Master Thesis

Implementation of a TPEG Decoder.

By
Rahul Goel

Supervised By
Prof. Dr.rer.nat. Ralf Moeller
Arbeitsbereiche Softwaresysteme

Prof.Dr.F. Mayer Lindenberg
Arbeitsbereiche Technische Informatik VI

At
Robert Bosch Group, Hildesheim
Department of Corporate Research/Advanced Engineering & Multimedia
Advisor: Frau Elisa Liebanas

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Information and Media Technologies

DECLARATION

I declare that:
I have carried out this work myself, all literally or content-related quotations from other sources are clearly pointed out, and no other sources or aids other than the ones specified are used.

Hildesheim, January 2006.
Rahul Goel.

ACKNOWLEDGEMENT

I am deeply indebted to my supervisor and guide Prof. Dr. Ralf Moeller for his invaluable technical guidance, constructive criticism and moral support provided during my entire tenure of work.

I would like to additionally thank him for providing me an opportunity to work under his guidance and gain valuable experience.

I am also thankful to Frau Elisa Liebanas, my advisor at Robert BOSCH for constant guidance and motivation without which this thesis would not have been possible.

I am grateful to her for providing me the direction and support throughout my tenure in Robert BOSCH Group and also giving me an opportunity to work on a development project during my thesis.

I would also like to give my special thanks to Prof. Ulrich Killat, Course Coordinator Information and Media Technologies and all the teaching staff of Technical University Hamburg-Harburg, Germany for giving me help and encouragement that I needed during my Masters Studies at the University.

Rahul Goel.
24 January, 2006.
Hildesheim, Germany**.**

CONTENTS

CHAPTER 1
INTRODUCTION
1.1 INTRODUCTION [1]

TPEG stands for Transport Protocol Experts Group. TPEG Technology is an upcoming European and ISO pre-standard. TPEG is a new bearer independent TTI (Travel and Traffic Information) broadcast protocol that was initiated by BBC in 1997.The TPEG Project obtained EC funding within the 5th Framework program covering research and development for new information society technologies. TPEG project dealt with the development of language independent and multi – modal traffic and travel information broadcasting for the European citizen. TPEG is developed with the support of major European broadcasters Union and is being developed by EBU (European Broadcasting Union, is an international association that groups all national network/public service broadcasters in Europe.) in full partnership with the receiving industry.

The name "Transport" was chosen to indicate two meaning "Transport "as in the context of traffic and travel ,and also meaning "Transport" in the context of moving information (data) from a service provider to the end user. It was foreseen that the TPEG technology would be able to handle information delivery as far outside the traditional TTI domain, as well as very effectively within this domain.

Digital Audio Broadcasting is a new transmission system bringing the benefits of digital to the world of radio. With Digital Radio capturing the radio market, it has become imperative to concentrate on the Data Services like TPEG which can be carried over DAB.TPEG messages are delivered from service providers to end users, and are used to transfer application data from the database of a service provider to an end user's equipment.

The motive behind TPEG was to develop a new protocol for use in the multimedia broadcasting environment to develop applications, service and transport features which will enable travel related messages to be coded, decoded and filtered.

During the TPEG project many ideas were developed and offered to CEN/ISO for standardization. Now the EBU is supporting the open TPEG Forum to continue development

and maintenance of the TPEG technology specifications and promote implementation of TPEG services.

The TPEG Forum was established with the following objectives:

1. To encourage implementation of TPEG in services and products.
2. To facilitate exchange of information about practical implementation of TPEG.
3. To maintain the existing TPEG standards.
4. To develop new applications of TPEG and, where appropriate to develop new specifications.

1.2 **MOTIVATION** [2,3]

TPEG is relatively new protocol for Travel and Traffic Information. But with the Digital Broadcasting going to replace the present system, it is imperative that in future TPEG will be the ISO standard as other protocol like JPEG, MPEG.

TPEG can be used on many language media (DAB, DVB, Internet etc.) for the broadcasting of language independent TTI. TPEG permits to achieve many innovative Traffic and Travel Information (TTI) services that can support all modes of transport with full language independence and location database independence. The existing system RDS-TMC (Radio Data System–Traffic Message Channel) suffers from the deficiency that it addresses only road traffic. TPEG already support the following three applications:

1. TPEG-RTM/Road Traffic Information
2. TPEG-PTI/Public Transport Information
3. TPEG-SNI/Service and Network Information.

Basically the application generates the TPEG data in the binary format which has to be decoded by the TPEG decoder in the receiver.

A number of other applications are feasible such as Parking Information, Weather Information, Road Congestion status and travel time information, Emergency and Environmental Information etc. TPEG in combination with DAB (digital radio) will revolutionalize the people's opportunities to get more and better traffic information thus making life simpler for many people. Digital Radio will use TPEG Technology to deliver road traffic messages,

public transport information and many other TTI services. The specification for the Road Traffic Message (RTM) Application and SNI (Service and Network) Application are stable, therefore it is required to implement Decoder for the respective applications. In this thesis the major aim is to develop decoder for the SNI application.

1.3 GOAL OF THE MASTER THESIS

1.3.1 PROBLEM DISCUSSION

The problem to be solved consisted of many parts consisting of software engineering problems and problems relating to the TPEG Protocol and Technology. The problem can be described as follows:

1) **Architecture View (Implementation View):** It was of utmost importance to have a portable system, which could be easily re-engineering with minimal effort to work on other environments such as Linux. Also the component had to be reusable.

2) **Robust Design:** The problem also consisted of producing a robust Design which fulfills the Use Case Model. It is felt that design is more important than the code because design visualizes the code and makes it easy for third party including clients to understand the code in a better way and also leaves sufficient room for further enhancements. Also it is a well known fact that coding is a relatively easier process, but having a design guides the programmer to deliver code in an effective way.

3) **Decoder Implementation and Memory Management:** Since there is a resource limitation in terms of memory, it is of prime importance to have effective memory optimization techniques incorporated in the software which prevents memory leaks and subsequent wastage of memory.

4) **Buffer Mechanism Decision:** Since the TPEG signal arrives over DAB continuously, there should be an appropriate Buffer mechanism which handles the incoming data.

1.3.2 IMPLEMENTATION VIEW

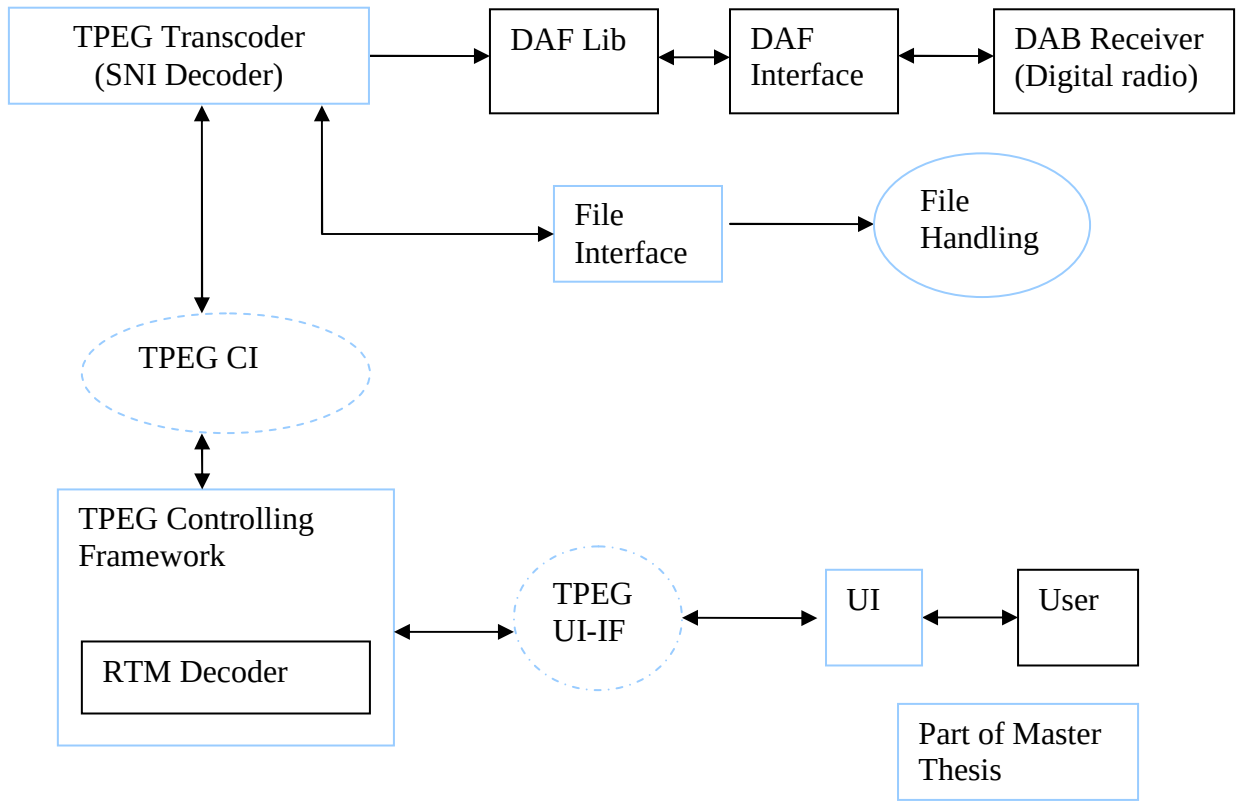

Figure 1.1: Implementation block diagram

The following is the brief explanation of the different components of the diagram:

- TPEG Transcoder
  - Decodes SNI Component
  - Returns the transcoded TPEG Message to TPEG Controlling Framework
  - This component is portable (with trivial modification) and reusable.
- TPEG Controlling Framework
  - Interface between GUI and Transcoder
  - This component is responsible for initiating the decoding process.
- TPEG CI
  - Standard Interface of the Transcoder Component
- TPEG-UI IF
  - Standard Interface between Framework and GUI

1.4 OVERVIEW OF THE REPORT

The second chapter discusses the TPEG Protocol and its characteristics. This chapter primarily deals with developing the background understanding of the TPEG Protocol. The third chapter discusses the Rational Unified Process and its importance as a software engineering process. The fourth chapter describes the process of Use Case Analysis in detail and also puts forward the important result in the analysis and design phase. Finally the fifth chapter discusses the importance of memory optimization and important techniques of memory optimization.

CHAPTER 2

TPEG PROTOCOL

2.1 TPEG BASICS [4]

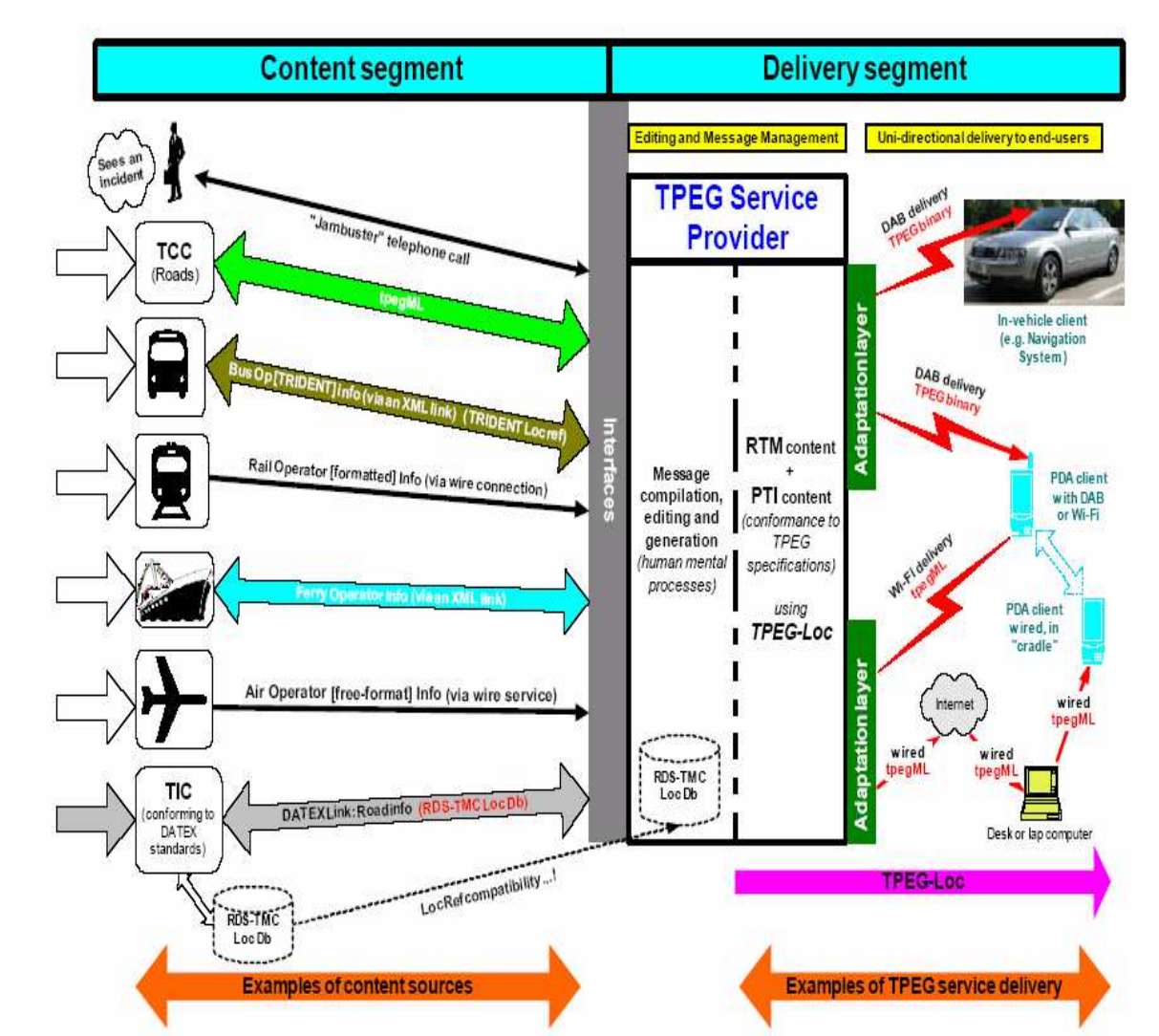

Figure 2.1: TPEG in a NUTSHELL

Traffic and Travel Information content is delivered to end-users by many mechanisms, especially from the public service broadcasters (PSB) who deploy spoken announcements, RDS-TMC, teletext and the internet to deliver such content. But, of course the content has to be collected and edited according to rigorous standards to ensure it is timely and accurate. TTI service provision is therefore all about collection, editing and delivery of information.

