“An Internet Protocol Testing Framework” ipTF

Submitted for the qualification of
M.Sc in Computer Science

at

Trinity College Dublin

1998

John Cashman B.Sc.

Distributed Systems Group
Department of Computer Science

“An Internet Protocol Testing Framework”
ipTF

Declaration

I hereby declare that this dissertation is my own work except where otherwise stated and it has not been previously submitted to this or any other university.

John Cashman
29th September 1998

I hereby agree that Trinity College Library may lend or copy this dissertation upon request.

John Cashman
29th September 1998

Abstract

As the Internet expands and proliferates it gives rise to new technologies that require supporting Internet protocols. Some recent examples include HTTP, POP, IMAP and IIOP. Internet applications are generally based on a client server model. This results in protocols being developed to support communication between the client and server. Additionally many existing protocols are updated regularly to support new and enhanced features.

There is a user requirement for software products that include native support for both new and upgraded Internet protocols as they become available. Software development organizations seek to meet this demand by including new protocol implementations into their software products. Many companies now support multiple Internet Protocol implementations as standard in their communication products. For example products such as Lotus Domino communication server currently include support for several Internet Protocols. (SMTP, HTTP, IIOP, IMAP4, POP3). As new protocols mature and they become generally available, corresponding implementations are added to the communications server.

Developers need to ensure that their protocol implementation can communicate/inter-operate with other products that implement the protocol. This is achieved by having the implementation conform to the protocol specification. Considerable resources are spent on the design and implementation of test applications that check protocols for conformance to specification. To date the general approach used in testing Internet Protocols is to design protocol specific test applications. As the number of new protocols increases and existing protocols are revised a common methodology for testing Internet Protocols would substantially reduce the time spent on developing protocol test applications.

Internet protocols are normally specified in RFCs. Other than this general procedural requirement there is no set of rules or guidelines that must be followed for specifying an Internet protocol. This lack of formality allows developers to design new protocols in an efficient and flexible manner but results in each Internet protocol having unique syntax and semantics. Designing a common protocol testing application is therefore difficult.

The goal of this dissertation is to examine the feasibility of applying a common method to testing multiple Internet Protocols. Object-oriented framework technology and design patterns were used to design a set of related classes both abstract and concrete that can be used as a basis for creating an Internet protocol testing application. To create a new application the Internet Protocol Test Framework (ipTF) is extended through inheritance and object composition. A sample implementation of the framework was completed using the DSG Mobile IIOP and Telnet protocols.

The framework is evaluated against the criteria of re-usability, simplicity and efficiency. The factoring out of common design structure and behavior of Internet protocols provides the basis for framework design. Syntactic or semantic similarities in protocols are used at a lower level to help refine the framework. Some additional improvements and refinements are suggested which could make the framework more black-box in nature.

"An Internet Protocol Testing Framework" ipTF

Abstract

John Cashman
M.Sc. Dissertation 1998

As the Internet expands and proliferates it gives rise to new technologies that require supporting Internet protocols. Some recent examples include HTTP, POP, IMAP and IIOP. Internet applications are generally based on a client server model. This results in protocols being developed to support communication between the client and server. Additionally many existing protocols are updated regularly to support new and enhanced features.

There is a user requirement for software products that include native support for both new and upgraded Internet protocols as they become available. Software development organizations seek to meet this demand by including new protocol implementations into their software products. Many companies now support multiple Internet Protocol implementations as standard in their communication products. For example products such as Lotus Domino communication server currently include support for several Internet Protocols. (SMTP, HTTP, IIOP, IMAP4, POP3). As new protocols mature and they become generally available, corresponding implementations are added to the communications server.

Developers need to ensure that their protocol implementation can communicate/inter-operate with other products that implement the protocol. This is achieved by having the implementation conform to the protocol specification. Considerable resources are spent on the design and implementation of test applications that check protocols for conformance to specification. To date the general approach used in testing Internet Protocols is to design protocol specific test applications. As the number of new protocols increases and existing protocols are revised a common methodology for testing Internet Protocols would substantially reduce the time spent on developing protocol test applications.

Internet protocols are normally specified in RFCs. Other than this general procedural requirement there is no set of rules or guidelines that must be followed for specifying an Internet protocol. This lack of formality allows developers to design new protocols in an efficient and flexible manner but results in each Internet protocol having unique syntax and semantics. Designing a common protocol testing application is therefore difficult.

The goal of this dissertation is to examine the feasibility of applying a common method to testing multiple Internet Protocols. Object-oriented framework technology and design patterns were used to design a set of related classes both abstract and concrete that can be used as a basis for creating an Internet protocol testing application. To create a new application the Internet Protocol Test Framework (ipTF) is extended through inheritance and object composition. A sample implementation of the framework was completed using the DSG Mobile IIOP and Telnet protocols.

The framework is evaluated against the criteria of re-usability, simplicity and efficiency. The factoring out of common design structure and behavior of Internet protocols provides the basis for framework design. Syntactic or semantic similarities in protocols are used at a lower level to help refine the framework. Some additional improvements and refinements are suggested which could make the framework more black-box in nature.

Acknowledgements

I would especially like to thank my dissertation supervisor Dr Vinny Cahill for his valuable assistance, encouragement and advice in the completion of this dissertation.

I wish also to thank Dr Brian O'Donovan of Lotus Development Ireland for his support given throughout this work. I am grateful to Ray and Tony for their advice on technical matters. Thanks to Alice and Tadhg for reviewing my work and to Deirdre for her consistent encouragement to complete this dissertation. To all my family and friends who supported me in many different ways over the past year I am very grateful.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

1. INTRODUCTION

As the Internet expands and proliferates it gives rise to new applications that require supporting Internet protocols. Some recent examples of this include HTTP, POP, IMAP and IIOP. HTTP is the underlying protocol of the World Wide Web (WWW). POP enables remote user access to Internet mailboxes. IMAP offers a similar service to POP but has additional features such as disconnected replication. IIOP is a protocol designed to relay remote object invocations between Object Request Brokers (ORB's). Internet applications are generally based on a client/server model. This results in protocols being developed to support communication between the client and server. Additionally many existing protocols are updated regularly to support new and enhanced features.

There is a user requirement for software products that include native support for both new and upgraded Internet protocols as they become available. Software development organizations seek to meet this demand by including new protocol implementations into their software products. Many companies now support multiple Internet Protocol implementations as standard in their communication products. For example products such as the Lotus Domino communication server [Domino] currently include support for several Internet Protocols (SMTP, HTTP, IIOP, IMAP4, POP3). As new protocols mature and become generally available, corresponding implementations are added to the communications server.

Developers need to ensure that their protocol implementations can communicate/inter-operate with other products that implement the protocol. This is achieved by having the implementation conform to the protocol specification. Considerable resources are spent on the design and implementation of test applications that check protocols for conformance to specification. To date, the general approach used in testing Internet Protocols has been to design protocol-specific test applications. As the number of new protocols increases and existing protocols are revised, a common methodology for testing Internet Protocols would substantially reduce the time spent on developing protocol test applications. It would also provide a consistent approach to testing a range of Internet protocols for conformance to specification.

Internet protocols are normally specified in *Request for Comments* (RFCs) documents. Other than this general procedural requirement there is no set of rules or guidelines that must be followed for specifying an Internet protocol. This lack of formality allows developers to design new protocols in an efficient and flexible manner. It results, however, in each Internet protocol specification having unique syntax and semantics. Designing a common protocol testing application is therefore difficult.

The goal of this dissertation is to examine the feasibility of applying a common approach or method to testing multiple Internet Protocols. The dissertation examines this concept with particular reference to conformance testing. Object-oriented framework technology and design patterns are used to design a set of related classes, both abstract and concrete, that can be used as a basis for creating an Internet protocol testing application. To create a new application the Internet Protocol Test Framework (ipTF) is extended through inheritance and object composition. A sample implementation of the framework was completed using the DSG Mobile IIOP and Telnet protocols.

The framework was evaluated against the criteria of re-usability, simplicity and efficiency. The factoring out of common design structure and behaviour of Internet protocols provides the basis for the framework design. Syntactic or semantic similarities in protocols are used at a lower level to help refine the framework. Some additional improvements and refinements are suggested which could make the framework more black-box in nature.

1.1. Background

This section briefly reviews some background material that is relevant to the dissertation. The first part focuses on the use of framework technology. This discussion makes reference to design patterns, which are used to help design and implement a framework. Following this, the general nature of communication protocols is discussed. Special reference is made to Internet and OSI protocols. In the final part general software testing concepts are outlined and the specific requirements for protocol conformance testing reviewed. An understanding of these areas is central to the design of the ipTF.

1.1.1. Frameworks and Design Patterns

A key problem in designing a re-usable protocol test application is how to factor out the common functionality and behaviour from the protocol specific parts. Framework technology was chosen as the design methodology because it promotes a re-usable software design and a component architecture.

Object-oriented framework technology is commonly used as a basis for developing a set of related applications. It builds on the traditional object-oriented design concepts of inheritance, polymorphism, encapsulation and dynamic binding. A framework is composed of a set of classes (abstract and concrete) and the defined relationships between them. New application instances are created by extending the basic framework either through inheritance or object composition. The framework concept has been applied successfully to a large number of problem domains including the design and development of network protocol software itself [Hüni94].

Creating a framework begins with general analysis of the problem domain and ideally, specific analysis of a selection of existing applications in the problem domain. Three applications are considered about right for this type of analysis (rule of three). Framework analysis differs from the traditional object-oriented analysis of “find the objects” exercise. Instead, the focus is on finding areas of common functionality between applications. Based on this type of analysis a framework is modeled which describes a set of abstract base classes, the relationships between them and a sample/default concrete realisation of an application instance.

Frameworks provide the application developer with a pre-defined structure for creating a particular application instance. A set of abstract classes is used as a design basis for all application instances. The developer is required only to extend the framework in order to create a new application. Extension is achieved either through inheritance (subclassing) or object composition.

A framework is more than a set of class libraries in that it defines the flow of control for a new application instance. Application development based on class libraries alone requires the developer to design the flow of control in the application and define the interaction between objects. Furthermore, no default behaviour is specified in class libraries that could serve as a guide for application developers.

Frameworks result in a longer-term saving as they leverage domain expertise, promote consistency and integration across applications, reduce maintenance and enhance developer productivity. They do however require more effort to build and learn and require extra documentation, maintenance and support.

Initially most frameworks are extended through inheritance. This is known as a white box framework. The developer must be aware of the structure of the framework and is responsible for design and coding of the extended classes. This allows for flexibility in design but can result in violating the encapsulated nature of the framework classes. As the framework evolves through re-use, common functionality is factored out and placed in generic classes. Design patterns are applied to resolve common problems encountered in the framework design. Most pattern solutions are based on a compositional structure. This leads to the framework becoming more black-box in nature. Though more difficult to achieve, a black-box framework is easier to use and has a higher re-use potential. In this regard it is the preferred type.

As stated in the previous paragraph design patterns help identify framework design issues and can be used to implement solutions that build and refine the framework. The notion of patterns is very broad and they can be applied in many contexts. Originally they were first applied in the design of buildings. In more recent years software developers have used them in the design and implementation of software systems. Software patterns are sometimes classified as architectural patterns, design patterns and idioms. Architectural patterns are the highest level of abstraction. They are capable of representing whole systems. Design patterns deal with micro-architectures or object structures. Idioms are low-level patterns found in programming languages.

In this dissertation we are primarily concerned with design patterns (object structures). The Design Patterns book [Gam94] is almost entirely devoted to patterns of this type. A set of creational, structural and behavioural design patterns is the subject of this book. These patterns provide solutions for common framework design issues. The abstract factory, strategy and adapter design patterns documented in this book were applied in the design of the ipTF. The application of these design patterns has significantly improved the re-use potential of the framework.

1.1.2. Network Communication Protocols

Before designing a network protocol testing framework it was useful to examine the general nature and characteristics of communication protocols. This analysis helped identify the structure and behaviour that is common to all protocols. Communication protocols are rules that govern the communication between different components in a distributed environment. Protocol engineering is concerned with developing communication protocol specifications, implementations and performing validation. A protocol specification defines the required behaviour of a protocol entity. It consists of five parts:

i. the **service** to be provided by the protocol
ii. assumptions about the environment in which the protocol is executed
iii. a **vocabulary** of messages used to implement the protocol
iv. a method of **encoding** (formatting) messages in the vocabulary
v. a set of **procedure rules** guarding the consistency of message exchanges

Protocols are sometimes compared to spoken languages as follows.

Spoken language:	Vocabulary	Syntax	Grammar	Semantics
Communication protocol:	Message types	Encoding rules	Procedure Rules	Service Specification

Each part of the protocol specification can define a hierarchy of elements. The protocol vocabulary, for example, can consist of a hierarchy of message classes. Some aspects of protocol design are relevant to conformance testing. For example a test message is generally taken from the protocol's message vocabulary. Messages are represented in the ipTF by the abstract class TestMsg. An understanding of protocol design issues is helpful in designing a protocol implementation test application.

1.1.3. Protocol Conformance Testing

When a protocol implementation is tested against the protocol specification to ensure compatibility with other implementations of the protocol this is referred to as ***protocol conformance testing***. A conformance test will fail only if the implementation and specification differ.

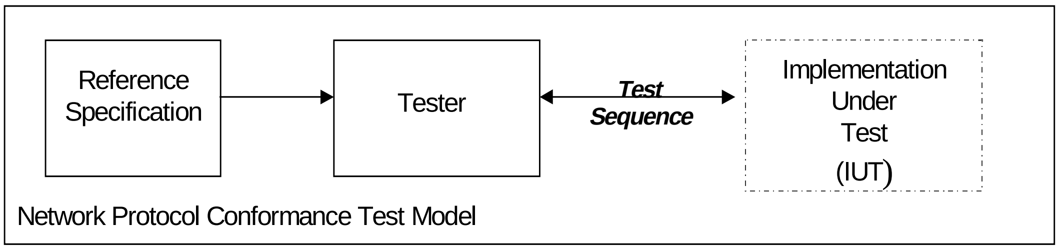

Reference Specification
Tester
Test Sequence
Implementation Under Test (IUT)
Network Protocol Conformance Test Model

Figure 1: Conformance Testing

Figure 1 describes the general model used for protocol conformance testing. Many protocols can be specified as a finite state machine, with a limited number of states and a finite set of inputs and outputs. Given a reference specification and an implementation of the protocol, input sequences are applied to test the protocol for conformance to its specification. The implementation effectively acts as a black box that accepts inputs and produces outputs in response. The resulting outputs are checked against the outputs as prescribed by the formal specification. This type of testing is referred to as black box testing.

A conformance protocol test passes if all observed outputs match outputs as prescribed by the formal specification. The series of input sequences used to exercise the protocol specification is known as the conformance test suite. [IS 9646] defines an abstract test suite structure that is used for testing of all Open Systems Interconnection (OSI) protocols. The hierarchy of this test suite structure is shown in *Figure 2.* A *Test Case* is the fundamental building block of the test suite. It stores a sequence of test events, which are used to test a particular feature of the protocol. A *Test Event* is the smallest indivisible test unit. A *Test Group* is simply a selection of *Test Cases*. This structure is sufficiently abstract to be applied to the testing of Internet protocols. For this reason the design of the ipTF test suite is based on this structure.

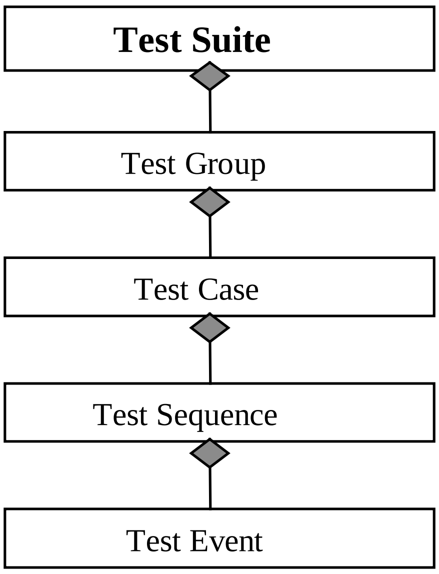


Figure 2: IS 9646 Test Suite Structure

The method for developing test cases is dependent on the specification formalism. Internet protocols are generally specified in natural language. This can lead to ambiguities of meaning and can make it difficult to check for completeness and correctness of the test suite. To fully test a protocol implementation, a series of sequences must be defined to form the test suite. This requires generating a set of sequences to fully test all functions described in the specification and to verify that the implementation will reject invalid inputs. It is almost impossible to check an unknown implementation for all possible behaviours simply by probing it and observing its responses. A representative subset of all possible test sequences is generally sufficient to cover conformance testing of protocols. Automated test case generation of appropriate test cases can greatly improve the efficiency of a testing application.

A test sequence that has the potential to uncover many faults is better than one that can only uncover a few faults. It is virtually impossible to establish in advance how many faults a given test sequence can potentially uncover. One approach to resolving this problem is that of using test adequacy criteria to distinguish good sequences from bad. Based on this paradigm algorithms have been developed [Kore96] to automate the generation of test sequences.

1.2. Problem Outline

Finding a common method for testing multiple Internet protocols is the main goal of this dissertation. Object-oriented framework technology was considered a suitable design methodology to help achieve this goal. When complete the framework should provide a structure capable of generating protocol test applications for a wide range of Internet Protocol implementations and revisions. For the purposes of this dissertation Internet protocols are those in the TCP/IP application layer. Assessing the suitability of framework technology to design and implement the protocol testing application is also an important objective.

The testing requirement for rapidly developing new and existing Internet protocols requires that the framework be adaptable, extensible, efficient and simple to implement. It is important that design consideration be given to these criteria especially as the framework evolves through refinement. The initial objective however, is to design the basic structure of the protocol test framework. The analysis of three domain applications (rule of three) combined with domain expertise gained through research of communication protocols and conformance testing provides the basis for generating this initial design. The application of design patterns and the factoring out of lower level common functionality such as protocol syntax and semantics should eventually lead to a framework design that accurately models the protocol testing application domain.

Ultimately the framework should have the capability of generating a test application for any new or existing protocol in an accurate, efficient, and simple way. The framework could then be used to build a multi-protocol test application for testing a communication server that implements multiple Internet protocols such as the Lotus Domino communication server.

1.3. Framework Design & Implementation

Following initial background research on a range of Internet protocols it became clear that most Internet protocol specifications have unique syntax and semantics. Protocols do however have some common structures and behaviour. For this reason the framework design was largely based on the common structure and behaviour of Internet protocols and the common features of protocol conformance test applications.

To capture the domain requirements an SMTP test application developed by Lotus International and a simple Java Beans Telnet implementation were analyzed for common design structures and functionality. Two existing framework models Conduits+ “A Framework for Network Software” [Huni94] and “A Framework-Based approach to the Development of Network Aware Applications” [Boll98] were especially relevant to the design of a framework for protocol testing.

From analysis the main classes identified were Protocol, ProtocolAdapter, TestSuite, Log, TestMsg, and MsgPanel. An initial framework design was created using these classes. Subsequently the framework was refined using design patterns and by factoring out common parts. TestFactory was introduced to implement the abstract factory pattern. Its purpose is to control the creation of protocol specific objects required by a new application instance. MsgReader and MsgComparitor classes were used to encapsulate message reading and comparison algorithms from TestEvent and TestMsg respectively. Removal of the comparison and reading algorithms from TestEvent and TestMsg classes allowed them to become generic.

Java was chosen as the framework development language. Java fully supports the object-oriented programming paradigm and by extension is suitable for framework development. Borland’s Java rapid application development tool JBuilder was selected as the development environment. Protocol implementations developed in C/C++ such as the DSG Mobile IIOP require interface adaption using the Java C/C++ native interface.

1.4. Achievements

The main output of this project is the design and implementation of a framework for testing multiple Internet protocols. An understanding of framework concepts, conformance testing and network protocol design were acquired during the course of this dissertation. Many aspects of object-oriented design were also encountered including framework development principles, design patterns, the Unified Modeling Language (UML) and use of the Rational Rose, a UML based design tool. Many new Java and C++ programming skills were developed during the implementation phase. The integration of the DSG Mobile IIOP implementation into the framework was perhaps the most difficult technical task. C++ was used as the language of development for the DSG Mobile IIOP implementation while the basic framework itself was written in Java. This required becoming familiar with Java’s native C++ API, the C++ DLL creation process and mapping of Java data types to C++ data types.

Meeting the original dissertation goals and the specific criteria established for the framework design required an assimilation of a wide range of new skills and knowledge to create an appropriate solution. Both the design and implementation of the ipTF framework give proof that a common approach to Internet protocol testing is feasible. More specifically the use of framework technology to achieve this is vindicated. This dissertation can be used as foundation material for subsequent research in developing a more extensive framework for Internet protocol testing.

1.5. Dissertation Roadmap

The structure of the dissertation is given below. Each chapter's content is described in brief.

Chapter 1. **Introduction**. The introductory chapter is intended as an overview of the dissertation. The general problem of protocol testing and the motivation for applying framework technology in this scenario is discussed. Necessary background information on framework technology, design patterns and protocol conformance testing is given. ipTF design considerations and implementation issues are reviewed. Dissertation achievements are summarised in the concluding part of this chapter.

