HELSINKI UNIVERSITY OF TECHNOLOGY
Department for Computer Science and Engineering

Jussi Hynninen

A software-based system for listening tests

This Master's Thesis has been submitted for official examination for the degree of Master of Science in Espoo on May 31st, 2001

Supervisor of the Thesis

Professor Matti Karjalainen

HELSINKI UNIVERSITY OF TECHNOLOGY
Department for Computer Science
and Engineering

ABSTRACT OF MASTER'S THESIS

Author:	Jussi Hynninen
Name of the thesis:	A software-based system for listening tests
Translation in Finnish:	Ohjelmistopohjainen järjestelmä kuuntelukokeita varten
Date:	31.5.2001
Number of pages:	90
Professorship:	S-89 Acoustics and audio signal processing
Field of study:	Information science
Supervisor:	prof. Matti Karjalainen

In this thesis, a software-based system is developed as a flexible generic platform for subjective audio testing. The system provides wide range of subjective audio tests, including standardized tests. It eliminates a lot of the complexity of setting up such experiments. As the system is software-based, little additional hardware in addition to a computer is required to perform tests.

The system is scalable and customizable, allowing creation of new kinds of tests that are not covered by existing standardized tests. Geared for multichannel audio, the system runs on SGI/IRIX platform and allows upto 32 channels of 24-bit digital audio output. Graphical user interface panels are used by test subjects for grading. Multiple test subjects can participate in a test at the same time, reducing the time needed for testing.

No analysis of test results is performed by the software. Instead, test data produced by each test session is stored in a tabulated text file for final analysis with generally available statistical tools.

The system has been used extensively in practical listening tests and it has been found reliable and meets all requirements of the specified tests.

Keywords: Listening tests, audio signal processing, psychoacoustics

TEKNILLINEN KORKEAKOULU Tietotekniikan osasto	DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Jussi Hynninen
Työn nimi:	A software-based system for listening tests
Käännös suomeksi:	Ohjelmistopohjainen järjestelmä kuuntelukokeita varten
Päivämäärä:	31.5.2001
Sivumäärä:	90
Professuuri:	S-89 Akustiikka ja äänenkäsittelytekniikka
Pääaine:	Informaatiotekniikka
Työn valvoja:	prof. Matti Karjalainen

Tässä työssä on kehitetty geneerinen ja joustava ohjelmistopohjainen järjestelmä subjektiivisia kuuntelutestejä varten. Järjestelmä tarjoaa laajan valikoiman subjektiivisia testityyppejä, sisältäen myös standardoituja testityyppejä. Järjestelmä helpottaa tälläisten testien rakentamista. Koska järjestelmä on ohjelmistopohjainen, tietokoneen lisäksi lisälaitteistoa tarvitaan vain vähän.

Järjestelmä on skaalautuva ja muunneltava ja antaa mahdollisuuden luoda uusia testityyppejä, jotka eivät sisälly standardoituihin tyyppeihin. Järjestelmä on suunnattu monikanavaisen audiotekniikan kuuntelukokeisiin, toimii SGI/IRIX-laitteistolla ja tarjoaa parhaimmillaan 32-kanavaisen, 24-bittisen digitaalisen äänen tuotannon. Koehenkilöt antavat vastaukset käyttäen graafisia paneeleja. Useampi koehenkilö voi suorittaa testiä samaan aikaan, jolloin säästetään aikaa.

Ohjelmisto ei tee testitulosten analyysiä. Sen sijaan testin tuottama tieto talletetaan taulukkomuotoisena tekstinä. Lopullinen analyysi voidaan tehdä yleisesti saatavilla olevilla tilastollisilla työkaluilla.

Järjestelmää on käytetty laajasti käytännön kuuntelukokeissa. Se on todettu luotettavaksi ja täyttää kaikki määriteltyjen testien tarpeet.

Avainsanat: Kuuntelukokeet, audio-signaalinkäsittely, psykoakustiikka

Preface

This work is a Master's Thesis carried out for the Laboratory of Acoustics and Audio Signal Processing of the Helsinki University of Technology (HUT), and it describes the listening test system (GuineaPig2) I developed during 1997-99 for the Acoustics Laboratory and Nokia Research Center (NRC).

The Acoustics laboratory was a great place to work; the atmosphere and the people were wonderful. Lunch and coffee breaks were inspirational. I'd specially like to say a big hello to the folks of "The Ladies Room": Hanna "Traktori" Järveläinen (for using the testing system I wrote), and Riitta "Beefcake" Väänänen, Ville "VBAP" Pulkki, and Matti "Mairas" Airas. The same goes to Miikka "Vasara" Huttunen, (a roommate here at work for a couple of years), and Martti "Hype" Rahkila.

I would also like to thank my supervisor professor Matti Karjalainen for the encouragement and help I received in the writing process and for being patient with my slow rate of writing the thesis.

Dedicated to the loving memory of Diiru, Iitu, Nalle, Sessu, Etti, and Liisa.

Espoo, 1.6.2001

Jussi Hynninen

Contents

1 Introduction **1**

2 Listening tests **3**

2.1 Systems Analysis of the Auditory Experiment . . . 3

2.1.1 Scaling . . . 4

2.1.2 An example of a traditional listening test . . . 6

2.1.3 Some applications for listening tests . . . 7

2.1.4 Other considerations when conducting listening tests . . . 8

2.2 Subjective test paradigms . . . 9

2.2.1 Single stimulus . . . 9

2.2.2 Paired comparison methods . . . 10

2.2.3 The rank order paradigm . . . 12

2.2.4 Adaptive methods . . . 13

3 System architecture **15**

3.1 Test configuration files . . . 16

3.2 Java-package hierarchy . . . 17

4 Sound player **19**

4.1 Audio output features . . . 19

4.1.1 Audio output devices and interfaces . . . 19

4.1.2 Support for multiple output devices . . . 20

4.1.3 Number of output channels . . . 21

4.1.4 Sample rates . . . 21

4.1.5 Audio precision . . . 21

4.2 Audio file formats . . . 22
4.3 Virtual players . . . 22
4.4 Volume levels . . . 24
4.4.1 Player output level . . . 25
4.4.2 Sample volume level . . . 25
4.4.3 Sample volume level calibration factor . . . 25
4.5 Mixing . . . 26
4.6 Drop-out handling . . . 26
4.7 Delay and latency . . . 26
4.8 Operations in synchrony . . . 27
4.9 Sound player C-module (sndplay) . . . 27
4.9.1 Initialization . . . 28
4.9.2 Sound generation loop . . . 28
4.9.3 Communication with Java-module . . . 30
4.10 Sound player Java-module . . . 30
4.10.1 Players . . . 31
4.10.2 Samples . . . 31
4.10.3 Volume levels . . . 32
4.10.4 Events . . . 32

5 Test engine **33**
5.1 Test items . . . 33
5.1.1 Defining test items . . . 34
5.1.2 Item results . . . 35
5.2 Playlists . . . 36
5.3 Sample playback sequence . . . 37
5.3.1 Fixed playback sequence . . . 37
5.3.2 Free sample switching . . . 38
5.4 Answering time limit . . . 39
5.5 Most comfortable listening level . . . 40
5.6 Multiple subjects concurrently . . . 41
5.6.1 Fixed sequence with multiple listeners . . . 41

5.6.2 Multiple independent listeners 41
5.7 Test process 41
5.7.1 Test session set-up 42
5.7.2 Test start 43
5.7.3 Test item testing 43
5.7.4 Test end 46
5.8 Test types 47

6 Subject's user interfaces 50

6.1 User interface panel 50
6.1.1 Basic panel interface 52
6.1.2 MCLL setting interface 52
6.1.3 Timeout warning interface 53
6.2 User interface components 53
6.2.1 Questions 53
6.2.2 Controls 54
6.2.3 Monitors 55
6.3 Provided user interface components 55
6.3.1 Scales 55
6.3.2 Multiple-choice 58
6.3.3 Rank order 59
6.3.4 Sample-play 59
6.3.5 Button 60
6.3.6 MCL level controller 60
6.3.7 Time-out indicator 61
6.3.8 Done-button 62
6.3.9 Test status monitor 62
6.4 Remote terminals 63
6.5 Tools 63
6.5.1 Font tester 63
6.5.2 UI panel tester 64
6.5.3 Remote UI server tester 65

6.5.4 Remote UI panel client as an application 65
6.5.5 Remote UI panel client as a Java applet 65

7 Test results processing **66**

7.1 Format of exported results file 66
7.2 Exported information 67
7.3 Results output configuration 68

8 Discussion **69**

8.1 Java 69
8.2 SGI 70
8.3 Sound player 70
8.4 Test engine 71
8.5 User interfaces 71
8.6 Result processing 72

9 Conclusions **73**

Bibliography **76**

A SGI Workstation Audio Features **77**

B Web links **80**

List of Symbols

dBfs	Decibels, full scale.
v_{pct}	Gain in percent scale.
v_{dBfs}	Gain in decibel scale.
v_{lin}	Gain in linear scale.

List of Abbreviations

ADAT	The Alesis ADAT Optical I/O interface
AF	Silicon Graphics Audio File Library
AIFF	Audio Interchange File Format
AIFF-C	Audio Interchange File Format with Compression
AL	Silicon Graphics Audio Library
ANOVA	Analysis of variance
API	Application Programming Interface
AWT	Java's Abstract Window Toolkit
CCR	Comparison Category Rating
CMOS	Comparative Mean Opinion Score
DAC	Digital to analog converter
DD	Dolby Digital
DM	IRIS Digital Media Libraries
DMOS	Degradation Mean Opinion Score
DTS	Digital Theater Systems
GP	GuineaPig
GP2	GuineaPig2
GUI	Graphical user interface
ITU	International Telecommunication Union
ITU-T	Telecommunication standardization sector of ITU
JDBC	Java Database Connectivity
JDK	Java Development Kit
JND	Just noticeable difference
JVM	Java Virtual Machine
LAN	Local area network
MCLL	Most comfortable listening level
MOS	Mean Opinion Score
RMI	Remote Method Invocation
RT	Real-time
SDDS	Sony Dynamic Digital Sound
SP	Sound player
SQL	Structured Query Language
TAFC	Two alternatives forced choice
UI	User interface
VP	Virtual player
WAVE	Microsoft RIFF WAVE File Format

Chapter 1

Introduction

guinea pig n 1: a small stout-bodied short-eared nearly tailless domesticated rodent (*Cavia cobaya*) often kept as a pet and widely used in biological research **2:** a subject of scientific research, experimentation, or testing

– *Merriam-Webster's Collegiate Dictionary*

In recent years subjective testing methods have been popularised and formalised in the field of audio by the likes of Gabrielsson [Gab79], Toole [Too82, Too85], and Bech [Bec93, Bec94]. In the field of telecommunications and perceptual coding, for example, a variety of subjective evaluation methods are employed as a means of final evaluation. Each standard employs a slightly different method of evaluation that must be strictly followed. These procedures are often very time-consuming and complex if performed manually. For those performing such tests on a regular basis, the manual preparation can become tiresome. Manual preparation can also lead to error especially when tests are large. It is often desirable to employ complex experimental design tools to improve the quality of the experiment. Block designs ([CC92], pp. 439–482) are examples of such tools. In the past computer-based subjective test systems have been designed to perform certain and limited test categories or to slave other audio-visual playback devices.

In my thesis I present "GuineaPig2" (GP2), a software-based subjective testing system. The system is a generic test platform for performing a wide range of subjective audio tests, at the same time eliminating a lot of the complexity of setting up such experiments. A typical GuineaPig2 test setup consists of a Unix based computer, digital audio output hardware, digital to analog converters (DAC) and reproduction transducers, as shown in figure 1.1. The GP2 test system has been developed to run on Silicon Graphics (SGI) platform, running SGI's Unix flavor, IRIX. SGI and IRIX offers a

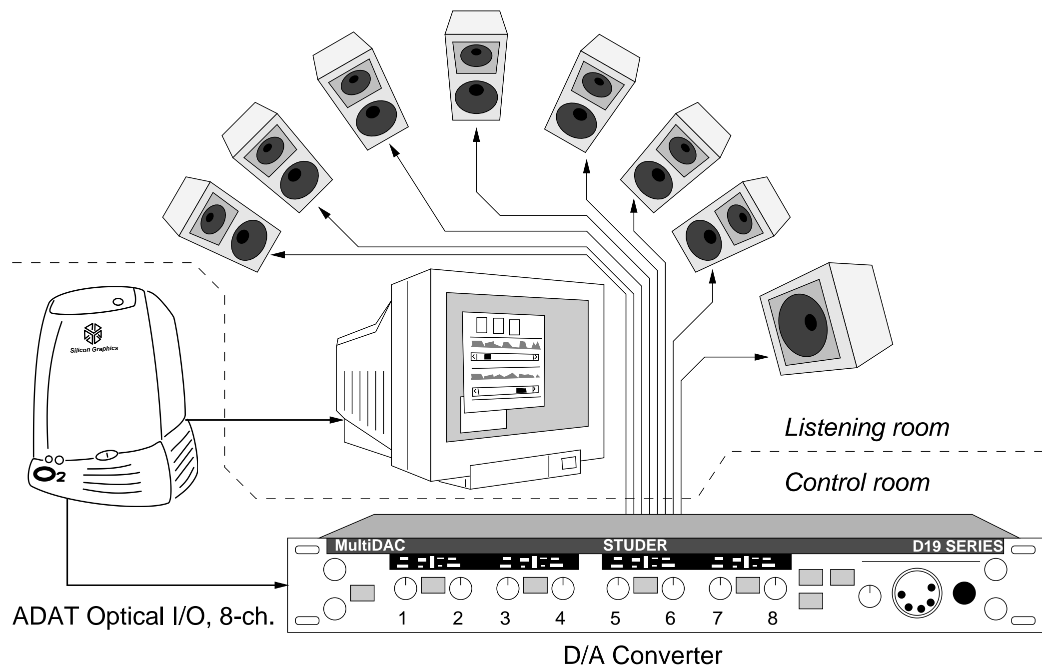

Figure 1.1: An example of a GuineaPig test system setup.

wide range of tools for high quality real-time multichannel audio and video support. For the most part, the system has been programmed in Java, with the time-critical and computation-intensive parts in C.

The GuineaPig2 system is based on the ideas and principles of the original GuineaPig system [HHRF96] (GP1) developed in 1995-1996 by Jussi Hynninen, Keijo Heljanko and Jussi Rinta-Filppula of Helsinki University of Technology as a course work[1] for the Laboratory of Acoustics and Audio Signal Processing. The GP1 ran on a Linux system and was written in Python and C.

In this thesis, the term GuineaPig (GP) is used to refer to the GuineaPig2 system. The original system will be referred to as GP1 if needed.

Structure of the thesis

Section 3 introduces the general structure of the system. The sound player that provides audio output is described in section 4 and section 5 presents the test engine that runs the whole test system. Section 6 describes the user interface panels that are used by the subjects to enter their answers. Finally, section 7 discusses how test data is exported from the test system for analysis.

[1] Tik-76.115 Software Project (Tietojenkäsittelyopin ohjelmatyö), <URL: `http://mordor.cs.hut.fi/tik-76.115/`>

Chapter 2

Listening tests

The following general discussion is based primarily on [Bla83, pp. 5–20], [Kar99, pp. 93–101,199-224], and [HZ99].

2.1 Systems Analysis of the Auditory Experiment

Generally, in an auditory experiment, a sound event with a known structure is presented in a precisely defined way to an experimental subject. The subject then describes, with regard to the particular attributes that are of interest, the auditory events that occur under these conditions. Descriptions can be in many forms, but it is essential that a description allows a quantitative evaluation of the auditory attributes of interest.

The ability to produce well controlled sound events has been aided most by the developement of digital signal processing technology. With the help of computers, precisely controlled sound signals, both simple and complex, can be easily produced.

The auditory events perceived by the test subjects are accessible to the experimenter only indirectly, by way of the subject's own description. The experimenter can never perceive the subject's auditory event directly!

The experimental arrangement may be represented symbolically with a "black box" shown in Fig. 2.1. In the figure, an external sound event s_i enters the experiment subject's auditory system and causes a psychophysical response, the auditory event h_i. The aim is to find the psychophysical function, $h_i = f(s_i)$, that describes the relation between the sound event and corresponding auditory event. However, as the auditory event can not be measured directly, we can use the subject's description b_i about the auditory event h_i, as they both are functions of s_i.

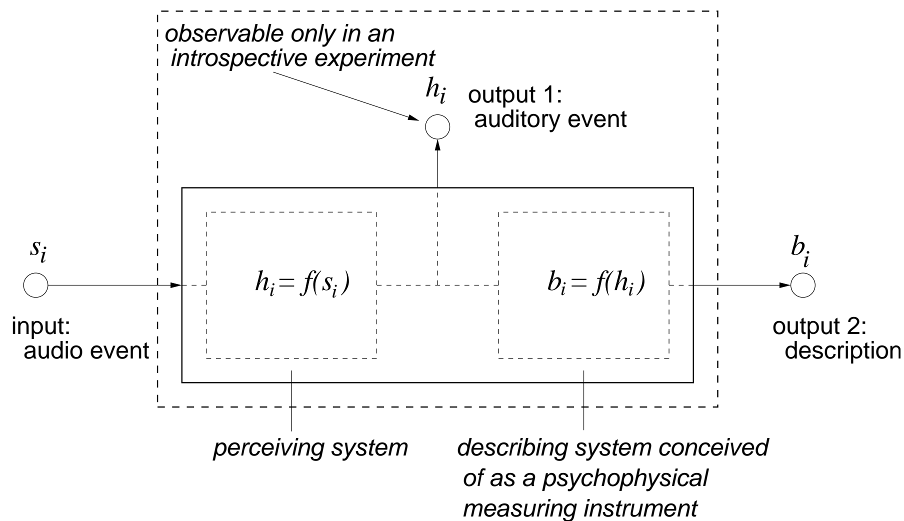

Figure 2.1: A simple schematic representation of the subject of an auditory experiment.

The goal of auditory experiments is to arrive at quantitative statements about these functions. If there are to be quantitative statements, measurements must be taken. The function $b_i = f(h_i)$ can be viewed as a "measurement instrument". For quantization of the measurement, various "scales" are used. The type of scale used in a measurement procedure determines which mathematical operations may be used in interpreting the measured results.

2.1.1 Scaling

In the theory of measurement, distinctions are made between scales of different complexity, namely, nominal, ordinal, interval, and ratio scales. These scales differ according to which of the following properties of numbers are applied: identity (each number is identical only with itself), rank order (the numerals are arranged in a specified order), and additivity (rules for addition are defined).

Nominal scale or category the signal or an attribute of the signal belongs to a certain class. Classes are usually labelled with numbers or symbols. For example, a sample is to be classified as either a *male or female* voice.

Ordinal scale the audio samples are arranged into an order based on some attribute of the samples, such as loudest sample first.

Interval scale Numeric scale where the differences of classes have been quantified.

Ratio Same as the interval scale, except that ratio is quantified instead of difference. For example, the sample is twice as loud or half as loud as another.

Multidimensional scale Auditory events are projected onto a multidimensional space, or on multiple scales. The listener quantifies each event on multiple scales or semantic differentials.

Numeric scaling is usually using the $p_i \in [1,5]$ or $p_i \in [1,10]$ scales. The most common scale is the MOS (Mean Opinion Score) which is used in, for example, the evaluation of sound quality in speech and voice coding.

A scale can also be defined with the help of a pair of conceptual opposites (a *semantic differential*), such as '*soft* $\leftrightarrow$ *hard*' or '*low* $\leftrightarrow$ *high*'. For example, the scale $p_i \in [-5,5]$ could be used (zero marks the neutral point).