To facilitate a good understanding of the processes, two segments have been coined which are shown in the figure 2.1. The *content segment* covers all possible sources of information that

must be collected and processed before the *delivery segment* can be deployed to send the information to the end-user.

2.2 REQUIREMENTS

The requirements which TPEG fulfills are:

1. **Bearer Independence (TPEG data is transported via DAB, DVB, Internet, GSM etc):** The same TTI is transportable via all bearer systems for digital signals (Digital Radio, Digital Television, Internet, Mobile Communication, Networks etc.) in order to reach the customers whenever they are and whichever vehicle they are using.

2. **Language Independence:** User is free to selects his preferred language.

3. **Personalized Information:** Through Advanced filter options (geographical area, severity of events, application selection etc.) the user is able to easily personalize the information presented to him/her.

4. **Multi and Inter Modality:** Information for all transport modes (rail, road, air ...) is provided in order to enable the journey with all transport modes and inter modal mobility.

5. **Independence from the location database on the terminal side**: A generic Location Information Referencing System enables the description of geographical points, segments and areas for all Information with geographical relevance, independent from Transport modes and applications. TPEG uses common forms of location referencing to enable the receiver need not have a location database.

6. **Addresses all kind of terminal** : It addresses all kind of terminal from cheap ones equipped with simple text displays up to devices with digital maps ,GPS receiver etc

7. **Ease Implementation both cost wise and technically**: Offers a high level of precision concerning event description and geographical referencing.

2.2.1 LANGUAGE INDEPENDENCE [5]

RDS-TMC has shown the way for information data delivery serving a mobile end-user who wishes to obtain content when in a locality using a language other than her/his native language. The concept is implemented such that the client device presents information in the language of choice of the end-user. RDS-TMC is limited because it relies upon pre-determined phrases - often not exactly what the service provider would wish to express.

TPEG technology easily satisfies the language independence requirement by using table code values across the “on-air” interface to deliver much of the content.

The tables are extensible, with legacy compatibility. Every table contains a so called “default word” which is a generic word for the content of the table and this allows a client device that does not have the most recent table installed to display the“default word” to convey a slightly more general meaning in the case when it cannot display a word for the actual transmitted code value.

There are various possible client device “models” which the TPEG tables method permits those with embedded tables and those without tables .Specific TPEG client devices(i.e. such as DAB based navigation systems) will be manufactured with the TPEG tables already installed, appropriate to the market in which they are sold. Thus they will be able to display all the words up to their time of manufacture and any extended words used by a service after that time will require the client device to resort to the use of the default word.

In the case of non specific client devices (i.e. devices not built specially for TPEG services and thus not internally equipped with the TPEG tables) then table downloading (of the appropriate language required) at the time of use is implemented, such as when accessing a web based service, delivering tpegML, and using a standard browser to render the content in a suitable language on an appropriate display. This situation will include any extended words and does not require the use of a default word.

TPEG technology goes a step further by "decomposing" the information into essentially single words, which can be more readily translated into various languages. Then the TPEG message construction concept allows for the available information about an event to be

assembled into potentially very rich and informing messages, exactly as the service provider would wish.

2.2.2 FILTERING [6]

TPEG technology has been developed in the context of broadcast service delivery, where messages are delivered to many, many client devices. At any point in time only some of the end-users would wish to receive particular information (e.g. information about traffic jams in a city more than 200 km away is not useful).To allow large amounts of information to be broadcast, and yet not overload the end-users with data of little use to them, the TPEG design philosophy, through explicit coding, is built on the idea of client filtering. This allows end-users to choose messages based on any number of criteria, such as type of event, location, mode of transport, direction of travel etc.

2.2.3 MULTIMODAL APPLICATIONS

TPEG is the first European TTI application that covers all modes of transport across the entire transport landscape. It can serve the motorist in the urban area as well as the bus passenger, the intercity traveler and the long distance driver. TPEG-RTM has been designed to cover Road Traffic Messages regardless of location. It is ideally suited to urban information because of the richness of content that it can offer. But furthermore TPEG technology is designed to facilitate many more applications covering many other aspects of the TTI domain. Already TPEG-PTI allows a service provider to deliver comprehensive public transport information about airplane, bus, ferry, tram, and train services.

It does not attempt to deliver full timetable information, which can be obtained from many other sources already, but it does allow very detailed service/disruption information changes to be delivered to end-users. With the ability to link information it is possible to deliver various alternate routings to a particular destination.

So TPEG technology extends multi-modal information services far beyond anything so far attempted by such technologies as RDS-TMC and puts the delivery of TTI back on track to be a ubiquitous source of information that ideally suits Europe’s mobility objectives.

2.2.4 TPEG HAS NO NEED FOR LOCATION DATABASE IN CLIENT DEVICE [7]

Public Service Broadcasters collect and deliver wide ranging multimodal content, but the possibility for data delivery provided by Europe's first TTI data technology RDS-TMC had significant limitations. The system is essentially limited to inter-urban road events and *every* decoder client must have a location database to interpret *any* message received. This has created a complex situation for all end-users, and this drawback is still not fully resolved.

TPEG technology overcomes this limitation by the introduction of TPEG-Loc, which is a method of delivering very rich location referencing information with *every* message, so that client devices do *no*t need a location database. The biggest advantage is that TTI for densely populated urban areas can now be delivered. Navigation systems with digital maps can "machine read" the location content and localize an event directly onto the map display. A text only client device (e.g. a PDA) is able to present locally found names such as a railway station name and a platform number, directly to an end-user as a text message.

The TPEG location referencing system is built on the principle that the location is generated when needed and not taken from predefined locations stored in a database.

This means that the service provider must have a vector map covering its service area. These maps are quite expensive and most providers using predefined locations like TMC do not have vector maps.

The availability of this map made it possible to develop a tool, which can generate locations on the fly according to the structure in the TPEG location referencing system. The vector map is built from multiple maps layers; each layer contains different types of information. The bottom layers generate contours of the country, lakes etc. By choosing adequate map layers, the area covered by the circle will filter out relevant information for that point, eg: road numbers, street names, community names etc.

It is possible to choose which map layers (e.g. national roads, city, streets etc.) are to be used for each type of location, and there are possibilities to define and edit specific map layers to get relevant information automatically. Even the type of location like different road objects,

building and geographical sites can be utilized by referring to table entries in the actual map layer. The output from the map tool produces the actual location in the locML format.

2.3 TPEG TRANSMISSION

TPEG Technology uses a byte oriented stream format, which may be carried on almost any digital bearer with an appropriate adaptation layer. TPEG messages are delivered from service providers to end users and the used to transfer application data from the database of a service provider to the end user equipment. TPEG is intended to operate via almost any simple digital data channel, and it assumes nothing of the channel other than the ability to convey a stream of bytes.

In Figure 2.2, a variety of possible transmission channels are shown. The only requirement of the channel is that a sequence of bytes may be carried between the TPEG generator and the TPEG decoder. This requirement is described as “transparency“. However it is recognized that data channels may introduce errors. Bytes may be omitted from a sequence, bytes may be corrupted or additional and erroneous data could be received. Therefore TPEG incorporates error detection feature at appropriate points and levels.

There are basically two formats for TPEG messages –tpegML and TPEG binary. The difference between tpegML and TPEG binary concerns only the format and not the content, as both variants are designed to map on onto each other precisely. Therefore the differences concern mainly the size of the data used and the accessibility. As there exist already a lot of software tools and libraries it is comparatively easy to handle messages in XML format as long as there are enough hardware resources and bandwidth. However, in an area of limited resources, one can save memory and/or bandwidth by using the binary format. For this reason the binary format is preferred for the DAB while on the internet it is possible to use both, the binary and the XML.

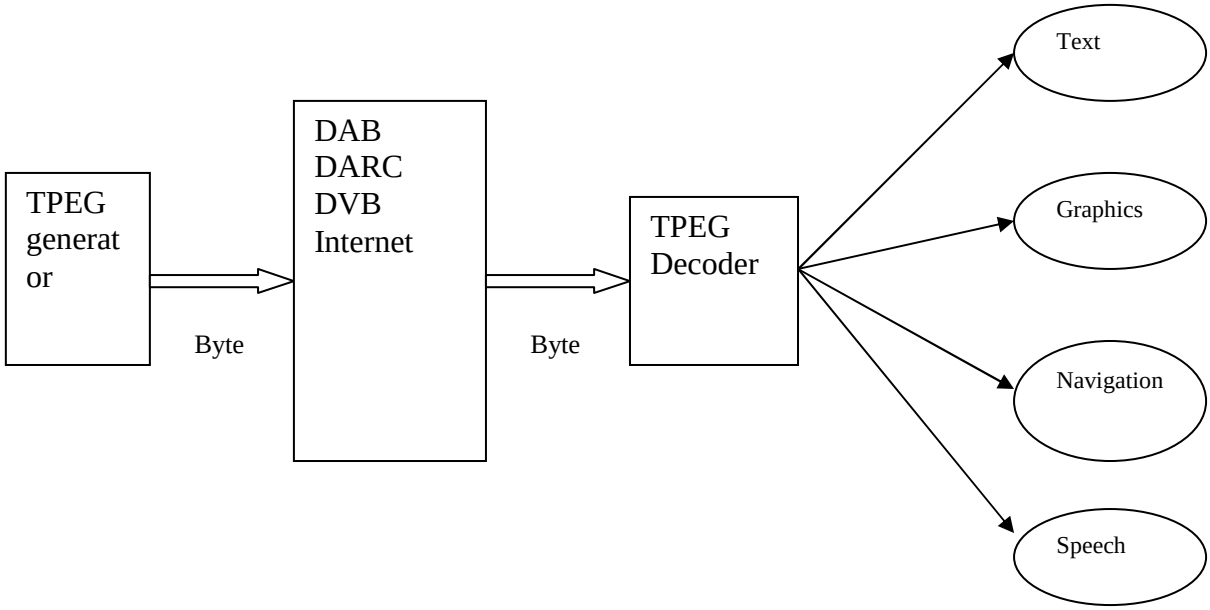

Figure 2.2: TPEG data may be delivered simultaneously via different bearer channels.

2.4 TPEG LAYER MODEL

The TPEG protocol is structured in a layered manner and employs a general purpose framing system which is adaptable and extensible, and which carries frames of variable length. This has been designed with the capability of explicit frame length identification at nearly all levels, giving greater flexibility and integrity, and permitting the modification of the protocol and the addition of new features without disturbing the operation of earlier receiver/decoder model. TPEG application contains all the information by a client TPEG decoder to present all the information intended for the end-user when it was originated by the service provider.

TPEG technology has been designed to be usable for a wide range of applications that require the efficient transmission of point to point and multicast and may easily be encapsulated in Internet Protocol.

- **Layer 1 is the physical layer:** This defines the transmission medium (radio waves, wire, optical etc.) .One particular bearer can make use of different physical layers.

- **Layer 2 is the Data Link layer:** This layer consists of wide range of bearers, which are suitable carriers for the TPEG protocol. An adaptation layer may be required in order to map the TPEG stream onto the bearer.

- **Layer 3 is the Network Layer:** This layer defines the meaning for synchronization and routing.

- **Layer 4 is the Packetization Layer:** Components are merged into a single stream and encrypted and/or compressed.

- **Layer 7 is the Application Layer:** top most level in TPEG.

<table>
<tr><td>Application Layer OSI Layer 7</td><td colspan="5">Service and Network Information Application</td><td></td><td colspan="3">(e.g.) Road Traffic Message Application</td></tr>
<tr><td>Packetization Layer OSI Layer 4,5,6</td><td colspan="9">TPEG Frame Structure</td></tr>
<tr><td>Network Layer OSI Layer 3</td><td colspan="9">Synchronization</td></tr>
<tr><td rowspan="2">Data Link Layer OSI Layer 2</td><td colspan="2">Adaptation</td><td colspan="2">Adaptation</td><td colspan="2">Adaptation</td><td></td><td colspan="2">Adaptation</td></tr>
<tr><td colspan="2">DAB</td><td colspan="2">DVB</td><td colspan="2">DARC</td><td></td><td colspan="2">Internet</td></tr>
<tr><td>Physical Layer OSI Layer 1</td><td colspan="4">Radio Wave</td><td></td><td colspan="4">“piece of wire”</td></tr>
</table>

Figure 2.3: TPEG Protocol Layers

CHAPTER 3
RATIONAL UNIFIED PROCESS
3.1 INTRODUCTION [8, 9]

The Rational Unified Process is iterative software engineering process developed and marketed originally by Rational Software and now IBM. The goal of this process is to produce, within a predictable schedule and budget, high quality software. The RUP is not a single concrete prescriptive process, but rather an adaptable process framework. As such, RUP describes how to develop software effectively using proven techniques. While the RUP encompasses a large number of different activities, it is also intended to be tailored, in the sense of selecting the development processes appropriate to a particular software project or development organization. Rational Software offers a product (known as the Rational Unified Process Product) that provides tools and technology for customizing and executing the process. [24]

The Unified Process has three distinguishing characteristics which are as follows: [10]

- **Use-Case Driven** – A large part of the RUP focuses of modeling. The process employs Use Cases to drive the development process from inception to deployment.

- **Architecture-Centric** - The process seeks to understand the most significant static and dynamic aspects in terms of software architecture. The architecture is a function of the needs of the users and is captured in the core Use Cases. RUP suggests a five view approach. The following summarizes the five views :

  The Logical View: This view of the architecture addresses the functional requirement of the system, what the system should do for its end users. It is an abstraction of the design model and identifies major design packages, subsystems and classes.

  The Implementation View: This view describes the organization of the static software modules in the development environment in terms of packaging and layering and in terms of configuration management.

  The Process View: This view addresses the concurrent aspects of the system at runtime-tasks, threads, or processes as well as their interactions.

The Deployment View: This view shows how the various executables and other runtime components are mapped to the underlying platforms or computing nodes.

The Use Case View: It consists of few key scenarios or use cases. Initially they are used to drive the discovery and design of the architecture.

- **Iterative and Incremental** - The process recognizes that it is practical to divide large projects into smaller projects or mini-projects. Each mini-project comprises an iteration that results in an increment. Iteration may encompass all of the workflows in the process. Iteration is planned using Use Cases.

3.2 FOUR PROCESS PHASES [11]

The Unified Process consists of cycles that may repeat over the long-term life of a system. A cycle consists of four phases: Inception, Elaboration, Construction and Transition. Each cycle is concluded with a release, there are also releases within a cycle. In each phase on progresses iteratively, and each phase consists of one or several iterations .The four phases are as follows:

- **Inception Phase** - During the inception phase the core idea is developed into a product vision. In this phase, reviewing and confirming of the understanding of the core business drivers is done. The basic stress is on understanding the business case for why the project should be attempted. The inception phase establishes the product feasibility and delimits the project scope.