Chapter 2. **Frameworks and Design Patterns** A review of literature and research undertaken in the area of frameworks and design patterns is included here. The survey reviews each area with reference to the current state of the art.

Chapter 3. **Network Protocols & Conformance Testing** This chapter reviews literature covering the nature of communication protocols and conformance testing in general.

Chapter 4. **Framework Design**. This chapter describes the framework design stage and highlights the main issues encountered. Rational Rose, a UML based design tool was used to generate Use Case diagrams, Object Models and Interaction Diagrams from the analysis phase. References are made to these models in discussions about the framework design. The ipTF framework structure and classes are described here. Refining the framework through the application of design patterns is also discussed in this chapter.

Chapter 5. **Implementation** This chapter describes the main features of the ipTF implementation and the main user-interface screens are highlighted here. The framework itself is written in Java. Interface adapter classes for IIOP are written in C/C++ to run natively within a Java framework. The practical application of applying design patterns in Java is discussed briefly and finally, a description of how to use the framework to create a new protocol test application is included.

Chapter 6. **Evaluation** This chapter assesses the framework design based on the criteria of re-usability, efficiency and simplicity. Particular attention is given to the criteria of framework re-usability. The framework is also evaluated in relation to possible improvements and refinements that could be incorporated into future versions of the framework.

2. FRAMEWORKS & DESIGN PATTERNS

The use of framework technology and design patterns is an important aspect of this dissertation. This chapter presents a literature survey of the research undertaken on frameworks and design patterns. The survey reviews each area with reference to the current state of the art. The first part of the review examines the background, definition, characteristics, and advantages of framework technology. Particular reference is made to design and implementation features. The second part of the review looks at the concept of design patterns. Their role in relation to framework development is examined. Architectural styles and pattern languages are discussed in the final part of the review.

2.1. Frameworks

The main goal of this project was to design a framework for testing Internet protocols. In this section framework technology is reviewed with reference to various articles and publications. The review looks at a range of current object-oriented technologies. It begins with a look at software development processes adopted in recent years. Procedural programming, object-oriented design and frameworks are contrasted. The main body of the review focuses on frameworks and design patterns. Other object-oriented technologies that provide a context for examining and understanding the framework concept are reviewed. These include concepts such as architectural styles and pattern languages.

Figure 3 offers a view of several separate but related object technologies and techniques. [Tepf97] describes this chart as the "The Unified Object Topology". It relates frameworks, kits, object patterns, domain models, architectural styles, and domain taxonomies. The topology provides a context to better understand and use these technologies in large-scale distributed systems. Each technology is represented in a two dimensional grid based on the attributes of Implementation and Domain.

Implementations are to some degree either abstract or concrete. An abstract implementation might be expressed in natural language whereas a concrete implementation might for example be expressed in machine executable code. The domain-dependency attribute describes development topics in relation to the application domain. If the topic is described using domain terms it is considered detailed otherwise it is considered domain independent.

[Tepf97] describes each technology in relation to Implementation and Domain concepts and suggests a path for developing frameworks based on the topology. The upper arch in *Figure 3* illustrates the suggested path for developing frameworks. The main technologies highlighted in the topology are examined with particular reference their role in framework creation.

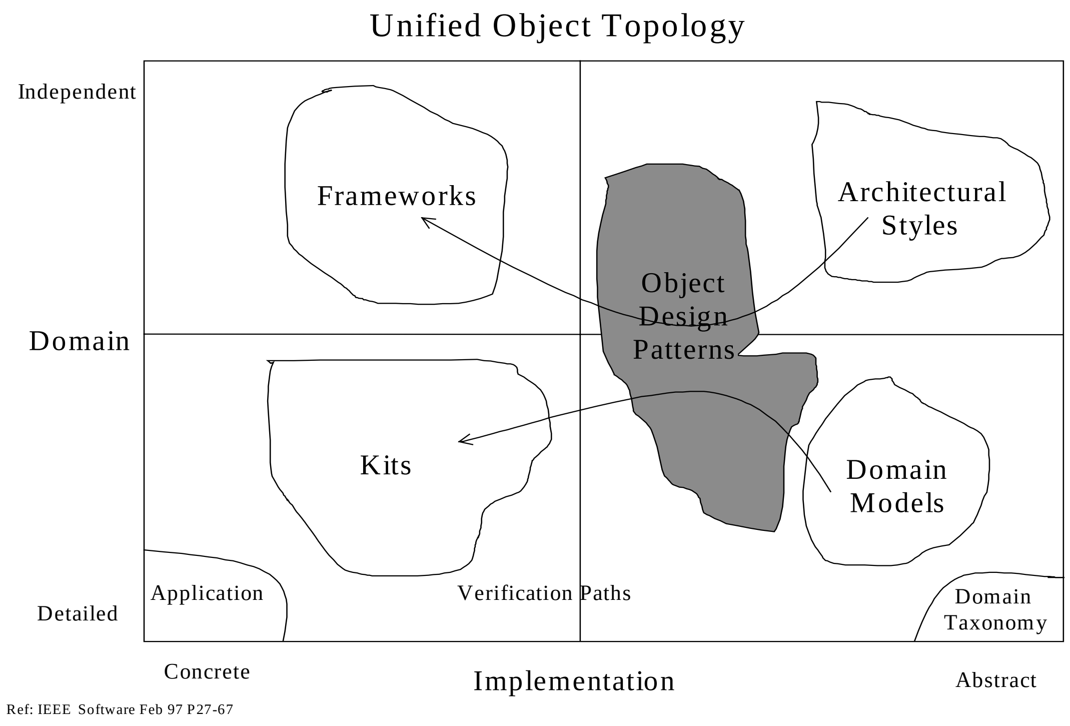


Figure 3: A Unified Object Topology

2.1.1. Background

[Tal94b] describes object-oriented frameworks in the context of other development approaches. It describes how Taligent Inc. has used frameworks to realise the benefits of object technology. A review of this paper illustrates the reasons why object-oriented frameworks have become state of the art in developing modern software systems. The paper begins by charting the key improvements in each of the three major software development approaches, procedural programming, object–oriented programming and framework-oriented programming.

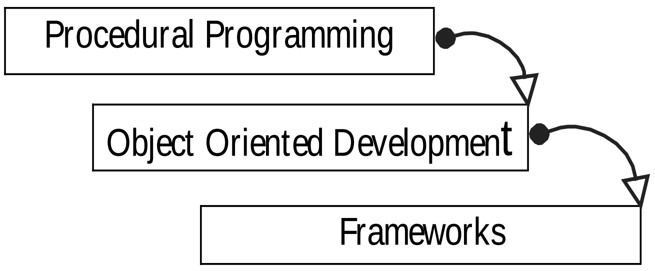


Figure 4: Software development processes

The *procedural programming* approach brought major improvements to software quality through increased clarity and reliability of programs. As demands on software capability grew some limitations of procedural programming became apparent. Most importantly a lack of extensibility became evident. Inflexible interfaces did not allow developers to make selective changes or extend the structure or behaviour of applications. Common functionality was difficult to factor out allowing little re-use in software design. There were problems ensuring that changes inter-operated correctly with other systems that depended on modifications. Minimal software design reuse leads to high maintenance costs. In summary procedural programming did not address large-scale programming issues. It was primarily concerned with low-level algorithm and procedural issues. The object-oriented approach sought to address these issues.

Object-oriented programming (OOP) emphasises the binding of data structures with methods that operate on that data. Both data and methods are stored in objects. Object classes are designed to correspond to the essential features of the problem. Emphasis is placed on effective modeling of solutions to real-world problems. The OOP principals of inheritance, encapsulation and polymorphism allow developers to break problems in smaller and more manageable modules. Encapsulation frees the developer from having to know implementation details. Inheritance allows the developer to derive new subclasses from existing ones and provide hooks for adding new extensions. Polymorphism provides a mechanism for the creation of multiple definitions for functions. These characteristics greatly improve class reuse. New components are added without disturbing existing ones.

OOP permits developers to work at a higher level of abstraction through classes and objects. This gives rise to improved extensibility, flexibility, interoperability and ease of maintenance. Developers design applications by assembling classes and objects and connecting them in a coherent way. The developer is responsible for implementing the structure and flow of control of the application. This is achieved by defining the relationships between the classes and objects.

Frameworks attempt to address this by providing the developer with a predefined structure for an application and freeing them to concentrate on implementation specifics.

2.1.2. Frameworks Definition

Frameworks build on OOP concepts by providing infrastructure and flexibility for deploying object-oriented technology (OOT). Ralph Johnson’s framework definition is a widely accepted one:

“*A framework is a set of classes that embodies an abstract design for solutions to a family of related problems*” [Joh88]

Other definitions offer alternative views of the framework concept:

“*Frameworks are sets of co-operating classes that make up re-usable design for a specific software class providing the entire domain independent infrastructure you need to implement an application.*” [Mel97]

“*A Framework is a set of prefabricated software building blocks that programmers can use, extend, or customise for specific computing solutions*” [TAL94]

"*A framework helps a developer provide solutions for problem domains and better maintain those solutions. It provides a well designed and thought out infrastructure so that when new pieces are created, they can be substituted with minimal impact on other pieces of the framework*" [Nel 94]

The four definitions highlight the main features of frameworks. A framework consists of a set of classes. This set of classes is more than a class library. Relationships exist between the classes that are defined in the library. These relationships model the basic design structure of the framework. Frameworks place strong emphasis on the re-use of both design and code. They are based on a component architecture of prefabricated building blocks that are used to generate an application instance. These features are examined in more detail in following sections.

2.1.3. Software Reuse

The demand for software productivity is ever increasing. Software reuse can help meet this demand by reducing the time to market. The reuse of software components is recognised as an important way to increase productivity in software development. In the past developers have sought to reuse code design through programming experience and examining old code designs. However the reuse of analysis and design can have significantly higher value. The framework concept makes it possible reuse both analysis and design as well as code.

"*An object oriented approach moves much of the software development effort up to the analysis phase of the life cycle. It is sometimes disconcerting to spend more time during analysis and design, but this extra effort is more than compensated by faster and simpler applications. Because the resulting design is cleaner and more adaptable, future changes are much easier*." [Rum91]

2.1.4. Frameworks v Class Libraries

In developing reusable software the aim is to produce extensible software components. Traditionally developers have tried to achieve this through providing domain-specific procedural libraries of functions or reusable class libraries. Domain applications were built using these individual components. When developing an application in this way the developer is responsible for defining the communications between the many small components. "*Class Libraries do not impose a particular design on an application; they just provide functionality that can help the application do its job. The developer must provide the interconnections between the libraries.*" [Lan95]

Figure 5 illustrates the difference between using class libraries and frameworks to build applications. Part A of the figure shows the application developer defining the basic application structure and adding components from the library. A framework is not simply a collection of classes. The wired in interconnections and rich functionality give the developer a basic infrastructure with which to work.

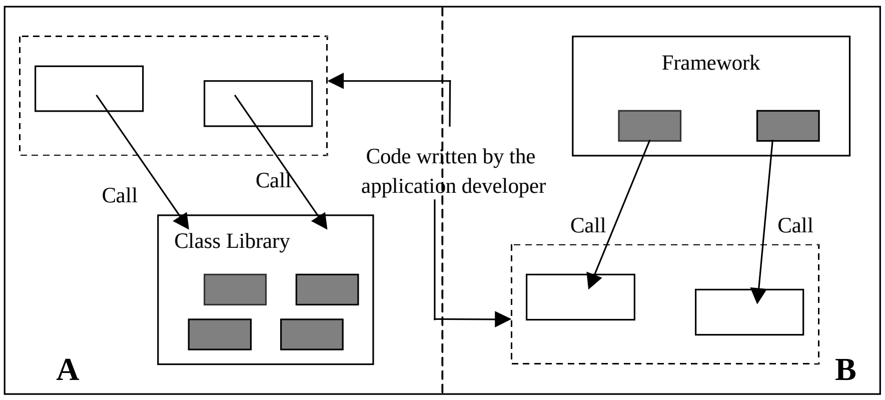


Figure 5: Class libraries v Frameworks

In designing a framework the application designer does not need to know how or when to call each function. The basic application design is defined in the composition of framework classes. Part B of *Figure 5* illustrates the application development process using frameworks. The framework model frees the developer to focus on implementing solutions for application specific problems. The standard code implemented in the framework reduces considerably the level of test and debug necessary in the client application.

2.1.5. Framework development process

Domain expertise is embodied in a framework. It captures the programming expertise necessary to solve a particular class of problem. The problem solving expertise encapsulated in the framework should be independent of both the original problem and future solutions. A well-designed framework should appear to have been designed for each program that uses it.

A major goal of object-orientated design is to produce well-structured software that is both extensible and re-usable. Re-usability is difficult to achieve. Frameworks attempt to address this by placing strong emphasis on software re-usability on a large scale. OOP language support for the properties of inheritance, polymorphism and dynamic binding make frameworks feasible. Inheritance allows several classes to share code. Dynamic binding lets a function call be bound to an object at runtime. Polymorphism allows variables and parameters take on different values and types. [Lan95]

With this in mind "*Building Object Oriented Frameworks*" [Tal94] describes four important considerations for designing frameworks. These are:

Completeness - Frameworks should support features required by clients and provide default implementations and built in functionality where possible. Concrete derivations of abstract classes and default member function implementations help the client understand the framework and let them focus on areas needing customisation.

Flexibility – Abstractions should be applicable to different contexts. The framework must be re-usable.

Extensibility – Hooks should be supplied which allow clients to customise the behaviour of the framework by deriving new classes.

Understandable – Client interactions with the framework should be clear and well documented. Sample applications that demonstrate the use of the framework should be provided.

"*The most profoundly elegant framework will never be reused unless the cost of understanding it and then using its abstractions is lower that the programmers perceived cost of writing them from scratch*" [Boo94]

From a client perspective an easy-to-use framework is the most important consideration. It should perform useful functions with little effort. It is recommended that a framework should work with little or no client code, even if default implementations are only placeholders. It should be possible to get from default behaviour to sophisticated solutions in small incremental steps.

Frameworks have most value when many applications are going to be developed within a specific problem domain. They are designed by generalising from concrete examples. Framework development is therefore an evolutionary process.

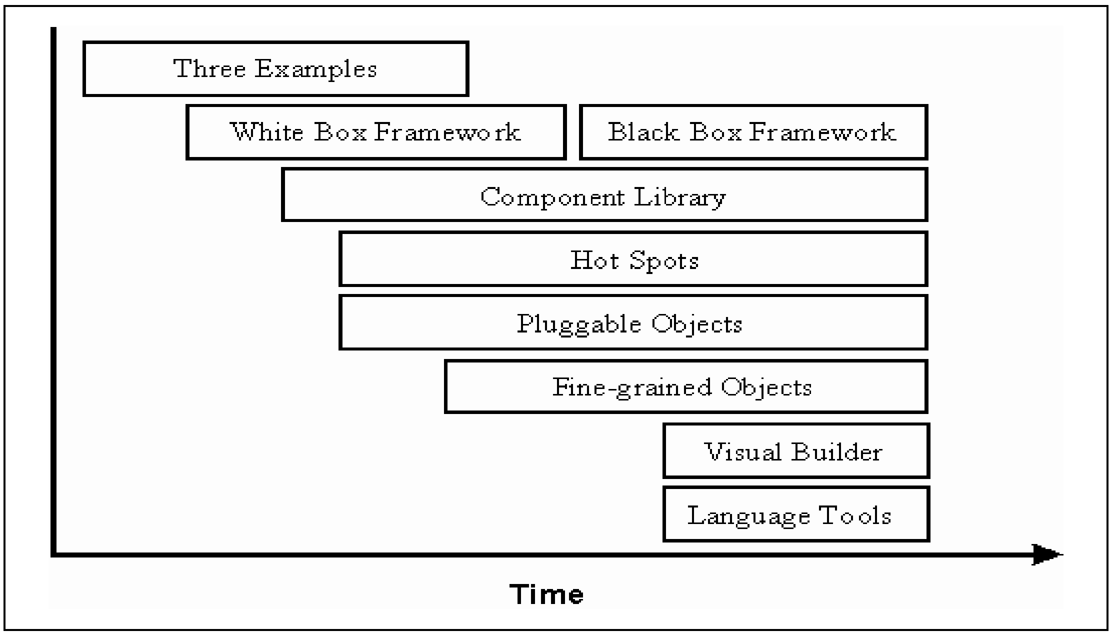


Figure 6: A framework evolution process [John96]

[Rob96] describes such a framework evolution process. The main steps in the process are described in *Figure 6*. The process suggests the analysis of three domain examples from which a white-box framework is created. The ability to generalise for many applications can only come by determining which abstractions are being reused accross applications. The framework gradually evolves into a black-box framework that is composed of discrete components or objects. As the framework stabilises, visual design tools are created to build and customise different versions of the framework. Programming language support features are often added to these tools.

2.1.6. Benefits & Costs

The benefit of using framework technology to develop a set of related applications is generally greater than the cost incurred. The main benefits and costs of using framework technology are described below.

Benefits

- *Better consistency and integration:* Because different applications use the same framework greater consistency is achieved across applications generated from the framework. Additionally application inter-operability is easier to achieve as generated applications have a similar infrastructure.
- *Improved reliability:* Framework refinement is achieved through client reuse of the framework. This improves the inherent reliability of the framework.
- *Reduced maintenance*: Frameworks embody domain expertise that can be leveraged by the developer. Fewer lines of code need testing thereby reducing the maintenance cost.
- *Increased developer productivity*: The framework determines the basic design and structure of the application. Developers are free therefore to concentrate on the unique features required in the application. This leads to an increased developer productivity. Frameworks can dramatically reduce the time to market and are seen as a tool for innovation.

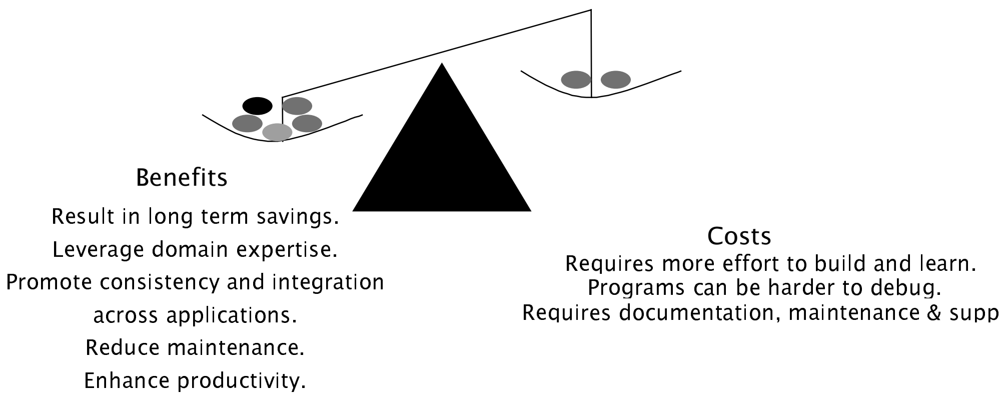

Benefits
Result in long term savings.
Leverage domain expertise.
Promote consistency and integration across applications.
Reduce maintenance.
Enhance productivity.
Costs
Requires more effort to build and learn.
Programs can be harder to debug.
Requires documentation, maintenance & supp

Figure 7: Framweork benifits v costs [Tal94a]

Costs

- *Difficult to build a framework:* Frameworks are difficult to design and build. A designer must find and represent the common functionality that exists in similar domain applications. This design must be capable of being re-used to create customised framework applications. *"Components and architectures do not become reusable by themselves. They must be designed with reuse in mind or redesigned for reuse. Designing for reuse takes longer time* *than designing systems or components without any thoughts of reuse. This extra time must be seen as investment.*” [Lan95]

- *Documentation required:* Frameworks must be documented. If the documentation is unclear or the structure of the framework is incorrect, problems arise when developers try to use it for application construction.

In summary, a well-designed framework should provide a generic application infrastructure that models the application domain accurately. This in turn will provide the developer with opportunities for re-using design and save considerable development time.

2.1.7. Framework types

Frameworks can be variously categorised according to different criteria. Three broad types of Frameworks are described in [Tal94a], application frameworks, domain frameworks and support frameworks. Application frameworks encapsulate expertise applicable to a wide variety of programs. They capture a horizontal slice of functionality that can be applied across many client domains. Graphical user interface (GUI) application frameworks such as Apple’s MacApp or Borland’s OWL are examples of this type of framework. Domain frameworks encapsulate expertise in a particular domain and capture a vertical slice of functionality for a particular client domain. Examples of domain frameworks are manufacturing, data access and multi-media frameworks. Support frameworks provide system-level services such as file access or device drivers. An application developer generally uses the framework directly or uses modifications produced by the systems provider. For the purposes of this dissertation we are concerned primarily with domain frameworks.