The most important methods in auditory experiments are based on nominal and ordinal judgements. These methods are especially well adapted to determine thresholds of perceptibility, difference thresholds, and points of perceptual equality.

In GP, the scaling and classification of sensation has been reduced to the following three (abstract) basic classes. Different scales in GP are merely special cases of these three types.

Multiple choices This component presents a set (defined by the tester) of discrete choices shown with symbols (labels). The symbol selected by the listened is the answer. Used primarily for nominal scales.

Numeric scale Provides a slider that represents a section of one-dimensional line defined by the tester. As the answer, the position on the line selected by the listener (a scalar value). The limits of the scale (endpoints of the line) and the resolution of the scale can be selected freely, allowing any numeric scale, for example a MOS-scale, to be easily implemented.

Ordering Presents a set of symbols (labels) that are arranged by the listener into the order requested in the question (ordinal scale). The symbols usually represent the samples that are compared. The ordering selected by the listener (an ordered set) is given as the answer.

The scale can be considered as the visualisation of a person's perception of the quality space and thus its presentation and interpretation are very important. Often unidimentional rating scales are used (for example, the MOS scale) but they may not always be sufficiently informative. What is actually being rated is the overall perception of a multidimensional event. Many authors have studied this aspect and the use of multidimensional scales is gaining increased support [Bec99, BR99]. Tests using multidi-

mensional scaling can be implemented easily with GP: the simplest way is by adding additional scaling components for the UI of the test subjects.

When considering the use of a scale, some of the following issues should be considered in depth and with care:

- type (ordinal, interval, ratio, category (or nominal)),
- resolution,
- anchors (name and placement),
- absolute scale length,
- uni- or multi-dimensional.

Many of these aspects are supported by the GP test system and are discussed in sections 5.8 and 6.3.1.

For a full discussion on the use of scales, the interested reader is referred to [NB94, pp.11–81], [MCC91, pp. 53–56], [SS93, pp. 66–94].

2.1.2 An example of a traditional listening test

This section given an example of a hypothetical listening test run without a fully computer-based testing system.

The experimenter collects and prepares the listening material for the test, then edits, and finally pre-records it to a tape. In the test, the subject(s) listen to the tape and record their answers, often by writing answers to a form. After the test session, the forms are returned to the experimenter who then has to enter the form data into a computer (analysis is probably done with statistical packages that run on computers).

With a tape, the presentation order is fixed. If badly designed, it can introduce bias to the results. A different presentation order for each subject can reduce the change of that but this also means that a new tape would have to be edited for each subject. Also, adaptive tests can not be pre-recorded, as the response by the subject determines the next operation.

The data entry is a tedious and boring job and can be very time-consuming. Errors can be introduced due to, for example: mis-interpreting the subject's answers (for example, due to bad handwriting), incorrect use of answering form (by the subject), or typing errors by the operator. Automatic form readers can speed up the data entry

and increase reliability. However, they usually require extra hardware and machines are not infallible either.[1]

2.1.3 Some applications for listening tests

This section describes some research areas where listening test are used.

Commercial applications

In *commercial applications* or *research & development* (R&D), the aim is usually to select, according to some criteria, an "optimal" choice, for example, the best cost/quality-ratio. Many factors can be used as the cost, such as:

- Cost of production.
- Amount of data.
- Average or maximum bit-rate.
- Computational load. Complex algorithms need more powerful processors which are more expensive and may also need more power to work.
- Noise level produced by a machine.

Usually many or all of these factors must be considered when evaluating the choices.

As a final measure for quality could be, for example:

- Percentage of test subjects that can not tell a difference between an original (unprocessed) and processed signal.
- An average grade given to a system based on some criteria. For example, how distracting the distortions in a coded signal are.
- How much noise a codec algorithm withstands before signal (such as a speech signal) becomes unintelligble.

[1] Consider the U.S. precidential elections in 2000, the state of Florida.

Psychoacoustical research

In *psychoacoustical research* (viewed here as more theoretical *basic research* where the aim is not so much to get just commercial advantage (as in R&D)) the aim is to learn more about a topic of interest related to hearing and audiology. Listening tests are used to collect information about some phenomena to formulate new theories or laws about them. Also, theories are tested or verified with listening tests.

Clinical hearing diagnostics

In *clinical hearing research* listening tests can be used as medical diagnostic tool, for example, for the assessment of hearing loss caused by *occupational noise*. The reader may personally be familiar with the *audiogram*, a graph showing hearing loss as a function of frequency as measured by an audiometer (instrument for measuring hearing sensitivity).

2.1.4 Other considerations when conducting listening tests

When conducting listening tests, it is important to eliminate any unwanted distractions or cues (acoustical, visual, etc.) from the *listening environment* that can cause errors and skew the final results. *Environmental or background noise* can be caused by many sources, such as air conditioning, or the hum from electronical equipment.

Computers very rarely work quietly enough (fan and harddisk noise) to allow them to be placed near the listeners. Computers could be placed outside the listening room, or in a special separate control room (see also Fig. 1.1).

The sound field of the testing environment can be most accurately controlled in an *anechoic chamber*, a soundproof room essentially free from reverberation. When the absence of reflection is desirable, anechoic chambers are generally used. For some tests, an *echo chamber* could be appropiate. Echo chamber is a highly reverberant room with reflecting surfaces that can be used when a *diffuse sound field* is wanted.

When choosing the *panel of test subjects* and test setup, one must consider what is the indented use of the results. For example, to find out how the “general public” finds the sound quality of a particular product, one would probably select a random selection of average people (buyers, consumers) as the panel of test subjects. When aiming for the best possible sound quality (HiFi, high fidelity), well trained and analytical listen-

Score	Quality of the speech	Effort required to understand the meanings of the sentences	Loudness preference
5	Excellent	No effort required	Much louder than preferred
4	Good	Attention necessary	Louder than preferred
3	Fair	Moderate effort required	Preferred
2	Poor	Considerable effort required	Quieter than preferred
1	Bad	No meaning understood with any feasible effort	Much quieter than preferred

Table 2.1: Opinion scales recommended by the ITU-T: *listening-quality scale* (mean listening-quality opinion score, MOS), *listening-effort scale* (MOS_{LE}), and *loudness-preference scale* (MOS_{LP}).

ers[2] who can detect even the slightest errors in the sound quality, should be used.

2.2 Subjective test paradigms

This section, adapted from the GP AES paper preprint [HZ99], introduces a few of the common test paradigms and methods employed within the audio industry that are supported by the GP platform.

2.2.1 Single stimulus

A simple listening test is the *single stimulus* (SS), in which a single stimulus is presented without context to a listener. The listener then gives a grade to the stimulus, for example, on how "clear" the stimulus sounds. Grading is performed on an interval scale without any references. Typically, this method is only applied when it is not possible to compare systems or samples simultaneously for example when comparing the sound quality of two rooms. The method is employed in the telecommunications industry and referred to as the MOS (mean opinion score) test [IT96a]. Table 2.1 illustrates the opinion scales recommended by the ITU-T.

Typically simple means and error variance measures are employed for telecommunications tests. Analysis of variance (ANOVA) model [Gab79] can be applied for more thorough analysis.

For the listener, this method is usually difficult. Human hearing is not very good for analyzing absolute properties, such as the frequency. For example, the *perfect pitch* is relatively rare[3].

[2] So called people with "golden ears".
[3] Expect among musicians and audio professionals.

2.2.2 Paired comparison methods

Listening tests are usually implemented as *paired comparisons* where two or more stimuli are compared. This is easier for the listener as human hearing is much more accurate for detecting differences of stimuli than absolute values. For example, it is much easier (at least for a layperson) to tell which one of two signals has higher frequency or pitch than to tell the frequency of one signal or to name the note it was.

In paired comparison tests, usually each test case contains two or three stimuli that are compared against each other. For example, when comparing codecs, one of the stimuli is an original, unprocessed signal. The other stimulus is the same signal as the first signal but processed using a codec. Both signals are presented to the listener and the listener is to judge wheather there is an audiable difference between the original signal and the processed signal. Another type of judgement is to give a grade for the processed sample based on how much or little there are distortions compared to the original. The first kind of the judgements would more probably be aimed for evaluating high-end audio coding where transparency[4] is the goal, such as movie surround audio formats (SDDS, DTS, Dolby Digital). The second kind of judgement could be for evaluating cellular-phone codecs where the goal would be to select a codec so that distortions caused by the codec are still tolerable or speech is still intelligible.

Several variations of this theme are employed in psychoacoustical testing.

Paired comparison

The basic pair comparison method [Dav63] simply consists of presenting the listener with two samples from which the superior/inferior is to be chosen. In practice this is a simple task which is easy for listeners to learn and comprehend, even when unfamiliar with subjective testing.

The method of pair comparisons has been used for over a century and is common in the field of psychometrics. The method has widespread acceptance due to it's simplicity and well proven analysis methods. The method is significantly slower than the single stimulus, particularly if full permutation set is employed. Incomplete block design methods can be employed to improve the efficiency whilst not loosing interaction information ([CC92, pp. 439–482]).

In GP, this method this method is referred as a *A/B*-test. The test is a simple comparison of two samples with a multiple-choice component (two choices, A and B) acting

[4]No audiable difference between original and processed signal.

Score	Opinion
3	Much Better
2	Better
1	Slightly Better
0	About the Same
-1	Slightly Worse
-2	Worse
-3	Much Worse

Table 2.2: The scale used in the Comparison Category Rating (CCR) method with the question as follows: *"The Quality of the Second Compared to the Quality of the First."*

as the answering component.

Scaled paired comparison

This method is an extension to the paired comparison method. For grading, it uses two (typically identical) interval scales, one for each sample. The MOS or degradation mean opinion score (DMOS) scales [IT96a] are often employed.

This type of test is common in the field of telecommunications and is also employed in audio testing [Zac98]. This method is also found in the field of visual testing where it is referred to as the *double stimulus continuous quality* method *Recommendation ITU-R BT.500-8* [IR98b].

When implemented with GP, the method (referred as *A/B Scale*) is similar as the basic paired comparison. The difference is that the single multiple-choice grading component is replaced by two appropiate scale-grading components.

An alternative method is to employ the comparative mean opinion score (CMOS) or comparison category rating (CCR) scale as found in *Recommendation ITU-T P.800* [IT96a]. The CCR scale is illustrated in table 2.2. In this case there is only a single scale employed to rate both samples. The scale allows for both positive and negative ratings thus allowing for comparison

ABX method

The ABX method [Cla82, Cla91] consists of presenting the listener with three samples, A, B and X. The aim is for the listener to select which sample of A or B is identical to X.

The benefit of this method is that the task is simple and there is a clear reference to compare against. In addition, the test allows for a very quick and easy assessment of listener reliability, by simply studying the number of correct answers.

The implementational difference between this test (referred as *A/B/X*) and the A/B test is simple: instead of two samples, three samples are compared. However in actuality, only two different samples are compared as one of samples A or B must be the same as sample X. For answering, a multiple choice question as in A/B test is used, only the question needs to be changed.

Triple stimulus hidden reference

The method of *double-blind triple stimulus with hidden reference* as standardised in *Recommendation ITU-R BS.1116-1* [IR97], is a practical extension of the ABX method to include interval scales.

This method provides the listener with three samples: Ref, A and B. Compared to the ABX method, it employs the ITU-R 5-point impairment scale. The listener must first select which sample is different to the reference and grade this sample only. The other sample must be given the maximum grade (i.e. 5).

This method is aimed at so called *expert* listeners who have been trained. The method benefits from the use of a reference and grading scales, providing detailed information on the tested systems. The hidden reference method also provides for a rapid check on listener reliability. The method is not intended for fast and dirty testing, but is well suited to high quality, small impairment tests.

A t-test is employed to post-select suitable listeners for the main analysis of results which is performed with an ANOVA.

2.2.3 The rank order paradigm

The rank order method ([Dav63, pp. 104–130]), ([LH98, pp. 691–700]) can be employed when three or more samples are to be compared. The most simple form of the rank order test is that of the paired comparison, where 2 samples are compared and a selection made upon which is found to be superior (or inferior). Whilst this test type is common, the more general rank order procedure does not find much favor. The procedure consists of asking the subject to arrange the samples in order of intensity or degree of the tested attribute.

This procedure benefits from its simplicity and the lack of assumptions:

- the task is very simple to comprehend and requires little training or instruction,
- there is no need to understand or interpret a scale,

- there are few assumptions regarding the data type and the distribution of the data. Data need not be normally distributed or there need not be any assumptions regarding the perceptual separation between samples,
- a large number of samples can be considered in a relatively brief test. Whilst certain authors [Mil56] consider there are some perceptual limitations in this respect, the test method itself is not restricted,
- complete or incomplete block designs may be employed,
- data handling and analysis are simple.

To perform such a test a few assumptions are required:

- All stimuli must be evaluated prior to judgement. This may thus lead to sensory fatigue when many samples are tested,
- the direction of ranking must be specified (i.e. which sample is better).

In practice, whilst this test can provide some knowledge of the rank order of samples, it is not possible to provide an absolute rating of quality. This is perhaps one of the reasons why this method lacks favor.

2.2.4 Adaptive methods

In *adaptive tests*, usually a parameter, such as signal gain or a filter parameter, is adjusted and a threshold value is looked for. For example, the gain of the first signal is adjusted so that it is just audible when a second signal is played at the same time. Another case would be to adjust one signal until it is perceived as loud as a second (the reference) signal. The adjustment can be carried out by the experimenter, the subject, or automatically. Methods using this procedure are also called *methods of adjustment*.

The two-alternative-forced-choice paradigm

The category of two-alternative-forced-choice (TAFC or 2AFC) methods is widely employed in psychometrics for the use of parameter or threshold estimation and are considered a subset of the sequential experiment class of procedures. Various forms of TAFC procedures exist which are described in detail in Levitt's paper [Lev71], considered a definitive summary of the topic.

The simple up-down or staircase method has been implemented to estimate the equivalence of two stimuli. However, the test may also be configured to perform absolute threshold experiments with only one stimulus (e.g. audiometry experiments).

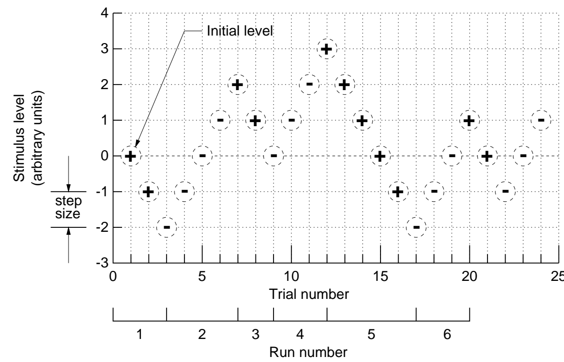

Figure 2.2: Example of simple up-down or staircase procedure to estimate the equivalence of two stimuli.

The method basically consists of presenting the listener sequentially with the reference and the test stimuli. The listener must judge whether or not the test stimulus is, say, louder, for example. Based upon a positive response, the following stimulus level is decreased (or increased following a negative response). This procedure continues until 6-8 reversals have occurred. A run is defined as the series of steps in one direction. Based upon this data, the overall parameter estimation can be made from estimating the 50% level (X_{50}), for example. Figure 2.2 illustrates the procedure.

It should be noted that to ensure that this method is correctly conducted a few rules must be applied, which have been implemented within the GP test system, including the:

- starting level,
- number of runs,
- reversal rules,
- stimuli step size.

The method is relatively efficient, but careful choice of step size is required to ensure efficiency and quality of results.

Chapter 3

System architecture

The general structure of the GP system is shown in Figure 3.1. The main modules consist of the *sound player*, *subject user interfaces*, *test engine*, *configuration files*, and *results processing*. The GP system is written in Java except the sound player that is partly written in C. Although there are no plans to port GP to other systems, the subject user interfaces benefit from Java's cross-platform portability. For more information about Java, see: [Suna, Sunb, GM96, AGH00, GJGB00, Kra96].

The **sound player** (Sec. 4) handles the audio output of the GP system. The sound player is written in C to get access to real-time performance and SG's digital media libraries. The sound player is controlled with a Java front-end module that the rest of the GP system uses.

Subjects use a graphical **subject user interfaces** (Sec. 6) to give their answers and to select samples to play. GP is designed so that subject user interface windows can be opened on many different platforms. In principle, any networked Java-compatible terminal can be used as the subject's answering panel.

The **test "engine"** (Sec. 5) runs and manages the test. It reads the information needed to run a test from *test configuration files*, then initializes the test and subject UIs. In general, it plays the samples selected by the experimenter for comparison in fixed sequence or selected by the subject, collects the answers of the subjects, and records them. The test engine allows multiple subjects to give answers at the same time. Also, multiple instances (sessions) of the same test can be performed in parallel independently of each other.

The **results processing** (Sec. 7) takes the raw result files generated by the test engine and converts them to human and computer readable tabular format for analysis by other tools. The GP system itself performs no analysis of the data.

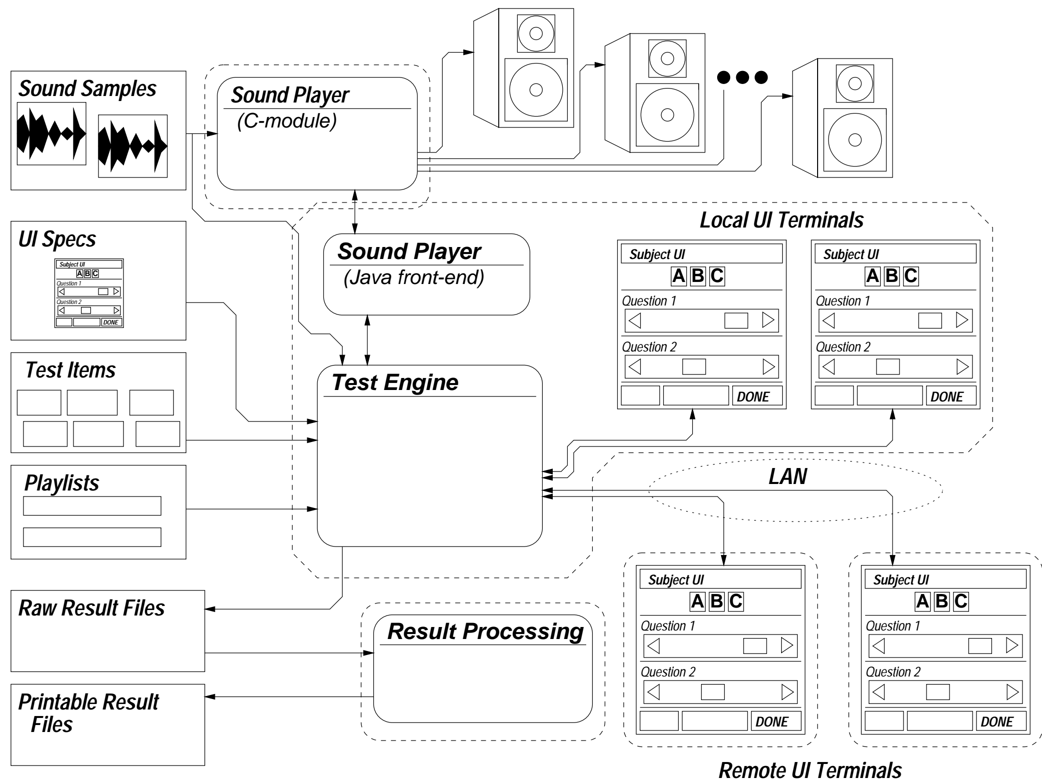

Figure 3.1: Modules of the GuineaPig system. Dashed lines indicate process boundaries.

3.1 Test configuration files

Tests are configured with simple text files that use standard Java properties-format and objects (Java class `java.util.Properties`) as their base.

The **test configuration file** defines the principal test properties, such as the test type, sample playback sequence (Sec. 5.3), time limits (Sec. 5.4) and sound player properties (sample rate, number of channels, etc.). Also includes the names of additional configuration files (see below).