- **Elaboration Phase** - During the elaboration phase the majority of the Use Cases are specified in detail and the system architecture is designed. This phase focuses on the requirements, but some software design and implementation is aimed at prototyping the architecture, mitigating certain technical risks by trying solutions, and learning how to use certain tools and techniques

- **Construction Phase** - During the construction phase the product is moved from the architectural baseline to a system complete enough to transition to the user community. The primary focus is on design and implementation.

- **Transition Phase** - In the transition phase the goal is to ensure that the requirements have been met to the satisfaction of the stakeholders. This phase is often initiated with a beta release of the application. Other activities include site preparation, manual completion, and defect identification and correction. The transition phase ends with a postmortem devoted to learning and recording lessons for future cycles.

3.3 CORE WORKFLOWS

The Unified Process identifies core workflows that occur during the software development process. These workflows include Business Modeling, Requirements, Analysis, Design, Implementation and Test. The workflows are not sequential and likely will be worked on during all of the four phases. The workflows are described separately in the process for clarity but they do in fact run concurrently, interacting and using each other's artifacts.

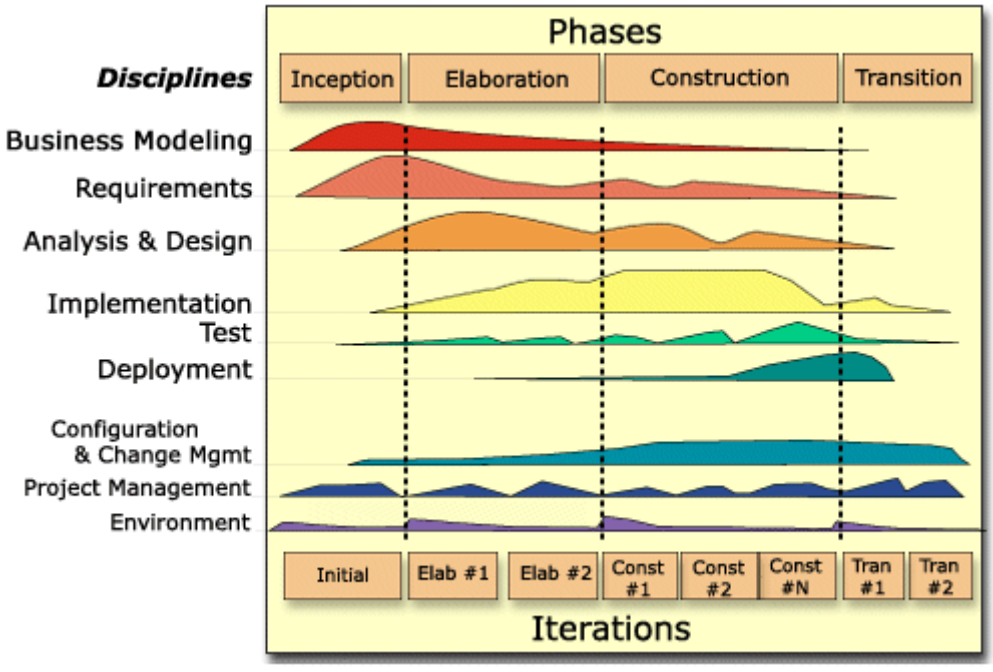

Figure 3.1: Rational Unified Process.

As the "humps" in Figure 3.1 illustrate, the relative emphases of the disciplines change over the life of the project. For example, in early iterations more time is spent on Requirements, and in later iterations more time is spent on Implementation. Configuration and Change

Management, Environment, and Project Management activities are performed throughout the project. However all disciplines are considered within every iteration.

3.4 BEST PRACTICES OF SOFTWARE ENGINEERING

1. Develop Iteratively
2. Manage Requirements
3. Use component Architectures
4. Model Visually
5. Continuously Verify Quality
6. Manage change

3.4.1 DEVELOP ITERATIVELY

Developing iteratively is a technique that is used to deliver the functionality of a system in a successive series of releases of increasing completeness. Each release is developed in a specific, fixed time period called iteration. The earliest iteration addresses the greatest risks. Each iteration includes integration and testing producing an executable release.
Iterations help to accomplish the following:

- Resolve major risks before making large investments.
- Enable early user feedback.
- Focus project short term objective milestones.
- Make possible deployment of partial implementations.

One applies the waterfall model within each iteration and the system evolves incrementally.

3.4.2 MANAGE REQUIREMENTS

Many of the failures in the software development are attributed to incorrect requirement definition from the start of the project to poor requirements management throughout the development lifecycle. Therefore it is very important to manage the requirements throughout the software lifecycle .Requirement management deals with the problem that the right problem is being solved and the right system is being built.
The following are the important aspects of requirement Management:

- Analyze the Problem
- Understand User Needs

- Define the System
- Manage Scope
- Refine the System Definition
- Manage Changing Requirements

Managing Requirements involve the translation of stakeholder requests into a set of key stakeholder needs and system features. These in turn are detailed into specifications for functional and nonfunctional requirements. Detailed specifications are translated into test procedures and user documentation.

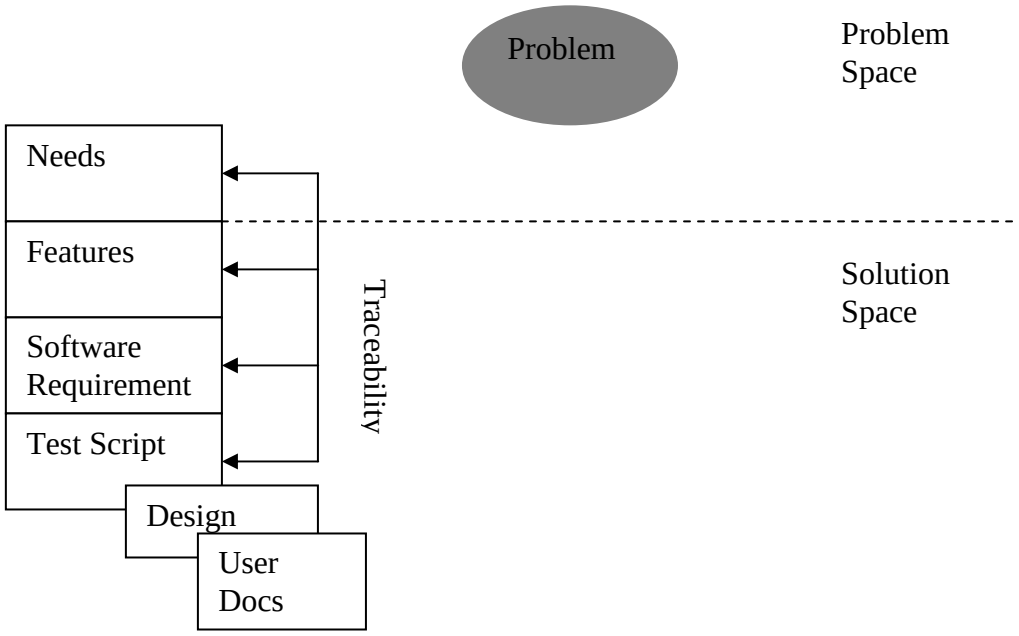

Figure 3.2: Traceability Block Diagram

Traceability allows us to:

- Assess the project impact of a change in a requirement.
- Assess the impact of a failure of a test on requirements (that is, if the test fails, the requirements may not be specified).
- Manage the scope of the project.
- Verify that all the requirements of the system are fulfilled by the implementation.
- Verify that the application does only what it is intended to do.
- Manage change.

3.4.3 USE COMPONENT ARCHITECTURE.

A software component can be defined as a physical, replaceable part of the system that packages implementation, and conforms and provides the realization of a set of interfaces. A component represents a physical piece of the implementation of a system, including software code or equivalent such as scripts or command files. A software system's architecture is perhaps the most important aspect that can be used to control the iterative and incremental development of a system throughout its life cycle.

The most important property of architecture is resilience – flexibility in the face of change. To achieve it, architects must anticipate evolution in both the problem domain and the implementation technologies to produce a design that can gracefully accommodate such changes. Key techniques are abstraction, encapsulation and object oriented analysis and design. The result is that applications are more maintainable and extensible.

3.4.4 MODEL VISUALLY

A model is a simplification of reality that provides a complete description of a system from a particular perspective. Modeling is important because it helps the development team visualize, specify, construct and document the structure and behavior of system architecture.

In building the visual model of a system, many different diagrams are needed to represent different views of the system. The UML provides a rich notation for visualizing models.
This includes the following key diagrams:

- Use Case diagrams to illustrate user interactions
- Class Diagrams to illustrate logical structure
- Object Diagrams to illustrate objects and links
- Deployment diagrams to illustrate physical structure of the software
- Activity Diagrams to show the mapping of software to hardware configurations.
- State chart diagrams to illustrate behavior
- Interaction diagrams to illustrate behavior.

3.4.5 CONTINUOUSLY VERIFY QUALITY

In many organizations, software testing accounts for 30% to 50 % of software development costs. Yet most people believe that software is not well tested before it is delivered. This contradiction is rooted in clear facts. First, testing software is enormously difficult. The different ways particular program may behave are almost infinite. Second, testing is typically done without clear methodology and without the required automation or tool support. While the complexity of software makes complete testing an impossible goal, a well conceived methodology and use of state of art tools can greatly improve the productivity and effectiveness of the software testing. It is lot less expensive to correct defects during development than to correct them after deployment. Important Points are:

- Test for key scenarios ensure that all requirements are properly implemented.
- Poor application performance hurts as much as poor reliability.
- Verify software reliability –memory leaks, bottlenecks.
- Test every iteration – automate test.

3.4.6 BEST PRACTICES REINFORCE EACH OTHER

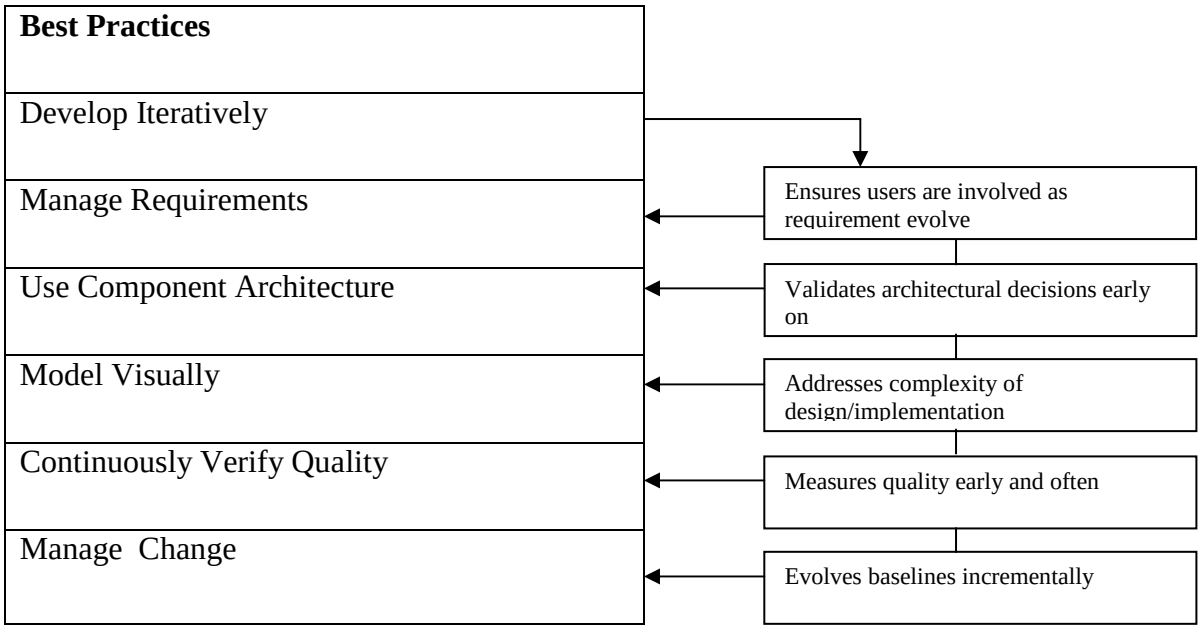

Figure 3.3: Best Practices Reinforce Each Other

Each of the best practices reinforces each other and in some case enables each other. Figure 3.3 shows an example how iterative development leverages the other five best practices. However, each of the other five practices also enhances iterative development .For example iterative development done without adequate requirements management can easily fail to converge into solution. Requirements can change at will, users cannot agree, and the iterations go on forever.

When requirements are managed, this is les likely to happen. Changes to requirements are visible, and the impact on the development process is assessed before the change is accepted. Convergence on a stable set of requirements is ensured .Similarly; each pair of best practices provides mutual support.

3.5 RUP IMPLEMENTS BEST PRACTICES [11]

The RUP captures the best practices in modern software development in a form that can be adapted for a wide range of projects and organizations. The Rational Unified Process provides each team member with the guidelines, templates and tool mentors necessary for the entire team to take full advantage of among others the following best practices. The UML provides a standard for the artifacts of development (semantic models, syntactic notation, and diagrams): the things that must be controlled and exchanged.

The following characteristics of RUP help to implements best practices:

- The dynamic structure of RUP creates the basis of iterative development
- The Project Management discipline describes how to set up and execute a project using phases and iterations
- The Use Case Model of the Requirements discipline and the risk list determine what functionality has to implemented during iteration
- The workflow details of the requirements discipline show the activities and artifacts that make requirement management possible.
- The iterative approach allows to identify components, and to decide which one to develop, which one to reuse and which one to buy.
- The UML used in the process represents the basis of visual modeling and has become the de facto modeling language standard.
- The focus on the software architecture allows articulating the structure: the components, the ways in which they integrate and the fundamental mechanisms and patterns by which they interact.

CHAPTER FOUR

USE CASE ANALYSIS

4.1 REQUIREMENT ENGINEERING [12]

The purpose of the requirement discipline is to:

- To establish and maintain agreement with the customers and other stakeholders on what the system should do.
- To provide system developers with a better understanding of the system requirements.
- To define the boundaries of (delimit) the system.
- To provide the basis for the planning the technical contents of iterations.
- To provide a basis for estimating cost and time to develop the system.
- To define a user interface for the system, focusing on the needs and goals of the users.

The analysis and design discipline gets its primary inputs (Use Case Model and Requirement Specification) from Requirements. Flaws in the Use Case Model can be discovered during Analysis and Design; change requests are then generated, and applied to the Use Case Model.

