

**AControllerTest**  
**A Testing Framework for AController**

Afzal Rehman Khaskheli

University of Tampere  
Department of Computer Sciences  
Computer Science  
M.Sc. thesis  
Supervisor: Jyrki Nummenmaa  
December 2008

University of Tampere

Department of Computer Sciences

Computer Science

Afzal Rehman Khaskheli: AControllerTest: A Testing Framework for AController

M.Sc. thesis, 44 pages, 5 index pages

December 2008

---

This thesis relates to the software testing in mobile phones for the audio module. Audio in mobile phones is a major concern of success. Audio processing in mobile phones involves many hardware and software modules and therefore needs careful testing at various stages of the development life cycle. This work is about enhancing the testing of the AController software module that interacts with the domestic operating system and various audio channels, such as music player and phone call. The interaction of the AController with other modules forms complex testing requirements. To perform testing at the module level of the AController, a new software module called as the AControllerTest is developed. The architecture of the AControllerTest module is explained with the help of package, class and sequence diagrams. The principles and theory taken into consideration during design and implementation phases are presented. The merits and demerits of the AControllerTest are highlighted. The working of the AControllerTest and its interaction with other modules is shown by giving comparative execution tables of the manual and the AControllerTest executions. The results are presented with respect to the change in the testing process and change in the development life cycle of the AControllerTest.

Key words and terms: Mobile devices, audio testing, module testing, AController, AControllerTest.

## **Acknowledgements**

I would like to record my gratitude to my thesis supervisor Jyrki Nummenmaa, for his supervision, guidance and advise. His kind support and continuous patience led the successful completion of the work. His vast experience always ushered a spirit to produce a work of the highest quality.

I owe the heartiest acknowledgements to Rasmus Rahunen, the mentor of the work. This work is his brain child. His logical, scientific, critical, encouraging, cordial and positive approach was instrumental in the fulfillment of the thesis goal. I would not exaggerate in saying that the success of this work is due to his continuous counseling.

I would like to pay special thanks to Mika P. Kangas for his insightful work judgment. Whenever, there was a query regarding the work, he was there to respond and came up with suggestions. It is due to his credentials that the work is being published.

I thank to all friends and colleagues, who were directly or indirectly, supportive of the work and always kept a watchful eye on the progress. I am indeed grateful to all of them. Thank you.

## Contents

|  |    |
|--|----|
| 1. Introduction .....  | 1  |
| 1.1 Software testing.....                                    | 2  |
| 1.2 Audio in mobile phones .....                             | 3  |
| 1.2.1 Digital audio and sound .....                          | 3  |
| 1.2.2 Conversion of sound to digital audio.....              | 4  |
| 1.2.3 Audio signal processing.....                           | 5  |
| 1.3 Testing in mobile phones.....                            | 5  |
| 1.4 Thesis contributions .....                               | 9  |
| 2. The AController module .....                              | 10 |
| 2.1 Overview of the AController .....                        | 10 |
| 2.2 Multimedia framework.....                                | 10 |
| 2.3 Design of the AController .....                          | 13 |
| 2.4 Need of testing the AController .....                    | 15 |
| 3. The AControllerTest module.....                           | 17 |
| 3.1 Overview of the AControllerTest.....                     | 17 |
| 3.2 Test framework explained .....                           | 20 |
| 3.2.1 Life cycle of the AControllerTest .....                | 20 |
| 3.2.2 Module testing with the AControllerTest .....          | 22 |
| 3.2.3 Specifications for the AControllerTest.....            | 23 |
| 3.2.4 Design and implementation of the AControllerTest ..... | 24 |
| 3.3 Software tools used in the development.....              | 28 |
| 3.3.1 Symbian (SDK) .....                                    | 28 |
| 3.3.2 Carbide.c++.....                                       | 28 |
| 3.4 Pros and cons .....                                      | 29 |
| 4. Testing with the AControllerTest .....                    | 30 |
| 4.1 Tools and problems .....                                 | 30 |
| 4.2 Test case execution.....                                 | 30 |
| 4.3 Results of test case executions.....                     | 38 |
| 5. Conclusions .....   | 40 |
| References .....   | 42 |

## Table of figures

|   |    |
|---|----|
| Figure 1- ADC Converter.....  | 4  |
| Figure 2- Symbian MMF.....  | 11 |
| Figure 3- AController's interaction with other components .....             | 17 |
| Figure 4- AController's interaction with other components using stubs ..... | 18 |
| Figure 5- Incremental model phases .....                                    | 21 |
| Figure 6- The class diagram showing the basic test framework.....           | 24 |
| Figure 7- The class diagram showing test case instantiation .....           | 25 |
| Figure 8- The sequence diagram of a typical test case execution.....        | 27 |

## Table of tables

|  |    |
|--|----|
| Table 1 – Example audio function.....  | 7  |
| Table 2 - Audio transfer from mono Bt headset to IHF during MO CS call ..... | 31 |
| Table 3 - Audio transfer from IHF to USB headset.....                        | 33 |
| Table 4 - Playing ringing tone with wired accessory.....                     | 34 |
| Table 5 - Digital radio recording case.....                                  | 35 |
| Table 6 - Video call.....  | 36 |

## Glossary

|                 |   |
|-----------------|---|
| AController     | A software module in the larger audio sub-system  |
| AControllerTest | A software module for testing the AController   |
| ADC             | Analog-to-digital converter   |
| AMapper         | A software module that maps changed hardware attributes to messages and sends them to the domestic OS                       |
| ARouter         | A software module that routes the audio to a proper channel or device   |
| AServer         | A software module that services higher level client modules and forwards the requests to AController for further processing |
| AStreamer       | A software module that provides APIs for streams  |
| BT              | Bluetooth   |
| Carbide         | GUI based development environment for Symbian   |
| CS              | Circuit switched  |
| DAC             | Digital-to-analog converter   |
| DL              | Downlink  |
| DSP             | Digital signal processor  |
| Emulator        | PC based software that mimics the mobile phone  |
| IHF             | Integrated hands free   |
| MMF             | Multimedia Framework- a repository of multimedia plug-ins   |
| MO              | Mobile originated   |
| MT              | Mobile terminated   |
| SMS             | Simple messaging service  |
| Stub            | Code hiding and replacing mechanism for testing purposes  |
| Symbian         | Common OS for mobile devices  |
| UL              | Uplink  |

## 1. Introduction

Audio is the feature of mobile phones which is required in almost all applications. For example, audio function is needed when a call is made or when music is played. Providing a good audio functionality is the core of mobile services. Apart from audio function, many other services add to the value of mobile phones. Based on the services and functions provided by mobile phones, the mobile industry is broadly divided into three generations of technology. First generation (1G) systems were analog and circuit-switched devices, able to send limited amount of data e.g. SMS. They had very slow processing speed, small storage size and provided lesser security to the data being sent as compared to following generations of mobile phones. Second generation (2G) used radio channels for voice and control channels, digital encoding and included GSM, D-AMPS (TDMA) and CDMA. In addition, 2G offered services like fax and SMS because of its better speed and improved security compared to 1G. The 2.5G which extends 2G systems to provide additional features such as packet-switched connection (GPRS) and enhanced data rates (HSCSD, EDGE). Finally, third generation (3G) technology provides higher data transfer speeds than previous generation and is capable of providing a rich set of multimedia applications other than audio [Glossary; Nurvitadhi, 2003]. Current mobile phones could better be termed as mobile computing devices. With the advent of 3G technology, it is now possible to support services such as audio and video conferencing, music download and image transfers. As voice signals are converted into digital data, the speech is treated as any other data in 3G systems. Third Generation systems use packet-switching technology, which is more efficient and faster than the traditional circuit-switched systems. It is possible to carry out multiple tasks at the same time, for example, playing music in a music player and downloading data from the Internet at the same time. Fourth generation (4G) development work has started. 4G will be able to provide even higher speeds, more storage, more reliable communication and higher security compared to its predecessors.

Some of the common present day mobile phone features are listed here:

- Music player – the player has the capability to play various formats of songs.
- Bluetooth – a short-range wireless technology used today for connecting and transferring information (e.g. recorded clip) between phone and other related devices.
- Video call – a feature that allows face-to-face conversations using the camera built into the phone and voice calling service.

- Integrated cam – a built in camera with ever increasing pixel resolution, having the capability to record audio and video synchronously.
- Infra-red (IR) – it helps to transfer data (e.g. songs) with a very small distance between the mobile phone and its relevant devices.
- FM radio – a built in radio that allows the users listen to music [Bhawani].

All these features of mobile phones have audio as the common function and they require rigorous software testing from the start of the development life cycle.

### **1.1 Software testing**

Software Testing is the process of executing a program or system with the intent of finding errors. That is to say, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results [Pan, 1999].

There are two types of software testing [Pirozzi]:

1. Manual software testing is the process in which a person carries out the task and no automatic tool is involved.
2. Automated software testing is the process of creating and using test scripts that can then be run automatically, repetitively, and through much iteration.

The selection of the type of testing is dependent on the phase of development life cycle. During the code implementation phase automatic testing could be performed. The technique used to conduct automatic tests is called the white-box testing and it is a verification technique that examines if the code works as expected. It takes into account the internal mechanism of a system or module and is run with predetermined input and it makes sure that the code produces predetermined outputs. Often programmers write stubs and drivers for white-box testing [White box]. There are various white box testing techniques used at the module level. These testing techniques are broadly divided into three types [Rajendran]:

- Functional Testing, which is based on followings:
  - Boundary value analysis: the edge conditions of boundaries are tested.
  - Equivalence partitioning: test cases are grouped into classes such that executing one test case is equivalent to executing any other test case in the same group



- Cause effect graphing: this checks if something is caused by some other factor, i.e., cause and effect.
- Structural Testing, which is based on followings:
  - Statement coverage: makes sure that every line of code is executed at least once in one of the test cases.
  - Branch coverage: makes sure that every branch of code is executed at least once in one of the test cases.
  - Condition coverage: makes sure that the condition in each predicate expression is evaluated in all possible ways.
  - Modified condition-decision coverage: makes sure that each Boolean operand can independently affect the outcome of a decision.
- Heuristic or Intuitive Testing, which could use any of the above techniques.

Rajendran [Rajendran] further classifies the types of defects found in the software as omissions, surprises and wrong implementations. Omissions are missing from the requirements in the implementation. Surprises are the implementations that are not found in the requirements. Wrong implementation refers to the incorrect implementation of a requirement. Functional testing techniques are devised to find omissions and wrong implementations. Structural testing techniques are devised to find surprises and wrong implementations. Heuristic or intuitive testing techniques try to find all types of defects.

This study aims to perform software testing at the module level for audio functions in mobile phones.

## **1.2 Audio in mobile phones**

This section explains the processing and complexity involved in audio functions provided by mobile phones.