Frameworks can be also categorised by how they are used as architecture driven or data driven. This is sometimes referred to as inheritance focused (white box) or composition focused (black box). Architecture driven frameworks use inheritance for customisation whereas data driven frameworks use object composition. Data driven frameworks have the advantage of being easy to use but can limit customisation. Architecture driven frameworks require the developer to undertake additional coding to achieve customisation. A hybrid approach combines an architecture driven base that accommodates extensions and a data driven layer for ease of use. This is the most commonly used approach.

2.1.8. Design Process

Identifying the primary abstractions, defining how clients interact with the framework and implementing, testing, and refining the framework design are the major steps in a framework development process.

Abstractions are identified through domain experience and/or through analysing a range of existing solutions. Common functionality is abstracted out using a bottom up strategy. The responsibilities of the framework itself are determined and the parts with which the client can interact are defined.

The traditional goal of object-oriented design is to find the objects that exist in the problem domain. Identifying this set of objects helped the developer create a structure for the application. Framework design has a higher level emphasis. It is more concerned with finding objects and the relationships between them that represent what is common across several applications. "*With or without case tools, early adopters of the object paradigm focused on the "find-the objects" exercise, deferring or forever losing the systems perspective of interactions between classes or between objects*." [Copl97].

When the basic structure of the framework is in place it is then examined for recurring design patterns. If any patterns are applicable then generic solutions can be applied to refine the basic framework structure.

2.2. Design Patterns

Design patterns play an important role in the generation of frameworks. Two alternative definitions of design patterns are given below.

"*Design Patterns are generic designs to problems that occur often during object oriented design*" [Lan95]

"*A pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces*" [App98]

Christopher Alexander [Alex77] was first to recognise and use patterns as a tool for designing buildings. Strong parallels exist between Alexander's use of architectural patterns in the design of buildings and the software designer's use of object-oriented patterns in the construction of software applications. Prefabricated modules are put together to create a (whole) building/application. The success of the building/application design is not just the sum of its parts but also depends on the relationships between the parts.

"*Alexander found that patterns helped him express the relationships between the parts of the house and the rules that transforms those relationships.*" [Copl97]

In recent years software developers have applied the pattern concept to software construction. Design patterns can help find and represent commonality between applications. Some patterns appear in different guises across several applications making them difficult to recognise. The goal of the software community is to create a body of literature that helps software developers resolve common problems in software development. This is part of an emerging culture which documents and supports sound software design.

Many framework design problems have already been encountered and solved by software designers. These solutions are often described as design patterns. The design pattern solutions are sometimes documented and made available publicly. [Gam94] has produced a seminal work that documents generic software patterns under the categories of creational, structural and behavioural. These patterns solve specific design problems and make the framework design more flexibile, elegant, and ultimately reusable. Creational patterns are concerned with the process of object creation. Structural patterns deal with the composition of objects and classes. Behavioural patterns characterise the ways in which classes or objects interact and distribute responsibility. Each pattern description embodies the essential insight into the solution from which others may learn.

The documentation process for design patterns has been formalised to help in the creation and development of a pattern language, which can be widely accepted and understood. Patterns are uniquely identified by name, have a solution description, and an applicable context.

"*Each pattern is a three part rule, which expresses the relationship between a certain context, a problem, and a solution*" [Alex77].

A pattern should address a recurring phenomenon, generally verified by examining three existing systems. This is referred to as the rule of three. Good patterns solve a problem, are a proven concept, provide an indirect or non-obvious solution to the problem, and describe a relationship for a given context. The Gamma et al work and many subsequent works in this area are an invaluable reference for both experienced and novice software designers.

2.2.1. Pattern Types

Distinctions are sometime drawn between architectural patterns, design patterns and idioms. *Architectural patterns* or architectural frameworks describe a fundamental schema for software systems. These are the high level patterns concerned with relationships between a set of pre-defined subsystems. Subsystem responsibilities are specified and rules and guidelines for organising the relationships are set out. *Design patterns* provide a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context. An *idiom* or coding pattern is a low-level pattern specific to a programming language.

The primary difference between the three types is the level of abstraction and detail. Architectural patterns are concerned with high level strategies and large scale components. They effect the entire structure of a software system. Design patterns apply to subsystems or components and are themselves micro architectures. Idioms relate to programming language paradigms and are the lowest level of object pattern. "*You can only master a language after its idioms become second nature*." [Copl97]

Patterns are sometimes collected and organised in categories within a catalogue. This facilitates the developer in finding a pattern of a particular type. [Gam94] is organised in this way. Such organisation adds structure to the pattern collection.

2.2.2. Pattern Formats

Several formats are used to describe patterns. The "*Gof format*" *and* "*Alexanderian form*" supply headings to make what is known as the "*canonical form*." Patterns using this form are generally specified with the following description headings.

Name:	Single word or phrase to identify the pattern
Problem Definition:	A statement describing the nature of the problem
Context/Preconditions:	Context in which the problem and solution is applicable
Solution Description:	Textual or graphic description of the structure, participants and collaborations that shows how the problem is solved
Examples:	One or more sample applications of the pattern
Post Conditions:	The state or configuration after the pattern has been applied
Rationale:	Can provide insight into deep structures and key mechanisms of the pattern
Related Patterns	Patterns, which are related in a given dcontext.
Known uses	Applications in which the pattern was used.

Table 1: Pattern Format Headings

2.3. Pattern Languages and Architecture

Patterns have a much wider usage and relevance than software construction. They are applied is several contexts including architectural design, process practice and even to marketing strategy. At the most abstract level software development is concerned with the architectural design of a software system. The architectural design of a software system can be compared to the architectural design of the central structure of a building. Patterns can play an important role in defining the architecture of the system. "*To me patterns are a literature that goes beyond documentation. They capture an important structure, a central idea, a key technique long known to expert practitioners. It can be an architectural structure*" [Copl97]

Software architecture can have many different meanings. It is argued that an architecture is a high level representation that determines the structure of a system and the underlying philosophy on which the system is implemented. [Mon97] describes the roles and qualities that a software architecture design should have. The design is at a level of abstraction at which the software developer can reason about the function, performance and reliability of a system. The design abstracts away from the implementation details and has a structure, abstractions for interactions and some global properties. An architectural style is essentially a design language that provides an architect with a vocabulary and framework with which to build design patterns to solve problems. Good software architectures should be resilient and adaptable to change and be living architectures capable of dynamically adapting.

Patterns are used to help define the architecture of a system. There are sets of related patterns that support the construction and evolution of whole architectures. Such a pattern collection is more than a catalogue of patterns. The relationships that exist between patterns are embodied in the collection itself. Patterns languages consist of tightly interwoven and cohesive patterns. "*A pattern language is a set of patterns which are used together to solve a problem*" [Rob96] The pattern language defines a collection of patterns and the rules to combine them into an architectural style.

Alexander developed an architecture, which was based on a directed graph of patterns. He termed this a pattern language and used it to help him design and construct buildings and towns. He applied this to large-scale problems by breaking them into smaller problems and using pattern graphs to connect them. The philosophical base of his architecture was embodied in his selection of patterns. Alexander implemented his architectural designs by creating real buildings from which he documented, reviewed, and ultimately refined his designs. The notion of a philosophical base in the development of a pattern language is somewhat elusive. Alexander described this as the "quality without a name". He believed certain qualities are aesthetically beautiful in a timeless and universally accepted way. It is these qualities that makes a structure "whole" and "alive".

Over the past ten years software designers have worked with Alexander's ideas to develop software pattern languages. "*To effectively build large object systems that realise a philosophy, we must use a pattern language. First however we must develop effective pattern languages*" [Kert97]. Kerth et al. describes three approaches i*ntrospection, artifactual* and *sociological* that are used for recognising recurrent situations in design. *Introspection* is defined as searching for an individual architectural style. People reflect on and relate to their own experience of building systems. An artifactual approach examines systems developed by other teams and is a more objective approach. The sociological approach studies how people involved in building similar systems discover recurring problems in system design. This latter approach has not been widely researched. Pattern languages generally evolve over time and should be carefully evaluated and documented. The expert knowledge embodied in the language must also be accessible to developers. Pattern languages represent a future challenge for software developers.

2.4. Summary

A brief history of software design and development is described in the first section. The review broadly covers object technologies and current methods used in software design. The key aspects of the design and development process of object-oriented frameworks are examined. Object-oriented design patterns play an important role in the design and practical realisation of a framework. A discussion of pattern concepts is included in the review. Software architecture concepts and the development of pattern languages are described in the concluding part of the review.

3. NETWORK PROTOCOLS & CONFORMANCE TESTING

The first part of this chapter examines the nature and design of communication protocols. Some of the important developments in communication protocol history are highlighted. The main characteristics of network protocols are described. Two important types of network protocol are Open Systems Interconnection (OSI) and Internet protocols. Much work has been completed on multi-protocol conformance testing of OSI protocols and this has relevance to the creation of a conformance testing application for multiple Internet protocols. The design and characteristics of the two types of protocol are compared and contrasted. Aspects of protocol conformance testing are examined in the final part of the review. These include test suite structure, test selection, and automated test generation.

3.1. Protocol Background

Communication protocols are rules that govern the communication between different components in a distributed environment. Early protocols were designed to transmit information over long distances. Among the first methods used for long distance communication was fire signals. The number of messages that could be sent by a single fire signal was, however, limited. The use of an array of five torch signals to represent the alphabet was a significant 2nd century B.C. improvement. In designing early protocols the main problem encountered was the difficulty in defining a set of signals which were clearly understood and unambiguous in meaning. This is still a major design issue in modern protocol development.

Not until the 18th Century with the introduction of the optical telegraph and later the electromagnetic needle telegraph was the basic design of protocols improved. By 1875 almost 200,000 miles of telegraph line were in operation. This became the basis for the development of early network protocols. Morse code signaling used dots and dashes to send and receive messages in a binary format. Eventually switching networks such as ARPA required the development of advanced network protocols. Today many different network protocols are defined for a wide range of applications. The need for standardisation of multiple protocols gave rise to standards organisations and bodies that assumed responsibility for specifying protocols. The Internet Architecture Board and OSI are two such organisations. They control the specification process for Internet protocols and OSI protocols respectively. Both protocol types are compared and contrasted below. They are also discussed with respect to conformance testing in a later section of this chapter.

3.2. Protocol Design and Specification

A protocol is a well-defined set of rules for exchange of information between computer systems. Assume we have two computer systems, *A* and *B*. All rules, formats and procedures that have been agreed upon between *A* and *B* are collectively called a protocol. To exchange control information between *A* and *B* the channel must be a two-way one. Control information typically includes procedures for start, suspend, resume, and conclude transmission. Transmission control error information is also sent between the systems. Protocols formalise the interactions between the systems by standardising the use of the communications channel.

A protocol typically contains agreement on the methods used for:

- Initiation and termination of data exchanges
- Synchronisation of senders and receivers
- Detection and correction of transmission errors
- Formatting and encoding of data

These methods can be defined at various levels of abstraction. At the lowest level, a format definition might consist of methods for encoding bits with analog electric signals. A level up might consist of methods for encoding individual characters into bit patterns. At higher levels character codes are grouped into message fields and message fields into packets or frames with a specific meaning or structure. Layered architectures that have multiple levels of abstraction are sometimes adopted for complex protocols.

The essential elements of a protocol definition are contained in the protocol's specification document. The specification is the basis for reliable protocol design. It outlines the structure and behaviour of the protocol and makes explicit all assumptions. Protocol engineering is concerned with developing protocol specifications and implementations, and performing validation. A protocol specification consists of five elements: the service to be provided, the assumptions about the environment, a vocabulary of messages, the method of encoding messages, and a set of procedure rules for message exchange [Holz91]. These basic building blocks of the specification are discussed in brief.

Service specification: This describes the service to be provided by the protocol. Text file transfer as a sequence of characters across data lines assuming protection against transmission errors is an example of a service specification.

Environment assumptions: This describes the environment in which the protocol is to be executed. The assumptions could, for example, minimally include the existence of two users and a reliable communication channel.

Protocol vocabulary: The protocol vocabulary defines the distinct message types. The basic IIOP message types include Request, LocateRequest, CancelRequest, Reply, LocateReply, CloseConnection, and MessageError. Taken together these form the message vocabulary of IIOP.

Message Format: These are the structures that are used to encode the protocol message vocabulary. Three low-level formats include bit oriented, character oriented and byte-count oriented. A bit oriented protocol transmits data as a stream of bits. Pre-defined bit sequences act as flags to signal the start and end of messages. Character oriented protocols enforce some minimum structure on the bit stream. For example, if the number of bits per character is fixed at *n* bits then all communication takes place in multiples of *n* bits.

Higher-level data formatting methods can be built on these lower level structures. For example if flow control techniques are added to detect the loss or re-ordering of frames, a sequence number field could be appended to the message. Similarly if more than one type of message is used, some indication of this must be included in the message. The detailed structure of the IIOP message vocabulary is illustrated in page 3-11.

Procedure rules: Procedure rules determine the order and meaning of possible message sequences. Sequences can be interpreted concurrently by a number of interacting processes. These processes can have different interpretations at various time intervals. Because of the many different time intervals that can take place, it is not always possible to reproduce protocol behaviour. To check the correctness of design some formal method of reasoning about protocol behaviour is preferred. Protocol behaviour is most often represented in a formal way as a finite state machine (FSM). The problem of devising a complete set of rules which is unambiguous for the exchange of information is the most difficult problem in protocol design. Two of the main types of protocol design are Internet and OSI protocols. These are discussed in the next section.

3.3. OSI v Internet Protocols

If examined in a top down fashion the architecture and goals of OSI and Intenet protocols are essentially the same. The principles and techniques used in each are very similar. Both have an underlying philosophy of promoting an open systems strategy. The fact that TCP/IP is older has had a strong influence on the development of OSI. The public availability of research on real world TCP/IP applications is also a major factor in that regard. To a lesser extent TCP/IP itself has been influenced by OSI. An example of this is the use of Abstract Syntax Notation (ASN.1) in defining the management information base (MIB) for the Simple Network Management Protocol (SNMP). To avoid overlap and make the best use of resources, an increasing level of cooperation between both bodies and their protocols is evident.

A weakness of OSI is that it seeks to be all things to all people. Many international standards bodies and committees are involved in the OSI standards process. A large number of steps and revisions are required to complete the specification of a new protocol. The Internet uses a public *Request for Comments* (RFC's) specification process. It is much simpler and more efficient than the equivalent OSI process. In addition, anyone with Internet access can participate in the process.

The Internet RFC process leads to greater flexibility and speed in the design of Internet protocols. A downside however is the lack of commonality between Internet protocols. OSI's more formal process facilitates a level of design which can encompass common structures and behaviour across protocols.

The OSI specification IS 9646 titled *“OSI – Conformance Testing Methodology and Framework*” is a relevant example of this. All OSI based standards are now written in accordance with the requirements of IS 9646. No such equivalent is specified for Internet protocols. More often than not, testing requirements are given little consideration during design and specification of Internet protocols. An example of this is highlighted in the OMG TSIG White Paper [1] “*The OMG RFP process gives cursory attention to testing*.” Much of the work completed for IS 9646 in the area of test methods, test suite structure and testing process is applicable to the design of a common method or approach to testing multiple Internet protocols. These design issues are discussed in more detail in the sections 3.5.1 to 3.5.3.

3.4. General Testing Concepts

Verification and Validation (V&V) is the process that ensures software conforms to its specification and meets the needs of the software customer. A software system should be verified and validated at analysis, design, implementation, and test development stages. The difference between validation and verification is summarised by Boehm [Boe79]:

- Verification: Are we building the product right?
- Validation: Are we building the right product?

Verification is checking that a program conforms to its specification. Validation involves checking that the application, as implemented, meets the expectation of the customer.

Both static and dynamic techniques of system analysis and checking are used in the V&V process. Static techniques are used for analysis and checking representations of the system such as requirement documents, design models and program code. Static techniques check for correspondence between a program and its specification. It does not however check that a system is operationally correct. Static techniques are useful for identifying errors in program logic.

Dynamic techniques are applied only when an executable program is available. They are used to exercise the application with real data inputs and check that the generated data outputs conform to the expected outputs. Dynamic testing techniques can be applied to test applications for performance, reliability or conformance to specification. Application performance can be judged using a statistical

[1] The Object Management Group (OMG) is responsible for the specification of CORBA related Internet protocols such as IIOP. The Test Special Interest Group is part of OMG.

testing approach. This approach is also used to test for reliability when combined with a reliability growth model [Som96]. Statistical testing is generally based on patterns of user input.

Defect testing is a form of dynamic testing that is used to check that an application conforms to its specification. A successful defect test is one that causes the system to perform incorrectly and demonstrates the presence of program faults. It does not however infer the absence of program faults or that a program completely conforms to its specification. Defect testing is generally not exhaustive in that sense. Test selection is therefore based on defining a subset of all possible tests.

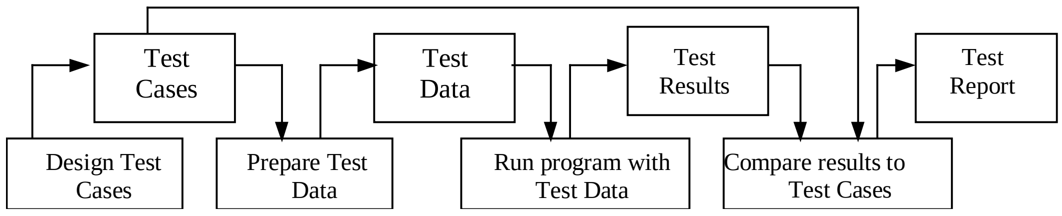


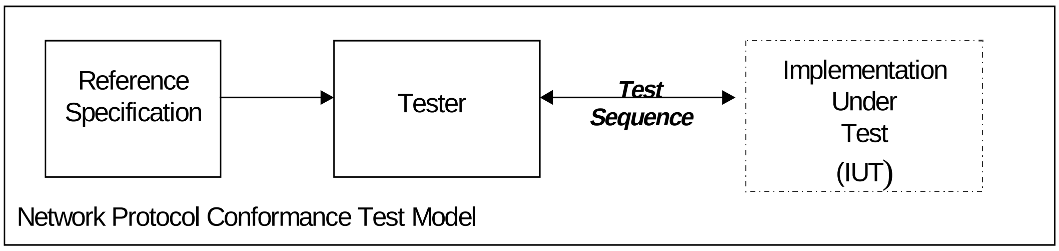
Figure 8: Defect Testing Process [Som96]

Dependent on which type of testing is required, conformance, performance or reliability, a test data set is generated. There is generally some overlap between such test data sets. Some indication of performance can be inferred from conformance testing and a level of conformance can be inferred from statistical testing. *Figure 8* shows the model for defect testing as described in [Som96]. In this model test data and test cases are distinguished. Test data are the inputs used to test the system. Test cases are input and output specifications and include a statement of function under test. The program is run using the test data and test results are compared against the test case output specifications. The ipTF testing process is based on this model of testing.

3.5. Protocol Conformance Testing

When a protocol implementation is tested against the protocol specification to ensure compatibility with other implementations of the protocol this is referred to as ***protocol conformance testing***. Testing a protocol for implementation specifics, reliability or performance is referred to as ***implementation assessment***. In this dissertation we are primarily concerned with protocol conformance testing.

A conformance test is used to check that the external behaviour of a given implementation of a protocol is equivalent to its formal specification [Holz91]. A validation test checks that the specification of the protocol is logically correct. Conformance testing will not highlight specification design errors, only implementation errors. For instance a conformance test will fail only if the implementation and specification differ. *Figure 9* describes a general model used for protocol conformance testing. The tester derives test sequences from the reference specification. The test sequences are then applied to the running implementation (IUT).



Reference Specification
Tester
Test Sequence
Implementation Under Test (IUT)
Network Protocol Conformance Test Model

Figure 9: Conformance Testing

Many protocols can be specified as a finite state machine. They have a limited number of states and a finite set of inputs and outputs. Given a reference specification and an implementation of the protocol, input sequences are applied to test the protocol for conformance to its specification. The implementation effectively acts as a black-box that accepts inputs and produces outputs in response. The resulting outputs are checked against the outputs as prescribed by the formal specification.

"Verification, by the standard IEEE definition, is a way of assessing whether the input/output pairs are correct" [Voas94]

This type of testing is referred to as black-box testing. In contrast, white-box testing requires that test cases be derived from knowledge of the program's internal structure or implementation. Protocol conformance testing is mainly based on the black-box approach.

The essential elements of a protocol conformance testing application are defined in IS 9646. These elements are applicable to protocol conformance testing in general. Test methods, an abstract test suite structure, a results evaluation process and a method for test case selection are all described by IS 9646. We examine each with respect to the development of an Internet protocol testing application or framework.

3.5.1. Test Methods

Four test methods are described by IS 9646, *local*, *remote*, *distributed* and *co-ordinated***.** These names are not very descriptive and do not indicate the nature of the test method. The *local* method requires that the IUT be built into the test application. The IUT is surrounded by an *Upper Tester* at the top of the layer under test and a *Lower Tester* which is underneath the layer under test. Test co-ordination procedures co-ordinate the actions of the upper and lower testers. The other three methods are all remote in the sense that the test application is external (remote) to the IUT. *Figure 10* shows an example of the remote test method as defined in IS 9646. It makes no assumptions about the internal design of the System Under Test (SUT) or the IUT. It effectively treats the SUT as a black-box. The *distributed* and *co-ordinated* methods are more complex in that they permit the tester to have control over events within the SUT. In the *co-ordinated* method, for example, test management

PDUs are sent to the SUT, which tell the SUT what actions to perform and what events to observe and report.