Relevant Requirements Artifacts:

A use case is an object-oriented modeling construct that is used to define the behavior of a system. The Use Case Model describes what the system will do. The Use Case Model serves as a contract between the customer, the users, and the system developers .It allows customers and users to validate that the system will become what they expect and allow system developers to ensure that what they build is what they expected. The Use Case Model consists of the use cases and actor. Each use case in the model is described in detail, showing step-by-step how the system interacts with the actors and what the system does in the use case. The Use Case Specifications is a document where all of the use case properties are documented (for example brief description and use case flow of events).

4.2 USE CASE DIAGRAMS [13, 14]

The use case view models the functionality of the system as perceived by outside users, called actors. A use case is a coherent unit of functionality expressed as a transaction among actors and the system. Use Case Diagrams are started by identifying as many actors as possible. One should ask how the actors interact with the system to identify an initial set of use cases. If actor supplies information, initiates the use case, or receives any information as a result of the use case, then there should be an association between them.

Associations are modeled as lines connecting use cases and actors to one another, with an optional arrowhead on one end of the line. The arrowhead is often used to indicating the direction of the initial invocation of the relationship or to indicate the primary actor within the use case.

Figure 4.1 shows the three types of relationships between use cases -- extends, includes, and inheritance -- as well as inheritance between actors. The extend relationships can be thought of as the equivalent of a "hardware interrupt" because one does not know when or if the extending use case will be invoked (perhaps a better way to look at this is extending use cases are conditional). Include relationships is the equivalent of a procedure call. The include relationship points to the use case to be included; and extend relationship points to the use case to be extended. Inheritance (Generalization) is applied in the same way as on the Class Diagram-- to model specialization of use cases or actors in this case. It can be said as relationship between a general use case and a more specific use case that inherits and adds features to it. It is also possible one use case uses another use case, by an arrow with the title uses.

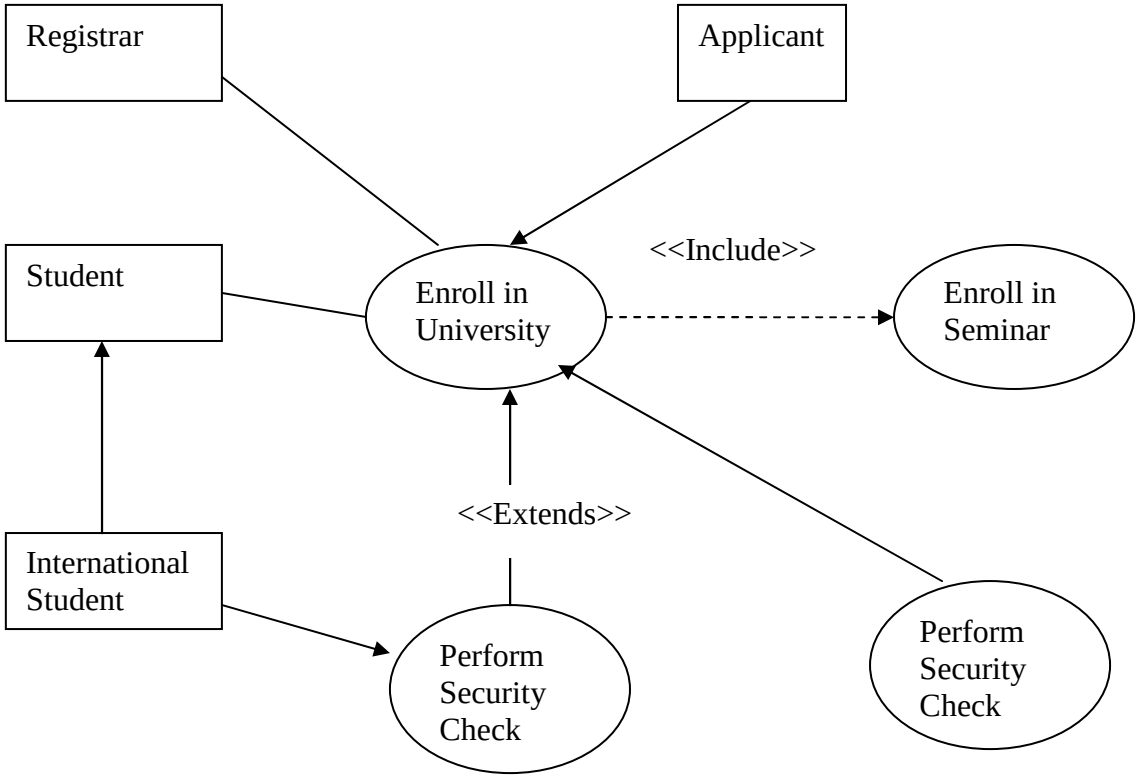

Figure 4.1: Use Case Diagram Example

4.3 USE CASE ANALYSIS [15, 12]

Use case analysis is performed to identify the initial classes of our system. After making an initial attempt at defining the architecture, the key abstractions and some key analysis mechanisms, use case analysis is performed. The initial architecture along with the software requirements defined in the requirement discipline guides and serve as an input to the Use Case Analysis activity. An instance of Use Case Analysis is performed for each use case to be developed during iteration. The focus during Use Case Analysis is on a particular use case. Use case analysis is performed by the designer, once per iteration per use case Realization.

Use Case Analysis Steps:

1. Supplement the Use Case Description
2. For each Use Case Realization
    - Find Classes from Use Case Realization
    - Distribute Use Case behavior to Classes.
3. For each resulting analysis class:
    - Describe responsibilities.
    - Describe Attributes and Associations.
    - Qualify Analysis Mechanisms
4. Unify Analysis Classes.

Stereotypes and Use Case Realization are two important concepts. The following is the explanation:

Stereotypes

Stereotypes define a new model element in terms of another model element.

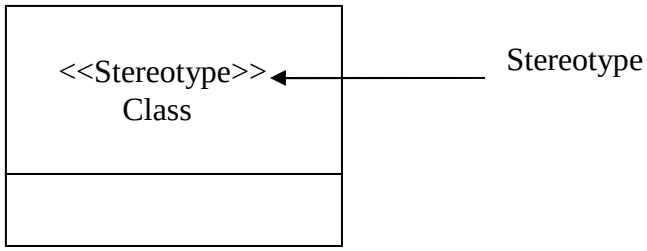

Figure 4.2: Stereotype Representation

A stereotype can be defined as:

An extension of the basic UML notation that allows defining a new modeling element based on an existing modeling element.

- The new element may contain additional semantics but still applies in all cases where the original element is used.
- The name of the stereotype is shown in guillemets.
- A unique icon may be defined for the stereotype, and the new element may be modeled using the defined icon or the original icon with the stereotype name displayed.
- Stereotypes can be applied to all modeling elements, including classes, relationships, and components and so on.
- Each UML element can have only one stereotype.

Use Case Realization

A use case realization describes how a particular use case is realized within the design model in terms of collaborating objects. A realization relationship is drawn from the use–case realization to the use case it realizes.

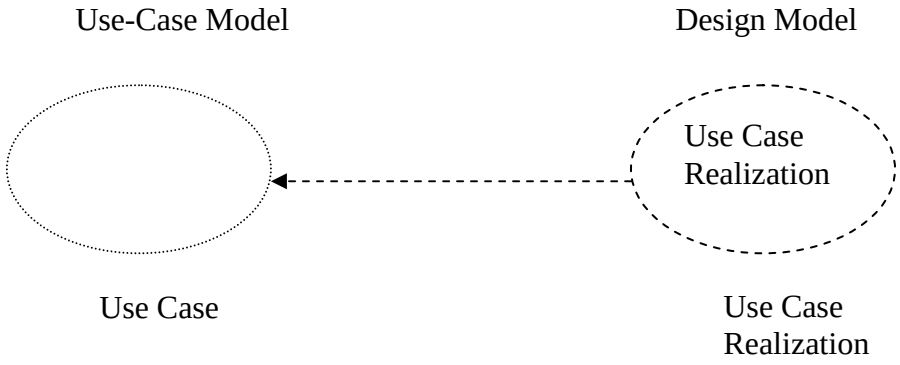

Figure 4.3: Depiction of Use Case Realization

A Use Case Realization can be represented using a set of diagrams like Sequence diagrams and Class Diagrams. During Use Case Analysis the Use Case Realization diagrams are outlined. In subsequent design activities these diagrams are refined and updated according to more formal class interface definitions.

4.3.1 SUPPLEMENT THE USE CASE DESCRIPTION

The use case description developed in the Requirement Engineering phase is enhanced to include enough details to begin developing a model. The primary aim of the Supplement the description of the Use Case is to capture additional information needed in order to understand the required internal behavior of the system that may be missing from the Use Case Description. In some cases it may be found out that some of the requirements were incorrect or not well understood then the original use case flow of events should be updated and iterate again the requirement engineering.

4.3.2 FIND CLASSES FROM USE CASE BEHAVIOR

The use case flow of events is analyzed to identify the analysis classes and allocate the use case responsibilities to the analysis classes. To find the objects that perform the use case, a "white box" description of what the system does from an internal perspective is needed.

The purpose of the find classes from use case behavior step is to identify a candidate set of model elements (analysis classes) that will be capable of performing the behavior described in the use case.

Analysis Classes

Finding a candidate set of roles is the first step in the transformation of the system from a mere statement of required behavior to a description of how the system will work. The analysis classes taken together represent an early conceptual model of the system. This conceptual model evolves quickly and remains fluid for some time as different representations and their implications are explored. Analysis classes are "proto-classes" which is essentially "clump of behavior" .These analysis classes are early conjectures of the composition of the system: they rarely survive intact into implementation. They provide with a way of capturing the required behaviors in a form that can be used to explore the behavior and composition of the system. Analysis classes permits to "play" with the distribution of responsibilities, re-allocating as necessary. The technique for finding analysis classes uses three different perspectives of the system to drive the identification of candidate classes:

These three perspectives are:

- The boundary between the system and its actors(Boundary Class)

- The information the system uses(Entity Class)
- The control logic of the system(Control Class)

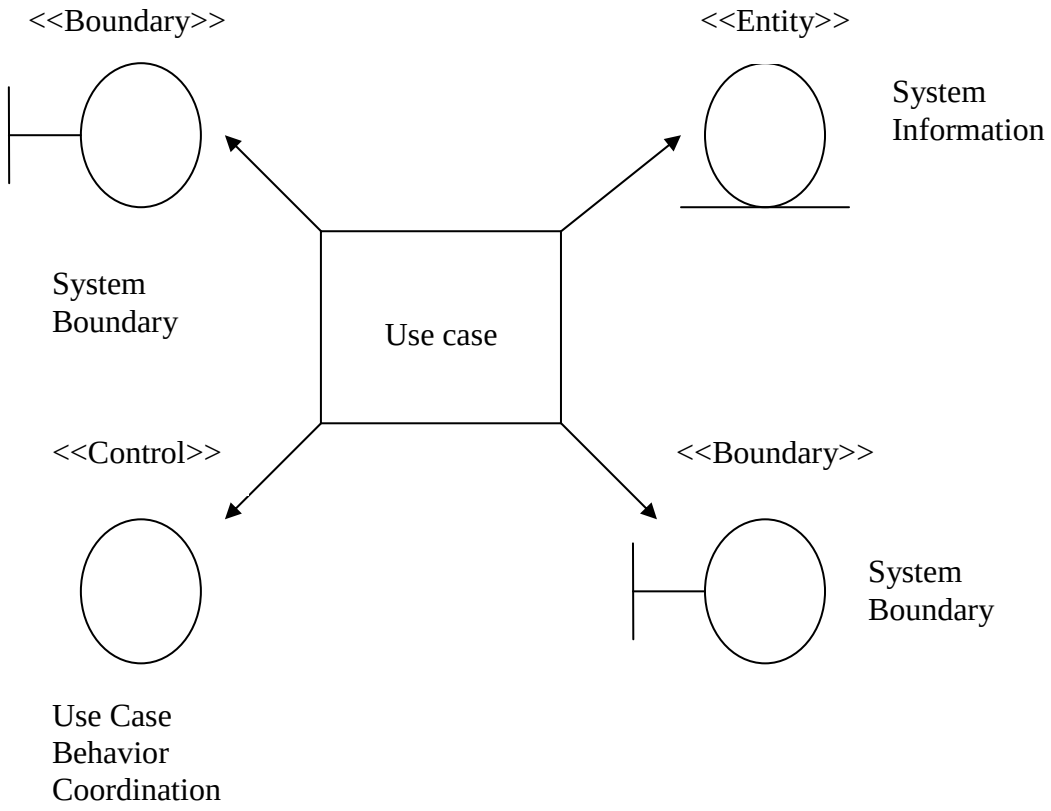

Figure 4.4: The complete behavior of use case is distributed to analysis classes.

Boundary Classes

These are intermediates between the interface and something outside the system. Boundary Classes insulate the system from changes in the surroundings (for example, changes in interfaces to other systems and changes in user requirements), keeping these changes from affecting the system. A system can have several types of boundary classes:

User Interface Classes: Classes that intermediate communication with human users of the system.

System Interface Classes: Classes that intermediate communication with other systems.

Device Interface Class: Classes that provide the interface to devices which detect external events .These boundary classes capture the responsibilities of the device or sensor.

A boundary classes is used to model interaction between the systems surroundings and its inner workings. Such interaction involves transforming and translating events and noting changes in the system presentation (such as interface).Boundary classes model the parts of the system that depend on its surroundings. They make it easier to understand the system because they clarify the system's boundaries and aid design by providing a good point of departure for identifying related services. Because boundary classes are used between actors and the working of the external system (actors can only communicate with boundary classes), they insulate the external forces from internal mechanisms and vice versa. One way of the initial identification of boundary classes is one boundary class per actor /use-case pair. This class can be viewed as having the responsibility of coordinating the interaction with the actor. This may be refined as a more detailed analysis is performed. This is particularly true for window-based GUI applications where there is typically one boundary class for each window, or one for each dialog box.

Entity Classes

Entity classes represent stores of information in the system. They are typically used to represent the key concepts the system manages. Entity objects are used to hold and update information about some phenomenon, such as an event, a person or a real life object. They are usually persistent, having attributes and relationships needed for a long period, sometimes for the lifetime of the system. The main responsibility of entity class is to store and manage information in the system. An entity object is not usually specific to one Use – Case Realization and sometimes it is not even specific to the system itself. The values of its attributes and relationships are given by an actor. Entity objects have the behavior as complicated as that of other object stereotypes.