### **1.2.1 Digital audio and sound**

Technically speaking, sound is vibration or wave of air molecules caused by the motion of an object. Sound is produced by small areas of high and low pressure of objects vibrating in various directions from the source. These objects are molecules and their movement forms waves. When sound waves hit a hard surface they bounce back and produce echo. Waves fall into the ear and then the corresponding signals travel to the brain. Since the sound waves are moving therefore they contain energy. Sound energy is

convertible to different forms such as electricity and vice versa. This conversion mechanism allows us to communicate over mobile phones. Sound waves vibrate at different rates as they move through a medium. This different rate of the movement is called the frequency measured in cycles per second, or Hertz. If an object has a higher frequency then it produces a higher of the sound also [Sound].

### 1.2.2 Conversion of sound to digital audio

Real world sound is analog in nature. It is possible to convert sound initially into electrical signals using a transducer (e.g. microphone) and then electrical signals are converted into digital signals using a circuit called ADC (Analog-to-Digital Converter). Later the digital signals could be manipulated by other digital equipment. Similarly, it is possible to convert digital signals back to the sound. Initially an analog electrical signal using a circuit called DAC (Digital-to-Analog Converter) is obtained, which then is processed by a transducer (e.g. loudspeaker) to produce sound. Mobile phones also work on the same mechanism. The voice is converted into digital by some switch or line like ISDN or DSL. [Ravindran et al., 2005] has presented the block diagram of a typical ADC:

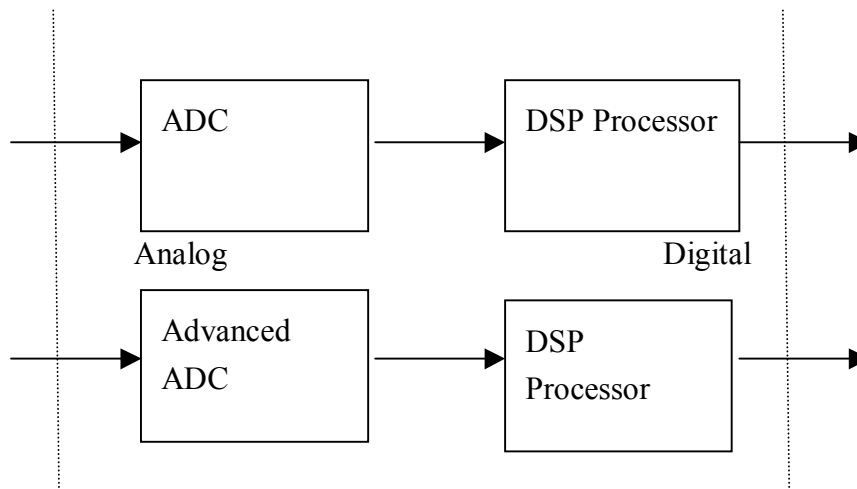


Figure 1- ADC Converter

An ADC circuit takes samples from the analog signal at different time intervals. Each sample is converted into a number, based on its voltage level. The frequency with which

the samples are taken is called the sampling rate. If in one second 30000 points are sampled, then it means that the sampling rate is 30000. Thus, each sample will be taken at 1/30000 second i.e. 33.33 micro seconds.

DAC takes number of samples from the digital signal based on the voltage level. But this conversion of the digital signal back to an analog signal will not produce the original signal as it does not have all the points of the original signal. DAC will try to map the missing points [Torres].

Therefore, the more sampling points are used, the more perfect will be the analog signal produced by the DAC at the cost more storage space for storing the samples.

### **1.2.3 Audio signal processing**

The digitized signal by ADC is processed with digital signal processor (DSP). The DSP is the core component in devices such as mobile phones, modems, multimedia PCs. One function of the DSP in mobile phones is to compress the digitized signal to decrease the required bandwidth and increase the storage space. This way different actions could be handled at the same time, which include, for example, talking and seeing each other's picture using the video. Signals in audio signal processing are speech or music signals, which use only part of the human's auditory area. Signal compression is done with various compression algorithms such as MP3 and AAC encoding. This compressed data needs to be decompressed for using it and it is done by the decoder in the DSP. The DSP processor includes components for handling other audio signal handling tasks, such as multiplexing and audio effects. Then audio enhancement is done to amplify and/or clarify some inaudible or poorly recorded audio by suppressing background noise or other sounds that interfere with the ability to hear the desired sounds. Different enhancement algorithms are needed to cater for mobile phones usage in different kinds of environments ranging from silent rooms to very noisy streets [Märsynaho, 2006].

As audio is such an important feature of mobile phones, it is important to verify audio quality during the development process. Assuring high quality of audio in mobile phones requires careful software testing process.

## **1.3 Testing in mobile phones**

Mobile phone testing has become complex with the continued addition of features in general. The efforts are being put in to reduce test time and cost. It is a well known fact that the testing never finds all errors in the software rather it makes sure that the product is stable. Therefore, mobile device testing could be defined as the process to assure the

quality of mobile devices, such as mobile phones and PDAs. The testing is carried out at both hardware and software levels. Present day mobile phones are not merely mobile phones rather they are mobile multimedia devices and they add to the complexity of testing audio functions. Bo et al. [2007] has described the difficulties in testing mobile devices:

- Complex environments - as they have to interact with end users, wireless signals and other devices in a context-sensitive way.
- Diversity - reduces the reusability and maintainability of test cases.
- Highly resourceful - constrains the processing ability, memory capacity and communication ability.
- Highly interactive - devices constantly accept activations from the users and send responses back for user to take further actions.

These mobile devices provide various audio types including phone audio, music audio, ringing tone audio, keypad audio. It is estimated that over 400 different audio implementations on mobile devices are available. All types of audio need to be tested with the given hardware and software [Mobile Audio, 2007].

As there are many types of audio in the mobile phones, there are many combinations of audio functionalities to test. Märsynaho [2006] has listed some use cases as shown in Table 1. The functionalities shown in the table vary from product to product due to the fact that there may be different usage of the operating platform, a different set of hardware components may be used. Furthermore, different combinations of the features given in the mobile phones makes the testing of the audio use cases more complex.

Table 1 – Example audio function

| <b>Incoming function</b><br>Vs<br><b>Ongoing function</b> | Incoming call with digital ringing tone  | Calendar alarm activated and proper alarm tone selected                                 | Alarm clock activated   | Incoming SMS  |
|---|--|---|---|---|
| Call ongoing  | Only beep sound is heard from near end's earpiece, if waiting call service is activated. If first call is ended, ringing tone is heard.                | Only beep sound is heard from near end's earpiece and notification is shown on display. | Only beep sound is heard from near end's earpiece and notification is shown on display. | Only beep sound is heard from near end's earpiece.  |
| MP3 playback. Headset in use.                             | Playback is not heard (volume is ramped down) during the call alarm, which is heard from both headset and IHF. If call is answered playback is paused. | Playback is paused and calendar alarm is heard from both headset and IHF.               | Playback is paused and clock alarm is heard from both headset and IHF.                  | Message ringing tone is not played. Beep sound is mixed to MP3 playback and is heard from both headset and IHF. |

|  |   |   |  |   |
|--|---|---|--|---|
| Silent profile activated   | No ringing tone is heard. Only notification is shown. Phone is vibrating if vibration is activated. | Calendar alarm is not heard, only notification is shown on display.             | Clock alarm alerts normally.   | No message ringing tone is heard, only notification is shown on display.  |
| Bluetooth in use. General profile activated.   | Ringing tone is heard from both BT and IHF.   | Calendar alarm is heard from both BT and IHF.                                   | Clock alarm is heard from both BT and IHF.                                   | Message ringing tone is heard from both BT and IHF.   |
| Radio playback, assuming that used phone has the radio functionality. Headset in use. General profile activated. | Radio playback is muted and incoming call starts to alarm and is heard from both headset and IHF.   | Radio playback is paused and calendar alarm is heard from both headset and IHF. | Radio playback is paused and clock alarm is heard from both headset and IHF. | Radio playback is muted and message ringing tone is heard from both headset and IHF.                              |
| Video recording, assuming that used phone has the video record functionality and camera.                         | Video recording is stopped and file saved, when incoming call starts to alarm.                      | Video recording is stopped and file saved, when calendar alarm starts.          | Video recording is stopped and file saved, when clock alarm starts.          | No message ringing tone is heard, only notification is shown in display. Video recording continues uninterrupted. |

Internally, various Symbian based components/modules form the audio handling framework in mobile phones using the Symbian phones. These software modules hide the internal details of the hardware and allow for the mixing of the audio from various components. All these software components need an interaction with various hardware components. This interaction is controlled by a component called the AController. The

AController sits in between the hardware and software modules and works with various streams.

The AController is composed of different sub-components, for example, a volume controller, a filter controller, an input router, an output router. Each sub-component is responsible for performing different tasks. As the AController interacts with many software and hardware streams, there are many routes and different sub-components involved in the process.

The audio processing involving AController requires testing at various stages of the software development life cycle. At the module level, automatic testing is carried out. However, at present, testing at the module level of AController does not cover all the expected possibilities i.e. testing the execution of the code in all the possible routes and the involvement of different sub-systems in the audio processing. Therefore, there is a great need to enhance the present audio testing mechanisms.

#### **1.4 Thesis contributions**

This thesis work aims to enhance the testing of the AController module by developing a software component at the module level. In this regard, a framework is developed such that more routes and sub-components of the AController and their interaction with various streams could be covered. The framework stubs the modules which interact with the AController so that testing could be performed during the development without caring about the functionality of other modules. The stubbed modules in the framework are implemented and one such module is the AControllerTest. The AControllerTest implements software test cases which are based on the IEEE specifications. Test cases represent the actual use cases and cover various audio routes and sub-components in combinations. In addition, test case executions improve the quality of the product from the start of the software development life cycle. The development is performed in the Symbian platform.

Chapter 2 of this study explains the AController module and the reasons to test it at the module level. The theoretical test framework for AController along with its possible advantages and disadvantages is given in Chapter 3. The development and working of the test framework is explained in Chapter 4. Finally, in Chapter 5, conclusions are drawn from the experiment and reference is made to possible future work.

## **2. The AController module**

This chapter gives an overview and a higher level design of the AController. In addition, it is explained that why there is a need to test the AController at the module level.

### **2.1 Overview of the AController**

The AController forms a subsystem of a larger audio system. It interacts with various modules in the system. Mainly it interacts with two other modules that are parts of the audio system and these are AMapper and AServer. The AMapper module is responsible for interacting with the domestic operating system for transforming physical stream attributes to hardware configuration settings. Once the logical streams are created by AServer they are mapped on to the physical streams created by the AStreamer. The logical streams are created in the AServer using the AStreamer module of the AController subsystem. After creating the logical streams, the AServer sends a request to the ARouter subsystem of the AController to provide a path to correct physical stream. The ARouter subsystem then sends the request sent by the AServer subsystem to the AMapper subsystem which in turn forwards the appropriate request to the domestic operating system.

The AController is there because of the modular design approach. The modular design organizes a complex system as a set of different components which are developed independently and build together. The AController provides various advantages of modular design which include [Stotts, 1982]:

- communication between different components is less complex,
- information is hidden from other modules
- access to the contents is restricted in a well-defined way, and
- a better way of isolating the hardware dependent sections of code.

The AController is implemented by modifying the functionality of the Multimedia Framework (MMF).

