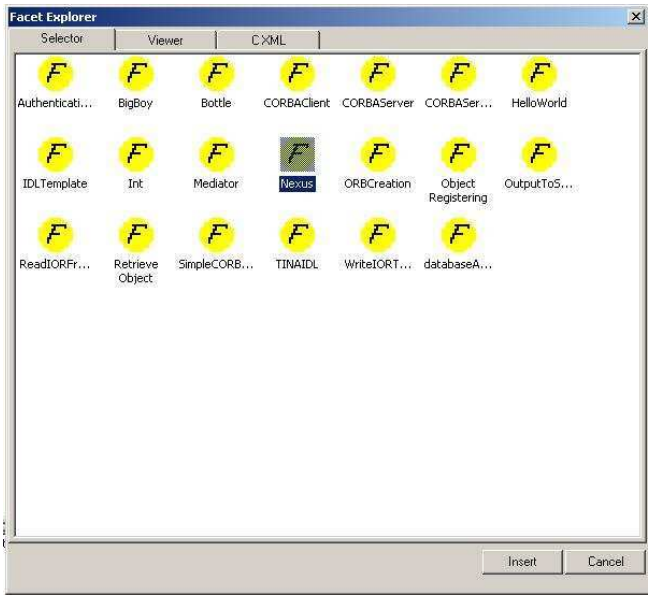


Figure 5.2: Facet Explorer

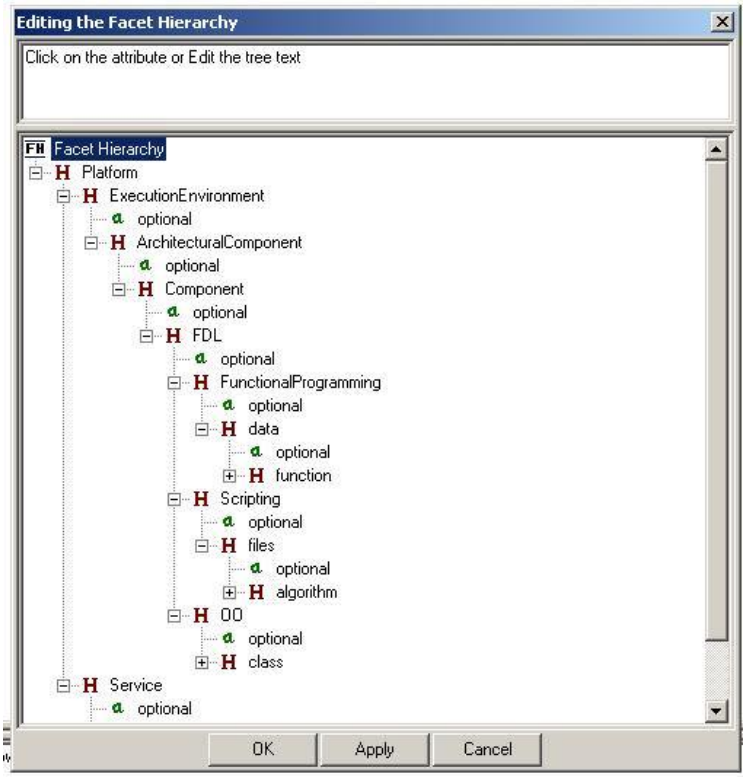


Figure 5.3: Facet Hierarchy

the optional attribute makes the FH level mandatory. Appendix C contains the DTD that describes the structure of the FH XML file.

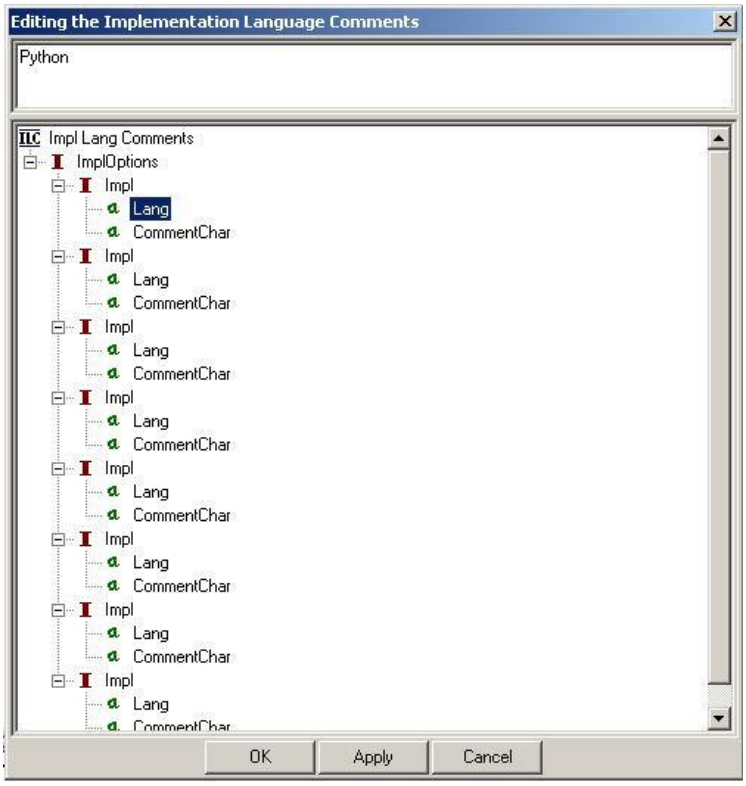


Figure 5.4: Implementation Language Comment Editor

Figure 5.4 shows the editor for the Implementation Language Comments. Implementation Languages with corresponding comments can be added, deleted and edited. The Editor translates graphical operations to changes in the underlying XML file. The list of Implementation Languages that a File can be characterised as is derived from the Implementation Language Comments. Hence if there is no definition for the C++ comment character, the FDE will not allow any C++ file to be added. Appendix C.2 contains the DTD that describes the structure of the Implementation Language Comment XML file with a sample XML file in appendix C.3.

The Help menu provides HTML format help for the FD and further information about the FDE.

5.3.2 Tree

Quick access to objects is enabled by the Tree. The Tree causes the Editing Window to load the default view of the object. The Tree also causes the ToolBar to load the tools related

to the selected object. Each object within the tree has an associated icon which helps to distinguish between objects. Table 5.1 lists the Icons and the objects that they belong to. Similar objects that are part of a Sub-Facet have a small Facet symbol in its top left hand corner. Table 5.1 will be referenced when examples are presented in Chapter 6.1 and 6.2

	Opened Facet
	Generic Definition
	Implementation Definition
	Contained Facet or SubFacet
	Package
	File
	Association
	Global Placeholder
	Global Variable Block
	Classes or Modules
	Methods or Functions
	Attributes
	Placeholders
	Variable blocks

Table 5.1: Tree Icons

5.3.3 ToolBar

Object manipulation is regulated by offering a limited set of tools in the ToolBar. The tools are grouped and associated with objects listed in table 5.1. The *Opened Facet Object* is only allowed to create an *Implementation Definition* of the Facet. The *Opened Facet* cannot be deleted or modified. The *Generic Definition* object is restricted to viewing its description, resources, context, data and underlying XML file. The *Implementation Definition* object expands on the *Generic Definition* object operations with its Technology view, deletion and creation of *Facets*, *Files*, *Packages*, *Associations*, *Global Placeholders* and *Global Variable Blocks*.

Facets created within an *Opened Facet* as part of a specific *Implementation Definition* are called *Contained Facets* or *SubFacets*. User interaction with the *Contained Facets* is restricted to viewing the Facet's description, resources, data, context, technology, underlying

xml file; deleting the *Contained Facet* and editing the *Contained Facet's Placeholders* and *Variable Block*s. Objects under the *Contained Facet* can only be edited.

The *File* object allows *Placeholders, Variable Blocks, attributes, methods* and *classes* to be created. We refer to objects that are contained in a *File* object as sub-file components. Text within the *File* can be added and removed. The *File* object can also be deleted.

Global Placeholders and *Global Variable Blocks* can be edited to contain child *Placeholders* and *Variable Blocks*, respectively.

Variable Blocks can contain *Placeholders*. Hence *Placeholders* can be created under *Variable Blocks*. *Classes, methods and attributes* abide by their structural relationship i.e. *classes* contain *methods* and *attributes* and *methods* can contain *attributes*.

5.3.4 Editable Windows

Editable windows are viewpoints that expose editable aspects of an editable object. Editable objects include Generic Definitions, Implementation Definitions, Global Placeholders, Placeholders, Global Variable Blocks, Variable Blocks, Files, Structural Components, Associations and SubFacets.

Generic Definition manipulation occurs from 5 viewpoints:

- Normal: presents the description field of the definition as described in subsection 4.2.1;
- XML: presents the XML file of the definition for the purposes of completeness;
- Resource: presents the set of resources used by the Facet particular to a definition;
- Data: presents the data description used by the Face particular to a definition; and
- Context: presents the context of the Facet particular to a definition.

In addition, a Technology window exists for editing the Technology profile of the Implementation definition.

Editable windows for the Global Placeholder and Global Variable allow the FD to choose Placeholders and Variable Blocks within the scope of the opened Facet. Files are edited using a *FileWindow*. The *FileWindow* provides management and position tracking of facet-specific and structural objects associated with a file.

Sub-Facets are edited using a dialog box consisting of a list of all facet-specific objects (contained objects under a Global are not duplicated in the list of editable facet-specific objects).

Structural components, Placeholder, Variable Blocks and Associations each have a specific window for editing their characteristics.

5.4 Design Patterns Used

Two major problems are encountered in the creation of the FDE. First, objects within the FDE need to access similar functionality. Secondly, GUI objects need simplified fully-meshed interactions. The Façade and Mediator design patterns are employed to solve these problems.

5.4.1 Façade

A recurring problem is a number of *client* classes needing to access a multitude of *task* methods across a number of classes. If the relationship between the *client* classes and *task* methods is hardwired, a labyrinth of interactions is created which results in poor flexibility and strong coupling of classes.

Figure 5.5 shows the Façade design pattern which is a high-level interface that hides structural detail of the *task* methods from the *client* classes [9]. *task* Methods are encapsulated within the Façade class. The *client* has a reference to an instance of a Façade class via which the *client* can use the functionality offered by the *task* methods.

The Façade allows a *client* to interact with a *task* method without having a reference to the task method's class object. Thus promoting weak coupling; overall flexibility; preventing a labyrinth of interactions and minimizing the number of class objects the *client* must interact with.

A Façade is usually implemented as a Singleton. The Singleton is a class which only has one instance within the application. The Façade is used by FDE classes for functionality such as the XMLExplorer, XML File Manipulation, Help File Launching and XML Node creation. Section 5.6 discusses the implementation of the Façade object within the FDE.

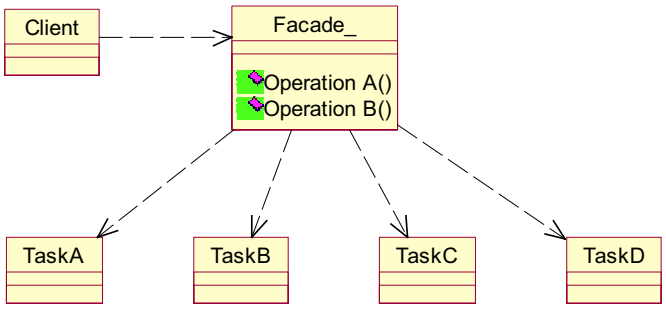


Figure 5.5: Façade Design Pattern [1]

5.4.2 Mediator

A recurring problem is a number of *colleague* classes needing to interact with each other in a fully meshed way. Keeping references of each *colleague* class becomes problematic to manage and creates strong coupling.

The Mediator design pattern encapsulates the *colleague* classes allowing them access to each other [9]. Figure 5.6 shows two *RealColleagues* that have access to each other via the *RealMediator*. The *RealMediator* keeps references to the *RealColleague* objects and allows the *RealColleagues* access to the referenced objects. In addition, the *RealMediator* must be designed such that the references to *RealColleagues* can be changed dynamically.

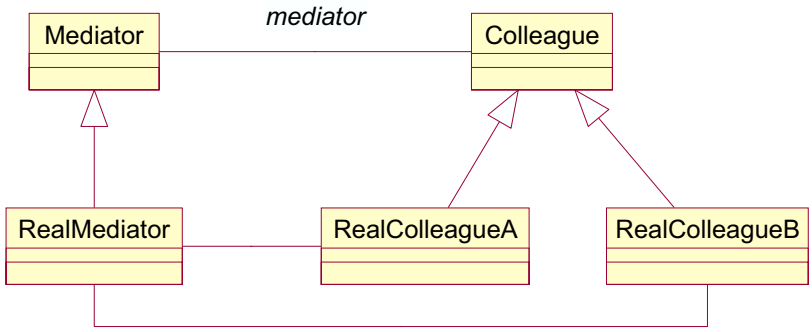


Figure 5.6: Mediator Design Pattern [1]

The Mediator is used to represent complex many-to-many class relationships as simple uni-directional associations. The centralised control of the mediator weakens coupling between classes, simplifies system design and introduces flexibility. Danger exists in allocating management of too many class relationships to a Mediator thereby increasing the complexity and rigidity of the Mediator and resulting in a break down of its flexibility and effectiveness.

Façade and Mediator design patterns are similar, although, the Mediator is considered more complex. The Mediator class is used to allow the key GUI components to interact in a fully meshed manner with each other and the FDE Façade. Section 5.6 discusses the implementation of the Mediator and Façade design patterns within the FDE.

5.5 Use Case Diagrams

The FDE is used by the Facet Developer (FD) hence, only one actor exists. Figure 5.7 defines what the FD uses the FDE for.

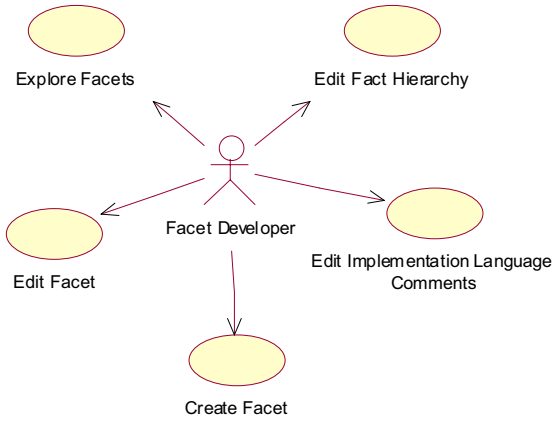


Figure 5.7: FDE Use Case

Prior to manipulating Facets, the FD will ensure that the FDE is configured correctly. Configuration of the FDE requires that the FD configure the Facet Hierarchy and Implementation Language Comments as he/she desires. Figures 5.4 and 5.3 show the editors for Implementation Language Comments and the Facet Hierarchy, respectively.

Once the FD has configured the FDE, Facets can be created and later edited. Creation of Facets entails a number of use cases such as creating a Sub-Facet, deleting a placeholder or editing a File. For brevity, the exhaustive list of use cases is omitted. Reaching a critical mass of Facets, the FD will want to browse the catalogue of Facets. The FD will be able to browse the Facet catalogue using the editor that is show in figure 5.2.

Section 5.6 solidifies the abstract use cases by introducing the FDE classes and their interactions.

5.6 Class Diagrams

The class diagrams in this section paint the structural detail of the FDE's design. The FDE has many classes, the most important of which are shown in figure 5.8. Figure 5.8 shows the core components of the GUI.

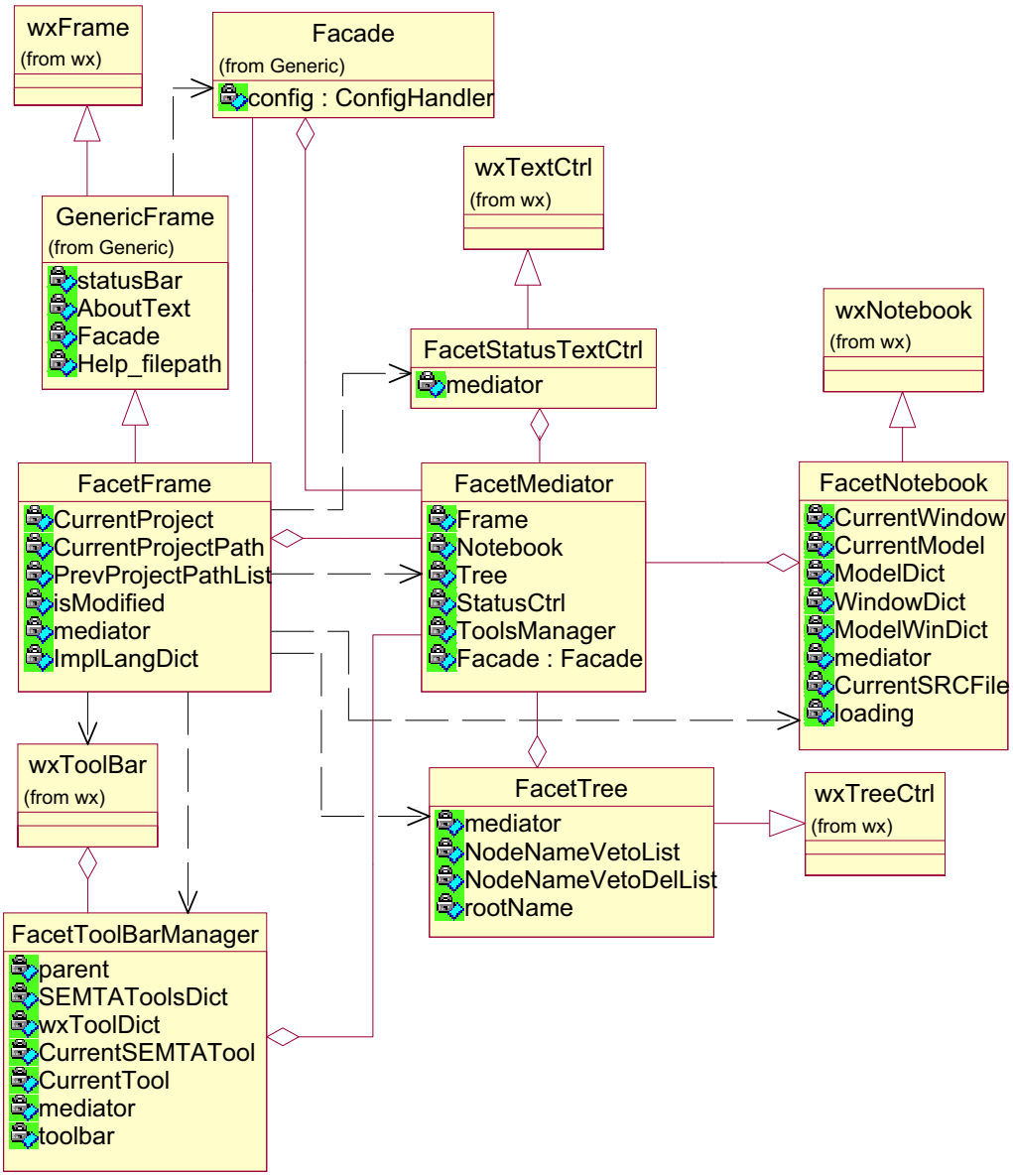


Figure 5.8: Facet IDE

Starting with the *FacetFrame*, we find that it inherits from the *GenericFrame* which in turn inherits from the wxPython class *wxFrame*. The *GenericFrame* introduces common functionality such as Help file launching and *Facade* initialisation. The *Facade* implements the Façade design pattern(sub-section 5.4.1) and is explained in sub-section 5.6.7.