The **test items file** contains the properties of the test items (Sec. 5.1) that are presented to the subject during the test session. A test item defines the samples (referred to with a *sample ID*, see below) that are compared in that case. In addition, *playlists* (Sec. 5.2) specify the order the test items are presented to the subject.

The **sample list** associates *sample IDs* that the GP system uses to refer to samples to the actual audio file names. Additional properties can be also included.

The **user interface file** defines the properties of the answering panel (Sec. 6) that the subject uses to give the answers. It defines the UI components that are to be used

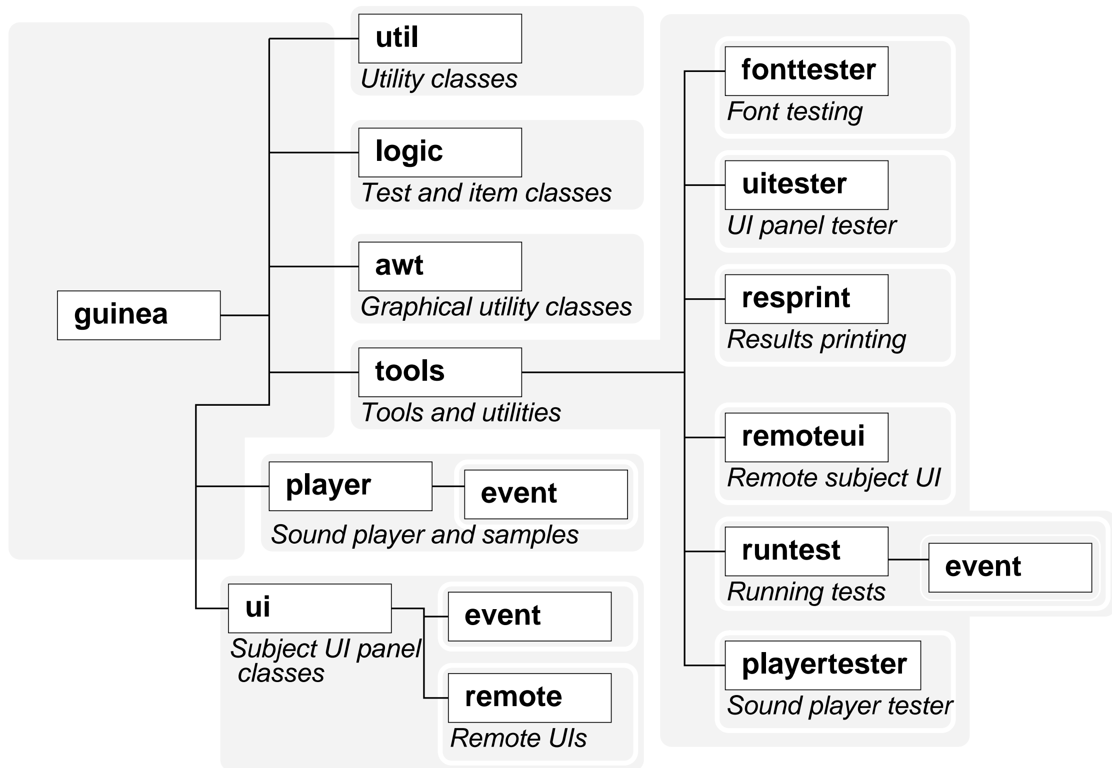

Figure 3.2: Java package hierarchy of the GP system.

(questions, controls and monitors) and their properties.

The **results output configuration file** allows customizing the output format of the *results processing* (Sec. 7). It can be used to select which information to print and the order of the fields. Custom formatting modules can be added to format special types of answers in a desired way.

3.2 Java-package hierarchy

Figure 3.2 shows the hierarchy of the Java packages the GP system consists of. It resembles the structure of the GP system shown in Fig. 3.1.

The `guinea.logic` package corresponds to the test engine. Test items, test types, sequences, etc. are defined there.

Subject UI panel and UI components are defined in the `guinea.ui` package. The `guinea.ui.remote` package contains classes used for remote terminals. UI event and listener classes are contained in the `guinea.ui.event` package.

Sound players, samples, and related classes are defined in the `guinea.player` package.

The `guinea.player.event` package contains sound player event and listener classes.

The `guinea.tools` package contains the tools and utilities that are used to create and test test configuration files, running a test, and for processing the results files to readable format.

The `guinea.util` package contains non-graphical utility classes used by other modules. The `guinea.awt` package contains UI utility classes used by the graphical tools and the UI panels.

Chapter 4

Sound player

The audio output of the GP system is handled by the *sound player* (SP). It is a simple program that plays sound samples directly from the hard disk and mixes them together in real time. By playing samples directly from hard disk, the memory usage of the player is quite small even with very large samples. Figure 4.1 shows the general structure of the SP in a simple configuration.

The player (Sec. 4.9) is written in C to gain real-time performance and to use SGI digital media libraries. The player is controlled by a higher-level Java front-end module (Sec. 4.10) that is used by the GP system.

The GP's player itself is not locked to any sample rate, number of channels, device or interface type or sound file format. It relies on SGI's standard audio libraries for most such things.

4.1 Audio output features

The sound player uses *Silicon Graphics Audio Library (AL)* that provides a uniform, device-independent programming interface to real-time audio I/O on all SGI workstations. When additional hardware options are available, the player can use them without modifications.

4.1.1 Audio output devices and interfaces

A variety of audio output interfaces are available, from regular analog stereo outputs to optical digital eight-channel 24-bit ADAT outputs. The selection of interfaces that are available depends on the model of the workstation and option cards that are used.

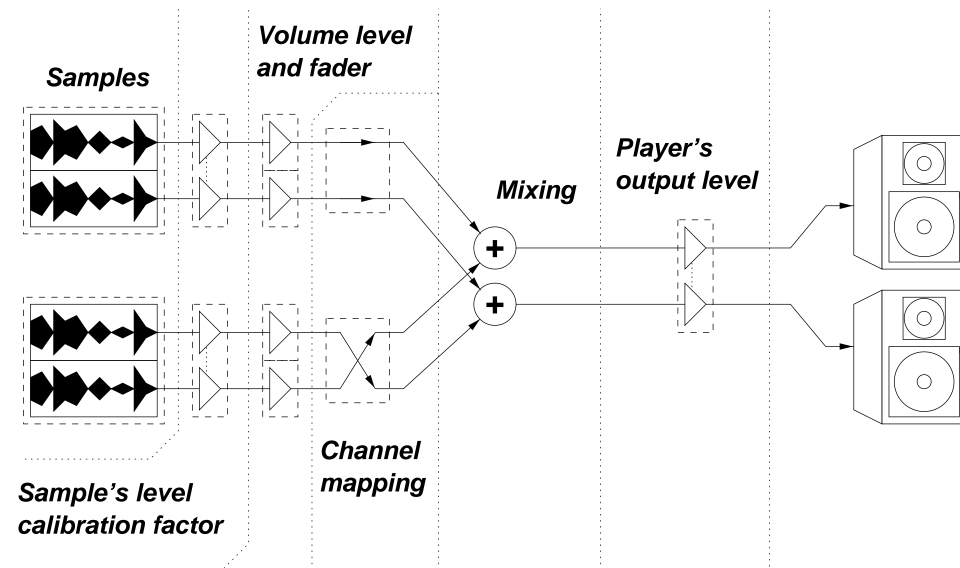

Figure 4.1: Structure of the sound player. Two stereo samples are played and mixed to stereo output. The lower sample has its left and right channels swapped.

Generally, at least analog stereo output is provided.

The GP system was developed for use with SGI O2 and Octane workstations with multichannel output in mind (more than two channel output) but any SGI workstation[1] can be used. The Octane systems has built-in analog stereo outputs as well as digital ADAT and AES audio output interfaces. The O2 system has two analog stereo output interfaces. In addition, a ADAT audio option card is used with the O2 that provides digital ADAT and AES output (the same as in Octane).

4.1.2 Support for multiple output devices

Multiple output devices can be used simultaneously to simulate a "wider" output port than is possible with only a single output device. Multiple[2] output devices are automatically syncronized so that their output signal arrives at the same time to the outputs of the machine. Several interfaces can be kept in sync more easily when they all are in the control of a single player process. *Silicon Graphics Digital Media libraries (DM)* provide methods that are used to keep several devices in sync.

Combining multiple devices looks like a single output port to the application. For example, when using two eight-channel ADAT interfaces, the application sees them as a single 16-channel audio port. Channels 1–8 would be sent to first ADAT interface, and

[1]A SGI workstation with IRIX 6.3 as the minimum.

[2]The maximum number of devices that can be used simultaneously is currently four. It can be increased easily by changing a definition in player's configuration file and recompiling.

channels 9–16 would be sent to the second interface. This port can then be sectioned into smaller virtual players (see Sec. 4.3).

4.1.3 Number of output channels

The maximum number of output channels depends on the number of output devices that are used simultaneously and the number of channels they support. Currently the player is configured so that four devices can be used simultaneusly. With four 8-channel ADAT devices, the maximum number of channels is 32. Also, the maximum number of channels a sample can have is 32 channels. If needed, the limits can be increased by editing the player configuration files and recompiling.

4.1.4 Sample rates

The GP sound player does not have any fixed limits on the sample rates it can use. In principle any sample rate can be used. In practice, the limits of the AL and the audio hardware limit the available sample rates. The O2 supports all sample rates from 4kHz to 48kHz with 1Hz resolution (nearly arbitrary sample rates on Octane). Some devices may not offer all possible sample rates but at least the most common sample rates (32kHz, 44.1kHz, and 48kHz) are usually supported. Digital output interfaces are generally limited to 32kHz, 44.1kHz, and 48kHz sample rates.

4.1.5 Audio precision

The AL uses internally 24-bit integer precision when processing audio data. The actual precision of the output signal may be more limited. Digital to analog converters (DACs) on SGI workstations usually use 16–18 bit precision. With digital output interfaces, full 24-bit output is possible.

The GP sound player uses floating point format when processing audio data. The data processing in the sound player uses a lot of multiplication operations which are faster with floating point that with integers. Newer MIPS processors (like the R5000 and R10000 used in O2 and Octane) are very well suited for floating-point processing. They also include some common operations used in signal processing, such as the "multiply-and-add" operation, which will be useful if real-time filtering is later added to the sound player.

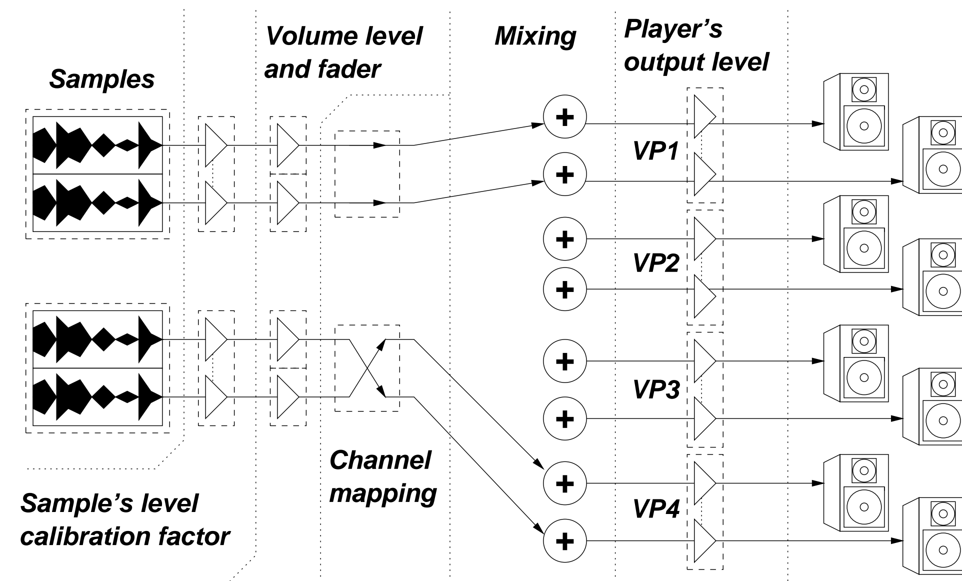

Figure 4.2: Example of a virtual player configuration: A sound player with an eight-channel output is divided into four virtual stereo players (VP1-4).

4.2 Audio file formats

The GP sound player itself does not include any methods for reading or decoding any particular audio file format. Instead, the sound player uses Silicon Graphics *Audio File Library (AF)* for reading audio files. The AF provides a uniform programming interface for reading and writing many audio file formats[3]. Also, AL can transparently read compressed[4] files. As the output, the AL provides the audio data as interleaved linear PCM data with selected precision.

The AIFF-C (Extended AIFF-C standard) is the recommended audio file format for use with GP (SGI has adopted AIFF-C as its default digital audio file format). Using a little-endian file format (such as WAVE) requires on-the-fly conversion from little-endian byte-order to big-endian that increases processing overhead, lowering the performance of the system.

4.3 Virtual players

Virtual players (VP) allow partitioning the output channels of the player into smaller

[3] Support for such file formats as AIFF-C, AIFF, WAVE, and MPEG-1 audio (layers 1 and 2) is included in AL.

[4] Realtime codecs for mu-law/A-law, CCITT G.722 ADPCM, GSM 06.10, MPEG-1 audio (layers 1 and 2), and others are included in AL.

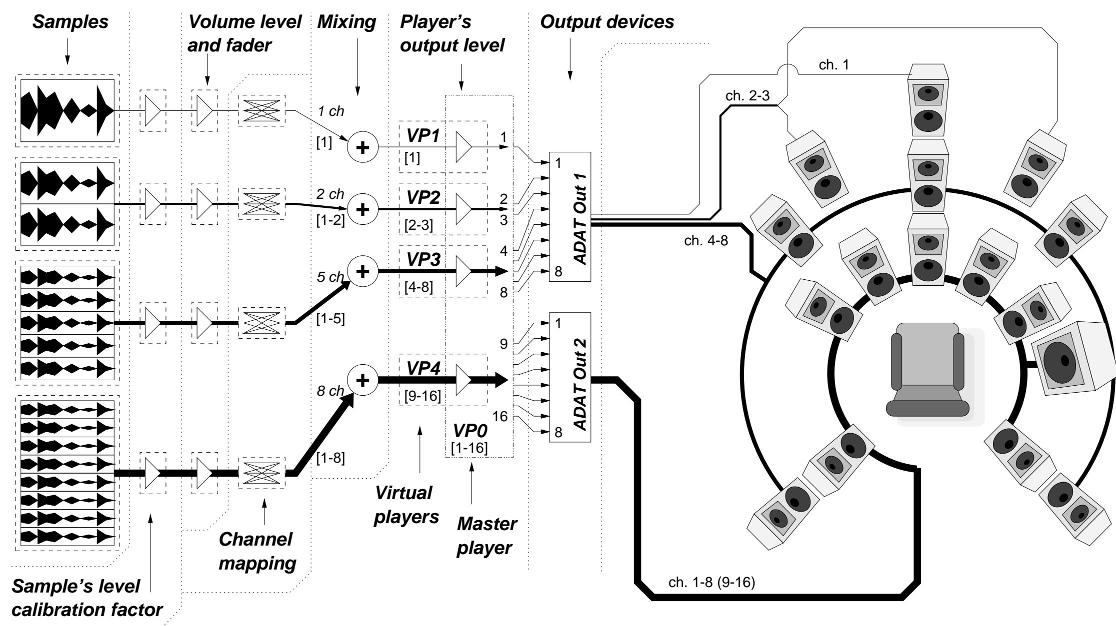

Figure 4.3: A more sophisticated player configuration with multiple output devices and virtual players. A sound player with 16-channel output (using two ADAT output devices) is divided into four virtual players (VP1-4) of different sizes.

sections. A virtual player acts in the same way as the original player. The outputs of the samples that are attached to a VP are automatically redirected to the channels of the original port that are assigned to the selected VP. Also, the output level of each VP can be set independently.

Figure 4.2 shows an example of a possible virtual player configuration. A player with a single eight-channel ADAT output port is divided into four virtual stereo players. Two stereo samples are played. The output of the upper sample goes to VP1, the output of the lower sample goes to VP4. Also, the lower sample has its left and right channels swapped.

Virtual players can also be used when multiple output devices (Sec. 4.1.2) are used. Virtual player configuration is done exactly the same way as with a single output device (as seen in previous example), since using several output devices look like a single output device to the application.

With multiple output devices it is easy to create tests that require output with more channels than a single output device can provide. Also, hardware tests could be performed by creating several virtual players which correspond to, for example, different sets of loudspeakers.

Figure 4.3 shows a more sophisticated test configuration. Two eight-channel ADAT output devices are combined into a 16-channel player. The player is then divided into

four virtual players with different number of channels: a mono VP, a stereo VP, a 5.0 VP, and a 7.1 VP configuration. The configuration can be used, for example, to compare different mixes with different number of channels.

4.4 Volume levels

All the volume levels used in the GP system are relative. They do not match the standard absolute volume scale. A more precise term for the volume levels used in the GP system would be *gain* or *gain factor*. In GP, both volume level and gain are considered to mean the same as gain.

In GP, volume levels (or gains) can be expressed using *linear*, *decibel*, or *percent* scale.

Linear scale A gain or volume level in linear scale is the factor that the signal is multiplied with. Internally the sound player uses linear scale, other scales are automatically converted to it. A level of 1.0 in linear scale means that the signal level is unchanged. A level of 2.0 (linear) doubles the amplitude of the signal, and a level of 0.0 mutes the signal.

Percent scale The percent scale is the same as linear scale multipled by 100 to get a scale from 0 to 100. The level 100% is equivalent to 1.0 (linear scale).

Decibel scale The decibel scale in GP is relative to the *full scale* of the signal scale. It is essentially the signal gain in logarithmic scale. The "decibel, full scale" is shortened as *dBfs*. The equivalent of gain 1.0 (linear) in decibel scale is 0.0dBfs. A gain of -6dBfs means that the signal level is 6 dB lower that the unscaled level (equivalent to 0.5 in linear scale). In GP, the term dB is considered to mean the same as dBfs.

The equations below in 4.1 relate the different scales to each other:

$$\begin{aligned} v_{pct} &= 100\, v_{lin} \\ v_{dBfs} &= 20\, \log v_{lin} \\ v_{lin} &= 10^{(v_{dBfs}/20)} \end{aligned} \tag{4.1}$$

Volume controls or gains are applied to signals by scaling the signal data by multiplying with the volume level or gain in linear scale.

The GP system has been designed for use with digital audio output, therefore GP does not set the gains of analog outputs. They should be set manually to appropiate levels

v_{lin}	*Linear*	2.00	1.00	0.10	0.00
v_{pct}	*Percent (%)*	200	100	10.0	0.00
v_{dBfs}	*Decibel (dBfs)*	$+6.02$	0.00	-20.0	$-\infty$

Table 4.1: Examples of equal gain levels with different scales.

when tests are performed. Similarly, the gains of the external DAC's analog outputs should be checked manually. Digital outputs on SGI do not have any gain settings for digital audio output. In either case, the output level of the player should be 0dBfs (1.0 linear) at maximum.

The GP provides three distinct volume level controls: *player output volume*, *sample volume*, and a *sample level calibration factor*.

4.4.1 Player output level

The player has a digital *output level* adjustment that is applied to the output signal after samples have been mixed together, before the audio signal goes out[5]. The output level of each virtual player (Sec.4.3) can be set independently of other VPs. In the GP system, the player output level is set to the MCL level (Sec. 5.5) of the test session.

4.4.2 Sample volume level

Each sample has a fader that is used to control the volume level of the sample. The level of each channel can be controlled independently[6]. The level change can be immediate or a fade can be applied. The length of the fade is selectable (unlimited) and the type of the fade can be either amplitude-linear or decibel-linear. Faders are also used for cross-fades with parallel switching (Sec. 5.3).