### **2.2 Multimedia framework**

The MMF is a repository for multimedia plug-ins and it offers a generic interface for the underlying hardware. It has its own controller which is responsible for managing selection, loading, and interaction with the multimedia plug-ins. The MMF is multithreaded component of the Symbian OS. It creates two separate threads for each plug-in and both threads are related to the client's process. The MMF works on a client-server mechanism and provides APIs for both sides. The MMF may consist of many proxy server objects due to the fact that a new proxy server object is created in every new



thread for each controller plug-in by the controller framework. The kernel then controls the communication between the client and the server threads. The MMF is composed of various layers and shown in Figure 2.

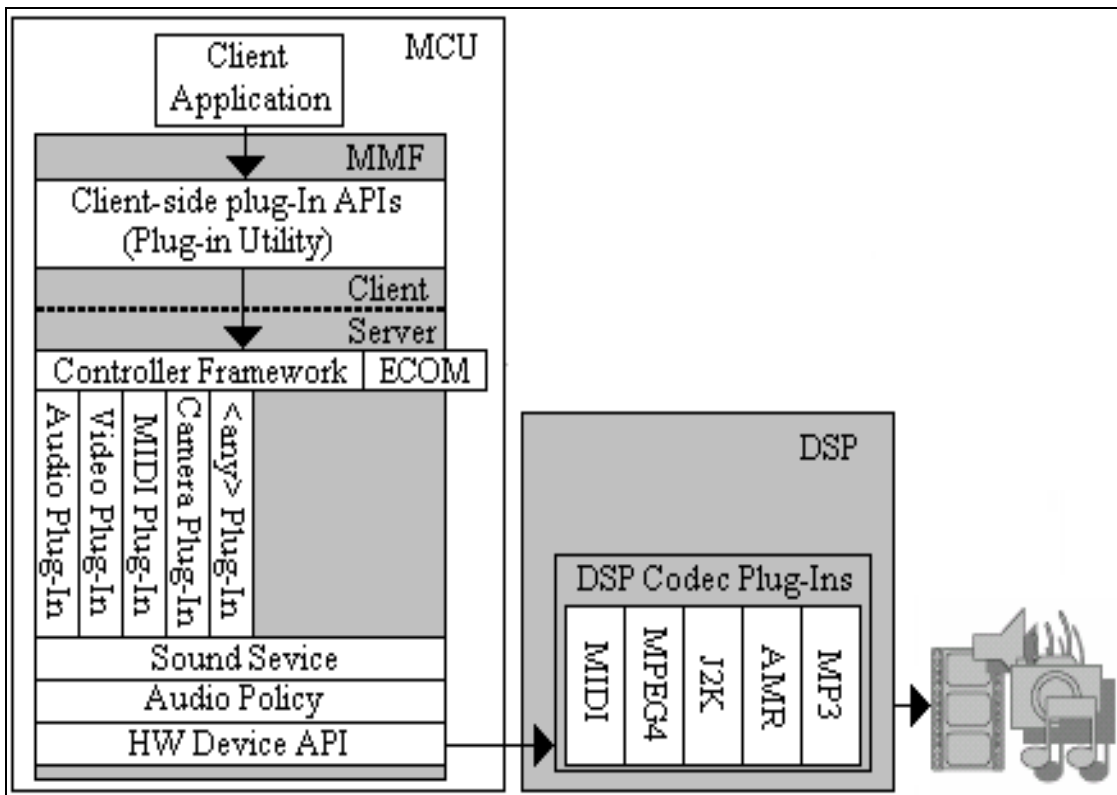


Figure 2- Symbian MMF

These layers of the MMF are briefly explained below:

- Client-side plug-in APIs available for applications to use plug-ins by implementing utility objects of the plug-in. The available features of the MMF API include [MMF Client]:
  - Audio playing, recording and conversion - an interface consisting of various classes provides methods to create, play and manipulate audio data stored in files, descriptors and URLs.
  - Audio streaming - an interface consisting of various classes provides methods for recording and playing audio streams, for example, audio from a web address.

- Tone playing - an interface consisting of a single class provides methods for playing and configuring single and sequenced tones as well as DTMF (Dual Tone Multi-Frequency) strings.
  - Video playing and recording - an interface consisting of various classes provides methods to create, play and manipulate video clips with or without audio tracks stored in files, descriptors and URLs.
- The controller framework is used to resolve selection and launching of plug-ins and also provides means for message passing between applications and the plug-ins. The controller plug-ins are in charge of data processing and control the flow of data between data sources and data sinks. For example, an audio controller plug-in might take data from a file source and output it to a speaker sink, or take data from a microphone source and save it to a file sink. A controller plug-in typically supports one or more multimedia formats, for example WAV or MPEG4 [MMF Controller].
- The sound device (DevSound) provides a common interface to the audio hardware via hardware device interface. This includes following audio functions [MMF DevSound]:
- Initialization and configuration of hardware devices, for example, setting microphone gain, setting stereo balance and so on.
  - The playing and recording of raw audio data.
  - The playing and dynamic control of tones with user specified frequencies.
  - The playing of DTMF strings.
- Audio policy is the decision making unit and it grants access to clients of the sound device based on the priority of requests. Audio policy rules are different for different products. The policy rules may depend on the product behavior, capability of the hardware and availability of system resources. The audio priority of the telephony application is always set to a higher priority than any other multimedia application. It means that the audio policy component must constantly monitor the call status and allow a ring tone to be played during an incoming call, even if it means preempting an ongoing audio playback. The audio policy also monitors changes of the profile settings and update attributes accordingly. The audio policy also keeps track of special audio preferences signed to certain applications, for example, allowing a clock alarm ring even when the phone is in a silent mode [MMF Client].

- The hardware device API is needed to interface the low-level hardware devices which also include optional hardware-accelerated codecs which are resident on the DSP. It abstracts the device's hardware components, such as speaker, microphone, display, or a secondary processor used on the device [Multimedia Framework, 2005].

The above layers of the MFF are modified to fit the needs of the AController. The design of the AController is explained in the next section

### **2.3 Design of the AController**

The AController implements logics for the lower-layer software between the Symbian MMF and the domestic operating system. The AController subsystem is mainly composed of two components namely AStreamer and ARouter. The AController's working is based on the concept of streams. An audio stream is a continuous flow of audio data that has no clearly defined beginning or end. The application can relay audio packets into the stream and these streams are processed as soon as they are received by the underlying subsystem. This differs from clip based methodology in which the entire audio data set has to be received before any audio processing or playback can occur [Johnson, 2004].

It should be noted that audio sources relate to devices which generate the audio packets and audio sinks relate to devices which accept these audio packets. Streams are responsible for communication between audio sources and sinks. Furthermore, an audio device refers to a hardware which can produce or consume audio packets and the system that holds information about audio devices is termed as the audio device repository.

The AServer uses the AStreamer component of the AController to create new streams. The AServer sets the properties of newly created streams and then sends them to ARouter for further processing. The ARouter deduces hardware attributes from active set of streams and sends these attributes to the AMapper for further actions.

Since the AController contains two additional modules along with its main functions, i.e., the AStreamer and the ARouter, these modules are further explained here.

The AStreamer subsystem works with streams and contains common information about physical and logical streams. In addition, it decides if the stream is logical or physical, and stream type and stream state are recorded to the stream attributes. Information about routing is recorded to stream source devices and sink devices. Streams can be in various statuses which are stopped, active and paused. Hence the system has the capability of working with different statuses.

The logical streams may have various attributes and various audio modes. The AStreamer system provides controlling mechanisms for the collection of logical audio streams and also an iterator for iterating the logical streams.

On the other hand, the physical streams mechanisms contain information related to state and volume of a physical stream. The physical streams represent data routes on the domestic operating system side. The physical streams can have various states like the logical streams. The possible physical stream data sources are generic playback, speech playback, generic recording, speech recording, tone, FM radio, CS call uplink and CS call downlink.

The AStreamer module also contains stream managing component. This component returns physical stream to the AController according to the physical stream type. For the logical streams, a collection of streams is maintained by the system.

In addition, the AStreamer module contains volume functions. Both the logical and the physical streams have the volume manipulation functions. Hence, the ARouter subsystem's functionality is multi-fold.

The ARouter module provides functionalities for various accessory types. It provides methods to connect, disconnect the audio accessory with the given audio parameters. An audio accessory can be in various states and the current status of the devices could also be queried. It is possible to change the present status of the accessory. The domestic operating system is also notified of any change in the accessory type or its status. The system also provides the functionality to deal with some specific accessory types.

The ARouter module implements the audio routing methods. These methods are responsible for opening and closing audio paths, controlling volume levels and audio routing. The system notifies of any change it detects in the streams and accessories, e.g. a volume change and then applies the HW settings accordingly. Apart from this, the ARouter:

- configures several audio filters in the hardware.
- provides a function to get the information about the device being queried based on the ID.
- allows a new property to be subscribed for a device in the central repository.
- includes functions to control physical audio streams. These are asynchronous functions to activate and stop physical streams.
- provides the possibility to update the radio hardware using radio hardware interface. Depending on analog or digital mode of the radio, different paths could be also selected.

- allows the controlling of the playback of tones and tone sequences. Methods are available to play single frequency or double frequency tones as well. The DTMF tones playing functionality is also available.
- implements mechanisms to control the volume levels of physical playback streams. The volume levels are defined in terms of pre-defined minimum and maximum values.

The AController module provides a generalized form of communication amongst various components in the audio system. The AController is used by various software and hardware platforms and the AController provides a common communications interface.

## **2.4 Need of testing the AController**

Mobile phones provide many options for creating audio applications. This multitude of audio applications gives users an option to choose from but it complicates the scenario for the developers and the testers of the mobile devices in that audio hardware of a device will receive simultaneous requests from multiple applications (clients). Consider for example, when a user receives a phone call while using a multimedia application such as music player. The device then decides whether to keep playing the media or stop it and play the phone ringing tone instead. To resolve such situations audio priorities and audio preferences are used. The audio policy component manages access priority to audio hardware on a device by resolving simultaneous requests from different clients. The AController module implements a part of the MMF API to provide multimedia services to applications. The MMF acts as a repository for plug-in multimedia processing units, and serves as the generic interface to a device's hardware [Music, 2006].

Testing in audio software development could be performed at various levels (DevSound, MMF and S60). The UI tests represent real use cases and are performed above the S60 level which is visible to the end user [Märsynaho, 2006]. To test the application during the development process, module/unit testing is performed.

According to Barriocanal et al. [2002] in unit testing:

- Testing is done during the process of the development.
- All the tests are run repeatedly whenever there is new software build available.
- Test cases are considered as code components and are delivered with the program code itself.

The AController's testing has been carried out at the end user testing level. The developers were able to know of some problem in the AController for some product with different releases only when the end user testing was performed. This approach always

produced setbacks for developers as they need to dig into the code at a very late stage of the development life cycle. In addition, at the end user level, the same set cannot be applied to different products because of different architectures and handling of the audio. Therefore, to avoid errors from the beginning of the development life cycle, speedup the process, enhance the testing capabilities, increase the test set, and make it easier for developers, a new software test module is proposed. The new test module is called as the AControllerTest. The module itself forms a part of the main component and will always be available with every new software build test release.