FacetFrame is the Graphical parent of all the graphical classes as they all run as children of the *FacetFrame*. Implying that the *FacetFrame* initialises the other major GUI classes which includes the *FacetStatusTextCtrl*, *FacetTree*, *FacetToolBarManager* and *FacetNotebook*. *FacetFrame* also initialises the *FacetMediator* which implements the Mediator design pattern(sub-section 5.4.2). The major GUI classes, *FacetMediator* and *Facade* are discussed in their respective sub-sections. Appendix D illustrates many of the miscellaneous class diagrams that can be referred to grasp a better understanding of the class relationships.

5.6.1 FacetMediator class

All the graphical components and the *Facade* can be registered or deregistered with the *FacetMediator*. Allowing each class access to another class via the *FacetMediator*. In particular, all classes need access to functionality residing in the *Facade*. The *FacetMediator* results in a flexible meshing of the major components in the FDE.

5.6.2 FacetFrame class

The *FacetFrame* fulfills the following use cases, listed in section 5.5:

1. Create Facet: Facets are created in conjunction with the *NewFacetWizard* class,
2. Explore Facets: Facets are explored in conjunction with the *FacetExplorer* class,
3. Edit Facet Hierarchy: the FH is edited in conjunction with functionality that exists in the *Facade* class.
4. Edit Implementation Language Comments: also carried out in conjunction with functionality in the *Facade* class.

Figure 5.9 illustrates the class structure of the wizard used for creating new Facets. *NewFacetWizard* is derived from a wxPython *wxDialog* class. Creation of a new Facet involves four steps:

1. specifying the name of the Facet carried out by the *NameFacet* class,
2. choosing the Facet Hierarchy for the Facet carried out by the *ChooseFacetHierarchy* class,
3. specifying the information for the Generic definition carried out by the *GenericInformation* class and

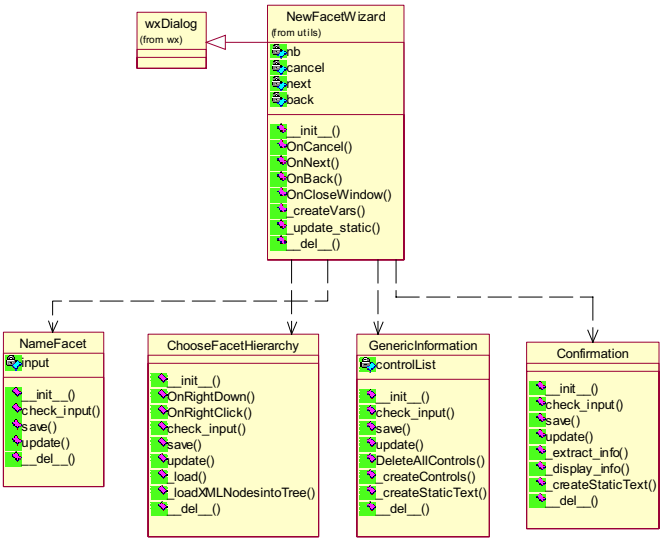


Figure 5.9: New Facet Wizard

4. confirmation of supplied information carried out by the *Confirmation* class.

The *NewFacetWizard* will verify all information including the absence of the Facet name in the *data* directory before creating the Facet. The *NewFacetWizard* creates the Facet as a directory containing a Generic definition. Hence fulfilling the use case of Facet Creation.

Facets are described within directories and not as individual files. Therefore, the *data* directory cannot be listed for files using the default *FileDialog*. The set of classes in figure 5.10 are developed to allow the FD to view and select Facets in the *data* directory. *FacetOpenDialog* lists all the Facets that a FD can open, using the *FacetOpenListPanel*. *FacetOpenListPanel* displays the Facet directories as a list of Facets. Similarly, the *FacetSaveDialog* produces a view of all Facets saving a Facet to the filesystem.

The FD can launch the *FacetExplorer* from the *FacetFrame*. The GUI of the *FacetExplorer* is shown in figure 5.2. The *FacetExplorer* inherits from the wxPython class *wxPanel*. *FacetExplorerNotebook* is initialised by the *FacetExplorer* and in turn initialises the necessary sub-windows to allow browsing of Facets.

Editing of the FH and Implementation Language Comment editors are essentially XML editors. Figure 5.12 shows both the editors inheriting from the *XMLEditorPanel*. The *XMLEditorPanel* has generic functionality to enable editing of XML files. The *FacetHierarchyEditor* implements *XMLEditorPanel* methods in a way that is specific for editing the FH. Similarly, the *ImplEditorPanel* is a specialisation of the *XMLEditorPanel* for the editing of implementation language comments.

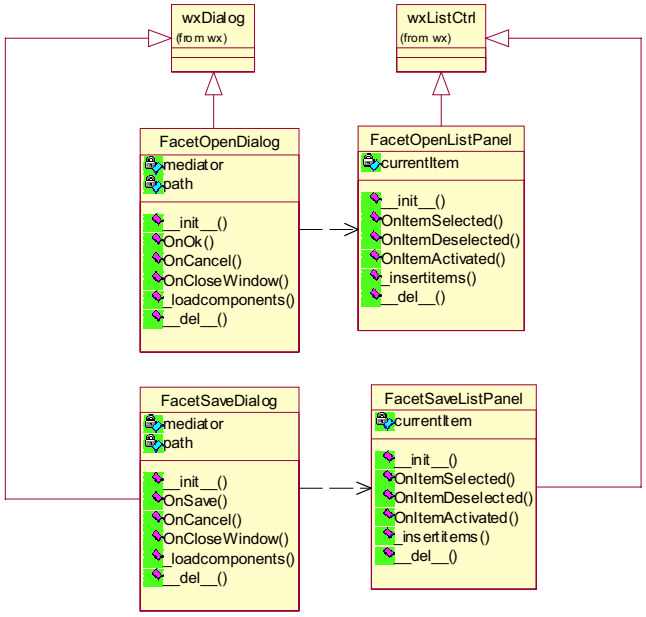


Figure 5.10: Facet Chooser

5.6.3 FacetStatusTextCtrl class

FacetStatusTextCtrl implements the Status GUI component. FacetStatusTextCtrl inherits from the wxPython class wxTextCtrl and is not directly linked to any of the use cases. Although, it is part of all use cases as far as producing feedback to the FD is concerned.

5.6.4 FacetToolBarManager class

The *FacetFrame* initializes the toolbar and is responsible for the low-level toolbar management. Higher-level management is done by the *FacetToolBarManager*. The *FacetToolBarManager* implements the ToolBar GUI component. The *FacetNotebook* accesses the *FacetToolBarManager* via the *FacetMediator* to load tools specific to the object that is selected in the *FacetTree*. *FacetToolbarManager* is part of fulfillment of the editing Facet use case.

5.6.5 FacetTree class

The *FacetTree* assists in Editing of Facets use case. A hierarchical structure of the objects within a Facet is displayed by the *FacetTree*. *wxTree* is the parent class for *FacetTree*.

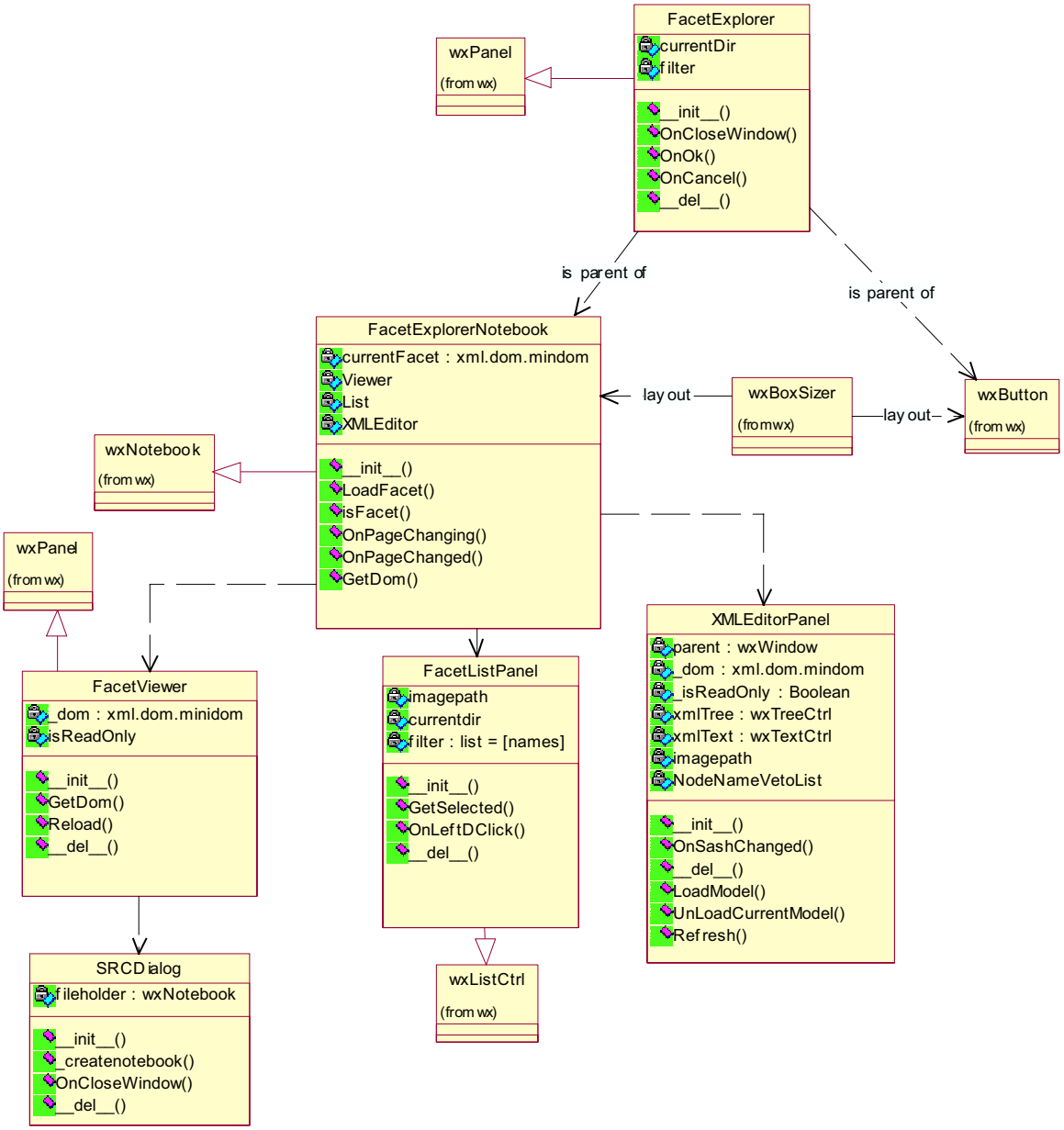


Figure 5.11: Facet Explorer

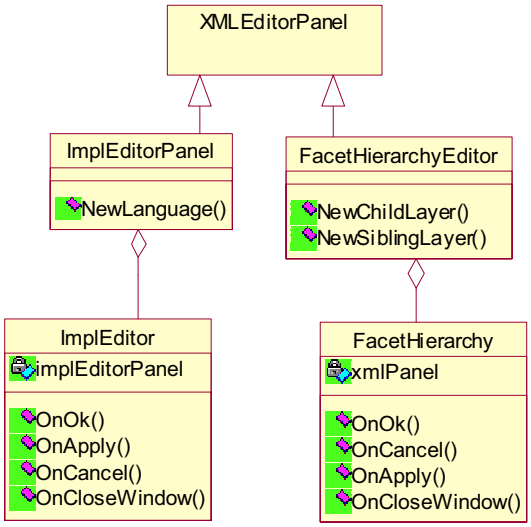


Figure 5.12: Facet Hierarchy and Implementation Language Comment Editors

5.6.6 FacetNotebook class

FacetNotebook is the key class for editing of Facets. The *FacetNotebook* takes advantage of functionality in the *Facade* class via the *FacetMediator*. Figure 5.13 illustrates the *FacetNotebook* class.

Many windows are initialised by the *FacetNotebook* and are used recursively to edit object such as Files and Facets. Recursive use is achieved by loading appropriate data models into specific windows. Hence, reducing processor overhead. Depending on the object, a specific data model is initialised and a unique reference assigned to the model for referencing within the FDE. All the windows inherit from a *WindowModel* class and all the data models inherit from a *DataModel* class. The class structure of child Windows and Datamodels are shown in Appendix D.

5.6.7 Facade class

The *Facade* class is used to manage dispersed functional interactions. Figure 5.14 shows the class structure of the *Facade*. The *Facade* offers the following functionality:

- About window: *AboutDlg* methods display an “about” message. The “about” message describes the applications, its purpose and those involved in its development.
- Extracting information from *.src* files: *SRCFile2Text* methods are used for this purpose.

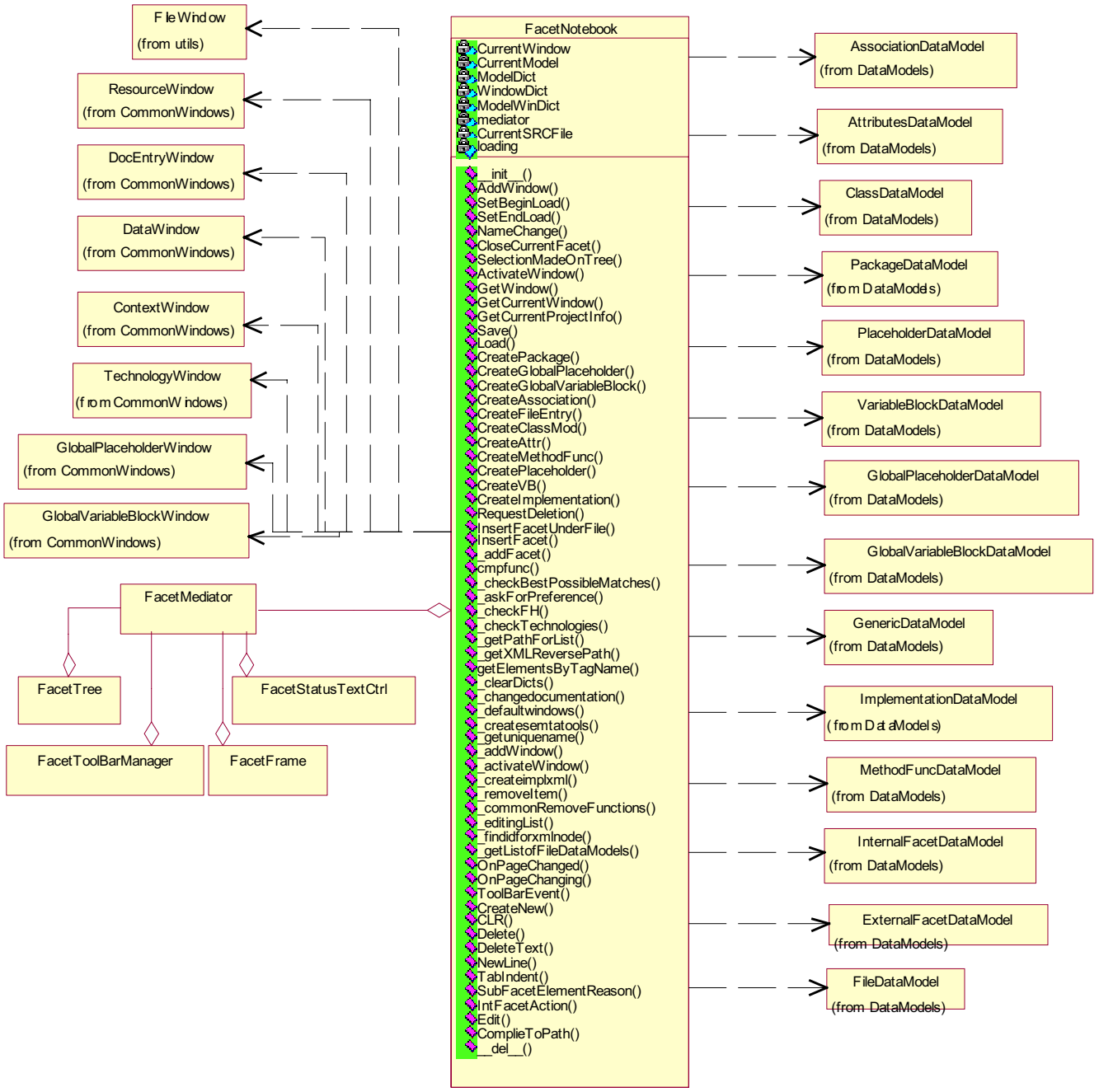


Figure 5.13: FacetNotebook

- Displaying HTML help information: *HTMLBrowser* is a *wxFrame* with a modified *wxHtmlWindow* that is used to display html files.
- Handling implementation language comments: *CommentHandler* methods are used by the *ImplLangEditor* to carry out changes to the XML file that holds the comments for the implementation languages.
- Managing a list of XPath strings: *TheXPath* methods allow XPath strings to be specified and executed on specified DOM objects.
- XML file input and output: *XMLHandler* methods are used specifically for input and output of XML files.
- Creating XML nodes: *CreateSEMTA* methods are used to create XML Nodes for XML files.