Taking the use case flow of events as input, noun phrases are underlined in the flow of events. These form the initial candidate list of analysis classes. Next a series of filtering steps are applied where some candidate classes are eliminated. The result of this filtering exercise is a refined list of candidate entity classes. Normally it is also easier to identify the entity classes by reading the use case specification carefully.

Control Class

Control Class provides coordinating behavior in the system. The system can perform some use cases without control classes by just using entity classes and boundary classes. This is

particularly true for the use cases that involve only the simple manipulation of stored information. More complex use cases require one or more control classes to coordinate the behavior of other objects in the system. Control classes effectively decouple boundary and entity objects from one another, making the system more tolerant of changes in the system boundary. They also decouple the use-case specific behavior from entity objects, making them more reusable across use cases and systems.

Control classes provide behavior that:

- Is surrounding independent (does not change when the surroundings change).
- Defines control logic (order between events) and transactions within use case.
- Changes little if the internal structure or behavior of the entity classes changes.
- Uses or sets the contents of several entity classes, and therefore needs to coordinate the behavior of these entity classes.
- Is not performed in the same way every time it is activated(flow of events feature several states)

Control objects (instances of control classes) often control other objects, so their behavior is of the coordinating type. Control classes encapsulate use-case-specific behavior. When the system performs the use case, a control object is created. Control objects usually die when their corresponding use case has been performed.

4.3.3 DISTRIBUTE USE –CASE BEHAVIOR TO CLASSES

The responsibilities of the use case is allocated to the analysis classes and this allocation is modeled by describing the way the class instances collaborate to perform the use case in Use Case Realization. The purpose of “Distribute Use – Case Behavior to Classes Use case “is:

- Express the use case behavior in terms of collaborating analysis classes.
- Determine the responsibilities of analysis classes.

The allocation of responsibilities in analysis is a crucial and sometimes difficult activity. The three stereotypes mentioned above makes the process easier by providing a set of canned responsibilities that can be used to build a robust system. These predefined responsibilities isolate the parts of the system that are most likely to change.

A driving influence on where the responsibility should go is the location of the data needed to perform the operation. The best case is that there is one class that has all the information needed to perform the responsibility .In that case, the responsibility goes with the data.
If this is not the case then the responsibility may need to be allocated to a “third party” class that has access to the information needed to perform the responsibility. Classes and relationship might need to be created to make this happen.

4.3.4 DESCRIBE RESPONSIBILITIES

At this point, analysis classes have been identified and use case responsibilities have been allocated to those classes. This was done on a use case by use case basis, with the focus primarily on the use-case flow of events. Now it is time to turn attention to each of the analysis classes and see what each of the use cases will require of them.

The ultimate objective of these class focused activities is to document what the class knows and what the class does. The resulting Analysis Model gives a big picture and a visual idea of the way responsibilities are allocated and what such an allocation does to the class collaborations. The purpose of describe responsibilities step is to namely describe the responsibilities of the analysis classes.

A responsibility is a statement of something an object can be asked to provide. Responsibilities evolve into one or more operations on classes in design; they can be characterized as:

- The actions that the object can perform.
- The knowledge that the object maintains and provides to other objects.

The View of participating classes(VOPC) class diagram contains the classes whose instances participate in the Use Case Realization Interaction diagrams, as well as the relationships required to support the interactions.

4.3.5 QUALIFY ANALYSIS MECHANISMS

At this point a good understanding of the analysis classes, their responsibilities, and the collaborations required to support the functionality described in the use cases has been developed.
The purpose of the Qualify Analysis Mechanisms step is to:

- Identify analysis mechanisms used by the class.
- Provide additional information about how the class applies the analysis mechanism.

An Analysis mechanism represents a pattern that constitutes a common solution to a common problem. These patterns may show patterns of structure, patterns of behavior or both. They are used during Analysis to reduce the complexity of Analysis, and to improve the consistency its consistency by providing designers with a shorthand representation for the complex behavior.

Mechanisms allow the Analysis effort to focus on translating the functional requirements into software concepts without bogging down into specification of relatively complex behavior needed to support the functionality but which is not central to it. Analysis mechanism often result from the instantiation of one or more architectural or analysis patterns. Persistence provides an example of analysis mechanisms. A persistent object is one that logically exists beyond the scope of the program that created it. During analysis, one does not want to be distracted by the details of how one is going to achieve the persistence. This gives rise to a “persistence” analysis mechanism that allows to speak of persistent objects and capture the requirements one will have on the persistence mechanism without worrying about what exactly the persistence mechanism will or how it will work As analysis classes are identified, it is important to identify the analysis mechanisms that apply to the identified classes. For example the classes that are persistent are mapped to the persistency mechanism.

4.3.6 UNIFY ANALYSIS CLASSES

The purpose of the Unify Analysis Classes is to ensure that each analysis classes represents single well defined concept, with non overlapping responsibilities. Different Use Cases will contribute to the same classes .A class can participate in any number of Use Cases. It is therefore important to examine each class for consistency across the whole system. Merge

Classes that define similar behaviors or that represent the same phenomenon. Merge entity classes that define the same attributes, even if their defined behavior is different; aggregate the behaviors of the merged classes.

4.4 IDENTIFY DESIGN ELEMENTS

In Identify Design Elements, the analysis classes are refined into design elements (design classes and subsystems). The purpose of Identify Classes and Subsystems is to refine the analysis classes into appropriate Design Model Elements. The architect performs Identify Design Elements once per iteration.

The following are the steps for the Identify Design Elements:

- Identify classes and subsystems
- Identify subsystem interfaces
- Identify reuse opportunities
- Update the organization of the design Model

4.4.1 IDENTIFY CLASSES AND SUBSYSTEMS

The purpose of Identify Classes and subsystems is to refine the analysis classes into appropriate design model elements (for example classes or subsystems). Analysis classes handle primarily functional requirements, and model objects from the “problem” domain; design elements handle the nonfunctional requirements, and model objects from the “solution” domain. It is in Identify Design Elements that the decision is taken which analysis “classes” are really classes , which are subsystems (which must be further decomposed), and which are existing components and do not need to be “designed “ at all.

4.4.2 IDENTIFY DESIGN CLASSES

If the analysis class is simple and already represents a single logical abstraction, then it can be directly mapped, one-to-one, to a design class. Typically, entity classes survive relatively intact into design. Throughout the design activities, some analysis classes can be split, joined, removed or otherwise manipulated. In general there is many to many mapping between analysis classes and design elements. The possible mapping includes the following:

An analysis class can become:

- One single class in the design model.
- A part of a class in the design model.
- An aggregate class in the design model (meaning that the parts in this aggregate may not be explicitly modeled in the Analysis class).
- A group of classes that inherits from the same class in the design model.
- A group of functionality related classes in the design model (for example, a package)
- A subsystem in the design model.
- A relationship in the design model.
- A relationship between analysis classes can become a class in the design model.
- Part of an analysis class can be realized by hardware and not modeled in the design model at all.
- Any combination of the above.

4.4.3 GROUP DESIGN CLASSES IN PACKAGES.

When identifying classes, one should group them into packages, for organizational and configuration management purposes. The design model can be structured into smaller units to make it easier to understand. By grouping Design Model elements into packages and subsystems, then showing how those groupings relate to one another, it's easier to understand the overall structure of the model.

There are different reasons for partitioning the design model:

- Packages and subsystem can be used as order, configuration, and order delivery units when a system is finished.
- Allocation of resources and the competence of different development teams might require that the project be divided among different groups at different sites.
- Subsystems can be used to structure the Design Model in a way that reflects the user types. Many changes requirement originate from users; subsystems ensure that changes from a particular user types will affect only the parts of the system that correspond to that user type.
- Subsystems are used to represent the existing products and services that the system uses.

Subsystem and Interfaces

A subsystem is a model element that has the semantics of a package, such that it can contain other model elements, and a class, such that it has behavior. A subsystem realizes one or more interfaces, which define the behavior it can perform. A subsystem encapsulates its implementation behind one or more interfaces. Interfaces isolate the rest of the architecture from the details of the implementation.

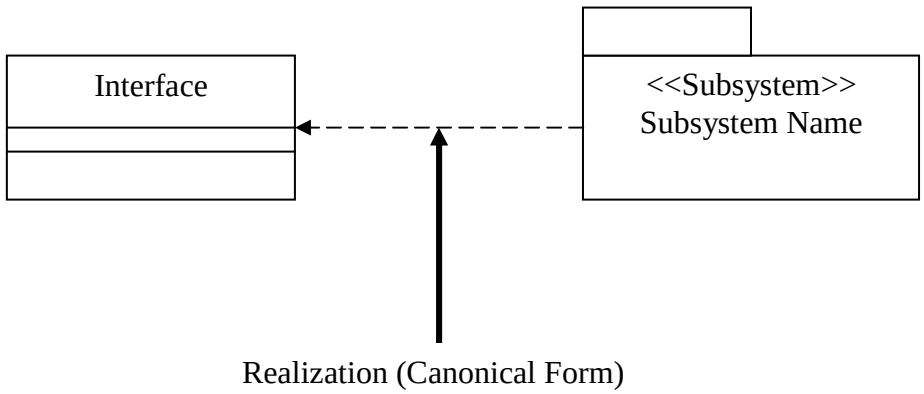

Figure 4.5: A Subsystem and Interface

An interface is a model element that defines a set of behaviors(a set of operations) offered by a classifier model element (specifically a class ,subsystem , or component).The relationship between interfaces and classifiers (subsystem) is not always one to one .An interface can be realized by multiple classifiers, and a classifier can realize multiple interfaces.

Interfaces are a natural evolution from the public classes of a package to abstractions outside the subsystem. All classes inside the subsystem are then private and not accessible from the outside. Operations defined for the interface are implemented by one or more elements contained within the subsystems. The benefit of this is that, unlike a package, the contents and internal behaviors of a subsystem can change with complete freedom, so long as the subsystem's interfaces remain constant. A subsystem provides interfaces by which the behavior it contains can be accessed. Packages provide no behavior; they are simple containers of things that have behavior. With packages it is impossible to substitute packages for another one unless they have the same public classes. The public classes and their public operations get frozen by the dependencies that external classes have on them. Thus the designer is not free to eliminate these classes or change their behavior.

Subsystems can be used to partition the system into parts that can be independently:

- Ordered, configured or delivered
- Developed, as long as the interface remains.
- Deployed across a set of distributed computational nodes.
- Changed without breaking other parts of the system.

4.3.4 IDENTIFY SUBSYSTEM INTERFACES

Interfaces define a set of operations that are realized by some classifier. In the Design Model, interfaces are principally used to define the interfaces for subsystems. This is not to say that they cannot be used for classes as well .But as a single class it is usually sufficient to define public operations on the class. These operators, in effect, define its "interface". Interfaces are important for subsystems because they allow separation of the declaration of behavior (the interface) from the realization of behavior (the specific classes within the subsystem that realize the interface).This de-coupling provides us with a way to increase the independence of development teams working on different parts of the system, while retaining precise definitions of the "contracts" between these different parts. The interfaces are completely defined using their signatures. This is important, as these interfaces will serve as synchronization points that enable parallel development.

4.4.5 IDENTIFY REUSE OPPORTUNITIES

The identification of reuse opportunities is an important architectural step. It will help to determine which subsystems and packages to develop, which one to reuse, and which one to buy. It is desirable to reuse existing components, wherever possible. The identification of the reuse opportunities is a unification effort, since it is determined if "things" that have been identified can be satisfied by what already exists.

The advent of commercially successful component infrastructure such as CORBA, the Internet, ActiveX and JavaBeans trigger a whole industry of off the shelf components for various domains, allowing buying and integrating components rather than developing them in-house. Reusable components provide common solutions to a wide range of common problems and may be larger than just collections of utilities or class libraries; they form the basis of reuse within an organization, increasing overall productivity and quality.

4.4.6 UPDATE THE ORGANIZATION OF THE DESIGN ELEMENTS

As new elements have been added to the design model, repackaging the elements of the design model is necessary. Repackaging achieves several objectives- reduces coupling between packages and improves cohesion within packages in the design model. The ultimate goal is to allow different packages (and subsystems) to be designed and developed independently of one another by separate individuals or teams. As new model elements are added to the system, existing packages may grow too large to be managed by a single team: the package must be split into several packages which are highly cohesive within the package but loosely coupled. Doing this may be difficult –some elements may be difficult to place in one specific package because they are used by elements of both packages. There are two possible solutions:

- Split the elements into several objects, one in each package
- Move the elements into a package in a lower layer, where all higher layer elements might depend upon it equally.

Layering provides a logical partitioning of packages into layers with certain rules concerning the relationship between layers. Restricting the inter-layer and inter-package dependencies makes the system more loosely coupled and easier to maintain. Failure to restrict dependencies causes architectural degradation, and makes the system brittle and difficult to maintain.