### 3. The AControllerTest module

This chapter is about the newly developed testing module which is called the AControllerTest. At first, the interaction of the AController module with other modules is discussed. Secondly, the interaction of the AController module with other modules including the AControllerTest is explained. Thirdly, the principles of design and implementation of the AControllerTest are discussed in depth. Fourthly, a brief explanation about the chosen development tools and environment is given. Lastly, the pros and cons of the AControllerTest are given.

#### 3.1 Overview of the AControllerTest

The AController is a sub-component of a larger audio system consisting of many sub-components. Its role is to identify and give the right path to the requested audio activity. At a general level, it interacts with other components as shown in Figure 3.

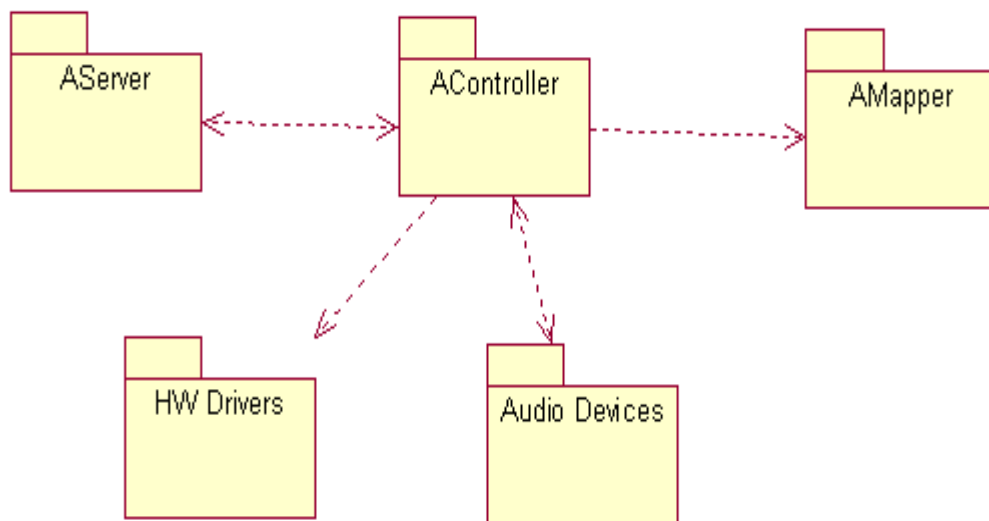


Figure 3- AController's interaction with other components

The AController consists of two sub-systems: ARouter and AStreamer. The ARouter provides an API for the AServer to communicate with it and change routing settings. Similarly, the AStreamer also provides an API to the AServer for changing stream related properties. The AServer component uses the API provided by AStreamer for creating new streams and setting stream specific properties (e.g., volume). The communication between the AServer and the AController component is bi-directional. The AServer sends requests to the AController for activating some settings. The AController then further decides which audio devices or hardware drivers to use for the received request. After deciding the audio device or the hardware driver, the AController sends a request to the AMapper to physically activate the relevant services on the domestic operating system.

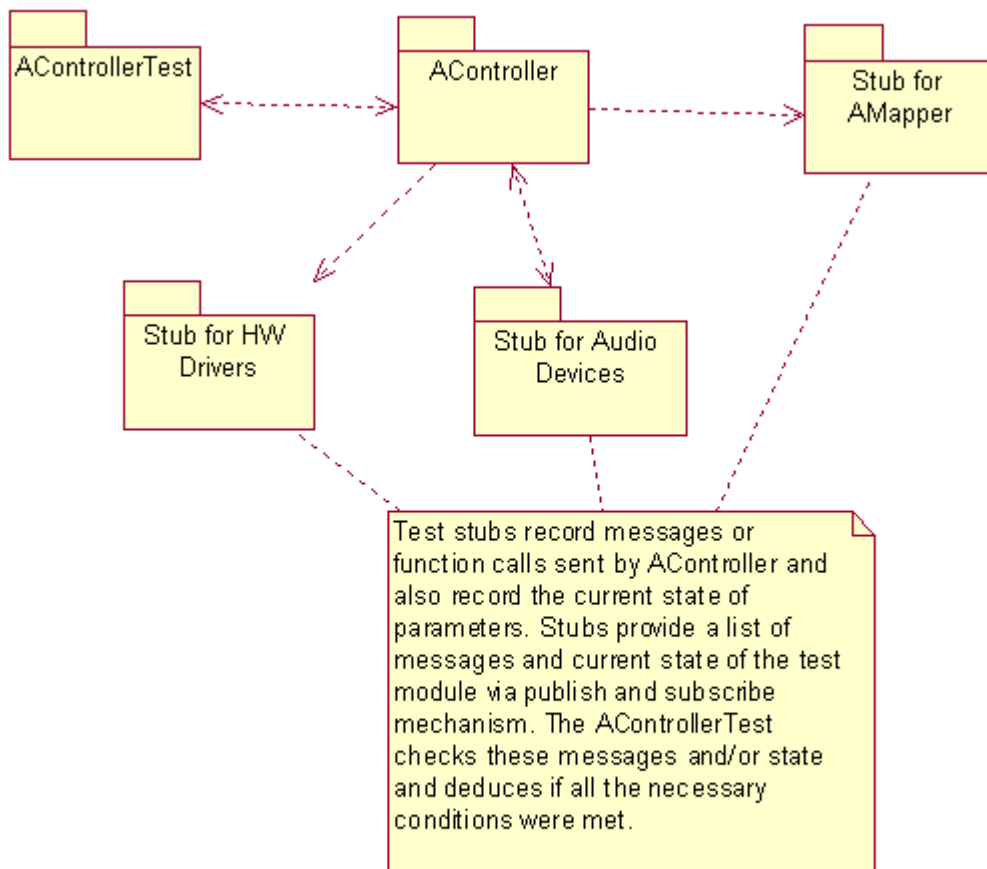


Figure 4- AController's interaction with other components using stubs



Furthermore, the modules which interact with the AController (AServer, Audio Devices, HWDrivers and AMapper) are stubbed in the new testing module (see Figure 4). Stubs are used when a program has many components and many of the components need to communicate with each other for the sake of testing purposes by hiding the real functionality of a component. These components may be placed remotely. Stubs substitute the real code and allow the calling or interacting program to interact with it as if it is doing with the real component. This way the development is made easier as the development of modules is independent of other interacting modules. Stubs can help in the testing of a large system. A stub may contain certain methods or complete sub-systems. In this manner, the higher layer of code could be tested with ease. The obvious benefit of stubs is to quickly implement different parts of a program without much thought of other components and concentrating on the part of the system that is under development. If a sub-system is developed in parallel with other sub-systems, it is always a better idea to stub the other systems and employ module/unit tests for the own system independently of others [Parrington]. A similar concept is applied to the development of the AControllerTest.

In this connection, the AMapper stub is responsible for publishing the status of the services to publish and subscribe component. The TestModule component (a module of the AControllerTest) is registered for checking test step results to the publish and subscribe component as a subscriber and is therefore notified of the success or failure of the request sent by it. Similarly, the AController module communicates with the Audio Device stubs, HW Driver stubs and AMapper stub modules for acquiring the functions that it needs from them.

It must also be noted that the stubbed AMapper component publishes its events so that the AControllerTest is able to know about the success or failure of its request to the system. Publishing and subscribing is a Symbian's way for threads to communicate with each other. It allows the setting, retrieving and monitoring of system-wide variables and provides a new inter process communication mechanism for peer to peer communication among threads. Neither the publishers know of the subscribers nor do the subscribers know about the publishers. In Symbian, the system-wide variables are usually known as 'properties'. This mechanism has three components [Shackman, 2005]:

- Properties – the system-wide variables. Properties contain two attributes: identity and type. A publisher and a subscriber share this information between them.

- Publishers – Threads that send messages. A property is published when its value is updated, using the Set() function. All the subscribers are notified of the change in the property.
- Subscribers – threads that receive messages. To know about some piece of information change in the property, a thread makes an asynchronous request. This is done by calling Subscribe() method. The caller needs to re-subscribe to get more notifications and to retrieve the property's value if required.

The details of the framework are given in the following section.

## **3.2 Test framework explained**

In the proposed framework, the module that replaces and automates the functions of the AServer module is termed as the AControllerTest. As the AControllerTest is responsible for starting the execution of test cases, the complete framework is therefore referred by it. In this section, the life cycle, module testing, specifications, design and implementation of AControllerTest are presented.

### **3.2.1 Life cycle of the AControllerTest**

The software development cycle of the AControllerTest is incremental and thus the incremental software development model is used. A software development life cycle model shows how software is or should be developed. It describes phases of the software cycle and the order in which those phases are executed. There are many such models. In all such models, each phase produces deliverables required by the next phase in the life cycle. Requirements are translated into design. Code is produced during implementation that is driven by the design. Testing verifies the deliverable of the implementation phase against requirements. Based on the software development model, the testing strategies also vary.

The AControllerTest's incremental model in principle is a waterfall model done in repetitions, i.e., development cycles are repeated many times during the development. Each cycle of the incremental model (i.e. waterfall model) is linear and sequential life cycle and is very easy to use and understand. Furthermore, in a waterfall model, each phase must be completed completely before the start of the next phase and each phase is reviewed at its completion to determine whether the project is on the right path and whether or not to continue or discard the project. Hence, the phases do not overlap each other in a waterfall model. Coming back to the incremental model, the project is divided into smaller cycles, so that it could be managed more easily. Each

iteration passes through the requirements, design, implementation and testing phases. A working version of software is produced in the first iteration of development cycle. This gives initial functioning software early on during the software life cycle. The following iterations then build on the initial software produced during the first iteration [Models].

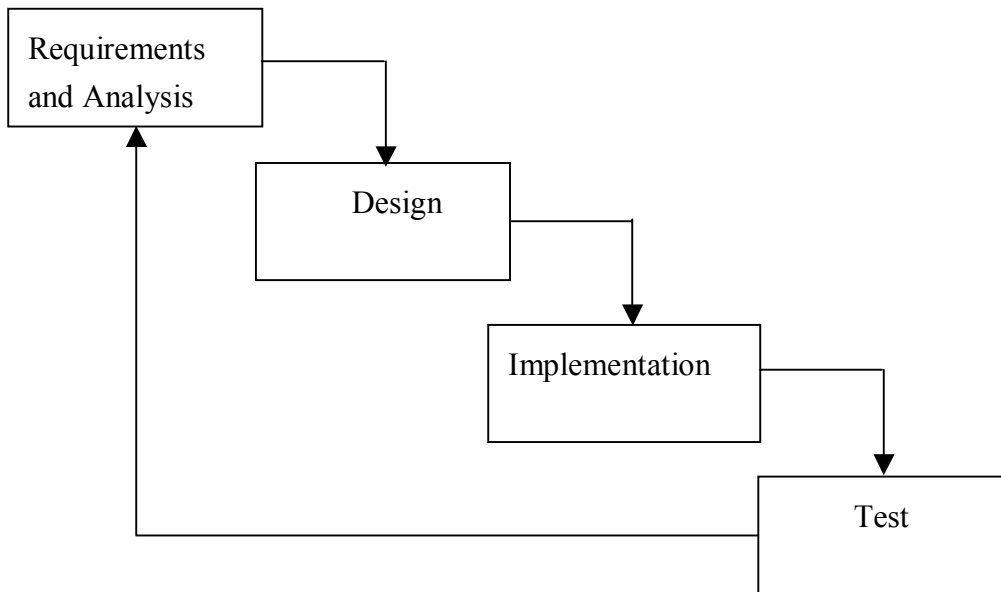


Figure 5- Incremental model phases

The AControllerTest's iterative lifecycle is carried out in repeating the following four phases in sequence as shown in [Cycle]:

Requirements phase - all the requirements of the software to be developed are collected and analyzed based on the needs. The aim of this iteration is to produce requirements specifications which could then be taken as input to the following phase of the iteration.