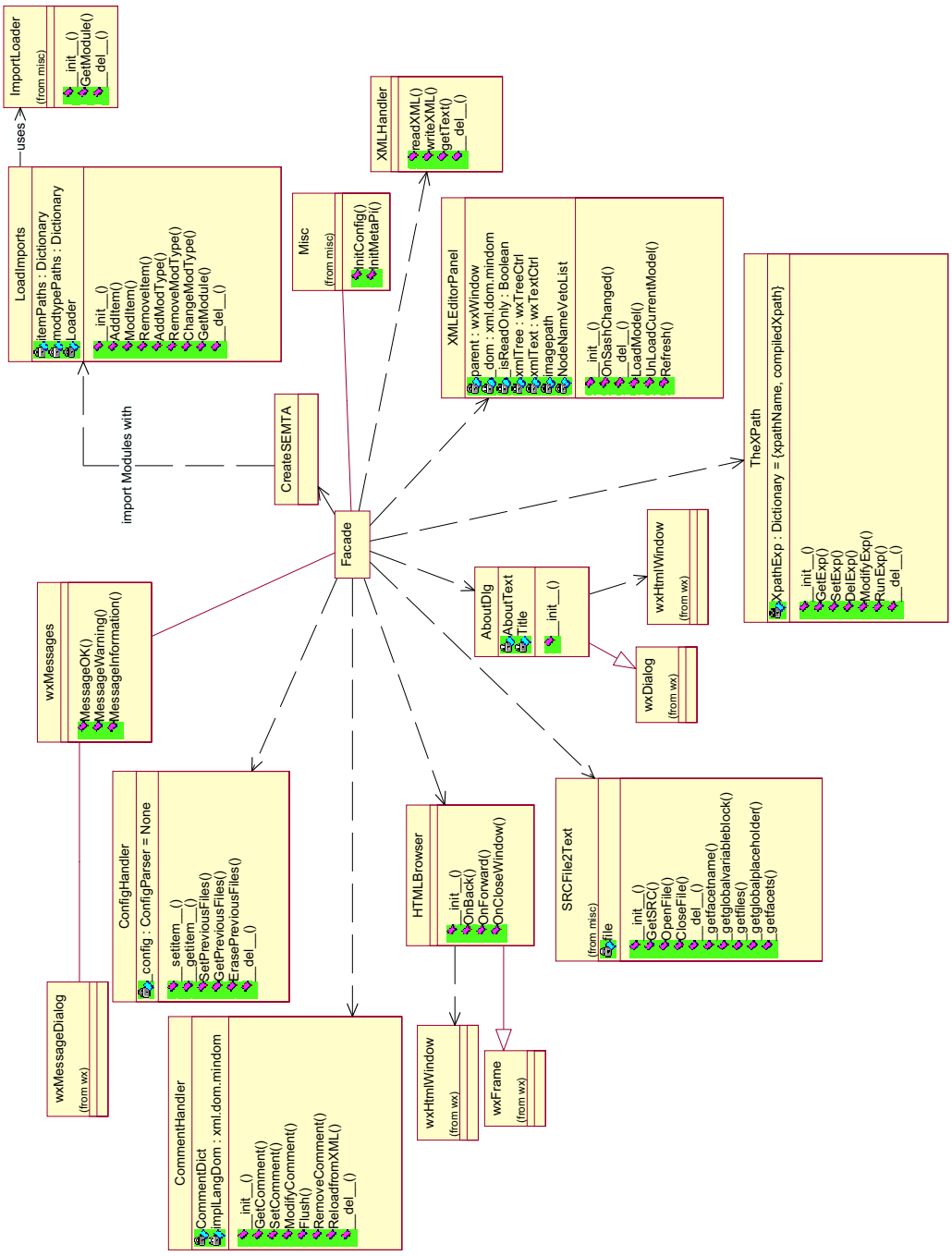


Figure 5.14: Facade

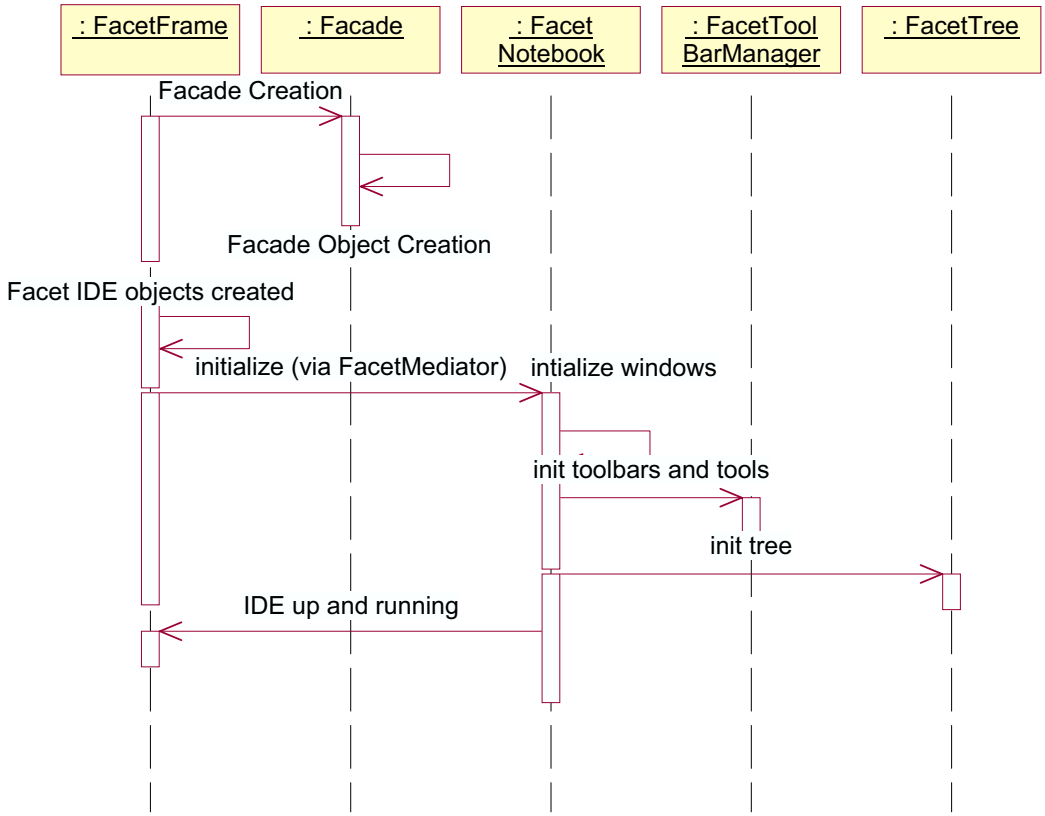


Figure 5.15: FDE Initialisation

5.7 Message Sequence Charts

The FDE has many dynamic processes which are not focused upon here. Rather, the following static processes are explained:

- FDE initialisation;
- Creating a Facet;
- Opening a Facet; and
- Saving a Facet.

5.7.1 FDE Initialisation

FDE initialization refers to starting the FDE once the FDE has been configured correctly. Figure 5.15 illustrates how the FDE is initialised. The *FacetFrame* initialises the Facade, which in turn, initializes its constituent classes. The *FacetFrame* registers the Facade with the *FacetMediator* and then moves on to create the GUI components.

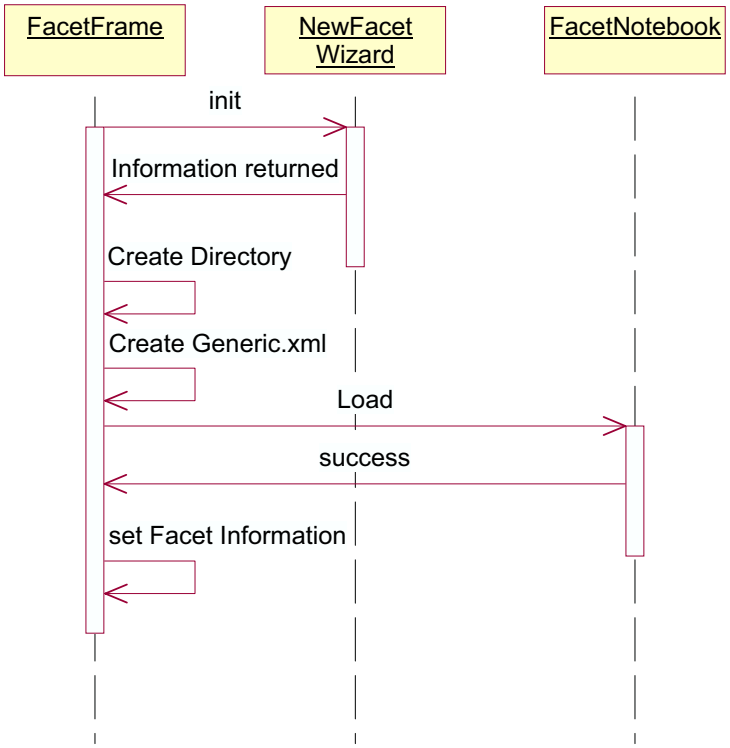


Figure 5.16: Creating a Facet

Each GUI component is registered with the *FacetMediator*. After the GUI components are created and registered, the *FacetFrame* invokes the *FacetNotebook* to initialise itself. Initialization of the *FacetNotebook* consists of initializing the *FacetToolBarManager* and the *FacetTree*.

The *FacetNotebook* initializes the *FacetToolBarManager* by creating multiple tools which are then grouped. The groups of tools are mapped to types of objects in the *FacetTree*. The *FacetTree* is initialised by loading a default tree structure, if no Facet is being loaded. Else, the Facet structure is loaded into the *FacetTree*. The FDE is now initialised and ready for use.

5.7.2 Creating a Facet

Figure 5.16 shows the process of creating a Facet once information has been gathered by the *NewFacetWizard*. The *NewFacetWizard* has a callback to the *FacetFrame* to create a Facet.

The *FacetFrame* creates the Facet directory based on the name of the Facet and then creates the Facet's Generic.xml file. All the information collected by the user is placed in the Generic.xml file.

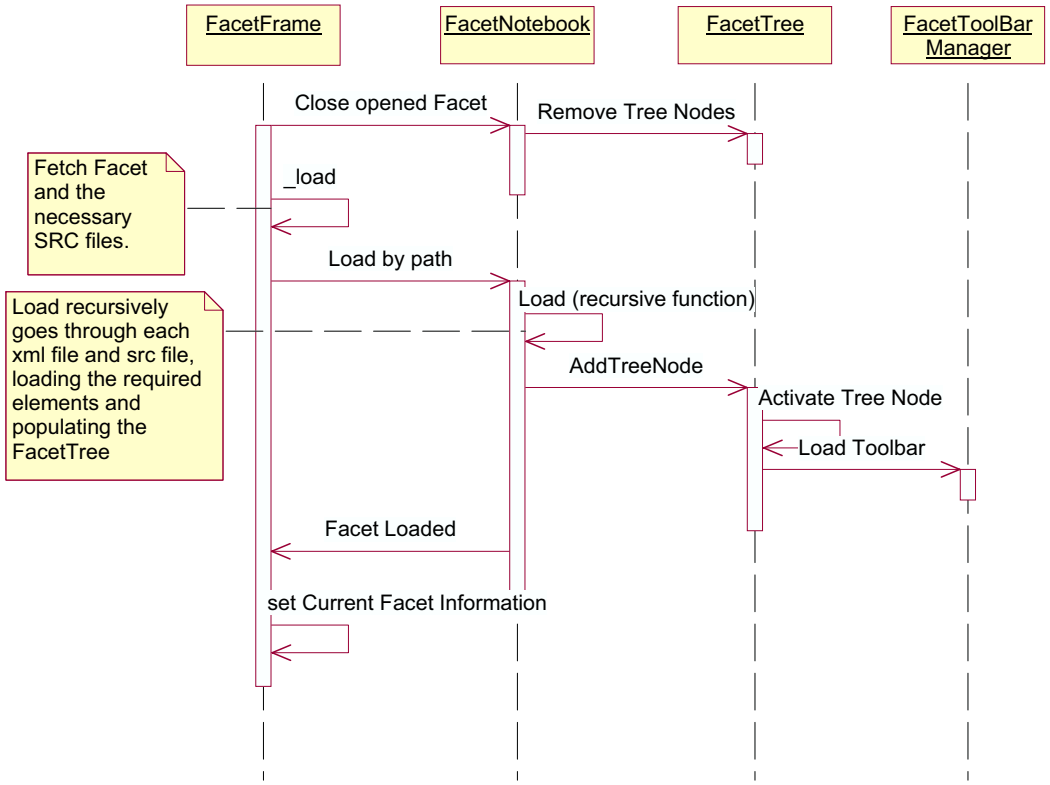


Figure 5.17: Opening a Facet

Once the Facet is created, the *FacetNotebook::Load* is called to load the created Facet into the GUI. If the Facet is successfully loaded, the *FacetFrame* sets the current Facet information such as Facet path and Facet name.

5.7.3 Opening a Facet

Figure 5.17 shows how a Facet is loaded into the FDE. Assuming that a Facet is already opened in the FDE, the opened Facet is first closed. If the opened Facet is not saved, it will be saved. The state of the GUI components are reset. Specifically, the list of objects and instantiated datamodels in the *FacetNotebook* are cleared followed by the removal of all nodes in the tree of the *FacetTree*.

The *FacetFrame* opens the Facet files for reading. The Facet Files include the Generic.xml, Implementation xml files and Implementation src files. *XMLHandler* is used via the *Facade* to read xml files. The SRC file is read using the *SRCFile2Text* via the *Facade*. The *FacetFrame* parses the loaded files to the *FacetNotebook* for loading of the Facet into the GUI.

The *FacetNotebook* applies a recursive function *Load* on the Facet file parsed to it. *Load*

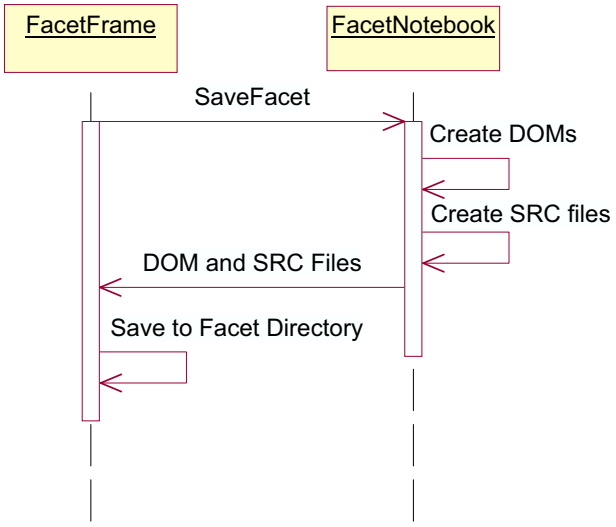


Figure 5.18: Saving a Facet

uses the *AddTreeNode* to add initialised objects into the *FacetTree* tree hierarchy. The *FacetTree* in turn activates the object forcing the *FacetToolBarManager* to associate an object type group with the object. Load completes by replying a successful or unsuccessful Facet loading to the *FacetFrame*.

If the Facet is successfully loaded, the *FacetFrame* sets the current Facet information such as Facet path and Facet name.

5.7.4 Saving a Facet

Figure 5.18 illustrates the process of saving a Facet. Saving of a Facet requires reaping information and packaging the information into XML format for storage onto the filesystem. *FacetFrame* calls the *SaveFacet* function in the *FacetNotebook*. Using the active list of objects within the *FacetNotebook*, the XML and SRC DOM objects are created.

FacetFrame receives the DOM objects which are then written to the filesystem using the *XMLHandler* via the *Facade*.

5.8 Chapter Summary

This chapter discusses the design considerations in creating the Facet Development Environment (FDE) thereafter describing the FDE and its make up.XML is chosen as the modelling language because of its ease of use; industry support by way of user groups and

programmes and the opportunity to integrate with UML via the XMI standard.

The DOM parser is chosen for its easy manipulation of XML documents. DOM also allows the XML document to be searched using XPath or XQuery.

Python is chosen as the implementation language for its ease of use, rapid prototyping features, portable GUI (wxPython), *glue language* capability and readily available supporting packages.

The filesystem is chosen because no additional software is required and rapid prototyping is encouraged.

The FDE description is made up of the user interface, use cases, class structure and message sequences of the FDE. The user interfaces is composed of six GUI components viz. Menu Bar, Toolbar, Tree, Documentation tab, Editing Windows tab and Status.

The FDE makes use of the Façade and Mediator design patterns. The Façade exposes distributed functional methods to clients. The Mediator simplifies many-to-many unidirectional associations. Both design patterns promote weak coupling and improved flexibility.

The use cases, class diagrams and message sequence charts detail the mechanics of the FDE. Chapter 6 attempts to solidify the concept of Facets through three examples. Collectively, the examples demonstrate the concepts embodied in Facets.

Chapter 6

Facet Examples

This chapter presents three examples which illustrate the Facet's objectives and concepts. The first example demonstrates a Facet whose functionality centres around providing a simple CORBA service. The *SimpleCORBAService* (section 6.1) illustrates the functioning of Facets, multi-granularity of Facets and their implementation-language or platform independence. Once the Facet concept has been solidified with the first example, the following examples focus on two key aspects: simultaneous co-existence of other reuse solutions with the Facet reuse solution(section 6.2) *and* containment of multiple programming paradigms within a single Facet(section 6.3).

6.1 Simple CORBA Service Facet

This section illustrates a simple CORBA Service as a Facet. The example steps through the Facet illustrating the multi-granularity and implementation language independence of Facets. The *SimpleCORBAService* Facet has definitions as follows:

- Problem: How to demonstrate a Simple CORBA Service with a single Object that possesses a minimal set of operations?;
- Intent: Demonstrate Client to Server operations;
- Forces: Requirement of a Simple CORBA server with a Single Object whose reference is written to File;
- Applicability: CORBA environments;
- Participants: CORBA Server, CORBA client, IDL Definition for client-server interaction; and
- Known Uses: Used to create CORBA services.

SimpleCORBAService uses the filesystem as a resource to read from and write to files. The design and business contexts are not well defined since the Facet is not specific to any business model or architecture. The deployment context points out the environment for a successful deployment of the CORBA Service. *SimpleCORBAService* is successfully deployed if the environment is set up correctly and a file exists which holds the servant's IOR. The operational context focuses on the requirements for the ongoing operation of the Facet. The data characteristics is defined by the definition in the IDL file.

Figure 6.1 shows a screen dump of the *SimpleCORBAService* Facet FDE Tree. We notice the Facet consisting of a Generic definition and two Implementation definitions, namely *Java* and *C++*. The presence of multiple implementations means that the facet can be applied where Java or C++ is the chosen implementation language. Therefore, the Facet is independent of the implementation language as long as Implementation definitions exist for the required implementation language. Similarly, the Facet can be argued to be platform independent. In this example, both Implementation definitions are structurally similar therefore we discuss only one, namely *Java*. Within the *Java* implementation there are a number of global Placeholders and global Variable Blocks. A File definition and three sub-Facets are also constituents of this implementation.

Figure 6.2 shows the structure of the *SimpleCORBAService* Facet in terms of contained Facets and highlights the associated Facet Hierarchy (FH) paths. *SimpleCORBAService* is made up of three Sub-Facets i.e. *CORBAServer*, *CORBAServerSideObject* and *CORBAClient*.

The *CORBAServer* facet is used to create a CORBA server that instantiates and manages a servant. *CORBAServerSideObject* creates the servant and allows the servant's logic/functionality to be defined. *CORBAClient* offers the functionality of a basic CORBA client that invokes operations on the CORBA servant. As an example of reuse, *CORBAServer* and *CORBAClient* utilise the same functionality offered by the *ORBCreation* Facet in different contexts.