4.5 ANALSYSIS AND DESIGN RESULTS [16,17,18,19]

4.5.1 USER INTERFACE PROTOTYPE

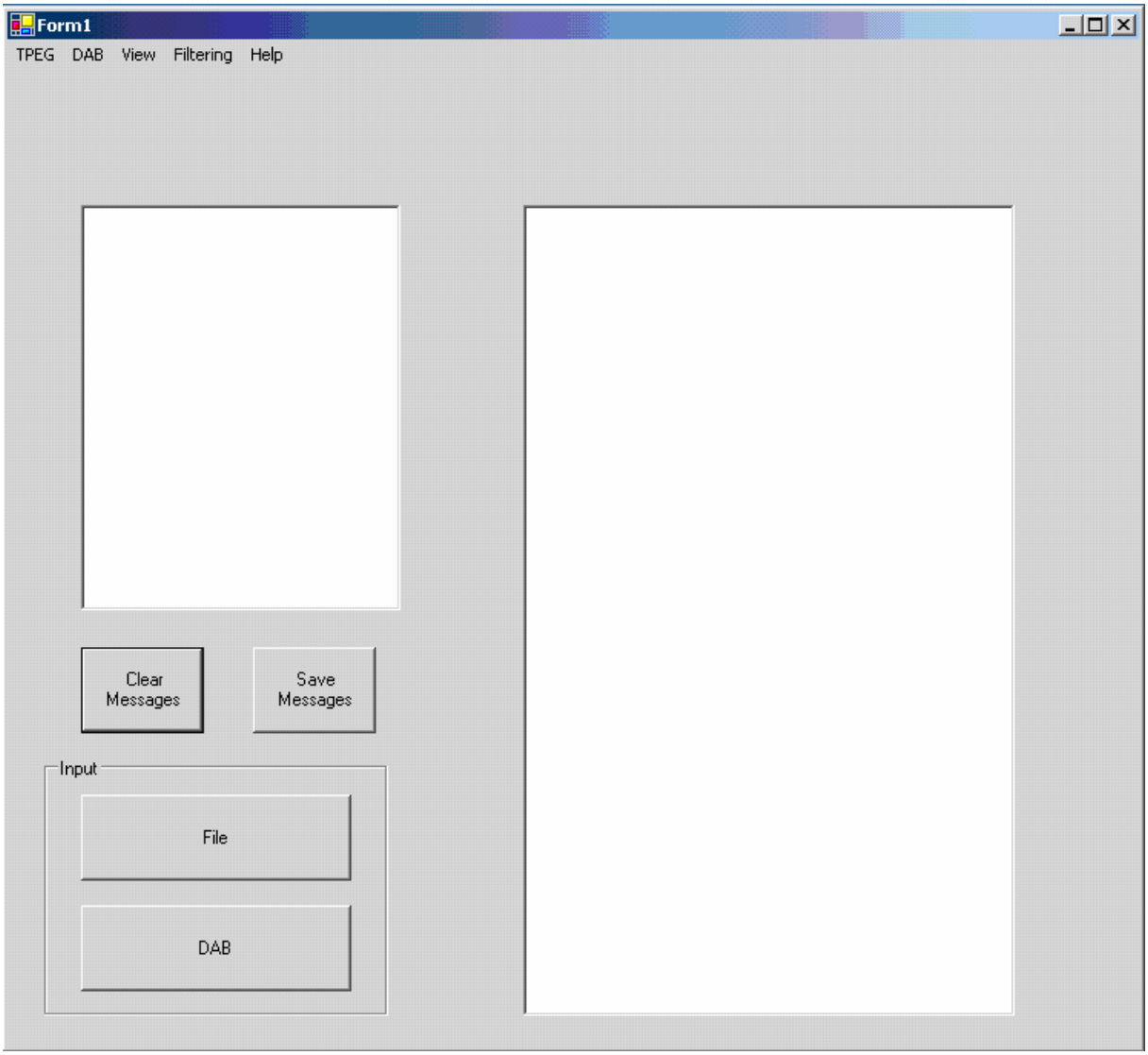

Figure 4.6: User Interface Prototype

Brief Explanation: The interface is implemented in VB.net. The menu controls the different options of the interface like save and read directory locations, filtering settings and switching from Engineering (with more detail output) to Normal View. There are two channels –File and DAB, as the decoder reads data from the two channels. The Output consists of list of decoded messages and corresponding messages in XML format.

4.5.2 USE CASE MODEL

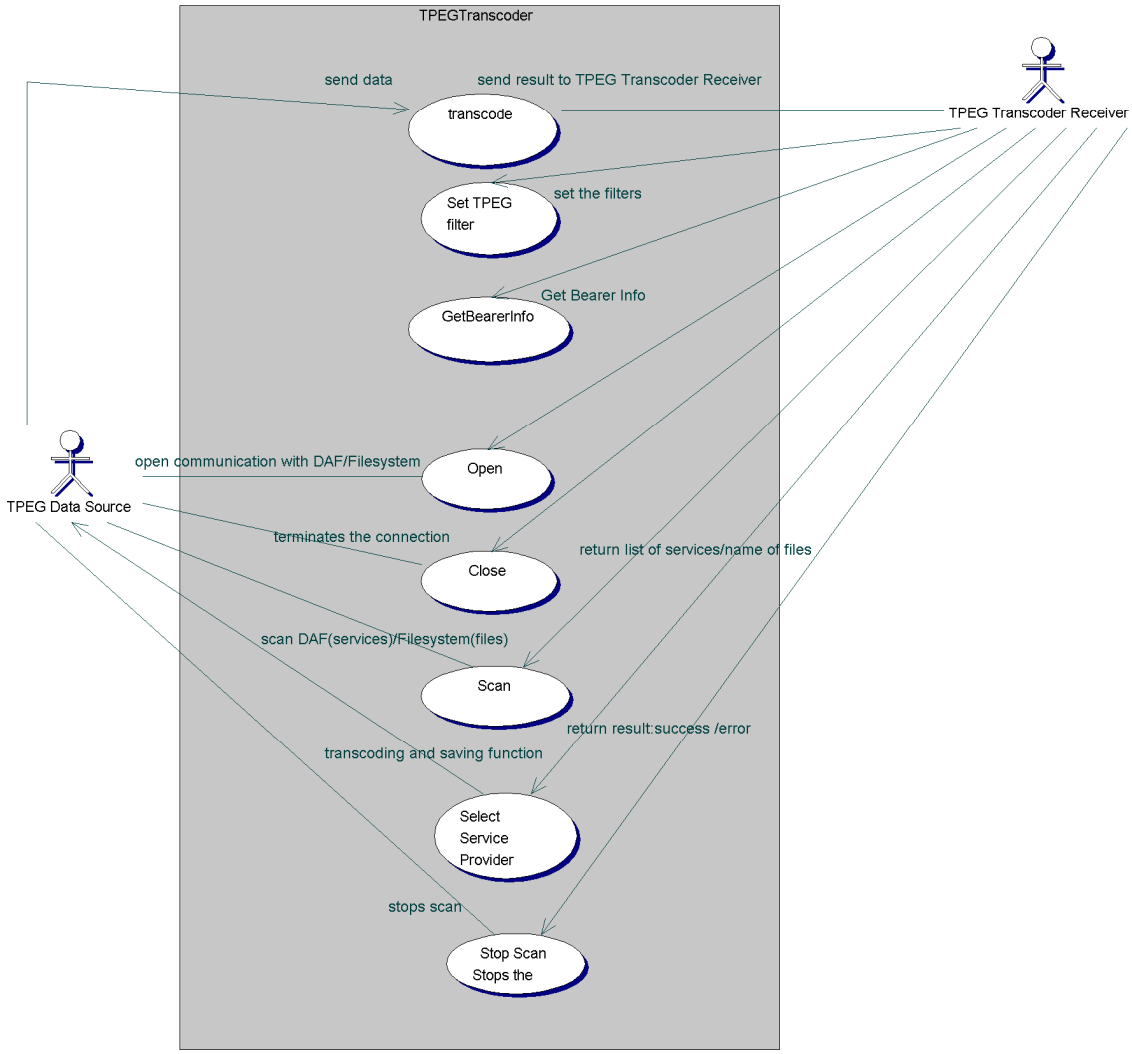

Figure 4.7: Use Case Model

Brief Explanation: The requirement engineering resulted in the Use Case Model. The following are the different use cases overview:

- Open: Opens the communication to the DAB/directory.
- Close: Closes the communication to the DAB/directory.
- Scan: Scans the DAB (for the list of services) or from directory (for the list of files).
- Select Service: Reads the data and start transcode.

The actor TPEG Data Source is the source of the TPEG Data–either Filesystem or DAB and the actor TPEG Transocoder Receiver is the TPEG Controlling Framework (Implementation View).

4.5.3 CLASS DIAGRAM

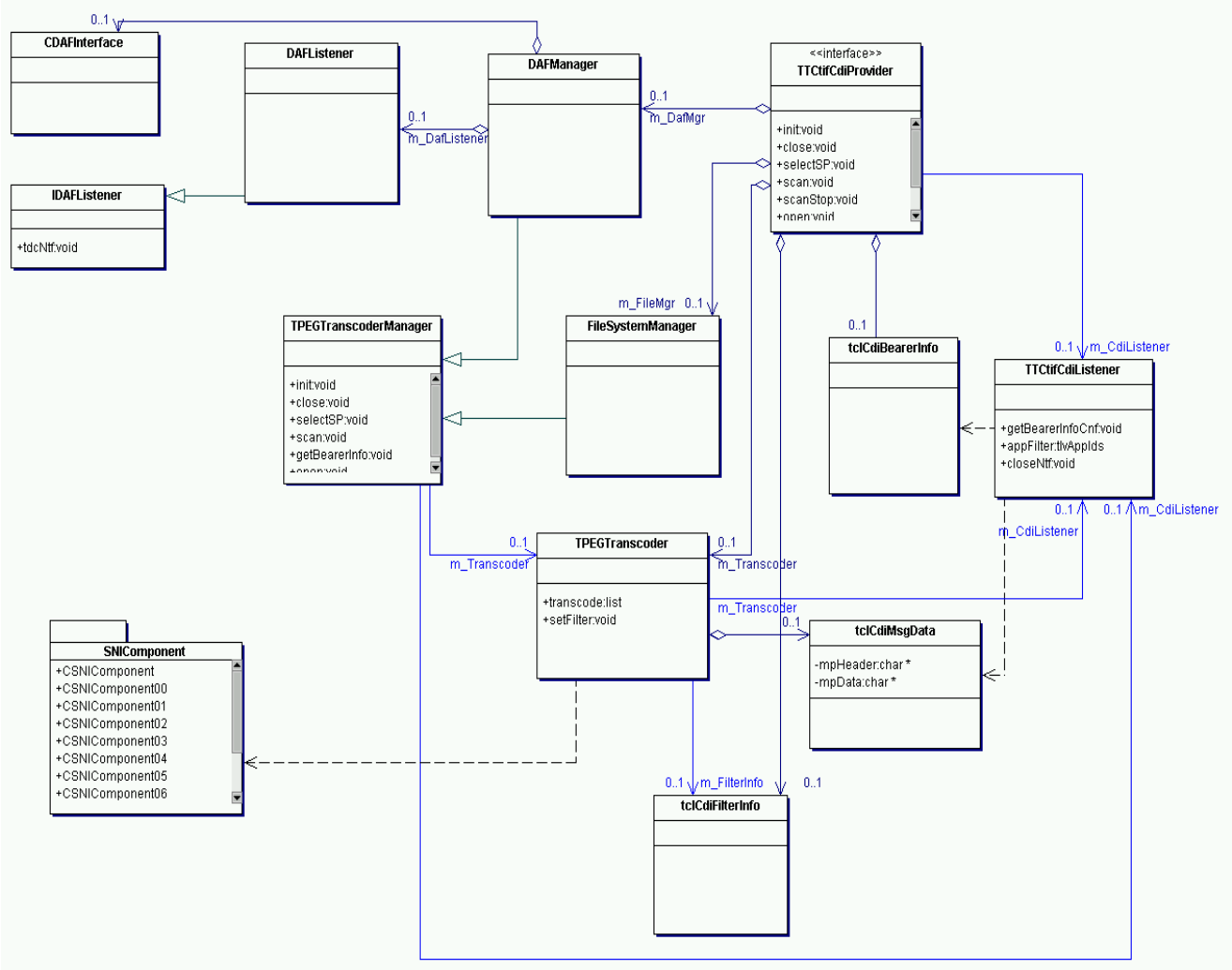

Figure 4.8: Class Diagram

Brief Explanation: The two classes TTCCdiProvider and TTCCdiListener are the interface classes for the transcoder component. They follow the event listener architecture. The Class DAFManager and FileSystemManager are controlling classes for the respective mediums. DAFListener is the interface to the DAF component which helps to interact with the DAB. TPEGTranscoder is the main class which is contains the logic for the transocding and decoding of the SNI Component.

4.5.4 STATE DIAGRAM

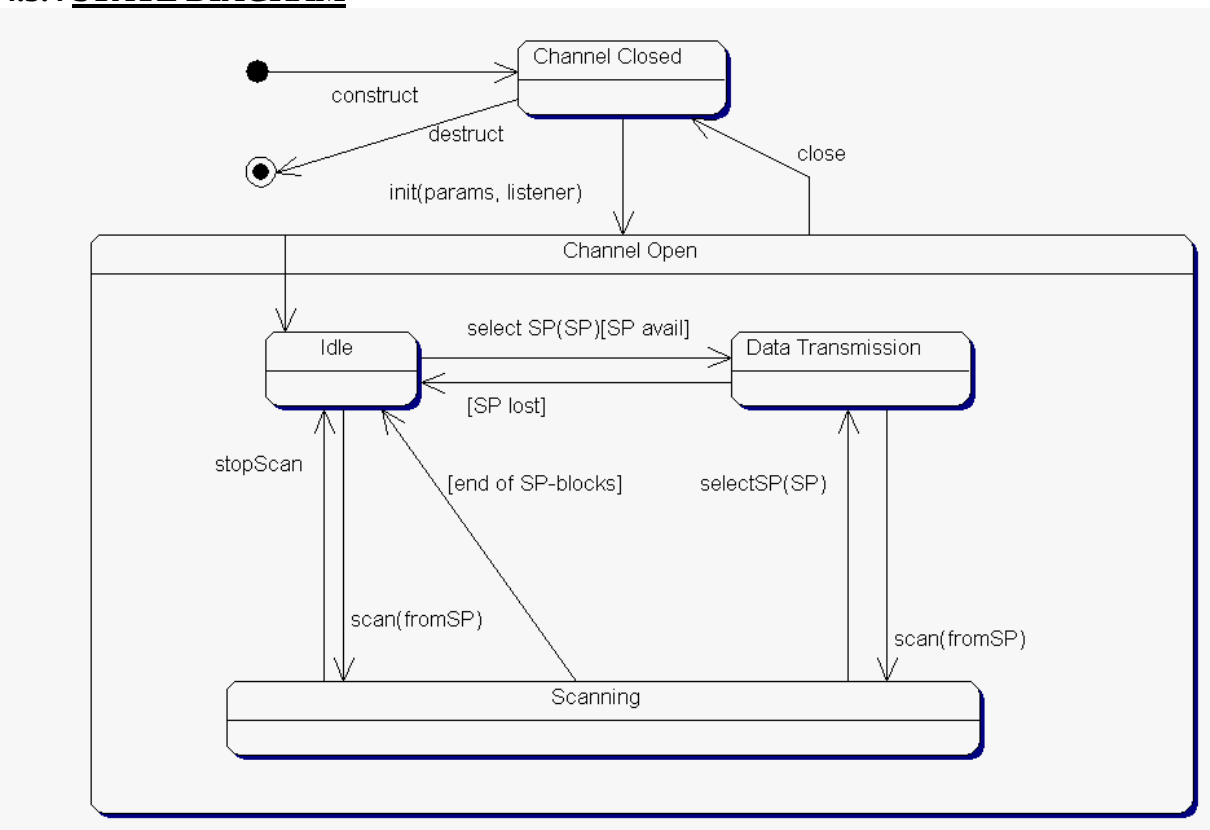

Figure 4.9: State Diagram

Brief Explanation: The above is the state diagram for the transcoder. The initial state is the "Channel Closed" state. Upon the initialization the state changes to "Channel Open, Idle". It can go to the "Scanning state" in case the Service (DAB of File) is not available .If the service (DAB of File) is available then it goes to the "Data Transmission" state.