Design phase - designing the main blocks and components of the system along with their interfaces and interactions. The design could be a new design or an extension to an earlier design. This iteration produces a system architecture document. The software is then built based on the system architecture document which in turn produces a software design document.

Implementation phase - the actual coding work is done in this phase based on the software design document. The system is first developed in smaller portions called units. They are able to stand alone from a functional aspect and are integrated later on to form the complete software package.

Test phase - when the product is successfully implemented and is ready to run, the system is tested against the initial requirements. All the errors are reported and corrected measures are taken accordingly.

The advantages of the incremental model are [Cycle]:

- Simple and easy to use.
- Phases are processed and completed one at a time.
- Generates working software quickly and early during the software life cycle.
- More flexible – less costly to change scope and requirements.
- Easier to test and debug during a smaller iteration.
- Easier to manage risk because risky pieces are identified and handled during its iteration.
- Each iteration is an easily managed milestone.

The disadvantages of the incremental model are:

- Each phase of an iteration is rigid and do not overlap each other.
- Problems may arise pertaining to system architecture because not all requirements are gathered up front for the entire software life cycle.

### **3.2.2 Module testing with the AControllerTest**

The AControllerTest is build to perform testing at the module (unit) level. It aims to verify if the audio is routed to the correct path/medium as per needed. To do this, a number of test cases is implemented. The implementation of the test cases in the AControllerTest is based on the specifications given in the design specifications which contain the following information [Standards, 1998]:

1. Test design specification identifier - specifies a unique identifier for the design specifications.
2. Features to be tested - identifies the test items and describes the features and combinations of features. A reference to the associated requirements is also given for the features.
3. Approach refinement - specifies refinements to the test plan. Includes test techniques to be used and methods for analyzing the tests. It also provides criteria for test case selection and conditions for valid input and output values.
4. Test identification - contains a test case identifier and a brief description of each test case associated with the design.
5. Feature pass/fail criteria - specifies the criteria to be used for deciding if a feature passes and fails.

Based on the information given in the design specifications test cases are formulated in the form of test case specifications as given in the following section.

### **3.2.3 Specifications for the AControllerTest**

The AControllerTest implements all the test cases based on the specification given by the IEEE Standards for software test documentation (IEEE standard 829-1998). This specification suggests what is to be included in a test specification document. In other words, it aims to identify the features to be tested based on the design specification and their associated tests. Each design may contain many test cases and specifications. A test case specification usually contains the following structure in the given order [Standards, 1998]:

1. Test case specification identifier:  
Each case must have an identifier. The identifier could be referenced in other associated documents, e.g., design specification.
2. Test items:  
Identify and briefly describe the items and features to be exercised by the test case. An item here refers to a component of the system and a feature refers to the functionality of the system.
3. Input specifications:  
Specify each input to execute the test case. The input values could be such as numbers, characters and actions like pressing a key. All other needed input should be identified.
4. Output specifications:  
Specify the expected outputs from execution of the test with specified inputs.
5. Environmental needs:  
Identify the hardware, software and/or other necessary conditions to run the test case.
6. Special procedural requirements:  
Description of the constraints on testing procedures. This step is not needed for the AController testing.
7. Inter-case Dependencies:  
Identify the test cases that must be run before running this test case. This step is not utilized in the AControllerTest as all the cases are independent of each other.

Based on the above test case specifications, the design of the AControllerTest is carried out.

### 3.2.4 Design and implementation of the AControllerTest

The design of the testing component which is based on the above specification is shown in the class diagram of Figure 6.

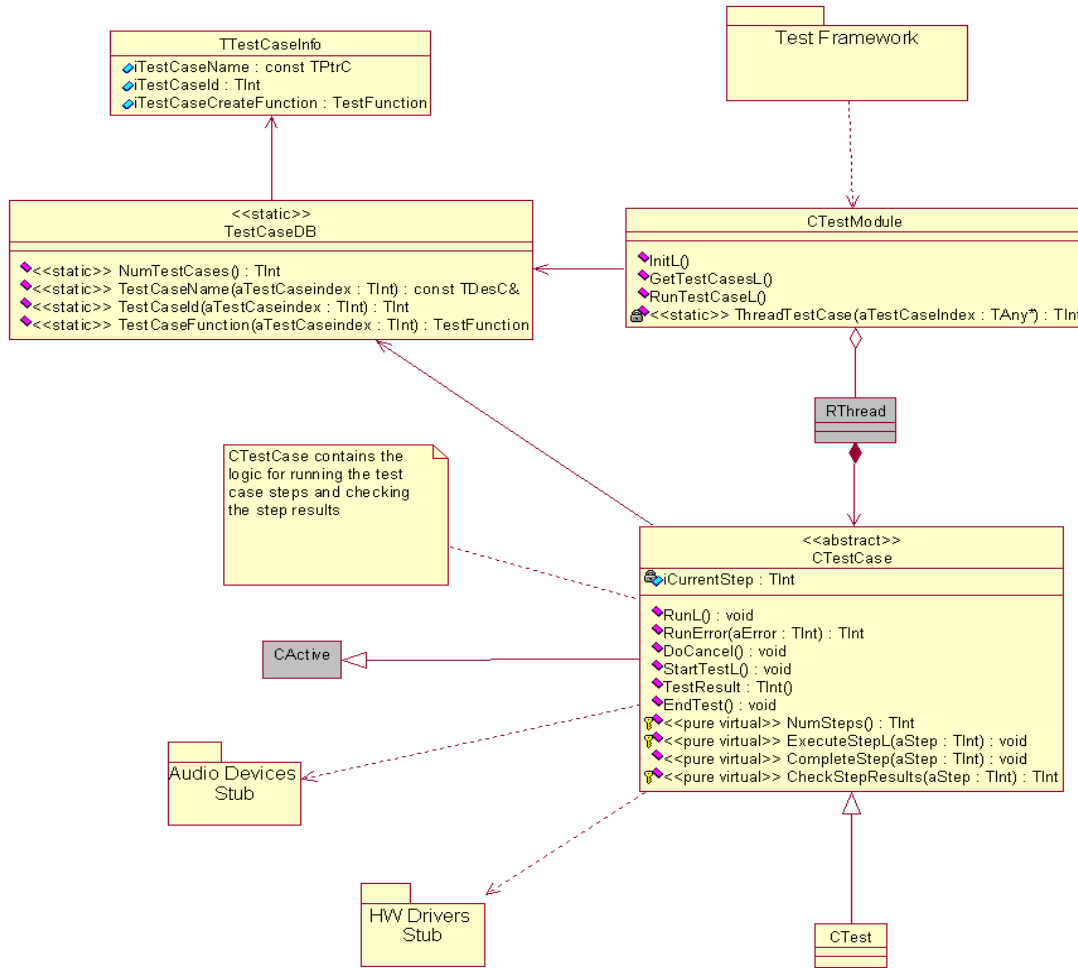


Figure 6- The class diagram showing the basic test framework

The class **TestCaseDB** is the repository of test cases. Every new test case's information is inserted into this. The **TestModule** class implements the controlling mechanisms related to the test cases and includes operations such as initialization of the test set in the repository and execution of test cases. This class is derived from the test framework. The

next class is CTestCase which includes all the common operations that are needed in all test case executions. These operations include instantiating test case, starting test case execution, cancelling test case execution, keeping track of total number of steps, executing each step and in turn checking the results of each step and indicating the completion of each step. This class forms the generalization of test cases. In this class diagram, the example specialization of the test case is represented with the CTest class.

The instantiation of a new test case is done by creating specialized objects of the CTestCase class. The class diagram given in Figure 7 explains how this is done. The diagram shows examples of two test cases, the FM radio playback and the MP3 playback.

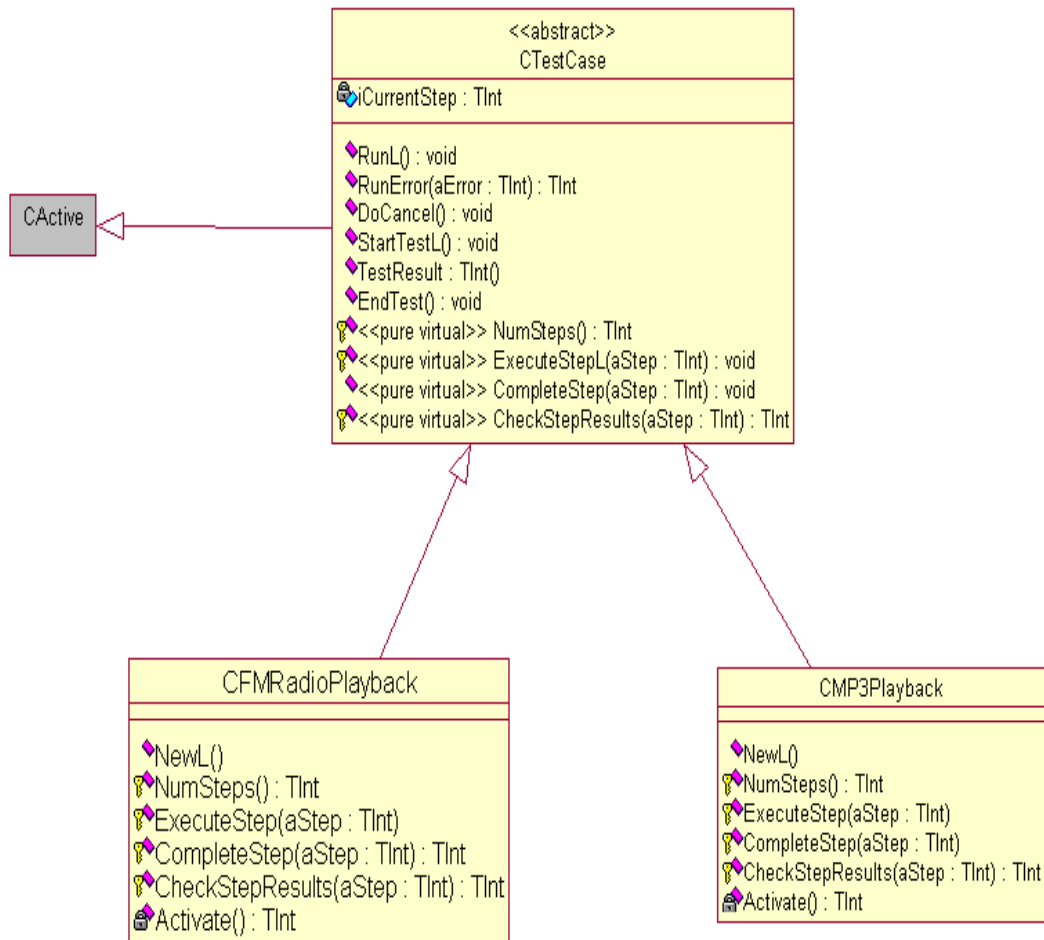


Figure 7- The class diagram showing test case instantiation

The sequence diagram in Figure 8 shows a typical test case sequence of AControllerTest component. The TestFramework component initiates the test case. The TestModule then initializes and gets the number of test cases in the module. The test case database collects information about the number of steps with respect to the test case. Once this is done, the processing of the test case starts and a respective thread is created. This new thread is needed as there are many test cases in the AControllerTest and if one of the test cases crashes (not merely fails) then the processing should continue with another test case and this is possible because the main thread is still executing. It must also be noted that separate test case runner threads are needed because the test cases are asynchronous in nature and the multi-threaded system makes the synchronization of test cases possible. The test runner thread stays alive till all the steps in the test case are executed and reported as a pass or a fail repetitively unless the test case has crashed. The main thread waits till the test runner thread completes its execution and then checks the test result from the thread exit reason. At the end of execution of all steps, a check is performed to see if the case is declared as a pass only if all the steps were declared as pass. Finally, the cleaning process of the objects starts and the execution of the test case ended.