Although not visible from figure 6.1, the global Placeholders and global Variable Blocks synchronise and aggregate sub-Facet Placeholders and Variable Blocks with the Facet's own. For instance (with reference to figure 6.1 and appendix E), the *SimpleCORBAService* global Placeholder called *CORBA Object Type* links with the following placeholders:

1. *Return Type* under the *Hello.idl* File within the *SimpleCORBAService* Facet (figure 6.1);
2. *Object Across Facets* global placeholder within the *CORBAServer* Facet (figure E.4). The global placeholder, in turn, links a number of placeholders within the *CORBAServer* Facet;

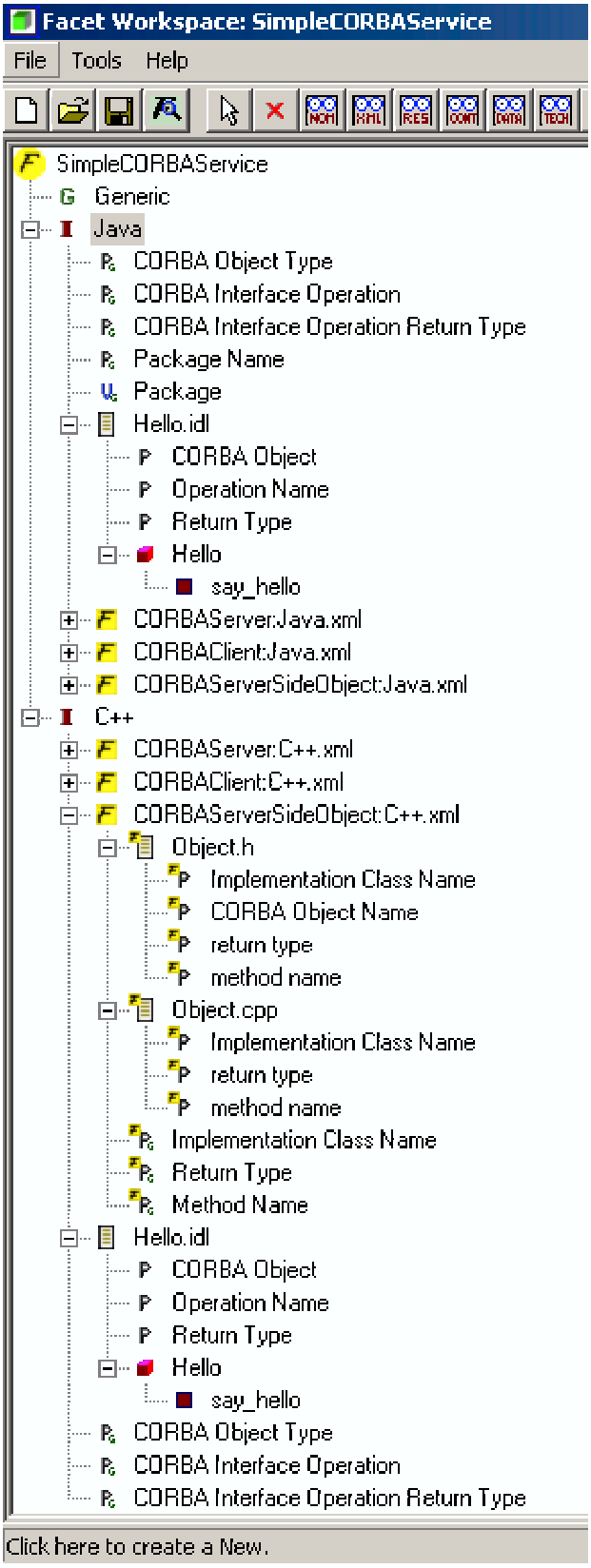


Figure 6.1: Simple CORBA Service Facet: FDE Tree Structure

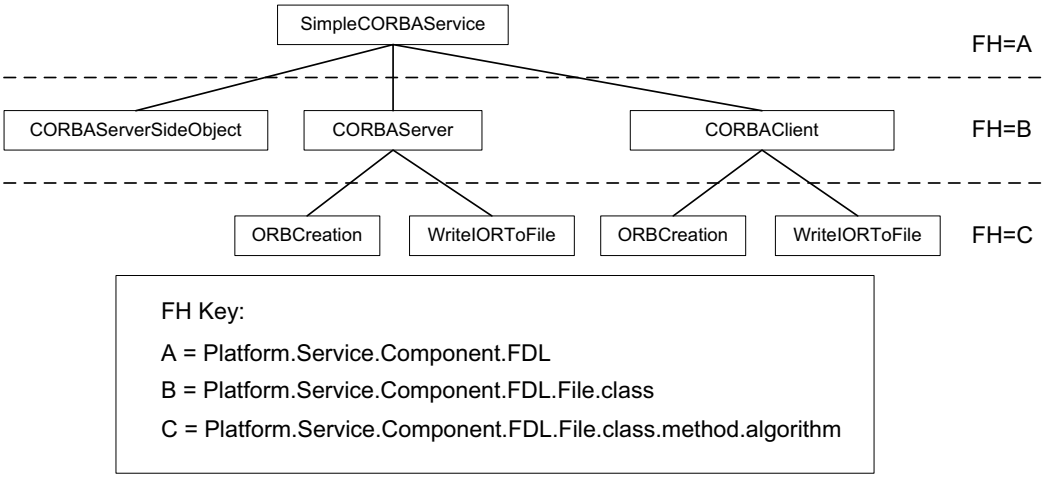


Figure 6.2: SimpleCORBAService Facet Structure

3. *CORBA Object Name* placeholder in the *CORBAServerSideObject* Facet (figure E.5); and

4. *CORBA Object Type* placeholder in the *CORBAClient* Facet (figure E.6).

Instead of the many Placeholders needing to be specified individually by the user, a single global Placeholders is specified without the user necessarily needing to know what underlying effects occur. Thus, the detail of the underlying Facets is abstracted from the user. The meta-π model that supports the implementation is essential as it makes the Facet data, resource and context agnostic. Appendix F contains the xml file that represents the meta-π model of the *SimpleCORBAService*, *Java* Implementation. Appendix E contains screen dumps for the Sub-Facet's FDE Tree.

We draw the readers attention to the Facet Hierarchy (FH) paths to the right of each level in figure 6.2. The FH paths should be correlated with the Facet Hierarchy structure shown in figure 4.3. The correlation will show that only Facets at a lower FH level are contained by Facets at a higher FH level. Thus, illustrating the **Rule of Containment** (Equation 3.12). The presence of multiple, ordered levels of containment is proof that the Facet concept is multi-granular.

6.2 Mediator Facet

This section presents a Facet which overlays another reuse solution. The *Mediator* Facet describes the *Mediator* design pattern with implementations. The *Mediator* Facet has definitions as follows:

- Problem: How to create a fully meshed association of classes?;
- Intent: Separate and encapsulate the interactions between a set of objects and allowing greater flexibility, control;
- Forces: An influx of hard-wired associations is undesirable. A dynamic approach is required;
- Applicability: Where a complex relationship exists between objects;
- Participants: Retrieve Object and Register Object; and
- Known Uses: Used to allow fully meshed interactions between GUI components in the FDE.

The *Mediator* Facet does not require any resources, instead, it is a resource to other Facets. The business, deployment and operational context of the Facet is undefined since it is not applied in an application. Applying a Mediator to link various objects, is the design context. The data flow that characterises the *Mediator* Facet, is the registering and retrieval of objects with the *Mediator* Facet.

Figure 6.3 shows the screen dump of the FDE Tree for the *Mediator* facet. The FDE Tree shows that the *Mediator* facet consists of a Generic definition and two Implementation definitions: *Python* and *Java*. Each implementation has a structural description, in terms of classes, methods and attributes, and is also described using Placeholders and Variable Blocks. Each implementation also makes use of the two Sub-Facets shown in figure 6.4.

The Mediator is a **design pattern** that is presented in section 5.4.2. We have defined the Mediator **design pattern** as a *Mediator* **Facet** by providing implementations for the Mediator **design pattern**. Hence, we have proven that other reuse solutions can exist simultaneously with the Facet reuse solution. This method can be applied using Enterprise Java Beans (EJBs) or MDA.

6.3 Web Server Tutorial Facet

This section presents a Facet that implements functionality using multiple programming paradigms such as Object-Oriented Programming, Functional Programming and Scripting. The *WebServerTutorial* Facet has definitions as follows:

- Problem: How to generate a dynamic server-side HTML page?;
- Intent: Demonstrate dynamic html generation;

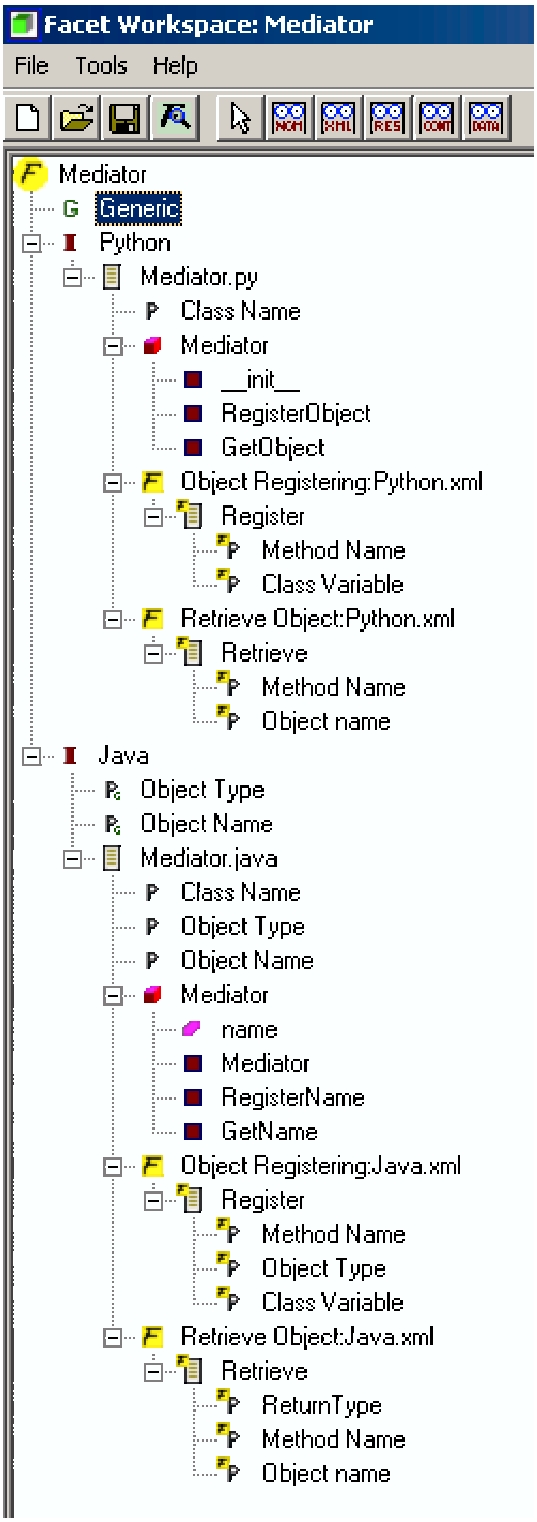


Figure 6.3: Mediator Facet: FDE Tree Structure

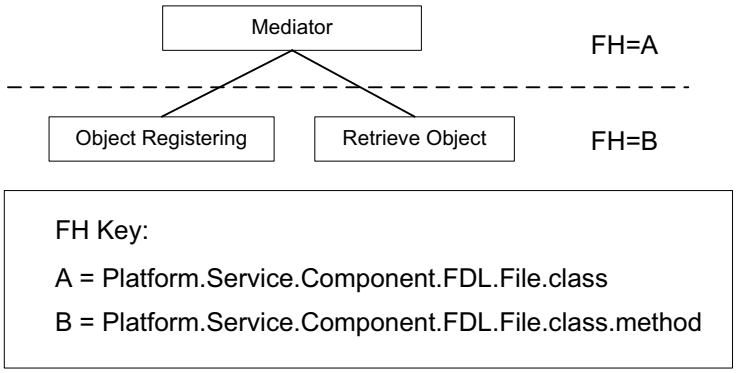


Figure 6.4: Mediator Facet Structure

- Forces: Users want to interact with a web site and cannot do so with static pages;
- Applicability: Where a web site's user interactiveness is to be improved;
- Participants: No other Facets; and
- Known Uses: Most web solutions.

Figure 6.5 shows that the *WebServerTutorial* Facet consists of a Generic Definition and three Implementation definitions, namely *JavaServlets*, *PHP* and *Perl*. The Implementations are significantly different from each other in that the *JavaServlets* implementation is object oriented programming, the *Perl* implementation is functional programming and the *PHP* implementation is script programming. The *WebServerTutorial* Facet caters for multiple programming paradigms therefore the Facet reuse solution can cater for multiple programming paradigms.

Since the implementations are quite different from each other, the necessary resources, context and data descriptions differ to a high degree. This example stresses the importance of having the reuse solution being agnostic of the resources, context and data. Without such information, the user will not be able to use the Facet effectively. *WebServerTutorial* does not contain any other Facets and its FH path is **Platform.Service.Component.File**

6.4 Chapter Summary

This chapter presents three examples to solidify the concept of Facets. The first example, *SimpleCORBAService*, demonstrates the functionality centric approach of Facets, multi-granularity of Facets and implementation-language or platform independence of Facets. *SimpleCORBAService* also showed how the facet-specific structural components were used and how the Facet Hierarchy enforces the Rule of Containment.

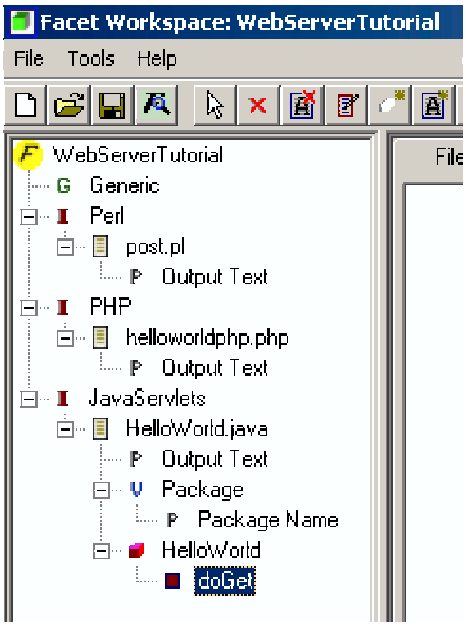


Figure 6.5: Web Server Tutorial Facet: FDE Tree Structure

The second example, *Mediator* Facet illustrated the Facet's capability to simultaneously accomodate other reuse solutions such as Design Patterns, EJBs and MDA.

The third example, *WebServerTutorial*, shows that Facets allow reuse to be enabled irrespective of the programming paradigm that is being used. *WebServerTutorial* facet contains three different programming paradigms.

Chapter 7 concludes the report and analyses the outcome of the work done with reference to the stated objectives. Recommendations for further work are also proposed.

Chapter 7

Conclusion

7.1 Discussion

Rapid Development Environments (RADs) are drivers for rapid service creation in the Next Generation Network. RADs are empowered by reuse solutions. Within the evolutionary Next Generation Network (NGN), multiple platforms or architectures that use multiple implementation languages exist. Reuse solutions that can be universally applied to the NGN are sought. The requirements for a NGN reuse solution are listed in Chapter 1.

By focusing the description mechanism of the reuse solution around functionality, the reuse solution is decoupled from the underlying implementation which results in a platform and implementation language independent reuse solution. By approaching the reuse problem from a functionality perspective development by functional composition, to achieve an application's global functional requirements, is promoted. Also, encouraging the effective use of the reuse solution since human beings are more comfortable with real functionality than with abstract objects.

Chapter 2 discusses contemporary reuse solutions with a view toward identifying worthwhile "best of breed" characteristics. Most reuse solutions punt reusability as components. Intelligent Network (IN) SIBs and Telecommunication Information Networking Architecture (TINA) RP-facets are functionality centric. IN SIBs promote rapid development by using a logical black-box approach i.e. the reusable component's external characteristics, such as inputs and outputs, are emphasized. TINA computational objects also use a black-box approach where the external input and output characteristics are defined by IDL definitions. Model Drive Architecture (MDA) and Reference Model for Open Distributed Processing (RM-ODP) introduce high-level reuse. MDA emphasizes the concept of developing Platform Independent Models (PIMs) which are later migrated to Platform Specific Models

(PSMs). MDA, however, is bound to the object-oriented paradigm. RM-ODP, as with Enterprise Java Beans (EJBs), emphasizes the business aspect of a reusable component. Design patterns possess a novel approach to explicitly deny solutions that have not been "tried and tested". In general, component reuse solutions are dedicated to particular implementation languages or platforms while higher level reuse solutions lack implementations of proposed solutions. The reuse solutions described here have diverse strengths and weakness. No single method provides the multi-granularity, platform-independence and implementation language-independence that is called for in section 1.4.

Chapter 3 introduces the Facet concept as a redefinition of the TINA RP-facet concept taking advantage of "best of breed" characteristics from chapter 2. Facets are defined mathematically to be platform and implementation language independent. The Facet Hierarchy categorizes the Facet's functionality and manages Facet containment. Facets are developed within a Facet Development Environment (FDE) by a Facet Developer (FD). The Facet Developer (FD) role is a job description that may exist in the Application Provider, Service Provider, 3^{rd} Party Service Provider or Vendor stakeholder domains. The FD verifies programmes from programmers, which are included into the catalogue of Facets by using the FDE. The FD interacts with the service creation teams to determine necessary Facets that can be used to develop the service.

Chapter 4 explains the key concepts that enable the Facet as a comprehensive approach to software reuse. Generic and Implementation definitions enable the decoupling of a reusable component specification from the implementation. The implementation can be specified for a specific implementation language or platform. The Generic and Implementation definitions are described by the meta-π model. Meta-π is a model at the M3 meta level consisting of six aspects. By utilising the six aspect of the meta-π model, resource, context and data agnostic features are imported into the reusable component. Placeholders and Variable Blocks are facet-specific structural components that facilitate code-level reuse by modifying source code directly.

Chapter 5 discusses the FDE. The necessary decisions for the development of the FDE are first discussed followed by user interface, use cases, class structure and static message sequences. XML is the chosen meta-language due to its favourable industry acceptance. The Document Object Model (DOM) parser is used for eXtensible Markup Language (XML) file processing. Although DOM is memory intensive (in comparison to the SAX parser), searching and easier XML manipulation make DOM the acceptable choice. The FDE is implemented using Python. Python facilitates rapid development together with a flexible platform independent GUI package. Facets are stored using the Filesystem due to unacceptable trade-offs with relational databases and a lack of freeware XQuery compliant databases. The FDE uses two important design patterns i.e. the Façade and Mediator design patterns.

The Façade wraps methods belonging to distributed classes in its own methods and offers them to clients. Advantages to the Façade are: client classes only need to know of the Façade class; weak coupling between classes is promoted and overall flexibility is improved. The Mediator class dynamically registers classes such that existing registered classes have access to each other. Thus providing a central point for managing complex interactions between classes. Advantages of the Mediator include weak coupling of classes; simplification of system design and improved overall flexibility.