These methods are generic for all network protocols and equally relevant for testing Internet protocols. The remote methods are appropriate for a client/server protocol architecture that has remotely distributed clients and servers.

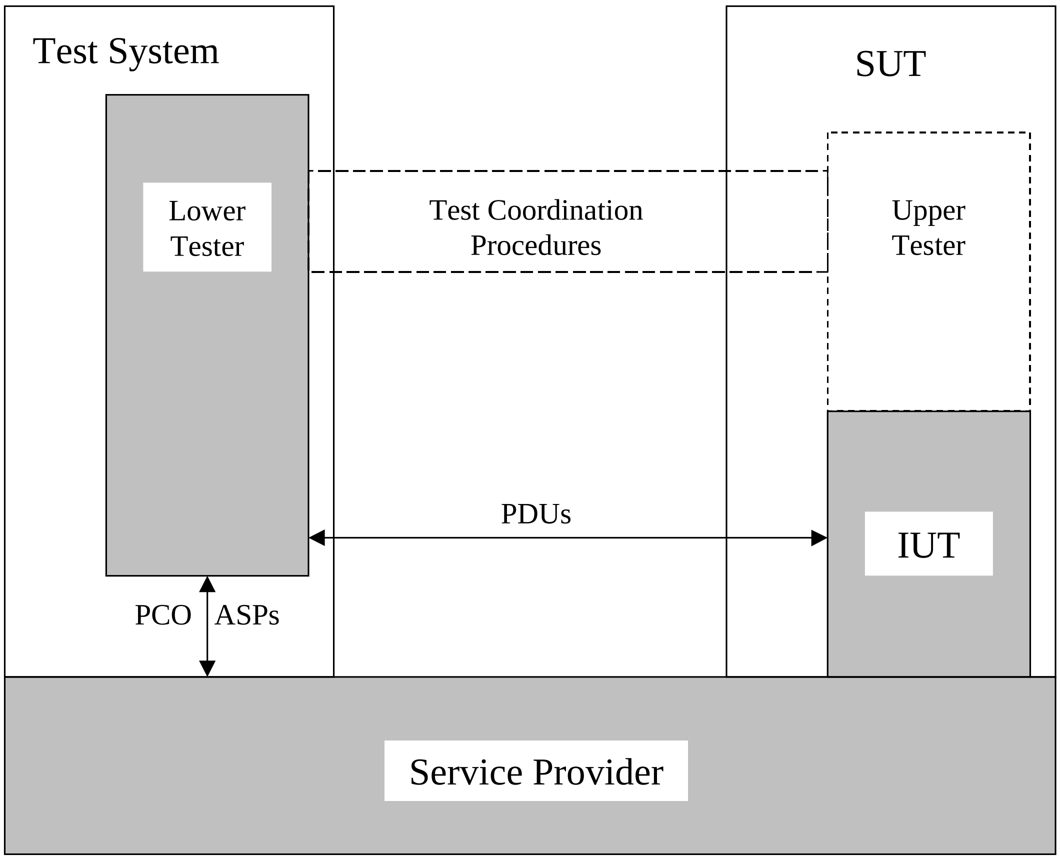


Figure 10: Remote Test Method [Knig93]

3.5.2. Test Suite Structure

A conformance protocol test passes if all observed outputs match the outputs which are prescribed by the formal specification. The series of input sequences used to exercise the protocol specification is known as the conformance test suite.

ISO 9646 defines an abstract test suite structure, which can be used in testing any ISO protocol. The test structure is defined in a top down hierarchical way. The containment hierarchy is Test Suite, Test Group, Test Case, Test Step and Test Event. A Test Case is the fundamental building block of the Test Suite. It performs a test that corresponds to a particular feature of the protocol under test. A test group is simply a particular selection of test cases. A Test Event is the smallest indivisible test unit such as sending or receiving of a *Protocol Data Unit* (PDU). A Test Step is a sequence of Test

Events. A sequence of test events is used for transitioning between protocol states. An equivalent test suite structure is valid for testing a range of Internet protocols. The test suite structure defined for use in ipTF has much in common with the IS 9646 structure. The design elements of the ipTF test suite structure are described in chapter 4 of this document.

3.5.3. Test Case Selection

Suitable test cases/data must be selected to test the implementation for conformance. Given that protocol testing is generally based on the black-box approach the protocol specification has a strong influence on the selection of test cases. The method for developing test cases is to a large degree dependent on the specification. Internet protocols are generally specified in natural language. This can lead to ambiguities and makes it difficult to check for completeness and correctness. Formal specification methods are commonly applied in protocol design. These include finite state machine (FSM) models, petri nets, formal grammars and high level programming languages. The FSM model is most often used for defining protocol specifications and for that reason most work on protocol testing is based on FSM models.

To fully test a protocol implementation, a series of sequences must be defined to form a test suite. This requires generating a set of sequences to fully test all functions described in the specification and which can also establish that the implementation rejects invalid inputs. It is almost impossible to check an unknown implementation for all possible behaviours simply by probing it and observing its responses. Normally a representative subset of sequences is chosen. [Knig93] suggests a test suite should include tests that cover the following generic elements of protocols:

- all the mandatory requirements
- all the optional requirements
- all messages which can be legitimately sent to the IUT
- all messages which can be legitimately received by the IUT
- particular protocol states
- time critical operations
- parameter variations
- invalid syntax or semantically incorrect events

A range of tests based on the above would provide good level of coverage for conformance testing. If however a protocol is specified using a deterministic FSM all possible protocol states and input combinations can be checked. A sample algorithm for an implementation with i states and j inputs might appear as follows:

```
initialise implementation
For every combination of state i and input j,
   Set state(i)
   Send input(j)
   Receive output(x)
   Verify output(x)
End for
```

A representative subset of all possible test sequences is however generally sufficient to cover conformance testing of protocols.

Testing for faults in a software system is another means of testing the system for conformance. Fault-free software means software which conforms to its specification. [Mill96] describes the concepts of the fault model and fault coverage of conformance test sequences for communication protocols specified as FSMs. The concepts of Unique Input Output (UIO) sequences and distinguishing sequences in FSMs are the basis for algorithms that generate test sequences which can fully verify the IUT.

3.5.4. Automated Test Case Generation

Automated generation of test cases can greatly improve the efficiency of any test application. A test sequence that has the potential to uncover many faults is better than one that can only uncover a few faults. It is virtually impossible to establish in advance how many faults a given test sequence can potentially uncover. Much research work has been completed in this area. Two papers worth examination are by Korel et al [Kore90] [Kore96]. Methods and algorithms are suggested to achieve automated test data generation. The approach is based on the use of test adequacy criteria to distinguish good sequences from bad. Based on this paradigm algorithms have been developed to automate the generation of test sequences.

3.6. IIOP & Telnet (Internet Protocols)

IIOP and Telnet Internet protocols are used as sample protocols in the testing framework. A brief description of the main aspects of these protocols is given for reference. Telnet was originally designed to work between any host and any terminal. It is specified in RFC 854 [Post83]. The IIOP protocol is specified by the OMG [Corba98].

3.6.1. Telnet Protocol

Telnet is a simple remote terminal protocol. A Telnet client establishes a TCP connection to a remote login server. Keystrokes from the client terminal are then passed to the server’s remote virtual terminal as if they were being typed at the local system. Information is sent back from the server to the client’s terminal. An Internet protocol (IP) address and port number identify the remote system. Telnet allows clients and servers to negotiate options, for example, to determine the format of data being sent across the wire. (7 bit ASCII or 8 Bit ASCII). Many Telnet implementations exist and it is relatively simple to implement.

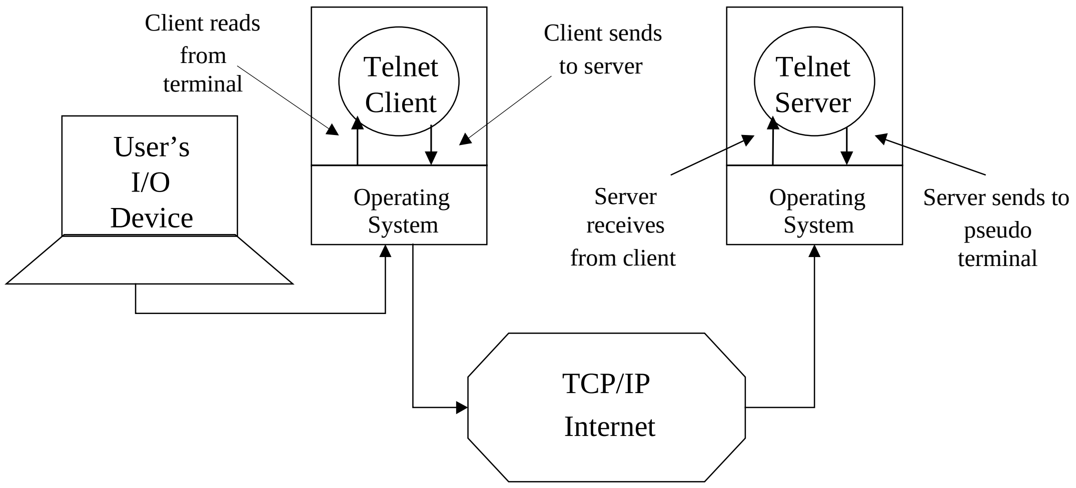


Figure 11: Telnet protocol client to server data path [Comer95]

Figure 11 describes the basic structure of a Telnet client and server implementation. It shows the data path of a remote terminal session as it travels from the user's keyboard to the remote operating system. When a user invokes the Telnet client it establishes a connection to the server. When the connection is established the client accepts keystrokes from the user's keyboard and sends them to the server. The server accepts the connection and passes the client keystrokes to the local operating system. The term *pseudo terminal* describes the operating system's entry point that allows the Telnet server to transfer characters to the operating system as if it were a local terminal. Output travels back from the server to the client over the same path. Usually a master server process waits for connections and creates a new slave to handle each connection. A full discussion of Telnet is available in [Comer95] and [Stev94]. Telnet was chosen as the initial sample protocol for ipTF development, because of its simplicity.

3.6.2. IIOP Protocol

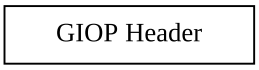
The OMG has defined the General Inter ORB Protocol (GIOP) protocol as the standard for communication between two independent CORBA ORB implementations. The goal of GIOP is to allow two independent ORB implementations communicate while still allowing for the creation of flexible ORB implementations. The GIOP specification does not specify a particular transport layer to be used. It states only that the transport layer must be connection oriented. The OMG has defined a specialisation of GIOP called IIOP for use in an Internet Environment. IIOP uses TCP/IP as its transport layer.

Encoding: GIOP defines a Common Data Representation (CDR) transfer syntax, which is used as the coding format for all IDL data types. The sender is responsible for determining the byte order and encoding of messages in CDR format. The receiver decodes the CDR message using the correct byte order.

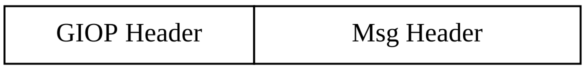
IOR & Connections: Objects are identified in IIOP using Interoperable Object References (IOR's). An IOR reference encodes the host name and port number of the remote server that holds the object and an object reference, which is managed internally by the ORB.

Message Formats: GIOP defines seven message types Request, Reply, CancelRequest, LocateRequest, LocateReply, CloseConnection and MessageError. All GIOP/IIOP messages have a GIOP Header. CloseConnection and MessageError contain only a GIOP Header, CancelRequest and LocateRequest have, in addition a message header and Request, Reply and LocateReply and Fragment have a GIOP Header, Message Header and a message body.

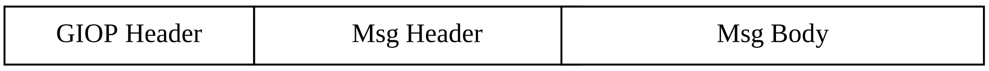
CloseConnection, MessageError



CancelRequest, LocateRequest



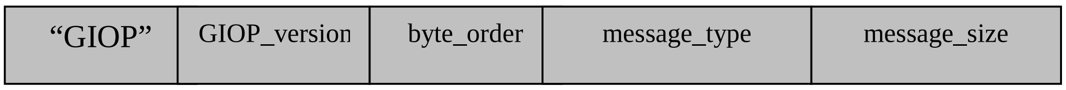
Request, Reply, LocateReply Fragment



3.6.3. IIOP message structure detail

In this section the structure of IIOP messages is examined in more detail. The GIOP header is common to all IIOP messages. The first field identifies this as a GIOP message and is always 4 characters "GIOP" (upper case), encoded in ISO Latin-1. GIOP_version contains the version number of GIOP being used. The field byte_order indicates the byte ordering used in subsequent elements of the message. The message type (Request, LocateRequest, CancelRequest, Reply, LocateReply, CloseConnection and MessageError) are indicated the message_type field. The size in octets of the message excluding this header is contained in the message_size field. The GIOP header field structure is illustrated below.

GIOP header fields



A *Request* message is composed of a GIOP message header, a request header and the request body. Service_context contains ORB service data being passed by the client to the server. The request_id is used to associate request messages with reply messages. TRUE is set in response_expected field if a reply messages is expected. The object_key identifies the object which is the target of the invocation.

The operation field holds an OMG IDL identifier for the operation being invoked. The request body contains all *in* and *out* parameters specified in the operation's OMG IDL definition.

Request message fields

GIOP Header	service_ context	request_ id	response_ expected	object _key	operation	request_ principal	**Request Body**

A *Reply* message consists of a GIOP header, a ReplyHeader structure and the reply body. The service_context contains ORB service data being passes from server to client. The request_id is used to associate requests with replies. reply_status indicates the completion status of the associated request. If the reply status is NO_EXCEPTION, the reply body will contain any operation return values and possibly any *inout* or *out* parameters of OMG IDL defined operations. If the reply_status is USER_EXCEPTION or SYSTEM_EXCEPTION the reply body contains the exception that was raised by the operation. If the reply_status is LOCATION_FORWARD, the reply body contains an Interoperable Object Reference (IOR) which the client can use to re-send the original request.

Reply message fields

GIOP Header	service_ context	request_id	reply_status	**Reply Body**

The *LocateRequest* message can be sent from a client to a server to determine a) if the object reference is correct b) if the server is capable of receiving a request directly for that object reference, and c) to find to what address requests for that object reference should be sent. The *LocateRequest* message is composed of the GIOP header and a LocateRequest header. The request id is used to associate LocateRequest messages with LocateReply messages. The object_key field identifies the object being located.

LocateRequest

GIOP Header	request_id	object_key

CancelRequest messages are sent from clients to servers to notify the server that the client no longer expects a reply for a pending Request or LocateRequest message. It is composed of a GIOP header and a CancelRequest header. The request_id field identifies the Request or LocateRequest message to which the cancel applies.

CancelRequest

GIOP Header	request_id

LocateReply messages are sent from servers to clients in response to LocateRequest messages. They consist of a GIOP header, LocateReply header and the locate reply body. The request_id field value matches the original LocateRequest request_id. The locate_status determines if a locate reply body exists. If the status is UNKNOWN_OBJECT no locate reply body exists. OBJECT_HERE indicates the server can receive requests directly for this object and no body exists. If the locate_status value is OBJECT_FORWARD the body contains an object reference (IOR).

LocateReply

GIOP Header	request_id	locate_status	IOR

Both CloseConnection and MessageError messages contain only the GIOP header. The message_type field in the GIOP header identifies these message as a CloseConnection or MessageError message.

CloseConnection & MessageError (GIOP Header Only)

"GIOP"	Version	Byte Order	Close Connection Message Error	Message Size

3.7. Summary

The main characteristics of network protocols were reviewed in this chapter. Special reference was made to both Internet and OSI protocol characteristics. The OSI IS 9646 standard provides a useful model for testing of network protocols. Test methods, test suite structure, test selection, and automated test generation concepts were discussed. These concepts are central to the creation of a network protocol test application or framework. The two Internet protocols Telnet and IIOP, which are used in the ipTF implementation, are described in the final part of the chapter.

4. DESIGN

The goal of this dissertation is to examine the feasibility of applying a common approach or method to testing multiple Internet Protocols. Framework technology was chosen as a design methodology to encapsulate this application design problem. A framework provides a basic solution to a class of problems. It should accurately model the problem domain and result in a re-usable software design and a component architecture. The problem to be addressed by this framework is the design of a common solution for testing a wide range of Internet protocols.

The analysis and design phase of the ipTF project is the subject of this chapter. To design a framework it is recommended that three applications be analysed (rule of three). This analysis helps the developer find common functionality that is inherent in the range of similar applications. An SMTP testing application developed by Lotus International was selected as one such application for analysis. The application tests a Lotus SMTP implementation for conformance to international standards. The design and implementation was fully documented and available for analysis. A sample Java Beans Telnet implementation was also used for analysis. This implementation was chosen because Telnet is a simple Internet protocol and the implementation was written in Java, the same language as the framework itself.

Conduits+ [Hüni95] is a framework that been used to reduce the complexity of network software and makes it easier to extend or modify network protocols. The initial framework design started out as white-box but gradually developed through the application of design patterns into a black box framework. The development process was followed for the ipTF. *Conduits+* also provided good examples of how protocol structure and behavior could be represented in a framework. These concepts needed to be addressed in the ipTF design. *A framework-Based approach to the Development of Network-Aware Applications* [Boll98] deals with issues of protocol data preparation and data transmission. This provided a useful guide for similar issues addressed in the ipTF.

The OSI IS 9646 specification (a methodology and framework for testing of ISO protocols) also had significant influence on the design. Elements of the structure and design of a multi-protocol testing application are outlined in the IS 9646 specification. Test methods, test suite structure, test case selection and the process of protocol conformance testing were relevant aspects outlined in this specification. [Boch94] describes the specific nature of protocol testing and the general methods used for testing protocol implementations. Some features of the distributed testing method described in this paper were included in the ipTF design.

Two sample Internet protocols (Telnet and IIOP) were chosen as a basis for the framework design and implementation. Telnet was selected because of its simple design and ease of implementation. IIOP represented a newer and more complex protocol. These protocols were significantly different each having a unique syntax and semantics. A framework capable of generating test applications for these two quite different Internet protocols would provide strong evidence that a common approach for testing multiple Internet protocols was feasible. In addition this would also support the use of framework technology as a suitable design methodology.

A framework relies on abstract classes to generate a basic design. The relationships between a framework's abstract classes and their interfaces are the key components in the design. Clients take the basic structure and extend it by implementing concrete methods for the abstract methods defined in the framework. The approach taken to the design of the ipTF was to define a set of abstract classes that encapsulated the common design structure and behavior of a protocol testing application. A concrete realisation of the abstract classes defined was first implemented to create a Telnet test application. Design patterns were then applied to the framework. Patterns for object creation (abstract factory), algorithm encapsulation (strategy) and interface adatpion (adapter) were used to resolve these design issues. All patterns used were taken from [Gam94]. The framework design was remodeled a second time to include the pattern solutions and finally an IIOP protocol testing implementation was added. The main elements of this design and the process used to achieve it are described in more detail in the following sections.

Design Process

The general process used for analysis and design of the ipTF is illustrated in *Figure 12*. Domain analysis was concerned with acquiring background knowledge on conformance testing, network protocols, and frameworks and design pattern.

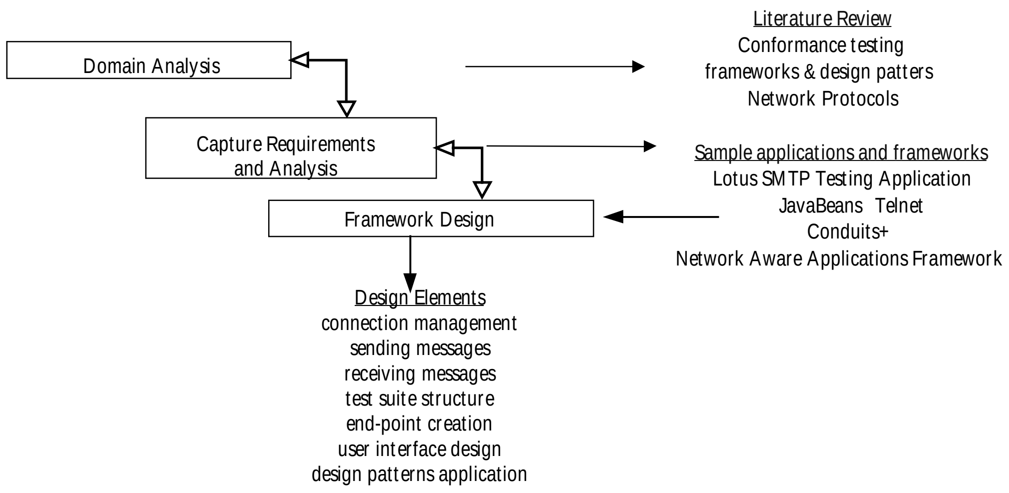


Figure 12: ipTF Design Process

A literature survey of these areas was completed to help establish this background information. The capturing of requirements and analysis is the second process phase illustrated in *Figure 12*. This phase involved analysis of existing protocol testing applications, examining features of similar frameworks and the selection of an appropriate framework development environment. Common structure and behavior across protocols was evident from the analysis. The main features implemented in the design phase were connection management, sending and receiving of messages, test suite design, protocol endpoint creation, and design of the applications user interface. These elements are shown as the output of the design phase in *Figure 12.*

General Testing Model

The first design issue addressed was deciding on a basic testing model to be used for conformance checking. The protocol implementation to be tested for conformance is known as the *Implementation Under Test* (IUT). Two different approaches to testing the IUT were evident from the analysis of existing systems. The first approach (adopted by the Lotus SMTP test application) implements relevant parts of the protocol to effect tests against the IUT. The alternative approach is testing the IUT against a reference implementation of the protocol. This is achieved by accessing the *application programming interface* (API) of the IUT. The API must provide functionality for creating endpoints, establishing connections and sending and receiving massages to and from the (usually remote) reference server.