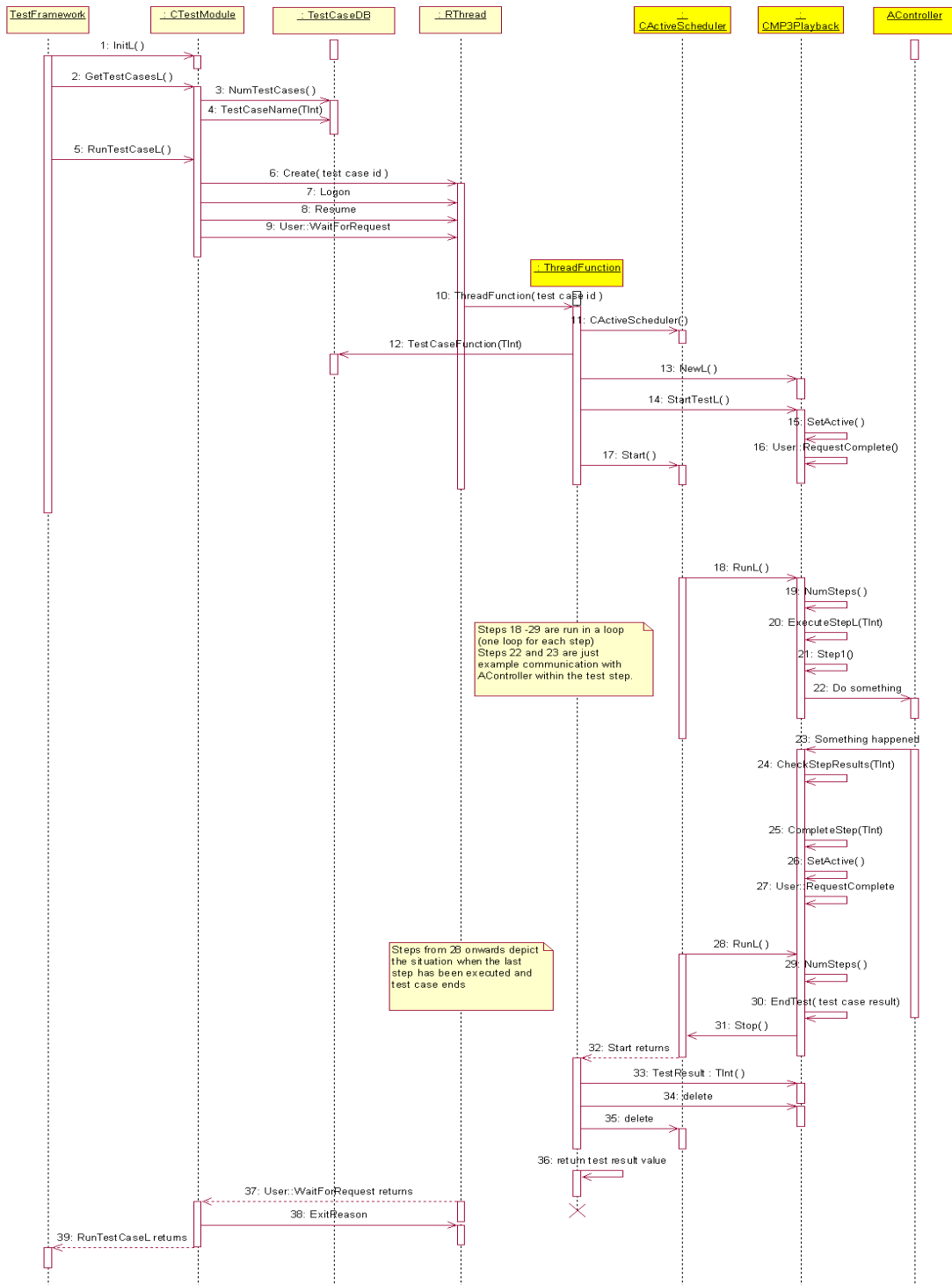


Figure 8- The sequence diagram of a typical test case execution

The implementation of the ACntrollerTest is carried out using the tools given in the following section.

### **3.3 Software tools used in the development**

#### **3.3.1 Symbian (SDK)**

The Symbian OS forms the most important component in the development of the S60 (formerly Series 60) based smart phones. It is specifically designed for mobile devices and has a number of advantages. Firstly, Symbian Ltd. is a consortium of leading mobile phone manufacturers and they all support the development with the Symbian. Secondly, it aims to develop a common operating system for mobile devices. Thirdly, the Symbian consortium offers licenses to the mobile device manufacturers. Manufacturers then develop their own user interfaces. These user interfaces could be licensed further. Fourthly, it offers low power and memory constraints which is a bottleneck in the development of mobile devices. Lastly, it is capable of pre-emptive kernel with multi-tasking capabilities and the ability to install third party software by the user. Symbian allows development on top of it to be carried out in various environments including C++ and Java [Symbian].

#### **3.3.2 Carbide.c++**

The underlying operating system of most of latest smart phones is Symbian. S60 provides a rich set of applications for these smart phones and it runs on top of Symbian OS. These S60 applications are developed using various tools. Carbide provides an environment for various tools that support Symbian C++ programming. Presently four versions of Carbide C++ are available [Forum Nokia, 2007]:

Carbide.c++Express—the complete tool for developers and learners for free. It can be used for developing commercial applications. For the sake of this thesis work Carbide.c++ Express is the chosen environment, since this work is to be carried out for educational purpose.

Carbide.c++Developer Edition—fully equipped for commercial developers and includes tools for on-device debugging and a graphical UI designer for S60 applications.

Carbide.c++Professional Edition— well suited for Symbian developers with performance enhancing capabilities. It facilitates on-device application performance analysis and enhanced debug support, with system-level and crash debugging.

Carbide.c++OEM Edition— best suited for device creators developing products. It has advanced device-creation features and more enhanced debugging capability for creation of devices based on Symbian OS.

### 3.4 Pros and cons

The AControllerTest's development comes with the regular white box testing benefits and problems. Some of them are listed here [Rajendran]:

Pros:

- It is now possible to test the AController's code without waiting for the other component's code.
- Testing and fixing of the problems is done at the same time.
- The cost is less as the bugs are removed at an earlier stage of the development.
- A standard testing technique is used to test the code.
- Debugging is easier as a small portion of the code is being tested with each test case.
- More in-depth testing is done as the cases are written by the programmers of the AController.
- It is made sure that the testing covers a large portion of the code.
- The testing and debugging cycle is short

Cons:

- The same test cases are repeated to test the same functionality of the code.
- Extra code is added to the actual code.
- The testing with the AControllerTest does not guarantee that the bugs will be prevented in the future cycles of the development.
- Module testing done with the AControllerTest does not guarantee that the system works seamlessly with other components in real hardware environment (e.g. due to timing issues).
- Functional testing cannot be replaced completely by module testing.

## **4. Testing with the AControllerTest**

This chapter is based on the testing performed with the AControllerTest. The included topics are tools used, problems faced, test case execution and results of test case execution.

### **4.1 Tools and problems**

The development, debugging and testing of the AControllerTest is carried out on the PC-based Series 60 emulator. The emulator mimics the real mobile device. Series 60 user interface has three logically divided areas: status pane, main pane and control pane. The status pane is an area where information about current application and status of the device (e.g., the battery charge indicator) is shown. The main pane displays the data of the application. The control pane holds the controlling soft keys. These soft keys are used to select the currently associated Options menu and labelled action. The emulator works with the PC mouse or the PC keyboard as any other PC based application [Edwards et al., 2004]. The Series 60 emulator is used to start the AControllerTest application.

The development process could be deemed as free of problems. However, there are couple of issue worth mentioning.

It is observed that some features could be enabled or disabled with some compilation flags in the AController. Therefore, it is impossible to fully run some of the implemented test cases. Some features are mutually exclusive. Therefore, if some flags are enabled to pass some test cases then others will fail at the same time. To verify all test cases, it is sometimes needed to run the test cases in different executions and re-compile the AController with a different set of flags in between the test runs to get all test case results.

It still takes some manual work to compile the stubs but this may be improved by writing some compilation scripts that would automate the process.

### **4.2 Test case execution**

This section explains the internal working of the AControllerTest by showing some manual test case steps in parallel with their equivalent in the AControllerTest executions. To do this, various test case examples are shown in the form of Tables 2, 3, 4, 5 and 6. Each table represents a test case and is given a proper heading to show the purpose of the test case.