4.5.5 USE CASE REALIZATION

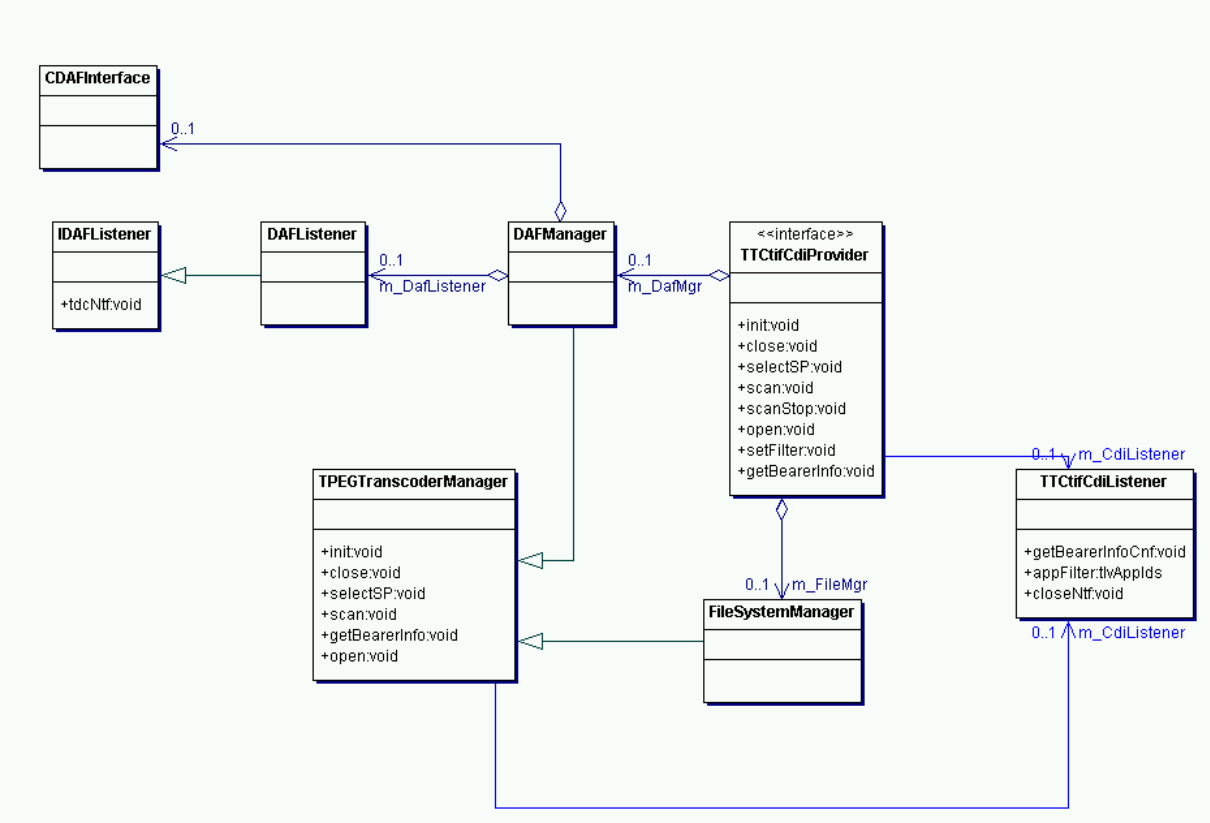

Figure 4.12: Use Case Realization for Scan Use Case

Brief Explanation: This is the Use Case Realization of the Scan Use case. It consists of the classes which participate to complete the scan use case.

CHAPTER FIVE
MEMORY MANAGEMENT
5.1 INTRODUCTION

The memory system often determines a great deal about the behavior of an embedded system: performance, power, and manufacturing cost. A great many software techniques have been developed over the past decade to optimize software to improve these characteristics.

The memory system is a main contributor to the performance and power consumption of embedded software; the total amount of memory required is also a main component of the manufacturing cost of the hardware. Because memory system behavior is so important to embedded systems, a great deal of attention has been paid over the past decade to the optimization of the memory characteristics of embedded software. [5, 6]

Memory management is usually divided into three areas: hardware, operating system, and application, although the distinctions are a little fuzzy. The Memory Management is mostly concerned with application memory management.

Hardware memory management: Memory management at the hardware level is concerned with the electronic devices that actually store data. This includes things like RAM and memory caches.

Operating system memory management: In the operating system, memory must be allocated to user programs, and reused by other programs when it is no longer required. The operating system can pretend that the computer has more memory than it actually does, and also that each program has the machine's memory to itself; both of these are features of virtual memory systems.

Application memory management: Application memory management involves supplying the memory needed for a program's objects and data structures from the limited resources available, and recycling that memory for reuse when it is no longer required. Because application programs cannot in general predict in advance how much memory they are going to require, they need additional code to handle their changing memory requirements.

Application memory management combines two related tasks:

Allocation: When the program requests a block of memory, the memory manager must allocate that block out of the larger blocks it has received from the operating system. The part of the memory manager that does this is known as the allocator.

Recycling: When memory blocks have been allocated, but the data they contain is no longer required by the program, then the blocks can be recycled for reuse. There are two approaches to recycling memory: either the programmer must decide when memory can be reused (known as manual memory management); or the memory manager must be able to work it out (known as automatic memory management).

5.2 FORM OF NEW AND DELETE [20]

Memory management concerns in C++ fall into two general categories: getting it right and making it perform efficiently. Getting things right means calling memory allocation and deallocation routines correctly. Making things perform efficiently, on the other hand, often means writing custom versions of the allocation and deallocation routines.

Use the same form in corresponding uses of new and delete

Ex: *string * stringArray =new string [100];*
…
delete stringArray;

Everything above appears to be in order – the use of new is matched with the use of delete, but the programs behavior is undefined. At the very least, 99 of the 100 string objects pointed to by stringArray are unlikely to be properly destroyed, because their destructors will not be properly destroyed.

When new operator is used two things happen – First memory is allocated, second one or more constructors are called for that memory .When delete is used two things happen: one or more destructors are called for that memory, and then the memory is de-allocated. Now the question is that does the pointer being deleted point to a single object or to an array of objects? The only way for delete to know is to use brackets in delete.

The correct code is as follows:

*string * stringArray =new string [100];*

…

delete [] stringArray;

Use Delete on pointer members in destructors.

Classes performing dynamic memory allocation will use new in the constructor to allocate the memory and will later use delete in the destructor to free up the memory. This is easy to get right when first writing the class, provided of course, that it is remembered to employ delete on all the members that could have been assigned memory in any constructor. However the situation becomes more difficult as classes are maintained and enhanced because the programmers making the modifications to the class may not be the ones who wrote the class in the first place. Under these conditions, it's easy to forget that adding a pointer member almost always requires each of the following:

- Initialization of the pointer in each of the constructors. If no memory is to be allocated to the pointer in a particular constructor, the pointer should be initialized to 0(i.e. the null pointer).
- Deletion of the existing memory and assignment of new memory in the assignment operator.
- Deletion of the pointer in the destructor.

Failing to delete the pointer in the destructor, however, exhibits fairly no obvious external symptoms. Instead it manifests itself as a subtle memory leak. It is also useful to delete the null pointer in the destructor.

5.3 OUT OF MEMORY CONDITIONS

When operator new can't allocate the memory requested, it throws an exception. To solve this problem a common C idiom is to define a type independent macro to allocate memory and then check to make sure the allocation succeeded. For C++, such a macro might look something as follows:

#define NEW (PTR, TYPE)

try

```
{
PTR =new TYPE;
}
catch
{
std::bad_alloc&
}
assert (0);
```

But the above macro suffers from the common error of using an assert to test a condition that might occur in the production code (after all one can run out of memory at any time), but it also has a drawback specific to C++: it fails to take into account the myriad ways in which new can be used. There are three syntactic forms for getting new objects of type T, and one need to deal with the possibility of exceptions for each of this forms:

```
new T;
new T (constructor arguments);
new T[size];
```

This simplifies the problem, because clients can define their own overloaded versions of operator new, so programs may contain an arbitrary number of different syntactic forms for using new. It is easier to set up things in such a way that if the request for memory is not satisfied, an error handling function is called. This strategy relies on the convention that when the operator new cannot satisfy a request, it calls a client specifiable error handling function often called a *new handler* before it throws an exception.
To specify the out of memory handling function, clients call *set_new_handler*, which is specified in the header <new> more or less like this:

```
typedef  void(* new_handler)();
new_handler set_new_handler(new_handler p) throw()
```

new_handler is a typedef for a pointer to a function that takes and returns nothing, and *set_new_handler* is a function that takes and returns a *new_handler*.

set_new_handler's parameter is a pointer to the function operator new should call if it can't allocate the requested memory. The return value of *set_new_handler* is a pointer to the function in effect for that purpose before *set_new_handler* was called.

set_new_handler is used as follows:

```
//function to call if operator new can't allocate enough memory
void noMoreMemory()
{
cerr<<"Unable to satisfy request to memory\n";
Abort ();
}
int main()
{
set_new_handler (noMoreMemory);
int *pBigDataArray =new int[1000000000];
…
}
```

As it seems likely, operator new is unable to allocate space for so many integers, *noMoreMemory* will be called, and the program will abort after issuing an error message. When operator new cannot satisfy a request for memory, it calls the new handler function not once, but repeatedly until it can find enough memory.

5.3.1 NEW_HANDLER FUNCTION

A well defined new-handler must do one of the following:

- *Make more memory available:* This allow operator new's next attempt to allocate the memory to succeed. One way to implement this strategy is to allocate a large block of memory at program start up and then release it the first time the new handler is invoked. Such a release is often accompanied by some kind of warning to the users that memory is low and the future requests may fail unless the memory is somehow made available.

- *Throw an exception:* Throw an exception of type std::bad_alloc or some type derived from std::bad_alloc .Such exceptions will not be caught by operator new, so they will

propagate to the site originating the request for memory .(Throwing an exception of different type will violate operator new's exception specification.) The default action when that happens is to call abort so if the new handler is going to throw an exception, definitely want to make sure it's from std::bad_alloc hierarchy.

- *Not Return:* Typically by calling abort or exit, both of which are found in the standard C library.

- *Deinstall the new-handler:* i.e. pass the null pointer to *set_new_pointer*. With no new handler installed, operator new will throw an exception of type std::bad_alloc when it attempts to allocate memory is unsuccessful.

- *Install a different new-handler:* If the current new handler can't make any more memory available, perhaps it knows of a different new handler that is more resourceful. If so, the current new handler can install the other new handler in its place (by calling *set_new_handler*). The next time operator new calls the new handler function it will get the one most recently installed.

This choice gives a considerable flexibility in implementing new handler functions.
C++ has no support for class specific new handlers. This behavior can be implemented by having each class provide its own versions of *set_new_handler* and operator new. The classes *set_new_handler* allows clients to specify the new handler for the class (just like the standard *set_new_handler* allows clients to specify the global new handler).The class's operator new ensures that the class specific new handler is used in place of the global new handler when memory for class objects is allocated. Consider a class X for which the memory allocation failures have to be handled:

```
Class X
{
public:
static new_handler set_new_handler(new_handler p);
static void * operator new (size_t size);
private:
static new_handler currentHandler;
};
```

Static class members must be defined outside the class definition.

```
new_handler X::currentHandler; //sets currentHandler to 0 (i.e., null) by default
```

The *set_new_handler* function in class X will save whatever pointer is passed to it. It will return whatever pointer had been saved prior to the call. This is what the standard version of *set_new_handler* does:

```
new_handler X::set_new_handler(new_handler p)
{
new_handler oldHandler = currentHandler;
currentHandler =p;
return oldHandler;
}
```

The X operator new can be defined as follows:

```
void * X::operator new(size_t size)
{
new_handler globalHandler = std::set_new_handler(currentHandler);
void *memory;
try
{
memory=::operator new(size);
}
catch(std::bad_alloc&)
{
std::set_new_handler(globalHandler);
throw;
}
std::set_new_handler(globalHandler);
return memory;
}
```

Clients of class X use its new-handling capabilities like this:

```
void noMoreMemory();                    //declaration of function to call if memory
                                        allocation for X objects  fails
X::set_new_handler(noMoreMemory);   //set noMoreMemory as X's new-handling function
X *px1 = new X;                     //if memory allocation fails, calls noMoreMemory
string *ps =new string;             //if memory allocation fails, calls the global new-
                                    handling function (if  there is one).
X::set_new_handler(0);              //set the X-specific new-handling function to nothing(i.e.
                                    Null)
X * px2 = new X;                    //if memory allocation fails, throw an exception immediately.
                                    (There is no new-handling function for class X).
```

Using *set_new_handler* is a convenient, easy way to cope with the possibility of out of memory conditions. Certainly it's a lot more attractive than wrapping every use of new inside a try block.

Until 1993, C++ required that operator new return 0 when it was unable to satisfy a memory request. The current behavior is for operator new to throw a std::bad_alloc exception, but a lot of C++ was written before compilers began supporting the revised specification. The C++ standardization committee didn't want to abandon the established test for 0 code base so they provided alternative forms of operator new (and operator new []) that continue to offer the traditional failure yields 0 behavior. These forms are called "nothrow" forms because, they never do a throw, and they employ nothrow objects (defined in the standard header <new>) at the point where new is used:

```
Class Widget { };
Widget *pw1= new Widget;
if (pw1==0)
Widget *pw2 = new (nothrow) Widget;
if (pw2==0)…
```

Regardless of whether one uses " normal" (i.e. exception –throwing) new or "nothrow" new, it's important to be prepared to handle memory allocation failures .The easiest way to do that is to take advantage of *set_new_handler.*

5.3.2 OPERATOR NEW AND DELETE [25]

Conventions should be followed when writing operator new and operator delete. When writing the operator new the following things should be taken care of.

Right Return Value: If the required memory is available then a pointer to it is returned .If not then the exception of type std::bad_alloc is returned.