If used in a framework context the first model would require part implementation of many different Internet protocols within the framework. This latter model was considered more suitable because of the clear separation between protocol design and protocol testing. In this model the test application developer need not be concerned with protocol implementation details. Access to the API of the IUT is all that is required. *Figure 13* illustrates the general testing model used in the ipTF. This model is based on the second model described above.

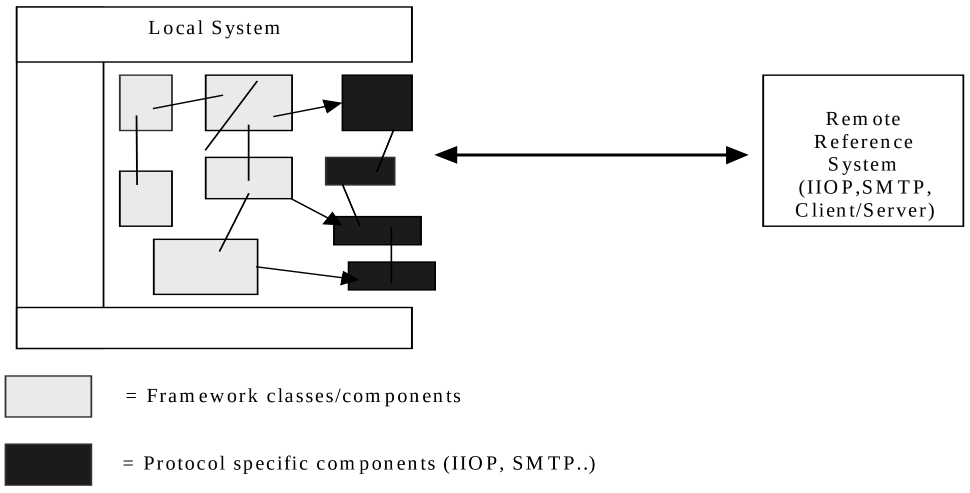

Local System
Remote Reference System (IIOP,SMTP, Client/Server)
= Framework classes/components
= Protocol specific components (IIOP, SMTP..)

Figure 13: ipTF testing model

This model shows the framework classes or components (lightly shaded blocks) that form the basic design structure of the protocol testing application. In addition they provide a set of abstract interfaces for implementing the protocol specific parts. These components are common for all protocol implementations being tested. Protocol specific components (darkly shaded blocks) must implement the framework's interfaces. These components encapsulate the implementation of the protocol client end-points, connection management, and message sending and receiving functionality etc. The two sets of components combine to form an application that tests the protocol implementation against a remote reference server implementation.

It is important to note that a reference implementation should be a valid implementation of the specification. In many cases a reference implementation of the protocol is not available from the relevant protocol standards body. In such cases an implementation from one of the main protocol vendors is used as a de-facto reference implementation.

Message format and protocol behavior

Two central features that any Internet protocol testing application must model are the format of messages and protocol behavior. In the ipTF design the abstract class *TestMsg* represents message data. This abstraction can be used to represent any type Internet protocol message. An instance of the *TestMsg* class is composed of message elements. The semantics and syntax of the message (elements) is determined by the protocol at runtime. A message is built using *addElement()* to add message element objects to message's element vector as required. A TestMsg is composed of an element vector component and a MsgReader component. This is illustrated in Figure 14. The MsgReader component is used by the application to read the message elements from a sequence file.

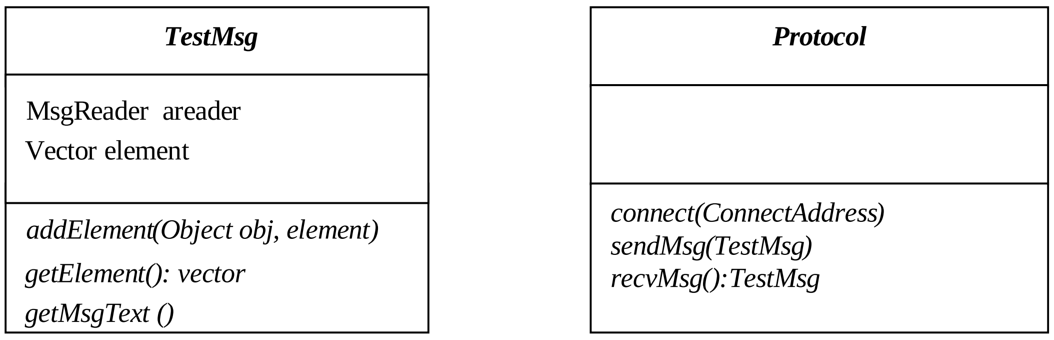


Figure 14: TestMsg and Protocol Classes

Test sequences are composed of a number of test messages. Each test sequences is stored in a file. The application must read a test sequence file to compose the test message objects. A protocol specific algorithm is required to read a message from the test sequence file and create a *TestMsg* object. The strategy design pattern is applied to encapsulate this algorithm in a *MsgReader* class. This allows the *TestMs*g class to become generic for all protocols.

The abstract class *Protocol* was defined to represent a generic protocol end point. Three methods *connect(ConnectAddress)*, *sendMsg(TestMsg)* and *recvMsg(TestMsg)* implement connection management, send and reception functionality in the *Protocol* class. Each of the three methods defined however must be adapted to use the API of the protocol implemented. This requires writing a separate protocol adapter class for each protocol implementation. The adapter pattern was used to address this framework design issue. The application of design patterns is discussed in more detail in sectio n 0.

TestMsg and *Protocol* correspond in part to information chunks and conduits as defined in [Hüni95] respectively. An information chunk is any type of protocol data that passes through a conduit. Likewise a *TestMsg* is passed by the test application to the *Protocol* class as a parameter of its *sendMsg()* method or is received by the *Protocol* class in the *recvMsg()* method. The Lotus SMTP application defines SMTPClient and SMTPServer classes that encapsulate protocol behavior, and a Message class that stores the data structure sent and received by the SMTPClient class. Both designs was considered during the design of the ipTF.

4.1.1. Test suite design

Test suite structure is and important design issue in the ipTF. IS 9646 defines a comprehensive abstract test suite structure for protocol conformance testing. This structure is described in chapter 3. The Lotus SMTP testing application defines a much simpler test suite hierarchy. It is composed of a *MessageList* that contains *Message(s).* The test suite hierarchy defined for the ipTF is based on the IS 9646 hierarchy and is illustrated in *Figure 15*.

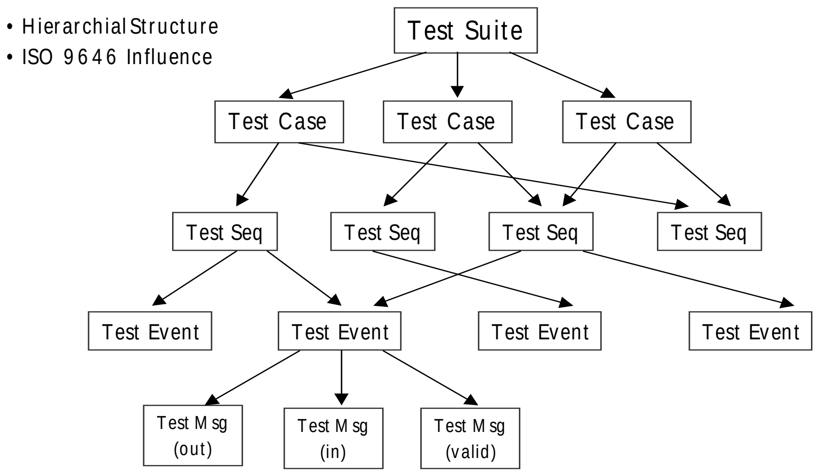


Figure 15: ipTF Test Suite Structure

TestEvent is the fundamental building block of the test suite. It contains three test messages: the message to be sent, the message received and a valid reply message for comparison. These are subsequently referred to as the *out message*, the *in message* and the *valid reply message* respectively. The protocol adapter extracts the out message from the *TestEvent* object and passes it to the IUT to send to the remote reference implementation. On receiving a reply from the remote reference the protocol adapter then stores the received message in the TestEvent object. A protocol specific comparison algorithm is used to compare the *in message* and the *valid reply message.*

The strategy pattern is used to encapsulate the comparison algorithm in a separate class. This allows *TestEvent* to become generic for all protocols. A test sequence is composed of a number of test events. Sequences are commonly used to transition protocol implementations into different states. A test case is a set of test sequences that tests a particular feature of the implementation. The test suite is composed of a number of test cases.

Test Method

In an earlier section the general testing model was discusses. A testing method was defined for the ipTF and is illustrated in Figure 16. The method used is similar to the remote testing method described in IS 9646. It describes remote methods that are used for testing client/server protocols operating in a distributed environment. Internet protocols generally follow a client/server model and operate in a distributed Internet environment. The distributed model was therefore considered suitable for testing of Internet protocols. *Figure 16* shows the distributed model as applied to the ipTF. It is based on the layered architecture of Internet protocols.

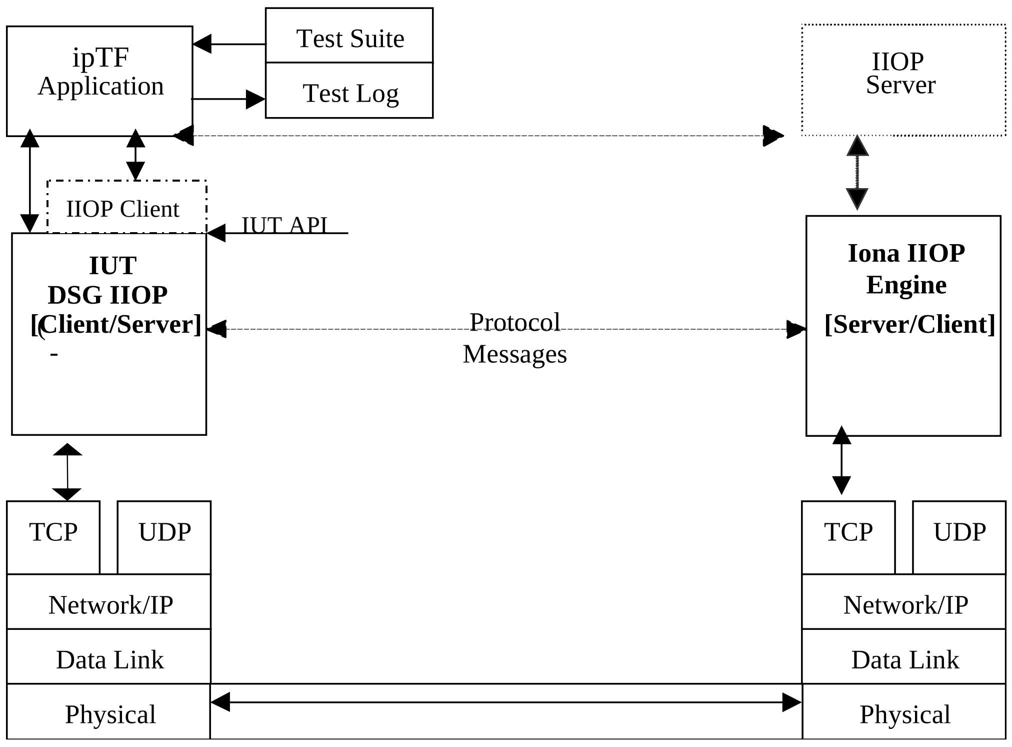


Figure 16: ipTF testing method

To test the DSG IIOP implementation an IIOP client and server must be implemented above the IIOP implementation layer. This is also illustrated in Figure 16. IIOP messages are sent and received between the client and server. These messages encapsulate object invocation and object location operations. The model also illustrates the service access point to the IUT API required by the framework. This API must be available to implement the general testing model as described in section 0.

Object Model & Classes

The main classes defined for the ipTF are listed in *Table 2*. Classes shown in *italics* are abstract. The relationship between these classes is shown in the ipTF object model and sequence diagrams. These can be viewed in ***Appendix A – Object Model*** & Sequence Diagram. The object model describes the main classes, methods and relationships between the classes. Pattern structures are also highlighted in this model. The sequence diagram illustrates the object interactions in the framework.

Protocol (EndPoint)	Abstract class that represents the protocol End-Point. Operations are defined for connection establishment and send and reception of messages.
ConnectAddress	Stores connection address details.
TestSuite	Stores test sequence, events and messages.
TestSequence	Selection of TestEvents.
TestEvent	Stores out, in & valid messages and comparitor object.
TestMsg	Abstract representation of a test message.
IPTF	Generic user interface part.
TestFactory	Creates protocol specific concrete classes (ie.TestSuite, MsgReader,MsgPanel etc.).
MsgPanel	Protocol specific user interface part
MsgReader	Reads message data from a file to create a test message.
MsgComparitor	Verifies received message against valid reply message.
Log	Generic class for storing a record of test messages sent/received and application state information.
ProtocolAdapter	Adapts the protocol specific interface to framework interface.

Table 2: ipTF Classes

The remaining features of the design are discussed in this section. These include connection addresses, logging, the user interface and the application of design patterns in the framework.

Connection Address: To establish a communication channel between a client and server each must be uniquely identified. This applies regardless of whether the protocol type is connection oriented or connectionless. Identification is achieved by allocating every client and server a unique connection addresses. In the Internet a connection addresses normally consists of the Internet host address and port number of the client or server. Additional address information may be required by some protocols. IIOP for example uses an IOR as a connection address. In the ipTF the abstract class ConnectionAddress can be extended to represent any type of Internet protocol connection address.

Log: Test applications generally create a dynamic record of tests completed and their results. In the ipTF model a record of each test event and its associated timings are stored in the generic Log class. In addition information about the testing application itself is recorded. For example the successful creation of each object by the TestFactory is recorded in the log. All records in the log are stored in a text format.

Application User Interface: The user-interface of the protocol testing application can be separated into two parts. The first part is generic for all protocols while the second part is protocol specific. The first part presents user interface elements for accessing common protocol test functions. These include connection establishment, sending messages, viewing the log and performing message comparisons. The second part allows for protocol specific user-interface features. Methods for implementing these features are defined in the *MsgPanel* abstract class.

Application of Design Patterns

Frameworks generally evolve through the factoring out of common functionality and by applying design patterns to common problems encountered in the framework. The original framework was implemented to test Telnet. Subsequently this was refined using design patterns and factoring out of common parts. The objective was to allow the basic framework structure to be extended through object composition rather than inheritance. Object composition leads to a greater amount of software re-use and is the preferred type of framework.

Three pattern types were considered applicable to the original design. These were abstract factory, strategy and adapter (wrapper). A full description of each pattern is given in the Design Patterns book by Gamma et al [Gam94].

TestFactory implements the abstract factory pattern. Its purpose is to centralise and control the creation of protocol specific objects required by a new application instance. It creates the protocol adapter, connection address, protocol specific user interface, test suite, message file reader and message comparitor objects.

MsgReader and *MsgComparitor* both implement the strategy pattern. *MsgReader* encapsulates the algorithm for creating a test message when reading it from a file. The algorithm would otherwise be contained in *TestMsg*. Doing so allows *TestMsg* to become more generic. Similarly the algorithm for comparing the *in message* to the *valid reply message* is encapsulated in the *MsgComparitor* class. Removal of the comparison algorithm from *TestEvent* allows it also to become a generic class.

A protocol specification does not define how the protocol should be implemented. Each implementation of a specific protocol can therefore have a unique API. The framework protocol class defines abstract methods for connecting to the remote server, sending a message and receiving a message. The API of the relevant protocol implementation should define interfaces for accessing this functionality. By applying the adapter pattern the framework (protocol class) method interfaces are adapted to the implementation interfaces using an adapter class.

Figure 17 illustrates the use of the protocol adapter pattern in the framework. The abstract class Protocol represents the protocol implementation (client) EndPoint. The methods connect(), sendMsg(), and recvMsg() are defined in this class. In a framework instance these methods are implemented by concrete protocol specific classes such as IIOPProtocol. At the protocol implementation level an API of methods is supplied for use by an application developer. This API typically has methods defined for connection management, and sending and receiving of messages. The signatures of the API defined methods differ from those defined in the framework. The framework methods must therefore be mapped to the appropriate protocol implementation methods.

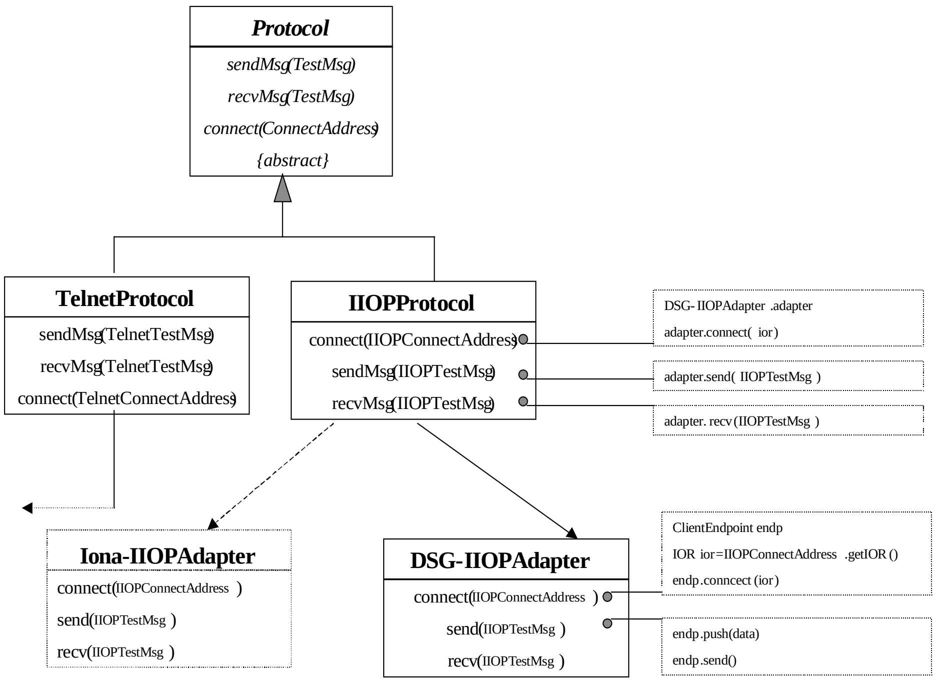


Figure 17: Protocol Adapter Pattern

A protocol adapter class was developed to map the interfaces between the framework and the protocol implementation. The DSG-IIOPAdapter class for example defines a connect() method for connection establishment. This connection method calls the appropriate connect method in the implementation API that achieves the connection functionality. The adapter class also adapts the framework methods sendMsg() and recvMsg() to the implementation API in a similar way.

Summary

This chapter discusses the main issues encountered in the framework design. The initial design was based on a Telnet implementation. Some design patterns were applied to make it more compositional in nature and an IIOP implementation added. However a new instance of the framework is still generated mainly through inheritance (white box framework). The factoring out of common functionality and the further application of design patterns is recommended for future versions of the framework. This recommendation is discussed in the evaluation chapter.

5. IMPLEMENTATION

The ipTF project plan and the schedule is described in the first section. The framework user interface design is illustrated and briefly reviewed in the second section. In the final section code segments, which implement object creation, logging, connection establishment, message sending/receiving, and message comparison functionality are explained in brief.

5.1. Project Plan and Schedule

Figure 18 shows the timeframe for completion of the main stages of the ipTF project. A six-month period was allocated for completion of the project. The overall project scope was determined on this basis. At the outset the main project phases were identified and milestone dates for the completion of each phase set. A brief description of the project work undertaken in each phase follows.