4.4.3 Sample volume level calibration factor

Each sample also has a static *sample level calibration factor* than can be used, for example, to calibrate the levels of a set of samples so that their levels are aligned. By setting the calibration factor for a sample, editing or scaling the sample files can be avoided. The same factor is applied to all channels of the sample. Calibration level can be changed at any time but no fader is provided.

[5] More precisely, before the audio data is written to the AL.

[6] The ability of setting the levels of individual channels of a sample is not used in the tests included in GP system. In all tests, same level is applied to all channels of a sample.

4.5 Mixing

The signals from any number of samples are mixed together to form the output signal. Each channel of a sample can be mapped to any one output channel of the player (or virtual player) with their own gain factor (see *channel mapping* in Fig. 4.1).

Channel mapping also allows directing the output of a mono sample to either left or right channel of the output (assuming stereo output), for example. A very simple downmix of a 5.1 signal to stereo could also be performed.

During the project, there was no need for a more complex mixing matrix so a simple mixer was implemented. When full matrix will become necessary, upgrading the channel mapping to a full mixing matrix would be relatively straightforward.

4.6 Drop-out handling

A *drop-out* or *framedrop* occurs when the player fails to write audio data into the AL output queue in time before it is supposed to go out. The drop-out is heard as a glitch in the output sound signal as zeros are output when no data is available in output audio queue.

When the player detects a drop-out, it generates a diagnostics event with more information about the drop-out. The event records when the signal was supposed to go out and the length of the drop-out in sample frames. The GP system reports drop-outs by printing event details to the console and test log file.

To compensate the drop-out, the player skips equivalent length of audio data (how many sample frames the device is behind schedule) from the start of the audio frame (buffer) so that the positions of the SGI audio output stream and the player's calculated output stream should match when next frame is written. If multiple audio devices are used at the same time, the same is done for each device.

4.7 Delay and latency

The *delay and latency* of player's operations can be measured and adjusted. By default the player calculates the output signal in blocks of 4096 sample frames (approx. 93ms when using 44.1kHz sample rate). The calculation is also double-buffered to lower the risk of a drop-out caused by random CPU load peaks. However, this doubles the delay and with other overheads the delay will be a little over 200ms. The delay can

be shortened by decreasing the length of the audio block with a test configuration parameter.

The delay affects operations that are to take effect immediately, for example, the subject presses a button to play a sample. If an operation is scheduled in advance, some operations can be made to take effect at some point of time with sample frame accuracy. So far only the starting time of the sample can be scheduled in advance. It is used when a fixed sample sequence is played to schedule the start of the next sample.

Most operations return a sample-frame time stamp that tells the time when the effect of the operation will arrive at the electrical output on the machine. This time stamp can be used to calculate the latency of the operation or the actual time and date of the event.

4.8 Operations in synchrony

Several samples can be started (or stopped) so that they start (or stop) at exactly the same time. Cross-fade operation also operates on several samples at the same time so that the faders' level slopes start at the same time for all samples.

Also, multiple audio devices are syncronized when the player starts so that their output should be in syncronity (see also Sec. 4.1.2 and 4.6).

4.9 Sound player C-module (sndplay)

The `sndplay` program handles the audio output of the GP system. It is a small program written in C that reads multiple sound samples directly from the hard disk and mixes them together in real time.

The sound generation, audio output, receiving, and sending messages run asynchronically using POSIX threads. Critical sections are protected with POSIX mutexes. In addition, POSIX semaphores are used in queues.

The sndplay is invoked by the GP Java module. Player parameters, such as audio device names and sample rate, are given as command line parameters.

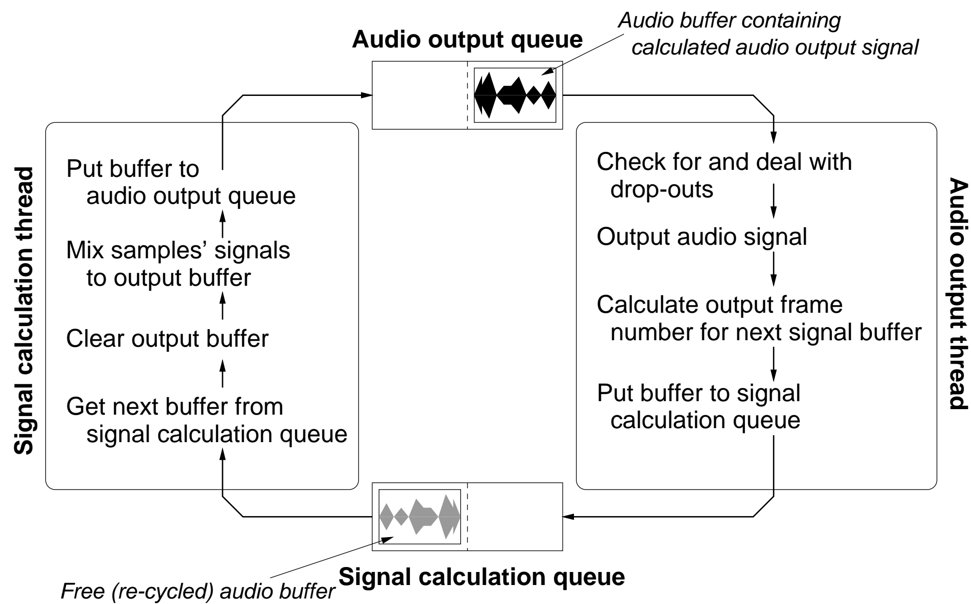

Figure 4.4: The player's sound signal generation loop.

4.9.1 Initialization

The sound player decodes the player parameters given to the program as command line options: the names of the audio devices, sample rate, and number of channels. A real-time priority is set for the player if possible[7].

Audio devices are then initialized. If multiple audio devices are used, the audio output of the devices are synchronized. An audio output and sound generation threads (Sec. 4.9.2) are created and started.

A socket is created that is used to communicate with the GP Java module (Sec. 4.9.3). Threads are created for receiving commands from the Java module and for sending messages to it.

4.9.2 Sound generation loop

The output audio signal is generated in real time one block at a time. The default length of the block (or buffer) is 4096 sample frames. The length of the buffer can be changed with a test configuration parameter.

Figure 4.4 shows the player's audio generation loop. It consists of two threads and two

[7] Real-time priority requires root-privileges. For this, the player executable is usually installed as suid-root. After setting the RT priority, the player switches the effective user ID of the process to the ID of the running user (real user ID), which generally should be other than root.

queues. The *signal calculation thread* receives an empty audio buffer from the *signal calculation queue* and generates the audio signal. The calculated buffer of audio data is placed into the *audio output queue*. The *audio output thread* outputs the audio buffers from the output queue to the actual audio hardware. The audio buffer is then recycled and placed into the calculation queue.

Audio signal calculation

When a free audio buffer is available in the calculation queue, the buffer is taken from the queue and zeroed. The audio frame number of the buffer is taken from the buffer. The audio frame number tells when this buffer will be sent to the audio outputs.

The table of all loaded samples is inspected to see which samples are active. An *active* sample is a sample that is playing currenly (*playing*) or has been given a command to start but hasn't yet started (*waiting*). The signals of the playing samples are mixed to the audio buffer. If a sample is waiting, the sample's starting audio frame number is compared to the audio frame number of this audio buffer to see if the output of this sample should start during the current audio buffer. If so, the sample is marked as playing, and the signal of the sample is mixed to the current audio buffer starting from the position specified by the sample's starting audio frame number.

After all active samples have been mixed into the current audio buffer, the buffer is moved in the audio output queue.

Output of audio signal

Audio buffers are taken from the audio output queue. The audio frame number of the buffer is taken from the buffer and compared to the audio frame counter of the audio port. If they do not match, a drop-out (Sec. 4.6) has occured. The player sends a notification about the drop-out and skips enough data from audio buffer to get to proper audio frame so that next buffer will be output at the right time.

Next, the audio data in the audio buffer is written to the audio port(s). A new audio frame number is calculated for the audio buffer and the buffer is moved to the audio calculation buffer.

Sound generation initialization

The sound generation loop is initialized by placing one or two empty (zeroed) audio buffers into the output queue with zero as the output frame number. The audio output thread outputs the silent buffer(s), places new valid output frame numbers to the audio buffers, and puts the audio buffers to the calculation queue.

4.9.3 Communication with Java-module

The sound player program talks with the Java-module using a socket[8] using a simple text protocol. Usually a command starts with the ID string of the player object (a sample or a virtual player) that the command applies to. The ID is followed by the command and variable number of parameters (the type and number of parameters depends on the command).

Replies and messages from the sndplay to the Java-module work similarly. A reply starts with the ID of the player object that this message applies to, followed usually the command sent to the player and a diagnostics code indicating whether the command was succesful or not. Succesful commands also return variable number of return values depending on the command. Unsuccesful commands may add a string describing the error after the diagnostics code.

A “receiver” thread waits for commands from the command socket. When a command is received, the player object ID, the command, and the parameters are extracted from the command. Finally, the command and parameters are passed to the player object specified by the ID for processing.

4.10 Sound player Java-module

The `guinea.player` Java package is used to control the sound player program (Sec. 4.9). It hides the implementation and low level details of the C-module. Java classes are provided for using the player in an object-oriented fashion. Classes are provided for the sound player, virtual players, sound samples, volume levels, and sound player events.

[8] Also Unix's standard input / output can be used. It is handy for testing the sound player manually.

4.10.1 Players

The `Player` interface is implemented by all player objects. It is a basic interface that provides basic methods such as: querying the properties of the player (does it support audio or video, or is it a virtual player), parent player (if this is a virtual player), communicate with the actual sound player program, adding/registering and removing player objects (samples, virtual players) or event listeners, and starting and stopping the player.

The `AudioPlayer` interface (an extension to the basic `Player` interface) is implemented by sound player objects. It provides additional methods for setting and getting the name of the audio device(s), sample rate, number of channels, and the player's output volume level.

The `VirtualPlayerSupport` interface is an extension that is implemented by players that support virtual players (Sec. 4.3). The interface provides methods for allocating new virtual players and freeing unused virtual players.

The `VirtualPlayer` is a class that is used as virtual player. Players of this class are returned when a virtual player is created for the `SoundPlayer` class.

The `SoundPlayer` class is the pricipal sound player that the GP uses to control the sound player engine (sndplay). It implements the *player*, *audio player*, and *virtual player support* interfaces to provide an audio player that supports creating virtual audio players.

4.10.2 Samples

Sample objects are based on abstract `Sample` class that defines basic features of samples. It provides methods for basic event handling (adding and removing event listeners, sending events), querying sample's properties (length, number of channels, sample rate), and basic operations on samples (start or stop sample, set position in sample, set volume level).

In GP all sound sample files are used via the `SoundSample` class (sub-class of `Sample`). Sound sample objects communicate with the external player (sndplay).

To implement parallel switching, a "virtual sample" of class `ParallelSample` is used. It looks just like a single sample to the GP system but it automatically handles multiple samples (`SoundSample` objects) at the same time and does the cross-fades when samples are switched.

4.10.3 Volume levels

The `Volume` class is a base class for expressing volume levels. Three sub-classes are provided that allow giving levels in either *linear*, *decibel*, or *percent* scales (see Sec. 4.4). Methods of the volume classes can be used to convert levels between the three scales. Throughout the GP systems, volume objects are used instead of plain numeric levels.

4.10.4 Events

Players and samples generate events to inform event listeners about changes in the status of player objects.

Players generate events when a player is started or a player has been stopped. Diagnostics events are sent when a drop-out (Sec. 4.6) occurs or the sound player program prints messages to its standard error stream.

Samples generate events when a sample starts, stops, or it has looped. If chosen, samples also send events about the current position in sample that is currently played. The event includes the audio frame number when the event occured.

Chapter 5

Test engine

The *test engine* of the GuineaPig system handles the running of a test and co-ordinates the tasks of the various parts of the system (the sound players, UI panels of the subjects, running the test itself (the test engine), and various configuration and results files).

Based on the parameters in the test configuration files, the engine initializes the sound players and UI panels and loads samples and test items. Then it performs the test by presenting the test items (specified in playlists) to the subjects and records their answers.

Due to lack of time, no graphical tools for creating and designing tests could be built as fantasized in the beginning of the project. Building good graphical tools, even rather small or limited ones, would have required a great deal of work. It was considered that it would be more important to get the essential basic core functionality working first before anything other non-essential features were built. Therefore, tests are currenly designed by writing test parameters in simple text configuration files.

The test engine does not really have any rigidly defined test types. Most test types are basically the same with different answer (or question) types or different parameters. Only more special test types need some additional code for them. The GP system includes example configurations to show how to implement many standard tests (see Sec. 5.8).

5.1 Test items

Test items define the parameters of individual test cases used in a test. They are also used to store the answers given by the subject to the questions as well as other infor-

In *test–items* file:

```
item1.A:          pirr44        Test item ID
item1.B:          pirr32

item2.A:          pirr22        Item parameters
item2.B:          pirr32
# Comments can be included      Optional comments

item3.A:          pirr8
item3.B:          pirr11        Parameter values
```

Figure 5.1: Example of a test item definition for an *A/B* or *A/B Scale* test. Three test items are defined with sound sample IDs as parameters A and B.

mation about the item tested. Any number of item parameters can be used.

Test items extend the abstract base Java-class `guinea.logic.Item`, usually a generic test item `guinea.logic.GenericItem` is used.

5.1.1 Defining test items

Defining text items requires specifying an *item ID*, the test item's Java-class name, list of names of parameters that are used, and the values of the parameters. In most cases the test item's class name need not be specified as GP uses a generic test item by default. If special test items are used, the class-name needs to be specified. Also, the list of parameter names usually does not need to be specified. The tests supplied with the GP package automatically provide the list of needed parameters. If additional parameters are used, the list of parameters needs to be specified. Item *templates* (see below) can be used to specify the Java-class name and the needed parameters list only once, without the need to repeat the definitions for each item separately.

Usually the parameters of a test item are the samples that are compared in that case. Parameters are Java-objects (`java.lang.Object`) so basically anything can be used as parameters. However, currently only strings (Java-class `java.lang.String`) are used. Parameters are ID strings that refer to the samples defined in the *samples-file*. For example, a test item for an A/B test has two parameters, A and B, which are the ID labels of the samples that are compared.

Test items are defined by writing item parameters into an ASCII-text *items-file*. The text format uses Java's properties file format as its base. Figure 5.1 provides an example of a test item definition. In the example, three items are defined, each having two parameters. The parameter names are prepended with the *item ID* of the test item. The item ID of a test item must be unique.

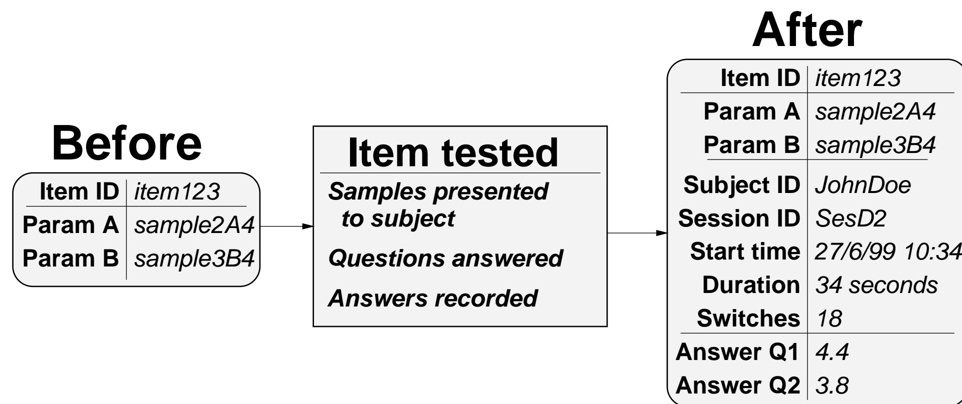

Figure 5.2: Test items are used both for providing test parameters as well as for storing the answers given by the subject. When an item is tested, IDs of the samples that are compared are read from item parameters. After item has been tested, answers to questions and other information are stored and later saved.

Loading items and item templates

Test items are loaded from the items-file at the start of test. Loading is done for each item by copying a *test item template* for the new item and then setting the parameter values from the parameters defined in test items file for that item. Any number of additional templates can be defined.

The templates and items form a tree-structured hierarchy. The test's *default template* is the root of the tree. It usually defines the names of the parameters that there are in each test item and possibly the class-name of the test item. For more specialized tests, the tester can override the default template with additional parameters. Also, if a value is not defined for a parameter in a test item (a leaf of the tree), the value is looked for in the item's template, which again looks for the value from its template if it is not set.

Item templates are currently of little use as all tests are quite simple with only a small number of parameters. They probably will be helpful if the tests are developed for more complex tests in the future.

5.1.2 Item results

Test items are also used to store the subject's answers to questions as well as other information. When an item is tested, a copy of the item is created for each subject. After the item has been tested, the answers and statistics information about the item are added to the copy (illustrated by Fig. 5.2).

Information that is stored about testing a particular test item:

- The *item ID* of the test item that was tested.
- The *parameters* of the test item. The parameters usually are IDs of the sound samples that are compared.
- The *answers* to the questions shown to the subject. The answers are saved to a table with the *question ID* of the question component in the subject's answer panel as the key. Java objects are used as answers. Currently, mainly numbers (the sub-classes of Java's abstract number class `java.lang.Number`) and strings (Java's string class `java.lang.String`). More complex answer types are possible, such as the ranking order answer used by the rank-order question component (Sec. 6.3.3). An answer for a question may be missing, usually because the time limit (Sec. 5.4) has expired.
- The *session ID* of the test session during which the test item was presented.
- The *subject ID* of the test subject to whom the item was presented to and who gave the answers.
- The *item start time* when this item was presented (both time and date are stored).
- The *duration* of the item, how much time the subject used to grade the test item.
- *Number of samples played* during the item testing. It can be interpreted also as the number of sample-switches. In free sequence tests, the number is the number of sample plays performed by pressing the sample-play buttons in the subject's panel. A high number may indicate that comparing the samples was difficult. In fixed sequence test the number is not really useful, the number is simply the number of samples played in the playback sequence (same sample played multiple times is counted as multiple sample-plays).

The tested items are stored into a session log and saved to a results file at the end on test. The data is available for analysis as a printable table with the *results processing* (Sec. 7.2).

5.2 Playlists

A playlist is used to define a subset of all defined test items for presentation to a test subject (or a group of test subjects). The playlist also defines the presentation order of the test items.

A playlist is a text file containing a list of test item IDs, one item ID per line, that are to be presented. The test items are presented in the order their item IDs appear in the playlist. All test items that have been defined do not have to be presented. Optional comment lines (lines starting with the #-character) and empty lines are ignored.

The GP system automatically looks for a playlist-file based on the ID of the test session. If no playlist is found for a specific session, a default playlist is looked for. If no playlist is defined, all test items that have been defined are presented. Without a playlist, the presentation order is undefined.

With playlists together with test items, tests can be created to conform to the needs of block designs, with easy implementation of efficient *balanced complete / incomplete block designs* experiments, for example [Dav63, pp. 83–103], [CC92, pp. 439–482]. Such designs can be created with commercially available software (e.g. SPSS Trailrun) and imported into GP by use of playlists.

5.3 Sample playback sequence

Tests allow for either fixed sequential playback, as with a tape, or for free switching (a free-sequence test) between samples. In tests using fixed sequential playback, the tester defines the playback sequence that is presented to the subject (or subjects). In tests using free switching, no fixed playback sequence is enforced, and the subjects play the samples as many times and in any order they like.

The subjects can give their answers only after all samples have been listened to (the fixed sequence has been played or with free switching, the subject has listened to all samples at least once).