Table 2 - Audio transfer from mono Bt headset to IHF during MO CS call

| Manual test step     | Expected results of manual test step     | AControllerTest step   | Expected results of AControllerTest   |
|----------------------|--|--|---|
| 1. Create MO CS call | 1. The call is created. No errors occur. | <ol style="list-style-type: none"> <li>1. Create MO CS call DL stream A</li> <li>2. Set stream A sink device to mono Bt headset</li> <li>3. Set volume of stream A to some given value</li> <li>4. Activate stream A</li> <li>5. Create MO CS call UL stream B</li> <li>6. Set stream B source device to mono Bt microphone</li> <li>7. Set stream B to un-mute</li> <li>8. Activate stream B</li> </ol> | <ol style="list-style-type: none"> <li>1. Stream A is created</li> <li>2. Sink device is set to mono BT headset</li> <li>3. Volume is saved correctly</li> <li>4. DL audio path power is set ON. Mono Bt analog volume is set correctly. DL audio signal is routed to mono Bt audio path</li> <li>5. Stream B is created</li> <li>6. Source device is set to mono Bt microphone</li> <li>7. Stream B gain is non-zero</li> <li>8. UL audio path power is set ON. Mono Bt microphone signal is routed to UL</li> </ol> |

|   |  |  |   |
|---|--|--|---|
| 2. Transfer audio to IHF                                | 2. Audio is transferred correctly to IHF. No errors occur.             | 9. Set sink device to IHF for stream A<br>10. Set stream B source device to microphone                     | 9. Sink device is set to IHF. Audio routing to mono Bt audio path is terminated. IHF speaker power is set ON. IHF analog volume is set correctly. DL audio signal is routed to IHF<br>10. Stream B source device is set to microphone. Mono Bt microphone audio path is terminated. Phone microphone power is set ON. Microphone signal is routed to UL |
| 3. Transfer audio to already connected mono Bt headset. | 3. Audio is transferred correctly to mono Bt headset. No errors occur. | 11. Set sink device to mono Bt headset for stream A<br>12. Set source device to Bt microphone for stream B | 11. Sink device is set to mono Bt headset. IHF speaker power is set OFF. DL audio signal is routed to mono Bt audio path. Bt analog volume is set.<br>12. Source device is set to Bt headset for stream B. Phone microphone power is set OFF. Mono Bt microphone signal is routed to UL. Bt gain is non-zero  |

|                  |                                  |  |   |
|------------------|----------------------------------|--|---|
| 4. End the call. | 4. Call ends and no errors occur | 13. Stop stream B<br>14. Delete stream B<br>15. Stop stream A<br>16. Delete stream A | 13. IHF power is set OFF and DL audio path power is set OFF<br>14. Stream B is deleted<br>15. Microphone power is set OFF and UL audio path power is set OFF<br>16. Stream A is deleted |
|------------------|----------------------------------|--|---|

Table 3 - Audio transfer from IHF to USB headset

| Manual test step  | Expected results of manual test step                                     | AControllerTest step   | Expected results of AControllerTest  |
|---|--|--|--|
| 1. Play a song in the music player. The song is heard from the IHF. | 2. The song is played and heard from the IHF. No errors occur.           | 1. Create audio playback stream A<br>2. Set stream A sink device to IHF<br>3. Set volume of stream A to the some given value<br>4. Activate stream A | 1. Stream A is created<br>2. Sink device is set to IHF<br>3. Volume is saved correctly<br>4. Stream A is activated. IHF speaker power is set ON. IHF analog volume is set. |
| 1. Attach the USB headset   | 2. Song is played and is heard through the USB headset. No errors occur. | 5. Set stream A sink device to USB<br>6. Set volume of stream A to some given value  | 5. Set the IHF power OFF. Set USB output power ON. Audio is routed to USB output<br>6. USB volume is set correctly.  |

|                            |  |  |   |
|----------------------------|--|--|---|
| 2. Stop the music playback | 3. Playback is stopped and no errors occur | 7. Stop stream A<br>8. Delete stream A | 7. USB power is set OFF<br>8. Stream A is deleted |
|----------------------------|--|--|---|

Table 4 - Playing ringing tone with wired accessory

| Manual test step                                     | Expected results of manual test step   | AControllerTest step  | Expected results of AControllerTest  |
|--|--|---|--|
| 1. Wired headset is connected.<br>Receive MT CS call | 1. Phone starts to play ringing tone from IHF and the attached wired headset. No errors occur. | 1. Create generic audio playback stream A<br>2. Set stream A first sink device to IHF<br>3. Set stream A second sink device to wired headset<br>4. Set volume of stream A to some given value<br>5. Activate stream A | 1. Stream A is created<br>2. First sink device is set to IHF<br>3. Second sink device is set to wired headset<br>4. Volume value is saved correctly<br>5. Generic playback path power is set ON. IHF speaker power is set ON. Wired headset speaker power is set ON. Generic playback audio signal is routed to IHF and wired headset. |
| 2. Reject the call                                   | 2. Ringing tone stops. No errors occur.  | 6. Stop stream A<br>7. Delete stream A  | 6. IHF and wired speaker headset power is set OFF. Generic playback path power is set OFF.<br>7. Stream A is deleted   |



Table 5 - Digital radio recording case

| Manual test step   | Expected results of manual test step  | AControllerTest step   | Expected results of AControllerTest  |
|--|---|--|--|
| <p>1. Wired headset is connected. Play a channel in FM radio</p> | <p>1. The radio plays and is heard from the wired headset. No errors occur.</p> | <p>1. Create FM radio playback stream A<br/>                 2. Set stream A sink device to wired headset<br/>                 3. Set stream A source device to FM radio receiver<br/>                 4. Set volume of stream A to some given value<br/>                 5. Activate stream A</p> | <p>1. Stream A is created<br/>                 2. Stream A sink device is set to wired headset<br/>                 3. Stream A source device is set to FM radio receiver<br/>                 4. Volume is saved correctly<br/>                 5. Stream A is activated. FM radio receiver power is set ON. Headset speaker power is set ON. Headset analog volume is set. FM radio signal is routed to headset.</p> |
| <p>2. Start recording the FM radio playback</p>                  | <p>2. Recording is started and no error occurs.</p>                             | <p>6. Create recording stream B<br/>                 7. Set stream B source device to FM radio receiver<br/>                 8. Activate stream B</p>  | <p>6. Stream B is created<br/>                 7. Stream B source device is set to FM radio receiver<br/>                 8. Generic recording audio path power is set ON. FM radio signal is routed to generic recording audio path</p>   |

|                           |   |  |  |
|---------------------------|---|--|--|
| 3. Stop recording         | 3. Recording is stopped and no errors occur | 9. Stop stream B<br>10. Delete stream B  | 9. Generic recording audio path power is set OFF<br>10. Stream B is deleted                          |
| 4. Stop FM radio playback | 4. FM radio is stopped and no errors occur  | 11. Stop stream A<br>12. Delete stream A | 11. FM radio receiver power is set OFF. Headset speaker power is set OFF.<br>12. Stream A is deleted |

Table 6 - Video call

| Manual test step   | Expected results of manual test step   | AControllerTest step   | Expected results of AControllerTest   |
|--|--|--|---|
| 1. Create a video call and both ends can hear each other | 1. Video call is setup correctly. The caller and the receiver can hear the audio and see the video at the same time. No delays occur. No errors occur. | 1. Create video call downlink (DL) stream A<br>2. Set stream A sink device to IHF<br>3. Set volume of stream A to some given value<br>4. Activate stream A<br>5. Create video call uplink (UL) stream B<br>6. Set stream B source device to Microphone | 1. Stream A is created<br>2. Sink device is set to IHF<br>3. Volume is saved correctly<br>4. DL audio path power is set ON. IHF speaker power is set ON. IHF analog volume is set correctly.<br>5. Stream B is created<br>6. Source device is set to Microphone<br>7. Stream B gain is non-zero |

|                               |  |   |  |
|-------------------------------|--|---|--|
|                               |  | <p>7. Set stream B to un-mute<br/>8. Activate stream B</p>                                    | <p>8. UL audio path power is set ON.<br/>Microphone power is set ON.<br/>Microphone signal is routed to UL</p>   |
| <p>2. Stop the video call</p> | <p>2. Video call is stopped and no errors occur.</p> | <p>9. Stop stream A<br/>10. Delete stream A<br/>11. Stop stream B<br/>12. Delete stream B</p> | <p>9. IHF power is set OFF and DL audio path power is set OFF<br/>10. Stream A is deleted<br/>11. Microphone power is set OFF and UL audio path power is set OFF<br/>12. Stream A is deleted</p> |

### 4.3 Results of test case executions

In this section, the results of AControllerTest are discussed by identifying the changes in the process, possible difficulties and benefits.

The AControllerTest is advantageous as it finds errors in the early development cycle. Errors are found long before they could be found in the manual testing phase. This has the advantage that if an error is found in the early stage then it's easier and cheaper in terms of time and money to fix it.

The AControllerTest assures that the newly written code is working and does not break the old code. The programmers have shown their willingness to change the code right away because the test cases for the AControllerTest are written for small parts of the code and it is easier to make fixes at this level rather at a later stage when the code has become old and changing it might cause side effects in terms of failures in some other parts of the code.

On the other hand, the AControllerTest has made the development process a bit more complex due to the fact that there are many boundary values that are to be taken into account. However, it gives more confidence to programmers that the code written by them is working. In addition, the development process could be argued as more flexible, as the code could be changed right away without affecting other modules.

The AControllerTest may help in deciding the product shipment decision at an early stage, if some serious errors are found and they require more time to be fixed. Although the AControllerTest reduces the effort required in the manual testing phase for quality assurance but it does not guarantee it's replacement.

The AControllerTest has changed the development life cycle for the AController. It was said in Chapter 2 that AController's development life cycle is incremental but now it can be seen more like test driven development. According to Jones [2004] test-driven development (TDD) could reduce the rate of errors, produced during the process of software development by 18% to 50%. The test-driven development itself is a practice of extreme programming in which tests and code are development in parallel to each other. The test-driven development is carried out in the following sequence [Desai et al., 2008]:

1. Add a new test for an unimplemented unit of functionality.
2. Run all previously written tests and see the newly added test fail.
3. Write code that implements the new functionality.
4. Run all tests and see them succeed.
5. Re-factor (rewrite to improve readability or structure).
6. Re-run the test cases.

#### 7. Start at the beginning (repeat).

The AController's design is getting more complex as new features are being added to the system. Therefore, new test cases are constantly being created in the AControllerTest. This new testing module has forced the programmers to write the code in small parts which could easily be tested independently of other modules. The programmers are more careful about the edge conditions and their consequences. The test cases are now run many times before and after the integration of AController with other components.

The benefits of the AControllerTest are summarized here:

- Test cases are being written in parallel with the code. Therefore, it is easier for the programmers to modify the code in case of an error or otherwise write the test cases that best fits the functionality of the code.
- Verifies within a few seconds that the programmer's code is working fine.
- Give the programmers a better insight of the code that they are working with.
- It is helpful to the management as it can verify that all the code is integrated and it works fine with the audio modules.
- Identifies audio errors where humans are weak at interpretations. For example, measuring different volume levels of an audio playback where the difference between the two consecutive volume levels is very narrow.
- Test cases are run for every small change in the code to verify that the code did not break any other code.
- It is a big time saver in the long run.
- Provides a larger set of regression test cases.
- It is believed to reduce the number of bugs in the long run.
- Provides automatic testing mechanism.
- Test cases represent real use cases.

Another aspect that might be of interest is the code coverage analysis. The AController's code coverage analysis is obtained using the Testwell CTC++ test coverage and analysis tool for C and C++ [Testwell]. The results show 67% code coverage for the existing functionality. Furthermore, the coverage is based on the coverage of number of files, methods, lines and boundary values. It is targeted to get a code coverage analysis of above 80% and work is continuing in this direction.

## 5. Conclusions

The latest mobile phones are termed mobile computing devices and they convert sound into digital audio. Digital audio is processed with a digital signal processor to produce digitized signals. Digitized signals are routed to various audio related components in the system including applications, headsets, speakers, music, ringing tones, and others. The routing of the digitized audio to various components requires complex internal processing. It is of extreme importance that the audio is routed to the right path with right attributes without errors. To do this, various software modules interact with each other in the audio system including the AController.