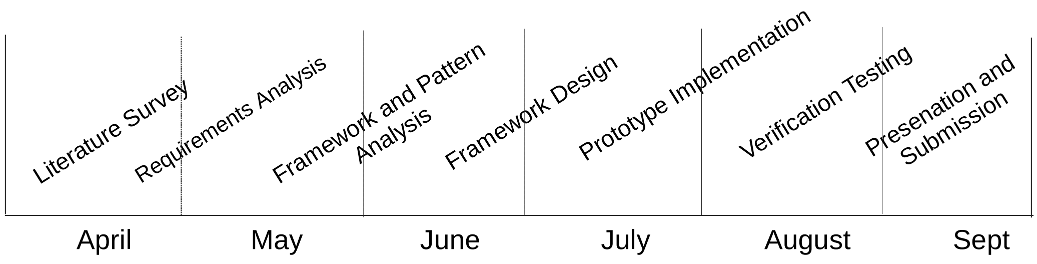


Figure 18: *ipTF project stages*

The project commenced with the literature survey phase. Background research on object-orientated design, UML modeling, frameworks, design patterns, OSI and Internet protocols, software testing concepts and protocol conformance testing was carried out. Following this an SMTP Protocol test application developed by Lotus International was reviewed as part of the requirements analysis phase. A number of meetings took place with the application developers in order to clarify the design goals and implementation specifics of the application. Use Case diagrams and an object model of the application design were made available for reference. An initial presentation of the ipTF dissertation goals and objectives was given at Lotus. This helped highlight potential project issues and clarified the possible scope of a feasible framework implementation.

Much of June was spent experimenting with design models for a framework application. The main components of a protocol testing architecture were identified and some initial design concepts were submitted for review. It was agreed that a mobile version of IIOP being developed by the DSG Group in Trinity was suitable for implementation in a framework prototype. The IIOP implementation was however developed in C++ while the framework itself was to be developed in Java. A DSG IIOP implementation would first require the resolution of Java to C++ integration issues.

A simple Java Beans Telnet protocol implementation was chosen as the basis for the initial framework implementation because of its simplicity and the fact that it was written in Java. The initial prototype was concerned mainly with implementing the basic framework structure and the creation of a suitable user interface. Rational Rose, a UML based tool, was used to create an initial design draft.

This initial design implemented the main framework classes *TestSuite*, *TestEvent*, *TestMsg*, *MsgPanel*, and *Protocol*. The application of design patterns to the framework was then undertaken. This resulted in the reorganisation of the existing framework classes and the addition of some new classes. A *TestFactory* class was created to implement the Abstract Factory Pattern. *Comparitor* and *MsgReader* classes were added to implement strategy patterns. These clasess encapsulated algorithms that were previously part of *TestEvent* and *TestMsg* respectively. The adapter/wrapper pattern was used to adapt the framework end-point, connection, and send/receive message interfaces to the protocol specific implementation API.

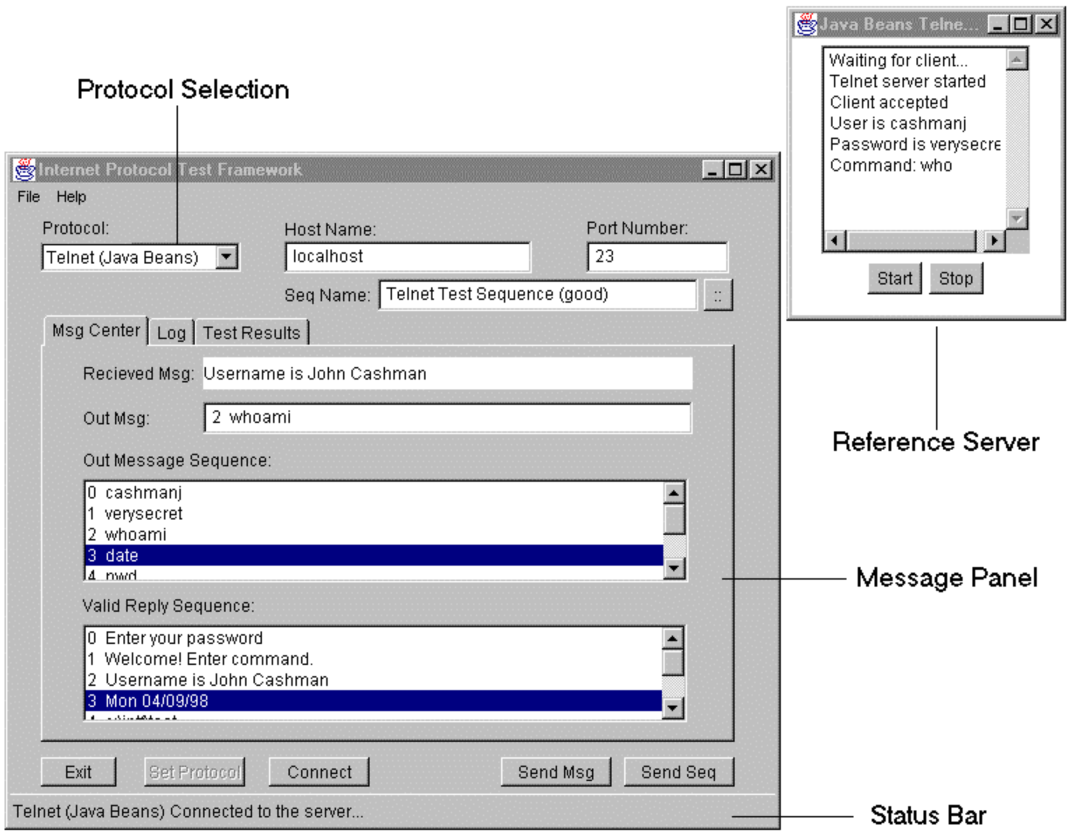
When Java to C++ integration issues were resolved using Java's native interface an ORB client and server were implemented in C++ for testing of IIOP. A second framework prototype was then completed to include both Telnet and IIOP protocol test implementations. The original user interface design was modified and improved upon. The completion of this second revision meet the original project goal of creating a framework which could be used as a basis for creating the two different protocol test applications. Finally a presentation of the framework was given and this dissertation document was written.

5.2. Languages and Tools

Framework creation is based on object-oriented design principles. The Java language was selected for this development because it is based on an object-oriented programming paradigm. The JBuilder[2] application development tool was chosen because of its full support for JDK 1.1 and the inclusion of an advanced user interface component library based on AWT[3]. This tool has an intuitive interface and is generally easy to use. Microsoft Developer Studio 5.0 was used for developing the C++ interface of the IIOP testing implementation. An example of the ipTF framework application developed in this environment is shown in ***Error! Reference source not found.***. The example shows the user-interface of the framework implementation for a simple Java Beans Telnet client and server implementation. Both the framework application and the reference server run of the same system in this example. The main user-interface elements and underlying framework code are described in the following sections.

[2] JBuilder is a rapid application development tool from Borland Inc. based on the Java programming language. It is fully compliant with the JDK 1.1 specification. More information is available at http://www.borland.com

[3] The Java Development Toolkit (JDK) provides a standard Abstract Widowing Toolkit (AWT) library.



Protocol: Telnet (Java Beans)
Host Name: localhost
Port Number: 23
Seq Name: Telnet Test Sequence (good)

Recieved Msg: Username is John Cashman
Out Msg: 2 whoami

Out Message Sequence:
```
0  cashmanj
1  verysecret
2  whoami
3  date
4  pwd
```

Valid Reply Sequence:
```
0  Enter your password
1  Welcome! Enter command.
2  Username is John Cashman
3  Mon 04/09/98
```

Telnet (Java Beans) Connected to the server...

```
Waiting for client...
Telnet server started
Client accepted
User is cashmanj
Password is verysecre
Command: who
```

Figure 19: ipTF framework application

5.3. User Interface Design

In this section the design of user interface elements in the ipTF is reviewed. Sample screen shots of the framework implementation are used to illustrate the main features of the user interface. The user interface consists of two main parts. The first part is generic for all protocols while the second part is protocol specific. *Figure 19* illustrates a working implementation of the framework using the Telnet protocol.

5.3.1. Msg Center view

The protocol specific part is implemented as a *MsgPanel* object. In the application this object is viewed in the Msg Center panel which is highlighted in *Figure 20*. The message panel can vary the representation of the user interface elements according to protocol requirements. In this implementation however the same panel format is used for both Telnet and IIOP. The panel represents a message sequence as a list box of messages. This message representation is textual. If additional interface options are required by a protocol implementation they are placed in the *MsgPanel* object. Regardless of the protocol implementation the *MsgPanel* object must provide functionality for determining the currently selected message. This information is required by the application when sending a message via *sendMsg()*. A method for returning the total number of messages in the sequence must also be implemented for similar reasons.

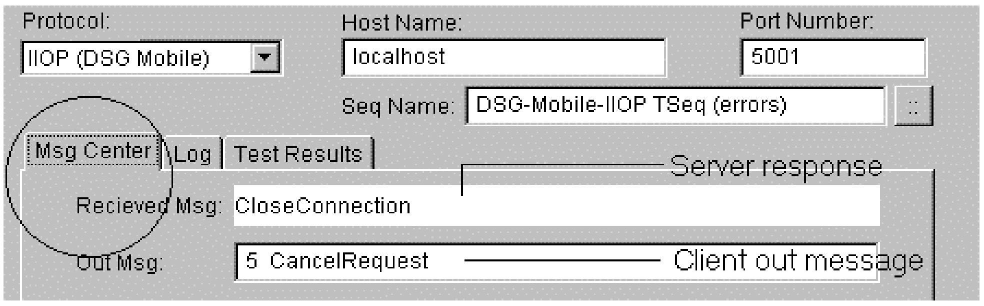
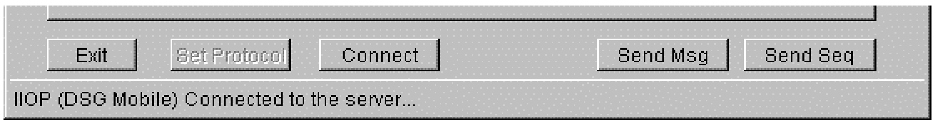


Figure 20: Message Center view

The generic parts include a drop down list for selecting the required protocol implementation, fields for entering the (usually remote) host name and port number of reference server and an option for selecting test message sequences. *Figure 21* shows the Exit, Connect, Send Msg and Send Seq buttons which provide access to matching functions for exiting the application, connecting to a (usually remote) server, sending a single message and sending a sequence of messages respectively.



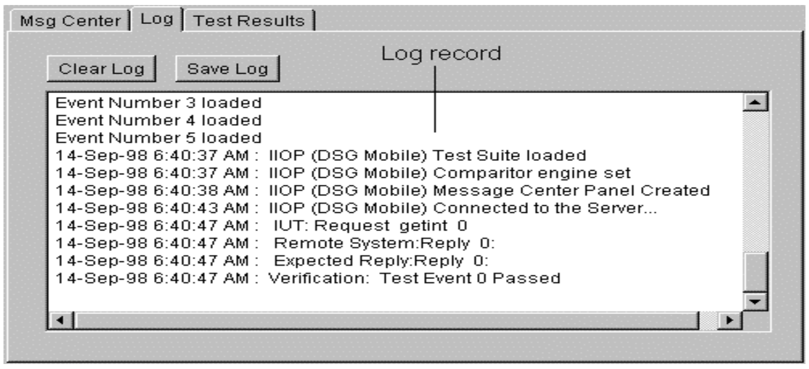
IIOP (DSG Mobile) Connected to the server...

Figure 21: Function options

A status bar at the foot of the main window gives information about the current state of application. The Log and Test Results view panels are also generic for all protocols. These are described in the next sections.

5.3.2. Log view

A textual record of every message sent and received by the application is written to the log. The date and time of each message is also recorded in the log. When a received message and a matching valid reply message are compared the result is also written to the log. Status information about the application itself is also recorded in the log. A sample of a Log record is shown in figure *Figure 22*. An option to save the log record to file is included.

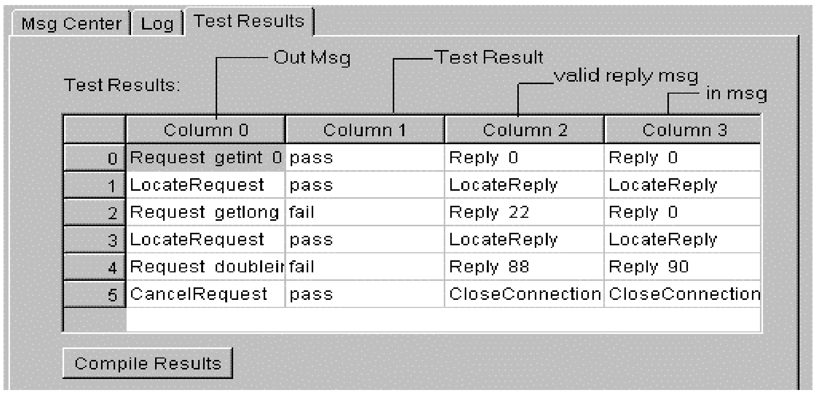


```
Event Number 3 loaded
Event Number 4 loaded
Event Number 5 loaded
14-Sep-98 6:40:37 AM :  IIOP (DSG Mobile) Test Suite loaded
14-Sep-98 6:40:37 AM :  IIOP (DSG Mobile) Comparitor engine set
14-Sep-98 6:40:38 AM :  IIOP (DSG Mobile) Message Center Panel Created
14-Sep-98 6:40:43 AM :  IIOP (DSG Mobile) Connected to the Server...
14-Sep-98 6:40:47 AM :  IUT: Request  getint  0
14-Sep-98 6:40:47 AM :  Remote System:Reply  0:
14-Sep-98 6:40:47 AM :  Expected Reply:Reply  0:
14-Sep-98 6:40:47 AM :  Verification:  Test Event 0 Passed
```

Figure 22: Log Record View

5.3.3. Test Results view

When a sequence of messages is sent and replies received the Test Results panel is used to compile and list the comparison results. *Figure 23* shows a sample result compiled for a DSG IIOP test sequence. Column 1 highlights individual test event results. Three test results are possible pass, fail, or no response. Columns 0, 2 and 3 show a textual representation of the out, valid reply and in messages respectively.

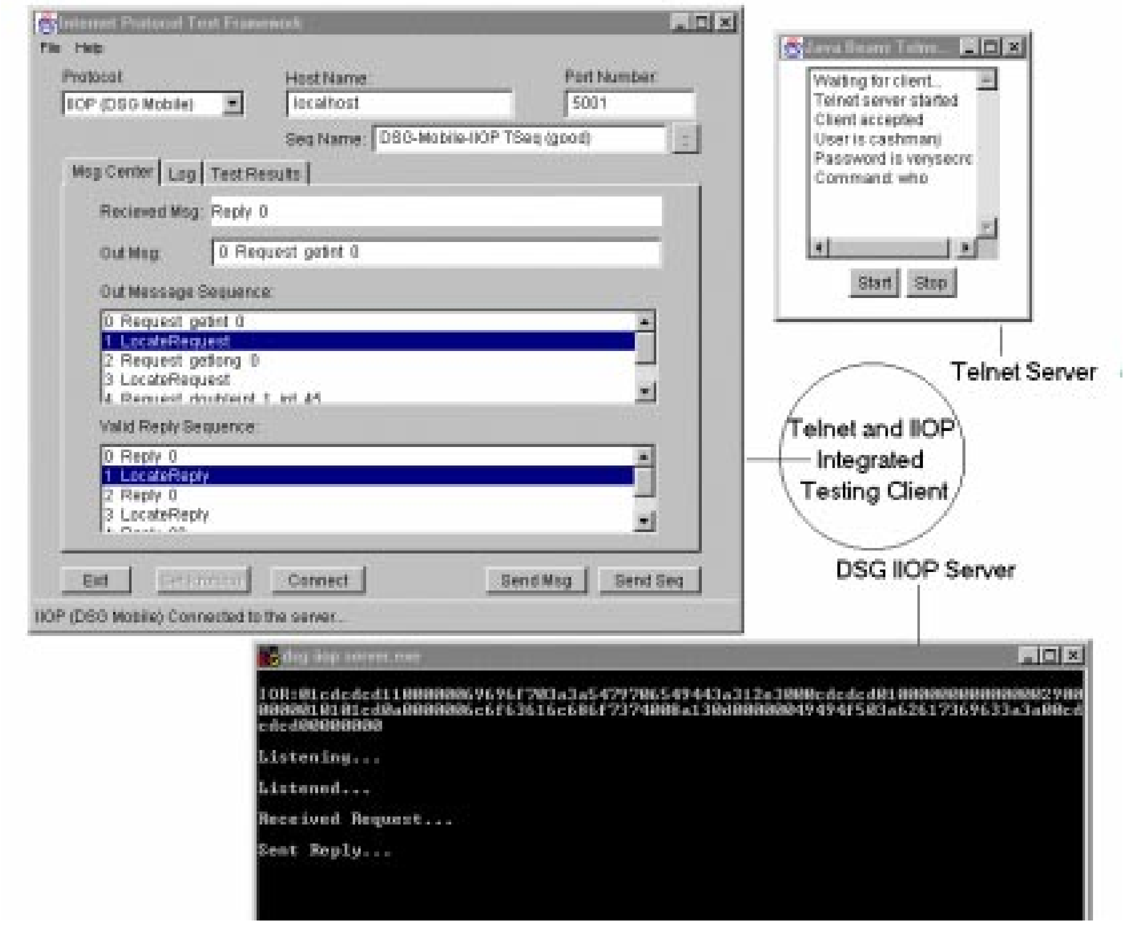


	Column 0	Column 1	Column 2	Column 3
0	Request getint 0	pass	Reply 0	Reply 0
1	LocateRequest	pass	LocateReply	LocateReply
2	Request getlong	fail	Reply 22	Reply 0
3	LocateRequest	pass	LocateReply	LocateReply
4	Request doubleir	fail	Reply 88	Reply 90
5	CancelRequest	pass	CloseConnection	CloseConnection

Figure 23: Comparison View

5.4. Multi-protocol implementation feature

The ipTF framework is capable of dynamically switching between testing of different protocols. An example of this multi-protocol testing feature is illustrated in *Figure 24*. In the example the framework dynamically connects to both Telnet and IIOP server implementations. Appropriate protocol test sequences are sent via the client to the respective Telnet and IIOP reference servers. This is possible because the protocol specific components are added as required while the basic framework remains the same. The only user interface element that changes with the protocol is the Msg Center panel (*MsgPanel* object).



Protocol: IIOP (DSG Mobile)
Host Name: localhost
Port Number: 5001
Seq Name: DSG-Mobile-IIOP TSeq (good)

Recieved Msg: Reply 0
Out Msg: 0 Request getint 0

Out Message Sequence:

```
0 Request getint 0
1 LocateRequest
2 Request getlong 0
3 LocateRequest
4 Request doubleint 1 int 45
```

Valid Reply Sequence:

```
0 Reply 0
1 LocateReply
2 Reply 0
3 LocateReply
```

IIOP (DSG Mobile) Connected to the server...

```
Waiting for client...
Telnet server started
Client accepted
User is cashmanj
Password is verysecrc
Command: who
```

```
IOR:01cdcdcd1100000069696f703a3a5479706549443a312e3000cdcdcd0100000000000000002900
0000010101cd0a0000006c6f63616c686f7374000a130d0000000049494f503a6261736963 3a3a00cd
cdcd00000000

Listening...

Listened...

Received Request...

Sent Reply...
```

Figure 24: Multi-protocol testing

5.5. Implementation Coding

The most important implementation features are reviewed in this section. These features include the creation of the application objects, connection establishment, sending and receiving of messages, logging and message comparison. Code segments that implement these features are highlighted in lines numbered 1 to 122.

5.5.1. Object Creation

IPTF.java is the main framework class. It controls the overall flow of control within the application. A user first selects a protocol from the protocol list (*Figure 25*).

Figure 25: Protocol Dropdown List

Based on this selection the protocol TestFactory object is created. For example if IIOP (DSG Mobile) is chosen from the list an IIOPTestFactory object is created in IPTF.java (line 3). All such events are recorded in the Log by the addLogEntry() method (line 4).

IPTF.java

```
1.  if (protoname.equals("IIOP (DSG Mobile)")){
2.     //Create IIOP TestFactory....
3.     tfactory=new IIOPTestFactory(this);
4.     addLogEntry(protoname + " Test Factory Creating Framework..");
5.  }

6.  connectadr=tfactory.setConnectAddress(); //Create connect address
7.  proto=tfactory.createProtocol();         //Create Client Endpoint
8.  tsuite=tfactory.loadTestSuite();      //Create the Test Suite
9.  comparitor=tfactory.getComparitor(); //Create file reader object
10. msgpanel=tfactory.createMsgPanel();  //Create protocol specific ui
```

After a new factory object is created its methods are called to create the protocol specific objects required by the application. Each object creation method is defined in a concrete sublclass of the abstract *TestFactory* class. The test factory passes back objects of type *ConnectAddress*, *Protocol*, *TestSuite*, *Comparitor* and *MsgPanel* to the calling method in IPTF.java. This implementation is illustrated in lines 6..10.