5.3.1 Fixed playback sequence

In tests using a fixed playback sequence, the tester defines a playback sequence of samples that is presented to the subject. The samples in the sequence are played sequentially. Optionally, pauses can be added between samples. The length of the pause is unlimited and can be given either in seconds or in number of sample frames. The samples to be played are referred with the test item parameter names that correspond to the samples that are compared. An example of a sequence is shown in Figure 5.3.

The playback sequence is configured by selecting the fixed sequence type and defining the playback sequence as a text string. Figure 5.4 shows an example configuration that corresponds to the sequence in Figure 5.3.

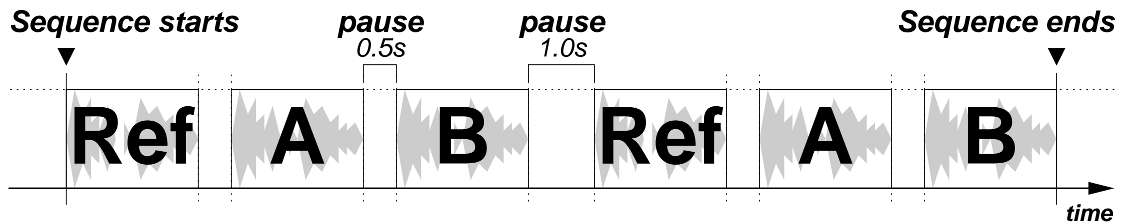

Figure 5.3: An example of a fixed sequence. Samples *Ref*, *A* and *B* are played in sequence with a pause of 0.5 seconds between them. Then a pause of one second followed by the same sequence again.

In test configuration file:

```
# Specify fixed sequence
sequenceType: fixed
# The fixed sample sequence
sequence:     Ref;0.5s,A;0.5s,B;1s,Ref;0.5s,A;0.5s,B
```

Figure 5.4: An example of a fixed sequence configuration. This configuration produces the sequence shown in Fig. 5.3.

Currently, playing a same sample more than one in a row is not quite robust due to the implementation of the sound player. The player can not reliably start a sample again just after it has stopped. The start of the sample may be delayed a bit compared to the expected starting time. Adding a pause (of at least half a second) between them should make the sequence more reliable. However, the same sample can be used multiple times in a sequence as long as there is at least one other sample between them. For example, the sequence **A;A;B;B** is a bit unreliable, but the sequence **A;B;A;B** is reliable.

5.3.2 Free sample switching

In free switching tests, no fixed playback sequence is enforced. The subjects can play the samples as many times and in any order they like. The subjects play samples by pressing buttons on the sample-play controller (Sec. 6.3.4) on the subject's UI panel.

Normally when the subject selects a sample to play, the currently playing sample is stopped and the new sample starts playing from the beginning of the sample. Switching can be done also in parallel, where all samples that are compared are playing at the same time but all but one are silenced. When the subject selects another sample, the currently playing sample is switched to the new selected sample using a cross-fade (see Fig. 5.5). The length of the cross-fade and the type of the fade (amplitude-linear or decibel-linear) are configurable.

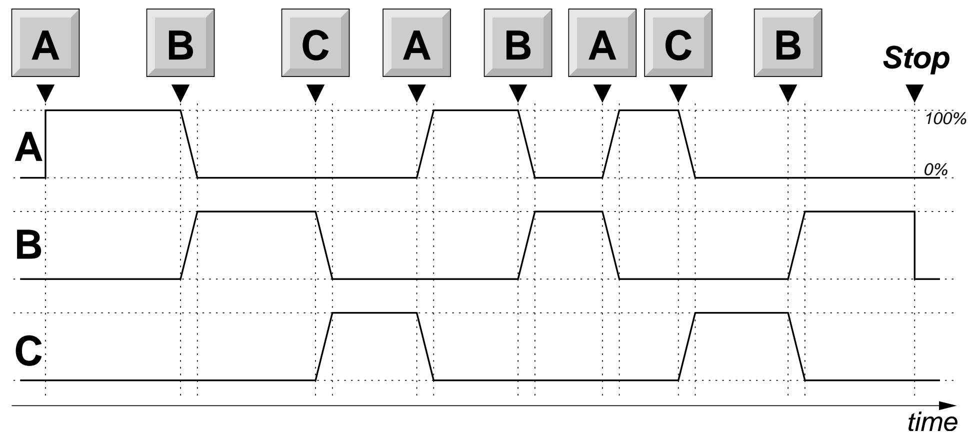

Figure 5.5: Free parallel switching (Sec. 5.3.2) using a crossfade. In this example, three samples A, B and C are compared. All samples are played concurrently but only one has a non-zero volume level. To switch to another sample, the subject presses the corresponding buttons (boxed letters on the top) on the subject's UI panel. The switch is done with a cross-fade (amplitude-linear fade).

5.4 Answering time limit

An *answer time limit* can be set for grading a test item. When the time limit is used, an indicator (Fig. 6.1d) is shown in the subject's UI panel that shows how much time is still left before the item "times out". Also, a different kind of warning is possible when remaining time is about to end. If a timeout occurs, the item is marked as "timed out". Then the item's answers are stored and the test proceeds to the next test item as normally. If no time limit is set, the subjects can take as much time as they need to give their answers.

The test engine uses the time-limit interface (Sec. 6.1.3) of the subject's answering panel to show remaining time and signaling the test engine about the timeout if it occurs. Special UI component, timeout indicator (Sec. 6.3.7) is used to implement the indicator. Figure 6.10 shows the indicator in action.

The time limit is configured in the test configuration file. The time limit is given in seconds. Also, a warning time can be specified. As shown in Figure 6.10, the warning timeout is the length of time before the answering time ends. If no warning time is specified, no warning is displayed before time runs out. If time limit is not specified, no time limit is enforced. Figure 5.7 shown a time limit configuration example.

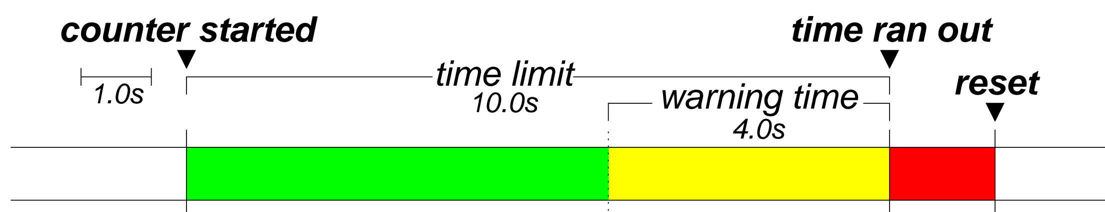

Figure 5.6: A timeline showing the states of the answering time indicator when answering time is limited. The total time allowed for answering the questions is 10 seconds. When there is less than four seconds to answer, the time indicator on the subject's UI changes from green to yellow to warn that time is about to end. If time runs out, the indicator turns to red for a second and then test proceeds to the next item.

In test configuration file:

```
# Time allowed for answering
itemTimeout:          10.0
# Warning before timeout
itemWarningTimeout:   4.0
```

Figure 5.7: Example of answering time limit configuration. This configuration matches the timeline shown in Fig. 5.6.

5.5 Most comfortable listening level

The *most comfortable listening level* (MCLL or MCL level) can be fixed or the test subject can select a comfortable level within a range defined by the experimenter. Whilst this type of control maybe be convenient in some tests, it should be included as part of the analysis to avoid level bias effects. The MCL level of the session is available from test results output (Sec. 7) for analysis.

The MCL level is set at the start of a test session before the first test item is presented. The test engine loads a test signal sample, sets initial output level and starts it looping. Then subject's answering panel is used to show a MCL setting panel (using subject panel's MCL setting interface (Sec. 6.1.2). Subject's panel then pops up the MCL level controller panel (Sec. 6.3.6, see also Fig. 6.9) to set the level. The subject uses a slider to adjust the level. As the subject adjusts the level, control events are sent to the test engine. The test engine catches the event, extracts the current level from the event and sets the player's output level (Sec.4.4.1) to the selected level. The subject presses a 'Done'-button when a comfortable level is found. MCL level panel is closed, test signal sample is unloaded and test proceeds to the first test item. The selected level is stored for analysis later. The tester can select any sample as the test signal sample.

The parameters of the MCL level settings are configured in the test configuration file. First it is decided whether the subject sets the listening level of the session or is it fixed by the tester. If level is fixed, the level is set to the default level specified in the config-

In test configuration file:

```
# Subject sets MCLL
MCLL.subjectSetsLevel:  true
# Default (or fixed) MCL level.
MCLL.default:          -20dB
# The limits of selectable listening level.
MCLL.max:                0dB
MCLL.min:              -40dB
```

Figure 5.8: Example of a MCLL configuration. The test subject sets the level. The allowable range the subject can select a level within is $[-40, 0]$dBfs with -20dBfs as the initial or default level.

uration file. It is also used as the initial level if the listening level is set by the subject. Finally, the minimum and maximum levels that the subject can use are defined. Figure 5.8 shows a configuration example.

5.6 Multiple subjects concurrently

The GP system allows testing multiple subjects at the same time using multiple remote terminals (Sec. 6.4). There are two slightly different variations of testing multiple subjects at the same time.

5.6.1 Fixed sequence with multiple listeners

The sample playback sequence is fixed and multiple subjects listen to the same signal. The system waits until all subjects have given their answers from their own terminals before proceeding to the next test item. Limited answering time may also be used.

5.6.2 Multiple independent listeners

Multiple subjects can perform the test at the same time but independenly of each other. The subjects can go trough the test at their own speed. This is essentially the same as performing multiple single-subject sessions of a test at the same without the need for several workstations. A different virtual player is assigned for each remote terminal. Also, independent listeners can each have a different playlist.

5.7 Test process

For running a test, a simple graphical tool is used. It allows to select a playlist file, set session ID and to add test subjects for testing (local or remote). Figure 5.9 shows the

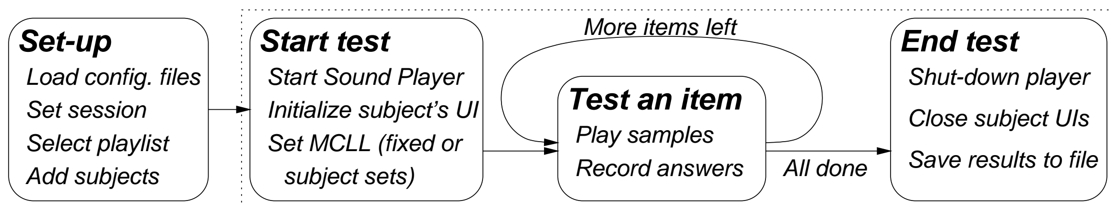

Figure 5.9: The principal phases of the testing process. First test configuration file is loaded, session parameters are set and subjects are added (Sec. 5.7.1). The test engine is started and its modules are initialized (Sec. 5.7.2). The list of test items is processed one at a time until all items have been tested (Sec. 5.7.3 and Fig. 5.10). Finally test is finished and results of the test (answers to questions) are saved to a file (Sec. 5.7.4).

phases of a test.

5.7.1 Test session set-up

At the set-up of a session, the test is started with a graphical tool. First, the parameters of the test are read from the test configuration file. The configuration file contains the names of additional files where more information is stored. Based of the information, the sound player is initialized and started, the UI module is initialized, and test items are loaded.

The sound player is started and initialized with the parameters (sample rate, number of channels, and the audio device to use) as specified in the test configuration file. Also, sound samples information is loaded from the samples-file. Sound sample objects are created but the actual sound files are not loaded at this point. They will be loaded on demand only when they are needed.

A *session ID* can be selected for the test session[1]. A session-specific playlist (Sec. 5.2) is loaded if one is found, otherwise a default playlist file is looked for. If no playlist is found, all test items are presented but the order of the items is undefined.

Test subjects are added to the session. Both local (a console user) and remote terminal users (Sec. 5.6) can be added. When a subject is added, a subject UI panel is created and initialized for the subject. The subjects appear on a table of subjects, where a suitable subject ID can be set or the default ID can be edited.

[1]A default, automatically generated ID is used if no other ID is provided.

5.7.2 Test start

When the test session is started, a *test thread* is created to run the test procedure. The different modules of the test system are initialized and started and a session log is initialized.

The item presentation list of the session is initialized. The (optional) playlist is applied to the list of items to select and re-order the wanted items for this session.

The subject panels are opened. Initially, the panel is disabled. If time limit is used, the time limit indicator is added and its parameters are set (the length of the time limit).

The MCL level is set. If it is fixed, the default level (specified in the test config. file) is used as the output level of the sound player. If the subject can set the MCLL, a test sample is played and the subject adjusts the level of the sample with a slider in the MCLL panel that pops up for this purpose. The level the subject selects is used as the output level of the player.

5.7.3 Test item testing

After test start procedures are done, the test engine starts to process the test items. The test engine loops through the list of items ("Test an item" in Fig. 5.9) that are tested until all items have been tested. Figure 5.10 shows what happens when a test item is tested.

Item initialization

Any necessary initializations needed by the current item may be performed at this point. It may include, for example, reading parameters from the test item and decoding them, or setting some other parameters. Currently GP does nothing at this point, much of what is needed is done at sample loading (see below).

Item samples initialization

The IDs of the samples that are used in the current test item are queried from the parameters of the current item. The sound samples are then taken from the set of samples the test uses (the samples defined in the samples-file), and the samples are loaded from disk.

If free switching (Sec. 5.3.2) is used, a *parallel sample* is created that plays the test sam-

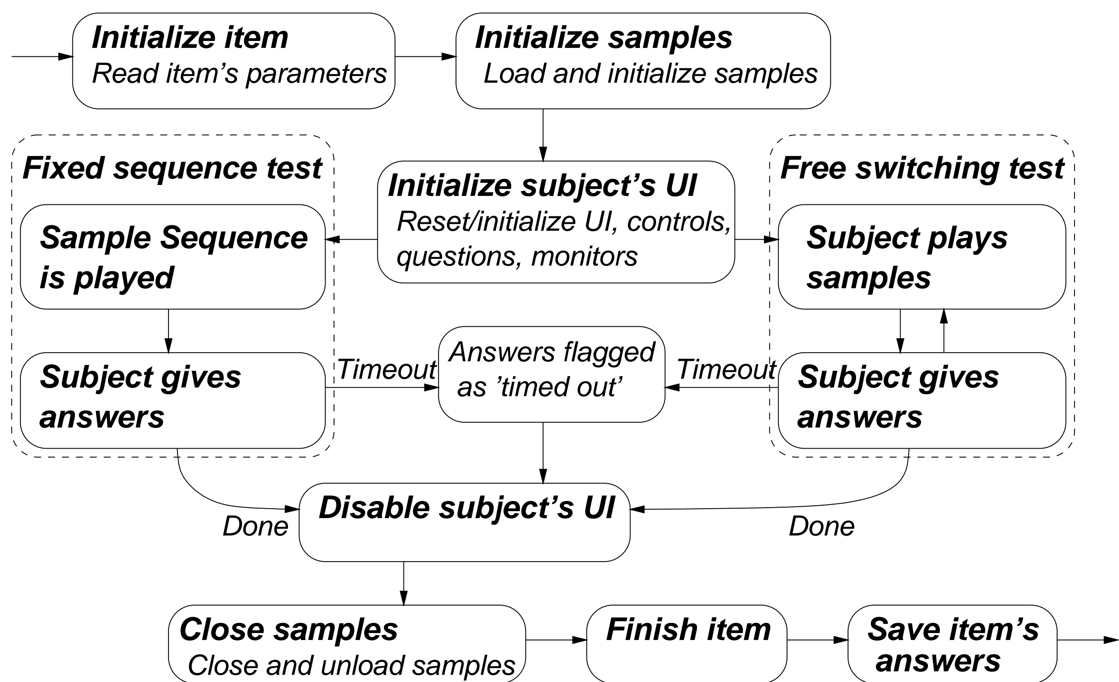

Figure 5.10: Testing of a single test item. See the subsections of Sec. 5.7.3 for more detailed description of each phase.

ples in parallel and performs the sample switch with a cross-fade. Also the parameters of the cross-fade are set.

Subject panel initialization

All the components in the subject's panel are reset to their initial state. Questions are set to unanswered state and possibly to a initial position (for example, in sliders).

Control components are enabled (to allow playing samples, for example). Also, the control's initial value should be set so that it matches the value that is used by test engine at the beginning of an item. For example, the position of a level control slider should match the initial volume level of a background sample.

By default, answering questions are disabled at the start of item testing. Answers can be given only after all samples have been listened to[2]. Also, the 'Done'-button on the subject's panel is enabled only after all questions have been answered.[3]

The test status monitor (Sec. 6.3.9) is set to show the index of the current test item and

[2]If desired, answering questions before samples have been listened can be enabled with a test configuration parameter.

[3]This also can be changed with a configuration parameter. This option is useful for creating a training session where answers are not needed.

the total number of items in this session.

Sample playback and grading

If fixed playback sequence (Sec. 5.3) is used, the sample sequence is played to the subject(s). Otherwise another method is called which allows customizations at the beginning of grading. GP system can automatically start playing a sample at the start of testing the item (for example, a background noise sample).

The time-limit indicator (Sec. 6.3.7) is activated if answering time is limited (Sec. 5.4). If no time limit is used, the subject may take as much time as he/she needs. In both cases the subject's panel signals the test engine when the subjects are done grading the item or the time limit has expired. The test engine waits until the signal has arrived before it proceeds to item testing cleanup.

The test engine listens to answer events arriving from the subject's panel. Answer event are generated when the subject gives an answer. When an answer event is received, the answer is extracted from the event and is recorded to a list of answers. The new answer replaces an old answer if there is already one given. The test engine enables the 'done'-button on the subject's panel when all questions have been answered (it can also be enabled earlier depending on other test parameters). It is also possible for a question component to send a `null`-answer. In that case, the test engine deletes the answer for that question (and also may disable the 'done'-button). A null-answer may mean that, for example, the last selection makes the answer invalid.

Control information is delivered the same way to the test engine. When the subject makes a control action (for example, adjusts a level slider or presses a button to play a sample), an event containing the new control value is sent to the test engine. The test engine handles automatically playing samples (using the sample-play controller described in sec. 6.3.4), and the MCLL setting (Sec. 5.5).

The processing of control events is currently a bit cumbersome and inflexible when additional controls are needed. Currently, a test class needs to be extended to catch and process any events generated by additional controls. A better way would be that custom plug-in components could be added more easily to handle the events generated by additional control components.

Monitoring components are used to show the subject some status information about the test. Unlike question and control components, monitor components do not generate events. The information it displays is send by the test engine to the subject panels. The test engine automatically shows which sample is currently playing (with

sample-play component) and test progress status indicator (Sec. 6.3.9).

Like with controls, adding custom monitoring components is rather inflexible. A test class has to be extended to generate the information that is sent to a monitor component for display.

Subject panel finishing

After the item has been processed (questions answered), the components on the subject's panel are disabled. It is again enabled when processing of the next item starts. Other user interface clean-up, etc. operations could be performed here.

Item samples finishing

Samples that are playing are first stopped. If a parallel sample was created to perform cross-fades, it is first destroyed. Then all the samples used in the current item are unloaded and disposed of.

Item finishing

Additional post-processing of the test item and its results may be added here. Also, resources allocated at the start of item should be released here.

In current tests included in GP, little is done here. The only test currently using it is the TAFC test that creates and adds the final answer based on the responses of the subject during testing the current item.

Item answers saving

A copy of the test item is made for each subject. The answers given by the subject are saved into the copy. Also other information is saved about testing the item and the subject (see Sec. 5.1.2).

5.7.4 Test end

After all test items have been tested, the sound player is terminated, and the subject panels are closed. Additional resources that were allocated at the start of the test are released.