Chapter 6 presents three examples that attempt to solidify the Facet concepts. Section 6.1 describes the *SimpleCORBAService* Facet, illustrating the basic Facet concepts of FH, containment, resources, data, context, placeholders and variable blocks. Section 6.2 illustrates the ability of Facets to accommodate other reuse solutions in its specification by using the *Mediator* Facet. And, section 6.3 illustrates the Facet's capability to address reuse across multiple programming paradigms using the *WebServerTutorial* facet.

7.2 Conclusion

We evaluate the Facet software reuse solution against the objectives laid out in section 1.4.

Facets are functionality centric. The Facet is described in the Generic definition as solving a particular problem. A solution to any problem realizes functionality. Hence the Facet is focused on functionality.

Facets are multi-granular. Each Facet can be categorised into a level of the Facet Hierarchy (FH). FH is a hierarchy of discrete levels that describe groups of functionality. The groups of functionality are arranged in such a manner that higher levels contain the lower levels. Hence the FH becomes a multi-level set of pigeon holes for functionality. Since Facets can be categorised into *one* of the multiple pigeon holes in the FH, the Facet is multi-granular.

Facets are platform-independent and implementation language-independent. By describing a Facet in terms of functionality, the Facet is detached from a particular implementation except, where the functionality is peculiar to a platform or implementation language. Also, the decomposition of a Facet into a Generic definition and any number of Implementation definitions implies that the FD can freely edit the Facet's Implementation definitions without changing its functional description. Hence the Facet can accommodate multiple platforms and/or implementation languages without changing its functional description.

Facets are resource, data and context agnostic. Meta-π has six aspects that address different issues. Resources, data and context of a Facet make up three of the six aspects.

The FD can input resource, data and context information into the FDE, which is formatted into the underlying meta-π model.

Facets enable platform migration. Platform migration is desirable when a service provider wishes to offer existing services on a different execution platform. If each Facet, contained by the service, has the capability to migrate to the new execution platform then platform migration is possible. The ability to migrate to a particular platform means that the Facet contains an Implementation definition for the target platform.

Facets abstract detail from the developer. The detail of a Facet is exposed via facet-specific structural components i.e. placeholders, variable blocks, global placeholders and global variable blocks. Facet-specific structural components are suppressed by the specifying values for each. Within a Facet, the FD specifies values for the Sub-Facet's facet-specific structural components. Hence the user of the Facet does not have to provide values for the Sub-Facet's facet-specific structural components. Thus abstracting the level of detail at the Sub-Facet level.

7.3 Recommendations for future work

This section discusses a few recommendations for future work. Meta-π begins with a novel approach to characterizing entities. The meta-π model should be expanded to become more robust and flexible in its ability to describe entities. Candidate expansion points are the resources, contexts and data aspects of the meta-π model.

Meta-π should also accommodate a mechanism to achieve cataloging of Facets. A catalogued list of facets will improve the probability of reuse from the perspective that the required Facets will be easier to find.

Accommodating various implementation languages forces various programming paradigms to be accommodated. Each programming paradigm has its own structural and interaction modelling languages. Together with the FDE, meta-π must be able to adequately describe an Implementation definition's structure and interactions using appropriate modelling languages.

For future work, it is proposed that a development environment be created that encompasses a process for using Facets to create services.

References

[1] D. V. Camp, “The object-oriented pattern digest.” Internet Web Site, 2002. http://patterndigest.com.

[2] D. X. Adamopoulos, G. Pavlou, and C. Papandreou, “Advanced service creation using distributed object technology,” *IEEE Communications Magazine*, vol. Vol 40, pp. 146–154, March 2002.

[3] I. Lovrek, ed., *Requirements for Service Creation Environments*, (Zagreb), 2nd International Workshop on Applied Formal Methods in System Design, June 1997.

[4] M. Chapman, S. Montessi, “Overall concepts and principle of TINA,” Tech. Rep. Version. 1.0, TINA Consortium, http://www.tinac.com/specifications/documents/overall.pdf, Feb 1995.

[5] R. Logewaran and C. L. Choo, “Issues on service creation in TINA,” in *Proceedings for TINA Workshop 2002*, pp. 9–12, Multimedia University, Cyberjaya, Malaysia, October 2002.

[6] J. Verhoosel, M. Wibbels, H. Betteram, and J.-L. Bakker, “Rapid service development on a TINA-based service deployment platform,” in *Proceedings of TINA ’99, Telecommunications Information Networking Architecture Conference*, (Turtle Bay Resort, Oahu, Hawaii, USA), 12-15 April 1999.

[7] TOSCA Group, “D11: User trial report on embedded methods and tools,” Tech. Rep. AC237/TTL/ALL/DS/R/0059/b1, TOSCA, Feb 1997. http://www.teltec.dcu.ie/tosca/publicdocs.html.

[8] Microsoft, “The component object model specification,” tech. rep., Microsoft, 1995. http://www.microsoft.com/com/resources/comdocs.asp.

[9] E. Gamma, R. Helm, J. Vlissides, and R. Johnson, *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley Longman Inc., October 1994.

[10] L. G. DeMichiel, L. U. Yalcinalp, and et al, “Enterprise JavaBeans specification, version 2.0,” tech. rep., Sun Microsystems, August 2001. http://java.sun.com/products/ejb/docs.html.

[11] B. Hayes, “The post-OOP paradigm,” *American Scientist*, vol. Vol. 91, pp. 106–110, March-April 2003.

[12] S. Ambler, “A realistic look at object-oriented reuse,” *Software Development Magazine*, pp. 12–20, January 1998.

[13] H. Hanrahan, “Intelligent networks.” Course Notes for ELEN509, University of the Witwatersrand, Johannesburg, May 2001.

[14] S. Trace, “Effective reuse,” tech. rep., Steel Trace, 2000. http://www.steeltrace.com/download/Whitepaper

[15] I. Jacobson, M. Gariss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley Professional, 1997.

[16] Object Management Group (OMG), “Meta object facility,” tech. rep., Object Management Group, March 2000. http://www.omg.org/mof/.

[17] A. Mili, S. F. O. Chmiel, R. Gottumkkala, and L. Zhang, “An integrated cost model for software reuse,” in *International Conference on Software Engineering (ICSE)*, (Limerick, Ireland), pp. 157–166, 2000.

[18] TINA Consortium, *Computational Modelling Concepts*, 17th May 1996. TINA Version 3.2.

[19] K.Raymond, “Reference Model of Open Distributed Processing (RM-ODP):Introduction,” tech. rep., CRC for Distributed Systems Technology, Centre for Information Technology Research, University of Queensland, 1995. http://gullfisk.agderikt. hia.no/kurs/ODS/litt/icodp95.pdf.

[20] ISO/IEC, “Reference model of open distributed processing,” Tech. Rep. JTC1.21.43, ISO, 1995.

[21] N. Mitra, “Introduction to XML web services,” in *International Conference on Intelligent Networks*, (Bordeaux, France), March 31-April 4 2003.

[22] D. Box, D. Ehnebuske, G. Kakivaya, and et al, “Simple object access protocol (SOAP) 1.1,” May 2000. http://www.w3.org/TR/SOAP/.

[23] Architecture Board MDA Drafting Team, “Model driven architecture: A technical perspective,” Tech. Rep. ab/2001-02-04, Object Management Group (OMG), Feb 2001.

[24] Object Management Group, “Common warehouse model (CWM),” tech. rep., OMG, 2002. http://www.omg.org/cwm.

[25] Object Management Group (OMG), “XML model interchange,” tech. rep., OMG, 2001. http://www.omg.org/xmi.

[26] OMG, “OMG unified modelling language specification,” Tech. Rep. Version 1.4, OMG, September 2001.

[27] O. Kath and et al, “Impacts of changes in enterprise software construction for telecommunications: Model driven architecture: Assessment of relevant technologies.,” tech. rep., Eurescom Project P1149, 2002.

[28] I. Schieferdecker, “TINA CTF - TINA conformance testing,” June 2000. http://www.iskp.uni-bonn.de/bibliothek/reports/GMD/2000/e-probl/TINACTF.pdf.

[29] TINA Consortium: TINA-CAT Workgroup, “Request for proposal: TINA conformance and testing framework,” Tech. Rep. Version 1.0: Approved and Released, TINA Consortium, http://www.tinac.com/compliance/TINAConfTestFrameworkRFP.pdf, July 1999.

[30] D. I. Schieferdecker and L. Mang, “TINA conformance testing framework,” tech. rep., GMD Fokus, May 2000.

[31] J. Soukup, J. O. Coplien, and D. C. Schmidt, *Pattern Languages of Program Design*, ch. 20: Implementing Patterns. Addison-Wesley, 1995.

[32] Technical Committee: JTC 1/SC 34, “Information processing – text and office systems – standard generalized markup language,” Tech. Rep. ICS 35.240.30, International Organization for Standardisation, 2001-08-13.

[33] World Wide Web Consortium (W3C), “Extensible markup language,” October 2000. http://www.w3.org/XML/.

[34] World Wide Web Consortium (W3C), “Document object model,” 2002. http://www.w3.org/DOM/.

[35] D. Brownell, “Simple API for XML project,” 2003. http://www.saxproject.org/.

[36] R. Bourret, “XML and databases,” January 2003. http://www.rpbourret.com/xml/XMLAndDatabases.htm.

Appendix A

Meta-π DTD

A.1 meta-π.dtd

```
<!ELEMENT SEMTA (metapi,Component)>

<!ELEMENT metapi (Creator, Definition, Data, Context,
Resources, Documentation, Extensions)>
<!ATTLIST metapi
        name    #PCDATA        #required
        type    #PCDATA        #required
>

<!ELEMENT Creator>
<!ATTLIST Creator
        name            #PCDATA        #required
         institute      #PCDATA        #required
        date            #PCDATA        #required
>

<!ELEMENT Definition (Description, Behaviour, Interactions , Logic)>

<!ELEMENT Description (Objectives)>
<!ELEMENT Objectives (#PCDATA)>

<!ELEMENT Behaviour (Rules, Policies, Limitations)>
<!ELEMENT Rules (#PCDATA)>
<!ELEMENT Policies (#PCDATA)>
<!ELEMENT Limitations (#PCDATA)>

<!ELEMENT Interactions (#PCDATA)>
```

```
<!ELEMENT Logic (#PCDATA)>

<!ELEMENT Data (Input, Algorithm, Output)>
<!ELEMENT Input (Items*)>
<!ELEMENT Algorithm (Items*)>
<!ELEMENT Output (Items*)>
<!ELEMENT Items (ValueDescription)>
<!ATTLIST Items
        source          #PCDATA        #required
        target          #PCDATA        #required
        type            #PCDATA        #required
>
<!ELEMENT ValueDescription (#PCDATA)>

<!ELEMENT Context (Business, Design, Operational, Deployment)>
<!ELEMENT Business (Environment*)>
<!ELEMENT Design (Environment*)>
<!ELEMENT Operational (Environment*)>
<!ELEMENT Deployment (Environment*)>
<!ELEMENT Environment (ValueDescription)>
<!ATTLIST Environment
        name    #PCDATA        #required
        type    #PCDATA        #required
        priority        (Default|Recommended|Minimum|Required) Default
>

<!ELEMENT Resources (Internal, Boundary, External)>
<!ELEMENT Internal (Resource*)>
<!ELEMENT Boundary (Resource*)>
<!ELEMENT External (Resource*)>
<!ELEMENT Resource (ValueDescription)>
<!ATTLIST Resource
        name    #PCDATA        #required
        type    #PCDATA        #required
        priority        (Default|Recommended|Minimum|Required) Default
>

<!ELEMENT Documentation (Entry*)>
<!ELEMENT Entry (DocEntry, AdditionalInfo)>
<!ATTLIST Entry
        time    #PCDATA        #required
        date    #PCDATA        #required
>
```

```
<!ELEMENT DocEntry (#PCDATA)>
<!ELEMENT AdditionalInfo (#PCDATA)>

<!ELEMENT Extensions (Children, Contains, Other)>
<!ELEMENT Children (Models)>
<!ELEMENT Models>
<!ATTLIST Models
        name    #PCDATA        #required
        path    #PCDATA        #required
        type    #PCDATA        #required
>
<!ELEMENT Files>
<!ATTLIST Files
        name    #PCDATA        #required
        path    #PCDATA        #required
>
<!ELEMENT Contains (Models*,Files*)>
<!ELEMENT Other (#PCDATA)>

<!ELEMENT Component (ComponentFiles, Create)>
<!ELEMENT ComponentFiles>
<!ATTLIST ComponentFiles
        formatter       #PCDATA        #implied
        transformer     #PCDATA        #implied
        evthandler      #PCDATA        #implied
>
<!ELEMENT Create (#PCDATA)>
```

Appendix B

Facet DTD

B.1 Facet.dtd

```
<!ELEMENT SEMTA (metapi)>

<!ELEMENT metapi (Creator, Definition, Data, Context,
Resources, Documentation, Extensions)>
<!ATTLIST metapi
        name    #PCDATA         #required
        implname        #PCDATA         #implied
        type    (Generic|Impl)          #required
>

<!ELEMENT Creator>
<!ATTLIST Creator
        name    #PCDATA         #required
        institute       #PCDATA         #required
        date    #PCDATA         #required
>

<!ELEMENT Definition (Description, Behaviour,
Interactions , Logic)>

<!ELEMENT Description (Problem, Intent, Forces,
Applicability , Participants , KnownUses, Structure,
FacetHierarchy , Technology?)>

<!ELEMENT Problem (#PCDATA)>
<!ELEMENT Intent (#PCDATA)>
<!ELEMENT Forces (#PCDATA)>
<!ELEMENT Applicability (#PCDATA)>
```

```
<!ELEMENT Participants (#PCDATA)>
<!ELEMENT KnownUses (#PCDATA)>
<!ELEMENT Structure (gp, gvb, File*, Package*, Facet*,
Association* )>
<!ELEMENT gp (gpinstance*)>
<!ATTLIST gp
        name    #PCDATA         #required
        value   #PCDATA         #implied
>
<!ELEMENT gpinstance>
<!ATTLIST gpinstance
        name    #PCDATA         #required
>

<!ELEMENT gvb (gvbinstance*)>
<!ATTLIST gvb
        name    #PCDATA         #required
        value   (Y|N)           #implied
>
<!ELEMENT gvbinstance>
<!ATTLIST gvbinstance
        name    #PCDATA         #required
>

<!ELEMENT File (variableblock*, placeholder*,
Facet*, classmod*, methodfunc*,  attributes *)>
<!ATTLIST File
        name    #PCDATA         #required
        implementationLanguage  #PCDATA #required
>

<!ELEMENT variableblock (vbinstance*)>
<!ATTLIST variableblock
        name    #PCDATA         #required
        description     #PCDATA #required
        value   #PCDATA         #required
>

<!ELEMENT vbinstance>
<!ATTLIST vbinstance
        value   (Y|N)           #implied
        from    #PCDATA         #required
        to              #PCDATA         #required
```

```
>

<!ELEMENT placeholder (pinstance*)>
<!ATTLIST placeholder
        name    #PCDATA         #required
        description     #PCDATA #required
        value   #PCDATA         #required
>

<!ELEMENT pinstance>
<!ATTLIST pinstance
        value   #PCDATA         #implied
        from    #PCDATA         #required
        to              #PCDATA         #required
>

<!ELEMENT Facet (FacetHierarchy, settings*)
<!ATTLIST Facet
        name    #PCDATA         #required
        type    ( ext | int )           #required
        from    #PCDATA         #required
        to              #PCDATA         #required
>

<!ELEMENT settings>
<!ATTLIST settings
        name    #PCDATA         #required
        value   #PCDATA         #implied
        from    #PCDATA         #required
        to              #PCDATA         #required
>
<!ELEMENT classmod (name, type, expcontrol,
stereotype , attribute *, methodfunc*)
<!ATTLIST classmod
        from    #PCDATA         #required
        to              #PCDATA         #required
>
<!ELEMENT name>
<!ATTLIST name
        value   #PCDATA         #required
        from    #PCDATA         #required
        to              #PCDATA         #required
>
<!ELEMENT type>
```

```
<!ATTLIST type
        value   #PCDATA        #required
        from    #PCDATA        #required
        to              #PCDATA        #required
>

<!ELEMENT expcontrol>
<!ATTLIST expcontrol
        value   #PCDATA        #required
        from    #PCDATA        #required
        to              #PCDATA        #required
>

<!ELEMENT stereotype>
<!ATTLIST stereotype
        value   #PCDATA        #required
        from    #PCDATA        #required
        to              #PCDATA        #required
>

<!ELEMENT attribute (name, type,  stereotype ,
 initvalue ,  expcontrol )>
<!ATTLIST attribute
        from    #PCDATA        #required
        to              #PCDATA        #required
>

<!ELEMENT methodfunc (name, return, stereotype,
expcontrol ,  args*)>
<!ATTLIST methodfunc
        from    #PCDATA        #required
        to              #PCDATA        #required
>

<!ELEMENT return>
<!ATTLIST return
        value   #PCDATA        #required
        from    #PCDATA        #required
        to              #PCDATA        #required
>

<!ELEMENT args (name, type)>

<!ELEMENT Package (File*, Package*)>
```

```
<!ATTLIST
        name    #PCDATA         #required
>

<!ELEMENT Association>
<!ATTLIST Association
        name            #PCDATA         #required
        sourcename      #PCDATA         #required
        destname        #PCDATA         #required
        sourcemult      #PCDATA         #implied
        destmult        #PCDATA         #implied
        type            #PCDATA         #required
>

<!ELEMENT FacetHierarchy (#PCDATA)>
<!ELEMENT Technology (t_item*)>

<!ELEMENT t_item>
<!ATTLIST t_item
        name    #PCDATA         #required
>

<!ELEMENT Behaviour (Rules, Policies, Limitations )>
<!ELEMENT Rules (#PCDATA)>
<!ELEMENT Policies (#PCDATA)>
<!ELEMENT Limitations (#PCDATA)>

<!ELEMENT Interactions (Relations, Collaborations )>
<!ELEMENT Relations (#PCDATA)>
<!ELEMENT Collaborations (#PCDATA)>

<!ELEMENT Logic (#PCDATA)>

<!ELEMENT Data (Input, Algorithm, Output)>
<!ELEMENT Input (Items*)>
<!ELEMENT Algorithm (Items*)>
<!ELEMENT Output (Items*)>
<!ELEMENT Items (ValueDescription)>
<!ATTLIST Items
        source          #PCDATA         #required
        target          #PCDATA         #required
        type            #PCDATA         #required
>
<!ELEMENT ValueDescription (#PCDATA)>
```

```
<!ELEMENT Context (Business, Design, Operational, Deployment)>
<!ELEMENT Business (Environment*)>
<!ELEMENT Design (Environment*)>
<!ELEMENT Operational (Environment*)>
<!ELEMENT Deployment (Environment*)>
<!ELEMENT Environment (ValueDescription)>
<!ATTLIST Environment
        name    #PCDATA         #required
        type    #PCDATA         #required
        priority            (Default|Recommended|Minimum|Required) Default
>

<!ELEMENT Resources (Internal, Boundary, External)>
<!ELEMENT Internal (Resource*)>
<!ELEMENT Boundary (Resource*)>
<!ELEMENT External (Resource*)>
<!ELEMENT Resource (ValueDescription)>
<!ATTLIST Resource
        name    #PCDATA         #required
        type    #PCDATA         #required
        priority            (Default|Recommended|Minimum|Required) Default
>

<!ELEMENT Documentation (Entry*)>
<!ELEMENT Entry (DocEntry, AdditionalInfo)>
<!ATTLIST Entry
        time    #PCDATA         #required
        date    #PCDATA         #required
>
<!ELEMENT DocEntry (#PCDATA)>
<!ELEMENT AdditionalInfo (#PCDATA)>

<!ELEMENT Extensions (Children, Contains, Other)>
<!ELEMENT Children (Models)>
<!ELEMENT Models>
<!ATTLIST Models
        name    #PCDATA         #required
        path    #PCDATA         #required
        type    #PCDATA         #required
>
<!ELEMENT Files>
<!ATTLIST Files
        name    #PCDATA         #required
```

```
        path    #PCDATA         #required
>
<!ELEMENT Contains (Models*,Files*)>
<!ELEMENT Other (ExamplesOfUse*)>
<!ELEMENT ExamplesOfUSe (#PCDATA)>
```