TestFactory.java defines abstract methods for creating each object. These methods are shown in lines 11..19. A concrete method is defined in the protocol specific subclass for each abstract method defined in *TestFactory*. Two concrete implementation examples are taken from IIOPTestFactory.java (lines 11..19). The *createProtocol()* method is responsible for creating an IIOP endpoint object and returning it to the calling method in IPTF.java (lines 20..23). The *getComparitor()* method creates an IIOP Comparitor object and likewise returns it to the calling method in IPTF.java (lines 24..27). All other objects are created in a similar way by the test factory.

TestFactory.java

```
11. public abstract class TestFactory {
12.    public TestFactory() {
13.    }
14.    public abstract ConnectAddress setConnectAddress();
15.    public abstract Protocol createProtocol();
16.    public abstract TestSuite loadTestSuite();
17.    public abstract MsgControlPanel createMsgPanel();
18.    public abstract Comparitor getComparitor();
19. }
```

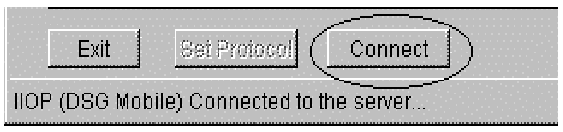
IIOPTestFactory.java

```
20. public Protocol createProtocol(){
21.    proto=new IIOPProtocol();
22.    return proto;
23. }

24. public Comparitor getComparitor(){
25.    IIOPComparitor tcomp=new IIOPComparitor();
26.    return (tcomp);
27. }
```

5.5.2. Connection establishment

Before messages can be sent between the client and server a protocol connection must be established. Connection establishment is implemented at the highest level in IPTF.java. Lines 28..29 illustrate the protocol and connection address objects being created and returned by the *TestFactory* object. The method *proto.connect(connectadr)* is then called by IPTF.java to establish the endpoint connection. (Line 30)



IIOP (DSG Mobile) Connected to the server...

IPTF.java

```
28. connectadr=tfactory.setConnectAddress();//Create connect address
29. proto=tfactory.createProtocol();  //Create Client Endpoint
30. String conmsg=proto.connect(connectadr);//connect
```

An abstract connect() method is defined in Protocol.java (line 31) and implemented by the concrete class IIOPProtocol (lines 32..41). This method calls the DSG Mobile IIOP API connect method that establishes a connection at the lowest level.

Protocol.java

```
31.  public abstract String connect(ConnectAddress connectadr);
```

IIOPProtocol.java

```
32.  client=new DSGIIOP();
33.
34.  public String connect(ConnectAddress connectadr){
35.     connectadr=(IIOPConnectAddress) connectadr;
36.     if(client.connect(aconnectadr.getIOR())==0){ //native C++
37.        return ("Connected to IIOP Server");
38.     }
39.     else
40.        return ("Failed Connection ...");
41.  }
```

Java is used to implement the framework and C++ to implement the DSG IIOP. An extra layer of adaption is required to convert between the two language interfaces. This type of adaption is made possible through Java's Native Interface (JNI) [Java98].

The Java native C++ connect function call is implemented in dsgiface.cpp. A Java IOR string is first converted to a C++ character array (line 45) and then to a stringified IOR (line 48). The client endpoint connects (line 48) to the remote server using the stringified IOR. If successful a connection is established to the remote server endpoint an ok status is returned and passed back to the calling Java function (line 54). A connection is now established between the client and server.

DSGIIOP.java

```
42. public native int connect(String s);  //Java native C++ method
```

dsgiface.cpp

```
43. long IPTF_MRIIOP_connect(struct hIPTF_MRIIOP *jc_this,struct
                                    Hjava_lang_String *javaior){
44.    char buffer[MAXJSLEN];
45.    javaString2CString(javaior, buffer, sizeof(buffer));
46.    IOR aior(buffer);
47.
48.    if(endp.Connect(aior) != OK){
49.       long status=-1;
50.       return status;
51.    }
52.    else{
53.       status=0;
54.       return status;
55.    }
56. }
```

5.5.3. Sending and Receiving Messages

After the client/server endpoint connection is established protocol messages can be sent between the client and server. The message to be sent to the remote server is selected by the user from the message list (*Figure 26*) or by the system itself when sending a sequence of messages.

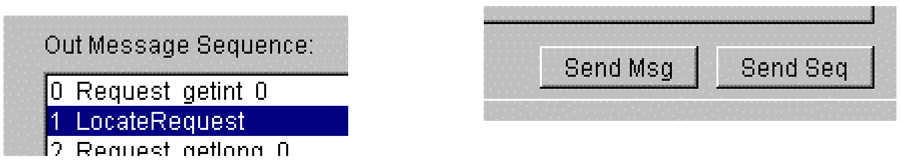


Figure 26: Message list, send message and send sequence user interface elements

The currently selected message is extracted from the *TestSuite* object and passed as a parameter to the protocol *sendMsg()* method (lines 57..67).

IPTF.java

```
57.   //get index of currently selected message.
58.   outmsgindex=msgpanel.getCurrentTEvent();

59.   //Extract Test event from Test Suite
60.   atestevent=tsuite.getTestEvent(outmsgindex);
61.   sendEvent(atestevent);

62.   public void sendEvent(TestEvent tevent){
63.      TestMsg inmsg;
64.      //send out TestMsg receive in TestMsg
65.      inmsg=proto.sendMsg(tevent.getOutMsg());
66.      tevent.setInMsg(inmsg);  //Store received TestMsg
67.   }
```

A concrete implementation of the *sendMsg()* method is realised in the protocol specific subclass. This method extracts the message elements from the *TestMsg* object and converts them into appropriate message parameters required by the protocol specific API (lines 71..79). The class DSGIIOP.java defines a send/receive native method (line 86). This method is implemented in dsgiface.cpp (line 87). The reply message is converted into a *TestMsg* object and passed back to the IPTF.java calling method. This message is then stored in the *TestEvent* object as the in message. This completes the send/receive implementation.

IIOPProtocol.java

```
67.   public TestMsg sendMsg(TestMsg msg){
68.       Vector v; int element=0, count=0;
69.       TestMsg inmsg=new IIOPTestMsg();
70.       String instr[]=New String[];

71.       v=msg.getData();        // get abstract message data
72.       int size=v.size();      // number of msg elements
73.       String signature[]=new String[size];  //API message type

74.       //convert message data to required format for Client API
75.       for  (int i=0; i < v.size();i++){
76.         Object obj=v.elementAt(i);
77.         signature[i]=(String)obj;
78.         count=i;
79.       }

80.       instr=client.sendMsg(signature, count);  //send message

81.       for  (int i=0; i < instr.length();i++){
82.         inmsg.addDataItem(instr[I]); //convert reply message
83.       }
84.       return inmsg; // return reply  TestMsg
85.   }
```

DSGIIOP.java

```
86.   public native String[] sendMsg(String s[], int i);
```

dsgiiop.cpp

```
87.   HArrayOfString *iptf_MRIIOP_sendMsg(struct Hiptf_MRIIOP
                           *jc_this,HArrayOfString *jopname, long msgtype ){..}
```

5.5.4. Message file reader

At the lowest level a test suite is composed of protocol messages. The test suite is normally stored in a persistent file format. Protocol message objects (*TestMsg's*) are written to and read from the file as required by the framework application. How they are read and written is dependent on the protocol message structure. A protocol specific file reading/writing algorithm is therefore required by the application. This algorithm is contained in the *MsgReader* class (line 88). The algorithm reads the message data from file and builds a *TestMsg* object (lines 89..98). The object is returned to the calling application method (line 99). Encapsulating the algorithm in the *MsgReader* object allows the *TestMsg* to become generic.

MsgReader.java

```
88. public abstract TestMsg readMsg();
```

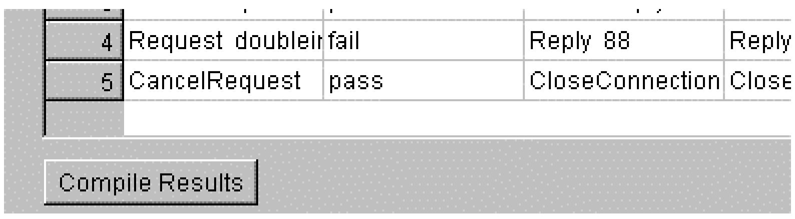
IIOPMsgReader.java

```
89.     BufferedReader br;
90.    public TestMsg readMsg(){
91.       String s1;
92.       msg=new IIOPTestMsg();

93.       try{
94.          s1=br.readLine();
95.          msg.addDataItem(s1);
96.          …………
97.          …………
98.       }
99.       return(msg);
100.   }
```

5.5.5. Message comparison

To verify the correct behaviour of the protocol implementation received messages are compared to matching valid reply messages. This message comparison functionality is protocol specific because the structure of each protocol message type differs. The comparison functionality is defined in the *Comapritor* class (lines 102..108) and implemented in a protocol specific subclass. An *IIOPComparitor* code segment is illustrated in lines 111..124. The encapsulation of the message comparison algorithm in this way allows the *TestEvent* object to become generic for all protocols.



4	Request doublei	fail	Reply 88	Reply
5	CancelRequest	pass	CloseConnection	Close

Figure 27: Message comparison compilation

The *TestEvent* object is passed to the message comparison method (line 101). The *Comparitor* object extracts the received and valid messages from the *TestEvent* and performs the comparison (lines 102..108). A protocol specific example for IIOP is shown in lines 109..122. The status of the comparison is returned to the calling application method. All message comparison results for a message sequence are compiled and illustrated as shown in *Figure 27*.

IPTF.java

```
101.   int verify=atestevent.verify(comparitor);
```

Comparitor.java

```
102.   public abstract class Comparitor {
103.      TestMsg inmsg;
104.      TestMsg validmsg;

105.      public Comparitor() {
106.      }

107.      public abstract int compare(TestEvent tevent);
108.   }
```

IIOPComparitor.java

```
109.   public class IIOPComparitor extends Comparitor {

110.      public IIOPComparitor() {
111.      }

112.      public int compare(TestEvent tevent){
113.         Vector inv=new Vector();
114.         Vector validv=new Vector();
115.         int status =-1;
116.         inmsg=tevent.getInMsg();
117.         validmsg=tevent.getValidReplyMsg();
118.         …………
119.         …………
120.         return(status);
121.      }
122.   }
```

5.6. Summary

A description of the ipTF implementation was given in this chapter. To give context to the application development a brief review of the main development stages in the project is given. This is followed by a description and illustration of the framework application’s user-interface. In the final part the coding of the main framework features is analysed. A selection of sample code segments taken from the application is used to support this analysis.

6. EVALUATION

The expanding number of new and revised Internet protocols has given rise to the need for testing multiple protocols. Currently most protocols are tested using protocol specific applications. This approach becomes very costly as the number of new protocol implementations increases. Full analysis, design, implementation and testing phases must be undertaken for each new application developed. This requires a significant resource commitment and takes considerable time to develop a new application from scratch. Exploring how this testing process could be completed in a more efficient and cost effective manner is the subject of this dissertation.

If a generic application could be developed to test a range of protocols this would save time and resources spent on application design and development. Because most Internet protocols have different syntax and semantics it is difficult to design a generic test application. The goal of this dissertation is to determine the feasibility of designing and implementing a generic application that tests multiple Internet protocols.

A generic application should implement features that are common to all protocol testing applications and at the same time allow for the implementation of protocol specific testing features. Frameworks had been successfully applied to resolve this general design problem of developing a set of related applications. Because frameworks focus on modeling of the common features of applications and have a high reuse potential this approach was adopted for the design and implementation of the generic Internet protocol testing application. Application analysis and design information is captured in the basic framework design. Much time and resources are saved through the reuse of analysis, design and code when developing many applications of a similar type. If a protocol testing framework can be successfully designed these benefits should follow.

Reusability is a fundamental criterion of a good framework. The framework should therefore be capable of generating a valid test application for any new protocol implementation. Beyond this the benefits of using the framework must outweigh the costs. Are new protocol test applications easy to design and implement? If a developer must take a long period to learn how to create a new application from the framework, its value is diminished. Essentially it must be easier and quicker to develop a new application using the framework than developing a similar application without using the framework. Other criteria include the ease of use and performance of the new application generated from the framework. These criteria must satisfy user requirements.

This dissertation presents a simple framework (ipTF) for the construction of Internet protocol testing applications. Abstractions are defined which model the basic structure and behaviour of protocol testing applications. The feasibility of a common method for testing multiple Internet protocols is first evaluated in this chapter. Having established the feasibility of this goal the framework itself is evaluated against the criteria of reusability, ease of use and performance. This type of analysis depends to some degree on the comparison of application metrics. The compilation of application metrics is however beyond the scope of the dissertation. Given this constraint possible framework improvements are suggested in the final part of this evaluation.

6.1. Design goals

The benefit of frameworks is that they can enable a higher level of code and design reuse than other design approaches if successfully implemented. Class libraries for example offer reuse through individual components. Reuse in frameworks is more than simply the reuse of components. Domain expertise is captured in the basic framework. A framework consists of a set classes that embodies the abstract design and implementation of the application. In this way the ipTF framework is a set of abstract and concrete classes that embodies the basic design of a protocol testing application. The reuse potential of the framework is based on the common structure and behaviour of Internet protocol testing applications.

Framework design proved to be a useful design technique for modeling this particular application. When designing a generic solution for a set of applications a top down approach to analysis is taken. This contrasts to the bottom up approach of finding objects that model a specific application. In the ipTF applying the top down approach involved analysing both the nature of protocols and the general structure and behaviour of protocol testing applications. The analysis revealed that protocol testing applications had common features such as connection establishment, sending and receiving of messages, message comparison and logging. This functionality is not however implemented in a uniform way across protocols. Furthermore the structure of protocol messages is different for most all protocols.

Most frameworks start out as white box frameworks. In a similar way a white-box approach was adopted in the initial ipTF framework design. The basic design is encapsulated in the relationships between the objects and defined in a set of abstract class interface definitions. Protocol specific extensions were implemented by inheriting from these abstract class interfaces. The initial framework design was verified through the creation of a test application for a simple Telnet protocol implementation. A framework is generally not stable after the first application has been developed from it. An analysis of the Telnet testing implementation identified the need to redesign the framework. The framework design was based entirely on inheritance with the common functionality defined in abstract base classes.

Two approaches were used to evolve and update the framework design. Firstly, common protocol testing functionality was factored out of application specific concrete classes and placed in abstract classes. Secondly, protocol specific functionality was encapsulated in classes that are composed rather that inherited from. This was achieved mainly through the application of design patterns. In the second iteration some factoring out was achieved and some design patterns were applied.

Design patterns were used to identify some common framework design problems. Patterns for object creation (abstract factory), the encapsulation of algorithms (strategy) and adapting the framework interface (adapter/wrapper) to the protocol specific API were applied. Some functionality was moved out of TestEvent and TestMsg classes into algorithm specific abstract classes to make them more generic. The protocol specific functionality of message comparison and message reading was encapsulated in classes that are composed. The adapter pattern adpated the protocol specific interface to the framework defined protocol interface. Common protocol functionality for establishing connections and sending/receiving messages was factored out and defined in a separate protocol class. The protocol adaptee class is protocol specific and is implemented through inheritance. The adapter class could have been made generic and the adaptee class compositional. However in this iteration of the ipTF the adapter class is defined as an abstract class and the adaptee class is extended through inheritance.

The framework approach allowed the design to evolve as the features and issues became more fully understood. The changes made between first and second iterations of the design support this. Only the Telnet protocol was implemented in the first design iteration. In this design test messages were read from file using a Telnet specific algorithm. This algorithm was included in the TestMsg class. In the second design iteration it became apparent that if IIOP were also implemented the TestMsg class would need to contain two protocol specific algorithms. The protocol specific algorithm was therefore encapsulated in a separate class allowing TestMsg to become generic. The second design iteration, which included both Telnet and IIOP implementations, proved to be more stable and flexible than the first iteration. The separation of the protocol specific parts from generic parts helped achieve this.

IIOP and Telnet testing applications were successfully generated from the second framework iteration. The implementation of this generic testing application for two very different protocols supports the use of a framework design approach. Could the ipTF be used to generate similar test applications for other Internet protocols? The common structure and behaviour of the protocols provides the basis for generic design within the framework. This suggests that it would be possible to apply the framework to protocols that have a similar design structure and behaviour to either IIOP or Telnet. Protocols that have additional features would require extensions to the framework design or may even require a redesign of the basic framework. For example both Telnet and IIOP protocols are stateless. Protocols such as SMTP, however, have multiple states. A testing application for SMTP would need to address the issue of storing state information. This type of functionality is not addressed in the current framework design. Other protocols may have particular design considerations. Despite this the basic framework design should be applicable in large part to most protocols. Only through reuse in testing many protocols can the basic design be verified as stable.

6.2. ipTF design

In this section the ipTF design is evaluated. The current iteration of the framework is based mainly on extension through inheritance. While there is considerable design reuse in the framework the level of code reuse is perhaps relatively low. How could this be improved? Default/sample classes exist for Telnet and IIOP implementations, which provide a useful reference for implementing protocol specific classes. It does however require that the developer becomes aware of the ipTF framework structure and design. If the design were compositional the developer would not need to be aware of the framework structure. They would be free to implement new functionality without effecting other classes. When composition is used a generic class holds only a reference to the composed object and is therefore not effected by changes in the composed object. As the basic design becomes stable a compositional approach is recommended for future iterations.

In many of the current ipTF abstract classes there is functionality which could be placed in lower level abstract classes thus allowing the original classes to become generic. An example of this is the MsgPanel class. This class represents the protocol specific user interface. It was made abstract because it was felt that different user interface elements might be required by different protocols. However the user-interface implemented for both Telnet and IIOP is identical. It is possible that additional user-interface elements may be required for other protocol implementations when the framework is reused. How to factor out common user-interface parts while allowing for the addition of protocol specific user-interface elements is a design issue. One approach to resolving this could be the application of the decorator design pattern. Common functionality is placed in a single generic class and decorated through a compositional structure that holds the protocol specific user-elements.

Another potential example of making classes generic is found in the test suite structure. A test suite is composed of a hierarchy of test sequences, test events and test messages. In the current revision only the TestEvent class is generic. Analysis of both the Telnet and IIOP implementations indicate that this structure could be modeled in a generic way by factoring out of the common functionality and placing it in generic TestMsg and TestSuite classes. This should be possible because the TestMsg and TestSuit classes are extended in an identical way in the current revision.

6.3. ipTF benefits & costs

The steps necessary to create an imaginary SMTP protocol testing application from scratch are compared to those of using the ipTF framework to create a similar application. The comparison is used to illustrate some of the costs and benefits of adopting the framework approach in this project.

Protocol specific application An examination of the SMTP protocol specification at a high level would be necessary to determine the protocol structure and behaviour. In addition lower level analysis of features such as message formats and procedure rules for communication would also be necessary. The general features of a protocol testing application would need to be established. The accuracy of this analysis would depend to a great extent on the individual developer's understanding of the domain. Furthermore if no existing protocol testing applications were available to the developer accurate analysis would be difficult. Based on the analysis the developer would model a new design. This design would be the basis for implementing the protocol specific testing application. It is likely that a number of design and implementation iterations would be undertaken before the application would meet user requirements.

Framework generated application: The analysis of protocol testing applications and the general features of protocols is captured to some degree in the basic ipTF framework design. The framework is still at early stage of development and would not capture all the design features required for some protocols. An analysis of SMTP's syntax and semantics would be required to enable implementation of the protocol specific parts of the framework. This analysis may reveal framework limitations. For example SMTP has a number of defined states and specifies valid input message sequences for transitioning between states. Testing of this type of protocol is often based on the FSM concept. The current framework deals only with stateless protocols and consequently there is no concept of keeping state information in the framework design.

The application code implemented in generic classes would be relevant for creating an SMTP application design. In addition the sample IIOP and Telnet implementations would act as a guideline for implementing protocol specific code. The developer will need to spend some time learning and understanding the framework. This should be a relatively short period as the basic design is documented in an object model and sequence diagrams. The Java implementation of the framework is based directly on this simple design model.

In summary, it is difficult to make a comparison between using and not using the framework for application development as comparable application metrics are not available. A full evaluation of the ipTF against protocol specific test applications is beyond the scope of this dissertation. However observations can be made that provide some guidance in respect of this type of evaluation.

The current framework design is in its second iteration. A Telnet protocol testing application was implemented using the first iteration. An analysis of this design led to a basic redesign of the framework. The Telnet application was then reimplemented using this second design iteration. When this was achieved to a satisfactory level the development of an IIOP testing application was undertaken. The ipTF framework provided support for developing this IIOP implementation in three ways. Firstly, it provided a structure for application analysis through the framework design. Secondly, the design of the framework was used as the basis for creating the IIOP application design. Thirdly, many of the concrete classes implemented for Telnet were used as a reference guide for the IIOP implementation. If the IIOP application were developed without using the framework these benefits would not have been available.

There were however costs associated with the framework. Firstly, considerable time was spent creating the initial framework. This included work on analysis, design, implementation and test phases of the framework development. The initial design was then remodeled to include factoring out of common functionality and application of design patterns. A full protocol testing application would require the definition of a complete set of protocol test sequences. The generated application's robustness, efficiency and performance would also need to be fully tested in a rigorous way.