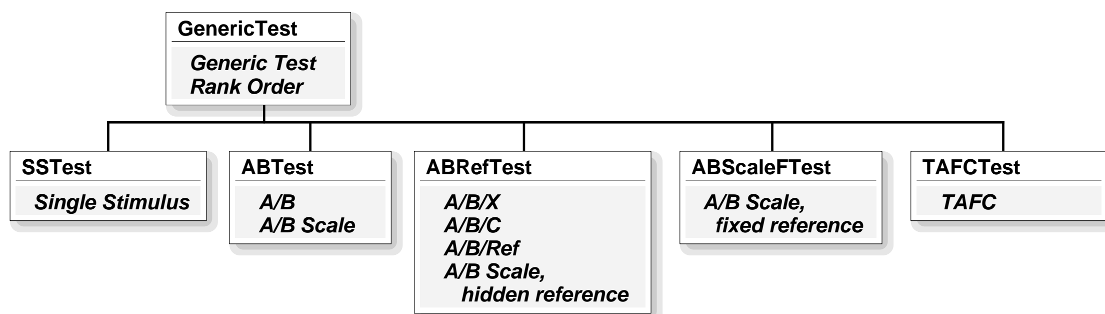

Figure 5.11: Test types of the GuineaPig system categorized from the point of view of their implementation. Each box shows which types of tests are implemented by that test class (Java-classes).

Test items with the answers of the subjects are added to the session log. Also the start and ending time of the test session, and the MCL level of the session are saved. Finally, the session log is written to a results file in Java's serialized format. The results can be printed out with results processing tools in sec. 7.

5.8 Test types

As already stated, the GP system and the test engine does not provide any rigidly defined tests. All tests are based on a **generic test** that provides most of the functionality needed by all test types. The GP system includes example configurations to show how to implement many standard tests using the GP system.

The categories of tests included in the GP system can be seen in figure 5.11 from the point of view of their implementation. Table 5.1 shows the parameters and results or question types of the tests that are included in the GP system.

The generic test is a comparison test that allows comparing any number of samples at the same time. Also, each sample can be assigned to a different virtual player. It could be used to compare several different sets of speakers, for example (as in Fig. 4.3).

The pre-defined test types in most cases are just shortcuts that define the needed parameters in test items and possibly also define other parameters needed by a particular test. Most tests can be implemented with just the generic test type alone, but some parameters need to be defined manually.

Included are test types and examples that can be used to implement the following tests. For each, a *test configuration file*, a *UI configuration file*, and an *items file* is provided as an example.

Test type	Parameters	Question and result types
Single Stimulus	A	grade(A)
A/B	A,B	preference(A,B)
A/B/X	A,B,X	preference(A = X,B = X)
A/B/Ref	A,B,Ref	grade(A ↔ Ref), grade(B ↔ Ref)
A/B Scale	A,B	grade(A), grade(B)
A/B Scale, FR	A,Ref	grade(A ↔ Ref)
A/B Scale, HR	A,B,Ref	grade(A ↔ Ref), grade(B ↔ Ref)
TAFC	B,Ref	level(B ↔ Ref)
Rank Order	$P_1, P_2, \ldots, P_N$	ranking order($P_1, P_2, \ldots, P_N$)

Table 5.1: Parameters and results of test types. A *grade* corresponds to a grader question component and a resulting scalar value. An arrow (↔) means the parameter on the left side is compared against the one on the right side. A *preference* is a multiple-choice question whose answer is one of the choices. Equal sign (=) means that the parameter on the left side is the same as on the right side. The *level* is the resulting level reached with the TAFC procedure.

Single Stimulus A Test in which a single stimulus signal is played and then graded (using a grader component, see Sec. 6.3.1).

A/B Simple paired comparison where two samples are played and the superior or inferior sample is selected [Dav63] using a multiple choice question component (Sec. 6.3.2).

A/B Scale Paired comparison with a scale for each sample. Also referred to as *double stimulus continuous quality method, ITU-T BT.500-8* [IR98b]. From the implementation point of view, this test is identical to the A/B test, just questions and results are of different type.

A/B/Ref Three samples are played. Samples A and B are graded against the reference (Ref). Compared to the A/B scale test, this test has an additional reference parameter against which A and B are compared.

A/B/X Three samples are played. The listener is to select which sample, A or B, is the same as X [Cla82, Cla91]. This is similar to both the A/B and A/B/Ref tests. As in the A/B/Ref test, it has a third sample parameter, but the question is the same as in the A/B test.

A/B Scale, Fixed Reference The subject gives a grade on how the sample compares to the reference. From the implementation point of view, this test very similar to the A/B test, parameters are just named differently and a scale is used as the question instead of a multiple-choice question.

A/B Scale, Hidden Reference The subject gives a grade for both samples on a scale specified by the test creator. One of the samples A or B is the same as the hidden

reference. This is similar to the A/B/Ref test, but the reference is not played (hence the “hidden” reference).

TAFC (Two alternatives forced choice) Two samples are played altering some parameter until the subject can no longer hear the difference between the samples [Lev71]. From implementation point of view, this is the most complicated test in the GP system. A lot of code is needed to run the TAFC procedure.

Rank order N samples are played and the listener grades them in rank order of intensity with a pop-up menu [LH98] (Fig. 6.6). This test is implemented directly using the generic test type. As the question, a special ranking order question component (Sec. 6.3.3) is used.

Chapter 6

Subject's user interfaces

The subjects use a user interface (UI) panel (Sec. 6.1) to control the test and to give their answers to the questions. Also, test status information can be displayed on the panel. Currently the UI panels implemented in GP are graphical but it is not necessary.

A selection of UI components are included (Sec. 6.3) that are used for answering questions (different grading scales, multiple-choice, rank-order), for controlling the test (playing samples, setting a level), and for providing status information for the subject. Additional custom components can be easily added.

Remote terminals enable testing multiple subjects at the same time (Sec. 6.4).

6.1 User interface panel

Figure 6.1 shows an example of a subject UI. Currently, all the subject UIs in the GP are graphical but they do not need to be. Any class that implements an interface for subject UIs can be used. That way it would be possible to use simpler, non-graphical interfaces for testing. For example, a module could be written that uses a (physical) panel of buttons to give answers. Currently, subject UIs are implemented with Java's Abstract Window Toolkit (AWT).

The UI is defined in its own configuration file that uses a simple, text format (Java properties format). The file defines what UI components to include and their parameters. Also other parameters can be included, such as the title of the window, the size of the window (can be fixed to fixed dimensions) and fonts. Two graphical tools are provided for testing UI configuration files and selecting fonts (Sec. 6.5).

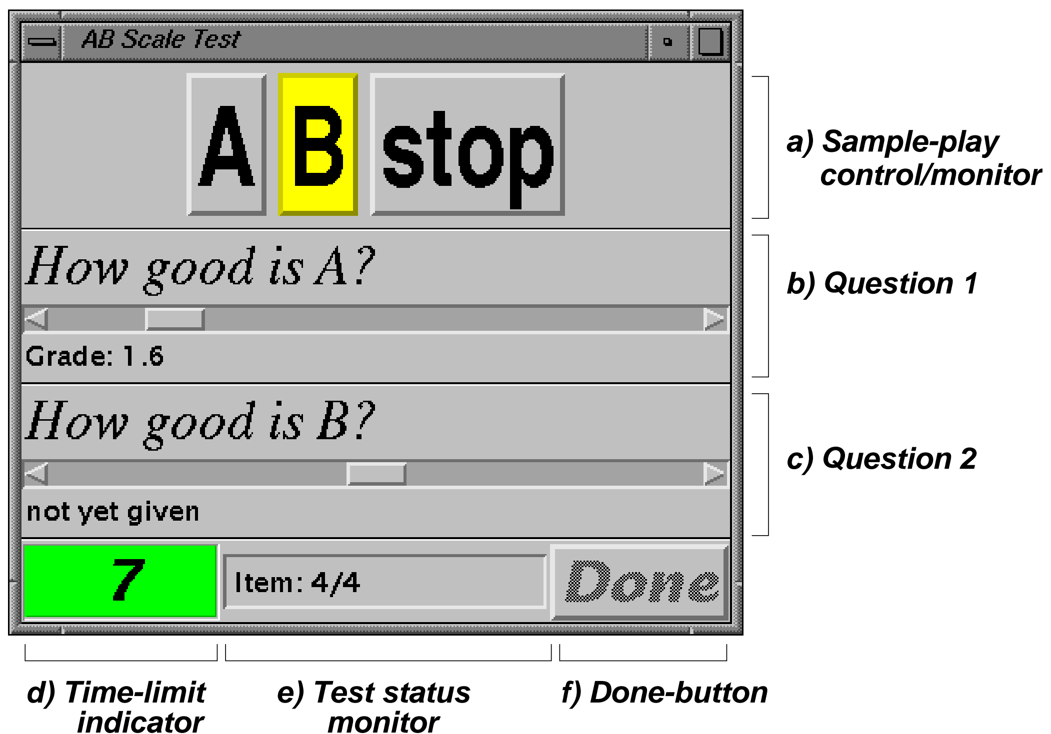

Figure 6.1: An example of subject's user interface (UI). a) is a sample-play control and/or monitor (Sec. 6.3.4) used to play samples and to show which sample is playing currently. b) and c) are scale questions (Sec. 6.3.1). First question has already been given a grade, second has not been answered yet. d) shows how much time (in seconds) there is left to answer (Sec. 6.3.7). e) shows that this is the last item in a set of four (Sec. 6.3.9). f) Subject presses the Done-button (Sec. 6.3.8) when he/she has completed grading this item. It it currently disabled because question two hasn't been answered yet. It is enabled when all questions have been answered.

6.1.1 Basic panel interface

The GP testing system controls subject user interface panels using a special Java interface `guinea.ui.SubjectUI` that defines the methods that can be used to control the panel. The interface defines methods such as:

- Opening and closing the panel window.
- Disposing of the panel window when it is no longer used. Releases the resources used by the panel.
- Resetting the panel to initial state.
- Enabling and disabling the panel and its components. Controls and questions can be enabled or disabled as a group (all questions or controls are enabled).
- Getting a list of question, control, or monitor component names that have been defined.
- Adding and removing event listeners.
- Setting the answer of a question or setting the value of a control or monitor component.

6.1.2 MCLL setting interface

The interface `guinea.ui.MCLLControl` (an extension to the basic panel interface in Sec. 6.1.1) allows the setting of the MCL level (Sec. 5.5) by the subject. The interface includes methods for:

- Setting the limits of the listening level that the subject can select.
- Set the text for the volume controller label and the window title.
- Set the default or initial volume level of the volume slider.
- Show or hide the level setting window.

The volume levels can be expressed using linear, decibel or percent scale (Sec. 4.4). GP uses the MCL level controller in Sec. 6.3.6 to implement the level setting.

6.1.3 Timeout warning interface

The interface `guinea.ui.WarningDisplayer` provides an extension to the basic panel interface (Sec. 6.1.1) to limit the time used for grading an item (Sec. 5.4). The principal methods included in the interface are:

- Whether to use a time limit or not.
- Setting the time limit and optional warning time.
- Starting the time limit counter.
- Resetting the time limit to inactive state.

The subject's panel in GP uses the time-out indicator in Sec. 6.3.7 to implement the time limit.

6.2 User interface components

The subject UI's components fall into three main categories:

- *Question* components are used to give answers (Sec. 6.2.1).
- *Control* components allow the subject to control some part of the test (Sec. 6.2.2).
- *Monitor* components provide status information about the test (Sec. 6.2.3).

Unlimited number of subject UI components can be defined. Any custom component can be used as long as it implements simple subject UI component interfaces. Currently all components are graphical (implemented using the Java's standard AWT components) but it is not necessary. Many of the parameters of the graphical component's outlook are user-configurable, such as labels, fonts and colors (where applicable).

6.2.1 Questions

The question components are used to give answers. GuineaPig's question component Java-interface `guinea.ui.QuestionInterface` must be implemented by a component to be able to be used as a question component. The interface provides methods for

enabling and disabling the component, setting and querying the answer of the question, getting and setting the question text, resetting the component, and adding and removing event listeners.

When a subject gives an answer (makes a selection, adjusts the grade on a grade component, etc.), an answer event (GP's Java event `guinea.ui.event.AnswerEvent` or its sub-class) is sent to the test engine. The event contains the question ID of the question component and the answer the subject gave. The test system then records the subject's answer for that question in the test item.

Question components generate Java objects (`java.lang.Object`) as answers. That way any kind of answer types, how simple, complex or special they may be, can be used. The GP system does not care what the answers are actually, it just logs them. More complex answer types may require a custom formatting plug-in module to export the answer for analysis with other tools (see Sec. 7 for results processing and custom formatting of special answer types).

6.2.2 Controls

Control components are used to control some parameters in a test. GuineaPig's control component Java-interface `guinea.ui.ControlInterface` must be implemented by a component to be able to be used as a control component. The interface provides methods for enabling and disabling the component, setting and querying the value of the controlled parameter, resetting the component, and adding and removing event listeners.

Controls are usually implemented with sliders or buttons. When an adjustment or a button is pressed, a control event (GP's Java event `guinea.ui.event.ControlEvent` or its sub-class) is sent to the test engine. The test engine then processes the event.

The control components that are currently used in tests are handled automatically by the test engine: *sample-play control* (Sec. 6.3.4) to play samples, *volume level control* (Sec. 6.3.6) to set the MCL level (Sec. 5.5), and *button* (Sec. 6.3.5) in TAFC-test.

Any number of additional controls can be added but they also require writing additional code to handle them. Special tests are made by sub-classing and extending existing test classes to catch and handle the control messages sent by the additional control components.

In the future a better way should be added to allow adding custom plug-in modules to handle different controls without need to extend default test classes. That way adding multiple different controls would be easier and more flexible.

6.2.3 Monitors

Monitor components provide some status information about the test to the subjects. GuineaPig's monitor component Java-interface `guinea.ui.MonitorInterface` must be implemented by a component to be able to be used as a monitor component. The interface provides methods for enabling and disabling the component, resetting the component, and setting the value of the monitored parameter.

Currently monitor components are used for two things. The sample-play controller (Sec. 6.3.4) also acts as a monitor by showing which sample is currently playing (this is more informative when a fixed sequence is used instead of free switching). Another is an item status monitor (Sec. 6.3.9) that shows how many test items have been completed so far.

Additional monitor components can be added but it requires writing additional code and extending a base test class to handle them as with control components. In future a better way should be added to allow adding plug-in modules to display additional information without need to extend default test classes.

6.3 Provided user interface components

The GP package provides some ready-made components for building tests: a *generic scale* component with three more specialized variations, a *multiple-choice* question, a *rank-order* question to rank several objects, a *simple button* , and a *sample-play* controller for playing samples.

6.3.1 Scales

A generic configurable grading component `BaseGrader` is used to implement various numeric grading scales. Figure 6.2 shows the general structure of grading components. On the top there is the text of the question. Below it there is an *adjustable component* that implements the grading. Below the adjustable component is an optional label area that is used to show the current grade and an adjective corresponding to the current grade.

The BaseGrader is not directly used as a grading component. It is an abstract class that defines most of functionality that graders need except the actual adjustable component. Sub-classes define the actual adjustable component that is used. Any component that implements Java's `java.awt.Adjustable` interface can be used as the ad-

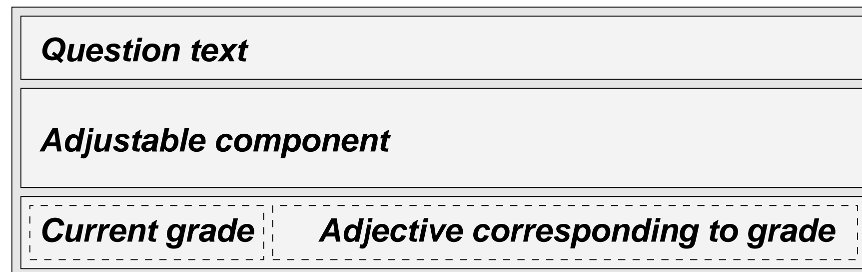

Figure 6.2: Basic structure of grading components.

justable component.

In general, configurable parameters for graders include:

- The *minimum and maximum* values of the scale.
- *Number of decimals* that is used in the answer.
- Whether to *show or hide value* to the subject.
- *Adjectives* can be associated with ranges of values.
- The *initial value* can be set to a fixed value or an automatically generated random value can be used.

grader components can be used both as a question and a control component.

GradeBar

Figure 6.3 shows the GradeBar grading component that is a sub-class of the BaseGrader component. It uses Java's `java.awt.ScrollBar` to implement the adjustable component. The position of the scroll-bar's knob is translated to a numeric grade. The minimum of the scale is at the left end of the line, the maximum is at the right end. The numeric display of the grade can optionally be disabled. Optionally, adjectives can be associated to ranges of values of the grade. The adjectives are configured the same way as Java's `java.text.ChoiceFormat` text formatter.

FiveGrade

A simple extended version `FiveGrade` implements a continuous grading scale with "anchors" derived from the ITU-R five-grade impairment scale given in *Recommendation ITU-R BS.1284* [IR98a]. This versions simply sets the scale parameters accord-

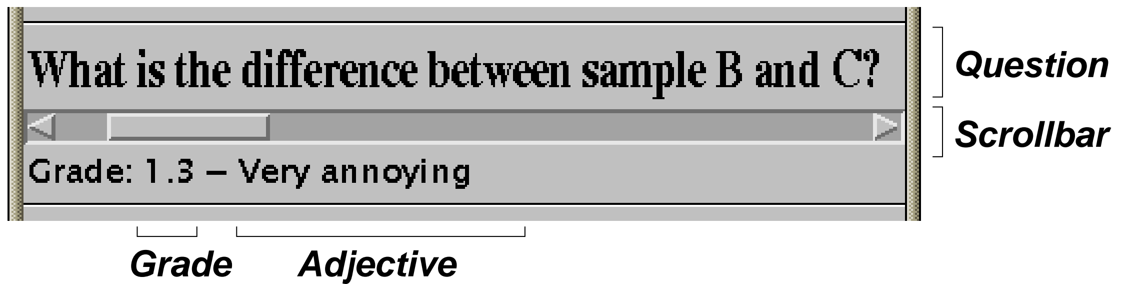

Figure 6.3: An example of a grading scale. A Java's scroll-bar is used to implement the adjustement. Below the slider, the current value of the grade is shown with an adjective associated to the value.

ingly and adds adjectives associated with ranges of values. The scale shown in Fig. 6.3 is actually a `FiveGrade`.

TenGrade

Another extended version `TenGrade` implements a ten-grade answering component. The scale goes from 0 to 10 with one decimal and shows the grade symbolically also ("Very unclear", "Rather unclear", "Midway", "Rather clear", "Very clear"). For example, *Recommendation ITU-T P.910* [IT96b].

Both `TenGrade` and `FiveGrade` components are actually not necessary to implement these scales, they are available mainly as shortcuts. Both can be implemented with the `GradeBar` component by setting the scale's range and adjectives manually.

VolumeGradeBar

The `VolumeGradeBar` is another extension to the normal `GradeBar` that gives volume level objects as answers instead of plain numeric values. It converts GradeBar's numeric answers to objects of the sound player's volume-classes (sub-classes of the Java class `guinea.player.Volume`). Decibel, linear, or percent scale can be used. The component is used by the MCL level setting control (Sec. 6.3.6).