B.2 FacetSource.dtd

```
<!ELEMENT FacetSource (Technology+, globalvariableblock*,
 globalplaceholder *, facets *, files ?)>
<!ELEMENT Technology (CDATA)>
<!ATTLIST FacetSource
        name    #PCDATA         #REQUIRED
        impl    #PCDATA         #REQUIRED
        fh              #PCDATA         #REQUIRED
>
<!ELEMENT link>
<!ATTLIST link
        name    #PCDATA #REQUIRED
        path    #PCDATA #REQUIRED
>

<!ELEMENT globalvariableblock (link+)>
<!ATTLIST globalvariableblock
        name    #PCDATA #REQUIRED
        value   (Y|N)   #REQUIRED
>

<!ELEMENT globalplaceholder (link+)>
<!ATTLIST globalplaceholder
        name    #PCDATA #REQUIRED
        value   #PCDATA #REQUIRED
>

<!ELEMENT facets (description, settings *)>
<!ELEMENT description>
<!ATTLIST description
        name    #PCDATA #REQUIRED
        fh              #PCDATA #REQUIRED
>
<!ELEMENT settings>
<!ATTLIST settings
        name    #PCDATA #REQUIRED
        value   #PCDATA #REQUIRED
```

>

<!ELEMENT files (CDATA)>
<!ATTLIST files
	filename	#PCDATA	#REQUIRED
>

Appendix C

Facet Hierarchy and Implementation Language

C.1 Sample Facet Hierarchy XML file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--This xml file will maintain the state of the hierarchical
structure which describes facet containment. xml is ideal for
this due to its naturally hierarchical description. The
optional attr serves to provide a mechanism to allow certain
levels of the hierarchy to be skipped. One such situation is
the FDL level. Not all software created will have an FDL
definition . where FDL refers to XML, IDL, etc.-->
<SEMTA>
    <Platform optional="No">
        <Architecture optional="No">
            <Component optional="No">
                <FDL optional="Yes">
                    <OO optional="No">
                        <Classes optional="No">
                            <Methods optional="No">
                                <Algorithms optional="
                                No"/>
                            </Methods>
                        </Classes>
                    </OO>
                    <FunctionalProgramming optional="No">
                        <DataStructures optional="No">
                            <Functions optional="No">
                                <Algorithms optional="
                                No"/>
```

```
                                        </Functions>
                                    </DataStructures>
                                </FunctionalProgramming>
                                <Scripting optional="No">
                                    <Files optional="No">
                                        <Algorithms optional="No"/>
                                    </Files>
                                </Scripting>
                            </FDL>
                        </Component>
                    </Architecture>
        </Platform>
</SEMTA>
```

C.2 DTD for Implementation Language Comments

```
<!ELEMENT ImplOptions (*Impl)>
<!ELEMENT Impl>
<ATTLIST Impl
        CommentChar             #PCDATA         #required
>
```

C.3 Sample Implementation Language Comments XML file