Calling an error handling function: Operator new actually tries to allocate memory more than once, calling the error handling function after each failure, the assumption being that the error handling function might be able to do something to free up some memory .Only when the pointer to the error handling function is null does operator new throw an exception.
In addition C++ requires that operator new returns a legitimate pointer even when 0 bytes are requested.
Pseudo Code for a non member new looks like as follows:

```
void * operator new (size_t size)
{
if (size==0)
{
size=1;
}
while (true)
 {
//attempt to allocate sizes bytes;
if (the allocation was successful)
return (a pointer to the memory);
//allocation was unsuccessful, find out what was the current error handling function is
new_handler globalhandler =set_new_handler(0);
set_new_handler(globalhandler);
```

```
if (globalHandler)(*globalHandler());
else throw std::bad_alloc();
}
}
```

Unfortunately there is no way to get at the error-handling function pointer directly, so only was to call *set_new_handler* to find out what it is.

For delete the things are less complex. The following is the pseudocode for the operator delete:

```
void operator delete (void *rawMemory)
{
if (rawMemory == 0) return; //do nothing if the null pointer is being deleted.
//deallocate the memory pointed to by the rawMemory
return;
}
```

One of the common mistakes is that the operator new can be inherited by subclasses. Most of the class specific versions of operator new are designed for a specific class, not for a class or any of its subclasses. That is, given an operator new for a class X, the behavior of that function is almost always carefully tuned for objects of size sizeof(X) – nothing larger and nothing smaller .Because of inheritance, however it is possible that the operator new in a base class will be called to allocate memory for an object of a derived class. The best way to handle this is to slough off calls requesting the “wrong” amount of memory to the standard operator new:

```
void * Base::operator new(size_t size)
{
if(size!=sizeof(Base)) //if size is wrong
return::operator new(size); //have standard operator
                            //new handle the request
                            //otherwise handle the request here
…
}
```

The member version is also simple which is as follows:

```
Class Base
{
public:
static void*operator new(size_t size);
static void operator delete (void *rawMemory,size_t size);
…
};
void  Base::operator delete(void *rawMemory ,size_t size)
{
if(rawMemory == 0) return; //do nothing if the null pointer is being deleted.

if(size!=sizeof(Base))
{
::operator delete(rawMemory);
return;
}
//deallocate the memory pointed to by the rawMemory
return;
}
```

5.4 AVOID HIDING THE NORMAL FORM OF NEW

A declaration of a name in an inner scope hides the same name in outer scopes, so for a function f at both global and class scope, the member functions will hide the global function:

```
void f ();                          //global function
Class X
 {
Public:
void f ();                          //member function
};
X x;
F ();                               //calls global function
x.f();                              //calls X::f
```

This does not causes confusion, because global and member functions are usually invoked using different syntactic forms .However , if an operator new is added taking additional parameters then it is as follows :

```
class X
{
public: void f();
//operator new allowing specification of a new-handling function
static void * operator new(size_t size,new_handler p);
};
void specialErrorHandler();                 //definition is elsewhere
X * px1 = new (specialErrorHandler) X;      //calls X:: operator new
X *px2 = new X;                             //error
```

By declaring the function called “operator new “inside the class, the access to the “normal” flow of new is blocked.

One solution is to write a class specific operator new that supports the “normal” invocation form. If it does the same thing as the global version, that can be efficiently and elegantly encapsulated as an inline function:

```
class X
{
public:
void f ();
static void * operator new (size_t size ,new_handler p);
static void * operator new(size_t size)
{
return::operator new (size);
}
X *px1 = new (specialErrorHandler) X;
//callsX::operator new (size_t,new_handler)

X * px2 =new X;                 //calls X::operator new (size_t)
};
```

An alternative is to provide a default parameter value for each additional parameter one add to the operator new:

```
Class X
{
public:
void f ();
static void * operator new (size_t size, new_handler p=0);
};
X * px1 = new ( specialErrorHandler) X;
X *px2 = new X;
```

Either way if it is needed to customize the behavior of the “normal” form of new , only the function has to be rewritten ;callers will get the customized behavior automatically when the relink.

5.5 OTHER MEMORY MANAGEMENT TECHINQUES [21, 22, 23, 26]

Shared Use of Memory

If some parts of the program work on data of the same type and size, but at strictly disjunctive times, they can all store their data at the same location. Drawbacks are that this is likely to introduce errors if the program is changed, so that the usage intervals start to overlap. Also initialization and/or cleanup of shared areas need to be considered carefully.

C and C++ offer the data type union to allow storing data of different types at the same memory location. This is expansion of the concept of shared use of the same location.
Shared use of the same memory location is potentially dangerous, but with good documentation of the intention and usage of shared memory areas the program should stay maintainable.

Data structure - Loosely associated values

If there is a structured data type with fields that are only used for a small number of allocated objects, those fields can be removed from the object and accessed with a lookup table. The

key for the lookup is usually the objects address. Removing the fields will shrink the objects size, but the tradeoff is a slower access to the off-loaded fields.

Example:

Let's assume, there is a structure like this:

```
struct example_t
{
   char * description;

   /* more fields here */
};
```

And description is used only for a small fraction of all created example_t structures, and then 4 bytes for most of them are being wasted. Those 4 bytes can be saved at the expense of a slower lookup and a lookup table. Overall the savings will only pay off if the lookup table needs less memory than the saved memory (4 bytes * number of example_t without description).

The lookup could be done with a hashtable that maps the object to the description string:

```
static Hashtable  description_table;
void store_description(struct example_t *ex, char * description)
{
   description_table.put(ex, description);
}
char * lookup_description(const struct example_t *ex)
{
   return description_table.get(ex);
}
```

This way 4 bytes can be saved for each allocated example_t. If the sum of those savings outweights the size of the lookup table (i.e. because a huge number of example_t get allocated and only very few have description strings) then overall this construct will reduce the memory consumption of program.

5.6 BUFFER

The buffer is very important consideration while implementing the TPEG Decoder. Since the data from the DAB arrives continuously and also the size of the binary TPEG file is different, therefore proper buffer mechanism is of utmost importance to avoid the loss of incoming data. The buffer stores the incoming TPEG data which in turn is decoded by the decoder. The buffer implemented in the software system was as follows:

Fixed Size Buffer: In this case only a limited number of messages can arrive at the same time to the decoder. It is because the message buffer has a fixed buffer size. The buffer size implemented in the system was 1kb. There were various reason to choose such an simple buffer mechanism:

- **Simple Code:** It yields in a simple code, which can be tested, modified and corrected easily.
- **Ease in Testing:** To test the code it was required to have a buffer which does not have an overflow so the size was chosen at 1kb.
- **TPEG over DAB:** Since the TPEG signals over DAB are in testing phase at present, therefore the possibility of buffer overflow is remote.
- **No effect On Design:** Since the Design of the module remains consistent irrespective of the choice of the buffer mechanism, it would be relatively easy to replace the fixed buffer with a floating self adjusting buffer later on.

BETTER ALTERNATIVE

Variable Self Adjusting Buffer: At the beginning of the decoding process the size of the message buffer is fixed. Assume this size is 1KB (Say for example it can store 200 messages).If at any moment there are more than 200 messages that has to be stored, then the size of the buffer is doubled. As soon as the size of the buffer is doubled there is buffer space to store new messages. If this size turns out to be not enough, the process will be repeated and the new size of the buffer will be of 800.

DISCUSSION OF THE SOLUTIONS

The process followed can be considered as one of the best methods to solve the given problem. Rational Unified Process helps to develop an iteration plan which controls the project lifecycle. Also the RUP prescribes certain important methods and techniques which when incorporated during the software engineering process give a robust design.

Requirement Engineering is of utmost importance which yields a clear vision of the long term expectations of the clients from the software. It also minimizes the risk in the failure of the project as continuous interaction with the client stabilizes the design to a fair extent.

Use Case Analysis leads to a robust, stable software design which is the basis of the implementation. Since the software has to be finally embedded into the navigation system memory issues have to be considered while implementation.

Memory Optimization is quite important and there should be a fair balance with implementation of memory management techniques to reduce the complexity of the software system .Therefore to limit the complexity of the software only important memory management features are incorporated in the code. Robust design and quality code helps future developers to work easily.

In the project, I have synchronized the code with model using the modeling tool Together. This helps to deliver a module which is consistent with the model. Also any change in the model automatically changes the code, thereby easing the process of reengineering. This technique is most efficient in delivering high quality code synchronized with the model view. The client can view the model abstraction of the source code and thereby suggest future improvement easily.

The major disadvantage of the process which was felt during the project was that it’s a time demanding process. But it was strongly felt by the company; that there is higher need to deliver high quality code along with design .This was the tradeoff between time and quality.
Time restriction forced to take certain strategic implementation decision but the consistency of code and model was maintained during the project lifecycle.

CONCLUSIONS

The TPEG Technology is an upcoming technology in the field of Travel and Traffic Information over Digital Audio Broadcasting. With the TPEG in the process of being standardized as an ISO Standard the importance of the thesis problem is evident.

There is future scope in the problem domain for better implementation of the buffer techniques .A better buffer mechanism is suggested already in the thesis. The better alternative suggests having a variable self adjusting buffer instead of the fixed buffer size. When there is a possibility of a buffer overflow then the buffer size increases to avoid the problem.

Further when there is real time signal of TPEG over DAB then it would be possible to test for the buffer overflow conditions. Some unpredictable errors can arise in the new buffer mechanism which might have to be corrected. But it can be said with confidence that such implementation corrections would be trivial. Also there is scope to find out the effectiveness of the memory management in the code.

This component will be used by Robert BOSCH Group in the GST Safety Channel project; Safety Channel is a three-year sub-project within the EC-supported GST (Global System for Telematics) Integrated Project and runs from March 2004 – March 2007. http://gstforum.org/en/subprojects/safety_channel/about_gst_safety_channel/

Applications are under active development across Europe to deliver safety-related information to drivers. Safety Channel aims to define and develop a solution that will ensure the creation and delivery of detailed safety information for all drivers, in all areas across Europe

Robert BOSCH Group is a member of the GST project and therefore this TPEG Decoder software component is a part of the GST project.

REFERENCES

1) Why TPEG? ; The TPEG Project; 2003;
http://www.tpeg.org/pdf/what_is_tpeg/T5_030807BM_D13_Why%20TPEG_6gr.pdf

2) Future TPEG Applications; The TPEG Project; 2003;
http://www.tpeg.org/pdf/standardisation/T5_030804DK_D13_future_3gr.pdf

3) End Users Friendly location Concepts; The TPEG Project; 2003;
http://www.tpeg.org/pdf/what_is_tpeg/T5_030804DK_D13_map%20tool_3gr.pdf

4) JPEG, MPEG and now TPEG why? ; The TPEG Project; 2003;
http://www.tpeg.org/pdf/what_is_tpeg/T5_031030BM_DK_D13_Quick%20guide_10.pdf

5) TPEG –Design Philosophy; The TPEG Project; 2003;
http://www.tpeg.org/pdf/what_is_tpeg/T5_030804BM_D13_Design%20philosophy_6gr.pdf

6) TPEG clients- how they will be used? The TPEG Project; 2003;
http://www.tpeg.org/pdf/what_is_tpeg/T5_030804DK_D13_terminals_3gr.pdf

7) TPEG Location Referencing; The TPEG Project; 2003
http://www.tpeg.org/pdf/what_is_tpeg/T5_030804BM_D13_TPEG%20Loc%20coding%20concept_4gr.pdf

8) How to migrate from code-centric to model-centric development using Rational Software Architect; Aaron Lloyd; http://www-128.ibm.com/developerworks/rational/library/04/r-3247/

9) The Rational Unified Process; Thomas Meloche; President and Fellow Melno Institute; 2003; http://www.menloinstitute.com/freestuff/whitepapers/rup.htm

10) The Rational Unified Process-An Introduction, Third Edition, Phillipe Kruchten, Addison Wesley.

11) Agile Modeling and the Rational Unified Process (RUP); Scott W.Ambler;
http://www.agilemodeling.com/essays/agileModelingRUP.htm

12) Mastering Object-Oriented Analysis and Design with UML, 2003, Rational Software.

13) A Survey of Approaches for Describing and Formalizing Use Cases; Russell R. Hurlbut;
http://www.iit.edu/~rhurlbut/xpt-tr-97-03.html

14) UML 2 Use Case Diagrams; The Object Primer 3rd Edition: Agile Model Driven Development with UML 2.
http://www.agilemodeling.com/artifacts/useCaseDiagram.htm

15) Getting from use cases to code, Part 1: Use-Case Analysis; Gary Evans; http://www-128.ibm.com/developerworks/rational/library/5383.html

16) Project Communication and the Unified Modeling Language; Art Taylor
http://www.informit.com/articles/article.asp?p=31942&seqNum=3&rl=1
17) Designing GUI Applications with Windows Forms**;** Erik Rubin, Ronnie Yates
http://www.informit.com/articles/article.asp?p=101720&seqNum=19
18) Lecture on UML and Use Cases; Dr. Harsh Verma; MIT Ecommerce Architecture Project, 2000 http://ac.mit.edu/ecap-general00/resources/uml.htm
19) The Unified Modeling Language-Reference Manual, James Rumbaugh, Ivar Jacobson, Grady Booch; Addison Wesley.
20) Effective C++: 50 Special Ways to Improve Your Program and Design; 2nd Edition; Scott Meyers
21) Memory Model for Multithreaded C++; Andrei Alexandrescu, Hans Boehm
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1680.pdf
22) C++ Memory Management: From Fear to Triumph, Part 3; George Belotsky; O'Reilly Emerging Telephony Conference, CA
http://www.linuxdevcenter.com/pub/a/linux/2003/08/07/cpp_mm-3.html?page=1
23) Rediscover the Lost Art of Memory Optimization in Your Managed Code; Erik Brown; MSDN Magazine January 2005;
http://msdn.microsoft.com/msdnmag/issues/05/01/MemoryOptimization/default.aspx
24) Introduction to Rational Unified Process; Prof.Kevin Englehart, Department of Electrical and Computer Engineering, University of New Brunswick.
http://www.ee.unb.ca/kengleha/courses/CMPE3213/OOAD/02UnifiedProcess_ie4_files/frame.htm
25) C++ Memory and Resource Management; Stephen Dewhurst;
http://www.informit.com/articles/article.asp?p=30642&seqNum=2
26) Code Optimization; Hansjorg Malthaner;
http://library.simugraph.com/articles/opti/optimizing.html
27) .NET Code Optimization; Richard Grimes; Net Newsletter; Jan 2003
http://www.windevnet.com/documents/s=7446/win1055785679994/
28) Managed C++: Read and Write Registry Keys and Values; Tom Archer, Archer Consulting Group; December 2004
http://www.codeguru.com/Cpp/W-P/ce/registry/article.php/c8859/
29) The C++ Standard Library – A Tutorial and Reference, Nicolai M.Josuttis; Addison Wesley