LineScale

The `LineScale` grading component is similar to the GradeBar component except that a custom `Slider` component (similar to Swing's JSlider component) is used as the adjustable component instead of the scroll-bar. The custom slider allows placing ticks and labels in any position along the grading line so that tick and label positions match

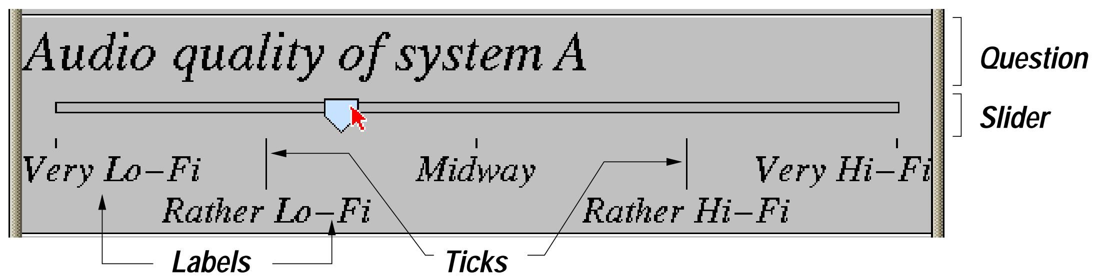

Figure 6.4: An example of a line scale grading component. A custom slider component (similar to Swing's JSlider component) is used to implement the adjustement component. Below the slider, any number of ticks with optional labels can be placed.

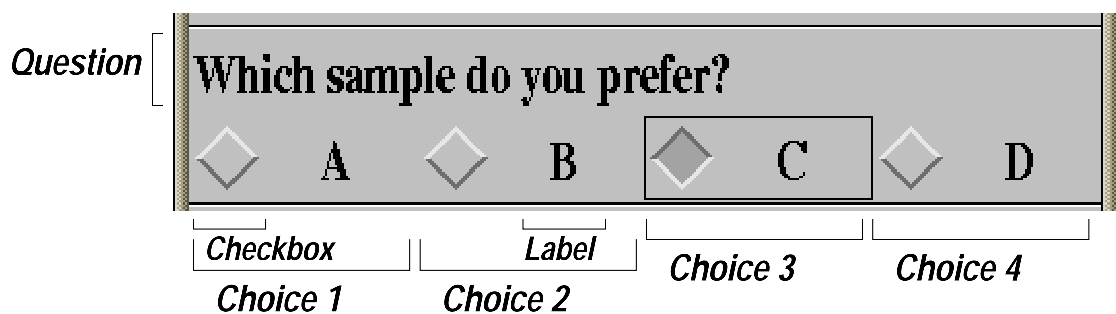

Figure 6.5: A multiple-choice question with a mutually exclusive check-box for each four choices. Each check-box has a string label.

exactly the position of the slider's knob for a particular grade. In a scroll-bar, the actual coordinate that matches a grade can not be reliably extracted.

The slider can also be configured either to send events always when the subject is moving the slider's knob or to send an event only when the subject releases the mouse button after the adjustment is done. With a GradeBar, it was not possible to send an event only when the adjustment has been finished, an event was sent always when the slider was being adjusted.

The current grade and adjectives part on the grading component is not shown by default, but it can be shown if so desired.

6.3.2 Multiple-choice

For making multiple-choice questions, the `CheckBoxChoice` component is provided. It presents a set of choices to the subject who selects one of the choices as the answer using mutually exclusive check-boxes.

Any number of choices can be added. When a check-box is selected, the panel sends

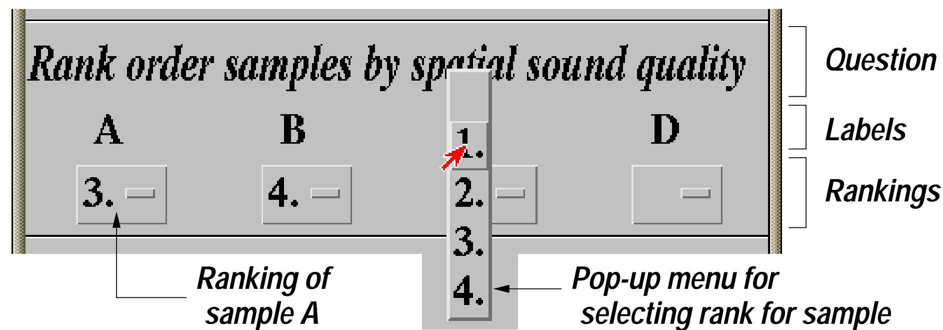

Figure 6.6: A rank order component for ranking multiple samples. A ranking is given for each sample using a pop-up menu.

an answer event to the test system with the choice as an argument. It is also possible to configure the choices to show a different label on the panel than the actual choice that is sent to the test system. If no labels are specified, the choices are used to construct the labels for the check-boxes. Currently only strings can be easily used as choices due to the simple text based configuration system.

This type of multiple-choice question could be implemented also in other ways. For example, a menu or list component could be used. It could also be beneficial to optionally allow selecting several choices (instead of only one, as currently). Also, it would be practical if it would be possible to use more than one row of check-boxes. The current component puts all choices on one row.

6.3.3 Rank order

The `RankOrder` question component is used to rank a set of labels (that usually correspond to samples) into an order according to some criteria [LH98]. Any number of labels can be added. Each label is given a rank using a pop-up menu. The component can be configured to allow or disallow ties and whether to allow incomplete ranking (not all labels have been ranked) as an answer. A special formatter is provided for results processing (Sec. 7) to customize the printout format of the rank-order answer.

6.3.4 Sample-play

The `PlayPanel` (Fig. 6.7) control component is used by the subjects to play samples during a free switching test (Sec. 5.3). It is also used in both free switching and fixed sequence tests as a monitor to indicate which of the samples is currently playing by highlighting the corresponding button.

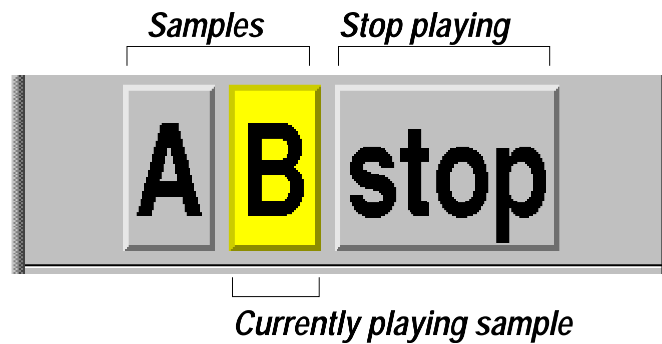

Figure 6.7: A `PlayPanel` controller / monitor component. The subject plays samples by pressing the corresponding buttons. A button to stop playback is also added automatically. The panel is also used as a monitor to show which of the samples is playing at the moment (the button corresponding to the playing sample is highlighted).

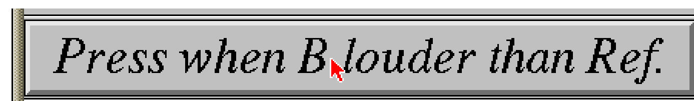

Figure 6.8: A simple button with a label. An event is sent when the button is pressed.

Any number of buttons with freely definable labels can be added. When a button is pressed, the panel sends a *control event* to the test engine with the button's choice as an argument. It is also possible to configure the buttons to show a different label on the panel than the button's choice that is sent to the test system. If no labels are specified, the choices are used to construct the labels for the buttons. Currently only strings can be easily used as choices due to the simple text based configuration system.

In future, more standard "tape playing" buttons could be added (stop, pause, rewind, etc.) with the familiar figures on them. Also, it could be nice to be able to configure the placement of the buttons relative to the sample buttons and which of them to be shown.

6.3.5 Button

The `Button` is a simple button control. It sends an event when the button is pressed. The label on the button is user-definable. The button is used in TAFC-test.

6.3.6 MCL level controller

This special component of the subject's panel allows the subject to set a comfortable listening level for a test session. Figure 6.9 shows the level setting panel. It is invoked

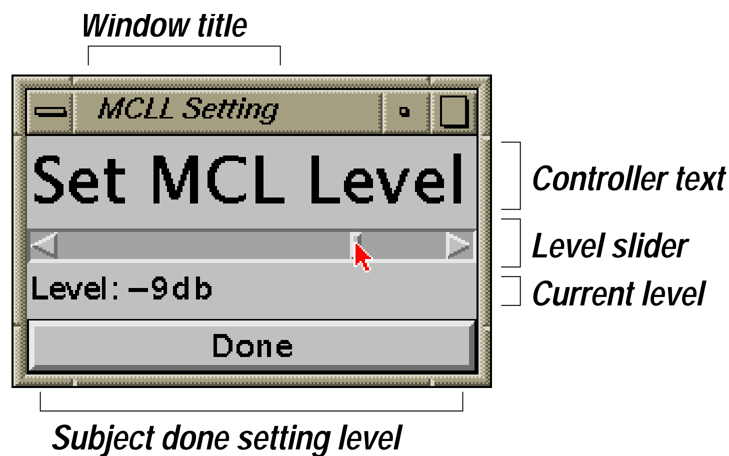

Figure 6.9: A panel for setting the most comfortable listening level (MCLL). The subjects uses the slider to set comfortable level within a range specified by the tester. The subject pressed the 'Done'-button when suitable level is found.

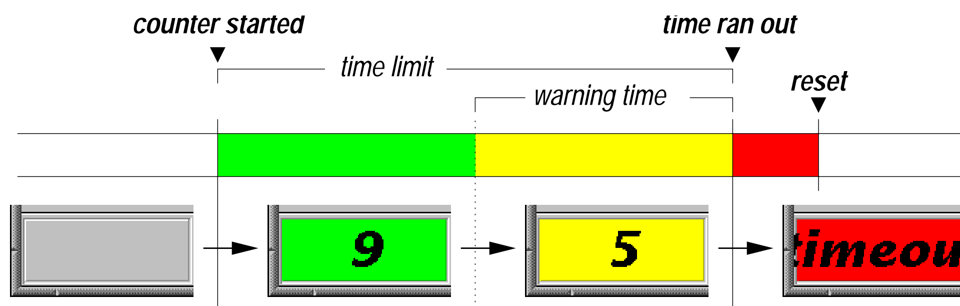

Figure 6.10: Time limit indicator shows how much time there is left for answering the questions about the current test item. When time limit is active, a count-down is shown. The color of the indicator is also used to show the status of the time limit.

at the start of a test session by the test engine if the level is to be set by the subject.

The subjects uses a slider to select comfortable level within a range specified by the tester. When the level is adjusted, an event containing the current selected level is sent to the test engine. After a suitable level is found, the subject presses the 'Done'-button.

6.3.7 Time-out indicator

The *time-out indicator* component (shown on lower left corner in Fig.6.1d) is used to implement the function of the *answering time limit interface* in Sec. 6.1.3. The indicator shows much time is remaining for answering the questions in the current item.

Figure 6.10 shows the function of the time-out indicator in more detail. When the

indicator is not active, it will be shown with the default background color. When the timeout indicator is activated, its color changes to green and a running counter is shown that shows how many seconds remains for answering. If an optional *warning time* is used, the indicator color changes to yellow when there are less than the warning time remaining before timeout.

If the timeout occurs, the indicator color changes to red and '*timeout*' is displayed in the place of the time counter. Then the test system is signalled about the timeout via the special 'Done'-button component. If in other hand, the subject presses the 'Done'-button on the subject UI before the timeout, the indicator is automatically reset. The interaction between the time-out indicator and the 'Done'-button is described in more detail below in Sec. 6.3.8.

6.3.8 Done-button

The 'Done'-button (see Fig. 6.1f) is a special button that is automatically added to the subject's panel. It is used by the subject to signal the test engine that he/she has completed grading the current test item. An event is sent when the subject presses the button.

The 'Done'-button is usually disabled at the start of each item and it is enabled only when all the questions have been answered. The test engine handles enabling and disabling the button.

When the answering time limit is used, the time-out indicator (Sec. 6.3.7) interacts with the 'Done'-button. If the subject finishes answering the questions in time, pressing the button also resets the time-out indicator to wait for next item. If a timeout occurs, the indicator signals the button which then signals the test engine by sending an event. In either case, a flag in the event tells whether the event was caused by the subject pressing the button or a time-out has occured.

6.3.9 Test status monitor

An optional *test status monitor* component is used to inform the subject about the progress of the test session. It is usually used to show how many test items have been done already and the total number of test items in the test session (see Fig. 6.1e).

The test engine uses special object that holds the index of the current test item and the total number of test items. The monitor component formats the information as a string. It would be possible to replace the monitor with a different kind of progress

display, for example, a progress bar or a percentage, as long as it understands the progress information object. The monitor can also display ordinary strings, allowing short one-line messages.

6.4 Remote terminals

The GP system allows testing several subjects concurrently. Remote terminals can be used as subject UIs in addition to local terminals (the workstation's console). A local terminal is running in the same process as the test system. A remote terminal runs on a different process than the test system on usually a different host. Remote subject terminals (clients) and the test process (server) communicate using sockets over a LAN (TCP/IP).

As subject UIs are written in Java, any Java-capable terminal (workstation, PC, laptop, etc) can be used. When testing the experimenter first starts a remote terminal server and waits for connections to it. Client terminals are started by running a client program that connects to the server. The server sends the subject UI's code to the client and the client starts it. In both communication end points, wrappers are created that emulate the subject UIs as local terminals, hiding the network between them.

All UI components must be serializable.

6.5 Tools

GP includes two tools to help creating configurations for subject panels: the *font tester* and the *UI tester.*

6.5.1 Font tester

The font tester (`gpFontTester`) is a utility that helps selecting fonts for the UI panel. It has three pop-up menus for selecting the font name, the font style, and the font size (Fig. 6.11). An editable example text is shown the result using the selected font. The corresponding Java font name of the selected font can be copied from a text field and pasted to a text editor.

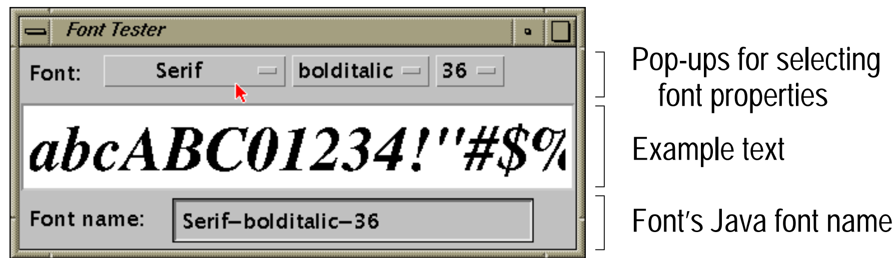

Figure 6.11: Font tester utility for selecting fonts to use on the subject's UI panel. Pop-up menus are used to select font properties and an editable text example shows the example string in selected font.

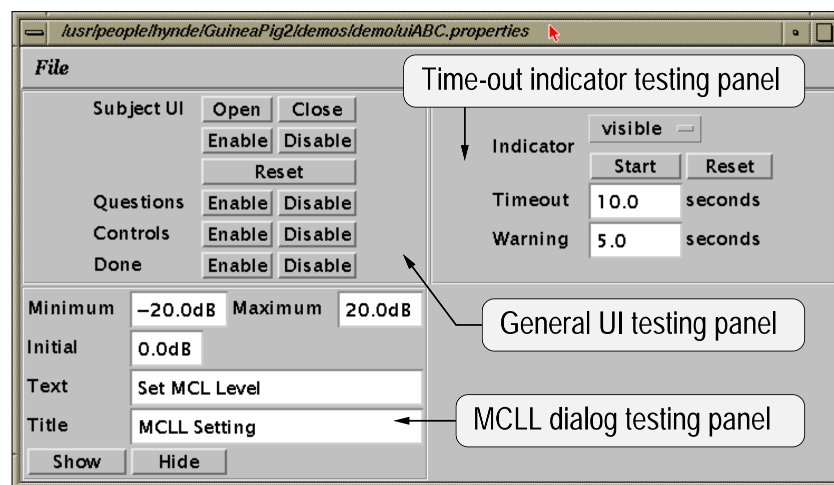

Figure 6.12: UI tester utility for testing UI configuration files and custom components.

6.5.2 UI panel tester

The UI tester (`gpUITester`) is a utility for testing subject panel configurations. It is also useful for testing new custom components developed by yourself to make sure they work correctly.

The tester shows the subject's UI panel and a testing panel (Fig. 6.12) where many features of the panel can be tested. First sub-panel tests general subject UI interface features: opening and closing the panel, and disabling, enabling, and resetting components. Second sub-panel tests the time limit indicator (if the panel implements it). Third sub-panel tests the MCLL setting panel of the UI (if the UI panel supports it).

When components are operated on the UI panel, the events they generate are printed on the shell. The type of the event (control or answer event) is displayed, the name of the source component (a control ID or a question ID), and the control event's arguments or the answer event's answer.

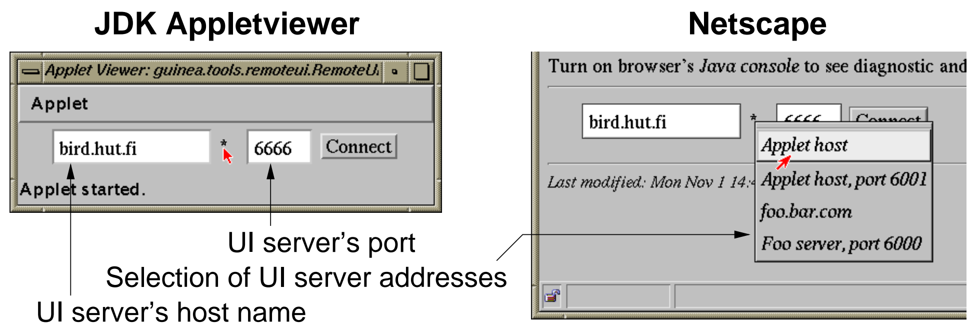

Figure 6.13: Java applet for launching a subject UI panel on remote terminal.

6.5.3 Remote UI server tester

The remote UI server tester (`gpUIServerTester`) is used to test whether remote UI panels work. The tester starts a remote subject UI server that waits for connections from remote terminals. When a connection is made, a subject UI panel is sent to the client and a UI panel tester (Sec. 6.5.2) is launched to test it.

6.5.4 Remote UI panel client as an application

The remote panel client (`gpRemoteUIClient`) is a small application that is run on the remote terminal. It connects to the remote UI server in the test server, downloads the UI panel from there, and starts it. A copy of the GP Java class files is needed available locally on the host that is used as the remote terminal.

6.5.5 Remote UI panel client as a Java applet

The remote UI client (see above) can also be invoked as a Java applet. Only a Java-capable web-browser (Fig. 6.13) is needed on the remote terminal. When a connection is made, the browser opens a new frame (window) for the UI panel (the UI is not displayed on the applet web page).

However, using an applet requires that there is a web-server running on the host where the remote UI server is run (it is usually the host that runs the test). The remote terminal needs to load the applet page and GP's Java class files from the web server to work. GP class files need to be copied to a directory where the applet can load them. GP includes an example applet page that can be used with minor modifications.

Chapter 7

Test results processing

The GP system does not perform any kind of analysis of the test data. The role of the GP system is to gather the data and export it for analysis by other statistical analysis packages (e.g. SAS, SPSS, Excel).

Each test session produces a *session log*[1] or *results*-file. The file contains copies of the test items presented during the session with the answers given by the subject(s) to the questions.

The log files are stored in the *Java's Serialization* [Sun98] format which is not generally readable by any other analysis tools. A conversion tool is used to convert the log files into a more readable tab-delimited ASCII text format (Sec. 7.1). The format of the output is configured with configuration files and command line options (Sec. 7.3). The simple text format is also easy to process further with many UNIX's common text processing tools, such as Perl or AWK.

7.1 Format of exported results file

The results are output as tabulated ASCII text, one line per test item. Different types of information from the items are printed as fields (columns) separated by ASCII TAB-character. The order and the type of the fields is specified with output configuration options (Sec. 7.3).

Comment lines may be included in the results file. Comment lines always start with the ASCII hash (`#`) character. Also, empty lines are considered comments.