```
<?xml version="1.0" encoding="iso-8859-1"?>
<ImplOptions>
        <Impl CommentChar="##" Lang="Python"/>
        <Impl CommentChar="//" Lang="IDL"/>
        <Impl CommentChar="//" Lang="Java"/>
        <Impl CommentChar="//" Lang="C++"/>
        <Impl CommentChar="#" Lang="PHP"/>
        <Impl CommentChar="#" Lang="Perl"/>
        <Impl CommentChar="" Lang="None"/>
        <Impl CommentChar="&lt;!--" Lang="HTML"/>
</ImplOptions>
```

Appendix D

Additional UML Diagrams

This Appendix adds to the UML diagrams from section 5.6.

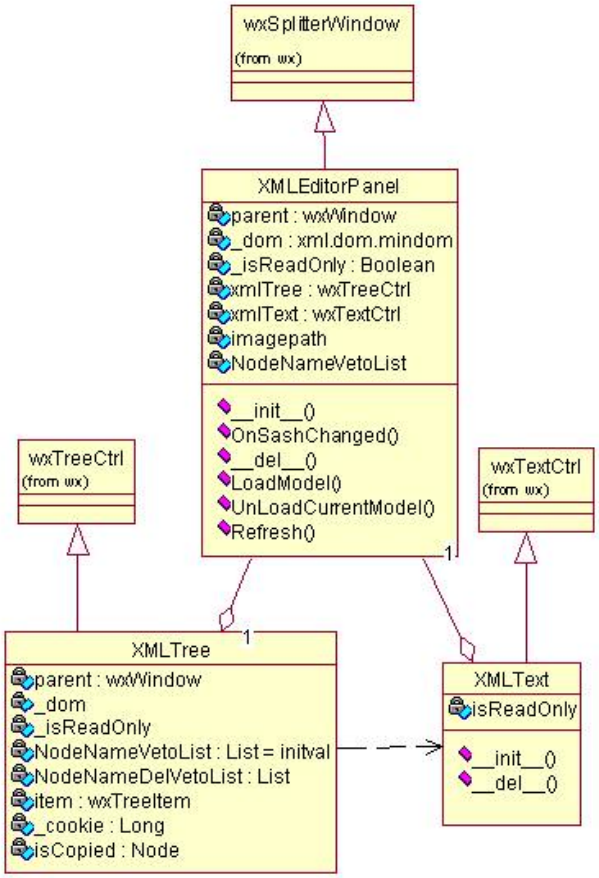


Figure D.1: XML Explorer

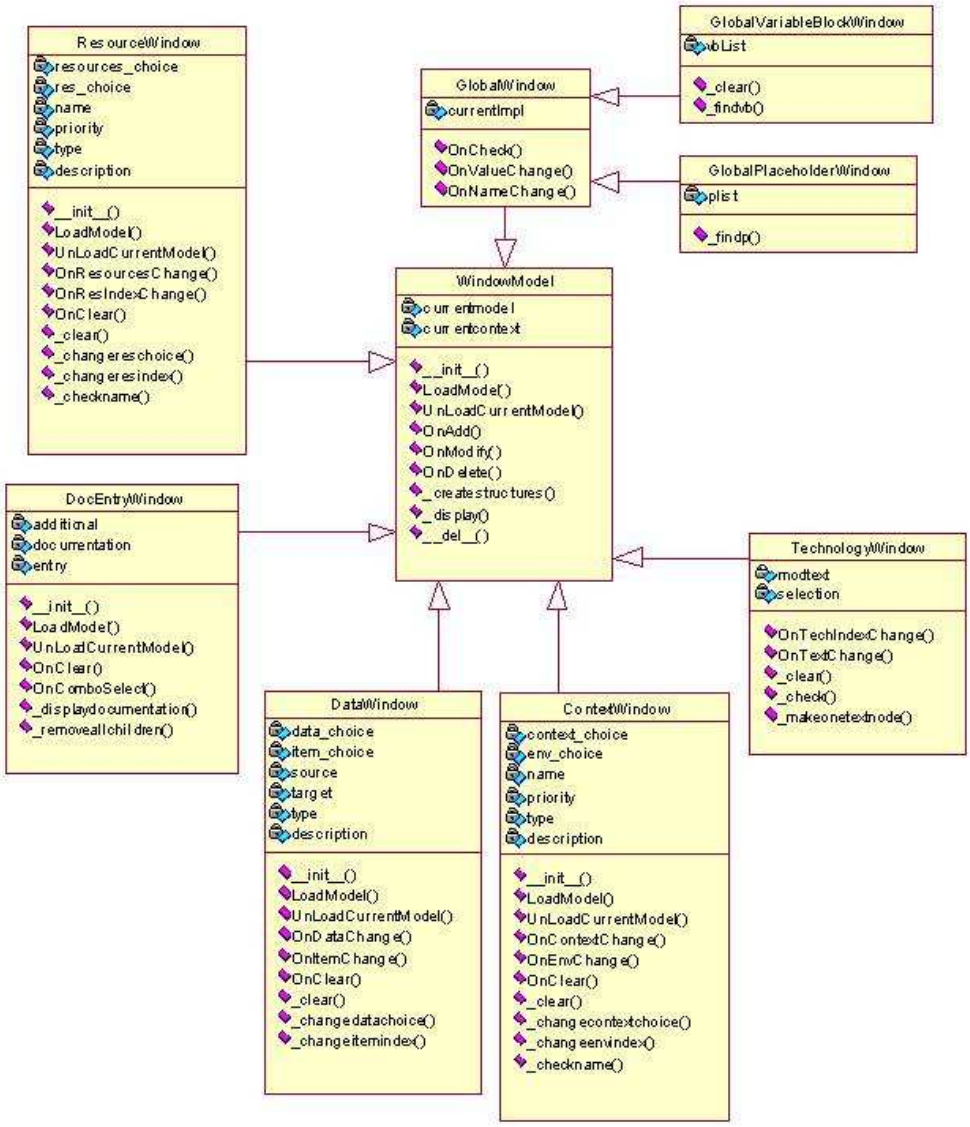


Figure D.2: Common Windows

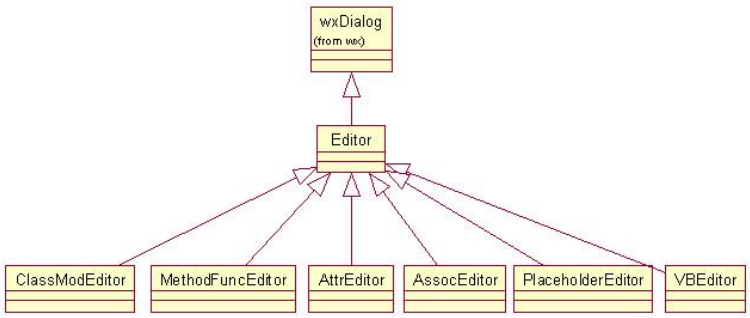


Figure D.3: Editor

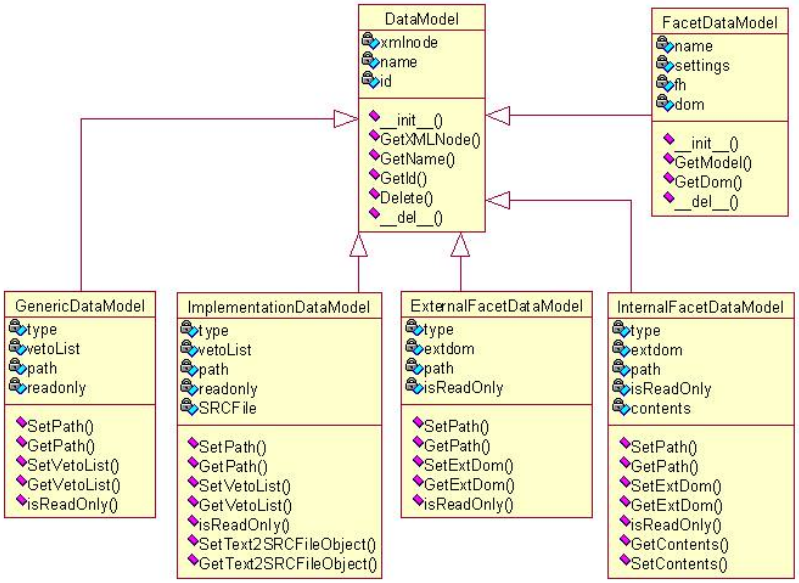


Figure D.4: Facet Data Models

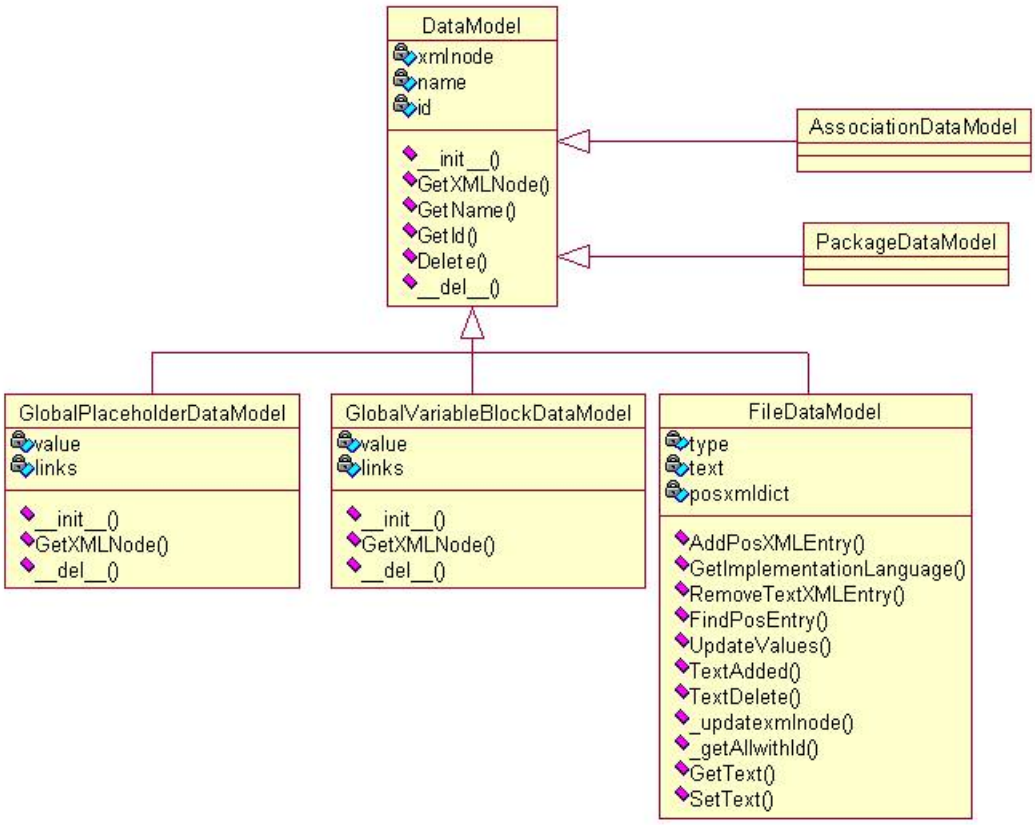


Figure D.5: Sub-Facet Data Models

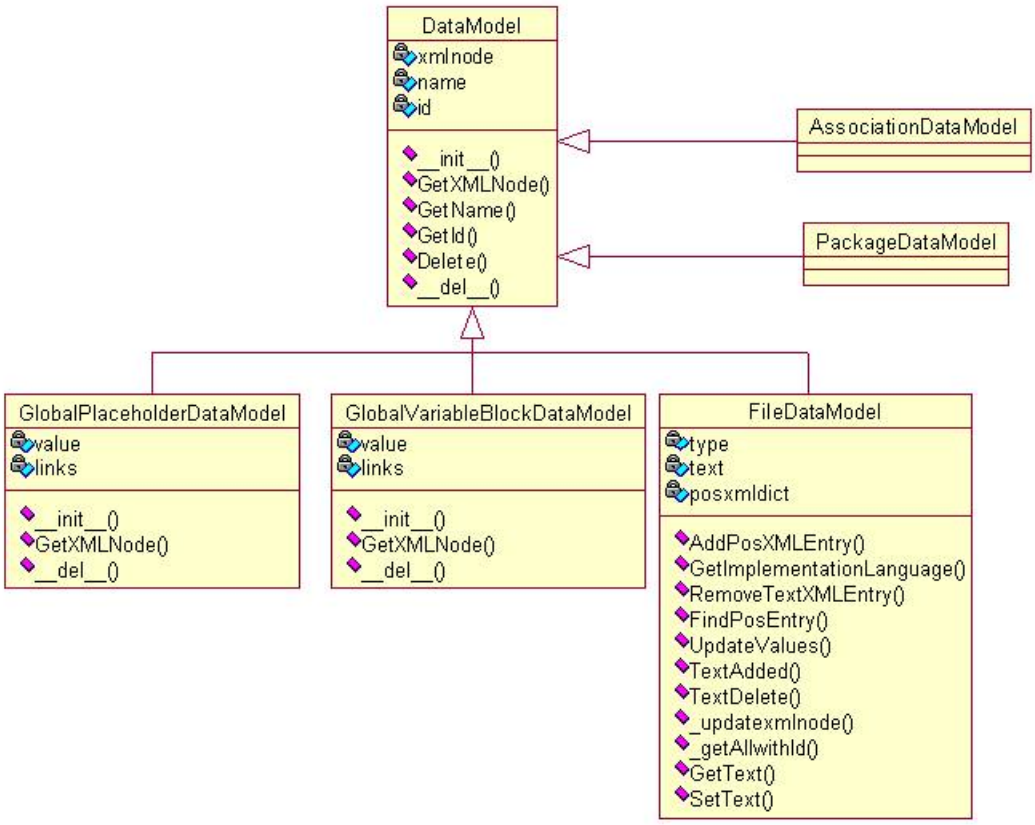


Figure D.6: Sub-File Data Models

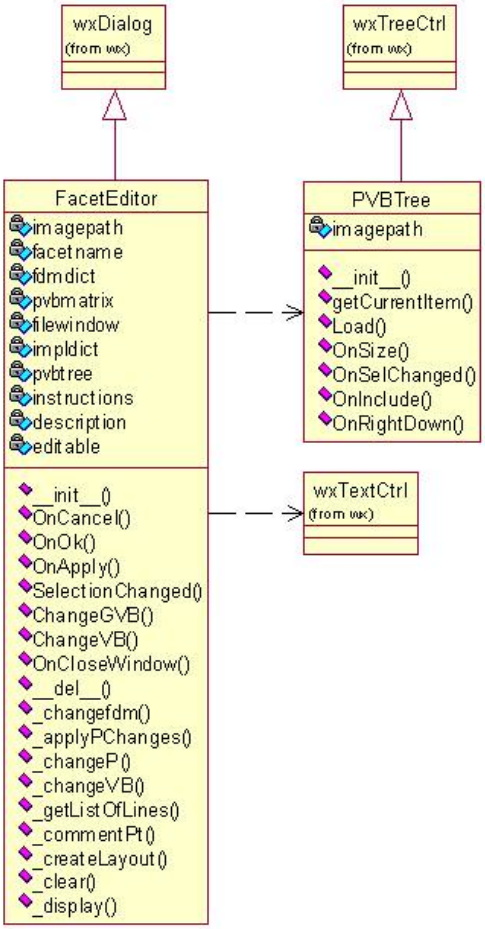


Figure D.7: Facet Editor

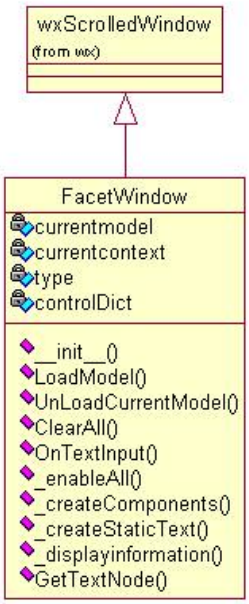


Figure D.8: Facet Window

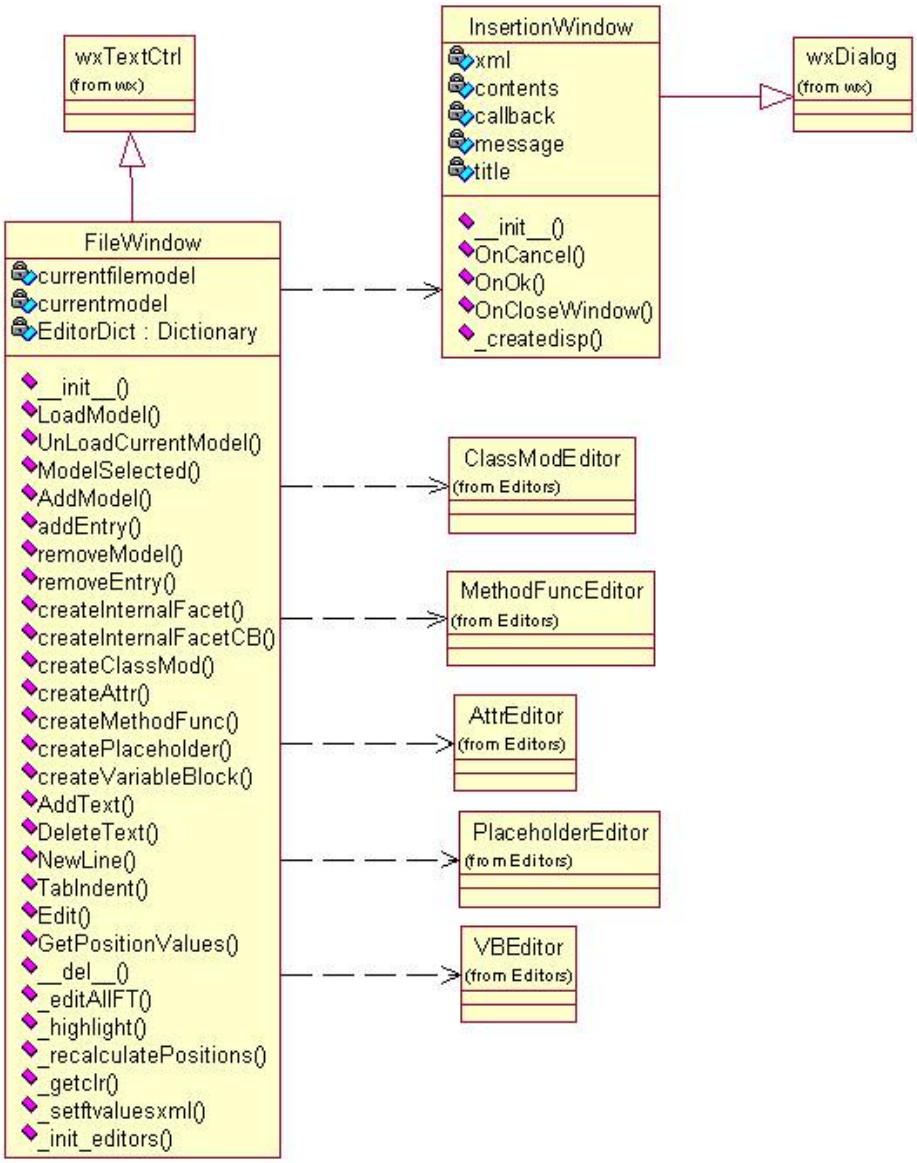


Figure D.9: File Window

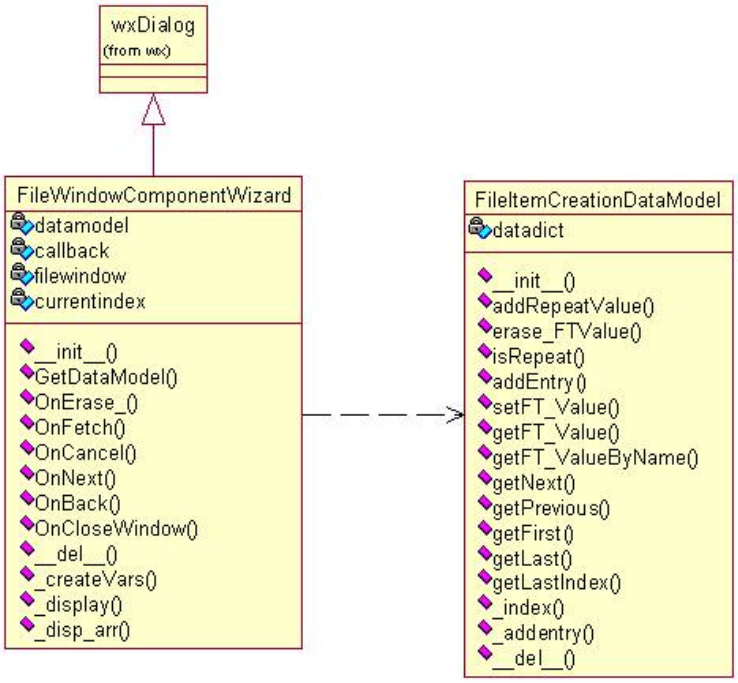


Figure D.10: File Window Component Wizard

Appendix E

Simple CORBA Service Facet

This Appendix contains screen dumps of three when the SimpleCORBAService Sub-Facet were being edited.

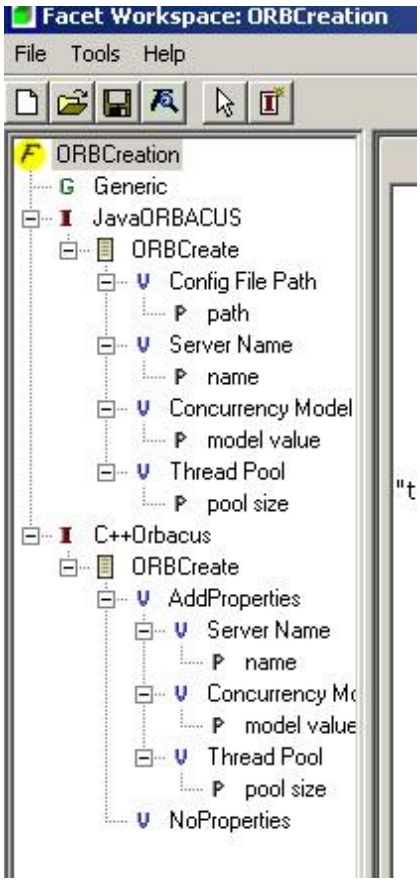


Figure E.1: ORB Creation Facet: IDE structure

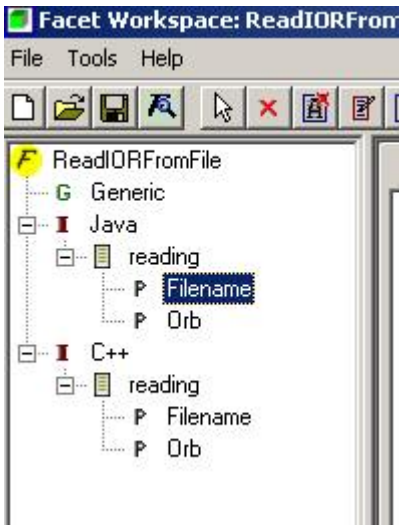


Figure E.2: Read IOR From File Facet: IDE Structure

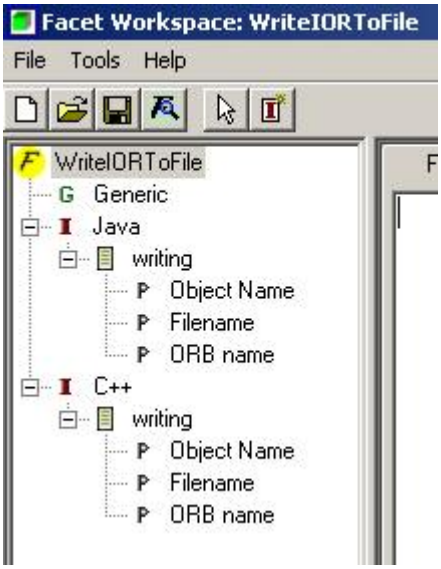


Figure E.3: Write IOR from File Facet: IDE Structure

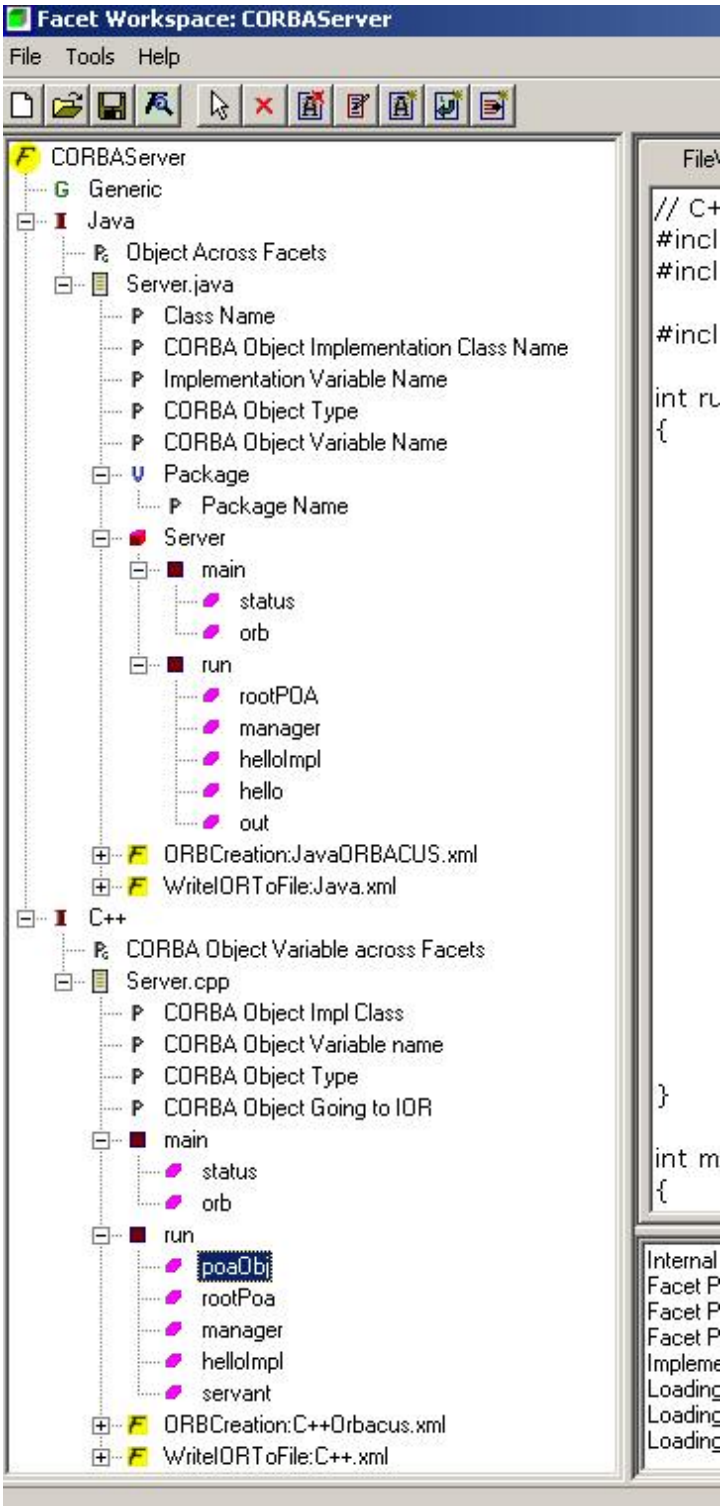


Figure E.4: CORBA Server Facet: IDE Structure

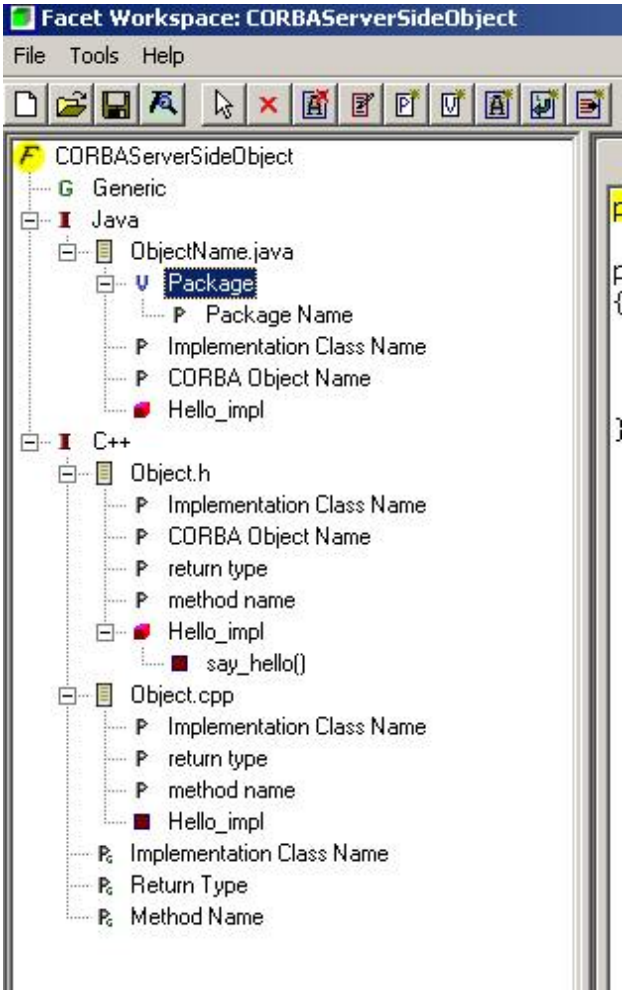


Figure E.5: CORBA Server Side Object

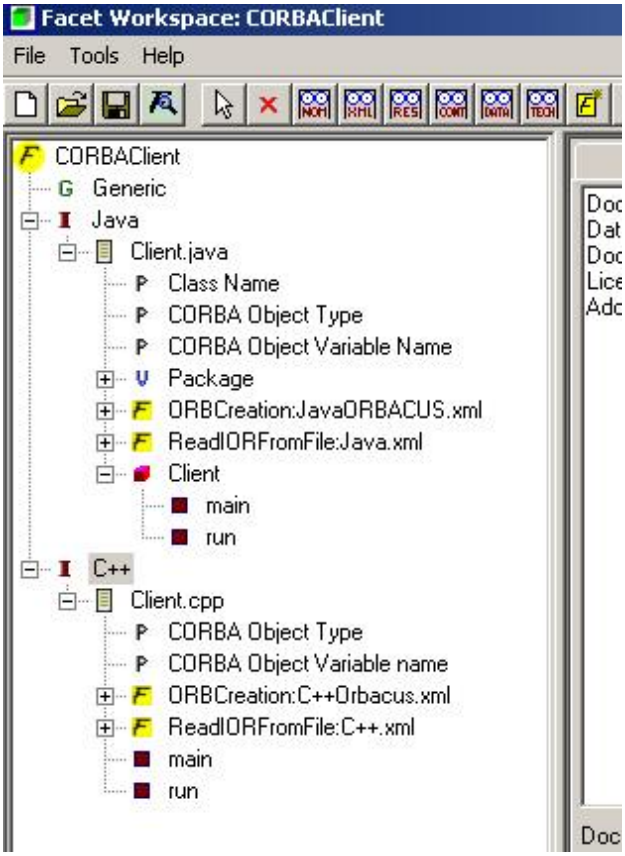


Figure E.6: CORBA Client Facet: IDE Structure

Appendix F

Sample meta-π xml file

Below is a meta-π file that represents the *Java* Implementation in the *SimpleCORBAService* Facet.