The current ipTF framework is by no means complete and requires redesign to improve its reuse potential. From the IIOP and Telnet implementation it is clear that a third design iteration would add significantly to it's reuse potential. In a third design iteration, for example, design patterns could be applied to make the framework more compositional. Common functionality could be factored out of the concrete classes into abstract classes. With each additional reuse of the framework this process of redesign should become less significant until eventually the basic design becomes stable. As the framework design overhead falls the basic design should improve. The benefit to the developer will therefore become greater with each framework reuse and subsequent redesign.

6.3.1. Protocol Revisions

Protocol specifications are often revised to include new features. Existing protocol implementations are updated to include these new features. Updated implementation must be tested to ensure conformance to the revised specification. The framework approach offers the flexibility to adapt existing protocol testing applications for protocol revisions. In most instances new revisions of a protocol will have much in common with the structure, behaviour, syntax and semantics of the older revision. Often the new protocol features require new message types and procedure rules. A compositional framework would allow new protocol components to be added without effecting the existing framework design or implementation. The new features and behaviour could be encapsulated in components that are plugged into the existing framework application. For example new protocol adaptee, message reading and message comparison components could replace existing components in the framework. Plugging in these components to the framework may be sufficient to entirely update the testing application for a protocol revision. The flexibility of the framework approach would allow for quick, efficient and accurate updating of the testing application for protocol revisions.

6.3.2. Multiple Protocol Testing

The ipTF provides a user-interface for switching between multiple protocol testing implementations. Instances of protocol testing applications are dynamically created at runtime by the application as required. It is for example, possible to run a sequence of Telnet tests followed by a sequence IIOP tests within the same ipTF generated application. The verification of all tests is recorded in a single log. This feature could be used for testing of communication of servers that implement multiple protocols. Protocol specific test application does not result in a consistent approach to testing protocols and cannot therefore offer this type of feature.

6.4. Conclusion

The ipTF framework achieves the basic design goal of modeling a generic protocol testing application. The current framework design and implementation was realised in a second iteration of the ipTF. Though the original goal was achieved there is significant scope to make the basic framework more generic and compositional in nature. This would afford an opportunity for greater reuse of the design and code. Using composition to extend the framework would make it easier for the developer to add protocol specific functionality without effecting the basic framework design. Design patterns could be applied to encapsulate the protocol specific functionality in classes that are composed rather than inherited from. Although application metrics were not available for comparison against other applications it is suggested from observation that the benefits of the framework would eventually outweigh the costs. If the framework is analysed and redesigned (if necessary) after reuse the greater will be the potential for further framework reuse.

6.5. Summary

The ipTF design and implementation were evaluated in this chapter. The need for testing of multiple Internet protocol testing was highlighted. It was suggested that a generic test application could be used to address this problem. This method would save resources and time over current protocol specific testing methods. Object-oriented frameworks are a design methodology for developing a set of related applications. The framework approach was therefore considered a suitable design methodology for creating such a generic application. An assessment of the framework approach was evaluated in terms of the feasibility of achieving a generic type protocol testing application. The ipTF framework realisation gives proof of this feasibility. The relative success of the framework was then evaluated in terms of its ability to represent the commonality of many protocol testing applications. The current framework is at an early stage of development. To date only IIOP and Telnet prototype testing implementations have been generated from the framework. An SMTP example illustrated potential limitations of the current framework design. Other protocol testing implementations may identify additional framework design issues.

Finally some suggestions are made for the future development of this framework. New prototypes should be implemented for additional protocols. Ideally each protocol being implemented should be representative of a particular protocol type. This would help identify and resolve basic protocol testing functionality issues for a subset of Internet protocols. Design patterns should be applied to the framework design as it evolves to make it more compositional in nature. This would make it simpler for the developer to reuse the framework. Eventually, when the design becomes stable, a full testing application should be implemented for a protocol. A set of application metrics could be derived for this framework application and used to fully evaluate the framework against a comparable protocol specific testing application.

7. REFERENCES

[Alex79] Alexander Christropher The Timeless Way of Building, Oxford Univ. Press, New York, 1979

[App98] Appleton Brad, Patterns and Software: Essential Concepts and Terminology http://www.enteract.com/~bradapp/docs/patterns-intro.hml, 1998

[Boch94] Bochmann Gregor V., Petrenko Alexandre, *Protocol Testing: review of methods and relevance for software testing*, ISSTA 94, Proceedings of the 1994 International symposium of software testing and analysis, p109-124

[Boe79] Boehm, B.W., (1979) Software engineering: R & D trends and defense needs. In *Research Directions in Software Technology* (Wegner, P., ed.) Cambridge MA: MIT Press

[Boll98] Bolliger Jürg, Gross Thomas, A Framework-Based Approach to the Development of Network Aware Applications, IEEE Transactions on Software Engineering, V24 No 6, May 98.

[Boo88] Booch Grady, *Best of Booch*, SIGS Reference Library, 1996

[Boo94] *Designing an Application Framework*, Dr. Dobbs Journal 19, No.2, 1994

[Copl97] Coplien James O., *Idioms and Patterns as Architectural Literature*, IEEE Sofware, January 97

[Corba98] The CORBA architecture and specification document contains the Internet Inter Orb Protocol (IIOP) specification (Chapter 13.7). The Object Management Group (OMG) is responsible for maintaining this document. CORBA version 2.2, Feb 1998 is the current revision. http://www.omg.org

[Comer95] Comer Douglas E., *Internetworking with TCP/IP*, 3rd Edition, Prentice Hall International Editions, Chapter 23 Applications: Remote Login (Telnet, Rlogin), 1995

[Domino] Lotus Notes (client) and Domino (server) is a popular distributed groupware product developed by Lotus Development Ltd. http://www.lotus.com

[Gam94] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994

[Holz91] Holzmann Gerard J., *Design and Validation of Computer Protocols*, Prentice Hall Software Series,1991

[Hüni95] Hüni Hermann, Johnson Ralph, Engel Robert. *A framework for Network Protocol Software*, OOPSLA'95 Proceedings, Austin, 1995, ftp://st.cs.uiuc.edu/pub/patterns/papers/conduits+.ps

[IS 9646] International Standard (IS) 9646 is a standard devoted to the subject of conformance testing implementations of Open Systems Interconnection (OSI) standards.

[Java98] The Java Development Toolkit (JDK 1.1) supports the Java Native Interface (JNI) for C/C++, http://www.javasoft.com

[Joh88] Johnson Ralph E, Brian Foote. *Designing Reusable Classes*, Journal of Object Oriented Programming, June/July 88

[Kert97] Norman L., Ward Cunningham, *Using Patterns to improve our Architectural Vision*, IEEE Computer, January, 1997

[Knig93] Knightson Keith G, OSI Protocol Conformance Testing, McGraw-Hill, 1993 p86.

[Kore90] Korel Bogdan, *Automated Software Test Data Generation*, IEEE Transactions on Software Engineering, August 1990

[Kore96] Korel Bogdan, Ferguson Roger, *The chaining approach for software test data generation*, ACM Transactions on Software Engineering and Methodology, V5 No. 1, Jan 96.

[Lan95] Landin N., Niklasson A. *Development of Object Oriented Frameworks* Masters Thesis, Dept. of Communications Systems, Lund University, 1995, http://www.tts.lth.se/Personal/bjornr/Papers/OOFW.ps

[Mel97] Mellor Stephen F, Johnson Ralph. *Why Explore Object Methods, Patterns and Architectures?* IEEE Software, Jan 97

[Mile94] Miller Raymond E, Snajoy Paul, *Structural analysis of Protocol Specifications and Generation of Maximal Fault Coverage Conformance Test Sequences*. IEEE/ACM transactions on Networking, Vol. 2 No. 5., October 1994

[Mon97] Monroe Robert T., Kompanek Andrew, Melton Ralph, Garland David, Carnegie Mellon University, *Architectural Styles, Design Patterns and Objects*. IEEE Computer, January 1997

[Post83] Postel J. B., Reynolds J. K., *Telnet Protocol Specification*, RFC 854, 15 pages, May 1993, The basic Telnet protocol specification. Many later RFC's describe specific Telnet options.

[Rob96] Roberts Don, Johnson Ralph, *Evolve Frameworks into Domain-Specific Languages*, Procedings of the 3rd International Conference on Pattern Languages for Programming, Monticelli, IL, USA, September 1996, http://st-www.cs.uiuc.edu/users/droberts/evolve.html

[Run91] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., *Object Oriented Modeling and Design*, Prentice Hall, 1991, p146, 7.3 Impact of an Object Oriented approach

[Some96] Ian Sommerville, Software Engineering, 5th edition, Addison Wesley, 1996

[Stev94] Stevens W. Richard, TCP/IP Illustrated Volume 1. The Protocols, Addison –Wesley, Chapter 26 Telnet and Rlogin, 1994

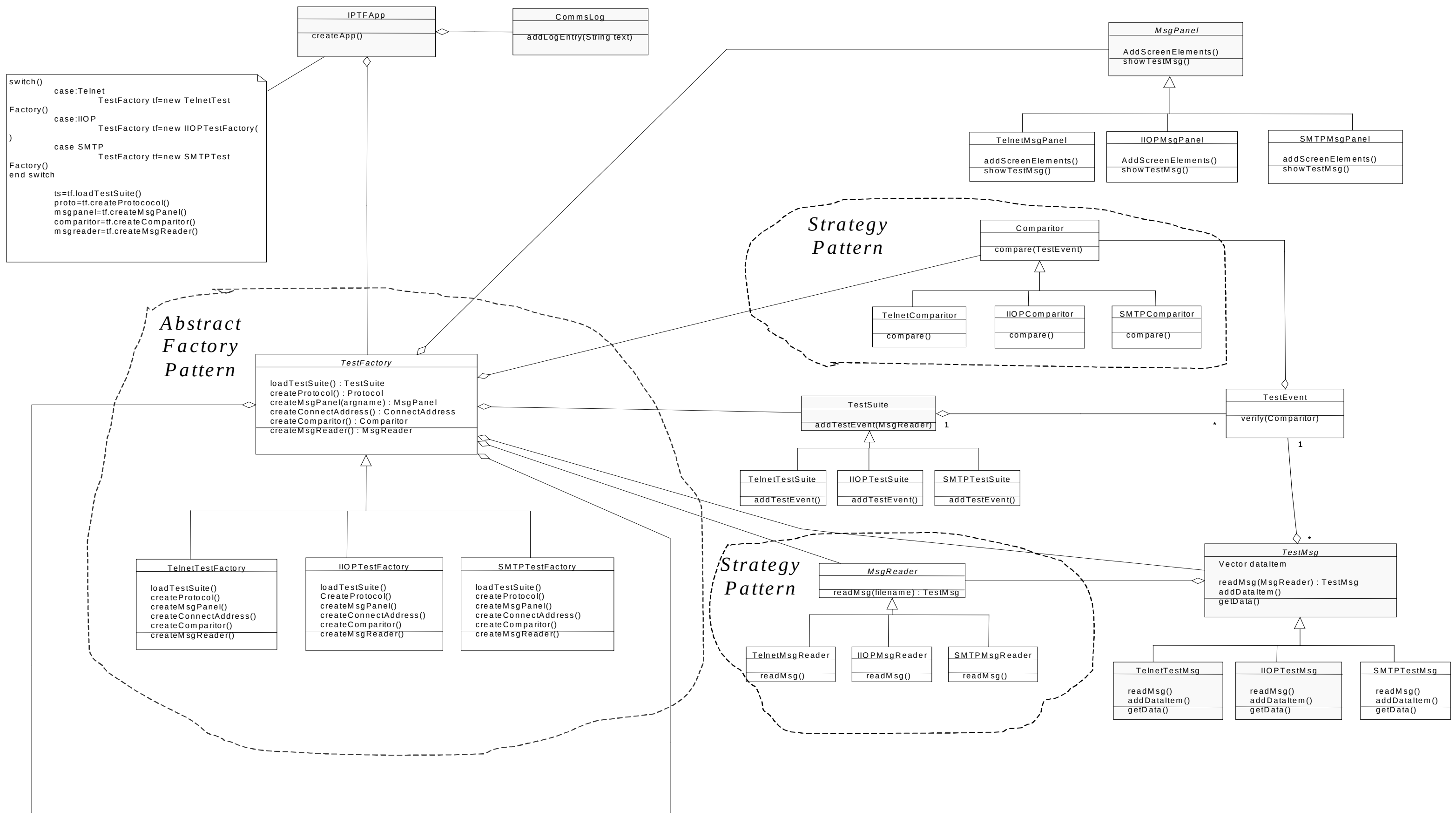
[Tal94a] Taligent Inc., Building Object Oriented Frameworks, A Taligent White Paper, 1994

[Tal94b] Taligent Inc., *Leveraging Object Oriented Frameworks*, A Taligent White Paper, 1994

[Tepf97] Tepfenhart William, Cusack James J., AT&T, A *Unified Object Topology*, IEEE Computer, January 1997

[Voas94] Voas Jeffrey M. , Miller Kieth W, *Software Testability: The New Verification*, IEEE Sortware, May 1995 or Reliable Software Technologies, http://www.rstcorp.com/testability.html

8. APPENDIX A – OBJECT MODEL & SEQUENCE DIAGRAM



```
switch()
          case:Telnet
                    TestFactory tf=new TelnetTest
Factory()
          case:IIOP
                    TestFactory tf=new IIOPTestFactory(
)
          case SMTP
                    TestFactory tf=new SMTPTest
Factory()
end switch

          ts=tf.loadTestSuite()
          proto=tf.createProtococol()
          msgpanel=tf.createMsgPanel()
          comparitor=tf.createComparitor()
          msgreader=tf.createMsgReader()
```

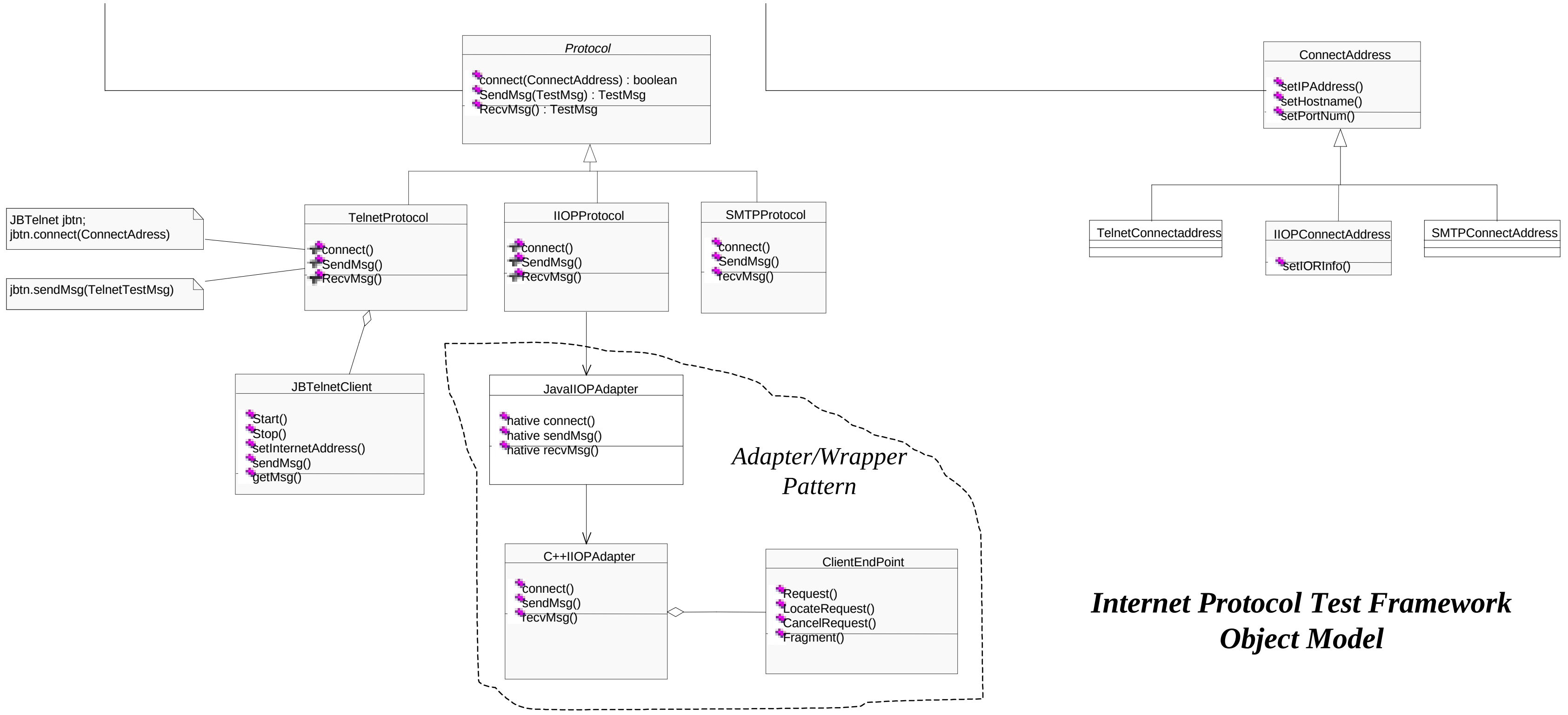


Figure 28: ipTF Object Model

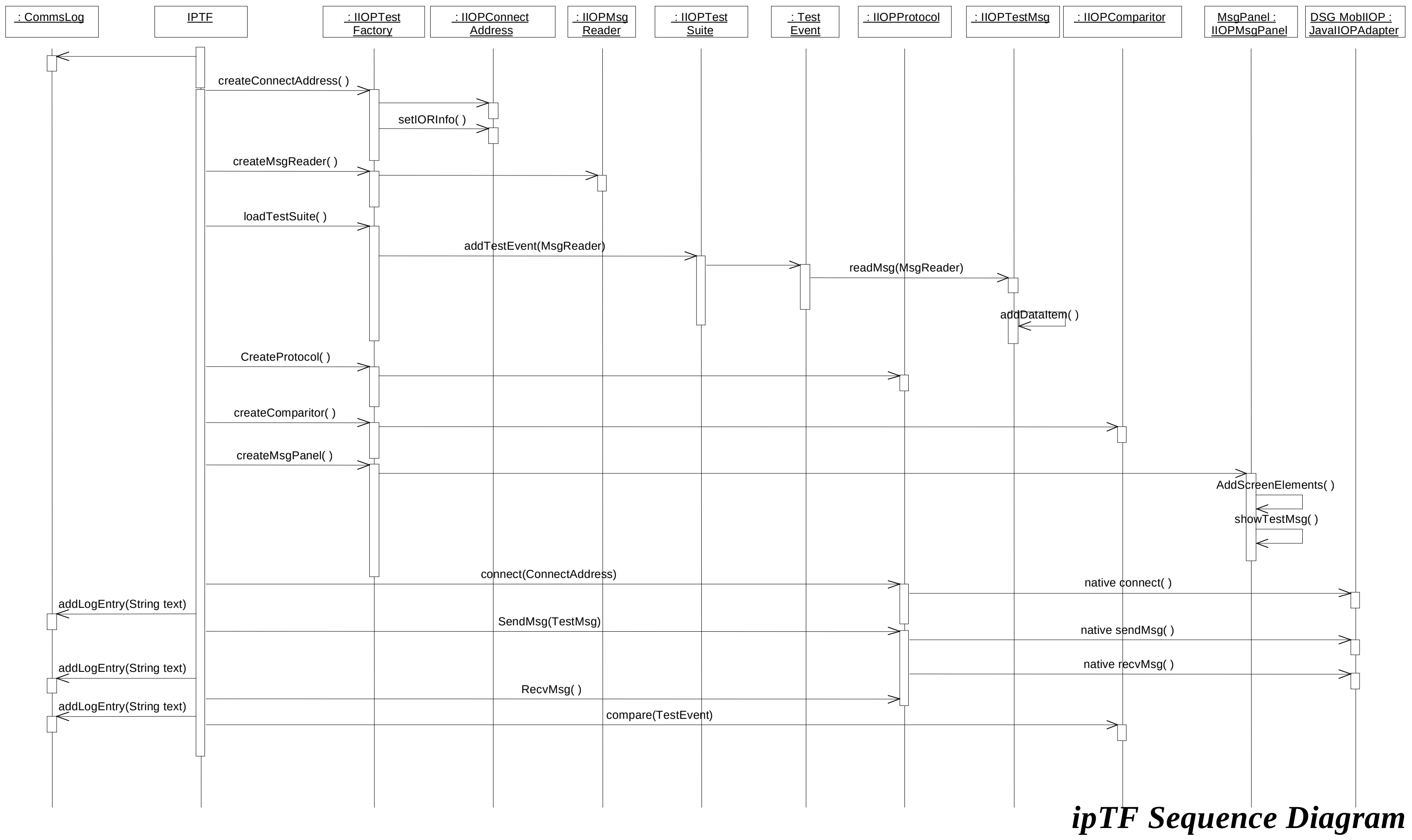


Figure 29: ipTF Sequence Diagram