Figure 7.1 shows an example of the results output.

[1] A serialized `guinea.logic.SessionLog` object.

```
# Session ID: Se1
#  Start time: Wed Jan 26 16:04:35 GMT+02:00 2000
#  End time:   Wed Jan 26 16:05:09 GMT+02:00 2000
#  MCLL: -11.0dB
#  Session files
#   Test Directory:   /usr/people/hynde/GuineaPig2/demos/demo
#   Test config file: /usr/people/hynde/GuineaPig2/demos/demo/./testABC.properties
#   Items file:       /usr/people/hynde/GuineaPig2/demos/demo/itemsABRef.properties
#   Sample-list file: /usr/people/hynde/GuineaPig2/demos/demo/samples.properties
#   UI config file:   /usr/people/hynde/GuineaPig2/demos/demo/uiABC.properties
#   Playlist file:    /usr/people/hynde/GuineaPig2/demos/demo/playlist
#
#ItemID SubjID  SesID   Time/s  Switch  A       B       Ref     gB      gA
item1   Se1a    Se1     9.5     5       pirr44  pirr32  pirr44  3.6     3.1
item3   Se1a    Se1     8.7     7       pirr8   pirr11  pirr8   7.8     7.1
item4   Se1a    Se1     6.4     3       pirr11  pirr16  pirr16  3.0     2.9
item2   Se1a    Se1     ABORTED 0       pirr22  pirr32  pirr32
```

Figure 7.1: Example of the output format of the results. Fields *A*, *B*, and *Ref* are item parameters. Fields *gA*, and *gB* are grades given for samples *A* and *B* against sample *Ref*.

7.2 Exported information

Information that can be exported for each test item includes:

- The **item ID** of the test item.
- The **session ID** of the test session.
- The **subject ID** of the subject to whom the item was presented to and who gave the answers.
- **Item start time** when this item was presented (both time and date is stored).
- **Item duration**, how much time the subject used to grade this test item.
- **Number of sample switches** the subject made between samples during this item. Actually this saves the how many times samples were played.
- The **item parameters** of this test item, such as the sample IDs of the parameters A and B (for example).
- The **Answers** to the questions shown to the subject in the subject's window.

In addition some per-session information can be included:

- **Starting time** of the session (both time and date are stored).

- **Ending time** of the session (both time and date are stored).
- Session's **MCL level** (Sec. 5.5) used for that session.

Also, the names of the test configuration files that were used in the session are printed (as comments). See example in Fig. 7.1.

7.3 Results output configuration

The results output format can be customized with configuration files and/or command line options. Customization options are:

- **Select fields** to print. By default, all possible fields are not printed in the output.
- **Order** of the fields.
- **Formatting** of the fields. For example, the format used to print a number (number of decimals, date format, etc.) is customizable. GP includes built-in formatters for numbers, dates, volume levels, and rank-order (Sec. 6.3.3). Custom plug-in formatters can be easily added by extending Java's text formatting classes[2]. A custom formatter is usually needed if special answer types are used with more complex custom question components.
- Simple **filtering** based on item, subject or session IDs. For example to only print the answers given to some specific test items.
- **Sub-fields** allow splitting a single parameter or answer field into multiple fields (columns). For example, the item's starting time and date information, which is actually a single Java date object, could be splitted into separate time and date fields.

[2] Any class that extends Java's `java.text.Format` can be used as a field formatter in the results conversion tool. For example, GP uses Java's text formatting classes `java.text.DecimalFormat` and `java.text.SimpleDateFormat` to format numbers and dates.

Chapter 8

Discussion

In this section the GuineaPig system is discussed as it is at present and as described in the thesis. The pros and cons of the GuineaPig system and related topics are explored, and thoughts for possible future developments are presented.

8.1 Java

Java is promoted as a *portable, interpreted, high-performance, simple, object-oriented* programming language platform [GM96]. It is a new language with many good features not included in many other languages as standard features, such as threads, a standard abstract window toolkit (AWT), and binary platform independence.

In practice however, at the time GP was being developed (in 1997-1999), the standard Java JDK 1.1 package was rather limited in many ways. For example, support for audio was very crude, and the AWT has only a limited selection of basic UI components. The development is said to be faster because Java is interpreted but in reality Java is a compiled language: programs are compiled into a platform-independent byte-code that is interpreted by the Java virtual machine (JVM) when the program is run. Also, Java does not offer an interactive console (a command line interpreter interface) usually associated with interpreted languages (such as Python or Lisp). An interactive console would be very helpful when testing and debugging large experimental systems like the GuineaPig.

The author's opinion is that the Java platform doesn't live up to the "hype" and is not the ideal platform large experimental systems[1].

[1]Many large projects that have been tried to use Java have been abandoned, such as Java's HotJava browser, Netscape's Javazilla web-browser (Netscape Navigator ported to Java), or a Java Office suite

8.2 SGI

The SGI platform is an excellent base for professional audio-visual applications. It has fine support for audio and video and has uniform programming interfaces (APIs) for controlling them across all SGI platforms. Also, precise synchronization of multiple audio and video devices is easy through the programming API.

The couple of recent years have been difficult for SGI. Interest for smaller SGI workstations has been declined as the much cheaper and faster Intel PCs have overtaken SGI workstations in processing power. However, the audio and video support of SGI is still unmatched.

Recently, SGI has become a supporter of Linux and has made many features found in SGI IRIX available also for Linux as open source. Among them are also the SGI digital media libraries[2] which are also used in GuineaPig. Therefore, Linux and IRIX would be the natural platforms for GuineaPig in future.

SGI's Java 1.1 kit worked quite well but SGI was slow to release a current version of JDK, the SGI's Java 1.2 or 2.0 version was still "in beta".

8.3 Sound player

The sound player is basically a simple mixing program for playback of pre-recorded or generated samples. Support for video and real-time filtering as envisioned initially had to be eliminated.

The player can read very large samples (much larger than system memory) without difficulty and supports many common file formats. The large sample support was initially aimed for use together with video. These features were very easy to implement, thanks to SGI's excellent audio libraries. However, playing samples directly from disk can be problematic specially if many large samples are played at the same time. An option to load whole samples into memory should be added. Multiple shorter samples could then be easily played, possibly with a larger sample being played directly from disk.

The player allows "wide" output transparently by combining multiple audio devices so that they look like a single player device. The outputs of the audio devices are au-

by Corel, or even Sun's own HotJava Browser. However, Java might just be better suited for embedded applications.

[2] SGI has just recently released its Digital Media Development Kit (dmSDK) and SGI Audio File library as open source for Linux and IRIX.

tomatically synchronized. Also, the output channels can be partitioned into smaller sections, each of which act as a different sound player.

8.4 Test engine

One weak point of the GP test engine is the rather limited support for dynamic or adaptive tests. Better support for them would certainly be welcome. At present, only the TAFC-test is included. The system can be extended for more adaptive cases, but it will require more or less programming effort. One could start by looking at the implementation of the TAFC-test in GP and go on from there.

The author would have liked to modularize the test engine more like the user interface components (discussed below). That way more variations for possible tests could be added by combining, adding or changing some smaller modules only, instead of subclassing an existing test class as now is required. For example, video support might be more easily added.

8.5 User interfaces

The subject user interfaces do not actually require a graphical toolkit such as Java's AWT or Swing. More abstract interfaces are used that can be implemented in many different ways.

For example, a box containing electrical buttons and potentiometers could be used to implement a multiple-choice question and various grading sliders. The box could be connected to the workstation running the GP system with a serial cable and customized UI components would be used instead of graphical components. One could in principle construct a component that takes an EEG- or MEG-response value as an answer that is then recorded.

Among the set of UI components in GP2 system, many commonly used types are included, such as multiple-choice, various grading scales (including standardized ones), and a rank-order. These can be easily customized and new types can be added without need to change the rest of the system.

The user interfaces in GP are graphical and use Java's AWT toolkit. The AWT toolkit is rather limited and lacks some common components, such as a slider. In GP, usually a scroll-bar is used to implement a grading scale. Later, a custom component (Sec. 6.3.1) was added as a better grading slider with more options.

An addition that could be welcome is a more familiar “play”, “stop”, “pause”, etc. symbolic buttons for playing samples. Also, a progress meter showing the position of current sample and to jump to a position, would be helpful when judging long samples.

The remote user interfaces are useful as they enable testing many people at the same time. In GP, the remote interface is implemented using a custom method, instead of Java’s *Remote Method Invocation* (RMI) system [Sun99]. The reason was that at the time when remote interfaces were added, the components were already implemented based on AWT components that can not be operated using RMI. In future it would be better to use the RMI as it is already built in the Java framework. It would require some changes in the way GP now uses to build the user interface, but as the subject UI interfaces are not tied to graphical modules, the change can certainly be made without too much trouble.

8.6 Result processing

GP2 does not include result analysis as there are already many sophisticated tools for such purposes, and it would have been wasteful to duplicate such functions. Instead, GP2 exports the data produced by the test in a simple tabulated text format that can easily be imported and analyzed by statistical tools. Tabular information is also easy to process by many Unix- or Windows-tools, such as Perl, AWK, Matlab, and Excel.

In future, it could be useful to include a SQL database-interface for storing test results for reading the test information from a database (test items, sample properties, etc.). For that, Java’s standard database access API, *Java Database Connectivity* (JDBC), could be used.

Chapter 9

Conclusions

In this thesis, the GuineaPig2 system is developed and described as a flexible generic platform for subjective audio testing. The system provides wide range of subjective audio tests, including standardized tests. It also eliminates a lot of the complexity of setting up such experiments. As the system is software-based, little additional hardware is required to perform tests.

The GuineaPig2 system is scalable and can be easily customized. Modular design allows creating new kinds of tests that are not covered by existing standardized tests.

In its current configuration, the system allows upto 32 channels of 24-bit digital audio output, supports the standard sample rates, and many common audio file formats. Multiple test subjects can participate in a test at the same time, reducing the time needed for testing.

Test data produced by each test session is logged to a simple tabulated text file. The file can then be imported to many generally available statistical tools for final analysis.

Bibliography

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java^{TM}Programming Language*. Addison-Wesley, Third edition, June 2000.

[Bec93] Soren Bech. Training of subjects for auditory experiments. *Acta Acustica*, 1:89–99, 1993.

[Bec94] S Bech. Perception of timbre of reproduced sound in small rooms: Influence of room and loudspeaker position. *Journal of the Audio Engineering Society*, 42(12):999–1007, 1994.

[Bec99] S Bech. Methods for subjective evaluation of spatial characteristics of sound. In *Proceedings of the AES 16th International Conference*. Audio Engineering Society, 1999.

[Bla83] Jens Blauert. *Spatial hearing*. MIT Press, 1983.

[BR99] J Berg and F Rumsey. Spatial attribute identification and scaling by repertory grid technique and other methods. In *Proceedings of the AES 16th International Conference*. Audio Engineering Society, 1999.

[CC92] W G Cochran and G M Cox. *Experimental design*. Wiley, 1992.

[Cla82] D L Clark. High resolution subjective testing using a double blind comparator. *Journal of the Audio Engineering Society*, 30(5), 1982.

[Cla91] D L Clark. Ten years of a/b/x testing. In *Proceedings of the 91st Convention of the Audio Engineering Society, Preprint 3167*, 1991.

[Dav63] H A David. *The method of paired comparisons*. Oxford University press, 1st edition, 1963.

[Gab79] A Gabrielsson. Statistical treatment of data for listening tests on sound reproduction systems. Technical Report Rep. TA 92, Department of Technical Audiology, Karolinska Inst., Sweden, 1979.

[GJGB00] James Gosling, Bill Joy, James Gosling, and Gilad Bracha. *The Java^{TM}Language Specification*. Addison-Wesley, Second edition, June 2000.

[GM96] James Gosling and Henry McGilton. The Java Language Environment – A White Paper. <URL:`http://java.sun.com/docs/white/langenv/`>, 1996.

[HHRF96] Jussi Hynninen, Keijo Heljanko, and Jussi Rinta-Filppula. GuineaPig – Overview. <URL:`http://www.acoustics.hut.fi/projects/GuineaPig/`>, 1996.

[HZ99] Jussi Hynninen and Nick Zacharov. Guineapig – a generic subjective test system for multichannel audio. In *Proceedings of the* 106th *AES Convention, Preprint 4871*. Audio Engineering Society, 1999.

[IR97] ITU-R. *Recommendation BS.1116-1, Methods for the subjective assessment of small impairments in audio systems including multichannel sound systems*. International Telecommunications Union Radiocommunication Assembly, 1997.

[IR98a] ITU-R. *Recommendation BS.1284, Methods for the subjective assessment of sound quality - General requirements*. International Telecommunications Union Radiocommunication Assembly, 1998.

[IR98b] ITU-R. *Recommendation BT.500-8, Methodology for the subjective assessment of quality of television pictures*. International Telecommunications Union Radiocommunication Assembly, 1998.

[IT96a] ITU-T. *Recommendation P.800, Methods for subjective determination of transmission quality*. International Telecommunications Union Radiocommunication Assembly, 1996.

[IT96b] ITU-T. *Recommendation P.910, Subjective video quality assessment methods for multimedia applications*. International Telecommunications Union Radiocommunication Assembly, 1996.

[Kar99] Matti Karjalainen. Kommunikaatioakustiikka. Report 51, Laboratory of Acoustics and Audio Signal Processing, Helsinki University of Technology, 1999.

[Kra96] Douglas Kramer. *The Java™ Platform – A White Paper*. Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A., May 1996.

[Lev71] H Levitt. Transformed up-down methods in psychoacoustics. *Journal of the Acoustical Society of America*, 49(2, part 2):467–477, 1971.

[LH98] H T Lawless and H Heyman. *Sensory evaluation of food*. Chapman and Hall, 1998.

[MCC91] M Meilgaard, G V Civille, and B T Carr. *Sensory evaluation techniques*. CRC Press, 1991.

[Mil56] G A Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97, 1956.

[NB94] J C Nunnally and I H Bernstein. *Psychometric theory*. McGraw-Hill, 3rd edition, 1994.

[SS93] H. Stone and J. L. Sidel. *Sensory evaluation practices*. Academic Press, 2nd edition, 1993.

[Suna] Java™ Technology Home Page. <URL:`http://www.javasoft.com/`>.

[Sunb] The Java™ Language: An Overview. <URL:`http://java.sun.com/docs/overviews/java/java-overview-1.html`>.

[Sun98] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A. *Java™Object Serialization Specification*, November 1998. Revision 1.43, JDK™1.2.

[Sun99] Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303-4900 U.S.A. *Java™Remote Method Invocation Specification*, December 1999. Revision 1.7, Java™2 SDK, Standard Edition, v1.3.0.

[Too82] F E Toole. Listening tests-turning opinion into fact. *Journal of the Audio Engineering Society*, 30(6), 1982.

[Too85] F E Toole. Subjective measurements of loudspeaker sound quality and listener performance. *Journal of the Audio Engineering Society*, 33(1), 1985.

[Zac98] N Zacharov. Subjective appraisal of loudspeaker directivity for mulitchannel reproduction. *Journal of the Audio Engineering Society*, 46(4):288–303, 1998.

Appendix A

SGI Workstation Audio Features

This appendix lists SGI workstation audio features. The list is from the **SGI Audio & MIDI FAQ** found at: `http://www.sgi.com/tech/faq/audio/general.html`

O2

- 2 independent, 16-bit stereo analog line-level outputs (2 RCA jacks + stereo 3.5mm jack). One output also drives internal speaker/headphone jack
- 1 16-bit stereo analog input, selectable between line-level (2 RCA jacks), video camera microphone (included), or external microphone (not included)
- supports all sample rates from 4kHz to 48kHz with 1Hz resolution
- audio sample rates can be slaved to video inputs or outputs
- sample-accurate timing information for precise synchronization
- easy-access front-panel volume controls (in addition to apanel)
- I/O can be scaled up to many channels using an optional expansion card (see below)

Octane

- 8-channel, 24-bit ADAT Optical input and output (2 optical connectors)
- stereo 18-bit analog line-level input and output (4 RCA connectors)
- stereo AES3 24-bit digital input and output (2 RCA connectors)
- Optical connectors can be used for ADAT I/O or SPDIF I/O
- Nearly arbitrary sample rates from 4kHz to 48kHz
- All inputs and outputs are independent and can be used simultaneously either with independent or synchronized sample-rates

- A/D, D/A, and/or digital output sample rates can be slaved to ADAT Optical or AES input clock A/D, D/A, and/or digital output sample rates can be slaved to video
- sample-accurate timing information for precise synchronization
- I/O can be scaled up to many channels using an optional expansion card (see below)
- Speaker output, mono microphone input
- Bundled powered desktop speakers

Low-Cost O2, Octane, and Onyx2 Digital Audio Expansion Option

- 1/2-length PCI option card
- 8-channel, 24-bit ADAT Optical input and output (2 optical connectors)
- stereo AES3 24-bit digital input and output (2x75 ohm BNC connectors)
- video house sync input (1 BNC connector)
- Multiple cards can be added and synchronized for scalable I/O
- Optical connectors can be used for ADAT I/O or SPDIF I/O
- Nearly arbitrary sample rates from 4kHz to 48kHz
- All inputs and outputs are independent and can be used simultaneously either with independent or synchronized sample-rates
- output sample rates can be slaved to ADAT Optical or AES input clock
- output sample rates can be slaved to video
- sample-accurate timing information for precise synchronization

SGI Audio Library (AL 2.0)

- Provides a device-independent interface to low-latency, real-time audio I/O
- AL 1.0 bundled with IRIX 5.3, 6.2 Developers Option; AL 2.0 bundled with later releases
- Supports multiple simultaneous applications using audio hardware
- AL 2.0 supports multiple independent or synchronized audio devices
- Supports scalable hardware capabilities, in terms of sample rates, gains, numbers of channels, and audio word-size
- Supports precise synchronization to other media, including MIDI & video

SGI Audio File Library (All Platforms)

- Supports audio file I/O to many formats with a uniform API
- Latest version supports: AIFF-C, AIFF, NeXT/Sun SND/AU, WAVE (RIFF), Berkeley/IRCAM/CARL SoundFile, MPEG1 audio bitstream, Sound Designer II, Audio Visual Research, Amiga IFF/8SVX, SampleVision, VOC, SoundFont2, Raw (headerless)

Appendix B

Web links

GuineaPig

- `http://www.acoustics.hut.fi/projects/GuineaPig2/`
  GuineaPig (version 2) web page
- `http://www.acoustics.hut.fi/projects/GuineaPig/`
  GuineaPig (original version) web page

Laboratory of Acoustics and Audio Signal Processing, HUT

- `http://www.acoustics.hut.fi/`
  Laboratory's homepage
- `http://www.acoustics.hut.fi/~hynde/`
  Jussi Hynninen at the Acoustics Laboratory

SGI

- `http://www.sgi.com/`
  SGI Web site
- `http://www.sgi.com/o2/`
  The Silicon Graphics O2 workstation
- `http://www.sgi.com/octane/`
  The Silicon Graphics Octane workstation
- `http://www.sgi.com/software/irix6.5/`
  IRIX 6.5
- `http://www.sgi.com/tech/faq/audio/`
  SGI Audio & MIDI FAQ
- `http://www.sgi.com/developers/devtools/sdk/dmsdk.html`
  Digital Media Software Development Kit v2.0 for SGI IRIX and Linux
- `http://www.sgi.com/developers/devtools/languages/java.html`
  SGI's Java Environment
- `http://techpubs.sgi.com/library/tpl/cgi-bin/init.cgi`
  SGI TechPubs Library.

Java

- `http://www.javasoft.com/`
  Sun's Java homepage

Acoustics

- `http://www.sfu.ca/sonic-studio/index.html`
  Handbook for Acoustic Ecology (a good glossary of acoustics terms)