The main task of the AController's is to route audio with some given attributes to right channel. It primarily interacts with two other software modules namely the AServer and the AMapper. The AServer creates logical streams using the AStreamer component of the AController and then sends the request of the logical stream to the ARouter for providing the right channel to the physical stream. The AMapper component interacts with the domestic operating system for transforming physical stream attributes to hardware configuration settings. In other words, the AController is an intermediary between the Symbian MMF and the domestic operating system.

The software development life cycle of the audio system including the AController is incremental, i.e., software modules are developed, integrated and released continuously. Therefore, it is necessary to test the functions of the AController with every new software release. The objective of this work is to test the functionality of the AController during the development phase of various audio modules. To achieve the objective, a module test framework for the AController is developed and implemented. The newly developed module test framework is called the AControllerTest.

The AControllerTest is designed and implemented on the concept of stubs. The stubs substitute the real code of software modules and allow the calling or interacting programs to interact with them as if they are doing with the real module. Usually, stubs are used for the sake of testing purposes as they hide the real functionality of a software module. This approach allows to quickly implement different parts of a program without much thought of other components and concentrating on the part of the system that is under development. Similarly, in the AControllerTest framework, modules of the AController are stubbed and they are called as AServer stub, Audio Devices stub, HW Drivers stub and AMapper stub. These stubbed software modules provide the functionality as if they were real non-stubbed modules. It must be noted here that the AMapper stub publishes the status of the services to publish and subscribe component. The TestModule of the

AServer module is registered for checking test step results to publish and subscribe component as a subscriber and is therefore notified of the success or failure of the request sent by it.

Furthermore, the AControllerTest implements the test cases based on the specification given by IEEE Standards for software test documentation which gives guidelines about what is to be included in a test case specification. The initial testing of the AControllerTest is performed on the PC-based Series 60 emulator.

With the development of the AControllerTest, the problems which were visible only when releases were available for the testers to perform the end user testing are now highlighted in the implementation phase. As the errors are found in the early stage, it is easier and cheaper in terms of time and money to fix them. Other benefits include the speeding-up the process, enhancing the testing capabilities, increasing the test sets which are grouped according to related audio features, and making it easier for developers to verify code changes. It is a motivating factor for the programmers to write the code as it gives them an immediate result to see if their code is working. It verifies within few minutes that the code written is working and does not break other parts of code. It is easier to change the code right away as the test cases are written for small parts of the code.

The AControllerTest has made the development process a bit more complex as there are many boundary values and conditions to be checked while developing the code. On the other hand, the development process may be considered as more flexible, because the code could be changed during the development process.

Efforts were made to cover all the functionality associated with the audio processing. The code coverage analysis of the AControllerTest shows 67% of the code coverage which is less than the targeted 80%. It must be noted that the AControllerTest reduces the efforts required in the manual testing phase for quality assurance but it does not provide full replacement of manual testing. Furthermore, it does not give a guarantee that the system will work when all the modules are combined with their actual (un-stubbed) functionality.

The AControllerTest still requires more efforts in various directions. New test cases are regularly being added to the testing module with the additions of new functionality. It is deemed highly advantageous if the module is taken more into use by writing automated scripts for testing the latest written code during the development life cycle on daily basis.

## References

- [Barriocanal et al., 2002] Elena Garcia Barriocanal, Miguel-Angel Sicilia Urban, Ignacio Aedo Cuevas, Paloma Diaz Perez, An experience in integrating automated unit testing practices in an introductory programming course, *ACM SIGCSE Bulletin*, **34** (4), 2002, 125 – 128.
- [Bhawani] Chetan Bhawani, Latest trends in mobile phones, Available as <http://ezinearticles.com/?Latest-Trends-in-Mobile-Phones&id=546868>.
- [Bo et al., 2007] Jiang Bo, Long Xiang, Gao Xiaopeng, Mobile test: A tool supporting automatic black box test for software on smart mobile devices, In: *Proc. of the Second International Workshop on Automation of Software Test*, 2007, 8.
- [Cycle] Software Development Life Cycle Models. Available as <http://codebetter.com/blogs/raymond.lewallen/archive/2005/07/13/129114.aspx>.
- [Desai et al., 2008] Chetan Desai, David Janzen, Kyle Savage, A survey of evidence for test-driven development in academia, *SIGCSE Bulletin*, 40 (2), 2008, 97-101.
- [Edwards et al., 2004] Leigh Edwards, Richard Barker, and the Staff of EMCC Software Ltd., *Developing Series 60 Application, A guide for Symbian OS C++ Developers*, Addison-Wesley, 2004.
- [Forum Nokia, 2007] Forum Nokia, Carbide Development Tools for Symbian OS C++, Available as [http://www.forum.nokia.com/info/sw.nokia.com/id/cae9ea59-eee0-4b98-aaa2-1b6ecd879222/Carbide\\_cpp\\_Introductory\\_White\\_Paper\\_V1\\_1\\_en.pdf.html](http://www.forum.nokia.com/info/sw.nokia.com/id/cae9ea59-eee0-4b98-aaa2-1b6ecd879222/Carbide_cpp_Introductory_White_Paper_V1_1_en.pdf.html).
- [Glossary] Mobile phone glossary, Available as [http://www.westlake.co.uk/Mobile\\_Phone\\_Glossary.htm](http://www.westlake.co.uk/Mobile_Phone_Glossary.htm).
- [Johnson, 2004] Nick Johnson, Audio streaming: How to successfully stream audio on Symbian OS v7.0s, July 2004. Available as [https://developer.symbian.com/wiki/download/attachments/50987025/Audi0Stream\\_SymbianOSv1.1.pdf?version=1](https://developer.symbian.com/wiki/download/attachments/50987025/Audi0Stream_SymbianOSv1.1.pdf?version=1).
- [Jones, 2004] Christopher G. Jones, Test-driven development goes to school, *Journal of Computing Sciences in Colleges*, **20** (1), 2004, 220-231.
- [MMF Client] Multi Media Framework Client Overview. Available as [http://www.symbian.com/developer/techlib/v9.1/docs/doc\\_source/guide/multimedia-subsystem-guide/n100ea/MMFClientOverview.guide.html](http://www.symbian.com/developer/techlib/v9.1/docs/doc_source/guide/multimedia-subsystem-guide/n100ea/MMFClientOverview.guide.html).
- [MMF Controller] Class CMMFController. Available as [http://www.symbian.com/developer/techlib/v9.1/docs/doc\\_source/reference/reference-cpp/N10296/CMMFControllerClass.html](http://www.symbian.com/developer/techlib/v9.1/docs/doc_source/reference/reference-cpp/N10296/CMMFControllerClass.html).



- [MMF DevSound] Class CMMFDevSound. Available as  
[http://www.symbian.com/developer/techlib/v9.1docs/doc\\_source/reference/reference-cpp/N10296/CMMFDevSoundClass.html](http://www.symbian.com/developer/techlib/v9.1docs/doc_source/reference/reference-cpp/N10296/CMMFDevSoundClass.html).
- [Mobile Audio, 2007] Mobile Audio Working Group of the Interactive Audio Special Interest Group, *Mobile audio report and recommendations*, MIDI Manufacturers Association, 2007. Also available as  
<http://www.iasig.org/pubs/mawg-rpt.pdf>.
- [Models] Software Process Models. Available as  
[http://www.elektroniklager.de/sw/e\\_dta-sw-process.htm](http://www.elektroniklager.de/sw/e_dta-sw-process.htm).
- [Music, 2006] S60 Platform: Music Application Developer's Guide, November 2006. Available as  
[http://www.forum.nokia.com/info/sw.nokia.com/id/d790ede9-31f7-4f76-8ed2-f6b643730743/S60\\_Platform\\_Music\\_Application\\_Developers\\_Guide.html](http://www.forum.nokia.com/info/sw.nokia.com/id/d790ede9-31f7-4f76-8ed2-f6b643730743/S60_Platform_Music_Application_Developers_Guide.html).
- [Märsynaho, 2006 ] Maaret Märsynaho, Testing smartphone audio software, University of Oulu, Department of Electrical and Information Engineering, Master's Thesis, 2006.
- [Nurvitadhi, 2003] Eriko Nurvitadhi, Trends in Mobile Computing: A Study of Mobile Phone Usage in the United States and Japan, 2003. Also available as  
<http://www.ece.cmu.edu/~enurvita/docs/IDthesis.pdf>.
- [Pan, 1999] Jiantao Pan, Software Testing, Available as  
[http://www.ece.cmu.edu/~koopman/des\\_s99/sw\\_testing/#reference](http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/#reference).
- [Parrington] Graham D Parrington, A Stub Generation System For C++. Available as  
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.8496>.
- [Pirozzi] Rob Pirozzi, Introduction to Software Testing, Available as  
[http://www.logigear.com/newsletter/introduction\\_to\\_software\\_testing.asp](http://www.logigear.com/newsletter/introduction_to_software_testing.asp).
- [Rajendran] R.Venkat Rajendran, White paper on Unit Testing. Available as  
<http://www.mobilein.com/WhitePaperonUnitTesting.pdf>.
- [Ravindran et al., 2005] Sourabh Ravindran, Paul Smith, David Graham, Varinthira Duangudom, David V. Anderson, Paul Hasler - Towards Low-Power on-Chip Auditory Processing. *EURASIP Journal on Applied Signal Processing*, 2005, 1082 – 1092.
- [Shackman, 2005] Mark Shackman, Publish and subscribe, Symbian developer network, 2005. Available as  
[http://developer.symbian.com/main/downloads/papers/publishandsubscribe/PublishAndSubscribe\\_v1.0.pdf](http://developer.symbian.com/main/downloads/papers/publishandsubscribe/PublishAndSubscribe_v1.0.pdf).

- [Sound] Anonymous, Background information for Sound, Available as  
[http://www.sciencetech.technomuses.ca/english/schoolzone/Info\\_Sound.cfm](http://www.sciencetech.technomuses.ca/english/schoolzone/Info_Sound.cfm).
- [Standards, 1998] IEEE standard for software test documentation, Software engineering committee of the IEEE computer society, 1998. Also available as  
<http://citeseer.ist.psu.edu/514745.html>.
- [Stotts, 1982] Paul David Stotts, Jr., A comparative survey of concurrent programming languages, *ACM SIGPLAN Notices* 17 (9), 1982, 76 – 87.
- [Symbian] Symbian OS, Available as  
<http://www.symbian.com/> and  
[http://developer.symbian.com/main/learning/press/books/s60tut/s60tut\\_sample.pdf](http://developer.symbian.com/main/learning/press/books/s60tut/s60tut_sample.pdf).
- [Testwell] Testwell CTC++Test Coverage Analyzer for C/C++, Available as  
<http://www.testwell.fi/ctcdesc.html>
- [Torres] Gabriel Torres, How analog-to-digital converter (ADC) works, Available as:  
<http://www.hardwaresecrets.com/article/317/2>.
- [White box] White box testing. Available as  
<http://agile.csc.ncsu.edu/SEMaterials/WhiteBox.pdf>.

Note: The hyperlinks used in the references were accessed before the end of October, 2008.