```
<?xml version='1.0' encoding='UTF-8'?>
<SEMTA>
  <metapi type='Implementation' name='SimpleCORBAService' implname='Java'>
    <Creator date='Fri Jun 06 10:33:31 2003' institute ='University of the Witwatersrand
      ' name='Bilal A.R. Jagot'/>
    <Definition>
      <Description>
        <Problem>How to demonstrate a Simple CORBA Service with a single Object that
          possesses a minimal set of operations?</Problem>
        <Intent>Client to Server operations. To say "hello" on the Server side.</Intent
          >
        <Forces>Simple CORBA server. Single Object. IOR in File.</Forces>
        <Applicability>Simple CORBA services</Applicability>
        <Participants>CORBA Server, CORBA client, IDL Definition for client-server
          interaction.</Participants>
        <KnownUses>None</KnownUses>
        <Structure>
        Place the tree that describes the .src file here.
        <Facet to="" from="" name='CORBAServer:Java.xml' type='external'><settings>
          Place the tree that describes the .src file here.
                <File name='Server.java' implementationLanguage='Java'>
```

<Facet to=’929’ from=’88’ name=’ORBCreation:JavaORBACUS.xml’ value=’ java .util. Properties props = System. getProperties () ; props.put(”org.omg.CORBA .ORBClass”, ”com.ooc.CORBA.ORB”); props.put(”org.omg.CORBA.ORBSingletonClass ”, ”com.ooc.CORBA.ORBSingleton”); props.put(”ooc.config”,”C:\\where\\config .txt”); // props.put(”ooc.orb.server_name ”, ” TheNameOfTheServerIfIMRisBeingUsed ”); // props.put(”ooc.orb.oa.conc_model”, ” threaded , thread_per_client , thread_per_request , thread_pool ”) ; // props.put(”ooc.orb.oa. thread_pool ”,” n>0”); int status = 0; org.omg.CORBA.ORB orb = null; try { orb = org.omg .CORBA.ORB.init(args, props); status = Server.run(orb) ; } catch(Exception ex) { ex. printStackTrace () ; status = 1; } if (orb != null) { try { orb. destroy () ; } catch (Exception ex) { ex. printStackTrace () ; status = 1; } } System. exit (status) ;’ type=’ internal ’><settings>Place the tree that describes the . src file here.<File name=’ORBCreate’ implementationLanguage=’Java’>< variableblock description=’If a new path to a configuration file needs to be specified , use this property . Be sure not to use the succeeding props .put statements else the config file values will be overriden ’ value=’yes ’ name=’Config File Path’><vbinstance to=’331’ from=’283’ value=’props. put(”ooc. config ”,” C:\\where\\config. txt ”);’/>

<placeholder description =’The absolute path to the config file using ”\\” instead of ”\”. Include the config file name.’ value=’C:\\where\\config. txt ’ name=’path’>

<pinstance to=’328’ from=’307’ value=’C:\\where\\config. txt ’/>

</placeholder>

</variableblock>

<variableblock description =’Use this variable block of you need to specify a name for the server because you are using IMR. If you are not using IMR, this property should not be set .’ value=’no’ name=’Server Name’>

<vbinstance to=’407’ from=’336’ value=’no’/>

<placeholder description =’Place the name of the server here is you are going to be using this variable block .’ value=’no’ name=’name’>

<pinstance to=’404’ from=’370’ value=’ TheNameOfTheServerIfIMRisBeingUsed’/>

</placeholder>

</variableblock>

<variableblock description =’The concurrency model that defined how the ORB will handle requests .’ value=’no’ name=’Concurrency Model’>

<vbinstance to=’508’ from=’412’ value=’no’/>

```
        <placeholder description ='The options are threaded ,
          threaded_per_client , thread_per_request or thread_pool . If it
          is thread_pool then the next property must be set also .' value
          ='no' name='model value'>
          <pinstance to ='505' from='448' value='threaded ,
            thread_per_client , thread_per_request , thread_pool '/>
        </placeholder>
      </variableblock >
      <variableblock description ='If the concurrency model is
        thread_pool then this property MUST be set.' value='no' name='
        Thread Pool'>
        <vbinstance to ='555' from='513' value='no'/>
        <placeholder description =' Specifies the number of threads in the
          pool that the ORB will manage. This value must be greater
          than zero . If it is not , a runtime error will be generated .'
          value='no' name='pool size'>
          <pinstance to ='552' from='549' value='n>0'/>
        </placeholder>
      </variableblock >
    </File>
  </ settings >
</Facet>
<Facet to ='1603' from='1304' name='WriteIORToFile:Java.xml' value =' try
    { String ref = orb. object_to_string (obj) ; String refFile = "
  ior . ref "; java . io . PrintWriter out = new java . io . PrintWriter ( new
   java . io . FileOutputStream( refFile )) ; out . println ( ref ) ; out . close
  () ; } catch( java . io . IOException ex) { ex. printStackTrace () ;
    return 1; }' type=' internal '>
  <settings >Place the tree that describes the . src file here.<File name
    ='writing' implementationLanguage='Java'><placeholder description='
    The name of the object whose IOR needs to be stringified and saved
    to file ' value='obj ' name='Object Name'><pinstance to='1354' from
    ='1351' value='obj'/>
      </placeholder>
      <placeholder description ='The name of the file to which the IOR
        must be saved . The file need not exist ' value=' ior . ref ' name='
        Filename'>
        <pinstance to ='1385' from='1378' value=' ior . ref '/>
      </placeholder>
      <placeholder description ='The variable that holds the ORB instance
        ' value='orb ' name='ORB name'>
        <pinstance to ='1333' from='1330' value='orb'/>
      </placeholder>
    </File>
```

```
        </settings>
      </Facet>
      <variableblock description='Is this Server part of a package?' value='
        yes' name='Package'>
        <vbinstance to='22' from='8' value='package hello;'/>
        <placeholder description='The name of the package if the server is
          part of one.' value='' name='Package Name'>
          <pinstance to='21' from='16' value='hello'/>
        </placeholder>
      </variableblock>
      <placeholder description='The name of the Class which must match the
        name of the file .java' value='Server' name='Class Name'>
        <pinstance to='42' from='36' value='Server'/>
      </placeholder>
      <placeholder description='The name of the class that has been created to
        implement the logic of the defined CORBA Object/Interface' value='
        Hello_Impl' name='CORBA Object Implementation Class Name'>
        <pinstance to='1234' from='1224' value='Hello_Impl'/>
        <pinstance to='1261' from='1251' value='Hello_Impl'/>
      </placeholder>
      <placeholder description='The variable name for the implementation
        object' value='' name='Implementation Variable Name'>
        <pinstance to='1244' from='1235' value='helloImpl'/>
        <pinstance to='1290' from='1281' value='helloImpl'/>
      </placeholder>
      <placeholder description='The type of the CORBA Objects' value='' name
        ='CORBA Object Type'>
        <pinstance to='1272' from='1267' value='Hello'/>
      </placeholder>
      <placeholder description='The name of the variable that holds the
        CORBA Object that will be stringified.' value='' name='CORBA Object
        Variable Name'>
        <pinstance to='1278' from='1273' value='hello'/>
      </placeholder>
    </File>
    <gp name='Object Across Facets' value='hello'>
      <gpinstance name='Server.java\CORBA Object Variable Name'/>
      <gpinstance name='Server.java\WriteIORToFile:Java.xml\writing\Object
        Name'/>
    </gp>
  </settings>
</Facet>
<Facet to='' from='' name='CORBAClient:Java.xml' type='external'>
```

<settings>Place the tree that describes the .src file here.<File name='Client .java ' implementationLanguage='Java'><Facet to='865' from='84' name='ORBCreation :JavaORBACUS.xml' value='java.util.Properties props = System. getProperties () ; props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB"); props.put ("org.omg.CORBA.ORBSingletonClass", "com.ooc.CORBA.ORBSingleton"); props .put("ooc.config","C:\\config.txt"); // props.put("ooc.orb.server_name ", " TheNameOfTheServerIfIMRisBei "); props.put("ooc.orb.oa.conc_model", " threaded ") ; // props.put("ooc.orb.oa . thread_pool ","n>0"); int status = 0; org.omg.CORBA.ORB orb = null; try { orb = org.omg.CORBA.ORB.init(args, props); status = Client .run (orb) ; } catch(Exception ex) { ex. printStackTrace () ; status = 1; } if (orb != null) { try { orb. destroy () ; } catch(Exception ex) { ex. printStackTrace () ; status = 1; } } System. exit (status) ; } static int run(org.omg.CORBA.ORB orb) { ' type=' internal '><settings>Place the tree that describes the .src file here.<File name=' ORBCreate' implementationLanguage='Java'><variableblock description='If a new path to a configuration file needs to be specified , use this property . Be sure not to use the succeeding props.put statements else the config file values will be overriden ' value='yes ' name='Config File Path'><vbinstance to='318' from='277' value='props.put("ooc. config ","C:\\config. txt ");'/>

<placeholder description ='The absolute path to the config file using "\\" instead of "\". Include the config file name.' value='C:\\config. txt ' name='path'>

<pinstance to ='315' from='301' value='C:\\config. txt '/>

</placeholder>

</variableblock >

<variableblock description ='Use this variable block of you need to specify a name for the server because you are using IMR. If you are not using IMR, this property should not be set .' value='no' name='Server Name'>

<vbinstance to ='394' from='323' value='no'/>

<placeholder description ='Place the name of the server here is you are going to be using this variable block .' value='no' name='name'>

<pinstance to ='391' from='357' value=' TheNameOfTheServerIfIMRisBeingUsed'/>

</placeholder>

</variableblock >

<variableblock description ='The concurrency model that defined how the ORB will handle requests .' value='yes ' name=' Concurrency Model'>

<vbinstance to ='444' from='397' value='props.put("ooc.orb.oa. conc_model", " threaded");'/>

```
        <placeholder description ='The options are threaded ,
          threaded_per_client , thread_per_request or thread_pool . If it
          is thread_pool then the next property must be set also .' value
          ='threaded ' name='model value'>
          <pinstance to ='441' from='433' value='threaded'/>
        </placeholder>
      </variableblock >
      <variableblock description ='If the concurrency model is
        thread_pool then this property MUST be set.' value='no' name='
        Thread Pool'>
        <vbinstance to ='491' from='449' value='no'/>
        <placeholder description =' Specifies the number of threads in the
          pool that the ORB will manage. This value must be greater
          than zero . If it is not , a runtime error will be generated .'
          value='no' name='pool size'>
          <pinstance to ='488' from='485' value='n>0'/>
        </placeholder>
      </variableblock >
    </File>
  </ settings >
</Facet>
<variableblock description ='If the client belongs under a package then
  this line should be used , specifying what the package name is .' value
  ='yes ' name='Package'>
  <vbinstance to ='14' from='0' value='package hello ;'/>
  <placeholder description ='The name of the package that this class is
    in .' value ='' name='Package Name'>
    <pinstance to ='13' from='8' value='hello '/>
  </placeholder>
</variableblock >
<placeholder description ='The name of the class . If the class name
  changes , the filename must change also ' value ='' name='Class Name'>
  <pinstance to ='35' from='29' value='Client '/>
</placeholder>
<Facet to ='1237' from='916' name='ReadIORFromFile:Java.xml' value='org.
  omg.CORBA.Object obj = null; try { String refFile = '' ior . ref
  ''; java . io . BufferedReader in = new java . io . BufferedReader ( new
  java . io . FileReader ( refFile )) ; String ref = in . readLine () ; obj =
  orb . string_to_object ( ref ) ; } catch ( java . io . IOException ex) {
  ex . printStackTrace () ; return 1; } Hello hello = HelloHelper .
  narrow(obj) ; return ' type=' internal '>
```

```
        <settings>Place the tree that describes the .src file here.<File name
          ='reading' implementationLanguage='Java'><placeholder description='
          The name of the file where the IOR resides' value='ior.ref' name='
          Filename'><pinstance to='989' from='982' value='ior.ref'/>
            </placeholder>
            <placeholder description ='The name of the variable that holds the
              ORB instance' value='orb' name='Orb'>
              <pinstance to='1132' from='1129' value='orb'/>
            </placeholder>
          </File>
        </settings>
      </Facet>
      <placeholder description ='The Type of CORBA Object you want to narrow'
        value ='' name='CORBA Object Type'>
        <pinstance to='1246' from='1241' value='Hello'/>
        <pinstance to='1260' from='1255' value='Hello'/>
      </placeholder>
      <placeholder description ='The name of the narrowed CORBA Object, on
        which operations can be carried out.' value ='' name='CORBA Object
        Variable Name'>
        <pinstance to='1252' from='1247' value='hello'/>
      </placeholder>
    </File>
  </settings>
</Facet>
<Facet to ='' from ='' name='CORBAServerSideObject:Java.xml' type='external'>
  <settings>Place the tree that describes the .src file here.<File name='
    ObjectName.java' implementationLanguage='Java'><variableblock description='
    If this class should belong in a package then a package name should be
    used.' value='yes' name='Package'><vbinstance to='14' from='0' value='
    package hello;'/>
      <placeholder description ='The name of the package that this class
        belongs to' value ='' name='Package Name'>
        <pinstance to='13' from='8' value='hello'/>
      </placeholder>
    </variableblock>
    <placeholder description ='The Implementation class name that must be the
      same as the .java filename' value='Hello_Impl' name='Implementation
      Class Name'>
      <pinstance to='39' from='29' value='Hello_Impl'/>
    </placeholder>
    <placeholder description ='The name of the CORBA Object that we want to
      extend.' value ='' name='CORBA Object Name'>
      <pinstance to='53' from='48' value='Hello'/>
```

```
        </placeholder>
      </File>
    </ settings >
</Facet>
<File name='Hello.idl ' implementationLanguage='IDL'>
  <placeholder  description ='The CORBA Object from the IDL.' value ='' name='
    CORBA Object'>
    <pinstance to ='17' from='17' value =''/>
    <pinstance to ='22' from='17' value='Hello'/>
  </placeholder>
  <placeholder  description ='The name of the Operation in the IDL file ' value
    ='' name='Operation Name'>
    <pinstance to ='40' from='31' value=' say_hello '/>
  </placeholder>
  <placeholder  description ='The return type of the IDL operation ' value ='
    name='Return Type'>
    <pinstance to ='30' from='26' value='void'/>
  </placeholder>
  <classmod to='46' from='7'>
    <name to='22' from='17' value='Hello'/>
    <type to ='16' from='7' value=' interface '/>
    <expcontrol to ='0' from='0' value =''/>
    <stereotype to ='0' from='0' value =''/>
    <methodfunc to='43' from='26'>
      <name to='40' from='31' value=' say_hello '/>
      <return to ='30' from='26' value='void'/>
      <expcontrol to ='0' from='0' value =''/>
      <stereotype to ='0' from='0' value =''/>
    </methodfunc>
  </classmod>
</File>
<gp name='CORBA Object Type' value='Hello'>
  <gpinstance name='CORBAServer:Java.xml\Server.java\CORBA Object Type'/>
  <gpinstance name='IDL\CORBA Object'/>
  <gpinstance name='CORBAClient:Java.xml\Client.java\CORBA Object Type'/>
</gp>
<gp name='CORBA Interface Operation' value='say_hello'>
  <gpinstance name='IDL\Operation Name'/>
</gp>
<gp name='CORBA Interface Operation Return Type' value=''/>
<gvb name='Package' value='yes'>
  <gvbinstance name='CORBAClient:Java.xml\Client.java\Package'/>
  <gvbinstance name='CORBAServer:Java.xml\Server.java\Package'/>
```

```
        <gvbinstance name='CORBAServerSideObject:Java.xml\ObjectName.java\Package
          '/>
      </gvb>
      <gp name='Package Name' value='hello'>
        <gpinstance name='CORBAServerSideObject:Java.xml\ObjectName.java\Package
          \Package Name'/>
        <gpinstance name='CORBAClient:Java.xml\Client.java\Package\Package Name
          '/>
        <gpinstance name='CORBAServer:Java.xml\Server.java\Package\Package Name
          '/>
      </gp>
    </Structure>
    <FacetHierarchy>Platform.ExecutionEnvironment.ArchitecturalComponent.Component.
      FDL.</FacetHierarchy>
    <Technology>
      <t_item name='Java'/>
      <t_item name='CORBA'/>
    </Technology>
  </Description>
  <Behaviour>
    <Rules>a course of action the model follows</Rules>
    <Policies>detailed version of the rules</Policies>
    <Limitations>At what point does the solution breakdown!</Limitations>
  </Behaviour>
  <Interactions>
    <Relations>Specify the inheritance, containment, etc.</Relations>
    <Collaborations>Specify the calls from one component to the next. MSCs are built
      from this element. If special requirements are necessary create a new element
      such as SDL here.</Collaborations>
  </Interactions>
  <Logic/>
</Definition>
<Data>
  <Input>
    <Item source="" type="" target="">
      <ValueDescription>This is a description of the input item.</ValueDescription>
    </Item>
  </Input>
  <Algorithm>
    <Item source="" type="" target="">
      <ValueDescription>This is a description of the item.</ValueDescription>
    </Item>
  </Algorithm>
  <Output>
```

```
    <Item source =" type =" target ="">
      <ValueDescription>This is a description of the output item.</ValueDescription
        >
    </Item>
  </Output>
</Data>
<!--The Context and Resources Section will fully qualify what is required for a
  successful implementation of the Facet.-->
<Context>
  <Business>
    <Environment priority='Minimum' type=" name="">
      <ValueDescription>Describing the environmental context.</ValueDescription>
    </Environment>
  </Business>
  <Design>
    <Environment priority='Minimum' type=" name="">
      <ValueDescription>Describing the environmental context.</ValueDescription>
    </Environment>
  </Design>
  <Operational>
    <Environment priority='Minimum' type=" name="">
      <ValueDescription>Describing the environmental context.</ValueDescription>
    </Environment>
  </Operational>
  <Deployment>
    <Environment priority='Minimum' type=" name="">
      <ValueDescription>Describing the environmental context.</ValueDescription>
    </Environment>
  </Deployment>
</Context>
<Resources>
  <Internal>
    <Resource priority='Minimum' type='type of Resource' name='name'>
      <ValueDescription>Describing the resource.</ValueDescription>
    </Resource>
  </Internal>
  <Boundary>
    <Resource priority='Minimum' type='type of Resource' name='name'>
      <ValueDescription>Describing the resource.</ValueDescription>
    </Resource>
  </Boundary>
  <External>
    <Resource priority='Minimum' type='type of Resource' name='name'>
      <ValueDescription>Describing the resource.</ValueDescription>
```

```
        </Resource>
      </External>
    </Resources>
    <Documentation>
      <Entry date='12/12/2002' time='00:00'>
        <DocEntry>What the author wants to say!!</DocEntry>
        <AdditionalInfo>What additional info that may support that above docentry or the
          user must just refer to.</ AdditionalInfo >
      </Entry>
    </Documentation>
    <Extensions>
      <Children/>
      <Contains/>
      <Other>
        <ExamplesOfUse>See Orbacus Manual</ExamplesOfUse>
      </Other>
    </Extensions>
  </metapi>
</SEMTA>
```

Appendix G

CD Guide

Figure G.1 shows the directory structure of the accompanying CD. It is suggested that the reader explore the **index.html** file to navigate through the contents of the CD.

The CD includes:

- A softcopy of this thesis;
- A UML Design of the FDE;
- Source code for the FDE;
- Instructions for the installation of the FDE; and
- Useful software.

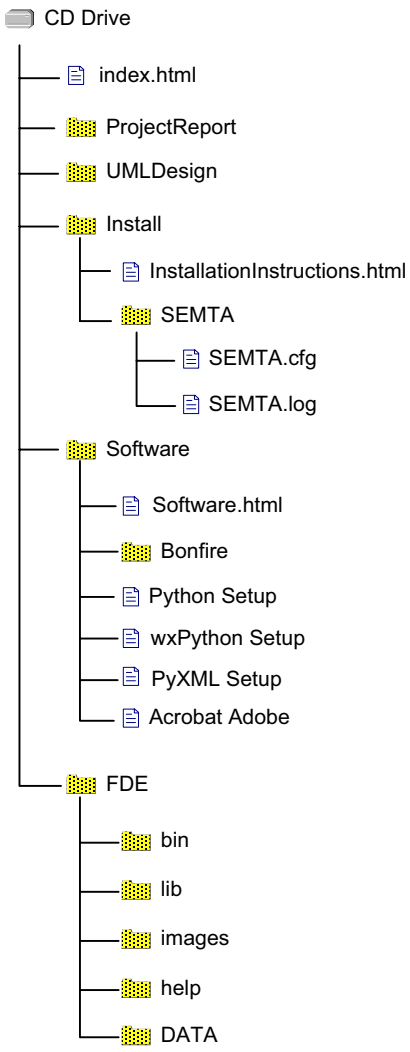


Figure G.1: Directory structure of accompanying CD